XOS
eXperimental Operating System
Version 1.0

Dr. K. Muralikrishnan
kmurali@nitc.ac.in
NIT Calicut

October 25, 2012

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 essential concepts of an operating system.

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

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

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 ...
- **Memory Management**, involves allocating memory for processes, demand paging (loading memory pages from the disk as and when required). Refer ...
- **System Calls**. XOS provides various system calls for the user processes to execute certain kernel level operations. Refer ...

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	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	Page Fault Handler	3584 – 4095	512
8	Timer Interrupt Routine	4096 – 4607	512
9	Interrupt 1 Routine	4608 – 5119	512
10	Interrupt 2 Routine	5120 – 5631	512
11	Interrupt 3 Routine	5632 – 6143	512
12	Interrupt 4 Routine	6144 – 6655	512
13	Interrupt 5 Routine	6656 – 7167	512
14	Interrupt 6 Routine	7168 – 7679	512
15	Interrupt 7 Routine	7680 – 8191	512
16 ⋮ 63	INIT and User Programs	8192 – 32767	512×48 $= 24576$

Fig. 2.1: Outline of the main memory

- **ROM Code**, Read Only Memory is hard coded assembly code which loads the opearting system startup code from the disk. Refer ...
- **OS Startup code**, loads the INIT process to memory, data structures

like FAT and Disk Free List, and Interrupt Routines from the disk. Refer ...

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

Chapter 3

OS Startup

3.1 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 (containing the OS startup code) into page 1 of the memory and to set the `IP` register value to 512 and start execution.

3.2 `INIT` Process

The Operating System starts execution with only a single user program - the `INIT` process. The `INIT` process code resides in the disk blocks 13 – 15 . During execution of the `INIT` process, 3 pages will be used for the code, and the 4th page of the logical address space will be used as the stack. The OS Startup Code is responsible for loading the `INIT` process into memory and starting its execution.

3.3 OS Startup Code

When the machine boots up, the *ROM* code loads the *OS startup code* into the memory. The OS startup code (instructions in page 1 of the memory) starts execution in the *Kernel mode*. It performs the following functions.

- It loads the Page Fault Exception Handler and Interrupt Service Routines from the blocks 1–9 of the disk into pages 7–15 of the memory.
- It loads the FAT from the block 10 of the disk into page 5 of the memory.
- It loads the Disk Free List from the block 11 of the disk into page 6 of the memory.

- It generates the Memory Free List and stores it in words 256–319 of page 2 of the memory.
- It loads the `INIT` process code from the hard disk into the memory by performing the following steps:
  - Sets the page table of the `INIT` process with disk address pointing to the code of the `INIT` process. (Refer ...)
  - Update the ready list and `PID` register.
  - Set the required page table entries.
  - Set the values of SP, BP and `IP` with values 1536 , 1536 and 0 respectively.
- Switch from *Kernel mode* to *User mode*.[1]

Note: All addresses are absolute addresses in Kernel mode.

Once the OS Startup is completed, the page in which the OS Startup Code resides in memory (Page 1) is used for loading files from the disk.

[1] This can be achieved by `IRET` instruction.

Chapter 4

Process Management

4.1 Introduction

Any program in its execution is called a **process**. Processes will be loaded into memory before they start their execution. Each process occupies **atmost 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, Refer ...

The XSM architecture supports demand paging doesn't 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 limitiations in number of PCBs in the Ready List (Refer ...) and the number of Per-Process Page Tables (Refer ...)

4.2 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.

Figure 4.1 shows the process structure.

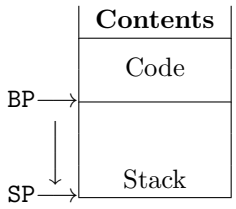

Fig. 4.1: Logical Address Space of a Process

4.3 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 **??**.

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

Fig. 4.2: Structure of Process Control Block

4.3.1 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

4.3.2 Registers

(`PID`, `SP`, `BP`, `IP`, `PTBR`, `PTLR`, `R0` – `R7`)

- Every process is allotted a unique integer identifier 0 – 32, known as the PID (Process Identifier) which is stored in the PID register. This register can be used as an operand in any instruction only when executing in the kernel mode. (Refer section **??**)
- The word address of the currently executing instruction is stored in the IP (Instruction Pointer) register. This register can be used as an operand in any instruction only when executing in the kernel mode.

