A Process Migration Subsystem for Distributed Applications: Design & Implementation

by

Syed Khaja Naseer

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER SCIENCE

January, 1996

A PROCESS MIGRATION SUBSYSTEM FOR DISTRIBUTED APPLICATIONS: DESIGN & IMPLEMENTATION

BY

SYED KHAJA NASEER

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE
In

COMPUTER SCIENCE

JANUARY 1996

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS
DHAHRAN, SAUDI ARABIA

COLLEGE OF GRADUATE STUDIES

This thesis, written by **SYED KHAJA NASEER** under the direction of his Thesis Advisor and approved by his Thesis Committee, has been presented to and accepted by the Dean of the College of Graduate Studies, in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE** in **COMPUTER SCIENCE.**

THESIS COMMITTEE

[illegible]
Dr. M. Bozyigit (Chairman)

[illegible]
Dr. S. Ghanta (Member)

[illegible]
Dr. M. AlMulhem (Member)

[illegible]
Dr. K. S. AlTawil (Member)

[illegible]
Department Chairman

[illegible]
Dean, College of Graduate Studies

6-3-96
Date

عمادة الدراسات العليا
COLLEGE OF GRADUATE STUDIES

My

Parents

Sisters & Brothers-In-Law

ACKNOWLEDGMENT

In the name of Allah, Most Gracious, Most Merciful.

022.005 *O Mankind! If ye have a doubt about the Resurrection, (consider) that We created you out of dust, then out of sperm, then out of a leech-like clot, then out of a morsel of flesh, partly formed and partly unformed, in order that We may manifest (Our power) to you; and We cause whom We will to rest in the wombs for an appointed term, then do We bring you out as babes, then (foster you) that ye may reach your age of full strength; and some of you are called to die, and some are sent back to the feeblest old age, so that they know nothing after having known (much), and (further), thou seest the earth barren and lifeless, but when We pour down rain on it, it is stirred (to life), it swells, and it puts forth every kind of beautiful growth (in pairs).*

022.006 *This is so, because God is the Reality: it is He Who gives life to the dead, and it is He Who has power over all things.*

022.073 *O Men! Here is a parable set forth! listen to it! Those on whom, besides God, ye call, cannot create (even) a fly, if they all met together for the purpose, and if the fly should snatch away anything from them, they would have no power to release it from the fly. Feeble are those who petition and those whom they petition!*

022.074 *No just estimate have they made of God: for God is He Who is strong and able to Carry out His Will.*

(The Holy Quran)

All Praise is due to ALLAH to whom belongs the dominion of the Heavens and the Earth. Peace and mercy be upon His Prophet. I thank Him for giving me the knowledge and patience to carry out this work.

Acknowledgment is due to King Fahd University of Petroleum and Minerals for support of this research.

First and foremost, I would like to express my humble gratitudes to my parents, without whose blessings, support, encouragement and motivation, I wouldn't have been what I am. I also thank my dearest sisters and brothers-in-law for their constant impetus and backing. I thank ALLAH (S.W.T) for blessing me with such a caring and affectionate family.

There have been several who influenced my work and life positively, and I consider myself fortunate to have known them. Foremost among them is my thesis advisor - Dr. Muslim Bozyigit, who has been fully involved in this research right from day one. He was always available to me, even during the peak times of the semester. I really enjoyed the numerous discussions that we had and I am very grateful to him for his patience, support and motivation. Especial thanks are due to my committee members Dr. Ghanta, Dr. Khalid, and Dr. Mulhem for their active support and invaluable suggestions.

I offer my humble gratitudes to my dear friends Nisar, Kaleem, Ather, Hadi, Feroze, Sajjad, Raheem, Atif and Azhar for their direct/indirect contributions to this work and for making my stay at KFUPM a pleasant experience. Special thanks are due to the my academic advisor Dr. Ghanta for his guidance and expert counsel. I would also like to express my thanks to the faculty and staff of the Information & Computer Science department - Dr. Mulhem (Chairman), Dr. Shafeeq, Dr. Sukairi, Dr. Yousef (COE), Mr. Yazdani, Mr. Suhaib, Mr. Husni, Mr. Said, Mr. Garout and Mr. Shahid, for providing an enjoyable work atmosphere.

Contents

List of Figures

List of Algorithms

THESIS ABSTRACT

Name:	SYED KHAJA NASEER
Title:	A PROCESS MIGRATION SUBSYSTEM FOR DISTRIBUTED APPLICATIONS: DESIGN & IMPLEMENTATION
Degree:	MASTER OF SCIENCE
Major Field:	INFORMATION & COMPUTER SCIENCE
Date of Degree:	JANUARY 1996

*Workstation-based Distributed Computing Systems (DCS) are getting popular in both academic and commercial communities due to the continuing trend of decreasing price/performance ratio of workstations and the rapid development of networking technology. However, the actual work load on individual workstations is usually diverse, and in some, the load may be much lower than their computing capacity. As a result, some workstations would often be under-utilized, while the others are overloaded. A preemptive process migration facility can be provided, in such a distributed system, to dynamically relocate executing processes among the component machines. A migration based relocation can help cope with dynamic fluctuations in load and service needs, meet real-time scheduling deadlines, bring a process to a special device, or improve the system's fault tolerance. Such a facility, however, has not been available in the context of conventional operating systems such as Un*x. Most of the work on providing a process migration facility has been on limited-domain distributed operating systems.*

This study outlines the design and implementation of a stand-alone Process Migration Subsystem (PMS) that provides the facility of migrating processes executing on a DCS. The subsystem can handle migration of independent processes as well as processes belonging to distributed applications. The communication among such processes is maintained transparently even after migration of any of the component processes.

Keywords: Process Migration, Checkpointing, Fault-Tolerance, Dynamic load-balancing.

King Fahd University of Petroleum and Minerals, Dhahran.
January 1996

خلاصة الرسالة

اسم الطالب الكامل : سيد خاجا نصير

عنوان الدراسة : نظام فرعى لتهجير العمليات للتطبيقات الموزعة: تصميم وتطبية

الدرجة : ماجستير في العلوم

التخصـص : معلومات وعلوم الحاسب الآلي

تاريخ الشهادة : يناير ١٩٩٦

أنظمة الحاسب الآلي الموزعة المعتمدة على المحطات الطرفية باتت منتشرة في المجالين الأكاديمي والتجاري وذلك نتيجة لاستمرار إنخفاض نسبة السعر / الأداء في المحطات الطرفية ، وأيضاً نتيجة التطور يكون عادةً متفاوت ، وأحياناً يكون السريع في تقنية الشبكات . ولكن حمل العمل الحقيقي على المحطة الطرفية أقل كثيراً من القوة الحسابية للمحطة الطرفية . ونتيجة لذلك ، بعض المحطات الطرفية تكون غير مستغلة بما فيه الكفاية ، بينما تكون المحطات الأخرى محملة فوق الطاقة . في حالة إستخدام " تهجير العمليات بالشفعة " في مثل هذه الأنظمة الموزعة لإحلال العلميات تحت التنفيذ بشكل متغير على المحطات الطرفية المكونة للنظام . عمليات التهجير المبنية على إعادة الإحلال تساعد على التعامل مع التذبذب في الأحمال والإحتياج للخدمات ، تحقيق المواعيد النهائية لجدولة العمليات الحقيقية ، وضع عملية ما على جهاز خاص ، وتحسين إمكانية النظام هذا النوع من التهجير لم يتوفر بعد في بعض أنظمة التشغيل مثل اليونكس . معظم .على إحتمال الأخطاء العمل في هذا المجال كان في أنظمة التشغيل الموزعة ذات النطاق المحدود.

هذه الدراسة تعطي التصميم والتطبيق لنظام فرعي مستقل لتهجير العمليات ، والذي يعطي القدرة على تهجير العمليات المنفذة على أنظمة الحاسب الآلي الموزعة . هذا النظام بإمكانه القيام بتهجير العمليات المستقلة وتلك التي تنتمي إلى تطبيقات موزعة . عملية الإتصال بين مثل هذه العمليات تبقى. موجدة حتى بعد تهجير أي عملية.

كلمات البحث : تهجير العمليات ، تحمل الأخطاء ، موازنة الحمل المتغير، نقطة المقارنة.

درجة الماجستير في العلوم

جامعة الملك فهد للبترول و المعادن

الظهران ، المملكة العربية السعودية

يناير ١٩٩٦

Chapter 1

Introduction

A *Distributed Computing System (DCS)*[1] is defined as a computing system consisting of at least two autonomous processors connected by a data communication network, that appear to the users of the system as a single computer [5]. It is based on the MIMD[2] architectural paradigm, wherein each processor executes instructions independent of the others. Thus, a DCS can be viewed as a virtual MIMD computer which has a different program running on every node, or possibly several different programs on every node if multiprocessing is involved. Users of a DCS are to be given the impression that they are using a single, integrated computing facility, although the facility is actually provided by more than one computer and the computers may be at different locations. using the network to coordinate their work and to transfer data among them. The limiting example of such a distributed computing system is the *Internet*, comprising of tens of thousands of computers ranging from PCs to Mainframes to Supercomputers, interconnected by the global Inter-Network.

[1]In this research, a DCS refers to a loosely-coupled distributed system
[2]Multiple Instruction - Multiple Data stream

In a DCS, resources may be shared by many processes; examples of shared resources include files. printers, and CPUs. This sharing of resources may lead to resource contention. as many processes try to access a particular resource at the same time. Access to resources need to be scheduled so as to avoid conflict and to maximize resource utilization. In a typical DCS, it is possible that some processors are assigned more work than others. Therefore, it is desirable for the workload in a DCS to be evenly distributed to maximize the CPU utilization and minimize the average response time. Load balancing algorithms try to assign the tasks to processors in such a way that the load on each processor is approximately the same. Thus, in a DCS, *Dynamic Scheduling* and *Load Balancing* play an important role.

Almost all the algorithms on Dynamic Scheduling and Load-Balancing that have been proposed in the literature [21] rely on some form of process migration facility. *Process Migration* can be defined as the ability to move a process's execution site, at any instant of time. from a *source* machine to a *destination* machine of the same (or different) architecture. A preemptive process migration facility can be provided, in a distributed system. to dynamically relocate running processes among the component machines. Such relocation can help cope with dynamic fluctuations in loads and service needs, meet real-time scheduling deadlines, bring a process to a special device, or improve the system's fault tolerance.

However, such a facility is not common in distributed operating systems, due largely to the inherent complexity of providing such facilities and the potential execution penalty if the migration policy and mechanism are not tuned correctly. There are several reasons why migration is hard to design and implement. The mechanism for moving processes must reliably and efficiently detach a migrant process from its source environment, transfer it with its context (the per-process data structures held in the kernel) and attach it to a new environment on the destination machine. This close association of the migration mechanism with the basic functions of the operating system, such as process scheduling, memory management and interprocess communication, has been the major bottleneck in providing a migration facility in conventional Un⋆x operating systems.

Inherent in the definition of process migration is the concept of checkpointing. In order to stop, transfer, and restart an executing process from one machine to another, we need to have some form of checkpointing and restoration mechanism. *Checkpointing* and rollback recovery is a standard technique proposed in providing a fault-tolerant computing environment [26]. The goal of checkpointing, here, is to establish a *recovery point* in the execution of the program, and save enough information to reconstruct the state of the program at this recovery point in the event of a failure. To restore this recovery point, one needs to retrieve the memory image of the process, setup the relevant process/kernel data structures, restore the state of the CPU registers and enable execution of the process with the new process image.

Research pertaining to classical checkpointing and rollback recovery in database systems is heavily documented. However, research on migration in the context of distributed systems has been very scarce. Most of the work on providing a remote execution facility has been in custom-made distributed operating systems such as Remote Unix [11], Butler [14], Condor [12], and Amoeba [22]. Further, there is little, if any, work on using checkpointing and process migration as a means of providing fault-tolerance in Un⋆x based distributed systems.

1.1 Workstation Based Distributed Environments

Workstation-based distributed computing environments are getting popular in both academic and commercial communities due to the continuing trend of decreasing cost/performance ratio and rapid development of networking technology. A workstation-based DCS is a Computer system in which most machines are autonomous personal multitasking workstations, each dedicated primarily to serving its local user, interconnected by high-speed local area networks (Figure 1.1). In most of these environments, workstations are assigned ownership to individual users to guarantee the privilege of their processing demands. However, the work loads on these workstations are usually much lower than their computing capacity, especially with the ever-increasing computing power of new hardware. As a result, the resources of such workstations are often under-utilized and many of them are frequently idle. Nevertheless, the demands on extra computing power would never stop. There are always cases where users would like to take all the advantage of any available CPU cycles on idling workstations for jobs that could not be processed on their own machines. In applications known as *Grand Challenges*, the continuity of the tasks is vital as restarting can be very expensive.

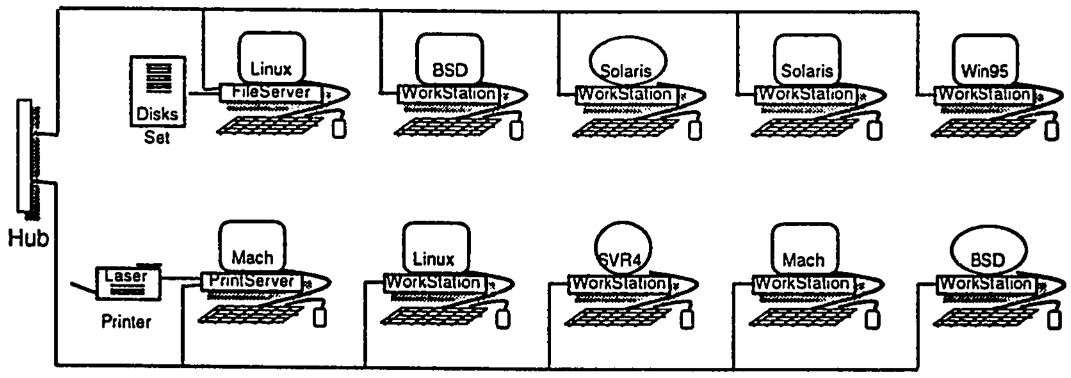

Figure 1.1: A Workstation based Distributed Computing System

Recently, the issue of how to effectively utilize computing power in workstation based distributed systems has sparkled many research ideas and experimental systems. Most of the work concentrates on topics of analyzing workstation usage patterns, designing algorithms for remote capacity allocation, and developing facilities for remote execution. One important issue of sharing resources in such workstation-based distributed environments is the reliability or fault-tolerant aspect of user programs. Since each workstation is considered a personal resource of its owner, the general policy regarding the control of these machines is to keep as much autonomy as possible in the hands of their own users. That is, although workstations can be shared by remote jobs from other users, these remote jobs will be pre-empted from the workstation whenever the local user needs to use the machine. As a result, jobs running on a remote workstation are not guaranteed to run until completion even without considering the possible failure of workstations or network connections. In order to maintain reliable execution of remote jobs, various mechanisms based on worm programs and checkpointing have been adopted for many implementations [10].

1.2 Motivations

The image that a computing system provides to its users and the way they think about the system, is largely determined by the operating system, not the underlying hardware. It is the OS that manages the hardware resources comprising the computing system and provides the base upon which user applications can be executed. Thus, the basic functionality of the OS is to present the user a virtual machine that is easier to program than the underlying hardware. The interface between the OS and the user programs is defined by a set of "extended instructions" that the OS provides. These extended instructions are known as "system calls" and are used to access the services provided by the OS. The structure of the Un$\star$x kernel is shown in Figure 1.2.

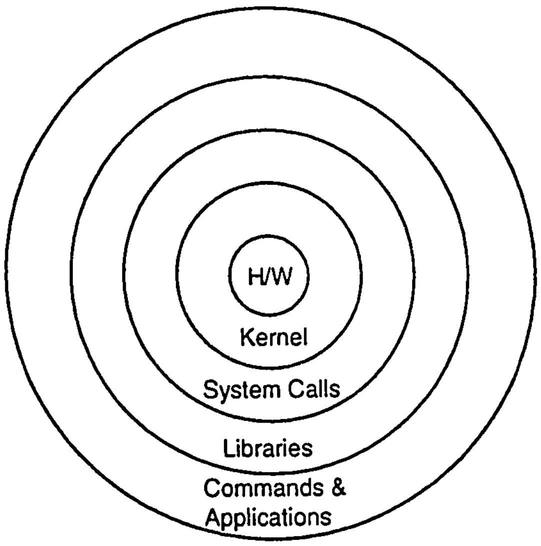

Figure 1.2: Structure of Unix

A fundamental concept in Unix is that of a process, which can be defined as a program in execution. The computing model defined by Unix is based on the process being the basic unit of computation. This model is called the *process model.* All the operations that are supported by the OS are at the process level. In other words, a process is an atomic entity in Unix.

This model has been very effective and efficient for contemporary applications. However, there is a need for a model that can support a finer granularity - at the level of computational state, rather than at the level of processes. In particular, this is required for long-running applications, for example, a cryptographic application that needs to decipher a key. Such applications need huge computational power and run for days and months. As the computation time increases, the probability of sustaining the computation decreases, due to the possibility of failure of the machine/hardware for any of the scores of reasons. Other area of such long-running applications include mathematical simulations, whether forecasting, oceanographic modeling, computational aerodynamics, reservoir modeling, artificial intelligence etc.

What is needed in such applications is a way of ensuring that the programs run to their completion. Some sort of guarantee of successful completion despite the possibility of failures. None of the contemporary OS provide such a guarantee.

In order to provide such guarantee, we need to change the fundamental *process-based* model and extend it into one that is oriented at the *computation state.* We need to have a mechanism that permits users to save a snapshot of the computational state of their processes, at regular intervals of time such that in the event of failure, they can be restored to their latest state, rather than being restarted from scratch. What we need is a mechanism for checkpointing and restoring the computational state of the process. No such support is provided by Un⋆x or any of its variants. This work intends to extend the existing process model of Un⋆x so as to give users the power to checkpoint/migrate their processes.

Un⋆x based distributed systems do provide some simple remote execution facilities for invoking operations on other machines, in the form of *rsh* and *rlogin* commands. In order to understand their limitations, consider the *rsh* command, which provides an extremely simple form of remote invocation under Unix. *rsh* takes as arguments the name of a machine and a command, and causes the given command to be executed on the given remote machine. *rsh* has the advantages of being simple and readily available, but it lacks two important features: *transparency* and *eviction*.

First, a process created by *rsh* does not run in the same environment as the parent process: the current directory may be different, environment variables are not transmitted to the remote process, and in many systems the remote process will not have access to the same files and devices as the parent process. In addition, the user has no direct access to remote processes created by *rsh*; the processes do not appear in listings of the user's processes and they cannot be manipulated unless the user logs in to the remote machine.

The second problem with *rsh* is that it does not permit eviction. A process started by *rsh* cannot be moved once it has begun execution. However, such a preemption facility is essential in order to provide fault-tolerance, real-time scheduling, load-balancing, load-sharing, or idle-workstation utilization.

Process Migration , therefore, provides additional flexibility that a system with only remote invocation lacks.

Such a facility, however, is not available in the context of conventional Un*x systems. The main reason has been the difficulty in isolating a process's state from one machine and reinstating it exactly on another. However, in the case of distributed operating systems that have been built from scratch, their design has been organized so as to minimize the amount of process's state information maintained within the kernel. Further, these systems are based on the *message passing* paradigm which provides the two key aspects of MIMD programming : Synchronization of processes and, read/write access for each processor to the memory of all other processors; which makes the task of incorporating a process migration facility relatively easy. Examples of such distributed operating systems are the Sprite [6], Charlotte [1], Demos/MP [16], and Accent among others.

The emergence of a large number of workstation computing environments based on Unix leads to the issue of how to effectively utilize the computing power available in such distributed systems. Resource utilization in such systems is very low, and as per the statistics of various studies [24], at least one-third of the total computing power is unused even at the busiest times of the day. This low utilization of the workkstation makes such systems ideal candidates for the provision of a process migration facility which is the basis for dynamic scheduling and load-balancing. Process migration would enable effective utilization of the vast computing power latent in such workstation based distributed computing environments.

The provision of a process migration facility raises another important issue - reliability or fault-tolerance aspects of a parallel application executing on the distributed system in the event of failure of the remote execution site. In a typical DCS, it is possible that some processors fail and some others are idle. In order to provide fault-tolerance and to maximize CPU utilization it is desirable to reallocate the tasks/processes executing on the faulty/failing processor to any other working processor on the DCS. The increasing popularity of such systems makes it extremely important to improve the systems reliability.

To visualize the importance of providing a fail-safe computing environment, a narrow scope study was conducted on our departmental DCS. Of the 200 odd workstations comprising the DCS, a set of about 35 machines were selected at random and the number of failures of the machines over a period of 30 days were recorded by a daemon process. The statics obtained are shown as a graph in Figure 1.3. A failure rate of an average of 4 machines per day was observed, and as the number of machines is increased, the rate is expected to grow linearly. Each failure may mean failure of a number of long-term or critical tasks. A Migration Subsystem can restart these tasks on active workstations.

While the crux of the current research has gone into the techniques for effective load-balancing, dynamic scheduling and resource utilization; very little has been done in the direction of fault-tolerant distributed computing within the context of process migration. The traditional hardware redundancy systems for improving fault-tolerance are not cost effective. Our work is motivated by the need for software fault-tolerance to improve the reliability/availability of the system without any extra cost ([15]).

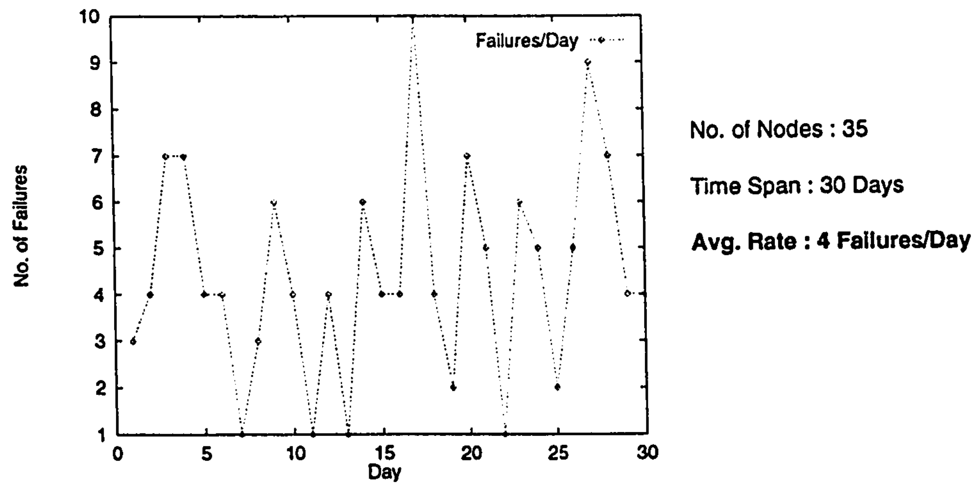

Figure 1.3: Rate of Failure of nodes on a DCS

The benefits and applications of a process migration facility in a DCS can be summarized as follows :

- Cope with dynamic fluctuations in loads and service needs.
- Bring a process to a special device.
- Achieve better overall throughput for concurrent/parallel applications.
- Improve system performance by reducing inter-machine communication costs/time.
- Enable effective utilization of resources (Idle machines).
- Provide fault-tolerance.

Although some distributed operating systems mentioned before do provide excellent example of the benefits of migrating processes during their execution, they are mostly restricted to research environments and stand nowhere as compared to the widely established user-base of standard Un⋆x systems. *The objective of the process migration subsystem is to allow user programs to be migratable among the machines comprising the DCS.* Such a process execution facility is especially suitable for non-interactive, compute-intensive, long-running applications such as scientific computations, simulations, image processing, neural-network or genetic algorithm based applications, rather than interactive and "fast-turnaround" applications that exist for relatively short periods of time.

Nowadays, *Network File System* (NFS) has become an integral part of distributed computing systems. NFS provides on-line shared file access that is transparent and integrated. The user can execute an arbitrary application program and use arbitrary files for input or output without having to worry whether the files are located on the local machine or have to be brought in from a remote machine. In fact, users need not and do not know where their files and directories are physically located. In the near future, a similar facility, say, *Network Computing System* (NCS) would allow transparent execution of user processes on any of the machines comprising the DCS. Users would submit their programs and applications at the local machine, but the actual execution site of their processes may be some other remote machine which is idle or better suited for that application. The task of providing application/process-level fault-tolerance, dynamic scheduling, dynamic load-balancing and proper resource utilization would be handled transparently by the NCS[3]. The provision of these features would make it more powerful distributed system. This research is a step towards building such a system.

