Ian Kenny

January 11, 2023

5CCS2OSC Operating Systems and Concurrency

Processes, scheduling, introduction to threads

In this lecture

In this lecture we will consider:

- Single-tasking and multitasking
- Programs and processes
- CPU scheduling issues and approaches
- Thread basics

A very brief history

For most of computing history, computer systems had one CPU. This meant that they could only perform one task at a time.

Even more significantly, these early computer systems could only run one program at a time, and that program could only do one thing at a time.

This meant that to switch between programs the user would have to shut one down completely and start another.

This is clearly not a very satisfactory state of affairs.

Multitasking

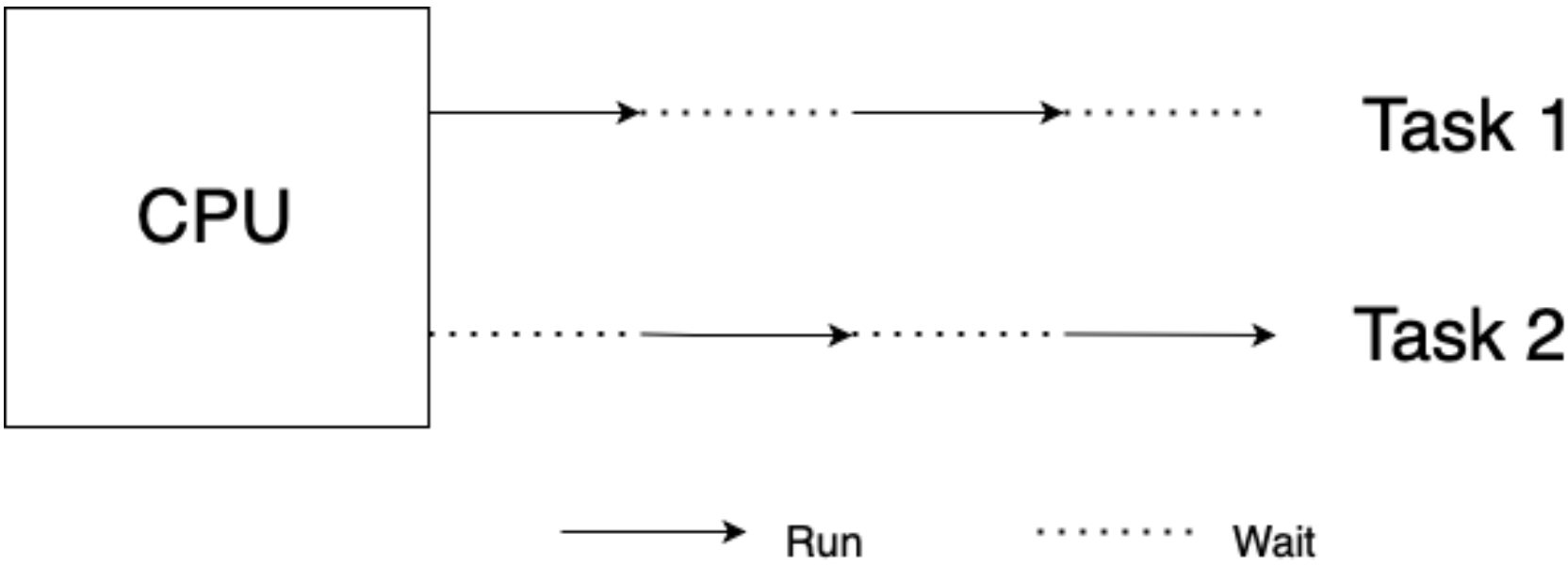
The solution to this at the time was to introduce *multitasking*.

With multitasking the CPU is able to switch between multiple tasks, allowing each of them to execute for a period of time. The CPU was still able to execute only one task at a time but could create the illusion that it was executing multiple tasks at the same time by doing so quickly and frequently.

This is the notion of *concurrency*: multiple tasks being able to make progress.

In the figure on the next page, a single CPU is running two tasks, allowing each to execute (and wait) for a fixed time period.

Multitasking



Types of task

As we shall see, however, simply allocating each task a period of time is not necessarily the best way of *scheduling* the tasks. Indeed, even in the early days other approaches were used.

Some tasks are *CPU-bound* and some are *Input/Output-bound* (I/O-bound). Some tasks use the CPU and I/O devices in roughly equal measures.

A task that is CPU-bound spends the majority of its time executing code on the CPU.

A task that is I/O-bound spends a lot of time using I/O devices, i.e. storage, networks, printers, etc.

