# Cross-Operating System Process Migration On a Massively Parallel Processor

Dimitrios Katramatos  Steve J. Chapin*
Dept. of Computer Science
University of Virginia
Charlottesville, VA 22903–2442
chapin,dk3x@cs.virginia.edu

Patricia Hillman
Dept. of Math. and Computer Science
Kent State University
Kent, OH 44242–0001
phillman@mcs.kent.edu

Lee Ann Fisk  David van Dresser†
Massively Parallel Computing Research Lab
Sandia National Labs
Albuquerque, NM 87185
lafisk@cs.sandia.gov

**Abstract**

As part of the Uniform Kernel project at Sandia National Labs, we developed process migration mechanisms for the Intel Paragon and Teraflop (ASCI Red) machines. The computing paradigm on these massively parallel processors divides the nodes into multiple logical partitions which run differing operating systems. This provides a unique environment for process migration, with homogeneous hardware and heterogeneous systems software. In this paper, we describe mechanisms that allow processes to (1) run on multiple operating systems, and (2) migrate between partitions on demand. The result is the first step towards a system which will allow partition boundaries to change dynamically, thus responding to changes in load mix and resource usage on the machine.

Keywords: process migration, heterogeneity, parallel systems

*This work was sponsored in part by DOE contract AR-8658.
†The Uniform Kernel project was supported by ARPA order B901.

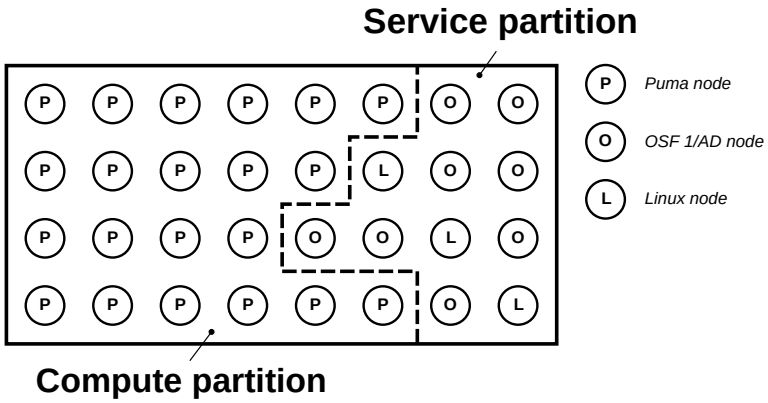


Figure 1: Node partitioning for an MPP

# 1 Introduction

Process migration for homogeneous machines has been implemented in several systems, including Condor [?], V [?], and Linux [?]. With the advent of heterogeneous computing, interest has increased in providing migration for systems with differing hardware or operating system software. Various motivations have been identified for including a migration mechanism in an operating system [1, 2, 18, 26]. In our case, we are primarily interested in two benefits: the utilization of operating system capabilities not available on the current node, and supporting fault resilience by migrating processes away from nodes that may have suffered a partial failure.

The Intel Paragon and Teraflop supercomputers are massively parallel multiprocessors (MPP) which divide their nodes into logical partitions: a service partition providing traditional operating system services such as file systems and internetworking, and a compute partition with an operating system tuned for performance at the expense of such functionality. On the original systems, these partitions were staticly created at boot time, and processes would execute on a single partition for their entire lifetimes. The Uniform Kernel project at Sandia National Labs investigated mechanisms to remove these restrictions, so that the nodes could change their partition membership and processes could migrate between partitions at run-time. As noted in Tritscher and Bemmerl [3, 4], an MPP can benefit from the presence of a process migration mechanism by using it to address issues like repartitioning of the nodes and fault resilience and to exploit its own special capabilities.

As part of our work on the Uniform Kernel project, we have developed mechanisms that allow a program to migrate between the partitions of the Intel Teraflop (ASCI Red) machine, located at Sandia National Labs in Albuquerque (see figure 1. The compute partition of the Teraflop machine runs Cougar, Intel's version of the Puma operating system developed in the Massively Parallel Computing Research Lab at Sandia. The service partition runs a version of OSF/1 AD called TOS, as well as a port of Linux performed at the MPCRL. In this paper, we will describe mechanisms supporting migration of processes from Puma to Linux.

Section ?? describes the salient features of the two operating systems. We outline the mechanisms and their implementations in section ??, and describe related work in section ??. Section ?? contains concluding remarks and description of future work.