[3]An ordinary distributed system provides these features only at the system-level

Chapter 2

Background

The image that a computer system presents to its users, and how they think about the system, is largely determined by the operating system, not the underlying hardware. Modern computer systems often have multiple CPUs. These can be organized as multiprocessors (with shared memory) or as multicomputers (without shared memory). The former tend to be tightly coupled, while the latter tend to be loosely coupled. The basic characteristics of a distributed system is that although it comprises of multiple autonomous CPUs, the image it presents to the users is that of an integrated single computer. Therefore, although shared-memory multiprocessors also offer a single system image, they do so by centralized control, so there really is only a single system, and hence cannot be considered as true distributed systems.

The operating system for loosely coupled, distributed systems can be classified roughly into two classes - *Network operating systems* and *Distributed operating systems. Network operating systems* allow users at independent workstations to communicate via a shared file system but otherwise leave each user as the master of his own workstation. A typical example is a network of workstations connected by a LAN. Each workstation has its own operating system. All the user's commands are normally run locally, right on the workstation. However, it is sometimes possible for a user to log into another workstation remotely by using a command such as *"rlogin machine-name"*, which allows the user's own workstation to behave as a remote terminal attached to the remote machine. Commands typed on the keyboard are sent to the remote machine, and output from the remote machine is displayed on the screen. To switch to a different remote machine, it is necessary first to logout, then to use the "rlogin" command to connect to another machine. At any instant, only one machine can be used, and the selection of the machine is entirely manual.

Communication and information sharing is provided by a shared, global file system accessible from all the workstations. The file system is supported by one or more machines called file servers, which accept requests from user programs running on the other (nonserver) machines, called clients, to read and write files. Workstations can import or mount these file systems, augmenting their local file systems with those located on the servers. While it does not matter where a client mounts a server in its directory hierarchy, it is important to notice that different clients can have a different view of the file system. The name of a file depends on where it is being accessed from, and how that machine has set up its file system. Because each workstation operates relatively independently of the others, there is no guarantee that they all present the same directory hierarchy to their programs. It is possible that the machines all run the same operating system, but this is not required. If the clients and servers run on different systems, as a bare minimum they must agree on the format and meaning of all the messages that they may potentially exchange. In a situation like this, where each machine has a high degree of autonomy and there are few system-wide requirements, people usually speak of a network operating system.

Network operating systems are, therefore, loosely-coupled software on loosely-coupled hardware. Other than the shared file system, it is quite apparent to the users that such a system consists of numerous computers. Each can run its own operating system and do whatever its owner wants. There is essentially no coordination at all, except for the rule that client-server traffic must obey the system's protocols. *Distributed operating systems*, on the other hand, present the entire collection of hardware and software into a single integrated system, much like a traditional time-sharing system. In other words, a distributed system is one that runs on a collection of networked machines but acts like a virtual uniprocessor. Figure 2.1 brings out the major differences between these two systems.

Characteristic	Network OS	Distributed OS
Looks like a virtual uniprocessor ?	No	Yes
All workstations run the same OS ?	No	Yes
Communication primitives	Shared files	Messages
Network protocols required ?	Yes	Yes
File sharing semantics	Not well-defined	well-defined

Figure 2.1: Comparison of Network and Distributed OS

The essential idea is that the users should not have to be aware of the existence of multiple CPUs in the system. Thus, a true distributed system has certain fundamental characteristics. There must be a single, global interprocess communication mechanism so that any process can talk to any other process. Process management must also be the same everywhere. How processes are created, destroyed, started, and stopped must not vary from machine to machine. There must be a single set of system calls available on all machines, and these calls must be designed so that they make sense in a distributed environment. The file system must look the same everywhere, too. As a logical consequence of having the same system call interface everywhere, it is imperative that identical kernels run on all the CPUs in the system. Doing so makes it easier to coordinate activities that must be global. For example, when a process has to be started up, all the kernels have to cooperate in finding the best place to execute it. Nevertheless, each kernel can have considerable control over its own local resources, and handles process/memory management.

No current system fulfills this requirement entirely, but a number of candidates are on the horizon. This study is a step towards enhancing the functionality of the conventional, centralized Uni⋆x OS so as to transform it into a true distributed operating system.

2.1 Literature Survey

A basic feature of a true distributed system is its ability to migrate executing processes among the machines, which would enable transparent load-balancing, dynamic scheduling, and fault-tolerance. Research on migration in the context of distributed systems has been very scarce. Most of the work on providing a remote execution facility has been in custom-made distributed operating systems such as V-System

[13], Butler [14], Remote Unix [11], Condor [12] and Amoeba [22]. Further, there is little work on using checkpointing and process migration as a means of providing fault-tolerance in standard Uni*x based systems [26]. This section discusses some of the attempts that were made in this direction.

2.1.1 Process Migration

The Distributed Automated Workload Balancing System (DAWGS) developed by Clark and McMillin [3], allows transparent remote execution of users's jobs in a work-station based environment. DAWGS uses a distributed scheduler based on a bidding scheme to detect idle machines and is capable of checkpointing and migration. The authors were able to provide such a facility by incorporating special sections of code in the kernel and modifying most of the system calls. Their environment comprised of homogeneous machines running the IBM's Academic operating system, a 4.3 BSD derivative. The major limitations of DAWGS is that the migration mechanism is scattered all over the kernel, thereby making the facility non-portable. Another drawback is that it doesn't support sockets, hence it cannot support migration of distributed applications comprising of multiple processes (tasks) that interact with each other over the network.

In [16], Powell and Miller present DEMOS/MP - a message-based operating system for multi-processor systems. It is based on the micro-kernel paradigm and provides a complete encapsulation of a process. There is no uncontrolled sharing of memory and contact with the operating system, I/O, and other processes is made through a process's links (buffered, one-way message channels). There is no process state hidden in the various functional modules of the operating system. On the other hand, the system servers each maintains its own state, thus no resource state (except for links) is in the process state. Once a process is taken out of execution, it is a simple matter to copy its state to another processor. Further, the processors are all identical and provide the same service thereby forming a homogeneous computing environment. However, this OS has not really left the research labs.

In [22], Tanenbaum describes the details of Amoeba - A micro-kernel based operating system. An important feature of Amoeba is that it has no concept of a "home machine". When a user logs in, it is to the system as a whole, not to a specific machine. The login shell runs on some arbitrary machine and the system automatically looks around for the most lightly loaded machine to run each user command. The system comprises of process servers, file servers, directory servers, compute servers and run servers which together provide the required transparency. Although a very powerful system, Amoeba is still a research tool. To support the huge unix base of applications software, a Unix emulation package was added later. The new design features (like contiguous allocation of virtual and physical memory, no paging or segmentation) may make it a relatively easy candidate for process migration support.

Doulas and Ramkumar [7] observe that most of the prior work on task migration has been of limited use because of the high cost of migration.The authors believe that the key reason for this high cost is that processes are typically large and heavy. Accordingly, they suggest that task migration can be efficiently implemented in an environment that supports a) small light-weight processes (threads) which are cheaper to move and, b) message driven execution. They use the Charm parallel programming environment to decompose applications into many lightweight tasks which are substantially smaller than a typical UNIX process.

Jeffery et al [9] presents four algorithms for checkpointing and restoring parallel programs running on shared-memory multiprocessors. Their criteria for a good checkpointing algorithm were: *efficiency*. *concurrency* and, *low-latency overhead.* However, in these algorithms, the authors checkpoint only the user's address space. The states of the kernel and the file system were not saved. Hence programs that rely on kernel and external states such as RPC and open files were not recoverable.

As it is clear from the above discussion none of these OS compare to the user-domain of Un⋆x systems.

2.1.2 Fault-Tolerance

The article by Yang and Qu [26] deals with the fault-tolerance aspects of remotely executing jobs in a workstation based distributed system. The focus of this paper is on the analysis of reliability and turnaround time for the execution of remote programs. The authors discuss the two common control policies for fault-tolerance : Optimistic (Non-Checkpointing) strategy and, Pessimistic (Checkpointing) strategy. The optimistic policy assumes that jobs running on remote workstations are most likely to complete successfully. Therefore, each remote job is scheduled to run only on one of the idling workstations until either it succeeds on that machine or it loses the machine (because of hardware failure or reclaim of control by the owner) and is restarted at another workstation. The pessimistic policy always prepares for the failure by periodically saving the states of a program during its execution, so that the program can be resumed from its most recent checkpoint.

The analysis of reliability and mean turnaround time of remote jobs running under these two policies is mainly analytical. The authors justify that the theoretical results derived from the analysis can serve as a basis for the choice of different parameters and policies in the development of fault tolerant systems for such distributed environments. The work is based on the following assumptions :

- The operating systems on each workstation provides transparent remote execution and scheduling facilities.
- The system software on individual workstations can co-operatively collect load information and schedule remote jobs to run on idling machines.

In order to support fault tolerant execution of jobs on remote workstations , the authors have adopted a control policy based on the notion of a *backup group* (Figure 2.2). A backup group is a collection of workstations which maintains the information of a running remote job, and is responsible for making sure that the remote job runs on one of the member machines. Should the running copy of the remote job crash due to any unforeseen reason, other member machines in the group will decide to choose another machine to continue the execution of the remote job.

A backup group is called an *n-backup group* if it consists of n workstations. An n-backup group intends to maintain, at all times, n workstations in the group. According to the authors, the notion of backup group provides information clustering

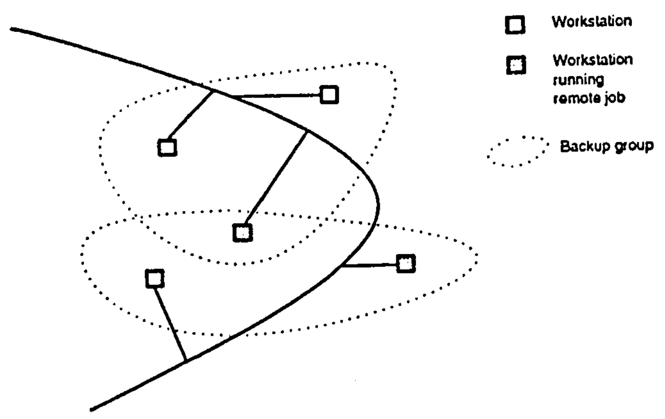

Figure 2.2: Backup group for the execution of remote jobs

in a distributed environment such that Only the members in a group need to have information about remote jobs which are under the control of the group. The authors, however, do not provide any details on constructing and maintaining such backup groups.

Srinivasan and Jha [19] presented some heuristics to improve safety and reliability of the distributed system by mapping tasks to processors. This work is based on the idea of allocating tasks with high execution times to more reliable processors and allocating large volumes of data to more reliable links. However, this approach does not provide any form of fault-tolerance and a processor failure would still result in a total system failure.

Tridandapani and Somani [25] proposed three scheduling strategies to improve fault tolerance by utilizing the idle processors. The concept of running a secondary version of each job on an idle processor was explored using preemption strategies to avoid throughput degradation. This approach is efficient if we are willing to spend the time to run a secondary version of each job. Although the concept of check-pointing was not explored, it is very likely to obtain improved fault tolerance and efficiency for jobs that have negligible communication time compared to processing time.

A fault-tolerant resource allocation algorithm in dynamic distributed systems was proposed in [17]. The scheduling of processes with some resource requirements was discussed under process and crash failures. The degree of fault tolerance was measured by the failure locality which is the maximum number of processes whose liveness conditions can not be satisfied because of a process failure. The work we present here is more general and is not restricted to resource allocation processes.

2.2 Objectives

The main objective of this study is to provide pre-emptive process migration capability to the standard Uni⋆x systems, as a step towards transforming it into a distributed system. As is clear from the above survey, no attempts have been made in this direction. To attain this objective, the following approach have been outlined for this research:

1. Determine the migration relevant internals of the Uni⋆x operating system.

2. Develop new system calls to support process migration and provide the user-level C library interface for them.

3. Implement a *Checkpoint* and *Restore* facility at the kernel level so as to be able to checkpoint and restore any process.

4. Develop a facility for transferring the checkpointed state of a process from the host machine to the destination machine of homogeneous architecture and Operating system.

5. Define the interactions of the proposed process migration subsystem with the existing file and process subsystems, as well as the user applications.

6. Provide migration support for independent as well as communicating processes.

7. The subsystem is expected to be stand-alone, implemented without modifying the existing system calls and kernel code.

Chapter 3

The Migration Subsystem

3.1 Statement of the problem

The objective of this study is to design and implement a process migration subsystem for the Uni$\star$x operating system. The work platform is a distributed system comprising of a network of PCs running the Linux operating system (a Unix clone for i386+ machines). The study includes development of *checkpoint* and *restore* mechanisms on which a process migration subsystem is implemented. The goal is to enable migration of independent processes as well as communicating processes belonging to distributed/parallel applications.

3.2 System Features

Ideally, a process migration subsystem is expected to possess the following features:

1. *Transparency*

   Migration should not affect the behavior of either the process or its peers. Its execution environment must appear the same (access to files and devices) and it should produce exactly the same results as obtained by execution on the home (original) machine. To the rest of the world the process should appear as if it had never left its home machine.

2. *No Residual Dependencies*

   A host need not have to maintain any part of the process's state after it has migrated away from it. For example, no message forwarding (in case of communicating processes) should be required.

3. *Consistency*

   Ability to migrate any user process at any instant of its execution, irrespective of whether it is executing in kernel/user mode, sleeping, or swapped out.

4. *Flexibility*

   The subsystem should provide the mechanism to allow a user to migrate independent process, a set of process belonging to any parallel application.

5. *Low-Latency*

   The checkpointing and restoration time should be kept low so that the overhead on the process is minimal.

6. *Reliability and fault-tolerance*

   Migration may fail in case of network or destination machine failure. However, in such cases there should be no loss of data and the effect should be as if the process has not been migrated.

7. *Security*

   Migrating a process to a remote machine should not compromise the security of the process. The owner of the remote machine (.ie. the super-user of that machine) should not be able to tamper around with this process.

3.3 Role of the subsystem in an NCS

The position of a migration subsystem in the futuristic NCS is depicted in Figure 3.1.

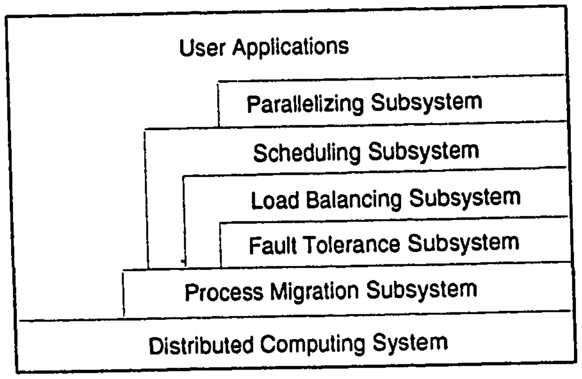

Figure 3.1: The PMS within a futuristic NCS

The user's application would be submitted to the parallelizing subsystem which would decompose the application into a set of parallel tasks. These tasks along with their communication/interaction patterns would be fed as input to the scheduling subsystem which would come up with a static serializable schedule as per the user's requirements. These set of tasks are then input to the load-balancing subsystem which does the job of dynamic scheduling based on the system load and the program's schedule. The fault-tolerance subsystem would then be used to ensure a fail-safe execution of the process. To allocate/deallocate a task to/from any of the hosts of the DCS, the load-balancing subsystem would interact with the process migration subsystem which would assign the task to appropriate hosts as per the load-balancer's requests. The load-balancing subsystem, scheduling subsystem, or user applications can directly interact with the process migration subsystem to avail a limited set of services.

3.4 Issues

The mechanism used to migrate a process depends on the state associated with the processes being migrated. If there existed such a thing as a stateless process, then migrating such a process would be trivial. In reality, processes have large amounts of state (Real-Estate !!!), and both the amount and variety of state seem to be increasing as operating systems evolve. The more the state, the more complex the migration mechanism is likely to be. Let us now consider the design and implementation issues that may be confronted in developing a migration subsystem.

- *The Migration Interface*

  The user-interface to the migration subsystem can be through four major functions:

1. *checkpoint(pid)*
2. *restore(pid)*
3. *migrateIn(pid,fromMachine)*
4. *migrateOut(pid,toMachine)*

In providing the *checkpoint* function, one can visualize three alternatives :

1. Provide these function at the user level so that he can include them in the source code of the program and have regular checkpointing so as to avoid loss of computation. Here the function *checkpoint* would checkpoint only the currently running process and not its descendents, if any[1].
2. A more general approach wherein the function would checkpoint the current process along with all its descendents, if any.
3. A generic approach, *checkpoint(pid)*, wherein the function would take a process's id as parameter and allow checkpointing of any arbitrary process, irrespective of whether it is running, sleeping or swapped out.

- *Migration mechanism and policy*

  In providing a migration facility, it is possible to separate policy from mechanism.

Mechanism $\Longrightarrow$ *What to migrate ?*

How to migrate ?

[1]like *fork()*

Policy $\Longrightarrow$ *When to migrate ?*

Which process to migrate ?

Where to migrate ?

The migration policies might differ, depending on whether the main concern is load sharing (avoiding, idle time on one machine when another has a non-trivial work queue), load balancing (such as keeping the work queues similar in length), or application concurrency (mapping application processes to machines to achieve high parallelism). However, the migration mechanism itself remains the same irrespective of the policy adopted.

- *Location of the mechanism and policy*

  The mechanism as well as the policy can be provided either as

  - a library utility outside the operating system, or
  - an integral part of the kernel, in the form of a system call.

  The policy is mostly associated with resource management[2]. System calls exist which provide the necessary statistics needed for the policy decisions. Hence policy is best provided as a library utility which uses the underlying functionality provided by the migration mechanism [1]. Also, since different situations demand different policies, a policy utility outside the operating system would be easier to test and replace.

  However, the migration mechanism is closely associated with the basic functions of the OS such as process scheduling, memory management, I/O and IPC. Hence it is a good candidate for inclusion in the kernel itself.

[2]Ex: Load Balancing and Scheduling rely on various machine usage statistics

- *Managing the process's state (What to Migrate ?)*

  A Process's state typically includes the following:

  - Virtual memory.

    In terms of bytes, the greatest amount of state associated with a process is likely to be the memory that it accesses. Thus the time to migrate a process is limited by the speed of transferring virtual memory. Handling the virtual memory involves sliding through the process's virtual memory tables, mapping virtual pages to physical page frames, determining which pages are dirty, checking whether they are in-memory or swapped-out and finally saving/transferring them.

  - Open files and Communication channels.

    If the process is manipulating files or devices, there will be state associated with these open channels, both in the virtual memory of the process as well as in the operating system kernel's memory. The state for an open file includes the internal identifier for the file, the current access position, and possibly cached file blocks. The state associated with a message channel (eg: Sockets, Pipes, etc.) includes buffered messages plus information about senders and receivers. Checkpointing involves chasing through the kernel pointers to access the data structures[3] related to these communication channels and saving their state, and setting them up back in the kernel upon restoring.

[3]Local file descriptor table, global file table and in-core inodes

- Execution state.

  This consists of information that the kernel saves and restores during a context switch, such as register values, condition flags and stacks (user/Kernel). This information is machine dependent and is the main bottleneck in providing a migration facility in a true heterogeneous distributed system comprising of machines of varying architectures.

- Other kernel state.

  Operating systems typically store other data associated with a process, such as the process's identifier, a user identifier, the current working directory, signal masks and handlers, resource usage statistics, references to the process's parent and children, and so on. Most of this information is in the *process control block (PCB)* for the process. However, handling some of the information such as pid of the waiting processes, the controlling terminal, pointer to the PDBR[4], the kernel stack and in-core inodes of the executable, among others, need to be mapped onto the new process's environment after migration.

• *Process state consistency*

The process to be migrated should be frozen (stopped) at some instant of its execution so as to ensure a consistent transfer. The issues involved here is *what to freeze and when to freeze ?* Three activities need to be frozen during migration :

[4]Page Directory Base Register : The register pointing to the page directory for that process

1. Process's execution
2. Outgoing communication
3. Incoming communication

The first two activities are trivial to freeze. However, to freeze incoming communication, we can :

- Inform all peers to stop sending
- Delay/Hold all incoming messages and forward them after migration
- Reject/Drop all incoming messages until the process has been migrated

The first option requires a complex protocol to stop and restart communication with the peers, especially handling those that are already on the way. Rejecting incoming messages requires that the sender use some form of protocol that retransmits rejected messages (a UDP socket would not work). Thus we are left with the option of delaying or holding the messages until it is safe to forward/redirect them to the new site of the process.

The overall throughput of the process is affected by the decision of *when* to freeze. The entire process of migration can be thought of as comprising of following stages: Selection, Negotiation, Marshall, Transfer, Demarshall, and Restart. *Selection* is the task of choosing a process for migration, and is usually a migration policy decision, *Negotiation* involves setting up the basic infrastructure for migration, such as checkpointing, *Marshalling* refers to the procedure involved in converting the pointers and data structures of the process's state into a form so that they can be reestablished, and *Demarshalling* is the reciprocal of marshalling.

There are three options for the time of freezing :

- Immediately after selection.
- After negotiations are completed.
- After a major part of the state has been transferred.

In the latter approach, after negotiation all the dirty pages of the process are marked clear and transferred in parallel with the process's execution. Once all the pages are transferred and the other process state has been setup, the process is stopped and any pages that are marked dirty are retransferred. Although these alternatives seem trivial, they do have a significant impact on the process's turnaround time.

- *Transferring the state (How to Migrate ?)*

  The overall problem in migration is to maintain a process's access to its state even after it migrates. For each portion of state, the system must do one of three things during migration:

  1. Transfer the state.
  2. Leave the state on the Home machine and arrange for forwarding.
  3. Ignore the state and sacrifice transparency.

  To transfer a piece of state, it must be extracted from its environment on the source machine, transmitted to the destination machine, and reinstated in the process's new environment on that machine. For state that is private to the process, such as its execution state, state transfer is relatively straightforward. Other state, such as internal kernel state distributed among complex data structures, will be much more difficult to extract and reinstate. An example of 'difficult' state is information about open files and the working set. Finally, some state may be impossible to transfer. Such state is usually associated with physical devices on the source machine. For example, the frame buffer associated with a display must remain on the machine containing the display. If a process with access to the frame buffer migrates, it will not be possible to transfer the frame buffer.

The second option for each piece of state is to arrange for forwarding. Rather than transfer the state to stay with the process, the system may leave the state where it is and forward operations back and forth between the state and the process. For example I/O devices cannot be transferred, but the operating system can arrange for output requests to be passed back from the process to the device, and for input data to be forwarded from the device's machine to the process. In the case of message channels, arranging for forwarding might consist of changing sender and receiver addresses so that messages to and from the channel can find their way from and to the process. This approach results in residual dependencies on the host machines through which the process has been migrated. If such residual dependencies are allowed, we would end up with the process's state being distributed among machines all over the system. This would lead to difficulties when the process migration mechanism is used to provide fault-tolerance as one would need to keep track of all the machines on which parts of the process's state exists. Hence, residual dependencies are best avoided as much as possible.

The third option, sacrificing transparency, is a last resort. If neither state transfer nor forwarding is feasible, then one can ignore the state on the source machine and simply use the corresponding state on the target Machine. An example of such a situation is memory-mapped I/O devices such as frame buffers.

In addition to the above issues, another problem is that of incompatibility of state representation among the different versions of the Unix family, such as 4.3 BSD, System 7 and others. In some versions, a part of the process's state is kept in the kernel, so as to improve efficiency/performance and hence the data structures used are different. This incompatibility forms a major bottleneck in providing a flexible and transparent process migration facility. Architecture incompatibility of the machines comprising the distributed system forms another bottleneck in providing cross-platform migratability. The architectural differences such as the number of registers and their size, the direction of stack growth, support for multitasking makes it difficult (if not impossible) to map the checkpointed state of a process from one kind of machine onto another. A flavor of the difficulty can be obtained by imagining the issues in mapping the Instruction Pointer (IP) of the executing program - the instruction sets of the two machines may be quite different.

Chapter 4

Design of the Process Migration Subsystem (PMS)

The Process Migration Subsystem (PMS) is meant to be a stand-alone subsystem that can be plugged onto an existing operating system, thereby empowering it with the ability to migrate active processes[1]. The implication of this is that the functionality of PMS should not be distributed among the components of the existing OS, but rather should be in the form of an enhancement module that can be attached to it, in order to support migration.

However, the issues involved in developing such a subsystem (see Section 3.4) make it imperative to tailor (certain components of) the PMS so that it can interoperate with the kernel in which it is to be embedded. This section describes the design of the PMS, and outlines the areas where the PMS needs to be tailored according to the OS in which it is to be embedded.

[1]An active process, in this context, implies any process that is presently in the process table (*task_list*) of the OS.

4.1 System Requirements

The basic functionality of the PMS is to provide migration capability to the OS, so that it can handle requests for pre-emptive migration of any process executing on the DCS. A major connotation of this requirement is the ability to checkpoint and restore active processes.

