Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System

Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny

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

1 Introduction

Condor is a distributed batch processing system for UNIX developed at the University of Wisconsin. This system schedules jobs on idle workstations in a network, resulting in more efficient resource utilization. It is of primary importance in Condor to ensure that the owner of a workstation does not pay a penalty for adding his or her workstation to the Condor pool of workstations. So, a job must have the ability to immediately vacate a workstation when the owner begins to use it, and either migrate to another idle workstation or queue until one becomes idle.

To allow migrating jobs to make progress, Condor must be able to start the vacated job from where it left off. Condor does this by writing a checkpoint of the process's state before vacating. A checkpoint file contains the process's data and stack segments, as well as information about open files, pending signals, and CPU state. Condor gives a program the ability to checkpoint itself by providing a checkpointing library. Programs submitted to be run by the Condor system are re-linked (but not re-compiled) to include this library.

2 Checkpoint and Migration of a UNIX Process

One of the major components of Condor is its facility for transparently checkpointing and restarting a process, possibly on a different machine. By "transparent" we mean that the user code is not specially written to accommodate the checkpoint/restart or migration and generally has no knowledge that such an event has taken place. This mechanism is implemented entirely at user level, with absolutely no modifications to the UNIX kernel. While implementing checkpoint/restart and migration at user level is great for portability, it should be pointed out from the beginning that our method does have some limitations. These are discussed in detail is section 4.2.

Process checkpoint/restart can enable process migration. Checkpointing and restarting generally implies that the state of a process is saved in stable storage using the file system. Note however that in an environment where processes can access files from any of a group of machines, (possibly via AFS or NFS cross mounting of file systems), then by transporting a checkpoint file to another machine we accomplish process migration. In this article we use the term "checkpoint" almost synonymously with "migrate". Not all systems which implement process migration however also provide the safety of checkpointing, because they may not allow "migrating" a process into a file. Condor provides access to files across machines, even in environments where files are not generally available by NFS or AFS, with a mechanism called "remote system calls". These are discussed in section 4.1.

Our general method for checkpointing is to write all of the process's state information to a file (or socket) at checkpoint time, then use the information in that file (or read from a socket) to restore the process's state (as much as possible) at restart time. From the point of view of the operating system, the process is not restarted or migrated at all - we simply create a new process, and manipulate its state so as to emulate the state of the old process as accurately as possible. The checkpointing process is invoked by a signal, and at restart time, things are manipulated so that it appears to the user code that the process has just returned from that signal handler. The code contained in the signal handler, the code required to install the handler, and the code used to record information about the process's state as it evolves, are all contained in the Condor checkpointing library.

3 Components of a Process

To checkpoint and restart a process, one must consider all the components which make up a process's state. UNIX processes consist generally of an address space, (generally divided into text, data, and stack), and "other" information about the process which is maintained by the kernel. The state of the process's registers, any special handling requested for various signals, and the status of open files fall into this category.

3.1 Text and Data Areas

Processes which are statically linked are born with their entire text loaded into virtual memory by the kernel, generally beginning at address 0. Since we use exactly the same executable file for both the original invocation and when restarting a process, we don't have to do anything special to save and restore the text. (Note that modern programming practice requires that text be loaded "read-only", so there is no chance that the text will be modified at run time.)

The "data" space of a UNIX process generally consists of 3 areas - initialized data, uninitialized data, and the heap. Initialized data is that data given values by the programmer at compile time. Uninitialized data is space allocated at compile time, but not given values by the programmer (the kernel will zero fill this area at load time). The heap is data allocated at run time by the `brk()` or `sbrk()` system calls (these are the system calls which underlie `malloc()`). A process's data generally begins at some pagesize boundary above the text, and is a contiguous area of memory. That is, the initialized data begins at the first pagesize boundary above the text, the uninitialized data comes next, followed by the heap which grows toward higher addresses at run time. Note that once the process begins execution, the initialized data may be overwritten, and thus at restart time, we cannot depend on the information in the executable file for this area. Instead the entire data segment is written to the checkpoint file at checkpoint time, and read into the address space at restart time. All one needs to know to accomplish this are the starting and ending addresses of the data segment.

3.2 Stack

The stack is that part of the address space allocated at run time to accommodate the information needed by the procedure call mechanism, procedure call arguments, and automatic variables and arrays. The size of the stack varies at run time whenever a procedure is entered or exited. On some systems the stack begins at a fixed address near the top of virtual memory and grows toward lower addresses, while on others it begins at an address in the middle of virtual memory (to allow space for heap allocation), and grows toward higher numbered addresses. Special care is taken to detect if the stack resides in the data area (for example, if a user-level threads package is in use by the program) and to locate all stack frames successfully if in this case. In any case, the stack can be written to a file provided only that one knows its beginning and ending addresses. Restoring the stack requires a bit more of a trick. The problem is that if the saved stack information were to overwrite the call frame of the procedure doing the restart, not only its local variables but also all information regarding what to do at the conclusion of this procedure would be lost. To avoid this, we use a pre-defined area in the data space as a temporary stack when executing that part of the restart code which restores the stack.

