A

Project Report

On

PROCESS HIBERNATION

A Freeze – Restart Facility for Linux Processes

Submitted by

Aniket Pansare	**B-3058592**
Hitesh Oswal	**B-3058588**
Rathish Ramchandran	**B-3058589**
Tejas Shah	**B-3058615**

Department of Information Technology

Pune Institute of Computer Technology,
Survey No. 27, Pune Satara Road,
Dhanakawadi, Pune – 411 043.

UNIVERSITY OF PUNE
- 2008 – 2009 -

CERTIFICATE

This is to certify that the project report entitled

"PROCESS HIBERNATION"

Submitted by

Aniket Pansare	**Exam No: B3058592**
Hitesh Oswal	**Exam No: B3058588**
Rathish Ramchandran	**Exam No: B3058589**
Tejas Shah	**Exam No: B3058615**

is a bonafide work carried out by them under the supervision of Prof. M. R. Khodaskar and it is approved for the partial fulfillment of the requirement of University of Pune, for the award of the degree of Bachelor of Engineering (Information Technology).

(Prof. M.R. Khodaskar)	(Prof. G.P. Potdar)	(Dr. A. N. Gaikwad)
Internal Guide	Vice Principal &	Principal PICT,
IT Department	HOD – IT, PICT	Pune

Place: Pune
Date:

ACKNOWLEDGEMENTS

Undertaking a project in the final year of bachelor's degree in Engineering is a mountainous task in itself, and it requires able assistance along the way to make it successful, interesting, motivating, learning, professional and above all easy! In this we have been blessed with many people who have served as a light and mentored us towards a successful realization of this project. It is with utmost humility and gratitude that we take this opportunity to recognize every one for their invaluable inputs. We address you all to express our heartfelt greetings.

We express our sincere gratitude towards the Head of our IT Department, ***Prof. G. P. Potdar*** *for his invaluable support.*

We received zestful guidance from our project guide ***Prof. M. R. Khodaskar*** *who provided us with new ideas throughout the project. We honor his consistent inputs and able guidance that has extensively served towards the completion of the task.*

We express our sincere gratitude to our project coordinator ***Prof. S. S. Pande****. We acknowledge his valuable criticism as well as timely appreciations. It has been a truly humbling experience to work under you.*

Thank you, Sir!

We would also like to express our sincere gratitude to our external guides ***Mr. Amar Mudrankit*** *and* ***Mr. Nikhil Mishra****, GS LAB, Pune, for their invaluable support and guidance for the purpose of gathering information, making & testing our project. Hope that we have amply justified the faith that they have reposed in us from the beginning.*

We sincerely thank all of the above for making this a very enriching experience.

Aniket Pansare
Hitesh Oswal
Rathish Ramchandran
Tejas Shah

TABLE OF CONTENTS

ABSTRACT

Process Hibernation is all about taking a snapshot of a current process running on a Linux operating system and saving it on a secondary storage. Later we are able to resume the process from the point it was frozen / check-pointed.

We're all familiar with the hibernate/deep-sleep features that are typical on our standard laptop. What if you could do this at the process level in network applications? You could kill whatever umpteen-gazillion applications you have running, reboot your computer, and then start your application back up whenever you like and they would be exactly the way they were when you left them.

This is what our project aims at achieving. It allows you to capture the state of a running process in Linux communicating in a network and save it to a file. This file can then be used to resume the process later on, either after a reboot or even on another machine. It also restores back the network connection that previously existed. The saved file is self-executing and self-extracting, so to resume a process, you simply need to run the file.

Our focus is on being able to **hibernate network programs** by preserving the sockets and then restoring the socket connection.

Keywords: Hibernation, file descriptor, socket descriptor.

LIST OF DIAGRAMS

LIST OF TABLES

1. INTRODUCTION

Hibernation is a feature seen in many operating systems where the contents of RAM are written to non-volatile storage, such as the hard disk (as either a file or on a separate partition) before powering off the system. Later the system can be restored to the state it was in when hibernation was invoked, so that programs can continue execution from their previous state. Hibernating and restoring from hibernate is also generally faster than a hard reboot and, if necessary, can be done without user interaction (unlike shutting down, which often requires the user to specify if open documents should be saved). To enable hibernation, the hard disk needs to have at least as much free space as there is RAM on the system.

The current scenario of hibernation lets you hibernate the entire system by copying the contents of the RAM onto the secondary memory. Linux lets you to hibernate the entire machine, but why can't it hibernate just one process like any network or user level computational processes, record the state of the process and then resume it later.

This is what our project aims at achieving. It allows you to capture the state of a running process in Linux and save it to a file. This file can then be used to resume the process later on, either after a reboot or even on another machine. This file is self-executing and self-extracting, so to resume a process, you simply run the file.

Thus, **Process Hibernation** allows you to capture the state of a running process in Linux and save it to a file. This file can then be used to resume the process later on, either after a reboot or even on another machine with the same architecture.

1.1 PROJECT IDEA

The basic idea is to freeze a running process in Linux operating system and resume it from the same point and thus providing support for network processes.

Our project concentrates on taking an image of a currently running process by saving the contents of the registers and various sections of that process into a file which will be stored on the hard disk in an ELF format. Thus while resuming that process, first all the registers and various section's data are restored and the process is successfully resumed from the same point where it was frozen / check pointed.

1.2 NEED OF PROJECT

In current scenario there is no facility provided for hibernating single process, so with the help of our project the user will be able to freeze / checkpoint individual process any time and multiple number of times on a machine having Linux operating system. The user can then resume that process on the same machine or on different machine having similar architecture and the execution will start form the same point where it was frozen.
Some basic needs of our project are as follows –

- To conserve RAM memory and CPU time for other use.
- To save the network bandwidth.
- To bypass the slow initialization on start-up.
- To provide a comprehensive solution for devices with less memory and processing capacity like mobile phones, palmtops, etc.

Consider you are running a huge application, say you are searching for a file in a very huge database, and the application fails. Now the application has searched more than thousands of files with no positive response. To avoid wasting our time in restarting the entire process again, we checkpoint/freeze the process at every unsuccessfully trial and resume the process from last check-pointed location, to avoid redundancy of search.

1.3 LITERATURE SURVEY

Process Hibernation differs from System Hibernation because instead of storing the contents of entire RAM onto the non-volatile storage just store the contents of the process we want to hibernate in an executable format so that whenever the user want to resume that process he will just have to run that executable and the process starts executing from the same point where is was frozen.

The most challenging task was to get the data of all the sections that the process was using and storing them in an executable file. The data collected has to be stored sequentially so that while resuming there are no segmentation faults and the process gets resumed successfully from the same point. Also the addresses and offsets need to be restored properly for smooth and efficient execution of the resumed process. We have taken help of Cryopid, an open source software for this purpose.

Older methods of process hibernation did not provide support for network processes and processes handling files. So we surveyed and focused on these two types of processes and thus tried to get more and more information related to them.

There are various types of processes and implementing hibernation technique for all those types of processes was difficult so we first started with normal process which runs on terminal of Linux operating system. Then we moved forward for network processes handling sockets i.e. hibernating client-server programs.

In handling network processes we had sockets which were to be handled and that too on both - the client side and the server side. So to restore the connection was the most important thing to be done. And then we thought of writing daemon programs at both the ends i.e. the client’s end and server’s end.

These daemon programs handled the tasks of freezing the processes at the time of hibernation and while resuming the connection back they were used to establish the socket connection first and then the processes and thus the communication starts from the same state where it was check-pointed.

Similarly while handling processes using file the most important task was to store the files in the same mode they were opened and also storing their offsets at the time of hibernation. Thus while resuming these processes the files which were in use were opened first and then their offsets were also restored.

2. PROBLEM STATEMENT AND SCOPE

2.1 PROBLEM STATEMENT

The goal of this project is to hibernate a running process in Linux operating system and resume it from the same point and providing support for network processes and processes handling files.

2.2 STATEMENT OF SCOPE

The primary functions of our project are_

Hibernating network processes:

The application allows you to freeze a network process and also allows resuming the communication by establishing the socket connection between the client-server processes with the help of daemon programs.

Handling files:

Our software allows user to freeze and resume the processes handling files by fetching file descriptor information and using it while resuming the process by opening the same files in same mode and mapping the old and new descriptors.

Can start and stop a process multiple times:

With the help of our software the user can freeze and resume processes multiple times as per his need.

Can migrate processes:

The user can also migrate processes from one computer to other having same architecture by just executing the ELF file on the new machine.

2.3 AREA OF PROJECT

- System Application

3. SOFTWARE REQUIREMENT AND SPECIFICATION

3.1 PURPOSE OF DOCUMENT

This document seeks to provide the Software Requirements Specification for the project ‘Process Hibernation’, to be developed as a part of Final Year Engineering Project at Pune Institute of Computer Technology, Pune with purpose of obtaining a degree in Information Technology from Pune University. This document will be used by faculty of Information Technology Department and External Guide.

3.2 SCOPE OF DOCUMENT

This document provides detailed description of the project ‘Process Hibernation’ by explaining the Goals and Objectives of the project, software context, its usage scenario, software design constraints, hardware and software resources required, limitations and validation criteria.

3.3 GOALS AND OBJECTIVES

The goal of this project is to hibernate a running process in Linux operating system and resume it from the same point. With this application one would be able to hibernate network processes that the user feels are hogging the system resources or due to other reasons. The application will provide a tool to hibernate network process through the terminal. Later when the user wants to resume the process, the executable created during hibernation must be run through the terminal and the process is resumed successfully.

3.4 SOFTWARE CONTEXT

This application provides a facility to hibernate processes by just providing the process-id (PID) of the process. For resuming one will have to run the executable which is created at the time of hibernation. This is done only on machines having Linux operating system.

3.5 MAJOR CONSTRAINTS

The following are some of the major constraints:

1. PID of the process that is to be hibernated must be provided.
2. Resume can be done only on the system with the same architecture as where hibernation was done before.
3. If the processes using files were hibernated then those files should be present in the same location at the time of resuming.

Also the operating system used should be Linux (2.6) as of now.

3.6 HARDWARE RESOURCES REQUIRED

- Pentium III processor and above
- Linux 2.6 kernel compatible Machine
- 256 MB RAM

3.7 SOFTWARE RESOURCES REQUIRED

- Linux 2.6 kernel Operating System
- CryoPID

4. PROJECT PLAN

4.1 PURPOSE

Before developing any software system we need to plan its steps systematically. There are various methods of process of software development like waterfall model, rapid application development model, incremental model, spiral model, fourth generation techniques etc.

- Requirement analysis
- Planning
- Design & analysis
- Coding
- Testing

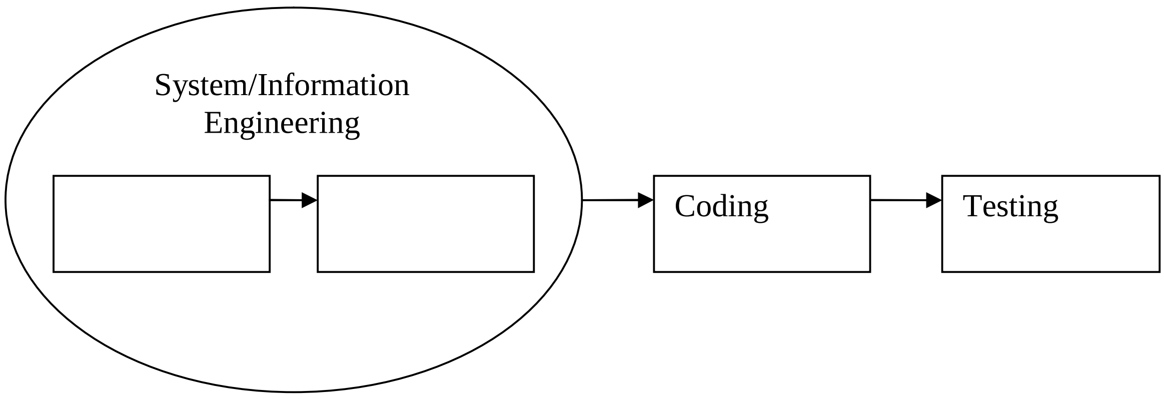


Fig 4.1 Software Development Model

This model is called as linear sequential model or waterfall model suggests a systematic sequential approach to software development that begins at the system level and progress through analysis, design, coding and testing.

4.1.1 System Information engineering and modeling

Because software is always a part of larger system, work begins by establishing requirements to software. This system view is essential when software must interact with other elements such as hardware people and database. System engineering and analysis encompass requirements gathering at the system level with a small amount of top level design and analysis. Information engineering encompass information gathering at the strategic business level and at the business area level.

4.1.2 Software Requirement Analysis

The requirement gathering process is intensified and focused specially on software. To understand the nature of the program to be built, the software engineer must understand the information domain of the software, as well as required function, behavior, performance and interface.

4.1.3 Design

Software design is actually a multi-step process that focuses on 4 district attributes of program data structures, software architecture, interface representation and procedural details. The design process translate requirement into representation of software configuration.

4.1.4 Code Generation

The design must be translated into machine readable form. The code generation step performs this task. If design is performed in detailed manner, code generation can be accomplished mechanistically.

4.1.5 Testing

Once code has been generated, program testing begins. The testing process focuses on the logical internals of the software, ensuring that all statements have been tested on the functional externals that is conducting test to uncover errors and ensure that the defined input will produced actual results that agree with required results.

4.2 PHASES

4.2.1 Project Plan of our project:

Table 4.1: Project plan

Phase	Task	Description
Phase 1	**Analysis**	**Analyze the current scenario about Process Hibernation**
Phase 2	**Literature survey**	**Collect raw data and elaborate on literature surveys. Study the current case.**
Phase 3	**Design**	**Assign the module and design the process flow control.**
Phase 4	**Implementation**	**Implement the code for all the modules and integrate all the modules.**
Phase 5	**Testing**	**Test the code and overall process whether the process works properly.**
Phase 6	**Documentation**	**Prepare the document for this project with conclusion and future enhancement.**

4.2.2 Survey on Phase Diagram:

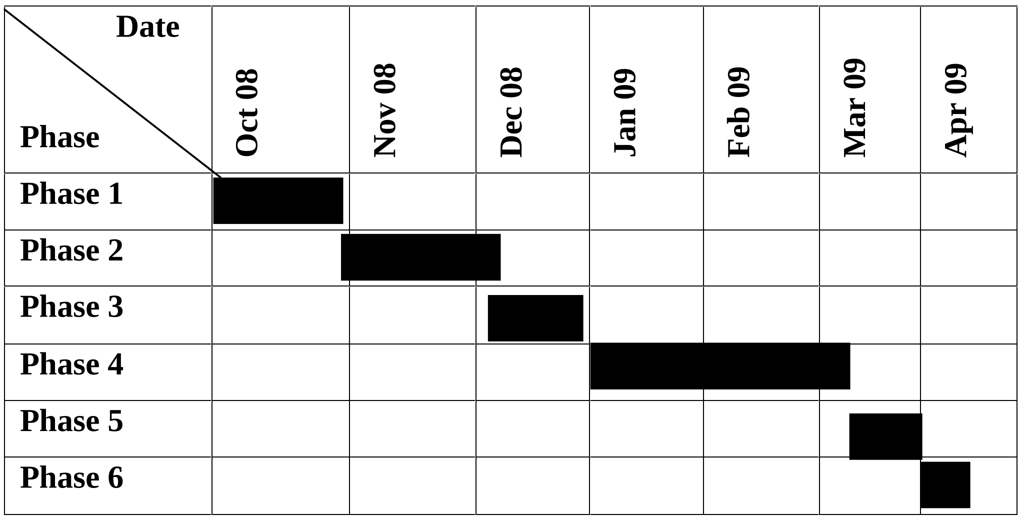


Fig 4.2 Phase Diagram

4.3 PROJECT SCHEDULE

Sr. No.	Work Task	Start Date	No. of Weeks	Done by
1.	Getting familiar with linux Process Hibernation.	5th Nov, 2008	4	Project Team
2.	Studying PTRACE.	4th Dec, 2008	1	Project Team
3.	Getting familiar with Cryopid.	10th Jan, 2009	1	Project Team
4.	Studying ELF file format.	18th Jan, 2009	2	Project Team
5.	Creating a process image.	1st Feb, 2009	3	Project Team
6.	Implementing file and socket handling.	20th Feb, 2009	3	Project Team
7.	Implementing daemon programs.	8th Mar, 2009	1	Project Team
8.	Testing.	26th Mar, 2009	2	Project Team
9.	Report Generation & Finalization.	25th Apr, 2009	2	Project Team

Table 4.2: Project Schedule

4.4 PROJECT ESTIMATES

4.4.1 Historical Data Used For Estimates

- Source code of process checkpoint tool named Cryopid was available.

