A PROCESS MIGRATION IMPLEMENTATION FOR A UNIX SYSTEM

Rafael Alonso
Kriton Kyrimis

CS-TR-092-87

May 1987

A Process Migration Implementation for a Unix System

Rafael Alonso
Kriton Kyrimis

Department of Computer Science
Princeton University

ABSTRACT

In this paper we describe the implementation of a process migration mechanism under version 3.0 of the Sun UNIX† operating system. Processes that do not communicate with other processes and that do not take actions that depend on knowledge of the execution environment (such as the process id), can be moved from one machine to another while running, in a transparent way. This is achieved by signaling a process to stop, saving all the kernel and memory information that is necessary to restart the process and then, by using this information, restarting the process on the new machine. This new functionality requires minor kernel modifications as well as the creation of a new signal and a new system call.

† UNIX is a trademark of AT&T Bell Laboratories.

A Process Migration Implementation for a Unix System

Rafael Alonso
Kriton Kyrimis

Department of Computer Science
Princeton University

1. Introduction

Process migration is the capability to move a process that is running on a certain machine to another, without interrupting its execution. This is a useful tool to have, as it can be used in various ways, ranging from system applications such as load balancing and process checkpointing, to applications for individual users such as moving a process from a machine that is about to go down, to another.

The interface to this mechanism should be transparent to the process that is being migrated. Specifically, the process should not know anything about the process migration mechanism and, after it is moved to another host, it should continue its execution as if it were still running on the original machine. The performance of this mechanism may vary depending on the purpose for which it is used. If it is used to even the load among a number of machines (load balancing), then it must introduce little overhead to the system, and must be able to move processes between machines quickly, requiring time comparable to that of the time required to load and start a program. On the other hand, if process migration is used for moving individual CPU intensive tasks in order to achieve better performance for those tasks, or to remove important tasks from a machine that is about to be halted, then it is acceptable for the mechanism to have a lower performance.

Implementing process migration in an operating system is a non—trivial task. One must be able to keep track of all the system resources that a process is using and to reallocate them to the migrated process on another machine. For example, one must know what files a process is using and have the ability to reopen these files on behalf of the migrated process on another machine, keeping track of the current offset within the file. If files are local to the machine that owns the device on which they are stored, this can prove to be very hard, if not impossible, as in many cases the file system cannot access the files of a remote machine directly.

In this paper we start by mentioning previous implementations of process migration and describing our implementation environment, explaining how it is different from that of these other implementations. Then, in Section 4, we present the user interface to the process migration mechanism and provide examples for it. First we present the interface as a casual user might see it and then as a programmer who wants to write new applications that use this mechanism. In the Section 5 we describe the additions and modifications that were made to the Sun 3.0 UNIX kernel to add the process migration capability to it. In Section 6 we present our measurements of the performance of the new kernel and the applications that were written on top of it and in Section 7 we discuss the limitations of our system. In the eighth section we describe some applications that can be implemented using our process migration mechanism and, in the last section, we present our conclusions.

2. Other implementations

There are only four other implementations of process migration of which we are currently aware:

In the DEMOS/MP operating system[1] all interaction with the kernel is achieved by exchanging messages with it. This extends to the kernel itself, which can use the message mechanism to communicate with the kernel of another machine. Process migration is achieved by having the original kernel transfer the state of the process to be migrated, to the destination kernel. After the process is created on the new host, all pending messages are forwarded to the new address of the process. Finally, the old process is destroyed and replaced with a degenerate process which acts as a forwarding address, so that further messages to the old process can be delivered correctly.

The Locus distributed UNIX system[2] attempts to distribute all the resources that a program is using, including the CPU, among all the machines in a network, in order to achieve network transparency of all resources. This system provides the *migrate* system call to change the execution site of a program that is already running.

The V—System[3] also provides a network transparent environment. Each machine runs a functionally identical copy of a distributed kernel that provides address spaces, processes that run in these address spaces, and interprocess communication. Address spaces are grouped into logical hosts, and processes are bound to a logical host, which allows them to have a globally unique address. Process migration is implemented with the *migrateprog* command. This command copies the state of a process to the destination machine and then repeatedly copies that part of the state that has changed since the previous copy, until relatively little information is copied. At this stage, the old process is frozen and any remaining modifications in its state are copied to the new machine. This pre—copying is made to reduce the time that a process remains frozen, thus increasing performance. The old process is then destroyed and the new process is bound to the logical host of the old process, so that the new process is indistinguishable from the old. Finally, the new process is allowed to continue. While the process is frozen, a kernel server that is executing inside the kernel sends "reply pending" packets to all processes that have sent messages to the process being migrated, so that these processes do not time out.

Finally, in the Sprite operating system,[4,5] process migration is implemented by moving a process on another machine, but having those system calls that have different effects if executed on different machines (like get time of day, get process id), executed on the original machine, by exchanging messages between the process and the kernel of the original machine. In this way, although a process may be physically located on a different machine, it is actually working under the kernel with which it started its execution.

All of these implementations have relied on special features of the operating system that create a distributed environment and make it relatively easy for two machines to cooperate in moving a process from one to the other. However, "ordinary" operating systems such ours, a UNIX implementation which evolved to its current state but was not designed for distributed use, do not have such features and implementing process migration under them is more difficult.

