Part IA Operating Systems Notes

1 Processor

1.1 Introduction

- Big-Endian numbers are stored with their most-significant-bit at the lowest memory address, little-endian with their least-significant-bit
- Computers logically takes values from memory, performs operations and then stores results back
  - CPU operates on registers, extremely fast pieces of on chip memory
  - Data loaded into registers before being operated on
- The fetch-execute cycle - CPU fetches and decodes instruction, generates control signals and operand information
  - In the Execution Unit control signals select a Functional Unit and operation
  - If Arithmetic Logic Unit chosen then reads registers, performs an operation and writes back
  - If Branch Unit, tests condition and possibly adds value to Program Counter
  - If Memory Access Unit, generate address and use bus to read/write value
  - Repeat
- Buses - Collection of shared communication wires
  - Address lines, data lines and control lines
  - Operates in master-slave manner, master initiates process, slave follows suit
  - Different types of bus
    - Processor Bus - Faster, for CPU to Cache
    - Memory Bus - To communicate with memory
    - PCI - To communicate with devices
    - Bridge - Connects two different buses
- Direct Memory Access (DMA)
  - Device can read and write to memory directly
  - Gives just one interrupt at the end of data transfer
- Layering - Arrange components into a stack, each can only communicate with layer above and below
- Multiplexing - When one resource is being consumed by multiple resources simultaneously
- Latency - How long something takes
- Bandwidth - The rate at which something occurs
- Jitter - The variation (statistical dispersal) in latency
- Impedance Mismatch - Where two components are operating at different speeds, can be handled by
  - Caching - Where a small amount of higher-performance storage is used to mask a larger lower-performance component
  - Buffering - Where memory is introduced between two components to soak up small variable imbalances in bandwidth

- Bottleneck - The one resource in a system that is the most constrained
- Tuning - Determining and eliminating bottlenecks
- Optimising the common case gets the best results
  - The 80/20 rule, 80% of the time is spent in 20% of the code
- **Operating System** - A program controlling the execution of all other programs. Objectives: convenience, efficiency and extensibility

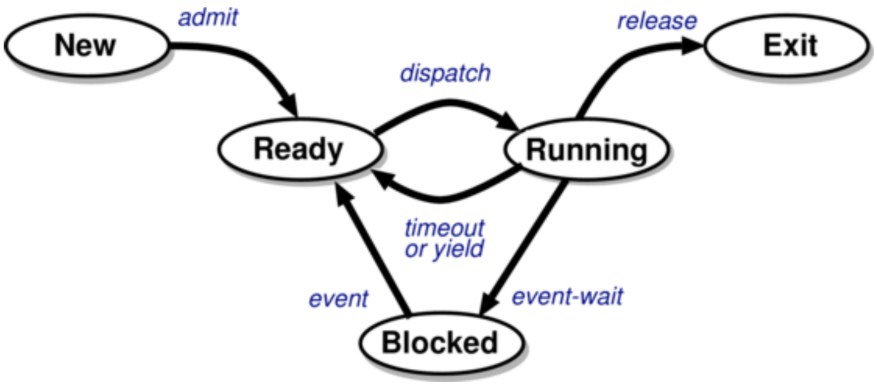
1.2 Protection

- Impose controls on access by **subjects** (e.g. users) to **objects** (e.g. files)
- A covert channel is some illegal method of information transfer, e.g. through garbage collector
- Protecting I/O
  - Make all I/O instructions privileged
  - Applications can't mask interrupts - but applications can rewrite interrupt vectors
  - Applications can't control I/O devices - but some devices accessed via memory
  - To control I/O must control memory access
- Memory Protection - Paging and segmentation
- CPU Protection - Prevent process hogging CPU, scheduling
- Kernel-Based Operating System
  - Applications can't do I/O due to protection, OS does it instead
  - Kernel provides an unprivileged instruction to transition to kernel mode, called a trap or software interrupt
  - OS services accessible via software interrupts called system calls
- Microkernel Operating System
  - OS services moved into privileged servers
  - Servers accessed for OS services, accessed using Inter-Process Communication (IPC)
  - IPC adds overhead, implementation tricky and often has redundant copies of OS data structures
- Dual-mode operation - Two modes of execution, kernel mode: for the OS, and user mode: for the user
- Mandatory Access Control - Specifies that level of access each subject has to each object, e.g. users to files
- The `PLEDGE(2)` system call allows a programmer to indicate explicitly which class of system call the wish to use at any point
- Authentication
  - Passwords currently used to authenticate user to system
    - People choose badly
    - Passwords hashed
    - Prefer key-based systems
  - Very difficult to authenticate system to user
- Mutual Suspicion

  - Encourage suspicion - system of user, users of each other, user of system etc.
  - Trojan horse - Inherits user's privileges when called
- Access Matrix - A matrix of subjects against objects, sparse so not all stored. Common representations
  - Access Control List - Stored by object - A list of subjects and their rights stored with each object
    * Often used in storage systems
    * Stored with file
    * ACL checked when file opened for read or write, or when code file is to be executed
  - Capabilities - Stored by subject - A list of objects and their rights stored with each subject
    * Store in subject's address space
    * Ensure subject can't forge capabilities
    * Easily accessible to hardware
    * Can be done in software (checked by encryption) or hardware (capabilities modified by special instructions; passed on program call)

1.3 Processes

- A process is a program in execution
  - A unit of protection and resource allocation
  - Each executed on a virtual processor, with its own virtual address space
  - Has one or more threads, each with a program counter, stack, and data section
- Process States - A process moves between a number of different states



  - New - Being created
  - Running - Instructions being executed
  - Ready - Waiting for the CPU (ready to run)
  - Blocked - Stopped, waiting for an event to occur
  - Exit - Has finished execution
- Process Creation - Nearly all systems are hierarchical: parent processes create child processes
  - Resource Sharing - Child shares none, some or all of parent's resources
  - Execution - Parent can either run concurrently with children or wait for their termination
  - Address Space - Child either duplicate of parent or has program loaded into it, e.g.

