OS Support for Fault-Tolerance in Multi-Core Architectures

(Using a Lightweight Version of FreeBSD)

Bharadhwaja Sharma BS

Bharadhwaja.sharma@iiitb.org

Kota Sai Krishna

SaiKrishna.Kota@iiitb.org

M Maninya

M.Maninya@iiitb.org

Mahesh Babu S

MaheshBabu.S@iiitb.org

Technical Report IIITB-TR-2012-008
April 2012

Abstract

With the increasing use of multicore architectures, computer systems are becoming more prone to faults, due to increased complexity of distributed processing leading to the possibility of faults affecting multiple areas of the system.

This project aims to implement fault tolerance through the FreeBSD operating system, using the checkpoint-recovery technique. This involves saving and restoration of system state; by saving the current state of the system periodically or before critical code sections in the memory, it provides the baseline information needed for the restoration of last state from the memory in the event of a fault in a system core. The fault is communicated through the fault interrupt to the scheduler.

So far, checkpoint of a process and its recovery has been achieved subject to certain conditions. Also, scheduling of processes on the faulty core can be discontinued. The checkpointing is done by the use of a coredump, which consists of the recorded state of the working memory of a computer program at a specific time, generally when the program has terminated abnormally (crashed). The restarted process and the restoration code are in independent address spaces. The restart procedure involves creation of a parent-child process relationship where the parent traces the child process, till it reaches the entry point from where it needs to restart.

Project URL: `https://sourceforge.net/projects/ckptrecovery/`

Contents

List of Figures

1 Introduction

Fault tolerance involves research into a wide range of related areas such as - system architecture, design techniques, operating systems, parallel processing, and real-time processing, to name a few [1].

This project focuses on fault tolerance and recovery for non common-mode permanent hardware faults, with the operating system using the Checkpoint-Recovery technique, which is described in later sections. The operating system chosen is FreeBSD, which is a free unix-like operating system. It has been chosen since it is widely regarded as a reliable, robust and complete operating system and for multicore architectures, the software running on the platform must be written in such a way that it can spread its workload across multiple executions [2]. The kernel, device drivers, and all of the userland utilities, such as the shell, are held in the same source code revision tracking tree [3].

Programming methodologies are progressing towards automating, wherever possible, tasks repeatedly performed secondary to the main task of designing algorithms and developing applications, and this significantly reduces the work of the programmer [6]. Automating the fault tolerance of a system can be done with the help of the operating system. Such a method would be specific to the operating system concerned. In this project, we propose to implement such a system for multi-core architectures using checkpoint-recovery.

We choose a certain time interval, and after every such interval we save the system state, this is known as checkpointing. Thus, a checkpoint represents the state of the system without any prior fault. Suppose a fault occurs in one of the cores during a time interval, then the system recovers the state of the most recent checkpoint and normal functions are made to resume in the remaining functional cores without affecting the priorities of the processes of the faulty core. The tasks of the faulty core are distributed among the healthy cores. This is the recovery stage. In this project, a system for checkpoint-recovery of a single process has been developed.

The rest of this section introduces the problem statement and gap analysis. Section 2 describes the system design which includes an overview of the design objectives, the requirements and the methods used. Block diagrams and architecture are also explained. Section 3 elucidates the implementation details. This includes the procedure to implement checkpoint and recovery, using coredump and restart. Flowcharts are also presented. Section 4 presents the observations and results of the work done so far. The last section 5 presents the conclusion and the future work and scope of the project.

1.1 Problem Statement

In multicore architectures, processes are executed in a distributed manner among all the cores. If one of the cores fails, it could lead to system failure because the cores

depend on each other to complete the tasks, as each core executes a different chunk of the task concurrently. In order to safely recover in the event of a hardware fault occurring in a core, the OS is made to perform backup functions by periodically backing up the state of each core. This involves storing the contents of various registers, physical memory addressing, etc.
The project describes a system for checkpoint and recovery of processes on the FreeBSD operating system, in the event of a hardware fault in a processor core.

1.2 Gap Analysis

Several systems exist that implement fault tolerance. One such mechanism is implemented using redundant hardware [7]. In this method, the system has an extra core and if a fault occurs, this extra core starts performing the tasks of the faulty core. While this method could work satisfactorily for a period of time, it is not very feasible for desktop machines due to the hardware overhead involved. Also there is the problem of the extra core itself being faulty.

Most fault tolerant systems use fault tolerant protocols specific to an application, or use additional hardware, and the OS does not play any significant role in ensuring fault tolerance or recovery. For instance, in desktop computers at present, the FreeBSD operating system simply crashes if one of the cores fails [3].

However, since the OS is closely involved in inter-process communication, and also controls access to memory and processor resources, we propose a system, where the OS takes care of implementing hardware fault tolerance using the checkpoint-recovery technique. This provides run-time support in the event of a fault, by rescheduling the tasks of the faulty core based on their priorities to the functional cores. This would be a more efficient, reliable and time-saving alternative, with reasonable performance degradation [4].

This implementation of checkpoint-recovery of a process differs from some existing implementations in that
(a) It does not require the executables to be linked with library, so processes can be checkpointed without change and
(b) the manner in which a checkpointed process is restarted.

