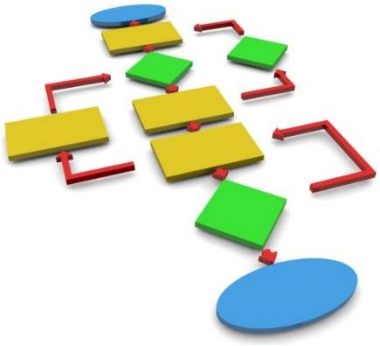
Lecture 2 – Process Management Concepts

We will begin today highlighting the terminology we associate with computations being hosted by an operating system. We will explain how computations are initiated and executed, how they are represented and accounted for within the operating system and the general lifecycle of their behaviour. We will then explain the mechanism through which hardware devices can communicate their management needs, and how software programs can communicate with the operating system to make system calls to use the various resources of the computer.

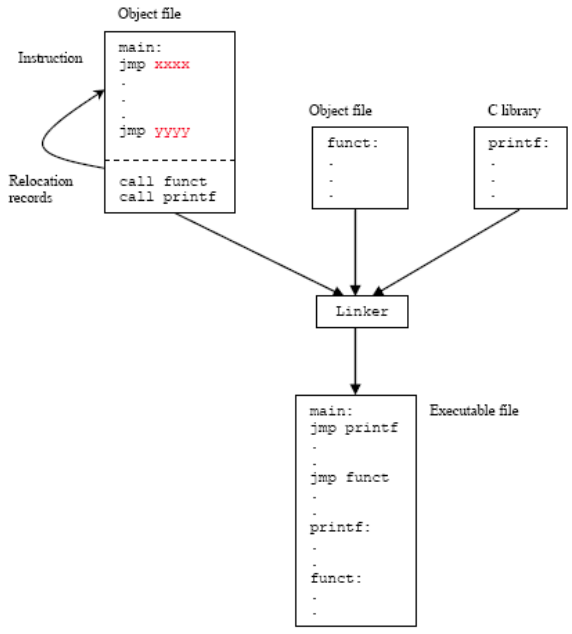


1. Programs, Processes, Processors

A **program** is a collection of instructions specifying a defined sequence of execution. It is the translation of an algorithm into a programming language. We use programming languages so that the specification of our algorithm is more specific and deterministic within a set of rules that can be translated to processor instructions. Algorithms expressed in natural language may be more ambiguous.

A **compiler** maps the program code into a series of machine instructions for a particular processor and these are stored in an object file. (.obj)

Many functions, commonly required by applications, may already be compiled into **libraries** which can be quickly added and used by a program to speed up the development time, so that the wheel does not have to be reinvented each time you want to make a cart, so to speak.



So once the object file has been created for your program code by the compiler, it must be linked with the object code of any libraries which implement functions that are referenced/used by it.

A **linker** programme brings together all of the object code from your program and libraries to create an **executable** file (.exe), a file which is formatted internally in a specifc way to enable the operating system to understand its content for loading it into a new process for execution.

The diagram demonstrates the procedure of creating an executable file from a collection of compiled program objects.

A **process** is an instance of a program in action. It is an operating system abstraction, essentially a data structure used to manage a processing activity. When the instructions associated with our program are actually being carried out, a process exists.

When we ask the operating system to execute a program, by either typing a command or clicking with the mouse, the operating system creates a new process. The executable file is loaded as a binary image into the main memory and the processor fetches instructions from it from a designated starting point, e.g. the first instruction of the main() function.

The executable binary file containing the program image has a format and content specific to a particular operating system and processor, it won't work on other systems. It includes any program parameters, stack space, data space and program code using the instruction set for a specific processor family.

A process is an execution context, a collection of kernel managed information needed while it is running.

Processes are dynamic entities whose lifetimes range from a few milliseconds to maybe months.

