Comp2009
Operating Systems

a)What is an operating system?

- NOT
  - Applications (word etc...)
  - Look & feel (GUI)
  - Hardware architecture
  - Software libraries

"Bridge between applications and hardware level"

- Definition depends very much on your perspective:
  - Engineer: *"resource manager"*
  - Computer Scientist: *"process abstraction"*

b)OS & Interfaces

"The real challenge is coming up with 'nice' abstractions for programmers"

- What are 'nice' abstractions?
  - Easy to understand for the user
  - Efficient to implement and use

c)Why study operating systems?

- Moores law not leading to speed increases for the user due to bad interfaces (especially in threads) so its hard for programs to make use of increases in cores (clock speeds have stopped increasing)

- Embedded systems are becoming more popular as CPUs are cheaper → they all need Oss

d)Components of an OS

- Process Abstraction
  - A process is roughly a program in execution
  - Multiprogramming: running multiple processes in the same CPU
  - Programs can have several threads of execution inside them
- Memory & Filesystems
- I/O
  - Talking to devices with drivers
  - Each device has its own pecularities

The Structure of an operating system

a)Structure
Bootstrap
Kernel
Ultilities

b)Types
Monolithic
Layered
Virtual Machines

c)Bootstrap
First part of an operating system to be loaded

Olden times:
- Primary bootstrap toggled by hand
- Secondary bootstrap loaded from tape followed by the OS

Modern:
- Primary bootstrap loaded from PROM
- OS loaded from disk or over network

d)Kernel
"Contains the heart of the OS"
- Drivers
- Filesystem code
- scheduler and other process management code
- memory management code

Bulk of this course is about the kernel

e)Kernel design
A simple library
- Can be called by the user/programmers
- Works out but doesn't allow asynchronus activity

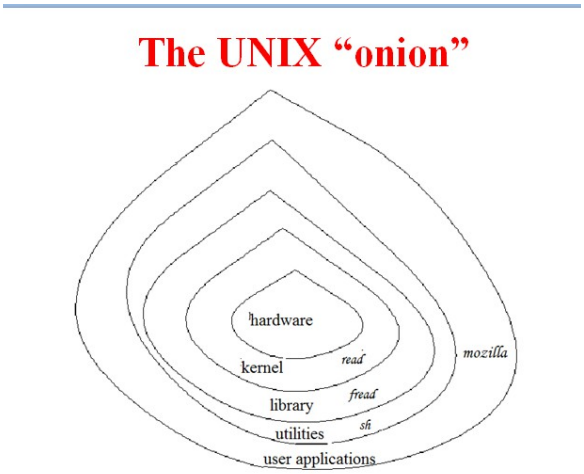
A simple library with Interrupts (MSDOS)
- Interrupts for I/O, keyboard, printing etc...
- Allows for some background activities
- Limited for multi-tasking

Protected OS
- Supports traps or 'software interrupts'
- Utilises hardware memory management to protect code & data from processes
- Typically has multitasking support

f)Ultilies
Several programs/ultilies are needed to setup and maintain an OS which may include

- File system checking and repair
- File system backup
- Printing
- Shell
- GUI (X windows..)
- User admin tools
- Filesystem tuning and exploring
- Queue management
- Compilers (sometimes needed for installing drivers)

g)Unix Onion
Way of modeling an OS

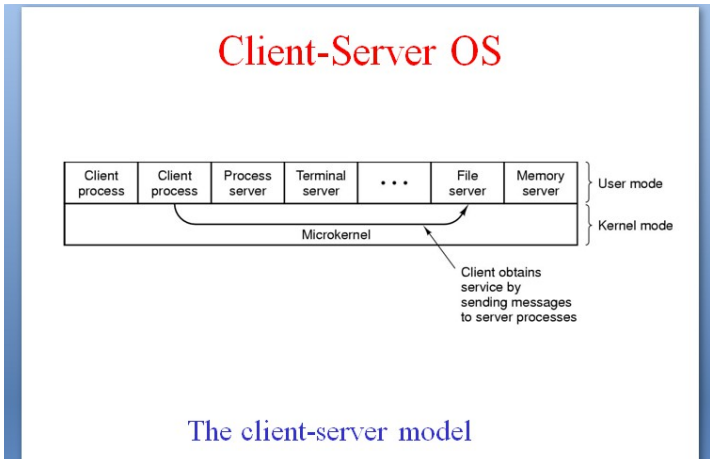


h)Interrupts

- Essential part of asynchronous activity
- In systems with memory management, interrupts cross domain boundaries by being run as *Supervisor* or *privledge* modes
- *Supervisor* mode allows the interrupt code to execute machine instructions and access the whole of memory
- Hardware supports interrupts by auto saving some registers such as the status register
- May switch computer to another stack

