CS4090 Project
Report

eXperimental Operating System

Redesign XSM architecture and implement eXpOS, Simulator, Debugger, System Level Compiler and Application Level Compiler.

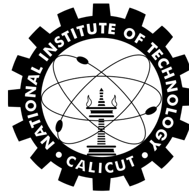
Submitted in partial fulfillment of the requirements for the award of the degree of

Bachelor of Technology
in
Computer Science and Engineering

Submitted by

Roll No	Names of Students
B120458CS	ANJANA BABU
B120012CS	CHRISTIN V. JOSE
B120152CS	LENY W. V.
B120363CS	KURIAN JACOB

Under the guidance of
Dr. K. Muralikrishnan



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

Department of Computer Science and Engineering

National Institute of Technology Calicut

Certificate

This is to certify that this is a bonafide record of the project presented by the students whose names are given below during Monsoon/Winter and Year 2015-2016 in partial fulfilment of the requirements of the degree of Bachelor of Technology in Computer Science and Engineering.

Roll No	Names of Students
B120458CS	ANJANA BABU
B120012CS	CHRISTIN V. JOSE
B120152CS	LENY W. V.
B120363CS	KURIAN JACOB

Dr. K. Muralikrishnan
(Project Guide)

Date: February 21, 2016

Abstract

¡Abstract here¿

Contents

Problem Definition 1

Introduction 2

1 SPL 3

2 Revised architecture 6

3 Enhancements to XOS 8

4 XSM Simulator 10

5 XFS Interface 11

Conclusion 12

Acknowledgements 13

References 14

List of Figures

2.1 The new architecture. 7

Problem Definition

Redesign XSM architecture and implement the existing Experimental Operating System. Incorporate the concept of asynchronous disk transfer and dynamic memory allocation. Addition of new state blocked for the process and make necessary changes in accordance with it. Introduction of disk and console interrupts and special ports for I/O operations. Modify the process virtual space to support dynamic link library.

Introduction

Project **eXpOS** or **eXperimental Operating System** is an educational platform to develop an operating system. It is an instructional tool for students to learn and implement OS data structures and functionalities on a simulated machine called **XSM (eXperimental String Machine)**. The OS is programmed using a custom language known as **SPL (System Programmer's Language)** and application programs, which run on the OS, are programmed using **ExpL (Experimental Language)**. **XFS Interface (eXperimental File System Interface)** is an external interface to access the filesystem of the eXpOS. The interface can format the disk, load/remove files, list files and copy blocks to a UNIX file.

Chapter 1

SPL

SPL or *System Programmer's Language* in reality is not a high level programming language, but an enhanced assembly language programming system for writing kernel mode programs for the XSM machine. This language is useful for implementation of an OS on top of the XSM machine. The language is minimalistic and consists only of very basic constructs. Programming using SPL requires an understanding of the underlying XSM architecture and operating system concepts.

SPL has been improved to support the new features provided by the architecture.

Registers
SPL allows the use of 25 registers (`R0-R15, BP, SP, IP, PTBR, PTLR, EIP, EC, EPN, EMA`) and 4 ports (`P0-P3`). `P0` and `P1` are used for standard input and standard output respectively.

Improved Statements
`goto` statement simulates an unconditional jump. `backup` and `restore` translates to `BACKUP` and `RESTORE` machine instructions respectively. The asynchronous `load` and `store` translates to `LOAD` and `STORE` machine instructions respectively.

Labels
SPL now supports labels. SPL does not officially support function calls, but a limited functionality can be achieved using labels and the `call` statement by following some conventions.

Modules

The concept of modules was introduced. From now on, each SPL program will be considered as a module. A module consists of a maximum of 1024 words which includes both the space allocated for code and data. The SPL compiler translates an SPL source program to a target XSM assembly module.

Each SPL module is designed to occupy a maximum of two pages of continuous memory in the XSM machine. A module typically contains protected mode code that carries out certain functions as determined by the OS programmer. The following suggests certain programming conventions which are recommend while designing SPL modules:

1. Each module shall be designed to have a single entry point, which is generally the starting memory address of the module in the memory. It is not recommended to transfer control directly into a location within a module from other modules.