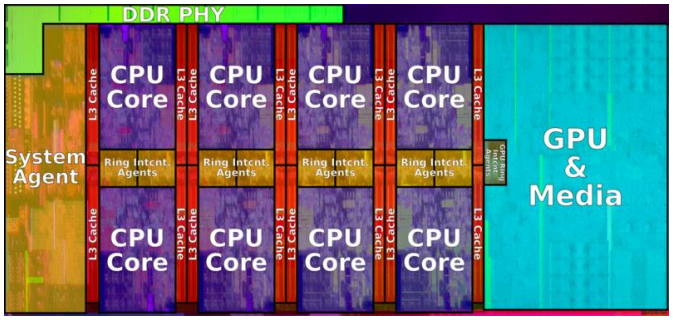
Processes may be persistent, implementing system services, i.e. they remain active in the system indefinitely.

The operating system itself may be made up of several processes managing different services such as the process manager, the memory manager, the file manager and so on.

A **processor** (CPU) is the hardware agent which runs a process by executing the instructions stored in the memory image loaded from the binary executable file.

The operating system code first sets up the processor with the information needed to start the process, i.e. where to begin fetching instructions and other data associated with its running state.

In most machines, there is only one processor as they are a relatively expensive component and interconnecting more than one to share other resources is more complex and expensive.



However, modern processors have multiple execution "cores" and are capable of executing the instructions of more than one process simultaneously.

Processor Time Sharing of Ready Processes

As there are far more processes than processors in a system, processes share the processor, by having the CPU's time divided amongst the ready processes when they have instructions to be executed. This gives the appearance that they are all progressing simultaneously, although more slowly perhaps than if there were not other processes also being executed.

Blocked Processes

Processes are not always ready to use the processor as sometimes they must wait for other devices that are perhaps supplying data to them or they must wait for resources that need to be assigned, or are already in use, or perhaps they must wait for user interaction or time periods that must occur before they can continue. These processes wait in queues and are not 'ready' and not competing for the CPU.

2. Representing and Accounting for Processes

A fundamental task of an operating system is process management – Creating, Controlling, Terminating executions – Managing the Execution Environment.

The operating system must allocate resources such as memory space and cpu time to processes, it must protect resources from activities of other processes, and provide communication and synchronisation mechanisms among processes that share the same resources or cooperate.

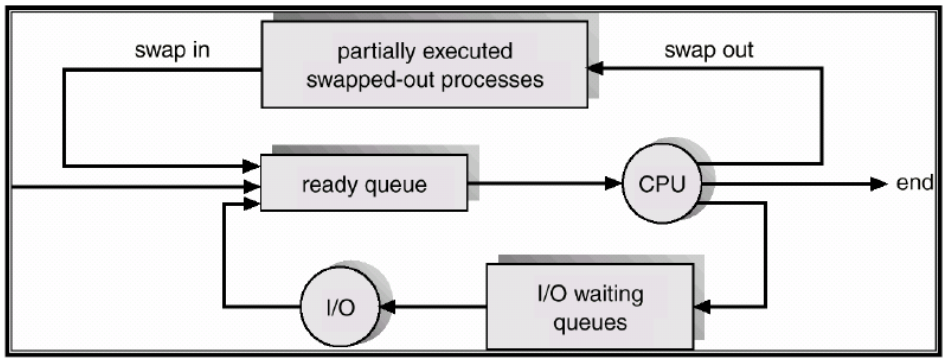
To achieve this fundamental task, like any program, an operating system uses data structures to represent the state of objects it is creating and dealing with. For each process, the operating system maintains a data structure known as a process control block (PCB) or process descriptor to keep a clear picture of what each process is doing, what point it has reached in its execution and what resources have been assigned to it.

A Process Control Block is used to keep track of the execution context (all resource information about the process and its activity) that can be used for independent scheduling of that process onto any available processor.

The image gives a basic idea of the category and type of information that is typically stored in a process control block (PCB)/process descriptor.