Other systems (such as ckpt and esky) have a complex mechanism of restoring the stack and register state of the checkpointed process as both are also used by the restoration code. This system seems to be simpler as the restarted process and the restoration code are in independent address spaces. The system runs only on user-level code and requires no modifications to the kernel [8].

2 System Design

The main requirements for the design of this project are a method to perform checkpointing of processes in all cores at regular intervals, and to recover these processes

in the event of a fault. A “coredump” has been used to achieve this, and the process is explained in this section.

2.1 Checkpoint and Recovery

Recovery from errors in fault tolerant systems can be characterized as either rollforward or rollback. When the system detects that it has made an error, rollforward recovery takes the system state at that time and corrects it, to be able to move forward. Rollback recovery reverts the system state back to some earlier correct version using checkpointing and moves forward from there [5]. Using this scheme, the performance will increase with only a small loss in reliability.
Checkpointing and rollback recovery are important mechanisms used to improve the fault-tolerance of a system. Should a transient fault occur, like a crash, a power failure, a hardware glitch, or even a transient software error, the most recently saved state can be restored and execution simply resumed.
An important requirement is that the checkpointing scheme be non-intrusive, meaning that it should not interfere in the regular operation of the threads that run user code accessing and modifying memory as they progress. Consequently, it must not suspend such threads for too long (it must exhibit low latency), nor cause excessive slowdowns or variations in their execution speed, making it difficult for the programmer to reason about the ability of their code to meet the required deadlines. Trivially, the checkpointing operation must complete in a bounded time [6].
The system design objective is to achieve fault tolerance in FreeBSD. The process states of all the cores are checkpointed at regular intervals of time. If the fault occurs between these intervals, then all the processes are restarted from the previous checkpoint and scheduled in the functioning core. This is shown in `Figure 1`.

- **Memory** - All the information (register states etc.) about the tasks that are currently performed on each core are stored and periodically updated.
- **Fault Interrupt Generator** - It periodically interacts with all the cores and if a fault occurs in any of the cores, it generates an interrupt to the scheduler.
- **Scheduler** - It divides the tasks among the cores and stores their copies in the memory. If it gets an interrupt from the fault interrupt generator, it identifies the core in which the fault has occurred and retrieves the most recently stored information of the tasks performed by that faulty core from the memory. Then it schedules those tasks again in the remaining functional cores without affecting their priorities. Then further scheduling is done only for the functional cores without considering the faulty core.

2.2 Coredump and Restart

Coredump
A core dump consists of the state of a program at a certain point in time. It is generally created when the program stops abnormally. One can think of it as a full-length "snapshot" of RAM, mainly for the purpose of debugging a program. It

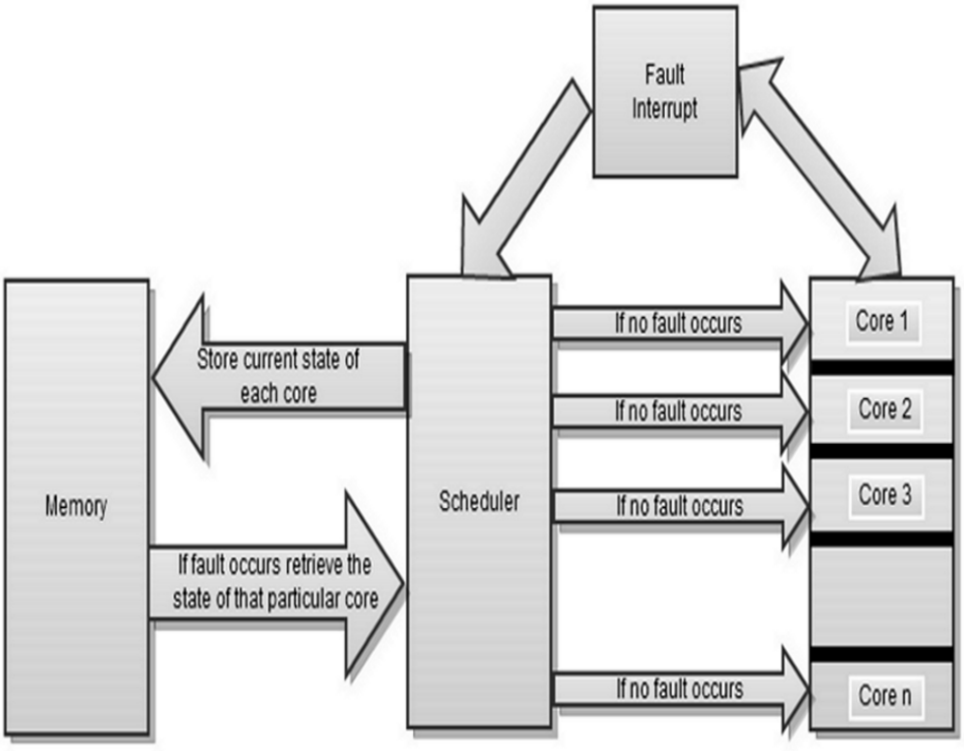

Figure 1: Checkpoint-Recovery Fault Tolerance Operation

is usually used by specialized programmers who maintain and occasionally have to debug problems in operating systems. Typically, a core dump or actually the report that results from the core dump presents the RAM contents as a formatted series of lines that indicate memory locations and the hexadecimal values recorded at each location.
Aside from the entire system memory or just part of the program that aborted, a core dump file may include additional information such as:

- The processor's state
- The processor register's contents
- Memory management information
- The program's counter and stack pointer
- Operating system and processor information and flags

Additional information tells exactly which instruction was executing at the time the core dump was initiated. Thus in theory it should be possible to restore the process to the same state it was in when the core was dumped [8]. There are some important things to consider when restarting from this state. Some of these issues are:

- How to handle file descriptors? Files may have changed, how can we handle sockets, pipes, seeks etc.

- Should the restarted process have the same process id as before?
- Signals that need to be blocked.
- How the process sees the time that has elapsed since the checkpoint.

These issues would define what we mean by "restarting" a process. However, we can observe that for compute-intensive jobs, where inter-process communication and signal handling aren't the major point of concern - the process address space has all the information necessary. Hence restarting from the address-space dump in the core would serve a worthwhile purpose [8].
The checkpointing is achieved by making the coredump of a process using the `gcore` command of gdb. The gcore utility creates a core image of the specified process, suitable for use with gdb(1). By default, the core is written to the file `core.<pid>`. The process identifier, pid, must be given on the command line [10]. Once core has been dumped, the program continues operation; it does not stop. Thus, gcore is especially useful for taking a snapshot of a running process. Using the contents of the coredump file, the process is restarted.

Restart
In the restart of the process implemented here, we have chosen to restart a very simple process, and have not considered the issues of file descriptors, signal handling and so on. The coredump and the executable file is given as the input to a parent process. The parent process creates a child process, which is to be restarted as the process that crashed. The child process should get a fixed memory space and get a copy of all the states of the crashed process, then it is detached from the parent process to restart the dumped process. The steps involved are shown in the flowchart [`Figure 2`].
In order to restore the address space and to find the entry point of the restart, the child process is "traced" by the parent process using the `ptrace` utility [9] [11]. Using ptrace allows one to set up system call interception and modification at the user level. The basic usage of 'ptrace' [12] can be summed up as follows:

- Parent process forks and execs the child; the child should express its willingness to be traced.
- Parent sends a signal to the child and waits for the child to stop.
- Once the child stops, the parent can manipulate it in different ways by invoking 'ptrace' with several different 'requests'.
- When the parent has completed all 'manipulations', it can restart the child and let it run to completion (or maybe arrange things in such a way that the child automatically stops at the next entry to a system call or stops after doing just one instruction).

The use of the exec() call and breaking at the entry point of the program handles the initialization of the process' address space and loading the executable code of the program.

