ENCE360: Operating Systems

Jordan Pyott

2021-05-06 16:57

Contents

Operating Systems **3**
- Course Information . . . 3
  - Textbooks / Resources . . . 4
- Readings . . . 4
- Lectures . . . 4
  - Lecture One - Introduction to Operating Systems . . . 4
  - Lecture Two - Processes and Threads (2.1, 2.2) . . . 9
  - Lecture Three - Processes and Threads (2.3, 2.4, 2.5) . . . 17
  - Lecture Four - Deadlocks, Starvation and Thread Patterns . . . 19
  - Lecture Five - Inter-process communication, Signals and Pipes . . . 20

Operating Systems

These notes are designed to be used in conjunction with the slide sets provided in the course, the slides will be more helpful for use in the labs *due to including code examples*, these notes will provide a good outline for studying for the final exam.

Course Information

The course covers core topics.

- Introduction to operating systems
- Processes and Threads
- Pipes
- Sockets
- Deadlocks
- Files and Directories
- Input/output
- Memory management - Caches
- Memory management - Virtual memory
- Virtualisation

Grade structure

Standard Computer science policy applies

- Average 50% over all assessment items
- Average at least 45% on all invigilated assessment items

Grading structure for course

- Lab Test (20%)
- Assignment (20%)
- Lab quizzes (10%)
  - Weekly Quiz Assessments
- Final Exam (50%)
  - Closed book and no calculator
  - Cheat sheet Double sided A4

Textbooks / Resources

- Modern Operating systems - Andrew Tanenbaum
- Xv6 - Online shorter method, lots of examples

> For information on these resources see the first lecture slides

Readings

Lectures

Lecture One - Introduction to Operating Systems

The beginning of computing

- Called the Analytical engine
- Charles Babbage 1972-1871
- Digital, programmable, *Turing complete*
- Punch card IO
- Unable to be engineered
- Would be very slow
- Ada Lovelace - worlds first programmer

1st Generation computers

In 1945 we moved on to hard-wired machines. These were just a plugged set of wires.

2nd Generation computers

Operating systems started to appear when systems were designed to be programmable, this came with programmed batch systems, they operated with one job at a time and had storage, this is the initial life of an operating system.

3rd Generation computers

- Multiprogramming
  - the ability to run multiple jobs at once
- First real operating systems
  - MULTICS/Unix/Linux, VMS and others

This bought the first initial need for security and segregation between users on the same machine.

4th Generation computers

This is the first view of **personal computers**, bringing the `BASIC` interpreter using Machine code, complexity hidden from the user, one program could be held in memory.

Usually had ~8 kb of memory to run the entire operating system.

Eventually got a GUI, use of mouse and the initial real world of what we call computers, could also store multiple applications in memory at once.

Then finally, we have modern day computers

- Personal
- Multiple applications at once
- Modern OS, *(Linux, MacOS and Windows)*

5th Generation computers

Wearable devices, quantum computing and further AI development, we are not there yet!

What is in a computer?

- CPU
- Memory
- Video Controller
- Keyboard Controller
- Optical disk controller
- Hard disk controller

This is even a simplistic model, a computer really looks like this:

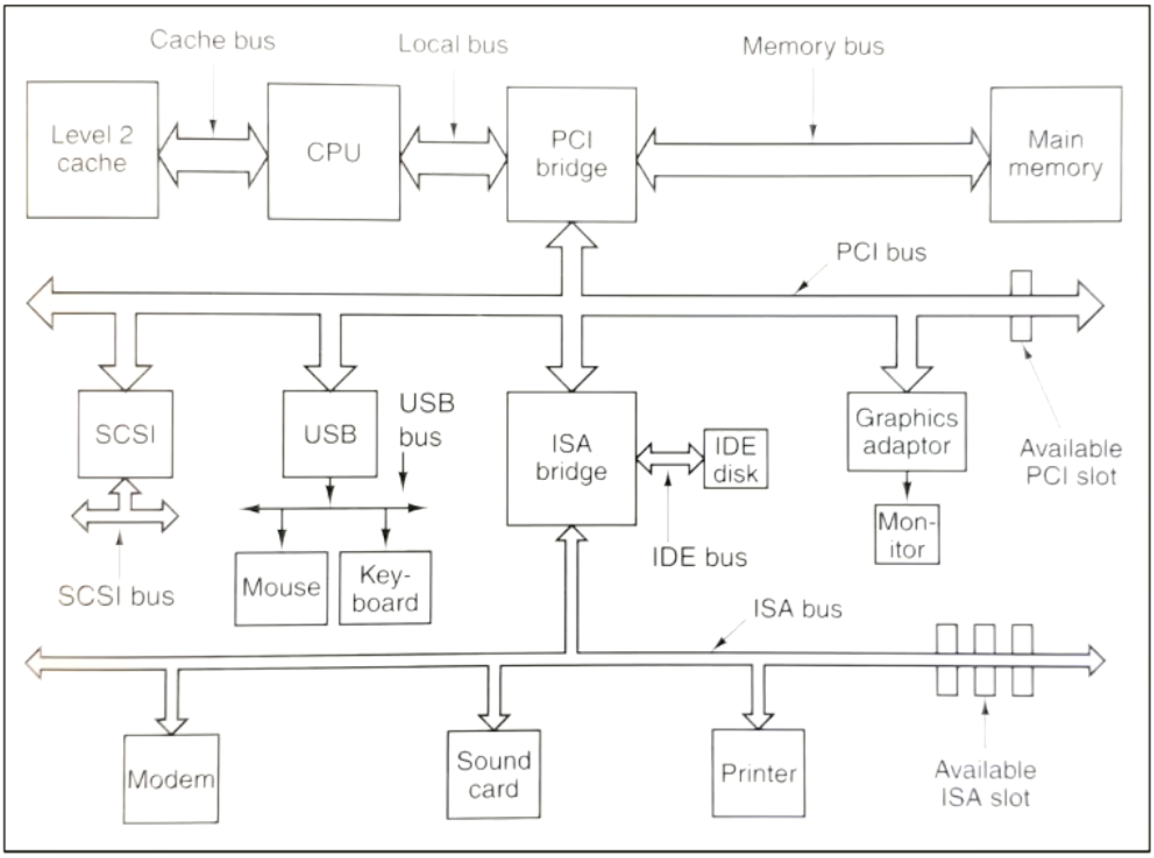

Figure 1: Computer Model

It is extremely difficult to have to write to all sectors of this without knowing its exact structure and expected input. This means that we need some interface to handle this for us. Hence, **operating systems**.

Storage hierarchy

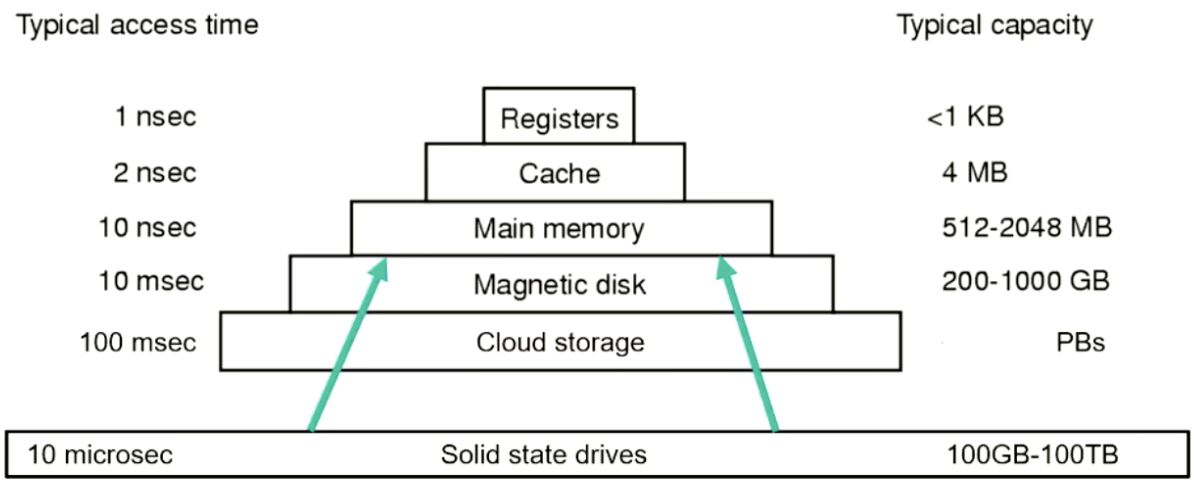

Figure 2: Storage hierarchy

In order for the operating system to work, it needs to have some method of handling storage, and knowing where we are allowed to write to, and have some method of creating an interface or abstraction to solve writing issues.

What is the main purpose of an operating system?

