Operating Systems (2019 - 2020)

No guarantees are provided on the accuracy, completeness or the correctness of these notes.

Overview

1. OS Structure
2. Processes and Threads
3. Inter-process Communication
4. Memory Management
5. Device Management
6. Disk Management
7. File System
8. Security

Operating System Characteristics

- OS's must allow sharing of data, programs and hardware, while also offering resource allocation. They must allow efficient and fair use of memory, CPU time and disk space
- Must allow access to resources (disks, RAM, network, GPU)
- Must offer mutual exclusion for some resources
    - For multiple writes to the same file, an OS may provide each application its own copy, or may offer mutex
- Must protect against corruption (accidental or malicious)
- Concurrency
    - Overlap I/O and computation, since I/O is slow, so avoid the CPU being idle while I/O device responds
    - Multiple users and programs in parallel
    - Could switch activities arbitrarily
    - This must be done safely
- Non-Determinism
    - Events occur in an unpredictable order
- Persistent data storage
    - File systems for disks, DVDs, etc.

Tutorial Question

What are the most important resources in:

- A supercomputer?
    - CPU cores and computation
    - Memory needed for data storage
- Workstations connected to servers through a network
    - Bandwidth and network connectivity. Efficient packet processing is difficult.
- Smartphone
    - Battery life or energy
    - Cellular network, Bluetooth and communication technologies
    - Sensors such as GPS, accelerometer, gyroscope

Types of Operating Systems

- Multiprocessor (Windows, OS X, Linux)

    - Often best-effort OS's - no guarantee on when a task is done
- Real time
    - Useful when real-time response is needed
- Embedded
    - Specialised, work on low resources
- Server OS
    - Share hardware and software resources
- Mainframes and supercomputers
    - Run on bespoke hardware
- Smartphones
    - Energy-efficient

Tutorial Question

What is the kernel?

The part of the OS which runs in privileged mode on the CPU. It always stays in memory and implements the commonly executed functions of the OS. It has complete access to all hardware unlike user processes.

Kernels

There are several different types:

- Monolithic
- Micro-kernels
- Hybrid

Monolithic

- A single executable within its own address space
- Used in Linux
- Benefits:
    - Efficient calls within the kernel, since once the system call is made, all the code executes as a single binary, so you can stay in kernel mode
    - Easier to write components for the kernel as all components share memory
- Problems:
    - Design is more complex as there are a lot more interactions
    - No protection between components of the kernel

Micro-kernels

- The part of the kernel which is in kernel mode is limited, usually includes functionality such as basic IPC, virtual memory, scheduling

- All other functionality is in user mode, such as application IPC, device I/O, device drivers, file servers
- The micro-kernel then provides IPC for these servers in user mode
- However, the application can directly talk to these user-mode services without switching into kernel-mode
- Benefits:
    - Kernel is less complex and less error-prone
    - Servers have cleaner interfaces
    - Specific services can crash and restart without crashing the kernel
- Problems
    - IPC within the kernel has a high overhead
    - When hardware performance was low, this was a large problem, meaning that early kernels tended to be monolithic
- In Android, many services are provided as user-mode servers

Hybrid

- Combines both of the above
- Design philosophy
- Some servers are moved into user-mode
- Many modern OS's use this approach

Case Study: Linux Kernel

- **Linux system calls** were implemented by putting arguments in registers or the stack.
- Then, issue a **trap** (synchronous interrupt) to switch from user to kernel mode.
- **Interrupt handlers** are the primary way to interact with devices, typically written in Assembly as they are very low level pieces of code, and need to be performant.
- Interrupt handlers initiate dispatching - they stop the process, save the state, start the driver and return.
- Hides I/O components under a virtual file system.
- Utilises virtual memory with paging.
- Dynamic loaded modules which can be added to the kernel at runtime, allowing for new functionality without a large kernel binary.
    - Can support new hardware, allowing Linux to work on different types of machines.
    - New device drivers can be added at run time.

Case Study: Windows NT

- NTOS kernel layer provides system calls
- Programs build on top of **Dynamic Code Libraries (DLLs)**
- Some services are in userspace, implemented as DLLs
- Device drivers in kernel mode
- Good example of a hybrid kernel
- Structured like a micro-kernel, but most components run in the same kernel address space, making it more efficient
- However, bugs can cause the whole kernel to crash
- **Hardware Abstraction Layer** allows for portability to many types of CPU (historical reasons meant that it only supported Intel)

- Windows kernel includes multiple system call APIs (Win32, POSIX, OS/2)

Tutorial Questions

1. Why is separation of user and kernel mode good OS design?
    1. Minimises the amount of code run with high privilege.
2. Which instructions are only in kernel mode and why?
    1. Disabling interrupts
        1. Must be run in kernel mode as it disables pre-emption of threads and hence concurrent operations. Allowing users to do this would stop the kernel from scheduling.
    2. Reading time of day
        1. Can be done in user mode.
    3. Changing memory
        1. Also in kernel mode, as could be used maliciously to attack processes, or may accidentally corrupt the memory.
    4. Setting time of day
        1. Kernel mode, as the time of the day is globally visible across programs, and processes make assumptions about time (e.g. monotone) so this may allow attacks.
3. Give one example where the execution of a user process switches from user mode to kernel mode and back
    1. Anything which requires a system call, such as reading a file. The read system call then transitions from user mode to kernel mode and back again.
4. Why is it infeasible to build an OS which is portable without any modification?
    1. A portable operating system can run on different CPU architectures. But changing the architecture changes the instruction set, and elements of the kernel such as interrupt handlers need knowledge of the specifics of the instruction set available.
    2. In general, microkernels are easier to port as a small part of the kernel is privileged. However, monolithic/hybrid kernels such as Windows NT and Linux can become portable using a harware abstraction layer of some sort.

Processes

> An abstraction for a running program

Processes believe that they are the only program running on the CPU. A CPU can interleave the execution of processes to provide concurrency. Furthermore, processes have isolation such that each process has its own address space. We can use IPC to allow communication between processes and to overcome this isolation, but processes such as Firefox don't need to have knowledge of any other processes on that machine.

Concurrency

Definitions:

- **Time-Slicing**: The OS switches the process running every, for example, 50ms. The context switch is done in a way so that the state of the process is saved and restored, so that the proccess is under the illusion of continuously running.
- **Pseudo-Concurrency**: a single processor switches between processes by interleaving.
- **Real Concurrency**: Utilises multiple physical processors or CPU cores to run processes in parallel. Usually there are fewer processors than processes.
- **Fairness**: Users may notice unfairness due to lag or freezing in an interactive application. Hence, the time-slices are usually very short.

We can increase CPU utilisation by context switching and running computation when waiting for I/O data.

Tutorial Question

If on average a process computes 20% of the time, then with 5 processes, we only have 100% usage only if each process never waits for I/O at the same time. A better estimate is:

$P(B) = p^n$ and $U = 1 - p^n$, where B is the event that all five processes are blocked, and U is the CPU utilisation.

How many processes need to be running to only waste 10% of the CPU if they spend 80% waiting for I/O?

We need $U = 1 - P(B) = 0.9$ which means that $P(B) = p^n = 0.1$, hence since $p = 0.8$ we get that $0.8^n = 0.1 \implies n = 10$ (aproximately).

Context Switching

A context switch switches execution of a process from process A to process B. This is done by a periodic time interrupt so that the kernel can make a decision in scheduling a different process. Hence, the scheduling decision is non-deterministic as the OS may switch processes in response to events and interrupts.

After a context switch occurs, the process may later be resumed, hence all information about the process needed to restart it safely needs to be stored. This is stored in a **proccess descriptor** or a **process control block (PCB)**, which is in the **process table**.

Additional data such as contents of the CPU registers must also be saved. These are saved on the stack by the hardware. By the time the interrupt handler runs, the hardware has already saved these registers.

The proccess has its own virtual machine:

- A virtual CPU
- Its own address space
- File descriptors, etc.

The following information should be stored in the PCB:

- Program counter
- Page table register
- Stack pointer
- Process management information
    - PID

    - Parent process
    - Process group
    - Priority
    - Which CPU it was running on
- File management information
    - Root directory
    - Working directory
    - Open file descriptors

Context switching is expensive: direct cost is saving and restoring proess state, but the indirect cost is due to cache invalidation. The caches need to be flushed: **Translation lookaside buffers** which cache mappings of virtual addresses to physical addresses are flushed on a context switch.

Process Lifecycle

Process Creation: Processes are created on system init., by user request, or by system calls. They may be foreground (interactive) processes, or background processes (e.g. daemons on Linux).

Process Termination:

- In **normal completion**, the process completes execution of the code body.
- System calls such as `exit()` (Unix) and `ExitProcess()` (Windows) can terminate a process.
- Abnormal exit is when the user has run into an error or an unhandled exception.
- Processes can abort other processes (e.g. killing from a terminal).
- Some processes such as daemons never terminate unless an error occurs.

Process Hierarchies:

- In UNIX, there is a process hierarchy. On startup, it runs the `init` process, which is the root of the tree. It reads a file showing how many terminals to run and forks off processes accordingly.
- In Windows, there is no heirarchy. When a child process is created, the parent is given a handle to control it which may be passed to another process.

Case Study: UNIX

Creation:

- `int fork()`:
    - `int fork(void)` is a system call which creates a new child process by making a copy of the parent process image.
    - The child inherits resources of the parent and will be executed concurrently with the parent.
    - It returns twice:
        - In the parent, it returns the PID of the child
        - In the child, it returns 0
        - On error, no child is created and -1 is returned
    - It may fail when memory cannot be allocated for the child. Global process limit or per-user limit could be exceeded.

Tutorial Question

Consider the following example:

`4 if (fork() != 0) printf("X\n");`

`5 if (fork() != 0) printf("Y\n");`

`6 printf("Z\n");`

What is the process hierarchy tree?

We have process A forking on line to create process B. Then, both process A and B fork on line 5, creating processes C (child of A) and D (child of B). Finally, all 4 process print "Z". Hence, we get the following structure:

`A ----> B ----> D`

`|-----> C`

and the following output (order not deterministic):

`> X` from process A.

`> Y` from process A.

`> Y` from process B.

`> Z` from process A.

`> Z` from process B.

`> Z` from process C.

`> Z` from process D.

- `int execve(const char *path, char *const argv[], char *const envp[])`:
  - Takes the path of the program, arguments to pass to its main function, and a list of environment variables, and changes the process image accordingly and executes it.
- `int waitpid(int pid, int *stat, int options)`:
  - Suspends execution of the parent until the process with the PID terminates normally or a signal is received.
  - `PID = -1` indicates waiting for any child, `0` indicates waiting for child in the same process group and `-gid` indicates wait for any child with process group given by `gid`.
  - Returns the pid of the terminated child process, `0` if `WNOHANG` is set in the options (so the call does not block) and there are no terminated children or `-1` on error.
- UNIX has both `fork` and `execve` because they take different arguments and cases. Windows has a single method, `CreateProcess`, which has the functionality of both, with 10 parameters.
- With child processes, the parent must take care to clean up resources, to avoid zombie processes.
- UNIX has had `fork` from the beginning as the copying semantics were implemented by hardware, hence it was easy to add into the kernel, but from a design stand point it causes issues.

Process Termination

- `exit(int status)`:
  - Terminates a process (called implicitly when the program finishes execution)

    - Never returns to the calling process, instead returning an exit status to the parent
- `kill(int pid, int sig)`:
    - Sends a signal `sig` to the process `pid`.

Interprocess Communication

Some types:

- Files - by writing and reading from files
- Signals in UNIX
- Events and exceptions in Windows
- Pipes
- Message Queues in UNIX
- Mailslots in Windows
- Sockets
- Shared memory
- Semaphores for synchronisation

UNIX Signals

A mechanism for IPC. Signal delivery is similar to hardware interrupts, used to notify processes when an event occurs. Processes can send signals if they have permission to do so: the real or effective user ID of the receiver must match tht of the sending process or the user must have privileges such as super-user. The kernel may send signals to any process.

Signals are generated:

- By an exception:
    - `SIGFPE` - floating point exception, e.g. division by zero
    - `SIGSEGV` - memory segement violation
- Kernel wants to notify of an event: `SIGPIPE` - process writes to a closed pipe
- User-triggered: `SIGINT` - If the user presses `Ctrl+C` (keyboard interrupt)
- In code, using the `kill()` function

A process can specify which signals it wants to handle. The default action for most signals is to terminate. But the process may also ignore it. `SIGKILL` and `SIGSTOP` cannot be ignored or handled.

`signal(SIGINT, my_handler)` can be used to handle a signal, for example, handling `SIGINT` can allow the process to do any cleanup.

UNIX Pipes

A **pipe** is a method of connecting the standard output of a process into the standard input of another. It allows for **one-way** communication between processes.

It is commonly used on the command line, e.g. `ls | less`.

We have two types of pipes; **unnamed** and **named**. They can be created using `int pipe(int fd[2])`, which takes two file descriptors (the read end and then the write end). The sender should close the read end and the receiver should close the write end. If the receiver reads from an empty pipe, it will block until data is written, and vice versa.

Named pipes (FIFOs): these are persistent pipes which outlive the process which created them, and are stored on the file system. Processes can open it like a regular file, but the pipe is established it in memory instead of the storage media, which is faster than a file.

Tutorial Question

1. When two process communicate through a pipe, the kernel allocated a buffer. What happens when the process at the write-end attempts to send additional bytes on a full pipe?
    1. The writing process blocks until the read-end reads bytes from the pipe, as there is nowhere for the kernel to hold that data.
2. What happens at the write-end when the pipe attempts to send additional bytes but the other process has closed the file descriptor associated with the pipe?
    1. The process receives an error, such as a `SIGPIPE`.
3. The process at the write-end wants to transmit a linked list data structure over a pipe. How can it do this?
    1. It cannot send the pointers as they are only meaningful in the address space of the sender. The data structure must be serialized, for example, send the elements one by one with a separator in between to indicate the end of the element.

