CO211 - Operating Systems

4th October 2019

Outline of the Course

- overview and introduction — structure, case studies
- processes and threads — abstractions that an OS uses to execute code
- inter-process communication (IPC) — allows multiple processes to communicate with each other
- memory management — allocation, abstraction for virtual memory, paging
- device management — types, drivers
- disk management — scheduling, caching, RAID
- file systems — basic abstractions for storage and implementation
- security — authentication, access control

Note that this follows a similar structure to most OS courses, and therefore we can reference content from other sources. *Operating Systems: Three Easy Pieces* is recommended, as it bridges between this course and the PintOS lab.

Overview

The general overview is that there is a system bus that interconnects different hardware components (including CPU and memory), and allows for communication between them.

The operating system provides abstractions for programs to use, meaning that they do not have to deal with the complex hardware. For example, a process abstraction expects an interface to the hardware, which allows programs to be used on different hardware. This means that the OS will need how to to control the hardware with drivers. The operating system has the following goals;

(1) **managing resources**

The operating system must be able to expose the resources efficiently to the application, and also share these resources fairly. Some examples are;

- CPU (multiple cores) — should decide what runs on each hardware thread
- memory — cache, RAM
- I/O devices — displays (GPUs), network interfaces
- internal devices — clocks, timers, interrupt controllers
- persistent storage

OS uses both time and space multiplexing for sharing. An example for the former is how the effect of parallelism can be achieved with a single CPU core by splitting up the time allocated per process, and an example for the latter is splitting up memory for each process.

On the other hand, with allocation, the OS must also support simultaneous resource access (such as to disks, RAM, network etc.). Continuing from this, it must also offer mutual exclusion, thus protecting risky operations (such as file writing). Generally, the OS aims to protect against corruption.

Finally, the operating system must also handle storing data, and enforce access control.

(2) **clean interfaces**

The OS should hide away the hardware, and applications use the hardware through an interface provided by the operating system. We can think of this as a virtual machine abstraction on top of the bare machine - similar to how the JVM works (but at a lower layer).

(3) **concurrency and non-determinism**

The operating system must be able to deal with concurrency, for example overlapping I/O and computation. This is because I/O devices tend to be slower, and while the device is working on the task, it shouldn't prevent the CPU from doing other work. An operating system may switch activities at arbitrary times, and this must be done safely - by offering synchronisation primitives. It should also protect processes by giving each program its own space, thus preventing interference.

Similarly, the OS is fundamentally non-deterministic, as it needs to handle interrupts (such as the network card receiving a packet, user interrupts, etc.).

Tutorial Questions

1) List the most important resources that must be managed by an operating system in the following settings;

   (a) supercomputer

   - computation time — primarily used for intensive computations
   - memory

   (b) networked workstations connected to a server

   - bandwidth — must handle packet processing and network traffic

   (c) smartphone

   - energy — limited power, can power off unused hardware
   - mobile network — (including other communication technology)
   - other sensors — issues of privacy, when to expose GPS etc.

   As this highlights, some uses will need specially designed operating systems. We also have general-process OS, as it takes a large amount of effort to implement a new operating system.

2) What is the **kernel** of an operating system?

   The part of the OS is always in memory, and runs in the privileged part of the CPU (user mode cannot access all functionality). Implements commonly used functions of the OS and has complete access to all hardware.

Kernel Design

- **monolithic kernel**

  Consider it as one large program that has all the functionality that you want an OS to perform.

  The kernel is a single executable with its own address space. There exists a **system call** interface that allows user mode applications to access the hardware. Software invokes functionality from the kernel by issuing system calls - the CPU must switch from user mode to kernel mode to support this. The kernel then executes some instruction on behalf of the application. Device drivers are part of the monolithic kernel.

  advantages

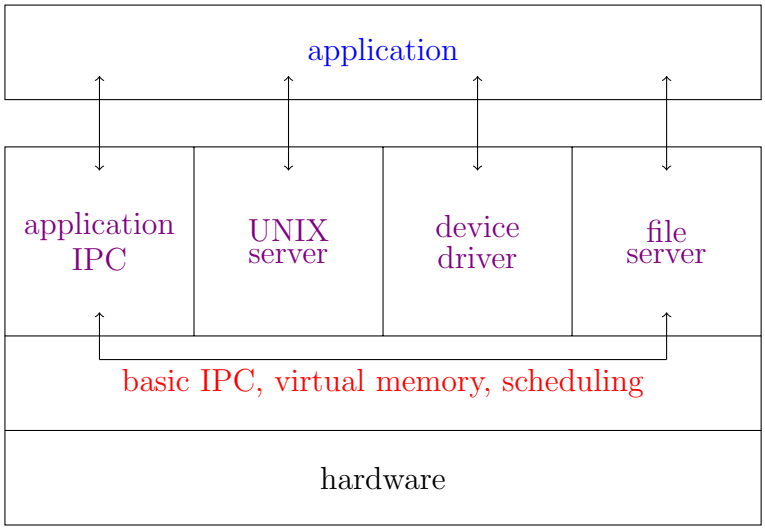
  - efficient calls within the kernel, as there it remains in kernel mode
  - flexible to write kernel components due to the shared memory (direct access with no limit to APIs)

  disadvantages

  - complex design
  - no protection between bits of kernel functionality, therefore any bugs within the kernel will crash the entire machine

- **microkernels**

  Only includes functionality that **requires** direct access to the hardware (or to be run in kernel mode). This is a minimalistic design and has the advantage of fewer bugs (due to the smaller amount of code).