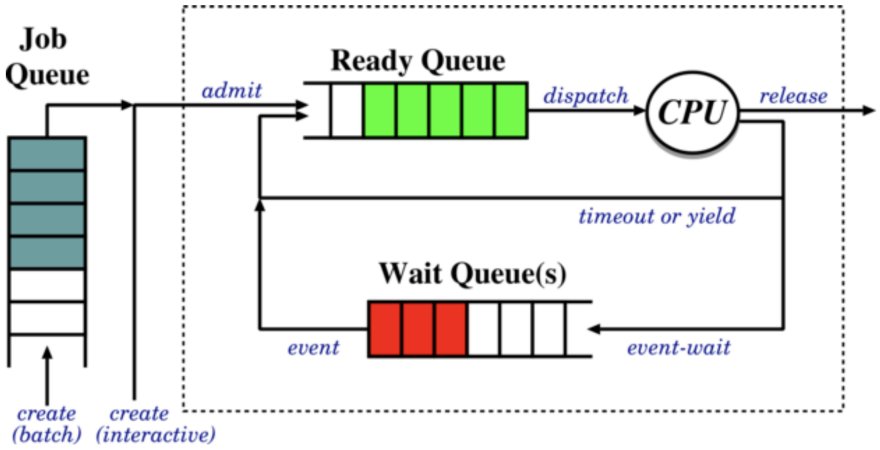
    - `fork()` - clones parent
    - `execve()` - replace process' memory space with a new program

- Process Termination - Termination occur in three circumstances
  - Process executes last statement and asks OS to delete it, resources deallocated
  - Process performs illegal operation, e.g. unauthorised memory access, attempts to execute privileged instruction
- Parent may kill child process, e.g. child exceeds allocated resources, child's task no longer required, or parent exiting ("cascading termination")
- Blocking - A process blocks on an event, e.g. an IO device completes an operation or another process sending a message
- Processes can be either
  - IO-bound - more time doing IO than computation, many short CPU bursts
  - CPU-bound - more time doing computation, few very long CPU bursts
- Most processes execute for at most a few milliseconds before blocking
- The OS maintains a data structure called a **process control block** (PCB), storing information about each process
  - Within this is the **Process Context**, the machine environment during the time the process is on the CPU
  - Process context contains: Program counter, General purpose registers, Processor status register, Caches, TLBs, Page tables ...
- Context Switching
  - The OS must:
    - Save the context of currently executing process
    - Restore the context of resuming process
  - Context switching is wasted time, no useful work done
  - Can be sped up with hardware support
- A thread represents an individual execution context
  - Each with associated **Thread Control Block** (TCB) with metadata about the thread
  - Managed by a scheduler
- Inter-Process Communication - Allows processes to share data
  - **Signals**
    - An asynchronous notification send from one process to another, kernel interrupts the process to deliver the signal
    - A range of different signals
  - **Pipes**
    - Between child and parent process
    - Returns a pair of file descriptors, an array with one a read file descriptor, one a write
    - Parent gets file descriptors, forks and then both can communicate via pipe
  - **Named Pipes**
    - Same as a pipe but has a name, not just two file descriptors

    - Don't have to be a parent and child process
    - Can just be treated as a file
  - **Shared Memory Segments**
    - Obtain a segment in memory that is shared between two (or more) processes
    - Read and write to via pointers
    - Must impose concurrency controls
  - **Files**
    - Multiple processes can share a file in memory
    - Locking - Removes the lock on a file and then lets other processes use it
    - Memory Mapped Files - Maps a file into memory so you can interact with it via a pointer
  - **Unix Domain Sockets**
    - Like network sockets, but use shared memory for interaction between local processes
    - Create a socket, bind to it, pair with another process and creates a full-duplex pipe

1.4 Scheduling

- Processes move between three queues in their lifetimes



  - Job Queue - Batch processes awaiting admission
  - Ready Queue - Processes in main memory, ready and waiting to execute
  - Wait Queue - Set of processes waiting for an IO device

- Non-Preemptive Scheduling
  - Will only change processes when one blocks, terminates or yields the CPU
  - Easy to implement, but open to denial of service
- Preemptive Scheduling
  - Will change a process if a higher priority process unblocks or a timer expires
  - Harder to implement, but solves denial of service problem
- Processes tend to spend more the 50% of the time idling, can do one of three things
  - Busy wait in scheduler

    - Repeatedly checks ready queue, can respond straight away - quick response time
    - Ugly, useless, wastes energy
  - Halt processor until interrupt arrives
    - Saves power, increases processor lifetime
    - Might take too long to stop and start
  - Invent idle process, always available to run
    - Gives uniform structure
    - Can do house cleaning
    - Uses up some memory, might slow interrupt response

