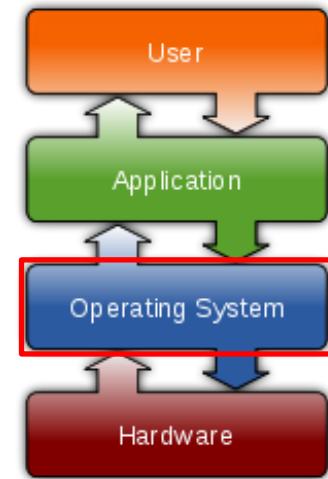


ENCE360

Operating Systems

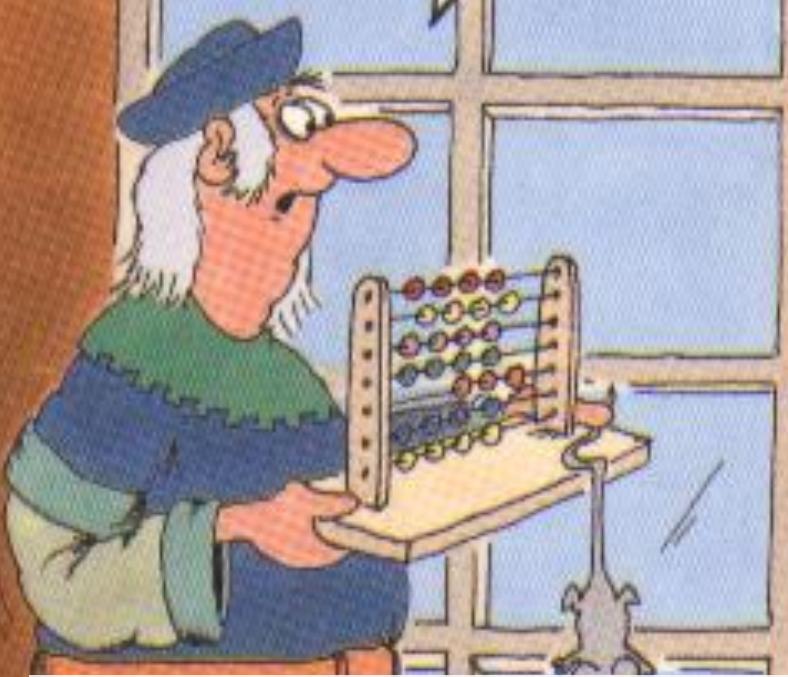


Introduction to Operating Systems

MODERN OPERATING SYSTEMS
(MOS)
By Andrew Tanenbaum
Chapter 1

OPERATING SYSTEMS: THREE EASY PIECES
R. Arpaci-Dusseau and A. Arpaci-Dusseau,
March, 2015
<http://www.osstep.org/>

The newest Computer Model, 16 Colours, on Harddisk, including Mouse.....



You should wait, Your Majesty - In 6 months it costs only half.....

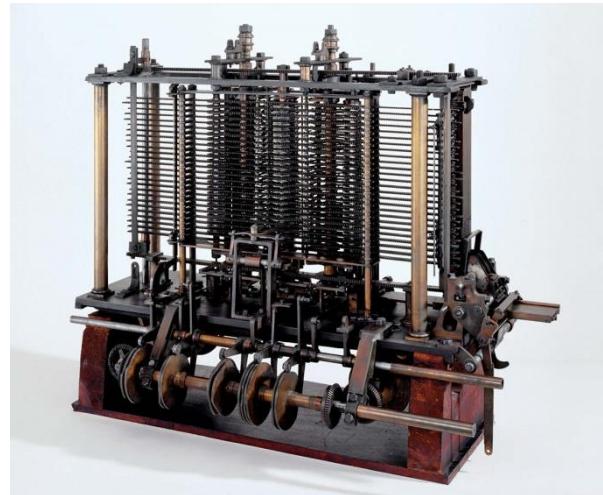


Why? A brief history of computers

0th Generation: doesn't work

Analytical engine

- Charles Babbage 1792-1871))
- Digital, programmable, *Turing complete*
- Punch card I/O
- Ada Lovelace world's first “programmer”
- Unable to be engineered
- Would have been very slow

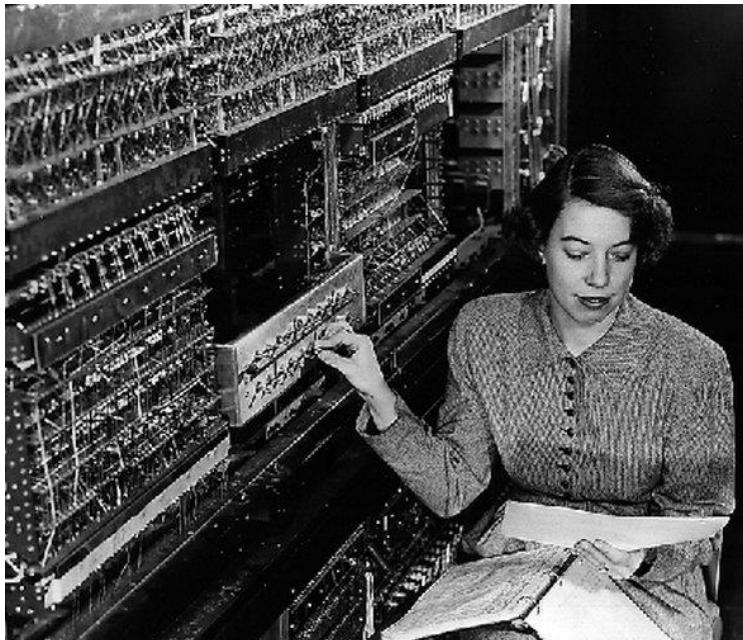


1st Generation: hard-wired

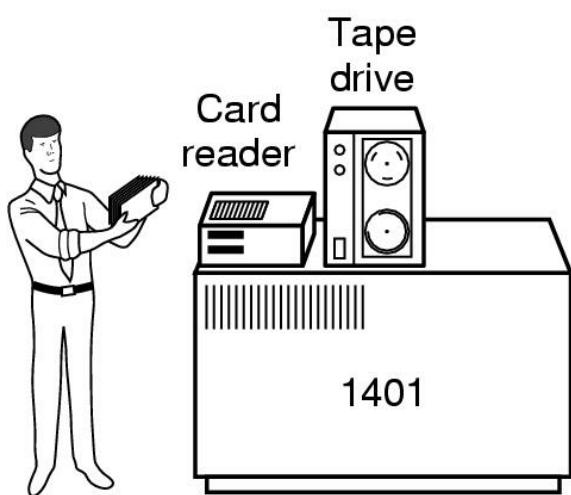
- Custom built,
“programmed” with wires
 - No operating system



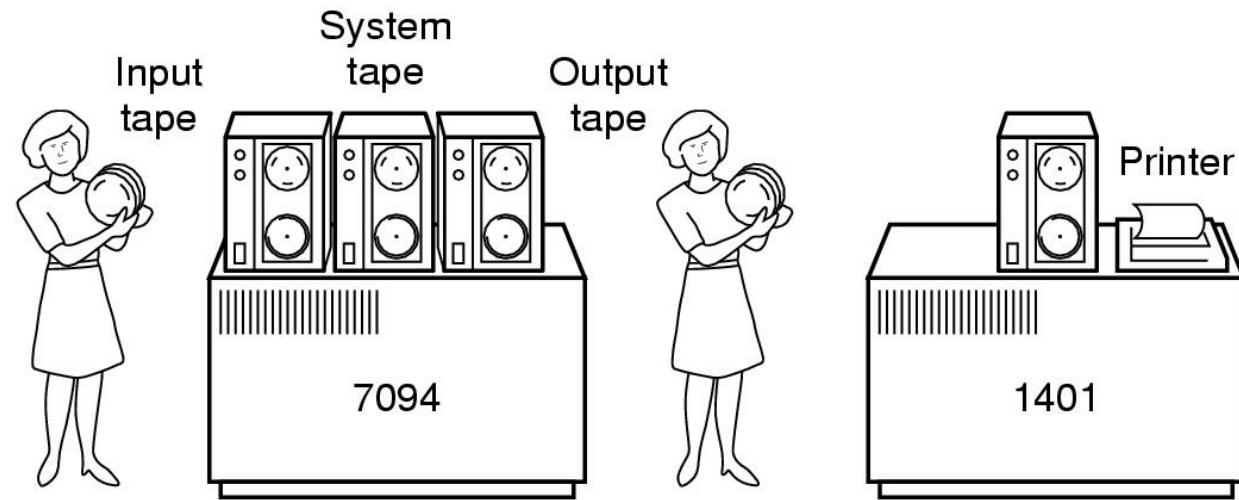
1907



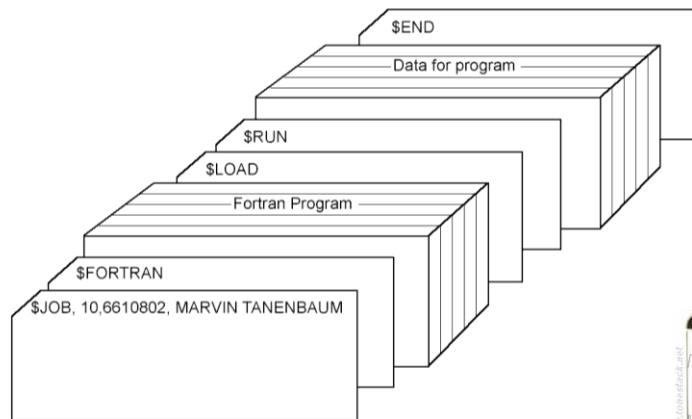
2nd Generation: programmable



(a)



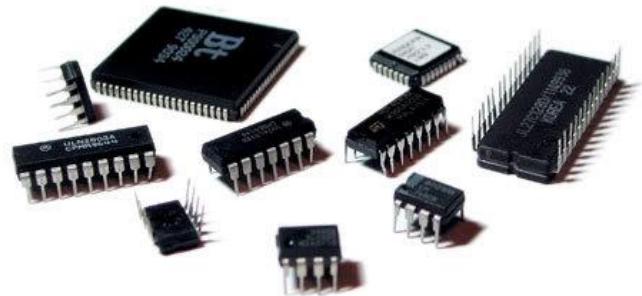
- Programmed batch systems
- One job at a time
- “Operating system” loaded, cleaned up jobs



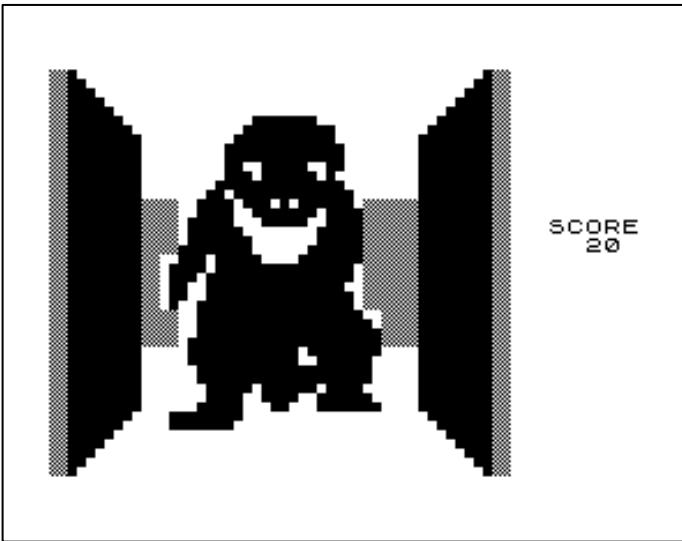
3rd Generation: multi-user



- Multiprogramming:
 - many jobs “running” at once
- First real operating systems
 - MULTICS/Unix/Linux, VMS, others



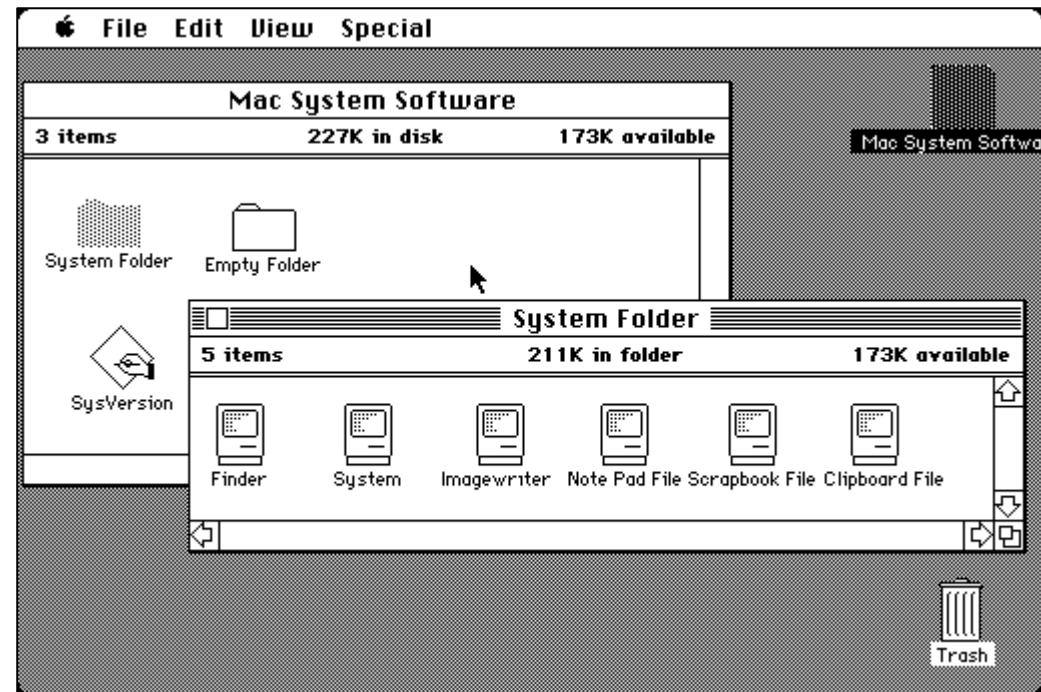
4th generation (1): personal



```
310 LET A=USR 18288
320 IF INKEY$="" THEN GOTO 320
330 POKE 17901, INT (RND*128)
335 IF PEEK 21623<>178 THEN POKE ((PEEK 16396+256*PEEK 16397)+6), 0
340 POKE 16522, CODE INKEY$
350 LET A=USR 18224
360 IF PEEK 16519>=128 THEN GOTO 5000
370 FOR N=0 TO 5
380 NEXT N
390 IF PEEK (PEEK 16514+17920)<>45 THEN GOTO 330
400 FOR N=0 TO 30
405 POKE 17901, INT (128*RND)
410 LET A=USR 18224
420 FOR M=0 TO 3
425 NEXT M
430 NEXT N
440 CLS
450 GOTO 2520
370 FOR N=0 TO 0
```

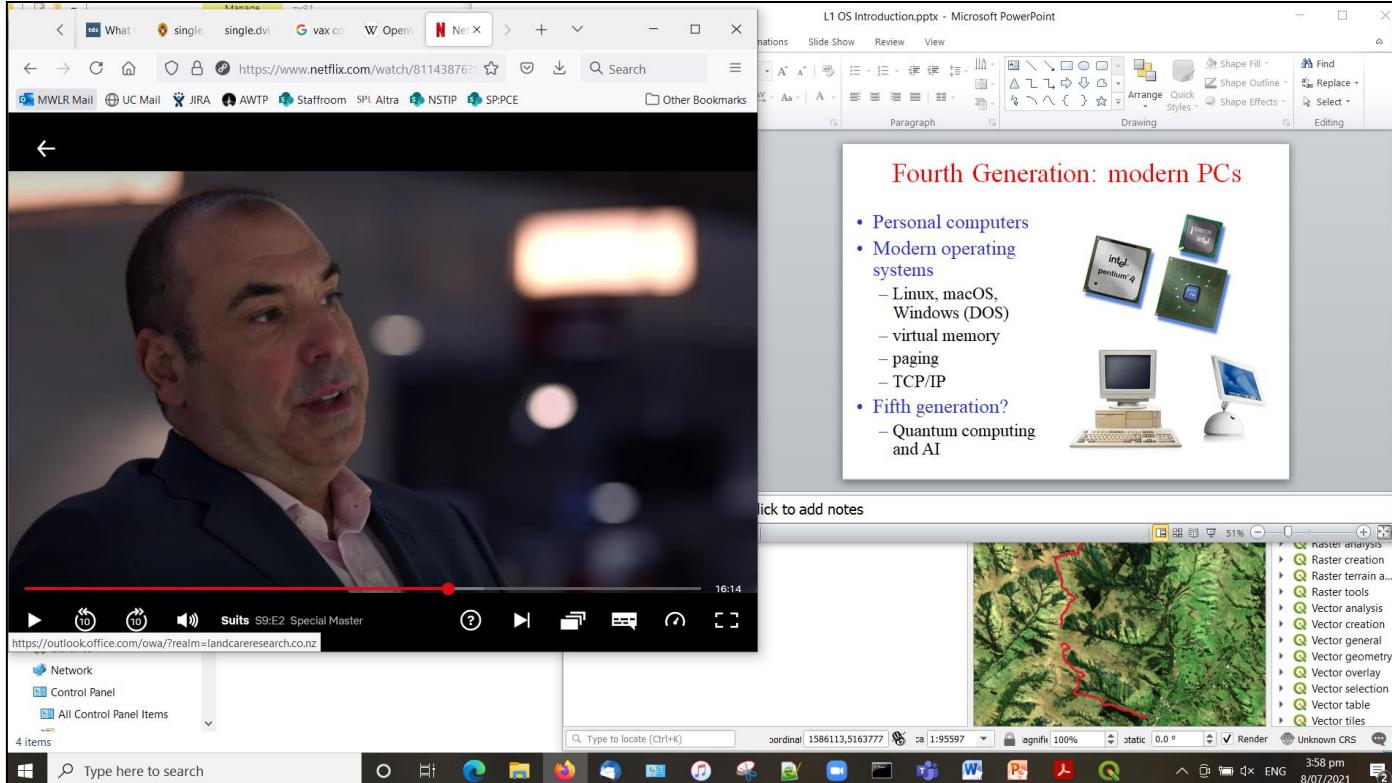
- Complexity hidden from the user (?)
 - BASIC interpreter
 - Machine code
- One program in memory

4th generation (2): GUI



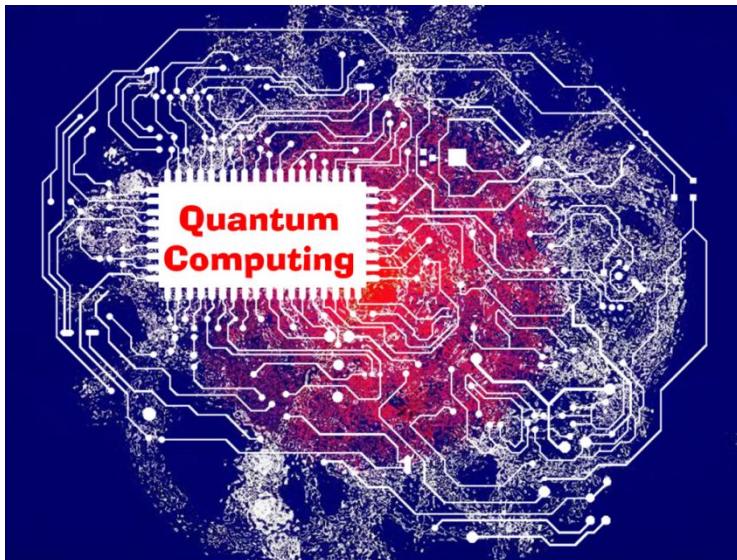
- Graphical user interface (and a mouse!)
- Multiple applications in memory at once
 - But only one runs at a time

4th Generation (2): modern PCs

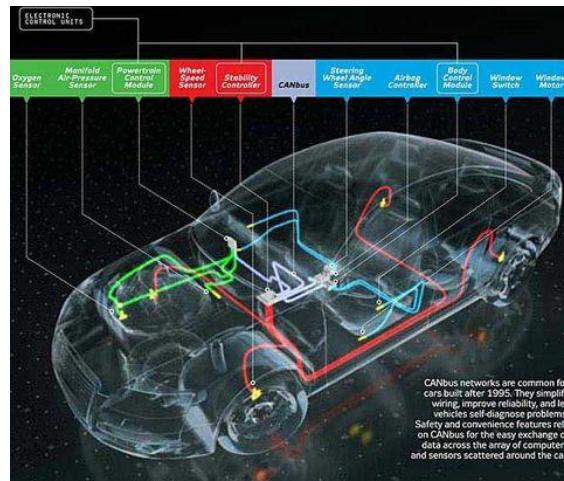


- Personal, multiple applications at once (really!)
- Modern OS (Linux, macOS, Windows)

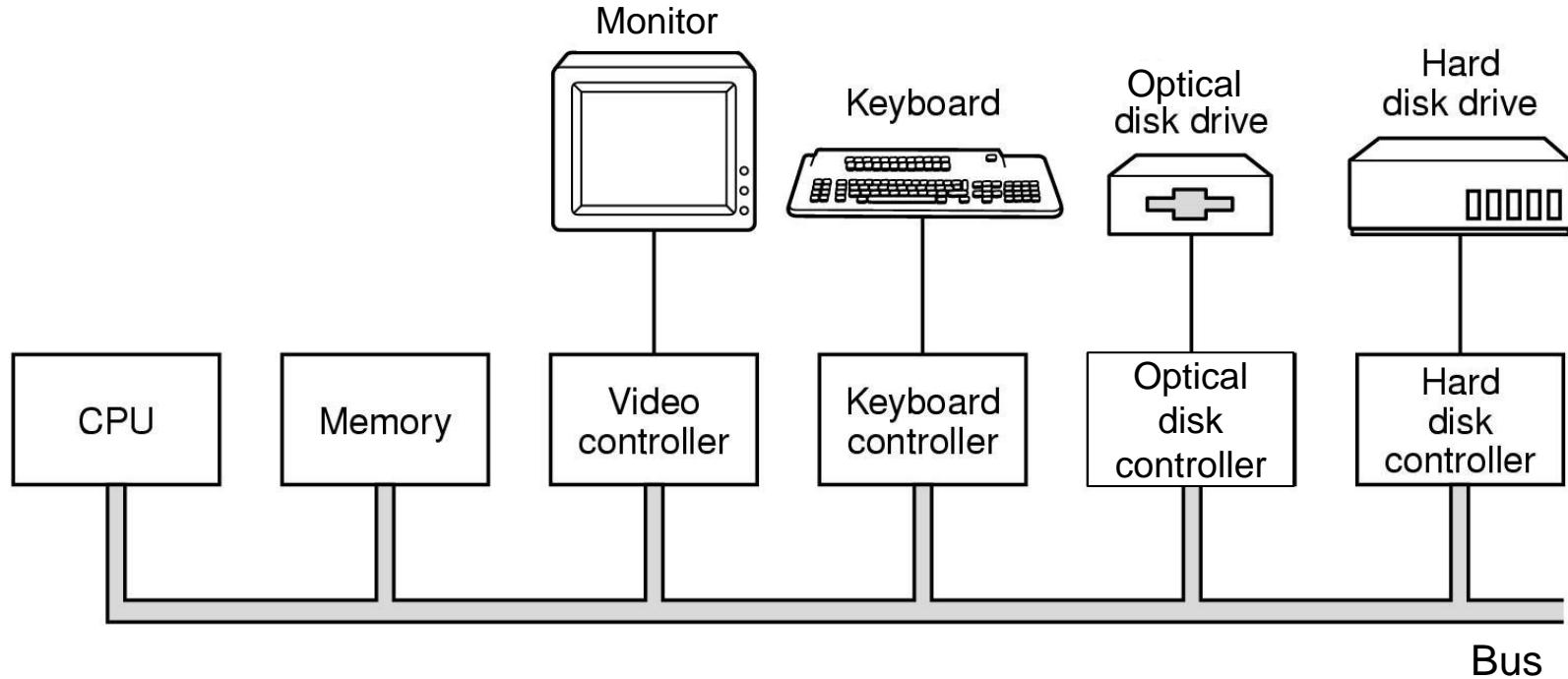
5th generation?



Operating systems everywhere...

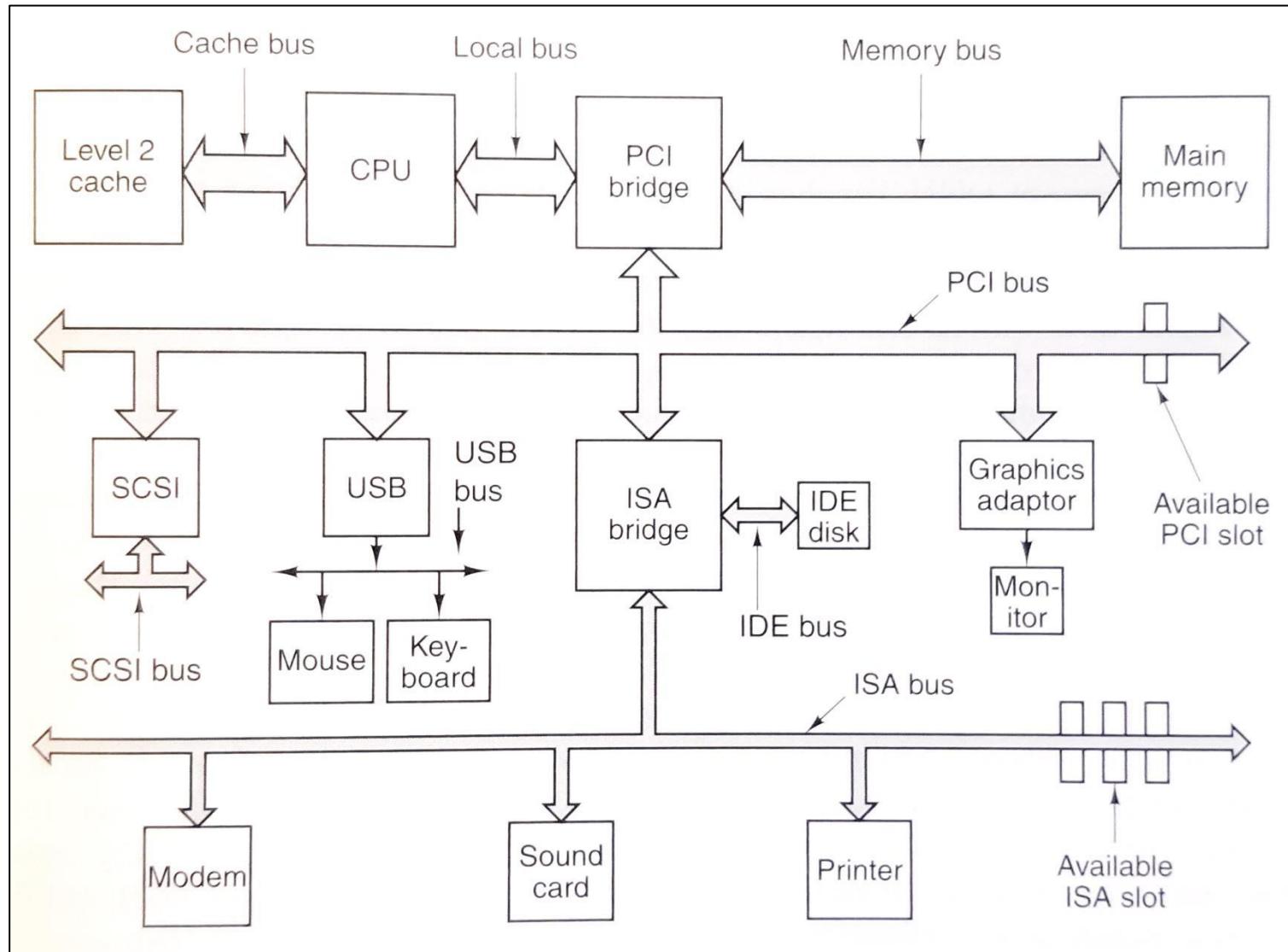


What's in a computer?

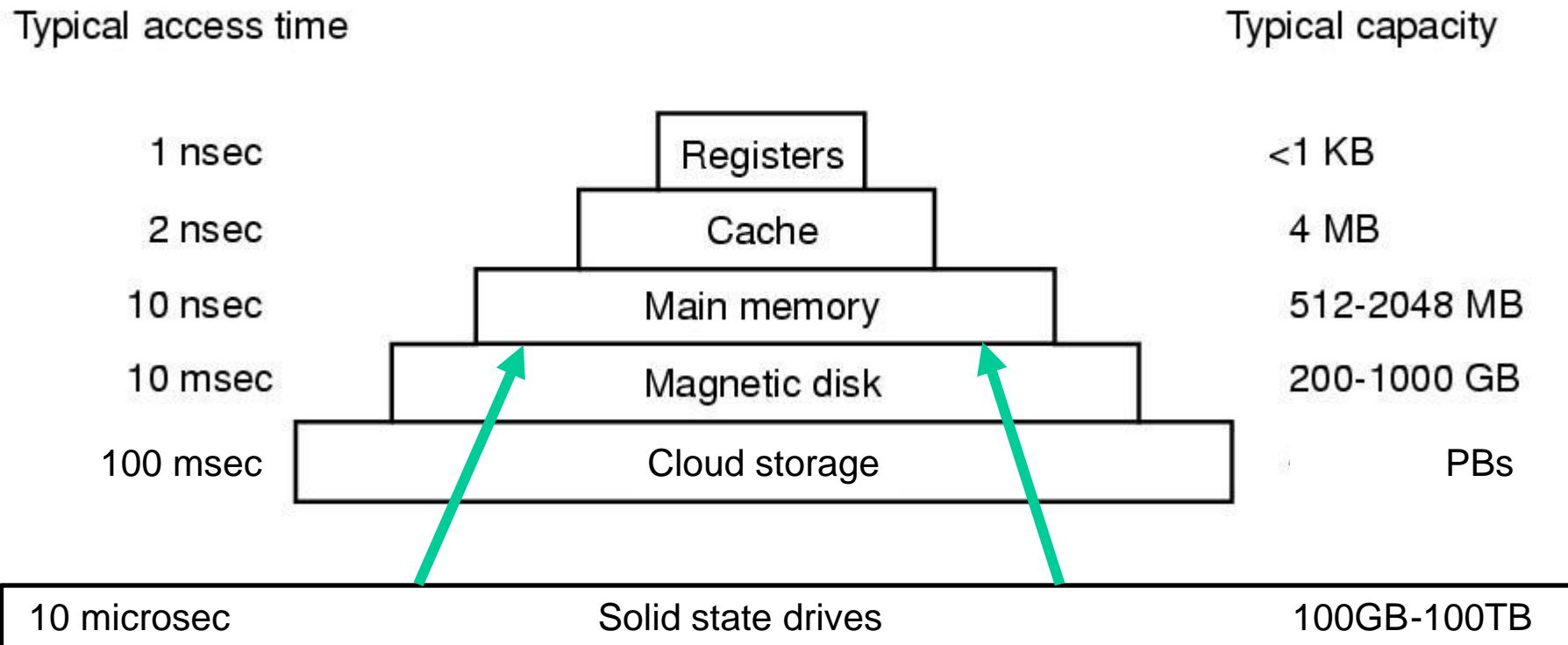


Who looks after all this stuff?

OK, what's *really* in a computer?



Storage Hierarchy

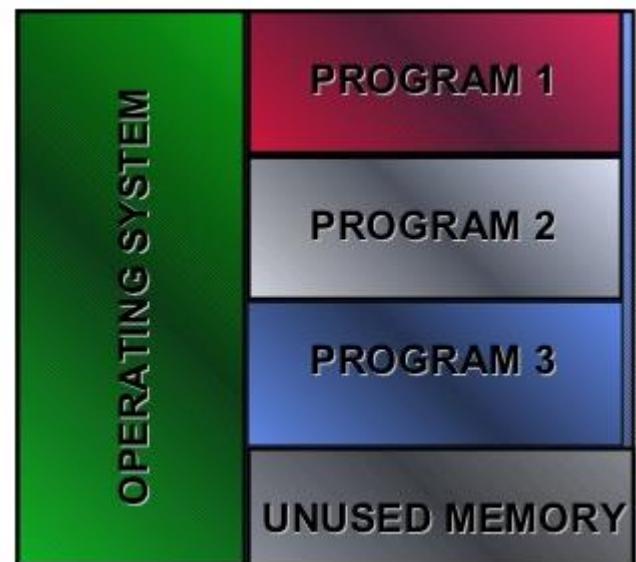
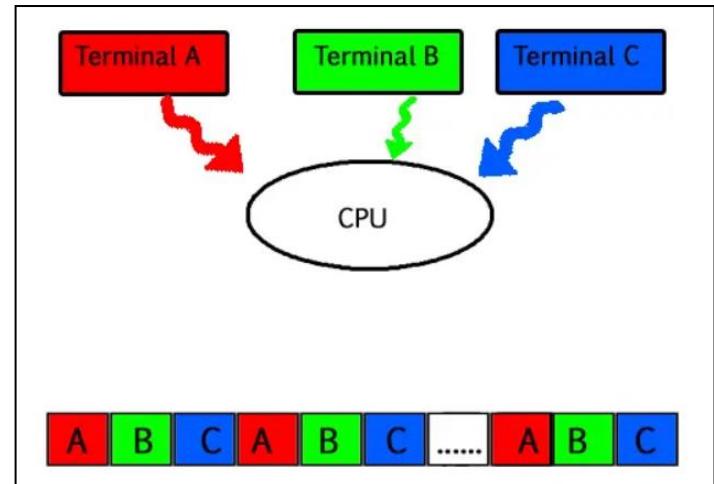
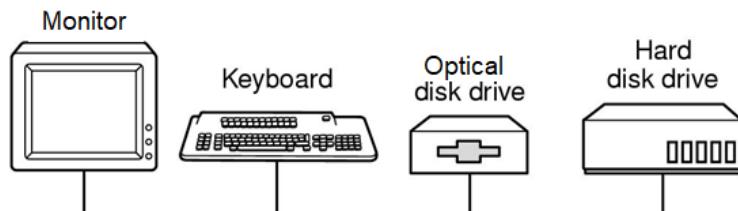


How to organise what's in the different layers?

What does an OS do? (1)

Resource management

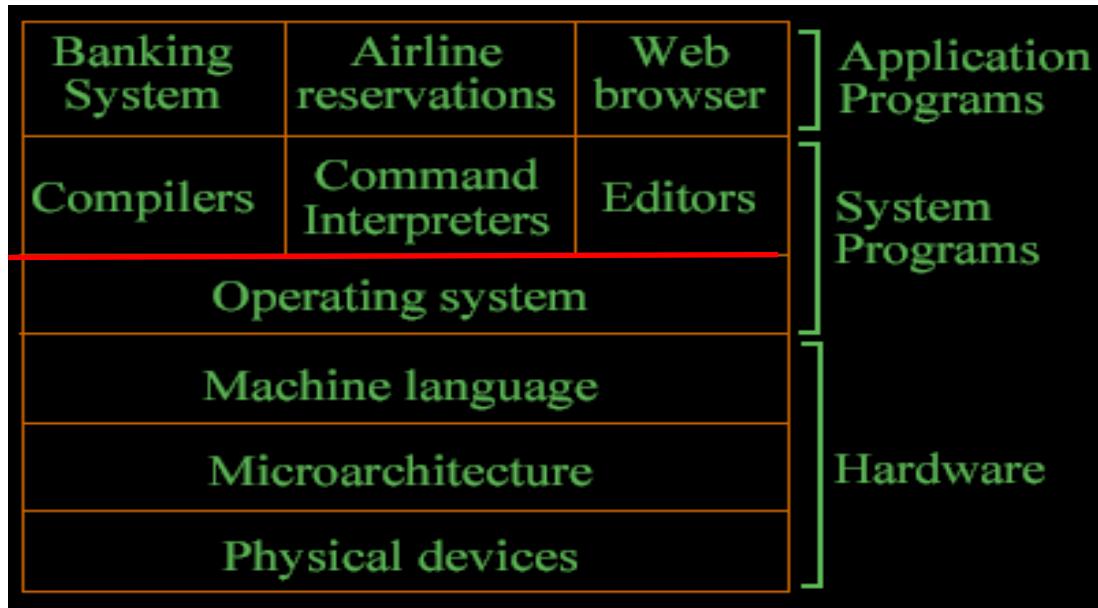
- Virtualization (sharing)
 - Time (CPU)
 - Space (memory)
- Concurrency
- Persistence (I/O)
- Protection



What does an OS do? (2)

“Extended machine” abstraction

- Hides complex details
- Protects the machine from faulty/malicious code



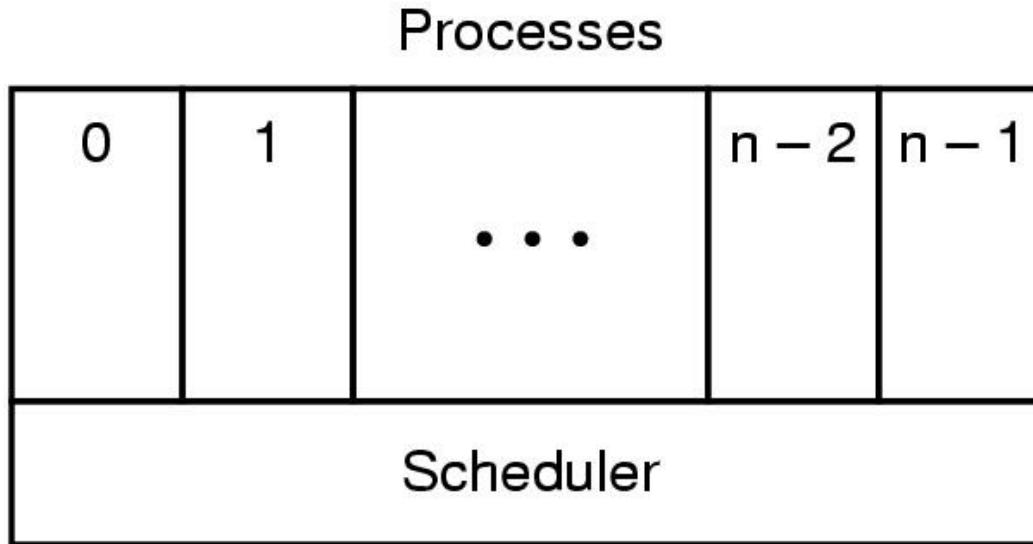
```
310 LET A=USR 16288
320 IF INKEY$="" THEN GOTO 320
330 POKE 17901, INT (RND*128)
335 IF PEEK 21623 <> 178 THEN POK
E (PEEK 16395+128*PEEK 16397) +6
6) 0
340 POKE 16522, CODE INKEY$
350 LET A=USR 16224
360 IF PEEK 16519 = 128 THEN GOT
O 5000
370 FOR N=0 TO 30
380 NEXT N
390 IF PEEK (PEEK 16514+17920) <
>45 THEN GOTO 330
400 FOR N=0 TO 30
405 POKE 17901, INT (128*RND)
410 LET A=USR 16224
420 LET M=0 TO 3
425 NEXT N
430 NEXT N
440 CLS
450 GOTO 2520
370 FOR N=0 TO 0L
```

NO!!!

Core OS concepts

- **Processes**
- **System calls and kernel mode**
- **Address spaces**
- Files, I/O and protection
- The shell

The typical process model: centralised control



- Processes are normal, sequential code
- The scheduler decides what runs, when
- Alternative cooperative models exist

OS API: system calls

Process management

Call	Description
pid = fork()	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process' core image
exit(status)	Terminate process execution and return status

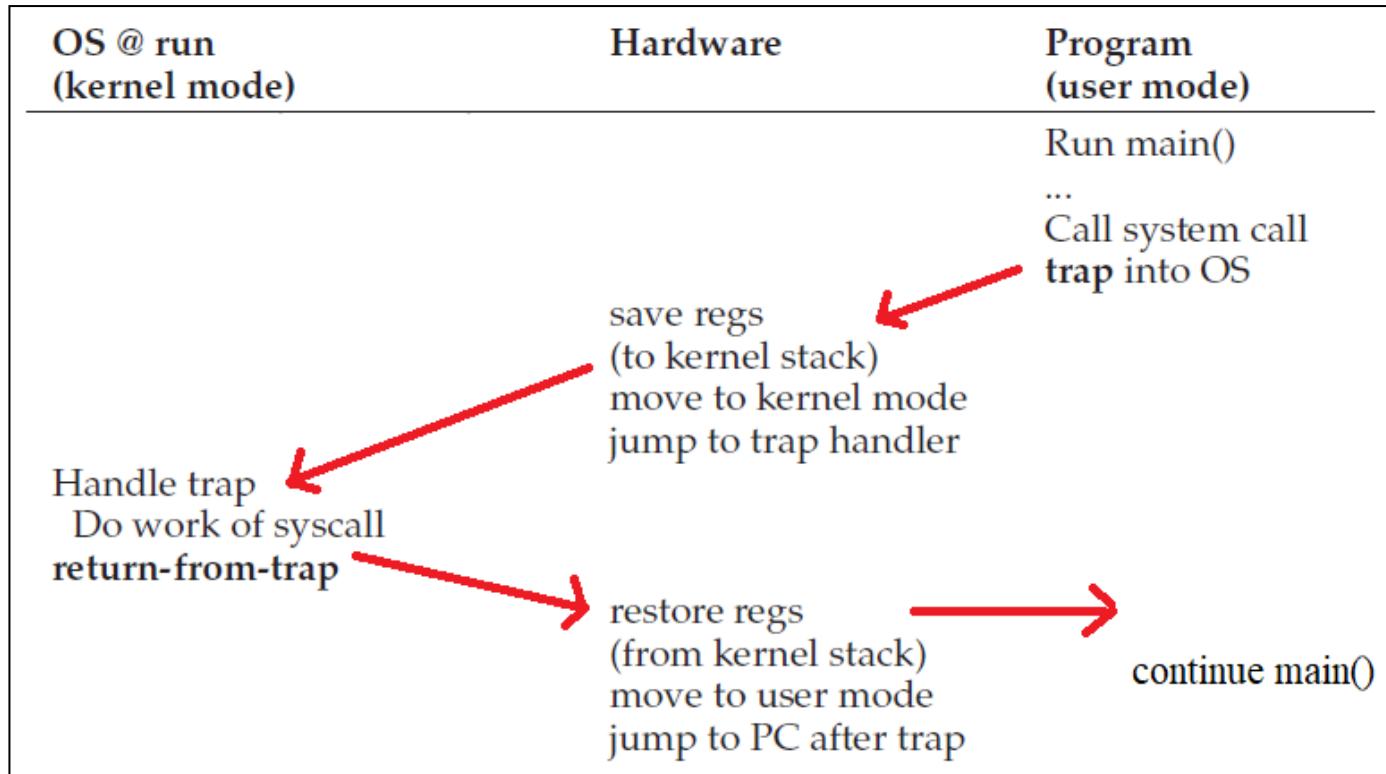
File management

Call	Description
fd = open(file, how, ...)	Open a file for reading, writing, or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information

Directory and file system management

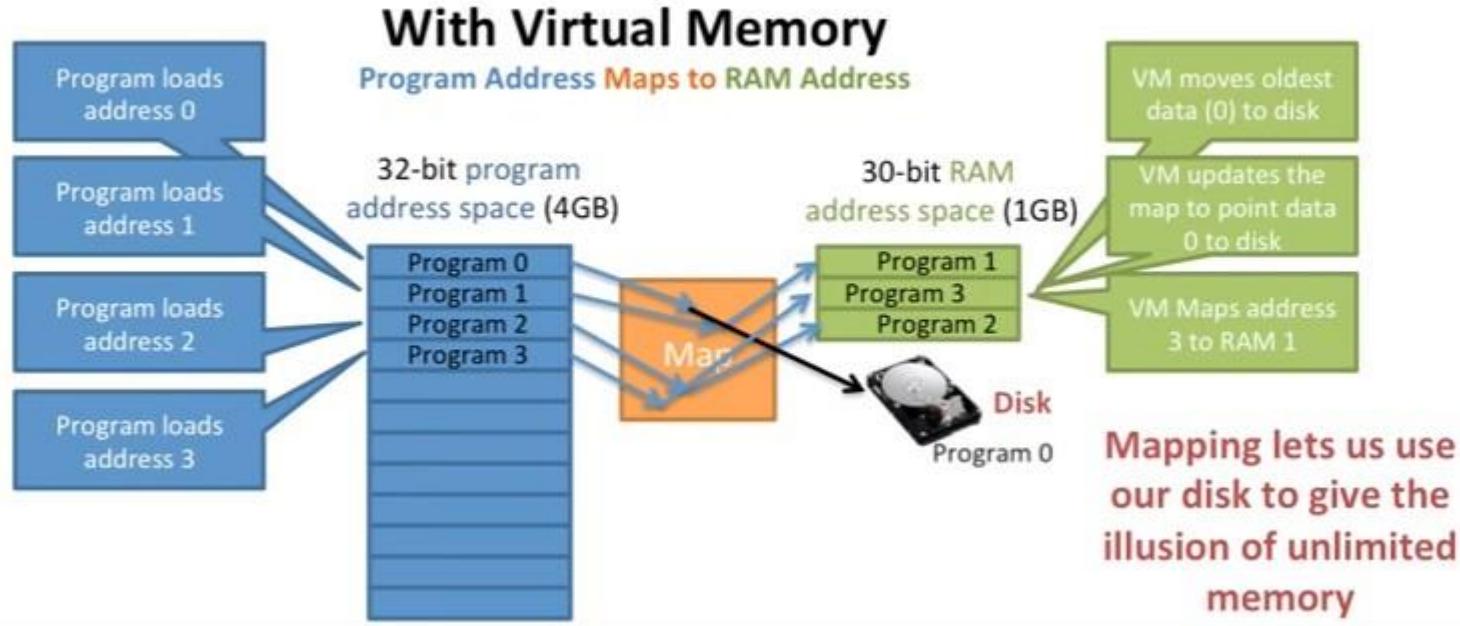
Call	Description
s = mkdir(name, mode)	Create a new directory
s = rmdir(name)	Remove an empty directory
s = link(name1, name2)	Create a new entry, name2, pointing to name1
s = unlink(name)	Remove a directory entry
s = mount(special, name, flag)	Mount a file system
s = umount(special)	Unmount a file system

Kernel mode



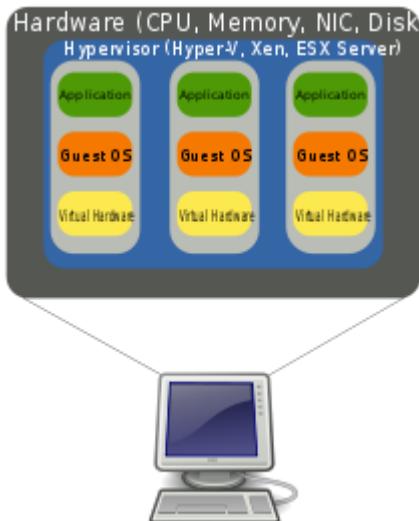
- **User has no direct access to the low-level routines**
- OS runs in kernel mode with higher privileges
- User mode system calls execute a TRAP instruction
- Hardware looks up the **Trap Table** to find address

Address spaces



Modern operating systems have **Virtual Memory**:

- Multiple programs in memory at once
- Idle memory can be paged/swapped to disk
- Gives illusion of “unlimited” memory (at a price)



Virtualization

Virtualization:

- run multiple operating systems simultaneously
- Originally on separate CPUs

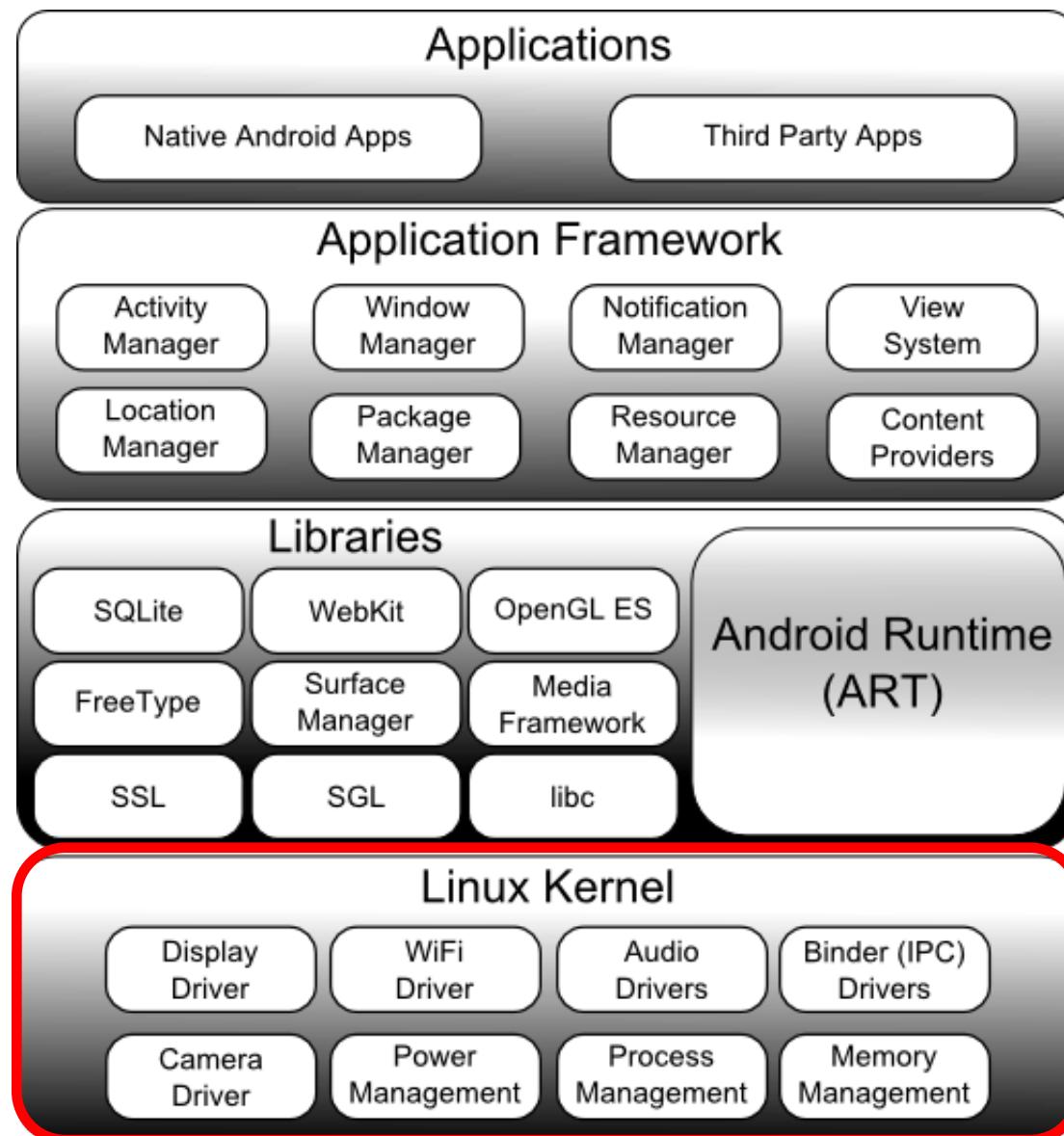
Teleportation: Virtual Machine is temporarily stopped and then resumed on a *different computer*

Products: Parallels, **VirtualBox**, Virtual Iron, Virtual PC, Hyper-V, **VMware** KVM, QEMU, Adeos, Mac-on-Linux, Win4BSD, Win4Lin Pro, vBlade, . . .

