TRIBHUVAN UNIVERSITY

INSTITUTE OF SCIENCE AND TECHNOLOGY

AMRIT SCIENCE CAMPUS

PROCESS HIBERNATION

By:

Bishnu Bidari (529/066)
Jagat Bahadur Shrestha (533/066)
Kedar Prasad Bhandari (534/066)

A PROJECT WAS SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE & INFORMATION TECHNOLOGY IN PARTIAL FULLFILLMENT OF THE REQUIREMENT FOR THE BACHELOR'S DEGREE IN COMPUTER SCIENCE AND INFORMATION TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND IT

LAINCHAUR, NEPAL

DECEMBER, 2013

TRIBHUVAN UNIVERSITY

INSTITUTE OF SCIENCE & TECHNOLOGY

AMRIT SCIENCE CAMPUS

DEPARTMENT OF COMPUTER SCIENCE & IT

Supervisor’s Recommendation

I hereby recommend that this project prepared under my supervision by Bishnu Bidari, Jagat Bahadur Shrestha & Kedar Prasad Bhandari entitled “**PROCESS HIBERNATION**” in partial fulfillment of the requirements for the degree of Bachelor of Science (B.Sc) in Computer Science and Information Technology be processed for the evaluation.

…....................................

SUPERVISOR,
Mr. Babu Ram Dawadi
Asst. Professor of IOE Pulchowk Campus
Visiting Lecturer at Amrit Science Campus
Tribhuvan University, Nepal

TRIBHUVAN UNIVERSITY

INSTITUTE OF SCIENCE & TECHNOLOGY

AMRIT SCIENCE CAMPUS

DEPARTMENT OF COMPUTER SCIENCE & IT

Letter Of Approval

The undersigned certify that they have read, and recommended to the institute of science and technology for acceptance, a project report entitled "**PROCESS HIBERNATION**" submitted by Bishnu Bidari, Jagat Bahadur Shrestha & Kedar Prasad Bhandari in partial fulfillment of the requirements for the Bachelor's degree in computer science and information technology.

....

Supervisor,
Mr. Babu Ram Dawadi
Asst. Professor of IOE Pulchowk Campus
Visiting Lecturer at Amrit Science Campus
Tribhuvan University, Nepal.

....

HOD/Coordinator,
Mr. Deepak Pudasaini
Asst. Professor of Amrit Science Campus
Tribhuvan University, Nepal.

....

External Examiner,

....

Internal Examiner,

DATE OF APPROVAL: / /

Abstract

This project work describes and implement some of the basic steps to achieve the goal of process hibernation on Linux. It should be clear here, hibernating a process is applicable only if it can be restarted from the point of stop at later favorable time in the same or different machine running Linux. During the problem identification phase of this project work, it has been discovered that, such tool is quite essential as the need of high processing capacity in the field of computing is increasing day by day. There are hundreds of scenarios where running a process, however single or multi-processor environment, for long period of time is crucial. Such includes the scientific research, high data processing, data mining and analysis etc.

At the beginning, the focus in this project, was to play with process beginning with a simple case: save and restore a single task, with simple memory layout, dis-regarding other task state such as files, signals etc. The checkpoints (a term used both for the act of saving state and the result) are created in the file system name space. Availability in the name space allows facilities to duplicate and transfer files to be applied; in this way replicated processes and process migration can be done rather naturally.

This documentation introduce two different methods of implementation for the purpose to achieve process hibernation and ultimately to achieve process migration. The first one is through kernel space, using Loadable Kernel Module (LKM), and the other one is through user-land, using /proc file system with PTRACE interface. The former one is more flexible option along with being tough to implement, whereas the another one is less flexible due the operations limited by the /proc file system.

Key-words: Process Hibernation, Process Migration, Checkpoint/restore.

Acknowledgement

We owe an enormous debt to many different people for helping us in the way of this project work. First of all we like to express our great gratitude towards our supervisor, Asst. professor **Mr. Babu Ram Dawadi** for helping and guiding us whenever we needed. Secondly, we like to thank our lecturer **Mr. Dilli Prasad Sharma** and **Mr. Bhoj Raj Ghimire** for their valuable suggestions and guidelines. Similarly, lecturer **Mr. Uttam Shrestha Rana** encouraged us with his kind help and moral support, we will be always grateful towards him.

Also, we like to thank our entire Computer Science and Information Technology (**CSIT) department**, especially Head Of Department, Asst. professor **Mr. Deepak Pudasaini** for the valuable encouragement and providing us such a nice academic environment.

We express our gratitude towards our **colleagues** and all our **family members** for their support and kind help even when we neglected responding them while we were busy doing our project work. Last but not the least, a BIG THANKS to Linux Torvalds (!) for making us Command-line User Interface (CUI) lover.

Table of Contents

List of figures

Abbreviations

CSIT	-	Computer Science and Information Technology
CPU	-	Central Processing Unit
LKM	-	Loadable Kernel Module
OS	-	Operating System
PCB	-	Process Control Block
PID	-	Process Identifier
VMA	-	Virtual Memory Area
CUI	-	Character User Interface

1. INTRODUCTION

There might be a scenario where, someone wants to capture the current snapshot of a process in a Linux machine after or before freezing the process. The frozen process is then can be loaded from the snapshot to resume. Such concept of process checkpointing can be very useful under a variety of circumstances. It can be used for process backup, live migration, faster boot-up service, etc.

An image is a description of a computation which can be executed by a computer. A process is an image in some state of execution. At any given time, the state of the process can be represented as two components: the initial state (the image) and the changes which have occurred due to execution. The total information, that is, the initial state together with the changes, gives us the state of a process. It may be desirable to preserve this state at certain points in time, due perhaps to the amount of computation required to reach that state. These points in time can be used for acts of saving state called checkpoints; this name is also used for the result of saving the state. The UNIX paradigm for manipulating objects is through the file system name space. The approach to accessing resources through name space entries has been applied to teletype devices, remote file systems, and system memory. The /proc virtual file-system attacked the problem of accessing process address spaces through name space entries in a distinguished directory /proc; the process objects were named by their process ids. Unfortunately these facilities did not provide complete file semantics; while entries of /proc could be read, analyzed, and modified, they could not be created; thus the interface could not use file system facilities for creating new processes.

1.1. Process creation

Operating systems need some ways to create processes. In a very simple system designed for running only a single application (e.g., the controller in a microwave oven), it may be possible to have all the processes that will ever be needed be present when the system comes up. In general-purpose systems, however, some way is needed to create and terminate processes as needed during operation.

There are four principal events that cause a process to be created:

1. System initialization.
2. Execution of process creation system call by running a process.
3. A user request to create a new process.
4. Initiation of a batch job.

When an operating system is booted, the first process “init” is created statically and all other process are created as child of init through system calls like fork() and clone() in association with family of exec() system calls.

The parent process is copied almost entirely, with changes only to the unique Process ID(PID), parent PID etc. Each new process gets its own user space and hence start executing as if independent. After calling to exec() family of system call the child process get replaced by the program given. And in this way a completely new process is created.

1.2. Detaching a processor from a process

If the processor is de-allocated during the execution of a process, it must be done in such a way that it can be restarted later as easily as possible.

There are two possible ways for an OS to regain control of the processor during a program’s execution in order for the OS to perform de-allocation or allocation:

1. The process issues a system call (sometimes called a software interrupt); for example, an I/O request occurs requesting to access a file on hard disk.
2. A hardware/ interrupt occurs; for example, a key was pressed on the keyboard, or a timer runs out (used in preemptive multitasking).

The stopping of one process and starting (or restarting) of another process is called a context switch or context change. In many modern operating systems, processes can consist of many sub-processes. This introduces the concept of a thread. A thread can be viewed as a sub-process; that is, a separate, independent sequence of execution within the code of one process. Threads are becoming increasingly important in the design of distributed and client–server systems and in software run on multi-processor systems.

1.3. Process model

The operating system's principal responsibility is in controlling the execution of processes. This includes determining the interleaving pattern for execution and allocation of resources to processes. One part of designing an OS is to describe the behavior that we would like each process to exhibit. Roughly processes are either RUNNING or NOT RUNNING. But there are some intermediate states which lies between them representing transitional phases. Which are READY, RUNNING, BLOCKED or SUSPENDED.

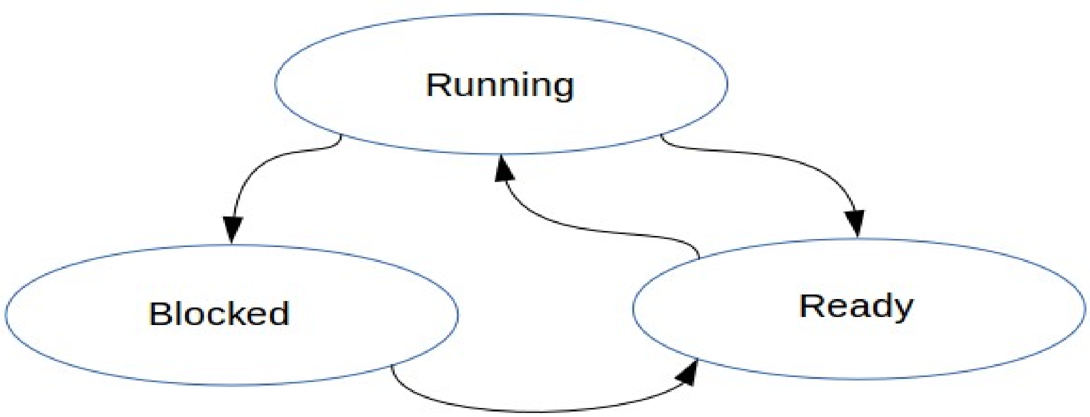


Fig. Process states.

1.4. Process description and control

Each process in the system is represented by a data structure called a Process Control Block

(PCB), or Process Descriptor in Linux, which performs the same function as a traveler's passport. The PCB contains the basic information about the job including:

-What it is ?

-How much of its processing has been completed ?

-Where it is stored ?

-How much it has “spent” in using resources ?

** **Process Identification**: Each process is uniquely identified by the user’s identification and a pointer connecting it to its descriptor.