  Note that both the application and servers run in user mode, and the kernel is in kernel mode. The kernel performs IPC between the servers, which are separated for device I/O, scheduling. file access etc.

  advantages

  - less complex kernel
  - clean interfaces for the servers
  - more reliable; one of the servers could crash and then restart, without bringing the entire kernel down

  disadvantages

  - performance overhead due to the requirement of message passing and transitioning between user mode and kernel mode (checks must be done to maintain the separation) - less of an issue now due to better hardware (e.g *Android*)

- **hybrid kernel** many modern designs use a combination of both

  This is a more structured design, however user-level servers can incur a performance penalty.

Linux Kernel

The structure of Linux system calls is to put arguments into registers Or on the stack, and then issue a trap to switch the CPU from user to kernel mode.

While C is the dominant language for the Linux kernel, the interrupt handlers are written in assembly, as they are low level pieces of code, and require fast performance (hence a low instruction count). Interrupt handlers are the primary means to interact with devices, it initiates dispatching which stops proxies, saves the state, starts the driver and returns.

Typically, we can split the Linux kernel into three parts;

- **I/O**

  One of the design philosophies under UNIX style operating system is to treat everything as a file, and use this file abstraction to expose different resources. Therefore, a lot of I/O resources can be hidden under this virtual file system.

- **memory management**

  Includes virtual memory with paging (and the abstractions associated with that).

- **process management**

  Includes process and thread abstraction, as well as synchronisation and scheduling between them.

In addition to this, Linux supports dynamically loaded modules into the kernel. This support was important as it allowed for the hardware configuration to change (new devices drivers could be loaded into the kernel, without recompiling).

Windows Kernel

The NTOS kernel layer implements Windows system call interface. This is an example of a hybrid kernel, as programs build on dynamic code libraries (DLLs) - which also make the kernel modular, however the executive servers in the kernel adopted the server model of the microkernel, but still runs in kernel space for the performance benefits. At the lower levels, there still exists a microkernel. In addition, there is also a hardware abstraction layer (HAL), as this was designed for portability.

It's also important to note that there are environment subsystems running in user mode allowing for different APIs to be exposed, including Win32, POSIX, and OS/2. While the Windows kernel was designed with a lot of flexibility, due to its nature as proprietary software, it only really focused support (until recently) on Win32 (and also Intel in terms of the HAL).

9th October 2020

Tutorial Questions

1. Why is the separation into a user mode and a kernel mode considered good OS design?

   Reduce the amount of code running in kernel mode, since a bug in user mode code should not bring down the entire system.

2. Which of the following instructions should only be allowed in kernel mode, and why?

   (a) disable all interrupts — only kernel mode

   if a user program were to disable interrupts, it would prevent the OS from scheduling processes

   (b) read the time of day clock — not privileged

   (c) change any memory — only kernel mode

   typically programs can only access its own memory, such that it cannot accidentally or maliciously interfere with other memory

   (d) set the time of day — typically kernel

   most programs assume monotonicity of the clock, and changing to an earlier time can cause bugs

3. Give an example in which the execution of a user process switches from user mode to kernel mode, and then back to user mode.

   Reading a file. Essentially anything that requires a system call, as it requires a switch from user mode to kernel mode, and then back.

4. A portable operating system is one that can be ported from one system architecture to another with little modification - explain why it is infeasible to build an OS that is portable without any modification.

At some point in the kernel, it will need to know about the ISA (instruction set architecture) of the CPU (hardware), and what instructions it can support. Some parts of the OS require assembly, and therefore requires modification. The hardware abstraction layer in the Windows kernel makes this easier.

Processes

One of the oldest abstractions in computing. This is an instance of a program being executed - this is useful as we can then execute multiple programs "simultaneously" on one processor, especially if not all resources are needed at the same time. This provides isolation between programs (own address space), and therefore doesn't interfere with other unrelated processes - if it needs to, then the IPC provided can be used. It also makes programming easier, as a programmer can assume it is the only process running.

Concurrency

It's important to note that there exists both pseudo-concurrency (on one CPU core), as well as real concurrency (across multiple CPU cores). The latter will still use the former per core, as the number of processes is much higher than the number of physical cores. In the case of multiple cores, we have to deal with conflicting accesses, whereas in the case with a single core, there is only one process really running at a time.

