COMP3231: Operating Systems

haezera

Contents

I. Operating Systems Overview

The two roles of the operating system

Role 1: An abstraction of hardware

- Hardware is difficult to work with
- The `OS` provides high-level functionality (think `write`, etc.)

Role 2: A resource manager

- The `OS` is responsible for allocating resources to users and processes
- The resource manager must ensure:
  1) No starvation
  2) Progress
  3) Allocation w.r.t a policy
  4) 'Efficiency'

User mode versus privilieged mode

In (most) operating systems, there exists two distinct modes.

1) Privilieged mode
   - This is the mode that the `kernel` runs in
   - Has elevated access to hardware (such as the ability to write to disk)
   - Can only be accessed by the operating system
   - Exists for security and for a single source of truth
   - Can deal with faults and errors from user applications
2) User mode
   - Where user applications live
   - Have to access elevated operations through the operating system

Why do we need privilege?

- The operating system as the `central` resoucre manager, controls access to critical operations (such as `write/read` from devices
- Since it is the single source of truth for prvilieged actions, this ensures consistent and secure alloaction of resources to applications
- Furthermore, security can be enforced in terms of prohibited behaviour that would *affect other applications*

When we refer to a `device` in operating systems, we refer to

A physical *or virtual* hardware component (such as a keyboard, printer, network interface) that interacts with

the computer system.

Privilege-less OS
There however does exist OS with no privilege. What consequences does this have?

- Any fault in any "user" application will crash the entire system
- Access to devices is unprotected - malicious things could be done to things like disks
- Access to other applications' data

For now, we can think of syscalls as the abstraction that applications use to access the OS' functions.

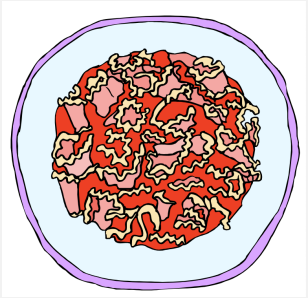
System libraries, syscalls and library functions
System libraries are utilised by applications to help with their application's end goal. There are three types of library functions w.r.t OS access

1) Pure: doesn't access privilieged mode at all
2) Hybrid: has both user-mode and privilieged-mode requirements
3) Syscalls: has *only* privilieged-mode requirements

The 'layers' of an operating system
We've briefly discussed some operations that an operating system completes. We could then be *motivated* to create some layered structure.

1) Processor allocation and multiprogramming (`compute management`)
2) Memory management
3) Devices (`device access`)
4) File systems (`partitioning of data`)
5) User applications

In theory, as we go down this list, they should depend on the previous parts, but in reality...



The OS has many interleaving parts that break this abstraction.

II. Theory of processes, threads and related models

A. What are processes and threads?

In the previous section, we often talked about how the OS was a *resource manager* - but to what is it a manager to?

To put it simply - `processes`

But what are processes? Consequentially, what are threads?

What is a thread?
A thread is a `unit of execution`. Threads execute some portion of instructions as instructed to. Threads contain

1) Program counter
   - Keeps track of which instruction the thread should execute
2) Registers
   - Threads require some memory to execute their instructions
3) Stack
   - Threads get their own *local variables*
4) State

What is a process?
A process is a `unit of work`. Processes manage the information required to handle threads. Processes (in our model) do not run anything themselves. Rather

Processes contain the necessary information to create and execute threads

Processes contain

1) Threads
   - The units of execution for the process' task
2) Process state
   - Ready, blocked or running
3) Address space
   - What regions of memory can threads use and allocate to?
4) Global variables
   - Shared variables between all threads
5) Open files
   - A list of open files
6) Child processes
   - Processes that inherit variables and other process items, but is a distinct process
7) Pending alarms
   - Timer-based signals from the operating system

The four models of threads and processes

1) Single process, single thread
2) Single process, multiple threads
3) Multiple processes, single thread
4) Multiple processes, multiple threads

When and why are processes created?

1) System initialisation
   - There are processes required to run an op-

erating system
  - Think about your window manager, background processes that deal with resources
2) Process creationg `syscall`
3) User request to create a new process
4) Initiation of a batch job

Batch job
A batch job is a non-interactive, pre-scheduled unit of work that runs without user intervention; typically in the background.

What are the types of process terminations?
- Normal exit (voluntary)
  - Like at the end of the process' execution
- Error exit (voluntary)
  - An error occured during the unit of work - exit cleanly
- Fatal error (voluntary)
  - A severe runtime error that is unrecoverable, and is terminated by the `OS`
- Killed by another process
  - For example, if a process is in deadlock or the system is out of resources

B. How are processes and threads managed in an OS?

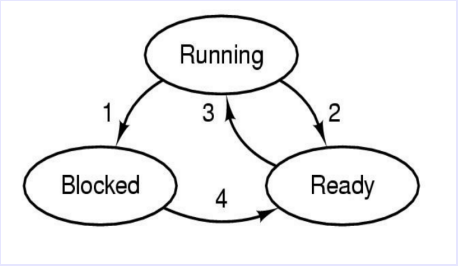
Process Control Block (PCB)
The `process control block` is a data structure that contains all of the information the OS needs to manage and execute a process (as outlined before).

At the creation of a process, a `PCB` is allocated with initialisation, and is 'scheduled'.

How are *all of* the processes stored in an OS?
Processes require multiple data structures to be 'stored'. There exists a `process table` which stores all of the processes in the OS.

There also exists the `ready queue` and `blocked queue(s)` which relate to thread and process states.

Thread and process states
Processes, and therefore their threads have three states:
1) Running
2) Blocked
3) Ready



Consider the following actions for the above diagram
1) The running process blocks for input (`I/O`)
2) Scheduler now needs to schedule a ready process
3) Scheduler picks a process that is ready
4) Input event finishes, and now blocked process is ready

The scheduler
The scheduler is an OS component which is responsible for **deciding which process or thread that runs next** on a `CPU`.

The ready queue
The ready queue is a queue of processes that are available to run on a CPU. The scheduler chooses from the ready queue w.r.t a pre-determined scheduling policy (for example, `FCFS`)

The blocked queue
When an unblocking event occurs, how do we admit this process back into the ready queue?
- Using a single blocked queue caused `head-of-line` blocking
- This is due to the fact that not all blocks are caused by the same cause

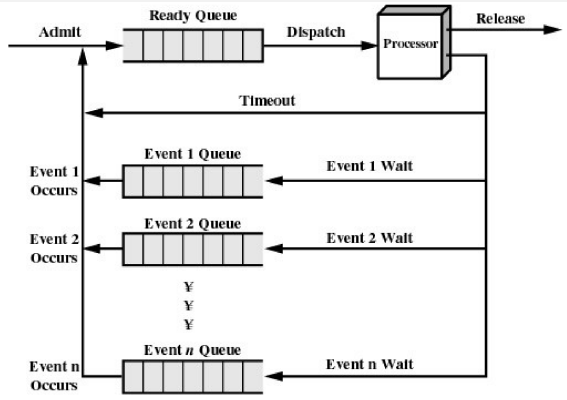
Therefore, having a queue for each blocking event is the optimal (and implemented) solution

C. Examples of process and thread model

The hamburger restaurant - why ready/blocked queues are important
Imagine a hamburger restaurant - we need to grill burgers, fry the fries, prepare the burgers, clean the restaurant and much more.

We can imagine that we *shouldn't wait for the burgers to grill*, when we could be frying the fries during this time. This is why blocking and these queues are so important.

A diagram of a ready queue and blocked queue interaction



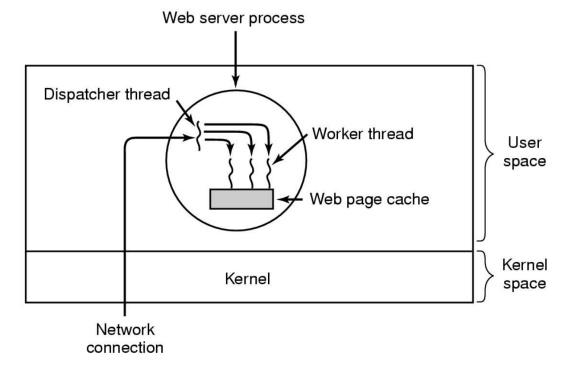
Global and local variables: heap or stack?

- Local variables are per-thread and allocated on the stack
- Global variables are shared across threads and alloacted in the data segment
- Dynamic memory can be both global or local (depending on the pointer scope)

Why might multiple threads be useful? A web server example

Consider a web server that runs a chat function for `Messenger`. Each time a user enters a chat, there must be some connection established.

- Imagine if we had one thread dealing with millions of different connections
- Users would receive messages from other users in wildly different times
- Rather, a per-user thread model created by a `dispatcher` thread would be much more efficient



Thread model versus finite state machine

We generally think about units of execution in terms of threads - where multiple threads can concurrently run independently and in parallel.

A *alternative* to this is the finite state machine (`FSM`) model. In the `FSM` model, systems are always in one state at a time, and transition to other states based on input. Instead of blocking, `FSM`s tend to loop whilst they are waiting for an appropiate input.

So why the thread model over finite state machines?

- Simpler to program than a state machine - we have functions available to us from modern OSes for the implementation of processes and threads
- Less resources associated, and importantly resources can be shared
- Threads can easily take advantage of the parallelism available on machines with more than one CPU
  - `FSM`s run in single-threaded event loops, not ideal to take advantage of multiple processors.

III. Concurrency and sychronisation

A. What is concurrency and it's implications?

With modern day compute, operating systems must be able to efficiently utilise multiple cores and parallelism capabilities.

Concurrency is when multiple tasks make progress during the same time period

A race condition

A race condition is when the behaviour or outcome of a program depends on the timing or ordering of events, particularly between threads.

A simple race condition example

Consider the following code

```
void increment() {
    int t;
    t = count;
    t += 1;
    count = t;
}

void decrement() {
    int t;
    t = count;
    t -= 1;
    count = t;
}
```

If two threads entered decrement and executed the instructions at the exact same time, then if the original count was `1000` - both threads would result with `999`.

But what if I use single-threaded processes!

While the user-level applications may be single-threaded, the kernel can interrupt at different points of the execution.
Therefore, even in single threaded processes, processes that share state may still have issues with concurrency.

Critical regions and their requirements
A critical region is a portion where a shared resource is accessed or modified.

The four formal requirements for a critical region are

1) Mutual exclusion: at most one thread is inside the critical region at a time
2) Progress: if not thread is in the critical region, and some wish to enter, one will succeed
3) Bounded waiting: no thread should wait forever to enter the region
4) Atomicity: the code inside a critical region appears to run atomically

B. Avoidance and detection of critical regions

Avoiding critical regions and mutual exclusion
Mutual exclusion is a property of a system that ensures no two processes or threads are in their critical sections at the same time.

