Operating Systems

An operating system is a program, called the **kernel**/**monitor**, that:

- **Manages resources**.
- Enables **application programmers to access resources** through a programming interface.

Criteria for Good Operating System Design

Application programmers will choose an operating system which suits them. The criteria for this are:

- **Efficient & Lightweight:** OS doesn't use too many CPU cycles.
- **Flexible:** Flexible resources.
- **Ease of Use:** Design doesn't obstruct the programmer.
- **Security:** Prevents accidental and malicious damage. *E.g. downloads which could damage the OS.*
- **Stability:** Efficient error handling to prevent crashes and errors.
- **Features:** Contains features suited to the task they're dealing with.

Resource Management

There are two categories of resources:

Hardware	Software
CPU, memory, disk, video, keyboard, printer, etc.	Controls hardware by running programs through the CPU.

Need for Management

Resources need managing because they are:

- **Limited:** not enough to go around. All computers have limitations.
- Need **protection**.
- Need to meet certain criteria

Programmers are responsible for **developing operating systems which manage resources effectively**. *E.g. Phones have strict memory, energy & CPU limitations which the OS needs to manage.*

Resource Protections

Resources need to be protected. This can be achieved by maintaining:

- **Security:** Prevent programs and users from accidentally or intentionally corrupting another program or its data.
- **Access Rights:** Ensure certain resources are only available to authorized programs.

Criteria for Good Resource Management

- **Responsiveness:** Ensures a program runs fast. E.g. processing network packets as they arrive.
- **Real Time:** Dealing with certain events in a small, fixed amount of time. E.g. controlling wing flaps.
- **Security:** Prevent accidental and malicious access or modification.

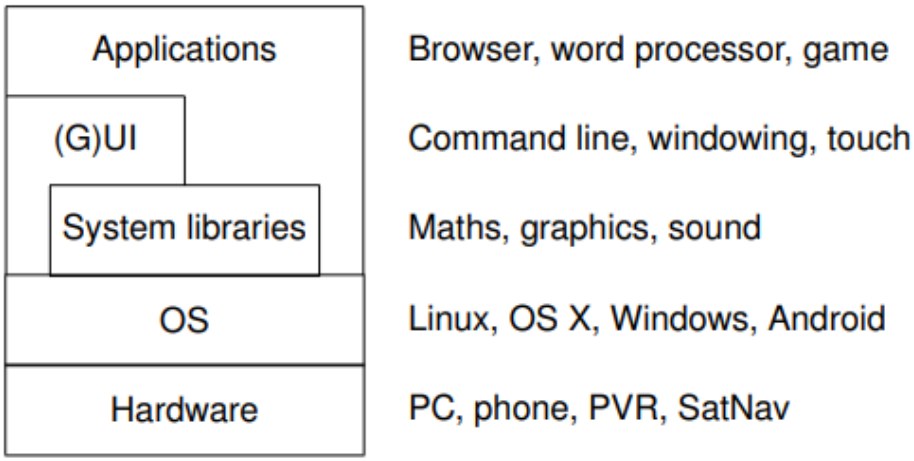
The Programming Interface

Operating systems must implement a **programming interface.** These must have a level of **portability** and **abstraction** because they need to be separate from the hardware. The programmer shouldn't need to be concerned with things like interrupts, network performance, etc.

- Operating systems **implement hardware functionality themselves** at a high-quality level and provide programmers with an **interface** to it. This **removes the need for programmers to do it themselves**.
- The OS programmer is more experienced so can avoid common pitfalls, hence getting **higher performance from hardware** than we could.

Degrees of Abstraction

Engineers working at a given level of abstraction shouldn't need to consider the levels below it. **The lower levels dictate how the higher levels are formed.** *For example, the resources the software is provided with are dictated by the operating system.*



Keeping these separate is vital for several reasons:

- It **promotes efficient development of each layer**. *E.g. If the UI is tied to the OS this prevents development of better OSs.*
- Enables switching out each level. *E.g. Changing the GUI but running the same OS and vice versa.*
- Failure to separate them can **lead to bugs in one layer impacting other layers**.
- Failure to separate them could **allow attackers to circumvent the OS's security**. *E.g. A browser gaining control of the machine.*

Types of Software

You can run all these software types on Intel hardware. There are others, but these are the most popular on the market. Some operating systems build upon pre-established OSs or alter them. Linux and Mac software are two such Unix derivatives!

Windows Software	Mac Software	Linux Software
• GUI is tied to the OS.	• Until recently, it could be run on Intel hardware. • Run on Linux. • GUI is tied to the OS.	• Runs all kinds of hardware such as Intel mainframes, phones, TVs, etc. • GUI is not tied to the OS so users can choose a UI.

History of Operating Systems

No Operating Systems

Around the 60s, programmers had to write **machine-specific** programs, this meant **no portability** and lots of **repeated code**. They rarely even saw the computer:

1. The program and data needed (collectively called a job) would be prepared on paper tape or punched card.
2. Jobs are given to operators who load and run them, collect the results and send them back to the programmer.
3. If a bug occurred, the programmer improves the program and the process repeats

Computer time was limited due to the **length of this process.** Scheduling was initially done by hand.

Programs & Libraries

Libraries & programs were developed to **make programming easier** by **simplifying repetitive tasks**:

- **System Libraries:** Collect common functions. *E.g. sqrt, open file, etc.*
- **Loaders:** Program management.
- **Debuggers.**
- **Hardware Interfacing:** I/O drivers.

This made programming and program management easier, but there was still lots of human intervention needed.

Automated Spooling

Operators would load multiple programs onto magnetic tape allowing the computer to load and run them **as fast as the hardware allowed**. This is called **spooling**.

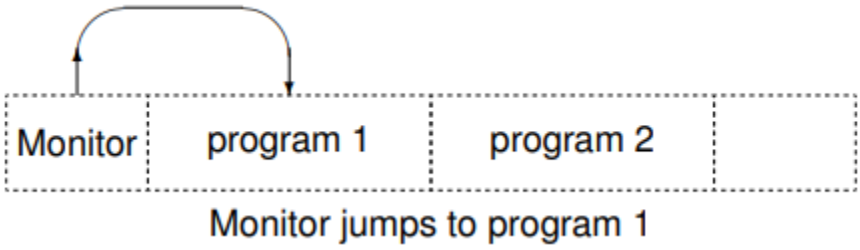
- Output was **also written to a magnetic tape** which could be attached to a printer. This was needed because printers are slower than computers.
- **Monitors/Supervisors**, which automatically loaded programs from tape, ran and output them, were introduced. These were directed by **job control languages (JCLs).**
- JCL allowed **several programs to be loaded together enabling batch processing** increasing efficiency by allowing **more time for running programs and less dealing with overheads** (loading and unloading).
- We study the **cooperative** approach. It's called this because programs need to cooperate by sending control back to the monitor when they finish.

