

Contents

Operating Systems and Concurrency	1
Lecture 1 introduction	1
The exam	1
Module content	2
Definitions	2
When an OS comes in handy?	2
Concurrency	3
Lecture conclusions	4
Lecture 2 more introduction	4
The basic cycle	4
Memory management	5
Interrupts	6
Hardware	6
Processes 1	7
OS abstraction	10
Processes 2 - Scheduling	11
Process schedulers	12
Performance assessment	13
Algorithms	13
(Processes 2)[http://moodle.nottingham.ac.uk/pluginfile.php/2720824/course/section/705997/processes3%20]	

Operating Systems and Concurrency

Lecture 1 introduction

Module goals:

- Introduce the fundamental concepts and principles of an Operating System and Concurrency.
- Help to understand how programs rely on the OS.
- Understand the basic writing of concurrent(also known as parallel) code.

What you should know by now(Labs):

- Fundamentals of OS concepts
- How to use operating system APIs and how OS schedulers work(HUGE CW we had)
- Concurrency

The exam

- It will be 120 minutes focused on:
 - Knowledge

- Understanding
- Application
- The exam will be 3 of 4 questions accounting for 75% of the module
- Sample questions and exams are on moodle

Past exams(links)

- Exam 2009-2010
- Exam 2010-2011
- Exam 2011-2012
- Exam 2012-2013

Module content

This table is confusing but he gave it out like that

Subject	Lectures	Lecturer
Introduction to OS	1-2	GDM
Processes, Thread	3-4	GDM
Concurrency and Deadlocks	5-6	GDM
Memory management,Swapping and virtual memory	4-5	IT
File Systems	3-4	IT
Visualization	1	IT
Revision	2	IT

Definitions

You should know these by now but let me refresh your memory

- File Systems: Physical location of a file(on disk)
- Abstraction: Covering up complex process with nice high level functions and methods
- Concurrency: Allowing programs to run at the same time(without them causing conflicts)
- Security: I can't be bothered to explain this one come on guys

When an OS comes in handy?

An OS is basically a massive layer of abstraction between code and the actual Hardware and this comes in handy on a series of occasions because as programmers we can forget about hardware limitations(sometimes) and focus on algorithms.

Examples of OS abstraction:

- OSs will find a way of dealing with data when memory(RAM) is full, of course this won't be super fast and efficient but it will make the program work(this is called Swap memory on some operating systems)
- OSs will optimize the usage of memory when only certain data in an array is needed instead of all of it
- OSs will handle processes so a computer can "Multi task" making you resource intensive program bearable to use while doing other tasks.

For these and many more reasons you love your OS because it allows you to be lazy and not have to worry about the hardware most of the time you write code.

Quote this next line on the exam for some ez marks:

"All problems in computer science can be solved by another level of indirection" Dr David Wheeler PhD in computer science 1951

Concurrency

One of the most important features of modern Operating Systems is their implementations of Concurrency. These allow programmers to implement code that will run asynchronously in multiple CPU cores theoretically speeding up your program to up to twice its normal speed. Modern CPUs are what we call multi-core or multi-threaded this means that one single CPU can do multiple operations at the same time, as long of course they don't rely on the result of another core computation.

Kernel mode

To achieve concurrency and what is called Multi-programming(concurrent code) most modern OSs work with multiple modes notably Kernel mode and User mode. The OS will transition between both modes in a controlled manner usually having a mode bit which will distinguish between both operation modes.

- Kernel mode is a more complete version of the lower level OS allowing all instructions available for the CPU
- User mode is one level of abstraction above Kernel mode and allows only a subset of instructions which can and will access Kernel mode when necessary

One way of visualizing this is User mode occurs on applications and programs where kernel mode occurs on the operating system level.

Lecture conclusions

The OS sits on top of hardware and has full access to its features while providing abstraction for the User/Programmer. It also controls the user by switching between different modes with different levels of access to the hardware.

