Operating system managing the CPU

How does an OS deal with signals that interrupt the fetch decode execute cycle?

The purpose of the CPU is to process data by fetching, decoding and executing instructions from the main memory. However, peripherals or software may require attention and so need to signal the CPU (known as interrupts). This is accomplished by two different methods:

A. Polling

A repeated cycle where the CPU **checks the status of a device**. If it requires attention, then the next address of the program counter is not run and instead the **instruction of the peripheral** that requires execution is dealt with. Otherwise, the fetch, decode, execute continues as normal (and another check-up is called at the end).

This is inefficient because repeated cycles of polling when there is no error and so no attention requirement for multiple peripherals, is unnecessary and wastes processing time.

Fetch → Decode → Execute → Send signal to peripheral → Act on received signal
(THEN REPEAT)

B. Interrupts

A device sends a signal on **the control bus** to the CPU to indicate that it requires attention. The CPU will then respond accordingly and **does not have to continually check** up on the peripherals.

This means that a CPU actually has an extra stage in the operation cycle:

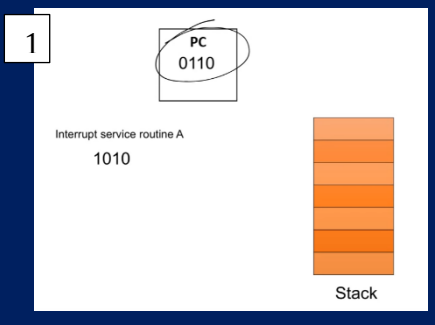
Fetch → Decode → Execute → Interrupts (if control signals are received)
(THEN REPEAT)

The CPU checks for interrupts before it begins to fetch the next instruction.
The CPU is designed to stop what it is doing and react to events once they occur to service the requirement of the interrupt.

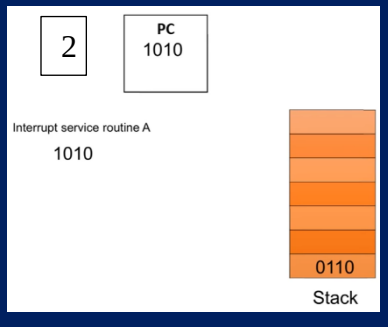
Interrupt service routine (ISR) = a **software routine** in an operating system or device driver whose execution is invoked by the reception of an interrupt.

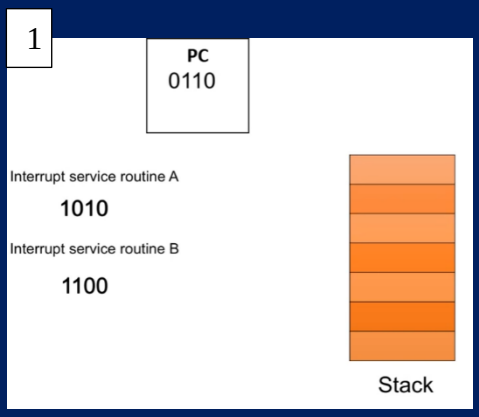
Stacks are employed

ISR's create a problem: the address for the next instruction stored in the program counter can't just be overwritten by the first instruction of the ISR. The program that was previously being executed must be continued after the ISR is complete.
This is solved by using a stack. The stack momentarily sores the program counter's instruction address until the ISR has been completed. Multiple interrupts may occur and so the stack has a hierarchy



The PC holds the next instruction address. Interrupt service routine A needs to be ran. The value of the PC is stored in the stack until the ISR has been completed. The value in the stack is then moved back into the PC so the CPU knows where to begin processing from again.