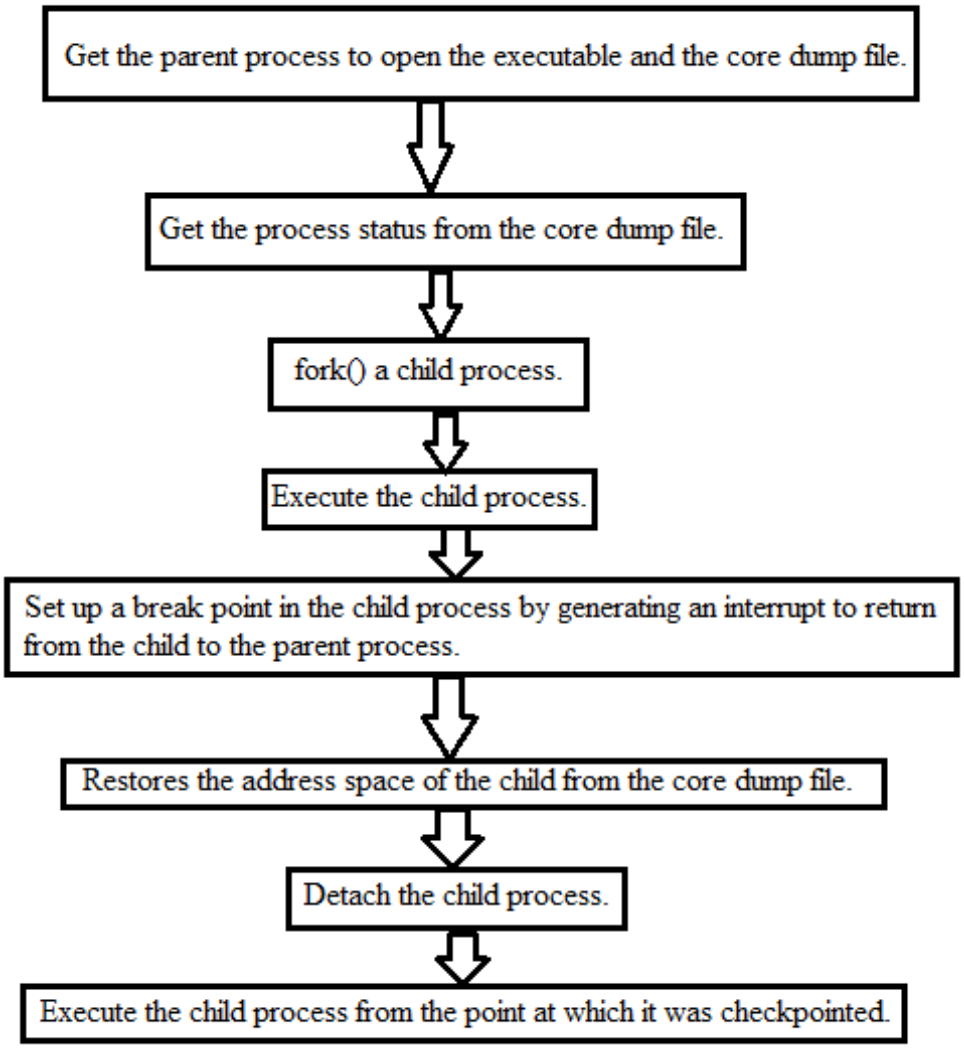

Figure 2: Flowchart - Restarting a Process

For instance, using PT_READ_D the contents of the USER area where register contents and other information is stored can be examined. The kernel stores the contents of registers in this area for the parent process to examine through ptrace. It is also possible to read register values at the time of a syscall entry or exit. This can be done by using ptrace with a first argument of PT_GETREGS which will place all the registers in a single call.
The register values and other information are read from the core dump file, then the child process is created and traced till the entry point is reached. This allows the address space to be initialized and the core to be loaded. Then the parent restores the address space of the child and it is ready to execute from where it was checkpointed.

3 Implementation Details

- The system works on FreeBSD 9.0
- The "checkpoint" file used is an ELF core file with type ET_CORE. ELF, or Executable Linkable Format, is a common file format used for executables, object codes and core dumps. It has been used here due to it's flexibility and extensibility. This implementation works on the i386 architecture (The architecture affects, among other things, the registers available etc.)
- In such a system, the stack starts at 0xbfffffff and "grows" to lower addresses
- The .text, .data and .bss segments of the executable are loaded at 0x0804000. Dynamic libraries are loaded by ld at 0x4000000 onwards

3.1 Check-Pointing

Here an attempt is made to checkpoint the process by getting the core image of the running process. From the generated core image, the restoration of the process takes place. The `gcore` command is used as follows:

```
gcore [-s] [-c core] [executable] pid
```

An executable file for a light-weight process is created and run. The process is check-pointed using the gdb debugging tool. Gdb tool is used for executing the process step by step. Using gdb's gcore command a memory dump is created after executing the process up to a certain number of execution steps. The execution is suddenly terminated and the restoration stage of the process will begin. gcore $< image - name >$ To attach a running process to gdb use:

```
gdb <executable filename> <process id>
```

Thus the core dump file is created. The checkpointing can be done at regular intervals. Along with the executable file of the process the core dump file is used to recover from the fault. In our case the process that is running is killed indicating fault occurrence.

Note: The core dump file which is created does not check-point the file descriptors. To checkpoint the file descriptors care must taken as many issues arise while check-pointing these file descriptors like what happens if the file is moved and what about sockets and pipes etc.

3.2 Recovery

The process is recovered from a fault from the previous checkpoint with the help of the core dump file. The steps followed for process recovery are:

- Open the executable and core files and read their ELF headers.

- From the NOTES program header of the core file, get the PR_STATUS structure (which has register values) of the checkpointed process.
- fork(), we now have a CHILD process (which will be the restarted image) and the PARENT process (that sets up the child).
- CHILD: ptrace(PT_TRACEME,...) and then exec() the executable file.
- PARENT: Setup a breakpoint in the child. This is done as follows: Store the instruction in the child at the entry point of the executable and replace it with the INT3 instruction (opcode 0xCC). Then do a ptrace (PT_CONTINUE,...). This allows the child process to run till it reaches the entry point (normally the address of the _start function). Once here, it will execute the INT3 instruction which causes a SIGTRAP to be generated and returns control to the parent process. (Allowing the child to run till the entry point allows the address space to be initialized and the code to be loaded). In the case of statically compiled binaries (e.g.: gcc with -static), instead of the entry point, we would want to break at the address of main()).
- PARENT: With the help of the LOAD sections in the core file, restore the address space of the child. (The program headers with type LOAD specify the virtual address and the location in the file where the contents of that address can be found).
- PARENT: Restore the registers of the child, read in from the NOTES section of the dumped core.
- PARENT: Detach the child (ptrace (PT_DETACH,...)).
- THE CHILD PROCESS IS NOW READY TO EXECUTE FROM THE POINT IT WAS CHECKPOINTED.

Here we use the mmap system call to allocate memory to the child process.

3.3 The cpuset Command

The cpuset command can be used to assign processor sets to processes, run commands constrained to a given set or list of processors, and query information about processor binding, sets, and available processors in the system [13].

The functioning of this command can be observed through the system monitor [Figure 3]. Under "CPU history" is shown the percentage usage of each CPU core. We observe that each core is being utilised to run some processes.