What's *not* an operating system?

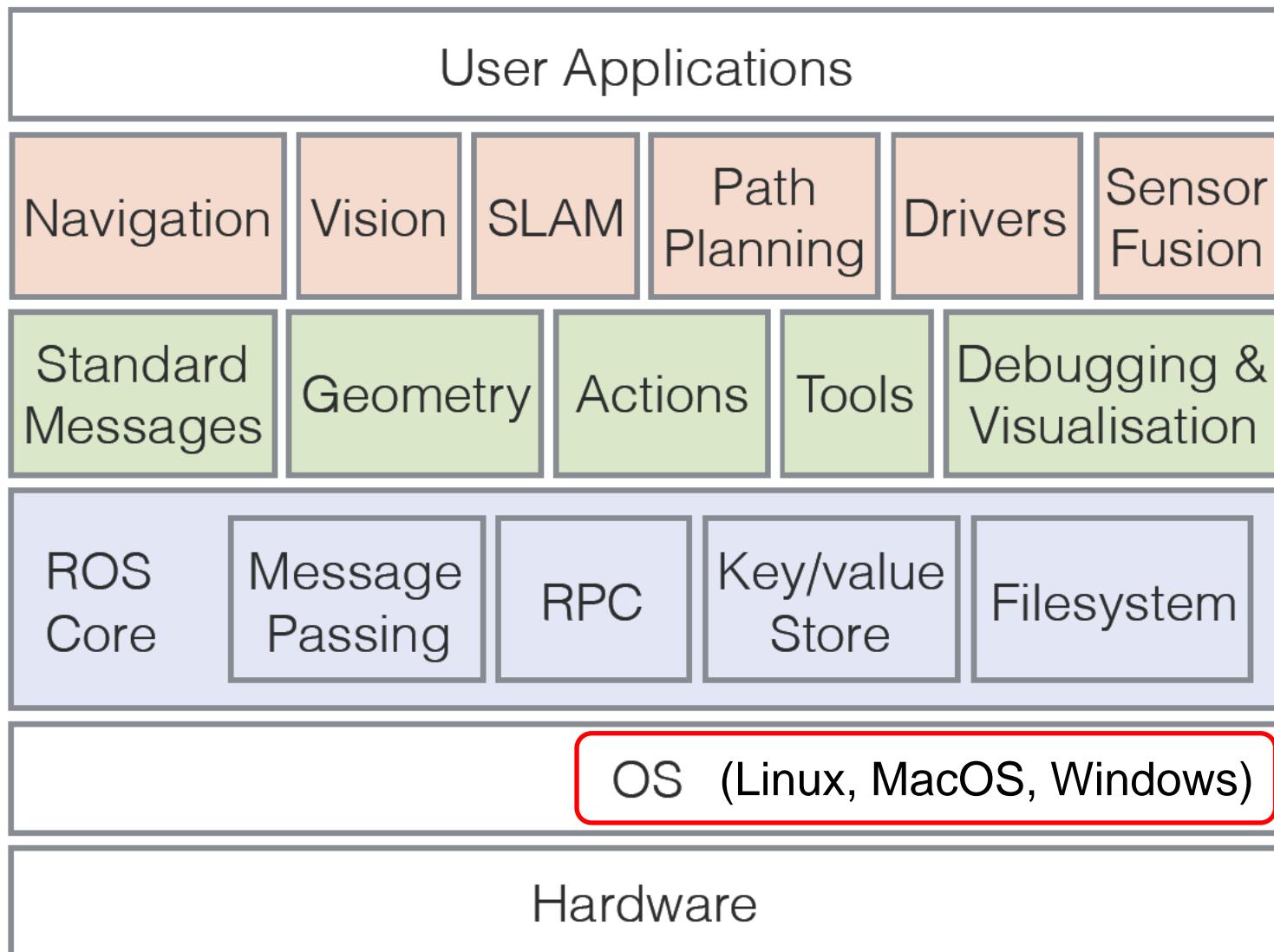


Android



- **NOT an operating system**
 - **Software stack for mobile devices running on (cut-down) Linux**
 - **Core libraries, middleware, user interface and applications**
- **Applications run as separate processes running Android Run Time (ART)**

Robot Operating System (ROS)



Sample Exam Question

If any of the following is not an operating system, explain why not :

- Linux
- Windows
- Android
- ROS
- macOS
- DOS
- iOS
- Arduino

Sample Exam Answer

If any of the following is not an operating system, explain why not:

- Linux
- Windows
- **Android**
 - Android Run Time relies on Linux kernel
- **ROS**
 - Libraries & tools to build robot apps & runs on Linux, MacOS or Windows
- macOS
- DOS
- iOS
- **Arduino**
 - IDE and set of C functions to build tiny runtime to run on small processors & memory. But no context switching or memory management – i.e. does not run on an OS. (Arduino also includes hardware spec.)

Operating Systems – Why?

- Operating systems are *everywhere*, in all shapes and sizes
 - PC, Smartphone
 - Game console, Hand-held
 - TV, Thermostat
 - Car
- While you may never write an OS (but you *may!*), you *will* develop software *for* an OS
- Understanding operating systems will make you a better computer scientist/software engineer
 - Use your computer more efficiently
 - Steal the OS tricks to speed up your code

“I want to make computers dance for me.”

Getting the most out of your computer(s)

FAST!!!

Cheap!!!

Reliable!!!

(pick two)



The need for speed

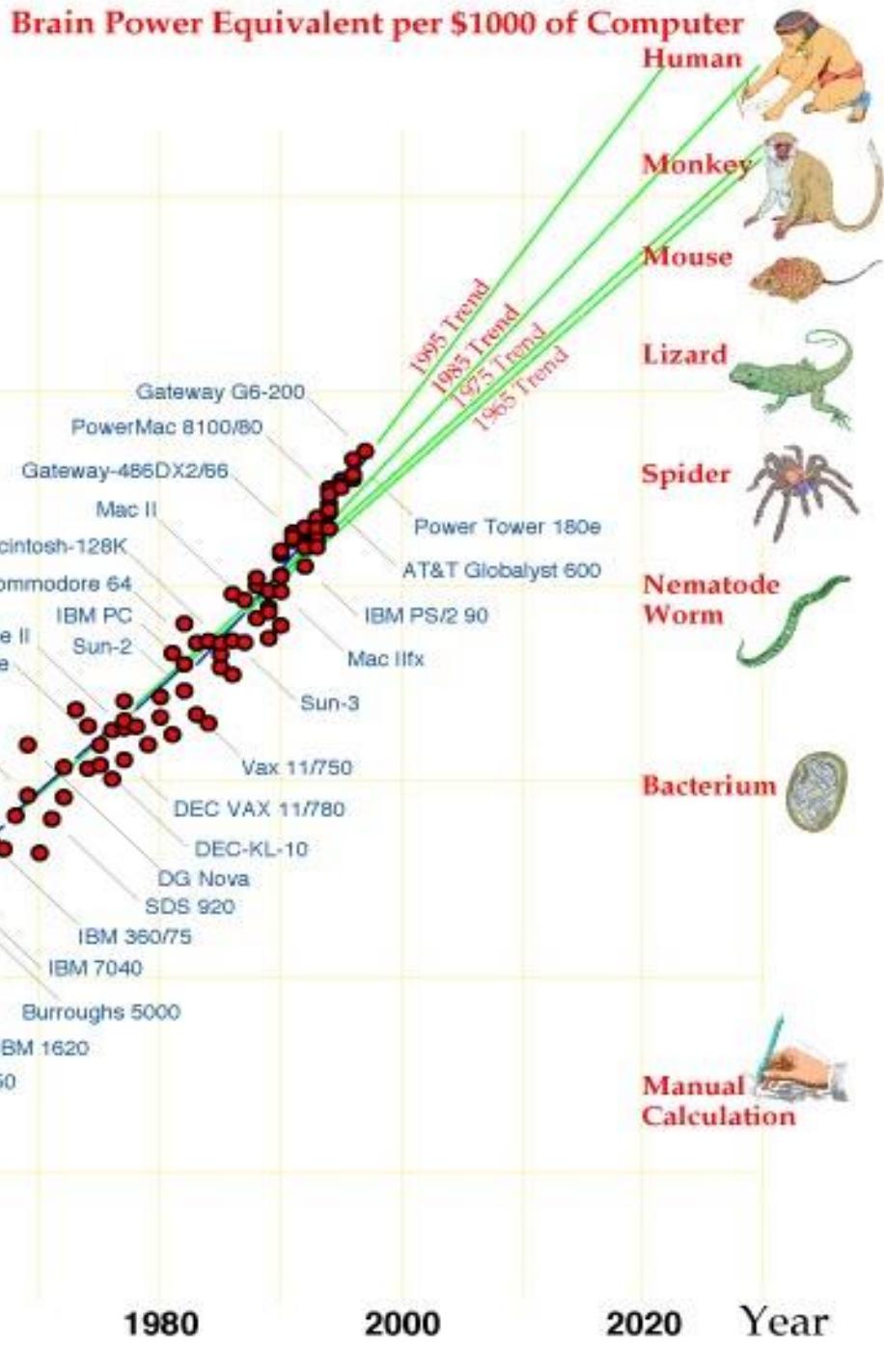
- 1980: 3MHz, real-time games (lo res)
- 1990: 16MHz, AI programs
- 2000: 1GHz, Computer Vision in real-time
- 2010: 3.9GHz 6-core (32nm)
- **2021:** Intel Core i9-10900K 5.3GHz, 10-core: fast deep learning inference on your laptop
- High-performance clusters such as NeSI can boost these numbers by 1000 times for:
 - Climate and weather modelling
 - Massive image recognition training
 - Deep learning for nationwide automated mapping

Moore's law at work

	Capacity	Speed
Logic	2x in 3 years	2x in 3 years
RAM	4x in 3 years	2x in 10 years
Disk	4x in 3 years	2x in 10 years

- Die size: 2x every 3 yrs
- Line width: halve / 7 yrs

Evolution of Computer Power/Cost



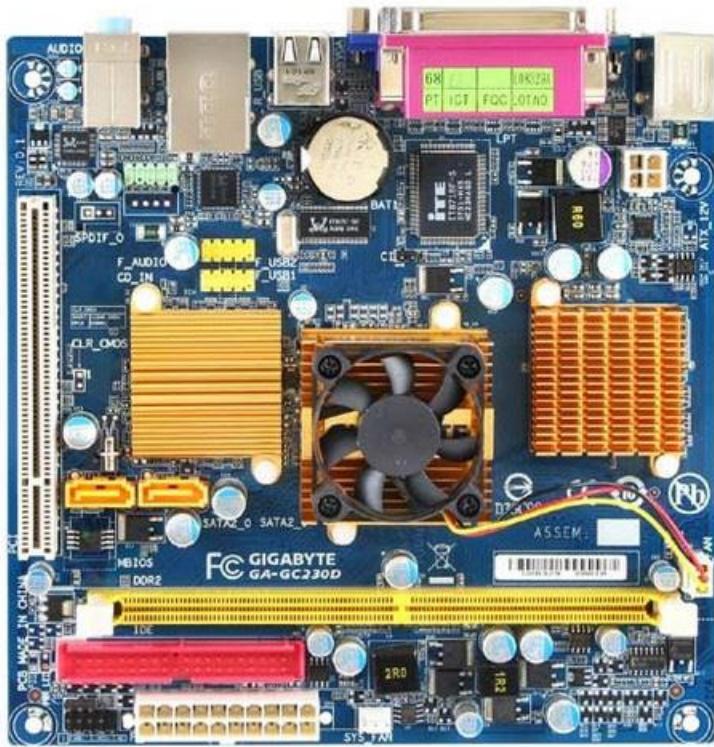
Aside: metric units

Exp.	Explicit	Prefix	Exp.	Explicit	Prefix
10^{-3}	0.001	milli	10^3	1,000	Kilo
10^{-6}	0.000001	micro	10^6	1,000,000	Mega
10^{-9}	0.000000001	nano	10^9	1,000,000,000	Giga
10^{-12}	0.000000000001	pico	10^{12}	1,000,000,000,000	Tera
10^{-15}	0.000000000000001	femto	10^{15}	1,000,000,000,000,000	Peta
10^{-18}	0.0000000000000001	atto	10^{18}	1,000,000,000,000,000,000	Exa
10^{-21}	0.00000000000000001	zepto	10^{21}	1,000,000,000,000,000,000,000	Zetta
10^{-24}	0.0000000000000000000000000001	yocto	10^{24}	1,000,000,000,000,000,000,000,000,000	Yotta

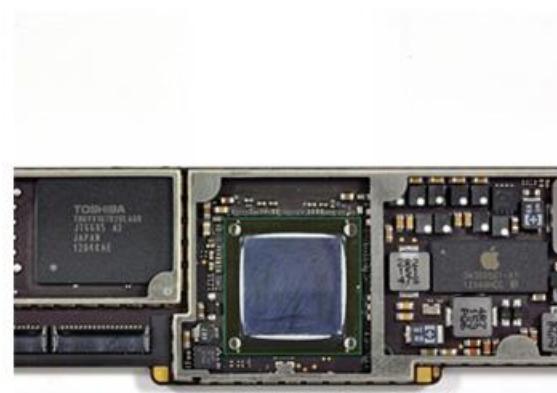
The metric prefixes

Max **64 bit** number = **18,446,744,073,709,551,615**
Virtual memory: access 18,000 **Petabytes** of memory
Or **access 18 million 1 Terabyte hard disks**

System on a chip (SoC) vs CPU the battle for the future of computing



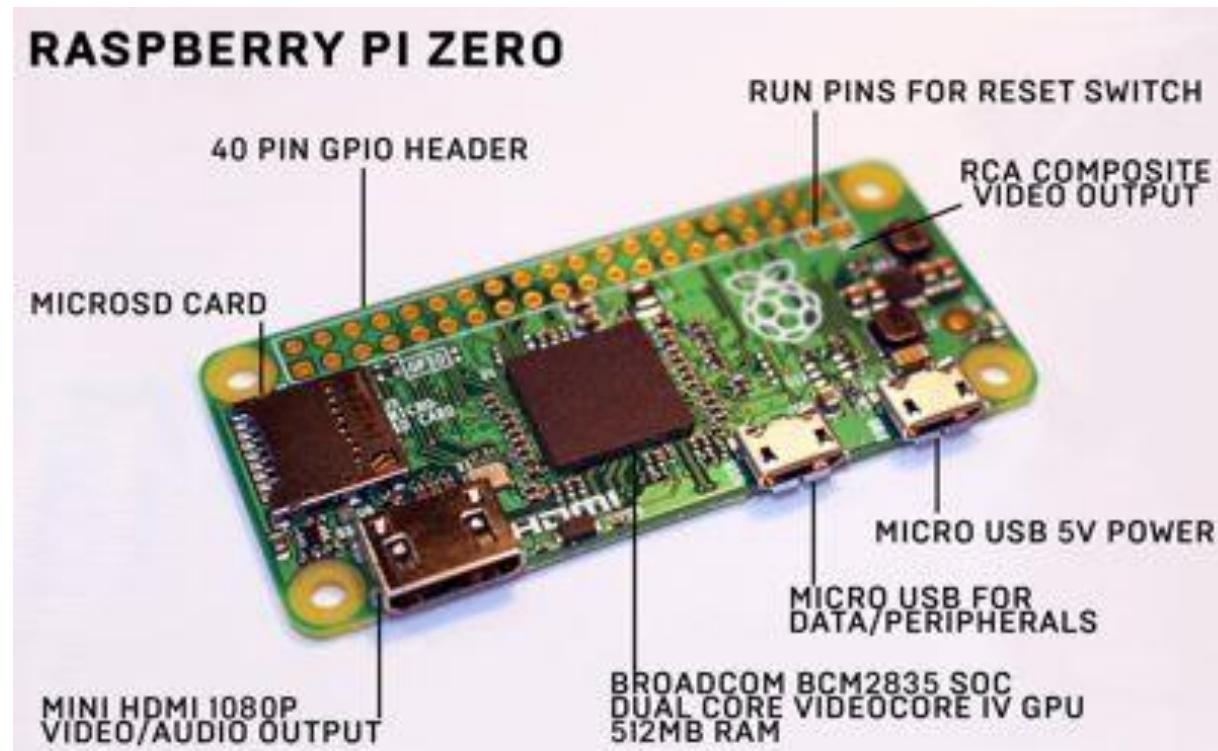
CPU computer



SoC computer (iPad)

SoC: \$5 Raspberry Pi Zero

- **1GHz** ARM11 core, GPU
- 512MB RAM
- 64GB micro card
- HDMI out
- USB2
- **Linux**



More FLOPS: GPGPUs

2021: Nvidia GeForce RTX 3090

- 10,496 CUDA cores, 328 tensor cores
- 24GB GDDR6X memory, 384-bit memory bus
- Up to 100 x CPU performance for some tasks
 - Parallel programming required



Intel IPP:

Integrated Performance Primitives

Intel software library of optimised primitives for:

- Vector/Matrix Mathematics
- Computer Vision
- Data Compression and Cryptography
- Ray Tracing/Rendering, Image Colour Conversion
- Signal Processing
- Speech Coding, Recognition
- String Processing

Achieve speedup via data parallelism

(one operation on multiple data items)

- Single Instruction, Multiple Data (SIMD) instructions using Streaming SIMD Extensions (SSE)
- Implemented in silicon

Counterparts: Sun: mediaLib for Solaris, Apple: vDSP, vImage etc.
for Mac OS, AMD: AMD Performance Library (APL)

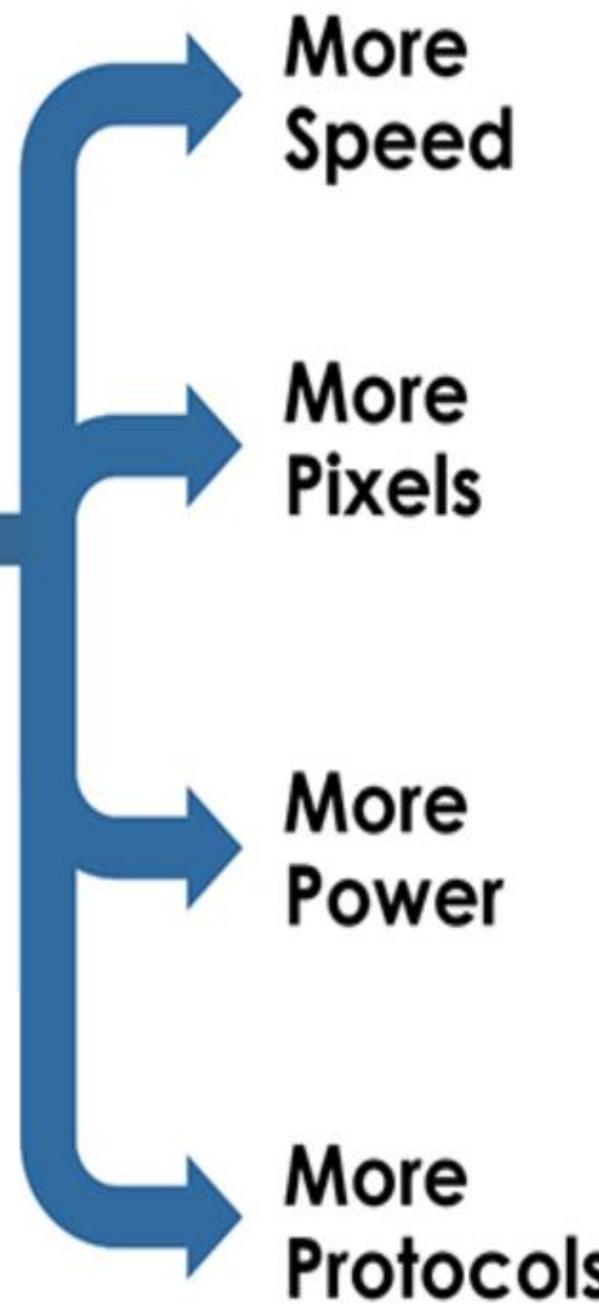
The Hard Drive over 65 years: data is king



- **1956:** IBM RAMAC 305
 - 5MB of data at \$10,000 a megabyte.
 - As big as two refrigerators
 - 50 24-inch (61cm) platters
- **1991:** Apple powerbook 100: 40MBytes
- **2021:**
 - 64 gigabyte USB stick
 - 20 terabyte HDD (Seagate HAMR)
 - 100 terabyte SSD Nimbus Data (\$40K)
 - Large institutions use **petabytes** of data



USB-C



40 Gbps



Two 4k



Up to 100w



OS for the next-gen hardware

Microsoft Windows 11

- announced June 2021
- built for desktop and mobile devices - easier for developers to create native applications that run across devices

Apple macOS 12 Monterey

- announced June 2021 – introduces Shortcuts, Universal Control (single keyboard and mouse can interact with multiple Macs and iPads at once)

Apple iOS 15 announced June 2021

Google Android 12 available late 2021

Linux distributions: one for every user

Overall: Ubuntu, Server: CentOS, Gaming: Fedora Games Spin,
Lightweight: Lubuntu, for Programmers: Fedora, Beginner-Friendly:
Manjaro (or Mint), Best-Looking: elementary OS, for Windows Users:
Robolinux, for Kids: Sugar on a Stick (SoS) ...

Getting the most out of your computer(s)

FAST!

Cheap!

Reliable!

(Pick two)



The need for speed

- 1980: 3MHz, real-time games (low resolution)
- 1990: 16MHz, AI programs
- 2000: 1GHz, Computer Vision in real-time
- 2010: 3.9GHz 6-core
- **2021:** Intel Core i9-10900K 5.3GHz, 10-core:
real-time video object detection on your laptop
- High-performance clusters such as NeSI can boost these numbers by 1000 times for:
 - Climate and weather modelling
 - Massive image recognition training
 - Deep learning for nationwide automated mapping

Moore's law at work

	Capacity	Speed
Logic	2x in 3 years	2x in 3 years
RAM	4x in 3 years	2x in 10 years
Disk	4x in 3 years	2x in 10 years

- Die size: 2x every 3 yrs
- Line width: halve every 7 yrs

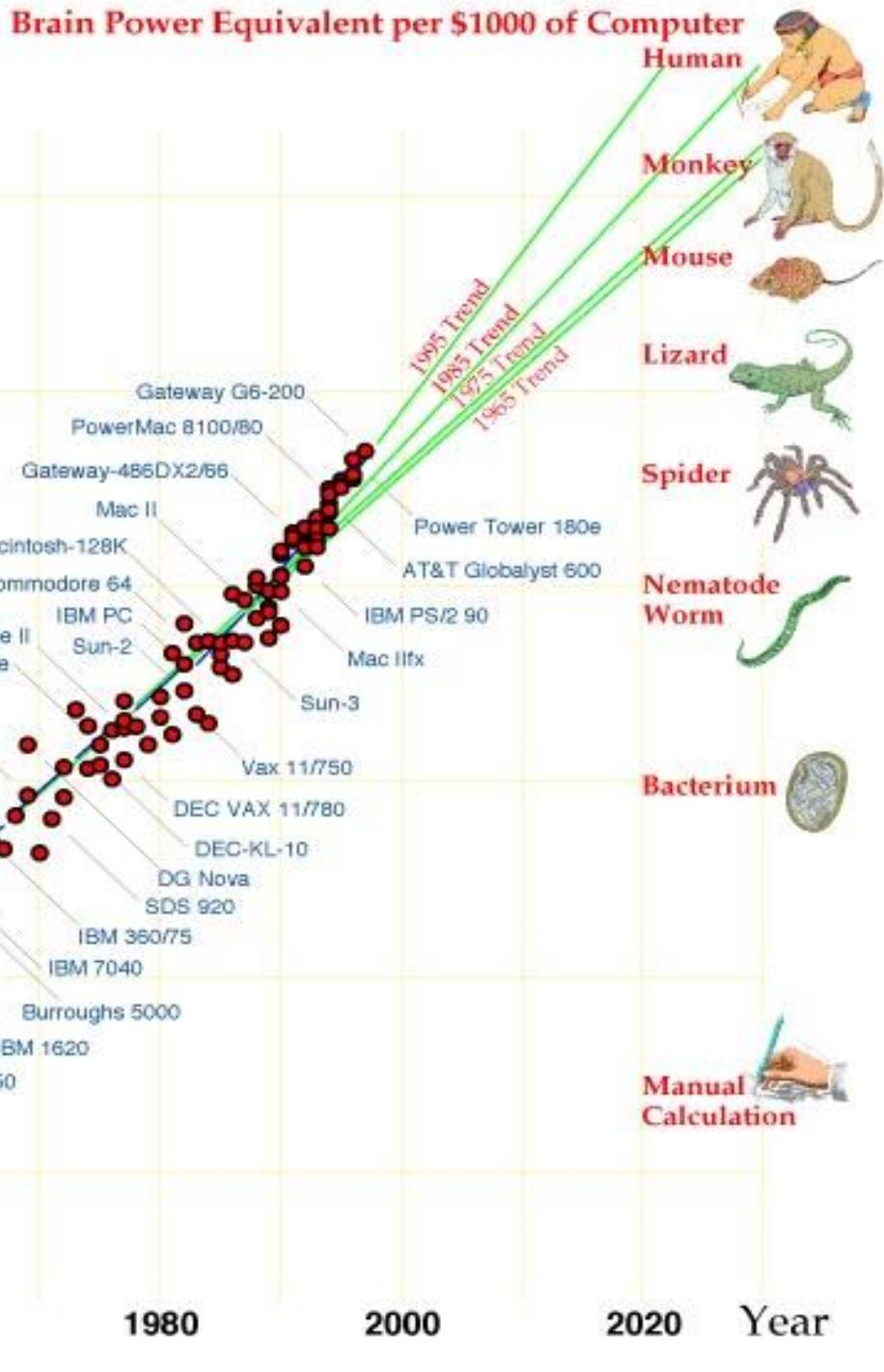
Aside: metric units

Exp.	Explicit	Prefix	Exp.	Explicit	Prefix
10^{-3}	0.001	milli	10^3	1,000	Kilo
10^{-6}	0.000001	micro	10^6	1,000,000	Mega
10^{-9}	0.000000001	nano	10^9	1,000,000,000	Giga
10^{-12}	0.000000000001	pico	10^{12}	1,000,000,000,000	Tera
10^{-15}	0.000000000000001	femto	10^{15}	1,000,000,000,000,000	Peta
10^{-18}	0.0000000000000001	atto	10^{18}	1,000,000,000,000,000,000	Exa
10^{-21}	0.00000000000000001	zepto	10^{21}	1,000,000,000,000,000,000,000	Zetta
10^{-24}	0.0000000000000000000000000001	yocto	10^{24}	1,000,000,000,000,000,000,000,000,000	Yotta

The metric prefixes

Max **64 bit** number = **18,446,744,073,709,551,615**
Virtual memory: access 18,000 **Petabytes** of memory
Or **access 18 million 1 Terabyte hard disks**

Evolution of Computer Power/Cost



High compute: computer vision



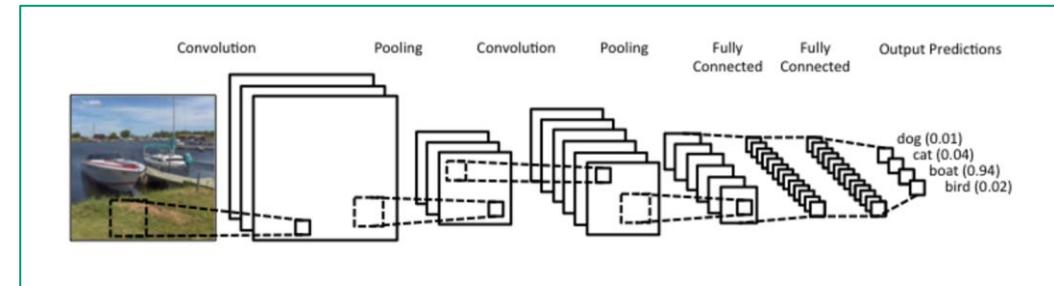
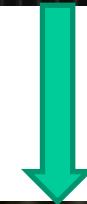
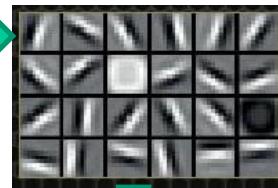
Classification

**Object
detection**

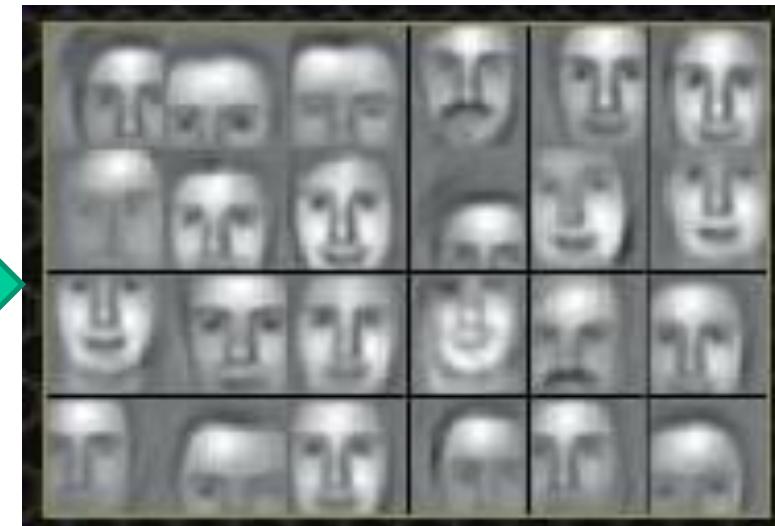


Autonomy

Convolutional Neural Networks



LeNet



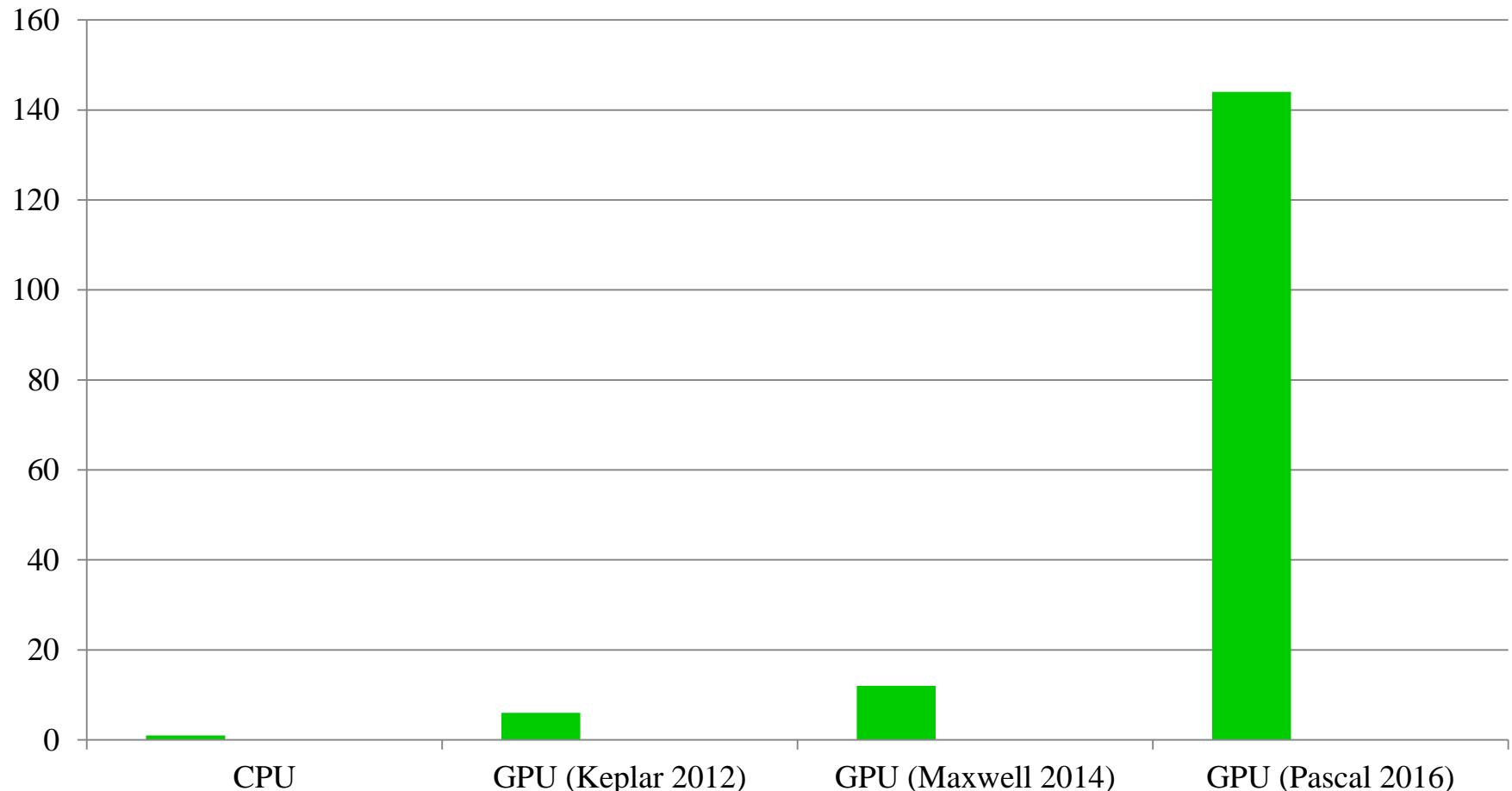
Massively parallel: GPGPUs

2021: Nvidia GeForce RTX 3090

- 10,496 CUDA cores, 328 tensor cores
- 24GB GDDR6X memory, 384-bit memory bus
- Up to 100 x CPU performance for some tasks
 - Parallel programming required



The (GP)GPU revolution



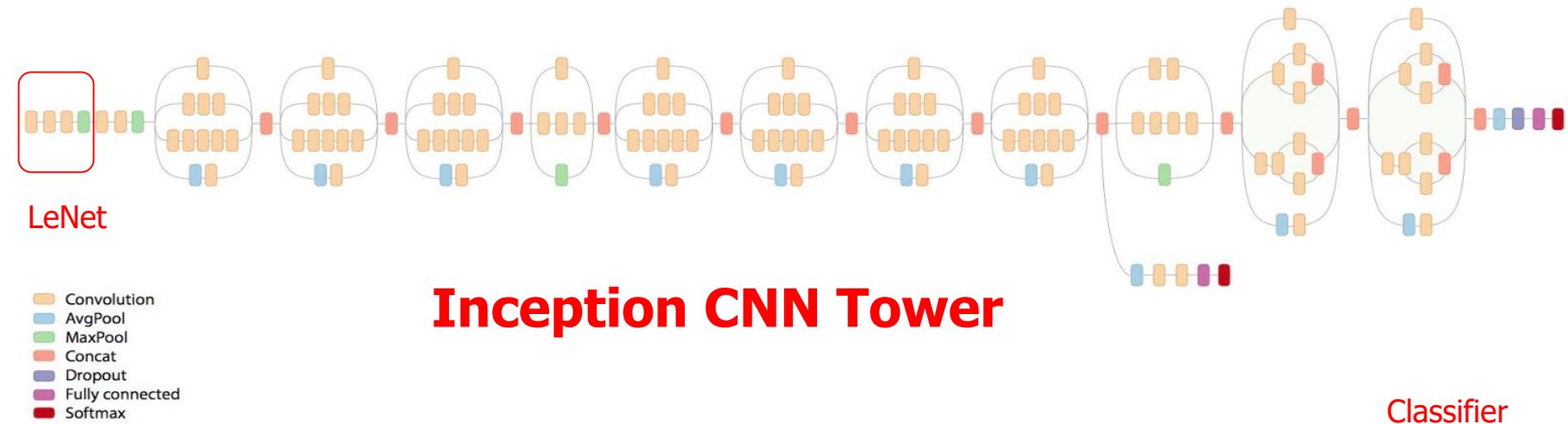
CUDA General-purpose GPUs offer **up to 250x** performance of CPU alone, for about **twice** the price of a CPU

Speed enables new applications



- Highly parallel execution
- Train models on millions of images (e.g. face recognition)
- Classify objects and detect images in video in *real time*

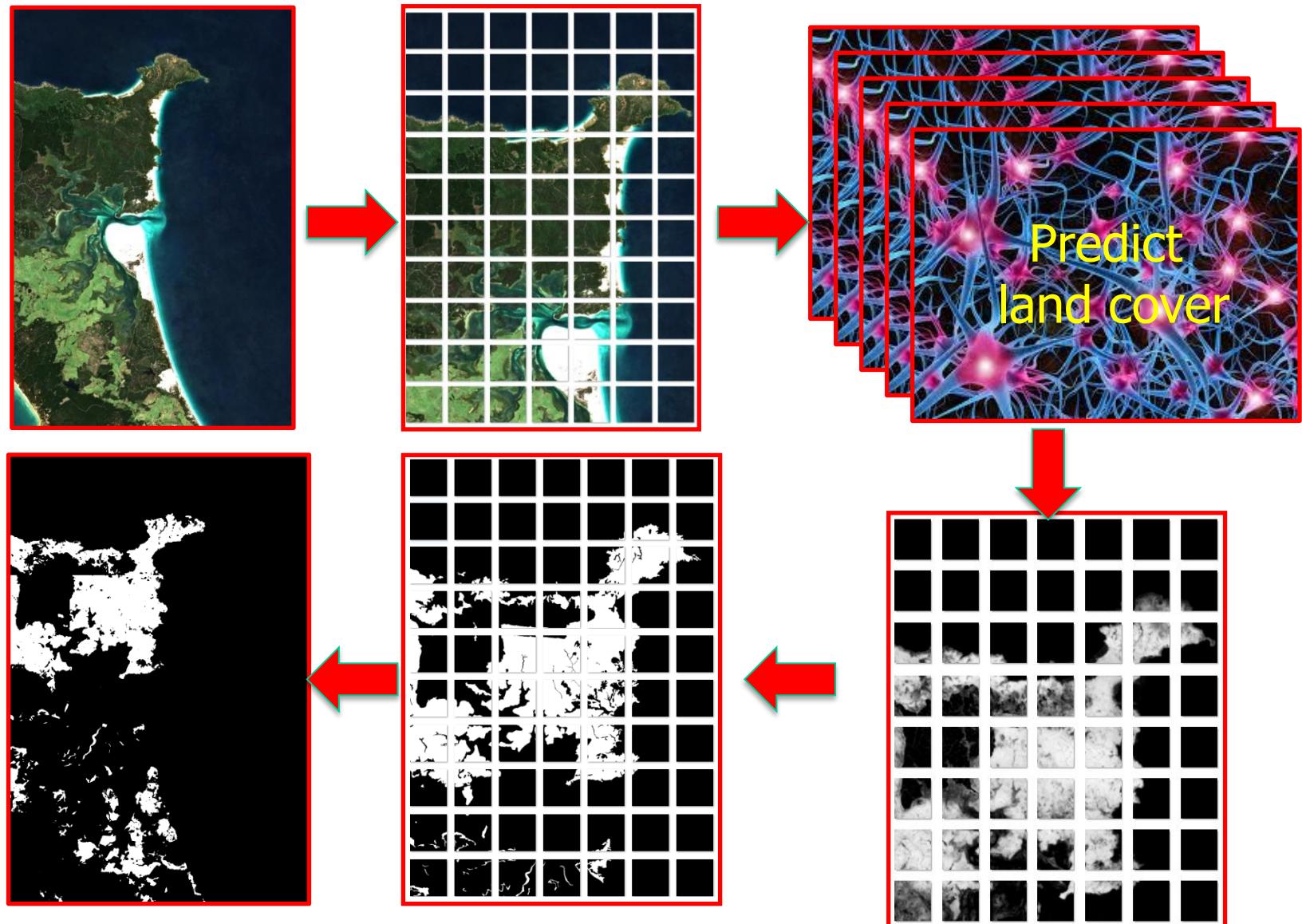
Inception: going deeper



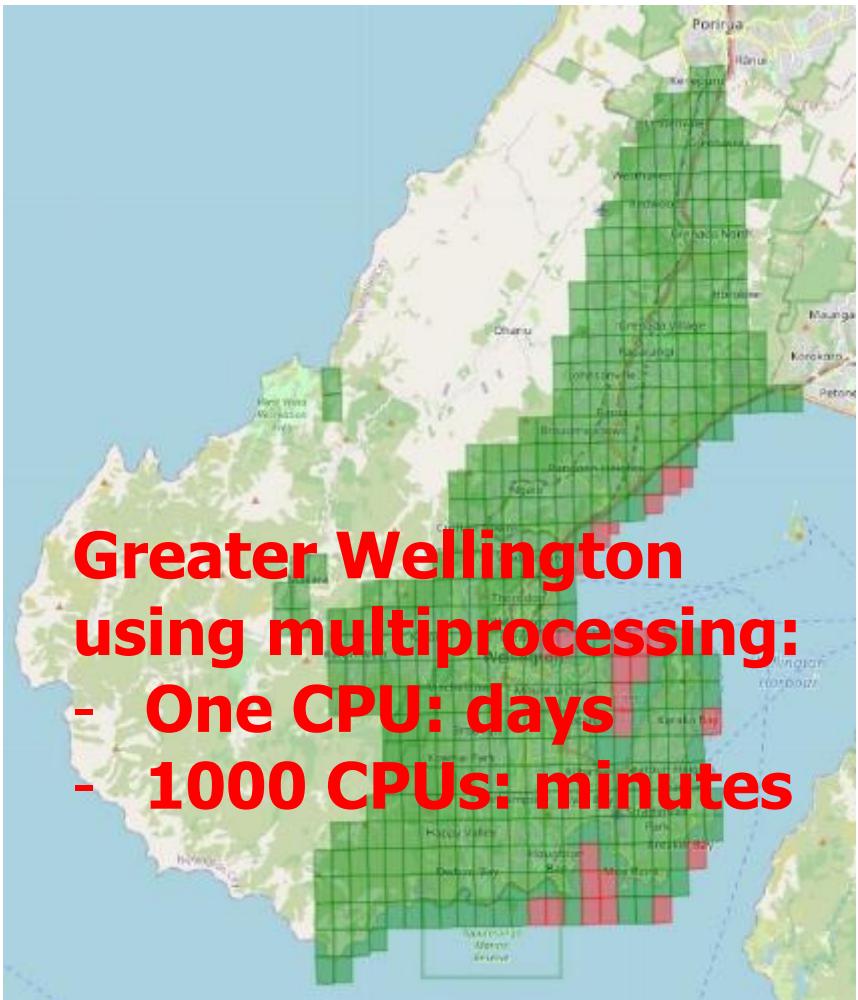
- Complex sub-networks of neuron layers give dramatic improvement in accuracy
- Novel network topologies:
 - Isolate objects in scenes
 - “Deep fake” audio, movies
 - Generate visualisations of your dreams



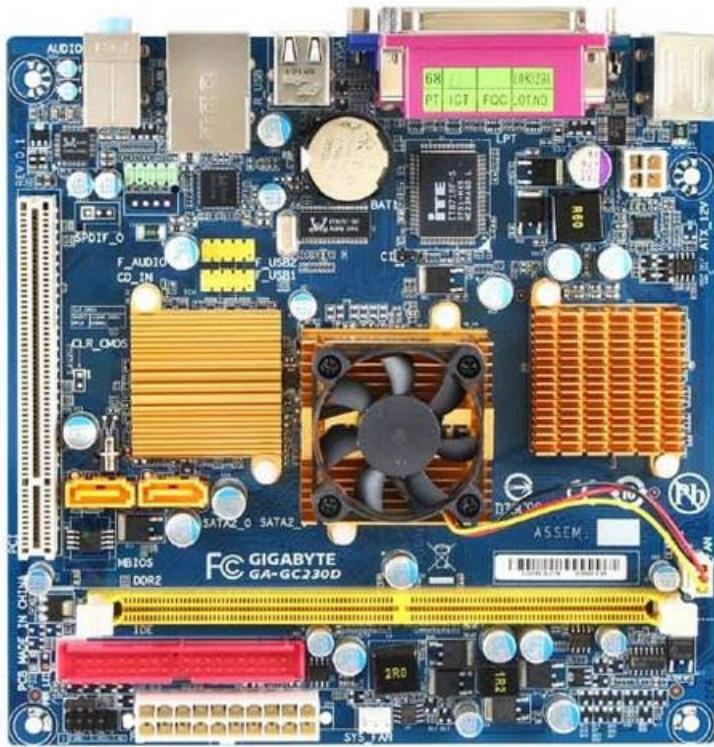
National-scale mapping from satellite



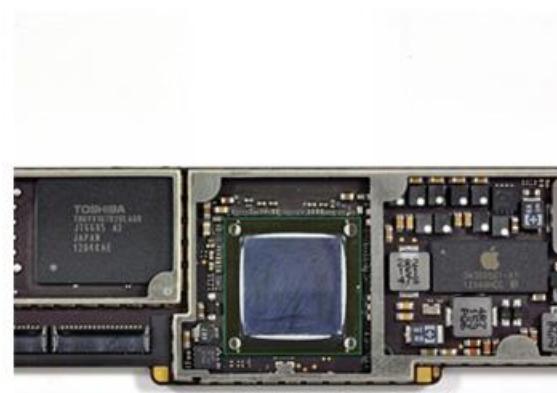
LiDAR point cloud processing



System on a chip (SoC) vs CPU the battle for the future of computing



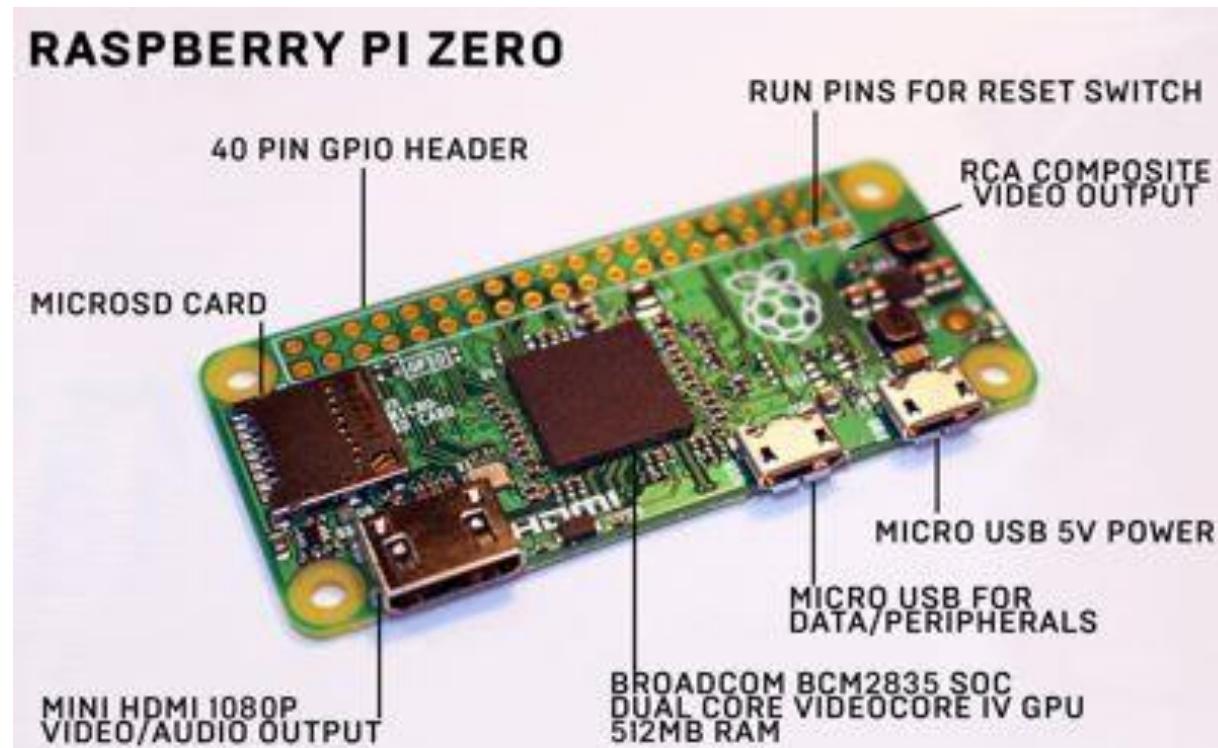
CPU computer



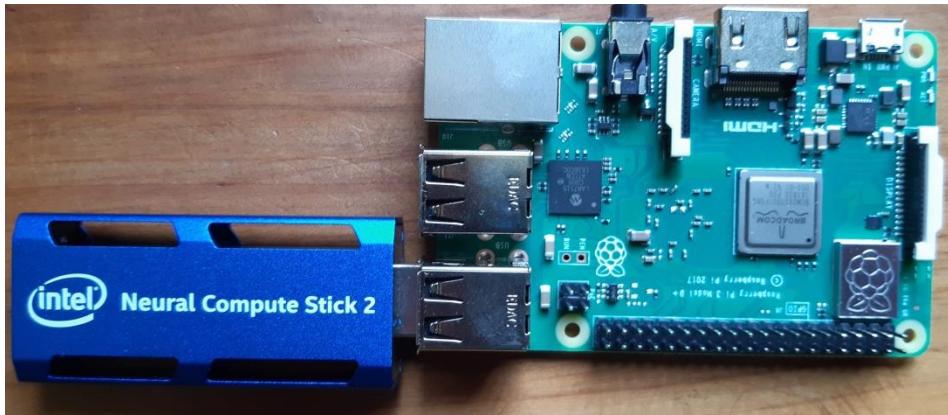
SoC computer (iPad)

SoC: \$5 Raspberry Pi Zero

- **1GHz** ARM11 core, GPU
- 512MB RAM
- 64GB micro card
- HDMI out
- USB2
- **Linux**



Computer vision in the field



**Raspberry Pi and
Neural Compute Stick:
computer vision in
the field**

| **Automatic Classifier**
Artificial Intelligence classifying animals

Intel IPP:

Integrated Performance Primitives

Intel software library of optimised primitives for:

- Vector/Matrix Mathematics
- Computer Vision
- Data Compression and Cryptography
- Ray Tracing/Rendering, Image Colour Conversion
- Signal Processing
- Speech Coding, Recognition
- String Processing

Achieve speedup via data parallelism

(one operation on multiple data items)

- Single Instruction, Multiple Data (SIMD) instructions using Streaming SIMD Extensions (SSE)
- Implemented in silicon

Counterparts: Sun: mediaLib for Solaris, Apple: vDSP, vImage etc.
for Mac OS, AMD: AMD Performance Library (APL)

The Hard Drive over 65 years: data is king

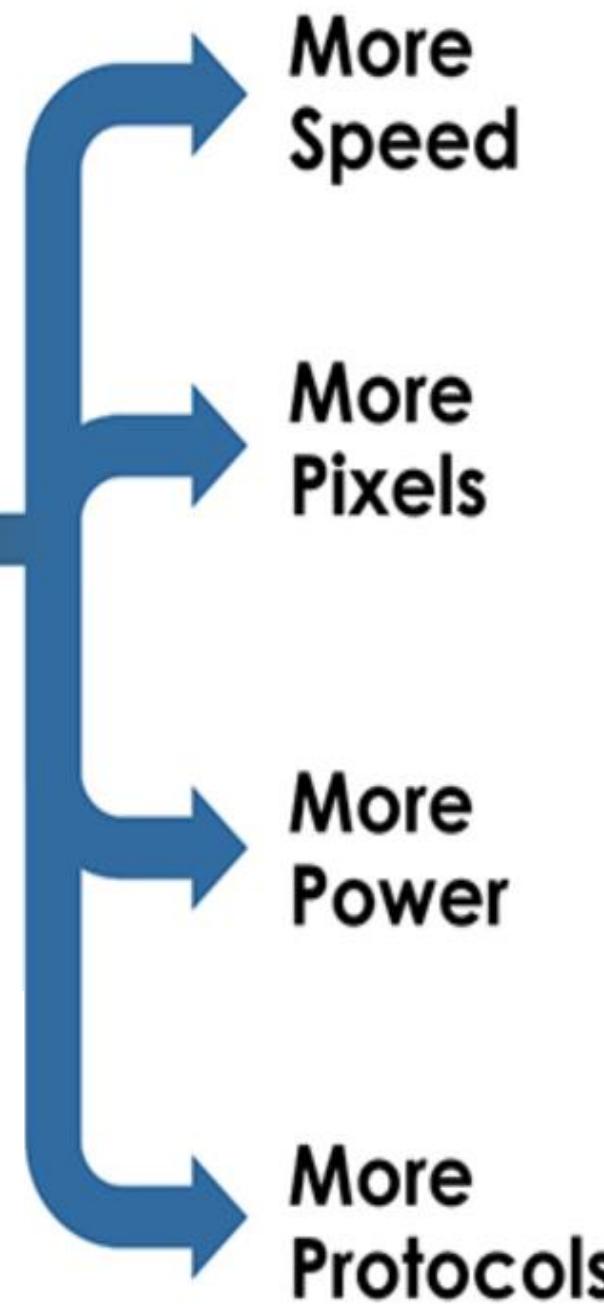
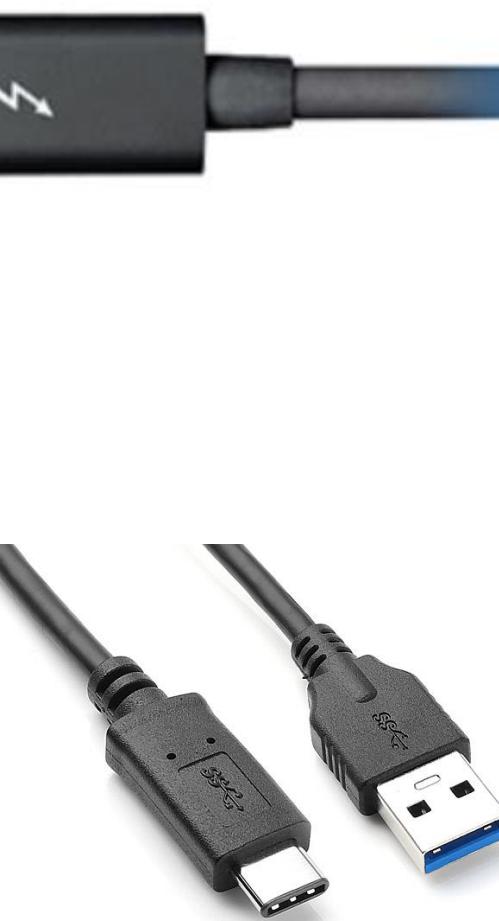


- **1956:** IBM RAMAC 305
 - 5MB of data at \$10,000 a megabyte.
 - As big as two refrigerators
 - 50 24-inch (61cm) platters
- **1991:** Apple powerbook 100: 40MBytes
- **2021:**
 - 64 gigabyte USB stick
 - 20 terabyte HDD (Seagate HAMR)
 - 100 terabyte SSD Nimbus Data (\$40K)
 - Large institutions use **petabytes** of data



The deep learning revolution started with *data*

USB-C



40 Gbps



Two 4k



Up to 100w



OS for the next-gen hardware

Microsoft Windows 11

- announced June 2021
- built for desktop and mobile devices - easier for developers to create native applications that run across devices

Apple macOS 12 Monterey

- announced June 2021 – introduces Shortcuts, Universal Control (single keyboard and mouse can interact with multiple Macs and iPads at once)

Apple iOS 15 announced June 2021

Google Android 12 available late 2021

Linux distributions: one for every user

Overall: Ubuntu, Server: CentOS, Gaming: Fedora Games Spin,
Lightweight: Lubuntu, for Programmers: Fedora, Beginner-Friendly:
Manjaro (or Mint), Best-Looking: elementary OS, for Windows Users:
Robolinux, for Kids: Sugar on a Stick (SoS) ...