** **Process Status**: This indicates the current status of the process; READY, RUNNING, BLOCKED, READY SUSPEND, BLOCKED SUSPEND.

** **Process State**: This contains all of the information needed to indicate the current state of the job.

** **Accounting**: This contains information used mainly for billing purposes and for performance measurement. It indicates what kind of resources the process has used and for how long.

1.4.1. Process descriptor

In order to manage processes, the kernel must have a clear picture of what each process is. It must know, for instance, the process's priority, whether it is running on the CPU or blocked on some event, what address space has been assigned to it, which files it is allowed to address, and so on. This is the role of the process descriptor , that is, of a task_struct structure whose fields contain all the information related to a single process. As the repository of so much information, the process descriptor is rather complex. Not only does it contain many fields itself, but some contain pointers to other data structures that, in turn, contain pointers to other structures.

There is some thread_info structure which contains the pointer to actual process descriptor table. The thread_info lies just above/below of kernel stack. So definitely thread_info is in main memory. But what about actual process descriptor task_struct? where is it located? If process descriptor resides in main memory, where is the actual place for it ?

How those structs are allocated depends on the architecture we're using. The relevant functions we want to examine are alloc_task_struct() and alloc_thread_info().

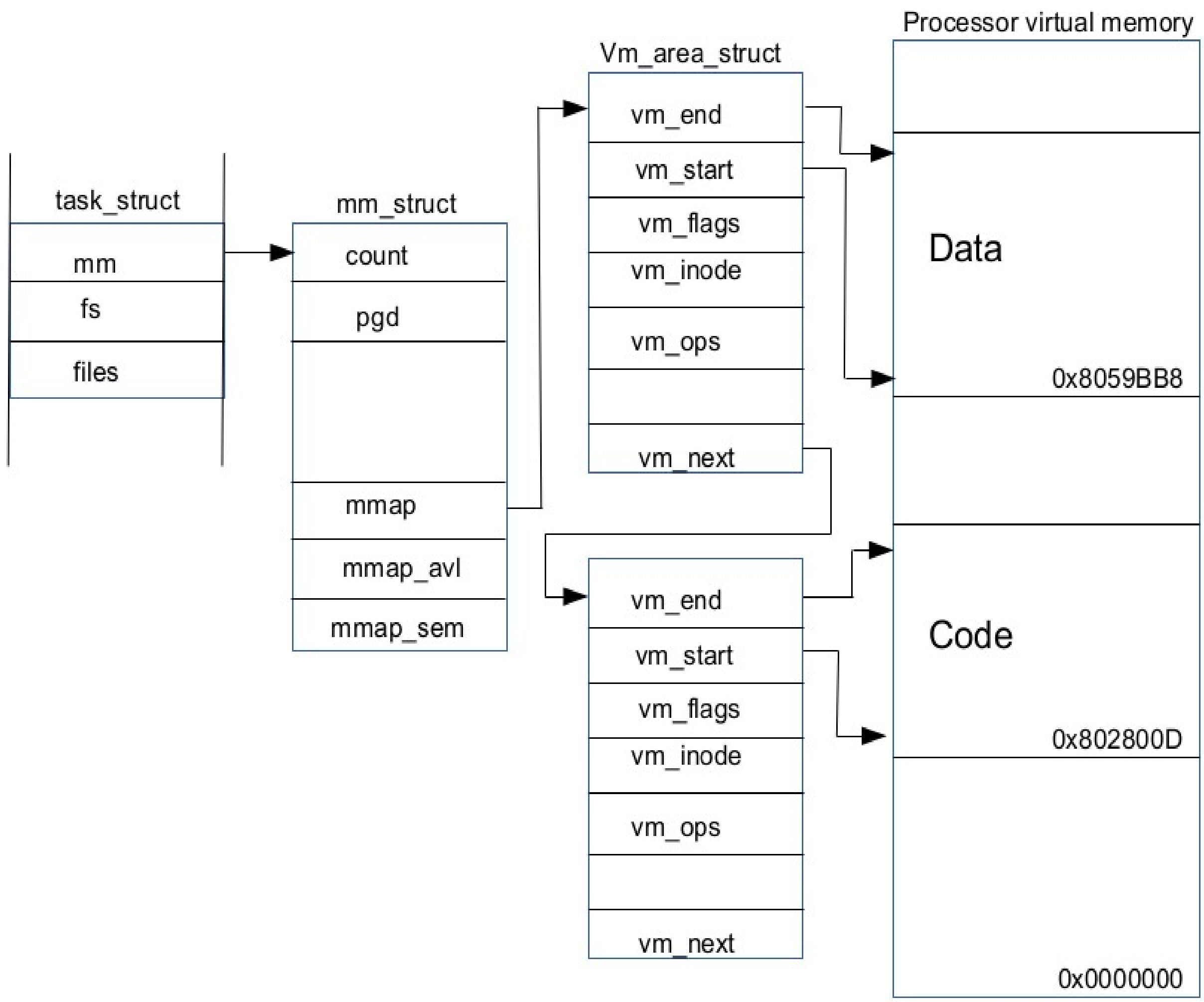


Fig. Structures in kernel-space to represent individual task (typical in linux).

1.5. Process switching

The task structure, which explains everything about a process, are linked to each other using pointer next and previous. As that forms a circular link list, all the processes rooted at init can be traversed. As the task structure contains information about task's state and priority Central Processing Unit (CPU) forms schedule accordingly. CPU divides its time to all the tasks on schedule queue.

In order to control the execution of processes, the kernel must be able to suspend the execution of the process running on the CPU and resume the execution of some other process previously suspended. This activity is called process switching , task switching, or context switching.

1.6. Checkpoint and Restore a process

Checkpoint and Restore is the ability to take a point-in-time snapshot of a running process (checkpoint), and revive it later, either on the same system or another system (restore). Here taking snapshot means taking all the information related to the current state of the process such as the task_struct, related mm_struct, and vma content. And restoring means forking some dummy process and restoring back the previously dumped information to that new process along with managing necessary pointers and fields.

1.7. Problems

At first blush, this sounds simple enough – dump the process' memory and stash it away, then later restore it and fix up a few references in the kernel, too easy! Not so fast there, there's a lot of subtle problems to be solved.

Early work in this project work took a very naive approach to the problem. As well as dumping memory, it made use of procfs, and syscalls to gather information from the kernel. File descriptors are a good example of this; for the process to keep running it needs all its file handles and sockets to be available when it's restored.

These informations, once collected and dumped into a file, in future can be massaged back into the kernel to perform a restore. For things that didn't fit these interfaces, some small patches to the kernel completed the functionality.

So how about PIDs? If we're wanting to migrate the process to another host, the PID had better be unused on the destination system. A process' PID can't change, as it has a parent and child relationship with other processes, which generally rely on having the PID. Consider also that some daemons write a pid-file to aid other processes in interacting with them.

Some parts of a process' state isn't even directly tied to the process itself in the kernel. Examples of this are outstanding signals and various buffers (eg. network sockets that haven't been read yet). These all need to be tracked down, stashed away, and carefully setup again to prevent any nasty surprises when the process is woken up. Code was written to allow extended peeking on sockets, so they can be inspected and replicated.

What about interprocess matters? Pipes are a common IPC technique, and they need to be correctly plumbed on the destination.

1.8. Process representation in Linux

The process control block, PCB in the Linux operating system is represented by the C structure task_struct. This structure contains all the necessary information for representing a process, including the state of the process, scheduling and memory-management information, list of open files, and pointers to the process's parent and any of its children. Some of the fields include :

```
struct task_struct {
        volatile long state;
        /*-1 unrunnable, 0 runnable, >0 stopped */
        void *stack;
        atomic_t usage;
        unsigned int flags;
        /* per process flags, defined below */
        … … …
        pid_t pid; /* process id*/
        pid_t tgid; /* task group id */
        … … …
        struct task_struct __rcu *real_parent;
        /*real parent process */
        struct task_struct __rcu *parent;
        /* recipient of SIGCHLD,wait4() reports */
        struct list_head children; /* list of my children */
        struct list_head sibling;/*linkage in my sibling list */
        … … …
        struct fs_struct *fs;  /* filesystem information */
        struct files_struct *files; /* open file information */
        struct mm_struct *mm, *active_mm;
        /*pointer to mm_struct which holds info about virtual
          memory being used */
        … … …
};
```

For example, the state of a process is represented by the field long state in this structure. Within the Linux kernel, all active processes are represented using a doubly linked list of task_struct, and the kernel maintains a pointer –current-- to the process currently executing on the system.

As an illustration of how the kernel might manipulate one of the field in the task_struct for a specified process, let's assume the system would like to change the state of the process currently running to the value new_state. If current is a pointer to the process currently executing, its state is changed with the following :

```
current->state = new_state;
```

1.9. Process migration

Process migration is the transfer of some (significant) subset of the informations to another location, so that an ongoing computation can be correctly continued. Talking in our sense it mens to transfer the previously dumped process snapshot to another machine and restart that process there using that snapshot. Process migration is most interesting in systems where the involved processors do not share main memory, as otherwise the state transfer is trivial, as it can be accomplished with pointer relocation. A typical environment where process migration is interesting is autonomous computers connected by a network. The message-passing systems ease implementation, and the state-full nature of most operating system kernels is an impediment to migrating processes. In any case, these process migration mechanisms demonstrate that the state of an executing process can be moved between homogeneous machines, and that the execution can be continued. This transfer of address spaces is what intrigues us.

