CONDOR TECHNICAL SUMMARY

Allan Bricker
Michael Litzkow
and
Miron Livny

Computer Sciences Department
University of Wisconsin - Madison
allan@chorus.fr, mike@cs.wisc.edu, miron@cs.wisc.edu

1. Introduction to the Problem

A common computing environment consists of many workstations connected together by a high speed local area network. These workstations have grown in power over the years, and if viewed as an aggregate they can represent a significant computing resource. However in many cases even though these workstations are owned by a single organization, they are dedicated to the exclusive use of individuals.

In examining the usage patterns of the workstations, we find it useful to identify three ‘‘typical’’ types of users. ‘‘Type 1’’ users are individuals who mostly use their workstations for sending and receiving mail or preparing papers. Theoreticians and administrative people often fall into this category. We identify many software development people as ‘‘type 2’’ users. These people are frequently involved in the debugging cycle where they edit software, compile, then run it possibly using some kind of debugger. This cycle is repeated many times during a typical working day. Type 2 users sometimes have too much computing capacity on their workstations such as when editing, but then during the compilation and debugging phases they could often use more CPU power. Finally there are ‘‘type 3’’ users. These are people who frequently do large numbers of simulations, or combinitoric searches. These people are almost never happy with just a workstation, because it really isn’t powerful enough to meet their needs. Another point is that most type 1 and type 2 users leave their machines completely idle when they are not working, while type 3 users often keep their machines busy 24 hours a day.

Condor is an attempt to make use of the idle cycles from type 1 and 2 users to help satisfy the needs of the type 3 users. The *condor* software monitors the activity on all the participating workstations in the local network. Those machines which are determined to be idle, are placed into a resource pool or ‘‘processor bank’’. Machines are then allocated from the bank for the execution of jobs belonging to the type 3 users. The bank is a dynamic entity; workstations enter the bank when they become idle, and leave again when they get busy.

2. Design Features

(1) No special programming is required to use condor. Condor is able to run normal UNIX[1] programs, only requiring the user to relink, not recompile them or change any code.

(2) The local execution environment is preserved for remotely executing processes. Users do not have to worry about moving data files to remote workstations before executing programs there.

[1] UNIX is a trademark of AT&T.

(3) The condor software is responsible for locating and allocating idle workstations. Condor users do not have to search for idle machines, nor are they restricted to using machines only during a static portion of the day.

(4) ‘‘Owners’’ of workstations have complete priority over their own machines. Workstation owners are generally happy to let somebody else compute on their machines while they are out, but they want their machines back promptly upon returning, and they don’t want to have to take special action to regain control. Condor handles this automatically.

(5) Users of condor may be assured that their jobs will eventually complete. If a user submits a job to condor which runs on somebody else’s workstation, but the job is not finished when the workstation owner returns, the job will be checkpointed and restarted as soon as possible on another machine.

(6) Measures have been taken to assure owners of workstations that their filesystems will not be touched by remotely executing jobs.

(7) Condor does its work completely outside the kernel, and is compatible with Berkeley 4.2 and 4.3 UNIX kernels and many of their derivitives. You do not have to run a custom operating system to get the benefits of condor.

3. Limitations

(1) Only single process jobs are supported, i.e. the fork(2), exec(2), and similar calls are not implemented.

(2) Signals and signal handlers are not supported, i.e. the signal(3), sigvec(2), and kill(2) calls are not implemented.

(3) Interprocess communication (IPC) calls are not supported, i.e. the socket(2), send(2), recv(2), and similar calls are not implemented.

(4) All file operations must be idempotent — read-only and write-only file accesses work correctly, but programs which both read and write the same file may not.

(5) Each condor job has an associated ‘‘checkpoint file’’ which is approximately the size of the address space of the process. Disk space **must** be available to store the checkpoint file **both** on the **submitting** and **executing** machines.

(6) Condor does a significant amount of work to prevent security hazards, but some loopholes are known to exist. One problem is that condor user jobs are supposed to do only remote system calls, but this is impossible to guarantee. User programs are limited to running as an ordinary user on the executing machine, but a sufficiently malicious and clever user could still cause problems by doing local system calls on the executing machine.

(7) A different security problem exists for owners of condor jobs who necessarily give remotely running processes access to their own file system.

