Notes on the Implementation of a Remote Fork Mechanism

Jonathan M. Smith

John Ioannidis

Computer Science Department
Columbia University
New York, NY 10027

ABSTRACT

We describe a method for implementing a *remote fork* , a primitive with the semantics of a UNIX *fork()* call which begins the execution of the child process on a remote machine.

We begin by examining the subject of process migration, and conclude that most of the relevant process state can be captured and transferred to a remote system without operating system support. We then show how our implementation is easily optimized to achieve a performance improvement of greater than 10 times when measuring execution time. We conclude with some comments on limitations and applications of the remote fork mechanism.

Notes on the Implementation of a Remote Fork Mechanism

Jonathan M. Smith

John Ioannidis

Computer Science Department
Columbia University
New York, NY 10027

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.

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". The remainder of this report describes the construction of such a mechanism.

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. This is illustrated in Figure 1:

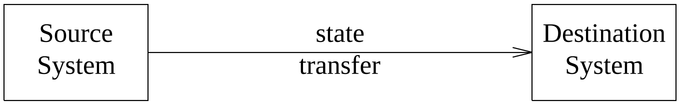

Figure 1: Process Migration

The *flow of execution* of the process which is being transferred is illustrated in Figure 2:

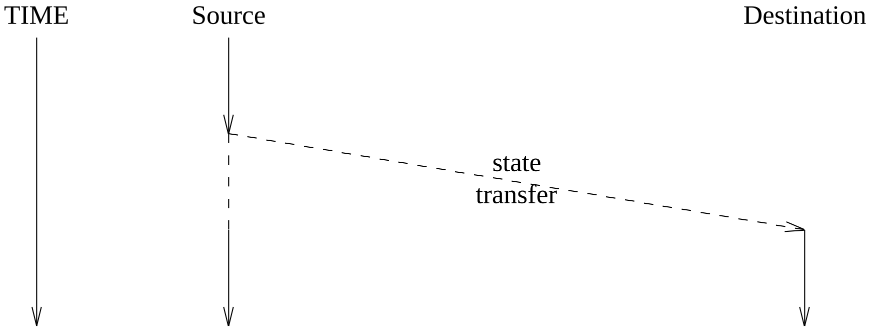

Figure 2: Flow of execution of rfork()-ing process

We can define constraints on the computation so that the subset transferred is sufficient to provide correct computation *with respect to the constraints.*

If we *transfer* the state of a process from one machine to another, we have *migrated* the process.

Process migration is most interesting in systems where the involved processors do not share main memory, as otherwise the state transfer is trivial. A typical environment where process migration *is* interesting is autonomous computers connected by a network.

2.1. Work in Process Migration

The introduction to the paper [6] describing DEMOS/MP's process migration scheme makes the following observation:

> Process migration has been proposed as a feature in a number of systems, but successful implementations are rare. Some of the problems encountered relate to disconnecting the process from its old environment and connecting it with its new one, not only making the the new location of the process transparent to other processes, but performing the transition without affecting operations in progress. In many systems, the state of a process is distributed among a number of tables in the system making it hard to extract that information from the source processor and create corresponding entries on the destination processor. In other systems, the presence of a machine identifier as part of the process identifier used in communication makes continuous transparent interaction with other processes impossible. In most systems, the fact that some parts of the system interact with processes in a location-dependent way has meant that the system is not free to move a process at any point in time.

Several systems provide facilities to migrate processes. Among these are the LOCUS [5, 16] system, the DEMOS/MP [6] system, Stanford's V [14] system, the Amoeba [12, 13] system, and the MOS [1, 2] system. DEMOS/MP, V, and Amoeba are message-based; the design is that of a small kernel for passing messages between processes, coupled with server processes which provide access to system resources. These systems have relatively little trouble implementing transparent process migration. The reason for this is the message-oriented system's designs; a small kernel of message-passing routines contains little state not in the process's context, and thus there is little at a given location that a process can be dependent upon.