Identifying critical regions
We defined before that critical regions are when a shared resource is accessed and/or modified. Consider the below code snippet

```
1  struct node {
2      int data;
3      struct node *next;
4  }
5  struct node *head;
6
7  void init(void) { head = NULL; }
8
9  void insert(struct item*) {
10     item->next = head;
11     head = item;
12 }
13
14 struct node *remove(void) {
15     struct node *t;
16     t = head;
17     if (t != NULL) {
18         head = head->next;
19     }
20     return t;
21 }
```

All we have to do is consider where `head` is accessed or modified. This is

- Lines `7`
- Lines `10-11`
- Lines `16-18`

Mutual exclusion option 1: test and set instruction
With critical regions, we have a requirement of mutual exclusion - but how do we do this? Some naive solutions include:

- Using a standard variable to signal entrance (creates another critical region)
- Taking turns (busy-waiting wastes resources)
- Disabling interrupts (wastes resources, only works in `kernel`)

So in reality, we need some help from **hardware**. The test and set instruction (`TSL`):

- Checks a value at some register `$lock` and if 0, sets to 1.
- *Guarantees these two operations happen atomically*

C. Techniques for mutual exclusions and their extensions

Mutual exclusion assembly example with test-and-set
We can then utilise `TSL` to ensure mutual exclusion

```
1  critical_region:
2      # some critical region code
3
4  enter_region:
5      tsl $register, $lock
6      jnz enter_region, $register
7      jal critical_region
8
9  leave_region:
10     li  $lock, 0
11     ret
```

Here's what the code is doing:

- Line 5 loads the value of the lock into the register and tests and sets it
- Line 6 loops if the lock was already taking (`$lock != 0`), otherwise returns to caller
- Line 7 otherwise enters the critical region
- Line 10 unlocks the lock if we have entered the region

But we're still busy-waiting!
That's true! Instead of busy waiting, we could `sleep` threads that attempt to enter the region but fail.

It is then up to the thread that is *in the critical region* to wake up the next process... but which process? We will answer this question in a later mutual exclusion option.

The producer-consumer problem
The problem involves a *producer* that creates data and a *consumer* that consumes data in a finite buffer.

- Consider that the buffer is finite - and so:
  - the producer shouldn't produce when the

buffer is full
  - the consumer shouldn't consume when the buffer is empty
- So how do we keep an accurate count?

We would eventually get to some stage where we lock the portion which inserts/removes items from the buffer and increments/decrements the count.

We are motivated to sleep when a consumer/producer can't do anything - but

```
1  # for the producer
2  prod() {
3      if (count == N)
4          sleep(producer)
5
6      # some code regarding the buffer
7
8      if (count == 1)
9          wakeup(con)
10 }
11
12 # for the consumer
13 con() {
14     if (count == 0)
15         sleep(con);
16
17     # some code regarding the buffer
18
19     if (count == N - 1)
20         wakeup(prod);
21 }
```

We can imagine with parallel execution that we could wake up the consumer before it even sleeps, which is clearly an issue. This motivates our next data structure

Mutual exclusion option 2: resources and semaphores

Test and set instructions let us impose mutual exclusion on a region of code - but the producer-consumer problem showed us that with some semblance of *limited resources*, we need more.

`Semaphores` are a primitive that include two main instructions:

- Wait (`P`): try to access the resources
- Signal (`V`): release a resource

When there are multiple waiting processes, we represent this with a linked list.

```
1  struct sempahore {
2      int count;
3      struct process *L;
4  }
5
6  P(semaphore S, process P) {
7      while (S.count <= 0) {
8          append P to S.L
9          sleep(P)
10     }
11     S.count--;
12 }
13
14 V(sempahore S){
15     S.count++;
16     if (S.count <= 1) {
17         fetch P from S.L
18         wakeup(P)
19     }
20 }
```

We can imagine that a *critical region*, is then simply a quasi-(producer consumer) problem with 1 resource. The buffer is the region of code itself, and threads can take and reliquinish that region of code.

Solving the producer-consumer problem with semaphores

```
1  #define N = 10;
2  semaphore mutex = 1;
3  sempahore buffer_empty_space = N;
4  sempahore buffer_filled_space = 0;
5
6  prod() {
7      while (TRUE) {
8          item = produce();
9          wait(buffer_empty_space);
10         wait(mutex);
11         insert_item();
12         signal(mutex);
13         signal(buffer_filled_space);
14     }
15 }
16
17 con() {
18     while (TRUE) {
19         wait(buffer_filled_space);
20         wait(mutex);
21         cons = insert_item();
22         signal(mutex);
23         signal(buffer_empty_space);
24     }
25 }
```

We can see that

- Line 9 indicates that we will be taking some empty space from the buffer
- Line 13 signals to a waiting process that we have filled some space
- Line 19 indicates that we will be taking something from the buffer

- Line 23 indicates that we have taken something, and there is more empty space

So, we've evolved test and set instructions → sempahores, making a higher level abstraction of 'resources' to solve our producer-consumer problem. But sempahores themselves are primitive (in the literal sense too) - they provide two functions. Is there an easier

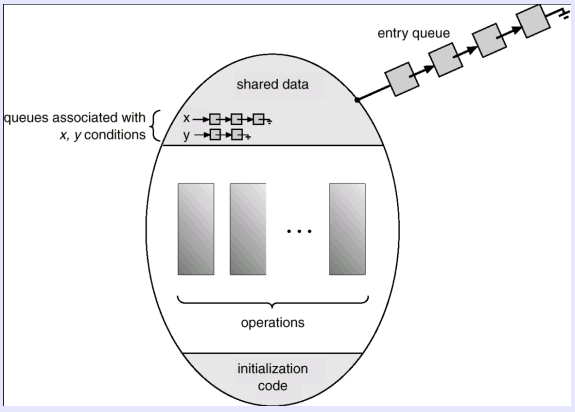
Mutual exclusion option 3: easing concurrency and monitors
Monitors are a higher level programming construct which involve *packaging* procedures, variables and data types into a special kind of module.

Importantly, only one process/thread can be in the monitor at one time, which then creates a queue-like behaviour.

Within the monitor, different threads may require other threads to finish to continue. In monitors, we use `Condition Variables` to solve this problem.

Condition variables To allow a process to wait within the monitor, condition variables can be used.
For some condition `condition x`, we can

- `x.wait()`: suspended until a signal
- `x.signal()`: resumes one suspended process.

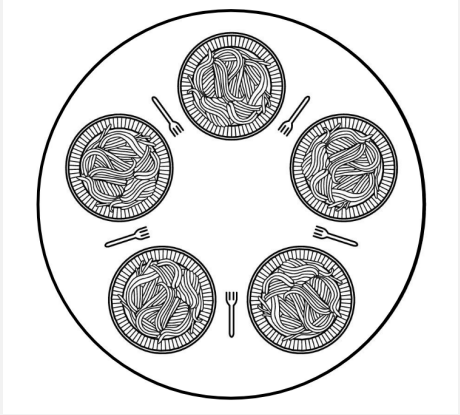


Key takeaway: monitors allow for you to enter a critical region/producer-consumer problem without the worry of manually signalling semaphores or mutexes.

Locks, sempahores and condition variables: why, and where?
We've introduced these three primitives, but what purposes to they serve?

- Locks: waiting for a shared resource
- Semaphores: to solve the producer/consumer problem
- Condition variables: waiting for an event

The dining philosophers problem
To conclude our chapter in concurrency and synchronisation, we look to the dining philosophers problem. Five philosophers can either eat or think about eating.



The philosophers can only pick up one fork at a time. How do we solve this problem?

```
1  monitor DiningPhilosophers {
2  condition can_eat[5];
3  state[5] = THINKING;
4
5  procedure pickup(i) {
6      state[i] = HUNGRY;
7      test(i);
8      if (state[i] != EATING)
9          wait(can_eat[i]);
10 }
11 procedure putdown(i) {
12     state[i] = THINKING;
13     test((i+4)%5);   # left
14     test((i+1)%5);   # right
15 }
16 procedure test(i) {
17     if (state[i] == HUNGRY &&
18         state[(i+4)%5] != EATING &&
19         state[(i+1)%5] != EATING) {
20         state[i] = EATING;
21         signal(can_eat[i]);
22     }
23 }
24 }
```

Heuristically identifying deadlocks
To maintain safe concurrent interactions, there must be a *global locking order*. That is, if some process A locks in the order $L_1 \rightarrow L_2 \rightarrow L_3$, then every other process which also utilises these locks must lock in this order.

To find deadlock conditions, we consider the lock order of each process, and find any inversions of the global locking order.

IV. MIPS R3000, and practical implementation of processes and threads

I assume you remember most of the instructions for MIPS from COMP1521. I mention some interesting implementation details about MIPS R3000, and as well include register conventions.

A. MIPS details

MIPS pipelining and the branch delay slot

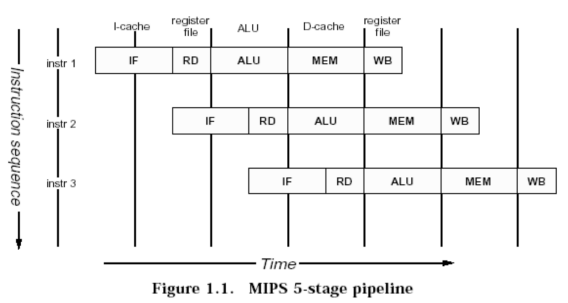


Figure 1.1. MIPS 5-stage pipeline

MIPS R3000 pipelines it's instructions for efficiency. We can see in the above diagram that there are three instructions being pipelined.

The `branch decision` is not known until the end of the `ALU` phase - so if instruction 2 relies on instruction 1, then it must wait until the end of instruction 1's `ALU` phase.

This is why there exists a branch delay for MIPS R300 jump instructions. For example

```
1 jal 1f
2 nop      # nothing occurs here
3 lw r4, (r6)
```

After the jump on line 1, the jump only takes place on the third line *before* it is executed. Once it returns, it will continue on line 3.

The program counter (PC)

The program counter is used to keep track of which instruction to run next. In `MIPS R3000`, instructions are `32-bits`, and thus `4` bytes.

Thus, after a (non-jump) instruction, the PC increments 4 bytes

```
PC += PC + 4
```

For a jump instruction, the program counter jumps the PC to the location, and then jumps back and then starts at 8 bytes forward of the jump (due to the branch delay).

