

Operating Systems Concepts (MSc Computing and JMC2)

Course Summary

Olav Beckmann

March 22, 2006

A few comments to start

This course summary is intended to give you an overview of the course material and to help you see how the different topics fit together. Do not take the amount of text I have written for a topic as an indication of how important that topic is — this is more of a function of how much time I had and the order I wrote this summary in (which is not from start to finish). The best guide on what is important is how much time I spent on it during the lectures.

1 Introduction

- What are Operating Systems and why do we need them?
 - Operating systems as a virtual machine
 - “Virtualising” access to diverse hardware
 - “Virtualising” constructs such as files
 - It is only by turning highly diverse, actual hardware into a virtual machine that it becomes realistic to program this machine and that selling software becomes a workable proposition.
- History operating systems
- Operating systems are computer programs — typically written in C or sometimes C++, with a little bit of assembler. In Chapter 6, we see a complete example of a workable operating system that you can compile and boot.
- Challenges to the idea of a virtual machine on which I can run any program, independently of the actual hardware:
 - What if the machine has too little memory? See Chapter 8.
 - What if the machine is already running lots of other application programs? We need timer interrupts (Chapter 3 and also Chapter 6) to make scheduling decisions on which process should get to run (Chapter 7).
 - What if other programs behave maliciously and try to steal your secrets? The operating system has privileged control over address translation hardware. This can be used to ensure that each application is only allowed access to its own data (see Chapter 8).
 - What if two applications try to access the same device, such as a printer. This is solved with various methods of ensuring mutual exclusion, see Chapter 4.
- One important thing to have straight right from the start:
 - *Processor*: “a piece of metal” (well, silicon, wires etc actually). It’s something you can touch, i.e. hardware. It’s fixed, doesn’t change once you’ve bought your computer.

- *Program*: A file on disk that contains the machine code (i.e. a sequence of instructions) for the processor to do something, such as wait for input from the user, add two numbers, draw something on screen etc. Example: the `calc.exe` calculator program which will be present on almost all computers running Windows.
- *Process*: One instance of a program running on a processor. Processes — running programs — are the key abstraction in operating system design. In practice, you can observe multiple processes by launching two copies of the calculator application, resulting in one processor, one program but two processes. A process is an abstraction because it consists of a number of things: the program code, its data and its state. One processor can only execute instructions from one process at a time, so part of the job of the operating system is to make it appear as if lots of processes are running at the same time, via time-slicing.

2 Computer Organisation

This chapter illustrates how computers work with a much-simplified design, the NARC (“not a real computer”).

2.1 How a Basic Computer Works

What I really mean by “computer” here is in the first instance a processor with some memory that is connected via a bus (a collection of wires). A little bit later (Chapter 3) we look at how to connect external devices for input and output to such a computer.

- A processor has some arithmetic and/or logic units and *registers*. Our NARC has just very few registers: an *accumulator* register *ACC*, a *program counter* *PC*, a *link* register *LNK* and a register to store the current instruction word which is being executed.
- Processors operate on data items (“words”) of a certain bitwidth, typically 32 bits. You will hear a processor being referred to as a “32-bit machine”; this means that it works on words of 32 bits. In that case, its registers will be 32 bits in size, and, since instructions typically have to fit into a register, instructions will also typically be 32 bits in length.
- Memory can be either word-addressed or byte-addressed. Word-addressed means that the address of word 1 (starting the count from zero) is 1. In a byte-addressed 32-bit machine, the address of word 1 would be 4. The NARC is word-addressed for simplicity, most normal machines are *byte-addressed*.
- Instruction set design: A crucial (arguable the most crucial) part of designing a processor is to design the instruction set. This means both deciding what the structure of one instruction is — in the case of the NARC this is 8 bits for the opcode and 24 bits for the operand — and what the list of instructions should be that this processor can execute. ¹
- One important thing to remember is that the program code is stored in memory, just as data is stored in memory. Chapter 5 addresses this in more detail, and Chapter 8 shows how we can make sure that we keep program code and data distinct.

2.2 The Fetch-Execute Cycle

A processor continuously executes the so-called fetch-execute cycle: fetch an instruction from memory (from the location described by the program counter register) and then execute this.

¹Having a fixed width for instructions, such as 32 bits and a fixed width for opcodes and operands makes building the processor a lot easier. There are, however, some notable designs that do not make this simplification, for example the Pentium’s i386 instruction set. There are also designs where more than one instruction is packed into a “very long instruction word” (VLIW). One notable, and very fun, example is Intel’s Itanium architecture.

- One really important thing to be clear about is this: the fetch-execute cycle, which is demonstrated as C code in the lecture notes, is implemented in a processor as a *digital circuit*, i.e. as hardware that continuously performs these steps.
- Instruction execution: The control unit of the processor generates a sequence of control signals that sets the multiplexers / demultiplexers on the processor in order to execute an instruction. What the sequence of control signals is that has to be generated depends on the instruction. This sequence of steps that has to be performed to execute a single instruction is sometimes known as *microcode*.

2.3 Memory-mapped Input and Output

- The most basic idea for connecting input and output devices to a simple computer is to map their data input and data output lines into the processor's (memory) address space.
- When we combine this with using (some of) the processor's data output lines for *selecting* either memory or an external device, we have **programmed I/O**: I can program when I want to read a key from the keyboard etc, and once I have selected the keyboard I need to keep reading it to see whether a key has been pressed.

