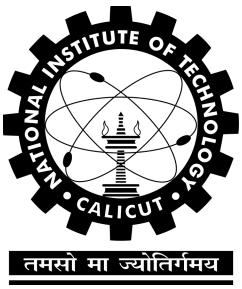
Major Project

EXPERIMENTAL OPERATING SYSTEM

B090084CS	Shamil CM
B090066CS	Sreeraj S
B090468CS	Vivek Anand T Kallampally

Under the guidance of
Dr. K. Muralikrishnan



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
National Institute of Technology Calicut
Calicut, Kerala 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 Winter 2013 in partial fulfilment of the requirement of the course of B.Tech Computer Science and Engineering.

B090084CS	Shamil CM
B090066CS	Sreeraj S
B090468CS	Vivek Anand T Kallampally

Faculty In-charge

Course Co-ordinator

Date:

Abstract

This paper introduces an operating system project that helps undergraduate computer science students acquire an elementary understanding of the practical aspects of an operating system. The specification of XOS (Experimental Operating System) has been laid out for students to build it from scratch in a bottom-up manner. XOS runs on a simulated machine hardware with a very simple instruction set and native filesystem. Unlike other common instructional operating systems, the complete development environment including custom programming languages, debugger, file system interface and a detailed implementation roadmap is provided.

Contents

Chapter 1

Introduction

Teaching operating systems has been a challenge at the undergraduate level. To tackle this problem several instructional operating systems like Nachos[2], OS/161[4], Pintos[5],
GeekOS [1] etc. have been developed by various universities. Nachos[2] has been one of the most popular instructional operating systems available and is being used in many institutes across the world [1]. Implementing Nachos is simple and it uses a mixed mode approach, where the operating system kernel is co-resident with the machine simulator and fused together as a single program. Nachos[2] and OS/161[4] runs on top of MIPS machine simulator. For these systems, a user's machine running on other platforms require cross compilers to MIPS.

Instructional operating systems like Minix and Xinu [1], provide a functional operating system on which modifications are to be done by students. Almost every other instructional operating system provides a skeleton of an operating system. However in XOS, only the specification has been laid out, and students learn to implement XOS from ground up using the tools provided. In this project, a simple high level language called APL (Application Programmer's Language) and its cross-compiler to XSM instruction set is provided to write user programs to test XOS. An XSM dependent language called SPL (System Programmers Language) and its cross-compiler is provided to program the OS itself. Most instructional operating systems use the UNIX filesystem for file management by the operating system. Instead, XOS provides a native file system known as XFS (Experimental File System). An interface between the UNIX filesystem and XFS filesystem is also provided.

XOS has features like multiprogramming, process management, a primitive filesystem and virtual memory. A sequence of stages are provided in a detailed roadmap which helps students to build XOS sequentially. Several simplifications have been made in XOS specification to make the system simple and manageable as a short-term project. These include absence of inter-process communication, device management, file caching, file permissions etc. Only the fundamental data structures and functions of a single-user, multiprogramming operating system has been retained. Limited support for process synchronization can be added to XOS as enhancements (refer section 4). The features that are omitted from XOS are not intended to be done in this platform. The project has been designed to provide an essential understanding of operating system concepts for a student undertaking a core theory course in operating systems at undergraduate level. XOS is not scalable and students intending to go in depth in topics like device management, file caching etc. are suggested to move on to more sophisticated platforms like Minix, GeekOS[1] etc.

In our experience with Nachos at junior year undergraduate operating systems laboratory course, we observed that students did not have the necessary programming expertise to comprehend the large code base of Nachos. We also observed that students faced a conceptual difficulty in understanding the separation between the operating system kernel and the machine simulator which are fused to-

gether as a single program in Nachos. XOS addresses these issues by allowing the student to build the operating system from scratch without having to learn an existing code base. The roadmap is designed in such a way that the student can read the roadmap and complete the project without any supervision. The project is expected to provide the student with a better insight into the operating systems concepts described in standard text books like [6]. Building XOS can be a supplementary project for a junior level undergraduate course in operating systems spanning over a 12 to 16 week term.

Chapter 2

Components

2.1 Introduction

The primary components of the project include a simulated machine hardware (XSM), file system (XFS) and the operating system (XOS). No code base for the operating system is provided and the operating system is completely implemented by the student. Apart from the primary components, various tools are provided as part of the development environment. They include languages like Application Programmer's Language (APL) and System Programmer's Language (SPL) and their cross compilers to XSM instruction set, XSM debugger, and a UNIX-XFS interface to transfer files between a UNIX machine and the XFS disk (the XFS disk is itself a UNIX file).

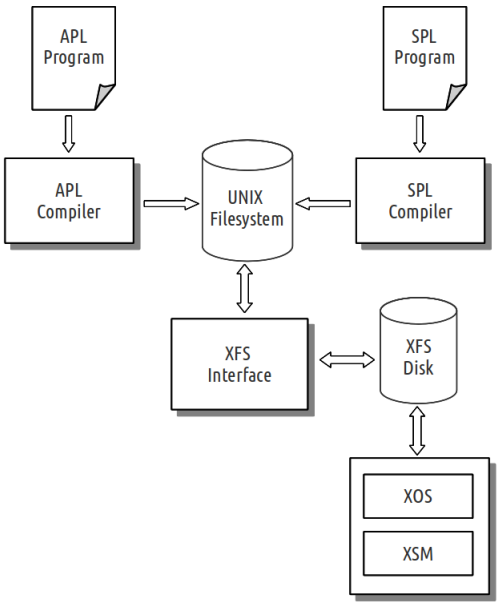


Figure 2.1: Components and their interaction

2.2 Experimental File System

The disk for XSM (which is a UNIX file) can be formatted using XFS or Experimental File System. XFS is the file system compatible with XOS. Since file system management is done by students building XOS, the disk organization must be easily understood and at the same time must give an insight on the data structures used in real file systems. Hence we chose to have a native file system for XOS.

The disk is formatted with this file system using the interface provided as part of the development environment. XFS is a simple file system with no directory structure. The data is organized into blocks of size equal to the page size in XSM memory. There are 512 blocks in XFS which holds the file and disk data structures, OS routines, user programs and data files. The various data structures in XFS include the Disk Free List which maintains information about used and unused blocks on the disk and the FAT or the File Allocation Table stores details of the files in the disk.

2.3 Experimental String Machine

XOS runs on a simulated machine hardware called XSM or Experimental String Machine. XSM uses an easy-to-understand native 2-address instruction set. The various components of the machine include registers, memory, timer and the disk. A UNIX file simulates the disk for XSM.

XSM has a timer which triggers after fixed number of instructions as compared to a timer interval in real machines. An instruction triggered timer was preferred over a clock-triggered timer to ensure that the timer interrupt is not invoked in between the simulation of a single instruction. Thus, an instruction in XSM is always atomic. Instructions are executed one after the other in a non-pipelined manner.

XSM has a memory of 64 pages. Size of each page is 512 words. A word is the smallest addressable unit in XSM as compared to a byte in MIPS. XSM is a string machine, and each word is stored internally as strings of size 16 characters. However, the XSM supports two data types, integer and strings and has instructions for both the data types. There are two privilege modes in XSM, the user mode and kernel mode. Switching between modes is done by instructions.

XSM instruction set supports load and store instructions to load data from the disk to memory and to store data from memory to disk respectively. Real machines usually implement transfer using a DMA (Direct Memory Access) controller which transfers data directly between disk and memory and signals the processor after the transfer is complete. This happens without intervention of the processor. However XSM, provides machine instructions to do this. It is a deviation from real machines and has been provided to avoid the complexity associated with device management by the operating system when more than one process is running. We decided to avoid device management and DMA altogether because it is beyond the scope of the project. We feel that such features may be attempted on more advanced platforms like Minix after the completion of this project.

XSM supports virtual memory management. The machine will transfer control to a specified location in memory where the exception handler is expected to reside upon encountering an exception after setting an EFR or the Exception Flag Register. Unlike MIPS machines[3], which has 3 registers for exception handling, XSM combines all three registers into a single register for simplicity. Exceptions in XSM include a page fault, illegal instructions or operands, illegal memory access and arithmetic exceptions.

XSM machine has capability of running an operating system capable of multiprogramming, file management and virtual memory on top of it. XOS has been designed to exploit the complete

capabilities of the XSM architecture, keeping in mind a simplistic design. The motivation behind the design of all components was sequentially building the concepts of operating systems and not on the intricate details associated with its implementation.

2.4 Experimental Operating System

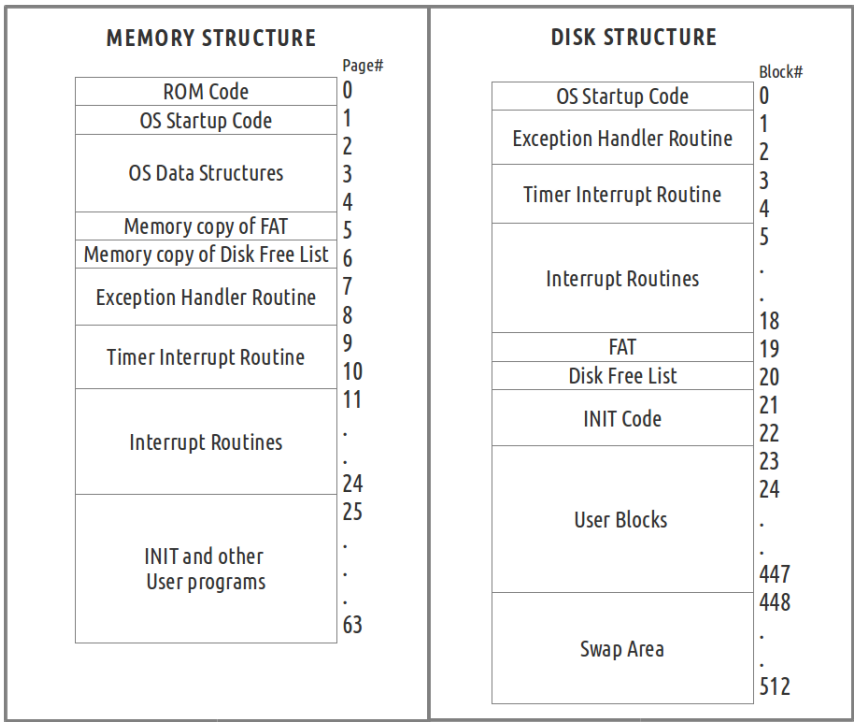


Figure 2.2: OS View of memory and disk

The specification for an Experimental Operating System or XOS is provided to the students. In this project, students will build XOS to meet the specification. XOS is a simulated operating system which runs on top of XSM which is a simulated machine hardware. The OS kernel unlike Nachos [2] resides in the memory of the simulated machine. The disk for XOS, which is a UNIX file, is formatted with the XFS filesystem. The disk permanently stores the OS routines and data structures as in real systems.

The various components of the operating system include routines like OS startup code, eight interrupt routines including the timer interrupt, the exception handler routine, and data structures like ready list of PCBs, per-process page tables, the system wide-open file table, memory free list, memory copy of disk data structures including FAT and the disk free list. The OS routines are to be programmed by the student building XOS and is loaded to the disk. The OS startup code which is also programmed by the student is responsible for loading all the other routines from disk to memory on OS startup. The OS startup code must be loaded in the first block of the XFS disk. When the system boots the bootstrap loader or the ROM code will copy this disk block to memory and pass control to it. The ROM code is hardcoded in the part of the machine memory which acts as the ROM. The OS routines will modify the memory-resident data structures, and make changes in the disk as and when required.