- Virtualization (sharing users)
  - Time (CPU)
  - Space (memory)
- Concurrency
- Persistence (I/O)
- Protection
- Hides complex details
- Protects the machine from malicious code

Core OS concepts

These will be expanded throughout the lectures

- Processes
- System calls and kernel mode
- Address spaces
- Files and IO
- The shell

Typical process model

- Processes are normal, sequencial code
- The schedular decides what runs and when
- Alternative cooperative models exists

OS API: System calls

Process management

Call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process' core image
exit(status)	Terminate process execution and return status

File management

Call	Description
fd = open(file, how, ...)	Open a file for reading, writing, or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information

Directory and file system management

Call	Description
s = mkdir(name, mode)	Create a new directory
s = rmdir(name)	Remove an empty directory
s = link(name1, name2)	Create a new entry, name2, pointing to name1
s = unlink(name)	Remove a directory entry
s = mount(special, name, flag)	Mount a file system
s = umount(special)	Unmount a file system

Figure 3: System Calls

These really work because the user has no direct access to the low level routines

- OS runs in kernel mode with higher privileges
- User mode system calls execute TRAP instructions
- Hardware looks up the **Trap table** to find address

Address spaces

Modern operating systems have **Virtual Memory**

- Multiple programs in memory at once
- Idle memory can be paged and swapped to disk
- Give an illusion of unlimited memory (*at a price*)

Sample exam question

Which of the following is NOT an operating system?

- Linux
- Windows
- Android - NOT
  - Is using Linux kernel
- ROS - NOT
  - Is using Linux kernel
  - Is a set of libraries to use with an OS to make system calls
- MacOS
- DOS
- iOS
- Arduino - NOT
  - Is a package

Lecture Two - Processes and Threads (2.1, 2.2)

> NOTE: See lecture slides in order to find code examples of using threads and processes

Processes Program counter

One program counter for each process, this allows us to keep processes independent and sequencial, this is the foundation of **multiprogramming**, the timer will interrupt a process and switch when its allocated time frame is done.

