XSM
eXperimental String Machine
Version 1.0

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

October 24, 2012

Contents

1 Introduction **3**

1.1 Operating System Functionality 3

1.1.1 Process Management 3

1.1.2 Multiprogramming 3

1.1.3 System Calls 3

2 Memory **4**

2.1 Introduction 4

2.2 Page Table 4

2.3 Address Translation 6

2.4 Memory Free List 7

3 Process **9**

3.1 Introduction 9

3.2 Process Structure 9

3.3 Registers Associated with a Process 10

3.4 Data Structures Associated with a Process 10

3.4.1 Ready List 10

3.4.2 Process Control Block (PCB) 11

3.4.3 The Page Table 11

3.5 Storage Details of the Data Structures 11

3.5.1 Ready List 11

3.5.2 Page Tables 13

3.5.3 Process Table 13

4 OS Startup **14**

4.1 ROM Code 14

4.2 OS Startup Code Specification 14

4.3 INIT Process 15

5 Halt System Call **16**

5.1 System Calls 16

5.2 Halt System Call 16

6 File System Calls **17**

6.1 Scratchpad 17

6.2 Global File Table and Local File Table 17

6.3 Modifications in the OS Startup Code 18

6.4 File System Calls 18

6.4.1 INT 1 18

6.4.2 INT 2 20

6.4.3 INT 3 21

6.4.4 INT 4 23

7 Multiprogramming **25**

7.1 Scheduler 25

8 Process System Calls **26**

8.1 Process System Calls 26

8.1.1 INT 5 26

8.1.2 INT 6 27

8.1.3 INT 7 28

8.2 INIT Process 29

Chapter 1

Introduction

1.1 Operating System Functionality

There are various functionalities associated with the operating system which are essential for the user programs to run and make use of the system resources. The functionalities and their details are explained below.

1.1.1 Process Management

Any program that the user wishes to execute is loaded into the memory (A program in memory is known as a process). For creating a new process,

1.1.2 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*. Refer chapter 7 to know more about multiprogramming.

1.1.3 System Calls

A process needs resources like disk, memory etc while executing. The OS caters to these needs of the process by providing an interface known as the *system call interface*. Refer chapter 6 and chapter 8 to know more about system calls.

Chapter 2

Memory

2.1 Introduction

- The basic unit of memory in the ESIM architecture is a word.
- The machine memory can be thought of as a linear sequence of words.
- A collection of 256 contiguous words is known as a *page*.
- The total size of the memory is 64 pages or 16384 (256×64) words.
- Each word in the memory is identified by the *word address* in the range 0 to 16383($256 \times 64 - 1$). Similarly, each page in the memory is identified by the *page number* in the range 0 to 63.

Example 2.1.1 *The* 256^{th} *word of the memory has a word address 255 and belongs to page 0. In general, the* n^{th} *word has the word address* $(n-1)$, *where* $1 \leq n \leq 16384$ *and belongs to the page* $\lfloor \frac{n-1}{256} \rfloor$. *Refer figure 2.2.*

2.2 Page Table

Before explaining the page table, we explain two well known terms:

- **Logical address :** It is the CPU generated address of the data.
- **Physical address :** It is the exact location of the data in the main memory.

Page no	**Contents**	**Word addr**
0	ROM code	0 – 255
1	OS Startup code	256 – 511
2	Static Page Tables	512 – 559
	Memory Free List	560 – 623
	Global File Table	624 – 719
	Ready List	720 – 731
	Unallocated	732 – 767
3	Process Table	768 – 959
	Unallocated	960 – 1023
4	File Allocation Table	1024 – 1535
5		
6	Disk Free List	1536 – 2047
7		
8	INIT process	2048 – 2815
9		
10		
11 – 55	⋮ User Programs ⋮	2816 – 14335
56	INT 0	14336 – 14591
57	INT 1	14592 – 14848
⋮	⋮	⋮
63	INT 7	16128 – 16383

Fig. 2.1: Outline of the main memory

Word address		**Page no.**
0	1^{st} word	0
1	2^{nd} word	
⋮	⋮	
255	256^{th} word	
⋮	⋮	$\lfloor \frac{i}{256} \rfloor$
i	$(i+1)^{th}$ word	
⋮	⋮	
⋮	⋮	63
⋮	⋮	
$256 \times 64 - 1$	$(256 \times 64)^{th}$ word	

