Systems Architecture 2
CM10195

Russell Bradford (Operating Systems)
Alan Hayes (Networking)

2021

CM10195

This unit follows on directly from Systems Architecture 1 (CM10194)

These units are to provide the material that everybody who might like to think they know about computer science should be able to recite in their sleep

Some of you may already know some of the material in this unit

- You are lucky, but don't assume you know it all!

For most of you this is new

- You are lucky because it's all good stuff!

Unit Outline

There are two major chunks of material:

1. Introduction to Operating Systems: taught by me
2. Introduction to Computer Networks: taught by Alan Hayes

Both are things that if your computer is working properly, you shouldn't notice them at all!

> *"Operating systems are like underwear - Nobody really wants to look at them"*
> *Bill Joy*

Unit Outline

Structure of this unit:

- Pre-recorded “lectures”, released week-by-week on Re:View/Panopto. The main delivery of the content of this unit. These will be in several chunks each week: they will vary in number and length as is appropriate for the topic being discussed, but generally there will be several short videos (instead of fewer long ones)
- These will be uploaded Monday mornings

Unit Outline

- Live Interactive Online sessions:
  - Wednesdays 12:15
  - Thursdays 15:15

  On Zoom. See Moodle for links. These sessions will be reactive (“make it up as we go along”) for things like questions, supplementary discussions, coursework, past papers, or anything (relevant) you want.

Unit Outline

These sessions will be recorded (to Re:View/Panopto) for the benefit of those people who can't make the timeslot

If you have a personal issue with being recorded, make sure your camera and microphone are off. You can still interact via chat

At first, we shall only be using the Wednesdays session as there won't be so much to talk about and to manage our workload/studyload. We shall use the Thursdays as and when they are needed

Unit Outline

Assessment

Usual combination of assessed coursework and exam

1. Essay on Operating System issues (15%)
2. Networking Assignment (15%)
3. End of unit exam (70%)

Unit Outline

Assessment

Coursework timelines (approximately, subject to change):

1. set Wed 24 Feb, due Fri 19 Mar
2. set Wed 24 Mar, due Fri 23 Apr

Feedback on coursework will be provided via Moodle.

The week starting 8th March is consolidation week: no new material

Easter break: two weeks starting 29th March

Unit Outline

Operating Systems

Outline content:

1. Introduction: What they are and what they do; history
2. Processes
3. Memory and memory management
4. Files and filesystems
5. (Peripherals and I/O)

Unit Outline

Networks

1. Introduction: What they are and what they need to do; history
2. Layering models
3. Addresses and names
4. Services (DNS, LDAP, SSL)
5. Application abstractions: data services and web services

Unit Outline

Resources

The subject of Operating Systems is nearly as old as that of computers and so there are *lots* of books

Unit Outline

Resources

Some books I found on my shelf:

- “Operating Systems Internal and Design Principles” W Stallings, Prentice Hall
- “Computer Systems Architecture A Networking Approach” R Williams, Addison-Wesley
- “Introduction to Operating Systems Behind the Desktop” J English, Palgrave
- “Operating Systems a Concept-Based Approach” D M Dhamdhere, McGraw Hill
- “Operating Systems Concepts with Java” A Silbershatz et al, Wiley

Unit Outline

Resources

N.B. These were given to me by the publishers so I'm not saying they are the best books out there

The thing to do is look at several and find one that suits you: they all contain roughly the same material

Unit Outline

Resources

Networking books

- “TCP/IP Illustrated Volume 1” W R Stevens, Addison-Wesley
- “Computer Networks, 4th Ed” A Tanenbaum, Pearson
- “The Art of Computer Networking” R Bradford, Pearson (Polish Edition: “Podstawy Sieci Komputerowych”, WKŁ)

These are definitely all good books!

Unit Outline

Resources

You don't need me to tell you that there is a large amount of material out there on the Web?

Wikipedia is fairly accurate in this area: but, as usual with Wikipedia, you should follow up the references and check with other sources

Unit Web page: `http://people.bath.ac.uk/masrjb/CourseNotes/cm10195.html` (link on Moodle)

Unit Outline

Resources

Contacting me:

If you have a question on the unit, please consider bringing it along to the LOIL so that everyone can get the benefit

Otherwise, email me — I don't monitor all the dozens of other ways of messaging (Moodle, Teams, etc.) and email is the only way to be sure of getting a message to me

I keep a 9-5 (approx) Monday–Friday week and am unlikely to respond out of those times (a long time a ago someone said "Get a life", so I did)

Standard Introductory Slides

Remember:

You are expected to do some work outside of lectures

Lectures are the *start* of the learning process, not the end!

These slides are reminders to me on what to say in lectures

They are often abbreviated in style, and so are not the whole story and would not be suitable to be quoted verbatim in an exam

Standard Introductory Slides

Do not rely purely on my notes for your revision

People who do this live to regret it

Like every Unit, you are expected to read around the subject for yourself

You need to take your own notes, read, and *participate*

You don't expect to get fit simply by paying to joining a gym...

“If you have college courses in CS, buy the books and spend day and night the few days before class going through the books and taking notes and answering questions and programming examples before the first class even starts. If you really want to do this in your life, that’s what you should do, not just wait for the education to be handed you. Those who finish at the top will always be in high demand. You can learn outside of school too but you have to put a lot of time into it. It doesn’t come easily. Small steps, each improving on the other, is what to expect, not instant understanding and expertise.”

Steve Wozniak, co-founder of Apple

Standard Introductory Slides

Computer Science is not a spectator sport

Anon

□

Operating Systems: Introduction

An Operating System (OS) is just a program, often called the *kernel* or *monitor*

Its purpose is to

- Manage the resources of the computer
- Provide the applications programmer (N.B., *not* the end user of applications) with a usable programming interface to access those resources

The interface that the end user interacts with is *not* part of the OS

It's just another program that uses the OS

If the end-user sees it, it's not part of this course!

Resources

So what are resources?

- Hardware: cpu, memory, disk, network, sound, video, keyboard, mouse, printer, camera, ...
- Software: anything that controls the above, though use of the cpu is a primary focus

Resources

Why do they need managing?

1. They are limited with not enough to go round
2. They need protection
3. They need to meet certain criteria

Resources

Limited?

Surely computers are so big and fast these days there is no scarcity on resources?

Actually, most computers are small and very limited!

E.g., mobile phones have strict limitations on memory, cpu power and *energy consumption*

Even big computers are not yet big enough for many people
My laptop is currently running about 370 programs

Resources

Why do they need managing?

1. They are limited with not enough to go round
2. They need protection
3. They need to meet certain criteria

Resources

Protection

Protection comes in many forms

- Preventing one program from accidentally (or intentionally) corrupting another program or data on the same or another machine: security
- Ensuring certain resources are only available to those programs that are allowed: authorisation
- Ensuring that a given program has the authorisation it claims to have: authentication
- Protecting you from your own stupid mistakes (Did you really want to delete that?)

Resources

Why do they need managing?

1. They are limited with not enough to go round
2. They need protection
3. They need to meet certain criteria

Resources

Criteria

Popular criteria include

- Responsiveness: making a program respond snappily or processing network packets as they arrive
- Real Time: certain events *must* be dealt with in a (small) fixed amount of time, e.g., the controlling flaps on an airplane's wing, video streaming
- Security: prevention of accidental or malicious access or modification

Programming Interface

The other purpose of an OS is to provide an interface for the programmer:

The programmer who has to write applications for the machine does not want to have to know the details of the hardware: think portability (c.f. von Neumann's model)

- How do I get the best performance out of this disk, network, video?
- How do I prod this hardware to get it to do what I want?
- How should I deal with interrupts?
- And so on

Programming Interface

We don't want to have to re-implement everything in every program

So the OS does this kind of thing for us

Early programmers, before OSs, had to do it all themselves

Much better to let someone else do the hard work
(A common theme in Computer Science)

Programming Interface

Having an expert do this stuff once and provide a standard interface to it for us to use is much better for us

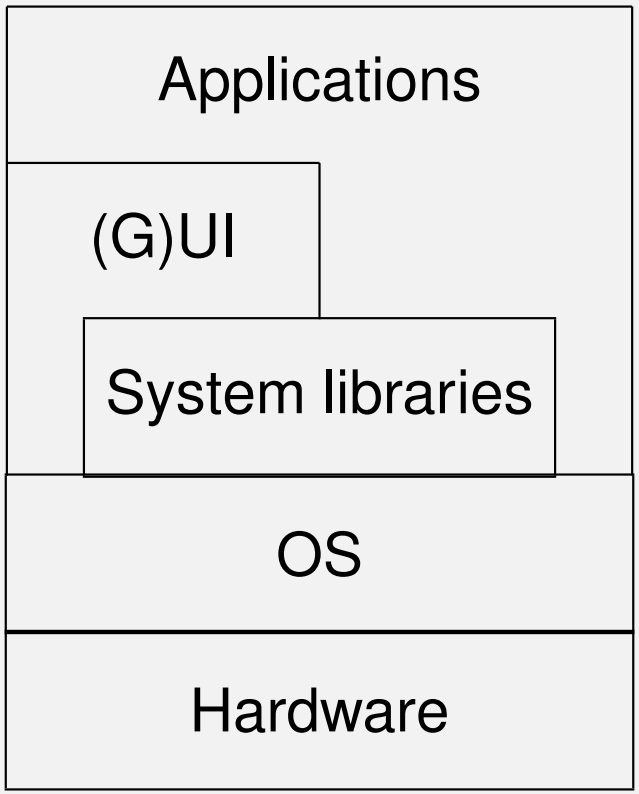
- We don't have do it
- The expert is better at it and (presumably) understands the hardware well
- The expert is a better programmer than us and can get better performance out of the hardware
- The programmer knows more Computer Science than us and knows the many pitfalls and necessary tricks that OS programming involves

Programming Interface

They do it so we don’t have to

Programming Interface

Layer Abstraction:



Browser, word processor, game

Command line, windowing, touch

Maths, graphics, sound

Linux, OS X, Windows, Android

PC, phone, PVR, SatNav

Important Point

Reemphasising a very important point:

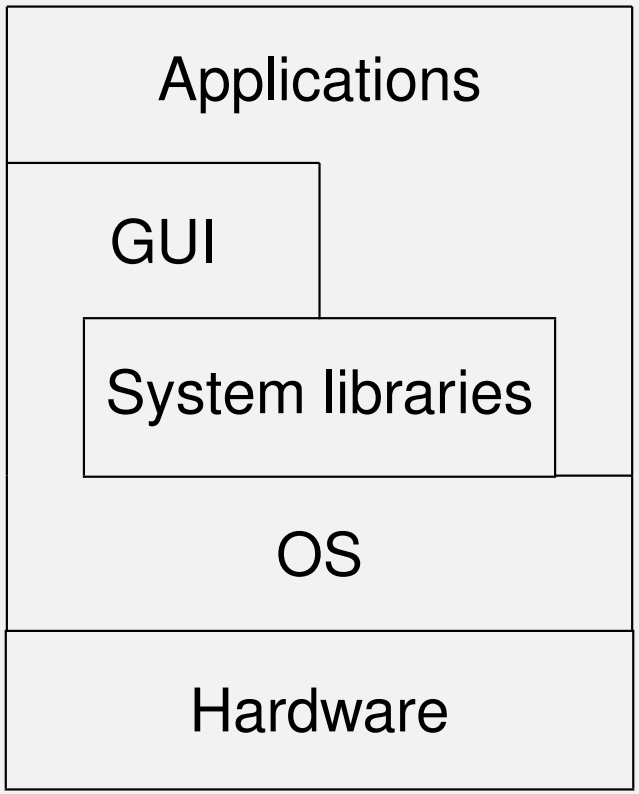
The GUI is not part of the OS

The GUI is just another program that uses the OS

There was a time when certain OS vendors tried to tie the GUI into the OS (to gain speed and commercial advantage)

Programming Interface

Bad Layer Abstraction:



Browser, word processor, game

Windows

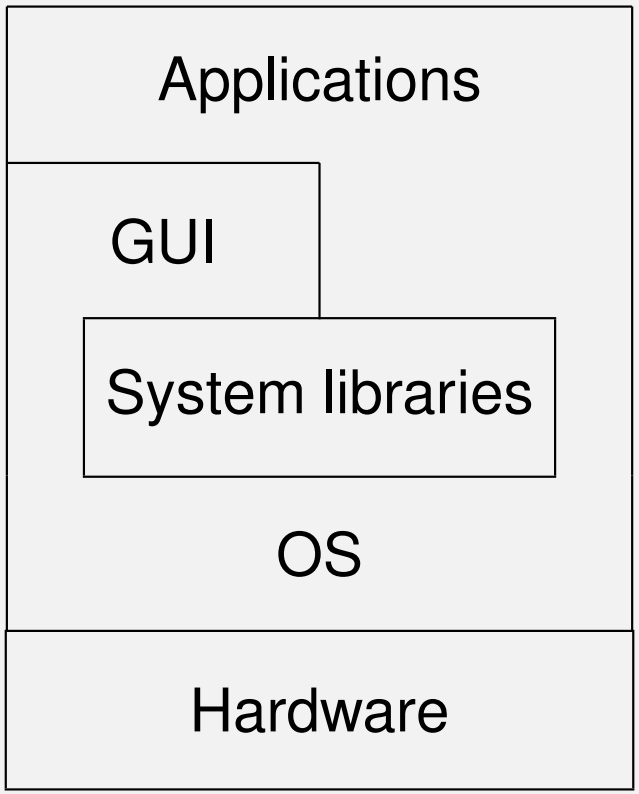
Maths, graphics, sound

Windows

PC

Programming Interface

Very Bad Layer Abstraction:



Browser, word processor, game

Windows

Maths, graphics, sound

Windows

PC

Programming Interface

This is poor design and should be avoided

It was a huge source of problems: bugs in the GUI or the application would cause the OS to crash, so a poorly written program could take out the whole machine

It was an easy way to circumvent the security the OS provides, thus allowing attackers to access the machine

Another Important Point

Only a small number of CPU cycles in the world deal with GUIs or windows: the PC is far from being the most common type of computer

Embedded systems out-sell PCs by orders of magnitude

This Unit has nothing to do with GUIs

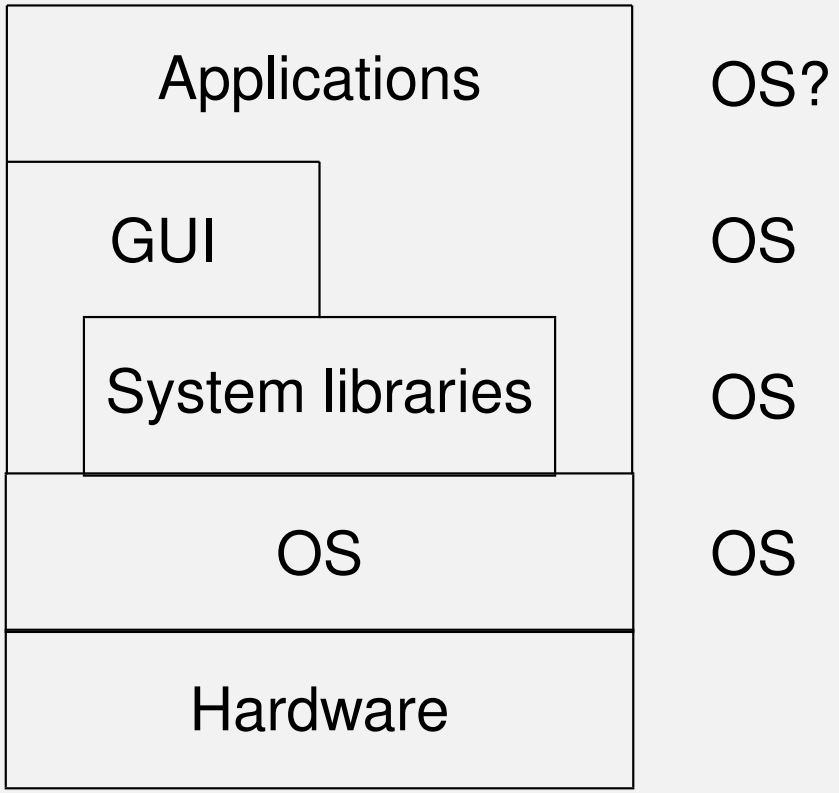
You will lose marks in this Unit if you start talking about GUIs as it means you don't understand the concept of an OS

It quite possible to have similar-looking GUIs running on different OSs

But we must be careful as some people don't realise the difference between an OS and everything else

Programming Interface

CS view:Marketer's view:



One more thing

There are a couple of vital aspects of an OS that are sometimes overlooked:

1. It should be efficient and lightweight: every CPU cycle that the OS uses is one that is taken away from the user's programs
2. It should be flexible and not get in the way of the programmer

So a perfect OS would be completely invisible!

Background

Operating Systems have been around nearly as long as computers

. . . but not quite as long

So long, though, that some (most!) people don't distinguish between computers and their OS

Computer Scientists ought to be more careful

Background

An OS is just a program so

- we can have different OSs on the same hardware
- we can have the same OS on different hardware

- On Intel hardware (“PC”) you can run Windows, MacOS, Linux (and lots of others)
- Current Mac hardware is the same as PC hardware, so you can run all of the above on a Mac (the new Mac M1 is effectively only slightly different)
- Linux runs on many kinds of hardware, including Intel, mainframes (IBM, Oracle/Sun), million processor clusters, phones (ARM), gadgets like satnavs and PVRs

N.B. when we say “Intel” or “x86” we mean Intel, AMD and all the compatible architectures

Background

A note on portability of OSs

Since an OS has to deal with the details of hardware, it is not entirely trivial to port an OS from one CPU architecture to another, e.g,. an Intel CPU to an ARM CPU

Well designed OSs minimise the hardware-dependent parts and try to keep most code hardware-independent

Most of the difficult code in the OS can be kept hardware independent, only the stuff that needs the actual physical access need be hardware dependent

Exercise. Have a browse through the Linux kernel source code

Background

Historically, some OSs tied themselves so closely to the hardware that porting is very difficult

For much of its lifetime Windows only ran on Intel, so the assumption of Intel hardware ran throughout the code

Microsoft have recently ported Windows to ARM chips

They are frantically trying to catch up with a market that is moving to phone/tablet and energy-conscious computing

And they paid the price of not abstracting away the hardware from the software

And note this will require rewrites of Windows applications, too

Background

In contrast, the Android OS that runs on phones shares a large amount of code with the Linux that runs everywhere else, even on supercomputers

The iPhoneOS that runs on iPhones shares code with OS X

Windows Phone 7 shares little with Windows 7 (though the phone GUI, was adapted to the PC and Windows Phone 8 shares with Windows 8)

Phone OSs are marketed strongly on their GUIs, not the actual OS; many phone vendors try to make it hard for users to access the OS

Background

For the new M1 Macs their operating system had to be ported from x86 to ARM

And Apple provides Rosetta, a clever bit of software that dynamically translates x86 code to ARM code so that user applications written for the old hardware will still run on the new hardware

With a moderate loss in speed

Background

You will often hear “Do you have a PC?” where the asker really means “Do you run Windows on your PC?”

“PC” means “Personal Computer” and refers to the hardware

I run Linux on my PC; many people run MacOS on their PCs

Such people get confused when I answer “Yes, I have a PC but it doesn’t run Windows”

“Or MacOS”

Macs are PCs, too

Background

The understanding of the general public is such that they get the hardware and software confused

And, more, they are usually thinking about the GUI, not the OS

Some software companies encourage and make capital out of this confusion

Background

There is some reason for this confusion: Microsoft and Apple tie their GUIs indivisibly to their OSs

Other OSs, notably Linux, allow the user a choice of many UIs and GUIs all running on the same OS kernel

And typical users base their choices on GUIs, not OSs

But we shouldn't be talking about GUIs: this Unit is about Operating Systems, taking the programmer's point of view

Background

As programmers, we should choose an OS for the features that it provides:

- Ease of use
- Efficiency
- Security
- Stability
- Suitability for the task in hand
- And so on

Background

Just a few recent operating systems:

- XP/Vista/Windows 7/Windows 8/Windows 10 from Microsoft. Large, resource intensive, highly featured, previously Intel processor only, now Intel and ARM
- OS X from Apple. Large, not quite so intensive, highly featured. Based on BSD (Unix), on Intel and ARM processors (Earlier: PowerPC)
- Unixes. Solaris from Sun/Oracle, IRIX from SGI, AIX from IBM, OSF/1 from DEC, etc. Large to medium.
- Unix derivatives (reimplementations). Various BSD (including MacOS X), Linux, Hurd, etc. From small to large.

Note, since the advent of smartphones, of the above OSs, Unix derivatives (Linux) are the most popular

Background

- Phones. Palm OS, Symbian, Windows CE/Mobile/Phone 7/Phone 8, Android, iPhone OS
- Experimental. Minix, Plan 9, Mach, Singularity, Amoeba, etc.
- Networking. NetWare (Novell), Cisco IOS, DD-WRT etc. For controlling networking hardware
- Distributed OSs. Management of collections of computers, or making a collection appear as a single large computer

Background

- Embedded. μCLinux, Windows Embedded, RTOS, etc. Small, resource frugal
- Real-time. QNX, μCLinux, etc. For controlling systems where a fast response is critical
- OSs for gadgets (MP3 players, etc.) Conservation of battery power is the largest problem
- Other. OS/2, MacOS 9, RISC OS, BeOS, z/OS (IBM). Various sizes

Again, remember that embedded OSs outnumber PC OSs by an order of magnitude

And we've not even mentioned historical OSs yet

Background

There are *lots* of operating systems out there, most we don't notice

The ones we do notice are failing in their purpose!

ARX project Arthur OS RISC OS AmigaOS Amiga Unix AEGIS Domain/OS vikek OS Apple DOS UCSD Pascal ProDOS GS/OS SOS Lisa OS Newton OS Mac OS 8 Mac OS 9 A/UX MkLinux Mac OS X v10.x iOS Atari DOS Atari TOS Atari MultiTOS XTS-400 BeOS Blue Eyed OS Cosmoe GCOS Burroughs MCP COS SIPROS SCOPE MACE KRONOS NOS NOS/BE RDOS AOS DG/UX CTOS DOS Deos HeartOS CP/M DR-DOS OS/8 ITS TOPS-10 WAITS TENEX TOPS-20 RSTS/E RSX-11 RT-11 VMS Domain/OS TSB Digital UNIX HP-UX Ultrix Guardian OSS OSE Towns OS Google Chrome UTX-32 INTEGRITY HDOS HT-11 HP-UX HP MIE OLERT-E Multics HeartOS DEOS iRMX ISIS-II BESYS CTSS GM OS GM-NAA I/O IBSYS IJMON SOS UMES OS/360 OS/VS SVS OS/VSn MVS/SE OS/390 z/OS DOS/360 z/VSE CP/CMS VM/370 VM/XA VM/ESA z/VM AIX/370 OpenSolaris UTS z/Linux BOS/360 MTS MUSIC/SP ORVY WYLBUR PC DOS/IBM DOS OS/2 J MultiJob GEORGE 2/3/4 TME ICL VME iVideOS LynxOS MicroC/OS-II Xenix MS-DOS Windows Singularity Midori TMX NetWare MontaVista RTXC

History

We are now going to look at some history: this is useful as it will illustrate the many important parts of OSs and why they are necessary

At first, computers had no operating systems (1960s)

- Every programmer had to write their programs for the particular machine they were using
- So no portability
- And lots of repeated code between programs (“write a character to the teletype”)

Remember: the more we make programmers do, the more likely they are going to make a mistake

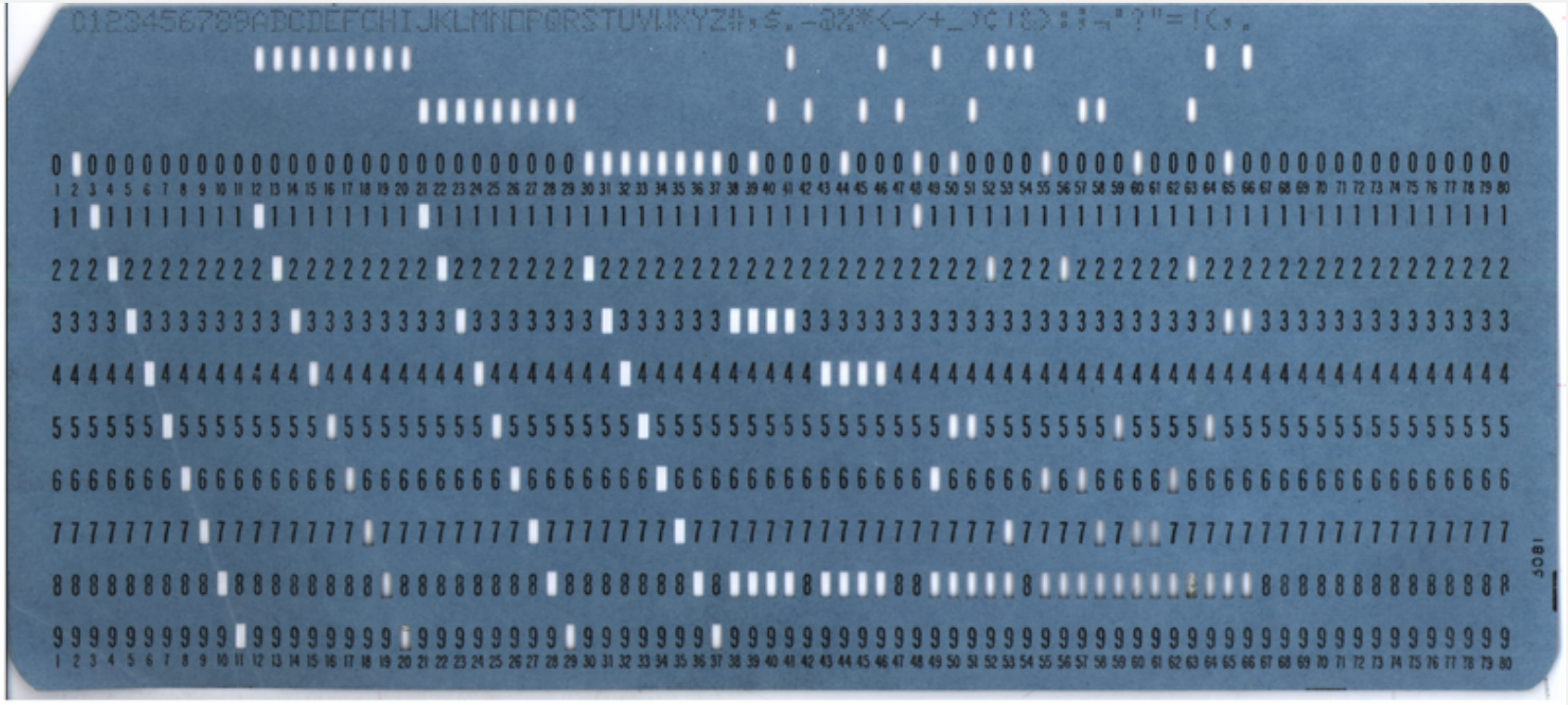
History

Furthermore, programmers rarely even saw the computer

- The program and the data (collectively called a *job*) would be prepared on paper tape or punched card
- Jobs would be given to operators who would the load and run them, then give the results back
- Usually, there would be a bug and the programmer would have to fix the program and go round the loop again
- As computer time was limited, there was an issue of *scheduling* jobs, initially done by hand
- Turnaround on jobs could be days

This concentrated the programmer's mind wonderfully!

History



From Wikipedia. Encodes a single 80 character line

History

From Wikipedia. 5 and 8 hole paper tapes

History

It was soon found there was a lot of repeated code between programs, so useful tools (programs and libraries of code) were developed to help manage repetitive tasks

- collecting common functions in *system libraries* (sqrt, open file, etc.)
- program management (loaders)
- debuggers
- Interfacing to hardware: I/O drivers (send file to printer, etc.)

