IMPLEMENTATION OF A CHECKPOINT-RESTART SYSTEM

by

NAVEEN LANKE

B. Tech., Srikrishnadevaraya University,
Anantapur, India, 2004

A REPORT

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2006

Approved by:

Major Professor
Daniel Andresen, Ph.D.

ACKNOWLEDGEMENTS

I would like to express my thanks and gratitude to Dr Daniel Andresen, my major professor for giving me constant encouragement, advice throughout this whole project as well as during my Masters in Computer Science at Kansas State University. I also thank him for patiently clearing my questions and giving me good leads during this project work.

I would like to thank Dr William Hankley and Dr Mitchell Neilsen for serving in my committee as well as for their valuable cooperation during this project.

I would like to thank Mr Prashanth Palakollu for his help in setting up, configuring and installing Fedora Linux on VM-Ware Player.

ABSTRACT

The main objective of the project is to checkpoint the current running process, save the image and restart the process from where it is check pointed. A process may be migrated either before it starts its execution (*non-preemptive migration)* or during the course of its execution (*preemptive process migration).* Preemptive process migration is costlier than non-preemptive process migration since the process environment must also accompany the process to its new node for an already executing process. In the project, we deal with preemptive process migration and here after the term process migration implicitly means preemptive process migration.

Checkpointing a given process is nothing but saving its state. The state usually includes register set, address space, allocated resources, and other related process private data. Restart mechanism resumes the process execution by restoring the checkpointed state of the process, on the destination machine.

Checkpointing can be implemented at three levels namely *kernel-level*, *user-level* and *application-level*. These levels differ in implementation, complexity, performance, transparency and reusability.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

1.1 Process

A process is basically a program in execution. It consists of the executable program (code), the data and stack, program counter, stack pointer and other registers and all the information needed to run the program.

Typically in any operating system, all the information about each process is stored in an OS table called the process table which is an array of structures, one for each process currently in existence and each process is uniquely identified with its Process Identifier (PID). In modern OS a process runs in its own address space and is separate from other processes. A process interacts with the operating system, or the kernel, by system calls; it can also communicate with other processes by inter-process communication mechanisms provided by the operating system.

1.2 Motivation for Process Migration

In days gone by, computers were prohibitively expensive. Most organizations had only a small number of computers, and each computer operated autonomously. However, as the cost of computers dramatically fell, it became reasonable to conceive the systems whose overall computational power could be increased by using more than one processor to simultaneously do the work on the same problem.

A common paradigm is explicit parallel programming: a programmer defines multiple concurrent tasks, encapsulating each in an OS process, and each process is run on a different processor. Most parallel programming environments decide where each process will be run at the time the process is born. Once a process begins, it must remain resident on the same processor until the completion of the task. Even if the scheduling of processes to processors turns out to be suboptimal, it cannot be changed. Often it is impossible to predict how the load on the system will change over time, and thus often impossible to optimally schedule processes to processors.

In an attempt to gain better performance, a number of researchers have developed a different class of parallel execution environments that allow processes to move from processor to processor dynamically at any point in the life of the process. A change in processor residency in

the middle of the lifetime of the process is typically called "**PROCESS MIGRATION**". By dynamically moving processes throughout their lifetimes, the system can potentially adapt better to changes in load that could not be foreseen at the start of the tasks. Proponents of process migration claim that this dynamic adaptation leads to a better system-wide utilization of available resources than static process scheduling.

With a pool of processing nodes dedicated to servicing the user load, an efficient process migration scheme can balance the user load effectively. Also, it is much more scalable. If the control of the pool is appropriately designed (i.e., distributed), as many nodes as desired can be added, incrementally, as the expected nominal load on the system increases. These are the goals that motivated the development of process migration.

1.3 Overview of process Migration

Process migration is the relocation of a process from its current location (*current node*) to another location (*destination node*). The flow of execution of a migrating process is illustrated in the figure below

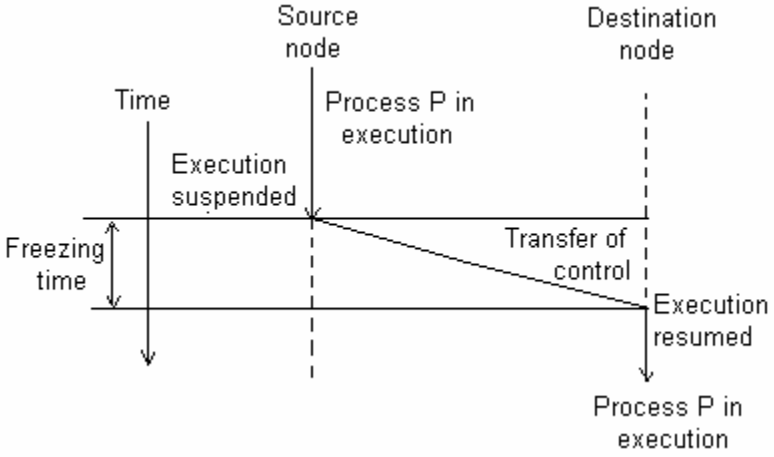


Figure 1 – Flow of execution of migrating process

A process may be migrated either before it starts its execution (*non-preemptive migration)* or during the course of its execution (*preemptive process migration).* Preemptive process migration is costlier than non-preemptive process migration since the process environment must also accompany the process to its new node for an already executing process. In this project, we deal with preemptive process migration and here after the term process migration implicitly means preemptive process migration.

The major sub activities involved in process migration are

1. *Checkpointing* (suspending and saving the state) the process on the source node.
2. Transferring the process state to the destination node.
3. *Restarting* (resuming the execution) the process on the destination node based on the saved state, from exactly the point of suspension.

1.4 Introduction to Checkpoint and Restart

Checkpointing a given process is nothing but saving its state. The state usually includes register set, address space, allocated resources, and other related process private data. Restart mechanism resumes the process execution by restoring the checkpointed state of the process, on the destination machine.

Ideally the checkpointing system should be able to save and restore the following:

- State of memory and CPU registers.
- Child processes of a checkpointed process.
- The status of open file descriptors held by the checkpointed process(es).
- The status of files `mmap( )`ed by the checkpointed process(es).

1.4.1 Approaches to Checkpointing

Checkpointing can be implemented at three levels namely *kernel-level*, *user-level* and *application-level*. These levels differ in implementation, complexity, performance, transparency and reusability.

Kernel-level Checkpointing

In kernel-level checkpointing, the operating system supports checkpointing and restarting processes. The checkpointing is transparent to applications; they do not have to be modified or linked with any special library to support checkpointing.