Fig. 2.2: Illustration of memory addressing

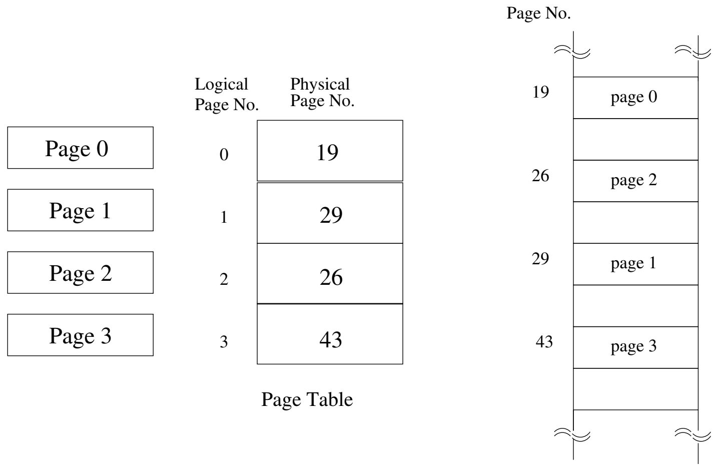

Fig. 2.3: Paging model of the ESIM architecture

Refer “Memory management strategies” in the book [**?**] to know more about paging.

The page table contains information relating to the actual location in the memory, i.e., the physical address, of the data specified by the logical address. Each entry of a page table contains the page number in the memory where the data specified by the logical address resides. Refer figure 2.3.

2.3 Address Translation

It is the process of obtaining the physical address from the logical address. It is done by the machine in the following way. Refer book [**?**] for more details.

1. The logical address generated by the CPU is divided by the page size (256) to get the *logical page number*.

2. The remainder got after performing the above division gives the *offset* within that page.

3. The *logical page number* is then used to index the page table to get the corresponding *physical page number* in the memory.

4. The *offset* got in step 2 is then used to refer to the word in the physical page containing the data.

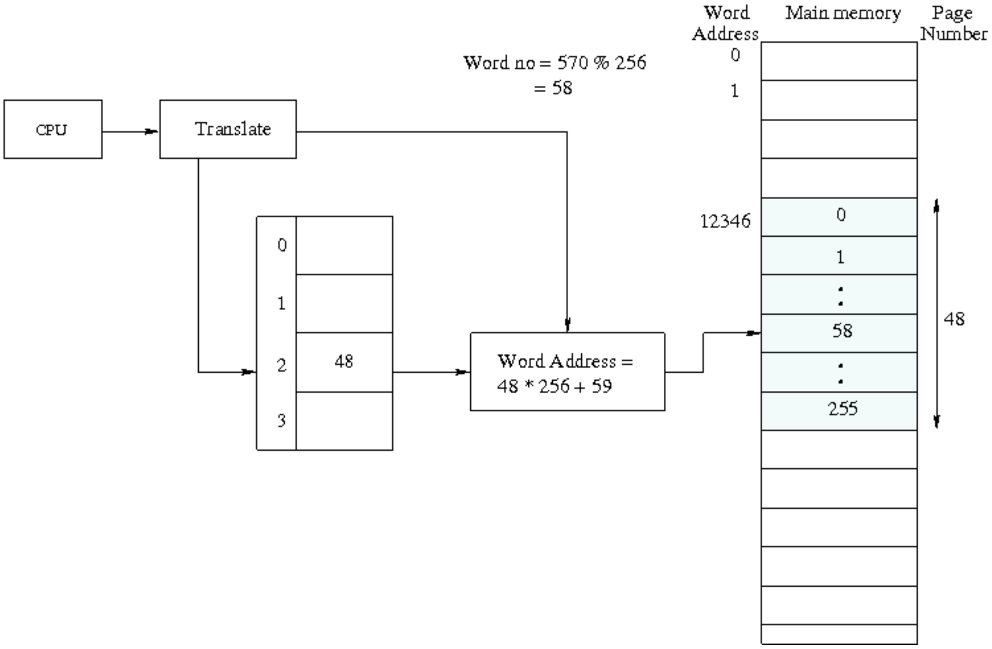

Fig. 2.4: Diagram illustrating address translation

Example 2.3.1 *Consider the scenario in figure 2.4. Here the logical address generated is 570, so the page number is* $\lfloor 570/265 \rfloor = 2$ *and word address is* $570 \ mod\ 256 = 58$*. The looked up value from the page table is* 48*. Thus the resultant physical address is* $48 \times 256 + 58$*.*

2.4 Memory Free List

- The free list of the memory consists of 64 entries. Each entry is of size one word.
- The total size of the free list is thus 64 words (64 (= no. of entries) × 1 (= size of one entry) = 64 words).
- It is present in the second 64 words of page 2 of the memory. Refer figure **??**.
- 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.

