Process Migration in Sprite: A Status Report

Fred Douglis John Ousterhout

Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

spriters@ginger.Berkeley.EDU

1 Introduction

This article discusses the history of process migration in Sprite. It focusses on implementation details, describing how we provide *fully transparent, preemptable* remote execution. Our limited experience with migration suggests that we have met our original goal of sharing processor cycles transparently while preserving response times for users. However, this capability has come at a much greater implementation cost than we had anticipated. Our greatest problem resulted from changes to the rest of the kernel breaking process migration when migration was not used regularly enough to learn about problems as they arose.

The next section describes the goals of process migration in Sprite. Section 3 discusses the implementation of migration, and Section 4 gives a preliminary evaluation of the utility of process migration in our environment.

2 Goals

We implemented process migration with three goals in mind:

1. **Take advantage of idle processors.** In a network of workstations, a number of processors may be unused at any given time. We wanted users to be able to improve throughput by running programs in parallel on multiple hosts.

2. **Share cycles transparently.** Just as the file system hides the location of files from users, the kernel should hide the location of processes. To a user taking advantage of idle machines, nothing should be different except the speed of execution. Above all, programs should not be aware if they are executing on a different machine from one moment to the next, and ideally no restrictions should be placed on what processes are allowed to migrate.

3. **Preserve response times for users.** We believed users would be reluctant to allow their workstations to be used in their absence, if they would suffer prolonged degraded response when they returned.

The next section describes how the implementation of process migration in Sprite addresses these goals.

3 Implementation

Process migration in Sprite combines *transparent remote execution* with *eviction.* In Sprite, each process appears to run on a single host throughout its lifetime. This host is known as the *home node* of the process. Regardless of where the process physically executes, it appears in listings of processes on its home node; any host-specific devices it opens, such as the display or system log file, are from its home node; even internet connections, such as *rcp* or *rlogin*, are performed through its home node. Any effects produced by the process are done as though the process were executing on its home node, including the extent to which it creates child processes: thus, its descendants have the same home node as the process regardless of where they execute. If any process is running someplace other than its home node, and the host on which it is executing is reclaimed—the workstation receives input from its console—then the process is immediately *evicted* by migrating it back to its home node.

Sprite supports fully transparent remote execution by providing support in the kernel for foreign processes; that is, processes that are not executing on their home node. Sprite has a Unix-like system call interface, in which processes trap into the kernel on the machine on which they execute. For foreign processes, the kernel distinguishes between location-dependent and location-independent calls. Location-dependent calls are system calls that may have a different effect depending on the host on which they are executed: for example, accessing the name of the host or the time-of-day clock. Location-independent calls are system calls that have the same effect regardless of their location: for example, calls that access the file system. Sprite kernels forward the location-dependent system calls of a foreign process to its home node for interpretation, but they handle location-independent calls without redirection. As the system has matured, we have attempted to make more and more calls location-independent in order to avoid the overhead and complexity of forwarding calls; most of the remaining location-dependent calls deal with process management and/or affect the user's perception of executing processes: the *fork*, *exec*, *exit*, and *wait* calls are all handled partially on a process's home node since the home node is a party to the creation and destruction of all processes.

Sprite supports eviction with a four-pronged policy:

1. A daemon process on each host monitors its load average and idle time, and the daemon initiates an eviction procedure on each foreign process on the host if it detects activity at the workstation's console. (The load average is used primarily to determine when to accept new foreign processes: if the load average is low and there has been no recent activity, foreign processes are allowed.)

2. The home node of a process is ultimately responsible for it. If the process is evicted, it is migrated back to its home node; the process may be migrated again if another host is available.

3. The process is suspended while it is migrated. Permitting an evicted process to execute while its virtual memory image is transferred to disk would reduce the time during which the process is frozen but also reduce the processing power available to the machine's owner while the evicted process continues to execute [5].

4. No residual dependencies may be left on the host after eviction. The time to migrate a process may be reduced substantially by retrieving the memory image of a migrated process from its previous host as pages are referenced [6]. However, copy-on-reference requires that the former host continue to dedicate resources and service requests from the evicted process

for a longer period of time than would be necessary in a system that copies the memory along with the rest of the process's state. We believe it is reasonable for a process's *home* to provide support for it throughout its lifetime, but another host should only need to provide resources to a foreign process as long as the process executes on that host.