Different OSs have different purposes and implementations but most of the time the focus on the following: Memory management, CPU scheduling, multi-programming, file system, communication, memory management, interrupt handling, GUI, Browser

Lecture 2 more introduction

When talking about Operating systems we are most of the time thinking about abstraction and because of that is important to understand how the computer fetches, decode and executes data.

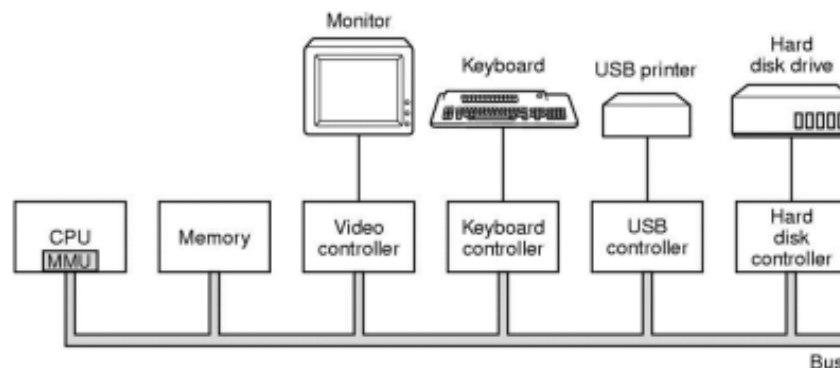


Figure 1: Simplified computer model(Tanenbaum)

The basic cycle

As mentioned before CPUs follow a basic cycle which consists of:

- fetching
- decoding
- executing

This means that regardless of what the CPU is doing it will always need to get some data, understand it and only then modify it. All CPUs are different as in they have different instruction sets and perform operations slightly differently.

However all CPUs have registers as they are essential as they provide really fast memory.

Registers are extremely fast compared to other types of memory. However they are fairly limited on what they can hold and many times are designed to contain important information for the OS such as a program counter and the mode bit.

Memory management

Given the following code the results will be most likely different for the print statement, every time the program is ran.

```
#include <stdio.h>
int iVar = 0;
void main() {
    int i = 0;
    while(i < 10) {
        iVar++;
        sleep(2);
        printf("Address:%u; Value:%d\n",&iVar, iVar);
        i++;
    }
}
```

This is because the program is printing the position of the variable in memory. And the C program does not care where their variables are stored it only asks the OS to store it. That means although the code might stay the same the OS will put the data on a different location as it sees fit. This is good as we can forget about where our variables and arrays are and only worry about names which are a lot easier to work with.

Physical and Logical Memory

This whole following paragraph is overly complicated just know that $physical\ address = logical\ address + offset$ and look at the picture

Memory is quite easily represented as a array of bits we have a position and a value which can be 0 or 1. This can also easily be translated into a array of bytes depending on how you want to see it. While working with memory as an Operating System you start at position 0 until the MAX(that being the amount of memory the computer). However when dealing with smaller scale programs physical memory is not necessary as we won't(at least we should) require the whole memory of the computer. Thanks to that we have something called logical addressing. This means that at the time you run a program it gets its own memory starting from 0 and going as far as it needs(the OS will define that). For the OS its quite simple to deal with this as it only needs an offset value for example if something in memory 0 logical is on address 1024 in real memory the

logical memory at address 1 will be on address 1025 on physical memory. This adds a layer of security as it prevents processes overwriting other's memory.



Figure 2: MMU = Memory management unit

Interrupts

An interrupt is a temporary pause on a process. This occurs for different reasons such as:

- Timer interrupts caused by the CPU clock (Allows for multi-tasking)
- I/O interrupts, due to I/O completion or error codes
- Software generated (Errors)

Because the CPU follows a basic cycle an interrupt will only happen after it has finished its cycle. This prevents an interrupt from corrupting data in a register and affecting the process when the interrupt is over. This means the CPU only checks for interrupts after every cycle.

Hardware

Moore's law is one of the most famous laws when it comes to computer hardware and it states:

“The number of transistors on an integrated circuit (chip) doubles roughly every two years”

This means that in theory computer performance should double every two years. However, this does not work like that because of many other computing bottlenecks. One of the main reasons this doesn't happen is because having twice as much power doesn't mean we can do two tasks at the same time. At

least not in exactly half the time because most of the times we require one task to end before we start the following task.