4.4.2 Estimation techniques applied and results

Estimation technique named COCOMO was used to generate estimates for the project. The estimate is given by,

4.4.2.1 Estimate for technique COCOMO

COCOMO estimates development effort as function of program size and a set of "cost drivers" that include assessment of product, hardware, personnel and project attributes. Each of the 15 attributes receives a rating on a six-point scale that ranges from "very low" to "extra high". An effort multiplier from the table below applies to the rating. The product of all effort multipliers results in an *effort adjustment factor (EAF)*.

Attributes	Rating	Value
Product attributes		
1) Required software reliability	High	1.15
2) Complexity of the product	Very High	1.16
3) Required storage capacity	Low	0.85
Hardware attributes		
1) Run-time performance constraints	Very High	1.30
2) Memory constraints	Nominal	1.00
Personnel attributes		
1) Analyst capability	Low	1.19
2) Applications experience	Low	1.13
3) Software engineer capability	Nominal	1.00
4) Programming language experience	Nominal	1.00
Project attributes		
1) Use of software tools	High	0.91
2) Application of software engineering methods	High	0.91
3) Required development schedule	Nominal	1.00

Table 4.3: COCOMO Estimation

Where the effort is calculated by formula

$\mathbf{E}=a_i(\mathbf{KLoC})^{(b)_i}.\mathbf{EAF}$

Where E is effort applied in person-months,

KLoC is the estimated number of thousands of lines of code = 4

EAF is the factor calculated above = 1.64.

Coefficient $\mathbf{a_i}$=2.8 and the exponent $\mathbf{b_i}$ =1.20

E comes out to be 24.23

The Development time in months **D** = 2.5 * $E^{0.32}$ = 6.93

Number of people required P = E/D = 3.5

4.4.3 Project Task Set.

The different task sets that are critical in the project are as follows: –

Sr.no	Task ID	Name of Task
1.	T-1	Detailed study of linux process architecture and working.
2.	T-2	Study of Cryopid and its working.
3.	T-3	Analysis & Preliminary Design.
4.	T-4	Study of PTRACE and its implementation.
5.	T-5	Study of Client – Server Communication.
6.	T-6	Duplication of file and socket Descriptors.
7.	T-7	Integrating modules & Testing.
8.	T-8	Deployment

Table 4.4: Project Task Set

4.4.4 Task Network.

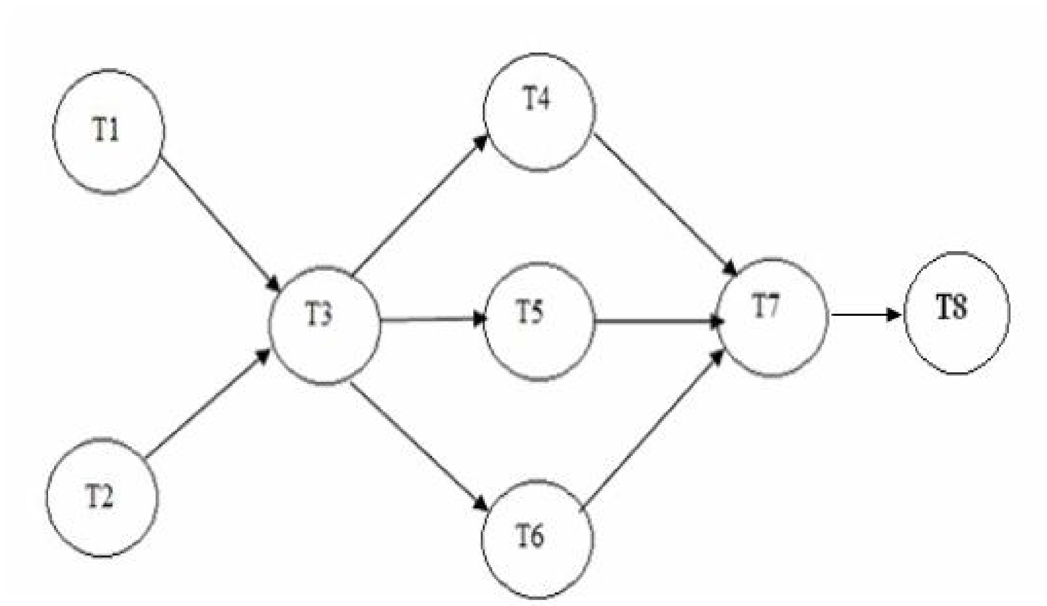


Fig 4.3: Task Network

4.5 RISK MANAGEMENT

Risk analysis and management is a series of steps that helps a software team to understand and manage uncertainty. A risk is a potential problem – it might happen, it might not. But, regardless of the outcome, it is really good idea to identify it, asses its probability of occurrences, estimate its impact and establish a contingency plan should the problem actually occur.

Recognize what can go wrong is the first step called ‘Risk Identification’. Next risk is analyzed to determine the like hood that it will occur and the damage that it will do if it does occur. Once this information is established, probability and impact rank risks. Finally a plan is developed to manage those risks with high probability and high impact.

Risks would be classified into two main categories as,

- Generic Risks
- Technical Risks

4.5.1 Generic Risks:

- The size estimate may be significantly low.
- Project members may get sick and this would result in delay in completion of their respective modules.
- The software may take longer time to build as it involves a new technology and complexity.
- Resource needed to build the software may prove insufficient.
- A few of the requirements may not be satisfied as the technology that is being built is new and time and resource may prove inadequate.
- The schedules might slip as team members are relatively inexperienced.

4.5.2 Technical Risks:

- The software debugging phase may take more time than expected.
- Software might take longer time as technology being built is new.

4.5.3 Risk Table:

Risks	Probability	Impact
Size estimate may be significantly low.	50%	Low
Delay in completion of modules.	60%	Marginal
Software might take longer time as technology being built is new.	80%	High
All requirements may not be satisfied due to lack of time and experience.	70%	High
The schedules might slip due to inexperience of members.	80%	High
Software debugging and testing phase may take longer time than expected.	50%	Low

Table 4.5: Risk Table

4.5.4 Risk Mitigation, Monitoring and Management (RMMM) Plan:

Risk planning can be organized into series of steps like Risk Mitigation, Monitoring and Management Plan. The RMMM plan documents all wok performed as part of risk analysis and is used by the project manager as part of the overall project plan. Each risk can be documented individually using risk information sheet. RIS can be maintained using database system.

Following is the RMMM plan:

Risk#1:	Size estimates may be significantly low.
Mitigation:	Keep on modularizing the software and estimate size of individual modules.
Monitor:	Modules to be written.
Management:	Increase members in more time consuming modules.

Risk#2:	Delay in completion of modules.
Mitigation:	Keep work of all members documented.
Monitor:	Work documentation.
Management:	Shift one or more members to code other member’s module.

Risk#3:	The software might take a longer time to build as it involves a new technology.
Mitigation:	Work out work schedules in consultation with external and internal guides.
Monitor:	Project schedule.
Management:	Revise schedule and increase hours of work.
Risk#4:	Resource needed to build software may prove insufficient.
Mitigation:	Estimate resource properly by consulting internal and external guides.
Monitor:	Resources.
Management:	Arrange for more resources.
Risk#5:	A few of the requirements may not be satisfied as the technology that is being built is new and time and resources may prove inadequate.
Mitigation:	Do proper requirement analysis at the beginning and review them periodically.
Monitor:	Requirements report.
Management:	Review work schedules if required and put more staff in more time consuming modules.
Risk#6:	The schedules might slip, as the team members are relatively inexperienced in project planning.
Mitigation:	Practice coding before beginning actual coding.
Monitor:	Project schedule.
Management:	Review schedules and consult internal and external guides for technical help.
Risk#7:	The software debugging phase may take more time than expected.
Mitigation:	Practice debugging and keep I touch with testing strategies.
Monitor:	Project schedule.
Management:	Review project schedules and put more efforts for debugging.

4.6 STAFF ORGANIZATION

4.6.1 Team Structure

Our team consists of four individuals namely Aniket Pansare, Tejas Shah, Hitesh Oswal and Rathish Ramchandran. We have decided to keep the team structure highly flexible throughout the project. Each individual shall contribute equally through all the phases of the project namely Problem Definition, Requirements Gathering and Analysis, Design, Coding, Testing and Documentation. Tasks shall be assigned to every member so as to speed up the progress. Also, through regular interaction each team member shall have complete knowledge of all the tasks being performed.

4.6.2 Management Reporting and Communication

The project group would either sit together and carry out certain tasks or perform task assigned to each member individually. An interaction at the start of every project meeting would inform the team about the tasks carried out by each member. The members would communicate through telephone or internet when not together. The project group will communicate with the internal and external guides every week or sooner as per project requirements with a view to collectively plan out further development and communicate the current status of the project.

4.7 TRACKING AND CONTROL MECHANISMS

4.7.1 Quality Assurance and Control

- All the necessary standards will be followed (OMG standards).
- Review of each phase (Synopsis Review, Report Review, Code Review, Testing Review, etc.) of SDLC helps to improve the project.
- Point out needed improvements in the product.
- Achieve technical work of more uniform or at least more predictable quality that can be achieved without reviews, in order to make technical work more manageable.

5. DESIGN AND IMPLEMENTATION

5.1 ARCHITECTURE

The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.

The basic architecture of the Process Hibernation:

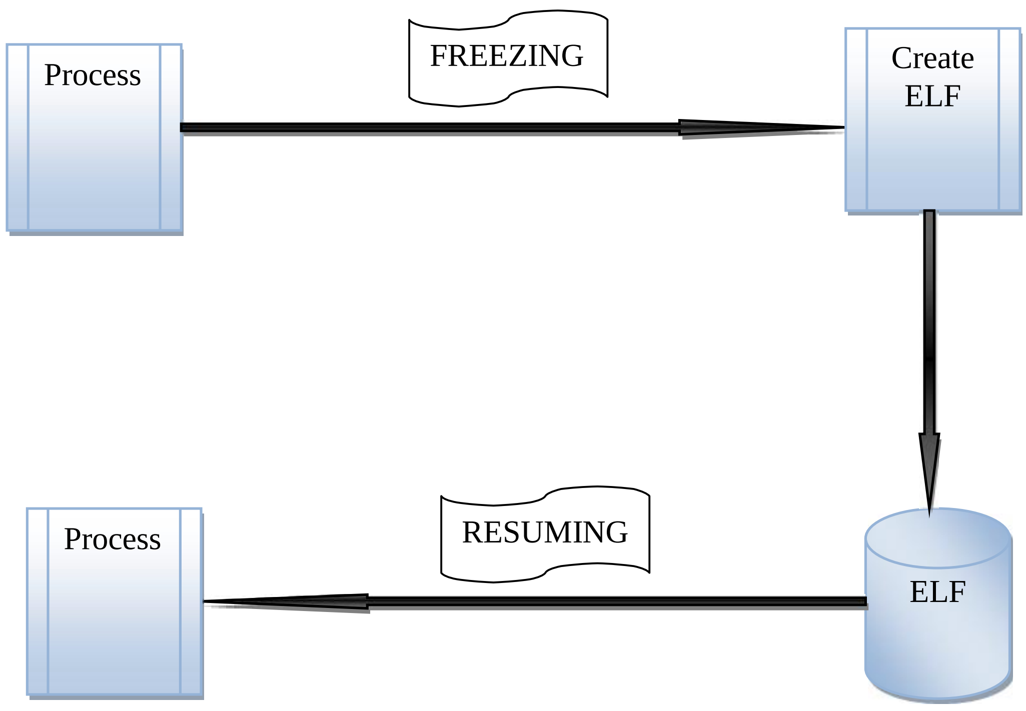


Fig 5.1 Architecture of the system

When a process is frozen, an ELF (binary executable) of that process is created. We take help of CryoPID, an Open Source software for this purpose. The ELF file contains all the resources the process was using before freeze like its registers, various sections, VMA dump etc. ELF is then stored onto the Hard disk. While resuming the process we just need to run that ELF file.

Now we need to see what an ELF is and how data is fetched from the process?

5.1.1. Fetching Process Information:

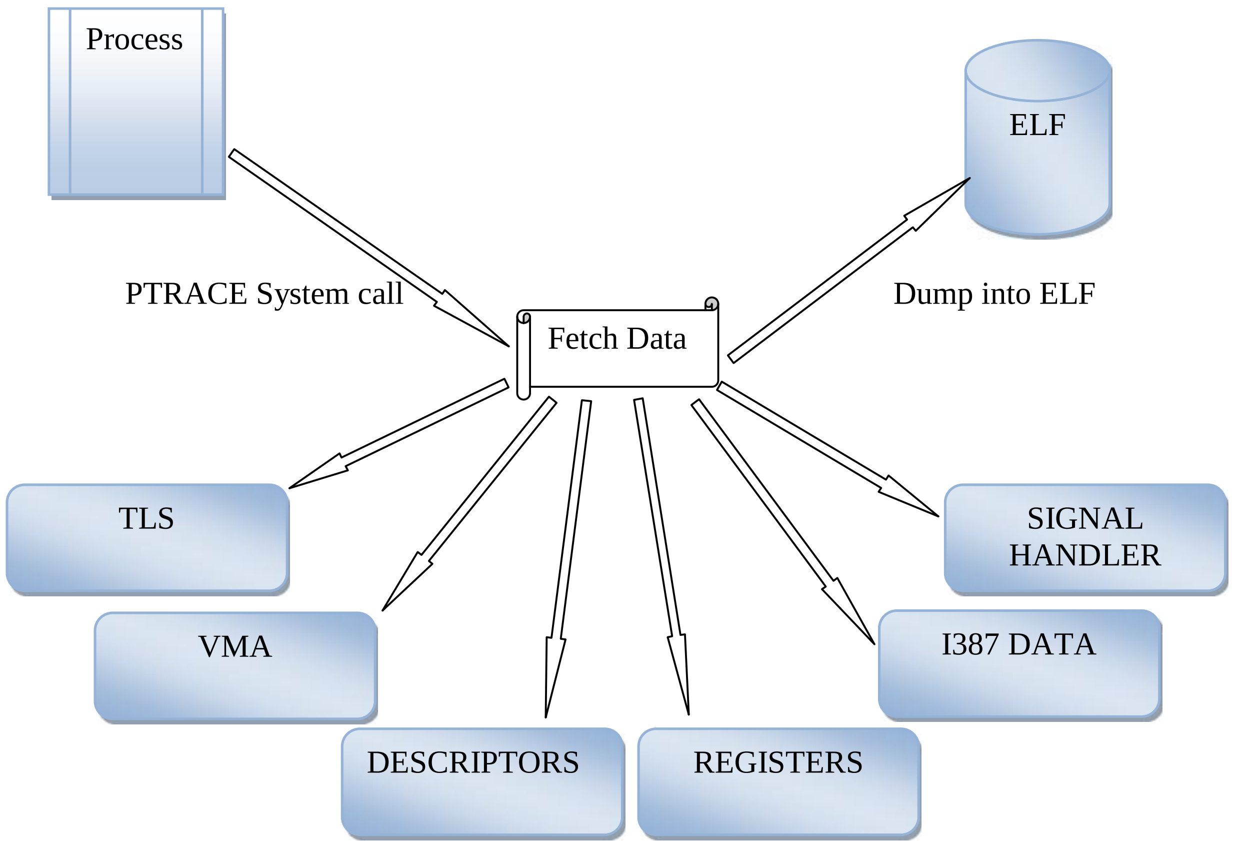


Fig 5.2 Fetching Process Information

Figure describes how data is fetched and stored into an ELF. We make use of PTRACE system call for fetching information related to the process. Information is also fetched from the /*proc*/<*pid*> of that particular process.