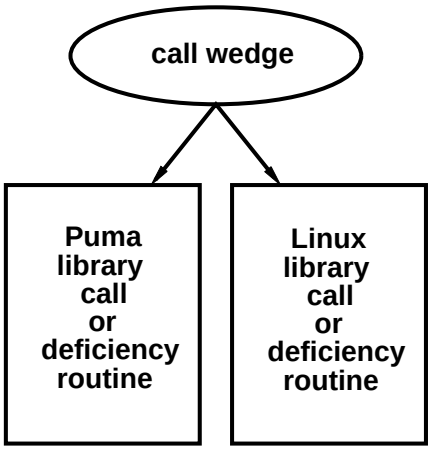


Figure 2: Function of a call wedge

## 2 The Unified Library

If we are to migrate a process from one partition to another, it must be able to execute on either partition. However, the operating systems on the separate partitions do not support the same API. Therefore, we developed a *Unified Library* which could be linked with a proram to produce an executable that can run on either partition.

The unified library contains both OSF library code and Puma library code. It is build by a special utility program which resolves name conflicts and inserts special code portion, called a call wedge, in each call included in the library. The call wedge directs a call to either the OSF routine or the Puma routine according to the value of a global flag included in the executable image, showing which operating system the image executes under (see figure **??**).

Upon entry in a procedure—even in the case of main()—the contents of a flag are checked. The flag indicates whether the process is running over Linux, OSF/1 AD, or Puma. The call wedge uses the flag value to select a branch of a switch for execution. If the current operating system supports the call, then its local implementation is used. If the call is unsupported, then a deficiency routine is called, which by default prints an error message and aborts the process. For our purposes, instead of aborting, the deficiency routine will invoke the migration operation. The call wedge is written is such a manner that it does not disturb the stack while choosing which routine to branch to. Therefore it does not affect the calling arguments.

## 3 Overview of the Operating Systems

### 3.1 The Puma Operating System

Puma is a message-passing multi-processor operating system developed by Sandia National Laboratories and the University of New Mexico. It is the successor to SUNMOS (Sandia / UNM Operating System). Like many of the operating systems developed for distributed processing, such as Amoeba, Chorus, Mach, and V, the Puma architecture is based on a message passing kernel. However, Puma has been developed for an environment in which the communication network is trusted and controlled by the kernel. In such an environment there is no need to authenticate messages and therefore many of the functions that the kernel or an application process needs to perform are simplified [15].

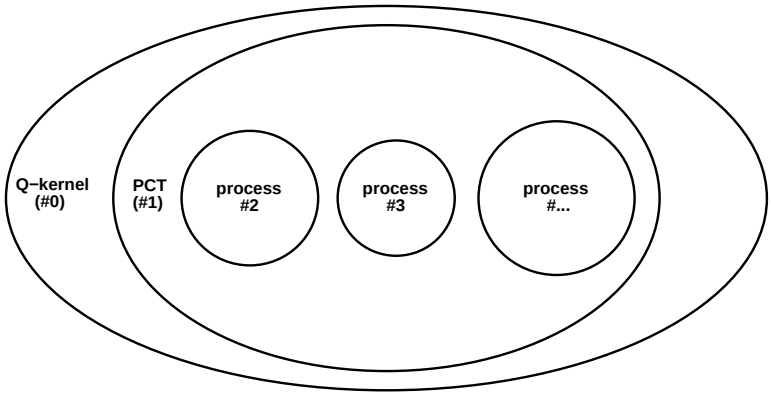


Figure 3: Levels of trust in Puma

The Puma architecture is based on three levels (see figure 2). The quintessential kernel (QK), which provides basic communication facilities and address space protection, is the lowest level. The process control thread (PCT) occupies the next level. The PCT provides process management facilities and group level protection. The server and application processes comprise the third level in the architecture. Kernels trust only the underlying hardware and the kernels on other nodes. Each PCT trusts only the hardware, kernels, and other PCTs. User-level processes trust everyone except other user-level processes. Thus, a malfunctioning process cannot corrupt the PCT's or the kernel's structures.

The QK supports the minimal set of functionality that requires execution in supervisor mode, such as handling interrupts and hardware faults. Additionally, the QK performs supervisor mode services on behalf of the PCT (and user processes), such as message dispatch and reception, context switching, virtual address manipulation, and running a process. It does not, however, perform management functions; that is the role of the PCT. It has read/write access to all memory in user space and is in charge of managing all operating system resources, including process loading, job scheduling, and memory management. In addition, the PCT may initiate contact with an available server on behalf of a user process. The QK and the PCT work together to provide a complete operating system [16].