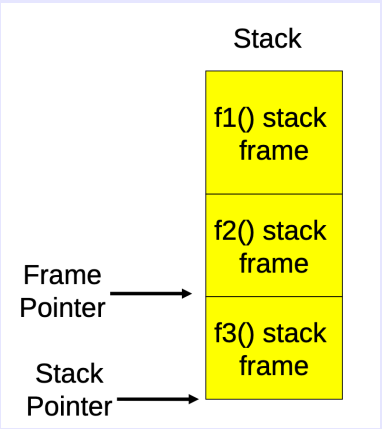
Compiler register conventions

Reg No	Name	Used for
0	zero	Always returns 0
1	at	(assembler temporary) Reserved for use by assembler
2-3	v0-v1	Value (except FP) returned by subroutine
4-7	a0-a3	(arguments) First four parameters for a subroutine
8-15	t0-t7	(temporaries) subroutines may use without saving
24-25	t8-t9	
16-23	s0-s7	Subroutine "register variables"; a subroutine which will write one of these must save the old value and restore it before it exits, so the *calling* routine sees their values preserved.
26-27	k0-k1	Reserved for use by interrupt/trap handler - may change under your feet
28	gp	global pointer - some runtime systems maintain this to give easy access to (some) "static" or "extern" variables.
29	sp	stack pointer
30	s8/fp	9th register variable. Subroutines which need one can use this as a "frame pointer".
31	ra	Return address for subroutine

Per-thread stack components: stack frames

Previously, we mentioned how threads have their own per-thread stack. *Similarly*, functions have their own per-function `stack frame`, which stores:

- The return address for the function
- Saved registers
- Passed function parameters
- Local varaibles



The stack grows downwards, and winds up as functions return (this is not strictly true and can be seen in the reverse way).

So how do we know where to go back to after a jump?

Using stack frames! The stack frame stores the appropiate return address (program counter) for the function. When we do

```
0x00: jal function_one
```

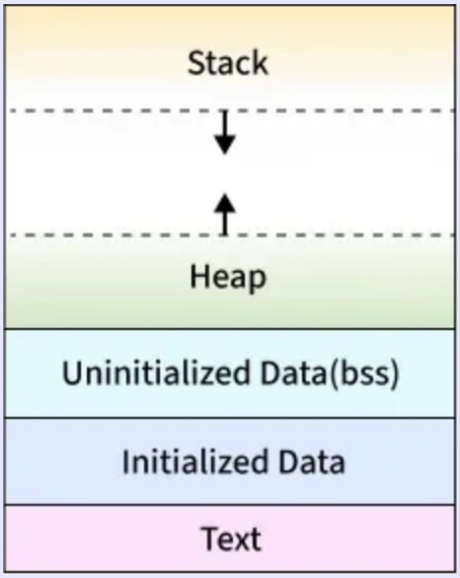
We are creating a new stack frame for the function `function_one`, with information including that the return address is `0x00 + 4`.

Okay, so we now know how per-thread stack space's are divvied up (using stack frames). What about processes?

Process memory layout
We consider a single-threaded process below.

Remember that the process contains everything a unit of work requires to run. At the minimum, the contains three segments:

1) Text
   - Contains the code (instructions) to run
2) Data
   - Global variables
3) Stack
   - Activation records of procedures/function/method
   - Local variables



Processes exist in both user and kernel mode, so how do they differ?

Processes in user and kernel mode
User processes
- User processes are scheduled by the kernel
- They are isolated from eachother
- The OS ensures no concurrency issues exist

Kernel processes
- Mostly the same as user processes, but...
- Kernel memory is shared between all processes
- Concurrency issues can exist with system calls

Now let's talk about threads - should they be instantiated at the user or kernel level?

User-level versus kernel-level threads
User-level threads
- Processes create threads without the knowledge of the OS, so...
- Have to deal with concurrency issues, `yield` within process
- Can create many more threads
- Does *not* take advantage of multiple CPUs
- If a thread blocks, the entire process blocks (wasting `I/O` time!)

Kernel-level threads
- Preemption - threads cannot hog CPU indefinitely
- True parallelism, overlap blocking `I/O` with CPU-bound
- But creation/destruction requires kernel entry/exit, which is expensive

Cooperative versus preemptive multithreading
User-level threads are not within the kernels control, and thus are not subject to the scheduler. This means that any form of multi-threading must be *cooperative* - that is, must be voluntary `yielded`.

Kernel-level threads are controlled by the kernel, and are thus preemptively multithreaded; they can be paused mid execution by the scheduler.

So both user and kernel level threads have their own advantages and disadvantages. For these notes, we generally assume kernel-level threads.

We have talked about the idea of *switching* between processes, as well as threads within processes. What's happening here?

B. Context switching: implementation and implications in the kernel

Context switch
A context switch can refer to both:
- a switch between threads
- a switch between processes

It is easy to wonder how we can switch back and forth, consider the per-thread/per-process memory we are required to have.

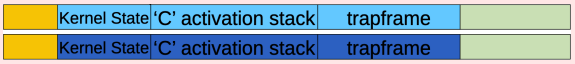
A switch can occur any time we enter the kernel, so
1) on a `syscall`
2) on an exception
3) on an interrupt

Trapframe
A trapframe is a snapshot in time of the CPU's state. It is used in context switches to save the state of execution for a thread/process.

They are often represented in `structs` with the registers as variables.

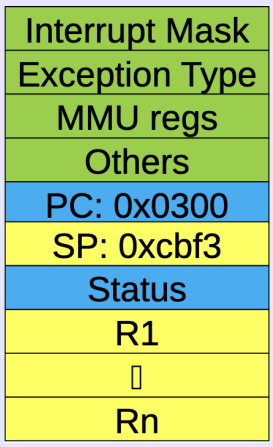
Thread switching: what occurs?
Let us walk through what occurs during a thread switch, from thread `T1 -> T2`.

1) We begin with the stack pointer pointing to T1's stack `$sp$`
2) After an exception/syscall/interrupt, we switch to the kernel stack
3) We push T1's trapframe onto the kernel stack.
4) We call some `C` code that processes the exception/syscall/interrupt
5) The kernel decides to context siwtch - we then push the kernel state onto the stack
6) We switch to another process, which then unwinds down from the above process



What's in the kernel state?
In the kernel state, a few important (and some still unlearned data structures) exist

- Page tables (for virtual memory mapping)
- Kernel stack pointer (to know where to unwind from)
- Scheduling information
- Locks, resourecs, etc.



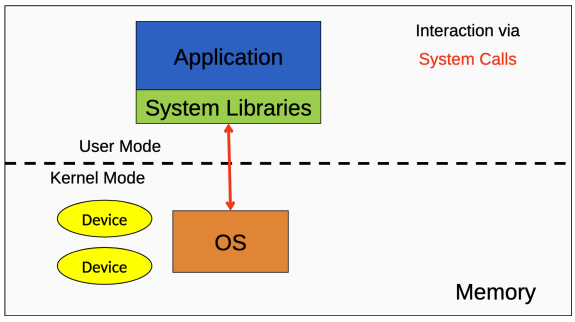
Some of the above registers are important for understanding system calls.

- Interrupt mask: a bitmask that determines which interrupts are enabled/disabled.
- Exception type: indicates what caused the trap (kernel entry), for e.g system call, page fault, etc
- MMU (Memory Management Unit) regs: stores info about the process' memory mapping
- PC: Program Counter
- SP: Stack Pointer
- Status: Contains information about condition codes, privilege codes and interrupt enable bits

Privilieged portions of memory
There exists memory addresses that are only accessible to the kernel. The exact ranges are usually configurable.

- This is done to protect kernel code and data

V. System calls



We have previously referred to `syscalls`, as ways to enter the kernel (the OS) and access privilieged operations. In this section, we consider what actually occurs during a syscall.

Some important registers for syscalls

Before we begin explaining how system calls are entered, processed and then exited, we must first consider some software/hardware details regarding MIPS R3000.

Coprocessor 0 (CP0)
The processor control registers are located in CP0. These registers control

- Exception/interrupt management registers
- Translation management registers

Specific instructions are used to manipulate CP0, and are *only accessible in kernel mode*. There are 3 important registers we would like to focus on:

- `c0_cause`: cause of the recent exception
- `c0_status`: current status of the CPU
- `c0_epc`: address of the instruction that caused the exception

What information do these registers contain?

c0_status

- `IM`: 8 bits of interrupt mask bits. 6 external, 2 software.
- `KU`: user or kernel mode? `0` is kernel, `1` is user
- `IE`: `0` is all interrupts masked, `1` is interrupts enabled
- `c, p, o`: current, previous, old which exists for `KU` and `IE`

c0_cause

- `ExcCode`: the code number of the exception taken

Each exception code has a mapping to a specfiic reasoning. The two important ones to know are `0` for interrupt and `8` for syscall.

c0_epc This register points to the address of where to *restart* execution after handling the exception.

Exception vectors Exception vectors are fixed memory addresses which the processor jumps to for specific exceptions.

This means there's no need for dispatching - thus no need for `c0_cause`. The exception code used for general exceptions is `0x80000080`.

Full walkthrough of an hardware exception handler

We begin with

```
PC: 0x12345678  | EPC: ?
Cause: ?        | Status: ?|?|?|?|1|1
```

where status is `KUo,IEo,KU,IEp,KUc,IEc`. An interrupt exception occurs at the current program counter.

```
PC: 0x12345678  | EPC: 0x12345678
Cause: 0        | Status: ?|?|1|1|0|0
```

Note the cause `0` which corresponds to interrupt. Also note that the current `KU, IE` have been shifted downwards to the previous flags.

```
PC: 0x80000080  | EPC: 0x12345678
Cause: 0        | Status: ?|?|1|1|0|0
```

We now place the program counter to the general exception vector placed in PC. Now we know what caused the exception, and where to restart once we deal with it. Let us assume the exception is dealt with, and has the following variable states

```
PC: 0x80001234  | EPC: 0x12345678
Cause: 0        | Status: ?|?|1|1|0|0
```

Now consider the following assembly code to return to program

```
1 lw r27, saved_epc
2 nop
3 jr r27
4 rfe
```

Line 1 stores the address where the exception occured back into the PC. We then do a `rfe` instruction, which is a `return from exception` instruction.

```
PC: 0x12345678  | EPC: 0x12345678
Cause: 0        | Status: ?|?|?|?|1|1
```

And we are now back at the user-level, executing our program from where an interrupt was raised.

MIPS and OS/161 Syscall Conventions

MIPS

- Syscalls are invoked by the `syscall` instruction
- Syscalls are represented by numbers, and must be agreed on by convention with the caller.
- A convention is required as to how user-level software indicates
  - Which system call is required
  - Where it's arguments are
  - Where the result should go

OS/161

- Arguments are passed and returned via the normal C function calling convention, i.e `def read_syscall(int fd)`
- Syscalls are defined by a syscall **number** - agreed on convention
- Register `v0` contains the syscall number
- On return, `a3` contains:
  - 0 for success, and `v0` contains the result
  - not 0 for failure, and `v0` has the error number

What occurs on the user side of the syscall?

- The user side is pretty straight forward
- Call the desired `syscall` with the correct arguments

What occurs on the kernel side of the syscall?

- Change to kernel stack
- Preserve registers by saving to kernel stack
- Leave saved registers somewhere accessible
- Complete the `syscall`
- Restore registers
- Switch back to the user stack
- Return to application

What occurs in a computer during an exception that causes a context switch?
Consider an interrupt on a thread causes a thread switch. Detail what occurs throughout the user and kernel level.

