# MACHINE, FILE SYSTEM AND OPERATING SYSTEM SPECIFICATION

ALBIN SURESH RAMNATH J SUMESH B

June 25, 2012

# Contents

# Figures

# Chapter 1

# Introduction

## 1.1 Background

A preliminary proposal for an elementary operating system was made in [GDKI11, KAG+11]. Our work involved the critical analysis of the machine specification, Operating System specification and the implemented code.

## 1.2 Motivation

The experimental operating system, NACHOS [CPA93], which is currently used by the students for Operating Systems laboratory has several drawbacks.

The main drawback of NACHOS is the fact that the operating system kernel is not running on the simulated machine's memory. The operating system runs outside the simulated machine which is conceptually wrong. Another drawback is the fact that the conceptual knowledge gained by a student working on NACHOS is not proportional to the manual work that a student has to put into it. So it was decided to design a simple architecture without any such drawbacks and provide a better and simpler interface to write the operating system using this architecture.

## 1.3 Structure of the project

This project was initiated with the aim of creating a one-semester course in operating system that covers the basics of operating system and gives a hands-on experience in writing a simple operating system. The machine corresponding to this architecture can be simulated by a simulator and the operating system, written by the student, will be running on the simulator.

This project in its entirety can be described as consisiting of five main stages.

1. The first stage consisted of designing a detailed specification for the machine as well as the Operating System. The machine was chosen as the extended version of SIM and was called ESIM . String data type and operations were added to SIM machine to convert it into ESIM . A detailed specification of the operating system to be implemented was also developed. This was done by [GDKI11] and [KAG+11]. These specifications were critically reviewed and modifications were done. Refer chapter 2 and chapter 11 for more details.

2. The second stage consisted of implementing the machine and file system. Implementation details for machine and file system are given in chapters 8 and 10. This was one of our primary tasks.

3. The third stage consisted of designing two compilers APSIL and SPSIL. This was done in [MKS12b] and [MKS12a]. SPSIL is the compiler which will be used by the students to write the Operating System code. APSIL is the compiler which will be used by the students to write programs to test the Operating System they have written. Complete documentation of SPSIL and APSIL are included in the appendix.

4. The fourth stage consisted debugging the machine, file system and the two compilers. This was primarily done by writing the Operating System code and checking for bugs.

5. The final stage consisted of integrating all these documentations and creating an environment where students can consult while doing the lab.

# Part I

# Machine Specification

# Chapter 2

# Introduction

## 2.1 Introduction

A detailed operating system specification was done in the work of [KAG+11]. This documentation was critically reviewed and modifications were done in many places to comply with our new design. The major modifications done are the following:

- Addition of 8 Kernel registers and 4 Temporary registers (Refer chapter 3).
- The ready queue data structure in the previous design was replaced with a ready list data structure for simplicity in design and implementation (Refer chapter 5).
- The maximum number of processes supported by the Operating System was reduced to 12 from 16 due to space constraints in memory (Refer chapter 5
- Interrupt specifications were modified due to the size constratits of interrupt code (Refer chapter 7).
- The memory layout was modified to incorporate the new design decisions (Refer chapter 4).

## 2.2 Brief Machine Description

The machine simulator is known as Extended Simple Integer Machine (ESIM ). It is an interrupt driven uniprocessor machine.

## 2.3 Components of the Machine

The various components of the machine are :

- **Disk :** It is a non-volatile storage that stores user programs (executables) and data files.
- **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 controller* [1].

[1] **DMA controller :** DMA (Direct Memory Access) is the hardware device in a real machine that facilitates the transfer of data from disk to the memory and vice versa

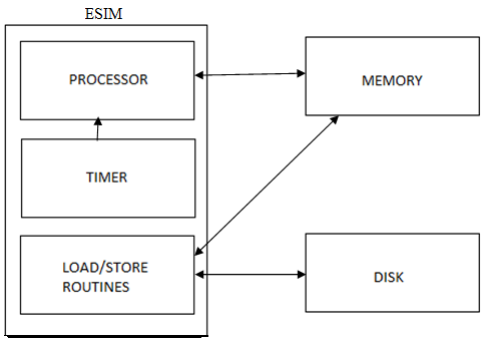



**Fig. 2.1:** Components of the Machine

## 2.4 Data types

The fundamental types supported by the machine are *integer* and *string*. A string is a sequence of characters terminated by `'\0'`. The machine interprets a single character also as a string.

**Example 2.4.1.** *The character "s" is stored as "s*`\0`*" in the memory and the word "ESIM" is stored as "ESIM*`\0`*" in the memory.*

`ESIM` supports a maximum string length of *16*.

# Chapter 3

# Registers

## 3.1 Introduction

The ESIM architecture maintains **12** registers each of size 1 word.

**Def 1.** ***Word :*** *It is the basic unit of memory.*

Each register can hold either an integer or an address of a string.

## 3.2 Register Set

There are 8 *General Purpose Registers*, R0–R7, which the user programs can use directly. These are followed by another 8 *Kernel Registers*, S0–S7 which are used only by the kernel. There are an additional 4 *Temporary Registers*, T0–T3 which are used by the compiler.[1] There are also 4 additional special purpose registers BP, IP, SP and PID which are used as Base pointer, Instruction pointer, Stack pointer and Process Identifier respectively. Figure 3.1 summarises the various registers and the sections where they are referred.

| Name | Register | Section |
|---|---|---|
| General Purpose Registers | R0–R7 | Used by the user programs to store data during various operations (Refer section 6.3.1 for the operations supported). |
| Kernel Registers | S0–S7 | Used by the OS to store data during various operations.(Refer section 6.3.2 for the operations supported). |
| Temporary Registers | T0–T3 | Used by the translator for storing intermediate data. |
| Stack Pointer | SP | Section 5.3 |
| Base Pointer | BP | Section 5.3 |
| Instruction Pointer | IP | Section 5.3 |
| Process Identifier | PID | Section 5.3 |

**Fig. 3.1:** Summary of the registers in ESIM architecture

[1] It is recommended that the programmer, system or otherwise, not use these temporary registers.

