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

Over the course of many years, the University of Wisconsin-Madison has developed a powerful distributed batch processing system for UNIX called Condor. Condor allows organizations to utilize otherwise idle CPU cycles in a pool of UNIX workstations. A Condor *pool* is a cluster of workstations connected via a network that are watched over by the Condor *central manager* [refer to Fig 1]. Users can submit jobs to Condor from any workstation in the pool. Condor will then find an idle workstation in the pool and run the job there until someone starts using that workstation again. Condor ensures that the file system and other machine characteristics on a *remote execution machine* appear to be identical to those of the job's originating *submitting machine* through a mechanism called *remote system calls*. Remote system calls allow Condor to provide necessary file access for the job even in environments where files are not generally available by more common file sharing mechanism such as NFS or AFS. When Condor detects activity on a workstation upon which it is currently running a job, Condor creates a *checkpoint* of the job before killing it. This checkpoint is written to disk and contains all of the process' state information necessary for Condor to restart the job exactly where it left off. Once Condor has created the checkpoint, it keeps it queued on disk until another workstation in the pool becomes idle and thus available. Once this happens, Condor transfers the checkpoint to this new workstation and restarts the job right where it left off, effectively *migrating* the process from one available workstation to another.

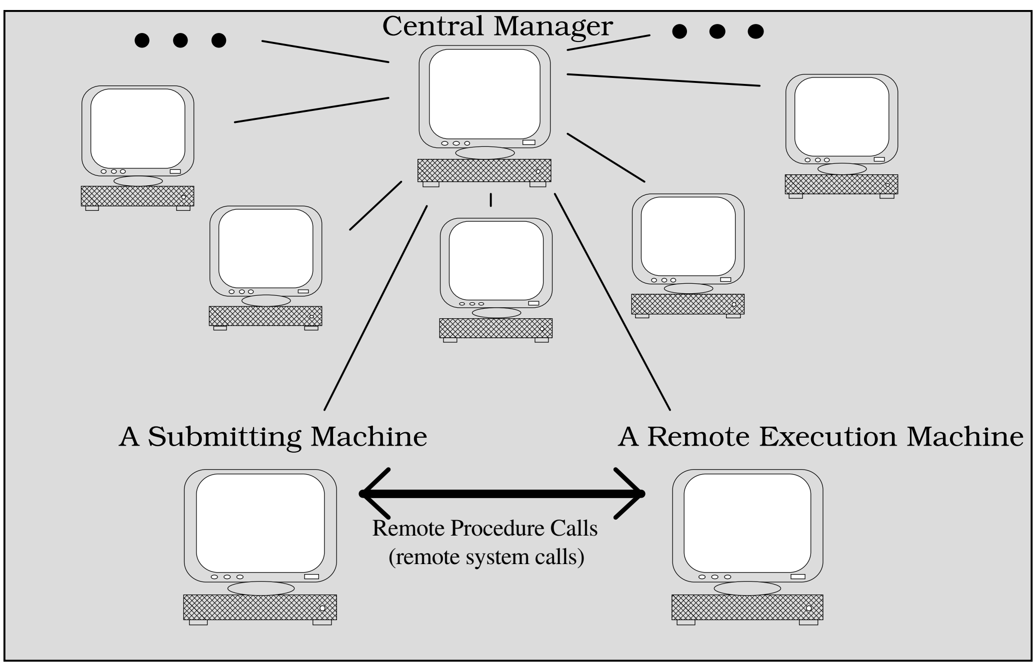

Figure #1

A complete discussion of Condor is beyond the scope of this article. Instead, we will focus on exactly how Condor transparently implements checkpoint/restart of a UNIX process, and how remote procedure calls transparently provide a consistent file system environment suitable for process migration.

The Basic Condor Framework

One of the major components of Condor is its facility for transparently checkpointing and subsequently 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.

Condor supports checkpoint and migration of most user programs without forcing the user to change a single line of their source code. However, checkpoint/migration support does require the user to link their binary with the Condor Checkpointing Library (the Condor system includes a utility that will do this with one command). This Checkpointing Library first installs a signal handler which contains the code to asynchronously checkpoint the process. Next, the Condor Checkpointing Library augments a wide array of UNIX system calls to support checkpoint/restart as well as migration.