Since UNIX does not provide means of communication between two kernels, our implementation was somewhat limited in scope, in the sense that not all processes can be migrated. Apart from badly behaved processes (which are discussed in Section 7), the main limitation is that our mechanism does not provide for the migration of sockets, which are the main way of doing inter—process communication under UNIX. In our discussion of our implementation's limitation we argue that this does not render it useless.

3. Implementation environment

Our implementation was made on Sun 2 workstations running version 3.0 of Sun O.S., which is a derivative of the Berkeley 4.2 BSD version of the UNIX operating system. The machines were connected to each other and a file server by a 10 Mbit Ethernet,† which provided the physical medium for moving processes from one machine to another. Each workstation's local files, along with the files that reside on the file server, are available on every machine by means of Sun's Network Filesystem[6,7] (NFS). This filesystem provides the ability to place the directory structure of a physical device (mount it) on an arbitrary place in the directory structure of a machine other than the one on which the device resides. On our particular system we followed the convention of the 8th research edition of the UNIX operating system of mounting the root directory of a machine to the "n" subdirectory of the root directory of all other machines. For example, the root directory of machine *brador* can be accessed from other machines as the directory */n/brador*.

4. User level description

Our process migration system provides certain new commands that the user can use to move processes from one machine to another. These commands are implemented by using a new signal and a new system call that our kernel provides. Knowledgeable users can use these new features directly to write their own process migration commands.

In this section we start by describing the three new commands, giving a typical example of how they can be used. These commands are easy for the naïve user to use, as all that is required to move a process from one machine to another using them is to specify the name of the destination host and the process id of the process to be migrated as command line arguments. We then

† Ethernet is a trademark of XEROX Corporation.

proceed to describe the kernel interface to the process migration mechanism, examining in some detail the implementation of two of these commands as an example.

4.1. User commands

Most of the implementation code for process migration is at the user level. By this we mean that all commands that have to do with process migration are user applications. However, if the available commands do not fit a specific need, users can easily write their own substitutes (see Section 4.3).

We have provided the following three commands, which should cover most of the common cases:

- *Dumpproc* — terminate a process (kill it) dumping to disk all the information that is necessary to restart it. The process is determined by specifying its process id with a command line option. For security reasons, only the superuser or the owner of the process can kill a process in this way.
- *Restart* — restart a process that was killed on some host with the *dumpproc* command. There are command line options to specify the process by its process id, and the name of the host on which the process was dumped (the default is the current machine). The process will be restarted on the host on which the command was given and at the terminal (or window) on which the command was typed. All files that had been open when the process was dumped will be available to the restarted process with the correct access modes and offset. Terminal modes such as "raw" (process input characters as soon as they are typed) or "noecho" (disable echoing of input characters) are preserved, so that visual applications such as screen editors can be restarted properly. Again, for security reasons, only the superuser or the owner of the original process can restart a process.
- *Migrate* — move a process from one machine to another. This is simply a combination of the two previous commands and can be used to avoid having to go to another terminal to type the *dumpproc* or restart command. The process id of the process to be moved, the name of the host from which the process is to be moved and the name of the destination host can be specified by the appropriate command line options (the default for both hosts is the current machine). The process is restarted on the terminal (or window) that the command was typed, even though the host that the process will run may not be the one on which that terminal is connected. *Migrate* calls *dumpproc* and *restart* internally, by using the remote shell command *rsh*[8] of the Sun 3.0 operating system, if necessary. Because of the way that *rsh* is implemented, certain terminal modes can not be preserved when moving a process to a remote host, thus, in these cases, making this command unsuitable for the migration of visually oriented programs. For example, a screen editor will not be able to act on keyboard input as soon as it is typed, thus the process will become useless.

4.2. An example

In this section we provide an example of a typical user interaction with the process migration mechanism. Thus, suppose that we are running a program on a machine called *brick* and that we wish to move it to a machine called *schooner*. This can be done in one of two ways. First, we must determine its process id, which can be easily done by using the UNIX *ps*[8] command. Let us assume that we have determined that the process id of the program we want to migrate is 1234. Now, we can either:

1. Type `dumpproc -p 1234` on a terminal (or window) on brick, then type `restart -p 1234 -h brick` on a terminal on schooner, followed, if we are dumping a visually oriented program, by whatever command† will cause that program to redraw the screen.

Or,

2. Type `migrate -p 1234 -f brick -t schooner` on *any* machine, not necessarily brick or schooner. If we are migrating a visually oriented program, the best option in this case is to type the command on schooner, so that the *restart* command can be executed locally and the terminal modes are preserved. As before, if we are migrating a visually oriented program, we must redraw the screen by using the appropriate command.

4.3. Writing new applications

As we mentioned in the previous section, it is possible to write new commands that handle process migration in a different way from that of the three commands that we described. To do so, the following two items are available to the programmer.

1. A new signal+, *SIGDUMP*. When a process receives this signal (which can be sent using the UNIX *kill* system call), the process is terminated, and all the information that is necessary to restart it will be dumped to disk. This information is in the form of three files, which are placed in the */usr/tmp* directory, named *a.outXXXXX, filesXXXXX* and *stackXXXXX,* where *XXXXX* is the process id of the dumped process.