We then run the following command:

```
cpuset -s 1 -l 0,3
```

The cpuset command counts cores 1 to 4 as 0 to 3. We observe on the monitor that cores 2 and 3 do not have any processes running, we have effectively "disabled" these two cores [Figure 4].

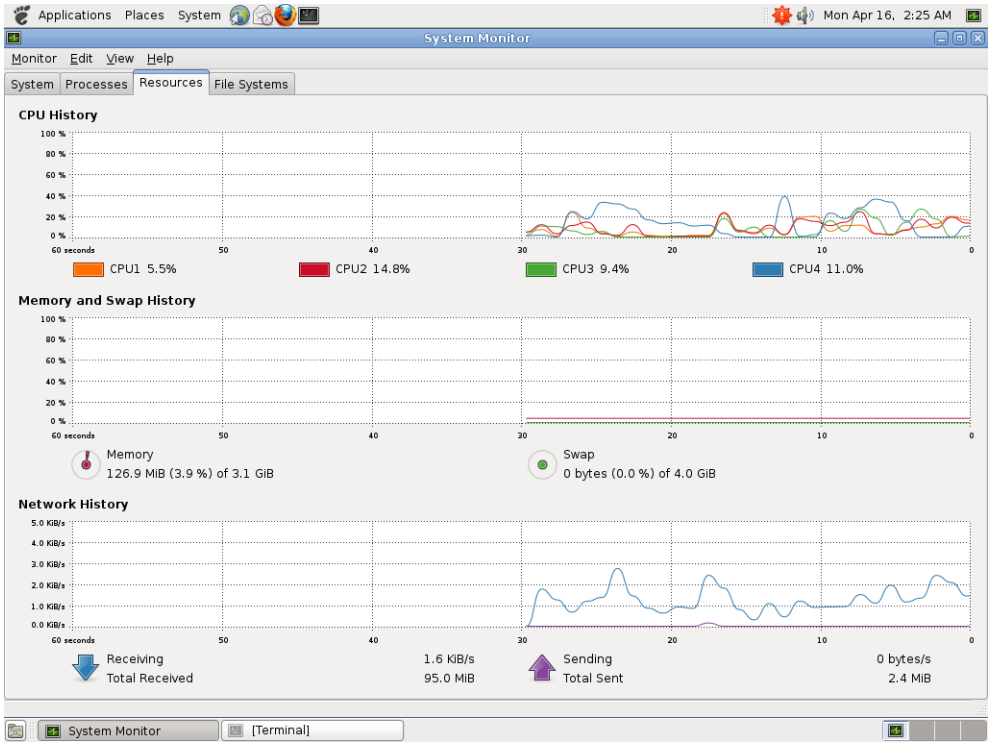

Figure 3: System Monitor showing CPU Usage

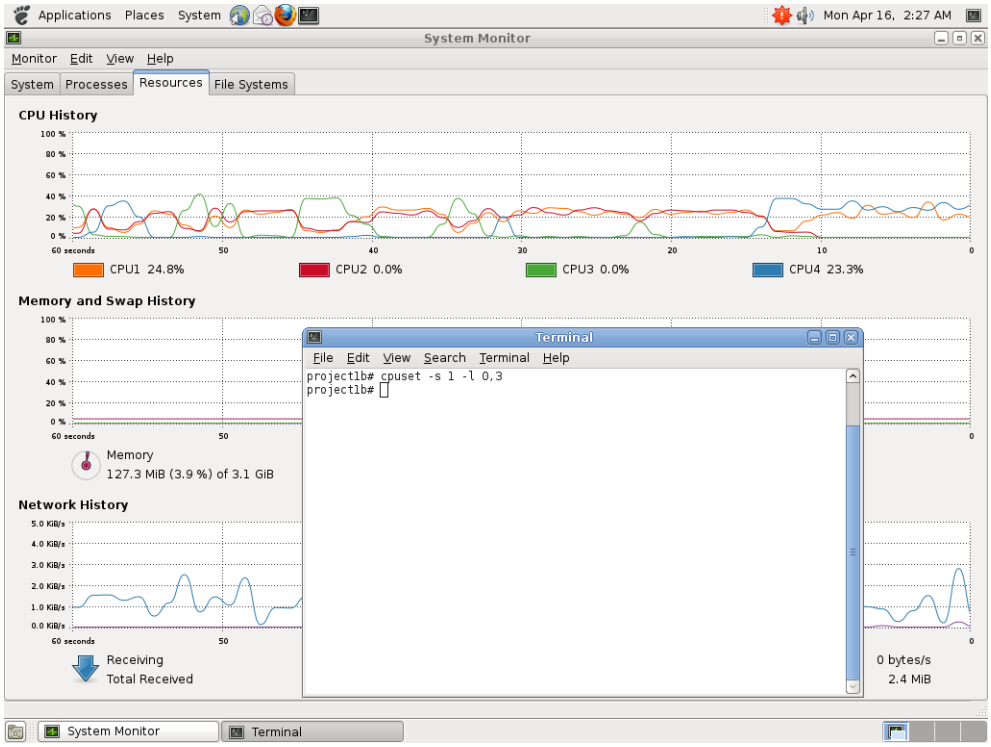

Figure 4: Confining Processes to Some Cores

We can also confine processes to a single core :

```
cpuset -s 1 -l 2
```

Now all the processes are running on core 3 [`Figure 5`].