LOCUS, being based on UNIX [8, 10] has a somewhat trickier job, as the UNIX process model [15] requires a great deal of context to be maintained. However, given that the file system is the main point of interface, and that the file system name space is global in LOCUS, process migration is eased somewhat; the MOS system, developed at the Hebrew University of Jerusalem, is also based on UNIX and attempts to emulate a single machine UNIX system using a collection of loosely connected independent homogeneous computers. MOS has a global name space; a feature of the implementation is the notion of a *universal i-node pointer* ; these effectively provide uniform (i.e., transparent remote) access to resources, as in the LOCUS system; thus while the process possesses context, a great deal of it is location-independent. This makes process migration somewhat less difficult.

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. The next section describes how this is accomplished.

3. Implementation Details

A UNIX process's address space typically looks like[1] the following:

[1] 1.) This layout has the attractive property that the top of the *bss* segment will intersect the bottom of the *stack* segment when there is no allocable memory left in the address space, an occurrence which was fairly common on the PDP-11 and rarely if ever on the VAX or the SUN. Unfortunately, this elegant layout has not been preserved in some implementations of UNIX. The AT&T 3B2 computer's UNIX System V Release 3.0, incorporating the WE 32100 processor, maps *text,*

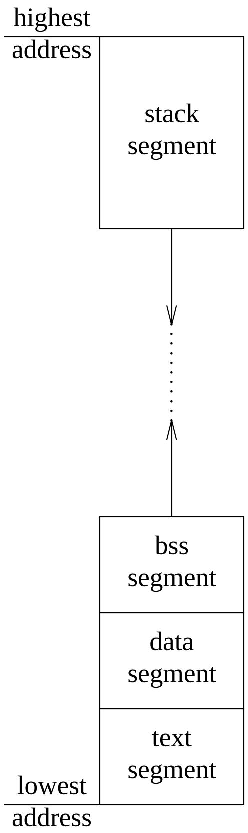

Figure 3: Address Space Layout, UNIX process

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*[11] 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.

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 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 [9] and just checkpoint the process in a globally accessible file. This has the advantage that the image of the executable file goes accross the ethernet 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, latency is reduced.

data, and *bss* into one segment, based at 0x80000000; the *stack* segment is based at 0xc0000000 and grows *up!*

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; UNIX binaries use *a.out* format.

Figure 4 illustrates what happens:

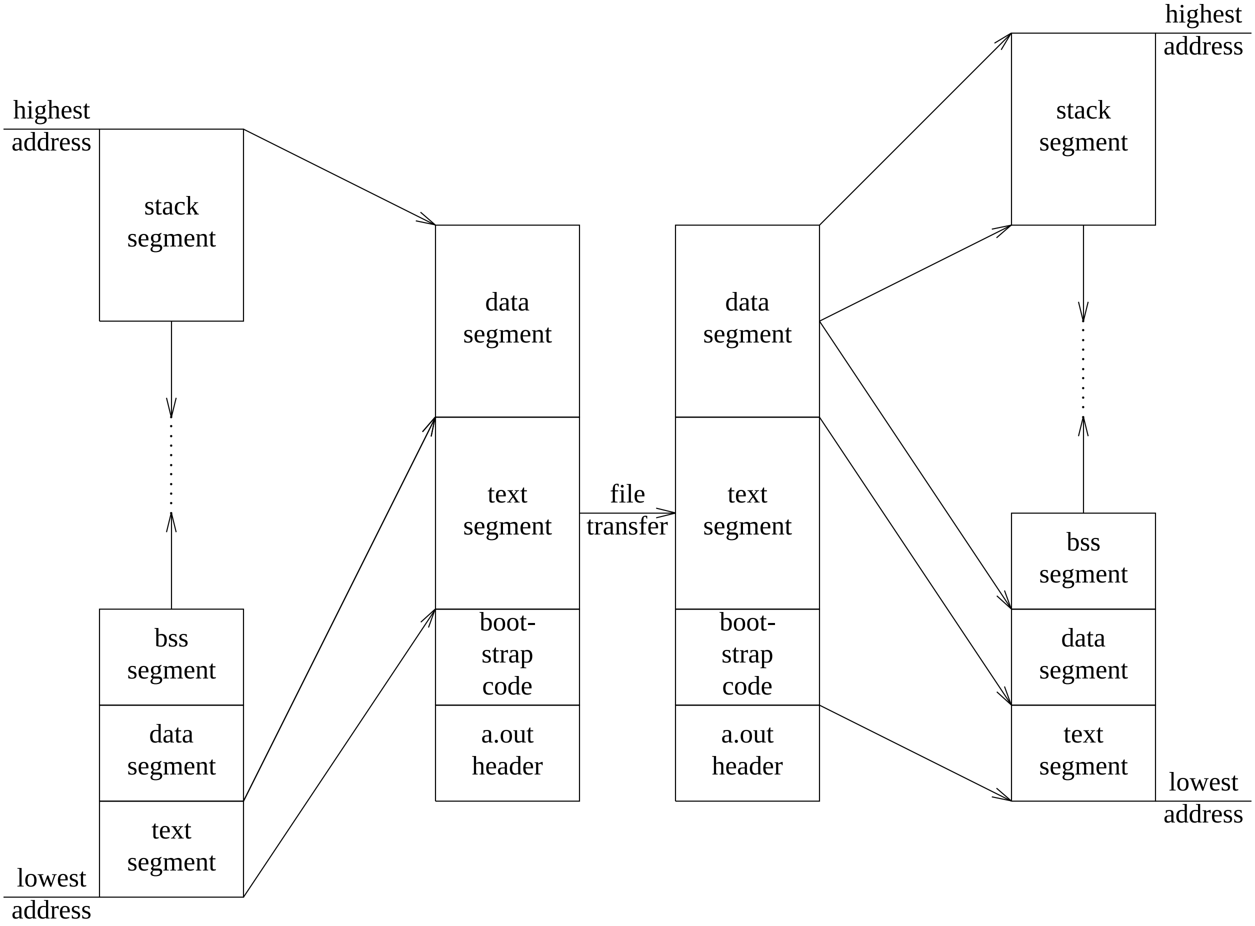

Figure 4: UNIX process migration

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 [17]. 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 it's exiting, 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 X-windows [7] , NeWS, or similar network window systems can eliminate many of the terminal interface problems, but they illustrate the problems with detailed examination of process state.

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 **rcp**[3] command to perform the remote file transfer and the **rsh**[3] 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 are presented in the next figure:

```
$ cat tst_frz.c
#include <stdio.h>

main()
{
  freeze( "checkpoint" );
  printf( "Hello there.0 );
  exit(0);
}

# AT&T 3B2/310 running SYS V Rel.3
$ time tst_frz
Hello there.

real    0m0.51s
user    0m0.02s
sys     0m0.21s
$ time rcp checkpoint garfield:/tmp/checkpoint

real    0m7.57s
user    0m0.10s
sys     0m0.55s
$ ls -l checkpoint tst_frz
-rwxr-xr-x   1 jms      phd         18668 Nov 27 11:28 checkpoint
-rwxr-xr-x   1 jms      phd         29756 Nov 27 11:17 tst_frz
$
# Hewlett-Packard HP9000 (Series 320) running HP-UX version 5.17
# tst_frz missing automatic segment sizing and COFF section headers
$ time tst_frz
Hello there.

real    0m0.82s
user    0m0.02s
sys     0m0.36s
$ time rcp checkpoint garfield:/tmp/checkpoint

real    0m6.28s
user    0m0.08s
sys     0m0.70s
$ ls -l tst_frz checkpoint
-rwxrwxrwx   1 jms      phd         18712 Nov 27 11:31 tst_frz
-rwxr-xr-x   1 jms      phd         22516 Nov 27 11:33 checkpoint
```

Figure 5: Simple timings of checkpoint/transfer

3.3. Implementation on the Sun workstation

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 highly optimized in that respect, file transfer (which was a problem when we were using **rcp**) became less of a delay problem. Another thing to keep in mind is that, this way, the data blocks that were just saved in the spool directory were in the server's disk buffers, which made the transfer even faster.