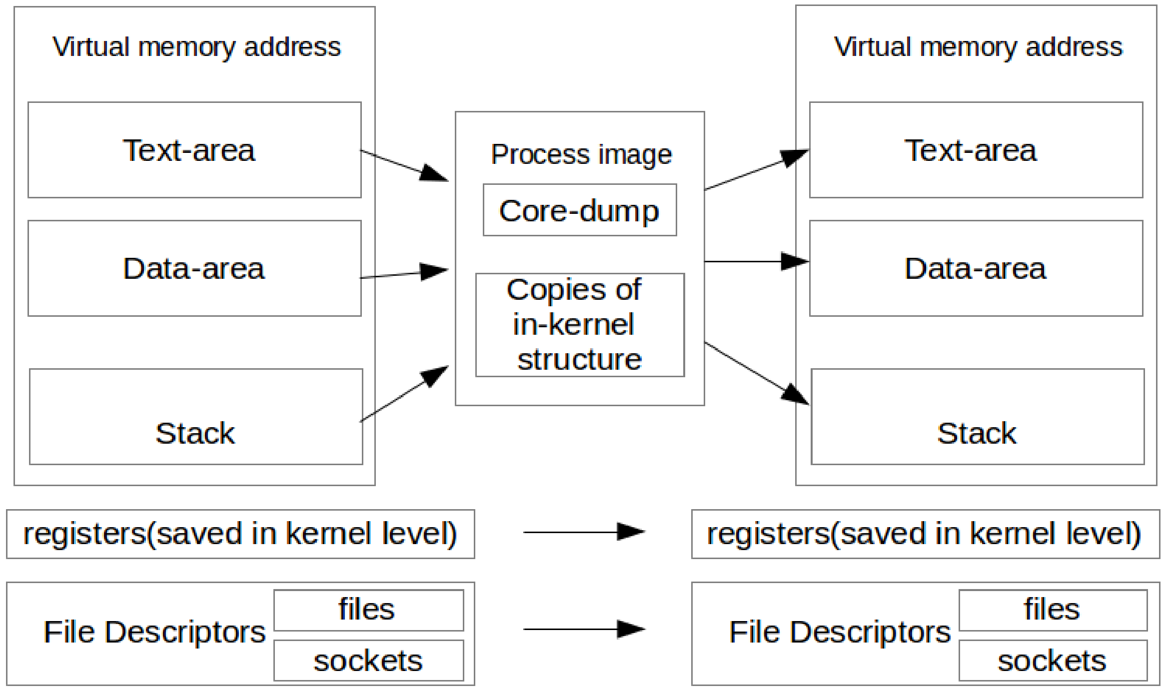


Fig. Process hibernation overview.

2. DESIGN GOALS

1. Built with existing operating systems

The system should work on general-purpose operating systems. The intention was not to write a new operating system, but wanted to leverage investments already made in existing systems. The Linux operating system was chosen as the implementation platform because it is rapidly growing, totally free, and has source code available.

2. Transparent to user applications

The package should be general-purpose and able to checkpoint legacy applications, so this forces to do the implementation in the kernel space.

3. No kernel modification

By using Loadable Kernel Module, LKM, virtually the same level of transparency as kernel patches was achieved but avoided changing the kernel itself. This makes it much easier to use.

4. Good performance

The performance of existing systems should not be degraded. Especially, it should not add any run-time overhead to the system other than the actual checkpoint and restart.

3. ASSUMPTIONS

The basic assumptions of this implementation are:

1. The operating system must support LKM. Kernel module is defined as a program developed separately from kernel that can be loaded and run int the privileged mode.
Many operating system support modules, Linux is one.

2. Process private data, such as address space, registers set, opened files, are accessible and

restorable from the kernel module.

3. Assume a homogeneous environment. All machines should have the same architecture and OS (Linux).

4. Assume the process can continue to access the same files on all machines. The files are either on a global file system (such as NFS) or installed locally (such as /bin, /lib).

4. USER INTERFACE

4.1. Checkpoint

The user should be able to specify which process to checkpoint and where to send the checkpointed image (disk or else). The user should also be able to control the checkpointing behavior, such as whether to kill the process after the checkpoint.

For example:

```
# hibernate <PID> -d outfile.dmp
```

The hibernate program when supplied with pid of the process to be dumped along with '-d' switch will dump the process image to output file 'outfile.dmp'.

4.2. Restart

The user should be able to pass a checkpointed image and restart the previous process from it.

For example:

```
#hibernate <PID> -r outfile.dmp
```

For restarting the process dumped on 'outfile.dmp', the hibernate program should be provided the pid of process along with '-r' switch which will be replaced by the image on 'outfile.dmp'. An automatic process migration system can be built on top of the checkpoint/restart capability and do things like load-balancing without human intervention.

5. EXECUTION FLOW-MODEL

5.1. Dumping the process

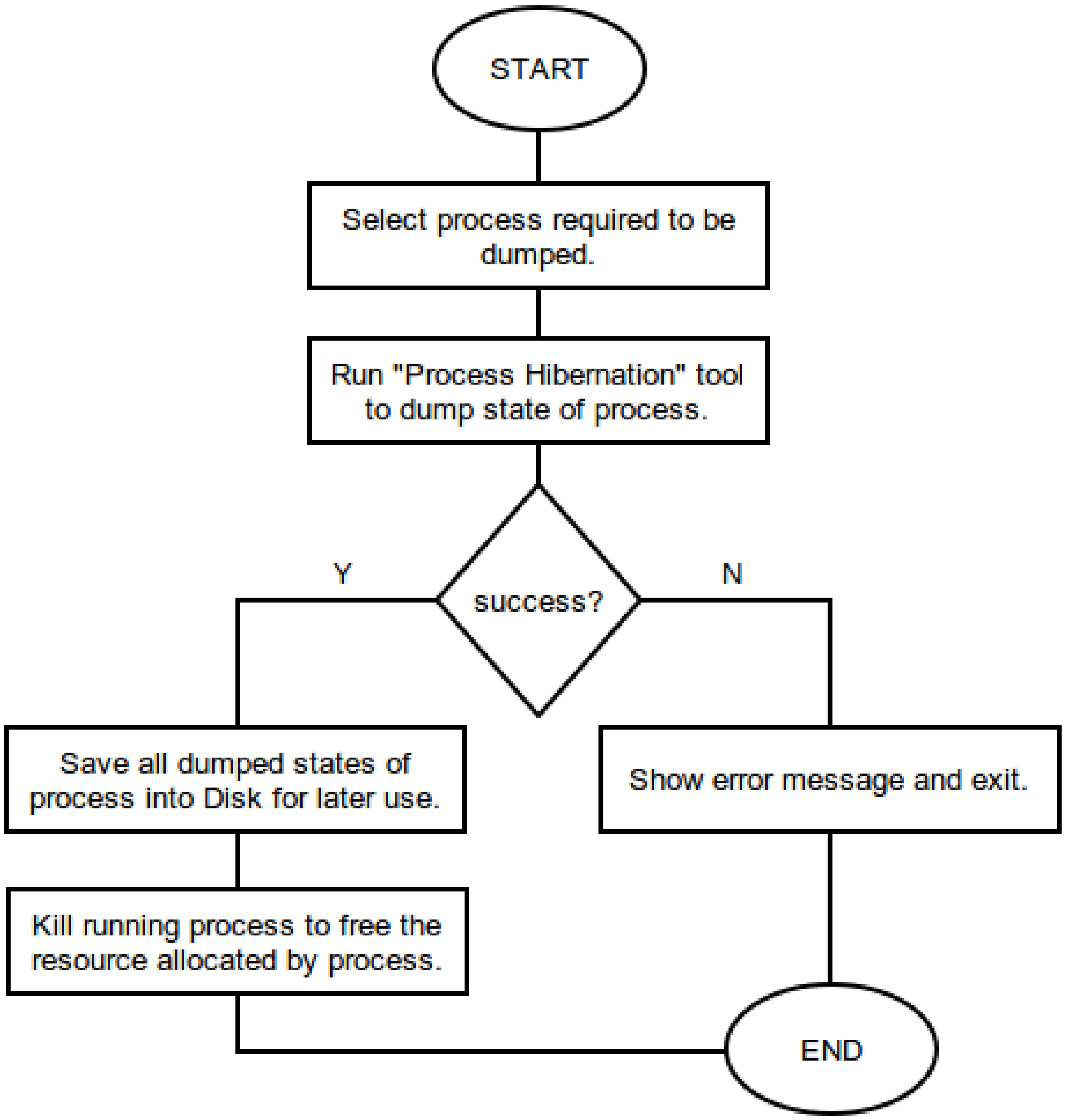


Fig. Flow-model process dumping.