Some attractive features result by splitting operating system functionality between the QK and the PCT. These include fault tolerance and multiple resource management policies. Software fault tolerance is realizable due to the small size and well defined functionality of the QK. Even if the PCT of a node faults the QK can remain running and the node can be reactivated by reloading the PCT. Multiple resource management policies are possible because, although there can be only one PCT per node, one may have several PCTs to choose from, each one having a different management policy.

Puma message passing is based on a new concept known as a portal [15, 16], which avoid delays due to buffer copying and switches between user and kernel modes. A portal is essentially an opening in a process's address space from which data can be directly read or to which data can be directly written using message passing. On an Intel Paragon, inter-node communication rates in excess of 160MB/sec have been achieved in contrast to just 55MB/sec when memory copies are required [15].

Another interesting feature of Puma is that it maintains various data structures about a process—including message passing structures— in user space. This gives a process the flexibility to manipulate these structures without having to pay the cost of trapping to the

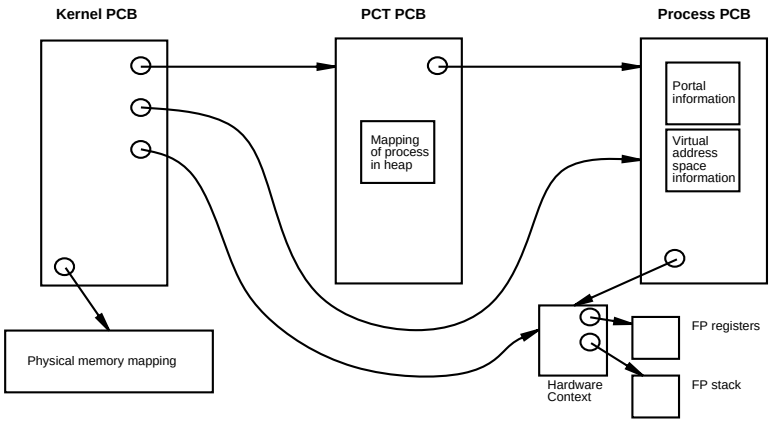


Figure 4: Puma process representation

kernel. As it is intended to be lightweight and fast, Puma does not have support for demand paging. Instead, Puma loads the whole executable image in memory.

Puma runs in the compute partition and is not stand-alone. Nodes are booted with Puma kernels via a special utility. Also, the PCTs and applications are loaded on the nodes with the help of another special utility, yod. Yod performs mesh allocation, and program load and execution, and additionally provides file I/O including stdin, stdout, and stderr. Puma does not have direct access to a filesystem nor does it support all operating system services found in a general-purpose operating system.

More information and publications about Puma and SUNMOS can be found at Sandia's web server: *http://www.cs.sandia.gov*.

### 3.2 Puma Process State

Puma supports the classic process concept, i.e. an address space with a single thread of execution. Because Puma does not have support for demand paging, the entire executable image of an active process resides in the physical memory of the node. This feature is advantageous during the desired migration because the process image is readily available. The Process Control Block (PCB) containing the process state is kept in the process's address space, and is shared with the QK and the PCT. This approach has the advantage that a process can access and modify its own data structures without having to issue kernel traps, but also raises the issues of reliability—the QK needs to validate the address values obtained from a shared structure—and of using atomic actions on the shared structures [5]. Figure 3 gives a simplified view of the Puma process representation. The QK also maintains a small amount of additional information about the state of the process, e.g. the register context.

The address space for a Puma process has an address range from 0x00000000 to 0x40000000 (i.e. from 0 to 1GB). A typical address space is shown in figure **??**.

### 3.3 The Linux Operating System

Linux is a freely available operating system for PCs. It is compatible with the POSIX 1003.1 standard and includes large areas of the functions of UNIX System V and BSD 4.3 [17, 10]. The operating system's source code is distributed under the GNU public license, thus everyone can use, copy, and modify the code freely.