j)Scheduling

- Most OS allow multitasking (multiple processes on the same cpu)
- MSDOS allowed one thread per process
- Most allow for numerous threads per process
- Modern OS include a full pre-emptive OS



This can be done over a network too

Hardware Overview & processes

a)Hardware

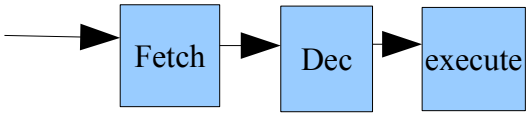
CPU

- Registers
- Program Counters
  - Points to next instructions to execute
- Stack pointer
  - Procedure calls
  - Points to stop of the stack

Pipelines/super scalar CPU

- *"nightmare for low level OS designers"*

- enhancing the fetch-decode-execute cycle



On each clock cycle one command is being fetched, another decoded & another executed improving efficiency

Superscale

- out of order execution

b)I/O

Controller

- Interface to the OS

Physical device

Controllers are standardised for disks (i.e. IDE)

Device Drivers

- Software in the OS for communicating with the controller

c)Communication with I/O

busy-waiting

- halts processes until I/O has completed
- Not very efficient

interrupts

- Used widely until recently
- Interrupt ~ handler (id) to each device
- Interrupts can have priorities
- Downsides not enough handlers (on hardware) for the number devices

Direct Memory Access

- Takes I/O off the CPU
- Handles interrupts
- To the CPU, I/O devices are just seen as memory address which the DMA manages

d)Processes & threads

Programs & process : process is code running

Multiprogramming : to the user ~ *"all processes run at the same time"*

- But limited resources
- OS Scheduler coordinates what runs when
  - Typically I/O calls cause a switch

Switch

- Save state of execution
  - CPU
  - Stack pointer
- Load a new one

Process Abstraction

a)Process
Single sequential stream of execution
Own address space in memory
I/O state

Processes are sequential – no concurrency within them

b)Process life cycle
Creation:
- Initialisation
- System call
- User Request
- Batch job

Creation adds a process to the process table and adds a reference to it in the queue of the scheduler

Termination:
- Normal exit
- Error Exit
- Fatal Error
- Killed

c)Creating a new process
Construct new PCB
Page tables for address space (to put process in)
Copy data from parent process (if its a child process)
Copy I/O state of parent (file handles etc....)

Can be speeded up (as in UNIX) by only copying when writing is needed, reading can be done from the same memory addresses.

d)Process protection
Early OS such as Mac and Windows 3.1-98 had no process protection and it was easy for a program with a bug to overwrite a memory address used by the OS or another process

- Assign each process a group of memory addresses (a space) for it to exist in
- In modern OS this is done using virtual memory addresses managed by a Memory Management Unit in hardware
  - MMU translates virtual addresses into physical addresses

This helps protect processes from each other as the MMU should know when one stops and another begins in the global memory space

e)Threads v processes

Processes can be made more concurrent by having several threads of execution

- Threads of execution: a program being executed in some address space

Threads *SHARE* the same address space as their parent!

Therefore threads are more lightweight than processes, however each thread still needs a seperate stack!!

Each thread has a Thread Control Block (TCB)

- Registers
- Scheduling info (state, priority)
- Accounting info
- Pointer to enclosing process? (sometimes)

h)Implementing Threads

User mode:

- Threads created virtually in the processes which manages its threads internally
- 'Software library'
- Falls down when the CPU cant see the threads within the process and can't manage their I/O requests correctly, unnecessarily blocking the whole process

Kernel Mode:

- OS handles threads

Hardware (hyper threading):

- Hardware is able to manage threads itself, usually fastest but can get in the way of the OS

Threads & Context Switching

a)Managing TCB

TCB contains:

- Stack, state registers etc... for each thread

OS tracks & stores TCBs in protected memory

- Every non-running TCB is in exactly one queue

b)Memory Footprinting

Example:

One process – 2 threads

- One address space
- 2 stacks & 2 sets of CPU registers

OS Designers have to consider:

- How to place stacks in memory
- Max size of stacks
- How to stop threads from over-writing each other
  - This is difficult
  - Best done by the process itself (software libraries)

c)Context Switching

"Loading and running a new thread"

Yielding:

- Program calls yield()
- This calls code from the kernel for yielding
- OS selects a new thread to run
- Switching happens (see below)
- House keeping of threads
- Return to thread (go up the stack)

Basic steps of switching

- Save status of the current execution (from CPU) in TCB and store in memory
- Load details of the next thread into the CPU
- Set next instruction of the original thread to be instruction AFTER switching (in this case house keeping)
- Set the next-instruction to run as the next instruction of the new thread to run
  - This sets the CPU to execute the thread

Code for this is low down in the kernel of an operating system and always written in assembly as this is fastest and also because higher level languages can not access registers directly in hardware!!

d)Returning to a thread

- Load values onto CPU from TCB
- Execute next instruction

next instruction will point to the instruction called by the kernel AFTER the context switch command. This has to be explicitly set in the switching process.

e)Different types of context switching
see slides

Concurrency

a)How can processes communicate?
Interprocess communication (IPC)
Seperate address spaces for each process – no overlapping

- Explicit shared-memory between process
  - i.e. one process writes to a file while another reads
  - Applies to threads too
- Message passing (works across a network)

b)Race Conditions
Processes racing for a shared resource, thinking they both have control of it and making conflicting alterations to it.
Context switch can happen at any time making strange things happen to the resource – testing wont always catch it!

c)Critical Regions
Parts of a program where shared resources are accessed
Mutual exclusion : if a thread is accessing a resource then no other threads are allowed to access that resource
Eliminates race conditions

Thread Control Methods

a)Peterson's algorithm
Variations of this have become the norm for thread control. Implantations optimise themselves so they're more useful over multiple threads.

The algorithm contains a flag for each thread which controls whether that process wants to access the critical region. It also contains an identifier to the thread being executed. All threads are blocked while a thread is in its critical region and the 'thread to be executed' is not them.

**see pseudo code on wikipedia*

b)Hardware Support
Home-cooked solutions are complex & break with modern hardware
- i.e. out of order execution
- disabling interrupts fails for multi-core machines

Atomic operations provided by hardware is a way around this
- Test-set-lock TSL instruction: TSL RX, LOCK
  - Reads value of memory location LOCK
  - Writes value in RX
  - Stores non-zero value in LOCK
- Used to get a comparison value
  - This is compared with 0 to see if the lock is on or off
  - 0 means that thread can enter critical region
  - Otherwise cause a blocking loop

Eliminates race conditions as actions are atomic
However its a busy-waiting solution which wastes CPU time as scheduler in OS doesn't know that a thread is blocked

c)Avoiding busy-waiting
Semaphores
- Mutexes
- Not used so much anymore

Monitors ← better
Sleep and Wake

d)Semaphores
Non-negative integer variable managed by the OS via 2 system calls
- Up +1 to semaphore
- Down -1 from semaphore
- Value only directly set at initialisation

OS makes sure changes are atomic
The semaphore moved down on enter region, up on leave region
- A bit like a barrier to a carpark

A mutex is a semaphore with only two states – up or down
Thread yields when it tries to access protected area thus avoiding busy-waiting
Yielding doesn't block thread forever, just for a time
Needs a semaphore for every protected area

Monitors

a)Murphy's law
If something can go wrong, given enough time its guaranteed to go wrong

b)Deadlocks
When threads are waiting for each other so nothing happens
- Too british

c)Limitations of semaphores
Two uses:
- Mutex
- Scheduling

Less control for the programmers

d)Monitors
Programming paradigm that separates the two concerns
- Mutex for mutual exclusion
- Condition variables for managing interleaving of threads

Essentially monitors are locks with a queue of threads internally which are asleep when the lock is taken and then race for the lock when its available

In Java
- synchronised methods
  - Can also be done with lock(...)
  - Ensures only threads with lock get to execute that code
- wait()
  - Needs to be in a while loo
  - Keeps thread asleep/in monitor queue
  - Gives lock back
- Notify()
  - Wakes up one of the threads in the monitor queue to start executing
- NotifyAll()
- Wakes up all of the threads in the monitor queue starting a race for the mutex lock

Scheduling

a)Issues

Fair about users
- proportion of cpu time per user

Fair about processes
- proportion of cpu time per process

Performance
- Maximal through put (jobs per hour)
  - Use resources effectively
  - Mainframe?
- Maximal response time
  - Actions response quickly
  - User input
  - All processes get to executed
  - Lots of context-switching
    - Overheads!!

