Condor Version 6.0.3 Manual

Condor Team, University of Wisconsin–Madison

February 8, 1999

Copyright and Disclaimer

Condor Usage Licenses

Following are licenses for use of Condor Version 6. Academic institutions should agree to the Academic Use License for Condor, while all others should agree to the Internal Use License for Condor.

Internal Use License for Condor

This is an Internal Use License for Condor Version 6. This License is to be signed by RECIPIENT (the "RECIPIENT"), and returned to the Condor Team at the Computer Sciences Department of the University of Wisconsin-Madison (the "PROVIDER"). The Condor Version 6 software program was developed by the Condor Team. All rights, title, and interest in Condor Version 6 are owned by the Condor Team. The subject computer program, including source code, executables, and documentation shall be referred to as the "SOFTWARE."

RECIPIENT and PROVIDER agree as follows:

1. Definitions.
   (a) The "Object Code" of the SOFTWARE means the SOFTWARE assembled or compiled in magnetic or electronic binary form on software media, which are readable and usable by machines, but not generally readable by humans without reverse assembly, reverse compiling, or reverse engineering.
   (b) The "Source Code" of the SOFTWARE means the SOFTWARE written in programming languages, such as C and FORTRAN, including all comments and procedural code, such as job control language statements, in a form intelligible to trained programmers and capable of being translated into Object Code for operation on computer equipment through assembly or compiling, and accompanied by documentation, including flow charts, schematics, statements of principles of operations, and architecture standards, describing the data flows, data structures, and control logic of the SOFTWARE in sufficient detail to enable a trained programmer through study of such documentation to maintain and/or modify the SOFTWARE without undue experimentation.
   (c) A "Derivative Work" means a work that is based on one or more preexisting works, such as a revision, enhancement, modification, translation, abridgment, condensation, expansion, or any other form in which such preexisting works may be recast, transformed, or adapted, and that, if prepared without authorization of the owner of the copyright in such preexisting work, would constitute a copyright infringement. For purposes hereof, a Derivative Work shall also include any compilation that incorporates such a preexisting work. Unless otherwise provided in this License, all references to the SOFTWARE include any Derivative Works provided by PROVIDER or authorized to be made by RECIPIENT hereunder.
   (d) "Support Materials" means documentation that describes the function and use of the SOFTWARE in sufficient detail to permit use of the SOFTWARE.
2. Copying of SOFTWARE and Support Materials. PROVIDER grants RECIPIENT a non-exclusive, non-transferable use license to copy and distribute internally the SOFTWARE and related Support Materials in support of RECIPIENT's use of the SOFTWARE. RECIPIENT agrees to include all copyright, trademark, and other proprietary notices of PROVIDER in each copy of the SOFTWARE as they appear in the version provided to RECIPIENT by PROVIDER. RECIPIENT agrees to maintain records of the number of copies of the SOFTWARE that RECIPIENT makes, uses, or possesses.
3. Use of Object Code. PROVIDER grants RECIPIENT a royalty-free, non-exclusive, non-transferable use license in and to the SOFTWARE, in Object Code form only, to:
   (a) Install the SOFTWARE at RECIPIENT's offices listed below;
   (b) Use and execute the SOFTWARE for research or other internal purposes only;
   (c) In support of RECIPIENT's authorized use of the SOFTWARE, physically transfer the SOFTWARE from one (1) computer to another; store the SOFTWARE's machine-readable instructions or data on a temporary basis in main memory, extended memory, or expanded memory of such computer system as necessary for such use; and transmit such instructions or data through computers and associated devices.
4. Delivery. PROVIDER will deliver to RECIPIENT one (1) executable copy of the SOFTWARE in Object Code form, one (1) full set of the SOFTWARE and related documentation in Source Code form, and one (1) set of Support Materials relating to the SOFTWARE within fifteen (15) business days after the receipt of the signed License.

5. Back-up Copies. RECIPIENT may make up to two (2) copies of the SOFTWARE in Object Code form for nonproductive backup purposes only.
6. Term of License. The term of this License shall be one (1) year from the date of this License. However, PROVIDER may terminate RECIPIENT's License without cause at any time. All copies of the SOFTWARE, or Derivative Works thereof, shall be destroyed by the RECIPIENT upon termination of this License.
7. Proprietary Protection. PROVIDER shall have sole and exclusive ownership of all right, title, and interest in and to the SOFTWARE and Support Materials, all copies thereof, and all modifications and enhancements thereto (including ownership of all copyrights and other intellectual property rights pertaining thereto). Any modifications or Derivative Works based on the SOFTWARE shall be considered a part of the SOFTWARE and ownership thereof shall be retained by the PROVIDER and shall be made available to the PROVIDER upon request. This License does not provide RECIPIENT with title or ownership of the SOFTWARE, but only a right of internal use.
8. Limitations on Use, Etc. RECIPIENT may not use, copy, modify, or distribute the SOFTWARE (electronically or otherwise) or any copy, adaptation, transcription, or merged portion thereof, except as expressly authorized in this License. RECIPIENT's license may not be transferred, leased, assigned, or sublicensed without PROVIDER's prior express authorization.
9. Data. RECIPIENT acknowledges that data conversion is subject to the likelihood of human and machine errors, omissions, delays, and losses, including inadvertent loss of data or damage to media, that may give rise to loss or damage. PROVIDER shall not be liable for any such errors, omissions, delays, or losses, whatsoever. RECIPIENT is also responsible for complying with all local, state, and federal laws pertaining to the use and disclosure of any data.
10. Warranty Disclaimer. RECIPIENT ACKNOWLEDGES AND AGREES THAT: (A) NEITHER THE CONDOR TEAM NOR THE BOARD OF REGENTS OF THE UNIVERSITY OF WISCONSIN SYSTEM (REGENTS) MAKE ANY REPRESENTATIONS OR WARRANTIES WHATSOEVER ABOUT THE SUITABILITY OF THE SOFTWARE FOR ANY PURPOSE; (B) THE SOFTWARE IS PROVIDED ON AN "AS IS, WITH ALL DEFECTS" BASIS WITHOUT EXPRESS OR IMPLIED WARRANTIES, INCLUDING WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE OR NONINFRINGEMENT; (C) NEITHER THE CONDOR TEAM NOR THE REGENTS SHALL BE LIABLE FOR ANY DAMAGE OR LOSS OF ANY KIND ARISING OUT OF OR RESULTING FROM RECIPIENT'S POSSESSION OR USE OF THE SOFTWARE (INCLUDING DATA LOSS OR CORRUPTION), REGARDLESS OF WHETHER SUCH LIABILITY IS BASED IN TORT, CONTRACT, OR OTHERWISE; AND (D) NEITHER THE CONDOR TEAM NOR THE REGENTS HAVE AN OBLIGATION TO PROVIDE DEBUGGING, MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.
11. Damages Disclaimer. RECIPIENT ACKNOWLEDGES AND AGREES THAT IN NO EVENT WILL THE CONDOR TEAM OR THE REGENTS BE LIABLE TO RECIPIENT FOR ANY SPECIAL, CONSEQUENTIAL, INDIRECT OR SIMILAR DAMAGES, INCLUDING ANY LOST PROFITS OR LOST DATA ARISING OUT OF THE USE OR INABILITY TO USE THE SOFTWARE EVEN IF THE CONDOR TEAM OR THE REGENTS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.
12. Compliance with Applicable Laws. RECIPIENT agrees to abide by copyright law and all other applicable laws of the United States including, but not limited to, export control laws.
13. U.S. Government Rights Restrictions. Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical

Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

14. Governing Law. This License shall be governed by and construed and enforced in accordance with the laws of the State of Wisconsin as it applies to a contract made and performed in such state.
15. Modifications and Waivers. This License may not be modified except by a writing signed by authorized representatives of both parties. A waiver by either party of its rights hereunder shall not be binding unless contained in a writing signed by an authorized representative of the party waiving its rights. The nonenforcement or waiver of any provision on one (1) occasion shall not constitute a waiver of such provision on any other occasions unless expressly so agreed in writing. It is agreed that no use of trade or other regular practice or method of dealing between the parties hereto shall be used to modify, interpret, supplement, or alter in any manner the terms of this License.

Academic Use License for Condor

This is an Academic Object Code Use License for Condor. This license is between you (the "RECIPIENT"), and the Condor Team at the Computer Sciences Department of the University of Wisconsin-Madison (the "PROVIDER"). The Condor software program was developed by the Condor Team. All rights, title, and interest in Condor are owned by the Condor Team. The subject computer program, including executables and supporting documentation, shall be referred to as the "SOFTWARE".

RECIPIENT and PROVIDER agree as follows:

1. A non-exclusive, non-transferable academic use license is granted to the RECIPIENT to install and use the SOFTWARE on any appropriate computer systems located at the RECIPIENT's institution to which the RECIPIENT has authorized access. Use of the SOFTWARE is restricted to the RECIPIENT and collaborators at RECIPIENT's institution who have agreed to accept the terms of this license.
2. The PROVIDER shall retain ownership of all materials (including magnetic tape, unless provided by the RECIPIENT) and SOFTWARE delivered to the RECIPIENT. Any modifications or derivative works based on the SOFTWARE shall be considered part of the SOFTWARE and ownership thereof shall be retained by the PROVIDER and shall be made available to the PROVIDER upon request.
3. The RECIPIENT may make a reasonable number of copies of the SOFTWARE for the purpose of backup and maintenance of the SOFTWARE, or for development of derivative works based on the SOFTWARE. The RECIPIENT agrees to include all copyright or trademark notices on any copies of the SOFTWARE or derivatives thereof. All copies of the SOFTWARE, or derivatives thereof, shall be destroyed by the RECIPIENT upon termination of this license.
4. The RECIPIENT shall use the SOFTWARE for research, educational, or other non-commercial purposes only. The RECIPIENT acknowledges that this license grants no rights whatsoever for commercial use of the SOFTWARE or in any commercial version(s) of the

SOFTWARE. The RECIPIENT is strictly prohibited from deciphering, disassembling, decompiling, translating, reverse engineering or otherwise deriving source code from the SOFTWARE, except to the extent applicable laws specifically prohibit such restriction.

5. The RECIPIENT shall not disclose in any form either the delivered SOFTWARE or any modifications or derivative works based on the SOFTWARE to any third party without prior express authorization from the PROVIDER.
6. If the RECIPIENT receives a request to furnish all or any portion of the SOFTWARE to any third party, RECIPIENT shall not fulfill such a request, and further agrees to refer the request to the PROVIDER.
7. The RECIPIENT agrees that the SOFTWARE is furnished on an "as is, with all defects" basis, without maintenance, debugging, support or improvement, and that neither the PROVIDER nor the Board of Regents of the University of Wisconsin System warrant the SOFTWARE or any of its results and are in no way liable for any use that the RECIPIENT makes of the SOFTWARE.
8. The RECIPIENT agrees that any reports, publications, or other disclosure of results obtained with the SOFTWARE will acknowledge its use by an appropriate citation. The appropriate reference for the SOFTWARE is "The Condor Software Program (Condor) was developed by the Condor Team at the Computer Sciences Department of the University of Wisconsin-Madison. All rights, title, and interest in Condor are owned by the Condor Team."
9. The term of this license shall not be limited in time. However, PROVIDER may terminate RECIPIENT's license without cause at any time.
10. Source code for the SOFTWARE is available upon request and at the sole discretion of the PROVIDER.
11. This license shall be construed and governed in accordance with the laws of the State of Wisconsin.

For more information:
Condor Team
Attention: Professor Miron Livny
7367 Computer Sciences
1210 W. Dayton St.
Madison, WI 53706-1685
miron@cs.wisc.edu
http://www.cs.wisc.edu/~miron/miron.html

CHAPTER

ONE

Overview

This chapter provides a basic, high-level overview of Condor, including Condor's major features and limitations. Because Condor is a system to implement a High-Throughput Computing environment, this section begins defining what is meant by High-Throughput Computing.

1.1 What is High-Throughput Computing (HTC) ?

For many research and engineering projects, the quality of the research or the product is heavily dependent upon the quantity of computing cycles available. It is not uncommon to find problems that require weeks or months of computation to solve. Scientists and engineers engaged in this sort of work need a computing environment that delivers large amounts of computational power over a long period of time. Such an environment is called a High-Throughput Computing (HTC) environment. In contrast, High Performance Computing (HPC) environments deliver a tremendous amount of compute power over a short period of time. HPC environments are often measured in terms of FLoating point Operations per Second (FLOPS). But a growing community is not concerned about FLOPS, as their problems are on a much larger scale. These people are concerned with floating point operations per month or per year. They are more interested in how many jobs they can complete over a long period of time instead of how fast an individual job can complete.

1.2 HTC and Distributed Ownership

The key to HTC is the efficiently harness the use of all available resources. Years ago, the engineering and scientific community relied on large centralized mainframe and/or big-iron supercomputers to do computational work. A large number of individuals and groups would have to pool their financial resources to afford such a machine. Users would then have to wait for their turn on the mainframe, and would only have a certain amount of time allotted to them. While this environment was inconvenient for the users, it was very efficient, since the mainframe was busy nearly all the time.

As computers became smaller, faster, and cheaper, users started to move away from centralized mainframes and started purchasing personal desktop workstations and PCs. An individual or small group could afford a computing resource that was available whenever they wanted it. It was usually far slower than the large centralized machine, but since they had exclusive access, it was worth it. Now, instead of one giant computer for a large institution, there might be hundreds or thousands of personal computers. This is an environment of distributed ownership, where individuals throughout an organization own their own resources. The total computational power of the institution as a whole might rise dramatically as the result of such a change, but because of distributed ownership individuals could not capitalize on the institutional growth of computing power. And while distributed ownership is more convenient for the users, it is also much less efficient. Many personal desktop machines sit idle for very long periods of time while their owners are busy doing other things (such as at lunch, in meetings, or at home sleeping).

1.3 What is Condor ?

Condor is a software system that creates a High Throughput Computing (HTC) environment by effectively harnessing the power of a cluster of UNIX workstations on a network. Although Condor can manage a dedicated cluster of workstations, a key appeal of Condor is its ability to effectively harness non-dedicated, preexisting resources in a distributed ownership setting such as machines sitting on people's desks in offices and labs.

1.3.1 A Hunter of Available Workstations

Instead of running a CPU-intensive job in the background on their own workstation, users submit their job to Condor. Condor will then find an available machine on the network and begin running the job on that machine. When Condor detects that a machine running a Condor job would no longer be available (perhaps because the owner of the machine came back from lunch and started typing on the keyboard), Condor checkpoints the job and then migrates it over the network to a different machine which would otherwise be idle. Condor then restarts the job on the new machine from precisely where it left off. If no machine on the network is currently available, then the job is stored in a queue on disk until a machine becomes available.

So, for example, say you submit a compute job that typically takes 5 hours to run. Condor may start running it on Machine A, but after 3 hours Condor notices activity on the keyboard. So Condor checkpoints your job and migrates it to Machine B. After two hours on Machine B, your job completes (and Condor notifies you via email).

Perhaps you have to run this 5 hour compute job 250 different times (perhaps on 250 different data sets). In this case, Condor can be a real time saver. With one command you can submit all 250 runs into Condor. Depending upon the number of machines in your organization's Condor pool, there could be dozens or even hundreds of otherwise idle machines (especially at night, for example) at any given moment running your job.

Condor makes it easy to maximize the number of machines which can run your job. Because Condor does not require participating machines to share file systems (via NFS or AFS for example), machines across the entire enterprise can run your job, including machines in different administrative domains. Condor does not even require you to have an account (login) on machines where it runs your job. Condor can do this because of its *Remote System Call* technology, which traps operating system calls for such operations as reading/writing from disk files and sends them back over the network to be performed on the machine where the job was submitted.

1.3.2 Effective Resource Management

In addition to migrating jobs to available machines, Condor provides sophisticated and distributed resource management. Match-making resource owners with resource consumers is the cornerstone of a successful HTC environment. Unlike many other compute cluster resource management systems which attach properties to the job queues themselves (resulting in user confusion over which queue to use as well as administrative hassle in constantly adding and editing queue properties to satisfy user demands), Condor implements a clean design called *ClassAds*.

ClassAds work in a fashion similar to the newspaper classified advertising want-ads. All machines in the Condor pool advertise their resource properties, such as available RAM memory, CPU type and speed, virtual memory size, physical location, current load average, and many other static and dynamic properties, into a *Resource Offer* ad. Likewise, when submitting a job, users can specify a Resource Request ad which defines both the required and desired set of resources to run the job. Similarly, a Resource Offer ad can define requirements and preferences. Condor then acts as a broker by matching and ranking Resource Offer ads with Resource Request ads, making certain that all requirements in both ads are satisfied. During this match-making process, Condor also takes several layers of priority values into consideration: the priority the user assigned to the Resource Request ad, the priority of the user which submitted the ad, and desire of machines in the pool to accept certain types of ads over others.

1.4 Distinguishing Features

Checkpoint and Migration. Users of Condor may be assured that their jobs will eventually complete even in an opportunistic computing environment. If a user submits a job to Condor which runs on somebody else's workstation, but the job is not finished when the workstation owner returns, the job can be checkpointed and restarted as soon as possible on another machine and will keep on executing in this manner until the job is completed. Condor's Periodic Checkpoint feature can periodically checkpoint the job even in lieu of migration in order to safeguard the accumulated computation time on job from being lost in the event of a system failure (such as the machine being shutdown, a crash, etc).

Remote System Calls. In Condor's Standard Universe execution mode, the local execution environment is preserved for remotely executing processes via Remote System Calls. Users do not have to worry about making data files available to remote workstations or even obtaining a login account on remote workstations before Condor executes their programs there. The program behaves under Condor as if it was running as the user whom submitted the job on the workstation where it was originally submitted, no matter on which machine it really ends up executing on.

No Changes Necessary to User's Source Code. No special programming is required to use Condor. Condor is able to run normal UNIX programs. The checkpoint and migration of programs by Condor is transparent and automatic, as is the use of Remote System Calls. These facilities are provided by Condor and, if these facilities are desired, only requires the user to re-link their program, not recompile it or change any code.

Sensitive to the desires of workstation owners. "Owners" of workstations have by default complete priority over their own machines. Workstation owners are generally happy to let somebody else compute on their machines while they are out, but they want their machines back promptly upon returning, and they don't want to have to take special action to regain control. Condor handles this automatically.

ClassAds. The ClassAd mechanism in Condor provides an extremely flexible and semantic-free, expressive framework for match-making Resource Requests and Resource Offers. One result is that users can easily request practically any resource, both in terms of what their job requires and/or what they desire for their job if available. For instance, a User can require their job run a machine with 64 megs of RAM, but state a preference for 128 megs if available. Likewise, machines could state for example in a Resource Offer ad that they prefer to run jobs from a certain set of users, and require that there be no interactive user activity detectable between 9 am and 5 pm before starting a job. Job requirements/preferences and resource availability constraints can be described in terms of powerful, arbitrary expressions, resulting in Condor being flexible enough to adapt to nearly any desired policy.

1.5 Current Limitations

Limitations on Jobs which can Checkpointed Although Condor can schedule and run any type of process, Condor does have some limitations on jobs that it can transparently checkpoint and migrate:

1. On some platforms, specifically HPUX and Digital Unix (OSF/1), shared libraries are not supported; therefore on these platforms applications must be statically linked (Note: shared library checkpoint support is available on IRIX, Solaris, and LINUX).
2. Only single process jobs are supported, i.e. the fork(2), exec(2), system(3) and similar calls are not implemented.
3. Signals and signal handlers are supported, but Condor reserves the SIGUSR2 and SIGTSTP signals and does not permit their use by user code.
4. Many interprocess communication (IPC) calls are not supported, i.e. the socket(2), send(2), recv(2), and similar calls are not implemented.
5. All file operations must be idempotent — read-only and write-only file accesses work correctly, but programs which both read and write to the same file may not.
6. Each Condor job that has been checkpointed has an associated *checkpoint file* which is approximately the size of the address space of the process. Disk space must be available to store the checkpoint file on the submitting machines (or on the optional Checkpoint Server module).

Note: these limitations only apply to jobs which Condor has been asked to transparently checkpoint. If job checkpointing is not desired, the limitations above do not apply.

Security Implications. Condor does a significant amount of work to prevent security hazards, but loopholes are known to exist. Condor can be instructed to run user programs only as user "nobody", a user login which traditionally has very restricted access. But even with access solely as user nobody, a sufficiently malicious individual could do such things as fill up /tmp (which is world writable) and/or gain read access to world readable files (which are the only files user nobody can access). Furthermore, where security of the machines in the pool is a high concern, only machines where the "root" user on that machine can be trusted should be admitted into the pool. (Note: Condor provides the administrator with IP-based security mechanisms to enforce this).

Jobs need to be relinked to get Checkpointing and Remote System Calls Although typically few to none source code changes are required, Condor requires that the jobs be relinked with the Condor libraries to offer checkpointing and remote system calls. This often precludes commercial software binaries from taking advantage of these services because commercial packages rarely make their object code available. However, one can certainly still submit and run commercial packages in Condor and still take advantage of Condor's other services.

Architecture	*Operating System*
Hewlett Packard PA-RISC (both PA7000 and PA8000 series)	HPUX 10.20
Sun SPARC Sun4m,c, Sun UltraSPARC	Solaris 2.5.x, 2.6
Silicon Graphics MIPS (R4400, R4600, R8000, R10000)	IRIX 6.2, 6.3, 6.4
Digital ALPHA	OSF/1 (Digital Unix) 4.x
Intel x86, Pentium series	Linux 2.x Solaris 2.5.x, 2.6

Table 1.1: Condor Version 6.0.3 supported platforms

1.6 Availability

Condor is currently available as a free download from the Internet via the World Wide Web at URL http://www.cs.wisc.edu/condor. Binary distributions of Condor version 6.x are available for the following UNIX platforms detailed below in Table 1.1. We define a platform to be an Architecture/Operating System combination. Condor support for Windows NT is expected in an upcoming release.

The Condor source code is no longer available for public download from the Internet. If you desire the Condor source code, please contact the Condor Team in order to discuss it further (see Section 1.8, on page 12).

1.7 Version History

This section provides descriptions of what features have been added or bugs fixed for each version of Condor. Each release series is covered in its own section.

1.7.1 Development Release Series 6.1

Please see the 6.1 manual for the 6.1 version history.

1.7.2 Stable Release Series 6.0

6.0 is the first version of Condor with *ClassAds*. It contains many other fundamental enhancements over version 5. It is also the first official stable release series, with a development series (6.1) simultaneously available.

Version 6.0.3

- Fixed a bug that was causing the hostname of the submit machine that claimed a given execute machine to be incorrectly reported by the *condor_startd* at sites using NIS.
- Fixed a bug in the *condor_startd*'s benchmarking code that could cause a floating point exception (SIGFPE, signal 8) on very, very fast machines, such as newer Alphas.
- Fixed an obscure bug in *condor_submit* that could happen when you set a requirements expression that references the "Memory" attribute. The bug only showed up with certain formations of the requirement expression.

Version 6.0.2

- Fixed a bug in the `fcntl()` call for Solaris 2.6 that was causing problems with file I/O inside Fortran jobs.
- Fixed a bug in the way the `DEFAULT_DOMAIN_NAME` parameter was handled so that this feature now works properly.
- Fixed a bug in how the `SOFT_UID_DOMAIN` config file parameter was used in the *condor_starter*. This feature is also documented in the manual now (see section 3.4.5 on page 85).
- You can now set the RunBenchmarks expression to "False" and the *condor_startd* will never run benchmarks, not even at startup time.
- Fixed a bug in `getwd()` and `getcwd()` for sites that use the NFS automounter. his bug was only present if user programs tried to call `chdir()` themselves. Now, this is supported.
- Fixed a bug in the way we were computing the available virtual memory on HPUX 10.20 machines.
- Fixed a bug in *condor_q* -analyze so it will correctly identify more situations where a job won't run.
- Fixed a bug in *condor_status* -format so that if the requested attribute isn't available for a given machine, the format string (including spaces, tabs, newlines, etc) is still printed, just the value for the requested attribute will be an empty string.
- Fixed a bug in the *condor_schedd* that was causing *condor_history* to not print out the first ClassAd attribute of all jobs that have completed
- Fixed a bug in *condor_q* that would cause a segmentation fault if the argument list was too long.

Version 6.0.1

- Fixed bugs in the `getuid()`), `getgid()`, `geteuid()`, and `getegid()` system calls.
- Multiple config files are now supported as a list specified via the `LOCAL_CONFIG_FILE` variable.
- `ARCH` and `OPSYS` are now automatically determined on all machines (including HPUX 10 and Solaris).
- Machines running IRIX now correctly suspend vanilla jobs.
- *condor_submit* doesn't allow root to submit jobs.
- The *condor_startd* now notices if you have changed `COLLECTOR_HOST` on reconfig.
- Physical memory is now correctly reported on Digital Unix when daemons are not running as root.
- New `$(SUBSYSTEM)` macro in configuration files that changes based on which daemon is reading the file (i.e. STARTD, SCHEDD, etc.) See section 3.4.1, "Condor Subsystem Names" on page 77 for a complete list of the subsystem names used in Condor.
- Port to HP-UX 10.20.
- `getrusage()` is now a supported system call. This system call will allow you to get resource usage about the entire history of your condor job.
- Condor is now fully supported on Solaris 2.6 machines (both Sparc and Intel).
- Condor now works on Linux machines with the GNU C library. This includes machines running RedHat 5.x and Debian 2.0. In addition, there seems to be a bug in RedHat that was causing the output from *condor_config_val* to not appear when used in scripts (like *condor_compile*). We put in explicit calls to flush the I/O buffers before *condor_config_val* exits, which seems to solve the problem.
- Hooks have been added to the checkpointing library to help support the checkpointing of PVM jobs.
- Condor jobs can now send signals to themselves when running in the standard universe. You do this just as you normally would:

  ```
  kill( getpid(), signal_number )
  ```

  Trying to send a signal to any other process will result in `kill()` returning -1.
- Support for NIS has been improved on Digital Unix and IRIX.
- Fixed a bug that would cause the negotiator on IRIX machines to never match jobs with available machines.

Version 6.0 pl4

NOTE: Back in the bad old days, we used this evil "patch level" version number scheme, with versions like "6.0pl4". This has all gone away in the current versions of Condor.

- Fixed a bug that could cause a segmentation violation in the *condor_schedd* under rare conditions when a *condor_shadow* exited.
- Fixed a bug that was preventing any core files that user jobs submitted to Condor might create from being transfered back to the submit machine for inspection by the user who submitted them.
- Fixed a bug that would cause some Condor daemons to go into an infinite loop if the "ps" command output duplicate entries. This only happens on certain platforms, and even then, only under rare conditions. However, the bug has been fixed and Condor now handles this case properly.
- Fixed a bug in the *condor_shadow* that would cause a segmentation violation if there was a problem writing to the user log file specified by "log = filename" in the submit file used with *condor_submit*.
- Added new command line arguments for the Condor daemons to support saving the PID (process id) of the given daemon to a file, sending a signal to the PID specified in a given file, and overriding what directory is used for logging for a given daemon. These are primarily for use with the *condor_kbdd* when it needs to be started by XDM for the user logged onto the console, instead of running as root. See section 3.9.4 on "Installing the *condor_kbdd*" on page **??** for details.
- Added support for the `CREATE_CORE_FILES` config file parameter. If this setting is defined, Condor will override whatever limits you have set and in the case of a fatal error, will either create core files or not depending on the value you specify ("true" or "false").
- Most Condor tools (*condor_on*, *condor_off*, *condor_master_off*, *condor_restart*, *condor_vacate*, *condor_checkpoint*, *condor_reconfig*, *condor_reconfig_schedd*, *condor_reschedule*) can now take the IP address and port you want to send the command to directly on the command line, instead of only accepting hostnames. This IP/port must be passed in a special format used in Condor (which you will see in the daemon's log files, etc). It is of the form: `<ip.address:port>`. For example: `<123.456.789.123:4567>`.

Version 6.0 pl3

- Fixed a bug that would cause a segmentation violation if a machine was not configured with a full hostname as either the official hostname or as any of the hostname aliases.

- If your host information does not include a fully qualified hostname anywhere, you can specify a domain in the `DEFAULT_DOMAIN_NAME` parameter in your global config file which will be appended to your hostname whenever Condor needs to use a fully qualified name.
- All Condor daemons and most tools now support a "-version" option that displays the version information and exits.
- The *condor_install* script now prompts for a short description of your pool, which it stores in your central manager's local config file as `COLLECTOR_NAME`. This description is used to display the name of your pool when sending information to the Condor developers.
- When the *condor_shadow* process starts up, if it is configured to use a checkpoint server and it cannot connect to the server, the shadow will check the `MAX_DISCARDED_RUN_TIME` parameter. If the job in question has accumulated more CPU minutes than this parameter, the *condor_shadow* will keep trying to connect to the checkpoint server until it is successful. Otherwise, the *condor_shadow* will just start the job over from scratch immediately.
- If Condor is configured to use a checkpoint server, it will only use the checkpoint server. Previously, if there was a problem connecting to the checkpoint server, Condor would fall back to using the submit machine to store checkpoints. However, this caused problems with local disks filling up on machines without much disk space.
- Fixed a rare race condition that could cause a segmentation violation if a Condor daemon or tool opened a socket to a daemon and then closed it right away.
- All TCP sockets in Condor now have the "keep alive" socket option enabled. This allows Condor daemons to notice if their peer goes away in a hard crash.
- Fixed a bug that could cause the *condor_schedd* to kill jobs without a checkpoint during its graceful shutdown method under certain conditions.
- The *condor_schedd* now supports the `MAX_SHADOW_EXCEPTIONS` parameter. If the *condor_shadow* processes for a given match die due to a fatal error (an exception) more than this number of times, the *condor_schedd* will now relinquish that match and stop trying to spawn *condor_shadow* processes for it.
- The "-master" option to *condor_status* now displays the `Name` attribute of all *condor_master* daemons in your pool, as opposed to the `Machine` attribute. This helps for pools that have submit-only machines joining them, for example.

Version 6.0 pl2

- In patch level 1, code was added to more accurately find the full hostname of the local machine. Part of this code relied on the resolver, which on many platforms is a dynamic library. On Solaris, this library has needed many security patches and the installation of Solaris on our development machines produced binaries that are incompatible with sites that haven't applied all the security patches. So, the code in Condor that relies on this library was simply removed for Solaris.

- Version information is now built into Condor. You can see the `CondorVersion` attribute in every daemon's ClassAd. You can also run the UNIX command "ident" on any Condor binary to see the version.
- Fixed a bug in the "remote submit" mode of *condor_submit*. The remote submit wasn't connecting to the specified schedd, but was instead trying to connect to the local schedd.
- Fixed a bug in the *condor_schedd* that could cause it to exit with an error due to its log file being locked improperly under certain rare circumstances.

Version 6.0 pl1

- *condor_kbdd* bug patched: On Silicon Graphics and DEC Alpha ports, if your X11 server is using Xauthority user authentication, and the *condor_kbdd* was unable to read the user's `.Xauthority` file for some reason, the *condor_kbdd* would fall into an infinite loop.
- When using a Condor Checkpoint Server, the protocol between the Checkpoint Server and the *condor_schedd* has been made more robust for a faulty network connection. Specifically, this improves reliability when submitting jobs across the Internet and using a remote Checkpoint Server.
- Fixed a bug concerning `MAX_JOBS_RUNNING`: The parameter `MAX_JOBS_RUNNING` in the config file controls the maximum number of simultaneous *condor_shadow* processes allowed on your submission machine. The bug was the number of shadow processes could, under certain conditions, exceed the number specified by `MAX_JOBS_RUNNING`.
- Added new parameter `JOB_RENICE_INCREMENT` that can be specified in the config file. This parameter specifies the UNIX nice level that the *condor_starter* will start the user job. It works just like the *renice*(1) command in UNIX. Can be any integer between 1 and 19; a value of 19 is the lowest possible priority.
- Improved response time for *condor_userprio*.
- Fixed a bug that caused periodic checkpoints to happen more often than specified.
- Fixed some bugs in the installation procedure for certain environments that weren't handled properly, and made the documentation for the installation procedure more clear.
- Fixed a bug on IRIX that could allow vanillia jobs to be started as root under certain conditions. This was caused by the non-standard uid that user "nobody" has on IRIX. Thanks to Chris Lindsey at NCSA for help discovering this bug.

- On machines where the `/etc/hosts` file is misconfigured to list just the hostname first, then the full hostname as an alias, Condor now correctly finds the full hostname anyway.
- The local config file and local root config file are now only found by the files listed in the `LOCAL_CONFIG_FILE` and `LOCAL_ROOT_CONFIG_FILE` parameters in the global config files. Previously, `/etc/condor` and user condor's home directory (˜condor) were searched as well. This could cause problems with submit-only installations of Condor at a site that already had Condor installed.

Version 6.0 pl0

- Initial Version 6.0 release.

1.8 Contact Information

The latest software releases, FAQs, and publications/papers regarding Condor and other High Throughput Computing research can be found at the official web site for Condor at http://www.cs.wisc.edu/condor.

In addition, there is an email listgroup at mailto:condor-world@cs.wisc.edu. The Condor Team uses this email listgroup to announce new releases of Condor and other major Condor-related news items. Membership into condor-world is automated by MajorDomo software. To subscribe or unsubscribe from the the list, follow the instructions at http://www.cs.wisc.edu/condor/condor-world/condor-world.html. Because many of us receive too much email as it is, you'll be happy to know that the condor-world email listgroup is moderated and only major announcements of wide interest are distributed.

Finally, you can reach the Condor Team directly. The Condor Team is comprised of the developers and administrators of Condor at the University of Wisconsin-Madison. Condor questions, comments, pleas for help, requests for commercial contract consultation or support, are all welcome; just send Internet email to mailto:condor-admin@cs.wisc.edu. Please include your name, organization, and telephone number in your message. If you are having trouble with Condor, please help us troubleshoot by including as much pertinent information as you can, including snippets of Condor log files. An archive of potentially useful previous email responses to/from condor-admin@cs.wisc.edu is also available on the web at http://www.cs.wisc.edu/condor/condor-admin.

CHAPTER

TWO

Users' Manual

2.1 Welcome to Condor

We are pleased to present Condor Version 6.0.3! Condor is developed by the Condor Team at the University of Wisconsin-Madison (UW-Madison), and was first installed as a production system in the UW-Madison Computer Sciences department nearly 10 years ago. This Condor pool has since served as a major source of computing cycles to UW faculty and students. For many, it has revolutionized the role computing plays in their research. An increase in one, and sometimes even two, orders of magnitude in the computing throughput of a research organization can have a profound impact on its size, complexity, and scope. Over the years, the Condor Team has established collaborations with scientists from around the world and has provided them with access to surplus cycles (one of whom has consumed 100 CPU years!). Today, our department's pool consists of more than 350 desktop UNIX workstations. On a typical day, our pool delivers more than 180 CPU days to UW researchers. Additional Condor pools have been established over the years across our campus and the world. Groups of researchers, engineers, and scientists have used Condor to establish compute pools ranging in size from a handful to hundreds of workstations. We hope that Condor will help revolutionize your compute environment as well.

2.2 What does Condor do?

In a nutshell, Condor is a specialized batch system for managing compute-intensive jobs. Like most batch systems, Condor provides a queueing mechanism, scheduling policy, priority scheme, and resource classifications. Users submit their compute jobs to Condor, Condor puts the jobs in a queue, runs them, and then informs the user as to the result.

Batch systems normally operate only with dedicated machines. Often termed compute servers, these dedicated machines are typically owned by one organization and dedicated to the sole purpose of running compute jobs. Condor can schedule jobs on dedicated machines. But unlike traditional batch systems, Condor is also designed to effectively utilize non-dedicated machines to run jobs. By being told to only run compute jobs on machines which are currently not being used (no keyboard activity, no load average, no active telnet users, etc), Condor can effectively harness otherwise idle machines throughout the network. This is important because often times the amount of compute power represented by the aggregate total of all the non-dedicated desktop workstations sitting on people's desks throughout the organization is far greater than the compute power of a dedicated central resource.

Condor has several unique capabilities at its disposal which are geared towards effectively utilizing non-dedicated resources that are not owned or managed by a centralized resource. These include transparent process checkpoint and migration, remote system calls, and ClassAds. Please be certain to have read section 1.3 for a general discussion about these capabilities before reading any further.

2.3 Condor Matchmaking with ClassAds

Before you start learning about how to submit a job, it is important to understand how Condor performs resource allocation. Understanding the unique framework by which Condor matches submitted jobs with machines is the key to getting the most from Condor's scheduling algorithm.

Condor is unlike most other batch systems, which typically involve submitting a job to one of several pre-defined job queues. These environments typically become both constrained and complicated, as system administrators scramble to add more queues in response to the variety of user needs. Likewise, the user is asked to make compromises and is left with the burden of not only keeping track of which queues have which properties, but also deciding which queue would be the optimal one to use for their jobs.

Instead, Condor simply acts as a matchmaker of ClassAds. Condor's ClassAds are analogous to the classified advertising section of the newspaper. Sellers advertise specifics about what they have to sell, hoping to attract a buyer. Buyers may advertise specifics about what they wish to purchase. Both buyers and sellers may have constraints that need to be satisfied. For instance, perhaps the buyer is not willing to spend more than X dollars, and the seller requires to receive a minimum of Y dollars. Furthermore, both want to rank requests from the other in such a fashion that is to their advantage. Certainly a seller would rank a buyer offering $50 dollars for a service higher than a different buyer offering $25 for the same service. In Condor, users submitting jobs can be thought of as buyers of compute resources and machine owners are the sellers.

All machines in the Condor pool advertise their attributes, such as available RAM memory, CPU type and speed, virtual memory size, current load average, and many other static and dynamic properties, in a machine ClassAd. The machine ClassAd also advertises under what conditions it is willing to run a Condor job and what type of job it would prefer. These policy attributes can reflect the individual terms and preferences by which all the different owners have graciously allowed their machine to be part of the Condor pool. For example, John Doe's machine may advertise that it is only willing to run jobs at night and when there is nobody typing at the keyboard. In addition, John Doe's machines may advertise a preference (rank) for running jobs submitted by either John Doe or one of his co-workers whenever possible.

Likewise, when submitting a job, you can specify many different job attributes, including whatever requirements and preferences for whatever type of machine you'd like this job to use. For instance, perhaps you're looking for the fastest floating-point machine available, i.e. you want to rank matches based upon floating-point performance. Or perhaps you only care that the machine has a minimum of 128 Meg of RAM. Or perhaps you'll just take any machine you can get! These job attributes and requirements are bundled up into a job ClassAd and "published" to Condor.

Condor then plays the role of a matchmaker by continuously reading through all of the job ClassAds and all of the machine ClassAds and matching and ranking job ads with machine ads, while making certain that all requirements in both ads are satisfied.

2.3.1 Inspecting Machine ClassAds with condor_status

Now would be a good time to try the *condor_status* command to get a feel for what a ClassAd actually looks like. *condor_status* displays summarizes and displays information from ClassAds about the resources available in your pool. If you just type *condor_status* and hit enter, you will see a summary of all the machine ClassAds similar to the following:

```
Name       Arch     OpSys        State      Activity   LoadAv Mem  ActvtyTime

adriana.cs INTEL    SOLARIS251   Claimed    Busy       1.000  64    0+01:10:00
alfred.cs. INTEL    SOLARIS251   Claimed    Busy       1.000  64    0+00:40:00
amul.cs.wi SUN4u    SOLARIS251   Owner      Idle       1.000  128   0+06:20:04
anfrom.cs. SUN4x    SOLARIS251   Claimed    Busy       1.000  32    0+05:16:22
anthrax.cs INTEL    SOLARIS251   Claimed    Busy       0.285  64    0+00:00:00
astro.cs.w INTEL    SOLARIS251   Claimed    Busy       0.949  64    0+05:30:00
aura.cs.wi SUN4u    SOLARIS251   Owner      Idle       1.043  128   0+14:40:15
```

...

condor_status can summarize machine ads in a wide variety of ways. For example, *condor_status -available* shows only machines which are willing to run jobs now, and *condor_status -run* shows only machines which are currently running jobs. *condor_status* can also display other types of ads other than just machine ads; *condor_status -help* lists all the options, and/or refer to the *condor_status* command reference page located on page 183.


```
MyType = "Machine"
TargetType = "Job"
Name = "alfred.cs.wisc.edu"
Machine = "alfred.cs.wisc.edu"
StartdIpAddr = "<128.105.83.11:32780>"
Arch = "INTEL"
OpSys = "SOLARIS251"
UidDomain = "cs.wisc.edu"
FileSystemDomain = "cs.wisc.edu"
State = "Unclaimed"
EnteredCurrentState = 892191963
Activity = "Idle"
EnteredCurrentActivity = 892191062
VirtualMemory = 185264
Disk = 35259
KFlops = 19992
Mips = 201
LoadAvg = 0.019531
CondorLoadAvg = 0.000000
KeyboardIdle = 5124
ConsoleIdle = 27592
Cpus = 1
Memory = 64
AFSCell = "cs.wisc.edu"
START = LoadAvg - CondorLoadAvg <= 0.300000 && KeyboardIdle > 15 * 60
Requirements = TRUE
Rank = Owner == "johndoe" || Owner == "friendofjohn"
CurrentRank =  - 1.000000
LastHeardFrom = 892191963
```

Figure 2.1: Sample output from *condor_status -l alfred*

To get a feel for what a typical machine ClassAd looks like in its entirety, use the *condor_status -l* command. Figure 2.1 shows the complete machine ad for workstation alfred.cs.wisc.edu. Some of these attributes are used by Condor itself for scheduling. Other attributes are simply informational. But the important point is that *any* of these attributes in the machine ad can be utilized at job submission time as part of a request or preference on what machine to use. Furthermore, additional attributes can be easily added; for example, perhaps your site administrator has added a physical location attribute to your machine ClassAds.

