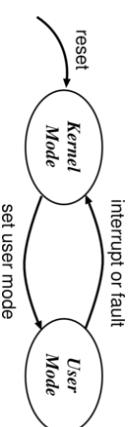


OS STRUCTURES

- Protection
- Low-level Mechanisms
- Authentication
- Access Matrix
- **OS Structures**
 - Dual-mode Operation, Kernels & Microkernels
 - Mandatory Access Control, `pLedge` (2)



Simply want to stop buggy (or malicious) program from doing bad things

- Trust boundary between user **application** and the **OS**
- Use hardware support to differentiate between (at least) two 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 OS
- Make certain instructions only possible in kernel mode, indicated by **mode bit**

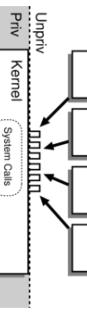
E.g., x86: Rings 0--3, ARM has two modes plus IRQ, Abort and FIQ

Often "nested" (per x86 rings); further inside can do strictly more. Not ideal – e.g., stop kernel messing with applications – but disjoint/overlapping permissions hard

6.1

KERNEL-BASED OPERATING SYSTEMS

Applications can't do I/O due to protection so the OS does it on their behalf



This means we need a secure way for application to invoke OS: a special (unprivileged) instruction to transition from user to kernel mode

Generally called a **trap** or a **software interrupt** since operates similarly to (hardware) interrupt...

OS services accessible via software interrupt mechanism called **system calls**

OS has vectors to handle traps, preventing application from leaping to kernel mode and then just doing whatever it likes

Alternative is for OS to emulate for application, and check every instruction, as used in some virtualization systems, e.g., QEMU

6.2

MICROKERNEL OPERATING SYSTEMS

We've protected "privileged instructions" via dual-mode operation, memory via special hardware, and the CPU via use of a timer. But now applications can't do much directly and must use OS to do it on their behalf

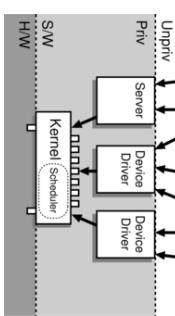
OS must be very stable to support apps, so becomes hard to extend

Alternative is **microkernels**: move OS services into (local) servers, which may be privileged

Increases both modularity and extensibility

Still access kernel via system calls, but need new ways to access servers: **Inter-Process Communication (IPC)** schemes

Given talking to servers (largely) replaces trapping, need IPC schemes to be extremely efficient



6.4

KERNELS VS MICROKERNELS

So why isn't everything a microkernel?

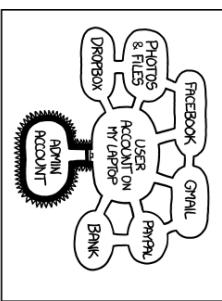
- Lots of IPC adds overhead, so microkernels (perceived as) usually performing less well
 - Microkernel implementation sometimes tricky: need to worry about synchronisation
 - Microkernels often end up with redundant copies of OS data structures
- Thus many common OSs blur the distinction between kernel and microkernel.
- E.g. Linux is "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
 - Unclear what the best OS structure is, or how much it really matters...

MANDATORY ACCESS CONTROL

From a user point of view, nowadays one often wants to protect applications from each other, all owned by a single user. Indeed, with personal single-user machines now common (phones, tablets, laptops), arguable that protection model is mismatched

Mandatory Access Control (MAC) mandates expression of policies constraining interaction of system users

E.g., OSX and iOS *Sandbox* uses subject/object labelling to implement access-control for privileges and various resources (filesystem, communication, APIs, etc)



<https://xkcd.com/1200/>

PLEDGE (2)

One way to reduce the ability of a compromised program to do Bad Things™ is to remove access to unnecessary system calls

Several attempts in different systems, with varying (limited) degrees of success:

- Hard to use correctly (e.g., Capsicum), or
- Introduce another component that needs to be watched (e.g., seccomp)

Observation:

- Most programs follow a pattern of `initialization()` then `main_loop()`, and
- The `main_loop()` typically uses a much narrower class of system calls than `initialization()`

Result? `pledge(2)` – ask the programmer to indicate explicitly which classes of system call they wish to use at any point, e.g., `stdio`, `route`, `inet`

VIRTUAL MACHINES AND CONTAINERS

More recently, trend towards encapsulating applications differently. Roughly aimed towards making applications appear as if they're the only application running on the system. Particularly relevant when building systems using **microservices**. Protection, or **isolation** at a different level

- **Virtual Machines** encapsulate an entire running system, including the OS, and then boot the VM over a hypervisor

E.g., Xen, VMWare ESX, Hyper-V

- **Containers** expose functionality in the OS so that each container acts as a separate entity even though they all share the same underlying OS functionality
- E.g., Linux Containers, FreeBSD Jails, Solaris Zones

6.5

6.6

6.7

6.8

SUMMARY

- Protection
 - Motivation, Requirements, Subjects & Objects
 - Design of Protection Systems
 - Covert Channels
- Low-level Mechanisms
 - I/O, Memory, CPU
- Authentication
 - User to System, System to User
 - Mutual Suspicion
- Access Matrix
 - Access Control Lists (ACLs) vs Capabilities
- OS Structures
 - Dual-mode Operation, Kernels & Microkernels
 - Mandatory Access Control, `pledge(2)`

7

[03] PROCESSES

OUTLINE

- Process Concept
 - Relationship to a Program
 - What is a Process?
- Process Lifecycle
 - Creation
 - Termination
 - Blocking
- Process Management
 - Process Control Blocks
 - Context Switching
 - Threads

1.1

PROCESS CONCEPTS

- Process Concept
 - Relationship to a Program
 - What is a Process?
- Process Lifecycle
- Process Management

WHAT IS A PROCESS?

The computer is there to execute programs, not the operating system!

Process \neq Program

- A program is **static**, on-disk
- A process is **dynamic**, a program *in execution*

On a batch system, might refer to *jobs* instead of processes

1.2

2.1

WHAT IS A PROCESS?

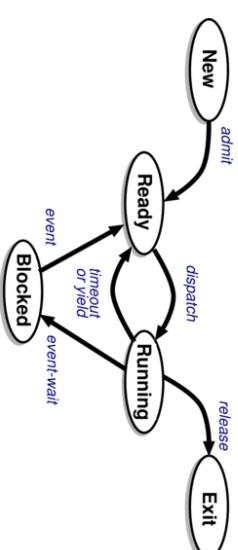
Unit of protection and resource allocation

- So you may have multiple copies of a process running
- Each process executed on a *virtual processor*

Has a virtual address space (later)

Has one or more threads, each of which has

1. **Program Counter:** which instruction is executing
2. **Stack:** temporary variables, parameters, return addresses, etc.
3. **Data Section:** global variables shared among threads



PROCESS LIFECYCLE

- Process Concept
- **Process Lifecycle**
 - Creation
 - Termination
 - Blocking
- Process Management

2.3

PROCESS CREATION

Nearly all systems are hierarchical:
parent processes create child processes

- Resource sharing:
 - Parent and children share all resources
 - Children share subset of parent's resources
 - Parent and child share no resources

2.4

PROCESS STATES

3.1

3.2

PROCESS CREATION

*Nearly all systems are hierarchical:
parent processes create child processes*

- Resource sharing
- Execution:
 - Parent and children execute concurrently
 - Parent waits until children terminate

3.3

EXAMPLES

Unix:

- `fork()` system call creates a child process, cloned from parent; then
- `execve()` system call used to replace the process' memory space with a new program

NT/2K/XP:

- `CreateProcess()` system call includes name of program to be executed

PROCESS TERMINATION

*Nearly all systems are hierarchical:
parent processes create child processes*

- Resource sharing
- Execution
- Address space:
 - Child duplicate of parent
 - Child has a program loaded into it

3.4

PROCESS TERMINATION

Occurs under three circumstances

1. 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

3.5

3.6

PROCESS TERMINATION

Occurs under three circumstances

1. Process executes last statement and asks the operating system to delete it
2. 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
3. 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")

3.7

EXAMPLES

Unix

- `wait()`, `exit()` and `kill()`

NT/2K/XP

- `ExitProcess()` for self
- `TerminateProcess()` for others.

3.8

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 {}?
- Complicated! Next year... ignore for now ;)

3.9

PROCESS TERMINATION

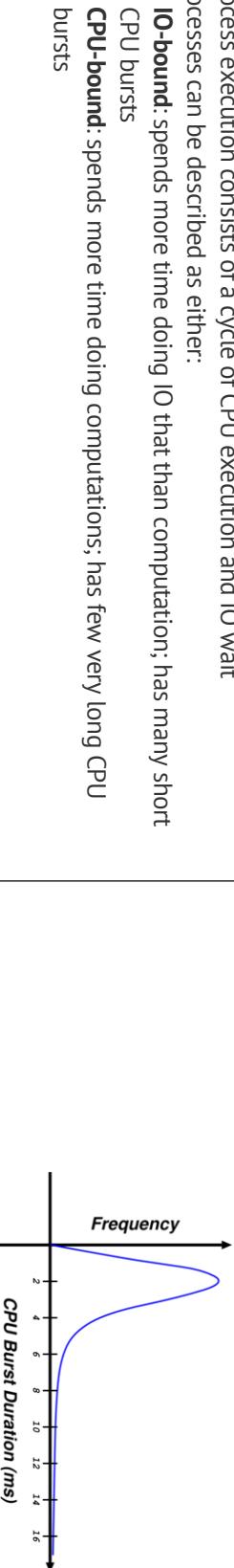
Occurs under three circumstances

1. Process executes last statement and asks the operating system to delete it
2. Process performs an illegal operation
3. 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")

3.10

CPU IO BURST CYCLE

- Process execution consists of a cycle of CPU execution and IO wait
- Processes can be described as either:
 1. **IO-bound**: spends more time doing IO than computation; has many short CPU bursts
 2. **CPU-bound**: spends more time doing computations; has few very long CPU bursts



PROCESS MANAGEMENT

- Process Concept
- Process Lifecycle
- **Process Management**
 - Process Control Blocks
 - Context Switching
 - Threads

3.11

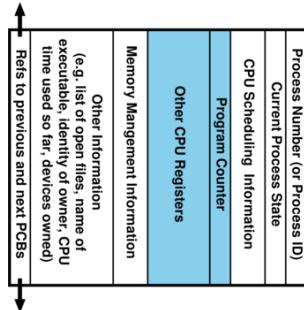
PROCESS CONTROL BLOCK

OS maintains information about every process in a data structure called a *process control block* (PCB). The *Process Context* (highlighted) is the machine environment during the time the process is actively using the CPU:

- Program counter
- General purpose registers
- Processor status register
- [Caches, TLBs, Page tables, ...]

Observe that most processes execute for at most a few milliseconds before blocking
We need multiprogramming to obtain decent overall CPU utilisation

3.12

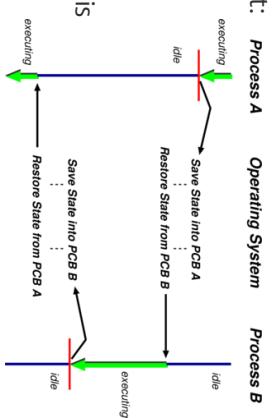


4.1

CPU IO BURST CYCLE

CONTEXT SWITCHING

- To switch between processes, the OS must:
 - Save the context of the currently executing process (if any), and restore the context of that being resumed.
- Note this is *wasted time* – no useful work is carried out while switching
- Time taken depends on hardware support
 - From nothing, to
 - Save/load multiple registers to/from memory, to
 - Complete hardware "task switch"



THREADS

A **thread** represents an individual execution context

Threads are managed by a **scheduler** that determines which thread to run

Each thread has an associated **Thread Control Block (TCB)** with metadata about the thread: saved context (registers, including stack pointer), scheduler info, etc.

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

Threads visible to the OS are **kernel threads** – may execute in kernel or address user space

SUMMARY

- Process Concept
 - Relationship to a program
 - What is a process?
- Process Lifecycle
 - Creation
 - Termination
 - Blocking
- Process Management
 - Process Control Blocks
 - Context Switching
 - Threads

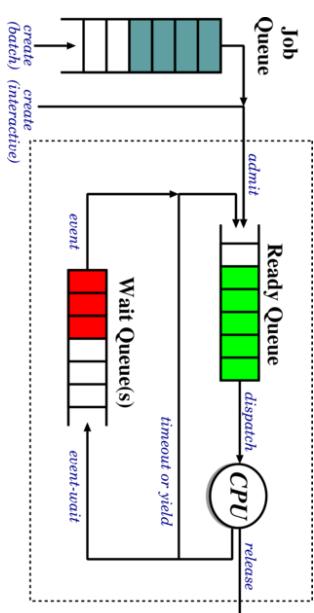
[04] SCHEDULING

OUTLINE

- Scheduling Concepts
 - Queues
 - Non-preemptive vs Preemptive
 - Idling
- Scheduling Criteria
 - Utilisation
 - Throughput
 - Turnaround, Waiting, Response Times
- Scheduling Algorithms
 - First-Come First-Served
 - Shortest Job First
 - Shortest Response Time First
 - Predicting Burst Length
 - Round Robin
 - Static vs Dynamic Priority

- 1.1
- Scheduling Concepts
 - Scheduling Criteria
 - Scheduling Algorithms
- 1.2

QUEUES



- 2.1
- Job Queue: batch processes awaiting admission
 - Ready Queue: processes in main memory, ready and waiting to execute
 - Wait Queue(s): set of processes waiting for an IO device (or for other processes)
 - Job scheduler selects processes to put onto the ready queue
 - CPU scheduler selects process to execute next and allocates CPU
- 2.2

PREEMPTIVE VS NON-PREEMPTIVE

OS needs to select a ready process and allocate it the CPU
When?

- ...a running process blocks (running → blocked)
- ...a process terminates (running → exit)
- ...a waiting process unblocks (blocked → ready)

If scheduling decision is only taken under these conditions, the scheduler is said to be **non-preemptive**

- ...a timer expires (running → ready)
- ...a waiting process unblocks (blocked → ready)

Otherwise it is **preemptive**

NON-PREEMPTIVE

- Simple to implement:
 - No timers, process gets the CPU for as long as desired
- Open to *denial-of-service*:
 - Malicious or buggy process can refuse to yield

Typically includes an *explicit* yield system call or similar, plus *implicit* yields, e.g., performing IO, waiting

Examples: MS-DOS, Windows 3.11

PREEMPTIVE

- Solves denial-of-service:
 - OS can simply preempt long-running process
- More complex to implement:
 - Timer management, concurrency issues

Examples: just about everything you can think of :)

IDLING

We will usually assume that there's always something ready to run. But what if there isn't?

This is quite an important question on modern machines where the common case is >50% idle

2.3

2.4

2.5

2.6

IDLING

Three options

1. Busy wait in scheduler, e.g., Windows 9x
 - Quick response time
 - Ugly, useless

1. Busy wait in scheduler
2. Halt processor until interrupt arrives, e.g., modern OSs
 - Saves power (and reduces heat!)
 - Increases processor lifetime
 - Might take too long to stop and start

2.7

IDLING

Three options

1. Busy wait in scheduler
2. Halt processor until interrupt arrives
3. Invent an idle process, always available to run
 - Gives uniform structure
 - Could run housekeeping
 - Uses some memory
 - Might slow interrupt response

In general there is a trade-off between responsiveness and usefulness. Consider the important resources and the system complexity

SCHEDULING CRITERIA

- Scheduling Concepts
- **Scheduling Criteria**
 - Utilisation
 - Throughput
 - Turnaround, Waiting, Response Times
- Scheduling Algorithms

2.8

2.9

Three options

3.1

SCHEDULING CRITERIA

Typically one expects to have more than one option – more than one process is runnable

On what basis should the CPU scheduler make its decision?

