Implementing remote *fork()* with checkpoint/restart

Jonathan M. Smith†
John Ioannidis
Computer Science Department
Columbia University
New York, NY 10027

Abstract

We describe a method for implementing *checkpoints* on a UNIX system. The method requires no special operating system support. The checkpoints (a term we use both for the act of saving state and the result) are created in the file system name space. Availability in the name space allows facilities to duplicate and transfer files to be applied; in this case, we get replicated processes and process migration rather naturally. We describe the process migration implementation.

Our process migration implementation was easily optimized to achieve an execution speed improvement of greater than 7 times over our first implementation; this was accomplished by a combination of a faster file transfer mechanism and a change in the underlying protocol. We have incorporated the mechanism into a library routine, *rfork().* We conclude with a discussion of advantages, limitations and applications of our approach.

1. Introduction

An *image* is a description of a computation which can be *executed* by a computer. A *process* is an image in some state of execution. At any given time, the state of the process can be represented as two components: the initial state (the *image*) and the *changes* which have occurred due to execution. The total information, that is, the initial state together with the changes, gives us the *state* of a process.

It may be desirable to preserve this state at certain points in time, due perhaps to the amount of computation required to reach that state. These points in time can be used for acts of saving state called *checkpoints;* this name is also used for the result of saving the state. The UNIX paradigm for manipulating objects is through the file system name space; see Ritchie and Thompson [5] for details. The approach to accessing resources through name space entries has been applied to teletype devices, remote file systems, and system memory. Killian [3] attacked the problem of accessing process address spaces through name space entries in a distinguished directory */proc;* the process objects were named by their process ids. Unfortunately these facilities did not provide complete file semantics; while entries of */proc* could be read, analyzed, and modified, they could not be created; thus the interface could not use file system facilities for creating new processes. The interface was designed to support debugging, which it did successfully. If one could create a new process, and alter it, we could copy processes for such reasons as replicated processing and remote execution. In this paper, we describe a checkpointing method, our implementation, and the construction of a library interface for a ‘‘remote fork’’ call. A process successfully executing a *fork()* operation generates two copies of its address space; these are often distinguished as *parent* and *child* by the return value of the *fork()* call. If the *child* process continues its execution with the containing address space located on a processor different form the *parent* process, we have achieved a ‘‘remote fork’’.

† This work was supported in part by equipment grants from the Hewlett-Packard Corporation and AT&T, and NSF grant CDR-84-21402. E-mail: jms@close.cs.columbia.edu

UNIX is a registered trademark of AT&T.

2. Process Migration

Process migration is the *transfer* of some (significant) subset of this information to another location, so that an ongoing computation can be correctly continued. Process migration is most interesting in systems where the involved processors do not share main memory, as otherwise the state transfer is trivial, as it can be accomplished with pointers. A typical environment where process migration *is* interesting is autonomous computers connected by a network. A survey of process migration mechanisms is available in Smith [7] ; a conclusion drawn from the survey is that message-passing systems ease implementation, and the stateful nature of most operating system kernels is an impediment to migrating processes.

In any case, these process migration mechanisms demonstrate that the state of an executing process can be moved between homogeneous machines, and that the execution can be continued. This transfer of address spaces is what intrigues us.

Assuming that we could effect such a transfer, what should be the primitive to be used in achieving it? We chose to implement a call akin to the *fork(2)* system call, that creates a child process on a remote machine. We called it *rfork().*

Because of implementation limitations, *rfork* is not a drop-in replacement for *fork,* but if proper care is taken, it can be a significant tool.

3. Implementation Details

A UNIX process's address space typically has the layout of the leftmost part of Figure 1. The stack segment is at the numerically higher addresses, while the text segment is at the numerically lower addresses. The address space, objects referenced by descriptors in the address space (e.g., open files), and system state (e.g., virtual-to-physical address mappings) comprise the state of a process. Since address mappings and similar state are transparent to the process, we will ignore such information as state. We are interested in the transfer of this state to some remote location. We note that the information in the address space is dependent on the architecture of the machine and the operating system; the program only runs on this kind of ''UNIX virtual machine'', and hence the proposed migration is referred to as *homogeneous,* as opposed to *heterogeneous*[8] process migration, where the process's state could be transferred between unlike machines. We note that heterogeneous migration can be accomplished by inserting an intervening ''virtual machine'', e.g., an interpreter. Falcone [1] presents some ideas along this line; a complete implementation of such a migration mechanism is obviously a non-trivial exercise.