2.4 Road-map for running jobs with Condor

The road to effectively using Condor is short one. The basics are quickly and easily learned. Unlike some other network-cluster solutions, Condor typically does not require you to make any changes to your program, even to do more advanced tasks such as process checkpoint and migration.

Using Condor can be broken down into the following steps:

Job Preparation. First, you will need to prepare your job for Condor. This involves preparing it to run as a background batch job, deciding which Condor runtime environment (or *Universe*) to use, and possibly relinking your program with the Condor library via the *condor_compile* command.

Submit to Condor. Next, you'll submit your program to Condor via the *condor_submit* command. With *condor_submit* you'll tell Condor information about the run, such as what executable to run, what filenames to use for keyboard and screen (stdin and stdout) data, and where to send email when the job completes. You can also tell Condor how many times to run a program; many users may want to run the same program multiple times with multiple different data files. Finally, you'll also describe to Condor what type of machine you want to run your program.

Condor Runs the Job. Once submitted, you'll monitor your job's progress via the *condor_q* and *condor_status* commands, and/or possibly modify the order in which Condor will run your jobs with *condor_prio*. If desired, Condor can even inform you every time your job is checkpointed and/or migrated to a different machine.

Job Completion. When your program completes, Condor will tell you (via email if preferred) the exit status of your program and how much CPU and wall clock time the program used. You can remove a job from the queue prematurely with *condor_rm*.

2.5 Job Preparation

Before submitting your program to Condor, you must first make certain your program is batch ready. Next you'll need to decide upon a Condor Universe, or runtime environment, for your job.

2.5.1 Batch Ready

Condor runs your program unattended and in the background. Make certain that your program can do this before submitting it to Condor. Condor can redirect console output (stdout and stderr) and keyboard (stdin) input to/from files for you, so you may have to create file(s) that contain the proper keystrokes needed for your file.

It is also very easy to quickly submit multiple runs of your program to Condor. Perhaps you want to run the same program 500 times on 500 different input data sets. If so, you need to arrange your data files accordingly so that each run can read its own input, and so one run's output files do not clobber (overwrite) another run's files. For each individual run, Condor allows you to easily customize that run's initial working directory, stdin, stdout, stderr, command-line arguments, or shell environment. Therefore, if your program directly opens its own data files, hopefully it can read what filenames to use via either stdin or the command-line. If your program opens a static filename every time, you will likely need to make a separate subdirectory for each run to store its data files into.

2.5.2 Choosing a Condor Universe

A Universe in Condor defines an execution environment. You can state which Universe to use for each job in a submit-description file when the job is submitted. Condor Version 6.0.3 supports three different program Universes for user jobs:

- Standard
- Vanilla
- PVM

If your program is a parallel application written for PVM, then you would ask Condor for the PVM universe at submit time. See section 2.9 for information on using Condor with PVM jobs.

Otherwise, you need to decide between Standard or Vanilla Universe. In general, Standard Universe provides more services to your job than Vanilla Universe and therefore Standard is usually preferable. But Standard Universe also imposes some restrictions on what your job can do. Vanilla Universe has very few restrictions, and can be used when either the Standard Universe's additional services are not desired or when the job cannot abide by the Standard Universe's restrictions.

Standard Universe

In the Standard Universe, which is the default, Condor will automatically make checkpoints (take a snapshot of its current state) of the job. So if a Standard Universe job is running on a machine and needs to leave (perhaps because the owner of the machine returned), Condor will checkpoint the job and then migrate it to some other idle machine. Because the job was checkpointed, Condor will restart the job from the checkpoint and therefore it can continue to run from where it left off.

Furthermore, Standard Universe jobs can use Condor's *remote system calls* mechanism, which enables the program to access data files from any machine in the Condor pool regardless of whether that machine is sharing a file-system via NFS (or AFS) or if the user has an account there. Even if your files are just sitting on your local hard-drive, or in /tmp, Condor jobs can access them. How it works is when your Condor job start up on some remote machine, a corresponding *condor_shadow* process also starts up on the machine where you submitted the job. As your job runs on the remote machine, Condor traps hundreds of operating system calls (such as calls to open, read, and write files) and ships them over the network via a remote procedure call to the *condor_shadow* process. The *condor_shadow* executes the system call on the submit machine and passes the result back over the network to your Condor job. The end result is everything appears to your job like it is simply running on the submit machine, even as it bounces around to different machines in the pool.

The transparent checkpoint/migration and remote system calls are highly desirable services. However, all Standard Universe jobs must be re-linked with the Condor libraries. Although this is a simple process, after doing so there are a few restrictions on what the program can do:

1. On some platforms, specifically HPUX and Digital Unix (OSF/1), shared libraries are not supported; therefore on these platforms applications must be statically linked (Note: shared library checkpoint support is available on IRIX, Solaris, and LINUX).
2. Only single process jobs are supported, i.e. the fork(2), exec(2), system(3) and similar calls are not implemented.
3. Signals and signal handlers are supported, but Condor reserves the SIGUSR2 and SIGTSTP signals and does not permit their use by user code.
4. Most interprocess communication (IPC) calls are not supported, i.e. the socket(2), send(2), recv(2), and similar calls are not implemented.
5. All file operations must be idempotent — read-only and write-only file accesses work correctly, but programs which both read and write to the same file may not.
6. Each Condor job that has been checkpointed has an associated *checkpoint file* which is approximately the size of the address space of the process. Disk space must be available to store the checkpoint file on the submitting machine (or on a Condor Checkpoint Server if your site administrator has set one up).

Although relinking a program for use in Condor's Standard Universe is very easy to do and typically requires no changes to the program's source code, sometimes users who wish to utilize Condor do not have access to their program's source or object code. Without access to either the source or object code, relinking for the Standard Universe is impossible. This situation is typical with commercial applications, which usually only provide a binary executable and only rarely provide source or object code.

Vanilla Universe

The Vanilla Universe in Condor is for running any programs which cannot be successfully re-linked for submission into the Standard Universe. Shell scripts are another good reason to use the Vanilla Universe. However, here's the down side: Vanilla jobs cannot checkpoint or use remote system calls. So, for example, when a user returns to a workstation running a Vanilla job, Condor can either suspend the job or restart the job from the beginning someplace else. Furthermore, unlike Standard jobs, Vanilla jobs must rely on some external mechanism in place (such as NFS, AFS, etc.) for accessing data files from different machines because Remote System Calls are only available in the Standard Universe.

2.5.3 Relinking for the Standard Universe

Relinking a program with the Condor libraries (condor_rt0.o and condor_syscall_lib.a) is a simple one-step process with Condor Version 6.0.3. To re-link a program with the Condor libraries for submission into the Standard Universe, simply run *condor_compile*. See the command reference page for *condor_compile* on page 151.

Note that even once your job is re-linked, you can still run your program outside of Condor directly from the shell prompt as usual. When you do this, the following message is printed to remind you that this binary is linked with the Condor libraries:

```
WARNING: This binary has been linked for Condor.
WARNING: Setting up to run outside of Condor...
```

2.6 Submitting a Job to Condor

condor_submit is the program for actually submitting jobs to Condor. *condor_submit* wants as its sole argument the name of a submit-description file which contains commands and keywords to direct the queuing of jobs. In the submit-description file, you will tell Condor everything it needs to know about the job. Items such as the name of the executable to run, the initial working directory, command-line arguments, etc., all go into the submit-description file. *condor_submit* then creates a new job ClassAd based upon this information and ships it along with the executable to run to the *condor_schedd* daemon running on your machine. At that point your job has been submitted into Condor.

Now please read the *condor_submit* manual page in the Command Reference chapter before you continue; it is on page 186 and contains a complete and full description of how to use *condor_submit*.

2.6.1 Sample submit-description files

Now that you have read about *condor_submit* and have an idea of how it works, we'll followup with a few additional examples of submit-description files.

Example 1

Example 1 below about the simplest submit-description file possible. It queues up one copy of the program "foo" for execution by Condor. Condor will attempt to run the job on a machine which has the same architecture and operating system as the machine from which it was submitted. Since no input, output, and error commands were given, the files stdin, stdout, and stderr will all refer to /dev/null. (The program may produce output by explicitly opening a file and writing to it.)

```
####################
#
# Example 1
# Simple condor job description file
#
####################

Executable     = foo
Queue
```

Example 2

Example 2 below queues 2 copies of program the program "mathematica". The first copy will run in directory "run_1", and the second will run in directory "run_2". In both cases the names of the files used for stdin, stdout, and stderr will be test.data, loop.out, and loop.error, but the actual files will be different as they are in different directories. This is often a convenient way to organize your data if you have a large group of condor jobs to run. The example file submits "mathematica" as a Vanilla Universe job, perhaps because the source and/or object code to program "mathematica" was not available and therefore the re-link step necessary for Standard Universe jobs could not be performed.

```
####################
#
# Example 2: demonstrate use of multiple
# directories for data organization.
#
####################

Executable     = mathematica
Universe = vanilla
input   = test.data
output  = loop.out
error   = loop.error

Initialdir     = run_1
Queue

Initialdir     = run_2
Queue
```

Example 3

The submit-description file Example 3 below queues 150 runs of program "foo" which must have been compiled and linked for Silicon Graphics workstations running IRIX 6.x. Condor

will not attempt to run the processes on machines which have less than 32 megabytes of physical memory, and will run them on machines which have at least 64 megabytes if such machines are available. Stdin, stdout, and stderr will refer to "in.0", "out.0", and "err.0" for the first run of this program (process 0). Stdin, stdout, and stderr will refer to "in.1", "out.1", and "err.1" for process 1, and so forth. A log file containing entries about where/when Condor runs, checkpoints, and migrates processes in this cluster will be written into file "foo.log".

```
####################
#
# Example 3: Show off some fancy features including
# use of pre-defined macros and logging.
#
####################

Executable     = foo
Requirements   = Memory >= 32 && OpSys == "IRIX6" && Arch =="SGI"
Rank      = Memory >= 64
Image_Size     = 28 Meg

Error   = err.$(Process)
Input   = in.$(Process)
Output  = out.$(Process)
Log = foo.log

Queue 150
```

2.6.2 More about Requirements and Rank

There are a few more things you should know about the powerful **Requirements** and **Rank** commands in the submit-description file.

First of all, both of them need to be valid Condor ClassAd expressions. From the *condor_submit* manual page and the above examples, you can see that writing ClassAd expressions is quite intuitive (especially if you are familiar with the programming language C). However, there are some pretty nifty expressions you can write with ClassAds if you care to read more about them. The complete lowdown on ClassAds and their expressions can be found in section 4.1 on page 141.

All of the commands in the submit-description file are case insensitive, except for the ClassAd attribute string values that appear in the ClassAd expressions that you write! ClassAds attribute names are case insensitive, but ClassAd string values are always case sensitive. If you accidently say

```
requirements = arch == "alpha"
```

instead of what you should have said, which is:

```
requirements = arch == "ALPHA"
```

you will not get what you want.

So now that you know ClassAd attributes are case-sensitive, how do you know what the capitalization should be for an arbitrary attribute ? For that matter, how do you know what attributes you can use ? The answer is you can use any attribute that appears in either a machine or a job ClassAd. To view all of the machine ClassAd attributes, simply run *condor_status -l*. The *-l* argument to *condor_status* means to display the complete machine ClassAd. Similarly for job ClassAds, do a *condor_q -l* command (Note: you'll have to submit some jobs first before you can view a job ClassAd). This will show you all the available attributes you can play with, along with their proper capitalization.

To help you out with what these attributes all signify, below we list descriptions for the attributes which will be common by default to every machine ClassAd. Remember that because ClassAds are flexible, the machine ads in your pool may be including additional attributes specific to your site's installation/policies.

Activity : String which describes Condor job activity on the machine. Can have one of the following values:

- **"Idle"** : There is no job activity
- **"Busy"** : A job is busy running
- **"Suspended"** : A job is currently suspended
- **"Vacating"** : A job is currently checkpointing
- **"Killing"** : A job is currently being killed
- **"Benchmarking"** : The startd is running benchmarks

AFSCell : If the machine is running AFS, this is a string containing the AFS cell name.

Arch : String with the architecture of the machine. Typically one of the following:

- **"INTEL"** : Intel CPU (Pentium, Pentium II, etc).
- **"ALPHA"** : Digital ALPHA CPU
- **"SGI"** : Silicon Graphics MIPS CPU
- **"SUN4u"** : Sun ULTRASPARC CPU
- **"SUN4x"** : A Sun SPARC CPU other than an ULTRASPARC, i.e. sun4m or sun4c CPU found in older SPARC workstations such as the Sparc~10, Sparc~20, IPC, IPX, etc.
- **"HPPA1"** : Hewlett Packard PA-RISC 1.x CPU (i.e. PA-RISC 7000 series CPU) based-workstation

- **"HPPA2"** : Hewlett Packard PA-RISC 2.x CPU (i.e. PA-RISC 8000 series CPU) based-workstation

ClockDay : The day of the week, where 0 = Sunday, 1 = Monday, . . ., 6 = Saturday.

ClockMin : The number of minutes passed since midnight.

CondorLoadAvg : The load average generated by Condor (either from remote jobs or running benchmarks).

ConsoleIdle : The number of seconds since activity on the system console keyboard or console mouse has last been detected.

Cpus : Number of CPUs in this machine, i.e. 1 = single CPU machine, 2 = dual CPUs, etc.

CurrentRank : A float which represents this machine owner's affinity for running the Condor job which it is currently hosting. If not currently hosting a Condor job, CurrentRank is -1.0.

Disk : The amount of disk space on this machine available for the job in kbytes (e.g. 23000 = 23 megabytes). Specifically, this is amount of disk space available in the directory specified in the Condor configuration files by the macro `EXECUTE`, minus any space reserved with the macro `RESERVED_DISK`.

EnteredCurrentActivity : Time at which the machine entered the current Activity (see `Activity` entry above). Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970).

FileSystemDomain : a domain name configured by the Condor administrator which describes a cluster of machines which all access the same networked filesystems usually via NFS or AFS.

KeyboardIdle : The number of seconds since activity on any keyboard or mouse associated with this machine has last been detected. Unlike `ConsoleIdle`, `KeyboardIdle` also takes activity on pseudo-terminals into account (i.e. virtual "keyboard" activity from telnet and rlogin sessions as well). Note that `KeyboardIdle` will always be equal to or less than `ConsoleIdle`.

KFlops : Relative floating point performance as determined via a linpack benchmark.

LastHeardForm : Time when the Condor Central Manager last received a status update from this machine. Expressed as seconds since the epoch.

LoadAvg : A floating point number with the machine's current load average.

Machine : A string with the machine's fully qualified hostname.

Memory : The amount of RAM in megabytes.

Mips : Relative integer performance as determined via a dhrystone benchmark.

MyType : The ClassAd type; always set to the literal string "Machine".

Name : The name of this resource; typically the same value as the `Machine` attribute, but could be customized by the site administrator.

OpSys : String describing the operating system running on this machine. For Condor Version 6.0.3 typically one of the following:

"HPUX10" (for HPUX 10.20)

"IRIX6" (for IRIX 6.2, 6.3, or 6.4)

"LINUX" (for LINUX 2.x kernel systems)

"OSF1" (for Digital Unix 4.x)

"SOLARIS251"

"SOLARIS26"

Requirements : A boolean which, when evaluated within the context of the Machine ClassAd and a Job ClassAd, must evaluate to TRUE before Condor will allow the job to use this machine.

StartdIpAddr : String with the IP and port address of the *condor_startd* daemon which is publishing this Machine ClassAd.

State : String which publishes the machine's Condor state, which can be:

"Owner" : The machine owner is using the machine, and it is unavailable to Condor.

"Unclaimed" : The machine is available to run Condor jobs, but a good match (i.e. job to run here) is either not available or not yet found.

"Matched" : The Condor Central Manager has found a good match for this resource, but a Condor scheduler has not yet claimed it.

"Claimed" : The machine is claimed by a remote *condor_schedd* and is probably running a job.

"Preempting" : A Condor job is being preempted (possibly via checkpointing) in order to clear the machine for either a higher priority job or because the machine owner wants the machine back.

TargetType : Describes what type of ClassAd to match with. Always set to the string literal "Job", because Machine ClassAds always want to be matched with Jobs, and vice-versa.

UidDomain : a domain name configured by the Condor administrator which describes a cluster of machines which all have the same "passwd" file entries, and therefore all have the same logins.

VirtualMemory : The amount of currently available virtual memory (swap space) expressed in kbytes.

2.6.3 Hetrogeneous submit: submit to a different architecture

There are times when you would like to submit jobs across machine architectures. For instance, let's say you have an Intel machine running LINUX sitting on your desk. This is the machine where you do all your work and where all your files are stored. But perhaps the majority of machines in your pool are Sun SPARC machines running Solaris. You would want to submit jobs directly from your LINUX box that would run on the SPARC machines.

This is easily accomplished. You will need, or course, to create your executable on the same type of machine where you want your job to run — Condor will not convert machine instructions hetrogeneously for you! The trick is simply what to specify for your **requirements** command in your submit-description file. By default, *condor_submit* inserts requirements that will make your job run on the same type of machine you are submitting from. To override this, simply state what you want. Returning to our example, you would put the following into your submit-description file:

```
requirements = Arch == "SUN4x" && OpSys == "SOLARIS251"
```

Just run *condor_status* to display the Arch and OpSys values for any/all machines in the pool.

2.7 Managing a Condor Job

This section provides a brief summary of what can be done once your jobs begin execution. The basic mechanisms for monitoring a job are introduced, but the commands that are discussed include a lot more functionality than is displayed in this section. You are encouraged to look at the man pages of the commands referred to (located in Chapter 5 beginning on page 148) for more information.

Once your jobs have been submitted, Condor will attempt to find resources to run your jobs. The existence of your requests is communicated to the Condor manager by registering you as a "submitter." A list of all the current submitters may be obtained through *condor_status* with the *-submitters* option, which would yield output similar to the following:

```
%  condor_status -submitters

Name                 Machine     Running  IdleJobs  MaxJobsRunning

ashoks@jules.ncsa.ui jules.ncsa       74        54       200
breach@cs.wisc.edu   bianca.cs.       11         0       500
breach@cs.wisc.edu   neufchatel       23         0       500
jbasney@cs.wisc.edu  froth.cs.w        0         1       500
wright@raven.cs.wisc raven.cs.w        1        48       200
```

```
                          RunningJobs           IdleJobs

wright@raven.cs.wisc                1                 48
ashoks@jules.ncsa.ui               74                 54
 jbasney@cs.wisc.edu                0                  1
  breach@cs.wisc.edu               34                  0

               Total              109                103
```

2.7.1 Checking on the progress of your jobs

At any time, you can check on the status of your jobs with the *condor_q* tool, which shows the status of all queued jobs along with other information. To identify jobs which are running, type

```
%  condor_q

-- Submitter: froth.cs.wisc.edu : <128.105.73.44:33847> : froth.cs.wisc.edu
 ID      OWNER            SUBMITTED    CPU_USAGE ST PRI SIZE CMD
 125.0   jbasney         4/10 15:35   0+00:00:00 U  -10 1.2  hello.remote
 127.0   raman           4/11 15:35   0+00:00:00 R  0   1.4  hello
 128.0   raman           4/11 15:35   0+00:02:33 I  0   1.4  hello

3 jobs; 1 unexpanded, 1 idle, 1 running, 0 malformed
```

The `ST` column (for status) shows the status of current jobs in the queue. The "U" stands for unexpanded, which means that the job has never checkpointed and when it starts running it will start running from the beginning. "R" means the the job is currently running. Finally, "I" stands for idle, which means the job has run before and has checkpointed, and when it starts running again it will resume where it left off, but is not running right now because it is waiting for a machine to become available.

Note: The CPU time reported for a job is the time that has been committed to the job. Thus, the CPU time is not updated for a job until the job checkpoints, at which time the job has made guaranteed forward progress. Depending upon how the site administrator configured the pool, several hours may pass between checkpoints, so do not worry if you do not observe CPU changing by the hour. Also note that this is actual CPU time as reported by the operating system; this is not wall-clock time.

Another useful method of tracking the progress of jobs is through the *user log* mechanism. If you have specified a `log` command in your submit file, the progress of the job may be followed by viewing the log file. Various events such as execution commencement, checkpoint, eviction and termination are logged in the file along with the time at which the event occurred.

2.7.2 Removing the job from the queue

A job can be removed from the queue at any time by using the *condor_rm* command. If the job that is being removed is currently running, the job is killed without a checkpoint, and its queue entry removed. For example:

```
%  condor_q

-- Submitter: froth.cs.wisc.edu : <128.105.73.44:33847> : froth.cs.wisc.edu
 ID      OWNER            SUBMITTED    CPU_USAGE ST PRI SIZE CMD
 125.0   jbasney         4/10 15:35   0+00:00:00 U  -10 1.2  hello.remote
 132.0   raman           4/11 16:57   0+00:00:00 R  0   1.4  hello

2 jobs; 1 unexpanded, 0 idle, 1 running, 0 malformed

%  condor_rm 132.0
Job 132.0 removed.

%  condor_q

-- Submitter: froth.cs.wisc.edu : <128.105.73.44:33847> : froth.cs.wisc.edu
 ID      OWNER            SUBMITTED    CPU_USAGE ST PRI SIZE CMD
 125.0   jbasney         4/10 15:35   0+00:00:00 U  -10 1.2  hello.remote

1 jobs; 1 unexpanded, 0 idle, 0 running, 0 malformed
```

2.7.3 While your job is running ...

When your job begins to run, Condor starts up a *condor_shadow* process on the submit machine. The shadow process is the mechanism by which the remotely executing jobs can access the environment from which it was submitted, such as input and output files.

It is normal for a machine which has submitted hundreds of jobs to have hundreds of shadows running on the machine. Since the text segments of all these processes is the same, the load on the submit machine is usually not significant. If, however, you notice degraded performance, you can limit the number of jobs that can run simultaneously through the `MAX_JOBS_RUNNING` configuration parameter. Please talk to your system administrator for the necessary configuration change.

You can also find all the machines that are running your job through the *condor_status* command. For example, to find all the machines that are running jobs submitted by "breach@cs.wisc.edu," type:

```
%  condor_status -constraint 'RemoteUser == "breach@cs.wisc.edu"'
```

```
Name       Arch     OpSys        State      Activity   LoadAv Mem  ActvtyTime

alfred.cs. INTEL    SOLARIS251   Claimed    Busy       0.980  64    0+07:10:02
biron.cs.w INTEL    SOLARIS251   Claimed    Busy       1.000  128   0+01:10:00
cambridge. INTEL    SOLARIS251   Claimed    Busy       0.988  64    0+00:15:00
falcons.cs INTEL    SOLARIS251   Claimed    Busy       0.996  32    0+02:05:03
happy.cs.w INTEL    SOLARIS251   Claimed    Busy       0.988  128   0+03:05:00
istat03.st INTEL    SOLARIS251   Claimed    Busy       0.883  64    0+06:45:01
istat04.st INTEL    SOLARIS251   Claimed    Busy       0.988  64    0+00:10:00
istat09.st INTEL    SOLARIS251   Claimed    Busy       0.301  64    0+03:45:00
...
```

To find all the machines that are running any job at all, type:

```
%  condor_status -run

Name       Arch     OpSys        LoadAv RemoteUser           ClientMachine

adriana.cs INTEL    SOLARIS251   0.980  hepcon@cs.wisc.edu   chevre.cs.wisc.
alfred.cs. INTEL    SOLARIS251   0.980  breach@cs.wisc.edu   neufchatel.cs.w
amul.cs.wi SUN4u    SOLARIS251   1.000  nice-user.condor@cs. chevre.cs.wisc.
anfrom.cs. SUN4x    SOLARIS251   1.023  ashoks@jules.ncsa.ui jules.ncsa.uiuc
anthrax.cs INTEL    SOLARIS251   0.285  hepcon@cs.wisc.edu   chevre.cs.wisc.
astro.cs.w INTEL    SOLARIS251   1.000  nice-user.condor@cs. chevre.cs.wisc.
aura.cs.wi SUN4u    SOLARIS251   0.996  nice-user.condor@cs. chevre.cs.wisc.
balder.cs. INTEL    SOLARIS251   1.000  nice-user.condor@cs. chevre.cs.wisc.
bamba.cs.w INTEL    SOLARIS251   1.574  dmarino@cs.wisc.edu  riola.cs.wisc.e
bardolph.c INTEL    SOLARIS251   1.000  nice-user.condor@cs. chevre.cs.wisc.
...
```

2.7.4 Changing the priority of jobs

In addition to the priorities assigned to each user, Condor also provides each user with the capability of assigning priorities to each submitted job. These job priorities are local to each queue and range from -20 to +20, with higher values meaning better priority.

The default priority of a job is 0, but can be changed using the *condor_prio* command. For example, to change the priority of a job to -15,

```
%  condor_q raman

-- Submitter: froth.cs.wisc.edu : <128.105.73.44:33847> : froth.cs.wisc.edu
 ID      OWNER            SUBMITTED    CPU_USAGE ST PRI SIZE CMD
 126.0   raman           4/11 15:06   0+00:00:00 U  0    0.3  hello
```



```
1 jobs; 1 unexpanded, 0 idle, 0 running, 0 malformed

%  condor_prio -p -15 126.0

%  condor_q raman

-- Submitter: froth.cs.wisc.edu : <128.105.73.44:33847> : froth.cs.wisc.edu
 ID      OWNER            SUBMITTED    CPU_USAGE ST PRI SIZE CMD
 126.0   raman           4/11 15:06   0+00:00:00 U  -15 0.3  hello

1 jobs; 1 unexpanded, 0 idle, 0 running, 0 malformed
```

It is important to note that these *job* priorities are completely different from the *user* priorities assigned by Condor. Job priorities do not impact user priorities. They are only a mechanism for the user to identify the relative importance of jobs among all the jobs submitted by the user to that specific queue.

2.7.5 Why won't my job run?

Users sometimes find that their jobs do not run. There are several reasons why a specific job does not run. These reasons range from failed job or machine constraints, bias due to preferences, insufficient priority, or the preemption "throttle" that is implemented by the *condor_negotiator* to prevent thrashing. Many of these reasons can be diagnosed by using the *-analyze* option of *condor_q*. For example, the following job submitted by user "jbasney" was found not to run for several days.

```
% condor_q

-- Submitter: froth.cs.wisc.edu : <128.105.73.44:33847> : froth.cs.wisc.edu
 ID      OWNER            SUBMITTED    CPU_USAGE ST PRI SIZE CMD
 125.0   jbasney         4/10 15:35   0+00:00:00 U  -10 1.2  hello.remote

1 jobs; 1 unexpanded, 0 idle, 0 running, 0 malformed
```

Running *condor_q*'s analyzer provided the following information:

```
%  condor_q 125.0 -analyze

-- Submitter: froth.cs.wisc.edu : <128.105.73.44:33847> : froth.cs.wisc.edu
---
125.000:  Run analysis summary.  Of 323 resource offers,
          323 do not satisfy the request's constraints
            0 resource offer constraints are not satisfied by this request
```

```
          0 are serving equal or higher priority customers
          0 are serving more preferred customers
          0 cannot preempt because preemption has been held
          0 are available to service your request

WARNING:  Be advised:
   No resources matched request's constraints
   Check the Requirements expression below:

Requirements = Arch == "INTEL" && OpSys == "SOLARIS251" && 0 &&
  Disk >= ExecutableSize && VirtualMemory >= ImageSize
```

We see that user "jbasney" has inadvertently expressed a `Requirements` expression that can never be satisfied due to the `... && 0 && ...` clause which always evaluates to false.

While the analyzer can diagnose most common problems, there are some situations that it cannot reliably detect due to the instantaneous and local nature of the information it uses to detect the problem. Thus, it may be that the analyzer reports that resources are available to service the request, but the job still does not run. In most of these situations, the delay is transient, and the job will run during the next negotiation cycle.

If the problem persists and the analyzer is unable to detect the situation, it may be that the job begins to run but immediately terminates due to some problem. Viewing the job's error and log files (specified in the submit command file) and Condor's `SHADOW_LOG` file may assist in tracking down the problem. If the cause is still unclear, please contact your system administrator.

2.8 Priorities in Condor

Condor has two independent priority controls: *job* priorities and *user* priorities.

2.8.1 Job Priority

Job priorities allow you to assign a priority level to each of your jobs in order to control their order of execution. To do this, use the *condor_prio* command — see the example in section 2.7.4, or the command reference page on page 168. Job priorities, however, do not impact user priorities in any fasion. No matter what you set a job's priority to be, it will not alter your user priority in relation to other users. Job priorities range from -20 to +20, with +20 being the best and -20 the worst.

2.8.2 User priority

Machines are allocated to users based upon that user's priority. User priorities in Condor can be examined with the *condor_userprio* command (see page 194), and Condor administrators can set and edit individual user priorities with the same utility. A lower numerical user priority value means higher priority, so a user with priority 5 will get more resources than a user with priority 50.

Condor continuously calculates the share of available machines that each user should be allocated. This share is inversely related to the ratio between user priorities; for example, a user with a priority of 10 will get twice as many machines as a user with a priority of 20. The priority of each individual user changes according to the number of resources he is using. Each user starts out with a priority of .5 (the best priority allowed). If the number of machines a user currently has is greater than his priority, the priority will numerically increase (worsen) over time, and if it is less then the priority, the priority will numerically decrease (improve) over time. The long-term result is fair-share access across all users. The speed at which Condor adjusts the priorities is controlled via an exponential half-life value (parameter `PRIORITY_HALFLIFE` which can be adjusted by the site administrator) which has a default of one day. So if a user with a user priority of 100 is utilizing 100 machines and then deletes all his/her jobs, one day later that user's priority will 50, two days later the priority will be 25, etc.

Condor enforces that each user gets his/her fair share of machines according to user priority both when allocating machines which become available and by priority preemption of currently allocated machines. For instance, if a low priority user is utilizing all available machines and suddenly a higher priority user submits jobs, Condor will imediately checkpoint and vacate jobs belonging to the lower priority user. This will free up machines that Condor will then give over to the higher priority user. Condor will not starve the lower priority user; it will preempt only enough jobs so that the higher priority user's fair share can be realized (based upon the ratio between user priorities). To prevent thrashing of the system due to priority preemption, the Condor site administrator can define a `PREEMPTION_HOLD` expression in Condor's configuration. The default expression that ships with Condor is configured to only preempt lower priority jobs that have run for at least one hour. So in the previous example, in the worse case it could take up to a maximum of one hour until the higher priority user receives his fair share of machines.

User priorities are keyed on "username@domain", for example "johndoe@cs.wisc.edu". (The domainname to use, if any, is also configured by the Condor site administrator). Thus, user priority and therefore resource allocation is not impacted by which machine the user submits from or even if the user submits jobs from multiple machines.

Finally, any job submitted to Condor can be specified as a "nice" job at the time the job is submitted (see page 190). Nice jobs will artificially have their numerical priority boosted by over one million. This effectively means that nice jobs will only run on machines that no other Condor job (i.e. non-niced job) wants. Similarly, the Condor administrators could set the numerical priority of any individual Condor user, such as a guest account, so that these guest accounts would only use cycles not wanted by other users of the system.

2.9 Parallel Applications in Condor: Condor-PVM

Condor has a PVM submit Universe which allows the user to submit PVM jobs to the Condor pool. In this section, we will first discuss the differences between running under normal PVM and running PVM under the Condor environment. Then we give some hints on how to write good PVM programs to suit the Condor environment via an example program. In the end, we illustrate how to submit PVM jobs to Condor by examining a sample Condor submit-description file which submits a PVM job.

NOTE: Condor-PVM is an optional Condor module. To check and see if it has been installed at your site, enter the following command:

```
ls -l ‘condor_config_val PVMD‘
```

(notice the use of backticks in the above command). If this shows the file “condor_pvmd” on your system, Condor-PVM is installed. If not, ask your site administrator to download Condor-PVM from http://www.cs.wisc.edu/condor in the “contrib” section of the ”Downloads” page and install it.

2.9.1 What does Condor-PVM do?

Condor-PVM provides a framework to run parallel applications written to PVM in Condor’s opportunistic environment. This means that you no longer need a set of dedicated machines to run PVM applications; Condor can be used to dynamically construct PVM virtual machines out of non-dedicated desktop machines on your network which would have otherwise been idle. In Condor-PVM, Condor acts as the resource manager for the PVM daemon. Whenever your PVM program asks for nodes (machines), the request is re-mapped to Condor. Condor then finds a machine in the Condor pool via the usual mechanisms, and adds it to the PVM virtual machine. If a machine needs to leave the pool, your PVM program is notified of that as well via the normal PVM mechanisms.

2.9.2 The Master-Worker Paradigm

There are several different parallel programming paradigms. One of the more common is the *master-worker* (or *pool of tasks*) arrangement. In a master-worker program model, one node acts as the controlling master for the parallel application and sends pieces work out to worker nodes. The worker node does some computation, and sends the result back to the master node. The master has a pool of work that needs to be done, and simply assigns the next piece of work out to the next worker that becomes available.

Not all parallel programming paradigms lend themselves to an opportunistic environment. In such an environment, any of the nodes could be preempted and therefore disappear at any moment. The master-worker model, on the other hand, is a model that can work

well. The idea is the master needs to keep track of which piece of work it sends to each worker. If the master node is then informed that a worker has disappeared, it puts the piece of work it assigned to that worker back into the pool of tasks, and sends it out again to the next available worker. If the master notices that the number of workers has dropped below an acceptable level, it could request for more workers (via `pvm_addhosts()`). Or perhaps perhaps the master will request a replacement node every single time it is notified that a worker has gone away. The point is that in this paradigm, the number of workers is not important (although more is better!) and changes in the size of the virtual machine can be handled naturally.

Condor-PVM is designed to run PVM applications which follow the master-worker paradigm. Condor runs the master application on the machine where the job was submitted and will not preempt it. Workers are pulled in from the Condor pool as they become available.

2.9.3 Binary Compatibility

Condor-PVM does not define a new API (application program interface); programs can simply use the existing resource management PVM calls such as `pvm_addhosts()` and `pvm_notify()`. Because of this, some master-worker PVM applications are ready to run under Condor-PVM with no changes at all. Regardless of using Condor-PVM or not, it is good master-worker design to handle the case of a worker node disappearing, and therefore many programmers have already constructed their master program with all the necessary logic for fault-tolerance purposes.

In fact, regular PVM and Condor-PVM are binary compatible with each other. The same binary which runs under regular PVM will run under Condor, and vice-versa. There is no need to re-link for Condor-PVM. This permits easy application development (develop your PVM application interactively with the regular PVM console, XPVM, etc) as well as binary sharing between Condor and some dedicated MPP systems.

2.9.4 Runtime differences between Condor-PVM and regular PVM

This release of the Condor-PVM is based on PVM 3.3.11. The vast majority of the PVM library functions under Condor maintain the same semantics as in PVM 3.3.11, including messaging operations, group operations, and pvm_catchout().

We summarize the changes and new features of PVM under running in the Condor environment in the following list:

- Concept of machine class. Under Condor-PVM, machines of different architectures attributes belong to different machine classes. Machine classes are are numbered 0, 1, . . ., etc. A machine class can be specified by the user in the submit-description file when the job is submitted to Condor.

- `pvm_addhosts()`. When the application needs to add a host machine, it should call `pvm_addhosts()` with the first argument as a string that specifies the machine class. For example, to specify class 0, a pointer to string "0" should be used as the first argument. Condor will find a machine that satisfies the requirements of class 0 and adds it to the PVM virtual machine.

  Furthermore, `pvm_addhosts()` no longer blocks under Condor. It will return immediately, before the hosts are actually added to the virtual machine. After all, in a non-dedicated environment the amount of time it takes until a machine becomes available is not bound. The user should simply call `pvm_notify()` before calling `pvm_addhosts()`, so that when a host is added later, the user will be notified via Condor in the usual PVM fashion (via a PvmHostAdd notification message).

- `pvm_notify()`. Under Condor, we added two additional possible notification requests, `PvmHostSuspend` and `PvmHostResume`, to the function `pvm_notify()`. When a host is suspended (or resumed) by Condor, if the user has called `pvm_notify()` with that host tid and with `PvmHostSuspend` (or `PvmHostResume`) as arguments, then the application will receive a notification for the corresponding event.

- `pvm_spawn()`. If the flag in `pvm_spawn()` is PvmTaskArch, then the string specifying the desired architecture class should be used. Typically, if you are using only one class of machine in your virtual machine, specify "0" as the desired architecture.

  Furthermore, under Condor we currently only allow one PVM task to be spawned per node, since Condor's typical setup at most sites will suspend or vacate a job if the load on its machine is higher than a specified threshold.

2.9.5 A Sample PVM program for Condor-PVM

Normal PVM applications assume dedicated machines. However, when running a PVM application under Condor, since Condor's environment is an opportunistic environment, machines can be suspended and even removed from the PVM virtual machine during the life-time of the PVM application.

