2018

Operating Systems Notes

UNIVERSITY OF CAMBRIDGE, PART IA

ASHWIN AHUJA

Table of Contents

Operating Systems

Introduction

Text

For both ASCII and Unicode, we represent text as a string as an array of characters. However, we do sometimes need to be careful about when the character ends, since most of the time, the length of the character is less than the word size of the machine.

ASCII: 7-bit code holding letters, numbers, punctuation and a few other characters. There are regional 8-bit variations. Used to be the widespread default, but now Unicode (especially UTF-8) is becoming popular.

Unicode: 8, 16 or 32-bit code intended to support all international alphabets and symbols. Unicode 9 has 128,172 characters out of a potential 1,114,112 code points

UTF-8: Has backwards compatibility with ASCII (the low 128 bits map directly to the ASCII characters). In order to deal with variable length, all characters (other than ASCII) are encoded as <len><codes> where 0xC0 ≤ len ≤ 0xFD encodes the length while 0x90 ≤ codes ≤ 0xFD. The top two bytes are unused.

Number

An n-bit register can represent 2^n different values. The highest value bit is called the Most Significant Bit and the Least Significant Bit is the lowest one.

For unsigned numbers, we treat subsequent bits simply as the representation of the next highest 2^n (n^{th} bit (starting at 0) indicates the number of 2^n)

We generally use hexadecimal instead, as binary is rather unwieldy, with each binary nibble (group of 4 bits) being converted to a single hexadecimal digit. We often use the 0x prefix to show that it is hexadecimal. We also sometimes use a dot to separate large numbers into 16-bit chunks.

Signed Numbers: There are two main options for signed numbers

1. Sign and Magnitude
    a. Top bit flags if negative
    b. The remaining bits make the value
    c. We can have $-(2^{n-1}-1)$ to $+(2^{n-1}-1)$
    d. Also have -0
2. Two's Complement
    a. To get from -x to x, invert every bit and add 1
    b. 100..000 = -2^{n-1}
    c. Representation range from -2^{n-1} to $+(2^{n-1}-1)$
    d. It is much easier to do arithmetic with this

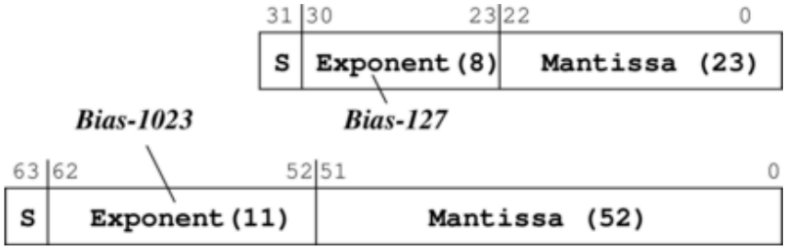
Floating Point: Use **mantissa** and **exponent**
In practise, use IEEE standard of normalised mantissa – has to start 10… The idea is that there is a 'decimal point' after the first digit of the mantissa. There is also a sign bit

Therefore, $n = (-1)^s((1+m) \times 2^{e-b})$

The standard reserves the e = 0 and maximum values as follows:

- M = 0
  - E = max means plus / minus infinity
  - E = 0 means plus / minus 0
- M != 0
  - E = max means NaNs
  - E = 0 means denorms
    - It is actually a number, but it is not normalised

We have single or double precision:



Biasing: For the exponent, instead of using any other signing system, we can simply imagine that everything is itself subtract some number, this is the bias factor. This means it is very easy to sort – since the lower the value of the exponent, the actual lower the value is.

It is important to note that the number of values is not increased (still 2^{32} or 2^{64}) but these are much more spread out. This offers a lot of precision near 0, but very low precision as it goes up.

Data Structures

The data structure is not interpreted by the machine – it is simply up to a programmer (or compiler) where things go and how they are stored.

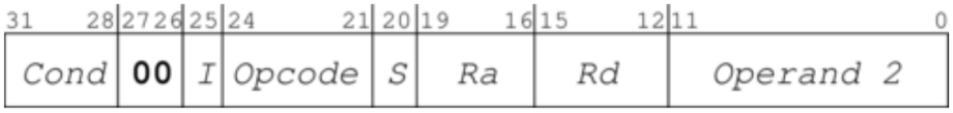
Fields in records are stored as an offset from a base address. In variable size structures, we explicitly store the addresses (pointers) within the structure.

Encoding

Instructions comprise of:

1. Opcode: what operation to do
2. Operand: where to get values to do this on
   a. Addressing Mode: How to use this value
      i. Do we need to go to memory – is this the address to get it from
      ii. Is this the actual value?
      iii. Is the memory address to go to stored within this memory address linked here?

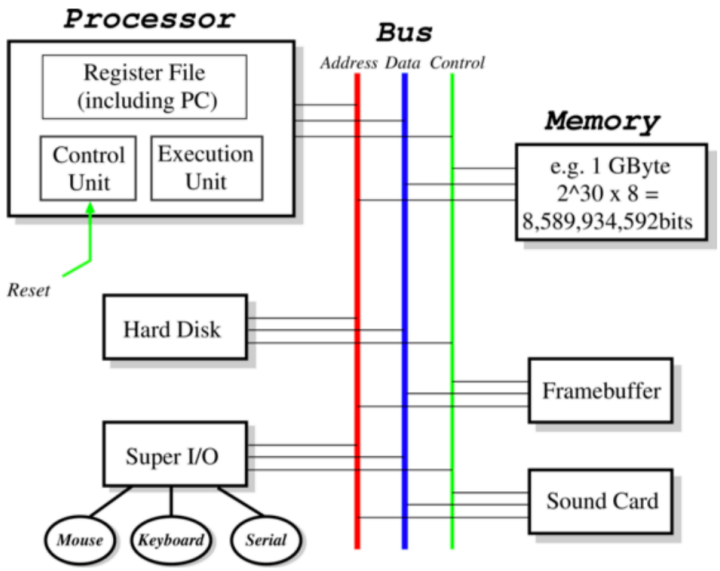
Generally, we use fixed length encoding, where the structure of the instruction is defined. For example, this is the structure of an ARM ALU operation:



Variable Length Encoding:

- It may give us better code density
- Makes it easier to extend the instruction set
- We have Huffman Encoding
    - Looks at the most probable instructions and assigns them to the shortest opcodes; infrequently user instructions get long opcodes.
- But VLE makes decoding much more challenging, and is generally bad for the cache as well
- We therefore, reasonably rarely use this.

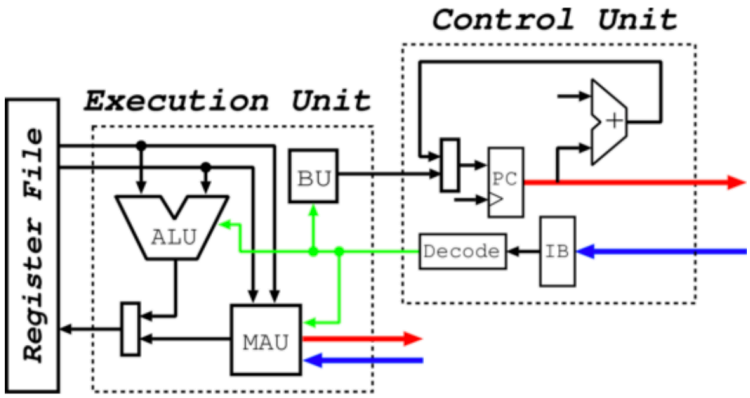
Model Computer



Processor (CPU) executes programs using:

1. Memory: stores programs and data
    a. Large byte array that holds any information on which we operate
2. Devices: for input and output
3. Bus: transfers information
4. Registers
    a. Extremely fast pieces of on-chip memory, generally 64-bits in size
    b. Modern CPUs have between 8 and 128 registers
    c. Data values are loaded from memory into registers before being operated on and being moved back again

Fetch-Execute Cycle



The CPU in turn, fetches and decodes the instruction, generating control signals and operand information. The PC (Program Counter) stores where the instruction is, going to get the instruction from memory before it is placed in the Instruction Buffer. Here, the decoding is done by a single decoding unit.

Inside the Execution Unit (EU), control signals select the Functional Unit (FU) – "instruction class" – and operation.

- If the Arithmetic Logic Unit (ALU) is the FU, then we have to read one or two registers, perform the operation and probably write back the result
- If the Branch Unit (BU), we test the flags and maybe add a value to the Program Counter (go to a different place)
- If it the Memory Address Unit (MAU), we generate the address (addressing mode) and use the bus to read or write the values.

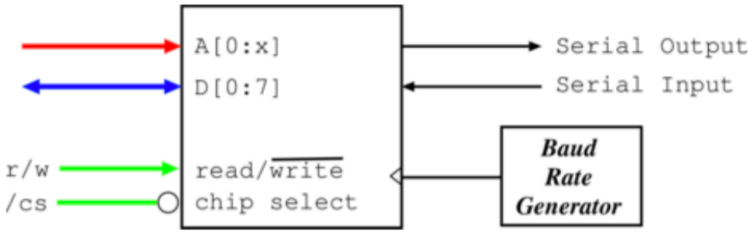
Input / Output Devices

These are devices connected to the processor via a bus

- Mouse, Keyboard
- Graphics Card

There are often two or more stages involved (conversion between protocols, RS-232, USB, etc.). Additionally, the connections may be indirect, for example a monitor is an I/O device which may be controlled by the Graphics Card.

UART (Universal Asynchronous Receiver / Transmitter)



Converts between parallel signals to serial signals. Therefore, has to store some number of bits internally.

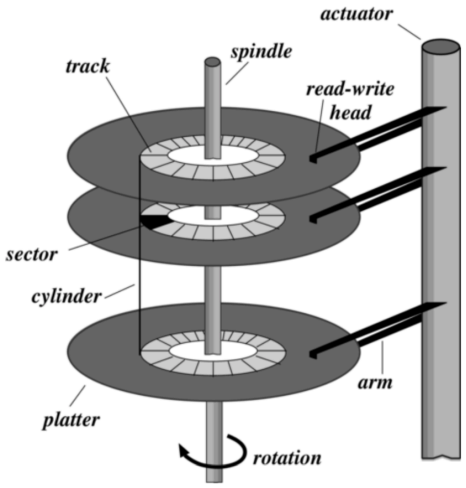
- It generally outputs to RS-232 – a standard for serial communication
- It has various baud rates (number of signal changes per second) (1,200 to 115,200)

- It is slow and simple (but very useful)

It makes up many serial ports on PCs

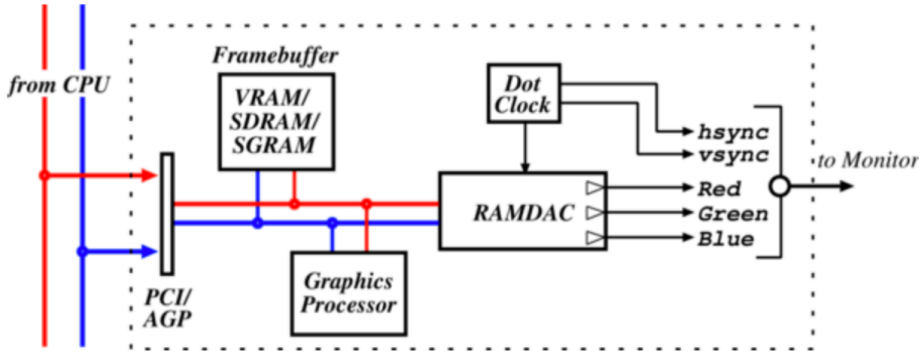
- It has a max throughput of ~14.4kb/s with variants up to 56kb/s
- It can also be connected to terminals to debug the device

Hard Disks



Whirling bits of petal, with a number of platters and read-write heads. The platters rotate on a spindle. They rotate up to 15,000 rpm and have ~2TBs per platter, and transfer at speeds until 2Gb/s

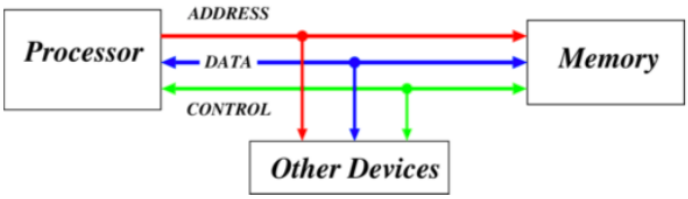
Graphics Cards



Essentially some RAM (**framebuffer**) and some digital-to-analogue circuitry (**RAMDAC**)

- RAM holds array of pixels: picture elements
- We have a colour depth which is the number of bits per pixel
  - 8-bit – LUT
  - 16-bit – RGB-555
  - 24-bit – RGB-888
- Memory requirement = xy x depth
- Full-screen 50Hz requires 125 MB/s or around 1Gb/s

Buses



They are a collection of shared communication wires – this is a low cost, versatile method but can be a potential bottleneck.

It typically comprises of:

1. Address lines
    a. Determines how many devices on the bus
2. Data lines
    a. Determines how many bits transferred at once
3. Control Lines
    a. Sends control signals and gets them from devices

It generally operates in a master-slave manner:

- Master decides to do something, eg read data
- Master puts address onto bus and asserts read
- Slave reads address from bus and retrieves data
- Slave puts data onto bus
- Master reads data from bus

Interrupts

Bus reads and writes are **transaction based:** CPU requests something and waits until it happens. However, reading a block of data from a hard-disk can take 2ms.

Interrupts allow a way to decouple CPU requests from device responses (also device unexpected signals)

- CPU uses bus to make request
- Device fetches data while CPU continues to do other stuff
- Device raises an interrupt when it has data
- On interrupt, CPU vectors to handler reads data and resumes using special instruction (rti)

Interrupts happen at any time but are deferred to an instruction boundary. Interrupt handlers must not trash registers and must know where to resume. CPU generally saves the values of these registers, restoring with rti.

Direct Memory Access

Interrupts are good, but still requires CPU time to copy data from the device to memory. It is even better if the device can read and write memory directly.

A generic DMA command, can include:

- Source address
- Source increment / decrement / do nothing

- Sink address
- Sink increment / decrement / do nothing
- Transfer size

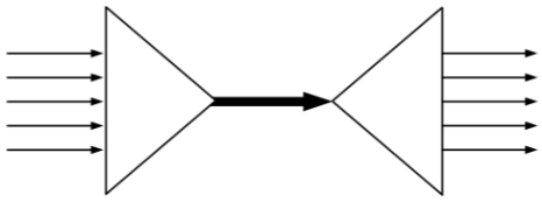
There is just one interrupt at the end of data transfer – effectively a confirmation that data transfer has occurred.

DMA channels may be provided by a dedicated DMA controller, or by the devices themselves – all that is required is that a device can become a bus master.