Processes can be basically classified into three types: CPU intensive, I/O intensive, and communication intensive. CPU intensive processes generally belong to *independent* applications - applications that do not interact with any other processes executing on the system. Processes of the latter two types are typically those belonging to distributed or parallel applications - applications comprising of two or more processes that execute on the same or different hosts of the DCS and interact with each other in order to produce the required output. This interaction pattern, which is defined and governed by the protocol followed by the application, can be either through application defined "well-known" files, or through IPC mechanism provided by the OS - such as shared memory, signalling, message passing, or communication channels such as pipes or sockets. However, since the processes of a distributed application are meant to be executed on different hosts, the most common approach is to use sockets as the channels for communication.

For distributed applications that are designed for static assignment of processes to host, the end-point addresses (in the case of socket-based communication) usually get assigned to some variables of the program during the initial stages of execution and are used for communication during the lifetime of the processes execution. However, if such applications are to be run on an environment that supports migration, the end-point addresses would not be valid once any of the component processes migrate over to another host.

Thus, from the users perspective, the PMS should be capable of not only migrating processes, but, in case of distributed applications, should also be able to sustain communication among the component processes transparently, even after migration. Apart from this, it should also be capable of migrating independent/parallel application's processes which are I/O intensive and perform read/write on files. The major criteria here is that the results produced by such applications should be the same, irrespective of whether they have been migrated or not.

4.2 Location of the PMS

The first and foremost decision in the design is regarding the location of the PMS. In order to facilitate portability, one could think of providing PMS entirely at the user-level, as an application program along with a new library archive. This approach can be termed as the *User-level PMS* or UPMS. To make his programs migratable, the user would then need to use the appropriate library functions and have the code recompiled. This would imply the availability of libraries for each programming language which the machine/environment supports. However, the major bottleneck in providing such a subsystem are the implementation difficulties that arise due to the need to access the system's registers (IP, SP, BP, Flags etc.) and the information embedded in the kernel data structures such as the PCB of the process, its open communication channels, scheduling, accounting and user-management information (refer to Section 3.4). Also, the possibility of a process being able to migrate out another process has to be completely ruled out due to the inherent segmentation features of the Un⋆x kernels. Each process in Un⋆x has its own private address-space and the segmentation features of the kernel prevents a process from accessing the address-space of any other process, unless two processes arrange to share a part of their address-space through the use of system calls for shared memory. Although it is theoretically possible to arrange so that the address-space of the process is accessible from other processes, it would be highly unacceptable due to the security breach that it would open up.

The close involvement of the PMS with the internal functioning of the kernel suggests the need for incorporating the subsystem as a part of the kernel itself. This alternative, wherein the PMS is incorporates as a new subsystem within the existing kernel, may be called as the *Kernel-based PMS* or KPMS. Such an approach, apart from being efficient, would allow an arbitrary process to migrate out any other process. Since the kernel of conventional Un⋆x systems is monolithic, it has access to the entire address-space on the machine, upon appropriate setting of the kernel registers. This approach would, however, require major modifications to the existing kernel routines - modifications/enhancements to parts of the *file & I/O subsystem*, *Process Control Subsystem* and even the *Memory Management Subsystem*. This would lead to a situation wherein the functionality of the KPMS is distributed all over the kernel, thereby making it difficult, if not impossible, to port it onto another kernel. The benefit of having the PMS as an integral part of the kernel is that it would facilitate implementation, as the modifications/enhancements to the existing routines would not be major. A significant advantage of such an approach would be that any existing application would become eligible for migration, as there would be no need to modify or recompile the application.

A third alternative would be to have part of the PMS at the user-level and part of it in the kernel. The part of the PMS that is to be plugged into the kernel, say the *Enhanced Kernel* or *E-Kernel*, would provide the interactions needed to extract the process related information embedded within the kernel, while the user-level component of PMS would handle all those aspects that do not require kernel-level intervention. This approach would lead to having the best of both the UPMS and KPMS. The process migration subsystem that is discussed in this design is this hybrid subsystem, wherein some components of the subsystem are embedded in the kernel, while the other parts operate entirely at the user-level. The PMS referred henceforth is precisely this subsystem. The layout of the kernel along with the PMS is shown as a block diagram in Figure 4.1.

4.3 Components of the PMS

The PMS comprises of three components: a set of distributed Migration Daemons (MDs), the PMS user-library, and the E-Kernel. The layout of these components is depicted in Figure 4.1.

To provide process migration capabilities, the PMS executes an MD on every host of the DCS that is willing to take part in migration. In order to be migratable, the user applications register themselves with the MD running on their host, through the use of PMS library routines, and interact with it to avail of its services.

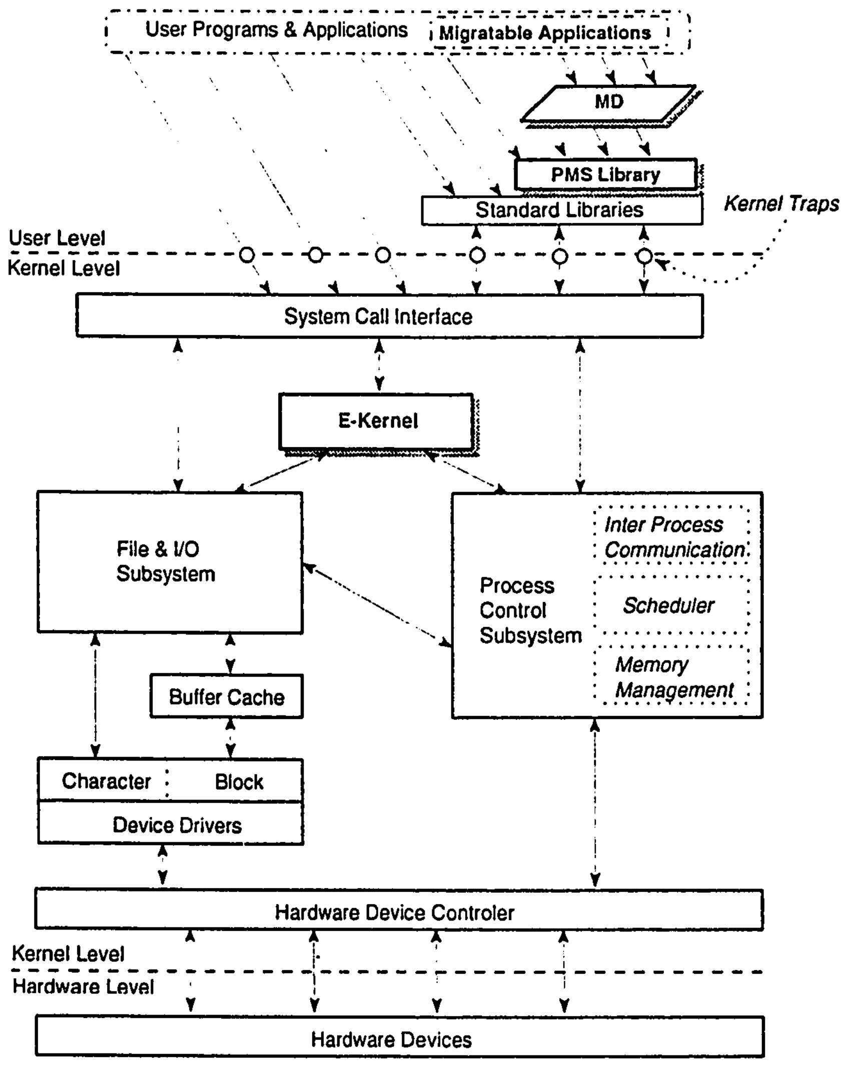

Figure 4.1: Block diagram of the kernel enhanced with the PMS

The MD, in turn, makes use of the PMS library routines to invoke the appropriate kernel routines for migration. These routines are provided by that part of the PMS which is embedded in the kernel, the E-Kernel, in the form of code that enhances the kernel's functionality, thereby empowering it with preemptive process migration capabilities. The MD is, in fact, a component of the distributed program that manages the migration of user applications (which might consist of one or more processes/tasks) executing on the hosts of the DCS. The PMS library forms the interface to the E-Kernel. The MD makes extensive use of the functions provided by the PMS library. User applications, however, may bypass the MD and access some PMS library functions to avail a limited set of services directly from the library.

Figure 4.2 further elaborates this layout from a system-level perspective. In the figure, P_1, P_2 and P_n are migratable processes, in that they have registered themselves with the local MD and hence can be migrated, either of their own volition or by some other process, whereas process P_3 is non-migratable and the MD doesn't even know of its existence. The MDs communicate with each other in order to exchange information regarding the location of processes, so that communication among the component processes of a distributed application can be sustained in the event of migration of any of them.

The implications of adopting this hybrid approach for the PMS and the required interactions among its components is discussed in the following section.

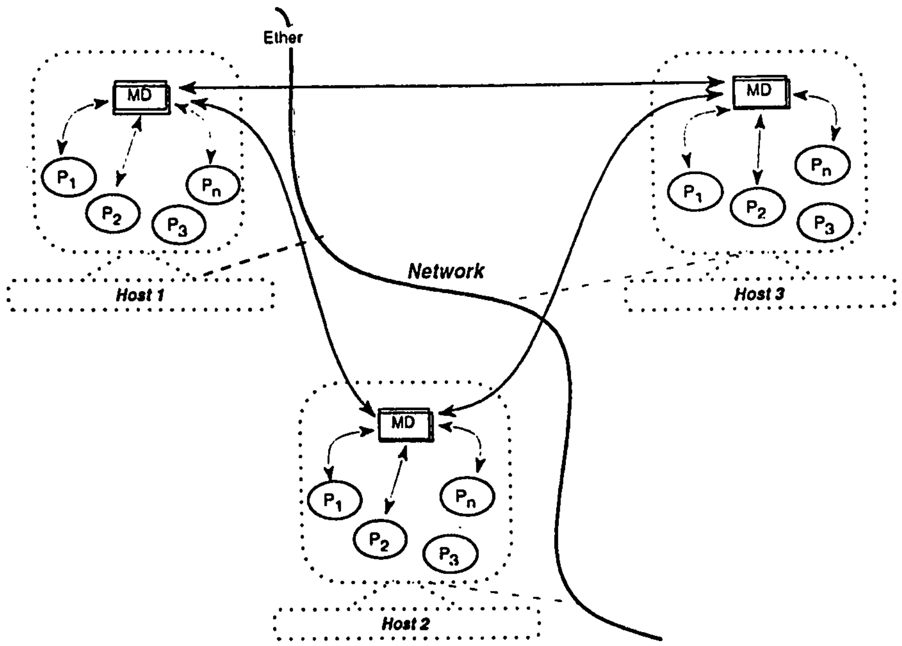

Figure 4.2: System view of the PMS

4.4 The Kernel-PMS Interface

Files and *Processes* are the two central concepts in a Unix system. The block diagram of Figure 4.1 shows these two as logical subsystems, along with the proposed *process migration subsystem*, and the interactions among them.

The figure demarcates three levels : User, Kernel, and Hardware. The system call and library interface represent the border between user programs and the kernel. System calls appear as ordinary function calls in C programs, and libraries map these function calls to the primitives needed to enter the operating system, through well-defined interface points called *Kernel Traps*. When a user's program invokes a system call, execution flow is as follows:

- Each call is vectored through a stub in *libc*, or the corresponding language library archive. Each call within *libc* is generally a *syscallX()* macro, where X is the number of parameters used by the actual routine that provides the functionality for the system call. For example, all the *exec* family of system calls finally boil down to the *execv* system call.

- Each *syscall* macro expands to an assembly routine which sets up the kernel stack frame and calls *_system_call()* through an interrupt. This function is the entry point for all the system calls, and is responsible for saving all registers, checking to make sure that a valid system call was invoked, placing the arguments to the system call onto the relevant registers and ultimately transferring control to the actual system call code via the offsets in the *_sys_call_table*.

- After the system call has executed, *_ret_from_sys_call()* is called. It places the return value on the appropriate register and checks to see if the scheduler should be run, and if so, invokes it.

- Upon return from the system call, the *syscallX()* macro checks for a negative return value, and if there is one, puts its absolute value in the global variable *_errno*, so that it can be accessed by library functions like *perror()*.

Assembly language programs, however, may directly invoke the above underscored functions and bypass the system calls library.

Summary of existing kernel subsystems

File & I/O Subsystem : The file subsystem manages files - allocating file space, administering free blocks, controlling access to files, and retrieving data for user applications. Processes interact with the file subsystem via a specific set of system calls, such as *open*, *close*, *read*, *write*, *stat*, *chown*, *chmod* etc. To access the data, the file subsystem uses a buffering mechanism that regulates/controls data flow between the kernel and secondary storage devices. The buffering mechanism interacts with block I/O device drivers to initiate data transfer to and from the kernel. Device drivers are the kernel modules that control the operation of peripheral devices. The file subsystem also interacts directly with "raw" I/O device drivers without the intervention of a buffering mechanism. Raw devices, sometimes-called character devices, include all devices that are not block devices, such as ttys, printers, Network cards, etc.

Process Control Subsystem : The process control subsystem is responsible for process synchronization, inter process communication, memory management, and process scheduling. The file subsystem and the process control subsystem interact, for example, when loading a file into memory for execution : the process subsystem has to read executable files into memory before executing them.

The memory management module controls the allocation and deallocation of memory and maps the virtual address space used by the applications onto physical memory pages. Two policies are widely used for managing memory: swapping and demand paging. In a swapping system, all memory pages belonging to a 'victim' process are swapped out to disk, whereas in a paging system the pages that are swapped out need not belong to any one particular process.

The scheduler module allocates the CPU to processes. It schedules them to run in turn until they voluntarily relinquish the CPU while awaiting a resource or until the kernel preempts them when their recent run time exceeds a time quantum. The scheduler then chooses the highest priority eligible process to run; the original process will run again when it becomes the highest priority process available.

The inter-process communication module handles communication, ranging from asynchronous signaling of events to synchronous transmission of messages between processes. Finally, the hardware control is responsible for handling interrupts and for communicating with the low-level hardware devices.

The Process Migration Subsystem

The proposed migration subsystem would provide the mechanism which would enable a process to be migrated from the kernel on the source machine to that on a target machine. In order to isolate an executing process, the migration subsystem needs to interact closely with both the file and the process control subsystems, so as to be able to extract the process's state embedded within them. This is precisely the job done by the E-Kernel component of the proposed PMS. One of the main objectives of this research is to define the possible interactions among these subsystems and develop the interfaces between them.

In order to be able to come up with the design of the E-Kernel, we need to know the internal mechanism and data structures of the existing subsystems.

4.4.1 Process Control Subsystem and PMS

A process can be defined as a program in execution, comprising of the program's code along with its run-time data and stack. The lifetime of a process can be divided into a set of states, each with a certain characteristic that describe the process. The various states and the transitions among them are depicted in Figure 4.3.

Consider a typical process as it moves through the state transition model. The events depicted illustrate various state transitions. The process enters the state model in the *created* state when the parent process executes the *fork* system call and, eventually moves into a state where it is ready to run. For simplicity, assume the process enters the state *ready to run in memory*. The scheduler will eventually pick the process to execute, and the process enters the state *kernel running*, where it completes its part of the *fork* system call. When the process completes the system call, it may move to the state *user running*, where it executes in user mode. After a period of time, the system clock may interrupt the processor, and the process enters

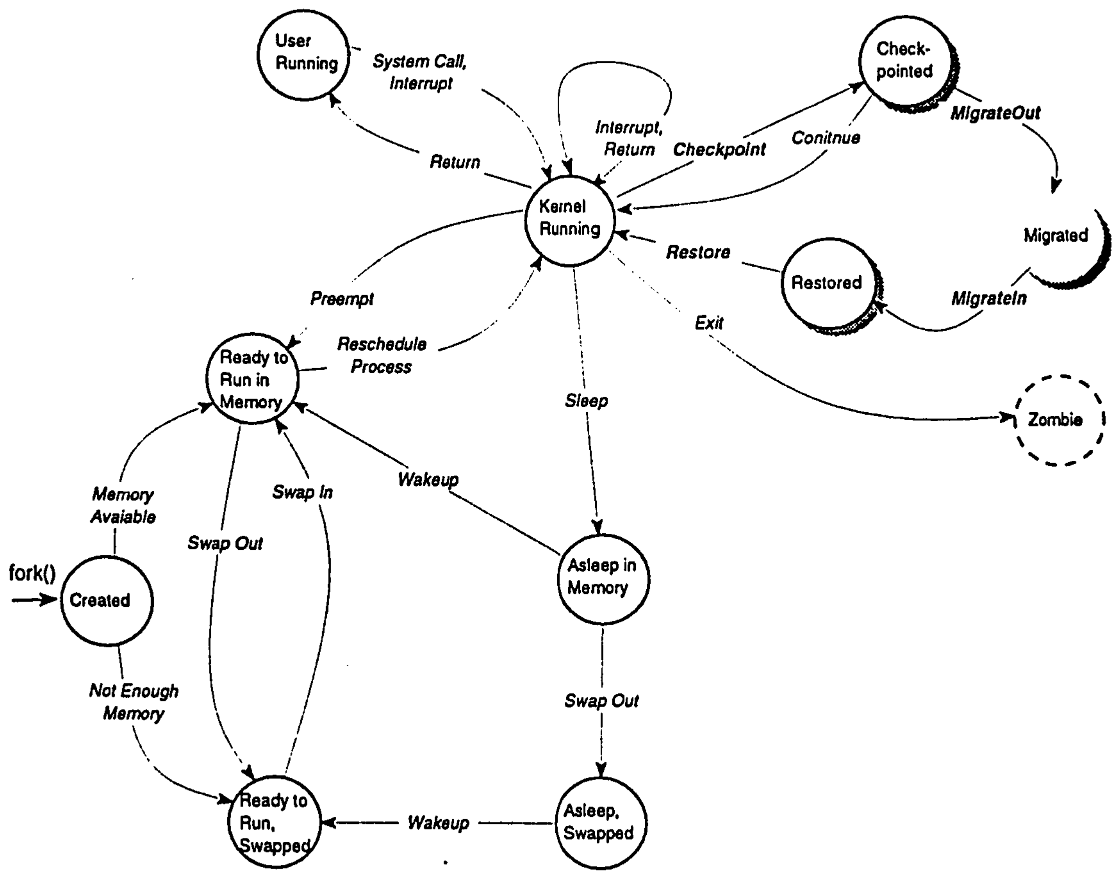

Figure 4.3: Enhanced process state transition diagram

state *kernel running* again. When the clock interrupt handler finishes servicing the clock interrupt, the kernel may decide to schedule another process to execute, so the first process is preempted and goes back to the *ready to run in memory* state. Eventually, the scheduler will choose the process to execute, and it returns to the state *user running*.

When a process executes a system call, it leaves the state *user running* and enters the state *kernel running*. Suppose the system call requires I/O from the disk, and the process must wait for the I/O to complete. It enters the state *asleep in memory*, putting itself to sleep until it is notified that the I/O has completed. Later, when the I/O completes, the hardware interrupts the CPU, and the interrupt handler awakens the process, causing it to enter the state *ready to run in memory*.

Suppose the system is executing many processes that do not fit simultaneously into main memory, and the swapper (process 0) swaps out the process to make room for another process that is in the state *ready to run swapped*. When evicted from main memory, the process enters the state *ready to run swapped*. Eventually, the swapper chooses the process as the most suitable to swap into main memory, and the process reenters the state *ready to run in memory*. The scheduler will eventually choose to run the process, and it enters the state *kernel* running and proceeds. When a process completes, it invokes the exit system call, thus entering the states *kernel running* and, finally, the *zombie* state.

The process has control over some state transitions at the user-level. First, a process can create another process. However, the state transitions the process takes from the *created* state depend on the kernel: the process has no control over those state transitions. Second, a process can make system calls to move from state *user running* to state *kernel running* and enter the kernel of its own volition. However, the process has no control over when it will return from the kernel; events may dictate that it never returns but enters the zombie state (ex: receiving an Abort signal). Finally, a process can exit of its own volition, but as indicated before, external events may dictate that it exits without explicitly invoking the exit system call. All other state transitions follow a rigid model encoded in the kernel, reacting to events in a predictable way according to well-defined rules, such as - No process can preempt another process executing in the kernel, for example; in order to maintain the consistency of the kernel data structures.

In this study, a process is empowered with additional functionality such that it may decide to migrate itself to another kernel, by invoking the new *migrateOut()* system call provided by PMS. This system call would lead it to the *kernel running* state, from which it would be checkpointed, marshalled and enter into the *checkpointed* state, ready to be transferred over to the target kernel. This state is similar to the *zombie* state, except that a process in the *zombie* state would have almost no state information, whereas a process in the *checkpointed* state has its entire state preserved. Upon completion of the *migrateOut()* system call, the process would end up in the *migrated* state, which is a virtual state in which the process is no longer in this kernel. Finally, a foreign kernel might migrate a process over to this local kernel, in which case a (daemon) process on the local kernel would invoke the *migrateIn()* system call and introduce the migrated process into the kernel through the restored state, which is an exact replica of the *checkpointed* state of the process as it was on the kernel from which it had migrated.

When the process is in the *checkpointed* or the *restored* state, it needs to preserve the complete state information of the process at the instant it was checkpointed. Let us now see exactly what comprises the *complete state information* of a process ?

After the process is created by *fork()*, the kernel loads the executable file (binary) of the process into memory during *exec()*. The loaded process consists of at least three parts, called *regions*: *text*, *data*, and the *stack*. The *text* and *data* regions correspond to the 'text' and 'data' sections of the executable file, but the stack region is automatically created and its size is dynamically adjusted by the kernel at run time. The *data* region can be viewed as comprising of two subregions: *initialized data* and *uninitialized data (bss)*. The statically allocated variables comprise the *initialized data* region whereas the variable and data-structures that are allocated dynamically form the *unititialized data* region. These components forming the state of the process are depicted in Figure 4.4

Because a process can execute in two modes: Kernel or User, it uses a separate stack for each mode. The *user stack*, apart from holding the pushed registers and parameters involved in the function calls, is also used to allocate memory for the local variables declared/used within the functions; whereas the *kernel stack* is used when the process is executing kernel code while handling system calls or interrupts, and is null when the process executes in user mode. The function and data entries on the kernel stack refer to functions and data in the kernel, not the user program,

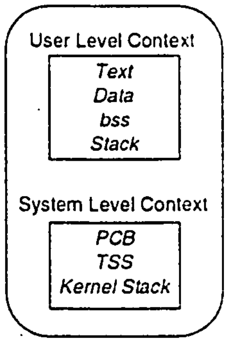

Figure 4.4: Process's state information

but its construction is the same as that of the user stack. An important point to be noted here is that *every process has its own user stack, but there is only one kernel stack that is used by all processes executing on that kernel.* In other words the kernel stack is a global data-structure. In fact, it plays a dual role - Acting as a private stack for the calling process, while handling system calls; and as a public area when processing interrupt handlers. All this information (except for the contents of the kernel stack, when it is being used for handling interrupts) constitute the PRIVATE state information of the process.

The most crucial information maintained by the process control subsystem comprises of the processor state that needs to be saved at each context switch and, the per-process data structure, called the *process control block* (PCB). The information needed for a context switch is basically the contents of the hardware registers, while the PCB comprises of the process management and accounting information. Some of the major fields of the PCB are shown in Figure 4.5. Thus, the *complete state information* of a process comprises of its *private* state, context switch information (*TSS*), PCB (*task_struct*) and its kernel stack (if the process was in kernel mode at that instant), as shown in Figure 4.4. This is precisely the information that needs to be preserved when the process is checkpointed (apart from the information related to I/O, mentioned in Section 4.4.2).

Process Management	Memory Management	File Management
Registers	Pointer to text segment	Umask value
Program Counter	Pointer to data segment	Root directory
Program Status Word	Pointer to bss segment	Working directory
Stack Pointer	Exit status	Executable's descriptor
Process State (R/S/Z..)	Signal status	Open files descriptors
Time Started	Real UID	Various flags
CPU time used	Effective UID	
Children's CPU time	Real GID	
Time of next alarm	Effective GID	
Message queue pointers	Bit maps for signals	
Pending signal bits	Various flag bits	
Process ID		
Parent process ID		
Process group		
Controling tty		
Various flag bits		

Figure 4.5: Major fields of the process control block

Migrating the process's state information

When a process is checkpointed, its *text* region need not be saved, as it can be easily obtained from the executable file (binary) of the program of this process. The contents of the *Initialized data*, *bss* and *stack* regions, however, have to be saved as they vary with the execution of the process and define the state in which the process is at that instant (user/kernel). Saving and reinstating this information is relatively straight forward. The contents of the *PCB* and *TSS*, however, need to be mapped onto the new process's image during restoration. In *TSS*, the registers that need special care are ESP and EBP, which depend upon the state (user/kernel) of the process at that instant. If the process was in a system call, at the instant of checkpointing, these registers would point to the kernel stack, otherwise they point to the user stack. For the *PCB*, most of its components are pointers to other data structures in the kernel and extracting their information implies chasing through the pointers and *marshalling* their contents. This is the reason that marshalling forms the critical part of checkpointing.