- Scheduling Criteria - A number of different criteria can be used to choose the next process to run
  - **CPU Utilisation** - Maximise the fraction of the time the CPU is actively being used
    - Keep machine as busy as possible
    - Penalised IO bound processes because they yield appearing that CPU not being used
  - **Throughput** - Maximise the number of processes that complete their execution per time unit
    - Get useful work completed at the highest possible rate
    - May penalise long-running processes as short-running processes will complete sooner, so preferred
  - **Turnaround Time** - Minimise the amount of time to execute a particular process
    - Ensures that every process completes in the shortest possible time
    - Compromises responsiveness
  - **Waiting Time** - Minimise the amount of time a process has been waiting in the read queue
    - Ensures an interactive system remains as responsive as possible
    - May penalise IO bound processes which spend a lot of time in wait queue
  - **Response Time** - Minimise the amount of time it takes from when a request was submitted until the first response is produced
    - Used in time sharing systems, ensures system remains responsive under load
    - May penalise longer running processes under heavy load

- Predicting Burst Lengths
  - Done using the length of previous bursts, and exponential averaging
  - The following formula is used,

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$$

    Where τ_i is the i^{th} predicted burst length, t_i the actual i^{th} burst length and τ_0 and α are constants

1.4.1 Scheduling Algorithms

- **First-Come First-Served**
  - Non-preemptive
  - Depends only on the order processes arrive in
  - Can experience the **convoy effect** later processes held up behind a long-running first process
- **Shortest Job First**
  - Non-preemptive
  - Associate with each process the length of its next CPU burst
  - Use lengths to schedule the process with the shortest time

  - Use FCFS to break ties

- **Shortest Remaining-Time First**
  - Preemptive
  - Preempt the running process if a new process arrives with CPU burst length less than the remaining time

- **Round Robin**
  - Preemptive scheduling algorithm for time-sharing systems
  - Define a small fixed until of time called a quantum, process at front of queue allocated the CPU for one quantum, when time elapses process preempted and appended to ready queue
  - It is fair, given n processes each has $\frac{1}{n}^{th}$ of the CPU time
  - No process waits for more than $(n-1)q$ time units to be allocated
  - But if q too large becomes FCFS, too small and context switch has too high an overhead

- **Priority Scheduling**
  - Associate a priority to each process, simplest can be system vs. user task
  - Can be preemptive or non-preemptive
  - Allocate the CPU to the highest priority process
  - Tie-breaking - Can do round robin
  - Low priority processes are not guaranteed to run

- **Dynamic Priority Scheduling**
  - Use *Priority Scheduling* but change priorities with time
  - Processes have base priorities and a dynamic effective priority
  - Every k seconds a process starved, increment effective priority. Once it runs, reset effective priority

2 Memory Management

2.1 Virtual Memory

- Relocation
  - Programmer can't know address that process will occupy at runtime
  - May want to swap in and out of memory, shouldn't have to be in the same place
  - Processes need to incorporate addressing info, e.g. pointers
  - Need to translate from logic to physical addresses

- Allocation
  - OS may choose to put processes in memory to make linking or relocation easier

- Protection
  - Protect one process from others
  - A process should not be able to modify its own code
  - Dynamically computer address (array subscripts) checked for sanity

- Sharing

  - Multiple processes sharing the same binary executable
  - Shipping data around between processes by passing shared data segment references
- Logical Organisation
  - Most physical memory linear address spaces, unlike programs, which are modular
  - Useful if OS could deal with modules, can be written and compiled independently
- Physical Organisation
  - Key OS function to organise flow between main memory and secondary storage
- Address binding problem
  - At compile time - requires knowledge of absolute addresses
  - At load time - Find position in memory on loading, update code with correct address
    - Must be done every-time code loaded
  - At run time - Hardware can automatically translate between program and real addresses
    - No changes required to program itself
    - Most popular and flexible scheme, providing we have the requisite hardware (MMU)
- Memory Management Unit
  - Relocation register holds value of the base address owned by the process
  - Relocation register contents added to each memory address before it is sent to memory
  - OS has privilege to update relocation register
- Allocation
  - Contiguous Allocation
    - OS must be in low memory due to location of interrupt vectors
    - Statically divide memory into multiple fixed sized partitions, with lowest partition being for the OS
    - When a process terminated partition becomes available for another
    - Must protect OS and user processes
  - Static Multiprogramming
    - Partition memory into when installing OS and allocate pieces to different job queues
    - Associate jobs to job queue according to size
    - Swap job back to disk when, blocked on IO or time sliced
    - Run job from another queue while swapping jobs
    - Problem of fragmentation and inability to grow partition size
  - Dynamic Partitioning
    - OS keeps track of which areas of memory are available
    - For a new process, OS searches of a hole large enough
      - First fit - use first hole that fits
      - Best fit - search all of list and find best fit
      - Worst fit - search all of list and find allocate largest hole
- External Fragmentation
  - Loaded processes leave little fragments which may not be used

  - External fragmentation exists when the total free memory is sufficient for a request but is unusable as it is split

- Compaction
  - Can be tricky
    * Require runtime relocation
    * Can be done more efficiently when processes move in and out of memory or some have hardware support

2.2 Paging

- A way of solving the address binding problem is **Paging**
  - Divide physical memory into frames, small fixed-size blocks
  - Divide logical memory into pages, blocks of the same size
  - CPU generated addresses have a page number and an offset
  - Page table associates page numbers with frame numbers
  - Far more pages then frames
  - Each process has its own page table
  - Pages usually 4kB

- Pros and Cons
  - Hardware support required, masking used so page size and number a power of two
  - Memory allocation easier, no external fragmentation, but internal fragmentation (all of final page may not be entirely used)
  - Significant per-page overhead, from page tables and disk IO more efficient for larger pages
  - Clear separation between process and OS view of memory - process sees single logical address space, OS does hard work
  - Process can not refer to memory it does not own, OS can map system resources into user address space, e.g. IO buffer
  - Adds overhead to context switch, page table must be swapped out

