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

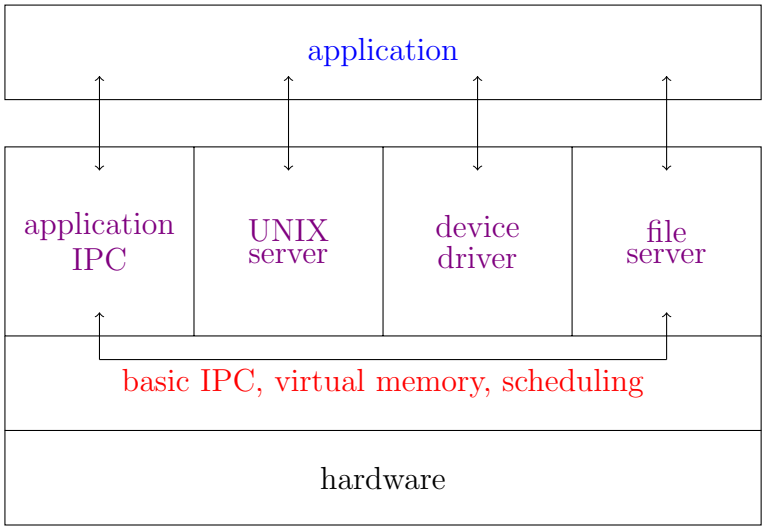
  - efficient calls within the kernel, as there it remains in kernel mode
  - flexible to write kernel components due to the shared memory (direct access with no limit to APIs)

  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)

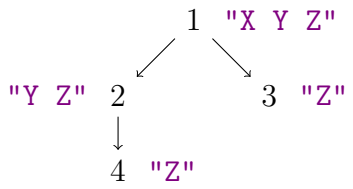
In UNIX `int fork(void)` creates a new child process, which is an exact copy of the parent process, inheriting all resources, and executed concurrently - however, different virtual address space. `fork` will return twice, however in the parent process it will return the child's process ID, but in the 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.

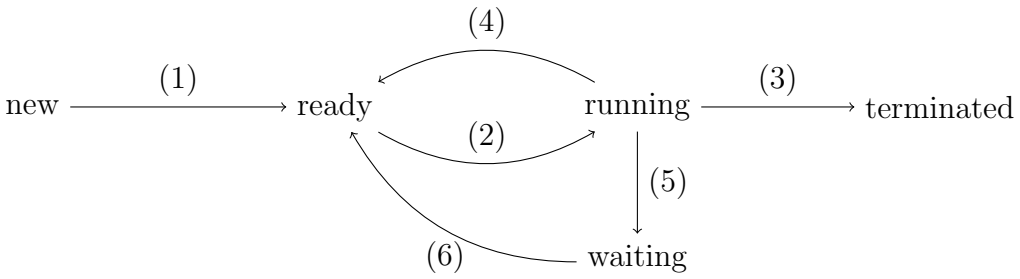
1. If in a multithreaded web server the only way to read from a file is the normal blocking `read()` system call, do you think user-level threads or kernel-level threads are being used?

   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.

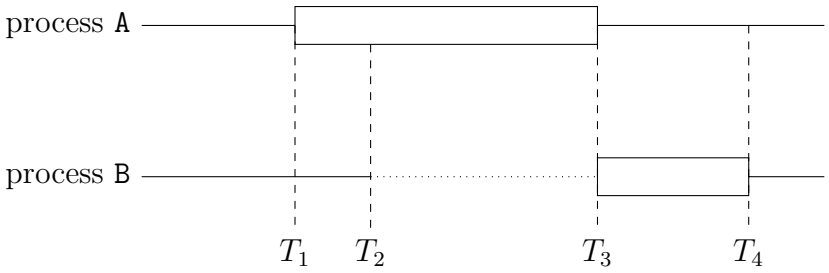
Mutual Exclusion

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.

24th October 2019

Happens-Before Relationship

In order to formalise the execution of events, we think about instructions that are executed as events in a trace. We then have a partial ordering denoted by $a \to b$. Consider two events, a, b, with a occurring before b in the trace;

- if a and b are in the same thread, then $a \to b$
- if a is `unlock(L)`, and b is `lock(L)`, then $a \to b$ (this can be used to enforce an ordering between threads)

This has the following properties;

- $\forall a.\ a \not\to a$ irreflexive
- $\forall a, b.\ a \to b \Rightarrow b \not\to a$ antisymmetric
- $\forall a, b, c.\ a \to b \wedge b \to c \Rightarrow a \to c$ transitive

Therefore, we can formally define a data race between a and b in the trace if and only if;

- they access the same memory location
- at least one is a write
- they are unordered according to the relation we just defined

When the ordering is drawn, we are assuming a particular execution order between threads (in terms of locks) - for example we can assume `T1` runs before `T2`, and therefore the lock in `T2` happens after the unlock in `T1`, but it can also be the other way around; therefore to notice race conditions we may still need to enumerate the executions.

```
1 int a, int b;
2 void T1 {                          void T2 {
3   a++;                               lock(L);
4   lock(L);                           b++;
5   b++;                               unlock(L);
6   unlock(L);                         a++
7 }                                  }
```

This is safe if the `lock(L)` in `T2` is after the `unlock(L)` in `T1`, however if `T2` were to lock first, then there is a data race between `a++` in both `T1` and `T2`.

Semaphores

A semaphore can be thought of as a signalling mechanism between two threads. A process will stop, waiting for a specific signal, and a process will continue if it has received a specific signal. The semaphore `s` can be accessed via these atomic operations;

- `down(s)` — waiting to receive a signal
- `up(s)` — triggering and sending a signal
- `init(s, i)` — initialising a semaphore

Semaphores have two private components; a counter (which is a non-negative integer), and a queue of processes currently waiting for that semaphore.

```
1  init(s, i): counter(s) = i
2              queue(s) = { }
3
4     down(s): if counter(s) > 0
5                counter(s) = counter(s) - 1
6              else
7                add P to queue(s)
8                suspend P
9
10       up(s): if queue(s) not empty:
11               resume a process in queue(s)
12             else
13               counter(s) = counter(s) + 1
```

This can be used to perform the following;

- **mutual exclusion**

  We can use a semaphore to implement mutual exclusion - via the use of a binary semaphore (where we initialise it to 1). Therefore a process can acquire the "lock" with `down(s)`, perform work in the critical section, and then release the "lock" with `up(s)`.