Layering

Means to control complexity by controlling interactions between components. Arrange components in a stack and restrict the component at layer X from relying on any other component other than that at layer X-1 and restrict it from providing service to any component other than that at layer X+1

Multiplexing



Method by which multiple signals are combined into a single signal over a shared medium. Any situation where one resource is being consumed by multiple consumers simultaneously.

Synchronous vs Asynchronous

There is a shared clock in synchronous, while no shared clock in asynchronous. In the case of Operating Systems, it affects whether two components operate in lock-step:

- Synchronous IO means the requester waits until the request is fulfilled before proceeding
- Asynchronous IO, the requester proceeds and later handles fulfilment of the request.

In the case of networking

- Asynchronous receiver needs to work out for itself when the transfer of data starts and ends
- Synchronous receiver has a channel over which that is communicated

Latency: How long something takes
Bandwidth: The rate at which something occurs
Jitter: The variation (statistical dispersion) in latency

We can concern ourselves with absolute or relative values of these and sometimes the distribution of the values is of interest.

Caching and Buffering

Impedance Mismatch: Two components are running at different speeds (latency, bandwidths). We can deal with this, in two particular ways:

- Caching
  - Small amount of higher-performance storage is used to mask the performance impact of a larger lower-performance components.
  - It relies on **locality** in time (finite resource) and space (non-zero cost)
  - The CPU has registers, L1, L2, L3 cache and main memory
- Buffering
  - Memory of some kind is introduced between two components to soak up small, variable imbalances in bandwidth
  - This doesn't help if one components speed on average exceeds the other.
  - Hard disk will have on-board memory into which the disk hardware reads data, and from which the OS reads data out.

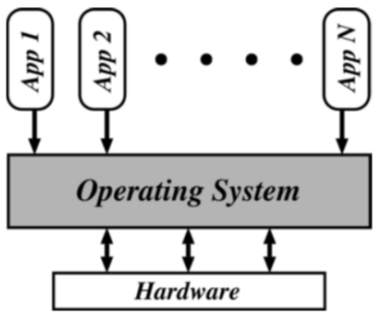
Bottlenecks, Tuning, 80/20 Rule

Bottleneck: One resource that is most constrained in a system. Performance optimisation and **tuning** focuses on determining and eliminating bottlenecks – however, it often introduces new ones. A perfectly balanced system has all resources simultaneously bottlenecked.

Often find out that optimising the common case gets most of the benefit anyway. This means that measurement is a prerequisite to performance tuning.

80/20 Rule: 80% time spent on 20% code

Operating System



What is an Operating System:

- A program controlling the execution of all other programs
- Controls all execution, multiplexes resources between applications and abstracts away complexity
    - For the abstraction – generally involves libraries and tools provided as part of the OS, in addition to a kernel
    - Therefore, no one really agrees precisely what an OS is
    - For our purpose, we focus on the **kernel**

Objectives:

1. **Convenience**
    a. Hide all the hardware interaction and other complexities
2. **Efficiency**
    a. Only does articulation work so minimises overheads
3. **Extensibility**
    a. It needs to be able to evolve to meet changing application demands and resource constraints.

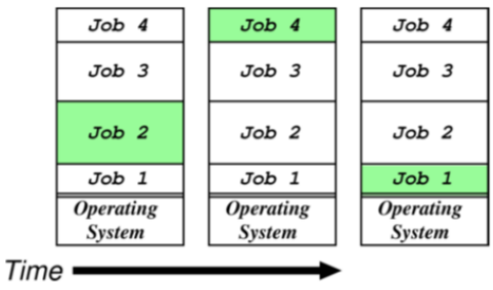
In the beginning: First stored machines operated "open shop". All programming was in machine code and everyone was treated equal (user, programmer, operator). Users signed up for blocks of time to do development, debugging, etc. Very little interactivity – so CPU utilisation reduced

Batch Systems: Introduction of tape drives allowed batching of jobs

1. Programmers put jobs onto cards
2. Cards read onto a tape
3. Operator carries input tape to computer
4. Results written to output tape
5. Output tape to printer

Spooling Systems: Spool jobs to tape for input to CPU, on a slower device not connected to CPU. There was an interrupt driven IO, with a magnetic disk to cache the input tape. The scheduling of the job was done specifically by the person, but all jobs had to return control to the OS (which would show list of jobs on a monitor), therefore we needed to trust the job to give control back.

Multi-Programming



- Use memory to cache jobs from disk
    - More than one job active at a time

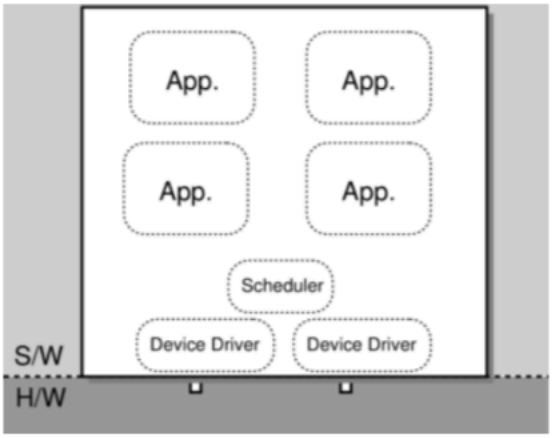
- Two stage scheduling
    - 1) Select jobs to run – **JOB SCHEDULING**
    - 2) Select resident job to run – **CPU SCHEDULING**
    - End up with one job computing while another waits for IO, causing competition for CPU and space in main memory

Batch Multi-Programming: Extension of batch system to allow more than one job to be resident simultaneously.

Users wanting more interaction between items leads to time-sharing

- CTSS (1961), TSO, Unix, VMS, Windows NT
- Use timesharing to develop code, then batch to run: give each user a terminal; interrupt on return; OS reads line and creates new job

Monolithic Operating Systems



These are the oldest kind of OS structure (DOS, MacOS). The applications and OS are bound in a big clump without clear interfaces. All the OS provides is a simple abstraction layer, making it easier to write applications

Problem is that applications can trash the OS, other applications, lock the CPU, abuse IO, etc. **Doesn't provide useful fault containment.**

Operating Systems Functions

1. Needs to securely multiplex resources
    a. Protect applications while sharing physical resources.
2. Also want to abstract away from hardware, i.e OS provides a virtual machine to:
3. Share CPU (in time) and provide each application with a virtual processor
4. Allocate and protect memory, and provide applications with their own virtual address space
5. Present a set of hardware independent virtual devices
6. Divide up storage spaces by using filing systems.

Processor

Protection

Protecting against:

1. Unauthorised release of information
    a. Reading or leaking data
    b. Violating privacy legislation
    c. Covert channels, traffic analysis
2. Unauthorised modification of information
    a. Changing access rights
    b. Can do sabotage without reading information
3. Unauthorised denial of service
    a. Causing a crash
    b. Causing high load
4. Effects of errors
    a. Isolate for debugging, damage control
5. Improper access
    a. **Impose access control by subjects (users) to objects (files)**

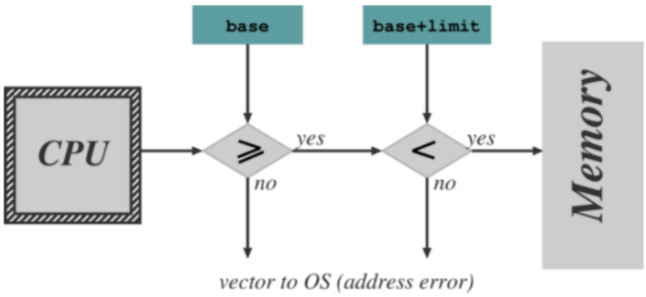
Design of Protection Systems – Saltzer and Schroeder, IEEE

1. The design should be public
2. The default should be no access
3. Check for current authority
4. Give each process minimum possible authority
5. Mechanisms should be simple, uniform and built in to lowest layers
6. Should be psychologically acceptable
7. Cost of circumvention should be high
8. Minimize shared cost

Low Level Protection

IO and Memory

- Try to make IO instructions privileged
    - Applications can't mask interrupts
    - Applications can't control IO devices
- But:
    - Some devices are accessed via memory, not special instructions
    - Applications can rewrite interrupt vectors
- Hence, protecting IO means also protecting memory
    - Define a base and a limit for each program and protect access outside allowed range
- **Implementing Memory Protection**



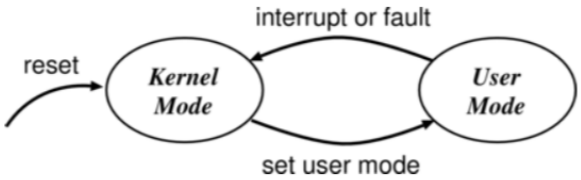
- Hardware checking every memory reference
  - Access out of range causes vector into OS (as for an interrupt)
  - Only allow update of base and limit registers when in kernel mode
  - May disable memory protection in kernel mode (although a bad idea)
- *In reality, more complex protection hardware is generally used*

CPU

- Need to ensure that the OS always stays in control
  - Prevent any application from hogging the CPU
  - Normally means using a timer
    - Set timer to initial value
    - Every tick, timer decrements value
    - When the value hits zero, there is an interrupt which changes control back to the OS
  - This requires that only the OS can load the timer and that the interrupt cannot be masked
    - We use the same scheme for the timer as for other devices
    - And re-use it to implement time-sharing

OS Structures

Dual-Mode Operation



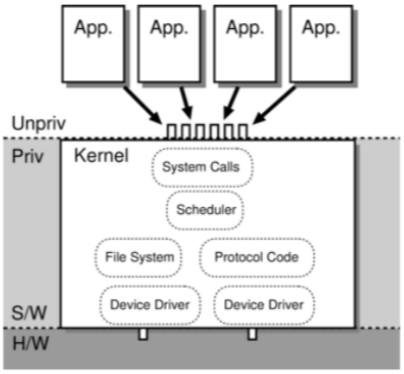
We want to stop buggy or malicious programs from doing bad things. Therefore:

- We trust the boundary between a user application and the OS
- We use hardware support to differentiate between two modes of operation
  - 1) User Mode: when executing on behalf of the user (application programs)
  - 2) Kernel Mode: when executing on behalf of the OS
    - Certain instructions only possible in kernel mode – indicated by **MODE BIT**
  - **Examples**
    - **X86 has rings 0-3**
      - **The rings can be nested with further inside being able to do strictly more**
      - This is not ideal

- We also want to be able to stop the kernel messing with the applications
- But disjoin permissions are generally hard

- **ARM has two modes plus:**
  - IRQ, Abort and FIQ

Kernel-Based Operating Systems



Applications can't do IO due to protection, so the OS does the IO on their behalf. To invoke the OS from an application, we have a special instruction to transition from user to kernel mode.

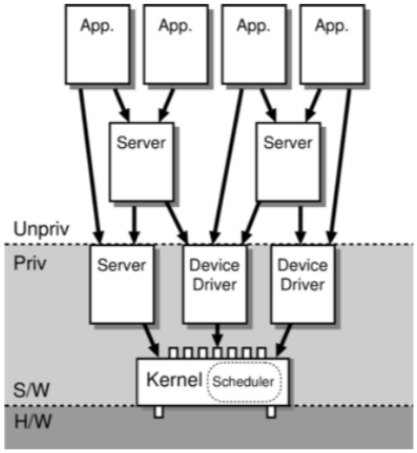
Software Interrupt (Trap): Operates similarly to hardware interrupt. The OS services are accessible via a software interrupt mechanism called **system calls**

The OS has vectors to handle Software Interrupts, preventing the application from going to kernel mode and then doing whatever it likes.

Alternatively:

- OS to emulate for every application
- And check every instruction
- Used in some virtualization systems (QEMU)

Microkernel Operating Systems



Applications can't do very much directly and must use the OS on their behalf

The OS must be very stable to support applications, so becomes very difficult to extend.

An alternative to kernel system is microkernels:

- Move OS services into local services, which may be privileged
- Increases modularity and extensibility
- You still access the kernel vis system calls but we need new ways to access the 'servers'
    - **Inter-Process Communication (IPC)**
- Given talking to the servers largely replaces trapping, we need IPC schemes to be extremely efficient.

Kernels vs Microkernels

- Lots of IPC adds overhead therefore, microkernels usually perform less well
- Microkernel implementation sometimes tricky: need to worry about synchronisation
- Microkernels often end up with redundant copies of OS data structure

Therefore, some common OSs blur distinction between kernel and microkernel

- Linux
    - Kernel but has kernel modules and some servers
- Windows NT
    - Was microkernel but moved back into kernel

Virtual Machines and Containers

- Alternative to kernels is encapsulating applications
- Making applications appear as if they're the only application running on the system
- This is particularly relevant when building systems using micro services.
    - This is **isolation** at a different level
- **Virtual Machines** encapsulates an entire running system, including the OS and then boots the VM over a hypervisor
- **Containers** expose functionality in the OS so that each container acts as a separate entity even though they all share the same underlying OS functionality.

Mandatory Access Control

- Mandates expression of policies constraining interaction of system users. For example, OSX and iOS Sandbox uses subject/object labelling to implement access control for privileges and various resources (filesystem, communication, APIs, etc)
- General idea is that applications are protected from each other.
- **Pledge (2)**
    - This is one way to reduce the ability of a compromised program to complete bad things
    - By removing access to unnecessary system calls
    - Several attempts in different systems with limited success
        - Hard to use correctly
        - Introduce another component that needs to be watched
    - Pledge(2)
        - Asks the programmer to indicate explicitly which class of system call they wish to use at any point

Authentication

Authenticating User to System

- **Passwords**
  - But people pick badly
  - Also, security of password file
    - We restrict access to login programme
    - Store scrambled using a one-way function
  - We often prefer key-based systems
- **Unix**
  - Password is DES-encrypted 25 times, using a 2-byte per-user salt to produce an 11-byte string
  - Salt and these 11 bytes are stored
- Can enhance everything with biometrics

Authenticating of System to User

- Want to avoid user talking to
  - Wrong computer
  - Right computer but not the login program
- Partial solution in the old days for directly wired terminals
  - Make the login character same as the one for terminal attention
    - Or just tell people to do terminal attention command before trying login
    - Control-Alt-Del
- Today micros used as terminals
  - Local software might have been changed – so you could carry your own version of the terminal program
  - However, hardware / firmware in public machines may also have been modified.
  - Also, wiretapping is easy.