Threads

Threads are an abstraction for execution which live in the same address space of meemory. With multi-threading, each process can contain one or more threads.

Per-Process	**Per-thread**
Unique address space	Program Counter (PC)
Global variables	Registers
Open files	Stack
Child processes	
Signals	

Benefits:

- Threads can execute in parallel and access and process the same data. They can block independently, allowing other threads in the same process to run. For example, we may have a processing thread, and input and output threads.
- While we could use proesses, it is difficult to communicate between address spaces, blocking activities will context switch from the process, and it is expensive to create and destroy processes as well as to context switch between them.

Disadvantages:

- Threads can corrupt another thread's stack and memory
- Concurrency bugs arise from accessing shared data such as global variables
- On a `fork`, what are the correct semantics?
    - On Linux, a child process created from a multi-threaded process is single threaded. But now, locks may not be released, as different threads dont hold different locks.
    - We could also create a process with the same number of threads.
- On signal arrival, what happens?
    - We could specify the thread which handles it.
    - We could create a separate thread on demand.

Case Study: PThreads

PThreads (Posix Threads) are implemented by most UNIX systems.

- To create a new thread, we can use `int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*sart_routine)(void*), void *arg)`/
  - This creates a new thread and returns 0 if the thread was successfully created or an error code (if memory isn't available in the process for example).
  - Arguments:
    - `attr` specifies thread attributes such as min stack size, can be `NULL` for defaults.
    - `start_routine` specifies the C function the thread will execute.
    - `arg` is an argument to be passed to `start_routine`.
- To terminate, we can use `void pthread_exit(void *value_ptr)`
  - This terminates the thread, makes `value_ptr` available to any successful join with the terminating thread.
  - Called implicitly when thread's start routine returns, but not for the initial thread.
  - By default if a process terminates then all of its threads will terminate, but for some processes such as a daemon, it should wait until all threads terminate.
    - If `main()` terminates before all threads have exited, then the default behaviour occurs.
    - If `main()` exits with `pthread_exit()` then the process waits until the last thread terminates or until `exit()` is called.
- To yield to the CPU, we can use `int pthread_yield(void)`, which releases the CPU to let another thread run. Returns `0` on success or an error code. Always succeeds on Linux.
  - Processes have a similar function, `nice()`, which reduces its priority so that another process may be scheduled.
- To join, we use `int pthread_join(pthread_t thread, void **value_ptr)`
  - Blocks the caller until the given `thread` terminates. The value passed by `pthread_exit` is then available at the memory location specified by `value_ptr`, which can be `NULL`.

Tutorial Question

How can we pass multiple arguments to the thread?

We can do this by passing a pointer to a struct, which can contain all the parameters.

Implementing Threads

There are multiple methods of implementing threads:

- User-level threads
  - The kernel is not aware.
  - The user process manages its own threads.
  - A library provides threads and switches between the threads. Languages with co-routines are effectively user level threads.
  - The process maintains the thread table.

    - Provides better performance, as thread creation and termination is done within the process. Can create a large number of threads efficiently and can switch between them fast.
    - Thread synchronisation is fast.
- Kernel threads
    - Implemented in the kernel.
    - Kernel level threads means that the kernel can pre-empt the process for better CPU utilisation.
        - If a user level thread has a blocking system call, then the whole process is blocked.
        - However, non-blocking I/O can be used, but this is harder to use and less elegant.
        - During a page-fault, the OS would block the whole process with user-level threads, but some of the threads may still be able to run. Kernel threads mean the specific thread at fault can be blocked.
    - Blocking system calls and page faults can be accomodated easily.
    - Disadvantages:
        - Higher thread creation and destruction overhead, as they require system calls (but cheaper than processes)
            - Can recycle threads through pools
        - Synchronisation more expensive
        - Switching more expensive
        - No application specific scheduling
- Hybrid approaches: use kernel threads and then utilise some user level threads in those kernel threads.

Tutorial Questions

Q1: Compare a single-threaded file server and a multithreaded server on a single CPU machine.

To receive a work request, dispatch it and do all processing, it takes 15ms, assuming the data is in the block cache. A disk operation is needed 1/3rd of the time (when we have a cache miss), which requires an additional 75ms, where the thread sleeps. Thread switching time is negligible. How many requests/sec can be handled in each case?

For a single threaded file server, we have $\frac{2}{3}(\frac{15}{1000}) + \frac{1}{3}(\frac{15}{1000} + \frac{75}{1000}) = 0.04\text{s}$ on average to handle a request, meaning that we can handle 25 requests per second.

In the multi-threaded case, each request needs 15ms CPU time and $\frac{1}{3} * 75 = 25\text{ms}$ I/O time. The probability that all n threads are sleeping: $(25/40)^n$. Then, CPU utilisation time is $1 - (5/8)^n$ so that in one second, the CPU handles $(1 - (5/8)^n) * 1000/15$ requests. For example, with 6 threads, we have 62.69 req/s.

Q2: If a multithreaded web server the only way to read from a file is the normal blocking `read()` call then which type of threading is better?

Kernel level threads as this would allow other unblocked threads to be scheduled as part of the process. User level threads would block the whole process.

Scheduling

Proccesses can be in one of the following states:

- New
  - Must be enabled, transition into ready state
- Ready
  - Can be scheduled so that they start running
- Running
  - Can be pre-empted and made ready
- Waiting
  - Blocked on I/O or a semaphore, transition from running to waiting to ready
- Terminated
  - When the process dies

Scheduling algorithms have a few goals:

- Fairness
- Avoid starvation
- Enforcing of a policy such as priorities
- Maximization of some resource utilisation such as CPU or I/O
- Minimize overhead from context switches and scheduling decisions

The goals depend upon the type of system:

- Batch systems
  - Throughput (jobs/time).
  - Turnaround time: time between job submission and completion.
- Interactive systems
  - Response time: time between request issue and response.
- Real-time system
  - Meet deadlines: soft deadlines lead to degredation, such as video playback (video quality decreases), hard deadlines such as real-time control systems e.g. on a plane.

We also have two different classes of scheduling:

- Preemptive
  - Let processes run for a maximum amount of some fixed time/
  - Clock interrupt causes scheduling decision to be made.
- Non-preemptive
  - Process runs until it blocks or voluntarily releases CPU.

And types of processes:

- CPU-Bound: spends most of the time using the CPU
- I/O-Bound: spends most of the time waiting for I/O, and using CPU briefly, such as the shell (waiting for user input), or a web server such as a file server.

Algorithms

First-Come-First-Served (Non-Preemptive)

- The runnable process is added to the end of the ready queue, and the running process runs until it blocks or terminates.
- There is no indefinite postponement, so there is no starvation as long as processes definitely terminate.
- If a long job is followed by very short jobs then average turnaround time is very low. Throughput would not be affected as to complete all processes takes a fixed amount of time.
    - This is known as head-of-line blocking, where long jobs block shorter jobs.

> Side note: For job scheduling, we may expect the user to provide an estimation of the time taken in order to make better decisions.

Round-Robin (Preemptive)

- We let a ready process run for some fixed time quantum and then context-switch.
- Fairness: ready jobs get an equal share of the CPU
- Response time is good for a small number of jobs
- Average turnaround time now depends upon the number of other jobs and their length. If there are a lot of jobs of similar length, then average turnaround time is high as all jobs will make progress at the same rate.
- The quantum needs to be in relation to switch overhead:
    - If context switch time is 1ms and the quantum is 4ms, then we have 20% overhead, compared to 0.1% with 1s.
    - With a large quantum we have a smaller overhead but worse response time. If we set the quantum to infinity, then we have FCFS.
    - With a small quantum, we have larger overhead but faster response time. Linux uses 100ms, Windows desktop uses 20ms and Windows server uses 180ms. The quantum should be muh larger than the context swtich but provide a good response time. As seen, desktops and servers can have a different response time.

Shortest Job First (Non-preemptive)

- When run times are known in advance, we can pick the shortest job first.
- This gives lowest turnaround time and job throughput (optimal, in fact) when all jobs are available simultaneously, and hence this approach may be used on compute clusters for example.

Shortest Remaining Time (Preemptive)

- Preemptive version of SJF, with runtimes known in advance.
- Choose a process where remaining times are shortest.
- When new process arrives with less execution time than current, we can run it.
- Allows new short jobs to get good service.
- However, we can have the current process be continually blocked by new processes (starvations)

Tutorial Question

What if a running process is nearly complete and the shorter job arrives?

We can consider the context switching overhead, and only switch if the dfference in remaining runtimes is greater than this overhead. We may disable preemption when the remaining run time reaches a threshold to avoid indefinite postponement.

In general, we do not know the run time. We could compute estimates based on heuristics (such as previous history) but this is not always applicable. We can use user-supplied estimates, but now we need to counteract cheating (stating lower runtime) to get higher priority, by, for example, terminating or penalising proess if it exceeds the estimated run time.

Fair-Share Scheduling

We may take into account who owns a process. Each user has some fraction of the CPU so that users cannot simply gain more CPU time by submitting more processes.

Priority Scheduling

Jobs can be run based on priority, where the job with highest priority runs. These can be *externally defined* by the user or by some *process-specific metric* such as its expected CPU burst. Priorities may be **static** (unchanging) or **dynamic** (change during execution).

General Purpose Scheduling

We want to:

- Favour short and I/O bound jobs
    - Provides good resource utilisation
    - Shorter response times
- Determine the nature of the job and adapt to changes
    - Procceses have periods where they are CPU bound and I/O bound
    - May use dynamic priorities for this

Multi-Level Feedback Queues

MLFQs is a form of priority scheduling, implemented by Windows Vista and 7, Mac OS X, Linux 2.6 - 2.6.23.

In MLFQs we have one queue for each priority level, and then we can run the job on the highest non-empty priority queue. Each queue can use a different scheduling algorithm (but we usually use round-robin).

We need to think about the nature of the current job (e.g. give I/O bound processes higher priority), and how to counteract starvation. A lower priority process may never run. To do this we can have a feedback mechanism:

- Job priorities computed periodically based upon how much CPU has been recently utilised (using an exponentially weighted moving average)
- Aging: increase priority as time goes on

However, many OS's have moved on from this:

- MLFQs are not flexible
    - Applications cannot control their priority
    - Priorities make no guarantees - for example, what does priority 15 mean?
- Does not react to changes quickly
    - Needs a warm-up period, as the system runs for a while to get better results

    - This is problematic for real-time and multi-media systems
- Cheating is a concern, applications can add meaningless I/O to increase priority
- Cannot donate priority to other processes

Lottery Scheduling

A more recent form of general purpose scheduling, lottery scheduling distributes lottery tickets to a job for various resources such as CPU time. At each scheduling decision, one ticket is chosen and the job with that ticket wins. The ticket distribution decides priorities. For example, if there are 100 tickets and P1 has 20 tickets, then it has a 20% chance of being scheduled.

- Now, the number of lottery tickets is meaningful, as it indicates the probability of being scheduled.
- New job with p% of tickets has p% chance of getting the resource at the next scheduling decision, making the system responsive.
- No starvation, as tickets are chosen, processes with low ticket counts eventually get chosen.
- Jobs can exchange tickets, to allow for priority donation, so jobs can co-operate to achieve goals.
- Adding and removing jobs affects all jobs proportionally to their priority
- However, we have an unpredictable response time - what if an interactive process is unlucky for a few lotteries?

Tutorial Questions

1. Interactive systems generally **don't** use non-preemptive processor scheduling.
    1. This is because preemptive scheduling guarantees a fast response to new requests, so smaller I/O bound requests are serviced quicker in favour of lengthier requests which may do more computation.
2. Turnaround times are less predictable in a preemptive system.
    1. This is because once a process gets the processor in a non-preemptive system it will run until it finishes or is blocked, hence we don't have the uncertainty regarding continually being preempted by other processes.
3. In priority scheduling, priorities may not be meaningful, even if the kernel honours them.
    1. If we do not make priorities meaningful, then a system which honours the priorities may produce questionable results.

Process Synchronisation

Processes must synchronise their operations when performing tasks. In this context, we may use *processes* and *threads* interchangeably.

Critical Sections and Mutual Exclusion

Definitions:

- **Critical sections of code** are where processes access a shared resource.
- **Mutual exclusion** is a method where if a process is executing its critical region, no other process may enter - they must first request permission.
- **Synchronisation mechanisms** are positioned at the entry and exit points of the critical section in order to protect it.

We have some requirements for mutual exclusion:

1. No two processes may be simultaneously inside a critical section.
2. No process running outside the critical section may prevent other processes from entering.
3. If no process is inside, then any process requesting permission to enter must be able to do so immediately.
4. No process requesting access must be delayed forever.
5. No assumptions can be made about the speeds of the processes relative to each other. (The scheduler may decide to allow one process to make more progress!)

Methods

Disabling Interrupts

We can disable interrupts, meaning the process may not be preempted.

- This only works in single core CPUs.
- It means the process cannot be preempted, which would slow the system.
- If there is a bug in the critical section, then the kernel cannot recover from this as it cannot interrupt it.

As a result, this is a privileged operation.

Peterson's Solution

If we have two processes (for example), then we could assign the first to only enter the region on turn 0, and the second on turn 1. While the thread is not on its turn, it busy waits. After executing the critical section, it changes the turn variable to let the other thread have its turn.

- If the first thread takes a long time in its non-critical region, then the second may finish its non-critical code and then attempt to access the critical section once more. At this point, it will be waiting for the first thread to take its turn.
- Effectively, the assumption that the threads would strictly alternate in their execution of the critical region is not necessarily true. We have broken requirements 3 and 5.