3 Input and Output

There are three major ways of doing input and output, and it is important to have a clear understanding of these three approaches and their advantages and disadvantages.

- Programmed I/O
- Interrupt-driven I/O
- Direct Memory Access (DMA)

3.1 Programmed I/O

Described in part in Section 2.3. Programmed I/O relies on *polling* for noticing when a device has some input available (or when output has completed in the case of output devices). Polling has various problems: we can miss input when we are not checking just at the right point in time. It also wastes time when there is actually no input happening.

3.2 Interrupt-Driven I/O

- Handling interrupts requires modifying the fetch-execute cycle.
- In a basic design, further interrupts have to be disabled while an interrupt is being processed.
- Each interrupt device needs an interrupt handler — the code that is going to run when that interrupt occurs. Note that this code can end up being run at almost any point in time because interrupts occur asynchronously, i.e. they can happen in between any two instructions the processor is executing. We have to take great care to make sure that running the interrupt handler will not interfere with the state of the process that was interrupted: after the interrupt handler code finishes, the registers of the processor, including the program counter, have to be back exactly to the values they had before the interrupt occurred. Chapter 6 shows in detail how to do this for a real system.
- When doing interrupt-driven I/O, device drivers, i.e. the code that handles devices, have a top half and a bottom half. The top half is called by a user process and requests or receives I/O. The bottom half is the interrupt handler code that is called when the device sends an interrupt signalling either that input is available or that output which was requested is complete.

- With interrupt-driven I/O, it is far less likely than with programmed I/O that we could miss input, but not impossible: if an interrupt occurs while interrupts are disabled, it will not be handled.
- A weakness of interrupt-driven I/O which is addressed by DMA is that all data from external devices has to pass through the processor: when data is read from disk into a location in memory, the processor has to read the data (when interrupted) and store it in memory. This is arguably a waste of the processor's time.

3.3 Direct Memory Access (DMA)

DMA relies on a DMA module, a separate piece of hardware from the processor.

- The job of the DMA module is to transfer data directly between memory and the external device, without involving the processor.
- In order to do this, the DMA module needs access to the system bus, which could of course be used by the processor. DMA has to therefore *steal* a cycle from the processor to access the bus, in effect pausing the processor for one cycle. Cycle stealing is not the same as an interrupt!
- DMA is given priority over the processor when both seek to access the bus because I/O is typically the slowest activity a computer performs, and it pays in terms of overall system utilisation (resource usage, throughput etc) to not let a slow activity (I/O) wait for a fast activity (the processor). For example, the result of I/O completing could be that a process terminates, freeing up resources.

3.4 Some Specific I/O Devices

- Hard Drives. Interesting issues with hard drives are
 - Cylinder skew
 - Drive geometry
 - Disk capacity
 - Disk arm scheduling algorithms²
 - RAID
- Clocks, see also Chapter 6.

4 Concurrency

Concurrency is of critical importance in building operating systems and a range of other systems that consist of multiple interacting processes. Virtually every textbook on operating systems will cover concurrency; this summary is therefore brief.

- Distinction between apparent and real concurrency
- Underlying the ideas of handling concurrency is the abstraction of a *process*: an instance of a running program, consisting of code, data and state.
- Any operating system that allows more than one process to be run concurrently (apparent concurrency) has to have a mechanism for *context-switching* between processes. This is shown in detail for a real operating system in Chapter 6.

²One of the best ones is known as the elevator algorithm which every user of a public lift intuitively knows, at least if a lift does not implement it.

- Processes can share code, and sometimes data as well. When we share data, we have to be careful about concurrent or time-sliced access to those data.
- Reasoning about concurrency is made hard by non-determinism, which in turn is largely a consequence of interrupt-driven I/O (in particular of the clock, which can cause process switching). Non-determinism in this context means that we do not know when looking at a sequence of instructions from two separate processes when and how these will be interleaved, i.e. when we will stop executing instructions from one process and continue executing the second process.
- Types of process interaction
 - Sharing, with mutual exclusion
 - Synchronisation
 - Communication
- **Critical Sections**

It's important that you have a clear understanding of what a critical section is and know how to recognise them.

4.1 Locks

Locks are the most basic concurrency construct. Locks, when implemented properly, achieve *mutual exclusion*, i.e. only one process can access shared data, or execute the code in a critical section, at one point in time.

- Implementing locks properly requires careful thought, and there are poor ways of doing it (disabling all interrupts) and better ways (indivisible instructions). See lecture notes and also the first assessed coursework.
- Locks come with *busy-waiting*: When a lock is set, another process that wants too access the critical section needs to keep on checking whether the lock has been lifted until it is allowed to proceed. This can be a waste of processor resources if the critical section is long.
- It's easy to lock out other processes from running if you make a mistake in using locks. For example: if you allow user processes to lock a resource, what happens if that process crashes before the resource is unlocked? This needs to be thought through very carefully.