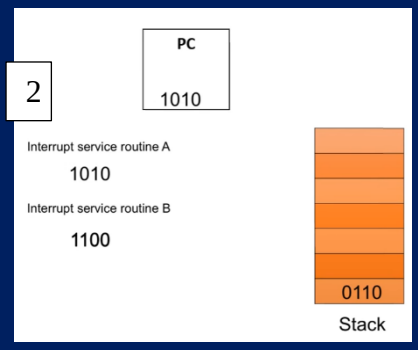




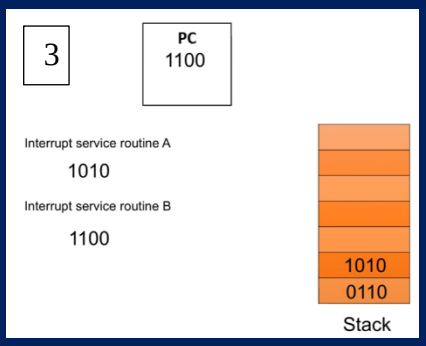
Take this example, The program is interrupted by ISR A, so the PC value is placed in the stack while ISR A's instructions are acted upon.

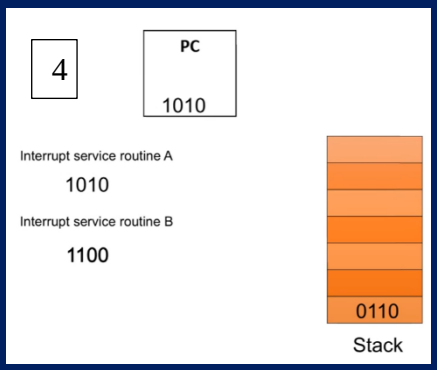
If another interrupt occurs, ISR B, with a higher priority - this in itself must interrupt ISR A . Don't forget that ISR A could consist of multiple instructions…..

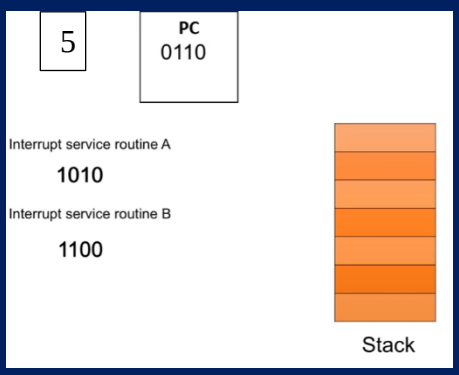
The value in the PC of ISR A is copied into the stack (along with any other proceeding memory address values) and ISR B is placed in the PC so acted upon. Once complete, as the stack has a hierarchy, the instruction it left off in ISR A is moved to the PC. Finally the original instruction address is copied back to the PC. Once ISR A is complete.



This will not always happen. Interrupt service routines (ISRs) are usually given their own priorities. It may well be that a new interrupt has a lower priority than the interrupt currently being executed so only once the current ISR has been completed will the next ISR be ran.







Interrupt hierarchy

Interrupts will always have a higher priority than normal programs being executed otherwise the interrupt service routines would never be called.

e.g. If the computer crashes and the CPU is stuck in a situation it can't resolve, the cycle must be interrupted by the user with a higher priority task such as running task manager/shutdown.exe.

Types of interrupts

Hardware
Power/reset buttons (calling shutdown or reset ISR)
Memory parity error (corruption data when transferred to the memory)
Hardware failure encountered.
Hard drive has retrieved the requested data.

Software
Illegal instruction (an instruction that is not mentioned in the CPUs opcode/instruction set, they commonly crash the computer)
Arithmetic overflow
New log-on request

Input/output I/O
Buffer nearly full
Signal the completion of a data transfer
Flushing a buffer (this empties/ writes all the data stored within the buffer to the storage once the data within it exceeds a certain limit.).
A peripheral requires more data (e.g. printer)
Buffer is empty

A buffer = a region of a physical memory storage used to temporarily store data while it is being moved from one place to another.

Hard drives also contain a buffer; by utilising a buffer the CPU can send a block of data to be saved without having to wait for the data to be physically stored onto the disk.
Buffers are used to write data to storage in larger sections rather than continually writing small pieces of data that would degrade performance. When the buffer is empty an interrupt is sent to the CPU to request for more data.

Buffers are also used for peripherals. When a key on a keyboard is pressed, an interrupt is sent to the CPU and the keystroke is stored in the keyboard's buffer. However, sometimes the OS may not be ready to deal with the keyboard stroke and more keys may have been pressed. Instead of losing keystrokes, the buffer stores the values until the OS is ready.