3.2.1 Manipulating the Stack

A clever assembler programmer could come up with code to accomplish these "tricks", but that would require different assembler modules for each machine architecture we wanted to run Condor on. Instead we use `setjmp()` and `longjmp()` for this purpose, and reduce the machine dependency to a single macro (one line of "C" code). To see how this works, first consider the actions of setjmp/longjmp in more mundane circumstances. One calls `setjmp()` with a pointer to a system defined type called a `JMP_BUF`. The `setjmp()` routine saves the current "context" into the `JMP_BUF`, and returns 0. The content of the `JMP_BUF` is generally considered to be opaque to the programmer, but it does contain all the information needed to return to the current point in the stack from any other procedure which is called by this one - possibly after many nested procedure calls. If somewhere in the called procedure `longjmp()` is called with a pointer to the `JMP_BUF` and some value other than 0, then we return to the point where the original `setjmp()` call was made. This time the return value is the one specified at the `longjmp()` call, i.e. something other than 0. Of course one of the contents of the `JMP_BUF` is the stack pointer as of the time of the `setjmp()` call. The single line of machine specific code mentioned earlier is a macro which places a value into the location corresponding to the stack pointer within a `JMP_BUF`.

To call the procedure `restore_stack()` we do a `setjmp()`, use the macro to switch the stack pointer stored in the `JMP_BUF` to a location in the data area, and then execute the corresponding `longjmp()`. It should be noted that this idea was not invented by us, and is in fact used for switching virtual stacks by a number of popular user-level threads packages. The return from the `restore_stack()` routine is similar. In this case the call to `longjmp()` uses a `JMP_BUF` which was saved in the data area at checkpoint time. Recall that the data area is restored before the stack, so the content of the `JMP_BUF` is valid at this time.

3.3 Shared Libraries

Until recently, Condor had required that jobs with checkpointing support be linked statically. However, vendors have begun releasing UNIX systems without full support for static linking. For example, only dynamic versions of some or all libraries are provided.

Processes on current UNIX systems may contain mapped segments in their address space in addition to traditional text, data, and stack segments. This mapped segment facility is used to support dynamically-linked, shared-text libraries. When a dynamically-linked process is created, the necessary libraries are mapped into the process's address space. The first time a specific library call is made, a fault occurs and a link is created between the text symbol and the function definition in the library. Processes may also load in new libraries at any point during their execution. There is no guarantee that libraries will be mapped into the same addresses for multiple executions of the same program.

Condor must checkpoint the dynamic library data of the running process, in addition to the stack and data segments. The system must also ensure on restart that each dynamic library is mapped into the same area of virtual memory where it lived before the checkpoint, so that the dynamic links that have been set up are still valid. To do this, Condor must have support from the operating system to find all active segments, read the data in these segments, create new segments at arbitrary addresses, and write (restore) data into those new segments.

3.3.1 Checkpointing Mapped Segments

To find all active segments, the checkpoint library uses the `ioctl()` interface to the `/proc` file system, available on a number of UNIX variants. In the `/proc` directory, there is a file for each running process, named by process-ID, which gives access to the process's memory contents. There is an `ioctl()` call to find the number of mapped memory segments in use by a given process and another to find information about each segment (virtual start address, size of segment in bytes, protection and attribute flags). Note that this call returns all process segments, including stack, data, and text segments.

On systems where the `/proc` file system is not available or does not provide the needed interface, the library could instead record the needed information at the time of `mmap()` calls. The `mmap()` function is used by the dynamic linker to create new segments and map shared libraries into the segments. Our method for augmenting system calls to record needed information is described below in section 3.4.1. Dynamic library support is currently only provided on systems which have the needed `/proc` interface.

Once the segment information is obtained, Condor must determine which segments are which, because some of the segments must be treated specially. The data and text segments are identified by comparing the addresses to the address of a static function (in the Condor checkpoint library) and the address of a global variable, respectively. The stack segment(s) are identified by comparing the addresses to the stack pointer value and a system defined constant for the stack ending address. All other segments are assumed to contain dynamic library text or data.

Once all segments are identified, Condor saves all segments except the static text segment (because the text can be retrieved from the `a.out` file at restart). The `write()` system call is used to write bytes from addresses in memory to the file. Note that Condor saves the dynamic library text in addition to dynamic library data. There are a number of reasons for doing this. First, this is a simple way for Condor to ensure that the library text matches the library data on restart when migrating to a new machine which may have different versions of system libraries. If Condor did not ensure this, it is possible that library text would look for variables in incorrect locations. A second reason is that Condor must ensure that the library text is mapped back into the same location on restart, so that dynamic links are still valid. This behavior negates one of the benefits of dynamic linking (smaller executables) by increasing the size of the checkpoint file. However, we felt that this cost was acceptable compared to the complexity of comparing library versions on the checkpoint and restart machines and only moving libraries when necessary. It should at least be no worse than static checkpointing.