Here, we include an extensively commented skeleton of a sample PVM program *master_sum.c*, which, we hope, will help you to write PVM code that is better suited for a non-dedicated opportunistic environment like Condor.

```
/*
 * master_sum.c
 *
 * master program to perform parallel addition - takes a number n
 * as input and returns the result of the sum 0..(n-1).  Addition
 * is performed in parallel by k tasks, where k is also taken as
 * input.  The numbers 0..(n-1) are stored in an array, and each
 * worker adds a portion of the array, and returns the sum to the
 * master.  The Master adds these sums and prints final sum.
 *
 * To make the program fault-tolerant, the master has to monitor
 * the tasks that exited without sending the result back.  The
 * master creates some new tasks to do the work of those tasks
```



```
 * that have exited.
 */

#define NOTIFY_NUM 5  /* number of items to notify */

#define HOSTDELETE 12
#define HOSTSUSPEND 13
#define HOSTRESUME 14
#define TASKEXIT 15
#define HOSTADD 16

/* send the pertask and start number to the worker task i */
void send_data_to_worker(int i, int *tid, int *num, int pertask,
            FILE *fp, int round)
{
     int status;
     int start_val;

     /* send the round number */
     pvm_initsend(PvmDataDefault); /* XDR format */
     pvm_pkint(&round, 1, 1);    /* number of numbers to add */
     status = pvm_send(tid[i], ROUND_TAG);

     pvm_initsend(PvmDataDefault); /* XDR format */
     pvm_pkint(&pertask, 1, 1);    /* number of numbers to add */
     status = pvm_send(tid[i], NUM_NUM_TAG);

     pvm_initsend(PvmDataDefault); /* XDR format */
     start_val = i * pertask; /* initial number for this task */
     pvm_pkint(&start_val, 1, 1);     /* the initial number */
     status = pvm_send(tid[i], START_NUM_TAG);

     fprintf(fp, "Round %d: Send data %d to worker task %d, ``
           ``tid =%x. status %d \n", round, start_val, i, tid[i], status);
}

/*
 * to see if more hosts are needed
 * 1 = yes; 0 = no
 */
int need_more_hosts(int i)
{
     int nhost, narch;
     char *hosts="0";  /* any host in arch class 0 */
     struct pvmhostinfo *hostp = (struct pvmhostinfo *)
                     calloc (1, sizeof(struct pvmhostinfo));

     /* get the current configuration */
     pvm_config(&nhost, &narch, &hostp);

     if (nhost > i)
        return 0;
     else
        return 1;
}

/*
 * Add a new host until success, assuming that request for
 * PvmAddHost notification has already been sent
 */
void add_a_host(FILE *fp)
{
     int done = 0;
     int buf_id;
     int success = 0;
     int tid;
     int msg_len, msg_tag, msg_src;
     char *hosts="0";  /* any host in arch class 0 */
     int infos[1];
```

```
     while (done != 1) {
        /*
        * add one host - no specific machine named
        * add host will asynchronously, so we need
        * to receive the notification before go on.
        */
        pvm_addhosts(&hosts,1 , infos);

        /* receive hostadd notification from anyone */
        buf_id = pvm_recv(-1, HOSTADD);

        if (buf_id < 0) {
            fprintf(fp, "Error with buf_id = %d\n", buf_id);
            done = 0;
            continue;
        }
        done = 1;

        pvm_bufinfo(buf_id, &msg_len , &msg_tag, &msg_src);
        pvm_upkint(&tid, 1, 1);

        pvm_notify(PvmHostDelete, HOSTDELETE, 1, &tid);

        fprintf(fp, "Received HOSTADD: ");
        fprintf(fp, "Host %x added from %x\n", tid, msg_src);
        fflush(fp);
    }
}

/*
 * Spawn a worker task until success.
 * Return its tid, and the tid of its host.
 */
void spawn_a_worker(int i, int* tid, int * host_tid, FILE *fp)
{
     int numt = 0;
     int status;

     while (numt == 0){
          /* spawn a worker on a host belonging to arch class 0 */
          numt = pvm_spawn ("worker_sum", NULL, PvmTaskArch, "0", 1, &tid[i]);

          fprintf(fp, "master spawned %d task tid[%d] = %x\n",numt,i,tid[i]);
          fflush(fp);

          /* if the spawn is successful */
          if (numt == 1) {
               /* notify when the task exits */
               status = pvm_notify(PvmTaskExit, TASKEXIT, 1, &tid[i]);

               fprintf(fp, "Notify status for exit = %d\n", status);

               if (pvm_pstat(tid[i]) != PvmOk) numt = 0;
          }

          if (numt != 1) {
               fprintf(fp, "!! Failed to spawn task[%d]\n", i);

               /*
                * currently Condor-pvm allows only one task running on
                * a host
                */
                while (need_more_hosts(i) == 1)
                    add_a_host(fp);
          }
     }
}
```



```
main()
{
    int n;                  /* will add <n> numbers n .. n-1 */
    int ntasks;             /* need <ntask> workers to do the addition. */
    int pertask;            /* numbers to add per task */
    int tid[MAX_TASKS];     /* tids of tasks */
    int deltid[MAX_TASKS];  /* tids monitored for deletion */
    int sum[MAX_TASKS];     /* hold the reported sum */
    int num[MAX_TASKS];     /* the initial numbers the workers should add */
    int host_tid[MAX_TASKS];/* the tids of the host that the *
                             * tasks <0..ntasks> are running on*/

    int i, numt, nhost, narch, status;
    int result;
    int mytid;    /* task id of master */
    int mypid;    /* process id of master */
    int buf_id;   /* id of recv buffer */
    int msg_leg, msg_tag, msg_src, msg_len;
    int int_val;

    int infos[MAX_TASKS];
    char * hosts[MAX_TASKS];
    struct pvmhostinfo *hostp = (struct pvmhostinfo *)
                   calloc (MAX_TASKS, sizeof(struct pvmhostinfo));

    FILE *fp;
    char outfile_name[100];

    char *codes[NOTIFY_NUM] = {"HostDelete", "HostSuspend",
            "HostResume", "TaskExit", "HostAdd"};

    int count;   /* the number of times that while loops */
    int round_val;
    int correct = 0;
    int wrong = 0;

    mypid = getpid();

    sprintf(outfile_name, "out_sum.%d", mypid);
    fp = fopen(outfile_name, "w");

    /* redirect all children tasks' stdout to fp */
    pvm_catchout(stderr);

    if (pvm_parent() == PvmNoParent){
        fprintf(fp, "I have no parent!\n");
        fflush(fp);
    }

    /* will add <n> numbers 0..(n-1) */
    fprintf(fp, "How many numbers? ");
    fflush(fp);
    scanf("%d", &n);
    fprintf(fp, "%d\n", n);
    fflush(fp);

    /* will spawn ntasks workers to perform addition */
    fprintf(fp, "How many tasks? ");
    fflush(fp);
    scanf("%d", &ntasks);
    fprintf(fp, "%d\n\n", ntasks);
    fflush(fp);

    /* will iterate count loops */
    fprintf(fp, "How many loops? ");
    fflush(fp);
    scanf("%d", &count);
    fprintf(fp, "%d\n", count);
```

```
fflush(fp);

/* set the hosts to be in arch class 0 */
for (i = 0; i< ntasks; i++) hosts[i] = "0";

/* numbers to be added by each worker */
pertask = n/ntasks;

/* get the master's TID */
mytid = pvm_mytid();
fprintf(fp, "mytid = %x; mypid = %d\n", mytid, mypid);

/* get the current configuration */
pvm_config(&nhost, &narch, &hostp);

fprintf(fp, "current number of hosts = %d\n", nhost);
fflush(fp);

/*
 * notify request for host addition, with tag HOSTADD,
 * no tids to monitor.
 *
 * -1 turns the notification request on;
 * 0 turns it off;
 * a positive integer n will generate at most n
 * notifications.
 */
pvm_notify(PvmHostAdd, HOSTADD, -1, NULL);

/* add more hosts - no specific machine named */
i = ntasks - nhost;
if (i > 0) {
    status = pvm_addhosts(hosts, i , infos);

    fprintf(fp, "master: addhost status = %d\n", status);
    fflush(fp);
}

/* if not enough hosts, loop and call pvm_addhosts */
for (i = nhost; i < ntasks; i++) {
    /* receive notification from anyone, with HostAdd tag */
    buf_id = pvm_recv(-1, HOSTADD);

    if (buf_id < 0) {
       fprintf(fp, "Error with buf_id = %d\n", buf_id);
    } else {
       fprintf(fp, "Success with buf_id = %d\n", buf_id);
    }

    pvm_bufinfo(buf_id, &msg_len , &msg_tag, &msg_src);
    if (msg_tag==HOSTADD) {
        pvm_upkint(&int_val, 1, 1);

        fprintf(fp, "Received HOSTADD: ");
        fprintf(fp, "Host %x added from %x\n", int_val, msg_src);
       fflush(fp);
    } else {
       fprintf(fp, "Received unexpected message with tag: %d\n", msg_tag);
    }
}

/* get current configuration */
pvm_config(&nhost, &narch, &hostp);

/* notify all exceptional conditions about the hosts*/
status = pvm_notify(PvmHostDelete, HOSTDELETE, ntasks, deltid);
fprintf(fp, "Notify status for delete = %d\n", status);

status = pvm_notify(PvmHostSuspend, HOSTSUSPEND, ntasks, deltid);
```



```
fprintf(fp, "Notify status for suspend = %d\n", status);

status = pvm_notify(PvmHostResume, HOSTRESUME, ntasks, deltid);
fprintf(fp, "Notify status for resume = %d\n", status);

/* spawn <ntasks> */
for (i = 0; i < ntasks ; i++) {
    /* spawn the i-th task, with notifications. */
    spawn_a_worker(i, tid, host_tid, fp);
}

/* add the result <count> times */
while (count > 0) {
    /*
     * if array length was not perfectly divisible by ntasks,
     *    some numbers are remaining. Add these yourself
     */
    result = 0;
    for ( i = ntasks * pertask ; i < n ; i++)
       result += i;

    /* initialize the sum array with -1 */
    for (i = 0; i< ntasks; i++)
        sum[i] = -1;

    /* send array partitions to each task */
    for (i = 0; i < ntasks ; i++) {
       send_data_to_worker(i, tid, num, pertask, fp, count);
    }

    /*
    * Wait for results.  If a task exited without
    * sending back the result, start another task to do
    * its job.
    */
    for (i = 0; i< ntasks; ) {
        buf_id = pvm_recv(-1, -1);
        pvm_bufinfo(buf_id, &msg_len , &msg_tag, &msg_src);
        fprintf(fp, "Receive: task %x returns mesg tag %d, ``
            ``buf_id = %d\n", msg_src, msg_tag, buf_id);
        fflush(fp);

        /* is a result returned by a worker */
        if(msg_tag == RESULT_TAG)  {
            int j;

            pvm_upkint(&round_val, 1, 1);
            fprintf(fp, "  round_val = %d\n", round_val);
            fflush(fp);

            if (round_val != count) continue;

            pvm_upkint(&int_val, 1, 1);
            for (j=0; (j<ntasks) && (tid[j] != msg_src); j++)
                ;
            fprintf(fp, "  Data from task %d, tid = %x : %d\n",
                j, msg_src, int_val);
            fflush(fp);

            if (sum[j] == -1) {
                sum[j] = int_val; /* store the sum */
                i++;
            }
       } else if (msg_tag == TASKEXIT) {
            /* A task has exited. */
            /* Find out which task has exited. */
            int which_tid, j;
            pvm_upkint(&which_tid, 1, 1);
            for (j=0; (j<ntasks) && (tid[j] != which_tid); j++)
```

```
                ;
               fprintf(fp, "  from tid %x : task %d, tid =  %x, ``
                   ``exited.\n",
                   msg_src, j, which_tid);
               fflush(fp);
               /*
                * If a task exited before sending back the message,
                * create another task to do the same job.
                */
               if (j < ntasks && sum[j] == -1) {
                    /* spawn the j-th task */
                    spawn_a_worker(j, tid, host_tid, fp);

                    /* send unfinished work to the new task */
                    send_data_to_worker(j, tid, num, pertask, fp, count);
               }
           } else if (msg_tag == HOSTDELETE) {
               /*
               * If a host has been deleted, check to see if
               * the tasks running on it has been finished.
               * If not, should create  new worker tasks to do
               * the work on some other  hosts.
               */
               int which_tid, j;

               /* get which host has been suspended/deleted */
               pvm_upkint(&which_tid, 1, 1);

               fprintf(fp, "  from tid %x : %x %s\n", msg_src, which_tid,
                   codes[msg_tag - HOSTDELETE]);
               fflush(fp);

               /*
                * If the task on that host has not finished its
                * work, then create new task to do the work.
                */
               for (j = 0; j < ntasks; j++) {
                    if (host_tid[j] == which_tid && sum[j] == -1) {
                         fprintf(fp, "host_tid[%d] = %x, ``
                           ``need new task\n",
                             j, host_tid[j]);
                         fflush(fp);

                         /* spawn the i-th task, with notifications. */
                         spawn_a_worker(j, tid, host_tid, fp);

                         /* send the unfinished work to the new task */
                         send_data_to_worker(j,tid,num,pertask,fp,count);
                    }
               }
           } else {
               /* print out some other notifications or messages */
               int which_tid;
               pvm_upkint(&which_tid, 1, 1);

               fprintf(fp, "  from tid %x : %x %s\n", msg_src,
                       which_tid,   codes[msg_tag - HOSTDELETE]);
               fflush(fp);
           }
       }

       /* add up the sum */
       for (i=0; i<ntasks; i++)
          result += sum[i];

       fprintf(fp, "Sum from  0 to %d is %d\n", n-1 , result);
       fflush(fp);

       /* check correctness */
```



```
          if (result == (n-1)*n/2) {
             correct++;
             fprintf(fp, "*** Result Correct! ***\n");
          } else {
             wrong++;
             fprintf(fp, "*** Result WRONG! ***\n");
          }

          fflush(fp);
          count--;
      }

     fprintf(fp, "correct = %d; wrong = %d\n", correct, wrong);
     fflush(fp);

     pvm_exit();
     exit(0);
}
```

2.9.6 Sample PVM submit file

Like submitting jobs in any other universe, to submit a PVM job, the user needs to specify the requirements and options in the submit-desciption file and run *condor_submit*. Figure 2.9.6 on page 46 is an example of a submit-description file for a PVM job. This job has a master PVM program called `master_pvm`.

In this sample submit file, the command `universe = PVM` specifies that the jobs should be submitted into PVM universe.

The command `executable = master_pvm` tells Condor that the PVM master program is `master_sum`. This program will be started on the submitting machine. The workers should be spawned by this master program during execution.

This submit file also tells Condor that the PVM virtual machine is consisted of two different classes of machine architectures. Class 0 contains machines with INTEL architecture running SOLARIS251; class 1 contains machines with SUN4x (SPARC) architecture running SOLARIS251.

By using `machine_count = <min>..<max>`, the submit file tells Condor that before the PVM program, there should be at least `<min>` number of machines of the current class. It also asks Condor to give it as many as `<max>` machines. During the execution of the program, the application can get more machines of each of the class by calling `pvm_addhosts()` with a string specifying the desired architecture class. (See the sample program in this section for details.)

The `queue` command should be inserted after the specifications of each class.

2.10 More about how Condor vacates a job

When Condor needs to vacate a job from a machine for whatever reason, it sends the job an asynchronous signal specified in the "KillSig" attribute of the job's classad. The value of this attribute can be specified by the user at submit time by placing the *kill_sig* command in the *condor_submit* submit-description command file.

If a program wanted to do some special work each time Condor kicks them off a machine, all it would need to do is setup a signal handler for some trappable signal as a "cleanup" signal. When submitting this job, specify this cleanup signal to use with *kill_sig*. However, whatever cleanup work the job does had better be quick — if the job takes too long to go away after Condor tells it to do so, Condor follows up with a SIGKILL signal which immediatly terminates the process.

A job that linked with the Condor libraries via the *condor_compile* command and subsequently submitted into the Standard Universe will checkpoint and exit upon receit of a SIGTSTP signal. Thus, SIGTSTP is the default value for KillSig when submitting into the Standard Universe. However, the user's code can checkpoint itself at any time by calling one of the following functions exported by the Condor libraries:

ckpt() Will perform a checkpoint and then return

ckpt_and_exit() Will checkpoint and exit; Condor will then restart the process again later, potentially on a different machine

For jobs submitted into the Vanilla Universe, the default value for KillSig is SIGTERM, which is the usual method to nicely terminate a program in Unix.

2.11 Special Environment Considerations

2.11.1 AFS

The Condor daemons do not run authenticated to AFS; they do not possess an AFS token, and therefore no child process of Condor will be AFS authenticated either. This means that you must set file permissions so that your job can access any necessary files residing on an AFS volume during its run without relying on having your AFS permissions.

So, if a job you submit to Condor needs to access files residing in AFS, you have the following choices:

1. Copy the files needed off of AFS to either a local hard disk where Condor can access them via remote system calls (if this is a Standard Universe job), or copy them to an NFS volume.

2. If you must keep the files on AFS, then you need to set a host ACL (using the AFS "fs setacl" command) on the subdirectory which will serve as the current working directory for the job. If the job is a Standard Universe job, then the host ACL needs to give read/write permission to any process on the submit machine. If the job is a Vanilla Universe job, then you need to set the ACL such that any host in the pool can access the files without being authenticated. If you do not know how to use an AFS host ACL, please ask whomever at your site is responsible for the AFS configuration.

How Condor deals with AFS authentication is something the Condor Team hopes to improve in a subsequent release.

Please also see section 3.9.1 on page 132 in the Administrators Manual for more discussion about this problem.

2.11.2 NFS Automounter

If your current working directory when you run *condor_submit* is accessed via an NFS automounter, Condor may have problems if the automounter later decides to unmount the volume before your job has completed. This is because *condor_submit* likely has stored the dynamic mount point as the job's initial current working directory, and this mount point could become automatically unmounted by the automounter.

There is a simple workaround: When submitting your job, use the *initialdir* command in your submit-description file to point to the stable access point. For example, say the NFS automounter is configured to mount a volume at mount point /a/myserver.company.com/vol1/johndoe whenever the directory /home/johndoe is accessed. In this example case, simply add the following line to your *condor_submit* submit-description file:

```
initialdir = /home/johndoe
```

2.11.3 Condor Daemons running as Non-root

Condor is normally installed such that the Condor daemons have root permission. This allows Condor to run the condor_shadow process and your job process with your UID and file access rights. When Condor is started as root, your Condor jobs can access whatever files you can.

However, it is possible that whomever installed Condor decided not to run the daemons as root, or did not have root access. That's a shame, since Condor is really designed to be run as root. To see if Condor is running as root on a given machine, enter the following command:

```
condor_status -master -l <machine-name>
```

where **machine-name** is the name of the machine you want to inspect. This command will display a condor_master ClassAd; if the attribute "RealUid" equals zero, then the Condor daemons are indeed running with root access and you can skip this section. If the "RealUid" attribute is not zero, then the Condor daemons do not have root access, and you should read on.

Please realize that using "ps" is not an effective method of determining if Condor is running with root access. When using the "ps" command, it may often appear that the daemons are running as the condor user instead of root. However, note that the "ps" command shows the current *effective* owner of the process, not the *real* owner. (See the *getuid*(2) and *geteuid*(2) Unix man pages for details.) In Unix, a process running under the real uid of root may switch its effective uid. (See the *seteuid*(2) man page.) For security reasons, the daemons only set effective uid to root when absolutely necessary (to perform a privileged operation).

If they are not running with root access, you need to make any/all files and/or directories that your job will touch readable and/or writable by the UID (user id) specified by the RealUid attribute. Often this may mean doing a "chmod 777" on the directory where you submit your Condor job.

2.12 Potential Problems

2.12.1 Renaming of argv[0]

When Condor starts up your job, it renames argv[0] (which usually contains the name of the program) to "condor_exec". This is conveinent when examining a machine's processes with "ps"; the process is easily identified as a Condor job.

Unfortunately, some programs read argv[0] expecting their own program name and get confused if they find something unexpected like condor_exec.


```
###########################################################
# sample_submit
# Sample submit file for PVM jobs.
###########################################################

# The job is a PVM universe job.
universe = PVM

# The executable of the master PVM program is ``master_pvm''.
executable = master_pvm

In = "in_sum"
Out = "stdout_sum"
Err = "err_sum"

###################  Architecture class 0  #################

Requirements = (Arch == "INTEL") && (OpSys == "SOLARIS251")

# We want at least 2 machines in class 0 before starting the
# program.  We can use up to 4 machines.
machine_count = 2..4
queue

###################  Architecture class 1  #################

Requirements = (Arch == "SUN4x") && (OpSys == "SOLARIS251")

# We need at least 1 machine in class 1 before starting the
# executable.  We can use up to 3 to start with.
machine_count = 1..3
queue

###############################################################
# note: the program will not be started until the least
#       requirements in all classes are satisfied.
###############################################################
```

Figure 2.2: A sample submit file for PVM jobs.

CHAPTER

THREE

Administrators' Manual

3.1 Introduction

This is the Condor Administrator's Manual. Its purpose is to aid in the installation and administration of a Condor pool. For help on using Condor, see the Condor User's Manual.

A Condor pool is comprised of a single machine which serves as the *Central Manager*, and an arbitrary number of other machines that have joined the pool. Conceptually, the pool is a collection of resources (machines) and resource requests (jobs). The role of Condor is to match waiting requests with available resources. Every part of Condor sends periodic updates to the Central Manager, the centralized repository of information about the state of the pool. Periodically, the Central Manager assesses the current state of the pool and tries to match pending requests with the appropriate resources.

Each resource has an owner, the user who works at the machine. This person has absolute power over their own resource and Condor goes out of its way to minimize the impact on this owner caused by Condor. It is up to the resource owner to define the policy of when requests are serviced and when they are denied on their resource.

On the other hand, each resource request has an owner as well, the user who submitted the job. These people want Condor to provide as many CPU cycles as possible for their work. Often the interests of the resource owners are in conflict with the interests of the resource requesters.

The job of the Condor administrator is to configure the Condor pool to find the happy medium that keeps both resource owners and the users of the pool satisfied. The purpose of this manual is to help you understand the mechanisms that Condor provides to enable you to find this happy medium for your particular set of users and resource owners.

3.1.1 The Different Roles a Machine Can Play

Every machine in a Condor pool can serve a variety of roles. Most machines serve more than one role simultaneously. Certain roles can only be performed by single machines in your pool. The following list describes what these roles are and what resources are required on the machine that is providing that service:

Central Manager There can be only one Central Manager for your pool. The machine is the collector of information, and the negotiator between resources and resource requests. These two halves of the Central Manager's responsibility are performed by separate daemons, so it would be possible to have different machines providing those two services. However, normally they both live on the same machine. This machine plays a very important part in the Condor pool and should be reliable. If this machine crashes, no further matchmaking can be performed within the Condor system (although all current matches remain in effect until they are broken by either party involved in the match). Therefore, you should choose a machine that is likely to be online all the time, or at least one that will be rebooted quickly if something goes wrong, as your central manager. In addition, this machine would ideally have a good network connection to all the machines in your pool since they all send updates over the network to the Central Manager, and all queries must go to the Central Manager.

Execute Any machine in your pool (including your Central Manager) can be configured for whether or not it should execute Condor jobs. Obviously, some of your machines will have to serve this function or your pool won't be very useful. Being an execute machine doesn't require many resources at all. About the only resource that might matter is disk space, since if the remote job dumps core, that file is first dumped to the local disk of the execute machine before being sent back to the submit machine for the owner of the job. However, if there isn't much disk space, Condor will simply limit the size of the core file that a remote job will drop. In general the more resources a machine has (swap space, real memory, CPU speed, etc.) the larger the resource requests it can serve. However, if there are requests that don't require many resources, any machine in your pool could serve them.

Submit Any machine in your pool (including your Central Manager) can be configured for whether or not it should allow Condor jobs to be submitted. The resource requirements for a submit machine are actually much greater than the resource requirements for an execute machine. First of all, every job that you submit that is currently running on a remote machine generates another process on your submit machine. So, if you have lots of jobs running, you will need a fair amount of swap space and/or real memory. In addition all the checkpoint files from your jobs are stored on the local disk of the machine you submit from. Therefore, if your jobs have a large memory image and you submit a lot of them, you will need a lot of disk space to hold these files. This disk space requirement can be somewhat alleviated with a checkpoint server (described below), however the binaries of the jobs you submit are still stored on the submit machine.

Checkpoint Server One machine in your pool can be configured as a checkpoint server. This is optional, and is not part of the standard Condor binary distribution. The checkpoint server is a centralized machine that stores all the checkpoint files for the jobs submitted in your pool. This machine should have lots of disk space and a good network connection to the rest of your pool, as the traffic can be quite heavy.

Now that you know the various roles a machine can play in a Condor pool, we will describe the actual daemons within Condor that implement these functions.

3.1.2 The Condor Daemons

The following list describes all the daemons and programs that could be started under Condor and what they do:

condor_master This daemon is responsible for keeping all the rest of the Condor daemons running on each machine in your pool. It spawns the other daemons, and periodically checks to see if there are new binaries installed for any of them. If there are, the master will restart the affected daemons. In addition, if any daemon crashes, the master will send email to the Condor Administrator of your pool and restart the daemon. The *condor_master* also supports various administrative commands that let you start, stop or reconfigure daemons remotely. The *condor_master* will run on every machine in your Condor pool, regardless of what functions each machine are performing.

condor_startd This daemon represents a given resource (namely, a machine capable of running jobs) to the Condor pool. It advertises certain attributes about that resource that are used to match it with pending resource requests. The startd will run on any machine in your pool that you wish to be able to execute jobs. It is responsible for enforcing the policy that resource owners configure which determines under what conditions remote jobs will be started, suspended, resumed, vacated, or killed. When the startd is ready to execute a Condor job, it spawns the *condor_starter*, described below.

condor_starter This program is the entity that actually spawns the remote Condor job on a given machine. It sets up the execution environment and monitors the job once it is running. When a job completes, the starter notices this, sends back any status information to the submitting machine, and exits.

condor_schedd This daemon represents resources requests to the Condor pool. Any machine that you wish to allow users to submit jobs from needs to have a *condor_schedd* running. When users submit jobs, they go to the schedd, where they are stored in the *job queue*, which the schedd manages. Various tools to view and manipulate the job queue (such as *condor_submit*, *condor_q*, or *condor_rm*) all must connect to the schedd to do their work. If the schedd is down on a given machine, none of these commands will work.

The schedd advertises the number of waiting jobs in its job queue and is responsible for claiming available resources to serve those requests. Once a schedd has been matched with a given resource, the schedd spawns a *condor_shadow* (described below) to serve that particular request.

condor_shadow This program runs on the machine where a given request was submitted and acts as the resource manager for the request. Jobs that are linked for Condor's Standard Universe, which perform remote system calls, do so via the *condor_shadow*. Any system call performed on the remote execute machine is sent over the network, back to the *condor_shadow* which actually performs the system call (such as file I/O) on the submit machine, and the result is sent back over the network to the remote job. In addition, the shadow is responsible for making decisions about the request (such as where checkpoint files should be stored, how certain files should be accessed, etc).

condor_collector This daemon is responsible for collecting all the information about the status of a Condor pool. All other daemons (except the negotiator) periodically send ClassAd updates to the collector. These ClassAds contain all the information about the state of the daemons, the resources they represent or resource requests in the pool (such as jobs that have been submitted to a given schedd). The *condor_status* command can be used to query the collector for specific information about various parts of Condor. In addition, the Condor daemons themselves query the collector for important information, such as what address to use for sending commands to a remote machine.

condor_negotiator This daemon is responsible for all the match-making within the Condor system. Periodically, the negotiator begins a *negotiation cycle*, where it queries the collector for the current state of all the resources in the pool. It contacts each schedd that has waiting resource requests in priority order, and tries to match available resources with those requests. The negotiator is responsible for enforcing user priorities in the system, where the more resources a given user has claimed, the less priority they have to acquire more resources. If a user with a better priority has jobs that are waiting to run, and resources are claimed by a user with a worse priority, the negotiator can preempt that resource and match it with the user with better priority.

NOTE: A higher numerical value of the user priority in Condor translate into worse priority for that user. The best priority you can have is 0.5, the lowest numerical value, and your priority gets worse as this number grows.

condor_kbdd This daemon is only needed on Digital Unix and IRIX. On these platforms, the *condor_startd* cannot determine console (keyboard or mouse) activity directly from the system. The *condor_kbdd* connects to the X Server and periodically checks to see if there has been any activity. If there has, the kbdd sends a command to the startd. That way, the startd knows the machine owner is using the machine again and can perform whatever actions are necessary, given the policy it has been configured to enforce.

condor_ckpt_server This is the checkpoint server. It services requests to store and retrieve checkpoint files. If your pool is configured to use a checkpoint server but that machine (or the server itself is down) Condor will revert to sending the checkpoint files for a given job back to the submit machine.

See figure 3.1 for a graphical representation of the pool architecture.

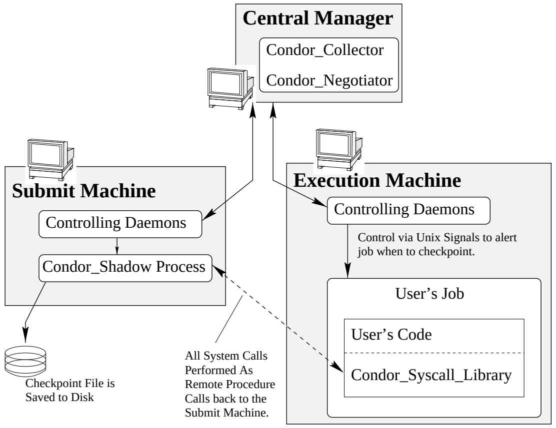

Figure 3.1: Pool Architecture

3.2 Installation of Condor

This section contains the instructions for installing Condor at your site. Condor's installation will setup a default configuration which you can then learn how to customize in the sections which follow.

Please read the the copyright and disclaimer information in section on page vi of the manual, or in the file LICENSE.TXT, before proceeding. Installation and use of Condor is acknowledgement that you have read and agreed to these terms.

The Condor binary distribution is packaged in the following 5 files and 2 directories:

DOC file containing directions for where to find the Condor documentation

INSTALL these installation directions

LICENSE.TXT by installing Condor, you agree to the contents of the LICENSE.TXT file

README general info

condor_install Perl script to install and configure Condor

examples directory containing C, Fortran and C++ example programs to run with Condor

release.tar tar file of the 'release directory', which contains the Condor binaries and libraries

Before you install, please consider joining the condor-world mailing list. Traffic on this list is kept to an absolute minimum. It is only used to announce new releases of Condor. To subscribe, send a message to majordomo@cs.wisc.edu with the body:

```
subscribe condor-world
```

3.2.1 Preparing to Install Condor

Before you install Condor at your site, there are a few important decisions you must make about the basic layout of your pool. These are:

1. What machine will be the Central Manager?
2. Will Condor run as root or not?
3. Who will be administering Condor on the machines in your pool?
4. Will you have a 'condor' user and will it's home directory be shared?
5. Where should the machine-specific directories for Condor go?
6. Where should the parts of the Condor system be installed?
   - Config Files
   - Release directory
     - User Binaries
     - System Binaries
     - Lib Directory
     - Etc Directory
   - Documentation
7. Am I using AFS?
8. Do I have enough disk space for Condor?

If you feel you already know the answers to these questions, you can skip to the 'Installation Procedure' section below, section 3.2.2. If you are unsure about any of them, read on.

What machine will be the Central Manager?

One machine in your pool must be the Central Manager. You should setup and install Condor on this machine first. This is the centralized information repository for the Condor pool and is also the machine that does match-making between available machines and waiting jobs. If the Central Manager machine crashes, any currently active matches in the system will keep running, but no new matches will be made. Moreover, most Condor tools will stop working. Because of the importance of this machine for the proper functioning of Condor, we recommend you install it on a machine that is likely to stay up all the time, or at the very least, one that will be rebooted quickly if it does crash. Also, because all the daemons will send updates (by default every 5 minutes) to this machine, it is advisable to consider network traffic and your network layout when choosing your central manager.

Will Condor run as root or not?

We strongly recommend that you start up the Condor daemons as root. Otherwise, Condor can do very little to enforce security and policy decisions. If you don't have root access and would like to install Condor, under most platforms you can run Condor under any user you'd like. However, there are serious security consequences of this. Please see section 3.10.1 on page 139 in the manual for details on running Condor as non-root.

Who will be administering Condor on the machines in your pool?

Either root will be administering Condor directly, or someone else would be acting as the Condor administrator. If root has delegated the responsibility to another person but doesn't want to grant that person root access, root can specify a condor_config.root file that will override settings in the other condor config files. This way, the global condor_config file can be owned and controlled by whoever is condor-admin, and the condor_config.root can be owned and controlled only by root. Settings that would compromise root security (such as which binaries are started as root) can be specified in the condor_config.root file while other settings that only control policy or condor-specific settings can still be controlled without root access.

Will you have a 'condor' user and will it's home directory be shared?

To simplify installation of Condor at your site, we recommend that you create a 'condor' user on all machines in your pool. The condor daemons will create files (such as the log files) owned by this user, and the home directory can be used to specify the location of files and directories needed by Condor. The home directory of this user can either be shared among all machines in your pool, or could be a separate home directory on the local partition of each machine. Both approaches have advantages and disadvantages. Having the directories centralized can make administration easier, but also concentrates the resource usage such

that you potentially need a lot of space for a single shared home directory. See the section below on machine-specific directories for more details.

If you choose not to create a condor user, you must specify via the CONDOR_IDS environment variable which uid.gid pair should be used for the ownership of various Condor files. See section 3.10.2 on "UIDs in Condor" on page 140 in the Administrator's Manual for details.

Where should the machine-specific directories for Condor go?

Condor needs a few directories that are unique on every machine in your pool. These are 'spool', 'log', and 'execute'. Generally, all three are subdirectories of a single machine specific directory called the 'local directory' (specified by the `LOCAL_DIR` parameter in the config file).

If you have a 'condor' user with a local home directory on each machine, the `LOCAL_DIR` could just be user condor's home directory ('`LOCAL_DIR = $(TILDE)`' in the config file). If this user's home directory is shared among all machines in your pool, you would want to create a directory for each host (named by hostname) for the local directory ('`LOCAL_DIR = $(TILDE)/hosts/$(HOSTNAME)`' for example). If you don't have a condor account on your machines, you can put these directories wherever you'd like. However, where to place them will require some thought, as each one has its own resource needs:

execute This is the directory that acts as the current working directory for any Condor jobs that run on a given execute machine. The binary for the remote job is copied into this directory, so you must have enough space for that. (Condor won't send a job to a machine that doesn't have enough disk space to hold the initial binary). In addition, if the remote job dumps core for some reason, it is first dumped to the execute directory before it is sent back to the submit machine. So, you will want to put the execute directory on a partition with enough space to hold a possible core file from the jobs submitted to your pool.

spool The spool directory holds the job queue and history files, and the checkpoint files for all jobs submitted from a given machine. As a result, disk space requirements for spool can be quite large, particularly if users are submitting jobs with very large executables or image sizes. By using a checkpoint server (see section 3.3.5 on "Installing a Checkpoint Server" on page 71 for details), you can ease the disk space requirements, since all checkpoint files are stored on the server instead of the spool directories for each machine. However, the initial checkpoint files (the executables for all the clusters you submit) are still stored in the spool directory, so you will need some space, even with a checkpoint server.

log Each Condor daemon writes its own log file which is placed in the log directory. You can specify what size you want these files to grow to before they are rotated, so the disk space requirements of the log directory are configurable. The larger the logs, the more historical information they will hold if there's a problem, but the more disk space they use up. If you have a network filesystem installed at your pool, you might want to place the log directories in a shared location (such as `/usr/local/condor/logs/$(HOSTNAME)`) so that you can view the log files from all your machines in a single location. However, if you take this approach, you will have to specify a local partition for the lock directory (see below).

lock Condor uses a small number of lock files to synchronize access to some files that are shared between multiple daemons. Because of problems we've had with file locking and network filesystems (particularly NFS), these lock files should be placed on a local partition on each machine. By default, they are just placed in the log directory. If you place your log directory on a network filesystem partition, you should specify a local partition for the lock files with the 'LOCK' parameter in the config file (such as `/var/lock/condor`).

Generally speaking, we recommend that you do not put these directories (except lock) on the same partition as `/var`, since if the partition fills up, you will fill up `/var` as well, which will cause lots of problems for your machines. Ideally, you'd have a separate partition for the Condor directories that the only consequence of filling up would be Condor's malfunction, not your whole machine.

Where should the parts of the Condor system be installed?

- Config Files
- Release directory
  - User Binaries
  - System Binaries
  - Lib Directory
  - Etc Directory
- Documentation

Config Files There are a number of config files that allow you different levels of control over how Condor is configured at each machine in your pool. In general, you will have 1 global configuration file for each platform. In addition, there is a local config file for each machine, where you can override settings in the global file. This allows you to have different daemons running, different policies for when to start and stop Condor jobs, and so on. Beginning with Condor version 6.0.1, you can use a single config file which is shared among all platforms in your pool, and have both platform-specific and machine-specific files. See section 3.9.2 on page 133 about "Configuring Condor for Multiple Platforms" for details.

In addition, because we recommend that you start the Condor daemons as root, we allow you to create config files that are owned and controlled by root that will override any other condor settings. This way, if the condor administrator isn't root, the regular

condor config files can be owned and writable by condor-admin, but root doesn't have to grant root access to this person. See section 3.10.3 on page 140 in the manual for a detailed discussion of the root config files, if you should use them, and what settings should be in them.

In general, there are a number of places that condor will look to find its config files. The first file it looks for is the global config file. These locations are searched in order until a config file is found. If none contain a valid config file, Condor will print an error message and exit:

1. File specified in `CONDOR_CONFIG` environment variable
2. `/etc/condor/condor_config`
3. `~condor/condor_config`

Next, Condor tries to load the machine-specific, or local config file. The only way to specify the local config file is in the global config file, with the `LOCAL_CONFIG_FILE` macro. If that macro isn't set, no local config file is used. Beginning with Condor version 6.0.1, this macro can be a list of files instead of a single file.

The root config files come in last. The global file is searched for in the following places:

1. `/etc/condor/condor_config.root`
2. `~condor/condor_config.root`

The local root config file is found with the `LOCAL_ROOT_CONFIG_FILE` macro. If that isn't set, no local root config file is used. Beginning with Condor version 6.0.1, this macro can also be a list of files instead of a single file.

Release Directory Every binary distribution contains a 'release.tar' file that contains four subdirectories: 'bin', 'etc', 'lib' and 'sbin'. Wherever you choose to install these 4 directories we call the 'release directory' (specified by the 'RELEASE_DIR' parameter in the config file). Each release directory contains platform dependent binaries and libraries, so you will need to install a separate one for each kind of machine in your pool.

- User Binaries:
  All of the files in the 'bin' directory are programs the end Condor users should expect to have in their path. You could either put them in a well known location (such as `/usr/local/condor/bin`) which you have Condor users add to their `PATH` environment variable, or copy those files directly into a well known place already in user's PATHs (such as `/usr/local/bin`). With the above examples, you could also leave the binaries in `/usr/local/condor/bin` and put in soft links from `/usr/local/bin` to point to each program.
- System Binaries:
  All of the files in the 'sbin' directory are Condor daemons and agents, or programs that only the Condor administrator would need to run. Therefore, we recommend that you only add these programs to the `PATH` of the Condor administrator.

- Lib Directory:

  The files in the 'lib' directory are the condor libraries that must be linked in with user jobs for all of Condor's checkpointing and migration features to be used. 'lib' also contains scripts used by the *condor_compile* program to help relink jobs with the condor libraries. These files should be placed in a location that is world-readable, but they do not need to be placed in anyone's PATH. The *condor_compile* script checks the config file for the location of the lib directory.

- Etc Directory:

  'etc' contains an 'examples' subdirectory which holds various example config files and other files used for installing Condor. 'etc' is the recommended location to keep the master copy of your config files. You can put in soft links from one of the places mentioned above that Condor checks automatically to find it's global config file.

Documentation The documentation provided with Condor is currently only available in HTML, Postscript and PDF (Adobe Acrobat). It can be locally installed wherever is customary at your site. You can also find the Condor documentation on the web at: http://www.cs.wisc.edu/condor/manual.

Am I using AFS?

If you are using AFS at your site, be sure to read the section 3.9.1 on page 131 in the manual. Condor does not currently have a way to authenticate itself to AFS. We're working on a solution, it's just not ready for version 6.0. So, what this means is that you are probably not going to want to have the `LOCAL_DIR` for Condor on AFS. However, you can (and probably should) have the Condor `RELEASE_DIR` on AFS, so that you can share one copy of those files and upgrade them in a centralized location. You will also have to do something special if you submit jobs to Condor from a directory on AFS. Again, read manual section 3.9.1 for all the gory details.

Do I have enough disk space for Condor?

The Condor release directory takes up a fair amount of space. This is another reason why it's a good idea to have it on a shared filesystem. The rough size requirements for the release directory on various platforms are listed in table 3.1.

In addition, you will need a lot of disk space in the local directory of any machines that are submitting jobs to Condor. See question 5 above for details on this.

Platform	*Size*
Intel/Linux	11 megs (statically linked)
Intel/Linux	6.5 megs (dynamically linked)
Intel/Solaris	8 megs
Sparc/Solaris	10 megs
SGI/IRIX	17.5 megs
Alpha/Digital Unix	15.5 megs

Table 3.1: Release Directory Size Requirements

3.2.2 Installation Procedure

IF YOU HAVE DECIDED TO CREATE A 'condor' USER AND GROUP, YOU SHOULD DO THAT ON ALL YOUR MACHINES BEFORE YOU DO ANYTHING ELSE.

The easiest way to install Condor is to use one or both of the scripts provided to help you: *condor_install* and *condor_init*. You should run these scripts as the user that you are going to run the Condor daemons as. First, run *condor_install* on the machine that will be a fileserver for shared files used by Condor, such as the release directory, and possibly the condor user's home directory. When you do, choose the "full-install" option in step #1 described below.

Once you have run *condor_install* on a file server to setup your release directory and configure Condor for your site, you should run *condor_init* on any other machines in your pool to create any locally used files that aren't created by *condor_install*. In the most simple case, where nearly all of Condor is installed on a shared file system, even though Condorinstall will create nearly all the files and directories you need, you will still need to use *condor_init* to create the `LOCK` directory on the local disk of each machine. If you have a shared release directory, but the `LOCAL_DIR` is local on each machine, *condor_init* will create all the directories and files needed in `LOCAL_DIR`. In addition, *condor_init* will create any soft links on each machine that are needed so that Condor can find its global config file.

If you don't have a shared filesystem, you will need to run *condor_install* on each machine in your pool to setup Condor. In this case, there is no need to run *condor_init* at all.

In addition, you will want to run *condor_install* on your central manager machine if that machine is different from your file server, using the "central-manager" option in step #1 described below. Run *condor_install* on your file server first, then on your central manager. If this step fails for some reason (NFS permissions, etc), you can do it manually quite easily. All this does is copy the `condor_config.local.central.manager` file from <release_dir>/etc/examples to the proper location for the local config file of your central manager machine. If your central manager is an Alpha or an SGI, you might want to add "KBDD" to the `DAEMON_LIST` parameter. See section 3.4 "Configuring Condor" on page 74 of the manual for details.

condor_install assumes you have perl installed in `/usr/bin/perl`. If this is not the case, you can either edit the script to put the right path in, or you will have to invoke perl directly from your shell (assuming perl is in your PATH):

```
% perl condor_install
```

condor_install breaks down the installation procedure into various steps. Each step is clearly numbered. The following section explains what each step is for, and suggests how to answer the questions *condor_install* will ask you for each one.

condor_install, step-by-step

STEP 1: What type of Condor installation do you want? There are three types of Condor installation you might choose: 'submit-only', 'full-install', and 'central-manager'. A submit-only machine can submit jobs to a Condor pool, but Condor jobs will not run on it. A full-install machine can both submit and run Condor jobs.

If you are planning to run Condor jobs on your machines, you should either install and run Condor as root, or as user 'condor'.

If you are planning to setup a submit only machine, you can either install Condor machine-wide as root or user 'condor', or, you can install Condor as yourself into your home directory.

The other possible installation type is setting up a machine as a central manager. If you do a full-install and you say that you want the local host to be your central manager, this step will be done automatically. You should only choose the central-manager option at step 1 if you have already run *condor_install* on your file server and you now want to run *condor_install* on a different machine that will be your central manager.

STEP 2: How many machines are you setting up this way? If you are installing Condor for multiple machines, and you have a shared file system, *condor_install* will prompt you for the hostnames of each machine you want to add to your Condor pool. If you don't have a shared file system, you will have to run *condor_install* locally on each machine, anyway, so it doesn't bother asking you for the names. If you provide a list, it will use the names to automatically create directories and files later. At the end, *condor_install* will dump out this list to a 'roster' file which can be used by scripts to help maintain your Condor pool.

If you are only installing Condor on 1 machine, you would just answer 'no' to the first question, and move on.

STEP 3: Install the Condor release directory The release directory contains four subdirectories: 'bin', 'etc', 'lib' and 'sbin'. bin contains user-level executable programs. etc is the recommended location for your Condor config files, and also includes an 'examples' directory with default config files and other default files used for installing condor. lib contains libraries to link condor user programs and scripts used by the Condor system. sbin contains all administrative executable programs and the Condor daemons.

If you have multiple machines with a shared filesystem that will be running Condor, you should put the release directory on that shared filesystem so you only have one copy of all the binaries, and so that when you update them, you can do so in one place. Note that the release directory is architecture dependent, so you will need to download separate binary distributions for every platform in your pool.

condor_install tries to find an already installed release directory. If it can't find one, it asks if you have installed one already. If you have not installed one, it tries to do so for you by untarring the release.tar file from the binary distribution.

NOTE: If you are only setting up a central manager (you chose 'central manager' in step 1) step 3 is the last question you will need to answer.

STEP 4: How and where should Condor send email if things go wrong? Various parts of Condor will send email to a condor administrator if something goes wrong that needs human attention. You will need to specify the email address of this administrator.

You will also need to specify the full path to a mail program that Condor will use to send the email. This program needs to understand the '-s' option, which is how you specify a subject for the outgoing message. The default on most platforms will probably be correct. On Linux machines, since there is such variation in Linux distributions and installations, you should verify that the default works. If the script complains that it cannot find the mail program that was specified, you can try 'which mail' from your shell prompt to see what 'mail' program is currently in your PATH. If there is none, try 'which mailx'. If you still can't find anything, ask your system administrator. You should verify that the program you end up using supports '-s'. The man page for that program will probably tell you.

STEP 5: Where should public programs be installed? It is recommended that you install the user-level condor programs in the release directory, (where they go by default). This way, when you want to install a new version of the Condor binaries, you can just replace your release directory and everything will be updated at once. So, one option is to have Condor users add `<release_dir>/bin` to their PATH, so that they can access the programs. However, we recommend putting in soft links from some directory already in their PATH (such as `/usr/local/bin`) that point back to the Condor user programs. *condor_install* will do this for you, all you have to do is tell it what directory to put these links into. This way, users don't have to change their PATH to use Condor but you can still have the binaries installed in their own location.