How can we transfer a running process from one machine to another? At the highest level of abstraction, we want to do the following:

1. Checkpoint the process
2. Move the image
3. Restart it

Since the existing facilities for transferring information are biased towards file transfer, we felt that the following realization of the abstraction was most effective:

1. Store the state of the process into a file.
2. Copy the file to a remote system. One way we can do this is by using a remote copy command (e.g., **rcp**). Alternatively, we can take advantage of the homogeneous namespace provided by a distributed file system, such as the Sun NFS [6] and just checkpoint the process in a globally accessible file. This has the advantage that the image of the executable file goes across the network at most twice (once when dumping and maybe once more when loading it, if the execution is restarted on a diskless workstation). As a mounted file system behaves like a virtual channel, in that efficiency is high once a connection is made, latency is reduced.
3. Restore the process from the file, at the remote system.

Since step #2 can easily be accomplished, we sought to achieve the other two steps.

Step #3 can only be accomplished by use of the *exec()* system call that will be invoked on the remote machine, as it is the only way to obtain a running copy of an image. Hence, step #1 must create the file in a format which *exec()* can use; on UNIX, executables are in *a.out()* format; see Section 5, ''File Formats'' in [10] for details. Figure 1 illustrates the mechanics.

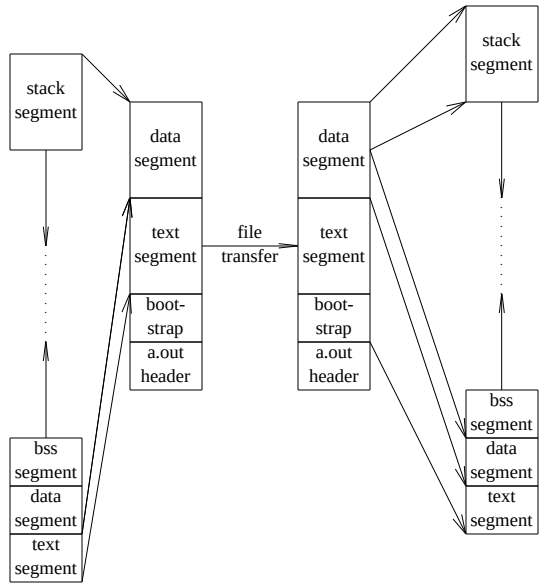

Figure 1: UNIX process migration

3.1. Commentary

It should be noted that our implementation of *rfork()* is entirely user level. No kernel modifications were required. Currently, the processes as migrated in this fashion are *deaf, dumb,* and *blind.* That is, they don't carry any of the state which the system retains in order for, e.g., file descriptors to make sense. If restrictions are adhered to, a set of library routines could be generated to provide the semantics of the *standard I/O* library; these would have extra data such as the file name associated with the opening of files; file positions, et cetera could easily be stored and restored. Of course, this may require special handling to deal with devices such as terminals; disk files seem relatively simple, especially with a network file system. Other examples are:

1. The *process group* in which a given process is contained. This also means that when a 'remote' child terminates, its originating process cannot be signaled and, of course, it cannot *wait()* for it.
2. The *current working directory* of a process.
3. The *signals* received by a process.
4. The time used by the process so far in its execution.
5. Children the process may have spawned with the *fork()* primitive.
6. The unique *process identifier* associated with the process.
7. State from active inter-process communications, such as pipelines, network connections, etc.
8. Other state related to specific facilities a particular version of the UNIX system may provide, such as *shared memory, semaphores,* and *page maps.*

It does not seem easy to deal with some of these items, e.g. maintaining the same *process id* can not be done from the user level. However, many of the more important system functions (namely those dealing with files, as mentioned above) can be provided by means of library routines combined with programmer conventions. For example, if all programmers adhere to the use of the **stdio** library routines for file I/O, we can do the following:

1. Replace the **stdio** library routines with routines which store extra information which we will need to restore some of the process state.
2. Capture the file name passed to *fopen()* and store it in a buffer associated with the file.
3. Save the value of the file pointer, as it will be changed only by our routines, such as *fseek(), fread(),* and *fwrite().*
4. Indicate in the extra information associated with a file that the file must be reopened when the process is about to be moved. When the process is restarted, the files will be reopened and repositioned.

General file descriptors would be too difficult to handle transparently. For example, consider the case where we have a program running which has reset the terminal modes using *ioctl();* it stores the previous terminal modes in some system dependent data structure. When the program terminates, it may reset the terminal modes; however if it has migrated to a system which is of a different type, it may cause an error even if we have reopened the file. Use of the X Window System [4], NeWS [2], or similar network window systems can eliminate many of the terminal interface problems, but they illustrate the problems with detailed examination of process state.