- The base address of the user stack is stored in the BP (Base Pointer) register.
- The address of the stack top is stored in the SP (Stack Pointer) register.
- The physical address of the Per-Process Page Table of the process is stored in the PTBR (Page Table Base Register). Refer ...
- 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. Refer ...

Each process has its own set of values for the various registers. Words 1 – 6 in the PCB stores the values of the registers associated with the process .

4.3.3 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. upto 8 entries in the PCB. It is stored in the PCB from words 15 to 31. Its structure is given below

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

Fig. 4.3: Structure of Process Control Block

- The OS maintains a system wide open file table which contains details of all the files that are opened by processes (Refer ...). 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 opeartions.

4.4 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)

4.5 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.

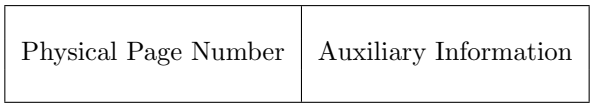

Fig. 4.4: Structure of a valid Page Table Entry

The Per-Process Page Table stores the physical page number corresponding to the logical page number of a process. The logical page number can vary from 0 to 4 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 pages)

When a process is loaded only the disk addresses of the program are stored, and by demand paging (Refer ...) pages are loaded/allocated for the process. Once all pages are loaded, each entry contains the page number where the data specified by the logical address resides in the memory and auxilary information which includes *valid/invalid bit*, *reference bit* etc. Refer ....

4.6 Running a process

Any program that the user wishes to execute is loaded into the memory in the following way. `INIT` process will return a filename to the OS corresponding to the program to be executed.

- The OS searches the memory copy of `FAT` for the address of the basic block. The basic block is loaded into Page 1 of the memory (earlier used for OS Startup)
- A Per-Process Page Table and PCB is initialized for the process with the first available PID by searching through the ready list for a PCB with `STATE` 0, i.e. *terminated* process. `PTBR` is loaded with the physical address of Per-Process Page Table and `PLTR` is set to 4.
- The Per-Process Page Table of the process is loaded with the disk addresses of the data blocks and the auxilary information is set as '00' (invalid and not referenced). The remaining entries are loaded with value 0, and auxilary information is set as '00'.
- The index, i of first 0 entry in the Per-Process Page Table is used to calculate `BP` and `SP` as $i \times 512$. `IP` is set to 0.

- The Per-Process Open File table in the PCB is filled with 0 entries initially.
- The process begins execution and pages are loaded as and when necessary upon page fault. (Refer ...)
- Once a process finishes its execution, the entry corresponding to its `STATE` in its PCB is set to 0.

4.7 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 Page 8 of the memory.

4.7.1 Scheduler

Whenever a timer interrupt occurs, the kernel temporarily halts the execution of the currently executing process and invokes the Timer Interrupt Service Routine.

Following are functionalities of the scheduler:

- The process switches from *User mode* to *Kernel mode* when an interrupt occurs.
- If a process is currently running, the scheduler saves the values of all the registers into the corresponding fields in the PCB of that process.
- The Ready List of PCBs is traversed from the current process' PCB in a circular fashion to get a (other than the INIT process' PCB [1]) with `STATE` value 1 (*ready*). The `STATE` is set to 2 (*running*) and the *PID* of the process is returned.
- If one such process is found, the `PID` is updated with the `PID` of this entry in the PCB. If no such process is found, then the `PID` is set 0 (`PID` of `INIT` process). Then all the registers of the machine are initialised with their corresponding values obtained from the `PCB` of the process specified by this PID.
- The process switches from *Kernel mode* to *User mode*.

[1] This can be accomplished by setting the PID of INIT process as 0 and searching the Ready List only from word 32 in page 3

Chapter 5

Memory Management

5.1 Introduction

XSM uses a paging mechanism for address translation (Refer ...). 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.

5.2 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 ...), 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. The structure of an entry in the page table is given below.

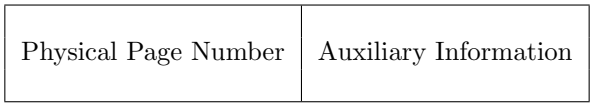

Fig. 5.1: Structure of Page Table Entry

- *Physical Page Number*: The logical page numbers generated by the CPU for a process can be in the range 0 to 3. The actual location of these logical pages are given by the physical page number. Address translation is done by the machine (Refer ...)