This is an example of **Pseudo parallelism**

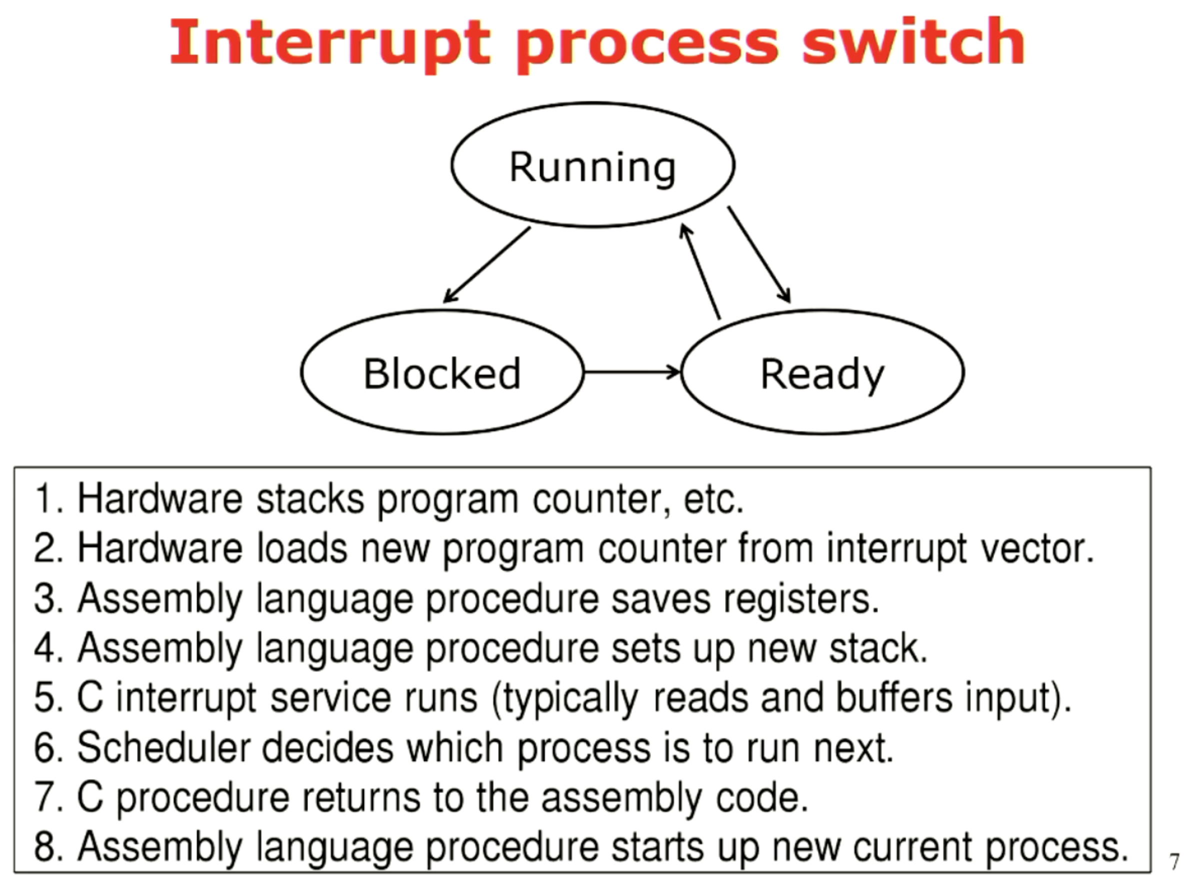

Figure 4: Process switching - Interruption

This is building up the stack

What is the stack?

The stack is going to hold the context of what I am doing, it is a LIFO representation of all the local variables, function calls and procedure calls.

> This will be talked about further later

Process Creation and Termination - Linux

Creation

1. System initialization
2. Process creation system call
3. User request to create a new process (shell)
4. Initiation of a batch job

Termination

1. Normal exit *voluntary*
2. Error exit *voluntary*
3. Fatal error *involuntary*
4. Killed by another process *Involuntary*

Linux Process Hierarchy

- Linux processes are a tree like structure of daemons and foreground processes
- All processes belong to a parent *except the init process*
  - Process group receives all signals from the creator
- Running a program starts a new process
- Windows has no concept of process hierarchy
  - Process is independent of its creator
- Each process has a process table
  - we will need to save this table in order to store state of a specific process
  - These are called process control blocks (PCB's)