If the buffer overflows then the motherboard warns the user by sounding a beep.

As ISRs are part of the device driver, the OS is effectively using the driver to process the interrupt.

The scheduler

A scheduler is a utility program that arranges jobs or a computer's operations into an appropriate sequence.

Processor scheduling – managing the activity of the processes by the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy

A computer has the appearance of carrying out tasks simultaneously but the CPU can only fetch//decode/execute one instruction at a time. The scheduler is responsible for making sure the processor time is used as efficiently as possible.

A process is a piece of software that is currently being managed by the scheduler inside the OS.

What is the difference between scheduling and interrupting?
Interrupts are hardware signals sent to the CPU to request processing time. They are external to the CPU and are software/driver generated. Interrupts are initially handled by the CPU before being passed over to the OS.
This differs from scheduling, which makes pre-emptive decisions

Pre-emptive decisions are decisions on which process should be ran next or which ones should be interrupted.

The main objectives of a scheduler are:

- Maximise **throughput**
- Be fair to all users on a **multi-user system**
- Provide **acceptable response time** to all users
- Ensure hardware resources are **kept as busy as possible**
- **Avoid deadlock** (processes waiting for each other) and starvation (when a process never gets any CPU time to complete in a timely manner, owing to a higher-priority task always getting an advantage over it.

Algorithm 1 - Round robin (Pre-emptive)

This approach makes special use of a form of queue (ready-to-run queue) called FIFO. Processes are despatched on a first in first out basis (FIFO).

Each process in turn being given a limited amount of CPU time called a time slice or a quantum. The operating system sets an interrupting clock or interval timer to generate interrupts at specific times. This method helps guarantee all users of the system get a reasonable response time and fair amount of processing time.

Even if a process does not complete during its time slice, the dispatcher gives the CPU to the next process and the previous process is put to the back of the FIFO queue (ready-to-run queue).
I.e. the **process is pre-empted** if it has not completed before the end of its time slice.

The first process to enter the queue will be the first process taken out of the queue for processing. This means the processing is fair to all jobs.

If every job is more or less equally important then this algorithm is optimal. The main advantage of round robin is that it is very simple to implement, owing to the fact it does not consider priorities.

All processes are given an equal time slice, which limits the chance for starvation.

On the downside, Round Robin does not take into account the priority of jobs so every job has equal CPU time or is processed until complete. It is based on arrival time sequence of the processes.

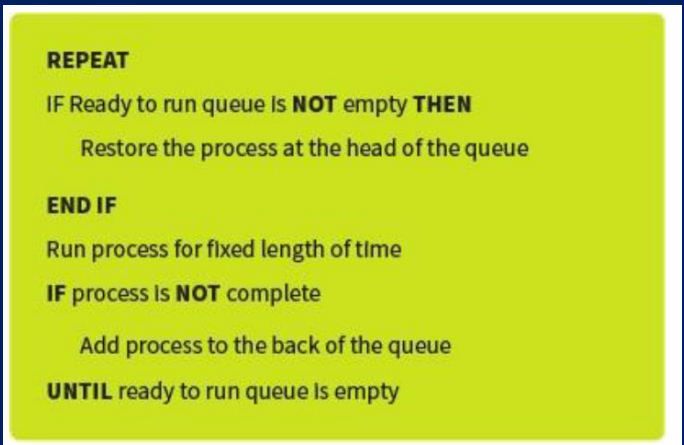


Figure 3.11 Round robin pseudocode.

Round robin is not suited to applications that have an interactive interface since the lag time between doing something on screen and the process being able to handle it would be noticeable. There is a delay between he time taken to swap processes. Round robin is best suited to situations where only a few simple processes are running at any given time.

Algorithm 2 – First come first served (Not Pre-emptive)

Jobs are processed in the order in which they arrive, with no system of priorities. If every job is more or less equally important then this algorithm is optimal and simple to implement.

Algorithm 3 – Shortest job first (SJF) (Not Pre-emptive)

Processes are sorted in the ready queue according to the estimated processor time needed.