4.4.2 The File & I/O subsystem and PMS

The internal representation of a file is given by an *inode (index node)*, which contains a description of the disk layout of the file's data and attributes such as the owner, access permissions, and access times. Every file has one inode, but it may have several names (links), all of which map into one inode. When a process refers to a file by a pathname, the kernel parses the file name one component at a time, checks that the process has permission to search the directories in the path, and eventually retrieves the inode for the file. Inodes are stored in the file system, but the kernel reads them into an *in-core inode table* when manipulating files. The kernel contains two other data structures: the *file table* and the *user file descriptor table.* Figure 4.6 shows the tables and their relationship to each other.

The *file table* is a global kernel structure, but the *user file descriptor table* is allocated per process. When a process opens or creats a file, the kernel allocates an entry from each table, corresponding to the file's inode. Entries in the three structures - *user file descriptor table*, *file table*, and *inode table*; maintain the state of the file and the user's access to it. The *file table* keeps track of the byte offset in the file where the user's next read or write will start, and the access rights allowed to the opening process. The *user file descriptor table* identifies all open files for a process, including I/O devices, which are treated as files in Unix. The kernel returns

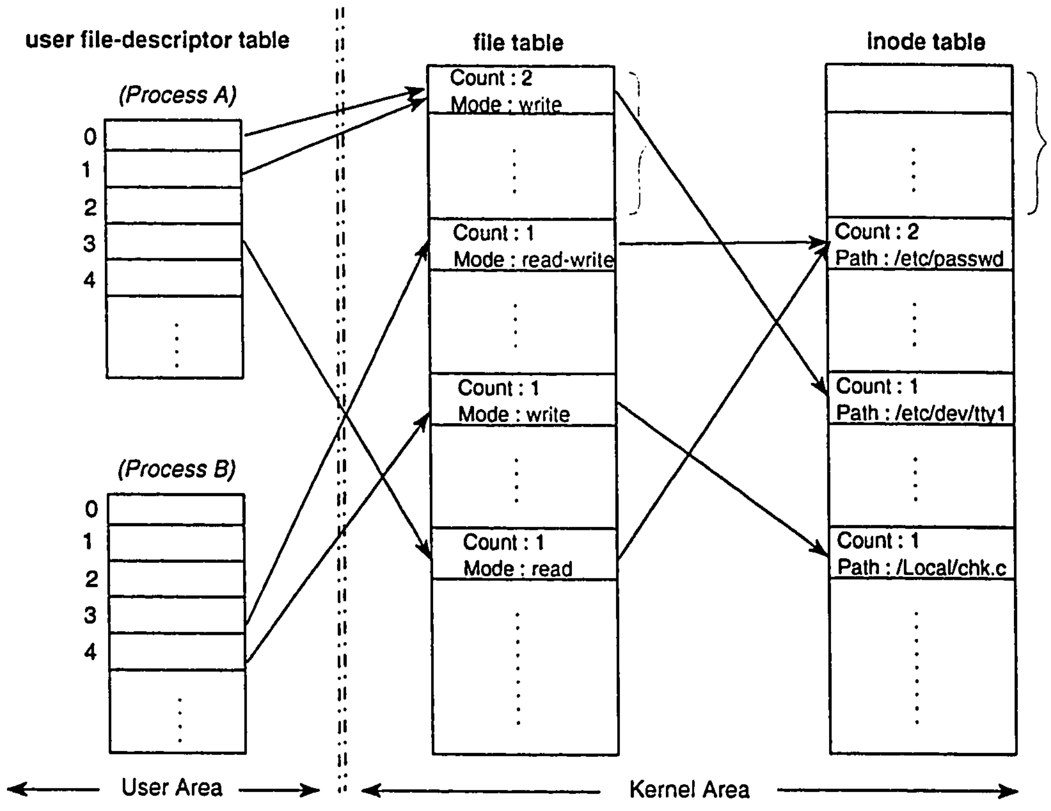

Figure 4.6: File descriptors, File table, and Inode table

a file descriptor for the open() and creat() system calls, which is an index into the *user file descriptor table*. When executing read() and write() system calls, the kernel uses the file descriptor as an index to access the *user file descriptor table* and follow pointers to the *file table* and *inode table* entries.

An installation may have several physical disk units, each containing one or more file systems. The kernel deals on a logical level with file systems. The conversion between logical device (file system) addresses and physical device (disk) addresses is done by the disk driver.

Migrating I/O related process state

As far as the PMS is concerned, to migrate a process, we need to take care of preserving the process's access to the files that it had opened and reestablish these onto the other kernel. As is obvious from Figure 4.6, we not only have to save the information of the *user file-descriptor table*, but also the information contained in the global *file table* and the in-core *inode table* which are kernel data structures. Since the *user file-descriptor table* is within the address space of the process, its contents can be preserved simply by saving the process's *bss* region. However, saving and reinstating this and the other two tables would require very tedious manipulations of the pointers involved, especially in cases where the pointers refer to physical addresses, as is usually the case for kernel address space in most of the Uni⋆x systems.

Another alternative would be to just save the file descriptors along with their corresponding read/write positions and user identification/access information, and ignore the other contents of the tables. During restoration, one would need to use these file descriptors to reopen the relevant files, and then change the contents of the newly allocated tables to the checkpointed values. Although this sounds fairly simple, the major flaw is that the file descriptors do not form a unique identification of the files existing on the filesystem. Instead of the file descriptors, we can extract and save the physical disk inode number from the *in-core inode table* during checkpointing and use this as identification, since inode numbers are guaranteed to be unique within a given filesystem. Since kernels routines/functions exist which take the physical inode number and return back the file descriptor, after filling up the tables appropriately, this seems to be the easy and efficient solution. Migration between different filesystems would complicate the situation. This could be handled by extracting and saving the filesystem ID from the *inode table* while checkpointing.

However, a much better and portable approach would be to keep track of the files that are opened by the process, by providing a proxy to the existing *fopen()* and *fclose()* library routines. These routines would save the full pathname of the file that is being opened in a user-transparent internal data structure within the process. During checkpointing, the read/write pointers for these files would be saved in the data structure and the files flushed. Restoring would then be a matter of re-opening these files by extracting their pathnames from this data structure and repositioning their read/write pointers. Apart from being straightforward, this approach would be highly portable, even across different file systems. This approach has been adopted by the PMS to deal with the files related state information during migration of the process, and the proxy routines are provided by the PMS library.

4.4.3 The Virtual Memory Management Module and PMS

Memory management in Uni⋆x relies on the segmentation and paging support provided by the underlying hardware. *Segmentation* is used to give each program several independent, protected address spaces. *Paging* is used to support an environment where large address spaces are simulated using a small amount of RAM and some secondary storage. When several programs are executing at the same time (Multitasking), either mechanism can be used to protect against interference from other programs. The memory management features apply to units called *segments.* Each segment is an independent, protected address space. Access to segments is controlled by data which describes its size, the privilege level required to access it, the kinds of memory references allowed (read, write, instruction fetch, stack push/pop, etc.) and whether it is present in memory or on disk. Segmentation is used to control memory access, which is useful for catching bugs during program development and preventing a misbehaving program from tampering with the address space of others. A frequent cause of software failures is the growth of the stack into the instruction code or data region of a program,. Segmentation can prevent this. The stack segment would have a maximum size enforced by the program or OS, and any attempt to grow beyond would generate an exception (software interrupt). The segmentation hardware translates a segmented (logical) address into an address for a continuous, unsegmented address space, called a *linear address.* If paging is enabled, the paging software translates the linear address into a physical address. If paging is not enabled, the linear address is used directly as the physical address.

To better understand the intricacies of memory management, let us look into some the *i486*'s memory management hardware. The process of translating a virtual address contained in the CS:IP registers into the physical page frame is depicted in Figure 4.7. Before going into the details of the above figure, here are some basic definitions:

- *Segment Selectors*

  A segment selector points to the information which defines a segment (called a segment descriptor). Any of the segment registers (cs,ds,es,ss,etc.) can act as a segment selector.

- *Segment Descriptors*

  A segment descriptor is a data structure in memory which provides the processor with the size and location of a segment, as well as control and status information.

- *Segment Descriptor Tables*

  A segment Descriptor table is an array of segment descriptors. There are two kinds of descriptor tables:

  1. The global descriptor table (GDT)
  2. The local descriptor tables (LDT)

  There is one GDT for all tasks, and an LDT for each task being run. A descriptor table is variable in length and may contain up to 8192 (2^{13}) descriptors.

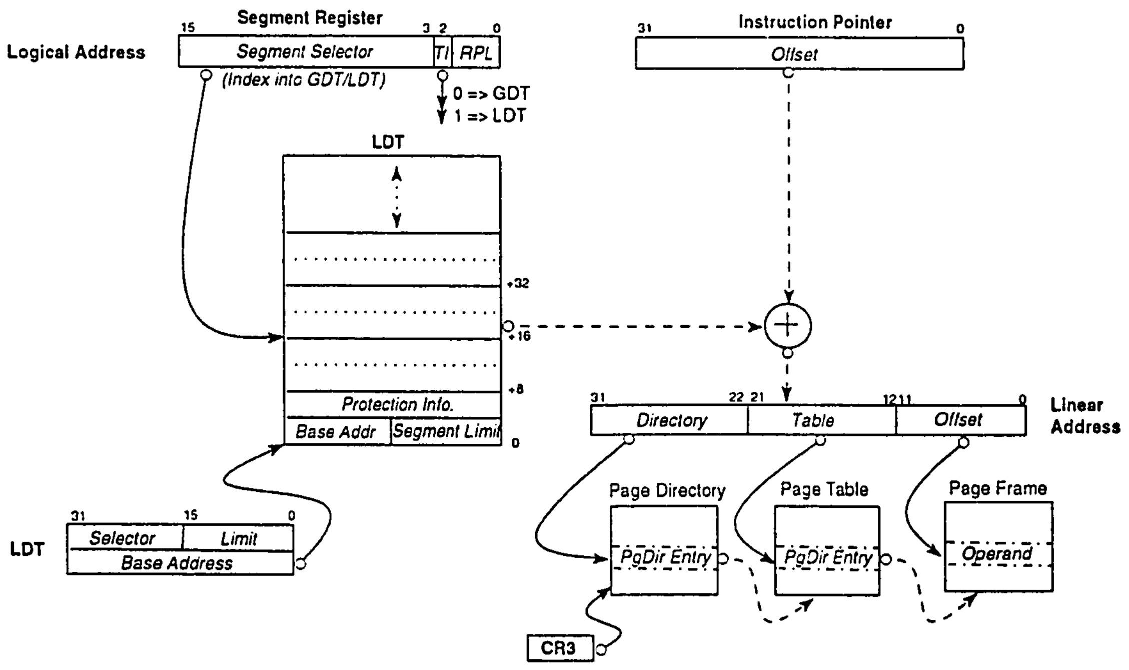

Figure 4.7: Mapping Logical address to Physical address

A virtual (Logical) address consists of the 16-bit segment selector for its segment and a 32-bit offset into the segment. A logical address is translated into a linear address by adding the offset to the base address of the segment. The base address comes from the segment descriptor, which comes from one of two tables, the global descriptor table (GDT) or the local descriptor table (LDT).

Unlike segments which usually are the same size as the data structures they hold, on the *i486*, pages are always 4K bytes. The paging mechanism treats the 32-bit linear address as having three parts, two 10-bit indexes into the page tables and a 12-bit offset into the page. A *page table* is an array of 32-bit entries. A page table is itself a page, and contains 4096 bytes of memory or, at most, 1K 32-bit entries. Two levels of tables are used to address a page of memory. The top level is called the *page directory.* It addresses up to 1K page tables in the second level. A page table in the second level addresses up to 1K pages in physical memory. All the tables addressed by one page directory, therefore, can address 1M or 2^{20} pages. Because each page contains4K or 2^{12} bytes, the tables of one page directory can span the entire linear address space of the *i486* ($2^{20} \times 2^{12} = 2^{32}$). The physical address of the current page directory is stored in the PDBR (the CR3 register of the TSS).

Paging is used by Uni⋆x in order to support demand-paged virtual memory. When a program's instruction attempts to access a page which is on disk, the program is interrupted in a special way - it allows the operating system to read the page from disk, update the mapping of linear addresses to physical addresses for that page, and restart the instruction. This process is transparent to the program.

Migrating VM related process state

The virtual memory of a process forms the greatest amount of state associated with it, usually in the order of 2^{20} pages. However, at any instant of time, only a fraction of these pages are in use by the process. Also, since pages are allocated and deallocated as per demand, during the process's execution, there might be *holes* in the address space of the process. The memory layout of a process is shown in Figure 4.8.

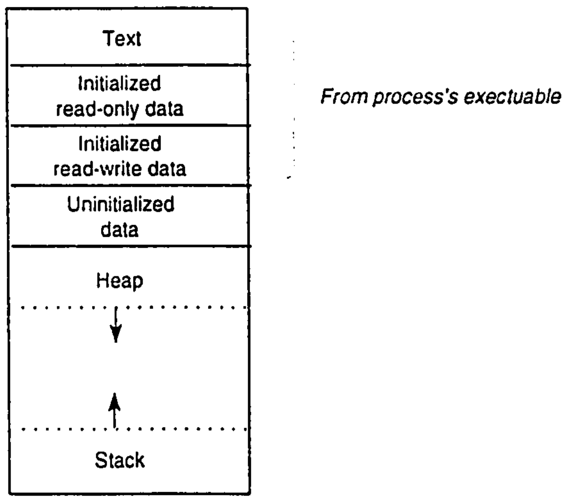
Text
Initialized read-only data
Initialized read-write data
Uninitialized data
Heap
Stack
From process's exectuable

Figure 4.8: Memory layout of a process

To provide efficient migration, PMS has to ensure that only those pages which contain information needed by the process after its migration should be migrated. To migrate a process, the MD would invoke the *sysCheckpoint* system call provided by the E-Kernel, which goes through the process's virtual address space, obtains the physical page frames corresponding to a given virtual address, and saves the contents of only those physical pages that have been accessed and are dirty. It also keeps track of the virtual addresses of the pages, so that during restoration, PMS can maintain the original virtual addresses. To reinstate the process, the MD at the destination host invokes the *sysRestore* system call, which reads the information from the checkpointed image file and interacts with the page protection mechanism of the kernel such that COW (Copy on Write) and READ-ONLY pages can be overwritten. If the virtual page to be restored doesn't exist in the process's page table, *sysRestore* obtains a free page from the kernel, overlays its contents from the image, and restores its protection bits as they were at the instant of checkpointing.

Some of the pages of the process may belong to the shared dynamically linked libraries used by the process. Saving these during checkpointing would be unwarranted, as they will be available at the destination host. However, the virtual address at which these pages have been loaded (in case they are already in use by some other processes executing on that site) may not be the same at the site of restoration. Saving theses addresses would be of no use as mapping them to the address existing at the destination site would be highly inefficient, if not impossible. During checkpointing, since the virtual address of the page, by itself, would provide no clues as to which function of which library this particular page belongs, we would need to lookup this address in the symbol-tables of all the dynamic libraries to obtain the information. Similarly, upon restoration, one would need to perform a reverse lookup in the symbol-tables of these libraries at the present host, to obtain the virtual address at which the required function's pages are presently loaded.

Inorder to overcome this bottleneck, the PMS simply ignores those pages which are shared. The implication is that processes that use shared memory for IPC are non-migratable. However, such processes do not exist in practise, since this form of IPC is never used in developing distributed applications. Further, using shared memory as the IPC medium has become outdated, and applications using this are virtually non-existent on any real DCS. While restoring the process, the PMS simply starts executing the process from scratch, with an argument of "-r", which forces it to go into restoration mode immediately upon execution of the first few instructions. By the time the process starts executing, the kernel would have set up all its dynamically linked library pages, and the PMS need not be bothered about them.

Apart from this, some of the entries of the process's page tables point to pages that contain the kernel routines and system calls that are used by this process. Since these pages will be available at the destination kernel, the PMS can safely ignore them. For pages that have been swapped out, trying to read their contents during checkpointing would result in a page-fault, and the kernel would have it loaded back into memory and restart the read instruction transparently. Hence, as far as the PMS is concerned, there is no difference between a page that is in-memory or swapped out to disk.

4.4.4 IPC and PMS

The Inter Process Communication (IPC) module of the process control subsystem is responsible for handling the communication between processes. The most simplest and primitive form of communication among process is through files, wherein one process writes to a "well-known" file and one or more processes read form it. PMS already provides support for migration of processes with open files. However, the standard IPC mechanisms supported by Uni*x systems are: *message queues*, *shared memory*, *semaphores*, *signals*, *pipes* and *sockets*[2].

Message queues allow processes to send formatted data streams to arbitrary processes, *shared memory* allows processes to share parts of their virtual address space, and *semaphores* allow processes to synchronize execution. A *pipe* is a bi-directional communication channel that can be used to pass data between two processes, while *signals* are used to inform processes of asynchronous events. All these IPC mechanisms are designed for a single-machine environment and the state information regarding these is embedded entirely within the kernel. Hence, from the PMS's perspective, processes using these are non-migratable due to the enormous overhead involved in extracting/reinstating the relevant state information from the kernel data structures. However, *sockets* form a basic communication mechanism used by distributed applications and hence it is very essential to provide support for migration of processes using this form of IPC. The following discussion describes the functionality needed by PMS to handle migration of processes using socket-based IPC.

A *socket* is an application program interface (API) to the network communication protocols. It is an abstract object from which messages can be sent and received. Sockets are created within a communication domain[3] much as files are created within a filesystem. Like files, each active socket is identified by a small integer called its *socket descriptor*, which is an index into the same *user file descriptor table* that is

[2]For details, refer to [2]
[3]such as Unix domain, Internet domain and XNS domain

used by files.

The easiest way to understand the socket abstraction is to envision the data structures in the operating system. When an application calls the *socket()* system call, the kernel allocates a new data structure to hold the information needed for communication, and fills in a new entry in the *user file descriptor table* of that process to contain a pointer to this kernel data structure. Figure 4.9 illustrates these data structures after a call to *socket()*.

Although the internal data structure for a socket contains many fields, the system leaves most of them unfilled when it creates the socket. The application that created the socket must make additional system calls to fill in information in the socket data structure before the socket can be used.

Migrating IPC related process state

To support migration of processes involved in communication using sockets, the PMS needs to obtain the socket related data structures for the process to be checkpointed. However, as depicted in Figure 4.9 and Figure 4.10, almost the entire data structures are embedded within the kernel.

The only socket related state information that is at the user-level is that contained in the *user file descriptor table*, which gets saved while checkpointing the file & I/O related process's state information (as discussed in Section 4.4.2). Arranging for marshalling the information embedded within the kernel would be oppressive, to say the least. A much better approach would be to store the user-level socket information in a PMS library data structure, through the provision of proxys for the existing socket related system calls, such as *socket()*, *bind()* and *sendto()*. Dur-

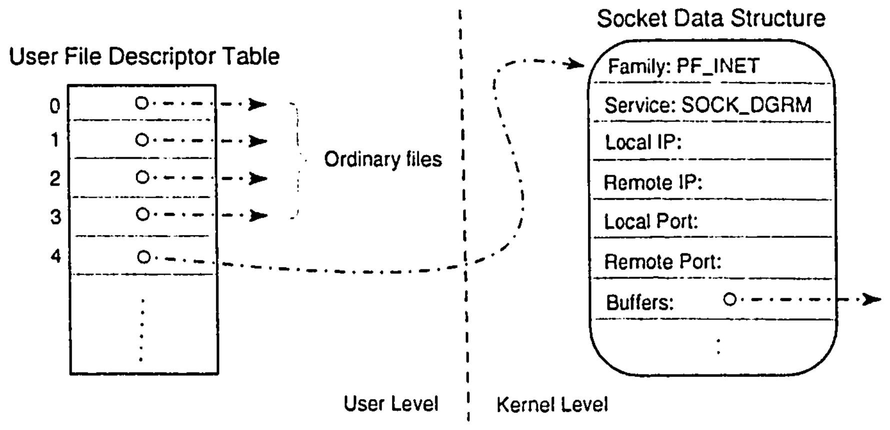

Figure 4.9: The file descriptor table with an open socket

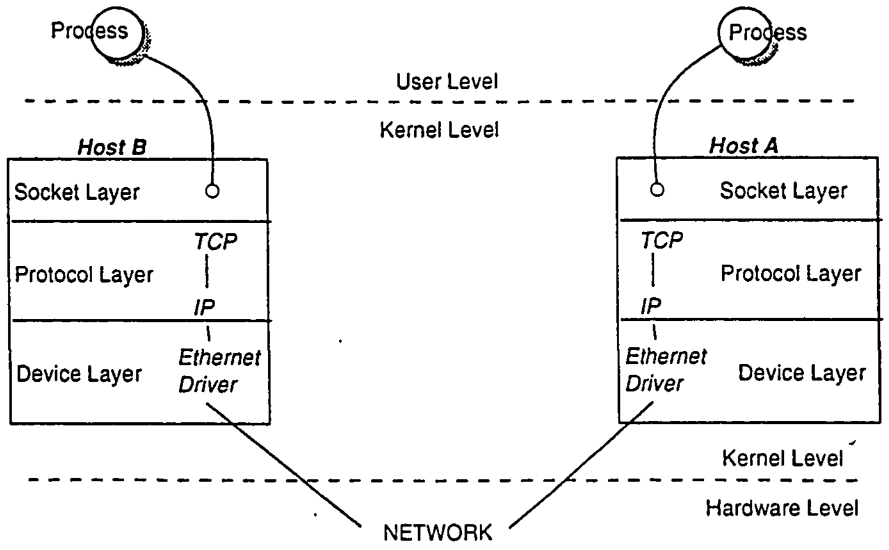

Figure 4.10: The Sockets Model

ing migration, the contents of this data structure would be preserved simply by checkpointing the process's address space through the *checkpoint* PMS routine, and the sockets closed. Upon restoration, the *restore* routine would setup these sockets by looking up the relevant information from this data structure and re-establishing them.

Apart from this, the PMS has to ensure that the communication among the component processes of a distributed application is sustained even after migration of any one of them. The objective is to ensure that the data being sent reaches the process to which it was originally destined. Our study has revealed that at the *protocol layer* level, the kernel routines convert the user-specified end-point address into a protocol specific network address. One approach to deal with the above problem is to enhance these routines such that they maintain an additional data structure containing the original and current locations of the migratable processes executing on the DCS. Prior to performing the address conversion, these routines would lookup the user-specified end-point address from this data structure and replace the original address with the current one. This approach, however, would require tampering the existing kernel routines, as this functionality cannot be provided as a new module of the E-Kernel. Further, it would be specific to the particular network protocol being used - adding a new protocol would imply changing some of its routines to provide this functionality.

The approach adopted by PMS to handle transparent communication, is to maintain this data structure at the user-level and do the translation at the end-point address itself. This data structure is thus maintained by the MD at each host, which interact with each other to keep this data structure updated. The details of the protocol followed in doing this is described in Section 4.5.3.

4.4.5 Design Alternatives

The basic prerequisite for process migration is the need to have the complete process's state information saved prior to migrating the process, and restoring this saved state upon resumption of execution at the target host. This implies provision of the functionality of *checkpointing* and *restoration*. From the design of the PMS discussed in Section 4.2 and depicted in Figure 4.1, the obvious candidate for performing this task is the E-Kernel. Thus, the functionality of checkpoint and restore is provided by the E-Kernel in the form of two new system calls - *sysCheckpoint()* and *sysRestore()*. Further two new PMS library routines have been provided which act as the application's interface to these system calls. This section discusses the other alternatives available, justifies the need for the E-Kernel, and bring out the role of the MD in checkpointing/migrating any arbitrary process.

Checkpointing

In order to checkpoint a process, the straightforward approach would be to make it checkpoint itself. In this situation, the sysCheckpoint system call wouldn't take any arguments and would checkpoint the process invoking it (i.e. the currently executing process) as the existing *fork()* system call. Thus, to checkpoint itself, the process would invoke sysCheckpoint() somewhere within its code. Since the system call would be serviced within its own context, the process would have complete access to its entire address space. Further, since the system call is handled in the kernel mode and since the Unix kernel is monolithic, the sysCheckpoint() kernel routine can access the global kernel data structures by setting up the appropriate segment registers.

However, this simple approach has two subtle flaws. The first one is that we cannot provide the functionality of having any other process checkpoint/migrate this process, due to the segmentation features of the kernel, as discussed in sections 4.2 and 4.4.3. Such a functionality is essential to provide fault-tolerance so that a remote process can migrate-in the processes from the failed node. Another situation is a distributed load-balancing application [18] wherein the load-balancer might need to checkpoint and migrate-out a resource-hungry process from the host to a remote machine on the DCS that is lightly-loaded or idle (see Section 4.12). The second flaw is more inconspicuous and is depicted in Figure 4.11.