User-level Checkpointing

Application programs to be checkpointed are linked with a checkpoint user library. Upon checkpointing, a checkpoint-triggering signal is sent to the process. The functions in the checkpoint library respond to the signal and save the information necessary to restart the process. On restart, the functions in the checkpoint library restore the execution environment for the process. To applications, checkpointing is transparent. But unlike kernel-level support, applications must be relinked to allow checkpointing.

Application-level Checkpointing

Applications can be coded in a way to checkpoint themselves either periodically or in response to signals sent by other processes. When restarted, the application must look for the checkpoint files and restore its state.

2 Problem Definition

2.1 Problem Statement

To design and implement a transparent checkpoint /restart package for Linux Operating system without entailing any changes to the kernel.

To design and develop a software system that migrate processes in a Linux network using the checkpoint /restart package.

2.2 Problem Description

Process migration is the relocation of a process from its current location (*current node*) to another node (*the destination node*). To migrate a process, we need to checkpoint its state, transfer the state to the destination node and restart the process at the node.

The basic part is the checkpoint /restart system, which deals only with the mechanism of checkpointing a process, and restarting the one checkpointed earlier. It does of course provide a clean interface, but it is the duty of the upper layers to define their own policies. A wide range of policies might be needed, 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 of similar length), or application concurrency (mapping application processes to machines in order to achieve high parallelism).

2.3 Objectives of the project

Resource utilization is a problem that is present not only in distributed systems but also in networked systems like a LAN of computers. In distributed systems efficient resource utilization is achieved by distributing the load using process migration. In the context of networked systems, process migration can lead to efficient overall system utilization by making use of the idle workstations. The advantage of efficient system utilization at the coarse granularity level (the process being the grain size), primarily motivated for undertaking this project.

3 Requirements Specification

3.1 Hardware and Software

- Pentium processor
- Ethernet Card
- Same Version of Linux Operating System
- ‘C’ Compiler

3.2 Overview of the platform

As the source code for Linux is open, I have chosen Linux Operating System because the project needs kernel support. Most of the programming in the project is done in ‘C’ language. For compiling these modules Linux supports a standard ‘C’ compiler called “GCC”.

Constraints

As the software is tightly coupled with the version of the operating system it only works with the version that it is built with. With the advent of newer versions we need to do little or a little modification to the source. Also all the systems in the LAN should be running the same version of the Linux OS.

4 System Analysis

4.1 System Requirement Analysis

System analysis is the process of gathering and interpreting facts, diagnosing problems and using the information to recommend improvements to the system. Basically 'analysis' specifies what the system should do. It does not state how the requirements are accomplished or how the system is implemented.

The system should meet the following requirements

- Transparency : Transparency means whether user applications need to be modified, recompiled or relinked, and whether at run time they know that they are being checkpointed and restarted.
- Security : The security provided by the operating system should not be violated. i.e. control should be exercised over the set of processes that can be migrated by a particular user.

4.2 Structured Analysis

The process migration system is intended to migrate the given process to the given destination node. The major sub activities involved in process migration are

1. *Checkpointing* (suspending and saving the state) the migrating process on the source node.
2. Transferring the process state to the destination node.
3. *Restarting* (resuming the execution) the process on the destination node based on the saved state, from exactly the point of suspension.

4.2.1. Data modeling

The primary data objects and their attributes are given below

Segments

This object contains attributes of each segment in the address space of the process. Its attributes are

segment_start	segment_end	protectionFlags	isShared

Table 1 - Segments

Memory

This object contains information regarding the entire structure of the process' address space.

start_code end_code	start_data end_data	start_brk	brk
start_stack	start_mmap	arg_start arg_end	env_start env_end

Table 2 – Memory

Header

This object contains information regarding the checkpointed process and identifies a valid file having a checkpointed image.

signature	num_segments	processIdenfication	exec_file_name	in_sys_call

Table 3 – Header

4.2.2. Functional modeling and Information flow

At the highest level the software can be viewed as an application that takes as input the process id and the remote host id and resumes the execution at the remote host.

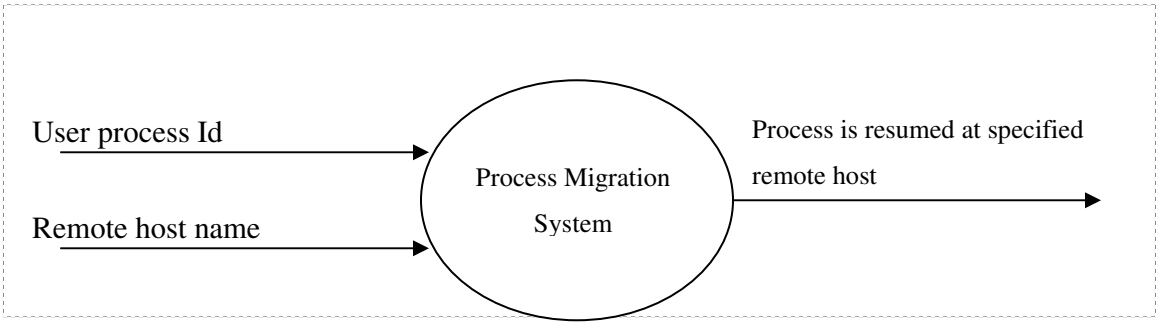


Figure 2 – Context level DFD of the Process Migration system

This is further divided into "migrate" and "restartd" subsystems. The "migrate" system consists of two subsystems, the "checkpoint" for saving the process state and "transfer" for transferring the saved image to the remote host.