Types of task

In the early days of computing I/O devices were exceptionally slow[1]. Indeed, relative to CPU speeds, they still are.

Without some form of multitasking, if a task initiates an interaction with an I/O device (e.g. reading from a disk drive), the CPU would have to sit idle until that interaction was complete. This is clearly unsatisfactory.

Some early multitasking approaches were able to detect the initiation of I/O and then switch to another task until the I/O was complete (this requires, e.g., a disk controller and a buffer able to hold the data until the CPU is ready for it).

[1] E.g. computer games loaded from cassette tape...

What *are* tasks?

So far we have been using the word ‘task’ to mean some activity that the CPU is undertaking.

We will now be more specific about what it is that the CPU is actually running, and how that relates to multitasking.

Programs

If you ask somebody what a computer does, they will probably tell you it "runs programs".

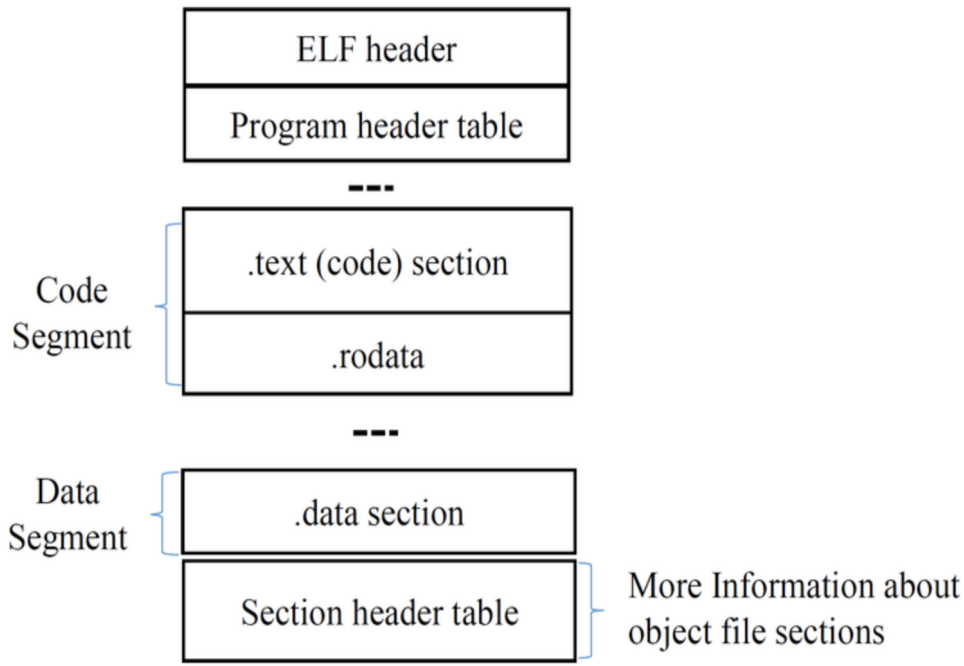
This is true but doesn't give us the full picture. We can say that Google Chrome is a program, but how many programs are we running if we start Google Chrome twice?

We can define a *program* as the contents of an *executable file* of some sort. But this is simply a file. A computer has to *load* the contents of the file into its memory before it can execute the program.

So, we want to define a *program in execution* as distinct from a program in a file.

Executable files

The figure below shows a simplified format for the ELF executable file used by Unix-based systems[2]. We see that the file contains the program instructions ('text') and constants (rodata) (in the code segment) and the data (in the data segment). Data here means the *global* data defined in the program.



[2]This format is used for some games consoles and other devices too. Windows and MacOS have their own formats.

Processes

When the operating system loads a program to run it creates what we can call a *process*.

A process is an *instance* of a program that is being executed by the computer.

If we start the same program multiple times then we will get multiple processes for the same program.

Earlier in this lecture we talked about ‘tasks’. We can now talk about processes instead, when referring to a program that is being executed by the computer.

Processes

When a computer system runs a process it must create certain areas in the RAM.

The code (text) and data segments from the executable file must be copied into the RAM.

The OS also allocates two memory segments for each process: the *stack* and the *heap*.

The stack is used for local variables, method parameters, etc.

The heap is used for storing *objects* (in Java).