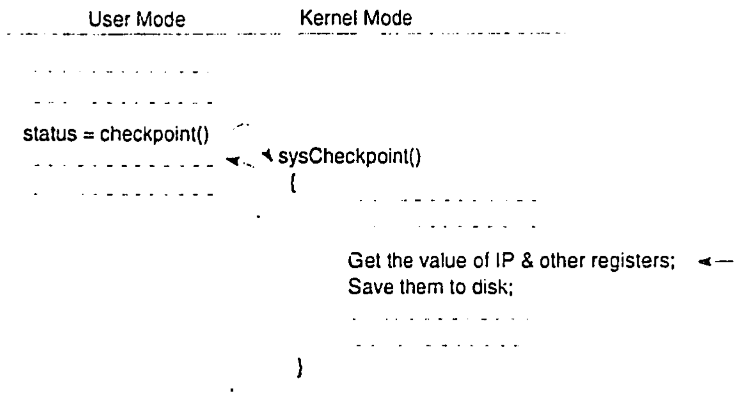

Figure 4.11: A scenario of the usage of sysCheckpoint()

The flaw is that upon restoration, the IP register would be pointing to the first instruction shown in the *sysCheckpoint()* routine of Figure 4.11. Hence, immediately after restoration, the contents of the IP and other registers would again be saved to the disk unnecessarily. This could lead to a deadlock situation if the process is performing checkpointing inorder to migrate over to another host. Immediately upon restoration, the process would again dump the contents of its registers to the disk and try to migrate to the specified host, which is now the same machine.

Another problem is that at the instant the process is checkpointed, it would be in the kernel mode. Our study of the Linux kernel has revealed that, in order to avoid the overhead in translating a virtual address to physical address, the kernel code is mapped directly into the physical memory locations, without going through the page tables. Hence, everything within the kernel refers to physical memory locations, thereby making it impractical to migrate a process that has been executing in kernel mode. This leads us with only two alternatives: make changes to the existing kernel routines, or disallow processes to be migrated while they are executing in the kernel mode. The first option has been ruled out, as it forces us to tamper with the existing routines - routines that are in no way concerned with the PMS, as such tampering would spread out the PMS related processing among the entire kernel. This is in violation of our design ideology, wherein we are interested in providing a stand-alone subsystem that can be easily incorporated into any existing kernel. The second option of not allowing a process to be checkpointed while it is executing in the kernel mode, apart from being too stringent, makes it impossible to have the process checkpoint itself, as checkpointing necessarily forces the process to manipulate its kernel-level data structures, which can be accessed only when the process is running in kernel mode.

Another issue can be raised by considering an example. A load-balancing system is run on the DCS to allow efficient resource utilization, as described in [18]. The load-balancing application spawns an *application controller* on one of the hosts on the DCS and an *application agent* on each machine, as shown in Fig.4.12.

Let process A be the application agent on host H. Among the many processes running on H, let P be a "resource-hungry" process. Process A decides that it is best to move P to some other machine, say X, and invokes the system call *migrate-Out(P,X)*. This results in the sequence of calls leading to *sysCheckpoint(P)*. At some time during checkpointing, the process P has to be stopped. Since the checkpoint system call is invoked in the context of process A, the kernel can easily stop process P by putting it out of the ready queue. After marshalling the relevant data structures, the state information would be transferred over to the kernel on X and the process restarted. However, consider the case wherein the process P wants to checkpoint itself. Since the *sysCheckpoint(P)* system call would now be invoked within the context of P itself, stopping the currently executing process (P) by removing it from the ready queue would bring the system to an instant halt.

Approach Adopted

A strategy that overcomes all these drawbacks is to assign the job of checkpointing to a special daemon process. If a process wants to checkpoint itself, it would contact the daemon with its own *pid* and request it to be checkpointed. This scenario is explained in detail in Section 4.5.2. Since it is another process that is invoking

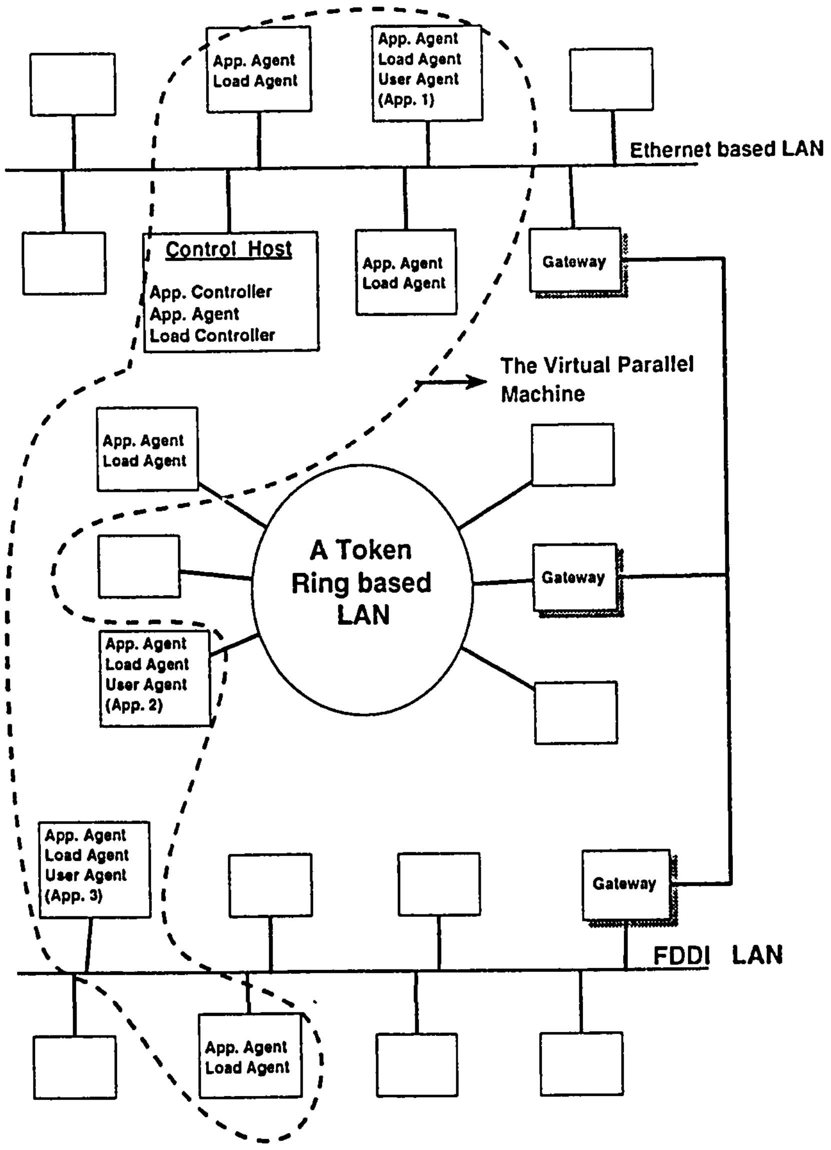

Figure 4.12: Software Architecture of the Load Balancing System

the *sysCheckpoint* system call, the calling process would not be in the kernel mode at the instant of checkpointing. Upon restoration, the process would start normal execution without any overhead or deadlock. This technique could also enforce security by allowing/disallowing certain processes to be checkpointed by others, since checkpointing is done solely by the daemon, which is a controlled process.

This approach has been adopted in the design of the PMS, and the job of checkpointing/restoration has been entrusted to the MD.

Migration

A process can, of its own volition, migrate over to another host, or any arbitrary active process can request it to migrate out of the current host. In order to support the latter case, we need to have asynchronous interaction among these two processes.

Let P_i and P_j be two processes executing on host A. At some time of its execution, P_j, which makes policy decisions regarding migration, decides to have P_i migrated over to some other host H. Since P_j cannot checkpoint P_i directly, it requests the MD to do so on its behalf. At this instant, P_i could possibly be executing in kernel mode. Since PMS disallows migration of such processes, the MD cannot straightaway checkpoint P_i. Hence, it sends a message to P_i asking it to get itself in a *safe* state (.ie. not in kernel mode). The problem now is: *How does P_i know that the MD has sent a message to it?* One approach is to have P_i keep polling its communication channel on which it communicates with the MD. However, this implies having a program-loop in the application's code, so that the channel can be polled in each of its iterations. Enforcing such an approach in the application is impossible, without having the programmer be aware of the need to ensure that the channel must be polled. The objective of the PMS is, however, to relieve the programmer from the intrinsic of the PMS as much as possible. Hence, this approach is straightaway rejected.

The correct method, and the method that has been adopted here, is to use the signalling mechanism of the OS to have asynchronous interactions. The MD would, upon sending the message to P_i, send a signal (SIGUSR1) to it. The PMS library routines that deal with the registration of the process with the MD, would take care of setting up a signal handler for catching this signal (SIGUSR1). Thus, upon receiving SIGUSR1, P_i will go into the signal handler, which gets the process into a consistent state and send back a message to the MD indicating its willingness to be checkpointed. The MD would then use E-Kernel's sysCheckpoint system call to have P_i checkpointed.

Thus, the MD forms an essential component of the PMS, not only to provide migration, but also to support checkpointing of arbitrary processes.

4.5 The Migration Daemon

In order to provide efficient process migration, we introduce the concept of a *Migration Daemon (MD)*. The idea is to have a daemon process executing at every node of the DCS that is willing to participate in migration. (Note: some of the nodes of the DCS may be too susceptible to failures and hence may be excluded from participating in migration). This daemon process is the one which controls the migration of processes from/to the machine. Its place would be that of the "application agent" in Figure 4.12. If a process wants to checkpoint itself, it would invoke the PMS library function *checkpoint(SELF)*. The library function would translate into code that sends a message to the local MD asking it to checkpoint the process which has sent this message. Since the MD is a different process, it would easily be able to checkpoint any other process through the *sysCheckpoint()* system call. The MD also plays a critical role in handling messages destined to the migrant process, as discussed in later sections.

The MD has a major role to play in checkpointing/migrating executing processes. The need for a daemon process is outlined below:

1. To ease implementation and make it efficient, it is better not to have a process invoke the *sysCheckpoint()* system call within its own context (see Section 5.2.1). Hence, it is the MD that will be invoking *sysCheckpoint()* on behalf of a requesting process.

2. The *sysCheckpoint/sysRestore* system calls provide a mechanism by which any process in the ready queue can be checkpointed/restored. Although restrictions can be incorporated to provide user-level security (like not allowing a user to checkpoint/restore any other user's processes), these restrictions would make it difficult to use this mechanism to provide fault-tolerance, wherein the basic requirement is the ability to save/restore any of the processes running on a faulty processor.

   In order to blend security with functionality, without loosing the power, we enforce the restriction that only the *super-user* (root) can use these system calls to checkpoint/restore any arbitrary executing process, whereas normal users can check/restore only processes owned by them. Hence, to support fault-tolerance, we need a process owned by the "root", and MD is that process.

3. In order to detect and tolerate the failure of processes running on a node in the DCS, we need to have a distributed control over the DCS. The distributed control is achieved by the Migration Daemon.

Thus, to fulfill these requirements, a MD handles checkpoint/migrate requests from the user programs executing on its node, as well as from the remote MDs running on other nodes: does the actual checkpoint/restore by invoking the *Checkpoint/Restore* system calls, within its context, on behalf of the calling process; and cooperates with other MDs in salvaging failed processes.

4.5.1 The Migration Protocol

As in any distributed application, the MD has its own protocol - the *Migration Protocol*, for interacting with user processes and the remote MDs. The Migration Protocol (MP) has two components : User-MD interactions, and MD-MD interactions. The essence of these interactions is brought in focus in the following sections.

4.5.2 Independent Applications

Let us consider the simplest case first - Independent Applications, where a process is a stand-alone entity, in that it has no interactions with any other processes executing on the DCS. One can think of four scenarios: A process checkpointing itself at regular intervals of time (for fault-tolerance), a process being checkpointed by another process executing on the same node, a process migrating itself, or a process being migrated out of the host by another process (load-balancer, for example).

In the first case, a process might want to have itself checkpointed at regular intervals of time, so that, in the event of a failure, it could be restored to its checkpointed state, without any significant loss of computation. In this case, the process would use a PMS library routine which sets up a timer and invokes the *checkpoint(SELF)* routine at the specified interval. The *checkpoint(SELF)* routine, after dealing with the open files and/or communication channels (if any) of the process, would send a "CHECKPOINT" request to the local MD asking it to checkpoint its state information by providing its own *pid* (say 'P_1'). The MD would then invoke *sysCheckpoint*(P_1) and send back a 'CONTINUE' message once process 'P_1' has been checkpointed. This sequence of interactions is depicted in Figure 4.13(a).

The distributed MDs, being incharge of providing a fail-safe DCS, have to ensure that the processes (not necessarily all) executing on their nodes are checkpointed at regular intervals of time, so that in the event of failure of the node, these processes may be reinstated on another active node. This situation requires that an MD be able to checkpoint any arbitrary process on its node. For any other process P_i (say, a load-balancer) that wants to checkpoint another process P_j, it has to request the MD to do so on its behalf[4]. Figure 4.13(b) depicts this scenario.

Process P_i invokes the PMS library routine *checkpoint*(P_j), which sends a "CHECKPOINT P_j" message to the local MD. The MD checks whether P_i is permitted to perform this operation on P_j and then forwards this request to P_j. Upon receiving this request from MD, P_j gets itself into a consistent state by checkpointing all its

[4]See Section 4.5 for details

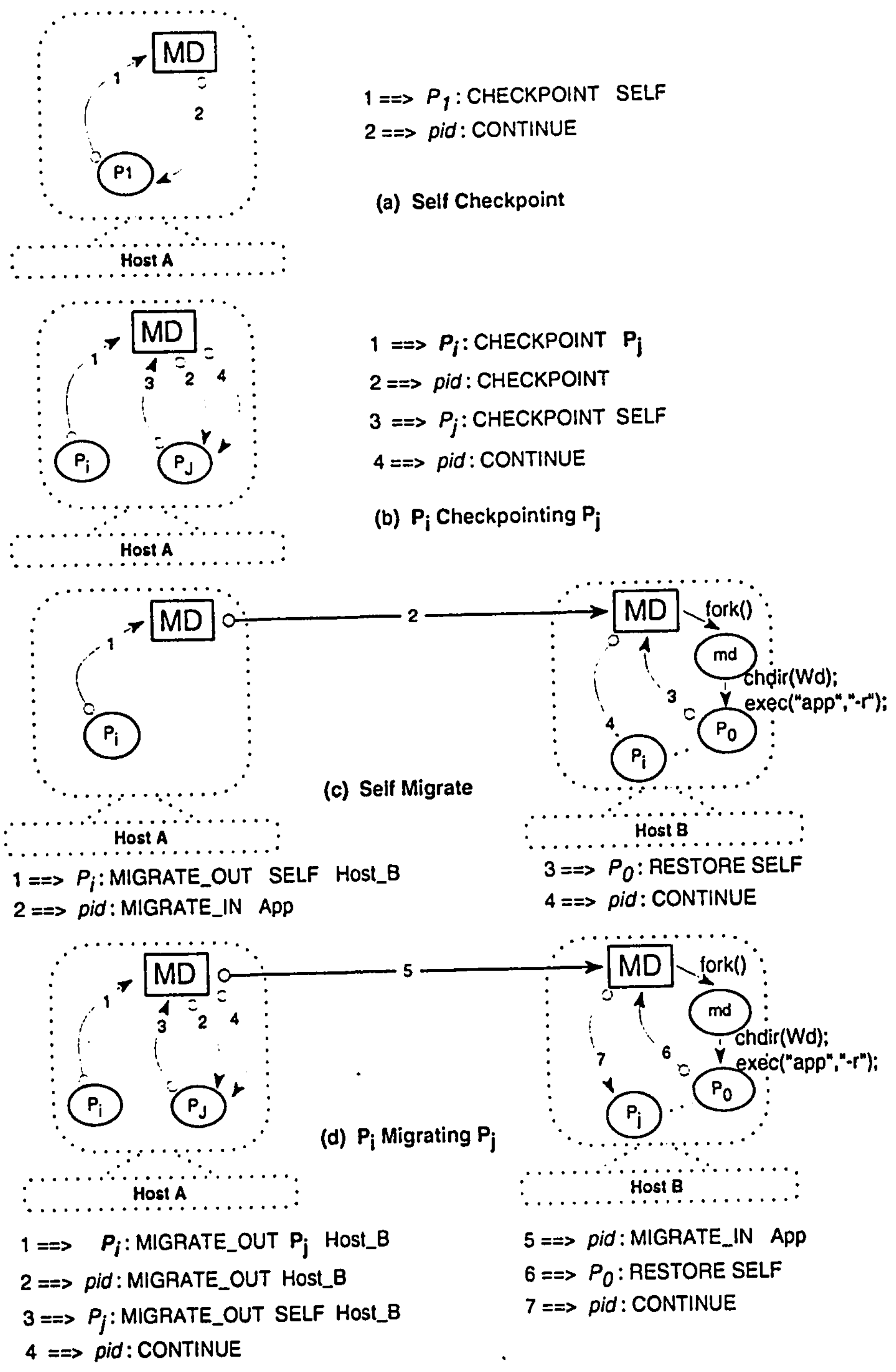

Figure 4.13: The Migration Protocols

open files and I/O channels and calls *checkpoint(SELF)*, thereby asks the MD to checkpoint its state information. The MD then invokes *sysCheckpoint(P_j)* to have P_j checkpointed and sends back the "CONTINUE" message upon completion. P_j then goes on with its normal execution.

The protocol required for a process to migrate itself onto another active node is depicted in Figure 4.13(c). To understand the need for such a situation, let us consider the DCS depicted in Figure 1.1. Suppose the machine with SVR4 is the fastest one on the DCS. The application programmer would like to make use of this fact and execute his application on it. However, after the computations/calculations are done, the program needs to produce its results in the form of graphs on the Laser printer connected to the MACH print-server. So the programmer places the PMS library call *migrateOut(SELF,print-server)* in the code so as to move the process over to the print-server for faster service. This library routine would now expand to code which, after checkpointing the process's files and I/O channels, sends the message "MIGRATE_OUT P_i print-server" to the local MD. The MD then gets P_i checkpointed by invoking *sysCheckpoint(P_i)* and sends a "MIGRATE_IN App"[5] message to the MD on 'DestHost'.

The remote MD, upon receipt of this message, forks a child process *md*, which sets its current working directory to 'Wd' and execs 'App' with the "-r" option so as to force it to go into restoration mode immediately upon execution. Process *md*, which has now become P_0 (i.e. process P_i in state zero - the initial state), immediately sets up its files and I/O as at the checkpointed state and sends a "RE-

[5]'App' is the name of P_i's executable

STORE P_0" message to the local MD. The MD then uses *sysRestore(P_0)* to have P_0 restored to its original checkpointed state and sends a 'CONTINUE' message upon completion, whereby the program resumes its normal execution, as at the instant of checkpointing.

Finally, the protocol that allows a process P_i to migrate-out any other process P_j is shown in Figure 4.13(d) and is self explanatory. Such a scenario occurs in the case of a DCS that has support for load-balancing, dynamic task scheduling or real-time processing.

4.5.3 Distributed Applications

A distributed application is basically a set of cooperating tasks belonging to a single program. These tasks communicate with each other and collectively provide the functionality of the application. From the PMS perspective, a distributed application is just a set of processes (tasks), the major difference being that these processes are involved in communication, with the processes on the same/different host(s) of the DCS.

Thus, apart from the design issues involved in independent application, the PMS now has the additional responsibility of maintaining the communication among these processes, even after they have migrated from their home site (the site of their initial execution/allocation). In order to handle this responsibility, the PMS needs to keep track of the current site of execution of all the processes belonging to the distributed application(s) running on the DCS at any instant of time.

The most common and portable medium of communication is the *socket*. The general sequence of *socket* related system calls needed to setup communication among two processes is depicted in Figure 4.14.

The process/task uses the *socket()* system call to create a new channel that can be used for network communication. The call returns a descriptor which is an index into the *user file descriptor table*. When a socket is created, it does not have any notion of endpoint addresses, neither the local nor the remote addresses are assigned. The task needs to invoke the *bind()* system call to specify the local endpoint address for the socket, which is denoted as an *IP Address - Port number* pair. In other words, it is as if the task is requesting the kernel to deliver to it whatever arrives on that endpoint. Usually, this endpoint is known/built-in to all the tasks of that distributed application. The *recvfrom()* and *sendto()* system calls are used to transfer data among the tasks.

Now, when a task is migrated, its endpoint address (IP-Port) is no longer valid. However, since the applications themselves have been designed to work on static allocation of tasks to hosts, they would be blissfully unaware of the migration of their peers and hence would still try to communicate with them at the same original addresses. It is thus the responsibility of the PMS to translate the invalid original endpoint address into a valid destination referring to the migrant task.

The idea is to develop a function that accepts the original IP-Port address and converts it into the present endpoint address of the migrated task. To do this the PMS needs to maintain a table - *Migration Table*, specifying the original and present address of all the migratable processes on the DCS.

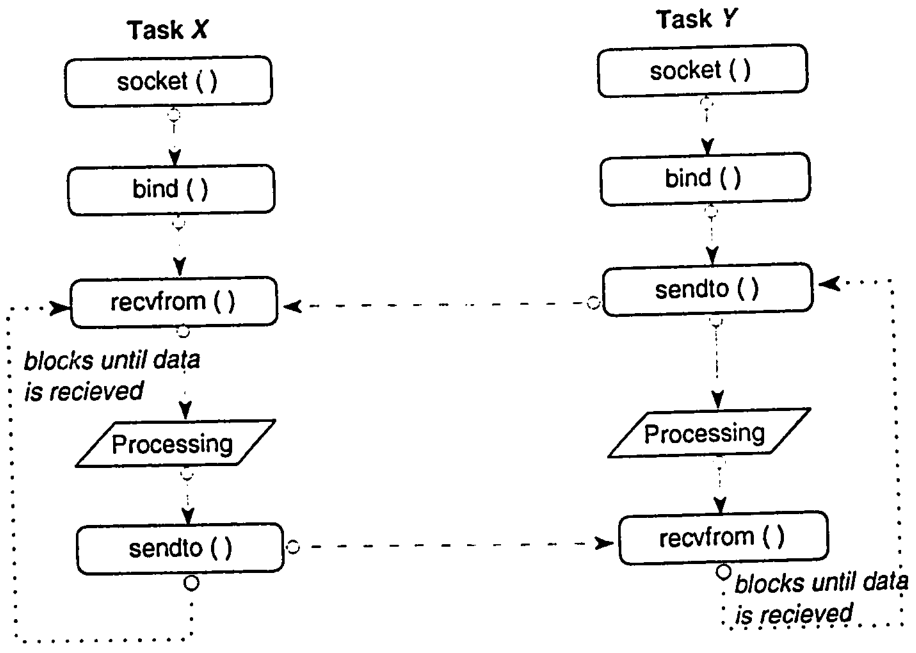

Figure 4.14: General scenario for socket communication

This functionality of the PMS can be provided either at the kernel level or at the user level, as has been discussed in Section 4.4.4. The most suitable candidate is the MD. Since the MD is a user level distributed application, it can easily gather and maintain the *migration table* information. In order to convert the task-specified endpoint address into the actual valid address, we have provided a PMS library function - *msendto()*. This function has exactly the same semantics as that of the *sendto()* system call. When a task uses *msendto()* to send data, the function sends a message to the local MD comprising of the specified endpoint address. The MD then looks up this address in its *migration table* and returns back the present IP-Port combination. This address is then used in *sendto()* to actually send the data.

Maintaining the *migration table* requires additional support, which can be provided by additional PMS library routines which are clones of the existing socket related system calls. When a socket is created, the *msocket()* and *mbind()* routines would inform the MD so that an entry can be created for it in the *migration table*. The MD would then inform all other MDs on the DCS about the availability of this new endpoint. Now, when a task migrates, the routine responsible for checkpointing the open communication channels saves the original endpoint address and sends a message to the local MD, informing it that a specific task is about to migrate out of the host. The MD then updates the task's entry in the *migration table* to indicate that the task is in-migration. If an address lookup request comes at this instant from *msendto()*, the MD would block the reply until the process has been migrated. Upon restoration on the destination host, the migrant task would inform the MD at that site about its old as well as new addresses. The MD updates its own information and then broadcasts this message to all its peers. Thus, at any instant of time, the contents of the *migration table* are maintained in a consistent manner, in that each MD has exactly the same copy of the table. Thus the combination of the *migration table*, PMS library routines, and the set of distributed MDs effectively solves the problem of address resolution for migrant tasks of distributed applications.