Many different metrics may be used, exhibiting different trade-offs and leading to different operating regimes

CPU UTILISATION

Maximise the fraction of the time the CPU is actively being used

Keep the (expensive?) machine as busy as possible

But may penalise processes that do a lot of IO as they appear to result in the CPU not being used

3.2

THROUGHPUT

Maximise the number of processes that complete their execution per time unit

Ensures every processes complete in shortest time possible

TURNAROUND TIME

Minimise the amount of time to execute a particular process

Get useful work completed at the highest rate possible

But may penalise long-running processes as short-run processes will complete sooner and so are preferred

3.3

WAITING TIME

Minimise the amount of time a process has been waiting in the ready queue

Ensures an interactive system remains as responsive as possible

But may penalise IO heavy processes that spend a long time in the wait queue

3.4

RESPONSE TIME

Minimise the amount of time it takes from when a request was submitted until the first response is produced

- Found in time-sharing systems. Ensures system remains as responsive to clients as possible under load
- But may penalise longer running sessions under heavy load

3.6

FIRST-COME FIRST-SERVED (FCFS)

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

E.g. given the following demand:

Process	Burst Time
P ₁	25
P ₂	4
P ₃	7

EXAMPLE: FCFS

Consider the average waiting time under different arrival orders

P₁, P₂, P₃:

- Waiting time P₁ = 0, P₂ = 25, P₃ = 29
- Average waiting time: $\frac{(0+25+29)}{3} = 18$

P₃, P₂, P₁:

- Waiting time P₁ = 11, P₂ = 7, P₃ = 0
- Average waiting time: $\frac{(11+7+0)}{3} = 6$

Arriving in reverse order is *three times as good!*

- The first case is poor due to the **convoy effect**: later processes are held up behind a long-running first process
- FCFS is simple but not terribly robust to different arrival processes

4.1

SCHEDULING ALGORITHMS

- Scheduling Concepts
- Scheduling Criteria
 - Scheduling Algorithms**
 - First-Come First-Served
 - Shortest Job First
 - Shortest Response Time First
 - Predicting Burst Length
 - Round Robin
 - Static vs Dynamic Priority

4.2

SHORTEST JOB FIRST (SJF)

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
- Use, e.g., FCFS to break ties

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

Waiting time for $P_1 = 0, P_2 = 6, P_3 = 3, P_4 = 7$. Average waiting time:
 $\frac{(0+6+3+7)}{4} = 4$

SJF is optimal with respect to average waiting time:

- It minimises average waiting time for a given set of processes
- What might go wrong?

4.4

SHORTEST REMAINING-TIME FIRST (SRTF)

Simply a preemptive version of SJF: preempt the running process if a new process arrives with a CPU burst length less than the remaining time of the current executing process

EXAMPLE: SRTF

As before:

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

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

Average waiting time: $\frac{(9+1+0+2)}{4} = 3$

4.5

EXAMPLE: SJF

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

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
- Use, e.g., FCFS to break ties

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

SJF is optimal with respect to average waiting time:

- It minimises average waiting time for a given set of processes
- What might go wrong?

4.6

SHORTEST JOB FIRST (SJF)

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
- Use, e.g., FCFS to break ties

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

SJF is optimal with respect to average waiting time:

- It minimises average waiting time for a given set of processes
- What might go wrong?

4.7

EXAMPLE: SRTF

Surely this is optimal in the face of new runnable processes arriving? Not necessarily – why?

- Context switches are not free: many very short burst length processes may thrash the CPU, preventing useful work being done
- More fundamentally, we can't generally know what the **future** burst length is!

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

4.8

PREDICTING BURST LENGTHS

- If we expand the formula we get:

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

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 exponential averaging scheme is a good predictor if the variance is small
- Since both α and $(1 - \alpha)$ are less than or equal to one, each successive term has less weight than its predecessor
- NB. Need some consideration of load, else get (counter-intuitively) increased priorities when increased load

4.9

PREDICTING BURST LENGTHS

- For both SJF and SRTF require the next "burst length" for each process means we must estimate it
- Can be done by using the length of previous CPU bursts, using exponential averaging:
 1. t_n = actual length of n^{th} CPU burst.
 2. τ_{n+1} = predicted value for next CPU burst.
 3. For α , $0 \leq \alpha \leq 1$ define:

$$\begin{aligned}\tau_{n+1} &= \alpha t_n + (1 - \alpha)\tau_n \\ &\quad \vdots \\ &= \alpha^j t_n + (1 - \alpha)^j \tau_0\end{aligned}$$

ROUND ROBIN

A preemptive scheduling scheme for time-sharing systems.

- Define a small fixed unit of time called a quantum (or time-slice), typically 10 – 100 milliseconds
- Process at the front of the ready queue is allocated the CPU for (up to) one quantum
- When the time has elapsed, the process is preempted and appended to the ready queue

4.11

ROUND ROBIN: PROPERTIES

Round robin has some nice properties:

- Fair: given n processes in the ready queue and time quantum q , 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 to choose the correct size quantum, q :

- q too large becomes FCFS/FIFO
- q too small becomes context switch overhead too high

PRIORITY SCHEDULING

Associate an (integer) priority with each process, e.g.,

Prio	Process type
0	system internal processes
1	interactive processes (staff)
2	interactive processes (students)
3	batch processes

Simplest form might be just system vs user tasks

TIE-BREAKING

What do with ties?

- Round robin with time-slicing, allocating quantum to each process in turn
 - Get preemptive and non-preemptive variants
- E.g., SJF is a priority scheduling algorithm where priority is the predicted next CPU burst time
 - Solution?
 - Per-process quantum based on usage?
 - Just ignore the problem?

4.12

4.13

4.14

4.15

STARVATION

Urban legend about IBM 7074 at MIT: when shut down in 1973, low-priority processes were found which had been submitted in 1967 and had not yet been run...

This is the biggest problem with static priority systems: a low priority process is not guaranteed to run – ever!

DYNAMIC PRIORITY SCHEDULING

Prevent the starvation problem: use same scheduling algorithm, but allow priorities to change over time

- 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

4.16

EXAMPLE: COMPUTED PRIORITY

First used in Dijkstra's THE

- Timeslots: $\dots, t, t+1, \dots$
- In each time slot t , measure the CPU usage of process j : u^j_t
- Priority for process j in slot $t+1$:
$$p_{t+1}^j = f(u_t^j, p_t^j, u_{t-1}^j, p_{t-1}^j, \dots)$$
- E.g., $p_{t+1}^j = \frac{p_t^j}{2} + k u_t^j$
- Penalises CPU bound but supports IO bound

Once considered impractical but now such computation considered acceptable

4.17

SUMMARY

- Scheduling Concepts
 - Queues
 - Non-preemptive vs Preemptive
 - Idling
- Scheduling Criteria
 - Utilisation
 - Throughput
 - Turnaround, Waiting, Response Times
- Scheduling Algorithms
 - First-Come First-Served
 - Shortest Job First
 - Shortest Response Time First
 - Predicting Burst Length
 - Round Robin
 - Static vs Dynamic Priority

4.18

[05] VIRTUAL ADDRESSING

OUTLINE

- Memory Management
 - Concepts
 - Relocation, Allocation, Protection, Sharing, Logical vs Physical Organisation
- The Address Binding Problem
 - Relocation
 - Logical vs Physical Addresses
- Allocation
 - Scheduling
 - Fragmentation
 - Compaction

MEMORY MANAGEMENT

- Memory Management
 - Concepts
 - Relocation, Allocation, Protection, Sharing, Logical vs Physical Organisation
- The Address Binding Problem
 - Relocation
 - Logical vs Physical Addresses
- Allocation
 - Scheduling
 - Fragmentation
 - Compaction

1.1

1.2

CONCEPTS

In a multiprogramming system, have many processes in memory simultaneously

- Every process needs memory for:
 - Instructions ("code" or "text")
 - Static data (in program)
 - Dynamic data (heap and stack)
- In addition, operating system itself needs memory for instructions and data
 - Must share memory between OS and k processes

2.1

2.2

1. RELOCATION

- Memory typically shared among processes, so programmer cannot know address that process will occupy at runtime
- May want to swap processes into and out of memory to maximise CPU utilisation
- Silly to require a swapped-in process to always go to the same place in memory
- Processes incorporate addressing info (branches, pointers, etc.)
- OS must manage these references to make sure they are sane
- Thus need to translate logical to physical addresses

2. ALLOCATION

- This is similar to sharing below, but also related to relocation
- I.e. OS may need to choose addresses where things are placed to make linking or relocation easier

3. PROTECTION

- Protect one process from others
- May also want sophisticated RWX protection on small memory units
- A process should not modify its own code (yuck)
- Dynamically computed addresses (array subscripts) should be checked for sanity

4. SHARING

- Multiple processes executing same binary: keep only one copy
- Shipping data around between processes by passing shared data segment references
- Operating on same data means sharing locks with other processes

5. LOGICAL ORGANISATION

- Most physical memory systems are linear address spaces from 0 to max
- Doesn't correspond with modular structure of programs: want segments
- Modules can contain (modifiable) data, or just code
- Useful if OS can deal with modules: can be written, compiled independently
- Can give different modules diff protection, and can be shared thus easy for user to specify sharing model

THE ADDRESS BINDING PROBLEM

- Memory Management
- The Address Binding Problem
 - Relocation
 - Logical vs Physical Addresses
- Allocation

6. PHYSICAL ORGANISATION

- Main memory: single linear address space, volatile, more expensive
- Secondary storage: cheap, non-volatile, can be arbitrarily structured
- One key OS function is to organise flow between main memory and the secondary store (cache?)
- Programmer may not know beforehand how much space will be available

2.3

2.4

3.1

THE ADDRESS BINDING PROBLEM

Consider the following simple program:

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

We can imagine that 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 ; add 3 to it  
str r2, [ry] ; and store result in y
```

where the expression `[addr]` means *the contents of the memory at address addr*.

Then the address binding problem is: what values do we give Rx and Ry?

Arises 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

3.2

LOGICAL VS PHYSICAL ADDRESSES

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



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

3.3

ADDRESS BINDING AND RELOCATION

Solution requires translation between *program* addresses and *real* addresses which can be done:

- At **compile time**:
 - Requires knowledge of absolute addresses, e.g. DOS .com files
- At **load time**:
 - Find position in memory after loading, update code with correct addresses
 - Must be done every time program is loaded
 - Ok for embedded systems / boot-loaders
- At **run-time**:
 - Get hardware to automatically translate between program and real addresses
 - No changes at all required to program itself
 - The most popular and flexible scheme, providing we have the requisite hardware (MMU)

3.3

ALLOCATION

- Memory Management
- The Address Binding Problem

- **Allocation**

- Scheduling
- Fragmentation
- Compaction

4.1

CONTIGUOUS ALLOCATION

How do we support multiple virtual processors 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:
 - Bottom partition contains OS, remainder 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!

4.2

DYNAMIC PARTITIONING

More flexibility if allow partition sizes to be dynamically chosen (e.g. OS/360 MVT):

- OS keeps track of which areas of memory are available and which are occupied
 - e.g. use one or more linked lists:
- For a new process, OS searches for a hole large enough to fit it:
 - **First fit:** stop searching list as soon as big enough hole is found
 - **Best fit:** search entire list to find "best" fitting hole
 - **Worst fit:** counterintuitively allocate largest hole (again, search entire list)

1. first and best fit perform better statistically both in time and space utilisation – typically for N allocated blocks have another $0.5N$ in wasted space using first fit
2. Which is better depends on pattern of process swapping
3. Can use buddy system to make allocation faster
4. When process terminates its memory returns onto the free list, coalescing holes where appropriate

4.3

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 MVT, ICL System 4
- Problems: fragmentation, cannot grow partitions

SCHEDULING EXAMPLE

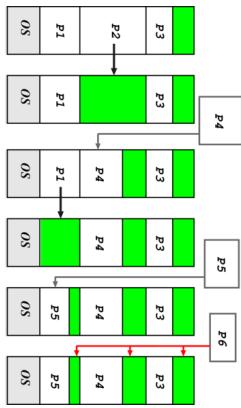
Consider a machine with total of 2560kB memory, and an OS requiring 400kB

- The following jobs are in the queue:

Process	Memory	Time
P_1	600kB	10
P_2	1000kB	5
P_3	300kB	20
P_4	700kB	8
P_5	500kB	15

4.4

EXTERNAL FRAGMENTATION



Dynamic partitioning algorithms suffer from external fragmentation: as processes are loaded they leave little fragments which may not be used. Can eventually block due to insufficient memory to swap in

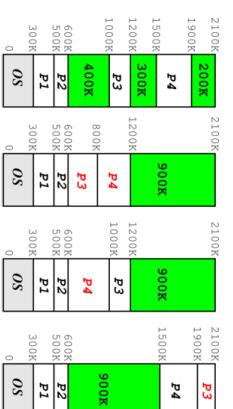
External fragmentation exists when the total available memory is sufficient for a request, but is unusable because it is split into many holes

Can also have problems with tiny holes when keeping track of hole costs more memory than hole! Requires periodic compaction

- Choosing optimal strategy quite tricky. Note that:
 - Require run-time relocation
 - 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 viable

4.6

COMPACTON



SUMMARY

- Memory Management
 - Concepts
 - Relocation, Allocation, Protection, Sharing, Logical vs Physical Organisation
 - The Address Binding Problem
 - Relocation
 - Logical vs Physical Addresses
 - Allocation
 - Scheduling
 - Fragmentation
 - Compaction

4.7

[06] PAGING

OUTLINE

- Paged Virtual Memory
 - Concepts
 - Pros and Cons
 - Page Tables
 - Translation Lookaside Buffer (TLB)
 - Protection & Sharing
- Virtual Memory
 - Demand Paging Details
 - Page Replacement
 - Page Replacement Algorithms
- Performance
 - Frame Allocation
 - Thrashing & Working Set
 - Pre-paging
 - Page Sizes

1.1

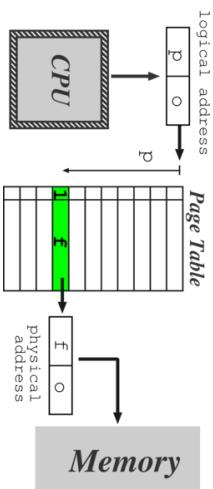
PAGED VIRTUAL MEMORY

- Paged Virtual Memory
- Concepts
- Pros and Cons
- Page Tables
- Translation Lookaside Buffer (TLB)
- Protection & Sharing
- Virtual Memory
- Performance

1.2

PAGED VIRTUAL MEMORY

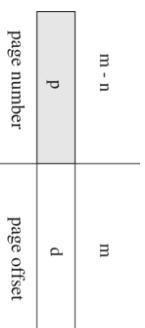
Another solution is to allow a process to exist in non-contiguous memory, i.e.,



- Divide **physical** memory into **frames**, small fixed-size blocks
- Divide **logical** memory into **pages**, blocks of the same size (typically 4kB)
- Each CPU-generated address is a page number p with page offset o
- Page table contains associated frame number f
- Usually have $|p| \gg |f|$ so also record whether mapping valid

2.1

PAGING PROS AND CONS



- Hardware support required – frequently defines the page size, typically a power of 2 (making address fiddling easy) ranging from 512B to 8192B (0.5kB – 8kB)
- Logical address space of 2^m and page size 2^n gives $p = m - n$ bits and $d = m$ bits
- Note that paging is itself a form of dynamic relocation: simply change page table to reflect movement of page in memory. This is similar to using a set of *base + limit* registers for each page in memory

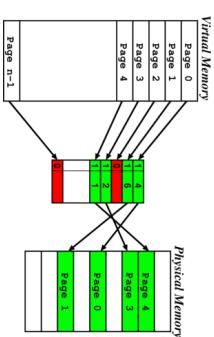
2..3

PAGING PROS AND CONS

- Clear separation between user (process) and system (OS) view of memory usage
 - Process sees single logical address space; OS does the hard work
 - Process cannot address memory they don't own – cannot reference a page it doesn't have access to
 - OS can map system resources into user address space, e.g., IO buffer
 - OS must keep track of free memory, typically in **frame table**
- Adds overhead to context switching
 - Per process page table must be mapped into hardware on context switch
 - The page table itself may be large and extend into physical memory

2..4

PAGE TABLES



PAGE TABLES

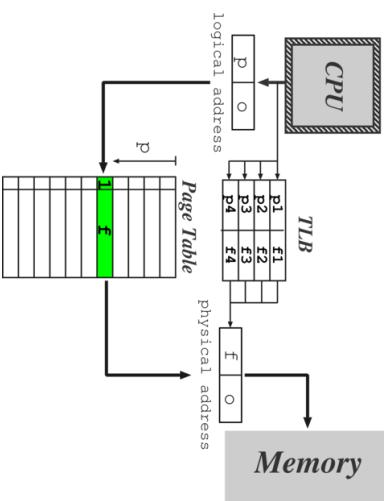
- Page Tables** (PTs) rely on hardware support:
- Simplest case: set of dedicated relocation registers
 - One register per page, OS loads registers on context switch
 - E.g., PDP-11 16 bit address, 8kB pages thus 8 PT registers
 - Each memory reference goes through these so they must be fast
 - Ok for small PTs but what if we have many pages (typically $O(10^6)$)
 - Solution: Keep PT in memory, then just one MMU register needed, the **Page Table Base Register** (PTBR)
 - OS switches this when switching process
 - Problem: PTs might still be very big
 - Keep a **PT Length Register** (PTLR) to indicate size of PT
 - Or use a more complex structure (see later)
 - Problem: need to refer to memory twice for every "actual" memory reference
 - Solution: use a **Translation Lookaside Buffer** (TLB)

2..5

TLB OPERATION

When memory is referenced, present TLB with logical memory address

- If PTE is present, get an immediate result
- Otherwise make memory reference to PTs, and update the TLB
- Latter case is typically 10% slower than direct memory reference

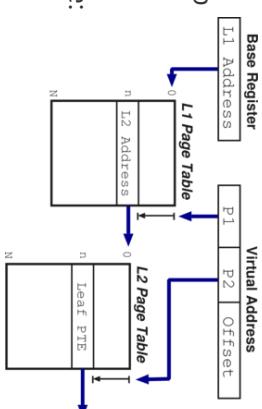


2.7

MULTILEVEL PAGE TABLES

Most modern systems can support very large (2^{32} , 2^{64}) address spaces, leading to very large PTs which we don't really want to keep all of in main memory

Solution is to split the PT into several sub-parts, e.g., two, and then page the page table:



2.8

EXAMPLE: VAX

A 32 bit architecture with 512 byte pages:

- Logical address space divided into 4 sections of 2^{30} bytes
 - Top 2 address bits designate **section**
 - Next 21 bits designate **page** within section
 - Final 9 bits designate **page offset**
 - For a VAX with 100 pages, one level PT would be 4MB; with sectioning, it's 1MB
- For 64 bit architectures, two-level paging is not enough: add further levels.
- For 4kB pages need 2^{52} entries in PT using 1 level PT
 - For 2 level PT with 32 bit outer PT, we'd still need 16GB for the outer PT
- Even some 32 bit machines have > 2 levels: SPARC (32 bit) has 3 level paging scheme; 68030 has 4 level paging

2.9

TLB ISSUES

As with any cache, what to do when it's full, how are entries shared?

- If full, discard entries typically Least Recently Used (LRU) policy
- Context switch requires TLB flush to prevent next process using wrong PTEs – Mitigate cost through **process tags** (how?)

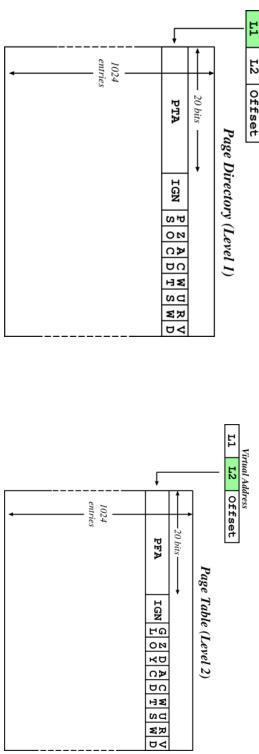
Performance is measured in terms of **hit ratio**, proportion of time a PTE is found in TLB. Example:

- Assume TLB search time of 20ns, memory access time of 100ns, hit ratio of 80%
- Assume one memory reference required for lookup, what is the **effective memory access time**?
 - $0.8 \times 120 + 0.2 \times 220 = 140$ ns
 - Now increase hit ratio to 98% – what is the new effective access time?
 - $0.98 \times 120 + 0.02 \times 220 = 122$ ns – just a 13% improvement
 - (Intel 80486 had 32 registers and claimed a 98% hit ratio)

2.10

- Divide the page number into two parts
- e.g., 20 bit **page number**, 12 bit **page offset**
- Then divide the page number into **outer** and **inner** parts of 10 bits each

EXAMPLE: X86



Page size is 4kB or 4MB. First lookup to the **page directory**, indexed using top 10 bits. The page directory address is stored in an internal processor register (cr3). The lookup results (usually) in the address of a page table. Next 10 bits index the **page table**, retrieving the **page frame address**. Finally, add in the low 12 bits as the **page offset**. Note that the page directory and page tables are exactly one page each themselves (not by accident)

2.11

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 between users
- If code is re-entrant (i.e. stateless, non-self modifying) it can be easily shared
- 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., many hundreds of MB shared code on this laptop. (Though nowadays, see unikernels!)

As the address goes through the page hardware, can check protection bits – though note this only gives *page granularity* protection, not byte granularity

Any attempt to violate protection causes hardware trap to operating system code to handle. The entry in the TLB will have a valid/invalid bit indicating whether the page is mapped into the process address space. If invalid, trap to the OS handler to map the page

Can do lots of interesting things here, particularly with regard to sharing, virtualization, ...

2.12

VIRTUAL MEMORY

- Paged Virtual Memory
- Virtual Memory
 - Demand Paging Details
 - Page Replacement
 - Page Replacement Algorithms
- Performance

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 (RWX). Might also distinguish whether may only be accessed when executing in kernel mode, e.g.,

Frame Number	K	R	W	X	V
0000	1	0	1	0	0

VIRTUAL MEMORY

Virtual addressing allows us to introduce the idea of **virtual memory**

- Already have valid or invalid page translations; introduce "non-resident designation and put such pages on a non-volatile backing store
- Processes access non-resident memory just as if it were "the real thing"

Virtual Memory (VM) has several benefits:

- **Portability**: programs work regardless of how much actual memory present; programs can be larger than physical memory
- **Convenience**: programmer can use e.g. large sparse data structures with impunity; less of the program needs to be in memory at once, thus potentially more efficient multi-programming, less IO loading/swapping program into memory
- **Efficiency**: no need to waste (real) memory on code or data which isn't used (e.g., error handling)

3.2

DEMAND PAGING DETAILS

When loading a new process for execution:

- Create its address space (page tables, etc)
- Mark PTEs as either invalid or non-resident
- Add PCB to scheduler

Then whenever we receive a page fault, check PTE:

- If due to invalid reference, kill process
- Otherwise due to non-resident page, so "page in" the desired page:
 - Find a free frame in memory
 - Initiate disk I/O to read in the desired page
 - 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

3.3

DEMAND PAGING: ISSUES

Above process makes the fault invisible to the process, but:

- Requires care to save process state correctly on fault
 - Can be particularly awkward on a CPU with pipelined decode as we need to wind back (e.g., MIPS, Alpha)
 - Even worse on on CISC CPU where single instruction can move lots of data, possibly across pages – we can't restart the instruction so rely on help from microcode (e.g., to test address before writing). Can possibly use temporary registers to store moved data
 - Similar difficulties from auto-increment/auto-decrement instructions, e.g., ARM
 - Can even have instructions and data spanning pages, so multiple faults per instruction, though **locality of reference** tends to make this infrequent
- Scheme described above is pure demand paging: don't bring in pages until needed so get lots of page faults and I/O when process begins; hence many real systems explicitly load some core parts of the process first

3.5

VM IMPLEMENTATION

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 (eg., Burroughs, OS/2) as it's more difficult (segment replacement is much harder due to segments having variable size)

PAGE REPLACEMENT

To page in from disk, we need a free frame of physical memory to hold the data we're reading in – but in reality, the size of physical memory is limited so either:

- Discard unused pages if total demand for pages exceeds physical memory size
- Or swap out an entire process to free some frames

Modified algorithm: on a page fault we:

1. Locate the desired replacement page on disk
2. Select a free frame for the incoming page:
 1. If there is a free frame use it, otherwise select a victim page to free
 2. Then write the victim page back to disk
 3. Finally mark it as invalid in its process page tables
3. Read desired page into the now free frame
4. Restart the faulting process

...thus, having no frames free effectively doubles the page fault service time

3.6

PAGE REPLACEMENT ALGORITHMS

FIRST-IN FIRST-OUT (FIFO)

Keep a queue of pages, discard from head. Performance is hard to predict as we've no idea whether replaced page will be used again or not: eviction is independent of page use frequency. In general this is very simple but pretty bad:

- Can actually end up discarding a page currently in use, causing an immediate fault and next in queue to be replaced – really slows system down
- Possible to have more faults with increasing number of frames (**Belady's anomaly**)

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, but serves as a good baseline for other algorithms

3.7

LEAST RECENTLY USED (LRU)

Replace the page which has not been used for the longest amount of time.

Equivalent to OPT with time running backwards. Assumes that the past is a good predictor of the future. Can still end up replacing pages that are about to be used substantially.

Generally considered quite a good replacement algorithm, though may require substantial hardware assistance

But! How do we determine the LRU ordering?

Can reduce overhead by adding a "dirty" bit to PTEs

- Can allow us to omit step (2.2) above by only writing out page was modified, or if page was read-only (e.g., code)

How do we choose our victim page?

- A key factor in an efficient VM system: evicting a page that we'll need in a few instructions time can get us into a really bad condition
- We want to ensure that we get few page faults overall, and that any we do get are relatively quick to satisfy

We will now look at a few **page replacement algorithms**:

- All aim to minimise page fault rate
- Candidate algorithms are evaluated by (trace driven) simulation using reference strings

3.8

PAGE REPLACEMENT

IMPLEMENTING LRU: COUNTERS

- Give each PTE a time-of-use field and give the CPU a logical clock (counter)
- Whenever a page is referenced, its PTE is updated to clock value
- Replace page with smallest time value

Problems:

- Requires a search to find minimum counter value
- Adds a write to memory (PTE) on every memory reference
- Must handle clock overflow

Impractical on a standard processor

3.10

APPROXIMATING LRU

Many systems have a reference bit in the PTE, initially zeroed by OS, and then set by hardware whenever the page is touched. After time has passed, consider those pages with the bit set to be **active** and implement **Not Recently Used** (NRU) replacement:

- Periodically (e.g. 20ms) clear all reference bits
 - When choosing a victim to evict, prefer pages with clear reference bits
 - If also have a **modified** or **dirty bit** in the PTE, can use that too
- | Referenced? | Dirty? | Comment |
|-------------|--------|--------------------------------|
| no | no | best type of page to replace |
| no | yes | next best (requires writeback) |
| yes | no | probably code in use |
| yes | yes | bad choice for replacement |

3.11

IMPROVING THE APPROXIMATION

Instead of just a single bit, the OS:

- Maintains an 8-bit value per page, initialised to zero
- Periodically (e.g. 20ms) shifts reference bit onto high order bit of the byte, and clear reference bit

Then select lowest value page (or one of) to replace

- Keeps the history for the last 8 clock sweeps
- Interpreting bytes as `u_ints`, then LRU page is $\min(\text{additional_bits})$
- May not be unique, but gives a candidate set

3.12

IMPLEMENTING LRU: PAGE STACK

- Maintain a stack of pages (doubly linked list) with most-recently used (MRU) page on top
- Discard from bottom of stack

Problem:

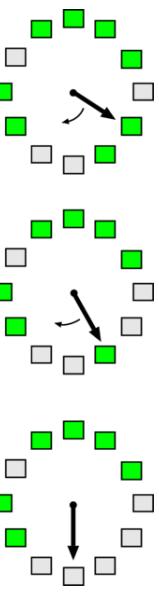
- Requires changing (up to) 6 pointers per [new] reference (max 6 pointers)
- This is very slow without extensive hardware support

Also impractical on a standard processor

3.13

FURTHER IMPROVEMENT: SECOND-CHANCE FIFO

Often implemented with a circular queue and a current pointer; in this case usually called **clock**



- Store pages in queue as per FIFO
- Before discarding head, check reference bit
- If reference bit is 0, discard else reset reference bit, and give page a second chance (add it to tail of queue)

Guaranteed to terminate after at most one cycle, with the worst case having the second chance devolve into a FIFO if all pages are referenced. A page given a second chance is the last to be replaced

3-14

OTHER REPLACEMENT SCHEMES

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

- **Least Frequently Used** (LFU): replace page with smallest count
 - Takes no time information into account
 - Page can stick in memory from initialisation
 - Need to periodically decrement counts
- **Most Frequently Used** (MFU): replace highest count page
 - Low count indicates recently brought in

3-15

PAGE BUFFERING ALGORITHMS

- Keep a minimum number of victims in a free pool
 - New page read in before writing out victim, allowing quicker restart of process
 - Alternative: if disk idle, write modified pages out and reset dirty bit
 - Improves chance of replacing without having to write dirty page
- (Pseudo) MRU: Consider accessing e.g. large array.
- The page to replace is one application has just finished with, i.e. most recently used
 - Track page faults and look for sequences
 - Discard the k th in victim sequence

Application-specific: stop trying to second-guess what's going on and provide hook for application to suggest replacement, but must be careful with denial of service

3-16

IMPLEMENTING SECOND-CHANCE FIFO

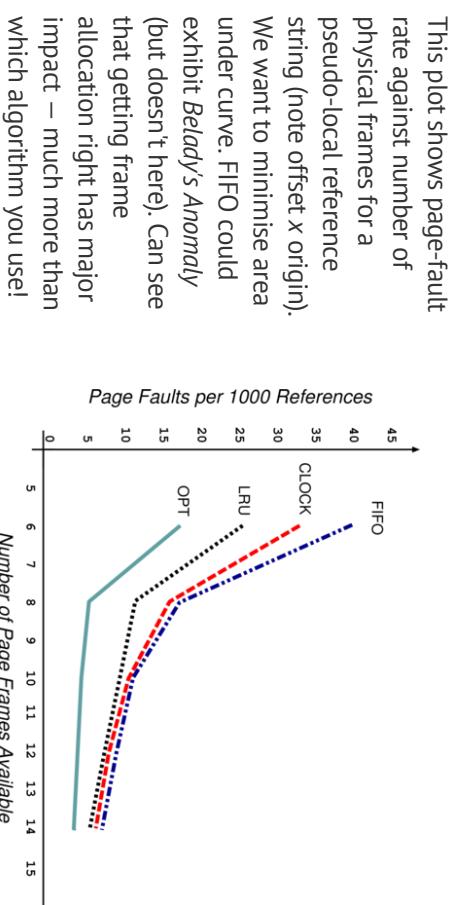
Often implemented with a circular queue and a current pointer; in this case usually called **clock**

- To clear "reference bit", mark page no access
- If referenced then trap, update PTE, and resume
- To check if referenced, check permissions
- Can use similar scheme to emulate modified bit

3-17

PERFORMANCE

- Paged Virtual Memory
- Virtual Memory
- Performance
 - Frame Allocation
 - Thrashing & Working Set
 - Pre-paging
 - Page Sizes



FRAME ALLOCATION

A certain fraction of physical memory is reserved per-process and for core OS 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 , remainder in 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

Could also allocate taking process priorities into account, since high-priority processes are supposed to run more readily. Could even care which frames we give to which process ("page colouring")

4.1

FRAME ALLOCATION: GLOBAL SCHEMES

Most page replacement schemes are global: all pages considered for replacement

- Allocation policy implicitly enforced during page-in
 - Allocation succeeds iff policy agrees
 - Free frames often in use so steal them!

For example, on a system with 64 frames and 5 processes:

- If using fair share, each processes will have 12 frames, with four left over (maybe)
- When a process dies, when the next faults it will succeed in allocating a frame
- Eventually all will be allocated
- If a new process arrives, need to steal some pages back from the existing allocations

4.2

PERFORMANCE COMPARISON

FRAME ALLOCATION: COMPARISON TO LOCAL

Also get **local page replacement schemes**: victim always chosen from within process

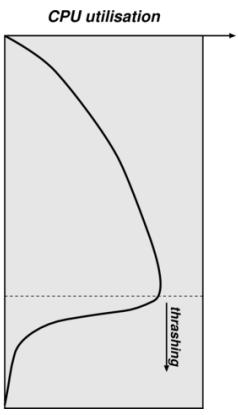
In global schemes the process cannot control its own page fault rate, so performance may depend entirely on what other processes page in/out

In local schemes, performance depends only on process behaviour, but this can hinder progress by not making available less/unused pages of memory

Global are optimal for throughput and are the most common

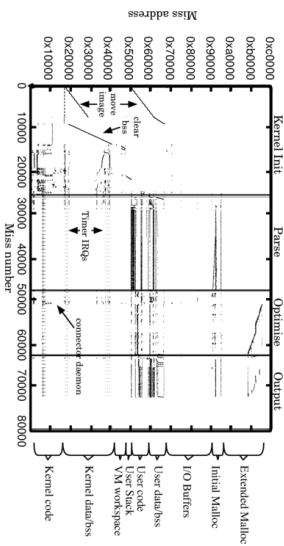
A process can technically run with minimum-free frames available but will have a very high page fault rate. If we're very unlucky, OS monitors CPU utilisation and increases level of multiprogramming if utilisation is too low: machine dies

Avoid thrashing by giving processes as many frames as they "need" and, if we can't, we must reduce the MPL – a better page-replacement algorithm will not help



4.5

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

THE RISK OF THRASHING

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

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

Avoid thrashing by giving processes as many frames as they "need" and, if we can't, we must reduce the MPL – a better page-replacement algorithm will not help

4.6

AVOIDING THRASHING

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

- Define the **Working Set** (WS) (Denning, 1967)

Set of pages that a process needs in store at "the same time" to make any progress

- Varies between processes and during execution
- Assume process moves through phases
 - In each phase, get (spatial) locality of reference