2. Arguments to a module (typically includes the function code indicating which function within the module is to be invoked, as well as the parameters to that function) and return values shall be passed either through registers R0, R1, R2.. or through stack as required.

   The SPL compiler given here uses the constants (given in constants.spl file) MOD_0 to MOD_7 as starting address of eXpOS kernel modules, which are (optionally) loaded into various pre-defined memory pages of the XSM machine when eXpOS starts up. In addition to these, all interrupt service routines can be programmed as SPL modules and loaded to the corresponding interrupt service routine locations in memory. The SPL predefined constants INT_0 to INT_18 can be used to invoke these modules.

3. A module may be entered as a result of:

   (a) Execution of a software interrupt from an application (unprotected mode) program.

   (b) Occurrence of a hardware interrupt/exception while an application was executing.

   (c) Call from another module.

   Case a) In this case, the arguments to the module are passed through the application program stack. The return values are also passed

through the same stack. The convention is that the application must save the state of its registers before making the call. (For instance, the eXpL compiler will save the context in the user stack before invoking a software interrupt). Thus, the interrupt routine need not concern itself about saving the context of the application and can use the registers R0-R15 without saving them. However, the application is not expected to save SP, BP registers before the call, and the module must save them for future return.

However, in this case, it is strongly recommended that the module allocates its own stack for local memory allocation and not use the application's stack. This is to avoid potential user level hacks into kernel mode interrupt modules through the stack.

Case b) This case applies to the exception handler, timer interrupt routine, disk interrupt routine and the console interrupt routine. The difference here from case a) is that the application does not have control over the transfer of control, and hence would not have saved its context. Thus, in this case, the module must save the register context of the application in its own stack (or elsewhere in the memory) before using the registers and must restore the context before returning to the application.

Case c) In this case, since the caller and the callee are both executing in protected mode, the same stack can be used. The recommended convention is that the caller must save its register context into the stack before making the call.

Chapter 2

Revised architecture

Register Set
Register set has been revamped to contain 20 general purpose registers (R0-R19) instead of 8 general purpose registers (R0-R7), 16 kernel registers and 4 temporary registers. 4 ports (P0-P3) have also been introduced for channelling I/O operations. Exception Flag Register (EFR) has been replaced with Exception Instruction Pointer (EIP), Exception Cause (EC), Exception Memory Address (EMA), Exception Page Number (EPN).

Instruction Set
The instruction set has been expanded to include asynchronous instructions `LOAD, STORE` and `IN.PORT` instruction has been added to support data transfer between registers and ports. Direct indexed addressing has been introduced for data transfer. `ENCRYPT` instruction to encrypt the contents of a given register. `BACKUP/RESTORE` instructions to save/restore the contents of registers to/from the stack easily.

Memory
Memory has been reorganised to contain the new interrupt handlers. The pages from 2-39 have been allocated for this purpose. The paging hardware is now improved. Each translation entry contains two more fields that indicate a dirty and write page.

Interrupts and Exception Handling
New hardware interrupts have been added to handle disk and console devices resulting in 4 hardware interrupts. Number of software interrupts have been increased to 15 from 7.