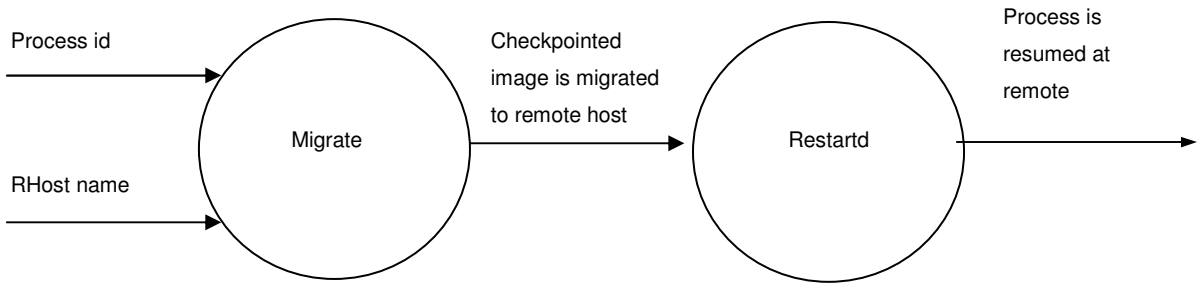


Figure 3 – Level 1 DFD of Process Migration system

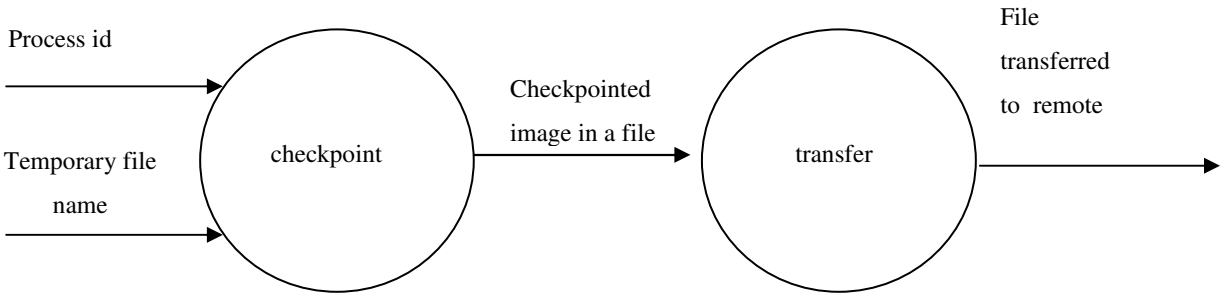


Figure 4 – Migrate system

The "restartd" daemon is subdivided into the receive daemon which receives the saved image of a process and the "rstrt" which restart based on this image.

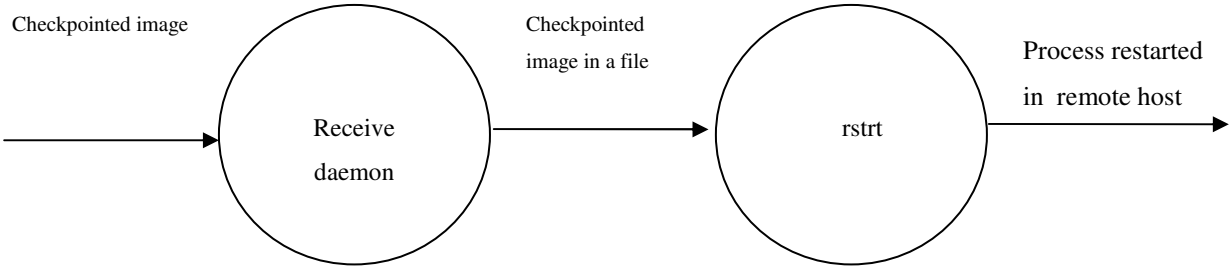


Figure 5 – Restart system

4.2.3. Behavioral modeling

The state transition diagram for the process migration system is given below

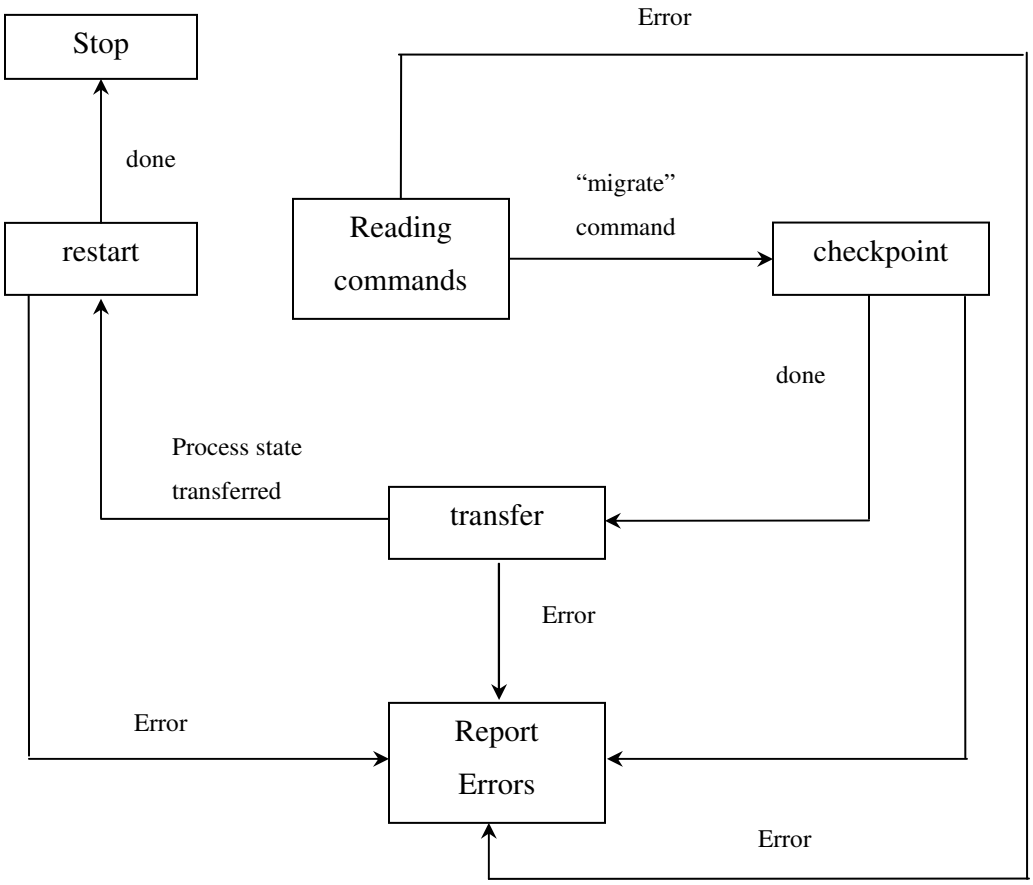


Figure 6 – State transition diagram

The process starts when the command is given at the terminal. When the checkpoint signal is given the process is checkpointed and the image file is created. The image file is transferred to another machine and it is restarted. At any time during the course of this process, if any errors arise, appropriate error messages are displayed to the user.

5 System Design

The design is not limited to Linux, but I have chosen Linux as a platform for the implementation.

The design of process migration involves the design of the following

- Checkpointing the process to be migrated (checkpoint sub-system)
- Transferring image of the process to the remote node (migrate sub-system)
- Resuming the process execution at the remote node using the checkpointed image (restart sub-system)

Since checkpoint and restart are inter related, they are treated as a single unit called the Checkpoint /Restart System (CRS).

5.1 Design Goals

In this project, the goal is to design and implement an application for process migration using a transparent checkpoint /restart system.

- **Generic**

  The design to be implemented easily on any general-purpose operating system.