For hibernating a process various sections need to be taken care of:

- TLS (Thread Local Storage)
- VMA (Virtual Memory Area)
- Descriptors (File, Socket)
- Signal handler
- Registers (Instruction pointer, general purpose, floating point registers)

This fetched data during freezing is restored back in the same order while resuming a process. So we need to maintain the order of data gathering.
First we would see what is PTRACE system call?

5.1.2 PTRACE:

PTRACE is a system call which controls the execution of another process. It also enables a process to change the core image of another process. The traced process behaves normally until a signal is caught. When that occurs the process enters stopped state and informs the tracing process by a **wait ()** call. Then tracing process decides how the traced process should respond.

The prototype of ptrace () is as follows.

```
#include <sys/ptrace.h>
long int ptrace(enum __ptrace_request request, pid_t pid, void *
addr, void * data)
```

Whenever ptrace is called, what it first does is to lock the kernel. Just before returning it unlocks the kernel. Let's see its working in between this for different values of request. The various ptrace calls are as follows:

- PTRACE_TRACEME
- PTRACE_ATTACH
- PTRACE_DETACH
- PTRACE_PEEKTEXT
- PTRACE_PEEKDATA
- PTRACE_PEEKUSER
- PTRACE_PEEKUSER
- PTRACE_POKETEXT
- PTRACE_POKEDATA
- PTRACE_POKEUSER
- PTRACE_SYSCALL
- PTRACE_SINGLESTEP
- PTRACE_KILL

PTRACE_ATTACH

Attaches to the process specified in *pid*, making it a traced "child" of the calling process; the behavior of the child is as if it had done a PTRACE_TRACEME. The calling process actually becomes the parent of the child process for most purposes (e.g., it will receive notification of child events and appears in *ps* output as the child's parent), but a getpid by the child will still return the PID of the original parent. The child is sent a SIGSTOP, but will not necessarily have stopped by the completion of this call; use wait2 to wait for the child to stop.

PTRACE_DETACH

Restarts the stopped child as for PTRACE_CONT, but first detaches from the process, undoing the reparenting effect of PTRACE_ATTACH, and the effects of PTRACE_TRACEME. Although perhaps not intended, under Linux a traced child can be detached in this way regardless of which method was used to initiate tracing.

TRACE_PEEKTEXT, PTRACE_PEEKDATA

Reads a word at the location *addr* in the child's memory, returning the word as the result of the ptrace() call. Linux does not have separate text and data address spaces, so the two requests are currently equivalent.

PTRACE_PEEKUSER

Reads a word at offset *addr* in the child's USER area, which holds the registers and other information about the process (see **<sys/**user.h**>** The word is returned as the result of the ptrace() call. Typically the offset must be word-aligned, though this might vary by architecture.

PTRACE_POKETEXT, PTRACE_POKEDATA

Copies the word *data* to location *addr* in the child's memory. As above, the two requests are currently equivalent.

PTRACE_POKEUSER

Copies the word *data* to offset *addr* in the child's USER area. As above, the offset must typically be word-aligned. In order to maintain the integrity of the kernel, some modifications to the USER area are disallowed.

PTRACE_GETREGS, **PTRACE_GETFPREGS**

Copies the child's general purpose or floating-point registers, respectively, to location *data* in the parent.

PTRACE_SYSCALL, **PTRACE_SINGLESTEP**

Restarts the stopped child as for PTRACE_CONT, but arranges for the child to be stopped at the next entry to or exit from a system call, or after execution of a single instruction, respectively. (The child will also, as usual, be stopped upon receipt of a signal.) From the parent's perspective, the child will appear to have been stopped by receipt of a SIGTRAP. So, for PTRACE_SYSCALL, for example, the idea is to inspect the arguments to the system call at the first stop, then do another PTRACE_SYSCALL and inspect the return value of the system call at the second stop. The *data* argument is treated as for PTRACE_CONT.

PTRACE_KILL

Sends the child a SIGKILL to terminate it.

Basic working of PTRACE can be seen through the following figure –

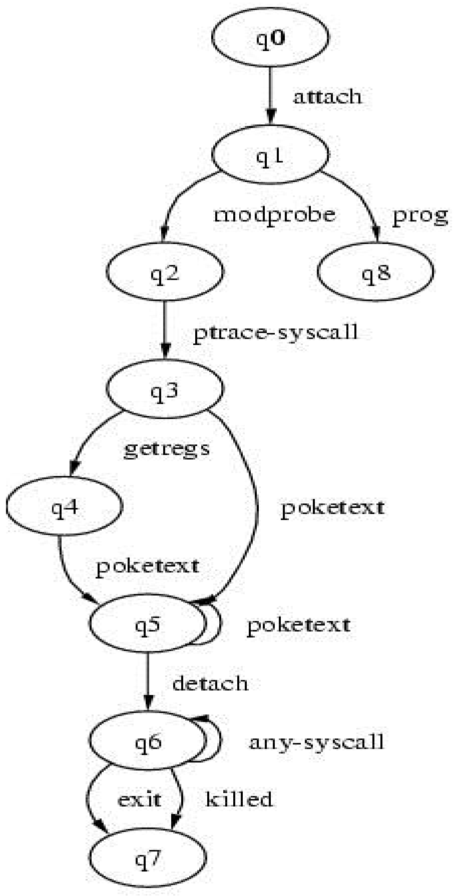


Fig 5.3 Working of PTRACE

Ptrace() is heavily used for debugging. It is also used for system call tracing. The debugger forks and the child process created is traced by the parent. The program which is to be debugged is exec'd by the child and after each instruction the parent can examine the register values of the program being run.

5.1.3 TLS (Thread Local Storage):

Thread-local storage (TLS) is a computer programming method that uses static or global memory local to thread. This is sometimes needed because all threads in a process share the same address space. In other words, data in a static or global variable is normally always located at the same memory location, when referred to by threads from the same process. Variables on the stack however are local to threads, because each thread has its own stack, residing in a different memory location.

All threads of a process share its virtual address space. The local variables of a function are unique to each thread that runs the function. However, the static and global variables are shared by all threads in the process. With *thread local storage* (TLS), you can provide unique data for each thread that the process can access using a global index. One thread allocates the index, which can be used by the other threads to retrieve the unique data associated with the index.

The constant TLS_MINIMUM_AVAILABLE defines the minimum number of TLS indexes available in each process. This minimum is guaranteed to be at least 64 for all systems. The maximum number of indexes per process is 1,088.

When the threads are created, the system allocates an array of LPVOID values for TLS, which are initialized to NULL. Before an index can be used, it must be allocated by one of the threads. Each thread stores its data for a TLS index in a *TLS slot* in the array. If the data associated with an index will fit in an LPVOID value, you can store the data directly in the TLS slot. However, if you are using a large number of indexes in this way, it is better to allocate separate storage, consolidate the data, and minimize the number of TLS slots in use.

The following diagram illustrates how TLS works.

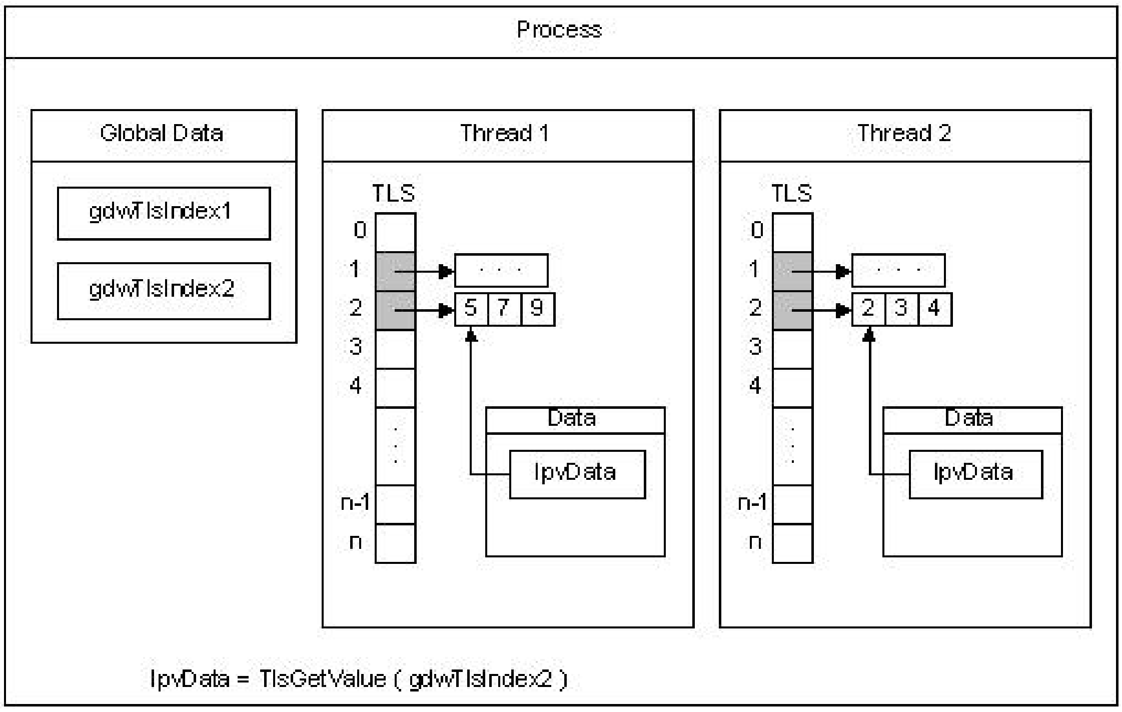


Fig 5.4 How TLS works

The process has two threads, Thread 1 and Thread 2. It allocates two indexes for use with TLS, gdwTlsIndex1 and gdwTlsIndex2. Each thread allocates two memory blocks (one for each index) in which to store the data, and stores the pointers to these memory blocks in the corresponding TLS slots. To access the data associated with an index, the thread retrieves the pointer to the memory block from the TLS slot and stores it in the lpvData local variable.

5.1.4 VMA (Virtual Memory Area):

Virtual memory is a computer system technique which gives an application program the impression that it has contiguous working memory (an address space), while in fact it may be physically fragmented and may even overflow on to disk storage. Systems that use this technique make programming of large applications easier and use real physical memory (e.g. RAM) more efficiently than those without virtual memory. Virtual memory differs significantly from memory virtualization in that virtual memory allows resources to be virtualized as memory for a specific system, as opposed to a large pool of memory being virtualized as smaller pools for many different systems.

As it can be seen from the figure 5.5, the Linux kernel stores information regarding a process in the following manner. In order to retrieve this information, we can access the /proc/<PID>/maps file in the Linux file system.

The map file has a particular format as can be seen. This contains all the files and libraries used during its execution. This file memory maps to executables and library files.

The map file for the process has the following format:

```
Address              Perms  Offset    Dev   Inode    Pathname
4001f000-40135000    r-xp   00000000  03:0c 45494    /lib/libc-2.2.4.so
```

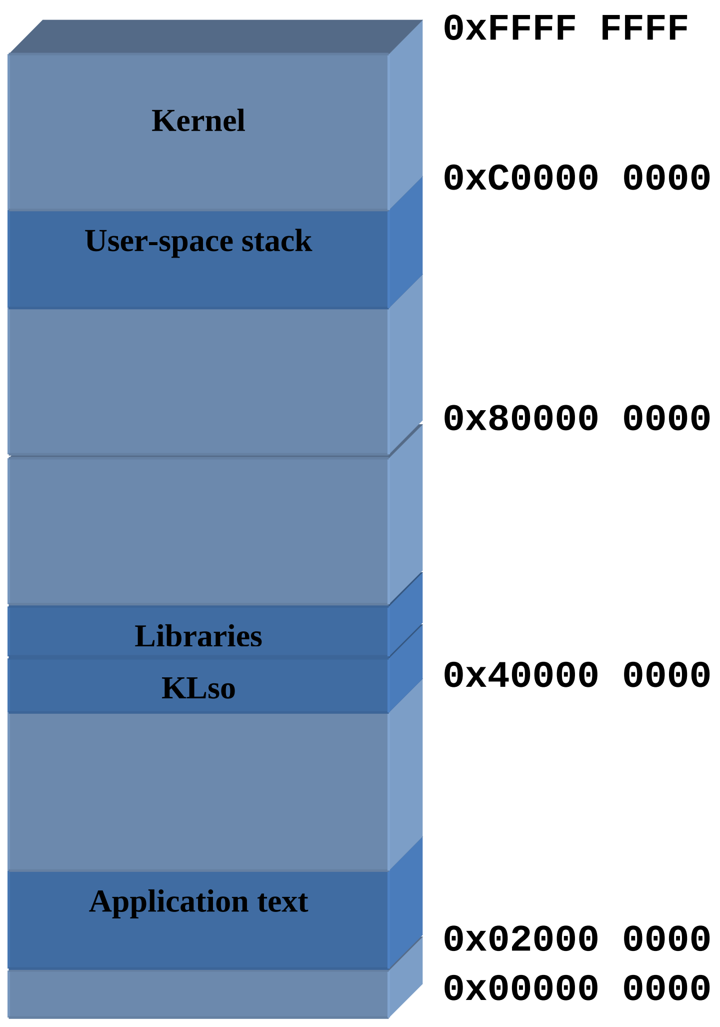


Fig 5.5 Virtual memory map

5.1.5 Descriptors (File, Socket):

When a process uses certain files or establishes a connection with another process. Various descriptors get involved. These descriptors need to be maintained during and after freeze to help the process read/write the same file or establish the connection with the same network.

For this purpose, descriptors need to be handled carefully. Following descriptors are:

1. File Descriptor
2. Socket Descriptors

- **File Descriptor:**

  A file descriptor is an abstract key for accessing a file. It is an unsigned integer used by a process to identify an open file.

  A file descriptor is an abstract value of type `Unix.file_descr`, containing information necessary to use a file: a pointer to the file, the access rights, the access modes (read or write), the current position in the file, etc.

  When a program opens a file, the operating system returns a corresponding file descriptor that the program refers to in order to process the file. A file descriptor is a low positive integer. The first three file descriptors (0,1, and 2,) are associated with the standard input (stdin), the standard output (stdout), and the standard error (stderr), respectively. Thus, the function scanf() uses stdin and the function printf() uses stdout.

  A typical UNIX program will open three files when it starts. These files are:

  - standard input (also known as stdin)
  - standard output (also known as stdout)
  - standard error (also known as stderr)

- **Socket Descriptor:**

  Sockets permit processes on one UNIX host to communicate over a network with processes on a remote host. Sockets can also be used to communicate with other processes within the same host. Local sockets can also exist within the file system. This is the type of socket that can be used only between processes within the same host. With each socket a socket descriptor is associated.

  When a program creates a socket, the operating system returns a corresponding socket descriptor that the program refers to in order to establish a connection over a network and transfer data.

Sockets require special treatment. They are not opened with the normal `open(2)` call. Instead, sockets are created with the `socket(2)` or `socketpair(2)` call. Other socket function calls are used to establish socket addresses and other operating modes.

While freezing the process handling sockets we need to store information related to the socket and re-establish the connection while resuming.

5.1.6 Signal handler:

A signal is a software interrupt delivered to a process. It is just a function that you compile together with the rest of the program. Instead of directly invoking the function, you use signal or sigaction to tell the operating system to call it when a signal arrives. This is known as establishing the handler.
There are two basic strategies you can use in signal handler functions:

- You can have the handler function note that the signal arrived by tweaking some global data structures, and then return normally.
- You can have the handler function terminate the program or transfer control to a point where it can recover from the situation that caused the signal.

When a process is executing a syscall, and at that very instant a freeze command is issued against it. Then we cannot freeze this process during its interrupt handling. We thus need to wait till the process comes out of the syscall and then freeze it. This part is done by the signal handler.

5.1.7 Registers (Instruction pointer, general purpose, floating point registers):

During its execution, process requires various registers for different purpose. These registers need to be taken care of and its value to be maintained after and before resume and freeze respectively. Such some registers are:

Instruction Pointer:

It is a processor register that indicates where the computer is in its instruction sequence. Depending on the details of the particular computer, the PC holds either the address of the instruction being executed, or the address of the next instruction to be executed. This along with Base Pointer is taken care of.

General Purpose register:

It is a small amount of storage available on the CPU whose contents can be accessed more quickly than storage available elsewhere. They can store both data and addresses, i.e., they are combined Data/Address registers.

These registers are handled using PTRACE_GETREGS option

Floating point registers:

It store floating point numbers in much architecture.

These registers are handled using PTRACE_FGETREGS option.