Parallelism

Parallelism occurs when we have two or more tasks running in parallel(at the exact same time) and those two tasks do not overwrite each other. This is often hard to achieve for a number of reasons. However newer CPUs do a lot of work to allow high level languages to address parallelism in an easier manner. Problems such as load balancing and process scheduling are major areas of work for CPU and OS developers as they need to make parallelism more abstract in order to promote smaller developers to use it efficiently and get the most computing efficiency out of their computers.

Previous exam question: “Describe how, in your opinion, recent developments in computer architecture and computer design have influenced operating system design?” One thing to consider in this question is that Windows XP didn’t support multi threading while today almost every single consumer grade CPU(for PCs) has at least 2 threads or even 2 cores.

Memory hierarchy

This is quite self explanatory but I will mention because it is in the slides. Faster memory is used for processing and slower memory for long term storage following this order:

- CPU cache
- RAM
- Hard Drives

Processes 1

The definition of a process is “the running instance of a program”. A program can be seen as passive as it sits on the disk of the computer while a process is one of its running instances. One program can also be split into multiple processes if designed that way. This allows multi-threaded CPUs to run the program in parallel if the OS supports this feature.

What is important to understand about that graph is that a process contain key features in memory. A stack, a heap, a data segment and the program code. Both the stack and heap are placed at opposite sides of the memory allocated as they might need to grow in order for the program to complete it’s task.

Because a computer core can only do one instruction at a time modern OSs use a simple technique to simulate multi-tasking by quickly pausing and starting

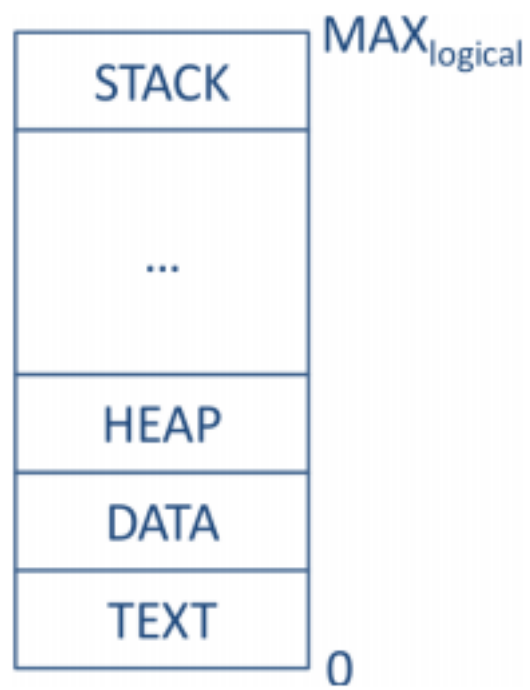


Figure 3: Representation of a process in memory

different processes. To achieve this every time a new process is created it is considered a “New” process and as soon as it is ready to be ran it will go into a ready state. The OS will look for processes in a ready state and if there are any it will block the current process(This is done with an interrupt which can also be caused by things such as waiting for input or a file read) for X amount of time and then run the ready process. The blocked process will stay blocked for a certain time to allow other process to be ran. The OS will keep performing that until every program finishes going into an Exit state. This behaviour is explained on the following graph.