Goal
- Distributing CPU time to optimise desired parameters

b)Bursts

Program typically
- Use CPU for some short period of time
- Then I/O
- Computer bound v I/O boundaries
  - time processes spends on each
  - usually I/O bound affects the system more as it takes more time

General idea: weight towards 'short burst'
- gives in precision of responsiveness

c)Scheduling Algorithms

FIFO – originally meant one program ran until done
- now runs until thread calls I/O when its taken off the scheduling queue and put on I/O queue
- good for through put, minimal switching
- long jobs block cpu, lack of responsiveness

Round Robin
- Processes in queue run for one 'time-quantum' each in a round robin
- Process wait for no longer than (n-1)q before executing
- Performance:
  - q too large → FIFO
  - q too small → overload of switching
  - Picking a good quantum time is crucial
- Good for short jobs
  - Executes them quickly
- Many context switches on long jobs mean they take longer
- Very had for lots of equally sizes processes great than q

Shortest job first & shortest remaining time left first

- SJF gives best average turnaround time for set processes
- Can be unresponsive though

More Scheduling

a)Shortest remaining time first

Pre-emptive version of SJF, if new shorter process arrives do it immediately
Instead of total time – use time of next cpu burst

- i.e. block a process when it goes to io

Helps I/O bound programs
Runs for time quantum then switches to next shortest process
Lots of short jobs can starve long processes of the CPU

b)Priority Scheduling

Assign each process a priority
Take process with lowest priority first
Scheduling can dynamically change priorities to avoid high-priority processes hogging the cpu
Priority class

- Two or more processes with the same priority

c)Multi-level Feedback Priority Scheduling

Queue of queues of jobs that need to be done
Each queue can have its own scheduling
Each queue has a priority in the main queue and everything in them has the same priority
Feedback: - moving processes from queue to queue depending on last execution, time to process etc....

d)Lottery

Scheduler gives each process a number of chances to execute depending on its priority
Process can then divide the 'tickets' among its threads
Although a process is never guaranteed to win, high priority processes are more likely to get executed than low priority processes and this system gives good responsiveness
If a process is unlucky it might not execute unless priorities are changed

Address Space

a)Protection
User processes need to be protected from hardware and each other
Memory abstraction: “virtual addresses”
Managed by OS
OS can protect processes from each other
Same memory can be shared for inter-process communication
ABSTRACTION

b)Early Memory Protection
IBM 360
Memory divided into 2KB blocks
Each block given a 4bit key
Keys then assigned to processes giving them access to that memory
OS manages keys
- checks process has the key for the memory its trying to access

Problems:
- Addresses not relative to process location in memory
- Address from programs end up referring to real physical addresses NOT GOOD

Static relocation:
- Making memory addresses (in the program) relative by adding an offset when the program is loaded

c)Virtual Addresses
Each process references a virtual address
OS & Hardware translates this to physical addresses
- AT RUNTIME

Dynamic relocation

d)Base/Limit registers
Limit → sets the amount of memory a program has
Base → start address of process in memory
Gives a window of operations
- if memory call is above limit then give an error
- otherwise add base to the memory address (so its relative) and perform operation

Done at runtime
Registers need to be stored on context-switches (like stack pointers etc...) but can be stored on the CPU

e)Swapping
Swapping processes in and out of memory via the disk
Useful for multiprogramming
Leaves fragmented gaps and no obvious solution to shared inter-process memory

f)Virtual Memory
Virtual addresses are translated into possibly non-continuous blocks of physical memory
- pages

Not all virtual addresses need to be in the physical memory at the same time
- use of the disk for swapping

Can be swapped in and out as needed

g)MMU
CPU now handles only virtual addresses
Translation handled by Memory Management Unit (usually on the cpu chip but not part of it)
Frees up CPU

