Usenix Winter Conference
San Francisco, California. 1992

SUPPORTING CHECKPOINTING AND PROCESS MIGRATION OUTSIDE THE UNIX KERNEL

Michael Litzkow
Marvin Solomon

Computer Sciences Department
University of Wisconsin—Madison

Abstract

We have implemented both checkpointing and migration of processes under UNIX as a part of the Condor package. Checkpointing, remote execution, and process migration are different, but closely related ideas; the relationship between these ideas is explored. A unique feature of the Condor implementation of these items is that they are accomplished entirely at user level. Costs and benefits of implementing these features without kernel support are presented. Portability issues, and the mechanisms we have devised to deal with these issues, are discussed in concrete terms. The limitations of our implementation, and possible avenues to relieve some of these limitations, are presented.

1. Introduction

Condor is a software package for executing long-running, computation-intensive jobs on workstations which would otherwise be idle. Idle workstations are located and allocated to users automatically. Condor preserves a large measure of the originating machine's execution environment on the execution machine, even if the originating and execution machines do not share a common file system. Condor jobs are automatically checkpointed and migrated between workstations as needed to ensure eventual completion. This paper describes the checkpointing and remote execution mechanisms.

The Condor package as a whole has been documented elsewhere [1,2,3]. This paper focuses on the actual mechanisms for remote execution and process migration, limitations on migrating processes without kernel support, portability issues, and the evolution of our implementation of these mechanisms.

In contrast to the process migration mechanisms offered by such systems as the V-system[4], Sprite[5], and Charlotte[6], Condor supports remote execution and process migration on a variety of UNIX® platforms, and is implemented completely outside the kernel. Because it requires no changes to the operating system, Condor is portable and can be used in environments where access to the internals of the system is not possible. Condor does pay a price for this flexibility in both the speed and completeness of its process migration.

The combination of remote execution and checkpointing means that a process may migrate among processors during its lifetime, and thus must see the same file system view wherever it is run. "In kernel" process migration systems such as Sprite and Charlotte were designed for environments that ensure such uniformity. While careful

---

Authors' address: Computer Sciences Department, 1210 W. Dayton St., Madison, WI 53706; mike@cs.wisc.edu, solomon@cs.wisc.edu.

administration of NFS® or AFS®, can ensure a homogeneous environment, heterogeneous environments are common, and requiring a uniform name space would severely limit the scope of Condor's usefulness. Therefore Condor uses a technique called "remote system calls" in which requests for file-system access are trapped and forwarded to a "shadow" process on the submitting machine.

Several systems have been implemented which offer process migration, but not checkpointing. Offering both mechanisms in combination has two advantages. First, jobs are immune to machine or network crashes. Because Condor jobs aren't killed by these events, users can submit large batches of jobs, and then go on to other work, (or leave for the weekend), without worrying about all their jobs being aborted by a sudden power outage or other disaster. Second, checkpointing allows a job to remain idle during periods when all workstations are busy with interactive work. In UNIX, a process, even if stopped, consumes resources (such as swap space). Because the most important benefit of having a workstation is immediate response, our policy is to ensure absolute priority for interactive use.

2. Remote System Calls

To understand Condor's remote system calls, one must first consider normal UNIX system calls. Every UNIX program, whether or not written in the C language, is linked with the "C" library. This library provides a large number of functions, including the standard I/O library (traditionally described in section 3 of the manual), as well as interfaces to the kernel facilities described in section 2. These latter functions are generally implemented as "stubs," which push their arguments and a "call number" identifying the facility onto the user stack, and execute a machine-defined *supervisor call* instruction. In some newer implementations, the mechanism is a bit different, but the general idea is the same; each system call has a corresponding function in the C library which comprises a very thin layer between the user code and the system. Figure 1 illustrates the normal UNIX system call mechanism.

