Project 1b
OS Support for Fault-Tolerance in Multi-Core Architectures

(Using a Lightweight Version of FreeBSD)

Team

Name	Roll Number	e-mail	Contact
Bharadhwaja Sharma BS	MT2011027	Bharadhwaja.sharma@iiitb.org	9738408417
Kota Sai Krishna	MT2011065	SaiKrishna.Kota@iiitb.org	8095081050
M Maninya	MT2011072	M.Maninya@iiitb.org	8861047403
Mahesh Babu S	MT2011076	MaheshBabu.S@iiitb.org	8970059635

Team Leader : Kota Sai Krishna

Contents

1 Introduction

Multicore architectures divide the computational work performed by a single microprocessor core and spread it over multiple execution cores, hence a multicore processor can perform more work within a given clock cycle. Thus, it is designed to deliver a better overall user experience and to enable this improvement, the software running on the platform must be written in such a way that it can spread its workload across multiple executions [1]. The increased complexity of such distributed processing leads to the possibility of faults affecting multiple areas of the system, making them more error-prone.

Faults in such a system can be distinguished as common-mode and non common-mode faults. Common-mode faults are multiple faults which have a single cause, i.e. they are not statistically independent. Examples are faults due to floods, aging or earthquake. Non common-mode faults would have a different cause for each fault, and may not be related to each other in any way. Conditions due to which such faults can occur are atmospheric noise, electronic component failures, high power dissipation and so on.

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. The kernel, device drivers, and all of the userland utilities, such as the shell, are held in the same source code revision tracking tree [2]. The scheduler in the OS is designed to maintain the state of the processes scheduled on each core in the memory until the processes are successfully executed. If a hardware fault occurs, an interrupt is generated and the scheduler retrieves the previous state of the faulty core from memory and schedules it on the functional cores based on its priority. From that instant, the scheduler works only with the functional cores, and thus, provides fault tolerance to prevent the failure of the system.

2 Project Description

2.1 Objective

The project is aimed at developing a fault tolerant model using the support of the FreeBSD operating system and an implementation of this model on a multicore architecture (Intel x86 architecture) for a desktop environment.

2.2 Description

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.

This project uses the Checkpoint-Recovery scheme to do this, which is a common technique for imbuing a program or system with fault tolerant qualities. The basic idea behind checkpoint-recovery is the saving and restoration of system state, i.e. 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 [3].

We choose a certain time interval and after every interval we save the system state, this is known as *checkpointing*. Thus, 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.

3 Gap Analysis

An extensive methodology has been developed in the field of fault tolerant computing over the past thirty years and a number of fault tolerant machines have been developed - most dealing with random hardware faults, while a smaller number deal with software, design and operator faults to varying degrees [4].

The fault tolerant mechanism can be implemented using redundant hardware [5]. 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 [2].

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 [6].

4 Project Architecture

4.1 Fault Tolerance Phases

Fault tolerance is the ability of an operational system to tolerate the presence of faults. It can be achieved in four phases, namely, error detection, damage assessment, error processing (recovery or compensation) and fault treatment. These four phases are to be implemented in the checkpoint-recovery technique for designing fault tolerant systems. `Figure 1` below illustrates these four phases.