As with any engineering problem, circumstances create uncertainty about the right way to save descriptor-accessed state. Two methods we've considered are:

1. Preserving the names of opened files (as mentioned above), with the intent of reopening them upon restart. File positions, etc., can be retrieved with system calls. The problem is that the files may be inaccessible or modified, so that data the running module had access to and knowledge about are not the same.
2. Flushing all writable file descriptors and draining all readable file descriptors into memory buffers. After restart, *read()* calls will be satisfied from the buffer. This remedies the previous accessibility problem, in that what was accessible before will remain accessible; readable data has been preserved in the address space. Unfortunately, many interesting files are large; set so large in fact that size limitations in *exec()* make the result of this approach non-executable.

Of course, neither of these approaches are fully general; consider the case where the state to be preserved involves an *active* entity, e.g., we want to save process ''b'' from

```
a | b | c
```

How do we deal with the states of processes ''a'' and ''c''?

Deciding which set of facilities to preserve is an engineering issue, and it serves as the set of constraints under which migration is achievable, as mentioned in the section above on process migration.

3.2. Performance

The checkpoint/restart facility was implemented and tested on Hewlett-Packard HP9000 (Series 320) and SUN-2 workstations connected via a 10Mbit Ethernet in Columbia University's Computer Science Department in the Fall of 1986. This in turn has been used to construct a process migration mechanism, by using the 4.[23] BSD **rcp** command to perform the remote file transfer and the **rsh** command to execute the newly transferred image. This process migration mechanism has been used to move running processes between 20 of these workstations. By distinguishing between the state saving (checkpointing) activity and the state transfer activity, we were able to measure and refine the performance of each activity independently. In the Fall of 1987, the checkpointing code was ported, with considerable effort, to the AT&T 3B2/310 computer. Some simple timing measurements were made with a trivial checkpointing program. The checkpoint required 18668 bytes of storage, where the original program

required 29756. The program required 0.51 sec. of real time, 0.02 sec. of user time, and 0.21 sec. of sys time. Copying (**rcp**) the checkpoint from the 3B2 to a VAX 11/750 took 7.57 sec. real, 0.10 sec. user, and 0.55 sec. sys. For a Hewlett-Packard HP9000 (Series 320) running HP-UX version 5.17 (code for automatic segment size determination and COFF section headers not in tst_frz) the program took 18712 bytes, while 22516 were needed by the checkpoint. Checkpointing required 0.82 sec. real, 0.02 sec. user, and 0.36 sec. sys. Copying took 6.28 sec. real, 0.08 sec. user, and 0.70 sec. sys.

3.3. Implementation using NFS

We implemented *rfork* for a network of Sun-2's running release 2.0 of the SUN version of UNIX. The first thing we did to enhance performance was take advantage of the Network File System. We set up a *spooling* directory, accessible from all the machines, where we stored the checkpointed image of the process to be moved. Since NFS is tuned for performance, file transfer (a performance problem when using **rcp**) incurred a much smaller delay. Another by-product of this method is buffer cache availability of the transferred blocks. The data blocks saved in the spool directory were in the server's disk buffers, which improved the transfer speed when the request for the blocks occurred within a short time after they were saved; this is usually the case.

The remote execution server of the release of the operating system that we were using, **rshd,** was creating problems with zombie processes, child processes not being properly terminated and waited for, etc. It also used the **yp** (yellow pages) distributed name server every time it wanted to map a host name into a host address, which resulted in an overhead of about 5 seconds of real time. Also, the virtual-circuit oriented mode of operation of **rsh** meant that there would be processes lingering in the local system for some time. For these reasons, we wrote our own simple remote execution server. Thus, on all the machines that we wanted to be able to move to, we run a small program that would accept a UDP datagram containing some rudimentary authentication information and the name of an executable program (usually the program just saved in the *rfork* spool directory).

We also wrote a notification system, again using UDP datagrams. This service ran on one of the workstations and accepted short messages from any other. Upon receipt, it would timestamp each message and log it to the terminal. We used this facility to obtain our timing measurements and monitor the progress of the migrating processes through the network.

3.3.1. Programming Example

We'll examine a program used for taking measurements on the HP-UX implementation, given as Figure 2.

```
/*
 * Merry-Go-Round
 */
#include <sys/types.h>
#include <stdio.h>
#include "../include/hosts.h"
#include "../include/ckrs.h"

#define INET_MONITOR INET_read
extern int diag_msgs_host;

char *CKRS_DIR="/src/doug/migrate";
/* inet addresses of machines to visit */
u_long visit[] =
{
   INET_wait, INET_fork, INET_kill,
   INET_select, INET_close, INET_read,
};

main(argc, argv)
int argc;
char **argv;
{
   char hostname[255], line[40];
   int i;

   diag_msgs_host = INET_MONITOR;

   for (i=0; i<6; i++){
      gethostname(hostname, 155);
      sprintf(line, "I am at %s.%d",
         hostname, getpid());
      report(INET_MONITOR, line);

      move_to(visit[i]);
   }

   report(INET_MONITOR, "I'm back");
}
```

Figure 2: Source, Merry-Go-Round

where the macro `move_to(_x)` is defined as `if(rfork(_x)==PARENT)exit(0)`. The *report()* call sends a message to the monitor daemon, which prints the originating internet address, the time and the message received. The *rfork()* call also sends three diagnostic messages: one when it starts dumping, one when it's finished dumping, and one when it resumes execution on the remote machine.

