Making Workstations a Friendly Environment for Batch Jobs

Miron Livny
Mike Litzkow

Computer Sciences Department
University of Wisconsin - Madison
{miron,mike}@cs.wisc.edu

1. Introduction

As time-sharing machines are replaced by powerful desktop computers and farms of workstations replace mainframes, more and more users turn to workstations when they need CPU cycles for their batch jobs. Unfortunately, they do not find workstations a very friendly environment for batch processing. Since these types of machines were originally designed as a single user environment, they lack most of the batch oriented features that are provided by almost any mainframe. In the past six years we have been working on the Condor batch system. The main objective of Condor has been to provide batch users with easy and reliable access to available cycles on UNIX workstations. A pool of more than 250 workstations is currently controlled by a Condor system in our department. Other Condor pools provide batch services at a wide range of academic and industrial settings. In the course of our work with UNIX workstations as providers of batch cycles we have composed a list of services that we think should be provided by the Operating System. In this paper we present the main services on this list and argue that they should be supported by all workstations.

2. Condor

Many organizations now own hundreds of powerful workstations which are connected by high capacity local area networks. It is common practice in such organizations to allocate each of these workstations to a single user who exercises full control over the workstation's resources. In such an environment we can find three types of users, *casual* users who seldom utilize the full capacity of their machines, *sporadic* users who for short periods of time fully utilize the capacity of the workstation they own, and *frustrated* users who for long periods of time have computing demands that are beyond the power of their workstations. Unlike the two other groups, the throughput of these frustrated users is limited by the power of their workstations. They often claim that their productivity could be significantly enhanced if they had access to the unutilized computing capacity of workstations owned by casual and sporadic users. Condor is a distributed batch system that was designed to meet the challenge posed by these users, namely to provide convenient access to unutilized workstations while preserving the rights of their owners.

2.1. Guiding Principles

Several principles have driven the design of Condor. First is the principal that workstation owners should always have the resources of the workstation they own at their disposal. This is to guarantee immediate response, which is the reason most people prefer a dedicated workstation over access to a time sharing system. The second principle is that access to remote capacity must be easy, and should approximate the local execution environment as closely as possible. Portability is the third principle behind the design of Condor. This is essential due to rapid developments in the workstations on which Condor operates. For example, the earliest version of Condor became operational in 1984 when it ran on a network of 20 VAX 11/750s. Eight years later we are running Condor on 250 workstations ranging over eight different hardware architectures and running a rainbow of UNIX variants; none of the original machines is still connected. Unfortunately, while portability has been one of the most important goals of Condor, and one of its biggest reasons for success, the price of that portability in terms of human effort and functional limitations has been significant. This is true even though every platform to which we have ported Condor runs some variation of "UNIX".

2.2. Mechanisms Used

Five mechanisms are basic to the operation of Condor. The first is a mechanism for determining when a workstation is not in use by its owner, and thus should become part of the pool of available machines. This is accomplished by measuring both the CPU load of the machine, and the time since the last keyboard or mouse activity. Second is a mechanism for "fair" allocation of these machines to users who have queued jobs. This task is handled by a centralized "machine manager". The manager allocates machines to waiting users on the basis of priority. The priority is calculated according to an algorithm that periodically increases the priority of those users who have been waiting for resources, and reduces the priority of those users who have received resources in the recent past. The purpose of the algorithm is to allow heavy users to consume very large amounts of CPU cycles, and at same time protect the response time for less frequent users.

Thirdly, Condor provides a remote execution mechanism which allows its users to run remotely on any machine in the pool the same programs that they had been used to running locally after only a relinking step. File I/O is redirected to the submitting machine, so that users don't need to worry about moving files to and from the machines where execution actually takes place. The fourth mechanism is responsible for stopping the execution of a Condor job upon the first user activity on the hosting machine. As soon as the keyboard or mouse becomes active, or the CPU load on the remote machine rises above a specified level, the Condor job is stopped. This provides automatic return of the use of the hosting workstation to its owner.

Finally Condor provides a transparent checkpointing mechanism which allows it to take a checkpoint of a running job, and migrate that job to another workstation when the machine it is currently running on becomes busy with non-Condor activity. This allows Condor to return workstations to their owners promptly, yet provide assurance to Condor users that their jobs will make progress, and eventually complete.

To meet the portability requirement, all these five mechanisms were implemented entirely outside the UNIX kernel. Over the years we have been able to remove from our implementation, almost all machine architecture dependent constructs. For example the current version of Condor does not include any assembler code whatsoever. Unfortunately variations in the operating systems have persistently worked against portability, and in some cases have forced us to implement the same mechanism in several different ways. Certainly the most difficult mechanism of Condor to make portable has been checkpointing which depends on the specific formats of the “a.out” and “core” files.