- **Transparency to User Applications**

  Transparency means whether user applications need to be modified, recompiled or relinked, and whether at run time they know that they are being checkpointed and restarted. Generally speaking, adding support into the kernel leads to better transparency and performance, but more implementation complexity and less portability. I want the package to be general-purpose and able to checkpoint existing applications transparently

- **No Kernel Patches**

  All the current checkpoint /restart packages have one common drawback: they require modification of existing code (either kernel or user applications), thus are difficult to deploy. Moreover, not all operating systems have source code available, so it's not always possible to patch the kernel. So, I want to implement the system with out patching the existing kernel.

- **Do As Much As Possible in User Space**

  Kernel programming is hard and error-prone. If any functionality can be done in user space, it is better not to do it in the kernel.

5.2 Data Design

The checkpointed image of a process is saved in the following format

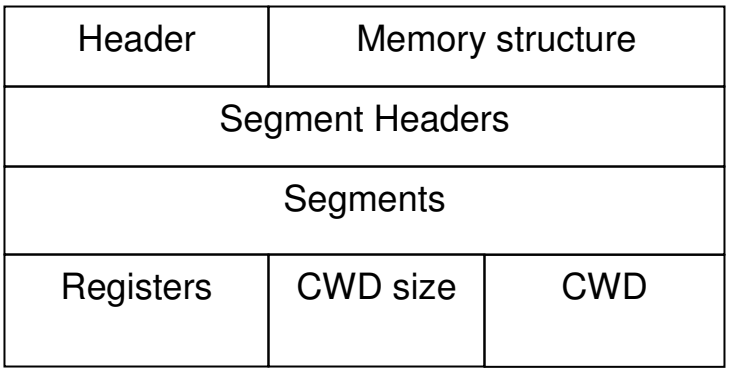


Figure 7 – Checkpoint file format

Header:

The header contains the information regarding the check pointed process viz. process identification, process group, name of the program executed by the process etc. The header starts with a signature "CRF" to signify that the file format is checkpoint /Restart File format.

Memory structure:

It contains information regarding the structure of the entire address space of the checkpointed process. For example the start and end addresses of code, data and stack segments are stored.

Segment headers:

The segment headers contain information about each segment i.e, start and end addresses of the segment, its protection flags, etc.

Segments:

The contents of each segment of the process' address space are stored.

Registers:

The contents of the registers at the time the process is check pointed are stored so that they can be restored when the process is restarted.

Size of current working directory:

Working directory of the check pointed process need to be saved so that the process can be restarted in the same working directory. Since path of the current working directory is a variable, its size is stored.

Current working directory:

The full path of the current working directory of the process is stored.

5.3 Architecture Design

In order to achieve the steps in process migration, with out violating the aforesaid design goals, a layered architecture has been employed. Normally, user processes communicate with the OS using the standard System Call Interface. Since transparency has to be achieved, CRS has been designed as an application program that runs in the user space just as any other user process. Given process identification, the corresponding process should be check pointed by the application. To do so, it makes use of the library developed here. The advantage of providing such a library interface is that any application such as a Periodic Check pointing application can make use of it as shown in the figure. Process Migration application makes use of the CRS for Check pointing and resuming the process to be migrated.

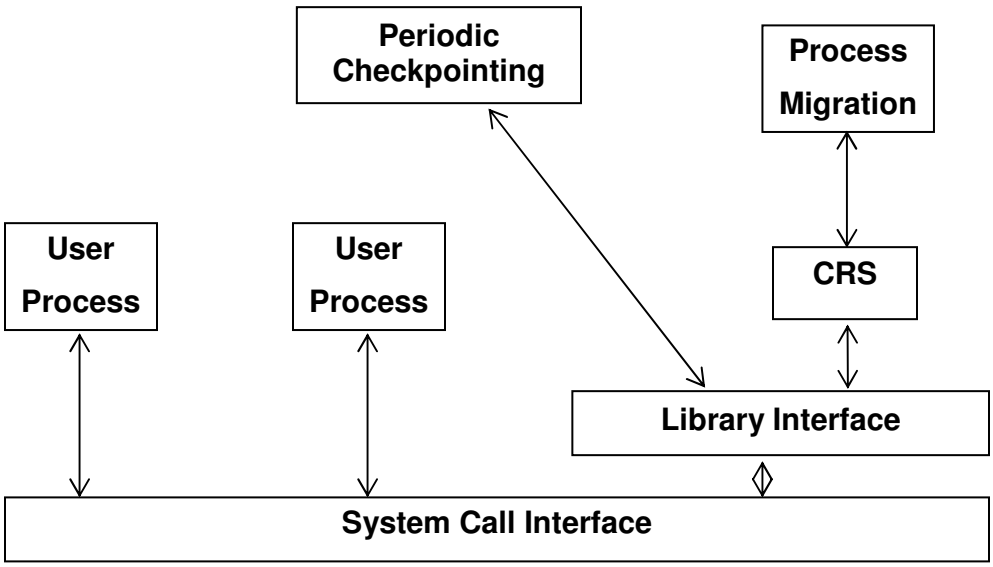


Figure 8 – Layered Architecture of the Process Migration application

The library has been implemented in such a way that, with out modifying the kernel, the desired functionality is achieved.

In order to migrate a process to a destination node, a client – server model has been used as shown below

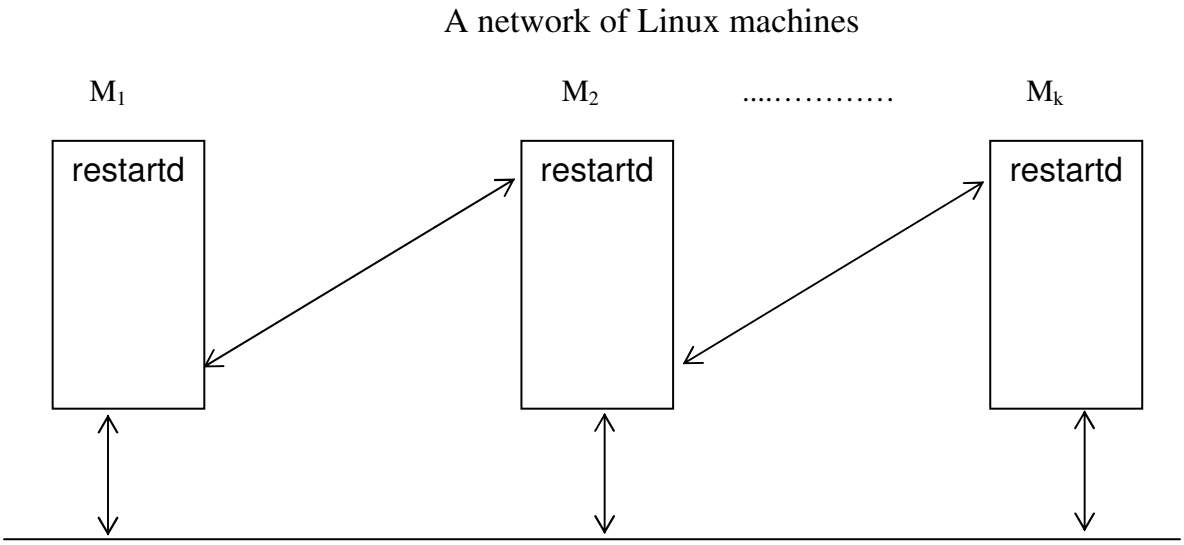


Figure 9 – The client-Server model

At each node, there exist two applications: a "migrate" client and a "restart" daemon. To migrate a process, the "migrate" application checkpoints the process and transfers its state to the destination node where, the "restart" daemon receives the checkpointed state and restarts the process.