- **ordering events**

  ```
  process A
    ...
    (critical section)
    up(s)
    ...
  end
  process B
    ...
    down(s)
    (critical section)
    ...
  end
  ```

  With the semaphore initialised to 0, this forces `process A` to execute its critical section before `process B` can execute its critical section, forcing an ordering.

- **multiple producers and multiple consumers**

  You can consider the number of processes that are allowed into the critical region as the initial value of the counter.

Consider the scenario where there is a shared data structure, such as a buffer, a set of producers (threads that are writing into the buffer), and a set of consumers (threads reading from the buffer if there are new elements). We then have the following constraints;

- producer
  - items can only be deposited in the buffer if there is space (block if it is full)
  - items can only be written if mutual exclusion is ensured (we don't want writes to interfere)
- consumer
  - items can only be fetched if buffer is non-empty (block if empty)
  - items can only be read if mutual exclusion is ensured (don't want to read incomplete items)
- buffer can hold between 0 and N items

```
1  var item, space, mutex: Semaphore
2  init(item, 0) // signalling between producer and consumer
3  init(space, N) // ^
4  init(mutex, 1) // initialised to 1 to ensure mutual exclusion
5
6  procedure producer()
7    loop
8      (produce item)
9      down(space)
10     down(mutex)
11     (deposit item)
12     up(mutex)
13     up(item)
14   end
15 end
16
17 procedure consumer()
18   loop
19     down(item)
20     down(mutex)
21     (fetch item)
22     up(mutex)
23     up(space)
24     (consume item)
25   end
26 end
```

Looking at the producer, we can see it performs the following steps;

(1) it first produces an item
(2) it attempts to `down` the `space` semaphore, it blocks if the buffer is full and will resume once a consumer reads
(3) it then enters the critical region, deposits the item, and exits
(4) it `up`s the `item` semaphore, signalling that there is an item

On the other hand, the consumer does the following;

(1) it attempts to `down` the `item` semaphore, if it is 0, there are no items to be read, and will block until a producer deposits

(2) it then enters the mutual exclusion zone, fetches it, exits, and frees space in the buffer, finally consuming the item

Monitors and Condition Variables

This is a higher level synchronisation primitive. The monitor protects shared data, and has a procedure that must be called to enter the monitor from outside. There are then internal procedures that can only be called from monitor procedures (the monitor itself is implicitly protected by a lock), and has one or more condition variables. Processes can only call entry procedures, and cannot directly access internal data - one process in the monitor at a time.

Condition variables are associated with high-level conditions, similar to what we used for the final example in semaphores, such as "some space has become available". This can be thought of as a signalling mechanism that something has occurred between two threads. It has the following operations;

- `wait(c)` release monitor lock, and waits for `c` to be signalled
- `signal(c)` wakes up a process waiting for `c`
- `broadcast(c)` wakes up all processes waiting for `c`

Unlike semaphores, signals do not accumulate, therefore if `signal` is called with no waiting thread, the signal is lost. Hence if `signal` was called before `wait`, then the `wait` would have to wait until the next `signal` is called (can be indefinite if no `signal` is called in the future - this causes bugs if a thread "misses" its signal). The same model as above can be done with monitors as follows;

```
1  monitor ProducerConsumer
2    condition not_full, not_empty
3    int count = 0
4
5    entry procedure insert(item)
6      while (count == N) wait(not_full) // block until signalled if full
7      insert_item(item)
8      count++
9      signal(not_empty) // wakes up a waiting thread
10
11   entry procedure remove(item)
12     while (count == 0) wait(not_empty)
13     remove_item(item)
14     count--
15     signal(not_full)
16 end monitor
```

Note the use of `while` as the condition, instead of `if` as we always need to re-check the condition. There is some subtlety on what happens on a signal;

- Hoare a process waiting for a signal is immediately scheduled
  - easy to reason about
  - inefficient - the signalling process switches out even if it has not finished yet with the monitor
  - places extra constraints on scheduler
- Lampson sending signal and waking up from a wait is not atomic
  - harder to understand, need to take extra care when waking up from `wait` (hence re-checking)
  - more efficiently, no constraints on scheduler
  - more error tolerant, if condition being notified is wrong, simply discarded when rechecked

This is usually used.

Two threads in the same process can synchronise using a kernel semaphore only if they are implemented by the kernel, because the kernel needs to be able to see the threads and manipulate them.

Deadlocks

A set of processes is deadlocked if each process is waiting for an event that only another process can cause. This is similar to a data race in the sense that this may not always happen, and execution may be able to proceed as expected. Consider the two processes;

P0:

```
1 down(scanner);
2 down(cd_writer);
3 scan_and_record();
4 up(cd_writer);
5 up(scanner);
```

P1:

```
1 down(cd_writer);
2 down(scanner);
3 scan_and_record();
4 up(scanner);
5 up(cd_writer);
```

If P0 downs the `scanner`, and then it context switches to P1, which downs the `cd_writer`, they are in deadlock. The each want to down the semaphore the other have downed, but neither can progress. For resource deadlock, the most common, these 4 conditions must hold;

- mutual exclusion — each resource is either available or assigned to one process
- hold and wait — process can request resources while it holds other resources
- no preemption — resources given cannot be forcibly revoked
- circular wait — two or more processes in a circular chain, waiting for a held resource

Brief tutorial questions;

1. Can the set of processes deadlocked include processes that are not in the circular chain in the corresponding resource allocation graph?

