13 Cluster Workload Management

James Patton Jones, David Lifka, Bill Nitzberg, and Todd Tannenbaum

A Beowulf cluster is a powerful (and attractive) tool. But managing the workload can present significant challenges. It is not uncommon to run hundreds or thousands of jobs or to share the cluster among many users. Some jobs may run only on certain nodes because not all the nodes in the cluster are identical; for instance, some nodes have more memory than others. Some nodes temporarily may not be functioning correctly. Certain users may require priority access to part or all of the cluster. Certain jobs may have to be run at certain times of the day or only after other jobs have completed. Even in the simplest environment, keeping track of all these activities and resource specifics while managing the ever-increasing web of priorities is a complex problem. Workload management software attacks this problem by providing a way to monitor and manage the flow of work through the system, allowing the *best* use of cluster resources as defined by a supplied policy.

Basically, workload management software maximizes the delivery of resources to jobs, given competing user requirements and local policy restrictions. Users package their work into sets of jobs, while the administrator (or system owner) describes local use policies (e.g., Tom's jobs always go first). The software monitors the state of the cluster, schedules work, enforces policy, and tracks usage.

A quick note on terminology: Many terms have been used to describe this area of management software. All of the following topics are related to workload management: distributed resource management, batch queuing, job scheduling, and, resource and task scheduling.

13.1 Goal of Workload Management Software

The goal of workload management software is to make certain the submitted jobs ultimately run to completion by utilizing cluster resources according to a supplied policy. But in order to achieve this goal, workload management systems usually must perform some or all of the following activities:

- Queuing
- Scheduling
- Monitoring
- Resource management
- Accounting

The typical relationship between users, resources, and these workload management activities is depicted in Figure 13.1. As shown in this figure, workload management software sits between the cluster users and the cluster resources. First, users submit jobs to a queue in order to specify the work to be performed. (Once a job has been submitted, the user can request status information about that job at any time.) The jobs then wait in the queue until they are scheduled to start on the cluster. The specifics of the scheduling process are defined by the policy rules. At this point, resource management mechanisms handle the details of properly launching the job and perhaps cleaning up any mess left behind after the job either completes or is aborted. While all this is going on, the workload management system is monitoring the status of system resources and accounting for which users are using what resources.

Image Not Available

Figure 13.1
Activities performed by a workload management system.

13.2 Workload Management Activities

Now let us take a look in more detail at each of the major activities performed by a cluster workload management system.

13.2.1 Queueing

The first of the five aspects of workload management is *queuing*, or the process of collecting together "work" to be executed on a set of resources. This is also the portion most visible to the user.

The tasks the user wishes to have the computer perform, the work, is submitted to the workload management system in a container called a "batch job". The batch job consists of two primary parts: a set of resource directives (such as the amount of memory or number of CPUs needed) and a description of the task to be executed. This description contains all the information the workload management system needs in order to start a user's job when the time comes. For instance, the job description may contain information such as the name of the file to execute, a list of data files required by the job, and environment variables or command-line arguments to pass to the executable.

Once submitted to the workload management system, the batch jobs are held in a "queue" until the matching resources (e.g., the right kind of computers with the right amount of memory or number of CPUs) become available. Examples of real-life queues are lines at the bank or grocery store. Sometimes you get lucky and there's no wait, but usually you have to stand in line for a few minutes. And on days when the resources (clerks) are in high demand (like payday), the wait is substantially longer.

The same applies to computers and batch jobs. Sometimes the wait is very short, and the jobs run immediately. But more often (and thus the need for the workload management system) resources are oversubscribed, and so the jobs have to wait.

One important aspect of queues is that limits can be set that restrict access to the queue. This allows the cluster manager greater control over the usage policy of the cluster. For example, it may be desirable to have a queue that is available for short jobs only. This would be analogous to the "ten items or fewer express lane" at the grocery store, providing a shorter wait for "quick tasks."

Each of the different workload management systems discussed later in this volume offers a rich variety of queue limits and attributes.

13.2.2 Scheduling

The second area of workload management is scheduling, which is simply the process of choosing the *best* job to run. Unlike in our real-life examples of the bank and grocery store (which employ a simple first-come, first-served model of deciding who's next), workload management systems offer a variety of ways by which the *best* job is identified.

As we have discussed earlier, however, *best* can be a tricky goal, and depends on the usage policy set by local management, the available workload, the type and availability of cluster resources, and the types of application being run on the cluster. In general, however, scheduling can be broken into two primary activities: *policy enforcement* and *resource optimization.*

Policy encapsulates how the cluster resources are to be used, addressing such issues as priorities, traffic control, and capability vs. high throughput. Scheduling is then the act of enforcing the policy in the selection of jobs, ensuring the priorities are met and policy goals are achieved.

While implementing and enforcing the policy, the scheduler has a second set of goals. These are resource optimization goals, such as "pack jobs efficiently" or "exploit underused resources."

The difficult part of scheduling, then, is balancing policy enforcement with resource optimization in order to pick the *best* job to run.

Logically speaking, one could think of a scheduler as performing the following loop:

1. Select the best job to run, according to policy and available resources.
2. Start the job.
3. Stop the job and/or clean up after a completed job.
4. Repeat.

The nuts and bolts of scheduling is, of course, choosing and tuning the policy to meet your needs. Although different workload management systems each have their own idiosyncrasies, they typically all provide ways in which their scheduling policy can be customized. Subsequent chapters of this book will discuss the various scheduling policy mechanisms available in several popular workload management systems.

13.2.3 Monitoring

Resource monitoring is the third part of any cluster workload management system. It provides necessary information to administrators, users and the scheduling system itself on the status of jobs and resources. There are basically three critical times that resource monitoring comes into play:

1. When nodes are idle, to verify that they are in working order before starting another job on them.

2. When nodes are busy running a job. Users and administrators may want to check memory, CPU, network, I/O, and utilization of other system resources. Such checks often are useful in parallel programming when users wish to verify that they have balanced their workload correctly and are effectively using all the nodes they've been allocated.

3. When a job completes. Here, resource monitoring is used to ensure that there are no remaining processes from the completed job and that the node is still in working order before starting another job on it.

Workload management systems query the compute resources at these times and use the information to make informed decisions about running jobs. Much of the information is cached so that it can be reported quickly in answer to status requests. Some information is saved for historical analysis purposes. Still other bits of the information are used in the enforcement of local policy. The method of collection may differ between different workload management systems, but the general purposes are the same.

13.2.4 Resource Management

The fourth area, resource management, is essentially responsible for the starting, stopping, and cleaning up after jobs that are run on cluster nodes. In a batch system resource management involves running a job for a user, under the identity of the user, on the resources the user was allocated in such a way that the user need not be present at that time.

Many cluster workload management systems provide mechanisms to ensure the successful startup and cleanup of jobs and to maintain node status data internally, so that jobs are started only on nodes that are available and functioning correctly.

In addition, limits may need to be placed on the job and enforced by the workload management system. These limits are yet another aspect of policy enforcement, in addition to the limits on queues and those enacted by the scheduling component.

Another aspect of resource management is providing the ability to remove or add compute resources to the available pool of systems. Clusters are rarely static; systems go down, or new nodes are added. The "registration" of new nodes and the marking of nodes as unavailable are both additional aspects of resource management.

13.2.5 Accounting

The fifth aspect of workload management is accounting and reporting. Workload accounting is the process of collecting resource usage data for the batch jobs that run on the cluster. Such data includes the job owner, resources requested by the job, and total amount of resources consumed by the job. Other data about the job may also be available, depending on the specific workload managment system in use.

Cluster workload accounting data can used for a variety of purposes, such as

1. producing weekly system usage reports,
2. preparing monthly per user usage reports,
3. enforcing per project allocations,
4. tuning the scheduling policy,
5. calculating future resource allocations,
6. anticipating future computer component requirements, and
7. determining areas of improvement within the computer system.

The data for these purposes may be collected as part of the resource monitoring tasks or may be gathered separately. In either case, data is pulled from the available sources in order to meet the objectives of workload accounting. Details of using the workload accounting features of specific workload management systems are discussed in subsequent chapters of this book.

14 Condor: A Distributed Job Scheduler

Todd Tannenbaum, Derek Wright, Karen Miller, and Miron Livny

Condor is a sophisticated and unique distributed job scheduler developed by the Condor research project at the University of Wisconsin-Madison Department of Computer Sciences.

A public-domain version of the Condor software and complete documentation is freely available from the Condor project's Web site at `www.cs.wisc.edu/condor`. Organizations may purchase a commercial version of Condor with an accompanying support contract; for additional information see `www.condorcomputing.com`.

This chapter introduces all aspects of Condor, from its ability to satisfy the needs and desires of both submitters and resource owners, to the management of Condor on clusters. Following an overview of Condor and Condor's ClassAd mechanism is a description of Condor from the user's perspective. The architecture of the software is presented along with overviews of installation and management. The chapter ends with configuration scenarios specific to clusters.

14.1 Introduction to Condor

Condor is a specialized workload management system for compute-intensive jobs. Like other full-featured batch systems, Condor provides a job queuing mechanism, scheduling policy, priority scheme, resource monitoring, and resource management. Users submit their jobs to Condor, and Condor places them into a queue, chooses when and where to run them based upon a policy, monitors their progress, and ultimately informs the user upon completion.

While providing functionality similar to that of a more traditional batch queuing system, Condor's novel architecture allows it to succeed in areas where traditional scheduling systems fail. Condor can be used to manage a cluster of dedicated Beowulf nodes. In addition, several unique mechanisms enable Condor to effectively harness wasted CPU power from otherwise idle desktop workstations. Condor can be used to seemlessly combine all of your organization's computational power into one resource.

Condor is the product of the Condor Research Project at the University of Wisconsin-Madison (UW-Madison) and was first installed as a production system in the UW-Madison Department of Computer Sciences nearly ten years ago. This Condor installation has since served as a major source of computing cycles to UW-Madison faculty and students. Today, just in our department alone, Condor manages more than one thousand workstations, including the department's 500-CPU Linux Beowulf cluster. On a typical day, Condor delivers more than 650 CPU-*days* to UW researchers. Additional Condor installations have been established over the years across our campus and the world. Hundreds of organizations in industry, government, and academia have used Condor to establish compute environments ranging in size from a handful to hundreds of workstations.

14.1.1 Features of Condor

Condor's features are extensive. Condor provides great flexibility for both the user submitting jobs and for the owner of a machine that provides CPU time toward running jobs. The following list summarizes some of Condor's capabilities.

Distributed submission: There is no single, centralized submission machine. Instead, Condor allows jobs to be submitted from many machines, and each machine contains its own job queue. Users may submit to a cluster from their own desktop machines.

Job priorities: Users can assign priorities to their submitted jobs in order to control the execution order of the jobs. A "nice-user" mechanism requests the use of only those machines that would have otherwise been idle.

User priorities: Administrators may assign priorities to users using a flexible mechanism that enables a policy of fair share, strict ordering, fractional ordering, or a combination of policies.

Job dependency: Some sets of jobs require an ordering because of dependencies between jobs. "Start job X only after jobs Y and Z successfully complete" is an example of a dependency. Enforcing dependencies is easily handled.

Support for multiple job models: Condor handles both serial jobs and parallel jobs incorporating PVM, dynamic PVM, and MPI.

ClassAds: The ClassAd mechanism in Condor provides an extremely flexible and expressive framework for matching resource requests (jobs) with resource offers (machines). Jobs can easily state both job requirements and job preferences. Likewise, machines can specify requirements and preferences about the jobs they are willing to run. These requirements and preferences can be described in powerful expressions, resulting in Condor's adaptation to nearly any desired policy.

Job checkpoint and migration: With certain types of jobs, Condor can transparently take a checkpoint and subsequently resume the application. A checkpoint is a snapshot of a job's complete state. Given a checkpoint, the job can later continue its execution from where it left off at the time of the checkpoint. A checkpoint also enables the transparent migration of a job from one machine to another machine.

Periodic checkpoint: Condor can be configured to periodically produce a checkpoint for a job. This provides a form of fault tolerance and safeguards the accumulated computation time of a job. It reduces the loss in the event of a system failure such as the machine being shut down or hardware failure.

Job suspend and resume: Based on policy rules, Condor can ask the operating system to suspend and later resume a job.

Remote system calls: Despite running jobs on remote machines, Condor can often preserve the local execution environment via remote system calls. Users do not need to make data files available or even obtain a login account on remote workstations before Condor executes their programs there. The program behaves under Condor as if it were running as the user that submitted the job on the workstation where it was originally submitted, regardless of where it really executes.

Pools of machines working together: *Flocking* allows jobs to be scheduled across multiple Condor pools. It can be done across pools of machines owned by different organizations that impose their own policies.

Authentication and authorization: Administrators have fine-grained control of access permissions, and Condor can perform strong network authentication using a variety of mechanisms including Kerberos and X.509 public key certificates.

Heterogeneous platforms: In addition to Linux, Condor has been ported to most of the other primary flavors of Unix as well as Windows NT. A single pool can contain multiple platforms. Jobs to be executed under one platform may be submitted from a different platform. As an example, an executable that runs under Windows 2000 may be submitted from a machine running Linux.

Grid computing: Condor incorporates many of the emerging Grid-based computing methodologies and protocols. It can interact with resources managed by Globus.

14.1.2 Understanding Condor ClassAds

The ClassAd is a flexible representation of the characteristics and constraints of both machines and jobs in the Condor system. *Matchmaking* is the mechanism by which Condor matches an idle job with an available machine. Understanding this unique framework is the key to harness the full flexibility of the Condor system.

ClassAds are employed by users to specify which machines should service their jobs. Administrators use them to customize scheduling policy.

Conceptualizing Condor ClassAds: Just Like the Newspaper. 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 list constraints that must be satisfied. For instance, a buyer has a maximum spending limit, and a seller requires a minimum purchase price. Furthermore, both want to rank requests to their own advantage. Certainly a seller would rank one offer of $50 higher than a different offer of $25. In Condor, users submitting jobs can be thought of as buyers of compute resources and machine owners are sellers.

All machines in a Condor pool advertise their attributes, such as available RAM memory, CPU type and speed, virtual memory size, current load average, current time and date, and other static and dynamic properties. This machine ClassAd also advertises under what conditions it is willing to run a Condor job and what type of job it prefers. These policy attributes can reflect the individual terms and preferences by which the different owners have allowed their machines to participate in the Condor pool.

After a job is submitted to Condor, a job ClassAd is created. This ClassAd includes attributes about the job, such as the amount of memory the job uses, the name of the program to run, the user who submitted the job, and the time it was submitted. The job can also specify requirements and preferences (or *rank*) for the machine that will run the job. For instance, perhaps you are looking for the fastest floating-point performance available. You want Condor to rank available machines based on floating-point performance. Perhaps you care only that the machine has a minimum of 256 MBytes of RAM. Or, perhaps you will take any machine you can get! These job attributes and requirements are bundled up into a job ClassAd.

Condor plays the role of matchmaker by continuously reading all the job ClassAds and all the machine ClassAds, matching and ranking job ads with machine ads. Condor ensures that the requirements in both ClassAds are satisfied.

Structure of a ClassAd. A ClassAd is a set of uniquely named expressions. Each named expression is called an *attribute*. Each attribute has an *attribute name* and an *attribute value*. The attribute value can be a simple integer, string, or floating-point value, such as

```
Memory = 512
OpSys = "LINUX"
```

```
        NetworkLatency = 7.5
```

An attribute value can also consist of a logical expression that will evaluate to TRUE, FALSE, or UNDEFINED. The syntax and operators allowed in these expressions are similar to those in C or Java, that is, `==` for equals, `!=` for not equals, `&&` for logical **and**, `||` for logical **or**, and so on. Furthermore, ClassAd expressions can incorporate attribute names to refer to other attribute values. For instance, consider the following small sample ClassAd:

```
        MemoryInMegs = 512
        MemoryInBytes = MemoryInMegs * 1024 * 1024
        Cpus = 4
        BigMachine = (MemoryInMegs > 256) && (Cpus >= 4)
        VeryBigMachine = (MemoryInMegs > 512) && (Cpus >= 8)
        FastMachine = BigMachine && SpeedRating
```

In this example, `BigMachine` evaluates to TRUE and `VeryBigMachine` evaluates to FALSE. But, because attribute `SpeedRating` is not specified, `FastMachine` would evaluate to UNDEFINED.

Condor provides *meta-operators* that allow you to explicitly compare with the UNDEFINED value by testing both the type and value of the operands. If both the types and values match, the two operands are considered *identical*; `=?=` is used for meta-equals (or, is-identical-to) and `=!=` is used for meta-not-equals (or, is-not-identical-to). These operators always return TRUE or FALSE and therefore enable Condor administrators to specify explicit policies given incomplete information.

A complete description of ClassAd semantics and syntax is documented in the Condor manual.

Matching ClassAds. ClassAds can be matched with one another. This is the fundamental mechanism by which Condor matches jobs with machines. Figure 14.1 displays a ClassAd from Condor representing a machine and another representing a queued job. Each ClassAd contains a `MyType` attribute, describing what type of resource the ad represents, and a `TargetType` attribute. The `TargetType` specifies the type of resource desired in a match. Job ads want to be matched with machine ads and vice versa.