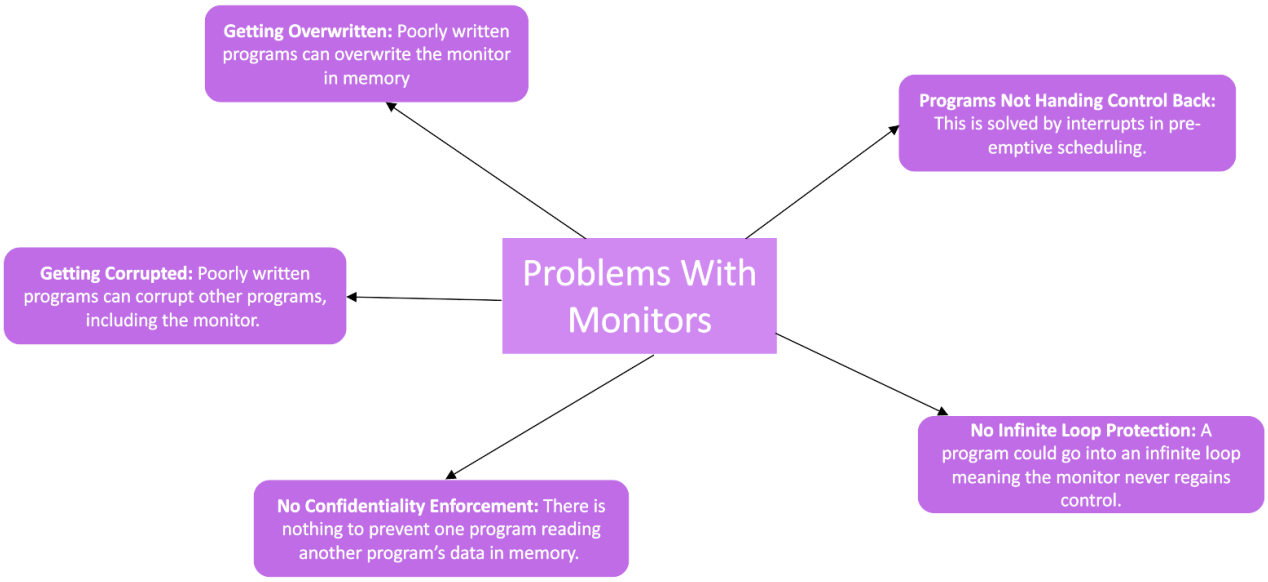
Modern computers use these same processes because the need to run jobs and manage resources is still there!

More About Monitors

A monitor operates as follows:

1. Loads one or more applications into memory.
2. Begin executing an application - the monitor code jumps to the program code.
3. When the program finishes or when a peripheral is needed, it will jump back to the monitor.
4. The monitor may then deal with the peripheral and execute another program while waiting for it. Or it begins executing the next program.





Monitors Enabling Batch Processing

Batch Processing: The process of loading multiple jobs into memory at once and executing these automatically.

- **Monitors enable efficient batch processing** using **multitasking** - switching to the program which needs the CPU the most at a given time, **scheduling**.
- Each program is given a **time-slice** which is its execution time.
- While one program waits for a **slower peripheral,** another program can run.
- Hence, the monitor needs to **decide which program to run next.**

For example, if a program is doing something with light CPU usage, like writing to a tape, the monitor can tell the computer to run another program in the meanwhile.

*Note: There is a **single stream of control**. The monitor doesn't run at the same time as programs, it simply uses a jump to give control to the program.*

Scheduling

Automatic monitors require a scheduling algorithm!
(*See the scheduling section later for more details!*)

Interrupts, Pre-emptive Scheduling & Timesharing

Interrupts are needed to **prevent runaway programs**! This approach is called **pre-emptive scheduling,** and it **enables timesharing**.

Timesharing: Several programs share CPU time so appear to be executing simultaneously.

1. A hardware clock/timer sends an interrupt after a period.
2. The interrupt service routine jumps to the OS.

Handling Interrupts

The OS can then choose to do one of the following:

- Resume the interrupted program – if its time slice has not ended yet.
- Kill the program if it has used its allotted resources.
- Run another program.

Peripherals & Interrupts

Peripherals also transfer control to the OS during their ISRs. They may do so while waiting for data as part of multi-tasking. When the data is detected, an interrupt occurs allowing the OS to take over. The OS then runs the program to deal with the data. **This balances the speed of input and the computer.**

Other Bits Worth Remembering

- Interrupts enable **terminals.** When a user makes a keystroke in the terminal, the current process is interrupted, the OS takes over and runs the appropriate program to deal with the keystroke.
- Computer-intensive programs will get a large time slice so will likely just resume after most interrupts.
- Interactive programs don't need much CPU so will get a small time slice and will likely deschedule the program after a few interrupts.

Hardware Protection Mechanisms

The problems with monitors are solved through **protection mechanisms.**

Operation Privileges

The primary problem is **protecting programs and the OS from each other!** We solve this by **preventing certain programs from executing certain operations** - implemented by **dividing machine instructions into multiple classes**.

- **Unprivileged Operations:** Any program can execute these. *E.g. loads, stores, jumps.*
- **Privileged Operations:** Only certain programs can execute these. *E.g. peripheral access, rebooting the machine.*

> **Every memory access must be checked** but implementation must be **fast** and **unobtrusive.** So, the hardware is used to achieve this.

The processor hardware can **run into two modes**:

- **Unprivileged:** Normal computations. Called *user mode.*
- **Privileged:** For systems operation. Called *kernel mode.*

This **protects and manages system resources** because **access to hardware is limited to the OS**. The OS always runs in privileged mode and programs in unprivileged mode.