Large Page Size	More Fragmentation	Low Overhead
Small Page Size	Less Fragmentation	High Overhead

- Page Tables
  - Simple but small - set of dedicated relocation registers, one register per page - Not used
  - Complex but usable - Keep page table in memory and have one MMU register, the **Page Table Base Register (RTBR)**, switched in context switch
  - Page table can be very large, keep **Page Table Length Register (PTLR)**

- Translation Lookaside Buffer (TLB)
  - This is a cache, for the page table
  - Test if page table entry is in TLB, if not get from memory and put in TLB
  - If full discard entries, least recently used
  - Context switch flush TLB, to prevent next process using wrong page table entries
  - Can tag each page table entry with its process and not flush buffer
  - Hit ratio, proportion of time page table entry is found in TLB

  - Effective Memory Access Time is a weighted average of the TLB lookup time and page table lookup time

- Modern systems require very large page tables, use multilevel page tables, e.g. split address into three, first part indicates which level 2 page table, second which page, third offset
- The more levels of page tables, the more memory accesses and higher overhead
- Page tables and page directories (tables of page tables) often one page in size
- Each page table entry can have **protection bits**, for read, write, execute, and whether it can only be accessed in kernel mode
- Pages can be shared, different logical addresses can be mapped to the same physical addresses
  - If code re-entrant (i.e. stateless and non-self-modifying) can easily be shared
  - Otherwise can use copy-on write
    1. Mark page as read-only for all processes
    2. If a process tries to write to page, will trap OS fault handler
    3. Allocate new frame, copy data, and create new page table mapping

2.2.1 Virtual Memory - Demand Paging

- Pages can be non-resident and put on a non-volatile backing store
- Processes access non-resident memory just as if it were the real thing
- Advantages
  - Portability - Programs work regardless of how much actual memory is present
  - Convenience - Programmer can use very large data structures
  - Efficiency - No need to waste memory on code or data which isn't used
- Programs reside on disk, pages loaded in on demand
- In practice
  - On loading process
    1. Create its address space
    2. Mark page table entries as either invalid or non-resident
    3. Add PCB to scheduler
  - When receiving a page fault
    1. If due to invalid reference, kill process
    2. Otherwise due to non-resident page
       (a) Find free frame in memory
           i. If there is a free frame use it, otherwise select victim page
           ii. The write victim page back to memory
           iii. Finally mark it as invalid in page table
       (b) Initiate disk IO to read in desired page
       (c) When IO is finished modify page table entry to show it is valid
       (d) Restart process
- Problems with Demand Paging
  - Process state must be correctly saved at time of fault

  - Instruction pipeline (a form of single instruction level concurrency) may have to be wound back
  - In a CISC system fault can happen half way through instruction, may need temporary registers for such instructions
  - Problems in auto-increment/ decrement instructions
  - Can have multiple faults per instruction
- Some systems preload core parts of the process
- Add a dirty bit to page table entries, set if page altered
  - Don't have to write back to memory if dirty bit not set or if read-only
- A reference bit or modified bit can be simulated by:
  - Clear reference bit by marking page no access
  - If referenced then trap, update page table entry and resume
  - Check if reference by checking permisions
- Page Replacement Algorithms are used to choose the victim page

2.2.2 Page Replacement Algorithms

- **First-in First-out**
  - Keep a queue of pages, discard from head
  - Independent of page use frequency
  - Can discard page in current use, causing immediate fault
  - Increasing number of frames can increase number of fault (Bélády's anomaly)
- **Optimal Algorithm**
  - Replace page which won't be used again for longest period of time
  - A baseline for judging other algorithms - can only be done in hindsight
- **Least Recently Used**
  - Replace the page that hasn't been used for the longest amount of time
  - Considered quite a good algorithm
  - Can be implemented using:
    * Counters - give each page table entry a time-of-use field and the CPU a logical counter
      · When a page is reference update the page table entry's time-of-use field
      · Replace the page with the smallest time value
      · But requires search to find minimum, adds a write to memory on every memory reference, must handle clock overflow
      · Impractical of a standard processor
    * Page Stack - Maintain a stack of pages (doubly linked list) with most-recently used on top
      · Discard from bottom of stack
      · Requires up to 6 pointers to be changed per reference (max when new reference)
      · Slow without extensive hardware support
    * Approximating LRU - Add a reference bit to page table entry, initially zeroed and set by hardware whenever page referenced
      · Implementing not recently used

- Periodically clear all reference bits
- When choosing a victim page, prefer pages with clear reference bits
- With a dirty bit

Referenced	Dirty	Comment
No	No	Best to replaced
No	Yes	Next best (requires write back)
Yes	No	Probably code in use
Yes	Yes	Bad choice

- Could instead have an 8-bit value per page, initialise to zero and periodically shift. Replace page with smallest value

- **Second-Chance FIFO**
  - Stores pages in a queue, before discarding head check reference bit
  - If reference bit is 0 then discard, else reset reference bit and add it to the tail
  - Can be implemented using a circular queue
- **Counting Algorithms** - Keep counter of the number of references to each page
  - **Least Frequently Used** - Replace page with smallest count
    - Need to periodic counts
  - **Most Frequently Used** - Replace page with highest count
    - Low count indicates recently brought in
- **Page Buffering**
  - Have a minimum number of free spaces, write in and then write back victim page
  - Means process can restart quickly
  - If disk idle write modified pages out and reset dirty bits