Figure 2 shows how we have altered the system call mechanism by providing a special version of the C library which performs system calls remotely. This library, like the normal C library, has a stub for each UNIX system call. These stubs either execute a request locally by mimicking the normal stubs or package it into a message which is sent to the *shadow* process. The *shadow* executes the system call on the initiating machine, packages the results, and sends them back to the stub. The stub then returns to the application program in exactly the same way the normal system call stub would have, had the call been done locally. The shadow runs with the same user and group ids, and in the same directory as the user process would have had it been executing on the submitting machine. This scheme ensures a uniform view of the file system, as well as avoiding certain security problems.

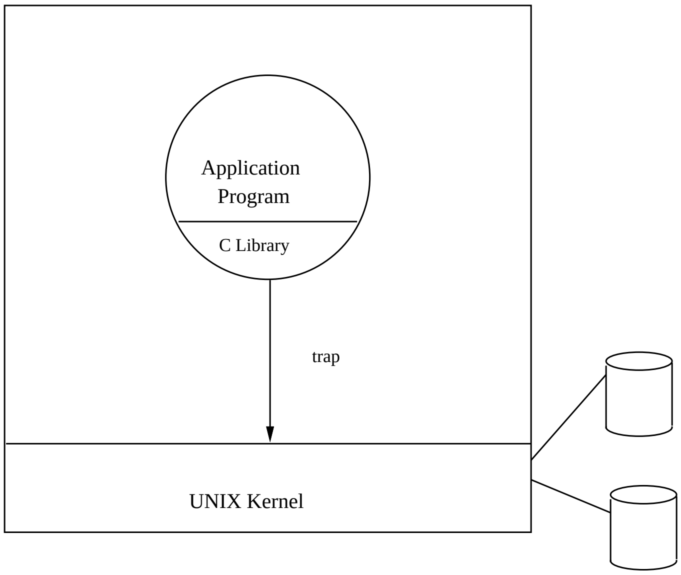

Figure 1. Normal UNIX System Calls

® AFS is a registered trademark of Transarc Corporation.