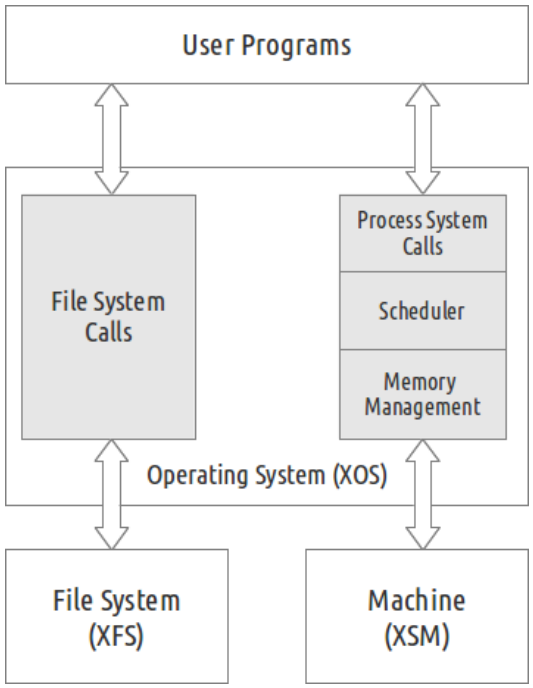


Figure 2.3: Components and their interaction

The various functionalities of XOS are process management, memory management and system calls. Process management includes scheduling and dispatching processes to the CPU. XOS is capable of multiprogramming (the ability to run more than one process simultaneously). Memory Management involves allocating memory for processes, demand paging (loading memory pages from the disk as and when required). The system calls provided by XOS include file system calls like Create, Delete, Open, Close, Read, Write, Seek and process system calls like Fork, Exec and Exit. Limited support for process synchronization has been suggested in the form of enhancements to XOS, through the implementation of system calls like Wait, Signal, Getpid, and Getppid. These system calls are mapped to 7 interrupt routines provided by the machine. The scheduler should be programmed and loaded into the timer interrupt routine. By specification, the scheduler of XOS follows a round-robin scheduling technique [6]. The page fault handler resides in the exception handler routine. For exceptions other than the page fault, XOS will exit the process. On a page fault exception, the exception handler must be programmed to implement a page replacement algorithm. The page replacement algorithm in XOS specification is the *second chance algorithm*[6]

XOS maintains a ready list of PCBs in memory. XOS limits the number of processes that can be run concurrently to 32. Each process has a per-process page table in memory. For each process the stack is of size 1 page and is present in the memory always. A process can have a maximum of 4 memory pages. Even though space for 32 processes to reside simultaneously in memory is not available, with pure demand paging, and swapping out of pages by the page replacement algorithm, this constraint can be overcome. The stack is never swapped out to the disk. When a OS routine returns back to the user program, the stack of the user process is used for passing the return address.

No code base for XOS is presented to the student. However a detailed roadmap which is divided into well-defined stages is provided so that the student can both understand the concepts and build the operating system to meet the specification.

Chapter 3

Development Tools

Various development tools are provided to the student for implementing XOS. Each of these tools and their purpose is described in the following sections.

3.1 Application Programmer's Language

Application Programmer's Language (APL) is a high-level language which can be used to write application programs to run on top of XOS. Its cross-compiler to XSM instruction set is provided as part of the development tools to the student. APL is a simple language which supports local and global variables and function calls. Interfaces to XOS system calls are provided through a family of built-in functions.

3.2 System Programmer's Language

System Programmer's Language (SPL) is a XSM dependent language which is used to program XOS itself. Its cross-compiler to XSM instruction set is also provided as part of the development tools. SPL unlike APL, is close to the machine instruction set. SPL statements can access most of the machine registers and memory directly. SPL has predefined constants specific to XOS. The SPL programmer can redefine these constants or define new constants. Apart from this, XOS provides *aliasing*, when registers can be assigned meaningful names for convenience.

3.3 Debugger

The XSM simulator can be run in the debug mode. In this mode, the machine will stop execution at predefined breakpoints in the XSM instructions loaded in the machine's memory. Breakpoints can be set through APL and SPL programs using the `breakpoint` instruction. From a breakpoint onwards, the execution can be single stepped, or continued to the next breakpoint in the debugger. Using the debugger, contents of registers, memory, XOS data structures etc, can be verified. The debugger has been modeled based on the GNU Debugger [7].

3.4 XFS Interface

The XFS disk is a UNIX file. XFS interface is a simple command-line interface that helps access this disk directly from your UNIX machine. XFS interface has functions to format the disk, display the contents of the disk, copy files from UNIX machine to the disk, view the disk free list etc. The OS routines and user programs are loaded to the disk before starting the OS using this interface.

Chapter 4

Roadmap

We felt the necessity for a roadmap to build XOS sequentially. A detailed roadmap divided into stages is provided on the onset to help students do the project. The first two stages help students get familiarized with the environment and development tools. Then the students sequentially start running a kernel program and then a user program on top of it. The next big step in the roadmap is when the student starts to run multiple programs and implements the scheduler. The initial stages are explained in detail. The students will learn more than what they do in these stages. The later stages include implementing the system calls, and virtual memory management. The final stage of the roadmap includes making a console for XOS. The entire roadmap is designed so that the project can be completed in a 12 to 16 week term. Moreover, instructions to implement enhancements to XOS which include system calls like Wait, Signal, Getpid, and Getppid and making a console for XOS are provided at the end of the roadmap. These system calls provide limited support for implementing process synchronization.

Chapter 5

Conclusions

The project is easier to implement compared to the existing instructional operating system due to the simplifications done, and the documentation. The project aims to be a better tool for basic undergraduate understanding of operating systems as it helps students get a feel of the elementary practical aspects of operating systems. It acts as a tool which will aid students to better comprehend the textbooks on operating systems or move on to a complex operating system platform later.

5.1 Acknowledgments

We thank the past contributers of the earlier versions of this project, Ajeet Kumar, Albin Suresh, Avinash, Deepak Goyal, Jeril K George, K Dinesh, Mathew Kumpalamthanam, Naseem Iqbal, Nitish Kumar, Ramnath Jayachandran, Sathyam Doraswamy, Sumesh B and Yogesh Mishra. We also thank Govind Nharekat and Nithin Mohan who are undergraduate computer science students at our institute, who were actively involved in testing out the system and providing valuable feedback.

5.2 Availability

The complete set of development tools, specification documents and roadmap is available at `xosnitc.github.com`. The entire source code is hosted at `http://github.com/xosnitc`. Students doing XOS project can join the mailing list `xos-users@googlegroups.com` and post queries.

Bibliography

[1] C. L. Anderson and M. Nguyen. A survey of contemporary instructional operating systems for use in undergraduate courses. *Journal of Computing Sciences in Colleges*, 21(1):183–190, 2005.

[2] W. A. Christopher, S. J. Procter, and T. E. Anderson. The nachos instructional operating system. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, pages 4–4. USENIX Association, 1993.

[3] J. Heinrich. *MIPS R4000 Microprocessor User's Manual*. MIPS technologies, 1994.

[4] D. A. Holland, A. T. Lim, and M. I. Seltzer. A new instructional operating system. *ACM SIGCSE Bulletin*, 34(1):111–115, 2002.

[5] B. Pfaff, A. Romano, and G. Back. The pintos instructional operating system kernel. In *ACM SIGCSE Bulletin*, volume 41, pages 453–457. ACM, 2009.

[6] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating system concepts*. J. Wiley & Sons, 2009.

[7] R. Stallman. *GDB manual: the GNU source-level debugger*, volume 675. Free Software Foundation, 1989.

Appendix A

Operating System (XOS) Specification

Contents

Chapter 1

Introduction

XOS (*Experimental Operating System*) is an experimental operating system which is designed to be run on the **XSM** (*Experimental String Machine*) architecture which is a simulated machine hardware. XOS is intended as an instructional tool to help students learn various aspects about operating systems.

XOS is programmed using a custom language, **SPL** (*System Programmer's Language*) which compiles to XSM compatible code. Application programs for XSM are written in **APL** (*Application Programmer's Language*).

The programs, data and operating system code is stored on a disk which has an **XFS** (*Experimental File System*) in it.

The various functionalities of XOS include

- **Process Management**, includes scheduling and dispatching processes to the CPU. XOS is capable of *multiprogramming* (the ability to run more than one process. simultaneously). Refer Chapter 3
- **Memory Management**, involves allocating memory for processes, demand paging (loading memory pages from the disk as and when required). Refer Chapter 4
- **System Calls**. XOS provides various system calls for the user processes to execute certain kernel level operations. Refer Chapter 6

Chapter 2

Memory Organization

The operating system organizes memory as given below:

Page No.	Contents	Word Address	# of words
0	ROM Code	0 – 511	512
1	OS Startup code / Scratchpad*	512 – 1023	512
2	Per-Process Page Tables	1024 – 1279	256
	Memory Free List	1280 – 1343	64
	System-wide Open File Table	1344 – 1471	128
	Unallocated	1472 – 1535	64
3	Ready List of PCBs	1536 – 2559	1024
4			
5	File Allocation Table	2560 – 3071	512
6	Disk Free List	3072 – 3583	512
7 – 8	Exception Handler	3584 – 4607	1024
9 – 10	Timer Interrupt Routine	4608 – 5631	1024
11 – 12	Interrupt 1 Routine	5632 – 6655	1024
13 – 14	Interrupt 2 Routine	6656 – 7679	1024
15 – 16	Interrupt 3 Routine	7680 – 8703	1024
17 – 18	Interrupt 4 Routine	8704 – 9727	1024
19 – 20	Interrupt 5 Routine	9728 – 10751	1024
21 – 22	Interrupt 6 Routine	10752 – 11775	1024
23 – 24	Interrupt 7 Routine	11776 – 12799	1024
25 ⋮ 63	INIT and User Programs	12800 – 32767	512 × 39 19968

Fig. 2.1: Outline of the main memory
*Page Number 1 (OS Startup Code) will be used as scratchpad after bootup

- **OS Startup code**, loads the INIT process to memory and sets up data structures like FAT, Disk Free List, and Memory Free List. It also loads

the Interrupt Routines and Exception handler from the disk to the memory. Refer Section 7.1

- **Per-Process Page Tables**, used for address translation of logical addresses to physical address. Refer Section 3.5
- **Memory Free List**, is a list of free memory locations in the memory. Refer Section 4.3
- **System-wide Open File Table**, contains a details of files which are opened by the processes. Refer Section 5.3
- **Ready List of PCBs**, is a list of Process Control Blocks, which indicates the ready and terminated processes. Refer Section 3.3
- **Memory Copy of File Allocation Table**, contains details about files stored on the disk, Refer Section 5.1
- **Memory Copy of Disk Free List**, contains details about used and used blocks in the disk, Refer Section 5.2
- **Exception Handler**, contains the kernel code to be executed during various exceptions, Refer Section 7.2
- **Timer Interrupt Routine**, contains the kernel code to be executed during a timer interrupt. Refer Section 7.3
- **Interrupt Routines**, contains kernel code to be executed during interrupts (1 to 7). Refer Section 7.4
- **INIT and User Programs**, is the memory space allocated for user programs in execution. Refer Section 3.7

Chapter 3

Process Management

3.1 Introduction

Any program in its execution is called a **process**. Processes will be loaded into memory before they start their execution. Each process occupies **at most 4 pages** of the memory. The processor generates logical addresses with respect to a process during execution, which is translated to the physical address. This translation is done by the machine using page tables.