   The first file is an executable obtained by dumping the text and data segments of the process, and prepending a suitable header that will make UNIX recognise the file as an executable. This file can be executed as an ordinary program. The result of such an execution will be similar to running the original program from the beginning, except that all static variables will be initialised to the values that they had when the process was killed. This gives us, incidentally, the *undump* utility for free. (This utility creates an executable file by combining an executable and a standard UNIX core dump.)

† In UNIX, ^L in most cases.

+ Signals in UNIX are essentially software interrupts, see references[9,10]

The second file contains all the information that is not needed by the kernel to restart the process, but must be used at user level if the process is to restart successfully. This information consists of:

— a "magic number" for identification purposes (arbitrarily set to octal 445).
— the name of the host on which the process was currently running at the time it was killed.
— the absolute path name of the current working directory.
— for each entry in the open file table of the process (which has a fixed size), an indicator specifying whether the entry refers to an open socket, open file or is unused. For open files, this indicator is followed by the absolute path name of the file, the file access flags (*e.g.*, read only etc.), and the file offset. Since the process migration mechanism does not currently support sockets, no extra information is kept in the case of a socket.
— the terminal flags, specifying such things as raw mode, echo/noecho, etc.

All path names in this file have been constructed by combining the names given by the process to the kernel whenever it changed directory or opened or created a file, and resolving any references to the current or parent directories. This means that symbolic links† are not resolved and this may cause problems when trying to reopen a file when restarting the process. Consider for example a file on a machine called *classic,* named */usr/foo.* If */usr* is a symbolic link to */n/brador/usr* (i.e., */usr* is mounted via NFS to the */usr* directory of the machine named *brador),* then */usr/foo* is actually */n/brador/usr/foo.* Now, let us assume that a program opens this file and is then terminated with the *SIGDUMP* signal. When the program is restarted, this file must somehow be reopened. One way would be to prepend */n/classic* to the old name and open */n/classic/usr/foo.* Because of the symbolic link, however, this would actually be */n/classic/n/brador/usr/foo.* Unfortunately, NFS does not allow this syntax, so using this file name would not produce the desired result.

The way to solve this problem is to resolve symbolic links before files are reopened. The Sun 3.0 operating system provides the *readlink()*[11] system call, which can be used iteratively to resolve all symbolic links in a pathname.

The third file contains all the information that is required by the kernel to restart a process. This information consists of:

— A "magic number" for identification purposes (arbitrarily set to octal 444).
— The user credentials (such as user and group id).

† Symbolic links are files containing the name of another file, so that the latter can be accessed by a different (and, presumably, more convenient) name. For example, on our system, a user's home directory, which can be accessed as */u/user,* is actually a symbolic link to a directory on the file server such as, say, */n/brador/u2/user.*

— The size of the stack when the process was terminated.
— The contents of the stack.
— The contents of all the registers.
— All the information kept in the user and process structures that is related to the disposition of signals, such as which signals are being caught or ignored, which functions are handling those signals that are caught, etc.

2. A new system call, *rest_proc()*, which is used to restart a process that was terminated using the *SIGDUMP* signal. It takes two arguments, the names of the *a.outXXXXX* and *stackXXXXX* files mentioned above. Its effect is to overlay the current process with a copy of the process from which the two files are created, resuming execution from the point where the process was killed. Normally, there is no return from this system call, as the process that invokes it is destroyed. If the system call does return, this means that either the system didn't have enough resources to create the new process, or that something was wrong with the two files (they did not exist or they had an incorrect format). Readers familiar with UNIX, should note that this system call is in many ways similar to the UNIX *execve()*[11] system call, which is used in conjunction with the *fork()*[11] system call to create new processes.

   Before issuing this system call to restart a process, a program should do the following:

   - Set its real and effective user id to that of the old process using the *setreuid()*[11] system call.
   - Change its current working directory to that of the old process, using the *chdir()*[11] system call.
   - Open all files that were open when the old process was running, assigning the same file numbers that they had in that program. These files must be opened with the correct access modes and positioned at the correct offset.

4.4. Example - dumpproc and restart

In this section we show as an example of using the new signal and system call, how the *dumpproc* and *restart* programs that we described in Section 4 are implemented.

Dumpproc:

- Kills the specified process with a *SIGDUMP* signal.
- Reads in the *filesXXXXX* file.
- Resolves symbolic links for the current working directory and all open files.
- If a file name points to a terminal, it is changed to */dev/tty,* to point to the current terminal of the process that will open it.
- Otherwise, if after resolving the symbolic links, a file is found to be local to the machine on which *dumpproc* is running (*i.e,* its name does not begin with */n),* the string *"/n/<machinename>"* is prepended to its name,

where <*machinename*> is the name of that machine.

- Overwrites the modified information on the *filesXXXXX* file.

Restart:

- Verifies that the three files containing the information about the process that is to be restarted exist, and that they have the correct format by checking their magic numbers.
- Reads the old user credentials from the *stackXXXXX* file and establishes them as its own. This is the only information that it reads from this file. Everything else is read from the *filesXXXXX* file.
- Reads in the old current working directory and establishes that as its own.
- For each one of the files contained in the *filesXXXXX* file, it reads the information contained there. If it is a file, it opens it with the correct access modes and positions the file pointer to the correct offset. If the file does not exist any more, or it was a socket, or it was unused, the null device */dev/null* is opened instead, so that restarted process can find an open file where it expects one, and to preserve the order of open file numbers. In the case of standard input, output and error output, if the file cannot be reopened, the terminal is opened instead of the null device, so that the user may have some control over the restarted program.
- Closes all files that were only opened to preserve the order of the file numbers. Now the only files that are redirected to the null device are those that could not be opened and those that correspond to sockets.
- Reads in the old terminal flags and sets those of the current terminal appropriately, so that the current terminal modes are those of the original process.
- Calls *rest_proc()* to restart the old program.