4.6 Fault-Tolerance

The PMS provides the basis on which a fail-safe DCS can be constructed. In order to provide fault-tolerance we need a roll-back recovery scheme, and checkpointing is the only possible solution. In order to roll-back, we need to have a latest copy of the process's image. Hence, processes which are to be fail-safe need to arrange such that they get checkpointed at a user-specified time interval. The PMS provides a library routine - *regCheckpoint()*, which takes an interval and expands to code that invokes *checkpoint(SELF)* at the specified interval. The steps involved in a process checkpointing itself are depicted in Figure 4.13(a). Upon failure of the node or the process, the process can be restarted from its checkpointed image (which is saved onto a centrally accessible host, via NFS) by simply executing the process with the "-r" option, as explained in Section 4.5.2.

Such a facility could be used by the fault-tolerance subsystem shown in Figure 3.1. The subsystem would monitor applications running on the DCS and, in the event of failure of any of the processes/hosts, would transparently reinstate them onto active hosts. The issues involved here are those dealing with the migration *policy*, rather than the *mechanism*. Since PMS provides the migration mechanism, the fault-tolerance subsystem would imply incorporating the decision-making logic.

Chapter 5

Implementation Details

The work platform chosen for the implementation of the PMS is a distributed system comprising of a network of PCs running the Linux operating system (a Unix clone for i386+ machines). The study includes development of *checkpoint* and *restore* mechanisms on which the pre-emptive process migration facility is implemented.

Prior to delving into the details of the PMS implementations, let us look into the features provided by Linux, and the reason for choosing it as the implementation bed.

5.1 Introduction to Linux

Linux is a clone of the Unix operating system that runs on IBM PC compatible machines with *Intel-386/486/Pentium* or equivalent processors. It was written from scratch by Linux Torvalds (a graduate student at the Helsinki University of Technology, Finland), in collaboration with an enthusiastic, world-wide group of volunteers interacting through the Internet.

Linux is a full-fledged operating system that provides all the capabilities normally associated with commercial Un*x systems. It supports most of the features found in other implementations of Unix, plus quite a few that aren't found elsewhere. It is a complete multitasking. multiuser operating system and is mostly compatible with a number of Unix standards at the source level, including IEEE POSIX.1, System V, and BSD. All its source code, including the kernel, device drivers, libraries, user programs, and development tools are freely distributable. The major features of Linux can be summarized as follows:

- Support for pseudo-terminals, loadable keyboard drivers, and virtual consoles.
- POSIX compatible job control.
- Kernel emulation for 387 FPU instructions.
- Support for a variety of filesystems such as ext2fs, Minix, Xenix, MSDOS and ISO 9660 CD-ROM filesystem.
- Complete implementation of TCP/IP networking; including SLIP, PLIP and PPP.
- Support for NFS, NIS, FTP, Telnet, NNTP and SMTP.
- Support for demand-paged loaded executables, disk paging and dynamically linked shared libraries.
- Excellent compilers for C, C++, Pascal, Modula-2, Oberon, Smalltalk and Fortran, standard utilities for text/word processing and the X-Window system

- Password security, file protection, multiple logins, virtual memory and multi-tasking.

- Provides workstation capabilities on top of inexpensive PCs.

The most obvious reason for choosing Linux as the implementation platform has been the free availability of its complete source code. This would allow us to make an indepth analysis of the internals of the kernel and help in coming up with a design that would be portable. However, the major reason is that since Linux is still under development and enhancements, it would be possible to incorporate process migration as an integral part of the system. Its free availability and the fact that it runs on PCs, has made it a very popular OS in a span of just a couple of years. It presently enjoys a wide user base in both academic and business environments. Enhancing it with PMS would make it a full-fledged distributed operating system.

5.2 PMS Implementation

As illustrated in Figure 4.1, the PMS comprises of three major components: the E-Kernel, the PMS library, and the set of MDs. The functionality of migration is basically handled by the E-kernel, which provides the *sysCheckpoint()* & *sysRestore()* system calls, and the PMS library routines *checkpoint()*, *restore()*, *migrateIn()* and *migrateOut()* that form the interface to these system calls. The MDs play a major role in checkpointing and migration, apart from ensuring sustained communication among migrating processes belonging to distributed applications. The following sections elaborate the implementation details of the each of these components of the PMS.

5.2.1 The E-Kernel

Checkpointing forms the fundamental mechanism needed to provide process migration capability. Checkpointing a process implies extracting its state information that is dispersed within the basic components of the kernel, namely - File & I/O subsystem, Process Control subsystem, and the Virtual Memory Management subsystem (refer to Section 4.4.1). This makes it imperative to incorporate the checkpoint and restore functionality within the kernel. The *E-Kernel* (or Enhanced-Kernel) is a PMS module that is responsible for extracting a process's state during checkpointing, and reinstating it on the destination host upon migration.

The E-Kernel comprises of a set of kernel-level routines that interface with the existing kernel in the form of two new system calls - *sysCheckpoint()* and *sysRestore()*. The basic functionality of these system calls is to extract/reinstate the process related information from/to the kernel.

Section 4.4.1 describes what exactly comprises the state information of a process. This state information is governed by the operating system on which the process executes. The characteristics of the OS are in turn defined by hardware support provided by the specific machine architecture which it manages. Thus, in the context of this study, the state of a process is determined by the *i486* machine architecture and *Linux* operating system.

Process related state

The process related state basically comprises of its PCB and TSS, as shown in Figure 4.5. The i486 processor provides hardware support for multitasking by storing the processor state information needed for a context switch in a data structure in memory, called the TSS. The fields of the TSS, as shown in Figure 5.1, are divided into two main categories:

1. Dynamic TSS fields which the processor updates with each task switch. These fields store :
   - The general registers (EAX, ECX, EDX. EBX, ESP, EBP, ESI and EDI).
   - The segment registers (ES, CS, FS, DS, GS and SS).
   - The flags register (EFLAGS).
   - The instruction pointer (EIP).
   - The selector for the TSS of the previous task.
2. Static TSS fields the processor reads, but does not change. These fields are setup when a task is created. These fields store :
   - The selector for the task's *ldt.* The logical address of the stacks for privilege levels 0, 1 and 2.
   - The *T* bit (debug trap bit) which, when set, causes the processor to raise a debug exception when a task switch occurs.
   - The base address for the *I/O* permission bitmap[1], which points to the beginning of the map.

[1]The i486 processor can generate exceptions for specific I/O addresses. these addresses are specified in the I/O permission bitmap of the TSS .

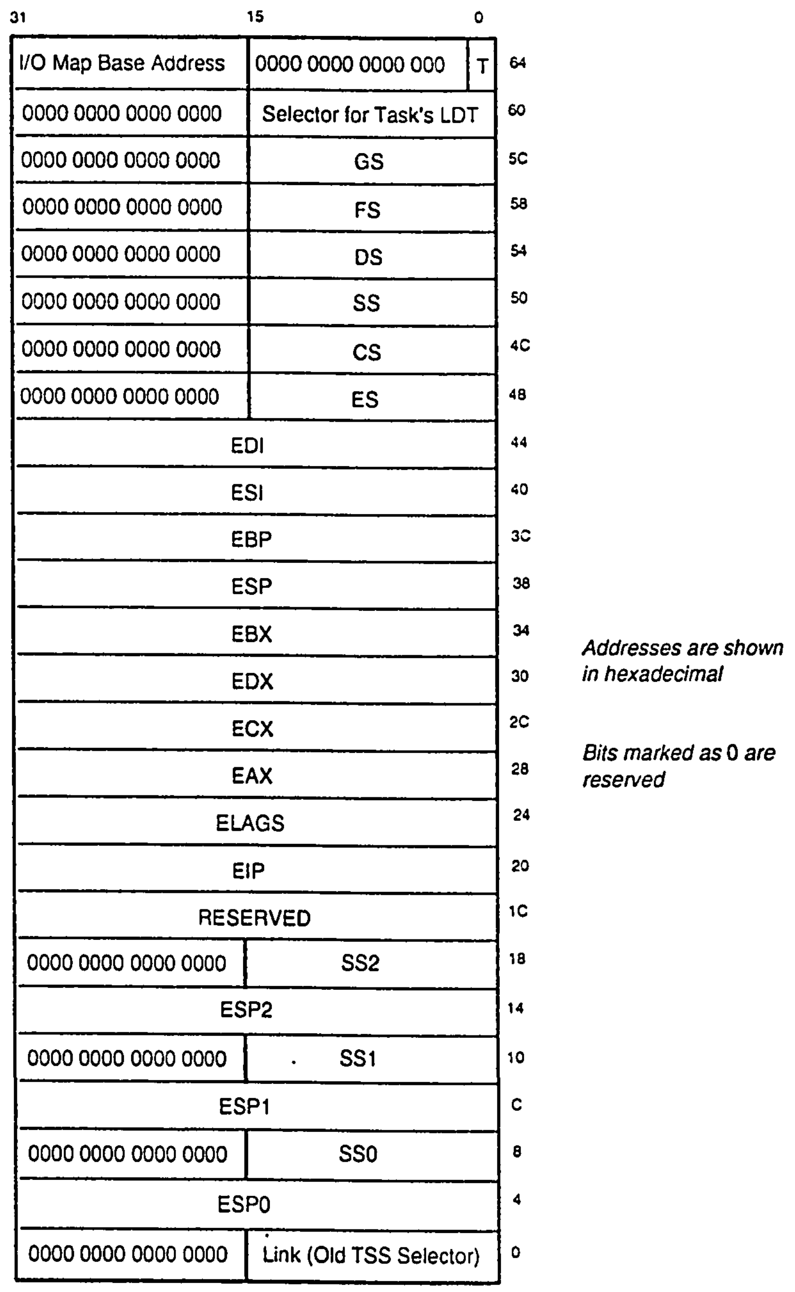

Figure 5.1: The Task State Segment

In the context of *Linux*, the per-process information is maintained in a data structure called the *task_struct*. The *TSS* itself is a part of the *task_struct* and can be accessed directly. The contents of the *task_struct* structure are described below:

- Memory Management information :

Process memory limits :	*unsigned long start_code, end_code, end_data, brk, start_stack;*
Page fault counts :	*unsigned long min_flt, maj_flt, cmin_flt, cmaj_flt;*
Local Descriptor table :	*struct desc_struct *ldt;*
No.of resident pages :	*unsigned short rss;*
The Task State Segment :	*struct tss_struct tss;*
Virtual Memory :	*struct vm_area_struct * mmap;*
Other information :	*int swappable; /* 0 ⇒ Process's pages not swappable */*
	unsigned long kernel_stack_page; / The kernel stack */*
	unsigned long saved_kernel_stack; / V86 mode stuff */*

- Scheduling information :

Present state of execution :	*volatile long state;*
Process priority :	*long counter, priority;*
Alarm intervals :	*unsigned long timeout, it_real_value, it_prof_value;*
Timer values :	*unsigned long it_real_incr, it_virt_incr;*
Accounting :	*long utime,stime,cutime,cstime,start_time;*

- Inter-process communication information :

Process identification :	*int pid, pgrp, session, leader, groups[NGROUPS];*

Process relations :	*struct task_struct *p_opptr, *p_pptr, *p_cptr, *p_ysptr;*
The wait queue :	*struct wait_queue *wait_chldexit;*
Status of signals :	*unsigned long signal, blocked, flags;*
Signal handlers :	*struct sigaction sigaction[32];*
Shared Memory :	*struct shm_desc *shm;*
Semaphores :	*struct sem_undo *semun;*

- File and I/O information :

Access control :	*unsigned short uid, euid, suid, gid, egid, sgid;*
Process's tty :	*int link_count, tty;*
User mask :	*unsigned short umask;*
Inode table :	*struct inode *pwd, *root, *executable;*
File descriptor table :	*struct file * filp[NR_OPEN];*

Virtual Memory related state

The VM related state of a process is in the form of memory pages spread over its entire virtual address space. Figure 5.2 shows the map of the virtual address space from the process's perspective.

Since pages are allocated and deallocated dynamically, care has to to be taken to ensure that only the working set of the process is extracted at the instant of checkpointing. From the perspective of migration, the working set comprises solely of the process's dirty *data* and *bss* pages. Pages belonging to shared libraries and the kernel are common among all the processes executing on that kernel, and hence can be ignored (see Section 4.4.3 for a detailed discussion).

0xc0000000	The invisible kernel	reserved
	initial stack	
	room for stack growth	4 pages
0x60000000	shared libraries	
brk	unused	
	malloc memory	
end_data	uninitialized data	
end_code	initialized data	
0x00000000	text	

Figure 5.2: A user process's view of memory

The E-Kernel system calls

The primary function of the E-Kernel is to provide the functionality for checkpointing and restoring any arbitrary process. The *sysCheckpoint()* and *sysRestore()* system calls would, therefore, need to have a parameter that specifies the *pid* of the process to be checkpointed or restored. These system calls, however, are not directly invoked by the process to be checkpointed, as discussed in Section 4. It is the MD that invokes these on behalf of the calling process. Algorithm 5.1 outlines the *sysCheckpoint()* system call.

The *task_struct*[2], which represents the Process Control Block (PCB) in Linux, is obtained from the *pid*. If *pid* doesn't refer to any valid process entry in the list of executable tasks, the algorithm returns a failure; otherwise the PCB of the process is obtained by using *pid* as an index into the *task list* and a checkpoint-file (CF) is created wherein the process's image can be saved.

[2]The task_struct contains the process's management and accounting information

```
Algorithm : sysCheckpoint
Input : Process Identifier (pid)
Output : Status (Success/Failure)
{
    if(pid refers to an entry in the task list)
        Get PCB corresponding to pid;
    else return(FAILURE);
    Setup the checkpoint-file (CF) for saving the process's image;
    Freeze the process by removing its entry from the ready-queue;
    Marshall the contents of the PCB to the CF;
    Marshall the contents of the Task State Segment (TSS);
    Set PDBR to that of process pid;
    for(all dirty pages in the process's data & bss space)
        if(page is dirty)
            { Save the virtual address of the page;
              Save its contents;
            };
    Save the contents of the process's user-stack;
    Reset the PDBR to that of the currently executing process;
    Restart the process pid by placing its entry back in the ready-queue;
    Close the CF;
    return(SUCCESS);
}
```

Algorithm 5.1: The *sysCheckpoint()* system call

The CF is always created at a fixed checkpoint directory (say "/checkpoint"). This fixed location would ensure that the file remains accessible from any other machine on the DCS in the event of the failure of the node of execution of the process. This is done by using NFS with the checkpoint directory at one (or more) node acting as the file server and having all other nodes mount this directory at the same mount point ("/checkpoint").

Once the CF is successfully created, the process identified by *pid* is stopped by temporarily removing its entry from the *ready-queue*. The contents of the *task_struct* are mainly pointers to other data structures. These pointers may not be valid for the restored process and hence they need to be marshalled and saved to CF. The Task State Segment (TSS), which contains the context switching information that represents the snapshot of the execution state of the process in terms of its register values, is then marshalled to the CF. Since the process *pid* cannot be in kernel mode at this instant, the contents of the kernel stack[3], need not be saved.

Since this system call is invoked by the MD on behalf of the calling process, it would not have any access to the calling process's address space, which is protected by the kernel's segmentation mechanism. In order to overcome this, *sysCheckpoint* has to manipulate the segmentation registers and the PDBR (Page Directory Base Register). The PDBR is therefore made to point to the base of the process *pid*'s address space, which is stored in the CR3 register of its TSS. The contents of all its

[3]The stack used by the kernel in the event of interrupt/exception while executing in the kernel mode

dirty data and bss pages are then saved to CF along with their virtual addresses. We need to save the virtual addresses due to the possibility of *holes* existing in the process's address space. By noting these virtual addresses, we can setup the pages at their exact locations upon restoration.

The contents of the user stack[4] are then dumped to CF and the PDBR restored to its original value. The process *pid* is restarted by placing its entry back into the *ready-queue* and finally the CF is closed before returning with a *success* status.

A reciprocal approach is adopted in restoring a process. The system call *sysRestore()* is not executed within the context of the calling process but handled by the MD. This algorithm is outlined as Algorithm 5.2.

The *pid*, specified as a parameter to the system call, is used as an index into the list of currently active processes (*task list*) on that machine. If *pid* refers to a valid entry, its corresponding PCB is obtained, otherwise the system call returns with a FAILURE status. The checkpoint-file (CF) that contains the image of the corresponding checkpointed process is then opened for reading. The process *pid* is then frozen by disabling its entry in the *ready-queue*. The contents of the saved PCB are read and carefully marshalled onto the PCB of *pid*. Since all the process management and accounting fields of the PCB, such as uid, gid, euid, egid, start_time, etc. are restored to the original saved values, this process (*pid*) will behave exactly as the original checkpointed process and will have precisely the same access restrictions and privileges. The context switching information of the checkpointed process

[4]The process's private stack

```
Algorithm : sysRestore
Input : Process Identifier (pid)
Output : Status (Success/Failure)
{
    if(pid refers to an entry in the task list)
        Get PCB corresponding to pid;
    else return(FAILURE);
    Open the checkpoint-file (CF) that contains the process's saved image;
    Freeze the process by removing its entry from the ready-queue;
    Read saved PCB from CF;
    Map the contents of this PCB onto pid's PCB;
    Read saved TSS from CF;
    Map the contents of this TSS onto pid's TSS;
    Set PDBR to that of process pid;
    while(not end of CF file)
        { Read address and contents of a page from CF;
          if(address not in pid's virtual space)
              { Get a free physical page;
                Map its virtual address to that read from CF;
              };
          Disable Copy-On-Write (COW) for this new page;
          Overlay its contents with the one read from CF;
          Enable COW for this page;
        };
    Close CF;
    Reset the PDBR to that of the currently executing process;
    Restart the pid by placing its entry back in the ready-queue;
    return(SUCCESS);
}
```

Algorithm 5.2: The *sysRestore()* system call

is read from CF and mapped onto the TSS for *pid*. The PDBR, which now points to the page directory of the currently executing process, is set to point to that of *pid* and the segment registers are modified so as to enable access to the address space of *pid*. The virtual address and contents of a page are read from the CF, until the CF file is exhausted. For every page read. if the virtual address read is not in the address space of process *pid*. we need to obtain a free page from the kernel and ensure that it is made available at exactly the same virtual address that is read. To enable write to this page. we need to disable the COW (Copy-On-Write) mechanism for that page. The contents of the page read are overlayed onto this page and COW re-enabled. The CF is then closed. the PDBR reset and the process *pid* restarted by re-enabling its entry in the *ready-queue*.

Since these are the only two major routines that are added to the kernel, providing the migration functionality on any other version of Unix would simply require customizing these two system calls to suit that specific operating system environment. This makes the PMS highly portable. The rest of the functionality of PMS is provided at the user-level by the library routines and the MDs executing on the nodes of the DCS.

5.2.2 The PMS library

The observant reader might have noticed that although we claim checkpoint/migration of processes with open files and communication channels, we do not handle either of these in our checkpoint/restore algorithms.

The major problem in dealing with open files is the impossibility of converting an inode number to the file's full pathname. The existing data structures maintained by the filesystem do not provide that information that is needed in order to traverse in a reverse fashion and extract the pathname given an inode number. Incorporating the ability to checkpoint/restore open files based on their inode numbers would require some significant modifications to the existing kernel code related to file handling/manipulations, thereby making PMS difficult to port to other versions of Unix.

Maintaining easy portability has been the major reason for splitting up the migration mechanism into two basic components - E-Kernel and PMS library routines. The library routines deal with the checkpoint/restore of open files and communication channels, apart from providing support for migration of processes by interacting with the MDs. However, a major drawback of this decomposition is that we need to recompile the user applications after adding these library routines to make them migratable. If this feature were incorporated in the kernel itself, any user application would be migratable without the need for recompiling/recoding, thereby providing the highest flexibility. Section 4.4.2 discusses the other possible alternatives that were available along with their merits and drawbacks.

To support migration, the PMS needs to maintain additional information related to the execution site of processes and their current communication state. This information is distributed among the MD at each site and the processes executing on that site. Further, each process needs to maintain information about its open files and communication channels. The contents of these data structures and their role in migration is depicted in Figure 5.3.

The *MTable* (Migration Table) is a data structure maintained by each MD. In

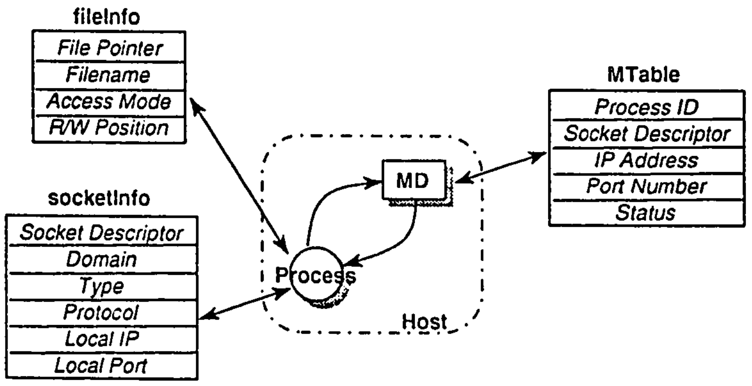

Figure 5.3: Data structures maintained by the PMS

fact, this is a distributed data structure that is maintained globally by the cooperating set of distributed MDs. The contents of the *MTable* is maintained exactly the same at all the MDs through the migration protocol described in Section 4.5.1. *fileInfo* and *socketInfo* are per process data structures. Every migratable application needs to include the "Migration.h" header file so that these data structures get compiled into the code of the process. The *fileInfo* structure, as its name suggests, is used to save the open-files related information that is needed during restoration. Similarly, the *socketInfo* structure is used to hold the socket related information for the sockets that were active during checkpointing. These data structures are basically used by the PMS library routines and their role in migration can be understood by looking into the functionality of these routines.

*mInit(int argc, char *argv[], char *type)*

This routine is to be used by every application that needs to be migratable. This should be the first executable statement within the program. The parameters *argc* and *argv* are the command-line argument count and pointer to the process's environment string respectively, while *type* can be one of *regCheckpoint Interval* or *regMigration*.

The type *regCheckpoint* is used to provide fault-tolerance, in cases where the process wants to have itself checkpointed at regular intervals of time. The additional argument *Interval* is then treated as the interval between two checkpoints. *mInit* sets up an interrupt handler to trap the SIGALRM interrupt and uses the *alarm()* system call to have the handler activated at the specified interval. Upon receiving the interrupt, the alarm handler uses the PMS library function *checkpoint(SELF)* to get the process checkpointed. The functioning of this routine is outlined in Algorithm 5.3.

In order to be migratable, the program has to get itself registered with the local MD so that the MD and other processes executing on the DCS know about its existence. It therefore invokes *mInit* with the 'type' argument as *regMigration*. This causes *mInit* to setup a channel for communicating with the local MD and sends a message to the MD specifying its *pid* and the name of its executable file *App*. MD creates an entry in its data structure for this process, to be used during migration of this process, either by the process itself or by some other process.