The XSM architecture supports demand paging and so the machine does not fix the number of processes that can be run on it. However XOS has limited the number of process running simultaneously to 32, due to limitations in number of PCBs in the Ready List (Refer Section 3.3) and the number of Per-Process Page Tables (Refer Section 3.5)

3.2 Process Structure

A process in the memory has the following structure.

- **Code Area :** These are pages of the memory that contain the executable code loaded from the disk.
- **Stack :** This is the user stack used for program execution. The variables and data used during execution of program is stored in the stack. It grows in the direction of increasing word address. The location of the stack is fixed at the 4th page of the process.

Figure 3.1 shows the process structure.

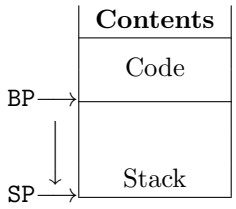


Fig. 3.1: Logical Address Space of a Process

3.3 Process Control Block (PCB)

It contains data pertaining to the current state of the process. The size of the PCB is **32 words**. Refer figure 3.2.

0	1	2	3	4	5	6	7 – 14	15 - 30	31
PID	STATE	BP	SP	IP	PTBR	PTLR	R0 – R7	Per-Process Open File Table	Free

Fig. 3.2: Structure of Process Control Block

3.3.1 Process Identifier (PID)

The process identifier is a number from 0 to 31, which identifies the processes in memory.

3.3.2 Process State

The process state corresponding to a process, indicated by `STATE` in the PCB stores the state of that process in the memory. A process can be in one of the following states.

- **0** for *terminated*, i.e. process has completed execution
- **1** for *ready*, i.e. process is waiting for the CPU to start execution.
- **2** for *running*, i.e. the process is currently running in the CPU

3.3.3 Registers

- **IP**: The word address of the currently executing instruction is stored in the `IP` (Instruction Pointer) register. The value of this register cannot be changed explicitly by any instruction.
- **BP**: The base address of the user stack is stored in the `BP` (Base Pointer) register.

- **SP**: The address of the stack top is stored in the `SP` (Stack Pointer) register.
- **PTBR**: The physical address of the Per-Process Page Table of the process is stored in the `PTBR` (Page Table Base Register).
- **PTLR**: The length of the Per-Process Page Table (No. of entries) is stored in the `PTLR` (Page Table Length Register). It is fixed as **4** for every process in XOS.

Each process has its own set of values for the various registers. Words 7 – 14 in the PCB stores the values of the registers associated with the process .

3.3.4 Per-Process Open File Table

The Per-Process Open File Table contains details of files opened by the corresponding process. Every entry in this table occupies 2 words. A maximum of 8 files can be opened by a process at a time, i.e. up to 8 entries in the PCB. It is stored in the PCB from words 15 to 30. Its structure is given below

1 word	*1 word*
Pointer to system-wide open file table entry	`LSEEK` position

Fig. 3.3: Structure of Per-Process Open File Table

For an invalid entry, the value of pointer to system wide open file table is set to -1 .

- The OS maintains a system wide open file table which contains details of all the files that are opened by processes (Refer Section 5.3). The entry in the Per-Process File Table points to the System-wide Open File Table entry corresponding to the file.
- It also stores the `LSEEK` position for the file, which indicates the word in the file to which the process currently points to for read/write operations.

3.4 Ready List

The list of PCBs stored in the memory is used as a Ready List by the operating system to schedule processes to CPU. The `STATE` in the PCB indicates whether a process is ready for execution or not. A new process in memory is scheduled for execution by circularly traversing through the list of PCBs stored in memory and selecting the first Ready process after the PCB of the currently running process in the list.

A maximum of 32 PCBs can be stored in the memory, and hence the maximum number of processes that can be run simultaneously is limited to 32. The PCB list is stored in pages 3 and 4 in the memory (words 1536 – 2559)

3.5 The Per-Process Page Tables

Every process in XOS has a Per-Process Page Table. A total of 32 PCBs and 32 Page Tables in total are available, which limits the number of processes that can be run to 32.

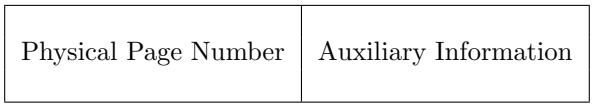


Fig. 3.4: Structure of a valid Page Table Entry

The Per-Process Page Table stores the physical page number corresponding to each logical page associated with the process. The logical page number can vary from 0 to 3 for each process. Therefore, each process has 4 entries in the page table. Per-Process Page Tables are stored in Page 2, words 1024 – 1279 in the memory (256 words = 32 processes × 4 entries)

When a process is loaded, the actual pages are not loaded into memory. In **demand paging**, the actual pages are loaded only when the pages are accessed for the first time (Refer Section 4.2). Once all pages are loaded, the first word of each entry contains the physical page number where the data specified by the logical address resides in the memory.

The second word contains **auxiliary information**. The first two bits of auxiliary information are reserved as *reference(R) bit* and *valid/invalid(V) bit*. The remaining bits are not used by XOS, but can be used for future enhancements. The details of bits in Auxiliary information is given below.

- **Reference Bit (R)**: Initially, this bit is set to 0 (unreferenced) by the machine. On a page access, this bit is set to 1 by the machine. This bit is used for page replacement by the OS.
- **Valid/Invalid Bit (V)** : This bit indicates whether the entry of the page table is valid or invalid. The *Valid/Invalid* bit has value 1 if the first word of this entry corresponds to a valid physical page number. It has value 0 if the entry is invalid.

  The first word of an invalid Per-process page table entry is either -1 (indicates that there is no physical page corresponding to the logical address) or a disk block number (the physical page corresponding to the logical address resides in this disk block and needs to be loaded to memory). The

0	1	2	...	15
R	V	\0	...	\0

Fig. 3.5: Structure of Auxiliary Information

Valid/Invalid bit is set by the OS. If memory access is made to a page whose page table entry is invalid, the machine transfers control to the Exception Handler routine, which is responsible for loading the correct physical page.

An example is given below

Physical Page Number	*Auxiliary Information (Reference and Valid Bit)*
36	01
311	00
-1	00
490	00

Fig. 3.6: Structure of Per-Process Page Table

In the above example :

- Reference bit of every entry is set as 0, indicating unreferenced
- The 1st entry is a valid page in memory as the valid bit is 1.
- The 2nd entry is invalid (valid bit is 0) and the disk block no corresponding to that entry is stored (311).
- The 3rd entry is invalid. There is no physical page associated with this logical address.
- The 4th entry is invalid and the disk block no stored is 490. This corresponds to a page in the swap area.

3.6 Multiprogramming

The operating system allows multiple processes to be run on the machine and manages the system resources among these processes. This process of simultaneous execution of multiple processes is known as *multiprogramming*.

To support multiprogramming in the system, the kernel makes use of the *scheduler* which is present in the Timer Interrupt Service Routine in Pages 9 and 10 of the memory.

3.7 INIT and User Processes

The `INIT` process is the first user program that is loaded by the OS after start up. The `INIT` and other user processes uses the memory pages 25 - 63 for execution (Code Area and Stack).

Chapter 4

Memory Management

4.1 Introduction

XSM uses a paging mechanism for address translation. XOS supports virtual memory, i.e. it supports execution of processes that are not completely in memory. It follows *pure demand paging* strategy for memory management. Pages are allocated as and when required during execution.

4.2 Paging

Paging is the memory management scheme that permits the physical address space of a process to be non-contiguous. Each process has its own page table (Refer Section 3.5), which is used for paging.

The Per-Process Page Table contains information relating to the actual location in the memory. Each valid entry of a page table contains the page number in the memory where the data specified by the logical address resides. The address of Page Table of the currently executing process is stored in `PTBR` and length of the page table is set to 4 in `PTLR` of the machine.

4.3 Memory Free List

The free list of the memory consists of 64 entries. Each entry is of size one word. Thus, the total size of the free list is thus 64 words. It is present in words 1280 to 1343 in memory. (words 256 to 319 of Page) of the memory. Refer Chapter 2. Each entry of the free list contains a value of either 0 or 1 indicating whether the corresponding page in the memory is free or not respectively. When a page is shared by more than one process, the entry stores the number of processes that share the page.

4.4 Virtual Memory

XOS allows virtual memory management, i.e. running processes without having all the pages in memory. It makes use of a backing store or ***swap*** in the disk to replace pages from the memory and allocate the emptied memory to another process. This increases the total number of processes that can be run simultaneously on the OS.

When a process starts executing, no memory pages are allocated for it. Initially its Per-process page tables are set with the block numbers of the disk blocks which contain the data blocks of the program. For each page table entry, the *Auxiliary Information* are intialized to 0 (invalid) and 0 (unreferenced). When a page is referenced for the first time, it triggers a page fault exception (since valid bit is set as 0). The *Exception Handler Routine* is responsible for loading the required page from the disk to the memory. This strategy of loading pages when accessed for the first time, is known as **Pure Demand Paging**.

On encountering a page fault exception, the Exception Handler Routine loads the required page from the disk to a free page in the memory. If no free page is available in the memory, a page replacement technique is used to select a victim page. The page replacement technique used in XOS is a *second chance algorithm* (Refer Silberschatz, Galvin, Gagne: Operating System Concepts) which uses the reference bits in the auxiliary information. The victim page is swapped out to to the disk (swap area) to accommodate the required page.

Chapter 5

Files

The operating system requires accessing the file system (XFS) while loading programs, and reading data from the files. The operating system maintains a memory copy of the file system data structures like FAT(File Allocation Table) and Disk Free List (Refer Chapter 2). It is loaded from the disk to the memory during operating system boot.

Apart from the file system data structures XOS maintains details about files opened by all processes in the System-wide Open File Table. XOS uses a `scratchpad` to access files in the memory which will be explained further in this chapter.

5.1 File Allocation Table (FAT)

File allocation table (FAT) is a table that has an entry for each file present in the disk. FAT is stored in page number 5 in the memory.

The structure of a FAT entry is shown below

0	1	2	3 – 7
File Name	File Size	Block # of basic block	... Unused ...

Fig. 5.1: Structure of a FAT entry

5.2 Disk Free List

The Disk Free List is a data structure used for keeping track of unused blocks in the disk. The memory copy of Disk Free List is stored in the *page number* 6. It is stored in *block number* 20 in the disk.

5.3 System Wide Open File Table

This data structure maintains details about all open files in the system. It is located from words 1344 to 1471 of the memory (in Page 2). System Wide Open File Table consists of a maximum of 64 entries. Therefore, there can be at most 64 open files in the system at any time. Each entry of the System Wide Open File Table occupies 2 words. It has the following structure as shown in figure 5.2.