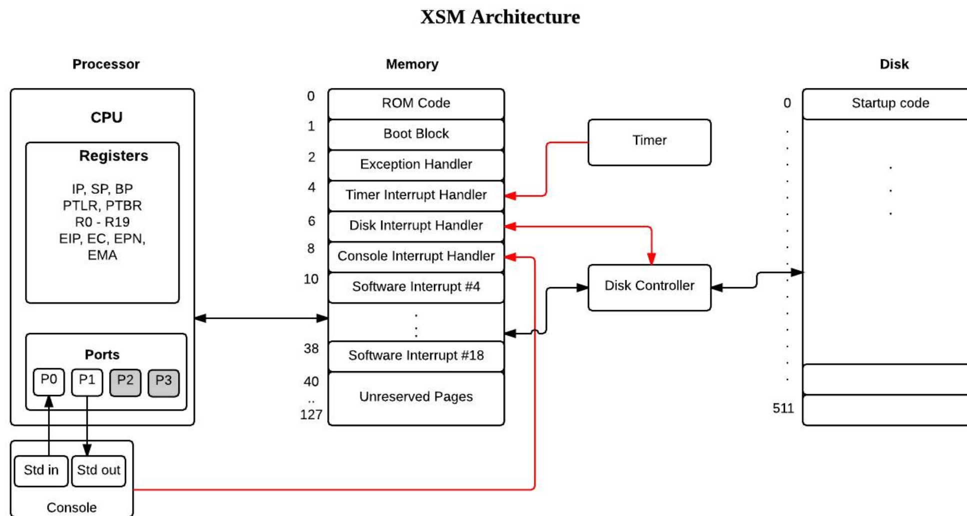


Figure 2.1: The new architecture.

Disk controller interrupt was added to support asynchronous disk access by the Operating System. Console interrupt, likewise support asynchronous console access. This was designed keeping in mind that whenever a process requires disk access, it will issue a disk access request and will get subsequently blocked. When the disk access is complete, an interrupt is fired which channels the data to the blocked process, and the process is unblocked.

The notion of asynchronous and synchronous read, write and load were introduced. In synchronous operation, the system waits for the operation to be completed whereas in asynchronous the system doesn't do so.

Chapter 3

Enhancements to XOS

In addition to the features in XOS, eXpOS supports multi-user mode, system calls for access control and synchronization, console and disk interrupts. Provision for asynchronous disk transfer and dynamic memory allocation is also added. eXpOS allocates 10 pages (`shared library:2, heap:2, code:4, stack:2`) per process.

OS Modules
OS Modules, such as Resource Manager, Process Manager, Memory Manager, File Buffer Manager, Device Manager and Context Switch Module, which are routines invoked from within system calls/ interrupts/ exception handler have been added.

High Level Library Interface
The High Level Library Interface is a unified system call and memory management interface provided in APL. The interface was introduced to reduce coupling between the underlying architecture and the high level interface. The interface is a mere wrapper for system calls and memory management routines, provided through the `exposcall` APL call. It takes a `fun_code`, which determines the action to be taken. If `fun_code` is negative, a memory management operation is invoked, otherwise, the corresponding system call is invoked.

Application Binary Interface
ABI or *Application Binary Interface* was designed to define the machine model, division of memory address space into regions, file format to be followed for the executable files, low level system call interface and low level runtime library interface. eXpOS executes only the files of XEXE Executable File Format.

Chapter 4

XSM Simulator

The XSM or *eXperimental String Machine* Simulator is used to simulate the XSM hardware. It is an interrupt driven uniprocessor machine. The machine handles data as strings. A string is a sequence of characters terminated by \0. The length of a string is at most 16 characters including \0. Each of these strings is stored in a word. The machine interprets a single character also as a string.

The simulator now has two more flags at invocation, for setting the intervals after which console (`--console`) or disk (`--disk`) interrupt is fired. The XSM is modified to support the new instructions, registers and ports.

Chapter 5

XFS Interface

XFS or *eXperimental File System* Interface is an external interface to access the filesystem of the eXpOS. The filesystem is simulated on a binary file called disk.xfs. The interface can format the disk, load or remove files, list files, copy files to and from the XFS file system to the UNIX file system and copy specified blocks of XFS disk to a UNIX file .

!!!...........to be modified...........!!!!

Conclusion

Acknowledgments

References

[1] Maurice J. Bach. *The Design of the UNIX Operating System*, Prentice-Hall, 1986.

[2] Andrew S. Tanenbaum, Herbert Bos. *Modern Operating Systems*, Pearson Education, 02-Apr-2014.

[3] eXpOS NITC Documentation, 2011-2015 Batch, `exposnitc.github.io`