This made programming and program management easier, but there was still lots of human intervention needed

History

The issue was to keep the big and expensive computer as busy as possible, running programs all the time

Idle time was a waste of money

So the operators would load many programs on to a fast medium, such as magnetic tape, and the computer would load and run them as fast as hardware allowed

This was called *spooling*, the first instance of addressing the disparity between human and computer speeds

History

Spooling would also be used on output: the output would be written to a mag tape, which could then later be attached to a printer

Again, this was because printers are slower than computers

History

Soon it became clear this could be automated: have a little program, called a *monitor* (or *supervisor*), that loads and runs programs and puts the results somewhere sensible

This would be directed by a *job control language*

History

A famous job control language from IBM was called JCL

Of course “JCL” means “Job Control Language”, but JCL was just one of a few job control languages

History

```
//IS198CPY JOB (IS198T30500),'COPY JOB',CLASS=L,MSGCLASS=X
//COPY01   EXEC PGM=IEBGENER
//SYSPRINT DD  SYSOUT=*
//SYSUT1   DD  DSN=OLDFILE,DISP=SHR
//SYSUT2   DD  DSN=NEWFILE,
//           DISP=(NEW,CATLG,DELETE),
//           SPACE=(CYL,(40,5),RLSE),
//           DCB=(LRECL=115,BLKSIZE=1150)
//SYSIN    DD  DUMMY
```

(From Wikipedia) Any guesses?

This copies OLDFILE to NEWFILE
This would be set on 9 punched cards

History

A Fortran program, with data:

```
//CONVERT JOB USER=UGA001,MSGCLASS=6,NOTIFY=UGA001
//*MAIN CLASS=NITE,LINES=40,ORG=UGAIBM1.LOCAL
// EXEC FORTVCLG,REGION=2000K
//FORT.SYSIN DD *
       READ(5,10) CENT
    10 FORMAT(F6.2)
       FAHR=(CENT*9.0/5.0)+32.0
       WRITE(6,20) CENT,FAHR
    20 FORMAT(F6.2,' CENT = ',F6,2,'FAHR')
       STOP
       END
/*
//GO.SYSIN DD *
100.00
/*
//
```

History

JCL also allowed

- different *classes* of job: some people are allowed more time or memory space than others
- automatic *accounting*: who to charge for what. Charges would be made for CPU time, memory usage and anything else they could think of (another recurrent issue in CS)
- programmers could specify things like *when* they want the program to run, how much disk or memory it needs, etc. E.g., at certain times of day it might be cheaper to run a program

If a job ran out of its allotted time or space it would be killed

History

JCL allowed several programs to be collected and loaded together in a single bunch

This is called *batch processing*

Running in batches is more efficient, as we spend more time running our programs and less time messing around in the overheads of loading and unloading

History

This might seem like ancient history, but these things are still happening

Modern large computers (like Bath's Balena cluster) are managed in just this way: and for the same reasons

We still run jobs; charges are made for time and memory; and so on

Turnaround is seconds or minutes rather than days, but the principle is the same

Exercise: look up *Portable Batch System*, PBS and compare with JCL

History

So the monitor was just a program

It would load an application into memory (from tape or wherever)

And then jump to the start of the application and start executing it

When the application finished, it would (be expected to) jump back to the monitor, so it could deal with the next program

History

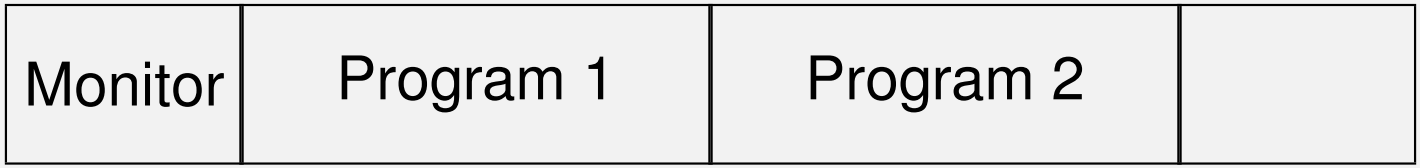
But if the application was badly written, it could overwrite the monitor

Machine memory

Either accidentally or deliberately

History

It was soon found to be more efficient to load more than one program into memory (when there was space)



The advantage being that if Program 1 was doing something like writing to a tape that takes a lot of time, but no CPU, the computer could run Program 2 in the meanwhile

When Program 2 pauses and Program 1 needs to run again, the computer could switch back to it

The decisions on what to run and actually doing the switching between programs was the job of the monitor

History

Now Program 1 could corrupt Program 2 as well as the monitor!

Or Program 1 could read confidential data out of Program 2

Some sort of protection of the monitor and other programs is needed

What happens if Program 1 goes into an infinite loop?

Control never returns to the monitor and Program 2 never gets to run

Some means of curtailing runaway programs is needed

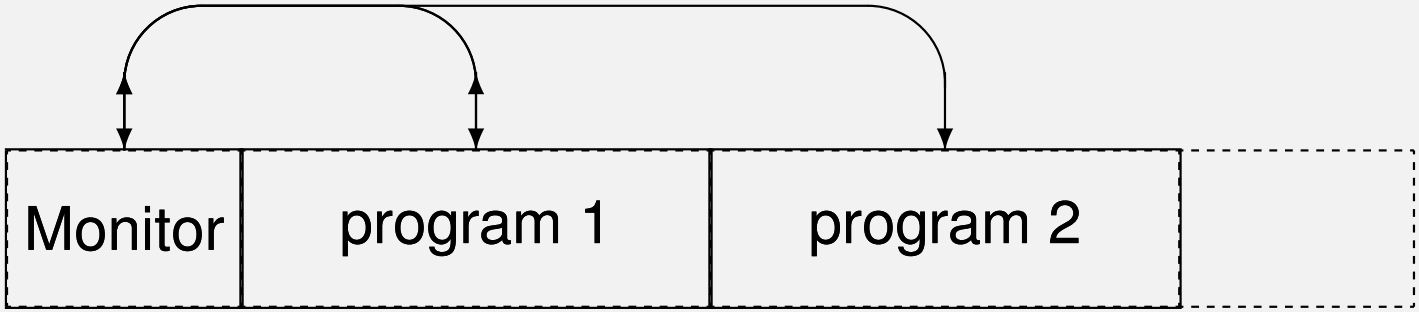
History

So the “monitor runs the programs”: what does this really mean?

Nothing sophisticated. The monitor code just jumps to the program code so the machine is now running the program

Take care over this point: the monitor doesn’t sit and watch the program running, the monitor is *not* running while the program is running

History



Monitor runs Monitor jumps to program 1 Program runs Tape needed Program calls monitor Monitor sets up tape Monitor decides to run another program while waiting for the tape Monitor jumps to program 2 Program 2 runs Etc.

History

Less graphically:

Monitor starts program $\rightarrow$ Prog 1 it runs... tape needed $\rightarrow$ Monitor sets up tape... Monitor decides to run another program $\rightarrow$ Prog 2 etc.

There is a *single stream of control* jumping between monitor and several programs

The monitor is not running when a user program is running, and vice-versa

History

This changing between multiple user programs is called *multitasking*. But only one thing is ever running

Multitasking improves the efficiency of use of a computer since while one program waits for a slow peripheral another program can run

History

So the monitor needs to make decisions on what to program to run next: it is *scheduling* the programs

The choices can be made according to many criteria

- how long a program has been running
- a *priority* of a program
- whether a program is likely to need CPU very soon, or can wait
- how much the owner of the program has paid
- And many more things

History

Early scheduling algorithms were very simple, e.g., keep running the same program until it's done; later algorithms tried to be more clever

Some programmers would write their programs to take advantage of deficiencies in the scheduling algorithm: in the worst case *starve* other programs of any CPU time at all!

It is tempting to make the scheduling algorithm complicated: but remember more time spent in the monitor deciding what to schedule next is less time for the programs

So there is a trade-off of making scheduling *fast* but *fair*

This is still an issue today: we'll look a little into scheduling later

History

A badly written (or malicious) program can bring the whole system down

If the program never hands control back to the OS (we'll call the monitor the operating system from now on), the OS never gets to run and schedule another program

If the program goes into an infinite loop the whole computer is jammed

This *cooperative* approach needs something extra

History

Interrupts can be used to solve the problem of runaway programs

A hardware clock or *timer* can be set to send interrupts regularly after an appropriate period of time has elapsed

When the interrupt is taken, the interrupt service routine jumps to the OS and so it can decide what to do next, including:

- resume running the interrupted program
- kill (no longer run and remove resources from) the program if it has used up its allotted resources
- switch to running some other program

Similarly, interrupts from peripherals like terminals or disks pass control to the OS

History

This is called *preemptive* scheduling and enables *timesharing*

Timesharing is where several programs share the available CPU time and so appear to be running simultaneously

Usually in a fairly transparent (to the programs) manner

Always mediated by the OS, of course

History

The same interrupt mechanism allowed the use of *terminals*, where users could now interact directly with the computer, not just via job submission

A program can sit and wait (i.e., not be scheduled to run by the OS) until the user hits a key on the terminal

When a key is hit, an interrupt happens, the OS takes over, schedules and runs the appropriate program to deal with the keystroke

History

Thus the waiting program uses no CPU resources until they are needed

Of course, while we say “the program is waiting”, is important to realised that it’s not “waiting”: the program is not even running

So interrupts like this are another way of bridging the gap between slow humans and fast computers

History

Typically, timer interrupts are set to go off fairly often

- Frequent interrupts mean several programs can get a slice of the CPU quite often
- With sufficiently frequent interrupts it appears to a human observer that several programs are running simultaneously
- An *interactive* program, one where a human is involved, will appear to be dedicated to that user: in reality humans are so slow we can't appreciate how little time the computer gives us
- It is important to remember that a single processor can only do one thing at a time: it is only the *appearance* of multiple programs running simultaneously

History

On the other hand, too frequent interrupts mean the OS is forever being called and using CPU time, so less time is available for the programs

This is another tradeoff: frequent interrupts for good interactive behaviour, rare interrupts for good compute behaviour

Clever scheduling algorithms in the OS try to give high priority but small slices of time to interactive programs; and lower priority but larger slices to compute-intensive programs

A “large slice of time” means the OS will allow a program to continue running for a relatively long amount of time before scheduling a different program

History

A “small slice of time” means the OS will deschedule the program after only a brief amount of running time

Thus, the OS can deal out CPU time to the programs in appropriately sized chunks

This is all part of the scheduling decision computations that happen potentially every time the OS runs

My PC is running at about 150 interrupts per second (timers and other stuff)

History

Low power gadgets like to keep the number of interrupts down, too, as it increases the amount of time the CPU can be idling in low power *sleep states*

Tuning an OS is very difficult and depends critically on the application

When an OS spends more time deciding what to do than doing useful work, it is called *thrashing*

Many early OSs had a big problem with thrashing

Question

Exercise. To think on: should the OS be subject to timer interrupts and preemption?

□

History

The programs and OS all live in the same computer memory: we need some way of protecting programs and the OS from each other

This has to be done by hardware support as it needs to be fast and unobtrusive: potentially every memory access needs to be checked

We shall start by looking at general hardware protection mechanisms

History

Certain operations, like accessing tape or a printer, must be reserved for use by the OS and not be accessible by a random user program

So in the hardware (CPU) machine instructions are divided into two (or more) classes

- Unprivileged operations. Like addition, jumps. Any program can execute these
- Privileged operations. Like access peripherals, reboot the machine. Only certain privileged programs can run these

And the processor can run in two (or more) modes

- Unprivileged. Normal computation, called *user mode*
- Privileged. For systems operation, called *kernel mode*

History

Modern processor architectures can have four or more levels of privilege, but for the most part it is rare that more than two levels are used in commodity computers

For example, the Intel x86 architecture has four *rings*. Ring 0 can execute any instruction, while Ring 3 is for user mode. Rings 1 and 2 are rarely used these days

OS/2 used Ring 2

The latest Intel and AMD architectures added a Ring -1 (for OS virtualisation)

History

Note that privilege is a state of the *processor*, not the program, but we tend to say “a privileged program” rather than “a program running with the CPU in privileged mode”

If an unprivileged program (i.e., a program running in an unprivileged mode) tries to execute a privileged operation the hardware causes an interrupt (also called a system *trap*) and sets the processor to privileged mode. The interrupt service routine then jumps to the OS

The OS can then decide what to do

For example, the OS may decide to disallow the operation, and kill the program (i.e., not run it any more)

History

The system starts in kernel (privileged) mode

1. The OS decides which process to schedule
2. It uses a special jump-and-drop-privilege instruction to start running the program
3. The program runs user mode (unprivileged)
4. The program finishes or decides it needs a system resource
5. The program executes a special “call OS” (or *syscall*) instruction that jumps to the OS
6. This enables privileged mode, so the OS regains control, with privilege
7. The OS decides what to do next

Of course, even if the program does not do a syscall, a timer interrupt will come along at some point, anyway

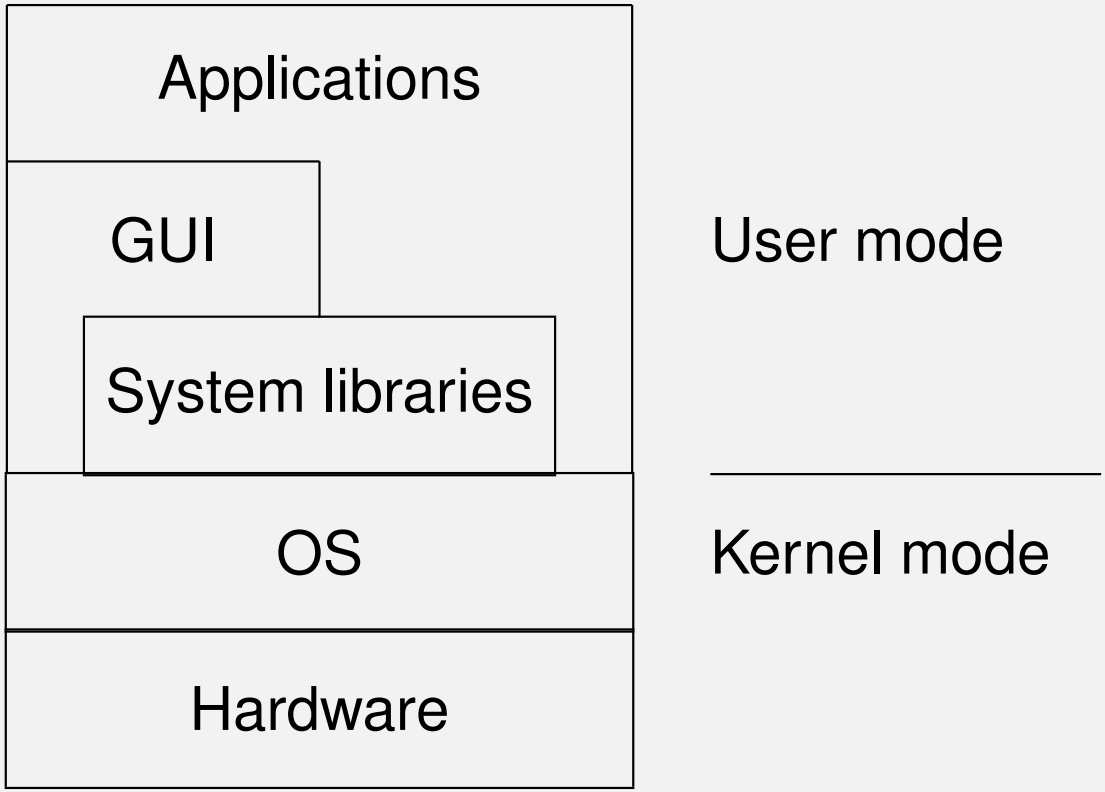
History

The syscall instruction always jumps to the same place in the OS. So the program cannot use it to gain privilege for itself and run its own code privileged

This to-ing and fro-ing between modes ensures that the OS is running in privileged mode and the user program is running in unprivileged mode

And the user program can never manage to get into privileged mode as every transition to privileged mode is tied by the hardware to a jump to the OS

History



There is a strict divide between kernel (OS) code and user code, controlled by the hardware

History

Unless there are bugs in the kernel code. . .

Incidentally, the system libraries usually include a bunch of "nice" interfaces to the syscalls: wrapping them to make using them easier

E.g., the "open file" syscall might need certain values (file name, etc.) to be placed in certain CPU registers; and the "open file" opcode to be placed in a register before the syscall

The `open` system library function simply hides these details from the programmer

History

Warning!

Switching back and forth between OS and programs is, in many operating systems, a relatively time-consuming operation, due to overheads that should become clear later

For now, just think of the overheads of saving and restoring the CPU state of the running program, just as for an interrupt

These overheads are another reason why you don't want timer interrupts too often

History

The result of all this messing with modes is certain operations like loading programs, or accessing hardware like a printer, are only available to the OS

If an unprivileged program tries to access the printer directly, that again trips an interrupt and the OS takes over anyway

Forcing access to hardware via the OS also provides protection and management for other system resources, like access to files or the network

In kernel mode, everything is possible

In user mode, only “safe” things are possible

History

Preemption and protection appeared in OSs for large mainframe computers and Unix for minicomputers in the late 1960s

When microcomputers (IBM PC) arrived in the early 1980s much of OS knowledge was thrown away and DOS (Disk Operating System) was non-preemptive, single process and no protection

This was because the earliest PC hardware did not support such things (no rings)

History

Support was rapidly added in later PC hardware, but DOS and, later, Windows 3.1 took no advantage of it: the lack of protection meaning a single bad program could mess up the OS and crash the entire computer

Windows NT was the first true OS from Microsoft (mid 1990s) for PCs, possibly as much as a decade after other OSs (such as Unix derivatives) were providing preemption and protection on the same hardware

Incidentally, Microsoft's need for backwards compatability with these early systems is a major reason why they have so many problems with security

History

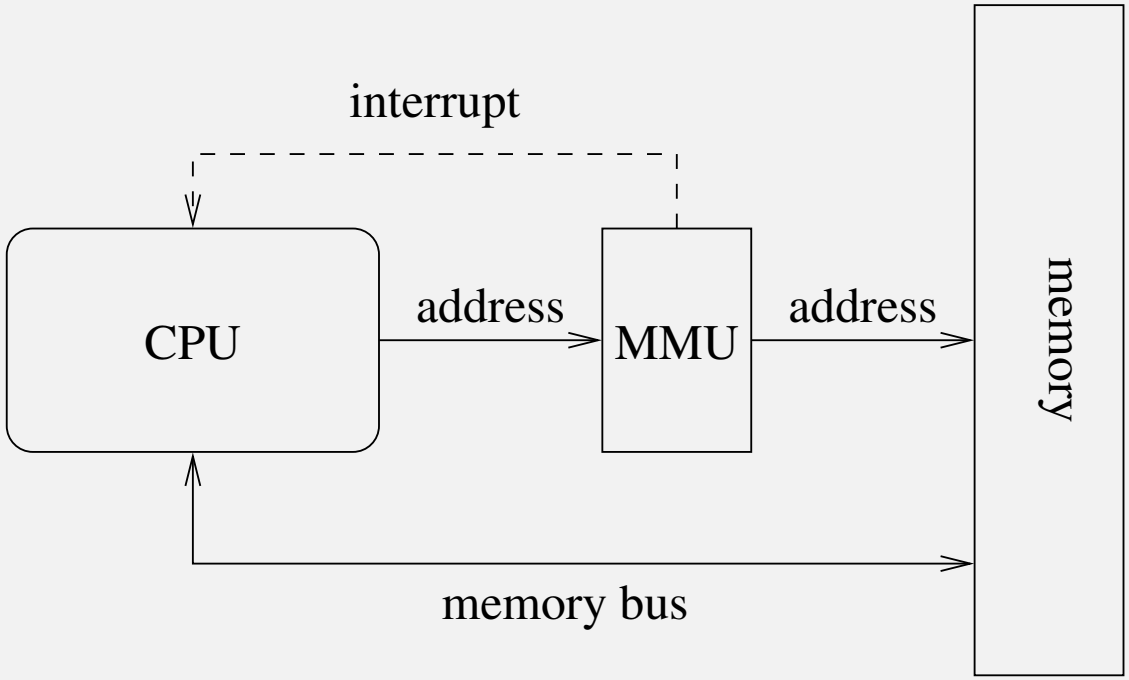
The next issue is memory protection: this must stop a program from writing and/or reading the memory used by another program or by the OS

The OS must be allowed to read and write any part of memory

Again, there must be hardware support to do this to make it fast

There is a table of flags in a special piece of hardware: the *memory management unit* (MMU). These flags say whether the *currently running* (user mode) program can read or write a given area of memory

History



One bit to say if an area is readable; another to say if it is writable

It is often useful to separate ability to read from ability to write

History

Setting these flags in the MMU is a privileged operation, of course

And if an unprivileged program tries to read or write to an area of memory for which it does not have the required permission (say some other program's or the OS's memory) the MMU raises an interrupt and the OS takes control again

It would not be feasible to have control like this on a byte-by-byte level, so memory is divided into blocks called *pages*

A page is just a contiguous area of memory: 4096 bytes is popular on modern machines, though current hardware can support 4MB pages

History

A page is marked as read/writable as a whole: this makes this technique practical

Exercise. How many flags (bits) are needed to cover 2GB? How many bytes of flags does that correspond to?

Note that these flags are part of a program's state that must be saved and restored when that program is re-scheduled

There is usually also an *executable* flag: can you execute code from this memory address?

History

Every read or write to memory is checked by the MMU before it is allowed: this means the hardware that does this check has to be very fast

We shall not be going into this in depth here, because in modern machines this is enhanced by the notion of *virtual memory*

This we shall cover later, but it builds on the ideas above and provides a much more flexible method of protection

History

Thus we can see some of the requirements of an operating system

- Resource management
  - in particular program scheduling (CPU time)
  - also disk, network, . . .
- Protection
  - in particular memory
  - also files, network data, . . .
- Efficiency
  - in particular with regards to time
  - also size, energy, . . .

History

By making privileged operations *only* available to the OS, the OS can enforce policy on access and ensure fair distribution of shared resources

History

In current large OSs we have:

- Windows. Preemptive multitasking from Windows NT (1996) onwards. Previously (Windows 95 etc.) was little more than a monitor with a pretty interface on top
- Linux. A Unix re-implementation. Preemptive multitasking from inception (1991). (Recall that Unix had preemption from early 1970s)
- MacOS. MacOS X is a Unix derivative (BSD), from 1999 onwards. Earlier systems (MacOS 9 and earlier) were completely different, with no preemption, only cooperative

History

- Solaris. A Unix derivative (System V). Preemptive multitasking from inception (1992), an extensive rewrite of the earlier SunOS (1983), another Unix variant (BSD)
- OS/2. Initially from Microsoft and IBM (1997), then just IBM as Microsoft went off to do its own thing. Intended to be the followup to DOS. Multitasking when the hardware could support it: OS/2 2.0 (1992) could run multiple copies of DOS/Windows simultaneously. Previously used a lot in bank ATMs (until IBM ended support in 2006). OS/2 3.0 became Windows NT

History

And thousands of others: but the major players in the PC market are either derived from Windows NT, or from Unix

In contrast, in the embedded market are things are much more mixed, with both purpose-built OSs and slimmed-down derivatives of the general-purpose OSs all having major representation

History

With Windows, rebooting is the first thing an admin tries to fix a problem; with Unix, it's the last

Anon.

□

Processes

We now look at the programs we want to run

The word *process* is used to describe

- the executable code *and*
- its data *and*
- the associated information the OS needs to run it

Note this is different from *processor* and *program*

Other words: task, job

Processes

A single program might possibly use more than one process

For example, one process to compute a picture and another to display it: this is called *structure by process*

Though this is perhaps the exception, these days. Quite often a program uses just one process

And structuring can be done by using multiple *threads of execution*, often running in parallel

But it is coming back in Web browsers using one process per tab to provide security isolation between tabs

Note to think about later: Web browsers use OS process protection and isolation mechanisms to provide tab protection and isolation

Processes

An OS needs to keep a lots of information about a process, including

- where in memory its code is
- where in memory its data is
- what permissions it has on those parts of memory (MMU flags)
- how much time it is allocated
- how much time it has used
- similarly for other shared resources, e.g., the amount of I/O or networking done
- the cpu's PC and registers
- flags from the MMU
- and lots more as we shall see later

It uses this information to schedule and protect the process

Processes

A process can be in one of several *states*. In a simplified model, the five main states are

1. New. A process that has just been created
2. Running. It is currently executing on the CPU
3. Ready. It is ready to run, but some other process (or the OS) is currently using the CPU
4. Blocked. Waiting for some event or resource to become available. E.g., waiting for a block of data to arrive from the disk
5. Exit. A process that has finished

Real OSs will have more states than this, but these are the important ones

Processes

We shall assume, for simplicity, that we have just one processor

The OS will have lists of processes in each state, so the scheduling decision is making the choice of which process to move between which states

Again, in real OSs, these will not be simple lists. They might be arranged in priority order, or might be some more sophisticated datastructure: e.g., a pair of lists, one for real-time processes and the other for non-real-time; or a tree

Processes

Example: in Unixes, processes are arranged in *trees*

```
systemd-+-ModemManager---2*[{ModemManager}]
        |-NetworkManager---2*[{NetworkManager}]
        |-Thunar---3*[{Thunar}]
        |-accounts-daemon---2*[{accounts-daemon}]
        |-agetty
        |-atd
        |-auditd---{auditd}
        |-avahi-daemon
        |-chrome-+-2*[cat]
        |        |-chrome-+-chrome-+-chrome---12*[{chrome}]
        |        |        |        |-chrome---19*[{chrome}]
        |        |        |        |-3*[chrome---11*[{chrome}]]
        |        |        |        |-chrome---15*[{chrome}]
        |        |        |        |-chrome---17*[{chrome}]
        |        |        |        |-chrome---16*[{chrome}]
        |        |        |        |-chrome---10*[{chrome}]
        |        |        |        `-chrome---23*[{chrome}]
        |        |        `-nacl_helper
        |        |-chrome-+-chrome
        |        |        `-7*[{chrome}]
```

Processes

This allows control of a whole bunch of processes as a group

A group within the tree has a *session leader*

For example, killing the session leader would typically kill all the processes in the group

In the example above, exiting the chrome session leader would kill it and all its subprocesses

Processes

So we have these five main states: New, Ready, Running, Blocked and Exit, and a process will be moved by the OS between them

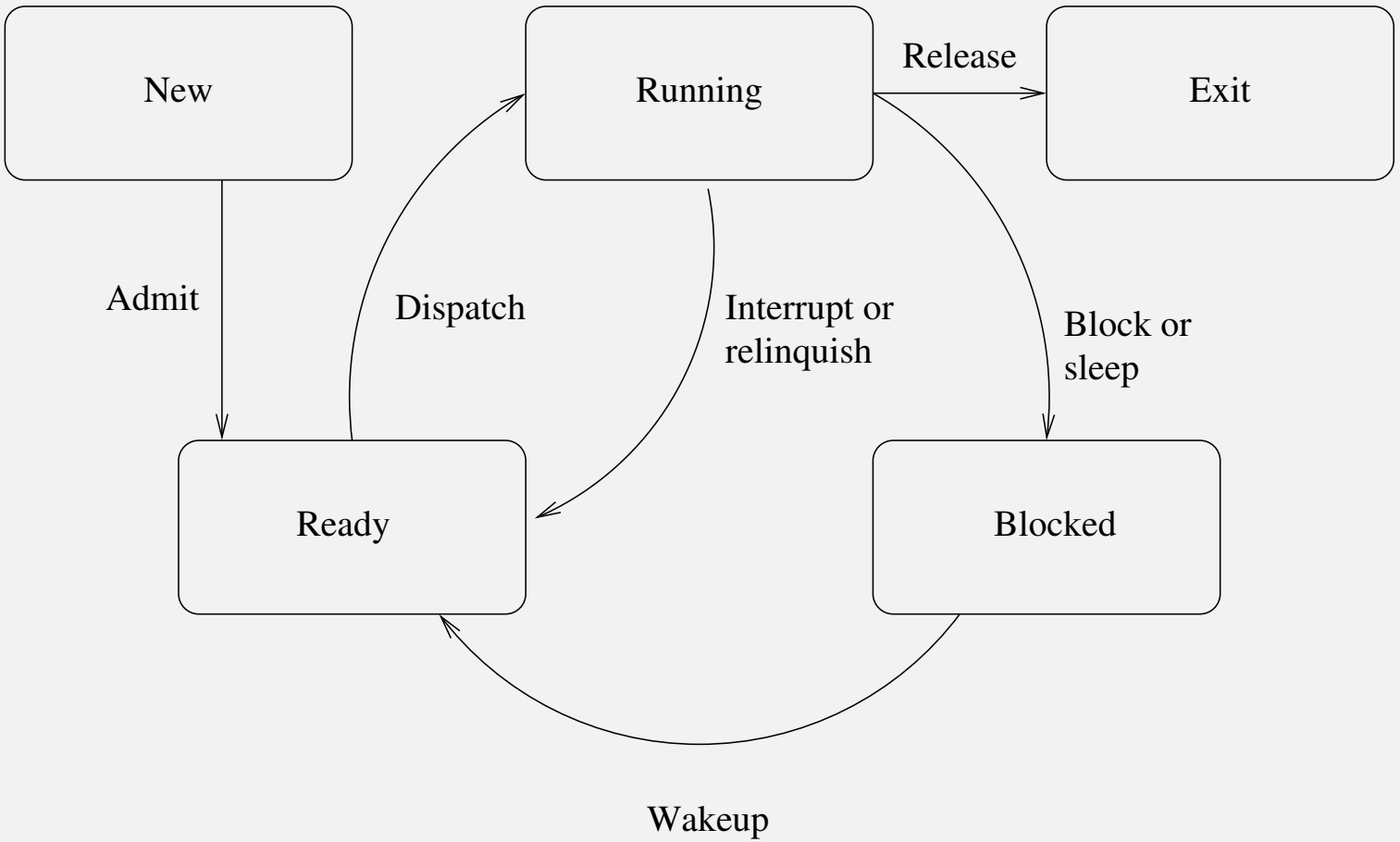
A new process will begin in the state New