4. Overview of Condor Software

In some circumstances condor user programs may utilize ‘‘remote system calls’’ to access files on the machine from which they were submitted. In other situations files on the submitting machine are accessed more efficiently by use of NFS. In either case the user program is provided with the illusion that it is operating in the environment of the submitting machine. Programs written for operation in the local environment are converted to using remote file access simply by relinking with a special library. The remote file access mechanisms are described in Section 5.

Condor user programs are constructed in such a way that they can be checkpointed and restarted at will. This assures users that their jobs will complete, even if they are interrupted during execution by the return of a hosting workstation’s owner. Checkpointing is also implemented by linking with the special library. The checkpointing mechanism is described more fully in Section 6.

Condor includes control software consisting of three daemons which run on each member of the condor pool, and two other daemons which run on a single machine called the **central manager**. This software automatically locates and releases ‘‘target machines’’ and manages the queue of jobs waiting for condor resources. The control software is described in Section 7.

5. Remote File Access

Condor programs executing on a remote workstation may access files on the submitting workstation in one of two ways. The preferred mechanism is direct access to the file via NFS, but this is only possible if those files appear to be in the filesystem of the executing machine, i.e. they are either physically located on the executing machine, or are mounted there via NFS. If the desired file does not appear in the filesystem of the executing workstation, condor provides called ‘‘remote system calls’’ which allows access to most of the normal system calls available on the submitting machine, including those that access files. In either case, the remote access is completely transparent to the user program, i.e. it simply executes such system calls as open(), close(), read(), and write(). The condor library provides the remote access below the system call level.

To better understand how the condor remote system calls work, it is appropriate to quickly review how normal UNIX system calls work. Figure 1 illustrates the normal UNIX system call mechanism. The user program is linked with a standard library called the ‘‘C library’’. This is true even for programs written in languages other than C. The C library contains routines, often referred to as ‘‘system call stubs’’, which cause the actual system calls to happen. What the stubs really do is push the system call number, and system call arguments onto the stack, then execute an instruction which causes a trap to the kernel. When the kernel trap handler is called, it reads the system call number and arguments, and performs the system call on behalf of the user program. The trap handler will then place the system call return value in a well known register or registers, and return control to the user program. The system call stub then returns the result to the calling process, completing the system call.

Figure 2 illustrates how this mechanism has been altered by condor to implement remote system calls. Whenever condor is executing a user program remotely, it also runs a ‘‘shadow’’ program on the initiating host. The **shadow** acts an agent for the remotely executing program in doing system calls. Condor user programs are linked with a special version of the C library. The special version contains all of the functions provided by the normal C library, but the system call stubs have been changed to accomplish remote system calls. The remote system call stubs package up the system call number and arguments and send them to the **shadow** using the network. The **shadow**, which is linked with the normal C library, then executes the system call on behalf of the remotely running job in the normal way. The **shadow** then packages up the results of the system call and sends them back to the system call stub in the special C library on the remote machine. The remote system call stub then returns its result to the