5.2. Restoring the dumped process

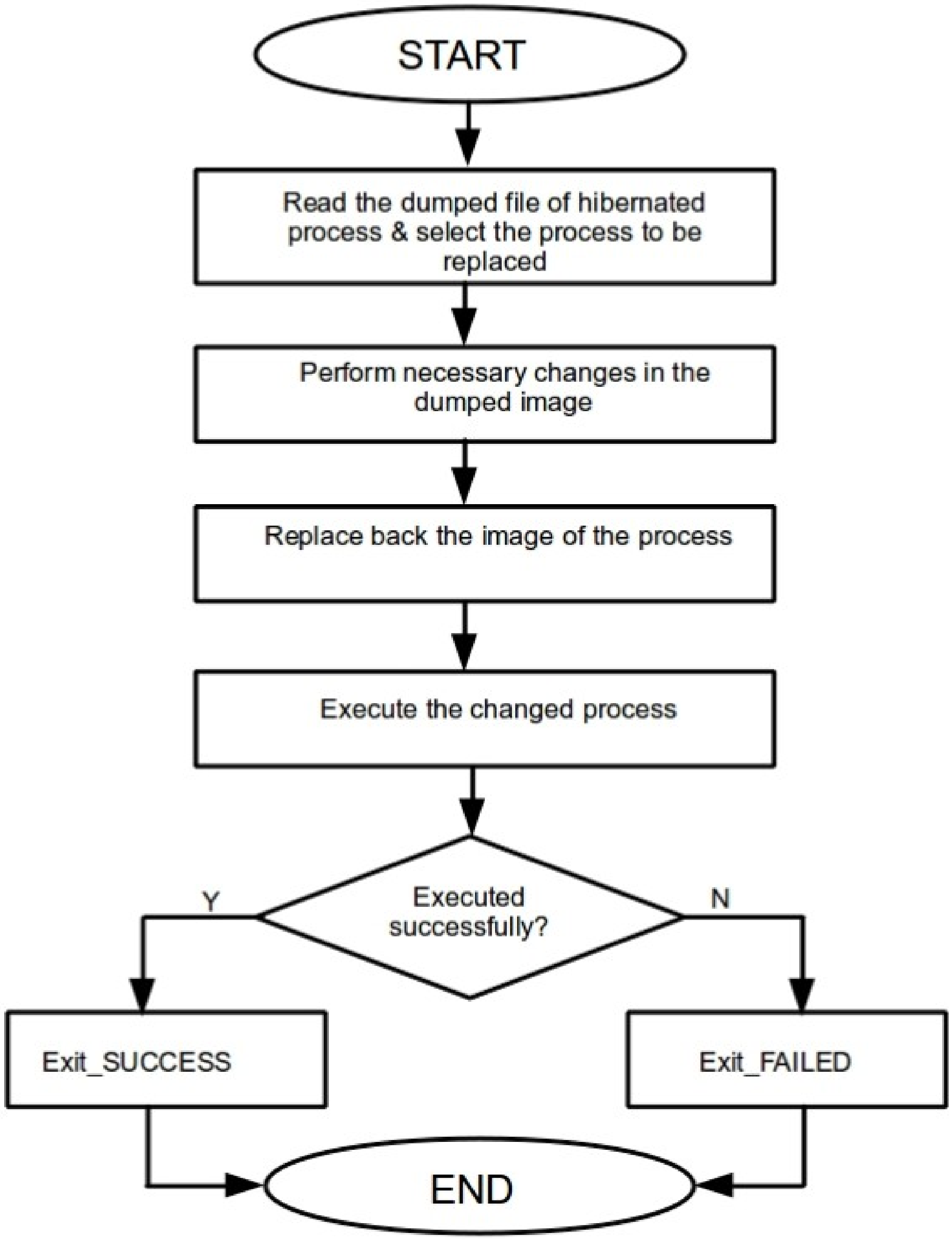


Fig. Flow-model process restoring.

6. SCOPES

This "**process hibernation**" tool is applicable to hibernate specific process which is running in system. It is applicable to all the computer users from general to advanced in-order to hibernate their program as needed. In general convention normal users do not use all the resource of computer so here is some specific field for which it is more applicable:

→ For system administrator to study large volume of system logs and maintaining the system.
→ Security analyst who need to run their program for month in-order to decrypt some encrypted data from suspects' computer.
→ From its high end applicability, the proposed tool is useful to researchers, students and engineers.

7. IMPLEMENTATION DETAILS

A process's address space typically has the layout as shown in figure below. The stack segment is at the numerically higher addresses, while the text segment is at the numerically lower addresses. The address space, objects referenced by descriptors in the address space (e.g., open files), and system state (e.g., virtual-to-physical address mappings) comprise the state of a process.

Address mappings and similar state informations are transparent to the processes. Then how could one process be dumped from another process? Either using kernel-space code i.e. LKMs which has the required privileges to access the task_struct of any process or using provided user-land interfaces such as PTRACE, which can attach to another process and make that process as its child, hence enabling it to peek / poke the data in it.

To meet the goal of hibernation one could use either of the ways described above. At the highest level of abstraction, it is desirable in followings:

1. Checkpoint the process.
2. Perform necessary changes in the image.
3. Restart, i.e. replace some dummy process with the image.

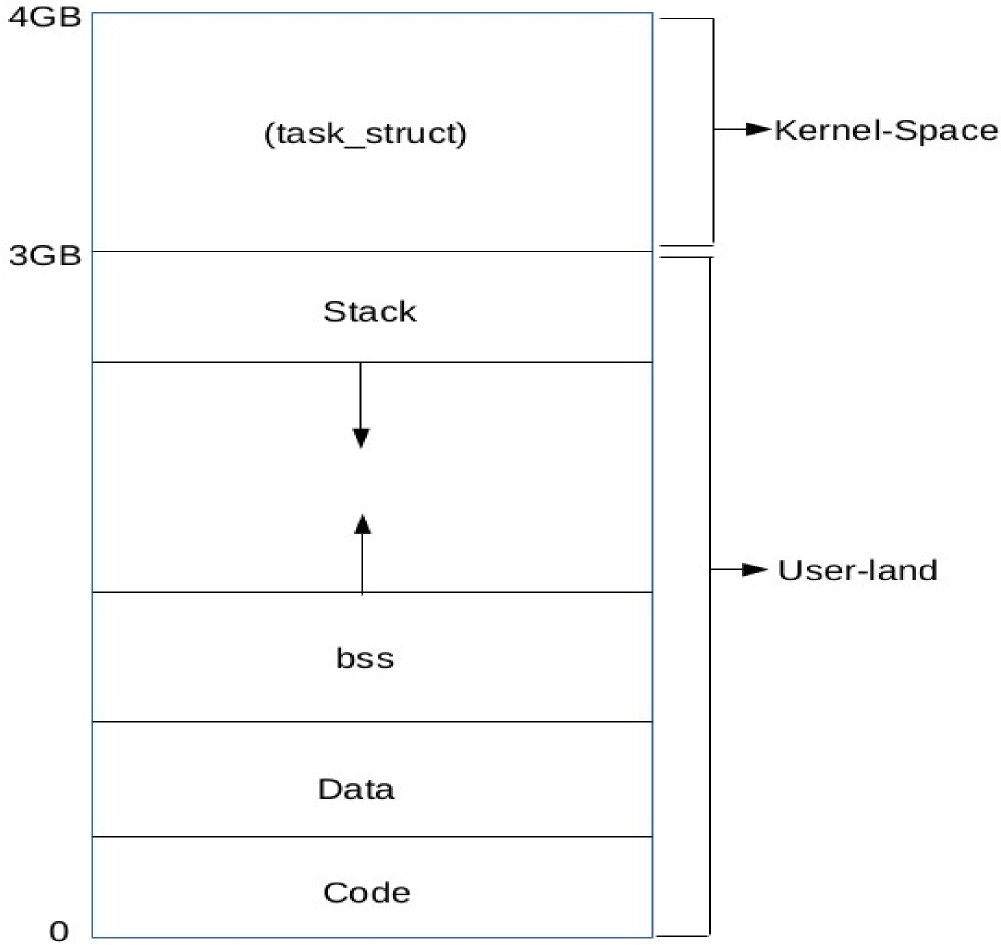


Fig. Process Virtual memory structure (typically on Linux-32bit machine).

7.1. Hibernating the process

It has been tried and recommended two approaches in order to dump the process running in the machine.

7.1.1. Using Kernel Module

As Kernel Module runs with higher privileges, it can see all the task_struct. And following the pointers provided there, the respective virtual memory area could be reached. But before that it is needed a way to execute code in kernel space. The only way to run code on kernel space from userland is through system-calls. But syscall_table is not exported in kernel since 2.6.x for preventing the system-call interception.

The other approach to take was the exploitation of the device file. In fact a character device file was created and wrote an LKM as driver to that device. Executing write() on that device file calls respective function written on that LKM, hence the kernel-space code can be executed there.

For example:

```
$ echo '<pid>' >> /dev/dev_file
```

As echo calls write to /dev/dev_file and that ultimately calls the device_write() function from loaded module. Similarly, various operations can be performed with kernel-space privileges. It can be used for printing that task_struct structure or for setting certain fields of it according to our needs.

One of such use is to make a process with non-root privileges to root. For this purpose, few lines of code was written in the device_write() function of the LKM to:

→ read pid from user-land

→ locate the task_struct

→ modify the user credential to make it as root by setting

```
    task_struct->real_cred->uid = 0;
    task_struct->real_cred->gid= 0;
```

(code can be found on Apendex I)

Likewise, when task_struct->files (files_struct) of two different processes, swaps the files associated with each processes like stdin, stdout etc. which are the standard input and output devices represented in files (linux specific). (AppendixII explains about it in detail).

As writing LKM allowed kernel-space codes to run, it is supposed that the list of virtual memory area pointed by task_struct->mm->vm_area can be printed and hence the respective data stored in the segments such as text_segment, data_segment, stack segments can be read out. If that comes in to reality, the process dumping phase is now completed. But due to lack of intense knowledge of the locking/accessing mechanisms of VMA, the implementation was stopped there.

7.1.2. Using user-land interface

After that the failure to move further in LKM, the focus was shifted to use user-land kernel interfaces like PTRACE and /proc/<pid>/mem file. PTRACE is a system call which can be used to read/write to process virtual memory. It in-fact attach with the process to be traced and become temporary parent. After that PTRACE can be used int the following way:

```
→ ptrace(PTRACE_ATTACH, pid, NULL, NULL);    /* start tracing
Process  pid */
→ ptrace(PTRACE_PEEKDATA, pid, &addr, &buf);      /* read data
from virtual memory of 'pid' at
               address 'addr' and put it on buf */
→ ptrace(PTRACE_POKEDATA, pid, &addr, &buf); /* write buf to
addr on the virtual memory
          of 'pid' */
→ ptrace(PTRACE_DETACH, pid, NULL, NULL);    /* stop tracing */
```

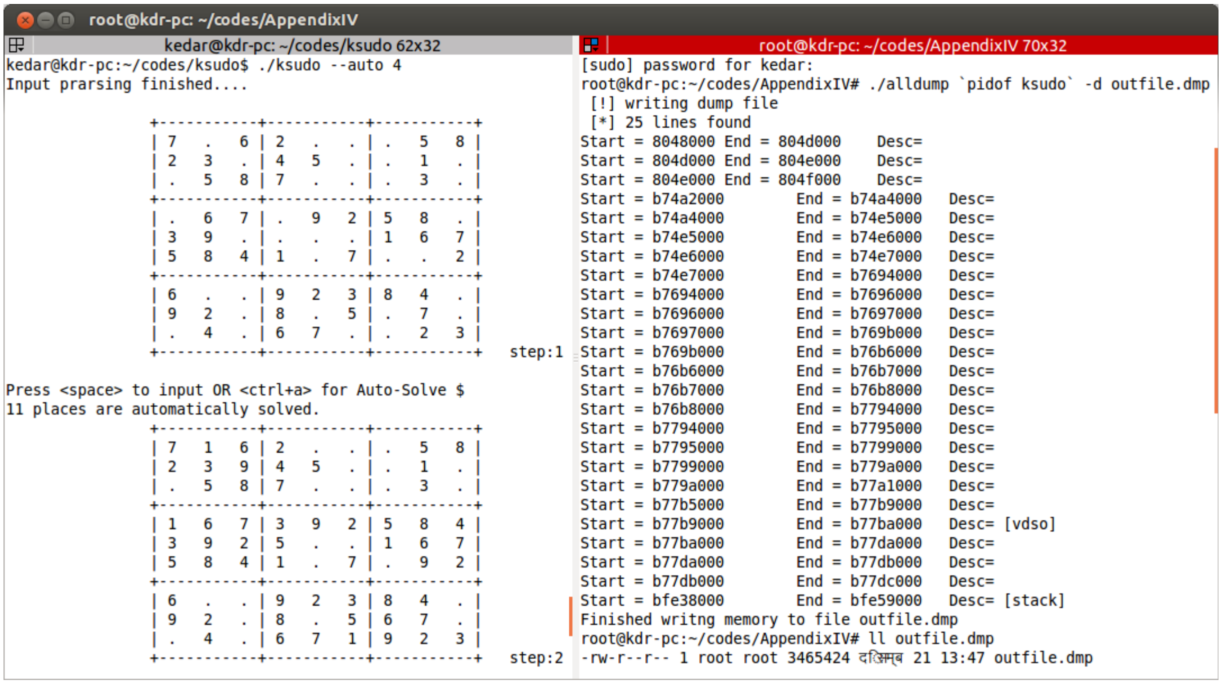
Using these functions the data/code hold by the process can be changed. Through this the variable data, can be changed like on some game cheater programs. (The code in Appendix III demonstrates the same).