Example 2.4.1 *Figure 2.5 indicates that pages 0, 1 and 63 of the memory are not free while pages 2 and 48 are free.*

Pg no.	**Contents**
0	1
1	1
2	0
⋮	⋮
48	0
⋮	⋮
63	1

Fig. 2.5: A sample free list of the memory

The entire structure of memory is outlined in figure **??**.

- The ready list is searched for an entry with value 0. The corresponding entry found is set to 1 and the index of this entry is returned as the PID of the process. If no free entry is found, an appropriate error code is returned.

- The page table for the process is initialized as follows :

  1. The 1^{st} entry of the page table contains the page number of the memory where the first code block of the program has been loaded.
  2. The 2^{nd} entry of the page table contains the page number of the memory where the second code block of the program has been loaded.
  3. The 3^{rd} entry of the page table contains the page number of the memory where the data block of the program has been loaded.
  4. The 4^{th} entry of the page table contains the page number of the memory reserved for the stack.

- Set the values of BP, SP and IP in the PCB as 768, 768 and 0 respectively.

- Once a process finishes its execution, the entry corresponding to it in the ready list is set to 0.

Chapter 3

Process

3.1 Introduction

Def 1 ***Process :*** *Any program written by the user is run as a process by the kernel.*

- The ESIM architecture supports a maximum of 12 processes to be run at a time.
- Each process occupies 4 pages of the memory.

3.2 Process Structure

A process in the memory has the following structure.

- **Code Area :** These are pages of the memory that contain the actual code to be run on the machine. It occupies 2 pages of the memory.
- **Data Area :** This section consists of string data that is used in the code which cannot be stored in a register. It occupies 1 page of the memory.
- **Stack :** This is the user stack used in program execution. It is used to pass arguments during function calls, storing activation record of a function etc. It occupies 1 page of the memory and grows in the direction of increasing word address.

Figure 3.1 shows the process structure.

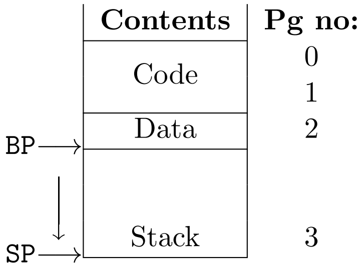

Fig. 3.1: Process Structure in memory. Arrow shows the direction of stack growth

3.3 Registers Associated with a Process

- Every process is allotted a unique integer identifier in the range 0 to 11, 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 **??** to know about the modes of operation)
- 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.

Each process has its own set of values for the various registers.

3.4 Data Structures Associated with a Process

The following are the various data structures associated with a process. They are explained in the following subsections.

3.4.1 Ready List

The *ready list :* is the data structure that maintains a circular list of all the active processes. Each entry of the ready list contains a value of either 1 or 0 indicating whether the corresponding process in the memory is present in the list or not.

3.4.2 Process Control Block (PCB)

It contains data pertaining to the current state of the process. Refer figure 3.2.

0	1	2	3	4–11	12–15
PID	BP	SP	IP	R0 – R7	Local File Table

Fig. 3.2: Structure of Process Control Block

Note that the size of each PCB (Process Control Block) is 16 words.

3.4.3 The Page Table

The *page table* stores the exact location in the memory of the data related to a process.

- Each process has 4 entries in the page table.
  - The zeroth entry corresponds to the first page of code area.
  - The first entry corresponds to the second page of code area.
  - The third entry corresponds to the data area.
  - The fourth entry corresponds to the stack.
- Each entry contains the page number where the data specified by the logical address resides in the memory. Refer figure 2.3.

3.5 Storage Details of the Data Structures

The data structures used by the processes are stored statically in the memory. Their storage details are as follows.

3.5.1 Ready List

- The ready list is located in words 209–220 of page 2 of the memory (refer fig **??**).
- The size of each ready list entry is one word.
- There are a total of 12 processes, thus accounting for the 12 words (12 × 1 word).

Pg no.	**Contents**
0	
1	
2	Static Page Tables Memory Free List Global File Table
3	Ready List Process Table
	⋮
7	
8 – 55	⋮ User Programs ⋮
56 – 63	INT 0 – 7
	⋮

(a) Main Memory

Word Address	Process
0	0
1	
2	
3	
⋮	
$4i$	i
$4i+1$	
$4i+2$	
$4i+3$	
⋮	
44	11
45	
46	
47	

(b) Structure of Page Table

Word Address	Process
0	0
1	1
2	2
⋮	
10	10
11	11

(c) Structure of Ready List

Word Address	Process
0	0
1	
⋮	
15	
⋮	
$16i$	i
$16i+1$	
$16i+2$	
⋮	
$16i+15$	
⋮	
176	

- All active processes have an entry 1 in the ready list corresponding to the location indexed by their respective PIDs.

