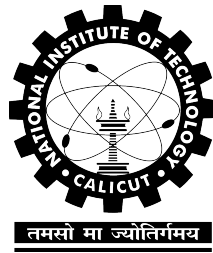# Extending eXpOS to Multi-core Environment

CS4090 Project Final Report

Submitted by

Arun Joseph   B150102CS

Under the Guidance of
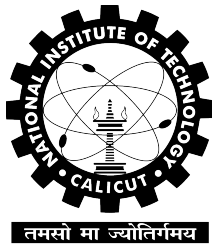**Dr. K. Muralikrishnan**

Department of Computer Science and Engineering
National Institute of Technology Calicut
Calicut, Kerala, India - 673 601

April 26, 2019

NATIONAL INSTITUTE OF TECHNOLOGY
CALICUT, KERALA, INDIA - 673 601

**DEPARTMENT OF COMPUTER SCIENCE AND
ENGINEERING**

तमसो मा ज्योतिर्गमय

2019

CERTIFICATE

Certified that this is a bonafide record of the project work titled

EXTENDING EXPOS TO MULTI-CORE ENVIRONMENT

done by

Arun Joseph

*of eighth semester B. Tech in partial fulfillment of the requirements for the
award of the degree of Bachelor of Technology in Computer Science and
Engineering of the National Institute of Technology Calicut*

Project Guide   **Head of Department**

Dr. K. Muralikrishnan   Dr. Saleena N

Associate Professor   Associate Professor

DECLARATION

I hereby declare that the project titled, **Extending eXpOS to Multi-core Environment**, is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which has been accepted for the award of any other degree or diploma of the university or any other institute of higher learning, except where due acknowledgement and reference has been made in the text.

Place :   Signature :
Date :   Name :
                                                       Reg. No. :

Abstract

Project eXpOS (eXperimental Operating System) is an on-line educational platform which helps undergraduate students to learn the working of an operating system. It is an instructional tool for students to learn and implement OS data structures and functionalities in a few months. This project aims at extending eXpOS to a dual-core environment.

ACKNOWLEDGEMENT

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Project eXpOS [1] is an educational platform for learning and developing a simple operating system on the XSM (eXperimental String Machine) Simulator. Earlier XOS [2] (previous version of eXpOS) was used, which was replaced with the current updated version.

eXpOS has features like multi-programming, process management, disk management, demand paging, synchronization and multi-user capabilities. The basic machine model consists of memory, disk and CPU. There are two privilege levels of execution, namely user mode (unprivileged) and kernel mode (privileged). The virtual address space of a process is divided into library, heap, code and stack. The Operating System consists of 15 interrupts, 26 system call routines, 8 modules, handlers for hardware interrupts like timer, disk and console, and exception handling. It also contains OS data structures like Process Table, Page Table, Open File Table, etc.

The eXpFS (eXperimental File System) provides a file abstraction for the programmer. It consists of three types of files - the root, data files and executable files. The ABI (Application Binary Interface) is the interface between a user program and the kernel. The support tools for eXpOS includes XFS

Interface, ExpL (Experimental Language) compiler, SPL (System Programmer's Language) compiler and XSM debugger.

1.1 Literature Survey

There are several Operating Systems developed for educational purposes by different universities. Some of them are mentioned below.

The Xv6 [3] is a Unix-like teaching operating system developed for MIT's operating systems course. It is written in ANSI C and runs on multiprocessor x86 machine. It has multiprocessor support by handling concurrency with lock and threads, unlike special case solutions for uni-processors such as enabling/disabling interrupts.

XINU [4] (Xinu Is Not Unix) is a student built operating system for understanding the inner workings of an OS developed at Purdue University. The course topics include memory management, scheduling, concurrent processing, device management, file systems, etc.

Nachos [5] (Not Another Completely Heuristic Operating System) is an instructional teaching software developed by University of California, Berkeley. It consists of Machine, Threads and User Programs classes. The machine has registers, memory and CPU, and roughly approximates the MIPS architecture. The threads run in a virtual address space and has various states (ready, running, blocked, created) and methods (sleep, yield, fork). Nachos requires that code is in "Noff" format and a MIPS Simulator executes code for user program running on top of the Nachos Operating System.