1 word	1 word
FAT Index	File Open Count

Fig. 5.2: Structure of an entry

- **FAT index :** It stores the index of the corresponding file in the FAT. An invalid entry is denoted by -1.
- **File Open Count :** File Open Count is the number of open instances of the file. When this becomes zero, the entry for the file is invalidated in the System Wide Open File Table.

The Per-Process Open File Table in the PCB of each process stores information about files opened by the corresponding process. Each entry in the Per-Process Open File Table has the index to the files entry in the System-wide Open File Table.

5.4 Scratchpad

There is a specific page of the memory which is reserved to store temporary data. This page is known as the *Scratchpad*. The scratchpad is required since any block of the disk cannot be accessed directly by a process. It has to be present in the memory for access. Hence, any disk block that has to be read or written into is first brought into the scratchpad. It is then read or modified and written back into the disk.

The *page number* 1 of the memory (Refer Chapter 2) is used as the scratchpad. Once the OS has booted up there is no need for the OS startup code. So this page can be reused as the scratchpad.

Chapter 6

System Calls

6.1 Introduction

System calls are interfaces through which a process communicates with the OS. Each system call has a unique name associated with it (Open, Read, Fork etc). Each of these names maps to a unique system call number. Each system call in turn causes a software interrupt to occur. Note that multiple system calls can be mapped to the same interrupt.

All the arguments to the system call are pushed into the user stack of the process which invokes the system call. The system call number is pushed as the last argument.

6.2 File System Calls

File system calls are used by a process when it has to create, delete or manipulate *Data files* that reside on the disk(file system). There are seven file system calls. An interrupt is associated with each system call. All the necessary arguments for a system call are available in the user stack with the system call number as the last argument.

6.2.1 Create

APL Syntax : `int Create(fileName)`
System Call No. : 1

This system call is used to create a new file in the file system whose name is specified in the argument. The return value of the `Create()` system call is 0 if it is a success, and -1 otherwise. If the file already exists, the system call returns 0 (success). It invokes Interrupt 1 Routine.

6.2.2 Open

APL Syntax : `int Open(fileName)`
System Call No. : 2

This system call is used to open an existing file whose name is specified in the argument. It calls Interrupt 2 Routine. The return value of the `Create()` system call is an integer value called `FileDescriptor`, which is the index of the corresponding file's entry in the Per-Process Open File Table.

6.2.3 Close

APL Syntax : `int Close(fileName)`
System Call No. : 3

This system call is used to close an open file. `FileDescriptor` is an integer value returned by the corresponding `Open()` system call. The return value of the `Close()` system call is 0 if it is a success, and -1 otherwise. It invokes Interrupt 2 Routine.

6.2.4 Delete

APL Syntax : `int Delete(fileDescriptor)`
System Call No. : 4

This system call is used to delete the file from the file system whose name is specified in the argument. The return value of the `Delete()` system call is 0 if it is a success, and -1 otherwise. It invokes Interrupt 1 Routine.

6.2.5 Write

APL Syntax : `int Write(fileDescriptor, wordToWrite)`
System Call No. : 5

This system call is used to write one word at the current seek position, into an open file (identified by `fileDescriptor`) from a string/integer variable (identified by `wordToWrite`). The return value of the `Write()` system call is 0 if it is a success or -1 otherwise. It invokes Interrupt 4 Routine.

6.2.6 Seek

APL Syntax : `int Seek(FileDescriptor, newLseek)`
System Call No. : 6

This system call is used to change the current value of the seek position in the per-process open file table entry of a file to the `newLseek` value. The return value of the `Seek()` system call is 0 if it is a success, and -1 otherwise. It invokes Interrupt 3 Routine.

6.2.7 Read

APL Syntax : `int Read(fileDescriptor, wordRead)`
System Call No. : 7

This system call is used to read one word at the current seek position, from an open file (identified by `fileDescriptor`) and store the word to a string/integer variable (identified by `wordRead`). The return value of the `Read()` system call is 0 if it is a success or -1 otherwise. It invokes Interrupt 3 Routine.

6.3 Process System Calls

Process system calls are used by a process when it has to duplicate itself, execute a new process in its place or when it has to terminate itself. There are three process system calls. An interrupt is associated with each system call. All the necessary arguments for a system call are available in the user stack with the system call number as the last argument.

6.3.1 Fork

APL Syntax : `int Fork()`
System Call No. : 8

This system call is used to replicate the process which invoked it. The new process which is created is known as the *child* and the process which invoked this system call is known as its *parent*. The return value of the `Fork()` system call to the parent process is the PID (*process identifier*) of the child process and -2 for the child process. It invokes Interrupt 5 Routine

6.3.2 Exec

APL Syntax : `int Exec(filename)`
System Call No. : 9

This system call is used to load the program, whose name is specified in the argument, in the memory space of the current process and start its execution.

The return value of the `Exec()` system call is -1 if it failed. It invokes Interrupt 6 Routine.

6.3.3 Exit

APL Syntax : `Exit()`
System Call No. : 10

This system call is used to terminate the execution of the process which invoked it and removes it from the memory . It schedules the next ready process and starts executing it. When there is no other ready process to run, it halts the machine. It invokes Interrupt 7 Routine.

Chapter 7

System Routines

The Operating System apart from its various data structures and interfaces it provides to the user processes, has certain routines to execute while start up and during interrupts. These routines are included as the Operating System Routines.

7.1 OS Startup Code

The OS Startup Code resides in the page 1 in the memory. When the machine boots up, the ROM Code loads the OS Startup Code from block 0 in the disk to page 1 in the memory. The OS Startup code initializes all data structures required for the OS, loads the FAT and Disk Free List from file system into the memory and starts execution of the `INIT` process.

7.2 Exception Handler

When the machine encounters an exception it sets EFR (Exception Flag Register) with details corresponding to the exception and calls the exception handler routine (pages 7 and 8 in memory).

Value of IP	BadVAddr	Cause	\0

Fig. 7.1: Structure of EFR

XOS handles all exceptions other than *Page Fault* by killing the process which caused the exception.

Page Fault Exceptions

The `Cause` field of `EFR` for Page Fault Exceptions is **0**. The logical page which caused the exception to occur (indicated by `BadVAddr` field in EFR) will not

have a corresponding valid entry in the page table of the process. If the page table entry contains a disk block number, the block is loaded from the disk to a free memory page, and this memory page number is stored in the page table entry . The Valid/Invalid bit is set to 1, and the exception handler returns back to the process.

7.3 Timer Interrupt Routine

The Timer Interrupt Routine is responsible for context switch, i.e. storing the state (values of the registers) of the currently executing process to the PCB, and setting the registers with values from the PCB of the next ready process in the Ready List of PCBs. A scheduler is responsible for selecting a ready process from this list. The Scheduler code is also contained in the Timer Interrupt Routine. The Timer Interrupt routine resides in pages 9 and 10 of the memory.

7.4 Interrupt Routines

The Interrupts from 1 to 7 are invoked by the user processes through system calls. Each interrupt routine has code corresponding to one or more system calls. Every interrupt routine occupies 2 pages in memory. Interrupt routines for interrupts 1 to 7 reside in memory pages 11 to 24.

Appendix B

Machine (XSM) Specification

Contents

Chapter 1

Introduction

1.1 Brief Machine Description

The machine simulator is known as *Experimental String Machine* (**XSM**). 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** (Refer Section 3). The machine interprets a single character also as a string.

1.2 Components of the Machine

- **Disk** : It is a non-volatile storage that stores user programs (executables) and data files. The Operating System code is also stored in the disk.
- **Memory** : It is a volatile storage that stores the programs to be run on the machine as well as the operating system that manages the various programs.
- **Processor** : It is the main computational unit that is used to execute the instructions.
- **Timer** : It is a device that interrupts the processor after a pre-defined specific time interval.
- **Load/Store** : It is a macro that performs the functionalities of DMA (Direct Memory Access) controller. (Refer Section 5)

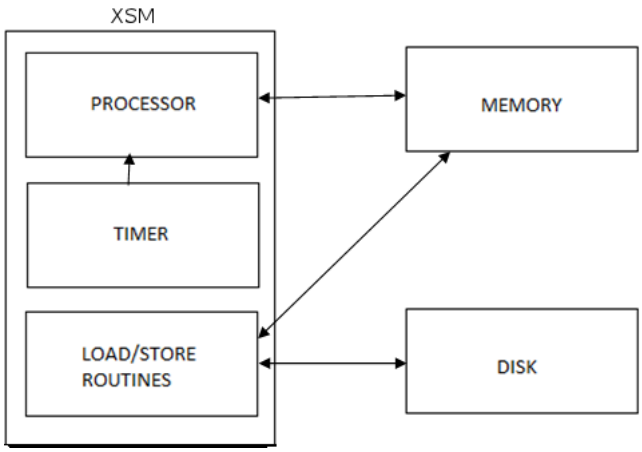


Figure 1.1: Components of the Machine

1.3 Supported Datatypes

XSM supports 2 different datatypes and their operations, namely **Strings** and **Integers**. However in the lowest level both integers and strings are internally stored as strings.

1.3.1 Strings

: Strings are sequence of characters which may include alphabets, numerals and special characters. Every string is terminated with a *null character* (\0). Operations that can be performed on strings include lexicographic comparisons.

1.3.2 Integers

: Apart from strings, XSM supports integers and its operations. The operations that can be performed on integers include arithmetic operations and comparison operations. A jump can also performed by checking if a register has 0 in it.

Chapter 2

Registers

2.1 Introduction

The XSM architecture maintains 34 registers (each one **word**).

2.2 Register Set

There are 28 General Purpose Registers (GPR), of which `R0 - R7` are Program Registers and `S0 - S15` are Kernel Registers. There are 4 temporary registers `T0 - T3` which are reserved for code translation. The registers `T0 - T3` are not intended to be used by the system programmer.

In addition to these 28 registers there are 6 Special Purpose Registers (SPR) namely `BP` (Base Pointer), `SP` (Stack Pointer), `IP` (Instruction Pointer), `PTBR` (Page Table Base Register) and `PTLR` (Page Table Length Register) and the `EFR` (Exception Flag Register).

Name	Register
Program Register	R0-R7
Kernel Register	S0-S15
Temporary Registers	T0-T3
Base Pointer	BP
Instruction Pointer	IP
Stack Pointer	SP
Page Table Base Register	PTBR
Page Table Length Register	PTLR
Exception Flag Register	EFR

Chapter 3

Memory

3.1 Introduction

- The basic unit of memory in XSM is a **word** (length = 16 bytes).
- The machine memory can be thought of as a linear sequence of **words**.
- A collection of 512 contiguous **words** is known as a **page**.
- The total size of the memory is 64 **pages** or 32768 (512 × 64) **words**.
- Each **word** in the memory is identified by the *word address* in the range 0 to 32767. Similarly, each **page** in the memory is identified by the *page number* in the range 0 to 63.
- The *page number* corresponding to a word is obtained by the formula,

$$page\ number = \lfloor \frac{word\ address}{512} \rfloor$$

3.2 Address Translation

There are two kinds of memory addresses,

- Logical address : When a process runs, CPU generates address for the data accessed by this process. This address is called the Logical address.
- Physical address : It is the actual location of the data in the main memory.

Address translation is the process of obtaining the physical address from the logical address. It is done by the machine in the following way.