The above approach is **strict alternation**. However, **Peterson's solution** fixes this.

We have a mechanism to enter and leave the critical region. The enter procedure registers the thread's interest in entering, and the leave procedure removes this interest. Then, the thread tries to let the other thread have its turn, and busy waits but only if the other thread is interested.

However, we are still busy-waiting, and ideally, we need mutual exclusion in the kernel to allow a blocking system.

Atomic Operations

One approach is to make the access atomic, so that it is done in one instruction. However, we must ensure that the high level code written is translated into a single assembly instruction.

In fact, in Peterson's solution above, we must ensure that setting the turn variable is atomic.

Lock Variables

When entering, we lock, and when exiting we unlock. The lock mechanism may naively be implemented as keeping track of a boolean, and busy-waiting while it is set to `true` (so that when another thread unlocks, it sets it to `false`) - but we want the locking mechanism to be atomic, which is difficult to do purely using software.

To this end, we can use an atomic instruction provided by most CPUs - `TSL(LOCK)`, which stands for **Test-and-Set-Lock**. It atomically sets the memory location of the lock to **true** and then returns the old value.

Locks which use busy waiting are called **spin locks**. These may appear frequently in the kernel itself, as there is no other mechanism for waiting, such as blocking. They should only be used when the wait is expected to be short (for example, the overhead of a context switch might be longer) . They also incur the **priority inversion problem**. This is where we have two processes - *H* (high priority) and *L* (low priority). Assume *H* is waiting for I/O, *L* acquires the lock, I/O arrives so that *H* is scheduled and *H* tries to acquire the lock. Now, the scheduler will always schedule *H* over *L*, hence causing a deadlock.

To fix this, some form of **priority donation** must be implemented in the kernel scheduler.

Lock granularity concerns the amount of data a lock protects. We may protect an entire data structure or specific parts of it.

Lock overhead measures the cost of using locks:

- Memory space needed
- Initialisation of the lock
- Time required to acquire and release

Lock contention is the number of processes waiting for a lock. More contention results in less parallelism. This is heavily related to granularity: coarser granularity has lower overhead but higher contention and less complexity. Finer granularity has higher overhead and more complexity but less contention.

To minimise contention, we want to use a finer granularity while handling the overheads. We should also release the lock as soon as it is not needed.

If we are only reading the data, and no thread is modifying it, then we do not need locks. If a thread also modifies, then we need to lock all threads. To do this efficiently, we can use **read/write locks**. Once a thread acquires a write lock, it has exclusive access. However, multiple threads can acquire the lock in read mode.

Race Conditions

Race conditions occur when multiple threads read and write shared data, so that the result depends upon the order of execution.

We can reason about possible **thread interleavings**, to find out what outcomes may be possible. There will be a distribution over all possibilities as a result of the scheduling algorithm. Bug may only occur in a certain execution order, which may have a low probability of occurring.

> An aside: memory models
>
> We assume **sequential consistency**, where operations of each thread appear in the same order as in the program. There are also weak memory models, which the compiler or the hardware may assume, which may result in non-intuitive behaviour. In fact, ARM and x86 have differing memory models.

Happens-Before Relationship

Formulated by Leslie Lamport, this gives us a partial order upon the events in a trace. If we have events a, b in a trace, then if a and b are in the same thread, then $a \rightarrow b$. Also, if a is unlock(L) and b is lock(L), then we also get that $a \rightarrow b$.

This relation is not reflexive, it is anti-symmetric ($a \rightarrow b$ means that $b \not\rightarrow a$) and it is transitive.

Now, we can formalise a data race.

A **data race** occurs between a and b iff:

- They access the same memory location
- At least one is a write
- They are unordered according to happens-before.

Locks can help ensure an ordering according to happens-before but there are still multiple executions, depending on, for example, which thread can hold the lock first.

Semaphores

A blocking synchronisation mechanism, where processes stop, waiting for a signal, and continue once it receives it. `down` waits to receive a signal, `up` transmits the signal, and `init` initialises a semaphore with a value.

A semaphore has a counter (non-negative) and a queue of waiters. When calling `down`, if the counter value is greater than 0, then it decrements the counter and continues. Otherwise, if it is 0, then the thread blocks and is added to the waiters.

On an `up`, if the queue is not empty then one process is resumed from the waiters. Otherwise, we increment the counter.

> The down and up operations need to be done atomically or protected by a lock. In the kernel there may be some spin locks protecting these operations.

A binary semaphore, initialised to 1, is effectively a lock, with `down` being called before entering the critical region and an `up` when leaving, meaning that only one process can be in the critical section at a time. The counter value is effectively the number of processes allowed inside the critical region.

If we initialise to 0, then we can ensure an ordering of events. Processes can call `down` to wait for another process to complete a task. This other process should call `up`.

Producer/Consumer With Semaphores

Suppose we have a shared data structure such as a buffer, and multiple producers writing to the buffer and multiple consumers. The producer must only be allowed to write if there is space, and mutex must hold. For consumers, items can be received only if it is not empty, and mutex must hold. The buffer can hold 0 to N items (inclusive).

We can use 3 semaphores: the `item` semaphore is initialised to 0, to ensure the buffer is not empty; the `space` semaphore is initialised to N, to ensure it is not full; the `mutex` semaphore ensures only one write or read is occurring at a time.

A producer then *produces* an item, calls `down` on the `space` and the `mutex` semaphores, then deposits it, calls `up` on the `mutex` and then the `item` semaphores.

A consumer then calls `down` on the `item` and the `mutex` semaphores, then fetches an item, calls `up` on the `mutex` and then the `space` semaphores and finally consumes the item.

This way, the producer may only deposit if the buffer is not full, and if no thread if depositing or fetching, and vice versa for the consumer.

Monitors

Monitors are a higher level synchronisation primitive, which protect some shared data. Only a single process can be inside the monitor at a time. Condition variables are associated with some high level condition. The operation `wait(c)` releases the monitor lock and waits for `c` to be signalled, `signal(c)` wakes up a process waiting for `c` and `broadcast(c)` wakes up all

processes waiting for `c`. In monitors, if a condition is signalled and nobody is waiting, the signal is lost forever - signals do not accumulate unlike for a semaphore.

Producer/Consumer With Monitors

We have conditions `not_full` and `not_empty`. Initialise `count` to 0. When inserting an item, while `count == N` then we `wait(not_full)`. Otherwise, we insert the item, increase the count, and `signal(not_empty)`. For removing an item, while `count == 0`, we `wait(not_empty)`, then remove an item, decrease the count and `signal(not_full)`. Threads should gain the monitor lock before inserting and removing.

The reason that we check the conditions in a loop is because we cannot guarantee that another thread has not modified the buffer in between checking the condition and continuing.

Java implements monitors by utilising the `synchronised` keyword. Only a single thread can call this method, and `wait` and `notify` can be used to signal on an object.

Signalling

We can define signalling so that a process which waits for a signal is immediately scheduled (as per. **Hoare**). However, this is inefficient as the process which signals is switched out and requires modification of the scheduler.

On the other hand, (as per. **Lampson**) we can make it so that sending a signal and waking up is not atomic. This is more efficient and error-tolerant (if the condition is incorrect, it is simply discarded) but needs more care when handling.

Tutorial Question

Two threads in the same process can synchronise using a kernel semaphore only if we are using kernel threads (instead of user space threads), as the kernel must block and synchronise these threads, which is not possible if they are implemented in the user space.

Deadlocks

Example: two processes want to scan a document and then save it to a CD. We can use semaphores: P0 `down`s the scanner first and P1 `down`s the CD first. By not maintaining an ordering for the resources, we end up in a deadlock.

In the dining philosophers example, each thread may successfully pick up the left chopstick and then end up waiting on the right one, in a cycle.

Deadlocks are a form of race condition, so they may not always occur.

Definition: A set of processes is deadlocked if each process is waiting for an event that only another process can cause.

The most common form is resource deadlock. The following 4 conditions must hold:

1. Mutual exclusion: each resource is available or attached to exactly one processes.
2. Hold and wait: process can either request resources while it holds other resources earlier.
3. No preemption: resources given cannot be revoked by force.
4. Circular wait: two or more processes in a circular chain, each waiting for a resource held by the next process.

Tutorial Questions

1. Can a set of processes deadlocked include processes that are not in the circular chain in the resource allocation graph?
    1. Yes. Only some may be in a cycle, but the others can wait upon those in a cycle without being part of the cycle itself.
2. Can a single processor system have no processes ready and no process running? Is this a deadlock?
    1. No. All the processes could be waiting on I/O, and as soon as one completes, it will start running again. Hence we cannot use this to detect a deadlock.
    2. Most architectures require a valid process to execute at all times, hence an idle process is created by the kernel to execute if no others are ready. This process could busy wait, but you want the CPU to be idle. You could constantly yield, but this would also consume CPU cycles (it is a system call), and is unnecessary due to preemption. A special instruction is used to put the CPU core to sleep (but timer interrupts still work for example), which reduces the power consumption of the core.

Resource Allocation Graphs

This is a directed graph: a directed arc from resource to process means the process owns that resource. A directed arc from process to resource means the process is blocked waiting for the resource,

A deadlock is then simply a cycle in this graph. The OS could build this graph and utilise an algorithm to find a cycle.

Strategies for Dealing with Deadlock

1. Ignore it
    1. If contention is low, then deadlocks are infrequent.
2. Detection & recovery
3. Dynamic avoidance
    1. Dynamically consider every request and decide whether it is safe to grant it (needs information regarding potential resource use).
4. Prevention
    1. Prevent deadlocks by making sure one of the 4 conditions for a deadlock is not possible.

Detection & Recovery

We can dynamically build the resource ownership graph. When an arc has been inspected it is marked. We can then do graph traversal: for each node, add the current node to visited/check if it is in there. From the current node, check any unmarked outgoing arc (backtracking if we run out of unmarked arcs). If we get back to the start without breaking, there are no cycles. (This is just a DFS.) However, this is computationally quite expensive.

To recover, we could:

- Pre-emption:
    - Temporarily take resource from owner and give to another process.
    - Relies on the process being able to handle this.
- Rollback:

    - Processes are periodically check-pointed (memory image, state).
    - On a deadlock, roll back to a previous state. (But again we are making assumptions about the program, e.g. if it has taken in IO input, or performed computation, this may be lost.)
    - We need to ensure the scheduling is non deterministic and makes a different choice.
    - Rollback is in fact done for failure recovery, but not for deadlocks.
- Killing processes:
    - Select a random process in the cycle and kill it (using a signal).
    - Many applications are able to gracefully handle being killed.

Dynamic Avoidance: Banker's Algorithm

Proposed by Dijkstra, the idea is similar to how a bank may work. N customers have a maximum credit limit. Each customer may ask for its maximum credit at some point, use it and then repay it. The banker knows that all customers don't need max credit at all points in time, so it reserves less than the max required.

We can apply this to processes. Now we can distinguish between safe and unsafe requests. We want to ensure that there are enough free resources to satisfy any maximum request from some customer. Then, this process can make progress and repay it.

A state is **safe** if there is a sequence of allocations that **guarantees** that all customers can be satisfied. We need to be able to reason about the entire sequence of allocations. The request is granted only if it leads to a safe state. An unsafe state may not lead to a deadlock, but a safe state guarantees that there is not a deadlock.

Tutorial Question

Consider the following system:

Proc	A	B	C	A	B	C
P1	0	1	0	7	4	3
P2	3	0	2	0	2	0
P3	3	0	2	6	0	0
P4	2	1	1	0	1	1
P5	0	0	2	4	3	1

The first three columns are the currently allocated resources, and the last three are the requested resources. We have: 2 A, 3 B and 0 C available instances of each resources. Is this safe?

- This is safe since we can allocate 2 B to P2, giving us 5 A, 3 B and 2 C. Then allocate to P4, so we get 7 A, 4 B, 3 C. Then allocate to P1, so we get 7 A, 5 B, 3 C. Then allocate to P3, so we get 10 A, 5 B, 5 C. Finally, allocate to P5, so that all processes finish.

Prevention

We can prevent one of the following conditions:

- Mutual exclusion: we could share the resources

- Hold and wait: processes request all the resources they will need before they start, and if any one is not available then it waits.
    - However most processes may not know all the resources it requires, and it makes fine grained concurrency difficult.
- No-preemption: we can revoke the lock, but now shared data structures may be in an inconsistent state (for example, if a process must give up the printer half way through).
- Circular wait:
    - We could force single resource per process, but this is not optimal.
    - We could enforce an artificial ordering for resources. For a large number of resources this can be difficult to organise, but is otherwise practical.

Communication Deadlock

Communication deadlock occurs when A sends a message to B, and blocks waiting on its reply, but message loss means that B is blocked waiting on A's message, creating a deadlock. The strategies above don't work. For communication protocols, we can implement timeouts.

Livelock

This is where processes and threads are not blocked but they are also not making progress. As an example, lets assume a scenario where processes would normally deadlock, but the processes, instead of blocking, abort the attempt and retry. Hence there is activity in the system but both processes are continually attempting to acquire the resource.

Another example is where a process which receives incoming messages and another processes them. The processing thread has a lower priority and if there is a high network activity then it never runs. This is a **receive livelock**.

Starvation

Starvation is a measure of unfairness: policy is biased against some specific thread. Suppose many jobs want a printer - a policy may be to give it to the smallest file, which gives a better turnaround, but a very large file may never have access to the printer.

Memory Management

- Every instruction cycle involves memory access.
- We need **allocation** and **protection**.
- If a process does something invalid, then we want it to be unable to interact with other processes

We cannot assume:

- How memory addresses are generated.
- What the memory address is used for (instructions or data).

Most bugs which occur (unintentional or malicious) are due to memory related bugs. Hence we need to have a layer of uniform abstraction for processes. This is usually done through **virtual memory**.