# Chapter 4

# Memory

## 4.1 Introduction

| 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 | ⋮<br>User Programs<br>⋮ | 2816 – 14335 |
| 56 | INT 0 | 14336 – 14591 |
| 57 | INT 1 | 14592 – 14848 |
| ⋮ | ⋮ | ⋮ |
| 63 | INT 7 | 16128 – 16383 |

**Fig. 4.1:** Outline of the main memory

- 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 ($256 \times 64$) 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 4.1.1.** *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 4.2.*

| 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. 4.2:** Illustration of memory addressing

## 4.2 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.

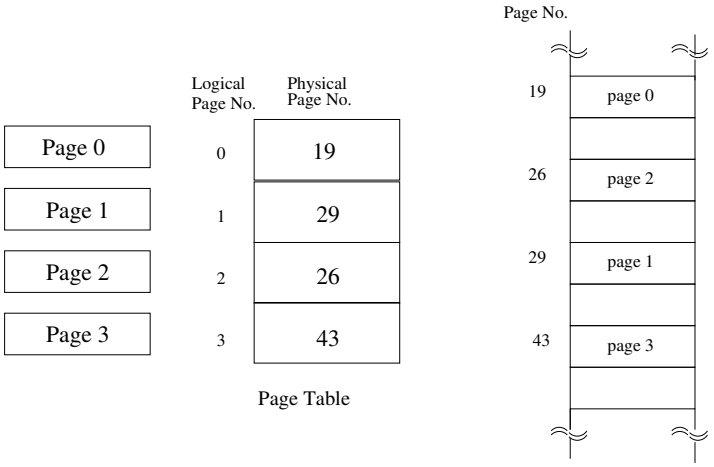



**Fig. 4.3:** Paging model of the ESIM architecture

Refer "Memory management strategies" in the book [SGG05] 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 4.3.

## 4.3 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 [Bac86] 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.

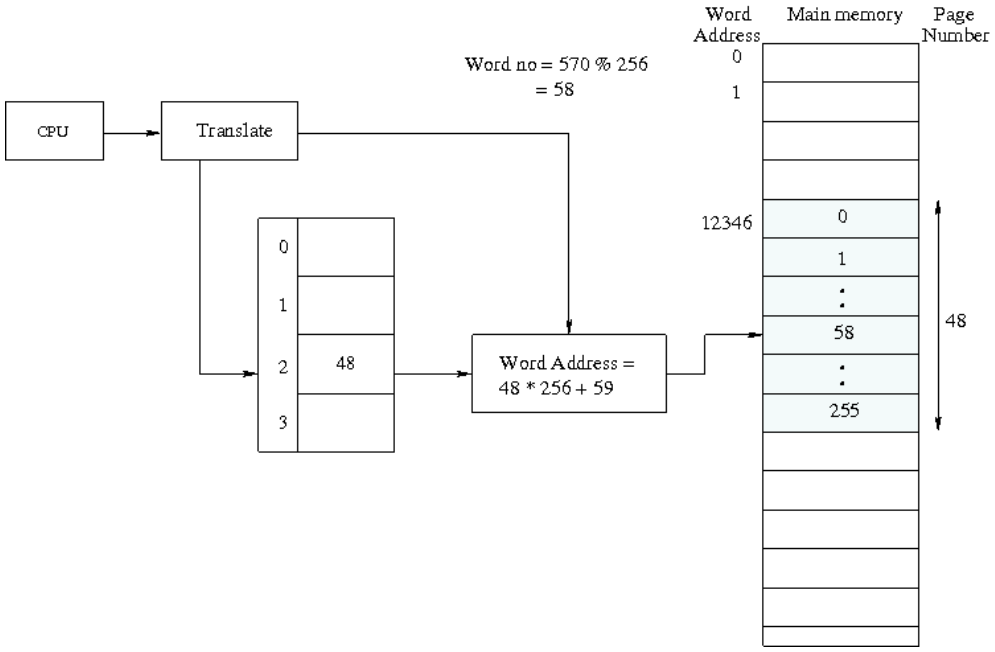



**Fig. 4.4:** Diagram illustrating address translation

**Example 4.3.1.** *Consider the scenario in figure 4.4. 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* $48 \times 256 + 58$.

## 4.4 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) $\times$ 1 (= size of one entry) = 64 words).
- It is present in the second 64 words of page 2 of the memory. Refer figure 4.1.
- 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.

| Pg no. | Contents |
|---|---|
| 0 | 1 |
| 1 | 1 |
| 2 | 0 |
| ⋮ | ⋮ |
| 48 | 0 |
| ⋮ | ⋮ |
| 63 | 1 |

**Fig. 4.5:** A sample free list of the memory

**Example 4.4.1.** *Figure 4.5 indicates that pages 0, 1 and 63 of the memory are not free while pages 2 and 48 are free.*

The entire structure of memory is outlined in figure 4.1.

# Chapter 5

# Process

## 5.1 Introduction

**Def 2.** ***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.

## 5.2 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 5.1 shows the process structure.

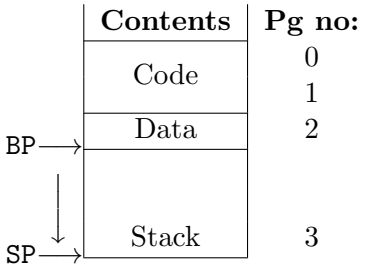



**Fig. 5.1:** Process Structure in memory. Arrow shows the direction of stack growth

## 5.3 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 6.2 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.

## 5.4 Data Structures Associated with a Process

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

### 5.4.1 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.

### 5.4.2 Process Control Block (PCB)

It contains data pertaining to the current state of the process. Refer figure 5.2.