Each ClassAd engaged in matchmaking specifies a `Requirements` and a `Rank` attribute. In order for two ClassAds to match, the `Requirements` expression in both ads must evaluate to TRUE. An important component of matchmaking is the `Requirements` and `Rank` expression can refer not only to attributes in their own ad but also to attributes in the candidate matching ad. For instance, the

Job ClassAd	Machine ClassAd
MyType = “Job”	**MyType** = “Machine”
TargetType = “Machine”	**TargetType** = “Job”
Requirements = ((Arch==“INTEL” && OpSys==“LINUX”) && Disk > DiskUsage)	**Requirements** = Start
Rank = (Memory * 10000) + KFlops	**Rank** = TARGET.Department==MY.Department
Args = “-ini ./ies.ini”	**Activity** = “Idle”
ClusterId = 680	**Arch** = “INTEL”
Cmd = “/home/tannenba/bin/sim-exe”	**ClockDay** = 0
Department = “CompSci”	**ClockMin** = 614
DiskUsage = 465	**CondorLoadAvg** = 0.000000
StdErr = “sim.err”	**Cpus** = 1
ExitStatus = 0	**CurrentRank** = 0.000000
FileReadBytes = 0.000000	**Department** = “CompSci”
FileWriteBytes = 0.000000	**Disk** = 3076076
ImageSize = 465	**EnteredCurrentActivity** = 990371564
StdIn = “/dev/null”	**EnteredCurrentState** = 990330615
Iwd = “/home/tannenba/sim-m/run_55”	**FileSystemDomain** = “cs.wisc.edu”
JobPrio = 0	**IsInstructional** = FALSE
JobStartDate = 971403010	**KeyboardIdle** = 15
JobStatus = 2	**KFlops** = 145811
StdOut = “sim.out”	**LoadAvg** = 0.220000
Owner = “tannenba”	**Machine** = “nostos.cs.wisc.edu”
ProcId = 64	**Memory** = 511
QDate = 971377131	**Mips** = 732
RemoteSysCpu = 0.000000	**OpSys** = “LINUX”
RemoteUserCpu = 0.000000	**Start** = (LoadAvg <= 0.300000) && (KeyboardIdle > (15 * 60))
RemoteWallClockTime = 2401399.000000	**State** = “Unclaimed”
TransferFiles = “NEVER”	**Subnet** = “128.105.165”
WantCheckpoint = FALSE	**TotalVirtualMemory** = 787144
WantRemoteSyscalls = FALSE	⋮
⋮	

Figure 14.1
Examples of ClassAds in Condor.

`Requirements` expression for the job ad specified in Figure 14.1 refers to `Arch`, `OpSys`, and `Disk`, which are all attributes found in the machine ad.

What happens if Condor finds more than one machine ClassAd that satisfies the constraints specified by `Requirements`? That is where the `Rank` expression comes into play. The `Rank` expression specifies the desirability of the match (where higher numbers mean better matches). For example, the job ad in Figure 14.1 specifies

```
Requirements = ((Arch=="INTEL" && OpSys=="LINUX") && Disk > DiskUsage)
Rank         = (Memory * 100000) + KFlops
```

In this case, the job requires a computer running the Linux operating system and more local disk space than it will use. Among all such computers, the user prefers those with large physical memories and fast floating-point CPUs (`KFlops` is a metric of floating-point performance). Since the `Rank` is a user-specified metric, *any* expression may be used to specify the perceived desirability of the match. Condor's matchmaking algorithms deliver the best resource (as defined by the `Rank` expression) while satisfying other criteria.

14.2 Using Condor

The road to using Condor effectively is a short one. The basics are quickly and easily learned.

14.2.1 Roadmap to Using Condor

The following steps are involved in running jobs using Condor:

Prepare the Job to Run Unattended. An application run under Condor must be able to execute as a batch job. Condor runs the program unattended and in the background. A program that runs in the background will not be able to perform interactive input and output. Condor can redirect console output (`stdout` and `stderr`) and keyboard input (`stdin`) to and from files. You should create any needed files that contain the proper keystrokes needed for program input. You should also make certain the program will run correctly with the files.

Select the Condor Universe. Condor has five runtime environments from which to choose. Each runtime environment is called a *Universe.* Usually the Universe you choose is determined by the type of application you are asking Condor to run. There are six job Universes in total: two for serial jobs (Standard and Vanilla), one for parallel PVM jobs (PVM), one for parallel MPI jobs (MPI), one for Grid applications (Globus), and one for meta-schedulers (Scheduler). Section 14.2.4 provides more information on each of these Universes.

Create a Submit Description File. The details of a job submission are defined in a *submit description* file. This file contains information about the job such as what executable to run, which Universe to use, the files to use for `stdin, stdout`, and `stderr`, requirements and preferences about the machine which should run the program, and where to send e-mail when the job completes. You can also tell Condor how many times to run a program; it is simple to run the same program multiple times with different data sets.

Submit the Job. Submit the program to Condor with the `condor_submit` command.

Once a job has been submitted, Condor handles all aspects of running the job. You can subsequently monitor the job's progress with the `condor_q` and `condor_status` commands. You may use `condor_prio` to modify the order in which Condor will run your jobs. If desired, Condor can also record what is being done with your job at every stage in its lifecycle, through the use of a log file specified during submission.

When the program completes, Condor notifies the owner (by e-mail, the user-specified log file, or both) the exit status, along with various statistics including time used and I/O performed. You can remove a job from the queue at any time with `condor_rm`.

14.2.2 Submitting a Job

To submit a job for execution to Condor, you use the `condor_submit` command. This command takes as an argument the name of the submit description file, which contains commands and keywords to direct the queuing of jobs. In the submit description file, you define everything Condor needs to execute the job. Items such as the name of the executable to run, the initial working directory, and command-line arguments to the program all go into the submit description file. The `condor_submit` command creates a job ClassAd based on the information, and Condor schedules the job.

The contents of a submit description file can save you considerable time when you are using Condor. It is easy to submit multiple runs of a program to Condor. To run the same program 500 times on 500 different input data sets, the data files are arranged such that each run reads its own input, and each run writes its own output. Every individual run may have its own initial working directory, **`stdin, stdout, stderr,`** command-line arguments, and shell environment.

The following examples illustrate the flexibility of using Condor. We assume that the jobs submitted are serial jobs intended for a cluster that has a shared file system across all nodes. Therefore, all jobs use the Vanilla Universe, the simplest one for running serial jobs. The other Condor Universes are explored later.

Example 1. Example 1 is the simplest submit description file possible. It queues up one copy of the program '`foo`' for execution by Condor. A log file called '`foo.log`' is generated by Condor. The log file contains events pertaining to the job while it runs inside of Condor. When the job finishes, its exit conditions are noted in the log file. We recommend that you always have a log file so you know what happened to your jobs. The *queue* statement in the submit description file tells Condor to use all the information specified so far to create a job ClassAd and place the job into the queue. Lines that begin with a pound character (#) are comments and are ignored by `condor_submit`.

```
# Example 1 : Simple submit file
universe = vanilla
executable = foo
log = foo.log
queue
```

Example 2. Example 2 queues two copies of the program '`mathematica`'. The first copy runs in directory '`run_1`', and the second runs in directory '`run_2`'. For both queued copies, '`stdin`' will be '`test.data`', '`stdout`' will be '`loop.out`', and '`stderr`' will be '`loop.error`'. Two sets of files will be written, since the files are each written to their own directories. This is a convenient way to organize data for a large group of Condor jobs.

```
# Example 2: demonstrate use of multiple
# directories for data organization.
universe = vanilla
executable = mathematica
# Give some command line args, remap stdio
arguments = -solver matrix
input = test.data
output = loop.out
error = loop.error
log = loop.log

initialdir = run_1
queue
initialdir = run_2
queue
```

Example 3. The submit description file for Example 3 queues 150 runs of program '`foo`'. This job requires Condor to run the program on machines that have greater than 128 megabytes of physical memory, and it further requires that the job not be scheduled to run on a specific node. Of the machines that meet the requirements, the job prefers to run on the fastest floating-point nodes currently available to accept the job. It also advises Condor that the job will use up to 180 megabytes of memory when running. Each of the 150 runs of the program is given its own process number, starting with process number 0. Several built-in macros can be used in a submit description file; one of them is the *$(Process)* macro which Condor expands to be the process number in the job cluster. This causes files '`stdin`', '`stdout`', and '`stderr`' to be '`in.0`', '`out.0`', and '`err.0`' for the first run of the program, '`in.1`', '`out.1`', and '`err.1`' for the second run of the program, and so forth. A single log file will list events for all 150 jobs in this job cluster.

```
# Example 3: Submit lots of runs and use the
# pre-defined $(Process) macro.
universe = vanilla
executable = foo
requirements = Memory > 128  && Machine != "server-node.cluster.edu"
rank = KFlops
image_size = 180

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

queue 150
```

Note that the `requirements` and `rank` entries in the submit description file will become the requirements and rank attributes of the subsequently created ClassAd for this job. These are arbitrary expressions that can reference any attributes of either the machine or the job; see Section 14.1.2 for more on requirements and rank expressions in ClassAds.

14.2.3 Overview of User Commands

Once you have jobs submitted to Condor, you can manage them and monitor their progress. Table 14.1 shows several commands available to the Condor user to view the job queue, check the status of nodes in the pool, and perform several other activities. Most of these commands have many command-line options; see the Command Reference chapter of the Condor manual for complete documentation. To provide an introduction from a user perspective, we give here a quick tour showing several of these commands in action.

When jobs are submitted, Condor will attempt to find resources to service the jobs. A list of all users with jobs submitted may be obtained through `condor_status` with the *-submitters* option. An example of this would yield output similar to the following:

```
% condor_status -submitters

Name                    Machine      Running IdleJobs HeldJobs

ballard@cs.wisc.edu  bluebird.c         0       11        0
nice-user.condor@cs. cardinal.c         6      504        0
wright@cs.wisc.edu   finch.cs.w         1        1        0
jbasney@cs.wisc.edu  perdita.cs         0        0        5
```

Command	Description
condor_checkpoint	Checkpoint jobs running on the specified hosts
condor_compile	Create a relinked executable for submission to the Standard Universe
condor_glidein	Add a Globus resource to a Condor pool
condor_history	View log of Condor jobs completed to date
condor_hold	Put jobs in the queue in hold state
condor_prio	Change priority of jobs in the queue
condor_qedit	Modify attributes of a previously submitted job
condor_q	Display information about jobs in the queue
condor_release	Release held jobs in the queue
condor_reschedule	Update scheduling information to the central manager
condor_rm	Remove jobs from the queue
condor_run	Submit a shell command-line as a Condor job
condor_status	Display status of the Condor pool
condor_submit_dag	Manage and queue jobs within a specified DAG for interjob dependencies.
condor_submit	Queue jobs for execution
condor_userlog	Display and summarize job statistics from job log files

Table 14.1
List of user commands.

```
                         RunningJobs          IdleJobs          HeldJobs

 ballard@cs.wisc.edu               0                11                 0
 jbasney@cs.wisc.edu               0                 0                 5
nice-user.condor@cs.               6               504                 0
  wright@cs.wisc.edu               1                 1                 0

               Total               7               516                 5
```

Checking on the Progress of Jobs. The condor_q command displays the status of all jobs in the queue. An example of the output from condor_q is

```
% condor_q

-- Schedd: uug.cs.wisc.edu : <128.115.121.12:33102>
 ID        OWNER            SUBMITTED     RUN_TIME ST PRI SIZE CMD
 55574.0   jane            6/23 11:33   4+03:35:28 R  0   25.7 seycplex seymour.d
 55575.0   jane            6/23 11:44   0+23:24:40 R  0   26.8 seycplexpseudo sey
 83193.0   jane            3/28 15:11  48+15:50:55 R  0   17.5 cplexmip test1.mp
 83196.0   jane            3/29 08:32  48+03:16:44 R  0   83.1 cplexmip test3.mps
 83212.0   jane            4/13 16:31  41+18:44:40 R  0   39.7 cplexmip test2.mps
 5 jobs; 0 idle, 5 running, 0 held
```

This output contains many columns of information about the queued jobs. The `ST` column (for status) shows the status of current jobs in the queue. An `R` in the status column means the the job is currently running. An `I` stands for idle. The status `H` is the hold state. In the hold state, the job will not be scheduled to run until it is released (via the `condor_release` command). The `RUN_TIME` time reported for a job is the time that job has been allocated to a machine as DAYS+HOURS+MINS+SECS.

Another useful method of tracking the progress of jobs is through the user log. 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. Here is a sample snippet from a user log file

```
000 (8135.000.000) 05/25 19:10:03 Job submitted from host: <128.105.146.14:1816>
...
001 (8135.000.000) 05/25 19:12:17 Job executing on host: <128.105.165.131:1026>
...
005 (8135.000.000) 05/25 19:13:06 Job terminated.
        (1) Normal termination (return value 0)
                        Usr 0 00:00:37, Sys 0 00:00:00  -  Run Remote Usage
                        Usr 0 00:00:00, Sys 0 00:00:05  -  Run Local Usage
                        Usr 0 00:00:37, Sys 0 00:00:00  -  Total Remote Usage
                        Usr 0 00:00:00, Sys 0 00:00:05  -  Total Local Usage
        9624  -  Run Bytes Sent By Job
        7146159  -  Run Bytes Received By Job
        9624  -  Total Bytes Sent By Job
        7146159  -  Total Bytes Received By Job
...
```

The `condor_jobmonitor` tool parses the events in a user log file and can use the information to graphically display the progress of your jobs. Figure 14.2 contains a screenshot of `condor_jobmonitor` in action.

You can locate all the machines that are running your job with 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    LINUX         Claimed    Busy       0.980  64    0+07:10:02
```

Image Not Available

Figure 14.2
Condor jobmonitor tool.

```
biron.cs.w INTEL    LINUX        Claimed    Busy       1.000  128    0+01:10:00
cambridge. INTEL    LINUX        Claimed    Busy       0.988  64     0+00:15:00
falcons.cs INTEL    LINUX        Claimed    Busy       0.996  32     0+02:05:03
happy.cs.w INTEL    LINUX        Claimed    Busy       0.988  128    0+03:05:00
istat03.st INTEL    LINUX        Claimed    Busy       0.883  64     0+06:45:01
istat04.st INTEL    LINUX        Claimed    Busy       0.988  64     0+00:10:00
istat09.st INTEL    LINUX        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    LINUX        0.980  hepcon@cs.wisc.edu   chevre.cs.wisc.
```

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

Removing a Job from the Queue. You can remove a job from the queue at any time 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 is removed. The following example shows the queue of jobs before and after a job is removed.

```
%  condor_q

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

2 jobs; 1 idle, 1 running, 0 held

%  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    RUN_TIME  ST PRI SIZE CMD
 125.0   jbasney         4/10 15:35   0+00:00:00 I  -10 1.2  hello.remote

1 jobs; 1 idle, 0 running, 0 held
```

Changing the Priority of Jobs. In addition to the priorities assigned to each user, Condor provides users with the capability of assigning priorities to any 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. Job priorities can be modified using the `condor_prio` command. For example, to change the priority of a job to −15, type

```
%  condor_q raman
```

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

1 jobs; 1 idle, 0 running, 0 held

%  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    RUN_TIME  ST PRI SIZE CMD
 126.0   raman           4/11 15:06   0+00:00:00 I  -15 0.3  hello

1 jobs; 1 idle, 0 running, 0 held
```

We emphasize that these *job* priorities are completely different from the *user* priorities assigned by Condor. Job priorities control only which one of *your* jobs should run next; there is no effect whatsoever on whether your jobs will run before another user's jobs.

Determining Why a Job Does Not Run. A specific job may not run for several reasons. These reasons include failed job or machine constraints, bias due to preferences, insufficient priority, and 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 had not run for several days.

```
% condor_q

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

1 jobs; 1 idle, 0 running, 0 held
```

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 == "IRIX6" &&
  Disk >= ExecutableSize && VirtualMemory >= ImageSize
```

The `Requirements` expression for this job specifies a platform that does not exist. Therefore, the expression always evaluates to FALSE.

While the analyzer can diagnose most common problems, there are some situations that it cannot reliably detect because of the instantaneous and local nature of the information it uses to detect the problem. The analyzer may report 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, the job may begin to run but immediately terminates and return to the idle state. 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, you should contact your system administrator.

Job Completion. When a Condor job completes (either through normal means or abnormal means), Condor will remove it from the job queue (therefore, it will no longer appear in the output of `condor_q`) and insert it into the job history file. You can examine the job history file with the `condor_history` command. If you specified a log file in your submit description file, then the job exit status will be recorded there as well.

By default, Condor will send you an e-mail message when your job completes. You can modify this behavior with the `condor_submit` "notification" command. The message will include the exit status of your job or notification that your job terminated abnormally.

14.2.4 Submitting Different Types of Jobs: Alternative Universes

A Universe in Condor defines an execution environment. Condor supports the following Universes on Linux:

- Vanilla
- MPI
- PVM
- Globus
- Scheduler
- Standard

The `Universe` attribute is specified in the submit description file. If the Universe is not specified, it will default to Standard.

Vanilla Universe. The Vanilla Universe is used to run serial (nonparallel) jobs. The examples provided in the preceding section use the Vanilla Universe. Most Condor users prefer to use the Standard Universe to submit serial jobs because of several helpful features of the Standard Universe. However, the Standard Universe has several restrictions on the types of serial jobs supported. The Vanilla Universe, on the other hand, has no such restrictions. Any program that runs outside of Condor will run in the Vanilla Universe. Binary executables as well as scripts are welcome in the Vanilla Universe.

A typical Vanilla Universe job relies on a shared file system between the submit machine and all the nodes in order to allow jobs to access their data. However, if a shared file system is not available, Condor can transfer the files needed by the job to and from the execute machine. See Section 14.2.5 for more details on this.

MPI Universe. The MPI Universe allows parallel programs written with MPI to be managed by Condor. To submit an MPI program to Condor, specify the number of nodes to be used in the parallel job. Use the `machine_count` attribute in the submit description file, as in the following example:

```
# Submit file for an MPI job which needs 8 large memory nodes
universe = mpi
executable = my-parallel-job
requirements = Memory >= 512
machine_count = 8
queue
```

Further options in the submit description file allow a variety of parameters, such as the job requirements or the executable to use across the different nodes.

By late 2001, Condor expects your MPI job to be linked with the MPICH implementation of MPI configured with the `ch_p4` device (see Section 9.6.1). Support for different devices and MPI implementations is expected, however, so check the documentation included with your specific version of Condor for additional information on how your job should be linked with MPI for Condor.

If your Condor pool consists of both dedicated compute machines (that is, Beowulf cluster nodes) and opportunistic machines (that is, desktop workstations), by default Condor will schedule MPI jobs to run on the dedicated resources only.

PVM Universe. Several different parallel programming paradigms exist. 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 out pieces of work 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 it assigns the next piece of work out to the next worker that becomes available.

The PVM Universe allows master/worker style parallel programs written for the Parallel Virtual Machine interface (see Chapter 11) to be used with Condor. Condor runs the master application on the machine where the job was submitted and will not preempt the master application. Workers are pulled in from the Condor pool as they become available.

Specifically, in the PVM Universe, Condor acts as the resource manager for the PVM daemon. Whenever a PVM program asks for nodes via a `pvm_addhosts()` call, the request is forwarded to Condor. Using ClassAd matching mechanisms, Condor finds a machine in the Condor pool and adds it to the virtual machine. If a machine needs to leave the pool, the PVM program is notified by normal PVM mechanisms, for example, the `pvm_notify()` call.

A unique aspect of the PVM Universe is that PVM jobs submitted to Condor can harness both dedicated and nondedicated (opportunistic) workstations throughout the pool by dynamically adding machines to and removing machines from the parallel virtual machine as machines become available.

Writing a PVM program that deals with Condor's opportunistic environment can be a tricky task. For that reason, the MW framework has been created. MW is a tool for making master-worker style applications in Condor's PVM Universe. For more information, see the MW Home page online at `www.cs.wisc.edu/condor/mw`.

Submitting to the PVM Universe is similar to submitting to the MPI Universe, except that the syntax for `machine_count` is different to reflect the dynamic nature of the PVM Universe. Here is a simple sample submit description file:

```
# Require Condor to give us one node before starting
# the job, but we'll use up to 75 nodes if they are
# available.
universe = pvm
executable = master.exe
machine_count = 1..75
queue
```

By using `machine_count = <min>..<max>`, the submit description file tells Condor that before the PVM master is started, there should be at least `<min>` number of machines given to the job. It also asks Condor to give it as many as `<max>` machines.

More detailed information on the PVM Universe is available in the Condor manual as well as on the Condor-PVM home page at URL `www.cs.wisc.edu/condor/pvm`.

Globus Universe. The Globus Universe in Condor is intended to provide the standard Condor interface to users who wish to submit jobs to machines being managed by Globus (`www.globus.org`).

Scheduler Universe. The Scheduler Universe is used to submit a job that will immediately run on the *submit* machine, as opposed to a remote execution machine. The purpose is to provide a facility for job *meta-schedulers* that desire to manage the submission and removal of jobs into a Condor queue. Condor includes one such meta-scheduler that utilizes the Scheduler Universe: the DAGMan scheduler, which can be used to specify complex interdependencies between jobs. See Section 14.2.6 for more on DAGMan.

Standard Universe. The Standard Universe requires minimal extra effort on the part of the user but provides a serial job with the following highly desirable services:

- Transparent process *checkpoint* and *restart*
- Transparent process migration
- Remote system calls
- Configurable file I/O buffering
- On-the-fly file compression/inflation

Process Checkpointing in the Standard Universe. A checkpoint of an executing program is a snapshot of the program's current state. It provides a way for the program to be continued from that state at a later time. Using checkpoints gives Condor the freedom to reconsider scheduling decisions through preemptive-resume scheduling. If the scheduler decides to rescind a machine that is running a

Condor job (for example, when the owner of that machine returns and reclaims it or when a higher-priority user desires the same machine), the scheduler can take a checkpoint of the job and preempt the job without losing the work the job has already accomplished. The job can then be resumed later when the Condor scheduler allocates it a new machine. Additionally, periodic checkpoints provide fault tolerance. Normally, when performing long-running computations, if a machine crashes or must be rebooted for an administrative task, all the work that has been done is lost. The job must be restarted from the beginning, which can mean days, weeks, or even months of wasted computation time. With checkpoints, Condor ensures that progress is always made on jobs and that only the computation done since the last checkpoint is lost. Condor can be take checkponts periodically, and after an interruption in service, the program can continue from the most recent snapshot.

To enable taking checkpoints, you do not need to change the program's source code. Instead, the program must be relinked with the Condor system call library (see below). Taking the checkpoint of a process is implemented in the Condor system call library as a signal handler. When Condor sends a checkpoint signal to a process linked with this library, the provided signal handler writes the state of the process out to a file or a network socket. This state includes the contents of the process's stack and data segments, all CPU state (including register values), the state of all open files, and any signal handlers and pending signals. When a job is to be continued using a checkpoint, Condor reads this state from the file or network socket, restoring the stack, shared library and data segments, file state, signal handlers, and pending signals. The checkpoint signal handler then restores the CPU state and returns to the user code, which continues from where it left off when the checkpoint signal arrived. Condor jobs submitted to the Standard Universe will automatically perform a checkpoint when preempted from a machine. When a suitable replacement execution machine is found (of the same architecture and operating system), the process is restored on this new machine from the checkpoint, and computation is resumed from where it left off.

By default, a checkpoint is written to a file on the local disk of the submit machine. A Condor checkpoint server is also available to serve as a repository for checkpoints.

Remote System Calls in the Standard Universe. One hurdle to overcome when placing an job on a remote execution workstation is data access. In order to utilize the remote resources, the job must be able to read from and write to files on its submit machine. A requirement that the remote execution machine be able to access these files via NFS, AFS, or any other network file system may significantly limit the number of eligible workstations and therefore hinder the ability of an environment to achieve high throughput. Therefore, in order to maximize throughput, Condor strives to be able to run any application on any remote workstation of a given platform without relying upon a common administrative setup. The enabling technology that permits this is Condor's Remote System Calls mechanism. This mechanism provides the benefit that Condor does not require a user to possess a login account on the execute workstation.

When a Unix process needs to access a file, it calls a file I/O system function such as `open()`, `read()`, or `write()`. These functions are typically handled by the standard C library, which consists primarily of stubs that generate a corresponding system call to the local kernel. Condor users link their applications with an enhanced standard C library via the `condor_compile` command. This library does not duplicate any code in the standard C library; instead, it augments certain system call stubs (such as the ones that handle file I/O) into remote system call stubs. The remote system call stubs package the system call number and arguments into a message that is sent over the network to a `condor_shadow` process that runs on the submit machine. Whenever Condor starts a Standard Universe job, it also starts a corresponding shadow process on the initiating host where the user originally submitted the job (see Figure 14.3). This shadow process acts as an agent for the remotely executing program in performing system calls. The shadow then executes the system call on behalf of the remotely running job in the normal way. The shadow packages up the results of the system call in a message and sends it back to the remote system call stub in the Condor library on the remote machine. The remote system call stub returns its result to the calling procedure, which is unaware that the call was done remotely rather than locally. In this fashion, calls in the user's program to `open()`, `read()`, `write()`, `close()`, and all other file I/O calls transparently take place on the machine that submitted the job instead of on the remote execution machine.

Relinking and Submitting for the Standard Universe. To convert a program into a Standard Universe job, use the `condor_compile` command to relink with the Condor libraries. Place `condor_compile` in front of your usual link command. You do not need to modify the program's source code, but you do need access to its unlinked object files. A commercial program that is packaged as a single executable file cannot be converted into a Standard Universe job.

For example, if you normally link your job by executing

```
% cc main.o tools.o -o program
```

Image Not Available

Figure 14.3
Remote System calls in the Standard Universe.

You can relink your job for Condor with

```
% condor_compile cc main.o tools.o -o program
```

After you have relinked your job, you can submit it. A submit description file for the Standard Universe is similar to one for the Vanilla Universe. However, several additional submit directives are available to perform activities such as on-the-fly compression of data files. Here is an example:

```
# Submit 100 runs of my-program to the Standard Universe
universe = standard
executable = my-program.exe
# Each run should take place in a seperate subdirectory: run0, run1, ...
initialdir = run$(Process)
# Ask the Condor remote syscall layer to automatically compress
# on-the-fly any writes done by my-program.exe to file data.output
compress_files = data.output
queue 100
```

Standard Universe Limitations. Condor performs its process checkpoint and migration routines strictly in user mode; there are no kernel drivers with Condor. Because Condor is not operating at the kernel level, there are limitations on what process state it is able to checkpoint. As a result, the following restrictions are imposed upon Standard Universe jobs:

1. Multiprocess jobs are not allowed. This includes system calls such as `fork()`, `exec()`, and `system()`.

2. Interprocess communication is not allowed. This includes pipes, semaphores, and shared memory.

3. Network communication must be brief. A job *may* make network connections using system calls such as `socket()`, but a network connection left open for long periods will delay checkpoints and migration.

4. Multiple kernel-level threads are not allowed. However, multiple user-level threads (green threads) *are* allowed.

5. All files should be accessed read-only or write-only. A file that is both read and written to can cause trouble if a job must be rolled back to an old checkpoint image.

6. On Linux, your job must be statically linked. Dynamic linking is allowed in the Standard Universe on some other platforms supported by Condor, and perhaps this restriction on Linux will be removed in a future Condor release.

14.2.5 Giving Your Job Access to Its Data Files

Once your job starts on a machine in your pool, how does it access its data files? Condor provides several choices.

If the job is a Standard Universe job, then Condor solves the problem of data access automatically using the Remote System call mechanism described above. Whenever the job tries to open, read, or write to a file, the I/O will actually take place on the submit machine, whether or not a shared file system is in place.

Condor can use a shared file system, if one is available and permanently mounted across the machines in the pool. This is usually the case in a Beowulf cluster. But what if your Condor pool includes nondedicated (desktop) machines as well? You could specify a `Requirements` expression in your submit description file to require that jobs run only on machines that actually do have access to a common, shared file system. Or, you could request in the submit description file that Condor transfer your job's data files using the Condor File Transfer mechanism.

When Condor finds a machine willing to execute your job, it can create a temporary subdirectory for your job on the execute machine. The Condor File Transfer mechanism will then send via TCP the job executable(s) and input files from the submitting machine into this temporary directory on the execute machine. After the input files have been transferred, the execute machine will start running the job with the temporary directory as the job's current working directory. When the job completes or is kicked off, Condor File Transfer will automatically send back to the submit machine any output files created or modified by the job. After the files have been sent back successfully, the temporary working directory on the execute machine is deleted.

Condor's File Transfer mechanism has several features to ensure data integrity in a nondedicated environment. For instance, transfers of multiple files are performed atomically.

Condor File Transfer behavior is specified at job submission time using the submit description file and `condor_submit`. Along with all the other job submit description parameters, you can use the following File Transfer commands in the submit description file:

transfer_input_files = < file1, file2, file... >: Use this parameter to list all the files that should be transferred into the working directory for the job before the job is started.

transfer_output_files = < file1, file2, file... >: Use this parameter to explicitly list which output files to transfer back from the temporary working directory on the execute machine to the submit machine. Most of the time, however, there is no need to use this parameter. If `transfer_output_files` is not specified, Condor will automatically transfer in the job's temporary working directory all files that have been modified or created by the job.

transfer_files = <ONEXIT | ALWAYS | NEVER>: If `transfer_files` is set to `ONEXIT`, Condor will transfer the job's output files back to the submitting machine only when the job completes (exits). Specifying `ALWAYS` tells Condor to transfer back the output files when the job completes *or* when Condor kicks off the job (preempts) from a machine prior to job completion. The `ALWAYS` option is specifically intended for fault-tolerant jobs that periodocially write out their state to disk and can restart where they left off. Any output files transferred back to the submit machine when Condor preempts a job will automatically be sent back out again as input files when the job restarts.

14.2.6 The DAGMan Scheduler

The DAGMan scheduler within Condor allows the specification of dependencies between a set of programs. A directed acyclic graph (DAG) can be used to represent a set of programs where the input, output, or execution of one or more programs is dependent on one or more other programs. The programs are nodes (vertices) in the graph, and the edges (arcs) identify the dependencies. Each program within the DAG becomes a job submitted to Condor. The DAGMan scheduler enforces the dependencies of the DAG.

An input file to DAGMan identifies the nodes of the graph, as well as how to submit each job (node) to Condor. It also specifies the graph's dependencies and describes any extra processing that is involved with the nodes of the graph and must take place just before or just after the job is run.

A simple diamond-shaped DAG with four nodes is given in Figure 14.4.

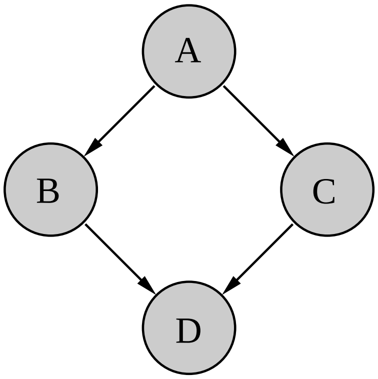

Figure 14.4
A directed acyclic graph with four nodes.

A simple input file to DAGMan for this diamond-shaped DAG may be

```
# file name: diamond.dag
Job  A  A.condor
Job  B  B.condor
Job  C  C.condor
Job  D  D.condor
PARENT A CHILD B C
PARENT B C CHILD D
```

The four nodes are named `A`, `B`, `C`, and `D`. Lines beginning with the keyword `Job` identify each node by giving it a name, and they also specify a file to be used as a submit description file for submission as a Condor job. Lines with the keyword `PARENT` identify the dependencies of the graph. Just like regular Condor submit description files, lines with a leading pound character (#) are comments.

The DAGMan scheduler uses the graph to order the submission of jobs to Condor. The submission of a child node will not take place until the parent node has successfully completed. No ordering of siblings is imposed by the graph, and therefore DAGMan does not impose an ordering when submitting the jobs to Condor. For the diamond-shaped example, nodes `B` and `C` will be submitted to Condor in parallel.

Each job in the example graph uses a different submit description file. An example submit description file for job `A` may be

```
# file name: A.condor
executable   = nodeA.exe
output       = A.out
error        = A.err
log          = diamond.log
universe     = vanilla
queue
```

An important restriction for submit description files of a DAG is that each node of the graph use the same log file. DAGMan uses the log file in enforcing the graph's dependencies.

The graph for execution under Condor is submitted by using the Condor tool `condor_submit_dag`. For the diamond-shaped example, submission would use the command