One method of creating the illusion of concurrency is time slicing. The OS switches the process currently running on the CPU with another runnable process, saving the original process' execution state, and then restoring it after it is switched back. Note that a runnable process isn't waiting for input, as we want to minimise the amount of time the CPU is idle. We also must ensure that the switching is fair - for example, if process `A` has a long execution time, compared to an interactive process `B`, letting `A` run for a long period would cause the interactive process to become unresponsive - therefore the time slice tends to be quite short (how often it lets a process run before switching).

1. If on average a process computes 20% of the time, then with 5 processes, we should have 100% CPU utilisation, right?

   Only in the ideal case, when they never wait for I/O at the same time. A better estimate is to look at the probability (assuming independence), with n being the number of processes, and p being the fraction of time a process is waiting for I/O. The probability that all are waiting for I/O would be p^n, and therefore the CPU utilisation would be $1 - p^n$.
2. How many processes nee to be running to only waste 10% of CPU if they spend 80% waiting for I/O?

$$1 - 0.8^n = 0.9 \Rightarrow 0.8^n = 0.1 \Rightarrow n = \log_{0.8}(0.1) \approx 10 \text{ concurrent processes}$$

Context Switches

A context switch is when the processor switches execution from process `A` to process `B`. This is done as part of a scheduling decision. With timer interrupts, the currently executing program passes control back to the kernel, which can then make a scheduling decision, changing what is currently running, possibly a different program and performing a context switch. This causes the order of execution between processes to become non-deterministic, as these events cannot be pre-determined.

This needs to be transparent to the process, therefore the state needs to be restored exactly, including anything currently in registers (this is saved by the hardware to the stack, before the hardware invokes the interrupt handler). This data is stored in a process descriptor, or a process control block (PCB), kept in the process table. The process has its own virtual machine;

- own virtual CPU

- own address space (stack, heap, text, data, etc.)
- resources it has access to (open file descriptors, etc.)

The information in registers (such as the program counter, page table register, stack pointer, etc), the process management information (process ID, parents, etc.), as well as file management information also needs to be stored (root directory, working directory, file descriptors, etc.).

It's also important to avoid unnecessary context switches as they are expensive, not just from the direct cost of managing state, but also the indirect cost to caching (as the old cache contents are no longer relevant). Therefore it has to balance fairness, and the frequency of context switches.

Process Lifecycle

Processes are created at the startup of a system, by the request of a user, or through a specific system call by a running process. These processes can be foreground processes, that the user interacts with, or background processes that provide services (such as printing or mail) or APIs that can be used by other processes (daemons).

A process can terminate under these conditions;

- normal completion, where the process completes execution
- through a system call (`exit()` in UNIX or `ExitProcess()` in Windows)
- abnormal exit, where the process has run into an error or unhandled exception - this is the importance of user and kernel space separation
- aborted, due to another process overruling its execution (such as killing from terminal)
- never - some processes such as daemons should run infinitely and never terminate (unless an error occurs)

UNIX allows for a process hierarchy (tree), by running `init` (typically), and all processes then form a tree. On the other hand, Windows has no notion of hierarchy, and rather the parent of a child process is given a token (a handle) to control it. This handle can be passed to another process.

10th October 2019

UNIX Processes (fork)

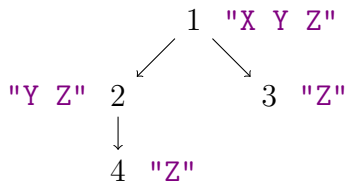
In UNIX `int fork(void)` creates a new child process, which is an exact copy of the parent process, inheriting all resources, and executed concurrently - however, different virtual address space. `fork` will return twice, however in the parent process it will return the child's process ID, but in the child it will return 0, thus the child knows it's the child. Additionally, if there is an error (such as exceeding the global process limit, or running out of memory when copying the parent), -1 will be returned to the parent.

```
1  #include <unistd.h>
2  #include <stdio.h>
3
4  int main() {
5    if (fork() != 0) {
6      printf("parent\n");
7    } else {
8      printf("child\n");
9    }
10   printf("common\n");
11 }
```

The parent and child processes start off with the same memory, but as they start writing to their own memory, they will diverge. In the tutorial question below, we'd end up with the following process tree (imagine the spaces in the strings are actually new lines);

```
#include <unistd.h>
#include <stdio.h>

int main() {
  if (fork() != 0)
    printf("X\n");
  if (fork() != 0)
    printf("Y\n");
  printf("Z\n");
}
```



However, note that because this creates new processes that run in parallel, the actual order of execution would be non-deterministic, and therefore the outputs can change the order in which they are printed.

UNIX Processes (execve)

While `fork` creates a copy of the parent process, we often want the child process to do something different; `int execve(const char *path, char *const argv[], char *const envp[])`.

- `path` full path name of the program to run
- `argv` arguments passed to main
- `envp` environment variables such as `$PATH` and `$HOME`

Running this changes the process image, and runs the new process. To start a new process, you could fork the current process, and if it is the child, then run `execve` to change the image. This has many useful wrappers, and `man execve` can be used as a reference.