   Yes - a process may be waiting on a resource that is held in a circular chain, but not holding a resource (that's needed in the chain)

2. Can a single-processor system have no processes ready and no process running? Is this a deadlocked system?

   Yes (all processes waiting for I/O) - but this is not a deadlocked system. Typically, we have an idle process created by the kernel that runs when there are no other processes ready (this powers down the core / lowers the core frequency). Therefore we cannot see if we are in a deadlock just by checking that there are no processes running and none are ready.

Resource Allocation Graph and Dealing with Deadlocks

This is a directed graph that models resource allocation, an arc from a resource to a process means that the process owns that resource, and an arc from a process to a resource means that the process is blocked waiting for that resource. If a cycle is present, then there is a deadlock. Some strategies are as follows;

(1) **ignore it** — when it is such a rare occurrence, it can be ignored

(2) **detection and recovery**

Dynamically build the resource ownership graph and look for cycles. Mark a visited arc, if we encounter a visited arc, then a cycle is present.

Once we've figured out we have a deadlock, we can preempt a process in the cycle - however this can break the program. A method that is less damaging would be to rollback to a previously checkpointed state, redo the computation, ensuring that the same scheduling decisions aren't made. Another strategy is to kill a random process in the cycle, assuming that the process can handle it.

(3) **dynamic avoidance** decide whether it is safe to grant access to resource when requested

Banker's Algorithm, by *Dijkstra*. Essentially, it models resources available similar to how a bank can give customers (threads) a credit limit, and the bank can reserve less than the sum of all the credit limits. Each process can have less than, or equal to, its limit.

A state is considered safe if there are enough resources to satisfy the maximum request from any customer. This creates some sequence of allocations that guarantees that all customers can be satisfied. For example, in the state below (with 2 free resources), it is safe, as we can satisfy `C`, which frees all 4 resources, allowing us to satisfy `B` or `D`, and so on.

	has	max
`A`	1	6
`B`	1	5
`C`	2	4
`D`	4	7

(4) **prevention** ensure at least one of the four conditions cannot occur

See next lecture.

6th November 2019

Deadlock Prevention

- attacking mutual exclusion condition — share the resource if it is safe
- attacking the hold and wait condition

  Requires all the processes to request resources before starting, however this has an issue as the process needs to know what it will need in advance (not realistic since we want finer lock granularity). Block if not all resources are available.
- attacking no preemption condition — difficult for a programmer to reason about
- attacking circular wait condition

  Force all processes to ask for resources in order - however this can be difficult to organise. This means that the process holding the highest (in this ordering) resource will never ask for a resource that has already been assigned.

Communication Deadlock

This can also happen over a network. For example, let `A` send a message to `B`, and blocks until `B` replies. If `B` never gets the message, due to some failure, `B` is blocked waiting for a message, and `A` is blocked as `B` will not reply. A common method of dealing with this is to use timeouts, and performing some recovery action if the request times out.

Livelock

This occurs when the processes and threads aren't blocked, but they are not making progress. For example, let `A` acquire `resource1`, and `B` acquire `resource2`. Now `A` tries to acquire `resource2`, but fails, releases locks, and then reacquires `resource1` - this cycle continues to happen.

Starvation

A particular thread can be **starved** when the scheduler makes biases against it. If it is never scheduled to get the resource, it cannot make any progress - a fair scheduling algorithm prevents this, however a priority based scheduling algorithm may cause this.

Tutorial Questions

1. Is this system in a safe state? (available: `A`: 2, `B`: 3, `C`: 0)

process	allocation			need		
	`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

   Yes, this system is in a safe state. Note that for this I will be using the notation (`A`, `B`, `C`) to denote how much we have available for each resource. We start at (2, 3, 0), by satisfying `P2` and freeing, we have (5, 3, 2), satisfying `P5` gives us (5, 3, 4), satisfying `P4` gives us (7, 4, 5), satisfying `P3` gives us (10, 4, 7), and finally satisfying `P1` gives us the expected (10, 5, 7).

2. Can we accept `P1`'s request for 2 instances of `B`?

   No, if we're at a point when we only have (2, 1, 0), nothing can be allocated its maximum resource.

3. Two processes, `A` and `B`, each need three records, 1, 2, and 3, in a database. If `A` asks for them in the order 1, 2, 3, and `B` asks for them in the same order, deadlock is not possible. However, if `B` asks for them in the order 3, 2, 1, then deadlock is possible. With three resources there are $3! = 6$ possible combinations each process can request resources. What fraction of all combinations is guaranteed to be deadlock free?

   Assuming that `A` does ask for it in the order 1, 2, 3, if `B` requests 1 first, then it will block. Whichever of the two processes acquires 1 first will win and run to completion. Therefore, $\frac{1}{3}$ of cases are deadlock free.

7th November 2019

This starts by answering the last question in the previous lecture.

Memory Management

Every instruction that is executed will involve at least one memory access. This is because the instruction itself is stored in memory. The operating kernel needs to provide memory allocation as well as memory protection (isolation between processes, and containing failures). It also needs to abstract away from the hardware. It's also important to consider the memory hierarchy, as it gets larger, it tends to be slower and have more latency.

Logical vs. Physical Address Space

Physical addresses are the actual addresses used to access DRAM (physical system memory), typically the program will not deal with physical memory addresses, and only deals with logical memory addresses. The memory management system binds local address space to physical address space.

For this to be done quickly it is typically built into hardware. The memory management unit (MMU) maps logical to physical addresses, for example a simple method is to have a relocation register, the

contents of which are added to the logical address in order to calculate the physical address. The benefit of doing this is that we can, at compile time, give it logical addresses for it to use. This can then be restricted at runtime by the hardware to a physical address range. Another program may also be running at the same time, with the same logical addresses, but this can then be mapped to a different location on physical memory - thus sharing physical memory between processes.