> *Note:* Privilege is a state of the processor, not the program, but we tend to say, "a privileged program" rather than "a program running with the CPU in privileged mode".

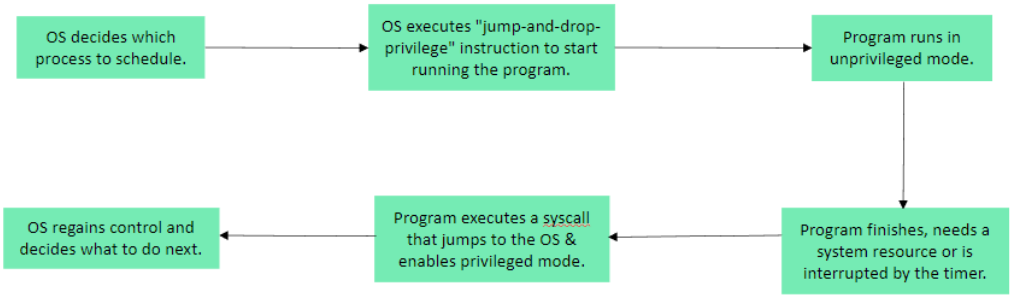
System Traps

If an unprivileged program tries to call a privileged operation:

1. The hardware causes an interrupt called a **system trap.**
2. The hardware sets the processor to privileged mode.
3. The ISR jumped to the OS.
4. The OS then decides what to do. *E.g. kill the program.*

Privilege on Startup

The **system starts in privileged mode** and the following occurs:



Syscalls

A syscall is an instruction which **allows a program to request service from the operating system**, such as reading or writing to a file. They prevent the program from gaining privilege by:

- **Always jumping to the same place in the OS.**
- **Tying the privilege transition to a jump to the OS.**

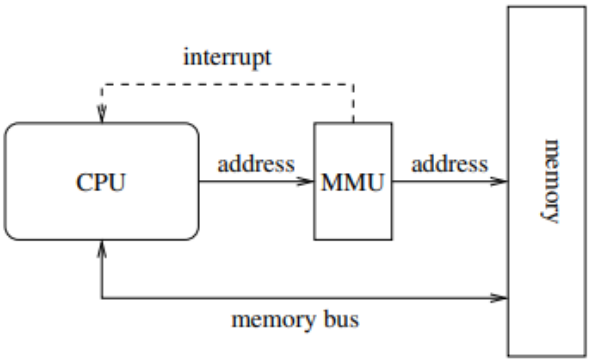
System libraries have interfaces which wrap syscalls to make using them easier.

Issues With Privilege

Bugs in kernel code or **improper separation between the OS and the rest of the system** can lead to privilege mismanagement.

Preventing Programs from Reading and Writing to Each Other

Confidentiality must be enforced while allowing the OS to read and write to certain parts of memory. Once again, this is implemented through hardware because of its speed.



The Memory Management Unit (MMU)

Contains a **table of flags** which determine whether the current program can read or write to a given area of memory.
- Memory is divided into **pages**. Each has a flag.
- 1 bit to say if its readable and 1 bit to say if its writeable.
- Setting flags is a privileged operation.
- Also, an **executable flag** which indicates whether code from this memory address can be executed.

If an unprivileged program tries to read/write to an area of memory it doesn't have permission to, the **MMU raises an interrupt and the OS takes control again**!

Each program has a set of flags which are part of its state.

Processes

Process: A set of code, data and metadata that an OS needs to carry out a certain operation.

A.k.a. a **task** or **job.**

Structure by Process: When one program uses multiple processes.

E.g. one process to compute a picture and one to display it.

Multiple threads of execution are used to structure processes, often in parallel.

Shell: A program that creates a new process based on user input.

Process Control Blocks

A process control block is a **collection of data a process needs to run** to protect and schedule them. These are stored in the kernel. This includes:

- Memory location of code.
- Memory location of data.
- **Relevant MMU flags.**
- Time allocated.
- **Memory, CPU & time used.**
- Number of other resources used. *E.g. I/O or networking.*
- CPU's PC and registers.
- The process identifier (PID) & parent process identifier (PPID).
- User identifiers.
- **The state of the process.**
- **A priority.**

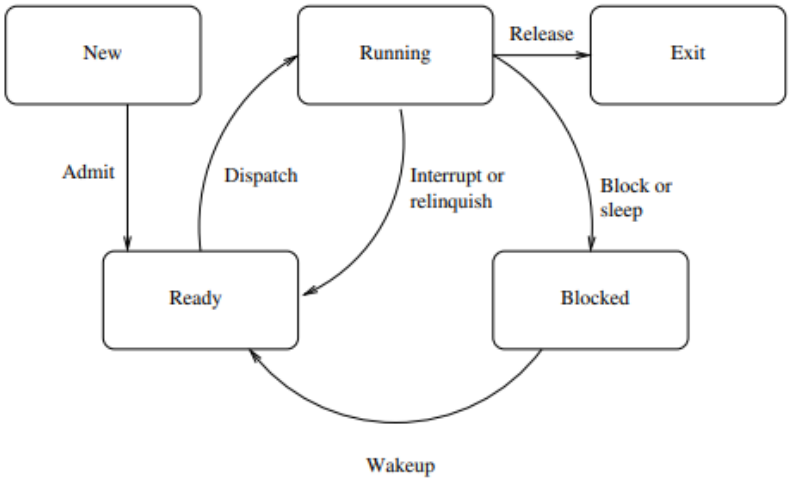
States

A process can be in one of the following states:

- **New:** Just been created. Data structures needed to manage the process are created and filled in.
- **Running:** Currently executing.
- **Ready:** Ready to run, but not currently running.
- **Blocked:** Can't run right now as it is waiting for an event or resource to become available. *E.g. waiting for data from a disk.*
- **Exit:** Finished. Resources used by the process are reclaimed.

Processes are grouped into sets corresponding to each state. Scheduling is deciding which processes to move to which state.

Note: These sets might be in priority order or might be some more sophisticated data structure.



State Transitions

A process begins in new and finishes in exit. Between, the OS decides will transition it between various states. Generally, it will follow these steps:

1. OS decides to schedule a ready process.
2. The process is **dispatched** - it changes the state to running and begins execution. A jump-and-drop privilege occurs.
3. If the process voluntarily suspends itself (**relinquish**), an interrupt occurs, or a timer interrupt arises, the process is moved to the ready state.
4. If the process needs a resource from the OS, it executes a syscall and waits for it. The OS moves it to blocked. Once the data is received, the OS moves it to ready.

