PROCESS MIGRATION IN NETWORK OF LINUX SYSTEMS

Ch.D.V. SUBBA RAO[1], Dr. M.M. NAIDU[1], K.V. SUBBAIAH[2] and N. RAJAKUMAR REDDY[3]

[1]Depart of Computer Science &Engg
S V University College of Engineering
Tirupati – 517 502, India.

[2]Dept of Computer Science & Engg
K S R M College of Engineering
Kadapa – 516 003, India

[3]Dept of Computer Science & Engg
Srikalahasteeswara Institute of Technology
Srikalahashti – 517 644, India

Abstract

Process migration is the act of transferring a process between two machines. It enables dynamic load distribution, fault resilience and data access locality. With the increasing deployment of distributed operating systems, process migration is receiving more attention in both research and product development. Our work deals with the design and implementation of process migration in network of homogeneous systems running on Linux. The Checkpoint /Restart System (CRS) has been developed, on top of which the migration system runs as an application. The software product developed has the following features: (1) it checkpoints a process transparently (2) it doesn't require modification of existing kernel and (3) provides a standard interface which can be used by the software packages that provide services like load balancing, load sharing, crash recovery.

Keywords

Process migration, Checkpoint and restart.

INTRODUCTION

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 of systems whose overall computational power could be increased by using more than one processor to simultaneously do 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 it is 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.

Thus 'Process Migration' is the heart of a Load Balancing system. Load Balancing is tightly coupled with the Distributed Operating Systems and has various applications and advantages. Some of them being:

- Efficient system utilization
- Fault tolerance with redundant computation
- Data access locality

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.

1. OVERVIEW OF PROCESS MIGRATION

Process migration is the relocation of a process from its current location (*current node*) to another node (*destination node*). The flow of execution of a migrating process is illustrated in the fig2.1

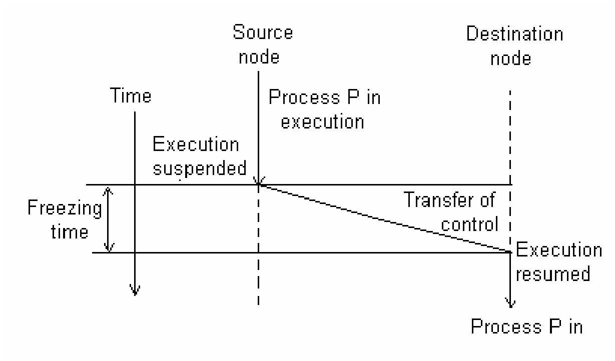


fig 2.1 Flow execution in migration process

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 paper, we deal with preemptive process migration and here after the term process migration implicitly means preemptive process migration [1,2].

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.
4. Forwarding messages meant for the migrant process.
5. Handling communication between cooperating processes that have been separated (placed on different nodes) as a result of process migration.

Note: Conventionally, the term "restart" has been used as a misnomer to "resume". We follow the same terminology.

2. 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 [3,4]. Ideally the checkpointing system should be able to save and restore the following:

- State of memory and CPU registers.
- All threads of a multithreaded process.
- Child processes of a check pointed process.
- The status of open file descriptors held by the check pointed process(es).
- The status of files `mmap()`ed by the checkpointed process(es).
- The status of the checkpointed process's signal handlers.
- The IPC status of checkpointed processes (only between simultaneously checkpointed processes).

3. 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. User level checkpointing is transparent to applications. 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.

4. DESIGN

Our design is not limited to Linux, but we have chosen Linux as a platform for our implementation. One more reason is that, its source code available.

The design of process migration involves the design of three subsystems. The subsystem namely *checkpoint subsystem* deals with checkpointing the process to be migrated, whereas the *migrate subsystem* deals with transferring the image of the process to the remote node and the *restart subsystem* resumes the process execution at the remote node using the checkpointed image.

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

4.1. Design Goals

Our goal is to design and implement an application for process migration using a transparent checkpoint /restart system.

- *Generic*

We want 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. We want our package to be general-purpose and able to checkpoint existing applications transparently, so we have to do it in the kernel space.