UCSA – become a class rep!

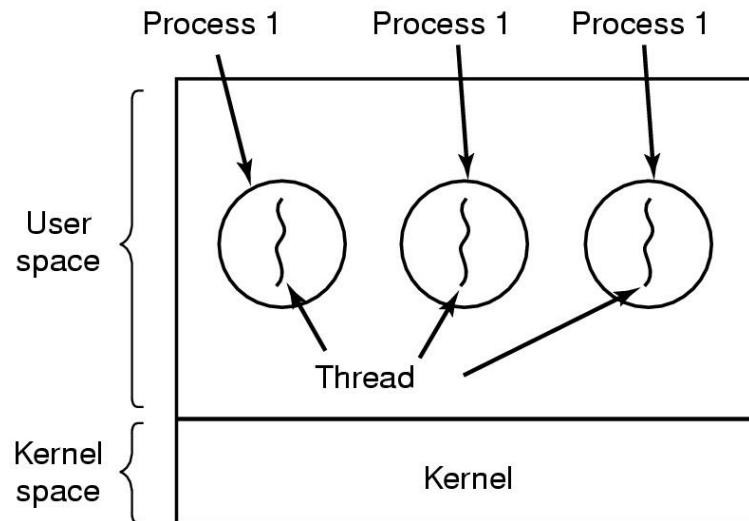


**WE NEED
YOU
TO BE A
CLASS
REP**

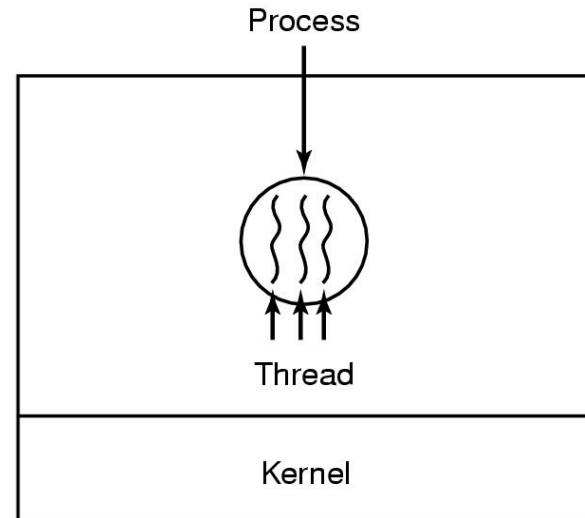
Talk to/email me if you'd like to be a class rep
(brent.martin@canterbury.ac.nz)

ENCE360

Operating Systems



(a)



(b)

Processes and Threads

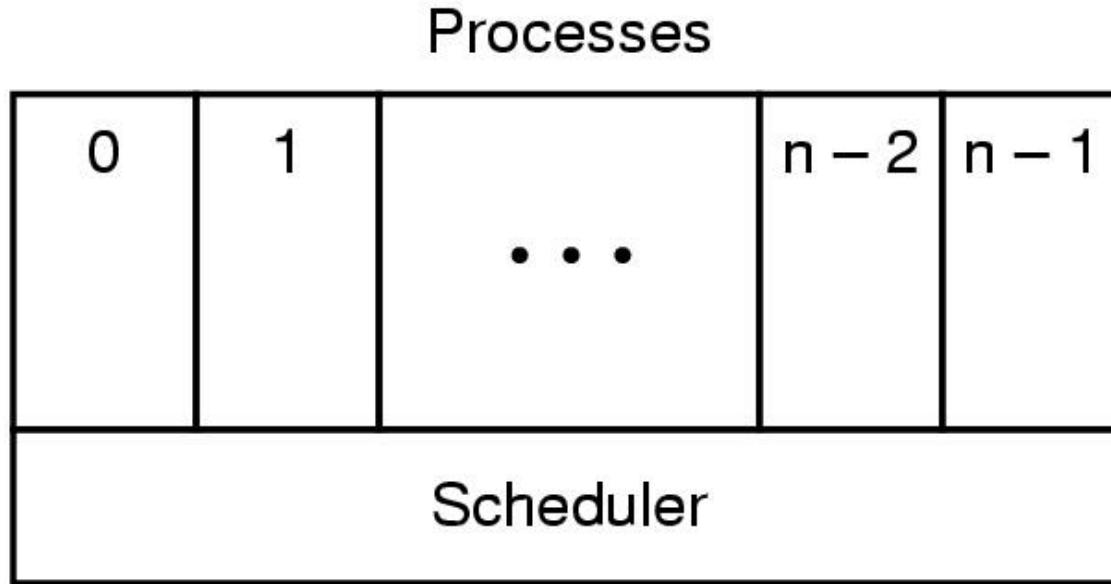


Processes and Threads

- 2.1 Processes
- 2.2 Threads
- 2.3 Interprocess communication
- 2.4 Classical IPC problems
- 2.5 Scheduling

Chapter 2
MODERN OPERATING SYSTEMS (MOS)
*By Andrew Tanenbaum
(+ code from OSTEP)*

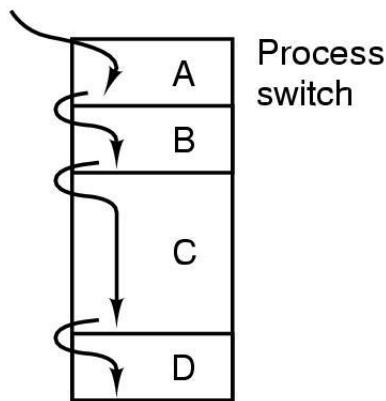
Recap: the process model



- Scheduler handles multiprogramming
- (User) process is sequential (for now!)
- (User) process is a *resource grouping*
 - Has its own context
 - Oblivious to other processes

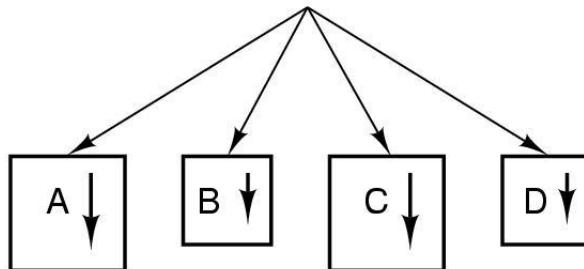
The process model (2)

One program counter

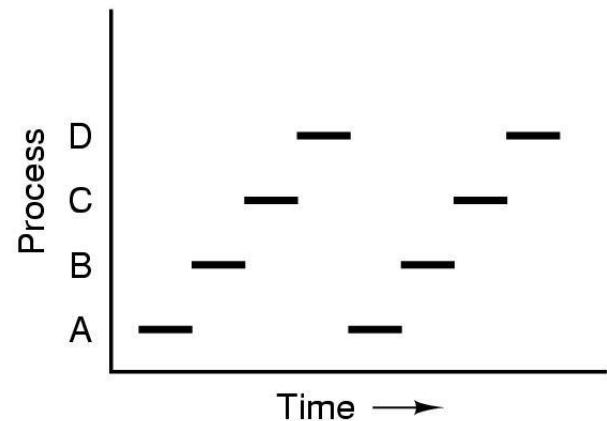


(a)

Four program counters



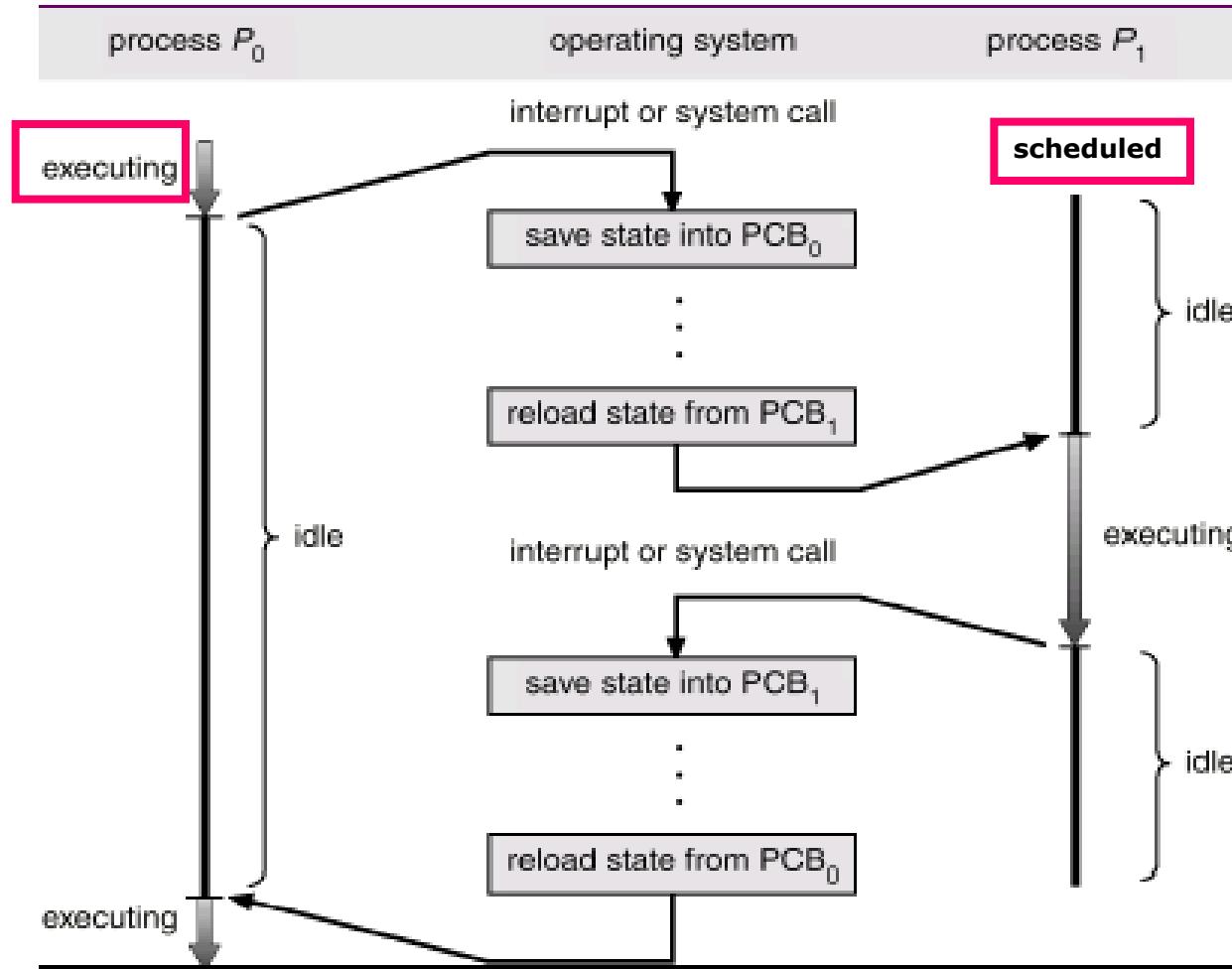
(b)



(c)

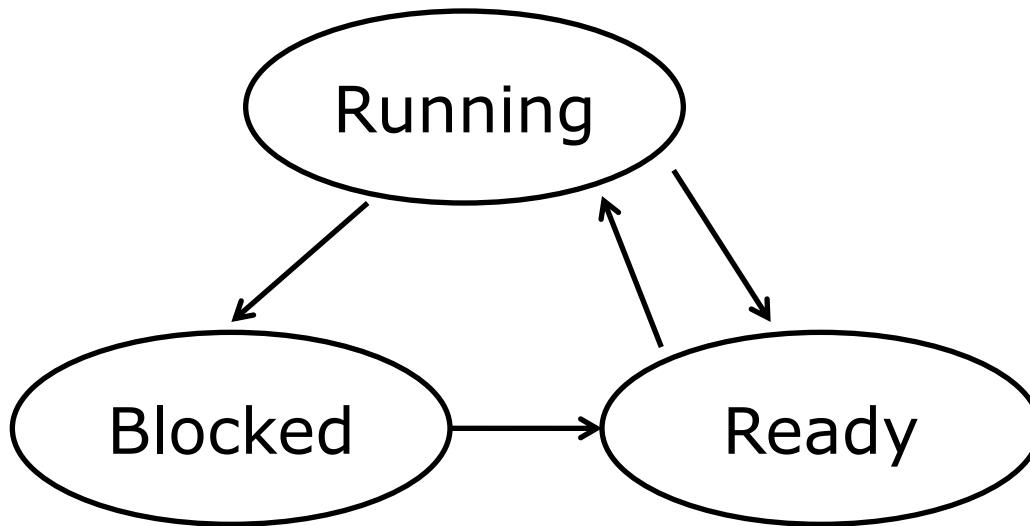
- a) Multiprogramming of four programs
- b) Conceptual model of 4 independent, sequential processes
- c) Only one program active at any instant
 - achieves speed up through better use of CPU time

(Pseudo) parallelism



- **Multiprogramming:** timer interrupt switches processes

Interrupt process switch



1. Hardware stacks program counter, etc.
2. Hardware loads new program counter from interrupt vector.
3. Assembly language procedure saves registers.
4. Assembly language procedure sets up new stack.
5. C interrupt service runs (typically reads and buffers input).
6. Scheduler decides which process is to run next.
7. C procedure returns to the assembly code.
8. Assembly language procedure starts up new current process.

Process Creation/termination

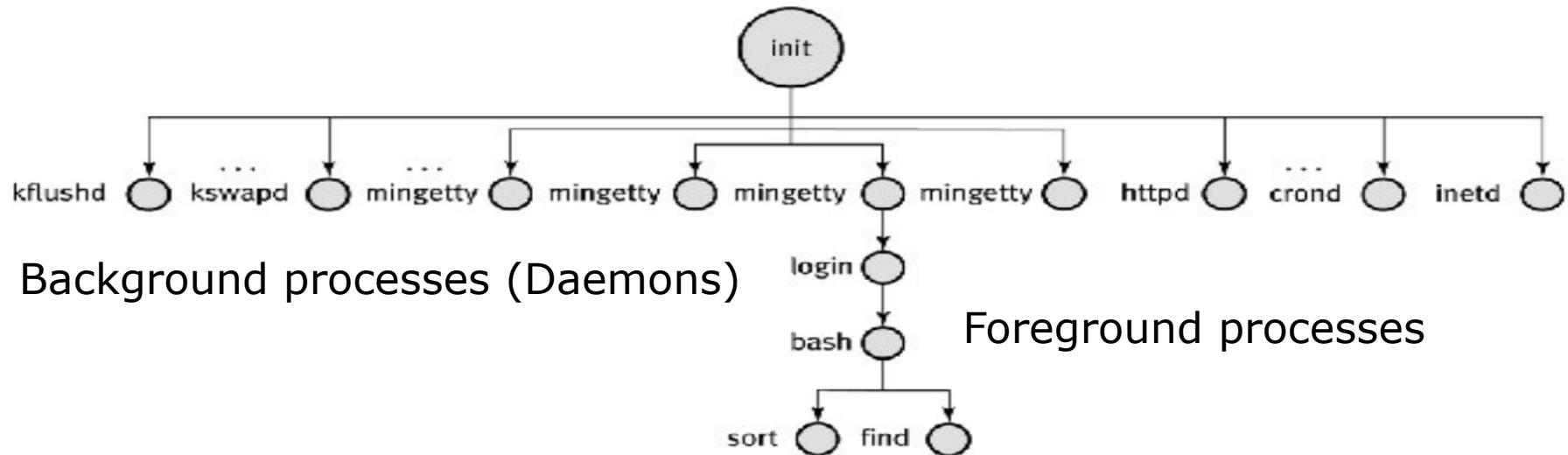
Creation:

1. System initialization
2. Process creation system call
3. User request to create a new process (shell)
4. Initiation of a batch job

Termination:

1. Normal exit (voluntary)
2. Error exit (voluntary)
3. Fatal error (involuntary)
4. Killed by another process (involuntary)

Linux process hierarchy



- All processes belong to their creator
 - "process group": receives all signals from creator
- Running a program starts a new process
- Windows has no concept of process hierarchy
 - Process is independent of its creator

Implementation of Processes

Process management	Memory management	File management
Registers Program counter Program status word Stack pointer Process state Priority Scheduling parameters Process ID Parent process Process group Signals Time when process started CPU time used Children's CPU time Time of next alarm	Pointer to text segment Pointer to data segment Pointer to stack segment	Root directory Working directory File descriptors User ID Group ID

- Process table holds context of each process
 - Process control blocks (PCBs)

Create process (Linux) – fork()

```
5 int main(int argc, char *argv[]) {  
6     printf("hello world (pid:%d)\n", (int) getpid());  
7     int rc = fork();  
8     if (rc < 0) {  
9         // fork failed  
10        fprintf(stderr, "fork failed\n");  
11        exit(1);  
12    } else if (rc == 0) {  
13        // child (new process)  
14        printf("hello, I am child (pid:%d)\n", (int) getpid());  
15    } else {  
16        // parent goes down this path (main)  
17        printf("hello, I am parent of %d (pid:%d)\n",  
18                rc, (int) getpid());  
19    }  
20    return 0;  
21 }
```

Get my process ID

Child process is a *clone* of its parent

hello world (pid:29146)

hello, I am parent of 29147 (pid:29146)

hello, I am child (pid:29147)

(Order may vary!)

Synchronising: wait()

```
6 int main(int argc, char *argv[]) {
7     printf("hello world (pid:%d)\n", (int) getpid());
8     int rc = fork();
9     if (rc < 0) {           // fork failed; exit
10        fprintf(stderr, "fork failed\n");
11        exit(1);
12    } else if (rc == 0) {   // child (new process)
13        printf("hello, I am child (pid:%d)\n", (int) getpid());
14    } else {                // parent goes down this path (main)
15        int rc_wait = wait(NULL);
16        printf("hello, I am parent of %d (rc_wait:%d) (pid:%d)\n",
17               rc, rc_wait, (int) getpid());
18    }
19    return 0;
20 }
```

hello world (pid:29266)

hello, I am child (pid:29267)

hello, I am parent of 29267 (rc_wait:29267) (pid:29266)

(Order always the same)

Changing what to run: exec*()

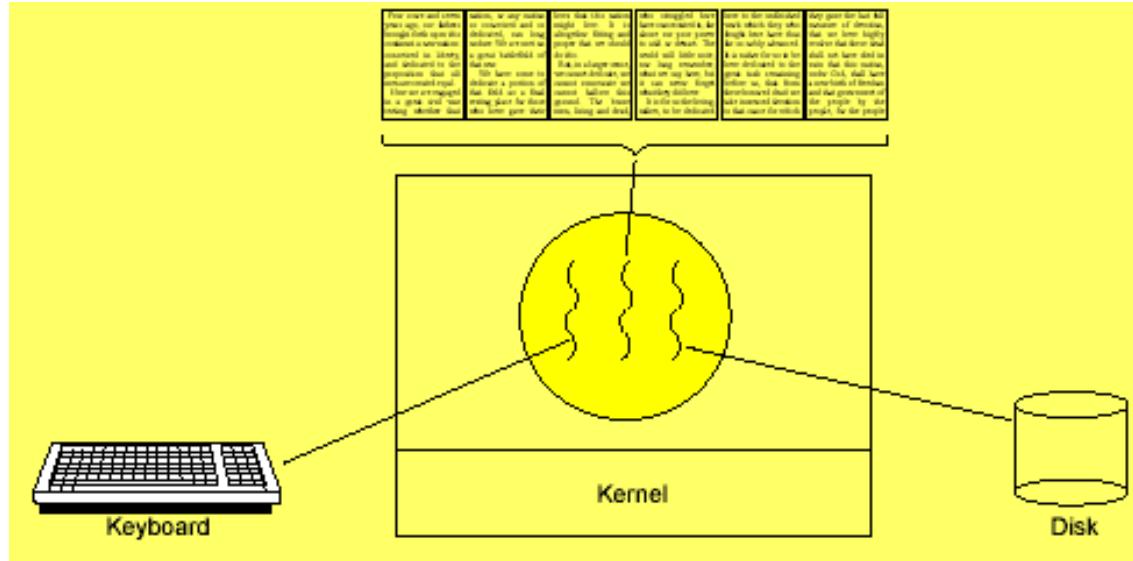
```
8 int main(int argc, char *argv[]) {
9     int rc = fork();
10    if (rc < 0) {
11        // fork failed
12        fprintf(stderr, "fork failed\n");
13        exit(1);
14    } else if (rc == 0) {
15        // child: redirect standard output to a file
16        close(STDOUT_FILENO);
17        open("./p4.output", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);
18
19        // now exec "wc"...
20        char *myargs[3];
21        myargs[0] = strdup("wc");      // program: wc (word count)
22        myargs[1] = strdup("p4.c");   // arg: file to count
23        myargs[2] = NULL;            // mark end of array
24        execvp(myargs[0], myargs);  // runs word count
25    } else {
26        // parent goes down this path (main)
27        int rc_wait = wait(NULL);
28    }
29    return 0;
30 }
```

Equiv. to `wc > ./p4.output`

Process summary

- A process is an independent resource group running a single program
- Linux: all processes are created and owned by a parent (unlike Windows)
- Forking a new process creates a clone of the parent, *including the same program counter*
- Exec*(file...) replaces the current program context with the new program file contents
 - Operations such as redirecting/piping output can be run before the program loads
- The code in a process runs sequentially
 - Or does it???

Why concurrent applications?

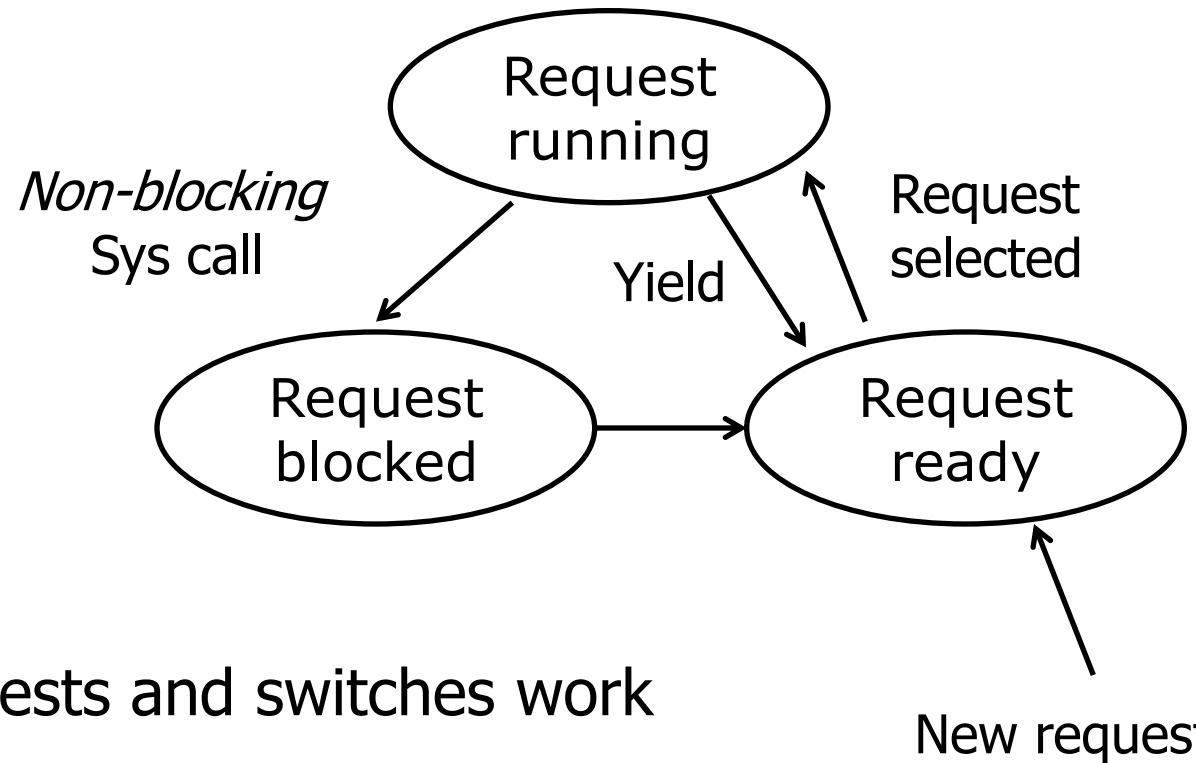


- Allows parallelism of independent operations in a single program ***that share common data***
- Example: word processor:
 - Task 1: respond to user input (update model)
 - Task 2: reformat document when model changes
 - Task 3: save periodically to disk
 - Task 4: spell check...

DIY concurrency: finite state machine

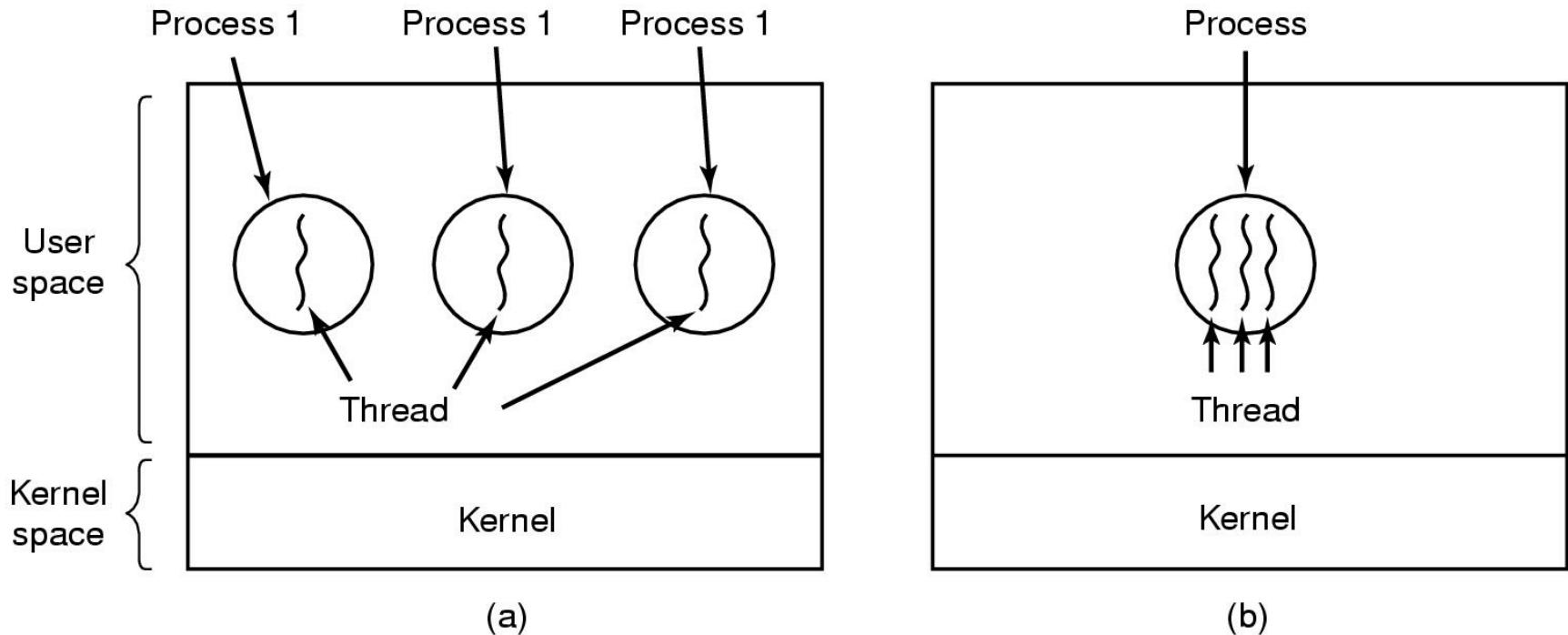
R1 Sys call return	
Request 2	✓
Request 1	✗

Job table



- Keeps track of requests and switches work when they “block”
- Cycles through jobs
 - Updating job statuses
 - Performing work
- Requires *non-blocking system calls* to be available

Threads



- Multiple threads of execution in a single process (“lightweight processes”)
- Can be implemented in the kernel or user libraries (or both)

Thread data model

Thread data space	Thread data space
Program counter Registers Stack State	Program counter Registers Stack State Unit of execution
Process data space Address space Global variables Open files Child processes Pending alarms Signals and signal handlers Accounting information	

- Threads all have access to process data space
 - But not to each others

Threading example

```
7 void *mythread(void *arg) {  
8     printf("%s\n", (char *) arg);  
9     return NULL;  
10 }  
11  
12 int  
13 main(int argc, char *argv[]) {  
14     pthread_t p1, p2;  
15     int rc;  
16     printf("main: begin\n");  
17     Pthread_create(&p1, NULL, mythread, "A");  
18     Pthread_create(&p2, NULL, mythread, "B");  
19     // join waits for the threads to finish  
20     Pthread_join(p1, NULL);  
21     Pthread_join(p2, NULL);  
22     printf("main: end\n");  
23     return 0;  
24 }
```

The thread is **not** a clone,
It runs a callback function

Wait for completion...
Order may vary

Threads versus processes

```
main()
{
    pid_t childPid = fork(); // Process
    if (childPid == 0)
        printf("I am the child process\n");
    else
        printf("I am the parent process\n");
}
```

```
main()
{
    pthread_t childId;
    pthread_create (&childId, NULL, ChildCode, NULL); // Thread
    printf("I am the parent thread\n");
}

void* ChildCode (void* arg) { printf("I am the child
thread\n"); }
```

POSIX threads API (e.g. Linux)

- Standard runtime library calls for managing threads:
 - ***pthread_create***
 - creates a thread to execute a specified function.
 - ***pthread_exit***
 - causes the calling thread to terminate without the whole process terminating.
 - ***pthread_kill***
 - sends a “kill” signal to a specified thread.
 - ***pthread_join***
 - causes the calling thread to wait for the specified thread to exit. Similar to ***waitpid*** for processes.
 - ***pthread_self***
 - returns the callers identity (The thread ID).
 - ***Pthread_yield***
 - Yields the CPU to another thread
- See man pages
 - eg *man pthread_create*
 - Many other calls

pthread_create

```
#include <pthread.h>
int
pthread_create(pthread_t      *thread,
              const pthread_attr_t *attr,
              void                * (*start_routine) (void*),
              void                *arg);
```

Creates a new thread:

- *thread: (pointer to) thread variable
- *attr: control attributes (usually NULL)
- *start_routine: function to run
- *arg: argument(s) to be passed into the function

pthread_join

```
int pthread_join(pthread_t thread, void **value_ptr);
```

Waits for a child thread to finish:

- *thread: (pointer to) thread variable
- **value_ptr: *handle* to the value returned by the thread
 - Generated by pthread_join() from pthread_exit() *after the thread has exited*
 - **Need to be very careful how you return values from a thread**

Returning data from a thread

```
void *mythread(void *arg) {  
    myret_t *rvals = Malloc(sizeof(myrret_t));  
    rvvals->x = 1;  
    rvvals->y = 2;  
    return (void *) rvvals;  
}
```

Good!

```
int main(int argc, char *argv[]) {  
    pthread_t p;  
    myret_t *rvvals;  
    myarg_t args = { 10, 20 };  
    Pthread_create(&p, NULL, mythread, &args);  
    Pthread_join(p, (void **) &rvvals);  
    printf("returned %d %d\n", rvvals->x, rvvals->y);
```

```
void *mythread(void *arg) {  
    myarg_t *args = (myarg_t *) arg;  
    printf("%d %d\n", args->a, args->b);  
    myret_t oops; // ALLOCATED ON STACK: BAD!  
    oops.x = 1;  
    oops.y = 2;  
    return (void *) &oops;  
}
```

Bad!

Threads versus processes

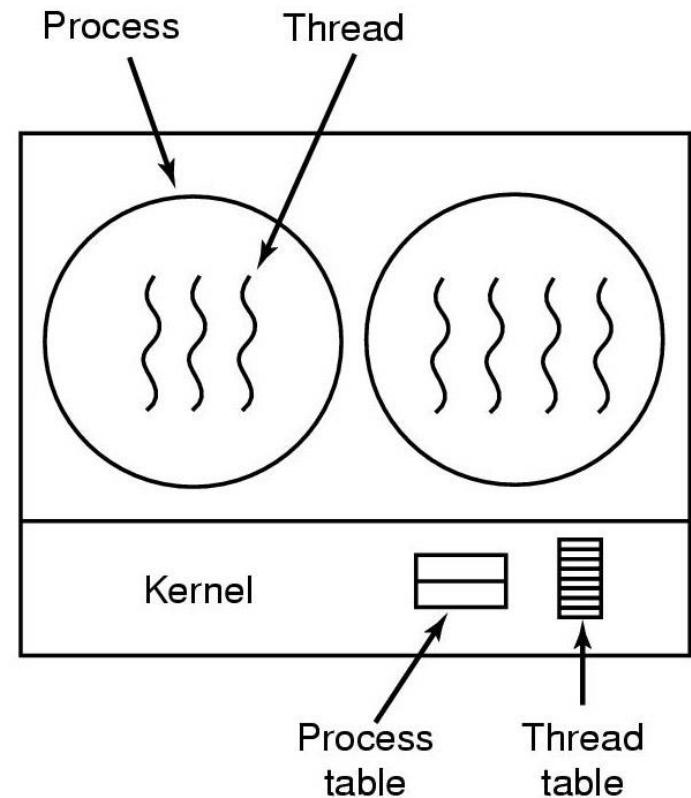
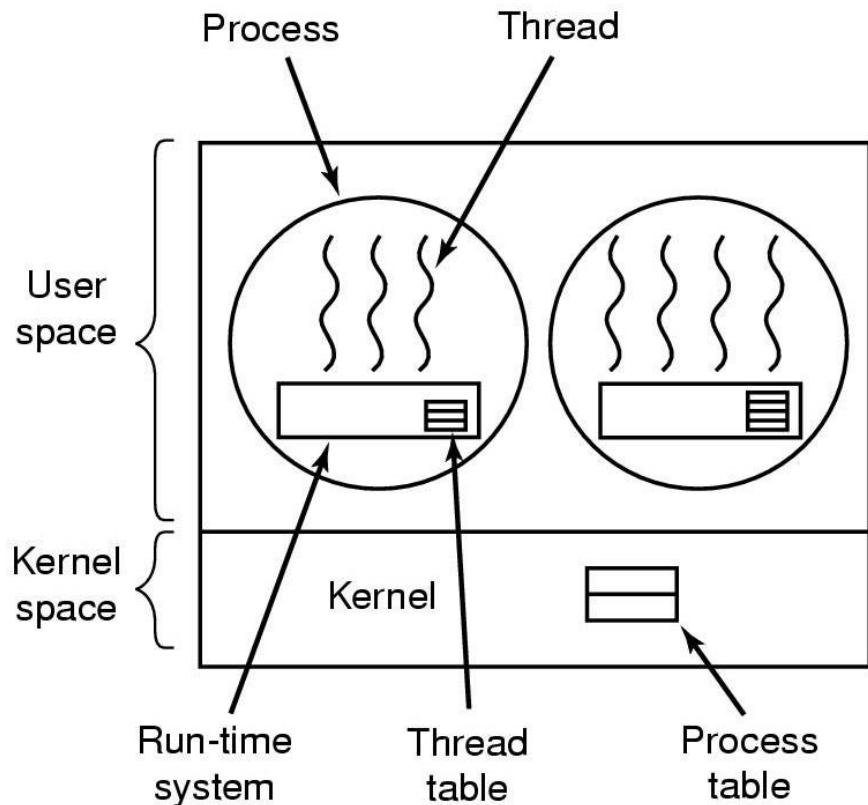
Threads: the good

- Access to the same data makes communication easy
- Very fast to create, *can* be fast to switch
- **Possible** performance gains from switching within a single process
- Can be spread across CPUs for further parallelisation

Threads: the not-so-good

- Difficult to write and debug the code
 - Ordering issues
 - Data access issues (more coming soon!)

Implementation options

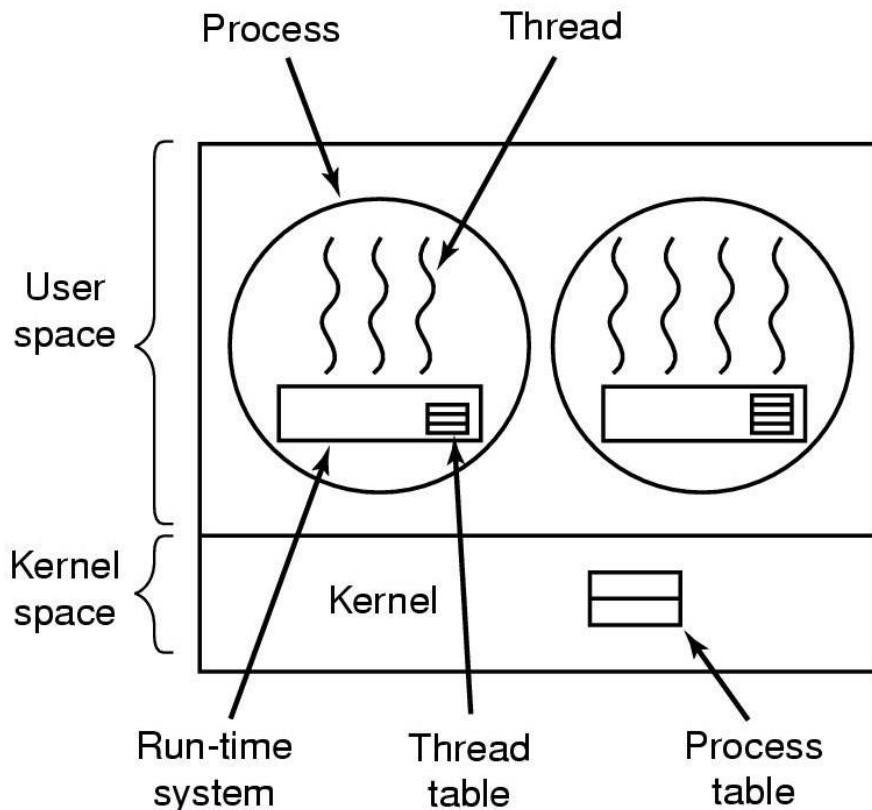


User-level

vs

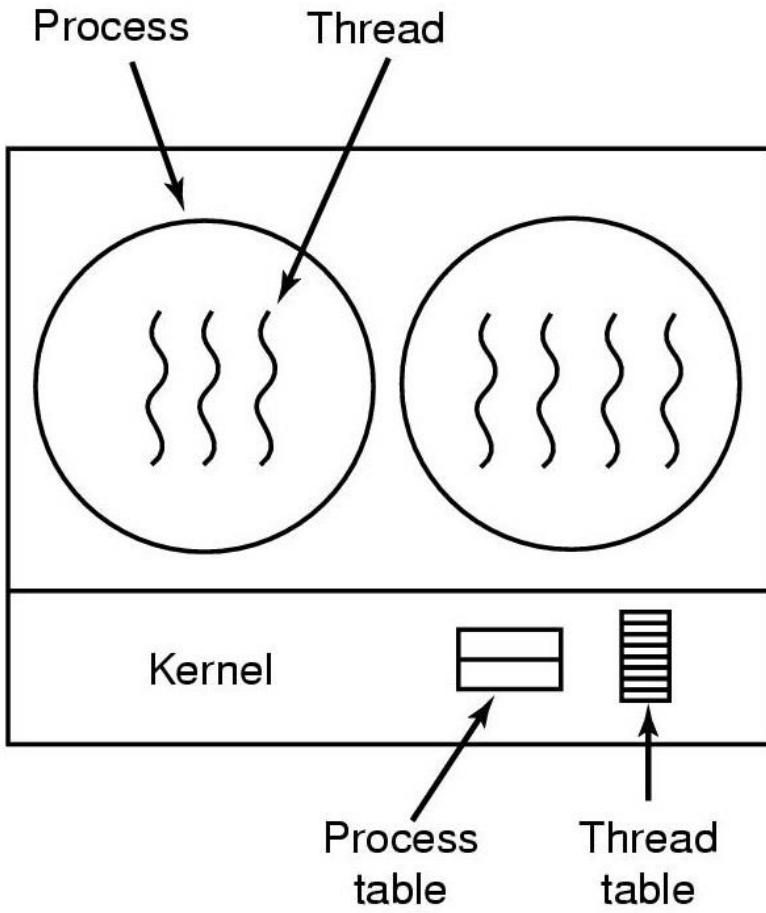
kernel

Implementing threads in user space



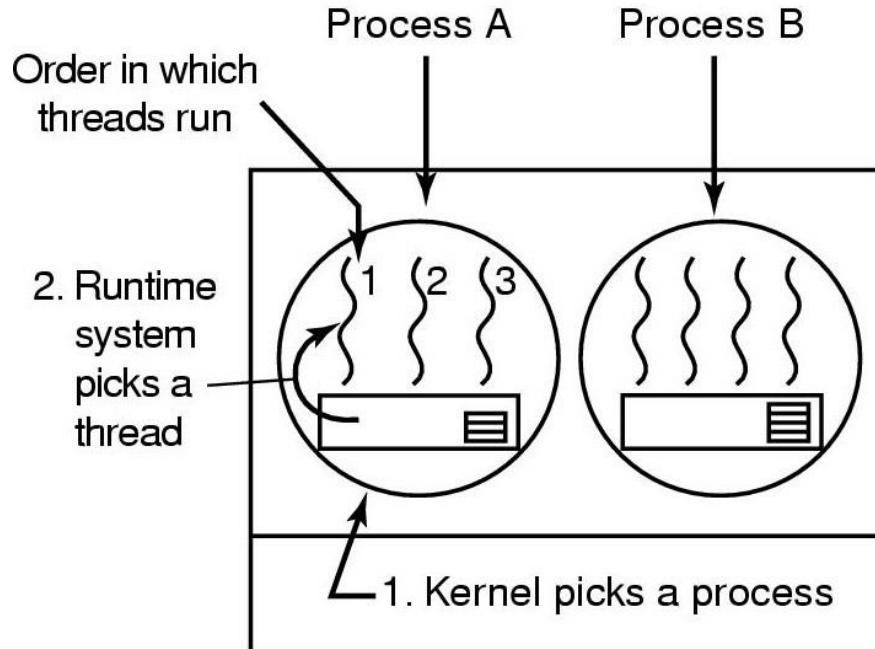
- Thread library run in user space runtime
 - Thread table
 - Scheduler
- Kernel not involved
 - Fast switching
 - Requires *non-blocking* system calls
- OS-independent

Implementing threads in the Kernel



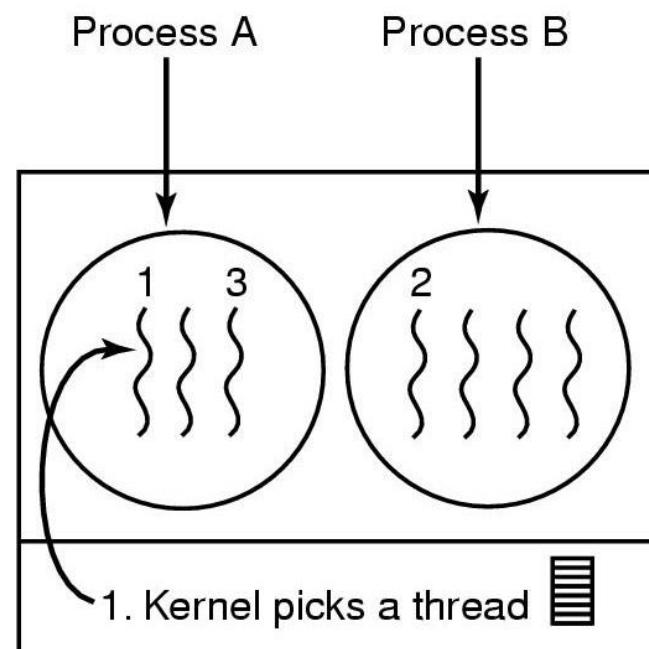
- Thread table lives in the kernel
- Scheduler schedules threads, not processes
- Usually slower to create and switch
- More control over what runs
 - Another thread in the same process, or:
 - A thread in a different process

Thread scheduling: user vs kernel



Possible: A1, A2, A3, A1, A2, A3

Not possible: A1, B1, A2, B2, A3, B3



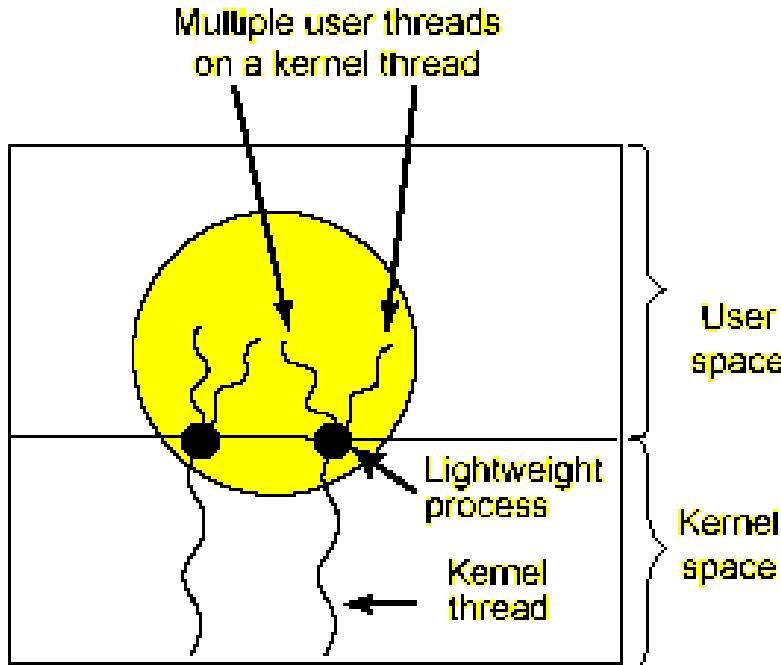
Possible: A1, A2, A3, A1, A2, A3

Also possible: A1, B1, A2, B2, A3, B3

Possible scheduling of threads over 30ms where:

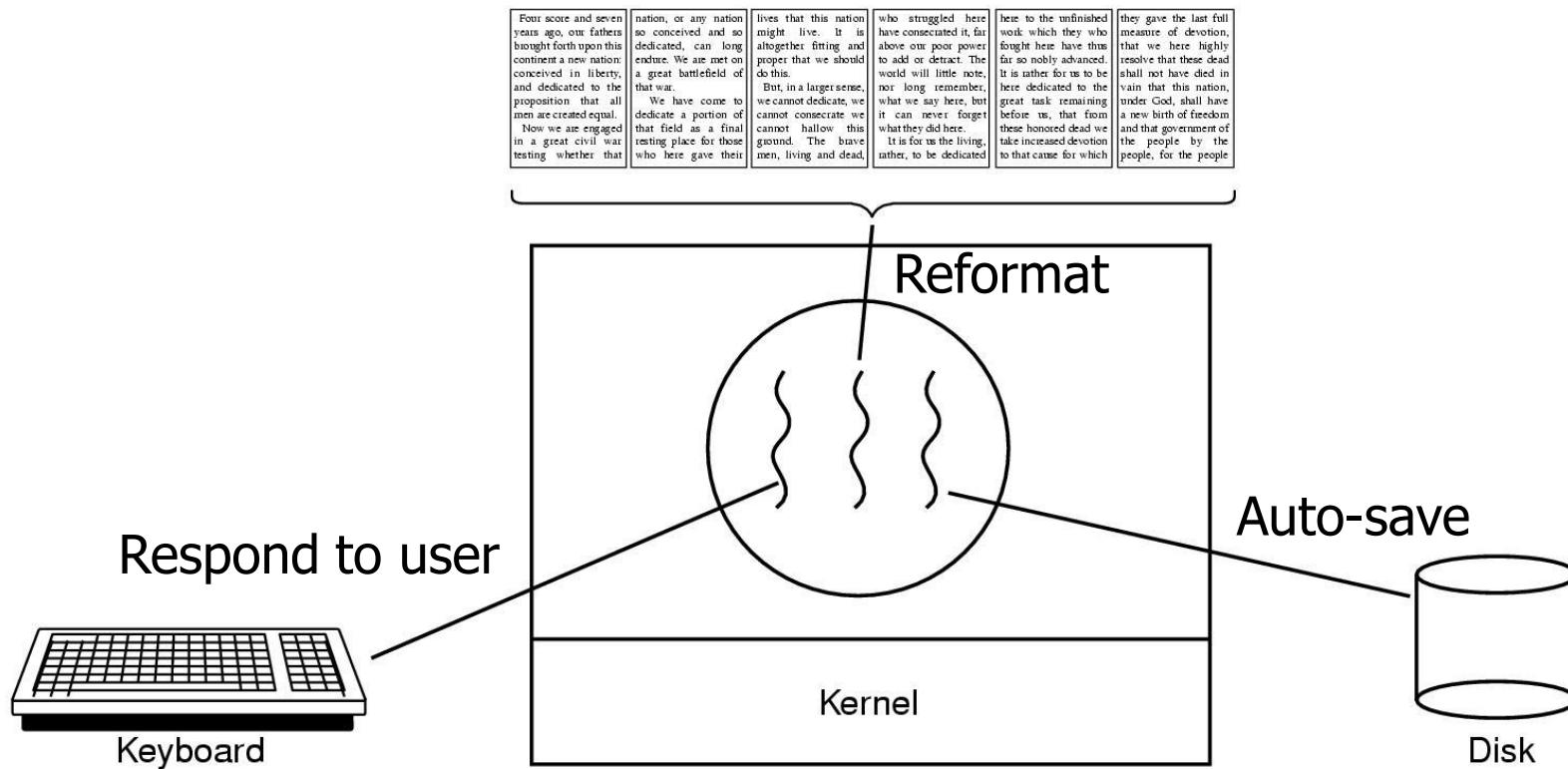
- 50ms process/thread quantum
- Each thread runs for 5ms and then blocks

Hybrid thread implementation



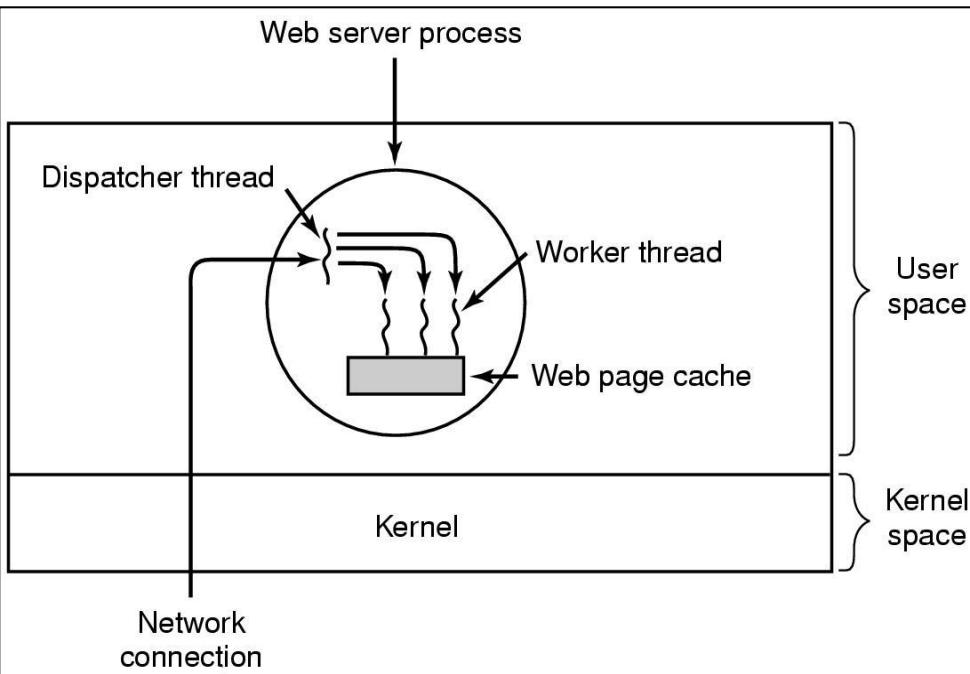
- User specifies how many kernel threads to use
- User-level threads are mapped to the kernel-schedulable ones.
- Combines advantages of both approaches

Patterns (1): specialised threads



- Each thread performs an independent job
- Different work patterns (user input vs batch)

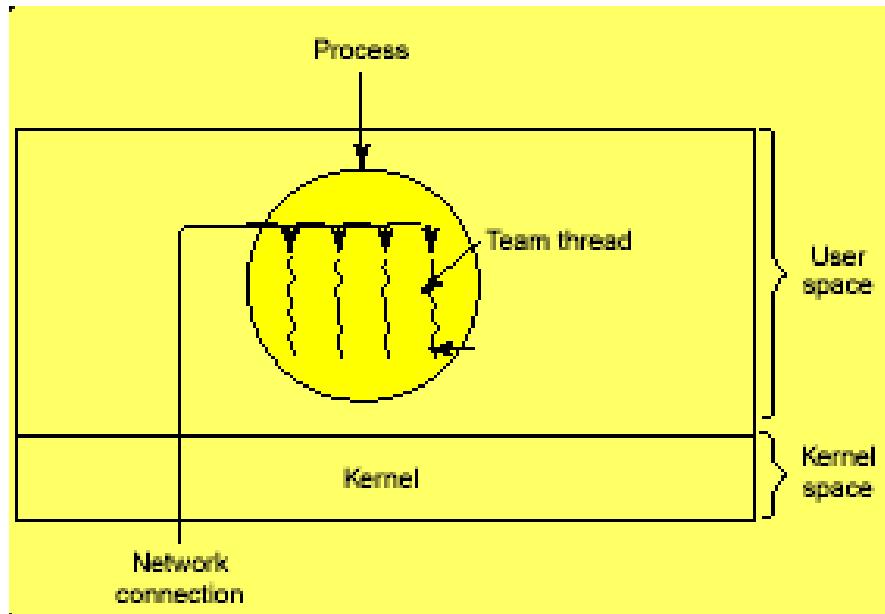
Patterns (2): dispatcher/worker



```
while (TRUE) {  
    getNextRequest(&buf );  
    handoffWork(&buf );  
}  
  
while (TRUE) {  
    waitForWork(&buf );  
    process(&buf );  
}
```

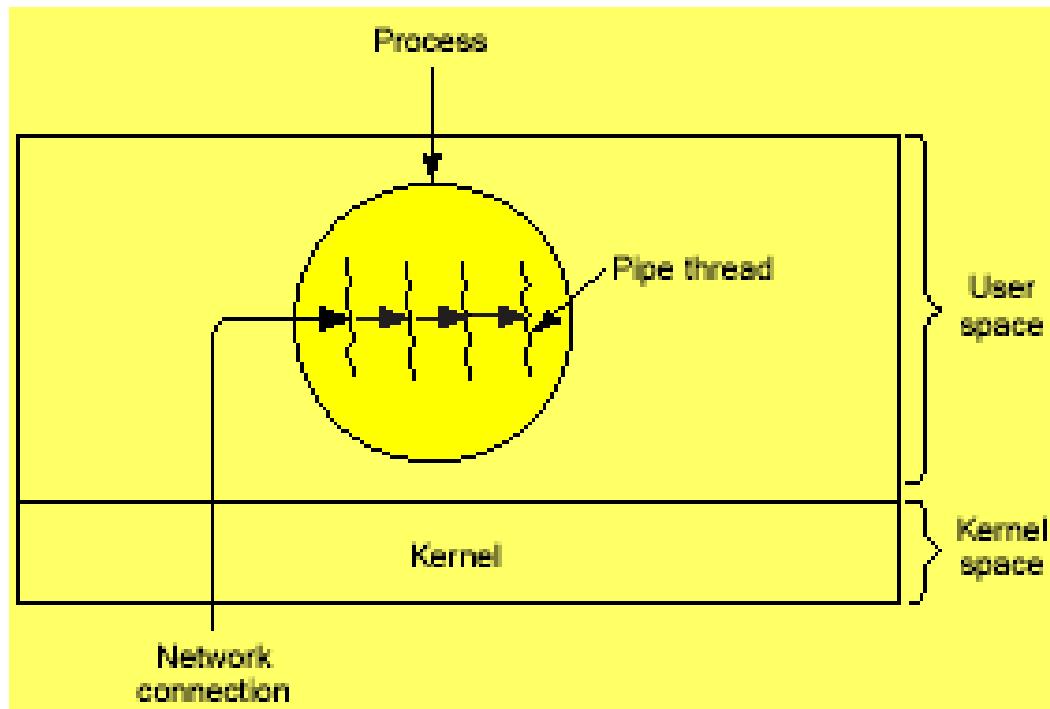
- Dispatcher thread handles incoming requests and farms out to pool of workers
- Workers get woken to perform the work
- Example: web server

Patterns (3): team/pool



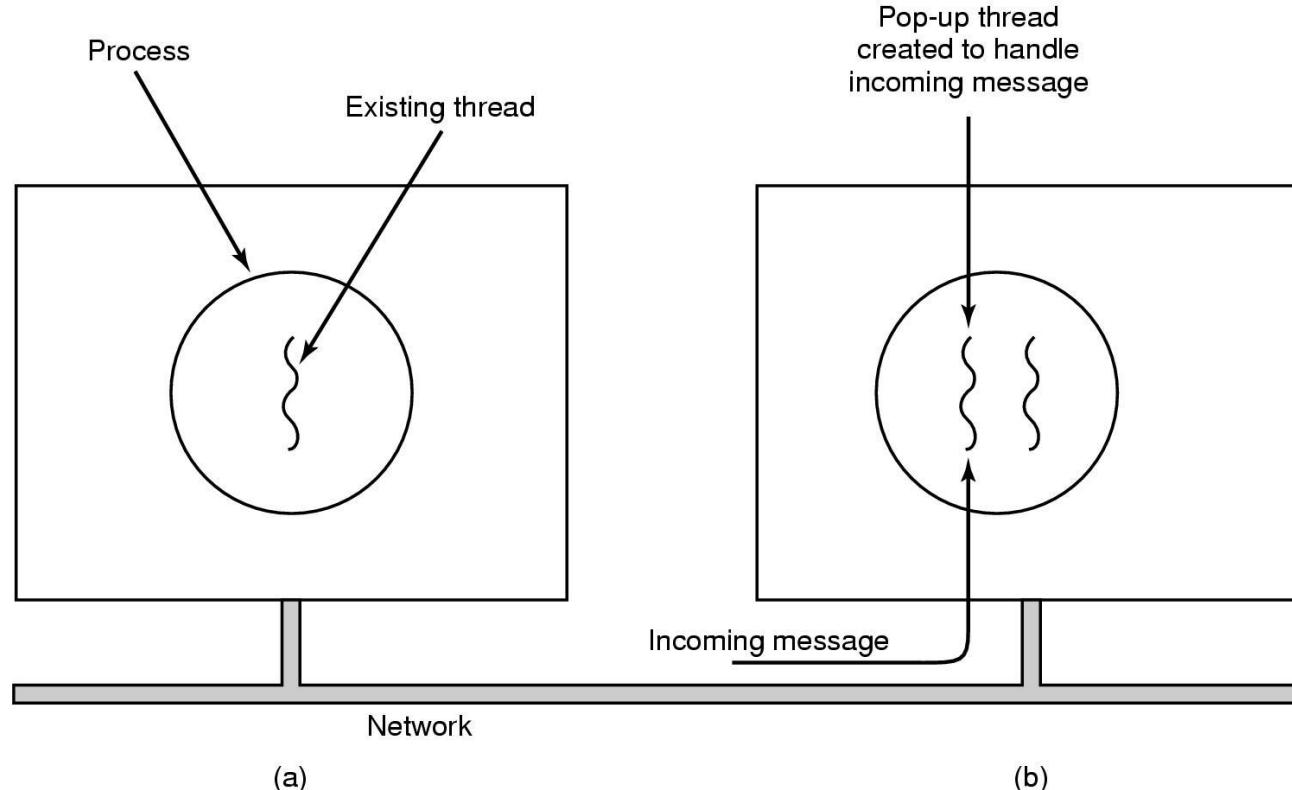
- All team members are equal:
 - Wait for incoming requests and grab one to process
- Managed pool: threads in pool created/destroyed by library
 - e.g. .NET System.Threading.ThreadPool
- Can be specialised too
 - Place inappropriate requests onto an internal queue

Patterns (4): pipeline



- Chain of *consumer/producers*:
 - Each thread consumes a request and produces a new one for the next thread
 - Specialised threads designed to minimise latency

Patterns (5): pop-up threads



- Thread created when needed to handle new request
 - Starts “fresh”: faster than context switching, easier to code

Lab (1)

```
int global = 5;

void* child(void* arg) {
    int local = 10;

    global++;
    local++;
    printf("[child] child id: 0x%x\n", (unsigned int) _self());
    printf("[child] global: %d local: %d\n", global, local);

    _exit(NULL);
}

int main() {
    _t childPid;
    int      local = 10;

    printf("[At start] global: %d local: %d\n", global, local);

    if (_create (&childPid, NULL, child, NULL) != 0)
    {
        perror("create");
        exit(1);
    } else {
        global++;
        local--;
        printf("[Parent] parent id : 0x%x\n", (unsigned int) _self());
        printf("[Parent] global: %d local: %d\n", global, local);
        sleep(1);
    }
    printf("[At end] global: %d local: %d\n", global, local);
    exit(0);
}
```

Lab (2)

```
int global = 5;

int main() {
    pid_t childPid;
    int local = 10;

    printf("[Start with] global: %d local: %d\n", global, local);

    childPid = fork();

    if(childPid < 0) {
        perror("fork");
        exit(1);
    }
    else if (childPid == 0) {
        global++;
        local++;
        printf("[Child] childPid: 0x%x\n", getpid());
        printf("[Child] global: %d local: %d\n", global, local);
        sleep(4); /* Wait 4 seconds */
    }
    else {
        global--;
        local--;
        /* waitpid(childPid, &status, 0); */
        printf("[Parent] childPid: 0x%x parent: 0x%x\n", childPid, getpid());
        printf("[Parent] global: %d local: %d\n", global, local);
        sleep(2); /* Wait 2 seconds */
    }

    printf("[At end (0x%x)] global: %d local: %d\n", getpid(), global, local);

    return EXIT_SUCCESS;
}
```

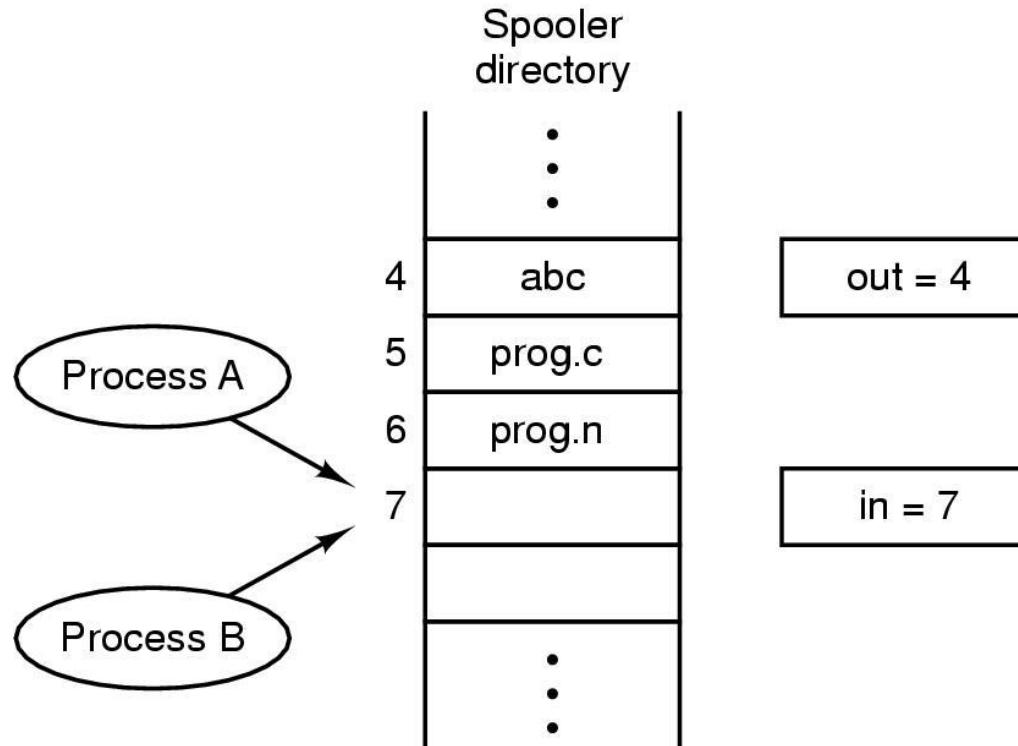


Inter-process communication

- 2.1 Processes
- 2.2 Threads
- 2.3 Inter-process communication**
- 2.4 Classical IPC problems**
- 2.5 Scheduling

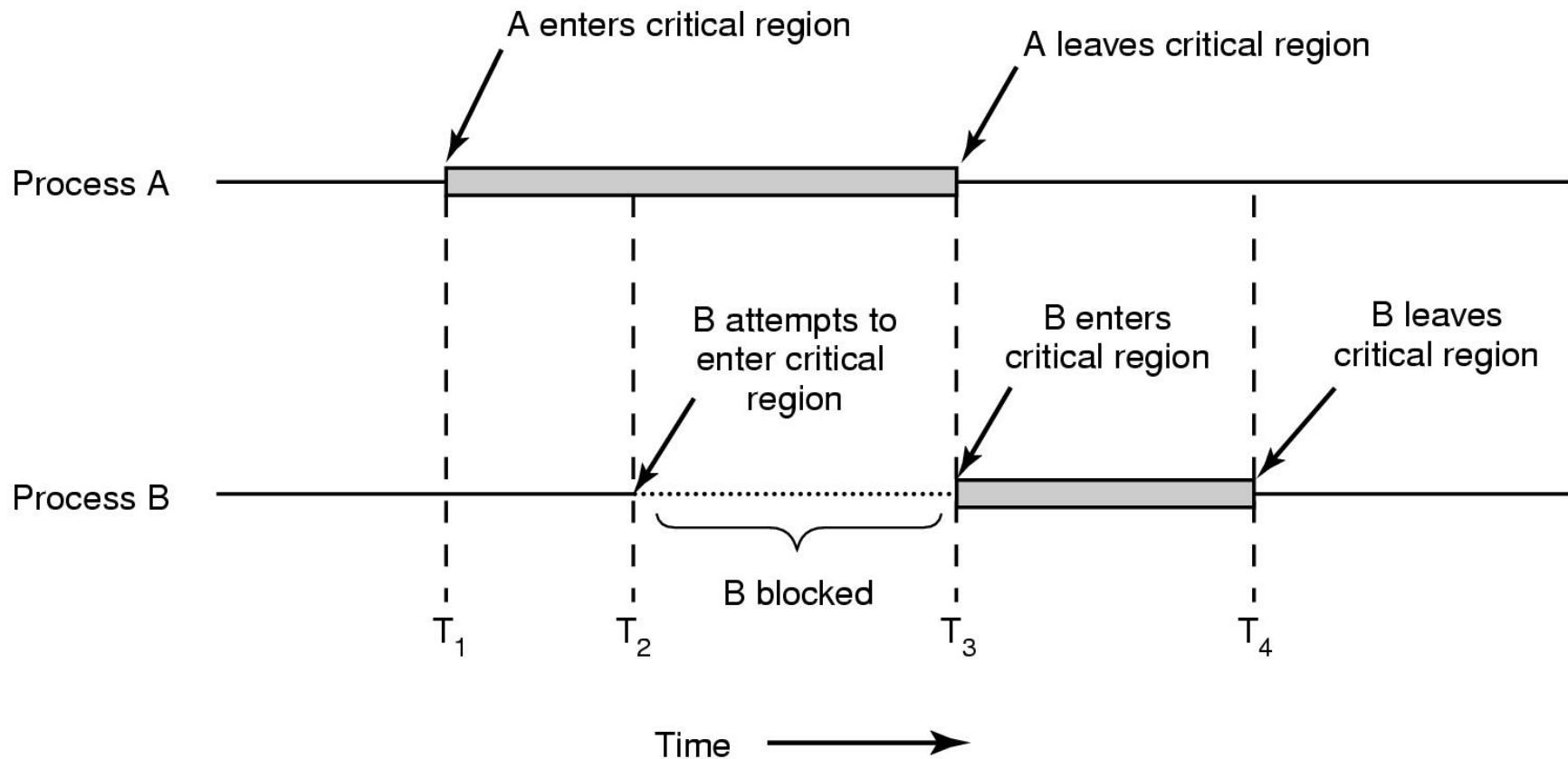
Chapter 2
MODERN OPERATING SYSTEMS (MOS)
*By Andrew Tanenbaum
(+ code from OSTEP)*

Race conditions



- Two or more processes try to access the same resource: outcome is order-dependent
- Example: printer spooler – multiple processes adding jobs
 - What happens when they get interrupted?

Critical Regions



- Mechanism needed to provide “mutual exclusion” of critical regions

Mutual exclusion

Four conditions to provide workable mutual exclusion:

1. No two processes simultaneously in critical region
2. No assumptions made about speeds or numbers of CPUs
3. No process running outside its critical region may block another process
4. No process must wait forever to enter its critical region

Non-solutions (1)

These solutions violate the four conditions:

1. Blocking process disables interrupts on CPU
2. Lock variable, updated just before entering the critical region
3. *Strict alternation*
4. *Peterson's solution*

Non-solutions(2): strict alternation

```
while (TRUE) {  
    while (turn != 0)      /* loop */ ;  
    critical_region();  
    turn = 1;  
    noncritical_region();  
}
```

Process 0

```
while (TRUE) {  
    while (turn != 1)      /* loop */ ;  
    critical_region();  
    turn = 0;  
    noncritical_region();  
}
```

Process 1

- Loops on block: (busy waiting) – “spin lock”
- What happens if one process is much slower than the other (or gets stuck)?
 - Which condition is being violated?

Nearly there (1): Peterson's solution

```
bool flag[2] = {false, false};  
int turn;
```

```
P0:      flag[0] = true;  
P0_gate: turn = 1;  
        while (flag[1] == true &&  
turn == 1)  
        {  
            // busy wait  
        }  
        // critical section  
        ...  
        // end of critical section  
        flag[0] = false;
```

```
P1:      flag[1] = true;  
P1_gate: turn = 0;  
        while (flag[0] == true &&  
turn == 0)  
        {  
            // busy wait  
        }  
        // critical section  
        ...  
        // end of critical section  
        flag[1] = false;
```

- Each process hands off the “turn” to the other process
- Blocking occurs in the critical region only
- Still “busy waiting” – prone to CPU “priority inversion problem”
- Some systems reorder memory access
 - Similar solutions exist in hardware

Nearly there (2): block on wait

```
#define N 100
int count = 0;

/* number of slots in the buffer */
/* number of items in the buffer */

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();
        if (count == N) sleep();
        insert_item(item);
        count = count + 1;
        if (count == 1) wakeup(consumer);
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {
        if (count == 0) sleep();
        item = remove_item();
        count = count - 1;
        if (count == N - 1) wakeup(producer);
        consume_item(item);
    }
}
```

/* repeat forever */
/* generate next item */
/* if buffer is full, go to sleep */
/* put item in buffer */
/* increment count of items in buffer */
/* was buffer empty? */

/* repeat forever */
/* if buffer is empty, got to sleep */
/* take item out of buffer */
/* decrement count of items in buffer */
/* was buffer full? */
/* print item */

- What happens when “wakeup” is sent to a process that isn’t asleep? How to fix?

Semaphores

- **Atomic** operations to raise or lower their value
 - Blocks on decrement if value is 0
- Used to count wakeup requests
 - solves the “lost wakeup” problem
- Also used as locks (mutex)
- POSIX implementation in C:

Operation	Pseudocode	POSIX
Increment/wakeup	<code>up()</code>	<code>sem_post()</code>
Decrement/sleep if 0	<code>down()</code>	<code>sem_wait()</code>

Producer-consumer with semaphores

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}

/* number of slots in the buffer */
/* semaphores are a special kind of int */
/* controls access to critical region */
/* counts empty buffer slots */
/* counts full buffer slots */

/* TRUE is the constant 1 */
/* generate something to put in buffer */
/* decrement empty count */
/* enter critical region */
/* put new item in buffer */
/* leave critical region */
/* increment count of full slots */

/* infinite loop */
/* decrement full count */
/* enter critical region */
/* take item from buffer */
/* leave critical region */
/* increment count of empty slots */
/* do something with the item */
```

Mutual exclusion (mutex) and order synchronisation (empty, full)

POSIX thread implementation (1)

- **Mutex** atomically locks/unlocks
 - Used for access to critical region

Operation	POSIX threads
Create mutex	<code>pthread_mutex_init()</code>
Destroy mutex	<code>pthread_mutex_destroy()</code>
Acquire lock or block thread	<code>pthread_mutex_lock()</code>
Acquire lock or fail	<code>pthread_mutex_trylock()</code>
Release lock	<code>pthread_mutex_unlock()</code>

POSIX thread implementation (2)

- **Condition variable** blocks until a condition is met (“signaled”)
 - Used to wake waiting threads
- Requires a mutex
 - Atomically yielded on block, acquired on wake up

Operation	POSIX threads
Create condition variable	<code>pthread_cond_init()</code>
Destroy condition variable	<code>pthread_cond_destroy()</code>
Block waiting for signal	<code>pthread_cond_wait()</code>
Signal and wake one thread	<code>pthread_cond_signal()</code>
Signal and wake all threads	<code>pthread_cond_broadcast()</code>

Producer-consumer with threads

```
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp;
int buffer = 0;

void *producer(void *ptr)                                /* how many numbers to produce */
{
    int i;

    for (i= 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
        buffer = i;                      /* put item in buffer */
        pthread_cond_signal(&condc);      /* wake up consumer */
        pthread_mutex_unlock(&the_mutex);/* release access to buffer */
    }
    pthread_exit(0);
}

void *consumer(void *ptr)                                 /* consume data */
{
    int i;

    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer == 0 ) pthread_cond_wait(&condc, &the_mutex);
        buffer = 0;                      /* take item out of buffer */
        pthread_cond_signal(&condp);      /* wake up producer */
        pthread_mutex_unlock(&the_mutex);/* release access to buffer */
    }
    pthread_exit(0);
}
```

Access inverted
compared to
semaphore soln

There be dragons...

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```

/* number of slots in the buffer */
/* semaphores are a special kind of int */
/* controls access to critical region */
/* counts empty buffer slots */
/* counts full buffer slots */

/* TRUE is the constant 1 */
/* generate something to put in buffer */
/* decrement empty count */
/* enter critical region */
/* put new item in buffer */
/* leave critical region */
/* increment count of full slots */

/* infinite loop */
/* decrement full count */
/* enter critical region */
/* take item from buffer */
/* leave critical region */
/* increment count of empty slots */
/* do something with the item */

- What happens if we reverse the “downs”?
- What happens if we reverse the “ups”?

Hiding the details: monitors

monitor *example*

integer *i*;
 condition *c*;

procedure *producer()*;

 .

 .

 .

end;

procedure *consumer()*;

 .

 .

 .

end;

end monitor;

- Only one process can enter at a time
- Language-level construct
- Example: Java “synchronised” methods:
 - Only one can run at a time in an object

Solving “lost wakeup” with monitors

monitor *ProducerConsumer*

condition *full, empty*;

integer *count*;

procedure *insert(item: integer)*;

begin

if *count = N* **then** *wait(full)*;

insert_item(item);

count := count + 1;

if *count = 1* **then** *signal(empty)*

end;

function *remove: integer*;

begin

if *count = 0* **then** *wait(empty)*;

remove = remove_item;

count := count - 1;

if *count = N - 1* **then** *signal(full)*

end;

count := 0;

end monitor;

procedure *producer*;

begin

while *true* **do**

begin

item = produce_item;

ProducerConsumer.insert(item)

end

end;

procedure *consumer*;

begin

while *true* **do**

begin

item = ProducerConsumer.remove;

consume_item(item)

end

end;

Distributed CPUs: Message Passing

```
#define N 100                                     /* number of slots in the buffer */

void producer(void)
{
    int item;
    message m;                                    /* message buffer */

    while (TRUE) {
        item = produce_item();                    /* generate something to put in buffer */
        receive(consumer, &m);                   /* wait for an empty to arrive */
        build_message(&m, item);                /* construct a message to send */
        send(consumer, &m);                     /* send item to consumer */
    }
}

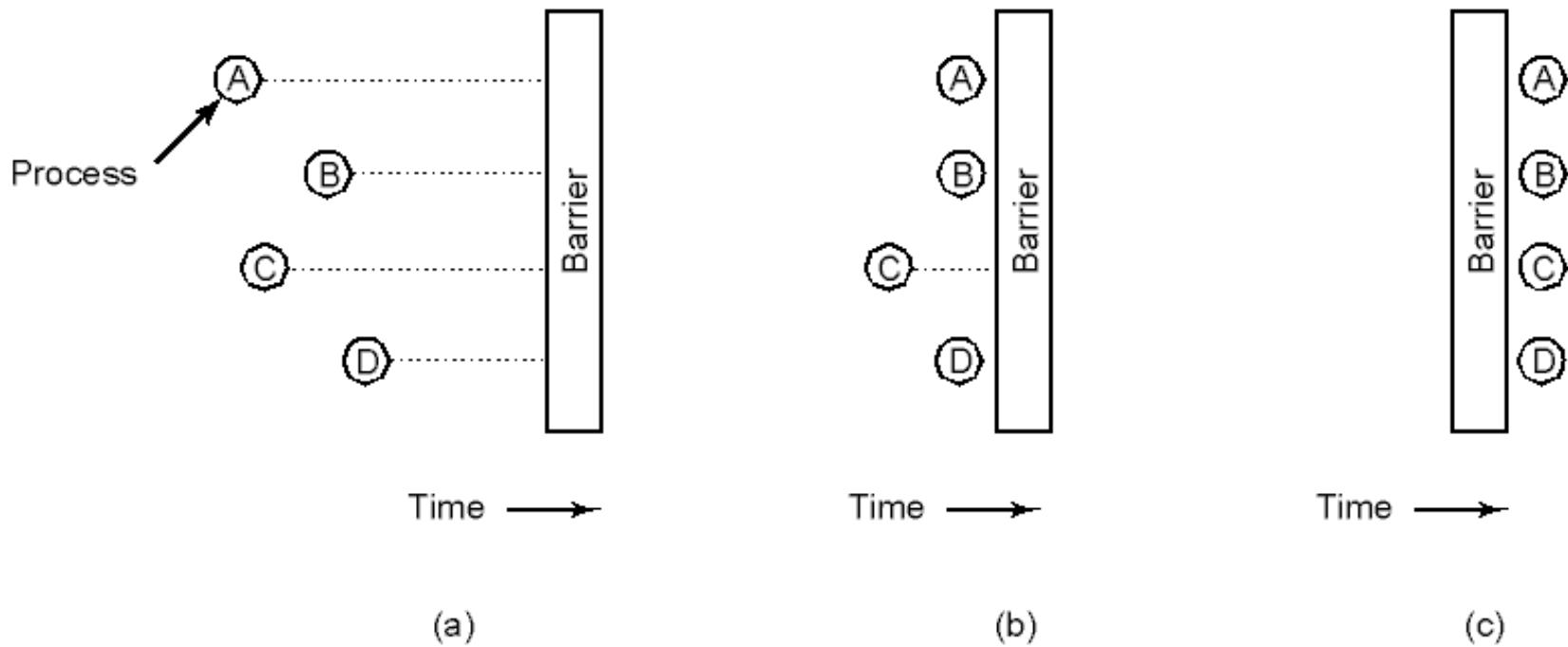
void consumer(void)
{
    int item, i;
    message m;                                    N empty messages act as a buffer

    for (i = 0; i < N; i++) send(producer, &m); /* send N empties */
    while (TRUE) {
        receive(producer, &m);                 /* get message containing item */
        item = extract_item(&m);              /* extract item from message */
        send(producer, &m);                  /* send back empty reply */
        consume_item(item);                 /* do something with the item */
    }
}
```

Receive blocks until a message is received

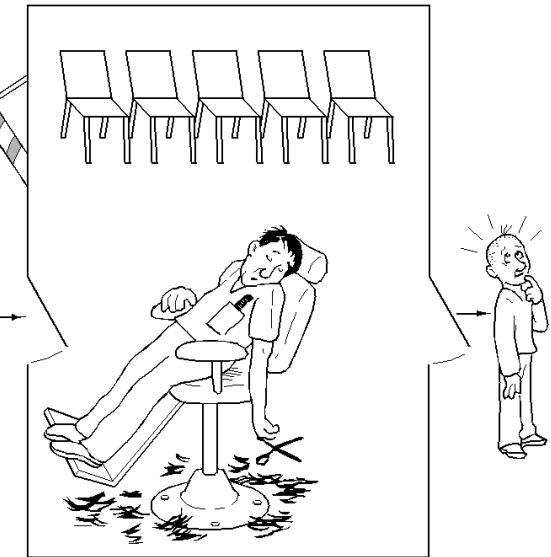
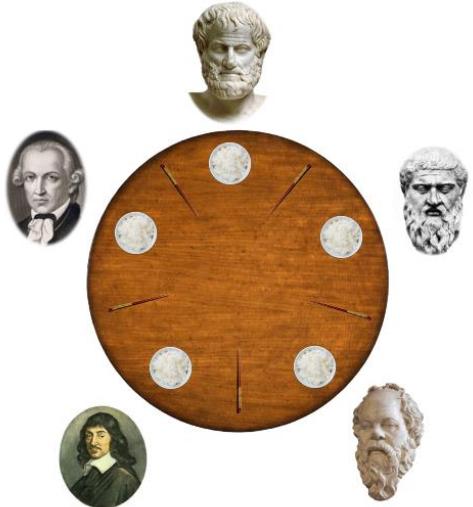
N empty messages act as a buffer

Multi-process synchronisation: barriers



- Synchronises distributed iterative algorithms
 - Blocks progress until all processes have completed
 - Example: large matrix operations (e.g. image analysis)

Some typical problems (looked at atypically)

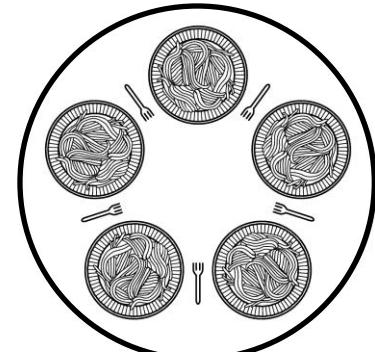


Dining Philosophers: deadlocks and starvation!

Five philosophers sitting at a circular table doing one of two things - eating or thinking. While eating, they are not thinking, and while thinking, they are not eating.

In order to eat, a philosopher needs both forks. A philosopher may only pick up one fork at a time. Each philosopher attempts to pick up the left fork first and then the right fork. When done eating, a philosopher puts both forks back down on the table and begins thinking. It is therefore not possible for all of them to be eating at the same time.

**What happens if they all try to eat at the same time?
What happens if they retry after n seconds?**



Demo: www.doc.ic.ac.uk/~jnm/concurrency/classes/Diners/Diners.html (requires applets)

Starving philosophers

```
#define N 5                                     /* number of philosophers */  
  
void philosopher(int i)                         /* i: philosopher number, from 0 to 4 */  
{  
    while (TRUE) {  
        think();                                 /* philosopher is thinking */  
        take_fork(i);                            /* take left fork */  
        take_fork((i+1) % N);                   /* take right fork; % is modulo operator */  
        eat();                                   /* yum-yum, spaghetti */  
        put_fork(i);                            /* put left fork back on the table */  
        put_fork((i+1) % N);                   /* put right fork back on the table */  
    }  
}
```

- This (non) solution suffers deadlock

Happy philosophers (with semaphores)

```
int state[N];                                /* array to keep track of everyone's state */
semaphore mutex = 1;                          /* mutual exclusion for critical regions */
semaphore s[N];                              /* one semaphore per philosopher */

void philosopher(int i)                      /* i: philosopher number, from 0 to N-1 */
{
    while (TRUE) {
        think();
        take_forks(i);
        eat();
        put_forks(i);
    }
}

void take_forks(int i)
{
    down(&mutex);
    state[i] = HUNGRY;
    test(i);
    up(&mutex);
    down(&s[i]);
}

void put_forks(i)
{
    down(&mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    up(&mutex);
}

void test(i)
{
    if (state[i] == HUNGRY &&
        state[LEFT] != EATING &&
        state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

Readers and writers: not all processes are created equal

Example: database access

- Multiple readers can access the db at the same time
- Writers require exclusive access
 - All readers need to exit before the writer can gain the lock
 - New readers may arrive while writer is waiting

```
typedef int semaphore;
semaphore mutex = 1;
semaphore db = 1;
int rc = 0;

/* use your imagination */
/* controls access to 'rc' */
/* controls access to the database */
/* # of processes reading or wanting to */

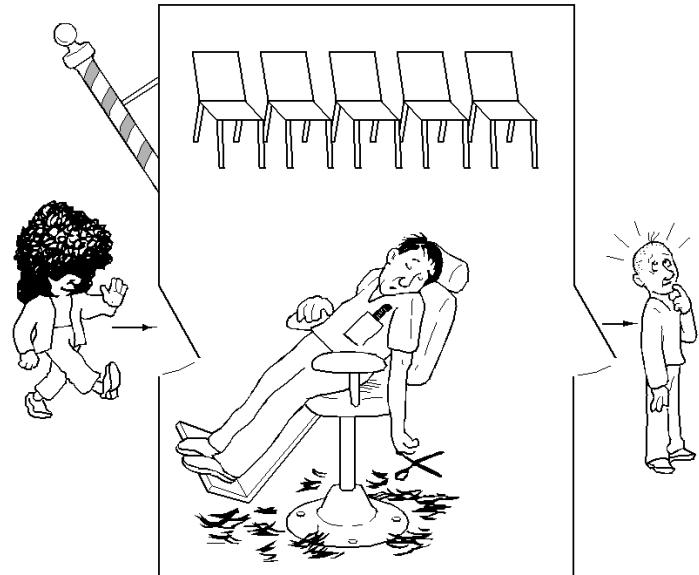
void writer(void)
{
    void reader(void)

    while (TRUE) {
        down(&mutex);
        rc = rc + 1;
        if (rc == 1) down(&db);
        up(&mutex);
        read_data_base();
        down(&mutex);
        rc = rc - 1;
        if (rc == 0) up(&db);
        up(&mutex);
        use_data_read();
    }
}

Is this a "fair" solution?
```

The Sleeping Barber Problem

A barber shop with barbers, barber chairs, and a number of chairs for waiting customers. When there are no customers, barbers sit in their chairs and sleep.



As soon as a customer arrives, he either awakens a barber or, if all barbers are cutting hair, sits down in one of the vacant chairs. If all of the chairs are occupied, the newly arrived customer simply leaves.

Sleeping barber solution

```
#define CHAIRS 5                                     /* # chairs for waiting customers */

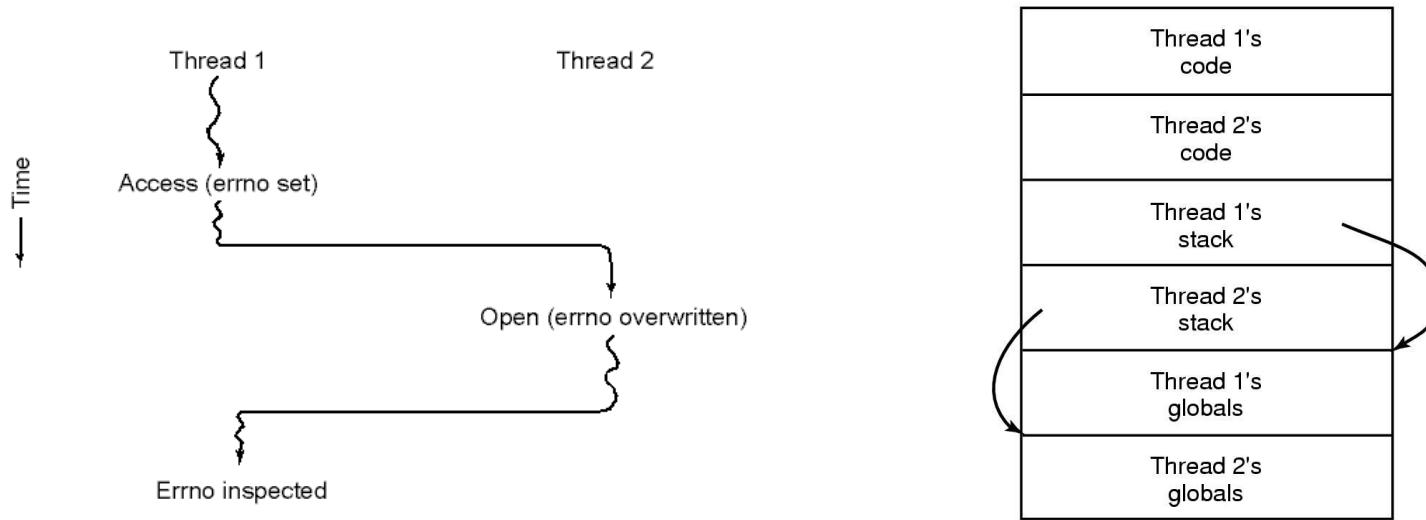
typedef int semaphore;                            /* use your imagination */

semaphore customers = 0;                         /* # of customers waiting for service */
semaphore barbers = 0;                           /* # of barbers waiting for customers */
semaphore mutex = 1;                            /* for mutual exclusion */
int waiting = 0;                                /* customers are waiting (not being cut) */

void barber(void)
{
    while (TRUE) {
        down(&customers);                      /* go to sleep if # of customers is 0 */
        down(&mutex);                         /* acquire access to 'waiting' */
        waiting = waiting - 1;                  /* decrement count of waiting customers */
        up(&barbers);                          /* one barber is now ready to cut hair */
        up(&mutex);                           /* release 'waiting' */
        cut_hair();                           /* cut hair (outside critical region) */
    }
}

void customer(void)
{
    down(&mutex);                           /* enter critical region */
    if (waiting < CHAIRS) {                 /* if there are no free chairs, leave */
        waiting = waiting + 1;              /* increment count of waiting customers */
        up(&customers);                   /* wake up barber if necessary */
        up(&mutex);                      /* release access to 'waiting' */
        down(&barbers);                  /* go to sleep if # of free barbers is 0 */
        get_haircut();                   /* be seated and be serviced */
    } else {
        up(&mutex);                      /* shop is full; do not wait */
    }
}
```

Making single-threaded code multithreaded



Many issues:

- Local variables are fine; globals may need thread-specific implementation (C++ has *thread_local* variable declaration)
- Libraries called may not be thread-safe
 - Who consumes signals?
 - Memory access (malloc etc) needs to be atomic
 - Can be solved via mutual exclusion of libraries (locking)
- Kernel processes may not be thread-aware (e.g. stacks)

Sample Exam Question

Operating systems support data structures (tables) for processes, threads, sockets and files/pipes. Each data structure contains a list of the allocated resource items.

Of the resources listed below,

- (a) Which are allocated to each thread?
- (b) Which of the remaining resources are allocated to each process?

Accounting information, Address space, Child processes, Children's CPU time, CPU time used, File descriptors, Global variables, Group ID, Parent process, Pending alarms, Pointer to data segment, Pointer to stack segment, Pointer to text segment, Priority, Process group, Process ID, Program counter, Registers, Root directory, Scheduling parameters, Signals and signal handlers, Stack pointer, State , Time of next alarm, Time when process started, User ID, Working directory

Items Per Thread

- Program counter
- Registers
- Stack pointer
- State

Items Per Process

- All the remaining items

Sample Exam Question

Describe the differences between creating a thread and creating a process using the code below in your discussion.

```
void main (void)
{
    pid_t childId = fork();
    if (childId == 0)
        printf("I am Tweedledee\n");
    else    printf("I am Tweedledum\n");
}
```

```
void* myFunction (void* arg) { printf("I am Tweedledum\n"); }
```

```
void main (void)
{
    pthread_t childId;
    pthread_create (&childId, NULL, myFunction, NULL);
    printf("I am Tweedledee\n");
}
```

Sample Exam Answer

(About 24 separate facts follow for full marks.)

After *fork()*, there are now **two processes with one thread each**, both continue executing the same line of code after *fork()*.

Whereas after *pthread_create()* (which has a **parameter which is a pointer to a function**), there is still only **one process** but now with **two threads**, where the **new thread begins executing code in a call-back function** (i.e. program counter pointing to first line in this function, not continuing from *pthread_create*).

Specifically, *fork()* creates a **separate child process**, after which the program counter points to the “**if**” in **both processes**. Processes are independent so that if one process ends the other continues. But if the main thread ends, all other threads may terminate.

Both threads share the **same global address space** but since each thread has its own stack, changing a local variable will not affect other threads. Whereas the child process has its **own address space** initialised with the **parents data** at the time of the *fork()* (except the *childId* value which returns **0 to the child and the child PID to the parent**). Unlike threads, processes cannot directly modify each other’s data without using “clever” communication mechanisms such as files, pipes or sockets.

Sample Exam Answer

Threads can communicate using globals, but processes need pipes, sockets, files, etc to communicate.

No guaranteed order of execution in either example.

Also there is **no guarantee that the child thread will print if the parent thread terminates first. But the child process will print even if the parent process terminates first.**

Compared to multiple processes (with one thread), multiple threads in a process are **more difficult to debug** because they can change the same globals or deadlock using semaphores.

Switching execution between threads is much faster than context switching processes and creating a thread is considerably easier (up to 100 times **easier/faster**) **than creating a process** because threads have relatively **little resources** attached to them being, **Program counter, Registers, Stack pointer, State.**

Sample Exam Question

Briefly describe the following three multi-threaded application models:

- (a) Team model,
- (b) Pipeline model,
- (c) Dispatcher/worker model

(a) Team model

Typically all team threads have the same functionality in order to process any request,
where

- * All threads are equal
- * Each thread processes incoming requests on its own

(Team model is used in some active network toolkits such as ANTS and JANOS)

(b) Pipeline model

One thread accepts new requests and may do some processing before passing the
request onto the next thread. The next does some more processing and passes this on
to the next and so forth creating a chain of producers and consumers

(c) Dispatcher/worker model

The process is multithreaded with all requests going to one, the dispatcher thread, while
the other worker threads process the requests. Workers may be specialized and
specific requests could be dispatched to specific workers

Sample Exam Question

Describe the difference between a mutex and semaphore.

Mutexes and semaphores are both used in multiple thread/process applications as synchronisation primitives which block if they have a value of 0, and are not blocked if they have a higher value - **but with the following differences:** (next slide)

Mutex	Semaphore
two states (binary semaphore)	more than two states (counting semaphore)
only the thread that locked or acquired the mutex can unlock it	a thread waiting on a semaphore can be signalled by a different thread
Used to ensure exclusive access to a resource. Protect small critical regions of code – i.e. held for the shortest time possible. Ensures serialisation to non re-entrant code.	Used to restrict the number of simultaneous users of a shared resource up to a maximum number (to synchronise resources such as adding/removing items in a queue)
<u>Locking</u> mechanism: e.g. one thread at a time can use a shared resource	<u>Signalling</u> mechanism: e.g. "I am done so now you can continue"
only a resource locking mechanism (mutual exclusion)	may be used for both resource locking and event notifications
<code>pthread_mutex_init()</code> <code>pthread_mutex_lock()</code> <code>pthread_mutex_unlock()</code> pthread C lib uses different functions	<code>sem_init()</code> <code>sem_wait()</code> <code>sem_post()</code>

Sample Exam Question

Discuss the following multi-tasking problems and solutions, if any.

- a) Busy waiting
- b) Starvation
- c) Race condition

Sample Exam Answer

- a) Busy waiting (or spinning) is a technique in which a process repeatedly checks to see if a condition is true, such as waiting for keyboard input or waiting for a lock to become available or delay execution for some amount of time. Busy waiting should be avoided, as the CPU time spent waiting could have been reassigned to another task. Most operating systems and threading libraries provide semaphores and delay (sleep) functions to be used instead.
- b) Starvation is a multitasking-related problem, where a process is perpetually denied necessary resources. Without those resources, the program can never finish its task. Two or more programs become deadlocked together, when each of them wait for a resource occupied by another program in the same set. On the other hand, one or more programs are in starvation, when each of them is waiting for resources that are occupied by programs, that may or may not be in the same set that is starving. This can be avoided through using semaphores to ensure that as a process begins acquiring a resource, no other process will.
- c) Race conditions arise in software when separate processes or threads of execution depend on some shared state whereby the result is unexpectedly and critically dependent on the sequence or timing of other events. The term originates with the idea of two signals racing each other to influence the output first. One possible solution to this problem is to add synchronization.

Sample Exam Question

In 1971, Edsger Dijkstra set an examination question on a synchronization problem where five computers competed for access to five shared tape drive peripherals. Soon afterwards the problem was retold by Tony Hoare as the “dining philosophers problem”.

- a) Name and briefly describe the two common multi-thread synchronization computing problems in concurrency that the dining philosophers problem is illustrating. [2 marks]
- b) Describe how these two problems are solved in terms of semaphores and mutexes. [2 marks]

Sample Exam Answer

- a) A classic multi-thread synchronization problem of **deadlock** and **resource starvation**. The philosophers never speak to each other, which creates a dangerous possibility of **deadlock when all threads are waiting for resources which cannot be obtained** - i.e every philosopher holds a left fork and waits perpetually for a right fork (or vice versa). **Starvation when some threads do not get access to any resources** (might also occur independently of deadlock) – i.e. if a philosopher is unable to acquire both forks due to a timing issue.

- b) In this problem of multiple threads accessing multiple resources, **the solution is a semaphore per philosopher and one mutex**.

To avoid deadlock, a semaphore is used for each philosopher's acquisition of both forks – i.e. to keep track of which threads are currently using resources=forks.