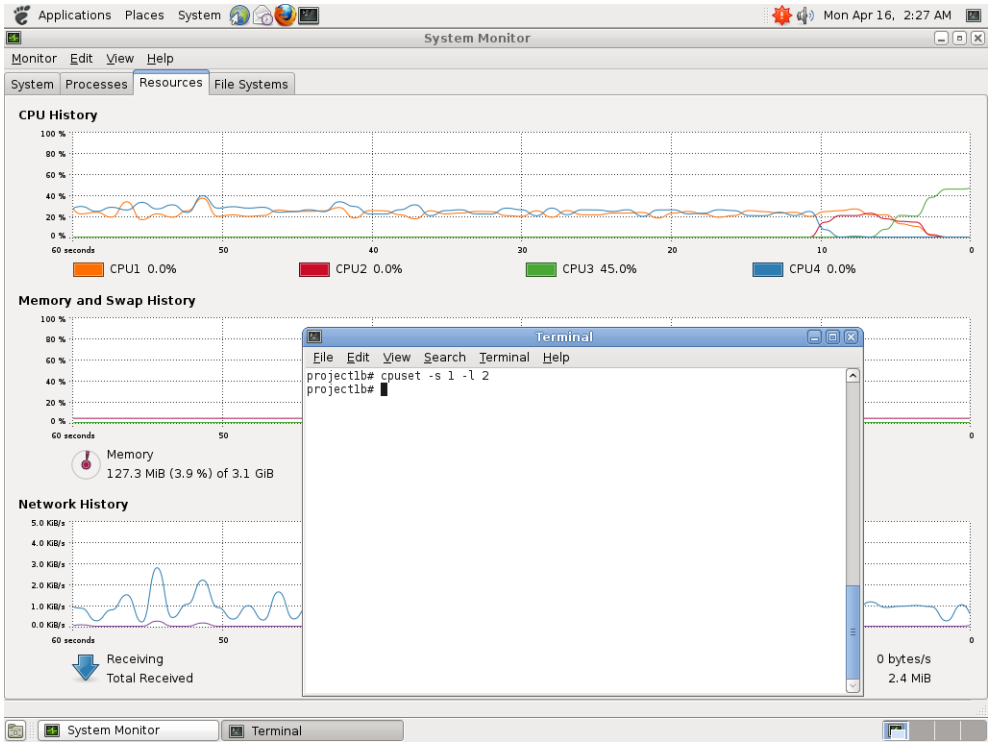

Figure 5: Confining Processes to a Single Core

We note that the `cpuset` command doesn't actually *disable* the faulty core, it just prevents processes from being scheduled on it. Thus for our objective of fault tolerance, if a fault occurs in a processor core, we can constrain processes to run only on the remaining cores using this command. This can be done for our restarted processes too.

4 Observations

Observations on core dump file

In FreeBSD systems, core dumps generally use the Executable and Linkable Format (ELF). Each ELF file is made up of one ELF header, followed by file data. The file data can include:

- Program header table, describing zero or more segments
- Section header table, describing zero or more sections
- Data referred to by entries in the program header table or section header table

The segments contain information that is necessary for runtime execution of the file, while sections contain important data for linking and relocation. Any byte in the entire file can be owned by at most one section, and there can be orphan bytes which are not owned by any section. The core dump file contains all the register values and these values are saved at the specified addresses. The values are to be read and then written on to those address spaces which are allocated to the child process in the restoration stage.

Observations on register structure

The register structure is studied and the register structure is used in $< machine/reg.h >$. In this file the structure "reg" is defined and all the register values can be accessed using this structure. To get the register values of the target process the ptrace function is used in the following way:

`ptrace(PT_GETREGS,exec_pid,(caddr_t)&regs,0).`

Thus the register values are stored in the address pointed to by regs and the target process is specified by the exec_pid. PT_GETREGS is the request argument sent for the ptrace utility.

Observation on injecting code into a targeted process

To allocate the memory that specified is for the process a system call has to be made by the process i.e. calling the mmap utility. To make the child (target) process call mmap() using ptrace the FreeBSD system call convention has to be followed.

5 Results

The results obtained were for a system that can checkpoint and restart simple lightweight processes, subject to the following conditions:

- They do not contain file descriptors (this has not been implemented currently).
- Programs must be single threaded.
- Programs that use the mmap() call to map files to the process' address space (processes that use malloc()) cannot be restarted.
- Only a single process will be checkpointed, thus programs that use fork(), exec() (or other things like system() and popen()) cannot be restarted.
- Signal handlers are not restored (default ones are used).

The restart of one such process is described here:
A simple C program was written that prints the numbers 0 to 99 in a loop, one number per line. Using the `gcore` command, this process was checkpointed every

5 seconds using a cron job. To test the system, the process is killed at some point by passing a SIGINT signal to it (ctrl-C), after noting the last number it printed. Then it is restarted by the restart program, which takes the checkpoint core file as input.