Process Identification Data	**Processor State Data**	**Process Control Data**
Unique process identifier	CPU Register State,	Flags, signals and messages.
Owner's user identifier	Pointers to stack and memory space	Pointers to other processes in the same queue
Group identifier	Priority	Parent and Child linkage pointers
	Scheduling Parameters	Access permissions to I/O Objects
	Events awaited	Accounting Information

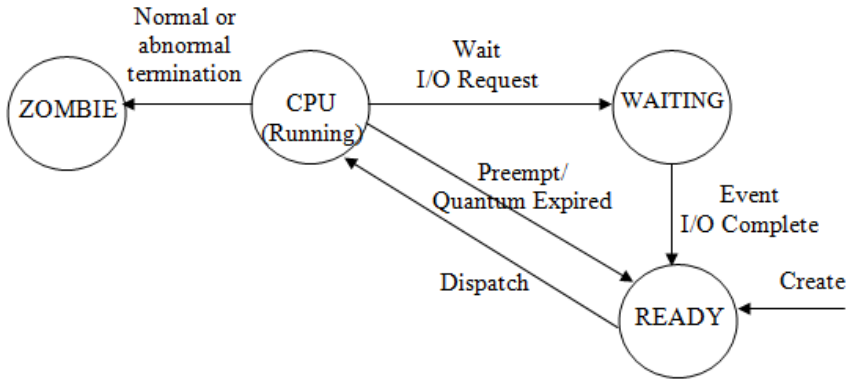
Queues are used to hold processes that are waiting for different resources. These queues are serviced using scheduling algorithms. A link to the Process Control Block of a process represents the process on one of the OS queues.



The PCB may be moved between different queues over the process lifetime depending on the priority or state of its execution.

Only processes that are ready to execute are held in the 'ready' queue and compete for the CPU.

3. General Lifecyle of a Process



Simplified Process State Transition Diagram

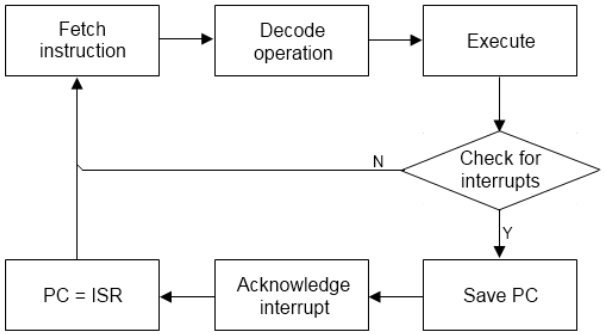
There will be periods where it needs the CPU to execute its instructions and there will be other periods where it is waiting on various other system resources, such as input/output devices, to provide data so that it can continue its execution.

A process is created and initially in the 'ready' state to execute. At some point it will be chosen from the ready queue and dispatched to the processor, where it starts executing, known as the 'running' state. After some time executing, it may complete or the operating system may decide to preempt its execution, place it back in the ready queue, and give the CPU to another process. That cycle continues until execution is complete. Sometimes when a process is running, its instructions may involve interaction with input/output devices. The process can then be moved from the running state to the waiting queue associated with that I/O device. When the I/O event has completed, the process is returned to the ready queue. At the end of its computation, a process descriptor may remain in the system as a 'zombie' data structure until its results have been collected by another program or until it is destroyed by the operating system.