UNIX Processes (waitpid)

The example below is an application, a simple command interpreter, for the two functions previously discussed, as well as a use of `int waitpid(int pid, int* stat, int options)`;

```
while (1) {
  read_command(command, parameters);
  if (fork() != 0) {
    // parent code here
    waitpid(-1, &status, 0);
  } else {
    // child code
    execve(command, parameters, 0);
  }
}
```

This suspends the execution of the calling process until the process with PID `pid` terminates, or a signal is received. If `pid` is set to the following values, it can wait for more than one child;

- `-1` wait for any child
- `0` any child in the same process group as the caller
- `-gid` wait for any child with the process group `gid`

This will return the `pid` of the terminated process, -1 if it is an error, or 0 if the call is not blocking and no children are terminated.

UNIX Processes (Termination)

A process can terminate from itself by executing `void exit(int status)`, which is also called implicitly once the program completes execution, and obviously does not return in the calling process (instead returns an exit status to the parent process). It can also be terminated by another process via `void kill(int pid, int sig)`, which sends the signal `sig` to the process associated with `pid`.

Design Philosophy

The UNIX design philosophy is to be simplistic. Having both `fork` and `execve` allows us to use the small building blocks, which have limited behaviour, to perform more complex tasks. This contrasts with Windows' `CreateProcess()` which combines both of them, however, it's much more complex and takes 10 parameters.

Process Communication

- **signals (UNIX)**

  Signals are an Inter-Process Communication mechanism, and they work similar to the delivery of hardware interrupts. If a process runs on behalf of root, the superuser, it has the permission to send signals to any process. The kernel can also send signals to any process. Some of the cases for signals being generated are as follows;

signal	meaning
`SIGINT`	interrupt from keyboard
`SIGABRT`	abort signal from `abort`
`SIGFPE`	floating point exception
`SIGSEGV`	invalid memory reference
`SIGPIPE`	broken pipe: writing to a pipe with no readers
`SIGALRM`	timer signal from `alarm`
`SIGTERM`	termination signal

  The default action for most signals is to terminate the process, however the receiving process may choose to do the following (`SIGKILL` and `SIGSTOP` cannot be ignored / handled);

  - ignore it
  - handle it manually with a signal handler;

```
signal(SIGINT, my_handler);

void my_handler(int sig) {
    printf("ignoring SIGINT");
}
```

- **pipes**

  This can be considered as a one-way communication channel between two processes. This essentially opens a byte stream from process `A` to process `B`, allowing `A` to send data to process `B`. This is commonly used in the command line, for example `cat file.txt | grep foobar` (the output of `cat` is now the input for `grep`). There are two types; unnamed (default) and named (can be referred to).

  This is opened with the `int pipe(int fd[2])` system call, which returns two file descriptors, the read end being in `fd[0]`, and the write end being in `fd[1]`. If the receiver is reading from an empty pipe, it blocks until data is written, and if the sender is writing to a full pipe, it blocks until data is read. The parent typically makes the system call, and then forks the process, passing

the file descriptors to the child. The sender should close the read end, and the receiver should close the write end.

A persistent pipe can outlive the process that created it - it is stored on the file system, and has different semantics since it is flushed when read from.

1. When two processes communicate through a pipe, the kernel allocated a buffer (say 64KB). What happens when the process at the write-end of the pipe attempts to send additional bytes on a full pipe?

   It cannot write to the buffer, therefore it will block (and the scheduler will choose another process to run) until the pipe is read from (and therefore freed up space in the buffer).
2. What happens when the process at the write-end of the pipe attempts to send additional bytes but the other process already closed the file descriptor associated with the pipe?

   The writing process will have an error returned to it.
3. The process at the write-end of the pipe wants to transmit a linked list data structure (with one integer field and a "next" pointer) over a pipe? How can it do this?

   The data must be serialised (as if it were going through a network). Since all processes have their own address spaces, the pointer would be meaningless.

- **shared memory**
- **semaphores**

Threads

Threads are also an abstraction for execution, but unlike processes, they are execution streams that share the same address space. When multithreading is used, each process can contain one or more threads. A thread lives within a process.

per process

- address space
- global variables
- open files
- child processes
- signals

per thread

- program counter (PC)
- registers
- stack

Threads allow for programs to execute in parallel, but more importantly they can block independently, therefore blocking in one part of the program (waiting for I/O, etc) does not affect the rest of it. This is useful, over having many processes, as processes have too much overhead, it is difficult to communicate between address spaces, and anything that blocks may switch out the entire application.

However, there can be issues with multiple threads. Since we are working with the same address space, we need to handle synchronisation, and prevent threads from interfering with each other accidentally (such as stack corruption).

Typically, when a `fork` is performed from a thread, only a single thread is created - however this can lead to issues if the parent is holding locks, the thread now also holds them. Generally, we want to avoid calling `fork` in a thread. While signalling and threading are compatible, there are many corner cases which can complicate the implementation.