Contiguous Memory Allocation

We typically split the main memory into two partitions, kernel memory (usually held in low memory, with the interrupt vector - well known address for the hardware to find the interrupt handler), and user memory (typically held in high memory). This separation must be maintained, such that the user memory cannot access kernel memory and corrupt the state of the machine. We need a base register for the smallest physical address, as well as a limit register which restricts the amount of memory we can give to a process. Therefore the physical range for a process' addresses are from the base register, to the base register + the limit register. However, this approach has drawbacks - if a process were to require more memory, it couldn't overwrite the process located after it, meaning it would have to move a large chunk of memory. Additionally, when a process is freed, it could lead to memory fragmentation;

- external fragmentation — there is enough memory, but not together (contiguous)
- internal fragmentation — able to find contiguous slot, but will have leftover space

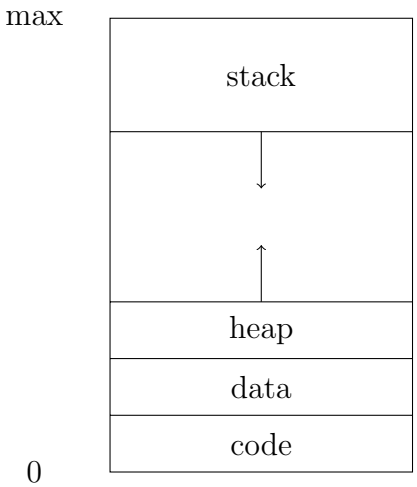
While this is very limited, it is still occasionally used. For example, this can be used in embedded system because we know exactly what we're running, and the amount of memory it requires.

Swapping

A process that isn't running (blocked, etc) does not have to exist in physical memory, and can be stored somewhere else (on disk) to make space for other processes. This is the swap space, which is a file or partition on the disk. However, reading and writing from disk has some latency, which gives a hit to performance. This will be revisited in demand paging.

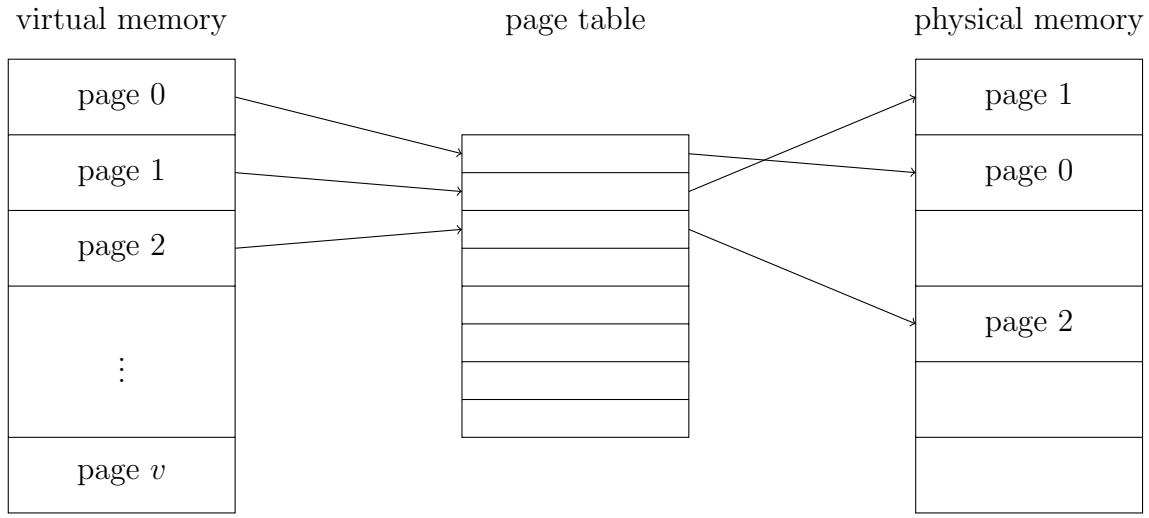
Virtual Memory with Paging

Instead of thinking of the memory we allocate to processes as a single fixed block of contiguous memory, we now subdivide memory into a number of fixed sized pages, which lives in virtual memory. The address space of the virtual / logical memory is typically much larger than the physical address space (the former is determined by how many bits we use to represent a memory address - 64 bits on modern hardware). These pages are then mapped to virtual memory, but do not have to be contiguous (for example, page 1 does not have to follow page 0, when mapped to physical memory). This however requires a more complex data structure. The virtual address space for a given process typically looks like the following;



The process is able to use the entire virtual address space, as if it were the only user. If a process attempted to access a part of virtual memory that isn't mapped, it would result in a segmentation fault.

Pages are fixed size blocks of virtual memory that live in the virtual address space. Once a page is mapped to physical memory, the fixed size block of memory is a page **frame** (the size of a frame and the size of a page are the same, and fixed). For example, we can think of all page sizes as 4KB blocks. To load a binary of some size, we'd calculate n pages to load the binary into memory, and find n free frames. The page table is then set up to translate logical to physical addresses.



Tutorial Questions

1. What is the advantage of a paged virtual memory system with;

   (a) a small page size — less unused memory (less fragmentation)

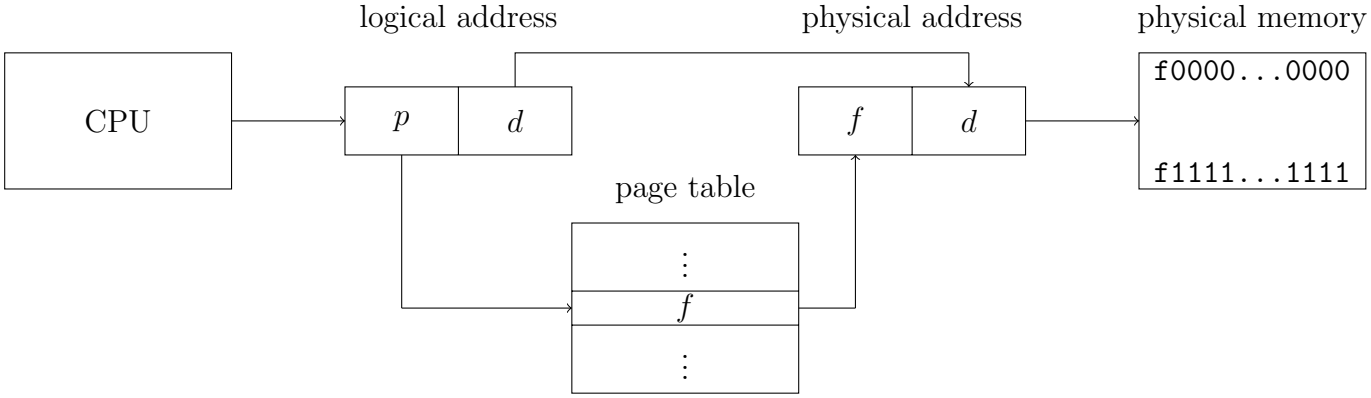
   (b) a large page size — less entries in page table (less overhead, and faster access)