```
condor_submit_dag diamond.dag
```

14.3 Condor Architecture

A Condor pool comprises a single machine that 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. The central manager periodically assesses the current state of the pool and tries to match pending requests with the appropriate resources.

14.3.1 The Condor Daemons

In this subsection we describe all the daemons (background server processes) in Condor and the role each plays in the system.

`condor_master`: This daemon's role is to simplify system administration. It is responsible for keeping the rest of the Condor daemons running on each machine in a pool. The master spawns the other daemons and periodically checks the timestamps on the binaries of the daemons it is managing. If it finds new binaries, the master will restart the affected daemons. This allows Condor to be upgraded easily. In addition, if any other Condor daemon on the machine exits abnormally, the `condor_master` will send e-mail to the system administrator with information about the problem and then automatically restart the affected daemon. The `condor_master` also supports various administrative commands to start, stop, or reconfigure daemons remotely. The `condor_master` runs on every machine in your Condor pool.

`condor_startd`: This daemon represents a machine to the Condor pool. It advertises a machine ClassAd that contains attributes about the machine's capabilities and policies. Running the `startd` enables a machine to execute jobs. The `condor_startd` is responsible for enforcing the policy under which 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 spawns the remote Condor job on a given machine. It sets up the execution environment and monitors the job once it is running. The starter detects job completion, sends back status information to the submitting machine, and exits.

`condor_schedd`: This daemon represents jobs to the Condor pool. Any machine that allows users to submit jobs needs to have a `condor_schedd` running. Users submit jobs to the `condor_schedd`, where they are stored in the *job queue*. The various tools to view and manipulate the job queue (such as `condor_submit`, `condor_q`, or `condor_rm`) connect to the `condor_schedd` to do their work.

`condor_shadow`: This program runs on the machine where a job was submitted whenever that job is executing. The shadow serves requests for files to transfer, logs the job's progress, and reports statistics when the job completes. 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 to the `condor_shadow`. The shadow 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.

`condor_collector:` This daemon is responsible for collecting all the information about the status of a Condor pool. All other daemons 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 `condor_schedd`). The `condor_collector` can be thought of as a dynamic database of ClassAds. The `condor_status` command can be used to query the collector for specific information about various parts of Condor. The Condor daemons also query the collector for important information, such as what address to use for sending commands to a remote machine. The `condor_collector` runs on the machine designated as the central manager.

`condor_negotiator:` This daemon is responsible for all the matchmaking within the Condor system. The negotiator is also responsible for enforcing user priorities in the system.

14.3.2 The Condor Daemons in Action

Within a given Condor installation, one machine will serve as the pool's central manager. In addition to the `condor_master` daemon that runs on every machine in a Condor pool, the central manager runs the `condor_collector` and the `condor_negotiator` daemons. Any machine in the installation that should be capable of running jobs should run the `condor_startd`, and any machine that should maintain a job queue and therefore allow users on that machine to submit jobs should run a `condor_schedd`.

Condor allows any machine simultaneously to execute jobs and serve as a submission point by running both a `condor_startd` and a `condor_schedd`. Figure 14.5 displays a Condor pool in which every machine in the pool can both submit and run jobs, including the central manager.

The interface for adding a job to the Condor system is `condor_submit`, which reads a job description file, creates a job ClassAd, and gives that ClassAd to the `condor_schedd` managing the local job queue. This triggers a *negotiation cycle.* During a negotiation cycle, the `condor_negotiator` queries the `condor_collector` to discover all machines that are willing to perform work and all users with idle jobs. The `condor_negotiator` communicates *in user priority order* with each

Image Not Available

Figure 14.5
Daemon layout of an idle Condor pool.

`condor_schedd` that has idle jobs in its queue, and performs matchmaking to match jobs with machines such that both job and machine ClassAd requirements are satisfied and preferences (rank) are honored.

Once the `condor_negotiator` makes a match, the `condor_schedd` claims the corresponding machine and is allowed to make subsequent scheduling decisions about the order in which jobs run. This hierarchical, distributed scheduling architecture enhances Condor's scalability and flexibility.

When the `condor_schedd` starts a job, it spawns a `condor_shadow` process on the submit machine, and the `condor_startd` spawns a `condor_starter` process on the corresponding execute machine (see Figure 14.6). The shadow transfers the job ClassAd and any data files required to the starter, which spawns the user's application.

If the job is a Standard Universe job, the shadow will begin to service remote system calls originating from the user job, allowing the job to transparently access data files on the submitting host.

When the job completes or is aborted, the `condor_starter` removes every process spawned by the user job, and frees any temporary scratch disk space used by the job. This ensures that the execute machine is left in a clean state and that resources (such as processes or disk space) are not being leaked.

Image Not Available

Figure 14.6
Daemon layout when a job submitted from Machine 2 is running.

14.4 Installing Condor under Linux

The first step toward the installation of Condor is to download the software from the Condor Web site at `www.cs.wisc.edu/condor/downloads`. There is no cost to download or use Condor.

On the Web site you will find complete documentation and release notes for the different versions and platforms supported. You should take care to download the appropriate version of Condor for your platform (the operating system and processor architecture).

Before you begin the installation, there are several issues you need to consider and actions to perform.

Creation of User Condor. For both security and performance reasons, the Condor daemons should execute with root privileges. However, to avoid running as root except when absolutely necessary, the Condor daemons will run with the privileges of user condor on your system. In addition, the user condor simplifies installation, since files owned by the user condor will be created, and the home directory of the user condor can be used to specify file locations. For Linux clusters, we highly recommend that you create the user condor on all machines before installation begins.

Location. Administration of your pool is eased when the release directory (which includes all the binaries, libraries, and configuration files used by Condor) is placed on a shared file server. Note that one set of binaries is needed for each platform in your pool.

Administrator. Condor needs an e-mail address for an administrator. Should Condor need assistance, this is where e-mail will be sent.

Central Manager. The central manager of a Condor pool does matchmaking and collects information for the pool. Choose a central manager that has a good network connection and is likely to be online all the time (or at least rebooted quickly in the event of a failure).

Once you have decided the answers to these questions (and set up the condor user) you are ready to begin installation. The tool called `condor_install` is executed to begin the installation. The configuration tool will ask you a short series of questions, mostly related to the issues addressed above. Answer the questions appropriately for your site, and Condor will be installed.

On a large Linux cluster, you can speed the installation process by running `condor_install` once on your fileserver node and configuring your entire pool at the same time. If you use this configuration option, you will need to run only the `condor_init` script (which requires no input) on each of your compute nodes.

The default Condor installation will configure your pool to assume nondedicated resources. Section 14.5 discusses how to configure and customize your pool for a dedicated cluster.

After Condor is installed, you will want to customize a few security configuration right away. Condor implements security at the host (or machine) level. A set of configuration defaults set by the installation deal with access to the Condor pool by host. Given the distributed nature of the daemons that implement Condor, access to these daemons is naturally host based. Each daemon can be given the ability to allow or deny service (by host) within its configuration. Within the access levels available, *Read*, *Write*, *Administrator*, and *Config* are important to set correctly for each pool of machines.

Read: allows a machine to obtain information from Condor. Examples of information that may be read are the status of the pool and the contents of the job queue.

Write: allows a machine to provide information to Condor, such as submit a job or join the pool.

Administrator: allows a user on the machine to affect privileged operations such as changing a user's priority level or starting and stopping the Condor system from running.

Config: allows a user on the machine to change Condor's configuration settings remotely using the `condor_config_val` tool's *-set* and *-rset* options. This has very serious security implications, so we recommend that you not enable Config access to any hosts.

The defaults during installation give all machines read and write access. The central manager is also given administrator access. You will probably wish to change these defaults for your site. Read the Condor Administrator's Manual for details on network authorization in Condor and how to customize it for your wishes.

14.5 Configuring Condor

This section describes how to configure and customize Condor for your site. It discusses the configuration files used by Condor, describes how to configure the policy for starting and stopping jobs in your pool, and recommends settings for using Condor on a cluster.

A number of configuration files facilitate different levels of control over how Condor is configured on each machine in a pool. The top-level or global configuration file is shared by all machines in the pool. For ease of administration, this file should be located on a shared file system. In addition, each machine may have multiple local configuration files allowing the local settings to override the global settings. Hence, each machine may have different daemons running, different policies for when to start and stop Condor jobs, and so on.

All of Condor's configuration files should be owned and writable only by root. It is important to maintain strict control over these files because they contain security-sensitive settings.

14.5.1 Location of Condor's Configuration Files

Condor has a default set of locations it uses to try to find its top-level configuration file. The locations are checked in the following order:

1. The file specified in the `CONDOR_CONFIG` environment variable.

2. '/etc/condor/condor_config', if it exists.

3. If user condor exists on your system, the 'condor_config' file in this user's home directory.

If a Condor daemon or tool cannot find its global configuration file when it starts, it will print an error message and immediately exit. Once the global configuration file has been read by Condor, however, any other local configuration files can be specified with the LOCAL_CONFIG_FILE macro.

This macro can contain a single entry if you want only two levels of configuration (global and local). If you need a more complex division of configuration values (for example, if you have machines of different platforms in the same pool and desire separate files for platform-specific settings), LOCAL_CONFIG_FILE can contain a list of files.

Condor provides other macros to help you easily define the location of the local configuration files for each machine in your pool. Most of these are special macros that evaluate to different values depending on which host is reading the global configuration file:

• HOSTNAME: The hostname of the local host.
• FULL_HOSTNAME: The fully qualified hostname of the local host.
• TILDE: The home directory of the user condor on the local host.
• OPSYS: The operating system of the local host, such as "LINUX," "WINNT4" (for Windows NT), or "WINNT5" (for Windows 2000). This is primarily useful in heterogeneous clusters with multiple platforms.
• RELEASE_DIR: The directory where Condor is installed on each host. This macro is defined in the global configuration file and is set by Condor's installation program.

By default, the local configuration file is defined as

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

14.5.2 Recommended Configuration File Layout for a Cluster

Ease of administration is an important consideration in a cluster, particularly if you have a large number of nodes. To make Condor easy to configure, we highly recommend that you install all of your Condor configuration files, even the per-node local configuration files, on a shared file system. That way, you can easily make changes in one place.

You should use a subdirectory in your release directory for holding all of the local configuration files. By default, Condor's release directory contains an '`etc`' directory for this purpose.

You should create separate files for each node in your cluster, using the hostname as the first half of the filename, and ".local" as the end. For example, if your cluster nodes are named "n01", "n02" and so on, the files should be called '`n01.local`', '`n02.local`', and so on. These files should all be placed in your '`etc`' directory.

In your global configuration file, you should use the following setting to describe the location of your local configuration files:

```
LOCAL_CONFIG_FILE = $(RELEASE_DIR)/etc/$(HOSTNAME).local
```

The central manager of your pool needs special settings in its local configuration file. These attributes are set automatically by the Condor installation program. The rest of the local configuration files can be left empty at first.

Having your configuration files laid out in this way will help you more easily customize Condor's behavior on your cluster. We discuss other possible configuration scenarios at the end of this chapter.

Note: We recommend that you store all of your Condor configuration files under a version control system, such as CVS. While this is not required, it will help you keep track of the changes you make to your configuration, who made them, when they occurred, and why. In general, it is a good idea to store configuration files under a version control system, since none of the above concerns are specific to Condor.

14.5.3 Customizing Condor's Policy Expressions

Certain configuration expressions are used to control Condor's policy for executing, suspending, and evicting jobs. Their interaction can be somewhat complex. Defining an inappropriate policy impacts the throughput of your cluster and the happiness of its users. If you are interested in creating a specialized policy for your pool, we recommend that you read the Condor Administrator's Manual. Only a basic introduction follows.

All policy expressions are ClassAd expressions and are defined in Condor's configuration files. Policies are usually poolwide and are therefore defined in the global configuration file. If individual nodes in your pool require their own policy, however, the appropriate expressions can be placed in local configuration files.

The policy expressions are treated by the `condor_startd` as part of its machine ClassAd (along with all the attributes you can view with `condor_status -long`).

They are always evaluated against a job ClassAd, either by the `condor_negotiator` when trying to find a match or by the `condor_startd` when it is deciding what to do with the job that is currently running. Therefore, all policy expressions can reference attributes of a job, such as the memory usage or owner, in addition to attributes of the machine, such as keyboard idle time or CPU load.

Most policy expressions are ClassAd Boolean expressions, so they evaluate to TRUE, FALSE, or UNDEFINED. UNDEFINED occurs when an expression references a ClassAd attribute that is not found in either the machine's ClassAd or the ClassAd of the job under consideration. For some expressions, this is treated as a fatal error, so you should be sure to use the ClassAd meta-operators, described in Section 14.1.2 when referring to attributes which might not be present in all ClassAds.

An explanation of policy expressions requires an understanding of the different stages that a job can go through from initially executing until the job completes or is evicted from the machine. Each policy expression is then described in terms of the step in the progression that it controls.

The Lifespan of a Job Executing in Condor. When a job is submitted to Condor, the `condor_negotiator` performs matchmaking to find a suitable resource to use for the computation. This process involves satisfying both the job and the machine's requirements for each other. The machine can define the exact conditions under which it is willing to be considered available for running jobs. The job can define exactly what kind of machine it is willing to use.

Once a job has been matched with a given machine, there are four states the job can be in: running, suspended, graceful shutdown, and quick shutdown. As soon as the match is made, the job sets up its execution environment and begins running.

While it is executing, a job can be suspended (for example, because of other activity on the machine where it is running). Once it has been suspended, the job can resume execution or can move on to preemption or eviction.

All Condor jobs have two methods for preemption: graceful and quick. Standard Universe jobs are given a chance to produce a checkpoint with graceful preemption. For the other universes, graceful implies that the program is told to get off the system, but it is given time to clean up after itself. On all flavors of Unix, a SIGTERM is sent during graceful shutdown by default, although users can override this default when they submit their job. A quick shutdown involves rapidly killing all processes associated with a job, without giving them any time to execute their own cleanup procedures. The Condor system performs checks to ensure that processes are not left behind once a job is evicted from a given node.

Condor Policy Expressions. Various expressions are used to control the policy for starting, suspending, resuming, and preempting jobs.

`START`**:** when the `condor_startd` is willing to start executing a job.

`RANK`**:** how much the `condor_startd` prefers each type of job running on it. The `RANK` expression is a floating-point instead of a Boolean value. The `condor_startd` will preempt the job it is currently running if there is another job in the system that yields a higher value for this expression.

`WANT_SUSPEND`**:** controls whether the `condor_startd` should even consider suspending this job or not. In effect, it determines which expression, `SUSPEND` or `PREEMPT`, should be evaluated while the job is running. `WANT_SUSPEND` does not control when the job is actually suspended; for that purpose, you should use the `SUSPEND` expression.

`SUSPEND`**:** when the `condor_startd` should suspend the currently running job. If `WANT_SUSPEND` evaluates to TRUE, `SUSPEND` is periodically evaluated whenever a job is executing on a machine. If `SUSPEND` becomes TRUE, the job will be suspended.

`CONTINUE`**:** if and when the `condor_startd` should resume a suspended job. The `CONTINUE` expression is evaluated only while a job is suspended. If it evaluates to TRUE, the job will be resumed, and the `condor_startd` will go back to the Claimed/Busy state.

`PREEMPT`**:** when the `condor_startd` should preempt the currently running job. This expression is evaluated whenever a job has been suspended. If `WANT_SUSPEND` evaluates to FALSE, `PREEMPT` is checked while the job is executing.

`WANT_VACATE`**:** whether the job should be evicted gracefully or quickly if Condor is preempting a job (because the `PREEMPT` expression evaluates to TRUE). If `WANT_VACATE` is FALSE, the `condor_startd` will immediately kill the job and all of its child processes whenever it must evict the application. If `WANT_VACATE` is TRUE, the `condor_startd` performs a graceful shutdown, instead.

`KILL`**:** when the `condor_startd` should give up on a graceful preemption and move directly to the quick shutdown.

`PREEMPTION_REQUIREMENTS`**:** used by the `condor_negotiator` when it is performing matchmaking, not by the `condor_startd`. While trying to schedule jobs on resources in your pool, the `condor_negotiator` considers the priorities of the various users in the system (see Section 14.6.3 for more details). If a user with a better priority has jobs waiting in the queue and no resources are currently idle, the matchmaker will consider preempting another user's jobs and giving those resources to the user with the better priority. This process is known as *priority preemption.* The `PREEMPTION_REQUIREMENTS` expression must evaluate to TRUE for such a preemption to take place.

`PREEMPTION_RANK`: a floating-point value evaluated by the `condor_negotiator`. If the matchmaker decides it must preempt a job due to user priorities, the macro `PREEMPTION_RANK` determines which resource to preempt. Among the set of all resources that make the `PREEMPTION_REQUIREMENTS` expression evaluate to TRUE, the one with the highest value for `PREEMPTION_RANK` is evicted.

14.5.4 Customizing Condor's Other Configuration Settings

In addition to the policy expressions, you will need to modify other settings to customize Condor for your cluster.

`DAEMON_LIST`: the comma-separated list of daemons that should be spawned by the `condor_master`. As described in Section 14.3.1 discussing the architecture of Condor, each host in your pool can play different roles depending on which daemons are started on it. You define these roles using the `DAEMON_LIST` in the appropriate configuration files to enable or disable the various Condor daemons on each host.

`DedicatedScheduler`: the name of the dedicated scheduler for your cluster. This setting must have the form

```
DedicatedScheduler = "DedicatedScheduler@full.host.name.here"
```

14.6 Administration Tools

Condor has a rich set of tools for the administrator. Table 14.2 gives an overview of the Condor commands typically used solely by the system administrator. Of course, many of the "user-level" Condor tools summarized in Table 14.2 can be helpful for cluster administration as well. For instance, the `condor_status` tool can easily display the status for all nodes in the cluster, including dynamic information such as current load average and free virtual memory.

14.6.1 Remote Configuration and Control

All machines in a Condor pool can be remotely managed from a centralized location. Condor can be enabled, disabled, or restarted remotely using the `condor_on`,

Command	Description
`condor_checkpoint`	Checkpoint jobs running on the specified hosts
`condor_config_val`	Query or set a given Condor configuration variable
`condor_master_off`	Shut down Condor and the `condor_master`
`condor_off`	Shut down Condor daemons
`condor_on`	Start up Condor daemons
`condor_reconfig`	Reconfigure Condor daemons
`condor_restart`	Restart the `condor_master`
`condor_stats`	Display historical information about the Condor pool
`condor_userprio`	Display and manage user priorities
`condor_vacate`	Vacate jobs that are running on the specified hosts

Table 14.2
Commands reserved for the administrator.

`condor_off`, and `condor_restart` commands, respectively. Additionally, any aspect of Condor's configuration file on a node can be queried or changed remotely via the `condor_config_val` command. Of course, not everyone is allowed to change your Condor configuration remotely. Doing so requires proper authorization, which is set up at installation time (see Section 14.4).

Many aspects of Condor's configuration, including its scheduling policy, can be changed on the fly without requiring the pool to be shut down and restarted. This is accomplished by using the `condor_reconfig` command, which asks the Condor daemons on a specified host to reread the Condor configuration files and take appropriate action—on the fly if possible.

14.6.2 Accounting and Logging

Condor keeps many statistics about what is happening in the pool. Each daemon can be asked to keep a detailed log of its activities; Condor will automatically rotate these log files when they reach a maximum size as specified by the administrator.

In addition to the `condor_history` command, which allows users to view job ClassAds for jobs that have previously completed, the `condor_stats` tool can be used to query for historical usage statistics from a poolwide accounting database. This database contains information about how many jobs were being serviced for each user at regular intervals, as well as how many machines were busy. For instance, `condor_stats` could be asked to display the total number of jobs running at five-minute intervals for a specified user between January 15 and January 30.

The `condor_view` tool takes the raw information obtainable with `condor_stats` and converts it into HTML, complete with interactive charts. Figure 14.7 shows a sample display of the output from `condor_view` in a Web browser. The site administrator, using `condor_view`, can quickly put detailed, real-time usage statistics about the Condor pool onto a Web site.

Image Not Available

Figure 14.7
CondorView displaying machine usage.

14.6.3 User Priorities in Condor

The job queues in Condor are not strictly first-in, first-out. Instead, Condor implements *priority queuing*. Different users will get different-sized allocations of machines depending on their current user priority, regardless of how many jobs from a competing user are "ahead" of them in the queue. Condor can also be configured to perform *priority preemption* if desired. For instance, suppose user A is using all the nodes in a cluster, when suddenly a user with a superior priority submits jobs. With priority preemption enabled, Condor will preempt the jobs of the lower-priority user in order to immediately start the jobs submitted by the higher-priority user.

Starvation of the lower-priority users is prevented by a fair-share algorithm, which attempts to give all users the same amount of machine allocation time over a specified interval. In addition, the priority calculations in Condor are based on *ratios* instead of absolutes. For example, if Bill has a priority that is twice as good as that of Fred, Condor will not starve Fred by allocating all machines to Bill. Instead, Bill will get, on average, twice as many machines as will Fred because Bill's priority is twice as good.

The `condor_userprio` command can be used by the administrator to view or edit a user's priority. It can also be used to override Condor's default fair-share policy and explicitly assign users a better or worse priority in relation to other users.

14.7 Cluster Setup Scenarios

This section explores different scenarios for how to configure your cluster. Five scenarios are presented, along with a basic idea of what configuration settings you will need to modify or what steps you will need to take for each scenario:

1. A uniformly owned, dedicated compute cluster, with a single front-end node for submission, and support for MPI applications.

2. A cluster of multiprocessor nodes.

3. A cluster of distributively owned nodes. Each node prefers to run jobs submitted by its owner.

4. Desktop submission to the cluster.

5. Expanding the cluster to nondedicated (desktop) computing resources.

Most of these scenarios can be combined. Each scenario builds on the previous one to add further functionality to the basic cluster configuration.

14.7.1 Basic Configuration: Uniformly Owned Cluster

The most basic scenario involves a cluster where all resources are owned by a single entity and all compute nodes enforce the same policy for starting and stopping jobs. All compute nodes are dedicated, meaning that they will always start an idle job and they will never preempt or suspend until completion. There is a single front-end node for submitting jobs, and dedicated MPI jobs are enabled from this host.

In order to enable this basic policy, your global configuration file must contain these settings:

```
START = True
SUSPEND = False
CONTINUE = False
PREEMPT = False
KILL = False
WANT_SUSPEND = True
WANT_VACATE = True
RANK = Scheduler =?= $(DedicatedScheduler)
DAEMON_LIST = MASTER, STARTD
```

The final entry listed here specifies that the default role for nodes in your pool is execute-only. The `DAEMON_LIST` on your front-end node must also enable the `condor_schedd`. This front-end node's local configuration file will be

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

14.7.2 Using Multiprocessor Compute Nodes

If any node in your Condor pool is a symmetric multiprocessor machine, Condor will represent that node as multiple virtual machines (VMs), one for each CPU. By default, each VM will have a single CPU and an even share of all shared system resources, such as RAM and swap space. If this behavior satisfies your needs, you do not need to make any configuration changes for SMP nodes to work properly with Condor.

Some sites might want different behavior of their SMP nodes. For example, assume your cluster was composed of dual-processor machines with 1 gigabyte of RAM, and one of your users was submitting jobs with a memory footprint of 700 megabytes. With the default setting, all VMs in your pool would only have 500 megabytes of RAM, and your user's jobs would never run. In this case, you would want to unevenly divide RAM between the two CPUs, to give half of your VMs 750 megabytes of RAM. The other half of the VMs would be left with 250 megabytes of RAM.

There is more than one way to divide shared resources on an SMP machine with Condor, all of which are discussed in detail in the Condor Administrator's Manual. The most basic method is as follows. To divide shared resources on an SMP unevenly, you must define different *virtual machine types* and tell the condor_startd how many virtual machines of each type to advertise. The simplest method to define a virtual machine type is to specify what fraction of all shared resources each type should receive.

For example, if you wanted to divide a two-node machine where one CPU received one-quarter of the shared resources, and the other CPU received the other three-quarters, you would use the following settings:

```
VIRTUAL_MACHINE_TYPE_1 = 1/4
VIRTUAL_MACHINE_TYPE_2 = 3/4
NUM_VIRTUAL_MACHINES_TYPE_1 = 1
NUM_VIRTUAL_MACHINES_TYPE_2 = 1
```

If you want to divide certain resources unevenly but split the rest evenly, you can specify separate fractions for each shared resource. This is described in detail in the Condor Administrator's Manual.

14.7.3 Scheduling a Distributively Owned Cluster

Many clusters are owned by more than one entity. Two or more smaller groups might pool their resources to buy a single, larger cluster. In these situations, the group that paid for a portion of the nodes should get priority to run on those nodes.

Each resource in a Condor pool can define its own RANK expression, which specifies the kinds of jobs it would prefer to execute. If a cluster is owned by multiple entities, you can divide the cluster's nodes up into groups, based on ownership. Each node would set Rank such that jobs coming from the group that owned it would have the highest priority.

Assume there is a 60-node compute cluster at a university, shared by three departments: astronomy, math, and physics. Each department contributed the funds for 20 nodes. Each group of 20 nodes would define its own Rank expression. The astronomy department's settings, for example, would be

```
Rank = Department == "Astronomy"
```

The users from each department would also add a Department attribute to all of their job ClassAds. The administrators could configure Condor to add this attribute automatically to all job ads from each site (see the Condor Administrator's Manual for details).

If the entire cluster was idle and a physics user submitted 40 jobs, she would see all 40 of her jobs start running. If, however, a user in math submitted 60 jobs and a user in astronomy submitted 20 jobs, 20 of the physicist's jobs would be preempted, and each group would get 20 machines out of the cluster.

If all of the astronomy department's jobs completed, the astronomy nodes would go back to serving math and physics jobs. The astronomy nodes would continue to run math or physics jobs until either some astronomy jobs were submitted, or all the jobs in the system completed.

14.7.4 Submitting to the Cluster from Desktop Workstations

Most organizations that install a compute cluster have other workstations at their site. It is usually desirable to allow these machines to act as front-end nodes for the cluster, so users can submit their jobs from their own machines and have the applications execute on the cluster. Even if there is no shared file system between the cluster and the rest of the computers, Condor's remote system calls and file transfer functionality can enable jobs to migrate between the two and still access their data (see Section 14.2.5 for details on accessing data files).

To enable a machine to submit into your cluster, run the Condor installation program and specify that you want to setup a *submit-only* node. This will set the `DAEMON_LIST` on the new node to be

```
DAEMON_LIST = MASTER, SCHEDD
```

The installation program will also create all the directories and files needed by Condor.

Note that you can have only one node configured as the dedicated scheduler for your pool. Do not attempt to add a second submit node for MPI jobs.

14.7.5 Expanding the Cluster to Nondedicated (Desktop) Computing Resources

One of the most powerful features in Condor is the ability to combine dedicated and opportunistic scheduling within a single system. *Opportunistic scheduling* involves placing jobs on nondedicated resources under the assumption that the resources might not be available for the entire duration of the jobs. Opportunistic scheduling is used for all jobs in Condor with the exception of dedicated MPI applications.

If your site has a combination of jobs and uses applications other than MPI, you should strongly consider adding all of your computing resources, even desktop workstations, to your Condor pool. With checkpointing and process migration, suspend and resume capabilities, opportunistic scheduling and matchmaking, Condor can harness the idle CPU cycles of any machine and put them to good use.

To add other computing resources to your pool, run the Condor installation program and specify that you want to configure a node that can both submit and execute jobs. The default installation sets up a node with a policy for starting, suspending, and preempting jobs based on the activity of the machine (for example, keyboard idle time and CPU load). These nodes will not run dedicated MPI jobs, but they will run jobs from any other universe, including PVM.

14.8 Conclusion

Condor is a powerful tool for scheduling jobs across platforms, both within and beyond the boundaries of your Beowulf clusters. Through its unique combination of both dedicated and opportunistic scheduling, Condor provides a unified framework for high-throughput computing.

15 Maui Scheduler: A Multifunction Cluster Scheduler

David B. Jackson

In this chapter we describe the Maui scheduler, a job-scheduling component that can interact with a number of different resource managers.

Like virtually every major development project, Maui grew out of a pressing need. In Maui's case, various computing centers including the Maui High-Performance Computing Center, Pacific Northwest National Laboratory, San Diego Supercomputer Center, and Argonne National Laboratory were investing huge sums of money in new, top-of-the-line hardware, only to be frustrated by the inability to use these new resources in an efficient or controlled manner. While existing resource management systems allowed the basic ability to submit and run jobs, they did not empower the site to maximize the use of the cluster. Sites could not *translate* local mission policies into scheduling behavior, and the scheduling decisions that were made were often quite suboptimal. Worse, the resulting system was often so complex that management, administrators, and users were unable to tell how well the system was running or what could be done to improve it.

Maui was designed to address these issues and has been developed and tested over the years at many leading-edge computing centers. It was built to enable sites to control, understand, and use their clusters effectively. Maui picks up where many scheduling systems leave off, providing a suite of advanced features in the areas of reservations, backfill, fairshare, job prioritization, quality of service, metascheduling, and more.

15.1 Overview

Maui is an *external* scheduler, meaning it does not include a resource manager but rather extends the capabilities of the existing resource manager. Maui uses the native scheduling APIs of OpenPBS, PBSPro and Loadleveler to obtain system information and direct cluster scheduling activities. While the underlying resource manager continues to maintain responsibility for managing nodes and tracking jobs, Maui controls the decisions of when, where, and how jobs will run.

System administrators control Maui via a master config file, `maui.cfg`, and text or Web-based administrator commands. On the other hand, end users are not required to learn any new commands or job submission language, and need not even know that Maui has been installed. While Maui provides numerous commands to provide users with additional job information and control, these commands are optional and may be introduced to the users as needed.

15.2 Installation and Initial Configuration

The Maui scheduler is available in many of the most popular cluster-building toolkits, including *Rocks* and *OSCAR*. For the most recent version of Maui, you can download the code from the Maui home page at `supercluster.org/maui`. This site also contains online documentation, FAQs, links to the Maui users mailing list, and other standard open source utilities. To build the code once it has been downloaded, you need simply to issue the standard `configure`, `make`, and `make install`.

15.2.1 Basic Configuration

The `configure` script will prompt you for some basic information regarding the `install` directory and desired resource manager type. It then creates the Maui home directory, builds executables in the `bin` subdirectory, and copies these to the `install` directory. Finally, the script creates an initial `maui.cfg` file using templates located in the `samples` subdirectory and user-supplied information. This file is a *flat text* config file used for virtually all scheduler configuration and contains a number of parameters that should be verified, particularly, `SERVERHOST`, `SERVERMODE`, and `ADMIN1`. Initially, these should be set to the name of the host where Maui will run, `NORMAL`, and the user name of the Maui administrator, respectively. At any time when Maui is running, the `schedctl` command can be used with the '`-l`' flag to *list* the value of any parameter whether explicitly set or not, while the '`-m`' flag can be used to dynamically *modify* parameter values. The online `parameters` documentation provides further details about these and all other Maui parameters.

15.2.2 Simulation and Testing

With the initial configuration complete, the next step is testing the scheduler to become familiar with its capabilities and to verify basic functionality. Maui can be run in a completely *safe* manner by setting `SERVERMODE` to `TEST`. In *test* mode, Maui contacts the resource manager to obtain up-to-date configuration, node, and job information; however, in this mode, interfaces to start or modify these jobs are disabled. To start Maui, you must make the parameter changes and issue the command `maui`. You may also use commands such as `showq`, `diagnose`, and `checknode` to verify proper scheduler-resource manager communication and scheduler functionality. Full details on the suite of Maui commands are available online or in documentation included with your distribution.

15.2.3 Production Scheduling

Once you've taken the scheduler for a test drive and have verified its proper behavior, you can run Maui *live* by disabling the default scheduler and changing the `SERVERMODE` parameter to `NORMAL`. Information on disabling the default resource manager scheduler is provided in the resource manager's documentation and in the online Maui migration guides located at `supercluster.org/documentation/maui`. These changes will allow Maui to start, modify, and cancel jobs according to the specified scheduling policies.

Out of the box, Maui essentially duplicates the behavior of a vanilla cluster scheduler, providing first-in, first-out scheduling with backfill enabled. The parameters documentation explains in detail each of the parameters needed to enable advanced scheduling features. In most cases, each site will require only a small subset of the available parameters to meet local needs.

15.3 Advanced Configuration

With the initial configuration and testing completed, you can now configure Maui to end your administration pilgrimage and reach the long-sought cluster mecca—running the right jobs at the right time, in the right way, at the right place. To this end, Maui can be thought of as an integrated scheduling toolkit providing a number of capabilities that may be used individually or together to obtain the desired system behavior. These include

- job prioritization,
- node allocation policies,
- throttling policies,
- fairshare,
- reservations,
- allocation management,
- quality of service,
- backfill,
- node sets, and
- preemption policies.

Each of these is described below. While this coverage will be adequate to introduce and initially configure these capabilities, you should consult the online Maui Administrators Manual for full details. We reiterate that while Maui possesses a wide range of features and associated parameters, most capabilities are disabled by default; thus, a site need configure only the features of interest.

15.3.1 Assigning Value: Job Prioritization and Node Allocation

In general, prioritization is the process of determining which of many options best fulfills overall goals. n the case of scheduling, a site will often have multiple, independent goals that may include maximizing system utilization, giving preference to users in specific projects, or making certain that no job sits in the queue for more than a given period of time. One approach to representing a multifaceted set of site goals is to assign weights to the various objectives so an overall value or priority can be associated with each potential scheduling decision. With the jobs prioritized, the scheduler can roughly fulfill site objectives by starting the jobs in priority order.

Maui was designed to allow component and subcomponent weights to be associated with many aspects of a job. To realize this fine-grained control, Maui uses a simple priority-weighting hierarchy where the contribution of a priority factor is calculated as `PRIORITY-FACTOR-VALUE * SUBFACTORWEIGHT * FACTORWEIGHT`. Component and subcomponent weights are listed in Table 15.1. Values for all weights may be set in the `maui.cfg` file by using the associated component-weight parameter specified as the name of the weight followed by the string `WEIGHT` (e.g., `SERVICEWEIGHT` or `PROCWEIGHT`).

By default, Maui runs jobs in order of actual submission, using the `QUEUETIME`. By using priority components, however, you can incorporate additional information, such as current level of service, service targets, resources requested, and historical usage. You can also limit the contribution of any component, by specifying a priority component *cap*, such as `RESOURCECAP`. A job's priority is equivalent to the sum of all enabled priority factors.

Each component or subcomponent may be used for different purposes. `WALLTIME` can be used to favor (or disfavor) jobs based on their duration; `ACCOUNT` can be used to favor jobs associated with a particular project; `QUEUETIME` can be used to favor those jobs that have been waiting the longest. By mixing and matching priority weights, sites generally obtain the desired job-start behavior. At any time, you can issue the `diagnose -p` command to determine the impact of the current priority-weight settings on idle jobs.

While most subcomponents are metric based (i.e., number of seconds queued or number of nodes requested), the credential subcomponents are based on priorities specified by the administrator. Maui allows you to use the `*CFG` parameters to rank

Component	Subcomponent
SERVICE (Level of Service)	QUEUETIME (Current queue time in minutes) XFACTOR (Current expansion factor) BYPASS (Number of times jobs were bypassed via backfill)
TARGET (Proximity to Service Target - Exponential)	TARGETQUEUETIME (Delta to queue-time target in minutes) TARGETXFACTOR (Delta to Xfactor target)
RESOURCE (Resources Requested)	PROC (Processors) MEM (Requested memory in MBytes) SWAP (Requested virtual memory in MBytes) DISK (Requested local disk in MBytes) NODE (Requested number of nodes) WALLTIME (Requested wall time in seconds) PS (Requested processor-seconds) PE (Requested processor-equivalents)
FS (Fairshare)	FSUSER (User fairshare percentage) FSGROUP (Group fairshare percentage) FSACCOUNT (Account fairshare percentage) FSCLASS (Class fairshare percentage) FSQOS (QoS fairshare percentage)
CRED (Credential)	USER (User priority) GROUP (Group priority) ACCOUNT (Account priority) CLASS (Class priority) QOS (QoS priority)

Table 15.1
Maui priority components.

jobs by individual job credentials. For example, to favor jobs submitted by users `bob` and `john` and members of the group `staff`, a site might specify the following:

```
USERCFG[bob]        PRIORITY=100
USERCFG[john]       PRIORITY=500
GROUPWEIGHT[staff]  PRIORITY=1000
USERWEIGHT          1
GROUPWEIGHT         1
CREDWEIGHT          1
```

Note that both component and subcomponent weights are specified to enable these credential priorities to take effect. Further details about the use of these component factors, as well as anecdotal usage information, are available in the Maui Administrators Manual.

Complementing the issue of job prioritization is that of node allocation. When the scheduler selects a job to run, it must also determine which resources to allocate to the job. Depending on the use of the cluster, you can specify different policies by using `NODEALLOCATIONPOLICY`. Legal parameter values include the following:

- `MINRESOURCE`: This algorithm selects the nodes with the minimum configured resources which still meet the requirements of the job. The algorithm leaves more richly endowed nodes available for other jobs that may specifically request these additional resources.
- `LASTAVAILABLE`: This algorithm is particularly useful when making reservations for backfill. It determines the earliest time a job can run and then selects the resources available at a time such that, whenever possible, currently idle resources are left unreserved and are thus available for backfilling.
- `NODEPRIORITY`: This policy allows a site to create its own node allocation prioritization scheme, taking into account issues such as installed software or other local node configurations.
- `CPULOAD`: This policy attempts to allocate the most lightly loaded nodes first.

15.3.2 Fairness: Throttling Policies and Fairshare

The next issue most often confronting sites is *fairness*. Fairness seems like a simple concept but can be terribly difficult to map onto a cluster. Should all users get to run the same number of jobs or use the same number of nodes? Do these usage constraints cover the present time only or a specified time frame? If historical information is used, what is the metric of consumption? What is the time frame? Does fair consumption necessarily mean equal consumption? How should resources be allocated if user X bought two-thirds of the nodes and user Y purchased the other third? Is fairness based on a static metric, or is it conditional on current resource demand?

While Maui is not able to address all these issues, it does provide some flexible tools that help with 90 percent of the battle. Specifically, these tools are *throttling policies* and *fairshare* used to control immediate and historical usage, respectively.

Throttling Policies. The term "throttling policies" is collectively applied to a set of policies that constrain instantaneous resource consumption. Maui supports limits on the number of processors, nodes, proc-seconds, jobs, and processor equivalents allowed at any given time. Limits may be applied on a per user, group, account, QoS, or queue basis via the `*CFG` set of parameters. For example, specifying `USERCFG[bob] MAXJOB=3 MAXPROC=32` will constrain user `bob` to running no more than 3 jobs and 32 total processors at any given time. Specifying `GROUPCFG[DEFAULT] MAXNODE=64` will limit each group to using no more than 64 nodes simultaneously unless overriding limits for a particular group are specified. `ACCOUNTCFG`, `QOSCFG`, and `CLASSCFG` round out the *CFG family of parameters providing a means to throttle instantaneous use on accounts, QoS's, and classes, respectively.

With each of the parameters, *hard* and *soft* limits can be used to apply a form of *demand*-sensitive limits. While hard limits cannot be violated under any conditions, soft limits may be violated if no other jobs can run. For example, specifying `USERCFG[DEFAULT] MAXNODE=16,24` will allow each user to cumulatively allocate up to 16 nodes while jobs from other users can use available resources. If no other jobs can use these resources, a user may run on up to 24 nodes simultaneously.

Throttling policies are effective in preventing cluster "hogging" by an individual user or group. They also provide a simple mechanism of fairness and cycle distribution. Such policies may lead to lower overall system utilization, however. For instance, resources might go unused if these policies prevent all queued jobs from running. When possible, throttling policies should be set to the highest feasible level, and the cycle distribution should be managed by tools such as fairshare, allocation management systems, and QoS-based prioritization.

Fairshare. A typical fairshare algorithm attempts to deliver a fair resource distribution over a given time frame. As noted earlier, however, this general statement leaves much to interpretation. In particular, how is the distribution to be measured, and what time frame should be used?

Maui provides the parameter `FSPOLICY` to allow each site to determine how resource distribution is to be measured, and the parameters `FSINTERVAL`, `FSDEPTH`, and `FSDECAY` to determine how historical usage information is to be weighted.

To control resource distribution, Maui uses fairshare targets that can be applied to users, groups, accounts, queues, and QoS mechanisms with both default and specific targets available. Each target may be one of four different types: *target*, *floor*, *ceiling*, or *cap*. In most cases, Maui adjusts job priorities to meet fairshare targets. With the standard target, Maui attempts to adjust priorities at all times in an attempt to meet the target. In the case of floors, Maui will increase job priority only to maintain *at least* the targeted usage. With ceilings, the converse occurs. Finally, with fairshare caps, job eligibility rather than job priority is adjusted to prevent jobs from running if the cap is exceeded during the specified fairshare interval.

The example below shows a possible fairshare configuration.