Topsy [6] (Teachable OPerating SYstem) is another small operating system desinged for teaching purposes at ETH Zurich. The system works on real x86 hardware.

Other similar teaching operating systems include Minix [7], GeekOS [8], TempOS [9], OS-161 [10], SimOS [11], PintOS [12]. The survey article [13] reports that Nachos is the most popular among the systems discussed here. A recent survey [14] indicates that the Xv6 system has gained popularity recently.

1.2   Problem Definition

The objective of the project are the following :-

- Conduct a design review of the existing eXpOS architecture.

- To extend the existing XSM architecture to a dual core system with identical registers.

- Extend the eXpOS Operating System to run on the extended architecture utilizing the multi-processing facility.

- Design and implementation of support tools like compilers, assemblers, etc., for the extended machine architecture.

Chapter 2

Design

This project focuses on improving the existing eXpOS system and adding multiprocessor support to the system. The first phase of the project reviewed the existing eXpOS system critically for design errors and proposing fixes to the design. Subsequently, enhancements the architecture and well as the OS to support multiprocessing is discussed.

2.1   Improvements to existing eXpOS Design

A design review of the existing eXpOS architecture was done and the following modifications were introduced:

1. **Design error:** If the *Fork* system call is called by a process which doesn't have heap pages, the heap pages will not be shared by the parent and the child as any subsequent page fault will allocate different pages for their heap and thus, violating the sharing semantics.
   Fix: The *Fork* system call was modified to allocate heap pages (if not allocated before) for the parent and thus, child process will share the heap pages after forking. The Exception handler was modified to allocate both the heap pages when a page fault occurs due to heap, instead of one page.

2. **Design error:** In the *Fork* system call, the parent process first acquires a PCB entry by invoking the *Get Pcb Entry* function in *Process Manager*. It then calls *Get Free Page* function in *Memory Manager* to allocate 3 pages for Stack/User Area Pages for the child process. During the acquiring of the free pages, the parent process could go to WAIT_MEM state (when free pages are not available), which may cause another process to acquire the same child process.
   Fix: A new state called ALLOCATED was introduced and the child process is changed to ALLOCATED state after acquiring it.

3. **Design error:** In the existing design of eXpOS, there was a possibility that a swapped out process will never be swapped in (a process is only swapped in when its state is READY) and resulting in only the idle process being scheduled.
   Fix: A swapper daemon is introduced as a new process similar to the idle program, with PID=15, and the *Pager Module* is called from the context of this process. This modification fixes the above problem, as there will always be two processes to perform the swap operation. To add fairness for the swapped out processes, if the TICK field of any swapped out process exceeds a threshold, that process is swapped in after swapping out another process in memory.

4. **Design error:** During an exception, the machine didn't store the contents of EIP, EC, EPN and EMA registers and pushed the IP value on top of the user stack.
   Fix: The exception handler hardware was corrected to perform the appropriate actions. The eXpOS roadmap and documentation were also updated for the same.

5. Other modifications include:

   - Corrections were done in the roadmap and documentation in accordance with the above changes in the eXpOS website.

- The user interface of the XFS-Interface was updated to the current eXpOS architecture.

- In XSM debugger, new commands like step n, list and watch-points, and commands for displaying data structures were added, for easier debugging.

2.2   NEXSM Architecture Specification

NEXSM is an extension of the XSM architecture with a dual core feature. The machine has two identical cores with the same set of internal registers sharing a common memory. All registers in XSM are present in both the cores. Additionally, NEXSM cores contain an additional register called the **core flag**. A few additional privileged instructions provide primitives for synchronization between the two cores. One of the processors is called the **primary core** and the other called the **secondary core**. The machine can operate in two modes – **active** mode and **reset** mode. In reset mode, the secondary is non-functional. The mode in which the machine operates can be controlled by the primary using a pair of special privileged instructions – START and RESET.