5. Implementation

To make the UNIX kernel capable of supporting process migration we had to make certain modifications and additions to it. In this section we start by describing the modifications we had to make in order to make the kernel keep track of certain information that we required, and then we describe the additional features we provide, which use these modifications to implement the process migration mechanism.

5.1. Kernel Modifications

The main problem in the Sun 3.0 UNIX implementation is that the kernel does not keep enough information about a process' current working directory and open files to enable us to deduce in a non-trivial way what these files are. Instead, it keeps information about where the file is located physically on disk, in a structure called an *inode.*[12,13] To overcome this, the kernel structures were augmented to include the names of these files in the following way:

One of the most important structures in the kernel, is what in UNIX terms is known as the *user* structure, which contains all the swappable information about the process that is currently being executed. A character string of fixed size was added to this structure, which contains the full path name of the current directory. By keeping this field up to date, the kernel can now know the name of the current working directory at any time and output it when dumping the state of a process during the execution of the *SIGDUMP* signal. This field is updated as follows. After each successful call to *chdir(),* which is the UNIX system call to change the current directory, we do the following: if the argument to the *chdir()* system call is an absolute path name, it is simply copied to the user structure; if it is a relative path name, it is combined with the value of the old current working directory in the user structure and the result is copied back. This field is initialised when the first call to *chdir()* is made with an absolute path name, with the updating procedure being skipped if the field has not been yet initialised. Since such a call is made early on during the UNIX startup procedure at boot time, and new processes inherit this field from their parent, we conclude that this field is correctly maintained for all processes.

Information about open files is contained in an array of pointers to *file* structures, one for each of the maximum number of open files that is allowed. Each file structure has been augmented with a pointer to a dynamically allocated character string containing the absolute path name of the file to which it refers. Again, by keeping this field up to date, the kernel knows at any moment the names of all open files, and can produce them during processing of the *SIGDUMP* signal. Dynamically allocated strings were used instead of fixed length strings, because file structures are not swappable and there is more than one process being executed at any time with, usually, more than one open file each. If we had used fixed size strings, they would have had to be large enough to accommodate large path names, even though most path names are usually of small length. This would have led to wasting large amounts of kernel memory, which is clearly undesirable. The field containing the path name of the open file is initialised after either a successful *open()*[11] (open a file) or a *creat()*[11] (create a file and open it for output) system call. In either case, the file name is copied from the arguments of each of these system calls into the entry of the file structure that is associated with the file that is being opened. If the file name is a relative path name, its name is combined with the name of the current working directory in the user structure to create the required absolute path name.

The required memory for the string that contains the file name is obtained by calling the kernel's memory allocator. When the file is closed, using the *close()*[11] system call, this memory is released. To make sure that the pointer to the name always has a correct value (either null or a valid pointer), the subroutine which allocates new file structures has been changed to initialise this pointer to a null value.

5.2. Kernel additions

Since the kernel now keeps track of all the information that we want to have about a process in order to migrate it, implementing the *SIGDUMP* signal is simply a matter of dumping the appropriate data from the kernel structures onto disk. The code is similar to that of a signal provided by standard UNIX (*SIGQUIT*), which causes a process to terminate (dumping a subset of the information we dump for our new signal) in a file named *core*.

In standard UNIX, new processes are created by first creating an identical copy of an old process by using the *fork()* system call and then overlaying the copy with an image of a new program, obtained from an executable file that is stored on disk. This system call cannot be used as it presently exists to restart a process that was dumped with the *SIGDUMP* signal. This is because *execve()* initialises the stack and clears the registers. To overcome this, we have added the *rest_proc()* system call to our kernel. This has been built upon the *execve()* system call which we have just described. For this purpose, the *execve()* system call has been slightly modified, to check a global flag which, if set, indicates that it is called from within *rest_proc()*. In that case, instead of calculating how much initial stack to allocate for the process, based on the command line arguments and the environment, it simply allocates as many bytes as are indicated in another global variable, thus making it possible to allocate as much stack as the process that is being restarted had when it was stopped.

Using this, the *rest_proc()* system call works as follows:

— It opens the *stackXXXXX* file, checking access permissions and verifying its format by checking the magic number at the beginning.

— Reads the user credentials and the size of the stack from that file.

— Sets the global flag indicating process migration and sets the variable that indicates the desired stack size.

— Calls *execve()* to execute the *a.outXXXXX* file, with the environment set to null. (As the environment of the old process was stored in its stack, it will be automatically restored when the stack is read in.)

— Resets the variable indicating process migration, so that further calls to *execve()* will work properly.

— Sets the user credentials to those already read. The old credentials were used to execute the *a.outXXXXX* file, so that only the owner of the process or the superuser is able to do it.

— Reads in the contents of the stack and registers.

— Reads in the information on the disposition of signals, and establishes it as that of the current process.