4.2 Semaphores

- Semaphores offer a key improvement over locks: busy-waiting is replaced by a queue of waiting processes that is managed by the operating system.
- Furthermore, semaphores generalise the binary state of locks (set or not set) to N states.
- All three basic types of process synchronisation can be implemented using semaphores:
 - Mutual exclusion (requires one semaphore)
 - Synchronisation (requires one semaphore)
 - Communication (requires three semaphores for a bounded buffer)

4.3 Monitors

Functionally, monitors are equivalent to semaphores, there is nothing that cannot be programmed with semaphores that can be programmed with monitors.

- Monitors are abstract data types that have member variables, access methods etc, but in addition, over and above a "normal" abstract data type carry the semantics that only one process is allowed to execute in the access methods of this type at a time.

- The advantage of monitors over semaphores is that we combine the data we are protecting (for example the buffer in a bounded-buffer communication) with control on how to access the data, and that this is enforced by the abstraction mechanism of the programming language. (When using semaphores to protect a bounded buffer, there is no reason why we couldn't forget to protect access to the buffer with semaphore calls in some parts of the code.)
- The `wait` and `signal` calls can be used for a process that is executing in monitor code to suspend itself (allowing other processes to run) and resume a suspended process, respectively.

4.4 Deadlock

All types of processor synchronisation primitives: locks, semaphores and monitors can lead to deadlock. In its most basic form, deadlock happens when two processes hold a resource each, and are each waiting for the other to release the resource the other process holds before releasing their own. Detecting the potential for deadlock in a multi-process environment is hard.

5 Programs as Data

This chapter was not part of the course last year, but that doesn't mean it's less important! The material is also not covered directly in the main recommended textbook, so this has to be studied carefully from the lecture notes.

The main aim of this chapter is to give you a practical and concrete understanding of what the steps are that are necessary in order to turn a source-code file into a running process. Four programs that are involved are

- Compiler: translates high-level source code into assembler code³
- Assembler: translates human-readable versions of machine code into (binary) machine code that can be interpreted directly by the processor. The human-readable and machine-readable versions have a direct correspondence, the only additional information in the human-readable version are names for variables and pieces of code which are convenient to us but which the processor does not need.
- Linker: combines separately compiled and separately assembled pieces of code together. Linkers allow us to develop libraries that contain frequently-used functionality that we do not want to include as source code in every program we write. Linkers also facilitate writing programs that consist of more than one file of source code that are developed and compiled separately.
- Loaders: take a linked binary and turn it into a code in the processor's memory that can be executed.

5.1 Assemblers

- Understand how labels in assembly code correspond to addresses in object code.
- Two-pass assemblers: The first pass reads the assembly code and generates a symbol table, which is a list of all labels and their addresses. The second part reads the assembly code together with the symbol table and generates the object code. The second part resolves names: when a variable is referred to by name in the assembly code, this has to be replaced by an address (a number) in the object code.

³You may wonder why you don't get assembler code as the output when calling the `gcc` 'compiler' on a simple source code program. The reason is because `gcc` actually calls a compiler, followed by an assembler, followed by a linker. Sometimes we refer to the whole sequence as compilation for convenience, but it is actually three steps.

5.2 Linkers

When a program is combined out of separately compiled sections of code, two issues need to be addressed:

- Relocation: when we concatenate (paste) together the object code for the different files, addresses need to be adjusted. Whereas we counted from zero when assembling each individual file, we may combine these in various different ways, resulting in various different starting addresses for the code and data from each of the separate files.
- Name binding: there is no point in combining pieces of code if they do not share functionality (functions) or data. So we need to have a mechanism for allowing code in one file to refer to names that are defined in a different file.
- To facilitate relocation, object files have to be augmented with information on which addresses need to be adjusted; there is no inherent way of recognising such addresses unless that information is carried over from the human-readable assembler code.
- To facilitate name binding, each object file contains an *external symbol table*, which lists the variables and functions that are either externally accessible in this file or that this file needs to access from somewhere else. External symbol tables list
 - The name of the symbol (e.g. the function or variable name)
 - The value of the symbol (i.e. the address)
 - The type of the symbol
 - * T: text, meaning code, i.e. in practice a function that can be called
 - * D: data, meaning a variable
 - * U: undefined, meaning this file has to be linked with another file that defines this label before the code can be executed. Undefined labels can be variables or functions. One common source of undefined labels are library functions that are called from separately compiled user code.
 - Note that where an undefined symbol occurs multiple times in an object file, the external symbol table holds one of its locations (where the actual value needs to be written by the linker). The object file contains the other locations in the form of a (zero-terminated) linked list.

5.3 Loaders

The fact that we cannot generally predict what order the programs that run on a computer system will be started in means that we need to be prepared to run a program at any starting address in memory. *Important note:* Paging solves this problem in a different, and arguably more elegant manner, by giving every process its own virtual address space starting at zero. There are, however, many systems that use the kind of absolute addressing discussed here.

- In order to solve this problem, loaders for systems that use absolute addressing have to perform *relocation*, i.e. adjusting addresses with respect to the actual starting address where the code is being stored in memory.

5.4 Libraries

Libraries are collections of object files that have been compiled separately, and which typically implement important functionality (such as basic I/O). When linking a collection of object files with unresolved external symbols against a library, the linker will extract those object files from the library that contain these undefined symbols.