2. Describe how a context switch affects the virtual memory system.

   The page table needs to be changed, as the page table is process specific. This updates the base register in the MMU. It also needs to clear cached address translations.

Address Translation

Memory is byte addressable, and therefore memory addresses should refer to a specific byte in memory. For a logical address space 2^m, and a page size of 2^n, the address generated by the CPU should be divided into the page number p (which is used as the index into the page table, which contains the base addresses of pages in physical memory), and the page offset d, which is which byte in the page we want to access. The page offset is the least significant (last) n bits, and the page number is the remaining $m - n$ bits. Due to the size of the frame being the same as the size of the page, the offset does not need to change;



Note that it's also important to maintain a list of free frames, which are then taken to update the page table for a new process.

Memory Protection

In the page table, we attach a valid-invalid bit to each entry;

- **valid** indicates a legal page (has been allocated, and is mapped to physical memory)
- **invalid** indicates the page is missing, either from the page not being in the process' virtual address space (page fault), or it exists but needs to be loaded from disk (demand paging) - kernel deals with this page fault

As each page table entry is just the frame address (as the offset is discarded), this can be stored in that part as simple book-keeping data.

Tutorial / Exam Question

An embedded system uses a 16-bit big-endian architecture. It supports virtual memory management with a one-level page table. It has a page size of 1 KByte. The least significant bit of each page table entry represent a valid bit; the second least significant bit is the modified (dirty) but. The following are the current entries in the page table;

```
0x2C00
0x2403
0xCC01
0x0000
0x7C01
```

Translate the following virtual memory addresses to physical memory addresses using the page table given above (if possible);

(i) `0xB85` `0b0000101110000101`

Looking at the first 6 bits, it's in page table entry 2, hence it is address `0b1100111110000101`, which is `0xCF85`

(ii) `0x1420` `0b0001010000100000`

Looking at the first 6 bits, it's in page table entry 5, which doesn't exist, hence it page faults (beyond end)

(iii) `0x1000` `0b0001000000000000`

Looking at the first 6 bits, it's in page table entry 4, hence it is address `0b0111110000000000`, which is `0x7C00`

(iv) `0xC9A` `0b0000110010011010`

Looking at the first 6 bits, it's in page table entry 3, which is marked as invalid.

The page size is 2^10 bytes, hence the least significant 10 bits are used for the offset.

13th November 2019

This starts with the exam question shown last lecture. Note that all of the translations we just manually performed must be done at every memory access. This overhead is outweighed by the flexibility of virtual memory.

Page Table Implementation

We need to look at the representation of the page table, as it can grow very large, as well as handle the performance impacts caused by the overhead of using page tables. The page table is kept in main memory, and the page table base register (PTBR) points to the base of the page table, and the page table length register (PTLR) indicates the size. This needs to be changed when the processor switches processes.

As previously mentioned, we now need to perform two memory accesses per data access; one for the page table, and one for the actual data. Most modern CPUs cache frequently translated memory addresses - this is done in hardware (in order to achieve very high performance), and uses this cache as associative memory (supporting parallel search). This is referred to as the translation look-aside buffer (TLB). The cache can be thought of as a table which holds the page number and frame number - before accessing the page table on memory, it checks if the page is in associative register, and if it is; it can obtain the frame number, otherwise the frame number has to be obtained from the table in memory. Some also store address-space identifiers (ASIDs) in entries, which uniquely identify each process to provide address-space protection. This cache needs to be flushed on a context switch (this is another overhead of context switches; initially the memory access will be slower) - the kernel can be mapped into every virtual address space to prevent this issue.

We can calculate the effective access time as follows;

$$
\begin{aligned}
\text{associative lookup} &= \epsilon \\
\text{assume memory cycle time} &= 1 && \mu\text{sec} \\
\text{hit ratio} &= \alpha && \text{page found in associative registers} \\
\text{EAT} &= (\epsilon + 1)\alpha + (\epsilon + 2)(1 - \alpha) \\
&= 2 + \epsilon - \alpha
\end{aligned}
$$

If our hit ratio is close to 1, then we have a very low overhead for paging.