A process just finished will be in the state Exit

In between the OS must decide, as part of its scheduling, where to place each process

There is a standard *finite state machine* that describes the allowed transitions between states

Processes



Process State Transitions

Processes

A typical transition is

1. The OS decides to schedule a process on the ready list
2. The process is *dispatched*, i.e., the OS marks its state as running and starts executing it (jump and drop privilege)
3. The process may choose to voluntarily suspend itself: *relinquish* (e.g., a clock program displaying the time might suspend itself for a minute)
4. Or an interrupt may arise, e.g., from a packet arriving on the network card, or a key being hit on the keyboard
5. Or a timer interrupt may happen when the process has used its *time slice*. In any of these three cases the OS moves the process to the Ready state

Processes

6. Or the running process may need some resource the OS must supply (e.g., for disk access) so it does a syscall and must wait until the resource is ready (e.g., the disk returns some data); the OS moves it to Blocked
7. In the case of a blocked process, perhaps data has returned from the disk and the process can *wake up* and become Ready again. Note that the process won't necessarily start running immediately, it is just ready to run when it gets its chance

And to make it clear: it's not the processes moving themselves between the states, it's the OS moving them between the lists of processes in each state

Processes

Remember, early OSs without timer interrupts had to rely on processes relinquishing control every once in a while: cooperative multitasking

User programs running on such OSs had to be explictly written to be cooperative

And so were often not

For example, Windows 3.1, MacOS 9

Exercise. Write a program that voluntarily relinquishes occasionally

Processes

New and Exit states happen just once per process

- New. For a process just created, perhaps code and data are not yet loaded into memory. The OS datastructures needed to manage the process have been created and filled in
- Exit. For a process that has just finished. Some tidying up is usually needed after a process ends, such as closing files or reclaiming memory or other resources it used

Processes

A real example:

```
USER       PID  PPID PRI %CPU %MEM STAT     TIME COMMAND
rjb       3974  4831  22  0.0  0.1 R+   00:00:00 ps
rjb       4495  4831  24  0.0  2.0 S    00:01:11 emacs
rjb       4538  4530  23  0.0  0.2 Ss+  00:00:00 bash
rjb       4540  4534  24  0.0  0.2 Ss   00:00:00 bash
rjb       4664  4556  21  0.0  0.6 S+   00:00:08 pine
rjb       4831  4829  24  0.0  0.2 Ss+  00:00:00 bash
rjb       7839  4831  15  0.0  0.1 Ss   00:00:00 firefox
rjb       7851  7839  14  0.0  0.1 S    00:00:00 run-mozilla.sh
rjb       7856  7851  24  0.2 16.6 Sl   00:31:47 firefox-bin
rjb      14880     1  16  0.0  3.1 Dsl  00:06:43 recollindex
```

Example processes under Linux

Processes

- S. Sleeping: like blocked (interruptible sleep; waiting for an event like a timer or other interrupt)
- D. Disk wait (uninterruptible sleep; waiting for requested I/O)
- R. Running or ready to run
- It is hard to catch new and exiting processes

s: session leader; +: foreground process group; l: multithreaded

Processes

Other columns of interest

- User. The user who owns the process
- PRI. Priority. In Linux, priorities are integers, larger indicates less important
- PID. Process identifier. An integer that uniquely identifies this process
- PPID. Parent PID. The PID of the process that started this process. This allows processes to be grouped in trees. Process number 1 is the parent of all processes
- CPU, MEM, TIME. How much of these resources this process is using

Processes

So we can see some more of the information that a process needs to collect and maintain:

- User identifiers (userids)
- A priority
- Statistics like memory and CPU used
- The state

But there are still more that will become clearer as we go along

This collection of data a process needs is called the *process control block*, or PCB

Processes

To pause and restart a process (e.g., on an interrupt) requires the saving and restoring of the process state: CPU registers, stack pointers, MMU flags, etc.

This will also be stored in the PCB

So process handling is very similar to the way interrupts are handled

Processes

Process creation is quite involved

- Allocate and create PCB structure
- Find a free PID
- Determine and allocate the necessary resources (in particular memory)
- Determine the initial priority of the process
- Insert PCB into the relevant list

This is what happens in the New state; it can now move to Ready

Again, the process might not start running immediately, as there could be some higher priority process that must run first

Processes

Most processes are created (*forked/spawned*) by other processes: of course, only the OS can actually create processes

A user process that wants a new process will ask the OS to create one (using a syscall)

Processes use resources like memory and CPU, so the OS must be involved

- A process decides it wants to start another process. E.g., a GUI process as a response to a user clicking on an icon
- It calls the OS kernel (syscall), telling it what process it want to start (e.g., “start the browser program”)
- The OS can now create a new process according to the specifications given
- The new process can now be scheduled

Processes

The original calling process will generally be the parent of the new process

Of course, the OS can choose not to create the new process if some policy says not to, or there is not enough memory, or some other reason

In that case, the original process usually gets a message back from the OS (via the value returned from the syscall) explaining the problem

Processes

The question arises: if processes are created by other processes, how do we get started?

This is the *bootstrapping problem*

In Unix, there is an ancestor process, sometimes called *init* or *systemd*, PID 1, that gets created at switch-on time and it serves to create all other processes

Bootstrapping is complicated as it has to determine the hardware and how it is connected and initialise it, set up all the appropriate datastructures the OS needs, start lots of service processes running, all before it can begin to look at what the user wants

This is quite often simply called *booting*

Processes

How does process 1 get started?

To get going, the system needs a small chunk of *non-volatile* memory, i.e., memory that doesn't lose its content when power is removed

The details of booting are very messy, but in essence:

- When the machine is switched on the processor jumps to a specific location in the non-volatile memory (0xFFFF0000 on PCs)
- At this location is a small program (the *boot loader*, or *bootstrap loader*) that is just big enough to load and run a larger program (from bigger non-volatile memory, or perhaps disk)

Processes

- This might be repeated until we have a big enough program running that is capable of reading from, say, the start of the hard drive
- This is often itself another bootstrap program (NTLDR, and GRUB are common) that might give the user a choice of operating systems to load, but usually just goes ahead and loads one from disk (or network, or whatever)
- This may require the bootloader to have some understanding of how data is laid out on the disk, which itself is non-trivial (see later)
- Eventually, enough of the operating system kernel is loaded that it get itself going properly, e.g., start init

Processes

Exercise. Some machines do not have disks (or other persistent storage), for example *thin clients*. Read about how these can boot

□

Scheduling

We now look at scheduling: how to choose which process to run next

This is an *extremely* difficult problem and still has not been solved to everybody's satisfaction

Scheduling

A list of scheduling algorithms, from Wikipedia:

Borrowed-Virtual-Time Scheduling (BVT), Completely Fair Scheduler (CFS), Critical Path Method of Scheduling, Deadline-monotonic scheduling (DMS), Deficit round robin (DRR), Dominant Sequence Clustering (DSC), Earliest deadline first scheduling (EDF), Elastic Round Robin, Fair-share scheduling, First In, First Out (FIFO), also known as First Come First Served (FCFS), Gang scheduling, Genetic Anticipatory, Highest response ratio next (HRRN), Interval scheduling, Last In, First Out (LIFO), Job Shop Scheduling (see Job shops), Least-connection scheduling, Least slack time scheduling (LST), List scheduling, Lottery Scheduling, Multilevel queue, Multilevel Feedback Queue, Never queue scheduling, $O(1)$ scheduler, Proportional Share Scheduling, Rate-monotonic scheduling (RMS), Round-robin scheduling (RR), Shortest expected delay scheduling, Shortest job next (SJN), Shortest remaining time (SRT), Staircase Deadline scheduler (SD), “Take” Scheduling, Two-level scheduling, Weighted fair queuing (WFQ), Weighted least-connection scheduling, Weighted round robin (WRR), Group Ratio Round-Robin: $O(1)$

Scheduling

And they are just the ones people can be bothered to write about on Wikipedia

Scheduling

Think of the problems

- Try to give each process its fair share of CPU time
- and no starvation of any process
- Try to make interactive processes respond in human timescales
- Try to give as much computation time as possible to compute-heavy processes
- Ensuring critical real-time processes are dealt with before it is too late

Scheduling

- Try to service peripherals in a timely way
- Understanding the various requirements of hardware: mice and printers are slow; networks and disks are medium; memory is fast
- Try to distribute work amongst multiple devices; e.g, CPUs and networks
- Try to make best use of the hardware and use it efficiently
- Try to make behaviour predictable: we don't want wildly erratic behaviour
- Try to degrade gracefully under heavy load
- And so on

Scheduling

And do it all quickly!

Scheduling

Here we shall concentrate on CPU scheduling, but remember the CPU is just one resource of many

A related problem is *I/O scheduling*, managing requests and responses to other devices, such as disks, to make best use of them

I/O scheduling is important, but we shall not talk about it here

But we will note in passing that the various schedulers for the various resources may not agree on what should be done next!

Scheduling

All this needs to be quantified somehow so we can use the numbers to make choices

Example measurements include:

- CPU cycles used
- Memory used
- Disk used
- Network used
- Etc.

Scheduling

And we can quantify results

- Throughput; more or fewer jobs finished in a given time
- Turnaround; response time: interactive response is snappy or sluggish
- Real-time; we *must* deal with this data now else the car will crash (deadlines)
- Money; we've been given money to get this data ready in the next hour
- Etc.

Scheduling

All this information was originally collected to figure out how much money to charge the user

These days most people are not so worried about charging as we all have our own computers. We are more concerned about making the best use of our computer

Though it's still important: cloud services (e.g., Amazon, Google, Microsoft) sell time on their machines

They charge based on disk storage, data input and output and compute (CPU) used

There's nothing new in Computer Science: just recurring fashions!

Scheduling

Algorithms

We now look a just a few of the simplest scheduling algorithms

Exercise. Have a look at textbooks for gruesome detail on the relative performances of these algorithms

Scheduling

Algorithms

Run until completion

First in, first out (FIFO); non-preemptive batch, as on pre-OS machines

- Good for large amounts of computation
- No overheads of multitasking
- Poor interaction with other hardware; can't process while printing (recall spooling)
- No interactivity

Clearly not suitable for modern machines?

Actually still the basis for large supercomputers

Scheduling

Algorithms

Shortest Job First

Shortest-time-to-completion runs next; non-preemptive

- No multitasking
- Good throughput
- Similar behaviour to FIFO on average
- Long jobs suffer and might get starved
- Difficult to estimate time-to-completion, so reliant on the job description for this information

Scheduling

Algorithms

Run until completion plus cooperative multitasking

Non-preemptive

- Weak multitasking
- Uses round-robin or similar to choose another task on relinquish
- Poor interactivity
- Easy for a process to starve other processes
- Hard to write “good citizen” programs

Was used on millions of personal computers for a long time

Scheduling

Algorithms

Preemptive Round Robin

Give each process, in turn, a fixed time slice

- Multitasking
- Gives interactive processes the same time as compute processes
- No starvation
- Better interactivity than cooperative systems
- Not good for either interactive or real-time; may have to wait a long time for a slice of time

Scheduling

Algorithms

Round Robin

More suited to systems where all the processes are of equal (or nearly equal) importance; e.g., dedicated appliances like network routers that have to decide how share network capacity fairly

Scheduling

Algorithms

Shortest Remaining Time

Time slice, pick next process by the estimate of the shortest time remaining; preemptive

- Good for short jobs
- Good throughput
- Long jobs still can be starved
- Still hard to make estimates of times

Scheduling

Algorithms

Least Completed Next

The process that has consumed the least amount of CPU time next

- All processes make equal process in terms of CPU time
- Interactive processes get good attention as they use relatively little CPU
- Long jobs can be starved by lots of small jobs

Scheduling

Algorithms

These algorithms have good points, but they also have bad points: so “obviously” we just need to tweak them a bit

But beware of patching and tweaking without having a good overview of what’s happening

Many a system has ended up with a scheduler that’s large, slow and impossible to understand

And impossible to fix when you stumble across the next deficiency

Scheduling

Algorithms

At the very least we need to take interactivity, priority, and more into account

How do we know if a process is interactive or compute intensive?

Watch how much I/O is happening and how long we are waiting for it: high I/O per compute is interactive, low is compute intensive

A process can be a mix of both, of course: it might move between the two over time

Scheduling

Algorithms

Similarly, priorities can be

- Static. Unchanging through the life of the process. Very simple, but unresponsive to change (e.g., a process that alternates interactivity with urgent computation)
- Dynamic. Priority responds to changes in the load. Harder to get right, more expensive to compute.
- Purchased. Pay more, get higher priority!

Scheduling

Algorithms

Highest Response Ratio Next

A variant of SRT, where we take the time a process has been waiting since its last time slice into account

$$\text{Dynamic priority} = \frac{\text{time so far in system}}{\text{cpu used so far}}$$

- A process executes repeated time slices until its priority drops below that of another process
- Tries to avoid starvation
- Long jobs will eventually get a slice

Scheduling

Algorithms

Highest Response Ratio Next

- New jobs get immediate attention as CPU time is near 0
- But now critical shorter jobs might not finish in time as they could get scheduled after a long-waiting job
- This needs frequent re-evaluation of priorities to get good behaviour, which implies small timeslices, and so lots of scheduling overhead

Scheduling

Algorithms

Multilevel Feedback Queueing

Can be used when we have no estimates on run times

- There are multiple FIFO run queues, RQ_0, RQ_1, ... RQ_n. with RQ_0 the highest priority, RQ_n, the lowest
- Queues are processed in FIFO fashion, in priority order, so RQ_1 does not get a look-in until RQ_0 has emptied
- Each process is allocated a *quantum* of time (a timeslice)
- A new process is admitted to the end (last) of RQ_0
- When the running process has used its quantum of time, it is interrupted and placed at the end of the next lower queue (demoted)

Scheduling

Algorithms

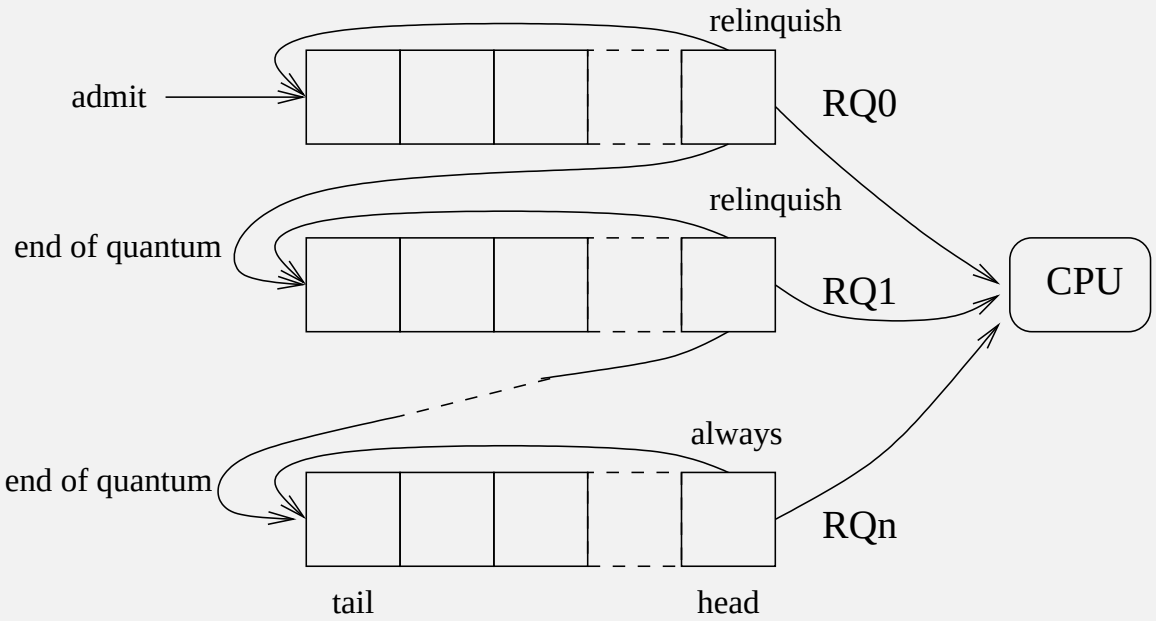
Multilevel Feedback Queueing

- If the running process relinquishes voluntarily before the end of the quantum, it gets placed back at the end of the *same* queue
- If it blocks for I/O, it will be promoted and placed at the end of the next higher queue (when ready to run)
- Demoted processes in RQ_n get placed back at the end of RQ_n

Scheduling

Algorithms

Multilevel Feedback Queueing



Multilevel Feedback Queueing

Scheduling

Algorithms

Multilevel Feedback Queueing

- This gives newer, shorter processes priority over older, longer ones
- I/O processes tend to rise, getting more priority
- Compute processes tend to sink, getting less

Old processes tend to starve with this, so a variant doubles the quantum for each level: RQ_0 gets 1, RQ_1 gets 2, RQ_2 gets 4, and so on

So compute intensive processes get a big bite, whenever they get a chance, at the potential cost of responsiveness to a new process

Scheduling

Algorithms

Another advantage of MFQ is that it does not need to do any arithmetic: it just moves processes between queues

Remember, in early machines, arithmetic was a lot more time-consuming than it is now

This scheme was used by Windows NT and Unix derivatives, as we shall see next

Scheduling

Algorithms

Traditional Unix scheduling

As used in older Unix derivatives — modern scheduling is much more sophisticated

Everything is based on timer interrupts every $1/60^{\text{th}}$ second

A priority is computed from the CPU use of each process

$$\text{Priority} = \text{base priority} + \frac{\text{CPU time used}}{2}$$

Scheduling

Algorithms

Traditional Unix scheduling

A process with the *smallest* priority value is chosen next (thus – mostly — a process that has used less CPU)

Processes of the same priority are treated round robin

Note that this is actually very similar in effect to Multilevel Feedback Queueing where a priority of n corresponds to RQ_n

The base priority depends on whether this is a system process or a user process, with user priority being lower (i.e., with a larger value)

Scheduling

Algorithms

The CPU use of a process is recorded and halved every second: this decays the influence of CPU usage over time and makes the priority based on *recent* behaviour

This algorithm gives more attention to processes that have used less CPU recently, e.g., interactive and I/O processes

$$\text{Priority} = \text{base priority} + \frac{\text{decayed CPU time}}{2}$$

Scheduling

Algorithms

Traditional Unix scheduling

Processes can choose to be *nice*

Generally, $-20 \leq \text{nice} \leq 19$, but only certain users (administrators) can use negative nices

$$\text{Priority} = \text{base priority} + \frac{\text{decayed CPU time}}{2} + \text{nice}$$

A process that has nice -20 can really jam up the system

But nice also enables a *purchased* priority

Scheduling

Algorithms

Traditional Unix scheduling

There are a few problems with the traditional technique

The priorities were recomputed once per second, all in a single pass, taking a significant chunk of time (on old machines)

It does not respond quickly enough to dynamic changes in the system

And does not scale to large numbers of processes

So this is not used in modern systems, where many 100s of processes is common

Scheduling

Algorithms

Fair Share Scheduling

And there are other problems that should be addressed

Modern machines can support many users simultaneously: what happens if user A has 9 processes and user B just 1?

Should A get 90% of the CPU time and B 10%?

Fair share scheduling is where each *user* (or group or other collective entity) gets a fair share, rather than each *process*

Scheduling

Algorithms

Fair share Scheduling in Unix

Recall processes are collected in groups in a tree

$$\begin{aligned}\text{Priority} = \text{base priority} &+ \frac{\text{CPU time used by process}}{2} \\ &+ \frac{\text{CPU time used by process group}}{2} + \text{nice}\end{aligned}$$

Scheduling

Algorithms

Fair share Scheduling in Unix

Modern Unix derivatives use much better, and much more complicated, scheduling algorithms than this

They can afford to be more complicated as CPUs are now much faster

Exercise. Read up on $O(1)$ scheduling and *The Completely Fair Scheduler*

Also have a look at scheduling for real-time systems: for when a process must *absolutely* get scheduled within a given time

□

Scheduling

Scheduling the CPU is clearly a difficult problem

It requires the collection and manipulation of many statistics about processes

Scheduling one resource (the CPU) is hard enough

We now look at a new problem that arises when we want to schedule *multiple* resources

Deadlock

Processes compete for resources like disks and network and the OS mediates this

To read from a disk, a process must call the OS kernel and wait for the kernel to reply

Terminology

When we say “a process waits for the kernel” we mean, of course, something entirely different

What actually happens is the kernel marks the process as blocked, and does not consider it for scheduling until the requested resource has arrived

There is no “waiting” happening: the process does not run when blocked

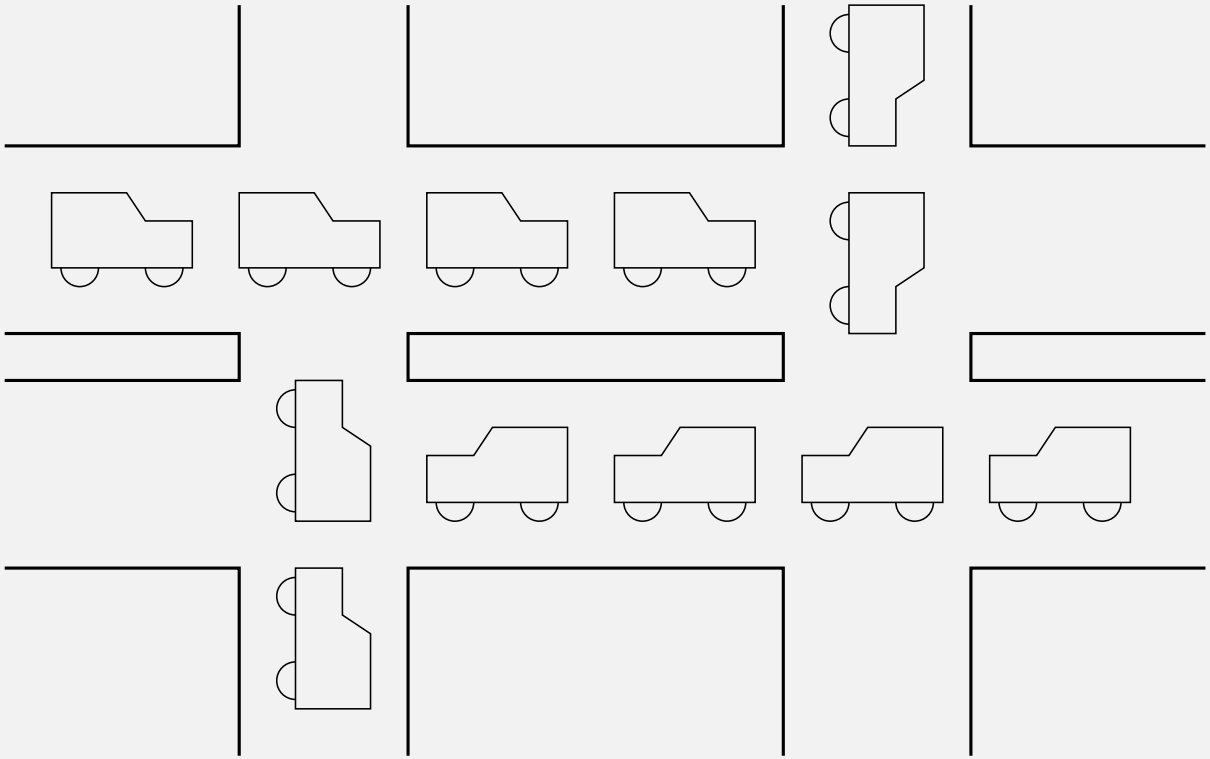
Deadlock

Processes compete for resources like disks and network and the OS mediates this

To read from a disk, a process must call the OS kernel and wait for the kernel to reply

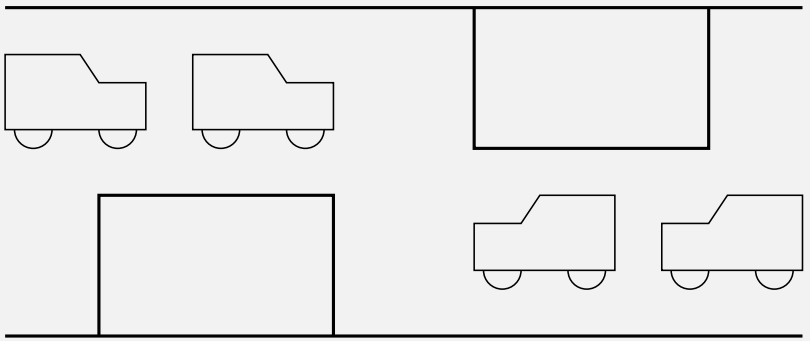
Sometimes the delay is infinite!

Deadlock



Gridlock/Deadlock

Deadlock



Gridlock/Deadlock

Deadlock

This can happen in an OS

Process P_1 wants to copy some data from disk D_1 to disk D_2, while process P_2 wants to copy some data from disk D_2 to disk D_1

- Initially P_1 is running and makes a request for access to D_2
- The OS takes over and gives P_1 exclusive access to D_2 (not a smart OS)
- The OS decides to run P_2
- P_2 runs and makes a request for access to D_1
- The OS takes over and gives P_2 exclusive access to D_1

Deadlock

- The OS decides to run P_1
- P_1 runs and makes a request for access to D_1
- The OS takes over and notices P_2 has locked D_1, so P_1 must wait until P_2 has finished with it; P_1 moves to state blocked
- The OS decides to run P_2: it can't run P_1 as it is blocked
- P_2 runs and makes a request for access to D_2
- The OS takes over and notices P_1 has locked D_2, so P_2 must wait until P_1 has finished with it; P_2 moves to state blocked
- Now both P_1 and P_2 are blocked and the OS can't run either process!

Deadlock

P_1 can't run until D_1 is free, but D_1 won't be free until P_2 runs

P_2 can't run until D_2 is free, but D_2 won't be free until P_1 runs

This is called *deadlock*

Deadlock can happen on any kind of shared resources that require exclusive access

And with more than two processes: think of three or more processes in a circle

Deadlock

A formal definition:

A set of processes D is *deadlocked* if

1. each process P_i in D is blocked on some event e_i
2. event e_i can only be caused by some process in D

Deadlock

Note that you can only get deadlock if

- there is more than one resource
- there is more than one process[1][2]

□

[1] It could technically happen with just one process, but that would be quite dumb programming to request for a resource you already have

[2] I've seen it happen

Deadlock

Deadlock is only possible if certain *necessary* conditions are met: the *Coffman Conditions*

1. **Mutual exclusion** Only one process can use a resource at a time
2. **Hold-and-wait** A process continues to hold a resource while waiting for other resources
3. **No preemption** No resource can forcibly be removed from a process holding it

All of these must hold for it to be *possible* to deadlock

Deadlock

It might seem easy to avoid these, but in practice it's harder than you think

Suppose we ensure Hold-and-wait never happens, e.g., requiring a process to drop other resources it holds whenever it gets blocked on a new request

When it gets the new resource it will have to go back and pick up the other resources again

Which may require it to drop the new resource while waiting. . .