2.2.1   Additional Registers

The only additional register in NEXSM, that is not present in XSM, is the core flag (CORE). The core flag is a **read-only machine register**. The content of the core flag is set to 0 in the primary core and 1 in the secondary core. The core flag allows a program to test whether it is currently executing in the primary or the secondary.
Usage Example: JZ CORE, [Address]
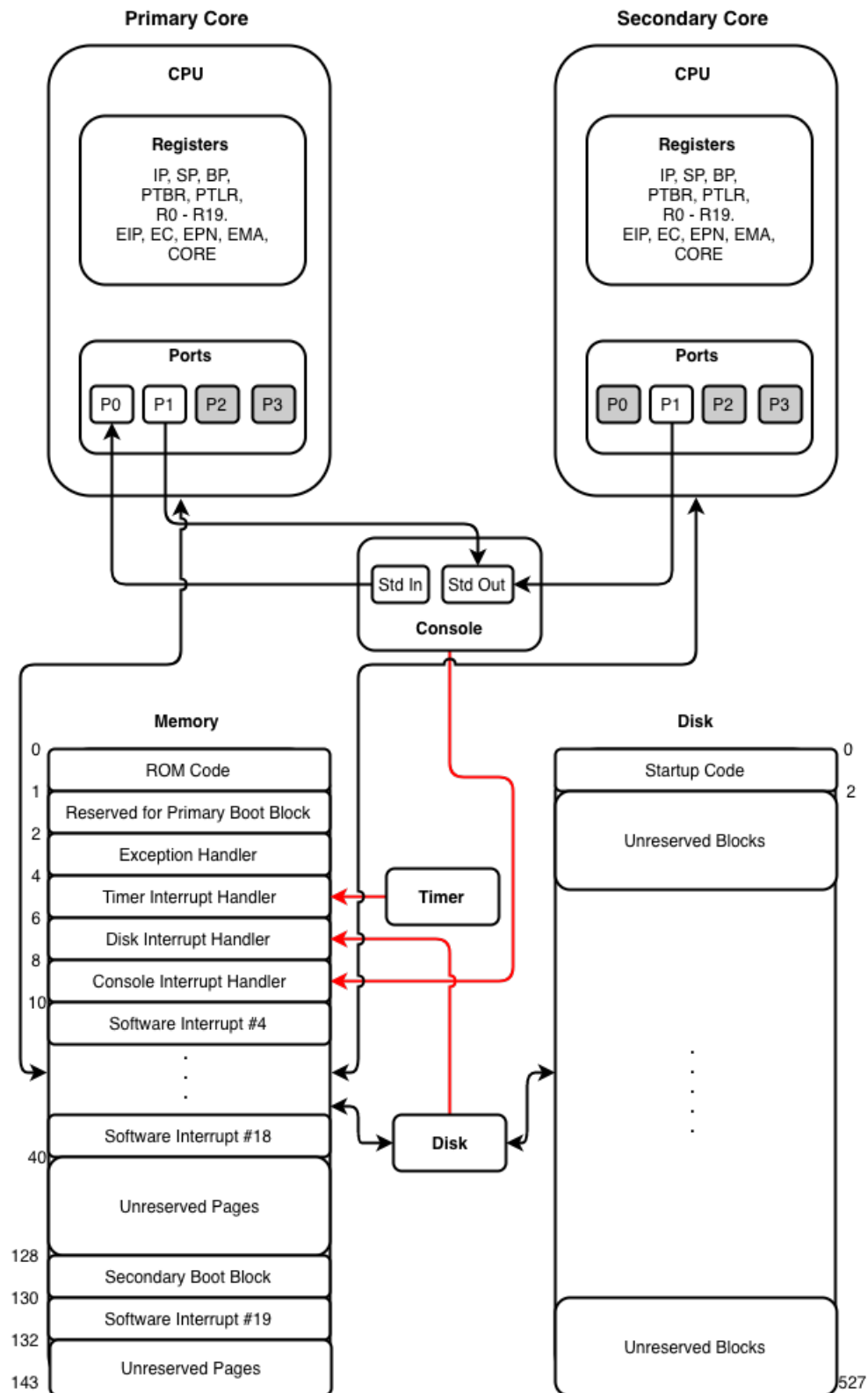
Figure 2.1: NEXSM Architecture

2.2.2   Additional Privileged Mode Instructions

- **Test and Set Lock**

  Syntax: TSL Rj, [loc]

  Semantics: The contents of the memory location (loc) is copied to register Rj. The value of [loc] is set to 1. This instruction is **atomic**. That is, when one of the cores is executing the TSL instruction, the memory bus is locked to avoid the other processor from simultaneously accessing [loc].