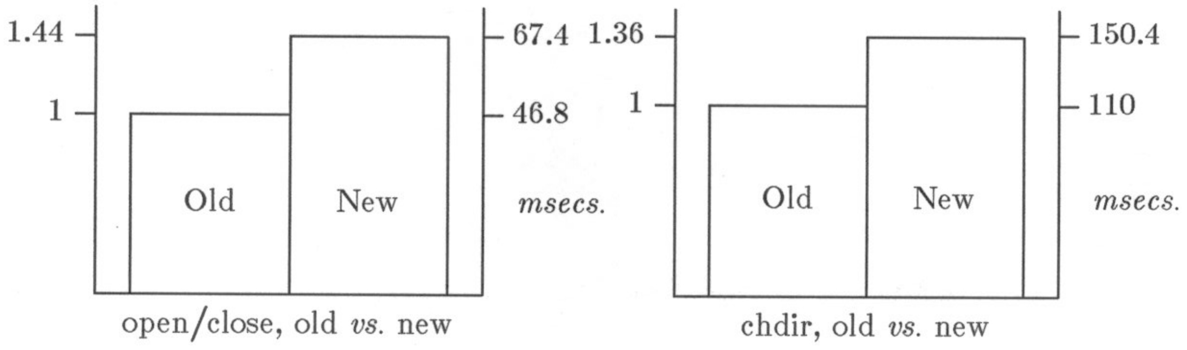
— Returns. At this point, the process running is a copy of the old process.

6. Performance evaluation

In this section we present the various measurements we made on our implementation. First, we compare the performance of the system calls we have modified to that of the unmodified ones in the original UNIX kernel. Next, we compare the performance of the new *SIGDUMP* signal and the *dumpproc* program to that of the *SIGQUIT* system of the original UNIX kernel, which is similar in function. We then compare the performance of the new system call *rest_proc()* and the *restart* program to that of the *execve()* system call, which is the closest to the new system call that the original kernel had. Finally, we compare the performance of the *migrate* application in various instances, as compared to running the *dumpproc* and *restart* applications separately.

6.1. System overhead

As we mentioned in the implementation section, the *open(), creat(), close()* and *chdir()* system calls were modified to keep track of the names of the current working directory and all open files, inside the kernel. For the *open()/close()* system calls, we gauged the overhead by measuring the system CPU execution time of a program that opens and closes a certain file for a hundred times, both under the standard UNIX kernel and under our new kernel. Since the *creat()* system call simply calls the same internal routine that *open()* calls, with slightly different arguments, we did not consider it necessary to measure its performance. For the *chdir()* system call, we measured the overhead by measuring the system CPU time of a program that executed one hundred sets of three calls to *chdir()*, one with an absolute path name as an argument, one with the parent directory ".." as an argument and one with a path relative to the current directory ".", in order to make certain that all cases of combining the new value of the current directory with the old one, kept in the user structure, were considered. The results are summarised in Figure 1, with the performance of the original UNIX kernel normalised to 1 shown on the left vertical axis and the actual times (average for one of *open()/close()* pairs or set of three *chdir()* system calls) shown on the right vertical axis. Our measurements show an overhead of about forty per cent (44% for *open()/close(),* 36% for *chdir()*).



Performance of modified system calls

fig. 1

6.2. Dumping a process

Since the *SIGDUMP* signal that is used to stop a process in order to move it to another a machine is so similar to the *SIGQUIT* signal that is used to terminate a process in order to obtain a core dump for debugging purposes, it is appropriate to compare the performance of the former to that of the latter. A program was started and then killed repeatedly in the following ways:

— By killing it with a *SIGQUIT* signal.

— By killing it with the new *SIGDUMP* signal.

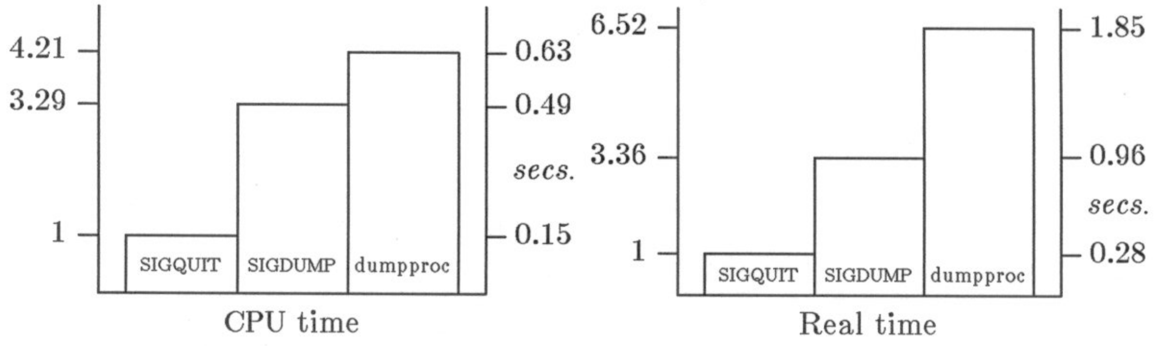
— By killing it with the *dumpproc* application program.

Since we were measuring the performance of the process migration mechanism, and not that of the file system, the program was chosen to be a small one, but which could still be used to verify that our mechanism was working correctly. The program increments and prints three counters (a register, a static variable allocated on the data segment and a variable allocated on the stack). On each iteration it inputs a line and appends it to an output file. This program, was always killed after its first prompt for input.