2.2.3 Frame Allocation

- System depends on what you want to achieve
  - Fairness
    - Share out equally, any remaining in a free pool
    - Divide frame in proportion to the size of each process
  - Minimise system wide page fault rate
    - Allocate all memory to few processes
  - Maximise level of multiprogramming
    - Allocate minimum amount of memory to a lot of processes
- Could allocate based on process priorities
- Global schemes can choose victims from any process
- Local schemes can choose victims from own process
  - Local schemes allow each process to control its fault rate, instead of other processes
  - Global is optimum for throughput and therefore most commonly used
- **Thrashing**
  - More processes entering system causes frames-per-process allocated to reduce

  - Eventually processes steal frames from others only to have them stolen back
  - Number of runnable processes plunges
  - OS monitors may see CPU usage down and increase level of multiprogramming, machine dies
  - Avoided by giving processes as many frames as they need and if not then suspend them
- Locality of reference - In a short time interval locations of referenced by a process tend to be grouped into a few regions

2.2.4 Working Set

- The set of pages that a process needs in store at "the same time" to make any progress
- Varies between processes and during execution, assume process moves through phases each with locality of reference
- Calculated by
  - Periodically sample reference bits
  - Define a window of size Δ of most recently used pages
  - If a page is in use then say it is in the working set
  - Sum of working set sizes of all processes gives demand D
  - If $D > m$ in danger of thrashing - suspend process
- The working set can be remembered and prepaged when a process is starts, reduces the number of page faults when a process starts
- Page Sizes

Small Pages	Large Pages
Large page table	Small page table
Low fragmentation	High fragmentation
More page faults	Fewer page faults

- Page faults costly, have to context switch into kernel

2.3 Segmentation

- View memory as a set of segments of no particular size or ordering
- Length of segment depends on complexity of function
- Segments have a number and a length
- Logical addresses specify segment and offset within segment, in a tuple e.g. `(segment, offset)`
- User is aware of memory structure
- May have segments for global variables, call stack, for each function etc.
- Loader takes each segment and maps it to a physical segment numbers
- Maintain a **Segment Table**
  - Kept in memory, pointed to by **Segment Table Base Register**
  - Also have a **Segment Table Length Register**
  - Changed as part of a context switch

- Each memory reference the following algorithm is executed (note requires 2 memory accesses)
  1. Program presents address $(s,\ d)$
  2. If $s \geq STLR$ then not valid
  3. Obtain table entry at reference $s + STBR$, a tuple $(b_s,\ l_s)$, with b_s the segment's base and l_S the segment's length
  4. If $0 \geq d \geq l_s$ then this is a valid address at location $(b_s,\ d)$, else fault
- Protection is provided per segment
  - **Protection bits** associated with each segment table entry and checked in the usual way
  - Instruction segments should not be self modifying
- Sharing
  - Each process has its own segment table
  - Sharing enabled when two processes have a mapping for the same physical location
  - Usually copy-on-write
  - Problem, jumps must provide destination address (where to jump to)
    - Processes could have different segment numbers for the same piece of code
    - Solved by specifying branches as program counter relative or relative to some register containing the current segment number
- An alternative sharing strategy
  - Have a system wide segment table, the **System Segment Table**
  - Assign each segment a unique **System Segment Number**
  - Process segment tables map from process segment numbers to system segment number
- Because segments are variable sized external fragmentation can occur, use best/first fit algorithm
  - Small average segment sizes, external fragmentation is small

2.4 Segmentation vs. Paging

- For protection and sharing easier to have a logical entity - Segments
- For allocation and demand access prefer paging
- Combining segmenting and paging
  - Paged segments
    - Divide segments into k pages, where $k = \lceil (l_i/2^n) \rceil$, with l_i the length of the segment
    - One page table per segment
    - High hardware cost and complexity, not portable
  - Software segments
    - Consider pages [m, ..., m + l] to be a segment
    - OS ensures protection and sharing kept consistent over region
    - Loss of granularity but simple and portable
- **Paging used** because
  - Allocation is easier
  - Cost of sharing/demand loading is minimised

  - More portable, can do software segments with paged hardware

- Implementation Considerations
  - Hardware support
  - Performance - Complex algorithms need more lookups per references, simpler preferred
  - Fragmentation
  - Relocation - Solves external fragmentation at a high cost, forces logical addresses to be computed dynamically
  - Swapping - Can be added to any algorithm, allowing more processes to access main memory
  - Sharing - Increases multiprogramming but requires paging or segmentation
  - Protection - Useful, necessary to share code/data, needs a couple of bits

3 Input/ Output

3.1 IO Subsystem

- Messiest part of OS
- Split into four device classes
  - **Block devices** e.g. disk devices, CD
    - Commands include `read`, `write`, and `seek`
    - Can have raw access or via filesystem (cooked) or memory mapped
  - **Character devices** e.g. keyboards, mice, serial
    - Commands `get` and `put`
    - Just a stream of characters, layer libraries on top for line editing
  - **Network devices**
    - Vary enough from block and character devices to have own interface
    - Deal with speed and types of failure
    - Often use Berkeley Socket Interface
  - **Miscellaneous**
    - e.g. current time, elapsed time, clock
    - In Unix use ioctl to cover other odd aspects of IO
- Programs access virtual devices, the OS handles the processor-device interface, these then implemented
  - In Kernel, e.g. files, terminal windows
  - In daemons, e.g. spooler, windowing
  - In libraries, e.g. terminal screen control, sockets
- Daemons are long running processes
- Polled Mode IO
  - Consider a device with `status`, `data`, and `command` registers
  - Operation could be as follows
    1. Host repeatedly reads `device-busy` bit until clear
    2. Host sets command bit (e.g. `read` or `write` bits) and puts data into `data` register if necessary.
    3. Host sets `command-ready` bit in status register