It is easy to get into a situation where the process never manages to get all the resources it needs: called *indefinite postponement*

Deadlock

A deadlock may be possible but will only actually happen if

4. **Circular Wait** There is a circular chain of processes where each holds a resource that is needed by the next in the circle

This says that deadlock is happening as in the formal definition

Deadlock

Dining Philosophers

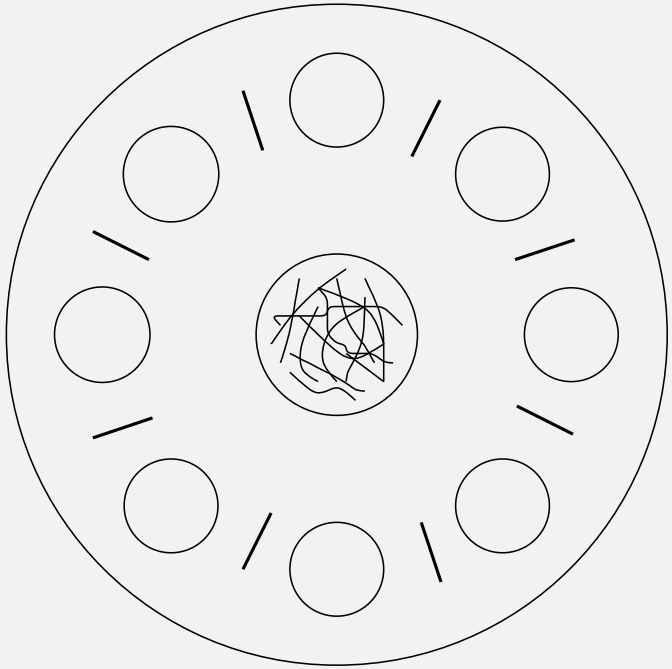
A popular illustration of deadlock is *The Dining Philosophers*

Some Philosophers wish to share a plate of spaghetti, but they have only been provided with chopsticks

Unfortunately, there is not quite enough chopsticks to go around

Deadlock

Dining Philosophers



Dining Philosophers

Deadlock

Dining Philosophers

Each Philosopher needs two chopsticks to eat, one from each side of their plate

We have

1. Mutual exclusion. Only one Philosopher can use a chopstick at a time
2. Hold-and-wait. Each Philosopher wants to eat and won't let go of a chopstick until they have eaten
3. No preemption. No-one is going to tell a Philosopher what to do!

Deadlock

Dining Philosophers

And if they all grab the left chopstick simultaneously

4. Circular Wait. There is a circular chain of Philosophers where each holds a chopstick that is needed by the next in the circle

Of course, if the Philosophers were a bit more friendly, or polite, there would not be a problem

Deadlock

Dining Philosophers

Exercise. Identify the conditions in the car gridlock scenarios

□

Deadlock

There are two approaches to the problem of deadlock

1. Prevention. Stopping it happening ever by preventing one of the conditions occurring
2. Detection and Breaking. Letting deadlock happen, but spotting when it does and then breaking it by destroying one of the conditions

Deadlock

Prevention can be further refined

1a. Prevention. Constrain resource allocation to prevent at least one of the four conditions (e.g., ensure hold-and-wait never happens)

1b. Avoidance. Be careful not to allocate a resource if it can be determined that it might possibly lead to a deadlock in the future: keeping the system in a “safe” state

Avoidance is harder to manage as it needs to predict future requests for resources, but tends to be more efficient as it can allocate resources that prevention would disallow

Deadlock

Prevention

We can prevent deadlocks by disallowing any of the conditions

Deadlock

Prevention

Breaking Mutual Exclusion

This, quite often, cannot be broken

For example, trying to read a disk at the same time as another process is writing to it is a physical impossibility

A lot of hardware only works if there is exclusive access, e.g., printers, sound cards, etc.

Therefore we should take care to not hold on to such a resource for longer than is absolutely necessary

Deadlock

Prevention

Breaking Hold-and-wait

We can require a process not to hold any resources if it ever gets blocked on another resource

This has the non-progress feature, as noted previously, and can be very inefficient with much grabbing and releasing to no avail

Deadlock

Prevention

Breaking Hold-and-wait

We might require a process to request all necessary resources simultaneously, blocking until all are available

- This might prevent the process from doing useful other work while one of the resources is unavailable but not yet needed by the process
- Resources given to a process might be only needed much later, denying them to other processes in the meantime
- It may be that a process does not even know what resources it might need in advance, so this can be impossible to do anyway

Deadlock

Prevention

Breaking Hold-and-wait

A variant of this is not even to admit a process until all resources are available: this is even worse

Perhaps a process only needs to write to disk at the end of a 2 hour compute session: do we really want to lock the disk for 2 hours?

Deadlock

Prevention

Breaking No Preemption

This may only be possible for certain kinds of resource, namely those whose state can easily be saved and restored

The OS might choose to preempt the holding process and take the resource away from it, giving it back later when the process is scheduled again

This would be confusing for the holding process as the resource might change while it was owned by another process

Deadlock

Prevention

Thus, the resource should be given back to the process in an equivalent state to it was in when it was preempted, so the process can continue from where it left off

For some resources this is possible, e.g., memory

For others, not. For example, a printer

Deadlock

Prevention

Breaking Circular Waits

One possible solution is to put an ordering on resources

$$R_1 < R_2 < R_3 < \ldots$$

E.g., (much simplified)

$$\text{disk } 1 < \text{disk } 2 < \text{printer} < \ldots$$

Deadlock

Prevention

Then:

> *A process that holds resource R may then only request resources that are after R in the order*

In our example, if you have grabbed the printer, you cannot grab a disk

If a process makes such a request, the OS simply refuses to grant it

The process might choose to drop the printer and re-request the disk

Deadlock

Prevention

Breaking Circular Waits

Now we cannot deadlock, as a deadlock would imply A has grabbed R_i and requested R_j; while B has grabbed R_j and requested R_i

For this to happen we would have both

$$i < j \qquad \text{and} \qquad j < i$$

and this is impossible

Deadlock

Prevention

Breaking Circular Waits

This suffers the same problems as Hold-and-wait, namely inefficiency and unnecessarily holding resources

Further, it works only if the process can make requests in increasing order; not always possible as it is not always possible to know what you need in advance

And if you have R_1 and R_3, but then want R_2 you have to drop R_3, get R_2, then regain R_3; very inefficient

This usually effectively reduces to the request-all-at-once scenario

Deadlock

Avoidance

In contrast, deadlock *avoidance* does not break the conditions, but rather is careful not to do anything that might possibly create a deadlock in the future

For each request, we have to decide whether granting the resource will potentially lead to a deadlock immediately or in the future

Not so easy, as it requires knowing what might possibly happen in the future

An unsafe request will not be granted by the OS

Deadlock

Avoidance

The Banker's Algorithm

This algorithm, proposed by Dijkstra, is an example of how to do deadlock avoidance

The name comes from the concept of a bank lending money and then having it repaid

It makes two tests

1. Feasibility test. To see if a request is possible
2. Safety test. To see if a request is safe (cannot lead to deadlock)

Deadlock

Avoidance

The Banker's Algorithm

Only requests that are both feasible and safe are granted

Note that a request that is feasible but not safe implies a resource is lying idle

But we are erring on the side of safety at the cost of efficiency

Deadlock

Avoidance

The Banker's Algorithm

A request is *feasible* if, after granting the request, the total of allocated resource does not exceed the actual resource

That is, if we can actually satisfy the request. Don't allocate 10GB of memory if you only have 2GB

Sometimes it can be all-or-nothing: allocate access to the sound card, or not

Deadlock

Avoidance

The Banker's Algorithm

A state is *safe* if there is *at least one* possible future sequence of resource allocations and releases by which all processes can complete their computation (never deadlock)

So make sure there is always an escape route of allocations and releases

More than one route is better, but make sure there is at least one

A request is *safe* if, after granting the request, this leads to a safe state

Deadlock

Avoidance

The Banker's Algorithm

Dijkstra's Banking Algorithm:

> *Grant an allocation request only if this leads to a safe state*

This will *ensure* we are always deadlock-free, but can sometimes deny an allocation that might have been OK: it might have caused a deadlock, but by chance didn't happen to do so on some particular occasion

Deadlock

Avoidance

The Banker's Algorithm

In the implementation of this algorithm, for each process we need to know

- The current allocation to that process
- The maximum allocation that process might ever want

Deadlock

Avoidance

The Banker's Algorithm

Example. There are 12GB of memory and three processes sharing it

	Current allocation	Maximum need
Process 1	1	4
Process 2	4	6
Process 3	5	8
Available	2	

Deadlock

Avoidance

The Banker's Algorithm

	Current allocation	Maximum need
Process 1	14	4
Process 2	460	6
Process 3	58	8
Available	20637412	

This is a safe state because all three processes can finish: we can demonstrate a path to completion for all processes Process 2 currently has 4GB, but might eventually need 6 If the 2GB available are given to Process 2 Process 2 can finish releasing 6GB Then 3GB can be given to Process 1 which can then

Deadlock

Avoidance

The Banker's Algorithm

Thus there exists *a* path to completion for all processes where every process gets all the resources it might need: this is what the Banker's algorithm requires

This path may or may not be the actual one taken, e.g., Process 3 *might* exit without requiring that extra 3GB; this, of course, leads to another safe state

But we still need to be careful with allocations, as it is possible to move from a safe state to an unsafe one

Deadlock

Avoidance

The Banker's Algorithm

	Current allocation	Maximum need
Process 1	1	4
Process 2	4	6
Process 3	56	8
Available	21	

If Process 3 requests 1GB and this is granted This is an unsafe state Not necessarily deadlocked, but now we can't guarantee completion No process can be guaranteed to get enough resources to complete If we are lucky, a process might be able to finish without its maximum possible need But an OS can't

Deadlock

Avoidance

The Banker's Algorithm

There are several problems with this algorithm

- There must be a fixed number of resources to allocate: a fair assumption, but not always true
- The population of processes must remain fixed: not in a general-purpose OS
- Processes must know their maximum needs in advance: very unlikely
- Safety detection is quite expensive to compute, particularly with multiple resources
- It can sometimes refuse a request that could have turned out to be OK (by luck, perhaps): this leads to idle resources

Deadlock

Detection and Breaking

Next: deadlock *detection* systems allow deadlocks to happen but rely on noticing and breaking them

The hope is that detection and breaking will be cheaper than avoidance: this is not always clear

The earliest detection system was *Detection by Operator*

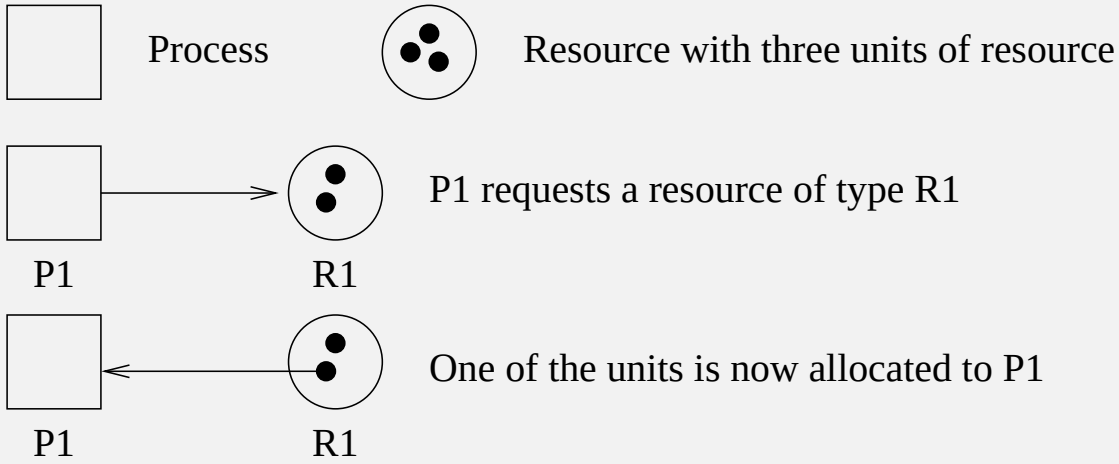
“The machine seems to have stopped. . . ”

Deadlock

Detection and Breaking

The chief method employed is to spot when the circular wait happens

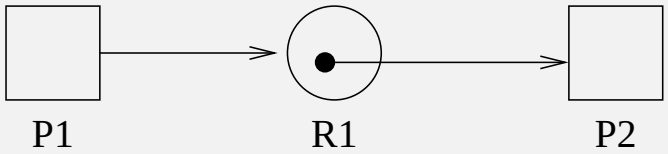
One method for deadlock detection uses *resource request and allocation graphs* (RRAG)



RRAGs

Deadlock

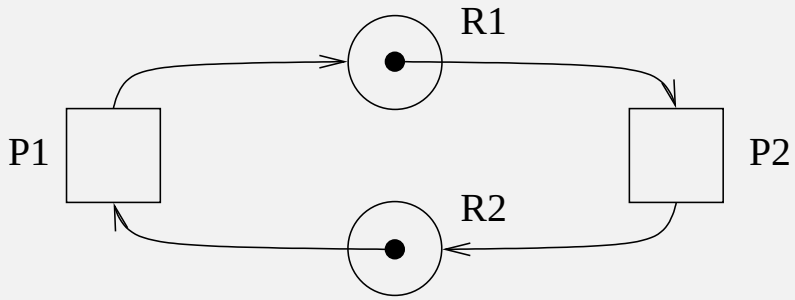
Detection and Breaking



P1 requests from R1, but it has no free units, so P1 will be blocked

Deadlock

Detection and Breaking



Circular Wait

P1 requests from R1, but it has been allocated to P2;
P2 requests from R2, but it has been allocated to R1:
deadlock

Deadlock

Detection and Breaking

So deadlock detection is just finding these kinds of loops in RRAGs

This can be done by *graph reduction*

For each process repeatedly

1. Remove all request links from the process to resources that are available (perhaps available after a reduction step)
2. When there are no requests links left, remove all links from allocated units of resource to the process

Deadlock

Detection and Breaking

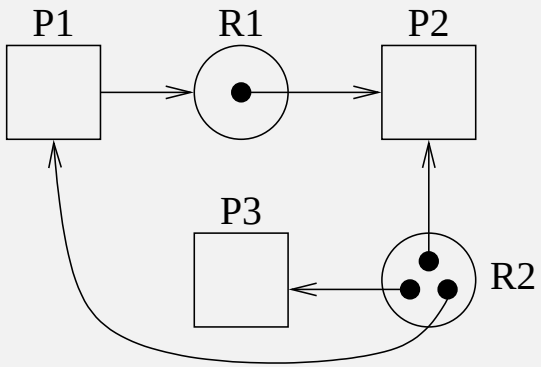
If we can reduce a RRAG by all processes, then there is no deadlock

1. Removing request links to available resources is allocating the requested resources to the process
2. Removing allocated links is the process finishing and returning its resources

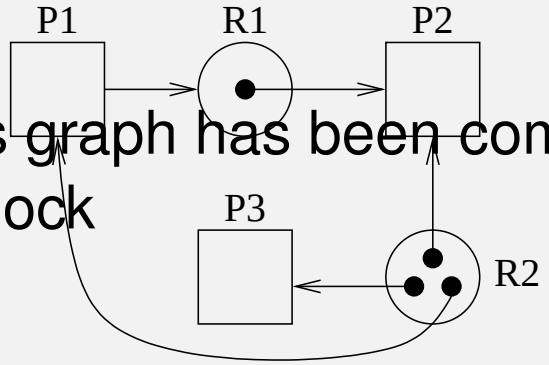
An example:

Deadlock

Detection and Breaking



A RAAG

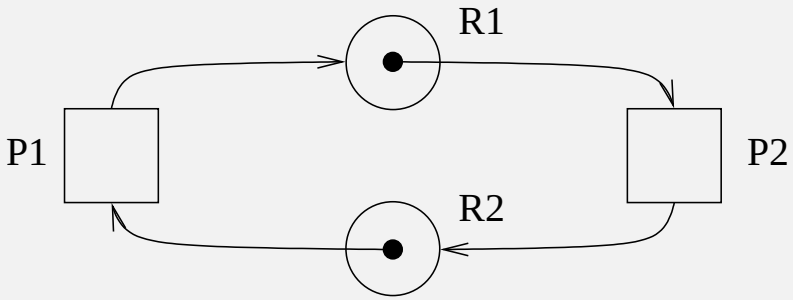


No more links, so this graph has been completely reduced and there will be no deadlock

We can satisfy P3's requests (none)

Deadlock

Detection and Breaking



Circular Wait

We can’t reduce this as no request links are removable

Thus this is deadlock

An advantage of this technique is that it isolates the parts that are deadlocking: we can see them in the graph

□

Deadlock

Detection and Breaking

So that leaves breaking the deadlock: as always there are lots of ways we can do this, none terribly satisfactory

- Kill one or more or all of the deadlocked processes: a bit drastic, but sometimes the only solution. But which process? For example, out of memory “OOM killers” are tricky to get right
- Preempt the blocking resources: better, if possible. If there are multiple resources causing the deadlock we have to choose which, as preempting just a few might free things up enough
- Add resources: rarely possible

Deadlock

Detection and Breaking

Exercise. Think about how you might apply deadlock prevention or breaking to a) Dining Philosophers and b) the car deadlock scenarios

Deadlock

In real life, a popular approach is simply to ignore the possibility of deadlock happening

Sometimes called the *Ostrich Algorithm*

There is not entirely stupid, as it argues that the costs associated with prevention or detection are large, and if deadlocks are rare, then the cost of an occasional reboot of the machine is small in comparison

In a carefully written OS, you can eliminate many of the possible causes of deadlock, or, at least, reduce the chances of them happening

Deadlock

Some resources are preemptable, e.g., memory (as we shall discuss in depth later), but a more general solution (also applied to memory) is *virtualisation*, where the OS pretends each process has sole access to a resource

We have already seen this for printers in the form of *spooling*

A process thinks it is writing to a printer, but it is actually writing to a tape, and the tape is later written to the printer

Deadlock

Similarly, for example, a process thinks it writes to a network card but the data is actually buffered by the OS somewhere in memory, to be sent later when the card is free

And so on for other kinds of devices: a process interfaces with its own virtualised device, there is no possibility of deadlock as every process can progress without waiting, and the OS sorts out transferring the data to or from the real device

But, of course, this new perspective just shifts the actual problem: when and in what order should the OS do the I/O?

This is called *I/O scheduling*

Deadlock

A printer could have simple first-in-first out queue, but other devices (disks, etc.) require something more sophisticated

For example, a typical disk driver will re-order writes to a disk match the physical movements of the write head

This is a topic we won't have time to go into!

Deadlock

Priority Inversion

One last word on deadlock, this one caused by process priorities and non-preemptible resources

Recall processes have priorities for scheduling purposes

- Suppose a low priority process L holds some resource
- A high priority process H is scheduled
- H requests the resource
- It can't get it as it is still held by L, so H is blocked
- Eventually, when L is done, H will be able to run

Deadlock

Priority Inversion

The low priority process is preventing the high priority process from running

This is called *priority inversion*

What is worse, other processes M of intermediate priority (that don't need the resource) can preempt L, preventing it running, and thus make the time H has to wait indefinitely long

If H is some real-time operation this can be serious

Deadlock

Priority Inversion

Fixes include

Priority inheritance The priority of H is temporarily loaned to L for the time it needs the resource. This ensures L can run and get out of the way

Deadlock

Priority Inversion

Priority ceilings Each *resource* is given a priority equal to the highest priority of any task that might want to grab that resource

When L gets the resource its priority is temporarily boosted to the priority ceiling of the resource: either immediately, or when another process tries to grab the resource

No other process that would want to grab the resource can be scheduled

Determining the ceiling is tricky, as it needs knowledge of the possible needs of processes

Deadlock

Priority Inversion

Disable scheduling preemption during use of non-preemptible resources. Only feasible if you keep the periods of use very short. Quite a popular solution for some resources, e.g., networks and disks, that are serviced very quickly

Exercise. Read up on these and other solutions

□

Process Protection

We now need to re-visit the idea of process protection

Recall by forcing access to resources via the kernel we can ensure that one user cannot interfere with the processes of another user

Thus a process must include the notion of *user*

This is usually encoded as a simple integer, the *userid*

Each user has their own unique userid and the OS uses this to determine whether one process can access files, other processes and so on

The userid also plays a role in Fair Share scheduling, of course

Process Protection

Userids are used everywhere

- Memory: a chunk of memory has a userid associated. This tells the kernel which processes are allowed to access it
- Files: each file has a userid associated. This tells the kernel which processes are allowed to access it
- Similarly for other resources

We shall see more of this when we get to memory and files

Exercise. Find out the userid allocated to you on the Uni's `linux.bath.ac.uk` machine

Process Protection

A new process (usually) inherits the userid of its parent process

Of course, this lead to another bootstrapping problem: how can a user get a process going in the first place?

If there are no processes running with my userid, how can I ever get a process to be created?

So there is a distinguished user, variously called the *superuser* or *root* or *administrator*

This is a **normal user**, but the OS allows it full access to other users' files, processes, etc.

In particular, root can suspend or kill any user's processes

Process Protection

Don't confuse the root user with kernel mode

Root's processes run in user mode, just like other users' processes

Hardware access is still mediated by the OS, but the inter-user protections are not enforced by the OS

In the OS there is the equivalent of

```
if uid_of_process == uid_of_resource or
   uid_of_process == uid_of_root
then
    allow access
else
    disallow access
```

Process Protection

Note that the privilege separation between superuser and normal user is used for protection of OS resources in exactly the same way as kernel mode and user mode is used for protection of hardware resources

It is the same idea being used in two different contexts

Process Protection

Critically, root can change the userid of its processes: by doing so it gives away its privileges, but thereby allows a normal user to have a process

When a user logs in to a system a process, owned by root, starts up, changes its userid to the user, and then starts other processes as that user

Process Protection

Many resources are restricted by the OS so only the superuser can use them: this provides an extra level of protection to resources that are sensitive

For example, shutting down the computer. We can't allow any user process to turn off the computer, so this operation is restricted by the kernel to the root user

Any shutdown program will need to have root ownership and this will be carefully policed by the system

Process Protection

Root is generally trusted by the kernel

So root-owned processes can completely trash everyone's programs and data on the machine if they want to

This is why you should keep the use of the administrator account to a minimum

Doing everyday stuff as administrator is just asking for trouble, and is throwing away many of those protection mechanisms that OSs have developed to provide

Process Protection

This user-level protection is what prevents my processes from interfering with your processes: as we have different userids, the kernel knows to keep them separate

In particular, if I download an application or web page that contains a malicious worm or virus, properly working protection will limit the damage that malware can do to just my files and my processes

Not ideal, but better than letting the malware have full reign over the entire machine

Process Protection

A big part of the spread of malware in Windows OSs is the weakness of this kind of barrier to their spread: too many programs run as administrator and this can ultimately cause the entire system to be affected

Note that if your OS *requires* the use of a virus checker, this is a strong sign that your OS is not confident in its implementation of process protection

Virus scanners address the *symptom*, not the *problem*

□

Process Protection

Capabilities

Root access is a bit all-or-nothing: it allows the root user all access to all (user mode) resources

And this is quite dangerous as it can too easily lead to accidental or malicious damage to the system through misuse

Process Protection

Capabilities

A refinement of the root idea is *capabilities*

This breaks access rights down into small parts

- Rights to access to the network driver
- Rights to access to the sound card
- Rights to access to the filesystem
- Rights to reboot the computer
- And so on

This can be broken all the way down to rights to access to individual files, say

Process Protection

Capabilities

We now have

```
if uid_of_process == uid_of_resource or
   process_has_capability(uid_of_process, resource) or
   uid_of_process == uid_of_root
then
    allow access
else
    disallow access
```

Process Protection

Capabilities

These capabilities are like tokens or keys that can be passed around, inherited by processes and so on

Capabilities allow finer control of security at the cost of a more complicated checking system

A few OSs, notably Flex, were built around the notion of capabilities and required hardware support to make things work with an acceptable speed

These never took off, though

But the idea has come back to modern OSs

Process Protection

Capabilities

In Android access to system resources are protected by capabilities: it calls them *permissions*

At either install time or the first time the application tries to access a resource the OS (not the application) asks the user whether that application should be allowed to access that resource

For example, WiFi network access, phone contact list access, SD card access, initiate phone calls, and so on

If so allowed by the user, the application can access those resources *and no others*

This mechanism would be great if only the user could be trusted to read and understand the list of requests. . .

Process Protection

Capabilities

Summary: user protection is useful and helpful

So don’t run things as root/administrator unless absolutely necessary

And don’t confuse it with kernel/user mode

□

Inter-Process Communication

We now look at how processes communicate amongst themselves

Many processes can be created, process, then exit without needing to refer to any other process

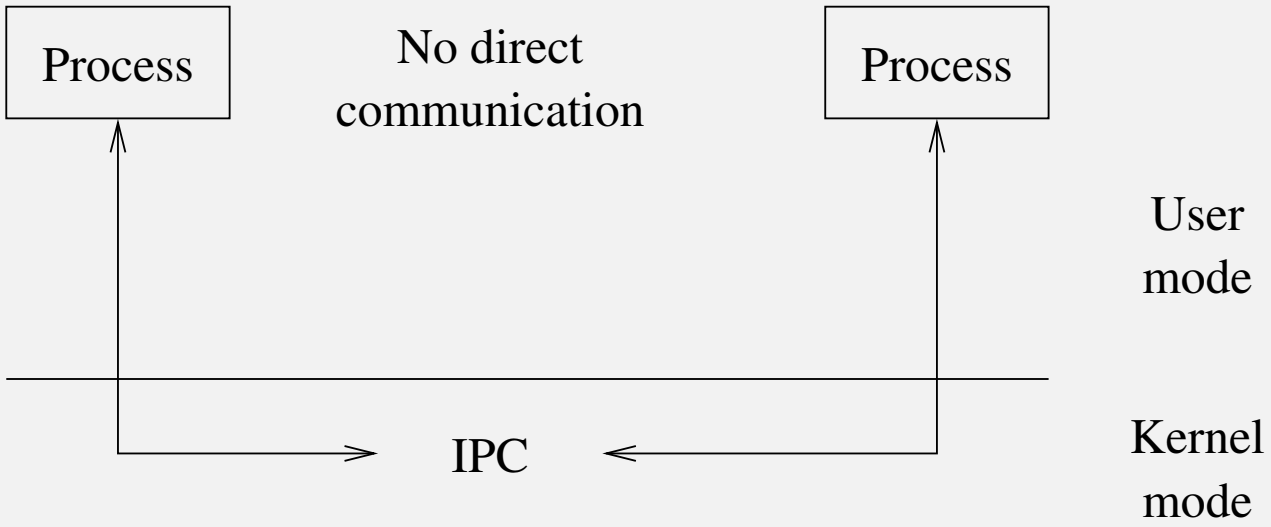
But there are many processes that need to send data to, or receive data from other running processes

For example, a new program starting might wish to tell the process managing the display that it wishes to pop up a window on the display

Or one process has to wait for another to finish some action (e.g., pop up a window) before it can progress itself: this is *synchronisation*

Inter-Process Communication

Inter-Process Communication (IPC) can be achieved in many different ways, but all must be, at base, supported by the OS; recall that by default the kernel tries to stop one process interfering with another



Inter-Process Communication

IPC contradicts this non-interference, and so must be treated very carefully by the kernel

There must be rules and restrictions, or else one process could just blast another process with data, preventing it from doing any useful work

Inter-Process Communication

We shall be looking at

- Files
- Pipes
- Shared memory
- Signals
- Semaphores (synchronisation)
- Software buses

as a sample of IPC mechanisms

Inter-Process Communication

Files

A simple way for two processes to communicate is using an existing resource, namely files

On the face of it this is just

- Process A wishes to send some data to process B
- A writes it to a file
- B reads it

This seems easy

But it's much harder than this

Inter-Process Communication

Files

- Which file to use? A and B need to agree on a filename to use, but this is not so easy. They can use a single "well-known" file, but this is problematic if many processes are all writing to the same file simultaneously. For example, C wants to communicate with D at the same time via the same file.
  They could have a separate file for each pair of processes, but to agree on a file name A and B must have previously communicated. . .
- How does B know when data has arrived? B might have to repeatedly poll the file until the data arrives. This doesn't scale well to large numbers of files or processes

Inter-Process Communication

Files

- The file protections must be set properly (recall userids) to allow only the authorised processes to read/write to them
- Files are quite slow relative to the mechanisms we are going to see later

In general, files are not used for IPC

But they should be considered as a choice when *huge* amounts of data need to be transferred

Exercise. Read about the mechanism of choice to transfer the data describing the first ever image of a black hole (April 2019)

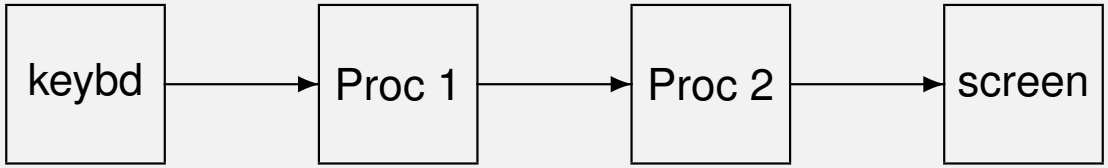
□

Inter-Process Communication

Pipes

A *pipe* is an IPC mechanism provided by some OSs

Conceptually, a pipe connects two processes together, taking output from one and feeding it as input to the other