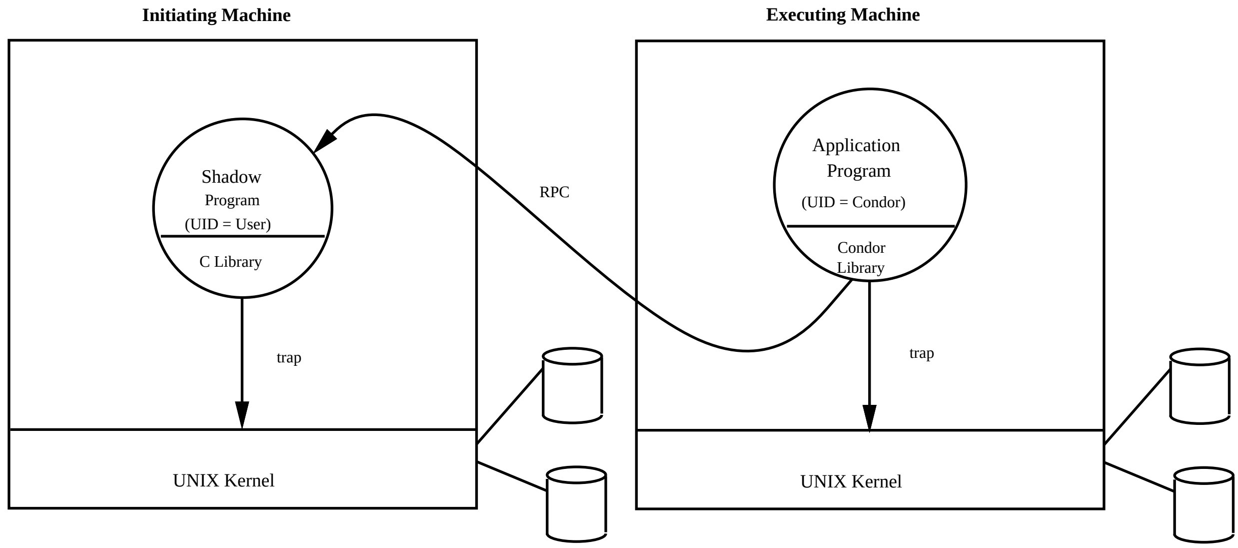

Figure 2. Remote System Calls

3. Checkpointing

Ideally, checkpointing and restarting a process means storing the process state and later restoring it in such a way that the process can continue where it left off. In other words, as far as the user code is concerned, the checkpoint never happened. In the most general case, the state of a UNIX process may include pieces of information which are known only to the kernel, or which may not be possible to recreate. For example if a process is communicating with other processes at the time of the checkpoint, and those processes are no longer extant at the time of the restoration, that part of the state cannot be reproduced. While some UNIX processes include state which cannot be saved and restored, there are a large number of jobs whose state is simple enough that they can be checkpointed and restarted without harm to their original mission. In the Condor project we have concentrated on providing practical means for remote execution, checkpointing, and migration of such jobs.

The state of a UNIX process includes the contents of memory (the text, data, and stack segments), processor registers, and the status of open files. Restoring the text segment is easy since it does not change and is available as part of the original executable file. Our approaches to saving and restoring each of the other items have changed over the years, mainly in response to portability issues. In the earliest versions of Condor, the data and stack were saved by writing them directly into a file, and the register contents were saved by architecture-specific assembler routines. Each time we ported Condor to a new hardware platform or a different version of UNIX we were “bitten” by some part of the checkpointing mechanism which was either very difficult or impossible to port. Thus we gradually changed our methods to rely on basic UNIX mechanisms rather than on specific implementations. In other words, we use standard mechanisms that have already been ported by the provider of the kernel and C library.

Our current approach is to create a new checkpoint file from pieces of the previous checkpoint and a core image. The checkpoint is itself a UNIX executable file (“a.out”). While core files are generally intended to aid in debugging a process which has committed an error, such as attempting an illegal memory access, they also serve as a portable mechanism for saving the state of a process at a given point in time. Not surprisingly, the information needed to debug a process and the information needed to restart it are almost exactly the same. The text for the new executable is of course an exact copy of the text from the original. The data area is copied directly from the core file to the initialized data area of the new executable file. The saved stack area is also copied into the new executable in a section which is not normally used by the UNIX process initialization mechanism. Restoring some of the other items in the new instantiation of the process is trickier. For example, although volatile state (such as register contents and the program counter) is recorded in the core image, restoring it directly would require machine-dependent assembler routines. Instead, we use the Unix signal-handling machinery and the setjmp/longjmp facilities in the C library.