PThreads

Posix Threads are defined by IEEE standard 1003.1c, and is implemented by most UNIX systems.

```
#include <pthread.h>
#include <sys/types.h>

pthread_t // type representing a thread
pthread_attr_t // type representing the attributes of a thread
```

Creating a thread is done with `int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void*), void *arg)`. It stores the newly created thread in `*thread`, and returns 0 if it was created successfully, or an error code otherwise (possibly due to lack of memory, due to the need for a stack). A function pointer is also required, as the thread will run the specified function with the arguments provided. The arguments are as follows;

- `attr` specifies attributes (`NULL` for default)
  such as minimum stack size, behaviour on process termination, etc.
- `start_routine` C function the thread will start executing
- `arg` argument to be passed into `start_routine` (can be `NULL` if none)
  if we want to pass in more arguments, pass in a struct, since it is in the same address space

A thread can be terminated with `pthread_exit(void *value_ptr)`, which terminates the thread, and makes the `value_ptr` available to any successful join (this is fine as threads reside in the same address space).

It's also important to note that a thread is automatically allocated for the main entry point (starting `main()`). If the main thread terminates without calling `pthread_exit()`, the entire process is terminated, however if it does call it, the remaining threads continue until termination (or `exit()` is called).

Yielding a thread with `int pthread_yield(void)` would be done for the same reasons as the system call `nice()` is done for processes (lowering priority in the scheduler). It releases the CPU to let another thread run, and will always return 0 (success) on Linux.

In order to join threads, `int pthread_join(pthread_t thread, void **value_ptr)` can be used. This blocks the caller until `thread` terminates, and the value can be accessed.

All of the content mentioned before assumes a kernel-level thread, such that all of the scheduling is managed by the kernel. However, a process can manage its own user-level threads. Threads in user-level tend to be more lightweight, as there is very low overhead of context switching, and synchronisation is fast. However, because it not visible to the kernel, it may preempt all the threads controlled by a process, instead of just a single one - if one of the threads perform a blocking system call, none of the other threads can run.

Tutorial Question

In this question, you are to compare reading a file using a single-threaded file server and a multithreaded server, running on a single-CPU machine. It takes 15ms to get a request for work, dispatch it, and do the rest of the necessary processing assuming that the data needed are in the block cache. A disk operation is needed $\frac{1}{3}$ of the time, requiring an additional 75ms, during which time the thread sleeps. Assume that thread switching time is negligible. How many requests per second can the server handle if it is;

- single-threaded?

  In this case, we should take the weighted average; in a cache hit, it takes the 15ms for the request. However, in a cache miss, it takes the 15ms, as well as the additional I/O operation, which gives

a total of 90ms. Taking the weighted average, with the probability given, it takes 40ms. This means that it can perform 25 requests per second.

- multithreaded?

  Each request needs 15ms of CPU time, and an average of $(\frac{1}{3} \cdot 75 =)$25ms I/O time. Therefore, the probability of a thread being blocked is $\frac{25}{40} = \frac{5}{8}$, as 25ms of the total 40ms is I/O. Assuming that they are independent, the probability of all n threads sleeping is $\frac{5}{8}^n$.

  With 100% CPU utilisation, we can do $\frac{1000}{15}$ requests per second, and therefore with the blocking, we will do

$$\left(1 - \frac{5}{8}^n\right) \cdot \frac{1000}{15} \text{ requests per second}$$

17th October 2019

This starts with the tutorial question in the last lecture.

Kernel Threads

The advantages of kernel threads are that it can easily accommodate blocking calls, such as I/O, allowing for other threads in the process to be scheduled. However, this has more scheduling overhead, as we need to transition to and from kernel space. This also causes synchronisation to become more expensive, as well as switching before more expensive (still remains cheaper than process switches). We are also stuck with what the kernel gives us in terms of scheduling.

An approach for to take advantage of both types of threads is to use kernel threads and multiplex user-level threads onto some / all of the kernel threads. This allows multiple user threads on a single kernel thread.

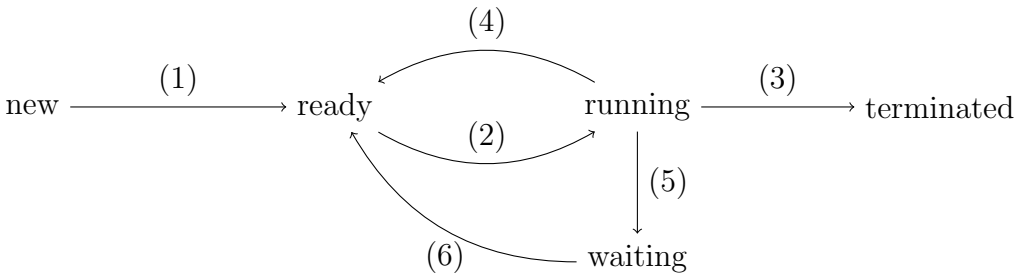
1. If in a multithreaded web server the only way to read from a file is the normal blocking `read()` system call, do you think user-level threads or kernel-level threads are being used?