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

**Fig. 5.2:** Structure of Process Control Block

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

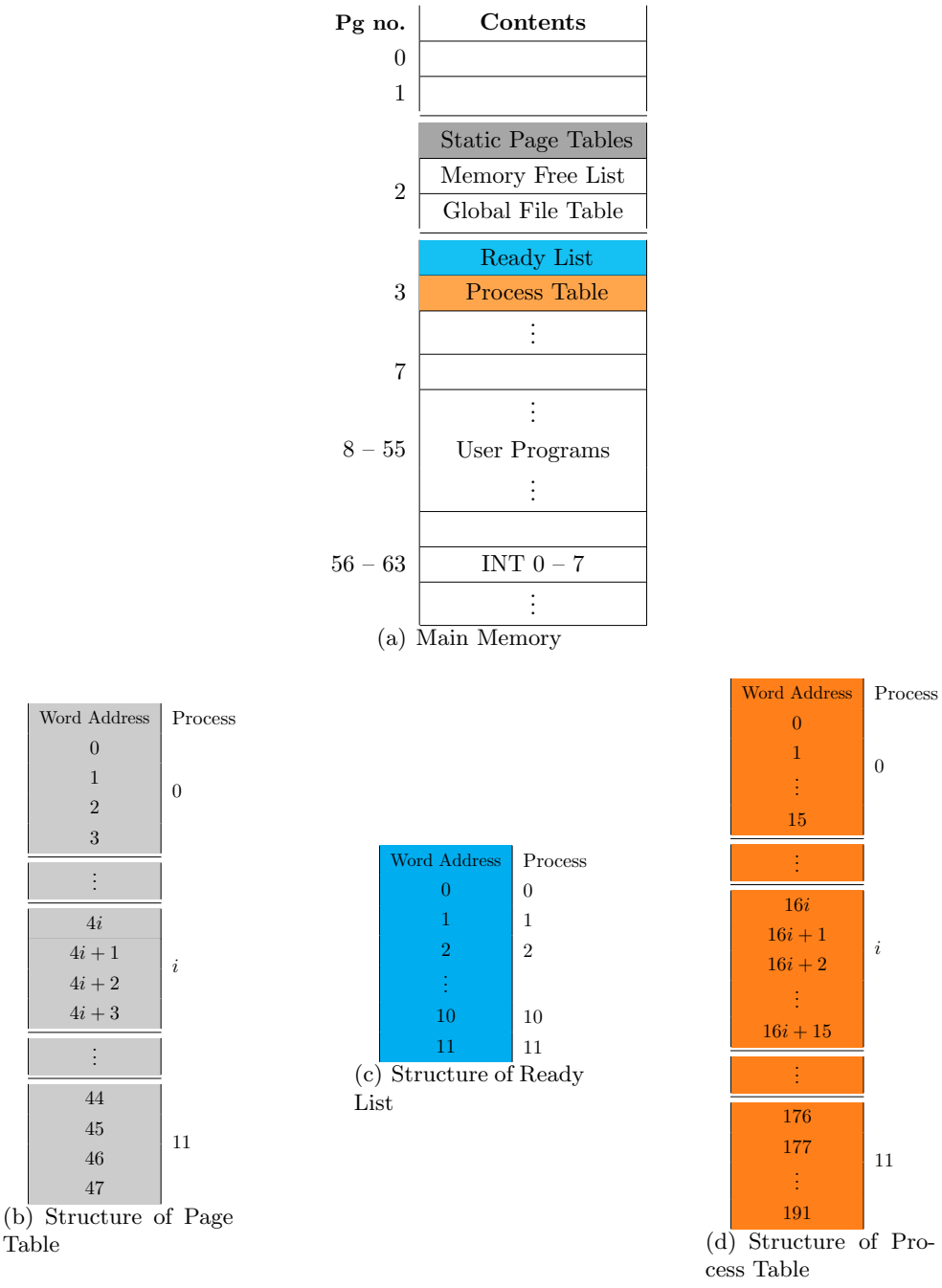
### 5.4.3 The Page Table

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 4.3.

## 5.5 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.




(a) Main Memory

(b) Structure of Page Table

(c) Structure of Ready List

(d) Structure of Process Table

**Fig. 5.3:** Data Structures associated with a process

### 5.5.1 Ready List

- The ready list is located in words 209–220 of page 2 of the memory (refer fig 4.1).
- 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).
- All active processes have an entry 1 in the ready list corresponding to the location indexed by their respective PIDs.

### 5.5.2 Page Tables

- The page tables of the 12 processes are stored in the first 48 words of page 2 of the memory. Refer figure 4.1.
- 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 5.3.

### 5.5.3 Process Table

- The page 3 of the memory contains the process table. Refer figure 4.1.
- 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 5.3.

# Chapter 6

# Instructions

## 6.1 Introduction

All instructions in the SIM architecture are present in the ESIM architecture as well. The additional instructions provided by the ESIM architecture can be classified into *privileged* and *unprivileged* instructions (Refer to the SIM manual for the instruction set and addressing modes).

## 6.2 Processor Modes

The ESIM architecture is interrupt driven and uses a single processor. There are two modes of operation, the user mode and the kernel mode.

- **User mode :** All unprivileged instructions can be executed in this mode.
- **Kernel mode :** Both privileged and unprivileged instructions can be executed in this mode. Initially, the machine starts in kernel mode.

The processor comes to know about the mode in which the system is running by looking at the value in the IP register.