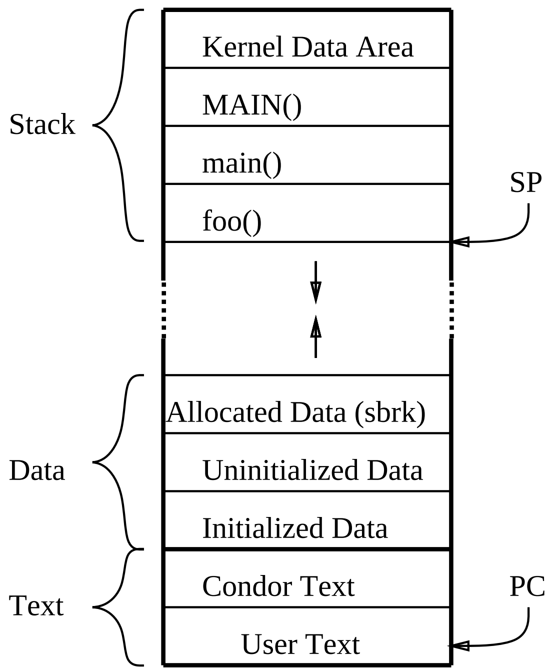

Figure 3. Normal Execution

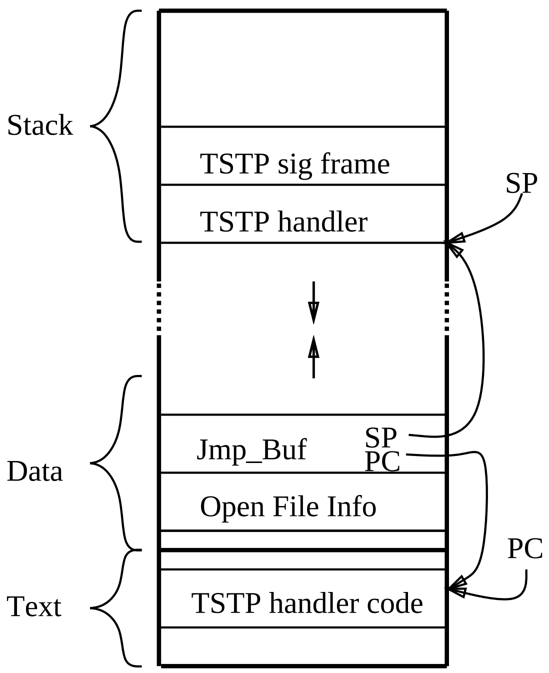

Figure 4. Checkpointing

Figures 3 through 7 illustrate various steps in checkpointing and restoring a typical UNIX process. Figure 3 depicts the virtual address space of an application program that has been linked with the Condor versions of the C library and startup routine. Routines written by the application programmer as well as those provided by the condor version of the C library co-exist at various locations in the Text segment. The data area consists of both the initialized and uninitialized data from the original executable as well as any data space allocated at run time, e.g. by the `sbrk` system call. The stack area consists of the per-process kernel data, (the "u_area", followed by stack frames for each function in the currently active execution stack. Since Condor must do its own initialization before any of the the user's code is called, the first frame on the stack is for the Condor routine `MAIN`. `MAIN` calls the user's `main` with the correct `argc`, `argv`, and `envp`.

The initialization code in `MAIN` establishes a signal handler for the `SIGTSTP` signal, which is used to inform a Condor job that it should checkpoint itself. Information about all files which the process currently has open is kept in a table by the Condor version of the `open` system call routine. The `TSTP` handler updates this information with the current file pointer location for each file using the `lseek` system call and uses the C library's `setjmp` call to record its state (including the program counter) in a buffer. Finally the handler sends itself a `SIGQUIT` signal to cause a core-dump and terminate. Figure 4 depicts the state of the process just before the `SIGQUIT` is received.

Figure 5 depicts the new checkpoint file being created from pieces of the previous checkpoint file and the core file. The data and stack areas come from the core file, while the text comes from the previous checkpoint file. The "other info" referred to is generally symbol table information, which may be preserved for the benefit of debuggers. Before a Condor process is executed for the first time, its executable file is modified to look exactly like a checkpoint file with a zero size stack area, so that every checkpoint is done the same way.

When the new process starts, UNIX will initialize it with the data saved in the executable file, thus the process is born with the data area already restored. Besides the data needed by the user code, the data area contains the jump buffer and the open file table which were saved by the `TSTP` handler in the previous execution. UNIX will initialize the process with a minimal stack, which must be replaced by the stack information saved in the checkpoint file. The code that restores the stack needs a special stack of its own so that it doesn't interfere with the "real" stack being restored. To move its stack pointer into the data segment, it establishes a handler for the `SIGUSR2` signal with a "signal stack" in the data segment, and then sends itself that signal. The USR2 handler uses the values stored