If you are installing Condor as neither root nor condor, there is a perl script wrapper to all the Condor tools that is created which sets some appropriate environment variables and automatically passes certain options to the tools. This is all created automatically by *condor_install*. So, you need to tell *condor_install* where to put this perl script. The script itself is linked to itself with many different names, since it is the name that determines the behavior of the script. This script should go somewhere that is in your PATH already, if possible (such as ~bin).

At this point, the remaining steps are different depending on what kind of installation you are doing. Skip to the appropriate section depending on what kind of installation you selected in STEP 1 above.

Full Install

STEP 6: What machine will be your central manager? Simply type in the full hostname of the machine you have chosen for your central manager. If *condor_install* can't find information about the host you typed by querying your nameserver, it will print out an error message and ask you to confirm.

STEP 7: Where will the 'local directory' go? This is the directory discussed in question #5 from the introduction. *condor_install* tries to make some educated guesses as to what directory you want to use for the purpose. Simply agree to the correct guess, or (when *condor_install* has run out of guesses) type in what you want. Since this directory needs to be unique, it is common to use the hostname of each machine in its name. When typing in your own path, you can use '`$(HOSTNAME)`' which *condor_install* (and the Condor config files) will expand to the hostname of the machine you are currently on. *condor_install* will try to create the corresponding directories for all the machines you told it about in STEP 2 above.

Once you have selected the local directory, *condor_install* creates all the needed subdirectories of each one with the proper permissions. They should have the following permissions and ownerships:

```
drwxr-xr-x   2 condor   root         1024 Mar  6 01:30 execute/
drwxr-xr-x   2 condor   root         1024 Mar  6 01:30 log/
drwxr-xr-x   2 condor   root         1024 Mar  6 01:30 spool/
```

If your local directory is on a shared file system, *condor_install* will prompt you for the location of your lock files, as discussed in question #5 above. In this case, when *condor_install* is finished, you will have to run *condor_init* on each machine in your pool to create the lock directory before you can start up Condor.

STEP 8: Where will the local (machine-specific) config files go? As discussed in question #6 above, there are a few different levels of Condor config file. There's the global config file that will be installed in `<release_dir>/etc/condor_config`, and there are machine-specific, or local config files that override the settings in the global file. If you are installing on multiple machines or are configuring your central manager machine, you must select a location for your local config files.

The two main options are to have a single directory that holds all the local config files, each one named '`$(HOSTNAME)`.local', or to have the local config files go into the individual local directories for each machine. Given a shared filesystem, we recommend the first option, since it makes it easier to configure your pool from a centralized location.

STEP 9: How do you want Condor to find its config file? Since there are a few known places Condor looks to find your config file, we recommend that you put a soft link from one of them to point to `<release_dir>/etc/condor_config`. This way, you can keep your Condor configuration in a centralized location, but all the Condor daemons and tools will be able to find their config files. Alternatively, you can set the CONDOR_CONFIG environment variable to contain `<release_dir>/etc/condor_config`.

condor_install will ask you if you want to create a soft link from either of the two fixed locations that Condor searches.

Once you have completed STEP 9, you're done. *condor_install* prints out a messages describing what to do next. Please skip to section 3.2.3.

Submit Only

This section has not yet been written

3.2.3 Condor is installed... now what?

Now that Condor has been installed on your machine(s), there are a few things you should check before you start up Condor.

1. Read through the `<release_dir>/etc/condor_config` file. There are a lot of possible settings and you should at least take a look at the first two main sections to make sure everything looks okay. In particular, you might want to setup host/ip based security for Condor. See the section 3.7 on page 122 in the manual to learn how to do this.
2. Condor can monitor the activity of your mouse and keyboard, provided that you tell it where to look. You do this with the `CONSOLE_DEVICES` entry in the condor_startd section of the config file. On most platforms, we provide reasonable defaults. For example, the default device for the mouse on Linux is 'mouse', since most Linux installations have a soft link from '`/dev/mouse`' that points to the right device (such as `tty00` if you have a serial mouse, `psaux` if you have a PS/2 bus mouse, etc). If you don't have a `/dev/mouse` link, you should either create one (you'll be glad you did), or change the `CONSOLE_DEVICES` entry in Condor's config file. This entry is just a comma seperated list, so you can have any devices in `/dev` count as 'console devices' and activity will be reported in the condor_startd's classad as `ConsoleIdleTime`.
3. (Linux only) Condor needs to be able to find the 'utmp' file. According to the Linux File System Standard, this file should be `/var/run/utmp`. If Condor can't find it there, it looks in `/var/adm/utmp`. If it still can't find it, it gives up. So, if your Linux distribution puts this file somewhere else, be sure to put a soft link from `/var/run/utmp` to point to the real location.

3.2.4 Starting up the Condor daemons

To start up the Condor daemons, all you need to do is execute `<release_dir>/sbin/condor_master`. This is the Condor master, whose only job in life is to make sure the other Condor daemons are running. The master keeps track of the daemons, restarts them if they crash, and periodically checks to see if you have installed new binaries (and if so, restarts the affected daemons).

If you're setting up your own pool, you should start Condor on your central manager machine first. If you have done a submit-only installation and are adding machines to an existing pool, it doesn't matter what order to start them in.

To ensure that Condor is running, you can run either:

```
ps -ef | egrep condor_
```

or

```
ps -aux | egrep condor_
```

depending on your flavor of Unix. On your central manager machine you should have processes for:

- condor_master
- condor_collector
- condor_negotiator
- condor_startd
- condor_schedd

On all other machines in your pool you should have processes for:

- condor_master
- condor_startd
- condor_schedd

(**NOTE**: On Alphas and IRIX machines, there will also be a '*condor_kbdd*' – see section 3.9.4 on page 137 of the manual for details.) If you have setup a submit-only machine, you will only see:

- condor_master

- condor_schedd

Once you're sure the Condor daemons are running, check to make sure that they are communicating with each other. You can run *condor_status* to get a one line summary of the status of each machine in your pool.

Once you're sure Condor is working properly, you should add "condor_master" into your startup/bootup scripts (i.e. `/etc/rc`) so that your machine runs *condor_master* upon bootup. *condor_master* will then fire up the neccesary Condor daemons whenever your machine is rebooted.

If your system uses System-V style init scripts, you can look in `<release_dir>/etc/examples/condor.boot` for a script that can be used to start and stop Condor automatically by init. Normally, you would install this script as `/etc/init.d/condor` and put in soft link from various directories (for example, `/etc/rc2.d`) that point back to `/etc/init.d/condor`. The exact location of these scripts and links will vary on different platforms.

If your system uses BSD style boot scripts, you probably have an `/etc/rc.local` file. Just add a line in there to start up `<release_dir>/sbin/condor_master` and you're done.

3.2.5 The Condor daemons are running... now what?

Now that the Condor daemons are running, there are a few things you can and should do:

1. (Optional) Do a full install for the *condor_compile* script. condor_compile assists in linking jobs with the Condor libraries to take advantage of all of Condor's features. As it is currently installed, it will work by placing it in front of any of the following commands that you would normally use to link your code: gcc, g++, g77, cc, acc, c89, CC, f77, fort77 and ld. If you complete the full install, you will be able to use condor_compile with any command whatsoever, in particular, make. See section 3.9.3 on page 136 in the manual for directions.
2. Try building and submitting some test jobs. See `examples/README` for details.
3. If your site uses the AFS network file system, see section 3.9.1 on page 131 in the manual.
4. We strongly recommend that you start up Condor (i.e. run the *condor_master* daemon) as user root. If you must start Condor as some user other than root, see section 3.10.1 on page 139.

3.3 Installing Contrib Modules

This section describes how to install various *contrib modules* in the Condor system. Some of these modules are separate, optional pieces, not included in the main distribution of Condor. For example, the checkpoint server, or DagMan. Others are integral parts of Condor taken from the development series that have certain features users might want to install. For example, the new SMP-aware *condor_startd*, or the CondorView collector. Both of these things come automatically with Condor version 6.1 and greater. However, if you don't want to switch over to using only the development binaries, you can install these seperate modules and maintain most of the stable release at your site.

3.3.1 Installing The SMP-Startd Contrib Module

Basically, the "SMP-Startd Contrib module" is just a selection of a few files needed to run the 6.1 startd in your existing 6.0 pool. For documentation on the new startd or the supporting files, see the version 6.1 manual.

See section 3.2 on page 51 for complete details on how to install Condor. In particular, you should read the first few sections that discuss *release directories*, pool layout, and so on.

To install the SMP-startd from the separate contrib module, you must first download the appropriate binary modules for each of your platforms. Once you uncompress and untar the module, you will have a directory with an `smp_startd.tar` file, a `README`, and so on. The `smp_startd.tar` acts much like the `release.tar` file for a main release. It contains all the binaries and supporting files you would install in your release directory:

```
sbin/condor_startd
sbin/condor_starter
sbin/condor_preen
bin/condor_status
etc/examples/condor_config.local.smp
```

condor_preen and *condor_status* are both fully backwards compatible, so you can use the new version for your entire pool without changing any of your config files. They each have just been enhanced to handle the SMP startd. See the version 6.1 man pages on each for details. The *condor_starter* is also backwards compatible, so you probably want to install it pool-wide, as well.

The SMP startd is backwards compatible, only in the sense that it still runs and works just fine on single-CPU machines. However, it uses different policy expressions to control its policy, so in this (more important) sense, it is not backwards compatible. For this reason, you must have some separate config file settings in effect on machines running the new version. Therefore, you must decide if you want to convert all your machines to the new version, or only convert your SMP machines. If you just convert the SMP machines, you

can put the new settings in the local config file for each SMP machine. If you convert all your machines, you will want to put the new settings into your global config file.

Installing Pool-Wide

Since you are installing new daemon binaries for all hosts in your pool, it's generally a good idea to make sure no jobs are running and all the Condor daemons are shut off before you begin. Please see section 3.8 on page 128 for details on how to do this.

You may want to keep your old binaries around, just to be safe. Simply move the existing *condor_startd*, *condor_starter*, *condor_preen* out of the way (for example, to "condor_startd.old") in the sbin directory, and move *condor_status* out of the way in bin.

You can simply untar the `smp_startd.tar` file into your release directory, and it will install the new versions (and overwrite your existing binaries if you haven't moved them out of the way). Once the new binaries are in place, all you need to do is add the new settings for the SMP startd to your global config file.

Once the binaries and config settings are in place, you can restart your pool, as described in section 3.8.1 on page 131 on "Restarting Your Condor Pool".

Installing Only on SMP Machines

If you only want to run the new startd on your SMP machines, you should untar the `smp_startd.tar` file into some temporary location. Copy the `sbin/condor_startd` file into `<release_dir>/sbin/condor_startd.smp`. You can simply overwrite `<release_dir>/sbin/condor_preen` and `<release_dir>/bin/condor_status` with the new versions. In case you have any currently running condor_starter processes, you should move the existing binary to condor_starter.old with "mv" so that you don't get starters that crash with SIGILL or SIGBUS. Once you have moved the existing starter out of the way, you can install the new version from your scratch directory.

Once you've got all the new binaries installed, all you need to do is edit the local config file for each SMP host in your pool to add the SMP-specific settings described below. In addition, you will need to add the line:

```
STARTD = $(SBIN)/condor_startd.smp
```

to let the *condor_master* know you want the new version spawned on that host.

Once the binaries are all in place and all the configuration settings are done, you can send a *condor_reconfig* command to your SMP hosts (from any machine listed in the `HOST_ALLOW_ADMINISTRATOR` setting in your config files), the *condor_master* should notice the new binaries on the SMP machines, and spawn them.

Notes on SMP Startd configuration

All documentation for the new Startd can be found in the version 6.1 manual. In the `etc/examples/condor_config.local.smp` file, you will see all the new config file settings you must define or change with the new version. Mainly, these are the new policy expressions. Look in the version 6.1 manual, in the "Configuring The Startd Policy" section for complete details on how to configure the policy for the 6.1 startd. In particular, you probably want to read the section titled "Differences from the Version 6.0 Policy Settings" to see how the new policy expressions differ from previous versions. These changes are not SMP-specific, they just make writing more complicated policies much easier. Given the wide range of SMP machines, from dual-CPU desktop workstations, up to giant, 128-node super computers, more flexibility in writing complicated policies is a big help.

In addition to the new policy expressions, there are a few settings that control how the SMP startd's view of the machine state effects each of the virtual machines it is representing. See the section "Configuring The Startd for SMP Machines" for full details on configuring these other settings of the SMP startd.

Finally, on SMP machines, each running node has its own *condor_starter*, and each starter maintains its own log file with a different name. Therefore, you want to list which files *condor_preen* should remove from the `log` directory, instead of having to list the files you want to keep. To do this, you specify a `INVALID_LOG_FILES` setting instead of a `VALID_LOG_FILES` setting. In both install cases, since you are using the new *condor_preen* in your whole pool, you should add the following to your global config file:

```
INVALID_LOG_FILES = core
```

since core files are the only unwanted things that might show up in your `log` directory.

3.3.2 Installing CondorView Contrib Modules

To install CondorView for your pool, you really need two things:

1. The CondorView server, which collects historical information.
2. The CondorView client, a Java applett that views this data.

Since these are totally seperate modules, they will each be handled in their own sections.

3.3.3 Installing the CondorView Server Module

The CondorView server is just an enhanced version of the *condor_collector* which can log information to disk, providing a persistant, historical database of your pool state. This

includes machine state, as well as the state of jobs submitted by users, and so on. This enhanced *condor_collector* is simply the version 6.1 development series, but it can be installed in a 6.0 pool. The historical information logging can be turned on or off, so you can install the CondorView collector without using up disk space for historical information if you don't want it.

To install the CondorView server, you must download the appropriate binary module for whatever platform you are going to run your CondorView server on. This does not have to be the same platform as your exisiting central manager (see below). Once you uncompress and untar the module, you will have a directory with a `view_server.tar` file, a `README`, and so on. The `view_server.tar` acts much like the `release.tar` file for a main release of Condor. It contains all the binaries and supporting files you would install in your release directory:

```
sbin/condor_collector
sbin/condor_stats
etc/examples/condor_config.local.view_server
```

You have two options to choose from when deciding how to install this enhanced *condor_collector* in your pool:

1. Replace your exisiting *condor_collector* and use the new version for both historical information and the regular role the collector plays in your pool.
2. Install the new *condor_collector* and run it on a seperate host from your main *condor_collector* and configure your machines to send updates to both collectors.

If you replace your existing collector with the enhanced version, because it is development code, there might be a bug or problem that would cause problems for your pool. On the other hand, if you install the enhanced version on a seperate host, if there are problems, only CondorView will be affected, not your entire pool. However, installing the CondorView collector on a seperate host generates more network traffic (from all the duplicate updates that are sent from each machine in your pool to both collectors). In addition, the installation procedure to have both collectors running is a more complicated process. You will just have to decide for yourself which solution you feel more comfortable with.

Before we discuss the details of one type of installation or the other, we explain the steps you must take in either case.

Setting up the CondorView Server Module

Before you install the CondorView collector (as described in the following sections), you have to add a few settings to the local config file of that machine to enable historical data collection. These settings are described in detail in the Condor Version 6.1 Administrator's Manual, in the section "condor_collector Config File Entries". However, a short explaination of the ones you must customize is provided below. These entries are also explained in the `etc/examples/condor_config.local.view_server` file, included in the contrib module. You should just insert that file into the local config file for your CondorView collector host and customize as appropriate at your site.

`POOL_HISTORY_DIR` This is the directory where historical data will be stored. There is a configurable limit to the maximum space required for all the files created by the CondorView server (`POOL_HISTORY_MAX_STORAGE`). This directory must be writable by whatever user the CondorView collector is running as (usually "condor").

NOTE: This should be a seperate directory, not the same as either the `Spool` or `Log` directories you have already setup for Condor. There are a few problems putting these files into either of those directories.

`KEEP_POOL_HISTORY` This is a boolean that determines if the CondorView collector should store the historical information. It is false by default, which is why you must specify it as true in your local config file.

Once these settings are in place in the local config file for your CondorView server host, you must to create the directory you specified in `POOL_HISTORY_DIR` and make it writable by whomever your CondorView collector is running as. This would be the same user that owns the `CollectorLog` file in your `Log` directory (usually, "condor").

Once those steps are completed, you are ready to install the new binaries and you will begin collecting historical information. Then, you should install the CondorView client contrib module which contains the tools used to query and display this information.

CondorView Collector as Your Only Collector

To install the new CondorView collector as your main collector, you simply have to replace your existing binary with the new one, found in the `view_server.tar` file. All you need to do is move your existing `condor_collector` binary out of the way with the "mv" command. For example:

```
% cd /full/path/to/your/release/directory
% cd sbin
% mv condor_collector condor_collector.old
```

Then, from that same directory, you just have to untar the `view_server.tar` file, into your release directory, which will install a new `condor_collector` binary, `condor_stats`, a tool that can be used to query this collector for historical information, and an example config file. Within 5 minutes, the *condor_master* will notice the new timestamp on your `condor_collector` binary, shutdown your existing collector, and spawn the new version. You will see messages about this in the log file for your *condor_master* (usually `MasterLog` in your `log` directory). Once the new collector is running, it is safe to remove your old

binary, though you may want to keep it around in case you have problems with the new version and want to revert back.

CondorView Collector in Addition to Your Main Collector

To install the CondorView collector in addition to your regular collector requires a little extra work. First, you should untar the `view_server.tar` file into some temporary location (not your main release directory). Copy the `sbin/condor_collector` file out of there, and into your main release directory's sbin with a new name (such as `condor_collector.view_server`). You will also want to copy the `condor_stats` program to your sbin release directory.

Next, you must configure whatever host is going to run your seperate CondorView server to spawn this new collector in addition to whatever other daemons it's running. You do this by adding "COLLECTOR" to the `DAEMON_LIST` on this machine, and defining what "COLLECTOR" means. For example:

```
DAEMON_LIST = MASTER, STARTD, SCHEDD, COLLECTOR
COLLECTOR = $(SBIN)/condor_collector.view_server
```

For this change to take effect, you must actually re-start the *condor_master* on this host (which you can do with the *condor_restart* command, if you run that command from a machine with "ADMINISTRATOR" access to your pool. (See section 3.7 on page 122 for full details of IP/host-based security in Condor).

Finally, you must tell all the machines in your pool to start sending updates to both collectors. You do this by specifying the following setting in your global config file:

```
CONDOR_VIEW_HOST = full.hostname
```

where "full.hostname" is the full hostname of the machine where you are running your CondorView collector.

Once this setting is in place, you must send a *condor_reconfig* to your entire pool. The easiest way to do this is:

```
% condor_reconfig `condor_status -master`
```

Again, this command must be run from a trusted "administrator" machine for it to work.

3.3.4 Installing the CondorView Client Contrib Module

This section has not yet been written

3.3.5 Installing a Checkpoint Server

The Checkpoint Server is a daemon that can be installed on a server to handle all of the checkpoints that a Condor pool will create. This machine should have a large amount of disk space available, and should have a fast connection to your machines.

NOTE: It is a good idea to pick a very stable machine for your checkpoint server. If the checkpoint server crashes, the Condor system will continue to operate, though poorly. While the Condor system will recover from a checkpoint server crash as best it can, there are two problems that can (and will) occur:

1. If the checkpoint server is not functioning, when jobs need to checkpoint, they cannot do so. The jobs will keep trying to contact the checkpoint server, backing off exponentially in the time they wait between attempts. Normally, jobs only have a limited time to checkpoint before they are kicked off the machine. So, if the server is down for a long period of time, chances are that you'll loose a lot of work by jobs being killed without writing a checkpoint.
2. When the jobs wish to start, if they cannot be retrieved from the checkpoint server, they will either have to be restarted from scratch, or the job will sit there, waiting for the server to come back on-line. You can control this behavior with the `MAX_DISCARDED_RUN_TIME` parameter in your config file (see section 3.4.10 on page 94 for details). Basically, this represents the maximum amount of CPU time you're willing to discard by starting a job over from scratch if the checkpoint server isn't responding to requests.

Preparing to Install a Checkpoint Server

Because of the problems that exist if your pool is configured to use a checkpoint server and that server is down, it is advisable to shut your pool down before doing any maintenance on your checkpoint server. See section 3.8 on page 128 for details on how to do that.

If you are installing a checkpoint server for the first time, you must make sure there are no jobs in your pool before you start. If you have jobs in your queues, with checkpoint files on the local spool directories of your submit machines, those jobs will never run if your submit machines are configured to use a checkpoint server and the checkpoint files cannot be found on the server. You can either remove jobs from your queues, or let them complete before you begin the installation of the checkpoint server.

Installing the Checkpoint Server Module

To install a checkpoint server, download the appropriate binary contrib module for the platform your server will run on. When you uncompress and untar that file, you'll have a directory that contains a `README`, `ckpt_server.tar`, and so on. The `ckpt_server.tar` acts much like the `release.tar` file from a main release. This archive contains these files:


```
sbin/condor_ckpt_server
sbin/condor_cleanckpts
etc/examples/condor_config.local.ckpt.server
```

These are all new files, not found in the main release, so you can safely untar the archive directly into your existing release directory. `condor_ckpt_server` is the checkpoint server binary. `condor_cleanckpts` is a script that can be periodically run to remove stale checkpoint files from your server. Normally, the checkpoint server cleans all old files by itself. However, in certain error situations, stale files can be left that are no longer needed. So, you may want to put a cron job in place that calls *condor_cleanckpts* every week or so, just to be safe. The example config file is described below.

Once you have unpacked the contrib module, you have a few more steps you must complete. Each will be discussed in their own section:

1. Configure the checkpoint server.
2. Spawn the checkpoint server.
3. Configure your pool to use the checkpoint server.

Configuring a Checkpoint Server

There are a few settings you must place in the local config file of your checkpoint server. The file `etc/examples/condor_config.local.ckpt.server` contains all such settings, and you can just insert it into the local configuration file of your checkpoint server machine.

There is one setting that you must customize, and that is `CKPT_SERVER_DIR`. The `CKPT_SERVER_DIR` defines where your checkpoint files should be located. It is better if this is on a very fast local file system (preferably a RAID). The speed of this file system will have a direct impact on the speed at which your checkpoint files can be retrieved from the remote machines.

The other optional settings are:

`DAEMON_LIST` (Described in section 3.4.7). If you want the checkpoint server managed by the *condor_master*, the `DAEMON_LIST` entry must have MASTER and CKPT_SERVER. Add STARTD if you want to allow jobs to run on your checkpoint server. Similarly, add SCHEDD if you would like to submit jobs from your checkpoint server.

The rest of these settings are the checkpoint-server specific versions of the Condor logging entries, described in section 3.4.3 on page 81.

`CKPT_SERVER_LOG` The CKPT_SERVER_LOG is where the checkpoint server log gets put.

MAX_CKPT_SERVER_LOG Use this item to configure the size of the checkpoint server log before it is rotated.

CKPT_SERVER_DEBUG The amount of information you would like printed in your logfile. Currently, the only debug level supported is D_ALWAYS.

Spawning a Checkpoint Server

To spawn a checkpoint server once it is configured to run on a given machine, all you have to do is restart Condor on that host to enable the *condor_master* to notice the new configuration. You can do this by sending a *condor_restart* command from any machine with "administrator" access to your pool. See section 3.7 on page 122 for full details about IP/host-based security in Condor.

Configuring your Pool to Use the Checkpoint Server

Once the checkpoint server is installed and running, you just have to change a few settings in your global config file to let your pool know about your new server:

USE_CKPT_SERVER This parameter should be set to "True".

CKPT_SERVER_HOST This parameter should be set to the full hostname of the machine that is now running your checkpoint server.

Once these settings are in place, you simply have to send a *condor_reconfig* to all machines in your pool so the changes take effect. This is described in section 3.8.2 on page 131.

3.3.6 Installing the PVM Contrib Module

For complete documentation on using PVM in Condor, see the section entitled "Parallel Applications in Condor: Condor-PVM" in the version 6.1 manual. This manual can be found at http://www.cs.wisc.edu/condor/manual/v6.1.

To install the PVM contrib module, all you have to do is download to appropriate binary module for whatever platform(s) you plan to use for Condor-PVM. Once you have downloaded each module, uncompressed and untarred it, you will be left with a directory that contains a pvm.tar, README and so on. The pvm.tar acts much like the release.tar file for a main release. It contains all the binaries and supporting files you would install in your release directory:

```
sbin/condor_pvmd
sbin/condor_pvmgs
sbin/condor_shadow.pvm
sbin/condor_starter.pvm
```


Since these files do not exist in a main release, you can safely untar the pvm.tar directly into your release directory, and you're done installing the PVM contrib module. Again, see the 6.1 manual for instructions on how to use PVM in Condor.

3.4 Configuring Condor

This section describes how to configure all parts of the Condor system. First, we describe some general information about the config files, their syntax, etc. Then, we describe settings that effect all Condor daemons and tools. Finally, we have a section describing the settings for each part of Condor. The only exception to this are the settings that control the policy under which Condor will start, suspend, resume, vacate or kill jobs. These settings (and other important concepts from the *condor_startd*) are described in section 3.5 on "Configuring Condor's Job Execution Policy".

3.4.1 Introduction to Config Files

The Condor configuration files are used to customize how Condor operates at a given site. The basic configuration as shipped with Condor should work well for most sites, with a few exceptions of things that might need special customization. Please see the section from the Installation section of this manual for details on where Condor's config files are found.

Each condor program will, as part of its initialization process, "configure" itself by calling a library routine which parses the various config files that might be used including pool-wide, platform-specific, machine-specific, and root-owned config files. The result is a list of constants and expressions which the program may evaluate as needed at run time.

Definitions in the configuration file come in two flavors, macros and expressions. Macros provide string valued constants which remain static throughout the execution of the program. Expressions can be arithmetic, boolean, or string valued, and can be evaluated dynamically at run time.

The order in which Macros and Expressions are defined is important, since you cannot define anything in terms of something else that hasn't been defined yet. This is particularly important if you break up your config files using the LOCAL_CONFIG_FILE setting described in sections 3.4.2 and 3.9.2 below.

Config File Macros

Macro definitions are of the form:

```
<macro_name> = <macro_definition>
```

NOTE: You must have whitespace between the macro name, the "=" sign, and the macro definition. Macro invocations are of the form:

```
$(macro_name)
```

Macro definitions may contain references to **previously defined macros**. Nothing in a config file can reference Macros which have not yet been defined. Thus,

```
A = xxx
C = $(A)
```

is a legal set of macro definitions, and the resulting value of "C" is "xxx". Note that "C" is actually bound to "$(A)", not its value, thus

```
A = xxx
C = $(A)
A = yyy
```

is also a legal set of macro definitions and the resulting value of "C" is "yyy". However,

```
A = $(C)
C = xxx
```

is not legal, and will result in the Condor daemons and tools exiting when they try to parse their config files.

Config File Expressions

Expression definitions are of the form:

```
<expression_name> : <expression>
```

NOTE: You must have whitespace between the expression name, the ":" sign, and the expression definition. Expressions may contain constants, operators, and other expressions. Macros may also be used to aid in writing expressions. Constants may be booleans, denoted by "true" (or "t") or "false" (or "f"), signed integers, floating point values, or strings enclosed in double quotes ("). All config file expressions are simply inserted into various ClassAds. Please see the appendix on ClassAds for details about ClassAd expression operators, and how ClassAd expressions are evaluated.

Note that expression which contain references to other expressions are bound to the expressions definition, not its current value, but expressions which contain macro invocations are bound to the current value of the macro. Thus


```
X : "xxx"
Y : X
X : "yyy"
```

will result in "Y" being evaluated as "yyy" at run time, but

```
X = "xxx"
Y : $(X)
X = "yyy"
```

will result in "Y" having a run time value of "xxx".

Other Syntax

Other than macros and expressions, a Condor config file can contain comments or continuations. A comment is any line begining with a "#". A continuation is any entry (either macro or expression) that continues across multiples lines. This is accomplished with the "\" sign at the end of any line that you wish to continue onto another. For example,

```
START : (KeyboardIdle > 15 * $(MINUTE)) && \
        ((LoadAvg - CondorLoadAvg) <= 0.3)
```

or,

```
ADMIN_MACHINES = condor.cs.wisc.edu, raven.cs.wisc.edu, \
                 stork.cs.wisc.edu, ostrich.cs.wisc.edu, \
                 bigbird.cs.wisc.edu
HOSTALLOW_ADMIN = $(ADMIN_MACHINES)
```

Pre-Defined Macros and Expressions

Condor provides a number of pre-defined macros and expressions that help you configure Condor. Pre-defined macros are listed as `$(macro_name)`, while pre-defined expressions are just listed as `expression_name`, to denote how they should be referenced in other macros or expressions.

The first set are special entries whose values are determined at runtime and cannot be overridden. These are inserted automatically by the library routine which parses the config files.

`CurrentTime` This expression provides the current result of the system call *time*(2). This is an integer containing the number of seconds since an arbitrary date defined by UNIX as the "beginning of time", hereafter referred to as the *UNIX date*.

`$(FULL_HOSTNAME)` This is the fully qualified hostname of the local machine (hostname plus domain name).

`$(HOSTNAME)` This is just the hostname of the local machine (no domain name).

`$(TILDE)` This is the full path to the home directory of user "condor", if such a user exists on the local machine.

`$(SUBSYSTEM)` This "subsystem" name of the daemon or tool that is evaluating the macro. The different subsystem names are described in section 3.4.1 below.

The final set are entries whose default values are determined automatically at runtime but which can be overridden.

`$(ARCH)` This setting defines the string used to identify the architecture of the local machine to Condor. The *condor_startd* will advertise itself with this attribute so that users can submit binaries compiled for a given platform and force them to run on the correct machines. *condor_submit* will automatically append a requirement to the job ClassAd that it must run on the same `ARCH` and `OPSYS` of the machine where it was submitted, unless the user specifies `ARCH` and/or `OPSYS` explicitly in their submit file. See the *condor_submit*(1) man page for details.

`$(OPSYS)` This setting defines the string used to identify the operating system of the local machine to Condor. See the entry on `ARCH` above for more information. If this setting is not defined in the config file, Condor will automatically insert the operating system of this machine as determined by*uname*.

`$(FILESYSTEM_DOMAIN)` This parameter defaults to the fully qualified hostname of the machine it is evaluated on. See section 3.4.5 on "Shared Filesystem Config File Entries" below for the full description of its use and under what conditions you would want to override it.

`$(UID_DOMAIN)` This parameter defaults to the fully qualified hostname of the machine it is evaluated on. See section 3.4.5 on "Shared Filesystem Config File Entries" below for the full description of its use and under what conditions you would want to override it.

Since `ARCH` and `OPSYS` will automatically be set to the right things, we recomend that you do not override them yourself. Only do so if you know what you are doing.

Condor Subsystem Names

IMPORTANT NOTE: Many of the entries in the config file will be named with the *subsystem* of the various Condor daemons. This is a unique string which identifies a given daemon within the Condor system. The possible subsystem names are:

- STARTD
- SCHEDD
- MASTER
- COLLECTOR
- NEGOTIATOR
- KBDD
- SHADOW
- STARTER
- CKPT_SERVER
- SUBMIT

In the description of the actual config file entries, "SUBSYS" will stand for one of these possible subsystem names.

3.4.2 Condor-wide Config File Entries

This section describes settings which effect all parts of the Condor system.

`CONDOR_HOST` This macro is just used to define the `NEGOTIATOR_HOST` and `COLLECTOR_HOST` macros. Normally, the *condor_collector* and *condor_negotiator* would run on the same machine. If for some reason they weren't, `CONDOR_HOST` would not be needed. Some of the host-based security macros use `CONDOR_HOST` by default. See section 3.7 on "Setting up IP/host-based security in Condor" for details.

`COLLECTOR_HOST` The hostname of the machine where the *condor_collector* is running for your pool. Normally it would just be defined with the `CONDOR_HOST` macro described above.

`NEGOTIATOR_HOST` The hostname of the machine where the *condor_negotiator* is running for your pool. Normally it would just be defined with the `CONDOR_HOST` macro described above.

`RELEASE_DIR` The full path to the Condor release directory, which holds the bin, etc, lib, and sbin directories. Other macros are defined relative to this one.

`BIN` This directory points to the Condor bin directory, where user-level programs are installed. It is usually just defined relative to the `RELEASE_DIR` macro.

`LIB` This directory points to the Condor lib directory, where libraries used to link jobs for Condor's standard universe are stored. The *condor_compile* program uses this macro to find these libraries, so it must be defined. `LIB` is usually just defined relative to the `RELEASE_DIR` macro.

`SBIN` This directory points to the Condor sbin directory, where Condor's system binaries (such as the binaries for the Condor daemons) and administrative tools are installed. Whatever directory `SBIN` points to should probably be in the `PATH` of anyone who is acting as a Condor administrator.

`LOCAL_DIR` The location of the local Condor directory on each machine in your pool. One common option is to use the condor user's home directory which you could specify with `$(tilde)`. For example:

```
LOCAL_DIR = $(tilde)
```

On machines with a shared filesystem, where either the `$(tilde)` directory or another directory you want to use is shared among all machines in your pool, you might use the `$(hostname)` macro and have a directory with many subdirectories, one for each machine in your pool, each named by hostnames. For example:

```
LOCAL_DIR = $(tilde)/hosts/$(hostname)
```

or:

```
LOCAL_DIR = $(release_dir)/hosts/$(hostname)
```

`LOG` This entry is used to specify the directory where each Condor daemon writes its log files. The names of the log files themselves are defined with other macros, which use the `LOG` macro by default. The log directory also acts as the current working directory of the Condor daemons as the run, so if one of them should drop a core file for any reason, it would wind up in the directory defined by this macro. Normally, `LOG` is just defined in terms of `$(LOCAL_DIR)`.

`SPOOL` The spool directory is where certain files used by the *condor_schedd* are stored, such as the job queue file, and the initial executables of any jobs that have been submitted. In addition, if you are not using a checkpoint server, all the checkpoint files from jobs that have been submitted from a given machine will be store in that machine's spool directory. Therefore, you will want to ensure that the spool directory is located on a partition with enough disk space. If a given machine is only setup to execute Condor jobs and not submit them, it would not need a spool directory (or this macro defined). Normally, `SPOOL` is just defined in terms of `$(LOCAL_DIR)`.

`EXECUTE` This directory acts as the current working directory of any Condor job that is executing on the local machine. If a given machine is only setup to only submit jobs and not execute them, it would not need an execute directory (or this macro defined). Normally, `EXECUTE` is just defined in terms of `$(LOCAL_DIR)`.

`LOCAL_CONFIG_FILE` The location of the local, machine-specific config file for each machine in your pool. The two most common options would be putting this file in the `$(LOCAL_DIR)` you just defined, or putting all local config files for your pool in a shared directory, each one named by hostname. For example:

```
LOCAL_CONFIG_FILE = $(LOCAL_DIR)/condor_config.local
```

or:

```
LOCAL_CONFIG_FILE = $(release_dir)/etc/$(hostname).local
```

or, not using your release directory:

```
LOCAL_CONFIG_FILE = /full/path/to/configs/$(hostname).local
```

Begining with Condor version 6.0.1, the `LOCAL_CONFIG_FILE` is treated as a list of files, not a single file. So, you can use either a comma or space seperated list of files as its value. This allows you to specify multiple files as the "local config file" and each one will be processed in order (with parameters set in later files overridding values from previous files). This allows you use one global config file for multiple platforms in your pool, define a platform-specific config file for each platform, and finally use a local config file for each machine. For more information on this, see section 3.9.2 on "Configuring Condor for Multiple Platforms" on page 133.

`CONDOR_ADMIN` This is the email address that Condor will send mail to when something goes wrong in your pool. For example, if a daemon crashes, the *condor_master* can send an *obituary* to this address with the last few lines of that daemon's log file and a brief message that describes what signal or exit status that daemon exited with.

`MAIL` This is the full path to a mail sending program that understands that "-s" means you wish to specify a subject to the message you're sending. On all platforms, the default shipped with Condor should work. Only if you have installed things in a non-standard location on your system would you need to change this setting.

`RESERVED_SWAP` This setting determines how much swap space you want to reserve for your own machine. Condor will not start up more *condor_shadow* processes if the amount of free swap space on your machine falls below this level.

`RESERVED_DISK` This setting determines how much disk space you want to reserve for your own machine. When Condor is reporting the amount of free disk space in a given partition on your machine, it will always subtract this amount. For example, the *condor_startd* advertises the amount of free space in the `EXECUTE` directory described above.

`LOCK` Condor needs to create a few lock files to synchronize access to various log files. Because of problems we've had with network filesystems and file locking over the years, we **highly** recommend that you put these lock files on a local partition on each machine. If you don't have your LOCAL_DIR on a local partition, be sure to change this entry. Whatever user (or group) condor is running as needs to have write access to this directory. If you're not running as root, this is whatever user you started up the *condor_master* as. If you are running as root, and there's a condor account, it's probably condor. Otherwise, it's whatever you've set in the CONDOR_IDS environment variable. See section 3.10.2 on "UIDs in Condor" for details on this.

HISTORY This entry defines the location of the Condor history file, which stores information about all Condor jobs that have completed on a given machine. This entry is used by both the *condor_schedd* which appends the information, and *condor_history*, the user-level program that is used to view the history file.

DEFAULT_DOMAIN_NAME If you don't use a fully qualified name in your /etc/hosts file (or NIS, etc.) for either your official hostname or as an alias, Condor wouldn't normally be able to use fully qualified names in places that it'd like to. You can set this parameter to the domain you'd like appended to your hostname, if changing your host information isn't a good option. This parameter must be set in the global config file (not the LOCAL_CONFIG_FILE specified above. The reason for this is that the FULL_HOSTNAME special macro is used by the config file code in Condor which needs to know the full hostname. So, for DEFAULT_DOMAIN_NAME to take effect, Condor must already have read in its value. However, Condor must set the FULL_HOSTNAME special macro since you might use that to define where your local config file is. So, after reading the global config file, Condor figures out the right values for HOSTNAME and FULL_HOSTNAME and inserts them into its configuration table.

CREATE_CORE_FILES Condor can be told whether or not you want the Condor daemons to create a core file if something really bad happens. This just sets the resource limit for the size of a core file. By default, we don't do anything, and leave in place whatever limit was in effect when you started the Condor daemons (normally the *condor_master*). If this parameter is set and "True", we increase the limit to as large as it gets. If it's set to "False", we set the limit at 0 (which means that no core files are even created). Core files greatly help the Condor developers debug any problems you might be having. By using the parameter, you don't have to worry about tracking down where in your boot scripts you need to set the core limit before starting Condor, etc. You can just set the parameter to whatever behavior you want Condor to enforce. This parameter has no default value, and is commented out in the default config file.

3.4.3 Daemon Logging Config File Entries

These entries control how and where the Condor daemons write their log files. All of these entries are named with the subsystem (as described in section 3.4.1 above) of the daemon you wish to control logging for.

SUBSYS_LOG This is the name of the log file for the given subsystem. For example, STARTD_LOG gives the location of the log file for the *condor_startd*. These entries

are defined relative to the LOG macro described above. The actual names of the files are also used in the VALID_LOG_FILES entry used by *condor_preen*, which is described below. If you change one of the filenames with this setting, be sure to change the VALID_LOG_FILES entry as well, or *condor_preen* will delete your newly named log files.

MAX_SUBSYS_LOG This setting controls the maximum length in bytes to which the various logs will be allowed to grow. Each log file will grow to the specified length, then be saved to a ".old" file. The ".old" files are overwritten each time the log is saved, thus the maximum space devoted to logging for any one program will be twice the maximum length of its log file.

TRUNC_SUBSYS_LOG_ON_OPEN If this macro is defined and set to "True" the affected log will be truncated and started from an empty file with each invocation of the program. Otherwise, new invocations of the program will simply append to the previous log file. By default this setting is turned off for all daemons.

SUBSYS_DEBUG All of the Condor daemons can produce different levels of output depending on how much information you want to see. The various levels of verbosity for a given daemon are determined by this entry. All daemons have a default level, D_ALWAYS, and log message for that level will be printed to the daemon's log, regardless of what you have set here. The other possible debug levels are:

D_FULLDEBUG Generally, turning on this setting provides very verbose output in the log files.

D_DAEMONCORE This provides log file entries for things that are specific to DaemonCore, such as timers the daemons have set, the commands that are registered, and so on. If both D_FULLDEBUG and D_DAEMONCORE are set, you get **VERY** verbose output.

D_PRIV This flag provides turns on log messages about the *privilege state* switching that the daemons do. See section 3.10.2 on UIDs in Condor for more details.

D_COMMAND With this flag set, a Any daemon that uses DaemonCore will print out a log message whenever a command comes in. The name and integer of the command are printed, whether the command was sent via UDP or TCP, and where the command was sent from. Because the *condor_kbdd* works by sending UDP commands to the *condor_startd* whenever there is activity on the X server, we don't recommend turning on D_COMMAND login in the *condor_startd*, since you will get so many messages that the log file will be fairly useless to you. On platforms that use the *condor_kbdd*, this is turned off in the *condor_startd* by default.

D_LOAD The *condor_startd* keeps track of the load average on the machine where it is running. Both the general system load average, and the load average being generated by Condor's activity there. With this flag set, the *condor_startd* will print out a log message with the current state of both of these load averages whenever it computes them. This flag only effects the *condor_startd*.

D_JOB When this flag is set, the *condor_startd* will dump out to its log file the contents of any job ClassAd that the *condor_schedd* sends to claim the *condor_startd* for its use. This flag only effects the *condor_startd*.

D_MACHINE When this flag is set, the *condor_startd* will dump out to its log file the contents of its resource ClassAd when the *condor_schedd* tries to claim the *condor_startd* for its use. This flag only effects the *condor_startd*.