4. Device sees `command-ready` set and sets `device-busy`
5. Device performs operation and writes to `data` register if necessary
6. Device clears `command-ready` and then clears `device-busy`

- Requires device and host to repeatedly poll device registers

- Interrupt Driven IO
  - Rather than polling, processors provide an interrupt mechanism to handle mismatch between CPU and device speeds
  - At the end of each instruction the processor checks interrupt lines for pending interrupts
  - If there is an interrupt then the processor context switches into Kernel mode and jumps to a known address
  - Once handler done can resume
  - Processor may have hardware vectoring for interrupts
  - Handling interrupts
    1. Implementation has two parts, the processor dependant interrupt handler and a per-device interrupt service
       * Save more registers and establish language environment
       * Demultiplex interrupt in software (decode) and invoke relevant interrupt service routine
    2. Interrupt service routine
       * Programmed IO - transfer data and clear interrupt
       * Direct memory access - acknowledge transfer; request any more pending; signal any waiting devices; clear interrupt and return

- **Blocking IO** - Process suspended until IO finished
  - Easy to use and understand
  - Insufficient for some needs
  - Control handed over to device

- **Nonblocking IO** - IO call returns as much as available
  - Returns almost immediately with count of bytes read or written (possibly 0)
  - Can be used by UI code
  - Essentially application level polled IO
  - You keep control

- **Asynchronous** - Process runs while IO executes
  - IO subsystem explicit signals when request completed
  - Most flexible (and potentially efficient)
  - Also mist complex to use

- OS can buffer data in memory to cope with impedance mismatches

- Buffering will not work if on average
  - Process demand > data rate
  - Data rate > capacity of system (buffer will fill and data will spill)

- Buffering Strategies
  - **Single Buffer**

    - OS assigns a single buffer to the user request
    - OS performs transfer moving buffer into userspace when complete
    - Make new buffer for more IO and reschedule application to consume it
    - OS tracks buffers
  - **Double Buffer**
    - Prevents need to suspend user process between IO operations
    - OS needs to manage buffers and processes to ensure it doesn't start consuming from an partially full buffer
  - **Circular Buffer**
    - Allows consumption from buffer at a fixed rate, potentially lower than the burst rate of the arriving data
    - Typically use circular linked list

- Other Issues
  - Caching - Fast memory hold copy of data for both reads and writes
  - Scheduling - Order IO requests in per-device queues
  - Spooling - Queue output for devices, useful if device is single user (can serve only one request at a time)
  - Device Reservations - System calls for acquiring or releasing exclusive access to a device
  - Error Handling - Will usually get some form of error code when requests fails logged into system error log

- The kernel must maintain state for IO components
  - Open file tables
  - Network connections
  - Character device states

- IO is a major factor in system performance
  - Reduce number of context switches
  - Reduce data copying
  - Reduce number of interrupts by using large transfers, polling
  - Use direct memory access where possible
  - Balance CPU, memory, bus and IO performance for highest throughput

3.2 Storage

- A file system is made up of two parts
  - **Directory Service** - Maps names to file identifiers and handles access and existance control
  - **Storage Service** - Provides a mechanism to store data on disk, including means to implement directory service

- A file is the basic abstraction for non-volatile storage
  - User abstraction
  - Typically a single contiguous logical address space
  - Can have varied internal structure

- OS split files into *text* and *binary*, where
  - Text - A sequence of lines each terminated by a special character and ending with a specific EOF character
  - Binary - A stream of bytes
- File have at least two names
  - **System file identifier** - A unique integer value associated with a given file, used by file system itself
  - **Human name** - What the user likes to see
  - **User file identifier** - May have if open, used to identify open files in applications
- Directories map human names to system file identifiers, non-volatile and therefore also stored on disk
- An open file is stored in memory as a **file control block**
- File typically have metadata e.g.
  - Location - Pointer to the file on device
  - Size
  - Type - Needed if the system supports different types of file
  - Protection
  - Time, date and user identification - For protection and usage monitoring
- A directory should be
  - Efficiency - Locate a file quickly
  - Naming - Allow two or more users to have the same file name or one file to have multiple names
  - Grouping - Allow for the logical grouping of files
- Directories have a DAG (directed acyclic graph) structure
  - Allows for multiple names per file, i.e. shared files and sub-directories and multiple aliases for the same thing
  - For each file keep a count of the number of references to that file
  - When a file is deleted decrement this counter, if is the counter is zero then the file can be removed from the disk
  - Must ensure that the file system is a DAG, otherwise can get into an infinite loop
- Directories are stored as files on disk
  - Must be a different type of file for traversal
  - Have explicit operations
    - Create/delete directories
    - List contents
    - Select current working directory
    - Insert a new file
- When a file is opened the directory service is recursively searched for the file given its name
- Eventually get system file identifier of file from which user file identifier is created and returned
- To close a file the file control block is copied back to disk and the user file identifier is invalidated

- Associate a cursor or file position with each open file, initialised to the start of the file, can read or write next n bytes from file
- Can have different access patterns
  - Sequential - Adds rewind to above
  - Direct Access - Can seek to place in file
  - Can have append-only etc.
- File owner should be able to control what can be done, by whom
  - Access control usually done by directory service
  - File usually only accessible if user has both directory and file access rights
- Garbage collect file system periodically
- Need some form of locking to handle simultaneous access
  - Can be mandatory or advisory
  - Locks may be shared or exclusive
  - Granularity may be file or subset