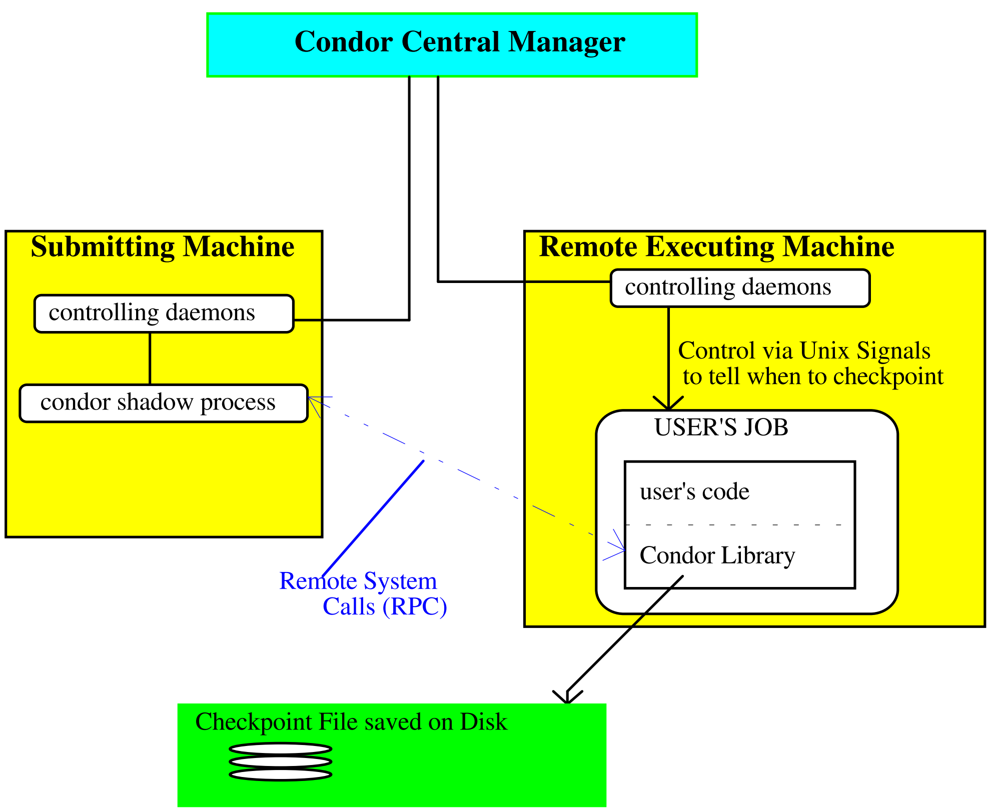

Figure #2

Condor controlling daemons (background UNIX programs) continuously monitor system activity. When Condor notices external system activity (such as someone typing on the keyboard) on a machine which is running a Condor user job, the daemon will send the job a "checkpoint" signal (see Figure 2). This signal will invoke code in the Checkpointing Library which will write out a checkpoint file to disk and then terminate the process. The checkpoint file is transferred back to the machine from where the Condor job was originally submitted. When the Condor central manager locates newly available idle machine in the pool, the job's original binary executable as well as the most recent checkpoint file for that job is transferred to the machine. Now when the user's job is executed, the Condor Checkpointing Library (which is linked in with the job's binary) will know that this is not the first time this process has run. Thus, it will perform a restart by reading in the accompanying checkpoint file and manipulating its state so as to emulate as accurately as possible the state at checkpoint time of the old process. The checkpointing process was invoked by a signal, and now at restart time, things are manipulated so that it appears to the user code that the process has just returned from that signal handler.

Inside the Checkpoint Library: How Condor Saves & Restores Each Component of a Process