5.4 User Interface Design

The interface of various subsystems of our software are given below

CRS:

- ck - [options] <pid> <file>
- resume <file>

Process Migration:

- migrate - [options] <file> <destination>

5.5 Process Design

The hierarchy chart of the system is shown below

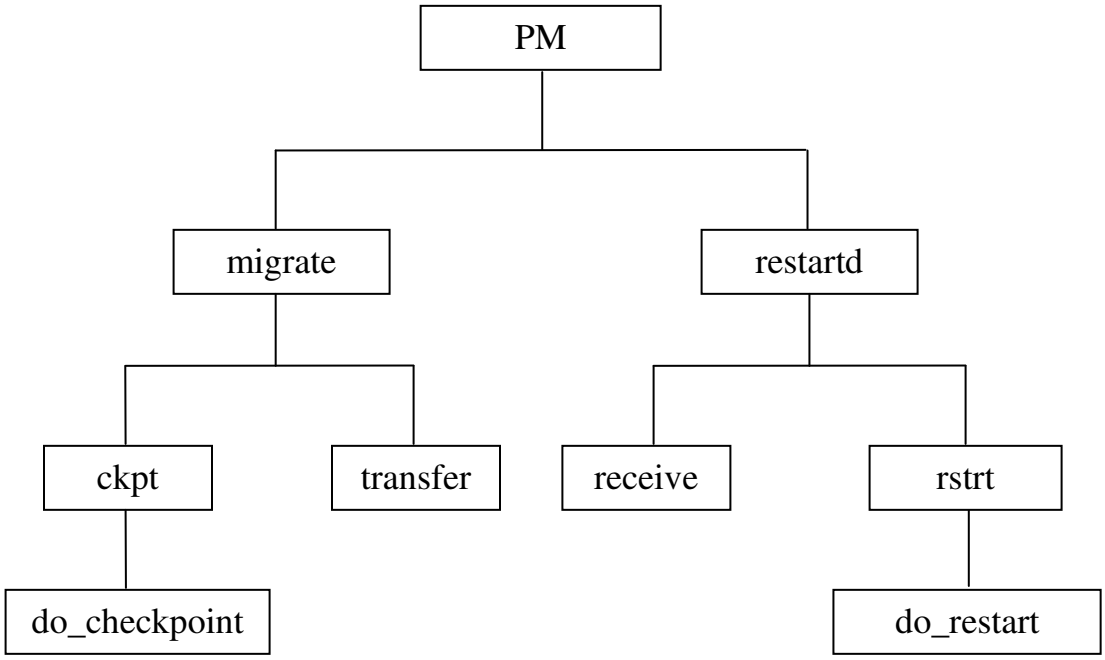


Figure 10 – Hierarchy Chart

The pseudo code for various subsystems of the software is given below

Checkpoint

1. select the process to be checkpointed
2. if the process is not a checkpointable process then go to step 6
3. freeze the execution of the process
4. collect and save all the data regarding current state of the process
5. resume or stop the execution of the freezed process depending on the option given
6. stop

restart

1. if the file containing the checkpointed image is not a valid checkpoint file, go to step 6
2. create a new process
3. change the working directory of the created process to the one that is saved in the checkpointed image
4. read the state of the checkpointed process from the file and restore it in the new process
5. change the state of the process as “runnable”
6. stop

migrate

1. read the identification of the process to be migrated and the destination node
2. checkpoint the process state
3. transfer the checkpointed state to the destination node

restartd

1. receive the checkpointed image from the remote client
2. restart the process at the destination node using the checkpointed image

6 Implementation

6.1 Overview

As already said the checkpointing can be done at three levels, and here it is done in kernel level. To checkpoint any arbitrary process, the checkpointing process must be in the kernel mode. And for this it has to make a system call. But as of now the Linux OS do not provide system calls for checkpointing a process. Adding a system call to the OS is not that simple and also requires change of source code, which violates our primary design goal. But most operating systems support dynamically loadable kernel modules. Kernel modules are typically loaded at boot time or on demand when the kernel requires certain functionality. They provide some entry points similar to system calls with which we can enter into kernel mode. Once the module is loaded, it becomes part of the kernel address space and can access everything in the kernel. By implementing part of the CRS as a kernel module, I have achieved virtually the same level of transparency as kernel patches but avoided changing the kernel. This makes it much easier to use. The process migration application makes use of the checkpoint and restart facilities provided by the CRS.

6.2 Realization of CRS

The CRS can checkpoint and restart any arbitrary process with out linking with any other library as it is the case with many other counterparts. Hence transparency is achieved.

CRS is realized as follows

1) Register a pseudo character device called /dev/crd (checkpoint restart device)
2) Write a device driver for this device with intended functionality, i.e basically the checkpoint (do_checkpoint) and restart (do_restart) procedures are implemented as part of the device driver.
3) To provide an abstraction a library is built which hides the details of the device and its entry points (usually the IOCTL entry points used for any file).
4) The checkpoint (ckpt) and restart (rstrt) programs are implemented using this library interface with out having to worry about the device, its entry points, parameters etc.