For each case, we measured the CPU and real time required to kill the process. The results are summarised in Figure 2, where the performance of the *SIGQUIT* signal is normalised to 1. *SIGDUMP* requires roughly three times as much time (both CPU and real) as *SIGQUIT*. Considering that *SIGDUMP* produces three dump files instead of the one that *SIGQUIT* produces, the result is very satisfactory. *Dumpproc* requires roughly four times as much CPU time and six times as much real time as the *SIGQUIT* signal. The extra CPU time is to be expected, as, in addition to using *SIGDUMP* to kill the process, the program has to modify the *filesXXXXX* file. The large discrepancy between CPU and real time can be explained by noting that the three files that are produced by *SIGDUMP* are created by the process that is being dumped. When *dumpproc* tries to open the *a.outXXXXX* file, it has to wait until the kernel switches its context to that of the process being dumped, so that the file can be created, and then wait again until the kernel switches its context back to it. To avoid busy loops, *dumpproc* simply sleeps for one second after each unsuccessful attempt to open *a.outXXXXX* (aborting after ten tries). Since the order of magnitude of the times involved is that of executing a UNIX signal (about 0.6 seconds for killing our particular test program with *SIGDUMP*), we feel that the performance of the new signal is quite adequate.

6.3. Restarting a process

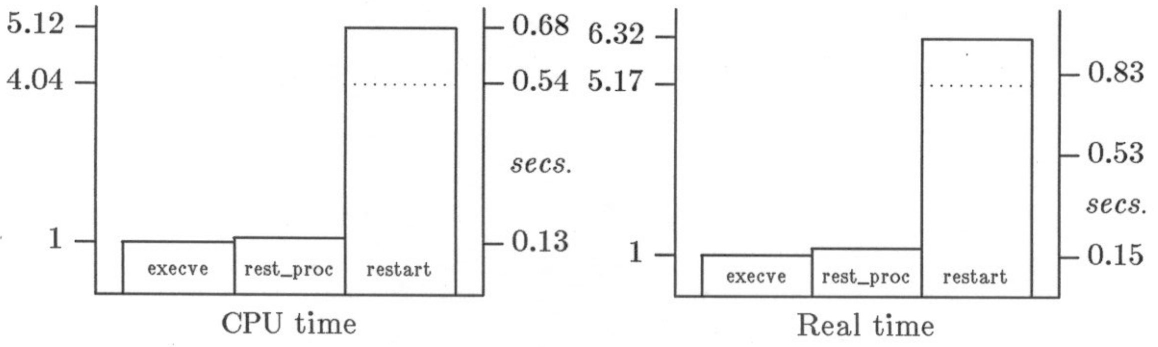
Since the *rest_proc()* system call that restarts processes that had been stopped on another host with the *SIGDUMP* signal is so similar to the *execve()* system call that executes new processes, it is again appropriate to compare the performance of the two, along with the *restart* application. For each case we measured the CPU and real time required by the system call or program in question to restart a test program (for *execve()* we measured the time required to execute the *a.outXXXXX* file). The performance of the system calls was obtained by adding timing code inside the kernel, as these system calls destroy the process that invoked them, making it hard to measure their performance at user level. The performance of *restart* was measured by timing its execution up to the point where it called *rest_proc()*, and adding to it the value already obtained by timing that system call. This is summarised in Figure 3, where the performance of *execve()* has been normalised to 1. The dotted line in *restart*'s



Relative performance of the *SIGQUIT* and *SIGDUMP* signals
and the *dumpproc* application

fig. 2

bar denotes the relative contributions of *restart* itself and *rest_proc()*, which were measured separately.



Relative performance of the *execve()* and *rest_proc()* system calls
and the *restart* application

fig. 3

We note that *rest_proc()* takes only slightly longer than *execve()*, which is entirely satisfactory. The *restart* application takes significantly longer (roughly five times more CPU time and six times more real time) than *execve()*. This can be justified by the fact that the application has to check the existence and verify the format of the three dump files and, most importantly, set that part of the process environment that can be set at user level, including the open files, which requires a large number of *open()* system calls. Nevertheless, considering the amount of work that is being done, the delay is not unacceptable, especially when we consider that our unit of measurement is the time required to execute a process, which, for our test program was less than 0.2 seconds, both in real and CPU time.