1. The logical address generated by the CPU is divided by the page size (512) to get the **logical page number**. The remainder is the **offset** of the data within that page.

2. A **page table** is used for address translation. It resides in the memory, the location of which is pointed to by **PTBR** (Page Table Base Register). The number of entries in the page table is stored in **PTLR** (Page Table Length Register). Each entry of the page table is two words long.

   - First word of a valid page table entry contains a physical page number corresponding to a logical page number.
   - The second word contains *auxiliary information*. The first bit in this word is used as **reference bit**. This bit is set to 0 initially. When a particular page is accessed, the machine sets the reference bit of the corresponding page table entry to 1 (indicating referenced).
     The second bit of the auxiliary information is used as a **Valid/Invalid bit**. It is set to 1 if the entry is a valid physical memory page or set to 0 otherwise. When an access to a page table entry with 0 as the Valid bit is made, machine sets the **EFR** (Exception Flag Register) with the IP of the instruction which caused the exception, logical page number of the memory location that caused the exception, and caused of the exception (page fault). The privilege mode is changed from **user** to **kernel** mode. The value of **IP** is set to page 7 or word address 3584 in the memory. If an access to a page whose page table entry has Valid/Invalid bit set to 1 is made, the physical address corresponding to the logical address is calculated using the method described here.

   The logical page number is used to index the page table to get the corresponding physical page number.

3. The **offset** is then used to refer to the word in the physical page containing the data.

The example below shows the address translation corresponding to the logical address 13532.

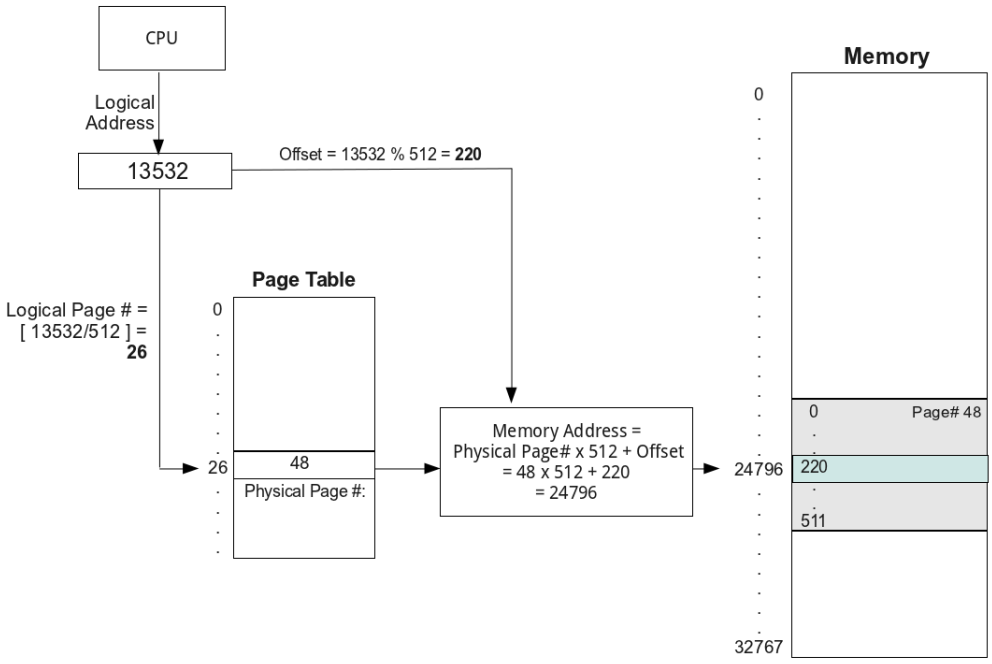


Figure 3.1: The logical address generated by the CPU is 13532, so the page number is $\lfloor 13532/512 \rfloor = 26$ and offset is 13532 mod 512 = 220. Let the 26^{th} entry in the page table be 48. Thus the resultant physical address is $48 \times 512 + 220 = 24796$.

3.3 ROM Code

It is a hard coded assembly level code present in page 0 of the memory. It is known as the ROM (Read Only Memory) code because in an actual machine it is burnt in the hardware. When the machine boots up, this code is executed. This code has the basic functionality of loading block 0 of the disk (which generally contains the OS startup code) into page 1 of the memory and to set the `IP` register value to 512.

Chapter 4

Disk Storage

Block : It is the basic unit of storage in the disk.

The disk can be thought of as consisting of a linear sequence of 512 **blocks**. The size of each **block** is equal to that of a page in the memory (512 words). The total disk capacity is $512 \times 512 = 262144$ **words**.

Any particular **block** in the disk is addressed by the corresponding number in the sequence 0 to 511 known as the *block number*.

0 - 511	512 - 1023	...	261632 - 262143
Block 0	Block 1	....	Block 512

Figure 4.1: Disk Structure

Chapter 5

Instructions

5.1 Introduction

Every instruction in XSM is 2 words long. The instructions provided by the XSM architecture can be classified into privileged and unprivileged instructions.

5.2 Classification

Unprivileged Instructions

1. `MOV`

    - Register Addressing:
      Syntax : `MOV Ri, Rj`
      Copies the contents of the register `Rj` to `Ri`.
    - Immediate Addressing:
      Syntax : `MOV Ri, INTEGER/STRING` Copies the `INTEGER/STRING` to the register `Ri`.
    - Register Indirect Addressing:
      Syntax : `MOV Ri, [Rj]`
      Copy contents of memory location pointed by `Rj` to register `Ri`.
      Syntax : `MOV [Ri], Rj`
      Copy contents of `Rj` to the location whose address is in `Ri`.
    - Direct Addressing:
      Syntax : `MOV [LOC], Rj`
      Copy contents of `Rj` to the memory address `LOC`.

Syntax : `MOV Rj, [LOC]`
Copy contents of the memory location `LOC` to the register `Rj`.

- Direct Indexed Addressing:
  Syntax : `MOV [LOC] Rj, Ri`
  Copy contents of `Ri` to the memory address `LOC` + (value in `Rj`)
  Syntax : `MOV [LOC] Index, Rj`
  Copy contents of `Ri` to the memory address `LOC` + `Index`. `Index` must be an integer value.
  Syntax : `MOV Ri, [LOC] Rj`
  Copy contents in the memory address `LOC` + (value in `Rj`) to the register `Ri`
  Syntax : `MOV Ri, [LOC] Index`
  Copy contents of the memory address `LOC` + `Index` to the register `Ri`. `Index` must be an integer value.

2. **Arithmetic Instructions**
   Arithmetic Instructions perform arithmetic operations on registers containing integers. If the register contains a non-integer value, an exception is raised (Refer Section **??**)

   - `ADD`, `SUB`, `MUL`, `DIV` and `MOD`.
     General Syntax : `OP Ri, Rj`
     The result of `Ri` op `Rj` is stored in `Ri`.
   - `INR`
     Syntax : `INR Ri`
     Increments the value of register `Ri` by 1.
   - `DCR`
     Syntax : `DCR Ri`
     Decrements the value of register `Ri` by 1.

3. **Logical Instructions**
   Logical instructions are used for comparing values in registers. Strings can also be compared according to the lexicographic ordering of ASCII.

   - `LT`
     Syntax : `LT Ri, Rj`
     Stores 1 in `Ri` if the value stored in `Ri` is less than that in `Rj`. `Ri` is set to 0 otherwise.

- `GT`
  Syntax : `GT Ri, Rj`
  Stores 1 in `Ri` if the value stored in `Ri` is greater than that in `Rj`. `Ri` set to 0 otherwise.
- `EQ`
  Syntax : `EQ Ri, Rj`
  Stores 1 in `Ri` if the value stored in `Ri` is equal to that in `Rj`. Set to 0 otherwise.
- `NE`
  Syntax : `NE Ri, Rj`
  Stores 1 in `Ri` if the value stored in `Ri` is not equal to that in `Rj`. Set to 0 otherwise.
- `GE`
  Syntax : `GE Ri, Rj`
  Stores 1 in `Ri` if the value stored in `Ri` is greater than or equal to that in `Rj`. Set to 0 otherwise.
- `LE`
  Syntax : `LE Ri, Rj`
  Stores 1 in `Ri` if the value stored in `Ri` is less than or equal to that in `Rj`. Set to 0 otherwise.

4. Branching Instructions Branching is achieved by changing the value of the `IP` to the word address of the target instruction specified by <`target_address`> .

   - `JZ`
     Syntax : `JZ Ri, <target_address>`
     Jumps to <`target_address`> if the contents of `Ri` is zero.
   - `JNZ`
     Syntax : `JNZ Ri, <target_address>`
     Jumps to <`target_address`> if the contents of `Ri` is not zero.
   - `JMP`
     Syntax : `JMP <target_address>`
     Unconditional jump to <`target_address`>

5. Stack Instructions

   - `PUSH`
     Syntax : `PUSH Ri`

Increment SP by 1 and copy contents of Ri to the location pointed to by SP.

- POP
  Syntax : POP Ri
  Copy contents of the location pointed to by SP into Ri and decrement SP by 1.
  For both these instructions Ri may be any register except IP.

6. Subroutine Instructions
   The CALL instruction copies the address of the next instruction to be fetched on to location SP + 1. It also increments SP by one and transfers control to the instruction specified by the <target_address>. The address of the instruction to be fetched is in IP + 2 (each instruction is 2 memory words). The RET instruction restores the IP value stored at location pointed by SP, decrements SP by one and continues execution fetching the next instruction pointed to by IP. The subroutine instructions provide a neat mechanism for procedure evocations.

   - CALL
     Syntax : CALL <target_address>
     Increments SP by 1, transfers IP+2 to location pointed to by SP and jumps to instruction specified by <target_address>
   - RET
     Syntax : RET
     Sets IP to the value pointed to by SP and decrements SP.

7. Input/Output Instructions

   - IN
     Syntax : IN Ri
     Transfers the contents of the standard input to Ri.
   - OUT
     Syntax : OUT Ri
     Transfers the contents of Ri to the standard output.

8. Debug Instruction
   Syntax : BRKP
   The machine when run in debug mode invokes the debugger when this

intruction is executed. This instruction can be used for debugging system code.

9. `END`
   Syntax : `END`
   This instruction is marks the end of a program.

10. `INT`
    Syntax : `INT n`
    Generates an interrupt to the kernel with `n` (1 to 7) as a parameter. It also disables the interrupts. (Read Section 6)

Privileged Instructions

There are four privileged instructions. They can only be executed in kernel mode (Refer to 5.3). These instructions are:

1. `IRET`
   Syntax : `IRET`
   `IRET` switches the mode from kernel to user mode. It then sets `IP` to the value pointed by `SP` and decrements `SP` by one. With the execution of the `IRET` instruction, interrupts are enabled. (Read Section 6)

2. `LOAD`
   Syntax : `LOAD` *`pagenum blocknum`*
   This instruction loads the block specified by the *`blocknum`*, from the disk, to the page specified by the *`pagenum`* in the memory. *`blocknum`* and *`pagenum`* should be numbers or registers containing numbers. An excpetion is raised (Refer Section **??**) for invalid arguments or illegal memory access.