3.3.2 Restoring Mapped Segments

On restart, Condor must restore all segments in the checkpoint file. The data segment is restored as in static checkpointing. Condor restores the stack segment last, and returns to the top of the stack as if returning from a signal handler.

To restore the dynamic libraries, Condor first uses the `mmap()` function to allocate the necessary mapped segment in the virtual address space. This call maps the file /dev/zero[1] from the appropriate start address to end address, to provide an initialized segment. The protection and attribute flags for this new segment are set to be those saved in the checkpoint file, except that the segment always has write access enabled and the memory is always marked as private. Write access is necessary so that the saved bytes in the checkpoint file can be written to the segment. The memory is not shared to ensure that our (possibly different) version of a system library does not interfere with other processes on the machine. Once the mapped segment is allocated, the `read()` system call is used to overwrite the memory with the saved bytes in the checkpoint file.

Note that the definitions for `mmap()` and `write()` must be forced to be static. Otherwise, we run into the possibility of overwriting the shared library text segment in which they are defined while we are in a call to either `mmap()` or `write()`, which can have disastrous consequences.

3.4 Files

Any files which are held open by a process at checkpoint time should be re-opened with the same "attributes" at restart time. The attributes of an open file include its file descriptor number, the mode in which it is opened, (e.g. read, write, or read-write), the offset to which it is positioned, and whether or not it is a duplicate of another file descriptor. Since much of this information is not made available to user code by the kernel, we record several attributes at the time the file descriptor is created via an `open()` or `dup()` system call. Information recorded includes the pathname of the file, the file descriptor number, the mode, and (if it is a dup) the base file descriptor number. The offset to which the file is positioned is captured at checkpoint time by use of the `lseek()` system call. All of this information is kept in a table in the process's data space, which is of course restored early in the restart process. Later in the restart process, we walk through the table and re-open and re-position all the files as they were at checkpoint time. Of course an important part of a file's state is its content - we assume that that this is stored safely in the file system, and that nobody tampers with it between checkpoint and restart times.

3.4.1 Augmenting System Calls

As described above, we need to record information from system calls such as `open()` without any modification of the user code. We do this by providing our own version of `open()` which records the information then calls the system provided `open()` routine. A straightforward implementation of this would result in a naming conflict, i.e. our `open()` routine would cover up the system `open()` routine.

The UNIX man pages make a distinction between "system calls" and "C library routines" (system calls are described in section 2, and library routines are described in section 3). However, from the programmer's point of view, these two items appear to be very similar. There may seem to be no fundamental difference between a call to `write()` and a call to `printf()` - each is simply a procedure call requesting some service provided by "the system". To see the difference, consider the plight of a programmer who wants to alter the functionality of each of these calls, but doesn't want to change their names. Keeping the names the same will be crucial if one wants to link the altered `write()` and `printf()` routines with existing code which should not be aware of the change. The programmer wanting to change `printf()` has at his or her disposal all the tools and routines available to the original designer of printf, but the programmer wanting to change `write()` has a problem. How can one get the kernel to transfer data to the disk without calling `write()`?

[1] /dev/zero is a special UNIX file which appears to contain an infinite number of zeros on reading and discards any data written to it.

We cannot call write() from within a routine called write() - that would be recursion, and definitely not what is wanted here. The solution is a little known routine called syscall(). Every UNIX system call is associated with a number (defined by a macro in <syscall.h>). One can replace an invocation of a system call with a call to the syscall routine. In this case the first argument is the system call number, and the remaining arguments are just the normal arguments to the system call. The following write() routine simply counts the number of times write() was called in the program, but otherwise acts exactly like the normal write().

```
int number_of_writes = 0;
write( int fd, void *buf, size_t len )
{
number_of_writes++;
return syscall( SYS_write, fd, buf, len );
}
```

Interestingly, this trick works even if the user code never calls write() directly, but only indirectly - for example via printf(). The condor checkpointing code uses this mechanism to augment the functionality of a number of system calls. For example, we augment the open() system call so that it records the name of the file being opened, and the file descriptor number returned. This information is later used to re-open the file at restart time.

3.5 Signals

An interesting part of a process's state is its collection of signal handling attributes. In UNIX processes, signals may be blocked, ignored, take default action, or invoke a programmer defined signal handler. At checkpoint time a table is built, again in the process's data segment, which records the handling status for each possible signal. The set of blocked signals is obtained from the sigprocmask() system call, and the handling of individual signals is obtained from the sigaction() system call. An interesting situation results if a a signal has been sent to a process, but that process has the signal blocked. Such a signal is said to be "pending". If a signal is pending at checkpoint time, the same situation must be re-created at restart time. We determine the set of pending signals with the sigispending() system call. At restart time the process will first block each pending signal, then send itself an instance of each such signal. This ensures that if the user code later unblocks the signal, it will be delivered.