More a Virtual Memory

a)Address Translation

Virtual Address divided into 2 parts

- Page number 4 bits
- Offset (address within pages) 12bits

MMU translates this into a physical address

- If page is in memory : gets page frame number for it
- Otherwise traps to OS asking it to load the page

see pictures from slides

b)Page Faults

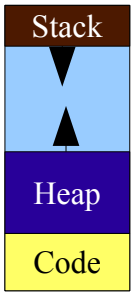
What if a page isn't in memory?
MMU triggers a system call (page fault)
OS picks some frame to replace

- Does this using some algorithm
  - Like scheduling
  - FIFO
  - Last used
- If page a modified – update the page on disk

Brings in new page from disk

Segmentation – a nice idea thats not actually used

a)Address Space
Space in memory given to a process
Finite space
- This could lead to parts of the process overwriting each other
  - i.e. stack overwriting the heap
  - Inefficient



b)Segments
Dividing a process into segments where each is represented as a virtual set of addresses
- i.e. one per stack, code, heap etc..

Can dynamically change in size but don't have to worry about overwriting as they're separate chunks of memory as they're not in the same place

Physical addresses calculated using a lookup table
- Number of the segment referenced sometimes send separately from address
- Or higher order bits of the virtual address used
- Translated into a physical address

Pure segmentation – implementing each segment as a BASE/LIMIT address space

c)Segmentation with paging
Splitting segments into pages so each segment has its own page table

Lots of overhead
- In software this would be very slow
- Requires a lot of hardware support

Context switch
- have to change the segmentation table to point to the new set of pages for the new process

Since processes can share pages they can share segments using this method

Unfortunately the only operating system with this was OS/2 which failed.

Input/Output

a)I/O hardware rates
Massive difference between speeds of different I/O devices and the cpu

b)Controllers
Most I/O devices have two components
- mechanical
- electrical

Electrical component is the device controller
- sometimes can handle multiple devices (like RAID controllers)

Controller's tasks
- convert serial bit stream into blocks of bytes
- error correction
- interface with buses

c)Polled I/O
idea: having a bit that flips when I/O device is read for servicing
used in pic chips a lot
simple system which services devices quickly

Disadvantages:
- takes up cpu time and blocks other processes
- lots of pins for lots of devices

e)Interrupts
When I/O device is ready, sends interrupt signal to CPU which will execute a subroutine
Many issues
- priorities
- race conditions
- sub routines within subroutines
- needs to be done in kernal mode
- time taken from interrupt issued to servicing

advantages:
- asynchronous
- cpu can continue processing after I/O call

disadvantages
- complex
- lots of complicated logic

f)Direct Memory Access
Memory Mapped I/O
DMA is a seperate unit away from the CPU to deal specifically with I/O
- saves cpu having to do lots of context switching as with interrupts

Still have interrupts to the CPU but this is only to say that the I/O operation it was given has been completed
DMA handles all the I/O requests itself and frees up the cpu
Instead the CPU can continue with operations within its cache or it will just halt/pause for the time that the DMA is writing its bytes to memory

DMA can write different sized blocks to memory in slices so large requests don't halt the cpu too much

e)Drivers
OS provices a uniform interface to the user about how to communicate with their hardware
Device drivers provide a way for the OS to communicate to the hardware as each driver is hardware specifically

f)Device Abstraction
Character Device
- stream of data that can be read or written to
- i.e. the mouse

Block Device
- information accessed in fixed size blocks
- Each block has its own address
- Allows random access

Filesystems

a)Low level formatting

Tracks
- set of bytes at a certain radius from the center

Sector
- a portion of a track

Block
- size written or read from the disk at any one time
- made up of sectors
- might not fill a track

Zoning
- sectors are equal number of bytes and are denser at the middle of the disk

Interleaving
- placing sectors of bytes interleaved along the track so that the disc will spin to the new segment during the time that data is set from the first block to main memory

Filesystems continued

a)What size should the block have?
Too small

- Lots of entries in house keeping
- External fragmentation
  - small gaps between sets of blocks too small for files to fit in
  - file has to be split up over the disk so access time is high

b)File system performance
Access time
Transfer speed
OS can't trust the disk as these stats are usually faked for marketting

access time = seek time + rotation latency + data transfer time
Usually seek time is the slowest part of the process