3. `STORE`
   Syntax : `STORE` *`blocknum pagenum`*
   This instruction stores the page specified by the *`pagenum`*, from the memory, to the block specified by the *`blocknum`*in the disk. *`blocknum`* and *`pagenum`* should be numbers or registers containing numbers. An excpetion is raised (Refer Section **??**) for invalid arguments or illegal memory access.

   The below example will store the 31st page in memory to the 64th block in the disk. It will then load it to the 15th page in memory.

```
MOV R1, 64
STORE R1, 31
LOAD 15, R1
```

4. `HALT`
   Syntax : `HALT`
   This instruction causes the machine to halt immediately.

5.3 Privilege Modes

The XSM architecture is interrupt driven and uses a single processor. There are two privilege modes of execution, the user mode and the kernel mode. The machine is initially in kernel mode. It switches to user mode when it encounters an IRET instruction. It switches back to kernel mode after an interrupt or an exception occurs.

- User mode : Only unprivileged instructions can be executed in this mode. Only registers `R0` to `R7`, `SP` and `BP` are allowed to be used in any instruction in this mode. Address translation occurs for all addresses in user mode.
- Kernel mode : Both privileged and unprivileged instructions can be executed in this mode. The value of `IP` and `EFR` cannot be explicitly changed by any instruction. Also these registers cannot be used in addressing memory locations. All other registers can be used in Kernel Mode. Address translation does not occur in kernel mode, as all addresses are physical addresses. In kernel mode interrupts are disabled.

Chapter 6

Interrupts

Interrupts are mechanisms by which the machine interrupts the execution of the processor and passes control to the kernel to execute interrupt (or exception) handler code. Interrupts might indicate errors, such as a memory access violation (page fault), a timer interrupt or a software interrupt invocation from a running program.

The process saves its current state before starting execution of the handler, and then resumes the state once the handler finishes its execution. Interrupts are disabled when the interrupt handler code is executing.

6.1 Exceptions

Exceptions are anomalous situations which changes the normal flow of execution. There is a flag associated with each exception and the details corresponding to the exception that occured is stored in `EFR` (Exception Flag Register). If an exception occurs in User Mode, the machine transfers control of execution, i.e. changes the value of IP to *page number* 23 (address = 11776) where the Exception Handler Routine resides. However in Kernel Mode, the machine halts when it encounters an exception.

The structure of `EFR` is given below

Value of IP	BadVAddr	Cause	\0

- **Value of IP**: Stores the value of IP at the point where the exception occured. The maximum length of IP is 5 digits.

- **BadVAddr** (Bad Virtual Address): This field is relevant when a Page Fault Exception occurs. The logical page number which caused a page fault exception is stored here. The length of this field is 2 characters.
- **Cause**: This field indicates a number which corresponds to the cause of the exception. eg : a page fault exception has value 0 stored in Cause field. The length of this field is 1 character.

Exceptions can be caused when the following events occur.

1. **Page Fault** : occurs when the page table entry corresponding to the logical address is invalid. The value stored in *cause* field of **EFR** for this exception is **0**.
2. **Illegal instruction** : occurs when an attempt is made to execute an instruction not belonging to the instruction set and also when the operands to the instruction is not legal. Eg: `MOV 4 R0`, `MOV IP 4` when executed in user mode. These instructions are considered illegal. The value stored in the *cause* field of **EFR** for this exception is **1**.
3. **Illegal memory access** : occurs when any address generated by the process lies outside its logical address space. The logical page number generated should be between 0 and the value of `PTLR`. The value stored in the *cause* field of **EFR** for this exception is **2**.
4. **Arithmetic exception** : occurs when divisor is 0. The value stored in the *cause* field of **EFR** for this exception is **3**.
5. **Illegal operands** : occurs when operands contain invalid data corresponding to the instruction. The value stored in the *cause* field of **EFR** for this exception is **4**.

6.2 Timer Interrupt

Timer Interrupt is automatically triggered by the machine, at set intervals of instructions. This interrupt cannot be invoked using an INT instruction.

It transfers control of execution, i.e. changes the value of IP to page number 9 (address = 4608). This is the timer interrupt which interrupts the processor. Generally it is supposed to contain the code for the scheduler of the operating system, which schedules the CPU time among the various active processes.

6.3 Software Interrupts

Software Interrupts interrupts are unprivileged and can be called from user mode. 7 Interrupt instructions are provided by the machine which transfers control to specific locations in memory, where **ISR** (Interrupt Service Routines) of the operating system is expected to be present.

The `INT` instruction

The instruction used to generate a software interrupt is `INT`.
Syntax : `INT n`
The `INT` instruction passes control to the Interrupt Service Routine (ISR) for this interrupt located at the physical address computed using the value n. The physical address of the ISR corresponding to interrupt number n is given by:

```
Physical Address = (9 + 2n) x Page Size
```

Note that the interrupts are disabled once this instruction is executed as interrupts cannot be executed in kernel mode.

The 7 `INT` instructions are :

- `INT 1`
  It transfers control of execution to *page number* 11 (address = 5632)
- `INT 2`
  It transfers control of execution to *page number* 13 (address = 6656)
- `INT 3`
  It transfers control of execution to *page number* 15 (address = 7680)
- `INT 4`
  It transfers control of execution to *page number* 17 (address = 8704)
- `INT 5`
  It transfers control of execution to *page number* 19 (address = 9728)
- `INT 6`
  It transfers control of execution to *page number* 21 (address = 10752)
- `INT 7`
  It transfers control of execution to *page number* 23 (address = 11776).

Brief memory outline for XSM is given below.

Page Number	**Contents**	**Word Address**	**No. of Words**
0	ROM Code	0 – 511	512
1	OS Startup Code	512 – 1023	512
2 – 6	OS Structures	1024 – 3583	2560
7 – 8	Exception Handler	3584 – 4607	1024
9 – 10	Timer Interrupt Routine	4608 – 5631	1024
11 – 12	Interrupt 1	5632 – 6655	1024
13 – 14	Interrupt 2	6656 – 7679	1024
15 – 16	Interrupt 3	7680 – 8703	1024
17 – 18	Interrupt 4	8704 – 9727	1024
19 – 20	Interrupt 5	9728 – 10751	1024
21 – 22	Interrupt 6	10752 – 11775	1024
23 – 24	Interrupt 7	11776 – 12799	1024
25 – 63	User Programs	12800 – 32767	19968

Appendix C

Filesystem (XFS) Specification

Contents

1 Introduction

XFS or *Experimental File System* is a file system architecture designed for **XOS** (*Experimental Operating System*). XFS is a simple filesystem which has no directory structure.

XFS spans the entire disk in **XSM** (*Experimental String Machine*). The disk consists of a linear sequence of 512 blocks. The basic unit of disk in XSM is a ***block***. The size of the block is equal to that of page in memory (512 words). The total capacity of the disk is 512 × 512 = **262144** words.

Any particular block in the disk is addressed by the corresponding number in the sequence 0 to 511 known as the *block number*.

2 Disk Organization

The disk is organized by the file system as shown below.

Block No	Contents	No. of Blocks
0	OS Startup Code	1
1-2	Exception Handler	2
3-4	Timer Interrupt Routine	2
5-6	Interrupt 1 Routine	2
7-8	Interrupt 2 Routine	2
9-10	Interrupt 3 Routine	2
11-12	Interrupt 4 Routine	2
13-14	Interrupt 5 Routine	2
15-16	Interrupt 6 Routine	2
17-18	Interrupt 7 Routine	2
19	File Allocation Table (FAT)	1
20	Disk Free List	1
21-23	INIT Code	3
24 – 447	User Blocks	424
448 – 511	Swap Area	64

Figure 1: Structure of the disk

- **OS Startup Code**: Block 0 is the location of operating system code required during machine boot.

- **Exception and Interrupt Handler**: Blocks from 1 - 18 are intended for Interrupt Routines for interrupt and exception handling which is done by the Operating System.
- **FAT** or File Allocation table contains details about the files stored on the disk.
- **Disk Free List** has 512 entries, one entry for each block in the disk, indicating whether it is used or unused.
- **INIT Code**: It has the code for INIT process, which is the first user program run by the OS after startup.
- **User Blocks**. Blocks from 24 to 447 are used to store user data files and user programs.
- **Swap Area**: The file system also provides a swap area for the operating system to implement demand paging. Swap Area is reserved exclusively for use by the operating system.

3 File

A file is a collection of blocks identified by a name. Every file in the disk has a *Basic Block* and several *Data Blocks*. They are defined as follows:

- **Data Blocks :** These blocks contain the actual data of a file.
- **Basic Block :** It consists of information about the data in a file.

Index	0–255	256–511
Content	Block List	Header

Figure 2: Structure of the basic block of a file

  - **Block List :** It contains block addresses of all data blocks in the file.
    - The block list consists of 256 entries.
    - Each entry is of size one word.
    - The value contained in an entry of the block list gives the block number of the corresponding data block in the disk. All invalid entries are marked with -1.

– **Header :** The header contains the meta information relating to the file. The header fields is 256 words long. It can be used to store details like permissions, ownership etc related to a file, similar to inode in Linux and UNIX operating systems. However XOS does not use the header field.

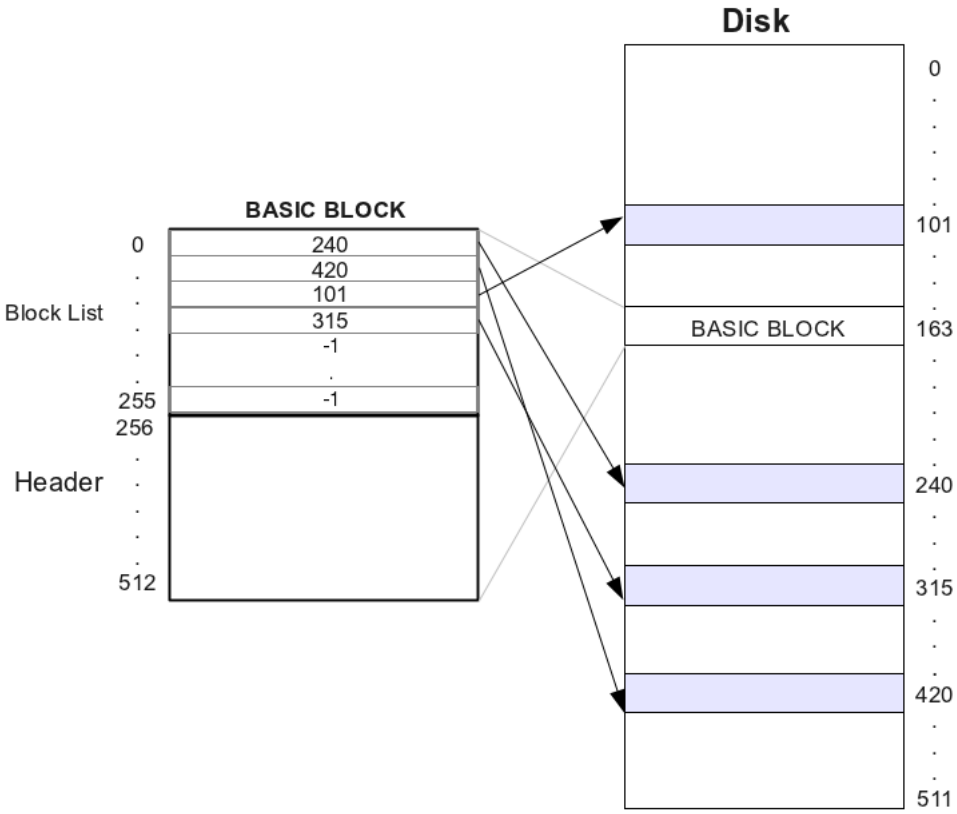


Figure 3: Example illustrating the basic block of a file

From the figure 3, we infer the following.