3.4. Observations

The overall migration time was roughly 1.3 seconds (with an aberrant value of 2.42 due to very heavy load on a server machine (`kill`)).

The average time to dump the executable (approximately 70Kbytes) was just under 700 milliseconds, and

the average time to start up the new process on the remote machine was just over 300 milliseconds.

In all cases, the dump file was stored and retrieved over NFS. The spool directory resided on a SUN-3/280 running release 3.5 of SunOS.

What we didn't do:

- For reasons stated previously, our migrating processes were blind, dumb and deaf. However, by using the monitor service, we could send status reports and monitor the status of the process as it moved from machine to machine.
- Again because we only did a prototype, we had no way of signaling completion of the remote process (death of an *rchild*). Conceivably, we could have a SIGNALing daemon on each machine that would listen on a UDP port, do some rudimentary authentication and send the appropriate SIGNAL to the local process.

Another useful program could implemented using *rfork()* to hop through a list of hosts, gathering user names at each host, storing the names in malloc()'ed memory, and returning to the originating system to report the findings. We believe that the functionality of *rfork()* is particularly appropriate to compute-intensive applications. A master program can start on one machine, then do a series of *rfork* operations to distribute itself to available machines to perform portions of a computation. Some experiments to verify this are planned; in particular for a computationally intensive task such as ray-tracing.

4. Conclusions

We have implemented a mechanism to create a process *checkpoint* which resides in the UNIX file system. This allows the checkpoint to be manipulated, e.g., to be copied and transferred across a network.

Using this mechanism, we were able to create a simple and elegant implementation of process migration. In addition, we implemented *rfork(). Rfork()* resembles the UNIX *fork()* system call, yet allows the child to continue executing on a remote machine. Once the design and initial implementation were complete, we analyzed the performance, and reimplemented pieces of the system (on several different machine architectures) in order to improve the response time. This improvement was dramatic; from about 7 seconds of real time on the HP9000 and the 3B2, to less than 1 second on the HPs and Suns using NFS. As the *fork()* primitive has proven useful in developing multiple process applications on uniprocessors, we have every reason to believe that *rfork()* can be a useful tool in developing distributed applications.

5. Acknowledgements

Prof. Gerald Q. Maguire, Jr. encouraged and assisted us. Perry Metzger wrote the first version of the remote execution daemon for the Suns.

Spencer Thomas and others wrote the code for GNU Emacs [9] `unexec()`, which we examined. Robert Herndon wrote a version of the `freeze()` library routine for the VAX from which we borrowed.

6. References

[1] Joseph R. Falcone, ''A Programmable Interface Language for Heterogeneous Distributed Systems,'' *ACM Transactions on Computer Systems* **5**(4), pp. 330-351 (November 1987).

[2] Sun Microsystems Inc., *NeWS Preliminary Technical Overview*, October 1986.

[3] T.J. Killian, ''Processes as Files,'' in *USENIX Proceedings* (June 1984), pp. 203-207.

[4] Ram Rao and Smokey Wallace, ''The X Toolkit: The Standard Toolkit for X Version 11,'' in *Proceedings, Summer 1987 USENIX Conference*, Phoenix, AZ (June, 1987), pp. 117-130.

[5] D.M. Ritchie and K.L. Thompson, ''The UNIX Operating System,'' *Communications of the ACM* **17**, pp. 365-375 (July 1974).

[6] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and R. Lyon, ''The Design and Implementation of the Sun Network File System,'' in *USENIX Proceedings* (June 1985), pp. 119-130.

[7] Jonathan M. Smith, ''A Survey of Process Migration Mechanisms,'' *ACM SIGOPS Operating Systems Review*, pp. 28-40 (July, 1988).

[8] Jonathan M. Smith and Gerald Q. Maguire,Jr., ''Process Migration: Effects on Scientific Computation,'' *ACM SIGPLAN Notices* **23**(3), pp. 102-106 (March 1988).

[9] Richard Stallman, *GNU Emacs Manual, Fourth Edition, Version 17*, Free Software Foundation, Inc., 100 Mass Ave., Cambridge, MA 02138 (February 1986).

[10] K. Thompson and D. M. Ritchie, *UNIX Programmer's Manual*, Bell Laboratories (1978). Seventh Edition.