Similarly procfs, the /proc/<pid>/mem, can be read/write using super-user privilege. Such file in fact have back end supported by kernel module, which respond to read and write request by calling respective kernel-space code. /proc/<pid>/maps file is another helper file which lists the virtual address range for various process segments. Hence, using those two files we can scan the code segment for our desired code, like the string being printed, and change them to something that interests (code at Appendix IV). And in the same way the stack, which holds information about local variable and there values, can be read and written back too. But writing back and starting that manipulated program correctly requires careful alternation and deep understanding of the data-structure there.

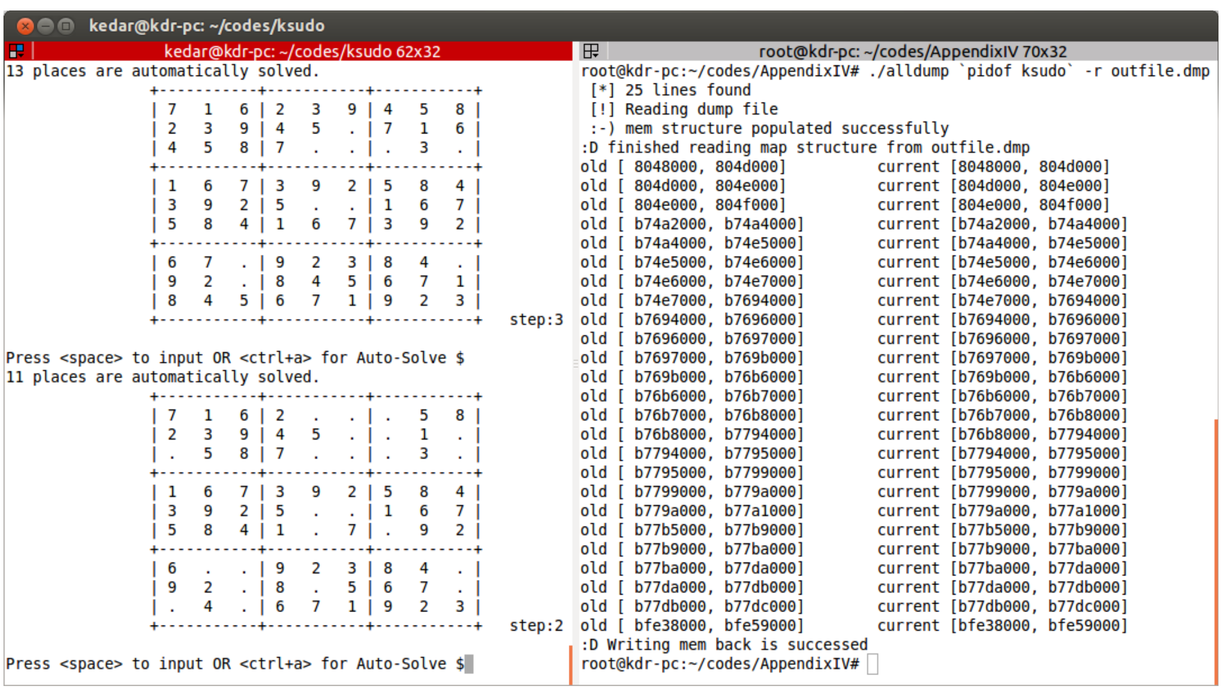
Taking a novice approach the whole stack segment was copied and replaced it back as a whole without any modification which in fact made the process rollback to the state that stack was copied. Yes this is the main success of this project work. But Every program restart result on somewhat change on the address space, virtual address in which code, stack, heap are located, and hence the pointer relocation become mandatory for the process to resume without any Segmentation Fault.

Dumping stack segment as it is from the mem file and later replacing back with out any modification made the program travel to its past. (Code can be found on Appendix V).

Below two snaps shows the use of the program in Appendix V to checkpoint and restore the executing sudoku solver program. The sudoku solver doesn't has built in mechanism to undo the changes but that facility can be achieved by checkpoint/restore in the memory.



Screen-1: Dumping the state of Sudoku Solver.



Screen-2: Restoring the Sudoku Solver to previously dumped state.

The mem file have limitation that no user-land process can lseek to arbitary point and write data except on that area where there exist some data previously. Because lseek-ing to some arbitrary area on mem file and writing data there may demand for adding new VMA to process's task_struct which is not handled by the procfs interface.

Hence, this shows two ways to make the process hibernation become reality. The first one is by using appropriate VMA accessing mechanism in kernel-space code. And the next one is by understanding the data-structure holding stack segment and then relocate the pointers there before writing to mem file.

8. DISCUSSION

Though above discussion of hibernation is conceptually straightforward, it is difficult in implementation. The difficulties lie in the following aspects. First, the source code of the kernel is very complicated due to both the complexity of a system and the aggressive optimization. Even though the information of a process is mainly stored in the task_struct, the structure itself is an extremely large monster. Also, checkpoint/restart-ing the process touches almost every part of the kernel implementation, including CPU scheduling, memory management, file system and so on. Thus it requires high familiarity with the kernel implementation and is also highly error prone.

However, it still has two draw backs. The first problem is that it is not so elegant in implementation. For the hibernation of a whole virtual machine, we just bitwise copy the suspended machine image, but process level hibernation requires dealing with a lot of system details.

The second problem is that it is still low-efficient. That is mainly because at least all the memory allocated for that process should be dumped and transfered. And it does take quite a lot of computing time, storage space.

As a result, it is not suitable to use the hibernation in load balancing. But it still has some applications in distributed computing, especially for scientific computing that need to be run for a significantly long time. It is quite possible that the computation would have to be paused due to some reason like system maintenance and power failure, and the hibernation can make the computation resist to that kind of system pause, without having to start all over again.

9. FUTURE WORKS

This project is successful to rollback any simple, CUI process on Linux machine. It does so by using the relatively less-flexible /proc virtual file-system interface. Being based on this project documentation and codes given on appendix, further study and work can be done in order to make process hibernation possible.

As this project tries to present two approaches of process hibernation, following two task are worth doing while trying process hibernation.

→ If the kernel module approach is used, the accessing mechanisms of the VMA need to be understood clearly.

→ If the user-land kernel interfaces (/proc/ or PTRACE) is to be used, the pointer relocation with appropriate offset must be done.

10. CONCLUSION

Process hibernation has long been studied. It approaches two different methods. The code on Appendix-V of this project get succeeded to make process rollback to it's past using the user-land kernel interface (/proc virtual file-system). This can be used to give any program with no built-in undo feature to have its state saved on dump and later undo to that state by replacing process image by that dump.

However, this project is not yet a mature job, so we still need further work to make it practically useful. With the result of this small implementation,it is believed that further study and implementation can be made to meet the ultimate goal of process hibernation.

11. APPENDICS

Appendix I: Char device exploitation using LKM.

Following program is a kernel module which can be compiled with the “Makefile” given below. That “make” will result on some files along with mk_root.ko . That mk_root.ko is the kernel module, which can be inserted into kernel using insmod command with super-user privilege. As this kernel module is driver to a character device /dev/devroot, that character device need to be created.

```
/*
* mk_root.c :
* a kernel module to give non-root shell the root privilege
*/
// Apendicx II

#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/fs.h>          /* for file structure */
#include <asm/uaccess.h>       /* for put_user */
#include <linux/slab.h>        /* for kmalloc */
#include <linux/sched.h>       /* for task_struct */
#include <linux/cred.h>        /* for user_credientials */

int init_module(void);
void cleanup_module(void);
static int device_open(struct inode *, struct file *);
static int device_release(struct inode *, struct file *);
static ssize_t device_write(struct file *, const char *,
size_t, loff_t *);
```

```
#define SUCCESS 0
#define DEVICE_NAME "devroot"      /* name that appears in
/proc/devices */
#define BUF_LEN 80       /* Max len of msg from teh device */

static int Major;        /* Major number assigned to our device
driver */
static int Device_Open = 0;   /* Is device open ?  used to
prevent multiple access to device */
static char *msg_Ptr=NULL;

static struct file_operations fops = {
.write = device_write,
.open = device_open,
.release = device_release
};

void print_cred(struct cred mycred ) {
printk("====== Printing cred =====\n");
printk(" uid = %d \n gid = %d\n", mycred.uid, mycred.gid);
printk(" suid = %d\n sgid = %d\n", mycred.suid, mycred.sgid);
printk(" euid = %d\n egid = %d\n", mycred.euid, mycred.egid);
printk(" fsuid = %d\n fsgid = %d\n", mycred.fsuid,
mycred.fsgid);
}

int root_me(int pid) {
     struct task_struct *task;
```

```
	struct cred *new_cred;

	printk("Lets search that pid supplied\n");
	for_each_process(task) {
		if(task->pid == pid)
			break;
	}
	printk("The task to be given root is  %s:%d\n",task->comm,
task->pid);
	new_cred = (struct cred *)kmalloc(sizeof(struct cred),
GFP_KERNEL);

	memcpy(new_cred, task->real_cred, sizeof(struct cred));

	new_cred->uid = 0;// making it root
	new_cred->gid = 0;
	new_cred->suid = 0;
	new_cred->sgid = 0;
	new_cred->euid = 0;
	new_cred->egid = 0;
	new_cred->fsuid = 0;
	new_cred->fsgid = 0;

	print_cred(*new_cred);
	print_cred(*(task->real_cred));

	memcpy(task->real_cred, new_cred, sizeof(struct cred));

	return 0;
}
```

```
 int init_module(void)
{
     Major = register_chrdev(0,DEVICE_NAME, &fops);

     if(Major < 0) {
          printk(KERN_ALERT "Registering char device failed
with %d\n",Major);
          return Major;
     }

     printk(KERN_INFO "Create a dev file with \n");
     printk(KERN_INFO "'mknod /dev/%s c %d 0'.\n", DEVICE_NAME,
Major);

     return SUCCESS;
}

void cleanup_module(void)
{
 unregister_chrdev(Major, DEVICE_NAME);
}

static int device_open(struct inode *inode, struct file *file)
{
     if(Device_Open)
          return -EBUSY;
     Device_Open++;

     try_module_get(THIS_MODULE);
```

```
    return SUCCESS;
}

static int device_release(struct inode *inode, struct file
*file)
{
    Device_Open--; /* we're now ready for our next caller */
    module_put(THIS_MODULE);
    return 0;
}

static ssize_t device_write(struct file *filep,
                   const char *buff,
                   size_t len,
                   loff_t *off)
{
ssize_t ret;
int pid;

msg_Ptr = (char *) kmalloc(len, GFP_KERNEL);
ret = copy_from_user( msg_Ptr, buff, len);
msg_Ptr[len]= '\0';
sscanf(msg_Ptr,"%d",&pid);

root_me(pid);
return len;
}
```