This might be part of a larger pipeline

And the pipes go via the kernel, not directly between processes

Inter-Process Communication

Pipes

Pipes have a fixed size: 4096 bytes is common

A writes to the pipe, B reads from the pipe and they do so independently of each other

This is like the way we pass data via files

But pipes also provide synchronisation

Inter-Process Communication

Pipes

A writes bytes into the pipe: if the pipe gets full, A is blocked by the OS until space is freed up by B reading some

B reads bytes from the pipe: if the pipe gets empty, B is blocked by the OS until bytes are available by A writing some

Thus the scheduling of A and B can be affected

Bytes are read out in the same order they were written in: FIFO

Note there are two kinds of communication here: (1) the data, and (2) synchronisation on production/consumption of the data

Inter-Process Communication

Pipes

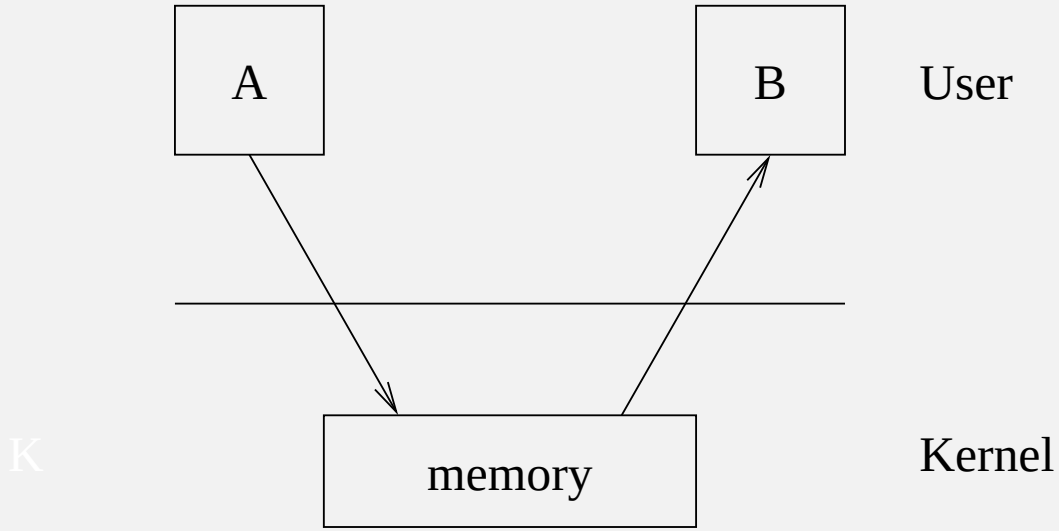
A pipe is implemented as a buffer (chunk of memory) held by the kernel, not directly accessible by user processes

A write to or read from the pipe involves a syscall

This is how the kernel can control blocking A and B, making sure A does not overfill the buffer and making sure B is not reading data that is not there

Inter-Process Communication

Pipes



Implementation of a Pipe

If A wants to write to the pipe, it makes a system call: the kernel can check for space in the buffer and block A if necessary

Symmetrically for B reading from the pipe

Inter-Process Communication

Pipes

Pipes are supported well by Unix and are very easy to create and use when using a shell

```
% ps | sort
```

The `%` is the shell prompt; `ps` is the "list processes" command; `sort` is a sorting program; the `|` is the notation for a pipe in this shell

So this displays a sorted list of processes

Aside

A *shell* is just a program that waits for you to type something and then possibly creates some new processes according to what you typed: the *command line* interface

Popular with Unix derivatives, unpopular with Windows derivatives

Inter-Process Communication

Pipes

Pipes are supported well by Unix and are very easy to create and use when using a shell

```
% ps | sort
```

The `%` is the shell prompt; `ps` is the "list processes" command; `sort` is a sorting program; the `|` is the notation for a pipe in this shell

So this displays a sorted list of processes

Inter-Process Communication

Pipes

Pipes are also easy to create within programs: see the POSIX function `pipe`

Aside

POSIX is a *library* standard: it contains a list of standard functions that provide a simple and uniform front end to OS syscalls (amongst doing useful other things) and describes their expected behaviours, e.g., `open`, `close` and `sqrt`

This helps portability between OSs by hiding some OS specific details (e.g., details of syscalls)

Unix derivatives are usually mostly compliant, Windows less so

Warning: remember some people regard such systems libraries as part of the OS

Even though they live and operate in user mode

Pipes

A typical sequence in a program is for a process to create a pipe then create a child process (i.e., ask the kernel to create a pipe then ask the kernel to make a new process)

(After a bit of technical fiddling) the pipe is now ready to use for IPC between parent and child

Inter-Process Communication

Pipes

Pipes are

- simple and efficient
- easy to use from programs and from a shell
- a powerful way of combining processes and programs
- used a great deal

Inter-Process Communication

Pipes

But also

- are unidirectional
- technical detail: are only between *related* processes. Often one is the parent of the other
- can trivially create deadlocks if you use them carelessly (A creates a child process B with two pipes A$\rightarrow$B and B$\rightarrow$A...)

Inter-Process Communication

Pipes

Pipes are so useful there have been a couple of extensions:

- Named Pipes: these can can be shared by unrelated processes (but have the naming problem that IPC using files have)
- Sockets: pipes between processes on different machines. The basis of the Internet

Inter-Process Communication

Sockets

A socket allows bidirectional IPC between two processes (pipes are unidirectional for mostly historical reasons)

The processes may be on the same or widely remote machines

The technical issues behind implementing sockets are clearly much more complicated than basic pipes, but they present the same kind of FIFO, byte oriented, blocking channel

We shall see some of those issues later in this Unit

A lot of the modern world is built on top of sockets!

Inter-Process Communication

Shared Memory

Next: shared memory

In early computers, all memory was shared between processes: one process could easily write to the memory allocated to another process

This is generally a bad idea, so is now prevented by the kernel (recall MMUs and read/write flags)

On the other hand, access to memory is very fast, so we might want to use it for IPC

Just like using files: A writes to memory, B reads from it

Again, this goes against the original design of an OS, so must be carefully set up and controlled

Inter-Process Communication

Shared Memory

And, also just like files we have the issues of

- Which area of memory to use? A well-known area, or per-process areas?
- How does B know when data has arrived? Memory is “always there” unlike files which can be created and removed; so when polling memory it can be hard to know if you are reading the data you want or some previous junk that happened to be lying around
- So A might write a special value to a specific memory location to flag that the data is complete; but again B must poll this location to see when this is done
- The memory protections must be set properly to allow only the authorised processes to read or write it

Inter-Process Communication

Shared Memory

The speed of shared memory means that it is very good for IPC, as long as it is supported by further mechanisms like signals or semaphores to flag when data is ready

More on shared memory when we get to memory management

Exercise. Compare shared memory and pipes

Inter-Process Communication

Signals

A signal is a software equivalent of a hardware interrupt: they can be sent to a process by the kernel or by a process

Also: *raised* and *initiated*

Just like a hardware interrupt, when a process receives a signal, it stops what it currently doing and goes off to execute a *signal handler*, in direct analogy with an interrupt handler

Handled within the user program: the signal handler is just some code in the program, written by the programmer

Inter-Process Communication

Signals

Again, what we lazily say is not quite what really happens

When a signal is raised (which needs a syscall) the OS takes over and notes the signal in the receiving process' PCB

When the OS next runs that process, it jumps to the signal handler code within the process, rather than to the place where the process was preempted

Inter-Process Communication

Signals

A process can send a signal to another process (or even itself) that has the same owner (same userid)

The normal restrictions on userids applies: only root can send a signal to another user's process

But remember that all signals are delivered to processes via the kernel

The kernel can also itself initiate a signal, e.g., a signal to indicate activity of a peripheral

Naturally, the kernel can send signals to any process

Inter-Process Communication

Signals

Use the POSIX function `kill()` to send a signal in a user program

And functions like `signal()`, `sigaction()`, `sigaddset()` and more to manage signals

Inter-Process Communication

Signals

A signal is in essence a flag (a single bit), but there are many different types of signal, e.g., HUP, INT, SEGV, KILL, PIPE, etc., to indicate different kinds of events

A process can, to some extent, choose how to react to a signal

- ignore it: that is, inform the kernel that it does not want to receive a particular signal, so the kernel will not note delivery in the PCB as above
- accept it and act on it: run the signal handler code
- suspend (voluntarily relinquish)
- terminate

Inter-Process Communication

Signals

Some signals cannot be ignored or otherwise acted upon, in particular a KILL signal, which always terminates the process (i.e., that process will be moved to the exit state)

This is why signals are regulated by the kernel; a user can kill their own processes, but not others'

And root (administrator) can kill any process

Signals are *asynchronous*, that is they might arrive at any point during the running of the program

Programs that use signals must be written accordingly

(And they will always arrive at the most inconvenient time. . .)

Inter-Process Communication

Signals

There exist default signal actions for each type of signal, but a program must include its own handler functions if it wants to do something other than the default action when it receives a signal

When a process receives a signal, it stops what it is doing, saves its state and calls the signal handler

Exercise. Rewrite the above sentence to describe what really happens. (No program you have ever written includes code to save its state!)

So this is very analogous to an OS interrupt

If and when the handler code finishes, the process continues from where it was interrupted

Inter-Process Communication

Signals

Example signals

```
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     16) SIGSTKFLT
17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU
25) SIGXFSZ     26) SIGVTALRM   27) SIGPROF     28) SIGWINCH
29) SIGIO       30) SIGPWR      31) SIGSYS      34) SIGRTMIN
35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3  38) SIGRTMIN+4
39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12
47) SIGRTMIN+13 48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14
51) SIGRTMAX-13 52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10
55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7  58) SIGRTMAX-6
59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

Inter-Process Communication

Signals

- INT: a general interrupt
- ILL: sent by the kernel to a process when it has tried to use a privileged or non-existent machine instruction
- KILL: non-ignorable terminate
- SEGV: sent by the kernel to a process when it has tried to access memory it shouldn't
- ALRM: a timer signal (*not* the preemption timer!)
- USR1 and USR2: signals for the use of user programs
- RT: a large number of signals are provided for real-time processing
- Signals 32 and 33 are not used in the OS in this example

Inter-Process Communication

Signals

Each signal has a default action

- INT, ILL, ALRM, SEGV, USR1, USR2: exit the program
- TSTP: suspend
- CONT: continue after a TSTP
- CHLD: ignore

Most default actions can be overridden by the program, some, notably KILL, cannot

Inter-Process Communication

Signals

Signals are

- Fast and efficient
- Asynchronous
- Used a very great deal
- Only transmit a small amount of information
- So often are used in concert with other IPC mechanisms
- Are a bit fiddly to program correctly

Inter-Process Communication

Semaphores

The action of process A waiting for process B to finish something before A can continue is very common

E.g., waiting for data to be written to an area of shared memory

It is a very simple form of IPC

Signals can be used, but an alternative is to use a *semaphore*

A signal is appropriate when you want to continue computing on something else while waiting; a semaphore is for *pausing* and waiting (i.e., blocked)

Inter-Process Communication

Semaphores

Invented by Dijkstra, semaphores have been used widely for may years

A semaphore is a variable whose value can only be accessed and altered by two operations *V* and *P* (Dijkstra is Dutch)

Alternative names are: signal and wait; post and wait; raise and lower; up and down; lock and unlock and others

Inter-Process Communication

Let S be a semaphore variable, usually residing in a chunk of shared memory

Start with $S = 1$

$P(S)$:
if $S = 1$ then set $S = 0$
else block on S

$V(S)$:
if one or more processes are blocking on S then allow one to proceed
else set $S = 1$

(There are many technical issues we are ignoring here. . .)

Inter-Process Communication

Semaphores

For synchronisation:

```
P(S)
...modify a resource...
V(S)
```

```
P(S) # wait for resource
...use resource...
V(S)
```

The second process will wait until the first has done a V to signal the resource is ready

Inter-Process Communication

Semaphores

If multiple processes attempt a $P(S)$ simultaneously only *one* will succeed and continue; the others will be blocked

So if we have code like

```
P(S)wait(S)
some code
V(S)signal(S)
```

being run by multiple processes using the shared semaphore S, only one process can execute the code at a time; the others will be blocked and get their turn later

More suggestively using names signal and wait (**not** the same signal as in signals, earlier!)

Inter-Process Communication

Semaphores

Generally, the code would be to access some shared resource (often shared memory, e.g., B shouldn't read until A has finished writing), so the semaphore makes sure only one process can access the resource at a time

The protected code is called a *critical section*: it is critical that only one process runs it at a time

```
P(S)                    P(S)
...resource...          ...same resource...
V(S)                    V(S)
```

To be effective, all accesses to the resource must be protected by the semaphore

Inter-Process Communication

Semaphores

This is a *binary semaphore*, as it take just two values, 0 and 1

There is a simple generalisation to a *counting semaphore*

Start with $S = n$

$P(S)$:
if $S > 0$ then set $S = S - 1$ else
block on S

$V(S)$:
if one or more processes are blocking on S then allow one to proceed
else set $S = S + 1$

This allows no more than n processes into the region at once

Inter-Process Communication

Semaphores

Semaphores were first used within OS kernels to protect shared resources but can be used in user programs to protect resources there, too: for example, a chunk of shared memory (e.g., shared memory IPC)

Inter-Process Communication

Semaphores

Correct implementation of user mode semaphores is very hard

We have to ensure that it works even if

1. the process is rescheduled in the middle between the test and the decrement of the count
2. there are multiple parallel processors accessing the semaphore simultaneously

Exercise. Read about the implementation of semaphores

Inter-Process Communication

Semaphores

Semaphores are widely used

- each semaphore only needs a few bytes of shared memory
- they are small and fast given hardware support
- and OK in software
- used both in OSs and user programs to protect critical resources
- and are widely available in POSIX libraries

Inter-Process Communication

Semaphores

On the other hand, semaphores are a very low-level mechanism and it is easy to cause deadlock

Suppose we have semaphore S_1 protecting file F_1 and semaphore S_2 protecting file F_2. Process A wants to read from F_1 and write to F_2, while process B wants to read from F_2 and write to F_1

To make things consistent in the read/writes, both processes must grab both semaphores

- Process A grabs semaphore S_1
- Process B grabs semaphore S_2
- A tries to grab S_2 and blocks
- B tries to grab S_1 blocks

Inter-Process Communication

Semaphores

Exercise. Identify the four conditions for deadlock in the above

Exercise: use a counting semaphore to solve the Dining Philosopher's problem

□

Inter-Process Communication

Application Level

Finally we look briefly at IPC at the application level, namely high level mechanisms for passing data between processes

Again, at base, this goes via the kernel (often using a mechanism we have already mentioned, e.g., pipes or shared memory, assisted by signals or semaphores), but the idea is to provide high level constructs so we (as programmers) don't have to be bothered with details

These are always implemented by system libraries and a fixed interface presented to the programmer regardless of the underlying implementation

Inter-Process Communication

Application Level

These came back into prominence with windowing GUIs where it was found necessary for applications to communicate with each other and with the system

Cut-and-paste and drag-and-drop are basic examples, where structured information needs to pass between components (processes)

The idea is much older than GUIs, of course: originally this was called a *software bus* in analogy with hardware buses that connect hardware components

Inter-Process Communication

Application Level

Popular implementations include

- CORBA (Common Object Request Broker Architecture)
- DCOP (Desktop COmmunication Protocol)
- Bonobo (based on CORBA)
- D-Bus
- COM (Component Object Model) and variants, including .NET

Inter-Process Communication

Application Level

These try to be language independent with each language having a set of *bindings* (standard functions) to access them

They focus on passing objects between components

So they need a standardised method of representing the data/objects in a *message*

This is called *message passing*, another important paradigm for IPC

Inter-Process Communication

Application Level

These kinds of framework tend to be very complicated as they need to support a wide variety of communications between a wide variety of components

For example, passing a picture from a program written in C to one written in Java

Exercise. Read up on some of these

Inter-Process Communication

So there are many IPC mechanisms: they are not mutually exclusive

Any of these mechanisms can be used in tandem

- Our program might employ D-Bus to pass data from one application to another (e.g., cut and paste)
- D-Bus might use pipes to communicate between processes
- And pass a filename between them
- and the data is communicated in the file

Inter-Process Communication

So, which IPC mechanism to choose?

As always, it depends on the application

The best way to choose is to have lots of experience of using them

- The level your program is at: low or high?
- The amount of data to be communicated: just a bit or a huge datafile?
- What is available?
- What your boss tells you to use
- and so on

Memory

We now turn to the next major topic: memory management

In the earliest computers the purpose of memory management was to share out a very limited resource, but it was soon found that inter-process *protection* was vital

Both needs are still true, particularly the limited aspect: you might have 2GB in your PC, but it's not enough!

Gates' Law: programs double in size every 18 months

(Really Wirth's Law: Software is decelerating faster than hardware is accelerating)

Memory

Physical Memory

We first consider how processes (code and data) should be laid out in memory

This is called *physical* memory layout to distinguish it from *virtual* memory, which comes later

Memory

Physical Memory

Memory in a process might be allocated or freed at several points

- Allocation only at process initialisation. Called *static* allocation. Featured in the earliest OSs
- Allocation while the process is running. Called *dynamic* allocation. Early systems did not support this and you had to know in advance how much memory your process would need at initialisation
- Freeing while the process is running
- Freeing at process end

Memory

Physical Memory

But also the kernel needs memory:

- Allocation and freeing within the kernel. The kernel has to be dynamic otherwise it would be very difficult to get started, e.g., creating processor control blocks

Memory

Physical Memory

Early operating systems were not dynamic

So they could only run a fixed number of processes

And the processes were of a fixed size

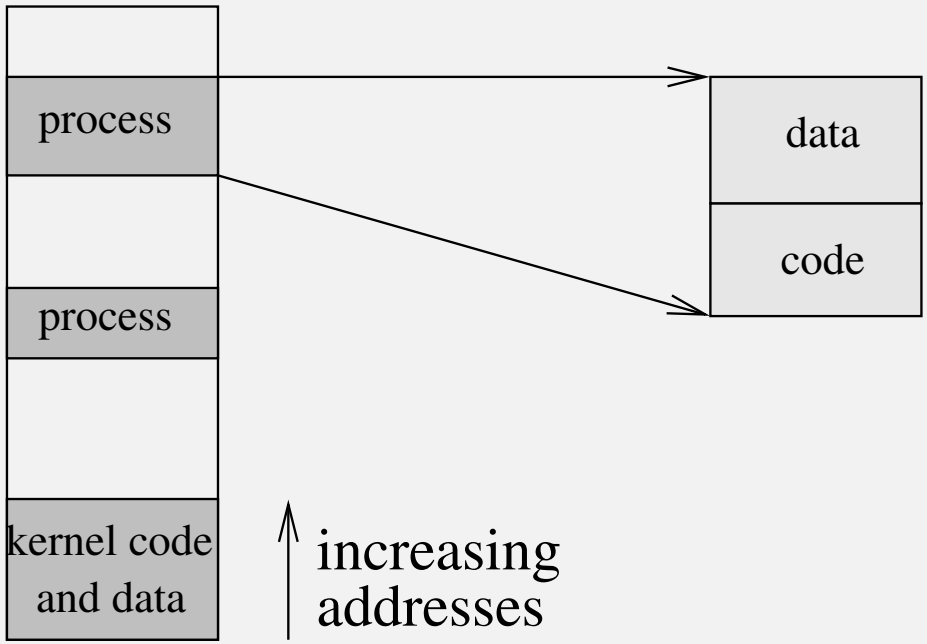
Reflecting this, early computer languages did not support dynamic allocation, e.g., FORTRAN, every array must be of a fixed size, declared in the source code

Dynamic allocation for both kernel and the processes was soon introduced in OSs, but computer languages took a while to catch up with the new facility

Memory

Physical Memory

Physical memory in an early computers looked something like this:



Memory Layout

Memory

Physical Memory

Remember the kernel itself needs code and data space

A gap above the kernel area allows for dynamic allocation of memory to itself

Memory

Physical Memory

But the earliest systems had no dynamic behaviour at all, both OS and programs were completely static

Again, some early languages (FORTRAN, again) did not have a stack, and thus no recursion

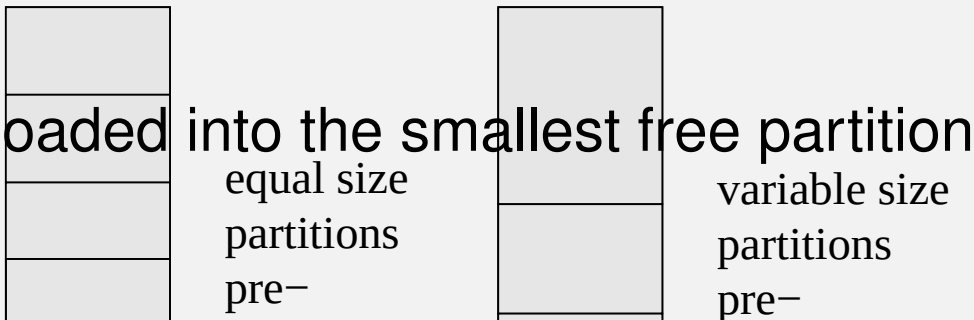
Memory

Physical Memory

Partitioning

The earliest and simplest memory layout is a static system called *partitioning*, where areas are allocated at boot time

A process is loaded into the smallest free partition it will fit into



Memory

Physical Memory

If you don’t have dynamic allocation even in the kernel (e.g., for allocating new PCBs), then having fixed partitions is ideal

Equal size is easy to implement, but usually causes wasted space when a process does not fill its allocation

And it can’t cope with larger processes

Variable size is not much harder to implement, but efficiency depends heavily on the choice of partition sizes as ideally they should match the expected process sizes

Memory

Physical Memory

Partitioning is a good arrangement if you only run a fixed set of applications that you know in advance, e.g., a stock manager plus a payroll system plus a employees record system

IBM's OS/360 (mid 1960s) had three partitions: one for spooling punched cards to disk; one for spooling disk to printers; and one to run jobs

Memory

Physical Memory

Overlays

In early systems, if a process was too big to fit in the memory allocated, the programmer could use *overlays*

This is where only *part* of the process code is loaded into memory at once: only partly *resident*

If a non-resident part of the process is needed, the programmer must know this and include code to load the needed part of the process into memory, overwriting a part of the process they do not need at the moment

If that part of the process is needed again later, the programmer has to reload the code

Memory

Physical Memory

This works, at the cost of some speed of execution, but only if you are an excellent programmer who can keep track on what parts of code are loaded at any particular time

A similar trick works with data: but with newly generated data you have to save it somewhere (e.g., disk) first, before overwriting it, so that it can be loaded back in later, when needed

This trick of swapping memory back and forth to the disk gets a big boost later

Memory

Physical Memory

We need to fit a process into a single contiguous chunk of memory as we can't spread it amongst several areas since

- it will be very complicated for the OS to keep track of which areas of memory are allocated to which process
- more importantly, you can't split code up in this way, having one instruction in one place and the next instruction somewhere else entirely
- similarly for data: we will have to keep track of what data is where

But when we come to virtual memory we shall see that exactly this *is* possible with modern hardware!

Memory

Language Support for Dynamic Allocation

These days dynamic allocation is common in languages

- Implicit memory management, e.g., Java. Where the language controls the creation and deletion of objects