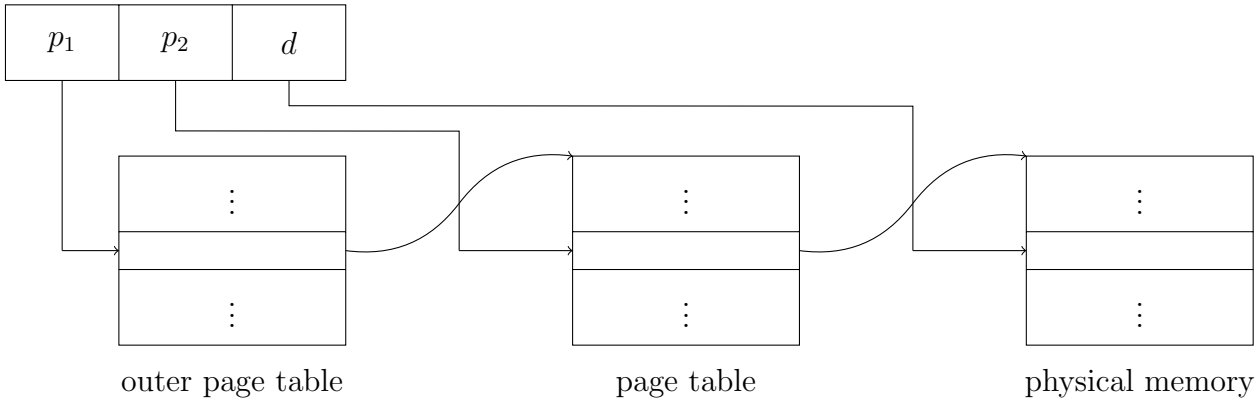
Page Table Size

As we are thinking of the page table as an array, each storing an address (4 bytes on 32-bit, 8 bytes on 64-bit), the sizes can be an issue. For example, on a 32-bit machine with 4KB pages, the page table will be at least 4MB, which is manageable. However, on a 64-bit machine, with 4KB pages, the page table will need 2^{52} entries. At 8 bytes per entry, that will be over 30 million GB. While we will have a very large address space, we are unlikely to actually need it all, and many entries in the page table will be unmapped. Some approaches are as follows;

- hierarchical page table

  In a two-level page table, the outer page table will point to an inner page table (which handles all addresses falling within that range), instead of a frame. The inner page table will only exist if there is at least one address in that range, meaning that we don't need to allocate a page table for the entire address space. From the inner page table, we can get the actual frame.

  In this scenario, we're still assuming a 32-bit machine with a 1K page size. The page offset therefore has to consist of 10 bits, and and a page number of 22 bits. Since we've split the page tables, we also need to split the page number into a 12-bit page number (p_1), and a 10-bit page offset (p_2). p_1 is the index into the outer page table, and p_2 is the displacement within the page that the outer page table is pointing too. This can be expanded further with larger address spaces.



However, this makes memory access even slower, since we need to do more accesses.

- hashed page table

  Ideally we'd want a structure for a page table that grows with the number of frames, which is a data structure that is linear with the physical memory. Here, the page table contains a chain of elements hashing to the same location, and we can search for a match of the virtual page number in the chain. This gives us the exact corresponding physical frame if a match is found. However, this is more difficult to implement as it has to be done in hardware, including the hashing function.

- inverted page table

  Index the page table by the frame address. Each entry then contains a page address and the process identifier. This way the structure grows with the frames, however this increases the time as each lookup now requires a linear search through the table.

Tutorial Question

Assuming that time for a memory access is 100ns, and for TLB access 20ns. Calculate the access times for a four-level paging system assuming a TLB hit ratio of

(a) 80%

The time for translation is $0.8 \cdot (100 + 20) + 0.2 \cdot (500 + 20)$, which is 200ns - thus 100% slower than unpaged memory access (with one level, it is 140ns, thus 40%).

(b) 98%

The time for translation is $0.98 \cdot (100 + 20) + 0.02 \cdot (500 + 20)$, which is 128ns - thus 28% slower than unpaged memory access (with one level, it is 122ns, thus 22%).

14th November 2019

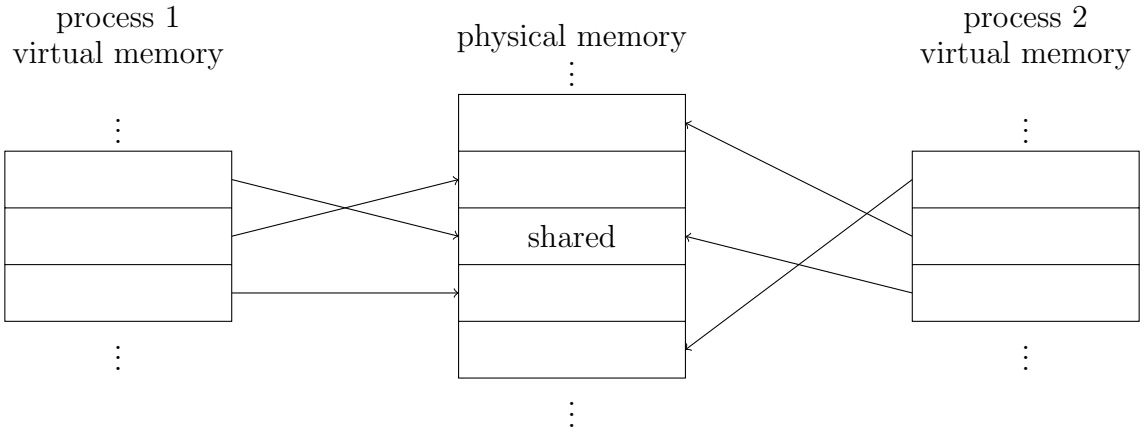
Tutorial Question

1. A pure paging system uses a three level page table. Virtual addresses are decomposed into four fields (a, b, c, d) with d being the offset. In terms of these constants, what is the maximum number of pages in a virtual address space?

   2^{a+b+c}, since we have $2a + b + c + d$ total addresses, and 2^d on each page.

Shared Memory

The idea is that there is memory accessible from two (or more) processes. This creates a mechanism for two processes to communicate. When process 1 attempts to access the shared page, the page table entry will point it to the same frame that process 2 points to, and vice versa.