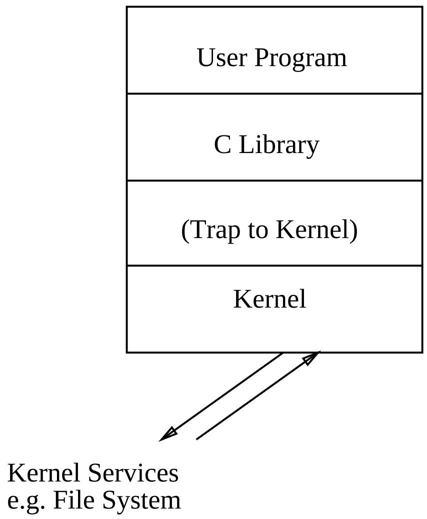

Figure 1: Normal UNIX System Calls

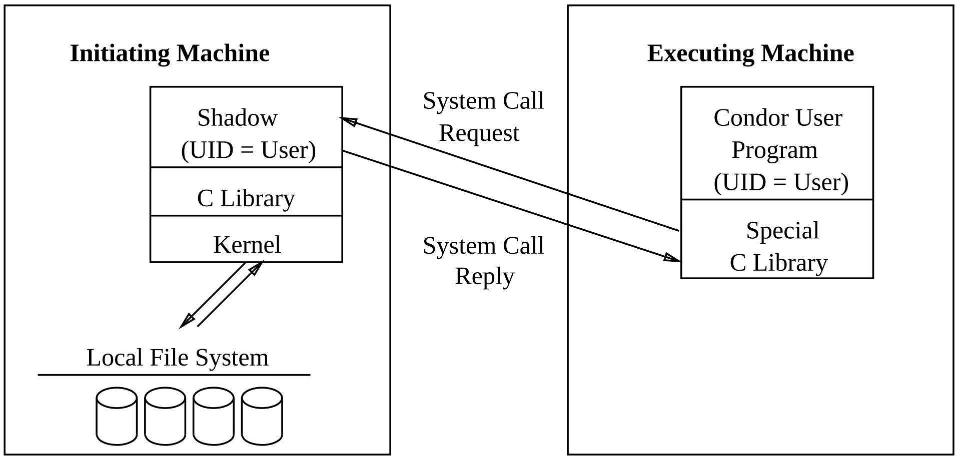

Figure 2: Remote System Calls

calling procedure which is unaware that the call was done remotely rather than locally. Note that the **shadow** runs with its UID set to the owner of the remotely running job so that it has the correct permissions into the local file system.

In many cases, it is more efficient to access files using NFS rather than via the remote system call mechanism. This is generally the case when the desired file is not physically located on the submitting machine, e.g. the file actually resides on a fileserver. In such a situation data transferred to or from the file would require two trips over the network, one via NFS to the shadow, and another via remote system call to the condor user program. The open() system call provided in the condor version of the C library can detect such circumstances, and will open files via NFS rather than remote system calls when this is possible. The condor open() routine does this by sending the desired pathname to the shadow

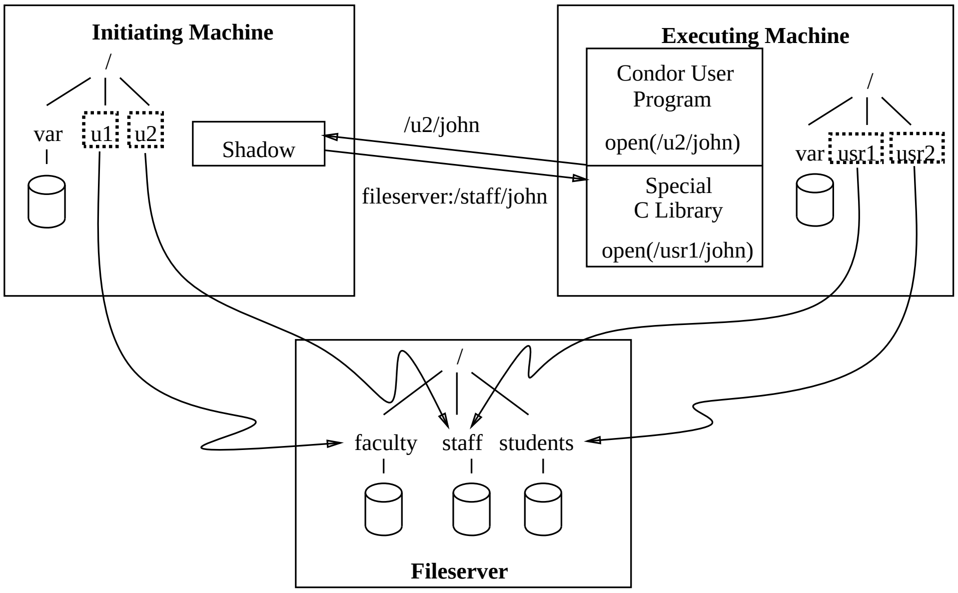

Figure 3: NFS File Access

program on the submitting machine along with a translation request. The shadow replies with the name of the host where the file physically resides along with a pathname for the file which is appropriate on the host where the file actually resides. The open() routine then examines the mount table on the executing machine to determine whether the file is accessible via NFS and what pathname it is known by. This pathanme translation is repeated whenever the user job moves from one execution machine to another. Note that condor does not assume that all files are available from all machines, nor that every machine will mount filesystems in such a way that the same pathnames refer to the same physical files. Figure 3 illustrates a situation where the condor user program opens a file which is known as ‘‘/u2/john’’ on the submitting machine, but the same file is known as ‘‘/usr1/jobn’’ on the executing machine.

