Operating Systems

Steven Hand

Michaelmas Term 2010

12 lectures for CST IA

Course Aims

- This course aims to:

  - explain the structure and functions of an operating system,
  - illustrate key operating system aspects by concrete example, and
  - prepare you for future courses. . .

- At the end of the course you should be able to:

  - compare and contrast CPU scheduling algorithms
  - explain the following: process, address space, file.
  - distinguish paged and segmented virtual memory.
  - discuss the relative merits of Unix and NT. . .

Course Outline

- Introduction to Operating Systems.

- Processes & Scheduling.

- Memory Management.

- I/O & Device Management.

- Protection.

- Filing Systems.

- Case Study: Unix.

- Case Study: Windows NT.

Recommended Reading

- *Concurrent Systems* or *Operating Systems*
  Bacon J [and Harris T], Addison Wesley 1997 [2003]

- *Operating Systems Concepts (5th Ed.)*
  Silberschatz A, Peterson J and Galvin P, Addison Wesley 1998.

- *The Design and Implementation of the 4.3BSD UNIX Operating System*
  Leffler S J, Addison Wesley 1989

- *Inside Windows 2000 (3rd Ed)* or *Windows Internals (4th Ed)*
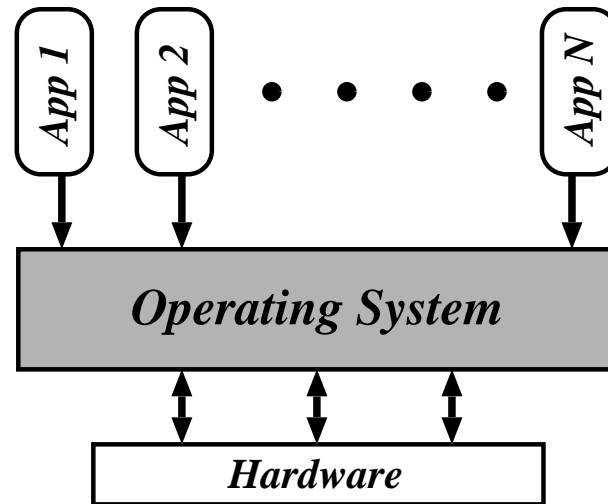 Solomon D and Russinovich M, Microsoft Press 2000 [2005]

What is an Operating System?

- A program which controls the execution of all other programs (applications).

- Acts as an intermediary between the user(s) and the computer.

- Objectives:

  - convenience,
  - efficiency,
  - extensibility.

- Similar to a government. . .

An Abstract View

- The Operating System (OS):

  - controls all execution.
  - multiplexes resources between applications.
  - abstracts away from complexity.

- Typically also have some *libraries* and some *tools* provided with OS.

- Are these part of the OS? Is IE a tool?

  - no-one can agree. . .

- For us, the OS ≈ the *kernel*.

In The Beginning. . .

- 1949: First stored-program machine (EDSAC)

- to $\sim$ 1955: "Open Shop".

  - large machines with vacuum tubes.
  - I/O by paper tape / punch cards.
  - user = programmer = operator.

- To reduce cost, hire an *operator*:

  - programmers write programs and submit tape/cards to operator.
  - operator feeds cards, collects output from printer.

- Management like it.

- Programmers hate it.

- Operators hate it.
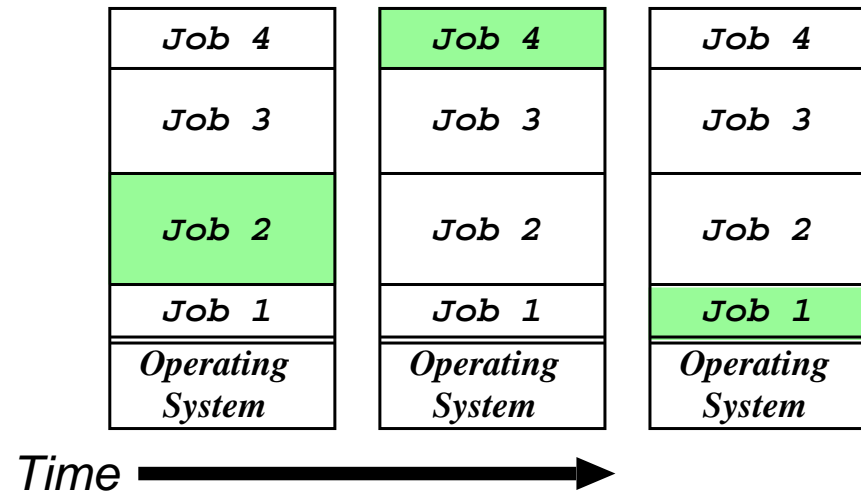
$\Rightarrow$ need something better.

Batch Systems

- Introduction of tape drives allow *batching* of jobs:

  - programmers put jobs on cards as before.
  - all cards read onto a tape.
  - operator carries input tape to computer.
  - results written to output tape.
  - output tape taken to printer.

- Computer now has a *resident monitor*:

  - initially control is in monitor.
  - monitor reads job and transfer control.
  - at end of job, control transfers back to monitor.

- Even better: *spooling systems*.

  - use interrupt driven I/O.
  - use magnetic disk to cache input tape.
  - fire operator.

- Monitor now *schedules* jobs. . .

Multi-Programming

Job 4		Job 4		Job 4
Job 3		Job 3		Job 3
Job 2		Job 2		Job 2
Job 1		Job 1		Job 1
Operating System		*Operating System*		*Operating System*

Time ➡

- Use memory to cache jobs from disk $\Rightarrow$ more than one job active simultaneously.

- Two stage scheduling:

  1. select jobs to load: *job scheduling*.
  2. select resident job to run: *CPU scheduling*.

- Users want more interaction $\Rightarrow$ *time-sharing*:

- e.g. CTSS, TSO, Unix, VMS, Windows NT. . .

Today and Tomorrow

- Single user systems: cheap and cheerful.

  - personal computers.
  - no other users $\Rightarrow$ ignore protection.
  - e.g. DOS, Windows, Win 95/98, . . .

- RT Systems: power is nothing without control.

  - hard-real time: nuclear reactor safety monitor.
  - soft-real time: mp3 player.

- Parallel Processing: the need for speed.

  - SMP: 2–8 processors in a box.
  - MIMD: super-computing.

- Distributed computing: global processing?

  - Java: the network is the computer.
  - Clustering: the network is the bus.
  - CORBA: the computer is the network.
  - .NET: the network is an enabling framework. . .

Monolithic Operating Systems
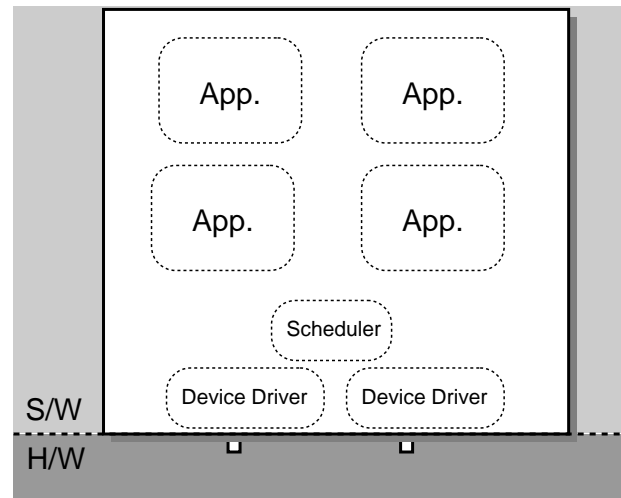
- Oldest kind of OS structure ("modern" examples are DOS, original MacOS)

- Problem: applications can e.g.

    - trash OS software.
    - trash another application.
    - hoard CPU time.
    - abuse I/O devices.
    - etc. . .

- No good for fault containment (or multi-user).

- Need a better solution. . .

Dual-Mode Operation

- Want to stop buggy (or malicious) program from doing bad things.

$\Rightarrow$ provide *hardware* support to distinguish between (at least) two different modes of operation:

1. *User Mode* : when executing on behalf of a user (i.e. application programs).
2. *Kernel Mode* : when executing on behalf of the operating system.

- Hardware contains a mode-bit, e.g. 0 means kernel, 1 means user.

interrupt or fault

reset   *Kernel Mode*   *User Mode*

set user mode
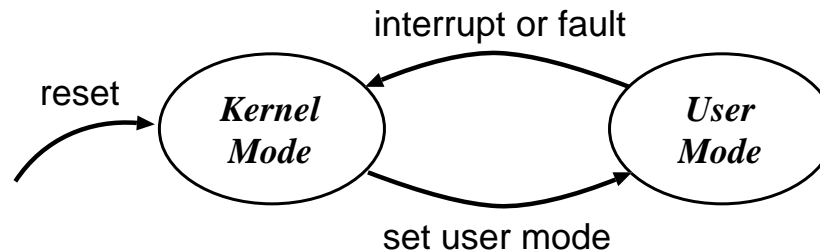
- Make certain machine instructions only possible in kernel mode. . .
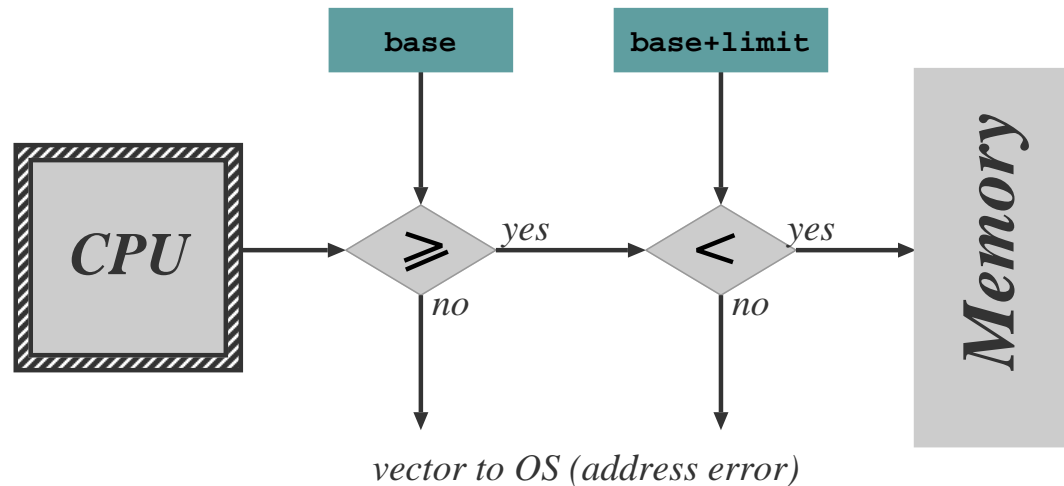
Protecting I/O & Memory

- First try: make I/O instructions privileged.

    - applications can't mask interrupts.
    - applications can't control I/O devices.

- But:

    1. Application can rewrite interrupt vectors.
    2. Some devices accessed via *memory*

- Hence need to protect memory also, e.g. define $base$ and $limit$ for each program:
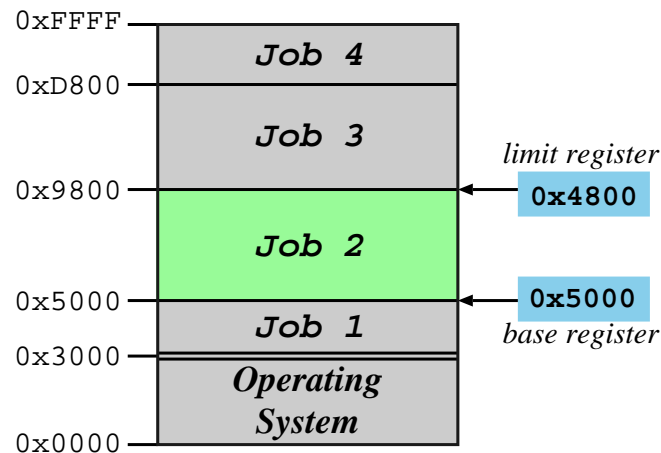
- Accesses outside allowed range are protected.

Memory Protection Hardware

- Hardware checks every memory reference.

- Access out of range $\Rightarrow$ vector into operating system (just as for an interrupt).

- Only allow *update* of base and limit registers in kernel mode.

- Typically disable memory protection in kernel mode (although a bad idea).

- In reality, more complex protection h/w used:
  - main schemes are *segmentation* and *paging*
 - (covered later on in course)

Protecting the CPU

- Need to ensure that the OS stays in control.

    - i.e. need to prevent any a malicious or badly-written application from 'hogging' the CPU the whole time.
    $\Rightarrow$ use a $timer$ device.

- Usually use a $countdown$ timer, e.g.

    1. set timer to initial value (e.g. 0xFFFF).
    2. every $tick$ (e.g. $1\mu s$), timer decrements value.
    3. when value hits zero, interrupt.

- (Modern timers have programmable tick rate.)

- Hence OS gets to run periodically and do its stuff.

- Need to ensure only OS can load timer, and that interrupt cannot be masked.

    - use same scheme as for other devices.
    - (viz. privileged instructions, memory protection)

- Same scheme can be used to implement time-sharing (more on this later).

Kernel-Based Operating Systems
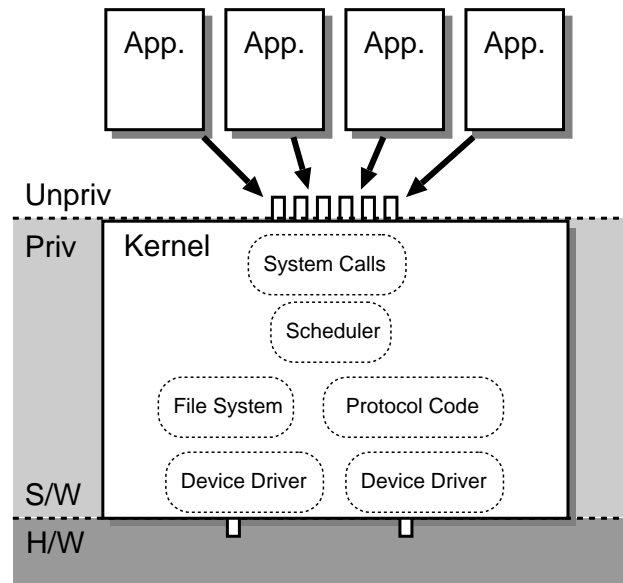
- Applications can't do I/O due to protection

  $\Rightarrow$ operating system does it on their behalf.

- Need secure way for application to invoke operating system:

  $\Rightarrow$ require a special (unprivileged) instruction to allow transition from user to kernel mode.

- Generally called a *software interrupt* since operates similarly to a real (hardware) interrupt. . .

- Set of OS services accessible via software interrupt mechanism called *system calls*.

Microkernel Operating Systems
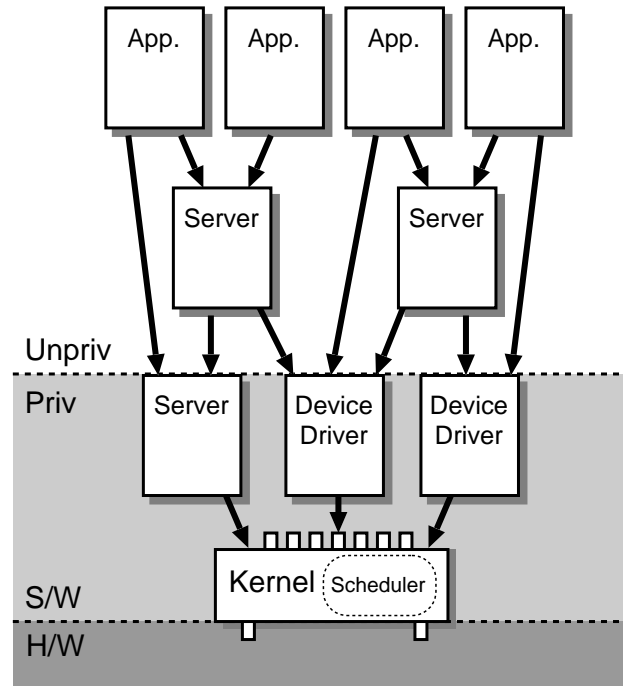
- Alternative structure:

  – push some OS services into *servers*.
  – servers may be privileged (i.e. operate in kernel mode).

- Increases both *modularity* and *extensibility*.

- Still access kernel via system calls, but need new way to access servers:

  ⇒ interprocess communication (IPC) schemes.

Kernels versus Microkernels

So why isn't everything a microkernel?

- Lots of IPC adds overhead

  $\Rightarrow$ microkernels usually perform less well.

- Microkernel implementation sometimes tricky: need to worry about concurrency and synchronisation.

- Microkernels often end up with redundant copies of OS data structures.

Hence today most common operating systems blur the distinction between kernel and microkernel.

- e.g. linux is a "kernel", but has kernel modules and certain servers.

- e.g. Windows NT was originally microkernel (3.5), but now (4.0 onwards) pushed lots back into kernel for performance.

- Still not clear what the best OS structure is, or how much it really matters. . .

Operating System Functions

- Regardless of structure, OS needs to *securely multiplex resources*:

  1. protect applications from each other, yet
  2. share physical resources between them.

- Also usually want to *abstract* away from grungy harware, i.e. OS provides a *virtual machine*:

  – share CPU (in time) and provide each app with a virtual processor,
  – allocate and protect memory, and provide applications with their own virtual address space,
  – present a set of (relatively) hardware independent virtual devices,
  – divide up storage space by using filing systems, and
  – do all this within the context of a security framework.

- Remainder of this part of the course will look at each of the above areas in turn. . .

Process Concept

- From a user's point of view, the operating system is there to execute programs:

  - on batch system, refer to *jobs*
  - on interactive system, refer to *processes*
  - (we'll use both terms fairly interchangeably)

- Process $\neq$ Program:

  - a program is *static*, while a process is *dynamic*
  - in fact, a process $\stackrel{\triangle}{=}$ "a program in execution"

- (Note: "program" here is pretty low level, i.e. native machine code or *executable*)

- Process includes:

  1. program counter
  2. stack
  3. data section

- Processes execute on *virtual processors*

Process States
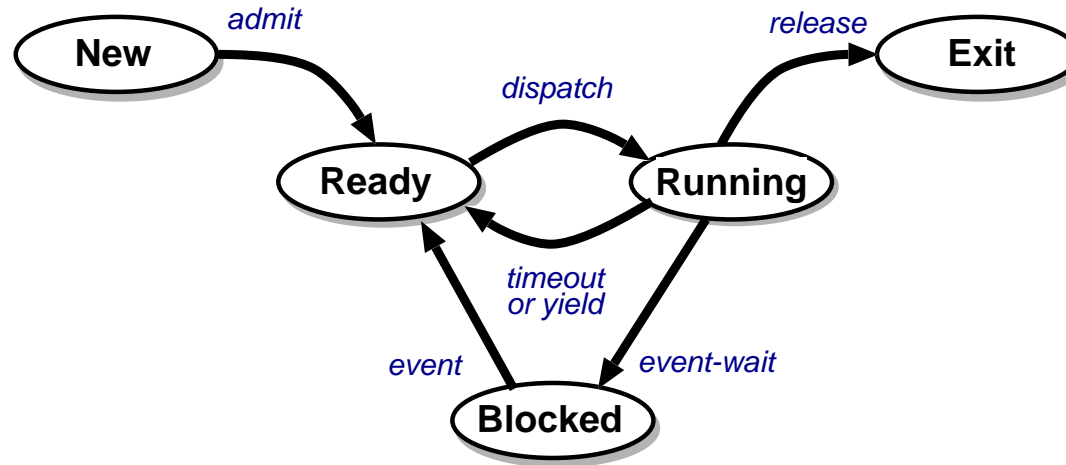
- As a process executes, it changes *state*:

  - New: the process is being created

  - Running: instructions are being executed

  - Ready: the process is waiting for the CPU (and is prepared to run at any time)

  - Blocked: the process is waiting for some event to occur (and cannot run until it does)

  - Exit: the process has finished execution.

- The operating system is responsible for maintaining the state of each process.

Process Control Block

| |
|---|
| **Process Number (or Process ID)** |
| **Current Process State** |
| **CPU Scheduling Information** |
| **Program Counter** |
| **Other CPU Registers** |
| **Memory Mangement Information** |
| **Other Information**
(e.g. list of open files, name of
executable, identity of owner, CPU
time used so far, devices owned) |
| **Refs to previous and next PCBs** |
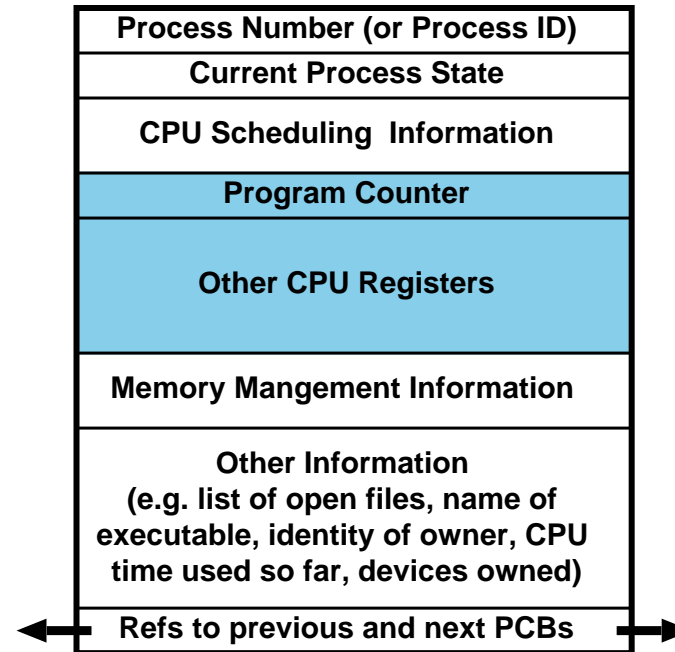
OS maintains information about every process in a data structure called a *process control block* (PCB):

- Unique process identifier
- Process state (*Running*, *Ready*, etc.)
- CPU scheduling & accounting information
- Program counter & CPU registers
- Memory management information
- . . .

Context Switching
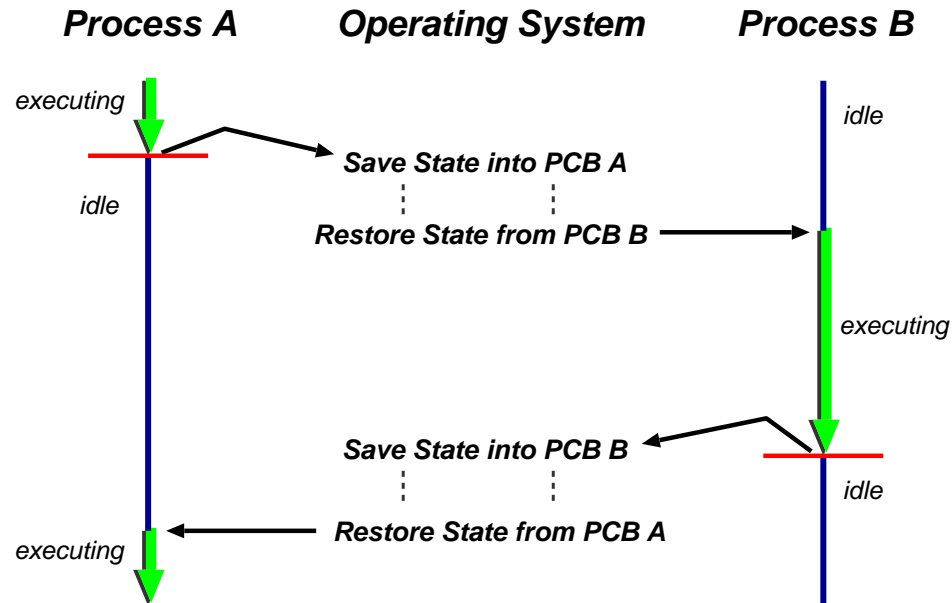
- *Process Context* = machine environment during the time the process is actively using the CPU.

- i.e. context includes program counter, general purpose registers, processor status register (with $\mathbf{C}, \mathbf{N}, \mathbf{V}$ and $\mathbf{Z}$ flags), . . .

- To switch between processes, the OS must:

  a) save the context of the currently executing process (if any), and
  b) restore the context of that being resumed.

- Time taken depends on h/w support.

Scheduling Queues
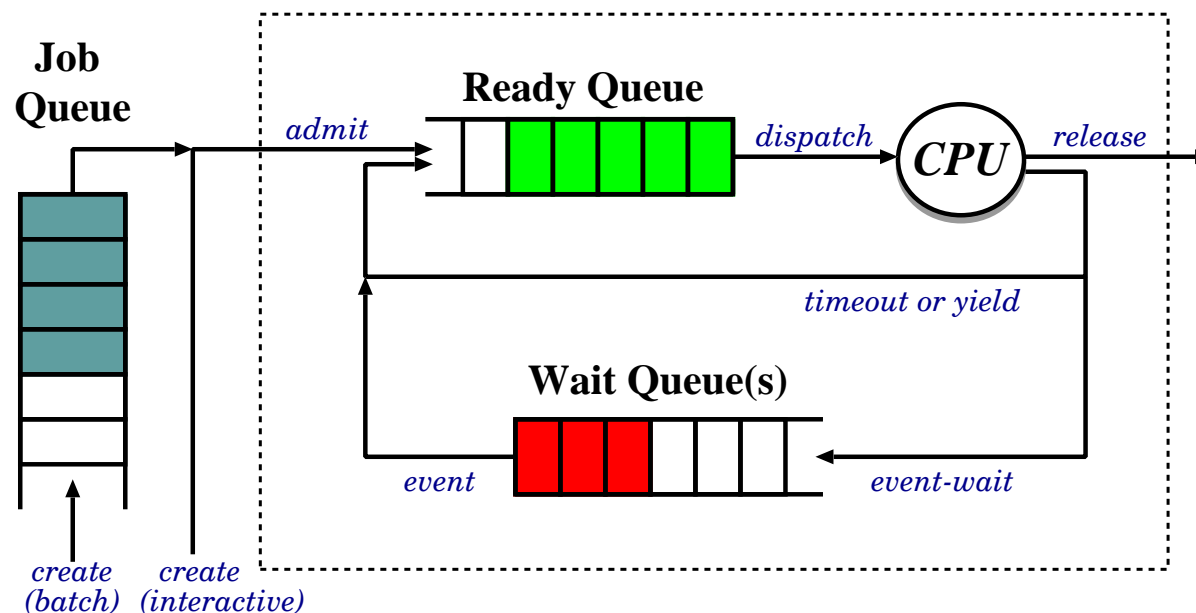
- Job Queue: batch processes awaiting admission.