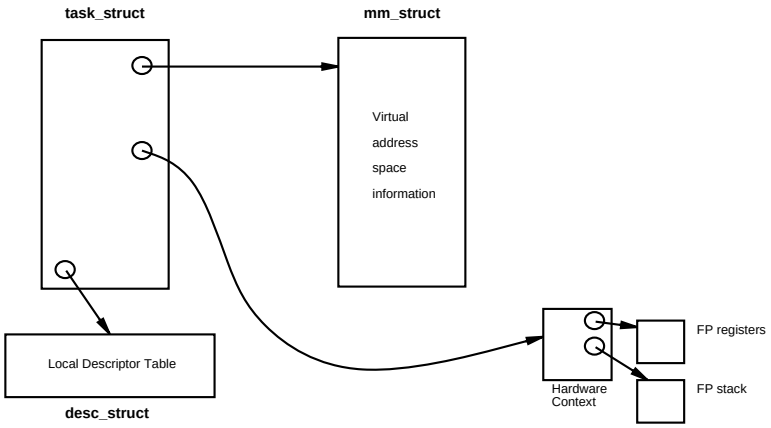


Figure 5: Linux process representation

Linux includes traditional operating system services such as multi-tasking support, demand-loading of executables with copy-on-write, demand paging, and support for TCP/IP protocol suite. The operating system's source code and additional documentation and utilities can be found at *http://sunsite.unc.edu/pub/Linux*. Linux version 2.0.0 was used in the present work.

### 3.4 Linux Process State

Linux, like Puma, supports the traditional definition of a process in that a process is an address space containing a single thread of execution. A simplified view of the Linux process representation is given in figure 4.

Once a process has been created by the fork() system call, it inherits a page directory that allows access to 1KB of page tables. The page tables in turn point to 1MB of 4KB pages which totals 4GB of memory. The user area of memory extends from 0 to 3GB and the area above 3GB is kernel space. The process's view of memory is given in figure **??**.

### 3.5 Mapping the Puma Process State onto a Linux Process State

In the computer system under consideration all processor nodes are of the same architecture—Pentiums—but are organized in a multiple-partition scheme with each partition running a different operating system. There is an obvious advantage to this case because it is certain that at the lowest level there will be compatibility—the instruction sets are the same. The same executable image runs on both operating systems with the use of a special library containing call wedges. Therefore data layouts and code sequences are not complicating factors.

A quick review of the process state in Puma and the task structure in Linux reveals that although they maintain similar information about a process—the virtual memory image, the process group, the register context, etc.—no obvious correspondence exists. There are pieces of information that are specific to one operating system or the other, such as the trap calling parameters structure in Puma. Other data serve a similar purpose, e.g. process id's, but are not directly corresponding. The only exceptions are the structures that maintain the register context of a process during a context switch. These structures will be the basis for the partial mapping of the process states between the two operating systems. A full

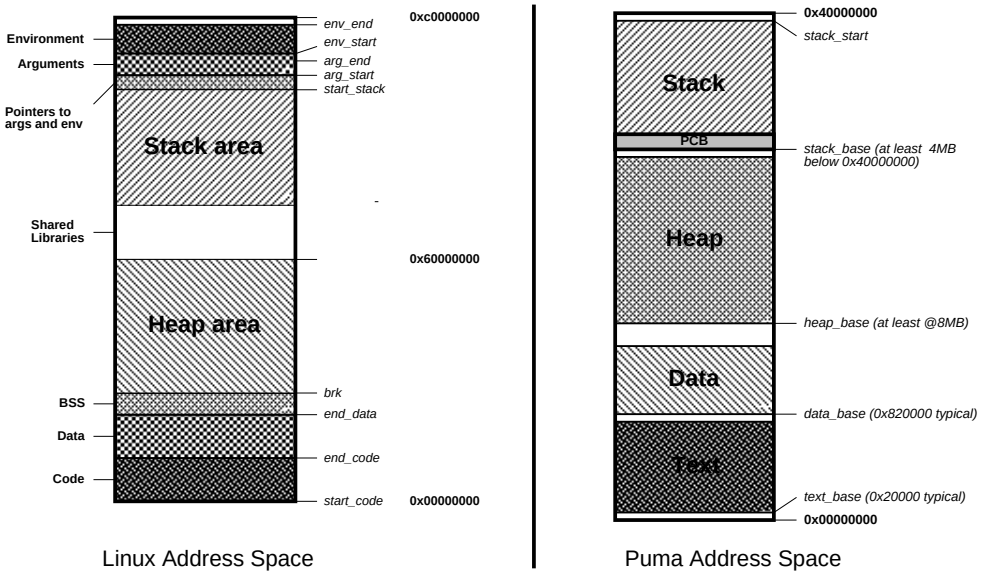


Figure 6: The address space of Linux and Puma processes

mapping is not necessary because the procedure followed by both systems during process creation includes the phase of initialization of basic data structures with default values. Only the information that can be directly mapped needs to be plugged into the corresponding members of the Linux structures to yield a process state representation in Linux suitable for the migrating process. The remaining members can be allowed to retain their default values.