The stack and the heap

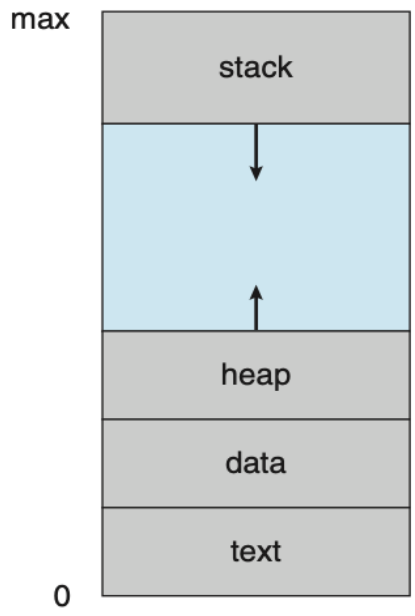
The code and data in a program are 'static', meaning (in this context) that the memory requirements for those sections don't change.

However, until the program is running as a process we do not know what its overall memory usage will be. Each time the program is executed it might be used differently; some methods might be called in one run but not in another, etc.

Local variables, method parameters and objects created by `new` or `malloc` *do not exist* in an executable file. They only exist when they are *created* and they are not created until a program is running as a process (and maybe not even then - some methods might never be called, for example).

Thus, the stack and the heap are created by the OS to allow a process to dynamically create the necessary variables, parameters and objects.

Process memory layout



This figure shows the layout of a process in RAM. The memory addresses start at 0 and end at 'max'. The arrows indicate that the stack and the heap grow towards each other in the RAM. As stated above, both of these structures are *dynamic* - they grow and shrink[3]

Image source: *Operating System Concepts*. A. Silberschatz, G. Gagne, and P. Galvin. Wiley.

[3] The OS will make sure they never overlap.

Process memory layout

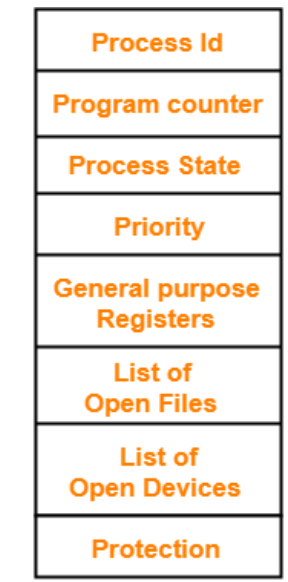
So, for each process that the OS has created, there will be a block of memory, as shown on the previous slide.

If two processes relate to the same program, i.e., they have the same code and data segments, then the OS might share these across multiple processes. The stack and heap are not shared.

The OS will manage that area of the RAM for each process that is running.

Process Control Block (PCB)

As stated above, the OS must keep track of the processes it starts. The structure in which the OS keeps data related to each process is called the *Process Control Block* (PCB). The details vary for each operating system but they all largely track the same things reflected in the figure below.



Process Control Block (PCB)

Image source: https://www.gatevidyalay.com/process-control-block-process-attributes/

Process Control Block (PCB)

Some notable entries in this structure are as follows:

- Process Id: each process has a unique ID number.
- Program counter: the address of the next instruction for the process.
- Process state: whether the process is READY, WAITING, etc.
- General purpose registers: the contents of the CPU registers for the process.

PCBs are usually stored by the OS using a linked list.

Process State

The state-transition diagram below shows the states that a process can be in and the transitions between them.