Mutual Suspicion

- Solution is to encourage lots of suspicion
  - System of the user
  - Users of each other
  - User of system
- Called programs should be suspicious of caller
  - OS calls should always check parameters
- Caller should be suspicious of called programs
  - Trojans

Access Matrix

- Matrix of subjects against objects
- Subjects (or principal)
  - Users (by UID)
  - Executing process in a protection domain

- Objects
    - Files, devices
    - Domains, processes
    - Message ports (in microkernels)
- The matrix is large and sparse, so we don't store it all
    - Store by Object (**Access Control List**): Store List of Subjects and Rights with each object
        - Often used in storage systems
            - System naming scheme provides for ACL to be inserted in naming path
            - If the ACL is stored on disk, the check is made in software, so we can use it only on low duty cycle
            - For higher duty cycle, we must cache results of the check
            - **Example**
                - Open file is a memory segment – on first reference, causes a fault which raises an interrupt which allows OS to check against ACL
            - ACL is checked when file is opened for read or write, on when code file is to be executed.
    - Store by Subject (**Capabilities**): Store list of objects and rights with each subject
        - Associated with active subjects so:
            - Store in the address space of the subject
            - Must make sure that the subject can't forge capabilities
            - It is easily accessible to hardware
            - Can be used with a high duty cycle.
        - Hardware capabilities
            - We have special machine instructions to modify capabilities
            - And support passing of capabilities on procedure (program call)
        - We also have software capabilities which can be checked by encryption and are generally nice for distributed systems.

Processes

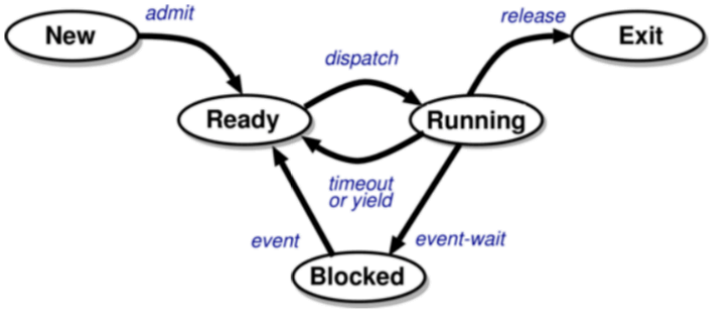
Process Concept

What is a process?

- A program is static, on-disk
- A process is dynamic, a program in execution
    - On a batch system, we might refer to jobs instead of processes.
    - A process is a unit of protection and resource allocation
        - Can have multiple copies of a process running
        - Each process executed on a virtual processor
    - It has a virtual address space of its own
    - It has one or more threads, each has:
        - **Program Counter**
        - **Stack**
            - Temporary Variables
            - Parameters

        - Return Addresses
    - **Data Section**
        - Global variables shared among threads

Process States



- **New:** being created
- **Running:** instructions are being executed
- **Ready:** waiting for the CPU, ready to run
- **Blocked:** stopped, waiting for an event to occur
- **Exit:** has finished execution

Process Lifestyle

Creation

- Systems are hierarchical – parent processes create child processes
- Resource sharing (3 options)
    - Parent and children share all resources
    - Children share subset of parent's resources
    - Share no resources
- Execution (2 options)
    - Parent and child execute concurrently
    - Parent waits until children terminate
- Address space
    - Child duplicate of parent
    - Then, child has a program loaded into it

System Calls (Unix)

- **Fork() –** creates a child process, cloned from the parent. Child receives snapshot of parent's address space. Parent then either detaches from child or waits for child.
- **Execve() –** used to replace the process' memory space with a new program

Termination

Occurs under three circumstances

1. Process executes last statement and asks the OS to delete it (exit)
    - Output data from child to parent (wait)
    - Process' resources are deallocated by the OS
2. Process performs an illegal operation
    - Makes an attempt to access memory to which it is not authorised
    - Attempts to execute a privileged instruction

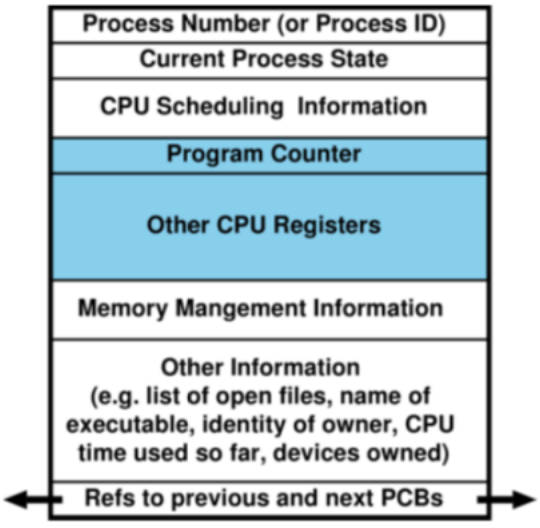
3. Parent may terminate execution of child processes (abort, kill) because
    - Exceeded allocated resources
    - No longer needed
    - Parent is exiting (**cascading termination**)

Blocking
A process blocks on an event such as when an IO device is completing an operation or if another process sends a message.

We can assume the OS provides so kind of general-purpose blocking primitive (await). We need to take care handling concurrency issues.

CPU IO Burst Cycle
- Process execution consists of CPU execution and IO wait
- Processes are either:
    - IO Bound
        - Spends more time in IO than computation
        - Many short CPU bursts
    - CPU Bound
        - Spend more time doing computations
        - Fewer, longer CPU bursts
- Most processes execute for at most a few milliseconds before blocking, therefore we need multiprogramming to obtain reasonable overall CPU utilisation.

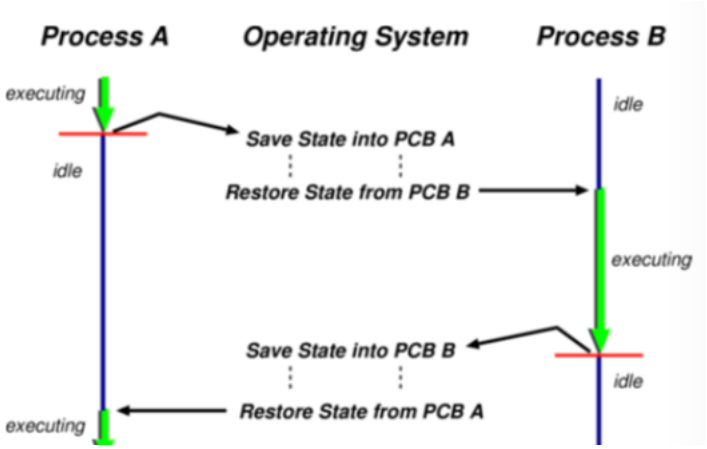
Process Management
Process Control Blocks



- OS uses this to maintain information about every process
- Is the machine environment during the time that the process is actively using the CPU
    - Program counter
    - General purpose registers
    - Processor status register

Context Switching

Context switches occur when the OS saves the state of one thread and restores the state of another. If this is between threads in different processes, process state also switches.



To switch between processes, OS must:

1. Save context of currently executing process
2. Restore context of that being resumed

This is wasted time – no useful work being carried out

The time taken for context switching depends on hardware changes:

- No changes
- Save and then load multiple registers from memory – complete task switch

Threads

- A thread represents an individual execution context. They are managed by a scheduler that determines which thread to run.
- Each thread has an associated **Thread Control Block (TCB)** with metadata about the thread: saved context (registers, including stack pointer), scheduler info, etc
- Threads visible to the OS are **kernel threads –** they may execute in kernel or address space.

Inter-Process Communication (IPC)

For meaningful communication to take place, two or more parties must exchange information according to a protocol

- Commonly-understood format (syntax)
- Mutually-agreed meaning (semantics)
- According to mutually understood rules (synchronisation)

There is communication between a range of parties – threads, processes, hosts.

IPC is communication between processes on the same host – with the key point being that it is possible to share memory between the processes. Given the protection boundaries imposed by the OS, by design, the OS involved in any communication between processes – otherwise equivalent to allowing one process to write over another's address space.

Inter-Thread Communication

If coordination is not implemented then problems can occur, therefore range of mechanisms to manage this:

- Mutexes
- Semaphores
- Monitors
- Lock-Free Data Structures

Inter-Host Communication

Passing data between different hosts:

- Traditionally different physical hosts
- Normally today virtual hosts

Key distinction is that there is now no shared memory, so some form of transmission medium must be used – **networking**

Much harder than IPC because real networks are:

1. Unreliable – data loss
2. Asynchronous – no guarantee about when data arrives

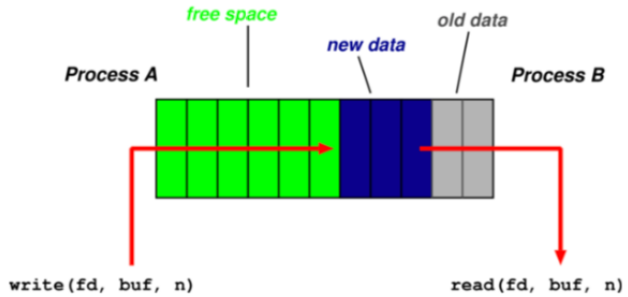
Signals

Simple asynchronous notifications on another process. They are a range of signals (28), defined as numbers.

- SIGHUP: hang up the terminal (1)
- SIGINT: terminal interrupt (2)
- SIGKILL: terminate the process (cannot be caught or ignored) (9)
- SIGTERM: terminate process (15)
- SIGSEGV: Segmentation fault – process made an invalid memory reference
- SIGUSR1/2: Two user signals (system defined numbers

Sigaction(2) specifies what function the signalled process should invoke on receipt of a given signal.

Pipes



One process can write to the pipe while another can read from the pipe. This is the simplest for of IPC.

Pipe(2) returns a pair of file descriptors (abstract indicator used to access a file) – one refers to the read file descriptor, one refers to the write side. With fork(2), you can now create a

new process and have the parent and child have the read/write file descriptors available and can therefore communicate.

FIFOs (Names Pipes)
Effectively the same as a pipe, except that it has a name, and isn't just an array of two file descriptors. This means that the parties can coordinate without being in a parent/child relationship – just need to share the name

Open(2) – will block by default until some other process opens the FIF for reading
Read(2)
Write(2)

Shared Memory Segments
Segment of memory that is shared between two or more processes:

- Shmget(2) – get a segment
- Schmat(2) – attach to it

Then we read and write via pointers (need to impose concurrency control to avoid collisions)

Finally:

- Shmdt(2) to detach
- Shmctl(2) to destroy once no-one using it

Files
Locking:

- Can be mandatory or advisory
- Advisory is more widely available
- Fcntl(2) sets, tests and clears the lock status
- Processes can then coordinate over access to files
- Read(2), write(2), seek(2) to interact and navigate

Memory Mapped Files:

- Mmap(2) maps a file into memory so you interact with it via a pointer
- Still need to lock or use some concurrency control mechanism

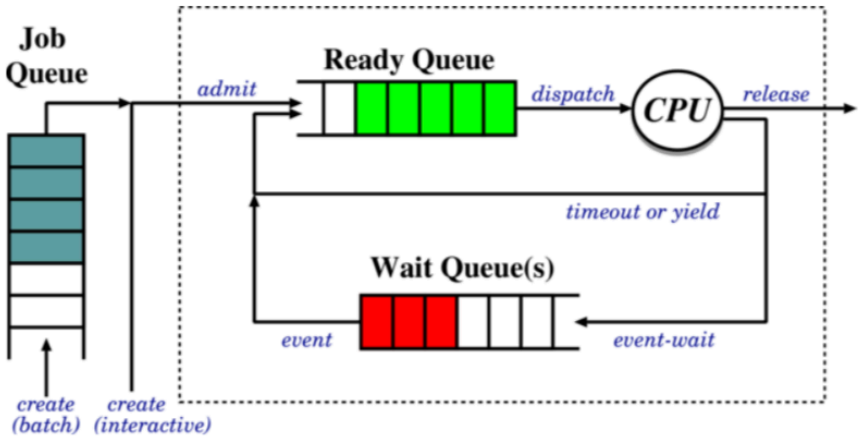
Unix Domain Sockets
Sockets: Commonly used in network programming – but there is effectively a shared memory version for use between local processes

- Socket(2) creates a socket, using AF_UNIX
- Bind(2) attaches the socket to a file
- They interact as with any socket
    - Accept(2), listen(2), recv(2), send(2)
    - Sendto(2), recvfrom(2)
- Socketpair(2) uses sockets to create a full-duplex pipe
    - Therefore, can read / write from both ends

Scheduling

Scheduling Concepts

Queues



1. **Job Queue**
    a. Batch processes awaiting admission
2. **Ready Queue**
    a. Processes in main memory, ready and waiting to execute
3. **Wait Queue**
    a. **Job** scheduler selects processes to put onto the ready queue
    b. **CPU** scheduler selects the process to execute next and allocates the CPU

Preemptive vs Non-Preemptive Scheduling

If a scheduling decision is only taken under the following conditions, it is said to be non-preemptive:

- A running process blocks
- A process terminates

Otherwise, it is said to be preemptive when it responds to the following things:

- A timer expires
- A waiting process unblocks

Non-Preemptive is much simpler to implement (needs no timers) and the process gets the CPU for as long as it needs. However, it is open to denial-of-service and malicious or buggy processes can refuse to yield. However, it typically includes an explicit yield system call (or similar) plus implicit yields – performing IO

Preemptive solves the denial-of-service problem, as the OS can simply pre-empt long-running processes. It is however, much more complex to implement, requiring timer management and dealing with concurrency issues.

Idling

There are three options of what to do when there isn't something ready to run:

1. Busy wait in scheduler
    a. Quick response time
    b. Fairly useless

2. Halt processor until an interrupt arrives
    a. This saves power and increases the processors lifetime
    b. But it might take too long to stop and start
3. Invent an idle process which is always available to run
    a. It gives uniform structure
    b. We can run house-keeping, but it uses some memory and might slow interrupt time

Trade-off between responsiveness and usefulness

Scheduling Criteria

Typically, we have more than one option of what to run – more than one process is runnable.

There are many different metrics, exhibiting different trade-offs and leading to different operating regimes.

1. **CPU Utilisation**
2. **Throughput**
    a. Maximise the number of processes that complete their execution per unit time.
    b. May penalise long-running processes as short-run processes will complete sooner and so are preferred.
3. **Turnaround Time**
    a. Minimise the amount of time to execute a particular process
4. **Waiting Time**
    a. Minimise the amount of time a process has been waiting in the ready queue.
    b. Ensures an interactive system remains as responsive as possible
    c. But it may penalise IO heavy processes that spend a long time in the wait queue.
5. **Response Time**
    a. Minimise the amount of time it takes from when a request was submitted until the first response is produced
    b. This is found in time-sharing systems – it ensures the system remains as responsive to clients as possible under load
    c. May penalise longer running sessions under heavy load.

Scheduling Algorithms

First-Come First-Served

Simplest possible scheduling algorithm, depending only on the order in which processes arrive.