6 The Operating System Kernel

- Operating system design issues
- Computer systems are built in layers: hardware, OS kernel, possibly OS modules that are not part of the kernel, system libraries, application libraries, user programs.
- OS design decisions include deciding on the interface (boundary) between these layers. An operating system is characterised by its *system call* interface.
- **Operating System Kernel**
 - What are the key functionalities of the kernel?
 - Simple Kernel design: what are the Simple Kernel system calls? I expect you to have an understanding of what the system calls shown on slide 9 do.
 - Figure 1 is an overview of the Simple Kernel source code by its file structure. I do not expect you to memorise this, but it might be useful if you want to find a particular OS functionality in Simple Kernel and look at its implementation.
- Simple Kernel Target Architecture: i386
 - I do expect you to know that there *are* general-purpose registers, and I expect you to know about the following special-purpose registers: `ebp`: base / frame-pointer, `esp`: stack pointer, `eip`: program counter / instruction pointer and `eflags`: processor status register.
 - I do not expect you to know details about the flags register, except that it controls whether or not interrupts are enabled and that its state has to be preserved on context switch and interrupts (see below).
- **Kernel Entry and Exit**
 - It is important that only one process can be executing in kernel mode at a time.
 - The only way in which an “unplanned” switch between processes can happen is as the result of an interrupt. We therefore guarantee mutual exclusion of processes in the kernel by disabling interrupts while a process is in the kernel.
 - The assembler functions `int_mutexon` and `int_mutexoff` disable and re-enable interrupts. `ENTERKERNEL` and `EXITKERNEL` are C macros which call these functions.
 - It is important to realise that all `system` calls must begin and end by calling these macros. Only system calls are permitted to be called from user programs. Other functions, such as the scheduler `dispatch` are only called from system calls.
- **Process Management in Simple Kernel**
 - Kernel representation of processes: process control block (PCB)
 - Memory layout: the stack for each process grows downwards (i.e. towards address 0). Note that *structures*, such as the `Context` structure are laid out in consecutive locations (words in the case of the `Context` struct) in memory, starting at the lowest (closest to zero) address. There is an illustration of this on slide 18. Similarly, for *arrays*, element 0 is at the lowest physical address.
 - Context switching at a conceptual level: saving registers from the processor in the PCB of one process and restoring to the processor from the PCB of another process.
- **Scheduling in Simple Kernel**
 - How are the ready queues stored? I do not expect you to memorise the functions for managing the FIFO queue data structure, but I do expect you to understand the ready queues data structure.

Directory	File	Functionality
system	console.c	console output
	init32.S	kernel initialisation in assembler, including setup_idt to set up interrupt descriptor table
	int.c	Level 2 interrupt handler int_interrupt and install system call
	int_asm.S	int_mutexon and int_mutexoff assembler functions, macros for creating level 1 interrupt handler assembler functions.
	io_asm.S	Assembler implementation of basic input and output functions.
	kbd.c	Functions for handling keyboard input
	main.c	main function, which calls user_init (defined in user.c) to create user processes and then calls startKernel.
	mem.c	Simple Kernel memory manager
	proc.c	Process handling functionality: process creation (create system call), scheduling (dispatch), ready queue management via setready and proc_init.
	queue.c	Generic process queue management functions. This is used for ready queues and waiting queues (processes that are suspended waiting on a semaphore).
	sem.c	Semaphore handling: initSema, P and V system calls.
	serial.c	Serial interface handling
	switch.S	Assembler implementation of the pswitch function for context-switching
	time.c	Implementation of the delay and time system calls, implementation of the tick timer-interrupt handler.
user.c	File containing processes written by the user. Defines user_init, which gets called from main to create these processes at system boot.	
include/icos	conP.h	Internal console header
	console.h	Public console header
	int.h	Public header for install system call
	intP.h	Internal header for interrupt handling. Defines ENTERKERNEL and EXITKERNEL macros.
	kbd.h	Public keyboard-handling header
	kbdP.h	Internal keyboard-handling header
	mem.h	Public memory management header
	memP.h	Internal memory management header
	proc.h	Public header for process management, declares create system call.
	procP.h	Internal header for process management, defined PCB (process control block) structure.
	queue.h	Process queue handling header
	sem.h	Semaphore interface. Defines semaphore structure and headers for initSema, P and V system calls.
	serial.h	Public serial interface header
	serialP.h	Internal serial interface header
	stdlib.h	Header which includes a collection of internal header files.
	systemP.h	Some internal constant definitions, including KERNEL_CS.
	time.h	Public time interface header, declares delay and time system calls.
	timeP.h	Internal time handling header
user.c	Header for user_init	

Figure 1: Simple Kernel source code structure. Note that the classification into internal and public headers is merely the intended use — there is no protection of functions in internal headers. Public headers are those defining functions that are intended to be called from user programs.

- What are the possible states a process can have in Simple Kernel?
- What causes each of the transitions between states?
- What is the Simple Kernel scheduling policy? Given a particular set of processes in the ready queues, which process will be chosen to run?
- Process creation: I only expect you to know that a PCB has to be allocated and that the initial stack pointer will point to the highest address in the allocated stack, so that the stack can grow downwards.

- **Semaphores**

- I expect you to know how the `P()` and `V()` functions affect the state a process is in.