```
bigobject x; // memory is allocated for x
x = foo();   // that memory is now inaccessible
```

- Explicit memory management, e.g., C. Where the programmer controls the creation and deletion of objects (`malloc` and `free`)

And several other approaches!

Dynamic Partitioning

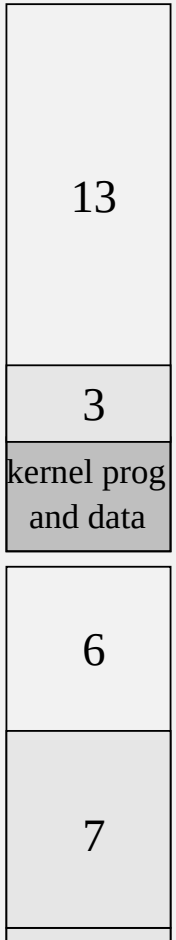
So we need to to be dynamic: create and allocate a partition as needed

A lot more complicated to implement, but this allows the process (i.e., the job submission) to say how big a partition it needs and the OS allocates just that

Memory

Physical Memory

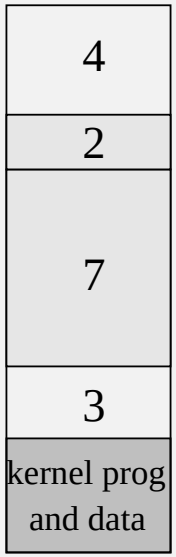
We can allocate sequentially, moving up memory



Memory

Physical Memory

The problem is when a process ends and we get memory back: it creates holes



We have space enough to run a process of size 5, but nowhere to put it

Memory

Physical Memory

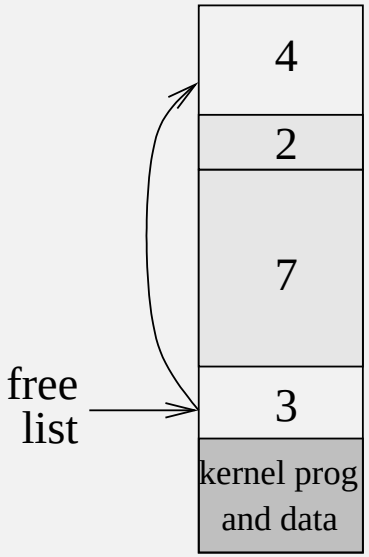
This is a general problem, called *fragmentation* and is very difficult to solve effectively

The more processes come and go, the worse the fragmentation gets

Memory

Physical Memory

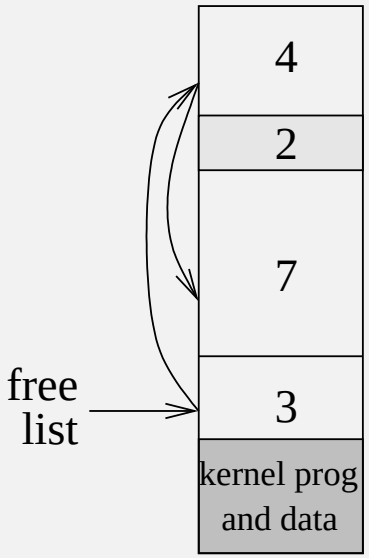
We need to keep a list of free blocks so we can track free space: a *freelist*

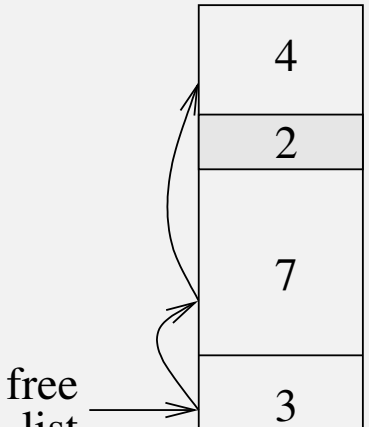


Memory

Physical Memory

When a block is freed, put it in the freelist. It helps to keep the freelist sorted in address order:

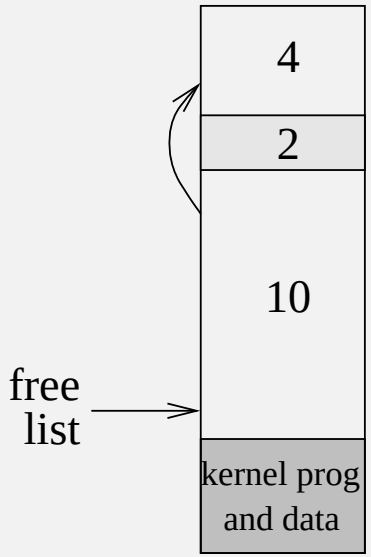




Memory

Physical Memory

Slightly more clever is to *coalesce* physically adjacent blocks



Memory

Physical Memory

When we want some space, we search the freelist

We don't want to waste space, so after choosing a big enough block we slice off the chunk we need and return the unused part to the freelist

But there might be several blocks on the freelist that we could use: which one to choose?

Strategies for choosing blocks include:

- Best Fit. Find the *smallest* available big enough hole. Slow as we always have to search the entire freelist and results in lots of small fragments that are effectively useless as they are too small to be allocated

Memory

Physical Memory

- First Fit. Use the *first* available big enough hole. Initially faster than Best Fit and tends to leave larger and more useful fragments. But fragments tend to be created near the front of the freelist, so we have to search further and further each time
- Worst Fit. Find the *biggest* available big enough hole. Strangely this works out better than you think. Slicing chunks off bigger blocks tends to leave larger fragments that are more likely to be useful. Marginally faster than Best Fit as we have larger and therefore fewer blocks in the freelist to search through

Memory

Physical Memory

- Next Fit. Continue looking from where we last allocated and take the next available big enough hole. Fast, and improves on First Fit by spreading small fragments across memory
- And many others

There are plenty of other memory management systems (e.g., Buddy memory allocation; Slab allocation; etc.) targeting the fragmentation problem

Allocation is **still a problem** in current machines where certain kinds of hardware need large contiguous chunks of physical memory, e.g., GPUs

Memory

Physical Memory

Note that fragments are created in two ways:

- when carved off a bigger block in the allocation
- when returned at process exit

The second generally gives us larger fragments, but both need to be addressed

Memory

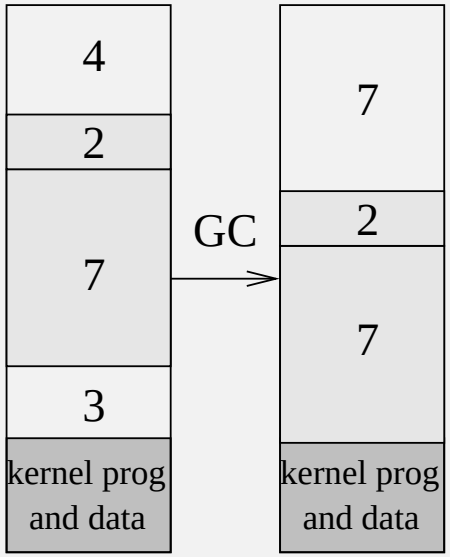
Physical Memory

If we can't find a big enough free space, we can consider *compaction* of memory using a technique called *garbage collection*

The OS stops all running processes (i.e., stops scheduling processes); shifts their code and data around to close up the gaps; then lets the processes continue (i.e., starts scheduling again)

Memory

Physical Memory



Memory

Physical Memory

GC is not often used in general-purpose OSs

- it is a very expensive operation to move all these blocks around
- this takes a lot of time away from running of processes
- the pause while things are moved is bad for interactive and real-time behaviour
- the erratic nature of when GCs are needed leads to unpredictable behaviour from the OS
- given the right kind of hardware support, better solutions completely avoiding the need for GC are possible

Memory

Physical Memory

GC *is* successfully used in user languages, e.g., Lisp, Java

There are ways of implementing GC to avoid the stop-and-copy (ephemeral GC), or mitigating the overhead (generational GC) but even so it is not popular for OSs

Memory

Physical Memory

Notice that all these rely on *relocatable* processes, namely ones that don't refer to specific locations in memory

Note that the code in a process now can't use an absolute "jump to memory location 42", but must use a relative "jump by *n* bytes"

And similarly for referencing data in memory

Unless we know in advance where our process is going to be placed in memory, we cannot have code that has fixed absolute addresses in it

These issues develop when we move to *virtual* memory later, but in general code should not assume it lives in a given place in memory

Memory

Physical Memory

So what happens when we can't find a suitable free space for a new process (even if we have GC)?

We may choose not to admit the process in the first place

Another possibility is the option of killing existing processes: we usually don't want to and only if the new allocation is for a process that is sufficiently important

Better is to *preempt* memory: take it away from one process and give it to another

Memory

Physical Memory

Remember that preemption takes a resource away from a process and returns it later in the same state

For memory this means the bits in the memory when it is returned are unchanged from what they were when it was taken away

Even though that memory has been used by some other process and written its own data or code into it

Memory

Physical Memory

We can preempt a process and copy the contents of the memory it occupies to somewhere else: usually disk

Note that only the data need to be saved: the code is already on disk in the file that contains the program

Copying to disk is a (relatively) very slow operation: even the fastest disks are quite slow

Even solid state disks

So this kind of memory preemption has a large overhead

This is a tradeoff of speed (time spent copying to and from disk) against process size (memory allocation)

Memory

Physical Memory

Swapping

The simplest case is preemption of the memory of an entire process

When a process makes a request for an allocation that the OS cannot immediately satisfy the OS can try *swapping*

This is where one or more other processes are selected by the OS and they are copied out to disk to make space

The best choice is usually a blocked process that couldn't have been run right now anyway

Memory

Physical Memory

When a swapped process is scheduled again it must be copied back by the OS into memory first

Which might require swapping out something else to make room

Data is retrieved from where it was saved, while code is copied back from the original program file — this is why some OS's don't like you deleting programs while they are running

Memory

Physical Memory

This differs from overlays in that it is the OS that does the swapping, not the process itself

This makes it transparent to the process and the programmer doesn't have to think about it

. . . but they should as swapping is very time consuming, and slows down the speed of execution of programs immensely

A good programmer will try to avoid the need for swapping by requesting memory allocations carefully

Something that often is forgotten these days!

Memory

Physical Memory

The OS will take swapping into account when scheduling

There is a clear interaction of scheduling and swapping processes: each will affect the other

Memory

Physical Memory

Variants:

- Only one process ever in memory, swapped as a whole when scheduled: simple, and used on very early systems
- Swapping of processes: only marginally harder, and fits well with a partitioning system and fits well with scheduling
- Swapping *parts* of a process: not so easy as the OS has to work harder to determine which parts of a process's code or data might not be needed in the near future

Memory

Physical Memory

The OS still has the difficult task of deciding which process or processes to swap to make room: e.g., one large one or two small ones? I/O intensive or CPU intensive?

An I/O intensive process is less likely to need to be scheduled soon; but it would a like fast response when it is needed and not wait for a slow swap back into memory

A CPU intensive process would like to be scheduled often; but is not so sensitive to a delay through being swapped

Paging

This is all augmented by the idea of *paging*

Paging is similar to swapping, but simpler in concept

And much harder in the hardware required

To describe paging we must first go back to pages

Memory

Virtual Memory

A big problem is memory fragmentation due to the irregular sizes of processes/partitions

So to fix this we chop everything up into equally sized chunks

Recall (from memory protection) a *page* is just a contiguous area of memory: e.g., 4096 bytes

Hardware is designed so copying pages in and out of memory from disk is as efficient as possible

Memory

Virtual Memory

Next, we introduce *virtual* vs. *physical* addresses

A physical address is what we are used to, just a numbering of the actual bytes in the system from 0 to n

A virtual address is a per-process fictional address

The user process sees only the virtual addresses: the system will translate them on the fly into physical addresses

Memory

Virtual Memory

The OS has tables, one per process, called *page tables*, that contains the virtual-physical address mappings for each page in each process

For example, with a page size of 4096 bytes, address 12298 is 10 bytes from the start of page 3: $12298 = 3 \times 4096 + 10$

Under the entry for page 3 in the page table for this process we might find the number 7, meaning physical page 7

So virtual address 12298 **in this process** refers to physical byte $7 \times 4096 + 10 = 28682$

Memory

Virtual Memory

In another process, virtual page 3 could be mapped to physical page 42

And then the same virtual address 12298 **in this process** refers to physical byte $42 \times 4096 + 10 = 172042$

The *same* virtual address in different processes is mapped to *different* physical addresses

We use pages, of course, to make this translation manageable

Memory

Virtual Memory

The table only contains entries for pages that are actually in use by that process: this keeps the tables to a reasonable size

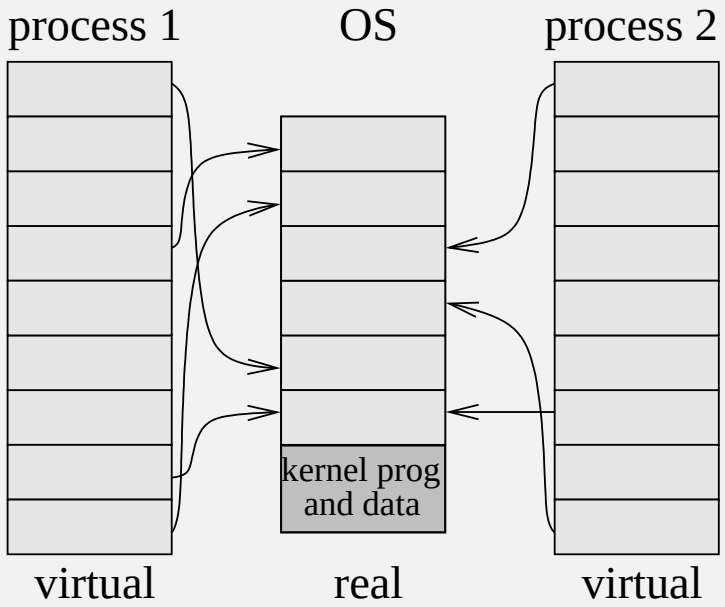
V page	P page
3	7
4	9123
5	121
10	1232
etc.	

Note: page tables contain page *mappings*, not pages

Note: though still called “tables”, in modern OSs they are likely to be more sophisticated datastructures, such as trees

Memory

Virtual Memory



Every process gets its own complete and separate address space, mapped into the physical address space

Even for the same userid: this is usually what you want, protection of one process from another

Memory

Virtual Memory

Where are these page tables?

In memory, of course: and a link to the table is kept in the process's PCB

But it sounds like, *on every memory access*, we have to do (a) a memory read of a page table to find the V to R mapping and then (b) a calculation to get the physical memory location and then (c) a memory access to the physical address we wanted

- every data read
- every data write
- every execute of an instruction

This is clearly not sensible as it would be very slow

Memory

Virtual Memory

So, to be practically useful, this is supported by a piece of hardware called the *translation lookaside buffer* (TLB), part of the memory management unit (MMU)

The TLB maintains its own copy of *a few* of the virtual-physical mappings from the page table of the current process and can translate very quickly between them

Memory

Virtual Memory

To repeat that: the table in the TLB is a *small subset* of the OS's page table mappings of the current process

Only a small subset as TLB memory is very limited since it is very expensive to make memory that runs fast enough to make this mechanism useful: it contains perhaps just a few dozens of the virtual to physical mappings

Note (again): the TLB contains copies of the page *mappings*, not pages

Memory

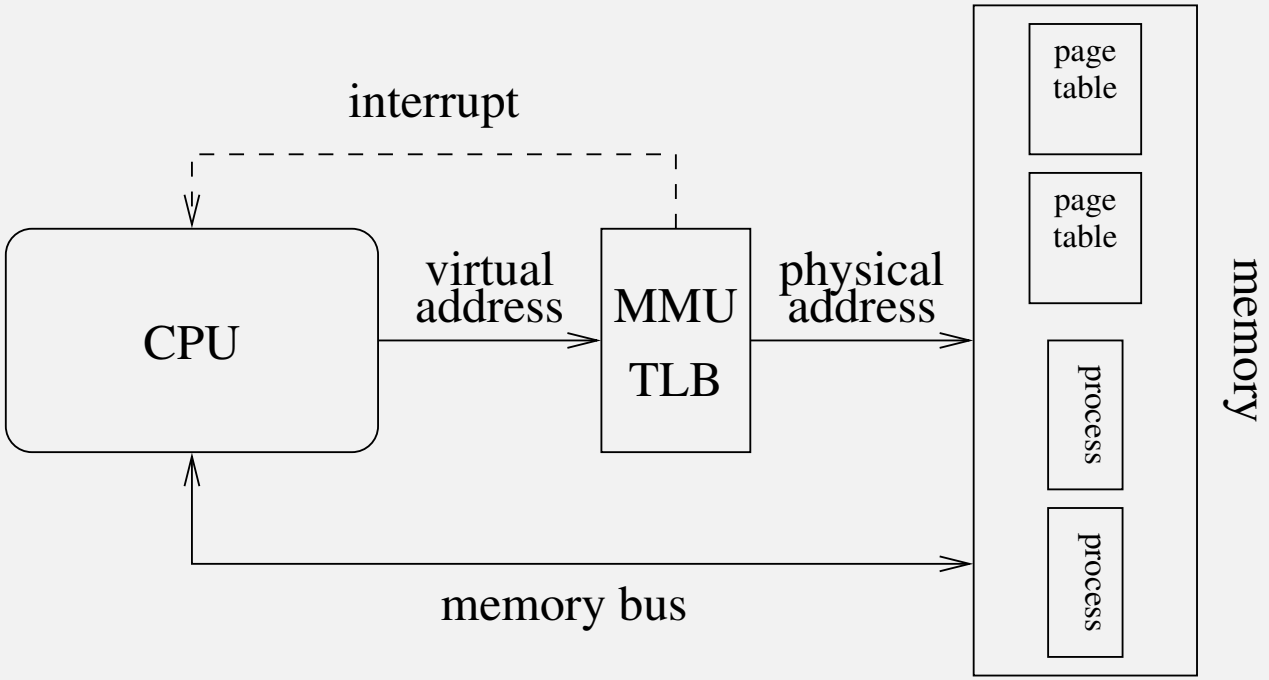
Virtual Memory

The Intel Nehalem architecture has a 64 entry data TLB (and a 512 entry level 2 TLB); and a separate 64 entry instruction TLB

Note that 64 entries typically corresponds to an area of $64 \times$ 4k page $=$ 256k bytes, so while not huge, this isn't so bad as it might seem as first

Memory

Virtual Memory



The MMU and TLB are often physically part of the CPU package, for speed of access

Memory

Virtual Memory

When presented with an address from the CPU the TLB first looks the virtual page up in its table. If it is there is—a *TLB hit*—the memory access goes ahead at full speed using the physical address computed from the real page index found there

If there is a *TLB miss* then it has to work a bit harder

There are two popular techniques used

Memory

Virtual Memory

In a *hardware managed* TLB, the CPU/TLB itself stops what it is doing and searches for the page number in the page table (in memory) for the current process: this is called a *page walk*

If it finds it, it installs it in the TLB table and carries on with the memory access

The OS is not involved in the page walk, it is purely hardware

Memory

Virtual Memory

The second technique, a *software managed* TLB, simply raises a *TLB miss* interrupt on a TLB miss

The OS then has to do the page walk

Memory

Virtual Memory

This deals with the case of when the requested page has already been allocated by the OS to the current process, so there is an entry in the page table for the page walk to find

In either software or hardware case, if the requested virtual page is not yet allocated by the OS to the process and so not in its page table, the OS needs to allocate a page

Memory

Virtual Memory

A hardware managed TLB will now raise a *page fault* interrupt to pass control to the OS

A software managed TLB is already running the OS

The OS allocates a physical page, installs the new page mapping into the page table for that process for that page and writes the relevant page mapping into the TLB

(When the process is rescheduled) the memory access can then proceed

Memory

Virtual Memory

Of course, the OS may choose not to allocate a page and it would likely then send a segmentation violation signal to the process

x86 and ARM processors have hardware managed TLBs

SPARC and MIPS are software managed

Terminology warning: a TLB miss when the page is already allocated and indexed in the page table is sometimes called a *minor* or *soft* page fault; while a miss on an unallocated page is a *major* or *hard* page fault

Memory

Virtual Memory

Speed relies crucially on the TLB containing a good proportion of the addresses currently being used: if a process writes wildly all over memory we are guaranteed to get TLB misses and slow memory access: lots of TLB misses and page walks or page fault interrupts

Fortunately, most well-written programs behave sensibly and tend to use the same addresses over and over, meaning lots of TLB hits

After a while, the TLB settles down, caching the indices of the pages the process is using, the *working set*

Memory

Virtual Memory

Note that a page fault can cost a lot of time

Register access	1 cycle
(L1 memory cache hit	$\approx$ 2 cycles)
(L3 memory cache hit	$\approx$ 50 cycles)
Main memory access	$\approx$ 200 cycles
TLB miss (page in memory)	$\approx 10,000$ cycles
Page fault (page on disk)	$\approx 1,000,000,000$ cycles

These are very rough figures and are the combined overhead of OS operations and memory architecture

Memory

Virtual Memory: Paging

At last we can talk about paging

Paging is copying pages to and from disk

Suppose there is a memory access

If there is a TLB hit, the memory access continues at full speed

On a TLB soft miss the usual case is that the page is still resident in physical memory (not been swapped out), so it's just a matter of updating the TLB to refer to it by copying the page table entry into the TLB

And then the access can continue as for a hit

Memory

Virtual Memory: Paging

But if the page has been swapped out (“paged out”), then its contents need to be read back from disk: thus a large cost in this case

Memory

Virtual Memory: Paging

If there is a TLB miss when the TLB table is full, the OS must choose which mapping to remove from the TLB to make space for the new one

Usually a *least recently used* (LRU) strategy is used as pages that haven't been touched recently often are not needed in the near future: this is called *temporal locality*

Thus the TLB hardware must also keep track of *when* pages are used, e.g., using a timestamp

Again, temporal locality is a feature of many programs, but it is easy to write programs that confound the LRU strategy

Memory

Virtual Memory: Paging

The OS may use the opportunity of a TLB miss to choose some pages to swap out

Note there are two separate issues here:

- which entry in the TLB to remove when the TLB table is full
- which page in physical memory to swap out when physical memory is full

They are related in that an infrequently used TLB entry implies an infrequently used page; but conversely an infrequently used page is probably not in the TLB table at all

Paging Strategies

Many strategies exist for choosing pages to evict (swap out)

- Random. Pick a random page. Simple and better than you think
- FIFO. First in first out. Poor, as pages that have been around for a long time tend to be the ones that are needed
- LRU. Least Recently Used. Good, but needs to keep track on a when a page was last used (different from the TLB page timestamp)
- LFU. Least frequently used. Increment a counter on the page on each access; remove pages with low counts. Not so good as pages just brought in because you need them tend to have low counts
- And so on.

Memory

Virtual Memory

The first time a (virtual) page is touched by a process it will cause a (major) page fault

The OS needs to allocate a new physical page for this process

Allocation of new pages is facilitated by the fixed page size: just find *any* unallocated page in physical memory and set it as the physical page mapping from the requested virtual page

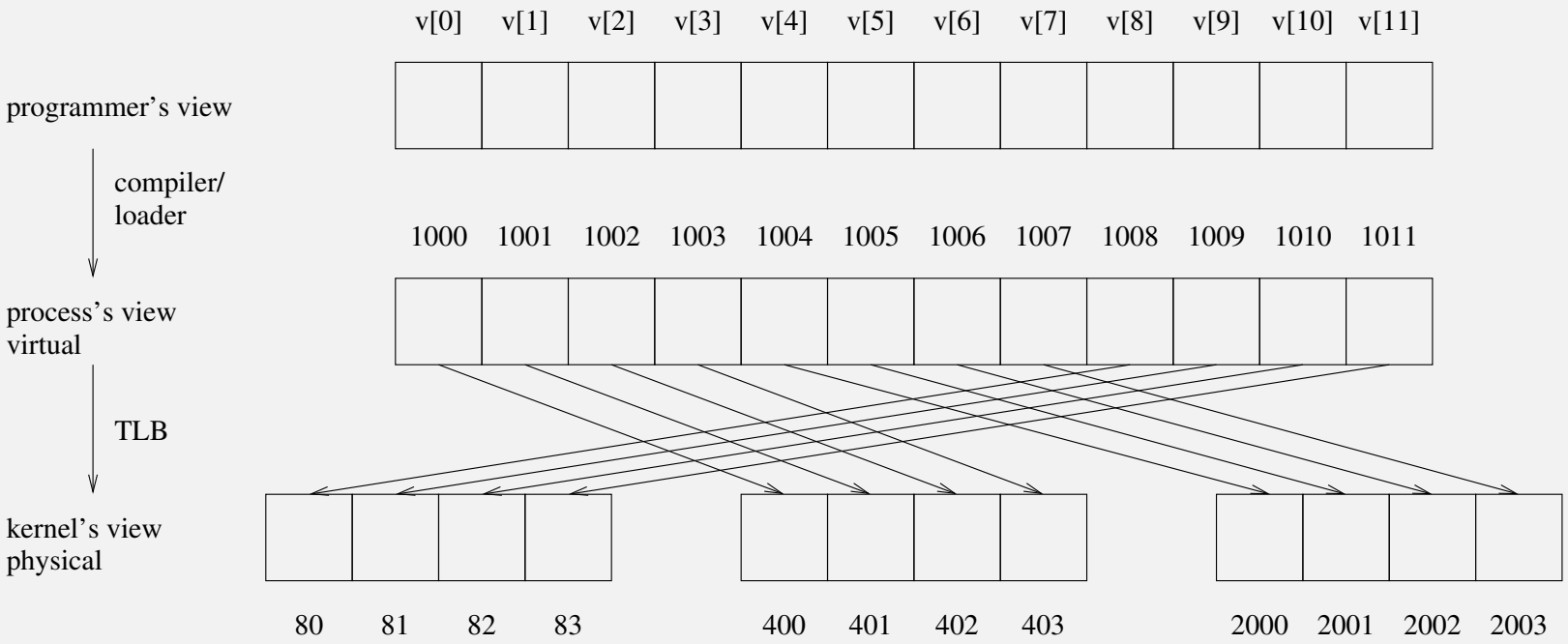
This simplification over earlier allocation methods is the big benefit of using pages

The first page on the freelist of pages is always suitable. No need to search, no size fit issues, no fragmentation issues

Memory

Virtual Memory

A single large datastructure (e.g., a vector, which you normally think of as a contiguous region of memory) in your process might actually be spread, in chunks, all over the place in physical memory



Memory

Virtual Memory

Similarly for code: a chunk of code spanning multiple pages may well be distributed all over physical memory

Code or data might be contiguous in the virtual address space, but definitely not contiguous in the physical address space

Memory

Virtual Memory

OSs often use *lazy* page allocation: don't allocate anything until the process actually accesses a page, so physical memory is only actually allocated on a page fault when we know we really need it

If process requests 10GB and only uses 1GB, this is not a problem: only 1Gb will be mapped in the page table

And the process's virtual size can easily be bigger than the physical memory size, either through unmapped or swapped pages

Memory

Virtual Memory

The cost is kept low though the use of the TLB, but remember a page fault is relatively expensive

And swapping is orders of magnitude slower still: we want to avoid swapping if at all possible

This is something in the hands of the programmer: don't use memory stupidly!

Memory

Virtual Memory

Note that the terms “paging” and “swapping” are near-indistinguishable these days

Swapping used to mean entire processes

Then *segments* (certain large areas) of memory

Now just pages are swapped

Note that when swapping a page back into memory, it doesn’t matter where in physical memory we put it : the page table/TLB ensures the process sees it in the same virtual place

Memory

Virtual Memory

TLBs are good but have limitations:

- they are quite small capacity, but usually big enough to be highly effective
- they rely on temporal locality to be effective: again OK for all but weird programs

Perhaps the biggest problem is whenever the OS does a *context switch*, i.e., chooses to schedule to run a different process. The TLB must be *flushed* as the incoming process will have a different set of virtual-physical mappings

The TLB will then be re-populated by a bunch of TLB misses and page faults as the incoming process runs

Memory

Virtual Memory

This is a major reason why a context switch is so expensive: on top of the cost for the save/restore of process state there is a large overhead for the subsequent TLB misses

Exercise. Read about the Spectre and Meltdown hardware bugs

Exercise. Think about the difference between vectors and linked lists in terms of virtual memory and TLBs

Exercise to think about: the page tables in memory can grow so large they need to be swapped themselves. . .

Memory

Virtual Memory

Examples. A “Hello world” program in C, Java, Python and Perl

	C	Java	Python	Perl
Resident size KB	430	16500	4300	1850
Minor Fault	150	3800	1200	530
Major Fault	0	0	0	0
Context switch	2	150	8	4

In Linux 3.11.10; 8GB memory

Numbers are approximate and vary on runs due to scheduling

Memory

Virtual Memory

So a page table is a list of pages a process has accessed and the relevant virtual-physical mapping. We have already seen every page also has some permissions attached:

- *read*: the process can read from this page
- *write*: the process can write to this page
- *execute*: the process can execute code on this page

Memory

Virtual Memory

If a process tries to access a page it does not have the appropriate permission for an interrupts happens and the OS sends a segmentation violation signal to the process

Even though, through the virtualisation, “all” memory is owned by the current process, it is still useful to have these permissions

This is so the process knows it is trying to read from/write to/execute some unexpected place in memory, rather than some place it should be. This catches many stupid programming errors

Further, permissions are useful when we have shared memory, too

Memory

Virtual Memory

Another big benefit of VM is the natural protection of one process from another: as all user mode memory accesses go though the TLB, the TLB will simply prevent it even being possible for one process to overwrite the memory of another

Or enable it if we want shared memory. Thus the TLB solves two big problems: memory protection and memory sharing

A process only sees the virtual address: it can access anywhere it wants and the TLB takes care of things

The kernel bypasses the TLB lookup and sees physical addresses, but can map back and forth for each process

Memory

Virtual Memory

Shared Memory

So now shared memory is very easy: just let the TLB do the mapping of virtual pages from different processes to the *same* physical pages

This allows *shared libraries* (`.so` in Unix; DLLs in Windows; `.dylib` in MacOS X/macOS)

Many programs need to do mundane stuff like read or writing to files, formatted printing, drawing on the screen and so on

So *libraries* of such code are provided that the programmer can use and not have to reimplement it all themselves

Memory

Virtual Memory

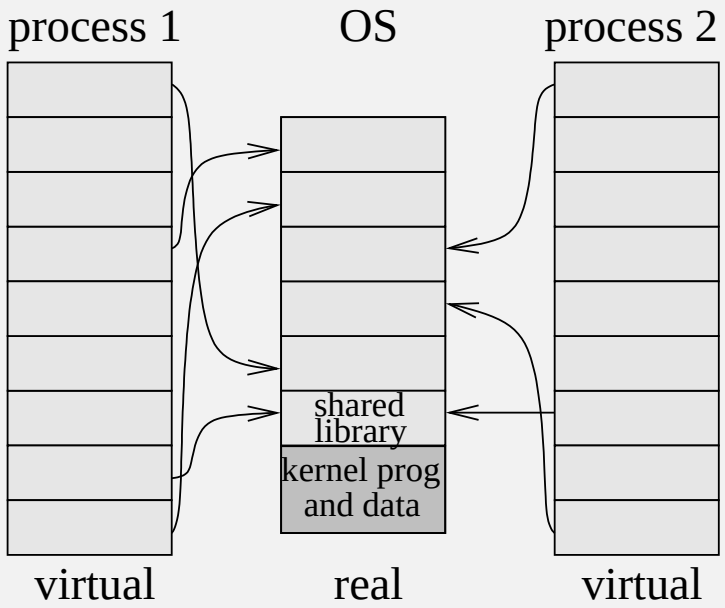
If 10 processes are in memory, each of them using the library `read` function, does that mean there are 10 copies of the code for `read` scattered about in memory?

Before the advent of shared libraries, yes

But now the use of virtual memory can let us *share* code between processes

Memory

Virtual Memory



Shared Libraries

Memory

Virtual Memory

Now the OS can load the code for `read` just once and direct all other processes to use that single copy

This reduces memory usage, reduces pages faults and has other beneficial properties (see caching)

This works well as all processes can share identical and unchanging code pages. But *data* in libraries couldn't be shared like this though?

Perhaps we would need a private copy of the data pages for each process, since if one process updates the data that would mess things up for another process also using that data

But there is another trick. . .

Memory

Virtual Memory

Copy on Write

Different processes can easily share data as long as they don't try to update it

Some data is read-only (e.g., a document containing an exam paper), so this could be stored in a page marked read-only, and this can be safely shared

Other data you *do* want to update (e.g., a document containing an exam answer template)

Such pages can be marked with another flag: *copy on write* — again, as long as the hardware supports this

Memory

Virtual Memory

With copy on write, a page is shared up until a process tries to modify it

At this point a page fault occurs and the OS takes over

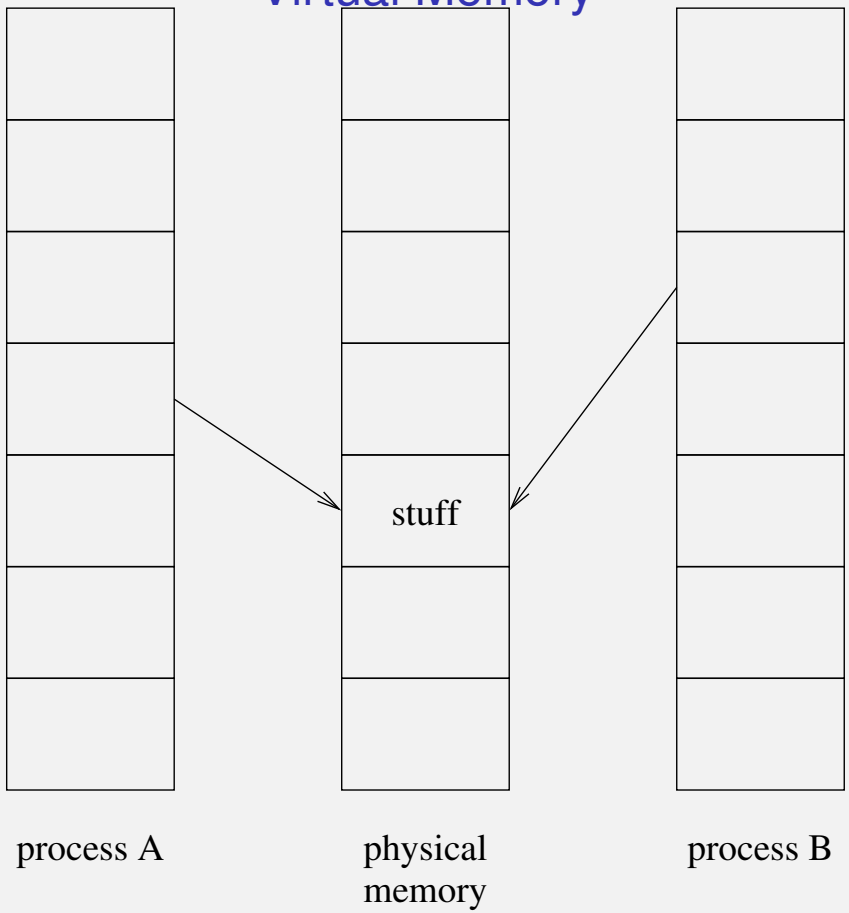
It makes a physical copy of the page and changes the page table and TLB for that process to point at the new copy

The write can then proceed on the private, unshared copy

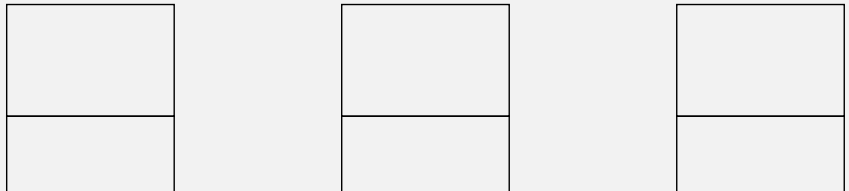
Other processes still see the original, unmodified data

Memory

Virtual Memory



Some data is shared; process B tries to update the data



Memory

Virtual Memory

This works really well for when a majority of data is shared with only a few changes here and there

And it only uses an extra amount of memory proportional to the size of the changes

So another reduction in memory use, page faults and so on

Memory

Virtual Memory

This is excellent, but comes at the cost of extra complexity as the OS now has to track if a shared page has already been loaded and where it is physically

And complexity in swapping as now it has to track which processes are using a page and it can't swap a page until no-one is using it

But this is offset by the fact we will need to swap less as we are using memory more efficiently

Other Tricks

OSs often keep a page full of zeros

If a process asks for a big block of zeroed memory, the OS will supply the appropriate number of virtual pages, all pointing at the single zeroed physical page: much faster than allocating and clearing out a load of physical memory

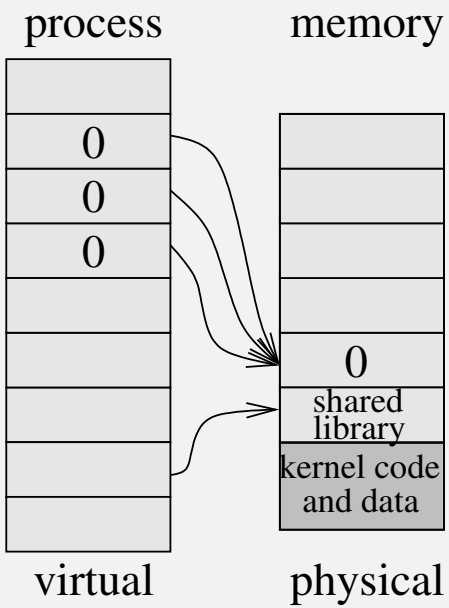
If the process writes to that block, the OS does a copy-on-write shuffle behind the scenes, allocating and clearing a new writable page

Thus only allocating and clearing pages that are actually used

Memory

Virtual Memory

Other Tricks

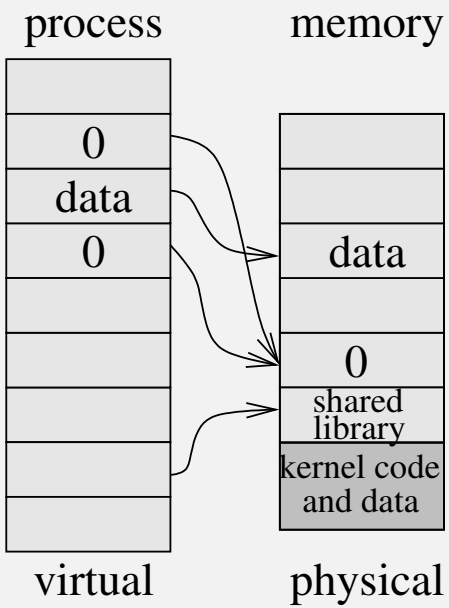


Zero Page

Memory

Virtual Memory

Other Tricks



Zero Page after a write

Memory

Virtual Memory

Other Tricks

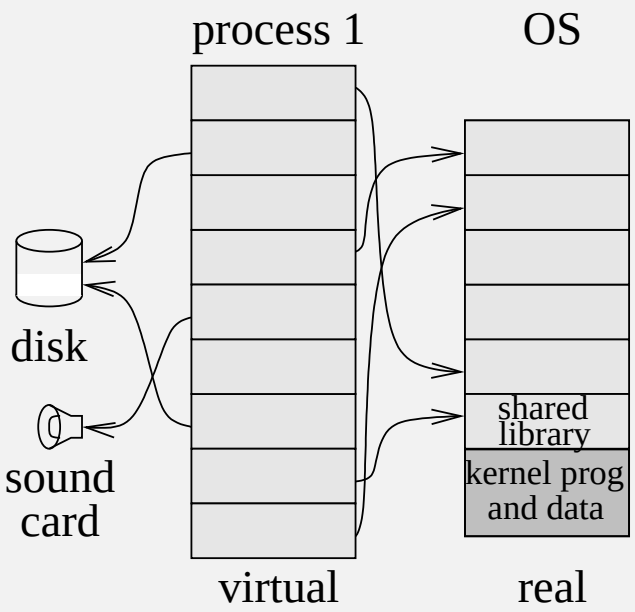
Virtual memory has other useful features like *memory mapping* of devices

Parts of the virtual address space can be mapped on to things other than memory, e.g., files, the screen, sound card

The OS can *mmap* (all or parts of) a file into memory: this means that reads and writes to “memory” are converted by the OS to reads and writes to that file (or screen, etc.)

Memory

Virtual Memory



Memory map

Memory

Virtual Memory

The hugely simplifies the problem for the programmer: rather than having to work out the fiddly details for a given piece of hardware, they can simply write to what looks like an area of memory and the OS sorts out all the details

Memory

Virtual Memory

Conclusion: TLBs solve many problems!

□

Filesystems

We now turn to files and filesystems

Current technology has main memory being limited in size (a few gigabytes) and *volatile*: the values disappear when you remove the power

To be able to manipulate more data and to make it *persistent* we turn to larger, but slower, devices like disks

And to organise everything we need *filesystems*

Filesystems

Note: not all applications want to use filesystems, in particular enterprise databases like to have direct access to disks themselves in order to optimise access for their very specific needs

Some people have experimented with making ordinary applications use DB-like access, mostly to a resounding failure

In general, a filesystem is what people want: a simple, efficient way of accessing their data

Filesystems

Another note: a filesystem is just an organisation of data, and doesn't need to be associated with *disks*

Filesystems can be found whenever we have large amounts of data that needs organising

USB keys, iPods, phones, ...

It's even occasionally useful to have a filesystem *in memory*, again as an organisational mechanism

Filesystems

Yet another note: and it's not necessary that the object or objects behind the filesystem store data

We can have it so that reading from one particular file actually returns keystrokes from the keyboard

Or writing to another file is actually sending sound to a soundcard

In fact, a Unix philosophy is "all devices are files"

This makes accessing devices incredibly easy for the programmer: just read and write

Exercise. Compare with using virtual memory to do the same

Filesystems

But, for now, we shall think of files in the traditional sense

A *file* is simply a named chunk of data stored somehow on a disk

Humans like easy names like `prog.c`, so there needs to be a mechanism to convert names to the place on disk where the data is stored

And when we have thousands or millions of files, meaning thousands or millions of names, we need some way of organising the names (even before we have thought of organising the data itself!)

Filesystems

Names

Notice the distinction between the *name* and the *data*

This is *very* important and the distinction runs throughout computer science

The same name can refer to different data (otherwise the whole thing would be useless, we could never fix bugs in `prog.c`)

Different names can refer to the same data. We tend to forget that, in real life, we can use different names to refer to the same thing: “Lewis Carroll” and “Charles Dodgson”

All but the simplest filesystems allow the same data to have multiple filenames

Filesystems

Names

For the philosophers:

It is possible to have a thing without a name (so how can we refer to it?)

It is possible to have a name without a thing it refers to

It is possible for names to have names

Exercise: read the introduction to the poem “Haddocks’ Eyes”, in “Through the Looking-Glass” by Lewis Carroll and explain the relevance

And explain the use of quotes ’“” in the above

Filesystems

Names

So names need to be organised; this is usually (but not always) done as a simple *hierarchy*

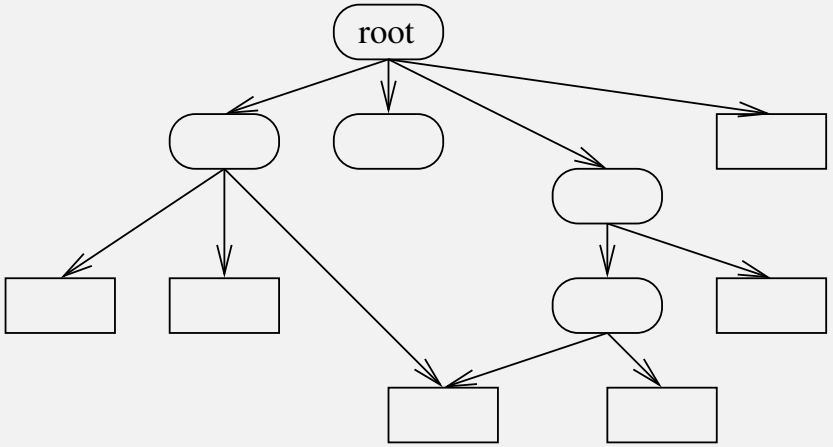
Rather than simply presenting all filenames to the user (a *flat* filesystem), we gather together related files and put them into a *directory*. Also called a *folder*

A directory is just a collection of (names of) files, but it allows us to simplify our thought processes

And (names of) directories can be collected in other directories and so on until we get to the top of the hierarchy, the *root*

Filesystems

Names



Files can appear at all levels But always within a directory In some systems, a file can be in more than one directory Generally, directories can only be within exactly *one* directory, for implementation reasons Directories can be empty

Filesystems

Names

The namespace hierarchy makes referring to a file easy

A Unix example: `/usr/bin/ls` refers to a file named `ls` inside a directory named `bin` inside a directory named `usr` which is in the root directory

The `/`s separate the names

The root directory is referred to as `/`, though its actual name is empty

Other OSs have similar ideas, but use different separators

Files can have multiple names: we might find that `/usr/local/bin/dir` refers to the same file as `/usr/bin/ls`

Filesystems

Names

The directory hierarchy forms a *directed acyclic graphs* (DAG)

This means: no loops

No loops means we can simply traverse the whole hierarchy and never get stuck in a loop and no unconnected loops if we delete a directory

We might find the same file twice, though

This is a tradeoff of flexibility vs. ease of system implementation

Filesystems

Names

To make referring to files even easier, each process has a *current working directory* (cwd)

This is just a prefix, stored in the PCB for each process, so that whenever the process asks for a file by an incomplete filename (not starting with a `/`), the kernel glues the cwd prefix on to the given name and uses that full name instead

So, with a cwd of `/u/cs/1/cs1abc` a process that asks for file `prog.c` gets file `/u/cs/1/cs1abc/prog.c`

With a cwd of `/u/cs/1/cs1def` a process that asks for file `prog.c` gets file `/u/cs/1/cs1def/prog.c`

Filesystems

Names

This is how different processes can refer to the same name `prog.c` but get different files

The cwd is a convenience for the programmer and may be changed as often as you like (`cd`, `chdir`)

Filesystems

Requirements

There are a lot of things we want from files

- create a new file
- delete a file
- open a file to access it
- read data from a file
- write data to a file
- close a file when we are done
- rename a file(rename a file)

That last one is actually a directory operation as we shall see in a moment

Filesystems

Requirements

And directories

- create a new directory
- delete a directory
- scan a directory for a filename or directory name
- add a file to a directory
- remove a file from a directory
- rename a file

The last three are intertwined

Filesystems

Requirements

This all is before we come to things like

- speed of access
- speed of update
- scalability to large numbers of files
- efficient use of disk space
- reliability
- protection/security
- simple backup and recovery

Filesystems

We shall be looking at the classical Unix filesystem as an example

Other filesystems are similar in their principles, though modern filesystems are immensely tweaked and tuned

They vary in their choice of datastructures and algorithms to implement the hierarchy for efficiency or other reasons

Filesystems

Records

Modern files tend to be essentially long arrays of bytes with no further structure

Early files had structure, namely *records*

This was a hangover from early systems using things like punched cards

A record is a fixed-size block of data, say 80 bytes

Records could only be read or written as a whole: this meant implementation on the hardware of the time was easy

Filesystems

Records

It also aligned with the way data was regarded at the time: records of peoples names, job classification, salary and so on (*fields*)

They would expect an entire record to be read or written at once

Modern filesystems are *byte oriented* and you can access them however you please

Filesystems

Inodes

The design of the traditional Unix filesystem is based on the *inode*

Each file has its own inode

The inode is a fixed size structure (stored on disk) that contains all the information about a file, its *metadata*

Filesystems

Inodes

Information in the inode includes

- Timestamps. Dates and times this file was last accessed and last modified
- Ownership. The userid of the owner of this file, for protection purposes
- Size. How big the file is currently
- Type. Whether this is a *plain file*, or a directory, or some other kind of *special file*
- Access permissions. Who can read or write or run this file (if it is a program)
- Reference count. The number of names this file has
- Pointers to areas on the disk where the actual data lives

Filesystems

Inodes

Notice that filenames are *not* in the inode

Filenames are stored in directories

A directory is essentially just a list of names of files and subdirectories, together with their inode numbers

Name	Inode
`foo.c`	23
`ff.html`	42
`mydata`	7

Originally just a table, these days clever datastructures are used to manage the large numbers of names we use

Filesystems

Inodes

This is how a file can have many names: multiple directory entries referring to the same inode

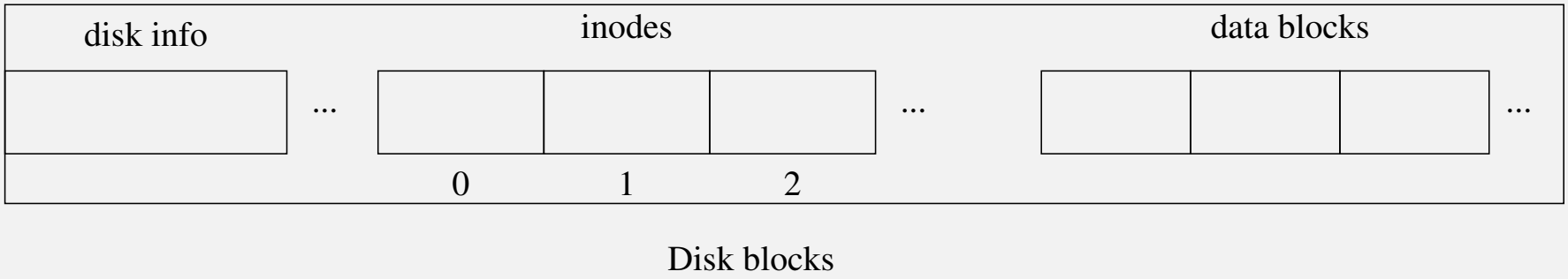
As a consequence a file cannot know its own name(s) as the names are independent of the file

In some sense, the inode number is the true name of the file

Filesystems

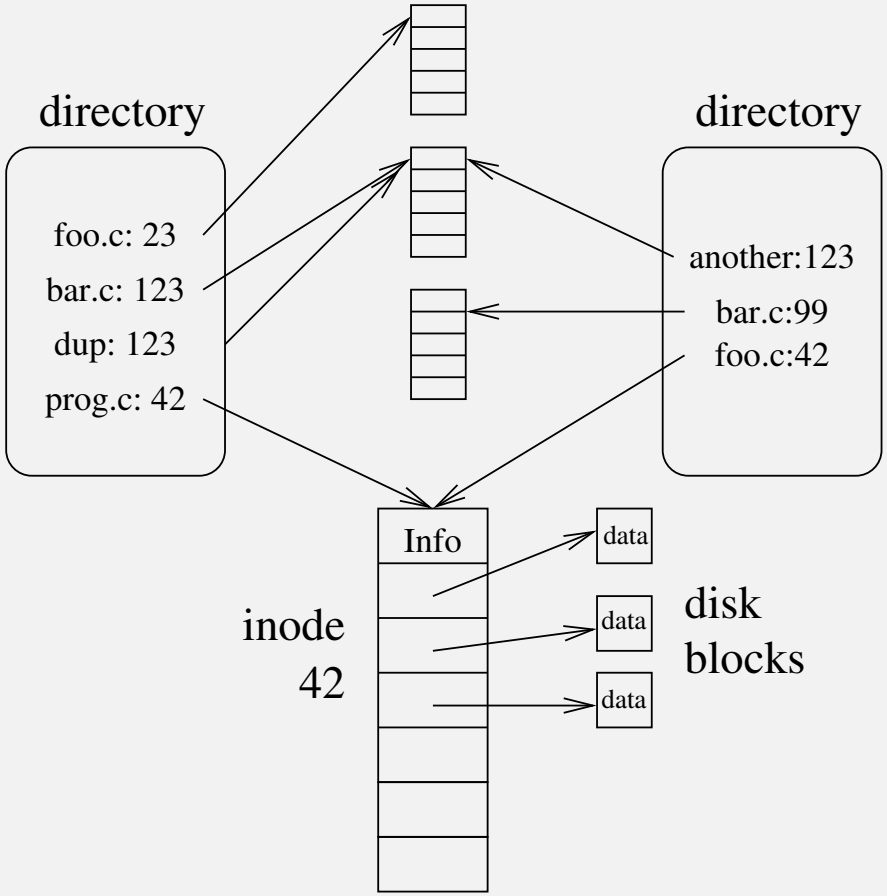
Inodes

As inodes are a fixed size, it is easy to put them in a simple array on disk and just refer to them by their index in the array: the *inode number*



Filesystems

Inodes



Filesystems

Inodes

There are a couple of special names always to be found in every directory

The name `..` refers back to the parent directory. This allows us to crawl up the hierarchy until we reach the root

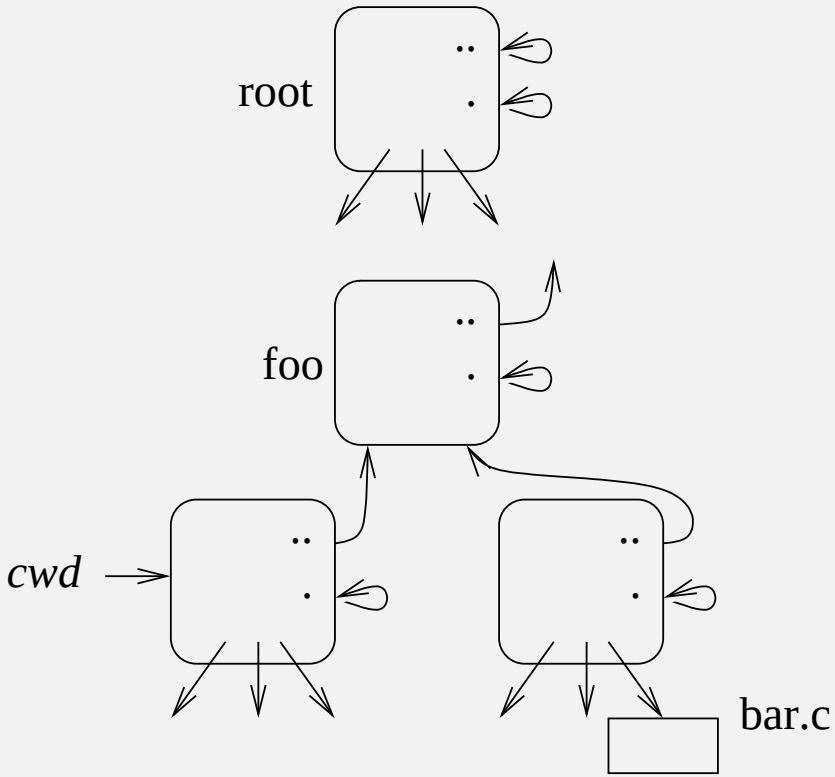
So `../foo/bar.c` is a name of a file in a sibling directory: up; across; then down in the hierarchy

The `..` of `/` is `/`

The name `.` refers a directory back to itself. This often turns out to be useful to do

Filesystems

Inodes



Filesystems

Inodes

The inode contains a reference count: the number of names the file/inode has

If the count drops to zero, the OS can remove the file

Deleting a file is a matter of

- Removing the name reference in the relevant directory
- Decrementing the reference count in the inode
- If the count reaches 0, we can free the inode and the disk blocks it refers to

Filesystems

Inodes

In fact, each time a program opens a file the OS increments the count; and decrements it when the program closes the file (possibly when the program exits)

So it is possible for a program to create a new file (inc); open it (inc); delete it (dec); and still be able to read and write to it

The file will only disappear when the program ends (dec)

No other process can see this file: there is no name in any directory

Filesystems

Inodes

The data on the disk is in *disk blocks*, fixed size areas on the disk, e.g., 512 or 1024 bytes

Having a fixed size allows for easy and fast allocation and deallocation

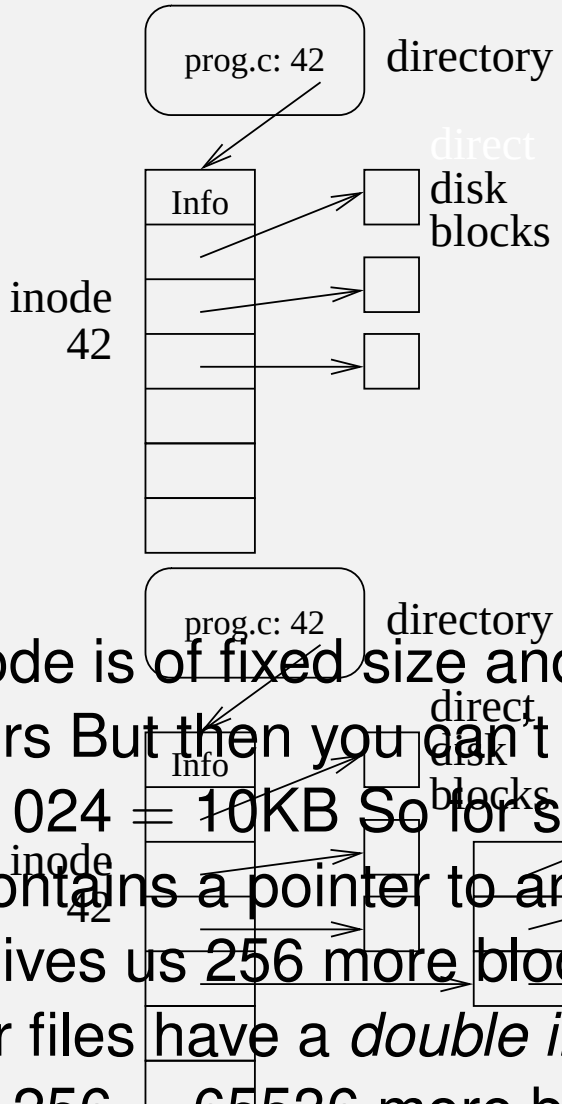
This is similar to pages in memory; but now physical location of blocks *is* important as discs are mechanical devices

Whole numbers of blocks are always allocated to files

This can lead to wastage, e.g., a 1025 byte file might need two blocks, but uses just over half of the space. Though there are lot of tricks in real filesystems to avoid the worst of this

Filesystems

Inodes



An inode is of fixed size and has space for, say, 10 block pointers But then you can't have files bigger than $10 \times 1024 = 10$KB So for such files we have an *indirect block*, that contains a pointer to an array of 256, say, block pointers This gives us 256 more blocks, which is 256KB more space Bigger files have a *double indirect block* This gives us

Filesystems

Inodes

Now every indirect block is overhead occupying space on the disk that could otherwise be storing data

But this is not so wasteful as you might think as most files are quite small; the overhead for large files is relatively small, too

Filesystems

Inodes

Caching the inode and the indirect blocks in memory helps reduce the lookup overhead

The space for the pointers is used for various other things when the inode refers to something other than a disk file

Filesystems

Inodes

For example, a *soft link* (similar to a Windows *shortcut*) to a file or directory

This is a special inode whose purpose is to say “don’t look at me, look at this file instead”

If you had a soft link named `foo` that linked to `bar` its content would be just the name “`bar`”

But the action of the OS when a program opens `foo` is not to present the data “bar”, but to close inode `foo` and open inode named by `bar` instead

In effect, this is another way for files to have multiple names, but it is very different from normal multiple names, called *hard links*

Filesystems

Inodes

A hard link is the normal reference to (the inode of) a file; a soft link is (a reference to an inode containing) a signpost saying “look over there”

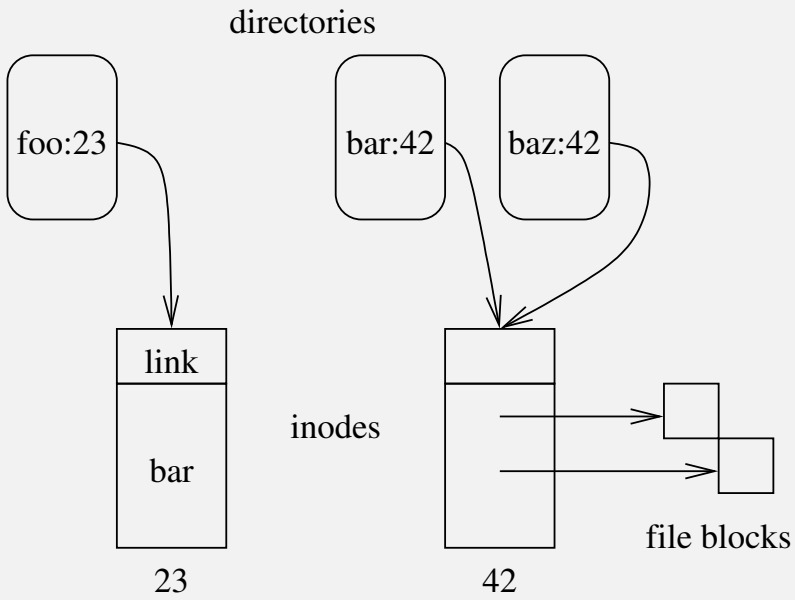
The soft link might point to a place where there is no file; a hard link *is* the file

And, as there are no inode references involved in a soft link, it can be the name of any file on any filesystem in the machine

Note: a hard link refers to the file, while a soft link refers to a *name* of the file. So a hard link is a name, while a soft link is a name of a name

Filesystems

Inodes



Filesystems

Inodes

Use `ls -li` to see the link details and inode number of a file under Unix