- **Dual-Core initialization**

  Syntax: START

  Semantics: If this instruction is executed by the primary while the machine is in reset mode, then secondary core starts parallel execution at the starting address of memory page 128 (physical address 65536). The START instruction is ignored if executed when the machine is in active mode.

- **Reset instruction**

  Syntax: RESET

  Semantics: The instruction, when executed in active mode sets the machine to reset mode. The instruction is ignored if executed in active mode.

2.2.3   Disk Organization

NEXSM has 16 additional blocks of disk space, with block numbers 512 to 527.

2.2.4   Memory Organization

NEXSM machine has 144 memory pages (as against 128 pages of XSM). The memory organization of pages 0 to 127 are exactly as in XSM. The

organization of the remaining 16 pages are as follows:

- Pages 128 and 129 are reserved for loading the bootstrap code for the secondary core.

- Pages 130 and 131 are reserved for an additional software interrupt INT 19.

- Pages 132 to 143 are available as free memory.

2.2.5 Dual-Core Bootstrap

1. When powered on, the machine starts in reset mode. The primary core starts execution and the secondary core is non-functional. Here, the functioning is similar to XSM.

2. When the machine is powered on, the primary executes a bootstrap code which loads the initialization code for the secondary core into memory page 128.

3. The secondary core starts execution when the primary core executes a START instruction.

4. Upon execution of the START instruction, the machine enters the active mode. The START instruction sets the IP value of the secondary core to physical address 65536 (page 128) and secondary core is powered on.

5. When the machine is running in active mode, if a RESET instruction is executed by either the primary or the secondary, then the machine goes back to reset mode and the secondary stops execution.

2.2.6 Interrupts and Exceptions

The way NEXSM machine enters privileged mode is similar to XSM, i.e., only when a software/hardware interrupt or exception occurs. The details are as follows:

1. The disk and the terminal interrupts apply to the primary core only.

2. Software interrupts, exceptions and timer interrupt applies to both the cores.

3. NEXSM permits an additional software interrupt INT19.

2.3 eXpOS Design for NEXSM

As NEXSM is only a dual-core extension XSM, the user interface of eXpOS undergoes no modifications. But the design of eXpOS changes when ported to the dual-core machine.

2.3.1 Modifications to Application Interface

The application interface of eXpOS undergoes minor modification when moving to NEXSM. The version of eXpOS running on NEXSM supports an additional software interrupt, INT 19. There are four system calls that gets mapped to INT 19 routine. Currently, these system calls are unused but could be used for testing future enhancements to the system. These are shown in Table 2.1.

2.3.2 Disk Organization

NEXSM disk has 16 additional disk blocks (block numbers 512 to 527). The Disk Organization of the new blocks is shown in Table 2.2.

System Call Name	Function Code	System Call Number
Test4	"Test4"	100
Test5	"Test5"	101
Test6	"Test6"	102
Test7	"Test7"	103

Table 2.1: INT 19 System Call Interface

Block Number	Contents
512 - 513	Secondary Bootstrap Code
514 - 515	Interrupt 19 Routine
516 - 517	Module 8: Access Control Module
518 - 519	Module 9: TestA Module
520 - 521	Module 10: TestB Module
522 - 523	Module 11: TestC Module
524 - 527	Unallocated

Table 2.2: Disk Organization

2.3.3   Memory Organization

NEXSM has 16 additional pages of memory (pages 128 to 143). The Memory Organization is shown is Table 2.3. The first four new pages are reserved by the machine. eXpOS reserves the next eight pages for storing OS code, while the remaining four pages are not used. Module 8 is called **Access Control Module** and contains code for synchronization between the two cores. Module 9, Module 10 and Module 11 are called TestA Module, TestB Module and TestC Module, respectively, and the present design doesn't use them. The OS maintains an **Access Lock Table** to hold the access lock variables.