 - From time to time get phase shift

4.7

4.8

CALCULATING WORKING SET

Then OS can try to prevent thrashing by maintaining sufficient pages for current phase:

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

This optimises CPU util but has the need to compute WSS_i (moving window across stream). Can approximate with periodic timer and some page reference script. After some number of intervals (i.e., of bits of state) consider pages with count < 0 to be in WS. In general, a working set can be used as a scheme to determine allocation for each process

4.9

PAGE SIZES

- Trade-off the size of the PT and the degree of fragmentation as a result
 - Typical values are 512B to 16kB – but should be reduce the numbers of queries, or ensure that the window is covered
 - Larger page size means fewer page faults
 - Historical trend towards larger page sizes
 - Eg., 386: 4kB, 68030: 256B to 32kB
- So, a page of 1kB, 56ms for 2 pages of 512B but smaller page allows us to watch locality more accurately. Page faults remain costly because CPU and memory much faster than disk

4.11

PRE-PAGING

- Pure demand paging causes a large number of PF when process starts
- Can remember the WS for a process and pre-page the required frames when process is resumed (e.g. after suspension)
- When process is started can pre-page by adding its frames to free list
- Increases I/O cost: How do we select a page size (given no hardware constraints)?

4.10

SUMMARY

- Paged Virtual Memory
 - Concepts
 - Pros and Cons
 - Page Tables
 - Translation Lookaside Buffer (TLB)
 - Protection & Sharing
 - Virtual Memory
 - Demand Paging Details
 - Page Replacement
 - Page Replacement Algorithms
- Performance
 - Frame Allocation
 - Thrashing & Working Set
 - Pre-paging
 - Page Sizes

5

[07] SEGMENTATION

OUTLINE

- Segmentation
 - An Alternative to Paging
- Implementing Segments
 - Segment Table
 - Lookup Algorithm
- Protection and Sharing
 - Sharing Subtleties
 - External Fragmentation
- Segmentation vs Paging
 - Comparison
 - Combination
- Summary
- Extras
 - Dynamic Linking & Loading

SEGMENTATION

- Segmentation
 - An Alternative to Paging
- Implementing Segments
- Protection and Sharing
- Segmentation vs Paging
- Summary
- Extras

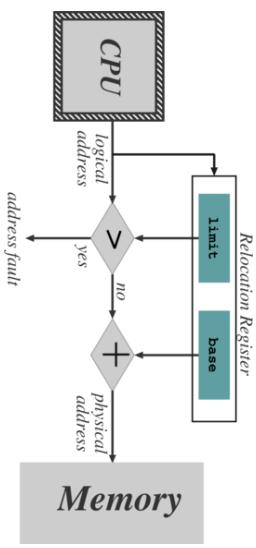
1.1

AN ALTERNATIVE TO PAGING

View memory as a set of segments of no particular size, with no particular ordering

This corresponds to typical modular approaches taken to program development

The length of a segment depends on the complexity of the function (e.g., sqrt)



1.2

2.1

2.2

WHAT IS A SEGMENT?

Segmentation supports the user-view of memory that the logical address space becomes a collection of (typically disjoint) segments

- Segments have a **name** (or a **number**) and a **length**. Addresses specify **segment**, and **offset** within segment

To access memory, user program specifies **segment + offset**, and the compiler (or, as in MULTICS, the OS) translates, in contrast to paging where the user is unaware of the memory structure and everything is managed invisibly

With paging, the user is unaware of memory structure – everything is managed invisibly

IMPLEMENTING SEGMENTS

Logical addresses are pairs, (segment, offset)

For example, the compiler might construct distinct segments for global variables, procedure call stack, code for each procedure/function, local variables for each procedure/function

Finally the loader takes each segment and maps it to a physical segment number

- If there are too many segments then the table is kept in memory, pointed to by **ST Base Register** (STBR)
- Also have an **ST Length Register** (STLR) since the number of segments used by different programs will diverge widely
- ST is part of the process context and hence is changed on each process switch
- ST logically accessed on each memory reference, so speed is critical

IMPLEMENTING SEGMENTS

- Segmentation
- **Implementing Segments**
 - **Segment Table**
 - **Lookup Algorithm**
- Protection and Sharing
- Segmentation vs Paging
- Summary
- Extras

2.3

3.1

IMPLEMENTING SEGMENTS

Segment	Access	Base	Size	Others!

Maintain a **Segment Table** for each process:

- If there are too many segments then the table is kept in memory, pointed to by **ST Base Register** (STBR)
- Also have an **ST Length Register** (STLR) since the number of segments used by different programs will diverge widely
- ST is part of the process context and hence is changed on each process switch
- ST logically accessed on each memory reference, so speed is critical

3.2

3.3

IMPLEMENTING SEGMENTS: ALGORITHM

1. Program presents address (s, d) .
 2. If $s \geq \text{STLR}$ then give up
 3. Obtain table entry at reference $s + \text{STBR}$, a tuple of form (b_s, l_s)
 4. If $0 \leq d < l_s$ then this is a valid address at location (b_s, d) , else fault
-
- The two operations b_s, d and $0 \leq d < l_s$ can be done simultaneously to save time
 - Still requires 2 memory references per lookup though, so care needed
 - E.g., Use a set of associative registers to hold most recently used ST entries
 - Similar performance gains to the TLB description earlier

PROTECTION

Segmentation's big advantage is to provide protection between components

That protection is provided *per segment*; i.e. it corresponds to the logical view

Protection bits associated with each ST entry checked in usual way, e.g. instruction segments should not be self-modifying, so are protected against writes

Could go further – e.g., place every array in its own segment so that array limits can be checked by the hardware

3.4

SHARING

Segmentation also facilitates sharing of code/data:

- Each process has its own STBR/STLR
- Sharing is enabled when two processes have entries for the same physical locations
- Sharing occurs at segment level, with each segment having own protection bits
 - For data segments can use copy-on-write as per paged case
- Can share only parts of programs, e.g., C library but there are subtleties

4.1

PROTECTION AND SHARING

- Segmentation
- Implementing Segments
- Protection and Sharing
 - Sharing Subtleties
 - External Fragmentation
- Segmentation vs Paging
- Summary
- Extras

4.2

SHARING: SUBTLETIES

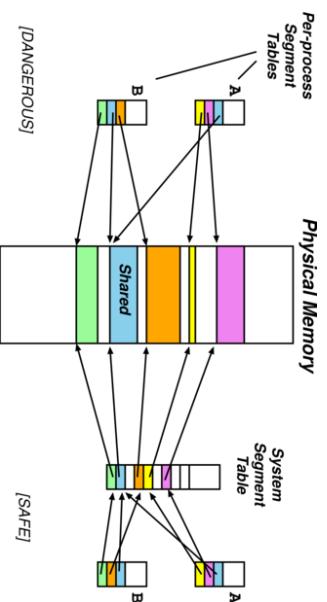
- For example, jumps within shared code
 - Jump specified as a condition + transfer address, i.e., (segment, offset)
 - Segment is (of course) this one
 - Thus all programs sharing this segment must use the same number to refer to it, else confusion will result
 - As the number of users sharing a segment grows, so does difficulty of finding a common shared segment number
 - Thus, specify branches as PC-relative or relative to a register containing the current segment number
- (Read only segments containing no pointers may be shared with different seg numbers)

EXTERNAL FRAGMENTATION RETURNS

Long term scheduler must find spots in memory for all segments of a program. Problem is that segments are variable size – thus, we must handle **fragmentation**

1. Usually resolved with best/first fit algorithm
 2. External frag may cause process to have to wait for sufficient space
 3. Compaction can be used in cases where a process would be delayed
- Tradeoff between compaction/delay depends on average segment size
- Each process has just one segment reduces to variable sized partitions
 - Each byte has its own segment separately relocated quadruples memory use!
 - Fixed size small segments is equivalent to paging!
 - Generally, with small average segment sizes, external fragmentation is small – more likely to make things fit with lots of small ones (box packing)

SHARING SEGMENTS



SEGMENTATION VS PAGING

- Segmentation
 - Implementing Segments
 - Protection and Sharing
 - Segmentation vs Paging
 - Comparison
 - Combination
- Summary
- Extras

4.5

4.4

4.6

SEGMENTATION VERSUS PAGING

- Protection, Sharing, Demand etc are all per segment or page, depending on scheme
- For **protection and sharing**, easier to have it per logical entity, i.e., per segment
- For **allocation and demand access** (and, in fact, certain types of sharing such as COW), we prefer paging because:
 - Allocation is easier
 - Cost of sharing/demand loading is minimised

logical view		allocation
segmentation	good	bad
paging	bad	good

5.2

SUMMARY

- Segmentation
- Implementing Segments
- Protection and Sharing
- Segmentation vs Paging
- **Summary**
- Extras

COMBINING SEGMENTATION AND PAGING

1. **Paged segments**, used in 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
 - Provision one page table per segment
 - Unfortunately: high hardware cost and complexity; not very portable
2. **Software segments**, used in most modern OSs
 - Consider pages $[m, \dots, m + l]$ to be a segment
 - OS must ensure protection and sharing kept consistent over region
 - Unfortunately, leads to a loss of granularity
 - However, it is relatively simple and portable

Arguably, main reason hardware segments lost is portability: you can do software segments with just paging hardware, but cannot (easily) do software paging with segmentation hardware

5.3

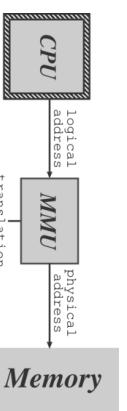
SUMMARY: VIRTUAL ADDRESSING

- Direct access to physical memory is not great as have to handle:
 - Contiguous allocation: need a large lump, end up with external fragmentation
 - Address binding: handling absolute addressing
 - Portability: how much memory does a "standard" machine have?
- Avoid problems by separating concepts of virtual (logical) and physical addresses (Atlas computer, 1962)
- Needham's comment "*every problem in computer science can be solved by an extra level of indirection*"

6.2

6.1

SUMMARY: VIRTUAL TO PHYSICAL ADDRESS MAPPING



- Runtime mapping of logical to physical addresses handled by the MMU. Make mapping per-process, then:
 - Allocation problem 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

6.3

EXTRAS

- Segmentation
- Implementing Segments
- Protection and Sharing
- Segmentation vs Paging
- Summary
- **Extras**
 - **Dynamic Linking & Loading**

6.4

DYNAMIC LINKING

Relatively new appearance in OS (early 80's). Uses *shared objects/libraries* (Unix), or *dynamically linked libraries* (DLLs; Windows). Enables a compiled binary to invoke, at runtime, routines which are dynamically linked:

- If a routine is invoked which is part of the dynamically linked code, this will be implemented as a call into a set of stubs
 - Stubs check if routine has been loaded
 - If not, linker loads routine (if necessary) and replaces stub code by routine
 - If sharing a library, the address binding problem must also be solved, requiring OS support: in the system, only the OS knows which libraries are being shared among which processes
 - Shared libs must be stateless or concurrency safe or copy on write
- Results in smaller binaries (on-disk and in-memory) and increase flexibility (fix a bug without relinking all binaries)

SUMMARY: IMPLEMENTATION CONSIDERATIONS

- **Hardware support**
 - Simple base reg enough for partitioning
 - Segmentation and paging need large tables
- **Performance**
 - Complex algorithms need more lookups per reference plus hardware support
 - Simple schemes preferred eg., simple addition to base
- **Fragmentation**: internal/external from fixed/variable size allocation units
- **Relocation**: solves external fragmentation, at high cost
 - Logical addresses must be computed dynamically, doesn't work with load time relocation
- **Swapping**: can be added to any algorithm, allowing more processes to access main memory
- **Sharing**: increases multiprogramming but requires paging or segmentation
- **Protection**: always useful, necessary to share code/data, needs a couple of bits

7.2

DYNAMIC LOADING

- At runtime a routine is loaded when first invoked
- The dynamic loader performs relocation on the fly
- It is the responsibility of the user to implement loading
- OS may provide library support to assist user

SUMMARY

- Segmentation
 - An Alternative to Paging
 - Segment Table
 - Lookup Algorithm
 - Protection and Sharing
 - Sharing Subtleties
 - External Fragmentation
- Segmentation vs Paging
 - Comparison
 - Combination
- Summary
- Extras
 - Dynamic Linking & Loading

7.3

8

[08] IO SUBSYSTEM

OUTLINE

- Input/Output (IO)
 - Hardware
 - Device Classes
 - OS Interfaces
- Performing IO
 - Pollled Mode
 - Interrupt Driven
 - Blocking vs Non-blocking
- Handling IO
 - Buffering & Strategies
 - Other Issues
 - Kernel Data Structures
 - Performance

INPUT/OUTPUT

- Input/Output (IO)
 - Hardware
 - Device Classes
 - OS Interfaces
- Performing IO
 - Pollled Mode
 - Interrupt Driven
 - Blocking vs Non-blocking
- Handling IO
 - Buffering & Strategies
 - Other Issues
 - Kernel Data Structures
 - Performance

1.1

1.2

IO HARDWARE

Very wide range of **devices** that interact with the computer via **input/output** (IO):

- Human readable: graphical displays, keyboard, mouse, printers
 - Machine readable: disks, tapes, CD, sensors
 - Communications: modems, network interfaces, radios
- All differ significantly from one another with regard to:
- **Data rate**: orders of magnitude different between keyboard and network
 - **Control complexity**: printers much simpler than disks
 - **Transfer unit and direction**: blocks vs characters vs frame stores
 - **Data representation**
 - **Error handling**

2.1

2.2

IO SUBSYSTEM

Results in IO subsystem generally being the "messiest" part of the OS

- So much variety of devices
 - So many applications
- So many dimensions of variation:
 - Character-stream or block
 - Sequential or random-access
 - Synchronous or asynchronous
 - Shareable or dedicated
 - Speed of operation
 - Read-write, read-only, or write-only

Thus, completely homogenising device API is not possible so OS generally splits devices into four classes

- Commands include read, write, seek
 - Can have raw access or via (e.g.) filesystem ("cooked") or memory-mapped
- Character devices** (e.g. keyboards, mice, serial):
- Commands include get, put
 - Layer libraries on top for line editing, etc
- Network Devices**
- Vary enough from block and character devices to get their own interface
 - Unix and Windows NT use the **Berkeley Socket** interface
- Miscellaneous**
- Current time, elapsed time, timers, clocks
 - (Unix) ioctl covers other odd aspects of IO

OS INTERFACES



Programs access **virtual devices**:
Terminal streams not terminals,
Windows not frame buffer, event
streams not raw mouse, files not disk
blocks, print spooler not parallel port,
transport protocols not raw Ethernet
frames

OS handles the processor-device interface: IO instructions vs memory mapped
devices; IO hardware type (e.g. 10s of serial chips); Polled vs interrupt driven; CPU
interrupt mechanism

Virtual devices then implemented:

- In kernel, e.g. files, terminal devices
- In daemons, e.g. spooler, windowing
- In libraries, e.g. terminal screen control, sockets

DEVICE CLASSES

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

- Commands include read, write, seek
 - Can have raw access or via (e.g.) filesystem ("cooked") or memory-mapped
- Character devices** (e.g. keyboards, mice, serial):
- Commands include get, put
 - Layer libraries on top for line editing, etc
- Network Devices**
- Vary enough from block and character devices to get their own interface
 - Unix and Windows NT use the **Berkeley Socket** interface
- Miscellaneous**
- Current time, elapsed time, timers, clocks
 - (Unix) ioctl covers other odd aspects of IO

PERFORMING IO

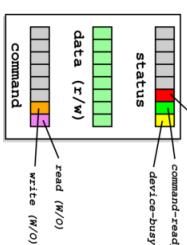
- Input/Output (IO)
- **Performing IO**
 - **Polled Mode**
 - **Interrupt Driven**
 - **Blocking vs Non-blocking**
- Handling IO

OS handles the processor-device interface: IO instructions vs memory mapped
devices; IO hardware type (e.g. 10s of serial chips); Polled vs interrupt driven; CPU
interrupt mechanism

Virtual devices then implemented:

- In kernel, e.g. files, terminal devices
- In daemons, e.g. spooler, windowing
- In libraries, e.g. terminal screen control, sockets

POLLED MODE



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:

- H repeatedly reads device-busy until clear
- H sets e.g. write bit in command register, and puts data into data register
- H sets command-ready bit in status register
- D sees command-ready and sets device-busy
- D performs write operation
- D clears command-ready & then clears device-busy

What's the problem here?

3.2

HANDLING INTERRUPTS

Split the implementation into two parts:

- At the bottom, the **interrupt handler**
- At the top, **N interrupt service routines** (ISR; per-device)

Processor-dependent interrupt handler may:

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

Device- (not processor-) dependent interrupt service routine will:

- For programmed IO device: transfer data and clear interrupt
- For DMA devices: acknowledge transfer; request any more pending; signal any waiting processes; and finally enter the scheduler or return

Question: Who is scheduling whom?

- Consider, e.g., network livelock

INTERRUPT DRIVEN

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

- At end of each instruction, processor checks interrupt line(s) for pending interrupt
 - Need not be precise (that is, occur at definite point in instruction stream)
- If line is asserted then processor:
 - Saves program counter & processor status
 - Changes processor mode
 - Jumps to a well-known address (or contents of a well-known address)

- Once interrupt-handling routine finishes, can use e.g. `rti` instruction to resume
- More complex processors may provide:
 - Multiple priority levels of interrupt
 - **Hardware vectoring** of interrupts
 - Mode dependent registers

3.3

BLOCKING VS NON-BLOCKING

From programmer's point of view, IO system calls exhibit one of three kinds of behaviour:

- **Blocking**: process suspended until IO completed
 - Easy to use and understand.
 - Insufficient for some needs.
- **Nonblocking**: IO 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 IO"
- **Asynchronous**: process runs while IO executes
 - IO subsystem explicitly signals process when its IO request has completed
 - Most flexible (and potentially efficient)
 - Also most complex to use

3.5

HANDLING IO

- Input/Output (IO)
- Performing IO
 - Handling IO
 - Buffering & Strategies
 - Other Issues
 - Kernel Data Structures
 - Performance

IO BUFFERING

To cope with various **impedance mismatches** between devices (speed, transfer size), OS may **buffer** data in memory. Various buffering strategies:

- **Single buffering**: OS assigns a system buffer to the user request
- **Double buffering**: process consumes from one buffer while system fills the next
- **Circular buffering**: most useful for bursty IO

Buffering is useful for smoothing peaks and troughs of data rate, but can't help if on average:

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

Details often dictated by device type: character devices often by line; network devices particularly bursty in time and space; block devices make lots of fixed size transfers and often the major user of IO buffer memory

4.1

SINGLE BUFFERING

OS assigns a single buffer to the user request:

- OS performs transfer, moving buffer to userspace when complete (remap or copy)
- Request new buffer for more IO, then reschedule application to consume (**readahead** or **anticipated input**)
- OS must track buffers
- Also affects swap logic: if IO is to same disk as swap device, doesn't make sense to swap process out as it will be behind the now queued IO request!

A crude performance comparison between no buffering and single buffering:

- Let t be time to input block and c be computation time between blocks
- Without buffering, execution time between blocks is $t + c$
- With single buffering, time is $\max(c, t) + m$ where m is the time to move data from buffer to user memory
- For a terminal: is the buffer a line or a char? depends on user response required

4.2

DOUBLE BUFFERING

- Often used in video rendering
- Rough performance comparison: takes $\max(c, t)$ thus
 - possible to keep device at full speed if $c < t$
 - while if $c > t$, process will not have to wait for IO
- Prevents need to suspend user process between IO operations
- ...also explains why two buffers is better than one buffer, twice as big
- Need to manage buffers and processes to ensure process doesn't start consuming from an only partially filled buffer

CIRCULAR BUFFERING

- Allows consumption from the buffer at a fixed rate, potentially lower than the **burst rate** of arriving data
- Typically use circular linked list which is equivalent to a FIFO buffer with queue length

4.4

OTHER ISSUES

- **Caching:** fast memory holding copy of data for both reads and writes; critical to IO performance
- **Scheduling:** order IO requests in per-device queues; some OSs may even attempt to be fair
- **Spooling:** queue output for a device, useful if device is "single user" (e.g., printer), i.e. can serve only one request at a time
- **Device reservation:** system calls for acquiring or releasing exclusive access to a device (are required)
- **Error handling:** generally get some form of error number or code when request fails, logged into system error log (e.g., transient write failed, disk full, device unavailable, ...)

4.5

PERFORMANCE

IO a major factor in system performance

- Demands CPU to execute device driver, kernel IO code, etc.
 - Context switches due to interrupts
 - Data copying
- Improving performance:
- Reduce number of context switches
 - Reduce data copying
 - Reduce number of interrupts by using large transfers, smart controllers, polling
 - Use DMA where possible
 - Balance CPU, memory, bus and IO performance for highest throughput.
- Improving IO performance remains a significant challenge...

KERNEL DATA STRUCTURES

To manage all this, the OS kernel must maintain state for IO components:

- Open file tables
- Network connections
- Character device states

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

Consider reading a file from disk for a process:

- Determine device holding file
- Translate name to device representation
- Physically read data from disk into buffer
- Make data available to requesting process
- Return control to process

4.6

SUMMARY

- Input/Output (IO)
 - Hardware
 - Device Classes
 - OS Interfaces
- Performing IO
 - Polling Mode
 - Interrupt Driven
 - Blocking vs Non-blocking
- Handling IO
 - Buffering & Strategies
 - Other Issues
 - Kernel Data Structures
 - Performance

4.7

[09] STORAGE

OUTLINE

- File Concepts
 - Filesystems
 - Naming Files
 - File Metadata
- Directories
 - Name Space Requirements