## 6.3 Classification

### 6.3.1 Unprivileged Instructions

All the instructions in the SIM architecture except the HALT instruction constitute the unprivileged instructions. In addition to that, we have five more instructions in the ESIM architecture, one interrupt service instruction and four instructions for string operations. They are:

1. `INT`
   Syntax : INT *no*
   This instruction generates an interrupt to the kernel with *no* as a parameter. It pushes the current IP+1 value into the stack and switches the machine from *User mode* to *Kernel mode*. The address of the first instruction of the specified ISR is stored into the IP register. Execution is started at the address specified IP. Refer section 7.2 to know more about interrupts.
2. `SIN Rn` - This instruction is used to take strings as input. The input string is stored in the data section of the program at the logical address specified by the value in Rn.
3. `SOUT Rn` - This instruction prints the string stored at the logical address specified by the value in Rn. Figure 6.1 illustrates this instruction.

| | |
|---|---|
| 0 | MOV R1 512 |
| 1 | SOUT R1 |
| 2 | MOV R1 513 |
| 3 | SIN R1 |
| 4 | SOUT R1 |
| ⋮ | ⋮ |
| 255 | ... |
| 256 | ... |
| ⋮ | ⋮ |
| 511 | ... |

(a) Code Section

| | |
|---|---|
| 512 | HELLO\0 |
| 513 | ... |
| ⋮ | ⋮ |
| 767 | ... |

(b) Data Section: initial contents

| | |
|---|---|
| 512 | HELLO\0 |
| 513 | ESIM\0 |
| ⋮ | ⋮ |
| 767 | ... |

(c) Data Section: after execution of SIN instruction

**Output**

HELLO
512
ESIM

(d)

**Fig. 6.1:** Example for SOUT instruction

4. STRCPY Ri Rj - This instruction copies the string stored in the data section at the logical address specified by the register Rj to the logical address specified by the register Ri.
5. STRCMP Ri Rj - This instruction compares the strings stored in the data section at the logical addresses specified by the registers Ri and Rj and returns a value 0 if the strings are equal and -1 otherwise. The returned value is stored in Ri.

### 6.3.2 Privileged Instructions

There are *four* privileged instructions. These instructions can be executed only in kernel mode. They are:

- IRET
  Syntax : IRET
  IRET tells the processor that the interrupt handler has finished. This instruction pops the return address of the process from the stack into the IP register and switches the machine from kernel mode to user mode. Refer section 7.2 to know more about the IRET instruction.
- LOAD [1]
  Syntax : LOAD *pg_no block_no*
  This instruction loads the block specified by the *block_no*, from the disk, to the page specified by the *pg_no*, in the memory.
- STORE[1]
  Syntax : STORE *block_no pg_no*
  This instruction stores the page specified by the *pg_no*, from the memory, to the block specified by the *block_no*, in the disk.

[1]These are macros which initialise the DMA controller with the arguments passed and invoke it for the actual transfer to take place.

- `HALT`
  Syntax : HALT
  This instruction causes the simulator to halt immediately.

# Chapter 7

# Interrupts

## 7.1 Introduction

Interrupts are mechanisms by which the user code interrupts the execution of the processor and passes control to the kernel to accomplish low level functionalities like disk access, arithmetic exception handling etc.

**Interrupt Service Routine(ISR)** : The kernel provides routines to accomplish the functionality for which an interrupt has been generated. These routines are known as Interrupt Service Routines.

**Note:** Every ISR should end with an IRET instruction.

## 7.2 The `INT` instruction

The instruction used to generate an 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.

Address computation is done as follows. The physical address of the ISR corresponding to interrupt number $n$ is given by:

$$\text{Physical Address} = (56 + n) \times \text{Page Size}$$

Figure 7.1 summarises the physical address to which the control is transferred for each interrupt. Note that the interrupts are disabled once this instruction is executed, since we do not allow interrupts to occur in kernel mode.

## 7.3 Types of Interrupts

There are 8 interrupts (numbered from 0 to 7) supported by the `ESIM` architecture. The interrupts 0 is a hardware interrupts and the remaining interrupts (1 to 7) are software interrupts.

Details of the *hardware interrupt* is as follows.

- `INT 0 :` This is the timer interrupt which interrupts the processor forcing a context switch. It contains the code for the scheduler of the operating system (refer section 15.1), which schedules the CPU time among the various active processes. Note that this interrupt is machine generated and cannot be called.

Details of *software interrupts* are as follows.

- `INT 1--4:` These interrupts are used for the various file system calls. (Refer section 14.4 for File System Calls)

| Interrupt No. | Word Address | | |
|---|---|---|---|
| | Page No. | Offset | Address |
| 0 | 56 | 0 | $56 \times 256 + 0$=14336 |
| 1 | 57 | 0 | $57 \times 256 + 0$=14592 |
| 2 | 58 | 0 | $58 \times 256 + 0$=14848 |
| 3 | 59 | 0 | $59 \times 256 + 0$=15104 |
| 4 | 60 | 0 | $60 \times 256 + 0$=15360 |
| 5 | 61 | 0 | $61 \times 256 + 0$=15616 |
| 6 | 62 | 0 | $62 \times 256 + 0$=15872 |
| 7 | 63 | 0 | $63 \times 256 + 0$=16128 |

**Fig. 7.1:** Interrupts and their locations in the 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 | ⋮<br>User Programs<br>⋮ | 2816 – 14335 |
| 56 | INT 0 | 14336 – 14591 |
| 57 | INT 1 | 14592 – 14848 |
| ⋮ | ⋮ | ⋮ |
| 63 | INT 7 | 16128 – 16383 |

**Fig. 7.2:** Outline of the main memory

- `INT 5--7:` These interrupts are used for the various process system calls. (Refer section 16.1 for Process System Calls)

The interrupts 1–7 are unprivileged and can be called from user mode.