Block Number	Contents	Word Address
57	Access Lock Table	29576 - 29583
128 - 129	Secondary Bootstrap Code	65536 - 66559
130 - 131	Interrupt 19 Routine	66560 - 67583
132 - 133	Module 8: Access Control Module	67584 - 68607
134 - 135	Module 9: TestA Module	68608 - 69631
136 - 137	Module 10: TestB Module	69632 - 70655
138 - 139	Module 11: TestC Module	70656 - 71679
140 - 143	Reserved for Future Use	71680 - 73727

Table 2.3: Memory Organization

2.3.4 Design Policies

The fundamental issue to be resolved while extending the OS to a dual-core machine is to ensure that concurrent updates of OS data structures from the two cores do not leave the OS in an inconsistent state. Here we impose a few simple to implement design level restrictions on the level of parallelism permitted so that a simple and comprehensible design is possible. The constraints imposed are the following:

Policy 1: *Atomicity constraints:*

1. A single process will never be scheduled simultaneously on both the cores. The scheduler will be designed to ensure this policy.

2. Only one core will run scheduler code that involves updates to kernel data structures at a given time. This makes implementation of the first policy straight-forward.

3. Only one core will be execute the critical section at a time. Upon entry into a system call the kernel checks whether the other core is running

critical kernel code. If that is the case, then the kernel waits for the other core to finish the critical code before proceeding.

Policy 2: *Hold and Wait conditions:*

A process, after acquiring an access lock will quickly perform the required action and release the lock immediately, before being scheduled out. Moreover, a second access lock will be acquired only after releasing the first. These constraints ensure that there will be no hold and wait or circular wait for access locks.

2.3.5   Access Lock Table

An additional layer of access locking is required to ensure the atomicity constraint is satisfied during access/update to OS data structures. The OS maintains these access lock variables in the Access Lock Table. As shown in Table 2.3, 8 words are allocated for the Access Lock Table, of which the last five words are unused. The various access lock variables are described in Table 2.4. The access lock variables are set before the execution of a critical section code, and are reset after its completion.

2.3.6   Access Control Module

The Access Control Module contains functions that implement atomic set and reset operations on the access lock variables. The functions present in the Access Control Module are shown in Table 2.5. The acquire lock functions are implemented using TSL instructions. If the other core has acquired the access lock variable, then the kernel waits for the release of the lock before proceeding. The general locking algorithm is:

```
AcquireLock() {
    ....
    ....
```

Field	Function
KERN_LOCK	Common access lock to be set before running any critical kernel code other than scheduling. Before performing any kernel function, this lock must be set by the kernel module/interrupt handler so that the other core waits till the critical action is completed.
SCHED_LOCK	Access lock to run the Scheduler Module. If one core has set the SCHED_LOCK in the Scheduler Module, the other core runs in a busy loop until execution of the Scheduler Module is completed.
GLOCK	A general purpose lock variable that is left unused for future use.

Table 2.4: Access Lock Table

```
    while ( tsl ( LockVarAddress ) == 1 ) {
        continue;
    }
}
```

Function Name	Arguments	Return Value
Acquire Kernel Lock	Nil	Nil
Acquire Scheduler Lock	Nil	Nil
Acquire Glock	Nil	Nil
Release Lock	LockVarAddress	Nil

Table 2.5: Access Control Module

2.3.7 Other Design Modifications

LOGIN, SHELL and SWAPPER DAEMON can only be run on the primary core. The *Pager Module* is invoked only from the primary core. In

dual-core operation, it might be required that both the cores schedule the IDLE process simultaneously. To satisfy the atomicity constraint, a new process called IDLE2 with (PID=14) is created. The scheduler will be modified to run IDLE2 on the secondary core whenever it finds that no other process can be scheduled. The standard IDLE (PID=0) will be scheduled under similar circumstances in the primary core. The primary will never execute IDLE2 and the secondary will never run IDLE. IDLE2 is run on the secondary when no other process is READY, or the OS is running *Pager Module* or *Logout* System Call on the primary core.

2.3.8 Boot Procedure

NEXSM specification stipulates that the secondary code bootstraps from the physical address 65536 (page 128) upon execution of the START instruction from the primary core. Hence, the bootstrap routine of the primary core must load the bootstrap code of the secondary core from disk block 512 to memory page 128 before issuing the START instruction. The START instruction is issued at the end of normal bootstrap by the primary core. The secondary bootstrap code will schedule the IDLE2 process for execution (setting its state appropriately) and from there normal dual-core execution starts.

2.3.9 Implementation Plan

The major changes to eXpOS are:

- Upon entry into a system call or exception handler, either from an application or from the scheduler, *AcquireKernLock()* function must be invoked. The lock must be released before invoking the scheduler or switch back to user mode using *ReleaseLock(KERN_LOCK)* function.