To avoid starvation one mutex protects the critical region of:

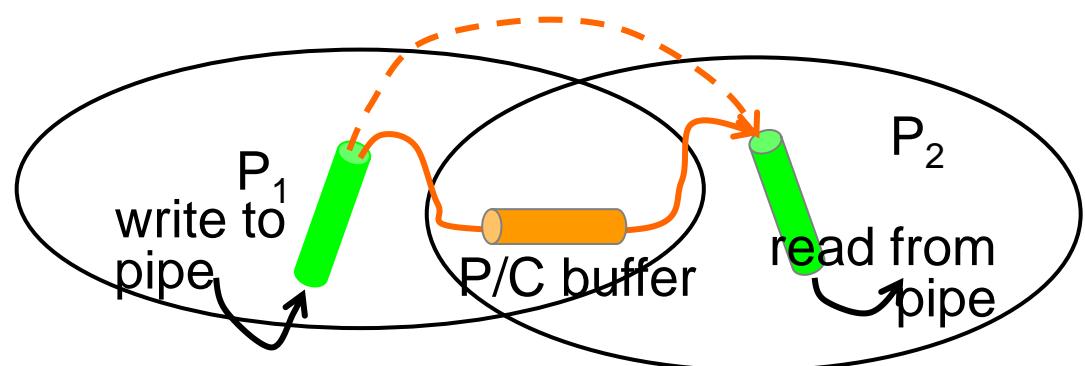
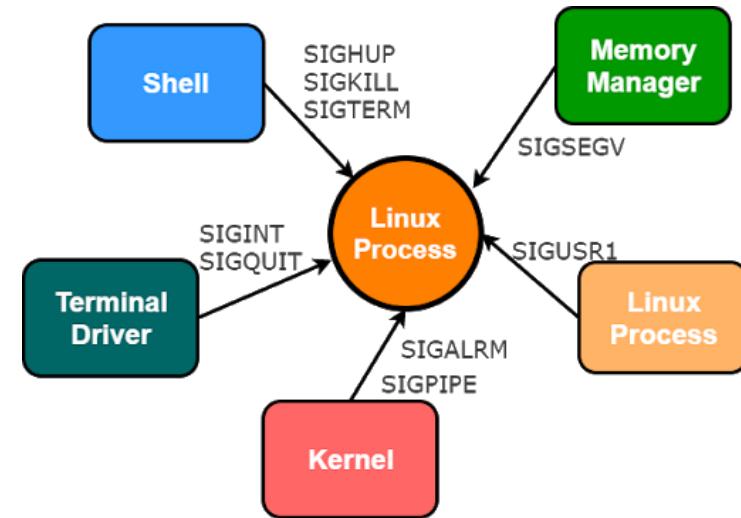
- **being hungry** and attempting to acquire both forks
- **wanting to think** and releasing both forks

ENCE360

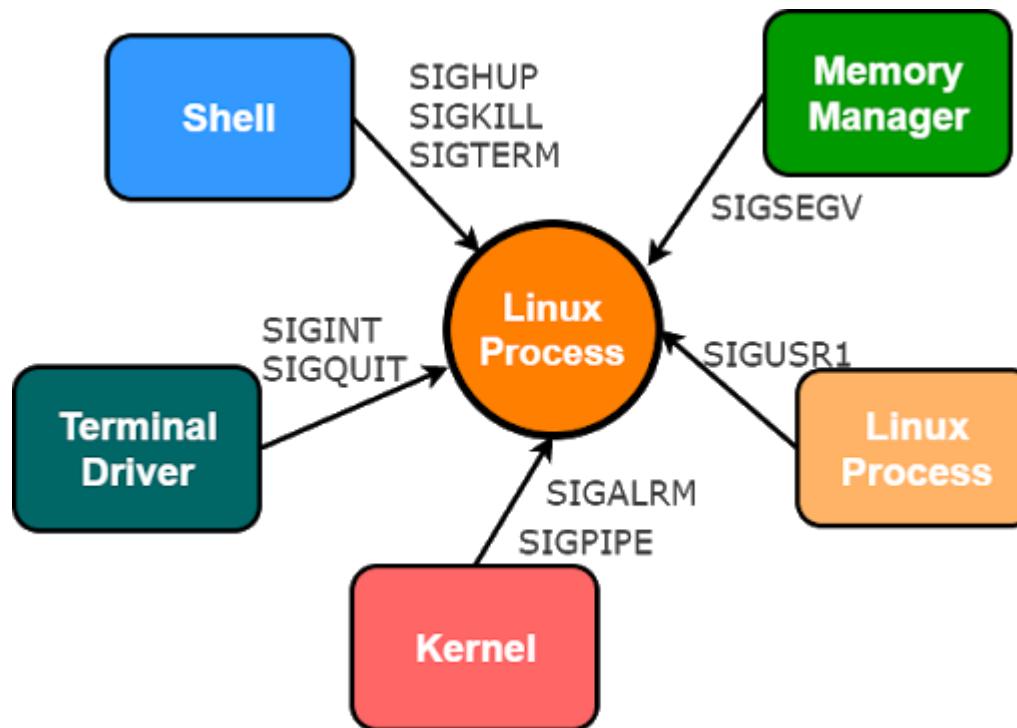
Operating

Systems

Inter-process communication:
signals and pipes



Inter-process communication: signals



Signals

- Used to communicate an exceptional condition:
 - From other processes (including the operating system)
 - by the user from the keyboard
- Fixed set of signals, e.g. (Linux):
 - **SIGINT**: the process is being interrupted; default behaviour is to terminate gracefully
 - **SIGQUIT**: forces the process to terminate and generate a core dump
 - **SIGILL** indicates an attempt to execute an illegal instruction (terminates, generates a core dump)
 - **SIGSEGV** indicates an attempt to access memory beyond that to which the process has permission (terminate, core dump)

Generating signals

- Signals may be generated from various sources:
 - **Hardware**: e.g. divide by zero
 - **Operating System**: e.g. file size limit exceeded
 - **User**: from keystrokes such as ctrl-Z (SIGTSTP), ctrl-C (SIGINT), or via the kill command
 - Other processes such as a child process notifying its parent that it has terminated (**SIGCHLD**), or sending a signal
- Can be sent via system calls:
 - **kill(pid, sig)**: send signal to a **specified process**
 - **raise(sig)**: send signal to **yourself** = **kill(getpid(), sig)**;
 - **alarm()**: send the SIGALRM system to yourself after a specified number of real seconds (terminates)

Directing signals

- signal numbers: 0-31
 - use constants such as SIGINT, SIGUSR1
- Pid: process *or processes* to receive signal
 - pid = 0: all processes in process group
 - if pid = -1: all processes user has permission over (except init)
 - Elevated user, or
 - All processes with same user id
 - Pid < -1: all processes *in process group* -pid

Signal actions

The receiving process can do one of three things upon receiving a signal:

- **Default:** execute action in default table (SIG_DFL)
 - by default, most signals terminate the process
- **Ignore:** set the action to SIG_IGN
 - All but two of the signals can be ignored. The two exceptions are:
 - **SIGSTOP** which stops the process from executing
 - **SIGKILL** which kills the process
- **Catch:** provide a custom handling routine
 - executes a special signal handling routine
 - corresponds to java *try – catch* pair e.g.
 `IOException(e){handle exception}`
 - not SIGSTOP and SIGKILL

Signal default handler behaviour (see man pages)

Signal	Description	Default action
SIGABRT	Process abort signal.	Core dump
SIGNALRM	Alarm clock.	Terminate
SIGFPE	Erroneous arithmetic operation.	Core dump
SIGHUP	<u>Hangup.</u>	Terminate
SIGILL	Illegal instruction	Core dump
SIGINT	Terminal interrupt signal.	Terminate
SIGKILL	Kill (cannot be caught or ignored).	Terminate
SIGPIPE	Write on a pipe with no one to read it.	Terminate
SIGQUIT	Terminal quit signal.	Core dump
SIGSEGV	Invalid memory reference.	Core dump
SIGTERM	Termination signal.	Terminate
SIGUSR1	User-defined signal 1.	Terminate
SIGUSR2	User-defined signal 2.	Terminate
SIGCHLD	Child process terminated or stopped.	Ignore
SIGCONT	Continue executing, if stopped.	Continue
SIGSTOP	Stop executing (cannot be caught or ignored).	Stop
SIGTSTP	Terminal stop signal.	Stop
SIGTTIN	Background process attempting read.	Stop
SIGTTOU	Background process attempting write.	Stop
SIGBUS	Bus error.	Core dump
SIGPOLL	<u>Pollable event.</u>	Terminate
SIGPROF	Profiling timer expired.	Terminate
SIGSYS	Bad system call.	Core dump
SIGTRAP	Trace/breakpoint trap.	Core dump
SIGURG	High bandwidth data is available at a socket.	Ignore
SIGVTALRM	Virtual timer expired / virtual alarm clock	Terminate
SIGXCPU	CPU time limit exceeded.	Core dump
SIGXFSZ	File size limit exceeded.	Core dump

```
#include <signal.h>
/* NOTE: signal() is deprecated */

void sighup(); // routines child calls when signalled from parent
void sigint();
void sigquit();
void sigquit2();

int main (int argc, const char * argv[])
{
    int child_pid;
    if((child_pid = fork()) < 0 ) exit(1); // fork() failed */
    if(child_pid == 0) {
        /* child */
        signal(SIGHUP,  sighup); // Name functions to receive signals
        signal(SIGINT,  sigint); //
        signal(SIGQUIT, sigquit); //
        // signal(SIGQUIT, sigquit2); // use a different handler
        for(; ); // loop forever
    }
    else {
        /* parent */
        printf("\nPARENT: sending SIGHUP signal to child\n\n");
        kill(child_pid,SIGHUP);
        printf("\nPARENT: sending SIGINT signal to child\n\n");
        kill(child_pid,SIGINT);
        printf("\nPARENT: sending SIGQUIT signal to child\n\n");
        kill(child_pid,SIGQUIT);
        // sleep(5);
    }
} // TO BE CONTINUED...

```

```
void sighup() {
    /* ignore the "hangup" signal */
    signal(SIGHUP,sighup); /* reset signal (it reverts to SIG_DFL otherwise) */
    printf("CHILD: I have received a SIGHUP\n");
}

void sigint() {
    /* ignore interrupt, including ctrl-c */
    signal(SIGINT,sigint); /* reset signal */
    printf("CHILD: I have received a SIGINT\n");
}

void sigquit() {
    /* Handle SIGQUIT and die gracefully (rather than core dump) */
    /* The parent will be unaware the child has received a SIGQUIT */
    sleep(1);
    printf("CHILD: My Parent process has killed me!!!\n");
    printf("CHILD: cleaning up...\n");
    sleep(5);
    exit(0);
}

void sigquit2(int sig)
{
    /* Exit after SIGQUIT with the correct status passed back to the caller */
    sleep(1);
    printf("CHILD: I have received a SIGQUIT and will exit\n");
    signal(SIGQUIT, SIG_DFL);
    kill(getpid(), SIGQUIT);
}
```

Handler details

- Multiple types of signal can be handled from one signal handler. For example:

```
void handler(int sigNum) {  
    if      (sigNum == SIGCHLD) printf("got a SIGCHLD\n");  
    else if (sigNum == SIGQUIT) printf("got a SIGQUIT\n");  
}  
  
signal(SIGCHLD, &handler);  
signal(SIGQUIT, &handler);
```

- SIGCHLD signal is sent to a parent process when one of its child processes exits.
 - not necessary to trigger it explicitly with “kill”.
 - enables the OS to clean up resources used by a child process after its termination

Signal summary

- Signals provide a basic communication technique:
 - They do not carry any information
 - Participating processes must know each others process IDs
 - The number of signals is limited
 - Cooperating processes must agree on the meaning of each signal
 - No easy way for the sending process to know if its signal was received
 - Signal manipulation can be tricky

Inter-process communication: pipes



Linux pipes

- Allows one process to pass information to another via a “pseudofile”
- **One-way** (unidirectional) queues that the OS kernel maintains in memory
- Guaranteed to provide FIFO delivery of information
- Two types:
 - **Unnamed pipes** can only be used between two related processes (e.g. child/parent or child/child). They disappear when the processes have finished.
 - **Named pipes** (FIFOs) are persistent since each pipe has an entry in the file system. Providing the permissions on the entry in the file system are set appropriately, any processes may communicate via this pipe.

Stream-based IPC Using Pipes

- Pipe Properties:
 - Synchronised byte stream
 - Operated as a bounded buffer with blocking
 - Each pipe is a one-way stream
 - One to one mapping
- No way to test a pipe for data
- Is it possible to do two-way communication using pipes?
 - Yes (via 2 pipes)

Sending data with pipes

Send to pipe (read & write ends open):

- writing to a full pipe blocks
- if receiver process waiting, then receiver is given message and block released
- if no receiver waiting (blocking), message is queued

Receive from pipe (read & write ends open):

- reading from an empty pipe blocks
- if message ready, obtain message from front of queue and leave
- may have multiple queued receivers

Terminating pipes

When read or write ends closed:

- Reading from a pipe with the write end closed returns EOF
- Writing to a pipe with the read end closed:
 - unnamed pipe: raises SIGPIPE
 - name pipe: blocks waiting for a reader to open the pipe

Unnamed pipes

- Example: combine several programs to perform multiple steps of a single task:

a | b | c

- Standard output of each process in pipe is forwarded to standard input of next process in pipe:

stdout of a goes to stdin of b

stdout of b goes to stdin of c

Creating unnamed pipes

- Parent and child processes share a (read,write) pair of file descriptors:
 - parent uses one of the file descriptors
 - child uses the other
- Generating a pipe using system call:
 1. Create using pipe() system call
 2. Fork a child process
 3. Pass information via the pipe

```
int data_pipe[2];
static char message[BUFSIZ];

int main (int argc, char *argv[]) {

    if (pipe(data_pipe) == -1) {
        perror("Pipe from");
        exit(2);
    }

    switch (fork()) { /* Fork a new process */
        case 0: /* Child process - reader */
            close(data_pipe[1]); // close the write end
            if (read(data_pipe[0], message, BUFSIZ) != -1)
            {
                printf("CHILD: Received %s\n",message);
            }
            break;

        default: /* Parent process - writer */
            close(data_pipe[0]); // close the read end
            if (write(parent_to_child[1], argv[1], strlen(argv[1])) != -1)
            {
                printf("PARENT: Sent %s\n",argv[1]);
            }
    }

    return EXIT_SUCCESS;
}
```

popen() and pclose()

`FILE *popen(const char *command, const char *mode)`

- Macro function:
 - Create a pipe
 - Fork
 - Child invokes shell and runs *command*
- Returns *one* file descriptor: ***mode*** indicates communication direction:
 - ‘w’: parent-to-child (parent writes)
 - ‘r’: child-to-parent (parent reads)
- `pclose(FILE *stream)`
 - waits for the child to terminate and returns the command exit status

Named pipes in the shell

- Created by the **mknod** or **mkfifo** commands
 - E.g. Mkfifo pipe1
- Can be accessed by name by any processes with permission:
 - in one virtual console1, type:
`ls -l > pipe1`
 - and in another type:
`cat < pipe1`
 - output of the command run on the first console shows up on the second console
 - The order in which you run the commands doesn't matter:
 - **Each end blocks waiting for the other**

Named pipes in code

- Same semantics as unnamed pipes
- Created using **mkfifo()** or **mknod()**
- Opened and closed using the standard **fopen** and **fclose** system calls
 - Specify mode ('r' or 'w')
- Use standard file I/O to access
- Resides in memory in kernel

Named pipe server example

```
#define FIFO_FILE          "MY_NAMED_PIPE"

int main(void)
{
    FILE *fp;
    char buffer[80];
    /* Create the FIFO if it does not exist */
    umask(0); // Set permission
    mknod(FIFO_FILE, S_IFIFO|0666, 0); // S_IFIFO file type

    while(1) // loop reading from the buffer
    {
        fp = fopen(FIFO_FILE, "r");
        fgets(buffer, 80, fp); //read from the client
        printf("Your message is here: %s\n", buffer);
        fclose(fp);
    }

    return(0);
}
```

Named pipe client example

```
#include <stdio.h>
#include <stdlib.h>
#define FIFO_FILE "MY_NAMED_PIPE"

int main(int argc, char *argv[])
{
    FILE *fp;
    fp = fopen(FIFO_FILE, "w")
    fputs(argv[1], fp); //send the message

    fclose(fp);
    return(0);
}
```

Named pipe

fgets() fputs() vs fread() fwrite() vs read() write()

- fopen is ANSI C, open is a system call
- fopen is higher level, supports quite a few stdio calls (fprintf, fscanf etc.)
- fopen is buffered, generally higher performance but need to be careful to fflush()
- fopen does end of line translation in text mode (e.g. useful for using windows text files in linux/mac)

Obviously you cannot use fopen for an anonymous/unamed pipe however!

Named pipe

fgets() fputs() vs fread() fwrite() vs read() write()

```
#include <fcntl.h> // for S_IFIFO , O_RDONLY, O_WRONLY
```

```
//create a named pipe:
```

```
mknod("./named_pipe_filename", S_IFIFO | 0666, 0);
```

```
FILE* fp1 = fopen( "./named_pipe_filename" , "w");
FILE* fp0 = fopen( "./named_pipe_filename" , "r");
fputs(buffer, fp1);
fgets(buffer, BUFSIZE, fp0);
//or
fwrite(buffer, 1, strlen(buffer), fp1);
fread(buffer, 1, BUFSIZE, fp0);
```

```
//or
```

```
int fp0 = open("./named_pipe_filename", O_RDONLY);
int fp1 = open( "./named_pipe_filename", O_WRONLY);
write(fp1, buffer, strlen(buffer));
read(fp0, buffer, BUFSIZE);
```

Unnamed vs named pipes

unnamed pipe:

```
// create and open an unnamed pipe:  
int pid[2];  
pipe(pid);  
  
write(pid[1], buffer, strlen(buffer));  
read(pid[0], buffer, BUFSIZE);
```

named pipe:

```
// create and open a named pipe:  
int pid0, pid1;  
mknod("./named_pipe_filename", S_IFIFO | 0666, 0);  
pid1 = open("./named_pipe_filename", O_WRONLY);  
pid0 = open("./named_pipe_filename", O_RDONLY);  
  
write(pid1, buffer, strlen(buffer));  
read(pid0, buffer, BUFSIZE);
```

**Even the read() and write() could be identical
for both named and unnamed pipes
if both used int pid[2]**

```
int pid[2];
```

```
//unnamed  
pipe(pid);  
//named:  
mknod("./named_pipe_filename",S_IFIFO | 0666, 0);  
pid[0] = open("./named_pipe_filename",O_RDONLY);  
pid[1] = open("./named_pipe_filename",O_WRONLY);
```

**// exactly the same read() and write() function calls
// for both the named pipe and unnamed pipe:**

```
read(pid[0], buffer, BUFSIZE);
```

```
write(pid[1], buffer, strlen(buffer));
```

Python named pipes

`os.mkfifo(path, mode=0o666, *, dir_fd=None)`

- Create a FIFO (a named pipe) named *path*.
- FIFOs are pipes that can be accessed like regular files.
- FIFOs exist until they are deleted
- Generally, FIFOs are used as rendezvous between “client” and “server” type processes.
- E.g. the server opens the FIFO for reading, and the client opens it for writing.
- Note that [mkfifo\(\)](#) doesn’t open the FIFO — it just creates the rendezvous point.

Python unnamed pipes

```
import os, sys

print "The child will write text to a pipe and "
print "the parent will read the text written by child..."

# file descriptors r, w for reading and writing
r, w = os.pipe()

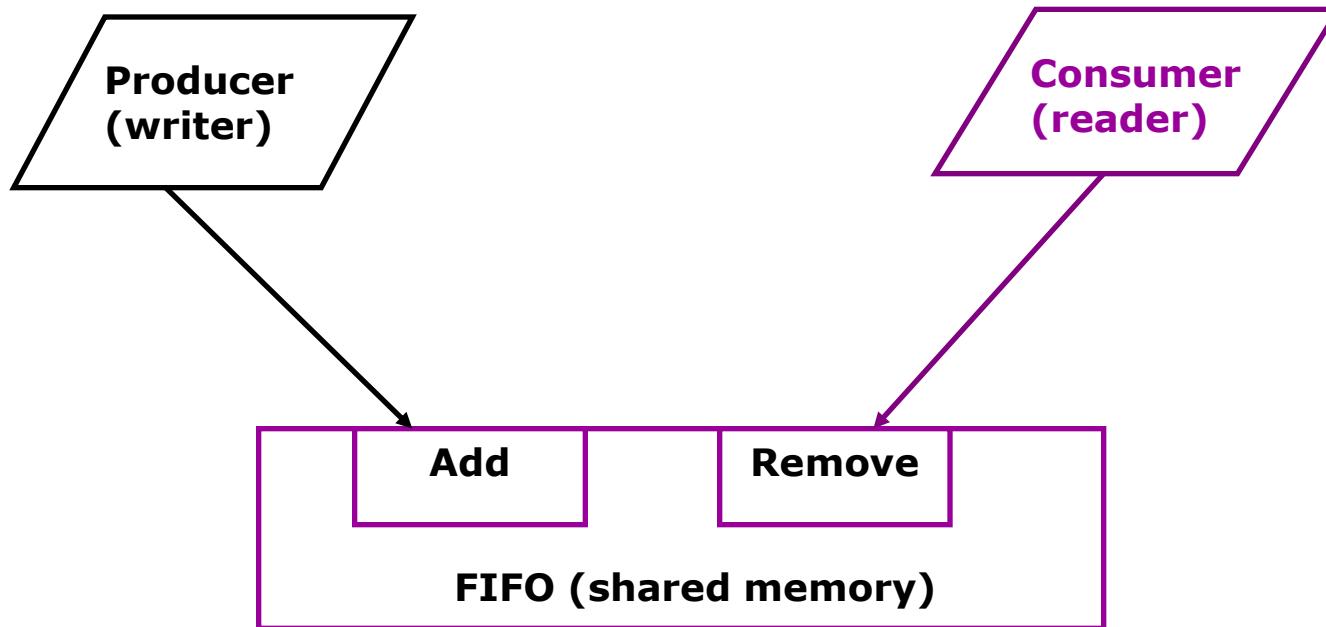
processid = os.fork()
if processid:
    # This is the parent process
    # Closes file descriptor w
    os.close(w)
    r = os.fdopen(r)
    print "Parent reading"
    str = r.read()
    print "text =", str
    sys.exit(0)
else:
    # This is the child process
    os.close(r)
    w = os.fdopen(w, 'w')
    print "Child writing"
    w.write("Text written by child...")
    w.close()
    print "Child closing"
    sys.exit(0)
```

Unnamed Pipe	Named Pipe
only between parent/child processes	between any processes
doesn't need mknod() because the OS knows this is an unnamed pipe from pipe()	needs to tell the OS that this is a pipe (of file type S_FIFO) using mknod()
pipe() does not use a filename and so this pipe does not appear in the file system	open() uses a filename and so this pipe exists as a filename in the file system
disappears when processes have finished	persistent (=> access through shell)
... so don't need to close	... so need to close when finished with
pipe() creates a pipe	mknod() (or mkfifo()) creates a pipe
pipe() also opens both ends of a pipe returning 2 file descriptors in an int array (usually before fork() in the parent so that the child inherits the file descriptors)	open() opens only one end of a pipe returning one file descriptor as an integer so only one file descriptor is available in each process
because pipe() opens both ends, need to close one in each process	open() only opens one end of the pipe
sending process write() to the pipe using input pid[1] – so need to close the output pid[0] so know when reader exits	sending process opens pipe with O_WRONLY, so just use file descriptor from open() to write() to pipe
receiving process read() from pipe using output pid[0] - so need to close the input pid[1] so know when writer exits	receiving process opens pipe with O_RDONLY, so just use file descriptor from open() to read() from pipe
byte stream: can only use read()/write()	file stream: can use read()/write() or fgets()/fputs() or fread()/fwrite()

Implementing pipes

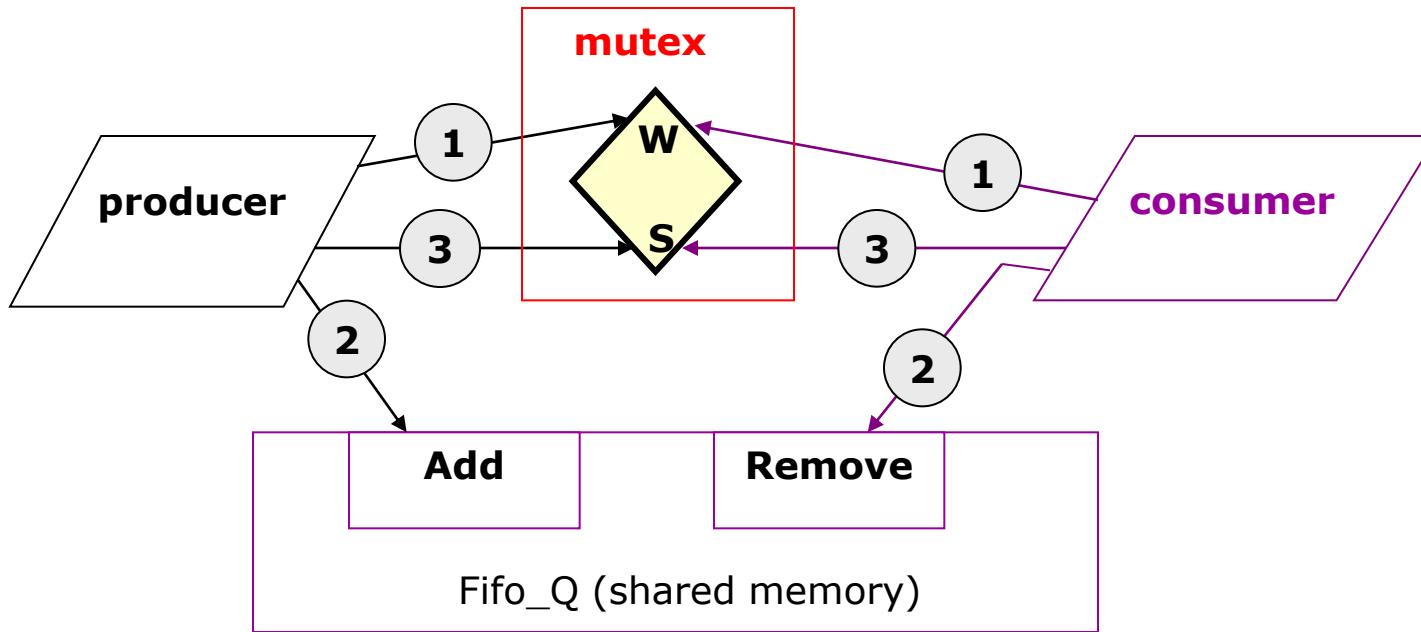
Example of implementing a pipe between two threads using shared memory:

- *Exclusive access to shared memory by reader and writer*
- *Processes block until space/data available*



Implementing pipes (2)

- Exclusive access to shared memory by reader and writer
- Processes block until space/data available



Note: **s** = signal() = sem_post() = up() = pthread_mutex_unlock()
w = wait() = sem_wait() = down() = pthread_mutex_lock()

Adding to Fifo_Q

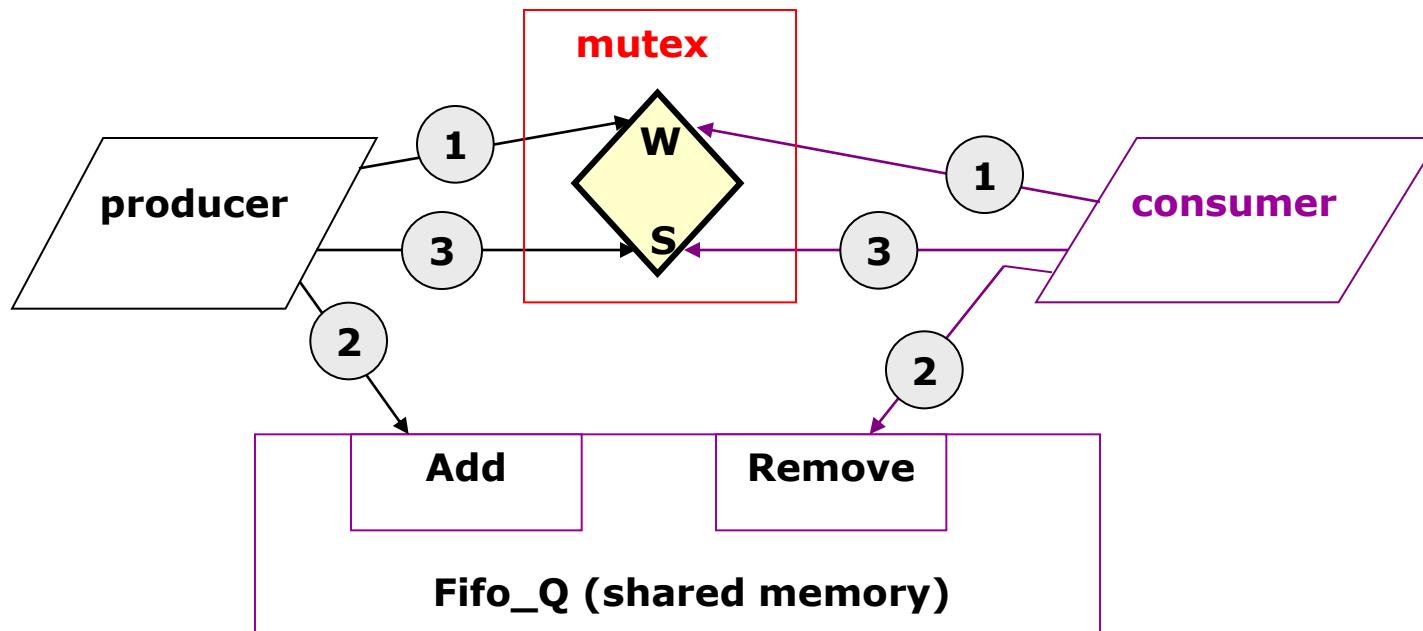
```
Protected_Add(P: packet_buffer)
{
    1 mutex.Wait;          // gain exclusive access
    2 Fifo_Q.Add(P);      // add to pipe
    3 mutex.Signal;       // release exclusive access
}
```

Removing from Fifo_Q

```
Protected_Remove(var P: packet_buffer)
{
    ① mutex.Wait;           // gain exclusive access
    ② Fifo_Q.Remove(P);    // remove from pipe
    ③ mutex.Signal;        // release exclusive access
}
```

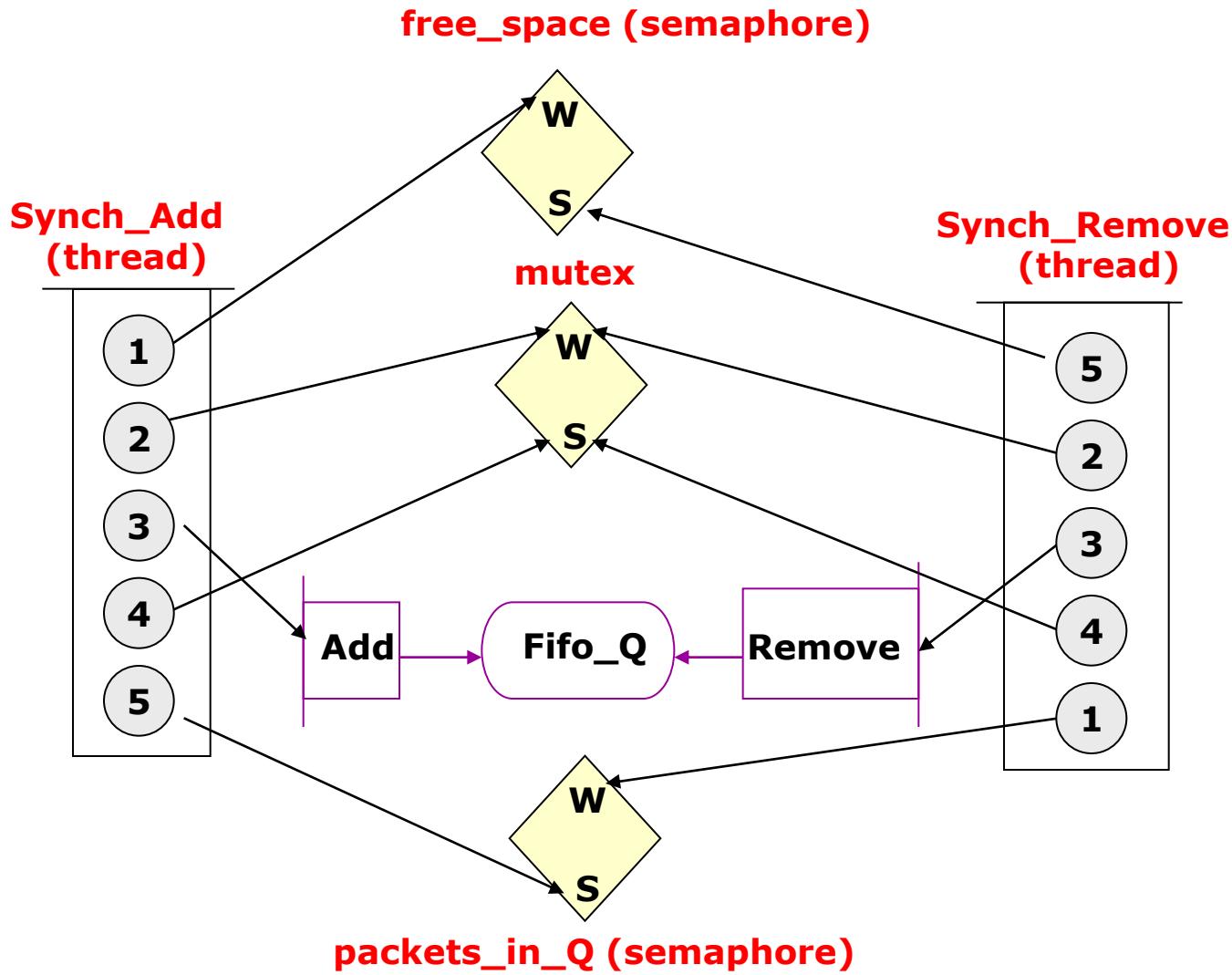
Implementing pipes (3)

- **Exclusive access to shared memory by reader and writer**
- Processes block until space/data available



Note: **s** = signal() = sem_post() = up() = pthread_mutex_unlock()
w = wait() = sem_wait() = down() = pthread_mutex_lock()

Implementing pipes (4)



Note: **s** = `signal()` = `sem_post()` = `up()` = `pthread_mutex_unlock()`
w = `wait()` = `sem_wait()` = `down()` = `pthread_mutex_lock()`

Revised Add to Fifo_Q

```
Synch_Add(P: packet_buffer)
{
    1 free_space.Wait;          // Get space in FIFO
    2 mutex.Wait;              // Gain exclusive access
    3 Fifo_Q.Add(P);           // Add to Q
    4 mutex.Signal;            // Release excl. access
    5 packets_in_Q.Signal;    // New packet ready
}
```

Revised Remove From Fifo_Q

```
Synch_Remove(var P: packet_buffer)
{
    1 packets_in_Q.Wait; // Wait for packet
    2 mutex.Wait;        // Gain exclusive access
    3 Fifo_Q.Remove(P); // Remove from pipe
    4 mutex.Signal;     // release excl. access
    5 free_space.Signal; // 1 more freed space!
}
```

Duplicating file descriptors

- file descriptor is a small positive integer that is used to refer to a file that is being manipulated.
 - Points to the open file description of the file being accessed
- Three standard file descriptors:
 - **0** standard input (STDIN_FILENO)
 - **1** standard output (STDOUT_FILENO)
 - **2** standard error (STDERR_FILENO)
- Dup2(int oldfd, int newfd) duplicates oldfd and uses newfd as the file descriptor number
 - Newfd points to same file as oldfd
 - If newfd was previously open, it is silently closed

Simple example: redirecting STDOUT

```
/*Duplicating file fds*/
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(void) {
int fd;

fd = open("my.file", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR)

dup2(fd , STDOUT_FILENO)

execl("/bin/ls","ls","-l",NULL);

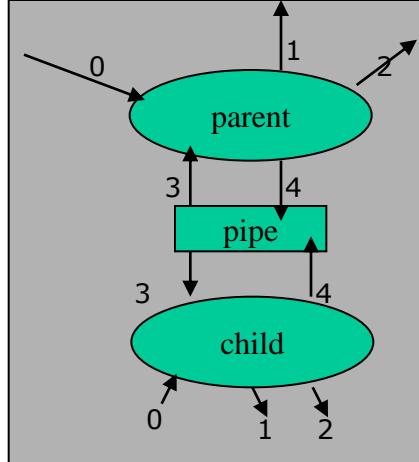
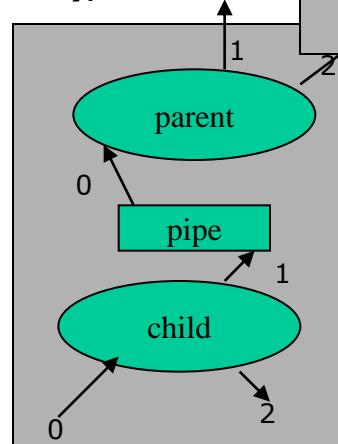
...
}
```

```

#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<fcntl.h>

void main(void)
{
    int fd[2];
    pid_t childpid;
    pipe(fd);
    if ((childpid=fork())==0) { // in child:
        dup2(fd[1], STDOUT_FILENO);
        close(fd[0]);
        close(fd[1]);
        execl("/bin/ls","ls","-l",NULL);
        perror("The exec of ls failed");
    } else { // in parent:
        dup2(fd[0], STDIN_FILENO);
        close(fd[0]);
        close(fd[1]);
        execl("/bin/sort", "sort","-n","+4",NULL);
        perror("the exec of sort failed");
    }
    exit(0);
}

```



0 stdin
1 stdout
2 stderr
3 pipe read
4 pipe write

File Descriptor Table (FDT) after fork parent

0 stdin
1 stdout
2 stderr
3 pipe read
4 pipe write

FDT after fork child

0 read pipe
1 stdout
2 stderr
3 pipe read
4 pipe write

FDT after dup2 parent

0 stdin
1 write pipe
2 stderr
3 pipe read
4 pipe write

FDT after dup2 child

0 stdin
1 write pipe
2 stderr

FDTs after execl

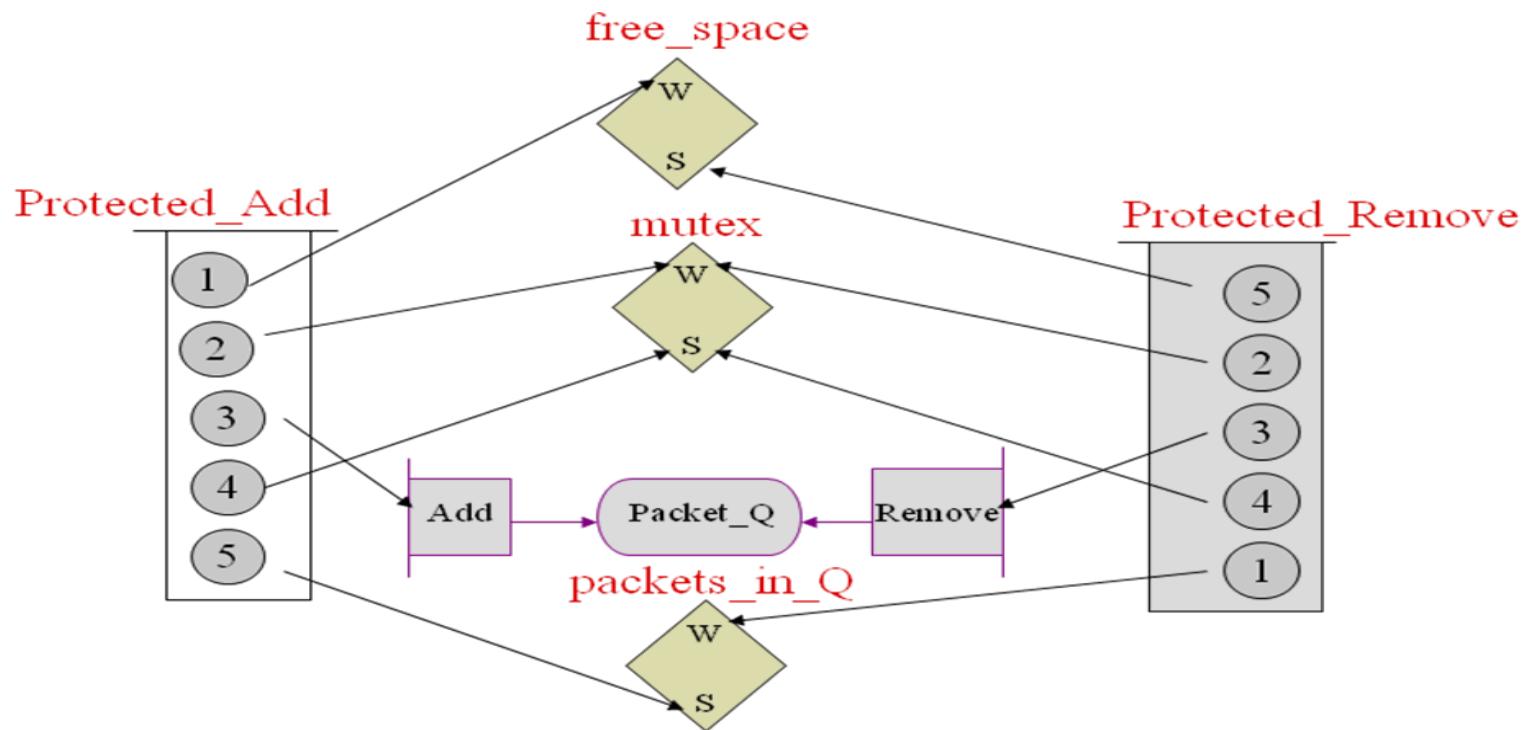
Sample Exam Question

Signals provide a very basic communication technique between processes. Describe signals, their purpose and limitations.

- They do not carry any information
- Participating processes must know each other's process IDs
- The number of signals is limited (examples)
- Cooperating processes must agree on the meaning of each signal
- No easy way for sending process to know if its signal was received
- Most signals terminate the process
- A signal handling routine can catch a signal instead of terminating the process - except SIGKILL, SIGSTOP
- if signal is not reset in signal handling routine, it cannot process more signals
- may come from hardware (/0), OS (file limit), user (ctrl-C), or other process
- only two user defined signals, SIGUSR1, SIGUSR2

Sample Exam Question

In terms of semaphores, mutexes, threads and processes,
explain the operation of a pipe using the following diagram
(which represents a pipe).
Ensure your explanation encompasses *semaphores*, *mutexes*,
threads, *processes* and the *initial state*.)



Sample Exam Answer

Protected_Add and ***Protected_Remove*** are two threads in a process connected by the queue, *Packet_Q* – which is shared memory in this process.

(This is a producer consumer configuration synchronised by two semaphores (*free_space* and *packets_in_Q*) and a *mutex*.)

In the application programs, this (one way) pipe appears like a file that has write-only permission in one process (where file write calls *Protected_Add*) and as a read-only file in the other process (where file read calls *Protected_Remove*).

- * reading from a pipe with the **write end closed returns EOF**
- * writing to a pipe (or FIFO) with the **read end closed raises SIGPIPE**

free_space is used to used to block *Add* until space is available in *Packet_Q*
packets_in_Q is used to block *Remove* until a packet is ready in *Packet_Q*.
mutex ensures that *Add* and *Remove* cannot happen at the same time.

Sample Exam Answer

Initially when the pipe is created, Packet_Q is empty, packets_in_Q=0, free_space=Q_Size, mutex=1.

Protected_Add runs its steps 1 to 5 as follows:

1. free_space.Wait: get access to space in Packet_Q
2. mutex.Wait: gain exclusive access to Add/Remove
3. Packet_Q.Add: add packet to Packet_Q
4. mutex.Signal: release exclusive access
5. packets_in_Q.Signal: packet is now ready to Remove

Protected_Remove runs its steps 1 to 5 as follows:

1. packets_in_Q.Wait: wait for packet
2. mutex.Wait: gain exclusive access to Add/Remove
3. Packet_Q.Remove: remove packet from Packet_Q
4. mutex.Signal: release exclusive access
5. free_space.Signal: one more freed space for Add

Sample Exam Question

Describe the differences between unnamed pipes and named pipes using the code below.

unnamed pipe:

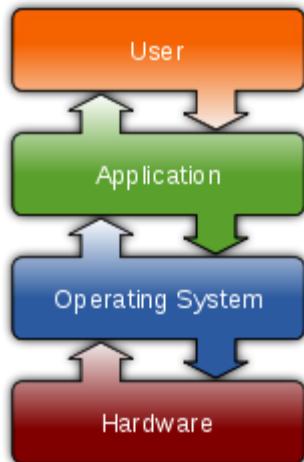
```
// create and open an unnamed pipe:  
int pid[2];  
pipe(pid);  
  
write(pid[1], buffer, strlen(buffer));  
read(pid[0], buffer, BUFSIZE);
```

named pipe:

```
// create and open a named pipe:  
int pid0, pid1;  
mknod("./named_pipe_filename", S_IFIFO | 0666, 0);  
pid1 = open("./named_pipe_filename", O_WRONLY);  
pid0 = open("./named_pipe_filename", O_RDONLY);  
  
write(pid1, buffer, strlen(buffer));  
read(pid0, buffer, BUFSIZE);
```

Sample Exam Answer

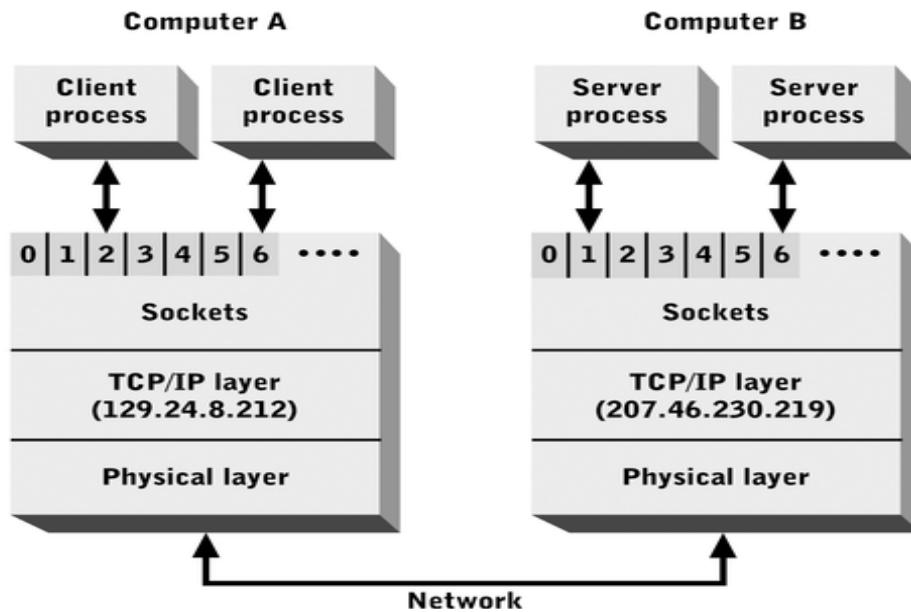
Unnamed Pipe	Named Pipe
only between parent/child processes	between any processes
doesn't need mknod() because the OS knows this is an unnamed pipe from pipe()	needs to tell the OS that this is a pipe (of file type S_FIFO) using mknod()
pipe() does not use a filename and so this pipe does not appear in the file system	open() uses a filename and so this pipe exists as a filename in the file system
disappears when processes have finished	persistent (=> access through shell)
... so don't need to close	... so need to close when finished with
pipe() creates a pipe	mknod() (or mkfifo()) creates a pipe
pipe() also opens both ends of a pipe returning 2 file descriptors in an int array (usually before fork() in the parent so that the child inherits the file descriptors)	open() opens only one end of a pipe returning one file descriptor as an integer so only one file descriptor is available in each process
because pipe() opens both ends, need to close one in each process	open() only opens one end of the pipe
sending process write() to the pipe using input pid[1] – so need to close the output pid[0] so know when reader exits	sending process opens pipe with O_WRONLY, so just use file descriptor from open() to write() to pipe
receiving process read() from pipe using output pid[0] – so need to close the input pid[1] so know when writer exits	receiving process opens pipe with O_RDONLY, so just use file descriptor from open() to read() from pipe
bytestream: can only use read()/write()	filestream: can use read()/write() or fgets()/fputs() or fread()/fwrite()



ENCE360

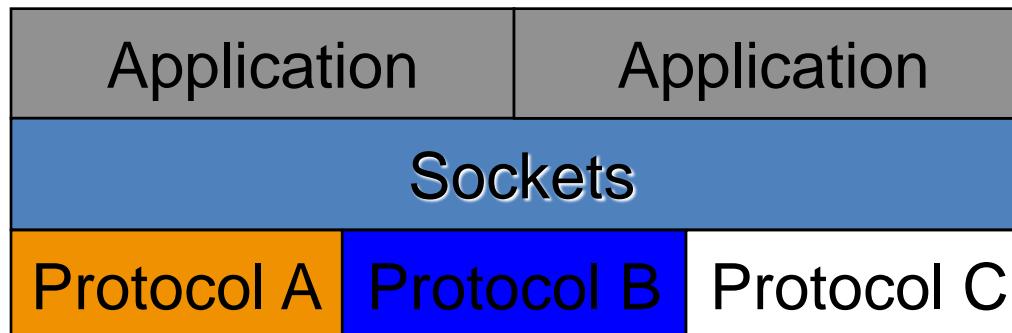
Operating Systems

IPC: sockets



What are sockets?

- Generic abstraction through which an application may send/receive data between processes
 - Including on other systems (unlike pipes)
- Allows an application to “plug-in” a network and communicates with other applications in the same network
- Supports different underlying protocol families



- Connect unrelated processes that may be on different computers.

Socket basics

- Two main *domains* of communication:

- **UNIX/LINUX domain:**

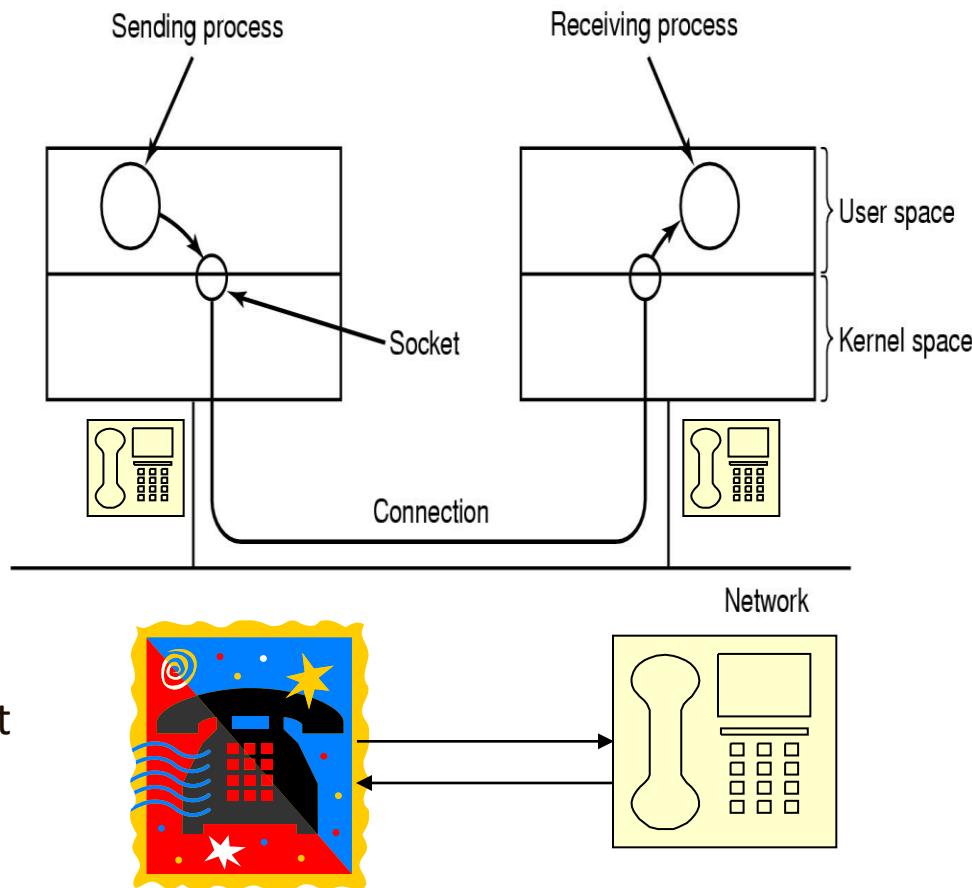
Sockets have actual file names. They can only be used with processes that reside on the same host.