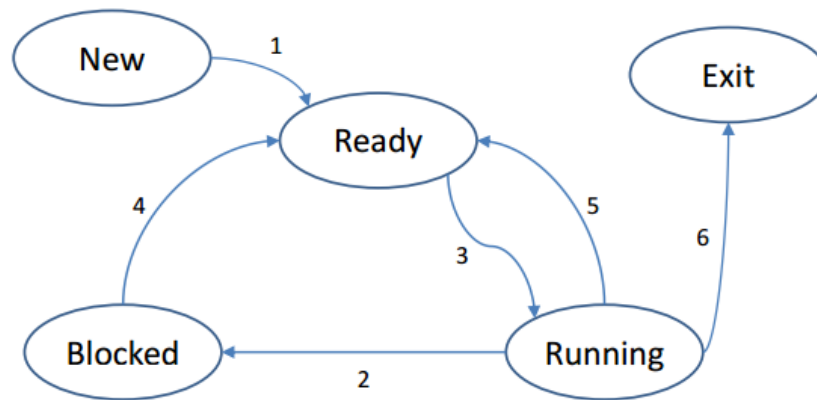


Figure 4: Representation of process states

Context switching

This occurs when the Operating System switches between multiple process in order to generate parallelism. However true parallelism can only occur when you have multiple processors. The actual context switch is the time taken by the CPU to save the current state of the process and switch to the following process. If a context switch is not done properly a process can lose information in registers and corrupt its algorithm.

The important maths

To do context switching we must limit the amount of time the CPU takes on each process. Because context switching has usually a rather static amount of time on the CPU we can define a long time slice for our processes or a short time slice.

- A short time slice means that processes will all run closer together. However because more CS(context switching) is happening the overall time taken by the CPU to run all processes will be longer
- A longer time slice means that process will have a lower response time as more time is taken by each process making the Operating system less(concurrent) but due to the smaller amount of CS the overall time for all processes to be done will be shorter.

This right here is why we did that whole COURSEWORK! smaller time slices reduce response time while higher ones reduce turnaround time(Ofc if we implemented a static time for CS) So yeah it was useless

Program control block

The program control block is what takes care of interrupting processes to allow for CS. These are kernel data structures and contain information that can be used by the OS such as:

- Process id(PID, UID, Parent ID)
- Process control information(State for scheduling)
- Process state information(Registers PC Stack pointer aka all the stuff saved from the CS)

Because these control low level features of the OS they can only be accessed on kernel mode(logical kernel data structure)

OS abstraction

To allow CS and scheduling to work properly the OS stores lots of information such as:

- Process tables(Process control blocks)
- Memory tables(Where logical memory is)
- I/O tables(Availability and status of all devices)
- File tables(File system information)

All this information shouldn't be accessed by the user or the programmer it is all used for abstraction.

Using the abstraction

To actually take advantage of lower level features such as multi threading we can make user mode calls through certain OS dependent libraries to use these functions.

- POSIX(linux library)
 - fork() - Unix

- clone() -Linux
- WIN32 API (shit OS library)
 - NtCreateProcess() -Windows(They can't even name their functions properly)

Because the OS will keep switching between multiple processes those need to be terminated with calls such as:

- exit(), kill() - Unix/linux
- TerminateProcess() - Windows(See they are so bad)

YEEY Finally some code

Here is an example of how to create a multiprocess program on Linux.(Because on windows this is probably like 1000 lines of useless code)

```
#include <stdio.h>

void main()
{
    int iStatus;
    int iPID = fork();
    if(iPID < 0)
    {
        printf("fork error\n");
    }
    else if(iPID == 0)
    {
        printf("hello from child process\n");
        execl("/bin/ls", "ls", "-l", 0);
    }
    else if(iPID > 0)
    {
        waitpid(iPID, &iStatus, 0);
        printf("hello from parent process\n");
    }
}
```

The code can be found on the code directory

Processes 2 - Scheduling

In order for an OS to run different processes in a parallel manner(true or not) we need to have some sort of schedulers which prevents the OS of for example repeat the same process over and over again while other processes need to be ran. However most of the times we also want certain processes to have priority over others. For example a mobile phone OS might prioritise the processes which

manage phone calls over third party games as the purpose of the device is to do phone calls.

Process schedulers

Process schedulers can come in different types. Those can be categorized in their Time horizon:

- Long term - Applies to new processes, focus on long processes and avoids CS when efficient and controls parallelism. This is common on computer clusters for high performance tasks. Although these types of schedulers focus on getting one task done at a time it will change between processes to achieve an efficient mix of CPU and I/O bound processes and maximise the usage of both. This focus less on responsiveness and multi-tasking and more on efficient usage of resources.
- Medium term - Controls swapping and focus on a fair split of resources, sometimes delaying some processes and working with smaller subsets of all processes.(Common on modern OSs in conjunction with short term schedulers)
- Short term - This runs a lot more often then the previous one's it is also closely linked to clock interrupts and I/O interrupts to avoid busy waiting(Running a process that cannot do something because it requires something that is not yet available). This is also responsible for organising the ready queue.