6. Checkpointing

To checkpoint a UNIX process, several things must be preserved. The text, data, stack, and register contents are needed, as well as information about what files are open, where they are seek’d to, and what mode they were opened in. The data, and stack are available in a core file, while the text is available in the original executable. Condor gathers the information about currently open files through the special C library. In condor’s special C library the system call stubs for ‘‘open’’, ‘‘close’’, and ‘‘dup’’ not only do those things remotely, but they also record which files are opened in what mode, and which file descriptors correspond to which files.

Condor causes a running job to checkpoint by sending it a signal. When the program is linked, a special version of ‘‘crt0’’ is included which sets up CKPT() as that signal handler. When CKPT() is called, it updates the table of open files by seeking each one to the current location and recording the file position. Next a setjmp(3) is executed to save key register contents in a global data area, then the process sends itself a signal which results in a core dump. The condor software then combines the original executable file, and the core file to produce a ‘‘checkpoint’’ file, (figure 4). The checkpoint file is

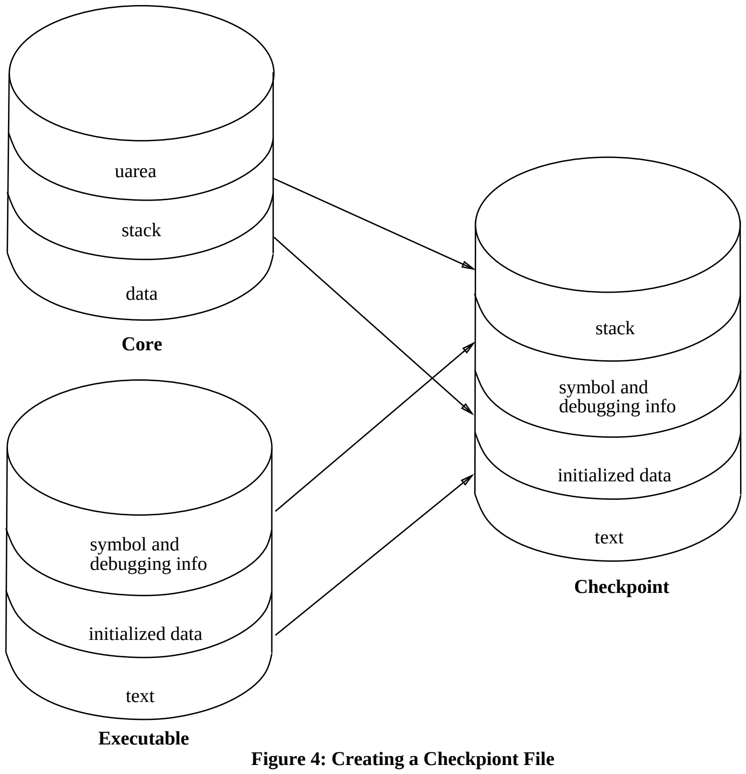

Figure 4: Creating a Checkpiont File

itself executable.

When the checkpoint file is restarted, it starts from the crt0 code just like any UNIX executable, but again this code is special, and it will set up the restart() routine as a signal handler with a special signal stack, then send itself that signal. When restart() is called, it will operate in the temporary stack area and read the saved stack in from the checkpoint file, reopen and reposition all files from the saved file state information, and execute a longjmp(3) back to CKPT(). When the restart routine returns, it does so with respect to the restored stack, and CKPT() returns to the routine which was active at the time of the checkpoint signal, not crt0. To the user code, checkpointing looks exactly like a signal handler was called, and restarting from a checkpoint looks like a return from that signal handler.

7. Control Software

Each machine in the condor pool runs two daemons, the **schedd** and the **startd**. In addition, one machine runs two other daemons called the **collector** and the **negotiator**. While the **collector** and the **negotiator** are separate processes, they work closely together, and for purposes of this discussion can be considered one logical process called the **central manager**. The **central manager** has the job of keeping track of which machines are idle, and allocating those machines to other machines which have condor jobs to run. On each machine the **schedd** maintains a queue of condor jobs, and negotiates with the **central manager** to get permission to run those jobs on remote machines. The **startd** determines whether its machine is idle, and also is responsible for starting and managing foreign jobs which it may be hosting. On machines running the X window system, an additional daemon the **kbdd** will periodically inform the **startd** of the keyboard and mouse ‘‘idle time’’. Periodically the **startd** will examine its machine, and update the **central manager** on its degree of "idleness". Also periodically the **schedd** will examine its job queue and update the **central manager** on how many jobs it wants to run and how many jobs it is currently running. Figure 5 illustrates the situation when no condor jobs are running.