- **Time Operations**

- Delay queue data structure: I will not ask you to write out the functions for managing the delay queues, but I do expect you to understand the data structure (i.e. for example the exercise on slide 32).
- The `tick()` function is a third-level interrupt handler (see below), written in C, which gets called on every hardware clock interrupt and which *calls the scheduler*.

- **Interrupts**

- What exactly happens with the stack on a function call? How do buffer overrun attacks work?
- How do interrupts differ from function calls?
- What does the (i386) hardware do on an interrupt: The device signals interrupt; when the CPU indicates that it is ready, the device puts the *8-bit interrupt number (= interrupt vector)* on the CPU pins. CPU then pushes `eflags` register, code segment register and program counter (`eip` register) onto the stack. *The processor also clears the IF flag in the eflags register, thus disabling any further interrupts for the time being.* The processor then jumps to the first-level interrupt handler routine, for which a *far pointer* is stored in an *8-byte interrupt descriptor*. The operating system sets up a global *interrupt descriptor table (IDT)*, which holds an 8-byte interrupt descriptor for every possible 8-bit interrupt vector, i.e. the IDT has 256 entries. The processor finds the interrupt descriptor for a particular 8-bit interrupt vector by scaling the interrupt vector number by 8 bytes and adding this to the base address of the IDT. The OS informs the processor at boot time of where the IDT is stored by loading a special register with the base address of the IDT.
- What is the point of having three levels of interrupt handlers?
- What do they each do:
 - * Level 1: assembler routine, generated by a macro for each interrupt number, saves all registers on the stack and calls second-level interrupt handler passing interrupt number as parameter. On return from the second-level interrupt handler, this assembler function also restores all registers from the stack. The first level interrupt handler exits with an `iret` instruction which has the effect of restoring the `eflags` register from the stack, *thus re-enabling interrupts*. See Figure 2 if you are having trouble with the assembler.
 - * Level 2: C function which takes the interrupt number as a parameter and selects the right function to run as the third-level interrupt handler. This is done via an array of function pointers which is managed by the OS.
 - * Level 3: C function that does the actual work to be done when a particular interrupt occurs. Example: `tick()` handles the clock interrupt.
- What happens with the stack when an interrupt occurs?
- Take a look at the solution to the tutorial exercise on interrupt handling.

- **Context Switching**

- I do not expect you be able to write out the assembler function `psswitch` in the exam!
- I do expect you to understand why this function has to be written in assembler.
- I do expect you to understand what this function does: registers being copied to one PCB in memory, restored from another.
- There is a brief summary of some useful features of the assembler used in Simple Kernel in Figure 2.

- **Kernel Initialisation**

- This initialises the process management data structures (the ready queues), sets up timer data structures (such as the delay queue), sets up the interrupt descriptor table and calls the scheduler for the first time to launch the first process to run.

1. Almost all registers are 32-bit. This means that we use the extended registers, i.e. `eax` rather than `ax` etc. The only exception are segment selector registers such as `ds`, which are 16-bit.
2. Registers are referred to in the assembler syntax with a `%`, i.e. for example `%eax`.
3. Assembler operations on 32-bit registers have an `l` suffix, such as `pushl`. The `l` stands for long and indicates that the operands are 32-bit.
4. Assembler functions which return an *integer* result use the extended accumulator register (`eax`) to return the function result. No space is allocated on the stack for the result of such functions. *Don't do this at home — or in an exam about assembler programming!*
5. For operations with two operands, the first operand is the *source* and the second operand the *destination*.

Figure 2: AT&T Assembler Syntax used in Simple Kernel — Points to note.

7 Scheduling Strategies

- **Priority Scheduling with Pre-emption**

- This is used in the Simple Kernel scheduler. I expect you to understand the scheduling algorithm used by Simple Kernel well, especially because you had to think about this for the assessed coursework.
- Priority scheduling means that each process is assigned a priority. In Simple Kernel, the scheduler will always select the process with the highest priority (if there is more than one, the one that has been waiting longest) to run. This means that a lower-priority process can be *pre-empted*: once a high-priority process becomes ready, any lower-priority process will be descheduled as soon as the scheduler is called.

- **Run-to-completion or Natural Break Scheduling**

- This means that each process runs until it suspends itself (in the case of Simple Kernel this could be by blocking on a semaphore or by calling `delay()`).

- **Time Sliced Scheduling**

- Each process is run until its time slice expires or until it suspends itself. Simple Kernel behaves like a time sliced system if all active processes have the same priority.

- **Static vs. Dynamic Priority Assignment**

- Simple Kernel is a *static* priority scheduling algorithm. This means that the priority of each process is assigned once at process creation and does not change. A *dynamic* priority scheduling algorithm varies the priority of processes depending on different factors, for example how much CPU time they have recently consumed (for example, if a process has recently consumed a lot of CPU time, this might be reason to lower its priority in the interest of fairness to other processes).
- Priority scheduling can be combined with assigning processes of different priorities different lengths of time to run each time they are scheduled (Simple Kernel does *not* do this). One example is that a system which is designed to have good interactive response would assign interactive processes a high priority and a very short time-slice. Non-interactive processes can be assigned low priority and long time slices.