## 7.4 Calling Convention

In this section, we explain the calling and returning conventions for interrupts.[1]

### 7.4.1 Calling Convention

Before switching the control to the ISR using the INT instruction the user program does the following:

1. Push a dummy value for storing the return value of the interrupt onto the stack.
2. Push the arguments to the interrupt.
3. Make the Interrupt call using the INT instruction.

The INT instruction pushes the IP+1 value on to the stack and then starts the execution of the corresponding ISR. When the ISR finishes its execution, IRET instruction is called. This IRET instruction pops the IP+1 value from the stack top into the IP register and the execution of the user program is resumed from the point where it was interrupted.

### 7.4.2 Returning Convention

After returning from the ISR using the IRET instruction the user program does the following:

1. Pop out the arguments from the stack.
2. Pop out the return value.

Figure 7.3 explains the state of the stack at various stages.

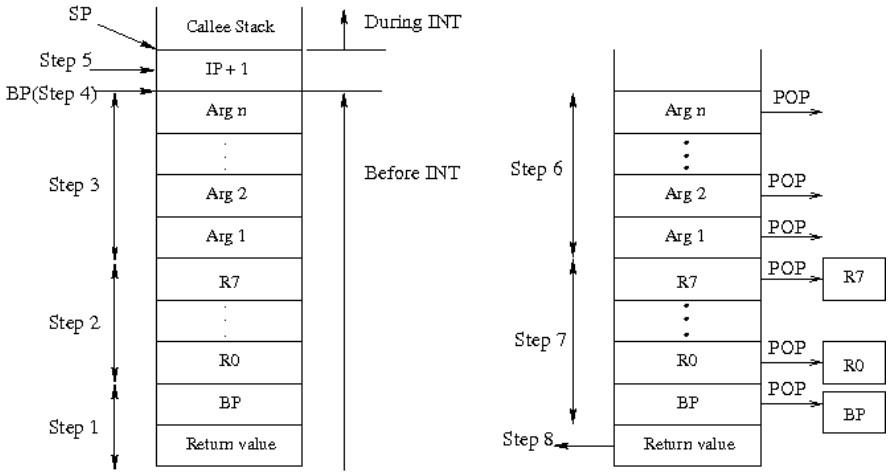



**Fig. 7.3:** Recommended calling and returning convention for interrupts

[1]The convention given above is already built into the AP-SIL compiler. It has been given only to help you understand the internal workings better.

# Part II

# Machine Implementation

# Chapter 8

# Machine Implementation

## 8.1 Machine

The implementation details for the machine are given below. details given include the header file, the corresponding code file, the various functions included in them and a short description of these functions.

1. *data.h*

   Consists of constants declared for the machine. These include the registers, and the size of various constiuents of memory. The entire memory is declared here as well.

2. *memoryConstants.h*

   Declarations for the structure of main memory is made here.

3. *instr.h*

   Declares the constants associated with each token.

4. *decode.lex*

   This is the lexical analyser which analyses each instruction and reurns the corresponding token.

5. *boot.h* and *boot.c*

   This file consists of the following functions:

   - void loadStartupCode() - Loads the OS Startup Code from disk to the proper location in memory
   - void initializeRegs() - Initializes the values of all the registers to zero.

6. *scheduler.h* and *scheduler.c*

   This file consists of the following functions:

   - void runInt0Code() - Causes the timer interupt leading to the control being passed on to the INT 0 code in memory.

7. *timer.h*

   This file contains the constant defining the number of clock cycles that makes up a timslice alloted to a single process. This file also consists of the following functions:

   - int isTimeZero() - Checks whether the timer counter reads zero.
   - void tick() - Decrements the timer counter.
   - void resetTimer() - Resets the timer counter.

8. *utility.h* and *utility.c*

   This file consists of the following functions:

- void emptyPage(int) - Clears the page speecified by the aguement.
- struct address translate(int) - Translates the virtual address passed as arguement to the corresponding page number and offset.
- printRegisters() - Prints the values of all the registers. Used for debugging purposes.
- void exception(char*) - Acts as the exception handler. Prints the instruction that caused the exception and terminates execution.

9. *simulator.h* and *simulator.c*

   This file consists of the following functions:

   - void Executeoneinstr(int) - This function simulates all the instructions available on the esim architecture.
   - void run(int, int) - This function acts as the bootloader. It loads the Startup code. It also calls Executeoneinstr() for every instruction that it reads.
   - int main(int, char**) - Makes the initial changes to the machine environment and the calls run().

# Part III

# File System Specification

# Chapter 9

# File System

## 9.1 Introduction

**Def 3.** ***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 (256 words).

## 9.2 Disk Structure

The basic structure of the disk is shown in figure 9.1.

| **Block No.** | 0 | 1 | 2 | ... | 8 | 9–10 | 11–12 | 13–16 | 17–511 |
|---|---|---|---|---|---|---|---|---|---|
| **Contents** | OS Startup code | INT 0 | INT 1 | ... | INT 7 | Free List | FAT | INIT | Data Blocks |

**Fig. 9.1:** Structure of the disk

## 9.3 Addressing

**Def 4.** ***Block number :*** *Any particular block in the disk is addressed by the corresponding number in the sequence 0 to 511 known as the* block number*.*

**Example 9.3.1.** *In figure 9.2, the* $2^{nd}$ *block of the disk has a block number 1. In general the* $i^{th}$ *block has the block number* $(i-1)$ *for* $1 \leq n \leq 512$*.*

## 9.4 Disk Free List

- The Free List of the disk consists of 512 entries. Each entry is of size one word.
- The total size of the free list is thus 2 blocks or 512 words (512(= no. of entries) $\times$ 1(= size of one entry) = 512 words).
- It is present in blocks 9 and 10 of the disk. Refer figure 9.1.
- Each entry of the free list contains a value of either 0 or 1 indicating whether the corresponding block in the disk is free or not respectively (It should be ensured that the first 13 entries are always marked used).