At some point the **central manager** may learn that *machine b* is idle, and decide that *machine c* should execute one of its jobs remotely on *machine b*. The **central manager** will then contact the **schedd** on *machine c* and give it ‘‘permission’’ to run a job on *machine b*. The **schedd** on *machine c* will then select a job from its queue and spawn off a **shadow** process to run it. The **shadow** will then contact the **startd** on *machine b* and tell it that it would like to run a job. If the situation on *machine b* hasn’t changed since the last update to the **central manager**, *machine b* will still be idle, and will respond with an OK. The **startd** on *machine b* then spawns a process called the **starter**. It’s the

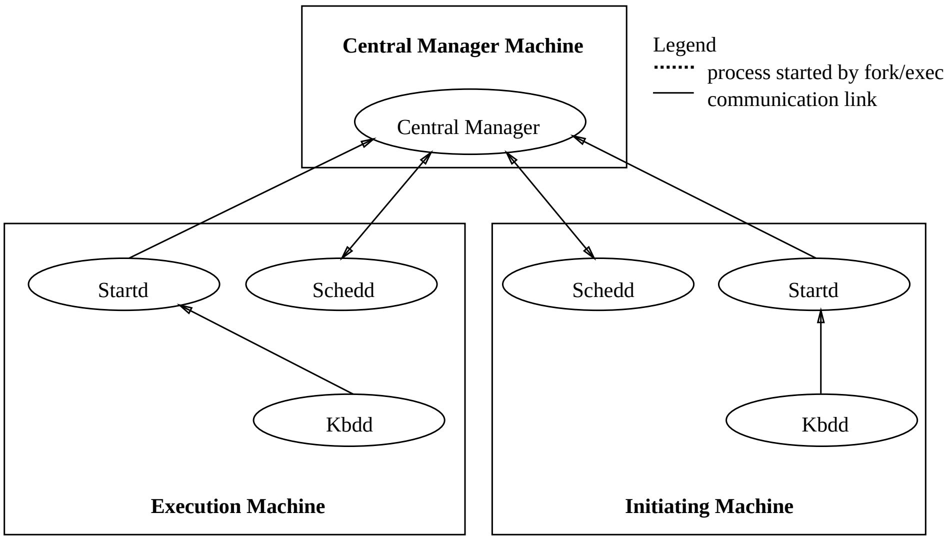

Figure 5: Condor Processes With No Jobs Running

starter's job to start and manage the remotely running job (figure 6).

The **shadow** on *machine c* will transfer the checkpoint file to the **starter** on *machine b*. The **starter** then sets a timer and spawns off the remotely running job from *machine c* (figure 7). The **shadow** on *machine c* will handle all system calls for the job. When the **starter's** timer expires it will send the user job a checkpoint signal, causing it to save its file state and stack, then dump core. The **starter** then builds a new version of the checkpoint file which is stored temporarily on *machine b*. The **starter** restarts the job from the new checkpoint file, and the cycle of execute and checkpoint continues. At some point, either the job will finish, or *machine b's* user will return. If the job finishes, the job's owner is notified by mail, and the **starter** and **shadow** clean up. If *machine b* becomes busy, the **startd** on *machine b* will detect that either by noting recent activity on one of the tty or pty's, or by the rising load average. When the **startd** on *machine b* detects this activity, it will send a ''suspend'' signal to the **starter**, and the **starter** will temporarily suspend the user job. This is because frequently the owners of machines are active for only a few seconds, then become idle again. This would be the case if the owner were just checking to see if there were new mail for example. If *machine b* remains busy for a period of about 5 minutes, the **startd** there will send a ''vacate'' signal to the **starter**. In this case, the **starter** will abort the user job and return the latest checkpoint file to the **shadow** on *machine c*. If the job had not run long enough on *machine b* to reach a checkpoint, the job is just aborted, and will be restarted later from the most recent checkpoint on *machine c*. Notice that the **starter** checkpoints the condor user job periodically rather than waiting until the remote workstation's owner wants it back. Checkpointing, and in particular core dumping, is an I/O intensive activity which we avoid doing when the hosting workstation's owner is active.