- Type is **AF_UNIX**

- **Internet (network) domain:**

Allows unrelated processes different hosts to communicate via the Internet

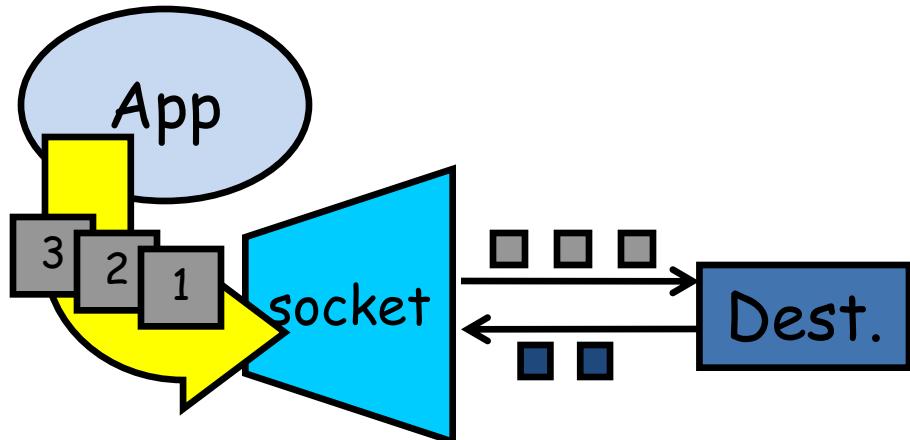
- Type is **AF_INET**



Sockets types (protocols)

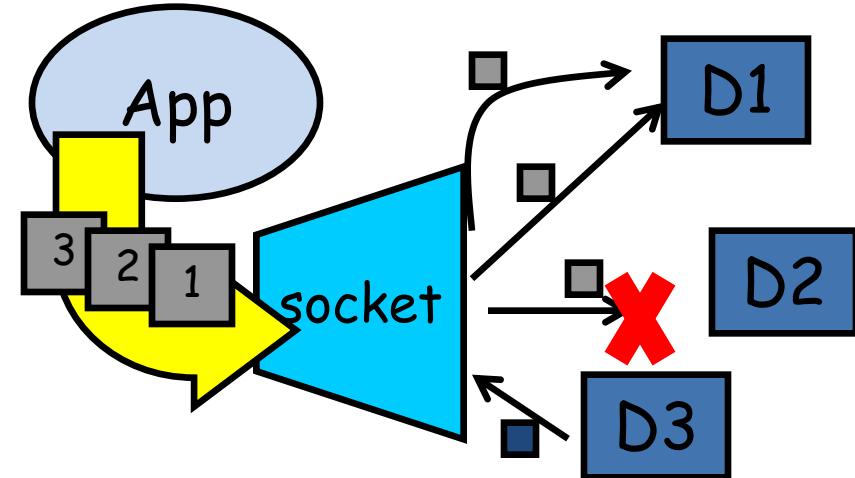
stream socket

- TCP (Transport Control Protocol)
- reliable delivery
- in-order guaranteed
- connection-oriented
- bidirectional

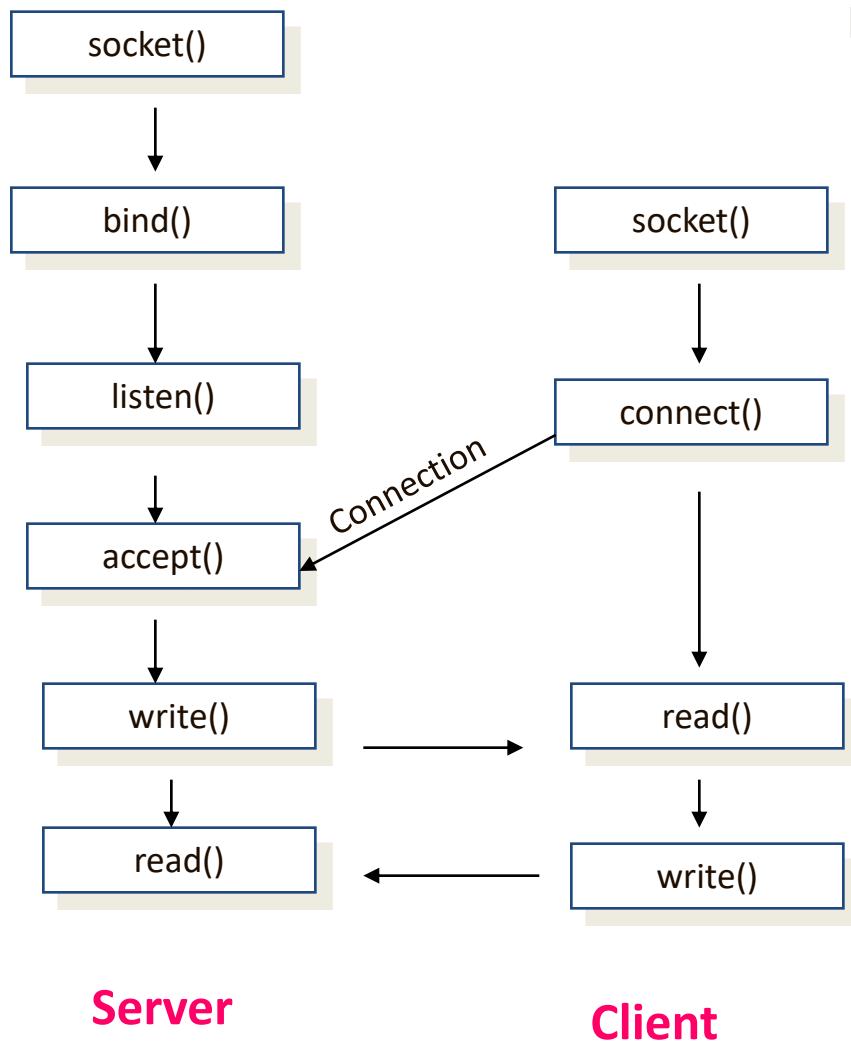


datagram socket

- UDP (e.g. ping)
(User Datagram Protocol)
- unreliable delivery
- no order guarantees
- no notion of “connection” where app indicates a destination for each packet
- can send or receive



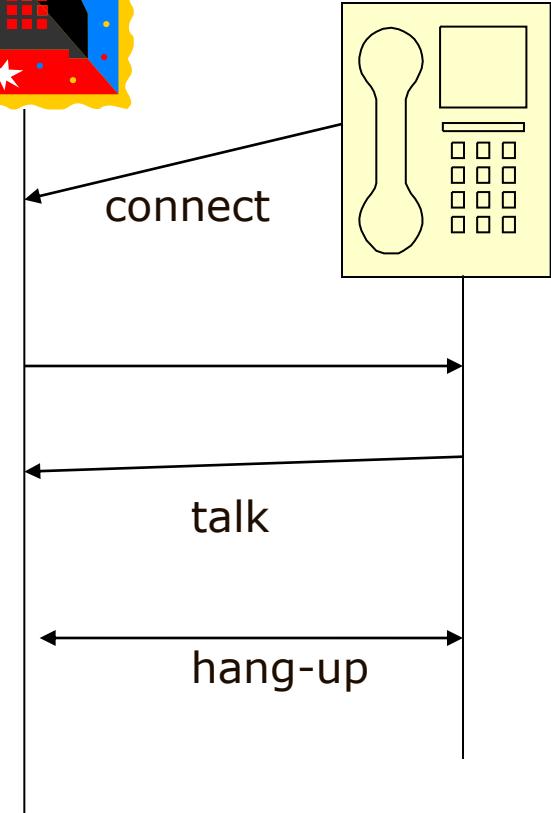
Connection-oriented: stream socket



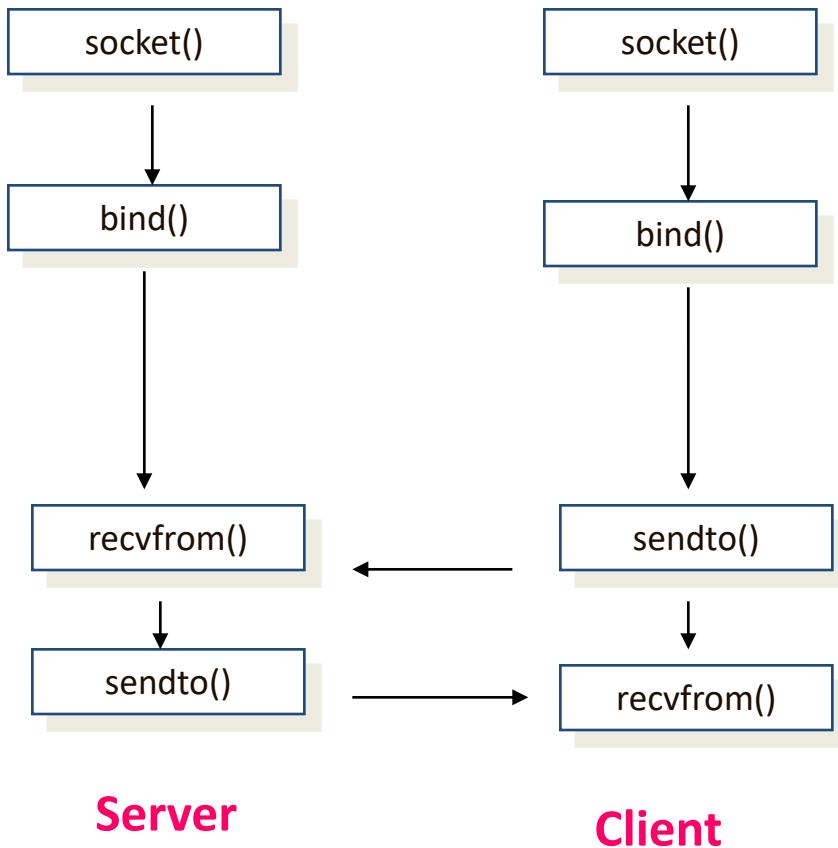
Phone installed



caller



Connectionless: datagram socket



Both sockets must use
bind() call

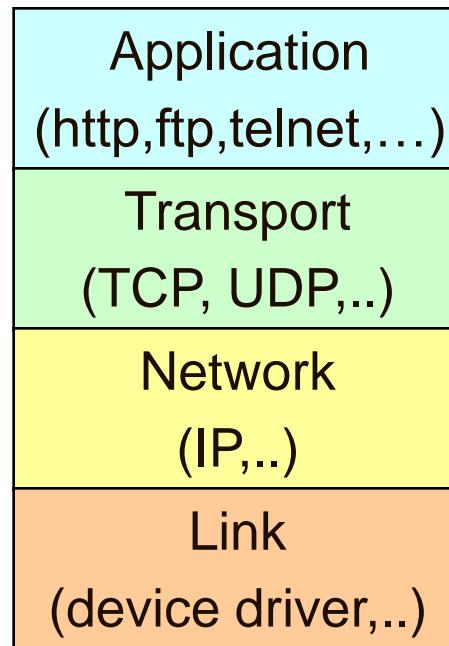
Each message independently
addressed and routed

We won't study these any further

Networking Basics

- Applications Layer
 - Standard apps
 - HTTP
 - FTP
 - Telnet
 - User apps
- Transport Layer
 - **TCP** (Transport Control Protocol)
 - **UDP** (User Datagram Protocol)
 - Programming Interface:
 - **Sockets**
- Network Layer
 - IP
- Link Layer
 - Device drivers

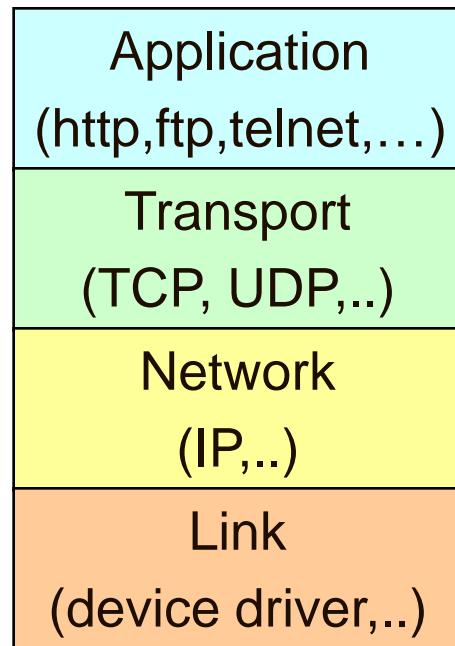
TCP/IP stack



Networking Basics

- TCP (Transport Control Protocol)
 - connection-oriented protocol
 - Provides a reliable flow of data between two computers
- Example applications
 - HTTP
 - FTP
 - Telnet

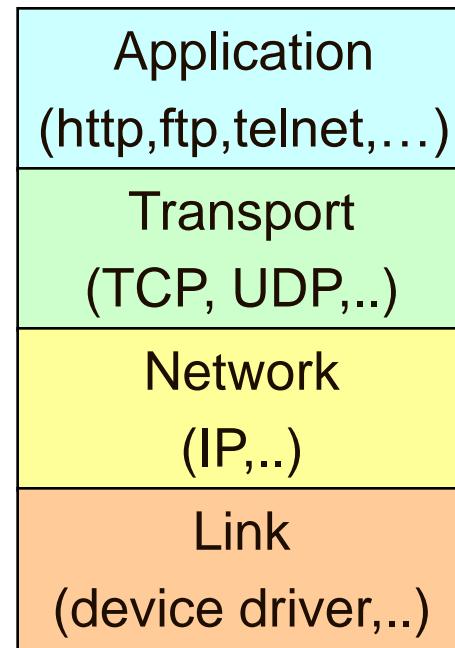
TCP/IP stack



Networking Basics

- **UDP** (User Datagram Protocol) is a protocol that sends independent packets of data, called *datagrams*, from one computer to another with no guarantees about arrival.
- Example applications:
 - Clock server
 - Ping

TCP/IP stack



Web protocols & services

All resources are identified by a unique **Uniform Resource Locator** (URL). The URL has four parts: protocol, host, port and resource.

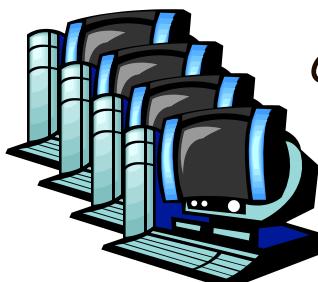
- **Protocol** – an optional header specifying the resource access protocol
- **Host** – the IP number or registered name of an Internet host computer or device.
- **Port** – an optional port number that specifies the **socket**.
- **Resource** – the complete path name of a resource on the host.

Names versus addresses

- Applications refer to the destination by name
- Within the communication domain sockets are referred to by addresses
 - May need to look up the name
 - usually done outside the operating system



atlas.cs.uga.edu
(128.192.251.4)



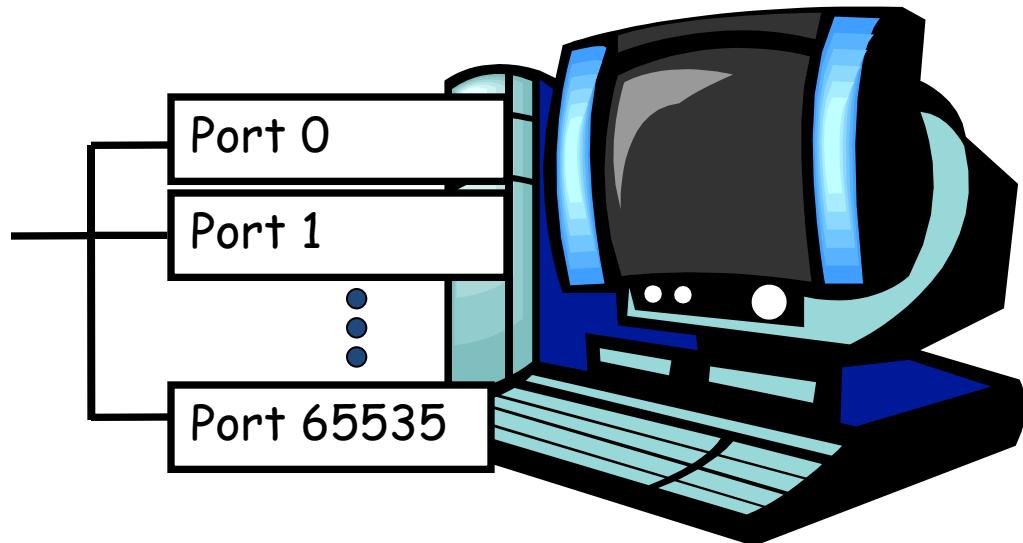
church.cse.ogi.edu
(129.95.50.2,
129.95.40.2)
(multi-homed)



www.google.com
(216.239.35.100)

Routing requests: ports

- Each host has 65,536 ports
- Some ports are *reserved for specific apps*
 - 20,21: FTP
 - 23: Telnet
 - 80: HTTP

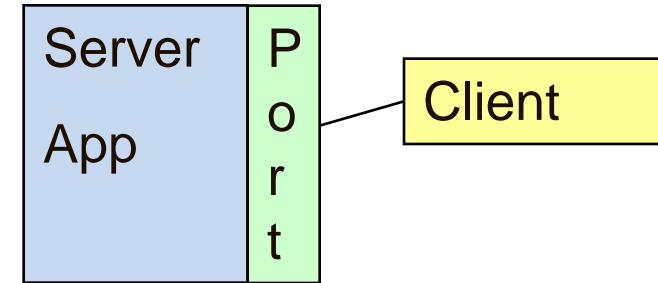
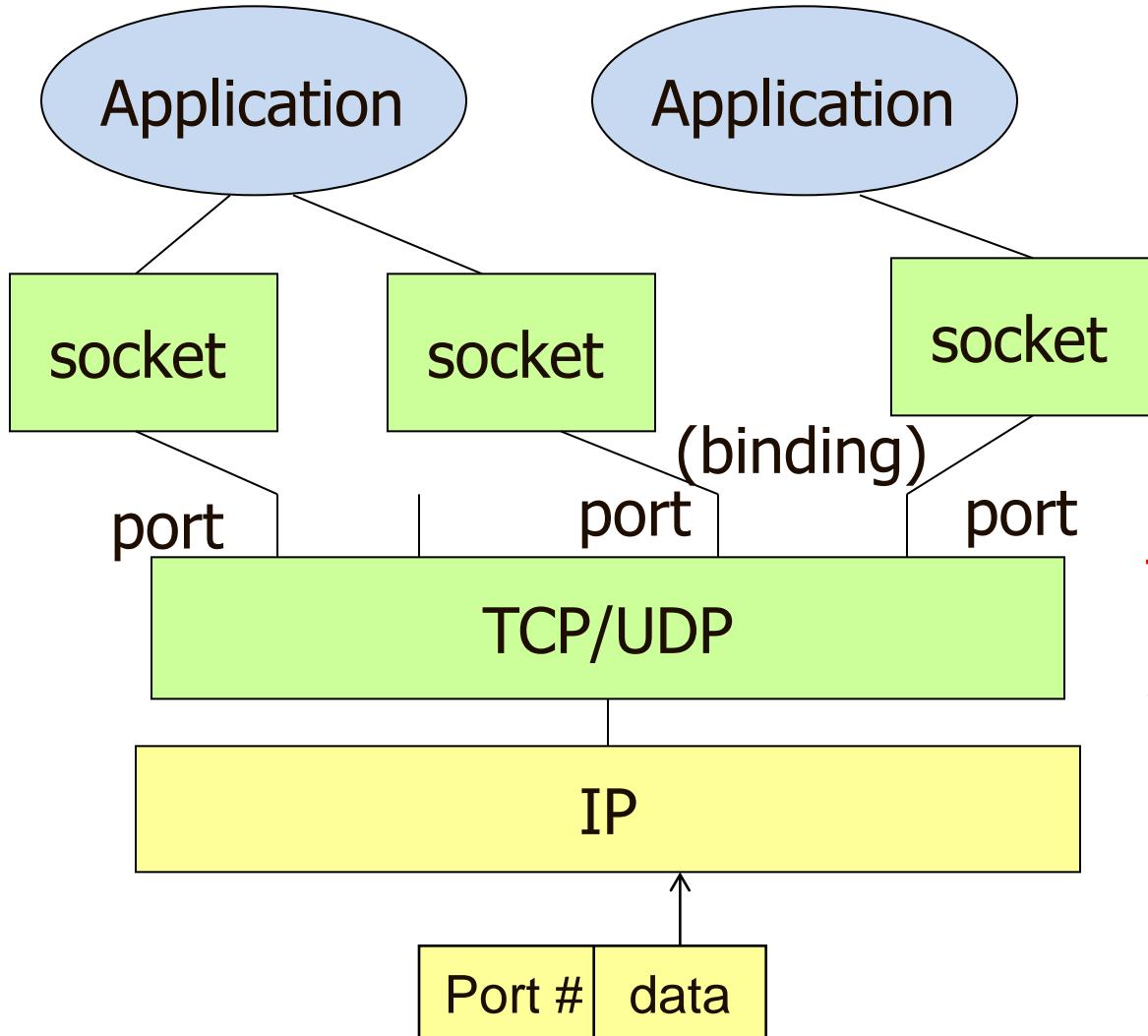


A socket provides an interface for an application to send/receive data to/from the network through a port

~1023 ports are reserved

User level process/services generally use port number value ≥ 1024 (see /etc/services)

Sockets and ports



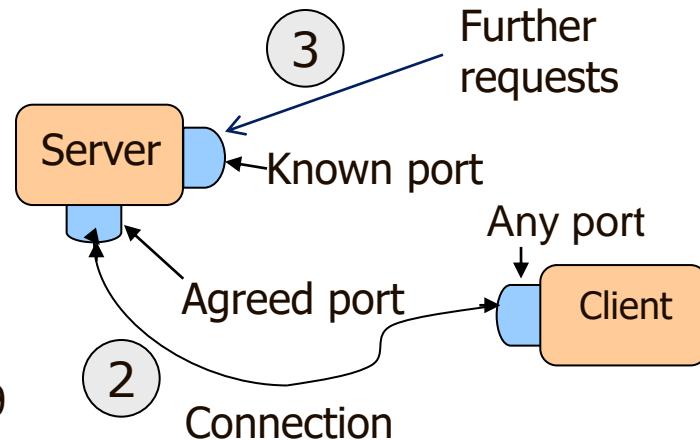
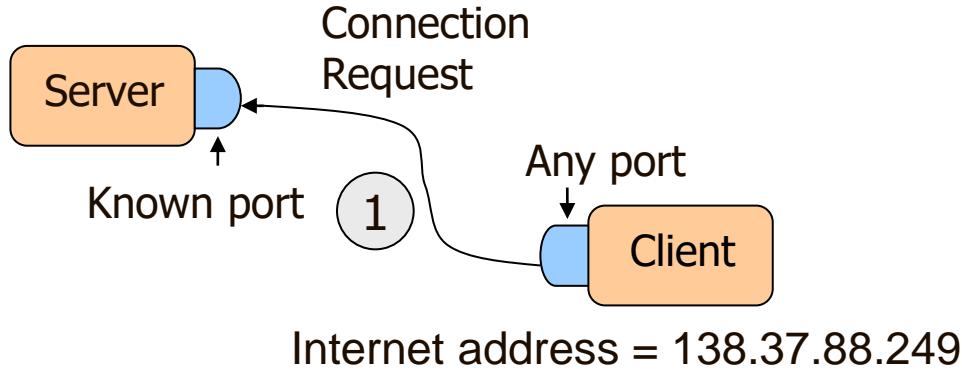
Transport layer routes messages to applications via the port number

The sockets API

- Network transport interface to higher-level applications
 - Basic building block of internet and WWW
- Very similar to performing file I/O
 - Socket handle is treated like file handle
 - Streams used in file I/O operation are applicable to socket-based I/O
- Programming language independent
 - A socket program written in Python can communicate with a socket program written in C or Java
- Socket: one end of a two-way communication pipe
 - Identified by an internet address and local port number

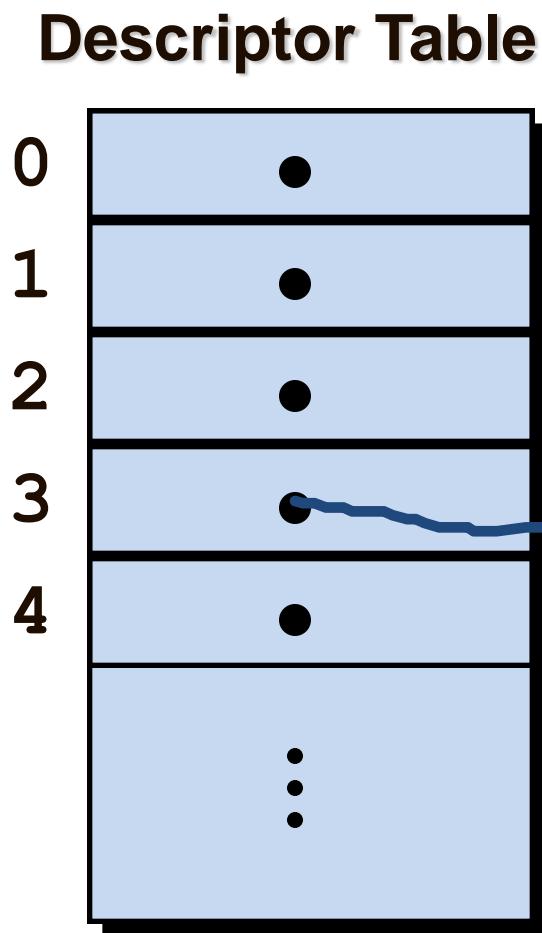
Establishing a connection

Internet address = 138.37.94.248



1. Client sends a request to the server by hostname and port number
2. Server accepts the connection, creating a new socket bound to a different port
 - Common to service the actual connection in a separate thread/process
3. The original server port is again ready to listen for other connection requests

Socket descriptor and data Structure



Family: PF_INET
Service: SOCK_STREAM
Local IP: 111.22.3.4
Remote IP: 123.45.6.78
Local Port: 2249
Remote Port: 3726
Protocol state: Established

Sent queue: empty
Received queue: empty

Addresses are empty when the socket is created
- Filled in when a connection is established

Person traps: network byte order

- All numeric values stored in a `sockaddr_in` must be in **network byte order**.
 - `sin_port` a port number.
 - `sin_addr` an IP address.

Common Mistake:
Ignoring Network Byte Order



Byte ordering

```
union {
    u_int32_t addr; /* 4 bytes address */
    char c[4];
} un;

un.addr = 0x8002c25f; /* 128.2.194.95 */
/* c[0] = ? */
```

- Big Endian →

c[0]	c[1]	c[2]	c[3]
128	2	194	95

 - Sun Solaris, PowerPC, ...
- Little Endian →

95	194	2	128
----	-----	---	-----

 - Intel x86, DEC Alpha, ...
- Network byte order = **Big Endian**

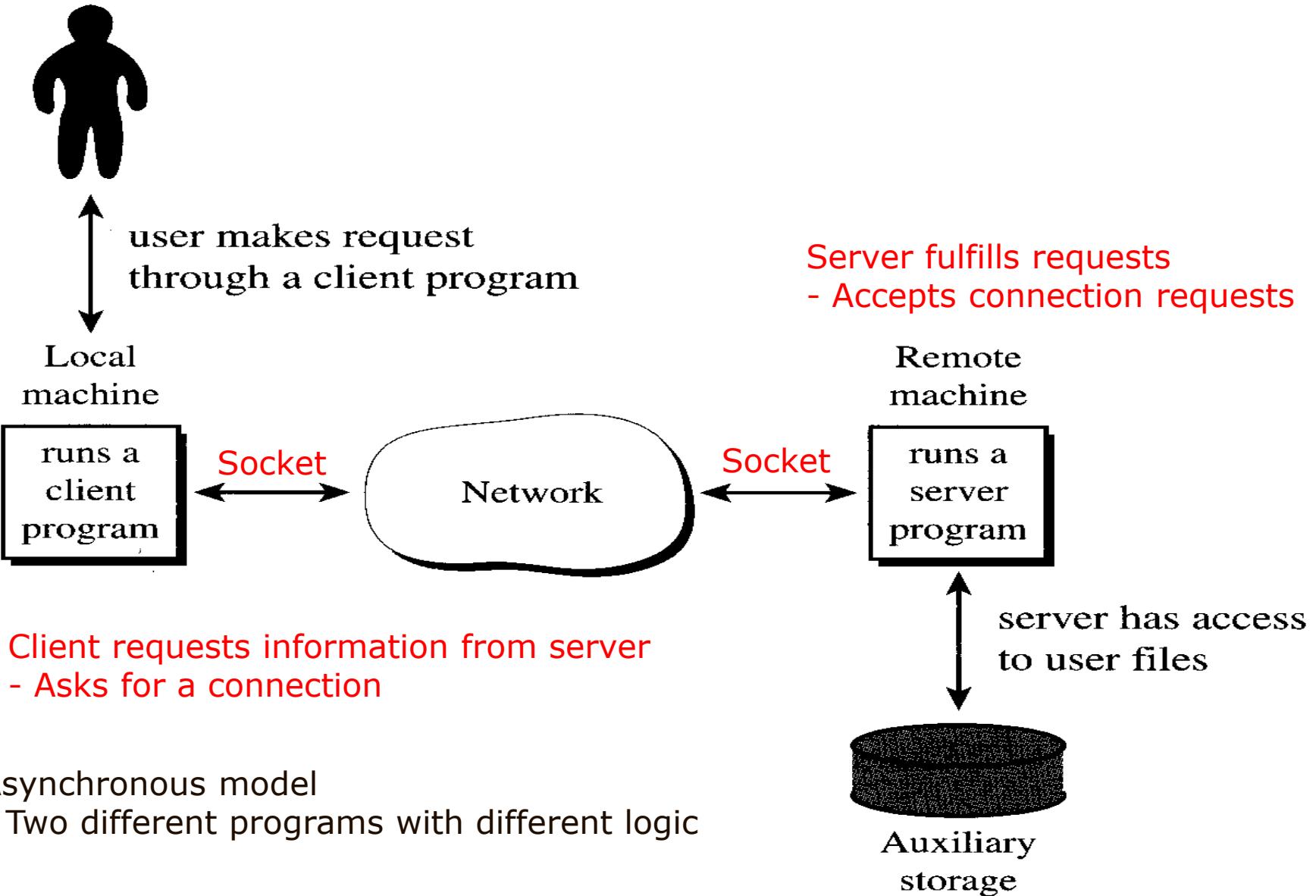
Network byte order functions

‘**h**’ : host byte order ‘**n**’ : network byte order
‘**s**’ : short (16bit) ‘**l**’ : long (32bit)

```
uint16_t htons(uint16_t);  
uint16_t ntohs(uint16_t);
```

```
uint32_t htonl(uint32_t);  
uint32_t ntohl(uint32_t);
```

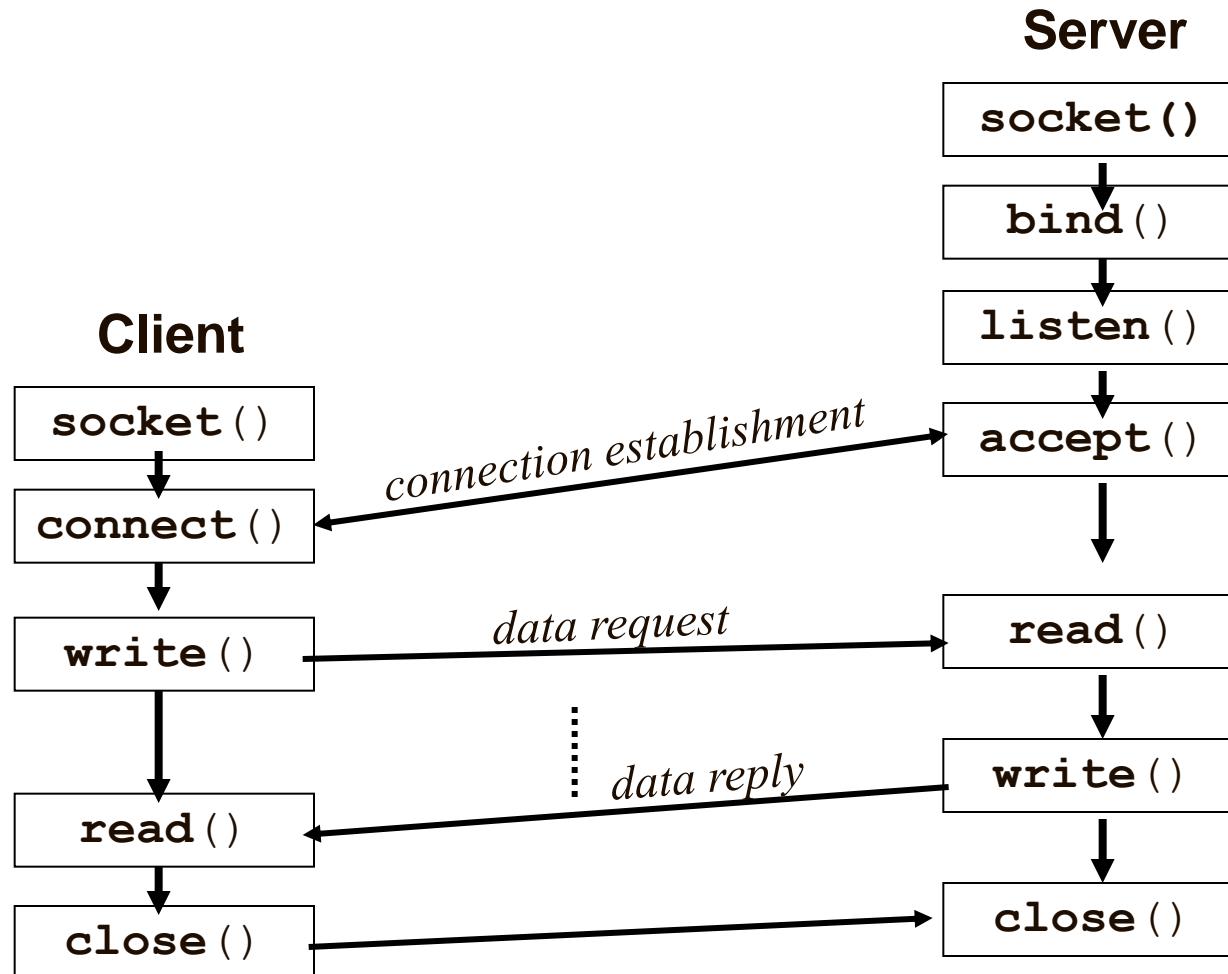
Sockets: client-server model



Socket API requirements

- Specify local and remote communication endpoints
- Initiate a connection (client)
- Wait for incoming connection (server)
- Send and receive data
- Terminate a connection gracefully
- Error handling

Socket client-server interaction



Simple Socket Client

```
#include <stdio.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <sys/socket.h>

int main() {
    int client_socket = socket(AF_INET, SOCK_STREAM, 0);      Create socket

    struct addrinfo *server_address = NULL;
    struct addrinfo hints;
    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_family = AF_INET;
    hints.ai_socktype = SOCK_STREAM;
    getaddrinfo( "localhost", "1234", &hints, &server_address);
    connect(client_socket, server_address->ai_addr, server_address->ai_addrlen); Send connection request  
to server address

    char data[512] = {'\0'};
    int message_length = sprintf(data, "hello server");
    message_length = write(client_socket, data, message_length);
    message_length = read(client_socket, data, 512);
    printf("from server: %s\n", data); Communicate

    close(client_socket); Close socket

    return 0;
}
```

Simple Socket Server

```
#include <stdio.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <sys/socket.h>

int main() {
    int server_socket = socket(AF_INET, SOCK_STREAM, 0); Create socket

    struct sockaddr_in server_address, client_address;
    server_address.sin_family = AF_INET;
    server_address.sin_addr.s_addr = INADDR_ANY;
    server_address.sin_port = htons(1234);
    bind(server_socket,(struct sockaddr *)&server_address,sizeof(struct sockaddr_in)); Bind to address, port

    listen(server_socket, 5); Listen for a request

    socklen_t l = sizeof(struct sockaddr_in); /* client address length */
    int client_socket = accept(server_socket, (struct sockaddr *) &client_address, &l); Accept request

    char data[512] = { '\0' };
    int message_length = read(client_socket, data, 512);
    printf("from client: %s\n", data);
    message_length = sprintf(data, "back to ya client");
    message_length = write(client_socket, data, message_length); Communicate

    close(client_socket); Close socket

    return 0;
}
```

Python sockets are much simpler!

Simple Python Socket Server

```
import socket

server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

server_socket.bind("", 1234)

server_socket.listen(5)

client_socket, address = server_socket.accept()

data = client_socket.recv(512)
print "from client: ", data
data = "back to ya client"
client_socket.send(data)

client_socket.close()
```

Simple Python Socket Client

```
import socket

client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

client_socket.connect(("localhost", 1234))

data = "hello server"
client_socket.send(data)
data = client_socket.recv(512)
print "from server: ", data

client_socket.close()
```

Create a socket: `socket()`

`int socket(int family, int type, int protocol)`

Example – internet socket:

```
if (( sock = socket(AF_INET,SOCK_STREAM,0)) == -1)
    { printf("error opening socket");
      exit(-1);
    }
```

- Creates a socket:
 - Not associated with a client or server process yet
- Socket type:
 - SOCK_STREAM vs. SOCK_DGRAM

Server code: get socket and initialize struct

```
main (int argc, char *argv[])
{
    int rc,                                /* system call return code */
        msgsock, sock,                      /* server/listen socket descriptors */
        adrlen,                            /* sockaddr length */
        cnt;                               /* number of bytes I/O */

    struct sockaddr_in      sockname;       /* Internet socket name */
    struct sockaddr_in      *nptr;          /* ptr to get port number after bind*/
    struct sockaddr         addr;           /* generic socket name */

    char buf[80];                          /* I/O buffer for messages - could be larger*/

/* Create socket */
if (( sock = socket(AF_INET, SOCK_STREAM, 0)) < 0 ) {
    perror("Get server socket");
    exit(1);
}
/* Initialize the fields in the Internet socket name structure*/
sockname.sin_family = AF_INET;           /* Internet address */
sockname.sin_port = 0;                   /* System will assign port # */
sockname.sin_addr.s_addr = INADDR_ANY;   /* "Wildcard" */
}
```

The sockaddr structures

Generic address (all)

```
struct sockaddr {  
    u_short sa_family;  
    u_short sa_data[14];  
}
```

Type returned by
some socket calls

Unix domain sockets

```
struct sockaddr_un {  
    short sun_family; //AF_UNIX  
    char sun_path (108); //path  
};
```

Internet domain sockets

```
struct sockaddr_in {  
    short sin_family; // AF_INET  
    u_short sin_port;  
    struct in_addr sin_addr;  
    char sin_zero[8]; // unused  
};
```

All have same length (number of bytes)

Using sockaddr_in structure

Examples

```
sock_addr.sin_addr.s_addr = htonl(INADDR_ANY);      // any incoming address
sock_addr.sin_port = htons(0);                      // Any free port
sock_addr.sin_family = AF_INET;                     // Internet domain (vs unix)
```

```
sock_addr.sin_addr.s_addr = inet_addr("127.59.69.7");    // Only this address
sock_addr.sin_port = htons(5115);                    // Specify this port
sock_addr.sin_family = AF_INET
```

Usage

```
struct sockaddr_in caller;           // Internet address struct
struct sockaddr_in *nptr;            // Ptr for receiving address information
struct sockaddr   addr;             // Generic socket address for library calls
```

Binding the server socket: bind()

```
int bind(int sockfd, struct sockaddr *myaddr, int addrlen)
```

Example:

```
if (bind(sock, (struct sockaddr *) &sock_addr, sizeof(sock_addr)) < 0)
    { printf(": error binding socket to local address");
      exit(-1);
    }
```

This call binds a (server) socket to an IP address and port
It is not associated with a client or server process yet

1. **sock** is a **file descriptor** for the socket
2. **sock_addr** is a **pointer to a structure** of type sockaddr

Server code – bind and get port number

```
if (bind(sock, &sockname, sizeof(sockname) ) < 0 ) {
    close(sock);                                /* always close socket */
    perror("network server bind");
    exit(2);
}

/*Get the port number assigned to the Internet socket and
print it for use as arg to client*/

adrlen = sizeof(addr);
if ( ( rc = getsockname( sock, &addr, &adrlen ) ) < 0 )
{
    perror("network server getsockname");
    close (sock);
    exit(3);
}

nptr = (struct sockaddr_in *) &addr;
printf("\n\\Server has port number: %d\\n", ntohs(nptr->sin_port));
```

Wait for connections: listen(), accept()

- **Listen()** establishes the socket as accepting connections
- Does not suspend the process execution
- Accepts messages only (can't send)
- Sock is a *file descriptor* for the socket (int)

```
/* listen for connections on a socket - "hint" max of 5 in buffer */

listen (sock, 5);
```

- **Accept()** waits for a request (blocks)
- Returns a new temporary socket connected to the caller
- Listening socket is freed to accept more requests

```
length = sizeof(caller);
if ((msgsock = accept(s, (struct sockaddr *)&caller, &length)) < 0) {
    printf("\nNetwork server accept failure %d\n", errno);
    perror("network server");
    close(s);
    exit(5);
} /* message received, process it */
```

Client programming

```
/* Expected parameters on command line:

argv[0] -- name of executable (client)
argv[1] -- the host name to which connection is desired (localhost)
argv[2] -- the port number to be used by the client: the
          value is the port number assigned to the
          server by the server's host system. */

{
    int sock,           /* socket descriptor */
    val,               /* scratch variable */
    cnt;               /* number of bytes I/O */

    struct sockaddr_in srvr;      /* Internet socket name (addr) */
    struct sockaddr_in *nptr;     /* pointer to get port number */

    char buf[80];        /* I/O buffer, for the message - could be larger */

/* For lookup in /etc/hosts file. */
    struct hostent *hp, *gethostbyaddr();
```

Client code to establish the socket

```
{  
    int sock,                      /* socket descriptor */  
        val,                         /* scratch variable */  
        cnt;                         /* number of bytes I/O messages*/  
  
    struct sockaddr_in srvr;        /* Internet socket name (addr) */  
    struct sockaddr_in *nptr;        /* pointer to get port number */  
  
    char buf[80];                  /* I/O buffer, kind of small , could be larger*/  
  
    /* For lookup in /etc/hosts file. */  
    struct hostent *hp, *gethostbyaddr();  
  
    if (( sock = socket(AF_INET, SOCK_STREAM, 0)) < 0 ) {  
        perror("network client");  
        exit(2);  
    }  
    /* Convert user-supplied port number to integer. */  
  
    srvr.sin_port = htons( atoi(argv[2]) );      /* Server port number */  
    srvr.sin_family = AF_INET;                   /* Internet domain */
```

Connecting to the server: connect()

```
int connect (int sockfd, struct sockaddr *myaddr, int addrlen)
```

Example:

```
if (connect(sock, ( struct sockaddr *) &sock_addr, sizeof sock_addr) == -1)
{ printf(": socket connection error");
  exit(-1);
}
```

- used in connection oriented communications
- Connects a client to a server

Client: getting the host address by name

```
hp = gethostbyname (argv[1]) ;

if ( hp == NULL ) {
    perror("network client");
    close (sock);
    exit(3);
}

else {
    printf("\tThe official host name is %s\n", hp -> h_name);
    printf("\tThe first host address is %lx\n",
           ntohl ( * (int * ) hp -> h_addr_list[0] ) );
    printf("\tAlias names for the host are:\n");

    while ( *hp -> h_aliases )          //get all aliases
        printf( "\t\t%s\n", *hp -> h_aliases++ );
}

bcopy ( hp -> h_addr_list[0], &srvr.sin_addr.s_addr,
        hp -> h_length ); /* save the first name */
```

```
struct hostent {
    char *h_name;      //official name
    char **h_aliases; // alias list
    int h_addrtype;   //address type
    int h_length;     // address length
    char **h_addr_list; // address list
#define h_addr h_addr_list[0]
/*backward compatibility*/
};
```

Establishing the connection

```
/* Client: Establish socket connection with (remote) server. */

if ( ( connect ( sock, &srvr, sizeof(srvr) ) ) < 0 ) {
    printf("network client %s connect failed %d\n", argv[0], errno);
    perror("network client on connect");
    close (sock);
    exit(4);
```

```
/* SERVER: Set up "infinite loop" to accept client requests.
   We will use a different method to get the client's address...*/

while (1) {
    if ( ( msgsock = accept (sock, 0, 0) ) < 0 ) {
        printf("Network server accept failed %d\n", errno);
        perror("network server accept");
        close (sock);
        exit(5);
    else { /* processing of the call comes here) */ }
```

Communicating: `read()` and `write()`

```
int read(int sockfd, char *buf, unsigned int nbytes)
int write(int sockfd, char *buf, unsigned int nbytes)
```

Examples:

```
if (write(sock, message, size) < 0 )
{ printf(": error writing to remote host");
  exit(-1);
}
```

```
bytes_received = read(sock, reply, sizeof(reply));
```

Client communication

```
if ( ( connect ( sock, &srvr, sizeof (srvr) ) ) < 0 ) {
    perror ("network client on connect");
    close (sock);
    exit(4);
}
/* Exchange data with server. Clear buffer bytes first. */

bzero ( buf, sizeof( buf) ); /* zero buffer, BSD. */
strcpy ( buf, "Hello server, please respond to client!" );
write ( sock, buf, sizeof(buf) );

/* Now read message sent back by server. */

if ( ( cnt = read (sock, buf, sizeof(buf) ) ) < 0 ) {
    perror("network client on read");
    close(sock);
    exit(5);
}
else
    printf("I am client and I have received this message %s\n", buf);
    /* Send a message with 0 bytes to end the conversation. */
bzero ( buf, sizeof( buf) ); /* zero buffer, BSD. */
write ( sock, buf, 0 );
close (sock);
exit(0);
}
```

Server communication

```
/* Server communication includes:  
Set up "infinite loop" to listen for clients */  
  
while (1) {  
    if ( ( msgsock = accept (sock, 0, 0 ) ) < 0 ) {  
        perror("network server accept");  
        close (sock);  
        exit(5);  
    }  
  
    /* Fork child process to handle client service request */  
  
    if ( ( fork() ) == 0 ) {      /* Child process to handle communication*/  
        int pid;  
        pid = getpid();          /* PID of child process */  
        close (sock);           /* Do not need listen socket in child. */  
  
        /* Find out who the client is. */  
        if ((rc = getpeername( msgsock, &addr, &adrlen )) < 0) {  
            perror("network server getpeername");  
            close(msgsock);  
            exit(6);  
        }  
        printf("\\n\\tnetwork server %d:", pid);  
        printf(" client socket from host %s\\t has port number %d\\n", inet_ntoa ( nptr -> sin_addr ),  
                nptr -> sin_port );
```

```

do {
    bzero(buf, sizeof(buf)); /* zero buf, BSD call. */
    if ((cnt = read (msgsock, buf, sizeof(buf))) < 0) {
        perror("network server read");
        close(msgsock);
        exit(7);
    }
    else
        if (cnt == 0) { //close on 0 bytes received
            printf("\nServer closing client connection\n");
            close (msgsock);
            continue; /* break out of loop */
        }
    else { /*Print the message received from client.

                Send a message back response*/
        printf("Server %d received your message\n",pid);
        printf("The message from client is %s\n", buf);
        bzero (buf, sizeof(buf)); /* zero buf, BSD. */
        strcpy(buf, "\nI am server and this is my message\n");
        write (msgsock, buf, sizeof(buf));
    } /* end of message-print else */

} /* end of do loop statement */

} while (cnt != 0); /* end of do loop*/
exit(0); /* Exit child process */
} /* End of child */

else /* Parent process */
    close(msgsock); /* Parent doesn't need socket. */
} /* end of while */

```

Exchange data between client and server – server's child processing

Assumption:
close the connection if you receive 0 bytes from client



The close() function

When finished using a socket, the socket should be closed:

- `status = close(s);`
 - `status`: 0 if successful, -1 if error
 - `s`: the file descriptor (socket being closed)
- closes a connection (for `SOCK_STREAM`)
- frees up the port to be used by others

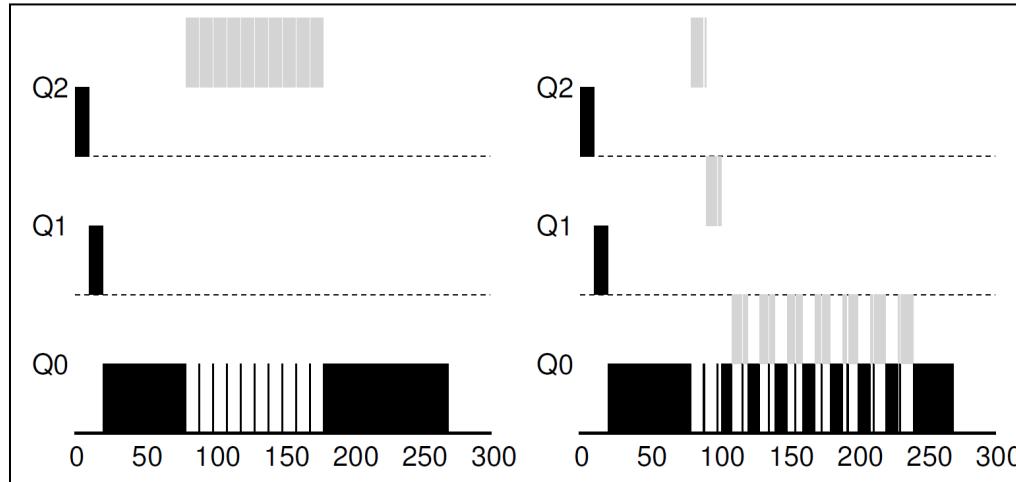
Sample Exam Question

List eight differences between stream and datagram sockets.

