Operating Systems

Todd Davies

December 26, 2014

Introduction

Operating systems provide an interface for computer users that permits them to gain access without needing to understand how the computer works. The software needed to achieve this is complex and this course introduces students to some of the details of design and implementation.

Aims

This course unit introduces students to the principles of operating system design and to the prevailing techniques for their implementation. The course unit assumes that students are already familiar with the structure of a user-program after it has been converted into an executable form, and that they have a rudimentary understanding of the performance trade-offs inherent in the choice of algorithms and data structures. Pertinent features of the hardware-software interface are described, and emphasis is placed on the concurrent nature of operating system activities. Two concrete examples of operating systems are used to illustrate how principles and techniques are deployed in practice.

Additional reading

Operating system concepts (8th edition)	Silberschatz, Abraham and Peter Baer Galvin and Greg Gagne	2009
Operating system concepts with Java (8th edition)	Silberschatz, Abraham and Peter Baer Galvin and Greg Gagne	2010

Contents

1 Recap of COMP12111 & COMP15111

There material in both the *Fundamentals of Computer Architecture* and the *Fundamentals of Computer Engineering* courses in the first year provides a good base for the course this year. The following re-visits that material and builds upon it.

1.1 Datapath and Control

From the point of view of the CPU all data is of a fixed size, the length of one word. Each word is usually moved around the architecture of the computer in a bit-parallel manner, that is to say that there are at least the same number of wires in a bus between any two components in the system as there are bits in the word.

The individual operations on each bit inside a word when the CPU performs an operation on the whole word are usually identical. This results in a very regular datapath with lots of duplicated (usually by as many times as the word length) hardware logic.

Control logic is derived after the datapath has been conceived. It governs which operation is performed at what time, and is different for each instruction in the instruction set.

A typical example of control logic might be to control the enable pin on a binary adder. The datapath will direct bits to the adder all the time, but the control logic will determine if the result is sent forward.

1.2 The MU0 Instruction Set Architecture

The MU0 is a very simple 16 bit word architecture, and as a result, the instruction set is also very simple. Each instruction can address one memory location, and consists of four bits for the instruction (allowing sixteen instructions to be coded) and twelve bits for the memory address as illustrated in Figure 1. Since we can only store twelve bits of memory address in the instruction, and the architecture is very simple, the system has 2^{12} words of memory, which is equivalent to 8 kB.

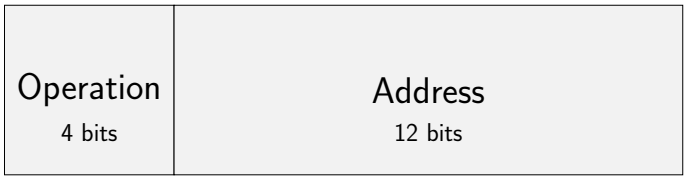

Figure 1: A generic MU0 instruction

The MU0 has two programmer visible registers, the Program Counter and the Accumulator. The Program Counter stores the address in memory of the next instruction to be executed, thus being twelve bits long. The Accumulator is sixteen bits long, and stores the result of the last arithmetic operation.

The instructions that the MU0 understands are listed in Table 1.

1.3 Maintaining Processor State

If the execution cycle of the MU0 was somehow disrupted, say because of an interrupt call, it would be handy to save the state of the processor before switching to a different task (e.g. running the interrupt handler).

The way to do this is to save the registers in memory, doing the other task, and then reloading them when it's time to resume execution of the program.

Op Code	Mnemonic	Description
0	LDA $[op]$	$[op] \rightarrow Acc$
1	STO $[op]$	$Acc \rightarrow [op]$
2	ADD $[op]$	$Acc = Acc + [op]$
3	SUB $[op]$	$Acc = Acc - [op]$
4	JMP $[op]$	$PC = S$
5	JGE $[op]$	If $Acc >= 0$ then $PC = S$
6	JNE $[op]$	If $Acc \neq 0$ then $PC = S$
7	STP	Stop

Table 1: The MU0 instruction set

1.4 The Fetch Execute Cycle

The fetch-execute cycle describes how a CPU executes instructions. First, the next instruction is fetched from memory (at the address pointed to by the PC), then the instruction is executed. Since some instructions access memory (such as load and store), and we can only do one memory access per clock cycle, one fetch-execute cycle takes two clock cycles, one for fetching, and one for execution.