- Ready Queue: set of all processes residing in main memory, ready to execute.

- Wait Queue(s): set of processes waiting for an I/O device (or for other processes)

- Long-term & short-term schedulers:

  - Job scheduler selects which processes should be brought into the ready queue.
  - CPU scheduler decides which process should be executed next and allocates the CPU to it.

Process Creation

- Nearly all systems are *hierarchical*: parent processes create children processes.

- Resource sharing:

  - parent and children share all resources, or
  - children share subset of parent's resources, or
  - parent and child share no resources.

- Execution:

  - parent and children execute concurrently, or
  - parent waits until children terminate.

- Address space:

  - child is duplicate of parent or
  - child has a program loaded into it.

- e.g. on Unix: `fork()` system call creates a new process

  - all resources shared (i.e. child is a clone).
  - `execve()` system call used to replace process' memory with a new program.

- NT/2K/XP: `CreateProcess()` syscall includes name of program to be executed.

Process Termination

- Process executes last statement and asks the operating system to delete it (exit):
  - output data from child to parent (wait)
  - process' resources are deallocated by the OS.

- Process performs an illegal operation, e.g.
  - makes an attempt to access memory to which it is not authorised,
  - attempts to execute a privileged instruction

- Parent may terminate execution of child processes (abort, kill), e.g. because
  - child has exceeded allocated resources
  - task assigned to child is no longer required
  - parent is exiting ("cascading termination")
  - (many operating systems do not allow a child to continue if its parent terminates)

- e.g. Unix has `wait()`, `exit()` and `kill()`

- e.g. NT/2K/XP has `ExitProcess()` for self termination and `TerminateProcess()` for killing others.

Process Blocking

- In general a process blocks on an *event*, e.g.

  - an I/O device completes an operation,
  - another process sends a message

- Assume OS provides some kind of general-purpose blocking primitive, e.g. `await()`.

- Need care handling *concurrency* issues, e.g.
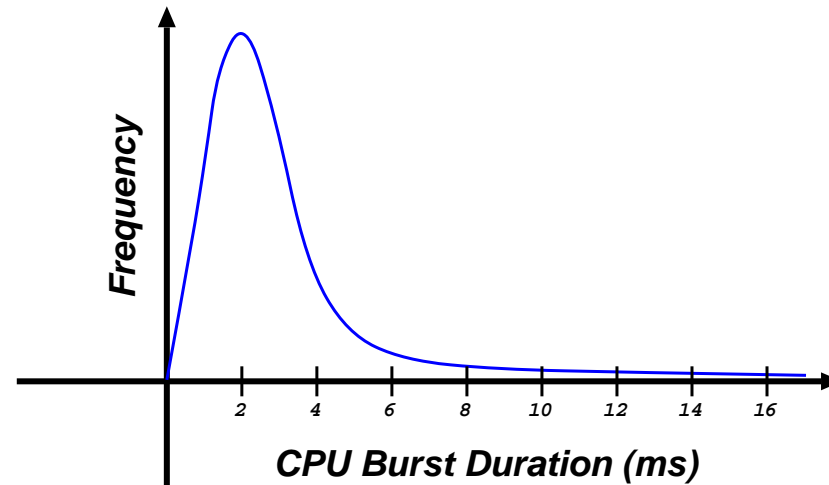
```
        if(no key being pressed) {
            await(keypress);
            print("Key has been pressed!\n");
        }
        // handle keyboard input
```

  What happens if a key is pressed at the first '{' ?

- (This is a *big* area: lots more detail next year.)

- In this course we'll generally assume that problems of this sort do not arise.

CPU-I/O Burst Cycle

- CPU-I/O Burst Cycle: process execution consists of an on-going *cycle* of CPU execution, I/O wait, CPU execution, . . .

- Processes can be described as either:

  1. I/O-bound: spends more time doing I/O than computation; has many short CPU bursts.
  2. CPU-bound: spends more time doing computations; has few very long CPU bursts.

- Observe most processes execute for at most a few milliseconds before blocking

$\Rightarrow$ need multiprogramming to obtain decent overall CPU utilization.

CPU Scheduler

Recall: CPU scheduler selects one of the ready processes and allocates the CPU to it.

- There are a number of occasions when we can/must choose a new process to run:

  1. a running process blocks (`running` $\rightarrow$ `blocked`)
  2. a timer expires (`running` $\rightarrow$ `ready`)
  3. a waiting process unblocks (`blocked` $\rightarrow$ `ready`)
  4. a process terminates (`running` $\rightarrow$ `exit`)

- If only make scheduling decision under 1, 4 $\Rightarrow$ have a *non-preemptive* scheduler:

  ✔ simple to implement
  ✘ open to denial of service
  – e.g. Windows 3.11, early MacOS.

- Otherwise the scheduler is *preemptive*.

  ✔ solves denial of service problem
  ✘ more complicated to implement
  ✘ introduces concurrency problems. . .

Idle system

What do we do if there is no ready process?

- halt processor (until interrupt arrives)

  ✔ saves power (and heat!)

  ✔ increases processor lifetime

  ✘ might take too long to stop and start.

- busy wait in scheduler

  ✔ quick response time

  ✘ ugly, useless

- invent idle process, always available to run

  ✔ gives uniform structure

  ✔ could use it to run checks

  ✘ uses some memory

  ✘ can slow interrupt response

In general there is a trade-off between responsiveness and usefulness.

Scheduling Criteria

A variety of metrics may be used:

1. CPU utilization: the fraction of the time the CPU is being used (and not for idle process!)

2. Throughput: # of processes that complete their execution per time unit.

3. Turnaround time: amount of time to execute a particular process.

4. Waiting time: amount of time a process has been waiting in the ready queue.

5. Response time: amount of time it takes from when a request was submitted until the first response is produced (in time-sharing systems)

Sensible scheduling strategies might be:

- Maximize throughput or CPU utilization

- Minimize average turnaround time, waiting time or response time.

Also need to worry about *fairness* and *liveness*.

First-Come First-Served Scheduling

- FCFS depends on order processes arrive, e.g.

Process	Burst Time	Process	Burst Time	Process	Burst Time
P_1	25	P_2	4	P_3	7

- If processes arrive in the order P_1, P_2, P_3:

  - Waiting time for P_1=0; P_2=25; P_3=29;
  - Average waiting time: $(0 + 25 + 29)/3 = 18$.

- If processes arrive in the order P_3, P_2, P_1:

  - Waiting time for P_1=11; P_2=7; P_3=0;
  - Average waiting time: $(11 + 7 + 0)/3 = 6$.
 - i.e. three times as good!

- First case poor due to *convoy effect*.

SJF Scheduling

Intuition from FCFS leads us to *shortest job first* (SJF) scheduling.

- Associate with each process the length of its next CPU burst.

- Use these lengths to schedule the process with the shortest time (FCFS can be used to break ties).

For example:

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

P_1	P_3	P_2	P_4

0   7   8   12   16
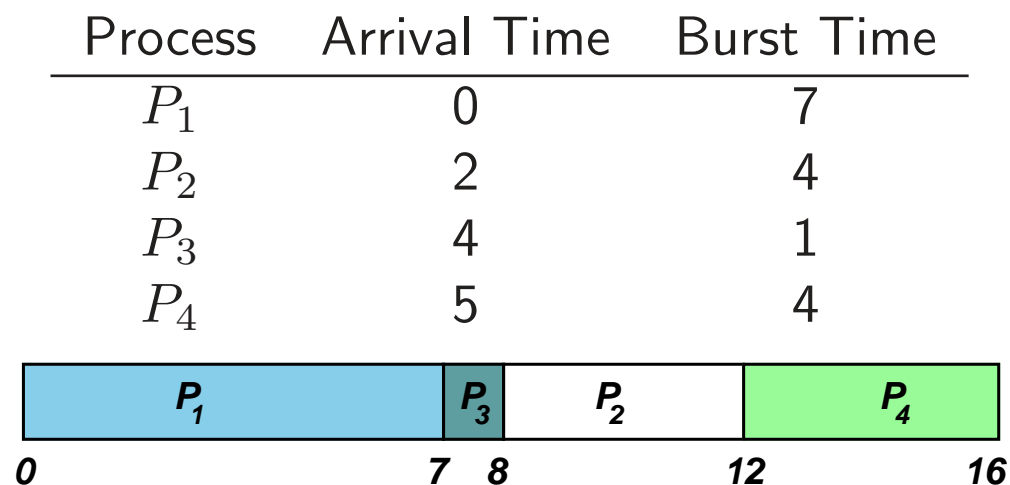
- Waiting time for P_1=0; P_2=6; P_3=3; P_4=7;

- Average waiting time: $(0 + 6 + 3 + 7)/4 = 4$.

SJF is optimal in the sense that it gives the minimum average waiting time for any given set of processes. . .

SRTF Scheduling

- SRTF $=$ Shortest Remaining-Time First.

- Just a preemptive version of SJF.

- i.e. if a new process arrives with a CPU burst length less than the *remaining time* of the current executing process, preempt.

For example:

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

P_1	P_2	P_3	P_2	P_4	P_1

0　　2　　4　5　　7　　　　11　　　　16

- Waiting time for $P_1=9$; $P_2=1$; $P_3=0$; $P_4=2$;

- Average waiting time: $(9 + 1 + 0 + 2)/4 = 3$.

What are the problems here?

Predicting Burst Lengths

- For both SJF and SRTF require the next "burst length" for each process $\Rightarrow$ need to come up with some way to predict it.

- Can be done by using the length of previous CPU bursts to calculate an exponentially-weighted moving average (EWMA):

  1. t_n = actual length of nth CPU burst.
  2. τ_{n+1} = predicted value for next CPU burst.
  3. For $\alpha, 0 \le \alpha \le 1$ define:

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

- 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.

- Choose value of α according to our belief about the system, e.g. if we believe history irrelevant, choose $\alpha \approx 1$ and then get $\tau_{n+1} \approx t_n$.

- In general an EWMA is a good predictor if the variance is small.

Round Robin Scheduling

Define a small fixed unit of time called a *quantum* (or *time-slice*), typically 10-100 milliseconds. Then:

- Process at head of the ready queue is allocated the CPU for (up to) one quantum.

- When the time has elapsed, the process is preempted and added to the tail of the ready queue.

Round robin has some nice properties:

- Fair: if there are n processes in the ready queue and the time quantum is q, then each process gets $1/n^{th}$ of the CPU.

- Live: no process waits more than $(n-1)q$ time units before receiving a CPU allocation.

- Typically get higher average turnaround time than SRTF, but better average *response time*.

But tricky choosing correct size quantum:

- q too large $\Rightarrow$ FCFS/FIFO

- q too small $\Rightarrow$ context switch overhead too high.

Static Priority Scheduling

- Associate an (integer) priority with each process

- For example:

Priority	Type		Priority	Type
0	system internal processes		**2**	interactive processes (students)
1	interactive processes (staff)		**3**	batch processes.

- Then allocate CPU to the highest priority process:

  - 'highest priority' typically means smallest integer
  - get preemptive and non-preemptive variants.

- e.g. SJF is priority scheduling where priority is the predicted next CPU burst time.

- **Problem**: how to resolve ties?

  - round robin with time-slicing
  - allocate quantum to each process in turn.
  - Problem: biased towards CPU intensive jobs.
    - ∗ per-process quantum based on usage?
    - ∗ ignore?

- **Problem**: starvation. . .

Dynamic Priority Scheduling

- Use same scheduling algorithm, but allow priorities to change over time.

- e.g. simple aging:

  - processes have a (static) *base priority* and a dynamic *effective priority*.
  - if process starved for k seconds, increment effective priority.
  - once process runs, reset effective priority.

- e.g. computed priority:

  - first used in Dijkstra's THE
  - time slots: . . . , t, $t+1$, . . .
  - in each time slot t, measure the CPU usage of process j: u^j
  - priority for process j in slot $t+1$:
    $$p^j_{t+1} = f(u^j_t, p^j_t, u^j_{t-1}, p^j_{t-1}, \ldots)$$
  - e.g. $p^j_{t+1} = p^j_t/2 + ku^j_t$
  - penalises CPU bound $\rightarrow$ supports I/O bound.

- today such computation considered acceptable. . .

Memory Management

In a multiprogramming system:

- many processes in memory simultaneously, and every process needs memory for:
  - instructions ("code" or "text"),
  - static data (in program), and
  - dynamic data (heap and stack).

- in addition, operating system itself needs memory for instructions and data.

$\Rightarrow$ must share memory between OS and k processes.

The memory magagement subsystem handles:

1. Relocation

2. Allocation

3. Protection

4. Sharing

5. Logical Organisation

6. Physical Organisation

The Address Binding Problem

Consider the following simple program:

```
int x, y;
x = 5;
y = x + 3;
```

We can imagine this would result in some assembly code which looks something like:

```
str #5, [Rx]            // store 5 into 'x'
ldr R1, [Rx]            // load value of x from memory
add R2, R1, #3          // and add 3 to it
str R2, [Ry]            // and store result in 'y'
```

where the expression '[addr]' should be read to mean "the contents of the memory at address addr".

Then the address binding problem is:

what values do we give Rx and Ry ?

This is a problem because we don't know where in memory our program will be loaded when we run it:

- e.g. if loaded at 0x1000, then x and y might be stored at 0x2000, 0x2004, but if loaded at 0x5000, then x and y might be at 0x6000, 0x6004.

Address Binding and Relocation

To solve the problem, we need to set up some kind of correspondence between "program addresses" and "real addresses". This can be done:

- at compile time:

  - requires knowledge of absolute addresses; e.g. DOS .com files

- at load time:

  - when program loaded, work out position in memory and update every relevant instruction in code with correct addresses
  - must be done every time program is loaded
  - ok for embedded systems / boot-loaders

- at run-time:

  - get some hardware to automatically translate between program addresses and real addresses.
  - no changes at all required to program itself.
  - most popular and flexible scheme, providing we have the requisite hardware, viz. a memory management unit or MMU.

Logical vs Physical Addresses

Mapping of logical to physical addresses is done at run-time by Memory Management Unit (MMU), e.g.
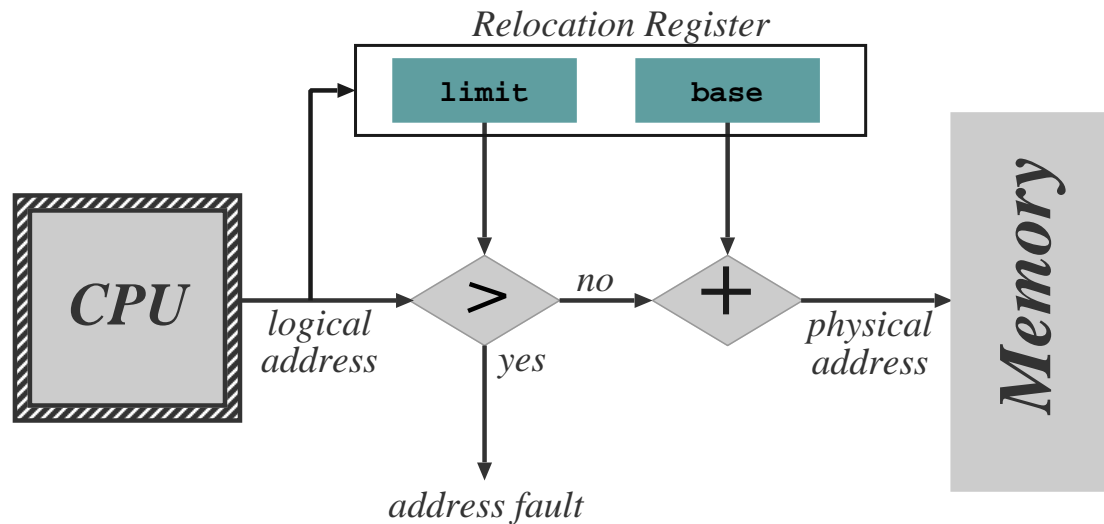
1. Relocation register holds the value of the base address owned by the process.
2. Relocation register contents are added to each memory address before it is sent to memory.
3. e.g. DOS on 80x86 — 4 relocation registers, logical address is a tuple (s, o).
4. NB: process never sees physical address — simply manipulates logical addresses.
5. OS has privilege to update relocation register.

Contiguous Allocation

Given that we want multiple virtual processors, how can we support this in a single address space?

Where do we put processes in memory?

- OS typically must be in low memory due to location of interrupt vectors

- Easiest way is to statically divide memory into multiple fixed size partitions:
  - each partition spans a contiguous range of physical memory
  - bottom partition contains OS, remaining partitions each contain exactly one process.
  - when a process terminates its partition becomes available to new processes.
  - e.g. OS/360 MFT.

- Need to protect OS and user processes from malicious programs:
  - use base and limit registers in MMU
  - update values when a new processes is scheduled
  - NB: solving both relocation and protection problems at the same time!

Static Multiprogramming
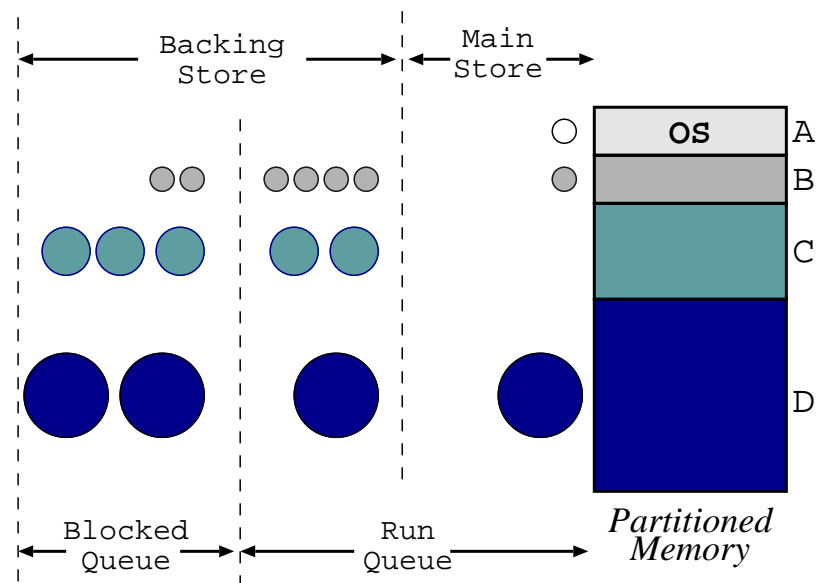
- partition memory when installing OS, and allocate pieces to different job queues.
- associate jobs to a job queue according to size.
- swap job back to disk when:
  - blocked on I/O (assuming I/O is slower than the backing store).
  - time sliced: larger the job, larger the time slice
- run job from another queue while swapping jobs
- e.g. IBM OS/360 MFT, ICL System 4
- **problems**: fragmentation (partition too big), cannot grow (partition too small).
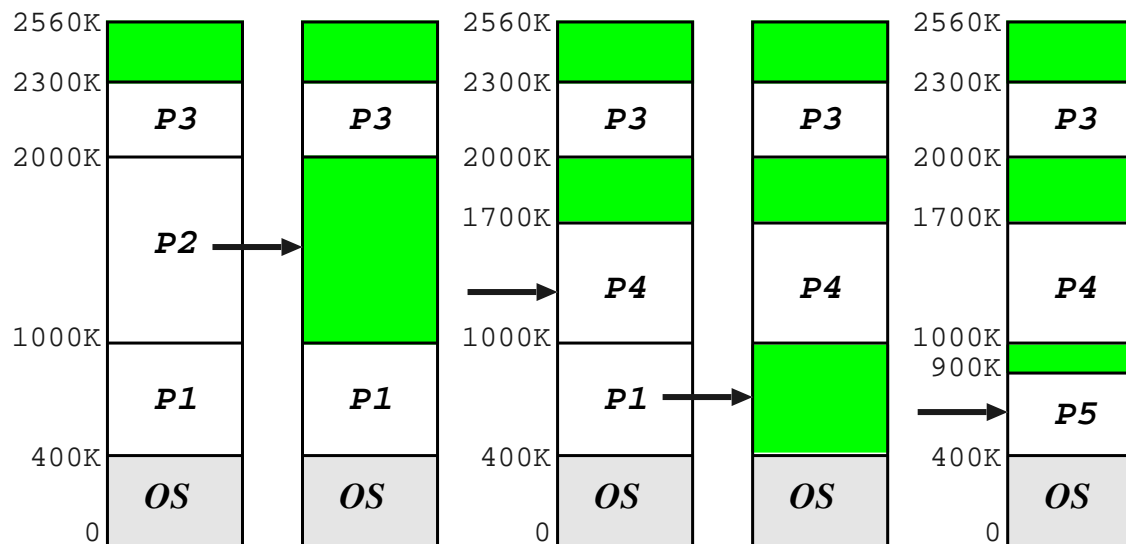
Dynamic Partitioning

Get more flexibility if allow partition sizes to be dynamically chosen, e.g. OS/360
MVT ("Multiple Variable-sized Tasks"):

- OS keeps track of which areas of memory are available and which are occupied.

- e.g. use one or more *linked lists*:

```
0000 0C04 |→   2200 3810 |→   4790 91E8 |
        B0F0 B130 |→   D708 FFFF |→
```
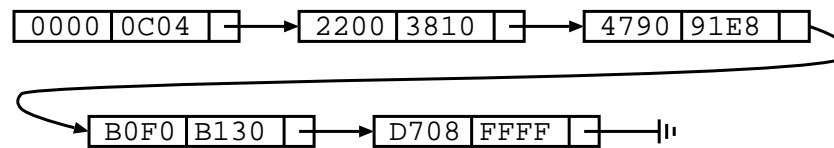
- When a new process arrives into the system, the OS searches for a hole large
  enough to fit the process.

- Some algorithms to determine which hole to use for new process:

  - **first fit**: stop searching list as soon as big enough hole is found.
  - **best fit**: search entire list to find "best" fitting hole (i.e. smallest hole which is
    large enough)
  - **worst fit**: counterintuitively allocate largest hole (again must search entire list).

- When process terminates its memory returns onto the free list, coalescing holes
  together where appropriate.

Scheduling Example

- Consider machine with total of 2560K memory, where OS requires 400K.

- The following jobs are in the queue:

Process	Memory Reqd	Total Execution Time
P_1	600K	10
P_2	1000K	5
P_3	300K	20
P_4	700K	8
P_5	500K	15

External Fragmentation
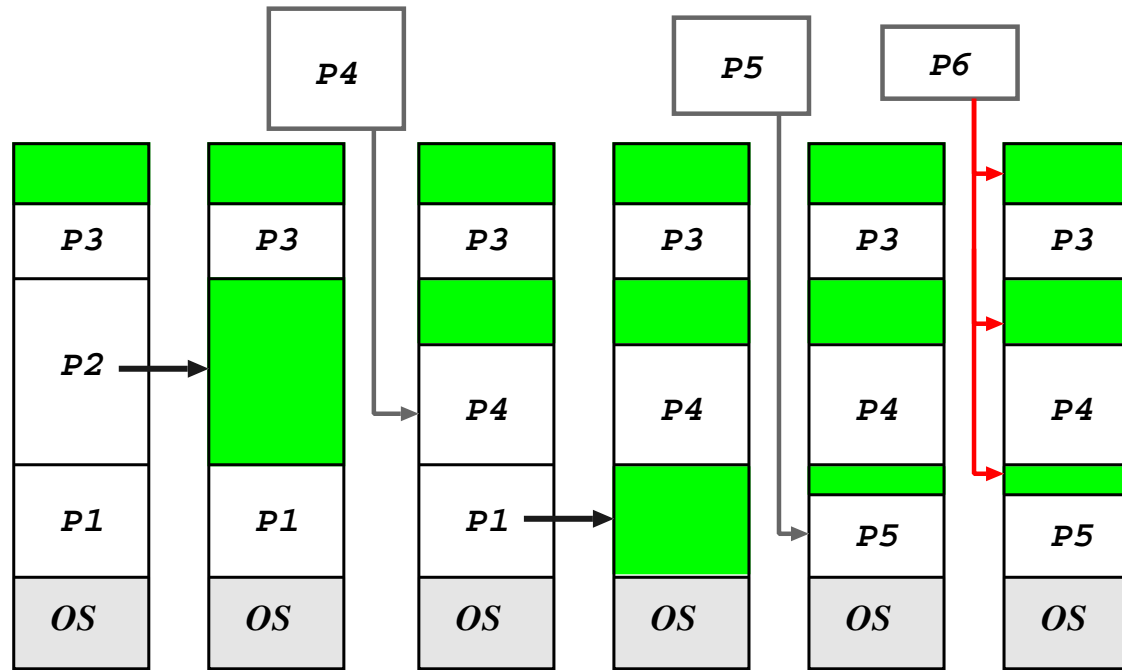
- Dynamic partitioning algorithms suffer from external fragmentation: as processes are loaded they leave little fragments which may not be used.
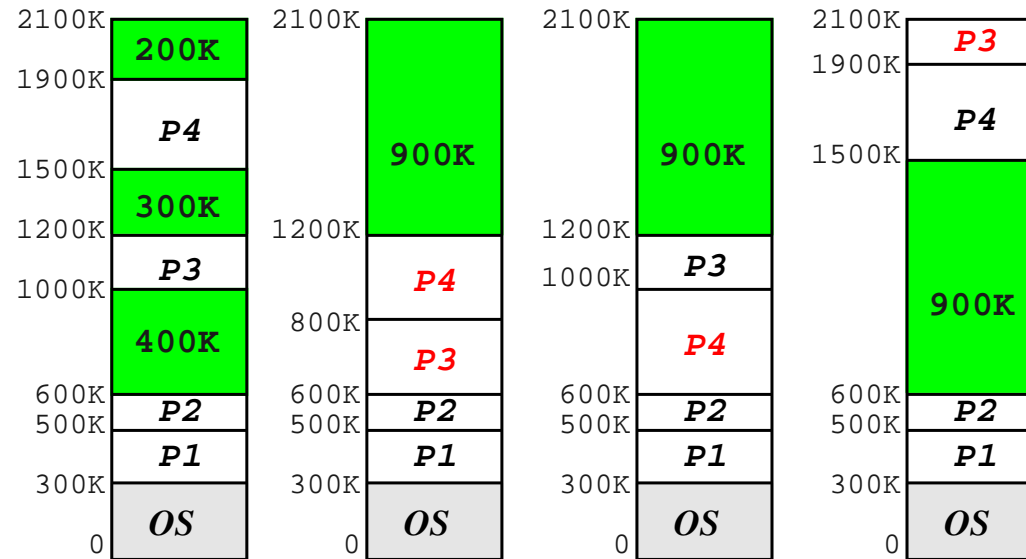
- External fragmentation exists when the total available memory is sufficient for a request, but is unusable because it is split into many holes.