- | • Stream Socket | vs | Datagram Socket |
|----------------------------------|----|--------------------------------|
| • uses TCP | | uses UDP |
| • bidirectional | | one way only – send or receive |
| • (synchronous) | | (asynchronous) |
| • stream data (large) | | single packet (small) |
| • connection orientated | | not connection orientated |
| • (persistent connection) | | (single-use connection) |
| • (need to close connection) | | (no connection to close) |
| • in order guaranteed | | no order of data guaranteed |
| • (so message boundaries not ..) | | (message boundaries preserved) |
| • reliable delivery | | unreliable delivery |
| • (acknowledge receipt) | | (do not acknowledge receipt) |
| • more complex | | simpler – less overhead |
| • application: HTTP, FTP | | application: ping |

ENCE360

Operating Systems

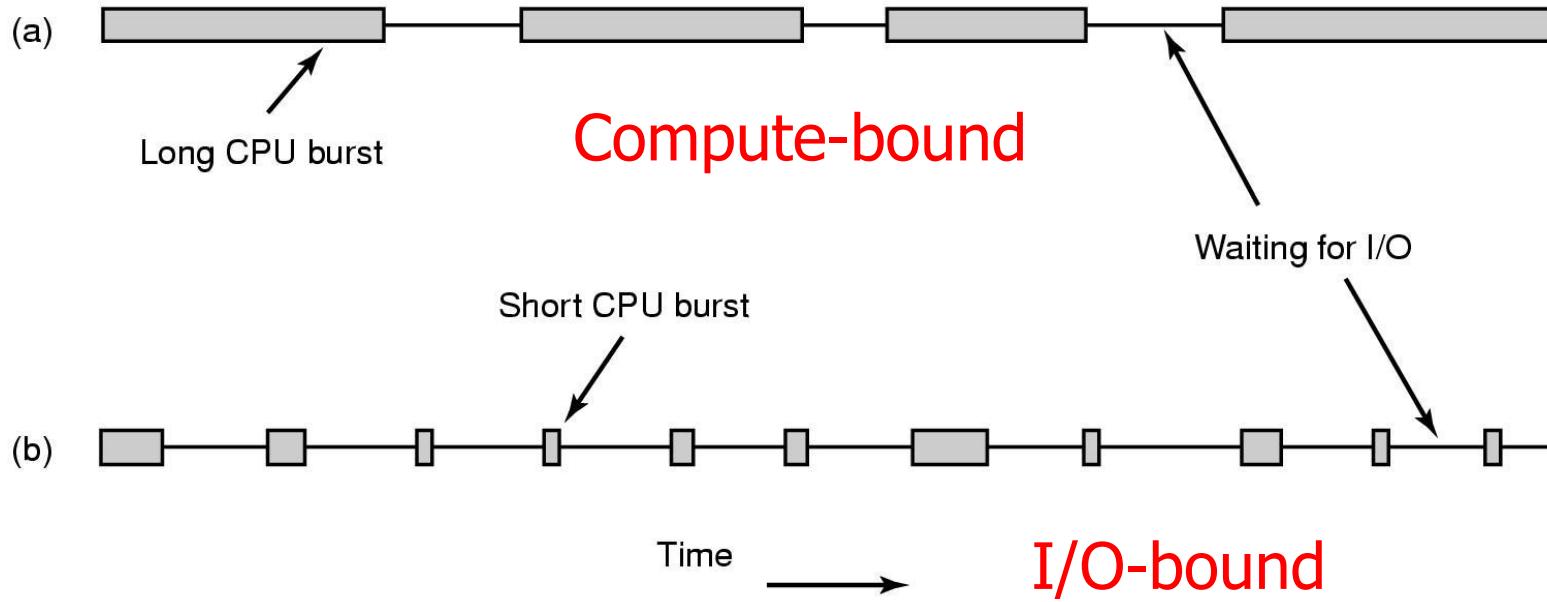


Scheduling

MOS(3) 2.4, 7.5

OSTEP 7, 8, 9

Introduction to Scheduling



- How to schedule a mixture of process behaviours?
- How to best use available resources?
- What are we trying to optimise?
- How to make it fair?

System and process types

Process types

- Nonpreemptive: every job runs to completion
- Preemptive: jobs can be interrupted
 - E.g. on receiving a timer interrupt

System types and their scheduler requirements

- Batch:
 - No users waiting
 - Nonpreemptive or long interruption cycles
- Interactive
 - User waiting (somewhere)
 - User workstations/devices and *servers*
 - Preemption is essential
- Real-time
 - Dominated by periodicity and need to meet deadline
 - Processes designed in advance to work together
 - Can be nonpreemptive or preemptive depending on load

Scheduling goals

All systems

Fairness - giving each process a fair share of the CPU

Policy enforcement - seeing that stated policy is carried out

Balance - keeping all parts of the system busy

Batch systems

Throughput - maximize jobs per hour

Turnaround time - minimize time between submission and termination

CPU utilization - keep the CPU busy all the time

Interactive systems

Response time - respond to requests quickly

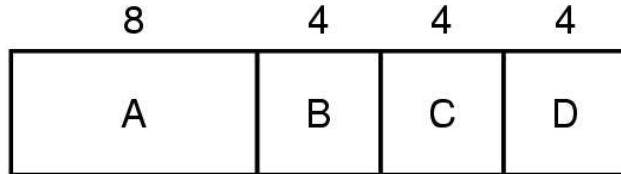
Proportionality - meet users' expectations

Real-time systems

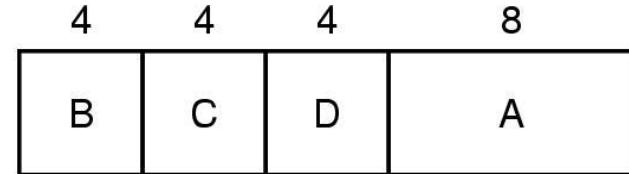
Meeting deadlines - avoid losing data

Predictability - avoid quality degradation in multimedia systems

Scheduling in batch systems



(a)



(b)

First come first served (FCFS)

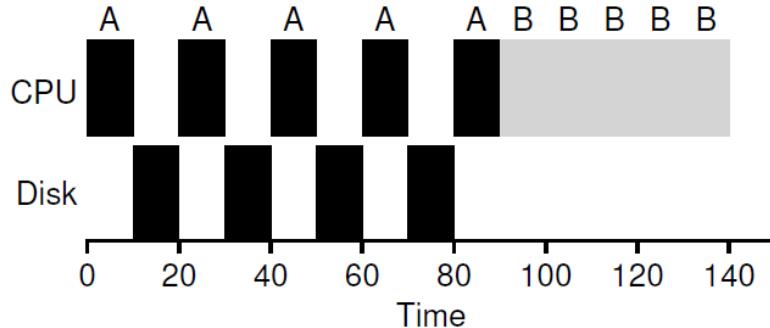
$$T_{\text{complete,avg}} = 14 \text{ mins}$$

Shortest job first (SJF)

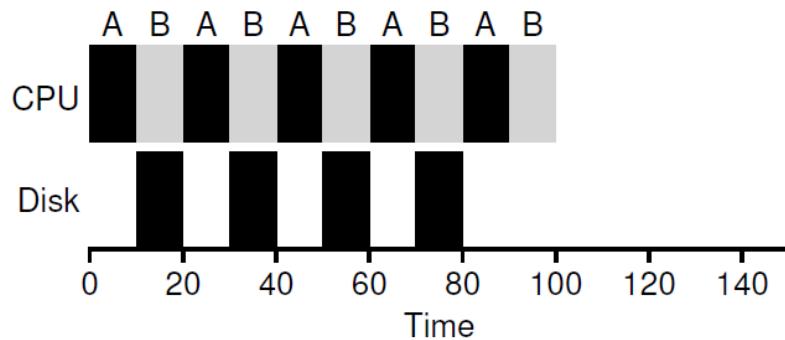
$$T_{\text{complete,avg}} = 11 \text{ mins}$$

- Requires knowledge of job lengths
- Nonpreemptive
 - Assumes (some) jobs arrive at the same time so can choose
- Preemptive: shortest time to completion next
 - Allows new short jobs to interrupt longer ones

Incorporating I/O



Process scheduling
(B has come later)



CPU burst scheduling

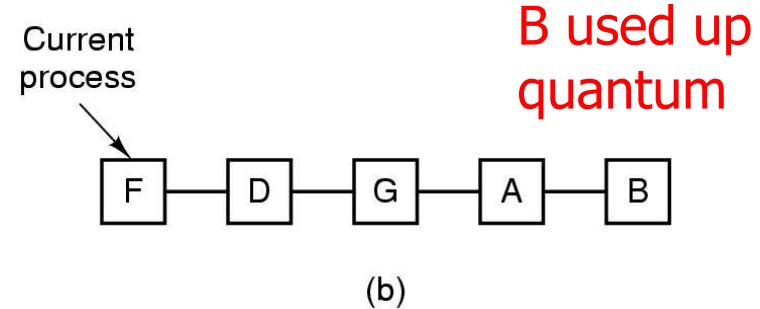
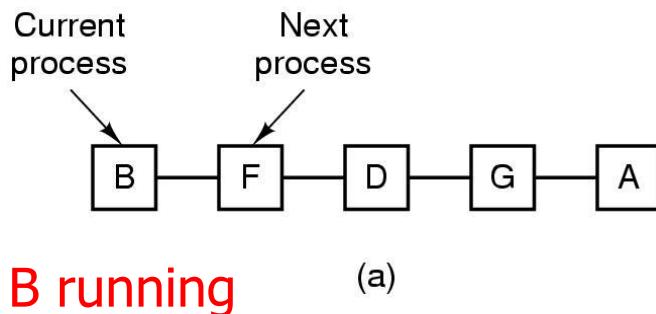
- Each CPU burst treated as a new job
- Long job fills the I/O “holes” of short job

Interactive schedulers

Complex! Many schemes:

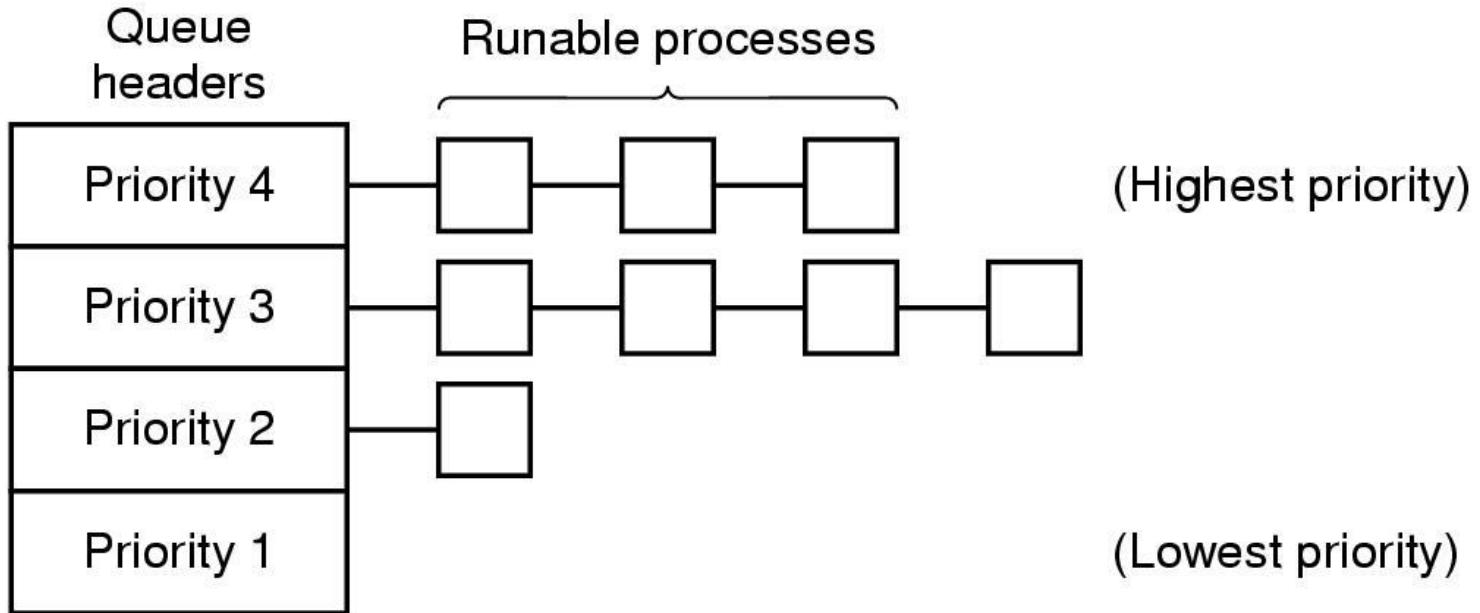
- Round-robin scheduling
- Priority scheduling with multiple queues
- Shortest process next
- Guaranteed scheduling
- Lottery scheduling
- Fair-share scheduling

Round robin scheduling



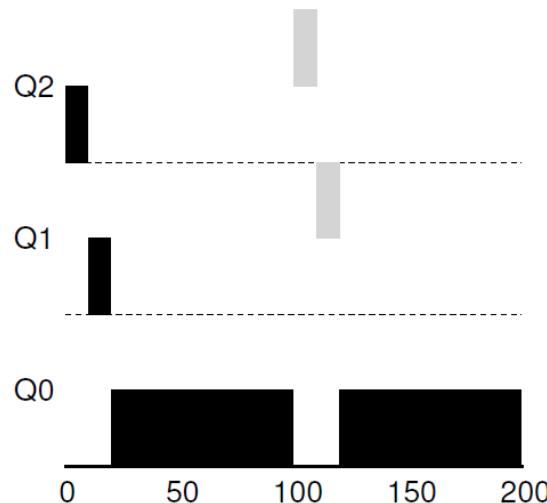
- Preemptive switch every m msecs
- Assumes *equal importance*
 - Non-optimal for mixture of process types
- Can be dominated by process switching time
 - How to decide length?

Multiple queue priority scheduling

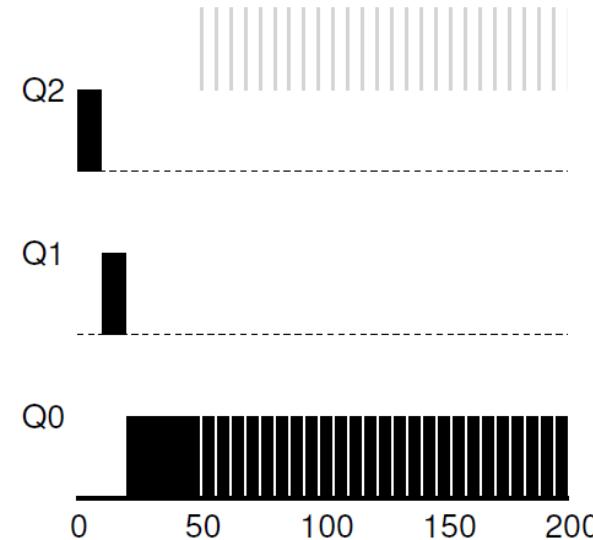


- Each queue only runs when higher empty
- Round-robin scheduling of each level
- Lower queues can starve (never run)
 - Answer: dynamic priority allocation

Dynamic scheduling



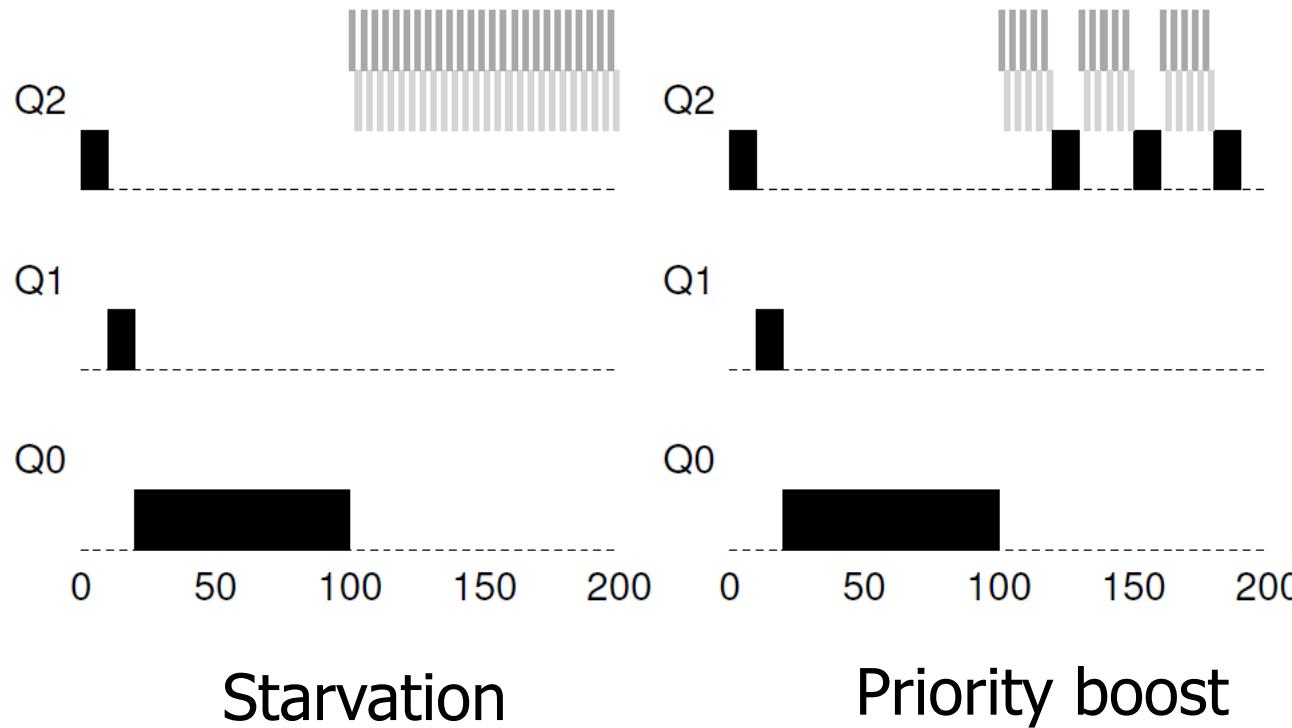
Rule 2:
Black = long batch job
Grey = interactive CPU job



Rule 3:
Black = long batch job
Grey = interactive I/O job

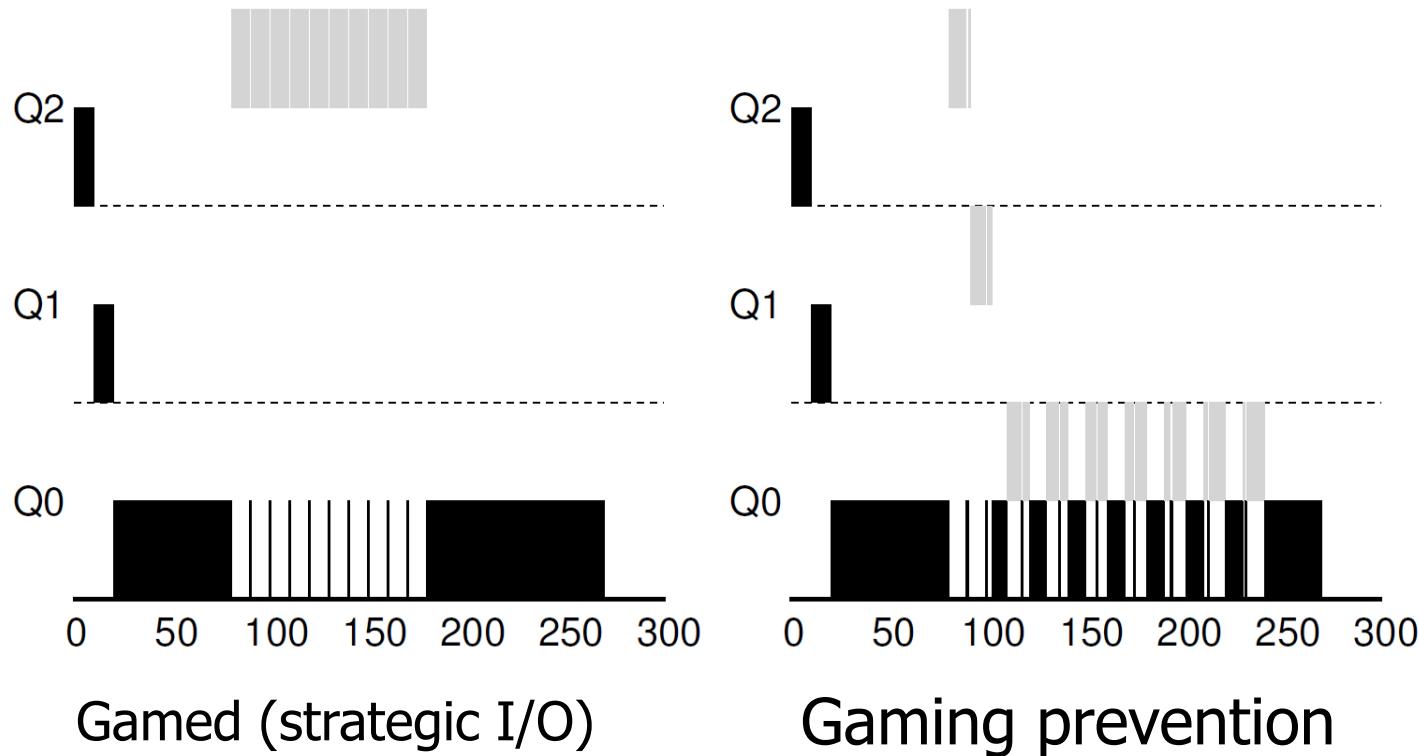
1. New jobs start with highest priority
2. Job using up its time slice downgraded
3. Job *not* using time slice stays the same

Problem 1: starvation



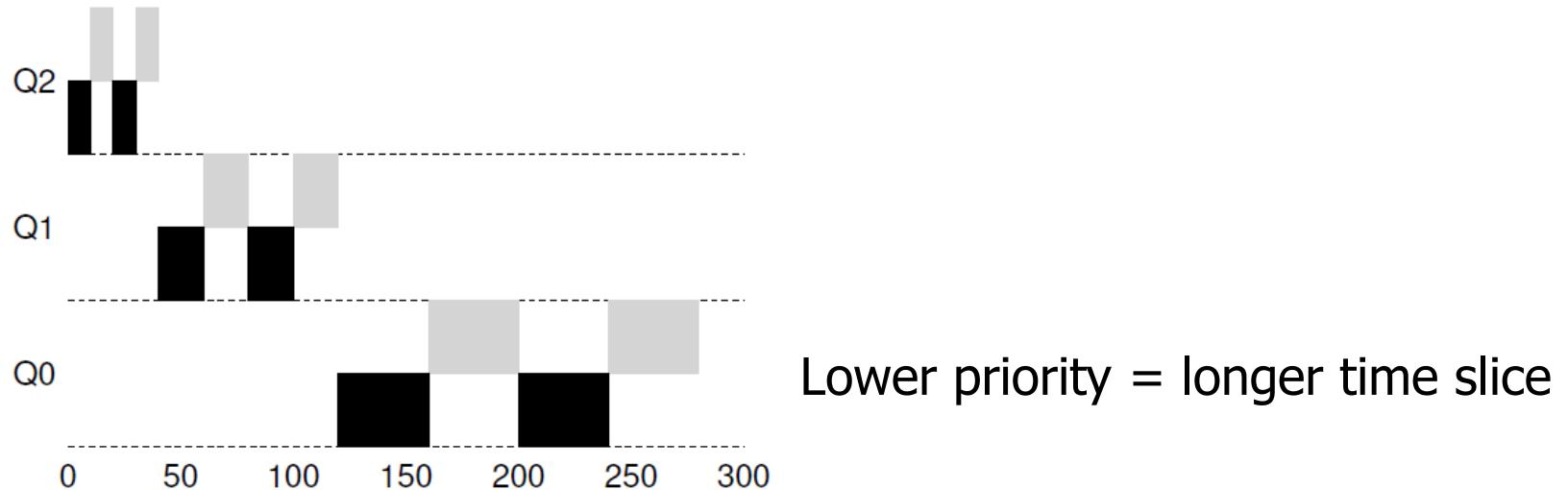
4. Priority boost rule: after n msecs, reset all processes to *highest* priority

Problem 2: gaming the system



- Replace rules 2 and 3 with:
Downgrade a process after m msec of CPU
 - Regardless of number of slices taken
 - Neutralises strategic I/O

Further tuning strategies

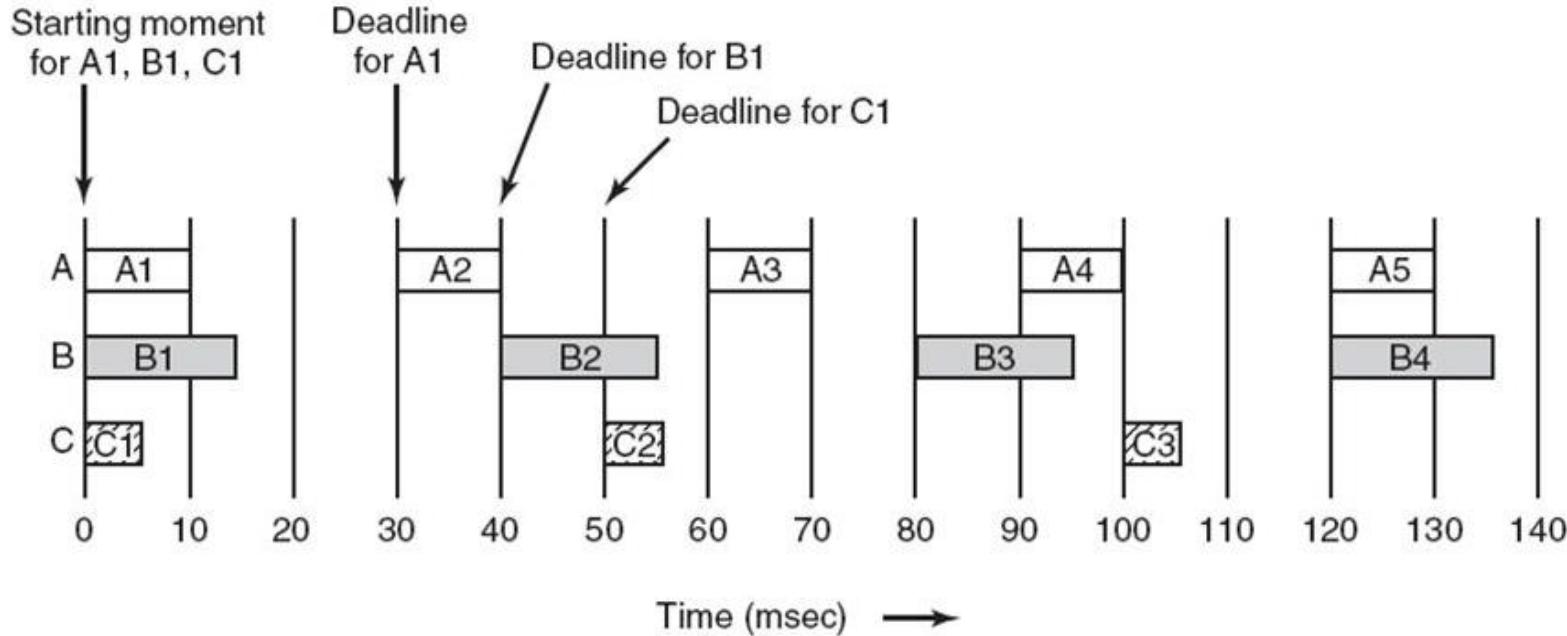


- Increase time period for lower priority jobs (CPU bound, e.g. batch jobs)
- Calculate *ad hoc* priority based on past job times
 - E.g. weighted sum decaying over time, e.g.
 - $T_{est_2} = (T_{est_0} + T_1)/2$
- User hints (e.g. “nice” on Linux/Unix)
- Scheduler parameters (system and user)

Schedule fairness

- Allocate CPU time per *user*, not per process
- Track CPU usage and schedule accordingly
- Lottery scheduling:
 - Each user given $1/n$ tickets
(priority adjusted)
 - Distribute amongst processes
 - Randomly select a ticket and schedule the owning process
 - Tickets can be given away
 - e.g. client to server process

Real-time scheduling



- Example: multimedia server
 - Different periodicity (deadline) and processing time requirements
 - How to schedule so deadlines always met?

Minimum condition

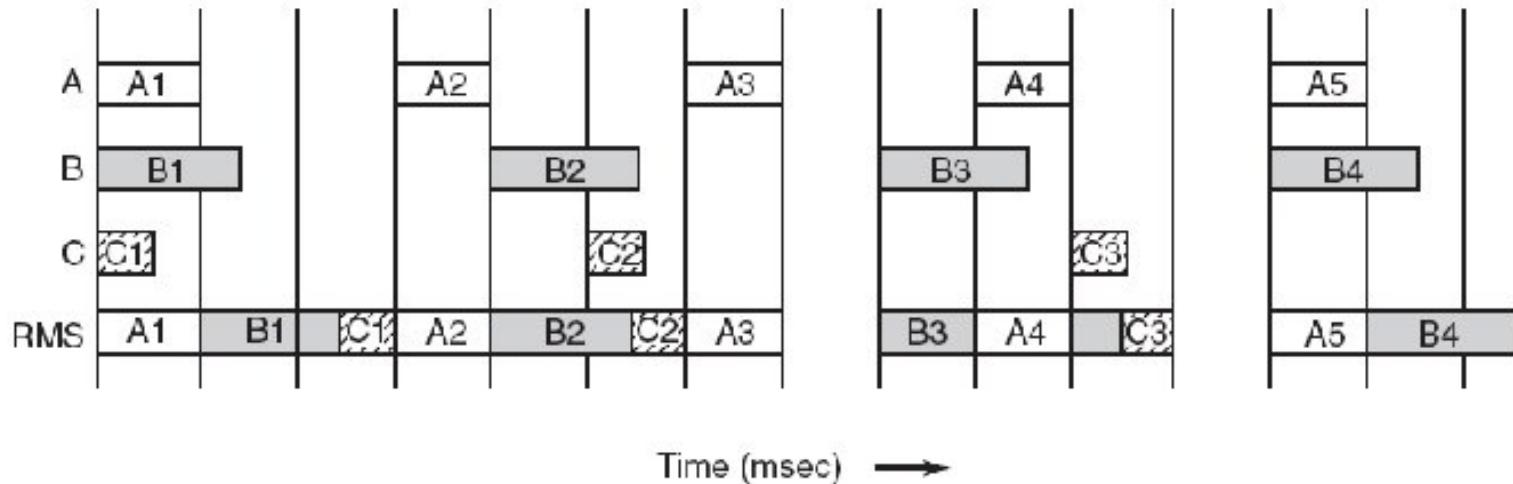
Schedulable real-time system

- Given
 - m periodic events
 - event i occurs within period P_i and requires C_i seconds
- Then the load can only be handled if

$$\sum_{i=1}^m \frac{C_i}{P_i} \leq 1$$

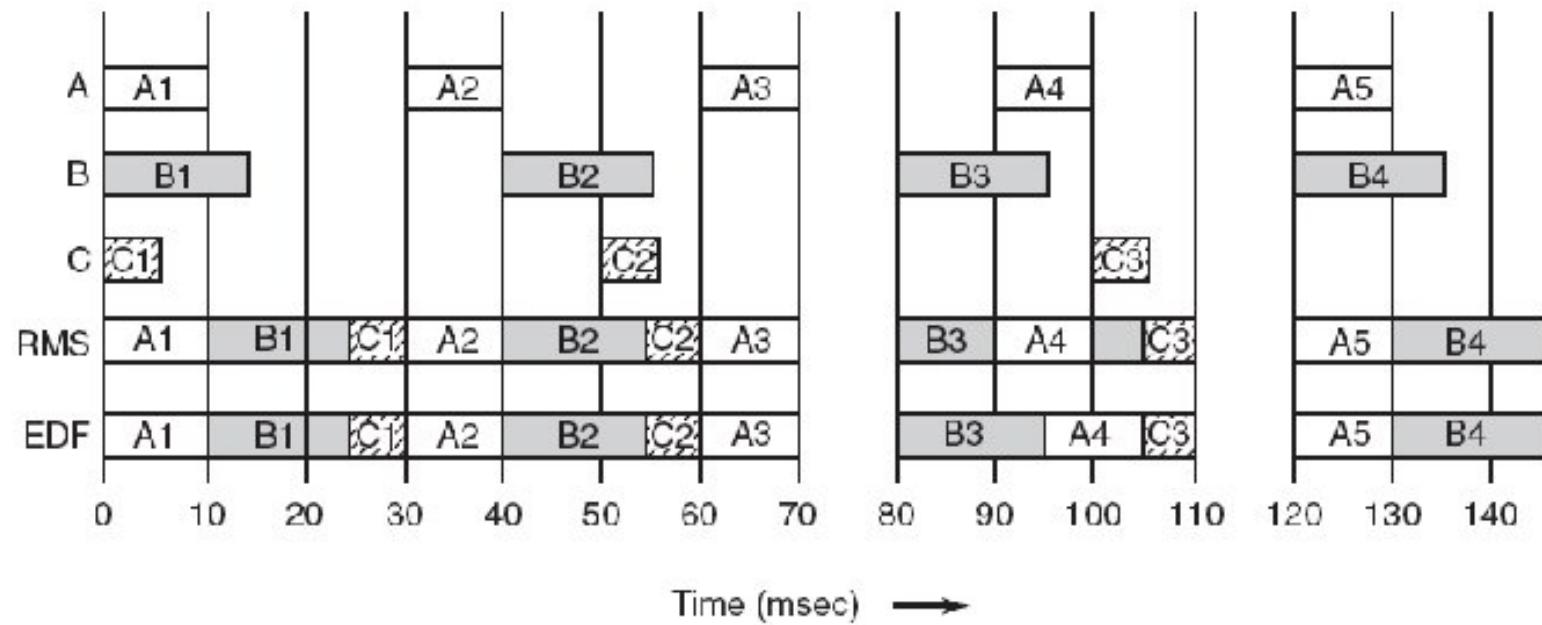
- (i.e. if all work can be completed in a cycle on average) – *necessary condition*

Rate Monotonic scheduling



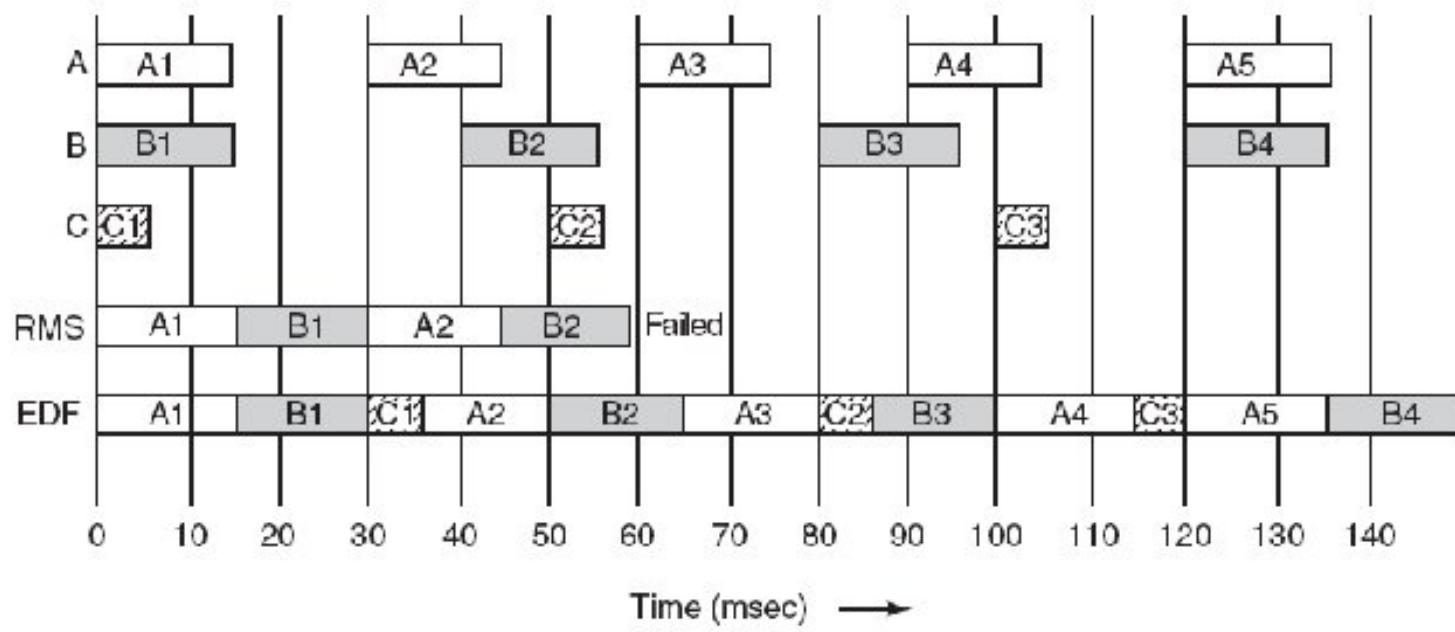
- Priority = $1/\text{deadline frequency}$ (**constant**)
- Always run highest priority process, interrupt current process if needed
- Strict conditions:
 - Each process completes within period
 - No inter-process dependencies
 - Same CPU time per burst
 - No other deadlines
 - Process preemption is instantaneous

Earliest deadline first



- Process with earlier deadline preempts the current process
 - Relaxes the constant frequency requirement

Effect of CPU load



- RMS is load-dependent: guaranteed to work (*sufficient condition*) if:

$$\sum_{i=1}^m \frac{C_i}{P_i} \leq m(2^{\frac{1}{m}} - 1)$$

- EDF always works, but more complex

Policy versus Mechanism

- Separate what is allowed to be done with how it is done
 - a process knows which of its children threads are important and need priority
- Scheduling algorithm parameterized
 - mechanism in the kernel
- Parameters filled in by user processes
 - policy set by user process

ENCE360

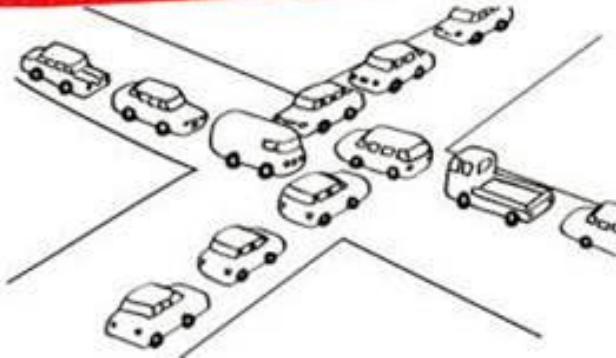
Operating Systems

Deadlocks

Introduction
The ostrich algorithm
Detection and recovery
Deadlock avoidance
Deadlock prevention



DEADLOCK



- Formal definition:
A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause
- Usually the event is release of an exclusively held *resource*
- None of the processes can ...
 - run
 - release resources
 - be awakened

Resources

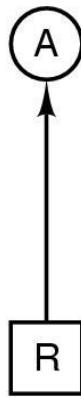
- Various sorts:
 - Physical devices: printers, tape drives
 - Database/data structures: tables
 - *Locks (e.g. semaphores)*
- Sequence of events: request, use, release
- Must wait if request is denied
 - requesting process may block
 - may fail with error code (and retry)
- Two types:
 - Preemptable: can be taken away from a process with no ill effects
 - Nonpreemptable: will cause the process to fail if taken away

Four Conditions for Deadlock

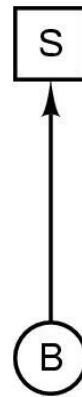
- Mutual exclusion condition
 - each resource assigned to *one* process or is available
- Hold and wait condition
 - process holding resources can request additional ones
- No preemption condition
 - previously granted resources cannot forcibly taken away
- Circular wait condition
 - must be a circular chain of 2 or more processes
 - each is waiting for resource held by next member of the chain

Deadlock Modeling

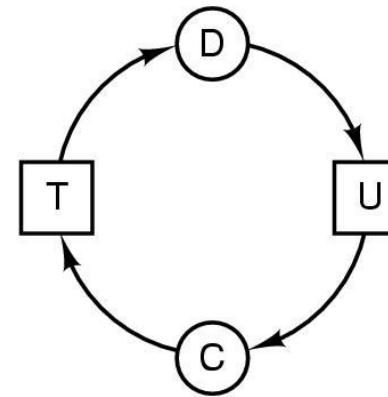
- Modeled with directed graphs



(a)



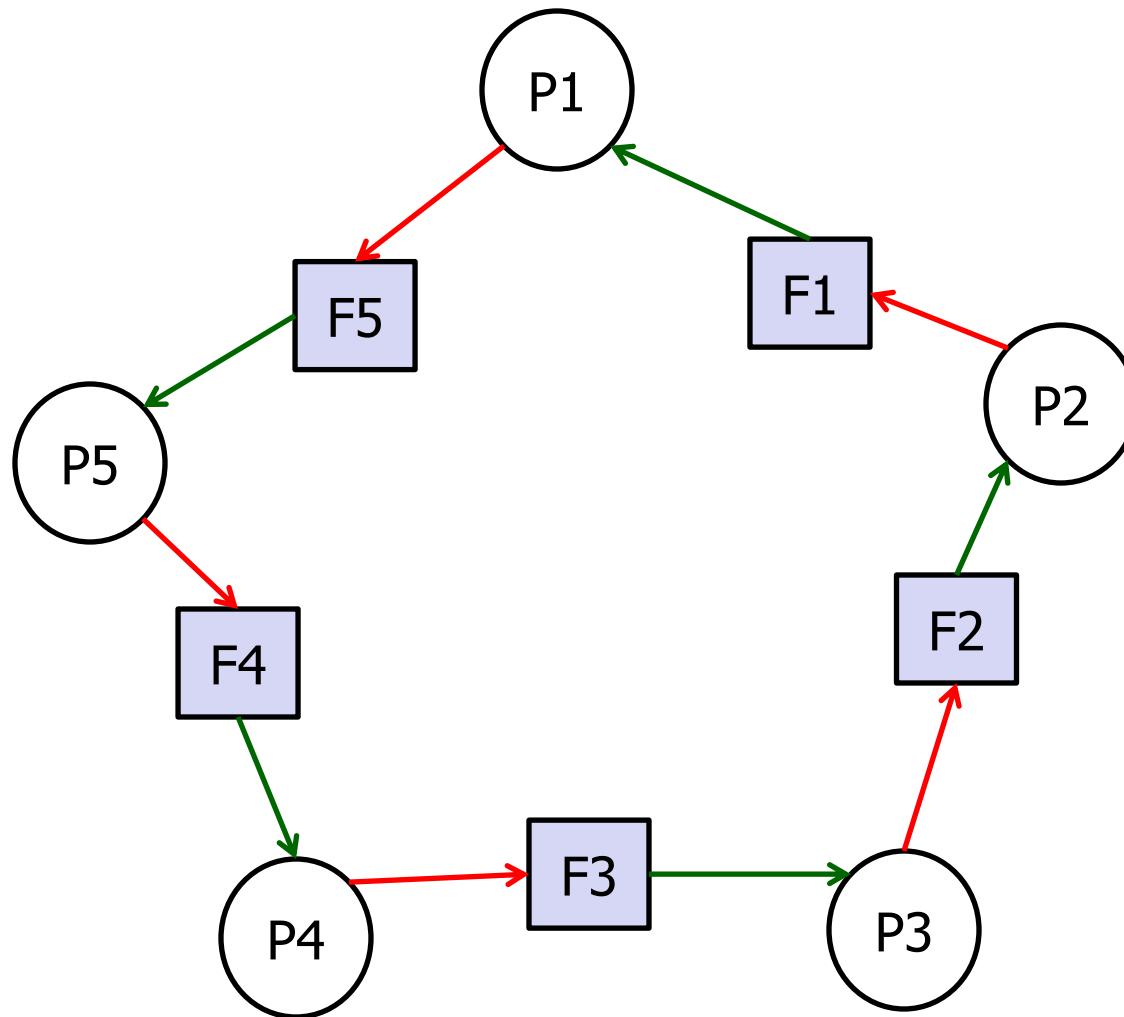
(b)



(c)

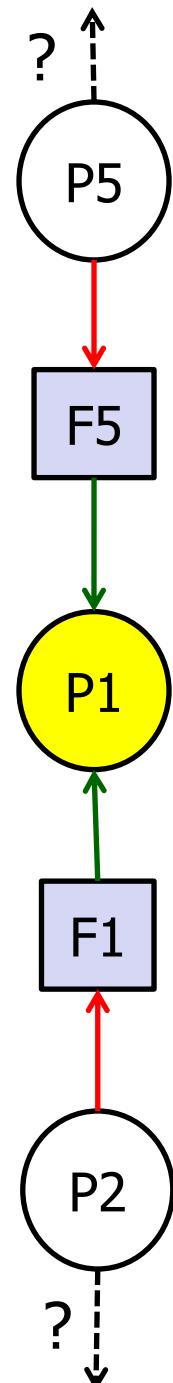
- resource R assigned to process A
- process B is requesting/waiting for resource S
- process C and D are in deadlock over resources T and U

Deadlocked philosophers



Philosophers solution

- Do non resource-dependent work (think)
- Get resources (forks):
 - In critical region:
 - Indicate need for resources
 - Check whether resources available (and signal)
 - Block until signalled that resources available
- Do resource-dependent work (eat)
- Release resources (forks)
 - In critical region:
 - Indicate resources no longer required
 - Check whether resources available
for dependent processes (and signal)
- Relies on cooperation between processes
 - Not feasible at operating system level



Tracing a deadlock

A

Request R
Request S
Release R
Release S

(a)

B

Request S
Request T
Release S
Release T

(b)

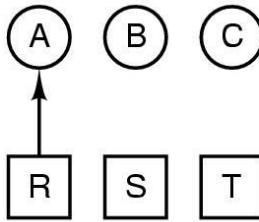
C

Request T
Request R
Release T
Release R

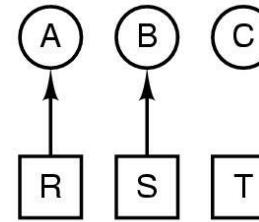
(c)

1. A requests R
2. B requests S
3. C requests T
4. A requests S
5. B requests T
6. C requests R
- deadlock

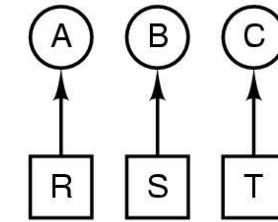
(d)



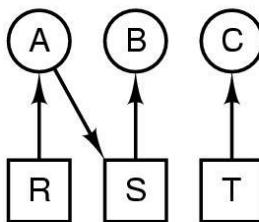
(e)



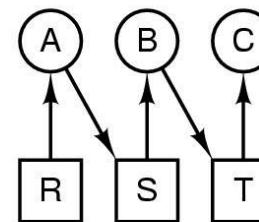
(f)



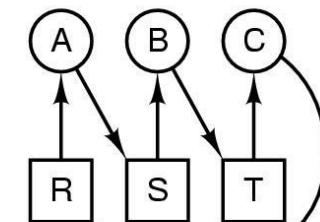
(g)



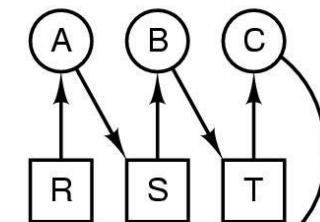
(h)



(i)



(j)



Circular dependency = deadlock

Deadlock Strategies

- Ostrich algorithm
 - ignore the problem
- Detection and recovery
 - take action when detected
- Dynamic avoidance
 - Avoidance by scheduling
 - Careful resource allocation
- Prevention
 - negate one of the four necessary conditions

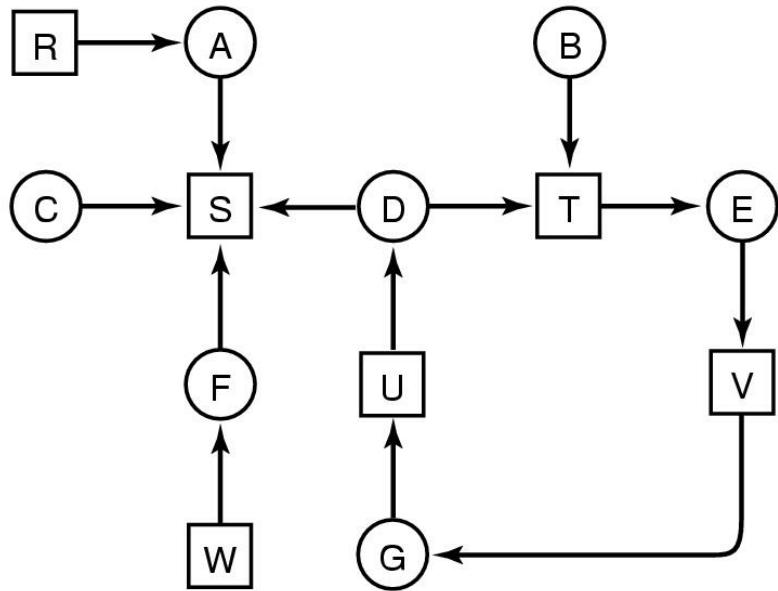
The Ostrich Algorithm

- Pretend there is no problem
- Reasonable if
 - deadlocks occur very rarely
 - cost of prevention is high
- UNIX and Windows take this approach
- Trade off between
 - Frequency and seriousness of deadlock
 - Cost/difficulty avoiding

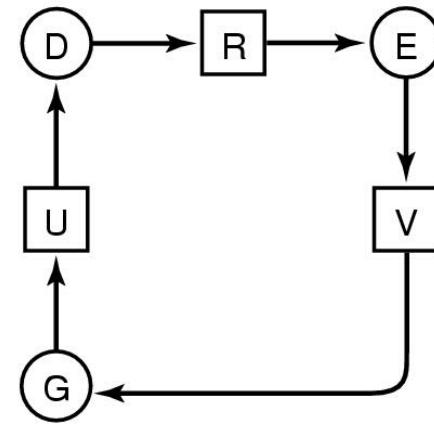
Deadlock detection

- Periodically checks resources held and requested to look for impasses
 - When resources requested
 - Every n minutes
 - When CPU usage stalls
- Takes steps to recover from deadlocks if detected
- Practical limitations
 - Requires visibility of process resource requirements
 - Usually known to the resource scheduler
 - Space-time expensive: order ($\text{processes} \times \text{resources}$)
 - Resolving deadlocks can be difficult/expensive

Deadlock detection



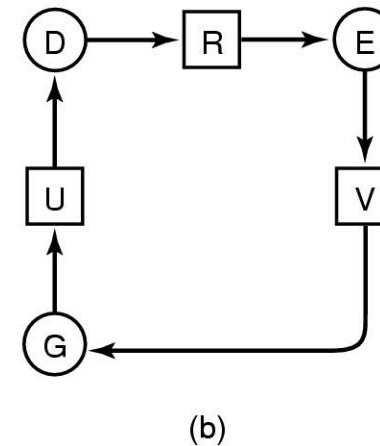
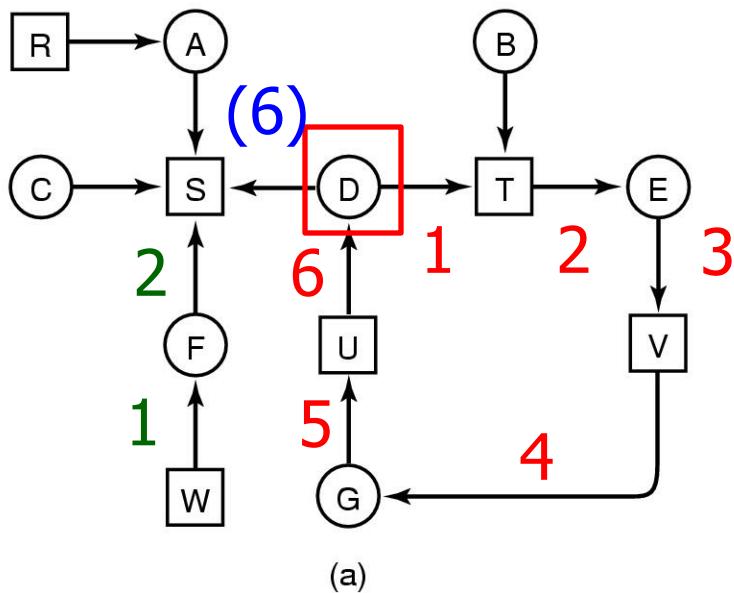
(a)



(b)

- Track the resource ownership and requests
- If a cycle exists within the graph, DEADLOCKED