Example:
Following demand:

Process	Burst Time
P_1	25
P_2	4
P_3	7

In the different arrival orders

1. P1, P2, P3
    a. Waiting time: P1 = 0, P2 = 25, P3 = 29
    b. Average Waiting Time = 18
2. P3, P2, P1
    a. Waiting time: P1 = 11, P2 = 7, P3 = 0
    b. Average Waiting Time = 6

Therefore, arriving in reverse order is three times as good. The first case is poor due to the **convoy effect:** The later processes are held up behind a long-running first process

FCFS is simple, but not robust to different arrival processes.

Shortest Job First

- Associate with each process, the length of its next CPU burst
- Use these lengths to schedule the process with the shortest time
- Use a different algorithm (such as FCFS) to break ties

Example:

Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

Waiting times: P1 = 0, P2 = 6, P3 = 3, P4 = 7
Average Waiting Time = 4

- SJF is optimal with respect to average waiting time
- But long processes might never get run

Shortest Response Time First

Preemptive version of SJF: we pre-empt the running process if a new process arrives with a CPU burst length less than the remaining time of the current executing process.

While this is technically optimal if we consider context switches to be instantaneous, this is clearly not true. Therefore, many very short burst length processes may thrash the CPU, preventing useful being done.

More importantly, we can't generally know what the future burst length is

Predicting Burst Length

- For both SJF and SRTF, we require the next burst length for each process means we must estimate it
- We can do this, by using the length of previous CPU bursts, using exponential averaging
    - 1) t_n = actual length of n^{th} CPU burst
    - 2) τ_{n+1} = predicted value for next CPU burst
    - 3) For α, $0 \leq \alpha \leq 1$, define:
        - $T_{n+1} = \alpha t_n + (1-\alpha)\tau_n$
- If we expand the formula, we get:

$$\tau_{n+1} = \alpha t_n + \ldots + (1-\alpha)^j \alpha t_{n-j} + \ldots + (1-\alpha)^{n+1} \tau_0$$

    - Where τ_0 is some constant
    - We choose the value of α according to our belief about the system, if we believe the history is irrelevant then we choose a value close to 1 and get $\tau_{n+1} \cong t_n$
    - In general, an exponential averaging scheme is a good predictor if the variance is small
    - Each successive term has less weight than its predecessor
    - We need some consideration of load, otherwise we get (counter-intuitively) increased priorities when increased load.

Round Robin

- Round-Robin is a preemptive scheduling scheme for time-sharing systems
- We define a small fixed unit of time called a quantum, typically 10-100 milliseconds
- The process at the front of the ready queue is allocated the CPU for up to one quantum
- When the time has elapsed, the process is pre-empted and appended to the ready queue.
- **Properties**
    - 1) Fair
    - 2) Live – no process waits more than (n-1)q time unites
    - 3) Typically get higher average turnaround time than SRTF, but better average response time
    - However, it is tricky to choose the correct size quantum
        - If q gets too large, it becomes FCFS / FIFO
        - If q gets too small, the context switch overhead gets too high

Static Priority Scheduling

We associate an integer priority with each process. The simplest form might be system vs user tasks. Then we allocate the CPU to the highest priority process – the highest priority is typically the smallest integer.

There are preemptive and non-preemptive variants – where a preemptive version means that we reassess when a higher priority item arrives.

Tie Breaking

- **Round-Robin with time-slicing**
- **However, this biases towards CPU intensive jobs**
- Solution
    - Per-process quantum based on usage

- Or we can just ignore the problem

Starvation: The biggest problem with static priority systems – a low priority process is not guaranteed to run ever

Dynamic Priority

- This prevents the starvation problem by using the same scheduling algorithm, but it allows the priorities to change over time.
- The processes have a static base priority and a dynamic effective priority
  - If the process is starved for some amount of time, we increment the effective priority
  - Once the process runs, we reset the effective priority

Computed Priority

- We have timeslots: 0 … t, t+1, …
- In each timeslot t, measure the CPU usage of process j : u^j
- Priority for process j in slot t + 1 =
  - $P^j_{t+1} = f(u^j_t, p^j_t, u^j_{t-1}, p^j_{t-1}, \ldots)$
  - Eg $p^j_{t+1} = ½\ p^j_t + ku^j_t$
- Penalises the CPU bound but supports IO bound
- This was once considered impractical but now such computation is considered entirely acceptable.

Memory Management

Virtual Addressing

Memory Management

Concepts

- In a multiprogramming system, we have many processes in memory simultaneously
- Every process needs memory for:
  - Instructions
  - Static data (in program)
  - Dynamic data (heap and stack)
- Also, operations system itself needs memory for instructions and data
  - Must share memory between OS and processes

1. **Relocation**
   i. Memory typically shared among processes, so programmer cannot know address that process will occupy at runtime
   ii. May want to swap processes into and out of memory to maximise CPU utilisation
   iii. Processes incorporate addressing info (branches, pointers, etc)
   iv. OS must manage these references to make sure they are correct
   v. Therefore, need to translate logical to physical addresses.
2. **Allocation**
   i. This is similar to sharing, but also related to relocation
      i. OS may need to choose addresses where things are placed to make linking or relocation easier.

3. **Protection**
    i. Protect one process from others
    ii. We may also want sophisticated RWX protection on small memory units
    iii. A process should not modify its own code
    iv. Dynamically computed addresses (array subscripts) should be checked
4. **Sharing**
    i. If multiple processes execute the same binary, we should keep only one copy
    ii. Shipping data around between processes by passing shared data references
    iii. Operating on the same data means sharing locks with other processes.
5. **Logical Organisation**
    i. Most physical memory system are linear address spaces from 0 to max
    ii. This doesn't correspond with modular structure of programs – we want segments
    iii. Modules can contain modifiable data, or just code
    iv. Therefore, it is useful if the OS can deal with modules which can be written and compiled independently.
    v. We can give different modules different protection, therefore easy for user to specify the sharing model.
6. **Physical Organisation**
    i. **Main Memory:** Single linear address space, volatile, more expensive
    ii. **Secondary Storage:** cheap, non-volatile, can be arbitrarily structured
    iii. **One key OS function** is to organise flow between main memory and the secondary key (cache)
    iv. The programmer may not know how much space will be available.

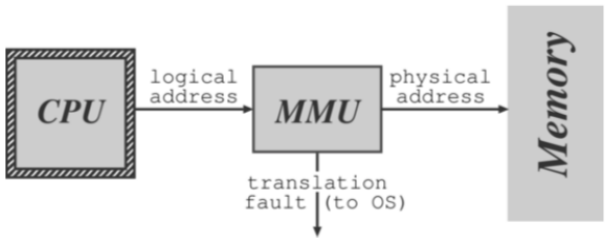
Address Binding Problem

The address binding problem is the problem that we do not know where in memory to store variables for a piece of code, since we have no idea where in memory, the program itself will be loaded from when we run it. Therefore, by placing it in a specific location, it could be in an area we don't have access to.

This can be done at:

1. Compile Time
    a. This requires knowledge of the absolute addresses
2. Load time
    a. Find the position in memory after loading, update the code with the correct addresses
    b. This must be done every time the program is loaded
    c. While okay for embedded systems and bootloaders, harder for actual full-size computers
3. Run time
    a. Get the hardware to automatically translate between program and real addresses
    b. It requires no changes at all to the program itself
    c. This is the most popular and flexible scheme, assuming the requisite hardware is included (Memory Management Unit)

The Mapping of logical to physical addresses is done at run-time by the Memory Management Unit (MMU)



1. Relocation register holds the value of the base address owned by the process
2. Relocation register contents are added to each memory address before it is sent to memory
3. **Example (DOS on 80x86)**
    a. There are 4 relocation registers, the logical address is a tuple (s, o)
    b. The process never sees physical address – simply manipulates logical addresses
4. The OS has privilege to update relocation register

Allocation

Contiguous Allocation

1. The OS must typically be in low memory due to the location of interrupt vectors
2. The easiest way is to statistically divide memory into multiple fixed size partitions
    a. Bottom partition contains OS, remainder each contain exactly one process
3. When a process terminates its partition becomes available to new processes
4. Protection
    a. Base and limit registers in MMU
    b. Update values when a new process is scheduled

Static Multiprogramming

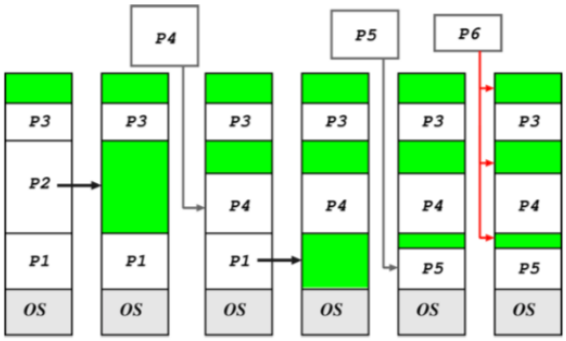
- Partition memory when installing the OS and allocate pieces to different job queues
- We associate jobs to a job queue according to size
- We swap the job back to disk when
    - It is blocked on IO (assuming IO is slower than the backing store)
    - Time sliced
- Run job from another queue while swapping jobs
- Problems: Fragmentation!

Dynamic Partitioning

- This gives us more flexibility if we allow the partition sizes to be dynamically chosen
- The OS keeps track of which areas of memory are available and which are used
    - Linked Lists
- For a new process, the OS looks for a hole large enough to fit it
    - 1) First Fit
        - Stop searching list as soon as a big enough hole is found
    - 2) Best Fit
        - Search entire list to find best fitting hole
    - 3) Worst Fit
        - Allocate largest hole

- First and Best perform best in terms of time and space utilisation (though typically for N allocated blocks, we have another 0.5N in wasted fit using first fit)
  - Which is better depends on the pattern of process swapping
  - We can use the buddy system to make allocation easier
    - Various forms of the buddy system
    - Initially, treat entire free space as a single block
    - When request is made, if its size is greater than half the block, then the entire block is allocated. Otherwise, the block is split into two equal companion buddies. If the size of the request is greater than half of one of the buddies then allocate to it, etc.
    - When a process terminates, the buddy block that was allocated to it is freed.
    - Try to merge it with a companion buddy in order to form a larger free block.
  - When process terminates its memory returns onto the free list, coalescing holes wherever appropriate.

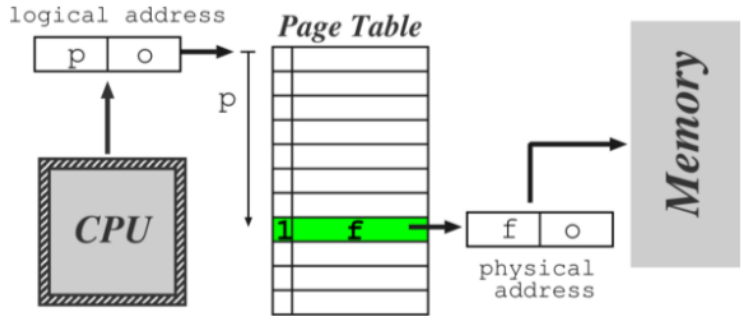
External Fragmentation



- As processes are loaded, they leave little fragments which may not be used.
- We can eventually block due to insufficient memory to swap in.
- Exists when the total available memory is sufficient for a request – but is unusable as it is split into many holes.
- We can also have problems with tiny holes when keeping track of holes costs more memory than the hole.
- **Compaction**
  - Remove the holes
  - Choosing the optimal strategy is hard, we note that:
    - 1) Requires run-time reallocation
    - 2) Can be done more efficiently when the process is moved into memory from a swap, so specific compaction is generally avoided.
    - 3) Some machines used to have hardware support.
  - We can also get fragmentation in the backing store, but in this case, compaction is not really viable.

Paging

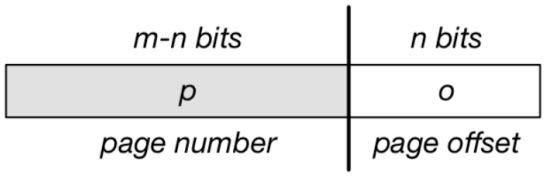
Paged Virtual Memory



Another solution is to allow a process to exist in non-contiguous memory. In order to do this, we:

1. Divide physical memory into frames (small fixed-size blocks)
2. Divide logical memory into pages (blocks of the same size – generally 4kB)
3. Each CPU-generated address is a page number with an offset
4. A Page Table contains an associated frame number f
    a. Usually have |p| >> |f| so also record whether the mapping valid

Paging Pros and Cons

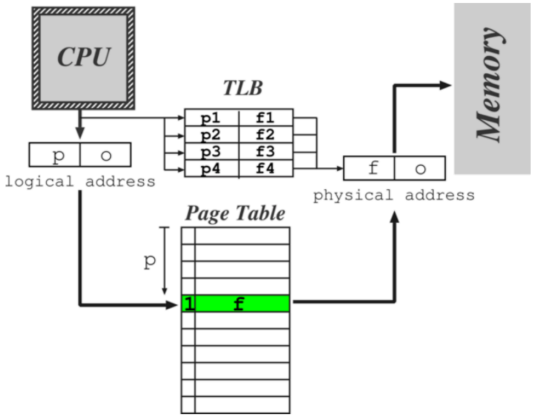


- Hardware support is required – generally defines the page size, typically a power of 2 ranging from 512B to 8192B
- Logical address space of 2^m and page size 2^n means p = m – n bits and offset = n bits
- Paging is a form of dynamic relocation – simply change page table to reflect movement of page in memory.
- Memory allocation becomes easier, but OS must maintain a page table per process
- No external fragmentation, but get internal fragmentation – a process may not use all of the final page.
- Small page sizes could fix this, but PTE (Page Table Entries) take up a lot of space
- Clear separation between user and system view of memory usage
    - Process sees a single logical address space; OS does all the hard work.
    - Process cannot address memory they don't own – cannot reference a page it doesn't have access to.
    - OS can map system resources into user address space – IO buffer
    - OS must keep track of free memory – **frame table**
- **Adds overhead to context switching**
    - Per process page table must be moved on context switch
    - Page Table may be large and extend into physical memory
- OS must keep Page Table per process

Page Tables

- Page Tables rely on hardware support

- 1) Set of dedicated relocation register
  - This requires one register per page and the OS loads the regsiters on a context switch.
  - Each memory reference goes through these so they must be fast.
  - This is okay for small PTs, but when we have many pages it is generally unfeasible.
    - Solution – keep PT in memory, then just need one MMU register – the (2) **Page Table Base Register** (PTBR)
    - OS must switch this when switching processes
    - However, the PT might still be too large – therefore have a **PT Length Register (PTLR) to indicate the size of the Page Table**
- 3) Translation Lookaside Buffer



  - Stops needing to refer to memory twice for every actual memory reference
  - This buffer contains some page references
  - If the PTE (Page Table Entry) is present, then we return the result immediately
  - Otherwise, find it in the PT and update the TLB
    - This is 10% slower than direct memory reference
  - **Issues**
    - What to do when full?
      - Discard least recently used entries
    - Context switch requires flushing of TLB
      - TLBs may support **process tags** (address space numbers) to improve the performance of this
  - **Hit Ratio:** Proportion of time a PTE is found using in a TLB

Example:
Assume TLB search time of 20ns, memory access time of 100ns, hit ratio of 80%

Assuming we are looking up one memory reference, the **effective memory access time is:**

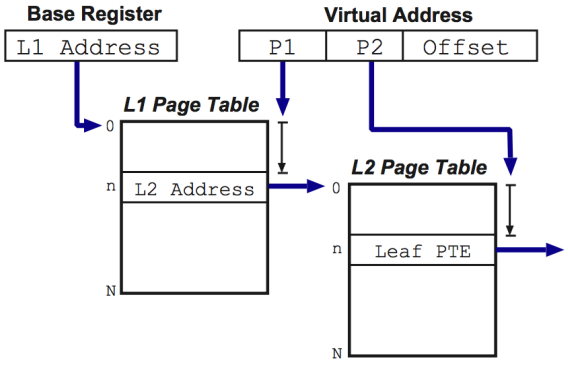
0.8 x 120 + 0.2 x 220 = 140ns

If we instead increase the hit ratio to 98%, the effective memory address time is = 122ns (13% improvement), therefore not that significant.