Above program can be compiled with the following “Makefile” which will generate mk_root.ko (kernel object) file along with others.

```
/*
* Makefile
* /
obj-m += mk_root.o

all:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD)
modules
clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD)
clean
install:
	sudo insmod ./mk_root.ko
	sudo mknod /dev/dev_root c 250 0
```

Compile the mk_root.c by issuing “make” command and then load the kernel module with “make install”. Once the kernel module is loaded, echo “<pid_of target>” >/dev/devroot will make that target process to run with root privileges. Output is shown by the following snapshot.

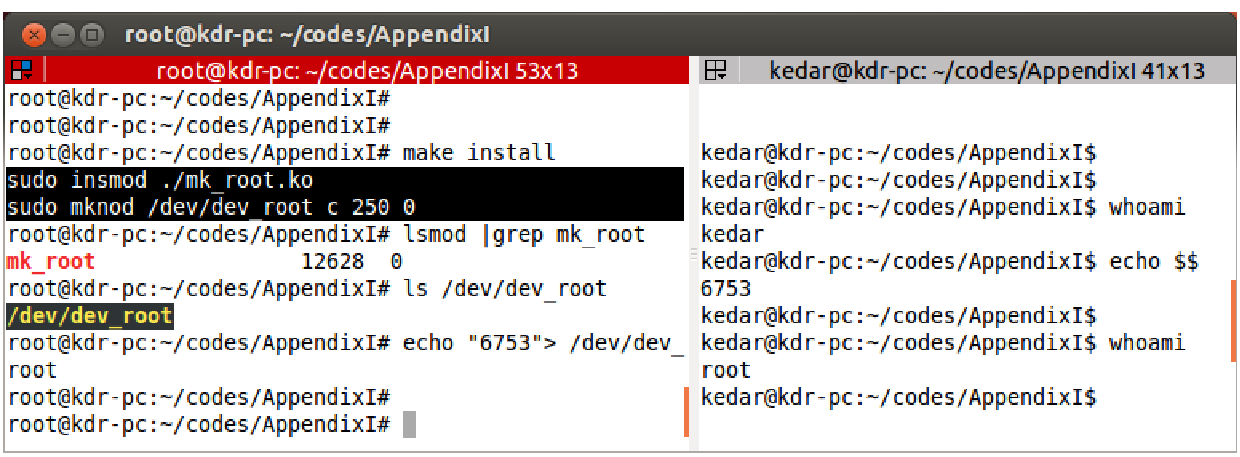


Fig. Char device exploitation using LKM.

Appendix II: Swapping file_struct of two different processes

Following is an LKM which when insmod'ed with two arguments, that is pid's of two different processes, swaps the files_struct of them. As files_struct contains informations about all opened files, swapping them will swaps the stdout as well. Hence the output is visible through each others execution space i.e. the shell.

/*

**** swap_processes.c***

****/***

```
#include <linux/module.h>

#include <linux/sched.h>

#include <linux/cred.h>

#include <linux/slab.h>

#include <linux/file.h>

#include <linux/fdtable.h>

#include <linux/fs_struct.h>

#include <linux/types.h>


static int pid1;

static int pid2;

static struct task_struct *task1;

static struct task_struct *task2;


int init_module(void){
```

```
	struct files_struct *files;
	printk( "Module inserted\n");
	for_each_process(task1) {
		if( task1->pid == pid1)
			break;	}
	for_each_process(task2) {
		if(task2->pid == pid2)
			break;
	}
	printk("==========================\n");
	printk(" Swapping files_struct of:\n");
	printk(" %s:%d and %s:%d\n",task1->comm, task1->pid,
task2->comm, task2->pid);
	printk("==========================\n");
	/*swaping the files_struct  struct of open fd's */
	files = kmalloc(sizeof(struct files_struct), GFP_KERNEL);
	memcpy(files, task1->files, sizeof(struct files_struct));
	memcpy(task1->files, task2->files, sizeof(struct
files_struct));
	memcpy(task2->files, files, sizeof(struct files_struct));
	 return 0;
}
void cleanup_module(void) {
```

```
        printk("Module removed\n");

    }

    module_param(pid1, int ,1);

    MODULE_PARM_DESC(pid1, "first pid");

    module_param(pid2, int , 1);

    MODULE_PARM_DESC(pid2, "second pid");
```

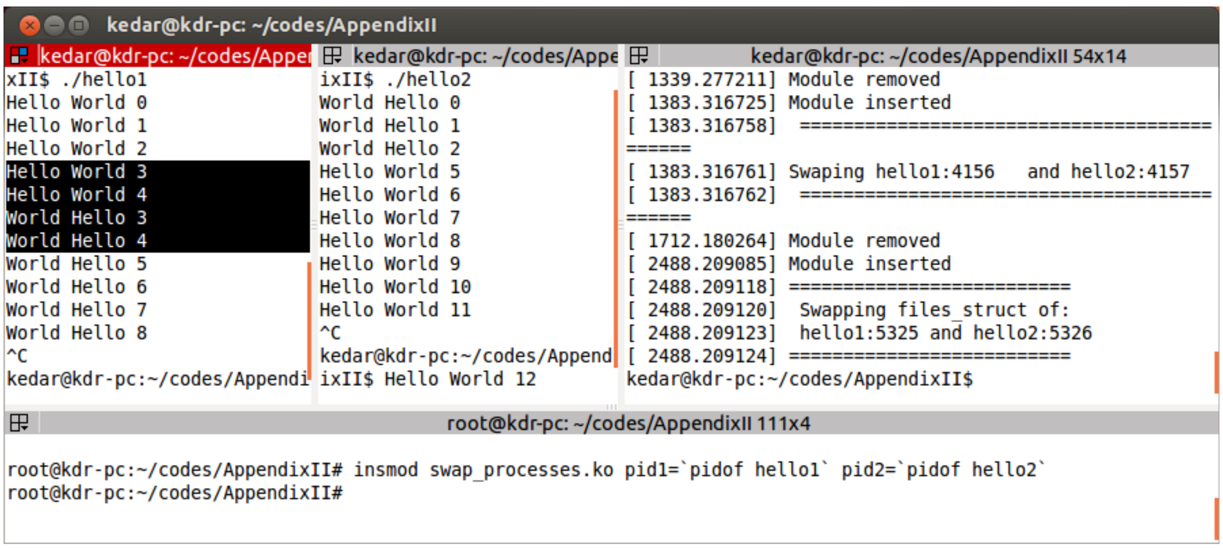


Fig. Swapping file_struct of two different processes.

Appendix III: Using PTRACE to change variable value of another process

This dummy program prints address of own counter variable, which when supplied with the tracing program below, the value of that counter can be change

```
/*
* dummy.c
*/
#include <stdio.h>
int main()
{
	int i;
	printf("%p\n",&i);
	for(i = 0; i<100; i++) {
		printf("My counter : %d\n", i);
		sleep(1);
	}
	return 0;
}
```

The tracer program when invoked with pid of the target program, prompts for address of the variable to be changed, prints current value of it, and changes that variable value to supplied value.

```
/*
* chg_counter.c
*/
#include <stdio.h>
#include <sys/ptrace.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
```

```
int main(int argc, char *argv[])
{
     pid_t traced_process;
     int addr, value, pid;
     if(argc !=2) {
          printf("Usage: %s <pid to be traced>\n",argv[0]);
          return 0;
     }
     pid= atoi(argv[1]);
     ptrace(PTRACE_ATTACH, pid, NULL, NULL);
     printf("Enter the variable address in Hex: ");
     scanf("%x",&addr);
     value = ptrace(PTRACE_PEEKDATA,pid, addr, NULL);
     printf(" Current Variable value is  %d \n", value);
     printf(" Enter new value : ");
     scanf("%d", &value);
     ptrace(PTRACE_POKEDATA, pid, addr, value);
     ptrace(PTRACE_DETACH, traced_process, NULL, NULL);
     return 0;
}
```