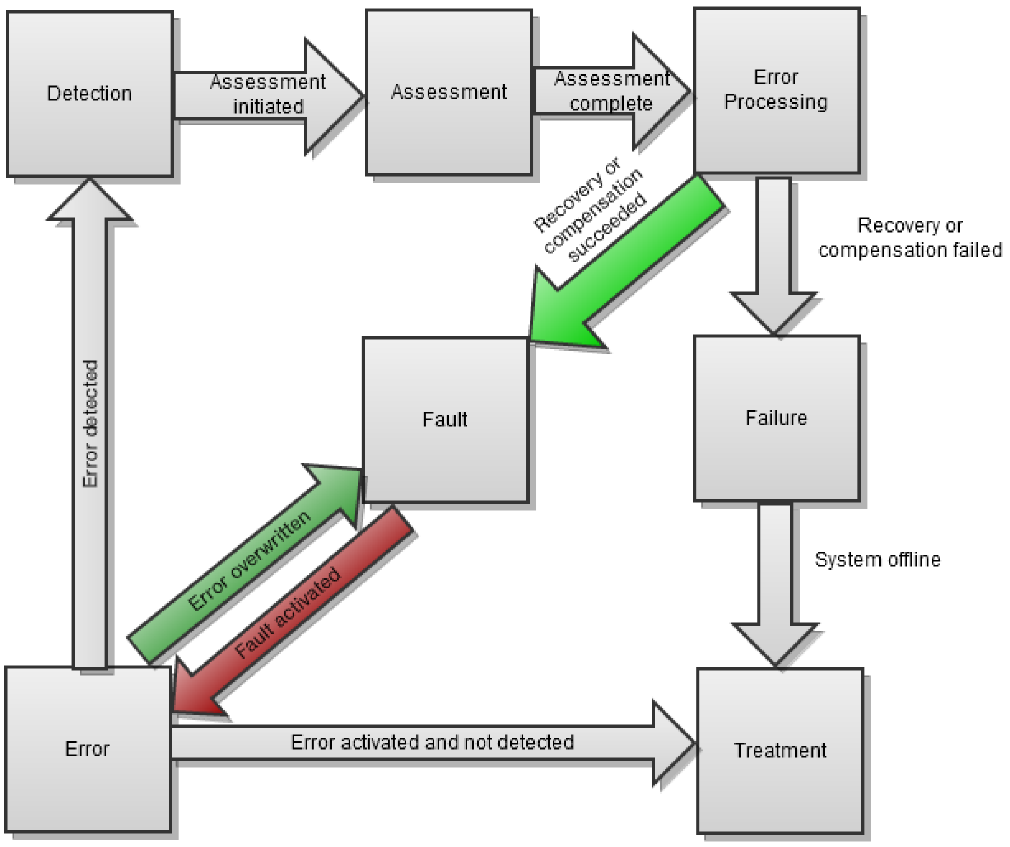

Figure 1: The Process of Faults, Errors and Failures, and the Four Phases of Fault Tolerance

If a fault occurs in a system and the system identifies that the fault has occurred, then it detects the cause for the fault. This is the *error detection* phase. During this process, if the error is rectified or overwritten, then it resets the fault. Once the cause is known, the severity of the fault is assessed in the *damage assessment* phase.

This project mainly focuses on the *error processing* phase, which can be done by recovery or compensation. Error recovery tries to replace the erroneous system state with the previous state, which is error-free. This recovery can be done using either backward recovery or forward recovery. Backward recovery means that the system is brought back to an error-free state recorded in a recovery point - a “snapshot” of the system state prior to the erroneous state. This can be done by the *checkpoint-recovery technique* (section 4.2). Forward recovery means the transformation of the erroneous state that consists of finding a new state from which the system can operate (often in degraded mode) [7].

If error processing fails, then the system reaches failure state in which it stops performing the tasks. In this case, or if the error occurred and was not detected at the beginning, the system enters the *fault treatment* phase. In this phase, the reason for the fault is examined and it is corrected.

The blocks in `Figure 1` are:

- **Fault** - If a fault occurs, it is activated and invokes the fault tolerance phases.
- **Error** - Check whether the error is temporary or permanent. If the error exists for more than the threshold time, then it goes to error detection, else it resets the fault interrupt.
- **Detection** - Identify the causes of the fault.
- **Assessment** - Examine the severity of the fault that has occurred.

- **Error Processing** - Rectify the error using recovery or compensation technique.
- **Failure** - If error processing fails, the system stops performing the tasks until the fault is treated.
- **Treatment** - Identify the source of fault and correct it.

4.2 The Checkpoint-Recovery Technique

Recovery from errors in fault tolerant systems can be characterized as either *roll-forward* or *roll-back*. When the system detects that it has made an error, roll-forward recovery takes the system state at that time and corrects it, to be able to move forward. Roll-back recovery reverts the system state back to some earlier correct version using *checkpointing* and moves forward from there. Using this scheme, the performance will increase with only a small loss in reliability [7].