**Example 9.4.1.** *Figure 9.3 indicates that the blocks 0, 1 and 511 of the disk are not free while blocks 2 and 48 are free.*

| Block | Contents | Block no. |
|---|---|---|
| 1 | $0^{th}$ word<br>$1^{st}$ word<br>⋮<br>$255^{th}$ word | 0 |
| 2 | $256^{th}$ word<br>$257^{th}$ word<br>⋮<br>$511^{th}$ word | 1 |
| ⋮ | ⋮ | ⋮ |
| 512 | ⋮<br>⋮<br>$(256 \times 512 - 1)^{th}$ word | 511 |

**Fig. 9.2:** Disk addressing

| Index | Content |
|---|---|
| 0 | 1 |
| 1 | 1 |
| 2 | 0 |
| ⋮ | ⋮ |
| 48 | 0 |
| ⋮ | ⋮ |
| 511 | 1 |

**Fig. 9.3:** A sample free list of the disk

## 9.5 File

A file is a collection of data 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 of a file.
  - The basic block structure is shown in figure 9.4.

| **Index** | 0–127 | 128–255 |
|---|---|---|
| **Content** | Block List | Header |

**Fig. 9.4:** Structure of the basic block of a file

  - The basic block consists of the *Block List* and the *Header*.
  - **Block List :** It is similar to an index in a book which tells which chapter starts from which page.
    - The block list consists of 128 entries.
    - Each entry is of size one word.
    - The size of the block list is thus 128 words (128(= no. of entries) x 1(= size of an entry) = 128 words).
    - The value contained in an entry of the block list gives the block number of the corresponding data block in the disk.
  - **Header :** The header contains the header information relating to the file. Currently this is unused, but at a later stage can be used to store information such as file modification date/time, author of the file etc.

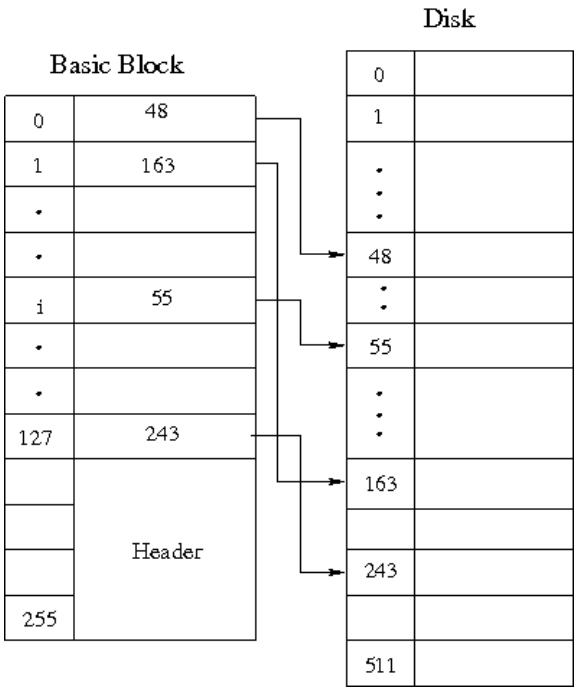



**Fig. 9.5:** Example illustrating the basic block of a file

**Example 9.5.1.** *Consider the example illustrated by figure 9.5. From the figure, we infer the following.*

- *The zeroth data block of the file resides at the disk block whose block number is 48.*
- *The first data block of the file resides at the disk block whose block number is 163.*
- *The ith data block of the file resides at the disk block whose block number is 55 where* $0 \leq i \leq 127$.
- *The 127th data block of the file resides at the disk block whose block number is 243.*

### 9.5.1 File Types

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

1. **Data files :** These files contain data or information that is used by the programs. They can occupy a maximum of 129 blocks (1 basic block + 0 - 128 data blocks).
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.

### 9.5.2 Executable File Format

Any executable file has the following format. Refer figure 9.6.

- It consists of the *Code section* and the *Data section*.
- **Code section :** This section contains the actual code to be run on the machine. It spans 2 blocks irrespective of the size of the code.
- **Data Section :** This section consists of data that is used in the code which cannot be stored in a register. The registers then store the logical address of the corresponding data residing in the data section. It spans 1 block.

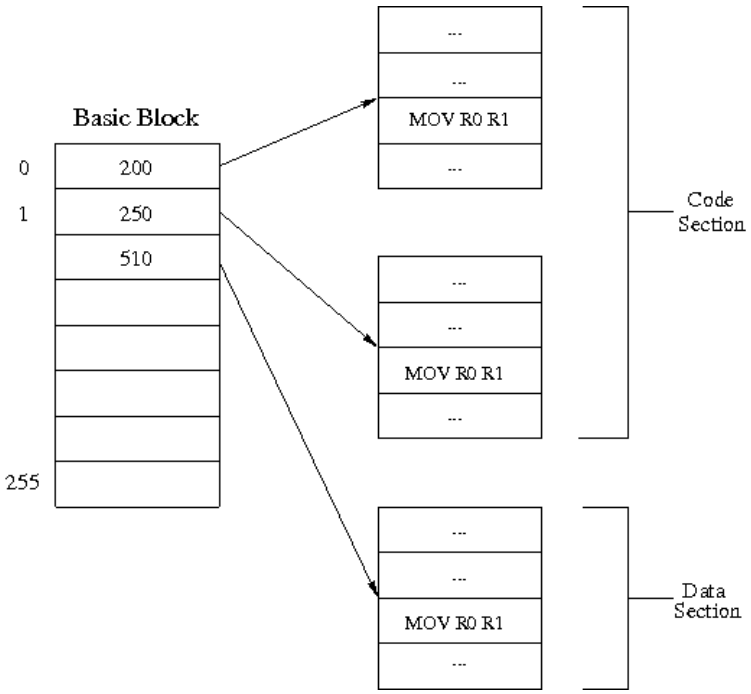



**Fig. 9.6:** Example illustrating the structure of an executable in the disk

## 9.6 File Allocation Table (FAT)

*File allocation table* (FAT), as the name suggests, is a table that has an entry for each file present in the disk.