- Can also have problems with tiny holes

Solution: compact holes periodically.

Compaction

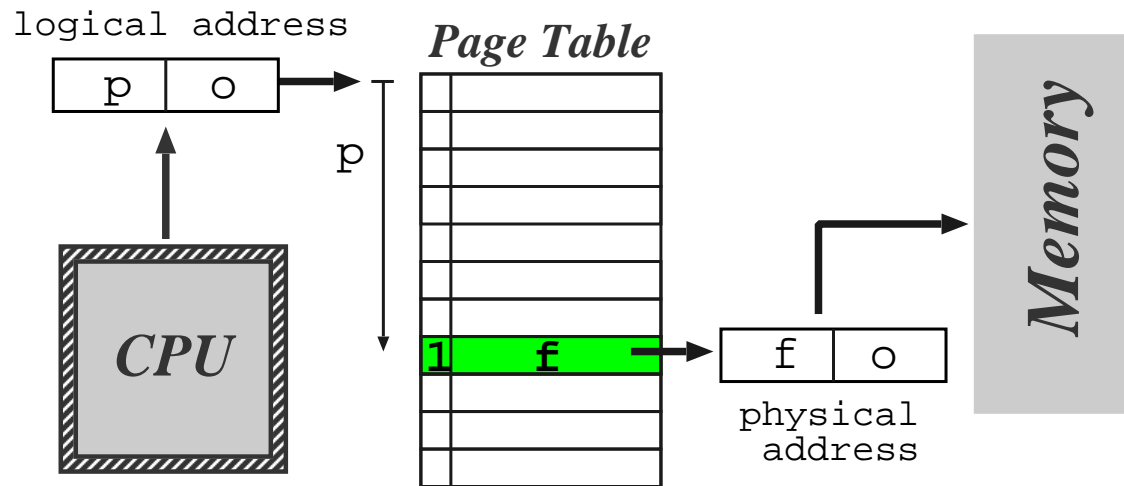Choosing optimal strategy quite tricky. . .

Note that:

- We require run-time relocation for this to work.

- Can be done more efficiently when process is moved into memory from a swap.

- Some machines used to have hardware support (e.g. CDC Cyber).

Also get fragmentation in *backing store*, but in this case compaction not really a viable option. . .

Paged Virtual Memory

Another solution is to allow a process to exist in non-contiguous memory, i.e.

- divide physical memory into relatively small blocks of fixed size, called frames

- divide logical memory into blocks of the same size called pages

- (typical page sizes are between 512bytes and 8K)

- each address generated by CPU comprises a page number p and page offset o.

- MMU uses p as an index into a page table.

- page table contains associated frame number f

- usually have $|\mathtt{p}| \gg |\mathtt{f}| \Rightarrow$ need valid bit

Paging Pros and Cons
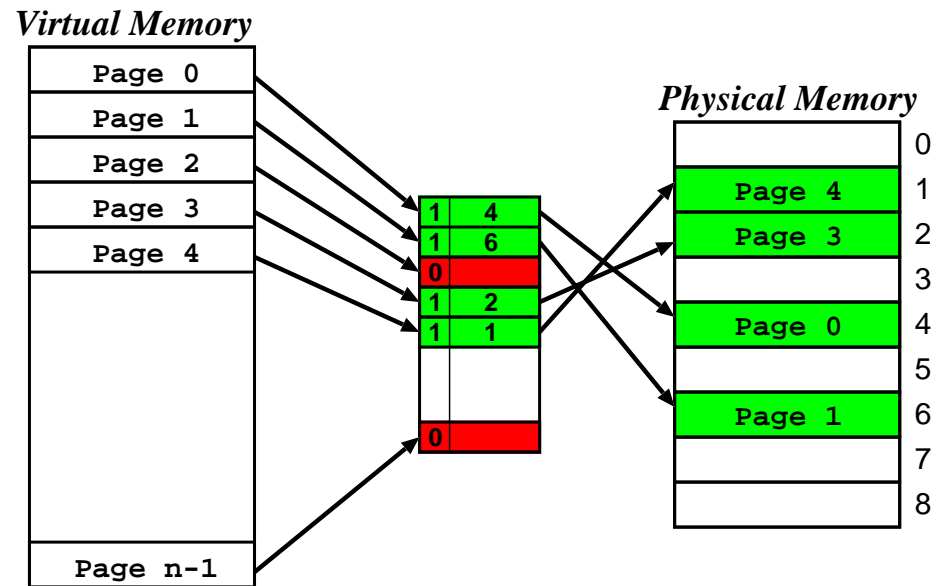
✔ memory allocation easier.

✘ OS must keep page table per process

✔ no external fragmentation (in physical memory at least).

✘ but get **internal fragmentation**.

✔ clear separation between user and system view of memory usage.

✘ additional overhead on context switching

Structure of the Page Table

Different kinds of hardware support can be provided:

- Simplest case: set of dedicated relocation registers

    - one register per page
    - OS loads the registers on context switch
    - fine if the page table is small. . . but what if have large number of pages ?

- Alternatively keep page table in memory

    - only one register needed in MMU (page table base register (PTBR))
    - OS switches this when switching process

- **Problem**: page tables might still be very big.

    - can keep a page table length register (PTLR) to indicate size of page table.
    - or can use more complex structure (see later)

- **Problem**: need to refer to memory *twice* for every 'actual' memory reference. . .

    $\Rightarrow$ use a translation lookaside buffer (TLB)

TLB Operation
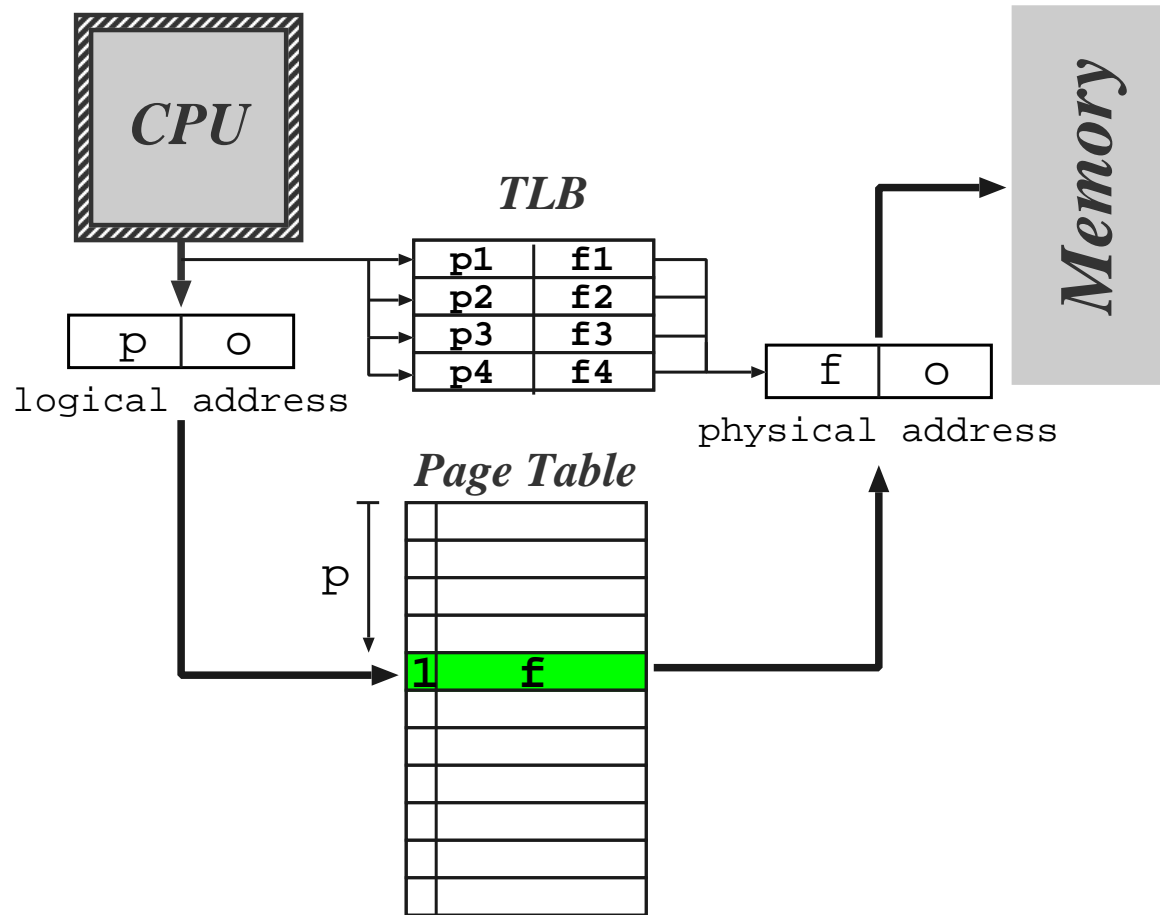
- On memory reference present TLB with logical memory address

- If page table entry for the page is present then get an immediate result

- If not then make memory reference to page tables, and update the TLB

TLB Issues

- Updating TLB tricky if it is full: need to discard something.

- Context switch may requires TLB flush so that next process doesn't use wrong page table entries.

  - Today many TLBs support process tags (sometimes called address space numbers) to improve performance.

- Hit ratio is the percentage of time a page entry is found in TLB

- e.g. consider TLB search time of $20ns$, memory access time of $100ns$, and a hit ratio of 80%

$\Rightarrow$ assuming one memory reference required for page table lookup, the $effective$ memory access time is $0.8 \times 120 + 0.2 \times 220 = 140ns$.

- Increase hit ratio to 98% gives effective access time of 122ns — only a 13% improvement.

Multilevel Page Tables

- Most modern systems can support very large $(2^{32}, 2^{64})$ address spaces.

- Solution – split page table into several sub-parts

- Two level paging – page the page table
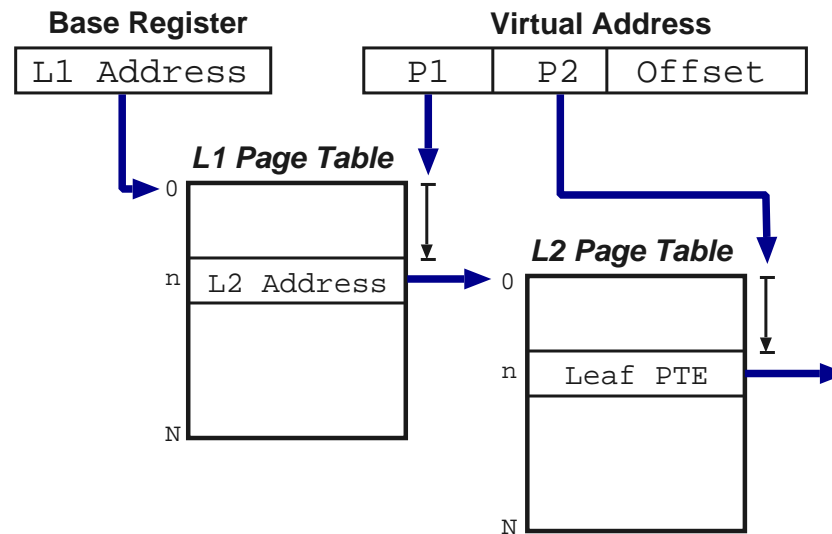
- For 64 bit architectures a two-level paging scheme is not sufficient: need further levels (usually 4, or even 5).

- (even some 32 bit machines have > 2 levels, e.g. x86 PAE mode).

Example: x86

Virtual Address

L1	L2	Offset

Page Directory (Level 1)

20 bits

PTA	IGN	PS	ZO	AC	CD	WT	US	RW	VD

1024 entries
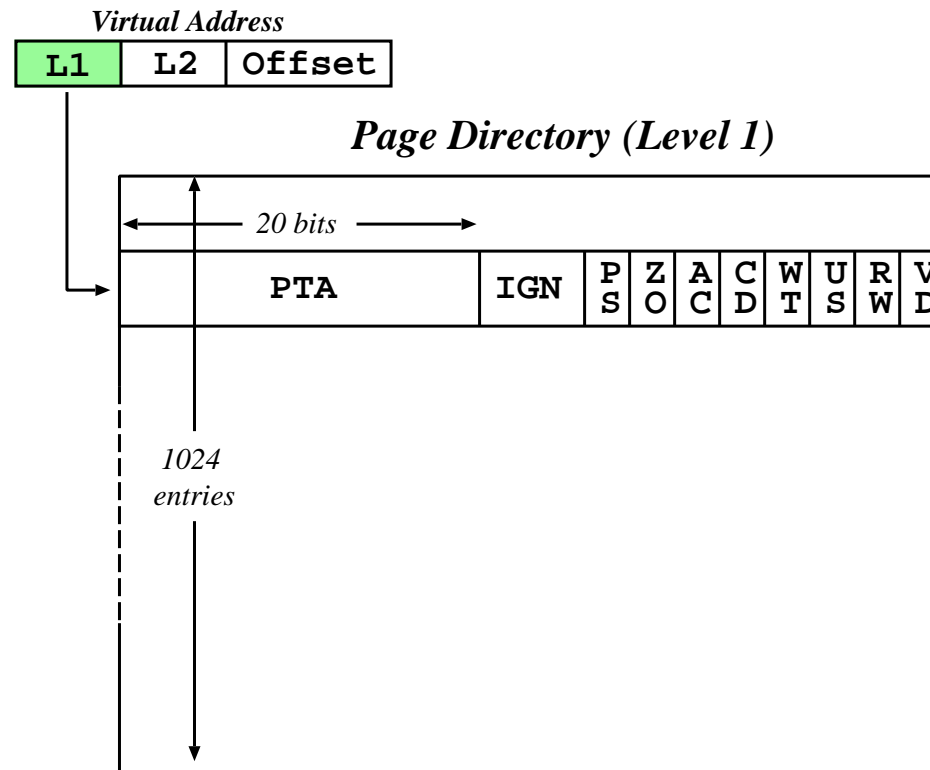
- Page size 4K (or 4Mb).

- First lookup is in the *page directory*: index using most 10 significant bits.

- Address of page directory stored in internal processor register (cr3).

- Results (normally) in the address of a *page table*.

Example: x86 (2)

Virtual Address

L1	L2	Offset

Page Table (Level 2)

←—— *20 bits* ——→

	PFA	IGN	G L	Z O	D Y	A C	C D	W T	U S	R W	V D

1024 entries
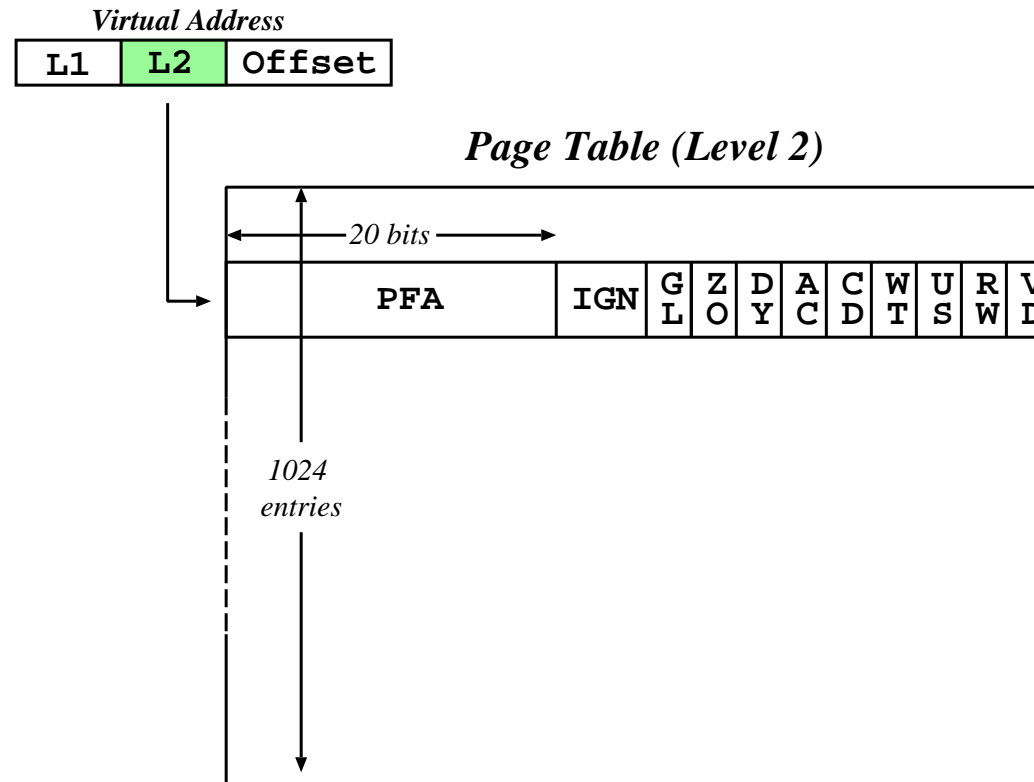
- Use next 10 bits to index into page table.

- Once retrieve page frame address, add in the offset (i.e. the low 12 bits).

- Notice page directory and page tables are exactly one page each themselves.

Protection Issues

- Associate protection bits with each page – kept in page tables (and TLB).

- e.g. one bit for read, one for write, one for execute.

- May also distinguish whether a page may only be accessed when executing in *kernel mode*, e.g. a page-table entry may look like:

Frame Number	K	R	W	X	V

- At the same time as address is going through page translation hardware, can check protection bits.

- Attempt to violate protection causes h/w trap to operating system code

- As before, have *valid/invalid* bit determining if the page is mapped into the process address space:

  − if invalid ⇒ trap to OS handler
  − can do lots of interesting things here, particularly with regard to sharing. . .

Shared Pages

Another advantage of paged memory is code/data sharing, for example:

- binaries: editor, compiler etc.

- libraries: shared objects, dlls.

So how does this work?

- Implemented as two logical addresses which map to one physical address.

- If code is *re-entrant* (i.e. stateless, non-self modifying) it can be easily shared between users.

- 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 table mapping.

- (may use this for lazy data sharing too).

Requires additional book-keeping in OS, but worth it, e.g. over 100MB of shared code on my linux box.

Virtual Memory

- Virtual addressing allows us to introduce the idea of virtual memory:

    – already have valid or invalid pages; introduce a new "non-resident" designation
    – such pages live on a non-volatile backing store, such as a hard-disk.
    – processes access non-resident memory just as if it were 'the real thing'.

- Virtual memory (VM) has a number of benefits:

    – portability: programs work regardless of how much actual memory present
    – convenience: programmer can use e.g. large sparse data structures with impunity
    – efficiency: no need to waste (real) memory on code or data which isn't used.

- VM typically implemented via demand paging:

    – programs (executables) reside on disk
    – to execute a process we load pages in *on demand*; i.e. as and when they are referenced.

- Also get *demand segmentation*, but rare.

Demand Paging Details

When loading a new process for execution:

- we create its address space (e.g. page tables, etc), but mark all PTEs as either "invalid" or "non-resident"; and then
- add its process control block (PCB) to the ready-queue.

Then whenever we receive a page fault:

1. check PTE to determine if "invalid" or not
2. if an invalid reference $\Rightarrow$ kill process;
3. 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 the PTE for this page to show that it is now valid
   - restart the process at the faulting instruction

Scheme described above is *pure* demand paging:

- never brings in a page until required $\Rightarrow$ get lots of page faults and I/O when the process first begins.
- hence many real systems explicitly load some core parts of the process first

Page Replacement

- When paging in from disk, we need a free frame of physical memory to hold the data we're reading in.

- In reality, size of physical memory is limited $\Rightarrow$

  - need to discard unused pages if total demand exceeds physical memory size
  - (alternatively could swap out a whole process to free some frames)

- Modified algorithm: on a page fault we

  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 the victim page back to disk, and
    (d) mark it as invalid in its process page tables
  3. read desired page into freed frame
  4. restart the faulting process

- Can reduce overhead by adding a dirty bit to PTEs (can potentially omit step 2c)

- **Question**: how do we choose our victim page?

Page Replacement Algorithms

- **First-In First-Out (FIFO)**

  - keep a queue of pages, discard from head
  - performance difficult to predict: have no idea whether page replaced will be used again or not
  - discard is independent of page use frequency
  - in general: pretty bad, although very simple.

- **Optimal Algorithm (OPT)**

  - replace the page which will not be used again for longest period of time
  - can only be done with an oracle, or in hindsight
  - serves as a good comparison for other algorithms

- **Least Recently Used (LRU)**

  - LRU replaces the page which has not been used for the longest amount of time
  - (i.e. LRU is OPT with -ve time)
  - assumes past is a good predictor of the future
  - **Question**: how do we determine the LRU ordering?

Implementing LRU

- Could try using counters

  - give each page table entry a time-of-use field and give CPU a logical clock (e.g. an n-bit counter)
  - whenever a page is referenced, its PTE is updated to clock value
  - replace page with smallest time value
  - **problem**: requires a search to find minimum value
  - **problem**: adds a write to memory (PTE) on every memory reference
  - **problem**: clock overflow. . .

- Or a page stack:

  - maintain a $stack$ of pages (a doubly-linked list)
  - update stack on every reference to ensure new (MRU)) page on top
  - discard from bottom of stack
  - **problem**: requires changing 6 pointers per [new] reference
  - possible with h/w support, but slow even then (and extremely slow without it!)

- Neither scheme seems practical on a standard processor $\Rightarrow$ need another way.

Approximating LRU (1)

- Many systems have a reference bit in the PTE which is set by h/w whenever the page is touched

- This allows not recently used (NRU) replacement:

  - periodically (e.g. 20ms) clear all reference bits
  - when choosing a victim to replace, prefer pages with clear reference bits
  - if we also have a modified bit (or dirty bit) in the PTE, we can extend NRU to use that too:

Ref?	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

- Or can extend by maintaining more history, e.g.

  - for each page, the operating system maintains an 8-bit value, initialized to zero
  - periodically (e.g. every 20ms), shift the reference bit onto most-significant bit of the byte, and clear the reference bit
  - select lowest value page (or one of them) to replace

Approximating LRU (2)
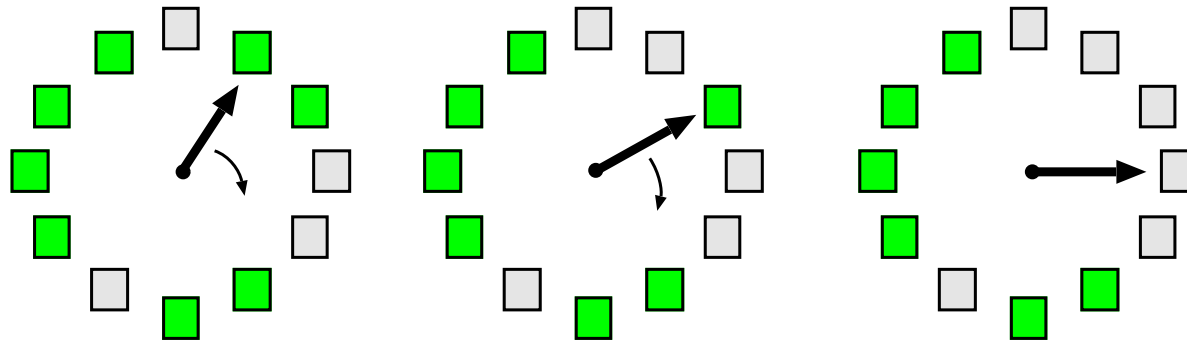
- Popular NRU scheme: second-chance FIFO

    – store pages in queue as per FIFO
    – before discarding head, check its reference bit
    – if reference bit is 0, then discard it, otherwise:
        * reset reference bit, and add page to tail of queue
        * i.e. give it "a second chance"

- Often implemented with circular queue and head pointer: then called clock.

- If no h/w provided reference bit can emulate:

    – to clear "reference bit", mark page no access
    – if referenced ⇒ trap, update PTE, and resume
    – to check if referenced, check permissions
    – can use similar scheme to emulate modified bit

Other Replacement Schemes

- **Counting Algorithms**: keep a count of the number of references to each page

  - LFU: replace page with smallest count
  - MFU: replace highest count because low count $\Rightarrow$ most recently brought in.

- **Page Buffering Algorithms**:

  - keep a min. number of victims in a free pool
  - new page read in before writing out victim.

- **(Pseudo) MRU**:

  - consider access of e.g. large array.
  - page to replace is one application has *just finished with*, i.e. most recently used.
  - e.g. track page faults and look for sequences.
  - discard the kth in victim sequence.

- **Application-specific**:

  - stop trying to second guess what's going on.
  - provide hook for app. to suggest replacement.
  - must be careful with denial of service. . .