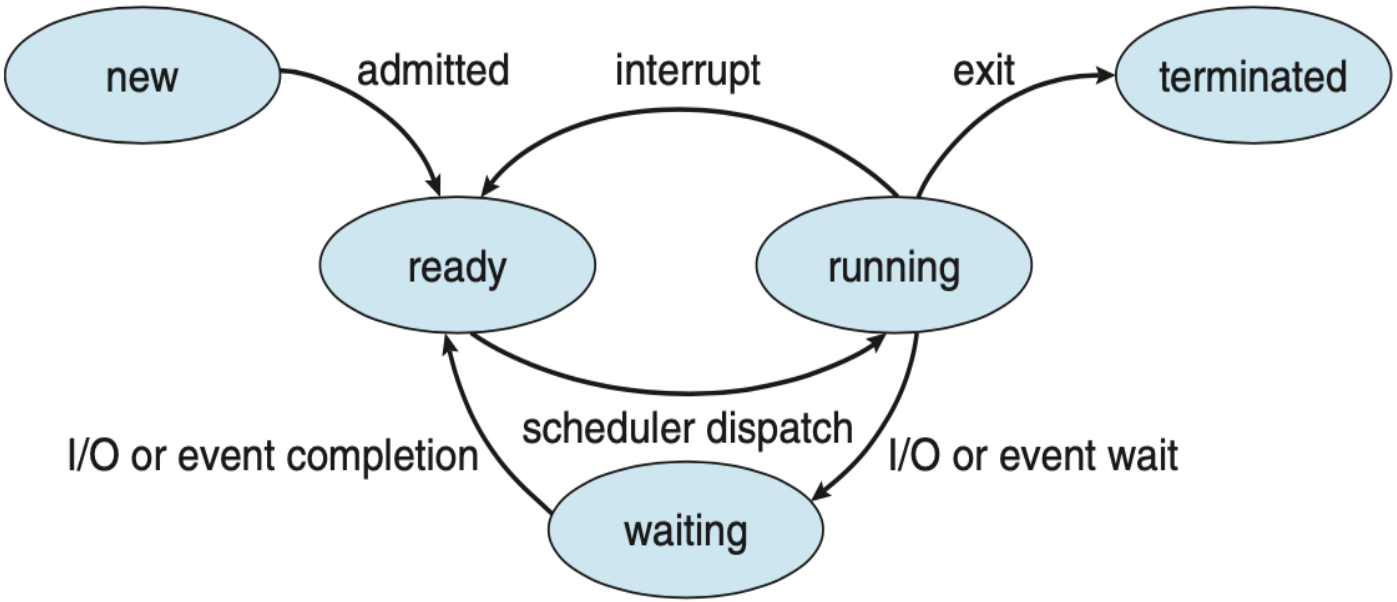


Image source: *Operating System Concepts*. A. Silberschatz, G. Gagne, and P. Galvin. Wiley.

Process State

When a process is first created its state is **new**. At this point the OS will make a decision about whether to *admit* the new process. It might be forced to wait if the system is already fully committed.

Once a process has been admitted it will transition to the **ready** state. This means it will wait to be executed.

When selected to execute via *scheduler dispatch*, the process will transition to the **running** state.

Process State

When a process is **running** it can be *interrupted* by the OS. This means the OS needs to run another process. The process that was interrupted will go back in the ready queue.

When **running**, a process might also issue an *I/O request* of some kind. If that is the case it will transition to the **waiting** state. Once the I/O request has been completed, the process will be put back in the ready queue.

When the process *exits* it will transition to the **terminated** state.

Scheduling processes

In a modern computer system, there will always be many processes running simultaneously. The user will start a number of processes (word processor, development environment (IDE), email client, browser, etc.), and the operating system itself will start some, possibly many, processes. If you open a 'task manager' of some kind on your own computer you will see this.

The question arises, therefore, how does the operating system decide which process(es) should be running at any given moment?

These decisions are made by *schedulers*. There are a number of schedulers. For example, the long-term scheduler selects which processes will be loaded from external storage into the RAM.

We will focus on *CPU scheduling*. The CPU scheduler makes the decisions about which of the processes in RAM will run.

Context switching

Before we consider some of these algorithms, we consider what happens when a CPU changes from running one process to running a different one.

When a process is running on the CPU, the value of the program counter (which holds the address of the next instruction) and the CPU's registers contain values that are specific to that process.

If a process has been interrupted so that another process can run, these CPU register values must be saved in the PCB so that they can be restored next time the process runs.

The next process that is being allowed to run may have its own values of these registers that need to be restored too (if it's not running for the first time).