Memory Hierarchy

The memory hierarchy consists of registers, L1 cache, L2/L3 cache, main memory, disk/ssd and the network. Moving up a layer consists of an increase in orders of magnitude, but increases the amount of data which can be stored.

	Registers	**L1**	**L2/L3**	**ram**	**ssd**	**hdd**	**network**
Latency	< 1ns	1ns	10ns	100ns	100µs	1ms	1s
Size	Bytes	KB	MB	GB	TB	TB	PB

Logical and Physical Memory

Memory management binds **logical addresses** to **physical addresses**.

- Logical addresses are generated by the CPU and is the address space seen by the process.
- Physical addresses are the addresses seen by the memory unit.

They are the same in compile- and load-time address-binding schemes, and different in execution-time address-binding schemes.

In some systems, such as embedded systems, there may be a one-to-one mapping (they are the same). In modern OSs, there is a run-time translation (usually done by virtual memory with paging).

This translation is typically done by the MMU as it must be fast. A simple method might be to simply add an offset to a logical address inside the MMU.

- This would constrain the logical address space to a certain subset of the physical space.
- At run time we can decide where the memory is actually allocated.
- Programs do not need to be written in a way which has knowledge of other processes - all programs can use the same logical space without conflicts in the physical space.

Contiguous Memory Allocation

Main memory is usually split into:

- **Kernel memory**
    - Needs to be protected except in kernel mode.
    - Typically in low memory, as it may be assumed that some specific interrupt vectors live at a specific memory address.
- **User memory**
    - Used by processes.
    - Held in high memory.

We can have **contiguous allocation** with **relocation registers**:

- **Base register** contains value of the smallest physical address.
- **Limit register** contains value of the available range of the addresses.
- Now we can check that memory access is valid by checking that it is inside this range.
- What is a process wants to allocate new memory? We can't necessarily extend the limit because another process may occupy the memory. We would have to move processes in order to extend them.
- Furthermore, the amount of processes is dynamic, so we would have "holes" in the memory map (**fragmentation**). We can allocate memory for a process if the hole is large enough.

    - First-fit: we allocate the first hole that is big enough
    - Best-fit: allocate the smallest hole that is big enough
        - Now we have to search the entire list.
        - But produces the smallest leftover hole.
    - Worst-fit: allocate largest hole
        - Must also search the entire list.
        - But produces the largest leftover hole.

Fragmentation comes in two different types:

- **External fragmentation**: total memory exists to satisfy the allocation request, but it is not contiguous.
    - Reduce this by **compaction**: move memory contents to place all free memory together in one block.
    - Leads to I/O bottlenecks.
- **Internal fragmentation**: allocated memory larger than requested memory, hence there is a chunk of memory which is not actually used by the process.

This type of memory allocation scheme is utilised, for example, in embedded systems when the set of processes is known before hand, so we can plan ahead since we know the memory required for each process.

Swapping

When a process is not running, it does not need to be in main memory. Hence, we can **swap** processes temporarily out of memory to disk. There is some **swap space** (a file or partition on disk). Transfer time is a major consideration in this. Hence, if there is a lot of swapping, there will be a large performance impact.

Virtual Memory with Paging

To deal with fragmentation, we treat memory as being subdivided into a set of fixed-size pages. Only part of a process needs to be in memory for execution, hence the logical address space can be much larger than the physical space. Now, we require a memory map so that pages in virtual memory are mapped to the pages in the physical memory.

In 64 bit machines, a virtual memory address is represented using 64 bits. The process can decide upon a memory layout but the operating system has full flexibility over how the process's memory is actually laid out in memory.

Paging

- **Frames** are fixed size blocks of physical memory. The OS keeps track of all free frames.
- **Pages** are blocks of the same size of logical memory.

To run a program of size n pages, we find n free frames and load the program. Then we need to set up a **page table** to translate logical to physical addresses.

Pages and frames have the same size (usually 4KB).

Tutorial Question

1. What is the advantage of using a small page size?

    1. Provides more granularity - if the process allocates a lot of small memory sizes, then we have less unused space.
    2. However, now we need a larger page table.
    3. A larger page size would mean less overhead for address translation and faster memory access.
2. How does a context switch affect the virtual memory system?
    1. A virtual address space is specific to a given process, hence on a context switch it must also change the page table for that process. It must change the base register in the MMU to point to the page table in memory. It must also clear any invalid cached address translations from the **TLB** (explained later).

Address Translation

An address generated is divided into:

- The **page number**, p (m - n bytes)
    - Used to index into the page table
    - Page table has base address of pages in physical memory
- The **page offset** , d (n bytes)
    - The byte offset into the page

For a virtual address [p | d], we look up p in the page table to get f, and then the physical address is [f | d].

We also need to maintain a list of **free frames**. This is a list of frames which are not mapped to any pages. The OS will use this to set up a new page table for a new process.

Memory Protection

Associate **protection bits** with pages. A **valid-invalid bit** is attached to each page table entry:

- **Valid** indicates a legal page. (LSB set to 1)
- **Invalid** indicates page is missing. (LSB set to 0)
    - Page is not in process' logical address space (page fault)
    - Need to load page from disk (demand paging)
    - Incorrect address (programmer error)

We can implement this information compactly: in the page table, the offset bits are not used as they are supplied by the logical address. In a 64 bit machine, the frame addresses can be stored in much less than 64 bits, since physical memory is much smaller. Hence these bits can be used for the protection bit and a modified (dirty) bit.

Tutorial Question

Suppose we have the following page table for a 16-bit big endian architecture with a 1KB page size:

`[0] -> 0x2C00 => 001011 0000000000`

`[1] -> 0x0403 => 000001 0000000011`

`[2] -> 0xCC01 => 110011 0000000001`

`[3] -> 0x0000 => 000000 0000000000`

`[4] -> 0x7C01 => 011111 0000000001`

Then, what are each of these virtual addresses translated to?

1. `0x0B85` is `000010 1110000101 => 110011 1110000101 => 0xCF85` (valid access)
2. `0x1420` is `000101 0000100000` which is a page fault as page 5 does not exist
3. `0x1000` is `000100 0000000000 => 011111 0000000000 => 0x7C00` (valid access)
4. `0x0C9A` is `000011 0010011010` which is a page fault as page 3 is marked as invalid in the page table.

Page Table Implementations

Page tables are kept in main memory, with the **Page-table base register** pointing to page table, and the **Page-table length register** indicating the size. These registers must be flushed and updated on a context switch.

Translation Look-aside Buffer

We also need to consider the efficiency of a page table: each data/instruction now requires two memory accesses. The solution to this is a form of **associative memory**: a fast-lookup hardware cache is utilised known as the **Translation Look-aside Buffer**, which actually allows for a parallel search across all entries by the hardware (effectively constant time). For a virtual address (p, d), we first check if p is in the associative memory. This cache must normally be flushed on a context switch.

On a system call, one approach is to ensure that the kernel pages live in the same address space as that of the current process, so that the TLB does not need to be flushed on a context switch.

Effective Access Time

The effective access time is: $t = (\epsilon + 1)\alpha + (\epsilon + 2)(1 - \alpha) = 2 + \epsilon + \alpha$, where ϵ is the associative lookup time, and memory cycle time is assumed to be $1\mu s$. WE also have α which is the hit ratio. A high hit ratio minimises the overhead of the TLB lookup.

Types of Page Tables

In order to reduce the size of a page table, several techniques have been introduced.

1. Hierarchical Page table
    - On a 32-bit machine with 4KB pages, the page table will be 4MB (at least). However, on a 64-bit machine with 4KB pages, then we require 2^{52} entries, which gives us a size of 30 million GB.
    - Even though we have a large virtual address space, most processes will only require a very small section to be mapped.
    - Hence we can introduce gaps.
    - We break up the logical address space into multiple page tables. A simple technique would be a two-level hierarchy, with an outer page table and an inner page table.
    - We only need to allocate an inner page table if there is a portion of it which is a valid mapping.
    - To allow this, we can split the first p bits into p1 and p2, where p1 is an index into the outer page table and p2 is an index into the inner page table.
    - We can have 3 or 4 level paging.
    - We have reduced the amount of memory required but increased the amount of lookups (higher access time). The TLB cache needs to be more effective.
2. Hashed/Inverted tables

- We can make the page table grow with frames instead of pages.
- However, how can we do constant time access?
- Hashed tables work by taking the logical address and the physical address and hashing them to lookup in a hash table.
- This is harder to implement in the hardware, but the hash table grows with the number of mapped addresses.
- An inverted page table grows in terms of physical memory but lookups are now a linear search. For smaller address spaces this is easier to implement in hardware.

Tutorial Questions

1. What is the access time for a 4-level paging system, with a TLB hit ratio of 80% and 98%. Memory access takes 100ns and TLB access takes 20ns.
    1. The time for translation is given by: $0.8 \times 120 + 0.2 \times 520 = 200$ns, since in a miss, we do 1 TLB access and 5 memory accesses. If the hit ratio is 98%, we get 128ns, which is 28% slower than unpaged (and 22% slower for a 1-level paging system).
2. A paging system uses a 3-level page table. Virtual addresses are decomposed into fields a, b, c, d, where d is the offset into the page. What is the maximum number of pages in a virtual address space?)
    1. We can have $2^{(a+b+c)}$ total, since there are $2^{(a+b+c+d)}$ total bytes, and 2^d bytes in each page.

Shared Memory

Processes can set up shared memory areas. The page tables can point to the same physical frame, and hence they can write to the same area of memory to allow for communication.

- This does not require intervention from the kernel and hence it is much faster and efficient to communicate.
- We can use a system call API: `shmget, shmat, shmctl, shmdt`, which can allocate shared memory, attach it to an address space, change associated properties, and detach.
- We have higher throughput compared to pipes, but we can also have bi-directional communication. There is no synchronisation from the kernel for uni-directional communication as there was for pipes (where processes would block to wait on the pipe).
- To achieve a similar effect, we can treat the two processes as being two threads in the same memory space, so we can utilise locks and other synchronisation primitives.
- Shared memory allows complex data structures such as pointers too be utilised.
- Pipes are effectively an abstraction of a file, making them more flexible.

Shared memory can also map files - virtual memory with paging can associate a mapping with a given file on disk. Mapping files into the address space of processes is an important feature of operating systems - dynamic linking of libraries relies on this mechanism. One library can be linked into the address space of many different processes (.o files on Linux, .dll files on Windows).

Case Study: Linux Virtual Memory Layout

In Linux, there is "low physical memory" which is where some system critical components are stored. I/O devices can also directly write to low memory without having to interface with the OS.

On a 32-bit machine, we have a 4GB address space. In virtual memory, user processes are mapped from 0 to 3GB, and 3GB above is reserved for the kernel for every process. This is done to reduce the cost of system calls.

Kernel code is reasonably small. With the remaining space, it tries to map as much of the physical memory into its space as possible, so that it can access significant amounts of physical memory directly. 896MB are directly mapped, and after this we have "on-demand mapping," where the kernel updates the page table of the process to create the mapping needed to refer to that physical memory, on-demand.

With 64-bit address spaces, we can directly map all of the physical memory into virtual memory, removing the need for on-demand mapping. A large address space is also more secure: the OS picks random locations for data and libraries, which means it is harder to attack, as it is more difficult to find where the libraries live in memory (scanning the address space is much easier in 32-bit).

On IA-32, there is a 4KB page size, 4GB virtual address space.

On x86-64 pages are larger (e.g. 4MB) especially for systems which allocate much more memory, such as database systems, and the page table is up to 4-levels. In x86-64, the outer page table is known as the page global directory. There is at least a 2-level page table. Offset bits contain the page status (dirty, read-only etc).

> In fact, due to the Meltdown attack, the Linux kernel now actually put the kernel memory in a separate virtual address space. This has caused a performance hit, as system calls now require a TLB flush and page table switch.

Demand Paging

The idea is that we only bring pages into memory when we need them. This means less memory is needed, response time is faster (especially when launching binaries), the system can support more users. Now, on a page access, if it is not in memory, we load it from the file.

- This is done through the valid/invalid bit in the offset, where `1` means the page is in memory.
- When the valid bit is not set, a page fault is triggered.
- Then, the kernel must decide the reason for the fault. If it was due to not in memory, it retrieves an empty frame, swaps the page into frame, resets the tables and sets valid bit to `1`, before restarting the last instruction.
- There is a large overhead the first time a page is brought into memory.

Performance:

- Let the page fault rate be $0 \leq p \leq 1$. Then the effective access time e is given by e = (1 - p) * mem. access + p({page fault overhead} + {swap page out} + {swap page in} + {restart ins overhead}).

If we don't have a lot of physical memory then page faults will occur a lot. The system may become slow due to increased I/O activity, where the disk is used as an extension of memory. This is known as **thrashing**.

With demand paging we also have:

- Copy-on-write
    - Allows parent and child to initially share the same pages in memory.
    - If either process modifies the shared page, trigger a page fault, and handle it by copying the page, and allocating a new page for the process which modified it, allowing

it to write to it.
    - This allows `fork()` to return very quickly.
- Memory-mapped files
    - Map file into virtual address space for paging
    - Simplifies programming model for I/O
    - Speeds up access
    - e.g. Memory map a 100GB file into memory, but no bits are touched. Access cost only occurs on the small portion of the file which is accessed through demand paging.

Page Replacement

If there is no free frame, then we can replace the page - find some unused page in memory to swap out.

Strategy for **page replacement** must:

- Minimise the number of page faults
    - Replace a page that will not be used ideally.
    - So avoid putting the same page into memory several times.
- Prevent over-allocation of memory
    - Page-fault service routine includes page replacement
    - We must have enough pages available to allocate new memory
- Use modified/dirty bits to reduce overhead of page transfers
    - If the page has been changed in memory, write it to disk as well.