1.4.1 Fetching instructions

Fetching is an operation that is the same for all instructions. First memory addressed by the PC is read and stored into the Instruction Register (IR). This is a 16 bit internal register that isn't visible to programmers. Once this has occurred, the PC is incremented. This means that the RAM must be able to send a word directly to the instruction register, so a datapath must be in place to allow this.

1.4.2 Executing instructions

It is obvious that different instructions will have different paths of execution within the processor, and will have different effects on components within the system.

`JMP` In order to execute the `JMP` instruction, the last twelve bits are read from the instruction register and transferred over to the PC. This means that there must be a datapath from the bottom twelve bits of the IR to the PC.

`STA` When `STA` is executed, the bottom twelve bits in the IR are used direct the contents of the accumulator to a location in memory. To do this, we need a datapath from the bottom twelve bits of the IR to the part of the RAM that takes addresses, and from the PC to the part of the RAM that takes data.

`ADD` To perform the `ADD` instruction, we need to fetch the bottom twelve bits of the IR and send it to the RAM. The result should be fed into the adder along with the contents of the accumulator. The result of the calculation should be sent to the accumulator. To do this, we need datapaths from the accumulator to the ALU, the RAM to the ALU and finally from the ALU to the accumulator.

Control Signals whenever two separate components within the system interact. For example, every time the CPU loads a word from the RAM, a control signal must be sent to say 'load', and every time the `ADD` command it executed, the ALU must be sent a control signal to say 'add' as opposed to subtract or shift.

Timing Timing is very important when executing the instructions. If the result of a load from RAM hasn't yet returned, but the control signal to the ALU to add is sent, then the wrong result will almost certainly occur! In order for everything to run smoothly, the critical path for each operation must be worked out, and time allowed for signals to propagate through even the longest critical path.

1.4.3 Deriving the datapaths from the operation of instruction

In order to produce a working processor, we need to look at all the instructions that can be executed by the processor, and examine what datapaths and control signals they require to work. Only when we have this information can we begin to actually design the hardware on the CPU.

Note that data going to one destination can only go to one source, so if you want multiple components to be able to send data to one other component, then you must use a multiplexer with control signals in order to achieve this.

1.5 Control Signals

The purpose of control signals is to make each component within the CPU function as intended for each specific instruction. Control signals include:

- Enable write for registers
- Enable write for memory
- Enable read for memory
- Multiplexer input select
- ALU actions (add, subtract, bypass)

Sometimes, one component (such as the ALU) may have control inputs that can be represented by more than two states (add, bypass, subtract). If this is the case, then multiple wires (a bus) is used to specify its action.

The first lab in the course shows the control signals sent for each instruction, the solution for which is shown in Tables 2 and 3.

En_IR	1	(enable write to IR)
En_PC	0	(enable write to PC)
En_ACC	X	(enable write to ACC)
byp	X	(ALU action: bypass)
add	X	(ALU action: add)
sub	X	(ALU action: subtract)
Ren	1	(RAM action: Read)
Wen	0	(RAM action: Write)
addr_Mux	1	(RAM address = PC, otherwise IR.S)

Table 2: The control signals in the MU0 fetch phase

2 What does an Operating System do?

The job of an operating system varies from system to system but on general, it is responsible for managing the resources of the system (including dealing with concurrency, security etc) and abstracting the implementation of the system from the running programs (such as what exact components are being utilised).

	lda	sta	add	sub	stp	jump	no jump
En_IR	0	0	0	0	0	0	0
En_PC	0	0	0	0	0	1	0
En_ACC	1	0	1	1	0	0	0
byp	1	X	0	0	X	X	X
add	0	X	1	0	X	X	X
sub	0	X	0	1	X	X	X
Ren	1	0	1	1	0	1	0
Wen	0	1	0	0	0	0	0
addr_Mux	0	0	0	0	X	0	1

Table 3: The control signals in the MU0 execute phase

2.1 Processes

A process is a program that is currently running on the system. It consists of a Thread (a set of instructions to be executed) and address space (a set of memory locations that can be accessed by the thread). In most systems, multi-threading is used to allow each process own multiple threads, and therefore execute in parallel.