3.5.2 Page Tables

- The page tables of the 12 processes are stored in the first 48 words of page 2 of the memory. Refer figure **??**.
- The size of each page table is 4 words (4(= no. of entries) × 1(= size of an entry)= 4 words).
- There are a total of 12 processes, thus accounting for the 48 words(12 × 4 words).
- The page tables are indexed by multiplying the PID of a process by the size of a page table to get the starting word address of the page table of that process. The indexing mechanism is illustrated in figure 3.3.

3.5.3 Process Table

- The page 3 of the memory contains the process table. Refer figure **??**.
- The process table contains the PCB of each of the 12 processes (Each entry occupies 16 words).
- There are a total of 12 processes, thus accounting for the 192 words (12 × 16 words).
- The process table is indexed by multiplying the PID of a process by the size of a PCB to get the starting word address of the PCB of that process. The indexing mechanism is illustrated in figure 3.3.

Chapter 4

OS Startup

4.1 ROM Code

It is a hard coded assembly level code present in page 0 of the memory. Refer figure **??**. It is also known as the ROM (Read Only Memory) code since 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 256 and start execution.

4.2 OS Startup Code Specification

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

- It loads the Interrupt Service Routines from the blocks 1–8 of the hard disk into pages 56–63 of the memory.

- It loads the FAT from blocks 11 and 12 of the hard disk into pages 4 and 5 of the memory.

- It loads the disk free list from Blocks 9 and 10 into pages 6 and 7 of the memory.

- It generates the memory free list and stores it in words 48–111 of page 2 of the memory.

- It loads the INIT process from the hard disk into the memory by performing the following steps:
    - Load the INIT process from blocks 14–16 of the hard disk to pages 8–10 of memory. Page 11 is allocated as the user stack.
    - Update the memory free list.
    - Update the ready list and PID register.
    - Set the required page table entries.
    - Set the values of SP, BP and IP with values 768, 768 and 0 respectively.
- Switch from *Kernel mode* to *User mode*.[1]

Note: All addresses are absolute addresses in Kernel mode.

4.3 INIT Process

The Operating System currently supports execution of only a single user program - the INIT process. Testing of the OS startup code can be done by loading the required user program as the INIT process. Modification to INIT will be done later.

[1] This can be achieved by calling IRET.

Chapter 5

Halt System Call

5.1 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).

5.2 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 6

File System Calls

6.1 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.

6.2 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 6.1.

FAT Index Entry	lseek

Fig. 6.1: 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 3.4.2), 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).

6.3 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.

6.4 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:

6.4.1 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 .

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

    - 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.

6.4.2 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.

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

    - 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.

6.4.3 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.

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

- 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 6.2.
   - 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.
   - 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.

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

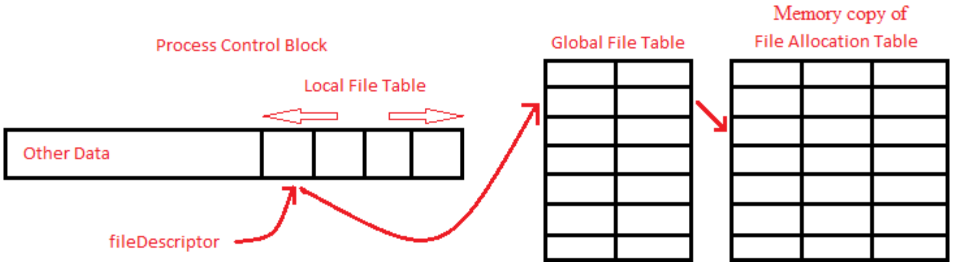

Fig. 6.2: Diagram showing the method of accessing FAT entry

6.4.4 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 6.2.
- The lseek position present in the GFT entry and `numWords` are used to index the block list in the basic block to find the block numbers of the block(s) to be written into. [6]

[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.

[6] The data block to which the lseek position is pointing to is got by dividing lseek by the block size.

- 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.

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 7

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].

7.1 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 8

Process System Calls

8.1 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:

8.1.1 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.

8.1.2 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.

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

- 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.

8.1.3 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 3.3).

- 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*.

[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.

[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.

8.2 INIT Process

The INIT process is the first user process loaded by the OS on the OS startup. INIT was previously defined in chapter 4 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.

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.