 - Structure
 - Implementation
- Files
 - Operations
 - Implementation
 - Access Control, Existence Control, Concurrency Control

1.1

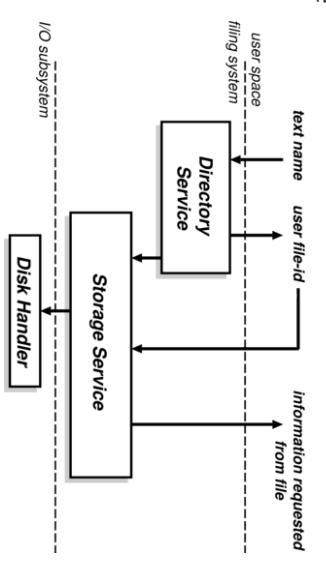
FILE CONCEPTS

- File Concepts
 - Filesystems
 - Naming Files
 - File Metadata
- Directories
- Files

1.2

FILESYSTEM

We will look only at very simple filesystems here, having two main components:



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

2.1

WHAT IS A FILE?

The basic abstraction for non-volatile storage:

- User abstraction – compare/contrast with segments for memory
- Many different types:
 - Data: numeric, character, binary
 - Program: source, object, executable
 - "Documents"
- Typically comprises a single contiguous logical address space
- Can have varied internal structure:
 - None: a simple sequence of words or bytes
 - Simple record structures: lines, fixed length, variable length
 - Complex internal structure: formatted document, relocatable object file

2.3

NAMING FILES

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

- **System file identifier** (SFID): (typically) a unique integer value associated with a given file, used within the filesystem itself
- **Human name**, e.g. hello.java: what users like to use
- May have a third, **User File Identifier** (UFID) used to identify open files in applications

Mapping from human name to SFID is held in a directory, e.g.,

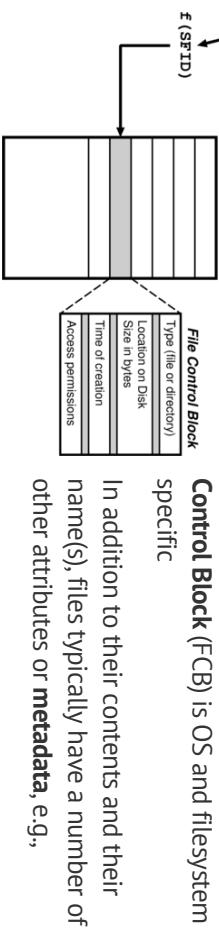
Note that directories are *also* non-volatile so they must be stored on disk along with files – which explains why the storage system sits "below" the directory service

Name	SFID
hello.java	12353
Makefile	23812
README	9742

2.4

FILE METADATA

NB. Having resolved the name to an SFID, the actual mapping from SFID to **File Control Block** (FCB) is OS and filesystem specific.



In addition to their contents and their name(s), files typically have a number of other attributes or **metadata**, 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**: data for protection, security and usage monitoring

2.5

WHAT IS A FILE?

OS split between *text* and *binary* is quite common where text files are treated as

- A sequence of lines each terminated by a special character, and
 - With an explicit EOF character (often)
- Can map everything to a byte sequence by inserting appropriate control characters, and interpretation in code. Question is, who decides:
- OS: may be easier for programmer but will lack flexibility
 - Programmer: has to do more work but can evolve and develop format

DIRECTORIES

- File Concepts
- Directories
 - Name Space Requirements
 - Structure
 - Implementation
- Files
 - allow one file have several different names
 - allow two (or, more generally, N) users to have the same name for different files
- Grouping: logical grouping of files by properties, e.g., "all Java programs", "all games"

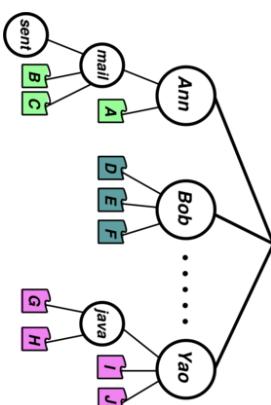
EARLY ATTEMPTS

- Single-level: one directory shared between all users
 - naming problem
 - 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.

Add a general hierarchy for more flexibility

3.1

STRUCTURE: TREE



Directories hold files or [further] directories, reflecting structure of organisation, users' files, etc

Create/delete files relative to a given directory

Efficient searching and arbitrary grouping capability

The human name is then the full path name, though these can get unwieldy, e.g., /usr/groups/X11R5/src/mit/server/os/4.2bsd/utils.c.
Resolve with **relative naming**, **login directory**, **current working directory**. Sub-directory deletion either by requiring directory empty, or by recursively deleting

3.2

REQUIREMENTS

A **directory** provides the means to translate a (user) name to the location of the file on-disk. What are the requirements?

- Efficiency: locating a file quickly.

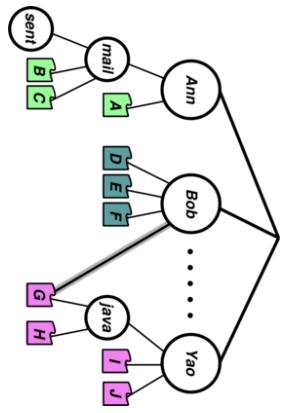
- Naming: user convenience
 - allow two (or, more generally, N) users to have the same name for different files

- Grouping
 - allow one file have several different names
 - allow two (or, more generally, N) users to have the same name for different files

3.3

3.4

STRUCTURE: DAG



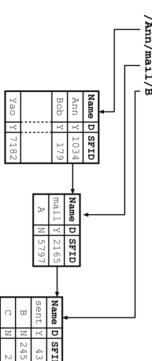
Hierarchy useful but only allows one name per file. Extend to **directed acyclic graph** (DAG) structure: allow shared subdirectories and files, and multiple aliases for same thing

Manage dangling references: use back-references or reference counts

Other issues include: **deletion** (more generally, permissions) and knowing when ok to free disk blocks; **accounting** and who gets "charged" for disk usage; and **cycles**, and how we prevent them

3.5

DIRECTORY IMPLEMENTATION



Directories are non-volatile so store as "files" on disk, each with own SFID

- Must be different types of file, for traversal
- Operations must also be explicit as info in directory used for access control, or could (eg) create cycles
- Explicit directory operations include:
 - Create/delete directory
 - List contents
 - Select current working directory
 - Insert an entry for a file (a "link")

3.6

FILES

- File Concepts
- Directories
- **Files**
 - Operations
 - Implementation
- Access Control, Existence Control, Concurrency Control

Basic paradigm of use is: open, use, close

Opening or creating a file:

UFID = open (<pathname>) or
UFID = create (<pathname>)

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

- Directory service recursively searching directories for components of <pathname>

- Eventually get SFID for file, from which UFIG created and returned
- Various modes can be used

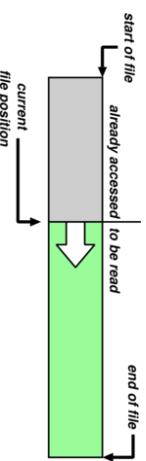
Closing a file: status = close(UFIG)

- Copy[new] file control block back to disk and invalidate UFIG

4.1

4.2

IMPLEMENTATION



Associate a cursor or file position with each open file (viz. Ufid), initialised to start of file

- Basic operations: read next or write next, e.g., `read(Ufid, buf, nbytes)`, or `read(Ufid, buf, nrecords)`

Access pattern:

- **Sequential:** adds `rewind(Ufid)` to above
- **Direct Access:** `read(N)` or `write(N)` using `seek(Ufid, pos)`
- Maybe others, e.g., append-only, indexed sequential access mode (ISAM)

4.3

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

- Can be mandatory or advisory
- Locks may be shared or exclusive
- Granularity may be file or subset

4.5

ACCESS CONTROL

File owner/creator should be able to control what can be done, by whom

- File usually only accessible if user has both directory and file access rights
- Former to do with lookup process – can't look it up, can't open it
- Assuming a DAG structure, do we use the presented or the absolute path

Access control normally a function of directory service so checks done at file open time

- E.g., `read`, `write`, `execute`, `(append?)`, `delete`, `[list]`, `rename`
- More advanced schemes possible (see later)

SUMMARY

- File Concepts
 - Filesystems
 - Naming Files
 - File Metadata
- Directories
 - Name Space Requirements
 - Structure
 - Implementation
- Files
 - Operations
 - Implementation
 - Access Control, Existence Control, Concurrency Control

4.4

[10] COMMUNICATION

OUTLINE

- Communication
 - Requirements
 - Inter-Thread Communication
 - Inter-Host Communication
 - Inter-Process Communication
- Inter-Process Communication
 - Concept
 - `fork(2), wait(2)`
 - Signals
 - Pipes
 - Named Pipes / FIFOs
 - Shared Memory Segments
 - Files
 - Unix Domain Sockets

COMMUNICATION

- Communication
 - Requirements
 - Inter-Thread Communication
 - Inter-Host Communication
 - Inter-Process Communication
- Inter-Process Communication
 - Concept
 - `fork(2), wait(2)`
 - Signals
 - Pipes
 - Named Pipes / FIFOs
 - Shared Memory Segments
 - Files
 - Unix Domain Sockets

1.1

1.2

REQUIREMENTS

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

- Data transferred must be in a commonly-understood format (**syntax**)
- Data transferred must have mutually-agreed meaning (**semantics**)
- Data must be transferred according to mutually understood rules (**synchronisation**)

In computer communications, the parties in question come in a range of forms, typically:

- Threads
- Processes
- Hosts

Ignore problems of discovery, identification, errors, etc. for now

2.1

2.2

INTER-THREAD COMMUNICATION

It is a common requirement for two running threads to need to communicate

- E.g. to coordinate around access to a shared variable

If coordination is not implemented, then all sorts of problems can occur. Range of mechanisms to manage this:

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

Not discussed here!

- You'll get into the details next year in **Concurrent and Distributed Systems**
- (Particularly the first half, on *Concurrency*)

2.3

INTER-HOST COMMUNICATION

Passing data between different hosts:

- Traditionally different physical hosts
- Nowadays often virtual hosts

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

Also not discussed here!

- In some sense it is "harder" than IPC because real networks are inherently:
 - **Unreliable**: data can be lost
 - **Asynchronous**: even if data is not lost, no guarantees can be given about when it arrived
- You'll see a lot more of this next year in **Computer Networking**

2.4

INTER-PROCESS COMMUNICATION

In the context of this course, we are concerned with **Inter-Process Communication** (IPC)

- What it says on the tin – communication between processes on the same host
 - Key point – it is possible to share memory between those processes
- Given the protection boundaries imposed by the OS, by design, the OS must be involved in any communication between processes
- Otherwise it would be tantamount to allowing one process to write over another's address space
 - We'll focus on POSIX mechanisms

2.5

INTER-PROCESS COMMUNICATION

In the context of this course, we are concerned with **Inter-Process Communication** (IPC)

- What it says on the tin – communication between processes on the same host
 - Key point – it is possible to share memory between those processes
- Given the protection boundaries imposed by the OS, by design, the OS must be involved in any communication between processes
- Otherwise it would be tantamount to allowing one process to write over another's address space
 - We'll focus on POSIX mechanisms

- Communication
 - **Inter-Process Communication**

- **Concept**
 - `fork(2), wait(2)`
 - Signals
 - Pipes
 - Named Pipes / FIFOs
 - Shared Memory Segments
 - Files
 - Unix Domain Sockets

3.1

CONCEPT

For IPC to be a thing, first you need multiple processes

- Initially created by running processes from a shell
- Subsequently may be created by those processes, ad infinitum
- (...until your machine dies from your fork bomb...)

Basic process mechanisms: `fork(2)` followed by `execve(2)` and/or `wait(2)`

Will look at that plus several other common POSIX mechanisms

3.2

SIGNALS

Simple asynchronous notifications on another process

- A range of signals (28 at my last count), defined as numbers
- Mapped to standard #defines, a few of which have standard mappings to numbers

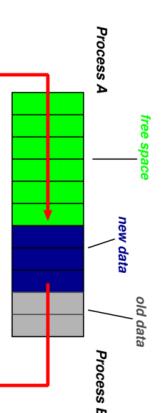
Among the more common ones:

- `SIGHUP`: hangup the terminal (1)
- `SIGINT`: terminal interrupt (2)
- `SIGKILL`: terminate the process [cannot be caught or ignored] (9)
- `SIGTERM`: terminate process (15)
- `SIGSEGV`: segmentation fault – process made an invalid memory reference
- `SIGUSR1 / 2`: two user signals [system defined numbers]

Use `sigaction(2)` to specify what function the signalled process should invoke on receipt of a given signal

3.3

PIPES



Simply put, `fork(2)` allows a process to clone itself:

- **Parent** process creates **child** process
 - Child receives copy-on-write (COW) snapshot of parent's address space
- Parent typically then either:
 - Detaches from child – hands responsibility back to init process
 - Waits for child – calling `wait(2)`, parent blocks until child exits

3.4

- Simplest form of IPC: `pipe(2)` returns a pair of **file descriptors**
- (`fd[0], fd[1]`) are the (read, write) fds

Coupled with `fork(2)`, can now communicate between processes:

- Invoke `pipe(2)` to get read/write fds
- `fork(2)` to create child process
- Parent and child then both have read/write fds available, and can communicate

3.5

NAMED PIPES / FIFOS

The same as `pipe(2)` – except that it has a name, and isn't just an array of two `fds`

- This means that the two parties can coordinate without needing to be in a parent/child relationship
- All they need is to share the (path)name of the FIFO

Then simply treat as a file:

- `open(2)`
- `read(2)`
- `write(2)`

`open(2)` will block by default, until some other process opens the FIFO for reading

- Can set non-blocking via `O_NDELAY`

3.6

FILES

Locking can be mandatory (enforced) or advisory (cooperative)

- Advisory is more widely available
- `fcntl(2)` sets, tests and clears the lock status
- Processes can then coordinate over access to files
- `read(2), write(2), seek(2)` to interact and navigate

Memory Mapped Files present a simpler – and often more efficient – API

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

3.7

UNIX DOMAIN SOCKETS

Sockets are commonly used in network programming – but there is (effectively) a shared memory version for use between local processes, having the same API:

- `socket(2)` creates a socket, using `AF_UNIX`
- `bind(2)` attaches the socket to a file
- The interact as with any socket
 - `accept(2), listen(2), recv(2), send(2)`
 - `sendto(2), recvfrom(2)`

Finally, `socketpair(2)` uses sockets to create a full-duplex pipe

- Can read/write from both ends

3.8

SHARED MEMORY SEGMENTS

What it says on the tin – obtain a segment of memory that is shared between two (or more) processes

- `shmget(2)` to get a segment
- `shmat(2)` to attach to it

Then read and write simply via pointers – need to impose concurrency control to avoid collisions though

Finally:

- `shmdt(2)` to detach
- `shmctl(2)` to destroy once you know no-one still using it

3.9

SUMMARY

- Communication
 - Requirements
 - Inter-Thread Communication
 - Inter-Host Communication
 - Inter-Process Communication
- Inter-Process Communication
 - Concept
 - `fork(2), wait(2)`
 - Signals
 - Pipes
 - Named Pipes / FIFOs
 - Shared Memory Segments
 - Files
 - Unix Domain Sockets

4

[11] CASE STUDY: UNIX

OUTLINE

- Introduction
- Design Principles
 - Structural, Files, Directory Hierarchy
- Filesystem
 - Files, Directories, Links, On-Disk Structures
 - Mounting Filesystems, In-Memory Tables, Consistency
- IO
 - Implementation, The Buffer Cache
- Processes
 - Unix Process Dynamics, Start of Day, Scheduling and States
- The Shell
 - Examples, Standard IO
- Summary

INTRODUCTION

- **Introduction**
- Design Principles
- Filesystem
- IO
- Processes
- The Shell
- Examples, Standard IO
- Summary

1.1

HISTORY (I)

First developed in 1969 at Bell Labs (Thompson & Ritchie) as reaction to bloated Multics. Originally written in PDP-7 asm, but then (1973) rewritten in the "new" high-level language C so it was easy to port, alter, read, etc. Unusual due to need for performance

6th edition ("V6") was widely available (1976), including source meaning people could write new tools and nice features of other OSes promptly rolled in

V6 was mainly used by universities who could afford a minicomputer, but not necessarily all the software required. The first really portable OS as same source could be built for three different machines (with minor asm changes)

Bell Labs continued with V8, V9 and V10 (1989), but never really widely available because V7 pushed to Unix Support Group (USG) within AT&T
AT&T did System III first (1982), and in 1983 (after US government split BellS), System V. There was no System IV

1.2

2.1

2.2

HISTORY (III)

By 1978, V7 available (for both the 16-bit PDP-11 and the new 32-bit VAX-11). Subsequently, two main families: AT&T "System V", currently SVR4, and Berkeley: "BSD", currently 4.4BSD

Later standardisation efforts (e.g. POSIX, X/Open) to homogenise

USDL did SVR2 in 1984; SVR3 released in 1987; SVR4 in 1989 which supported the POSIX.1 standard

In parallel with AT&T story, people at University of California at Berkeley (UCB) added virtual memory support to "32V" [32-bit V7 for VAX] and created 3BSD

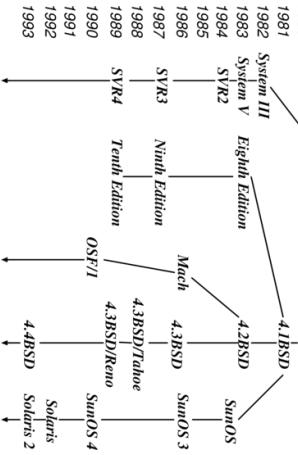
Best known Unix today is probably Linux, but also get FreeBSD, NetBSD, and (commercially) Solaris, OSF/1, IRIX, and Tru64

2.3

SIMPLIFIED UNIX FAMILY TREE

Linux arises (from Minix?) around 1991 (version 0.01), or more realistically, 1994 (version 1.0). Linux version 2.0 out 1996. Version 2.2 was out in 1998/ early 1999?

You're not expected to memorise this



DESIGN FEATURES

Ritchie & Thompson (CACM, July 74), identified the (new) features of Unix:

- A hierarchical file system incorporating demountable volumes
- Compatible file, device and inter-process IO (naming schemes, access control)
- Ability to initiate asynchronous processes (i.e., address spaces = heavyweight)
- System command language selectable on a per-user basis

Completely novel at the time: prior to this, everything was "inside" the OS. In Unix separation between essential things (kernel) and everything else

Among other things: allows user wider choice without increasing size of core OS; allows easy replacement of functionality – resulted in over 100 subsystems including a dozen languages

Highly portable due to use of high-level language

Features which were not included: real time, multiprocessor support

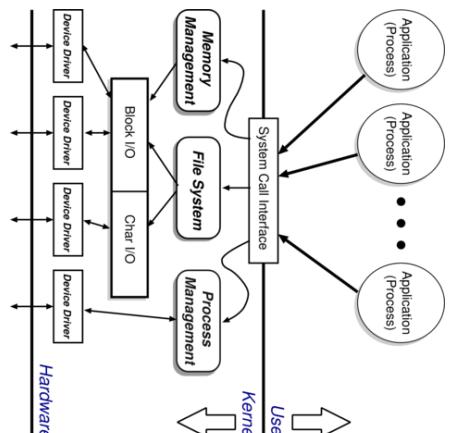
3.2

FILESYSTEM

- Introduction
- Design Principles
- **Filesystem**
 - Files, Directories, Links, On-Disk Structures
 - Mounting Filesystems, In-Memory Tables, Consistency
- IO
 - Processes
 - The Shell
 - Summary

STRUCTURAL OVERVIEW

Clear separation between user and kernel portions was the big difference between Unix and contemporary systems – only the **essential** features *inside* OS, not the editors, command interpreters, compilers, etc.



Processes are unit of scheduling and protection: the command interpreter ("shell") just a process

No concurrency within kernel

All IO looks like operations on files: in Unix, everything is a file

3.3

FILE ABSTRACTION

File as an unstructured sequence of bytes which was relatively unusual at the time: most systems lent towards files being composed of records

- Cons: don't get nice type information; programmer must worry about format of things inside file
- Pros: less stuff to worry about in the kernel; and programmer has flexibility to choose format within file!

Represented in user-space by a file descriptor (fd) this is just an opaque identifier – a good technique for ensuring protection between user and kernel

4.1

FILE OPERATIONS

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)`

The kernel keeps track of the current position within the file

Devices are represented by special files:

- Support above operations, although perhaps with bizarre semantics
- Also have `ioctl` for access to device-specific functionality

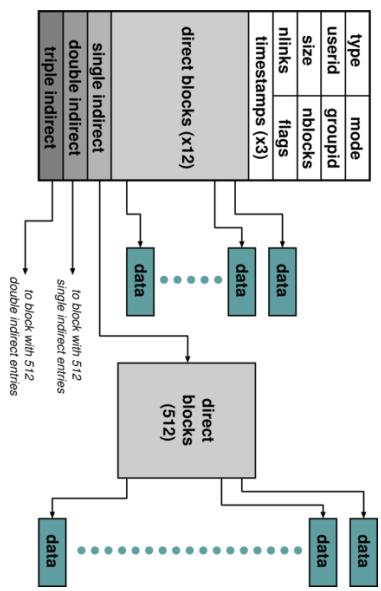
4.3

ASIDE: PASSWORD FILE

- `/etc/passwd` holds list of password entries of the form user-name:`:encrypted-passwd:home-directory:shell`
- Also contains user-id, group-id (default), and friendly name.
- 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. To login:
 - Get user name
 - Get password
 - Encrypt password
 - Check against version in `/etc/password`
 - If ok, instantiate login shell
 - Otherwise delay and retry, with upper bound on retries
 - Publicly readable since lots of useful info there but permits offline attack
 - Solution: shadow passwords (`/etc/shadow`)

4.4

FILE SYSTEM IMPLEMENTATION



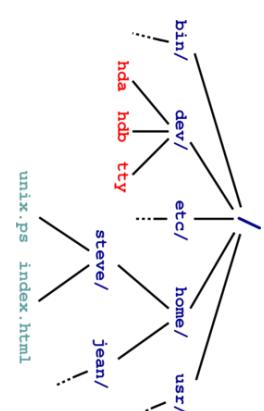
4.5

Inside the kernel, a file is represented by a data structure called an **index-node** or **i-node** which hold file meta-data: owner, permissions, reference count, etc. and location on disk of actual data (file contents)

DIRECTORY HIERARCHY

Directories map names to files (and directories) starting from distinguished root directory called `/`

Fully qualified pathnames mean performing traversal from root



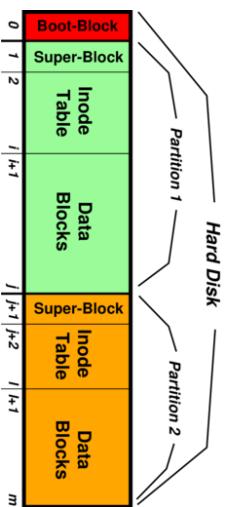
Every directory has `.` and `..` entries: refer to self and parent respectively. Also have shortcut of **current working directory** (`cwd`) which allows relative path names; and the shell provides access to home directory as `~username` (e.g. `~mort/`). Note that kernel knows about former but not latter

Structure is a tree in general though this is slightly relaxed

I-NODES

- Why don't we have all blocks in a simple table?
- Why have first few in inode at all?
- How many references to access blocks at different places in the file?
- If block can hold 512 block-addresses (e.g. blocks are 4kB, block addresses are 8 bytes), what is max size of file (in blocks)?
- Where is the filename kept?

ON-DISK STRUCTURES



A disk consists of a boot block followed by one or more partitions. Very old disks would have just a single partition. Nowadays have a **boot block** containing a partition table allowing OS to determine where the filesystems are

Figure shows two completely independent filesystems; this is not replication for redundancy. Also note $|{\text{inode table}}| \gg |{\text{superblock}}|, |{\text{data blocks}}| \gg |{\text{inode table}}|$

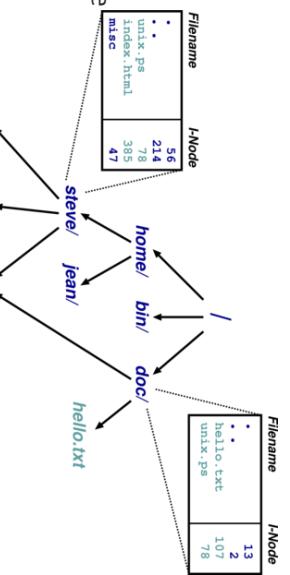
4.7

DIRECTORIES AND LINKS

Directory is (just) a file which maps filenames to i-nodes – that is, it has its own i-node pointing to its contents

An instance of a file in a directory is a (hard) link because the reference count in the 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 partition is just a contiguous range of N fixed-size blocks of size k for some N and k , and a Unix filesystem resides within a partition

Common block sizes: 512B, 1kB, 2kB, 4kB, 8kB

Superblock contains info such as:

- Number of blocks and free blocks in filesystem
- Start of the free-block and free-inode list
- Various bookkeeping information

Free blocks and inodes intermingle with allocated ones

On-disk have a chain of tables (with head in superblock) for each of these. Unfortunately this leaves superblock and inode-table vulnerable to head crashes so we must replicate in practice. In fact, now a wide range of Unix filesystems that are completely different; e.g., log-structure

4.8

4.9

4.10

MOUNTING FILESYSTEMS

Entire filesystems can be mounted on an existing directory in an already mounted filesystem

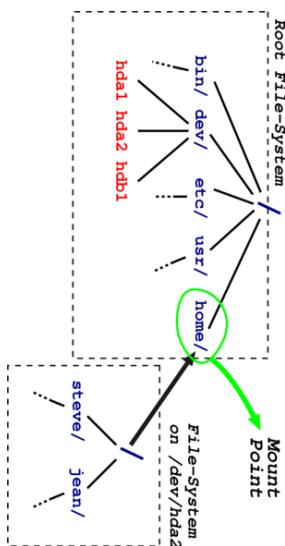
At very start, only `/` exists so must mount a root filesystem

Subsequently can mount other filesystems, e.g.