- *Auxilary Information*: The 2nd word for the entry contains auxilary information, which are a sequence of flags. The strucure of auxilary information is given below

0	1	2	...	15
VI	R	\0	...	\0

  - *Valid/Invalid Bit* (VI): is a value 1 or 0 for valid and invalid respectively. It is valid if and only if the entry is a physical memory location. It is invalid if either the entry points to location on the disk or if its not in the logical address space of the process.
  - *Reference Bit* (R): This bit is set to 1 by the machine every time a page is accesed. This bit is used for page replacement (Refer ...) by the OS.

5.3 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.

5.4 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. Thus increasing the total number of processes that can be run simultaneously on the OS. The stratgy used by XOS is pure demand paging.

5.4.1 Pure Demand Paging

XOS starts executing a process with no pages of that process in memory. Pages are loaded into the memory from the disk only when it is required by the CPU.

The page table is loaded with the disk address of the data blocks corresponding to the process and all entries are set as invalid and not referenced initially ('00' as auxilary information). Page Fault Exceptions occur when a required page is not currently loaded into the memory. The Page Fault Handler is responsible for loading the memory form the disk.

5.4.2 Page Fault Exception Handler

A Page Fault Exception is caused by the hardware whenever it translates the logical address to an invalid entry in the page table (indicated by the *valid/invalid* bit). XOS stores the disk addresses of the data blocks corresponding to the process in its page table initially.

The Page Fault Exception Handler takes the following actions dduring a page fault,

1. Check if a page is free in the memory free list. If no free page is found, using the page replacement technique described (Refer ...) a page is allocated in the memory.

2. If the Page Number field is a valid disk block number(non zero entry), the corresponding block from the disk is loaded into the allocated page. Otherwise the process is requesting for user stack space. The allocated page will be used as the user stack.

3. The page table is updated with allocated page's physical page number and the auxillary information is set as '10' (valid and not referenced). The reference bit will be automatically updddated by the machine.

5.4.3 Page Replacement

The page replacement technique used in XOS is a *second chance algorithm*(Refer ...) which uses the *reference bits* in the auxilary information. The steps are as follows

1. It scans the valid entries in the Per-Process Page Tables of all processes and checks for reference bits.

2. The scanning starts from the current page table entry which issued a page fault.

3.

Chapter 6

Halt System Call

6.1 System Calls

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

All the arguments to the system call are pushed as arguments into the user stack while calling the corresponding interrupt. The system call number is pushed as the last argument (Refer section **??** for calling convention).

6.2 Halt System Call

Syntax : `Halt()`
Syscall no : 0

The Halt system call is used to halt the machine. Halt system call invokes the interrupt INT 5. This interrupt consists of a single instruction, the HALT instruction, which halts the simulator.

Chapter 7

File System Calls

7.1 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 (if required).

The page 1 of the memory (fig **??**) 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.

7.2 Global File Table and Local File Table

Before explaining the system calls, we introduce two data structures : *Global File Table* and *Local File Table*.

- **Global File Table** It is a table consisting of a list of all the open files in the system. Refer fig **??** for location in memory. Since each of the 12 processes can open 4 files at a time, this table consists of a maximum of 48 entries. Each entry of the global file table has the following structure as shown in figure 7.1.

Fig. 7.1: Structure of a GFT entry

  - **FAT index entry :** It is used to index the memory copy of the file allocation table(section **??**) to get information about that particular file.

- **lseek :** It is used to get the current position of the next character that will be read from the file. By default, when a file is opened, this parameter has a value 0.

- **Local File table** In addition to the fields discussed earlier(section 4.3), the PCB has an additional field known as the *Local File Table*. The local file table consists of 4 entries each of size one word. Each entry corresponds to a file opened by that particular process and stores the global file table index of that file. Thus a process can open a maximum of 4 files.

  The local file table is indexed by a *file descriptor*(an integer value ranging from 0 to 3).

7.3 Modifications in the OS Startup Code

- The Global File Table in the memory must be initialised with NULL values.
- The Local File Table entries in the PCB of the INIT process must be initialised with NULL values.

7.4 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.

Interrupt specifications for different *File system calls* are as follows:

7.4.1 INT 1

The file system calls *Create* and *Delete* invoke INT 1. INT 1 handles these system calls as follows.

1. **Create :** This system call is used to create a new file in the file system whose name is specified in the argument.
   Syntax : `int Create(fileName)`
   Syscall no : 1

   - First of all, the memory copy of the FAT is searched for a free entry. If no free entry is found, an appropriate error code is returned.
   - Next, the memory copy of the disk free list is searched to find a free block number.If no free block is found, an appropriate error code is returned. This block is used as the basic block of the file to be created.