Pausing and restarting a process (*e.g. on an interrupt*) requires **saving its state in the PCB** and **retrieving it when it is next scheduled**.

Creating a Process

When a process wants to **fork/spawn** another process, it executes a sycall prompting the OS to create it with the new state. During the new state, the OS will:

- Allocate and create PCB structure.
- Insert the PCB into the relevant kernel list of PCBs.
- Find a free PID.
- Determine and allocate resources needed.
- Calculate its initial priority.

The OS rarely spawns processes. (*E.g. the* ***root process***)

The OS won't create a process if a **policy says not to** or **there's not enough memory**. In this case, the syscall returns an appropriate error to the process.

Scheduling Algorithms

Operating systems must:

- Give each process a fair share of CPU time.
- Make interactive processes respond in human timescales.
- Maximise CPU time for compute-heavy processes.
- Ensure time-critical processes are dealt with quickly.
- Understand the requirements of hardware. (*E.g. mice are slow, but memory is fast)*
- Distribute work amongst devices. (*E.g. CPUs*)

Scheduling Criteria

- How long it's been **running**.
- Its **priority**.
- Likelihood of it **needing the CPU** soon and if it can wait.
- How much the owner **paid** to run it.

A scheduling algorithm must be *fast* but *fair* – more time spent scheduling = less time spent executing programs.

Types of Priorities

- **Static:** Doesn't change.
- **Dynamic:** Adapts to changing progress during system execution.
- **Purchased:** Pay more = higher priority.

Quantifying Algorithms

You can measure the performance of a scheduling algorithm using:

- **Resources** used. (*E.g. memory, CPU cycles, disk*)
- **Throughput**: number of jobs completed in a given time.
- **Turnaround**: response time. Relevant for interactive processes.
- **Real-time performance**: ability to meet strict deadlines.

<table>
<tr><th colspan="2">Run Until Completion</th></tr>
<tr><td colspan="2">• First-in, first-out.
• Non-preemptive.
• Used on pre-OS machines.</td></tr>
<tr><th>Advantages</th><th>Disadvantages</th></tr>
<tr><td>• Good for large computations.
• No multi-tasking overheads.</td><td>• Poor interaction with other hardware due to lack of multi-tasking.
• No interactivity.</td></tr>
</table>

Shortest Job First

- Shortest-time-to-completion runs next.
- Non-preemptive.

Advantages	Disadvantages
• High throughput.	• Starves longer jobs. • Time-to-completion is difficult to estimate so the OS relies on the job description.

Run Until Completion w/ Cooperative Multi-tasking

- Processes are run until they are completed.
- Non-preemptive.
- Uses round-robin to choose next task.

Disadvantages

- Weak multi-tasking: relies on programmer to relinquish control.
- Poor interactivity: does work but takes ages to reach interactivity process.
- Easy for a process to starve other processes.
- Hard to write “good citizen” programs.

Pre-emptive Round Robin

- Each process is given a fixed time slice and is moved to the back of the queue when completed.
- Pre-emptive.
- Multi-tasking.

Advantages	Disadvantages
• No starvation.	• Fixed slicing leads to compute-heavy and interactive processes having the same CPU time. • Interactivity is better but still not good enough for real-time or user interaction.

Round Robin

- Each process is given a fixed time slice and is moved to the back of the queue when completed.
- Good where all processes are similar.

Shortest Remaining Time

- Shortest-time-to-completion runs next.
- Pre-emptive: time-slices are given.

Advantages	Disadvantages
• Good for short jobs. • Good for throughput.	• Long jobs can be starved. • Hard to make time estimates.

Least Completed Next

- Process that has consumed the least amount of CPU time is chosen next.

Advantages	Disadvantages
• All processes will make equal progress. (By CPU time) • Good for interactive processes as they use little CPU time.	• Long jobs can be starved.

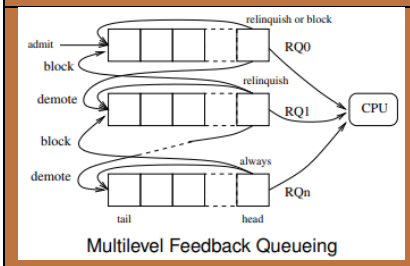
Highest Response Ratio Next

- A variant of shortest remaining time.
- Considers which process has the shortest time to completion and how long each process has been waiting.

$$Dynamic\ Priority = \frac{Time\ in\ System}{CPU\ Used\ So\ Far}$$

Advantages	Disadvantages
• Avoids starvation: Long jobs eventually get a slice. • New jobs get immediate attention.	• Short, critical jobs may not be completed in time. • Lots of scheduling overhead: Priorities need to be recalculated frequently.

Multi-level Feedback Queuing



Multilevel Feedback Queueing

- Useful where processes do not have runtime estimates.
- There are multiple FIFO run queues, RQ0, RQ1, . . . RQn. With RQ0 the highest priority and RQn the lowest.
- Queues are also processed in FIFO priority order. *E.g. RQ1 will not receive CPU time until all processes in RQ0 have been processed.*
- If a process appears in a higher queue, we go back to that queue.
- New processes are added to RQ0 and receive a **quantum** of time. Once this runs out, it is **demoted** to RQ1.
- If it relinquishes then it is placed at the back of RQ0.
- If it blocks for I/O it is **promoted**.
- This repeats for lower queues.

Advantages	Disadvantages
• Prioritises newer, shorter processes. • Prioritises I/O processes. • No arithmetic so little overhead.	• Compute processes receive less CPU time. • Old processes tend to starve.

Amending Algorithms

Some of these disadvantages can be fixed by tweaking the algorithm. However, **this can lead to large, slow and confusing schedulers.** You need to have an extensive knowledge of what's happening to do this.

Traditional Unix Scheduling

- Timer interrupts every 1/60th second.
- Processes with the smallest priority are chosen next.
- Base priority depends on it being a system or user process.
- Same-priority processes are tackled round robin style.

$$\text{Priority} = \text{base priority} + \frac{\text{decayed CPU time}}{2} + \text{nice}$$

Decayed CPU Time: Total CPU used but halved every second to make it based on *recent* behaviour.