```
mount( "/dev/hda2",
"/home", options )
```

Provides a unified name-space: e.g. access `/home/mort/` directly (contrast with Windows9x or NT)

Cannot have hard links across mount points: why? What about soft links?



ACCESS CONTROL

Owner			Group			World		
R	W	E	R	W	E	R	W	E
Green	Red	Red	Red	Red	Red	Red	Red	Red
Green	Red	Red	Red	Red	Red	Red	Red	Red
Green	Red	Red	Red	Red	Red	Red	Red	Red

Owner			Group			World		
R	W	E	R	W	E	R	W	E
Green	Red	Red	Red	Red	Red	Red	Red	Red
Green	Red	Red	Red	Red	Red	Red	Red	Red
Green	Red	Red	Red	Red	Red	Red	Red	Red

= 0640

= 0755

Access control information held in each inode

- Three bits for each of owner, group and world: read, write and execute
- What do these mean for directories? Read entry, write entry, traverse directory

In addition have setuid and setgid bits:

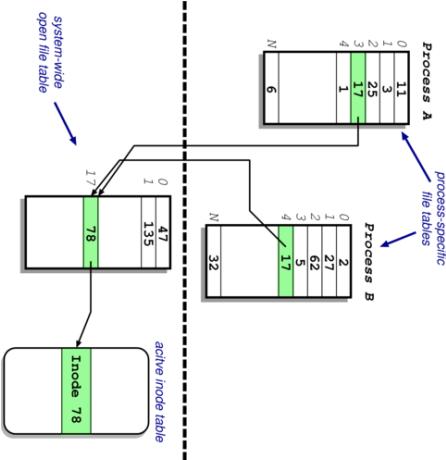
- Normally processes inherit permissions of invoking user
- Setuid/setgid allow user to "become" someone else when running a given program
- E.g. prof owns both executable test (0711 and setuid), and score file (0600)

IN-MEMORY TABLES

Recall process sees files as file descriptors

In implementation these are just indices into process-specific open file table

Entries point to system-wide open file table. Why?
These in turn point to (in memory) inode table



CONSISTENCY ISSUES

To delete a file, use the `unlink` system call – from the shell, this is `rm <filename>`

Procedure is:

- Check if user has sufficient permissions on the file (must have write access)
- Check if user has sufficient permissions on the directory (must have write access)
- If ok, remove entry from directory
- Decrement reference count on inode
- If now zero: free data blocks and free inode

If crash: must check entire filesystem for any block unreferenced and any block double referenced

Crash detected as OS knows if crashed because root fs not unmounted cleanly

4.14

UNIX FILESYSTEM: 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 filesystems 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

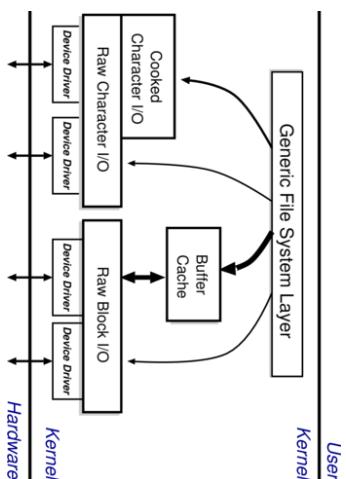
IO

- Introduction
- Design Principles
- Filesystem
- **IO**
 - **Implementation, The Buffer Cache**
 - Processes
 - The Shell
 - Summary

5.1

IO IMPLEMENTATION

- Everything accessed via the file system
- Two broad categories: block and character; ignoring low-level gore:
 - Character IO low rate but complex – most functionality is in the "cooked" interface
 - Block IO simpler but performance matters – emphasis on the buffer cache



THE BUFFER CACHE

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

On read do:

- Locate relevant blocks (from inode)
- Check if in buffer cache
- If not, read from disk into memory
- Return data from buffer cache

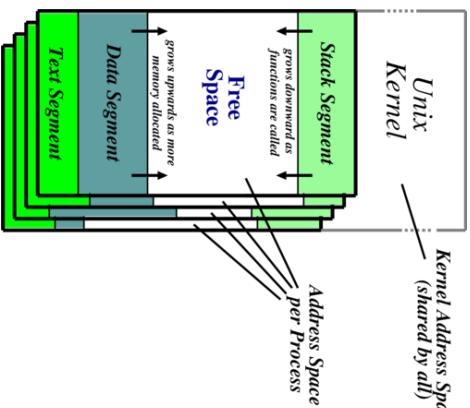
On write do same first three, and then update version in cache, not on disk

- "Typically" prevents 85% of implied disk transfers
- But when does data actually hit disk?
- Call sync every 30 seconds to flush dirty buffers to disk
- Can cache metadata too – what problems can that cause?

5.2

PROCESSES

- Introduction
- Design Principles
- Filesystem
- I/O
- **Processes**
 - **Unix Process Dynamics, Start of Day, Scheduling and States**
 - The Shell
 - Summary



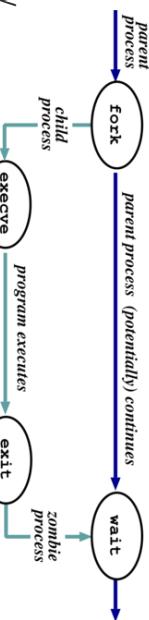
6.1

UNIX PROCESS DYNAMICS

Process is represented by an opaque process id (pid), organised hierarchically with parents creating children. Four basic operations:

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

`fork()` nearly always followed by `exec()` leading to `vfork()` and/or copy-on-write (COW). Also makes a copy of entire address space which is not terribly efficient



6.3

START OF DAY

Kernel (/vmlinix) loaded from disk (how – where's the filesystem?) and execution starts. Mounts root filesystem. Process 1 (/etc/init) starts hand-crafted

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

- Opens terminal special file (e.g. /dev/ttys0)
- Duplicates the resulting fd twice.
- Forks an /etc/ttys process.

Each ttys process next: initialises the terminal; outputs the string login: & waits for input; execve()'s /bin/login login then: outputs "password:" & waits for input; encrypts password and checks it against /etc/passwd; if ok, sets uid & gid, and execve() shell Patriarch init resurrects /etc/ttys on exit

6.4

UNIX PROCESSES

Recall: a process is a program in execution

Processes have three segments: text, data and stack. Unix processes are heavyweight

Text: holds the machine instructions for the program

Stack: used for activation records (i.e. storing local variables, parameters, etc.)

- Unix Process Dynamics, Start of Day, Scheduling and States

Processes

• Processes

▪ Unix Process Dynamics, Start of Day, Scheduling and States

• The Shell

• Summary

UNIX PROCESS SCHEDULING (I)

- Priorities 0-127; user processes \geq PUSER = 50. Round robin within priorities, quantum 100ms.
- Priorities are based on usage and nice, i.e.

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

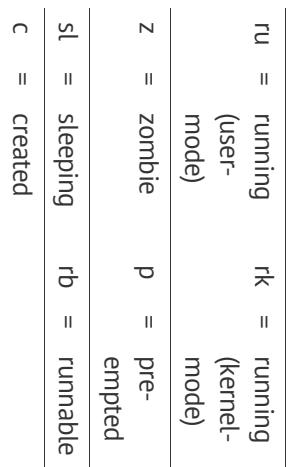
gives the priority of process j at the beginning of interval i where:

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

- and nice_j is a (partially) user controllable adjustment parameter in the range $[-20, 20]$
- load_j is the sampled average length of the run queue in which process j resides, over the last minute of operation

6.5

UNIX PROCESS STATES



NB. This is simplified – see *Concurrent Systems* section 23.14 for detailed descriptions of all states/transitions

6.6

THE SHELL

- Introduction
- Design Principles
- Filesystem
- IO
- Processes
- **The Shell**
 - Examples, Standard IO
- Summary

7.1

UNIX PROCESS SCHEDULING (III)

- Thus if e.g. load is 1 this means that roughly 90% of 1s CPU usage is "forgotten" within 5s
- Base priority divides processes into bands; CPU and nice components prevent processes moving out of their bands. The bands are:
 - Swapper; Block IO device control; File manipulation; Character IO device control; User processes
 - Within the user process band the execution history tends to penalize CPU bound processes at the expense of IO bound processes

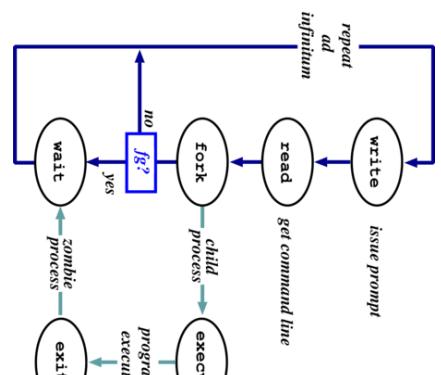
THE SHELL

Shell just a process like everything else.
Needn't understand commands, just files

Uses path for convenience, to avoid needing
fully qualified pathnames

Conventionally & specifies background

Parsing stage (omitted) can do lots: wildcard
expansion ("globbing"), "tilde" processing