   Kernel-level thread, as it loses the point of being a multithreaded web server if the entire application blocks on a file read.

Process States

The states of a process are as follows;

- new — the process is being created
- ready — runnable and waiting for the processor
- running — executing on a processor
- waiting (blocked) — waiting for an event
- terminated — process is being deleted



(1) once the process has been initialised / enabled (PCB created) and exists as an entity

(2) selected by the scheduler to execute

(3) exits in some way

(4) preempted - scheduler decides to stop running a process on a CPU core and returns it to the ready state

(5) performing some blocking I/O operation

(6) after the blocking operation completes

Scheduling

The states above are for a single process, and as such, multiple processes can be in the ready state (able to run on a CPU core, but not running). The job of the scheduler is to decide which one should run. A scheduling algorithm has the following properties;

- ensure fairness — all processes are "competing" for CPU time
- avoid starvation — no process should never be assigned to the running state
- enforce policy — may need to respect priorities
- maximise resource utilisation — make sure all CPU cores are busy
- minimise overhead
- system specific;
  - batch systems — e.g. a compute cluster
    we want to minimise the time between job submission and completion (turnaround time), and maximise throughput (the number of jobs per unit of time)
  - interactive systems — desktop system with UI
    we want fast response times
  - real-time systems

We also need to consider the types of scheduling;

non-preemptive

- cannot stop it until it stops itself
- let a process run until it blocks or voluntarily releases CPU

preemptive (most modern operating systems)

- let a process run for a maximum amount of fixed time
- requires a clock interrupt

We can also classify the nature of processes;

CPU-bound

- bottlenecked resource is the CPU (most of the time it is doing computation)
- performance limited by how fast it can run computations

I/O-bound

- occasionally uses CPU
- most of time is spent waiting for I/O
- for example, a terminal waiting for user to enter command

Some common scheduling algorithms are as follows;

- **first-come-first-served** (non-preemptive)

  The ready state is kept as a queue, and new processes are added to the back of the queue. The head of the queue is the next process to be scheduled, and when a waiting process finishes waiting it is added to the back of the queue.

advantages

- no indefinite postponement as all processes are eventually scheduled
- very easy to implement

disadvantages

- in the case a long job is followed many short jobs, head of line blocking occurs, and the average turnaround time suffers

- **round-robin scheduling**

  The general structure is similar to first-come-first-served, but we have the addition of preemption. We keep a process running until it blocks (like in FCFS), but we also preempt it, and place it in the back of the ready queue, once it exceeds some time quantum.

  advantages

  - fair due to ready jobs getting equal share of CPU
  - good response time for a small number of jobs

  disadvantages

  - low turnaround time when run-times are different (a short job would need less time quanta)
  - poor turnaround time when run-times are similar (all finish at the same time)

  However, we need to decide on the round robin quantum (time slice). For example, with a quantum of 4ms, and 1ms for context switching, 20% of the time becomes overhead. For a 1s quantum, only 0.1% is overhead. Therefore for large quantum, there is less overhead, but a worse response time (as the quantum approaches infinity, we go back to FCFS). The reverse is true for small quantum. The typical values lie between 10ms-200ms, Linux uses 100ms, Windows client uses 20ms, and Windows server uses 180ms.

- **shortest job first** (non-preemptive)

  If we know all the run-times in advance, we can pick the jobs that require the least CPU time first. This method is optimal when all the jobs are given simultaneously,

- **shortest remaining time**

  This is a preemptive version of SJF - when a new process arrives with a shorter execution time than the remaining time for the currently running process, it should be run. This allows new short jobs to get good service.

  However, these two methods require knowing the run-times, which isn't always possible.

Some scheduling algorithms take priority into account (priority scheduling). The priority of a job may be defined by the user, or based on some metrics determined by the OS. They can also be static (and remain constant) or dynamic (changes during execution). The goal is to run jobs based on their priority.

In general, we want to favour short and I/O bound jobs - this allows for good resource utilisation and short response times (I/O bound jobs are waiting anyways, and therefore don't need much CPU time). A general-purpose scheduler can quickly determine the nature of a certain job, and then adapt to those changes.

Multilevel Feedback Queues (MLFQ)

A form of priority scheduling is a multilevel feedback queue, which is implemented by many operating systems. This has a queue for each priority level, and runs a job from the highest non-empty priority queue, usually using round-robin. However, this has the issue that if high priority jobs keep being added, then something of a lower priority might never be run, leading to starvation. A way around this is to have a feedback mechanism in place, where the job priority is recomputed periodically based on how

much CPU they have used recently. This is an exponentially-weighted moving average. Additionally, a job's priority it should increase as it waits.