8. Control Expressions

The condor control software is driven by a set of powerful ''control expressions''. These expressions are read from the file ''˜condor/condor_config'' on each machine at run time. It is often convenient for many machines of the same type to share common control expressions, and this may be done through a fileserver. To allow flexibility for control of individual machines, the file ''˜condor/condor_config.local'' is provided, and expressions defined there take precedence over those defined in condor_config. Following are examples of a few of the more important condor control expressions with explanations. See condor_config(5) for a detailed description of all the control expressions.

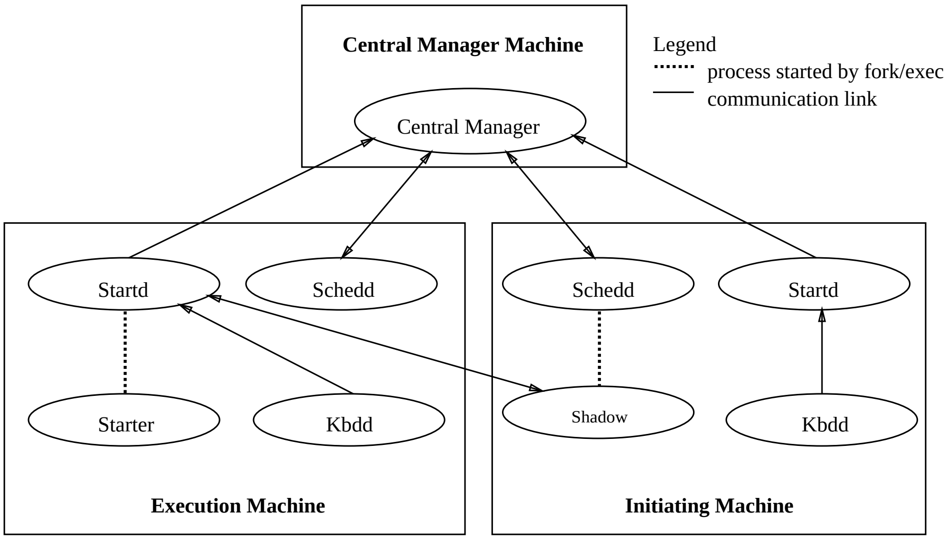

Figure 6: Condor Processes While Starting a Job

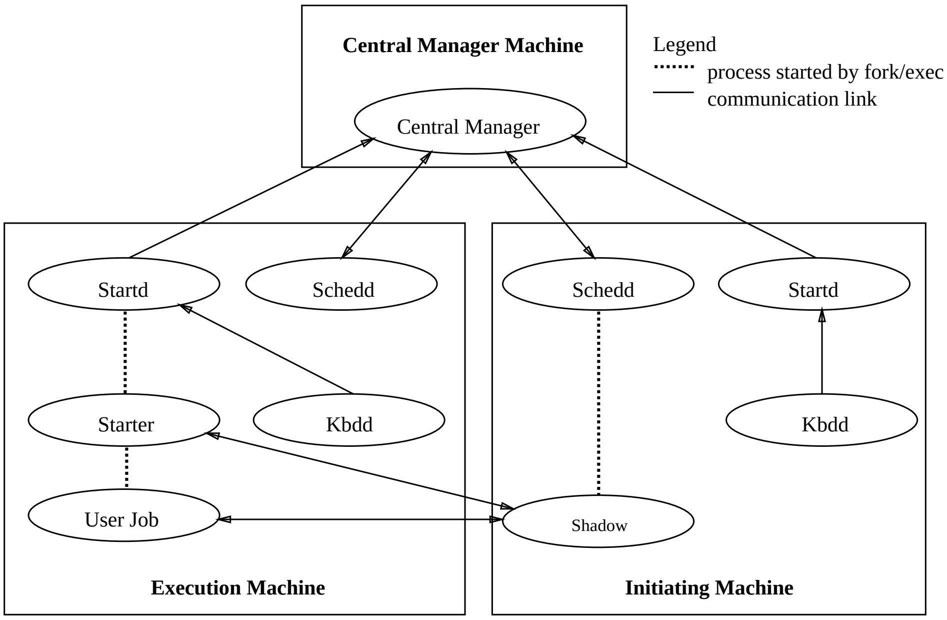

Figure 7: Condor Processes With One Job Running

8.1. Starting Foreign Jobs

This set of expressions is used by the **startd** to determine when to allow a foreign job to begin execution.

```
BackgroundLoad = 0.3
StartIdleTime    = 15 * $(MINUTE)
CPU_Idle         = LoadAvg <= $(BackgroundLoad)
START            : $(CPU_Idle) && KeyboardIdle > $(StartIdleTime)
```

This example of the START expression specifies that to begin execution of a foreign job the load average must be less than 0.3, and there must have been no keyboard activity during the past 15 minutes.