A basic strategy:

- Find location of desired page on disk
- Find a free frame.
    - If a frame is found, use it.
    - Otherwise, use replacement algorithm to select a **victim frame.**
    - Read desired page into newly freed frame.
    - Update page and frame tables.
    - Restart the process.

To pick the victim frame:

- Pick the frame which will cause the least page faults.
- **Reference string**: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5...
    - These strings show the memory accesses, and are generated randomly or by analysing a system.
    - An algorithm's performance is then based on how well it does on these reference strings.
- The more frames we have, the less page faults.

Simplest strategy: **First-In-First-Out**

- Replace the oldest page.
- May replace a heavily used page in the future.
- Suffers from Belady's Anomaly: with more frames, we might actually have more page faults (eventually it will go down).
- Not used in practice!

What is the optimal algorithm?

- Replace the page that wont be used for the longest period of time.
- Can't do this in practice, but assuming we can analyse the reference string, then we can devise a strategy, where we replace the page that won't be used until latest in the future.
- Useful for measuring how well a page replacement strategy performs.

The most common class of practical algorithms is **LRU (Least Recently Used).**

- Each page entry has a counter.
- When a page is accessed, the current hardware clock is copied into the counter.
- We then check which page as the lowest counter (used longest ago).
- However copying the counter is expensive, so we can use approximations.
- Reduce the counter to a bit (reference bit).
- If it has been accessed, set it to 1, otherwise it is 0.
- Periodically set the reference bits to 0.
- Hence we can create a set of pages which has not been recently used.
- Although there is no total ordering, we can achieve this efficiently in hardware.
- The reference bit can be stored in the offset bits.

The second chance algorithm:

- Order the pages by when they were first brought into memory (using a linked list).
- We consider these pages in turn.
- We check whether each page has been recently accessed in turn by checking the reference bit, giving the page a second chance - although pages may have been brought into memory a while ago they may still be active.
- The linked list is a circular queue - by the time we loop around, some of the reference bits should have been reset.
- Easy to implement.

We can also have algorithms which take into account how many accesses have occurred to a page (counting algorithms).

- **LFU: Least Frequently Used**. Replace a page with the smallest access count.
    - The issue with this is that newer pages will be the obvious selection, so we need a mechanism to not select earlier pages.
    - Pages with heavy usage are also unlikely to be "forgotten". We could reset the counters to 0 but this would lose information. Solution is to have an ageing mechanism.
- **MFU: Most Frequently Used**. Although counter intuitive, replacing the page with the largest count this may work due to how software generally works, and working sets (covered below).

Locality of Reference

Programs have a property that there is a set of pages which are accessed repeatedly. Programs tend to access the same set of pages in space and time. Effectively, there are vertical blocks of references across memory in certain chunks of the execution.

This is due to for example, traversal over an array. This accesses contiguous memory. There also is often a data structure which is repeatedly used for a period of time, giving temporal locality.

Locality of access gives better performance with virtual memory with paging, as touching the same addresses repeatedly as it will be in TLB. The data will also be in cache, meaning main memory does not need to be used. Furthermore, within the same page, a page fault will not occur.

For good performance, the system must maintain the program's favoured subset of pages in main memory, otherwise thrashing occurs, where excessive paging activity means the processor is not utilised, as pages are repeatedly used from secondary storage. When memory runs out, thrashing occurs as a page that is being used must be swapped.

In cloud computing, the limiting resource is memory as opposed to the CPU, due to the large number of CPUs, and the programs requiring a lot of pages to be in memory to make progress.

Working Set Model

The **working set** `W(t, w)` is the set of pages that is referenced during the interval of time `t - w` to `t`.

This set changes over time, as it is dependent on what the process is doing, and the process transitions between working sets. The OS temporarily maintains pages outside of the current working set in memory. The memory management must reduce misallocation and ensure that the working set can be in memory.

Page Replacement Using Working Set

We can add the time of last use to the clock replacement algorithm. Then, we can decide whether the time of last use falls inside the window of the working set, then it is needed. Otherwise, it is no longer part of the current working set and can be replaced. If page is dirty, also write-back to memory.

If we assume the working set is much lower than what is needed, then thrashing will occur. Hence time intervals between paging faults will be very short. Increasing the working set will decrease the number of page faults, but it will converge asymptotically. Once we overestimate the working set, we have diminishing returns. We can start giving the process more memory until we start seeing a decrease in the rate of the interfault time/page faults.

Global Strategy

Fully allocate all available memory, and use page fault frequency to move excess memory around - "globally optimise it". (Used by Linux.)

Local Strategy

Each process gets fixed allocation of physical memory. We need to detect changes in working set size to change this size. (Used by Windows.)

Linux Page Replacement

- Uses a variation of the clock algorithm to approximate LRU page-replacement strategy.
- Two linked lists + reference bits.
- Active list contains active pages, with most recently used pages near head. Pages in the working set should be in this list.
- Inactive list contains inactive pages, with least-recently used pages near the tail.
- `kswapd` is a swap daemon so that pages in inactive list are reclaimed when memory is low, using a dedicated swap partition or file. Must handle locked and shared pages.
    - Asynchronously and continuously runs in the background, so that memory can be reused when it is needed.
    - Runs when the machine is idle/spare cycles are available.
    - On a `malloc`, a large amount of pages may be allocated.
    - Don't want all the physical memory to be allocated, as `malloc` will be slow.

    - If a page hasn't been modified, we can simply use it as is without swapping (already exists on disk).
- `pdflush` is a kernel thread which periodically flushes dirty pages to the disk. Allows pages to be easily evicted without having to write.
    - Also runs when the machine is idle in terms of I/O load.

Tutorial Question

Suppose we have the following reference string for how pages are referenced in a virtual address space:

`1 2 1 3 2 1 4 3 1 1 2 4 1 5 6 2 1`

There are 3 empty frames available. Assume paging decisions are done on demand (when a page fault occurs).

What are the contents of the frames after each memory reference, under a LRU replacement policy, and then a second chance clock policy. How many page faults in either case?

1. LRU Replacement Policy:

   `REF: 1 2 1 3 2 1 4 3 1 1 2 4 1 5 6 2 1`

   `FR0: 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2`

   `FR1: - 2 2 2 2 2 2 3 3 3 3 4 4 4 6 6 6`

   `FR2: - - - 3 3 3 4 4 4 4 2 2 2 5 5 5 1`

   `PF?: Y Y N Y N N Y Y N N Y Y N Y Y Y Y`

   11 page faults.
2. Second Chance Clock Policy:

   `REF: 1 2 1 3 2 1 4 3 1 1 2 4 1 5 6 2 1`

   `FR0: 1 1 1 1 1 1 4 4 4 4 4 4 4 5 5 5 5`

   `FR1: - 2 2 2 2 2 2 1 1 1 1 1 1 6 6 6`

   `FR2: - - - 3 3 3 3 3 3 2 2 2 2 2 2 1`

   `PF?: Y Y N Y N N Y N Y N Y N N Y Y N Y`

   9 page faults.

Device Management

We need to manage I/O devices. They are connected to the CPU through an I/O bus. There are two bridges, the north bridge and the south bridge, which all I/O devices connect to. The closer an I/O device is to the CPU the higher bandwidth it has available. For example, the graphics card needs more bandwidth to communicate with the CPU and DRAM, hence it is connected to the north bridge through multiple PCIe lanes. Devices connected to the south bridge are slower.

Objectives:

- Fairness in access to shared devices.
- Exploit parallelism for I/O devices.
- Provide an abstraction of I/O devices, hiding the complexity of device handling.

Device Independence

- Independence from the device type (such as a disk (HDD/SDD) or DVD), done by a device driver.
- Independence from the instance (e.g. which disk in the machine).
- Inside a device class we have variations:
    - Unit of data transfer: **character** (keyboard, operate on single piece of data) or **block** (hard-disk, operate on large piece of data).
        - Using a bigger block size as opposed to a single character for some devices is due to performance. Even if a disk exposed a single character, it would require a large number of requests to read a lot of data, whereas the rate of data fetched by a keyboard is comparatively slow.
        - When you have block, you can also cache blocks in the buffer cache for faster access, which is not necessary for character devices.
    - Supported operations: read, write, seek
    - Whether operations are sync or async.
    - Speed differences in devices (bandwidth)
    - Shareable (such as disks) or single-user (such as printer, DVD)
    - Types of error conditions:
        - Some devices, such as magnetic HDDs may develop faults.
        - Others, such as mice do not report errors.

I/O Layering

- Inside the hardware there are several **controllers** which can interface with the device.
- Between device drivers and hardware is an interrupt handler. The device, for example if it is asynchronous (a network card receiving a packet), will gain attention from the CPU by invoking the interrupt handler, so that the driver may read the information. The interrupt handler will then call the appropriate device driver procedure.
- The driver is inside kernel space (as it must be privileged to interface with the controller) and is a piece of software which knows for example which hardware registers the controller exposes.
- The operating system then attempts to unify the interfaces exposed by the drivers into a uniform abstraction at the system call level to be utilised by the userspace.

Interrupt handlers

- Process interrupts
- For block devices, an interrupt will occur after a block of data has been processed. Then the next block can be requested from the CPU.
    - Block devices may be further sped up by interrupting after a set number of blocks have been processed.
- Character devices interrupt for each character that has been processed.

Device Drivers

- Implemented by the manufacturer.
- Handles one device type, but may be able to handle multiple instances.
- For a block device, it knows how to read/write blocks
- Access device registers

- Initiate operations
- Schedule requests for some devices
    - For disks, concurrent requests may be received, so needs a scheduling policy
- How to handle errors, for example in storage devices with faults, they may try to write to a different location on the disk.
    - Only when the error can't be dealt with will the driver report the error to the OS

Device Independent OS Layer

- Needs to provide a standard interface for driver developers.
- A device driver may make arbitrary decisions on how to interface with the OS - the disk driver may directly interface with the disk management of the kernel.
- However this makes driver development difficult.
- The kernel may provide:
    - Mapping logical to physical devices (related to naming). Processes should not refer directly to physical devices in the system, as it requires assumptions of the user's system. The hardware may change, hence this strategy would break the application. Disks are, for example, C:\, D:\ or /dev/sda
    - Request validation - for a USB input device such as a mouse for example, a write request may be rejected.
    - Allocation decision - devices which require exclusive access for example
    - Protection and user access validation
    - Buffering of blocks (so the cache can be used across devices)
    - Error reporting interface

Dedicated and Shared Device Allocation

- For a dedicated device, a process must try to gain access
    - Simple policy is to reject requests if it is already being accessed.
    - Otherwise, use a queue.
    - May be allocated for long periods.
- For a shared device such as disks, window terminals, etc.
    - The OS provides abstractions such as a file system to support concurrent access.
    - Allows sharing safely, by having permissions, directory structures, so processes do not need to interface with blocks directly.
- For devices which cannot be shared easily, use **spooling**.
    - Two devices cannot write to the printer at the same time for example.
    - Blocking user access causes delays and bottlenecks.
    - Spooling is done to an intermediate medium (such as a file on disk)
    - File is later printer by a **spooler daemon**
        - Printer is then only allocated to this daemon
        - Normal processes are not allowed access
    - This reduces I/O time and gives greater throughput
    - Processes have the illusion that the may send data to the printer at any time.

Buffered I/O

- Buffered I/O - all of the I/O requests go through the buffer cache (default):

    - Output: use data transferred to OS output buffer, and the process continues and only suspends if the buffer is full.
    - Input: OS reads ahead, with reads normally satisfied by the buffer. Process blocks directly when the buffer is empty.
    - Used to smooth peaks in I/O traffic.
    - Caters for differences in data transfer units between devices.
- Unbuffered I/O - data transferred directly from user space to the device
    - Each r/w causes physical I/O
    - Device handler used in each transfer
    - Not having the buffer may reduce latency
    - If we know that the data being read is never being touched again then we may not want to pollute the cache
    - A database system may use unbuffered I/O since transactions must be durable and consistent, and must be written to disk (no guarantee when the OS will flush the cache). The database already also caches, so another layer of caching will decrease performance.

User Level I/O

- I/O operations are: open. close, read, write, seek.
- APIs can be synchronous or asynchronous, blocking or non-blocking. The application should decide which one is useful.
- In UNIX, everything is a file, including devices.

Direct Memory Access

We can map registers of the I/O device into memory locations. The MMU knows whether an address belongs to the I/O device. For example, on a Raspberry Pi, the clock has a special address.

With an abstraction like this, we can do **programmed I/O**: repeatedly read from the memory location to check for updates. However this is inefficient, we want the device to interrupt - **interrupt driven I/O**. If we have a lot of interrupts, this is also inefficient, so we can use **DMA**: where the device uses *direct memory access* to move data to main memory without going through the CPU.

A disk could use DMA to transfer data to memory, by using a DMA controller as a bridge.

- CPU programs the DMA controller as to what data should be read from which device at which main memory address.
- The DMA controller initiates the request with the disk controller.
- Disk controller directly transfers the data to main memory.
- Once the entire transfer has complete, it sends an acknowledgement back to the DMA controller. Then it raises an interrupt with the CPU.
- DMA controllers could be complex, by writing different blocks to different memory addresses for example.
- GPUs, network controllers etc. all have DMA capabailities.

Types of I/O API

Blocking I/O:

- I/O call returns when the operation is completed.
- Process is suspended, so I/O is "instantaneous" from the point of view of the process.

- Easy to reason about, but if any other tasks need to be done, multi-threading must be employed.