However, this has a few drawbacks;

- priorities make no guarantees - assume a system of 16 queues, and a job is given a priority of 15, this can mean nothing if there are many jobs of priority 16
- priority assignment requires a warm-up period, when the operating system needs to work out what the job does
- cheating is a concern - a program may issue I/O requests to boost priority
- cannot donate priority

Lottery Scheduling

By *Waldspurger and Weihl*. Jobs receive lottery tickets for the resources they need (such as CPU time). At each scheduling decision, one ticket is chosen at random, and the job holding that ticket wins. Priorities in this scheme are done by biasing the number of tickets - in a system with 100 tickets for CPU time, and giving a job 20 tickets means that it will have 20% of the CPU time in the long run. This also has additional nice properties;

- no starvation (as every job will almost certainly be done at some point)
- highly responsive, as it will have the number of tickets needed to get a certain percentage chance of getting the resource at the **next** decision
- supports priority donation, as a process can give tickets to another
- adding jobs / removing jobs affects other jobs proportionally

However, the main obvious drawback is the unpredictable response time, and if an interactive process is unlucky, it can be unresponsive.

23rd October 2019

Tutorial Questions

State which of the following are true and which are false, justifying answers.

1. Interactive systems generally use non-preemptive processor scheduling.

   False. They use preemptive scheduling to guarantee a fast response to new requests. Service trivial, I/O-bound, interactive requests quickly.

2. Turnaround times are more predictable in preemptive than in non-preemptive systems.

   False. In non-preemptive systems, a process will run to completion or until it blocks once it gets a processor.

3. One weakness of priority scheduling is that, while a system may faithfully honour the priorities, the priorities themselves may not be meaningful.

   True. The (actual) priority of a job, and how meaningful it is, often depends on what other jobs are running.

Synchronisation

One example of synchronisation we've already seen is the joining of two `pthreads`. Note that we can often use processes and threads interchangeably, as the concepts are relevant to both. A lot of the system calls that the kernel exposes for synchronisation are exposed through programming languages, as the language must have the ability to control threads.

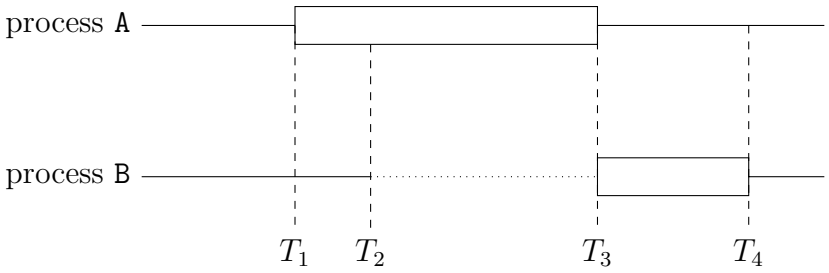
Mutual Exclusion

This goes through a standard example of race conditions due to shared data;

```
1 void extract(int acc, int sum) {
2   int b = accs[acc];
3   accs[acc] = b - sum;
4 }
```

The code above is a critical section (processes access a shared resource), and we need mutual exclusion (such that ensures that if a process is in a critical section, no other process can execute it, hence processes must request permission to enter). Therefore, some synchronisation mechanism is required at the entry and exit of this section. The requirements for mutual exclusion are as follows;

- no two processes may be simultaneously inside a critical section
- no processes running outside the critical section may prevent other processes from entering the critical section (any process requesting permission to enter should be allowed to when there is no process inside that section)
- no process should be delayed from entering the critical section forever
- cannot assume about the progress of processes (while it may be easy to assume that two threads are making the same progress, it is really up to the scheduler)



T_1: `A` enters the critical region

T_2: `B` attempts to enter the critical region, `B` is blocked

T_3: `A` leaves the critical region, `B` is unblocked, and enters the critical region

T_4: `B` leaves the critical region

Some methods of preventing this are as follows;

- **disabling interrupts**

  A very simple way of doing this is to disable interrupts; therefore we can have `CLI()` before line 2, and `STI()` after line 3. As no timer interrupts can occur, the processor cannot context switch to another thread. This has some major issues;

  - only works on single-processor systems, as we have true parallelism (such that multiple processes can truly run at the same time - no context switching needed) with multiple processors
  - slows down the system, as nothing else can run during that time
  - because there is no way for the kernel to take back control, a bug in this critical section cannot be recovered from - this mechanism is typically only used by kernel code

- **strict alternation** software solution

  The idea here is to maintain a global `turn` variable. While the thread is not on its turn, it simply "busy waits" for the variable to change to its turn. Once it is, it can then assume that any other thread attempting to access the critical section is now waiting in the loop. After it has finished execution, it can change the turn.

thread 0

```
while (true) {
  while (turn != 0)
    /* busy wait */;
  critical();
  turn = 1;
  noncritical0();
}
```

thread 1

```
while (true) {
  while (turn != 1)
    /* busy wait */;
  critical();
  turn = 0;
  noncritical1();
}
```