Performance Comparison
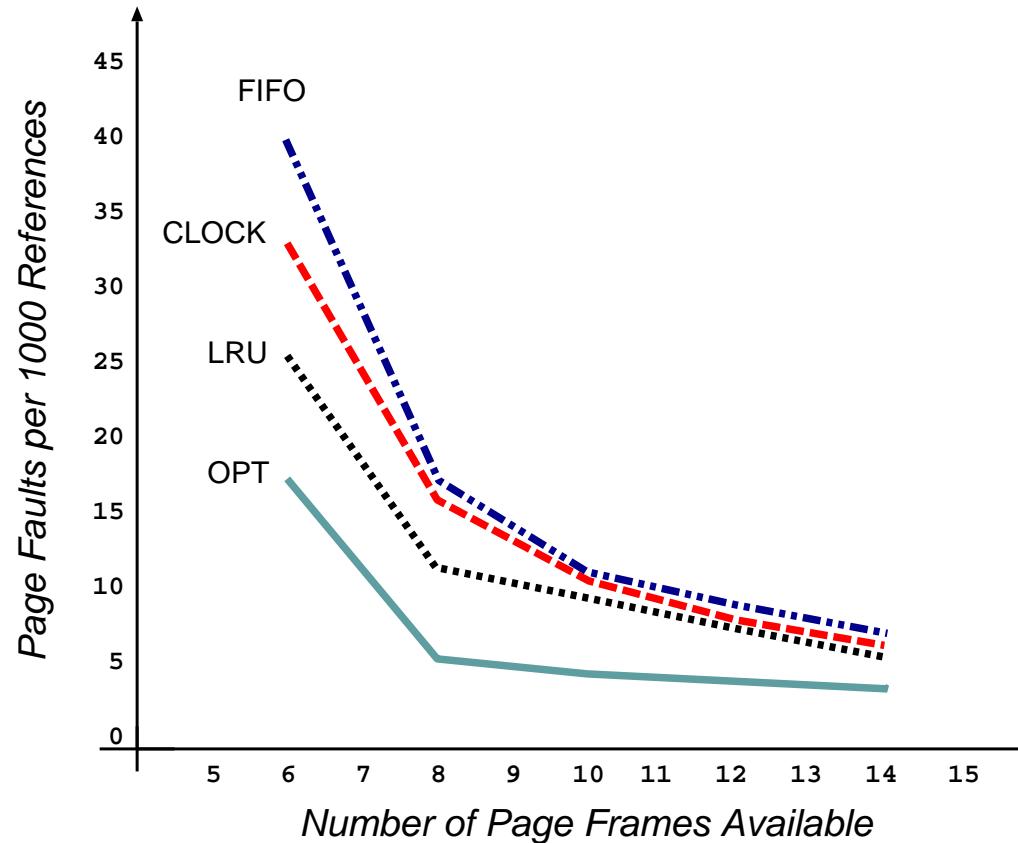
Graph plots page-fault rate against number of physical frames for a pseudo-local reference string.

- want to minimise area under curve

- FIFO can exhibit Belady's anomaly (although it doesn't in this case)

- getting frame allocation right has major impact. . .

Frame Allocation

- A certain fraction of physical memory is reserved per-process and for core operating system code and data.

- Need an *allocation policy* to determine how to distribute the remaining frames.

- Objectives:

  - Fairness (or proportional fairness)?
    * e.g. divide m frames between n processes as m/n, with any remainder staying in the free pool
    * e.g. divide frames in proportion to size of process (i.e. number of pages used)
  - Minimize system-wide page-fault rate?
    (e.g. allocate all memory to few processes)
  - Maximize level of multiprogramming?
    (e.g. allocate min memory to many processes)

- Most page replacement schemes are *global*: all pages considered for replacement.

$\Rightarrow$ allocation policy implicitly enforced during page-in:

  - allocation succeeds iff policy agrees
  - 'free frames' often in use $\Rightarrow$ steal them!

The Risk of Thrashing
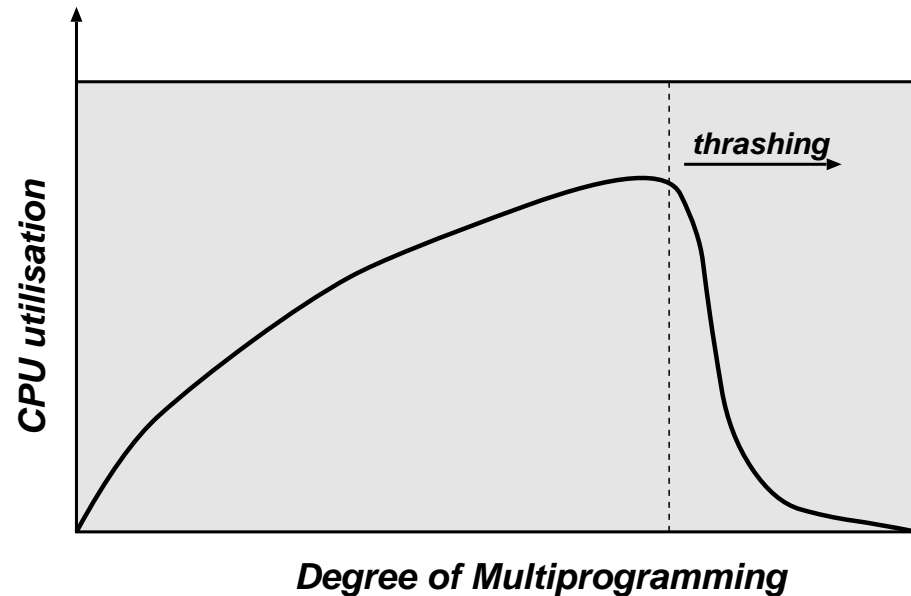
- As more and more processes enter the system (multi-programming level (MPL) increases), the frames-per-process value can get very small.

- At some point we hit a wall:

  - a process needs more frames, so steals them
  - but the other processes need those pages, so they fault to bring them back in
  - number of runnable processes plunges

- To avoid thrashing we must give processes as many frames as they "need"

- If we can't, we need to reduce the MPL: **better page-replacement won't help!**

Locality of Reference
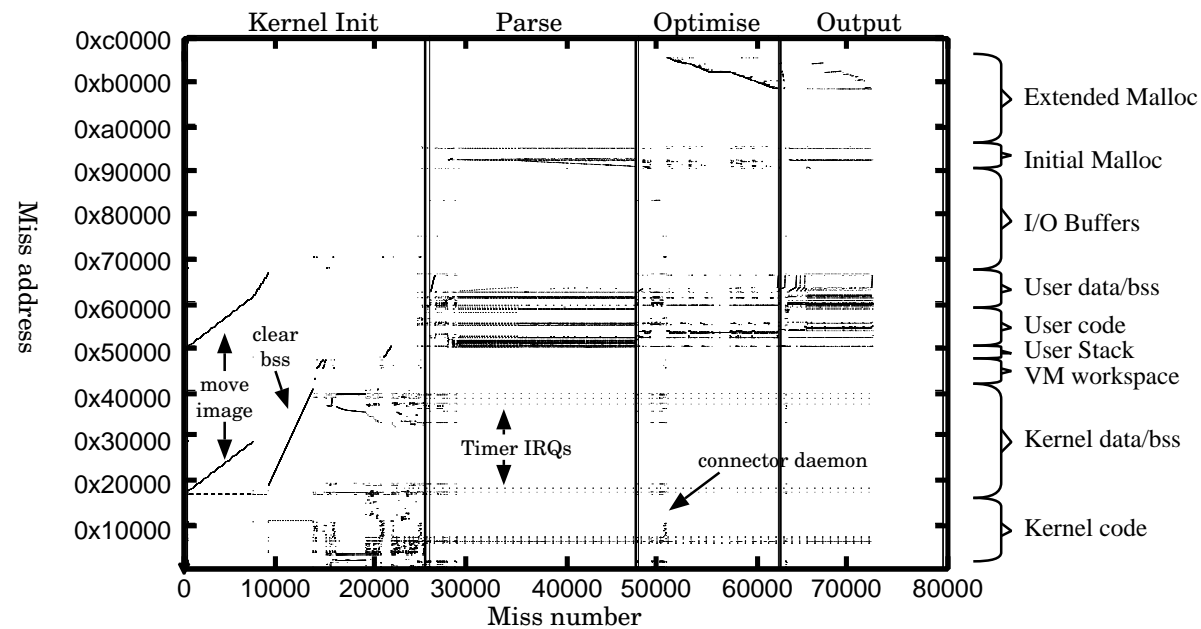
Locality of reference: in a short time interval, the locations referenced by a process tend to be grouped into a few regions in its address space.

- procedure being executed

- ... sub-procedures

- ... data access

- ... stack variables

Note: have locality in both space and time.

Avoiding Thrashing

We can use the locality of reference principle to help determine how many frames a process needs:

- define the Working Set (Denning, 1967)

  - set of pages that a process needs to be resident "the same time" to make any (reasonable) progress
  - varies between processes and during execution
  - assume process moves through $phases$:
    * in each phase, get (spatial) locality of reference
    * from time to time get $phase\ shift$

- OS can try to prevent thrashing by ensuring sufficient pages for current phase:

  - sample page reference bits every e.g. 10ms
  - if a page is "in use", say it's in the working set
  - sum working set sizes to get total demand D
  - if $D > m$ we are in danger of thrashing $\Rightarrow$ suspend a process

- Alternatively use page fault frequency (PFF):

  - monitor per-process page fault rate
  - if too high, allocate more frames to process

Segmentation

Limit   Base

	Limit	Base
0	1000	5900
1	200	0
2	5000	200
3	200	5700
4	300	5300

Segment Table

procedure 0

stack 1

main() 2

symbols 4

sys library 3
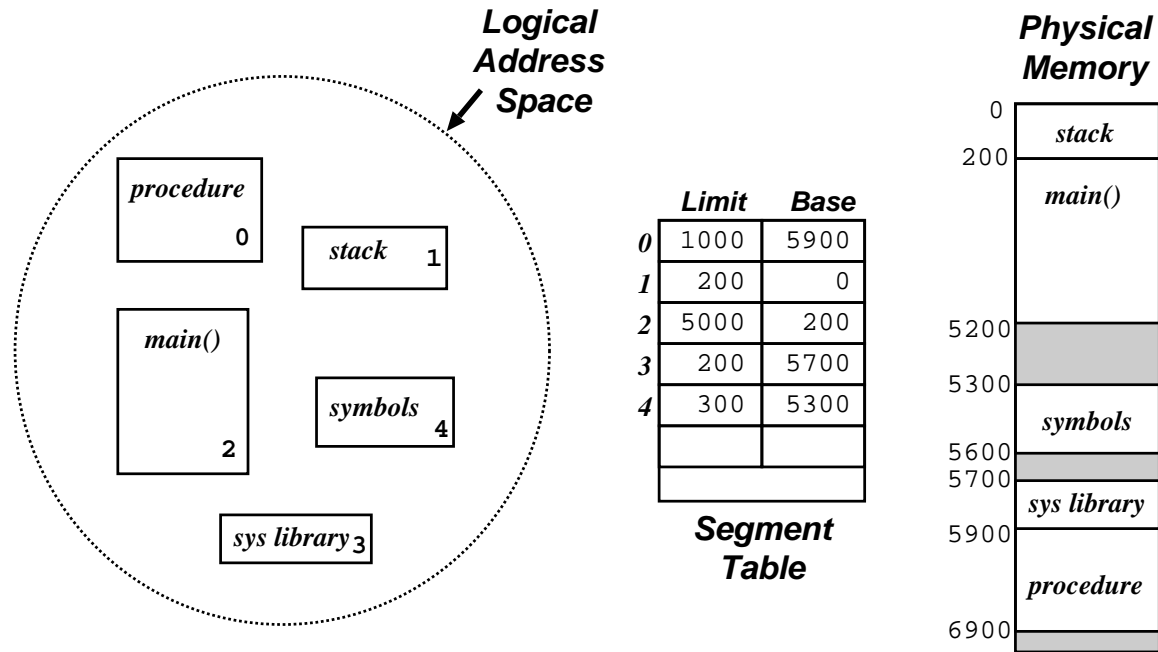
- When programming, a user prefers to view memory as a set of "objects" of various sizes, with no particular ordering

- Segmentation supports this user-view of memory — logical address space is a collection of (typically disjoint) segments.

  - Segments have a name (or a number) and a length.
  - Logical addresses specify segment and offset.

- Contrast with paging where user is unaware of memory structure (one big linear virtual address space, all managed transparently by OS).

Implementing Segments

- Maintain a segment table for each process:

Segment	Access	Base	Size	Others!

- If program has a very large number of segments then the table is kept in memory, pointed to by ST base register STBR

- Also need a ST length register STLR since number of segs used by different programs will differ widely

- The table is part of the process context and hence is changed on each process switch.

Algorithm:

1. Program presents address (s, d).
   Check that $s <$ STLR. If not, fault

2. Obtain table entry at reference $s+$ STBR, a tuple of form (b_s, l_s)

3. If $0 \le d < l_s$ then this is a valid address at location (b_s, d), else fault

Sharing and Protection

- Big advantage of segmentation is that protection is per segment; i.e. corresponds to logical view (and programmer's view)

- Protection bits associated with each ST entry checked in usual way

  - e.g. instruction segments (should be non-self modifying!) can be protected against writes
  - e.g. place each array in own seg $\Rightarrow$ array limits checked by h/w

- Segmentation also facilitates sharing of code/data

  - each process has its own STBR/STLR
  - sharing enabled when two processes have identical entries
  - for data segments can use copy-on-write as per paged case.

- Several subtle caveats exist with segmentation — e.g. jumps within shared code.

Sharing Segments
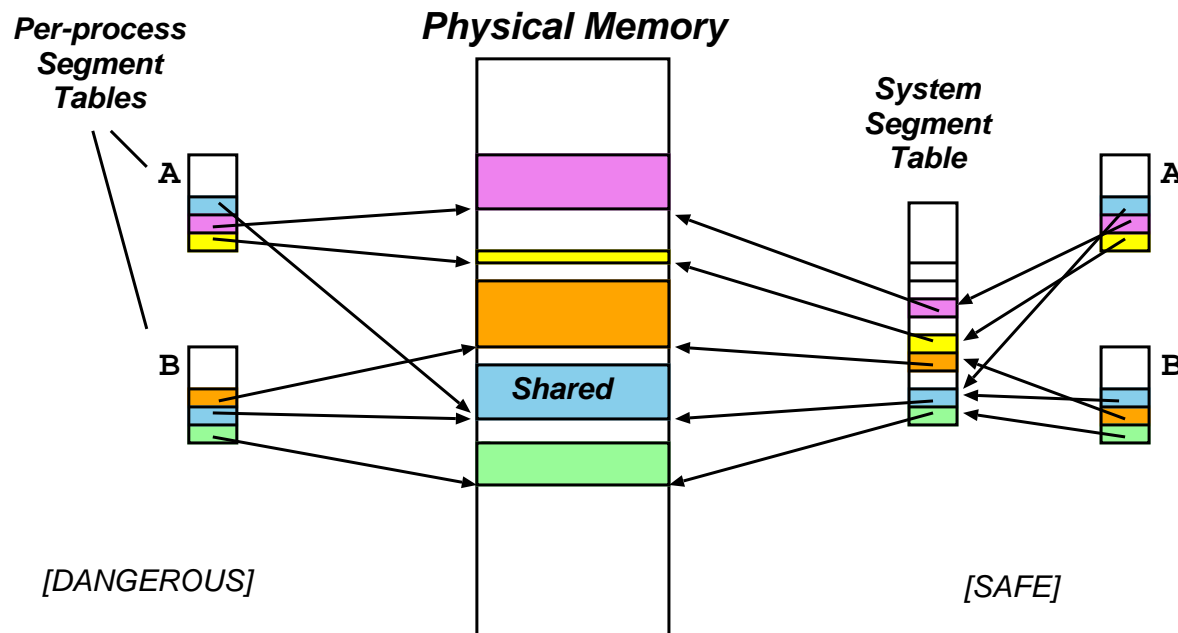
Sharing segments: dangerously (lhs) and safely (rhs)

- wasteful (and dangerous) to store common information on shared segment in each process segment table

  - want *canonical* version of segment info

- assign each segment a unique System Segment Number (SSN)

- process segment table maps from a Process Segment Number (PSN) to SSN

External Fragmentation Returns. . .

- Long term scheduler must find spots in memory for all segments of a program... but segs are of variable size $\Rightarrow$ leads to fragmentation.

- Tradeoff between compaction/delay depends on the distribution of segment sizes. . .

  - One extreme: each process gets exactly 1 segment $\Rightarrow$ reduces to variable sized partitions
  - Another extreme: each byte is a "segment", separately relocated $\Rightarrow$ quadruples memory use!
  - Fixed size small segments $\equiv$ paging!

- In general with small average segment sizes, external fragmentation is small (consider packing small suitcases into boot of car. . .)

Segmentation versus Paging

	logical view	allocation
Segmentation	✔	✘
Paging	✘	✔

$\Rightarrow$ try combined scheme.

- E.g. paged segments (Multics, OS/2)

    - divide each segment s_i into $k = \lceil l_i/2^n \rceil$ pages, where l_i is the limit (length) of the segment and 2^n is the page size.
    - have seperate page table for every segment.

    ✘ high hardware cost / complexity.

    ✘ not very portable.

- E.g. software segments (most modern OSs)

    - consider pages $[m, \ldots, m + l]$ to be a "segment"
    - OS must ensure protection / sharing kept consistent over region.

    ✘ loss in granularity.

    ✔ relatively simple / portable.

Summary (1 of 2)

Old systems directly accessed [physical] memory, which caused some problems, e.g.

- Contiguous allocation:

  - need large lump of memory for process
  - with time, get [external] fragmentation
  $\Rightarrow$ require expensive compaction

- Address binding (i.e. dealing with *absolute* addressing):

  - "int x; x = 5;" $\rightarrow$ "movl $0x5, ????"
  - compile time $\Rightarrow$ must know load address.
  - load time $\Rightarrow$ work every time.
  - what about swapping?

- Portability:

  - how much memory should we assume a "standard" machine will have?
  - what happens if it has less? or more?

Turns out that we can avoid lots of problems by separating concepts of logical or virtual addresses and physical addresses.
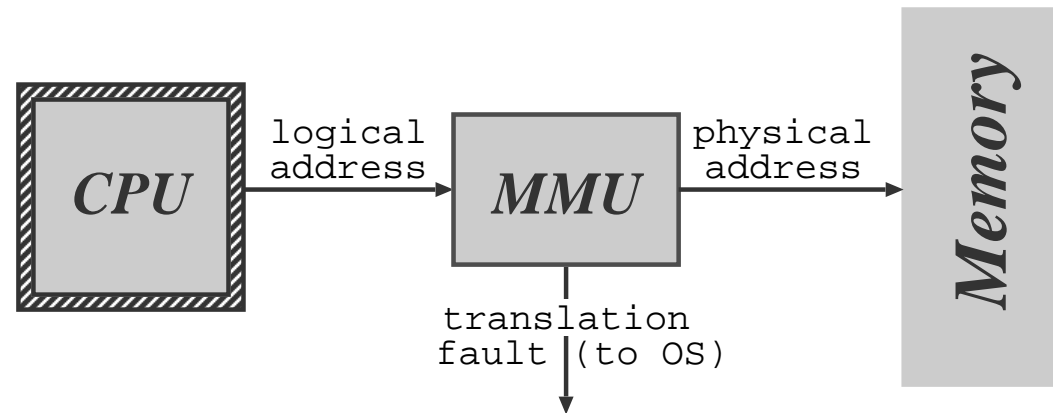
Run time mapping from logical to physical addresses performed by special hardware (the MMU). If we make this mapping a per process thing then:

- Each process has own address space.

- Allocation problem solved (or at least split):
  - virtual address allocation easy.
  - allocate physical memory 'behind the scenes'.

- Address binding solved:
  - bind to logical addresses at compile-time.
  - bind to real addresses at load time/run time.

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

I/O Hardware

- Wide variety of 'devices' which interact with the computer via I/O:

  - Human readable: graphical displays, keyboard, mouse, printers
  - Machine readable: disks, tapes, CD, sensors
  - Communications: modems, network interfaces

- They differ significantly from one another with regard to:

  - Data rate
  - Complexity of control
  - Unit of transfer
  - Direction of transfer
  - Data representation
  - Error handling

$\Rightarrow$ hard to present a uniform I/O system which masks all complexity

I/O subsystem is generally the 'messiest' part of OS.

I/O Subsystem

Diagram labels:
- *Unpriv* / *Priv* / *H/W* (left side)
- **Application-I/O Interface** — *Virtual Device Layer*
- **I/O Buffering**, **I/O Scheduling** — *Common I/O Functions*
- **Device Driver** ... **Device Driver** — *Device Driver Layer*
- **Keyboard**, **HardDisk** ... **Network** — *Device Layer*
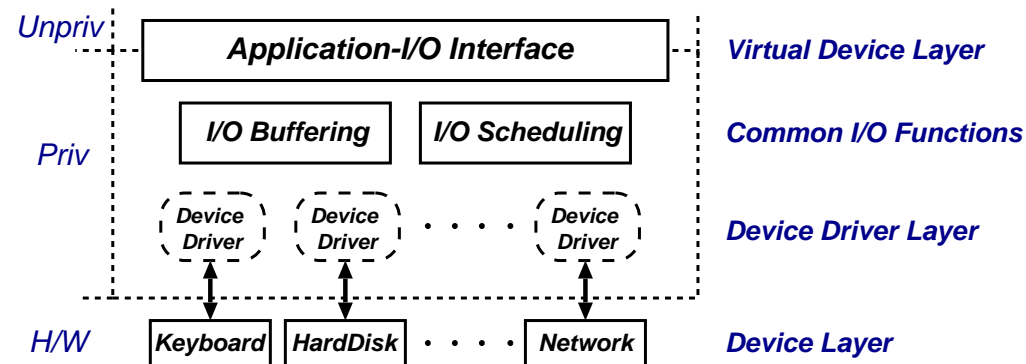
- Programs access virtual devices:

  - terminal streams not terminals
  - windows not frame buffer
  - event stream not raw mouse

  - files not disk blocks
  - printer spooler not parallel port
  - transport protocols not raw ethernet

- OS deals with processor–device interface:

  - I/O instructions versus memory mapped
  - I/O hardware type (e.g. 10's of serial chips)
  - polled versus interrupt driven
  - processor interrupt mechanism

Polled Mode I/O
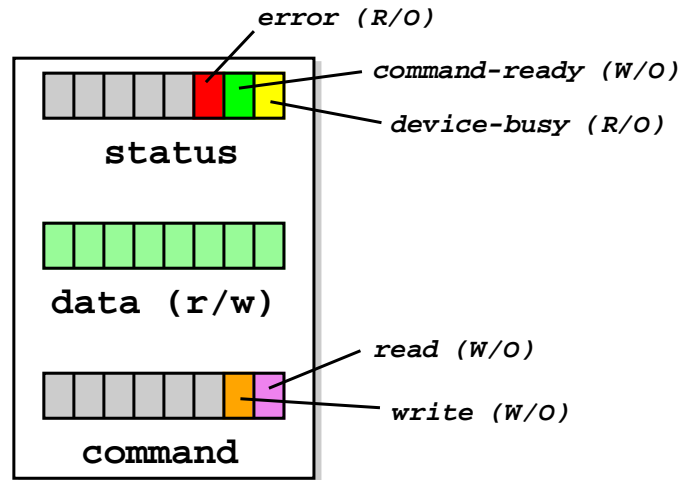
- Consider a simple device with three registers: `status`, `data` and `command`.
- (Host can read and write these via bus)
- Then polled mode operation works as follows:

  - Host repeatedly reads `device_busy` until clear.
  - Host sets e.g. `write` bit in `command` register, and puts data into `data` register.
  - Host sets `command_ready` bit in `status` register.
  - Device sees `command_ready` and sets `device_busy`.
  - Device performs write operation.
  - Device clears `command_ready` & then `device_busy`.

- What's the problem here?

Interrupts Revisited

Recall: to handle mismatch between CPU and device speeds, processors provide an interrupt mechanism:

- at end of each instruction, processor checks interrupt line(s) for pending interrupt

- if line is asserted then processor:
  - saves program counter,
  - saves processor status,
  - changes processor mode, and
  - jump to a well known address (or its contents)

- after interrupt-handling routine is finished, can use e.g. the `rti` instruction to resume where we left off.

Some more complex processors provide:

- multiple levels of interrupts

- hardware vectoring of interrupts

- mode dependent registers

Interrupt-Driven I/O

Can split implementation into low-level *interrupt handler* plus per-device *interrupt service routine*:

- interrupt handler (processor-dependent) may:

  - save more registers
  - establish a language environment (e.g. a C run-time stack)
  - demultiplex interrupt in software.
  - invoke appropriate interrupt service routine (ISR)

- Then interrupt service routine (device-specific but not processor-specific) will:

  1. for programmed I/O device:
     - transfer data.
     - clear interrupt (sometimes a side effect of tx).
  1. for DMA device:
     - acknowledge transfer.
  2. request another transfer if there are any more I/O requests pending on device.
  3. signal any waiting processes.
  4. enter scheduler or return.

Question: who is scheduling who?

Device Classes

Homogenising device API completely not possible

$\Rightarrow$ OS generally splits devices into four *classes*:

1. Block devices (e.g. disk drives, CD):

   - commands include `read`, `write`, `seek`
   - raw I/O or file-system access
   - memory-mapped file access possible

2. Character devices (e.g. keyboards, mice, serial ports):

   - commands include `get`, `put`
   - libraries layered on top to allow line editing

3. Network Devices

   - varying enough from block and character to have own interface
   - Unix and Windows/NT use *socket* interface

4. Miscellaneous (e.g. clocks and timers)

   - provide current time, elapsed time, timer
   - `ioctl` (on UNIX) covers odd aspects of I/O such as clocks and timers.