- **Scheduling in UNIX**

- Unix scheduling is a dynamic priority scheduling algorithm (see above). See also Tanenbaum Chapter 10 (Section 10.3).

- **Summary of (Traditional) UNIX Scheduling**

There are many different versions of UNIX and many slightly different versions of the UNIX scheduler. Below I describe one version which is meant to be consistent in itself and which will be very similar to most versions of the UNIX scheduling algorithm, but does not necessarily exactly correspond to any one of them.

- * Each process is assigned a priority number by the Operating System. This depends on the type of process it is and is hard-coded (i.e. cannot be changed while the system is running). Let us call this priority number *base*, and let us assume that the range is $-20 \leq \text{base} \leq 20$. *-20 is the highest priority, 20 is the lowest priority!*
- * A user can decide that they want to be nice to other users and give one of their processes a lower priority. For this purpose, every process has a value, called *nice* associated with it. The range of possible values for *nice* is $-20 \leq \text{nice} \leq 20$. Users can set *nice* to any value between 0 and 20, only the superuser can set *nice* to a negative value.
- * When a process first starts, its priority is calculated as $\text{priority} = \text{base} + \text{nice}$ (making sure the value does not fall outside the range $[-20, 20]$).
- * Let us assume that there is one ready queue for each priority (i.e. 40 ready queues).
- * When the scheduler runs, it selects the first process from the highest non-empty queue to run (this part is just like Simple Kernel).
- * A process that is selected to run is assigned a time-slice (“quantum”) of 100ms. A process runs until it blocks on a semaphore (in which case the scheduler is called again) or until its quantum has been used up (in which case it rejoins the end of the ready queue for its current priority). The effect of this is that we get round-robin scheduling for processes of equal priority (just as in Simple Kernel).
- * The main difference between this scheduling algorithm and Simple Kernel is that priorities are re-calculated once every second, taking into account recent CPU utilisation:
- * Let CPU_i refer to the *number of time slices (quanta) for which a process has been running during last second*. Since each time-slice is 100ms, the maximum value is 10 (if a process was running during every time slice) and the minimum value is 0 (if the process was not running).
- * We define the *CPU utilisation penalty* for the last second, CPU_usage_i , as follows:

$$\text{CPU_usage}_i = \text{CPU}_i + \frac{\text{CPU_usage}_{i-1}}{2} . \quad (1)$$

So if a process runs during an entire 1-second interval, its `CPU_usage` value will be 10. If it then does not get re-scheduled during the next few seconds, `CPU_usage` value will then be 5, 2, 1 and 0 (assuming we round down).

- * At the end of every 1 second interval, the priorities of all processes are re-calculated based on

$$\text{priority} = \text{CPU_usage}_i + \text{nice} + \text{base} \quad (2)$$

again making sure the value does not fall outside the range $[-20, 20]$.

- * This scheduling algorithm is designed for multi-user systems and avoids CPU-hogging by individual processes via the priority penalty that is assigned to processes based on their recent CPU utilisation.

• Scheduling in LINUX

- See also Tanenbaum Section 10.3.

- Linux has three different scheduling classes: “real-time” FIFO, “real-time” round-robin and timesharing. The two “real-time” classes are not genuinely real-time, no deadlines can be specified. We will only look at the timesharing scheduling class.

– Summary of Linux Timesharing Scheduling

- * Linux timesharing scheduling differs from UNIX scheduling, but like UNIX scheduling, it is a mixed static-dynamic priority scheduling algorithm.
- * This description, like the description of the UNIX scheduler is primarily meant to be consistent within itself and gives a good representation of what the Linux scheduler does, but it may not completely correspond to the scheduler in a specific version, especially with respect to the terms I use.
- * Each process has a static priority, which we will call `priority`, and a “niceness”, which we will call `nice`. *Static priority and niceness have a fixed relationship, as follows:*

$$\text{priority} = 20 - \text{nice} . \quad (3)$$

- * The default static priority for user processes is 20 (note this corresponds to niceness 0). The valid range of priorities is $[1, 40]$, 1 being lowest and 40 highest.

- * Linux allows the priority of a process to be adjusted when the process is started:

```
/bin/nice -n <niceness> <command>
```

The above command starts a process with a specified niceness. `/bin/nice -n 5` starts a process with niceness 5, corresponding to priority 15 (i.e. lower than the default). Only root can invoke `/bin/nice` with a negative number resulting in a higher-than-default priority.

- * In addition to the static priority, each Linux process also has a “quantum”, which one can think of as a dynamic priority. `quantum` is initialised to the value of `priority`, i.e. 20 by default.
- * The timer interrupt is set up to give a clock tick every 10ms. This 10ms interval is referred to as a “jiffy”. On every clock tick, the quantum of the *running* process is decremented by 1.

- * Scheduling decisions are taken in terms of the goodness of a process:

$$\text{goodness} = \text{quantum} + \text{priority} \text{ if } \text{quantum} > 0, \text{ otherwise } 0 . \quad (4)$$

- * When the scheduler runs, the process with the highest goodness greater than 0 is selected to run. Therefore, a process can be descheduled if its quantum and therefore its goodness reach 0, or if it blocks on a semaphore, or if a process with higher goodness becomes ready. The scheduler is not run every 10ms (that would be too expensive); instead, the scheduler runs when the quantum of the running process becomes zero, and after semaphore operations etc.