1) CPU: some interrupt is raised, and an exception is generated
2) CPU: changes to kernel mode and calls the exception handler at `0x8000080`
3) Exception handler: switches to the kernel stack pointer
4) Exception handler: saves the user registers for the interrupt process to the stack
5) Exception handler: determines the source of the interrupt - let's say it was a `timer` interrupt, and switches to the timer interrupt handler.
6) Timer interrupt handler: acknowledges the interrupt, and calls the scheduler
7) Scheduler: chooses a new process to run, and then invokes the kernel to switch
8) Kernel: saves the current stacks kernel context to the kernel stack
9) Kernel: switches to the new thread's stack by moving `$sp`
10) Kernel: reads the new process in-kernel context from the stack
11) Kernel: restores the user registers for the new process
12) Kernel: set the processor back to user mode
13) Kernel: jump to new user process' PC

Note where different portions of the original thread state is saved (user + kernel state)

MIPS R3000 does not have specific exception handlers!
Besides for specifc exceptions (e.g `TLB` misses in a specific memory region), MIPS forwards all exceptions to the general exception vector `0x80000080`.

From there, different specific handlers/routines can be called, after the handler figures out the specific exception that occured.

VI. Deadlocks and Livelocks

We have talked to a decent extent about `resources` and how to avoid problems with shared resources between multiple threads/processes. In this seciton, we talk about the problem of deadlocks and livelocks - where progress cannot be made due to a resource.

What is a resource?
Formally, a resource is a physical *or logical/digital* entity required for process execution, that is controlled by the `OS`.

What is a deadlock?
A deadlock is a state of a system in whcih a finite set of processes are waiting for an event that can only be caused by another process in the same set.

- Suppose process P_1 holds resource A and requests resource B
- Suppose process P_2 holds resource B and requests resource A

Clearly there is no progress that can be made here.

This should motivate some formal requirement regarding the *ordering* of how resources are taken.

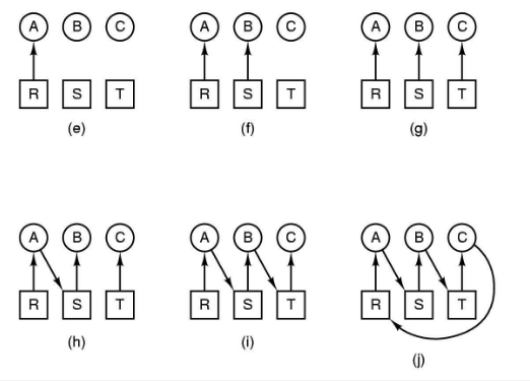
The four conditions for a deadlock

1) Mutual exclusion
   - Each resource is either assigned to a process or is available
2) Hold and wait condition
   - A process which already has resources can request more resources
3) No preemption condition
   - Resources granted to other processes cannot be forcibly taken away
4) Circular wait condition
   - Must be a circular chain of requests of 2 or more processes

We can very quickly, heuristically argue why these conditions are required

1) If a resource was infinite, then no deadlocks could possibly occur
2) If a process could only hold one resource, then it would never depend on another resource held by another process
3) If a process could be preempted, then a deadlocked process could just take it
4) There is a requirement for $A->B$ and $B->A$ for no progress to be made (consider a circular path)

Representing deadlocks
Deadlocks are often represented by directed graphs, with the tail at the resource and the head at the process.



Here $\{R, S, T\}$ are resources and $\{A, B, C\}$ are resources.

The four broad approaches for dealing with deadlocks is

1) Just ignore the problem
2) Prevent the problem (negate one of the conditions)
3) Detection and recovery
4) Dynamic avoidance (careful resource allocation)

Approach 1: The Ostrich Algorithm
The first approach is to do nothing - this is reasonable only if

1) deadlocks occur very rarely
2) cost of prevention is high

This is a tradeoff between `convenience` and `correctness`.

Approach 2: Deadlock Prevention
In deadlock prevention, we aim to negate one of the four conditions of a deadlock. Consider how we could do this for each condition:

Mutual exclusion

- Not feasible - requires finite resources to become infinite
- If we removed locks, then concurrency issues

Hold and wait

- Requires processes to request all resources before starting
- This is of course difficult to know at runtime
- We *could* make a process give up all resources if it is required to block holding a resource, but this could cause `livelock` (explained after).

No preemption

- Not feasible to take away a resource from a process mid execution
- Hard to identify what resources are crucial to a process, etc.

Circular wait

- We could attack this by carefully ordering resources

What is a live lock?
Livelocked processes are not blocked, change state regularly but never make progress.

Imagine two resources constantly releasing and re-requesting a lock, but ordering makes it so that neither of the two processes can enter the critical region.

Approach 3: Detection and Recovery
Assuming a system is deadlocked, we require a method to detect this and restore progress.

Consider the following graph model. We have processes $P = \{P_1, \dots, P_n\}$ and resources $R = \{R_1, \dots, R_k\}$. The `OS` then holds a resource allocation graph (RAG)of

$$P_i \to R_j : \text{ request edge} \tag{1}$$
$$R_j \to P_i : \text{ assignment edge} \tag{2}$$

If there exists any cycles, then there is a deadlock. For multiple-instance resources, this doesn't necessarily mean there is a deadlock. We need

- `avail`: a $k \times 1$ vector of free instances of resources R_j
- `alloc`: a $n \times k$ matrix where `alloc[i][j]` is instances of R_j held by P_i
- `req`: a $n \times k$ matrix where `req[i][j]` is instances of R_j that P_i still needs to finish

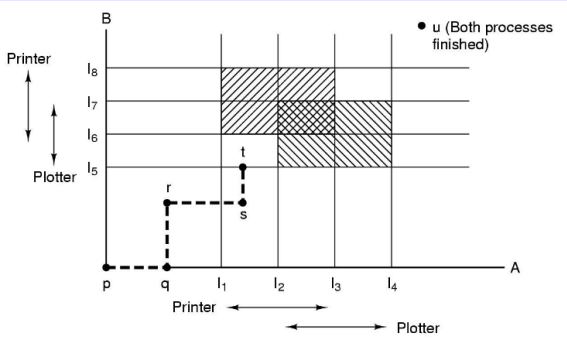
We therefore have the algorithm

```
1  finish = false for n x 1
2
3  repeat
4      found = false
5      for i from 1 to n
6          if !finish && req <= work:
7              avail[i] = avail[i]
8                       + alloc[i]
9              finish[i] = true
10             found = true
11         end if
12     end for
13 until found = false
14
15 deadlock = [
16     f for i in finish where i is false
17 ]
```

- Line 6 considers whether there is enough resources to fulfill the request
- If there is, line 7 completes that request, and then restores the freed resources
- If any process finishes without being found, then they are deadlocked.

But how do we recover?
We generally recover by simply killing one or more processes of the deadlocked transaction.

Approach 4: Deadlock Avoidance
Deadlock avoidance is possible, but we need to know *the maximum number of each resource required* by each process. There are two main ways we can represent this



Resource trajectories

- In the above figure, we see a resource trajectory graph of two process A and B
- Consider the regions $I_1 \to I_4$ for A and $I_6 \to I_8$ for B.
- Both require the printer, and so t has entered an unsafe state

	Has	Max
A	1	6
B	2	5
C	2	4
D	4	7

Free: 1

Banker's algorithm

- Instead consider we are given a table of resources, with processes A, B, C, D
- The `Has` column is the currently held resources, the `Max` is the max number of resources
- `Free` denotes the number of free resources available in the system
- In this case, we should not allocate any more resources, otherwise the system is in an unsafe state (zero resources)
- This is called the *Banker's Algorithm*

What is starvation?
In the first section of the course, we discovered that an operating system should ensure that starvation does not occur.

Starvation is when a process *never receives* the resource it is waiting for, despite the resource repeatedly becoming free.

- This may be due to an inappropiate or inefficient allocation algorithm

VII. File management and file systems

We discuss file systems and virtual file systems in the context of `UNIX` systems. Before this, we consider conventions about files and dircetories and what must be decided by the file system.

A. File system components

File names
- Textual names, which may have restrictions
  - Only certain characters
  - Limited length
  - Certain format
- Can be case sensitive
- Names may obey inventions (e.g `.c` for C files)

File structure (or the absence of)
- Files are generally just a sequence of bytes
- *Most* OS have unstructured files
- Structured files may have advantages in efficiency (the OS knows where things are before hand), but are hard to port to other OS.

File types
- Regular files, directories, device files, streams/pipes

File access types
1) Sequential access
   - Linearly scan the file from the `0`-th position
   - No jumping around (no `SEEK_SET`)
2) Random access
   - Can be read in any order
   - The read can be one of two options
     a) Move file pointer (`seek`) and then read, or
     b) Each read specifies the file pointer

Giving the OS context in unstructured files: Executable Link Format
Through a **Executable Linkable Format (ELF)**, the OS is informed of the location of important components. If the file is an executable, for example
- Provides information about the file; 32-bit vs 64-bit, endiannes, entry point address
- How parts of the file should be mapped into virtual memory
  - `text`, `data`, `stack`, `heap`
- Context for the linker

File directories
- Provides mapping between the file names and the file (data) themselves
- Contains informations about files

  - attributes, location, ownership and more
- Directories themselves are specially structured files owned by the OS

Tracking the current working directory + relative path names
Constantly using the absolute path name is tedious, and so the current working director (`cwd`) `./` is convenient. We can also do *relative* paths - for example `../` indicating the parent directory of the `cwd`.

Access rights and simultaneous access rights

- We might want some users to be able to read, but not write
- Might not want some users to be able to delete
- Access rights come with the following operations
  - `execution`, `reading`, `appending`
  - `updating`, `rights change`, `deletion`
- The `owner` as all rights

One might wonder how the `OS` deals with simultaneous access to files. Typically one of two approaches

1) User may lock entire when when updated, or
2) User may lock individual records/ranges of the file while updating

That is (most) all of the file semantics/properties we care about when we wish to implement a file system. We now consider the implementation of file systems in a UNIX-like system.

B. UNIX File Systems and Components

The UNIX Storage Stack: explained

Application
↓
FD table
↓
OF table
↓
Virtual file system
↓
File system(s)
↓
Buffer cache
↓
Disk scheduler
↓
Device driver
↓
Disk controller + disk

We will explain the stack bottom up

- The disk controller handles disk geometry and exposes the disk as a linear sequence of 'blocks'
- The device driver abstracts device specific (so disk, solid state, etc) protocol. Also exposes the linear sequence of blocks
- The file system abstracts the linear sequence of blocks into a directory hierarchy
- The buffer cache sits between the file system and disk scheduler for performance, keeping *recently accessed disk blocks*
- The virtual file system aggregates multiple file systems to be one interactable file system across multiple devices
- The open file table is a global table of all the open files on a system
- The file descriptor table is a per-process table of the files open in a process
- The application interacts with files through *file descriptors*.