I/O Buffering

- Buffering: OS stores (its own copy of) data in memory while transferring to or from devices

  - to cope with device speed mismatch
  - to cope with device transfer size mismatch
  - to maintain "copy semantics"

- OS can use various kinds of buffering:

  1. single buffering — OS assigns a system buffer to the user request
  2. double buffering — process consumes from one buffer while system fills the next
  3. circular buffers — most useful for bursty I/O

- Many aspects of buffering dictated by device type:

  - character devices $\Rightarrow$ line probably sufficient.
  - network devices $\Rightarrow$ bursty (time & space).
  - block devices $\Rightarrow$ lots of fixed size transfers.
  - (last usually major user of buffer memory)

Blocking v. Nonblocking I/O

From the programmer's point of view, I/O system calls exhibit one of three kinds of behaviour:

1. Blocking: process suspended until I/O completed

   • easy to use and understand.
   • insufficient for some needs.

2. Nonblocking: I/O call returns as much as available

   • returns almost immediately with count of bytes read or written (possibly 0).
   • can be used by e.g. user interface code.
   • essentially application-level "polled I/O".

3. Asynchronous: process continues to run while I/O executes

   • I/O subsystem explicitly signals process when its I/O request has completed.
   • most flexible (and potentially efficient).
   • . . . but also most difficult to use.

Most systems provide both blocking and non-blocking I/O interfaces; modern systems (e.g. NT, Linux) also support asynchronous I/O, but used infrequently.

Other I/O Issues

- Caching: fast memory holding copy of data

  - can work with both reads and writes
  - key to I/O performance

- Scheduling:

  - e.g. ordering I/O requests via per-device queue
  - some operating systems try fairness. . .

- Spooling: queue output for a device

  - useful for "single user" devices which can serve only one request at a time (e.g. printer)

- Device reservation:

  - system calls for acquiring or releasing exclusive access to a device (careful!)

- Error handling:

  - e.g. recover from disk read, device unavailable, transient write failures, etc.
  - most I/O system calls return an error number or code when an I/O request fails
  - system error logs hold problem reports.

I/O and Performance

- I/O is a major factor in overall system performance

  - demands CPU to execute device driver, kernel I/O code, etc.
  - context switches due to interrupts
  - data copying, buffering, etc
  - (network traffic especially stressful)

- Improving performance:

  - reduce number of context switches
  - reduce data copying
  - reduce # interrupts by using large transfers, smart controllers, adaptive polling (e.g. Linux NAPI)
  - use DMA where possible
  - balance CPU, memory, bus and I/O for best throughput.

Improving I/O performance is a major remaining OS challenge

File Management
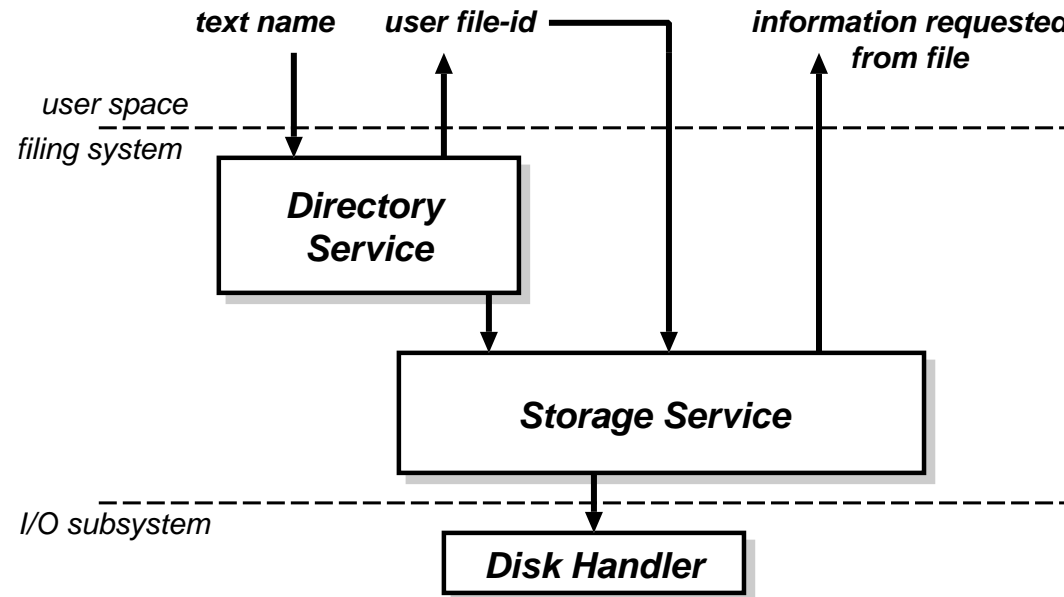
Filing systems have two main components:

1. Directory Service

   - maps from names to file identifiers.
   - handles access & existence control

2. Storage Service

   - provides mechanism to store data on disk
   - includes means to implement directory service

File Concept

- Basic abstraction for non-volatile storage.

- Typically comprises a single contiguous logical address space.

- Internal structure:

  1. None (e.g. sequence of words, bytes)
  2. Simple record structures
     - lines
     - fixed length
     - variable length
  3. Complex structures
     - formatted document
     - relocatable object file

- Can simulate 2,3 with byte sequence by inserting appropriate control characters.

- All a question of who decides:

  - operating system
  - program(mer).

Naming Files

Files usually have at least two kinds of 'name':

1. system file identifier (SFID):

   - (typically) a unique integer value associated with a given file
   - SFIDs are the names used within the filing system itself

2. human-readable name, e.g. `hello.java`

   - what users like to use
   - mapping from human name to SFID is held in a *directory*, e.g.

Name	SFID
hello.java	12353
Makefile	23812
README	9742

   - directories also non-volatile $\Rightarrow$ must be stored on disk along with files.

3. Frequently also get user file identifier (UFID)

   - used to identify *open* files (see later)

File Meta-data
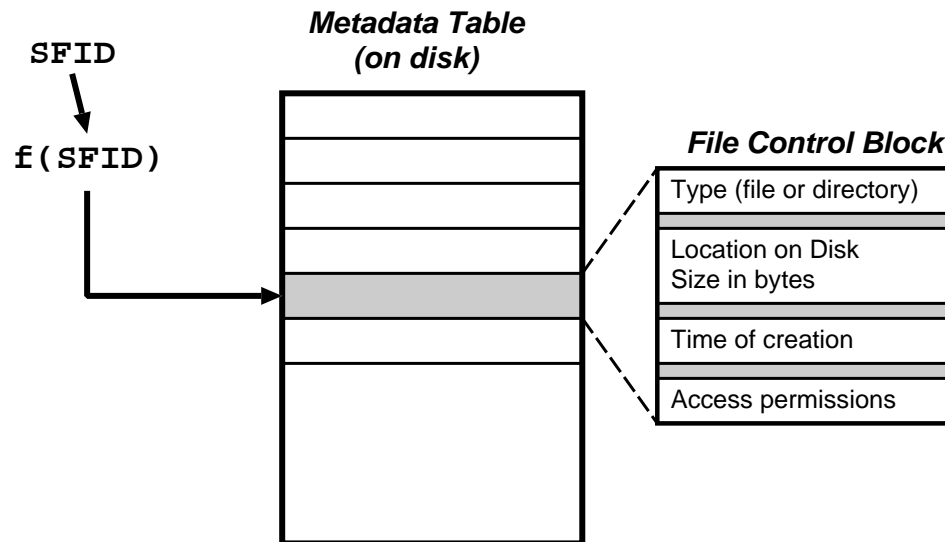
As well as their contents and their name(s), files can have other attributes, e.g.

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

Together this information is called **meta-data**. It is contained in a file control block.

Directory Name Space (I)

What are the requirements for our name space?

- Efficiency: locating a file quickly.

- Naming: user convenience
  - allow two (or more generally N) users to have the same name for different files
  - allow one file have several different names

- Grouping: logical grouping of files by properties (e.g. all Java programs, all games)

First attempts:

- Single-level: one directory shared between all users

  $\Rightarrow$ naming problem
  $\Rightarrow$ grouping problem

- Two-level directory: one directory per user

  - access via $pathname$ (e.g. `bob:hello.java`)
  - can have same filename for different user
  - but still no grouping capability.

Directory Name Space (II)
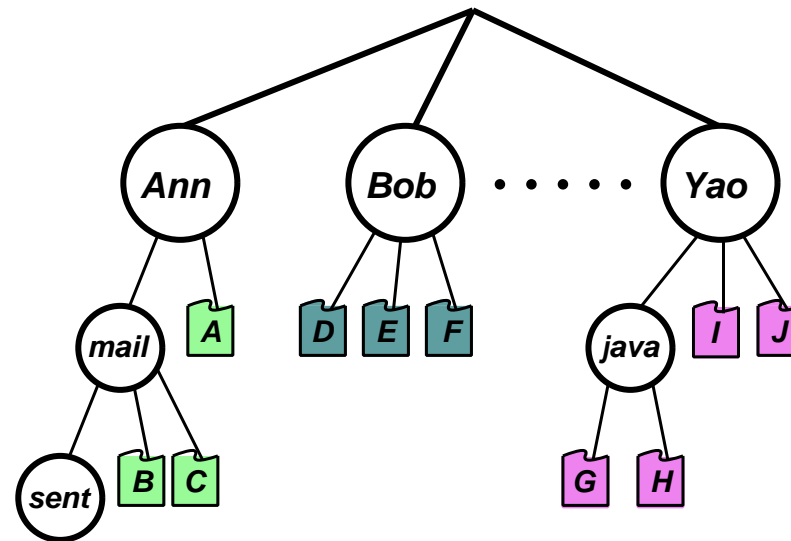
- Get more flexibility with a general hierarchy.

  – directories hold files or [further] directories
  – create/delete files relative to a given directory

- Human name is full path name, but can get long:
  e.g. /usr/groups/X11R5/src/mit/server/os/4.2bsd/utils.c

  – offer relative naming
  – login directory
  – current working directory

- What does it mean to delete a [sub]-directory?

Directory Name Space (III)
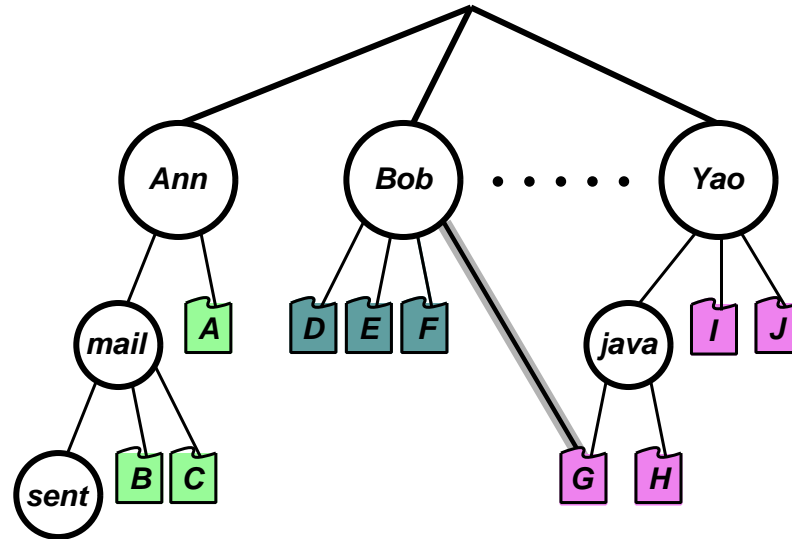
- Hierarchy good, but still only one name per file.

$\Rightarrow$ extend to directed acyclic graph (DAG) structure:

  - allow shared subdirectories and files.
  - can have multiple aliases for the same thing

- **Problem**: dangling references
- Solutions:

  - back-references (but require variable size records); or
  - reference counts.

- **Problem**: cycles. . .

Directory Implementation

/Ann/mail/B

Name	D	SFID
Ann	Y	1034
Bob	Y	179
⋮	⋮	
Yao	Y	7182

Name	D	SFID
mail	Y	2165
A	N	5797

Name	D	SFID
sent	Y	434
B	N	2459
C	N	25
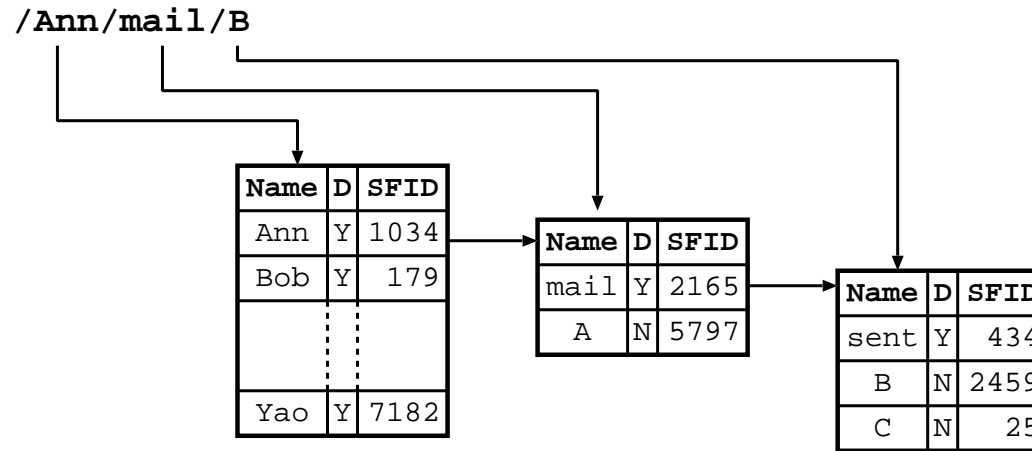
- Directories are non-volatile ⇒ store as "files" on disk, each with own SFID.

- Must be different types of file (for traversal)

- Explicit directory operations include:

  – create directory
  – delete directory
  – list contents
  – select current working directory
  – insert an entry for a file (a "link")

File Operations (I)

UFID	SFID	File Control Block (Copy)	
1	23421	location on disk, size,...	
2	3250	"	"
3	10532	"	"
4	7122	"	"

- Opening a file: `UFID = open(<pathname>)`

  1. directory service recursively searches for components of `<pathname>`
  2. if all goes well, eventually get `SFID` of file.
  3. copy file control block into memory.
  4. create new `UFID` and return to caller.

- Create a new file: `UFID = create(<pathname>)`

- Once have `UFID` can read, write, etc.

  – various modes (see next slide)

- Closing a file: `status = close(UFID)`

  1. copy [new] file control block back to disk.
  2. invalidate `UFID`

File Operations (II)
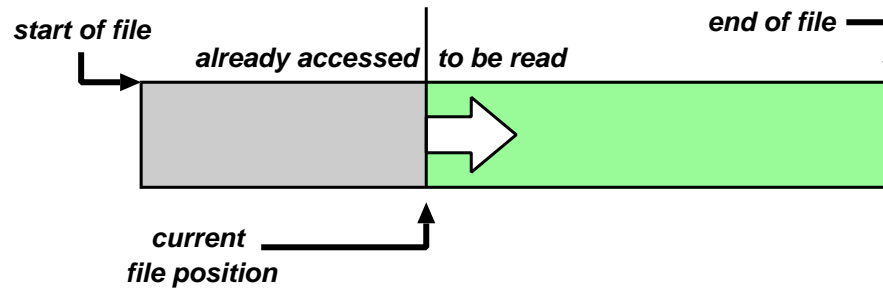
- Associate a cursor or file position with each open file (viz. UFID)

  - initialised at open time to refer to start of file.

- Basic operations: *read next* or *write next*, e.g.

  - `read(UFID, buf, nbytes)`, or `read(UFID, buf, nrecords)`

- Sequential Access: above, plus `rewind(UFID)`.

- Direct Access: *read N* or *write N*

  - allow "random" access to any part of file.
  - can implement with `seek(UFID, pos)`

- Other forms of data access possible, e.g.

  - append-only (may be faster)
  - indexed sequential access mode (ISAM)

Other Filing System Issues

- **Access Control**: file owner/creator should be able to control what can be done, and by whom.

  - normally a function of directory service $\Rightarrow$ checks done at file $open$ time
  - various types of access, e.g.
    * read, write, execute, (append?),
    * delete, list, rename
  - more advanced schemes possible (see later)

- **Existence Control**: what if a user deletes a file?

  - probably want to keep file in existence while there is a valid pathname referencing it
  - plus check entire FS periodically for garbage
  - existence control can also be a factor when a file is renamed/moved.

- **Concurrency Control**: need some form of $locking$ to handle simultaneous access

  - may be mandatory or advisory
  - locks may be shared or exclusive
  - granularity may be file or subset

Protection

Require protection against unauthorised:

- release of information
  - reading or leaking data
  - violating privacy legislation
  - using proprietary software
  - covert channels

- modification of information
  - can also "sabotage" by deleting or hiding information

- denial of service
  - causing a crash
  - causing high load (e.g. processes or packets)
  - changing access rights

Also wish to protect against the effects of errors:

- isolate for debugging
- isolate for damage control

In general, protection mechanisms impose controls on access by $subjects$ (e.g. users) on $objects$ (e.g. files or devices or other processes).

Protection and Sharing

If we have a single user machine with no network connection in a locked room then protection is easy. . . but in practice we want to:

- share facilities (for economic reasons)

- share and exchange data (application requirement)

Some mechanisms we have already come across:

- user and supervisor levels

    - usually one of each
    - could have several (e.g. MULTICS rings)

- memory management hardware

    - relocation hardware
    - bounds checking
    - separate address spaces

- files

    - access control list
    - groups etc

Design of Protection System

- Some other protection mechanisms:

  - lock the computer room (prevent people from tampering with the hardware)
  - restrict access to system software
  - de-skill systems operating staff
  - keep designers away from final system!
  - use passwords (in general challenge/response)
  - use encryption
  - legislate

- ref: Saltzer + Schroeder Proc. IEEE Sept 75

  - design should be public
  - default should be no access
  - check for current authority
  - give each process minimum possible authority
  - mechanisms should be simple, uniform and built in to lowest layers
  - should be psychologically acceptable
  - cost of circumvention should be high
  - minimize shared access

Authentication of User to System (1)

Passwords currently widely used:

- ideally want a long sequence of random characters issued by system, but user would write it down

- if allow user selection, they will use dictionary words, car registration, their name, their date of birth, their pet's name, their pet's date of birth, etc.

- best bet probably is to encourage the use of an algorithm to remember password

- other top tips:

  - don't reflect on terminal, or overprint
  - add delay after failed attempt
  - use encryption if line suspect

- what about security of password file?

  - only accessible to login program (CAP, TITAN)
  - hold scrambled, e.g. UNIX
    * only need to write protect file
    * need scrambling to work without password (e.g. use a "one way" function)

Authentication of User to System (2)

E.g. passwords in UNIX:

- simple for user to remember

      arachnid

- sensible user applies an algorithm

      !r!chn#d

- password is DES-encrypted 25 times using a 2-byte per-user 'salt' to produce a 11 byte string

- salt followed by these 11 bytes are then stored

      IML.DVMcz6Sh2

Really require unforgeable evidence of identity that system can check:

- enhanced password: challenge-response.

- id card inserted into slot

- fingerprint, voiceprint, face recognition

- smart cards

Authentication of System to User

User wants to avoid

- talking to the wrong computer
- or talking to the right computer, but not the login program :-)

Partial solution in old days for directly wired terminals:

- make login character same as terminal attention, or
- always do a terminal attention before trying login

But, today PCs used as terminals $\Rightarrow$

- local software may have been changed
- so carry your own copy of the terminal program
- but hardware / firmware in public machine may have been modified

Anyway, still have the problem of comms lines:

- wiretapping is easy
- workstation can often see all packets on network

$\Rightarrow$ must use encryption of some kind, and trust encryption device (e.g. a smart card)

Mutual suspicion

- We need to encourage lots and lots of suspicion:

  - system of user
  - users of each other
  - user of system

- Called programs should be suspicious of caller (e.g. OS calls always need to check parameters)

- Caller should be suspicious of called program

- e.g. Trojan horse:

  - a 'useful' looking program — a game perhaps
  - when called by user (in many systems) inherits all of the user's privileges
  - it can then copy files, modify files, change password, send mail, etc. . .
  - e.g. Multics editor trojan horse, copied files as well as edited.

- e.g. Virus:

  - often starts off as Trojan horse
  - self-replicating (e.g. ILOVEYOU, Code Red, Stuxnet)

Access matrix

Access matrix is a matrix of subjects against objects.

Subject (or principal) might be:

- users e.g. by uid
- executing process in a protection domain
- sets of users or processes

Objects are things like:

- files
- devices
- domains / processes
- message ports (in microkernels)

Matrix is large and sparse $\Rightarrow$ don't want to store it all.

Two common representations:

1. by object: store list of subjects and rights with each object $\Rightarrow$ *access control list*

2. by subject: store list of objects and rights with each subject $\Rightarrow$ *capabilities*

Access Control Lists

Often used in storage systems:

- system naming scheme provides for ACL to be inserted in naming path, e.g. files

- if ACLs stored on disk but check is made in software $\Rightarrow$ must only use on low duty cycle (or performance will suffer)

- for higher duty cycle must cache results of check

- e.g. Multics: open file = memory segment.
  On first reference to segment:

  1. interrupt (segment fault)
  2. check ACL
  3. set up segment descriptor in segment table

- most systems check ACL

  - when file opened for read or write
  - when code file is to be executed

- access control by program allows arbitrary policies (see. e.g. Unix later)

- sometimes even this isn't enough (e.g. MAC)

Capabilities

Capabilities associated with active subjects, so:

- store in address space of subject

- must make sure subject can't forge capabilities

- easily accessible to hardware

- can be used with high duty cycle
  e.g. as part of addressing hardware

  - Plessey PP250
  - CAP I, II, III
  - IBM system/38
  - Intel iAPX432

- have special machine instructions to modify (restrict) capabilities

- support passing of capabilities on procedure (program) call

Can also use *software* capabilities:

- checked by encryption

- nice for distributed systems

Password Capabilities

- Capabilities nice for distributed systems but:

  - messy for application, and
  - revocation is tricky.

- Could use timeouts (e.g. Amoeba).

- Alternatively: combine passwords and capabilities.

- Store ACL with object, but key it on capability (not implicit concept of "principal" from OS).

- Advantages:

  - revocation possible
  - multiple "roles" available.

- Disadvantages:

  - still messy (use 'implicit' cache?).

Covert channels

Information leakage by side-effects: lots of fun!

At the hardware level:

- wire tapping

- monitor signals in machine

- modification to hardware

- electromagnetic radiation of devices

By software:

- leak a bit stream as:

  | file exists | page fault | compute a while | 1 |
  | no file | no page fault | sleep for a while | 0 |

- system may provide statistics
  e.g. TENEX password cracker using system provided count of page faults

In general, guarding against covert channels is prohibitively expensive.

(only usually a consideration for military types)

Unix: Introduction

- Unix first developed in 1969 at Bell Labs (Thompson & Ritchie)

- Originally written in PDP-7 asm, but then (1973) rewritten in the 'new' high-level language C

  $\Rightarrow$ easy to port, alter, read, etc.

- 6th edition ("V6") was widely available (1976).

  - source avail $\Rightarrow$ people could write new tools.
  - nice features of other OSes rolled in promptly.

- By 1978, V7 available (for both the 16-bit PDP-11 and the new 32-bit VAX-11).

- Since then, two main families:

  - AT&T: "System V", currently SVR4.
  - Berkeley: "BSD", currently 4.3BSD/4.4BSD.

- Standardisation efforts (e.g. POSIX, X/OPEN) to homogenise.

- Best known "UNIX" today is probably $linux$, but also get FreeBSD, NetBSD, and (commercially) Solaris, OSF/1, IRIX, and Tru64.