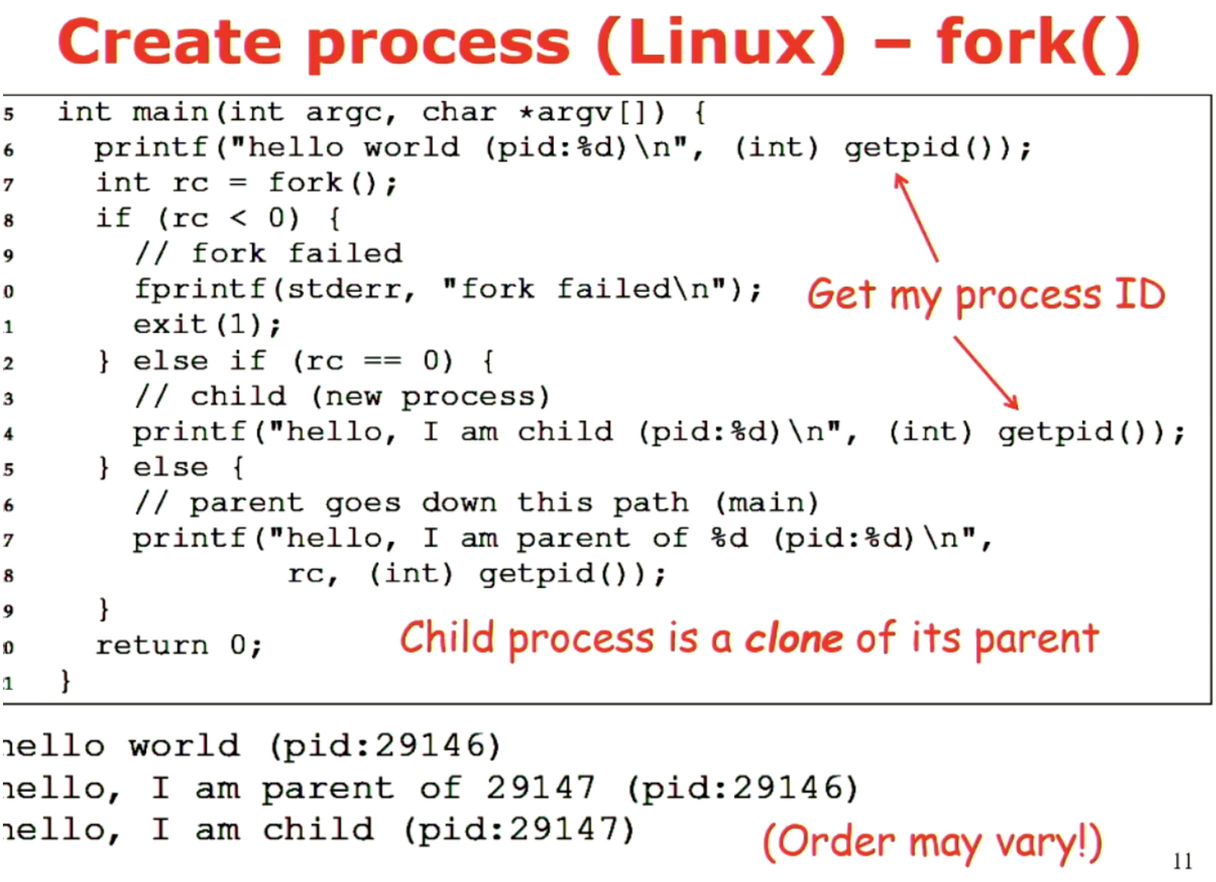

Figure 5: Create Process

In the above block of code, the `fork()` function is being called to create our process. We then can see our process ID and the process is now a *clone of its parent* at the point of creation.

The wait() function

- We can use wait in order to wait for any process to be stopped, we can also use `waitpid()` in order to check if a single process has stopped, this can be useful to detect crashes and log exits.

The exec() function

We can use the `exec()` to run a new process as a child of the current process.

This is extremely useful for re-directing output, for logging, helpful output and more versatile output to play with.

Process summary

- A process is an independent resource group running a single program

- Unix: all processes are created and owned by a parent
- Forking a new process creates a clone of the parent *including the same program counter*
- Exec*(file…) replaces the current program context with the new program file contents
  - Operations such as redirecting and piping output can be run before the program loads
- The code in a process runs sequentially
  - or, does it???

Why concurrent applications?

- Allows parallelism of independent operations in a single program *that share common data*
- Example: word processor (basic):
  - Task 1: respond to user input (updating model)
  - Task 2: reformat document when model changes
  - Task 3: save periodically to disk
  - Task 4: spell check

Concurrent applications will allow us to treat these as all independent tasks, treating these as processes will run into issues as they do not know when other processes have completed tasks this means we will have to use signals every time that we want to achieve something that requires information from another process. *we can do this, and will, but not today*.