When a process is to be restored from its checkpointed image, it simply needs to be executed with the "-r" argument in its command line. This would cause the process to be executed from scratch. However, being the first executable statement in the program, *mInit* checks for the presence of the "-r" option in the command-line and recognizes it to mean that

```
Algorithm : mInit
Input : Command-line arguments (argc & argv), and Initialization type (type)
Output : Status(SUCCESS/FAILURE)
{
    Get current ip-addr and pid;
    Setup signal handler for SIGUSR1;
    Create a new socket and connect it to the local MD;
    If(type == REG_CHECKPOINT)
        { Extract the value of interval specified in type;
          Setup handler for the SIGALRM signal;
          Invoke alarm(interval) to setup alarm every interval seconds;
        }
    else if(type == REG_MIGRATION)
        { Compose message "REG_MIGRATION pid App";
          Send message to local MD;
        }
    else return(FAILURE);
    if(argv[1] == "-r")
        Invoke PMS library function restore(SELF);

    /** This point is reached only during the normal execution **/
    Store current ip-addr and pid as orginalIP and originalPID;
    return(FAILURE);
}
```

Algorithm 5.3: The *mInit* PMS library function

the process wants to be restored from its saved image. It therefore goes into restoration mode by setting up a communication channel with the local MD and invoking the library function *restore(SELF)*, which sends a message requesting its process image be overlayed with the saved one. *originalIP* and *originalPID* are PMS variables maintained by the process and are used to update the entries of the *MTable* whenever the process migrates.

checkpoint(int pid, int flag)

This is the programmer's interface to the *sysCheckpoint* system call. Algorithm 5.4 describes its functionality. Before invoking *sysCheckpoint*, this routine invokes the *checkOpenFiles* and *checkOpenSockets* routines to get all the information related to the open files and communication channels of the process saved into the *fileInfo* and *socketInfo* data structures. This information is used during restoration to setup the files/channels exactly as they were at the instant of checkpointing. The *checkpoint* routine then sends a 'CHECKPOINT *pid*' message to the local MD requesting the actual checkpointing of the process, as depicted in Figure 4.13(a). After completion of the checkpointing, the MD sends back a 'CONTINUE' message, whereupon the process resumes normal execution. The *flag* argument of this routine can take three values: BLOCK, NO_BLOCK, and EXIT. The BLOCK argument implies that the process is to remain blocked even after completion of checkpointing, NO_BLOCK implies resumption of normal execution, while EXIT specifies that the process is to exit after checkpointing. These flag values are used while migrating the process.

```
Algorithm : checkpoint
Input : Process ID (pid), and flag
Output : Status
{
    If(pid != SELF)
            Send "CHECKPOINT pid" message to the local MD;
    else
          { Set mypid to the ID of this process;
            For(every open file)
               { Save file-related state in fileInfo;
                 Close the file;
               };
            For(every socket)
               { Save socket-related information in socketInfo;
                 Close the connection;
               };
            Send "CHECKPOINT mypid" message to the local MD;
            Await for "CONTINUE" message from the MD;
          }
     return(checkpoint status);
}
```

Algorithm 5.4: The *checkpoint* PMS library function

restore(int pid)

This routine is normally invoked by *mInit*. To restore a process, the executable of the process is invoked with the "-r" argument. Upon execution, *mInit* interprets this argument to mean restoration and sends a 'RESTORE SELF' request to the local MD to get itself restored. After the process's image has been overlayed, the process would be in exactly the state at which it was checkpointed - awaiting for the "CONTINUE" ,message from the MD, as depicted in Algorithm 5.4. However, since the process would now be at the end of the *checkpoint* routine, it would simply return back to its normal execution. Since *sysRestore()* doesn't setup the files and sockets that were open by the process, the process would now be executing in a state where it is not aware of the files/sockets that were originally open. This problem of restoring the files and sockets is handled by the SUGUSR1 signal handler that was setup by *mInit*. The MD, after completion of *sysRestore()*, would send the SIGUSR1 signal to process *pid*, prior to sending the "CONTINUE" message. Upon receiving this signal the handler would take up the job of setting up the open files and communication channels (Algorithm 5.6) through the information contained in the relevant data-structures of the checkpointed process, by invoking the *restoreOpenFiles* and *restoreOpenSockets* routines.

*migrateOut(int pid, char *destHost)*

A typical use of this routine could be by an application that makes the policy decisions for migration, say the Load-Balancer. When invoked, this routine sends a 'MIGRATEOUT *pid destHost*' message to the local MD. The MD would then request the process identified by *pid* to get itself migrated over to the host *destHost* by sending it the message 'MIGRATE_OUT SELF *destHost*'. Process *pid*, upon receipt of this message, gets itself checkpointed by invoking *checkpoint(SELF)* and asks the MD to forward the 'MIGRATE_IN *APP*' message to the *destHost* machine. The remote MD at *destHost* would then get this process, identified by application name (APP), restored onto its kernel by invoking the *migrateIn* function. The protocol involved is discussed in Section 4.5.1.

*migrateIn(int pid, char *fromHost)*

This routine is normally invoked by the MD at a remote host upon receipt of the 'MIGRATE_IN *APP*' message form its peer. The MD then forks off a child process and *exec*'s the application identified by *App* with the '-r' argument, forcing it to go into restoration mode immediately upon start of execution. The child process would then go through *mInit* and *restore* to obtain its image tat was checkpointed on node *fromHost* and restores it onto the node at which this routine is executed.

*mfopen(char *fileName, char *mode)*

The user applications must use this routine instead of the standard *fopen* for opening files. The syntax of this routine is exactly the same as that of *fopen*. The function of *mfopen* is to create an entry in the *fileInfo* data structure and store the pathname and mode of access, prior to invoking *fopen* to do the actual job of opening the file specified by *fileName*. This saved information forms part of the checkpointed state of the process.

*mfclose(FILE *fp)*

This is the proxy for the standard *fclose* routine. It removes the entry for the open file identified by *fp* from *fileInfo* and invokes *fclose* to close it.

checkOpenFiles(void)

This routine is invoked implicitly by *checkpoint* and does the job of checkpointing the open files. It goes trough all the open files of the calling process and stores the current read/write position of the file in the *fileInfo* data structure. It then closes the files by invoking the standard library routine *fclose*.

restoreOpenFiles(void)

The *restore* routine makes use of this routine to get the open files of the checkpointed process restored to their original state, after having done with restoring the process's state information. This routine looks up the *fileInfo* data structure and reopens the files using *fopen*. Files that were originally opened in the WRITE mode need to be reopened in APPEND mode, as otherwise the previous contents will be destroyed. It then sets the read/write pointer of the file to the position at the instant of checkpointing, by extracting this information from *fileInfo*.

msocket(int domain, int type, int protocol)

The functionality of this routine is akin to that of *mfopen*, except that here we are concerned with open communication channels rather than open files. It creates an entry in the *socketInfo* data structure and stores all its arguments so that they can be used while reopening the socket during restoration. Finally, it invokes *socket()* system call to get a new channel allocated for communication.

*mbind(int sd, struct sockaddr *addr, int addrLen)*

The application must use this routine in lieu of the existing system call *bind()*, in order to bind a local end-point to the socket. This routine stores this end-point address in the *sockInfo* data structure, which is needed upon restoration in order to have the process setup its communication channels. It then sends an 'MT_UPDATE' message to the local MD specifying the end-point address at which the socket has been bound and presently available for communication. This information is needed by the MD to provide communication among migrating processes, as explained in Section 5.2.3.

*msendto(int sd, char *msg, int msgL, int flags, struct sockaddr *dest, int AddrL)*

This is the major routine that provides transparent communication among migrating processes. The functioning of this routine is depicted in Algorithm 5.5. It basically sends the specified destination end-point address to the MD for lookup in its *Migration Table*. The MD sends back the current valid end-point address, if the process has been migrated. Otherwise, it simply reflects back the same message. Upon receipt of the reply from the MD, the data is sent directly to the *ip-port* specified in the reply.

mclose(int sd)

This routine removes the entry for *sd* from the *socketInfo* structure. It also send an 'MT_DELETE' message to the local MD, along with the present end-point address, asking it to delete the entry belonging to this socket.

```
Algorithm : msendto
Input : socket descriptor (sd), data, length, flags, destination address (ip-port)
Output : Status
{
    Set retries to 0; Set delay to MAX_DELAY;
    While(retries < MAX_RETRIES)
        { Compose message "MT_CHECK_STATUS sd ip-port";
          Send message to local MD;
          Await reply from MD;
          Set stat to the status field of reply;
          if(stat== AVAILABLE)
            { Extract current ip-port from the reply;
              Send data to the end-point address ip-port;
              return(SUCCESS);
            }
          else if(stat == IN_MIGRATION)
            { Increment retries;
              Sleep for delay seconds;
              Continue;
            }
          else return(stat);
        };
    return(FAILURE);
}
```

Algorithm 5.5: The *msendto* PMS library function

checkOpenSockets(void)

This is implicitly called by *checkpoint* while checkpointing the process. It scans through the *socketInfo* structure and for every open socket, sends the "MT_CHANGE_STATUS *sd ip-port* IN_MIGRATION" message to the local MD. It then closes the socket using the *close()* system call.

restoreOpenSockets(void)

This routine is invoked by *restore* to re-establish the connections that were open at the time of checkpointing. It creates a new socket as per the specification saved in the *socketInfo* data structure. The new socket descriptor is then mapped at the old value, so that the application can still refer to the same old descriptor. If the socket was bound at the instant of checkpointing, it is rebound and the new end-point address *ip-port* is obtained. An "MT_MIGRATE" message is then sent to the local MD, specifying the previous and the present *ip-port* address.

The PMS provides the flexibility that a process can checkpoint/migrate any other process. During its normal execution, a process, wouldn't know when to expect such a request from the MD. Obviously it cannot keep waiting for the message nor can it keep polling the MD, as discussed in Section 4.4.5. This scenario implies the need for the process to arrange for asynchronous intimation of the arrival of a message. The *mInit* function which is invoked by the process during the initial stages of its execution (either normal or restored), does the job of setting up a signal handler to catch the SIGUSR1 signal. The functionality of this handler is described in Algorithm 5.6.

```
Algorithm : sigusr1Handler
Input : None
Output : None
{
    Read the incoming message;
    Extract the type and status fields from the message;
    If(type == CONTINUE)
        { If(status == RESTORED)
            { For(each entry in fileInfo)
                { Set filename to fileInfo.Filename;
                  Set mode to fileInfo.Mode;
                  If(mode == WRITE)
                    Set mode to APPEND;
                  Open the file filename in access mode mode;
                  Map new file-pointer to fileInfo.FilePointer;
                  Set current read/write position to fileInfo.Position
                };
              For(each entry in socketInfo)
                { Create a new socket as per the entry;
                  If(socket was bound)
                    { Bind socket to any available ip-port;
                      Send "MT_UPDATE ip-port" message to local MD;
                    };
                };
            };
          return;
        }
    else if(type == CHECKPOINT)
            Invoke checkpoint(SELF) to get this process checkpointed;
    else if(type == MIGRATE_OUT)
          { Extract destHost from the message;
            Invoke migrateOut(SELF.destHost) to migrate over to destHost;
          }
    else return;
}
```

Algorithm 5.6: The *sigusr1Handler* PMS library function

Prior to sending any message to a process, the MD would send the SIGUSR1 signal to it. The signal handler would then read the incoming message and extract the *type* and *status* fields of the message. If the *type* field specifies "CHECKPOINT", the signal handler would invoke *checkpoint(SELF)* to get the process checkpointed. If *type* specifies "MIGRATE_OUT", it would extract the *destHost* filed from the message and call *migrateOut(SELF,destHost)*, thereby having the process migrated to *destHost*. However, if *type* is "CONTINUE", it would mean that the process has been either checkpointed or restored recently. If *status* is "CHECKPOINTED", the handler simply exits. Otherwise, if *status* specifies that the process has been "RESTORED", it invokes *restoreOpenFiles* and *restoreOpenSockets* to have the process's files and sockets restored to their original state.

5.2.3 The Distributed MD

The concept of a daemon process - the *Migration Daemon*, has been introduced to enable migration to be implemented at the user-level. The MD is a privileged process that is run *setuid* to root. It is a true daemon process and gets started during system boot time through the */etc/rc* files. The need and benefits of dedicating a separate process to handle checkpoint/restore has already been mentioned in the Section 4.5. The MD is, in reality, a distributed program, with the component MDs executing on each of the host of the DCS that is willing to participate in process migration. The interactions of the processes with the MD, and among the MDs running on the hosts is depicted in Figure 4.2.

The set of MDs perform two major tasks: Pre-emptive migration of active processes, and maintaining the communication among the components processes of a migratable distributed application.

In order to provide migration, the MDs follow a well-defined protocol, called the *Migration Protocol.* The details of this protocol are discussed in Section 4.5.1. The role of MD in checkpointing a process is depicted in Figure 4.13(a) and (b). The interactions among the process and the MDs during the course of migration is illustrated in Figure 4.13(c) and (d). However, another major role played by the MD is in sustaining transparent communication among the migrant processes belonging to distributed applications. This scenario is depicted in Figure 5.4.

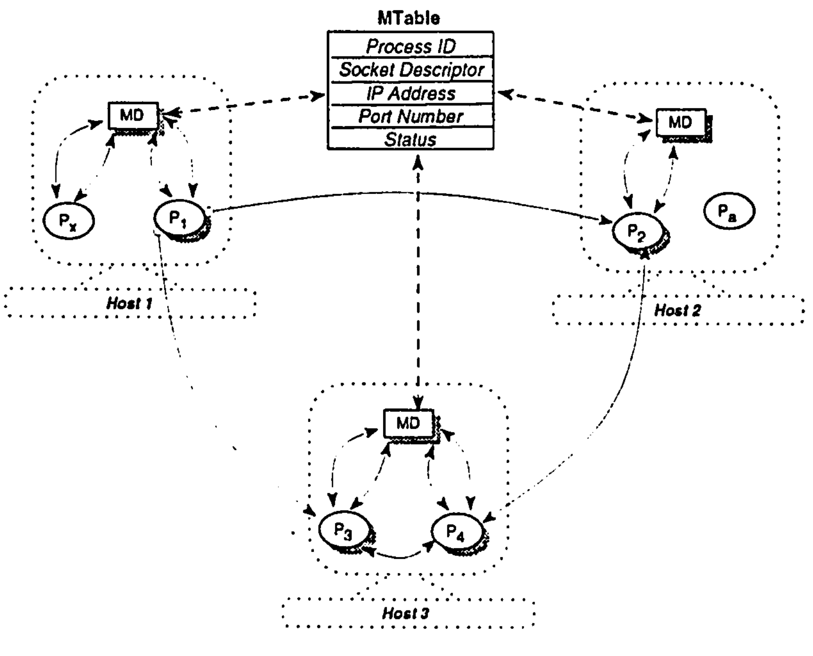

Figure 5.4: Role of MD in a distributed application

Role of MD in a distributed application

The scenario depicted in Figure 5.4 is a set of process - P_1, P_2, P_3 and P_4, belonging to a distributed applications. These processes interact with each other and cooperate to produce the desired results. Let the initial allocation of these processes be as shown in the figure. Since these processes are migratable, upon execution, they would invoke the PMS library function *mInit* to get themselves registered with their local MDs. Consider one such scenario - process P_1 invokes *mInit*. This would cause it to send a "REG_MIGRATION P_1" message to the MD at *Host 1*. Upon receipt of this message, the MD would enter the pid (P_1) in its *MTable*, and broadcast this message to all other MDs on the DCS, which would then update their own *MTable*. The idea is that whenever the contents of the *MTable* are to be update, the MD which is the producing the change will be responsible for broadcasting this change to all other MDs, in order to ensure that the contents of the *MTable* are maintained consistently, such that it appears as a single global table.

Later, at some instant, P_1 would create its socket through *msocket*, thereby generating the message "MT_UPDATE P_1 *sd*" leading to the *socket descriptor* field of the *MTable* being updated. To bind this socket, P_1 would invoke *mbind* causing the *IP Address* and *Port Number* fields of the *MTable* entry to be filled in. The *status* of the socket would then be marked as AVAILABLE. A similar sequence of actions by P_2 and P_3 would create two more entries in the (virtual) global *MTable*. Now, when P_1 wants to send some data to P_3, the *msendto* routine would cause it to send a request to its local MD for a *MTable* lookup of the end-point address identified by the (IP-Address,Port-Number) pair. The MD would check the corresponding fields of the table and reflect back the same message, since the *status* field is set to the flag value AVAILABLE. P_1 would then extract the end-point address from the incoming message and blindly send the message at that address. From the application's perspective, this process of address translation is completely transparent.

Suppose that P_2 has to be migrated to *Host 1*. The *migrateOut* routine would cause the *status* field of the corresponding entry of the *MTable* to be marked as IN_MIGRATION. If any address-resolution requests for P_2 arrive at this instant at any of the hosts, the MD would cause the calling process to sleep for a pre-defined interval and retry. Meanwhile, upon completion of migration, P_2, which is now P_2', would go through *mInit* and *restore* functions. These functions would restore the socket and create a new entry in the *MTable* for P_2'. Further, the old entry would be made to point to this new one. Now, if P_1 send a message at the old address of P_2, the MD would transform it into the new one referred to P_2' by *MTable* lookup.

Thus, the only overhead involved in communication is that of the table lookup by the local MD. The actual communication occurs directly among the relevant processes and doesn't go through the MD, which would have been a major bottleneck.

Chapter 6

Experimentation & Results

The provision of a process migration facility opens up a lot of new areas - dynamic load-balancing, real-time scheduling, fault-tolerance etc. However, for the purpose of demonstrating the functionality of PMS, within the scope of this study, our experiments were classified into four basic classes : *Independent Applications*, *Distributed Applications*, *Hybrid Applications* and applications demonstrating the feasibility of *Fault-Tolerance*. The following sections briefly outline these experiments.

6.1 Independent Applications

This class of applications comprise of user-programs which execute as a single process. This process would be a stand-alone entity, in that it has no communication/interaction with any other processes currently running on the system. Such processes can further be classified into two types: *Compute Intensive* and *I/O Intensive*.

6.1.1 Compute Intensive Processes

A *compute intensive* or *CPU intensive* process is one which spends most of its time in doing some computation. Typical examples of such applications are simulation programs, Neural Network or Artificial Intelligence based programs and scientific applications. Such applications are usually long-running and execute for hours or days. The time spent in checkpointing such a process is proportional to the amount of memory used by it, which is in turn governed by the size of the data structures used by the program. A representative process that was selected for our experimentation was that of an application that computes the value of a formula for infinity.

6.1.2 I/O Intensive Processes

An *I/O intensive* process is one that deals mostly with files. A heavy percentage of its time is spent in reading data values from one or more files and generating the results in the same/different file(s). Quite a significant number of applications that exist on Un*x belong to this category. Examples include *sort, tr, nroff* and *troff*.

The representative process in this class is one that reads data from two different files and merges them. character-wise, into a single output file. A reciprocal of this process would be one that splits a huge file, character-wise, into two separate files. Both these case were tested on PMS. The size of each of the data files was approximately 18 MB. The results produced without migration were compared with that produced with the process being migrated around the DCS during the course of its execution. The results were found to be the same in both the cases, thereby proving the consistency and correctness of the migration mechanism provided by PMS.

6.2 Distributed Applications

Distributed applications are programs that comprise of two or more tasks or processes. which interact (communicate) among themselves to produce the desired results. The processes belonging to such applications can therefore be categorized as *communication intensive*. The representative application of this class that was used for testing the PMS comprised of a server process and n client processes. The clients produce some data and send it to the server which manipulates it and sends it back. In order to test the persistency of communication provided by PMS, a process was chosen at random and migrated over to an arbitrary host. This sequence of random migrations was conducted for a given number of iterations. The results obtained were found to be the same with/without migration.

6.3 Hybrid Applications

The processes of this class, as the name suggest, exhibit the characteristic of both compute intensive and I/O intensive processes. For the purpose of experimentation, the process selected was of an application that reads from a file and sends the data to its component process. The component process would then lookup the data value received into its database (another data file), compute a new value and return back another set of data. The process was migrated during its execution and the data produced were compared with those from a stationary execution. The results were found to be the same in both the cases.

6.4 Fault-Tolerance

In order to provide a fail-safe environment, a process can have itself registered with the PMS, specifying the granularity of reliability needed by it in the form of an interval between two successive checkpoints. The PMS then takes charge of having the process checkpointed regularly at that specified interval, for the duration of the process's execution. In case of failure of the process or its execution site, it can be restored on any other host of the DCS simply by invoking it with the "-r" command-line option. This functionality of the PMS was tested with over 100 runs of processes belonging to all the three classes, and it was found that the processes were instantly restored back to their latest checkpointed state.

6.5 Timing Measurements

6.5.1 Checkpointing

The time taken to checkpoint a process can be viewed from two different perspectives: *User* and *System*. From the user's perspective, the checkpointing time can be considered as the time interval between the instant the *checkpoint* PMS library function is invoked and the process gets checkpointed. This checkpoint time was measured for 100 random runs of the representative processes belonging to the class Compute Intensive, I/O Intensive, Communication Intensive and Hybrid applications. The results are plotted as Figure {6.1}, {6.2, 6.3, 6.4}, {6.5} and {6.6} respectively. The average user-level checkpoint time for these processes is specified in Table 6.1. and compared in the graph shown in Figure 6.7.