- The `fileName` specified in the argument and the free block number obtained in the previous step are stored in the *file name* field and *basic block number* field of the free FAT entry, respectively.
- The *file size* field of the FAT entry is initialized to zero.
- Each entry of the block list in the basic block is initialized to zero.[1]
- The updated copies of FAT and disk free list in the memory are committed to the disk.
- The return value of this system call is 0 in case of success and the appropriate error code in case of failure.

2. **Delete :** This system call is used to delete the file from the file system whose name is specified in the argument.
   Syntax : `int Delete(fileName)`
   Syscall no : 2

   - The memory copy of the FAT is searched using the `fileName` to get the corresponding FAT entry. If no entry is found, an appropriate error code is returned.
   - If the file is already open an appropriate error code is returned. We adopt the following steps to check if the file is open.
     - The *FAT index entry* of each global file table entry is used to fetch the filename of the corresponding open file from the memory copy of the FAT .
     - Each of the filenames obtained in the previous step is compared with the `fileName`. If match is found, we conclude that the file is currently in open.
   - The *basic block number* field in this FAT entry obtained, is then used to load the basic block of the file into the scratchpad.
   - Each entry in the block list of the basic block is used to find the data blocks of the file. Then, entries in the memory copy of the disk free list corresponding to these data blocks are set to zero, thereby freeing them.
   - Finally, the FAT entry of the file is removed.
   - The updated copies of FAT and disk free list in the memory are committed to the disk.
   - The return value of this system call is 0 in case of success and the appropriate error code in case of failure.

[1] This can be achieved by loading the basic block into the scratchpad, updating it and then committing back the updated basic block.

7.4.2 INT 2

The file system calls *Open* and *Close* invoke INT 2. INT 2 handles these system calls as follows.

1. **Open :** This system call is used to open an existing file whose name is specified in the argument.
   Syntax : `int Open(fileName)`
   Syscall no : 3

   - First of all, a free entry is searched in the local file table of the process. If there are no free entries, in the case where a process already has 4 open files, an appropriate error code is returned.
   - Then, the global file table is searched for a free entry. If there is no free entry, an appropriate error code is returned else a new global file table entry is created and the fields are filled with appropriate values in the following manner:
     - The memory copy of FAT is searched using the `fileName` and the corresponding index of that file in the FAT [2] is stored as the *FAT index*. If the file does not have an entry in the FAT, an appropriate error code is returned.
     - The *lseek* field is set to zero.
   - The index of this global file table entry is stored in its local file table.
   - The index of this entry in the local file table is returned as a return value of the system call. This is known as the file descriptor.

2. **Close :** This system call is used to close an open file. The file can only be closed by the process which opened it or by its children.
   Syntax : `int Close(fileDescriptor)`
   Syscall no : 4

   - The `fileDescriptor` is used first to access the local file table entry of the file. An appropriate error code is returned if the `fileDescriptor` is out of the range specified.
   - The global file table entry indexed by this local file table entry is removed. [3]
   - The local file table entry of the process is then removed.
   - The return value of this system call is 0 in case of success and the appropriate error code in case of failure.

[2] By index, we mean the sequential position (starting from 0) of that entry in the data structure mentioned.

[3] A suggested way to remove an entry is to store an integer -1 in that word.

7.4.3 INT 3

The file system calls *Read* and *Seek* invoke INT 3. INT 3 handles these system calls as follows.

1. **Seek :** This system call is used to change the current value of the seek position in the global file table entry of a file.
   Syntax : `int Seek(fileDescriptor, lseek)`
   Syscall no : 5

   - The `fileDescriptor` is used first to access the local file table entry of the file. An appropriate error code is returned if the `fileDescriptor` is out of the range specified.
   - This local file table entry is then used to access the global file table entry of the file.
   - Then the FAT index field in the global file table entry is used to access the FAT entry of the file.
   - The *file size* got from this FAT entry is checked to be greater than `lseek`. Otherwise an appropriate error code is returned.[4]
   - The *lseek* field in the GFT entry is then changed to the new value specified in the argument (`lseek`).
   - The return value of this system call is 0 in case of success and the appropriate error code in case of failure.

2. **Read :** This system call is used to read data from an open file.
   Syntax : `int Read(fileDescriptor, mem_loc, numWords)`
   Syscall no : 6

   - First of all, the basic block of the file specified by the `fileDescriptor` is loaded in the scratchpad. This is done in the following way:
     - The `fileDescriptor` is used first to access the local file table entry of the file. An appropriate error is returned if the `fileDescriptor` is out of the range specified.
     - This local file table entry is then used to access the global file table entry of the file.
     - Then the *FAT index* field in the global file table entry is used to access the FAT entry of the file.
     - The basic block address present in the FAT entry is then used to load the basic block (containing block list and file header info) into the scratchpad. Refer figure 7.2.
   - The *lseek* position present in the GFT entry and `numWords` are used to index the block list in the basic block to find the address of the block(s) to be read.