3.6 Processor State

Saving and restoring the state of a process's CPU is potentially the most machine dependent part of the checkpointing code. Various processors have different numbers of integer and floating point registers, special purpose registers, floating point hardware, instruction queues, and so on. One might think that it would be necessary to build an assembler code module for each CPU to accomplish this task, but we have discovered that mechanisms already available within the UNIX system call set can be leveraged to do this work with (in most cases) no assembler code at all. This is the reason that a checkpoint is always invoked by sending the process a signal. A characteristic of the signal mechanism is that the signal handler could do anything at all with the CPU, but when it returns, the interrupted user code should continue on without error. This means that the signal handling mechanism provided by the system saves and restores all the CPU state we need.

4 Summary of Checkpoint and Restart

Now that we've seen some of the details, we can look at the "big picture" of how checkpoint/restart is accomplished. When our original process is born, condor code installs a handler for the checkpointing signal, initializes its data structures, then calls the user's `main()` routine. At some arbitrary point during execution of the user code, the checkpoint signal is delivered invoking the `checkpoint()` routine. This routine records information about open files and our current location on the stack in the data area, writes the data, shared library, and stack areas into a checkpoint file, then the process either immediately exits or returns from the signal handler and continues processing (depending on whether the checkpoint signal was a *vacate* signal or a *periodic* checkpoint signal). At restart time condor executes the same program with a special set of arguments. The special arguments cause the `restart()` routine to be called instead of the user's `main()`. The `restart()` routine overwrites its own data segment with that stored in the checkpoint file. Now it has the list of open files, signal handlers, etc. in its own data space, and restores those parts of the state. Next, it switches its stack to a temporary location in the data space, and overwrites its own process's stack with that saved in the checkpoint file. The `restart()` routine then returns to the stack location that was current at the time of the `checkpoint()`, i.e. the `restart()` routine returns to the `checkpoint()` routine. Now the `checkpoint()` routine returns, but recall that this routine is a signal handler - it therefore returns to the user code that was interrupted by the checkpoint signal. The user code resumes from where it left off, and is none the wiser.

4.1 Remote File Access

Process migration requires that a process can access the same set of files from different machines. While this functionality is provided in many environments via a networked file system, it is often desirable to share computing resources between machines which don't have a common filesystem. For example, we have migrated processes between our site at the University of Wisconsin, and several sites in Europe and Russia. Since these sites certainly do not share a common file system, condor provides its own means of location independent file access. This is done by maintaining a process (called the "shadow") on the machine where the job was submitted which acts as an agent for file access by the migrated process. All calls to system routines that use file descriptors by the user's code are re-routed by RPCs to the shadow. We do this at the system call level, so that whether the user code uses `write()` directly, or calls `printf()` (or even some other routine we've never heard of which ultimately exercises the `write()` system call), correct action results.

4.2 Limitations

While the designers of truly distributed operating systems such as Sprite and V Kernel have carefully defined and implemented their process models to accommodate migration, we users of UNIX are not so fortunate. There are a lot of details in a process's state which are implicit, known only to the kernel, or are otherwise difficult or impossible to re-create. In condor we have taken the viewpoint that we can save and restore enough of a process's state to accommodate the needs of a wide variety of real-world user code. There is however, no way we can save all the state necessary for every kind of process. The most glaring lack of course is our inability to migrate one or more members of a set of communicating processes. In fact no attempt is made to deal with processes which execute `fork()` or `exec()`, or communicate with other processes via signals, sockets, pipes, files, or any other means. This is not to say that some inventive users have not found ways to use Condor for communicating processes, but they have changed their code to accommodate our limitations.

Another major limitation is the fact the the condor checkpointing code must be linked in with the user's code. This is fine for folks who build and run their own software, but it doesn't work for users of third party software who don't have access to the source. We have considered schemes to provide a "checkpointing C library" for third party programs which are dynamically linked, but so far have not implemented anything. A major obstacle to such work is the fact that shared library implementations vary widely across platforms, and such a facility would not be very portable.

5 Conclusion

In this report, we have described Condor's user-level checkpoint and migration service for UNIX processes. This service was developed with the goals of portability and user transparency in mind, and it has been used successfully to enhance the flexibility of the Condor scheduler and provide forward progress for long running Condor jobs even when machines are available for limited periods of time. It has also been used as a standalone product (i.e., outside of the Condor resource management system) and incorporated into Load Leveler, Load Sharing Facility, CoCheck, Codine and other products.