Non-blocking I/O:

- I/O call returns with whatever is available (for example, read may return with 0 bytes read). In Linux, this is why `read` and `write` return the amount of data read or written, due to the distinction in blocking and non-blocking semantics.
- We switch from blocking to non-blocking I/O using `fctnl` system call on a file-descriptor.
- Now, application level polling must be done for I/O operations. The application must periodically retry. This may be difficult, for example busy waiting while waiting on an I/O operation. However, now concurrent I/O operations are possible in a single thread.

Asynchronous I/O:

- Process executes in *parallel* with the I/O operation
  - No blocking in interface procedure
- I/O subsystem notifies the process upon completion
  - Using a callback function, or process signal
- Using a blocking or non-blocking API, there is no support for notification of completion from the OS.
- The application issues an `aio_read()` system call, which immediately returns, so the application can execute other code. There will then be a movement of data from kernel space to user space once the call has been completed.
- This is flexible and efficient, but harder to use and initially less secure.
  - Now the application does not need to continuously poll.
  - Must interrupt the process in a safe way: one way is a signal, as an IPC mechanism.
  - Another way is to use a separate thread, which is a dedicated thread for receiving the async. notification.
  - Malicious processes could provide incorrect callback data, so the kernel needs to validate.

Case Study: Linux

Device Drivers

In Linux, **Loadable Kernel Modules** provide device drivers, containing object code loaded on demand. They are dynamically linked to the kernel, provided by hardware vendors or kernel developers.

Binary compatibility is required: the driver must be compiled for the Linux kernel version. Linux distributions must ensure that 3rd party drivers are compatible, or compile drivers from source against the kernel headers.

Kmod is a kernel subsystem managing modules without user intervention.

- Can load modules on demand, if you plug in a new device.
- Knows about module dependencies, so loads all dependencies as well.

A kernel module has two functions at a minimum: `init_module` and `cleanup_module`. The other functions depend on the type of module you are loading. Due to the monolithic design, the kernel module can call into the internal API. The module is loaded by `insmod module.o` (restricted to root, as it is a privileged operation).

I/O Management

Kernel provides common interface for I/O system calls. Groups devices into device classes performing similar functions. Classes are managed differently to account for different features.

Major and minor identification numbers: a major number refers to a type of device, mapped to a device driver. Multiple instances distinguished by different minor numbers.

Major numbers are separate for block and character devices. `/proc/devices` will list the current registered devices. For example, for block we may have hard-disks, ramdisk. For character devices we may have USB devices, input devices, memory itself (since memory is byte addressable), terminal devices. Memory is the character device with major number 1.

Devices are represented as files: entries in `/dev` contains files which represent a given instance of a device. Looking at the permission column we see a `c` or `b` in the first position, representing *character* or *block*.

Now, interacting with a device is the same as opening the file and writing to it. The OS will then use the major number to handle the request.

Device Access

There is a **virtual file system (VFS)** which redirects system calls to the correct device driver. Most drivers implement common file operations such as read, write and seek.

Some operations cannot be captured through the file abstraction, such as ejecting the disk. Hence, there is a system call `ioctl`, which sends special commands, e.g. `ioctl(cdrom, CDROMEJECT, 0)`. Receiving status info. from a printer also falls into this.

Character Devices

Transmits a stream of bytes, represented by a `device_struct`, containing the driver name, the pointer to the driver's `file_operations` structure. Registered drivers referenced by a `chrdevs` vector. The file ops struct maintains operations supported by the driver (read, write, seek, close etc.).

This allows us to immediately check whether a device supports and operation.

Block Devices

There is a block I/O subsystem, which contains a number of layers. We can place common code and modularise. There is a caching layer to minimise the time spent accessing block devices. I/O operations may be clustered or reordered for better performance. Need to handle speed differences in block devices.

If a block exists in the block cache, it can immediately be retrieved. Otherwise, add to a queue. Direct I/O is where the driver bypasses the kernel cache directly.

I/O Classes

We have different types of devices:

- Character: files and devices
- Block: devices
- Pipes: inter-process communication
- Socket: network interface.
    - Allow bidirectional communication
    - Can be used to exchange information locally and also across a network

- While pipes have to be identified by machine specific file descriptors, sockets can work across machines
- TCP (stream sockets) and UDP (datagram sockets) are two types.

I/O API

I/O calls are now simply: `create(filename, permission)` returning a file descriptor for reading and writing, and `open(filename, mode)`. We also have `close(fd)`, `read(fd, buffer, numbytes)`, `write(fd, buffer, numbytes)`. Some I/O classes have special calls, such as `pipe(&fd[0])`, `dup(oldfd)`, `iotl(fd, operation, &termios)` and `mknod(filename, permission, dev)` which creates a special device file, for example, a character of block device (for example those under /dev).

File descriptors are handles represented as numbers. Each process has a **file descriptor table** with 3 file descriptors by default: `0 - stdin`, `1 - stdout`, `2 - stderr`. By default, the refer to the terminal where the program was started.

For AIO, we use a control structure which contains various parameters such as the `fd`. Then, use the `aio_read(&structure)` call. We can use `aio_error` to check whether the call is in progress, finally using `aio_return` to check success (or otherwise register a callback to process the result).

Tutorial Question

In which of the 4 I/O software layers (user-level, device independent, device drivers and interrupt handlers) is the following done:

1. Computing track, sector and head for a disk read
    1. Device drivers, but could also be done by the disk controller, as it is a low level abstraction.
2. Maintaining a cache of recently used blocks.
    1. Done by the device independent software, as the buffer cache is reused across block devices.
3. Writing commands to the drive registers.
    1. Device driver knows about the registers, so if updating the registers takes more time or is specific to the device, then it should be done by the driver.
    2. The interrupt handler when it is handling specific interrupts may also do this, if it is a quick and time critical operation.
4. Checking to see if the user is permitted to use the device
    1. In the device-independent OS software since it can be uniformly applied across all devices.
5. Converting binary integers to ASCII for printing, or ASCII to UTF
    1. How data is encoded in the disk is low level, so might be done in device driver or controller.
    2. However in general this would be done in the user-level API, as it does not require privileged mode.

Disk Management

History of disks

- Disk capacity has grown exponentially, by increasing the density of storage cells.

- However access speeds have not improved at the same rate (at best linearly).
    - Hard-disks are mechanical devices, so there are physical limitations due to the speed at which mechanical parts can actually move.
    - For example, the read-write head is a mechanical device which needs to move to the location at which it needs to read.
    - SSDs have an advantage in that mechanical parts are not used, so access speeds can be used.
    - However, for large amounts of storage, hard-disks are more cost effective.
- This poses a challenge: reading TBs of data is slow, but disks can contain that amount of data.
- There are new storage technologies which will allow capacities to continue improving in the future.
- Important as amount of data which is generated (e.g. Youtube) is also increasing exponentially.

Structure of a hard-disk

- A hard-disk has magnetic platters which allow storage of data, and read-write heads which can read from the surface of the platters.
- The surface of the platters has tracks split into sectors.
- Tracks on top of each other are known as a cylinder: all information stored as part of a cylinder can be read in one go.
- An old floppy disk is still magnetic media and has the same organisation (floppy vs hdd):
    - Cylinders: 40 vs 16,000
        - The tracks are spaced much closer together and so a HDD can fit many more tracks.
    - Tracks per cylinder: 2 vs 16
        - 2 tracks per cylinder since we have two surfaces: one on the top and the bottom.
    - Sectors per track: 9 vs 63
    - Bytes per sector: 512 in both cases
    - Sectors per disk: 720 vs 781,422,768
    - Capacity: 360KB vs 1TB
- We are reaching physical limits with how close we can put tracks: magnetising a part of the track may magnetise tracks close by.
    - HAMR: Head Assisted Magnetic Recording
    - Materials that can be magnetised, upon heating, allow better control over the magnetisation.
    - Idea is to use a laser to heat up the part of the surface being modified.
- A hard-disk stores a fixed amount of data per sector, hence the area of a sector needs to be roughly the same. Hence, the hard-disk has more sectors on the outside tracks than on the inside.

Disk Addressing

- Initially, exposed the geometry - refer to a location by (cylinder, surface, sector)
- Now, logical sector addressing (or logical block addresses) are used
    - Sectors numbered from 0 to n

    - Makes disk management easier
    - Works around BIOS limitations
        - e.g. assumption that we have 6 bits for the sector, 4 bits for the head, 14 bits for the cylinder
        - Imposes 8GB limit

> Side note on disk capacity: for storage, manufacturers use powers of 10, and powers of 2 for memory. For examination purposes, use one of them consistently.

Disk Formatting

- Before a disk is used, it must be formatted.
- A low level format:
    - Disk sector layout:
        - Preamble
            - Needs to understand where the sector starts, using this preamble.
        - Data
        - Error correcting code
            - Used to find when particular bits in the sector have been corrupted. (Sector I/O error, for example due to physical damage).
    - Cylinder skew
    - Interleaving
- There is also a high level formatting: boot block, free block list, root directory, empty file systems.

Tutorial Question

A disk controller with enough memory can perform read-ahead, reading blocks on the current track into memory before the CPU asks for them.

Should it also do a write-behind, reporting back to the CPU that a block has been written once it is stored in the disk controller's memory? Or wait until it has written it?

- Consistency should be ensured: in a power failure, data may be lost.
- Disk controllers have a capacitor which can power the hard-disk and the disk controller memory long enough to write everything in memory during a power failure - hence write-back is possible.

Disk Performance

Seek time: Time taken for the head to cover the distance to seek to the track.

Rotational delay (Latency time): Time taken for the disk to rotate so that the correct sector is under the head.

Transfer time: Time taken for the disk to rotate so that all the data in the sector can be read.

Rotational delay and transfer time is dependent upon how fast the disk can spin. To reduce the seek time, need to decrease the time it takes for the head to move. Increasing storage density helps as tracks are closer together. In software, we want to exploit locality of data.

For a typical disk with 512 byte sectors, the seek times are <1ms, for adjacent cylinders, and 8ms on average. Average rotation time is 4ms, and the transfer rate is 100MB.

Disk scheduling aims to minimise seek and latency times, by ordering requests with respect to the head position. The seek time is 2 to 3 times larger than the latency time and is more important to reduce.

Total access time: t_{seek}, $t_{latency} = 1/(2r)$. Transfer time is the time taken for all the sectors we wanted to be read. $t_{transfer}$: b/rN where b is the number of bytes to be transferred, N is the number of bytes per track and r is the rotation speed in rps. Modern disks will always read the entire track into the whole memory. Total access time is the sum of the 3 terms.

Example:

Average seek time: 10ms, rotation speed: 10,000 rpm, 512 byte sectors, 320 sectors per track. File size: 2560 sectors (1.3MB).

Case A:

- Read file stored as compactly as possible on disk. File occupies all sectors on 8 adjacent tracks (8 * 320 = 2560).
- Time to read first track: average seek = 10ms, rotational delay = $1/(2 * (10,000/60))$ = 3ms, read 320 sectors = full rotation = 6ms. Total: 19ms.
- Time to read the next track is 3 + 6 = 9ms
- Total: 19 + 7 * 9 = 82ms.

Case B:

- Read file with all sectors randomly distributed
- Average seek = 10ms, rotational delay = 3ms, read 1 sector = $512/(512 * 320 * (10000/60))$. Total = 13.01875 ms. Total time = 33.328 seconds.

Disk Scheduling

If there is no ordering of requests then we have random seek patterns, which is OK for lightly loaded disks but bad for heavy loads. This is a **first-come-first-serve** strategy, and it is fair (the request will always be satisfied) but for practical purposes causes performance issues. Both the disk controller and the OS can order disk requests.

Shortest-Seek-Time-First:

- Order requests according to shortest seek distance from current head position.
- Discriminates against innermost and outermost tracks.
- e.g. Queue: 98, 183, 37, 122, 14, 130, 60, 67, with head at 53 initially is executed in the order: 53, 60, 67, and so on
- Unfair and unpredictable
- Requests may be unlucky and not serviced for a long period of time, if newer requests arrive.
- The read-write head will on average stay towards the middle, so unfair to requests on innermost and outermost tracks.

SCAN Scheduling:

- Choose requests which result in shortest seek time in a **preferred direction**.
- Enforces a rule where requests are only handled as part of a single sweep of the read-write head.
- Say head is at 53, and the direction is towards 0. It finishes requests in the queue along the way. Once it runs out, it chooses a new direction.

- All requests, even those which are extreme, will get serviced.
- Performance may be less predictable: while the seek is occurring, new requests may still arrive, so extreme location requests can still be delayed.

C-SCAN:

- Service requests in *one direction only*
- When head reaches the inner most request, jump instantly to the outermost request without servicing any requests along the way.
- Lower variance for requests on extreme tracks.
- No guarantee on fairness, but less delays are likely.
- Used practically.

N-Step SCAN:

- Services only requests in the queue when it began (new requests cannot join the current queue - new requests can join a different queue, which are serviced during a return sweep, or otherwise on the next sweep.).
- Doesn't delay requests indefinitely.

Case Study: Linux Disk Scheduling

- I/O requests are placed in a request list, at the level of the block device.
- One of the properties a block device is assumed to have is to service a queue of requests. A disk may reorder the list.
- Block device associates memory pages with requests.
- The reordering may be done within the device driver, but the kernel may also do some prioritisation.
- The device driver knows about the geometry of the disk hence it is better at reordering.
- The Linux kernel will, for requests to adjacent blocks, will attempt to merge them. But synchronous read requests may starve during large writes.
    - Two process issuing requests. A issues read requests, processing a small amount of data retrieved from each read. B issues large writes. The requests for writes are continuous, and can fill up the queue, while the reads are sporadic.
    - This is a common pattern.
- Deadline scheduler: ensures reads are performed by a deadline. Services requests if they have been waiting for too long.
- Anticipatory scheduler: delay after read request completes.
    - Process will issue another synchronous read operation before its quantum expires.
    - Add a delay in anticipation of reading of an adjacent, sequential block (likely). Then these can be handled in one seek.
    - Reduces excessive seeking.
    - Can lead to reduced throughput if the process does not act in this way. We can anticipate process behaviour from past usage patterns.