[4] Seek is allowed only *within* a file.

- Each time the block to be read is loaded into the scratchpad before reading its contents.
- The contents read are then copied into the buffer that is specified as an argument to the system call (`mem_loc`). If the `mem_loc` is out of the address space of the process, an appropriate error code is returned.
- The return value of this system call is the number of words successfully read. In case of an error, an appropriate error code is returned.

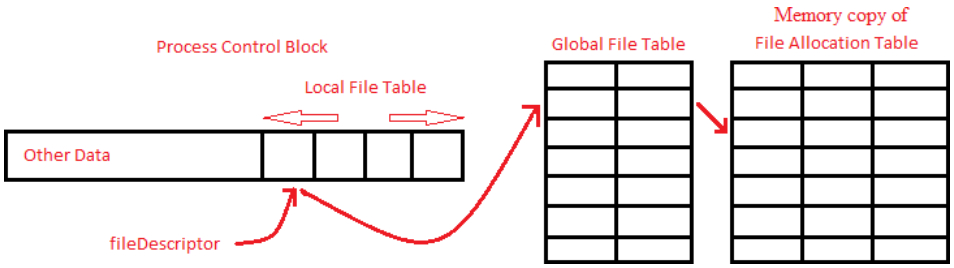

Fig. 7.2: Diagram showing the method of accessing FAT entry

7.4.4 INT 4

The file system call *Write* invoke INT 4. INT 4 handles these system calls as follows.

Write : This system call is used to write data into an open file.
Syntax : `int Write(fileDescriptor, mem_loc, numWords)` [5]
Syscall no : 7

- First of all, the basic block of the file specified by the `fileDescriptor` is loaded into the scratchpad. This is done in the following way:
  - The `fileDescriptor` is used first to access the local file table entry of the file. An appropriate error is returned if the `fileDescriptor` is out of the range specified.
  - This local file table entry is then used to access the global file table entry of the file.
  - Then the FAT index field in the global file table entry is used to access the FAT entry of the file.
  - The basic block address present in the FAT entry is then used load the basic block (containing block list and file header info) into the scratchpad. Refer figure 7.2.

[5]It is advisable to have a maximum of 1 block for any data file if it has to be modified using `write` system call since if the modification spans multiple blocks the entire procedure to access a block (outlined above) has to be repeated.

- The lseek position present in the GFT entry and `numWords` are used to index the block list in the basic block to find the block numbers of the block(s) to be written into. [6]
- Each time the block to be written into is loaded into the scratchpad before performing the write operation.
- After loading the specified block, the content to be written is copied from the user memory location (`mem_loc`) into this block. If `mem_loc` is out of the address space of the process, an appropriate error code is returned.
- If the write operation exhausts all the currently allocated blocks, new blocks are allocated as required. This is done in the following way.
  - The memory copy of the disk free list is used to get the block number of a free block.
  - A new basic block entry is created using this free block number and added to the block list of the basic block. Successive write operations are then performed the usual way.
- Once all the write operations are over for that block, it is stored back into the disk.
- The updated copies of FAT and disk free list in the memory are committed to the disk.
- The return value of this system call is the number of words successfully written. In case of an error, an appropriate error code is returned.

[6] The data block to which the lseek position is pointing to is got by dividing lseek by the block size.
The data block number calculated above is used to index the block list in the basic block to get the exact location of the data block in the disk. The data block is then loaded from the disk into the scratchpad.
If the words to be read are split across multiple data blocks, the above procedure is repeated.

Chapter 8

Multiprogramming

To support multiprogramming in the system, the kernel makes use of the *scheduler* which is present in the interrupt service routine INT 0[1].

8.1 Scheduler

Whenever a timer interrupt occurs, the kernel temporarily halts the execution of the currently executing process and invokes INT 0. Refer book [**?**] for more details. Following are functionalities of the scheduler:

- If a process is currently running, the scheduler saves the values of all the registers into the corresponding fields in the PCB of that process.
- The scheduler scans the ready list starting from the current PID and checks for the presence of a process other than the INIT process.[2] If one such process is found, the PID is updated with the index of this entry in the ready list. If no such process is found, then the PID is set to the index of the INIT process in the ready list. Then all the registers of the machine are initialised with their corresponding values obtained from the PCB of the process specified by this PID.
- The process switches from *Kernel mode* to *User mode*.