The above concept is illustrated as shown in the figure below,

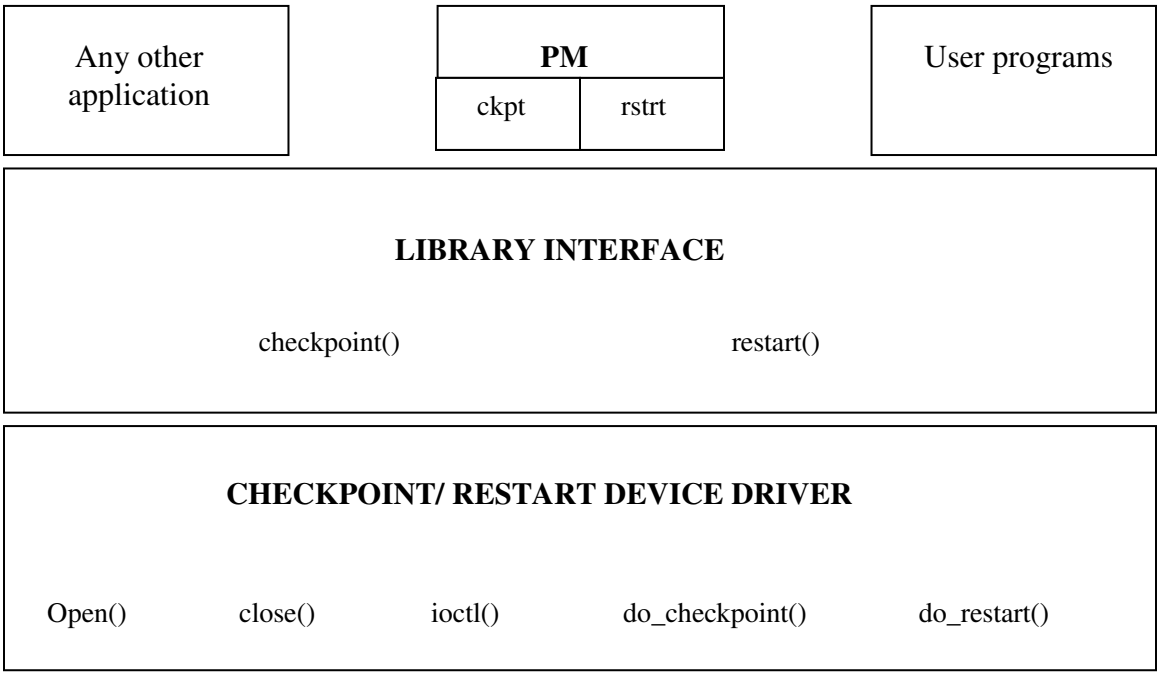


Figure 11 – Application

6.3 The Library Interface

The device file is encapsulated in a helper library and the following programming interface is provided:

int checkpoint (int fd, int pid, int flags)

Three parameters are passed: descriptor of the file the checkpoint image should be written to, pid of the target process and the flags.

Flags can be OR-ed by the following:

- CRS_KILL: If set, the target process is killed immediately, or it would continue to run.
- CRS_NO_BINARY_FILE: If set, code sections of the binary don't get dumped.
- CRS_NO_SHARED_LIBRARIES: If set, shared libraries don't get dumped.

These flags are taken from Epckpt. I have also learnt another thing from Epckpt that a file descriptor be passed instead of a path name to the checkpoint routine. The kernel then writes checkpoint image to this descriptor. In this way the image is not necessarily saved to local disk. Instead, it can be sent directly to a remote machine via a socket. This significantly reduces the checkpoint overhead for process migration. Also, I have used a function called "pack_write" of Epckpt, to write the checkpointed image to a file, one page at a time.

int restart (const char * filename, int pid, int flags)

Three parameters are passed: file name of the checkpointed image, a pid and flags. This function loads the image, and then replaces "current" process by the checkpointed process.

The last two parameters are mainly for notification after the process is resumed:

- pid: If the RESTART_NOTIFY flag is set, when the restart is finished the kernel will send a SIGUSR1 signal to the process specified by pid. If pid is 0, send it to the parent process.
- flags: Two flags are supported. RESTART_NOTIFY tells the kernel to notify a certain process when the restart is done, and RESTART_STOP tells the kernel to stop the restarted process immediately (usually use RESTART_NOTIFY at the same time so that another process can continue it).

As mentioned before, checkpointed image can be sent to a remote machine, by giving a socket descriptor. The same thing can't be done for restart i.e., receive checkpoint image from remote machine and directly restart it without saving to the disk because, mmap( ) that is used while restoring the process' address space, requires a concrete file.

6.4 Subsystems

6.4.1. Internals of do_checkpoint( )

This function is implemented as a part of the device driver. It also takes care of the security issues i.e., it ensures that the user have required access privileges to the process being checkpointed. The components of a process that are currently checkpointed are:

Address Space

The entire address space of the process, i.e., code, data, stack and the extended segment, is checkpointed. All the addresses are “virtual” and are translated into physical addresses via page directories and page tables. The operating system sets up page directories and page tables, and then address translation is done by hardware automatically.

In Linux, each process has its own page directory and page tables that are initialized in a “fork” and switched to in a context switch. All the processes have an address space from 0 to 4GB, but mapped to different physical memory regions due to their different page structures.

The kernel also has its own page directory and page tables, but it is special because it just maps the physical address to itself. For instance, a pointer of 0x00004000 in a process may actually points to a physical address of 0x01234000, but in the kernel a virtual address is exactly its physical address.

During the checkpoint we need to access the target process' address space from the kernel. Common C functions like strcpy wouldn't work because pointers in kernel and user space have completely different meanings. Instead, the Linux provides a set of helper functions (copy_from_user(), copy_to_user(), etc) that allow data transfer between kernel and the current process.

But now we want to access a process's address space that is not the current one. Suppose we want to access the address *addr* in a process *p*. The physical address of *addr* is a function of both *p* and *addr*. The kernel source provides functions for accessing a process’ address space. These functions take care of the address translation and swapped out pages, i.e., if a swapped out page need to be accessed, these functions swap in the required page and access it. We make use of these functions while checkpointing a process which take *p* and *addr* as parameters and by using *p*'s page directory and page tables, they fetch the required pages from swap to main memory.

Register Set

Register set is easy to do, as long as you know where it is. On Linux, there is a simple connection between the locations of process' task structure and its register set location:

Assuming struct task_struct *p is a pointer to the task structure, the corresponding register set location is:

struct pt_regs *regs = ((struct pt_regs *)(2*PAGE_SIZE + (long)p)) - 1;

CWD

The Current working directory in which the process was running is also saved in the image and while resuming it is restored by calling 'chdir' in the user space.

6.4.2. Internals of do_restart()

This function does the exact reverse of do_checkpoint(). In LINUX, the process that is running is identified by the global variable “current”. When this function is called by a process, then its “address space” is restored with that of the saved one in checkpoint image. Similarly the “register set” is also restored. It is ensured that the current working directory (CWD) is restored in user space before issuing the call to do_restart(). Then the process is signaled to run or stop as per the parameter flags.

6.4.3. Internals of migrate

The migrate program checkpoints the given process and sends the checkpointed process image to the specified remote host by establishing a TCP connection with the Restartd server residing at the remote host.

6.4.4. Internals of Restartd

The restartd is server daemon sleeps at the port 1023 and responds to only network request for that port. After receiving request from the client it accepts the image and saves it to a file and gives this file as input to the rstrt program. The rstrt program (which calls the do_restart( )) is similar to the execve() system call in that, it restores the state of the checkpointed process in to a newly created process and makes it the currently running process. The only difference is that execve() does it from a binary file but rstrt does it from the checkpoint image file.