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

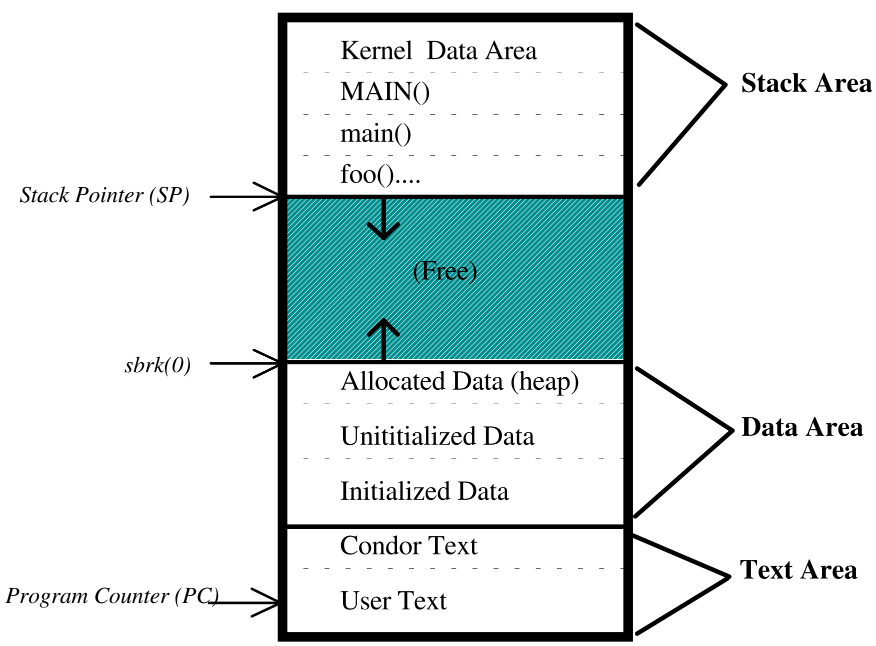

Figure #3

Text and Data Areas

Statically linked UNIX processes 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 a restarting a process, we do not have to do anything special to save and restore the text area. (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 and/or uninitialized data contain global and statically declared variables. 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`(), which are UNIX system calls typically used by the C function `malloc`(). A process' 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 exactable file for this area. Instead the entire data segment is written to the checkpoint file at checkpoint time, and read back into the same address space at restart time. All one needs to know to accomplish this are the starting and ending addresses of the data segment. The starting address is platform specific, but is usually a static value which can be found as a linker directive or, on some versions of UNIX, in the man pages. The ending address of the data segment is effectively the top of the heap. This address can be obtained within UNIX via the `sbrk()` system call. Condor restores the data space early in the restart process.

Stack

The stack area 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. When porting Condor to a new UNIX platform, Condor must be told in which direction the stack grows.

Preserving the state of the stack requires saving and restoring two distinct pieces of information. First of all, the *stack context* (sometimes referred to as the *stack environment*) must be saved. The stack context is a small collection of stack-related information which most notably contains the *stack pointer* (which keeps track of a process' current position within the stack data area). Second, the actual data which makes up the stack itself (referred to as the *stack space*) must be saved.

To save and restore the stack context, Condor uses the standard C functions `setjmp()` and `longjmp()`. One calls `setjmp()` with a pointer to a system defined type called a JMP_BUF. `setjmp()` saves the current stack context into the JMP_BUF, and returns 0. If `longjmp()` is then called with a pointer to the JMP_BUF and some value other than 0, the stack context saved in JMP_BUF is restored and we return back in the code to the point where the original `setjmp()` call was made. This time the return value from `setjmp()` is the one specified in the `longjmp()` call, i.e. something other than 0.

However, a limitation of `setjmp()`/`longjmp()` is that the JMP_BUF does not contain the actual data contained in the stack space itself, only pointers into the stack space. The developers of `setjmp()`/`longjmp()` designed it to work within the life-span of a single process. However, when Condor restarts a checkpointed process, it first creates a new process, and then manipulates its state so as to emulate the state of the old process. This new process will have its own new stack space. Therefore, Condor additionally needs to save the actual contents of the stack space at checkpoint time. At restart time, Condor needs to replace the new processes' stack space with the checkpointed stack space before utilizing `longjmp()` to reset the stack context back to what it was at checkpoint time.

Saving the contents of the stack space into the checkpoint file is trivial. Again, all we need to know is the stack's start and end points. The start of the stack is a well known static location which is defined as a constant on some platforms, or can be obtained from the man pages on others. The end of the stack, by definition, is pointed to by the stack pointer. Thus, to determine the end of the stack, Condor does a `setjmp()` and pulls the stack pointer value out of the JMP_BUF.

Restoring the stack is more tricky. Unlike restoring the data area, we would like to be able to use the stack space (for local variables) while we are replacing it! Directly replacing a processes' stack space with the space saved in the checkpoint file is a sure way to send the process off to never-never land! To avoid this, Condor moves the stack pointer to a safe buffer reserved in the processes' data area. Moving the stack pointer is accomplished with yet another call to `setjmp()`, manually manipulating the stack pointer in the JMP_BUF to point to our buffer in the data area, followed by a `longjmp()`. Then, with the Condor stack restore procedure using a secure stack space in the data area, we can safely restore a new process' original stack space with the one previously saved in the checkpoint file.