Tutorial Question

Suppose that the current position of the disk arm is over cylinder 200. The disk request queue contains requests for sectors on the following cylinders: 400, 20, 19, 74, 899.

1. FCFS policy:
    1. Same order.

2. SSTF policy:
    1. 74, 20, 19, 400, 899
3. SCAN policy:
    1. Travelling towards highest: 400, 899, 74, 20, 19
4. C-SCAN policy:
    1. Travelling towards highest: 400, 899, 19, 20, 74

Solid State Drives

- Organised similarly to memory: planes containing blocks containing pages.
- Access at a given granularity (unlike memory) - does not need to be byte addressable, making it cheaper to manufacture.
- SSD vs HDD:
    - More bandwidth (1GB/s r/w vs 100MB/s) however older buses may not account for higher transfer rates.
    - Similar latencies (microseconds vs milliseconds).
- Cost per storage unit is much higher than SSDs: (storage) GB/$, (throughput) MB/s/$, IOPS/$ (I/O per second).
    - IOPS quantify: given individual requests, how many can be handled? Parallel execution improves this.
    - Hard-disks must service requests sequentially. With SSDs, this is not the case - as long as requests are not in the same block, the requests can be handled at the same time.
    - SSDs perform poorly on storage per $, and throughput: since although a SSD can give high throughput, due to the cost of SSDs, the throughput/$ is much lower.
    - For data centers (e.g. Youtube), HDDs are used.
    - SSDs can be used in mobile phones for example as moving parts can cause failures.

RAID

Problem: disk performance has not improved, looking at IOPS and throughput. However CPU performance has improved exponentially (Moore's law). Solution is to use parallel disks.

RAID: Redundant Array of Inexpensive Disks.

- Use an array of physical drives appearing as a single virtual drive.
- Store data distributed over array of physical disks to allow a parallel operation (**striping**)
- Use redundant disk capacity to respond to disk failure
    - **Mean-time-to-failure** is the average time for the disk to fail. By having multiple disks, this is lower.
- Expose the array through a RAID controller as a single disk.

RAID Levels have different properties in terms of performance, level of redundancy, degree of space efficiency (how much of the capacity is for data vs redundancy).

RAID Level 0 (Striping)

- Use multiple disks and spread out data
- The capacity is the sum of each individual disk.
- Disks can seek/transfer data concurrently, as long as data is on different disks.
- No redundancy/fault tolerance. As soon as a disk fails, it is not recovered. Don't need to worry about consistency.

RAID Level 1 (Mirroring)

- Data is mirrored across disks
- Reads can be serviced by either disk (fast)
- Writes update both disks in parallel (slower)
- Failure recovery is simple
    - But higher storage overhead, since half the actual capacity is redundant

RAID Level 2 (Bit-level Hamming)

- Level 0 and 1 does not help with transfer rates.
- Parallel access by striping at bit-level across disks, using the Hamming error-correcting code.
- Data being read is read from multiple disks.
- Corrects single-bit errors (by checking other disks) and detects double-bit errors.
- Very high throughput for r/w
    - All disks must participate in I/O requests so there is no concurrency of request servicing.
    - **Read-modify-write** cycle for writing: read ECC information , modify the data, recompute error correcting code, and then write.
- Only utilised if high error rates are expected
    - ECC disks become a bottleneck.
    - Flexibility in which error correcting code to use.
    - High storage overhead.

RAID Level 3 (Byte-Level XOR)

- Single parity strip is used, by XORing across the data and storing this as parity information (whether it is odd or even for example).
- Reconstruct missing data from the parity and remaining data.
- Lower storage overhead than RAID level 2, but only one I/O request can take place at a time.
- This calculation is easy to carry out, can be done in the disk controller.
- Partitioning is done at the byte level, so similarly to Level 2, all disks partake in I/O and hence there is a lack of concurrency.

RAID L2 and L3 were useful when disks were prone to error.

RAID Level 4 (Block-Level XOR)

- Parity strip is handled on a block basis.
- Each disk operates independently.
- One extra disk used to parity information of blocks.
- Can recover one disk failure.
- Could service multiple reads concurrently (as long as blocks being read are on different disks).
- Parity disks are a bottleneck: every write updates the parity disk.

RAID Level 5 (Block-Level Distributed XOR)

- Like RAID 4, but the parity information is distributed throughout the disks. The parity information would live on different disks.
- There is some potential for write concurrency, as long as the parity information which must be updated isn't on a disk being written to.
- Contention for parity information.

- Reconstruction of a disk is more complicated (and slow).
    - Also need to recompute the parity information which was on the disk based upon the data of the other disks.
    - Modern controllers are able to do this as a background process.

Category	Level	Description	I/O Data Transfer (R/W)	i/o request rate (r/w)
Striping	0	Non-redundant	+/+	+/+
Mirroring	1	Mirrored	+/0	+/0
Parallel	2	Redundant with Hamming code	++/++	0/0
	3	Bit interleaved parity	++/++	0/0
Independent	4	Block interleaved parity	+/-	+/-
	5	Block interleaved distributed parity	+/-	+/- or 0

+ - better than single disk, 0 is the same, 0 is worse.

Disk Cache

Buffer in main memory for disk sectors contains a copy of some sector. It has a finite space so we need a **replacement policy**.

Least Recently Used:

- Replace the block that was in the cache longest with no references
- The cache is a stack of blocks
- In fact we use a stack of pointers
- However it does not account for the number of accesses a block has experienced.

Least Frequently Used:

- Replace the block with the fewest references
- Counter associated with each block, incremented with each access
- Some blocks may be referenced many times in a short period of time, which is misleading.

Frequency Based Replacement:

- LRU stack divided into a new section and an old section
- Block referenced means it is move to the top of the stack
- Only increment reference count if not already in the new stack
- However, blocks age out too quickly as they haven't been cached long enough to gather a larger count.
    - Solution is to use 3 sections and only replace blocks from the old section.

In fact, pages and blocks are often handled in the same way.

File Systems

Overview

- Long term, nonvolatile, online storage.
- Sharing of information or software.
- Concurrent access to shared data.
- Organisation and management of data.
- A file is a named collection of data of arbitrary size, allowing appending, writing, reading, access etc.
- The type of a file is denoted by its extension
    - Earlier operating systems such as Windows used the extension to decide the type
    - These days it is a convention
- Special files are character or block devices for example
- Regular files may be ASCII or binary
- Directories are files
- Links are special files which are "pointers"

File user functions

- Create, delete, open, close, read, write, seek, truncate (erase contents but keep all other attributes), rename, read attributes, write attributes.

Filesystem Functions

- Logical name to physical disk translation, e.g. /home/user/item.txt -> disk 2, block 391
- Management of disk space (allocation and deallocation)
- File locking for exclusive access
- Performance optimisation (caching and buffering)
    - Although caching can occur at the block level too
- Protection against system failure (back-up and restore)
    - If a subset of the blocks have been lost, then parts of the files have been lost.
    - Without backups, we may be able to reconstruct some data, especially for loss of metadata.
- Security (stop unauthorised access)

File attributes

- Basic information
    - File name
    - File type
    - File organisation (sequential access, random access)
    - File creator (program which created the file)
- Address information
    - Volume: disk drive, partition
    - Start address
    - Size used
    - Size allocated
        - We distinguish between allocated and used size - we can only allocate at the block level, so we may allocate a 4KB block for a 1KB file for example
- Access control information

    - Owner
    - Authentication
    - Permitted actions: read, write, delete, for owner/ others
- Usage information
    - Creation time
    - Last modified (and who modified it potentially)
    - Last read
    - Last archived
    - Expiry date (when the file will be automatically deleted)
    - Access activity counts (reads and writes)
- Some metadata stored in Linux:
    - File type: `b` - block device, `c` - character device, `d` - directory, `l` - link, `-` - regular file
    - User id of owner
    - If device: then the major device number and minor number, otherwise the file size
    - inode number
    - Date of creation
    - File name
    - Number of (hard) links which exist to the file
- The `stat` system call can also access file attributes

File System Organisation

How can a filesystem map files to all the blocks it requires?

Dyanamic Space Management: file size is naturally variable. Space is allocated in blocks.

- Block size too large: wastes space for small files, more memory needed for buffer space.
- Block size too small: wastes space for large files
    - High overhead in terms of management data.
    - Blocks may be spread across the disk, so higher file transfer time.

We have 4 different approaches for allocation of blocks, in increasing complexity.

Contiguous File Allocation

- Place file data at contiguous address on storage device.
- Easy to keep track of: only need first and last blocks.
- External fragmentation.
- Poor performance if files grow and shrink over time.
- Leaves holes.
- File growing beyond size originally specified and no contiguous free blocks available: must be transferred to a new area of adequate size, leading to additional I/O
- For optical mediums (read-only), writes are not permitted, so file systems for DVDs, Blu-Ray etc. use this file system.

Block Chaining

- Linked list organisation
- We record the pointer to the first block, which contains a pointer to the next block and so on.
- Blocks no longer contiguous: appending and truncation is easy
- In each disk block, waste space in storing pointers

    - Now mixing data with meta-data, so must account for this in calculations, adding complexity
- Random access operations to a given byte in the file requires walking pointers (linear time)
- Have to seek across the hard-drive, which is very expensive.
- Insertion/deletion by modifying pointer in previous block
- Large block sizes result in significant fragmentation, small block sizes means seeks are higher.

Block Allocation Table

- Used by early Windows/MS-DOS (File Allocation Table - FAT) and USBs
- Store the pointers in the block allocation table. The block number stores the data but is also used as an index into the block allocation table to find the next block.
- Directory entries store the first block.
- There is no pointer in the blocks
- Can load the entire block allocation table into memory, which means random access is much faster (single disk access required)
- Block allocation table grows linearly with the number of blocks, instead of linearly with the number of files.
    - Even on an empty disk, all blocks must be represented in the block allocation table.
- Useful when disks were small.
- File allocation table (FAT16/32) was stored on disk but cached in memory.
- Files can become highly fragmented and require periodic defragmentation.
    - We can ensure that blocks are allocated contiguously and more efficiently to reduce seeks, by updating the block allocation table.

Index Blocks

- Used by ext4, NTFS
- Each file has one or more **index blocks**
- The index block contains a list of pointers that point to file data blocks
- File's directory entry points to its index block
- Last few entries in the index block can store pointers to more index blocks, allowing for chaining.
- Searching can take place in index blocks
- Place index blocks near corresponding data blocks for quicker access
- Cache index blocks into memory
- Amount of metadata grows with the number of files
- We can also use a tree hierarchy to keep track of pointers, giving logarithmic access instead of linear (inodes).
- The height of the tree determines the size of the tree
    - However too large of a tree is inefficient for smaller files
    - Hence, we can have an indirect pointer, which points to a 1-level tree, doubly indirect pointer, which points to a 2-level tree (indirect pointer), triply indirect pointer and so on
    - This allows for flexibility, depending on the file size.
- In Linux, access control, user ID etc. is also stored within the inode.

Tutorial Question

Consider a disk with block size 1024 bytes. Each disk address can be stored in 4 bytes. Block linkage is used for file storage - each block contains the address of the next block in the file.

How many block reads will be needed to access the 1022nd data byte? The 510100th data byte?

Each block stores 1020 bytes of "useful data" and 4 bytes for the address.

Note that 500 * 1020 = 510000, 498 * 1024 = 509952.

To access 1022nd byte - we store 1020 bytes in the first block. Hence it is on the 2nd disk block, requiring 2 disk reads (read the pointer in the first block, and then read the second block).

The 510100th byte is on the 501st disk block - so 501 disk reads are required.

If we used FAT: there are 1024 data bytes per block. Each block can represent 1024/4 = 256 data blocks. We assume that we don't have the table in memory.

The 1020th byte is on the 1st block. We require 2 reads: one for the table, and one for the block.

The 510100th byte is on the 499th data block. At best, all the pointers are adjacent entries in the file allocation table. 2 blocks is the minimum required to store 499 pointers, since each block can store 256 pointers. Hence, at best, we require 3 reads (2 reads on the table), and at worst, we require 500 reads (assuming that we do 500 reads jumping through the allocation table).

Tutorial Question

In an OS, an inode contains 6 direct pointers, 1 pointer to a (singly) indirect block, and 1 pointer to a doubly indirect block. Each pointer is 8 bytes. Assume a disk block is 1024 bytes, and each indirect block fills a single block.

1. What is the maximum file size for this file system?
    1. 6 * 1024 + 128 * 1024 (we can fit 128 pointers in a block) + 128^2 * 1024 (we can fit 128 in the first layer, and in each indirect block, 128 as well) = 16.13 MB
2. What is the maximum file size if the OS used triply indirect pointers?
    1. Now we can store 128^3 * 1024 more, which is 2.02 GB.

Now how many disk block reads are required to read the 1020th byte? The first direct pointer can access this, so 2 reads again. For the 510100th byte, we need 4 reads: follow a double indirect pointer to an indirect pointer, to a direct pointer, to a block.

Free Space Management

Use a **free list**:

- Linked list of blocks containing locations of free blocks.
- Blocks are allocated from the beginning of the free list.
- Newly freed blocks are appended to the end of the list.
- Problem: we would like to allocate contiguous blocks as much as possible.

Use a **bitmap**:

- Contains one bit for each disk block
- Indicates whether the block is in use

- We can more efficiently determine available contiguous blocks at certain locations, by loading it into memory
- However, we still need to scan linearly over the bitmap to find a free block until we can find a region large enough to satisfy allocation.
- Used by modern operating systems.