4 Unix Case Study

4.1 Design

4.1.1 Unix design features

- A hierarchical file system incorporating demountable volumes
- Compatible file, device and inter-process IO
- Ability to initiate asynchronous processes (`fork` command)
- System command language selectable on a per-user basis
- A separation between a kernel and everything else (first of its kind)
- Highly portable due to being written in a high level language

4.1.2 Structure

- Clear separation between user and kernel portions, only essential features inside the OS, not editors, command interprets, compilers, etc.
- Processes are unit of scheduling and protection
- The command interpreter (Shell) just a process
- No concurrency in kernel
- All IO looks like operations on files, in Unix everything is a file

4.2 File System

4.2.1 File Abstraction

- Files are an unstructured sequence of bytes
  - Don't get type information
  - Programmer has more flexibility
  - Less stuff to worry about in the kernel
- You have to get the kernel to do all IO
- Files represented by a file descriptor in user space, an opaque identifier
- File Operations
  - `fd = open(pathname, mode)`
  - `fd = creat(pathname, mode)`
  - `bytes = read(fd, buffer, nbytes)`
  - `count = write(fd, buffer, nbytes)`
  - `reply = seek(fd, offest, whence)`
  - `reply = close(fd)`
- `ioctl` used for special files

4.2.2 Directory Hierarchy

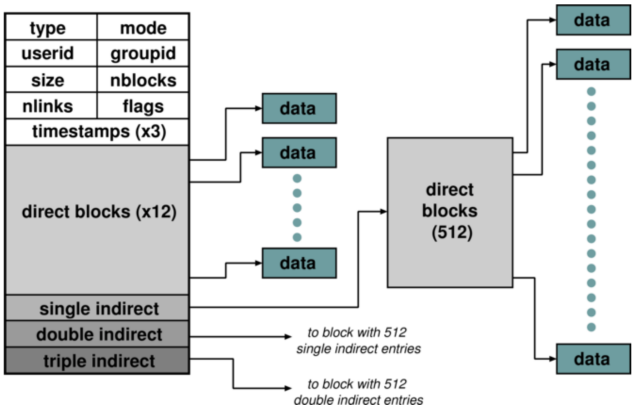
- Start with a distinguished root directory called `/`
- Fully qualified pathnames mean performing a traversal from root
- Each directory has entries `.` and `..`, referring to self and parent respectively
- Have a shortcut for current working directory
- Has a relaxed tree structure, close to a DAG

4.2.3 Password file

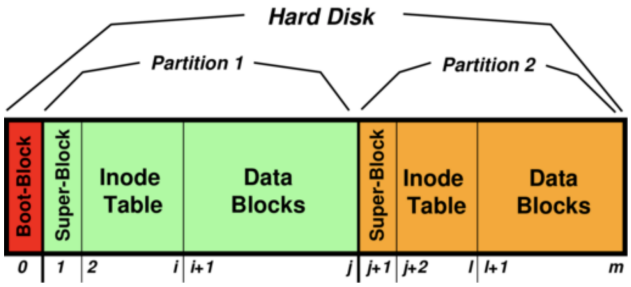
- `/etc/passwd` holds a life of password of the form `name:encrypted-password:home-directory:shell`
- Also contains user-id, group-id and a friendly name
- Uses a hash to encrypt passwords
- Publicly readable because has a lot of useful information, but allows for offline attack
- Solution: hashes stored in `/etc/shadow`, this is only accessible to privileged users

4.2.4 Implementation

- In the kernel files are represented using **index-nodes** or **i-nodes**
- I-Nodes hold metadata about the file (owner, permissions, reference count, etc.) and its location on disk
- The filename is stored in the directory



- Directory just a file mapping filenames to i-nodes
- Directories has their own i-nodes
- An instance of a file in a directory is a hard link
- Directories can have at most 1 link - to stop cycles
- Also get soft or symbolic links, normal file which contains a filename
  - Can have cycles of these because files can be marked as visited
- A disk consists of a boot block followed by one or more partitions



- Note, $|datablocks| \gg |inodetable| \gg |superblock|$
- A partition is just a contiguous range of N fixed-size blocks, and a Unix filesystem resides wthin a partition
- Superblock contains info such as
  - Number of blocks and free blocks in filesystem
  - Start of the free-block and free-inode list
  - Various bookkeeping information
- Keep a chain of tables for free blocks and inodes, with each chains head in the superblock

- Superblock replicated, so it if is lost system can recover
- Entire filesystems can be mounted on an existing directory
  - Begin with root, and mount root filesystem
  - Provides a unified namespace
  - Can not have hard links across mount points
    - Refers to an inode and system assumes inode in same filesystem (disk)
  - Can have soft links because they are just a pathname

4.2.5 In-memory tables

- File descriptors are just indices into a process-specific open file table
- Entries in process-specific open file tables then point to a system wide open file table
- The entries in system wide open file table, then point inodes

4.2.6 Access and Consistency

- Access information held in each inode
- One bit for read, write, and execute for owner, group, and world
- In a directory, read entry, write entry, traverse directory
- Normally processes inherit permissions from invoker
- Setuid and setgid allow user to become someone else when running a given program
- The `unlink` command used to delete file
  1. Check if user has write permissions on file and directory
  2. If ok, remove entry from directory
  3. Decrement reference count on inode
  4. In now zero, free data blocks and free inode
- If system crashes during deletion must check entire file system of any block unreferenced or double referenced

4.3 Processes and IO

4.3.1 Implementation and Buffer Cache

- Everything accessed via the filesystem
- Two categories of device
  - Character IO - low rate but complex, most functionality is in the cooked interface
  - Block IO - simpler but more performance, uses buffered cache
- The Buffer Cache - Keep a copy of parts of the disk in memory for speed
  - Executed as follows
    1. Locate relevant blocks in memory (from inode)
    2. Check if in buffer cache
    3. If not, read from disk into memory (buffer cache)
    4. Perform action of data in memory