[1] Unlike other interrupts, INT 0 is called by the machine and not by the user program.

[2] This can be accomplished by setting the PID of INIT process as 0 and searching only the entries from 1–11 in the ready list.

Chapter 9

Process System Calls

9.1 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.

Interrupt specifications for different *Process system calls* are as follows:

9.1.1 INT 5

The process system call *Fork* invokes INT 5. INT 5 handles these system calls as follows.

Fork : This system call is used to create a new process having the same code area, data area and list of open files as that of the process which invoked this system call.

The new process that is created is known as the *child* process, and the process which invoked this system call is known as its *parent*.

The register values in the PCB of the child process are initialized with the current register contents.
Syntax : `int Fork()`
Syscall no : 8

- A vacant entry is searched for in the *Ready list*.
- If no entry is found, in the case when there are already 12 processes that are active, an appropriate error code is returned.
- The index of this vacant ready list entry is the PID for the child process that is created.

- The PID entry in the PCB of the child process is updated with this new PID.
- All the registers (except PID) and the local file table of the parent process is replicated in the PCB of the child process.
- The code pages, the data page and the stack page of the parent process is replicated for this child process.
- The control is returned back to the parent process.
- The return value of this system call is the PID of the child process.

9.1.2 INT 6

The process system call *Exec* invokes INT 6. INT 6 handles these system calls as follows.

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 and start its execution .
Syntax : `int Exec(filename)`
Syscall no : 9

- The entire process area of the currently executing process is replaced by that of the program specified in the argument (`filename`).
- If the file specified by `filename` is not an executable [1] then, an appropriate error code is returned.
- The memory copy of the FAT is searched to get the location of the basic block of the file specified by `filename`, which is then loaded into the scratchpad.
- This is then used to get the location in the disk of the blocks of the file to be loaded.
- The 2 code blocks and 1 data block of the file are loaded from the disk into the corresponding locations in the memory of the code blocks and data block of the current process.
- The PCB of the current process is modified to hold the values for that of the new process. The PID and page table, however, remains unchanged.[2]
- The return value of this system call is 1 in case of a failure. Nothing is returned in case of a success.

[1] Executables in E`SIM` must end with an extension `.sim`

[2] This is because the mappings remain the same as the code blocks and data block of the specified executable are loaded into the same locations as of the current process. Since, no new process table entry is created, the PID also remains the same.

9.1.3 INT 7

The process system call *Exit* invokes INT 7. INT 7 handles this system call as follows. **Exit :** This system call is used to terminate the execution of the process which invoked it and removes it from the memory . It loads the next available process.
Syntax : `Exit()`
Syscall no : 10

- The entire address space of the currently executing process is set free by setting a value 0 in the memory free list corresponding to the pages occupied by that process.
- The local file table is traversed and the global file table entry is removed.
- The ready list entry corresponding to this process is set to zero thereby releasing all the data structures used by the process (fig **??**).
- The ready list is then searched for the next available process. The INIT process is excluded in this search.[3] If one such process is found, the PID is updated with the index of this entry in the ready list. If no such process is found, then the PID is set to the index of the INIT process in the ready list.
- All the registers of the machine are initialised with their corresponding values obtained from the PCB of the process specified by the new PID.
- The process switches from *Kernel mode* to *User mode*.

9.2 INIT Process

The INIT process is the first user process loaded by the OS on the OS startup. INIT was previously defined in chapter 3 as a normal user program. Since multiprogramming functionalities have been added to the OS, INIT must be modified. The modified specification of INIT process is as follows:

- It provides an interface for the users to run other user programs.
- The user enters the name of a valid executable file (which should be made available in the disk) in the shell. If the specified file is not found, an appropriate error code is returned.
- If the specified executable file is found, the INIT process forks and does exec on the that file.
- Entering the keyword HALT instead of the name of an executable file invokes the Shutdown system call.

[3]This can be accomplished by setting the PID of INIT process as 0 and searching only the entries from 1–11 in the ready list.

All the user processes other than INIT are added to entries 1-11 of the ready queue keeping the 0th entry (corresponding to INIT) untouched. INIT loads the first process and thereafter all context switches occur among the other processes in the ready queue. INIT is switched back only when the ready queue (entries 1-11) is free so that the user can load another executable file via the shell.