4. Hardware and Software Communication with the Operating Systems

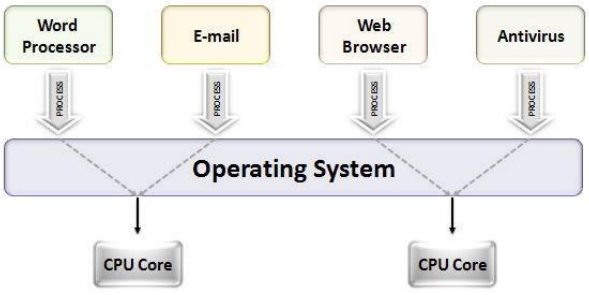
When a hardware device requires attention from the Operating System it generates an electrical signal known as a Hardware Interrupt. For example, it may have received data or completed an I/O task. The use of hardware interrupts is much more efficient for managing large numbers of hardware devices. The alternative is for the CPU to periodically poll these devices, i.e. repeatedly asking them all in turn if they need attention. The interrupt mechanism means the CPU can focus instead on executing processes and the hardware can work in parallel until a hardware interrupt signal occurs.

Usually a support chip/system, an interrupt controller, is used to manage, field and identify interrupts and send the details of devices needing attention to the processor. This system can prioritise the order in which the processor might respond to various interrupts received.



The image shows the stages involved in executing an instruction within a processor. An instruction is first fetched from some memory location within a process as indicated by the program counter (PC) register, the instruction is decoded and identified and any operands it might need are also fetched. When all the data is onboard the processor, the instruction can then execute and write any resulting outputs to registers or memory before going on to fetch the next instruction. This execution sequence must be **indivisible** for correct operation, i.e. it must execute in its entirety or not at all. So it is only at the end of the sequence, after the outputs of the instruction are written, that the processor will check for interrupt signals that might have been generated during the instruction cycle.

If an interrupt is detected, then the location of the next instruction for the current process is saved temporarily on the process's stack and the memory location of the code that deals with servicing interrupts (ISR) is placed in the Program Counter (PC) register instead. This is where the processor will fetch its next instruction from, so an interrupt mechanism is an automatic system that causes the CPU to stop executing the current process and start executing the interrupt service routine when the current process instruction is complete.



A Hardware Interrupt mechanism is also very important for implementing a **multitasking environment** where lots of processes compete to use the CPU.

When a task is to be scheduled to use the processor for a set amount of time, a hardware clock timer is initialized. When the time expires an interrupt signal triggers the interrupt handler which invokes the operating system scheduler to select another process. This scheme prevents one process from hogging the CPU indefinitely. It is a preemptive mechanism to give the operating system control of what the processor is executing.

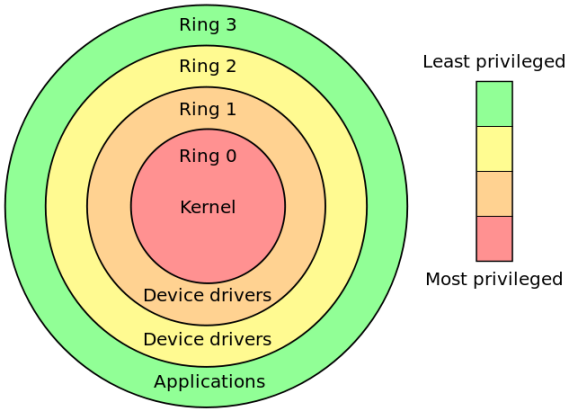
So operating system code is triggered by interrupts. The kernel is **event driven**, it executes only in response to an event as that is the mechanism for switching the CPU away from the current process. There are three categories of event that can trigger the CPU to switch to the operating system code.

Hardware Interrupts raised by hardware devices that can occur at any time.

Software Interrupts (Traps) that are generated by program instructions in order to invoke OS functionality.

Exceptions: Faults or Errors that occur during the execution of an instruction, such as dividing by 0 or a missing memory page/page fault. The operating system can fix temporary faults like that but exception errors will cause a process to be terminated.

Can the code of an ordinary process access the hardware directly, bypassing the operating system? How is that prevented?
The answer to this must be no, because then the operating system would have no control over the system.
In order to enforce hardware protection, certain processor instructions are restricted and cannot be executed by ordinary user processes.