Context switching

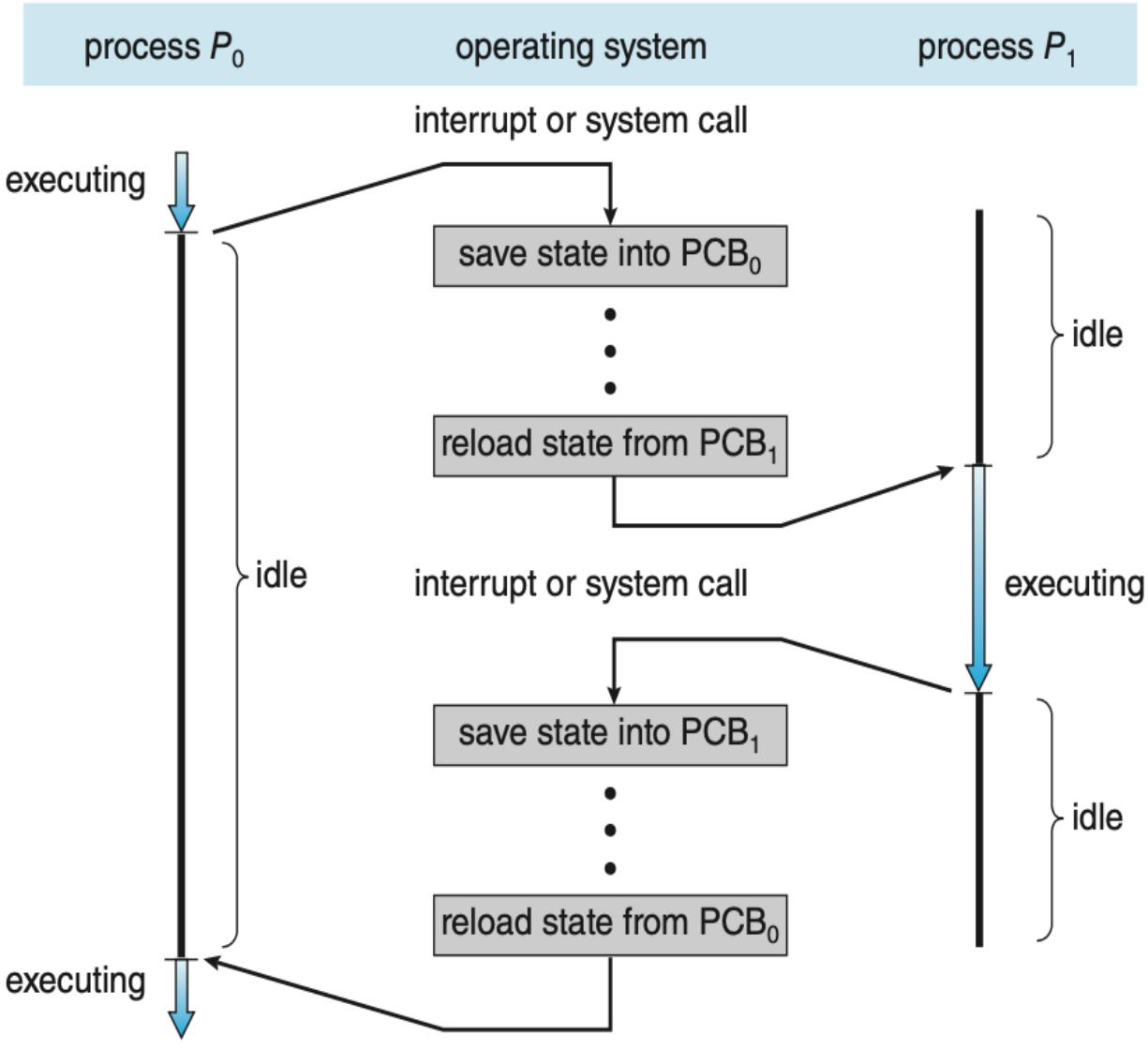


Image source: *Operating System Concepts*. A. Silberschatz, G. Gagne, and P. Galvin. Wiley.

Context switching

Clearly we want context switching to happen as quickly as possible otherwise it would be a dominant factor in the overall timeliness of the CPU's operations. Modern OSs and CPUs can perform a context switch in about 1-2 μs.

Scheduling algorithms

CPU scheduling algorithms select processes to execute. How they make these decisions depends on the overall scheduling scheme and the type of process 'favoured' by the algorithm.

In older operating systems, processes were expected to *yield* to each other: a form of *cooperative multitasking*. Older Windows and Mac operating systems used this approach. If a process did not yield, it would never relinquish the CPU. This did not require a scheduler.

As systems started to run more and more processes, classes of algorithms developed based on the ideas of *preemptive multitasking* and *non-preemptive* multitasking.

Scheduling algorithms

With **preemptive multitasking**, the OS will interrupt a running process to allow another process to run.

With **non-preemptive multitasking**, a process will control the CPU until it terminates or enters a waiting state.

Modern scheduling algorithms are usually preemptive.

We will now look at some of these algorithms.

First come, first served (FCFS)

The most basic scheduling algorithm simply operates a regular *first in, first out* (FIFO) queue. FCFS is *non-preemptive*.

The process that has been in the queue the longest is selected to run.

We can analyse the performance of scheduling algorithms to check if they are meeting the needs of a system.

Average waiting time (AWT)

The waiting time of a process is the amount of time spent in the ready queue.

The **average waiting time** of the processes in a queue is the average of all weighting times. Typically we want to minimise average waiting time.

FCFS AWT: example

Consider four processes P1, P2, P3, P4 and their associated burst times[4]. The two tables below show the processes arriving in the queue in two different orders. 'AWT' = average waiting time.

Process	Burst time (ms)	Wait time (ms)
P1	27	0
P2	4	27
P3	6	31
P4	5	37
AWT		24
P2	4	0
P4	5	4
P3	6	9
P1	27	15
AWT		7

We can see that the order of arrival of the processes could have a big impact on the average waiting time.