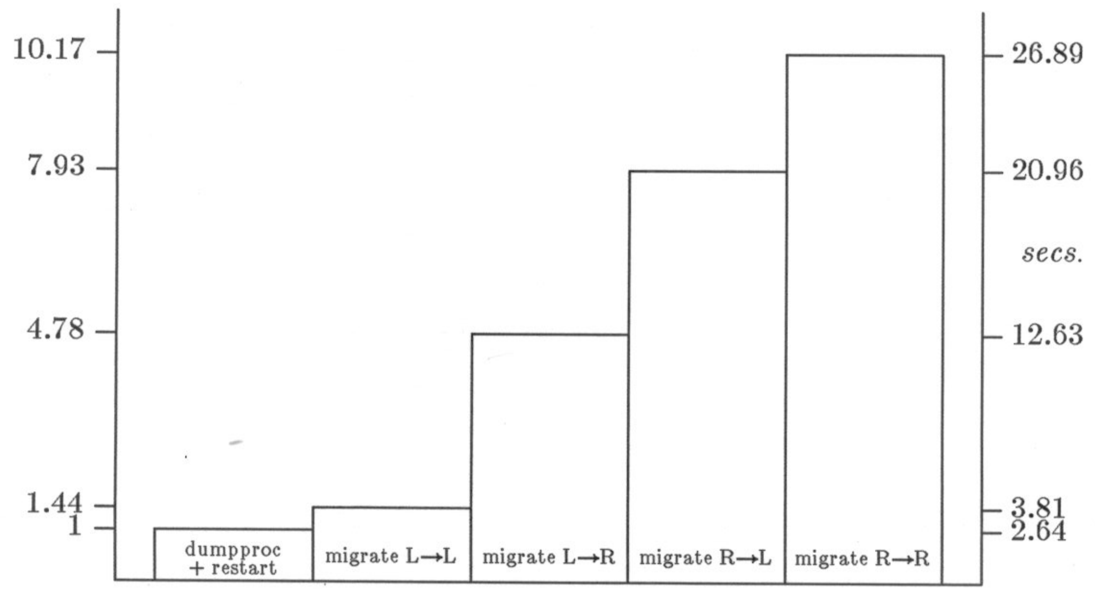
6.4. Migrating a process

In addition to the *dumpproc* and *restart* applications, we have the *migrate* application which combines the actions of *dumpproc* and *restart,* so that the user need not move to another terminal to kill or restart his or her process. *Migrate* has been implemented by executing the other two applications internally, by means of the UNIX remote shell facility *rsh,* if any of those programs needs to be executed on a remote machine. *Rsh* requires a lot of time to establish a connection with another machine, so, depending on where the process was originally running and to where it is to be restarted, *migrate* may take as much as ten times more as it would take to run *dumpproc* and *restart* on the appropriate machines. For our test program, this amounts to almost half a minute, which may be too large a sacrifice for having the capability to avoid an extra command. Still, the functionality is there if it is needed and, since the problem lies with the application and not with the process migration mechanism, it is always possible to write a better application which, by use of a UNIX daemon process and a well known port can achieve more satisfactory results: instead of using *rsh* to start processes remotely, applications will simply send messages to the daemon, who will start the processes on their behalf. The measurement results are summarised in Figure 4, where the performance of the *dumpproc/restart* combination is normalised to 1. (The difference between the local→remote and remote→local cases is due to the fact that, in each case, different programs are executed with a remote shell. In the first case, *dumpproc* is executed locally and *restart* remotely. In the second case, it is *dumpproc* that is executed remotely and *restart* is executed locally.)

7. Limitations

Although our system has been fully implemented, not all processes can be successfully migrated under our current design. The main limitation is the inability to redirect pipes and sockets to the migrated process. This means that, if a process was communicating with another by means of a socket, then, when it is migrated, it will no longer be able to do so. The best we can do in our current implementation is to redirect socket I/O to a file, which is probably of little use. However, processes that qualify for process migration are those that have lots of CPU activity and little I/O activity and are probably running by themselves without communicating with other programs. This means that, even without preserving sockets, the process migration mechanism is still useful.

The other limitation has to do with programs that "know" things about their environment, such as their process id or the name of the host in which they are running. For example, if a process repeatedly opens a temporary file whose name consists of a fixed prefix to which the process id is appended, then, after the process is migrated and the process id is changed, it will no longer be able to locate that file. (This will happen if the program requests the process id from the system every time, instead of doing so only once and then storing it in a variable for later use.) A more serious example is that of a process that acts differently depending on which machine it is running (*e.g.*, uses hardware floating point operations if running on host A, otherwise emulates them in software



Real time performance of the *migrate* application, compared to running the *dumpproc* and *restart* applications separately (L=Local machine, R=Remote machine)

fig. 4

— if that process is migrated from host A to some other host after it decides to use hardware operations, it will crash).

One solution that could be implemented to solve the first of these two problems is to add an extra field for an old process id and maybe even an old host name in the user structure, and change the *getpid()*[11] (get process id) and *gethostname()*[11] (get the name of the machine on which the process is running), system calls to return those new fields if the process has been migrated. This would require providing new system calls, that would return the real values, regardless of whether the process has been migrated or not. Programs that would use the new system calls would know of the existence of the process migration mechanism and would therefore be able to avoid transferring the problem of knowing things about their environment from one set of system calls to another. Programs that use the old system calls will be made to believe that they are running in their old environment. This would eliminate the temporary file problem, but will aggravate the problem of deciding what to do depending on the environment. Although processes that were migrated before making such decisions can run under the current system, they will make the wrong decision and crash if these modifications are implemented. However, there should be very few, if any, applications that fall in this category and such a modification is worth considering.

A further caveat is that processes that wait for one or more of their children to complete should not be migrated while waiting. When such a process is moved to another machine, it ceases being the parent of what used to be its children, and waiting for them will produce undefined results.

A final point which must be mentioned even though it is not actually a limitation, is the fact that our system does not tolerate much heterogeneity. Processes can be migrated to a similar CPU or to one whose instruction set is a superset of that of the original machine. Doing otherwise would result in trying to execute machine instructions on the destination machine which the machine does not have. (For example, we can migrate a program from a Sun 2 workstation having a Motorola 68010 processor to a Sun 3 workstation with a Motorola 68020 processor which is upward—compatible with the 68010, but we cannot migrate programs in the other direction.)

8. Applications

In this section we present some possible applications of the process migration mechanism, namely process checkpointing and load balancing.