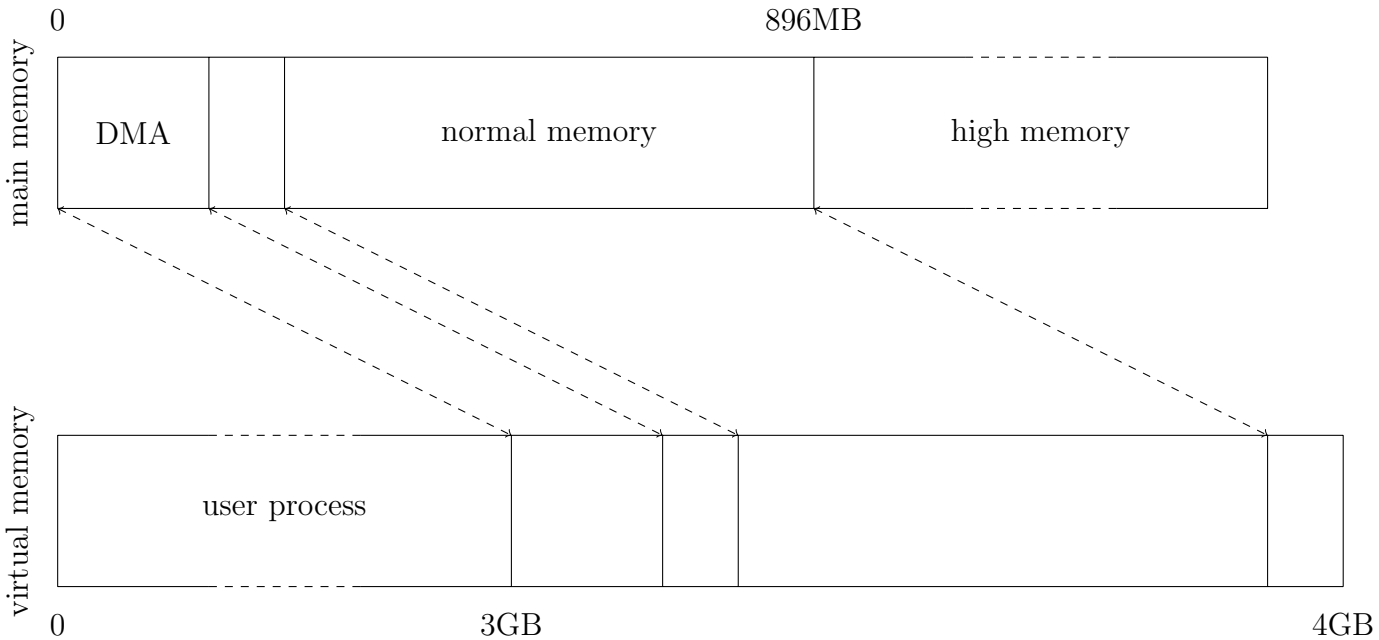
After this is established, it appears as memory both processes can access, therefore there is no need for kernel involvement (hence no need for the context switching overhead). To establish this, there are the following system calls;

call	explanation
`shmget`	allocates a shared memory segment
`shmat`	attaches a shared memory segment to the address space of a process
`shmctl`	change properties associated with a shared memory segment
`shmdt`	detaches a shared memory segment from a process

It would be better for two processes to communicate via pipes due to the flexibility of bi-directional communication, however it's not better for uni-directional communication as there is no synchronisation provided by the kernel (compared to pipes). Because the kernel provides no abstractions for this, synchronisation between two processes using the same shared memory will have to be done in a similar way to two threads concurrently executing on the same memory (locking, etc).

As this can also be mapped to a file (and not just a location on main memory), this can be used for dynamic linking of libraries - thus allowing for libraries to be shared.

Linux Virtual Memory Layout (32-bit)



Note that the n^{th} page of the kernel address space (3GB - 4GB in virtual memory) maps to the n^{th} page frame of main memory. For legacy reasons, we have the interrupt vectors stored at low addresses. DMA is used for direct memory access from I/O devices, such as network cards, to allow them to write to memory, bypassing the CPU. The kernel address space is mapped into virtual memory as well, meaning that when a system call is performed, the page tables don't have to be switched thus removing that overhead. The additional space at the end of virtual memory is used for on-demand mapping, which is done when the kernel wants to access physical memory beyond the 896MB boundary, by creating inserting a mapping into the page table (this isn't needed for 64-bit architectures, as we can map the entirety of main memory). A process attempting to access beyond the 3GB boundary will result in a page fault, as it is essentially attempting to access privileged kernel memory. Another benefit of having a larger address space is that the operating system can randomise the locations of data structures and libraries, making it more difficult for an attacker to locate a vulnerable library.

On IA-32, the page size is usually 4KB, with a 4GB virtual address space, whereas on x86-64, there are larger page sizes (such as 4MB), and up to a four-level page table. The implementation of the two-level page table in x86 is as previously discussed (the outer page table is referred to as the **page global directory**). As the offset bits are unused, it will be used to store the metadata, such as dirty, read / write, etc.

Meltdown (Not Examinable)

```
1 ...
2 if (v == 0) {
3   w = kernel_mem[addr];
4   x = w & 0x01;
5   y = x * 4096;
6   z = user_mem[u];
7 }
```

By speculative execution, it executes the code in the branch. It will not page fault straight away, as that branch may not be reached - however the instructions will be executed. Looking at the last bit of the data stored in the kernel address will result in an access to the user memory at either 0 or 4096, bringing it into cache. The attacker can then check how long it takes to access these (the one with the faster access has been brought into cache). To fix this, Linux moved the kernel address space into its own virtual address space, thus requiring a context switch. Due to the additional complexity, the processor cannot speculate across this.

Demand Paging

The idea behind this is to think of pages for programs which aren't currently running as swapped out. We are then only loading pages on demand; consider the example of running a large binary - we don't need to load the entire binary into memory before execution, as we only need the instructions for the main entry point. When we encounter a page fault, it may still be caused by an actual invalid reference, but also may be caused by the page not being in memory (thus requiring the OS to load it in).

To indicate whether something is in memory, we use the valid-invalid bit, with everything initially set to 0. If it is 1 during address translation, we simply bring it in as before, but if it is 0, we let the OS trap the page fault. This is then checked against another table, if it is still invalid, we abort, otherwise we handle the request. This is done by obtaining an empty frame, swapping the page into the frame, setting the table's valid bit to 1, and restarting the instruction that caused the fault.

We can reason about this in a similar way, with a page fault rate $0 \leq p \leq 1$, where if $p = 0$, we have no page faults. The effective access time is $(1 - p) \cdot$ memory access $+ p \cdot$ (page fault overhead + page swap out? + page swap in + restart overhead). If a free frame is not available, a page may have to be swapped out. Overall, we have a much higher overhead; with insufficient memory, this leads to thrashing due to the I/O overhead the OS must perform for this swapping.