D_SYSCALLS This flag is used to make the *condor_shadow* log remote syscall requests and return values. This can help track down problems a user is having with a particular job since you can see what system calls the job is performing, which, if any, are failing, and what the reason for the failure is. The *condor_schedd* also uses this flag for the server portion of the queue management code. So, with D_SYSCALLS defined in SCHEDD_DEBUG you will see verbose logging of all queue management operations the *condor_schedd* performs.

3.4.4 DaemonCore Config File Entries

Please read section 3.6 on "DaemonCore" for details about DaemonCore is. There are certain config file settings that DaemonCore uses which affect all Condor daemons (except the checkpoint server, shadow, and starter, none of which use DaemonCore yet).

HOSTALLOW... All of the settings that begin with either HOSTALLOW or HOSTDENY are settings for Condor's host-based security. Please see section 3.7 on "Setting up IP/host-based security in Condor" for details on all of these settings and how to configure them.

SHUTDOWN_GRACEFUL_TIMEOUT This entry determines how long you are willing to let daemons try their graceful shutdown methods before they do a hard shutdown. It is defined in terms of seconds. The default is 1800 (30 minutes).

SUBSYS_ADDRESS_FILE Every Condor daemon that uses DaemonCore has a command port where commands can be sent. The IP/port of the daemon is put in that daemon's ClassAd so that other machines in the pool can query the *condor_collector* (which listens on a well-known port) to find the address of a given daemon on a given machine. However, tools and daemons executing on the same machine they wish to communicate with don't have to query the collector, they can simply look in a file on the local disk to find the IP/port. Setting this entry will cause daemons to write the IP/port of their command socket to the file you specify. This way, local tools will continue to operate, even if the machine running the *condor_collector* crashes. Using this file will also generate slightly less network traffic in your pool (since *condor_q*, *condor_rm*, etc won't have to send any messages over the network to locate the *condor_schedd*, for example). This entry is not needed for the collector or negotiator, since their command sockets are at well-known ports anyway.

SUBSYS_EXPRS This entry allows you to have the any DaemonCore daemon advertise arbitrary expressions from the config file in its ClassAd. Give the comma-separated list of entries from the config file you want in the given daemon's ClassAd.

NOTE: The *condor_negotiator* and *condor_kbdd* do not send ClassAds now, so this entry does not effect them at all. The *condor_startd*, *condor_schedd*, *condor_master*, and *condor_collector* do send ClassAds, so those would be valid subsystems to set this entry for.

OTHER NOTE: Setting SUBMIT_EXPRS has the slightly different effect of having the named expressions inserted into all the job ClassAds that *condor_submit* creates. This is equivalent to the "+" syntax in submit files. See the *condor_submit*(1) man page for details.

OTHER NOTE: Because of the different syntax of the config file and ClassAds, you might have to do a little extra work to get a given entry into the ClassAd. In particular, ClassAds require quote marks (") around your strings. Numeric values can go in directly, as can expressions or boolean macros. For example, if you wanted the startd to advertise a macro that was a string, a numeric macro, and a boolean expression, you'd have to do something like the following:

```
STRING_MACRO = This is a string macro
NUMBER_MACRO = 666
BOOL_MACRO = True
EXPR : CurrentTime >= $(NUMBER_MACRO) || $(BOOL_MACRO)
MY_STRING_MACRO = "$(STRING_MACRO)"
STARTD_EXPRS = MY_STRING_MACRO, NUMBER_MACRO, BOOL_MACRO, EXPR
```

3.4.5 Shared Filesystem Config File Entries

These entries control how Condor interacts with various shared and network filesystems. If you are using AFS as your shared filesystem, be sure to read section 3.9.1 on "Using Condor with AFS"

UID_DOMAIN Often times, especially if all the machines in the pool are administered by the same organization, all the machines to be added into a Condor pool share the same login account information. Specifically, does user X have UID Y on all machines within a given Internet/DNS domain? This is usually the case if a central authority creates user logins and maintains a common /etc/passwd file on all machines (perhaps via NIS/Yellow Pages, distributing the passwd file, etc). If this is the case, then set this macro to the name of the Internet/DNS domain where this is true. For instance, if all the machines in this Condor pool within the Internet/DNS zone "cs.wisc.edu" have a common passwd file, UID_DOMAIN would be set to "cs.wisc.edu". If this is not the case you can comment out the entry and Condor will automatically use the fully qualified hostname of each machine. If you put in a "*", that means a wildcard to match all domains and therefore to honor all UIDs - dangerous idea.

Condor uses this information to determine if it should run a given Condor job on the remote execute machine with the UID of whomever submitted the job or with the UID of user "nobody". If you set this to "none" or don't set it at all, then Condor jobs will always execute with the access permissions of user "nobody". For security purposes, it is not a bad idea to have Condor jobs that migrate around on machines across an entire organization to run as user "nobody", which by convention has very restricted access to the disk files of a machine. Standard Universe Condor jobs are perfectly happy to run as user nobody since all I/O is redirected back via remote system calls to a shadow process running on the submit machine which is authenticated as the user. If you only plan on running Standard Universe jobs, then it is a good idea to simply set this to "none" or don't define it. Vanilla Universe jobs, however, cannot take advantage of Condor's remote system calls. Vanilla Universe jobs are dependent upon NFS, RFS, AFS, or some shared filesystem setup to read/write files as they bounce around from machine to machine. If you want to run Vanilla jobs and your shared filesystems are via AFS, then you can safely leave this as "none" as well. But if you wish to use Vanilla jobs with Condor and you have shared filesystems via NFS or RFS, then you should enter in a legitimate domain name where all your UIDs match (you should be doing this with NFS anyway!) on all machines in the pool, or else users in your pool who submit Vanilla jobs will have to make their files world read/write (so that user nobody can access them).

Some gritty details for folks who want to know: If the submitting machine and the remote machine about to execute the job both have the same login name in the passwd file for a given UID, and the `UID_DOMAIN` claimed by the submit machine is indeed found to be a subset of what an inverse lookup to a DNS (domain name server) or NIS reports as the fully qualified domain name for the submit machine's IP address (this security measure safeguards against the submit machine from simply lying), **THEN** the job will run with the same UID as the user who submitted the job. Otherwise it will run as user "nobody".

Note: the `UID_DOMAIN` parameter is also used when Condor sends email back to the user about a completed job; the address `Job-Owner@UID_DOMAIN` is used, unless `UID_DOMAIN` is "none", in which case `Job-Owner@submit-machine` is used.

`SOFT_UID_DOMAIN` This setting is used in conjuction with the `UID_DOMAIN` setting described above. If the `UID_DOMAIN` settings match on both the execute and submit machines, but the uid of the user who submitted the job isn't in the passwd file (or password info if NIS is being used) of the execute machine, the *condor_starter* will normally exit with an error. If you set `SOFT_UID_DOMAIN` to be "True", Condor will simply start the job with the specified uid, even if it's not in the passwd file.

`FILESYSTEM_DOMAIN` This setting is similar in concept to `UID_DOMAIN`, but here we need the Internet/DNS domain name where all the machines within that domain can access the same set of NFS file servers.

Often times, especially if all the machines in the pool are administered by the same organization, all the machines to be added into a Condor pool can mount the same set of NFS fileservers onto the same place in the directory tree. Specifically, do all the machines in the pool within a specific Internet/DNS domain mount the same set of NFS file servers onto the same path mount-points? If this is the case, then set this macro to the name of the Internet/DNS domain where this is true. For instance, if all the machines in the Condor pool within the Internet/DNS zone "cs.wisc.edu" have

a common passwd file and mount the same volumes from the same NFS servers, set `FILESYSTEM_DOMAIN` to "cs.wisc.edu". If this is not the case you can comment out the entry, and Condor will automatically set it to the fully qualified hostname of the local machine.

`HAS_AFS` Set this to "True" if all the machines you plan on adding in your pool all can access a common set of AFS fileservers. Otherwise, set it to "False".

`FS_PATHNAME` If you're using AFS, Condor needs to know where the AFS "fs" command is located so that it can verify the AFS cell-names of machines in the pool. The default value of `/usr/afsws/bin/fs` is also the default that AFS uses.

`VOS_PATHNAME` If you're using AFS, Condor needs to know where the AFS "vos" command is located so that it can compare fileserver names of volumes. The default value of `/usr/afsws/etc/vos` is also the default that AFS uses.

`RESERVE_AFS_CACHE` If your machine is running AFS and the AFS cache lives on the same partition as the other Condor directories, and you want Condor to reserve the space that your AFS cache is configured to use, set this entry to "True". It defaults to "False".

`USE_NFS` This setting influences how Condor jobs running in the Standard Universe will access their files. Condor will normally always redirect the file I/O requests of Standard Universe jobs back to be executed on the machine which submitted the job. Because of this, as a Condor job migrates around the network, the filesystem always appears to be identical to the filesystem where the job was submitted. However, consider the case where a user's data files are sitting on an NFS server. The machine running the user's program will send all I/O over the network to the machine which submitted the job, which in turn sends all the I/O over the network a second time back to the NFS file server. Thus, all of the program's I/O is being sent over the network twice.

If you set this macro to "True", then Condor will attempt to read/write files without redirecting them back to the submitting machine if both the submitting machine and the machine running the job are both accessing the same NFS servers (if they're both in the same `FILESYSTEM_DOMAIN`, as described above). The result is I/O performed by Condor Standard Universe programs is only sent over the network once.

While sending all file operations over the network twice might sound really bad, unless you are operating over networks where bandwidth as at a very high premium, practical experience reveals that this scheme offers very little real performance gain. There are also some (fairly rare) situations where this scheme can break down.

Setting `USE_NFS` to "False" is always safe. It may result in slightly more network traffic, but Condor jobs are ideally heavy on CPU and light on I/O anyway. It also ensures that a remote Standard Universe Condor job will always use Condor's remote system calls mechanism to reroute I/O and therefore see the exact same filesystem that the user sees on the machine where she/he submitted the job.

Some gritty details for folks who want to know: If the you set `USE_NFS` to "True", and the `FILESYSTEM_DOMAIN` of both the submitting machine and the remote machine about to execute the job match, and the FILESYSTEM_DOMAIN claimed by the submit machine is indeed found to be a subset of what an inverse lookup to a DNS (domain name server) reports as the fully qualified domain name for the submit machine's IP address (this security measure safeguards against the submit machine from simply lying), **THEN** the job will access files via a local system call, without redirecting them to the submitting machine (a.k.a. with NFS). Otherwise, the system call will get routed back to the submitting machine via Condor's remote system call mechanism.

USE_AFS If your machines have AFS and the submit and execute machines are in the same AFS cell, this setting determines whether Condor will use remote system calls for Standard Universe jobs to send I/O requests to the submit machine, or if it should use local file access on the execute machine (which will then use AFS to get to the submitter's files). Read the setting above on USE_NFS for a discussion of why you might want to use AFS access instead of remote system calls.

One important difference between USE_NFS and USE_AFS is the AFS cache. With USE_AFS set to "True", the remote Condor job executing on some machine will start messing with the AFS cache, possibly evicting the machine owner's files from the cache to make room for its own. Generally speaking, since we try to minimize the impact of having a Condor job run on a given machine, we don't recomend using this setting.

While sending all file operations over the network twice might sound really bad, unless you are operating over networks where bandwidth as at a very high premium, practical experience reveals that this scheme offers very little real performance gain. There are also some (fairly rare) situations where this scheme can break down.

Setting USE_AFS to "False" is always safe. It may result in slightly more network traffic, but Condor jobs are ideally heavy on CPU and light on I/O anyway. "False" ensures that a remote Standard Universe Condor job will always see the exact same filesystem that the user on sees on the machine where he/she submitted the job. Plus, it will ensure that the machine where the job executes doesn't have its AFS cache screwed up as a result of the Condor job being there.

However, things may be different at your site, which is why the setting is there.

3.4.6 Checkpoint Server Config File Entries

These entries control whether or not Condor user a checkpoint server. In addition, if you are using a checkpoint server, this section describes the settings that the checkpoint server itself needs to have defined. If you decide to use a checkpoint server, you must install it seperately (it is not included in the main Condor binary distribution or installation procedure). See section 3.3.5 on "Installing a Checkpoint Server" for details on installing and running a checkpoint server for your pool.

NOTE: If you are setting up a machine to join to UW-Madison CS Department Condor pool, you **should** configure the machine to use a checkpoint server, and use "condor-ckpt.cs.wisc.edu" as the checkpoint server host (see below).

USE_CKPT_SERVER A boolean which determines if you want a given machine machine to use the checkpoint server for your pool.

CKPT_SERVER_HOST The hostname of the checkpoint server for your pool.

CKPT_SERVER_DIR The checkpoint server needs this macro defined to the full path of the directory the server should use to store checkpoint files. Depending on the size of your pool and the size of the jobs your users are submitting, this directory (and it's subdirectories) might need to store many megabytes of data.

3.4.7 condor_master Config File Entries

These settings control the *condor_master*.

DAEMON_LIST This macro determines what daemons the *condor_master* will start and keep its watchful eyes on. The list is a comma or space seperated list of subsystem names (described above in section 3.4.1). For example,

```
DAEMON_LIST = MASTER, STARTD, SCHEDD
```

NOTE: On your central manager, your DAEMON_LIST will be different from your regular pool, since it will include entries for the *condor_collector* and *condor_negotiator*.

NOTE: On machines running Digital Unix or IRIX, your DAEMON_LIST will also include "KBDD", for the *condor_kbdd*, which is a special daemon that runs to monitor keyboard and mouse activity on the console. It is only with this special daemon that we can aquire this information on those platforms.

SUBSYS Once you have defined which subsystems you want the *condor_master* to start, you must provide it with the full path to each of these binaries. For example:

```
MASTER          = $(SBIN)/condor_master
STARTD          = $(SBIN)/condor_startd
SCHEDD          = $(SBIN)/condor_schedd
```

Generally speaking, these would be defined relative to the SBIN macro.

PREEN In addition to the daemons defined in DAEMON_LIST, the *condor_master* also starts up a special process, *condor_preen* to clean out junk files that have been left lying around by Condor. This macro determines where the *condor_master* finds the preen binary. It also controls how *condor_preen* behaves by the command-line arguments you pass to "-m" means you want email about files *condor_preen* finds that it thinks it should remove. "-r" means you want *condor_preen* to actually remove these files. If you don't want preen to run at all, just comment out this setting.

PREEN_INTERVAL This macro determines how often *condor_preen* should be started. It is defined in terms of seconds and defaults to 86400 (once a day).

`PUBLISH_OBITUARIES` When a daemon crashes, the *condor_master* can send email to the address specified by `CONDOR_ADMIN` with an obituary letting the administrator know that the daemon died, what it's cause of death was (which signal or exit status it exited with), and (optionally) the last few entries from that daemon's log file. If you want these obituaries, set this entry to "True".

`OBITUARY_LOG_LENGTH` If you're getting obituaries, this setting controls how many lines of the log file you want to see.

`START_MASTER` If this setting is defined and set to "False" when the master starts up, the first thing it will do is exit. This may seem strange, but perhaps you just don't want Condor to run on certain machines in your pool, yet the boot scripts for your entire pool are handled by a centralized system that starts up the *condor_master* automatically. This is certainly an entry you'd most likely find in a local config file, not your global config file.

`START_DAEMONS` This setting is similar to the `START_MASTER` macro described above. However, the *condor_master* doesn't exit, it just doesn't start any of the daemons listed in the `DAEMON_LIST`. This way, you could start up the daemons at some later time with a *condor_on* command if you wished.

`MASTER_UPDATE_INTERVAL` This entry determines how often the *condor_master* sends a ClassAd update to the *condor_collector*. It is defined in seconds and defaults to 300 (every 5 minutes).

`MASTER_CHECK_NEW_EXEC_INTERVAL` This setting controls how often the *condor_master* checks the timestamps of the daemons it's running. If any daemons have been modified, the master restarts them. It is defined in seconds and defaults to 300 (every 5 minutes).

`MASTER_NEW_BINARY_DELAY` Once the *condor_master* has discovered a new binary, this macro controls how long it waits before attempting to execute the new binary. This delay is here because the *condor_master* might notice a new binary while you're in the process of copying in new binaries and the entire file might not be there yet (in which case trying to execute it could yield unpredictable results). The entry is defined in seconds and defaults to 120 (2 minutes).

`SHUTDOWN_FAST_TIMEOUT` This macro determines the maximum amount of time you're willing to give the daemons to perform their fast shutdown procedure before the *condor_master* just kills them outright. It is defined in seconds and defaults to 120 (2 minutes).

`MASTER_BACKOFF_FACTOR` If a daemon keeps crashing, we use *exponential backoff* so we wait longer and longer before restarting it. At the end of this section, there is an example that shows how all these settings work. This setting is the base of the exponent used to determine how long to wait before starting the daemon again. It defaults to 2.

`MASTER_BACKOFF_CEILING` This entry determines the maximum amount of time you want the master to wait between attempts to start a given daemon. (With 2.0 as the

`MASTER_BACKOFF_FACTOR`, you'd hit 1 hour in 12 restarts). This is defined in terms of seconds and defaults to 3600 (1 hour).

`MASTER_RECOVER_FACTOR` How long should a daemon run without crashing before we consider it *recovered*. Once a daemon has recovered, we reset the number of restarts so the exponential backoff stuff goes back to normal. This is defined in terms of seconds and defaults to 300 (5 minutes).

Just for clarity, here's a little example of how all these exponential backoff settings work. The example is worked out in terms of the default settings.

When a daemon crashes, it is restarted in 10 seconds. If it keeps crashing, we wait longer and longer before restarting it based on how many times it's been restarted. We take the number of times the daemon has restarted, take the `MASTER_BACKOFF_FACTOR` (defaults to 2) to that power, and add 9. Sounds complicated, but here's how it works:

```
1st crash:  restarts == 0, so, 9 + 2^0 = 9 + 1 = 10 seconds
2nd crash:  restarts == 1, so, 9 + 2^1 = 9 + 2 = 11 seconds
3rd crash:  restarts == 2, so, 9 + 2^2 = 9 + 4 = 13 seconds
...
6th crash:  restarts == 5, so, 9 + 2^5 = 9 + 32 = 41 seconds
...
9th crash:  restarts == 8, so, 9 + 2^8 = 9 + 256 = 265 seconds
```

If the daemon kept dying and restarting, after the 13th crash, you'd have:

```
13th crash:  restarts == 12, so, 9 + 2^12 = 9 + 4096 = 4105 seconds
```

This is bigger than the `MASTER_BACKOFF_CEILING`, which defaults to 3600, so the daemon would really be restarted after only 3600 seconds, not 4105. Assuming a few hours went by like this, with the *condor_master* trying again every hour (since the numbers would get even more huge, but would always be capped by the ceiling). Eventually, imagine that daemon finally started and didn't crash (for example, after the email you got about the daemon crashing, you realized that you had accidentally deleted its binary so you reinstalled it). If it stayed alive for `MASTER_RECOVER_FACTOR` seconds (defaults to 5 minutes). We'd reset the count of how many restarts this daemon has performed. So, if 15 minutes later, it died again, it would be restarted in 10 seconds, not 1 hour.

The moral of the story is that this is some relatively complicated stuff. The defaults we have work quite well, and you probably won't want to change them for any reason.

`MASTER_EXPRS` This setting is described above in section 3.4.4 as `SUBSYS_EXPRS`.

`MASTER_DEBUG` This setting (and other settings related to debug logging in the master) is described above in section 3.4.3 as `SUBSYS_DEBUG`.

MASTER_ADDRESS_FILE This setting is described above in section 3.4.4 as SUBSYS_ADDRESS_FILE

3.4.8 condor_startd Config File Entries

These settings control general operation of the *condor_startd*. Information on how to configure the *condor_startd* to start, suspend, resume, vacate and kill remote Condor jobs can be found in a separate top-level section, section 3.5 on "Configuring The Startd Policy". In there, you will find information on the startd's *states* and *activities*. If you see entries in the config file that are not described here, it is because they control state or activity transitions within the *condor_startd* and are described in section'3.5.

STARTER This macro holds the full path to the regular *condor_starter* binary the startd should spawn. It is normally defined relative to `$(SBIN)`.

ALTERNATE_STARTER_1 This macro holds the full path to the special *condor_starter.pvm* binary the startd spawns to service PVM jobs. It is normally defined relative to `$(SBIN)`, since by default, *condor_starter.pvm* is installed in the regular Condor release directory.

POLLING_INTERVAL When a startd is claimed, this setting determines how often we should poll the state of the machine to see if we need to suspend, resume, vacate or kill the job. Defined in terms of seconds and defaults to 5.

UPDATE_INTERVAL This entry determines how often the startd should send a ClassAd update to the *condor_collector*. The startd also sends update on any state or activity change, or if the value of its START expression changes. See section 3.5.5 on "condor_startd States", section 3.5.6 on "condor_startd Activities", and section 3.5.3 on "condor_startd START expression" for details on states, activities, and the START expression respectively. This entry is defined in terms of seconds and defaults to 300 (5 minutes).

STARTD_HAS_BAD_UTMP Normally, when the startd is computing the idle time of all the users of the machine (both local and remote), it checks the `utmp` file to find all the currently active ttys, and only checks access time of the devices associated with active logins. Unfortunately, on some systems, `utmp` is unreliable, and the startd might miss keyboard activity by doing this. So, if your `utmp` is unreliable, set this setting to "True" and the startd will check the access time on all tty and pty devices.

CONSOLE_DEVICES This macro allows the startd to monitor console (keyboard and mouse) activity by checking the access times on special files in `/dev`. Activity on these files shows up as "ConsoleIdle" time in the startd's ClassAd. Just give a comma-separated list of the names of devices you want considered the console, without the "`/dev/`" portion of the pathname. The defaults vary from platform to platform, and are usually correct.

One possible exception to this is that on Linux, we use "mouse" as one of the entries here. Normally, Linux installations put in a soft link from `/dev/mouse` that points to the appropriate device (for example, `/dev/psaux` for a PS/2 bus mouse, or `/dev/tty00` for a serial mouse connected to com1). However, if your installation doesn't have this soft link, you will either need to put it in (which you'll be glad you did), or change this setting to point to the right device.

Unfortunately, there are no such devices on Digital Unix or IRIX (don't be fooled by `/dev/keyboard0`, etc, the kernel does not update the access times on these devices) so this entry is not useful there, and we must use the *condor_kbdd* to get this information by connecting to the X server.

STARTD_JOB_EXPRS When the startd is claimed by a remote user, it can also advertise arbitrary attributes from the ClassAd of the job its working on. Just list the attribute names you want advertised. Note: since these are already ClassAd expressions, you don't have to do anything funny with strings, etc.

STARTD_EXPRS This setting is described above in section 3.4.4 as SUBSYS_EXPRS.

STARTD_DEBUG This setting (and other settings related to debug logging in the startd) is described above in section 3.4.3 as SUBSYS_DEBUG.

STARTD_ADDRESS_FILE This setting is described above in section 3.4.4 as SUBSYS_ADDRESS_FILE

3.4.9 condor_schedd Config File Entries

These settings control the *condor_schedd*.

SHADOW This macro determines the full path of the *condor_shadow* binary that the *condor_schedd* spawns. It is normally defined in terms of `$(SBIN)`.

SHADOW_PVM This macro determines the full path of the special *condor_shadow.pvm* binary used for supporting PVM jobs that the *condor_schedd* spawns. It is normally defined in terms of `$(SBIN)`.

MAX_JOBS_RUNNING This setting controls the maximum number of *condor_shadow* processes you're willing to let a given *condor_schedd* spawn. The actual number of *condor_shadow*'s might be less than that if you reached your RESERVED_SWAP limit.

MAX_SHADOW_EXCEPTIONS This setting controls the maximum number of times that a *condor_shadow* processes can have a fatal error (exception) before the *condor_schedd* will simply relinquish the match associated with the dying shadow. Defaults to 5.

SCHEDD_INTERVAL This entry determines how often the *condor_schedd* should send a ClassAd update to the *condor_collector*. It is defined in terms of seconds and defaults to 300 (every 5 minutes).

JOB_START_DELAY When the *condor_schedd* has finished negotiating and has a lot of new *condor_startd*'s that it has claimed, the *condor_schedd* can wait a certain delay before starting up a *condor_shadow* for each job it's going to run. This prevents a sudden, large load on the submit machine as it spawns many shadows simultaneously, and having to deal with their startup activity all at once. This macro determines how how long the *condor_schedd* should wait in between spawning each *condor_shadow*. Defined in terms of seconds and defaults to 2.

ALIVE_INTERVAL This setting determines how often the schedd should send a keep alive message to any startd it has claimed. When the schedd claims a startd, it tells the startd how often it's going to send these messages. If the startd doesn't get one of these messages after 3 of these intervals has passed, the startd releases the claim, and the schedd is no longer paying for the resource (in terms of priority in the system). The macro is defined in terms of seconds and defaults to 300 (every 5 minutes).

SHADOW_SIZE_ESTIMATE This entry is the estimated virtual memory size of each *condor_shadow* process. Specified in kilobytes. The default varies from platform to platform.

SHADOW_RENICE_INCREMENT When the schedd spawns a new *condor_shadow*, it can do so with a *nice-level*. This is a mechanism in UNIX where you can assign your own processes a lower priority so that they don't interfere with interactive use of the machine. This is very handy for keeping a submit machine with lots of shadows running still useful to the owner of the machine. The entry can be any integer between 1 and 19. It defaults to 10.

QUEUE_CLEAN_INTERVAL The schedd maintains the job queue on a given machine. It does so in a persistent way such that if the schedd crashes, it can recover a valid state of the job queue. The mechanism it uses is a transaction-based log file (this is the job_queue.log file, not the SchedLog file). This file contains some initial state of the job queue, and a series of transactions that were performed on the queue (such as new jobs submitted, jobs completing, checkpointing, whatever). Periodically, the schedd will go through this log, truncate all the transactions and create a new file with just the new initial state of the log. This is a somewhat expensive operation, but it speeds up when the schedd restarts since there are less transactions it has to play to figure out what state the job queue is really in. This macro determines how often the schedd should to this "queue cleaning". It is defined in terms of seconds and defaults to 86400 (once a day).

ALLOW_REMOTE_SUBMIT Starting with Condor Version 6.0, users can run *condor_submit* on one machine and actually submit jobs to another machine in the pool. This is called a *remote submit*. Jobs submitted in this way are entered into the job queue owned by user "nobody". This entry determines whether you want to allow such a thing to happen to a given schedd. It defaults to "False".

QUEUE_SUPER_USERS This macro determines what usernames on a given machine have *super-user access* to your job queue, meaning that they can modify or delete the job ClassAds of other users. (Normally, you can only modify or delete ClassAds that you own from

the job queue). Whatever username corresponds with the UID that Condor is running as (usually "condor") will automatically get included in this list because that is needed for Condor's proper functioning. See section 3.10.2 on "UIDs in Condor" for more details on this. By default, we give "root" the ability to remove other user's jobs, in addition to user "condor".

SCHEDD_LOCK This entry specifies what lock file should be used for access to the SchedLog file. It must be a separate file from the SchedLog, since the SchedLog may be rotated and you want to be able to synchronize access across log file rotations. This macro is defined relative to the LOCK macro described above. If, for some strange reason, you decide to change this setting, be sure to change the VALID_LOG_FILES entry that *condor_preen* uses as well.

SCHEDD_EXPRS This setting is described above in section 3.4.4 as SUBSYS_EXPRS.

SCHEDD_DEBUG This setting (and other settings related to debug logging in the schedd) is described above in section 3.4.3 as SUBSYS_DEBUG.

SCHEDD_ADDRESS_FILE This setting is described above in section 3.4.4 as SUBSYS_ADDRESS_FILE

3.4.10 condor_shadow Config File Entries

This setting effects the *condor_shadow*

MAX_DISCARDED_RUN_TIME If the shadow is unable to read a checkpoint file from the checkpoint server, it keeps trying only if the job has accumulated more than this many seconds of CPU usage. Otherwise, the job is started from scratch. Defaults to 3600 (1 hour). This setting is only used if USE_CKPT_SERVER is True.

SHADOW_LOCK This entry specifies what lock file should be used for access to the ShadowLog file. It must be a separate file from the ShadowLog, since the ShadowLog may be rotated and you want to be able to synchronize access across log file rotations. This macro is defined relative to the LOCK macro described above. If, for some strange reason, you decide to change this setting, be sure to change the VALID_LOG_FILES entry that *condor_preen* uses as well.

SHADOW_DEBUG This setting (and other settings related to debug logging in the shadow) is described above in section 3.4.3 as SUBSYS_DEBUG.

3.4.11 condor_shadow.pvm Config File Entries

These settings control the *condor_shadow.pvm*, the special shadow that supports PVM jobs inside Condor. See section **??** "Installing PVM Support in Condor" for details.

PVMD This macro holds the full path to the special *condor_pvmd*, the Condor PVM Daemon. This daemon is installed in the regular Condor release directory by default, so the macro is usually defined in terms of $(SBIN).

PVMGS This macro holds the full path to the special *condor_pvmgs*, the Condor PVM Group Server Daemon, which is needed to support PVM groups. This daemon is installed in the regular Condor release directory by default, so the macro is usually defined in terms of $(SBIN).

SHADOW_DEBUG This setting (and other settings related to debug logging in the shadow) is described above in section 3.4.3 as SUBSYS_DEBUG.

3.4.12 condor_starter Config File Entries

This setting effects the *condor_starter*.

JOB_RENICE_INCREMENT When the starter spawns a Condor job, it can do so with a *nice-level*. This is a mechanism in UNIX where you can assign your own processes a lower priority so that they don't interfere with interactive use of the machine. If you have machines with lots of real memory and swap space so the only scarce resource is CPU time, you could use this setting in conjunction with a policy that always allowed Condor to start jobs on your machines so that Condor jobs would always run, but interactive response on your machines would never suffer. You probably wouldn't even notice Condor was running jobs. See section 3.5 on "Configuring The Startd Policy" for full details on setting up a policy for starting and stopping jobs on a given machine. The entry can be any integer between 1 and 19. It is commented out by default.

STARTER_LOCAL_LOGGING This macro determines whether the starter should do local logging to its own log file, or send debug information back to the *condor_shadow* where it will end up in the ShadowLog. It defaults to "True"

STARTER_DEBUG This setting (and other settings related to debug logging in the starter) is described above in section 3.4.3 as SUBSYS_DEBUG.

3.4.13 condor_submit Config File Entries

If you want *condor_submit* to automatically append an expression to the Requirements expression or Rank expression of jobs at your site use the following entries:

APPEND_REQ_VANILLA Expression to append to vanilla job requirements.

APPEND_REQ_STANDARD Expression to append to standard job requirements.

APPEND_RANK_STANDARD Expression to append to vanilla job rank.

APPEND_RANK_VANILLA Expression to append to standard job rank.

IMPORTANT NOTE: The APPEND_RANK_STANDARD and APPEND_RANK_VANILLA macros were called "APPEND_PREF_STANDARD" and "APPEND_PREF_VANILLA" in previous versions of Condor.

In addition, you can provide default Rank expressions if your users don't specify their own:

DEFAULT_RANK_VANILLA Default Rank for vanilla jobs.

DEFAULT_RANK_STANDARD Default Rank for standard jobs.

Both of these macros default to the jobs preferring machines where there is more main memory than the image size of the job, expressed as:

```
((Memory*1024) > Imagesize)
```

3.4.14 condor_preen Config File Entries

These settings control *condor_preen*.

PREEN_ADMIN This entry determines what email address *condor_preen* will send email to (if it's configured to send email at all... see the entry for PREEN above). Defaults to $(CONDOR_ADMIN).

VALID_SPOOL_FILES This entry contains a (comma or space separated) list of files that *condor_preen* considers valid files to find in the SPOOL directory. Defaults to all the files that are valid. If you change the HISTORY setting above, you'll want to change this setting as well.

VALID_LOG_FILES This entry contains a (comma or space separated) list of files that *condor_preen* considers valid files to find in the LOG directory. Defaults to all the files that are valid. If you change the names of any of the log files above, you'll want to change this setting as well. In addition the defaults for the SUBSYS_ADDRESS_FILE are listed here, so if you change those, you'll need to change this entry, too.

3.4.15 condor_collector Config File Entries

These settings control the *condor_collector*.

CLASSAD_LIFETIME This macro determines how long a ClassAd can remain in the collector before it is discarded as stale information. The ClassAds sent to the collector might also have an attribute that says how long the lifetime should be for that specific ad. If that attribute is present the collector will either use it or the CLASSAD_LIFETIME, whichever is greater. The macro is defined in terms of seconds, and defaults to 900 (15 minutes).

`MASTER_CHECK_INTERVAL` This setting defines often the collector should check for machines that have ClassAds from some daemons, but not from the *condor_master* (*orphaned daemons*) and send email about it. Defined in seconds, defaults to 10800 (3 hours)

`CLIENT_TIMEOUT` Network timeout when talking to daemons that are sending an update. Defined in seconds, defaults to 30.

`QUERY_TIMEOUT` Network timeout when talking to anyone doing a query. Defined in seconds, defaults to 60.

`CONDOR_DEVELOPERS` Condor will send email once per week to this address with the output of the *condor_status* command, which simply lists how many machines are in the pool and how many are running jobs. Use the default value of "condor-admin@cs.wisc.edu". This default will send the weekly status message to the Condor Team at University of Wisconsin-Madison, the developers of Condor. The Condor Team uses these weekly status messages in order to have some idea as to how many Condor pools exist in the world. We would really appreciate getting the reports as this is one way we can convince funding agencies that Condor is being used in the "real world". If you do not wish this information to be sent to the Condor Team, you could enter "NONE" which disables this feature, or put in some other address that you want the weekly status report sent to.

`COLLECTOR_NAME` The parameter is used to specify a short description of your pool. It should be about 20 characters long. For example, the name of the UW-Madison Computer Science Condor Pool is "UW-Madison CS".

`CONDOR_DEVELOPERS_COLLECTOR` By default, every pool sends periodic updates to a central *condor_collector* at UW-Madison with basic information about the status of your pool. This includes only the number of total machines, the number of jobs submitted, the number of machines running jobs, the hostname of your central manager, and the `COLLECTOR_NAME` specified above. These updates help us see how Condor is being used around the world. By default, they will be sent to condor.cs.wisc.edu. If you don't want these updates to be sent from your pool, set this entry to "NONE".

`COLLECTOR_DEBUG` This setting (and other settings related to debug logging in the collector) is described above in section 3.4.3 as `SUBSYS_DEBUG`.

3.4.16 condor_negotiator Config File Entries

These settings control the *condor_negotiator*.

`NEGOTIATOR_INTERVAL` How often should the negotiator start a negotiation cycle? Defined in seconds, defaults to 300 (5 minutes).

`NEGOTIATOR_TIMEOUT` What timeout should the negotiator use on it's network connections to the schedds and startds? Defined in seconds, defaults to 30.

`PRIORITY_HALFLIFE` This entry defines the half-life of the user priorities. See section 2.8.2 on User Priorities for more details. Defined in seconds, defaults to 86400 (1 day).

`PREEMPTION_HOLD` If the `PREEMPTION_HOLD` expression evaluates to true, the *condor_negotiator* won't preempt the job running on a given machine even if a user with a higher priority has jobs they want to run. This helps prevents thrashing. The default is to wait 2 hours before preempting any job.

`NEGOTIATOR_DEBUG` This setting (and other settings related to debug logging in the negotiator) is described above in section 3.4.3 as `SUBSYS_DEBUG`.

3.5 Configuring The Startd Policy

This section describes how to configure the *condor_startd* to implement the policy you choose for when remote jobs should start, be suspended, (possibly) resumed, vacated (with a checkpoint) or killed (no checkpoint). This policy is the heart of Condor's balancing act between the needs and wishes of resource owners (machine owners) and resource users (people submitting their jobs to Condor). Please read this section carefully if you plan to change any of the settings described below, as getting it wrong can have a severe impact on either the owners of machines in your pool (in which case they might ask to be removed from the pool entirely) or the users of your pool (in which case they might stop using Condor).

Much of this section refers to ClassAd expressions. You probably want to read through section 4.1 on ClassAd expressions before continuing with this.

To define your policy, you basically set a bunch of expressions in the config file (see section 3.4 on "Configuring Condor" for an introduction to Condor's config files). These expressions are evaluated in the context of the startd's ClassAd and the ClassAd of a potential resource request (a job that has been submitted to Condor). The expressions can therefore reference attributes from either ClassAd. First, we'll list all the attributes that are included in the Startd's ClassAd. Then, we'll list all the attributes that are included in a job ClassAd. Next, we'll explain the the `START` expression, which describes to Condor what conditions must be met for the machine to start a job. Then, we'll describe the `RANK` expression, which allows you to specify which kinds of jobs a given machine prefers to run. Then, we'll discuss in some detail how the *condor_startd* works, in particular, the Startd's *states* and *activities*, to give you an idea of what is possible for your policy decisions. Finally, we offer two example policy settings.

3.5.1 Startd ClassAd Attributes

The *condor_startd* represents the machine on which it is running to the Condor pool. It publishes a number of characteristics about the machine in its ClassAd to help in match-making with resource requests. The values of all these attributes can be found by using *condor_status -l hostname*. The attributes themselves and what they represent are described below:

Activity : String which describes Condor job activity on the machine. Can have one of the following values:

- **"Idle"** : There is no job activity
- **"Busy"** : A job is busy running
- **"Suspended"** : A job is currently suspended
- **"Vacating"** : A job is currently checkpointing
- **"Killing"** : A job is currently being killed
- **"Benchmarking"** : The startd is running benchmarks

AFSCell : If the machine is running AFS, this is a string containing the AFS cell name.

Arch : String with the architecture of the machine. Typically one of the following:

- **"INTEL"** : Intel CPU (Pentium, Pentium II, etc).
- **"ALPHA"** : Digital ALPHA CPU
- **"SGI"** : Silicon Graphics MIPS CPU
- **"SUN4u"** : Sun ULTRASPARC CPU
- **"SUN4x"** : A Sun SPARC CPU other than an ULTRASPARC, i.e. sun4m or sun4c CPU found in older SPARC workstations such as the Sparc~10, Sparc~20, IPC, IPX, etc.
- **"HPPA1"** : Hewlett Packard PA-RISC 1.x CPU (i.e. PA-RISC 7000 series CPU) based-workstation
- **"HPPA2"** : Hewlett Packard PA-RISC 2.x CPU (i.e. PA-RISC 8000 series CPU) based-workstation

ClockDay : The day of the week, where 0 = Sunday, 1 = Monday, . . ., 6 = Saturday.

ClockMin : The number of minutes passed since midnight.

CondorLoadAvg : The load average generated by Condor (either from remote jobs or running benchmarks).

ConsoleIdle : The number of seconds since activity on the system console keyboard or console mouse has last been detected.

Cpus : Number of CPUs in this machine, i.e. 1 = single CPU machine, 2 = dual CPUs, etc.

CurrentRank : A float which represents this machine owner's affinity for running the Condor job which it is currently hosting. If not currently hosting a Condor job, CurrentRank is -1.0.

Disk : The amount of disk space on this machine available for the job in kbytes (e.g. 23000 = 23 megabytes). Specifically, this is amount of disk space available in the directory specified in the Condor configuration files by the macro `EXECUTE`, minus any space reserved with the macro `RESERVED_DISK`.

EnteredCurrentActivity : Time at which the machine entered the current Activity (see `Activity` entry above). Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970).

FileSystemDomain : a domain name configured by the Condor administrator which describes a cluster of machines which all access the same networked filesystems usually via NFS or AFS.

KeyboardIdle : The number of seconds since activity on any keyboard or mouse associated with this machine has last been detected. Unlike `ConsoleIdle`, `KeyboardIdle` also takes activity on pseudo-terminals into account (i.e. virtual "keyboard" activity from telnet and rlogin sessions as well). Note that `KeyboardIdle` will always be equal to or less than `ConsoleIdle`.

KFlops : Relative floating point performance as determined via a linpack benchmark.

LastHeardForm : Time when the Condor Central Manager last received a status update from this machine. Expressed as seconds since the epoch.

LoadAvg : A floating point number with the machine's current load average.

Machine : A string with the machine's fully qualified hostname.

Memory : The amount of RAM in megabytes.

Mips : Relative integer performance as determined via a dhrystone benchmark.

MyType : The ClassAd type; always set to the literal string "Machine".

Name : The name of this resource; typically the same value as the `Machine` attribute, but could be customized by the site administrator.

OpSys : String describing the operating system running on this machine. For Condor Version 6.0.3 typically one of the following:

- "HPUX10" (for HPUX 10.20)
- "IRIX6" (for IRIX 6.2, 6.3, or 6.4)
- "LINUX" (for LINUX 2.x kernel systems)
- "OSF1" (for Digital Unix 4.x)
- "SOLARIS251"
- "SOLARIS26"

Requirements : A boolean which, when evaluated within the context of the Machine ClassAd and a Job ClassAd, must evaluate to TRUE before Condor will allow the job to use this machine.

StartdIpAddr : String with the IP and port address of the *condor_startd* daemon which is publishing this Machine ClassAd.

State : String which publishes the machine's Condor state, which can be:

- **"Owner"** : The machine owner is using the machine, and it is unavailable to Condor.
- **"Unclaimed"** : The machine is available to run Condor jobs, but a good match (i.e. job to run here) is either not available or not yet found.
- **"Matched"** : The Condor Central Manager has found a good match for this resource, but a Condor scheduler has not yet claimed it.
- **"Claimed"** : The machine is claimed by a remote *condor_schedd* and is probably running a job.
- **"Preempting"** : A Condor job is being preempted (possibly via checkpointing) in order to clear the machine for either a higher priority job or because the machine owner wants the machine back.

TargetType : Describes what type of ClassAd to match with. Always set to the string literal "Job", because Machine ClassAds always want to be matched with Jobs, and vice-versa.

UidDomain : a domain name configured by the Condor administrator which describes a cluster of machines which all have the same "passwd" file entries, and therefore all have the same logins.