Unix Family Tree (Simplified)

Year					
1969		First Edition			
1973		Fifth Edition			
1974					
1975					
1976		Sixth Edition			
1977					
1978		Seventh Edition	32V		
1979			3BSD		
1980			4.0BSD		
1981			4.1BSD		
1982	System III				
1983	System V	Eighth Edition	4.2BSD		SunOS
1984	SVR2				
1985					
1986			Mach	4.3BSD	SunOS 3
1987	SVR3	Ninth Edition			
1988				4.3BSD/Tahoe	
1989	SVR4	Tenth Edition		4.3BSD/Reno	
1990			OSF/1		SunOS 4
1991					
1992					Solaris
1993				4.4BSD	Solaris 2
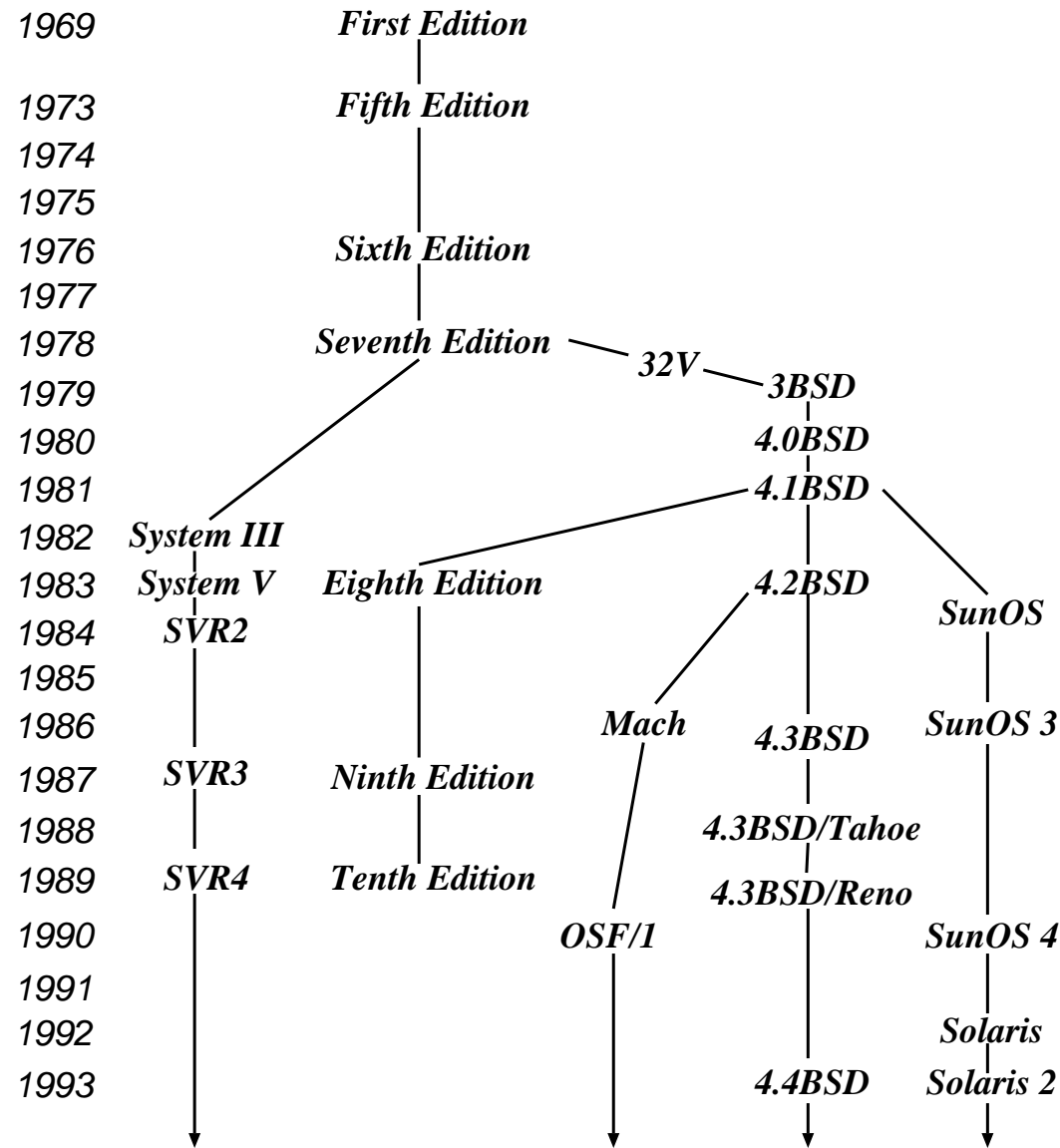
Design Features

Ritchie and Thompson writing in CACM, July 74, identified the following (new) features of UNIX:

1. A hierarchical file system incorporating demountable volumes.

2. Compatible file, device and inter-process I/O.

3. The ability to initiate asynchronous processes.

4. System command language selectable on a per-user basis.

5. Over 100 subsystems including a dozen languages.
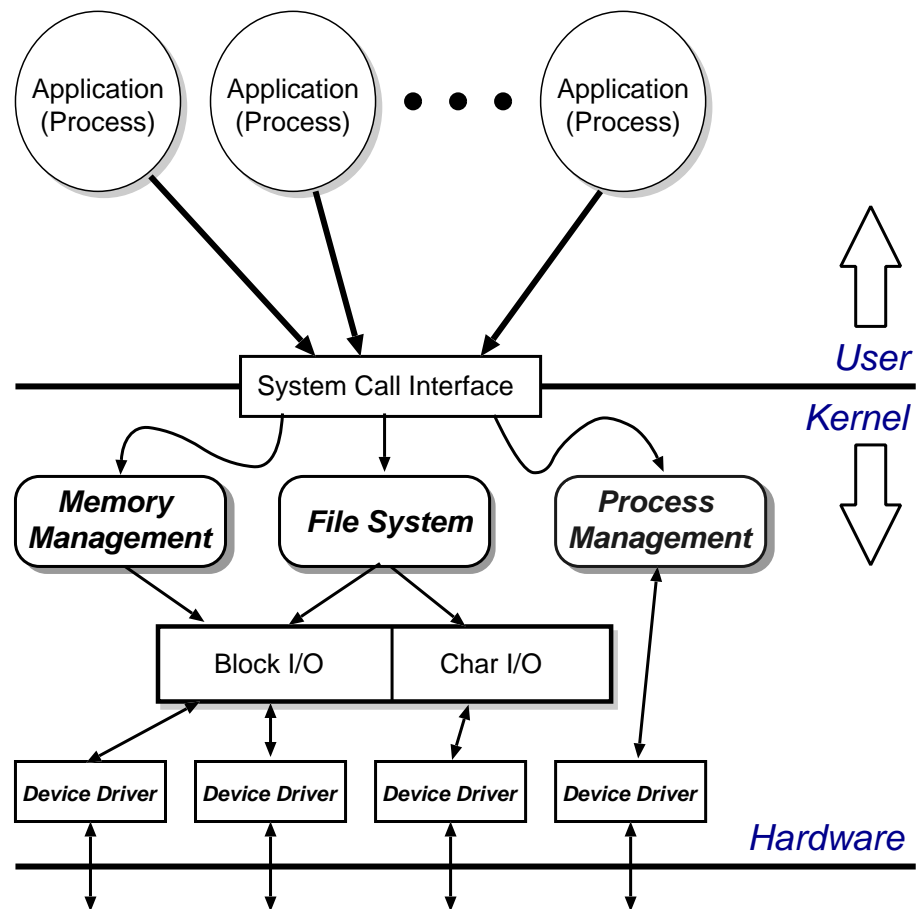
6. A high degree of portability.

Features which were not included:

- real time

- multiprocessor support

Fixing the above is pretty hard.

Structural Overview

- Clear separation between user and kernel portions.

- Processes are unit of scheduling and protection.
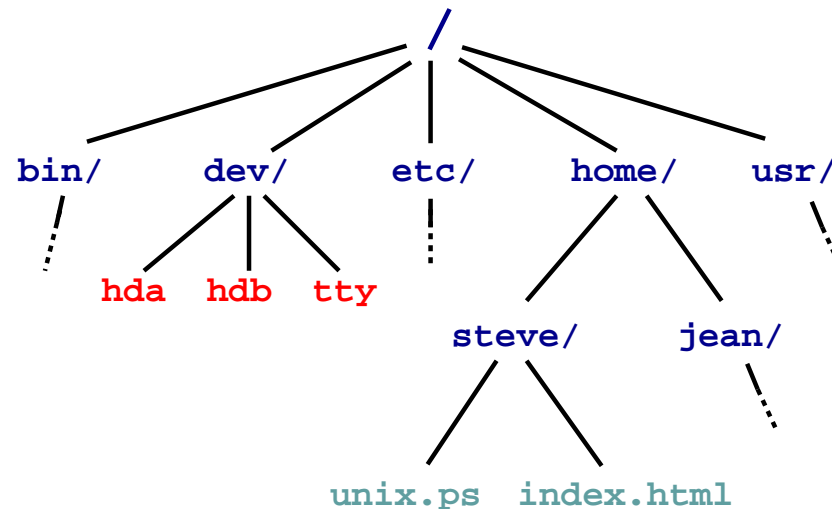
- All I/O looks like operations on files.

File Abstraction

- A file is an unstructured sequence of bytes.

- Represented in user-space by a *file descriptor* (fd)

- Operations on files are:

  - *fd* = **open** (*pathname*, *mode*)
  - *fd* = **creat**(*pathname*, *mode*)
  - bytes = **read**(*fd*, *buffer*, *nbytes*)
  - count = **write**(*fd*, *buffer*, *nbytes*)
  - reply = **seek**(*fd*, *offset*, *whence*)
  - reply = **close**(*fd*)

- Devices represented by special files:

  - support above operations, although perhaps with bizarre semantics.
  - also have `ioctl`'s: allow access to device-specific functionality.

- Hierarchical structure supported by directory files.

Directory Hierarchy

```
                            /
          _____/_____|_____
         /        |         |       \       \
      bin/      dev/      etc/     home/     usr/
       ⁝       / |  \      ⁝       /    \      ⁝
            hda hdb tty          steve/  jean/
                                 /    \       ⁝
                            unix.ps  index.html
```
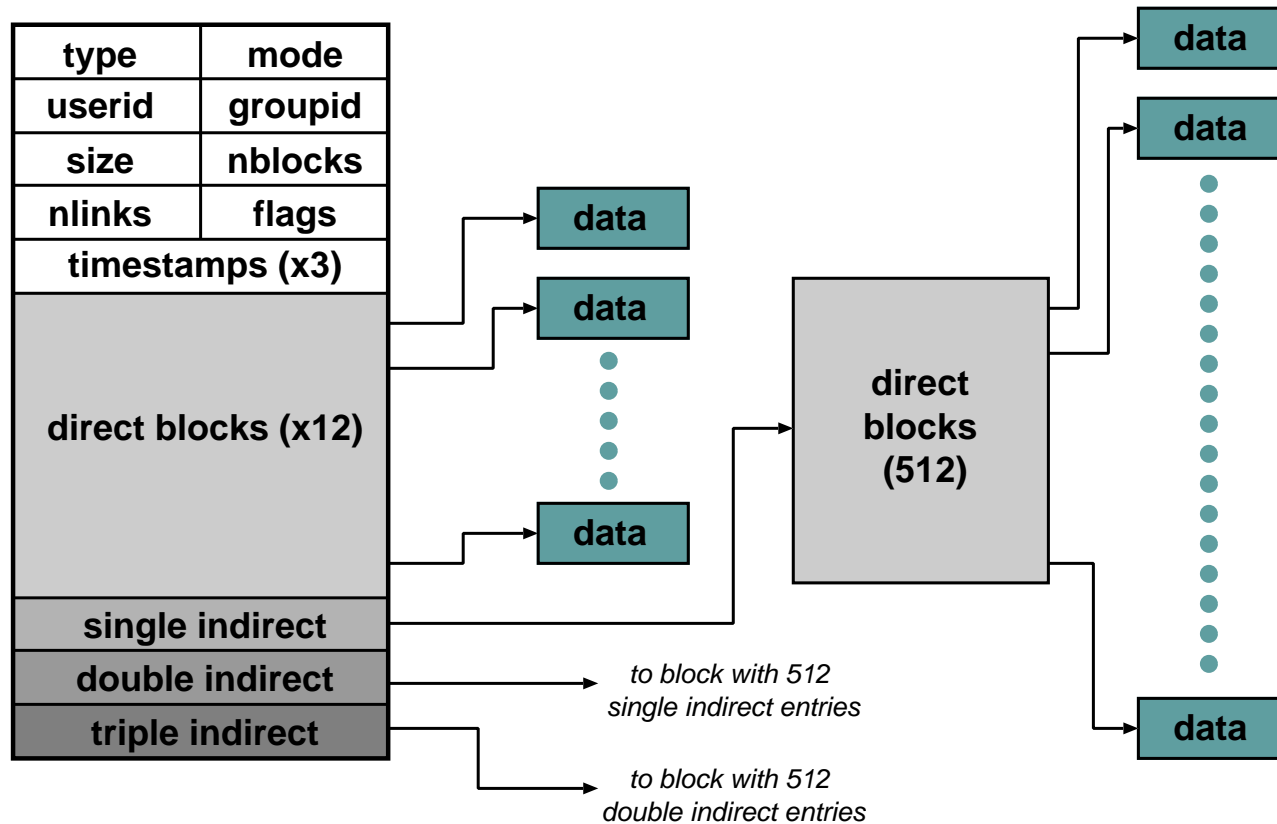
- Directories map names to files (and directories).

- Have distinguished root directory called '/'

- Fully qualified pathnames ⇒ perform traversal from root.

- Every directory has '.' and '..' entries: refer to self and parent respectively.

- Shortcut: current working directory (cwd).

- In addition shell provides access to home directory as *~username* (e.g. ~steve/)

Aside: Password File

- `/etc/passwd` holds list of password entries.

- Each entry roughly of the form:

    user-name:*encrypted-passwd*:*home-directory*:*shell*

- Use one-way function to encrypt passwords.

  - i.e. a function which is easy to compute in one direction, but has a hard to compute inverse (e.g. person to phone-number lookup).

- To login:

  1. Get user name
  2. Get password
  3. Encrypt password
  4. Check against version in `/etc/password`
  5. If ok, instantiate login shell.

- Publicly readable since lots of useful info there.

- **Problem**: off-line attack.
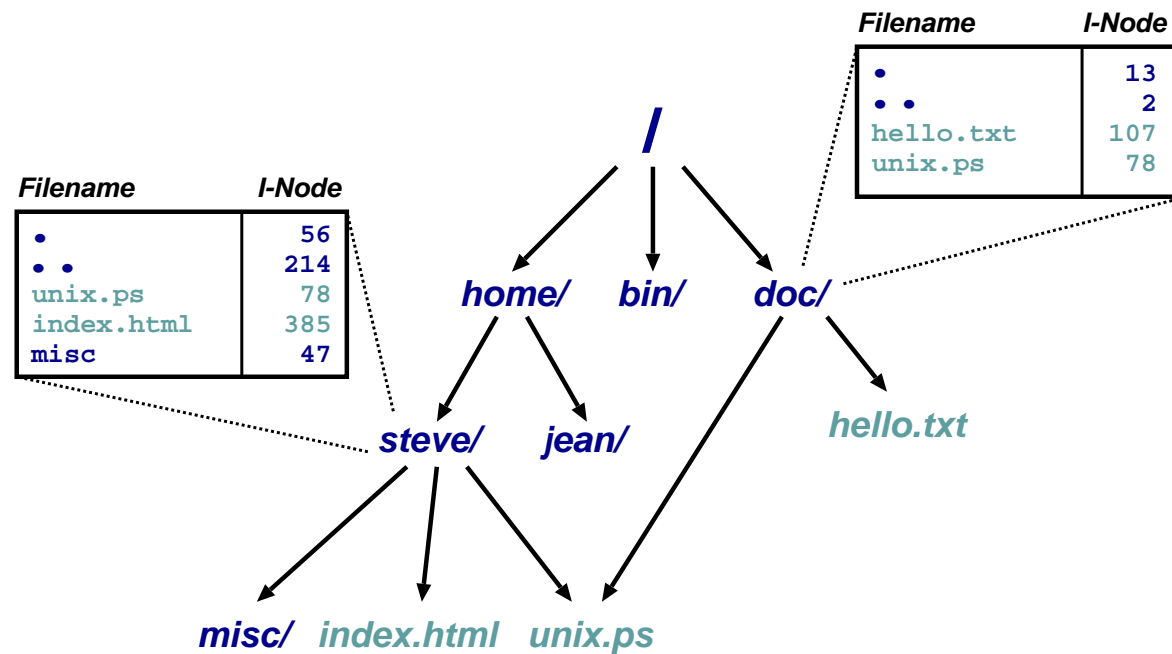
- **Solution**: shadow passwords (`/etc/shadow`)

File System Implementation

type	mode
userid	groupid
size	nblocks
nlinks	flags
timestamps (x3)	
direct blocks (x12)	
single indirect	
double indirect	
triple indirect	

data

data

data

direct blocks (512)

to block with 512 single indirect entries

to block with 512 double indirect entries

data

data

data

- In kernel, a file is represented by a data structure called an index-node or i-node.

- Holds file meta-data:

  a) Owner, permissions, reference count, etc.
  b) Location on disk of actual data (file contents).

- **Question**: Where is the filename kept?

Filename	I-Node
.	13
. .	2
hello.txt	107
unix.ps	78

Filename	I-Node
.	56
. .	214
unix.ps	78
index.html	385
misc	47

/

home/   bin/   doc/

steve/   jean/

hello.txt

misc/   index.html   unix.ps

- Directory is a file which maps filenames to i-nodes.

- An instance of a file in a directory is a (hard) link.

- (this is why have reference count in i-node).

- Directories can have at most 1 (real) link. Why?

- Also get soft- or symbolic-links: a 'normal' file which contains a filename.

On-Disk Structures
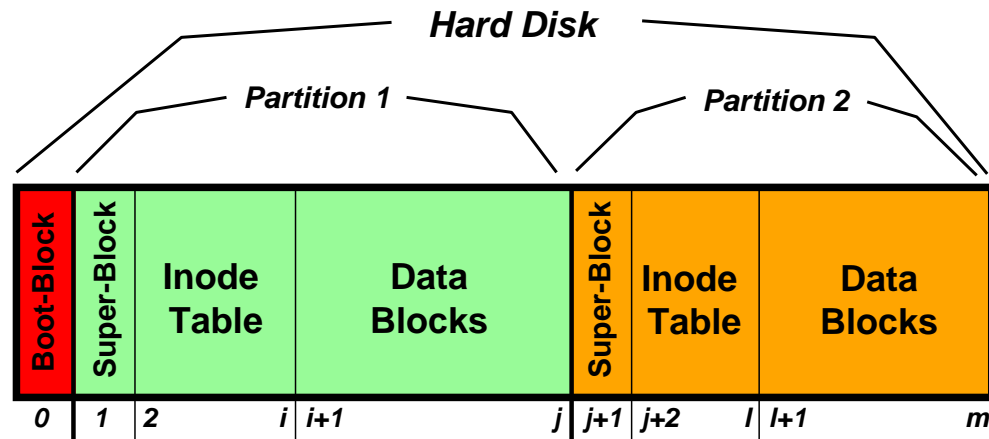

- A disk is made up of a boot block followed by one or more partitions.
- (a partition is a contiguous range of N fixed-size blocks of size k for some N, k).
- A Unix file-system resides within a partition.
- The file-system superblock contains info such as:
  - number of blocks in file-system
  - number of free blocks in file-system
  - start of the free-block list
  - start of the free-inode list.
  - various bookkeeping information.

Mounting File-Systems
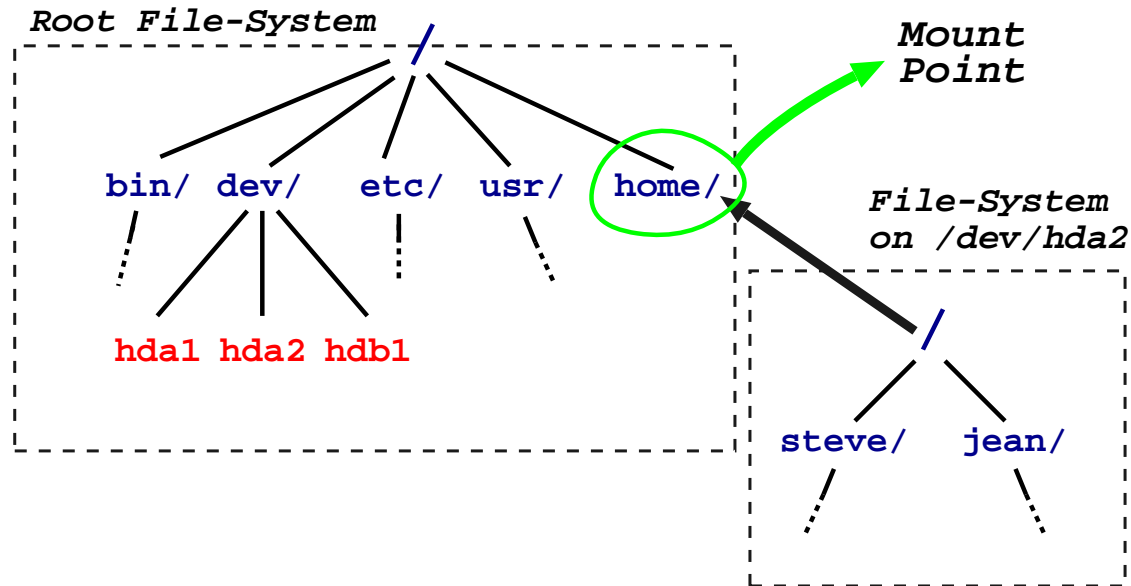
- Entire file-systems can be mounted on an existing directory in an already mounted filesystem.

- At very start, only '/' exists ⇒ need to mount a root file-system.

- Subsequently can mount other file-systems, e.g. `mount("/dev/hda2", "/home", options)`

- Provides a unified name-space: e.g. access `/home/steve/` directly.

- Cannot have hard links across mount points: why?

- What about soft links?

In-Memory Tables
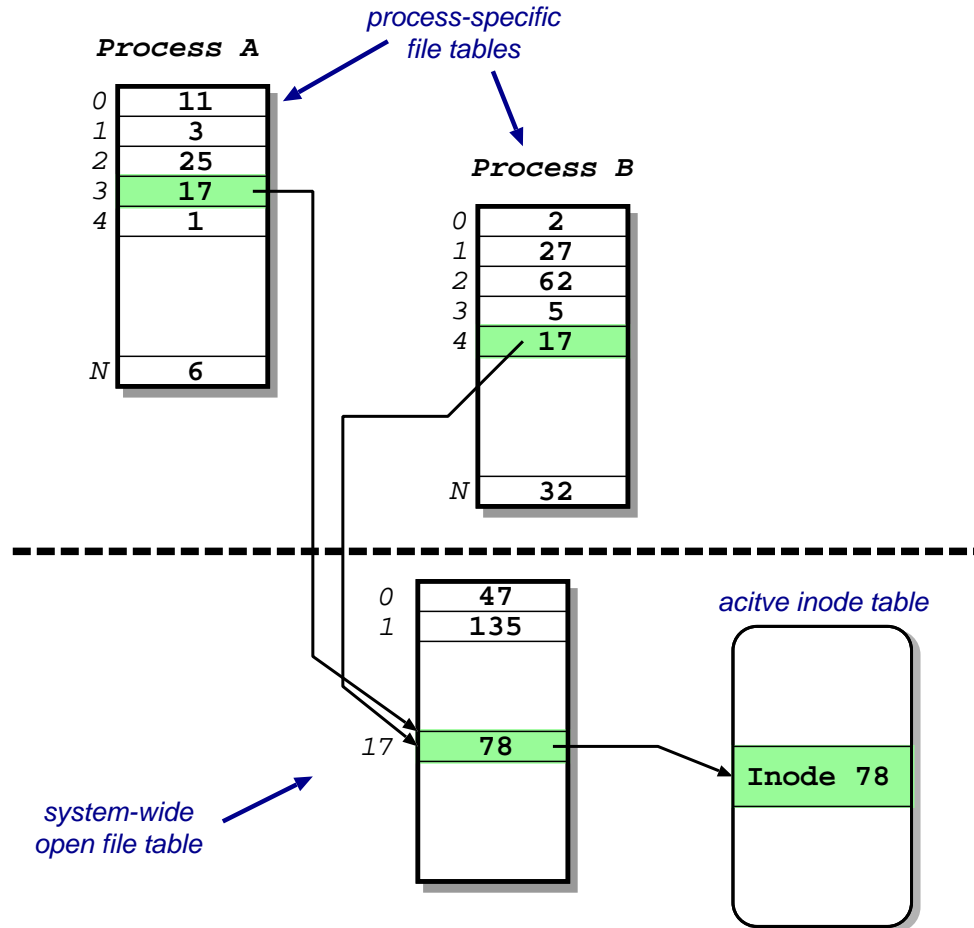
- Recall process sees files as file descriptors
- In implementation these are just indices into a process-specific open file table.
- Entries point to system-wide open file table. Why?
- These in turn point to (in memory) inode table.

Access Control

Owner			Group			World		
R	W	E	R	W	E	R	W	E

= 0640

Owner			Group			World		
R	W	E	R	W	E	R	W	E

= 0755
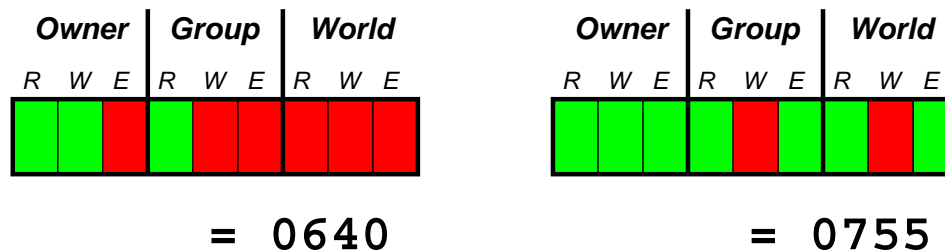
- Access control information held in each inode.

- Three bits for each of owner, group and world: { read, write and execute }

- **Question**: What do these mean for directories?

- In addition have setuid and setgid bits:

  - normally processes inherit permissions of invoking user.
  - setuid/setgid allow the user to "become" someone else when running a particular program.
  - e.g. `prof` owns both executable `test` (0711 and setuid), and `score` file (0600)
    $\Rightarrow$ any user can run it.
    $\Rightarrow$ it can update `score` file.
    $\Rightarrow$ but users can't cheat.

- **Question**: and what do *these* mean for directories?

Consistency Issues

- To delete a file, use the `unlink` system call.

- From the shell, this is `rm <filename>`

- Procedure is:

  1. check if user has sufficient permissions on the file (must have write access).
  2. check if user has sufficient permissions on the directory (must have write access).
  3. if ok, remove entry from directory.
  4. Decrement reference count on inode.
  5. if now zero:
     a. free data blocks.
     b. free inode.

- If the system crashes: must check entire file-system:

  - check if any block unreferenced.
 - check if any block double referenced.

- (We'll see more on this later)

Unix File-System: Summary

- Files are unstructured byte streams.

- Everything is a file: 'normal' files, directories, symbolic links, special files.

- Hierarchy built from root ('/').

- Unified name-space (multiple file-systems may be mounted on any leaf directory).

- Low-level implementation based around inodes.

- Disk contains list of inodes (along with, of course, actual data blocks).

- Processes see file descriptors: small integers which map to system file table.

- Permissions for owner, group and everyone else.

- Setuid/setgid allow for more flexible control.