Multilevel Page Tables

Most modern systems can support large address spaces, therefore requires large PTs, which we don't want to keep all of in main memory. Therefore, we can split the Page Table into several sub-parts, and then page the page table.

We divide the page number into two parts – 'Page Number' and 'Page offset'. Then within the page number, we split into a few layers, each of which refers to the offset within the page table which will give the address of the next page layer's page table (or the page itself in the case of the last layer).



Example: VAX

- Logical address space divided into 4 sections of 2^{30} bytes
- The top 2 address bits designate the section
- Next 21 bits designate the page within the section
- The final 9 bits designate the offset

It must be noted that for some architectures (64-bit), two level paging is not enough therefore we need 4-5 level paging.

Example: x86

- Page size is 4kB or 4MB
- First lookup is to the page directory – index using 10 bits
  - Results stored within an internal processor register (cr3)
- The lookup results in the address of a page table
  - Should be noted that the page directory and page tables are all each one page
- Next 10 bits index the page table, retrieving the page frame address
- Final 12 bits get the page offset

Protection Issues

We associate protection bits with each page, kept in the page tables (and TLB). Could be one bit for read, one for write and one for execute. We might also distinguish whether it may only be accessed when in kernel mode.

As the address goes through the page hardware, we can check the protection bits – any attempt to violate protection leads to a hardware trap. This only gives page granularity protection, not byte granularity.

Shared Pages

- By having two logical addresses which map to one physical address, we can share pages.
    - Generally, should only do this if the code is re-entrant (stateless, non-self-modifying)
    - Otherwise, can use copy-on-write technique
        - Mark page as read-only in all processes
        - If a process tries to write to page, will trap to OS fault handler
        - Can then allocate new frame, copy data, and create new page mapping
- This generally requires additional book-keeping in OS, but worth it – large amounts of shared data.

Virtual Memory

- Virtual addressing allows us to introduce the idea of virtual memory
    - Already have valid or invalid pages – we introduce a **non-resident** designation
    - They live on a non-volatile backing store, such as a hard disk
    - The processes access non-resident memory as if it was memory
- Benefits
    - Portability – programs will work despite how much actual memory present
    - Convenience
    - Efficiency
- **Demand Paging**
    - Programs are on the disk
    - To execute a process, we load pages in on demand – as and when they are referenced
    - We also get demand segmentation, but this is rarer – segment replacement is much harder.
    - **Process**
        - When loading a new process for execution
            - 1) We create its address space (PTs, etc) but mark the PTEs as either invalid or non-resident
            - 2) Add the PCB (Process Control Block) to the scheduler
        - Whenever we receive a page fault
            - 1) Check PTE to determine if invalid or not, if invalid kill process
            - 2) Otherwise page in the desired page
                - Find a free frame in memory
                - Initiate disk I/O to read in the desired page into the new frame
                - When I/O is finished, modify PTE, make page valid
                - Restart process at faulting instruction

- This is **pure demand paging**
  - Never brings in a page until required
  - Hence, many real systems explicitly load some core parts of the process first.
- Issues
  - 1) When paging in from disk we need a free frame of physical memory to hold the data we're reading from
    - In reality the size of physical memory is limited so we either
      - Discard unused pages if total demand for pages exceeds physical memory size
      - Swap out an entire process to free some frames
  - 2) Requires care to save process state correctly on fault
  - 3) Can we particularly awkward on a CPU with pipelined decode as we need to wind back
  - 4) Even worse on CISC CPU where a single instruction can move lots of data – can't restart the instruction, **therefore rely upon help from microcode**
    - Can possibly use temporary registers to store moved data
  - 5) Similar difficulties with auto-increment / auto-decrement instructions
  - 6) Can even have instructions and data spanning pages, so multiple faults per instruction
    - **Locality of reference makes this unlikely**
- Modified Algorithm for page fault
  - 1) Locate the desired replacement page on disk
  - 2) To select a free frame for the incoming page
    - (a) if there is a free frame, use it
    - (b) Otherwise select a victim page to free
    - (c) Write this page to disk and mark it as invalid in Process Page Tables
  - 3) Read desired page into now free frame
  - 4) Restart faulting process

Page Replacement

We can reduce the overhead by adding dirty bit to the PTEs

- Only write out page to disk if it was modified or marked read-only

Picking Victim Page:

- Want to avoid getting rid of a page when we'd need it in a few instructions time
- Ensure that we minimise the number of page faults

Page Replacement Algorithms

1. First-in First-out (FIFO)
   a. Keep queue of pages and discard from the head.
   b. The performance is hard to predict as we've no idea whether the replaced page will be used again or not
   c. In practise, very simple, but very bad:
      i. Can actually end up discarding a page currently being used.

        ii. Possible to have more faults with increasing number of frames – **Belady's Anomaly**
2. Optimal Algorithm (OPT)
    a. Replace the page which will not be used for the longest period of time
    b. Optimal – serves as baseline, but very hard / impossible to implement
3. Least Recently Used (LRU)
    a. Replace the page which has not been used for the longest amount of time
    b. Equivalent to OPT with the time running backwards
        i. Assuming that the past is a good predictor of the future.
    c. However, can still end up replacing pages that are about to be used.
    d. Generally considered quite good as an algorithm, but hard to implement
4. Least Frequently Used
    a. Replace page with smallest count
    b. Takes no time information into account
    c. Page can stick in memory from initialisation
    d. Need to periodically decrement counts
5. Most Frequently Used
    a. Replace highest count page
    b. Low counts indicate recently brought in

Implementing LRU

1. Counters
    a. Give each PTE a time-of-use field and give the CPU a logical clock
    b. Whenever a page is referenced, its PTE is updated to clock value
    c. Replace page with smallest time value
    d. Problems
        i. Requires a search to find minimum counter value
        ii. Adds a write to memory (PTE) on every memory reference
        iii. Must handle a clock overflow
    e. It's generally impractical on a standard processor
2. Page Stack
    a. Maintain a stack of pages (doubly linked list) with most-recently (MRU) page on top
    b. Discard from the bottom of the stack
    c. Problems
        i. Requires changing up to 6 pointers per new reference
        ii. Very slow without extensive hardware support
    d. Also impractical on a standard processor
3. Approximating LRU
    a. Reference bit in the PTE, zeroed by the OS then set by hardware whenever the page is touched.
    b. After time has passed, consider those with the bit set to be active and implement Not Recently Used replacement.
    c. Periodically, clear all reference bits
    d. When choosing a victim, prefer to remove pages with clear reference bits
    e. If you have a dirty bit, then you can use that as well:

Referenced?	Dirty?	Comment
no	no	best type of page to replace
no	yes	next best (requires writeback)
yes	no	probably code in use
yes	yes	bad choice for replacement

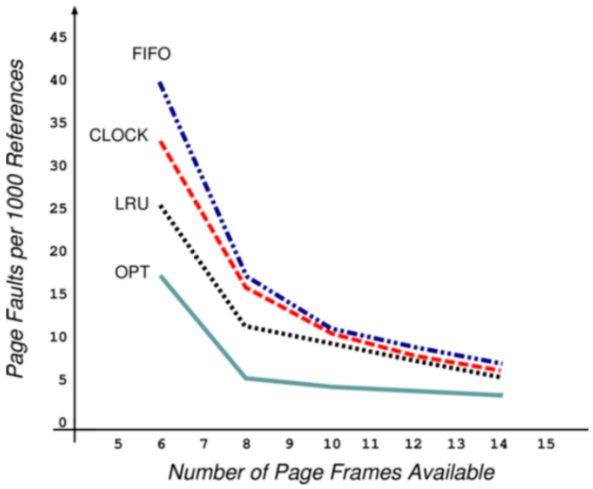
f. **Improvement 1**
   i. Instead of a single bit, OS maintains an 8-bit value per page
   ii. Periodically shift instead of erasing, keeping newest value as the most significant bit
   iii. Then select lowest value page to replace

g. **Improvement 2 (Second-Chance FIFO)**
   i. Store pages in queue as per FIF
   ii. Before discarding head, check reference bit
   iii. If the reference bit is 0, then discard, otherwise, reset the reference bit and give the page a 'second chance' and add it to the tail of queue.
   iv. Guaranteed to terminate after at most one cycle
      1. Worst case is going into a FIFO system.
   v. **Implementing Second-Chance FIFO**
      1. Implemented with a circular queue and a current pointer – in this case it is usually called a clock
      2. If no hardware, then reference bit can emulate
         a. To clear reference bit, mark page as no access
         b. If reference, then trap update PTE and resume
         c. To check if referenced, check the permissions
         d. Can use a similar scheme to emulate a modified bit.

Page Buffering Algorithms

- Allows us to keep a minimum number of victims in a free pool
- A new page is read in before writing out the victim, allowing a quicker restarting of the process
- Alternative
  - If the disk is idle, write modified pages out and reset dirty bit
  - Improves the chance of replacing without having to write a dirty bit.
- This is pseudo MRU (Most recently used)
  - The page to replace is one application has just finished with
  - Track page faults and look for sequences
  - Discard the kth in victim sequence
- **Application Specific:**
  - Provide hook for application for application to suggest which page to replace.

Performance



The plot shows page-fault rate against the number of physical frames for a pseudo-local reference string. We want to minimise the area under the curve. FIFO could exhibit Belody's Anomaly (doesn't here). We can see that getting frame allocation right has major impact right has a major impact – much more than which algorithm you use.

Frame Allocation

A certain fraction of physical memory is reserved per-process and for core OS code and data. We need an allocation policy to determine how to distribute the remaining frames.

Allocation objectives:

1. Fairness
2. Minimise system-wide page-fault rate
3. Maximise level of multiprogramming

Could also allocate taking process priorities into account, since high priority processes are supposed to run more readily. Could care which frames we give to which processes – **page colouring**.

Global Schemes

- Most page replacement schemes are global – all pages are considered for replacement
    - Allocation policy implicitly enforced during page-in
    - Allocation succeeds if and only if the policy agrees
- For example, on a system with 64 frames and 5 processes
    - If using a fair share, each process will have 12 frames, with 4 left over
    - When a process dies, when the next faults it will succeed in allocating a frame
    - Eventually all will be allocated
    - If a new process arrives, we need to steal some pages back from existing allocations

Comparison to Local

- Also get **local page replacement schemes:** The victim is also chosen from within the process

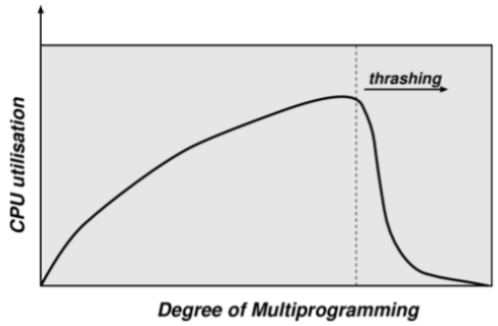
- In global schemes, the process cannot control its own page fault rate, so performance may depend entirely on what other processes page in / out.
- In local schemes, the performance depends only on process behaviour, but this can hinder progress by not making available less / unused pages of memory
- Global schemes are more optimal for throughput and are the most common.

Locality of Reference

In a short time interval, the locations referenced by a process tend to be grouped into a few regions in its address space

1. Procedures being executed
2. Sub-procedures
3. Data access
4. Stack variables

Thrashing



More processes entering the system causes the frames-per-process allocated to reduce. Eventually, we hit a wall with processes stealing other processes' frames, but then need them back so fault. Ultimately, the number of runnable processes plunges.

A process can technically run with minimum-free frames available but will have a very high page fault rate. If we're very unlucky, OS monitors CPU utilisation and increases the level of multiprogramming if utilisation is too low – therefore the machine will die.

Avoiding Thrashing

1. We can avoid thrashing by giving processes as many frames as they need, and if we can't we have to reduce the MPL (multiprogramming amounts) – a better page-replacement algorithm won't help.
2. We can use the locality of reference principle to help determine how many frames a process needs
    a. Define a **working set**
        i. Set of pages that a process needs in store at the same time to make any progress
    b. Varies between processes and during execution
    c. Assume process moves between phases
    d. In each phase, get locality of reference
    e. From time to time, get phase shift.

Calculating Working Set

- Sample page reference bits every 10ms
- Define window size Δ if most recent page references
- If a page is in use, say it's in the working set
- Gives an approximation to the locality of the program
- Given the size of the working set for each process WSS_i, sum working set sizes to get total demand D
- If D > size, we are in danger of thrashing, therefore we should suspend a process

This optimises the CPU utilisation but has the need to compute the size of the working set. We can approximate it with a periodic timer and some page reference script. After some number of intervals, we can consider pages with a count > 0 as in the working set.

In general, a working set can be considered as a scheme to determine allocation for each process.

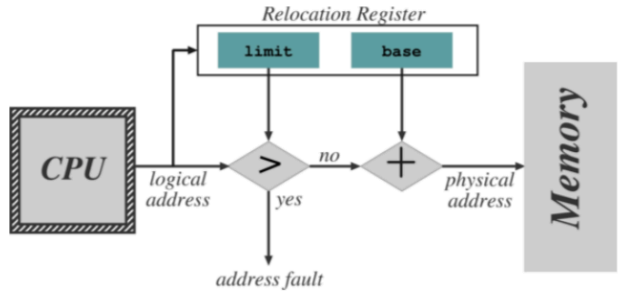
Pre-Paging

- Pure demand paging causes a large number of Page Frames when the process starts
- Can remember the working set for a process, and pre-page the required frames when the process is resumed
- When the process is started, we can pre-page by adding its frames to the free list.
- Increases IO cost: How do we select a page size (given no hardware constraints)?