```
# maui.cfg
FSPOLICY   DEDICATEDPS
FSDEPTH    7
FSINTERVAL 24:00:00
FSDECAY    0.80

USERCFG[DEFAULT]  FSTARGET=10.0
USERCFG[john]     FSTARGET=25.0+
GROUPCFG[staff]   FSTARGET=20.0-
```

In this case, fairshare usage will track delivered system *processor seconds* over a seven-day period with a 0.8 decay factor. All users will have a fairshare *target* of 10 percent of these processor seconds—with the exception of `john`, who will have a *floor* of 25 percent. Also, the group `staff` will have a fairshare *ceiling* of 20 percent. At any time, you can examine the fairshare status of the system by using the `diagnose -f` command.

15.3.3 Managing Resource Access: Reservations, Allocation Managers, and Quality of Service

In managing any cluster system, half of the administrative effort involves configuring it to handle the *steady-state* situation. The other half occurs when a very important user has a special one-time request. Maui provides two features, advance reservations and QoS, to handle many types of such special requests.

Advance Reservations. Reservations allow a site to set aside a block of resources for various purposes such as cluster maintenance, special user projects, or benchmarking nodes. In order to create a reservation, a start and end time must be determined, as well as the resources to be reserved and a list of those who can access these resources. Reservations can be created dynamically by scheduler administrators using the `setres` command or managed directly by Maui via config file parameters.

For example, to reserve `nodeA` and `nodeB` for a four-hour maintenance at 2:30 P.M., you could issue the following command:

```
> setres -s 14:30 -d 4:00:00 'node[AB]'
```

A reservation request can specify allocation of particular resources or a given quantity of resources. The following reservation will allocate 20 processors to users `john` and `sam` starting on April 14 at 5:00 P.M.

```
> setres -u john:sam -s 17:00_04/14 TASKS==20
```

With no duration or end time specified, this reservation will default to an infinite length and will remain in place until removed by a scheduler administrator using the `releaseres` command.

Access to reservations is controlled by an access control list (ACL). Reservation access is based on job credentials, such as user or group, and job attributes, such as wall time requested. Reservation ACLs can include multiple access types and individuals. For example, a reservation might reserve resources for users A and B, jobs in class C, and jobs that request less than 30 minutes of wall time. Reservations may also overlap each other if desired, in which case access is granted only if the job meets the access policies of all active reservations.

At many sites, reservations are used on a permanent or periodic basis. In such cases, it is best to use *standing* reservations. Standing reservations allow a site to apply reservations as an ongoing part of cluster policies. The parameter `SRPERIOD` can be set to `DAY`, `WEEK`, or `INFINITE` to indicate the periodicity of the reservation, with additional parameters available to determine what time of the day or week the reservation should be enabled. For example, the following configuration will create a reservation named `development` that, during primetime hours, will set aside 16 nodes for exclusive use by jobs requiring less than 30 minutes.

```
SRPERIOD[development]     DAY
SRDAYS[development]       Mon Tue Wed Thu Fri
SRSTARTTIME[development] 8:00:00
SRENDTIME[development]    17:00:00
SRMAXTIME[development]    00:30:00
SRTASKCOUNT[development] 16
```

At times, a site may want to allow access to a set of resources only if there are no other resources available. Maui enables this conditional usage through reservation *affinity*. When specifying any reservation access list, each access value can be associated with positive, negative, or neutral affinity by using the '+', '-', or '=' characters. If nothing is specified, positive affinity is assumed. For example, consider the following reservation line:

```
SRUSERLIST[special]  bob john steve= bill-
```

With this specification, `bob` and `john`'s jobs receive the default positive affinity and are essentially *attracted* to the reservation. For these jobs, Maui will attempt to use resources in the `special` reservation first, before considering any other resources. Jobs belonging to `steve`, on the other hand, can use these resources but are not attracted to them. Finally, `bill`'s jobs will use resources in the `special` reservation only if no other resources are available. You can get detailed information about reservations by using the `showres` and `diagnose -r` commands.

Allocation Managers. Allocation management systems allow a site to control total resource access in real time. While interfaces to support other systems exist, the allocation management system most commonly used with the Maui scheduler is QBank (`http://www.emsl.pnl.gov:80/mscf/docs/qbank-2.9`), provided by Pacific Northwest National Laboratory. This system and others like it allow sites to provide distinct resource allocations much like the creation of a bank account. As jobs run, the resources used are translated into a charge and debited from the appropriate account. In the case of QBank, expiration dates may be associated with allocations, private and shared accounts maintained, per machine allocations created, and so forth.

Within Maui, the allocation manager interface is controlled through a set of `BANK*` parameters such as in the example below:

```
BANKTYPE              QBANK
BANKHOST              bank.univ.edu
BANKCHARGEPOLICY      DEBITSUCCESSFULWC
BANKDEFERJOBONFAILURE TRUE
BANKFALLBACKACCOUNT   freecycle
```

This configuration enables a connection to an allocation manager located on `bank.univ.edu` using the QBank interface. The unit of charge is configured to be *dedicated processor-seconds* and users will be charged only if their job completes successfully. If the job does not have adequate allocations in the specified account, Maui will attempt to redirect the job to use allocations in the `freecycle` account. In many cases, a *fallback* account is configured so as to be associated with lower priorities and/or additional limitations. If the job is not approved by the allocation manager, Maui will defer the job for a period of time and try it again later.

Quality of Service. Maui's QoS feature allows sites to control access to special functions, resources, and service levels. Each QoS consists of an access control list controlling which users, groups, accounts, and job queues can access the QoS privileges. Associated with each QoS are special service-related priority weights and service targets. Additionally, each QoS can be configured to span resource partitions, preempt other jobs, and the like.

Maui also enables a site to charge a premium rate for the use of some QoS services. For example, the following configuration will cause user `john`'s jobs to use QoS `hiprio` by default and allow members of the group `bio` to access it by request:

```
USERCFG[john] QLIST=hiprio:normal QDEF=hiprio
GROUPCFG[bio] QLIST=hiprio:medprio:development QDEF=medprio
QOSCFG[hiprio] PRIORITY=50 QTTARGET=30 FLAGS=PREEMPTOR:IGNMAXJOB \
     MAXPROC=150
```

Jobs using QoS `hiprio` receive the following privileges and constraints:

- A priority boost of 50 * `QOSWEIGHT` * `DIRECTWEIGHT`
- A queue-time target of 30 minutes
- The ability to preempt lower priority `PREEMPTEE` jobs
- The ability to ignore `MAXJOB` policy limits defined elsewhere
- A cumulative limit of 150 processors allocated to QoS `hiprio` jobs

A site may have dozens of QoS objects described and may allow users access to any number of these. Depending on the type of service desired, users may then choose the QoS that best meets their needs.

15.3.4 Optimizing Usage: Backfill, Node Sets, and Preemption

The Maui scheduler provides several features to optimize performance in terms of system utilization, job throughput, and average job turnaround time.

Backfill. Backfill is a now common method used to improve both system utilization and average job turnaround time by running jobs out of order. Backfill, simply put, enables the scheduler to run any job so long as it does not delay the start of jobs of higher priority. Generally, the algorithm prevents delay of high-priority jobs through some form of reservation. Backfill can be thought of as a process of filling in the resource *holes* left by the high priority jobs. Since holes are being filled, it makes sense that the jobs most commonly backfilled are the ones requiring the least time and/or resources. With backfill enabled, sites typically report system utilization improvements of 10 to 25% and a slightly lower average job queue time.

By default, backfill scheduling is enabled in Maui under control of the parameter `BACKFILLPOLICY`. While the default configuration generally is adequate, sites may want to adjust the job selection policy, the reservation policy, the depth of reservations, or other aspects of backfill scheduling. You should consult the online documentation for details about associated parameters.

Allocation Based on Node Set. While backfill improves the scheduler's performance, this is only half the battle. The efficiency of a cluster, in terms of actual work accomplished, is a function of both scheduling performance and individual job efficiency. In many clusters, job efficiency can vary from node to node as well as with the *node mix* allocated. Since most parallel jobs written in popular languages such as MPI or PVM do not internally load balance their workload, they run only as fast as the slowest node allocated. Consequently, these jobs run most effectively on homogeneous sets of nodes. While many clusters start out as homogeneous, however, they quickly evolve as new generations of compute nodes are integrated into the system. Research has shown that this integration, while improving scheduling performance due to increased scheduler selection, can actually decrease average job efficiency.

A feature called *node sets* allows jobs to request sets of common resources without specifying exactly what resources are required. Node set policy can be specified globally or on a per job basis and can be based on node processor speed, memory, network interfaces, or locally defined node attributes. In addition to forcing jobs onto homogeneous nodes, these policies may also be used to guide jobs to one or more types of nodes on which a particular job performs best, similar to job preferences available in other systems. For example, an I/O-intensive job may run best on a certain range of processor speeds, running slower on slower nodes while wasting cycles on faster nodes. A job may specify `ANYOF:PROCSPEED:450:500:650` to request nodes in the range of 450 to 650 MHz. Alternatively, if a simple procspeed-homogeneous node set is desired, `ONEOF:PROCSPEED` may be specified. On the other hand, a communication-sensitive job may request a network-based node set with the configuration `ONEOF:NETWORK:via:myrinet:ethernet`, in which case Maui will first attempt to locate adequate nodes where all nodes contain VIA network interfaces. If such a set cannot be found, Maui will look for sets of nodes containing the other specified network interfaces. In highly heterogeneous clusters, the use of node sets has been found to improve job throughput by 10 to 15 percent.

Preemption. Many sites possess workloads of varying importance. While it may be critical that some jobs obtain resources immediately, other jobs are less sensitive to turnaround time but have an insatiable hunger for compute cycles, consuming every available cycle for years on end. These latter jobs often have turnaround times on the order of weeks or months. The concept of *cycle stealing*, popularized by systems such as Condor, handles such situations well and enables systems to run low-priority preemptible jobs whenever something more pressing is not running. These other systems are often employed on compute farms of desktops where the jobs must vacate whenever interactive system use is detected.

Maui's QoS-based preemption system allows a dedicated, noninteractive cluster to be used in much the same way. Certain QoS objects may be marked with the flag `PREEMPTOR` and others with the flag `PREEMPTEE`. With this configuration, low-priority "preemptee" jobs can be started whenever idle resources are available. These jobs will be allowed to run until a "preemptor" job arrives, at which point the preemptee job will be checkpointed if possible and vacated. This strategy allows almost immediate resource access for the preemptor job. Using this approach, a cluster can maintain nearly 100 percent system utilization while still delivering excellent turnaround time to the jobs of greatest value.

Use of the preemption system need not be limited to controlling low-priority jobs. Other uses include optimistic scheduling and development job support.

15.3.5 Evaluating System Performance: Diagnostics, Profiling, Testing, and Simulation

High-performance computing clusters are complicated. First, such clusters have an immense array of attributes that affect overall system performance, including processor speed, memory, networks, I/O systems, enterprise services, and application and system software. Second, each of these attributes is evolving over time, as is the usage pattern of the system's users. Third, sites are presented with an equally immense array of buttons, knobs, and levers which they can push, pull, kick, and otherwise manipulate. How does one evaluate the success of a current configuration? And how does one establish a causal effect between pushing one of the many provided buttons and improved system performance when the system is constantly changing in multiple simultaneous dimensions?

To help alleviate this problem, Maui offers several useful features.

Diagnostics. Maui possesses many internal diagnostic functions that both locate problems and present system state information. For example, the *priority* diagnostic aggregates priority relevant information, presenting configuration settings and their impact on the current idle workload; administrators can see the contribution associated with each priority factor on a per job and systemwide average basis. The *node* diagnostic presents significant node-relevant information together with messages regarding any unexpected conditions. Other diagnostics are available for jobs, reservations, QoS, fairshare, priorities, fairness policies, users, groups, and accounts.

Profiling Current and Historical Usage. Maui maintains internal statistics and records detailed information about each job as it completes. The `showstats` command provides detailed usage information for users, groups, accounts, nodes, and the system as a whole. The `showgrid` command presents scheduler performance statistics in a job size/duration matrix to aid in analyzing the effectiveness of current policies.

The completed job statistics are maintained in a flat file located in the `stats` directory. These statistics are useful for two primary purposes: driving simulations (described later) and profiling actual system usage. The `profiler` command allows the processing of these historical scheduler statistics and generation of usage reports for specific time frames or for selected users, groups, accounts, or types of jobs.

Testing. To test new policies, you can run a `TEST` mode instance of Maui concurrently with the production scheduler. This allows a site to analyze the effects of the new policies on the scheduling behavior of the test instance, while safely running the production workload under tried and true policies. When running an instance of Maui in test mode, it is often best to create a second Maui directory with associated `log` and `stats` subdirectories. To run multiple, concurrent Maui instances, you should take the following into account:

- **Configuration file:** The test version of Maui should have its own `maui.cfg` file to allow specification of the `SERVERMODE` parameter and allow policy differences as needed by the test.
- **User interface port:** To avoid conflicts between different scheduler instances and client commands, the test version of the `maui.cfg` file should specify a unique parameter value for `SERVERPORT`.
- **Log and statistics files:** Both production and test runs will create and update log and statistics files. To avoid file conflicts, each instance of the scheduler should point to different files using the `LOGDIR` and `STATDIR` parameters.
- **Home directory:** When Maui was initially installed, the `configure` script prompted for a home directory where the default `maui.cfg` file could be found. To run multiple instances of Maui, you should override this default by using the `-c` command line flag or by specifying the environment variable `MAUIHOMEDIR`. The latter approach is most often used, with the variable set to the new home directory before starting the test version of the scheduler or running test version client commands.

Once the test version is started, all scheduler behavior will be identical to the production system with the exception that Maui's ability to start, cancel, or otherwise modify jobs is disabled. You can, however, observe Maui's behavior under the new set of policies and validate the scheduler either directly via client commands or indirectly by analyzing the Maui log files.

Simulation. Simulation allows a site to specify a workload and resource configuration trace file. These traces, specified via the `SIMWORKLOADTRACEFILE` and `SIMRESOURCETRACEFILE`, can accurately and reproducibly replicate the workload and resources recorded at the site. To run a simulation, an adjusted `maui.cfg` file is created with the policies of interest in place and the parameter `SERVERMODE` set to `SIMULATION`. Once started, Maui can be stepped through simulated time using the `schedctl` command. All Maui commands continue to function as before, allowing interactive querying of status, adjustment to parameters, or even submission or cancellation of jobs.

This feature enables sites to analyze the impact of different scheduling policies on their own workload and system configuration. The effects of new reservations or job prioritizations can be evaluated in a *zero-exposure* environment, allowing sites to determine ideal policies without experimenting on a production system. Sites can also evaluate the impact of additional or modified workloads or changes in available resources. What impact will removing a block of resources for maintenance have on average queue time? How much benefit will a new reservation dedicated exclusively to development jobs have on development job turnaround time? How much pain will it cause nondevelopment jobs? Using simulation makes it easier to obtaining answers to such questions.

This same simulation feature can be used to test a new algorithm against workload and resource traces from various supercomputing centers. Moreover, with the simulator, you can create and plug in modules to emulate the behavior of various job types on different hardware platforms, across bottlenecking networks, or under various data migration conditions.

The capabilities and use of simulation cannot be adequately covered in a chapter of this size. Further information is given in the Simulation section of the Maui Administrators Manual.

15.4 Steering Workload and Improving Quality of Information

A good scheduler can improve the use of a cluster significantly, but its effectiveness is limited by the scheduling environment in which it must work and the quality of information it receives. Often, a cluster is underutilized because users overestimate a job's resource requirements. Other times, inefficiencies crop up when users request job constraints in terms of job duration or processors required that are not easily packed onto the cluster. Maui provides tools to allow fine tuning of job resource requirement information and steering of cluster workload so as to allow maximum utilization of the system.

One such tool is a *feedback* interface, which allows a site to report detailed job usage statistics to users. This interface provides information about the resources requested and those actually used. Using the FEEDBACKPROGRAM parameter, local scripts can be executed that use this information to help users improve resource requirement estimates. For example, a site with nodes with various memory configurations may choose to create a script such as the following that automates the mailing of notices at job completion:

```
Job 1371 completed successfully}.  Note that it requested nodes
with 512 MBytes of RAM yet used  only 112 MBytes.  Had the job provided a
more accurate estimate, it would have, on average, started 02:27:16
earlier.
```

Such notices can be used to improve memory, disk, processor, and wall-time estimates. Another route that is often used is to set the allocation manager charge policy so that users are charged for requested resources rather than used resources.

The `showbf` command is designed to help tailor jobs that can run immediately. This command allows you to specify details about your desired job (such as user, group, queue, and memory requirements) and returns information regarding the quantity of available nodes and the duration of their availability.

A final area of user feedback is job scaling. Often, users will submit parallel jobs that scale only moderately scale, hoping that by requesting more processors, their job will run faster and provide results sooner. A job's completion time is simply the sum of its queue time plus its execution time. Users often fail to realize that a larger job may be more difficult to schedule, resulting in a longer queue time, and may run less efficiently, with a *sublinear* speedup. The increased queue-time delay, together with the limitations in execution time improvements, generally results in larger jobs having a greater average turnaround time than smaller jobs performing the same work. Maui commands such as `showgrid` can provide real-time job efficiency and average queue-time stats correlated to job size. The output of the `profiler` command can also be used to provide per user job efficiency and average queue time correlated by job size and can alert administrators and users to this problem.

15.5 Troubleshooting

When troubleshooting scheduling issues, you should start with Maui's diagnostic and informational commands. The `diagnose` command together with `checknode` and `checkjob` provides detailed state information about the scheduler, including its various facilities, nodes, and jobs. Additionally, each of these commands initiates an extensive internal sanity check in the realm of interest. Results of this check are reported in the form of `WARNING` messages appended to the normal command output. Use of these commands typically identifies or resolves 95 percent of all scheduling issues.

If you need further information, Maui writes out detailed logging information in the directory pointed to by the `LOGFILE` parameter (usually in `${MAUIHOME}/log/maui.log`). Using the `LOGLEVEL` and `LOGFACILITY` parameters, you can control the verbosity and focus of these logs. (Note, however, that these logs can become *very* verbose, so keeping the LOGLEVEL below 4 or so unless actually tracking problems is advised.) These logs contain a number of entries, including the following:

`INFO:` provides status information about normal scheduler operations.

`WARNING:` indicates that an unexpected condition was detected and handled.

`ALERT:` indicates that an unexpected condition occurred that could not be fully handled.

`ERROR:` indicates that problem was detected that prevents Maui from fully operating. This may be a problem with the cluster that is outside of Maui's control or may indicate corrupt internal state information.

`Function header:` indicates when a function is called and the parameters passed.

A simple `grep` through the log file will usually indicate whether any serious issues have been detected and is of significant value when obtaining support or locally diagnosing problems. If neither commands nor logs point to the source of the problem, you may consult the Maui users list (`mauiusers@supercluster.org`) or directly contact Supercluster support at `support@supercluster.org`.

15.6 Conclusions

This chapter has introduced some of the key Maui features currently available. With hundreds of sites now using and contributing to this open source project, Maui is evolving and improving faster than ever. To learn about the latest developments and to obtain more detailed information about the capabilities described above, see the Maui home page at `www.supercluster.org/maui`.

16 PBS: Portable Batch System

James Patton Jones

The Portable Batch System (PBS) is a flexible workload management and job scheduling system originally developed to manage aerospace computing resources at NASA. PBS has since become the leader in supercomputer workload management and the de facto standard job scheduler for Linux.

Today, growing enterprises often support hundreds of users running thousands of jobs across different types of machines in different geographical locations. In this distributed heterogeneous environment, it can be extremely difficult for administrators to collect detailed, accurate usage data or to set systemwide resource priorities. As a result, many computing resources are left underused, while others are overused. At the same time, users are confronted with an ever-expanding array of operating systems and platforms. Each year, scientists, engineers, designers, and analysts waste countless hours learning the nuances of different computing environments, rather than being able to focus on their core priorities. PBS addresses these problems for computing-intensive industries such as science, engineering, finance, and entertainment.

PBS allows you to unlock the potential in the valuable assets you already have, while at the same time reducing demands on system administrators, freeing them to focus on other activities. PBS can also help you effectively manage growth by tracking use levels across your systems and enhancing effective utilization of future purchases.

16.1 History of PBS

In the past, computers were used in a completely interactive manner. Background jobs were just processes with their input disconnected from the terminal. As the number of processors in computers continued to increase, however, the need to be able to schedule tasks based on available resources rose in importance. The advent of networked compute servers, smaller general systems, and workstations led to the requirement of a networked batch scheduling capability. The first such Unix-based system was the Network Queueing System (NQS) from NASA Ames Research Center in 1986. NQS quickly became the de facto standard for batch queuing.

Over time, distributed parallel systems began to emerge, and NQS was inadequate to handle the complex scheduling requirements presented by such systems. In addition, computer system managers wanted greater control over their compute resources, and users wanted a single interface to the systems. In the early 1990s

NASA needed a solution to this problem, but after finding nothing on the market that adequately addressed their needs, led an international effort to gather requirements for a next-generation resource management system. The requirements and functional specification were later adopted as an IEEE POSIX standard (1003.2d). Next, NASA funded the development of a new resource management system compliant with the standard. Thus the Portable Batch System was born.

PBS was quickly adopted on distributed parallel systems and replaced NQS on traditional supercomputers and server systems. Eventually the entire industry evolved toward distributed parallel systems, taking the form of both special-purpose and commodity clusters. Managers of such systems found that the capabilities of PBS mapped well onto cluster systems.

The latest chapter in the PBS story began when Veridian (the research and development contractor that developed PBS for NASA) released the Portable Batch System Professional Edition (PBS Pro), a complete workload management solution. The cluster administrator can now choose between two versions of PBS: OpenPBS, an older Open Source release of PBS; and PBS Pro, the new hardened and enhanced commercial version.

This chapter gives a technical overview of PBS and information on installing, using, and managing both versions of PBS. However, it is not possible to cover all the details of a software system the size and complexity of PBS in a single chapter. Therefore, we limit this discussion to the recommended configuration for Linux clusters, providing references to the various PBS documentation where additional, detailed information is available.

16.1.1 Acquiring PBS

While both OpenPBS and PBS Pro are bundled in a variety of cluster kits, the best sources for the most current release of either product are the official Veridian PBS Web sites: `www.OpenPBS.org` and `www.PBSpro.com`. Both sites offers downloads of the software and documentation, as well as FAQs, discussion lists, and current PBS news. Hardcopy documentation, support services, training and PBS Pro software licenses are available from the PBS Online Store, accessed through the PBS Pro Web site.

16.1.2 PBS Features

PBS Pro provides many features and benefits to the cluster administrator. A few of the more important features are the following:

Enterprisewide resource sharing provides transparent job scheduling on any PBS system by any authorized user. Jobs can be submitted from any client system, both local and remote, crossing domains where needed.

Multiple user interfaces provide a graphical user interface for submitting batch and interactive jobs; querying job, queue, and system status; and monitoring job progress. Also provided is a traditional command line interface.

Security and access control lists permit the administrator to allow or deny access to PBS systems on the basis of username, group, host, and/or network domain.

Job accounting offers detailed logs of system activities for charge-back or usage analysis per user, per group, per project, and per compute host.

Automatic file staging provides users with the ability to specify any files that need to be copied onto the execution host before the job runs and any that need to be copied off after the job completes. The job will be scheduled to run only after the required files have been successfully transferred.

Parallel job support works with parallel programming libraries such as MPI, PVM, and HPF. Applications can be scheduled to run within a single multiprocessor computer or across multiple systems.

System monitoring includes a graphical user interface for system monitoring. PBS displays node status, job placement, and resource utilization information for both standalone systems and clusters.

Job interdependency enables the user to define a wide range of interdependencies between jobs. Such dependencies include execution order, synchronization, and execution conditioned on the success or failure of another specific job (or set of jobs).

Computational Grid support provides an enabling technology for meta-computing and computational Grids, including support for the Globus Toolkit.

Comprehensive API includes a complete application programming interface for sites that wish to integrate PBS with other applications or to support unique job-scheduling requirements.

Automatic load-leveling provides numerous ways to distribute the workload across a cluster of machines, based on hardware configuration, resource availability, keyboard activity, and local scheduling policy.

Distributed clustering allows customers to use physically distributed systems and clusters, even across wide area networks.

Common user environment offers users a common view of the job submission, job querying, system status, and job tracking over all systems.

Cross-system scheduling ensures that jobs do not have to be targeted to a specific computer system. Users may submit their job and have it run on the first available system that meets their resource requirements.

Job priority allows users the ability to specify the priority of their jobs; defaults can be provided at both the queue and system level.

User name mapping provides support for mapping user account names on one system to the appropriate name on remote server systems. This allows PBS to fully function in environments where users do not have a consistent username across all the resources they have access to.

Full configurability makes PBS easily tailored to meet the needs of different sites. Much of this flexibility is due to the unique design of the scheduler module, which permits complete customization.

Broad platform availability is achieved through support of Windows 2000 and every major version of Unix and Linux, from workstations and servers to supercomputers. New platforms are being supported with each new release.

System integration allows PBS to take advantage of vendor-specific enhancements on different systems (such as supporting `cpusets` on SGI systems and interfacing with the global resource manager on the Cray T3E).

For a comparison of the features available in the latest versions of OpenPBS and PBS Pro, visit the PBS Product Comparison Web page: `www.OpenPBS.org/product_comparison.html`.

16.1.3 PBS Architecture

PBS consists of two major component types: user-level commands and system daemons. A brief description of each is given here to help you make decisions during the installation process.

PBS supplies both command-line programs that are POSIX 1003.2d conforming and a graphical interface. These are used to submit, monitor, modify, and delete jobs. These *client commands* can be installed on any system type supported by PBS and do not require the local presence of any of the other components of PBS. There are three classifications of commands: user commands that any authorized user can use, operator commands, and manager (or administrator) commands. Operator and manager commands require specific access privileges. (See also the security sections of the PBS Administrator Guide.)

The *job server* daemon is the central focus for PBS. Within this document, it is generally referred to as the *Server* or by the execution name `pbs_server`. All commands and the other daemons communicate with the Server via an Internet Protocol (IP) network. The Server's main function is to provide the basic batch services such as receiving or creating a batch job, modifying the job, protecting the job against system crashes, and running the job. Typically, one Server manages a given set of resources.

The *job executor* is the daemon that actually places the job into execution. This daemon, `pbs_mom`, is informally called *MOM* because it is the mother of all executing jobs. (MOM is a reverse-engineered acronym that stands for Machine Oriented Mini-server.) MOM places a job into execution when it receives a copy of the job from a Server. MOM creates a new session as identical to a user login session as possible. For example, if the user's login shell is `csh`, then MOM creates a session in which `.login` is run as well as `.cshrc`. MOM also has the responsibility for returning the job's output to the user when directed to do so by the Server. One MOM daemon runs on each computer that will execute PBS jobs.

The *job scheduler* daemon, `pbs_sched`, implements the site's policy controlling when each job is run and on which resources. The Scheduler communicates with the various MOMs to query the state of system resources and with the Server to learn about the availability of jobs to execute. The interface to the Server is through the same API (discussed below) as used by the client commands. Note that the Scheduler interfaces with the Server with the same privilege as the PBS manager.

16.2 Using PBS

From the user's perspective, a workload mangement system enables you to make more efficient use of your time by allowing you to specify the tasks you need run on the cluster. The system takes care of running these tasks and returning the results to you. If the cluster is full, then it holds your tasks and runs them when the resources are available.

With PBS you create a *batch job* that you then submit to PBS. A batch job is a shell script containing the set of commands you want run on the cluster. It also contains directives that specify the resource requirements (such as memory or CPU time) that your job needs. Once you create your PBS job, you can reuse it, if you wish, or you can modify it for subsequent runs. Example job scripts are shown below.

PBS also provides a special kind of batch job called *interactive batch*. This job is treated just like a regular batch job (it is queued up and must wait for resources to become available before it can run). But once it is started, the user's terminal input and output are connected to the job in what appears to be an `rlogin` session. It appears that the user is logged into one of the nodes of the cluster, and the resources requested by the job are reserved for that job. Many users find this feature useful for debugging their applications or for computational steering.

PBS provides two user interfaces: a command-line interface (CLI) and a graphical user interface (GUI). You can use either to interact with PBS: both interfaces have the same functionality.

16.2.1 Creating a PBS Job

Previously we mentioned that a PBS job is simply a shell script containing resource requirements of the job and the command(s) to be executed. Here is what a sample PBS job might look like the following:

```
#!/bin/sh
#PBS -l walltime=1:00:00
#PBS -l mem=400mb
#PBS -l ncpus=4
#PBS -j oe

cd ${HOME}/PBS/test
mpirun -np 4 myprogram
```

This script would then be submitted to PBS using the *qsub* command.

Let us look at the script for a moment. The first line tells what shell to use to interpret the script. Lines 2–4 are resource directives, specifying arguments to the "resource list" ("-l") option of `qsub`. Note that all PBS directives begin with `#PBS`. These lines tell PBS what to do with your job. Any `qsub` option can also be placed inside the script by using a `#PBS` directive. However, PBS stops parsing directives with the first blank line encountered.

Returning to our example above, we see a request for 1 hour of wall-clock time, 400 MBytes of memory and 4 CPUs. The fifth line is a request for PBS to merge the stdout and stderr file streams of the job into a single file. The last two lines are the commands the user wants executed: change directory to a particular location, then execute an MPI program called '`myprogram`'.

This job script could have been created in one of two ways: using a text editor, or using the *xpbs* graphical interface (see below).

16.2.2 Submitting a PBS Job

The command used to submit a job to PBS is `qsub`. For example, say you created a file containing your PBS job called '`myscriptfile`'. The following example shows how to submit the job to PBS:

```
% qsub myscriptfile
12322.sol.pbspro.com
```

The second line in the example is the job identifier returned by the PBS server. This unique identifier can be used to act on this job in the future (before it completes running). The next section of this chapter discusses using this "job id" in various ways.

The qsub command has a number of options that can be specified either on the command-line or in the job script itself. Note that any command-line option will override the same option within the script file.

Table 16.1 lists the most commonly used options to qsub. See the PBS User Guide for the complete list and full description of the options.

Option	Purpose
-l list	List of resources needed by job
-q queue	Queue to submit job to
-N name	Name of job
-S shell	Shell to execute job script
-p priority	Priority value of job
-a datetime	Delay job under after datetime
-j oe	Join output and error files
-h	Place a hold on job

Table 16.1
PBS commands.

The "-l resource_list" option is used to specify the resources needed by the job. Table 16.2 lists all the resources available to jobs running on clusters.

16.2.3 Getting the Status of a PBS Job

Once the job has been submitted to PBS, you can use either the qstat or xpbs commands to check the job status. If you know the job identifier for your job, you can request the status explicitly. Note that unless you have multiple clusters, you need only specify the sequence number portion of the job identifier:

```
% qstat 12322
Job id        Name         User   Time Use S Queue
------------- ------------ ------ -------- - -----
12322.sol     myscriptfile jjones 00:06:39 R submit
```

Resource	Meaning
arch	System architecture needed by job
cput	CPU time required by all processes in job
file	Maximum single file disk space requirements
mem	Total amount of RAM memory required
ncpus	Number of CPUs (processors) required
nice	Requested "nice" (Unix priority) value
nodes	Number and/or type of nodes needed
pcput	Maximum per-process CPU time required
pmem	Maximum per-process memory required
wall time	Total wall-clock time needed
workingset	Total disk space requirements

Table 16.2
PBS resources.

If you run the `qstat` command without specifing a job identifier, then you will receive status on all jobs currently queued and running.

Often users wonder why their job is not running. You can query this information from PBS using the "-s" (status) option of `qstat`, for example,

```
% qstat 12323
Job id        Name         User   Time Use S Queue
------------- ------------ ------ -------- - -----
12323.sol     myscriptfile jjones 00:00:00 Q submit
  Requested number of CPUs not currently available.
```

A number of options to `qstat` change what information is displayed. The PBS User Guide gives the complete list.

16.2.4 PBS Command Summary

So far we have seen several of the PBS user commands. Table 16.3 is provided as a quick reference for all the PBS user commands. Details on each can be found in the PBS manual pages and the PBS User Guide.

16.2.5 Using the PBS Graphical User Interface

PBS provides two GUI interfaces: a TCL/TK-based GUI called **xpbs** and an optional Web-based GUI.

The GUI xpbs provides a user-friendly point-and-click interface to the PBS commands. To run `xpbs` as a regular, nonprivileged user, type

Command	Purpose
qalter	Alter job(s)
qdel	Delete job(s)
qhold	Hold job(s)
qmsg	Send a message to job(s)
qmove	Move job(s) to another queue
qrls	Release held job(s)
qrerun	Rerun job(s)
qselect	Select a specific subset of jobs
qsig	Send a signal to job(s)
qstat	Show status of job(s)
qsub	Submit job(s)
xpbs	Graphical Interface (GUI) to PBS commands

Table 16.3
PBS commands.