Thus all the information related to the process is fetched and stored into the ELF with the help of CryoPID. This executable file created, is mainly consisting of three parts:

1) ELF Header
2) Resume Code
3) Process Image

We will see all this parts in detail later on. First let us see the Design of an ELF file and what exactly it consists of.

5.1.8 ELF (Executable and Linkable Format):

ELF is a common standard file format for executables, object code, shared libraries, and core dumps. The three main types of ELF files are executable, relocatable, and shared object files. These file types hold the code, data, and information about the program that the operating system and/or link editor need to perform the appropriate actions on these files. The three types of files are summarized as follows:

- An *executable file* supplies information necessary for the operating system to create a process image suitable for executing the code and accessing the data contained within the file.
- A *relocatable file* describes how it should be linked with other object files to create an executable file or shared library.
- A *shared object file* contains information needed in both static and dynamic linking.

ELF structure in two different points of view:

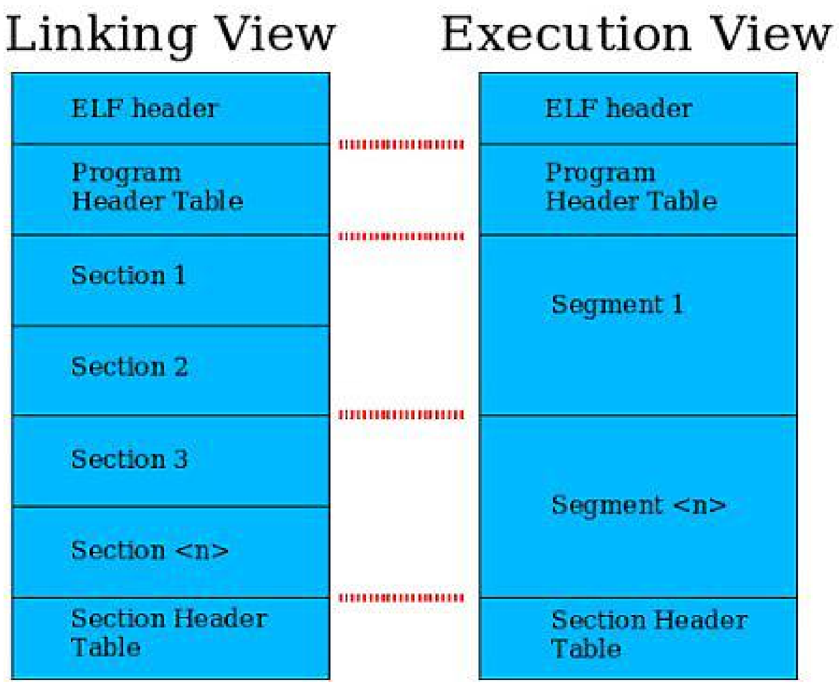


Fig 5.6 Object file format

ELF Sections:

Data Representation

```
Name            Size Alignment   Purpose
====            ==== =========   =======
Elf32_Addr       4        4      Unsigned program address
Elf32_Half       2        2      Unsigned medium integer
Elf32_Off        4        4      Unsigned file offset
Elf32_Sword      4        4      Signed large integer
Elf32_Word       4        4      Unsigned large integer
unsigned char    1        1      Unsigned small integer
```

The ELF Header:

The ELF Header is the only section that has a fixed position in the object file. It is always the first section of the file. The other sections are not guaranteed to be in any order or to even be present. The ELF Header describes the type of the object file (relocatable, executable, shared, core), its target architecture, and the version of ELF it is using. The location of the Program Header table, Section Header table, and String table along with associated number and size of entries for each table are also given. Lastly, the ELF Header contains the location of the first executable instruction.

```
#define EI_NIDENT       16
typedef struct {
    unsigned char       e_ident[EI_NIDENT];
    Elf32_Half          e_type;
    Elf32_Half          e_machine;
    Elf32_Word          e_version;
    Elf32_Addr          e_entry;
    Elf32_Off           e_phoff;
    Elf32_Off           e_shoff;
    Elf32_Word          e_flags;
    Elf32_Half          e_ehsize;
    Elf32_Half          e_phentsize;
    Elf32_Half          e_phnum;
    Elf32_Half          e_shentsize;
    Elf32_Half          e_shnum;
    Elf32_Half          e_shstrndx;
} Elf32_Ehdr;
```

The Section Header Table:

All sections in object files can be found using the Section header table. The section header, similar to the program header, is an array of structures. Each entry correlates to a section in the file. The entry provides the name, type, memory image starting address (if loadable), file offset, the section’s size in bytes, alignment, and how the information in the section should be interpreted.

Section Header

```
typedef struct {
    Elf32_Word      sh_name;
    Elf32_Word      sh_type;
    Elf32_Word      sh_flags;
    Elf32_Addr      sh_addr;
    Elf32_Off       sh_offset;
    Elf32_Word      sh_size;
    Elf32_Word      sh_link;
    Elf32_Word      sh_info;
    Elf32_Word      sh_addralign;
    Elf32_Word      sh_entsize;
} Elf32_Shdr;
```

The Program Header Table

The program header table is an array of entries where each entry is a structure describing a segment in the object file or other information needed to create an executable process image. Each entry in the program header table contains the type, file offset, physical address, virtual address, file size, memory image size, and alignment for a segment in the program. The program header is crucial to creating a process image for the object file. The operating system copies the segment (if it is loadable, i.e., if `p_type` is PT_LOAD) into memory according to the location and size information.

Program header

```
typedef struct {
    Elf32_Word      p_type;
    Elf32_Off       p_offset;
    Elf32_Addr      p_vaddr;
    Elf32_Addr      p_paddr;
    Elf32_Word      p_filesz;
    Elf32_Word      p_memsz;
    Elf32_Word      p_flags;
    Elf32_Word      p_align;
} Elf32_Phdr;
```

5.2 DESIGN SPECIFICATIONS

5.2.1 Use case Diagram

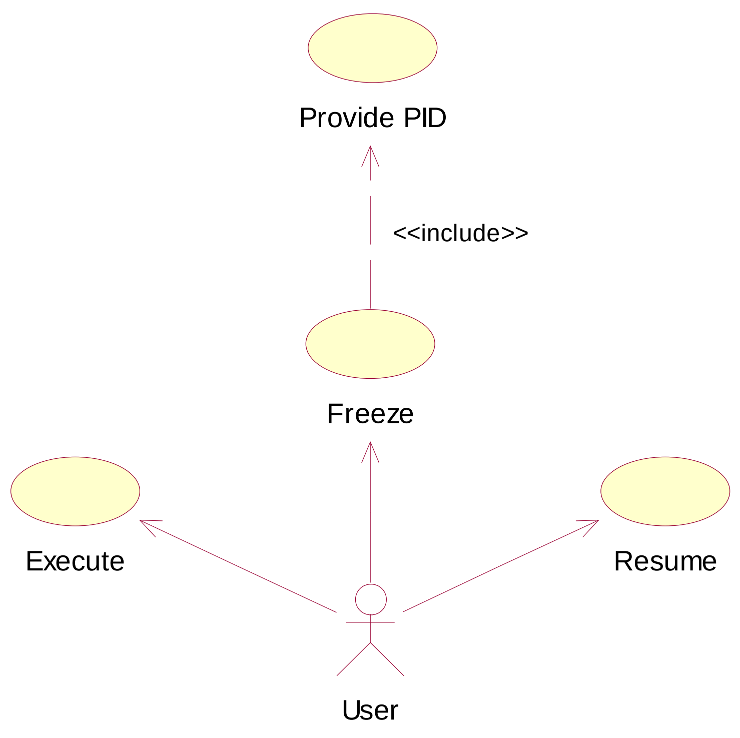


Fig 5.7 Use Case Diagram

In the above use case diagram the user of the system have the following main functionalities –

1. Execute: The user can execute any process which he/she wants to run and can also terminate it whenever he wishes do to so.

2. Freeze: The user can hibernate/checkpoint any running process and it is done by providing the PID (process identifier) of that process.