The ability of our system to create an image of a process at a random point in its execution and then restart it on another machine (or possibly the same one), is exactly what we need to implement process checkpointing. If we have a program that has been running for a long time and for which it would be undesirable to have it restarted from the beginning in case of a system crash, we may write an application to take periodic snapshots of it and save those snapshots by moving them to a directory managed by the application (perhaps renaming them appropriately) which would then allow us to restart a program at its n—th checkpoint. The application should also make copies of all files that were open when the process was checkpointed, so that if the actual files were modified after the checkpoint, the copies can be used instead of the modified ones, thus presenting a consistent view of the files to the checkpointed program.

We can also use the process migration mechanism to achieve a better balance of the computational load in a distributed system. CPU bound jobs can be moved from busy nodes of the network to others that are idle, or have a much smaller load. Candidates for migration can be best selected from the processes that have been running for more than a certain amount of time. This will ensure that there is a high probability that the candidate program will keep running for some time, and that it is worth paying the overhead of moving it to another machine. In the case of load balancing, the *migrate* application may be too slow in terms of real time response and a more efficient one would have to be written, so that the delay of having a program run on a loaded system is not substituted with the low response time of *migrate*.

Another application (similar to the load balancing one described above) would be useful in the case where there are several CPU—bound jobs with a large expected runing time in a system (for example, the CPU hogs mentioned in [14]). These jobs can be run in one machine during the day (or not at all!), when users want to use the majority of the machines in the network. At night, when the load on most machines is low, these jobs can be distributed evenly throughout the system, and thus make efficient use of the network resources.

9. Conclusions

We feel that our implementation of process migration has been successful. Programs that do not communicate with other processes and that do not take actions that depend on knowledge of their environment can be successfully migrated to other machines, with complete user transparency. This is achieved efficiently, as stopping a process and restarting it on another machine requires a time comparable to that of killing the process to obtain a core dump and then restarting the process at the beginning on another machine using the standard UNIX system calls.

Since our current implementation does not migrate processes that use sockets, the next step in our research will be to examine whether support for sockets can be added to our system. Another interesting subject for future work is to implement one of the applications described in Section 8 and measure the performance of our mechanism in that context, as all our current measurements are of individual invocations of the process migration mechanism and not of any systemwide application.

In retrospect, implementing our system was relatively easy, despite the fact that this was our first experience with the UNIX kernel. Most of our time was spent in understanding the workings of UNIX. Once such an understanding was achieved, coding our modifications and additions was relatively straightforward, due in part to the modularity of the UNIX kernel and in part to the fact that most of our code was a relatively simple variation of existing kernel code.

10. Acknowledgements

We would like to thank Marc Staveley for assisting us in understanding the workings of the Sun 3.0 UNIX kernel, and for his help in the debugging our kernel modifications. We would also like to thank Larry Rogers for helping us in our first steps through the sources of the Sun operating system.

References

1. Michael L. Powell and Barton P. Miller, "Process Migration in DEMOS/MP," *Proceedings* 9th *ACM Symposium on Operating Systems Principles, Operating Systems Review*, vol. 17, no. 5, pp. 110—119, October 1983.
2. David A. Butterfield and Gerald J. Popek, "Network Tasking in the Locus Distributed Unix System," in *Proceedings of the Summer 1984 Usenix Conference*, pp. 62—71.
3. Marvin M. Theimer, Keith A. Lantz, and David R. Cheriton, "Preemptable Remote Execution Facilities for the V—System," *Proceedings* 10th *ACM Symposium on Operating Systems Principles, Operating Systems Review*, vol. 19, no. 5, pp. 2—12, December 1985.
4. Fred Douglis, "Process Migration in the Sprite Operating System," U. C. Berkeley Technical Report, 1987.

5. John Ousterhout, Andrew Cherenson, Fred Douglis, Michael Nelson, and Brent Welch, "An Overview of the Sprite Project," *;login: The USENIX Association Newsletter*, vol. 12, no. 1, January/February 1987.

6. Mordecai B. Rosen and Michael J. Wilde, "NFS Portability," in *Proceedings of the Summer 1986 Usenix Conference*, pp. 299—305.

7. S. R. Kleiman, "Vnodes: An Architecture for Multiple File System Types in Sun UNIX," in *Proceedings of the Summer 1986 Usenix Conference*, pp. 238—247.

8. *UNIX Commands Reference Manual,* Sun Microsystems Ltd., 1985.

9. D. M. Ritchie and K. Thompson, "The UNIX Time—Sharing System," *ACM Communications*, July, 1974.

10. Brian W. Kernighan and Dennis M. Ritchie, *UNIX Programming — Second Edition,* Bell Laboratories, 1978.

11. *UNIX Interface Reference Manual,* Sun Microsystems Ltd., 1986.

12. K. Thompson, "UNIX implementation," *Bell System Technical Journal*, vol. 57, no. 6, pp. 1931—1946, July—Aug. 1978.

13. Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry, "A Fast File System for UNIX," *ACM Transactions on Computer Systems*, vol. 2, no. 3, pp. 181—197, August 1984.

14. Will E. Leland and Teunis J. Ott, "Load—balancing Heuristics and Process Behavior," *Proceedings of Performance '86 and ACM SIGMETRICS, ACM Performance Evaluation Review*, vol. 14, no. 9, pp. 54—69, May 1986.