```
% ln -s somefile link1
% ls -li link1
3154340 lrwxrwxrwx 2 rjb users 6 2010-04-22 10:38 lin
k1 -> somefile
% ln link1 link2
% ls -li link*
3154340 lrwxrwxrwx 2 rjb users 6 2010-04-22 10:38 lin
k1 -> somefile
3154340 lrwxrwxrwx 2 rjb users 6 2010-04-22 10:38 lin
k2 -> somefile
```

Filesystems

Inodes

When a program opens a file, the OS must find where on disk the file lives

Say we are looking for the file `prog.c` with a cwd of `/home/rjb`

- The name is incomplete, so the OS prepends the cwd giving `/home/rjb/prog.c`
- The OS reads the block containing the root directory off disk and scans through it for the name `home`
- It finds it and gets the inode number for `home`
- It reads the inode off disk and finds it refers to a directory
- It reads the block containing the directory off disk
- It scans the directory for the name `rjb`

Filesystems

Inodes

- It finds it and gets the inode number for `rjb`
- It reads the inode off disk and finds it refers to a directory
- It reads the block containing the directory off disk
- It scans the directory for the name `prog.c`
- It finds it and gets the inode number for `prog.c`
- It reads the inode off disk and finds it refers to a file
- It reads the blocks containing the file off disk

This must be done for every file opened

Again, caching can be used to great effect: keeping copies of the inodes and directories in memory, rather than re-reading them every time

Filesystems

Inodes

If we want more than one filesystem on a disk, or more than one kind of filesystem, we can split the disk into separate *partitions*

A partition is just a chunk of disk owned by a single filesystem

So we can have multiple filesystems on a single disk, e.g., two Unix filesystems and a Windows filesystem

Each filesystem has its own inode tables (or whatever it requires) and are logically quite separate

Filesystems

Inodes

Note that inode 23 on one partition is different to inode 23 on another partition, meaning we can't have hard links across filesystems

We *can* have soft links across filesystems, as soft links are by names, not inode numbers: this is really why soft links were invented

Filesystems

Mounting

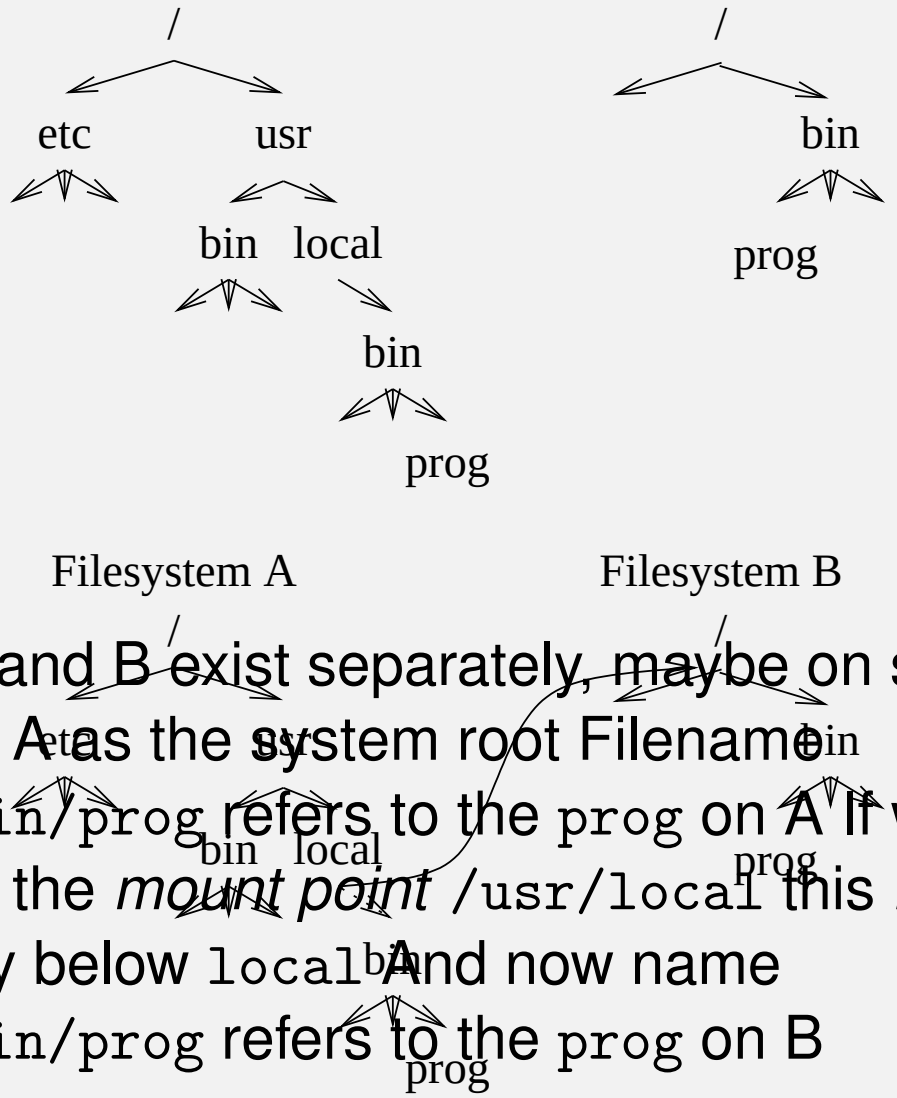
Under Unix, a filesystem can be *mounted* on another filesystem

The name comes from when disks needed to be physically mounted on the drives by system operators

A *mount point* is a special inode that says: “now go and look at this filesystem”

Filesystems

Mounting



Filesystems A and B exist separately, maybe on separate disks, with filesystem A as the system root Filename `/usr/local/bin/prog` refers to the `prog` on A If we mount filesystem B at the *mount point* `/usr/local` this *hides* the part of the hierarchy below `local` And now name `/usr/local/bin/prog` refers to the `prog` on B

Filesystems

Mounting

When the file lookup reads the inode for the mount point at `/usr/local` it switches filesystem and continues looking from the root of B

This means that we can have many partitions presented as a single unified name space

And partition B could be a separate disk; or on a USB key; or on a read-only medium like a CD

Filesystems

Mounting

Note that B will have its own inode table, so there can't be a hard link of, say, a name in `/usr/bin` to a name in `/usr/local/bin`

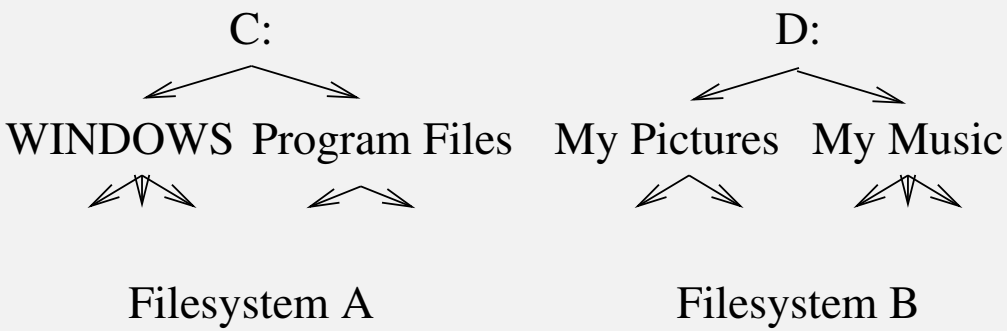
In fact, B might even have a completely different kind of filesystem, perhaps not based on inodes

Or can be on a separate machine if this was a mount of a *network* disk

Filesystems

Mounting

This is completely different from Windows where each partition is separate and has a prefix like `C:`



Filesystems

Mounting

Going the other way, mechanisms exist for gluing several disks together to make them appear as a single partition: this can be for making huge filesystems out of small disks, or for reliability through redundancy (RAID)

Filesystems

Other filesystems you might like to look at

- btrfs
- ext4
- FAT, VFAT
- FUSE
- GFS (Global File System)
- Google File System
- HFS+
- ISO 9660
- JFFS2
- Lustre
- NFS
- NTFS
- OCFS2
- procfs
- Reiser
- ReFS (Resilient File System)
- UnionFS
- ZFS

Also see “List of file systems” on Wikipedia

Filesystems

Exercise. Solid state disks (SSDs) are common these days. What differences do they bring to the way filesystems should be implemented?

Exercise. Read about the various kinds of RAID filesystems and the benefits they bring

Filesystems

Dilbert
by Scott Adams

Filesystems

File Permissions

We now look at file protection

It is not good if every user of a computer can read or update any file on the system

Quite apart from issues of privacy, there are many files that are essential to the good running of the OS itself (configuration files, system library files, etc.)

So we need to limit access somehow

Filesystems

File Permissions

There are generally three ways to access a file

- to read from it
- to write to it
- to run it as a program

Other ways include *append only*, where you can add to the end of a file but otherwise can't update it: useful for logging (you can't accidentally — or by choice – alter the earlier part of the log)

Filesystems

File Permissions

Similarly, for a directory

- list the directory's contents (read)
- create/delete/rename a file (write)
- search a directory

Filesystems

File Permissions

Files have owners, generally they inherit the userid of the process that created it

There are also *groups*, collections of users

For example, “First year computing” and “Computing Staff”

Groups allow us to allow or deny access to large numbers of people

Each user can be in several groups

Filesystems

File Permissions

Unfortunately groups are quite inflexible and are set by the systems operators: normal users cannot create groups

We have to rely on the operators creating the right groups

Each file has a single group ownership to go alongside the user ownership

Filesystems

File Permissions

Unix permissions mostly work OK but are very inflexible regarding groups

You can't specify that just users Alice and Bob should have read/write access but others read only, unless the operators have created a group containing just Alice and Bob

A more general mechanism is *access control lists* (ACLs) that allows a low more flexible way of allowing and disallowing access

Exercise. Read about ACLs

Filesystems

File Permissions

Every file (in Unix) has a collection of permission bits

- for the owner
- for the group
- for everybody else not included in the above

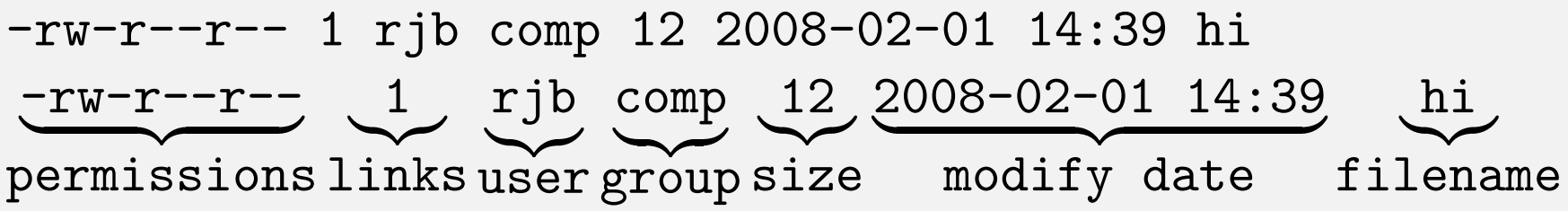
And for each of

- read access
- write access
- execute access

And a couple of others

Filesystems

File Permissions



Filesystems

File Permissions