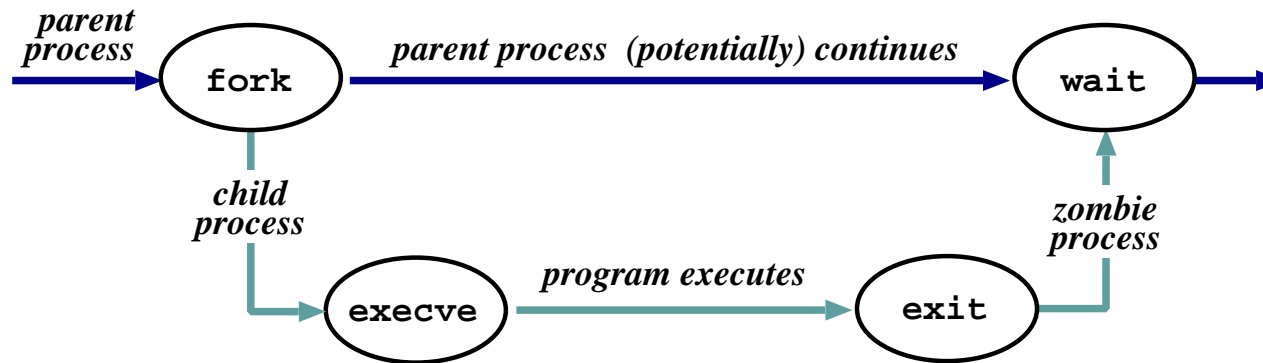- Care needed to ensure consistency.

Unix Processes
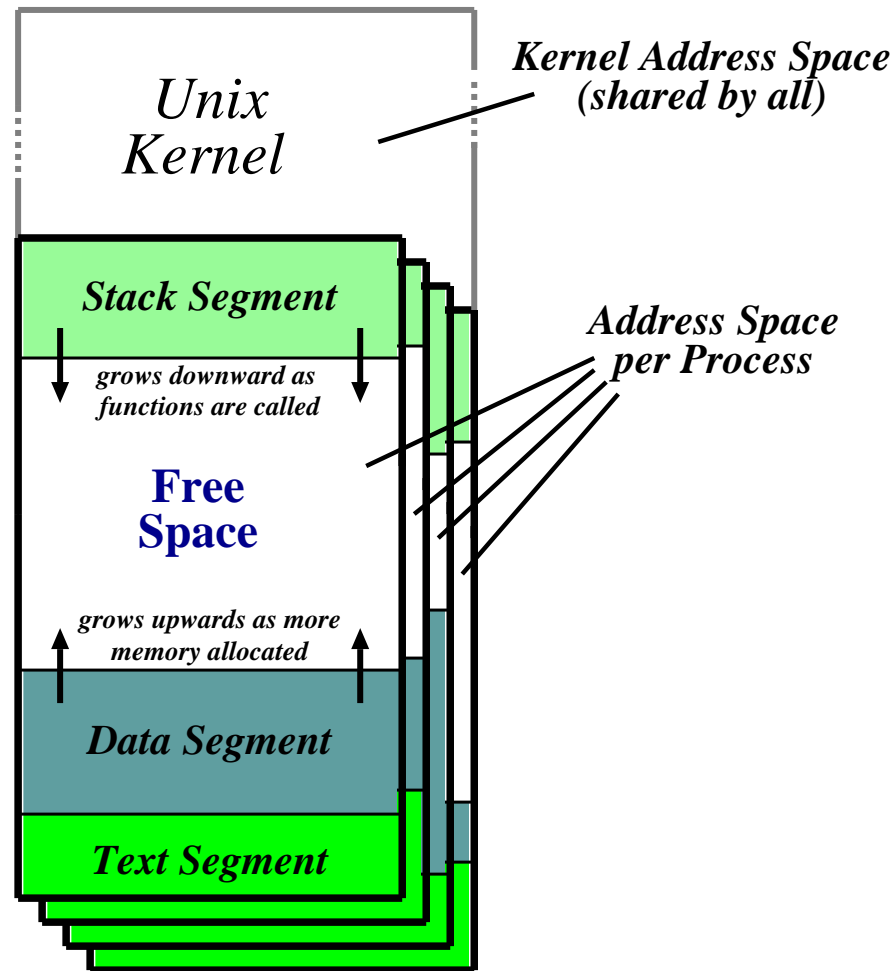
- Recall: a process is a program in execution.

- Have three segments: text, data and stack.

- Unix processes are heavyweight.

Unix Process Dynamics


- Process represented by a *process id* (pid)

- Hierarchical scheme: parents create children.

- Four basic primitives:

  - *pid* = **fork** ()
  - reply = **execve**(*pathname*, *argv*, *envp*)
  - **exit**(*status*)
  - *pid* = **wait** (*status*)

- **fork()** nearly *always* followed by **exec()**

  ⇒ **vfork()** and/or COW.

Start of Day

- Kernel (`/vmunix`) loaded from disk (how?) and execution starts.

- Root file-system mounted.

- Process 1 (`/etc/init`) hand-crafted.

- init reads file `/etc/inittab` and for each entry:

  1. opens terminal special file (e.g. `/dev/tty0`)
  2. duplicates the resulting fd twice.
  3. forks an `/etc/tty` process.

- each tty process next:

  1. initialises the terminal
  2. outputs the string "`login:`" & waits for input
  3. execve()'s `/bin/login`

- login then:

  1. outputs "`password:`" & waits for input
  2. encrypts password and checks it against `/etc/passwd`.
  3. if ok, sets uid & gid, and execve()'s shell.

- Patriarch init resurrects `/etc/tty` on exit.

The Shell
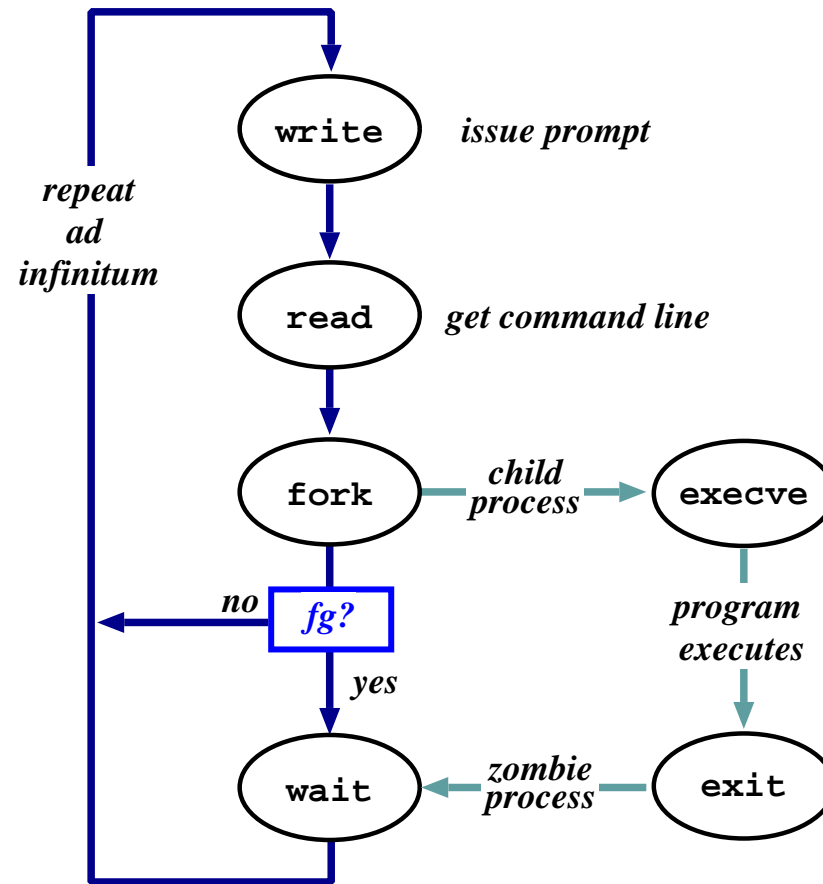
- The shell just a process like everything else.

- Uses path (= list of directories to search) for convenience.

- Conventionally '&' specifies run in background.

- Parsing stage (omitted) can do lots. . .

Shell Examples

```
# pwd
/home/steve
# ls -F
IRAM.micro.ps                gnome_sizes           prog-nc.ps
Mail/                        ica.tgz               rafe/
OSDI99_self_paging.ps.gz     lectures/             rio107/
TeX/                         linbot-1.0/           src/
adag.pdf                     manual.ps             store.ps.gz
docs/                        past-papers/          wolfson/
emacs-lisp/                  pbosch/               xeno_prop/
fs.html                      pepsi_logo.tif
# cd src/
# pwd
/home/steve/src
# ls -F
cdq/          emacs-20.3.tar.gz  misc/       read_mem.c
emacs-20.3/  ispell/              read_mem*  rio007.tgz
# wc read_mem.c
     95      225     2262 read_mem.c
# ls -lF r*
-rwxrwxr-x   1 steve   user      34956 Mar 21  1999 read_mem*
-rw-rw-r--   1 steve   user       2262 Mar 21  1999 read_mem.c
-rw-------   1 steve   user      28953 Aug 27 17:40 rio007.tgz
# ls -l /usr/bin/X11/xterm
-rwxr-xr-x   2 root    system 164328 Sep 24 18:21 /usr/bin/X11/xterm*
```

- Prompt is '#'.

- Use `man` to find out about commands.

- User friendly?

Standard I/O

- Every process has three fds on creation:

  - **stdin**: where to read input from.
  - **stdout**: where to send output.
  - **stderr**: where to send diagnostics.

- Normally inherited from parent, but shell allows redirection to/from a file, e.g.:

  - `ls >listing.txt`
  - `ls >&listing.txt`
  - `sh <commands.sh`.

- Actual file not always appropriate; e.g. consider:

  ```
  ls >temp.txt;
  wc <temp.txt >results
  ```

- Pipeline is better (e.g. `ls | wc >results`)

- Most Unix commands are filters, i.e. read from **stdin** and output to **stdout** $\Rightarrow$ can build almost arbitrarily complex command lines.

- Redirection can cause some buffering subtleties.

Pipes
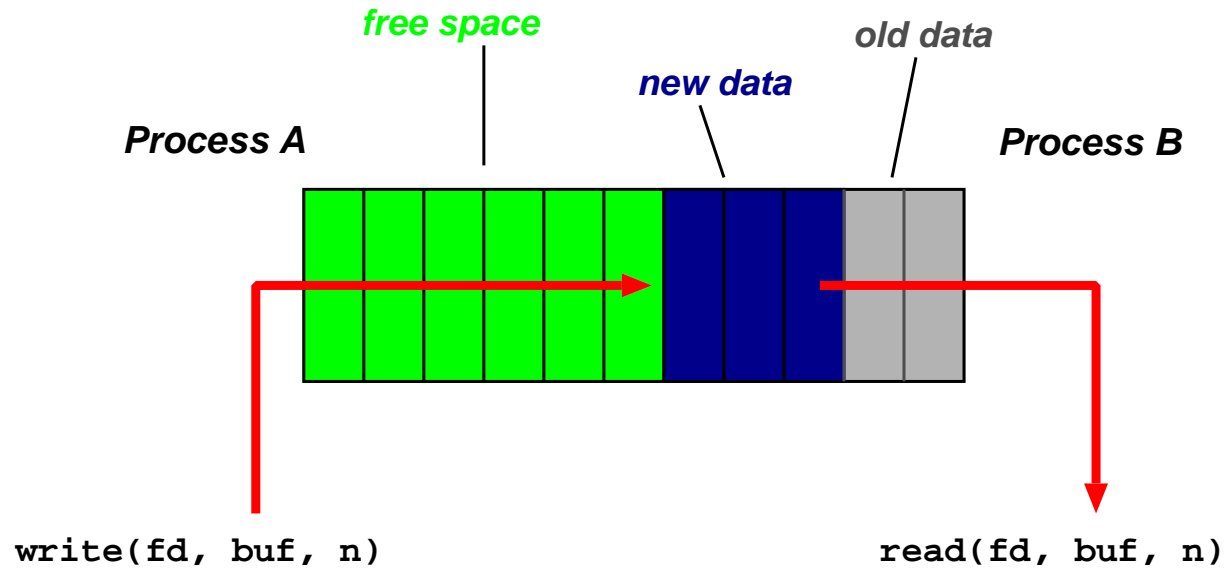
- One of the basic Unix IPC schemes.

- Logically consists of a pair of fds, one for each 'end' of the pipe.

- e.g. reply = **pipe**(int fds[2])

- Concept of "full" and "empty" pipes.

- Only allows communication between processes with a common ancestor (why?).

- Named pipes address this. . .

Signals

- Problem: pipes need planning $\Rightarrow$ use signals.

- Similar to a (software) interrupt.

- Examples:

  - `SIGINT` : user hit Ctrl-C.
  - `SIGSEGV` : program error.
  - `SIGCHLD` : a death in the family. . .
  - `SIGTERM` : . . . or closer to home.

- Unix allows processes to catch signals.

- e.g. Job control:

  - `SIGTTIN`, `SIGTTOU` sent to bg processes
  - `SIGCONT` turns bg to fg.
  - `SIGSTOP` does the reverse.

- Cannot catch `SIGKILL` (hence `kill -9`)

- Signals can also be used for timers, window resize, process tracing, . . .

I/O Implementation

User

Kernel

Generic File System Layer

Buffer
Cache

Cooked
Character I/O

Raw Character I/O

Raw Block I/O

Device Driver | Device Driver | Device Driver | Device Driver

Kernel

Hardware
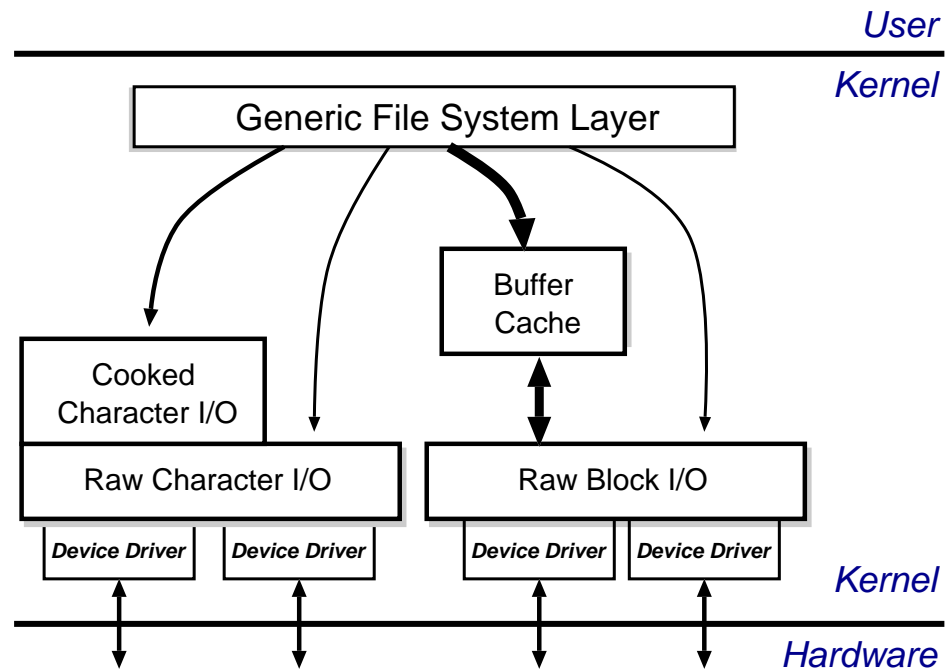
- Recall:
  - everything accessed via the file system.
  - two broad categories: block and char.

- Low-level stuff gory and machine dependent $\Rightarrow$ ignore.

- Character I/O is low rate but complex $\Rightarrow$ most code in the "cooked" interface.

- Block I/O simpler but performance matters $\Rightarrow$ emphasis on the buffer cache.

The Buffer Cache

- Basic idea: keep copy of some parts of disk in memory for speed.

- On read do:

  1. Locate relevant blocks (from inode)
  2. Check if in buffer cache.
  3. If not, read from disk into memory.
  4. Return data from buffer cache.

- On write do $same$ first three, and then update version in cache, not on disk.

- "Typically" prevents 85% of implied disk transfers.

- **Question**: when does data actually hit disk?

- **Answer**: call sync every 30 seconds to flush dirty buffers to disk.

- Can cache metadata too — problems?

Unix Process Scheduling

- Priorities 0–127; user processes $\geq$ PUSER $= 50$.

- Round robin within priorities, quantum 100ms.

- Priorities are based on usage and nice value, i.e.

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

gives the priority of process j at the beginning of interval i where:

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

and $nice_j$ is a (partially) user controllable adjustment parameter $\in [-20, 20]$.

- $load_j$ is the sampled average length of the run queue in which process j resides, over the last minute of operation

- so if e.g. load is 1 $\Rightarrow$ approximately 90% of 1 seconds CPU usage will be "forgotten" within 5 seconds.

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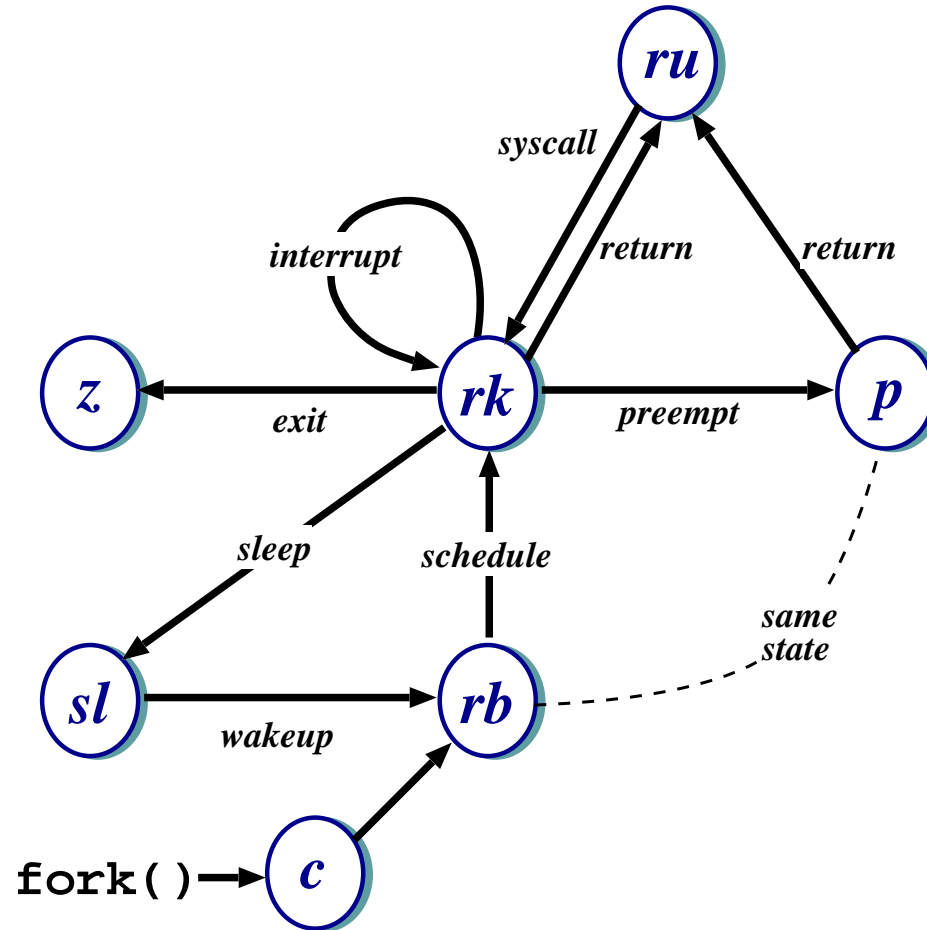
ru = running (user-mode)   rk = running (kernel-mode)
z   = zombie   p   = pre-empted
sl = sleeping   rb = runnable
c   = created

- **Note**: above is simplified — see CS section 23.14 for detailed descriptions of all states/transitions.

Summary

- Main Unix features are:

  - file abstraction
    - ∗ a file is an unstructured sequence of bytes
    - ∗ (not really true for device and directory files)
  - hierarchical namespace
    - ∗ directed acyclic graph (if exclude soft links)
    - ∗ can recursively mount filesystems
  - heavy-weight processes
  - IPC: pipes & signals
  - I/O: block and character
  - dynamic priority scheduling
    - ∗ base priority level for all processes
    - ∗ priority is lowered if process gets to run
    - ∗ over time, the past is forgotten

- But Unix V7 had inflexible IPC, inefficient memory management, and poor kernel concurrency.

- Later versions address these issues.

Windows NT: History

After OS/2, MS decide they need "**N**ew **T**echnology":

- 1988: Dave Cutler recruited from DEC.

- 1989: team ($\sim$ 10 people) starts work on a new OS (micro-kernel architecture)

- July 1993: first version (3.1) introduced

- (name compatible with windows 3.1)

Bloated and suckful $\Rightarrow$

- NT 3.5 released in September 1994: mainly size and performance optimisations.

- Followed in May 1995 by NT 3.51 (support for the Power PC, and more performance tweaks)

- July 1996: NT 4.0
  - new (windows 95) look 'n feel
  - some desktop users but mostly limited to servers
  - for performance reasons, various functions pushed back into kernel (most notably graphics rendering functions)
  - ongoing upgrades via service packs

Windows NT: Evolution

- Feb 2000: NT 5.0 aka Windows 2000

  – borrows from windows 98 look 'n feel
  – both server and workstation versions, latter of which starts to get wider use
  – big push to finally kill DOS/Win 9x family (but fails due to internal politicking)

- Windows XP (NT 5.1) launched October 2001

  – home and professional $\Rightarrow$ finally kills win 9x.
  – various "editions" (media center, 64-bit) & service packs (SP1, SP2, SP3)

- Server product Windows Server 2003 (NT 5.2) released 2003

  – basically the same modulo registry tweaks, support contract and of course **cost**
  – a plethora of editions. . .

- Windows Vista (NT 6.0) limped onto the scene Q4 2006

  – new $Aero$ UI, new $WinFX$ API
  – missing Longhorn bits like $WinFS$, Msh

- Windows Server 2008 (also based on NT 6.0, but good) landed Feb 2008

- Windows 7 (NT 6.1 for now) released October 2009. . .

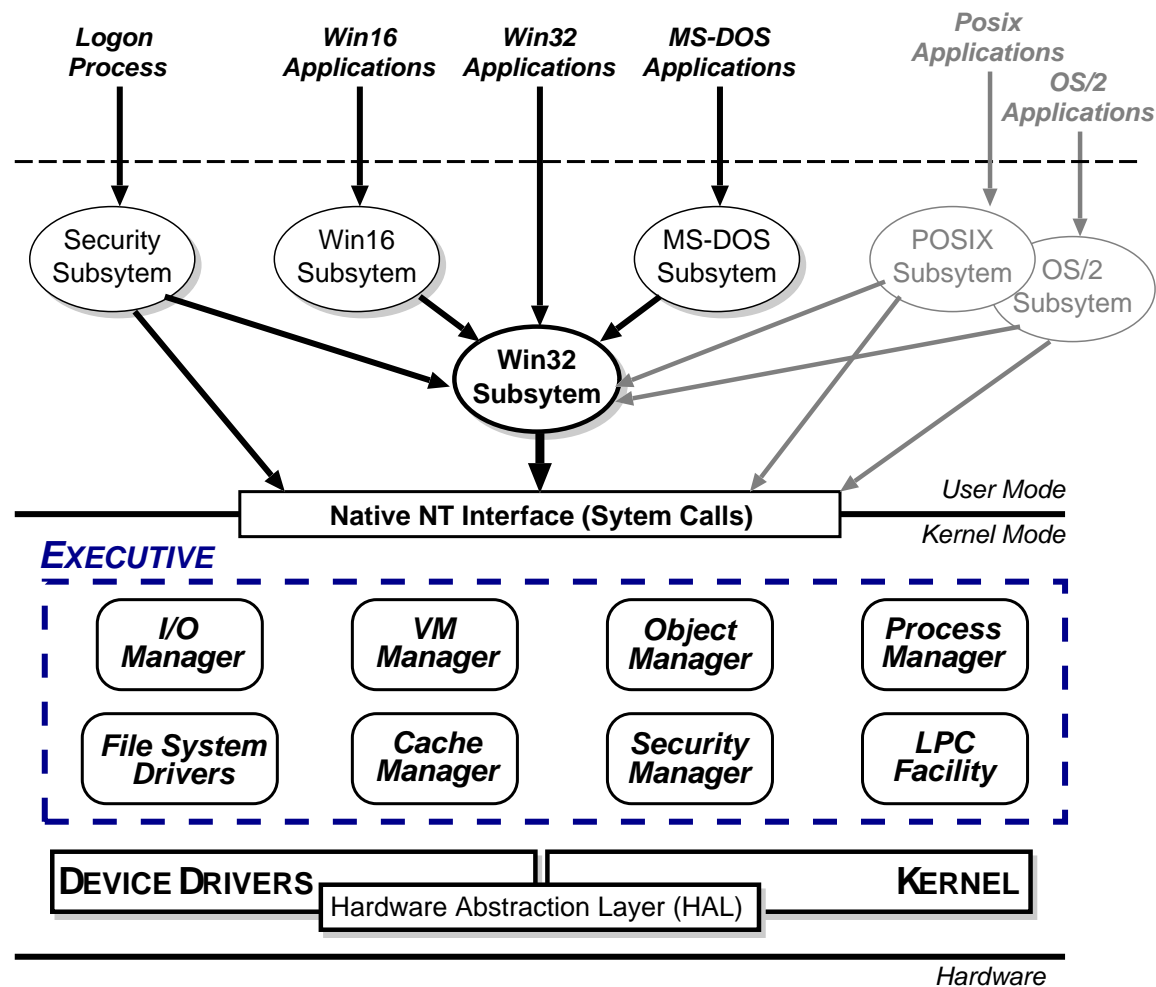# NT Design Principles

Key goals for the system were:

- portability

- security

- POSIX compliance

- multiprocessor support

- extensibility

- international support

- compatibility with MS-DOS/Windows applications

This led to the development of a system which was:

- written in high-level languages (C and C++)

- based around a micro-kernel, and

- constructed in a layered/modular fashion.