Because of the homogeneous underlying architecture, we decided that a direct conversion of the process states for the direction Puma→Linux was more appropriate. This decision does not preclude a future implementation of the migration mechanism using an independent process state representation analogous to the external data representation for networking.

By comparing the address space that a process is given under Puma with that under Linux it is clear that a Puma process's address space can always fit in a Linux process's address space because it is smaller. The actual size of the address space for a Puma process is limited by the size of the physical memory available to the node since there is no demand paging support in Puma. Also, although a Linux process's view of memory is different in the general case than the view a Puma process has, there are certain similarities such as the type and order of the segments. Also, there are no strict restrictions, in the address range 0x00000000 to 0x60000000, imposed on the base addresses and sizes of the segments. It is therefore possible to plug the segments of the Puma process in a Linux address space at the exact same locations where they used to be in the Puma address space. This fact guarantees that the process image will be reconstructed exactly the way it was; there will be no need for translations etc. which would complicate the problem. Actually, the placement of a segment such as the stack segment in a different address range without proper translation would cause erratic binding of contents to all pointer references and thus a meaningless stack would occur. No such problems are encountered in the present case.

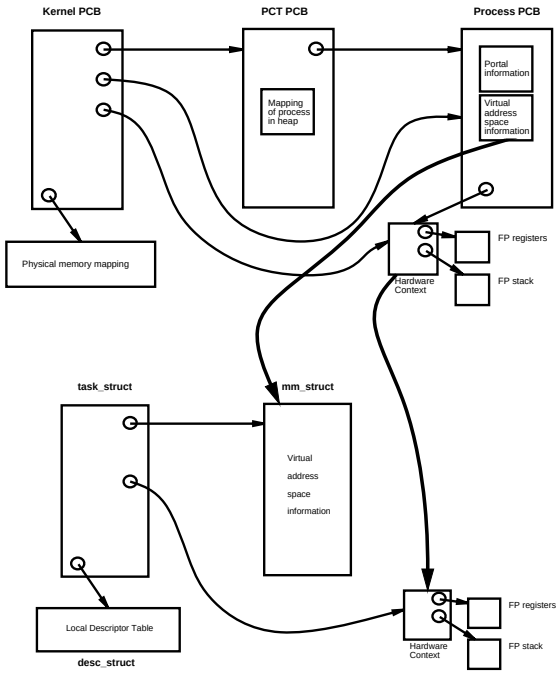


Figure 7: The mapping

# 4 Related Work

Process migration has been implemented as a feature of several operating systems [26, ?]. A process state is represented by two components: the initial state—the initial executable image of a process before execution begins—and the changes that have occurred due to execution. Process migration mechanisms follow variations of the following general steps:

1. The process is frozen and its state is saved.
2. The saved state is transferred from the source to the destination processor and resources are released at the source system.
3. References to the process are rebound and the process is restarted.

Heterogeneity has not been addressed in most earlier migration implementations. This was not a significant limitation because most of the work was conducted for workstation clusters typically consisting of compatible machines. However, to fully utilize current systems that are heterogeneous in either hardware or system software requires a heterogeneous migration mechanism.

Shub [18, 19] presents a "fat binary" approach to the heterogeneous migration issue. His prototype mechanism is implemented under the V system [24] which already supports homogeneous migration, and focuses on a restricted class of C programs with no unions. A process consists of a single address space with a single thread of control and a migration is requested by a process itself. The translation techniques used by Shub are based on a combination of translation procedures and the availability of a single compiler that can target every machine in the system. The translation procedures use the symbol tables generated by the compiler and a heap storage management package to find all data elements of a process and translate them to the representation of the target machine.

Theimer and Hayes [20] propose a different approach. In effect, the running state of a process is translated from its representation on one machine to an equivalent, machine-independent source code representation which is then recompiled on the destination machine. When compiled and executed on the target machine the program reconstructs the state of the process and then allows the continuation of the process's execution. This method avoids complications from details of machine and compiler-dependent data translation procedures, but at a significant cost in migration time.