3. Resume: To resume the process bask the user have to just run th ELF file created at the time of hibernation.

5.2.2 Class Diagram

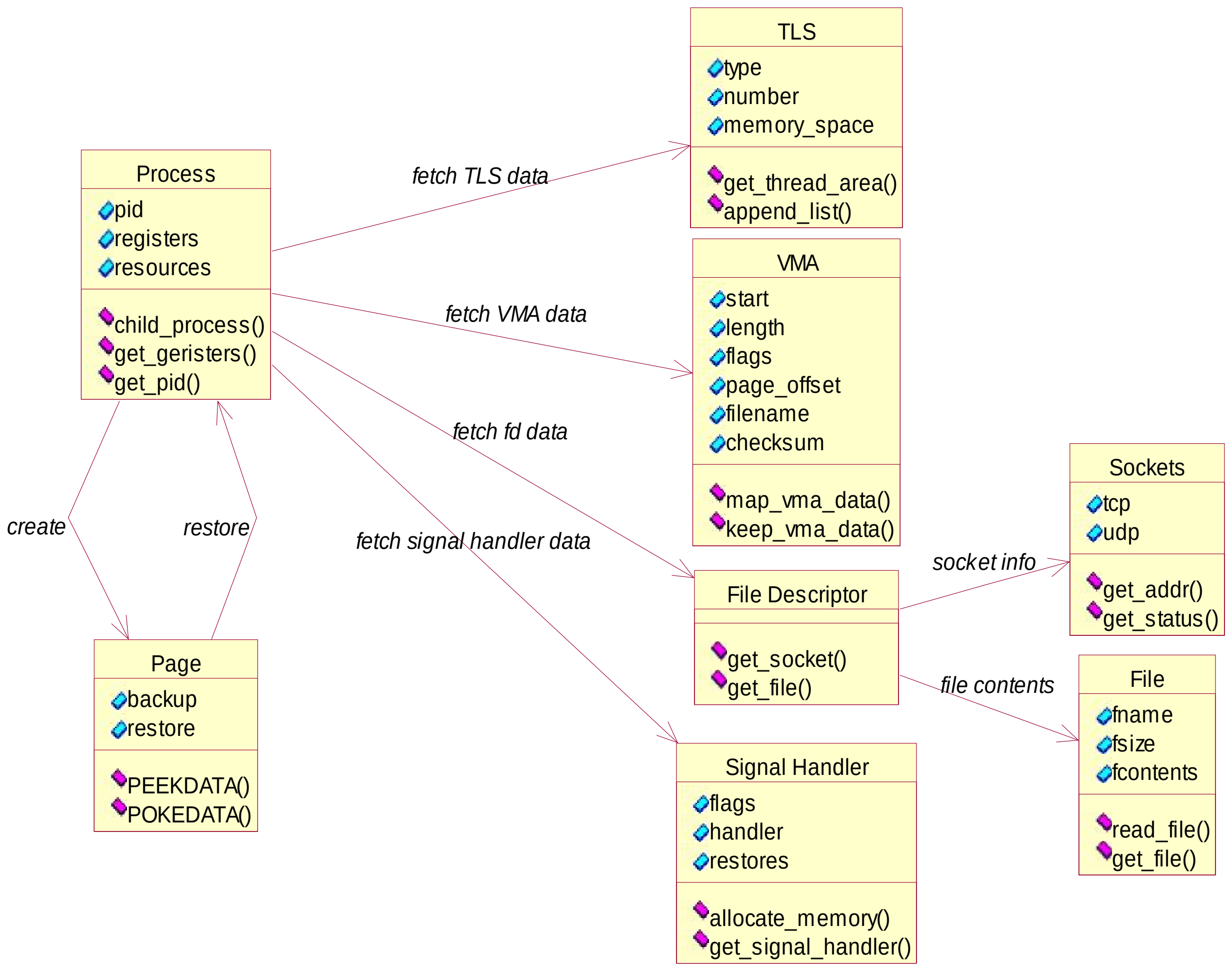


Fig 5.8 Class Diagram

5.2.3 Activity Diagram

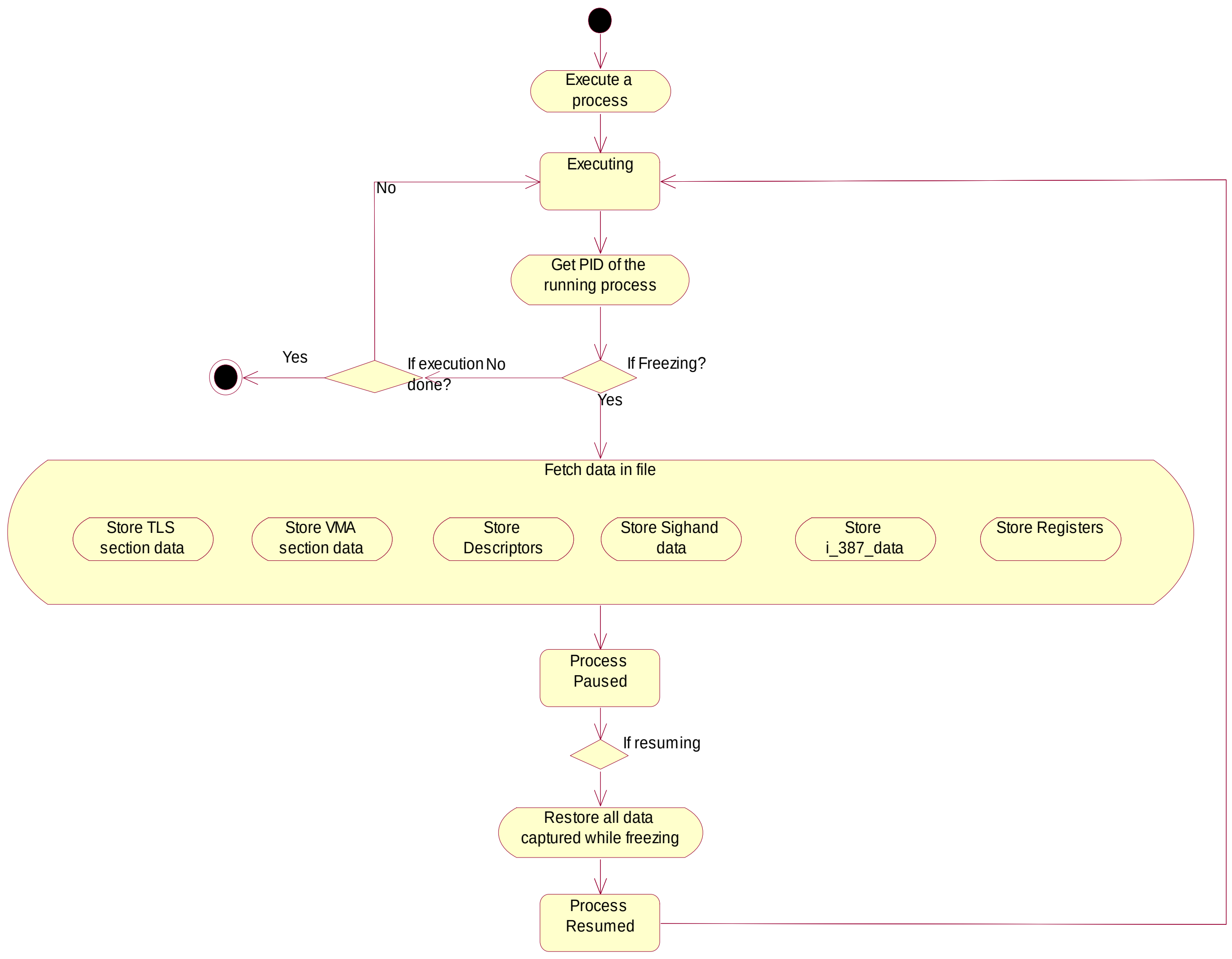


Fig 5.9 Activity Diagram

5.7.4 Sequence Diagram

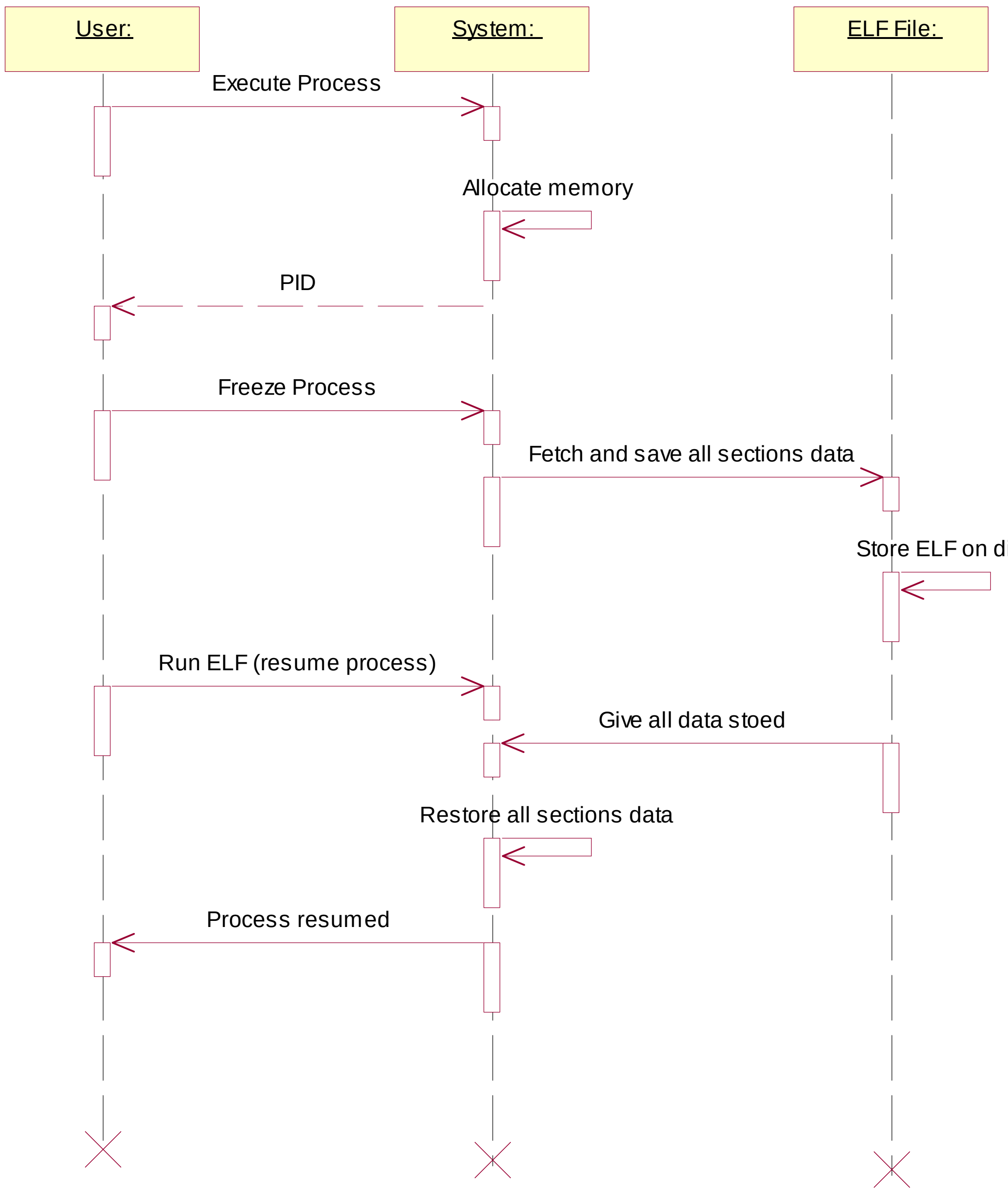


Fig 5.10 Sequence Diagram

5.2.5 Collaboration Diagram

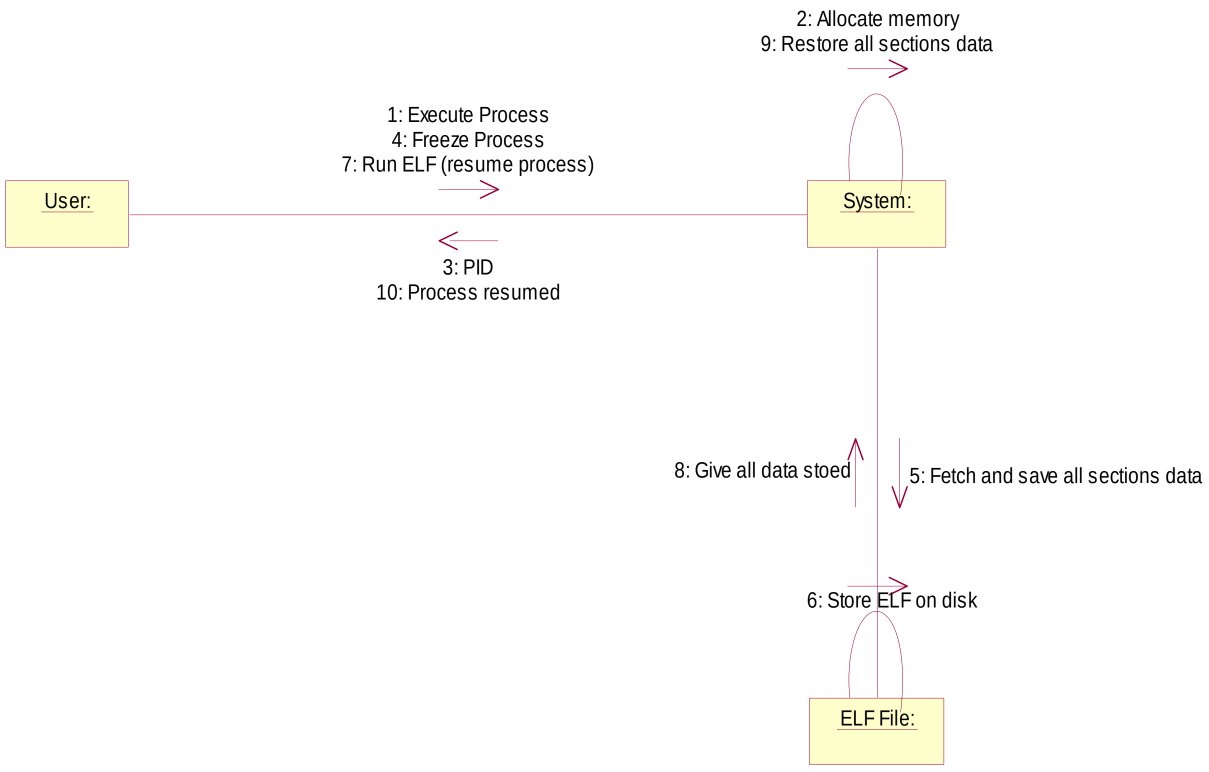


Fig 5.11 Collaboration Diagram

5.2.6 Component Diagram

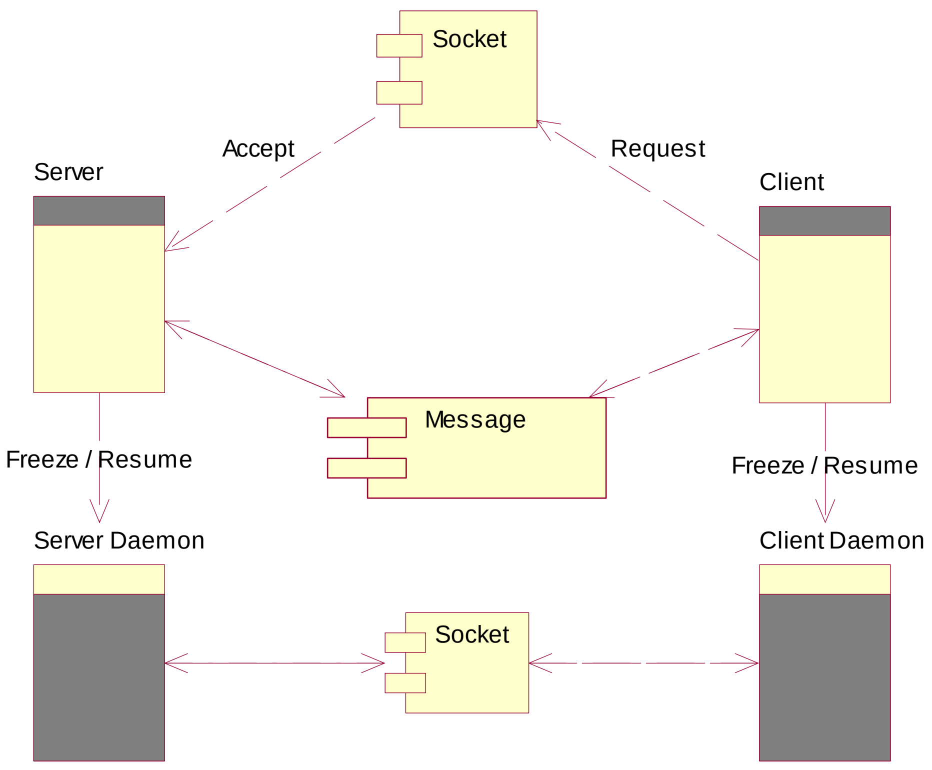


Fig 5.12 Component Diagram

5.2.7 Deployment Diagram

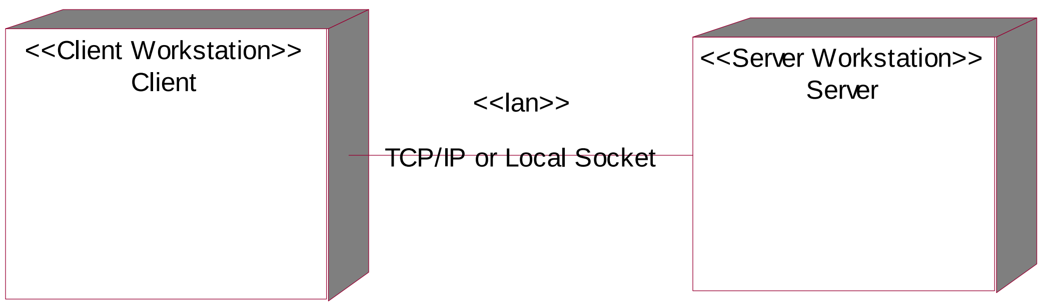


Fig 5.13 Deployment Diagram

5.3 IMPLEMENTATION DETAILS

Whenever a process is freezed an ELF of the process is generated. The Elf generated is mainly divided into three parts.

- The ELF header
- Resume code
- Process image.

The ELF header holds the information about different sections in the ELF. The Resume code is the section which helps in restoring the process image. It contains the actual code needed for handling of network processes and processes handling files. The process image contains all the information that is needed to resume the process such as the vma area, register contents , tls section , descriptors etc.

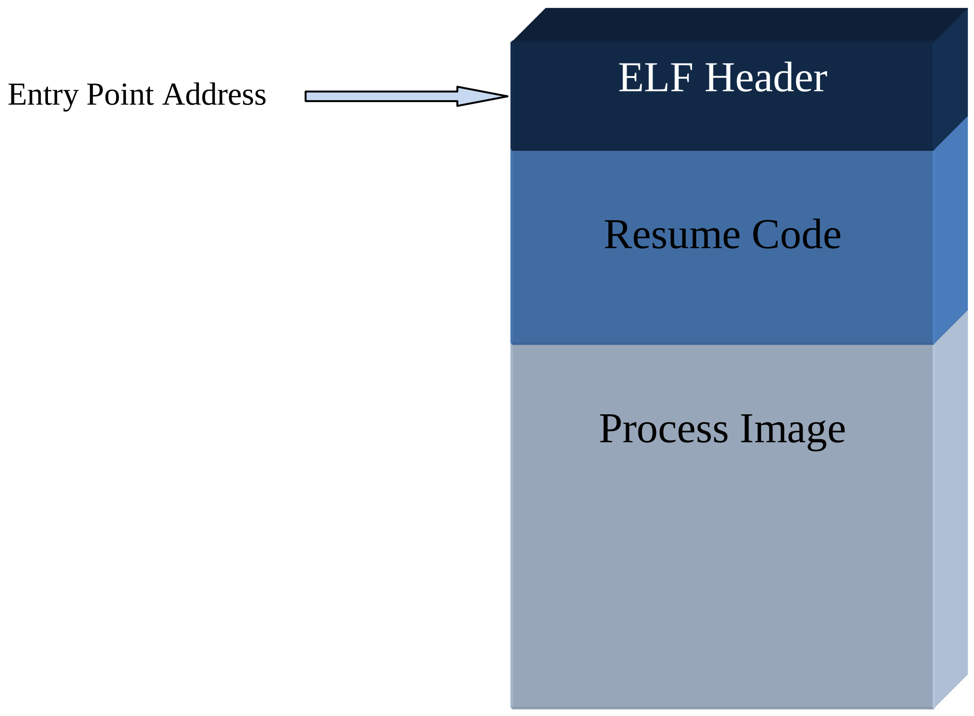


Fig 5.14 Structure of ELF file created during freeze

5.3.1 Creating and Handling ELF:

5.3.1.1 Fetching Socket and File Information

In order to be able to freeze the network processes or the processes handling files we need to fetch the information about open file and socket descriptors by that particular process. This information is fetched with the help of ptrace system call by cryopid. All the information like the type of socket, port no, filename, offset, mode in which it is opened, etc. is fetched. Thus we have the file and the socket descriptor info that were been used by the process that needs to be frozen.

5.3.1.2 Generating ELF

The ELF file is generated with the help of CryoPID which contains all the information related to the process along with files and the socket descriptors that were being opened by that particular process.

5.3.1.3 Change In ELF Generated

We need to make a change in the Resume Code of the ELF in order to be able to freeze and hibernate the network processes and processes handling files. Before resuming the process we need to re-establish the socket connection and the file descriptors that were opened earlier by the process. This is done by making some changes on the Resume Code part of the ELF.

Thus whenever the ELF is run first it will reopen the file and the socket descriptors, establish the connection and then map the old file and socket descriptors to the new one’s respectively.

5.3.2 Handling Network Processes:

Network process can be considered as a multiple client, single server architecture. In such kind of an operation two processes are involved, a client process and a server process. So whenever you want to freeze a client process its corresponding server child process communicating with respective client must also be froze. But this is not enough we also need to take care of the TCP connection between the client and the server and save the save the socket information while freezing.

In order to do this two programs are implemented,

Daemon Client

Daemon Server

5.3.2.1 Daemon client

Daemon client is a program running on the client side, having a connection with the daemon server program present on the server side. Daemon client program continuously waits in a state to accept a pid to freeze or resume.

Whenever you want to freeze the client process, you need to enter the clients pid. Daemon client communicates with the daemon server and asks it to freeze the servers child talking with our client. Then it also freezes the client. Thus we have two ELF’s generated on both client side and server side.

While resuming you need to enter the clients ELF name you need to resume. The daemon client program then communicates with the daemon server program and tells it to resume the corresponding servers ELF. Then it resumes the clients ELF.

5.3.2.2 Daemon server

Daemon server is a program running on the server side , listening for the request coming from the client daemon. Depending upon whether it’s a freeze or resume request it freezes or resumes the corresponding servers child that was communicating with the client.

Whenever freeze request comes the daemon server program identifies the server’s child that is communicating with client that needs to be frrozen. This is done with the help of LSOFF command and thus identifying the server’s child. Then the server’s child is frozen and ELF is generated.

Whenever the resume request comes the daemon server runs the corresponding servers child ELF and sends an acknowledgement message back to the client.