Page Sizes

- Trade-off between the size of the PT and the degree of fragmentation as a result
- Typical values are between 512B and 16kB
    - Should we reduce the number of faults, or ensure that the window is covered efficiently
- Larger page size means that there are fewer page faults
    - Historical trend towards larger pages

Segmentation



Segmentation is an alternative to paging:

- Memory is viewed as a set of segments of no particular size with no particular ordering
- Corresponds to typical modular approaches taken to program development
- The length of a segment depends on the complexity of the function

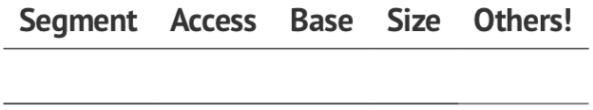
A segment supports the user-view of memory that the logical address space becomes a collection of typically disjoint segments. Segments have a **name** (or a number) and a length. Addresses specify segment and offset within the segment.

To access the memory, the user program specifies the segment + offset and the compiler translates.

This contrasts with paging where the user is unaware of memory structure – everything is managed invisibly by the OS.

Implementing Segments

- The logical pairs are (segment, offset)
- Compiler may construct different segments for global variables, procedure call stack, code for each procedure / function, local variables for each procedure / function
- Finally, the loader takes each segment and maps it to a physical segment number
- We maintain a **Segment Table for each process**



    - If there are too many segments, then the table is kept in memory pointed to by the **Segment Table Base Register (STBR)**
    - We also have a **Segment Table Length Register (STLR)** since the number of segments used by different programs will diverge widely
    - ST is part of the process context and therefore is changed on a process switch
    - Logically accessed on each memory reference, therefore speed is critical
- **Algorithm**
    1. Program presents address (s, d)
    2. If s ≥ STLR then give up
    3. Obtain the table entry at reference s + STBR, which is a tuple of form (bs, ls)
    4. If 0 ≤ d < l, then this is a valid address at location (bs, d) otherwise trap
        - Concatenation and checking validity of d can be done simultaneously to save time
        - Still requires 2 memory references per lookup
            - Could use some kind of buffer

Protection and Sharing

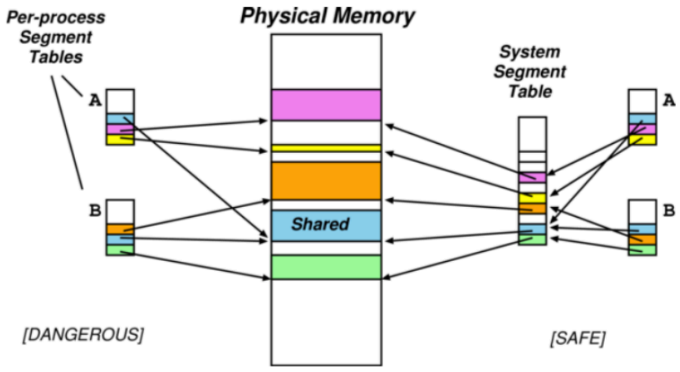
Protection

- The big advantage with segmentation is to provide protection between components
    - The protection is provided per segment
- Protection bits associated with each Segment Table Entry are checked as with page table protection bits
- We could go even further and make it easier by doing things like placing the array in its segment so that the array limits are checked by hardware.

Sharing

Segmentation also facilitates sharing of code / data:

- Each process has its own STBR / STLR
- Sharing is enabled when two processes have entries for the same physical locations
- Sharing occurs at segment level – with each segment having own protection bits
    - For data segments, can use copy-on-write as for paging
- Can share only parts of programs
- **Subtleties**
    - Example: There are jumps within the shared code
        - A jump is specified as a condition + transfer address (segment, offset)
        - Segment is this one
        - Therefore, all programs using the segment must use the same number to refer to it, otherwise confusion
        - As number of users increases, so does difficulty of finding a common shared segment number
        - Therefore, specify branches as PC-relative or relative to a register containing the current segment number
        - (Read only segments containing no pointers may be shared between different segments)



    - Wasteful to store common information on shared segment in each process segment table
    - Assign each segment a unique **Shared Segment Number (SSN)**
    - **Process Segment Table** maps from a Process Segment Number to System Segment Number.

External Fragmentation

- Long term scheduler must find spots in memory for all segments of a program
- Segments are various size, therefore we must handle fragmentation
    - 1) Usually resolved with best / first fit algorithm
    - 2) External fragmentation may cause process to have to wait for sufficient space
    - 3) Compaction can be used in cases where the process would be delayed
- Trade-off between compaction and delay depends on average segment size
    - Each process has one segment reduced to various sized partitions
    - Fixed size small segments equivalent to paging
    - Generally, with small average sizes, external fragmentation is small.
    - But larger segments means that fewer memory lookups through the segment tables.

Segmentation vs Paging

- Protection, Sharing, Demand are all per segment or page, depending on scheme
- For protection and sharing, it is easier to have it per logical entity – segment
- For allocation and demand access, paging is better
    - Allocation is easier
    - Cost of sharing / demand loading is minimised

	Logical View	**Allocation**
Segmentation	Good	Bad
Paging	Bad	Good

Combining Segmentation and Paging

1. **Paged Segments**
    a. Divide each segment into $k = \left\lceil \frac{l_i}{2^n} \right\rceil$ pages, where l_i is the limit (length) of the segment
    b. One page table per segment
    c. However, high hardware cost and complexity, therefore not very portable
2. **Software Segments**
    a. Consider pages [m, …, m+l] to be a segment
    b. OS must ensure protection and sharing kept consistent over region
    c. But leads to loss of granularity
    d. However, relatively simple and portable

Can generally do segments using segmentation hardware, but hard to do software paging with segmentation hardware.

Memory Summary

Direct access to memory is hard to handle:

1. Contiguous Allocation: hard – leads to external fragmentation
2. Address binding: handling absolute addressing
3. Portability: how much memory exists

We avoid these problems by separating the virtual (logical) memory and physical addresses. The mapping between these is handled by the MMU. It makes mapping per-process, then:

- Allocation problem split:
    - Virtual address allocation easy
    - Allocate the physical memory behind the scene
- Address binding solved
    - Bind to logical addresses at compile time
    - Bind to real addresses at run time

Modern operating systems use paging hardware and fake segments in software.

Implementation Considerations

1. **Hardware Support**
    a. Simple base register enough for partitioning

   b. Segmentation and paging need large tables
2. **Performance**
   a. Complex algorithms need more lookups per reference plus hardware support
3. **Fragmentation**
   a. Internal and external depending on whether we have fixed or variable allocation units
4. **Relocation**
   a. This solves external fragmentation at a high cost
   b. Logical addresses must be computed dynamically, this doesn't work with load time relocation
5. **Swapping**
   a. Can be added to any algorithm, allowing more processes to access main memory
6. **Sharing**
   a. Increases multiprogramming, but it requires paging or segmentation
7. **Protection**
   a. Always useful – but generally needs some protection bits
   b. Sometimes very necessary to share code / data – need to be able to do this safely

Dynamic Linking and Loading

Dynamic Linking is a new appearance in OSs (80s). It uses shared objects or Dynamically Linked Libraries (Windows). It enables a compiled binary to invoke, at runtime.

If a routine is invoked which is part of the dynamically linked code, this will implemented as a call into a set of stubs (small program routine that substitutes for a longer program that is loaded later). The stubs check if the routine has been loaded. If not, it loads it and then replaces the stub code **by routing.**

If sharing a library, the addressing binding problem must be solved, requiring the OS support (only the OS knows which libraries are being shared among which processes)

Smaller libraries must be stateless or concurrency safe, or copy on write/

Dynamic Linking means there are smaller binaries (on disk and in memory) and increases the flexibility.

Dynamic Loading: routing loaded when first invoked – with the dynamic loader performing relocation on the fly. It is the responsibility of the user to implement loading, but the OS may provide library support.

Input / Output

IO Subsystem

Input / Output

IO Hardware

Very wide range of devices that interact with the computer via IO:

1. Human Readable

    a. Graphical displays, keyboard, mouse, printers
2. Machine Readable
    a. Disks, tapes, CD, sensors
3. Communications
    a. Modems, network interfaces, radios

All have their own specifications:

- **Data rate**
- **Control complexity**
- **Transfer unit and direction:** blocks vs characters vs frame stores
- **Data representation**
- **Error Handling**

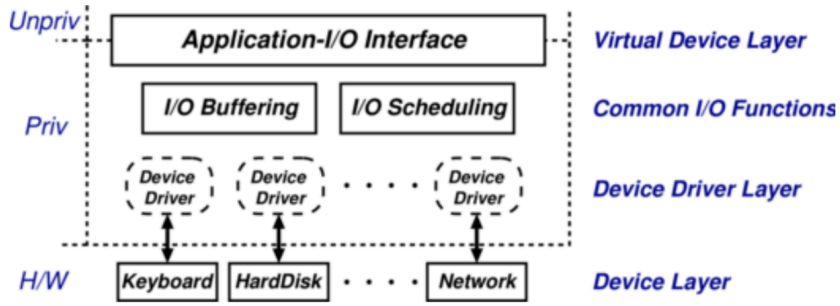
IO Variety

1. **Variety in Devices**
2. **Variety in Applications**
3. **Other dimensions of variation**
    a. Character-stream or block
    b. Sequential or random-access
    c. Synchronous or asynchronous
    d. Shareable or dedicated
    e. Speed of operation
    f. Read-write, read-only or write-only

All the variety in the IO leads to the IO subsystem being the 'messiest' part of the OS. It is impossible to completely homogenise the device API. Therefore, we split the OS into four classes:

1. Block Devices
    a. Commands include read, write, seek
    b. We can have raw access, or via the filesystem or memory-mapped
2. Character Devices
    a. Commands include get and put
    b. We layer libraries on top for line editing
3. Network Devices
    a. Vary enough from block and character devices to get their own interfaces
    b. Unix and Windows NT use the Berkeley Socket interface
4. Miscellaneous
    a. Current time, elapsed time, timers, clocks
    b. (Unix) ioctl covers other odd aspects of the IO

OS Interfaces



Programs access virtual devices:

- terminal streams not terminals
- windows not frame buffers
- event streams not raw mouse
- files not disk blocks

The OS handles the processor-device interface:

- IO instructions vs memory mapped devices
- IO hardware type
- Polled vs Interrupt driven
- CPU interrupt mechanism

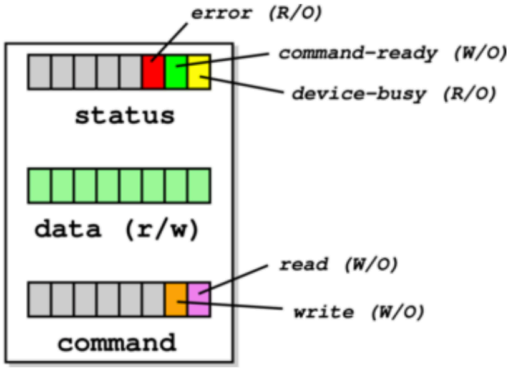
Virtual devices then implemented:

1. In kernel
    a. Files
    b. Terminal devices
2. In daemons
    a. Spooler
    b. Windowing
3. In libraries
    a. Terminal screen control
    b. Sockets

Performing IO

Polled Mode

Polled mode lets two parties communicate by one polling the other and giving / receiving information. Generally, there are three registers: (1) status, (2) data and (3) command. The host can read and write to all of these individually.



Consider Host and the Device communicating over polled mode:

1. H repeatedly reads device-busy until clear
2. H sets a command bit in the command register (for example the write bit), and puts data into the data register
3. H sets command-ready bit in status register
4. D sees command-ready and sets device-busy
5. D performs write operation
6. D clears command-ready and then clears device-busy

Interrupt Driven

Rather than polling, processors provide an interrupt mechanism to handle mismatch between CPU and device speeds:

- At end of each instruction, processor checks interrupt line for pending interrupt
- If line is asserted then processor
    - 1) Saves PC and processor status
    - 2) Changes processor mode
    - 3) Jumps to a well-known address
- Once interrupt handling done, rti to resume previous
- More complex processors may provide
    - **Multiple priorities of interrupt**
    - **Hardware vectoring of interrupts**
    - **Mode dependent registers**

Handling Interrupts

1. At bottom, interrupt handler
2. At top, N interrupt service routines – per device

The processor-dependent interrupt handler may:

- Save more registers and establish a language environment
- De-multiplex interrupt in software and invoke relevant ISR

Device-dependent IRS:

- For programmed IO: transfer data and clear interrupt
- For DMA devices, acknowledge transfer, request any more pending and signal any waiting processes
- Enter the scheduler and return

Blocking vs Non-Blocking

IO system calls exhibit one of three types of behaviours:

1. Blocking
    a. Process suspended until IO completed
    b. Easy to use and understand
    c. Insufficient for some needs - inefficient
2. Non-blocking
    a. IO call returns as much as available
    b. Returns almost immediately with count of bytes read or written (can be 0)
    c. Can be used for things like UI code

   d. Essentially application-level polled IO
3. Asynchronous
   a. Process runs while IO executes
   b. IO subsystem explicitly signals process when its IO request has completed
   c. Most flexible (and potentially efficient)
   d. But most complex to use

Handling IO

Buffering

Cope with various **impedance mismatches** between devices (speed, transfer size), OS may buffer data in memory. It is useful for smoothing peaks and troughs of data rate, but can't help if on average

- Process demand > data rate (process will take time waiting)
- Data rate > capability of the system (buffers will fill and data will spill)
- Downside: can introduce **jitter** which is bad for real-time

The details are often dictated by device type:

- Character devices often by line
- Network devices have lots of transfer bursts
- Block devices make fixed size transfers and often major user of IO buffer memory

1. **Single Buffering**
   a. OS assigns a single buffer to the user request
      i. The OS performs the transfer moving the buffer to user-space when complete
      ii. Request new buffer for more IO, then reschedule application to consume
         1. **Readahead** or **anticipated input**
      iii. OS must track buffers
      iv. Also affects swap logic – if IO is to same disk as swap device, doesn't make sense to swap process out as it will be behind the now queued IO request.
   b. Performance
      i. Let t be time to input block and c be computation time between blocks
      ii. Without buffering, execution time between blocks is t + c
      iii. With buffering, time is max(c, t) + m where m is the time to move data from buffer to memory
2. **Double Buffering**
   a. Process consumes from one buffer while system fills another
      i. Often used in video rendering
   b. Rough performance comparison: takes max(c, t) therefore
      i. Possible to keep device at full speed if c < t
      ii. If c > t, process will not have to wait for IO
   c. Prevents the need to suspend user process between IO operations
      i. Also explains why two buffers are better than one, twice as big
   d. However, need to manage buffers and processes to ensure that process doesn't consume from partially filled buffer
3. **Circular Buffering**

   a. Allow consumption from the buffer at a fixed rate, potentially lower than the burst rate of arriving data
   b. Circular linked list – FIFO buffer with queue length