Nice: An offset which can be specified for certain processes. Only admin users can give negative nices. Enables **purchased priority.**

Problems with this Approach
• Priorities are recalculated every second increasing the overhead. • Doesn't respond to system changes. • Lacks scalability.

Fair Share Unix Scheduling

Allocates priorities so users get a fair share of execution time rather than processes. Useful where, for example, user A has 9 processes and user B has 2.

Recall that Unix processes are collected into groups in a tree (**process groups**).

$$Priority = Base\ Priority + \frac{CPU\ Time\ Used}{2} + \frac{CPU\ Time\ Used\ By\ Process\ Group}{2} + Nice$$

Modern UNIX derivatives use newer, better scheduling algorithms.

Deadlock

Processors also must schedule other processes such as disk access.

Recall: When a process needs I/O it calls the kernel. This then marks the process as blocked until the resource has arrived.

Definition	***Basic Conditions***
A set of processes D is deadlocked if: 1. Each process P_i in D is blocked on some event e_i. 2. Event e_i can only be caused by some process P_i.	You can only get deadlock if: • There is more than one resource. • There is more than one process.

The Coffman Conditions

A set of conditions which must be true to make deadlock possible:

1. **Mutual Exclusion:** Only one process can hold a resource at a time.
2. **Hold-and-Wait:** A process holds a resource while waiting for other resources.
3. **No Pre-emption:** Resources can't be forcibly removed from processes.

Deadlock happens if:

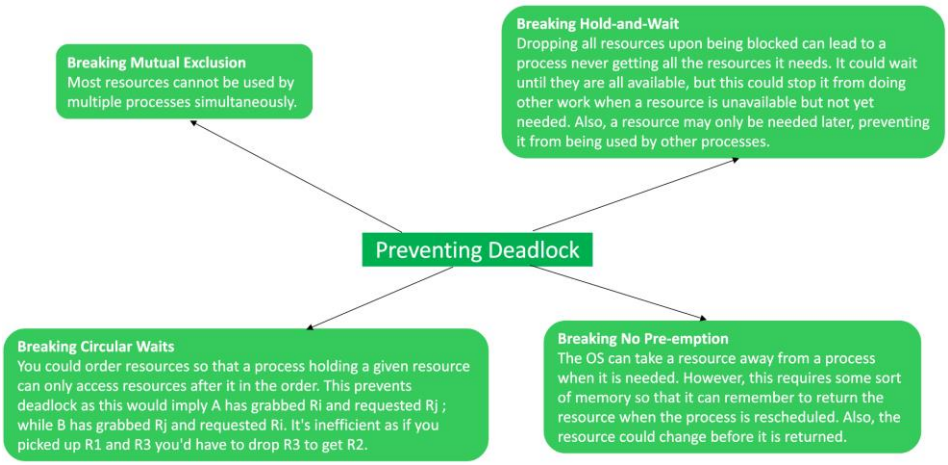
4. **Circular Wait:** There is a circular chain of processes where each holds a resource that is needed by the next in the circle.

Preventing the Coffman Conditions can lead to processes never getting all resources they need, called **indefinite postponement**.

Preventing Deadlock

You can prevent deadlock using one of the following approaches:

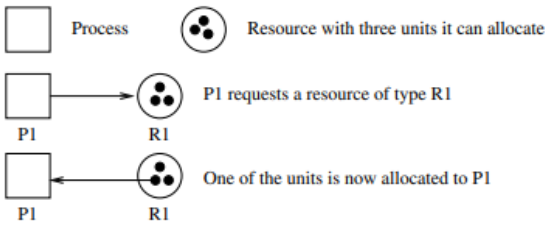
1. Limit resource allocation to prevent one of the four conditions.
   E.g. stopping hold-and-wait from occurring by making a blocked process drop its resources.

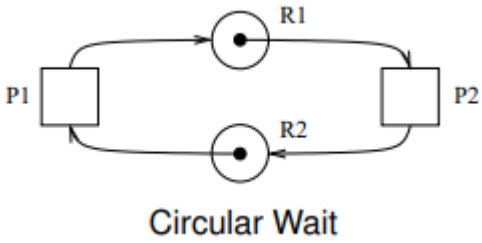


2. Avoid deadlock by not allocating resources that might lead to a future deadlock. This is more efficient but needs to predict future requests for resources. For each request, the OS must decide whether granting the resource can lead to deadlock.

Deadlock Detection Systems

These allow deadlocks to happen, notice and break it. This can be done by detecting circular waits using **resource request and allocation graphs (RRAGs).**





RRAG graphs work by removing certain edges until there are no edges left or a loop is found.

The Ostrich Algorithm

A popular approach to deadlock: ignore the possibility of it happening.

It could be argued that the overheads required to prevent deadlock are too great for such a rare problem. Hence, the occasional reboot is small in comparison.

Virtualisation: When a creates a simulated, or virtual, computing environment as opposed to a physical environment.

For example, the OS can pretend each process has sole access to a resource. There is no possibility of deadlock as every process can progress without waiting, and the OS sorts out transferring the data to or from the real device. However, this introduces the problem of **I/O scheduling.**

Process Protection

Administrator Users

Processes have **UserIDs**. A new process will inherit the UserID of its parent process.

All systems have a **superuser/root/administrator** which can access other users' resources including their files and processes. Specifically, they can suspend or kill any user's processes. Root's processes run in user mode, just like other users' processes. The OS simply doesn't enforce inter-user protections for root.

Root **can give processes to other users by changing the UserID of its processes**, but doing so revokes its privileges. When a user logs in to a system, a process, owned by root, starts. Then, root changes the UserID of this process to the user, effectively switching to that user. It can then start other processes as that user.

The **OS restricts certain resources to the administrator for extra protection**. For example, the shutdown process can only be run by the administrator.

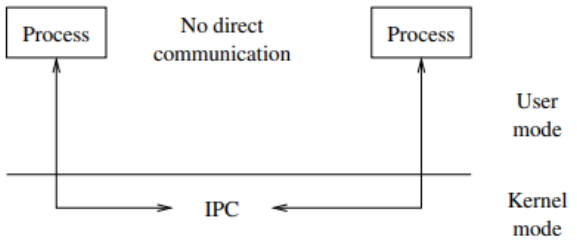
Root-owned processes can interfere with other user's programs and data. Hence, the user of administrator account should be kept to a minimum.