[4] In the context of FCFS, the burst time is the total time it takes to execute the process (technically, excluding I/O time).

Shortest Job First (SJF)

Instead of allocating 'priority' to the processes based on the order in which they arrive, we can take into account some process metrics, e.g., the burst time of the process. Doing this we can choose the job that will be completed soonest as the next one to execute.

This algorithm can be shown to minimise average waiting time.

The major problem with this algorithm is estimating the burst time. Solving this requires data to be kept on the previous burst times of a process (if available) and the computation of some sort of average burst time. We didn't have this problem with FCFS because that algorithm doesn't care about burst times.

SJF AWT: example

Consider the same four processes P1, P2, P3, P4 being scheduled using SJF. In this example, the processes are all assumed to arrive at the same time.

Process	Burst time (ms)	Wait time (ms)
P2	4	0
P4	5	4
P3	6	9
P1	27	15
		7

We have seen that this is potentially a lower average waiting time than for FCFS (FCFS can only achieve this AWT if the processes arrive in this exact order).

Shortest Job First (SJF)

SJF can be preemptive or non-preemptive.

Non-preemptive SJF will, of course, allow each process to complete or enter a wait state before another process is allowed to run.

With preemptive SJF, when a new process arrives in the ready queue, the scheduler will compare the (estimated) remaining burst time of the currently-running process and the (estimated) burst time of the newly-arrived process, and if the latter is lower the existing process will be preempted.

Starvation

With SJF it could be the case that so many short jobs arrive that a longer job does not get the chance to execute. This is called **starvation** and should be avoided.

To avoid starvation an **age** factor can be brought in to the assessment of each process. The longer a process stays in the queue the higher its age gets. This can then be factored into the choice to ensure that, eventually, even long jobs will get executed.

This means that we are affording a **priority** value to each process that is somehow a combination of its burst time and its age.

This approach can be generalised to create a *priority scheduler*.

Priority Scheduling

With a more generalised priority scheduling algorithm we can consider burst time to be only one possible factor in the priority of a process. We can build a function that determines the priority of all processes based upon a whole set of factors.

These priority factors can be internally or externally defined.

Internal factors could include: burst time, use of I/O devices, memory requirements etc.

External factors could include: who owns the machines, who submitted the jobs, etc.

Priority Scheduling AWT: example

Consider the processes P1, P2, P3, P4, P5 being scheduled using Priority Scheduling. Two different allocations of priorities are shown. Again, the processes are assumed to have arrived all at the same time.

Process	Burst time (ms)	Priority	Wait time
P1	10	3	27
P2	15	4	37
P3	2	2	25
P4	25	1	0
P5	6	5	52
AWT			28.2
Process	Burst time (ms)	Priority	Wait time
P1	10	2	6
P2	15	4	41
P3	2	5	56
P4	25	3	16
P5	6	1	0
AWT			23.8

Priority Scheduling

Again, priority scheduling can be preemptive or non-preemptive.

With non-preemptive priority scheduling, if a new, higher-priority process arrives it must wait until the currently-running process has terminated or entered a wait state.

With preemptive priority scheduling the existing process would be interrupted.

Round-Robin Scheduling

The round-robin algorithm operates in a similar way to FCFS but has preemption.

We again use a FIFO queue to hold the processes, but this time the queue is treated as circular. This means that, while a process is not terminated or waiting, it will remain in the queue. Once it has been to the front of the queue it will be returned to the back.

For this algorithm, a time quantum is specified. The time quantum is the maximum burst time allowed for a process in any one run. The time quantum does not typically vary per process.

Round-Robin Scheduling

There are two possibilities.

If a running process signals it is finished, or has entered a wait state, before its time quantum has expired, then it is preempted for the next process in the queue.

Otherwise, if the process *reaches* its maximum time quantum, it is preempted (unless no other process is in the queue).

Round-Robin: example

Consider a round robin algorithm in which the quantum is set to 2ms, and the following set of processes in the ready queue in the following order.

Process	Burst time (ms)
P1	5
P2	9
P3	2
P4	1
P5	6
P6	11

Round-Robin: example

The execution times will be as follows.

Process	P1	P2	P3	P4	P5	P6	P1	P2	P5	P6	P1	P2	P5	P6	P2	P6	P2	P6	P6
Time	2	2	2	1	2	2	2	2	2	2	1	2	2	2	2	2	1	2	1

The waiting time for a given process is the time that it spends not executing. This is essentially the sum of the execution times of the processes between its runs.

We can see that, by definition, the round-robin algorithm is preemptive.