```
$ ./prod
$ /Users/mort/src
$ ls -F
avk-scripts/          kat-aka/
backup-p-scripts/     mrt-0/
bibbz-0.9.1/          ocal/
c-cutlts/             ocal/
drtrace/              ocaml-libs/
exapraxa-gae/        ocaml-mrt/
external/              ocaml-pst/
junk/                 ocaml-org/
$ cd python-scripts/
$ ls -LF
total 224
-rw-r--r--  1 mort  staff  17987  2 Jan  2010 LICENSE
-rw-r--r--  1 mort  staff  1592   5 Jan  2013 README.md
-rwxr-xr-x  1 mort  staff  6206   2 Dec  2013 bberry.py*
-rwxr-xr-x  1 mort  staff  7286  14 Jul  2015 bibjson.py*
-rwxr-xr-x  1 mort  staff  7205  2 Dec  2013 cell.py*
-rw-r--r--  1 mort  staff  1860   2 Dec  2013 cc4uni-fafe.f.py
-rwxr-xr-x  1 mort  staff  1153   2 Dec  2013 filebombe.py*
```

Prompt is \$. Use man to find out about commands. User friendly?

7.2

SUMMARY

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`

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

- Pipeline is better (e.g. `ls | wc >results`)
- Unix commands are often filters, used to build very complex command lines
- Redirection can cause some buffering subtleties

SHELL EXAMPLES