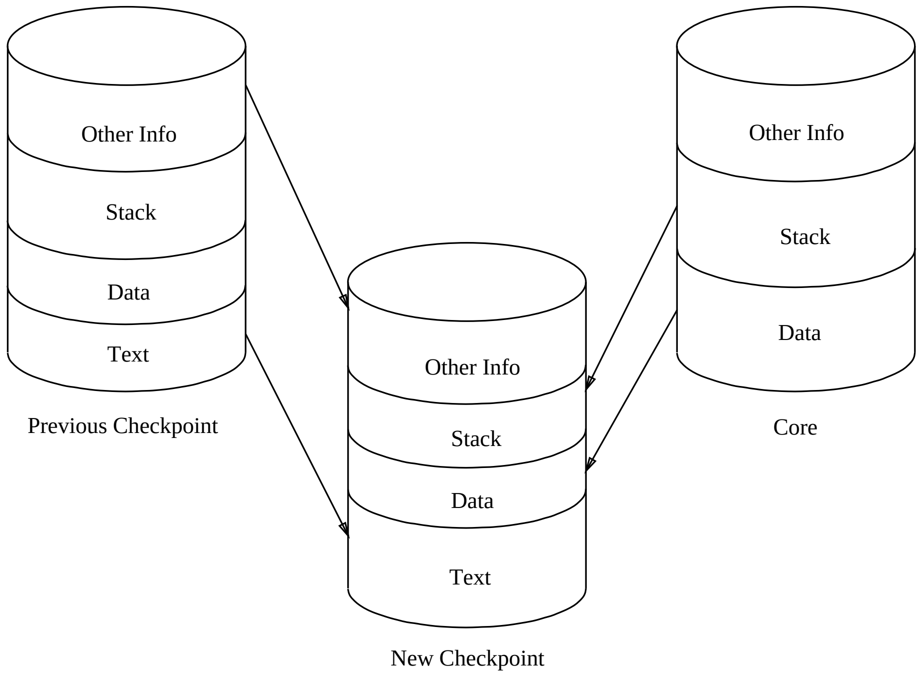

Figure 5. Creating a New Checkpoint

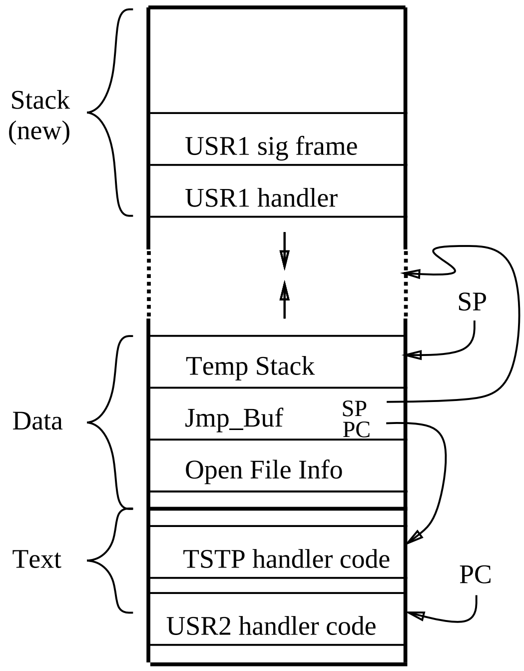

Figure 6. New Process

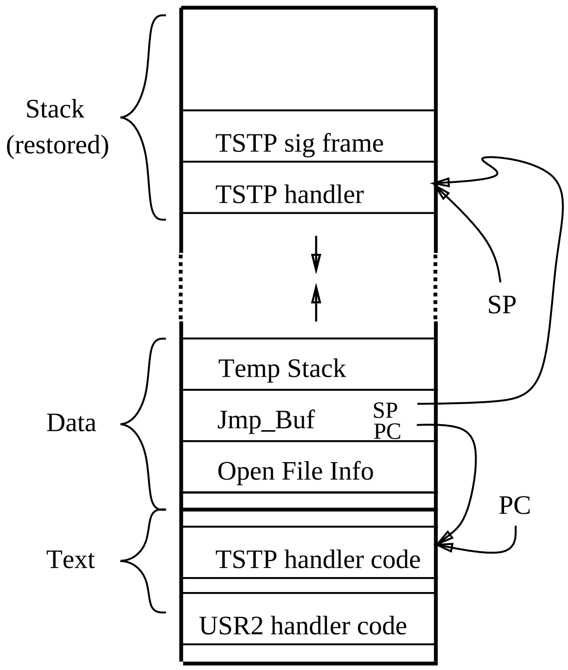

Figure 7. Restored Stack

in the file table to reopen all files which were open at the time of the checkpoint, and seek them to their correct offsets. Figure 6 shows the state of the process just before the stack is overwritten with the saved stack information.