Other expressions are used to determine when to suspend, resume, and abort foreign jobs.

8.2. Prioritizing Jobs

The **schedd** must prioritize its own jobs and negotiate with the **central manager** to get permission to run them. It uses a control expression to assign priorities to its local jobs.

```
PRIO      : (UserPrio * 10) + $(Expanded) - (QDate / 1000000000.0)
```

‘‘UserPrio’’ is a number defined by the jobs owner in a similar spirit to the UNIX ‘‘nice’’ command. ‘‘Expanded’’ will be 1 if the job has already completed some execution, and 0 otherwise. This is an issue because expanded jobs require more disk space than unexpanded ones. ‘‘QDate’’ is the UNIX time when the job was submitted. The constants are chosen so that ‘‘UserPrio’’ will be the major criteria, ‘‘Expanded’’ will be less important, and ‘‘QDate’’ will be the minor criteria in determining job priority. ‘‘UserPrio’’, ‘‘Expanded’’, and ‘‘QDate’’ are variables known to the **schedd** which it determines for each job before applying the PRIO expression.

8.3. Prioritizing Machines

The **central manager** does not keep track of individual jobs on the member machines. Instead it keeps track of how many jobs a machine wants to run, and how many it is running at any particular time. This keeps the information that must be transmitted between the **schedd** and the **central manager** to a minimum. The **central manager** has the job of prioritizing the machines which want to run jobs, then it can give permission to the **schedd** on high priority machines and let them make their own decision about what jobs to run.

UPDATE_PRIO : Prio + Users - Running

Periodically the **central manager** will apply this expression to all of the machines in the pool. The priority of each machine will be incremented by the number of individual users on that machine who have jobs in the queue, and decremented by the number of jobs that machine is already executing remotely. Machines which are running lots of jobs will tend to have low priorities, and machines which have jobs to run, but can't run them, will accumulate high priorities.

9. Acknowledgements

This project is based on the idea of a ''processor bank'', which was introduced by Maurice Wilkes in connection with his work on the Cambridge Ring.[2]

We would like to thank Don Neuhengen and Tom Virgilio for their pioneering work on the remote system call implementation; Matt Mutka and Miron Livny for first convincing us that a general checkpointing mechanism could be practical and for ideas on how to distribute control and prioritize the jobs; and David Dewitt and Marvin Solomon for their continued guidance and support throughout this project.

This research was supported by the National Science Foundataion under grants MCS81-05904 and DCR-8512862, by a Digital Equipment Corporation External Research Grant, and by an International Business Machines Department Grant. Porting to the SGI 4D Workstation was funded by NRL/SFA.

10. Copyright Information

Authors: Allan Bricker, Michael J. Litzkow, and others.
University of Wisconsin, Computer Sciences Dept.

11. Bibliography

(1) Mutka, M. and Livny, M. ''Profiling Workstations' Available Capacity For Remote Execution''. *Proceedings of Performance-87, The 12th IFIP W.G. 7.3 International Symposium on Computer Performance Modeling, Measurement and Evaluation.* Brussels, Belgium,

[2]Wilkes, M. V., Invited Keynote Address, 10th Annual International Symposium on Computer Architecture, June 1983.

December 1987.

(2) Litzkow, M. ‘‘Remote Unix — Turning Idle Workstations Into Cycle Servers’’. *Proceedings of the Summer 1987 Usenix Conference.* Phoenix, Arizona. June 1987

(3) Mutka, M. *Sharing in a Privately Owned Workstation Environment.* Ph.D. Th., University of Wisconsin, May 1988.

(4) Litzkow, M., Livny, M. and Mutka, M. ‘‘Condor — A Hunter of Idle Workstations’’. *Proceedings of the 8th International Conference on Distributed Computing Systems.* San Jose, Calif. June 1988

(5) Bricker, A. and Litzkow M. ‘‘Condor Installation Guide’’. May 1989

(6) Bricker, A. and Litzkow, M. Unix manual pages: condor_intro(1), condor(1), condor_q(1), condor_rm(1), condor_status(1), condor_summary(1), condor_config(5), condor_control(8), and condor_master(8). January 1991