We can use a **finite state machine**

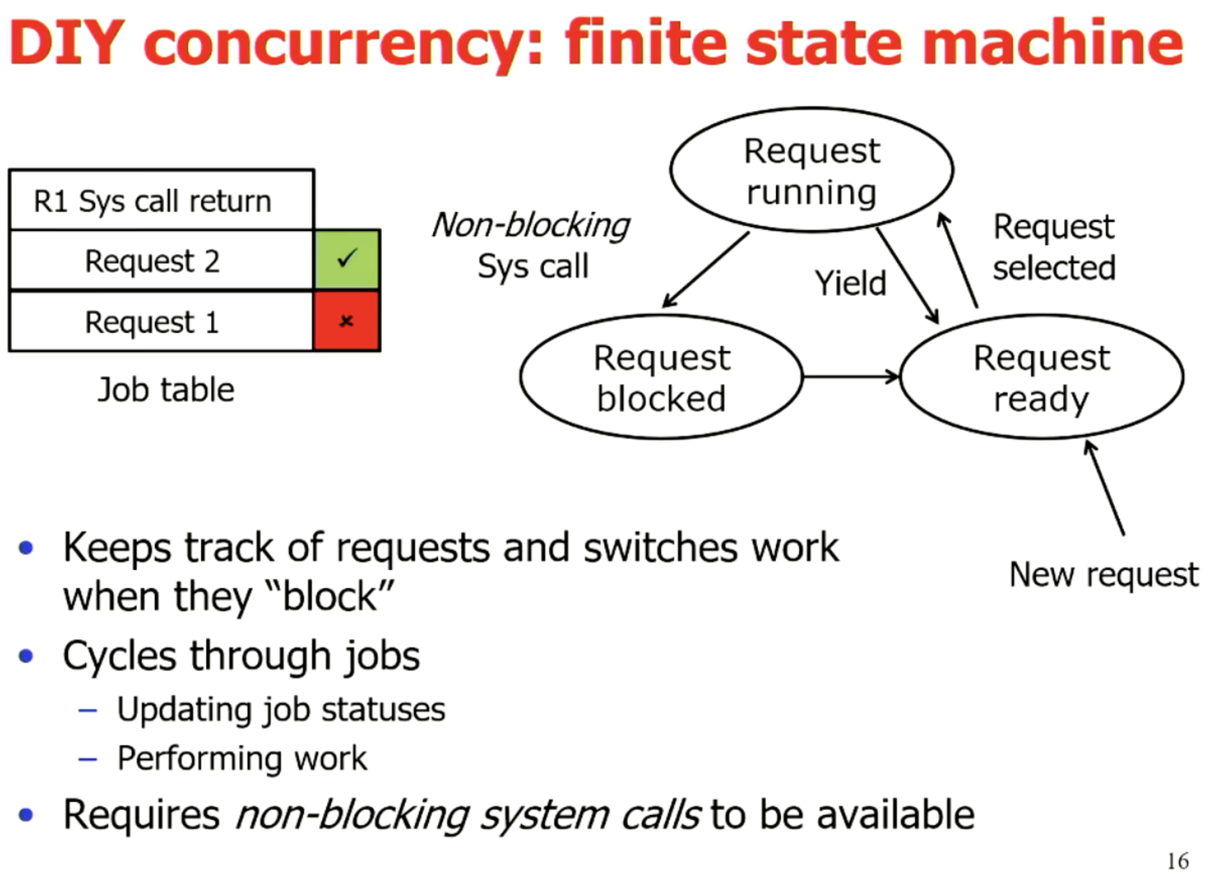

Figure 6: DIY Concurrency

Instead of doing this with a finite state machine, we can use **Threads**

Threads Threads have there own thread table, this is called a process data space, multiple threads can access the same process data space.

- Threads have access to process data space
  - but not direct access to each other
- The thread is not a clone, it runs a callback function
- Waits for completion
- The order on the stack can vary as each thread has its own stack

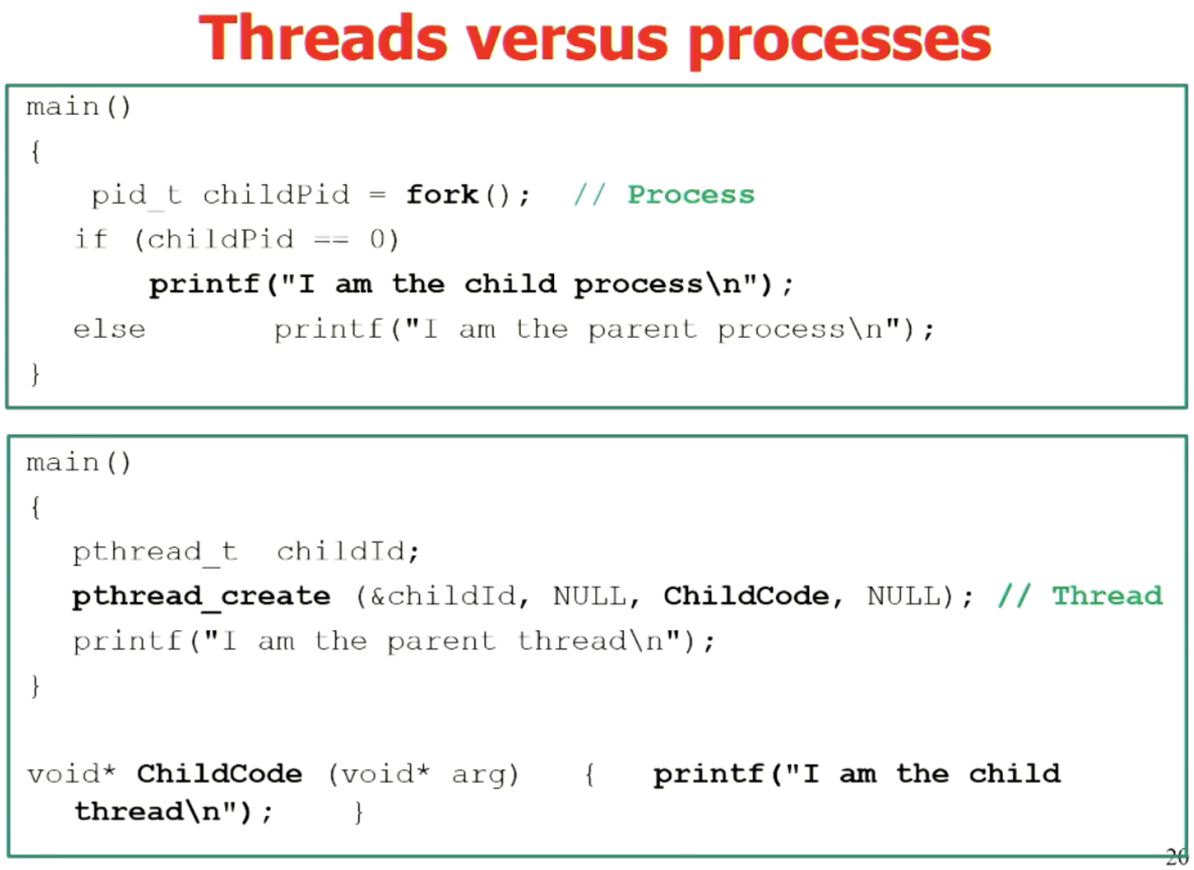

Figure 7: Threads vs Processes - code example

POSIX threads API

- Standard runtime library calls for managing threads:
  - `pthread_create`
    * creates a thread to execute a specified function
  - `pthread_exit`
    * Causes the calling thread to terminate without the whole process terminating
  - `pthread_kill`
    * sends a *kill* signal to a specified thread.
  - `pthread_join`
    * Causes the calling thread to wait for the specified thread to exit. Similar to `waitpid()` for processes
    * Waits for the child thread to finish