Locality and speed: getting close
Many file systems are designed for spinning disks - the mechanism of a spinning disk works by reading disks as they spin.

Consider sequentially reading some file subdivided into blocks $F = \{F_1, F_2, ..., F_b\}$. We can imagine that as

$$||F_{i+1} - F_i|| \to \infty$$

where, reading the file becomes very slow (imagine a file evenly spread out across the entire revolution).

Blocks being close together (locality) is important for read speed

So how should blocks be allocated? If we use disks, should we just allocate them sequentially? We explore the following methods of 'block' allocation:

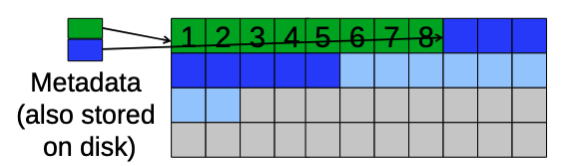
1) Contiguous allocation (sequential)
2) Dynamic allocation
   - Blocks as linked lists
   - File allocation table
   - Index nodes (`inode`)

What is a 'block'?
Blocks are `fixed` (but generally configurable) partitions of disk space. Data is split up into 'blocks' when written to disk.

What happens when you delete a block in a file system?
When you 'delete' a block, the block simply gets marked as writeable. The block itself does not get zeroed out, so it does not require an additional `I/O`

operation.

Contiguous allocation



Write the file to disk in sequential blocks.

- ✓ Easy bookkeeping (starting block + length)
- ✓ ↑ performance for sequential operations
- ✗ External fragmentation
- ✗ As files are deleted, free space becomes divided into small (mostly unusable) space

Fragmenting our disk: externally and internally

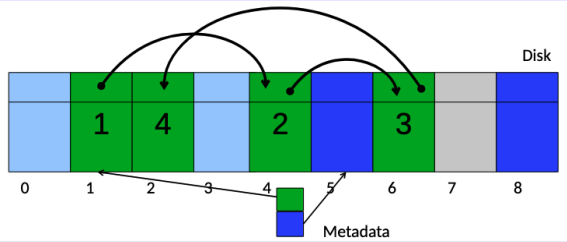
External fragmentation is

- space wasted externally to allocated memory regions
- memory may exist, but is not contiguous (and thus cannot be written for contiguous alloc)

Internal fragmentation is

- space wasted internally to allocated memory regions
- for example, block sizes are too large and are thus not utilised appropiately

Dynamic allocation: linked list allocation



We place blocks whether there is space, and represent the ordering of files in a linked list. The block number of the next block is stored in each block.

- ✓ One single metadata entry per file (starting block number, etc)
- ✓ Best for sequentially accessed files
- ✗ Bad for random access (we have to linearly scan!)
- ✗ Blocks are scattered across the disk

Dynamic allocation: File Allocation Table (FAT)

We keep a $1:1$ copy of the file system (linked list alloc) in a separate table in memory.

- ✓ Random access is faster, as in-memory scan is fast.
- ✗ Memory usage becomes big, fast.

`200 GB` is `200 * 10^6 * 10^3` bytes. With `1K` byte blocks, we have `200 * 10^6` blocks. This means there are `200 * 10^6` FAT entries in RAM, which means it takes up

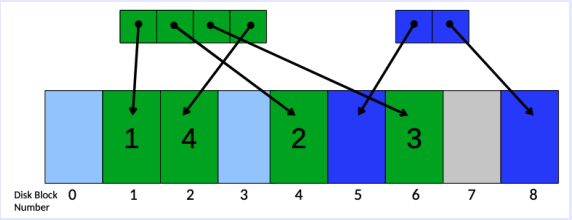
`800 MB`

which is substantial.

What happens if something corrupts or we lose the table in memory?

For `FAT` implementations, there exists redundancies stored on disk to be pulled into memory in case any corruption occurs.

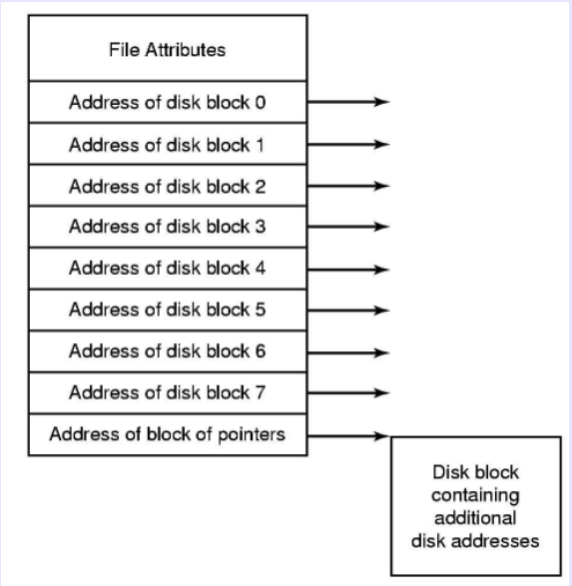
Dynamic allocation: inode-based



The key idea is to keep a separate table (called a `inode`) for each file.

- Only keep table for open files in memory (borrowing ideas from `FAT`)
- So we get fast random acess

`inodes` are allocated dynamically, and free-space management is required for inodes. Inode's keep pointers to addresses of disk blocks, and then an indirection pointer to another block of pointers.



Blocks + an indirection blocks

Comparing the three methods for different situations

Consider a file with 100 records. Compare how many `I/O` operations are required for contiguous allocation, linked allocation and indexed (direct pointer) allocation compares with the following 6 scenarios

1) The record is added at the beginning
2) The record is added in the middle
3) The record is added at the end
4) The record is removed from the beginning
5) The record is removed from the middle
6) The record is removed from the end

	Contiguous	Linked	Indexed
a.	100r/101w	0r/1w	0r/1w
b.	50r/51w	50r/2w	0r/1w
c.	0r/1w	100r/2w	0r/1w
d.	0r/0w	1r/0w	0r/0w
e.	49r/49w	50r/1w 51r/1w	0r/0w
f.	0r/0w	99r/1w	0r/0w

Note for indexed allocation, that deletes just requires an update in memory noting that the block is writeable/free.

For linked list, we need to write the new block but also update the previous blocks "next pointer", so we require 2 writes for some scenarios.

Implementing inodes: keeping track of free space

So how do we keep track of these blocks, and how do we manage free space? There are two approaches

1) Linked list of free blocks in free blocks on disk
2) Bitmaps of free blocks and free i-nodes on disk

Free block list One free block creates a linked list to information about other free blocks

- List of all unallocated blocks
- Store within the fre blocks themselves
- Only one block of pointers need to be kept in main memory

Bitmap Individual bits in a bit vector represent used/free blocks.

- One bit represents one block
- Expensive to search
- Easy to find contiguous free space (consecutive `0`s)

Implementation of directories

- Directories are stored like normal files, inside data blocks
- File systems assign special meanings to the content of these files
  - A directory file is a *list* of directory entries
  - A directory entry contains a file name, attributes and the file `inode` number

The `inode` number can then be used to access the file's contents.

So how do we choose what block size to use?

- Larger blocks require less meta data but incur more internal fragmentation
- Smaller blocks incur less internal fragmentation
- Larger blocks are better for sequential operations, less `I/O`
- But for random access, smaller blocks load less unrelated data

So block size is a compromise, and is dependent on the requirements of the machine.

So we know how file systems are implemented (generally for `disks`), but how do *virtual* file systems combine them to create a single unified abstraction for file usage?

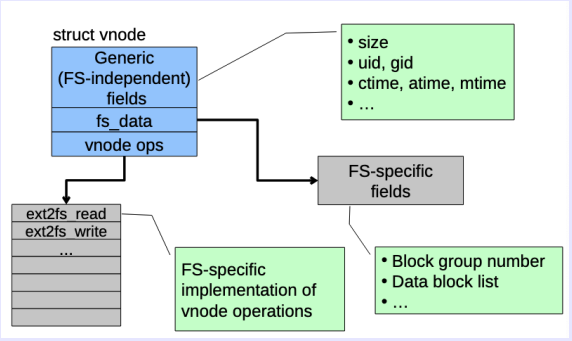
Providing a unified abstraction: vnodes/VFS

There are two major components to a virtual file system. The `vfs`, which represents all file systems:

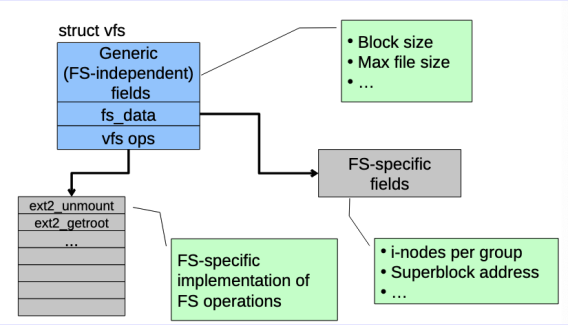
- Contain pointers to functions to do file-system-wide operations (e.g unmount, mount)

`vnodes`, represent a file (`inode`) in an underlying (abstracted) file system.

- Contains pointers to functions to manipulate files/inodes (e.g open, close, etc.)



vnode representation



vfs representation

So we abstract file systems with virtual file systems. Now how do applications interact with files?

File descriptor tables and file descriptors
File descriptors (`fd`) are indexes into file descriptor tables. File desriptor tables are (sometimes not literally) a table of pointers to vnodes, along with more information.

```
struct fd {
    off_t offset;
    int readable;
    int writeable;
    int executable;
    struct vnode *v;
}
```

Therefore when a file is opened in an application, a new file descriptor is created in it's per-process file descriptor table (which is also stored in the global open file table).

What file descriptor tables are necessary?
At the very minimum, there needs to be a *per-process* file descriptor

- When files are opened by two different processes, the reading/seeking/writing/etc. of one process shouldn't necessarily effect another file.

But some system calls like `fork` require two processes to share a file pointer. So we also need a *global open file table*

- This is used to point to one reference of a file descriptor from two different processes.

So now we know how applications deal with virtual file systems, how virtual file systems aggregate file systems and how file systems abstract a block sequence. How do we make these operations fast?

Buffers and caches
Buffer:

- Temporary storage used when transferring data between two entities
- Useful when entities transfer at different rates (e.g memory bus versus disk)

Cache:

- Fast storage used to temporarily hold data to speed up repeated access to data

How do file systems utilise buffers and caches?
Buffers

- Writes to disk can become very slow if we write directly to disk, as the speed of memory and the speed of disk is not in sync
- Rather, we write to a *buffer*, which is then heuristically flushed to disk
- We can also read blocks into the buffer ahead-of-schedule to increase efficiency

Caches

- We often access repeated segments of data
- When we read segments of data, we cache them in a finite, small memory set

Of course, buffers and caches are related (and often integrated). There is no point of reading a block into a buffer, and separately reading it into a cache.