This relies on the knowledge of how long a process requires, in order to complete its task. Most processes are unable to supply this information, so it is up to the OS to analyse historical data and statistical information on the process being run. Most burst processes are quite predictable. The algorithm is similar to round robin only the processes are not pre-empted after a set amount of time, but rather are allowed to complete.

Processes are completed in the order within the ready-to-run queue. SJF operates on the basis of where a process is placed in the ready-to-run queue initially. Shorter jobs are always added to the front, while longer ones go at the back.

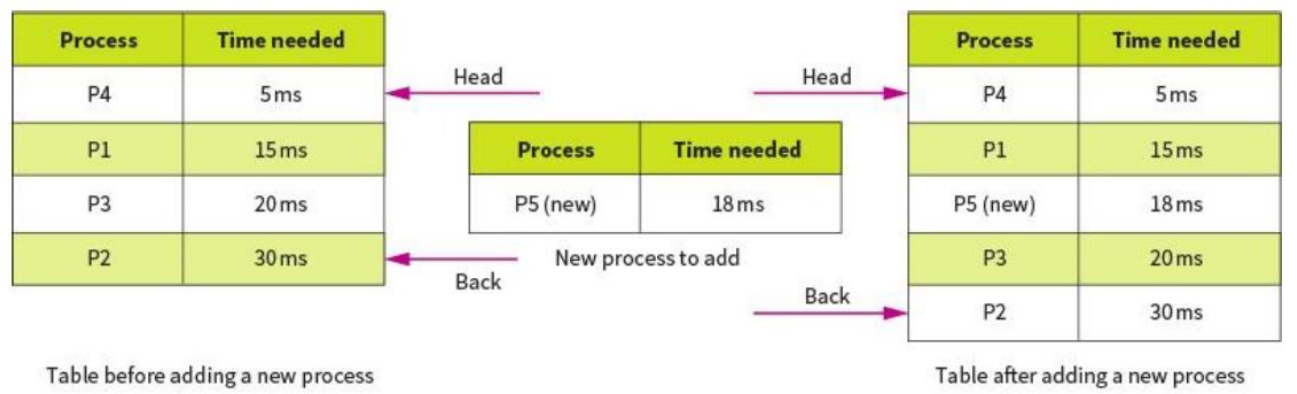


Figure 3.12 Running a new process.

SJF ensures that user interface based processes have higher priorities – cutting down their waiting time - than more processor-intensive tasks (simple mouse clicks and keyboard strokes require smaller time slices).

Reducing average waiting time increases responsiveness and throughput.

SJF does not have to swap processes as often as round robin.

Drawbacks

SJF requires more processing to perform each swap due to the statistical analysis.
SJF is quite complex as if requires manipulating queues so processes are added into the correct places.
SJF can starve longer processes as they have lower priority compared to short processes that `jump' ahead of them in the queue.

Algorithm 4 – Shortest remaining time (Pre-emptive)

Processes are ordered in a queue so that the process with the shortest estimated time to completion is next to run.
The SRT algorithm is very similar to SJF, with the biggest difference being that when a process is interrupted and pre-empted, the time-remaining on that process is looked at and then the process is added ahead of any other processes that have a long time left to run.

As time progresses, processes that require a lot of CPU time will slowly get a chance to run and the time they will require will reduce. This increases their priority and increases their chances of running, so reduces starvation.
A key design is that the queue must be in the shortest time order at all times, not just when a process is added.

This tends to reduce number of waiting jobs, and the number of small jobs behind big jobs.
However, it requires advanced knowledge of how long a job will take.
This is easier for batch jobs that may run overnight (scientific/commercial jobs that are run regularly).

SRT has advantages over SJF if the OS is actively pre-empting longer-running processes.

- The processor is allocated to the job closest to completion but it can be pre-empted by a newer ready job with shorter time to completion.

Algorithm 5 – Multi-level feedback queues (Pre-emptive)

These algorithms are designed to:

- Give preference to short jobs
- Give preference to I/O bound processes
- Separate processes into categories based on their need for the processor

The algorithm implements several job queues and jobs may move between the queues that best suit their needs and depending on how much processor time they need.