- The zeroth data block of the file resides in the disk in block number 240
- The first data block of the file resides in the disk in block number 420. The second data block of the file resides in the disk in block number

101. The third data block of the file resides in the disk in block number 315

- There are no more data blocks for the file. So rest of the entries of the basic block are marked as -1

3.1 File Types

There are two types of files in the XSM architecture. They are:

3.1.1 Data files

These files contain data or information that is used by the programs. They can occupy a maximum of 257 blocks (1 basic block + maximum 256 data blocks). Data files have an extension `.dat` in the filename.

3.1.2 Executable files

These contain programs that the user wishes to run on the machine. They occupy 4 blocks (1 basic block + 3 data blocks) of the disk. Executable files have an extension `.xsm` in the filename.

4 File Allocation Table (FAT)

File allocation table (**FAT**) is a table that has an entry for each file present in the disk. FAT of the filesystem consists of 64 entries. Thus there can be a maximum of 64 files.

- Total size of the FAT is thus 512 words, which occupies 1 block.
- It is a disk data structure and occupies block number 19 on the disk.

4.1 Structure of FAT

The structure of a FAT entry is shown below

0	1	2	3 – 7
File Name	File Size	Block # of basic block	... Free ...

Figure 4: Structure of a FAT entry

The FAT entry consists of the

1. **File Name :** It is an identification of a file. It can be of maximum 15 characters (and thus requires 1 word). Typical file names are `student.dat`, `calc.xsm`.

2. **File size :** It indicates the number of words required for the data blocks of the file. The number of data blocks of the file can vary from 0 to 256, and so the maximum file size is 131072 words (256×512). It occupies one word in the FAT entry.

3. **Block number of basic block :** It contains the block number where the basic block of a file resides in the disk. It occupies one word in the FAT entry.

For an invalid FAT entry, the value for basic block is stored as -1.

5 Disk Free List

The Disk Free List is a data structure used for keeping tracking of unused blocks in the disk.

- The Free List of the disk consists of 512 entries. Each entry is of size one word.
- The size of the free list is thus 1 block or 512 words.
- It is present in blocks 20 of the disk. Refer figure 1.
- For each block in the disk there is an entry in the Disk Free List which contains a value of either 0 or 1 indicating whether the corresponding block in the disk is free or used respectively. Each entry in the Disk Free List is of size one word.
- Blocks 0 to 20 are system reserved and are marked as 1 in the Disk Free List so that they cannot be used for saving user files. The file system also ensures that user files are not stored on the Swap Area.

Appendix D

Application Programmer's Language (APL) Specification

Contents

1 Introduction

APL or *Application Programmers Language* is a simple and statically typed programming language. The features and constructs of this language are minimal and mainly intended for testing an experimental operating system. The compiler of APL runs on XSM (*eXperimental String Machine*) architecture.

This document describes the programming constructs, syntax and semantics of APL. The structure of APL is similar in some aspects to programming languages like C and Java. A typical APL program is orgnaized in the following way.

```
Global Declarations
...
Function Definitions
...
Main Function
```

2 Lexical Elements

2.1 Comments and White Spaces

APL allows only line comments. Line comments start with the character sequence **//** and stop at the end of the line. White spaces in the program including tabs, newline and horizontal spaces are ignored.

2.2 Keywords

The following are the reserved words in APL and it cannot be used as identifiers.

```
read   print     if       then       else     endif
while    do    endwhile    break    continue  integer
string  main   return      decl     enddecl   Create
 Open   Write    Seek       Read      Close    Delete
 Fork    Exec    Exit    breakpoint
```

2.3 Operators and Delimiters

The following are the operators and delimiters in APL

```
(   )   {    }   [   ]   /   *   +   -   %
>   <   >=   <=  !=  ==  ;   =   &&  ||  !
```

2.4 Identifiers

Identifiers are names of variables and user-defined functions. Identifiers should start with an alphabet, and may contain both alphabets and digits. Special characters are not allowed in identifiers.

```
identifier -> (alphabet)(alphabet | digit)*
```

2.5 Literals

There are integer literals and string literals in APL. An integer literal is a sequence of digits representing an integer. Negative integers are represented with a negative sign preceding the sequence of digits. Any sequence of characters enclosed within double quotes (") are considered as string literals. However APL restricts string literals to size of atmost 16 characters including the '\0' character which is implicitly appended at the end of a string value.

Examples of literals are : `19, -35, "Hello World"`

3 Data Types

3.1 Primitive Types

There are two primitive datatypes in APL.

1. **Integer** : An integer value can range from -32767 to +32768. An integer type variable is declared using the keyword `integer`
2. **String** A string type represents the set of string values. A string value can be atmost 16 characters long. String type variables is declared using the keyword `string`.

3.2 Arrays

Arrays are sequence of elements of a single type. Arrays can be of **integer** or **string** data types. APL allows the use of single-dimensional arrays only, i.e. linear arrays. Array elements are accessed by the array name followed an index value within square brackets (e.g. arr[10]).

4 Declarations and Scope

Declarations should be made for variables and functions defined in the APL program.

4.1 Global Variables

Global variables are declared in the first section of the program within a **decl** ... **enddecl** block. Global variables can be accessed from any function in the program. Global variables can be of integer, string, integer array or string array datatypes. Global variables are declared with its datatype followed by the variable name. If the variable refers to an array the size of the array must be given in square brackets. The general form of declarations is as follows

type variable_name;
type variable_name[size];

4.2 Function Declaration

For every function except the **main()** function defined in a APL program, there must be a declaration. All functions have global scope and is declared in the first section within **decl ... enddecl** block, along with the global variables.

A function declaration should specify the name of the function, the name and type of each of its arguments and the return type of the function. A function can have integer/string arguments. Parameters may be passed by value or reference. Arrays cannot be passed as arguments. If a global variable name appears as an argument, then within the scope of the function, the new declaration will be valid and global variable declaration is suppressed. Different functions may have arguments of the same name. For arguments that are passed by reference, the argument name is preceded by an ampersand(&) in the function declaration. The return type of a function must be either integer or string.

The general form of declarations is as follows

type function_name (type1 argument1,argument2,...; type2 argument1,argument2,...;...);

Examples for global declarations

```
decl
        integer x,y,a[10],b[20];
        integer f1(integer a1,a2; string b1; integer &c1), f2();
        string t, q[10], f3(integer x);
        integer swap(integer &x, &y);
enddecl
```

4.3 Local Variables

Local variables can be declared anywhere inside a function definition except in the body of **if** and **while**. Local variables will have a function scope, i.e. it can only be accessed in the function in which it is declared. Arguments of a function are treated as local variables. Local variables can be integer or string. Arrays cannot be declared locally. All globally declared variables are visible inside a function, unless suppressed by a re-declaration. The general form of declarations is as follows

type variable_name;

5 Function Definition and Main Function

Every APL program must have a **main()** function and zero or more user-defined functions. Every function other than the **main()** function must be declared within the **decl ... enddecl** block. The general form of a function definition is given below

type function_name(ArgumentList)
{

Function Body
}

The function body must contain a return statement and the return value must be of the return type of the function. The arguments and return type of each function definition should match exactly with the corresponding declaration. Every declared function must have a definition. The signature of the function in the delcaration should match the definition of the function which includes the return type, and the names, passing method and datatypes of the arguments. The language supports recursion and static scope rules apply.

5.1 main()

The **main()** function must be a zero argument function of type integer. Program execution begins from the body of the **main()** function. The **main()** function need not be declared. The **main()** function definition follows all user-defined function definitions. The definition part of **main()** should be given in the same format as any other function.

6 Expressions

An expression specifies the computation of a value by applying operators and functions to operands. Function call in APL are treated as expressions, and the value of the expression is its return value. APL supports arithmetic and logical expressions

6.1 Arithmetic Expressions

Any integer value, variable, function returning an integer or 2 or more arithmetic expressions connected by arithmetic operators termed as arithmetic expressions. APL provides five arithmetic operators, viz., +, -, *, / (Integer Division) and % (Modulo operator) through which arithmetic expressions may be combined. Expression syntax and semantics are similar to standard practice in programming languages and normal rules of precedence, associativity and paranthesization hold. APL is strongly typed, and hence the types of the oprands must match the operation.

6.2 Logical Expressions

Logical expressions may be formed by combining arithmetic expressions using relational operators. The relational operators supported by APL are

```
<, >, <=, >=, ==, !=
```

Standard meanings apply to these operators. The operators take two arithmetic expressions as operands and the result will be a boolean value, either of 1(true) or 0(false). Only realational operator that can be applied to two strings is == (to check equality). This also considered as a Logical expression. Logical expressions themselves may be combined using logical operators, && (logical and) , || (logical or) and ! (not).

6.3 Function Call

All functions except the **main()** function can be invoked from any other function including itself. The general form of a function call is

function_name(value1,value1...);

Function calls are treated as expressions. The function takes in the values of its arguments and returns a value of type equal to the return type of the function. This value is treated as the evaluated result of the function call.

7 Statements

Statements control the execution of the program. All statements in APL are terminated with a semicolon ;

7.1 Assignment Statement

The APL assignment statement assigns the value of an expression to a variable, or an indexed array of the same type or a string value to a string variable. **=** is known as the assignment operator. Initialization during declaration is not allowed in APL. The general syntax is as follows
variable_name = string_value / array_variable / expression

7.2 If Statement

If statements specify the conditional execution of two branches according to the value of a boolean expression. If the expression evaluates to true, the **if** branch is executed, otherwise, if present, the **else** branch is executed. The **else** part is optional. The general syntax is as follows

if *(logical expression)* ***then***
    statements;
else
    statements;
endif;

7.3 While Statement

While statement iteratively executes a set of statements based on a condition which is a logical expression. The statements are iteratively executed as long as the logical expression evaluates to true.

while *(logical expression)* ***do***
    statements;
endwhile;

7.4 Break statement

Break statement is a statement which is used in a **while** loop block. This statement stops the execution of the loop in which it is used and passes the control of execution to the next statement after the loop. This statement cannot be used anywhere else other than **while** loop. The syntax is as follows

break ;

7.5 Continue statement

Continue statement is a statement which is also used only in a **while** loop block. This statement skips the current iteration of the loop and passes the control to the next iteration after checking the loop condition. The syntax is as follows

continue ;

7.6 Return statement

Return statement in a function passes the control from the callee to the caller function and returns a value to the caller function. All functions including the **main()** must have exactly one **return** statement and it should be the last statement in the funtion body. The return type of the function should match the type of the expression. The return type of main is integer. The syntax is as follows

return *expression;*

7.7 Read/Print statements

The standard input and output statements in APL are **read** and **print** respectively. The read statement reads an integer value from the standard input device into an integer variable or an indexed array variable or a string value into a string variable. The print statement outputs a string literal or the value of string variable or an arithmetic expression into the standard output.

read *variable_name;*
print *expression / string;*

7.8 Breakpoint statement

A **Breakpoint** statement is used to debug the program. The program when run in debug mode pauses the execution at this instruction.

breakpoint;

8 System Calls

System Calls allow the programs written in APL to interact with the operating system running on the XSM architecture. The following system calls are supported by APL.