User-level Protections

Users can only access processes and data which has their UserID. This **protects one user's processes from interfering with another user's processes.** For example, if the user protections have been set up properly, virus damage should be limited to the files and processes of that user.

Inter-process Communication

Some processes need to send data to and/or receive data from other processes. *E.g. one process must wait for another to finish some action.*



The OS generally stops processes interfering with one another. Hence, **Inter-Process Communication (IPC)** must be facilitated by the OS.

We will discuss a few methods of communication.

Files

Generally, process A writes to a file and process B reads the file. Good for large batches of data.

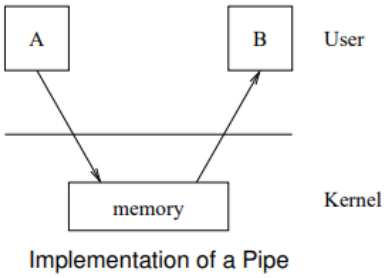
Implementation

- A and B must agree on a file to use. One solution is to use the same file every time. This can cause problems if other processes want to communicate via the same file simultaneously.
- To agree on a filename, they must have previously communicated.
- B must repeatedly poll the file to see if the data has arrived. This doesn't scale well.
- UserIDs must be set properly to only allow authorised processes to read/write to the file.

Pipes

Pipe: A buffer only accessible to the kernel.

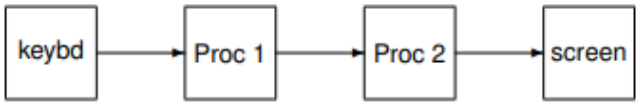
Pipes connect two processes together. They take the output of one and feed it as input to another.



Implementation of a Pipe

- They go via the kernel.
- Pipes may be part of a larger pipeline.
- Data transfer can affect the scheduling of processes involved.
- FIFO.
- Automatically synchronises the writing and reading which makes it ideal for continuous data transfer.

A **executes a syscall** to check that there's space to write to the pipe. If there is, it will. When B wants to read from the pipe it **executes a syscall** to check that there's data to read. If there is, it will. They do this independently of each other.



If the pipe gets full, the writing process is blocked by the OS until space is freed by the receiving process. Similarly, if the pipe gets empty while the receiving process is reading from it, the receiving process is blocked until more bytes are made available by the writing process.

Advantages	***Disadvantages***
• Simple and efficient. • Easy to use. • Used a lot.	• Unidirectional. • Are only between related processes (*i.e. parent and children processes)* • Can lead to deadlocks.

Sockets

Pipes between processes on different machines. They allow bidirectional IPC between two processes and form the basis of the internet.

Shared Memory

Recall that the MMU prevents multiple processes from accessing the same memory. However, its speed makes it an ideal medium for IPC.

Memory is used in IPC exactly how files are and has the same problems. A memory flag can be set up to indicate when A has finished writing but this must still be polled by B. Furthermore, the memory protections (read/write flags) must be set up to authorise this access.

Memory Management

We are concerned with memory distribution between processes, because it is a limited resource, and inter-process protection, which we have already covered.

Gates' Law: Programs double in size every 18 months .

Physical Memory

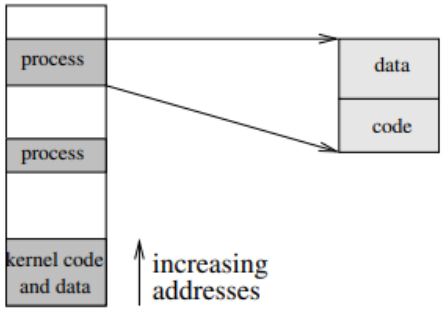
There are numerous ways to allocate and free memory.

- **Static Allocation:** Memory is allocated on process initialisation. Static operating systems can only run a fixed number of processes.
- **Dynamic Allocation:** Memory is allocated while the process is running.
- Freeing memory while the processes is running.
- Freeing memory when the process ends.

Kernel memory must be dynamic. For example, it must be able to expand and shrink when processor control blocks are created and deleted.

Early Operating Systems

Early operating systems had static allocation. Dynamic allocation was introduced later:



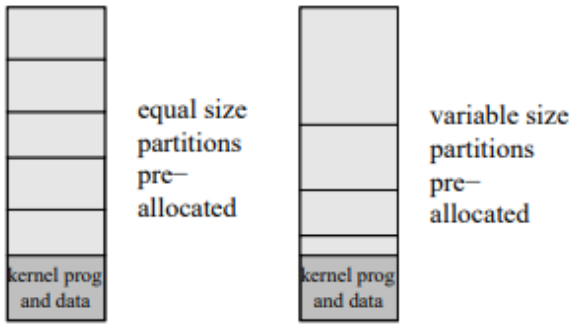
Memory Layout

The gap above the kernel is vital because it allows for it to dynamically allocate memory to itself.

Static Partitioning

Sections of memory are allocated **at boot time**. Processes are loaded into the **smallest partitions that will fit them**.

- Ideal if dynamic partitions aren't possible.
- Good if you run a fixed set of applications known in advance.



A process must be stored in one continuous

Equal Size

- Easy to implement.

- Usually causes wasted space when a process doesn't fill its allocated partition.
- Can't cope with larger processes.

Variable Size

- Harder to implement.
- Efficiency depends on the choice of partition sizes.

chunk of memory because **all a program's instructions must be in the same location**.

Similarly, all data must be stored in a continuous chunk of memory.

Overlays

Used if a process is too big to fit into the allocated memory; only part of the process is loaded into memory (**partly resident**). If a non-resident part is needed, the programmer must swap it with a part of the process which isn't needed. Only excellent programmers can achieve this, and it reduces the speed of execution.

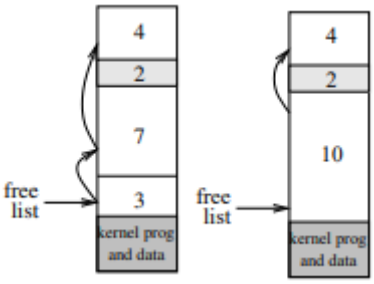
Similarly, data can be swapped out, but it must be saved to the disk because it doesn't always have a copy on the disk unlike processes.