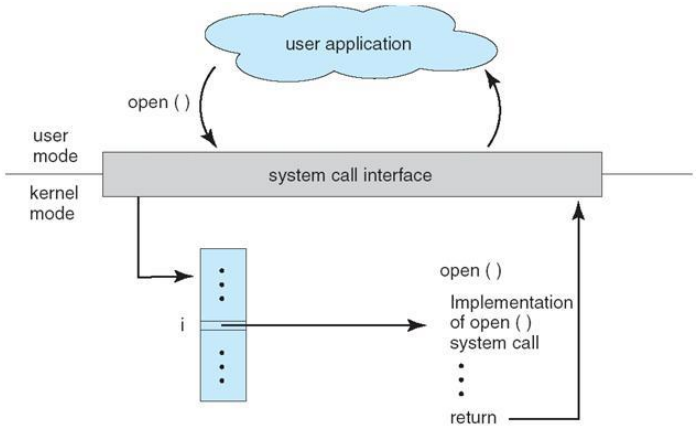


The processor must be capable of executing in at least two privilege modes, User Mode or Supervisor Mode. When executing a user process the processer is in the least privileged User Mode, and can only execute a subset of its instruction set. To execute operating system code, the processor must be switched to a more privileged Supervisor Mode in a controlled manner and can then execute its full instruction set.

When an interrupt or exception event is detected, the processor switches its mode automatically to a higher privilege level to execute operating system code and then the operating system code switches the processor back to user mode when handing back control to a user process. The mechanism to elevate privilege is automatic and outside of user control.

System Calls
Processes need to communicate with the operating system too in order to obtain protected system services like accessing the hard disk, keyboard, mouse or other hardware, creating new processes, doing interprocess communication or configuring kernel services.

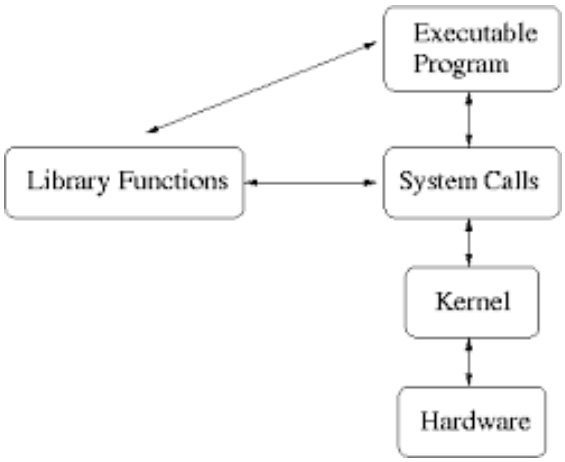
Switching to Supervisor Mode
A special processor instruction known as a **software interrupt** is the mechanism for doing this.



When a processor encounters a software interrupt instruction as part of the process's code, it essentially traps the process execution. The processor will stop executing the current process, and save its CPU state in its PCB structure stack space.

It then proceeds to treat the software interrupt like a hardware interrupt. It fetches the next instruction from a designated area of memory where the operating system has placed an interrupt handler, which in turn then leads to execution of the system call required. The execution of the system call could be postponed to a queue and another process selected to run on the processor instead, so it is important to realise when a process makes a system call, it is voluntarily giving up its current slice of CPU time. When the interrupt routine eventually completes, a callback may be sent to the interrupt handler to restore that process to the ready state so it can continue its execution.

This mechanism ensures that **access and entry to the critical operating system code and hardware is done in a controlled way at a designated entry point** automatically determined by the processor. Once a user process executes a software interrupt, when making a system call, code execution is not under user program control anymore. The operating system is in control at all times of what is executed when the CPU is in privileged mode.



When writing application programs, System Calls are often accessed through wrapper libraries linked with the user program at compile time.

The wrapper function checks the correct parameters are used and will invoke the software interrupt in the appropriate way. The wrapper function's role is to map the kernel functions to an API for the specific language your program is written in.

A wrapper API for system calls serves to help application code portability across different kernel and language implementations.