After the stack is restored, the USR2 handler calls `longjmp` with the state saved by the `setjmp` in the `TSTP` handler of the original process. The stack pointer and program counter are restored to their values as of the `setjmp` call, and the `SIGTSTP` handler simply returns, restoring other volatile state (such as processor registers) as it was before the `SIGTSTP` signal. Figure 7 shows the state of the process after the `longjmp`, and just before the `TSTP` handler returns.

Condor uses inherently portable mechanisms to restore a process's stack and registers from their checkpointed values, and does not resort to any assembler code whatsoever. Since individual `write` calls are not traced, the file recovery scheme requires that all file updates be *idempotent* (repeating operations does no harm). For typical UNIX file usage, this restriction seems not be much of a problem.

4. Network File Systems

The remote system call scheme described above assures correct operation in inhomogeneous file-naming environments. If, however, a network file system such as NFS is in use, an important optimization is possible. For example, consider the common situation in which a single file server serves an entire local-area network. Each I/O request is forwarded from the "worker" machine to the shadow, which then accesses the file over the network using NFS. If the stub executed the I/O request directly, the results would be the same, but one network round-trip would be avoided. The situation is even worse if the file is physically located on the worker machine: Two network round-trips are used when none are needed. To improve the performance in these cases, Condor incorporates a mechanism to avoid the use of remote system calls when a file is more directly accessible. At the time of an `open` request, the stub sends a name translation request to the submitting machine. The shadow process responds with a translated pathname in the form of *hostname:pathname*, where the *hostname* may refer to a file server, and the *pathname* is the name by which the file is known on the server (which may be different from the pathname on the submitting machine, because of mount points and symbolic links). The stub then examines the mount table on the machine where it is executing, and if possible accesses the file without using the shadow process. Whenever a process is checkpointed and restarted on another machine, the name translation process is repeated, since access to remotely mounted files may vary among the execution machines. Figure 8 illustrates an example where the pathnames of the target file on the initiating and executing machines are different. The `open` routine sends a request to the shadow for a translation of the pathname "/u2/john", and the shadow responds with the external name "fileserver:/staff/john". The `open` routine then translates the external name to the name by which the file is known on the executing machine, namely "/usr1/john", and opens the file using normal system calls, e.g. via NFS.

5. Limitations

Condor has been found to be an extremely useful tool for a particular class of application: A single-process, computation-intensive, long-running job. Other kinds of application are currently hindered by Condor's limitations,