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 [4] containing some rudimentary authentication information and the name of an executable program (usually the program just saved in the rfork spool directory).

In addition to that, 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.

Given all that, we got the measured performance improvement of ˜0.5 second versus ˜7.0 seconds, a factor of 14.

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 an signalling daemon on each machine that would listen on a UDP port, do some rudimentary authentication and send the appropriate SIGNAL to the local process.

An example of a useful program that used rfork is presented in Figure 6.

```
for i in <list of hosts>
   move to host (i)
   run w to get the names of users and store
      the results in malloced memory
return to the host of origin
report the findings
```

Figure 6: Useful program with rfork()

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 rforks to distribute itself to all the available machines to do part of a computation. Some experiments to verify this are planned.

4. Conclusions

We have implemented a mechanism that preserves most of the semantics of the *fork()* system call, yet allows the child to continue executing on a remote machine. Once the design for the user-level implementation was complete, we analyzed the performance problems and reimplemented pieces of the system on different machine architectures in order to show greatly improved performance (from ˜7 seconds of real time on the HP9000 and the 3B2, down to ˜.5 seconds on the Suns.) 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

We would like to acknowledge the encouragement and assistance of Prof. Gerald Q. Maguire, Jr. We would also like to acknowledge the contribution of Perry Metzger, who wrote the first version of the remote execution daemon for the Suns.

6. References

[1] Amnon Barak and Amnon Shiloh, ‘‘A Distributed Load-balancing Policy for a Multicomputer,’’ *SOFTWARE - PRACTICE AND EXPERIENCE* **15**(9), pp. 901-913 (September 1985).

[2] Amnon Barak and Ami Litman, ‘‘MOS: A Multicomputer Distributed Operating System,’’ *SOFTWARE - PRACTICE AND EXPERIENCE* **15**(8), pp. 725-737 (August 1985).

[3] BSD, *UNIX User’s Manual, 4.2 BSD,* University of California, Berkeley (1982).

[4] BSD, *Unix Programmer’s Manual (4.2BSD),* University of California, Berkeley (1983).

[5] David A. Butterfield and Gerald J. Popek, ‘‘Network Tasking in the LOCUS Distributed UNIX System,’’ in *USENIX Summer 1984 Conference Proceedings* (June 1984), pp. 62-71.

[6] Michael L. Powell and Barton P. Miller, ‘‘Process Migration in DEMOS/MP,’’ in *Proceedings of the Ninth ACM Symposium on Operating Systems Principles* (1983).

[7] 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.

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

[9] 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.

[10] Jonathan M. Smith, ‘‘Approaches to Distributed UNIX Systems,’’ Technical Report CUCS-223-86, Columbia University Computer Science Department (1986).

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

[12] Andrew S. Tanenbaum and Robbert Van Renesse, ‘‘Distributed Operating Systems,’’ *ACM Computing Surveys* **17**(4,), pp. 419-470 (December 1985).

[13] A.S. Tanenbaum, S.J. Mullender, and R. Van Renesse, ‘‘Using sparse capabilities in a distributed operating system,’’ in *Proceedings of the 6th International Conference on Distributed Computer Systems (IEEE)* (1986), pp. 558-563.

[14] Marvin M. Theimer, Keith A. Lantz, and David R. Cheriton, ‘‘Preemptable Remote Execution Facilities for the V-System,’’ in *Proceedings, 10th ACM Symposium on Operating Systems Principles* (1985), pp. 2-12.

[15] K.L. Thompson, ‘‘UNIX Implementation,’’ *The Bell System Technical Journal* **57**(6, Part 2), pp. 1931-1946 (July-August 1978).

[16] Bruce J. Walker, Gerald J. Popek, Robert English, Charles Kline, and Greg Thiel, ‘‘The LOCUS Distributed Operating System,’’ *ACM SIGOPS Operating Systems Review (Ninth ACM Symposium on Operating Systems Principles)* **17**, pp. 49-70 (October 1983).

[17] P.J. Weinberger, ‘‘The Version 8 Network File System,’’ in *USENIX Proceedings* (June 1984), p. 86.