        * Generated by `pthread_join()` from `pthread_exit()` after the thread has exited
        * Need to be vary careful about how you return values from a thread
        * We need to return to the heap not the stack in order to make it global, note this will need to be freed
    - `pthread_self`
        * Returns the callers identity *The thread ID*
    - `pthread_yeild`
        * Yields the CPU to another thread
    - There are many more calls available in the **man pages**

Threads vs Processes

- Access to the same data makes communication easy
- Very fast to create, can be fast to switch
- Possible performance gains from switching withing a possible process
- Can be spread across CPU's for further parallelisation
- Difficult to write and debug the code
    - Ordering issues
    - Data access issues

How are threads and processes implemented?

- Threads and processes can be implemented both at the User level and in the kernel.
    - Threads and processes in the kernel are not run by the scheduler, nor handled in the user space

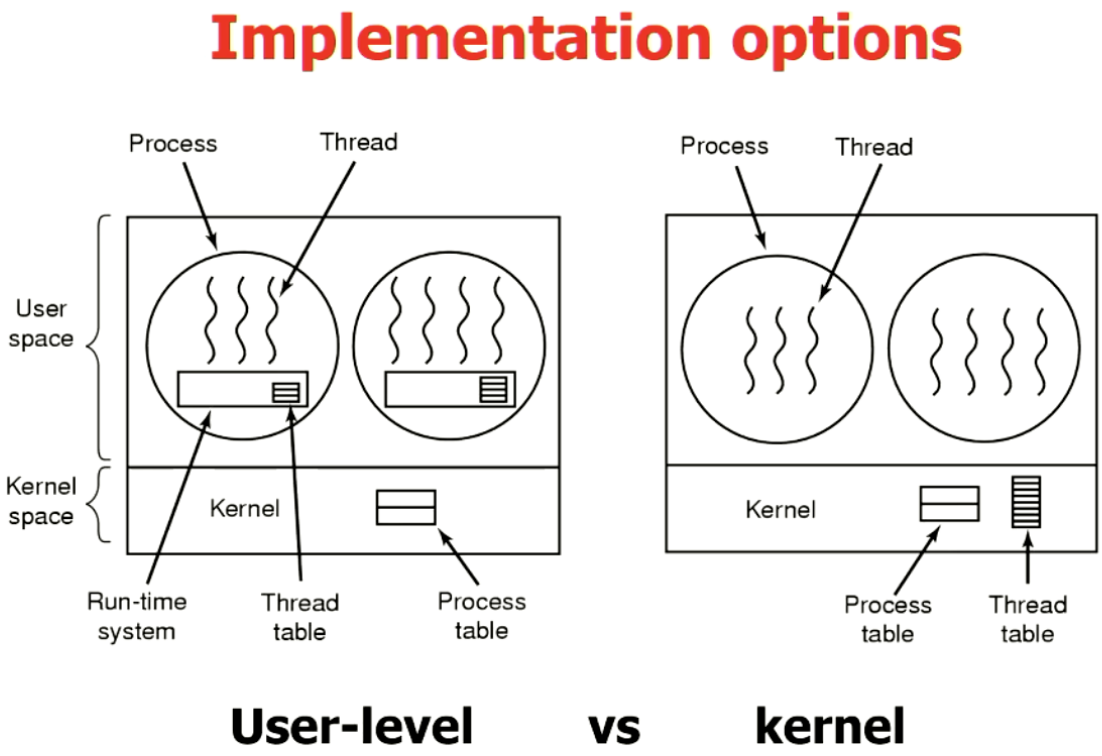

Figure 8: User-level vs Kernal implementation

Lecture Three - Processes and Threads (2.3, 2.4, 2.5)

Critical Regions

- Mechanism to provide "mutual exclusion" of critical regions
- No two processes in critical region at the same time
- No assumptions about CPU's

Mutual Exclusion

Four conditions to provide workable mutual exclusion:

- No two processes simaltainously in critical region
- No assumptions made about speeds or numbers of CPU's
- No process running outside its critical region may block another process
- No process must wait forever to enter its critical region

Non-solutions, these violate the above conditions

1. Bocking process disables interrupts on CPU
2. Lock variable, updated just before entering the critical region
3. *strict alternation* (*lecture slide 6*)

- Loops on block
- What happens if one process is much slower than the other, or halts?
    - Which condition is violated?