- FAT of the filesystem consists of 32 entries. Thus there can be a maximum of 32 files.
- Each entry is of size 16 words.
- Total size of the FAT is thus 512 words (32 (= number of entries) $\times$ 16(= size of one entry) = 512 words).
- It is a disk data structure and occupies block numbers 11 and 12 of the disk. Refer figure 9.1.

The structure of a FAT entry is shown in figure 9.7.

| 0 | 1 | 2 | 3 – 15 |
|---|---|---|---|
| File Name | File Size | Block no: of basic block | ... Free ... |

**Fig. 9.7:** 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.txt`, `calc.sim`.
2. **File size :** It indicates the number of words occupied by a file. It varies from 0 words to (128 $\times$ 256) words (depending upon the number of data blocks it has). 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.

# Part IV

# File System Implementation

# Chapter 10

# File System Implementation

## 10.1 File System

The implementation details for the file system are given below. details given include the header file, the corresponding code file, the various functions included in them and a short description of these functions.

1. *createDisk.h* and *createDisk.c*

   This file consists of the following functions.

   - void createDisk(int) : Creates a disk file if it does not exist. If it does the function also has the option of formatting the disk.

2. *fileSystem.h* and *fileSystem.c*

   The header file consists of all the constants that have been defined for implementing the filesystem. This file consists of the following functions.

   - void listAllFiles() - Lists all files in the FileSystem.
   - int deleteExecutableFromDisk(char*) - Deletes anexecutable file from the filesystem
   - int removeFatEntry(int) - Removes the fat entry for a file.
   - int getDataBlocks(int*, int) - Retrieves the datablocks for a file which already exists on the filesystem.
   - int loadExecutableToDisk(char*) - Loads executable file t disk.
   - int CheckRepeatedName(char*) - Checks whether a file already exists on the filesystem. If it does it returns the fat entry for that file.
   - int FindFreeBlock() - Allocates and returns an empty block in the filesystem.
   - int FindEmptyFatEntry() - Searches and returns an empty fat entry in the filesystem.
   - void FreeUnusedBlock(int*, int) - Frees the blocks which are allocated on the disk. These are passed as the first arguement.
   - void AddEntryToMemFat(int, char*, int, int) - Popuates the various fields of FAT on the disk.
   - int writeFileToDisk(FILE*, int) - Commits the changes made to the memory copy of the file to the underlying filesystem.
   - int loadOSCode(char*) - loads the Startup code onto the filesystem.
   - int loadIntCode(char*, int) - loads the interrupt code code to the proper place on the filesystem depending on the arguement..
   - int initializeInit() - Makes a dummy entry for init on the filesystem.
   - int loadInitCode(char*) - loads init code onto the filesystem.

3. *fileUtility.h* and *fileUtility.c*

   This file consists of the following functions:

   - emptyBlock(int) - Empties the memory copy of the disk
   - int getInteger(char*) - Converts the arguement from char* to int and returns the int version.
   - void storeInteger(char*, int) - Converts the second arguement to integer and stores it in the location specified by the first arguement.
   - int readFromDisk(int, int) - Reads an entire page from the block number specified by the second arguement to the memory location specified by the first arguement.
   - int writeToDisk(int, int) - Writes an entire page to the block number specified by the second arguement from the memory location specified by the first arguement.
   - int loadFileToVirtualDisk() - Creates a memory copy of the disk.
   - void clearVirtDisk() - Clears the entire memory copy of the disk.

4. *interface.h* and *interface.c*

   This file consists of the following functions:

   - void menu() - Displays the menu available for the filesystem.
   - int main() - Displays the menu and does the various functions as the user requires.

# Part V

# Operating System Specification

# Chapter 11

# Introduction

The OS provides an interface to the user to interact with the hardware. Users write programs that make use of various resources like disk, memory, processor etc. These programs are run as processes on the machine.

The system programmers use the language SP-SIL for writing the Operating System. User programs to test the various functionalities of the Operating System can be written in AP-SIL. Refer the documents [MKS12b] and [MKS12a] for the complete documentation of these tools.

A detailed operating system specification was done in the works of [GDKI11] and [KAG+11]. This documentation was critically reviewed and modifications were done in many places to comply with our new design. The major modifications done are the following:

- Introduction of INIT process (refer section 12.3)
- Introduction of Halt() system call (refer chapter 13)
- Exception handler interrupt has been excluded due to some limitations in the design.
- The distribution of system calls were changed. This was due to the limitations in interrupt code size.

## 11.1 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.

### 11.1.1 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,

- 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.

### 11.1.2 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 15 to know more about multiprogramming.

### 11.1.3 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 14 and chapter 16 to know more about system calls.

# Chapter 12

# OS Startup

## 12.1 ROM Code

It is a hard coded assembly level code present in page 0 of the memory. Refer figure 4.1. 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.

## 12.2 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 4.1) 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.

[1] This can be achieved by calling IRET.

## 12.3 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.

# Chapter 13

# Halt System Call

## 13.1 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 7.4 for calling convention).

## 13.2 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 14

# File System Calls

## 14.1 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 4.1) 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.