Multilevel Queue Scheduling

Modern computer systems run many different processes which can be classified by the systems they belong to. For example, time-critical system processes, UI processes, background processes, etc.

It may not be sensible to put all of these processes in the same queue, dealt with by the same scheduling algorithm. Priority values can be used but there are more flexible methods.

Multilevel queue scheduling involves multiple queues, each containing a particular class of process, arranged in a priority order. So, the highest priority processes are placed in the top level queue, and the lowest in the bottom queue.

These individual queues can also be organised by priority and each can use a different scheduling algorithm.

In this model, processes do not move between queues.

Multilevel Feedback Queue Scheduling

An even more flexible model is to allow processes to move between queues.

With this model the processes are allocated to queues according to their burst time characteristics, and moved between queues on the same basis.

For example, processes that take up a lot of CPU time might be moved to lower priority queues.

Current operating systems

Both Linux and Windows use preemptive priority scheduling.

MacOS appears to use a multilevel feedback queue.

Introduction to Threads

We have seen that in order for users to be able to run multiple programs at the same time (and multiple instances of the same program), operating systems introduced the concept of the *process*. Multiple processes can be loaded into RAM at the same time, and the CPU scheduler allows each of them to execute according to some scheduling algorithm.

This means that the user can, for example, run a word processor, email client, music player, browser, etc. 'at the same time'.

But what about these individual processes? They also need to do more than one thing at a time.

Introduction to Threads

Think about the word processor.

When the user is typing document they might wish to check the spelling, repaginate the document, save the document, etc.

Without some way for the process to do these tasks ‘at the same time’ the application would stop completely while saving, spell checking, repaginating, etc.

None of these tasks could be done ‘in the background’ while the user continues to type.

Introduction to Threads

In order to allow each process to appear to do more than one thing at a time, the concept of the *thread* was introduced.

Conceptually, a thread is a strand of execution *within* a process. A process may have many threads executing 'at the same time'.

Each process can start threads. Threads can also start threads. In the Java language, for example, there is a `Thread` class and a `Runnable` interface that can be used to create new threads.