Caching: Fast memory holding copy of data for both reads and writes – critical to IO performance

Scheduling: order IO requests in per-device queues

Spooling: Queue output for a device, useful if a device is a single user (can only serve one request at a time)

Device Reservation: System calls for acquiring or releasing exclusive access to a device

Error handling: Some form of error number or code when request fails, logged into system error log

Kernel Data Structures
To manage this, OS kernel must maintain state for IO components:
- Open file tables
- Network connections
- Character device states

This results in many complex and performance critical data structures to track buffers, memory allocation, dirty blocks

In order to read a file from disk for a process:
1. Determine device holding the file
2. Translate the name to device representation
3. Physically read data from disk into buffer
4. Make data available to requesting process
5. Return control to process

Performance
IO is a major factor in system performance:
- Demands CPU to execute device driver, kernel IO code, etc
- Context switches due to interrupts
- Data copying

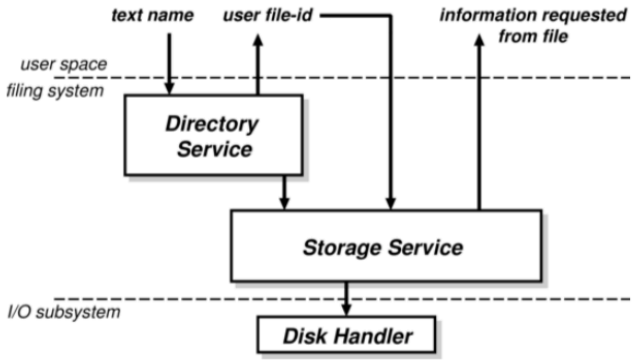
How to improve performance:
1. Reduce number of context switches
2. Reduce data copying
3. Reduce number of interrupts
   a. Large transfers
   b. Smart controller
   c. Polling
4. Use DMA where possible

5. Balance CPU, memory, bus, IO performance for highest throughput

Storage

File Concepts

Filesystems



1. **Directory Service**
    a. Mapping names to file identifiers and handling access and existence control
2. **Storage Service**
    a. Providing mechanism to store data on disk, and including means to implement directory service

What is a file?

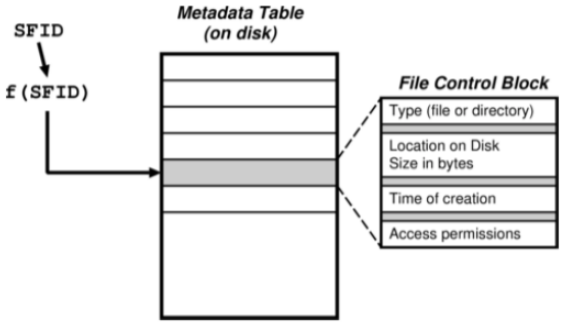
- A basic abstraction for non-volatile storage
- User abstraction (how it's actually stored may differ)
    - Though typically comprises of a single contiguous address space
- Can have a varied internal structure
    - **None** – simple sequence of words or bytes
    - **Simple record structures** – lines, fixed length, variable length
    - **Complex internal structure** – formatted document, relocatable object file
- We can map everything to a byte sequence by inserting appropriate control characters, and interpretation in code. We can decide this:
    - OS – may be easier for the programmer, but this will lack flexibility
    - Programmer – Has to do more work, but we can evolve and develop the format

Naming Files

Two kinds of name:

1. **System File Identifier (SFID):** unique integer value associated with a given file – used within the filesystem itself
2. **Human Name:** what users use to call the file
    a. Hold the mapping from human names to SFID is held in a directory
    b. Mapping from SFID to **File Control Block (FCB)** is OS and filesystem specific
        i. In addition to their contents and their names, files typically have a number of other attributes or metadata
3. *May have a third, **User File Identifier (UID)** used to identify open files in applications*
4. *Directories are also non-volatile, so they may be stored on disk along with the files*
    a. *Storage system is below the directory system*

File Metadata



- **Location:** pointer to file location on device
- **Size:** current file size
- **Type:** needed if system supports different types
- **Protection:** controls who can read, write
- **Time, date** and **user identification:** data for (1) protection, (2) security and (3) usage monitoring

Directories

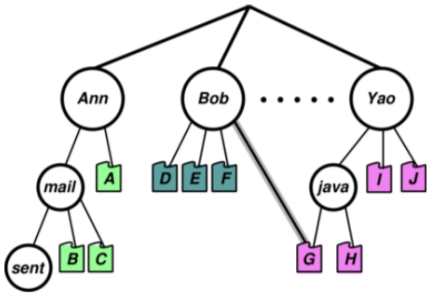
Provides the means to translate a user name to the location of the file on-disk:

Requirements

1. **Efficiency**
    a. Locating a file **quickly**
2. **Naming**
    a. *User convenience*
    b. Also, a number of users to have the same name for different files
    c. Allow one file to have different names
3. **Grouping**
    a. Allow grouping of files by properties

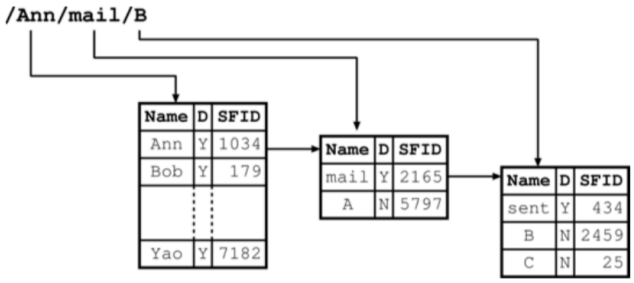
Structure

- Early Attempts
    - **Single-level:** one directory between all users
        - Naming problem
        - Grouping problem
    - **Two-level:** one directory per user
        - Can have the same filename for different user
        - But, still have no grouping capability
    - We add a general hierarchy for more flexibility
- **Tree**
    - Directories hold files or more directories
    - We create or delete files relative to a given directory
    - **Efficient searching and arbitrary grouping capabilities**
    - But, human name is the full path name
        - We can resolve this with **relative naming, login directory, current working directory**
        - Sub-directory deletion either by recursively deleting
- **Directed Acyclic Graph**



- **Only one name** per file
- Allow shared subdirectories and files
- Multiple aliases for the same thing
- **Deletion** (and more generally permissions)
- **Knowing when okay to free disk blocks**
- **Accounting** (who gets charged for disk usage)
- How to prevent cycles

- **Directory Implementation**



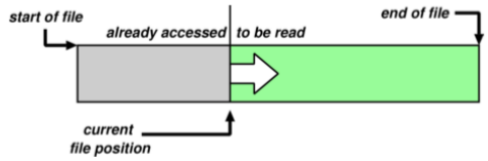
- Consider directories as files on disk – with own SFID
- There must be different types of files, for traversal
- Operations must also be explicit as info in the directory is used for access control
- Explicit directory controls include:
  - 1) Create / delete directory
  - 2) List contents
  - 3) Select current working directory
  - 4) Insert an entry for a file (a link)

Files

Operations

- Basic paradigm is (1) open, (2) use, (3) close
- **Opening**
  - UFID = open(<pathname>)
- **Creating**
  - UFID = create(<pathname>)
- Directory service recursively searches directories for components of <pathname>
- We can eventually get the SFID for the file, from which the UFID is created and returned
- Can use various modes
- **Closing**
  - Status = close(UFID)

Implementation



We associate a cursor or file position with each open file (with its UFID), initialised to the start of the file. We have basic operations: read next, or write next:

- Read(UFID, buf, nbytes)
- Write(UFID, buf, nrecords)

There are a couple of different access patterns:

1. **Sequential**
    a. Adds rewind(UFID) to above
2. **Direct Access**
    a. Read(N) or write(N) using seek(UFID, pos)
3. Others
    a. Append-only
    b. Indexed Sequential Access Mode (ISAM)

Access Control: A File only accessible if user has both directory and file access rights**.** Former is due to lookup process**.** Access control is normally a function of directory service so checks done at file open time.

Existence Control: When a file gets deleted, we want to keep file in existence when valid pathname referencing. Also, we need to check the entire FS periodically for garbage. Finally, Existence control can also be a factor when a file is renamed or moved.

Concurrency Control: Need some sort of locking to handle simultaneous access. This can be mandatory or advisory and the locks can be shared or exclusive. Finally, granularity may be file or subset.

Unix

- First developed in 1969 at Bell Labs as response to bloated Multics.
- Originally written in PDP-7 asm, then rewritten in C so it was easy to port, alter and read. Unusual due to need for performance
- V6 was widely available, including source, meaning people could write new tools and new features of other OSes was promptly rolled in
    - Mainly used by universities who could afford a minicomputer
    - But not necessarily all the software required
    - It was first portable OS
- Bell Labs continued with V8, V9, V10, never really widely available
    - Because V7 pushed to Unix Support Group within AT&T
- AT&T did System III first and in 1983, System V (no system IV)
- V7
    - From 1978
    - Two families – AT&T “System V” **SVR4** and Berkeley **BSD**

        - **Berkeley added virtual memory support and created 3BSD**
- Later, there were standardisation efforts (POSIX, X/OPEN) to homogenise
- USDL did SVR2 in 1984
    - SVR3 in 1987, SVR4 in 1989, supported the POSIX.1 standard
- 4BSD development supported by DARPA
    - OS support for TCP/IP wanted
    - 4.2BSD released at end of original DARPA project (1983)
    - 4.3BSD released with some minor tweaks (1986)
    - 4.3BSD Tahoe included better TCP/IP congestion control (1988)
    - 4.3BSD Reno had further congestion control
    - 4.4BSD had large rewrite
        - Very different structure
        - Includes LFS, Mach VM, stackable FS, NFS
- Unix today mostly used in Linux, also FreeBSD, NetBSD, Solaris

Design Features

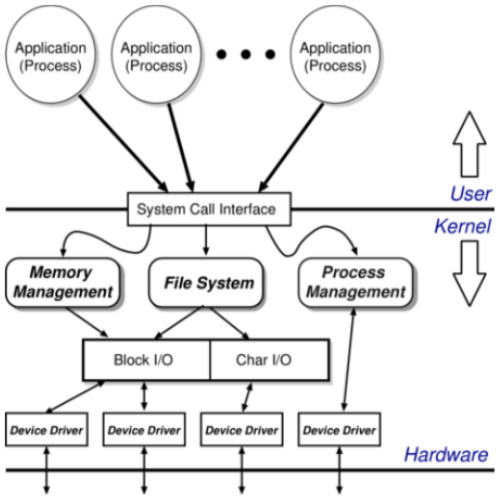
- Hierarchical file system incorporating demountable volumes
- Compatible file, device and inter-process IO (naming schemes, access control)
- Ability to initiate asynchronous processes
- System command language selectable per-user

This is completely novel at the time as everything was inside the OS. In Unix, the separation between essential things (kernel) and everything else. Among other things, this allows a use of wider choice, without increasing the size of the core OS – over 100 subsystems. It is highly portable due to the use of the high-level language.

Features which were not included were real-time features and multiprocessor support

Basic Structure

Clear separation between the user and the kernel portions. Only the essential features inside the OS, not other stuff. Processes are **unit of scheduling** and **protection** – the command interpreter (shell) just a process. There is no concurrency within the kernel. Also, **everything like a file – IO like a file operation**.



Filesystem

Abstraction

- File is an unstructured sequence of bytes – most systems lend towards file being composed of records
- Don't get nice type information and programmer must worry about format of things inside the file
- Less to worry about in kernel and programmer has flexibility
- Represent a file in user-space by a file descriptor (fd) – opaque identifier – good technique for ensuring protection between user and kernel

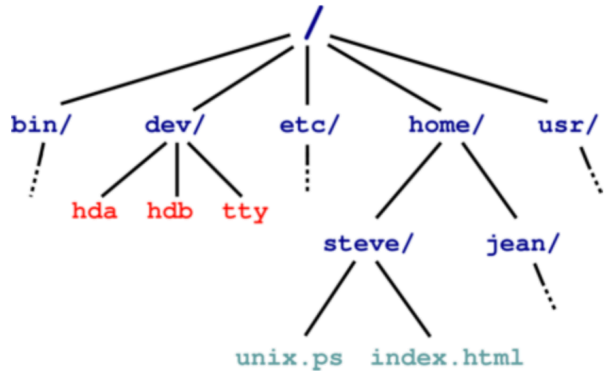
File Operations

1. fd = open(pathname, mode)
2. fd = create(pathname, mode)
3. bytes = read(fd, buffer, nbytes)
4. count = write(fd, buffer, nbytes)
5. reply = seek(fd, offset, whence)
6. reply = close(fd)

The kernel keeps track of the current position within the file – **cursor**. Also, devices are represented by special files:

- These tend to support the above operations with other semantics
- There is also ioctl (input-output control) – device-specific system call

Directory Hierarchy

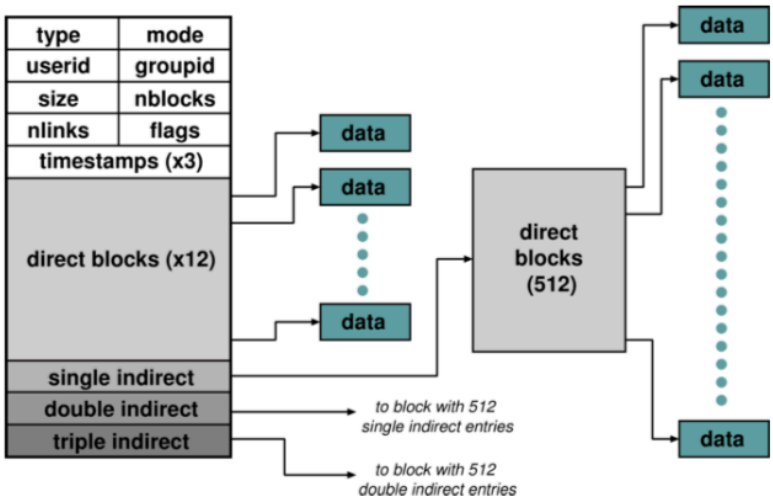


- Directory maps names to files (and directories)
- Starts from **distinguished root directory called /**
- **Fully Qualified Pathnames:** performing traversal from root
- Every directory has a '.' and '..' entries
    - . refers to self
    - .. refers to parent
    - Also, tend to have a shortcut of the current working directory (cwd) which allows relative path names and the shell provides access to home directory as ~username
        - The kernel knows about the current working directory but not necessarily about the username directories