4. *Peterson's solution*

- Each process hands off the "turn" to the other process
- Blocking occurs in the critical region only
- Still "busy waiting" - prone to CPU "priority inversion problem"
- Some systems reorder memory access
    - Simpler solutions exist in hardware, this system is sometimes built into the CPU

Block on wait - mutual exclusion

- Sends signals between process to wakeup the other process, we then sleep our own process
    - Uses a counter, when it reaches 0, it sleeps itself and parses a signal to the other process
- What happens when a wakeup signal is sent to a process that isn't asleep?
    - This is because we are assuming that this is an atomic operation, however it is not,

Semaphores - An actual used solution to solve mutual exclusion

Atomic operations to raise or lower their value

> NOTE: fix this up from the slides

Implementation:

- Setting lock by decrementing semaphore
- Releases by incrementing semaphore

POSIX thread implementation

- Mutex atomically locks/unlocks
    - Uses for access to critical region

Operation	POSIX Threads
Create condition variable	`pthread_cond_init()`
Destroy condition variable	`pthread_cond_destroy()`
Block waiting for signal	`pthread_cond_wait()`
signal and wake one thread	`pthread_cond_signal()`
signal and wake all threads	`pthread_cond_broadcast()`

> NOTE: add code example from lecture slides

This is a good solution, however it falls apart when we are using Distributed CPU's, the solution to this is using **message passing**.

We can also use **Barriers** in order to solve synchronisation problems, this is the use of signals to wait for multiple threads to complete before we can continue executing on a particular thread.

Readers and Writers

- Multiple readers can access the area at the same time
- Writers require exclusive access
  - All readers need to exit before the writer can gain the lock
  - New readers may arrive while writer is waiting

> Is there a better solution to make this system more fair for the writer using Semaphores or Mutex's

Lecture Four - Deadlocks, Starvation and Thread Patterns

We use semaphores such as `mutex`, in order to prevent deadlocks and starvation.

How to make single-threaded code into multi-threaded code

This is reasonably difficult to do, here are some of the issues:

- Local variables are fine; global variables may need to be thread specific
  - Some languages implement *thread_local* declarations (C++)
- Libraries called may not be thread-safe
  - Who consumes signals?
  - Memory access needs to be atomic

    - Can be solved via mutual exclusion libraries (locking)
    - If they are thread safe, this will be outlined in the man pages

- Kernel processes may not be thread-aware (stacks)

Thread Patterns

Dispatcher/worker thread pattern

- Dispatcher thread handles incoming requests and farms out to pool of worker threads
- Workers get woken to perform their task
- Example of this layout is a Web Server

Team/pool

- All team members are equal:
    - wait for incoming requests and grab one to process
- Managed pool: threads in pool created and destroyed by library
- Can be specialised too (running different code)
    - place inappropriate requests into an internal queue (to redirct to other threads)

Pipeline

- Chain of consumer/producers
    - Each thread consumes a request and produces a new one for the next thread
    - Specialised threads designed to minimise latency

pop-up threads

- Thread created when needed to handle new request
    - starts a fresh thread, this is faster than context switching in threads

Lecture Five - Inter-process communication, Signals and Pipes

Signals are a simple route of communication, that is primarily for exception handling, but there are also signals for other things

- Fixed set of signals (Unix):
    - `SIGINT`: The process is being interrupted; terminates quietly
    - `SIGQUIT`: Forces the process to end and core dump
    - `SIGILL`: FIXME - fill here

- `SIGSTOP`: which stops the process from executing (*This cannot be stopped*)
- `SIGKILL`: which kills the process (*This cannot be stopped*)

Signals are implemented in hardware (division by zero), handled by the OS, (file size limit exceeded), and is primarily handled by the user (via keystrokes)

Other processes such as a child process notifying its parent that it has terminated (`SIGCHLD`), or sending a signal.

- Signal numbers range from $\{0, \dots, 31\}$
  - We will refer to these signals by name rather than reference code
- uses `PID`: process or processes to receive a signal
- if `pid = -1`: all processes user has permission over
  - Elevated user (*such as sudo or root privileges*)
  - All processes with same user ID
- if `pid < -1`: All processes in *process group*