One of the most recent works on heterogeneous migration is the Tui system by Smith and Hutchinson [1]. Tui is a prototype that is able to migrate type-safe ANSI-C programs between four different architectures: Solaris executing on a SPARC processor, SunOS on an m68020, Linux on an i486 and AIX on a PowerPC. A program is considered to be type-safe if it is possible to uniquely determine the type of each data value within the program. During migration a special program on the source node fetches the memory of the migrating process, scans and locates all data values, converts them into an intermediate form, and writes them on disk. On the destination node, another special program uses the corresponding executable for the destination node's architecture to extract the text segment and data type information and along with the intermediate file generated in the previous phase reconstructs and restarts the process.

Jul and Steensgaard [**?**] describes the Heterogeneous Emerald system, which supports migration of objects between heterogeneous machines running the Emerald run-time system. By the nature of the Emerald programming language, programs are guaranteed to be type-safe.

Process Introspection [**?**] uses an augmented compiler that it automatically inserts library calls at migration points. The library calls check to see if migration is called for, and if so save the state of the process, including the call stack, in a platform-independent format. When the process is restarted, the library routines rebuild the proper state from the saved information.

Our work focuses on the issue of enabling process migration—initially one-way—from a multiprocessor operating system, Puma, to a general-purpose UNIX-compatible operating system that both run on the same CPU architecture. We can take advantage of the homogeneous architecture to avoid translation issues due to different instruction sets and/or data representations, allowing for significant simplification of the mechanisms. In addition, because the internal state can be treated as a "black box," we can ignore issues of type safety. In contrast to all of the above methods except Tui, the source and destination operating systems are different and do not share a common run-time system. Therefore, we must translate the operating system-dependent metadata, such as page tables and process table entries, describing the state of the process.

The steps followed by the migration mechanism obey the general three steps described above. A process state has to be extracted, modified, copied from source to destination and then reconstructed and restarted. The compiled binary image is transferred from source to destination and restarted. We use two special programs, one on the source node and one on the destination node, to perform a migration, as seen in the Tui approach. Few changes are necessary to a process's image to correctly reconstruct and restart the process due to the homogeneity of the underlying hardware.

## 5 Conclusions and Future Work

Puma and Linux are based on different philosophies. Linux is a UNIX-compatible general-purpose operating system and provides all typical operating system services. Puma, on the other hand, is a lightweight and fast operating system designed to run on the nodes of a massively parallel processor and allow processes to exploit the memory and the communication capabilities of the nodes. However, it lacks basic operating system services.

The versions of Puma and Linux for the x86 family of processors can run the same executable image if a special library—a unified library—is used during the linking phase of the compilation. The library uses call wedges to redirect system calls to the appropriate routines according to the operating system under which the image executes. The call wedge mechanism can be used as the basis for a mechanism to migrate processes between operating systems.

During preliminary testing certain important elements of the support environment, such as a unified library for x86 machines, a Linux portal interface [11], and a Linux kernel port to an MPP node, were still under development. Therefore the experimental migration mechanism implementation and the testing program had to be adapted to the environment available at the time. The successful test of the mechanism proved that migration from Puma to Linux is possible. As the remaining underlying functionality is added to the system, the migration mechanisms will be extended commensurately. In particular, we will add a marker-based communication forwarding mechanism (cf. the Chandy-Lamport global state recording algorithm [13]), as well as investigating methods to reduce migration time through analysis of address space characteristics (e.g. empty pages) and through on-the-fly compression.

## Acknowledgements

The authors are indebted to the Puma team at the MPCRL and the University of New Mexico who worked on the Uniform Kernel project, including Nicholas Droux, David Greenberg, Michael Levenhagen, Barney Maccabe, Rolf Riesen, and Brian Sanchez.

## References

[1] Peter Smith, Norman Hutchinson. (February, 1996). Heterogeneous Process Migration: The Tui System. *Technical Report, 96-04, University of British Columbia.*

[2] Eric Jul, Henry Levy, Norman Hutchinson, Andrew Black. (February 1988). Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems.*

[3] S. Tritscher and T. Bemmerl. (February, 1992). Seitenorientierte Prozessmigration als Basis fuer Dynamischen Lastausgleich. *GI/ITG Pars Mitteilungen, no 9* pages 58–62.