This also has issues;

- by doing this we've assumed a form of alternation, that it switches between the threads (switches from thread 0 to thread 1, and vice versa); this means that if thread 0 wishes to enter the critical region again, after finishing a short non-critical region, it must wait for thread 1 to enter the critical region and set `turn`

  thread 1 can take a long time in its non-critical region, causing non-critical code to prevent entry to critical code
- we are also performing a busy wait - this wastes CPU time as we are continuously checking a global variable, therefore we need kernel support to prevent this

- **Peterson's solution** software solution

```
int turn = 0;
int interested[2] = { 0, 0 };

void enter_critical(int thread) {
  int other = 1 - thread; // thread is 0 or 1
  interested[thread] = 1;
  turn = other;
  while (turn == other && interested[other])
    /* busy wait */;
}

void leave_critical(int thread) {
  interested[thread] = 0;
}
```

While this still uses the global `turn`, we have an additional `interested` variable. Note that when a thread enters, it marks that it is interested. When both thread 0 and thread 1 attempt to enter the section, turn only allows one thread to enter. If thread 0 is in the critical section, then thread 1 must wait for thread 0 to set interested to 0, which can only happen after thread 0 leaves, and vice versa.

- **lock variables**

We can utilise a TSL (test and set lock) instruction, which is an atomic instruction provided by most CPUs. `TSL(LOCK)` atomically sets the memory location `LOCK` to 1, and returns the old value. Note that locks that rely on busy waiting are called **spin locks** - these can still be used if we have a very short wait time, as we don't need to handle the overhead from context switching. Spin locks are still used by the kernel, as it cannot use a blocking abstraction.

It's also important to consider lock granularity (the amount of data a lock is protecting). Note that in the `extract` example at the start, attempting to withdraw from different accounts shouldn't interfere with each other, and therefore it shouldn't be a global lock, but rather a lock per account.

Similarly, we should also consider the overhead of using locks, such as the memory space from storing data about them, the time used for initialisation, and the time needed to acquire and release them. With higher lock contention (the number of processes waiting for a lock), we have less parallelism.

coarser granularity

- less lock overhead (less locks)
- more lock contention
- less complex to implement

finer granularity

- more lock overhead
- less lock contention
- more complex to implement

To maximise concurrency, we need to choose a finer lock granularity (understanding the tradeoffs). The goal is to make the critical sections smaller, and release locks as soon as they aren't needed. For example, in the code below, we should release the outer lock `L_accs` after creating the account, as it is only needed for that part.

```
1 void addAccount(int acc, int balance) {
2   lock(L_accs);
3   createAccount(acc);
4   lock(L[acc]);
5   accs[acc] = balance;
6   unlock(L[acc]);
7   unlock(L_accs);
8 }
```

Additionally, we should differentiate between locks held for reading and writing. Two threads attempting to **read** the same data should be allowed to do so, and it reduces parallel unnecessarily if they block each other. `lock_RD(L)` acquires lock `L` in read mode, and `lock_WR(L)` acquires it in write mode. In write mode, no other thread can acquire either a read or a write lock, however multiple threads can acquire a lock in read mode.

Priority Inversion

Assume we have two processes, `H` and `L` with high and low priority, respectively. Our scheduler should always schedule `H` if it is runnable. Now, `H` is waiting for I/O, is therefore blocked, and `L` is scheduled. `L` acquires lock `A` for a critical section. I/O arrives for `H`, and it is unblocked, `L` is preempted and `H` is scheduled. `H` then attempts to acquire lock `A`, but `L` is holding that lock.

If we were to use a busy wait in software, the kernel does not know that `H` is blocked, and will continue to schedule it, thus not allowing `L` to be scheduled, and the lock is never released. This is called priority inversion, as a higher priority process is being blocked from running by a lower priority process.

Therefore, preemptive scheduling needs to take into account the lock implementation and mutual exclusion.

Race Condition

This occurs when multiple threads or processes read and write shared data, and the final results depends on the relative timing of their execution (on the exact process or thread interleaving).

Consider the following three threads (tutorial question);

`T1: a = 1; b = 2`

`T2: b = 1;`

`T3: a = 2;`

1. How many possible interleavings are there? 12 interleavings
2. If all thread interleavings are as likely to occur, what is the probability to have `a = 1`, and `b = 1` after all threads complete execution?

   $\frac{1}{12}$, as `T2` must occur after `T1`, and `T3` must occur before `T1`.

From this, we can see why multithreaded applications are difficult to debug, as the results can be unpredictable (and only occasionally cause bugs).

Memory Models

It's important to remember that modern CPUs can execute instructions out of order in the interest of performance. We've assumed the operation of each thread appear in program order (and each operation executes atomically). This is not necessarily what the CPU or the compiler assumes, and can lead to unexpected behaviour. Therefore, we should not rely on expected behaviour of a memory model, and just avoid data races (such that they are protected and will work regardless of the model). We assume strong memory models in this course.