Nearly all systems have many processes running at any one time, on a Linux system, use `htop` or `ps aux` to see what processes are running.

2.2 Address Space

Address space (aka memory space) is a term used to speak about a section of memory. This could be the whole memory available to the system, the memory that a specific program has access to etc.

When a program starts, it assumes that it does so from memory address `0`. On a single process system this is okay, however this presents a problem on systems where multiple processes run concurrently, since no two processes can share the same memory space.

Sometimes, operating systems may even running pause programs, move them out of memory (onto secondary memory such as hard drive) and later on swap it back in at a different place in memory.

In both cases, a technique called *Relocation* is used to make every running program able to safely assume that it has sole use of memory.

In order to facilitate relocation, operating systems abstract away the implementation of the hardware, and instead provide a virtual machine for each program. This enables programs to behave as though they have the whole system to themselves, and it also lets the operating system easily stop programs interfering with each other (such as providing disjoint memory spaces for each program).

2.3 Modes of operation

It is often necessary to prevent some programs from executing some operations, such as manipulating memory, or allocating CPU time. In order to achieve this, operating systems nearly always implement different 'modes' of operation that processes can run under. The two most common modes are *user* and *system*.

All the processes owned by the operating system will run under system mode, which is very permissive and lets programs perform operations with the potential for misuse. Programs that the user might run are usually executing under user mode. User mode is less permissive, and restricts certain operations, yet the restricted operations aren't usually required for normal programs.

2.3.1 System calls

If there was an eventuality where a program needed to perform privileged operation that wasn't permitted under it's current mode, then it can use a system call to achieve the same result. The premise is that the operating system will provide a 'gatekeeper' function that will perform the requested operation, but only after the parameters have undergone checks to ensure that the application isn't behaving badly. The execution of the user program will (of course) pause while the system call is running.

Lots of functions in languages that you already know might just be wrappers around system calls, albeit often with slightly more functionality. The course notes make a good example; `fread` in C uses the `UNIX` system call `read`.

3 Engineering an Operating System

Like a lot of things in Computer Science, operating systems can often be conceptualised as being built up of several layers. As you inspect further and further into the OS, looking deeper into each layer as you go, the level of abstraction decreases.

The outermost layer could be seen to be the UI, which is obviously a very abstracted way of thinking about a computer. The kernel is probably the lowest layer, since the details of how the hardware is managed is, also fairly obviously, a very low level of abstraction.

Different operating systems can contain a different number of layers of abstraction so that they are best suited to their purpose. Some operating systems will contain little more than a microkernel, which will have the bare minimum of logic required to keep a computer running.

Some components of an operating system are monolithic, most notably the Linux kernel. A monolithic component is easier to design (especially for a kernel) since there is less inter-process communication to worry about, and trap falls like race conditions can be more easily avoided when everything is running on a single thread.

See this StackOverflow answer for an explanation (and follow the link to the Linus V Tanenbaum showdown on the topic): `http://stackoverflow.com/questions/1806585/why-is-linux-called-a-monolithic-kernel`

I feel obliged to point out that monolithic, in this instance, refers to the fact that the program is running under *one* process that is in system mode. It doesn't mean that the code is all in one big file, or even in one big project, since many monolithic projects employ some degree of modularity in the development process at least.

3.1 Managing processes

When operating systems were first developed, they usually ran one process at a time. This was obviously limiting for the users, since only one person could do stuff at a time. In light of this, timesharing was developed, which is a way for the computer to be used by multiple users at once, even here, users would all share one CPU and one PC though.