5.3.2.3 Freezing network processes

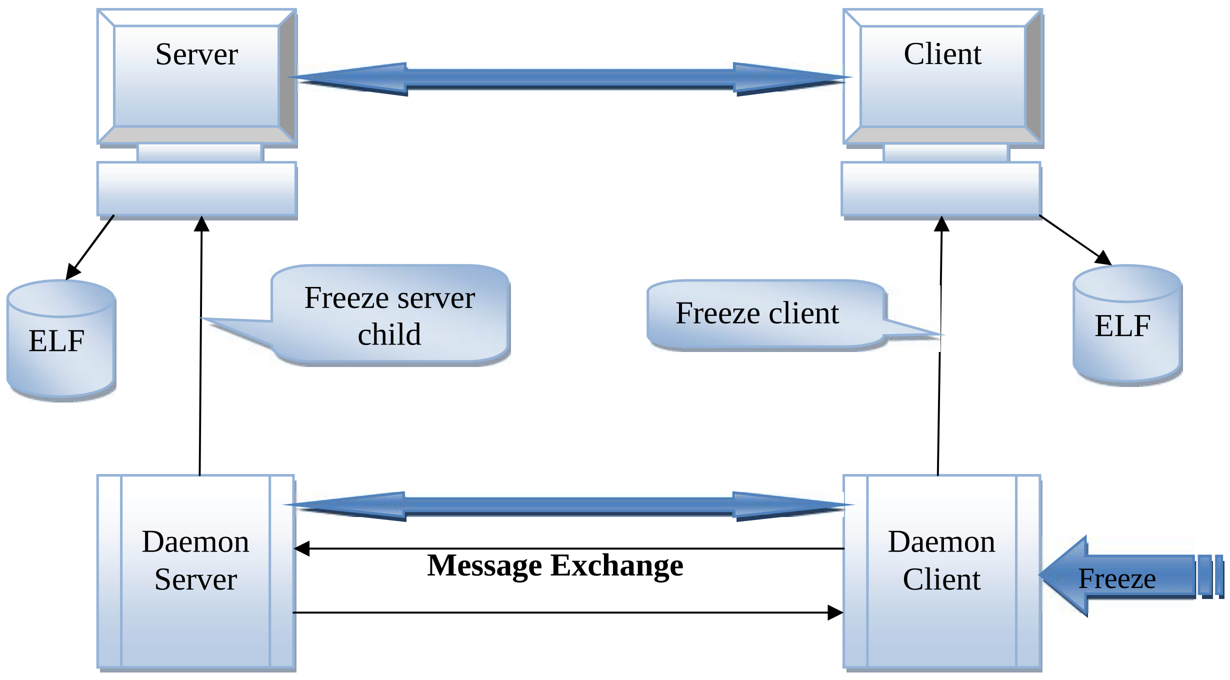


Fig 5.15 Freezing network processes

The different steps in hibernating network processes are as follows –

1. First we run the server side program which is waiting on the accept call. Then we run the client program. As soon as the client process is executed the connection is established between the client and the server. The client – server program can be any program that is transferring data or it can be a general chat application.
2. There are daemon programs running in background, i.e. the client daemon program and the server daemon program.
3. Now when the user wants to hibernate the client, it simply need to provide the PID (process identifier) to the client daemon program. The client daemon program then informs the server daemon to hibernate that child of the server which is associated with this client. So the daemon server program creates an ELF of the respective server's child and stores it on the server's hard-disk.

4. As soon as the ELF of the server's child is created the daemon server program inform the client daemon and thus it freezes the client process and the ELF is created on the client's hard-disk.
5. In this way the hibernation of client-server processes i.e. network processes takes place and the ELF's are stored onto their respective hard-disks.

5.3.2.4 Resuming network processes

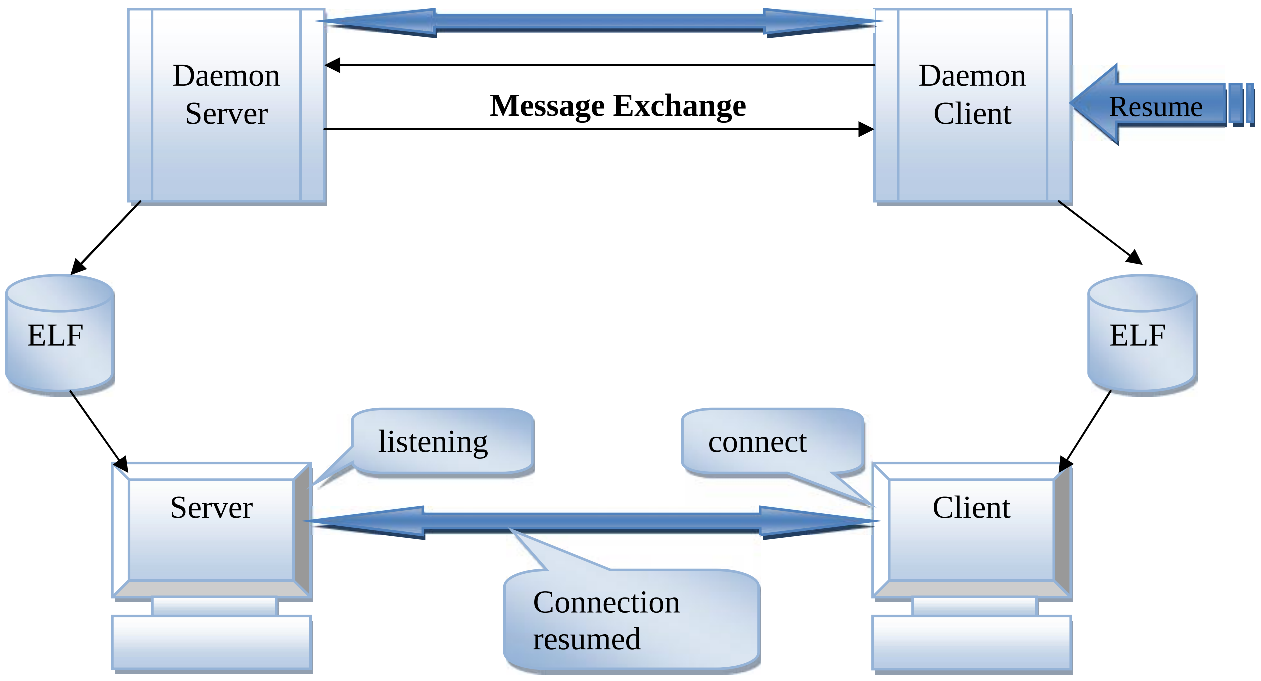


Fig 5.16 Resuming network processes

Now while resuming the network processes the following steps are carried out –

1. First we run the daemon programs i.e. the server daemon and the client daemon programs.
2. Now when the user wants to resume the process it tells the client daemon by giving 'Y/y' as input to the client daemon program. The client daemon then informs the server daemon to resume the server's ELF to which the particular client was associated.

3. But before restoring the process we need to re-establish the socket connection between the client and the server that existed earlier. So when you run the servers ELF, it first does a listen and then waits on the accept call.
4. Now the clients ELF is run which creates the socket and gives the connect call to the servers ELF that is running. As soon as it receives the connect call from the client side it accepts that call and thus the connection get established between the two.
5. Now this connection is having new socket descriptor and the one stored in the ELF is the old one. So for successful restart of the client server programs we need to do mapping of the socket descriptors. By doing mapping the data coming to the old descriptor is directed towards the new descriptors and thus the communication gets resumed successfully from the same point where it was hibernated.

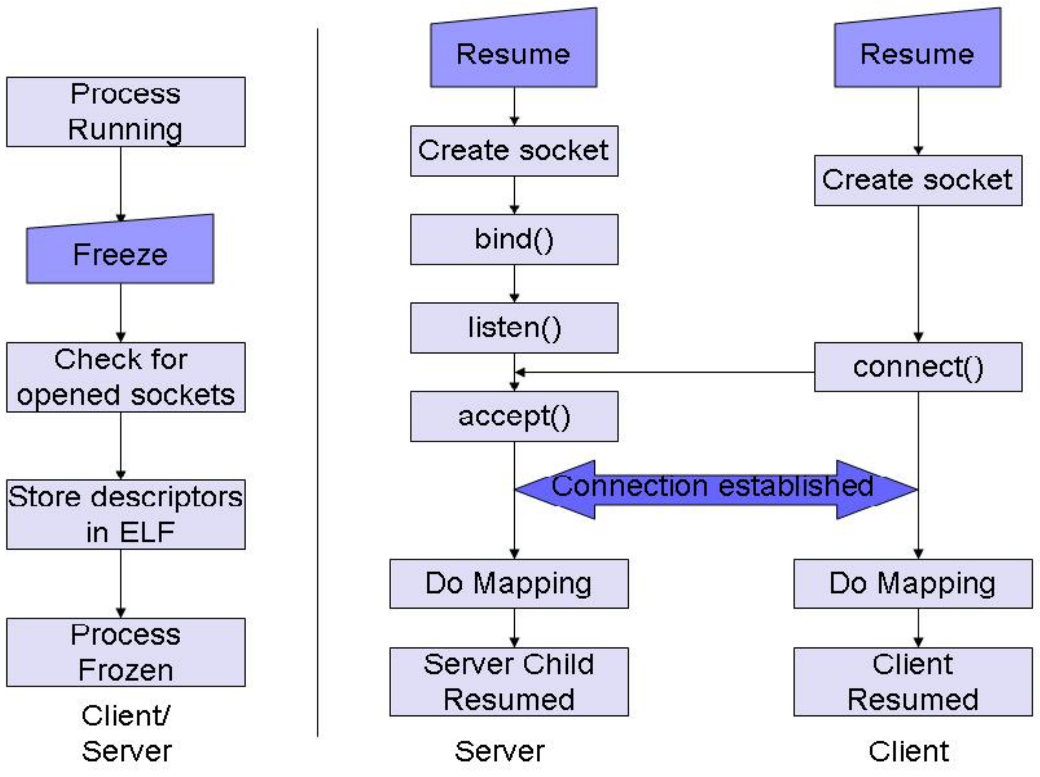


Fig 5.17 Handling Network Processes

5.3.3 Handling Files:

Similarly, while freezing processes handling file we need to take care of the file descriptors that were opened earlier by that particular process. Here, we do not need the help of daemon programs for freeze and resume.

While freezing a process handling files we store all the information about the files that were opened by that particular process in to the ELF. This includes the name of the file, path, offset, mode in which it was opened, etc. The figure below explains the freeze and resume operation.

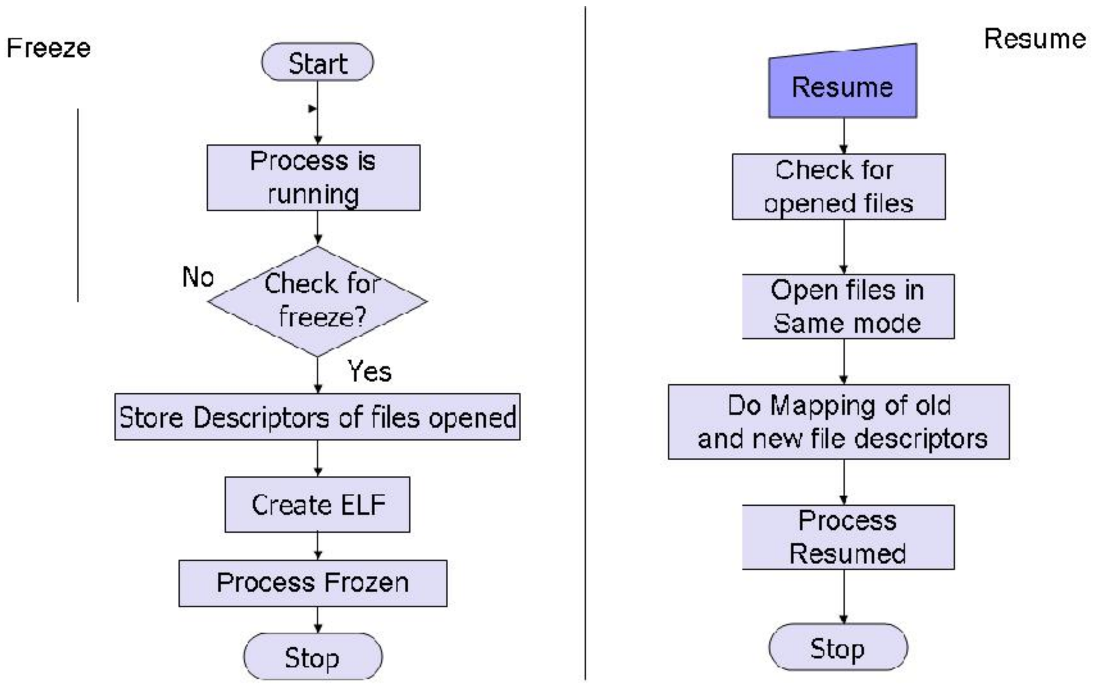


Fig 5.18 Handling processes using files

While resuming you just run the ELF. Before restoring the process image we first re-open the all the files in the same mode and offset as existed while freezing. Thus we get new file descriptors, which we need to map it to old one’s. This is done with the help of *dup2()* system call.

5.3.3.1 Freezing process handling files:

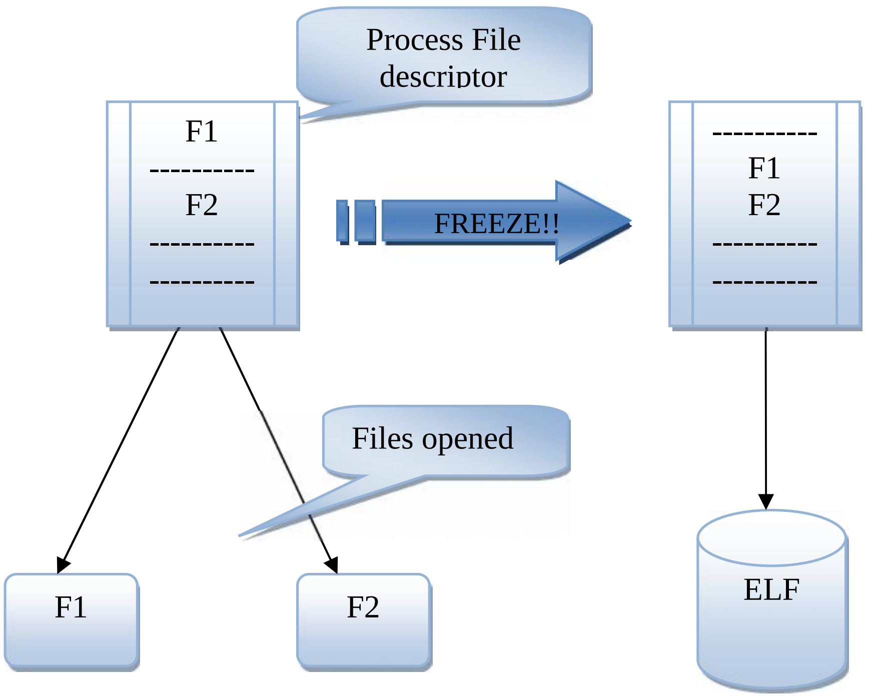


Fig 5.19 Freezing process handling files

Hibernation of processes handling files is same as that of hibernation of network processes. But while resuming processes handling files we need to open all the files in the same mode at the same location as they were before hibernation.

5.3.3.2 Resuming process handling files:

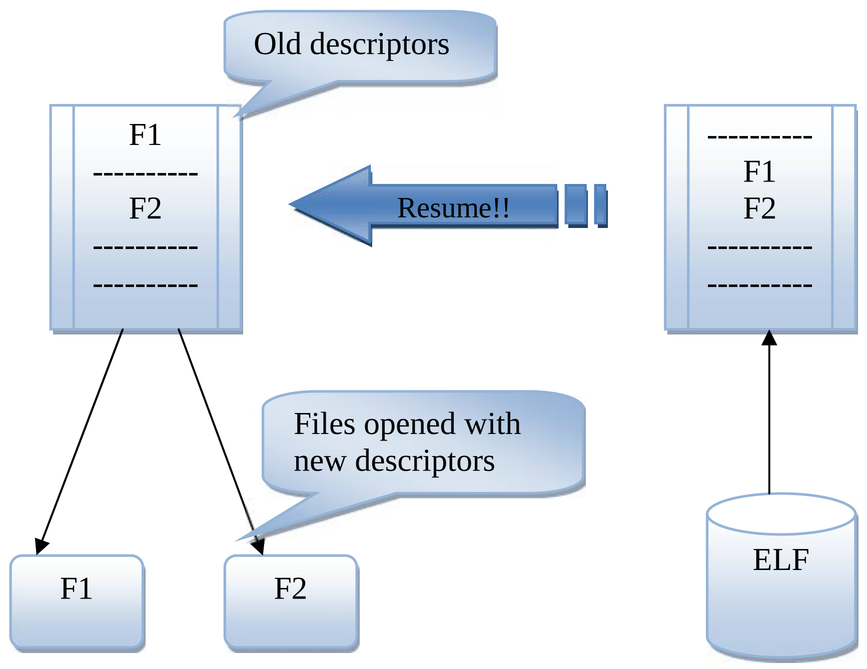


Fig 5.20 Resuming process handling files

Now when we open these file at the resuming, they are opened with the new file descriptors so we need to do mapping of the file descriptors in the same way as that of the socket descriptors for successful processing of the programs.