Exam question “Where do the process schedulers fit in with the state transitions?” Short term is extremely important for I/O and clock interrupts

Don't get confused systems have all of these events schedulers that run at the same time. A single event scheduler most of the times is not efficient you need to make quick decisions at ms intervals but at the same time you need to focus on efficiency. The only unusual type of scheduler in this list is Long term as most common day tasks don't require that amount of efficiency but rather improve response time.

Pre-emptive schedulers

Schedulers can be differed as well in pre-emptive and non-preemptive schedulers.

A pre-emptive scheduler is a scheduler that has a clock interrupt, that simply means that if a process is takes X amount of time and has not being interrupted then it will stop it to allow other processes to run. This is the most common type of scheduler in modern operating systems.

A non-preemptive scheduler is different as a process will only stop if it voluntary says so to the CPU. For example while a process is waiting for a file to be read

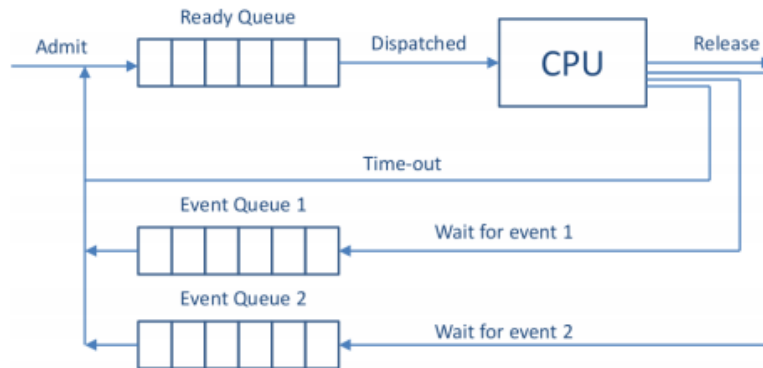


Figure 5: Queues in an OS, Understand how different schedulers manage different parts of this diagram

than it will stop and allow other processes to run.(I/O Interrupt). This is only used on really old Operating systems such as Windows 3.1

Performance assessment

Because we have so many types of schedulers we need some sort of performance measurement so we can benchmark different schedulers. The most common criteria for schedulers are:

Learn this its ez marks its literally averaging out.

- Response time: When a process starts(usually the average is used for comparison)
- Turnaround time: When a process finishes(usually the average)
- Predictability: This makes sure processes are running at roughly the same amount of time between CS.

Some relations that are obvious is that smaller time slices decrease response time while bigger time slices(clock interrupts) will decrease turnaround times considering the CS time.

Algorithms

Here are a list of popular algorithms for schedulers.

- FCFS (First come First serve)
- Shortest job First
- Round robin

- Priority queue

FCFS

This is fine as it is considered *fair*, processes created first will get done first. However small processes that could be done a lot before without affecting bigger jobs too much are left behind.

Shortest job First

This focus on finishing jobs quickly and reducing the average turnaround time. However it is not *fair* as a long job started first will only finish after every other small process is done. Even if they are created a long time after the first job. In theory a long process might never finish if new short processes keep being added to the CPU (*starvation* is the fancy name for this).

Round robin

This algorithm is pretty simple it basically means every job is ran for a small amount of time until all processes are finished. Time slices are important to improve Response time and Turnaround. > Time is not wasted here if a process finish earlier the next one starts.

Priority Queues

This is a round robin implementation where processes have priorities, higher priorities get executed first while shared priorities are round robin to the end. If priorities are set right for CPU and I/O bound this will yield the best overall results

(Processes 2)[<http://moodle.nottingham.ac.uk/pluginfile.php/2720824/course/section/2720824/Processes2.pdf>]