8.1 Create

Create s a file with the specified filename in the filesystem. The return value of this system call is 0 in case of success and the appropriate error code in case of failure.

integer ***Create****(string filename);*

NOTE: filename must not exceed 10 characters

8.2 Open

Returns a file descriptor of the file in the filesystem with the specified filename. The file descriptor is an integer value. If the **Open** fails, an appropriate error code is returned .

integer ***Open****(string fileName);*

8.3 Read

Reads one word from a file which has the specified file descriptor, into a string/integer variable. The return value of this system call is 0, if it is a success. If the **Read** fails, an appropriate error code is returned.

integer ***Read****(integer fileDescriptor, string/integer variable);*

8.4 Write

Writes one word from a string/integer variable, to a file in the filesystem with the specified file descriptor. The return value of this system call is 0, if it is a success. If the **Write** fails, an appropriate error code is returned.

integer ***Write****(integer fileDescriptor, string/integer variable);*

8.5 Seek

Seek is used to change the read/write head position in a file. It moves the head to the specified number of words from the beginning of the file. The return value of this system call is 0 in case of success and the appropriate error code in case of failure.

integer ***Seek****(integer fileDescriptor, integer numWords);*

8.6 Close

This system call is used to close an open file. The return value of this system call is 0 in case of success and the appropriate error code in case of failure.

integer ***Close****(integer fileDescriptor);*

8.7 Delete

This system call is used to delete the file from the file system whose name is specified in the argument. The return value of this system call is 0 in case of success and the appropriate error code in case of failure.

integer ***Delete****(string fileName);*

8.8 Fork

This system call is used to create a copy of the current process in the system. The return value of this system call is the PID of the child process for the parent, and an appropriate code for the child.

integer ***Fork****();*

8.9 Exec

This system call is used to load the program, whose name is specified in the argument, in the memory space of the current process (overwriting existing contents) and start its execution. An appropriate error code is returned in case of failure.

integer ***Exec****(string fileName);*

8.10 Exit

This system call is used to terminate the execution of the process which invoked it and remove it from the memory.

void ***Exit****();*

8.11 Getpid

This system call returns the processID or thbe PID of the current process on success and an appropriate error code is returned in case of failure.

integer ***Getpid****();*

8.12 Getppid

This system call returns the processId of the parent process of the current process and an appropriate error code is returned in case of failure.

integer ***Getppid****();*

8.13 Wait

This system call blocks the current process till the process with processID given as argument signals or exits. The return value of this system call is 0 in case of success and the appropriate error code in case of failure.

integer ***Wait*** *(integer processID);*

8.14 Signal

This system call is used to resume execution of all process waiting for the current process. The return value of this system call is 0 in case of success and the appropriate error code in case of failure.

integer ***Signal*** *();*

Appendix E

System Programmer's Language (SPL) Specification

Contents

1 Introduction

SPL or *System Programmers Language* is an untyped programming language designed for implementation of an operating system on XSM *(Experimental String Machine)* architecture. The language is minimalistic and consists only of basic constructs required for the implementation. Programming using SPL requires a basic understanding of the underlying XSM architecture and operating system concepts.

2 Lexical Elements

2.1 Comments and White Spaces

SPL allows only single line comments. Comments start with the character sequence `//` and stop at the end of the line. White spaces in the program including tabs, newline and horizontal spaces are ignored.

2.2 Keywords

The following are the reserved words in SPL and it cannot be used as identifiers.

```
alias     else       if     store    while
define    endif    ireturn  break   continue
  do    endwhile     load    then     read
print  breakpoint    halt   inline
```

2.3 Operators and Delimiters

The following are the operators and delimiters in SPL

```
(   )   ;    [   ]   /   *   +   -   %
>   <   >=   <=  !=  ==  =   &&  ||  !
```

2.4 Registers

SPL allows the use of 30 registers for various operations. (R0-R7, S0 - S15, BP, SP, IP, PTBR, PTLR, EFR)

2.5 Identifiers

Identifiers are used as symbolic names for constants and aliases for registers. Identifiers should start with an alphabet but may contain alphabets, digits

and/or underscore (_). No other special characters are allowed in identifiers.

Examples: **var1**, **new_page** . Invalid identifiers include 9blocks , $n etc.

2.6 Literals

Integer and String literals are permitted in SPL. An integer literal is a sequence of digits representing an integer. Negative integers are represented with a negative sign preceding the sequence of digits.
A string literal is a sequence of characters which are enclosed within double quotes (" "). eg : `"alice"`

3 Register Set

SPL doesnt allow the use of declared variables. Instead a fixed set of registers is provided. The register set in SPL contains 30 registers. There is a direct mapping between these registers and the machine registers in XSM.

R0-R7	Program Registers
S0-S15	Kernel Registers
BP	Base Pointer
SP	Stack Pointer
IP	Instruction Pointer
PTBR	Page Table Base Register
PTLR	Page Table Length Register
EFR	Exception Flag Register

3.1 Aliasing

Any register can be referred to by using a different name. A name is assigned to a particular register using the **alias** keyword. Each register can be assigned to only one alias at any particular point of time. However, a register can be reassigned to a different alias at a later point. Aliasing can also be done inside the **if** and **while** block. However, an alias defined within the if and while blocks will only be valid within the block. No two registers can have the same alias name simultaneously.

4 Constants

Symbolic names can be assigned to values using the define keyword. Unlike aliasing, two or more names can be assigned to the same value. A constant can only be defined once in a program.

4.1 Predefined Constants

SPL provides a set of predefined constants. These predefined constants can be assigned to different values explicitly by the user using **define** keyword. These constants are mostly starting addresses of various OS components in the memory. The predefined set of constants provided in SPL are

The predefined set of constants provided in SPL are

Name	Default Value
SCRATCHPAD	512
PAGE_TABLE	1024
MEM_LIST	1280
FILE_TABLE	1344
READY_LIST	1536
FAT	2560
DISK_LIST	3072
EX_HANDLER	3584
T_INTERRUPT	4608
INTERRUPT	5632
USER_PROG	12800

5 Expressions

An expression specifies the computation of a value by applying operators to operands. SPL supports arithmetic and logical expressions.

5.1 Arithmetic Expressions

Registers, constants, and 2 or more arithmetic expressions connected using arithmetic operators are categorized as arithmetic expressions. SPL provides five arithmetic operators, viz., +, -, *, / (Integer Division) and % (Modulo operator) through which arithmetic expressions may be combined. Expression syntax and semantics are similar to standard practice in programming

languages and normal rules of precedence, associativity and paranthesization hold.
Examples:
(5*x) + 3
10 % 4

5.2 Logical Expressions

Logical expressions may be formed by combining arithmetic expressions using relational operators. The relational operators supported by SPL are

```
<, >, <=, >=, ==, !=
```

Standard meanings apply to these operators. A relational operator will take in two arguments and return 1 if the relation is valid and 0 otherwise.
The relational operators can also be applied to strings. `>, >, <=, >=` compares two strings lexicographically. `!=` and `==` checks for equality in the case of strings. eg:
"adam" < "apple" // This returns 1
"hansel" == "gretel" // This returns 0
Logical expressions themselves may be combined using logical operators, && (logical and) , || (logical or) and ! (not).

5.3 Addressing Expression

Memory of the meachine can be directly accessed in an SPL program. A word in the memory is accessed by specifying the addressing element, i.e. memory location within [ ]. This corresponds to the value stored in the given address. An arithmetic expression or an addressing expression can be used to specify the address.

Examples of addressing expressions:
[1024], [R3], [R5+[R7]+128], [FAT + R2] etc.

6 Statements

Statements control the execution of the program. All statements in SPL are terminated with a semicolon ;

6.1 Define Statement

Define statement is used to define a symbolic name for a value. Define statements should be used **before any other statement** in an SPL program. The keyword **define** is used to associate a literal to a symbolic name.

define *constant_name value;*

```
define DISK_BLOCK 437;
```

6.2 Alias Statement

An **alias** statement is used to associate a register with a name. **Alias** statements can be used anywhere in the program except within **if** and **while** statements.

alias *alias_name register_name ;*

```
alias counter S0;
```

6.3 Breakpoint Statement

The **Breakpoint** statement is used to debug the program. The program when run in debug mode pauses the execution at this instruction.

breakpoint;

This instruction translates to **BRKP** machine instruction.

6.4 Assignment Statement

The SPL assignment statement assigns the value of an expression or value stored in a memory address to a register or a memory address. = is the assignment operator used in SPL. The operand on the right hand side of the operator is assigned to the left hand side. The general syntax is as follows

Register / Alias / [Address] = Register / Number / String / Expression / [Address] ;

```
S0 = S2 * 10 + 5;
counter = counter + 1;
[PTBR + 3] = [1024] + 10;
S1 = "hello world";
```

6.5 If Statement

If statements specify the conditional execution of two branches according to the value of a logical expression. If the expression evaluates to 1, the **if** branch is executed, otherwise the **else** branch is executed. The **else** part is optional. The general syntax is as follows

if *(logical expression)* ***then***
    statements;
else
    statements;
endif;

6.6 While Statement

While statement iteratively executes a set of statements based on a condition. The condition is defined using a logical expression. The statements are iteratively executed as long as the condition is true.

while *(logical expression)* ***do***
    statements;
endwhile;

6.7 Break statement

Break statement is a statement which is used in a while loop block. This statement stops the execution of the loop in which it is used and passes the control of execution to the next statement after the loop. This statement cannot be used anywhere else other than while loop. The syntax is as follows

break *;*

6.8 Continue statement

Continue statement is a statement which is also used only in a while loop block. This statement skips the current iteration of the loop and passes the control to the next iteration after checking the loop condition. The syntax is as follows

continue *;*

6.9 ireturn Statement

ireturn statement or the Interrupt Return statement is used to pass control from kernel mode to user mode.

ireturn*;*

The **ireturn** is generally used at the end of an interrupt code. This instruction translates to **IRET** machine instruction.

6.10 Read/Print Statements

The **read** and **print** statements are used as standard input and output statements. The **read** statement reads a value from the standard input device and stores it in a register.
NOTE: String read or printed must not exceed 10 characters
The **print** statement outputs value of a register or an integer/string literal or value of a memory location.

read *Register;*
print *Register / Number / String / Expression / [Address];*

6.11 Load / Store Statements

Loading and storing between filesystem and memory is accomplished using **load** and **store** statements in SPL. **load** statement loads the block specified by *block_number* from the disk to the the page speficied by the *page_number* in the memory. **store** statement stores the page specified by *page_number* in the memory to the the block speficied by the *block_number* in the disk. The *page_number* and *block_number* can be specified using arithmetic expressions.

load *(page_number, block_number);*
store *(page_number, block_number);*

6.12 halt Statement

halt statement is used to halt the machine.

halt;

This instruction translates to **HALT** machine instruction.

6.13 Inline Statement

The **inline** statement is used give XSM machine instructions directly within an SPL program.

***inline** "MACHINE INSTRUCTION";*

```
inline "JMP 11776";
```