```
setenv DISPLAY your_workstation_name:0
xpbs
```

To run `xpbs` with the additional purpose of terminating PBS Servers, stopping and starting queues, or running or rerunning jobs, type

```
xpbs -admin
```

Note that you must be identified as a PBS operator or manager in order for the additional "-admin" functions to take effect.

The optional Web-based user interface provides access to all the functionality of `xpbs` via almost any Web browser. To access it, you simply type the URL of your PBS Server host into your browser. The layout and usage are similar to those of `xpbs`. For details, see The PBS User Guide.

16.2.6 PBS Application Programming Interface

Part of the PBS package is the PBS Interface Library, or IFL. This library provides a means of building new PBS clients. Any PBS service request can be invoked through calls to the interface library. Users may wish to build a PBS job that will check its status itself or submit new jobs, or they may wish to customize the job status display rather than use the `qstat` command. Administrators may use the interface library to build new control commands.

The IFL provides a user-callable function that corresponds to each PBS client command. There is (approximately) a one-to-one correlation between commands and PBS service requests. Additional routines are provided for network connection management. The user-callable routines are declared in the header file '`PBS_ifl.h`'. Users request service of a batch server by calling the appropriate library routine and passing it the required parameters. The parameters correspond to the options and operands on the commands. The user must ensure that the parameters are in the correct syntax. Each function will return zero upon success and a nonzero error code on failure. These error codes are available in the header file '`PBS_error.h`'. The library routine will accept the parameters and build the corresponding batch request. This request is then passed to the server communication routine. (The PBS API is fully documented in the PBS External Reference Specification.)

16.3 Installing PBS

PBS is able to support a wide range of configurations. It may be installed and used to control jobs on a single system or to load balance jobs on a number of systems. It may be used to allocate nodes of a cluster or parallel system to both serial and parallel jobs. It can also deal with a mix of these situations. However, given the topic of this book, we focus on the recommended configuration for clusters. The PBS Administrator Guide explains other configurations.

When PBS is installed on a cluster, a MOM daemon must be on each execution host, and the Server and Scheduler should be installed on one of the systems or on a front-end system.

For Linux clusters, PBS is packaged in the popular *RPM* format (Red Hat's Package Manager). (See the PBS Administrator Guide for installation instructions on other systems.) PBS RPM packages are provided as a single tar file containing

- the PBS Administrator Guide in both Postscript and PDF form,
- the PBS User Guide in both Postscript and PDF form (PBS Pro only),
- multiple RPM packages for different components of PBS (see below),
- a full set of Unix-style manual pages, and
- supporting text files: software license, README, release notes, and the like.

When the PBS tar file is extracted, a subtree of directories is created in which all these files are created. The name of the top-level directory of this subtree will reflect the release number and patch level of the version of PBS being installed. For example, the directory for PBS Pro 5.1 will be named '`PBSPro_5_1_0`'.

To install PBS Pro, change to the newly created directory, and run the installation program:

```
cd PBSPro_5_1_0
./INSTALL
```

The installation program will prompt you for the names of directories for the different parts of PBS and the type of installation (full, server-only, execution host only). Next, you will be prompted for your software license key(s). (See Section 16.1.1 if you do not already have your software license key.)

For OpenPBS, there are multiple RPMs corresponding to the different installation possibilities: full installation, execution host only, or client commands only. Select the correct RPM for your installation; then install it manually:

```
cd pbspro_v5.1
rpm -i RPMNAME...
```

Note that in OpenPBS, the RPMs will install into predetermined locations under '`/usr/pbs`' and '`/usr/spool/PBS`'.

16.4 Configuring PBS

Now that PBS has been installed, the Server and MOMs can be configured and the scheduling policy selected. Note that further configuration of may not be required since PBS Pro comes preconfigured, and the default configuration may completely meet your needs. However, you are advised to read this section to determine whether the defaults are indeed complete for you or whether any of the optional settings may apply.

16.4.1 Network Addresses and PBS

PBS makes use of fully qualified host names for identifying the jobs and their location. A PBS installation is known by the host name on which the Server is running. The name used by the daemons or used to authenticate messages is the canonical host name. This name is taken from the primary name field, `h_name`, in the structure returned by the library call `gethostbyaddr()`. According to the IETF RFCs, this name must be fully qualified and consistent for any IP address assigned to that host.

16.4.2 The Qmgr Command

The PBS manager command, `qmgr`, provides a command-line administrator interface. The command reads directives from standard input. The syntax of each directive is checked and the appropriate request sent to the Server(s). A `qmgr` directive takes one of the following forms:

```
command server [names] [attr OP value[,...]]
command queue  [names] [attr OP value[,...]]
command node   [names] [attr OP value[,...]]
```

where `command` is the command to perform on an object. The `qmgr` commands are listed in Table 16.4.

Command	Explanation
active	Set the active objects.
create	Create a new object, applies to queues and nodes.
delete	Destroy an existing object (queues or nodes).
set	Define or alter attribute values of the object.
unset	Clear the value of the attributes of the object.
list	List the current attributes and values of the object.
print	Print all the queue and server attributes.

Table 16.4
`qmgr` commands.

The `list` or `print` subcommands of `qmgr` can be executed by the general user. Creating or deleting a queue requires PBS Manager privilege. Setting or unsetting `server` or `queue` attributes requires PBS Operator or Manager privilege.

Here are several examples that illustrate using the `qmgr` command. These and other `qmgr` commands are fully explained below, along with the specific tasks they accomplish.

```
% qmgr
Qmgr: create node mars np=2,ntype=cluster
Qmgr: create node venus properties="inner,moonless"
Qmgr: set node mars properties = inner
Qmgr: set node mars properties += haslife
Qmgr: delete node mars
Qmgr: d n venus
```

Commands can be abbreviated to their minimum unambiguous form (as shown in the last line in the example above). A command is terminated by a new line character or a semicolon. Multiple commands may be entered on a single line. A command may extend across lines by marking the new line character with a backslash. Comments begin with a pound sign and continue to the end of the line. Comments and blank lines are ignored by `qmgr`. See the `qmgr` manual page for detailed usage and syntax description.

16.4.3 Nodes

Where jobs will be run is determined by an interaction between the Scheduler and the Server. This interaction is affected by the contents of the PBS '`nodes`' file and the system configuration onto which you are deploying PBS. Without this list of nodes, the Server will not establish a communication stream with the MOM(s), and MOM will be unable to report information about running jobs or to notify the Server when jobs complete. In a cluster configuration, distributing jobs across the various hosts is a matter of the Scheduler determining on which host to place a selected job.

Regardless of the type of execution nodes, each node must be defined to the Server in the PBS nodes file, (the default location of which is '`/usr/spool/PBS/server_-priv/nodes`'). This is a simple text file with the specification of a single node per line in the file. The format of each line in the file is

```
node_name[:ts] [attributes]
```

The node name is the network name of the node (host name), it does not have to be fully qualified (in fact, it is best kept as short as possible). The optional ":ts" appended to the name indicates that the node is a timeshared node.

Nodes can have attributes associated with them. Attributes come in three types: properties, `name=value` pairs, and `name.resource=value` pairs.

Zero or more properties may be specified. The property is nothing more than a string of alphanumeric characters (first character must be alphabetic) without meaning to PBS. Properties are used to group classes of nodes for allocation to a series of jobs.

Any legal node `name=value` pair may be specified in the node file in the same format as on a `qsub` directive: `attribute.resource=value`. Consider the following example:

```
NodeA resource_available.ncpus=3 max_running=1
```

The expression `np=N` may be used as shorthand for `resources_available.ncpus=N`, which can be added to declare the number of virtual processors (VPs) on the node. This syntax specifies a numeric string, for example, `np=4`. This expression will allow the node to be allocated up to N times to one job or more than one job. If `np=N` is not specified for a cluster node, it is assumed to have one VP.

You may edit the nodes list in one of two ways. If the server is not running, you may directly edit the nodes file with a text editor. If the server is running, you should use qmgr to edit the list of nodes.

Each item on the line must be separated by white space. The items may be listed in any order except that the host name must always be first. Comment lines may be included if the first nonwhite space character is the pound sign.

The following is an example of a possible nodes file for a cluster called "planets":

```
# The first set of nodes are cluster nodes.
# Note that the properties are provided to
# logically group certain nodes together.
# The last node is a timeshared node.
#
mercury    inner moonless
venus      inner moonless np=1
earth      inner np=1
mars       inner np=2
jupiter    outer np=18
saturn     outer np=16
uranus     outer np=14
neptune    outer np=12
pluto:ts
```

16.4.4 Creating or Adding Nodes

After pbs_server is started, the node list may be entered or altered via the qmgr command:

```
create node node_name [attribute=value]
```

where the attributes and their associated possible values are shown in Table 16.5.

Below are several examples of setting node attributes via qmgr:

```
% qmgr
Qmgr: create node mars np=2,ntype=cluster
Qmgr: create node venus properties="inner,moonless"
```

Once a node has been created, its attributes and/or properties can be modified by using the following qmgr syntax:

```
set node node_name [attribute[+|-]=value]
```

Attribute	Value
state	`free`, `down`, `offline`
properties	any alphanumeric string
ntype	`cluster`, `time-shared`
`resources_available.ncpus` (np)	number of virtual processors > 0
`resources_available`	list of resources available on node
`resources_assigned`	list of resources in use on node
`max_running`	maximum number of running jobs
`max_user_run`	maximum number of running jobs per user
`max_group_run`	maximum number of running jobs per group
queue	queue name (if any) associated with node
reservations	list of reservations pending on the node
comment	general comment

Table 16.5
PBS node syntax.

where attributes are the same as for `create,` for example,

```
% qmgr
Qmgr: set node mars properties=inner
Qmgr: set node mars properties+=haslife
```

Nodes can be deleted via `qmgr` as well, using the `delete node` syntax, as the following example shows:

```
% qmgr
Qmgr: delete node mars
Qmgr: delete node pluto
```

Note that the `busy` state is set by the execution daemon, `pbs_mom`, when a load-average threshold is reached on the node. See `max_load` in MOM's config file. The `job-exclusive` and `job-sharing` states are set when jobs are running on the node.

16.4.5 Default Configuration

Server management consist of configuring the Server and establishing queues and their attributes. The default configuration, shown below, sets the minimum server settings and some recommended settings for a typical PBS cluster.

```
% qmgr
Qmgr: print server
# Create queues and set their attributes
```

```
#
# Create and define queue workq
#
create queue workq
set queue workq queue_type = Execution
set queue workq enabled = True
set queue workq started = True
#
# Set Server attributes
#
set server scheduling = True
set server default_queue = workq
set server log_events = 511
set server mail_from = adm
set server query_other_jobs = True
set server scheduler_iteration = 600
```

16.4.6 Configuring MOM

The execution server daemons, MOMs, require much less configuration than does the Server. The installation process creates a basic MOM configuration file that contains the minimum entries necessary in order to run PBS jobs. This section describes the MOM configuration file and explains all the options available to customize the PBS installation to your site.

The behavior of MOM is controlled via a configuration file that is read upon daemon initialization (startup) and upon reinitialization (when `pbs_mom` receives a SIGHUP signal). The configuration file provides several types of runtime information to MOM: access control, static resource names and values, external resources provided by a program to be run on request via a shell escape, and values to pass to internal functions at initialization (and reinitialization). Each configuration entry is on a single line, with the component parts separated by white space. If the line starts with a pound sign, the line is considered to be a comment and is ignored.

A minimal MOM configuration file should contain the following:

```
$logevent 0x1ff
$clienthost server-hostname
```

The first entry, `$logevent`, specifies the level of message logging this daemon should perform. The second entry, `$clienthost`, identifies a host that is permitted to connect to this MOM. You should set the *server-hostname* variable to the name of the host on which you will be running the PBS Server (`pbs_server`). Advanced MOM configuration options are described in the PBS Administrator Guide.

16.4.7 Scheduler Configuration

Now that the Server and MOMs have been configured, we turn our attention to the PBS Scheduler. As mentioned previously, the Scheduler is responsible for implementing the local site policy regarding which jobs are run and on what resources. This section discusses the recommended configuration for a typical cluster. The full list of tunable Scheduler parameters and detailed explanation of each is provided in the PBS Administrator Guide.

The PBS Pro Scheduler provides a wide range of scheduling policies. It provides the ability to sort the jobs in several different ways, in addition to FIFO order. It also can sort on user and group priority. The queues are sorted by queue priority to determine the order in which they are to be considered. As distributed, the Scheduler is configured with the defaults shown in Table 16.6.

Option	Default Value
`round_robin`	`False`
`by_queue`	`True`
`strict_fifo`	`False`
`load_balancing`	`False`
`load_balancing_rr`	`False`
`fair_share`	`False`
`help_starving_jobs`	`True`
`backfill`	`True`
`backfill_prime`	`False`
`sort_queues`	`True`
`sort_by`	`shortest_job_first`
`smp_cluster_dist`	`pack`
`preemptive_sched`	`True`

Table 16.6
Default scheduling policy parameters.

Once the Server and Scheduler are configured and running, job scheduling can be initiated by setting the Server attribute scheduling to a value of true:

```
# qmgr -c "set server scheduling=true"
```

The value of scheduling is retained across Server terminations or starts. After the Server is configured, it may be placed into service.

16.5 Managing PBS

This section is intended for the PBS administrator: it discusses several important aspects of managing PBS on a day-to-day basis.

During the installation of PBS Pro, the file '`/etc/pbs.conf`' was created. This configuration file controls which daemons are to be running on the local system. Each node in a cluster should have its own '`/etc/pbs.conf`' file.

16.5.1 Starting PBS Daemons

The daemon processes (`pbs_server`, `pbs_sched`, and `pbs_mom`) must run with the real and effective `uid` of root. Typically, the daemons are started automatically by the system upon reboot. The boot-time start/stop script for PBS is '`/etc/init.d/pbs`'. This script reads the '`/etc/pbs.conf`' file to determine which daemons should be started.

The startup script can also be run by hand to get status on the PBS daemons, and to start/stop all the PBS daemons on a given host. The command line syntax for the startup script is

```
/etc/init.d/pbs [ status | stop | start ]
```

Alternatively, you can start the individual PBS daemons manually, as discussed in the following sections. Furthermore, you may wish to change the options specified to various daemons, as discussed below.

16.5.2 Monitoring PBS

The node monitoring GUI for PBS is `xpbsmon`. It is used for displaying graphically information about execution hosts in a PBS environment. Its view of a PBS environment consists of a list of sites where each site runs one or more Servers and each Server runs jobs on one or more execution hosts (nodes).

The system administrator needs to define the site's information in a global X resources file, '`PBS_LIB/xpbsmon/xpbsmonrc`', which is read by the GUI if a personal '`.xpbsmonrc`' file is missing. A default '`xpbsmonrc`' file is created during installation defining (under `*sitesInfo` resource) a default site name, the list of Servers that run on the site, the set of nodes (or execution hosts) where jobs on a particular Server run, and the list of queries that are communicated to each node's `pbs_mom`.

If node queries have been specified, the host where '`xpbsmon`' is running must have been given explicit permission by the `pbs_mom` daemon to post queries to it; this is done by including a `$restricted` entry in the MOM's config file.

16.5.3 Tracking PBS Jobs

Periodically you (or the user) will want track the status of a job. Or perhaps you want to view all the log file entries for a given job. Several tools allow you to track a job's progress, as Table 16.7 shows.

Command	Explanation
qstat	Shows status of jobs, queues, and servers
xpbs	Can alert user when job starts producing output
tracejob	Collates and sorts PBS log entries for specified job

Table 16.7
Job-tracking commands.

16.5.4 PBS Accounting Logs

The PBS Server daemon maintains an accounting log. The log name defaults to '`/usr/spool/PBS/server_priv/accounting/yyyymmdd`' where `yyyymmdd` is the date. The accounting log files may be placed elsewhere by specifying the `-A` option on the `pbs_server` command line. The option argument is the full (absolute) path name of the file to be used. If a null string is given, for example

```
# pbs_server -A ""
```

then the accounting log will not be opened, and no accounting records will be recorded.

The accounting file is changed according to the same rules as the log files. If the default file is used, named for the date, the file will be closed and a new one opened every day on the first event (write to the file) after midnight. With either the default file or a file named with the `-A` option, the Server will close the accounting log and reopen it upon the receipt of a SIGHUP signal. This strategy allows you to rename the old log and start recording anew on an empty file. For example, if the current date is December 1, the Server will be writing in the file '`20011201`'. The following actions will cause the current accounting file to be renamed '`dec1`' and the Server to close the file and starting writing a new '`20011201`'.

```
# mv 20011201 dec1
# kill -HUP (pbs_server's PID)
```

16.6 Troubleshooting

The following is a list of common problems and recommended solutions. Additional information is always available on the PBS Web sites.

16.6.1 Clients Unable to Contact Server

If a client command (such as `qstat` or `qmgr`) is unable to connect to a Server there are several possible errors to check. If the error return is 15034, *No server to connect to*, check (1) that there is indeed a Server running and (2) that the default Server information is set correctly. The client commands will attempt to connect to the Server specified on the command line if given or, if not given, the Server specified in the default server file, '`/usr/spool/PBS/default_server`'.

If the error return is 15007, *No permission*, check for (2) as above. Also check that the executable `pbs_iff` is located in the search path for the client and that it is `setuid` root. Additionally, try running `pbs_iff` by typing

```
pbs_iff server_host 15001
```

where `server_host` is the name of the host on which the Server is running and 15001 is the port to which the Server is listening (if started with a different port number, use that number instead of 15001). The executable `pbs_iff` should print out a string of garbage characters and exit with a status of 0. The garbage is the encrypted credential that would be used by the command to authenticate the client to the Server. If `pbs_iff` fails to print the garbage and/or exits with a nonzero status, either the Server is not running or it was installed with a different encryption system from that used for `pbs_iff`.

16.6.2 Nodes Down

The PBS Server determines the state of nodes (up or down), by communicating with MOM on the node. The state of nodes may be listed by two commands: `qmgr` and `pbsnodes`.

```
% qmgr
Qmgr: list node @active

% pbsnodes -a
Node jupiter
        state = down, state-unknown
        properties = sparc, mine
```

```
                ntype = cluster
```

A node in PBS may be marked `down` in one of two substates. For example, the state above of node "jupiter" shows that the Server has not had contact with MOM on that since the Server came up. Check to see whether a MOM is running on the node. If there is a MOM and if the MOM was just started, the Server may have attempted to poll her before she was up. The Server should see her during the next polling cycle in ten minutes. If the node is still marked `down, state-unknown` after ten minutes, either the node name specified in the Server's node file does not map to the real network hostname or there is a network problem between the Server's host and the node.

If the node is listed as

```
        % pbsnodes -a
        Node jupiter
                state = down
                properties = sparc, mine
                ntype = cluster
```

then the Server has been able to communicate with MOM on the node in the past, but she has not responded recently. The Server will send a `ping` PBS message to every free node each ping cycle (10 minutes). If a node does not acknowledge the ping before the next cycle, the Server will mark the node `down`.

16.6.3 Nondelivery of Output

If the output of a job cannot be delivered to the user, it is saved in a special directory '`/usr/spool/PBS/undelivered`' and mail is sent to the user. The typical causes of nondelivery are the following:

- The destination host is not trusted and the user does not have a .rhost file.
- An improper path was specified.
- A directory in the specified destination path is not writable.
- The user's `.cshrc` on the destination host generates output when executed.

The '`/usr/spool/PBS/spool`' directory on the execution host does not have the correct permissions. This directory must have mode 1777 (`drwxrwxrwxt`).

16.6.4 Job Cannot Be Executed

If a user receives a mail message containing a job identifier and the line "Job cannot be executed," the job was aborted by MOM when she tried to place it into execution. The complete reason can be found in one of two places: MOM's log file or the standard error file of the user's job.

If the second line of the message is "See Administrator for help," then MOM aborted the job before the job's files were set up. The reason will be noted in MOM's log. Typical reasons are a bad user/group account or a system error.

If the second line of the message is "See job standard error file," then MOM had already created the job's file, and additional messages were written to standard error.