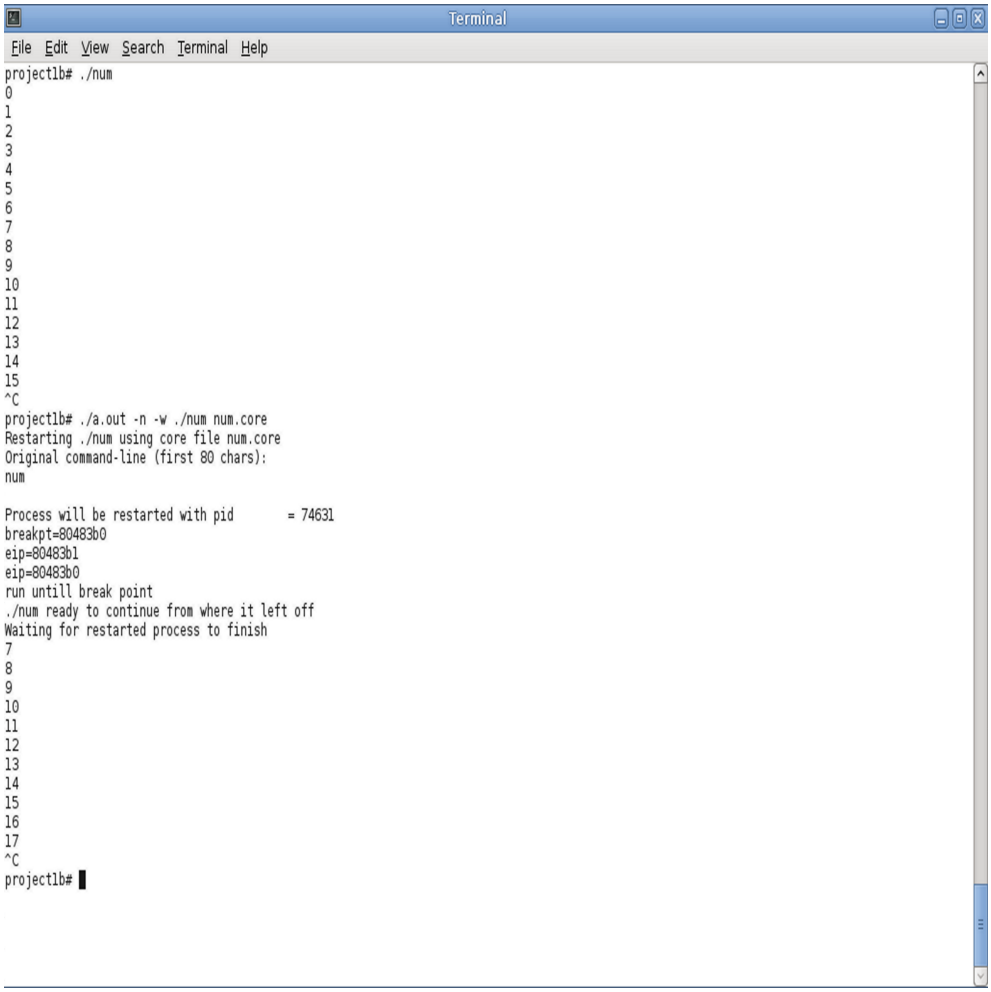

Figure 6: Killing and Restarting a Process

When the process restarts, we observe that it starts printing from a point exactly after its previous print or a few numbers before that. This is because it is printing from the last checkpointed state [`Figure 6`]. To migrate a restarted process from one core (the faulty one) to another, we use the `cpuset` command as follows: Find out which core the process is running on using the top command as shown [`Figure 7`]. We observe that the process is running on core 2.