Dynamic Partitioning

Fragmentation: When there are lots of small free sections of memory. They are either carved off a bigger block during an allocation or are return at process exit.

Allocating memory sequentially often leads to memory fragmentation. For example, there is enough space here to run a process of size 5 but nowhere to put it.

Combating Fragmentation



There are numerous methods for preventing fragmentation:

- **Freelist:** A sorted list of all free blocks.
- Coalescing adjacent free blocks.

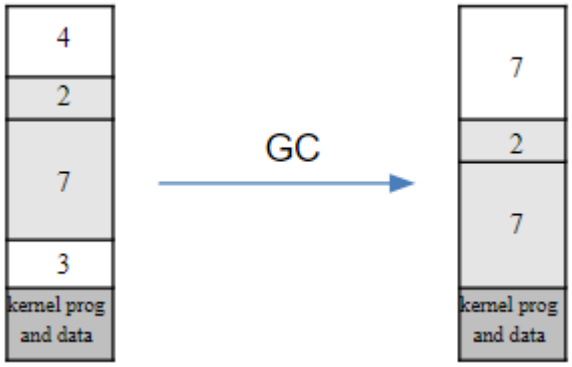
Keeping a freelist allows the OS to track which blocks are empty.

When we want some memory space, we search the freelist, select a big enough block, slice off how much we need and return the remaining partitions to the freelist.

Strategies for Choosing a Block

- **Best Fit:** Smallest available big enough hole.
- **First Fit:** First available big enough hole.
- **Worst Fit:** Biggest available big enough hole.
- **Next Fit:** Next available big enough hole from last allocation.

Garbage Collection



When there isn't enough free space for a process, the OS can compact memory:

1. Stops running all processes.
2. Shifts all process code and data to close the gaps.
3. Re-schedules processes.

Disadvantages

- **Time Consuming:** Moving lots of bytes around which takes away time from the running of processes.
- **Bad for Interactive and Real-time Processes:** Same reason.
- **Better solutions are possible** given the necessary hardware.

Memory Pre-emption

If even garbage collection can't create enough space, you could not admit a process to memory, kill an existing process or pre-empt memory. Pre-emption is the best option.

Pre-emption requires **saving the data** of a process, **taking its memory away** and **giving it to another process**, **returning the memory** and **loading the data and instructions back** into it so the original process can resume.

Unlike instructions, a process' data isn't stored on the disk. So, this needs to be backed up when the memory is taken away from it. Disks are slow so this has a **large overhead.**

There are several methods for memory pre-emption:

Swapping

One or more processes are **copied out of the OS to make space for another process**. Ideally, these would be blocked processes. Swapped processes must be **copied back into memory** before it can be scheduled again – code is copied back from its program file and data from where it was saved to.

- Programmers don't have to do the swapping.
- Slows down execution speed.
- Good programmers avoid swapping.

Paging

Memory is broken down into equal pages. Hardware is designed so copying these in and out of memory

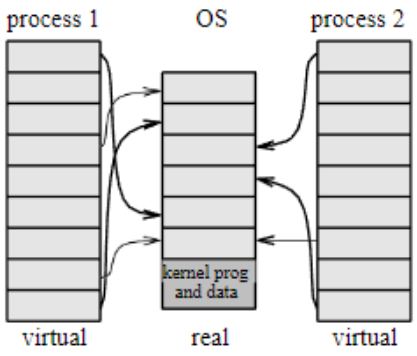
from a disk is as efficient as possible. Paging/swapping is the process of copying these pages to and from the disk. It enables code and data to be continuous in the virtual address space but not in the physical.

Page: A continuous area of memory.

Virtual Addresses/Memory	*Physical Addresses/Memory*
A **per-process fictional address**. • User processes only see these. When processes want to access memory, the OS translates them into physical addresses which helps to **prevent fragmentation**.	Numbering the bytes in memory from 0 to n.

Page Tables

Proc	Virtual Page	Physical page
1	3	7
2	3	42



Each process has a page table containing the virtual-physical address mapping for each page it is using. These are stored in **kernel memory** with a link being kept in the process's PCB.

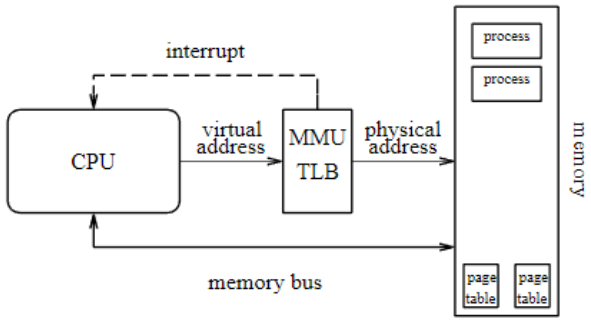
The **virtual address** is where a process appears in the virtual page space. *E.g. Given a page is 4096 bytes, virtual address page 3 begins at byte 12298 (3*4098).* The same virtual address in different processes are always mapped to different physical addresses.

Even if they have the same UserID, every process has its own separate virtual address space which is mapped onto the physical address space.

The Transition Lookaside Buffer

The memory management unit (MMU) translates virtual addresses to physical addresses with the help of the **transition lookaside buffer** (TLB) inside of it.

The TLB is **a cache which stores copies of a few mappings of the current process to allow for quick translation**. The speed of memory access relies upon the **relevance of the mappings** in the TLB. Lots of TLB misses results in slower memory access because more misses and page faults occur.



On every memory access, we must:

1. Check the TLB for the virtual page. If it is present, a **TLB hit** occurs.
2. Read the page table to find the mapping.
3. Calculate the physical memory location.
4. Access the memory of the physical address.

TLB Misses

If a mapping is not present in the TLB, a **TLB miss** occurs. This can be handled in one of two ways:

- **Hardware Managed TLB:** The CPU completes a **page walk** where it searches for the virtual page in the page table in memory. If it finds it, it adds it to the TLB table and continues.
- **Software Managed TLB:** The MMU raises an interrupt and the OS takes over to complete a **page walk**.

Page Faults

If the requested virtual page hasn't been allocated by the OS to the process, the **OS allocates a physical page, installs the new page mapping** into the page table for that process and **writes this to the TLB**. The memory access can then proceed when the process is rescheduled.

- **Hardware Managed TLB:** The MMU will raise an interrupt to pass control to the OS.
- **Software Managed TLB:** OS is already running because of the page walk.