[4] R. Zajcew et al. (January 1993). An OSF/1 UNIX for Massively Parallel Multicomputers. *Proceedings of the Winter USENIX Conference*, pages 449–468.

[5] Puma Quintessential Kernel Guide.
*http://www.cs.sandia.gov/~rolf/puma/puma_os/qk_guide/qkguide1.html.*

[6] Executable and Linkable Format (ELF).
Tool Interface Standards

Portable Formats Specification, Version 1.1 .
*http://sunsite.unc.edu/pub/Linux/GCC/elf.doc.tar.gz*

[7] Gerald Q. Maguire, Jr., and Jonathan M. Smith. (March, 1988). Process Migration: Effects on Scientific Computation. *SIGPLAN Notices, Vol 23, No 3*, pages 102–106.

[8] Building an OSF/Puma Unified Executable.
*http://www.cs.sandia.gov/~rolf/puma/unified/unidoc_2.html.*

[9] Puma User's Guide.
*http://www.cs.sandia.gov/~rolf/puma/puma_os/usr_guide/usrguide_toc.html.*

[10] M. Beck, H. Boehme, M. Dziadzka, U. Kunitz, R. Magnus, D. Verworner. Linux Kernel Internals. *Addison Wesley Longman Limited, England, 1996.*

[11] Puma Library Writer's Guide.
*http://www.cs.sandia.gov/~rolf/puma/puma_os/lib_writer/libwriter_toc.html.*

[12] Puma PCT Writer's Guide.
*http://www.cs.sandia.gov/~rolf/puma/puma_os/pct_writer/pctwriter_toc.html.*

[13] Mukesh Singhal, Niranjan G. Shivaratri. Advanced Concepts in Operating Systems. *McGraw Hill, USA, 1994.*

[14] Puma Overview.
*http://www.cs.sandia.gov/~rolf/puma/puma_os/overview.html.*

[15] Stephen R. Wheat, Arthur B. Maccabe, Rolf Riesen, David W. van Dresser, and T. Mack Stallcup. (1994). PUMA: An Operating System for Massively Parallel Systems. *Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences*, pages 56–65. IEEE Computer Society Press, 1994.

[16] Lance Shuler, Chu Jong, Rolf Riesen, David W. van Dresser, Arthur B. Maccabe, Lee Ann Fisk, and T. Mack Stallcup. (June, 1995). *Proceedings of the Intel Supercomputer User's Group. 1995 Annual North America User's Conference.*

[17] Matt Welsh and Lar Kaufman. Running Linux. *O'Reilly & Associates, Inc., USA, 1995.*

[18] Charles M. Shub. (February, 1990). Native Code Process-Originated Migration In A Heterogeneous Environment. *Proceedings of the 1990 Computer Science Conference*, pages 266–270.

[19] F. Brent Dubach, Robert M. Rutherford, Charles M. Shub. (February, 1989). Process-Originated Migration in a Heterogeneous Environment. *Proceedings of the 1989 Computer Science Conference*, pages 98–102.

[20] Marvin M. Theimer and Barry Hayes. (May, 1991). Heterogeneous Process Migration by Recompilation. *IEEE 11th International Conference on Distributed Computing Systems, Arlington, Texas*, pages 18–25.

[21] M. L. Powell and B. P. Miller. (October, 1983). Process migration in DEMOS/MP. *Proceedings of the 9th Symposium on Operating systems Principles*, pages 110–119.

[22] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch. (February, 1988). The Sprite Network Operating System. *IEEE Computer*, pages 23–26.

[23] A. Barak and A. Litman. (August, 1985). MOS: a Multicomputer Distributed Operating System. *Software—Practice and Experience*, 15(8):725–737.

[24] D. R. Cheriton. (March, 1988). The V Distributed System. *Communications of the ACM*, 31(3):314–333.

[25] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. (Summer, 1986). Mach: A New Kernel Foundation for UNIX Development. *Proceedings of the Summer USENIX Conference*, pages 93–112.

[26] M. J. Smith. (July, 1988). A Survey of Process Migration Mechanisms. *Operating Systems Review*, 22(3):28–40.

[27] D. L. Eager, E. D. Lazowska, J. Zahorjan. (May, 1988). The Limited Performance Benefits of Migrating Active Processes For Load Sharing. *Proceedings of the 1988 Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 63–72.