But are buffer caches an all-around win? Write-through caches
No, not necessarily. While buffer caches are important to deal with memory/disk speed discrepancies and improve read performance, for portable storage devices and other scenarios, buffers present the risk for **data to be lost**.

Therefore, implementing a write-through cache which write immediately can be utilised for certain devices at the sake of performance.

Which cached blocks should survive? Which should we discard?

- Since cache space is finite, we need to make decisions on what to remove.
- Generally prioritised in terms of system criticality - directory blocks and inode blocks are crucial
- Data blocks will corrupt only the file that they are associated with
- So remove low-priority blocks first

C. Practical implementation of `inodes` and file system components

We now look to `ext2` and `ext3` filesystems to explore specific components of `inodes` and others to gain a better grasp of what is happening.

`ext2 file system`: indirection pointers
We previous discussed that `inodes` have direct block pointers, which hold direct disk block num-

bers.

- `ext2` has 12 direct blocks
- If there are more blocks then 12, what occurs?

`ext2` also has three indirect pointers; single, double and triple.

1) Single indirect points to a block which contains direct blocks (2 reads)
2) Double indirect points to a block with blocks which contains direct blocks (3 reads)
3) Triple indirect points to a block with blocks with blocks which contain direct blocks (4 reads)

How many blocks can each indirect hold?
Assume 1K-byte blocks, with 4-byte block numbers.

- The single indirect contains 256 blocks.
- The double indirect contains 256^2 blocks.
- The triple indirect contains 256^3 blocks.

So in total in `ext2`, we have a total of $256+256^2+256^3 = 16843020$ blocks, which is approximately 16 GB

Where is the block number in the tree?
Assume 4K blocks, 4 byte block numbers and 12 direct blocks. Consider the following code excerpt

```
lseek(fd, 1048576, SEEK_SET)
write(fd, "x", 1)
lseek(fd, 5242880, SEEK_SET)
write(fd, "x", 1)
```

Find the block number of the 1048576-th byte, and the 5242880-th byte.

First consider the tree of blocks. We have:

- `0 - 11` for direct blocks
- $4096/4 = 1024$ blocks in single indirect, so `12 - 1035`
- $1024^2 = 1048576$ blocks in double indirect, so `1036 - 1049611`
- We won't need the triple indirect

The 1048756-th byte is block $1048756/4096 = 256$. Therefore, it is in the single indirect. Specifically, it is the $256-12 = 244$-th block in the single indirect.

The 5242880-th byte is block $5242880/4096 = 1280$, therefore it is in the double indirect. Specifically, it is the $1280 - 1036 = 244$-th block of the double indirect.

You may be wondering where `inodes` are stored. While this is discretionary to the file-system, there are some considerations to be made.

Storing inodes and file system attributes: the Berkeley Fast File System (FSS)
There are four main components to the disk:

- Boot block: containing code to bootstrap the OS
- Super block: containing attributes of the FS (size, number of inodes, etc)
- Inode array: array of inodes
- Data blocks

If we stored these linearly, then we would have long seek times (seek to inode, then seek to data block), as well as redundancy issues (superblock is lost = catastrophic for FS).

The *Berkeley Fast Filesystem (FSS)* accounts for this by

- Partitioning the disc into multiple block groups
- Each block group has:
  - A copy of the super block
  - A copy of group descriptors
  - Data block bitmap
  - Inode bitmap
  - Inode table
  - Data blocks

Group descriptors tell us:

- Location of bitmaps, counter for free blocks and inodes, number of directories in the group

We now turn to consider the implications of multiple processes interacting with file systems, and requirements of `consistency` across these processes.

D. File system consistency with journalling

Systems crash and are unreliable; when crashes occur, it is imperative that once the system has recovered, that processes within the system have a reliable, universal view of the file system.

When deleting a file, what steps does a UNIX system take? Why journalling is important.
When deleting a file, a UNIX file system takes the following steps

1) Mark disk blocks as free
2) Remove the directory entry
3) Mark the i-node as free

Therefore, if there is a problem in between one of these steps, the file system may have an inconsistent state.

Journalling
Journalling involves writing to a buffer of sorts, which is used as a reference/record in case a crash occurs.

Journal entries involvefile system operations - and if changes were not flushed to disk, the OS is still able to understand what changes occured through journal entries.

`ext3 filesystem`: transactions
Transactions in the verb—ext3— FS have four steps:

In Progress
↓
Completed
↓
Committed
↓
Checkpointed

The steps entail:

- In progress: FS updates are still buffered in RAM
- Completed: Updates are buffered in ram - no additional updates for this transaction
- Committed: Updates are written to the journal and marked as 'committed'. This transaction can now be 'replayed'
- Checkpointed: Updates are written to the file system, and the transaction is removed from the journal

The `Journaling Block Device (JBD)` sits in between the file system and the block device/journal, and ensures file systems transactions occur with the above steps in mind.

VIII. Memory hierarchy, caching and performance

We have talked at length across all of our sections about the requirement for performance with respect to file systems, processes/threads and more. We now consider performance with respect to memory sources, and caching performance improvements.

Memory hierarchy: speed ↑, costs ↑!

Registers
↓
Cache
↓
Main memory
↓
Magnetic disk
↓
Magnetic tape

As we descend in the above memory hierarchy, we incur:

- Decreasing costs per bit
- Increasing capacity
- Increasing access time

This means that we cannot have super fast access with large capacity - memory has a tradeoff.

To compensate for slower access times as we descend in the memory hiearchy, *caching* is utilised all across hardware and the operating system to create performant operations.

CPU Cache
We have previously talked about what caching is. The CPU cache (`SRAM`) sits between the CPU and main memory (`DRAM`), and aims to reduce memory latency from CPU calls to memory.

$$\text{CPU} \tag{3}$$
$$\updownarrow \tag{4}$$
$$\text{CPU Cache/}\texttt{SRAM} \tag{5}$$
$$\updownarrow \tag{6}$$
$$\text{Main memory/}\texttt{DRAM} \tag{7}$$

The CPU cache holds recently used data and instructions to save memory access.

But how much do caches actually speed up operation?
This entirely depends on the *hitrate* of the cache - how often it is utilised. Given H, the hitrate of the cache, T_C the time to retrieve from cache and T_M the time to retrive from memory, we have

$$T_{\text{effective}} = H \cdot T_C + (1 - H) \cdot T_M$$

where it is assumed that $T_C < T_M$.

What are some other examples of cache? Spinning disks are slow!
While the `CPU Cache` is in itself a cache for main memory, main memory is too a cache for hard drives/spinning disks!

IX. Memory management

In memory management, we aim to answer to main concerns:

1) Parititioning memory given processes are smaller than memory
2) Partitioning memory given processes are larger than memory

A. Simpler partitioning methods, and their components

Splitting up memory between processes: the challenge
Processes have differing requirements for memory, and also change their requirements for memory dynamically.

We consider 2 broad strategies for partitioning memory for processes:

1) Fixed partitioning (equal sized, variable sized)
2) Dynamic partitioning

Fixed partitioning for process memory
Fixed partitioning comes in two flavours:

1) Equal-sized partitioning
2) Variable-sized partitioning

Equal-sized partitioning
Equal sized partitioning involves pre-configuring equal partitions across memory.

✓ Simple and heuristic method
✗ A process larger than the partition cannot run
✗ Internal fragmentation, with processes being of varying sizes

Variable-sized partitioning
Variable-sized partitioning involves creating partitions of different sizes with queues (or a single universal queue) assigning processes to the smallest partition possible.

✓ Reduces internal fragmentation by allocating to smaller partitionins
✓ Increases memory utilisation by using available memory (for universal queue)
✗ Reduces mostly unused, large partitions (for per-partition queue)
✗ Increases internal fragmentation (for universal queue)

Dynamic partitioning for process memory
Dynamic partitioning allocates the required memory for a process. However, you can begin to imagine some problems with doing this naively

- Do we allocate processes back to back?
  - Then our memory becomes very ”front-heavy”, which isn't great
- How do we deal with gaps in between the memory when processes finish executing?

Then our main concerns for dynamic partitioning are:

1) How do we represent the free space available?
2) How do we allocate the free space available?
3) How do we deal with small, left-over partitions of memory?

Dynamic partitioning: representing free memory
We represent available memory in dynamic partitioning using linked lists. We can imagine that the minimum structure for such a linked would be

```
struct available_memory {
    addr_t addr;
    int size;
    struct available_memory *next;
}
```

Consider free process $P = \{P_1, P_2, P_3\}$, we consider X to be free space. Consider the following scenario when P_2 exterminates.

$$P_1 \to P_2 \to P_3 \implies P_1 \to X \to P_3$$
$$X \to P_2 \to P_3 \implies X \to X \to P_3$$

That is, when the process P_2 terminates, the previously held memory block is still held there. We can begin to think of some issues, such as:

We might have a lot of little partitions; how do we combine them into one?
How do we relocate processes to fill small gaps?

Reducing external fragmentation: compaction and relocation criterion
In dynamic partitioning, we will be often dealt with small partitions of memory inbetween processes. *Compaction* is the process of *shuffling memory contents* to place all free memory together in one large block.

- This requires us to be able to relocate *running programs*
- Generally requires hardware support
- How do we relocate the memory for processes?

There are three types of **memory binding** that are crucial for relocation:

1) At `compile/link` time
   - Compiler/linker binds the process
   - This memory is *not* possible to relocate, as it requires a recompile
2) At `load` time
   - As the process is loaded, memory addresses are binded
   - This memory is possible to reloacted, but requires a reload in memory
3) At `run` time
   - The memory is alloacted at run time, when required
   - This is the easiest memory to re-locate (as it will just allocate in the new region)

Dynamic partitioning: allocating partitions to processes
So we represent free portions of memory with a linked list, but how do we go about actually allocating this memory to processes?

First-fit
Linearly scans the list for the first entry that fits. If the block is too big, splits the block to the size of the process.

✓ Fast lookup, reduces search time
✗ Biases allocation to one end of memory

Next-fit
Like first-fit, but searches from the last allocated process

✓✗ *Supposed* to spread allocation uniformly (in reality, it does not)
✗ Performs worse than first-fit as it breaks up large free space at the end of memory

Best-fit
Find the partition of memory that *best* fits the process size

✓ Reduces external fragmentation, as hole left is as small as possible
✗ But the holes left are often unusable, as they are very small
✗ Have to search the entire space

Worst-fit
Find the block worst in fit, so it leaves a usable block

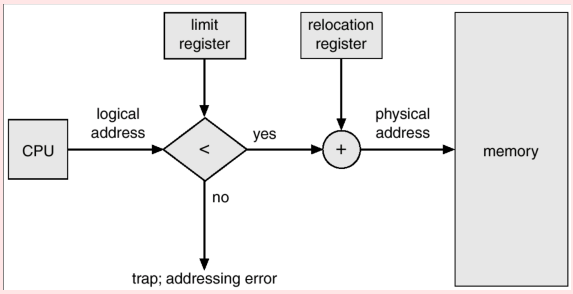
✗ In general, empirically worst than best-fit

In conclusion, **first-fit** is generally better than the others and commonly implemented. In modern OSes, there exists more sophisticated allocation strategies.