VirtualMemory : The amount of currently available virtual memory (swap space) expressed in kbytes.

3.5.2 Job ClassAd Attributes

This section has not yet been written

3.5.3 condor_startd START expression

The most important expression in the startd (and possibly in all of Condor) is the startd's `START` expression. This expression describes what conditions must be met for a given startd to service a resource request (in other words, start someone's job). This expression (like any other expression) can reference attributes in the startd's ClassAd (such as `KeyboardIdle`, `LoadAvg`, etc), or attributes in a potential requester's ClassAd (such as `Owner`, `Imagesize`, even `Cmd`, the name of the executable the requester wants to run). What the `START` expression evaluates to plays a crucial role in determining what state and activity the startd is in.

It is technically the `Requirements` expression that is used for matching with other jobs. The startd just always defines the `Requirements` expression as the `START` expression. However, in situations where the startd wants to make itself unavailable for further matches,

it sets its `Requirements` expression to False, not its `START` expression. When the `START` expression *locally evaluates* to true, the startd advertises the `Requirements` expression as "True" and doesn't even publish the `START` expression.

Normally, the expressions in the startd ClassAd are evaluated against certain request ClassAds in the *condor_negotiator* to see if there is a match, or against whatever request ClassAd currently has claimed the startd. However, by locally evaluating an expression, the startd only evaluates the expression against its own ClassAd. If an expression cannot be locally evaluated (because it references other expressions that are only found in a request ad, such as `Owner` or `Imagesize`), the expression is (usually) undefined. See the ClassAd appendix for specifics of how undefined terms are handled in ClassAd expression evaluation.

NOTE: If you have machines with lots of real memory and swap space so the only scarce resource is CPU time, you could use the `JOB_RENICE_INCREMENT` (see section 3.4.12 on "condor_starter Config File Entries" for details) so that Condor starts jobs on your machine with low priority. Then, you could set up your machines with:

```
START : True
SUSPEND : False
VACATE : False
KILL : False
```

This way, Condor jobs would always run and would never be kicked off. However, because they would run with "nice priority", interactive response on your machines would not suffer. You probably wouldn't even notice Condor was running the jobs, assuming you had enough free memory for the Condor jobs so that you weren't swapping all the time.

3.5.4 condor_startd RANK expression

A startd can be configured to prefer running certain jobs over other jobs. This is done via the `RANK` expression. This is an expression, just like any other in the startd's ClassAd. It can reference any attribute found in either the startd ClassAd or a request ad (normally, in fact, it references things in the request ad). Probably the most common use of this expression is to configure a machine to prefer to run jobs from the owner of that machine, or by extension, a group of machines to prefer jobs from the owners of those machines.

For example, imagine you have a small research group with 4 machines: "tenorsax", "piano", "bass" and "drums". These machines are owned by 4 users: "coltrane", "tyner", "garrison" and "jones", respectively.

Say there's a large Condor pool in your department, but you spent a lot of money on really fast machines for your group. You want to make sure that if anyone in your group has Condor jobs, they have priority on your machines. To achieve this, all you have to do is set the Rank expression on your machines to refer to the `Owner` attribute and prefer requests where that attribute matches one of the people in your group:

```
RANK : Owner == "coltrane" || Owner == "tyner" \
       || Owner == "garrison" || Owner == "jones"
```

The RANK expression is evaluated as a floating point number. However, just like in C, boolean expressions evaluate to either 1 or 0 depending on if they're true or false. So, if this expression evaluated to 1 (because the remote job was owned by one of the blessed folks), that would be higher than anyone else (for whom the expression would evaluate to 0).

If you wanted to get really fancy, you could still have the same basic setup, where anyone from your group has priority on your machines, but the actual machine owner has even more priority on their own machine. For example, you'd put the following entry in Jimmy Garrison's local config file `bass.local`:

```
RANK : Owner == "coltrane" + Owner == "tyner" \
       + (Owner == "garrison") * 10 + Owner == "jones"
```

Notice, we're using "+" instead of "| | ", since we want to be able to distinguish which terms matched and which ones didn't. Now, if anyone who wasn't in the John Coltrane quartet was running a job on "bass", the RANK would evaluate numerically to 0, since none of those boolean terms would evaluate to 1, and 0+0+0+0 is still 0. Now, suppose Elvin Jones submits a job. His job would match this machine (assuming the START was true for him at that time) and the RANK would numerically evaluate to 1 (since one of the boolean terms would evaluate to 1), so Elvin would preempt whoever else was using the machine at the time. After a while, say Jimmy decides to submit a job (maybe even from another machine, it doesn't matter, all that matters is that it's Jimmy's job). Now, the RANK would evaluate to 10, since the boolean that matches him gets multiplied by 10. So, Jimmy would preempt even Elvin, and his job would run on his machine.

The RANK expression doesn't just have to refer to the Owner of the jobs. Suppose you have a machine with a ton of memory, and others with not much at all. You could configure your big-memory machine to prefer to run jobs with bigger memory requirements:

```
RANK : ImageSize
```

That's all there is to it. The bigger the job, the more this machine wants to run it. That's pretty altruistic of you, always servicing bigger and bigger jobs, even if they're not yours. So, perhaps you still want to be a nice guy, all else being equal, but if you have jobs, you want to run them, regardless of everyone else's Imagesize:

```
RANK : (Owner == "coltrane" * 1000000000000) + Imagesize
```

This scheme would break down if someone submitted a job with an image size of more 10ˆ12 kbytes. However, if they did, this Rank expression preferring their job over yours wouldn't be the only problem Condor had *grin*

3.5.5 condor_startd States

The *condor_startd* could be in a number of different *states*, depending on whether or not the machine is available to run Condor jobs, and if so, what stage in the Condor protocol has been reached. The possible states are:

Owner The machine is being used by the machine owner, or at least is not available to run Condor jobs. When the startd first starts up, it begins in this state.

Unclaimed The machine is available to run Condor jobs, but is not currently doing so in any way.

Matched The machine is available to run jobs, and has been matched by the negotiator with a given schedd. That schedd just hasn't claimed this startd yet. In this state, the startd is unavailable for further matches.

Claimed The machine has been claimed by a schedd.

Preempting The machine was claimed by a schedd, but is now preempting that claim because either the owner of the machine came back, the negotiator decided to preempt this match because another user with higher priority has jobs waiting to run, or the negotiator decided to preempt this match because it found another request that this resource would rather serve (see the RANK expression below).

See figure 3.2 on page 105 for the various states and the possible transitions between them.

3.5.6 condor_startd Activities

Within some of these states, there could be a number of different *activities* the startd is in. The idea is that all the things that are true about a given state are true regardless of what activity you are in. However, there are certain important differences between each activity, which is why they are separated out from each other within a given state. In general, you must specify both a state and an activity to describe what "state" the startd is in. This will be denoted in this manual as "state/activity" pairs. For example, "Claimed/Busy". The following list describes all the possible state/activity pairs:

- Owner

  Idle This is the only activity for Owner state. As far as Condor is concerned the machine is "Idle" (not doing anything for Condor).

- Unclaimed

  Idle This is the normal activity of Unclaimed machines. The machine is still "Idle" in that the machine owner is willing to let someone run jobs on it, but Condor is still not using the machine for anything.

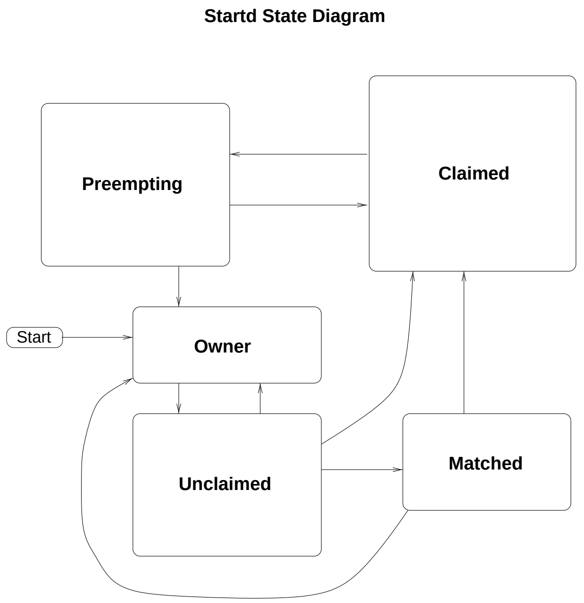

Figure 3.2: Startd States

Benchmarking The startd could also be running benchmarks to determine the speed on this machine. It only does this when the machine is in the Unclaimed state. How often it does so is determined by the `RunBenchmarks` expression described below.

- Matched

Idle When Matched, the machine is still "Idle" as far as Condor is concerned.

- Claimed

Idle In this activity, the startd has been claimed, but the schedd that claimed it has yet to *activate* the claim by requesting a *condor_starter* to be spawned with would service a given job.

Busy Once a *condor_starter* has been started and the claim is active, the startd moves to the Busy activity to signify that it's actually doing something as far as Condor is concerned.

Suspended If the job (and it's *condor_starter* is suspended by Condor, the startd goes into the Suspended activity. The match between the schedd and startd has not been broken (the claim is still valid), but the job is not making any progress and Condor is no longer generating a load on the machine.

- Preempting

Vacating Vacating simply means that the job that was running is in the process of checkpointing. As soon as the checkpoint process completes, the startd moves into either the Owner state or the Claimed state, depending on why it began preempting in the first place.

Killing Killing means that the startd has requested the running job to exit the machine immediately, without checkpointing.

NOTE: It is by the activity that the startd keeps track of the *Condor Load Average*, which is the load average generated by Condor on the machine. We make the assumption that whenever the startd is in the following activities, it is generating a load average of 1.0: busy, benchmarking, vacating, killing. In all other activities (idle, suspended) it is not generating any load at all.

Figure 3.3 on page 107 gives the overall view of all startd states and activities, and shows all the possible transitions from one to another within the Condor system. This may seem pretty daunting, but it's actually easier to handle than it looks.

Various expressions are used to determine when and if many of these state and activity transitions occur. Other transitions are initiated by parts of the Condor protocol (such as when the *condor_negotiator* matches a startd with a schedd). The following section describes the conditions that lead to the various state and activity transitions.

3.5.7 condor_startd State and Activity Transitions

This section will trace through all possible state and activity transitions within the startd and describe the conditions under which each one occurs. Whenever a transition occurs, the startd records when it entered its new activity and/or new state. These times are often used to write the expressions that determine when further transitions occurred (for example, you might only enter the Killing activity if you've been in the Vacating activity longer than a given amount of time).

Owner State

When the startd is first spawned, it enters the Owner state. The startd will remain in this state as long as the `START` expression locally evaluates to false. So long as the `START` expression locally evaluates to false, there is no possible request in the Condor system that could match it, so the machine in unavailable to Condor and stays in the Owner state. For example, if the `START` expression was:

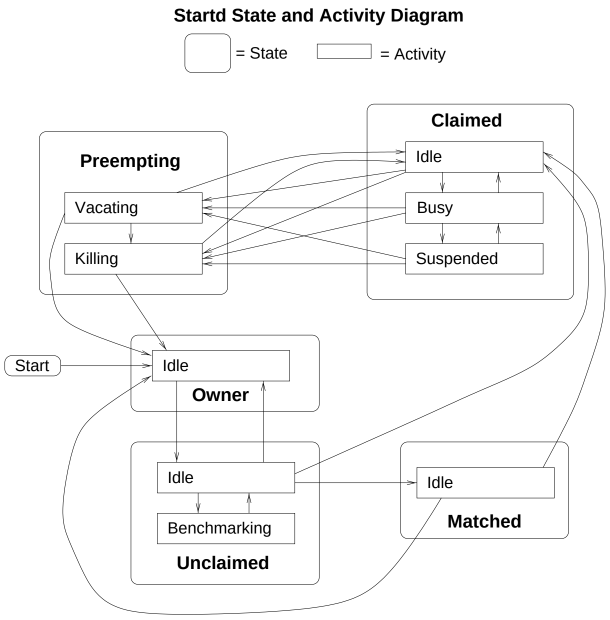

Figure 3.3: Startd States and Activities

```
START : KeyboardIdle > 15 * $(MINUTE) && Owner == "coltrane"
```

and if `KeyboardIdle` was only 34 seconds, then the machine would still be in the Owner state, even though it references Owner, which is undefined. `False && anything` is False, even `False && undefined`

If, however, the `START` expression was:

```
START : KeyboardIdle > 15 * $(MINUTE) || Owner == "coltrane"
```

and `KeyboardIdle` was still only 34 seconds, then the machine would leave the Owner state and go to Unclaimed. This is because "False —— undefined" is undefined. So, while this machine isn't available to just any body, if user "coltrane" has jobs submitted, the machine is willing to run them. Anyone else would have to wait until `KeyboardIdle` exceeds 15 minutes. However, since "coltrane" might claim this resource, but hasn't yet, the startd

goes to the Unclaimed state.

While in the Owner state the startd only polls the status of the machine every `UPDATE_INTERVAL` to see if anything has changed that would lead it to a different state. The idea is that you don't want to put much load on the machine while the Owner is using it (frequently waking up, computing load averages, checking the access times on files, computing free swap space, etc), and there's nothing time critical that the startd needs to be sure to notice as soon as it happens. If the `START` expression evaluates to True and it's 5 minutes before we notice it, that's a drop in the bucket of High Throughput Computing.

The startd can only go to the unclaimed state from the Owner state, and only does so when the `START` expression no longer locally evaluates to False. Generally speaking, if the `START` expression locally evaluates to false at any time, the startd will either transition directly to the Owner state, or to the Preempting state on its way to the Owner state, if there's a job running that needs preempting.

Unclaimed State

When it's in the Unclaimed state, another expression comes into effect, `RunBenchmarks` . Whenever the `RunBenchmarks` evaluates to True while the startd is in the Unclaimed state, the startd will transition from the Idle activity to the Benchmarking activity and perform benchmarks to determine `MIPS` and `KFLOPS`. The startd automatically inserts an attribute, `LastBenchmark`, whenever it runs benchmarks, so commonly `LastBenchmark` is defined in terms of this attribute, for example:

```
BenchmarkTimer = (CurrentTime - LastBenchmark)
RunBenchmarks : $(BenchmarkTimer) >= (4 * $(HOUR))
```

Here, a macro, `BenchmarkTimer` is defined to help write the expression. The idea is that this macro holds the time since the last benchmark, so when this time exceeds 4 hours, we run the benchmarks again. The startd keeps a weighted average of these benchmarking results to try to get the most accurate numbers possible. That's why you would want the startd to run them more than once in its lifetime.

NOTE: LastBenchmark is initialized to 0 before the benchmarks have ever been run. So, if you want the startd to run benchmarks as soon as it is is unclaimed if it hasn't done so already, just include a term for `LastBenchmark` as in the example above.

NOTE: If `RunBenchmarks` is defined, and set to something other than "False", the startd will automatically run one set of benchmarks when it starts up. So, if you want to totally disable benchmarks, both at startup, and at any time thereafter, just set `RunBenchmarks` to "False" or comment it out from your config file.

From the Unclaimed state, the startd can go to two other possible states: Matched or Claimed/Idle. Once the *condor_negotiator* matches an Unclaimed startd with a requester at a given schedd, the negotiator sends a command to both parties, notifying them of the match. If the schedd gets that notification and initiates the claiming procedure with the startd before the negotiator's message gets to the startd, the Match state is skipped entirely, and the startd goes directly to the Claimed/Idle state. However, normally, the startd will enter the Matched state, even if it's only for a brief period of time.

Matched State

The Matched state is not very interesting to Condor. The only noteworthy things are that the Startd lies about its `START` expression while in this state and says that `Requirements` are false to prevent being matched again before it has been claimed, and that the startd starts a timer to make sure it doesn't stay in the Matched state too long. This timer is set with the `MATCH_TIMEOUT` config file parameter. It is specified in seconds and defaults to 300 (5 minutes). If the schedd that was matched with this startd doesn't claim it within this period of time, the startd gives up on it, goes back into the Owner state (which it will probably leave right away to get to the Unclaimed state again, and wait for another match).

At any time while the startd is in the Matched state, if the `START` expression locally evaluates to false, the startd enters the Owner state directly.

If the schedd that was matched with the startd claims it before the `MATCH_TIMEOUT` expires, the startd goes into the Claimed/Idle state.

Claimed State

The Claimed state is certainly the most complicated State in the startd. It has the most possible activities, and the most expressions that determine what it will do next. In addition the *condor_checkpoint* and *condor_vacate* commands only have any effect on the startd when its in the Claimed state. In general, there are two sets of expressions that take effect, depending on if the universe of the request that claimed the startd is Standard or Vanilla. The Standard Universe expressions are the "normal" expressions, for example:

```
WANT_SUSPEND            : True
WANT_VACATE             : True
SUSPEND                 : $(KeyboardBusy) || $(CPUBusy)
...
```

The Vanilla expressions have "VANILLA" appended to the end, for example:

```
WANT_SUSPEND_VANILLA    : True
WANT_VACATE_VANILLA     : True
SUSPEND_VANILLA         : $(KeyboardBusy) || $(CPUBusy)
...
```


For the purposes of this manual, we'll refer to the regular expressions. Keep in mind that if the request was a Vanilla Universe, the Vanilla expressions would be in effect, instead. The reason for this is that the resource owner might want the startd to behave differently for Vanilla jobs, since they can't checkpoint. For example, they might want to let Vanilla jobs remain suspended for much longer than standard jobs.

While Claimed, the `POLLING_INTERVAL` takes effect, and the startd starts polling the machine much more frequently to evaluate its state. If the owner starts typing on the console again, we want to notice as soon as possible and start doing whatever that owner wants at that point.

In general, when the startd is going to kick off a job (usually because of activity on the machine that signifies that the owner is using the machine again) the startd will go through successive levels of getting the job out of the way. The first and least costly to the job is suspending it. This even works for Vanilla jobs. If suspending the job for a little while doesn't satisfy the machine owner, (the owner is still using the machine after a certain period of time, for example), the startd moves on to vacating the job, which involves performing a checkpoint so that the work it had completed up until this point is not lost. If even that does not satisfy the machine owner (usually because its taking too long and the owner wants their machine back *now*), the final, most drastic stage is reached: killing. Killing is just quick death to the job, without a checkpoint or anything. For Vanilla jobs, vacating and killing are basically equivalent, though a vanilla job can request to have a certain *softkill signal* sent to it at vacate time so that it can perform application-specific checkpointing, for example.

The `WANT_SUSPEND` expression determines if the startd will even evaluate the `SUSPEND` expression to consider entering the Suspended activity. Similarly, the `WANT_VACATE` expression determines if the startd will even evaluate the `VACATE` expression to consider entering Preempting/Vacating. If one or both of these expressions evaluates to false, the startd will skip that stage of getting rid of the job and proceed directly to the more drastic stages.

When the startd first enters the Claimed state, it goes to the Idle activity. From there, it can transition either to the Preempting state (if a *condor_vacate* comes in, or if the `START` expression locally evaluates to false). Or, it can transition to the busy activity if the schedd that has claimed the startd decides to activate the claim and start a job.

From Claimed/Busy, the startd can go to many different state/activity combinations.

Claimed/Idle If the starter that is serving a given job exits (because the jobs completes, for example), the startd will go back to Claimed/Idle.

Claimed/Suspended If both the `WANT_SUSPEND` and `SUSPEND` expressions evaluate to true, the startd will suspend the job. `WANT_SUSPEND` basically determines if the startd should even consider the `SUSPEND` expression. If `WANT_SUSPEND` is false, the startd will look at other expressions instead and skip the Suspended activity entirely.

Preempting/Vacating If `WANT_SUSPEND` is false and `WANT_VACATE` is true, and the `VACATE` expression is true, the startd will enter the Preempting/Vacating state and start checkpointing the job.

The other reason the startd would go from Claimed/Busy to Preempting/Vacating is if the *condor_negotiator* matched the startd with a "better" match. This better match could either be from the startd's perspective (see section 3.5.4 on the `RANK` Expression above) or from the negotiator's perspective (because a user with a better user priority has jobs that should be running on this startd).

Preempting/Killing If `WANT_SUSPEND` is false and `WANT_VACATE` is false, and the `KILL` expression is true, the startd will enter the Preempting/Killing state and start killing the job (without a checkpoint).

Claimed/Busy While it's not really a state change, there is another thing that could happen to the startd while it's in Claimed/Busy, which is that either a *condor_checkpoint* command could arrive, or the `PeriodicCheckpoint` expression could evaluate to true. When either of these things occur, the startd requests that the job begin a periodic checkpoint. Since the startd has no way to know when this process completes, there's no way periodic checkpointing could be its own state. However, for the purposes of all the expressions and the Condor Load Average computations, periodic checkpointing is Claimed/Busy, just like a job was running.

You already know what happens in Claimed/Idle, so now we'll discuss what happens in Claimed/Suspended. Again, there are multiple state/activity combinations that you can reach from Claimed/Suspended:

Preempting/Vacating If `WANT_VACATE` is true, and the `VACATE` expression is true, the startd will enter the Preempting/Vacating state and start checkpointing the job.

Preempting/Killing If `WANT_VACATE` is false, and the `KILL` expression is true, the startd will enter the Preempting/Killing state and start killing the job (without a checkpoint).

Claimed/Busy If the `CONTINUE` expression evaluates to true, the startd will resume the computation and will go back to the Claimed/Busy state.

From the Claimed state, you can only enter the Owner state, other activities in the Claimed state (all of which we've already discussed), or the Preempting state, which is described next.

Preempting State

The Preempting state is much less complicated than the Claimed state. Basically, there are two possible activities, and two possible destinations. Depending on `WANT_VACATE` you either enter the Vacating activity (if it's true) or the Killing activity (if it's false).

While in the Preempting state (regardless of activity) the startd advertises its `Requirements` expression as False to signify that it is not available for further matches,

either because it is about to go to the owner state anyway, or because it has already been matched with one preempting match, and further preempting matches are disallowed until the startd has been claimed by the new match.

The main function of the Preempting state is to get rid of the starter associated with this resource. If the *condor_starter* associated with a given claim exits while the *condor_startd* is still in the Vacating activity, it means the job successfully completed its checkpoint.

If the startd is in the Vacating activity, it keeps evaluating the `KILL` expression. As soon as this expression evaluates to true, the startd enters the Killing activity.

When the starter exits, or if there was no starter running when the startd enters the Preempting state (because it came from Claimed/Idle), the other job of the preempting state is completed: notifying the schedd that had claimed this startd that the claim is broken.

At this point, the startd will either enter the Owner state (if the job was preempted because the machine owner came back) or the Claimed/Idle state (if the job was preempted because a better match was found).

Then the startd enters the Killing activity, it begins a timer, the length of which is defined by the `KILLING_TIMEOUT` macro. This macro is defined in seconds and defaults to 30. If this timer expires and the startd is still in the Killing activity, something has gone seriously wrong with the *condor_starter* and the startd tries to vacate the job immediately by sending SIGKILL to all of the *condor_starter*'s children, and then to the *condor_starter* itself. After this, the startd enters the Owner state.

3.5.8 condor_startd State/Activity Transition Expression Summary

The following section is meant to summarize the information from the previous sections to serve as a quick reference. If anything is unclear here, please refer to the previous sections for clarification.

`START` When this is true, the startd is willing to spawn a remote Condor job.

`RunBenchmarks` While in the Unclaimed state, the startd will run benchmarks whenever this is true.

`MATCH_TIMEOUT` If the startd has been in the Matched state longer than this, it will go back to the Owner state.

`WANT_SUSPEND` If this is true, the startd will evaluate the `SUSPEND` expression to see if it should transition to the Suspended activity. If this is false, the startd will look at either the `VACATE` or `KILL` expression, depending on the value of `WANT_VACATE`.

`WANT_VACATE` If this is true, the startd will evaluate the `VACATE` expression to determine if it should transition to the Preempting/Vacating state. If this is false, the startd will evaluate the `KILL` expression to determine when it should transition to the Preempting/Killing state.

`SUSPEND` If `WANT_SUSPEND` is true, and the startd is in the Claimed/Busy state, it will enter the Suspended activity if `SUSPEND` is true.

`CONTINUE` If the startd is in the Claimed/Suspended state, it will enter the Busy activity if `CONTINUE` is true.

`VACATE` If `WANT_VACATE` is true, and the startd is either in the Claimed/Suspended activity, or is in the Claimed/Busy activity and the `WANT_SUSPEND` is false, the startd will enter the Preempting/Vacating state whenever `VACATE` is true.

`KILL` If `WANT_VACATE` is false, and the startd is either in the Claimed/Suspended activity, or is in the Claimed/Busy activity and the `WANT_SUSPEND` is false, the startd will enter the Preempting/Killing state whenever `KILL` is true.

`KILLING_TIMEOUT` If the startd is in the Preempting/Killing state for longer than `KILLING_TIMEOUT` seconds, the startd will just send a SIGKILL to the *condor_starter* and all its children to try to kill the job as quickly as possible.

`PERIODIC_CHECKPOINT` If the startd is in the Claimed/Busy state and `PERIODIC_CHECKPOINT` is true, the startd will begin a periodic checkpoint.

`RANK` If this expression evaluates to a higher number for a pending resource request than it does for the current request, the startd will preempt the current request (enter the Preempting/Vacating state). When the preemption is complete, the startd will enter the Claimed/Idle state with the new resource request claiming it.

3.5.9 Example Policy Settings

The following section provides two examples of how you might configure the policy at your pool. Each one is described in English, then the actual macros and expressions used are listed and explained with comments. Finally the entire set of macros and expressions are listed in one block so you can see them in one place for easy reference.

Default Policy Settings

These settings are the default as shipped with Condor. They have been used for many years with no problems. The Vanilla expressions are identical to the regular ones. (They aren't even listed here. If you don't define them, the regular expressions are used for Vanilla jobs as well).

First, we define a bunch of macros which help us write the expressions more clearly. In particular, we use:

`StateTimer` How long we've been in the current state.

`ActivityTimer` How long we've been in the current activity.

`NonCondorLoadAvg` The difference of the system load and the Condor load (i.e the load generated by everything but Condor).

`BackgroundLoad` How much background load we're willing to have on our machine and still start a Condor job.

`BackgroundLoad` How much background load we're willing to have on our machine and still start a Condor job.

`HighLoad` If the `$(NonCondorLoadAvg)` goes over this, the CPU is "busy" and we want to start evicting the Condor job.

`StartIdleTime` How long the keyboard has to be idle before we'll start a job.

`ContinueIdleTime` How long the keyboard has to be idle before we'll resume a suspended job.

`MaxSuspendTime` How long we're willing to let the job be suspended before we move on to more drastic measures.

`MaxVacateTime` How long we're willing to let the job be checkpointing before we give up on it and have to kill it outright.

`KeyboardBusy` A boolean string that evaluates to true when the keyboard is being used.

`CPU_Idle` A boolean string that evaluates to true when the CPU is idle is being used.

`CPU_Busy` A boolean string that evaluates to true when the CPU is busy.

```
##  These macros are here to help write legible expressions:
MINUTE          = 60
HOUR            = (60 * $(MINUTE))
StateTimer      = (CurrentTime - EnteredCurrentState)
ActivityTimer   = (CurrentTime - EnteredCurrentActivity)

NonCondorLoadAvg        = (LoadAvg - CondorLoadAvg)
BackgroundLoad          = 0.3
HighLoad                = 0.5
StartIdleTime           = 15 * $(MINUTE)
ContinueIdleTime        = 5 * $(MINUTE)
MaxSuspendTime          = 10 * $(MINUTE)
MaxVacateTime           = 5 * $(MINUTE)

KeyboardBusy            = KeyboardIdle < $(MINUTE)
CPU_Idle                = $(NonCondorLoadAvg) <= $(BackgroundLoad)
CPU_Busy                = $(NonCondorLoadAvg) >= $(HighLoad)
```

Now, we define that we always want to suspend jobs, and if that's not enough, we'll always try to gracefully vacate.

```
WANT_SUSPEND            : True
WANT_VACATE             : True
```

Finally, we define the actual expressions. Start any job if the CPU is idle (as defined by our macro), and the keyboard has been idle long enough.

```
START           : $(CPU_Idle) && KeyboardIdle > $(StartIdleTime)
```

Suspend a job if either the CPU or Keyboard is busy.

```
SUSPEND         : $(CPU_Busy) || $(KeyboardBusy)
```

Continue a suspended job if the CPU is idle and the Keyboard has been idle for long enough.

```
CONTINUE        : $(CPU_Idle) && KeyboardIdle > $(ContinueIdleTime)
```

Vacate a job if we've been suspended for too long.

```
VACATE          : $(ActivityTimer) > $(MaxSuspendTime)
```

Kill a job if we've been vacating for too long.

```
KILL            : $(ActivityTimer) > $(MaxVacateTime)
```

Finally, define when we do periodic checkpointing. For small jobs, checkpoint every 6 hours. For larger jobs, only checkpoint ever 12 hours.

```
LastCkpt        = (CurrentTime - LastPeriodicCheckpoint)
PERIODIC_CHECKPOINT : ((ImageSize < 60000) && ($(LastCkpt) > \
       (6 * $(HOUR)))) || ($(LastCkpt) > (12 * $(HOUR)))
```

For clarity and reference, the entire set policy settings are included once more without comments:

```
##  These macros are here to help write legible expressions:
MINUTE          = 60
HOUR            = (60 * $(MINUTE))
```



```
StateTimer      = (CurrentTime - EnteredCurrentState)
ActivityTimer   = (CurrentTime - EnteredCurrentActivity)

NonCondorLoadAvg        = (LoadAvg - CondorLoadAvg)
BackgroundLoad          = 0.3
HighLoad                = 0.5
StartIdleTime           = 15 * $(MINUTE)
ContinueIdleTime        = 5 * $(MINUTE)
MaxSuspendTime          = 10 * $(MINUTE)
MaxVacateTime           = 5 * $(MINUTE)

KeyboardBusy            = KeyboardIdle < $(MINUTE)
CPU_Idle                = $(NonCondorLoadAvg) <= $(BackgroundLoad)
CPU_Busy                = $(NonCondorLoadAvg) >= $(HighLoad)

WANT_SUSPEND            : True
WANT_VACATE             : True

START           : $(CPU_Idle) && KeyboardIdle > $(StartIdleTime)
SUSPEND         : $(CPU_Busy) || $(KeyboardBusy)
CONTINUE        : $(CPU_Idle) && KeyboardIdle > $(ContinueIdleTime)
VACATE          : $(ActivityTimer) > $(MaxSuspendTime)
KILL            : $(ActivityTimer) > $(MaxVacateTime)

##
##  Periodic Checkpointing
LastCkpt        = (CurrentTime - LastPeriodicCheckpoint)
PERIODIC_CHECKPOINT : ((ImageSize < 60000) && ($(LastCkpt) > \
       (6 * $(HOUR)))) || ($(LastCkpt) > (12 * $(HOUR)))
```

UW-Madison CS Condor Pool Policy Settings

Due to a recent increase in the number of Condor users and the size of their jobs (many users here are submitting jobs with an `Imagesize` of over 100 megs!), we have had to customize our policy to try to handle this range of `Imagesize` better.

Basically, whether or not we suspend or vacate jobs is now a function of the `Imagesize` of the job that's currently running (which is defined in terms of kilobytes). We have divided the `Imagesize` into three possible categories, which we define with macros.

```
BigJob          = (ImageSize > (30 * 1024))
MediumJob       = (ImageSize <= (30 * 1024) && ImageSize >= (10 * 1024))
SmallJob        = (ImageSize < (10 * 1024))
```

Our policy can be summed up with the following few sentences: If the job is "small", it goes through the normal progression of suspend to vacate to kill based on the tried and true times. If the job is "medium", when the user comes back, we start vacating the job right away. The idea is that if we checkpoint immediately, all our pages are still in memory, checkpointing will be fast, and we'll free up memory pages as soon as we checkpoint. If we suspend, our pages will start getting swapped out and when we finally want to checkpoint (10 minutes later), we'll have to start swapping out the user's pages again, they'll see reduced performance, and checkpointing will take much longer. If the job is "big", don't even bother checkpointing, since we won't finish before the owner gets too upset and we might as well not even bother putting the wasted load on the network and checkpoint server.

We use many of the same macros defined above, so please read the previous section for details on these.

We only want to suspend jobs if they are "small", and we only want to vacate jobs that are "small" or "medium". We still want to always suspend Vanilla jobs, regardless of their size. In fact, Vanilla jobs still use the default settings described above.

```
WANT_SUSPEND            : $(SmallJob)
WANT_VACATE             : $(MediumJob) || $(SmallJob)
WANT_SUSPEND_VANILLA    : True
WANT_VACATE_VANILLA     : True
```

Now, we define the actual expressions. We actually do this with macros and simply define the expressions with the macros. This may seem really strange, but we do it because it makes it easier to do special customized settings (for example, for testing purposes) and still reference the very complicated defaults. There will be a brief example of this at the end of this section.

First, `START`, `SUSPEND` and `CONTINUE`, which are all just like they always were. However, notice that because `WANT_SUSPEND` is now different, only small jobs will get suspended (and only jobs that are suspended look at the `CONTINUE` expression).

```
CS_START        = $(CPU_Idle) && KeyboardIdle > $(StartIdleTime)
CS_SUSPEND      = $(CPU_Busy) || $(KeyboardBusy)
CS_CONTINUE     = (KeyboardIdle > $(ContinueIdleTime)) && $(CPU_Idle)
```

Since `WANT_SUSPEND` depends on `Imagesize`, our `VACATE` expression has to depend on size as well. If it's a small job, we'd be suspended, so we want to look at how long we've been suspened. However, for medium jobs, we want to vacate if the user is back (CPU or Keyboard is busy). There's no mention of large jobs here, since `WANT_VACATE` is false for those jobs.

```
CS_VACATE   = ($(MediumJob) && ($(CPU_Busy) || $(KeyboardBusy)))  \
            || ($(SmallJob) && ($(ActivityTimer) > $(MaxSuspendTime)))
```

For big jobs, we want to kill if the user is back (CPU or Keyboard is busy). In addition, since large jobs can get put into Preempting/Vacating because of negotiator preemption,

we want to make sure we're not taking too long to do that. Therefore, if we're currently Vacating and we've exceeded our MaxVacateTime, move on to killing. This last bit also covers small and medium jobs, since they'll be vacating already when they start looking at the `KILL` expression.

```
CS_KILL     = ($(BigJob) && ($(CPU_Busy) || $(KeyboardBusy))) \
            || ((Activity == "Vacating") && \
                    ($(ActivityTimer) > $(MaxVacateTime)))
```

Here's where we actually define the expressions in terms of our special macros:

```
START       : $(CS_START)
SUSPEND     : $(CS_SUSPEND)
CONTINUE    : $(CS_CONTINUE)
VACATE      : $(CS_VACATE)
KILL        : $(CS_KILL)
```

The Vanilla expressions are set to the old standard defaults.

```
SUSPEND_VANILLA    : $(CPU_Busy) || $(KeyboardBusy)
CONTINUE_VANILLA   : $(CPU_Idle) && KeyboardIdle > $(ContinueIdleTime)
VACATE_VANILLA     : $(ActivityTimer) > $(MaxSuspendTime)
KILL_VANILLA       : $(ActivityTimer) > $(MaxVacateTime)
```

Periodic checkpoint also takes image size into account. Since we kill large jobs right away at eviction time, we want to periodically checkpoint them more frequently (every 3 hours), since that's the only way they make forward progress. However, with all those large periodic checkpoints going on on so frequently, we don't want to bog down our network or our checkpoint server. So, we only periodic checkpoint small or medium jobs except every 12 hours, since they get the privilege of checkpointing at eviction time.

```
#
#  Periodic Checkpointing (uncomment to enable)
LastCkpt             = (CurrentTime - LastPeriodicCheckpoint)
PERIODIC_CHECKPOINT  : (($(LastCkpt) > (3 * $(HOUR))) \
      && $(BigJob)) || (($(LastCkpt) > (12 * $(HOUR))) && \
      ($(SmallJob) || $(MediumJob)))
```

For clarity and reference, the entire set of policy settings are included once more, without comments:

```
StateTimer      = (CurrentTime - EnteredCurrentState)
ActivityTimer   = (CurrentTime - EnteredCurrentActivity)
```

```
NonCondorLoadAvg   = (LoadAvg - CondorLoadAvg)
BackgroundLoad     = 0.3
HighLoad           = 0.5
StartIdleTime      = 15 * $(MINUTE)
ContinueIdleTime   = 5 * $(MINUTE)
MaxSuspendTime     = 10 * $(MINUTE)
MaxVacateTime      = 5 * $(MINUTE)

KeyboardBusy       = KeyboardIdle < $(MINUTE)
CPU_Idle           = $(NonCondorLoadAvg) <= $(BackgroundLoad)
CPU_Busy           = $(NonCondorLoadAvg) >= $(HighLoad)

BigJob       = (ImageSize > (30 * 1024))
MediumJob    = (ImageSize <= (30 * 1024) && ImageSize >= (10 * 1024))
SmallJob     = (ImageSize < (10 * 1024))

WANT_SUSPEND            : $(SmallJob)
WANT_VACATE             : $(MediumJob) || $(SmallJob)
WANT_SUSPEND_VANILLA    : True
WANT_VACATE_VANILLA     : True

CS_START    = $(CPU_Idle) && KeyboardIdle > $(StartIdleTime)
CS_SUSPEND  = $(CPU_Busy) || $(KeyboardBusy)
CS_CONTINUE = (KeyboardIdle > $(ContinueIdleTime)) && $(CPU_Idle)
CS_VACATE   = ($(MediumJob) && ($(CPU_Busy) || $(KeyboardBusy)))  \
            || ($(SmallJob) && ($(ActivityTimer) > $(MaxSuspendTime)))
CS_KILL     = ($(BigJob) && ($(CPU_Busy) || $(KeyboardBusy))) \
            || ((Activity == "Vacating") && \
                   ($(ActivityTimer) > $(MaxVacateTime)))

START       : $(CS_START)
SUSPEND     : $(CS_SUSPEND)
CONTINUE    : $(CS_CONTINUE)
VACATE      : $(CS_VACATE)
KILL        : $(CS_KILL)

SUSPEND_VANILLA   : $(CPU_Busy) || $(KeyboardBusy)
CONTINUE_VANILLA  : $(CPU_Idle) && KeyboardIdle > $(ContinueIdleTime)
VACATE_VANILLA    : $(ActivityTimer) > $(MaxSuspendTime)
KILL_VANILLA      : $(ActivityTimer) > $(MaxVacateTime)

#
#  Periodic Checkpointing (uncomment to enable)
LastCkpt             = (CurrentTime - LastPeriodicCheckpoint)
PERIODIC_CHECKPOINT  : (($(LastCkpt) > (3 * $(HOUR))) \
```



```
      && $(BigJob)) || (($(LastCkpt) > (12 * $(HOUR))) && \
      ($(SmallJob) || $(MediumJob)))
```

As a final example, we show how our default macros can be used to setup a given machine for testing. Suppose we want the machine to behave just like normal, but if user "coltrane" submits a job, we want that job to start regardless of what's happening on the machine, and we don't want the job suspended, vacated or killed. For example, we might know "coltrane" is just going to be submitting very short running programs to test something and he wants to see them execute right away. Anyway, we could configure any machine (or our whole pool, for that matter) with the following 5 expressions:

```
START      : ($(CS_START)) || Owner == "coltrane"
SUSPEND    : ($(CS_SUSPEND)) && Owner != "coltrane"
CONTINUE   : $(CS_CONTINUE)
VACATE     : ($(CS_VACATE)) && Owner != "coltrane"
KILL       : ($(CS_KILL)) && Owner != "coltrane"
```

Notice that you don't have to do anything special with the `CONTINUE` expression, since if Coltrane's jobs never suspend, they'll never even look at that expression.

3.6 DaemonCore

This section is a brief description of *DaemonCore*. DaemonCore is a library that is shared among most of the Condor daemons which provides common functionality. Currently, the following daemons use DaemonCore:

- *condor_master*
- *condor_startd*
- *condor_schedd*
- *condor_collector*
- *condor_negotiator*
- *condor_kbdd*

Most of DaemonCore's details are not interesting for administrators. However, DaemonCore does provide a uniform interface for the daemons to various UNIX signals, and provides a common set of command-line options that can be used to start up each daemon.

3.6.1 DaemonCore and UNIX signals

One of the most visible features DaemonCore provides for administrators is that all daemons which use it behave the same way on certain UNIX signals. The signals and the behavior DaemonCore provides are listed below:

SIGHUP Causes the daemon to reconfigure itself.

SIGTERM Causes the daemon to gracefully shutdown.

SIGQUIT Causes the daemon to quickly shutdown.

Exactly what "gracefully" and "quickly" means varies from daemon to daemon. For daemons with little or no state (the kbdd, collector and negotiator) there's no difference and both signals result in the daemon shutting itself down basically right away. For the master, graceful shutdown just means it asks all of its children to perform their own graceful shutdown methods, while fast shutdown means it asks its children to perform their own fast shutdown methods. In both cases, the master only exits once all its children have exited. In the startd, if the machine is not claimed and running a job, both result in an immediate exit. However, if the startd is running a job, graceful shutdown results in that job being checkpointed, while fast shutdown does not. In the schedd, if there are no jobs currently running (i.e. no *condor_shadow* processes), both signals result in an immediate exit. With jobs running, however, graceful shutdown means that the schedd asks each shadow to gracefully vacate whatever job it is serving, while fast shutdown results in a hard kill of every shadow with no chance of checkpointing.

For all daemons, "reconfigure" just means that the daemon re-reads its config file(s) and any settings that have changed take effect. For example, changing the level of debugging output, the value of timers that determine how often daemons perform certain actions, the paths to the binaries you want the *condor_master* to spawn, etc. See section 3.4 on page 74, "Configuring Condor" for full details on what settings are in the config files and what they do.