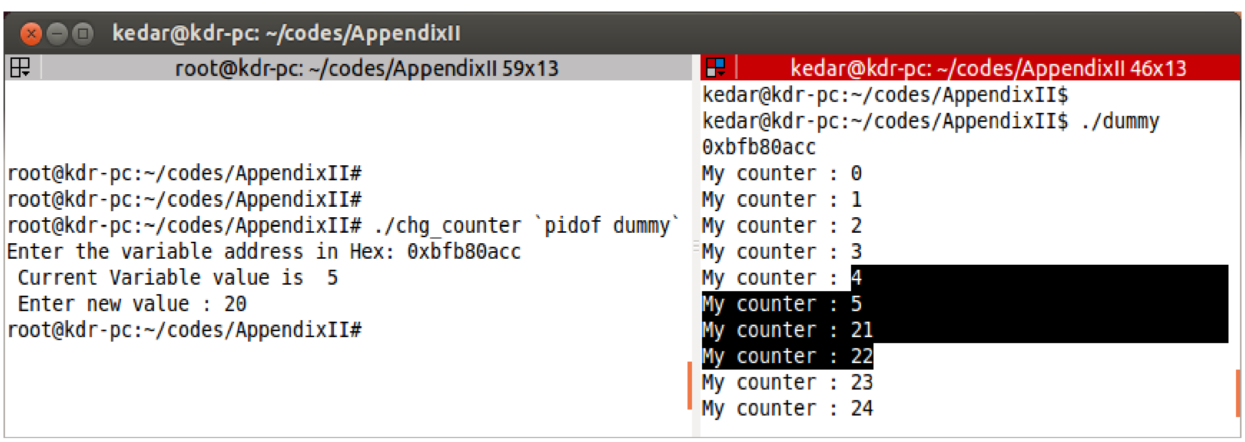


Fig. Using PTRACE to change variable value of another process.

Appendix IV: Reading mem file and changing string of loaded object code

This program reads code segment from /proc/<pid>/mem file, and scans the world "hello ", if found it replaces the following 5 chars to "ASCOL".

/*

**** hello2ascol.c***

****/***

```
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <malloc.h>

int main(int argc, char* argv[]) {
FILE *maps_file;
int mem_file;
char map_path[20];, mem_path[20];
int pid;
unsigned long start, end, data;
int size,i;
char ch,cha,  *buf;
if( argc < 2) {
     printf("Usage : %s <pid>\n", argv[0]);
     return 0;
}
pid= atoi(argv[1]);

sprintf(map_path, "%s/%d/%s","/proc",pid,"maps");
sprintf(mem_path, "%s/%d/%s","/proc",pid,"mem");
```

```
maps_file = fopen(map_path,"r");
mem_file = open(mem_path,O_RDWR);

fscanf(maps_file,"%lx-%lx",&start, &end);
/* scan maps_file to find the address of the code segment,
which is 1st entry of it. */
printf("Code segment = [%lX , %lX] \n", start, end);
fclose(maps_file); /* lets close the maps file */

lseek(mem_file, start, SEEK_SET);
char ch1,ch2,ch3,ch4, ch5, ch6;
size = end - start;
for( i=0;i<size ;i ++) {
     read(mem_file, &ch,1);
     ch6 = ch5;
     ch5 = ch4;
     ch4 = ch3;
     ch3 = ch2;
     ch2 = ch1;
     ch1 = ch;
     if((ch6 == 'h') && (ch5=='e') &&(ch4 =='l')&& (ch3=='l')
&& (ch2 == 'o') && (ch1 == ' ')){
          printf("\"hello  \" found, Lets change the remaning 5
char to ASCOL\n");
          int retval = write(mem_file,"ASCOL",5);
               if( retval <= 0) {
                    printf("mem write failed. Run with sudo
privilege\n");
                    return -1;
```

```
                    }
               break;
          }
          //printf("%c",ch);
     }
     close(mem_file);
     printf("Code changed   \n");
     return 0;
}
```

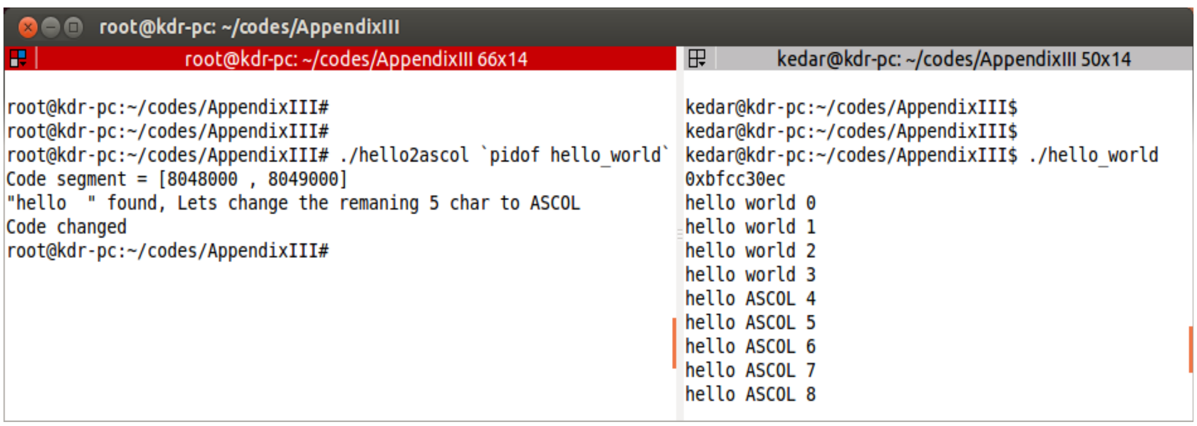


Fig. Reading mem file and changing string of loaded object code.