But how do we protect processes from accessing eachother's memory?
We now know how to dynamically partition memory, but how do we ensure that processes do not access eachother's memory?

- There exists `base` and `limit` registers
- These registers bound the accessible region for each process
- They must then be changed at `load`, `relocation` and context switch.

The checking of whether accessed memory is within the bound is done by hardware.



Flow diagram for hardware support

To summarise so far, we know:

- We represent free portions of memory with a linked list
- We (generally) dynamically parition memory, using first-fit
- We 'compact' memory within processes where possible to reduce external fragmentation
- We bound the accessible memory for a given process to ensure security

How do these above processes and structures interact with concurrency and context switching? If we don't have enough memory, how do we make room for another process?

B. Concurrency, timesharing and swapping

If our system has to run n processes such that the sum of their required memory is available memory, then we must be able to compromise with the available memory space and 'yield' memory.

Timesharing
There has been this idea of *timesharing* mentioned, where processes in a computer take turns with compute. Similarly, in memory management, we have to consider how processes 'timeshare' memory regions.

Swapping: making space whilst not running
Swapping involves moving the process *temporarily* out of memory into a `backing store`,and then brough back into memory when required to run.

- Swapping transfers *the entire process*
- The backing store is a fast-disk
- Prioritisation is generally used to swap out lower-priority process

Swapping time is proportional to the amount of data in the process, so it is *slow*.

C. Virtual memory

Virtual memory aims to be an abstraction of allocating memory space, by serving translated, `virtual` addresses to applications.

Paging
Paging is the idea of partitioning physical memory into small equal sized chunks called `frames`.

- Then each process' address space gets split into the same-sized chunks called `pages`
- These pages then represent virtual addresses - which have:
  1) A page number
  2) A page offset (as a page can contain main addresses)

The operating system then maintains a *page table*, which contains the frame location for each page, and acts as a translation layer.

✓ No external fragmentation - every single page can be used!

✓ Minimal internal fragmentation with appropiate sizing

Why should we use paging?
Paging is convenient as memory that is not *necessarily contiguous* can be treated as such by the user application. Therefore, many of our issues that could not be solved (compacting compile time memory) are solved by paging.

Page tables and virtual addresses as indices
We have pages that translate virtual memory address to physical memory address. With a `32`-bit address, consider that:

- A one-layer table would have 2^{32} entries!
- So this should motivate some lazy-alloaction...

But we can't lazy-allocate a single layer table. This motivates a multi-layer table.

- Different portions of the virtual memory address indicate indices into different layers of the page table
- The first layer of the page table is *allocated*, and then further layers are `NULL` until utilised.

What do you mean we use the virtual address to index?
For example, in a `16`-bit address

`011010;0111000;101`

Page table entries
What sort of information is stored in these page tables? Well a page table should contain

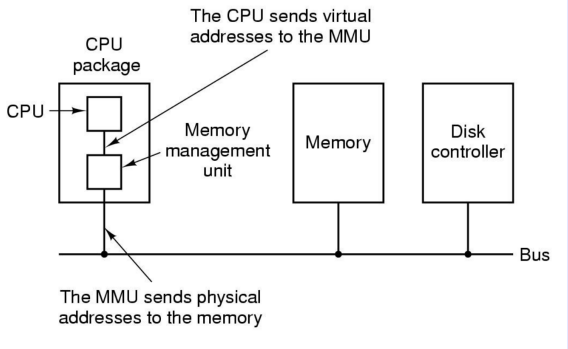
```
struct pte {
    int page_frame_number;
    int present;
    int read, write, exec;
    int caching;
    int modified;
    int reference;
}
```

- The page frame number is used to combine with the offset to create the physical address.
  - It is the *start* of the address range of the page
- `present`: is this page mapped?
- `read, write, exec`: protection bits
- `caching`: should we bypass the cache? (discussed later)
- `modified`: has the page been modified in memory (for writeable)
- `reference`: has the page been accessed?

Memory management unit (MMU)
The `MMU` is hardware responsible for translating

$$\text{Virtual address} \implies \text{Physical address}$$

The mechanism for this is to use portions of the virtual address to access different layer(s) of the page table, and then use some portion as an `offset`.



Memory Management Unit (MMU)

What if a page hasn't been mapped yet? Page faults!
Referencing an invalid page will lead to a page fault (an exception) in the operating system. There are broadly two types of pagee faults

1) Illegal address (out of bounds)
   - We can just signal or kill the process
2) Page not resident (not mapped)
   - Find an unused frame, load the page from disk and update the page table

Shared pages
Much memory is shared between similar processes. It is possible to share pages if it is not self-modifying, and appears at the same address.

So we now know how modern operating systems divvy up memory to be used by processes - but you may notice some performance issues here.

- Virtual memory references are inefficient;
- We have to look up the physical memory address to fetch the PTE
- And another time, to fetch/store the data

How do we improve page tables?

D. Page table improvements and alternatives; inverted, hashed and TLB cache

We know that page tables grow w.r.t virtual address space - but it would be more desired to have this grow with the size of memory itself to save space.

Process IDs in page tables
In a multiprocessor system, pages are allocated to many different processes. Therefore, process identification (`PIDs`) in the form of Address Space

Identifiers (ASIDs) are used to ensure that the accessed process is for the right process.

Inverted Page Table (IPT)
An array of page numbers indexed by physical frame number which is hashed into an index. Contains information regarding the page.

- Compute the hash of the page number, and extract index
- Index into page table, match PID and page number in the entry
- If match, use index value as a frame # for translation
- If not match, chain

IPTs have a few bonuses

✓ IPT grows with the size of RAM, not address space
✓ Saves a lot of space, especially as addresses get bigger

IPT improvement: Hashed Page Table (HPT)
Hash Page Tables (HPT) improve inverted page tables by using a hash function on the virtual address space to access page table entries.

- Page table size is still relative to physical memory size (due to the hash indexing)

We now consider the improvement of page table accesses. Consider that page tables can lead to *two* physical memory addresses

- One to fetch the page table entry
- Another to actually `access/write` the data

How do we improve on this?

Translation Lookaside Buffer (TLB)
Given a job to translate a virtual address, the CPU first looks at the TLB

- If a match exists, then there exists a caching of the PFN of this virtual address
- Otherwise, theres a TLB miss, and then
  - We walk the page table - if it exists, we load this into the TLB
  - Otherwise, we enter a page fault, and also load this into the TLB

The TLB is basically an array of frame numbers, which holds recently used page table entry information.

```
struct tlb_entry {
    int virtual_address;
    int page_frame_number;
    // protection bits, asid, etc...
}
```

The TLB can either be hardware or software loaded - in MIPS R3000, the TLB is software loaded and thus the hardware will generate a TLB miss exception which the software deals with.

E. MIPS R3000 Address Space Layout

Hardware generally has standardised partitions of the address space that are used for specific components of the operating system.

MIPS R3000 Address Space Layout

MIPS R3000

- `kuseg`: 2 gigabytes, user + kernel, TLB translated
- `kseg0`: 512 megabytes, fixed translation window (1:1) to physical, cacheable, kernel code and data
- `kseg1`: 512 megabytes, fixed translation window, not cacheable, kernel only, used for devices
- `kseg2`: 1 gigabyte, kernel only, TLB translated, dynamic kernel memory

F. TLB Exception handling in MIPS R3000

Remember the special exception vectors we were talking about in system calls?

Special exception vectors for TLB refill
For TLB misses in `kuseg` (the user allocatable memory region), there is a special exception handler that is optimised for TLB refill.

- Does not need to check the exception type
- Does not need to save any registers (only `$k0` and `$k1`)
- Does not check if the page table exists

Some other virtual memory related exceptions
These exceptions are handled by the general exception vector

- `TLB Mod`: TLB modify exception, attempt to

write to a read-only page
- `TLB Load`: Attempt to load from a page with an invalid translation
- `TLB Store`: Attempt to store to a page with an invalid translation

Amdahl's Law
Amdahl's Law states that the theoretical speed up of a task is with respects to the part that can be parallelised. Given
- f, the fraction of thee task that must remain serial
- $(1-f)$ the fraction of the task that can be parallelised
- N the number of processors

Then the maximum speed up is

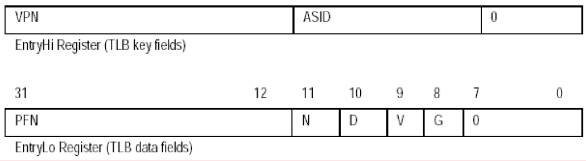
$$S(N) = \frac{1}{f + \frac{1-f}{N}}$$

There exists special registers in MIPS R3000 that are for TLB refills.

MIPS R3000 TLB Properties
In `MIPS R3000`, there exists two components in the TLB entry
- `EntryHi` which contains `page #` and ASID
- `EntryLo` with contains `frame #` and protection bits

There are three special `c0` registers for TLB refills
- `c0_EntryHi`: used to read/write individual TLB entries
- `c0_EntryLo`: used to read/write individual TLB entries
- `c0_Index`: used as an index to TLB entries

MIPS R3000 TLB Bits



MIPS R3000

- N = Not cacheable
- D = Dirty
- G = Global (ignore ASID)
- V = Valid

Exam question style: finding TLB matches in a table
In MIPS R3000, the final 12 bits is the offset. Therefore, the first 20 bits is the VPN given a virtual address.

EntryHi	EntryLo
`0x00028200`	`0x0063f400`

Given that the current EntryHi is `0x00000200`, we have that the ASID is `0x200`, therefore we should only consider EntryHi's that end with `0x200`.
- Consider we look for `a = 0x00028123`
- The VPN is `a >> 12`, therefore `0x00028`
- This matches with our table entry, and the ASID matches too.
- Check the permission bits of EntryLo.
- Valid bit is set to 0, therefore INVALID mapping.
- Otherwise, physical address would be `0x0063f123`

EntryHi	EntryLo
`0x0005b200`	`0x002af200`

- Consider we look for `a = 0x0005b888`.
- The VPN matches, `0x0005b`
- The ASID matches (if it doesn't check for global)
- Now consider the low bits.
- We have `0x200 = 001100000000`
- Dirty bit is 0, so read only, and a valid mapping.

MIPS R3000 TLB management instructions
- `TLBR`: reads in to EntryHi and EntryLo w.r.t index register
- `TLBP`: reads in EntryLo given an EntryHi
- `TLBWR`: writes EntryHi and EntryLo to a pseudo-random position
- `TLBWI`: writes EntryHi and EntryLo to aa location pointed to by the index register

G. Paging performance considerations

We now consider different performance considerations of paging, such as
- Should we load pages on demand?
- Should we predictively load pages for processes?

Principle of Locality: things close together are used together

Programs tend to reuse data and instructions they have used recently.