## 14.2 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 4.1 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 14.1.

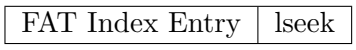



**Fig. 14.1:** Structure of a GFT entry

  - **FAT index entry :** It is used to index the memory copy of the file allocation table(section 9.6) 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 5.4.2), 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).

## 14.3 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.

## 14.4 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:

### 14.4.1 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.

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

- 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.

### 14.4.2 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.

### 14.4.3 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.

[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.

- 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 14.2.
   - 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.

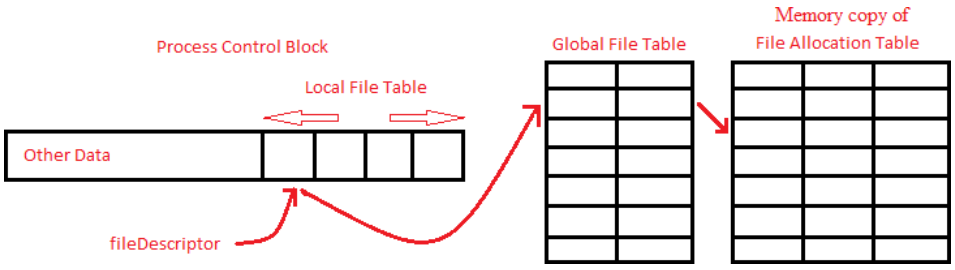



**Fig. 14.2:** Diagram showing the method of accessing FAT entry

### 14.4.4 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

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

[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.

- 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 14.2.
- 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]
- 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 15

# 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].

## 15.1 Scheduler

Whenever a timer interrupt occurs, the kernel temporarily halts the execution of the currently executing process and invokes INT 0. Refer book [Cro96] 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 16

# Process System Calls

## 16.1 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:

### 16.1.1 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.

### 16.1.2 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.

### 16.1.3 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 5.3).
- 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*.

[1] Executables in ESIM 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.

[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.

## 16.2 INIT Process

The INIT process is the first user process loaded by the OS on the OS startup. INIT was previously defined in chapter 12 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.

# Chapter 17

# Future Work

In the project so far we have documented the machine and the operating system. However there is no step-by-step instruction for the student showing him the way he has to proceed for designing the operating system. This roadmap is intended to do the same.

The current machine supports both integer and string data type. However the registers can hold only integers. these have led to various problems while implementing the operating system. One such problem is the non availability of using predefined strings in kernel code. Converting the current machine to a pure *String* machine will solve this problem.

The project was developed as a means for replacing NACHOS. To reach out to a wider audience it would be better to host this on a public domain which would be accessible to teachers and students.

Since the student is working directly on the machine and developing the operating system from scratch, it is advisable to have a more advanced debugging interface.

Finally the deployment of the finished tool in Operating system Lab.

# Chapter 18

# Conclusion

The entire project was started as a way for increasing the knowledge gained by a student taking Operating System lab. The current lab utilizes NACHOS [CPA93] and students gain little knowledge from it. Moreover the clerical overhead involved in implementing an Operating System on NACHOS is way more than the knowledge gained from it.

The current project was done taking in mind the difficulties that were faced while using NACHOS. It was designed in a way so as to minimize the clerical overhead and give the student a feel of how an Operating System functions. How successful we have been, only time can tell. However from our experience we feel that this new project is above NACHOS in terms of conceptual knowledge gained and ease of use.

There have been some decisions which deviate from how an Operating System works. This includes fixing the location of INIT process and Exec not checking whether the new file to load is an executable or not. These problems arose because the Kernel codes do not have a stack. One solution was to do what we did and another one was to add a kernel stack. The latter would have involved changes in design and so we did not choose it. We expect this to be solved in the next version of the project where the entire machine would have only string data type.

Writing an Operating System is not so tedious on this machine as it was in NACHOS. We had to write an Operating System for testing and debugging. From that experience, we can clearly say that the student in fact learns not only a lot about Operating System but also many new other concepts. One such concept is the use of memory de-referencing.

On an overall note, the project has ended on a high. Though it has its faults, it is better than the tool currently being utilized. The next version, if it goes as planned, will take it a level higher. Nevertheless we feel that what we have done is ready to be put forth before the student community to aide them in learning Operating Systems.

# Bibliography


[Bac86] Maurice J. Bach. *The design of the UNIX operating system.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.

[CPA93] Wayne A. Christopher, Steven J. Procter, and Thomas E. Anderson. The nachos instructional operating system. In *In Proceedings of the Winter 1993 USENIX Conference*, pages 479–488, 1993.

[Cro96] Charles Crowley. *Operating Systems: A Design-Oriented Approach.* McGraw-Hill Professional, 1996.

[GDKI11] Jeril K George, K Dinesh, Mathew Kumpalamthanam, and Naseem Iqbal. Design and implementation of an experimental operating system : File system specification. B.Tech thesis, NIT Calicut, May 2011.

[KAG+11] Ajeet Kumar, Avinash, Deepak Goyal, Nitish Kumar, Sathyam Doraswamy, and Yogesh Mishra. Design and implementation of an experimental operating system : Architectural specification. B.Tech thesis, NIT Calicut, May 2011.

[MKS12a] Shamil C M, Vivek Anand T Kallampally, and Sreeraj S. Ap-sil language specification. B.Tech thesis, NIT Calicut, February 2012.

[MKS12b] Shamil C M, Vivek Anand T Kallampally, and Sreeraj S. Sp-sil language specification. B.Tech thesis, NIT Calicut, February 2012.

[SGG05] Abraham Silberschatz, Greg Gagne, and Peter Baer Galvin. *Operating System Concepts, Seventh Edition.* John Wiley & Sons, Inc, 7 edition, 2005.