Virtual memory allows us to do the following;

- **copy-on-write**

  This is useful in `fork`, as we share many of the pages (thus it would be very wasteful to copy it all). When either process modifies a shared page, only then do we copy them.

- **memory-mapped files**

  When we memory map a file, associating it with pages in the virtual address space, we bring them in on the first access with demand paging. This is a very efficient way of performing I/O on large files, as we do not need the overhead of system calls.

Page Replacement

When we are out of free frames on main memory, we need to find an unused page that we can swap out. Our policy must minimise the number of page faults (ideally replacing one that will not be used), ensure that we do not over-allocate memory, and ensuring that only modified (dirty) pages to write to disk (we shouldn't write an unmodified page to the disk). The dirty bit is set by hardware on a write. A basic replacement algorithm is as follows;

- find location of desired page on disk
- find a free frame, if one is not found, we select a victim frame
- read the desired page into the frame
- update page and frame tables
- restart the process

Some eviction algorithms are as follows;

- **first-in-first-out (FIFO)**

  This simply evicts the oldest page - this may however replace a heavily used page. This suffers from Belady's anomaly, where we have can have more page faults with more physical frames.

- **optimal algorithm**

  The theoretical optimal is to evict the page that won't be used for the longest time. This obviously cannot be done (perfectly or easily) in practice, but can be used to judge how well another page replacement strategy performs.

27th November 2019

Page Replacement

This continues with page eviction algorithms.

- **least recently used (LRU)**

  The idea for this algorithm is to evict the least recently used (since it's possible that the program no longer requires this page), via the use of a counter (per page entry). On a reference, the clock is copied into the counter, and when a page needs to be replace, we choose to evict the one with the lowest counter. However, this is expensive as we need to copy the entire counter, and we use single register approximations.

- **reference bit**

  This is an approximation of LRU, but does not provide proper order. We associate a reference bit with each page, initially set to 0. When it is referenced, it is set to 1 (can be set by hardware), and on a replacement it attempts to find a page with the bit set to 0. Periodically, all the reference bits are reset to 0.

- **second chance / clock**

  This uses the idea of a reference bit, as well as an order of when pages are brought into memory. When a page is brought into memory, it is stored in a linked list. We have a pointer into the linked list (clock hand) which points at the oldest page. If this page has a reference bit of 0, then we evict it, however if it is 1, we set it to 0, and move to the second oldest page. This continues until we find a page that can be evicted. The idea behind this is that while a page may be old, if it was recently accessed, it may still be in use, and we wouldn't want to evict that.

- **least frequently used (LFU)**

  The previous algorithms do not take into account the frequency of accesses, just that it was or wasn't. For this algorithms, we maintain a count of the number of references. Here we replace the page with the smallest count - however this may replace a page that was recently brought in. Additionally, if there was a heavily used page that is no longer needed, it may not be evicted - we need to reset counters or age the counters.

- **most frequently used (MFU)**

  Replace pages with a large count - if we haven't used it recently, then it's likely it is no longer needed. While this may seem counterintuitive, memory accesses tend to fall within working sets; after a data structure is no longer used, it will no longer experience more references.

Locality of Reference

A property of many programs is that it tends to favour a subset of pages, which it accesses most frequently. There is locality of space as well as time - for example, iterating over a data structure gives spatial locality, and doing this repeatedly gives us temporal locality. This also gives better performance due to the caching in the TLB, preventing the walk through the page tables.

We define the working set of pages as $W(t, w)$ - the set of pages referenced by a process during the time interval $t - w$ to t. The working set can change over time for a given process. This can be used with the WS clock algorithm, by tracking the "time of last use". At each page fault, we do the following check on the page we're pointing at;

- if the referenced bit is set to 1, reset it to 0, and move to the next page
- if the referenced bit is not set, we calculate its age
  - if the age is less than the working set age, continue (as the page is in the WS)
  - if the age is older than the working set age, we replace it (writing back to disk if required)

We however need to model the size of the working set. This can be done by observing the page fault frequency - if we estimate incorrectly (too low), we will encounter many page faults, thus having a low interval between page faults. If w is too high, thus we are assuming the working set is larger than it is, we will have diminishing returns (the page fault frequency doesn't change, as the entire working set is now in pages). We can tune this dynamically.

A local page replacement strategy is when each process gets a fixed allocation of physical memory, and we need to pick up the changes in the working set size. On the other hand, a global strategy dynamically shares memory between runnable processes, and considers page fault frequency to tune allocation (gives more to a process by taking from another). Linux uses a global page replacement strategy, whereas Windows uses local.

Linux Page Replacement

Linux uses a variation of the clock algorithm (to approximate LRU). The memory manager uses two linked list (as well as reference bits) - the active list containing active pages, with the most recently

used pages near the head of the active list, as well as an inactive list, which has the least-recently used pages near the tail. Unused pages from the active list are moved to the head of the inactive list. It only replaces pages in the inactive list.

From a practical point of view, we cannot simply perform this eviction on a fault, as we don't tend to have the granularity of allocating memory in just pages. For example, if the memory was full, and we performed a `malloc` for a large amount of memory, we don't want to do many evictions as that would be slow. Instead, this is done asynchronously by the following (when the processor is idle);

- `kswapd` (swap daemon)

  reclaims pages in the inactive list when memory is low to a dedicated swap partition or file, and handles locked and shared pages

- `pdflush` (kernel thread)

  periodically flushes dirty pages to disk (allows for easier eviction) - done when there is low I/O load

Tutorial Question

Suppose that pages in a virtual address space are referenced in the following order;

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

There are 3 empty frames available. Assume that paging decisions are made on demand, i.e. when page faults occur. Show the contents of the frames after each memory reference (and how many page faults occur in each case), assuming

(a) the LRU replacement policy

(b) the clock policy