- The scheduler module must be modified to invoke *AcquireSchedLock()* function before initiating scheduling actions. Upon completion of schedul-

ing actions, the scheduler must release the lock by invoking *Release-Lock(SCHED_LOCK)* function before setting any process into execution.

- When the scheduler running on the secondary core finds that paging was initiated from the primary core, it will simply schedule IDLE2.

- The *Logout/Shutdown* system call will be invoked only from primary core as the SHELL process will be scheduled to run only on the primary. When the scheduler running on the secondary core finds that *Logout/Shutdown* system call is initiated from the primary core, it will simply schedule IDLE2.

- System Status Table is modified to include CURRENT_PID2 (stores the PID of the currently running process on the secondary core) and LOGOUT_STATUS (sprecifies whether Logout is initiated on the secondary core).

Chapter 3

Implementation

3.1 Modifications to ExpL

The ExpL compiler is updated to provide support for the additional software interrupt, INT 19. The ExpL library is modified to add the four new system calls, Test4, Test5, Test6 and Test7.

3.2 Modifications to SPL

A new register (CORE flag) and three additional keywords, namely *tsl*, *start* and *reset* are added. The additional instructions in NEXSM include:

- **TSL Expression**
  Syntax: **tsl (ADDRESS)**
  Semantics: The contents of the memory location specified by AD-DRESS is returned and the value at ADDRESS is set to 1. This statement translates to a sequence of instructions that uses the TSL machine instruction.

- **START Statement**
  Syntax: **start;**

Semantics: The **start** instruction when executed from primary core of the NEXSM machine will start the secondary core into parallel execution. This statement translates to the START machine instruction.

- **RESET Statement**
  Syntax: **reset;**
  Semantics: The **reset** instruction when executed from primary core of the NEXSM machine will freeze the secondary core. This statement translates to the RESET machine instruction.

3.3 Modifications to XFS-Interface

A few commands were modified in the XFS-interface to load primary and secondary OS startup code, INT 19 code and Module 8-11 code to their recognized XFS disk blocks, as shown in Table 2.2.

3.4 Modifications to XSM Simulator

The XSM machine simulator was upgraded to the NEXSM machine simulator, which adheres to the NEXSM architecture specifications. The NEXSM debugger contains all the commands similar to the XSM debugger, but displays the contents of both cores, instead of a single core. A new command for displaying the Access Lock Table is also added.

3.5 Testing and Deployment

The system was implemented and tested against test cases designed in the existing eXpOS system to ensure that the system runs successfully in the the dual-core NEXSM machine.

We note that, except for the complexity in the design, the amount of extra code that had to be added to the exiting eXpOS system was only approximately 500 lines of XSM assembly code, showing that the existing eXpOS design was easily adaptable to a dual-core extension, though such an extension was not planned when the design was proposed.

All the the components of the modified eXpOS system has been updated on the project site [1]. A new stage (stage 28) has been added to the project implementation roadmap for guiding the students through the steps of dual-core OS implementation. The system is expected to be deployed for laboratory use during Monsoon 2019.

Chapter 4

Conclusions and Future work

This chapter summarizes some experimental observations and discusses some of the future enhancements to the system.

4.1 Experimental Observations

The OS was run against test cases of concurrent execution of processes in single core and dual-core. The number of execution cycles in various test cases for single core and dual-core execution are summarized in Table 4.1, Table 4.2 and Table 4.3.

Table 4.1 shows the running times in single core and dual-core operations of a merge sort program which concurrently sorted sections of a shared sequence of integers stored in the heap and merged them sequentially. The program used semaphores to synchronize the merge. The table shows the program ran with 2, 4 and 8 concurrent processes for sorting sections. The data shows that two core operation resulted in reduction of the running time by at least 33%.

No. of processes	Single core	Dual-core	% decrease
2	6847	5123	33
4	6992	5218	34
8	7281	5330	27

Table 4.1: Observations with shared heap

Table 4.2 shows the running times in single core and dual-core of a program that forked 2, 4 and 8 concurrent CPU bound processes which did neither share memory nor used any synchronization. Clearly such process can be expected to be considerably faster when parallelism is available, and the experiments substantiate this prediction. The two core execution results in reduction of running time by 45% in this test case.