- *No Kernel Patches*

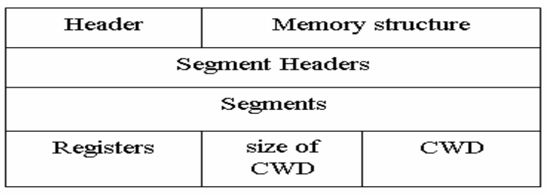
All the current checkpoint /restart packages have one common drawback: they require modification of existing code (either kernel or user applications), thus are difficult to deploy. Moreover, not all operating systems have source code available, so it's not always possible to patch the kernel. So, we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

4.2. Data Design:

The checkpointed image of a process is saved in the following format as in fig 4.1.



Header	Memory structure	
Segment Headers		
Segments		
Registers	size of CWD	CWD

fig. 4.1. Checkpoint File Format

Header:

The header contains information regarding the checkpointed 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:

This part 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 checkpointed are stored so that they can be restored when the process is restarted.

Size of current working directory:

Working directory of the checkpointed process needs 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.

4.3. 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 checkpointed by our application. To do so, it makes use of the library developed by us. The advantage of providing such a library interface is that any application such as a Periodic Checkpointing application can make use of it as shown in the figure. Process Migration application makes use of the CRS for Checkpointing and resuming the process to be migrated.

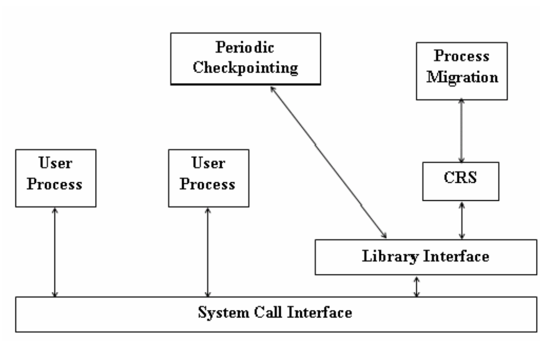


Fig. 4.2. Layered Architecture of the Process Migration application

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

In order to migrate a process to a destination0 node, a client – server model has been used as in fig 4.3

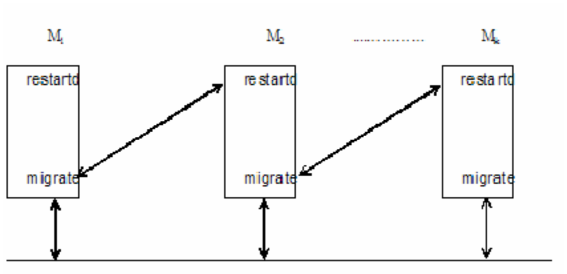


Fig.4.3. The Client-Server model

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, whereas the "restart" daemon receives the checkpointed state and restarts the process.

4.4. Component Level Design

The hierarchy chart of the system is as shown in fig4.4

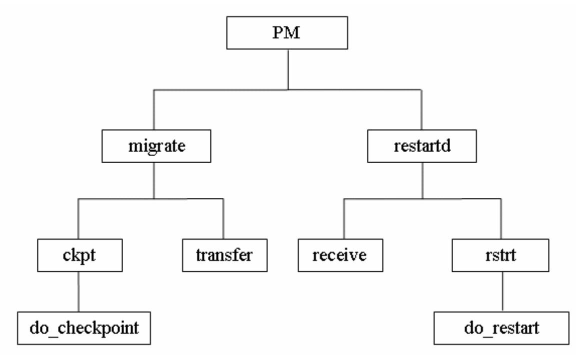


Fig. 4.4. Hierarchy Chart

The algorithm for various subsystems of our 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 then 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.

restart

1. Receive the checkpointed image from the remote client.
2. Restart the process at the destination node using the checkpointed image.

5. REALIZATION OF KERNEL-LEVEL CHECKPOINTING.

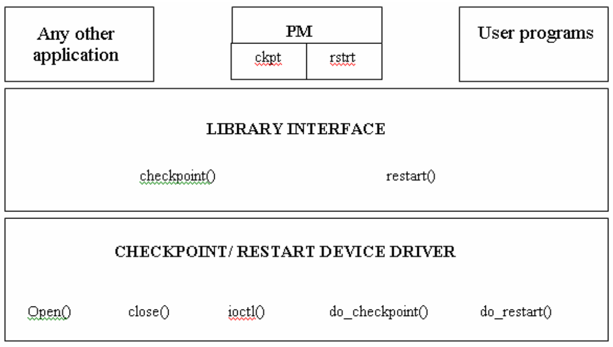
As already said the checkpointing can be done at three levels, and we are doing it in kernel level. To checkpoint any arbitrary process, the checkpointing process must be in the kernel mode. And for this it has to make a system call.