- * Eventually, the goodness of all ready processes reaches 0. When that happens the quantum of *all* processes is recalculated based on

$$\text{quantum} = \frac{\text{quantum}_{\text{old}}}{2} + \text{priority}. \quad (5)$$

The period between re-calculations of the quanta is referred to as a scheduling *epoch*. Note that those processes that did not exhaust their quantum in the previous epoch will have some old quantum remaining and will thus get a larger initial quantum in the next epoch.

- * Consider a Linux system that has two CPU-intensive processes A and B. A is started normally (i.e. with niceness 0 or priority 20), B is started with `/bin/nice -n 15`, i.e. with priority 5. A will have an initial quantum of 20, therefore a goodness of 40. B will have an initial quantum of 5, goodness 25. A will run first, and will run for 20 jiffies, i.e. 200ms until its quantum hits 0. Then B will run for 50ms. At the end of the epoch, the quantum of both A and B will revert to its original value because neither had not used its full quantum. Therefore, A will get about 80% of CPU time, B will get 20%, which corresponds exactly to the ratio of their priorities.

– Priority Inversion

- * This was the topic of the assessed exercise.
- * I expect you to understand what the priority inversion problem is. *I also expect you to understand the kind of timing diagrams as I showed on page 2 of the lecture slides about the assessed exercise.*
- * I will not ask you to explain how to fix the priority inversion problem in Simple Kernel.

8 Memory Management

- Advantages and disadvantages of single contiguous store allocation
- Overlays: This is essentially memory management by the application program writer, based on their knowledge of the calling structure of their code.
- Fixed partition store allocation: this is not used in most modern operating systems for memory management, but similar schemes are now used in managing large servers that provide different “virtual servers” within one machine, each of which is guaranteed to provide a specified amount of resources.
- **Dynamic Partition Store Allocation**
 - Each job is allocated a contiguous region of memory, corresponding to its job size, at job creation.
 - One advantage over fixed partition store allocation is that space is not wasted in that the size of the allocation corresponds to the actual job size, rather than fitting into a fixed partition that may be larger than the job size. Note that paging does not have this advantage, since the amount of space used is always in whole pages, resulting in some (limited) wastage.
 - The key disadvantage is store fragmentation, which means that a new job may not be able to start because the amount of memory it requires is not available as a single contiguous chunk even though enough memory is available overall. This can be fixed via compaction and relocation. It is also overcome by paging.
 - Another disadvantage of dynamic partition store allocation is that we cannot run jobs that require more memory space than we have physically available. This is also overcome by paging.

- The store fragmentation problem can be overcome by periodic compaction. Note that compaction is expensive because we need to physically move the memory regions that the currently running jobs occupy. Since paged stores do not suffer from fragmentation, they do not require compaction.
- Compaction requires running programs to be *relocatable*.
- The key mechanism for enabling relocation is *virtual addressing* — addresses are not hard-coded, but are relative within each job. An indirection function is used to look up the actual physical address. One neat implementation of this indirection function is to store a base and limit register for each job. Adding the base register to the relative address gives the physical address. The limit register can be used to check that an address does not fall outside the section allocated for the current job. *Note that this virtual addressing scheme is different from the virtual addressing scheme used in paging.*
- We briefly discussed four placement algorithms for dynamic partition store allocation: best fit, worst fit, first fit and buddy. These algorithms are all to do with *where to allocate a job, given a store with a number of different free regions that are large enough*. Note that paged systems do not need to deal with this issue.

- **Swapping**

- Swapping moves the memory image for entire processes to secondary storage (disk) in order to free up space for other processes to run. This can increase overall CPU utilisation if processes block for a long time on semaphores etc. It also means that there is less need to compact a fragmented store. It still does not resolve some key problems, as stated in the lecture notes.

- **Paging**

- Paging overcomes several disadvantages of the dynamic partition memory management scheme, in return for some overhead in space and time required for memory management.
- Some useful facts when talking about paging: Kilobytes: $1\text{KB} = 2^{10}\text{B} = 1024\text{B}$. Megabytes: $1\text{MB} = 2^{20}\text{B} = 1024\text{KB}$. ($1024 \times 1024 = 1048576$ but I do not expect you to remember this.) Gigabytes: $1\text{GB} = 2^{30}\text{B} = 1024\text{MB}$.
- Application programs in a paged system use a *virtual address space*, which is limited in size only by the bit-width of virtual addresses. So on an i386 machine, where virtual addresses are 4 bytes or 32 bits wide, the virtual address space is $2^2\text{GB} = 4\text{GB}$. The mapping of virtual addresses to physical addresses in the computer's memory is done transparently by the operating system (with help from the hardware). This is an obvious advantage in terms of programmability.
- The virtual address space is divided into *pages* of equal size (for example 4KB in size). Similarly, physical memory is notionally divided into page frames of the same size.
- A virtual page can be mapped to any physical page. The mapping is managed by the operating system. Assuming a process occupies contiguous space in virtual memory⁴, there is no reason why its pages need to be
 - * mapped to contiguous physical memory (therefore paged systems do not have to deal with fragmentation)
 - * all present in physical memory (therefore we can start running a process even if not all its pages fit into physical memory).