Open 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 descirptor number, the mode, and (if it is a dup) the base file descriptor number. The offset at which each file descriptor is positioned is captured at checkpoint time by performing an `lseek`() system call upon each descriptor. All of this information is kept in a table in the process' data space, and recall that the data space is restored early in the restart process. Later in the restart process, we walk through this table and re-open and re-position all of the files as they were at checkpoint time. Of course an important part of a file's state is its content; we assume that this is stored safely in the file system, and that nobody tampered with it between checkpoint and restart times.

An interesting point is the method used to record information from a system call such as `open`() without the need for any modification of the user code. We do this by providing our own version of `open`() and `close`() which records the information in the table, then calls the original system-provided `open`()/`close`() routines. A straightforward implementation of this would result in a naming conflict, i.e. our augmented `open`() routine would cover up the system `open`() routine. To learn how we avoid the naming conflict see the sidebar "Augmenting UNIX System Calls".

Signals

An easily overlooked part of a process' 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' 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 each individual signal is obtained from the `sigaction`() system call. During restart, Condor restores signal state by stepping through the table. An interesting situation results if a signal had been sent to a process while 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. To handle pending signals, we determine the set of pending signals with the `sigispending`() system call at checkpoint time. During restart, the Condor library code will first block each pending signal, then send itself an instance of each such pending signal. This ensures that if the user code later unblocks the signal, it will be delivered.

CPU State

Saving and restoring the state of a process' 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 the signal mechanism 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 UNIX 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.

Pulling it all Together: A Complete Checkpoint and Restart Cycle

Now that we have seen some of the details, we can look at the "big picture" of how a Condor checkpoint/restart is accomplished (see Listing #1 which contains selected functions from Condor that illustrate this). When our original process is born, Condor code installs a signal handler for the checkpointing signal, initializes its data structures, and then calls the user's `main()` routine. At some arbitrary point during execution of the user code, the process will receive a checkpoint signal which invokes the `checkpoint()` routine. This routine records information about the stack context, signal state, and open files into data structures in the process' data area. Then, it writes the data and stack spaces into a checkpoint file, and the process exits. At restart time, Condor executes the same program with a special set of arguments. The special arguments cause the `restore()` routine to be called instead of the user's `main()`. The `restore()` 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' stack with that saved in the checkpoint file. The `restore()` 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 restores all CPU registers and 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.

Location Independent File System Access via Remote System Calls

Process migration requires that a process can access the same set of files in a consistent fashion from different machines. While this functionality is provided in many environments via a networked file system (NFS, for example), it is often desirable to share computing resources between machines which do not have a common filesystem. For example, we have migrated processes between our site at the University of Wisconsin-Madison, 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 a "shadow") on the machine where the job was submitted which acts as an agent for file access by the migrated process (Fig #2). All calls to system routines that use file descriptors by the user's code are augmented by the Condor Checkpoint Library so that they are re-routed via Remote Procedure Calls (RPCs) to the shadow. (See the sidebar Augmenting System Calls.) The shadow process then executes the system call and passes back the result to the Condor Checkpoint Library, which passes back the result to the user code. We do this re-direction for functions at the system call level, so that whether the user code uses `write()` directly, or calls `printf()`, or even some other routine we have never heard of which ultimately exercises the `write()` system call somewhere along the line, correct action results.

Limitations

While the designers of truly distributed operating systems such as sprite and vkernel have carefully defined and implemented their process models to accommodate migration, we users of UNIX are not so fortunate. In Condor we have taken the viewpoint that we can save and restore enough of a process' 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 deficiency 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 which 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 were forced to change their code to accommodate our limitations. Another major limitation is the fact the Condor checkpointing library must be linked in with the user's code. This is fine for folks who build and run their own software, but it does not work for users of third party software who do not 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.