Filesystem Layout

- Fixed disk layout (with inodes)
    - Boot block
    - Superblock
        - Accessed by kernel
        - Contains information about the filesystem
        - Number of inodes
        - Number of data blocks
        - Start of inode and free space bitmap
        - First data block
        - Block size
        - Maximum file size
    - Free inode bitmap
        - We don't put the inodes at a fixed position, for locality of access.
        - So now we need to keep track of where the inodes are (and where they are free).
    - Free block bitmap
    - inodes + data

File System Directories

Directory maps symbolic file names to a logical disk.

- Allows with file organisation
- Allows for uniqueness of names (can't have files with the same name in the same directory).

Hierarchical file system:

- Root indicates where on disk the root directory begins
- Root directory points to other directories which contain entries for their files
- Pathname is path from root to file
- Absolute path name is from root
- Can also have relative paths

Directory operations:

- Open/close
- Search: find file in directory system using pattern matching on a string and wildcard characters
- Create/delete
- Link/unlink
- Change directory
- List all files
- Read/write attributes (same as files)
- Mount

    - Can extend file system with new files at runtime
    - Creates link in directory to directory in a different file system

Linux Steps in looking up `/usr/ast/mbox`:

- Look up `usr` in root directory, `/`, (there is a convention that inode 2 contains `/`) yielding (e.g.) inode 6.
- Inode 6 says `/usr` is in block 132.
- Lookup in block 132, `/usr/ast` is inode 26.
- Inode 26 says that `/usr/ast` is in block 406.
- Lookup block 406, `/usr/ast/mbox` is inode 60.
- At each step, check the permissions to see if user can enter the file.

Linux directory representation:

- `dirent` struct is the content of a directory file, which contains the name, the inode, offset to the dirent, length of the directory name.

Links

A link is a reference to a directory or file in another part of the file system, allowing alternative names (locations in the tree).

A **hard link** references the same inode/address of the file. (Only supported for files in Unix). At some other point in the file system, we refer to the same inode.

- We cannot have hard links to directories because then the file system is no longer a tree - it is a graph, which allows for cycles, and hence traversal is much more difficult.
- Traversal is a common operation.

A **symbolic/soft link** is a special file, which includes a reference to the full path name of file/directory, and is created as a directory entry. When doing traversals, we don't follow the link as it is a special file. We can link to directories.

When deleting files, we must search for all links and remove them.

- For symbolic links, we can leave the link and cause an exception when it is used.
    - Soft links can be dangling.
- For hard links, keep a link count with the file (number of hard links remaining) and delete the file when the count is 0.
    - This is why in the metadata, the number of hard links is stored (effectively, reference counting).
    - Hard links cannot be dangling.
- Directory traversal algorithms may loop.

Mounting

- Now, we can have multiple file systems mounted under the root directory.
- We can have soft-links but not hard-links.
    - Not all file systems use inodes.
- Mount point is a directory in the native file system assigned to the root of the mounted file system.
    - Typically it is an empty directory, as what is already in the directory will be overwritten.
- Manage mounted directories with **mount tables**.
    - Store information about location of mount points and devices.

    - When native FS encounters a mount point, use the mount table to determine the device and type of mounted FS.

Case Study: Linux ext2fs

- Goals: high performance, robust FS with advanced features. Typical block sizes: 1024, 2048, 4086, 8192 bytes (configurable).
- 5% of blocks are reserved for root, allowing root processes to continue to run after a malicious/errant user process consumes all FS disk space. -
    - Hence, important system processes (such as error loggers) can continue to run.
- ext2 inode:
    - First 12 pointers are direct
    - 13th pointer is an indirect pointer
    - 14th pointer is a doubly indirect pointer
    - 15th pointer is a triply indirect pointer.
- Block groups
    - These are clusters of contiguous blocks.
    - File system attempts to store related data in the same block group.
    - Reduces the seek time for accessing groups of related data.
    - Block group structure has a superblock, group descriptors, block allocation bitmap, inode allocation bitmap, inode table and data blocks.
    - Multiple block groups, with the file system as a sequence of block groups.
    - Attempt to allocate entire file in a single block group.
    - Superblock will still contain critical data about the entire FS
        - Redundant copies of superblock in some block groups
        - If we have corruption at the start of the system, we lose all data as superblock is lost, so we have some recovery method.
    - Allows for concurrency, as long as writing occurs in different block groups.
- When creating the file system, we must decide upon maximum number of inodes, as creating block groups must contain an inode allocation table. Hence the number of files is pre-determined upon file system creation.

Security

Goals

- Prevent unauthorised access
- Permit authorised sharing of resources
- Data confidentiality
    - Prevent theft of data
- Data integrity
    - Prevent destruction/alteration of data
- System availability
    - Prevent denial of service attacks

Tutorial Question

1. Why are security and protection important even for computers that do not contain sensitive data?
    1. The machine could still be taken control of, and used, for example, as a botnet, or for data mining, spreading viruses, identity theft.
2. Sharing and protection are conflicting goals. What are examples of sharing in OSs and what protection mechanisms are necessary?
    1. Sharing virtual memory regions (OS page protection, which processes can access which pages)
    2. Sharing processor cores (context switches and interrupts to avoid starvation)
    3. Sharing files (access control lists and encryption)

Types of Security

Policy specifies what security is provided: what is protected, who has access, what is permitted.

Mechanisms specify how to implement policies. Policies can change over time, but mechanisms can stay the same.

People related security

- A large number of computer crime is committed by **insiders**.
    - Employees need privileges to carry out duties
- Social engineering
    - People are not security conscious
    - Phishing attacks
    - Tailgating into buildings
- People working around security measures for convenience
    - Reusing passwords
    - Providing weak passwords
- Incorrect security expectations
    - Assuming that sender email address cannot be forged

Hardware security

- With physical access to a computer/peripherals you can
    - Read contents of memory and disks by taking the disk out physically
        - Disk encryption can prevent this
    - Listen to network traffic including unencrypted passwords
    - Alter contents of memory/disks
    - Forge messages on a network
    - Steal or destroy the machine
- Hardware can have flaws
    - Side channel attacks on CPUs
    - Incorrect implemented access control checks such as Meltdown

Software Security

- Software bugs may allow attackers to compromise the system
    - Gain root privileges
    - Crash the application

    - Steal data
    - Compromise data integrity
    - Deny access to the system
- Can exploit:
    - Buffer overflow
    - Integer overflow
    - String formatting vulnerabilities

Access Control

Authentication

- Verify identity of principal (refers to either the user or a process on behalf of the user) based on:
    - Personal characteristics
        - Fingerprints
        - Voiceprints
        - Retina patterns
        - Signature analysis
        - Signature motion analysis
        - Typing rhythm
        - High equipment cost and false positives/negatives are an issue.
    - Possession
        - For example owning a key to a lock.
        - Can ensure physical security of computers
        - Coded magnetic cards, RFID cards, implanted sensors.
        - Impersonation attacks if key lost (use 2FA to counter this)
        - High equipment costs
    - Knowledge
        - Based on secret knowledge - a password
        - Cheap to implement
        - People may pick common words - dictionary attacks can exploit this. For example, login name, first name, street names, dictionary words, or modifications to the words.
        - Password reuse: as soon as one website is compromised, others can also be compromised.
        - Security is as good as the security of the weakest system.
        - Well chosen password can only be cracked by exhaustive search.
- Passwords used to store user passwords in a protected file, but this was vulnerable to data theft/abuse by admins.
- Modern OSs use a one way hashed password.
- Compare the hashed version to the input.
- One way function is a function that is easy to compute, but hard to invert.
- Pre-image resistance: given a hash value h, it is infeasible to find M such that H(M) = h
- Guessing is the only feasible way to find the cleartext from the encrypted - use a slow hash function to limit the number of guesses.
- Rainbow tables: given a hash function, compute a **rainbow table** of H(k) for many popular passwords k.
- If H(password) leaks, compare it with all available H(k).

- We can protect against this using a **salt**.
    - Add a salt value which is a random value for that hash of the password.
    - The salt is stored along side userid and hashed password.
    - Now we need to compute a rainbow table per user.
    - Salt values are usually large values.
    - Duplicate passwords are not the same when hashed.

Authorisation

- Allow principals to perform action only when authorised
- Normally there is a default policy: no access or all access?
- **Principle of Least Privilege**: give a user the least amount of privs. needed to allow them to do the assigned task.
    - If the account is compromised, then damage is minimised.
    - Unfortunately for convenience, more than necessary privs. may be given.
- **Protection Domains**
    - Define a set of **access rights** as a set of objects and operations permitted on them.
    - Principal executing in **domain** D has access rights specified by that domain.
- **Access Control Matrix** specifies authorisation policy: rows represent principals (users, user groups) and columns represent target objects (files, devices, processes). Entries are permitted actions.
    - Expensive to implement matrix as a global 2d array
    - Can store as an **ACL (Access Control List)**, which is a column in the matrix, assigned to the object, or as **Capabilities** which is a row in the matrix, assigned to the principal.
    - In practice, ACLs are used extensively, but there is often some support for capabilities.
- **ACL**
    - Files are associated with metadata, hence the ACL can be stored within the inode as part of the file.
- **Capabilities**
    - Possession of capability gives right to perform operations specified by it, similar to possession of a key.
    - They are protected objects - a malicious principal could modify it.
    - Protected pointer to object specifying permitted operations on object (such as a file descriptor)
        - Only accessed indirectly.
        - OS provides procedures to create, delete and modify capabilities.
        - Principal given handle.
    - Alternatively, an encrypted capability is given to the user.
        - The OS can check the capability for tampering.
- ACL is more popular, although capabilities are better for principle of least privilege, since a decision must be made on what privilege a principal can have.
    - Revocation is very difficult with capabilities, since the user owns the capability. The user can copy the capability. Capabilities use revocation lists, and the OS checks against this list. With ACL, the permission bits can be changed easily.
    - Capabilities are better for rights transfers, simply by moving the capability.
    - Persistence: ACLs are convenient and intuitive to store alongside files, whereas capabilities must be stored separately.

Case Study: UNIX

- Users are principals, identified by a UID, with root having UID 0 and can access any resource.
- In UNIX, there are 3 operations: read, write and execute.
- Groups: users can belong to one or more groups and files can belong to one group. Groups have particular rights.
- For directories, the read right means the contents can be listed, the write access right means files inside can be modified (created/deleted) and execute means the directory can be entered and files can be accessed.
- Consider the description:
  - `-rw-r--r-- 1 prp lsds 2517522`
  - First `-` refer to the file type
  - Next 3 chars, `rw-` are the access rights of group members.
  - Next 3 chars, `r--` are the access rights of the owner.
  - Next 3 chars, `r--` are the access rights of everybody else.
  - `prp` is the file owner
  - `lsds` is the name of the group the file belongs to.
- We also have permission bits `s` - setuid, setgid and `t` (stick bit).
- What happens when user A executes a program for A which has execute privileges?
  - Launching a program runs with A's prvileges.
  - Can access any files which A can.
- How does changing password work? Although it is a privileged operation, users can change their own password.
  - The **SUID** bit means that the file switches effective UID to file owner (not executor) when executed.
  - This increased privileges when using system programs.
  - For `/usr/bin/passwd`, it runs as root.
- A process has three IDs:
  - Real UID: ID of the user who started the process.
  - Effective UID: effective ID of the process (ID of the privileges the process actually has, allowing setuid to work), which is used in access control checks.
  - Saved UID: a saved ID to which the effective ID can be changed to, allowing a process to decrease the privileges it has.
    - Effective UID can be saved
  - Upon starting, effective UID = real UID
  - If a setuid file, then effetive UID = ID of file owner
  - Non-root processes can change their EUID to their real UID or their saved UID.

Tutorial Question

Consider a file with the permissions: `-rwsrwxrwx`. What implications does this file have?

1. Everyone has full root access, since we have done `rwx`, so even though group permissions are `rws`, the `rwx` permission applies to everyone.

DAC vs MAC

Windows has a much more advanced permission system compared to UNIX, for historic reasons.

- Discretionary Access Control (DAC):
    - Principals determine who can access their objects.
    - If you are the owner of a file, you can set the permissions.
    - Decentralises the managing of permissions.
    - Can backfire if users set bad permissions.
- Mandatory Access Control (MAC):
    - Precise system rules that determine access to objects.
    - Privileged entity sets the access control rights (usually root).
    - Much more secure, can set a global policy, but DAC is much more flexible.
    - In a cloud context, MAC can prevent data leakage by reasoning about global data usage.

Bell-La Padula Model

- Objects and principals have a assigned **security level**: unclassified, confidential, top secret.
- Two rules:
    - **Simple security property**: a process running at security level k can read only objects at its level or lower.
    - The *** property**: a process running at security level k can only write to objects at its level or higher, although it cannot read it. Also cannot write to a lower level.
- Now information flow is in one direction. Can't involuntarily disclose secret data to a lower level. We can make data more secure.
- This is MAC, as the processes cannot decide which level they operate on.
- However, we have not enforced data integrity.

Biba Model

- Like Bell-La Padula, but from the perspective of data integrity.
- **Simple integrity principle**: a process running at level k can write to objects at its level or lower.
- **The integrity * principle**: a process running at level k can read only objects at its level or higher.
- Integrity is concerned with who generated the information. We can generate objects at a lower integrity level, since we have higher integrity. However, we cannot write at a higher integrity level, since we are not trusted.
- We cannot read down, as the data is not trusted to be at the right integrity, and can corrupt our integrity data. Only want to input data which is as good as our integrity level.

Design Principles For Security

- Give each process the least privilege possible
    - Default should be no access
- Protection mechanism should be simple and uniform
    - More complex mechanisms are more bug-prone, and users can specify them incorrectly.
- Scheme is psychologically acceptable.
- System design should be publicly documented, security through obscurity is a bad idea, since if it is reverse-engineered then all security is broken.