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, we 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 [5].

5.1. Realization of Checkpoint/ Restart system (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.
5. The above concept is illustrated as shown in the figure



5.2. The library Interface

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

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

Flags can be OR-ed by the following:

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

These flags are taken from Epckpt [6]. We 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, we 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:

1. 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.
2. 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.

5.3. Subsystems

5.3.1. 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 we 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.

5.3.2. Internals of do_restart()

This function is also implemented as part of the device driver. It does the exact reverse of do checkpoint (). In LINUX, the process that is running and also in kernel mode is identified by the global variable "current". When a process calls this function, 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.

5.3.3. 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.

5.3.4. 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 (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 [7,8].

6. CONCLUSIONS AND FUTURE WORK

The Checkpoint/ Restart system has been developed which runs on a network of homogeneous LINUX machines. Since we have used TCP sockets for communication between the client and server, reliability and network independence are achieved. The current version of our software can migrate only well defined, CPU bound processes. It cannot deal with processes with open files, child process, signals, and socket communication.

Due to the neat interfaces provided at various layers of the software which we have designed, several applications like Load Balancing, Periodic Checkpointing can be developed on top of our modules. Also the CRS can be further enhanced to overcome the aforesaid limitations.

REFERENCES

[1] Yeshayahu Artsy and Raphael Finkel, "Designing a process migration facility: The Charlotte experience," IEEE Computer volume:22, issue:9, pp. 47-56, September, 1989.

[2] Pradeep K. Sinha, “Distributed Operating Systems: Concepts and Design”, Wiley-IEEE Press, 1996.

[3] R. Koo and S Toueg, “Checkpointing and rollback recovery for distributed systems,” IEEE Transactions on Software Engineering,Volume: SE-13 issue:1, pp.23-31, January, 1987.

[4] J.M. Smith, “A survey of process migration mechanisms,” ACM-SIGOPS Operating systems review, pp. 28-40, 1988.

[5] A Borg, W. Graetsch, F. Herrmann, and W. Oberle, “Fault tolerance under UNIX,” ACM Transactions on Computer Systems, volume: 7, issue:1, pp. 1-24, February, 1989.

[6] EPCKPT – A Checkpoint utility for LINUX kernel http://www.cs.rutgers.edu/~edpin/epckpt.

[7] Alessandro Rubini and Jonathan Corbet, “Linux Device Drivers”, 2nd Edition, Oreilly Publications, 2001.

[8] M Beck et al., “Linux Kernel Internals”, 2nd Edition, Addison-Wesley,2000

Ch.D.V. Subba Rao is an Associate Professor in the Dept of CSE, Sri Venkateswara University, Tirupati, India. He received his B.Tech. (CSE) and M.E (CSE) from S. V. University and M. K. University respectively in the years 1991 and 1998. His areas of interest are Distributed systems, Checkpointing and Fault-tolerance, Operating systems, Computer Architecture and Programming Language Concepts.

K. Venkata Subbaiah is an Assoc. Professor of CSE, KSRM College of Engineering, Kadapa, India. He obtained his MCA and M.Tech from Osmania University and Sambalpur University respectively in the years 1997 and 2002.

Dr. M.M. Naidu is a Senior Professor in the Dept of CSE, Sri Venkateswara University, Tirupati, India. He received his Ph.D. from IIT-Delhi, India’s premier academic institution. He possesses 30 years of vast teaching and research experience. His areas of interests include Distributed database systems, Software Engg., Software Architecture, Information System Design and Computer Networks.

N. Rajakumar Reddy is an Assoc. Professor of CSE, Srikalahasteeswara Institute of Technology, Srikalahashti, India. He obtained his B.Tech and M.Tech from Bangalore University and Kakatiya University respectively in the years 1992 and 2002.