- Typically prevents 85% of implied disk reads
- Call sync every 30 seconds or so to flush dirty buffers to disk
- If power off will lose buffer cache
- Can cache metadata too - meaning access permission may not change immediately

4.4 Processes

- Processes have three sections
  - Text - Holds the machine code
  - Data - Contains variables and their values
  - Stack - Used for activation records - local variables etc.
- Process represented by opaque process id (`pid`)
- Organised hierarchically, with parents creating children
- Four basic operations
  - `pid = fork()`
  - `reply = execve(pathname, argv, envp)`
  - `exit(status)`
  - `pid = wait(status)`
- On poweron
  1. Kernel loaded from disk
  2. Mounts root filesystem
  3. Process 1 (`/etc/init` starts
  4. `init` reads file `/etc/inittab` and for each entry
     (a) Opens terminal special file (allows device IO)
     (b) Duplicates the resulting fd twice
     (c) Forks an `/etc/tty` process
  5. Each tty process, initialises the terminal; outputs string `login:` and waits for input; `execve()`'s `/bin/login`
  6. `login` then outputs `password:` tests inputted string, if correct, sets uid and gid and `execvs()`'s shell

4.4.1 Scheduling

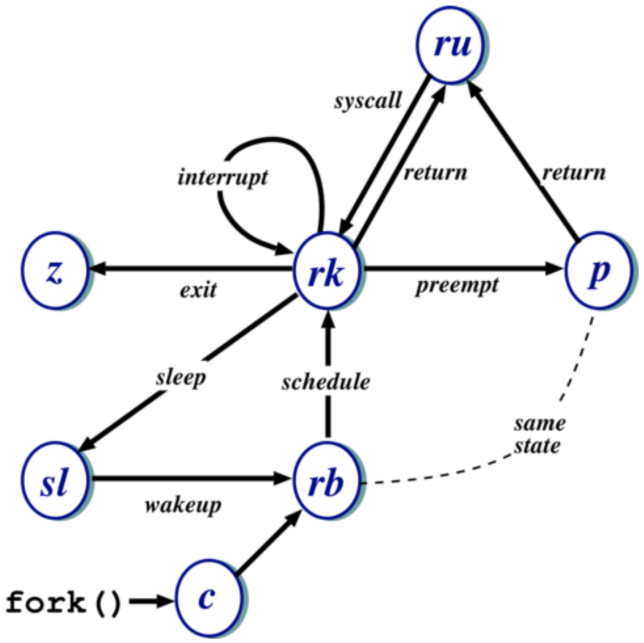
- Processes have priorities, 0 to 127
- user process priorities $\geq$ PUSER = 50, PUSER priority of poweruser
- Round robin scheduling within priorities, quantum of 100ms
- Priorities are based of usage and *nice*, the priority of a process j at the start of an interval i is

$$P_j(i) = Base_j + \frac{CPU_j(i-1)}{4} + 2 \times nice_j$$

where

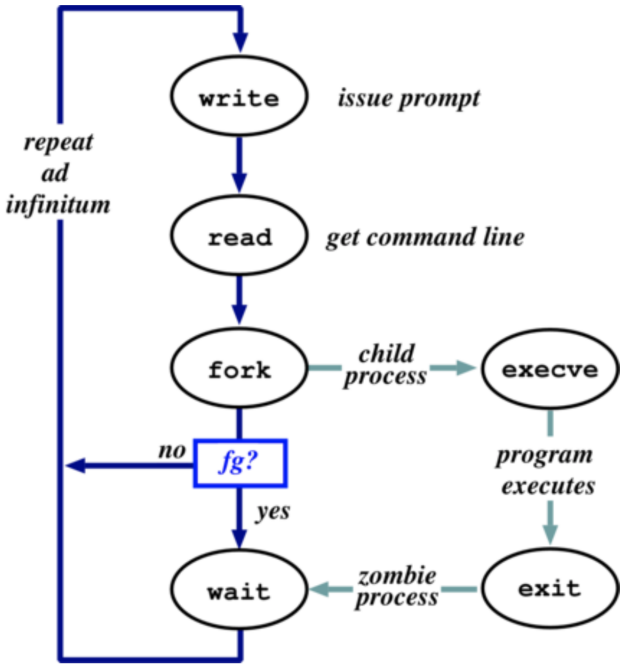
$$CPU_j(i) = \frac{2 \times load_j}{(2 \times load_j + 1)} \times CPU_j(i-1) + nice_j$$

- Where $nice_j$ is a partially user controlled adjustment parameter in the range $[-20,\ 20]$
- *length* is the sampled average of the run queue in which process j resides, over the last minute
- Base priority divides processes into bands, these bands are
  - Swapper
  - Block IO device control
  - File manipulation
  - Character IO device control
  - User processes
- **Process States**



ru	running (user-mode)	rk	running (kernel-mode)
z	zombie	p	preempted
sl	sleeping	rb	runnable
c	created		

- A process becomes preempted when another of a higher priority becomes available
- **The shell**
  - Just a process
  - Just uses and runs files
  - Conventionally & specifies background



- **Standard IO**
  - Every process has three file descriptors on creation
    - `stdin` - Where to read input from
    - `stdout` - Where to send output
    - `stderr` - Where to send diagnostics
  - Shell can redirect of standard IO to/from files, using <, >, and &>, for `stdin`, `stdout`, and `stderr` respectively
  - A pipe `|` can be used instead of a temporary file