7 Testing

The software has been constructed and tested in small segments. Unit testing focuses verification effort on the smallest unit of software design - the module. The following modules have been identified and tested

do_checkpoint

do_restart

transfer

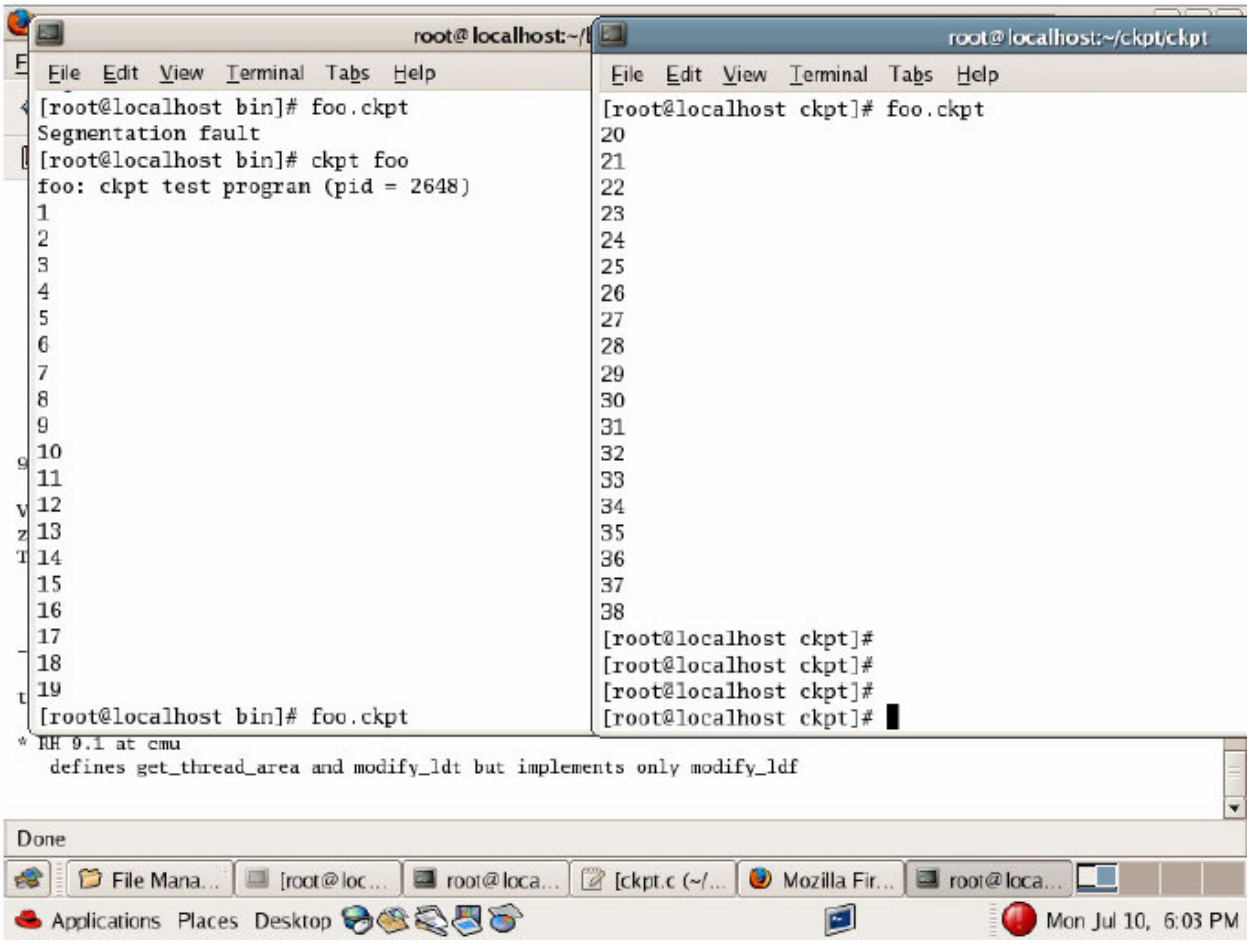
received

Since do_checkpoint and do_restart are like a function and its inverse, they should be tested in conjunction. A process that is restarted from its checkpointed image should run exactly from the point it was checkpointed. This forms the basis for testing the correctness of both do_checkpoint and do_restart. At the lowest level, the information regarding data structures and address space of the process that was restarted using its checkpointed image can be compared with that of the checkpointed process. I have done this comparison using the process specific information that is maintained in the /proc file system in Linux. The very purpose of /proc is this.

The interfaces between various modules have been successfully tested for correctness. CRS has been tested for checkpointing and restarting a process on a workstation. “migrate” and “restartd” have been tested in conjunction for migrating a process from one node to another on a network. The “migrate” and “restartd” are client and server respectively. The client transfers the checkpointed image over a TCP socket, which the server receives, by listening to a standard port (1023). Testing these modules is simple because, it involves ensuring that the sent file is received.

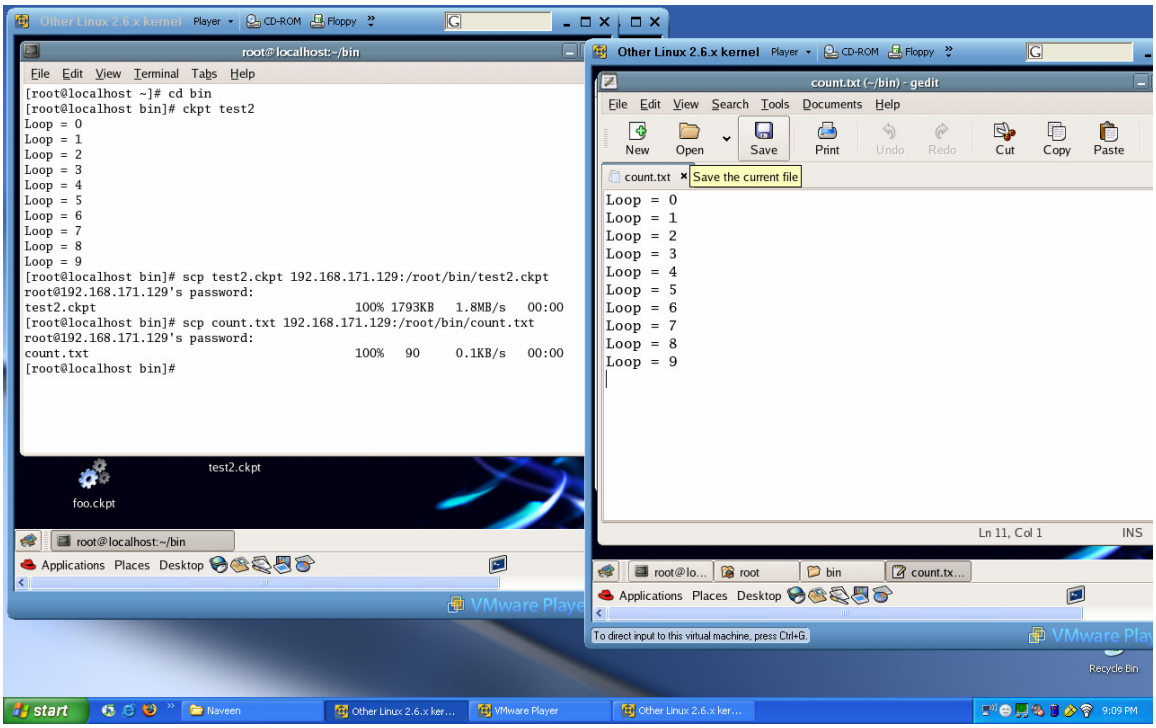
Finally I have verified that all system elements have been properly integrated and perform allocated functions.