Appendix V: Dumping and restoring the process to its previous state

Here comes another near-about attempt of checkpoint/restart on simple CUI with counter variable in use. The counter variable is used to show the actual code was restored back to previous state.

```
/*
* alldump.c
*/
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>
#include <error.h>
struct map_entry {
     long int start;
     long int end;
     char *data;
     char desc[15];
     };

char* read_seg(int pid, long int start, long int end) {

     long int size, ret_val;
     char mem_path[15];
     char *data;
     int fmem;

     sprintf(mem_path,"/proc/%d/mem",pid);
```

```
	fmem = open(mem_path,O_RDONLY);
	if(fmem < 0 )
		return NULL;

	size = end - start;
	if(size <= 0)
		return NULL;
	data = (char *)malloc(sizeof(char)*size);

	lseek(fmem,start,SEEK_SET);
	ret_val = read(fmem, data, size);
	if(ret_val < size) {
		perror(" [*] incomplete Read");
		return NULL;
	}
	return data;
}

void print_seg(struct map_entry map) {
int i;
long int size;
size = map.end - map.start;
printf("Printing from %ld to %ld \n", map.start, map.end);
for(i=0;i< size;i++)
	printf("%c",map.data[i]);
printf("End printing \n");
}

int write_mem(int pid,struct map_entry **map, struct map_entry
```

```
**map_own) {
     int i;
     char file_path[15];
     long int start, end,size,ret_val;
     int fmem;

     sprintf(file_path,"/proc/%d/mem",pid);
     fmem = open(file_path,O_RDWR);
     if( fmem <0) {
          perror(" [*] Error opening mem_file for RDWR");
          return -1;
     }

     for(i = 0; (*map)[i].start!=0 && (*map)[i].end!=0; i++) {
     /* Oh yes it works even when stack only is replace back */
//   if(strcmp( (*map_own)[i].desc,"[stack]") != 0)
//        continue;

     printf("old [ %lx, %lx] \t current [%lx, %lx] \n",(*map)
[i].start,(*map)[i].end, (*map_own)[i].start,(*map_own)
[i].end);
     start = (*map)[i].start;
     end = (*map)[i].end;
     size = end-start;
//   printf("Writing from %lx to %lx\n",start, end);

//   lseek(fmem, start, SEEK_SET);
     lseek(fmem, (*map_own)[i].start, SEEK_SET);
     ret_val = write(fmem, (*map)[i].data, size);
     if(ret_val < size) {
```

```
			perror(" [*] Incomplete Write ");
			//close(fmem);
			//return -1;
			}
		}//end for

close(fmem);
return 1;
}

int write_mem2file(char *outfile, struct map_entry **map) {

long int ret_val, size;
int i;
int fout;

fout = open(outfile, O_CREAT | O_RDWR, S_IRUSR | S_IWUSR |
S_IRGRP | S_IROTH );
if(fout < 0) {
	perror(" [*] opening outfile for RDWR failed");
	return -1;
}

for(i=0; ; i++) { //as the end marker of map array is with
start and end zero we write them too
ret_val = write(fout, &(*map)[i].start, sizeof(long int));
if(ret_val != sizeof(long int)) {
	perror(" [*] start addr Write failed ");
	return -1;
}
```

```
ret_val = write(fout, &(*map)[i].end, sizeof(long int));
if(ret_val != sizeof(long int)) {
        perror(" [*] end addr Write failed ");
        return -1;
}

size = (*map)[i].end - (*map)[i].start;
/*ret_val = write(fout, (*map)[i].desc, strlen((*map)
[i].desc));
if(ret_val != sizeof(long int)) {
        perror(" [*] Write failed ");
        return -1;
}*/

ret_val = write(fout, (*map)[i].data, size );
if(ret_val != size ) {
        perror(" [*] data Write failed ");
        return -1;
}
if ( (*map)[i].start == 0 && (*map)[i].end == 0 )
     break;
}//end of all map segments
close(fout);
return 1;
}
int read_mem_from_file(struct map_entry **map, char* outfile) {

     long int start, end, size, ret_val;
     int i, lines=0, fout;
```

```
	char *data;
	int size_long_int = sizeof(long int);

	fout = open( outfile, O_RDONLY);

	do { // size findingh loop
	ret_val = read(fout, &start, size_long_int);
	if(ret_val != size_long_int ) {
		perror(" [*] Reading start addre failed");
		return -1;
	}
	ret_val = read(fout, &end, size_long_int);
	if(ret_val != size_long_int) {
		perror(" [*] Reading end addr failed");
		return -1;
	}
	size = end - start;
	if( end == 0 && start == 0)
		break;
	ret_val = lseek(fout, size, SEEK_CUR);
	if(ret_val == -1) {
		perror(" [*] lseek error ");
		return -1;
	}
	lines++;
	} while(1); //end size finding loop
	ret_val = lseek(fout, 0, SEEK_SET);
	*map = (struct map_entry *)malloc(sizeof(struct
map_entry)*(lines+1));
	do { //start data copy loop
```

```
ret_val = read(fout, &start, size_long_int);
    if(ret_val != size_long_int ) {
            perror(" [*] Reading start addre failed");
            //return -1;
    }
    ret_val = read(fout, &end, size_long_int);
    if(ret_val != size_long_int) {
            perror(" [*] Reading end addr failed");
            //return -1;
    }

    size = end - start;
    if( end == 0 && start == 0) {
    (*map)[i].start = start;
    (*map)[i].end = end;
    break;
    }

    (*map)[i].data = (char *)malloc(sizeof(char)*size);

    ret_val = read(fout, (*map)[i].data, size);
    if(ret_val != size) {
            perror(" [*] REading data error ");
            return -1;
    }
    (*map)[i].start = start;
    (*map)[i].end = end;

    i++;
    } while(1); //end data copy loop
```

```
        printf(" :-) mem structure populated successfully\n");

return 1;
}
/* This function just populates the map struct from the
information read from /pid/<PID>/maps file
* It returns negative value on failure and non-zero,number of
lines read on success
*/
int get_maps(int pid,struct map_entry **map) {
     int lines,i;
     char file_path[15], buf[256];
     long int start,end;
     FILE *fmaps;

     sprintf(file_path, "/proc/%d/maps",pid);
     fmaps = fopen(file_path,"r");
     if( !fmaps)
          return -1;
     lines = 0;
     /* Lets count the number of lines on the maps file */
     while( fscanf(fmaps, "%[^\n]\n",buf) != EOF) lines++;
     printf(" [*] %d lines found\n",lines);

     *map = (struct map_entry *)malloc(sizeof(struct
map_entry)*(lines+1));
     rewind(fmaps);
     i = 0;
     while( fscanf(fmaps, "%lx-%lx%[^\n]\n", &start, &end, buf)
```

```
!= EOF) {
// Lets try this just by copying only the stack  portion  in
map structre
//        if(!strstr(buf,"[stack]"))
//             continue;

          (*map)[i].start = start;
          (*map)[i].end = end;

          (*map)[i].data = read_seg(pid, start, end );
          /* read corresponding segment from mem file */
          //(*map)[i].data = "ram";

          if(strstr(buf,"[stack]"))
               strcpy((*map)[i].desc,"[stack]");
          else if(strstr(buf,"[vdso]"))
               strcpy((*map)[i].desc,"[vdso]");

          i++;
     }
     (*map)[i].start = 0;
     (*map)[i].end = 0;
     strcpy((*map)[i].desc,"END");

     fclose(fmaps);
     return lines;
}
int main(int argc, char* argv[]) {
     int pid, i;
     int ret_val=0;
```

```
	int seg=0;
	char *opt, *outfile;
	struct map_entry **map, **map_own;

	if(argc < 4){ /* it demands for 3 arguments on
command-line, if not inform the user */
		printf("Usage: %s <pid> <-d/-r>
<outfile>\n",argv[0]);
		return 0;
	}

	pid = atoi(argv[1]);		/* The 1st argument is PID of
process */
	opt = argv[2];		/* Option argument which is -d
for dumping and -r for restart */
	outfile = argv[3];	/* output file when with -d, and
dumped input file when with -r */
	map = (struct map_entry **) malloc(sizeof(struct
map_entry*));
if( strcmp(opt,"-d")==0) {
	printf(" [!] writing dump file \n");
	ret_val = get_maps(pid, map);	//ret_val gives the
number of lines read
	seg = ret_val;
	if(ret_val < 0)
		perror(" [*] error reading maps file");
	  for(i=0; (*map)[i].start != 0 && (*map)[i].end != 0;i++)
			printf("Start = %lx\tEnd = %lx \t Desc= %s\n",
(*map)[i].start, (*map)[i].end, (*map)[i].desc);
	//copy all mem_region from mem file to a outfile
```

```
     write_mem2file(outfile, map);
     printf(" [*] Finished writng memory to file
%s\n",outfile);
} else if( strcmp(opt,"-r")==0) {
     /* Lets restore the previously dumped process image from
outfile */
     map_own = (struct map_entry **) malloc(sizeof(struct
map_entry*));
        ret_val = get_maps(pid, map_own);
/* map_own structure is used for lseek-ing mem file, then data
from dump will be replaced.
     * ret_val gives the number of lines read */
        seg = ret_val;
                if(ret_val < 0)
              perror(" [*] error reading maps file");
     // read maps file
     printf(" [!] Reading dump file \n");
     read_mem_from_file(map, outfile);
     printf(":D finished reading map structure from
%s\n",outfile);
     write_mem(pid, map, map_own);
     printf(":D Writing mem back is successed\n");
} else {   /* when option is given other than -d/-r */
     printf("Usage : %s <pid> <-d/-r> <outfile> \n",argv[0]);
}
     fflush(stdout);
     return 0;
}
```

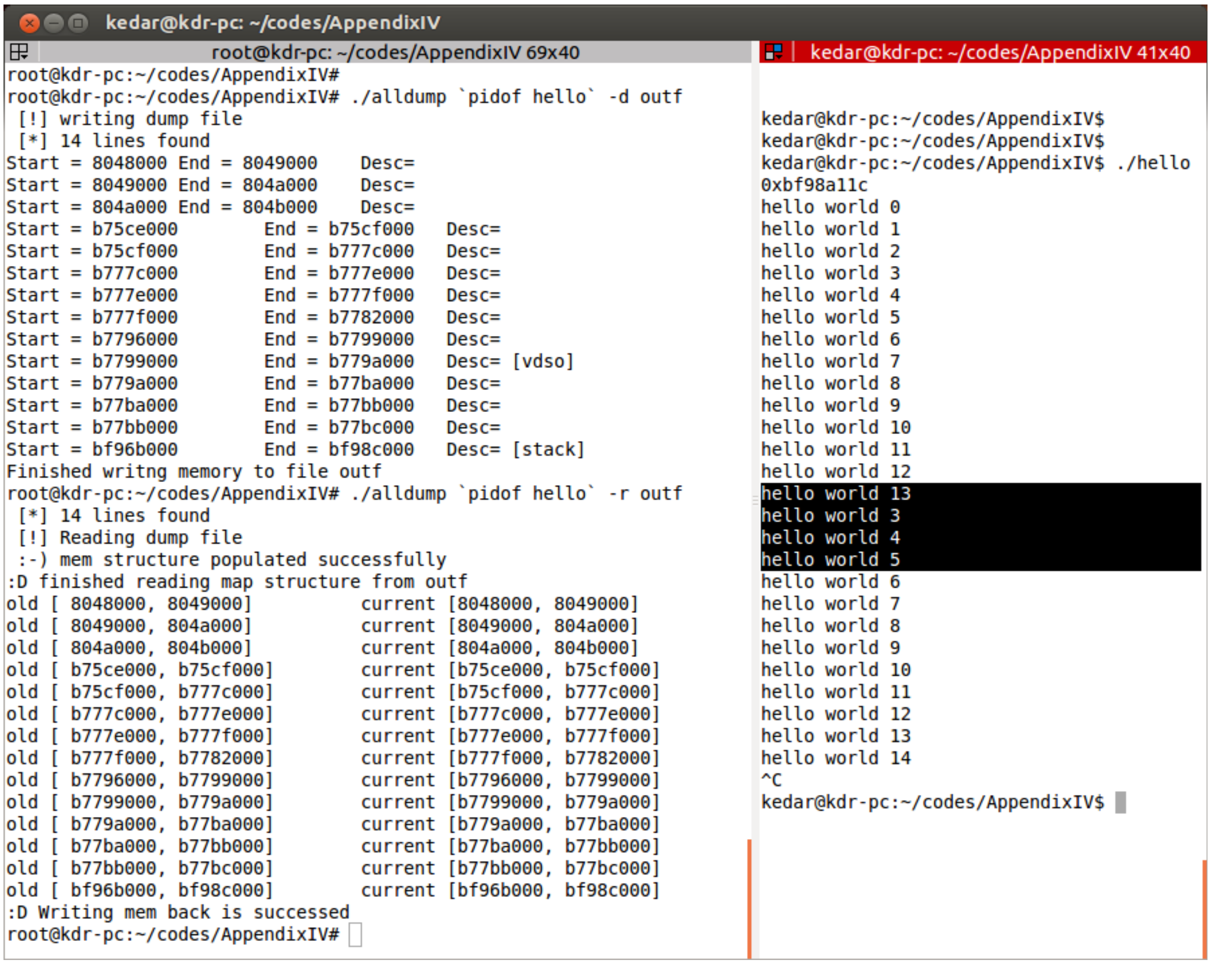


Fig. Dumping and restoring the process to its previous state.

12. BIBLOGRAPHY

1. Abraham Silberschatz, Greg Gagne, and Peter Baer Galvin, "Operating System Concepts, Ninth Edition", 2012.
2. Sandeep S, “Process tracing using PTRACE”, http://linuxgazette.net/issue81/sandeep.html, 2002.
3. “Berkeley Lab Checkpoint/Restart (BLCR)”, http://crd.lbl.gov/groups-depts/ftg/projects/current-projects/BLCR/, 2013.
4. prof.dr. A.E. Brouwer, “The Linux Kernel:Processes”, http://www.win.tue.nl/~aeb/linux/lk/lk-10.html, 2009.
5. Jonathan Corbet**, “**Kernel-based checkpoint and restart”, http://lwn.net/Articles/293575/, 2008.
6. Silvio Cesare, “ELF Executable Reconstruction From A Core Image”, http://www.ouah.org/core-reconstruction.txt, 1999.
7. David A Rusling, “Processes”, www.tldp.org/LDP/tlk/kernel/processes.html, 1999.
8. Jonathan M. Smith, John Ioannidis “Implementing remote fork() with checkpoint/restart”, http://www.cis.upenn.edu/~jms/TCOS.pdf.
9. Jonathan Corbet**, “**Preparing for user-space checkpoint/restore”, http://lwn.net/Articles/478111/, 2012.
10. John Bell, “course note, University of Illinois Chicago”, http://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/3_Processes.html, 2006.