Structural Overview

Diagram showing the NT structural overview:

- **Logon Process** → **Security Subsystem**
- **Win16 Applications** → **Win16 Subsystem**
- **Win32 Applications** → **Win32 Subsystem**
- **MS-DOS Applications** → **MS-DOS Subsystem**
- **Posix Applications** → **POSIX Subsystem**
- **OS/2 Applications** → **OS/2 Subsystem**

Security Subsystem, Win16 Subsystem, MS-DOS Subsystem, POSIX Subsystem, and OS/2 Subsystem connect to **Win32 Subsystem** and the **Native NT Interface (Sytem Calls)**.

User Mode / *Kernel Mode*

EXECUTIVE
- *I/O Manager*
- *VM Manager*
- *Object Manager*
- *Process Manager*
- *File System Drivers*
- *Cache Manager*
- *Security Manager*
- *LPC Facility*

DEVICE DRIVERS — Hardware Abstraction Layer (HAL) — **KERNEL**

Hardware

- **Kernel Mode**: HAL, Kernel, & Executive

- **User Mode**: environmental subsystems, protection subsystem

HAL

- Layer of software (`HAL.DLL`) which hides details of underlying hardware

- e.g. low-level interrupt mechanisms, DMA controllers, multiprocessor communication mechanisms

- Several HALs exist with same interface but different implementation (often vendor-specific, e.g. for large cc-NUMA machines)

Kernel

- Foundation for the executive and the subsystems

- Execution is never preempted.

- Four main responsibilities:

  1. CPU scheduling
  2. interrupt and exception handling
  3. low-level processor synchronisation
  4. recovery after a power failure

- Kernel is objected-oriented; all objects are either dispatcher objects (active or temporal things) or control objects (everything else)

Processes and Threads

NT splits the "virtual processor" into two parts:

1. A **process** is the unit of resource ownership.
   Each process has:

   - a security token,
   - a virtual address space,
   - a set of resources (object handles), and
   - one or more threads.

2. A **thread** are the unit of dispatching.
   Each thread has:

   - a scheduling state (ready, running, etc.),
   - other scheduling parameters (priority, etc),
   - a context slot, and
   - (generally) an associated process.

Threads are:

- co-operative: all threads in a process share address space & object handles.

- lightweight: require less work to create/delete than processes (mainly due to shared virtual address space).

CPU Scheduling

- Hybrid static/dynamic priority scheduling:

  - Priorities 16–31: "real time" (static priority).
  - Priorities 1–15: "variable" (dynamic) priority.
  - (priority 0 is reserved for zero page thread)

- Default quantum 2 ticks ($\sim$20ms) on Workstation, 12 ticks ($\sim$120ms) on Server.

- Threads have base and current ($\geq$ base) priorities.

  - On return from I/O, current priority is boosted by driver-specific amount.
  - Subsequently, current priority decays by 1 after each completed quantum.
  - Also get boost for GUI threads awaiting input: current priority boosted to 14 for one quantum (but quantum also doubled)
  - Yes, this is true.

- On Workstation also get quantum stretching:

  - ". . . performance boost for the foreground application" (window with focus)
  - fg thread gets double or triple quantum.

- If no runnable thread, dispatch 'idle' thread (which executes DPCs).

Object Manager
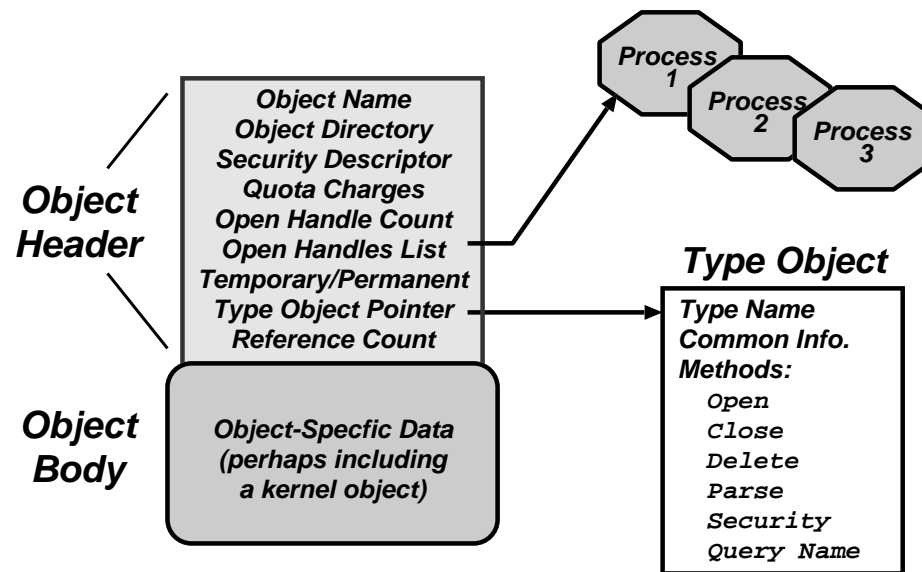
- Every resource in NT is represented by an object

- The Object Manager (part of the Executive) is responsible for:

  - creating objects and object handles
  - performing security checks
  - tracking which processes are using each object

- Typical operation:

  - `handle = open(objectname, accessmode)`
  - `result = service(handle, arguments)`

Object Namespace
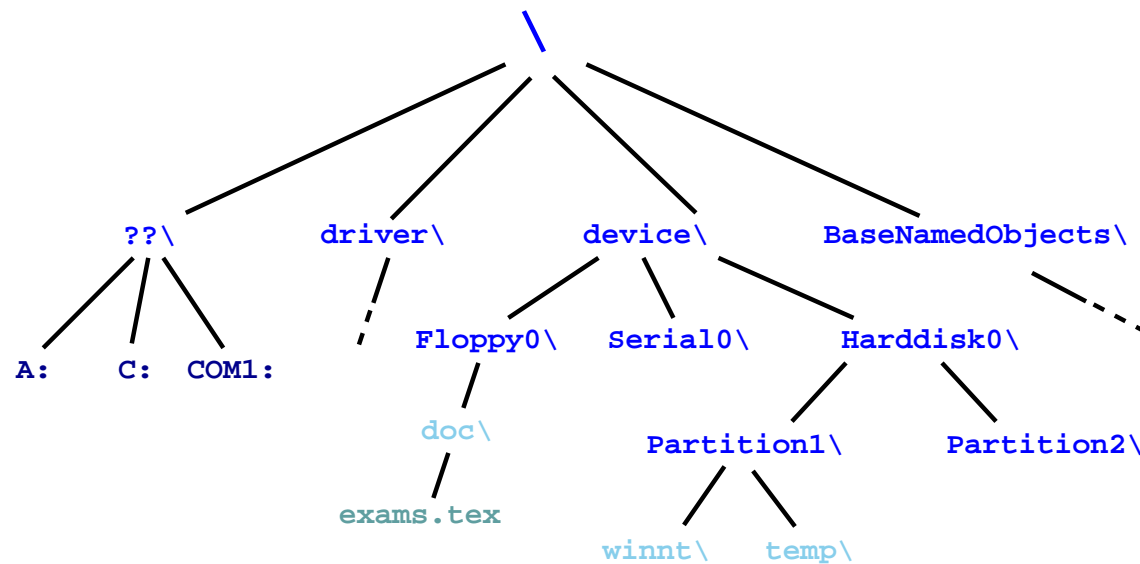
- Recall: objects (optionally) have a name

- Object Manger manages a hierarchical namespace:

    - shared between all processes ⇒ sharing
    - implemented via directory objects
    - each object protected by an access control list.
    - naming domains (using `parse`) mean file-system namespaces can be integrated

- Also get symbolic link objects: allow multiple names (aliases) for the same object.

- Modified view presented at API level. . .

Process Manager

- Provides services for creating, deleting, and using threads and processes.

- Very flexible:

  - no built in concept of parent/child relationships or process hierarchies
  - processes and threads treated orthogonally.
  $\Rightarrow$ can support Posix, OS/2 and Win32 models.

Virtual Memory Manager

- NT employs paged virtual memory management

- The VMM provides processes with services to:

  - allocate and free virtual memory
  - modify per-page protections

- Can also share portions of memory:

  - use section objects ($\approx$ software segments)
  - section objects are either based (specific base address) or non-based (floating)
  - also used for memory-mapped files

Security Reference Manager

- NT's object-oriented nature enables a uniform mechanism for runtime access and audit checks

  - everytime a process opens handle to an object, check process's security token and object's ACL
 - compare with Unix (file-system, networking, window system, shared memory)

Local Procedure Call Facility

- LPC (or IPC) passes requests and results between client and server processes within a single machine.

- Used to request services from the various NT environmental subsystems.

- Three variants of LPC channels:

  1. small messages ($\leq$ 256 bytes): copy messages between processes
  2. zero copy: avoid copying large messages by pointing to a shared memory section object created for the channel.
  3. quick LPC: used by the graphical display portions of the Win32 subsystem.

I/O Manager

I/O Requests

File System Driver

I/O Manager

Intermediate Driver

Device Driver

HAL
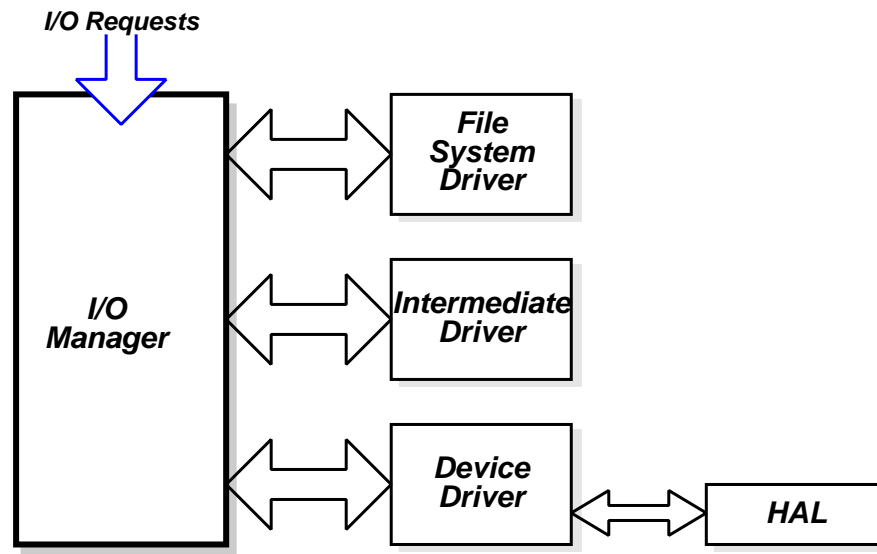
- The I/O Manager is responsible for:
  - file systems
  - cache management
  - device drivers

- Basic model is asynchronous:
  - each I/O operation explicitly split into a request and a response
  - an I/O Request Packet (IRP) used to hold parameters, results, etc.

- File-system & device drivers are stackable. . .

Cache Manager

- Cache Manager caches "virtual blocks":

  - viz. keeps track of cache "lines" as offsets within a $file$ rather than a volume.
  - disk layout & volume concept abstracted away.
  - $\Rightarrow$ no translation required for cache hit.
  - $\Rightarrow$ can get more intelligent prefetching

- Completely unified cache:

  - cache "lines" all live in the virtual address space.
  - decouples physical & virtual cache systems: e.g.
    * virtually cache in 256K blocks,
    * physically cluster up to 64K.
  - NT virtual memory manager responsible for actually doing the I/O.
  - so lots of FS cache when VM system lightly loaded, little when system thrashing

- NT also provides some user control:

  - if specify `temporary` attrib when creating file $\Rightarrow$ data will never be flushed to disk unless absolutely necessary.
  - if specify `write_through` attrib when opening a file $\Rightarrow$ all writes will synchronously complete.

File Systems: FAT16
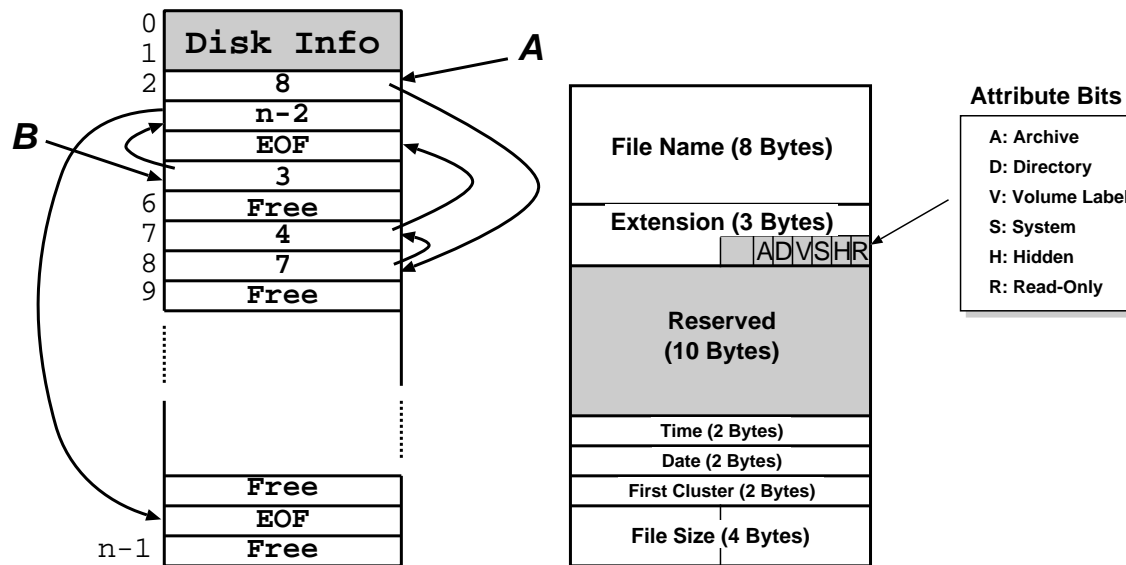
- A file is a linked list of clusters (= a set of 2^n contiguous disk blocks, $n \geq 0$)

- Each entry in the `FAT` contains either:

  - the index of another entry within the `FAT`, or
  - a special value `EOF` meaning "end of file", or
  - a special value `Free` meaning "free".

- Directory entries contain index into the `FAT`

- FAT16 could only handle partitions up to $(2^{16} \times c)$ bytes $\Rightarrow$ max 2Gb partition with 32K clusters (and big cluster size is bad)

File Systems: FAT32

- Obvious extension: instead of using 2 bytes per entry, `FAT32` uses 4 bytes

$\Rightarrow$ can support e.g. 8Gb partition with 4K clusters

- Further enhancements with `FAT32` include:

  - can locate the root directory anywhere on the partition (in `FAT16`, the root directory had to immediately follow the `FAT(s)`).
  - can use the backup copy of the `FAT` instead of the default (more fault tolerant)
  - improved support for demand paged executables (consider the 4K default cluster size . . .).

- `VFAT` on top of `FAT32` adds long name support and internationalization:

  - names now unicode strings of up to 256 characters.
  - want to keep same directory entry structure for compatibility with e.g. DOS
  $\Rightarrow$ use *multiple* directory entries to contain successive parts of name.
  - abuse `V` attribute to avoid listing these

Still pretty primitive. . .

File-Systems: NTFS
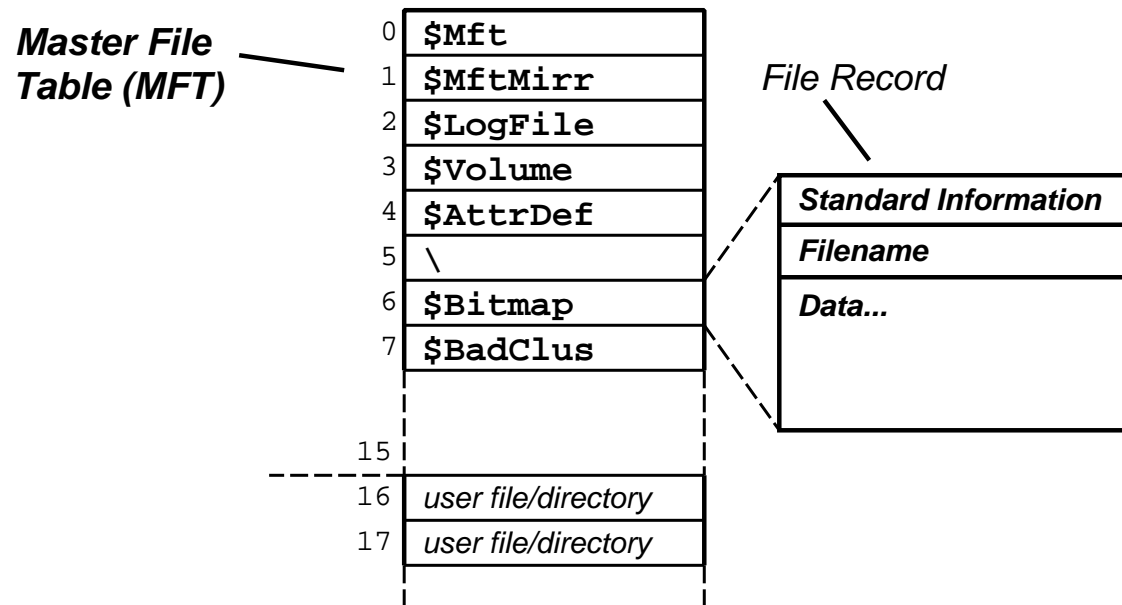
- Fundamental structure of NTFS is a volume:
  - based on a logical disk partition
  - may occupy a portion of a disk, and entire disk, or span across several disks.
- NTFS stores all file records in a special file called the Master File Table (MFT).
- The MFT is indexed by a file reference: a 64-bit unique identifier for a file
- A file itself is a structured object consisting of set of attribute/value pairs of variable length. . .

NTFS: Recovery

- To aid recovery, all file system data structure updates are performed inside transactions:

  - before a data structure is altered, the transaction writes a log record that contains redo and undo information.
  - after the data structure has been changed, a commit record is written to the log to signify that the transaction succeeded.
  - after a crash, the file system can be restored to a consistent state by processing the log records.

- Does not guarantee that all the user file data can be recovered after a crash — just that metadata files will reflect some prior consistent state.

- The log is stored in the third metadata file at the beginning of the volume (`$Logfile`)

  - in fact, NT has a generic log file service
  ⇒ could in principle be used by e.g. database

- Overall makes for far quicker recovery after crash

- (modern Unix fs [ext3, xfs] use similar scheme)

NTFS: Fault Tolerance

Hard Disk A   Hard Disk B

Partition A1
Partition A2
Partition A3

Partition B1
Partition B2
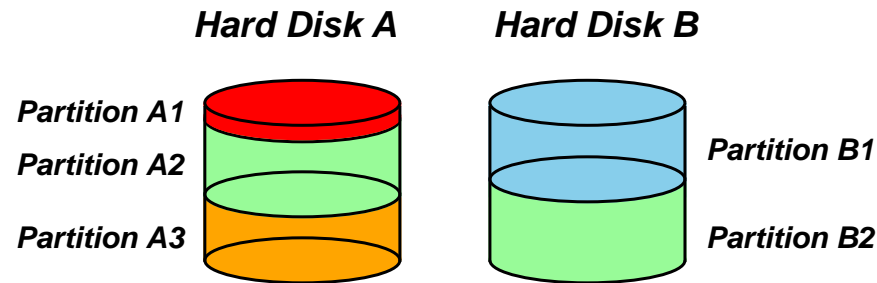
- `FtDisk` driver allows multiple partitions be combined into a logical volume:

  – e.g. logically concatenate multiple disks to form a large logical volume
  – based on the concept of RAID = **R**edundant **A**rray of **I**nexpensive **D**isks:
  – e.g. RAID level 0: interleave multiple partitions round-robin to form a stripe set:
    * logical block 0 → block 0 of partition A2, logical block 1 → block 0 of partition B2, logical block 2 → block 1 of partition A2, etc
  – e.g. RAID level 1 increases robustness by using a mirror set: two equally sized partitions on two disks with identical data contents.
  – (other more complex RAID levels also exist)

- `FtDisk` can also handle sector sparing where the underlying SCSI disk supports it

- (if not, NTFS supports cluster remapping in software)

NTFS: Other Features

- Security:

  - security derived from the NT object model.
  - each file object has a security descriptor attribute stored in its MFT record.
  - this atrribute holds the access token of file owner plus an access control list

- Compression:

  - NTFS can divide a file's data into compression units (sets of 16 contiguous clusters in the file)
  - NTFS also has support for sparse files
    * clusters with all zeros not actually allocated or stored on disk.
    * instead, gaps are left in the sequences of VCNs kept in the file record
    * when reading a file, gaps cause NTFS to zero-fill that portion of the caller's buffer.

- Encryption:

  - Use symmetric key to encrypt files; file attribute holds this key encrypted with user public key
  - Not really that useful: private key pretty easy to obtain; and administrator can bypass entire thing anyhow.

Environmental Subsystems

- User-mode processes layered over the native NT executive services to enable NT to run programs developed for other operating systems.

- NT uses the Win32 subsystem as the main operating environment

  - Win32 is used to start all processes.
  - Also provides all the keyboard, mouse and graphical display capabilities.

- MS-DOS environment is provided by a Win32 application called the virtual dos machine (VDM), a user-mode process that is paged and dispatched like any other NT thread.

  - Uses virtual 8086 mode, so not 100% compatible

- 16-Bit Windows Environment:

  - Provided by a VDM that incorporates Windows on Windows
  - Provides the Windows 3.1 kernel routines and stub routings for window manager and GDI functions.

- The POSIX subsystem is designed to run POSIX applications following the POSIX.1 standard which is based on the UNIX model.

Summary

- Main Windows NT features are:

  - layered/modular architecture:
  - generic use of objects throughout
  - multi-threaded processes
  - multiprocessor support
  - asynchronous I/O subsystem
  - NTFS filing system (vastly superior to FAT32)
  - preemptive priority-based scheduling

- Design essentially *more advanced* than Unix.

- Implementation of lower levels (HAL, kernel & executive) actually rather decent.

- But: has historically been crippled by

  - almost exclusive use of Win32 API
  - legacy device drivers (e.g. VXDs)
  - lack of demand for "advanced" features
  - "feature interaction", aka huge swathes of complex poorly implemented user-space code written by idiots

- Continues to evolve. . .

Course Review

- **Part I: Operating System Functions**

  - OS structures: required h/w support, kernel vs. μ-kernel

  - Processes: states, structures, scheduling

  - Memory: virtual addresses, sharing, protection

  - I/O subsytem: polling/interrupts, buffering.

  - Filing: directories, meta-data, file operations.

  - Protection: authentication, access control

- **Part II: Case Studies**

  - Unix: file abstraction, command 'extensibility'

  - Windows NT: layering, objects, asynch. I/O.

— empty —

Glossary and Acronyms: A–H

AGP	Advanced Graphics Port
ALU	Arithmetic/Logic Unit
API	Application Programming Interface
ARM	a 32-bit RISC microprocessor
ASCII	American Standard Code for Information Interchange
BSD	Berkeley Software Distribution (Unix variant)
BU	Branch Unit
CAM	Content Addressable Memory
COW	Copy-on-Write
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
DMA	Direct Memory Access
DOS	**1.** a primitive OS (Microsoft); **2.** Denial of Service
DRAM	Dynamic RAM
FCFS	First-Come-First-Served (see also FIFO)
FIFO	First-In-First-Out (see also FCFS)
Fork	create a new copy of a process
Frame	chunk of physical memory (also *page frame*)
HAL	Hardware Abstraction Layer

Glossary and Acronyms: I–N

I/O	Input/Output (also IO)
IA32	Intel's 32-bit processor architecture
IA64	Intel's 64-bit processor architecture
IDE	Integrated Drive Electronics (disk interface)
IPC	Inter-Process Communication
IRP	I/O Request Packet
IRQ	Interrupt ReQuest
ISA	**1.** Industry Standard Architecture (bus); **2.** Instruction Set Architecture
Interrupt	a signal from hardware to the CPU
IOCTL	a system call to control an I/O device
LPC	Local Procedure Call
MAU	Memory Access Unit
MFT	Multiple Fixed Tasks (IBM OS)
MIPS	**1.** Millions of Instructions per Second; **2.** a 32-bit RISC processor
MMU	Memory Management Unit
MFT	Multiple Fixed Tasks (IBM OS)
MVT	Multiple Variable Tasks (IBM OS)
NT	New Technology (Microsoft OS Family)
NTFS	NT File System

Glossary and Acronyms: O–SM

OS	Operating System
OS/2	a PC operating system (IBM & Microsoft)
PC	**1.** Program Counter; **2.** Personal Computer
PCB	**1.** Process Control Block; **2.** Printed Circuit Board
PCI	Peripheral Component Interface
PIC	Programmable Interrupt Controller
PTBR	Page Table Base Register
PTE	Page Table Entry
Page	fixed size chunk of virtual memory
Poll	[repeatedly] determine the status of
Posix	Portable OS Interface for Unix
RAM	Random Access Memory
ROM	Read-Only Memory
SCSI	Small Computer System Interface
SFID	System File ID
Shell	program allowing user-computer interaction
Signal	event delivered from OS to a process
SJF	Shortest Job First
SMP	Symmetric Multi-Processor

Glossary and Acronyms: SR–X

SRAM	Static RAM
SRTF	Shortest Remaining Time First
STBR	Segment Table Base Register
STLR	Segment Table Length Register
System V	a variant of Unix
TCB	**1.** Thread Control Block; **2.** Trusted Computing Base
TLB	Translation Lookaside Buffer
UCS	Universal Character Set
UFID	User File ID
UTF-8	UCS Transformation Format 8
Unix	the first kernel-based OS
VAS	Virtual Address Space
VLSI	Very Large Scale Integration
VM	**1.** Virtual Memory; **2.** Virtual Machine
VMS	Virtual Memory System (Digital OS)
VXD	Virtual Device Driver
Win32	API provided by modern Windows OSes
XP	a recent OS from Microsoft
x86	Intel familty of 32-bit CISC processors