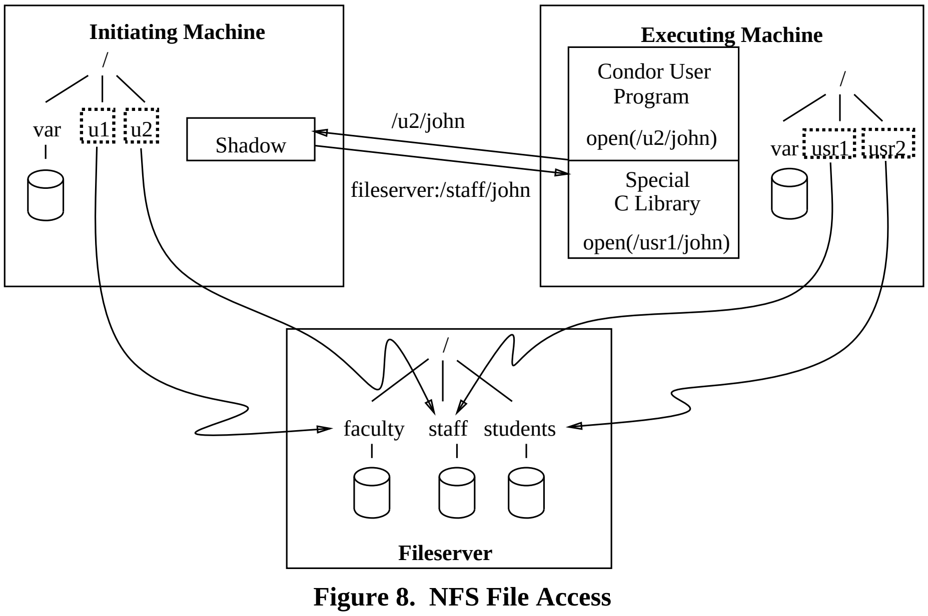

Figure 8. NFS File Access

some of which could be relatively easily lifted (and probably will be in future releases), and others of which appear to be inherent. Condor currently does not support applications that use signals, timers, memory-mapped files, or shared libraries. We expect some of these restrictions will be relaxed in future versions of Condor. The most painful limitation is that Condor does not support any sort of inter-process communication. Aside from the well-known difficulty of achieving a consistent snapshot of a multi-process program, there is the problem of hidden state in the form of messages inside pipes. Implementing multi-process Condor jobs would also greatly complicate the Condor scheduling algorithm (which is beyond the scope of this paper).

The high cost of process migration limits the usefulness of Condor for small jobs. Process migration involves many steps: First the process causes a core dump. The core file and executable module are combined to produce a new executable, which is returned to the submitting machine, where it resides until a new execution site becomes available. The executable is then transferred over the network, and execution resumes. Transferring a Condor job with a 6 megabyte address space between two DECstation 3100's on an Ethernet we obtained rates of about 250K Bps (bytes/sec) for dumping core, 130K Bps for updating the checkpoint file, 250K Bps for transferring the new checkpoint file from the execution site to the originating machine, and 250K Bps for transferring the new checkpoint file from the originating machine to a new execution site. Altogether, it takes about two minutes to migrate such a process. In practice the job must wait on the originating machine until the scheduler can locate a new execution site, and the scheduler only runs once every 10 minutes.

There are however some significant benefits to an implementation which does not depend on access to the kernel code. Most obvious of these is the portability which may be obtained. Condor currently runs on ten different hardware/software platforms: the IBM R6000 running AIX, the IBM RT/PC running AOS, Sun 3 and Sun 4 workstations running SunOS, the Silicon Graphics 4D workstation running IRIX, the Hewlett-Packard 9000 under 4.3 BSD UNIX, the Digital Equipment DECstation and VAXstation running Ultrix, and the Sequent Symmetry running Dynix. A related but somewhat different issue is availability in the real world. If our process migration and checkpointing facilities were embedded in an operating system, then it is unlikely many others would run that system even if it were ported to their hardware and freely available. If Condor required a homogeneous server-based file environment, many sites would not be able to use it. As it is, Condor is available via anonymous ftp (contact the first author for details) and is running at many sites throughout the United States and Europe.

6. Future Work

Work with Condor is ongoing at Wisconsin. We are particularly interested in moving Condor to larger groups of machines, including groups which are connected by wide-area rather than local-area networks. We are also interested in integrating constrained forms of inter process communication with Condor such as Linda[7]. The three areas of continuing effort relating to the mechanisms described here are efficiency, functionality, and portability. With regard to efficiency, many methods are possible for saving, migrating, and restoring the necessary pieces of a UNIX process. In some circumstances, we could transfer processes directly between execution sites rather than always sending a checkpoint file back to the originating site. We could use data compression to reduce the volume of data transferred and stored. We could read the stack and data directly from a core file into a new instantiation of a process rather than converting the core file to an executable module.