6. TEST PLAN AND SPECIFICATIONS

6.1. INTRODUCTION

6.1.1. Goals and objectives

Software testing is one of the elements in a software project that is often referred to as verification and validation. Verification refers to the set of activities that ensure that software correctly implements a specific function. Validation refers to a different set of activities that ensure that the software has been built is traceable to customer requirements. The results of testing will not only helps to know which parts of the system are working below average but also helps to make the system more user friendly. Testing is considered as an unavoidable part of any responsible effort to develop a software system.

6.1.2. Statement of scope

Software testing is a critical element of software quality assurance and represents the ultimate review of the specification, design and code generation. The testing process involves executing a program with the intent of finding an error. Test plan for this project involves the testing of the individual components for module interface, local data structures, boundary conditions to ensure that module operates properly, all independent paths and error handling paths are tested. It is then followed by integrating the individual components in the system, simultaneously performing regression testing so that there are no side-effects evolved due to the integration of the individual components.

Finally the system as a whole is tested, verified and validated.

6.2 TEST DELIVERABLES

- **Test plan document:**

  The test plan document describes all tests to be performed. It defines a test schedule for each module to be tested separately and also for testing the file system as a whole.

- **Test cases:**

  Each module is separately tested only for validity.
- **Sample inputs and outputs:**

  We provide the sample inputs for integration testing.

6.3. TEST PLAN

6.3.1. Testing strategies

6.3.1.1. Unit Testing

In Unit Testing strategy, initially the main focus is on each component individually, ensuring that it functions properly as a unit. Unit testing makes a heavy use of white-box testing techniques and the step can be conducted in parallel for multiple components. Various unit test considerations include testing of module interface, local data structures, boundary conditions are tested to ensure that module operates properly; all independent paths and error handling paths are tested.

Units or components in our project: the major components in our project are hibernating client-server programs (i.e. network processes) and restarting them and hibernating and restarting processes handling files. These are tested individually, also for large processes which consume lot of memory.

- **Hibernating Processes:**

  This includes functions which stores the information of various sections the process was using onto the hard-disk and helps to create a process-image which is stored in ELF.

  Testing is done to keep track of all the data fetched during hibernation is appropriate and correct and also testing is done to avoid segmentation fault which occurs due to inappropriate addresses and offsets.

- **Resuming Processes:**

  While resuming processes, we simply run the ELF file which is created and stored on the hard-disk at the time of hibernation. As soon as we run this ELF the same scenario is created which was there at the time of hibernation and also the values of different sections are stored back to their original places and the process starts executing from the same point.

  Testing need to be done to check that the values are stored back to their original positions only, if not then there will be segmentation fault and also the process will fail to resume.

6.3.1.2. Integration Testing

The Integration Testing is on design and the construction of the software architecture. The individually tested components are integrated as a complete software package. Black-Box test-case design techniques are the most prevalent during integration. Top-down integration testing is an incremental approach to construction of program structure, thereby verifying major control or decision points early in the test process. Bottom-up integration testing begins construction and testing with atomic modules i.e. components at the lowest levels in the program structure.

This includes issues like –

- **Hibernating and Resuming Network Processes:**

  Hibernation of network process is same as normal process hibernation. But while resuming network processes, before executing the ELFs of client and server we need to create and test the connection between them and then only the processes are resumed successfully. The different socket calls are made from both the server and the client side which are written inside the daemon programs. After the connection gets established and it is tested successfully the communication and data transfer gets resumed.

6.3.1.3. Validation Testing

As the software has been integrated a setoff higher order tests are conducted. Validation criteria, established during requirement analysis must be tested.

6.3.1.4. System Testing

The product is tested in the context of the entire system. Different Linux systems were used for system testing and the execution of processes was monitored.

6.4 RESPONSIBILITIES AND SCHEDULE

Each developer is responsible for running adequate tests on their assigned unit. This may be improved with some outside testing from other group members. The overall test of the system will be the responsibility of each group members collaborating on the testing process as well as test outcomes.

In general testing should occur at the completion of each unit as then again if the initial testing failed and further development was required. After the integration phase more testing will occur .If any failure during this stage can be pin-pointed to a unit the development and testing will fall back to that unit followed by further integration testing.

6.5 TEST CASES

Table 6.1: Test Cases

Test case Id	Objective	Prerequisites	Step No	Step Description	Expected Result	Actual Result	Remark
1	To check Handling of TLS segment	1. Software should be properly installed	1.	Check return value of ptrace	Return value of ptrace should not be -1	Return value of ptrace should not be -1	Pass
		2. Process should be frozen properly	2.	Check value of chunk->type	Value of chunk->type should be "TLS"	Value of chunk->type is "TLS"	Pass
			3.	Check if chunk->tls.u = u	Value of chunk->tls.u should be equal to value of u	Value of chunk->tls.u is equal to value of u	Pass
			4.	Check if chunk is appended in process image in list.h	Chunk should be appended in process image in list.h	Chunk is appended to the process image in list.h	Pass
2	To check Handling of one VMA area	1. Software should be properly installed	1.	Check values in chunk->vma in cp_w_vma.c	Values in chunk->vma should be present and must be correct	Values are present in chunk->vma	Pass
		2. Process should be frozen properly	2.	Check if chunk->vma->data contains syscall function	Chunk->vma->data contains syscall function	Chunk->vma->data contains syscall function	Pass
			3.	Check if chunk is appended in process image in list.h	Chunk should be appended in process image in list.h	Chunk is appended to the process image in list.h	Pass
3	To check Handling of File descriptors	1. Software should be properly installed	1.	Check is value of chunk->fd.console to check if it contains console informtion	Value in chunk->fd.console must be present and correct	Value in chunk->fd. console is present and correct	Pass
		2. Process should be frozen properly	2.	Check value of chunk->fd.socket to check if it contains socket information	Value in chunk->fd.socket must be present and correct	Value in chunk->fd.socket is present and correct	Pass
			3.	Check value of chunk->fd.file to check if it contains file information	Value in chunk->fd.file must be present and correct	Value in chunk->fd.file is present and correct	Pass
			4.	Check value of chunk->fd.fifo to check if it contains fifo information	Value in chunk->fd.fifo must be present and correct	Value in chunk->fd.fifo is present and correct	Pass
			5.	Check if chunk is appended in process image in list.h	Chunk should be appended in process image in list.h	Chunk is appended to the process image in list.h	Pass

4	To check Handling of Sighand	1. Software should be properly installed	1.	Check value of chunk->type	Value of chunk->type should be "CP_CHUNK_SIGHAND"	Value of chunk->type is "CP_CHUNK_SIGHAND"	Pass
		2. Process should be frozen properly	2.	Check return value of ptrace	Chunk should be appended in process image in list.h	Chunk is appended to process image in list.h	Pass
				Check value of chunk->type			
5	To check Handling of i387Data	1. Software should be properly installed	1.	Check return value of ptrace	Return value of ptrace should not be -1	Return value of ptrace should not be -1	Pass
		2. Process should be frozen properly	2.	Check value of chunk->type	Value of chunk->type should be "CP_CHUNK_I387_DATA"	Value of chunk->type is "CP_CHUNK_I387_DATA"	Pass
			3.	Check if chunk is appended in process image in list.h	Chunk should be appended in process image in list.h	Chunk is appended to the process image in list.h	Pass
6	To check Handling of Registers	1. Software should be properly installed	1.	Check value of chunk->type	Value of chunk->type should be "REGS"	Value of chunk->type is "REGS"	Pass
		2. Process should be frozen properly	2.	Check if chunk is appended in process image in list.h	Chunk should be appended in process image in list.h	Chunk is appended to the process image in list.h	Pass

7. FUTURE WORK

Adding some of the following features can further enhance the application and can increase its usability –

- Hibernating and restarting processes handling pipes.
- Hibernating and restarting X-applications.
- Handling hibernation and restart of shared memory processes.

Process Hibernation will be very helpful for devices having low processing power and less memory space like mobiles say, because there can be many processes running on a mobile phone in background but at a time we can only see and operate only a single process. So instead of keeping those processes running them in background we can simply hibernate them and can again resume whenever required. Thus instead of context switching we can hibernate them and then can resume them any time as per one’s need n requirement.

Thus process hibernation has got a long way to go in future as there are always developments in the types of devices and the processes that run on them. Therefore, if we are able to hibernate and restart all such processes it will help the users to perform their work fast and without any loss of time and data if in case the system restarts accidently or if there are any kind of failures.

8. CONCLUSION

In the performance oriented field of network processes, our project provides a feasible solution to freeze and restart network processes and thus will help in faster execution of network processes by hibernating the processes consuming lot of bandwidth and then resuming them again later as per the requirement.

The software also provides support for the hibernation of processes performing file operations and restarting them from the same point and from the same offset.

Thus by hibernating and restarting various processes we help the user of the Linux operating system to perform their work faster and also check-pointing large processes at regular intervals which will help the user of restart that process faster without any loss of time and data at the time of system restart and failure.

Hibernation also seems to be a very good feature for executing any urgent or important process immediately, by freezing the currently running processes and then resuming them back, after successful execution of the urgent ones.

REFERENCES

[1] CryoPID - A Process Freezer for Linux
http://cryopid.berlios.de/

[2] Executable and Linkable Format (ELF)
http://www.skyfree.org/linux/references/ELF_Format.pdf
http://www.muppetlabs.com/~breadbox/software/ELF.txt

[3] Playing with PTRACE, Part I, Part II
http://www.linuxjournal.com/article/6100
http://www.linuxjournal.com/node/6210

[4] Linux cross reference
http://www.isovarvas.com/lxr/http/source/include/asm-arm/irq.h?a=ppc

[5] Advanced UNIX Programming by Richard Stevens, 2nd edition, 2005, ISBN: -03-215-2594-9

[6] The Design of Unix Operating System by Maurice J. Bach, 2006, ISBN:-81-203-0516-7

[7] Understanding The Linux Kernel by Daniel Bovet, 3rd edition, 2007, ISBN 10: 81-8404-083-0

[8] Linux Socket Programming by example by Warren W. Gay, 2000, ISBN:- 0-7897-2241-0

A] Installation Guide for 'Process Hibernation':

In order to be able to hibernate a process, you must have following things on your computer.

1) Linux operating system
2) Freeze software.
3) Hardware support.

Let us see the Freeze software in detail and how to install it on your machine. There are basically three different folders as shown below:

```
--/freeze
  |
  |--/src
     |
     |--/arch-i386
     |
```

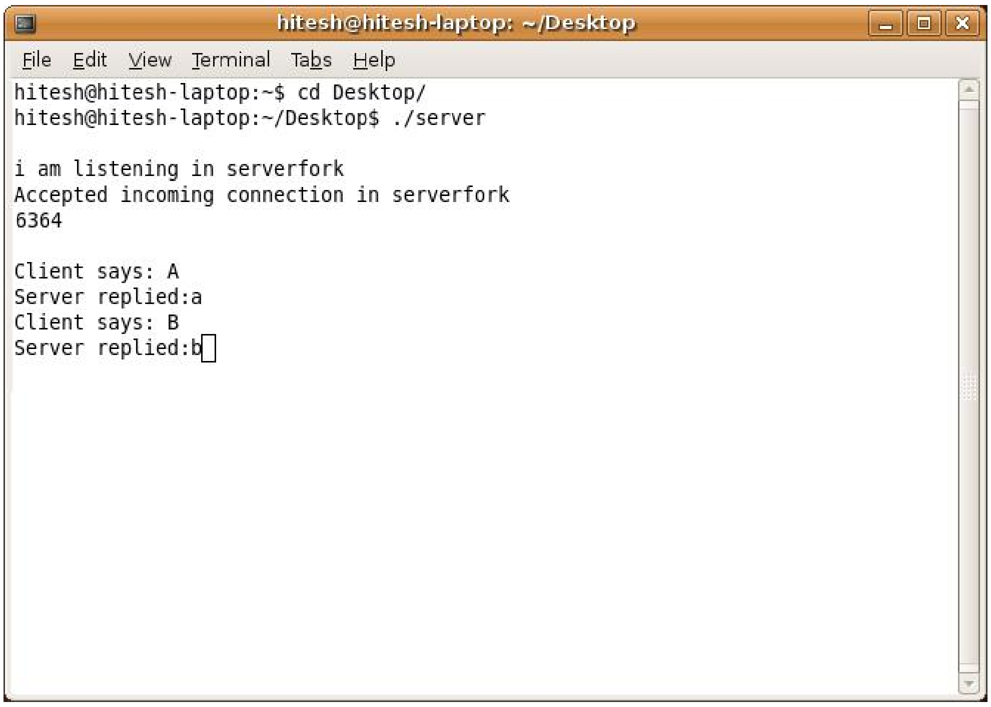
Out of this, 'freeze' folder contains a dietlibc.a file and folders. It also contains a folder named 'src', which has folder named 'arch-i386'. Each of these folders contains its respective module and source. You need to execute 'make' in the 'src' folder if object files and executable are not present.

Once the 'make' is done a binary executable named 'freeze' gets created along with the daemon server and client executables. This 'freeze' executable is used for hibernating a process and creating its ELF. It takes PID of the process as an argument.

Your task is just to execute daemon client and server programs. Then just enter the PID of the client program that you need to freeze in the daemon client program.

B] Screenshots of Steps to Hibernate:

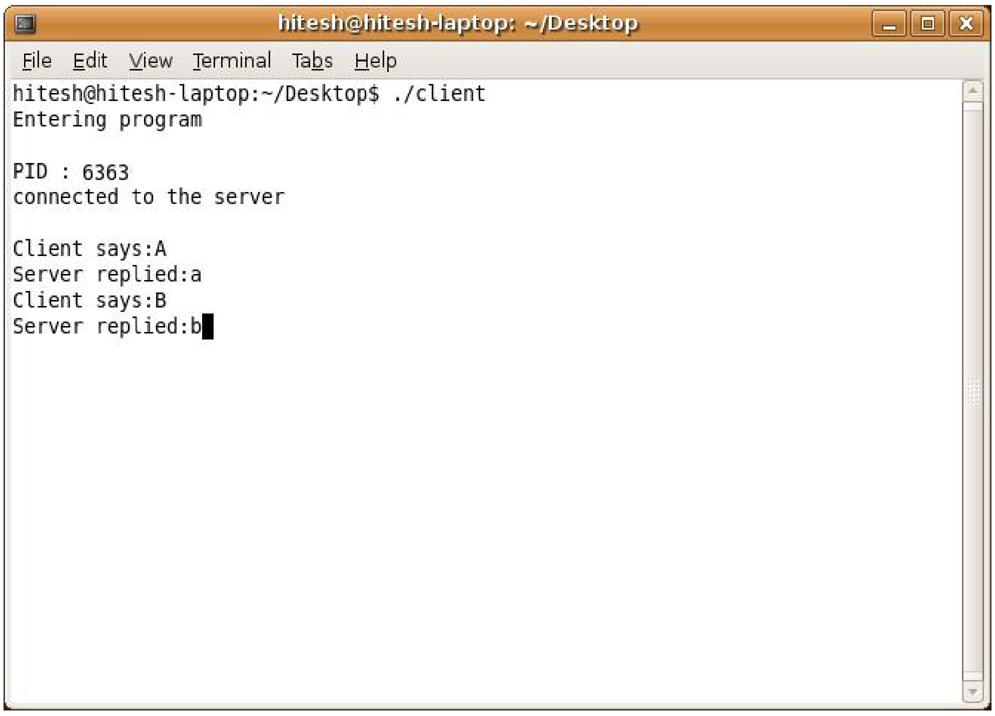
1) Start server and client programs that need to be frozen. A TCP connection gets established and data transfer takes place between the client and the server.



```
hitesh@hitesh-laptop:~$ cd Desktop/
hitesh@hitesh-laptop:~/Desktop$ ./server

i am listening in serverfork
Accepted incoming connection in serverfork
6364

Client says: A
Server replied:a
Client says: B
Server replied:b
```