Introduction to Threads

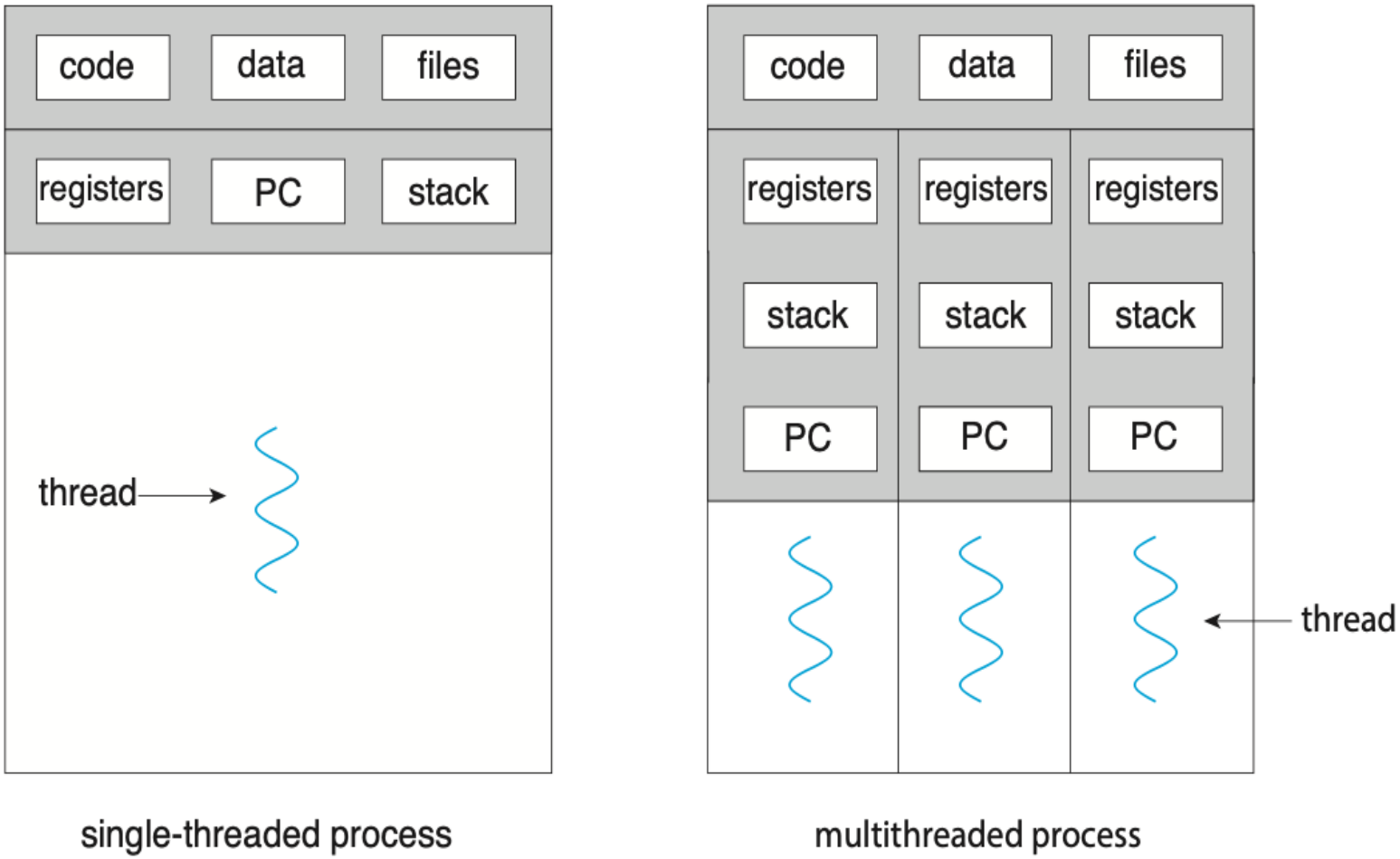


Image source: *Operating System Concepts*. A. Silberschatz, G. Gagne, and P. Galvin. Wiley.

Introduction to Threads

The typical case is that a process creates a thread to handle some subtask of its overall task.

For example, a web server might start a new thread to handle each new connection from a browser. User interfaces are also a classic example of the threaded approach. An application can put its UI implementation into one or more threads so that it remains responsive even when the application is performing some heavy duty task.

From the word processor example, the application might start a thread to do the spellchecking, saving the file, repagination, etc.

Introduction to Threads

Processes are largely independent of each other. They do not share memory, unless using a shared memory model for *inter-process communication* (another model is the message passing interface).

When a process creates a thread, however, the thread shares the address space (exists in the same memory block) of the process. This applies to all of the threads created under that process. In practice this means they all access the same heap - the heap allocated to the process - and the globals, etc.

However, each thread gets its own stack. This is because each thread will do different things, call different methods, etc. So it's easier to just give each thread a stack.

It is this sharing of data that causes the class of problems related to *synchronisation* that we will talk about in the coming weeks.

Introduction to Threads

In the coming weeks we are going to look in detail about the use of threads in Java.

For the rest of this lecture we will consider the relationship between processes and threads, but first we will think about CPUs.

CPUs

Consider the single CPU system. As we already discussed, such a design can perform only one task at a time. The idea of processes was introduced to enable the system to have multiple tasks in the RAM and then, using a scheduler, switch between them to give the appearance that the tasks were being performed simultaneously. Only one task was ever being performed at any one instant, however.

If threads are introduced to this model, the fact remains that only one thing at a time can actually happen. Within the running process, the threads would have to take turns to perform whatever task it is they are implementing.

Multicore processors

However, in recent years the trend in CPU design has been towards getting performance increases by increasing the number of CPUs in a processor rather than continually trying to ramp up the clock speed.

These processors are called multicore processors. They essentially have 2, 4. 8, etc. *cores* per processor. A core is a CPU. All cores can execute in parallel - literally at the same time.

This changes the picture with processes (and threads) a lot. The OS and the CPU are now able to schedule multiple processes to literally execute at the same time by allocating them to different cores. And just as a single core CPU can handle multiple processes 'simultaneously' by scheduling them, each core in a multicore processor is able to do that.

This potentially increases the *throughput* of processes considerably.

User threads and kernel threads

When a user (programmer) creates threads (including in Java, and when using the pthreads library), these are known as *user threads*.

Threads that are owned by the kernel (i.e. the core operating system) are known as *kernel threads*.

The operating system will (usually) map user threads to kernel threads for execution in a *one-to-one* fashion. Threads can also be mapped many-to-one and many-to-many but these mappings are less common.

Scheduling

In the kernel, the OS actually schedules threads not processes. This might seem odd but since the kernel maps user threads to kernel threads, and kernal threads are the basic unit of execution for the OS, this makes sense.

This also applies to Java programs that create threads. Java threads are actually managed as kernel threads by the OS.

Next week

Next week we will start to look at Java threads in detail, and consider the Java Memory Model.