This does not have a ready-to-run queue but instead relies on three queues of differing priority. Level 0 (high) > level 1 (medium) > level 2 (low). Each queue works on a first come first serve basis. However, the head of level 0 is always run before level 1 and level 2.

The aim is to maximise throughput. For example, due to bottleneck of transfer speeds, it is more efficient to keep the I/O devices as continuously busy as possible. When programs try to simultaneously send data to e.g. the printer a bottleneck does not occur.

MFQ implements a promotion and demotion system to prevent starvation:

- New processes are always added at the tail of level 0.
- If a process gives up processing time on its own accord then it will be pre-empted to the back of the same level queue.
- If a process is pre-empted/stopped by the scheduler then it will be denoted to a lower queue.
- If a process blocks for I/O, it will be promoted one level.

Each process is given a fixed time slice. If a process uses the full amount of time, it will be stopped by the scheduler. This is known as pre-emptive scheduling and for MFQ it will result in the process being demoted, unless it is already in level 2.

A process that requires pre-emptive scheduling is not considered to be well behaved and could end up monopolising the CPU if allowed to stay at level 0. This could be a CPU-heavy task or one that is poorly programmed.

Well-designed programs give up time to ensure overall system liveliness. It will be pre-empted back into the same level queue.

Blocked process, waiting for I/O devices are always promoted since:

1. They will already have to wait a certain amount of time before they become unblocked anyway – it makes little sense making them wait any longer.

2. It ensures I/O devices are used to maximum efficiency, a process must be ready to send more data as soon as an interrupt has occurred.
3. Sending data to an I/O device is not a CPU-intensive task so has only a small-time-slice.

For these reasons, promoting blocked processes does not negatively impact other processes and would increase the throughput of other processes.

First come first serve (FCFS)	Shortest job first (SJF)	Shortest remaining time (SRT)	Round robin (RR)	Multi-level feedback queues (MFQ)
The first job to enter the ready queue will be the first to enter the running state.	Jobs are sorted in the ready queue according to the estimated time needed. There is no pre-empting so each job will be run to completion. Uses statistical analysis and historical data to estimate processing time.	The ready queue is sorted on the estimated time to complete the process. Processes that arrive having a shorter time to completion are moved higher up in the queue. The queue is always in the shortest time order, not just when a process is added like in SJF. Reduces starvation as slowly the more CPU intensive tasks will be move up the queue.	Each process is given a maximum length of processor time (a time slice) in the running state after which it is put back into the ready queue. All jobs are treated fairly. Starvation is greatly reduced. Noticeable latency on UI applications due to swap times and lack of priority.	Several ready queues are used, each with different scheduling algorithms. Jobs are able to move between queues as their priorities change
Not pre-emptive	Not pre-emptive	Pre-emptive	Pre-emptive	Pre-emptive
First come first serve is fair in the terms of processing jobs that arrive but long processes can cause other processes to wait	SJF is better alternative to first come first serve as it maximises throughput but it relies on being able to calculate estimated processing time so is more complex. Can also lead to starvation.	SRT maximises throughput but longer jobs may take longer time to process when short jobs keep arriving in the queue.	Round robin is the fairest of them all but can be inefficient, as all process given same priority of processing time.	MFQ is the best overall approach which modern operating systems use. It can utilise advantages of many different queues with different scheduling techniques. Tasks move to different queues depending on their priorities.
Few simple processes running at any one time.	User-interface based applications. Responsiveness is key.	SRT has advantages over SJF if the OS is actively pre-empting longer-running processes.	Few simple processes running at any one time.	Can prioritise (promote) and demote certain tasks between queues. All tasks are also given equal time slices. New tasks added to the back of level 0 to reduce starvation.

Comparing the algorithms

Pre-emptive means that the operating system carries out some criteria to decide how long to allocate to any one task before giving another task a turn to use the operating system.
The act of taking control of the operating system from one task and giving it to another is called pre-empting.