3.6.2 DaemonCore and Command-line Arguments

The other visible feature that DaemonCore provides to administrators is a common set of command-line arguments that all daemons understand. The arguments and what they do are described below:

-b Causes the daemon to start up in the background. When a DaemonCore process starts up with this option, disassociates itself from the terminal and forks itself so that it runs in the background. This is the default behavior for Condor daemons, and what you get if you specify no options at all.

-f Causes the daemon to start up in the foreground. Instead of forking, the daemon just runs in the foreground.

NOTE: when the *condor_master* starts up daemons, it does so with the -f option since it has already forked a process for the new daemon. That is why you will see -f in the argument list of all Condor daemons that the master spawns.

-c filename Causes the daemon to use the specified filename (you must use a full path) as its global config file. This overrides the `CONDOR_CONFIG` environment variable, and the regular locations that Condor checks for its config file: the condor user's home directory and `/etc/condor/condor_config`.

-p port Causes the daemon to bind to the specified port for its *command socket*. The master uses this option to make sure the *condor_collector* and *condor_negotiator* start up on the well-known ports that the rest of Condor depends on them using.

-t Causes the daemon to print out its error message to `stderr` instead of its specified log file. This option forces the -f option described above.

-v Causes the daemon to print out version information and exit.

-l directory Overrides the value of `LOG` as specified in your config files. Primarily, this option would be used with the *condor_kbdd* when it needs to run as the individual user logged into the machine, instead of running as root. Regular users would not normally have permission to write files into Condor's log directory. Using this option, they can override the value of `LOG` and have the *condor_kbdd* write its log file into a directory that the user has permission to write to.

-pidfile filename Causes the daemon to write out its PID, or process id number, to the specified file. This file can be used to help shutdown the daemon without searching through the output of the "ps" command.

Since daemons run with their current working directory set to the value of `LOG`, if you don't specify a full path (with a "/" to begin), the file will be left in the log directory. If you leave your pidfile in your log directory, you will want to add whatever filename you use to the `VALID_LOG_FILES` parameter, described in section 3.4.14 on page 96, so that *condor_preen* does not remove it.

-k filename Causes the daemon to read out a pid from the specified filename and send a SIGTERM to that process. The daemon that you start up with "-k" will wait until the daemon it is trying to kill has exited.

3.7 Setting Up IP/Host-Based Security in Condor

This section describes the mechanisms for setting up Condor's host-based security. This allows you to control what machines can join your Condor pool, what machines can find out information about your pool, and what machines within your pool can perform administrative commands. By default, Condor is configured to allow anyone to view or join your pool. You probably want to change that.

First, we discuss how the host-based security works inside Condor. Then, we list the different levels of access you can grant and what parts of Condor use which levels. Next, we describe how to configure your pool to grant (or deny) certain levels of access to various machines. Finally, we provide some examples of how you might configure your pool.

3.7.1 How does it work?

Inside the Condor daemons or tools that use DaemonCore (see section 3.6 on "DaemonCore" for details), most things are accomplished by sending commands to another Condor daemon. These commands are just an integer to specify which command, followed by any optional information that the protocol requires at that point (such as a ClassAd, capability string, etc). When the daemons start up, they register which commands they are willing to accept, what to do with them when they arrive, and what access level is required for that command. When a command comes in, Condor sees what access level is required, and then checks the IP address of the machine that sent the command and makes sure it passes the various allow/deny settings in your config file for that access level. If permission is granted, the command continues. If not, the command is aborted.

As you would expect, settings for the access levels in your global config file will affect all the machines in your pool. Settings in a local config file will only affect that specific machine. The settings for a given machine determine what other hosts can send commands to that machine. So, if you want machine "foo" to have administrator access on to machine "bar", you need to put "foo" in bar's config file access list, not the other way around.

3.7.2 Security Access Levels

The following are the various access levels that commands within Condor can be registered with:

`READ` Machines with `READ` access can read information from Condor. For example, they can view the status of the pool, see the job queue(s) or view user permissions. `READ` access does not allow for anything to be changed or jobs to be submitted. Basically, a machine listed with `READ` permission cannot join a condor pool - it can only view information about the pool.

`WRITE` Machines with `WRITE` access can write information to condor. Most notably, it means that it can join your pool by sending ClassAd updates to your central manager and can talk to the other machines in your pool to submit or run jobs. In addition, any machine with `WRITE` access can request the *condor_startd* to perform a periodic checkpoint on any job it is currently executing (after a periodic checkpoint, the job will continue to execute and the machine will still be claimed by whatever schedd had claimed it). This allows users on the machines where they submitted their jobs to use the *condor_checkpoint* command to get their jobs to periodically checkpoint, even if they don't have an account on the remote execute machine.

IMPORTANT: For a machine to join a condor pool, it must have `WRITE` permission **AND** `READ` permission! (Just `WRITE` permission is not enough).

`ADMINISTRATOR` Machines with `ADMINISTRATOR` access have special Condor administrator rights to the pool. This includes things like changing user priorities (with "*condor_userprio -set*"), turning Condor on and off ("*condor_off* <*machine*>"), asking Condor to reconfigure or restart itself, etc. Typically you would want only a couple machines in this list - perhaps the workstations where the Condor administrators or sysadmins typically work, or perhaps just your Condor central manager.

IMPORTANT: This is host-wide access we're talking about. So, if you grant `ADMINISTRATOR` access to a given machine, **ANY USER** on that machine now has `ADMINISTRATOR` rights. Therefore, you should grant `ADMINISTRATOR` access carefully.

`OWNER` This level of access is required for commands that the owner of a machine (any local user) should be able to use, in addition to the Condor administrators. For example the *condor_vacate* command that causes the *condor_startd* to vacate any running condor job is registered with `OWNER` permission, so that anyone can issue *condor_vacate* to the local machine they are logged into.

`NEGOTIATOR` This is a special access level that means the specified command must come from the Central Manager of your pool. The commands that have this access level are the ones that tell the *condor_schedd* to begin negotiating and that tell an available *condor_startd* that it has been matched to a *condor_schedd* with jobs to run.

3.7.3 Configuring your Pool

`NEGOTIATOR` permission is handled automatically by Condor, and there is nothing special you need to configure. The other four permissions, `READ`, `WRITE`, `ADMINISTRATOR` and `OWNER` need to be specified in the config files. See the section on Configuring Condor for details on where these files might be located, general information about how to set parameters, and how to reconfigure the Condor daemons.

`ADMINISTRATOR` access defaults to only your central manager. `OWNER` access defaults to the local machine, and any machines listed with `ADMINISTRATOR` access. You can probably leave that how it is. If you want other machines to have `OWNER` access, you probably want them to have `ADMINISTRATOR` access as well. By granting machines `ADMINISTRATOR` access, they would automatically have `OWNER` access, given how `OWNER` access is configured.

For these permissions, you can optionally list an ALLOW or a DENY.

- If you have an ALLOW, it means "only allow these machines". No ALLOW means allow anyone.

- If you have a DENY, it means "deny these machines". No DENY means to deny nobody.
- If you have both an ALLOW and a DENY, it means allow the machines listed in ALLOW except for the machines listed in DENY.

Therefore, the settings you might set are:

```
HOSTALLOW_READ = <machines>
HOSTDENY_READ = ...
HOSTALLOW_WRITE = ...
HOSTDENY_WRITE = ...
HOSTALLOW_ADMINISTRATOR = ...
HOSTDENY_ADMINISTRATOR = ...
HOSTALLOW_OWNER = ...
HOSTDENY_OWNER = ...
```

Machines can be listed by:

- Individual hostnames - example: condor.cs.wisc.edu
- Individual IP address - example: 128.105.67.29
- IP subnets (use a trailing "*") - examples: 144.105.*, 128.105.67.*
- Hostnames with a wildcard "*" character (only one "*" is allowed per name) - examples: *.cs.wisc.edu, sol*.cs.wisc.edu

Multiple machine entries can be separated by either a space or a comma.

For resolving something that falls into both allow and deny: Individual machines have a higher order of precedence than wildcard entries, and hostnames with a wildcard have a higher order of precedence than IP subnets. Otherwise, DENY has a higher order of precedence than ALLOW. (this is intuitively how most people would expect it to work).

In addition, you can specify any of the above access levels on a per-daemon basis, instead of machine-wide for all daemons. You do this with the subsystem string (described in section 3.4.1 on "Subsystem Names"), which is one of: "STARTD", "SCHEDD", "MASTER", "NEGOTIATOR", or "COLLECTOR". For example, if you wanted to grant different read access for the *condor_schedd*:

```
HOSTALLOW_READ_SCHEDD = <machines>
```


3.7.4 Access Levels each Daemons Uses

Here are all the commands registered in Condor, what daemon registers them, and what permission they are registered with. With this information, you should be able to grant exactly the permission you wish for your pool:

STARTD:

`WRITE` : All commands that relate to a schedd claiming the startd, starting jobs there, and stopping those jobs.

The command that *condor_checkpoint* sends to periodically checkpoint all running jobs.

`READ` : The command that *condor_preen* sends to find the current state of the startd.

`OWNER` : The command that *condor_vacate* sends to vacate any running jobs.

`NEGOTIATOR` : The command that the negotiator sends to match this startd with a given schedd.

NEGOTIATOR:

`WRITE` : The command that initiates a new negotiation cycle (sent by the schedd when new jobs are submitted, or someone issues a *condor_reschedule*).

`READ` : The command that can retrieve the current state of user priorities in the pool (what *condor_userprio* sends).

`ADMINISTRATOR` : The command that can set the current values of user priorities (what *condor_userprio -set* sends).

COLLECTOR:

`WRITE` : All commands that update the collector with new ClassAds.

`READ` : All commands that query the collector for ClassAds.

SCHEDD:

`NEGOTIATOR` : The command that the negotiator sends to begin negotiating with this schedd to match its jobs with available startds.

`WRITE` : The command which *condor_reschedule* sends to the schedd to get it to update the collector with a current ClassAd and begin a negotiation cycle.

The commands that a startd sends to the schedd when it must vacate its jobs and release the schedd's claim.

`OWNER` : The command that *condor_reconfig_schedd* sends to get the schedd to re-read it's config files.

`READ` : The "queue management" command, which all tools which view the status of the job queue send (such as *condor_q*, *condor_submit*, etc). Queue commands that want to write information to the job queue (such as *condor_submit* or *condor_rm*) do their own user-level authentication, so you can't update anything in a given ClassAd without being the owner of that ClassAd. Therefore, this command only requires `READ` access, since any writes are authenticated separately.

MASTER: All commands are registered with `ADMINISTRATOR` access:

reconfig : Master and all its children reconfigure themselves

restart : Master restarts itself (and all its children)

off : Master shuts down all its children

on : Master spawns all the daemons it's configured to spawn

master_off : Master shuts down all its children and exits

3.7.5 Access Level Examples

Notice in all these examples that `ADMINISTRATOR` access is only granted through a HOSTALLOW setting to explicitly grant access to a small number of machines. We recommend this.

- Let anyone join your pool. Only your central manager has administrative access (this is the default that ships with Condor)

```
HOSTALLOW_ADMINISTRATOR = $(CONDOR_HOST)
HOSTALLOW_OWNER = $(FULL_HOSTNAME), $(HOSTALLOW_ADMINISTRATOR)
```

- Only allow machines at NCSA to join or view the pool, Central Manager is the only machine with `ADMINISTRATOR` access.

```
HOSTALLOW_READ = *.ncsa.uiuc.edu
HOSTALLOW_WRITE = *.ncsa.uiuc.edu
HOSTALLOW_ADMINISTRATOR = $(CONDOR_HOST)
HOSTALLOW_OWNER = $(FULL_HOSTNAME), $(HOSTALLOW_ADMINISTRATOR)
```

- Only allow machines at NCSA and U of I Math department join the pool, EXCEPT do **not** allow lab machines to do so. Also do not allow the 177.55 subnet (perhaps this is the dial-in subnet). Allow anyone to view pool statistics. Only let "bigcheese" administer the pool (not the central manager).


```
HOSTALLOW_WRITE = *.ncsa.uiuc.edu, *.math.uiuc.edu
HOSTDENY_WRITE = lab-*.edu, *.lab.uiuc.edu, 177.55.*
HOSTALLOW_ADMINISTRATOR = bigcheese.ncsa.uiuc.edu
HOSTALLOW_OWNER = $(FULL_HOSTNAME), $(HOSTALLOW_ADMINISTRATOR)
```

- Only allow machines at NCSA and UW-Madison's CS department to view the pool. Only NCSA machines and "raven.cs.wisc.edu" can join the pool: (Note: raven has the read access it needs through the wildcard setting in `HOSTALLOW_READ`). This example also shows how you could use "\" to continue a long list of machines onto multiple lines, making it more readable (this works for all config file entries, not just host access entries, see section 3.4 on "Configuring Condor" for details).

```
HOSTALLOW_READ = *.ncsa.uiuc.edu, *.cs.wisc.edu
HOSTALLOW_WRITE = *.ncsa.uiuc.edu, raven.cs.wisc.edu
HOSTALLOW_ADMINISTRATOR = $(CONDOR_HOST), bigcheese.ncsa.uiuc.edu, \
                          biggercheese.uiuc.edu
HOSTALLOW_OWNER = $(FULL_HOSTNAME), $(HOSTALLOW_ADMINISTRATOR)
```

- Allow anyone except the military to view the status of your pool, but only let machines at NCSA view the job queues. Only NCSA machines can join the pool. The central manager, bigcheese, and biggercheese can perform most administrative functions. However, only "biggercheese" can update user priorities.

```
HOSTDENY_READ = *.mil
HOSTALLOW_READ_SCHEDD = *.ncsa.uiuc.edu
HOSTALLOW_WRITE = *.ncsa.uiuc.edu
HOSTALLOW_ADMINISTRATOR = $(CONDOR_HOST), bigcheese.ncsa.uiuc.edu, \
                          biggercheese.uiuc.edu
HOSTALLOW_ADMINISTRATOR_NEGOTIATOR = biggercheese.uiuc.edu
HOSTALLOW_OWNER = $(FULL_HOSTNAME), $(HOSTALLOW_ADMINISTRATOR)
```

3.8 Managing your Condor Pool

There are a number of administrative tools Condor provides to help you manage your pool. The following sections describe various tasks you might wish to perform on your pool and explains how to most efficiently do them.

All of the commands described in this section must be run from a machine listed in the `HOST_ALLOW_ADMINISTRATOR` setting in your config files, so that the IP/host-based security allows the administrator commands to be serviced. See section 3.7 on page 122 for full details about IP/host-based security in Condor.

3.8.1 Shutting Down and Restarting your Condor Pool

There are a couple of situations where you might want to shutdown and restart your entire Condor pool. In particular, when you want to install new binaries, it is generally best to make sure no jobs are running, shutdown Condor, and then install the new daemons.

Shutting Down your Condor Pool

The best way to shutdown your pool is to take advantage of the remote administration capabilities of the *condor_master*. The first step is to save the IP address and port of the *condor_master* daemon on all of your machines to a file, so that even if you shutdown your *condor_collector*, you can still send administrator commands to your different machines. You do this with the following command:

```
% condor_status -master -format "%s\n" MasterIpAddr > addresses
```

The first step to shutting down your pool is to shutdown any currently running jobs and give them a chance to checkpoint. Depending on the size of your pool, your network infrastructure, and the image-size of the standard jobs running in your pool, you may want to make this a slow process, only vacating one host at a time. You can either shutdown hosts that have jobs submitted (in which case all the jobs from that host will try to checkpoint simultaneously), or you can shutdown individual hosts that are running jobs. To shutdown a host, simply send:

```
% condor_off hostname
```

where "hostname" is the name of the host you want to shutdown. This will only work so long as your *condor_collector* is still running. Once you have shutdown Condor on your central manager, you will have to rely on the `addresses` file you just created.

If all the running jobs are checkpointed and stopped, or if you're not worried about the network load put in effect by shutting down everything at once, it is safe to turn off all daemons on all machines in your pool. You can do this with one command, so long as you run it from a blessed administrator machine:

```
% condor_off `cat addresses`
```

where `addresses` is the file where you saved your master addresses. *condor_off* will shutdown all the daemons, but leave the *condor_master* running, so that you can send a *condor_on* in the future.

Once all of the Condor daemons (except the *condor_master*) on each host is turned off, you're done. You are now safe to install new binaries, move your checkpoint server

to another host, or any other task that requires the pool to be shutdown to successfully complete.

NOTE: If you are planning to install a new *condor_master* binary, be sure to read the following section for special considerations with this somewhat delicate task.

Installing a New condor_master

If you are going to be installing a new *condor_master* binary, there are a few other steps you should take. If the *condor_master* restarts, it will have a new port it is listening on, so your `addresses` file will be stale information. Moreover, when the master restarts, it doesn't know that you sent it a *condor_off* in its past life, and will just start up all the daemons it's configured to spawn unless you explicitly tell it otherwise.

If you just want your pool to completely restart itself whenever the master notices its new binary, neither of these issues are of any concern and you can skip this (and the next) section. Just be sure installing the new master binary is the last thing you install, and once you put the new binary in place, the pool will restart itself over the next 5 minutes (whenever all the masters notice the new binary, which they each check for once every 5 minutes by default).

However, if you want to have absolute control over when the rest of the daemons restart, you must take a few steps.

1. Put the following setting in your global config file:

   ```
   START_DAEMONS = False
   ```

   This will make sure that when the master restarts itself that it doesn't also start up the rest of its daemons.

2. Install your new *condor_master* binary.

3. Start up Condor on your central manager machine. You will have to do this manually by logging into the machine and sending commands locally. First, send a *condor_restart* to make sure you've got the new master, then send a *condor_on* to start up the other daemons (including, most importantly, the *condor_collector*).

4. Wait 5 minutes, such that all the masters have a chance to notice the new binary, restart themselves, and send an update with their new address. Make sure that:

   ```
   % condor_status -master
   ```

   lists all the machines in your pool.

5. Remove the special setting from your global config file.

6. Recreate your `addresses` file as described above:

```
% condor_status -master -format "%s\n" MasterIpAddr > addresses
```

Once the new master is in place, and you're ready to start up your pool again, you can restart your whole pool by simply following the steps in the next section.

Restarting your Condor Pool

Once you are done performing whatever tasks you need to perform and you're ready to restart your pool, you simply have to send a *condor_on* to all the *condor_master* daemons on each host. You can do this with one command, so long as you run it from a blessed administrator machine:

```
% condor_on `cat addresses`
```

That's it. All your daemons should now be restarted, and your pool will be back on its way.

3.8.2 Reconfiguring Your Condor Pool

If you change a global config file setting and want to have all your machines start to use the new setting, you must send a *condor_reconfig* command to each host. The easiest way to do this is:

```
% condor_reconfig `condor_status -master`
```

3.9 Setting up Condor for Special Environments

The following sections describe how to setup Condor for use in a number of special environments or configurations. See section 3.3 on page 65 for installation instructions for the various "contrib modules" that you can optionally download and install in your pool.

3.9.1 Using Condor with AFS

If you are using AFS at your site, be sure to read section 3.4.5 on "Shared Filesystem Config Files Entries" for details on configuring your machines to interact with and use shared filesystems, AFS in particular.

Condor does not currently have a way to authenticate itself to AFS. This is true of the Condor daemons that would like to authenticate as AFS user Condor, and the *condor_shadow*, which would like to authenticate as the user who submitted the job it is serving. Since neither of these things can happen yet, there are a number of special things people

who use AFS with Condor must do. Some of this must be done by the administrator(s) installing Condor. Some of this must be done by Condor users who submit jobs.

AFS and Condor for Administrators

The most important thing is that since the Condor daemons can't authenticate to AFS, the `LOCAL_DIR` (and it's subdirectories like "log" and "spool") for each machine must be either writable to unauthenticated users, or must not be on AFS. The first option is a **VERY** bad security hole so you should **NOT** have your local directory on AFS. If you've got NFS installed as well and want to have your `LOCAL_DIR` for each machine on a shared file system, use NFS. Otherwise, you should put the `LOCAL_DIR` on a local partition on each machine in your pool. This means that you should run *condor_install* to install your release directory and configure your pool, setting the `LOCAL_DIR` parameter to some local partition. When that's complete, log into each machine in your pool and run *condor_init* to set up the local Condor directory.

The `RELEASE_DIR`, which holds all the Condor binaries, libraries and scripts can and probably should be on AFS. None of the Condor daemons need to write to these files, they just need to read them. So, you just have to make your `RELEASE_DIR` world readable and Condor will work just fine. This makes it easier to upgrade your binaries at a later date, means that your users can find the Condor tools in a consistent location on all the machines in your pool, and that you can have the Condor config files in a centralized location. This is what we do at UW-Madison's CS department Condor pool and it works quite well.

Finally, you might want to setup some special AFS groups to help your users deal with Condor and AFS better (you'll want to read the section below anyway, since you're probably going to have to explain this stuff to your users). Basically, if you can, create an AFS group that contains all unauthenticated users but that is restricted to a given host or subnet. You're supposed to be able to make these host-based ACLs with AFS, but we've had some trouble getting that working here at UW-Madison. What we have instead is a special group for all machines in our department. So, the users here just have to make their output directories on AFS writable to any process running on any of our machines, instead of any process on any machine with AFS on the Internet.

AFS and Condor for Users

The *condor_shadow* process runs on the machine where you submitted your Condor jobs and performs all file system access for your jobs. Because this process isn't authenticated to AFS as the user who submitted the job, it will not normally be able to write any output. So, when you submit jobs, any directories where your job will be creating output files will need to be world writable (to non-authenticated AFS users). In addition, if your program writes to `stdout` or `stderr`, or you're using a user log for your jobs, those files will need to be in a directory that's world-writable.

Any input for your job, either the file you specify as input in your submit file, or any files your program opens explicitly, needs to be world-readable.

Some sites may have special AFS groups set up that can make this unauthenticated access to your files less scary. For example, there's supposed to be a way with AFS to grant access to any unauthenticated process on a given host. That way, you only have to grant write access to unauthenticated processes on your submit machine, instead of any unauthenticated process on the Internet. Similarly, unauthenticated read access could be granted only to processes running your submit machine. Ask your AFS administrators about the existence of such AFS groups and details of how to use them.

The other solution to this problem is to just not use AFS at all. If you have disk space on your submit machine in a partition that is not on AFS, you can submit your jobs from there. While the *condor_shadow* is not authenticated to AFS, it does run with the effective UID of the user who submitted the jobs. So, on a local (or NFS) file system, the *condor_shadow* will be able to access your files normally, and you won't have to grant any special permissions to anyone other than yourself. If the Condor daemons are not started as root however, the shadow will not be able to run with your effective UID, and you'll have a similar problem as you would with files on AFS. See the section on "Running Condor as Non-Root" for details.

3.9.2 Configuring Condor for Multiple Platforms

Beginning with Condor version 6.0.1, you can use a single, global config file for all platforms in your Condor pool, with only platform-specific settings placed in separate files. This greatly simplifies administration of a heterogeneous pool by allowing you to change platform-independent, global settings in one place, instead of separately for each platform. This is made possible by the `LOCAL_CONFIG_FILE` parameter being treated by Condor as a list of files, instead of a single file. Of course, this will only help you if you are using a shared filesystem for the machines in your pool, so that multiple machines can actually share a single set of configuration files.

If you have multiple platforms, you should put all platform-independent settings (the vast majority) into your regular condor_config file, which would be shared by all platforms. This global file would be the one that is found with the `CONDOR_CONFIG` environment variable, user condor's home directory, or `/etc/condor/condor_config`.

You would then set the `LOCAL_CONFIG_FILE` parameter from that global config file to specify both a platform-specific config file and optionally, a local, machine-specific config file (this parameter is described in section 3.4.2 on "Condor-wide Config File Entries").

The order in which you specify files in the `LOCAL_CONFIG_FILE` parameter is important, because settings in files at the beginning of the list are overridden if the same settings occur in files later in the list. So, if you specify the platform-specific file and then the machine-specific file, settings in the machine-specific file would override those in the platform-specific file (which is probably what you want).

Specifying a Platform-Specific Config File

To specify the platform-specific file, you could simply use the `ARCH` and `OPSYS` parameters which are defined automatically by Condor. For example, if you had Intel Linux machines, Sparc Solaris 2.6 machines, and SGIs running IRIX 6.x, you might have files named:

```
condor_config.INTEL.LINUX
condor_config.SUN4x.SOLARIS26
condor_config.SGI.IRIX6
```

Then, assuming these three files were in the directory held in the `ETC` macro, and you were using machine-specific config files in the same directory, named by each machine's hostname, your `LOCAL_CONFIG_FILE` parameter would be set to:

```
LOCAL_CONFIG_FILE = $(ETC)/condor_config.$(ARCH).$(OPSYS), \
                    $(ETC)/$(HOSTNAME).local
```

Alternatively, if you are using AFS, you can use an "@sys link" to specify the platform-specific config file and let AFS resolve this link differently on different systems. For example, perhaps you have a soft linked named "condor_config.platform" that points to "condor_config.@sys". In this case, your files might be named:

```
condor_config.i386_linux2
condor_config.sun4x_56
condor_config.sgi_64
condor_config.platform -> condor_config.@sys
```

and your `LOCAL_CONFIG_FILE` parameter would be set to:

```
LOCAL_CONFIG_FILE = $(ETC)/condor_config.platform, \
                    $(ETC)/$(HOSTNAME).local
```

Platform-Specific Config File Settings

The only settings that are truly platform-specific are:

`RELEASE_DIR` Full path to where you have installed your Condor binaries. While the config files may be shared among different platforms, the binaries certainly cannot. Therefore, you must still maintain separate release directories for each platform in your pool. See section 3.4.2 on "Condor-wide Config File Entries" for details.

MAIL The full path to your mail program. See section 3.4.2 on "Condor-wide Config File Entries" for details.

CONSOLE_DEVICES Which devices in /dev should be treated as "console devices". See section 3.4.8 on "condor_startd Config File Entries" for details.

DAEMON_LIST Which daemons the *condor_master* should start up. The only reason this setting is platform-specific is because on Alphas running Digital Unix and SGIs running IRIX, you must use the *condor_kbdd*, which is not needed on other platforms. See section 3.4.7 on "condor_master Config File Entries" for details.

Reasonable defaults for all of these settings will be found in the default config files inside a given platform's binary distribution (except the RELEASE_DIR, since it is up to you where you want to install your Condor binaries and libraries). If you have multiple platforms, simply take one of the condor_config files you get from either running *condor_install* or from the <release_dir>/etc/examples/condor_config.generic file, take these settings out and save them into a platform-specific file, and install the resulting platform-independent file as your global config file. Then, find the same settings from the config files for any other platforms you are setting up and put them in their own platform specific files. Finally, set your LOCAL_CONFIG_FILE parameter to point to the appropriate platform-specific file, as described above.

Not even all of these settings are necessarily going to be different. For example, if you have installed a mail program that understands the "-s" option in /usr/local/bin/mail on all your platforms, you could just set MAIL to that in your global file and not define it anywhere else. If you've only got Digital Unix and IRIX machines, the DAEMON_LIST will be the same for each, so there's no reason not to put that in the global config file (or, if you have no IRIX or Digital Unix machines, DAEMON_LIST won't have to be platform-specific either).

Other Uses for Platform-Specific Config Files

It is certainly possible that you might want other settings to be platform-specific as well. Perhaps you want a different startd policy for one of your platforms. Maybe different people should get the email about problems with different platforms. There's nothing hard-coded about any of this. What you decide should be shared and what should not is entirely up to you and how you lay out your config files.

Since the LOCAL_CONFIG_FILE parameter can be an arbitrary list of files, you can even break up your global, platform-independent settings into separate files. In fact, your global config file might only contain a definition for LOCAL_CONFIG_FILE, and all other settings would be handled in separate files.

You might want to give different people permission to change different Condor settings. For example, if you wanted some user to be able to change certain settings, but nothing else, you could specify those settings in a file which was early in the LOCAL_CONFIG_FILE list, give that user write permission on that file, then include all the other files after that one. That way, if the user was trying to change settings she/he shouldn't, they would simply be overridden.

As you can see, this mechanism is quite flexible and powerful. If you have very specific configuration needs, they can probably be met by using file permissions, the LOCAL_CONFIG_FILE setting, and your imagination.

3.9.3 Full Installation of condor_compile

In order to take advantage of two major Condor features: checkpointing and remote system calls, users of the Condor system need to relink their binaries. Programs that are not relinked for Condor can run in Condor's "vanilla" universe just fine, however, they cannot checkpoint and migrate, or run on machines without a shared filesystem.

To relink your programs with Condor, we provide a special tool, *condor_compile*. As installed by default, *condor_compile* works with the following commands: *gcc*, *g++*, *g77*, *cc*, *acc*, *c89*, *CC*, *f77*, *fort77*, *ld*. On Solaris and Digital Unix, *f90* is also supported. See the *condor_compile*(1) man page for details on using *condor_compile*.

However, you can make *condor_compile* work transparently with all commands on your system whatsoever, including *make*.

The basic idea here is to replace the system linker (*ld*) with the Condor linker. Then, when a program is to be linked, the condor linker figures out whether this binary will be for Condor, or for a normal binary. If it is to be a normal compile, the old *ld* is called. If this binary is to be linked for condor, the script performs the necessary operations in order to prepare a binary that can be used with condor. In order to differentiate between normal builds and condor builds, the user simply places *condor_compile* before their build command, which sets the appropriate environment variable that lets the condor linker script know it needs to do its magic.

In order to perform this full installation of *condor_compile*, the following steps need to be taken:

1. Rename the system linker from ld to ld.real.
2. Copy the condor linker to the location of the previous ld.
3. Set the owner of the linker to root.
4. Set the permissions on the new linker to 755.

The actual commands that you must execute depend upon the system that you are on. The location of the system linker (*ld*), is as follows:

```
Operating System              Location of ld (ld-path)
```

```
Linux                          /usr/bin
Solaris 2.X                    /usr/ccs/bin
OSF/1 (Digital Unix)           /usr/lib/cmplrs/cc
```

On these platforms, issue the following commands (as root), where *ld-path* is replaced by the path to your system's *ld*.

```
mv /[ld-path]/ld /[ld-path]/ld.real
cp /usr/local/condor/lib/ld /[ld-path]/ld
chown root /[ld-path]/ld
chmod 755 /[ld-path]/ld
```

On IRIX, things are more complicated in that there are multiple *ld* binaries that need to be moved, and symbolic links need to be made in order to convince the linker to work, since it looks at the name of it's own binary in order to figure out what to do.

```
mv /usr/lib/ld /usr/lib/ld.real
mv /usr/lib/uld /usr/lib/uld.real
cp /usr/local/condor/lib/ld /usr/lib/ld
ln /usr/lib/ld /usr/lib/uld
chown root /usr/lib/ld /usr/lib/uld
chmod 755 /usr/lib/ld /usr/lib/uld
mkdir /usr/lib/condor
chown root /usr/lib/condor
chmod 755 /usr/lib/condor
ln -s /usr/lib/uld.real /usr/lib/condor/uld
ln -s /usr/lib/uld.real /usr/lib/condor/old_ld
```

If you remove Condor from your system latter on, linking will continue to work, since the condor linker will always default to compiling normal binaries and simply call the real ld. In the interest of simplicity, it is recommended that you reverse the above changes by moving your ld.real linker back to it's former position as ld, overwriting the condor linker. On IRIX, you need to do this for both linkers, and you will probably want to remove the symbolic links as well.

3.9.4 Installing the *condor_kbdd*

The condor keyboard daemon (*condor_kbdd*) monitors X events on machines where the operating system does not provide a way of monitoring the idle time of the keyboard or mouse. In particular, this is necessary on Digital Unix machines and IRIX machines.

<u>NOTE</u>: If you are running on Solaris, Linux, or HP/UX, you do not need to use the keyboard daemon.

Although great measures have been taken to make this daemon as robust as possible, the X window system was not designed to facilitate such a need, and thus is less then optimal on machines where many users log in and out on the console frequently.

In order to work with X authority, the system by which X authorizes processes to connect to X servers, the condor keyboard daemon needs to run with super user privileges. Currently, the daemon assumes that X uses the `HOME` environment variable in order to locate a file named `.Xauthority`, which contains keys necessary to connect to an X server. The keyboard daemon attempts to set this environment variable to various users home directories in order to gain a connection to the X server and monitor events. This may fail to work on your system, if you are using a non-standard approach. If the keyboard daemon is not allowed to attach to the X server, the state of a machine may be incorrectly set to idle when a user is, in fact, using the machine.

In some environments, the keyboard daemon will not be able to connect to the X server because the user currently logged into the system keeps their authentication token for using the X server in a place that no local user on the current machine can get to. This may be the case if you are running AFS and have the user's X authority file in an AFS home directory. There may also be cases where you cannot run the daemon with super user privileges because of political reasons, but you would still like to be able to monitor X activity. In these cases, you will need to change your XDM configuration in order to start up the keyboard daemon with the permissions of the currently logging in user. Although your situation may differ, if you are running X11R6.3, you will probably want to edit the files in /usr/X11R6/lib/X11/xdm. The Xsession file should have the keyboard daemon startup at the end, and the Xreset file should have the keyboard daemon shutdown. As of patch level 4 of Condor version 6.0, the keyboard daemon has some additional command line options to facilitate this. The -l option can be used to write the daemons log file to a place where the user running the daemon has permission to write a file. We recommend something akin to $HOME/.kbdd.log since this is a place where every user can write and won't get in the way. The -pidfile and -k options allow for easy shutdown of the daemon by storing the process id in a file. You will need to add lines to your XDM config that look something like this:

```
condor_kbdd -l $HOME/.kbdd.log -pidfile $HOME/.kbdd.pid
```

This will start the keyboard daemon as the user who is currently logging in and write the log to a file in the directory $HOME/.kbdd.log/. Also, this will save the process id of the daemon to /.kbdd.pid, so that when the user logs out, XDM can simply do a:

```
condor_kbdd -k $HOME/.kbdd.pid
```

This will shutdown the process recorded in /.kbdd.pid and exit.

To see how well the keyboard daemon is working on your system, review the log for the daemon and look for successful connections to the X server. If you see none, you may have a situation where the keyboard daemon is unable to connect to your machines X server. If this happens, please send mail to condor-admin@cs.wisc.edu and let us know about your situation.

3.10 Security In Condor

This section describes various aspects of security within Condor.

3.10.1 Running Condor as Non-Root

While we strongly recommend starting up the Condor daemons as root, we understand that that's not always possible. The main problems this causes are if you've got one Condor installation shared by many users on a single machine, or if you're setting up your machines to execute Condor jobs. If you're just setting up a submit-only installation for a single user, there's no need for (or benefit from) running as root.

What follows are the details of what effect running without root access has on the various parts of Condor:

condor_startd If you're setting up a machine to run Condor jobs and don't start the *condor_startd* as root, you're basically relying on the goodwill of your Condor users to agree to the policy you configure the startd to enforce as far as starting, suspending, vacating and killing Condor jobs under certain conditions. If you run as root, however, you can enforce these policies regardless of malicious users. By running as root, the Condor daemons run with a different UID than the Condor job that gets started (since the user's job is started as either the UID of the user who submitted it, or as user "nobody", depending on the `UID_DOMAIN` settings). Therefore, the Condor job cannot do anything to the Condor daemons. If you don't start the daemons as root, all processes started by Condor, including the end user's job, run with the same UID (since you can't switch UIDs unless you're root). Therefore, a user's job could just kill the *condor_startd* and *condor_starter* as soon as it starts up and by doing so, avoid getting suspended or vacated when a user comes back to the machine. This is nice for the user, since they get unlimited access to the machine, but awful for the machine owner or administrator. If you trust the users submitting jobs to Condor, this might not be a concern. However, to ensure that the policy you choose is effectively enforced by Condor, the *condor_startd* should be started as root.

In addition, some system information cannot be obtained without root access on some platforms (such as load average on IRIX). As a result, when we're running without root access, the startd has to call other programs (for example, "uptime") to get this information. This is much less efficient than getting the information directly from the kernel (which is what we do if we're running as root). On Linux and Solaris, we can get this information directly without root access, so this is not a concern on those platforms.

If you can't have all of Condor running as root, at least consider whether you can install the Condorstartd as setuid root. That would solve both of these problems. If you can't do that, you could also install it as a setgid sys or kmem program (depending on whatever group has read access to `/dev/kmem` on your system) and that would at least solve the system information problem.

condor_schedd The biggest problem running the schedd without root access is that the *condor_shadow* processes which it spawns are stuck with the same UID the *condor_schedd* has. This means that users submitting their jobs have to go out of their way to grant write access to user or group condor (or whoever the schedd is running as) for any files or directories their jobs write or create. Similarly, read access must be granted to their input files.

You might consider installing *condor_submit* as a setgid condor program so that at least the `stdout`, `stderr` and `UserLog` files get created with the right permissions. If *condor_submit* is a setgid program, it will automatically set it's umask to 002, so that creates group-writable files. This way, the simple case of a job that just writes to `stdout` and `stderr` will work. If users have programs that open their own files, they'll have to know to set the right permissions on the directories they submit from.

condor_master The *condor_master* is what spawns the *condor_startd* and *condor_schedd*, so if want them both running as root, you should have the master run as root. This happens automatically if you start the master from your boot scripts.

condor_negotiator

condor_collector There is no need to have either of these daemons running as root.

condor_kbdd On platforms that need the *condor_kbdd* (Digital Unix and IRIX) the *condor_kbdd* has to run as root. If it is started as any other user, it will not work. You might consider installing this program as a setuid root binary if you can't run the *condor_master* as root. Without the *condor_kbdd*, the startd has no way to monitor mouse activity at all, and the only keyboard activity it will notice is activity on ttys (such as xterms, remote logins, etc).

3.10.2 UIDs in Condor

This section has not yet been written

3.10.3 Root Config Files

This section has not yet been written

CHAPTER

FOUR

Miscellaneous Concepts

4.1 An Introduction to Condor's ClassAd Mechanism

ClassAds are a flexible mechanism for representing the characteristics and constraints of machines and jobs in the Condor system. ClassAds are used extensively in the Condor system to represent jobs, resources, submitters and other Condor daemons. An understanding of this mechanism is required to harness the full flexibility of the Condor system.

A ClassAd is is a set of uniquely named expressions. Each named expression is called an *attribute*. Figure 4.1 shows an example of a ClassAd with ten attributes.

```
MyType       = "Machine"
TargetType   = "Job"
Machine      = "froth.cs.wisc.edu"
Arch         = "INTEL"
OpSys        = "SOLARIS251"
Disk         = 35882
Memory       = 128
KeyboardIdle = 173
LoadAvg      = 0.1000
Requirements = TARGET.Owner=="smith" || LoadAvg<=0.3 && KeyboardIdle>15*60
```

Figure 4.1: An example ClassAd

ClassAd expressions look very much like expressions in C, and are composed of literals and attribute references composed with operators. The difference between ClassAd expressions and C expressions arise from the fact that ClassAd expressions operate in a much more

dynamic environment. For example, an expression from a machine's ClassAd may refer to an attribute in a job's ClassAd, such as `TARGET.Owner` in the above example. The value and type of the attribute is not known until the expression is evaluated in an environment which pairs a specific job ClassAd with the machine ClassAd.

ClassAd expressions handle these uncertainties by defining all operators to be *total* operators, which means that they have well defined behavior regardless of supplied operands. This functionality is provided through two distinguished values, `UNDEFINED` and `ERROR`, and defining all operators so that they can operate on all possible values in the ClassAd system. For example, the multiplication operator which usually only operates on numbers, has a well defined behavior if supplied with values which are not meaningful to multiply. Thus, the expression `10 * "A string"` evaluates to the value `ERROR`. Most operators are *strict* with respect to `ERROR`, which means that they evaluate to `ERROR` if any of their operands are `ERROR`. Similarly, most operators are strict with respect to `UNDEFINED`.

4.1.1 Syntax

ClassAd expressions are formed by composing literals, attribute references and other sub-expressions with operators.

Literals

Literals in the ClassAd language may be of integer, real, string, undefined or error types. The syntax of these literals is as follows:

Integer A sequence of continuous digits (i.e., `[0-9]`). Additionally, the keywords `TRUE` and `FALSE` (case insensitive) are syntactic representations of the integers 1 and 0 respectively.

Real Two sequences of continuous digits separated by a period (i.e., `[0-9]+.[0-9]+`).

String A double quote character, followed by an list of characters terminated by a double quote character. A backslash character inside the string causes the following character to be considered as part of the string, irrespective of what that character is.

Undefined The keyword `UNDEFINED` (case insensitive) represents the `UNDEFINED` value.

Error The keyword `ERROR` (case insensitive) represents the `ERROR` value.

Attributes

Every expression in a ClassAd is named by an *attribute name*. Together, the (name,expression) pair is called an *attribute*. An attributes may be referred to in other expressions through its attribute name.

Attribute names are sequences of alphabetic characters, digits and underscores, and may not begin with a digit. All characters in the name are significant, but case is *not* significant. Thus, `Memory`, `memory` and `MeMoRy` all refer to the same attribute.

An *attribute reference* consists of the name of the attribute being referenced, and an optional *scope resolution prefix*. The three prefixes that may be used are `MY.`, `TARGET.` and `ENV.`. The semantics of supplying a prefix are discussed in Section 4.1.2.

Operators

The operators that may be used in ClassAd expressions are similar to those available in C. The available operators and their relative precedence is shown in figure 4.2. The operator

```
-                    (high precedence)
*   /
+   -
<   <=   >=   >
==  !=   =?=  =!=
&&
||                   (low precedence)
```

Figure 4.2: Relative precedence of ClassAd expression operators

with the highest precedence is the unary minus operator. The only operators which are unfamiliar are the `=?=` and `=!=` operators, which are discussed in Section 4.1.2.

4.1.2 Evaluation Semantics

The ClassAd mechanism's primary purpose is for matching entities who supply constraints on candidate matches. The mechanism is therefore defined to carry out expression evaluations in the context of two ClassAds which are testing each other for a potential match. For example, the *condor_negotiator* evaluates the `Requirements` expressions of machine and job ClassAds to test if they can be matched. The semantics of evaluating such constraints is defined below.