```
$ ./prod
```

```
/Users/mort/src
```

```
$ ls -F
```

```
avk-scripts/          kat-aka/
backup-p-scripts/     mrt-0/
bibbz-0.9.1/          ocal/
c-cutlts/             ocal/
drtrace/              ocaml-libs/
exapraxa-gae/        ocaml-mrt/
external/              ocaml-pst/
junk/                 ocaml-org/
$ cd python-scripts/
$ ls -LF
total 224
-rw-r--r--  1 mort  staff  17987  2 Jan  2010 LICENSE
-rw-r--r--  1 mort  staff  1592   5 Jan  2013 README.md
-rwxr-xr-x  1 mort  staff  6206   2 Dec  2013 bberry.py*
-rwxr-xr-x  1 mort  staff  7286  14 Jul  2015 bibjson.py*
-rwxr-xr-x  1 mort  staff  7205  2 Dec  2013 cell.py*
-rw-r--r--  1 mort  staff  1860   2 Dec  2013 cc4uni-fafe.f.py
-rwxr-xr-x  1 mort  staff  1153   2 Dec  2013 filebombe.py*
```

7.3

8.1

MAIN UNIX FEATURES

- 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)
 - Thus can recursively mount filesystems
- Heavy-weight processes
- IO: 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 V7 had inflexible IPC, inefficient memory management, and poor kernel concurrency
- Later versions address these issues.

SUMMARY

- Introduction
- Design Principles
 - Structural, Files, Directory Hierarchy
- Filesystem
 - Files, Directories, Links, On-Disk Structures
 - Mounting Filesystems, In-Memory Tables, Consistency
- IO
 - Implementation, The Buffer Cache
- Processes
 - Unix Process Dynamics, Start of Day, Scheduling and States
- The Shell
 - Examples, Standard IO
- Summary

8.2

9

[12] CASE STUDY: WINDOWS NT

OUTLINE

- Introduction
- Design Principles
- Design
 - Structural
 - HAL, Kernel
 - Processes and Threads, Scheduling
 - Environmental Subsystems
- Objects
 - Manager, Namespace
 - Other Managers: Process, VM, Security Reference, IO, Cache
- Filesystems
 - FAT16, FAT32, NTFS
 - NTFS: Recovery, Fault Tolerance, Other Features
- Summary

1.1

INTRODUCTION

- **Introduction**
- Design Principles
- Design
 - Structural
 - HAL, Kernel
 - Processes and Threads, Scheduling
 - Environmental Subsystems
- Objects
 - Manager, Namespace
 - Other Managers: Process, VM, Security Reference, IO, Cache
- Filesystems
 - FAT16, FAT32, NTFS
 - NTFS: Recovery, Fault Tolerance, Other Features
- Summary

1.2

PRE-HISTORY

- Microsoft and IBM co-developed OS/2 – in hand-written 80286 assembly! As a result, portability and maintainability weren't really strong features so in 1988 Microsoft decided to develop a "new technology" portable OS supporting both OS/2 and POSIX APIs
- Goal: A 32-bit preemptive multitasking operating system for modern microprocessors
- Originally, NT was supposed to use the OS/2 API as its native environment, but during development NT was changed to use the Win32 API, reflecting the popularity of Windows 3.0

2.1

NEW TECHNOLOGY

After OS/2, MS decide they need "New Technology":

- 1988: Dave Cutler recruited from DEC
- 1989: team (~10 people) starts work on a new OS with a micro-kernel architecture
- Team grew to about 40 by the end, with overall effort of 100 person-years
- July 1993: first version (3.1) introduced. Sucked
- September 1994: NT 3.5 released, providing mainly size and performance optimisations
- May 1995: NT 3.51 with support for the Power PC, and more performance tweaks
- July 1996: NT 4.0 with "new" (Windows 95) look 'n' feel. Saw some desktop use but mostly limited to servers. Various functions pushed back into the kernel, notably graphics rendering

2.3

DESIGN PRINCIPLES

- Introduction
- **Design Principles**
- Design
- Objects
- Filesystems
- Summary

2.4

KEY GOALS

- **Portability:** hence written in C/C++ with the HAL to hide low-level details
- **Security:** new uniform access model implemented via object manager, and certified to US DOD level C2
- **POSIX compliance:** believed this would win sales, but desire to support both POSIX and OS/2 (later WIN32) impacted overall design
- **Multiprocessor support:** most small OSs didn't have this, and traditional kernel schemes are less well suited
- **Extensibility:** because, sometimes, we get things wrong; coupled with the above point, most directly led to the use of a micro-kernel design
- **International support:** sell to a bigger market; meant adopting UNICODE as fundamental internal naming scheme
- **Compatibility with MS-DOS/Windows:** don't want to lose customers, but achieved partial compatibility only...

CONTINUED EVOLUTION

- Feb 2000: NT 5.0 aka Windows 2000. Borrows from windows 98 look 'n' feel. Provides server and workstation versions, latter of which starts to get wider use.
- Oct 2001: Windows XP (NT 5.1) launched with home and professional editions. Finally kills Win9x. Several "editions" including Media Center [2003], 64-bit [2005] and Service Packs (SP1, SP2). 4.5 million lines of code
- 2003: Server product 2K3 (NT 5.2), basically the same modulo registry tweaks, support contract and of course cost. Comes in many editions
- 2006: Windows Vista (NT 6.0). More security, more design, new APIs
- 2009: Windows 7 (NT 7.0). Focused more on laptops and touch devices
- 2012: Windows 8 (NT 8.0). Radical new UI with tiles, focused on touch at least as much as supporting mouse/keyboard
- 2013: Windows 8.1 (NT 8.1). Back off the UI a bit, more customisation
- 2015: Windows 10 (NT 10.0). More connectivity, for and between devices

3.1

OTHER GOALS

- **Reliability:** NT uses hardware protection for virtual memory and software protection mechanisms for operating system resources
- **Compatibility:** applications that follow the IEEE 1003.1 (POSIX) standard can be compiled to run on NT without changing the source code
- **Performance:** NT subsystems can communicate with one another via high-performance message passing
- **Preemption:** of low priority threads enable the system to respond quickly to external events
- Designed for **symmetrical multiprocessing**

DESIGN

- Introduction
- Design Principles
- **Design**
 - Structural
 - HAL, Kernel
 - Processes and Threads, Scheduling
 - Environmental Subsystems
- Objects
- Filesystems
- Summary

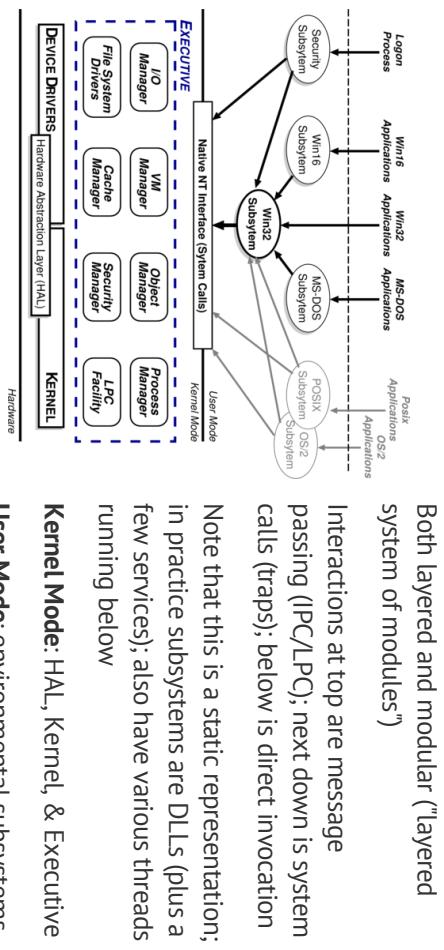
3.3

THE RESULT

Development of a system which was:

- Written in high-level languages (C and C++)
- Hence portable to other machines, with
- Processor-dependent code isolated in a dynamic link library (HAL)
- Based around a micro-kernel
- Hence extensibility and multiprocessor support
- E.g. environmental subsystems

STRUCTURAL OVERVIEW



3.4

Both layered and modular ("layered system of modules")
 Interactions at top are message passing (IPC/LPC); next down is system calls (traps); below is direct invocation
 Note that this is a static representation; in practice subsystems are DLLs (plus a few services); also have various threads running below

Kernel Mode: HAL, Kernel, & Executive

User Mode: environmental subsystems, protection subsystem

4.1

4.2

KERNEL MODE

Hardware Abstraction Layer (HAL): Layer of software (HAL.DLL) hiding hardware details, e.g., interrupt mechanisms, DMA controllers, multiprocessor communication mechanisms. Many implementations to the same interface

Kernel: Foundation for the executive and the subsystems, its execution is never preempted (it can be interrupted, but will always resume)

Four main responsibilities:

1. **CPU scheduling:** hybrid dynamic/static priority scheduling
2. **Interrupt and exception handling:** kernel provides trap handling when exceptions and interrupts are generated by hardware or software. If the trap handler can't handle the exception, the kernel's exception dispatcher does. Handle interrupts by either ISR or internal kernel routine
3. **Low-level processor synchronisation:** spin locks that reside in global memory to achieve multiprocessor mutual exclusion, normally provided by HAL
4. **Recovery after a power failure**

4.3

PROCESSES AND THREADS

NT splits the *virtual processor* into two parts:

- A **process**, the unit of resource ownership. Each has:
 - A security token
 - A virtual address space
 - A set of resources (object handles)
 - One or more threads
- A **thread**, the unit of dispatching. Each has:
 - A scheduling state (ready, running, etc.)
 - Other scheduling parameters (priority, etc.)
 - A context slot
 - An associated process (generally)

Threads have one of **six states**: ready, standby, running, waiting, transition, terminated. They are **co-operative**: all in a process share the same address space & object handles; **lightweight**: less work to create/delete than processes (shared virtual address spaces)

4.4

CPU SCHEDULING

A process starts via the CreateProcess routine, loading any dynamic link libraries that are used by the process and creating a primary thread. Additional threads can be created via the CreateThread function

Hybrid static/dynamic priority scheduling:

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

Default quantum 2 ticks (~20ms) on Workstation, 12 ticks (~120ms) on Server

4.5

KERNEL

Kernel is **object oriented**; all objects either dispatcher objects or control objects

- Dispatcher objects have to do with dispatching and synchronisation, i.e. they are active or temporal things like
 - Threads: basic unit of [CPU] dispatching
 - Events: record event occurrences & synchronise
 - Timer: tracks time, signals "time-outs"
 - Mutexes: mutual exclusion in kernel mode
 - Semaphores: as above, but work in user mode too
- Control objects represent everything else, e.g.,
 - Process: representing VAS and miscellaneous other bits
 - Interrupt: binds ISR to an interrupt source [HAL]

4.6

CPU SCHEDULING

Some very strange things to remember:

- When thread blocks, it loses 1/3 tick from quantum
- When thread preempted, moves to head of own run queue

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

- On return from IO, 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

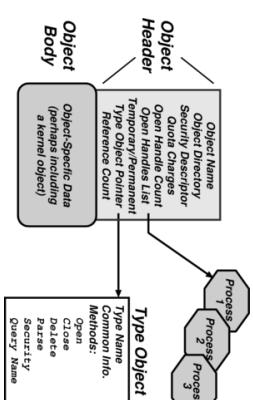
On Workstation also get quantum stretching:

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

4.7

OBJECTS

- Introduction
- Design Principles
- Design
- Objects
 - Manager, Namespace
 - Other Managers: Process, VM, Security Reference, IO, Cache
- Filesystems
- Summary



OBJECTS AND MANAGERS

- In Unix, everything is a file – in NT, everything is an object
- Every resource in NT is represented by an (**executive**) **object**
 - Kernel objects are re-exported at executive level by encapsulation
 - Objects comprise a header and a body, and have a type (approximately 15 types in total)

4.8

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. It 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
- 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

5.1

5.2

THE OBJECT MANAGER

Responsible for:

- Creating and tracking objects and object handles. An object handle represents an open object and is process-local, somewhat analogous to an fd
- Performing security checks
- Objects are manipulated by a standard set of methods, namely `create`, `open`, `close`, `delete`, `query_name`, `parse` and `security`. These are usually per type ("class") and hence implemented via indirection through the associated type object. Not all will be valid (specified) for all object types
 - `handle = open(objectname, accessmode)`
 - `result = service(handle, arguments)`
- A process gets an object handle by creating an object, by opening an existing one, by receiving a duplicated handle from another process, or by inheriting a handle from a parent process

5.3

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

...thus can support Posix, OS/2 and Win32 models

- It's up to environmental subsystem that owns the process to handle any hierarchical relationships (e.g. inheritance, cascading termination, etc)
- E.g., as noted above, in Win32: a process is started via the `CreateProcess()` function which loads any dynamic link libraries that are used by the process and creates a primary thread; additional threads can be created by the `CreateThread()` function

5.4

VIRTUAL MEMORY MANAGER

Assumes that the underlying hardware supports virtual to physical mapping, a paging mechanism, transparent cache coherence on multiprocessor systems, and virtual address aliasing. NT employs **paged virtual memory management**. The VMM provides processes with services to:

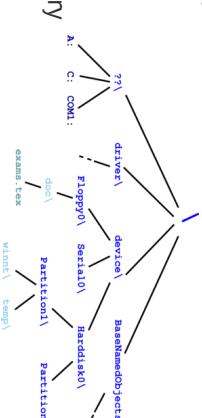
- Allocate and free virtual memory via two step process: reserve a portion of the process's address space, then commit the allocation by assigning space in the NT paging file
- Modify per-page protections, in one of six states: valid, zeroed, free, standby, modified and bad
- Share portions of memory using **section objects** (~software segments), based versus non-based, as well as memory-mapped files
 - A **section object** is a region of [virtual] memory which can be shared, containing: max size, page protection, paging file (or mapped file if mmap) and based vs non-based (meaning does it need to appear at same address in all process address spaces (based), or not (non-based)?)

5.5

THE OBJECT NAMESPACE

Objects (optionally) have a name, temporary or permanent, given via the NT executive

The Object Manager manages a hierarchical namespace, shared between all processes. The namespace is implemented via directory objects analogous to filesystem directories



Each object is protected by an access control list. Naming domains (implemented via `parse`) mean filesystem namespaces can be integrated

Object names structured like file path names in MS-DOS and Unix. Symbolic link objects allow multiple names (aliases) for the same object. Modified view presented at API level: the Win32 model has multiple "root" points (e.g., C::, D::, etc) so even though was all nice & simple, gets screwed up

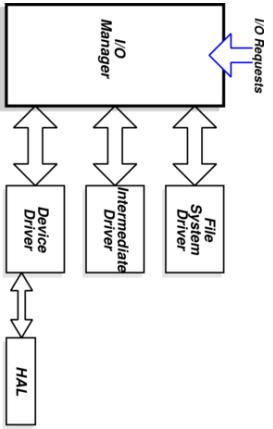
SECURITY REFERENCE MANAGER

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

- Every time a process opens handle to an object, check that process's security token and object's ACL
- Compare with Unix (filesystem, networking, window system, shared memory, ...)

5.7

IO MANAGER



The IO Manager is responsible for file systems, cache management, device drivers

Keeps track of which installable file systems are loaded, manages buffers for IO requests, and works with VMM to provide memory-mapped files

Controls the NT cache manager, which handles caching for the entire IO system
(ignore network drivers for now)

5.8

IO OPERATIONS

Basic model is asynchronous:

- Each IO operation explicitly split into a request and a response
- **IO Request Packet** (IRP) used to hold parameters, results, etc.

This allows high levels of flexibility in implementing IO type (can implement synchronous blocking on top of asynchronous, other way round is not so easy)
Filesystem & device drivers are stackable (plug'n'play)

5.9

LOCAL PROCEDURE CALL FACILITY

Local Procedure Call (LPC) (or IPC) passes requests and results between client and server processes within a single machine

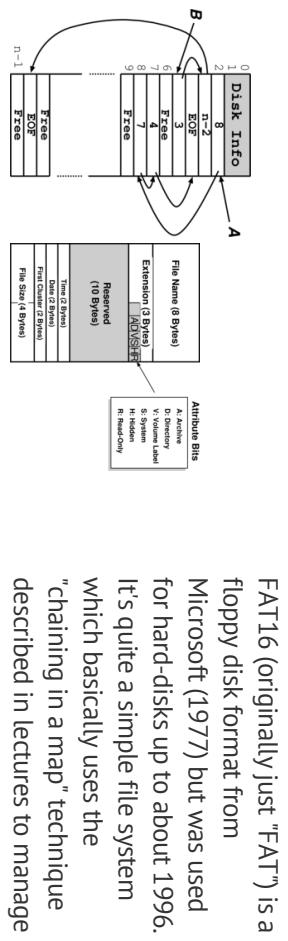
- Used to request services from the various NT environmental subsystems
 - Three variants of LPC channels:
 1. small messages (≤ 256 bytes): copy messages between processes
 2. zero copy: avoid copying large messages by pointing to a shared memory section object created for the channel
 3. quick LPC: used by the graphical display portions of the Win32 subsystem

5.10

CACHE MANAGER

- Caches "virtual blocks", keeping track of cache "lines" as offsets within a file rather than a volume – disk layout & volume concept abstracted away
 - No translation required for cache hit
 - Can get more intelligent prefetching
- Completely unified cache:
 - Cache "lines" all in virtual address space.
 - Decouples physical & virtual cache systems: e.g. virtually cache in 256kB blocks, physically cluster up to 64kB
- NT virtual memory manager responsible for actually doing the IO
 - Allows lots of FS cache when VM system lightly loaded, little when system is thrashing
- NT/2K also provides some user control:
 - If specify temporary attrib when creating file means it will never be flushed to disk unless necessary
 - If specify write through attrib when opening a file means all writes will synchronously complete

FILE SYSTEMS: FAT16



FAT16 (originally just "FAT") is a floppy disk format from Microsoft (1977) but was used for hard-disks up to about 1996. It's quite a simple file system which basically uses the "chaining in a map" technique described in lectures to manage files

A file is a linked list of clusters: a cluster is 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 means a max 2GB partition with 32kB clusters (and big cluster size is bad)

- NT/2K also provides some user control:
 - If specify temporary attrib when creating file means it will never be flushed to disk unless necessary
 - If specify write through attrib when opening a file means all writes will synchronously complete
- NT/2K also provides some user control:
 - If specify temporary attrib when creating file means it will never be flushed to disk unless necessary
 - If specify write through attrib when opening a file means all writes will synchronously complete

5.11

FILESYSTEMS

- Introduction
- Design Principles
- Objects
- Filesystems
 - FAT16, FAT32, NTFS
 - NTFS: Recovery, Fault Tolerance, Other Features
- Summary

FILE SYSTEMS: FAT32

- Obvious extetension: instead of using 2 bytes per entry, so can support e.g. 8Gb partition with 4kB 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 4kB default cluster size)
 - VFAT on top of FAT32 does long name support: unicode strings of up to 256 characters
- Want to keep same directory entry structure for compatibility with, e.g., DOS so use multiple directory entries to contain successive parts of name
 - Abuse V attribute to avoid listing these

Still pretty primitive...

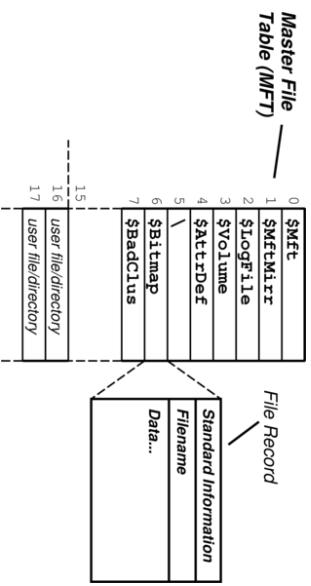
6.1

FILESYSTEMS: NTFS

NTFS FORMAT

NTFS uses clusters as the underlying unit of disk allocation:

- A cluster is a number of disk sectors that is a power of two
- Because the cluster size is smaller than for the 16-bit FAT file system, the amount of internal fragmentation is reduced
- NTFS uses logical cluster numbers (LCNs) as disk addresses
- The NTFS name space is organized by a hierarchy of directories; the index root contains the top level of the B+ tree



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

6.4

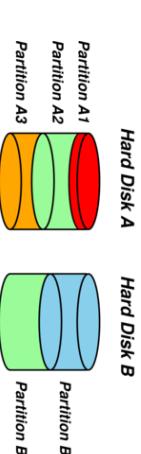
NTFS: RECOVERY

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):
- NT has a generic log file service that could be used by e.g. databases
 - Makes for far quicker recovery after crash
 - Modern Unix filesystems eg., ext.3, xfs use a similar scheme

6.5

NTFS: FAULT TOLERANCE



FTDisk driver allows multiple partitions be combined into a logical volume:

- Logically concatenate multiple disks to form a large logical volume, a volume set
- Based on the concept of RAID = Redundant Array of Inexpensive Disks
 - E.g., RAID level 0: interleave multiple partitions round-robin to form a stripe set
 - 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 s/w cluster remapping

6.7

NTFS: OTHER FEATURES (I)

Security

- Security derived from the NT object model
- Each file object has a security descriptor attribute stored in its MFT record
- This attribute contains the access token of the owner of the file plus an access control list

Compression

- NTFS can divide a file's data into compression units (blocks of 16 contiguous clusters) and supports sparse files
 - Clusters with all zeros not allocated or stored
 - 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

6.8

NTFS: OTHER FEATURES (I)

Encryption

- Use symmetric key to encrypt files; file attribute holds this key encrypted with user public key
- Problems:
 - Private key pretty easy to obtain; and
 - Administrator can bypass entire thing anyhow

6.9

SUMMARY

- Introduction
- Design Principles
- Design
- Objects
- Filesystems
- **Summary**

SUMMARY

Main Windows NT features are:

- Layered/modular architecture
- Generic use of objects throughout
- Multi-threaded processes & multiprocessor support
- Asynchronous IO 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
- Continues to evolve: Singularity, Drawbridge, Windows 10, ...

7.1

SUMMARY

- Introduction
- Design Principles
- Design
 - Structural
 - HAL, Kernel
- Processes and Threads, Scheduling
- Environmental Subsystems
- Objects
 - Manager, Namespace
 - Other Managers: Process, VM, Security Reference, IO, Cache
- Filesystems
 - FAT16, FAT32, NTFS
 - NTFS: Recovery, Fault Tolerance, Other Features
- Summary