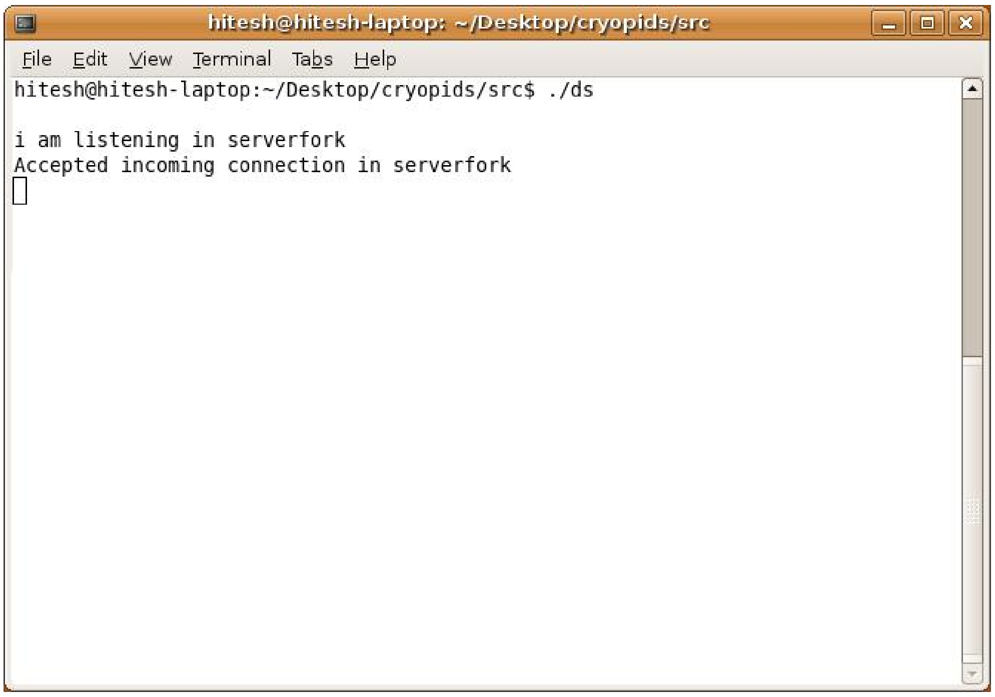
```
hitesh@hitesh-laptop:~/Desktop$ ./client
Entering program

PID : 6363
connected to the server

Client says:A
Server replied:a
Client says:B
Server replied:b
```

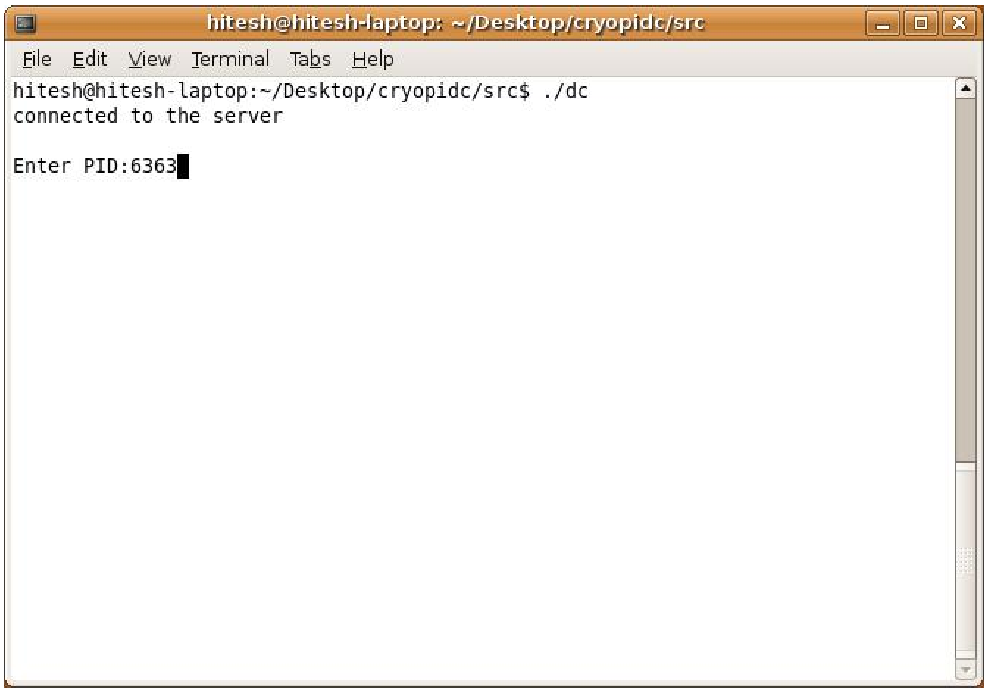
2) Get the PID of the client program that we need to freeze.

3) Start the daemon server and daemon client programs. A connection is established between them.


hitesh@hitesh-laptop: ~/Desktop/cryopids/src
File Edit View Terminal Tabs Help

```
hitesh@hitesh-laptop:~/Desktop/cryopids/src$ ./ds

i am listening in serverfork
Accepted incoming connection in serverfork
```

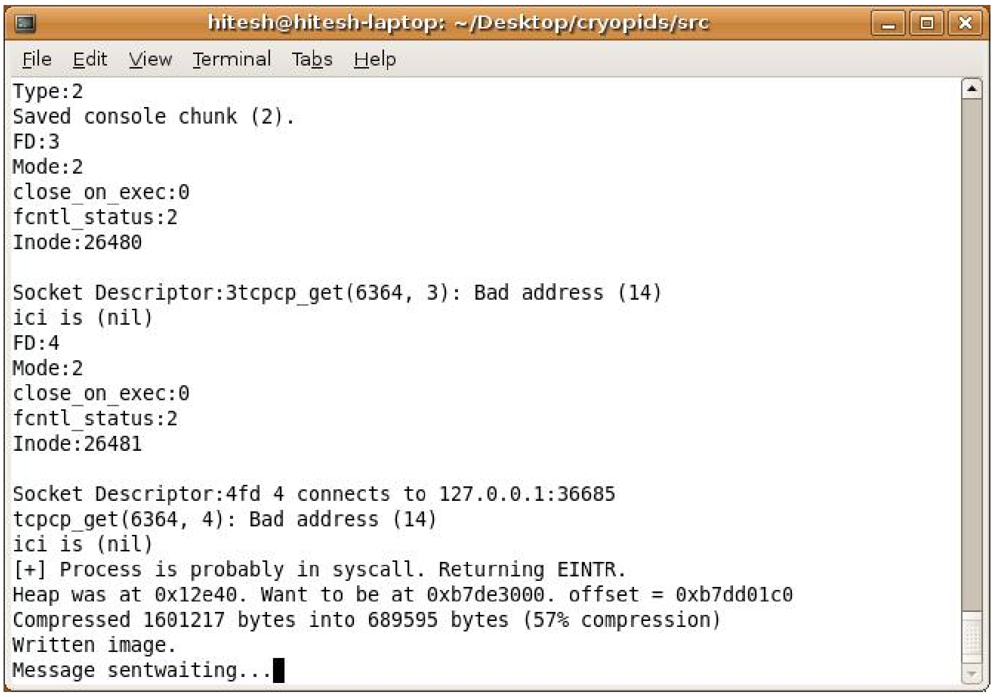

hitesh@hitesh-laptop: ~/Desktop/cryopidc/src
File Edit View Terminal Tabs Help

```
hitesh@hitesh-laptop:~/Desktop/cryopidc/src$ ./dc
connected to the server

Enter PID:6363
```

4) Enter the PID of the client that needs to be freezed in daemon client program.

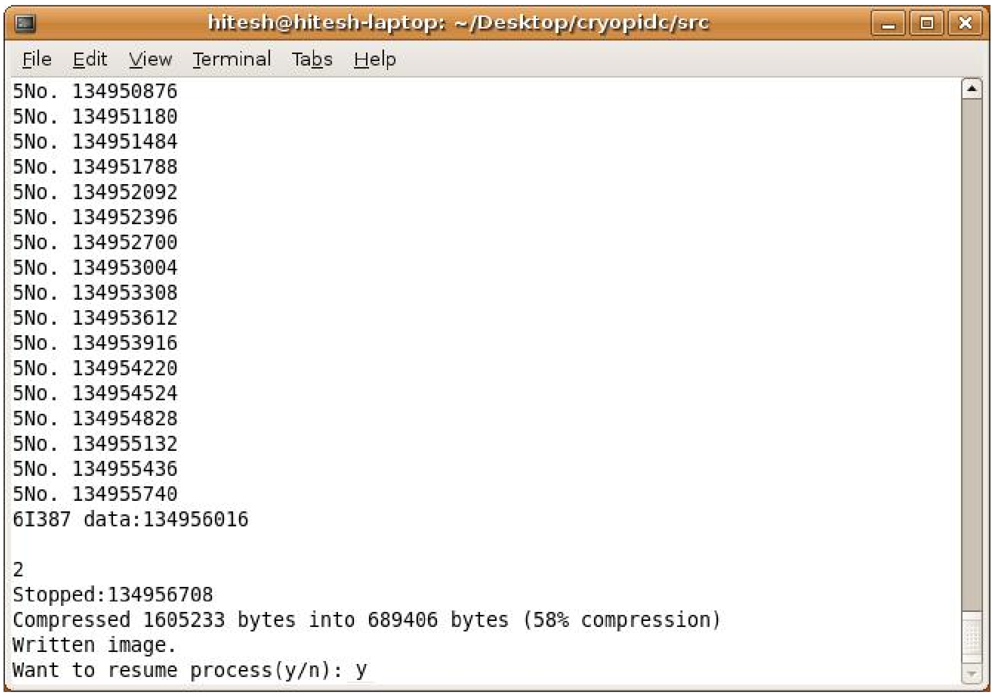
5) First the ELF of the corresponding servers child is created then the clients ELF i


hitesh@hitesh-laptop: ~/Desktop/cryopids/src
File Edit View Terminal Tabs Help

```
Type:2
Saved console chunk (2).
FD:3
Mode:2
close_on_exec:0
fcntl_status:2
Inode:26480

Socket Descriptor:3tcpcp_get(6364, 3): Bad address (14)
ici is (nil)
FD:4
Mode:2
close_on_exec:0
fcntl_status:2
Inode:26481

Socket Descriptor:4fd 4 connects to 127.0.0.1:36685
tcpcp_get(6364, 4): Bad address (14)
ici is (nil)
[+] Process is probably in syscall. Returning EINTR.
Heap was at 0x12e40. Want to be at 0xb7de3000. offset = 0xb7dd01c0
Compressed 1601217 bytes into 689595 bytes (57% compression)
Written image.
Message sentwaiting...
```

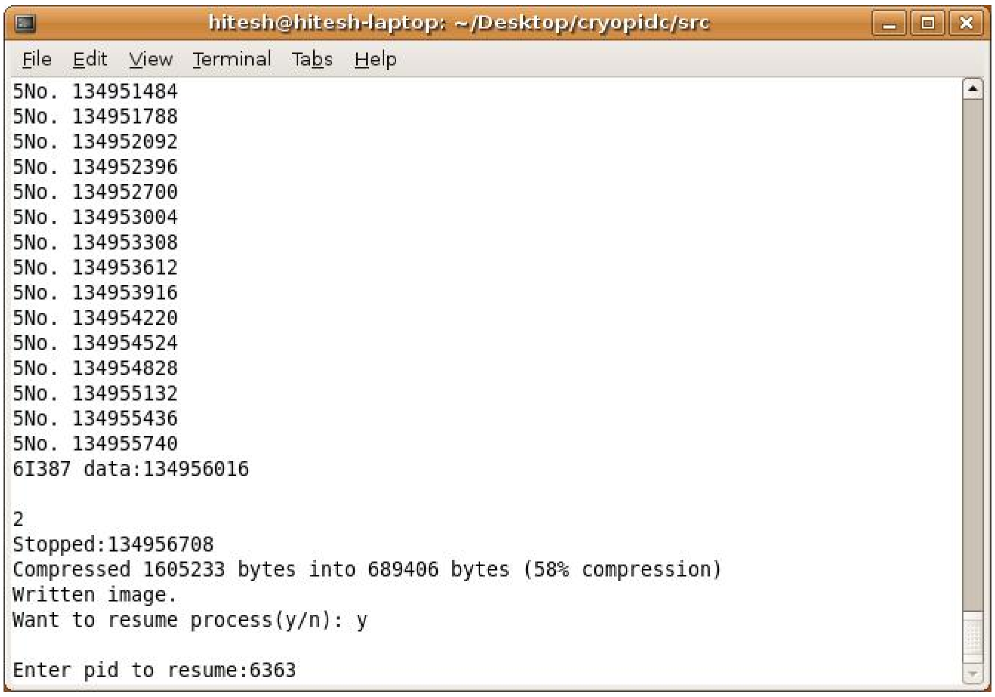

hitesh@hitesh-laptop: ~/Desktop/cryopidc/src
File Edit View Terminal Tabs Help

```
5No. 134950876
5No. 134951180
5No. 134951484
5No. 134951788
5No. 134952092
5No. 134952396
5No. 134952700
5No. 134953004
5No. 134953308
5No. 134953612
5No. 134953916
5No. 134954220
5No. 134954524
5No. 134954828
5No. 134955132
5No. 134955436
5No. 134955740
6I387 data:134956016

2
Stopped:134956708
Compressed 1605233 bytes into 689406 bytes (58% compression)
Written image.
Want to resume process(y/n): y
```

6) Enter if you want to freeze or resume the process. If resume enter Y.

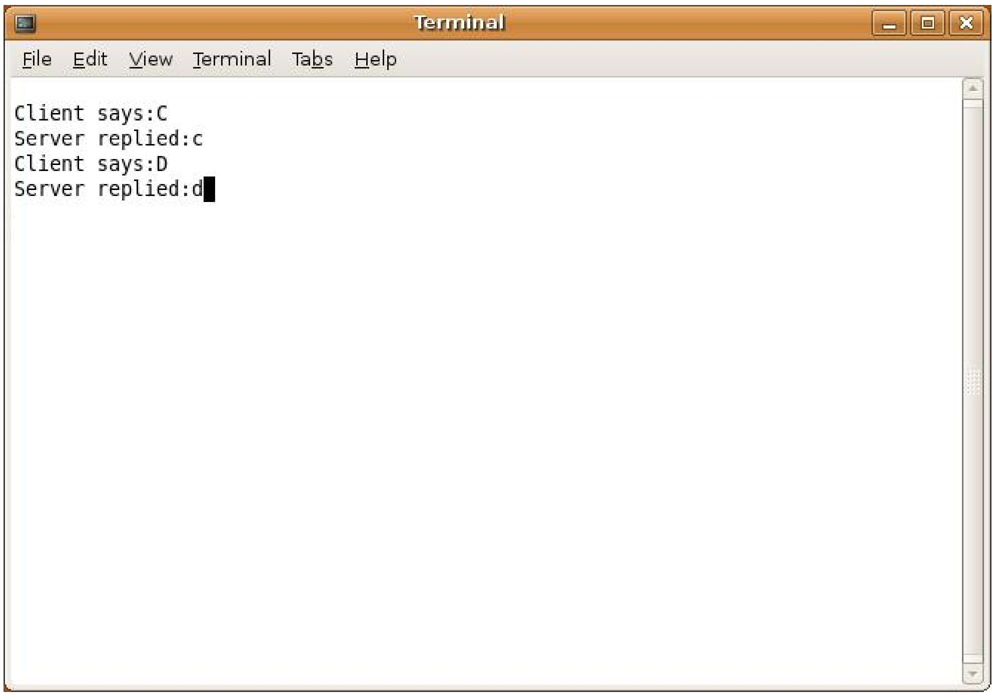
7) Enter the clients ELF name that is same as old PID in the client daemon. The client and the servers ELF are executed and processes are resumed on both sides.


hitesh@hitesh-laptop: ~/Desktop/cryopidc/src
File Edit View Terminal Tabs Help

```
5No. 134951484
5No. 134951788
5No. 134952092
5No. 134952396
5No. 134952700
5No. 134953004
5No. 134953308
5No. 134953612
5No. 134953916
5No. 134954220
5No. 134954524
5No. 134954828
5No. 134955132
5No. 134955436
5No. 134955740
6I387 data:134956016

2
Stopped:134956708
Compressed 1605233 bytes into 689406 bytes (58% compression)
Written image.
Want to resume process(y/n): y

Enter pid to resume:6363
```



```
Client says:C
Server replied:c
Client says:D
Server replied:d
```

8) The data transfer continues from the same point where it was freezed. Thus resume is successful.