```
-rw-r--r-- 1 rjb comp 12 2008-02-01 14:39 hi
```

Three groups of three flags, with one extra at the start

If the userid of the process matches the userid of the file, use the first set

Else if the groupid of the process matches the groupid of the file, use the second set

Else use the third set

Filesystems

File Permissions

```
-rw-r--r-- 1 rjb comp 12 2008-02-01 14:39 hi
```

- The `r` flag indicates permission to read
- The `w` flag indicates permission to write
- The `x` flag indicates permission to execute

A process running as user `rjb` can read and write this file

A process running as group `comp` can only read this file

Other processes can only read this file

Filesystems

File Permissions

```
-rwxr-x--- 1 rjb comp 24 2008-02-01 15:11 prog
```

- User `rjb` can read, write and execute this
- Users in the group `comp` can read and execute, but not write
- Others can do nothing at all

Filesystems

File Permissions

For a directory

```
drwxr-xr-x 2 rjb comp 4096 2008-02-01 15:11 .
```

The first flag is set if this is a directory

- `r` read: you can list the contents of this directory
- `w` write: you can create, delete and rename files in this directory
- `x`: you can search this directory; also can set as cwd

Note that if the directory permission is `w` you can delete a file even if you cannot read it; even if you don't own it

Filesystems

File Permissions

Permissions are set by the Unix `chmod` command

Other flags include

- `t` sticky, “restricted deletion flag”: when set on a directory only the owner of a file may delete it
- `s` setuid on a program: execute this program with the userid of the *file* not the user

Filesystems

File Permissions

Other OSs have less support for access permissions

- DOS and Windows pre-XP just have a “read-only” permission which can be set and cleared by any user
- DOS and Windows pre-XP don’t really have users
- MacOS pre-X are similar

Filesystems

To wrap up filesystems here are a few remarks

We don’t have time to go into how disks work (SSD or spinning), for example, how data blocks are managed on the medium

But be aware this is also a lot of complicated detail!

Filesystems

Next:

- As disks are relatively slow, there are caching tricks used to help speed things up
- *Disk caching* is using “spare” memory to keep a cache copy of some of the disk contents
- The speed benefits are huge as long as you balance the use of memory for cache against the use of memory for everything else
- With a good filesystem a program will use the disk just once or twice and spend most of its time using the disk cache (barely a “file” system at all!)
- And then there is memory mapped disks as previously mentioned

Filesystems

Reliability

- Filesystems *must* be bug free. We can get away with the occasional bug elsewhere, but if the user loses their data this is a big problem
- Sometimes we have to put up with a little data loss (e.g., power loss during a write), but we must *never* have data corruption
- Thus filesystem programmers tend to be very conservative: only well-tested systems should be used
- If there is any corruption in the filesystem structure data can be lost
- We can't rely on users making backups!

Filesystems

Reliability

- Unfortunately, there can be external influences, such as power loss just as inode pointers are being updated
- This could leave the filesystem in an inconsistent state
- Modern filesystems try very hard never to let this happen
- So things like *transactional*, *log structured* and *journalling* filesystems have been created
- Similarly, disk technology is very good these days, but disks still have problems
- So there are hardware monitoring systems like SMART that watch a disk for impending problems

Filesystems

Reliability

- And filesystems that check data for errors as it is read
- And collections of disks in a *redundant array of inexpensive disks* (RAID) that spreads data across multiple disks so that if one fails the data is retrievable
- And so on

A lot of research has been put into filesystems: and it's still ongoing

Parkinson's Law: data expands to fill the space available

□

Conclusion of OS

Operating systems are still very much a current topic

We have only scratched the surface — there are many other things a real OS would have to implement

There are still lots of hard problems to solve, such as scheduling

And, as hardware changes, OSs must change, too

OSs for low-power devices (in particular mobile phones) are a huge source of research

Conclusion of OS

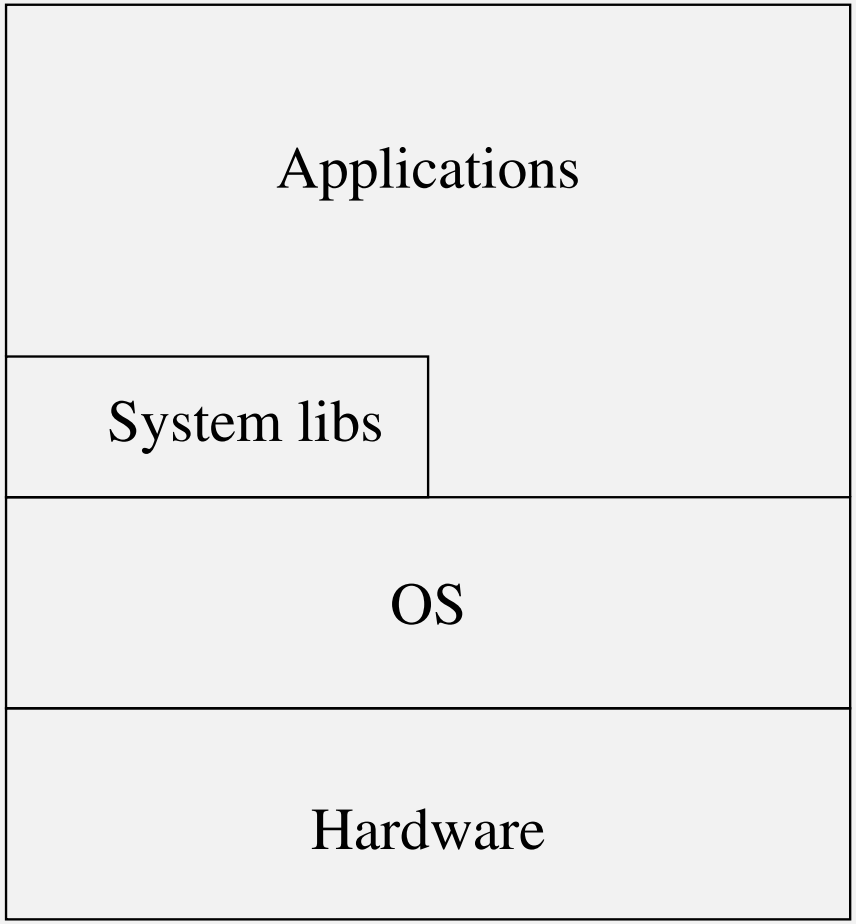
At the other end of the scale, people are still developing OSs on large machines

OS *virtualisation* is important in the era of cloud computing

Where several users (customers) are sharing the same hardware, but each has their own, private OS running their own, private applications

Originally, OSs were the software closest to the hardware: with OS virtualisation, this is no longer necessarily true

Conclusion of OS



Traditional OS

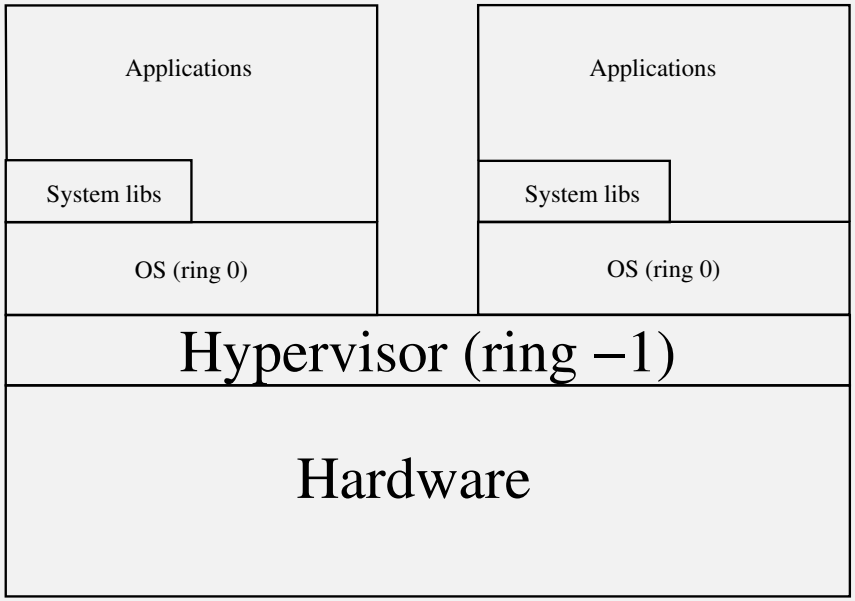
Conclusion of OS

Sometimes an application only runs on a specific OS

But repeatedly rebooting a machine with a different OS every time a user wants to run a different application is not a good approach

So the solution is to have multiple, simultaneous OSs on a single machine

Conclusion of OS



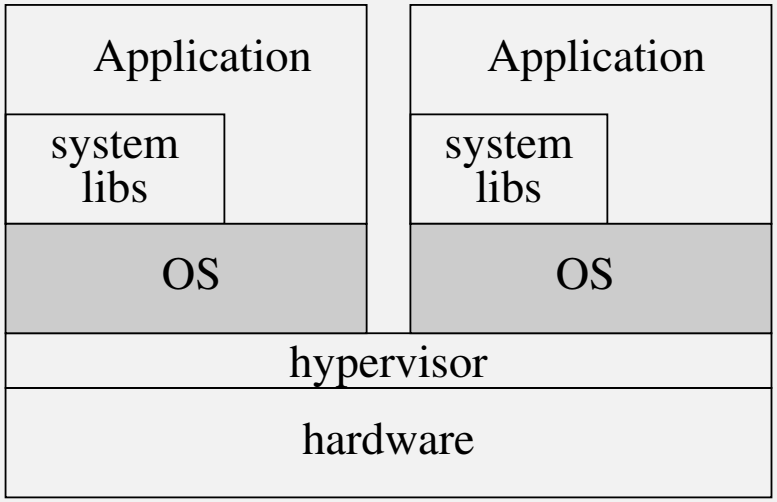
Virtualised OSs

Hypervisors appeared in IBM mainframes in the late 1960s

Conclusion of OS

There are several ways OS virtualisation is done

Conclusion of OS



Bare metal virtualisation has a thin layer, the *hypervisor*, to manage the hardware, allowing each OS to see separate "virtual hardware" which they manage

Conclusion of OS

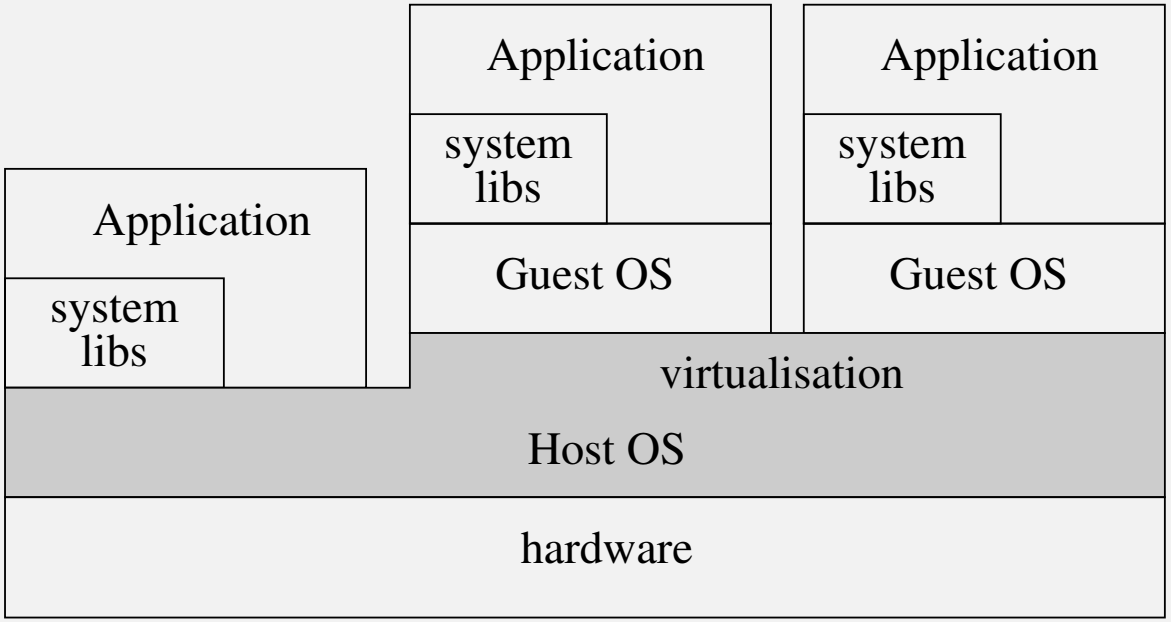
The OSs can be completely different, e.g., Windows and Linux, and each believe they have the whole machine

Modern X86 architectures provide a Ring -1 to support this

Examples: Xen, Hyper-V

Good for sharing the computer amongst users who have requirements for different OSs

Conclusion of OS

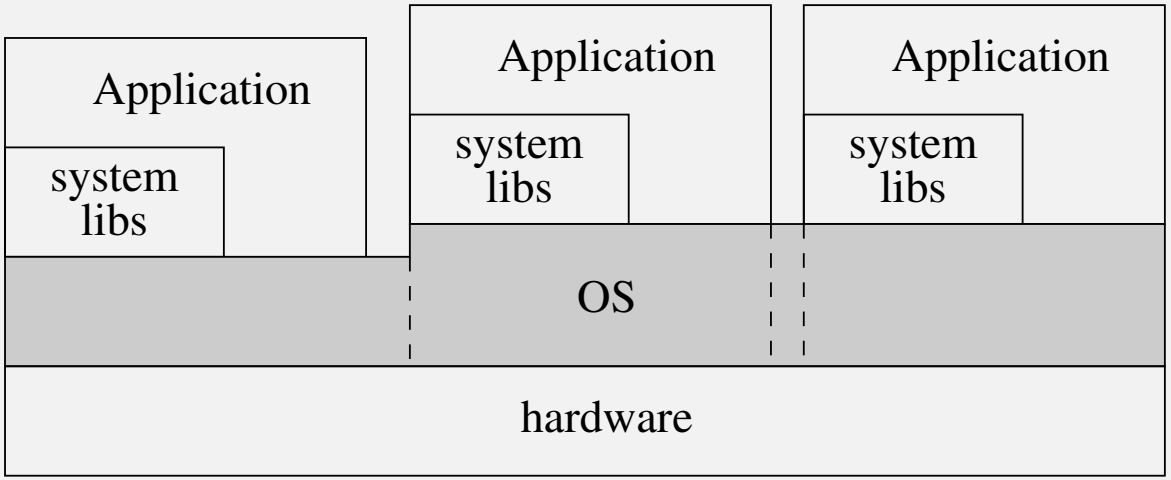


Hosted virtualisation has a normal *host* OS that runs virtualisation code. One or more *guest* OSs run on top of that

Examples: VMWare, VirtualBox, Parallels

Good for when you need sophisticated management of the guest OSs by the host OS, for example in Cloud provision

Conclusion of OS



Not quite OS virtualisation, but with the same target applications is *containers*. The applications share the same OS, but the OS is rigidly partitioned so each container cannot see or influence what is happening in other containers (e.g., CPU limits)

Conclusion of OS

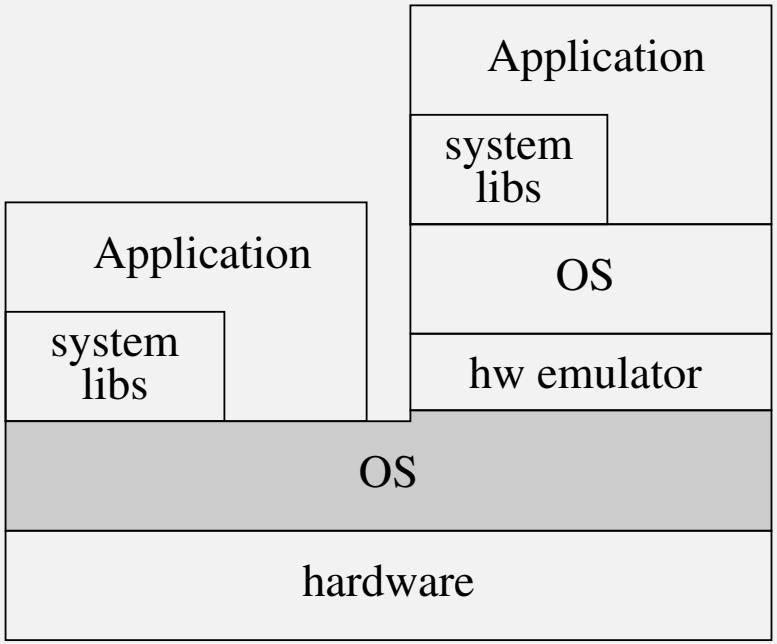
With containers, the applications must run on the same OS kernel, but can have different systems libraries and other software (e.g., RedHat in one and Ubuntu in another)

We might think of this as a kind of multiple user modes

Examples: Solaris containers, Docker

Good for application delivery, where an application needs a lot of specific system library support: so we deliver the systems libraries with the application!

Conclusion of OS



And then there are variants that do *hardware virtualisation* by emulating different kinds of hardware, e.g., we might have our OS running on an ARM emulation running on X86 hardware

Or on an X86 emulation on ARM hardware

Conclusion of OS

These emulations are a lot slower than the native hardware, but provide a flexibility to the customer

Examples: Qemu (emulates several kinds of hardware), Bochs (emulates X86)

Exercise. Compare with Apple's new Rosetta software that allows Intel code to run on Arm hardware (only user code, though)

Conclusion of OS

Exercise. Read up on Cloud Services, Software as a Service (SaaS), Infrastructure as a Service (IaaS), Platform as a Service (PaaS), Software Appliances

Conclusion of OS

All of these techniques are applied in cloud computing, where users buy time on a large, remote machine

Welcome to the 1960s!

Conclusion of OS

Exercise. On Mars, the autonomous helicopter drone Ingenuity (brought by the lander Perseverance) runs Linux on a 500Hz (not MHz!) processor. Read about this

Exercise. Play with an OS you are not familiar with (Mac, Win or Lin or other) and learn the ways it does things. Write, compile and run a program

Exercise. Read about the advances in persistent memory: comparable in speed to main memory, but retains data when power cycled like disk (*non-volatile*). What changes would we need from an OS to deal with such a technology?

□