Password File

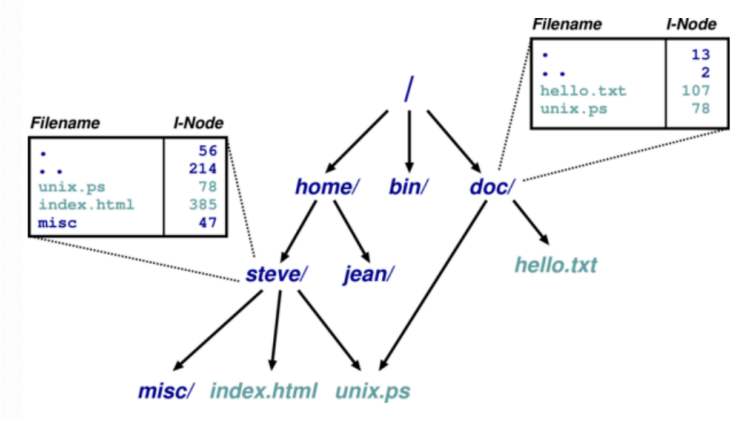
- /etc/passwd holds the list of password entries
    - Form is: username:encryptedPassword:homeDirectory:shell
    - This is publically readable
        - Has useful information
        - But permits offline attack
            - We can instead have a shadow password file
- Also contains user-id, group-id and friendly name
- Uses a one-way function to encrypt password
    - So process is:
        - (1) get user name
        - (2) get password
        - (3) encrypt password
        - (4) check against version in /etc/passwd
        - (5) If ok, initiate login shell
        - (6) Otherwise, delay and retry with upper bound on retries

File System Implementation



Inside kernel – file is represented by an **index-node** (inode) which holds the meta-data

Directories and Links



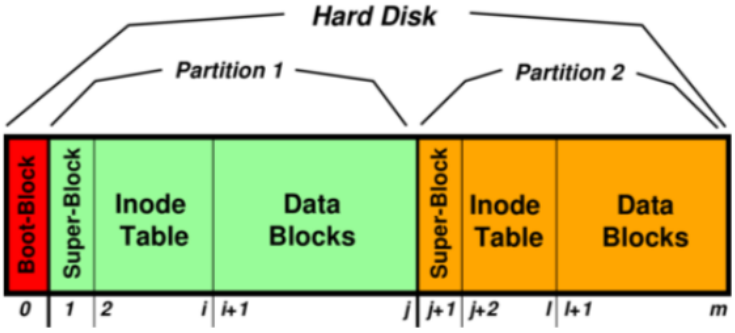
A directory is a file which maps filenames to i-nodes, that is it has its own i-node pointing to its contents. An instance of a file in a directory is a hard link, hence the reference count in the i-node. Directories can have at most 1 real link

Hard Link vs Soft Link

Hard link is a directory entry that directly associated a name with a file on a filesystem. Any copies have the same inode number as the original. Therefore, reference count in the inode goes up – when it is at 0, you can delete it. It is important to note that a directory can have at most 1 real link – since two items cannot have the same inode number.

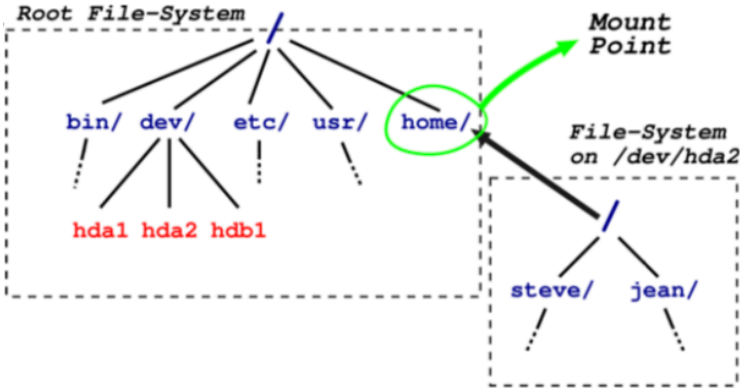
A soft or symbolic link is a file that contains a reference to another file or directory – has the filename with the absolute path – **importantly, this means it can pass mount points!** We create a special file which does not have any content but has information about the file name it links to. Therefore, we have separate metadata for each of the versions of the file.

On-Disk Structures



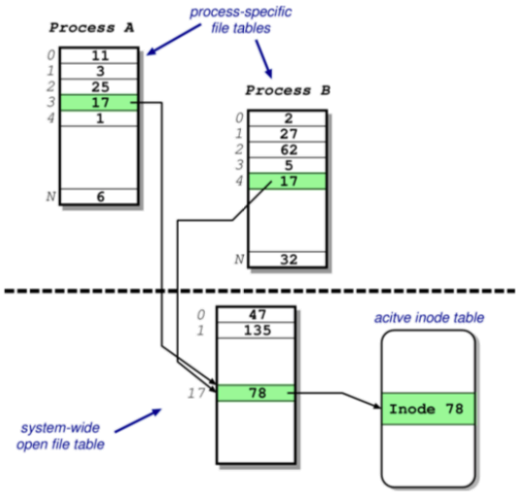
- Disk consists of a boot block followed by one or more **partitions**.
    - Boot block contains a partition table, allowing the OS to determine where the filesystems are
    - **Partition**
        - Contiguous range of fixed-size blocks
        - Unix filesystem resides within a partition
- From figure, important to note, that the size of the inode table is much greater than the size of the super-block and the data blocks is much larger than the inode table
- **Superblock**
    - Number of blocks and free blocks in fs
    - Start of free block and free inode list
    - Various other bookkeeping information
- Free blocks and inodes intermingle with allocated ones
- On-disk, we have a chain of tables (with head in superblock) for each partition
    - Leaves superblock and inode-table vulnerable to head crashes – therefore must replicate in practise
    - In reality, wide range of Unix fs are just completely different – log-structure

Mounting Filesystems



- Filesystems are mounted on an existing directory in an already mounted filesystem
- (Must mount a root filesystem since only / exists in the beginning)

In-Memory Tables



Processes see files as file descriptors. In the implementation, (1) these are just indices into process-specific open file table. (2) Entries point to the system-wide open file table. (3) Finally, these point to the inode table in memory.

Access Control

- The access control information is held in each node, with three bits each for the owner, group and world (r, w, execute). For directories, the equivalent for execute is a 'traverse directory'.
- Also have setiud and setgid bits
    - Allow user to become someone else when running a given program

Consistency Issues

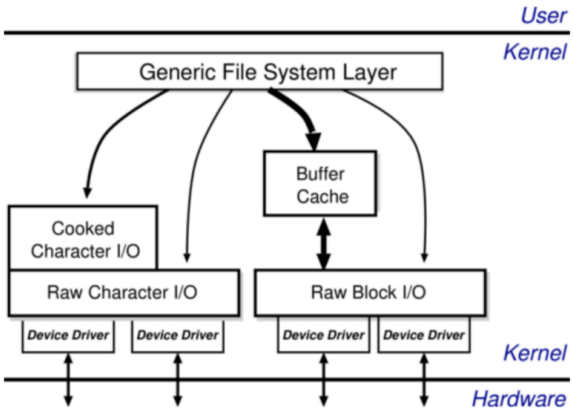
- In order to delete a file, we unlink it (rm <filename>)
- Deletion Procedure
    - 1) Check user has sufficient permissions on (i) file and (ii) directory (write access)
    - 2) Remove entry from directory
    - 3) Decrement reference count on inode
    - 4) If zero, free data blocks and free inode

- If there is a crash, we must check the entire filesystem for any block unreferenced and any block double referenced
    - Detect crash as OS knows if crashed due to root fs not unmounted cleanly

Summary

1. Files are unstructured byte streams
2. Everything is a file
3. Hierarchy built from root
4. Unified name-space (multiple filesystems mounted on any leaf)
5. Disk contains list of inodes and data blocks
6. Processes see file descriptors – map to file tables
7. Permissions for owner, group and world
8. Setuid / setgid allow for more flexible control

IO



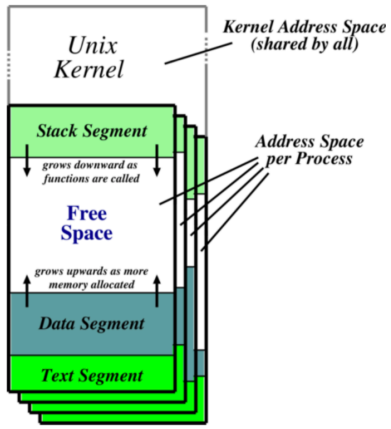
To access IO, we access exactly like accessing files (literally accessed through the file system). There are two broad categories:

1. Character
    a. Character IO is low rate but complex, there most functionality is in the interface
2. Block
    a. Simpler but performance matters – emphasis on the **buffer cache**

Buffer Cache: Keep copy of some parts of the disk in memory

- On Read
    - Locate relevant blocks (from inode)
    - Check if in buffer cache
    - If not, read from disk into memory and return data from the buffer cache
- On write
    - Locate relevant blocks (from inode)
    - Check if in buffer cache
    - If not, read from disk into memory
    - Update version in cache, not on disk
    - **Call sync every 30 seconds to flush dirty buffers to disk**
- We can also cache metadata too

Processes

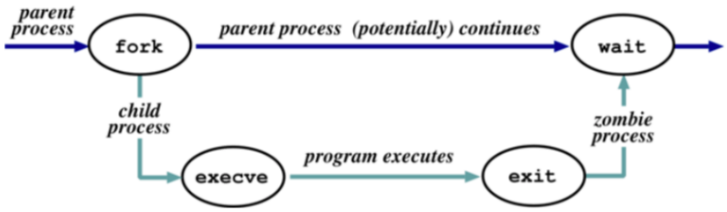


Processes have three segments:

1. Text
    a. Holds the machine instructions for the program
2. Data
    a. Contains variables and their values
3. Stack
    a. Used for activation records
        i. Storing local variables
        ii. Parameters

The process is represented by an opaque process id (pid), organised hierarchically with parents creating children. This creates a copy of the entire address space – inefficient.

- Pid = fork()
- Reply = execve(pathname, argv, envp)
- Exit(status)
- Pid = wait(status)



Startup

- The kernel (/vmunix) is loaded from disk directly and the execution starts
- Mounts file system and the/etc/init process is started
- Reads file /etc/inittab and for each entry
    - Opens terminal special file
    - Duplicates the resulting fd twice
    - Forks an /etc/tty process
- Each tty process next gets carried out
- Next, initialises the terminal and completed login
- If login, then sets uid and gid and execve() shell

- A Patriarch init resurrects /etc/tty on exit

Process Scheduling

- Round robin within priorities (0-127) (quantum of 100ms)
  - User processes have a priority greater than 50
- Priorities are based on usage and nice (**partially user controllable adjustment parameter in the range [-20, 20]**)

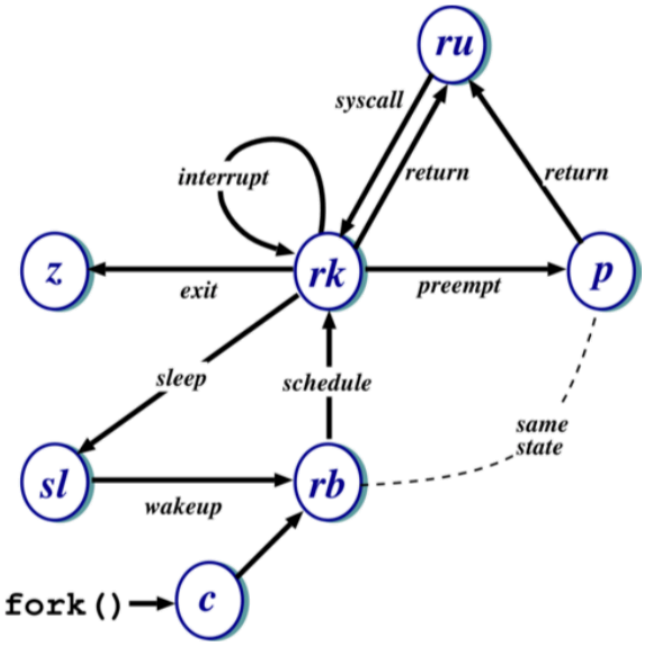
$$P_j(i) = \text{Base}_j + \frac{\text{CPU}_j(i-1)}{4} + 2 \times \text{nice}_j$$

- Where:

$$\text{CPU}_j(i) = \frac{2 \times \text{load}_j}{(2 \times \text{load}_j) + 1}\text{CPU}_j(i-1) + \text{nice}_j$$

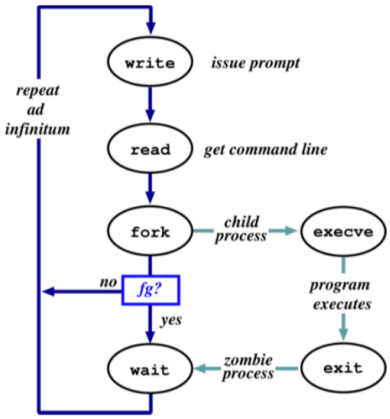
- And load_j is the sampled average length of the run queue in which process j resides, over the last minute of operation.
- This if the load is 1, 90% of 1s CPU usage is forgotten with 5s
- Base Priority divides processes into bands
  - CPI and components prevent processes moving out of their bands
  - Bands are
    - 1) Swapper
    - 2) Block IO device control
    - 3) File manipulation
    - 4) Character IO device control
    - 5) User processes
  - Within the user process band, the execution history tends to penalise CPU bound processes, aiding IO bound processes

Simplified Process States



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

The Shell



- Simply a process – doesn't necessarily need to understand commands, just files.
- It uses the path for convenience to avoid needing fully qualified pathnames.
- The parsing stage can do lots of things, for example wildcard expansion and tilde processing
- '&' specifies background
- '|' separates commands

- Every process has 3 file descriptors on creation
  1. Stdin: where to read input from
  2. Stdout: where to send output
  3. Stderr: where to send diagnostics
- These are normally inherited from the parent, the shell allows redirection to/from a file
- Unix commands are often filters – used to build complex command lines
- **Redirection can cause some buffering subtleties**

Main Unix Features

1. **File Abstraction**
   a. File unstructured sequence of bytes
   b. Plus, special files
2. **Hierarchical Namespace**
   a. DAG
   b. Thus, recursively mount filesystems
3. **Heavy-weight processes**
4. **IO: Block and character**
5. **Dynamic priority scheduling**
   i. Base Priority for all processes
   ii. Priority lowered if process gets to run
   iii. Over time, past is forgotten
6. **Inflexible IPC, Inefficient Memory Management and Poor Kernel Concurrency**
   a. In V7
   b. Late versions fix this