Now, processes execute within their own virtual CPU, while the real CPU switches back and forth giving time to each task. This is good, since the order with which the CPU gives it's time to processes can be used to effectively make some processes run faster than others.

We have already learnt that a process is made up of a thread and some address space, but we've not been over how a process is created. In order to do this, an existing process must create new child process.

(Processes also have register values and external interfaces too, but they're not as important)

Processes can be killed by themselves (with a variety of different exit codes), by other processes (using something like the `kill`) command, or in some systems, by the parent process terminating.

The lifecycle of a process, and the states it occupies can be represented in a state machine like diagram, as shown in Figure 2.

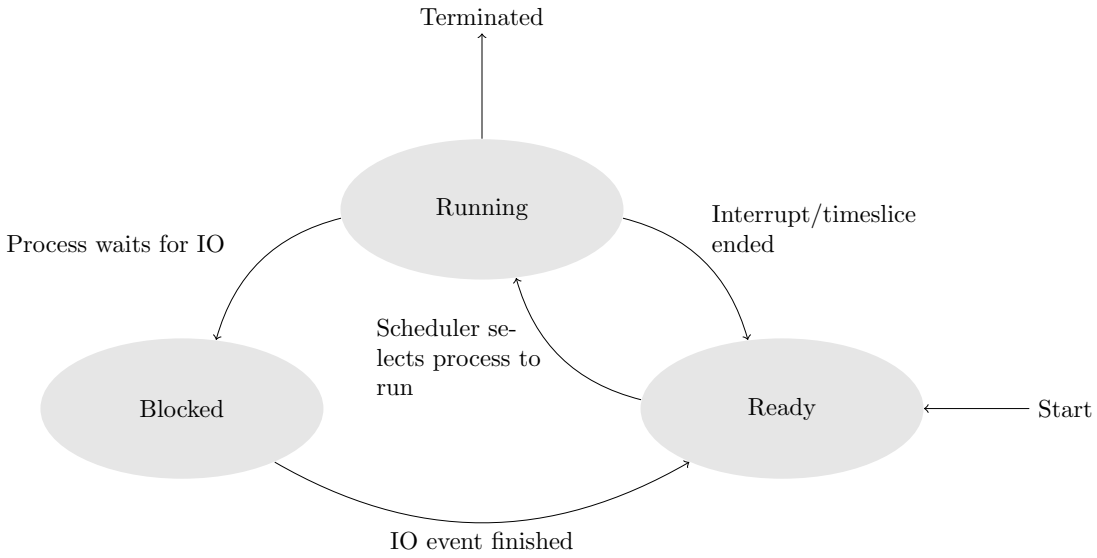

Figure 2: The lifecycle of a process

3.1.1 Scheduling

One particurlaly troublesome section of the lifecycle of a process is going from the *ready* state to the *running* state (and vice versa). In order to do this, a process to be resumed must be chosen from a pool of processes waiting for CPU time, and the registers, IO operations etc must be loaded and made ready.

In order to do this, a PCB (Process Control Block) table is maintained, which contains all the nececarry information for each process to be paused and resumed. This includes (among other things):

- Process ID
- Parent ID
- Saved registers
- Memory, IO management information
- CPU scheduling info

On some operating systems, timeslices are given to processes, where on others, they are assigned to threads. Henceforth, processes can often be made more efficient by using threads in order to maximise the use of their timeslice (switching to non-blocked threads whenever there is a block).

3.1.2 Context switching

A context switch occurs whenever the CPU pauses execution for one process and resumes it for another. It's sometimes and expensive operation, and depending on the processor, can take anywhere from one micro-second to one nano-second.

It is important to realise that not all context switches are equal even on the same machine. If the processor is switching between two threads of the same process, it won't have to switch in most of the information about memory, since the threads will use the same memory space. Because of this, the memory cache might also be more efficient due to locality of reference.

It is important to remember that there are plenty of things that do need to be included in the context switch when two threads of the same process are swapping, including registers (including the PC), and the stack.