In the area of functionality, the most often requested item is support for signals. This feature is non-trivial, because much of the information regarding user defined signal handlers, the handling of signals on special stacks, the blocking and unblocking of signals, and so forth is maintained by the kernel in ways which vary among UNIX implementations. Nonetheless, a restricted form of signal facility could probably be provided that would significantly increase the number of applications supported. Unfortunately both optimizations and more general UNIX semantics work against portability. It is often easy to find a solution to one of the optimization or functionality problems which works on some platforms, but not others.

7. Conclusions

Condor has accomplished checkpointing and process migration on "vanilla" UNIX systems. Both facilities are generally only implemented inside an operating system, if at all. Although this design decision incurs a cost, both in efficiency and generality of application programs supported, it makes Condor available to far wider user community,

8. Acknowledgements

Parts of this research have been supported by the National Science Foundation under grant DCR-8521228, by a Digital Equipment Corporation External Research Grant, and by an International Business Machines Joint Study agreement. The port to the Silicon Graphics 4D workstation was funded by NRL/SFA Inc.

A great many people have contributed time, guidance, and ideas to Condor; a chosen few have also contributed code. We wish to particularly thank our users who have been very patient and supportive throughout the development of Condor, and most especially those brave souls who are not users of Condor, but have allowed their machines to be candidates for remote execution anyway. The original idea for Condor was suggested by David DeWitt, based on a suggestion from Maurice Wilkes. It was Dewitt who insisted that Condor must be implemented without any changes whatever to the UNIX kernel. Miron Livny and Matt Mutka first convinced us that checkpointing was both possible and necessary. Livny and Mutka also provided much guidance in the overall philosophy and structure behind the design of Condor, as well as the name "Condor" itself. Allan Bricker implemented portions of Condor and contributed many useful ideas. It was Allan who first suggested using the core file for saving the state of a process and the `signal` mechanism for getting the registers restored.

Availability

Condor is available via anonymous ftp from "shorty.cs.wisc.edu", (128.105.2.8). Mail "condor-request@cs.wisc.edu" if you would like to be on our mailing list.

References

[1]. M. Litzkow, "Remote Unix—Turning Idle Workstations Into Cycle Servers," *Proceedings of the Usenix Summer Conference*, Phoenix, Arizona, June 1987.

[2]. M. Litzkow, M. Livny, and M Mutka, "Condor—A Hunter of Idle Workstations," *8th International Conference on Distributed Computing Systems*, Jan Jose, Calif, June 1988.

[3]. M. Litzkow and M. Livny, "Experience With the Condor Distributed Batch System," *Proceedings of the IEEE Workshop on Experimental Distributed Systems*, Hunstville, AL Oct. 1990.

[4]. M. Theimer, K. Lantz, and D. Cheriton, "Preemptable remote execution facilities for the V-System," *Proceedings of the 10th Symposium on Operating System Principles*, December 1985.

[5]. F. Douglis and J. Ousterhout, "Transparent Process Migration: Design Alternatives and the Sprite Implementation," *Software Practice and Experience*, to appear.

[6]. Y. Artsy and R. Finkel, "Designing a process migration facility: The Charlotte experience," *IEEE Computer*, September 1988.

[7]. N. Carriero and D. Gelernter, "The S/Net's Linda Kernel," *ACM Transactions on Computer Systems*, Vol.4, No. 2, May 1986.

Michael Litzkow has been a member of the technical staff at the University of Wisconsin—Madison since 1983. Past projects include the Csnet Nameserver and a fileserver for the Charlotte distributed operating system. He received his B.S. in computer sciences from the University of Wisconsin in 1983.

Marvin Solomon received a B.S. degree in Mathematics from the University of Chicago and M.S. and Ph.D. degrees in Computer Science from Cornell University. In 1976, he joined the Department of Computer Sciences at the University of Wisconsin—Madison, where he is currently Professor. Dr. Solomon was Visiting Lecturer at Aarhus (Denmark) University during 1975-76 and Visiting Scientist at IBM Research in San Jose California during 1984-85. His research interests include programming languages, program development environments, operating systems, and computer networks. He is a member of the IEEE Computer Society and the Association for Computing Machinery.