Lazy Page Allocation

Physical memory is only allocated on a page fault meaning page faults always occur when a virtual page is requested by a process for the first time.

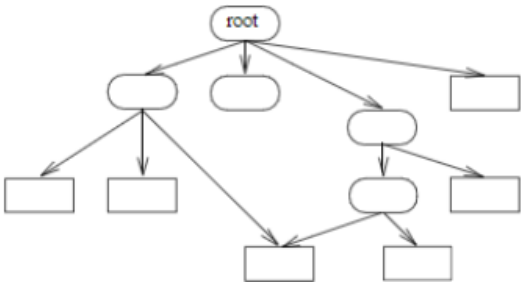
This allows a process' virtual space to be larger than the physical space.

Filesystems

Filesystems are an organisation of data. Modern filesystems are byte oriented.

Files

A file is **a named chunk of data stored on a disk**, generally a long array of bytes with no structure.



*Directory hierarchies form **directed acyclic graphs (DAGs)**.*

Hierarchies

Files are organised using hierarchies.

- Related files are grouped in a directory/folder.
- Top-most hierarchy is called the **root**.
- Directories can only be inside one directory.
- Directories can be empty.

Directory: A collection of names. They contain:

- Files, subdirectories and each file's inode number.

Each process has a **current working directory (cwd)** which is a prefix stored in the process' PCB that is referenced when a process asks for a file by an incomplete filename.
E.g. a process with cwd "/xmorg/downloads/" will interpret "image.psd" as "/xmorg/downloads/image.psd".

Requirements

Alongside the basic functionality like creating and deleting files and directories, we must design filesystems that prioritise:

- Speed of access.
- Reliability.
- Protection/security.
- Backup and recovery.

Records

Unlike modern files, early files called **records** had structure because they were a fixed-size block of data.

The Unix Filesystem

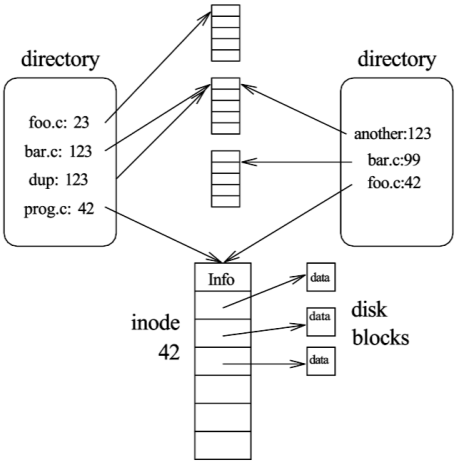
Inodes

Unix introduced the concept of **inodes**. Each file has an inode which is a **fixed size data structure** that contains all information about the file including:

- **Timestamps:** Last access and modification.
- **Ownership:** UserID of the owner for protection.
- **Size.**
- **Type:** Plain file, directory or special file.
- **Access Permissions:** Who can read or write or run this file (if it's a program).
- **Reference Count:** Number of names it has.
- **Pointers:** To areas on the disk where the actual data lives.

Remember: FIlenames are stored in the directory, not inodes!

They are stored in an array on the disk and are referred to by their **inode number**.



Deleting a file is as simple as:

1. Removing the name from the relevant directory.
2. Decrementing the reference count in the inode.
3. If the count reaches 0 the OS can free the inode and the disk block it refers to.

Disk Blocks

A disk block is **a fixed size area on the disk.** Each inode contains **block pointers** which indicate the addresses of the **disk blocks** for that file. Their fixed size enables fast allocation and deallocation but can lead to wastage.

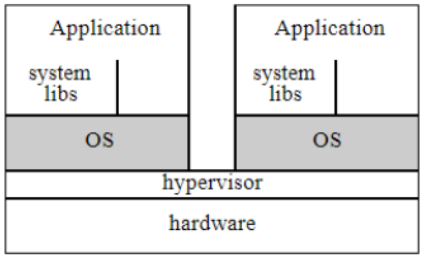
Reading Files

When a file is read, the directory address is followed recursively to find the blocks containing the file. For example, if you wanted to open the file “image.psd” with a cwd of “/xmorg/downloads/” it would:

1. Prepend the cwd to make: “/xmorg/downloads/image.psd”.
2. Search the root disk blocks for the name “xmorg”.
3. Once found, get the inode for “xmorg”. It sees that it refers to a directory and reads the disk block containing the directory.
4. Search the “xmorg” disk blocks for the name “downloads”.
5. Once found, get the inode for “downloads”. It sees that it refers to a directory and reads the disk block containing the directory.
6. Search the “downloads” disk blocks for the name “image.psd”.
7. Once found, get the inode for “image.psd”. It sees that it refers to a file and reads the disk block containing the file.

OS Virtualisation

Some applications only run on specific OSs. OS virtualisation **allows you to run multiple OSs on one machine**. This can also be useful where several users are sharing hardware, but each have their own OS.

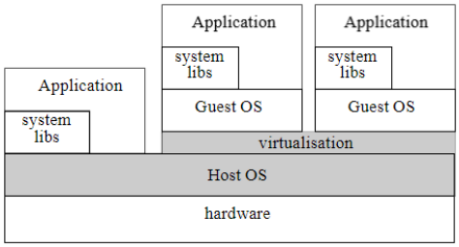


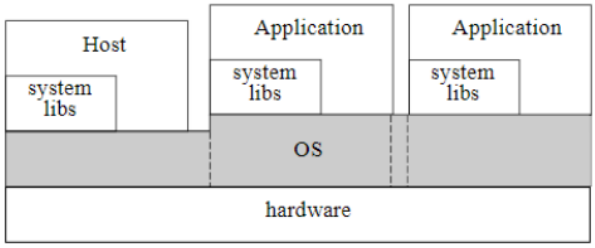
Bare Metal Virtualisation

Hypervisors help to implement virtual OSs by managing the hardware, allowing each OS to see a separate "virtual hardware".

Hosted Virtualisation

Has a **host OS** that runs virtualisation code for one or more **guest OSs** that run on top of it.





Not Quite OS Virtualisation

Sections are isolated meaning applications share the OS but are separated into partitions. This allows them to run on the same OS but have different system libraries and software.

Hardware Virtualisation

Emulates different kinds of hardware.