Pages that contain useful information which are not currently held in physical memory are stored on disk. So if all pages of a process are currently on disk, this has the same effect as swapping a process out, but with the benefit that not all pages need to be returned to memory when we resume executing.

⁴This may not always be the case if we are also using segmentation, but that doesn't matter here.

– Address Mapping for Paging

- * This is covered in the lecture notes and also a tutorial exercise. I expect you to understand this well.
- * Each address is split into the bits that represent the page number and the bits that represent the offset within the page. For 4KB pages, 12 bits represent the offset within the page. Therefore, with 32-bit virtual addresses, 20 bits are used for the page number.
- * The (process) page table is used to hold the mapping from virtual pages to physical pages. The page table needs to have an entry for each virtual page, i.e. with 20-bit page numbers, the page table has to have 2^{20} entries. Each page table entry holds the corresponding physical page number (if the page is in memory, else it holds an address for where to find the page on disk), as well as some status bits. The physical page number will typically have the same number of bits as the virtual page number (i.e. 20 in our example). In practice, each page table entry will be a full number of bytes, and more likely a full number of words wide (so in our example, where we have 20 bits for the physical page number, in practice the page table entry will be 24 or 32 bits wide).
- * A *page fault* occurs when a lookup to the page table finds that the virtual memory location we require is not in physical memory but on disk. Page faults will result in the process being suspended by the operating system so that other processes can run while the page is loaded to memory from disk.
- * The status bits a the page table entry include: presence bit — indicating whether the page is in physical memory, “dirty” bit — indicating whether the page has been modified and protection bits, which can be used to restrict the way in which a page is used (for example read-only data). The dirty bit can be used to decide what to do if a page has to be “paged out” from physical memory: if the page has not been modified, it need not be copied to disk but can simply be over-written.
- * Paging is potentially expensive, since every memory access has to go through the virtual-to-physical indirection. *Translation Lookaside Buffers (TLBs)* are designed to make the most common indirection lookups very fast. The idea is that if we have just looked up one address we are likely to require other addresses on the same page (spatial locality), and require similar addresses again within a short period of time (temporal locality). TLBs store a small number of the most recent translations (say 64) by keeping a copy of the page table entry in a small piece of very fast memory. If the TLB has $64 = 2^6$ entries, the 6 least significant bits of the virtual page number give the index into the TLB. Many different page table entries could map to this line in the TLB; the idea is that it is likely that it will be the one that we need. We can check whether we have a “hit” by comparing the remaining bits. See the diagrams in the lecture notes and solutions to tutorial exercises.
- * Paging has many benefits and some costs. In terms of space the cost is that we always allocate space in multiples of pages, resulting in on average half a wasted page per process. In terms of time, the cost is the indirection lookup of virtual-to-physical address translation. TLBs can help to hide this cost.

– Paging: Policy Issues

- * Key policy decisions when implementing memory management systems are: placement (trivial for paging: any physical page), replacement (which page should be evicted to free up physical memory space) and fetching (when should pages be loaded from disk).
- * Examples of replacement algorithms are first-in first-out, least recently used, least frequently used and not recently used.
- * Examples of fetch policies are demand-fetching and anticipatory fetching. Demand-paging can result in poor CPU utilisation if there aren’t other processes to run while we wait for pages to be loaded.
- * Locality of reference is a property of programs. Programmers can improve the performance of their programs in a paged memory system by thinking carefully about locality

of reference.

- * Thrashing means that the processor spends more time paging than doing useful work. One way to understand whether or not a program will cause thrashing is to consider the *working set*, that is the set of pages a program is referencing repeatedly within a given time window. We can avoid thrashing if a program's working set is in physical memory. This is modelled by some page replacement and page fetch algorithms.
 - * The other key design issue in a paging memory system is the page size. The lecture notes discuss this issue.
- Segmentation
- * Segmentation is a logical division of the address space depending on function (for example executable program code, read-only data, read-write data).
 - * Segmentation can, like paging, be implemented via an indirection table (process segment table).
 - * Paging and segmentation can be combined (see lecture notes).

9 Some Security Issues

- Design Issues
 - Security has to be considered at OS design time, it is always much harder to deal with if it becomes an afterthought. It is also generally true that system bloat is bad for security.
 - When choosing the right security design, it is important to consider what we are trying to protect against. Do we fully trust internal users? Do we trust the provider of our external Internet connection? Etc...
 - Finally, it is important to consider that more people probably lose data due to accidental deletion than due to malicious external interference!
- Basic cryptography: it's essentially a function mapping a string of bytes to another string of bytes. What is the key idea behind public (symmetric) key cryptography? What are one-way functions? (functions which mathematically are not invertible).
- What is the (very) basic idea behind digitally signing an email message?
- What are some of the issues to consider when designing a password authentication system? UNIX passwords use one-way-functions for encryption. Is that in itself enough? What are some of the other issues to consider?
- Other authentication systems are based on physical objects, finger / palm prints or iris scans.
- A different, and no less interesting security issue is how to prevent users from within a system leaking out information. There are many entertaining tricks to do with covert information channels, such as the example with the zebras picture that I showed you.