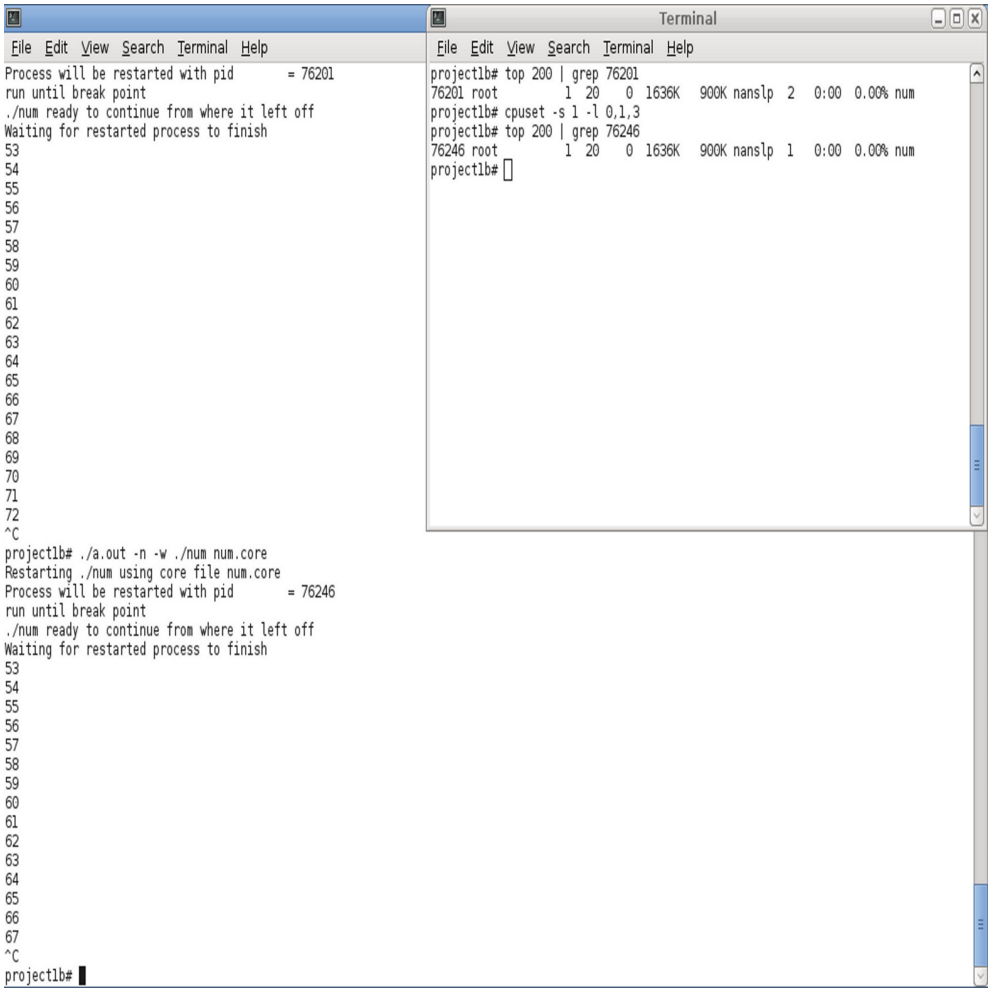

Figure 7: Migrating a Process to a Different Core

We kill the process, and then using the `cpuset` command, we constrain it to run only on cores 0,1,3.

```
cpuset -s 1 -l 0,1,3
```

When the process is restarted, we find that it runs on core 1. It will never be scheduled to core 2.

6 Conclusion and Future Scope

The aim of the project was to make the FreeBSD OS fault tolerant, and a step was taken in that direction. To get a fault tolerant system, the checkpoint and recovery technique was followed. A process was successfully restarted from its coredump file, under certain conditions.
Fault tolerance is an active research area, and OS support is one of the most important fields in this regard. Future scope for this project would be to include file

descriptors, signal handling and handling of larger programs with dynamic allocation of memory. Also, it should be possible to disable faulty cores and force specific processes to run on desired cores.

References

[1] D. A. Rennels, "Fault-tolerant computing - concepts and examples," *IEEE Trans. Computers*, vol. 33, no. 12, pp. 1116–1129, 1984.

[2] *Intel Multi-Core Processor Architecture Development Backgrounder*, Intel Corporation, 2005, white paper.

[3] "Freebsd hackers forums." [Online]. Available: www.freebsd.org/

[4] L. Ryzhyk and I. Kuz, "Towards operating system support for application-specific fault-tolerance protocols," in *Proceedings of the 2nd International Workshop on Object Systems and Software Architectures*, Victor Harbor, South Australia, Australia, Jan 2006, pp. 63–67.

[5] M. Hiller, "Software fault-tolerance techniques from a real-time systems point of view," Department of Computer Engineering, Chalmers University of Technology, SE-412 96 Goteborg, Sweden, Tech. Rep. 98-16, 1998. [Online]. Available: www.cs.drexel.edu/~bmitchel/course/cs575/ClassPapers/hiller98software.pdf

[6] A. Cunei and J. Vitek, "A new approach to real-time checkpointing," in *Proceedings of the 2nd international conference on Virtual execution environments*, ser. VEE '06. New York, NY, USA: ACM, 2006, pp. 68–77. [Online]. Available: http://doi.acm.org/10.1145/1134760.1134771

[7] S. Rajesh, C. V. Kumar, R. Srivatsan, S. Harini, and A. P. Shanthi, "Fault tolerance in multicore processors with reconfigurable hardware unit," 2008. [Online]. Available: http://www.docstoc.com/docs/41330861/Fault-Tolerance-in-Multicore-Processors-Using-Reconfigurable

[8] A. Shankar, "Process checkpointing and restarting," 2004, 2005. [Online]. Available: http://geocitiessites.com/asimshankar/checkpointing/index.html

[9] P. Padala, "Playing with ptrace," 2002. [Online]. Available: http://www.linuxjournal.com/article/6100?page=0,1

[10] "Freebsd man pages-gcore." [Online]. Available: http://www.freebsd.org/cgi/man.cgi?query=gcore&sektion=1

[11] "Freebsd man pages-ptrace." [Online]. Available: http://www.freebsd.org/cgi/man.cgi?query=ptrace&sektion=2&apropos=0&manpath=FreeBSD+9.0-RELEASE

[12] "Tracing tricks with ptrace." [Online]. Available: http://pramode.net/articles/lfy/ptrace/pramode.html

[13] "The unix and linux forums - man pages." [Online]. Available: http://www.unix.com/man-page/FreeBSD/1/cpuset/