Availability

Work on Condor is continuous and on-going, and it is available for free. Both binary-only distributions as well as distributions with complete source code are available for many different UNIX platforms via anonymous FTP over the Internet to ftp.cs.wisc.edu in the /condor directory. << Jon: Please insert Dr Dobb BBS/Compuserve info, etc, here>>

About the Authors:
***Todd** is the Director of the Model Advanced Facility, which pilots leading-edge computing technology into the College of Engineering at the University of Wisconsin-Madison. **Michael** is a researcher in the Computer Science Department at the University of Wisconsin-Madison, and is the primary author of Condor. For correspondence relating to Condor, email condor@cs.wisc.edu.*

Sidebar - Augmenting UNIX System Calls

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()` ? 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. For instance, 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 via standard C library calls - 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. -- *Michael Litzkow*

Listing #1

```
/***********************************************************
  This is the signal handler which actually effects a checkpoint.  This function
  must be previously installed as a signal handler, since we assume the signal
  handling code provided by the system will save and restore important
  elements of our context (register values, etc.).  A process wishing
  to checkpoint itself should generate the correct signal, not directly call this function.
*/
void
Checkpoint( int sig, int code, void *scp )
{

	if( SETJMP(Env) == 0 ) {	// Save place here
		// dprintf() will log messages into condor system admin log files
		dprintf( D_ALWAYS, "About to save MyImage\n" );

		// This routine will step through our File state table and fills in
		// information, for instance lseek all file descriptors to fill in where
		// the file pointer for each open file is
		SaveFileState();

		// Here we fill in our Signal state table (i.e. which signals are pending, etc)
		SaveSignalState();

		// These will now write out our data & stack into the checkpoint file
		MyImage.Save();
		MyImage.Write();

		dprintf( D_ALWAYS,  "Ckpt exit\n");

		// now that we have saved all of our state, we terminate this process
		// we terminate with a signal so that the condor controlling daemons
		// (of whom one is our parent) will know that we exited after a checkpoint,
		// as opposed to the user code exiting on its own
		terminate_with_sig( SIGUSR2 );	// this exits

	} else {

		// We get here from the longjmp in RestoreStack() during restart!

		// Patch registers handles any messy CPU register business which
		// is not handled by UNIX itself as part of a signal handler.  Note that
		// on all platforms except HPUX, this is a null procedure
		patch_registers( scp );
```

```
		// Close the checkpoint file, etc,  before we re-open all files
		MyImage.Close();

		// Re-open and lseek back all previously opened files, then
		// re-install any user signal handlers and block/resend any pending
		// signals.
		RestoreFileState();
		RestoreSignalState();

		return;	// here we go back to user code (end of signal handler)
	}
}

/****************************************************************
  Given an "image" object containing checkpoint information which we have
  just read in from disk, this method actually effects the restart.
*/
void
Image::Restore()
{
	int		save_fd = fd;
	int		user_data;

		// Overwrite our data segment with the one saved at checkpoint
		// time.
	RestoreSeg( "DATA" );

		// We have just overwritten our data segment, so the image
		// we are working with has been overwritten too.  Fortunately,
		// the only thing that has changed is the file descriptor, which we also
		// saved on the stack above at the start of this function.
	fd = save_fd;

		// Now we're going to restore the stack, so we move our execution
		// stack to a temporary area (in the data segment), then call
		// the RestoreStack() routine.
	ExecuteOnTmpStk( RestoreStack );

		// RestoreStack() also does the jump back to user code
	fprintf( stderr, "Error, should never get here\n" );
	exit( 1 );
}

void
```

```
RestoreStack()
{
	// This function rewrites the stack data area.  Thus, we are called from
	// ExecuteOnTmpStk() which has repositioned the stack pointer into a safe
	// temporary chunk of memory in the data area

	// First, call our routine to restore the stack data from the checkpoint file
	MyImage.RestoreSeg( "STACK" );

	// Now, restore the stack context, i.e. put the stack pointer back where
	// it was at checkpoint time.  Do this via a LONGJMP using a JMP_BUF
	// we created at checkpoint time in the Checkpoint() routine.
	// Note this will move execution back to the else clause in Checkpoint() routine!!
	LONGJMP( Env, 1 );
}

static void (*SaveFunc)();
static jmp_buf Env;

const int	TmpStackSize = 4096;
static char	TmpStack[ TmpStackSize ];   // this buffer will end up in the data area

/*
  Execute the given function on a temporary stack in the
  data area.
*/
void
ExecuteOnTmpStk( void (*func)() )
{
	jmp_buf		env;
	SaveFunc = func;  // save in global, - we're going to lose our stack frame

	if( SETJMP(env) == 0 ) {
			// First time through - move SP
		if( StackGrowsDown() ) {
			JMP_BUF_SP(env) = (long)TmpStack + TmpStackSize;
		} else {
			JMP_BUF_SP(env) = (long)TmpStack;
		}
		LONGJMP( env, 1 );
	} else {
			// Second time through - call the function
		SaveFunc();
	}
}
```