Migrating a process consists of several steps. First, the process is signalled: at the time the signal is handled in the kernel, the state of the process within the kernel is simple, well-defined, and easily recreatable on the target host. Second, the process state is encapsulated and sent to the target. This includes information such as register contents, program counter, signal masks, and so on. Third, the virtual address space of the process is written to swap files, and the page tables are encapsulated and transferred, along with references to the swap files. The process demand-pages its virtual image from the swap files once it resumes execution. Finally, the open files of the process are encapsulated and transferred.

Encapsulating the state of files and transferring open files, while keeping file caches consistent, was the single hardest problem we faced. Since file servers keep track of which hosts are reading or writing each file, migrating a process requires that file servers atomically update their notion of how the process's files are accessed. For example, if a process has a file open for writing, and it forks and migrates a child, the file would then be open for writing on two different hosts; Sprite's cache consistency algorithm dictates that the file be made non-cacheable on the machines on which the two processes are executing [2]. File management was further complicated by the need to support shared stream offsets: when one of those two processes writes to the file, the next operation from the other process must reflect the new offset resulting from that write—even though the operation takes place on another host.

4 Evaluation

Because process migration has been in day-to-day use for only a few weeks as of this writing, we have difficulty assessing its effectiveness. Migration was clearly well-accepted once it was made available to other users, despite any initial instability. However, the real "proof of the pudding" will come once migration has remained in regular use despite ongoing changes to the rest of the system. The history of process migration in Sprite is telling: although migration first worked for simple test cases as early as the fall of 1986, and we presented a paper on migration in the fall of 1987 [1], we only started using migration regularly in the late fall of 1988. We had trouble getting migration to work because we were trying to hit a moving target: the rest of the system was evolving rapidly, and before we were to put migration into general use, changes to the rest of the system made it unusable. As we understand it, other systems have experienced similar problems with process migration because it interacts intimately with so many parts of the kernel [4].

In retrospect, our greatest mistake was to fail to put process migration into general use at the first opportunity: there was a window of time when we potentially could have started using migration (if we had been prepared with suitable user-level utilities to manage such things as host selection). Now that migration is used routinely for remote execution, we should discover quickly if anything should cause it to stop working. However, we could have avoided some poor design decisions—from the perspective of migration—elsewhere in the system if their impact had been apparent earlier.

Our greatest success was to put a fully transparent, preemptable form of remote execution into general use. Sprite's process migration system has a number of advantages over a less transparent form of remote execution such as *rexec*:

- Remote processes appear in a list of processes on the home machine, so the user need not be aware of where the processes are physically executing. One may perform operations on processes regardless of their location, such as sending them signals.

- Processes may be moved at any time. not only at specific times such as *fork* and *exec* (e.g., LOCUS [3]) . In an environment in which "eviction" is an issue, this generality is important.

- Migration is transparent enough to let nearly any program run on multiple hosts during its lifetime. (Exceptions include the X window system display server and the user-level process that interfaces Sprite to the internet.) A process behaves in all respects as though it executes on a single host. Also, programs do not need to be coded specially to take advantage of migration.

We look forward to using process migration regularly and plan to evaluate its performance more quantitatively in the near future.

References

[1] F. Douglis and J. Ousterhout. Process migration in the Sprite operating system. In *Proceedings of the 7th International Conference on Distributed Computing Systems*, pages 18–25, Berlin, West Germany, September 1987. IEEE.

[2] M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, 6(1):134–154, February 1988.

[3] G. J. Popek and B. J. Walker, editors. *The LOCUS Distributed System Architecture.* Computer Systems Series. The MIT Press, 1985.

[4] M. Theimer. Personal communication.

[5] M. Theimer. *Preemptable Remote Execution Facilities for Loosely-Coupled Distributed Systems.* PhD thesis, Stanford University, 1986.

[6] E. Zayas. Attacking the process migration bottleneck. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 13–22, Austin, TX, November 1987.