No. of processes	Single core	Dual-core	% decrease
2	1084	582	46
4	2117	1097	48
8	4273	2345	45

Table 4.2: Observations with no sharing

Table 4.3 shows the running times in single core and dual core of a program that forked 2, 4 and 8 concurrent I/O bound processes that accessed the disk of the machine. Since the I/O operations result in a longer idle time for the processes, the utility of parallel execution is expected to diminish, as demonstrated by the experiment. In this case, when the number of concurrent processes increases, the efficiency is seen to increase substantially. This is because as the number of processes increases while some process waits for disk, the other processes make progress using parallelism available. We see that the reduction in running time increases from 16% with two processes to 63% with eight processes.

No. of processes	Single core	Dual-core	% decrease
2	3481	2922	16
4	7636	4294	44
8	12526	4624	63

Table 4.3: Observations with file sharing

4.2   Suggestions for future work

We propose the following enhancements to the system which may be considered in the future.

- Optimizing the access locks using OS data structure locks, instead of a Kernel Lock.

- Adding a directory structure in eXpFS file system.

- Adding crash recovery to the eXpFS file system.

- Conducting performance comparison between single core and dual-core OS operations.

References

[1] eXpOS Team, "Experimental operating system," 2018.

[2] eXpOS Team, "Experimental operating system," 2013.

[3] R. Cox, F. Kaashoek, and R. Morris, "Xv6, a simple unix-like teaching operating system," 2018.

[4] J. W. Carissimo, "XINUan easy to use prototype for an operating system course," *ACM SIGCSE Bulletin*, vol. 27, pp. 54–56, 1995.

[5] W. A. Christopher, S. J. Procter, and T. E. Anderson, "The Nachos instructional operating system," in *Proceedings of the USENIX Winter 1993 Conference*, p. 4, 1993.

[6] G. Fankhauser, C. Conard, E. Zitzler, and B. Plattner, "Topsy - a teachable operating system," tech. rep., Computer Engineering and Networks Laboratory, ETH Zurich, 2001.

[7] A. S. Tanenbaum, "A unix clone with source code for operating systems courses," *ACM SIGOPS Operating Systems Review*, vol. 21, pp. 20–29, 1987.

[8] D. Hovenmeyer, J. Hollingsworth, and B. Bhattacharjee, "Running on the bare metal with GeekOS," in *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, pp. 315–319, 2004.

[9] R. S. Pinto, P. Nobile, M. Edwin, L. P. Jnior, H. J. Luz, and F. J. Monaco, "Operating system from the scratch: A problem-based learning approach for the emerging demands on os development," *Procedia Computer Science*, vol. 18, pp. 2472–2481, 2013.

[10] D. A. Holland, A. T. Lim, and M. I. Seltzer, "A new instructional operating system," in *Proceedings of the 33rd SIGCSE technical symposium on Computer science education*, pp. 111–115, 2002.

[11] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta, "Complete computer system simulation: the SimOS approach," *IEEE P&DT*, vol. 3, no. 4, pp. 34–43, 1995.

[12] B. Pfaff, A. Romano, and G. Back, "The pintos instructional operating system kernel," in *Proceedings of the 40th ACM Technical Symposium on Computer Science Education*, SIGCSE '09, pp. 453–457, 2009.

[13] C. L. Anderson and M. Nguyen, "A survey of contemporary instructional operating systems for use in undergraduate courses," *J. Comput. Sci. Coll.*, vol. 21, no. 1, pp. 183–190, 2005.

[14] S. Giraddi, P. Kalwad, and S. Kanakareddi, "Teaching operating systems - programming assignments approach," *Journal of Engineering Education Transformations*, vol. 31, no. 3, pp. 69–74, 2018.

[15] A. Tanenbaum, *Operating Systems: Design and Implementation*. Prentice-Hall, 1987.

[16] M. J. Bach, *The Design of the UNIX Operating System*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1986.

[17] N. Nisan and S. Schocken, *The Elements of Computing Systems: Building a Modern Computer from First Principles*. The MIT Press, 2008.

[18] C. Crowley, *Operating Systems: A Design-Oriented Approach.* McGraw-Hill, 1996.

[19] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts.* Wiley Publishing, 8th ed., 2008.