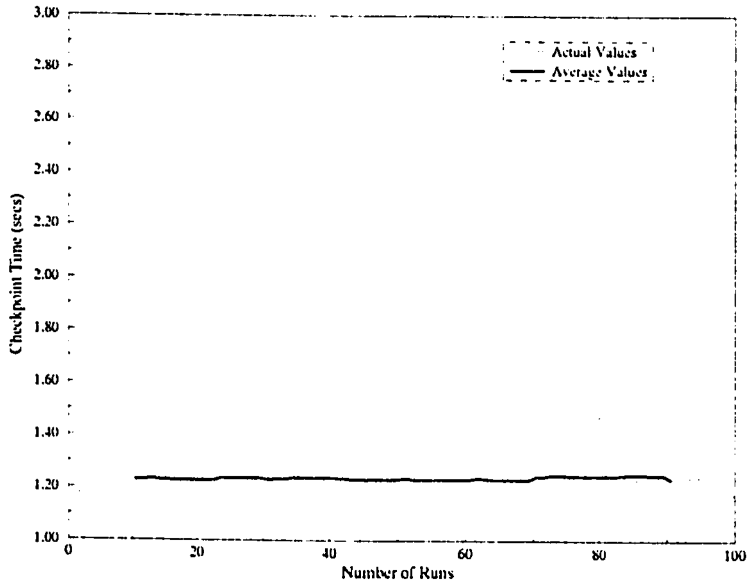

Figure 6.1: The Checkpoint times for a *Compute Intensive* process

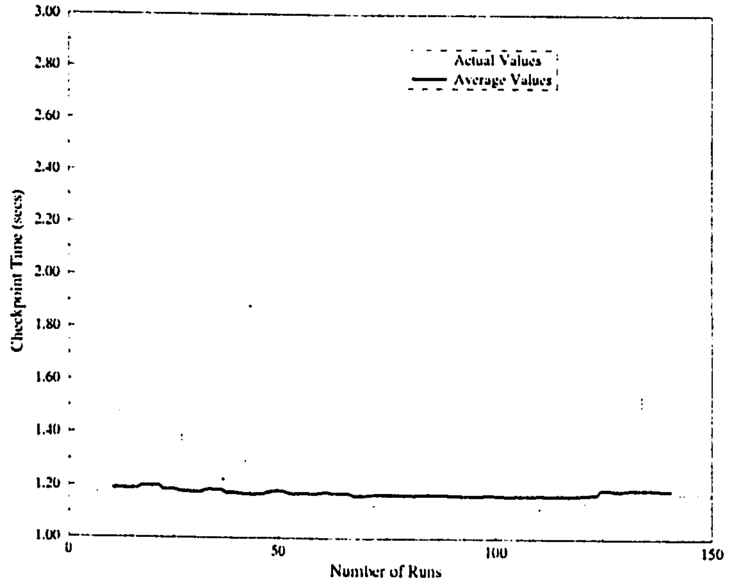

Figure 6.2: An *IO-Intensive Intensive* process with 1 file

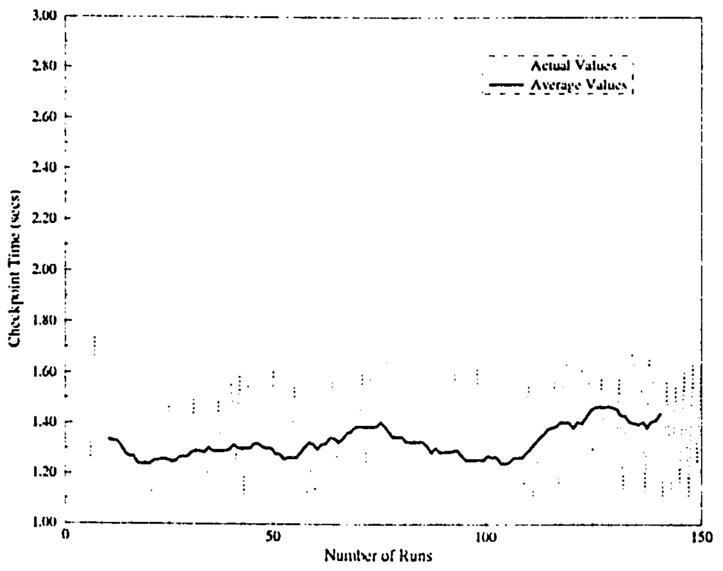

Figure 6.3: An *IO-Intensive Intensive* process with 2 file

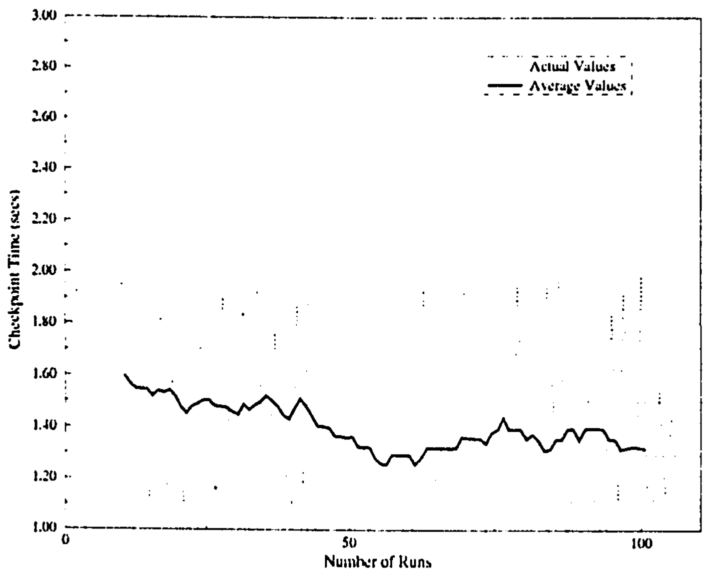

Figure 6.4: An *IO-Intensive Intensive* process with 3 file

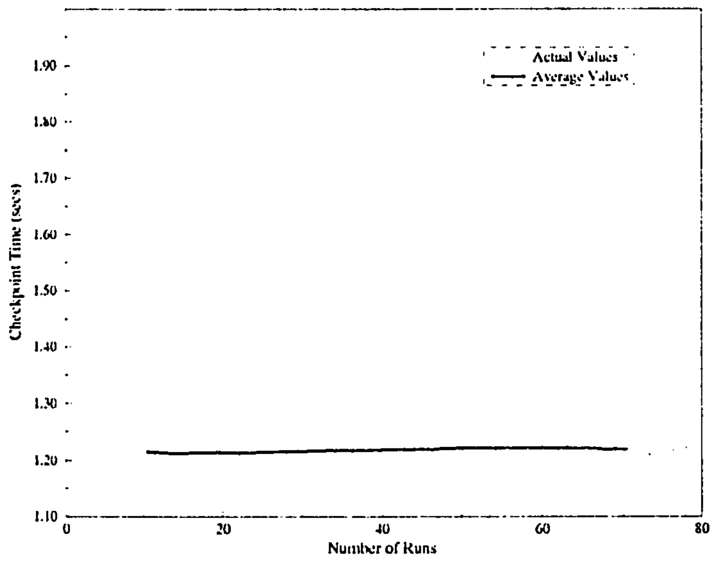

Figure 6.5: A *Communication Intensive* process

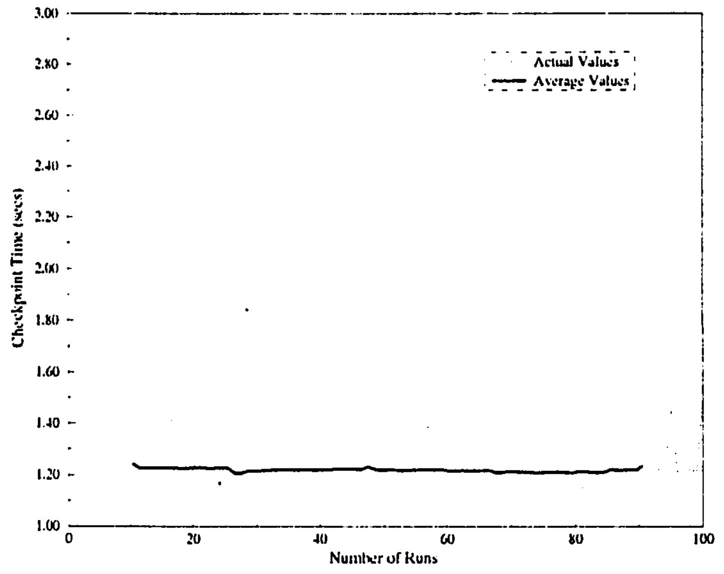

Figure 6.6: A *Hybrid* process

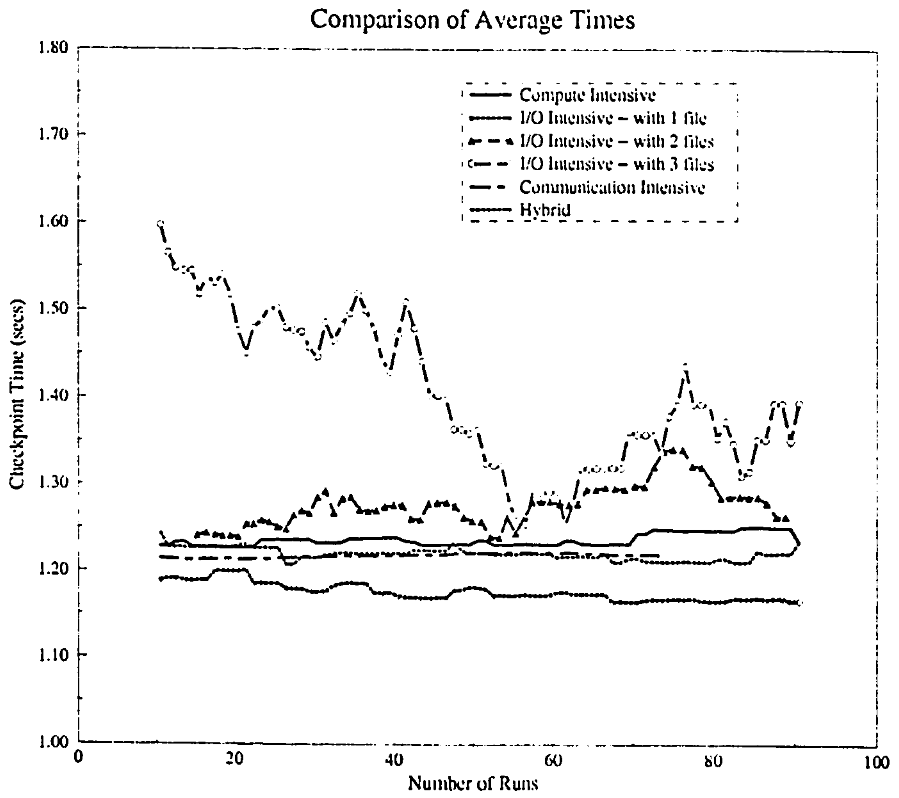

Figure 6.7: Comparison of the average checkpoint times

Application Class	Checkpoint Time (secs)
Compute Intensive	1.24
I/O Intensive - 1 File	1.17
I/O Intensive - 2 File	1.35
I/O Intensive - 3 File	1.41
Communication Intensive	1.21
Hybrid Application	1.22

Table 6.1: The average checkpoint times

The periodic variations observed in almost all of these figures can be attributed to the periodic execution of the "cron" daemon, since these measurements were taken in a realistic environment with multiple processes executing on the system. From Figure 6.2 and 6.3, it is clear that as the number of open files increases, the variations in the checkpoint time increase due to contention for disk access among the various processes executing at that instant. The variations are negligible in case of CPU and Communication intensive applications (Figure 6.1, 6.5 and 6.7).

From the system's perspective the checkpoint time is the interval between the instant the *sysCheckpoint* PMS system call is invoked and the completion of the system call. This time has been found to be a constant 10000 μseconds (or 10 ms)for the representative process. Similarly, the *restoration time* has also been found to be a constant 10000 μseconds. This is in accordance with the logical reasoning that the *checkpoint time* must be the same as the *restoration time*, since whatever has been saved during checkpointing has to re-instated exactly during restoration.

6.5.2 Migration

Let us consider the timing measurements from the user's perspective. Let UT_m denote the average time it takes to migrate a given process, ut_c be the average checkpoint time, and ut_r be the average time taken to restore it. The migration time of a process can then be formulated as:

$$UT_m = ut_c + ut_r + ut_o$$

where ut_o is the overhead involved in interacting with the MDs.

For the representative compute-intensive process, we have a ut_c of 1 second 241038 μseconds and ut_r of 1 second and 242189 μseconds. The migration time T_m has been measured to be 2 seconds 675059 μseconds. Therefore, the overhead t_o in MD-MD interaction that is needed for migrating a process form one host to another is a negligible 191832 μseconds.

Similarly, if ST_m is the system time taken in migration, then we would have

$$ST_m = st_c + st_r$$

where st_c is the system-time in checkpointing the process and st_r the system-time spent in restoring it. There would be no overhead here due to the fact that the MD-MD interaction is handled at the user-level. From our measurements, both st_c and st_r have been found to be a constant 10000 μseconds, thereby giving a ST_m of 20000 μseconds.

The average migration times for processes belonging to the three classes are summarized in Table 6.2 and compared in the graph shown in Figure 6.8. The additional overhead observed in Communication and I/O intensive applications can be attributed to the processing involved in saving/re-instating the datastructures for dealing with the open files and sockets.

Application Class	Migration Time (secs)
Compute Intensive	2.49
I/O Intensive	2.58
Communication Intensive	2.54

Table 6.2: The average migration times

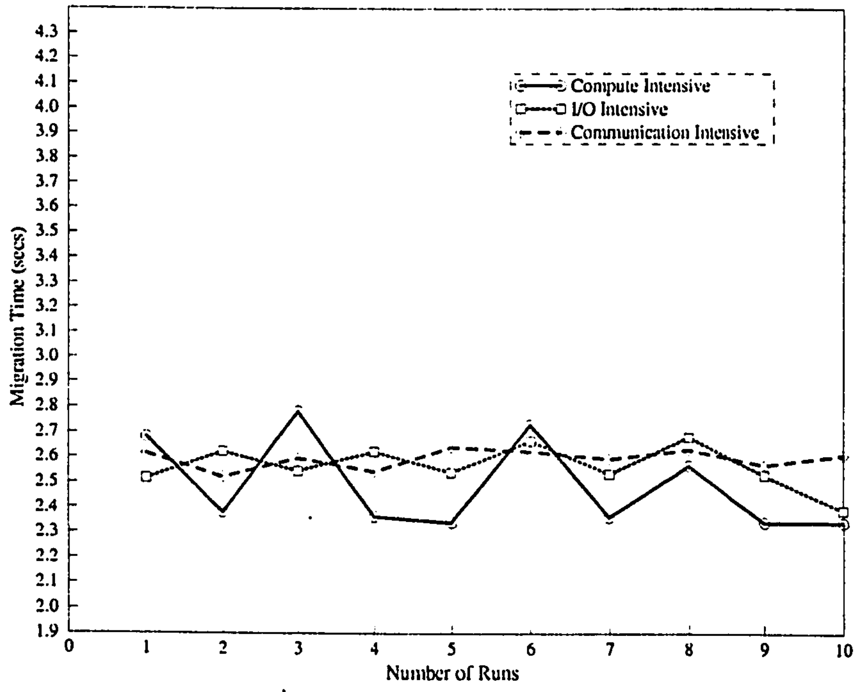

Figure 6.8: Comparison of the average migration times

Chapter 7

Conclusion and Future Work

In this research, the primary objective has been to provide process migration capability in standard Un⋆x systems, by extending the contemporary *process-based* model. Towards this end, we have designed and developed the PMS - Process Migration Subsystem, which allows migration of independent as well as communicating processes executing under the *Linux* OS. The achievements of this work can be summarized as follows:

- The functionality of the operating system has been enhanced with two new system calls to support checkpointing and/or migration of processes.
- The mechanism has been implemented as a stand-alone subsystem to allow it to be portable and easily incorporated into any existing Un⋆x system.
- The mechanism can deal with independent as well as communicating processes. This allows development of 'migratable' distributed applications. The communication among the set of processes belonging to such applications is sustained transparently, even after repeated migration of any of the component

processes.

However, PMS has two important implications which need further exploration:

1. PMS provides an infrastructure for an efficient process/application level fault-tolerance, and hence may be easily incorporated in a system such as NCS (section 3.3).
2. PMS can be incorporated into a dynamic load-balancing framework [18], where the tasks are migrated based on an objective function.

Further, in case of self-checkpointing processes, there may be a need to characterize the overhead imposed by checkpointing, with respect to the parameters such as the number of pages and files involved. Also, for certain class of applications, the overhead imposed by PMS can be characterized to decide whether a process needs to be checkpointed/restored, as opposed to restarting it.

Finally, PMS allows migration only across equivalent/compatible machine architectures and versions of the OS. Although providing migration across heterogeneous platforms comprising of machines of varying architectures remains to be a major problem, migration over different Un*x versions is a challenging but realistic problem that can be pursued within the scope of the further research in this direction.

Appendix A

Installing the Linux OS

A.1 Introduction

Linux is a freely-distributable implementation of UNIX for 80386 and 80486 machines. It supports a wide range of software, including X Windows, Emacs, TCP/IP networking (including SLIP), the works. This document assumes that you have heard of and know about Linux, and just want to sit down and install it.

A.2 Getting Linux

Before you can install Linux, you need to decide on one of the "distributions" of Linux which are available. There is no single, standard release of the Linux software—there are many such releases. Each release has its own documentation and installation instructions.

Linux distributions are available both via anonymous FTP and via mail order on diskette, tape, and CD-ROM. The Linux Distribution HOWTO (see sunsite.unc.edu in the file /pub/Linux/docs/HOWTO/Distribution-HOWTO) includes a list of many Linux distributions available via FTP and mail order.

The release of Linux covered in this appendix is the Slackware distribution, maintained by Patrick J. Volkerding (volkerdi@mhd1.moorhead.msus.edu). It is one of the most popular distributions available; it is very up-to-date and includes a good amount of software including X-Windows, TeX, and others. The Slackware distribution consists of a number of "disk sets", each one containing a particular type of software (for example, the d disk set contains development tools such as the gcc compiler, and so forth). You can elect to install whatever disk sets you like, and can easily install new ones later. Slackware is also easy to install; it is very self-explanatory.

The version of Slackware described here is 2.0.0, of 25 June 1994. Installation of later versions of Slackware should be very similar to the information given here.

Information on other releases can be found in the Linux Installation and Getting Started manual from the LDP. You can also find other releases of Linux on various FTP sites, including sunsite.unc.edu:/pub/Linux/distributions.

The instructions here should be general enough to be applicable to releases other than Slackware.

A.3 Hardware Requirements

The actual hardware requirements for the system change periodically, Linux is evolving. However, at the very least, a hardware configuration that looks like the following is required:

- Any ISA, EISA or VESA Local Bus 80386, 80486, or Pentium system will do. Currently, the MicroChannel (MCA) architecture (found on IBM PS/2 machines) is not supported. Many PCI bus systems are supported (see the Linux PCI HOWTO for details). Any CPU from the 386SX to the Pentium will work. You do not need a math coprocessor, although it is nice to have one.

- You need at least 4 megabytes of memory in your machine. Technically, Linux will run with only 2 megs, but most installations and software require 4. The more memory you have, the happier you'll be. I suggest 8 or 16 megabytes if you're planning to use X-Windows.

- Of course, you'll need a hard drive and an AT-standard drive controller. All MFM, RLL, and IDE drives and controllers should work. Many SCSI drives and adaptors are supported as well.

  Linux can actually run on a single 5.25" HD floppy, but that's only useful for installation and maintenance.

- Free space on your hard drive is needed as well. The amount of space needed depends on how much software you plan to install. Most installations require somewhere in the ballpark of 40 to 80 megs. This includes space for the software, swap space (used as virtual RAM on your machine), and free space for users, and so on.

It's conceivable that you could run a minimal Linux system in 10 megs or less, and it's conceivable that you could use well over 100 megs or more for all of your Linux software. The amount varies greatly depending on the amount of software you install and how much space you require. More about this later.

- You also need a Hercules, CGA, EGA, VGA, or Super VGA video card and monitor. In general, if your video card and monitor work under MS-DOS then it should work under Linux. However, if you wish to run X Windows, there are other restrictions on the supported video hardware.

Linux will co-exist with other operating systems, such as MS-DOS, Microsoft Windows, or OS/2, on your hard drive. (In fact you can even access MS-DOS files and run some MS-DOS programs from Linux.) In other words, when partitioning your drive for Linux, MS-DOS or OS/2 live on their own partitions, and Linux exists on its own. You do NOT need to be running MS-DOS, OS/2, or any other operating system to use Linux. Linux is a completely different, stand-alone operating system and does not rely on other OS's for installation and use.

In all, the minimal setup for Linux is not much more than is required for most MS-DOS or MS Windows systems sold today. If you have a 386 or 486 with at least 4 megs of RAM, then you'll be happy running Linux. Linux does not require huge amounts of diskspace, memory, or processor speed. The more you want to do, the more memory (and faster processor) you'll need. In my experience a 486 with 16 megabytes of RAM running Linux outdoes several models of workstation.

A.4 Installing the OS

Linux is a Unix clone for the 386/486 based PCs, with the features of a full-fledged OS such as true multitasking, virtual memory, shared libraries etc.
In order to facilitate installation, linux provides a **setup** package through which one can select the packages that are required to be installed.

A.5 Configuring the kernel

In order to access the resources, the kernel needs to know the characteristics of the devices that are available. The process of Configuring the kernel is to let the kernel know what devices are connected to the system so that it can load the appropriate device drivers to handle them. Having unnecessary device drivers will make the kernel bigger, and can under some circumstances lead to problems (probing for a nonexistent controller card).

In order to Configure the kernel, do a **"make config"** from the */usr/src/linux* directory. This queries the installer about the various devices that are accessible to the system and stores the responses in the *config.in* file. Once the system configuration has been specified. do a **"make dep"** to set up all the dependencies correctly. This basically uses the information from the *config.in* file to set the **#ifdef** parameters of the appropriate device header files and also creates a file *.depend* which specifies the dependencies among the source-code files. Finally, do a **"make clean"** to remove the intermediate files used during the process of configuration.

A.6 Compiling the kernel

Customizing the kernel to your specific system configuration requires recompilation of the kernel code. This is done by a **"make zImage"** from */usr/src/linux* to create a compressed kernel image. In order to boot your new kernel, you'll need to copy the kernel image (found in */usr/src/linux/zImage* after compilation) to the place where your regular bootable kernel is found. For some, this is on a floppy disk, in which case you can **"cp /usr/src/linux/zImage /dev/fd0"** to make a bootable floppy. If you boot Linux from the hard disk, LILO uses the kernel image that is specified in the file */etc/lilo/config*. The kernel image file is usually /vmlinuz (or /zImage, or /etc/zImage). To use the new kernel, copy the new kernel image over the old one ("cp /usr/src/linux/zImage /vmlinuz"). A better option is to make a copy of it in / ("cp /usr/src/linux/zImage /") and edit */etc/lilo/config* to specify an entry for both the new kernel image as well as the old (in case the new one does not work)as shown below :

```
Image = /zImage
 root = /dev/hda3
 label = linux

 Image = /vmlinuz
 root = /dev/hda3
 label = oldlin
```

Finally, we need to have LILO update its loading map, so as to be able to boot the new kernel image. This is done by running **"/sbin/liloconfig"**. Now when the system is rebooted. it will be able to use the new kernel image. If you need to change the default root device, video mode, ramdisk size, etc. in the kernel image, use the 'rdev' program (or alternatively the LILO boot options when appropriate). No need to recompile the kernel to change these parameters.

A.7 Configuring the X-Windows system

In order to be able to use the GUI features of the X-Windows system, we need to specify the characteristics of the Terminal, Keyboard, Mouse etc. The Linux setup package provides a set of *Xconfig* files (in */var/X11/lib/X11/Sample.XConfig-files/*) corresponding to various commonly used I/O devices. We just need to copy the appropriate file to */var/X11/lib/X11/Xconfig* (and edit some of its parameters, if needed), which is the default file used by the **"startx"** command for invoking the X-Window system.

A.8 Possible problems

The most common problem encountered during installing Linux, along with other bootable operating systems (such as DOS, OS/2, etc.) is that the system hangs upon reboot (with the partial message **LI** on the screen). The reason for this is that although we had partitioned the disk and allocated space for the other OS (using *fdisk*), we had not installed them in their respective partitions. So, when Linux gets loaded, it tries to find these other OS in their partitions but couldn't, and hangs.

To overcome this problem, we can do one of the following :

- Install the OS immediately after setting up the partitions.
- Install the OS upon completion of the *setup*, but before reboot.
- Reboot the system with a Linux boot floppy and install the OS.
- Make all other OS partitions as in-active by setting off the BOOT flag through *fdisk*.

Another problem occurs during X-Window configuration. When the *startx* command is issued, the screen might clear and start wavering. This is because the Kernel tries to start X-Window system with the highest resolution specified in the Xconfig file. To set this right, press *Ctrl–* (decreases the screen resolution, *Ctrl+* is used to increase the resolution).

Bibliography

[1] Yeshayahu Artsy and Raphael Finkel, *"Designing a Process Migration Facility: The Charlotte Experience,"* IEEE Computer, pages 47–56, September 1989.

[2] Maurice J. Bach, *The Design of the Unix Operating System,* Prentice Hall International Editions, September 1989.

[3] Henry Clark and Bruce McMillin, *"DAWGS - A Distributed Compute Server Utilizing Idle Workstations,"* IEEE, pages 732–741, 1990.

[4] Intel Corporation, *i486 Microprocessor Programmer's Manual,* Osborne McGraw-Hill, 1990.

[5] George F. Coulouris and Jean Dollimore, *Distributed Systems: Concepts and Design,* Addison-Wesley, 1991.

[6] F. Douglis and J. Ousterhout, *"Transparent Process Migration: Design Alternatives and the Sprite Implementation,"* Software - Practice and Experience, 21(8):757–785, August 1991.

[7] Nick Doulas and Balkrishna Ramkumar, *"Efficient Task Migration for Message Driven Parallel Execution on Nonshared Memory Architectures,"* International Conference on Parallel Processing, pages II–170 to II–173, 1994.

[8] M. Bozyigit K. M. Al-Tawil and S. K. Naseer, *"Tolerating Node Failures on a Network of Workstations using Process Migration,"* FTCS, The 12th Annual International Symposium on Fault-Tolerant Computing, 1996, In submission.

[9] Kai Li, Jeffrey F. Naughton, and James S. Plank, *"Low-Latency, Concurrent Checkpointing for Parallel Programs,"* IEEE Transactions on Parallel and Distributed Systems, 5(8):874–879, August 1994.

[10] Kai Li, J.F. Naughton, and J.S. Plank, *"An Effecient Checkpointing Method for Multicomputers with Wormhole Routing,"* International Journal of Parallel Programming, 20(5):159–180, June 1991.

[11] Michael J. Litzkow, *"Remote Unix: Turning Idle Workstations into Cycle Servers,"* Proceedings of the USENIX 1987 Summer Conference, pages 381–384, 1987.

[12] Michael J. Litzkow, Miron Livny, and Matt W. Mutka, *"Condor - A Hunter of Idle Workstations,"* Proceedings of the 1988 Conference on Distributed Computing Systems, pages 104–111, 1988.

[13] M.Theimer, K.Lantz, and D.Cheriton, *"Preemptible Remote Execution Facilities for the V-System."* ACM Operating Systems Review, 19(5):2–12, December 1985.

[14] David A. Nichols, *"Using Idle Workstations in a Shared Computing Environment"* ACM Operating Systems Review, pages 5–12, November 1987.

[15] David Powell, *"Distributed Fault Tolerance: Lessons from Delta-4,"* IEEE Micro, pages 36–47, February 1994.

[16] Michael L. Powell and Barton P. Miller, *"Process Migration in DEMOS/MP,"* ACM, pages 110–119, 1983.

[17] Injong Rhee, *"Optimal Fault-Tolerant Resource Allocation in Dynamic Distributed Systems,"* Proceedings of the Seventh IEEE Symposium on Parallel and Distributed Processing, pages 460–467, San Antonio, Texas, October 25-28 1995.

[18] S.Ghanta S. Nisar Ul Haq, M. Bozyigit and S.K. Naseer, *"Design of a Load Balancing Framework for Distributed and Parallel Applications,"* International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '95), pages 979–993, Athens, Georgia, November 3-4 1995.

[19] S. Srinivasan and N.K. Jha, *"Task Allocation for Safety and Reliability in Distributed Systems"* 1995 International Conference on Parallel Processing, pages II–206 to II–213, 1995.

[20] W. Richard Stevens. *Unix Network Programming*, Prentice Hall International Editions, 1990.

[21] Tony T.Y. Suen and Johnny S.K. Wong, *"Efficient Task Migration Algorithm for Distributed Systems,"* IEEE Transactions on Parallel and Distributed Systems, 3(4):488–499, July 1992.

[22] Andrew S. Tanenbaum. *Modern Operating Systems*, Prentice Hall, 1992.

[23] Andrew S. Tanenbaum. *Distributed Operating Systems*, Prentice Hall International Editions, 1995.

[24] Marvin M. Theimer and Lantz Keith A, *"Finding Idle Machines in a Workstation-Based Distributed System,"* IEEE Transactions on Software Engineering, 15(11):1444–1457, November 1989.

[25] S. Tridandapani and Arun K. Somani, *"Efficient Utilization of Spare Capacity for Fault Detection and Location in Multiprocessor Systems,"* FTCS, The Twenty-Second Annual International Symposium on Fault-Tolerant Computing, pages 440–447, Boston, Massachusetts, 1992.

[26] Cui-Qing Yang and Yaoshuang Qu, *"Fault-Tolerance in the Execution of Remote Jobs on Idling Workstations,"* Concurrency: Practice and Experience, 7(1):43–60, February 1995.