Literals

Literals are self-evaluating, Thus, integer, string, real, undefined and error values evaluate to themselves.

Attribute References

Since the expression evaluation is being carried out in the context of two ClassAds, there is a potential for namespace ambiguities. The following rules define the semantics of attribute references made by ad A that is being evaluated in a context with another ad B:

1. If the reference is prefixed by a scope resolution prefix,
   - If the prefix is `MY.`, the attribute is looked up in ClassAd A. If the named attribute does not exist in A, the value of the reference is `UNDEFINED`. Otherwise, the value of the reference is the value of the expression bound to the attribute name.
   - Similarly, if the prefix is `TARGET.`, the attribute is looked up in ClassAd B. If the named attribute does not exist in B, the value of the reference is `UNDEFINED`. Otherwise, the value of the reference is the value of the expression bound to the attribute name.
   - Finally, if the prefix is `ENV.`, the attribute is evaluated in the "environment." Currently, the only attribute of the environment is `CurrentTime`, which evaluates to the integer value returned by the system call `time(2)`.
2. If the reference is not prefixed by a scope resolution prefix,
   - If the attribute is defined in A, the value of the reference is the value of the expression bound to the attribute name in A.
   - Otherwise, if the attribute is defined in B, the value of the reference is the value of the expression bound to the attribute name in B.
   - Otherwise, if the attribute is defined in the environment, the value of the reference is the evaluated value in the environment.
   - Otherwise, the value of the reference is `UNDEFINED`.
3. Finally, if the reference refers to an expression that is itself in the process of being evaluated, there is a circular dependency in the evaluation. The value of the reference is `ERROR`.

Operators

All operators in the ClassAd language are *total*, and thus have well defined behavior regardless of the supplied operands. Furthermore, most operators are *strict* with respect to `ERROR` and `UNDEFINED`, and thus evaluate to `ERROR` (or `UNDEFINED`) if either of their operands have these exceptional values.

- **Arithmetic operators:**
  1. The operators `*`, `/`, `+` and `-` operate arithmetically only on integers and reals.

2. Arithmetic is carried out in the same type as both operands, and type promotions from integers to reals are performed if one operand is an integer and the other real.
3. The operators are strict with respect to both `UNDEFINED` and `ERROR`.
4. If either operand is not a numerical type, the value of the operation is `ERROR`.

- **Comparison operators:**
  1. The comparison operators `==`, `!=`, `<=`, `<`, `>=` and `>` operate on integers, reals and strings.
  2. Comparisons are carried out in the same type as both operands, and type promotions from integers to reals are performed if one operand is a real, and the other an integer. Strings may not be converted to any other type, so comparing a string and an integer results in `ERROR`.
  3. The operators `==`, `!=`, `<=`, `<` and `>=` `>` are strict with respect to both `UNDEFINED` and `ERROR`.
  4. In addition, the operators `=?=` and `=!=` behave similar to `==` and `!=`, but are not strict. Semantically, the `=?=` tests if its operands are "identical," i.e., have the same type and the same value. For example, `10 == UNDEFINED` and `UNDEFINED == UNDEFINED` both evaluate to `UNDEFINED`, but `10 =?= UNDEFINED` and `UNDEFINED =?= UNDEFINED` evaluate to `FALSE` and `TRUE` respectively. The `=!=` operator test for the "is not identical to" condition.

- **Logical operators:**
  1. The logical operators `&&` and `||` operate on integers and reals. The zero value of these types are considered `FALSE` and non-zero values `TRUE`.
  2. The operators are *not* strict, and exploit the "don't care" properties of the operators to squash `UNDEFINED` and `ERROR` values when possible. For example, `UNDEFINED && FALSE` evaluates to `FALSE`, but `UNDEFINED || FALSE` evaluates to `UNDEFINED`.
  3. Any string operand is equivalent to an `ERROR` operand.

4.1.3 ClassAds in the Condor System

The simplicity and flexibility of ClassAds is heavily exploited in the Condor system. ClassAds are not only used to represent machines and jobs in the Condor pool, but also other entities that exist in the pool such as checkpoint servers, submitters of jobs and master daemons. Since arbitrary expressions may be supplied and evaluated over these ads, users have a uniform and powerful mechanism to specify constraints over these ads. These constraints may take the form of `Requirements` expressions in resource and job ads, or queries over other ads.

Requirements and Ranks

This is the mechanism by which users specify the constraints over machines and jobs respectively. Requirements for machines are specified through configuration files, while requirements for jobs are specified through the submit command file.

In both cases, the `Requirements` expression specifies the correctness criterion that the match must meet, and the `Rank` expression specifies the desirability of the match (where higher numbers mean better matches). For example, a job ad may contain the following expressions:

```
Requirements = Arch=="SUN4u" && OpSys == "SOLARIS251"
Rank         = TARGET.Memory + TARGET.Mips
```

In this case, the customer requires an UltraSparc computer running the Solaris 2.5.1 operating system. Among all such computers, the customer prefers those with large physical memories and high MIPS ratings. Since the `Rank` is a user specified metric, *any* expression may be used to specify the perceived desirability of the match. The *condor_negotiator* runs algorithms to deliver the "best" resource (as defined by the `Rank` expression) while satisfying other criteria.

Similarly, owners of resources may place constraints and preferences on their machines. For example,

```
    Friend        = Owner == "tannenba" || Owner == "wright"
    ResearchGroup = Owner == "jbasney" || Owner == "raman"
    Trusted       = Owner != "rival" && Owner != "riffraff"
    Requirements  = Trusted && ( ResearchGroup || LoadAvg < 0.3 &&
                         KeyboardIdle > 15*60 )
    Rank          = Friend + ResearchGroup*10
```

The above policy states that the computer will never run jobs owned by users "rival" and "riffraff," while the computer will always run a job submitted by members of the research group. Furthermore, jobs submitted by friends are preferred to other foreign jobs, and jobs submitted by the research group are preferred to jobs submitted by friends.

Note: Because of the dynamic nature of ClassAd expressions, there is no *a priori* notion of an integer valued expression, a real valued expression, etc. However, it is intuitive to think of the `Requirements` and `Rank` expressions as integer valued and real valued expressions respectively. If the actual type of the expression is not of the expected type, the value is assumed to be zero.

Querying with ClassAd Expressions

The flexibility of this system may also be used when querying ClassAds through the *condor_status* and *condor_q* tools which allow users to supply ClassAd constraint expressions from the command line.

For example, to find all computers which have had their keyboards idle for more than 20 minutes and have more than 100 MB of memory:

```
% condor_status -const 'KeyboardIdle > 20*60 && Memory > 100'

Name       Arch     OpSys        State      Activity   LoadAv Mem  ActvtyTime

amul.cs.wi SUN4u    SOLARIS251   Claimed    Busy       1.000  128   0+03:45:01
aura.cs.wi SUN4u    SOLARIS251   Claimed    Busy       1.000  128   0+00:15:01
balder.cs. INTEL    SOLARIS251   Claimed    Busy       1.000  1024  0+01:05:00
beatrice.c INTEL    SOLARIS251   Claimed    Busy       1.000  128   0+01:30:02
...
...
                     Machines Owner Claimed Unclaimed Matched Preempting

    SUN4u/SOLARIS251        3     0       3         0       0          0
    INTEL/SOLARIS251       21     0      21         0       0          0
    SUN4x/SOLARIS251        3     0       3         0       0          0
           SGI/IRIX6        1     0       0         1       0          0
         INTEL/LINUX        1     0       1         0       0          0

               Total       29     0      28         1       0          0
```

The similar flexibility exists in querying job queues in the Condor system.

CHAPTER

FIVE

Command Reference Manual (man pages)

condor_checkpoint

checkpoint jobs running on the specified hosts

Synopsis

condor_checkpoint [**-help**] [**-version**] [**hostname ...**]

Description

condor_checkpoint causes the startd's on the specified hosts to perform a checkpoint on any running jobs. The jobs continue to run once they are done checkpointing. If no host is specified, only the current host is sent the checkpoint command.

A periodic checkpoint means that the job will checkpoint itself, but then it will immediately continue running after the checkpoint has completed. *condor_vacate*, on the other hand, will result in the job exiting (vacating) after it checkpoints.

If the job being checkpointed is running in the Standard Universe, the job is checkpointed and then just continues running on the same machine. If the job is running in the Vanilla Universe, or there is currently no Condor job running on that host, then *condor_checkpoint* has no effect.

Normally there is no need for the user or administrator to explicitly run *condor_checkpoint*. Checkpointing a running condor job is normally handled automatically by Condor by following the policies stated in Condor's configuration files.

Options

Supported options are as follows:

-help Display usage information

-version Display version information

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_compile

create a relinked executable for submission to the Standard Universe

Synopsis

condor_compile *cc* | *CC* | *gcc* | *f77* | *g++* | *ld* | *make* | ...

Description

Use *condor_compile* to relink a program with the Condor libraries for submission into Condor's Standard Universe. The Condor libraries provide the program with additional support, such as the capability to checkpoint, which is required in Condor's Standard Universe mode of operation. *condor_compile* requires access to the source or object code of the program to be submitted; if source or object code for the program is not available (i.e. only an executable binary, or if it is a shell script), then the program must submitted into Condor's Vanilla Universe. See the reference page for *condor_submit* and/or consult the "Condor Users and Administrators Manual" for further information.

To use *condor_compile*, simply enter "condor_compile" followed by whatever you would normally enter to compile or link your application. Any resulting executables will have the Condor libraries linked in. For example:

```
condor_compile cc -O -o myprogram.condor file1.c file2.c ...
```

will produce a binary "myprogram.condor" which is relinked for Condor, capable of checkpoint/migration/remote-system-calls, and ready to submit to the Standard Universe.

If the Condor administrator has opted to fully install *condor_compile*, then *condor_compile* can be followed by practically any command or program, including make or shell-script programs. For example, the following would all work:

```
condor_compile make

condor_compile make install

condor_compile f77 -O mysolver.f

condor_compile /bin/csh compile-me-shellscript
```

If the Condor administrator has opted to only do a partial install of *condor_compile*, the you are restricted to following *condor_compile* with one of these programs:


```
cc (the system C compiler)

acc (ANSI C compiler, on Sun systems)

c89 (POSIX compliant C compiler, on some systems)

CC (the system C++ compiler)

f77 (the system FORTRAN compiler)

gcc (the GNU C compiler)

g++ (the GNU C++ compiler)

g77 (the GNU FORTRAN compiler)

ld (the system linker)
```

Note that if you use explicitly call "ld" when you normally create your binary, simply use:

```
condor_compile ld <ld arguments and options>
```

instead.

Author

Condor Team, University of Wisconsin–Madison

Copyright

See the *Condor Version 6.0.3 Manual* for additional notices.

condor_config_val

Query a given condor configuration variable

Synopsis

condor_config_val [**-name** *daemon_name*] [**-pool** *hostname*] [**-address** *¡ip:port¿*] [**-master** | **-schedd** | **-startd** | **-collector** | **-negotiator**] *variable* [**variable** ...]

Description

condor_config_val can be used to quickly see what the current condor configuration is on any given machine. Given a list of variables, *condor_config_val* will report what each of these variables is currently set to. If a given variable is not defined, *condor_config_val* will halt on that variable, and report that it is not defined. By default, *condor_config_val* looks in the local machine's configuration files in order to evaluate the variables.

Options

Supported options are as follows:

-name *daemon_name* Query the specified daemon for its configuration

-pool *hostname* Use the given central manager to find daemons

-address *¡ip:port¿* connect to the given ip/port

-master | **-schedd** | **-startd** | **-collector** | **-negotiator** The daemon to query (if not specified, master is default)

variable ... The variables to query

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_history

View log of condor jobs completed to date

Synopsis

condor_history [**-help**] [**-l**] [**-f** *filename*] [*cluster* | *cluster.process* | *owner*]

Description

condor_history displays a summary of all condor jobs listed in the specified history files. If no history files are specified (with the **-f** option), the local history file as specified in Condor's configuration file (condor/spool/history by default) is read. The default listing summarizes each job on a single line, and contains the following items:

ID The cluster/process id of the condor job.

OWNER The owner of the job.

SUBMITTED The month, day, hour, and minute the job was submitted to the queue.

CPU_USAGE Remote CPU time accumulated by the job to date in days, hours, minutes, and seconds.

ST Completion status of the job (C = completed and X = removed).

COMPLETED The time the job was completed.

PRI User specified priority of the job, ranges from -20 to +20, with higher numbers corresponding to greater priority.

SIZE The virtual image size of the executable in megabytes.

CMD The name of the executable.

If a job ID (in the form of cluster_id or cluster_id.proc_id) or an owner is provided, output will be restricted only to jobs with the specified IDs and/or submitted by the specified owner.

Options

Supported options are as follows:

-help Get a brief description of the supported options

-f ***filename*** Use the specified file instead of the default history file

-l Display job ads in long format

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_master

The master Condor Daemon

Synopsis

condor_master

Description

condor_master This daemon is responsible for keeping all the rest of the Condor daemons running on each machine in your pool. It spawns the other daemons, and periodically checks to see if there are new binaries installed for any of them. If there are, the master will restart the affected daemons. In addition, if any daemon crashes, the master will send email to the Condor Administrator of your pool and restart the daemon. The *condor_master* also supports various administrative commands that let you start, stop or reconfigure daemons remotely. The *condor_master* will run on every machine in your Condor pool, regardless of what functions each machine are performing.

See section 3.1.2 in Admin Manual for more information about *condor_master* and other Condor daemons.

Author

Condor Team, University of Wisconsin–Madison

Copyright

See the *Condor Version 6.0.3 Manual* for additional notices.

condor_master_off

Shutdown Condor and the *condor_master*

Synopsis

condor_master_off [**-help**] [**-version**] [**hostname ...**]

Description

condor_master_off shuts down all of the condor daemons running on a given machine. It does this cleanly without a loss of work done by any jobs currently running on this machine, or jobs that are running on other machines that have been submitted from this machine. At the end of the shutdown process, unlike *condor_off*, *condor_master_off* also shuts down the *condor_master* daemon. If you want to turn condor back on on this machine in the future, you will need to restart the *condor_master*.

Options

Supported options are as follows:

-help Display usage information

-version Display version information

hostname ... Turn shutdown condor on this list of machines

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_off

Shutdown condor daemons

Synopsis

condor_off [**-help**] [**-version**] [**hostname ...**]

Description

condor_off shuts down all of the condor daemons running on a given machine. It does this cleanly without a loss of work done by any jobs currently running on this machine, or jobs that are running on other machines that have been submitted from this machine. The only daemon that remains running is the *condor_master*, which can handle both local and remote requests to restart the other condor daeomns if need be. To restart condor running on a machine, see the *condor_on* command.

Options

Supported options are as follows:

-help Display usage information

-version Display version information

hostname ... Turn condor off on this list of machines

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_on

Startup condor daemons

Synopsis

condor_on [**-help**] [**-version**] [**hostname ...**]

Description

condor_on starts up all of the condor daemons running on a given machine. This command assumes that the *condor_master* is already running on the machine. If this is not the case, *condor_on* will fail complaining that it can't find the address of the master. *condor_on* will tell the *condor_master* to start up the condor daemons specified in the configuration variable DAEMON_LIST.

Options

Supported options are as follows:

-help Display usage information

-version Display version information

hostname ... Turn condor on on this list of machines

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_preen

remove extraneous files from Condor directories

Synopsis

condor_preen [**-mail**] [**-remove**] [**-verbose**]

Description

condor_preen examines the directories belonging to Condor, and removes extraneous files and directories which may be left over from Condor processes which terminated abnormally either due to internal errors or a system crash. The directories checked are the LOG, EXECUTE, and SPOOL directories as defined in the Condor configuration files. *condor_preen* is intended to be run as user root (or user condor) periodically as a backup method to ensure reasonable file system cleanliness in the face of errors. This is done automatically by default by the *condor_master*. It may also be explicitly invoked on an as needed basis.

Options

Supported options are as follows:

-mail Send mail to the CONDOR_ADMINISTRATORS as defined in the Condor configuration files instead of writing to the standard output

-remove Remove the offending files and directories rather than just reporting on them

-verbose List all files found in the Condor directories, even those which are not considered extraneous

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_prio

change priority of jobs in the condor queue

Synopsis

condor_prio [**-p** *priority*] [+ | - *value*] [**-n** *schedd_name*] *cluster* | *cluster.process* | *username* | *-a*

Description

condor_prio changes the priority of one or more jobs in the condor queue. If a cluster_id and a process_id are both specified, *condor_prio* attempts to change the priority of the specified process. If a cluster_id is specified without a process_id, *condor_prio* attempts to change priority for all processes belonging to the specified cluster. If a username is specified, *condor_prio* attempts to change priority of all jobs belonging to that user. If the -a flag is set, *condor_prio* attempts to change priority of all jobs in the condor queue. The user must specify a priority adjustment or new priority. If the -p option is specified, the priority of the job(s) are set to the next argument. The user can also adjust the priority by supplying a + or - immediately followed by a digit. The priority of a job ranges from -20 to +20, with higher numbers corresponding to greater priority. Only the owner of a job or the super user can change the priority for it.

The priority changed by *condor_prio* is only compared to the priority of other jobs owned by the same user and submitted from the same machine. See the "Condor Users and Administrators Manual" for further details on Condor's priority scheme.

Options

Supported options are as follows:

-p *priority* Set priority to the specified value

+ | **- *value*** Change priority by the specified value

-n *schedd_name* Change priority of jobs queued at the specified schedd

-a Change priority of all the jobs in the queue

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_q

Display information about jobs in queue

Synopsis

condor_q [**-help**] [**-global**] [**-submitter** *submitter*] [**-name** *name*] [**-pool** *hostname*] [**-analyze**] [**-long**] [**-analyze**] [{*cluster* | *cluster.process* | *owner* | ***-constraint*** *expression* ...}]

Description

condor_q displays information about jobs in the Condor job queue. By default, *condor_q* queries the local job queue but this behavior may be modified by specifying:

- the **-global** option, which queries all job queues in the pool
- a schedd name with the **-name** option, which causes the queue of the named schedd to be queried
- a submitter with the **-submitter** option, which causes all queues of the named submitter to be queried

To restrict the display to jobs of interest, a list of zero or more restrictions may be supplied. Each restriction may be one of:

- a *cluster* and a *process* matches jobs which belong to the specified cluster and have the specified process number
- a *cluster* without a *process* matches all jobs belonging to the specified cluster
- a *owner* matches all jobs owned by the specified owner
- a **-constraint** *expression* which matches all jobs that satisfy the specified ClassAd expression. (See section 4.1 for a discussion of ClassAd expressions.)

If no *owner* restrictions are present in the list, the job matches the restriction list if it matches at least one restriction in the list. If *owner* restrictions are present, the job matches the list if it matches one of the *owner* restrictions *and* at least one non-owner restriction.

If the **-long** option is specified, *condor_q* displays a long description of the jobs in the queue. Otherwise, a one line summary of information is displayed as follows:

ID The cluster/process id of the condor job.

OWNER The owner of the job.

SUBMITTED The month, day, hour, and minute the job was submitted to the queue.

CPU_USAGE Remote CPU time accumulated by the job to date in days, hours, minutes, and seconds. (If the job is currently running, time accumulated during the current run is *not* shown.)

ST Current status of the job. U = unexpanded (never been run), R = running, I = idle (waiting for a machine to execute on), C = completed, and X = removed.

PRI User specified priority of the job, ranges from -20 to +20, with higher numbers corresponding to greater priority.

SIZE The virtual image size of the executable in megabytes.

CMD The name of the executable.

The *-analyze* option may be used to determine why certain jobs are not running by performing an analysis on a per machine basis for each machine in the pool. The reasons may vary among failed constraints, insufficient priority, resource owner preferences and prevention of preemption by the `PREEMPTION_HOLD` expression. If the *-long* option is specified along with the *-analyze* option, the reason for failure is displayed on a per machine basis.

Options

Supported options are as follows:

-help Get a brief description of the supported options

-global Get queues of all the submitters in the system

-submitter *submitter* List jobs of specific submitter from all the queues in the pool

-pool *hostname* Use hostname as the central manager to locate schedds. (The default is the `COLLECTOR_HOST` specified in the configuration file.

-analyze Perform an approximate analysis to determine how many resources are available to run the requested jobs

-name *name* Show only the job queue of the named schedd

-long Display job ads in long format

Restriction list The restriction list may have zero or more items, each of which may be:

cluster match all jobs belonging to cluster

cluster.proc match all jobs belonging to cluster with a process number of *proc*

-constraint *expression* match all jobs which match the ClassAd expression constraint

A job matches the restriction list if it matches any restriction in the list Additionally, if *owner* restrictions are supplied, the job matches the list only if it also matches an *owner* restriction.

General Remarks

Although *-analyze* provides a very good first approximation, the analyzer cannot diagnose all possible situations because the analysis is based on instantaneous and local information. Therefore, there are some situations (such as when several submitters are contending for resources, or if the pool is rapidly changing state) which cannot be accurately diagnosed.

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_reconfig

Reconfigure condor daemons

Synopsis

condor_reconfig [**-help**] [**-version**] [**hostname ...**]

Description

condor_reconfig reconfigures all of the condor daemons in accordance with the current status of the condor configuration file(s). Once reconfiguration is complete, the daemons will behave according to the policies stated in the configuration file(s). The only exception is with the DAEMON_LIST variable, which will only be updated if the *condor_restart* command is used. In general, *condor_reconfig* should be used when making changes to the configuration files, since it is faster and more efficient then restarting the daemons.

Options

Supported options are as follows:

-help Display usage information

-version Display version information

hostname ... Reconfigure condor on this list of machines

Author

Condor Team, University of Wisconsin–Madison

Copyright

See the *Condor Version 6.0.3 Manual* for additional notices.

condor_reconfig_schedd

Reconfigure condor schedd

Synopsis

condor_reconfig_schedd [**-help**] [**-version**] [**hostname ...**]

Description

condor_reconfig_schedd reconfigures the condor_schedd in accordance with the current status of the condor configuration file(s). Once reconfiguration is complete, the daemon will behave according to the policies stated in the configuration file(s). This command is similar to the *condor_reconfig* command except that it only updates the schedd. The schedd is the condor daemon responsible for managing user's jobs submitted from this machine.

Options

Supported options are as follows:

-help Display usage information

-version Display version information

hostname ... Reconfigure condor on this list of machines

Author

Condor Team, University of Wisconsin–Madison

Copyright

See the *Condor Version 6.0.3 Manual* for additional notices.

condor_reschedule

Update scheduling information to the central manager

Synopsis

condor_reschedule [**-help**] [**-version**] [**hostname ...**]

Description

condor_reschedule updates the information about a given machines resources and jobs to the central manager. This can be used if one wants to see the current status of a machine. In order to do this, one would first run *condor_reschedule*, and then use the *condor_status* command to get specific information about that machine. *condor_reschedule* also starts a new negotiation cycle between resource owners and resource providers on the central managers, so that jobs can be matched with machines right away. This can be useful in situations where the time between negotiation cycles is somewhat long, and an administrator wants to see if a job they have in the queue will get matched without waiting for the next negotiation cycle.

Options

Supported options are as follows:

-help Display usage information

-version Display version information

hostname ... Reconfigure condor on this list of machines

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_restart

Restart the *condor_master*

Synopsis

condor_restart [**-help**] [**-version**] [**hostname ...**]

Description

condor_restart restarts the *condor_master* on the local machine, or all the machines specified in the hostname list. If, for some reason, the *condor_master* needs to be restarted again with a fresh state, this is the command that should be used to do so. Also, if the DAEMON_LIST variable in the condor configuration file has been changed, one must restart the *condor_master* in order to see these changes. A simple *condor_reconfigure* is not enough in this situation. *condor_restart* will safely shut down all running jobs and all submitted jobs from the machine being restarted, shutdown all the child daemons of the *condor_master*, and restart the *condor_master*.

Options

Supported options are as follows:

-help Display usage information

-version Display version information

hostname ... A list of machines to restart the *condor_master* on.

Author

Condor Team, University of Wisconsin–Madison

Copyright

See the *Condor Version 6.0.3 Manual* for additional notices.

condor_rm

remove jobs from the condor queue

Synopsis

condor_rm [**-n** *schedd_name*] *cluster* | *cluster.process* | *username* | *-a*

Description

condor_rm removes one or more jobs from the condor job queue. If a cluster_id and a process_id are both specified, *condor_rm* attempts to remove the specified process from the queue. If a cluster_id is specified without a process_id, *condor_rm* attempts to remove all processes belonging to the specified cluster. If a username is specified, *condor_rm* attempts to remove all jobs belonging to that user. Only the owner of a job, user condor, or user root can remove any given job.

Options

Supported options are as follows:

-n ***schedd_name*** Remove jobs in the queue of the specified schedd

-a Remove all the jobs in the queue

Author

Condor Team, University of Wisconsin–Madison

Copyright

See the *Condor Version 6.0.3 Manual* for additional notices.

condor_status

Display status of the Condor pool

Synopsis

condor_status [*help options*] [*query options*] [*display options*] [*custom options*] [*hostname* ...]

Description

condor_status is a versatile tool that may be used to monitor and query the Condor pool. The *condor_status* tool can be used to query resource information, submitter information, checkpoint server information, and daemon master information. The specific query sent and the resulting information display is controlled by the query options supplied. Queries and display formats can also be customized.

The options that may be supplied to *condor_status* belong to five groups:

- **Help options** provide information about the *condor_status* tool.
- **Query options** control the content and presentation of status information.
- **Display options** control the display of the queried information.
- **Custom options** allow the user to customize query and display information.
- **Host options** specify specific machines to be queried

At any time, only one *help option*, one *query option* and one *custom option* may be specified. Any number of *custom* and *host options* may be specified.

Options

Supported options are as follows:

-help (Help option) Display usage information

-diagnose (Help option) Print out query ad without performing query

-avail (Query option) Query *condor_startd* ads and identify resources which are available

-claimed (Query option) Query *condor_startd* ads and print information about claimed resources

-ckptsrvr (Query option) Query *condor_ckpt_server* ads and display checkpoint server attributes

-master (Query option) Query *condor_master* ads and display daemon master attributes

-pool *hostname* Query the specified central manager. (*condor_status* queries `COLLECTOR_HOST` by default)

-schedd (Query option) Query *condor_schedd* ads and display attributes

-server (Query option) Query *condor_startd* ads and display resource attributes

-startd (Query option) Query *condor_startd* ads

-state (Query option) Query *condor_startd* ads and display resource state information

-submitters (Query option) Query ads sent by submitters and display important submitter attributes

-verbose (Display option) Display entire classads. Implies that totals will not be displayed.

-long (Display option) Display entire classads (same as **-verbose**)

-total (Display option) Display totals only

-constraint *const* (Custom option) Add constraint expression

-format *fmt attr* (Custom option) Register display format and attribute name. The *fmt* string has the same format as `printf(3)`, and *attr* is the name of the attribute that should be displayed in the specified format.

General Remarks

- The information obtained from *condor_startds* and *condor_schedds* may sometimes appear to be inconsistent. This is normal since startds and schedds update the Condor manager at different rates, and since there is a delay as information propagates through the network and the system.
- Note that the `ActivityTime` in the `Idle` state is *not* the amount of time that the machine has been idle. See the section on *condor_startd* states in the *Administrator's Manual* for more information.

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_submit

Queue jobs for execution on remote machines

Synopsis

condor_submit [**-v**] [**-n** *schedd_name*] [**-r** *schedd_name*] *submit-description file*

Description

condor_submit is the program for submitting jobs to Condor. *condor_submit* requires a submit-description file which contains commands to direct the queuing of jobs. One description file may contain specifications for the queuing of many condor jobs at once. All jobs queued by a single invocation of *condor_submit* must share the same executable, and are referred to as a "job cluster". It is advantageous to submit multiple jobs as a single cluster because:

- Only one copy of the checkpoint file is needed to represent all jobs in a cluster until they begin execution.
- There is much less overhead involved for Condor to start the next job in a cluster than for Condor to start a new cluster. This can make a big difference if you are submitting lots of short running jobs.

SUBMIT DESCRIPTION FILE COMMANDS

Each condor job description file describes one cluster of jobs to be placed in the condor execution pool. All jobs in a cluster must share the same executable, but they may have different input and output files, and different program arguments, etc. The submit-description file is then used as the only command-line argument to *condor_submit*.

The submit-description file must contain one *executable* command and at least one *queue* command. All of the other commands have default actions.

The commands which can appear in the submit-description file are:

executable = <name> The name of the executable file for this job cluster. Only one executable command may be present in a description file. If submitting into the Standard Universe, which is the default, then the named executable must have been re-linked with the Condor libraries (such as via the *condor_compile* command). If submitting into the Vanilla Universe, then the named executable need not be re-linked and can be any process which can run in the background (shell scripts work fine as well).

input = <pathname> Condor assumes that its jobs are long-running, and that the user will not wait at the terminal for their completion. Because of this, the standard files which normally access the terminal, (stdin, stdout, and stderr), must refer to files. Thus, the filename specified with **input** should contain any keyboard input the program requires (i.e. this file becomes stdin). If not specified, the default value of /dev/null is used.

output = <pathname> The **output** filename will capture any information the program would normally write to the screen (i.e. this file becomes stdout). If not specified, the default value of /dev/null is used.

error = <pathname> The **error** filename will capture any error messages the program would normally write to the screen (i.e. this file becomes stderr). If not specified, the default value of /dev/null is used.

arguments = <argument_list> List of arguments to be supplied to the program on the command line.

initaldir = <directory-path> Used to specify the current working directory for the Condor job. Should be a path to a preexisting directory. If not specified, *condor_submit* will automatically insert the user's current working directory at the time *condor_submit* was run as the value for **initialdir**.

requirements = <ClassAd Boolean Expression> The requirements command is a boolean ClassAd expression which uses C-like operators. In order for any job in this cluster to run on a given machine, this requirements expression must evaluate to true on the given machine. For example, to require that whatever machine executes your program has a least 64 Meg of RAM and has a MIPS performance rating greater than 45, use:

```
requirements = Memory >= 64 && Mips > 45
```

Only one requirements command may be present in a description file. By default, *condor_submit* appends the following clauses to the requirements expression:

1. Arch and OpSys are set equal to the Arch and OpSys of the submit machine. In other words: unless you request otherwise, Condor will give your job machines with the same architecture and operating system version as the machine running *condor_submit*.
2. Disk > ExecutableSize. To ensure there is enough disk space on the target machine for Condor to copy over your executable.
3. VirtualMemory >= ImageSize. To ensure the target machine has enough virtual memory to run your job.
4. If Universe is set to Vanilla, FileSystemDomain is set equal to the submit machine's FileSystemDomain.

You can view the requirements of a job which has already been submitted (along with everything else about the job ClassAd) with the command *condor_q -l*; see the command reference for *condor_q* on page 170. Also, see the Condor Users Manual for complete information on the syntax and available attributes that can be used in the ClassAd expression.

rank = <ClassAd Float Expression> A ClassAd Floating-Point expression that states how to rank machines which have already met the requirements expression. Essentially, rank expresses preference. A higher numeric value equals better rank. Condor will give the job the machine with the highest rank. For example,

```
requirements = Memory > 60
rank = Memory
```

asks Condor to find all available machines with more than 60 megabytes of memory and give the job the one with the most amount of memory. See the Condor Users Manual for complete information on the syntax and available attributes that can be used in the ClassAd expression.

priority = <priority> Condor job priorities range from -20 to +20, with 0 being the default. Jobs with higher numerical priority will run before jobs with lower numerical priority. Note that this priority is on a per user basis; setting the priority will determine the order in which your own jobs are executed, but will have no effect on whether or not your jobs will run ahead of another user's jobs.

notification = <when> Owners of condor jobs are notified by email when certain events occur. If *when* is set to "ALWAYS", the owner will be notified whenever the job is checkpointed, and when it completes. If *when* is set to "COMPLETE" (the default), the owner will be notified when the job terminates. If *when* is set to "ERROR", the owner will only be notified if the job terminates abnormally. Finally, if *when* is set to "NEVER", the owner will not be mailed, regardless what happens to the job.

notify_user = <email-address> Used to specify the email address to use when Condor sends email about a job. If not specified, Condor will default to using :

```
job-owner@UID_DOMAIN
```

where `UID_DOMAIN` is specified by the Condor site administrator. If `UID_DOMAIN` has not been specified, Condor will send the email to :

```
job-owner@submit-machine-name
```

getenv = <True | False> If **getenv** is set to *True*, then *condor_submit* will copy all of the user's current shell environment variables at the time of job submission into the job ClassAd. The job will therefore execute with the same set of environment variables that the user had at submit time. Defaults to *False*.

environment = <parameter_list> List of environment variables of the form :

```
<parameter> = <value>
```

Multiple environment variables can be specified by separating them with a semicolon (" ; "). These environment variables will be placed into the job's environment before execution. The length of all characters specified in the environment is currently limited to 4096 characters.

log = <pathname> Use **log** to specify a filename where Condor will write a log file of what is happening with this job cluster. For example, Condor will log into this file when and where the job begins running, when the job is checkpointed and/or migrated, when the job completes, etc. Most users find specifying a **log** file to be very handy; its use is recommended. If no **log** entry is specified, Condor does not create a log for this cluster.

universe = <vanilla | standard | pvm | scheduler> Specifies which Condor Universe to use when running this job. The Condor Universe specifies a Condor execution environment. The *standard* Universe is the default, and tells Condor that this job has been re-linked via *condor_compile* with the Condor libraries and therefore supports checkpointing and remote system calls. The *vanilla* Universe is an execution environment for jobs which have not been linked with the Condor libraries. *Note:* use the *vanilla* Universe to submit shell scripts to Condor. The *pvm* Universe is for a parallel job written with PVM 3.3, and *scheduler* is for a job that should act as a metascheduler. See the Condor User's Manual for more information about using Universe.

image_size = <size> This command tells Condor the maximum virtual image size to which you believe your program will grow during its execution. Condor will then execute your job only on machines which have enough resources, (such as virtual memory), to support executing your job. If you do not specify the image size of your job in the description file, Condor will automatically make a (reasonably accurate) estimate about its size and adjust this estimate as your program runs. If the image size of your job is underestimated, it may crash due to inability to acquire more address space, e.g. malloc() fails. If the image size is overestimated, Condor may have difficulty finding machines which have the required resources. *size* must be in kbytes, e.g. for an image size of 8 megabytes, use a *size* of 8000.

machine_count = <min..max> If **machine_count** is specified, Condor will not start the job until it can simultaneously supply the job with *min* machines. Condor will continue to try to provide up to *max* machines, but will not delay starting of the job to do so. If the job is started with fewer than *max* machines, the job will be notified via a usual PvmHostAdd notification as additional hosts come on line. **Important:** only use **machine_count** if an only if submitting into the PVM Universe. At this time, **machine_count** must be used only with a parallel PVM application.

coresize = <size> Should the user's program abort and produce a core file, **coresize** specifies the maximum size in bytes of the core file which the user wishes to keep. If **coresize** is not specified in the command file, the system's user resource limit "coredumpsize" is used (except on HP-UX).

nice_user = <True | False> Normally, when a machine becomes available to Condor, Condor decides which job to run based upon user and job priorities. Setting **nice_user** equal to *True* tells Condor not to use your regular user priority, but that this job should have last priority amongst all users and all jobs. So jobs submitted in this fashion run only on machines which no other non-nice_user job wants — a true "bottom-feeder" job! This is very handy if a user has some jobs they wish to run, but do not wish to use resources that could instead be used to run other people's Condor jobs. Jobs submitted in this fashion have "nice-user." pre-appended in front of the owner name when viewed from *condor_q* or *condor_userprio*. The default value if *False*.

kill_sig = <signal-number> When Condor needs to kick a job off of a machine, it will send the job the signal specified by *signal-number*. *signal-number* needs to be an integer which represents a valid signal on the execution machine. For jobs submitted to the Standard Universe, the default value is the number for `SIGTSTP` which tells the Condor libraries to initiate a checkpoint of the process. For jobs submitted to the Vanilla Universe, the default is `SIGTERM` which is the standard way to terminate a program in UNIX.

+<attribute> = <value> A line which begins with a '+' (plus) character instructs *condor_submit* to simply insert the following *attribute* into the job ClasssAd with the given *value*.

queue [number-of-procs] Places one or more copies of the job into the Condor queue. If desired, new **input**, **output**, **error**, **initialdir**, **arguments**, **nice_user**, **priority**, **kill_sig**, **coresize**, or **image_size** commands may be issued between **queue** commands. This is very handy when submitting multiple runs into one cluster with one submit file; for example, by issuing an **initialdir** between each **queue** command, each run can work in its own subdirectory. The optional argument *number-of-procs* specifies how many times to submit the job to the queue, and defaults to 1.

In addition to commands, the submit-description file can contain macros and comments:

Macros Parameterless macros in the form of `$(macro_name)` may be inserted anywhere in condor description files. Macros can be defined by lines in the form of

```
<macro_name> = <string>
```

Two pre-defined macros are supplied by the description file parser. The `$(Cluster)` macro supplies the number of the job cluster, and the `$(Process)` macro supplies the number of the job. These macros are intended to aid in the specification of input/output files, arguments, etc., for clusters with lots of jobs, and/or could be used to supply a Condor process with its own cluster and process numbers on the command line.

Comments Blank lines and lines beginning with a '#' (pound-sign) character are ignored by the submit-description file parser.

Options

Supported options are as follows:

-v Verbose output - display the created job class-ad

-n ***schedd_name*** Submit to the specified schedd. This option is used when there is more than one schedd running on the submitting machine

-r ***schedd_name*** Submit to a remote schedd. The jobs will be submitted to the schedd on the specified remote host, and their owner will be set to "nobody".

Exit Status

condor_submit will exit with a status value of 0 (zero) upon success, and a non-zero value upon failure.

Examples

Example 1: The below example queues three jobs for execution by Condor. The first will be given command line arguments of '15' and '2000', and will write its standard output to 'foo.out1'. The second will be given command line arguments of '30' and '2000', and will write its standard output to 'foo.out2'. Similarly the third will have arguments of '45' and '6000', and will use 'foo.out3' for its standard output. Standard error output, (if any), from all three programs will appear in 'foo.error'.

```
####################
#
# Example 1: queueing multiple jobs with differing
# command line arguments and output files.
#
####################

Executable     = foo

Arguments      = 15 2000
Output  = foo.out1
Error   = foo.err1
Queue
```



```
Arguments      = 30 2000
Output  = foo.out2
Error   = foo.err2
Queue

Arguments      = 45 6000
Output  = foo.out3
Error   = foo.err3
Queue
```

Example 2: This submit-description file example queues 150 runs of program 'foo' which must have been compiled and linked for Silicon Graphics workstations running IRIX 6.x. Condor will not attempt to run the processes on machines which have less than 32 megabytes of physical memory, and will run them on machines which have at least 64 megabytes if such machines are available. Stdin, stdout, and stderr will refer to "in.0", "out.0", and "err.0" for the first run of this program (process 0). Stdin, stdout, and stderr will refer to "in.1", "out.1", and "err.1" for process 1, and so forth. A log file containing entries about where/when Condor runs, checkpoints, and migrates processes in this cluster will be written into file "foo.log".

```
####################
#
# Example 2: Show off some fancy features including
# use of pre-defined macros and logging.
#
####################

Executable     = foo
Requirements   = Memory >= 32 && OpSys == "IRIX6" && Arch =="SGI"
Rank           = Memory >= 64
Image_Size     = 28 Meg

Error   = err.$(Process)
Input   = in.$(Process)
Output  = out.$(Process)
Log = foo.log

Queue 150
```

General Remarks

- For security reasons, Condor will refuse to run any jobs submitted by user root (UID = 0) or by a user whose default group is group wheel (GID = 0). Jobs submitted by user root or a user with a default group of wheel will appear to sit forever in the queue in an unexpanded state.

- All pathnames specified in the submit-description file must be less than 256 characters in length, and command line arguments must be less than 4096 characters in length; otherwise, *condor_submit* gives a warning message but the jobs will not execute properly.
- Somewhat understandably, behavior gets bizzare if the user makes the silly mistake of requesting multiple Condor jobs to write to the same file, and/or if the user alters any files that need to be accessed by a Condor job which is still in the queue (i.e. compressing of data or output files before a Condor job has completed is a common mistake).

See Also

Condor User Manual

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_userprio

Manage user priorities

Synopsis

condor_userprio [**-res**] [**-set** *username priority*]

Description

condor_userprio with no arguments, lists the priorities of all the users that have a priority value larger than the minimum (The minimum priority value is 0.5, and it corresponds to the highest possible priority). The -set option is used to change a user's priority, and requires special privilage.

A user's priority changes according to his usage of resources. The central manager uses this value during the negotiation phase to decide how many resources to allocate to each user.

Options

Supported options are as follows:

-res Also display the number of resources used by each user. Note that this value reflects the number of resources used at the end of the last negotiation cycle.

-set *username priority* Change priority of the specified user

Author

Condor Team, University of Wisconsin–Madison

Copyright

condor_vacate

Vacate jobs that are running on the specified hosts

Synopsis

condor_vacate [**-help**] [**-version**] [**hostname ...**]

Description

condor_vacate causes the condor startd to checkpoint any running jobs and make them vacate the machine. The job remains in the submitting machine's job queue, however.

If the job on the specified host is running in the Standard Universe, then the job is first checkpointed and then killed (and will then restart somewhere else where it left off). If the job on the specified host is running in the Vanilla Universe, then the job is not checkpointed but is simply killed (and will then restart somewhere else from the beginning). If there is currently no Condor job running on that host, then *condor_vacate* has no effect. Normally there is no need for the user or administrator to explicitly run *condor_vacate*. Vacating a running condor job off of a machine is handled automatically by Condor by following the policies stated in Condor's configuration files.

Options

Supported options are as follows:

-help Display usage information

-version Display version information

Author

Condor Team, University of Wisconsin–Madison

Copyright

See the *Condor Version 6.0.3 Manual* for additional notices.