2.3. Experience

The current version of Condor has been in general use by researchers in our department since 1988. Our Condor pool contains some 250 heterogeneous workstations, and by the end of the first three years had served more than 250,000 jobs. Thousands of CPU days were consumed by jobs belonging to a very active community of Condor users during this period. Condor has been so successful at providing batch cycles that researchers have been able to make fundamental changes in the kinds of computations they run.

3. Operating System Services Needed

In this section we detail those operating system services which we found lacking in the process of building and maintaining our distributed batch system on UNIX. In some cases, the service was available, but implemented so differently on various platforms as to cause major problems in portability. In other cases the needed services were provided only by some UNIX platforms, or not at all. To be truly useful in building a portable batch system that can operate in a heterogeneous environment, these services need to be provided by all platforms, and use a standard interface.

3.1. Checkpointing and Location Independent Execution

One of the cornerstones of any batch system is a checkpointing facility. Running large numbers of jobs that may take days, weeks, or months of CPU time on a system that lacks such a facility is like taking off in an airplane without knowing that there is adequate fuel on board. You may arrive safely at your destination without incident, but you may also “loose everything” in a crash. Many things can and will go wrong, and inevitably some of the jobs will be aborted. It can be a nightmare to figure out which jobs should be resubmitted, and it is very unfortunate to start them from scratch after they have already accumulated several days or even weeks of CPU time. Less than a decade ago, most or our processing capacity was protected by a checkpointing facility. Today, most processing environments do not provide such a facility at all.

Another feature which is essential in a distributed batch system is location independent execution. One method of providing such location independence is to require system administrators to provide a unform view of the filesystem on every machine in the group by utilizing a network file system. This method works well for small groups of machines, but is generally not practical for large groups. Another method of providing location independence would be to force the user to list in advance all files and directories which will be touched by a batch process. We feel that such a system places an unnecessary burden on the user, and furthermore leads to errors. Avoidance of such errors is more important in a batch system than an interactive one, because the user may not be present to take corrective action at the time the error is discovered. We feel that it is desirable to provide batch jobs with an environment that closely resembles the environment on the user’s own machine. That way the user can develop an application in interactive mode, and then submit requests to the batch system without any further changes.

While in recent years great strides have been made toward providing location independent execution, to date the checkpointing issue has not been addressed at all. We feel that such a service could, and should, be provided by the operating system. Unfortunately, recent versions of UNIX not only don't address the checkpointing issue, but threaten to make even our limited checkpointing facility unworkable. The following examples should serve to illustrate both the reasons why a kernel level checkpointing facility would be preferable, and some of the ways that various flavors of UNIX have gotten in the way of our user level implementation. We contend that if system designers aren't willing to tackle checkpointing themselves, they should at least take some care to leave the door open for user level implementations.

3.1.1. Resolving Naming Conflicts

A requirement for checkpointing is the ability to recreate at restart time the status of files which were open at checkpoint time. To do so, one must keep track of which files the user process has open. Condor does this by providing a version of the `open` routine which records the pathname of the file, then executes the `open` system call to actually accomplish the operation. This leads to a naming conflict between the user level `open` routine and the `open` system call. The "standard" C library contains a routine which is designed to resolve such conflicts. This routine, called `syscall,` takes a system call number and the corresponding arguments and executes the call without the need to utter the system call's name in the code. The AIX folks have simply neglected to provide the `syscall` routine, and this is a major reason the R6000 port took months of effort rather than the usual 2 weeks. Of course if checkpointing were implemented at the kernel level, no such conflict would arise.

3.1.2. Memory Mapped Files

A requirement for location independent execution is the ability to read and write the same files, regardless where a process happens to be run. Condor provides this feature by catching the I/O calls and when necessary redirecting those calls over the network to a "shadow" process running on the machine where the files are actually located. A feature of several recent versions of UNIX is memory mapped files. While this is a very useful feature, we are alarmed at the possibility that some future version of UNIX might base essential system services on top of it. For example there has been considerable talk about the desirability of rewriting the UNIX standard I/O package to use memory mapped files. Since I/O to memory mapped files is driven by page faults, it will not be possible to redirect those file operations unless some means is provided for user level code to catch and handle such faults. Again, a kernel level implementation would not have such a problem. If system designers are unable to provide truly location independent file access, we hope they will at least continue to provide a version of the standard I/O package which does not depend on memory mapped files.

3.1.3. Executable and Core Files

Another requirement for checkpointing is the ability to save and restore the state of a process's data and stack segments. Condor utilizes the "core" file for this purpose. Core files are generated when UNIX programs are terminated abnormally, and contain among other things the data and stack segments of the process at the time of its termination. While this mechanism is provided for the purpose of debugging, it also provides essential information about the state of the process which can be used for checkpointing.

While use of the core facility as a basis for creating a checkpoint has generally worked out well, there have been a few problems caused by the rather wide variety of both format and content of core files provided by various UNIX implementations. For example, The SunOS core file doesn't contain enough information for us to determine the actual size of the stack at the time the core was created. This has forced us to use a very inefficient implementation of checkpointing on SUNs. This is another type of problem which would not arise for a kernel level checkpointing facility. Failing such an implementation, it would be very helpful if the format of core files could be standardized, and those standards included complete information about the process's data and stack areas.