8 Screenshots

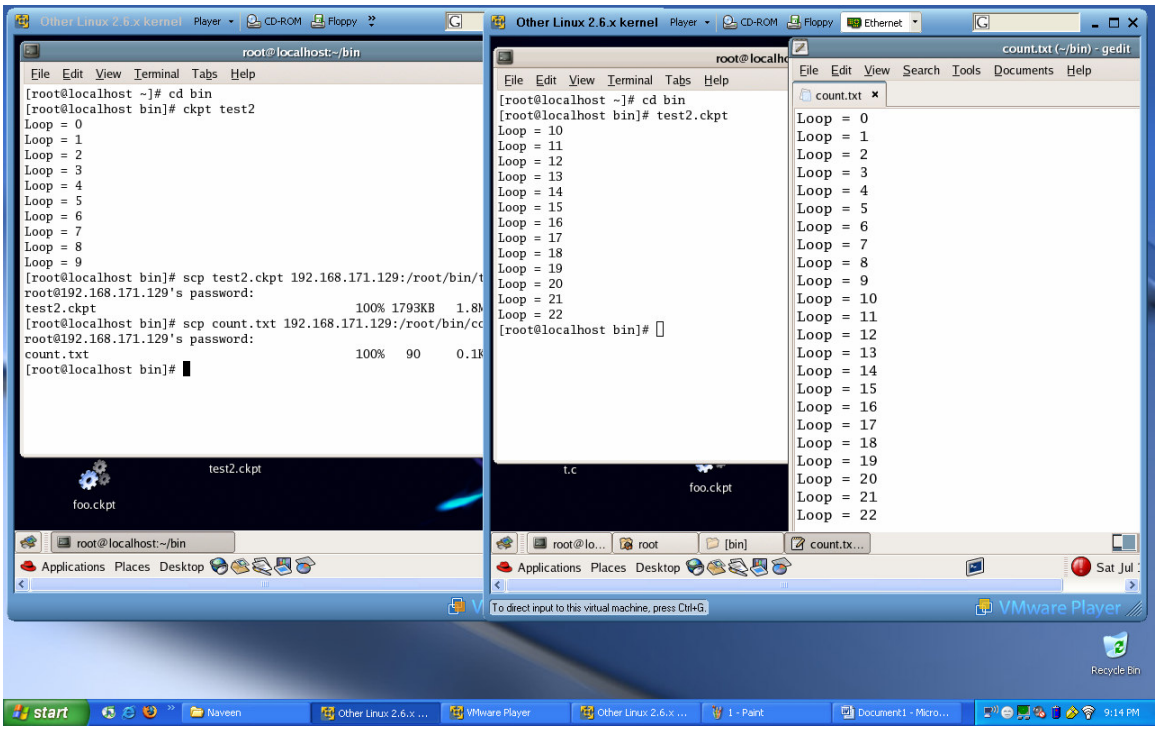


To demonstrate this application, we need a program that runs for a while. The sample program used here prints numbers from 1. When checkpoint signal is given, the image is saved. When the file containing the image is run, the process resumed and started printing from where it was stopped.

This shows the basic checkpoint restart system. The process is restarted on the same system. The migrating of the process is shown in the next page.



Here, the program is printing the numbers and also writing to a file. After the process is checkpointed, the image file and the open file are transferred to the second machine.



The process is resumed in the second machine. The process is able to write to the file from where it was stopped.

CONCLUSION

The current version of the software runs in a network of homogeneous machines running Linux operating system. The application is able to deal with normal processes and also process that open files. When the current running process that opens a file is checkpointed, the image file keeps track of the number of file descriptors that are open along with the file names, file descriptors and the current offset. When the image file is run, based on the information saved, the files are opened and the pointer is positioned based on the offset.

9.1 Limitations

The current version of our software can migrate only well defined, CPU bound processes. It cannot deal with processes with the following

- child process
- signals
- socket communication

9.2 Scope for future work

Several applications like Load Balancing, Periodic Checkpointing can be developed by making use of this Checkpoint Restart System. Also the application can be further enhanced to overcome the aforesaid limitations.

REFERENCES AND/OR BIBLIOGRAPHY

[1]. Alessandro Rubini & Jonathan Corbet, "Linux Device Drivers", 2nd Edition, 2001,

http://www.oreilly.com/catalog/linuxdrive2/

[2]. A survey of process migration schemes, Jeremy Elson.
www.circlemud.org/~jelson/writings/process-migration

[3]. M Beck et al. "Linux Kernel Internals", 2nd edition Addison-Wesley, 2000

[4]. Andrew S. Tanenbaum, "Modern Operating Systems", PHI Pvt. Ltd., 2000.

[5]. Roger S. Pressman, "Software Engineering – Practitioner's Approach", 3rd edition, McGraw-Hill, Inc.

[6]. James Gardner, "Learning Unix", 2nd edition.

APPENDIX A
USER MANUAL

1. How to install

1. Unzip the folder. Open a terminal and change the directory to the one where the folder is placed.
2. Type “make install”
3. This will install the files necessary to run the application. The files are installed in “bin” directory.

2. How to run

1. Copy a sample program to the bin directory (test2.c, foo.c).
2. Open a terminal and change the directory to “bin”.
3. run “ckpt <program-name>”
4. Checkpoint using ^Z.
5. Transfer the image file and any other files the program uses.
6. Ex: scp <file> 192.168.171.129:/root/bin
7. Open a terminal in the destination machine and change the directory to the “bin” directory.
8. run “<program>.ckpt”
9. The process resumes from the point where it was check pointed.