You can then motivate that we can exploity this 'locality' of references. We could reasonably predict that given some instructions and data, we have some idea of what data/instructions will be needed.
1) **Temporal** locality: recently accessed items are likely to be accessed in the near future (caching)
2) **Spatial** locality: items whose addresses are near one another tend to be referenced close together in time

Working set: what does a program need in a given time period?

The pages required by an application in some time window Δ is called it's memory working set

Note that it is defined in *pages*. Therefore, the *working set* $\approx$ program's locality. If Δ^* is an estimator for Δ, the true locality window

- As $\Delta^* \ll \Delta$, the time window does not encompass the entire locality
- As $\Delta^* \gg \Delta$, the time window compasses severl localities

So what is a good Δ?
A good Δ approximates what is required for execution without having too much irrelevant information.

- The working set tends to change gradually

This motivates one method of loading pages - `demand` paging.

Demand paging: get what you need
In demand paging, we reload non-resident pages on demand (when we require them). With the principle of locality, it should be true that we rarely access non-resident pages.

But what occurs when we have a lot of processes in memory that all demand more and more pages? Eventually, with enough multiprogramming, every process will keep requesting pages but never have the **resident set** required.

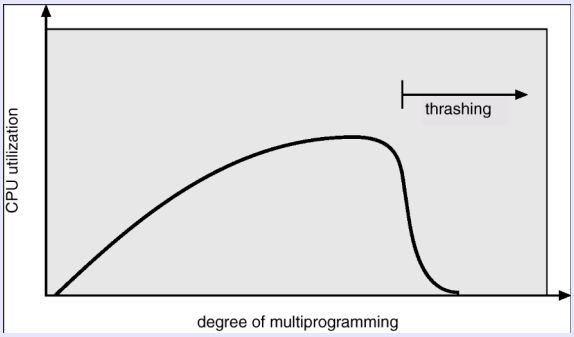
What is the required resident set?
The `resident set` refers to the pages that are loaded in the page table/physical memory, and have valid mappings.

A process' *required* resident set is the pages it requires to execute it's current task.

Thrashing: too many requests!
As # of processes ↑, the amount of memory for each process ↓.

- Therefore the available resident set for each process decreases
- With this, the amount of demand paging increases
- But since every process demands more pages, processes don't get there required resident set

So while multiprogramming ↑ CPU utilisation, at some point processes do not have the required resident set and cannot run.



Thrashing occurs beyond a certain point

To recover from thrashing, we can simply suspend a few processes to provide more memory.

A demonstrative example of locality: what's faster?
We previously talked about how things that are close together (have locality) are often accessed together. Consider the below code snippet.

```
int array[10000][10000];
int i, j;
for (int i = 0; i < 10000; ++i) {
    for (int j = 0; j < 10000; ++j) {
        array[i][j] = 0;
        array[j][i] = 0;
        // what's faster?
    }
}
```

In this case, accessing the arrays in order `[i][j]` is faster, as they are more local (2D arrays are an array of arrays).

- Row wise access has more spatial locality, and is more cache friendly
- Column-wise access skips between many memory addresses

H. Virtual memory management policies

There are a few considerations for virtual memory that you may have noticed so far.

- Page table format; multi-level? inverted? hashed?
- Page size
- Fetch policy
- Replacement policy
- Resident set size
- Page cleaning policy

Page size: how much memory should pages hold?
As page size ↑

- ✓ Decreases number of pages
- ✓ Increases TLB coverage (more data per TLB entry)
- ✓ Increassing swapping I/O throughput

✗ Increases internal fragmentation
✗ Increases page fault latency (more data to read from disk)

Fetch policy: how should we fetch pages?
We have two options

1) Demand paging
   - Lots of page faults when a process first starts
   - But we make no assumptions about the processes
2) Pre-paging
   - Pre-paging brings in pages in anticipation of a program's working set
   - Wastes `I/O` if pages are not used

Replacement policy: when pages are full, which one should we evict?
When the page table is full, which page should be chosen to vacate a frame?

- Page removed should be the page least likely to be referenced in the near future
- There are some frames that are **locked** - important to kernel operation for e.g
- The frame table has a `pinned` bit/flag to keep track of this

Consider some replacement policies

First-in, first-out (FIFO)
Just remove the oldest page. The age of a page is not necessarily related to it's usage in the near future however

Least Recently Used (LRU)
Choose the leaast recently used page. Assumes if a page has not been references for a long time, it is unlikely to be referenced in the near future

- Is true under the principle of locality
- A time stamp is required for each page (but is proxied)

What are some proxies for least recently used?

LRU proxy: clock page replacement
In clock page replacement, the frames are ordered in a circular fashion and a 'clock hand' (the CPU clock) points to one frame at a time

- Have a `reference` bit in the frame table
- Set to `1` when page is used
- As you scan for a replacement, if
  1) If `R = 0` evict this page
  2) If `R = 1`, set to `0`

Since recently used pages (`R = 1`) get a 'second chance', it is also called second chance replacement.

Resident set size: how many frames should each process have?

- Fixed allocation
  – Gives a process a fixed number of pages
  – High utilisation is an issue, requirement for different processes has high skew
- Variable allocation
  – Number of pages allocated to a process varies over the lifetime of the process

We differ variable allocation by the scope of 'variability'

Global scope
The OS keeps a global list of free frames, and allocates frames to process on a need basis.

✓ Automatic balancing across system
✗ Does not provide guarantees for high-priority processes

Local scope
The OS allocates page frames based on a few criterion

- Application type
- Program request
- Other criteria like priority, etc

Cleaning policy: how do we deal with dirty (written) pages?
Pages can be written to and modified in memory. These changes need to be written to disk eventually. There are two main approaches:

1) Demand cleaning
   - Write out page only when it is selected for replacement
   - However this means the replaced page has high latency until ready (write times are long)
2) Pre-cleaning
   - Pages are written out in batches in the background
   - Increases likelihood of replacing clean frames
   - Overlap `I/O` with current CPU activity

It is clear that it is faster to replace clean pages as no writes occur, so we want to replace clean pages where possible.

X. Multiprocessor systems and their implications

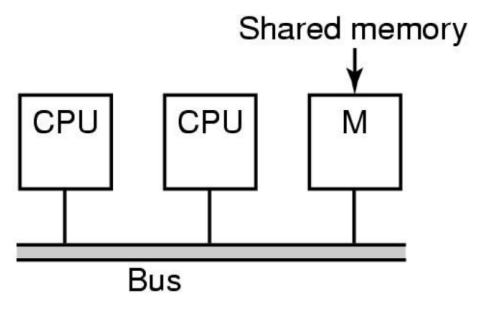
In modern compute, we have seen an exponential growth and then tapering of CPU clock-rate, but a constant exponential growth of *core count*.

As the number of *CPUs/cores* grow, the way we handle

concurrent memory accesses for multiple processors becomes an issue.

Bus-based uniform memory access multiprocessors
Bus-based `UMA` multiprocessors are structured such that each CPU/core share a single memory bus to access memory.

- Bus controller/mechanism resolves parallel access
- Access to all memory occurs at roughly the same speed for all processors



Bus-based multiprocessors

There are clearly considerations to be made with sharing a bus and memory - which we will consider next.

Consideration 1: Adding caches, and cache consistency
Each CPU has it's own cache to increase memory fetch performance, as constantly using the `bus` would grow to be unscalable.

But how do we deal with consistency of cached memory across CPUs?

- Generally handled by hardware
- Writes to one cache propagates to *or* invalidates entries on other caches
- But cache transactions consume bus bandwidth

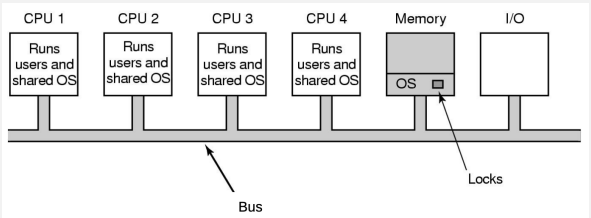
Consideration 2: How do we run an OS for multiple processors?
There are two approaches to running an OS for multiple processors

1) Per-CPU OS
   - ✓ Simpler to implement
   - ✓ Scalable, as there is no shared serial sections
   - ✗ Each processor has it's own scheduling queue
   - ✗ Each processor has it's own memory partition
2) Shared OS (Symmetric Multiprocessors)
   - ✓ Load and resource balancing between processors
   - ✗ Concurrency in the kernel, so synchronisation available

Why should we share the kernel? What advantages/disadvantages does this have?
Sharing the kernel between multiprocessors has the following implications

- ✓ Get global scheduling, which ensures that CPU's are treated equally
- ✓ Automatic load balancing
- ✗ Treating equally not always great, some CPUs have hot caches
- ✗ Per-CPU OS gives more control, by avoiding contention, by is more complex

Symmetric multiprocessors: synchronising the kernel



Symmetric multiprocessor

There are a few approaches to deal with synchronisation in the OS

1) A big lock around the entire kernel
   - Obviously this will be a large bottle neck, as this is essentially a single-threaded kernel
2) Mutex independent parts of the kernel
   - Requires complex lock ordering and analysis
   - But far more efficient and allows parallel computation

Now we know how to synchronise the kernel, how do we synchronise variables?

Why can't we preempt spinlocking processes?
Consider a process P_1 that is spin-locking for the access of some variable v.

- Given preemption is allowed, the scheduler will attempt to block
- But preemption of lock holders extends the time of the spin-locks
- Therefore we waste more time in context switching

Multiprocessor synchronisation: read, test *then* set
In `test-and-set`, we spinlock at the test until the variable is ready to be taken. Why is this not plausible for multiprocessor systems?

- Read the above example for reasoning
- So we have to block all other CPUs from

accessing the bus during `TSL`

We instead **read, then test and set** on a multi processor system.

1) Read the lock variable, if it is not taken
2) Attempt to `test-and-set`
3) If not successful, continue spin-locking on the read

```
start:
    // spin on read
    while (lock == 1)
        ;

    // try taking
    r = TSL(lock);

    // got taken already
    // re-spin
    if (r == 1)
        goto start;
```

This means that only the processor is spinlocking instead of the entire machine being blocked.

Hold on; why don't we just block like a uniprocessor system?

In our original discussion about test-and-set, we said that blocking a waiting thread was more efficient than spinlocking, so why don't we do this?

- Switching to another process is *costly*
- Spinning wastes CPU time *directly* - but switching to another process is *atomic*
- If a lock becomes available mid-way through a switch, then we must switch back
- And this wastes time during the switch

As a rule of thumb:

If the lock will be held for longer than $2\times$ the switching overhead, then it's more efficient to block.

Blending blocking and spinning with a hybrid lock

An easy way to blend the positives of switching and spinning is to have a hybrid lock that

- Attempts to take the lock with test-and-set
- If it fails, spins with read then test-and-set for $\approx$ 1000 cycles
- Then switches threads