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 [3]. The checkpoint-recovery technique is illustrated in `Figure 2` below.

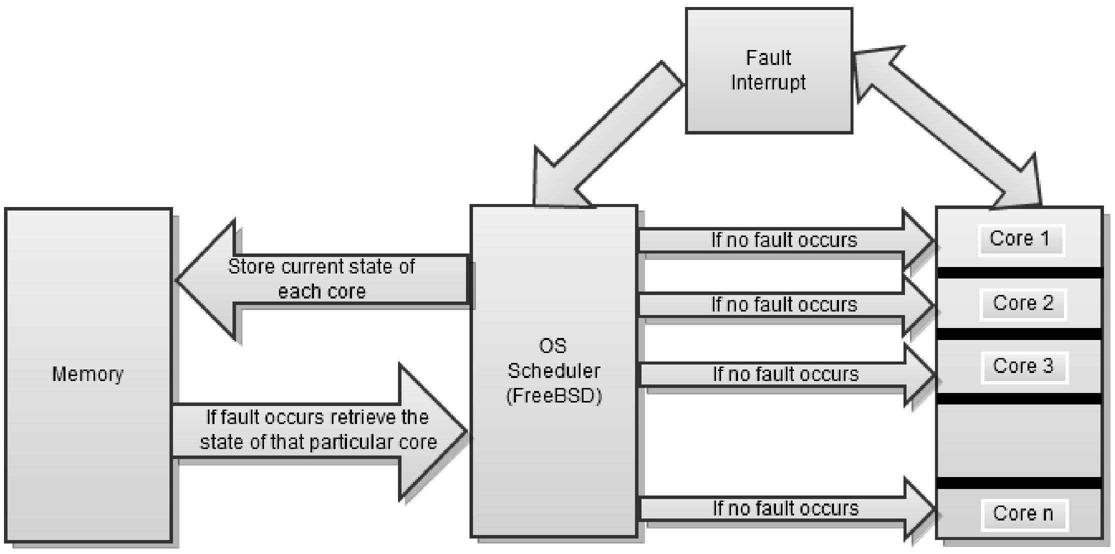

Figure 2: Checkpoint-Recovery Fault Tolerance Operation

- **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.
- **OS 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.

5 Implementation Plan

1. Setup the FreeBSD working environment in the Intel multicore system and download the source code from the FreeBSD repository.

2. Analyze the source code of the FreeBSD operating system and create a schematic for the scheduling algorithms used in it to help in clear analysis.

3. Identify the tasks to be performed and list out the changes that need to be made in the code.

4. Implement the checkpoint-recovery method in C code by saving the checkpoint details at different memory locations, which involves saving the current state of the system.

5. Deploy the new C code into the source code by identifying a suitable location, ensuring no disturbance to the program flow.

6. Build and compile the modified source code and integrate it with the operating sytem scheduler.

7. Design fault simulation techniques for testing the system.

8. Test the performance of the system by running the modified source code and simulating faults in one of the cores through interrupt signals.

6 Milestones

Milestone	Task	Due Date
1	- Setup FreeBSD on an Intel multicore system. - Create schematics to describe the scheduling algorithms used by FreeBSD.	February 7, 2012
2	- Identify the tasks that need to be performed to use the checkpoint-recovery technique. - List out the changes required to be made.	February 16, 2012
3	- Implement the checkpoint-recovery method in C.	February 23, 2012
4	- Locate a suitable place in the source code and deploy the new code.	March 1, 2012
5	- Build and compile the modified code.	March 15, 2012
6	- Create fault simulations using interrupts. - Test, verify and debug the functionality of the system.	March 30, 2012
7	- Refine the results. - Prepare the final documentation.	April 14, 2012
8	- Submissions and final project review.	April 19, 2012

References

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

[2] http://www.freebsd.org/.

[3] 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

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

[5] 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

[6] 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.

[7] 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