It is interesting to speculate on the fact that the information in core files and executable files is quite closely related. Consider the information needed to checkpoint a typical batch process. One needs the process's text, and data areas, the register contents, and the status of files which were open at the time of the checkpoint. While neither the executable or core formats contain information about open files, the executable contains both the text, and those parts of the data segment which are to be initialized. Also while register contents are implicit in an executable file, (most registers will be zeroed), explicit contents could be included as easily. If one would then add a mechanism for recording and restoring the state of open files, the executable format could also serve both the debugging and checkpoint/restart functions. This would greatly increase the efficiency of creating a checkpoint as almost all the time to create a checkpoint is taken up by wholesale copying of large portions of the core file to the new executable.

Yet another way in which system designers could facilitate checkpointing and restarting of processes would be to provide for a more flexible way of creating processes. If one could initialize a process by specifying that various parts of the process are to be read from different files, we could also avoid the overhead of converting a core file to an executable file. In this case one would specify that the text of the process should come from the original executable file, but the data, stack, and registers should come from various locations in the core file.

3.2. Resource Management

A problem with utilizing unused cycles on user dedicated workstations is the lack of ability to control resource usage by a process. When "borrowing" a machine from a user, a batch system would like to closely determine the amount of physical memory and the size of the time slice allocated to batch processes. One problem we have with the Condor system is that if a batch job runs on a user's machine overnight, it will generally acquire almost all the physical pages on the system. When the user returns to work in the morning, the machine will seem sluggish at first because the window system, text editors, and mail system will all have to re-establish their working sets.

Another problem is determining the best strategy for when to create checkpoints of batch jobs. Ideally, one would like to create a checkpoint only when the machine's dedicated users returns from an idle period. One might then also take periodic checkpoints at a fairly long interval, (perhaps once a day), to protect against machine crashes. Unfortunately, this does not work out because part of the process of creating a checkpoint is forcing a core dump which causes a significant burst of disk activity and is annoying to the machine's owner. To avoid this Condor takes fairly frequent checkpoints of its jobs while the machine's owner is inactive. If we could closely control the time slice given to this activity, we could spread the activity out over a long enough time so that it would not be noticeable to an interactive user.

3.3. Resource Availability

A very important part of Condor is the determination of when a machine is in use by it's regular user, and when it is "idle". For Condor to consider a machine to be available for batch processing, two conditions must be met. The CPU load must be below a certain threshold, and the time since last keyboard or mouse activity must be above a particular threshold. Many UNIX systems maintain a number called the "load average", which can be used to determine CPU load. Unfortunately this calculation includes time spent waiting for I/O access to networked file systems, and thus is not always a good indicator of CPU load. Also there is no standard interface for obtaining this information, and one must go poking through the kernel's memory at run time to obtain it. Not only will the information be found at different memory locations in various UNIX kernels, the format of the information also varies from one implementation to another. For example some systems keep this number as a floating point value while others keep it as a scaled integer, furthermore various implementations utilize different scaling factors. Also while every UNIX system we have seen maintains some idea of CPU load, not all of them calculate the traditional "load average". Some just provide instantaneous data about the length of the run queue, and user software must calculate its own idea of load average if that is wanted. Information about "load" should contain more detail about load on various parts of the system such as I/O, CPU, and paging. Also this information should be provided through a standard system call or set of system calls.

Determining time since the last keyboard or mouse activity has also turned out to be highly problematic. In general workstation users will run some kind of a window system, which means that all keystrokes and mouse activity will go directly to the window system and will then be parceled out to the correct processes depending on which window has the "focus" at the time of the activity. In such a case only the window system will know how long it has been since any mouse or keyboard activity. Unfortunately, there is no standardization of how such information will be provided if indeed it is provided at all. We have introduced a very simple extension to the X server which will provide this information, and it is distributed with the X server from MIT. A number of vendors however provide their users with "value added" versions of X which do not include this feature. In such an environment, we must resort to programming tricks to get the needed information, and quite often the tricks just don't work very well. Perhaps the nicest solution to this problem is provided in SunOS where there are devices associated with both the keyboard and mouse. It is possible to query these devices for their respective times of last activity using the standardized and well documented `stat` system call. While several other UNIX platforms have similar devices, none of the others keeps the information about access time updated. While any of several means of disseminating this information would be workable for us, it should be provided by a standard interface across UNIX systems.

4. Conclusion

In this paper we have argued that batch processing is not something of the past or something which applies only to mainframe computers. We believe distributed batch processing on workstations is an area of considerable and increasing importance, and as such should be given serious consideration by operating system designers. We have detailed the most important of the services we found lacking, and have attempted to show both the reasons these services are important, and the need for the services to be presented in a uniform way across platforms.