Deadlock detection: one of each resource



- For each node N:
 - Initialise L to [], all outgoing arcs to unvisited
 - Add N to L. If N appears twice, **DEADLOCK: STOP**;
 - If unvisited *outgoing* arc: follow to next node
 - Else back up to parent

Detection: multiple resources

Resources in existence
($E_1, E_2, E_3, \dots, E_m$)

Resources available
($A_1, A_2, A_3, \dots, A_m$)

Current allocation matrix

C_{11}	C_{12}	C_{13}	\dots	C_{1m}
C_{21}	C_{22}	C_{23}	\dots	C_{2m}
:	:	:		:
C_{n1}	C_{n2}	C_{n3}	\dots	C_{nm}

Row n is current allocation to process n

Request matrix

R_{11}	R_{12}	R_{13}	\dots	R_{1m}
R_{21}	R_{22}	R_{23}	\dots	R_{2m}
:	:	:		:
R_{n1}	R_{n2}	R_{n3}	\dots	R_{nm}

Row 2 is what process 2 needs

- Checks current requests against available resources
- A process P_i can run if $R_i \leq A$
- No deadlock if *all* requests can complete

Detection example: multiple resources

Tape drives
Plotters
Scanners
CD Roms

$$E = (4 \quad 2 \quad 3 \quad 1)$$

Resources in existence

Tape drives
Plotters
Scanners
CD Roms

$$A = (2 \quad 1 \quad 0 \quad 0)$$

Resources available

Current allocation matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Request matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

- P3 can run: $A = (2 \ 2 \ 2 \ 0)$
- Then P2 can run: $A = (4 \ 2 \ 2 \ 1)$
- Then P1 can run
- *What if P2 also needs a CD ROM?*

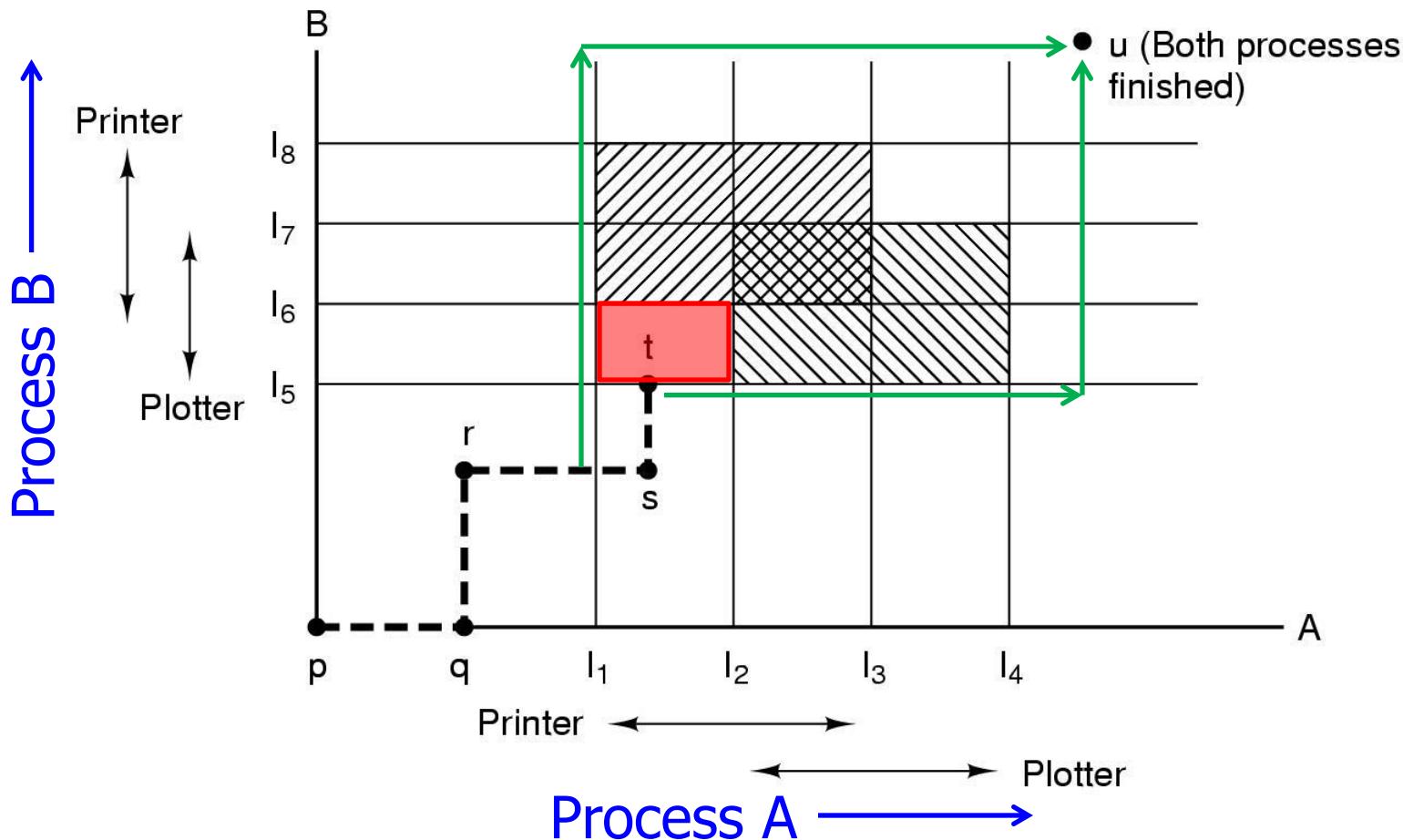
Recovery from Deadlock

- Preemption: take a resource from another process
 - depends on nature of the resource
- Rollback
 - checkpoint a process periodically
 - use this saved state
 - restart the process if it is found deadlocked
- Recovery through killing processes
 - kill one of the processes in the deadlock cycle *or another resource holder*
 - the other processes get its resources
 - choose process that can be rerun from the beginning

Deadlock avoidance

- Tries to predict when deadlocks may occur
- Checks required resources requested to determine how to avoid deadlocks:
 - Process scheduling: suspend one or more processes to avoid resource clash
 - Resource scheduling: grant resources in a manner that avoids *unsafe states*
- Strong assumptions make implementation difficult/impractical
 - Requires visibility of process *future* resource requirements (static scheduling)
 - Assumes resources requested will be *held to completion* (most pessimistic outlook)

Resource state trajectories

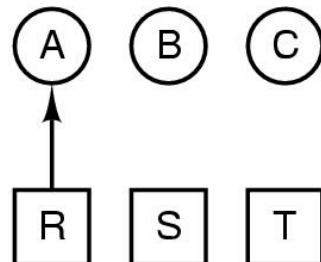


- Shaded areas: both processes have same resource (**not possible**)
- Need to avoid states that can't avoid the shaded areas (**dead ends**)

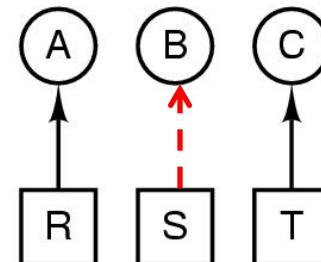
Brute force: process scheduling

1. A requests R
2. C requests T
3. A requests S
4. C requests R
5. A releases R
6. A releases S
- no deadlock

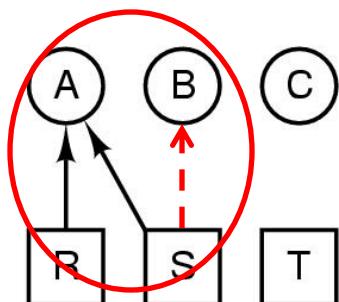
(k)



(l)

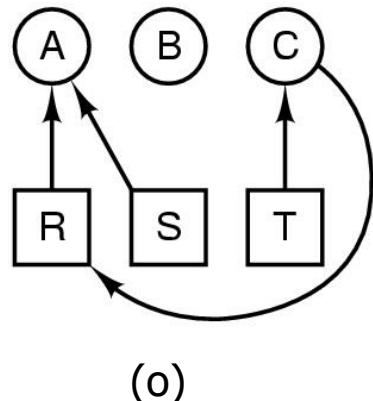


(m)

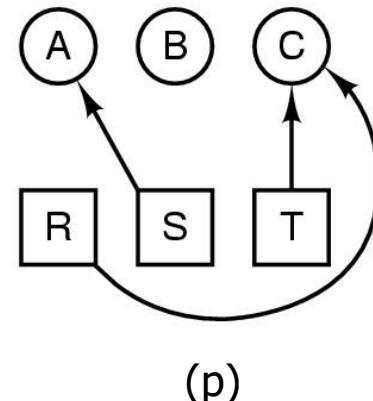


(n)

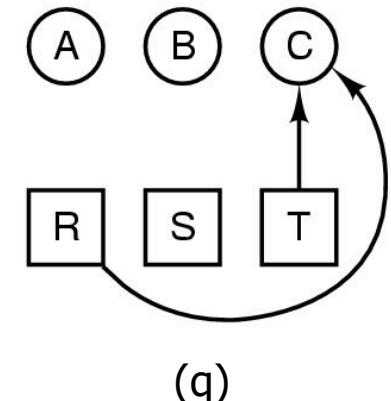
*Potential
deadlock
detected:
B suspended*



(o)

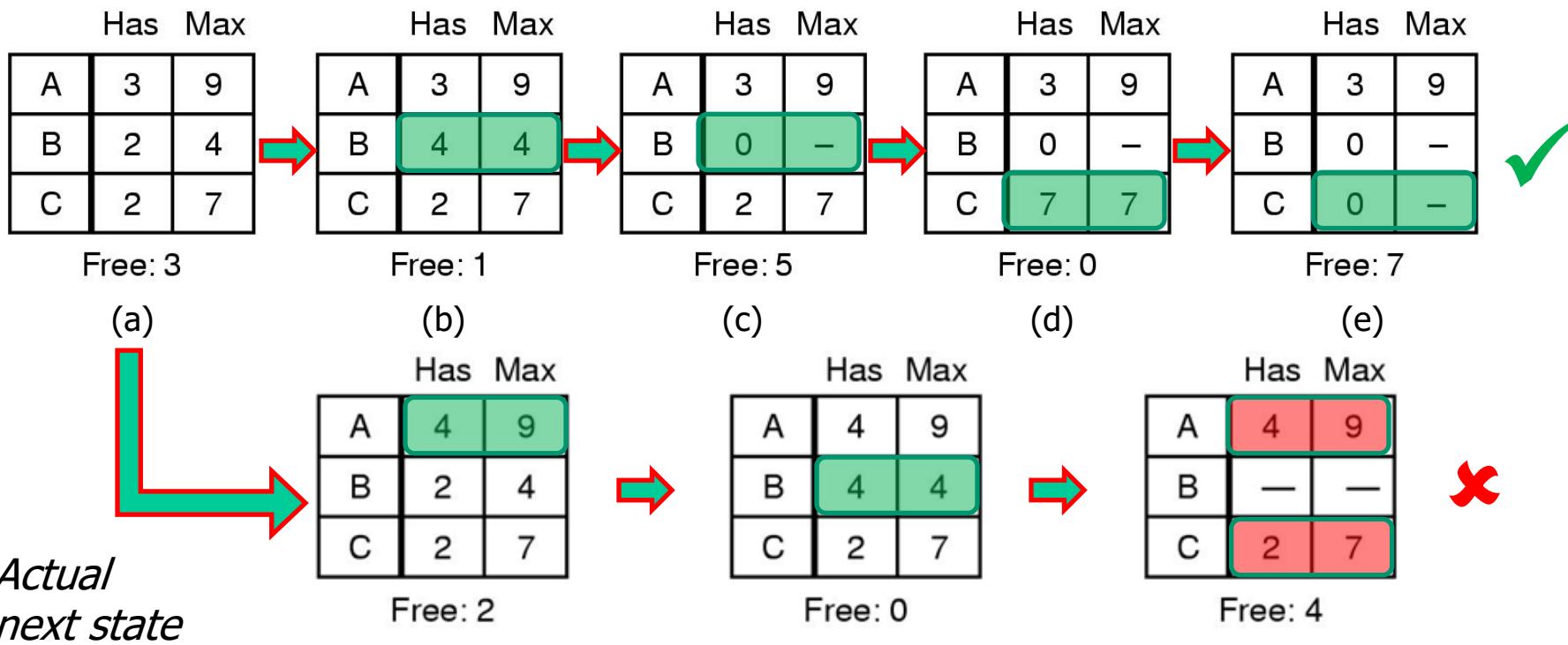


(p)



(q)

Safe and unsafe states



- Assume processes will grab and hold *requested* resources from this point until completed
- Safe states *guaranteed* to be able to complete
- Unsafe states *may not* be able to complete (avoid)

The Banker's algorithm for a single resource (Dijkstra 1965)

	Has	Max
A	0	6
B	0	5
C	0	4
D	0	7

Free: 10

Safe

	Has	Max
A	1	6
B	1	5
C	2	4
D	4	7

Free: 2

Safe

	Has	Max
A	1	6
B	2	5
C	2	4
D	4	7

Free: 1

Unsafe

- Decide which requests to grant based on potential for deadlock
 - Look for a row r whose needs \leq available. If none, **UNSAFE**
 - Remove r and add its resources to $free$
 - Repeat until all complete (**SAFE**)
- Grant request that leads to a safe state (if any) first

Banker's algorithm for multiple resources

Process
Tape drives
Plotters
Scanners
CD ROMs

A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

Resources assigned

Process
Tape drives
Plotters
Scanners
CD ROMs

A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

Resources still needed

$$\begin{aligned} E &= (6342) && \text{Existing} \\ P &= (5322) && \text{Possessed} = \text{Resources} \\ A &= (1020) && \text{Available} = \text{Assigned} \end{aligned}$$

- Same algorithm as detection but traces all possible states to completion
- **Problem: assumptions are unrealistic**
 - Static number of processes
 - Known usage requirements

Prevention: break *one* condition

- Mutual exclusion condition:
 - Remove competition over resource
- Hold and wait condition
 - Don't allow additional resources to be requested
- No preemption condition
 - Allow processes to surrender resources
- Circular wait condition
 - Avoid ordering issues

Deadlock Prevention

Attacking the mutual exclusion condition

- Some devices (such as printer) can be spooled
 - only the printer daemon uses printer resource
 - thus deadlock for printer reduced/eliminated
- Not all devices can be spooled
- Principle:
 - avoid assigning resource when not absolutely necessary (minimise time locked)
 - as few processes as possible actually claim the resource (hand off)

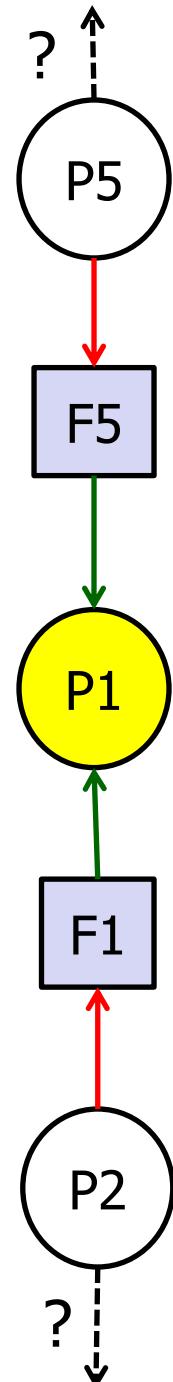
Deadlock Prevention

Attacking the hold and wait condition

- Require processes to request resources before starting
 - a process not allowed to wait for further resources
- Problems
 - may not know required resources at start of run
 - also ties up resources other processes could be using
- Dynamic variation: drop and reacquire
 - Release all currently held resources
 - Request all required (old and new)

Philosophers solution

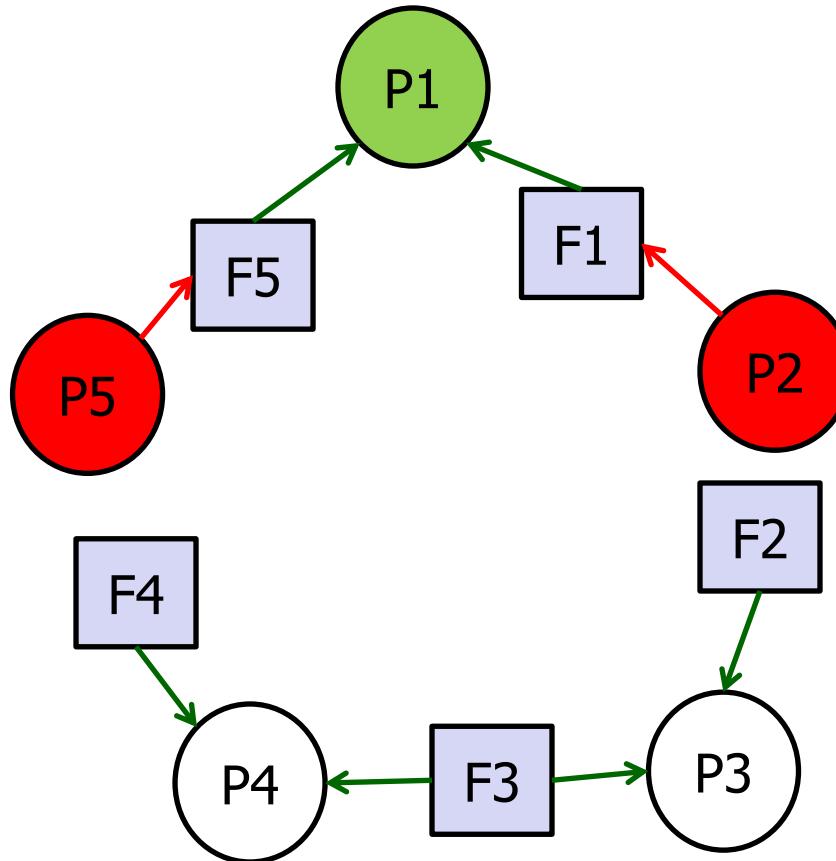
- Do non resource-dependent work (think)
- Get resources (forks):
 - In critical region:
 - Indicate need for resources
 - Check whether resources available (and signal)
 - Block until signalled resources available
- Do resource-dependent work (eat)
- Release resources (forks)
 - In critical region:
 - Indicate resources no longer required
 - Check whether resources available
for dependent processes (and signal)
- Relies on cooperation between processes
 - Not feasible at operating system level



Philosophers solution (2) simpler but less efficient

- Do non resource-dependent work (think)
- Get resources (forks):
 - In critical region:
 - Block until resources acquired
- Do resource-dependent work (eat)
- Release resources (forks)
 - In critical region:
 - Release resources

Hungry philosophers: mutex lock

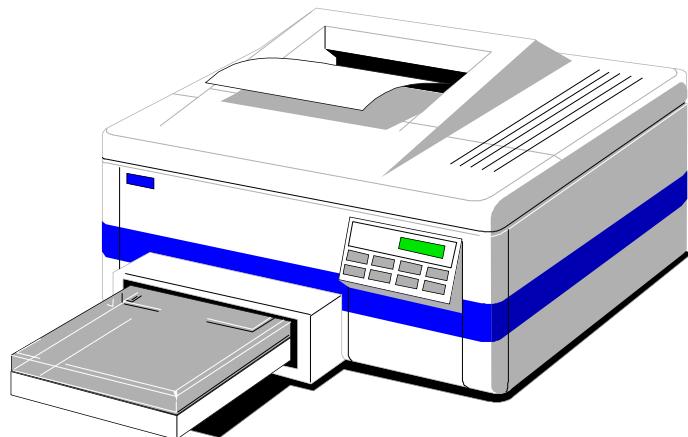


- P1 wins the mutex and grabs the forks
- P2/P5 wins the mutex next but blocks on P1
- P3/P4 could be eating, but blocks waiting for the mutex
- Not deadlocked, but wastes resources

Deadlock Prevention

Attacking the no preemption condition

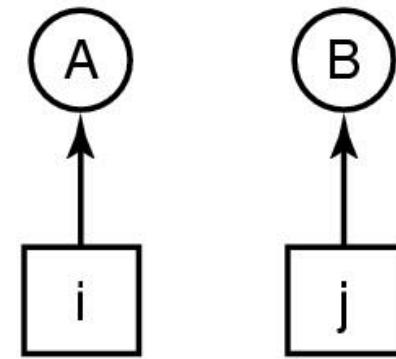
- Not always viable/practical
 - Take a printer away halfway through job?
 - Take database locks away during a transaction?



Deadlock Prevention

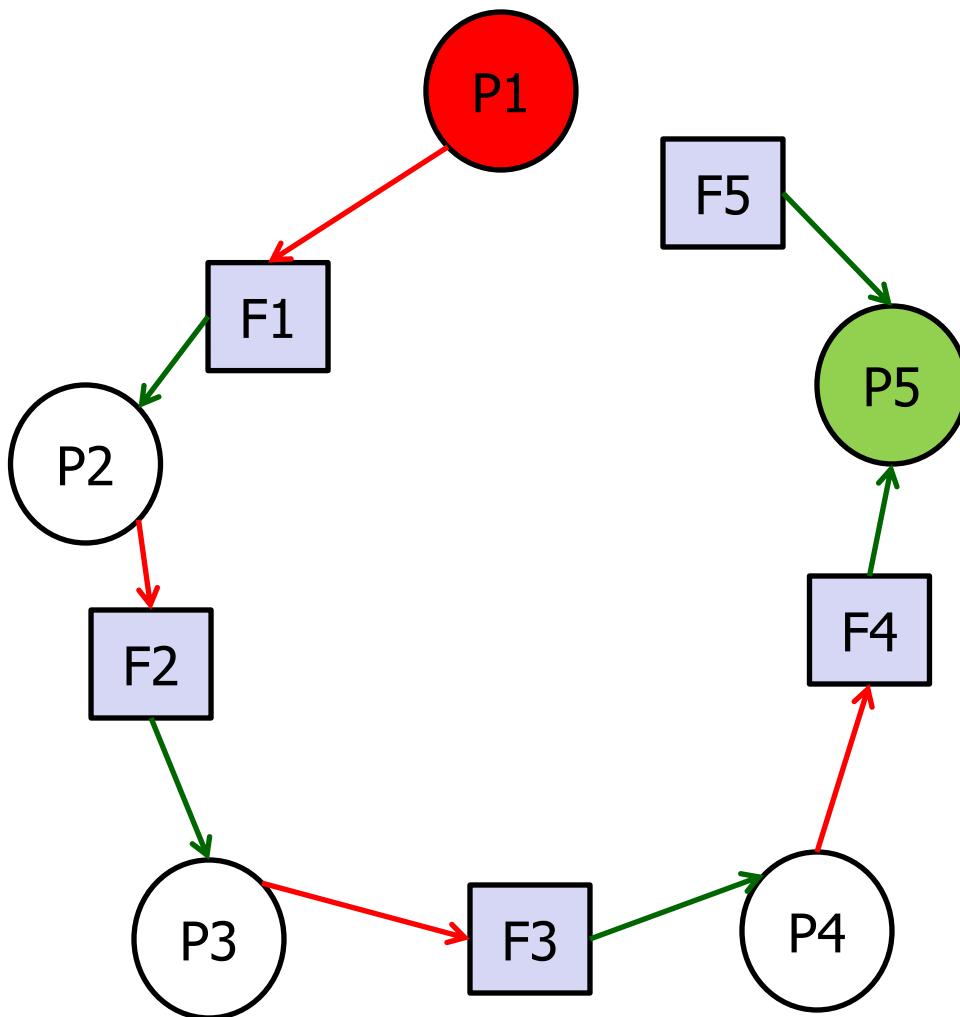
Attacking the circular wait condition

- 1. Imagesetter
- 2. Scanner
- 3. Plotter
- 4. Tape drive
- 5. CD Rom drive



- Strict ordering of resource allocation
 - Not allowed a resource lower than current resource(s)
- Examples:
 - Linux memory mapping code enforces locking partial orderings
 - Database exclusive write locks on tables: order alphabetically by table name to avoid deadlock

Unlocked philosophers – enforced order



- All grab left fork except P1
- P1 must grab right fork and blocks
- P5 gets both forks and eats
- Then P4, P3, P2, then P1
- Deadlock avoided – always a break in the chain

Deadlock prevention - summary

Condition	Approach
Mutual exclusion	Spool everything
Hold and wait	Request all resources initially
No preemption	Take resources away
Circular wait	Order resources numerically

The third way: Two-Phase Locking

- Phase One
 - process tries to lock all records it needs, as required
 - Hold and wait
 - if needed record found locked, start over
 - (no real work done in phase one, or *reversible*)
 - E.g. database run unit journal – delay updates or rollback
- If phase one succeeds, it starts second phase,
 - Perform/commit updates
 - release locks
- Note similarity to requesting all resources at once
- Algorithm works where programmer can arrange
 - program can be stopped, restarted

ENCE360

Operating Systems

File Systems

- **Files and directories**
- **File system implementations**
- **Example file systems**

MOS (3rd ed) Ch 4

The logical file system

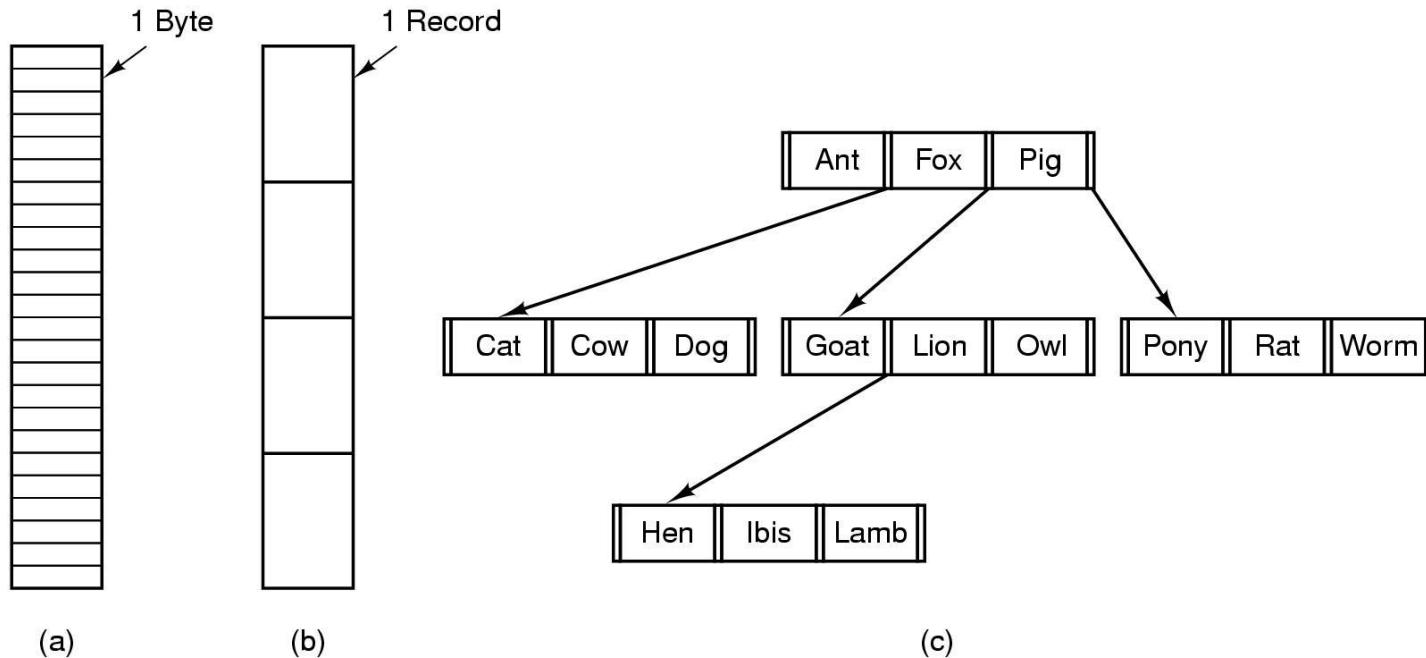
Essential requirements of long-term storage

1. Must store large amounts of data
2. Information stored must survive the termination of the process using it
3. Multiple processes must be able to access the information concurrently

Considerations

1. How do you find the information (file) you want?
 1. Find a file on disk
 2. Find information within a file
2. How do you control data (file) access by users?
3. How do you know which disk areas (blocks) are free?

File structure



- Three kinds of files
 - Byte sequence: no inherent structure
 - Record sequence: fixed length records (not used)
 - Tree: indexed records (some mainframes)

Internal file structures

File Types

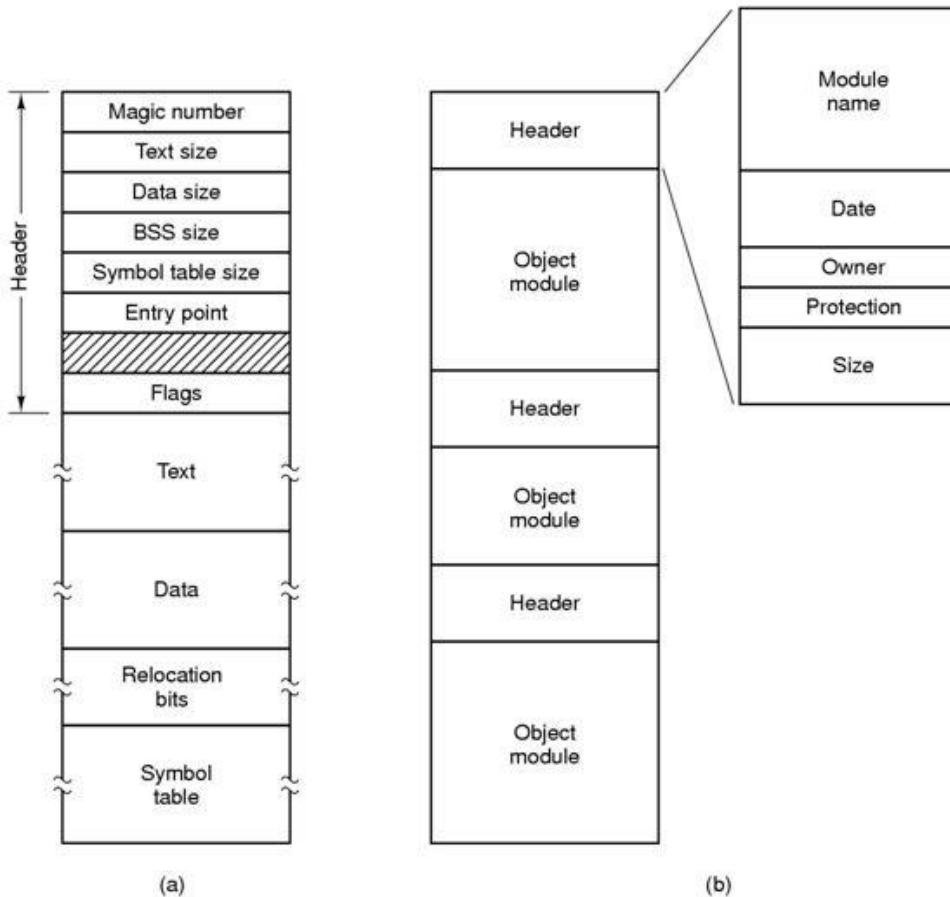


Figure 4-3. (a) An executable file. (b) An archive.

File access

- Sequential access
 - Read all bytes/records from the beginning
 - Convenient when medium was mag tape
 - Still relevant for small files, multimedia
- Random access
 - bytes/records read in any order
 - essential for data base systems, caches, etc
 - Two methods:
 - Read from a specified position
 - move file marker (seek), then read

File operations (API)

- 1. Create
- 2. Delete
- 3. Open
- 4. Close
- 5. Read
- 6. Write
- 7. Append
- 8. Seek
- 9. Get
 attributes
- 10. Set
 Attributes
- 11. Rename

Directories

Path Names

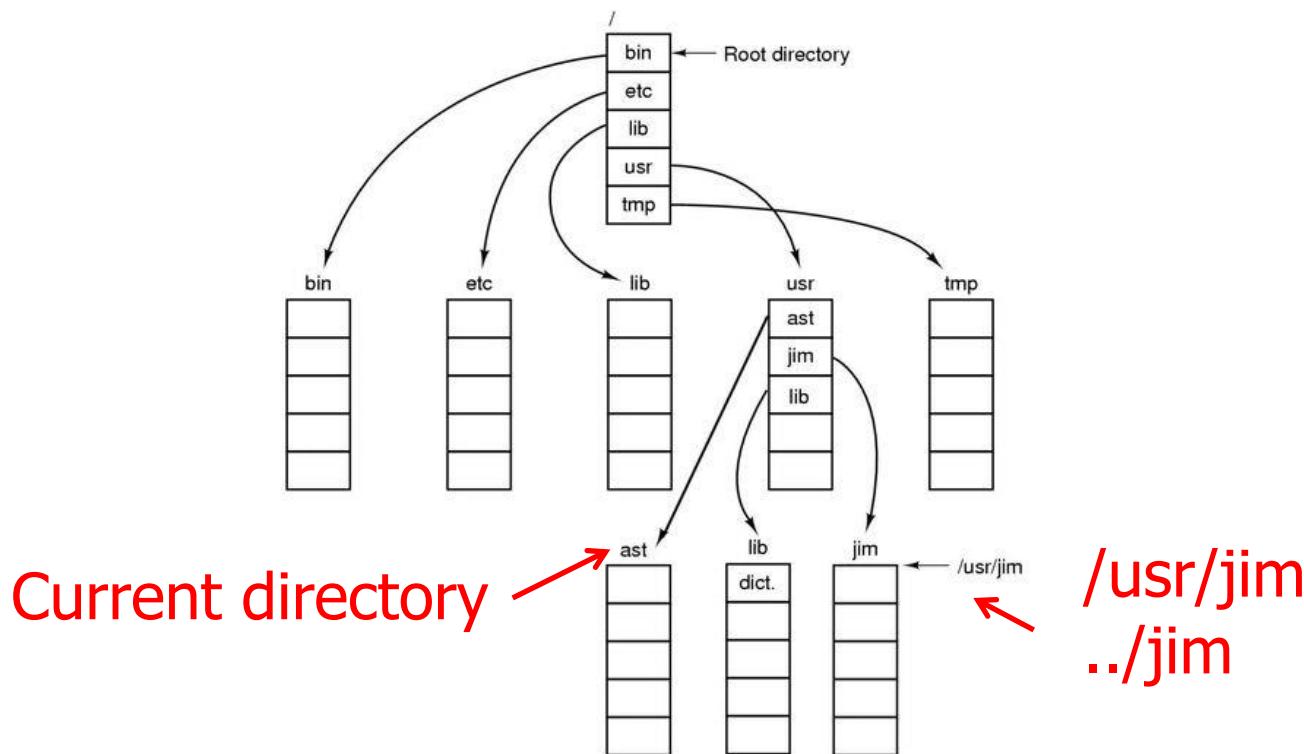


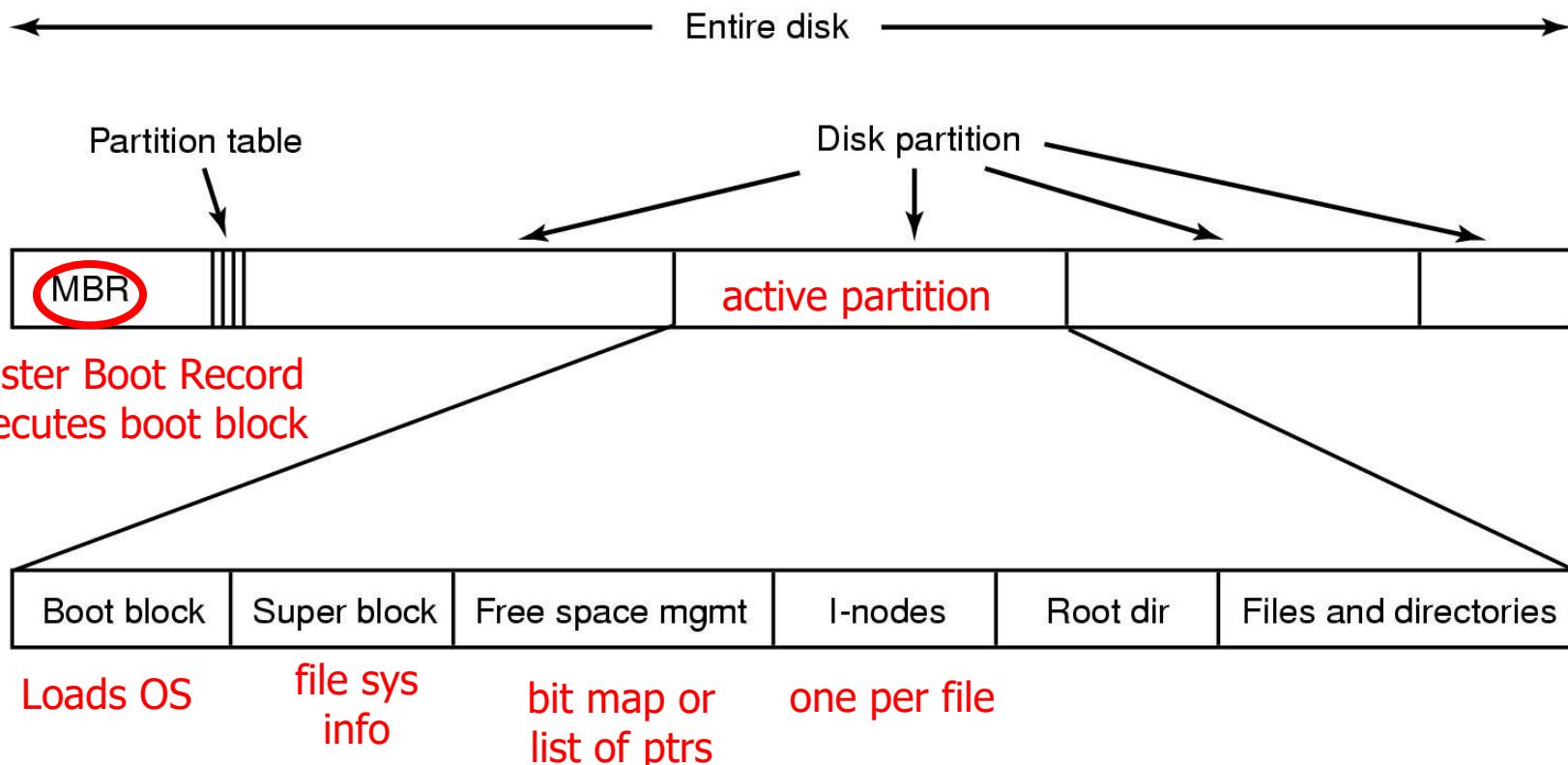
Figure 4-8. A UNIX directory tree.

Directory operations (API)

- 1. Create
- 2. Delete
- 3. Opendir
- 4. Closedir
- 5. Readdir
- 6. (Rename)
- 7. Link
- 8. Unlink

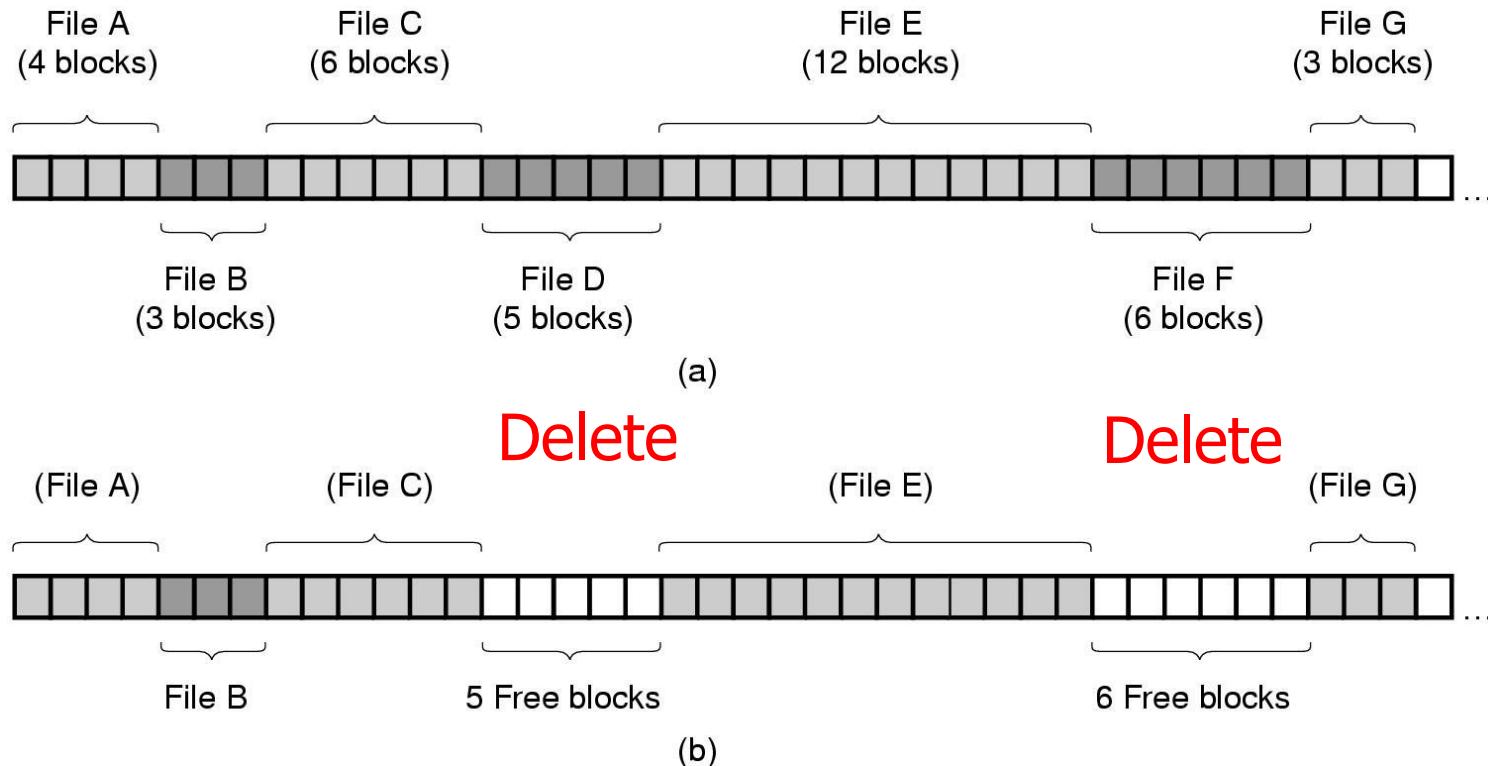
File system implementation

Example file system layout



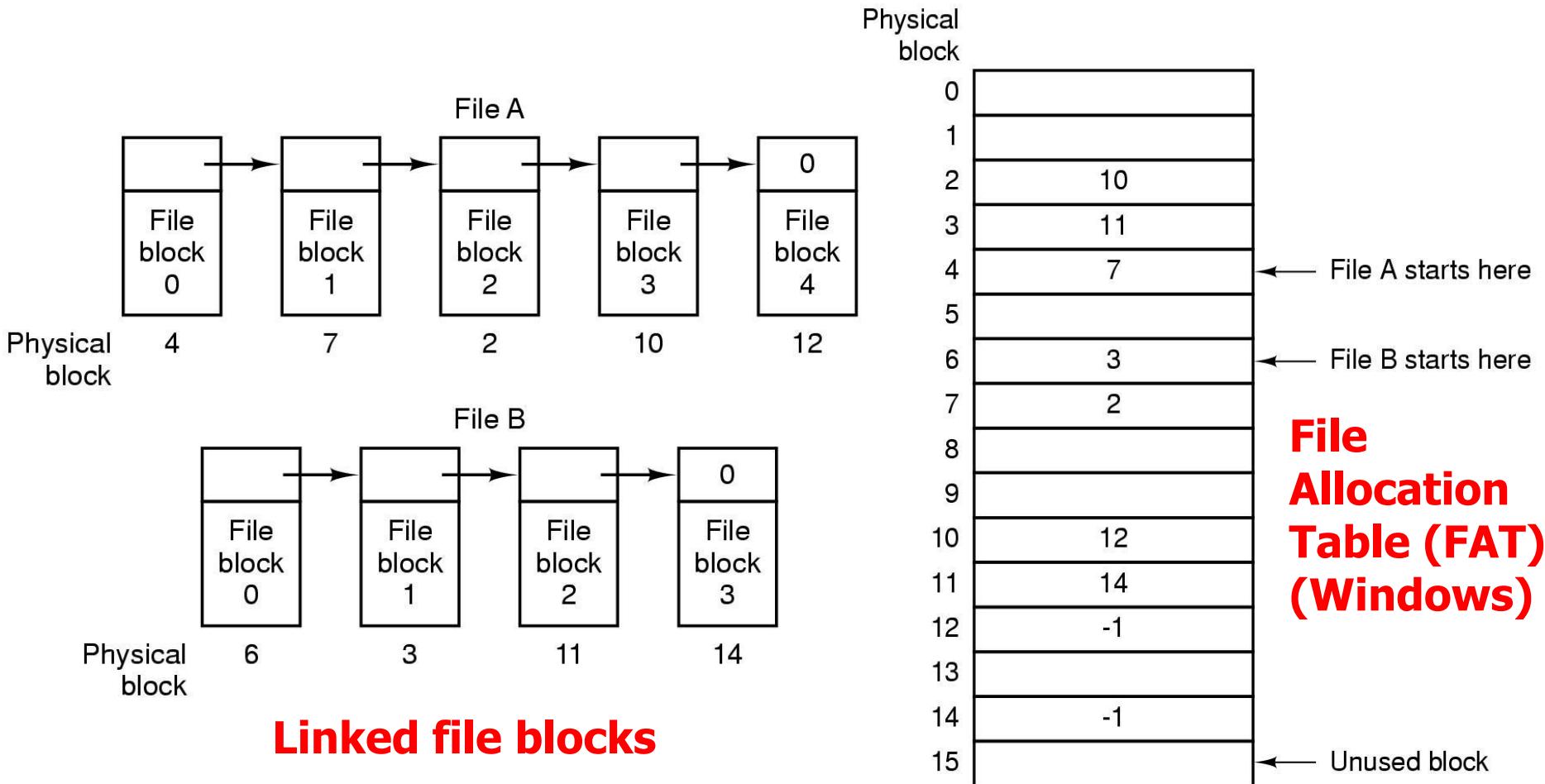
Files read/written in fixed size *blocks* (e.g. 1KByte)

Implementing files (1): contiguous allocation



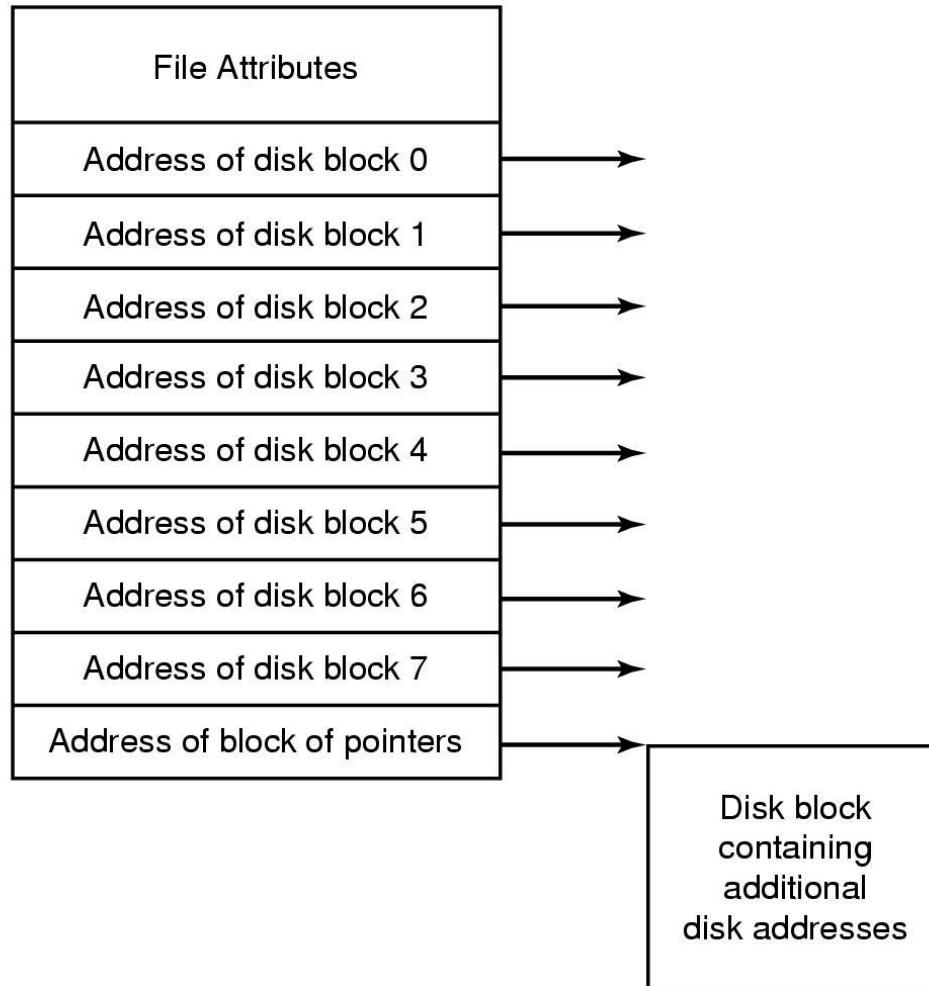
- Simple and efficient to implement
- Rapidly becomes fragmented
- Requires file size to be known in advance (ok for CD-ROMs)

Implementing files (2): linked lists and file allocation tables



- Linked disk blocks is slow, creates odd block size
- FAT (in memory) is fast but space-inefficient

Implementing files (3): i-nodes

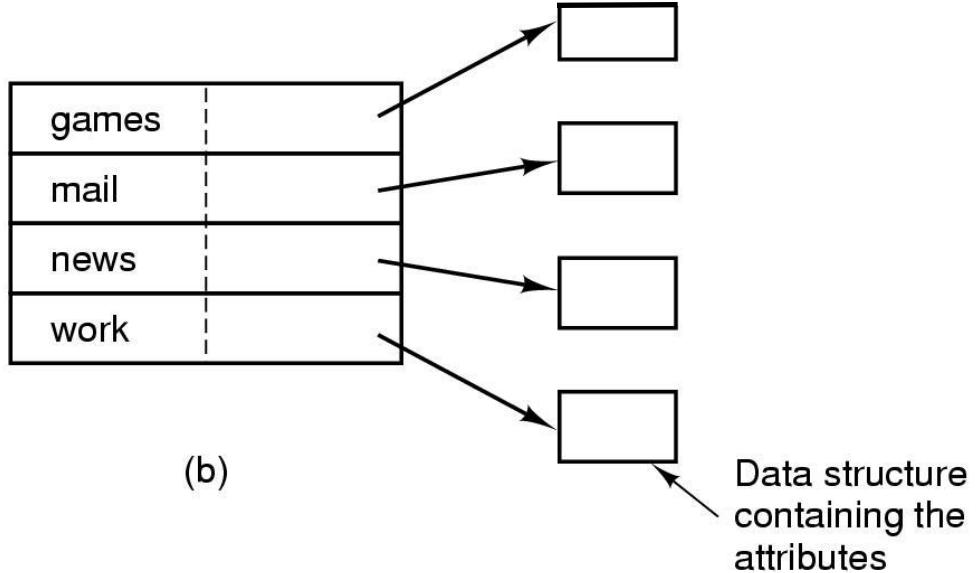


- One per file
- Read into memory when file opened

Implementing directories: directory structure

games	attributes
mail	attributes
news	attributes
work	attributes

(a)

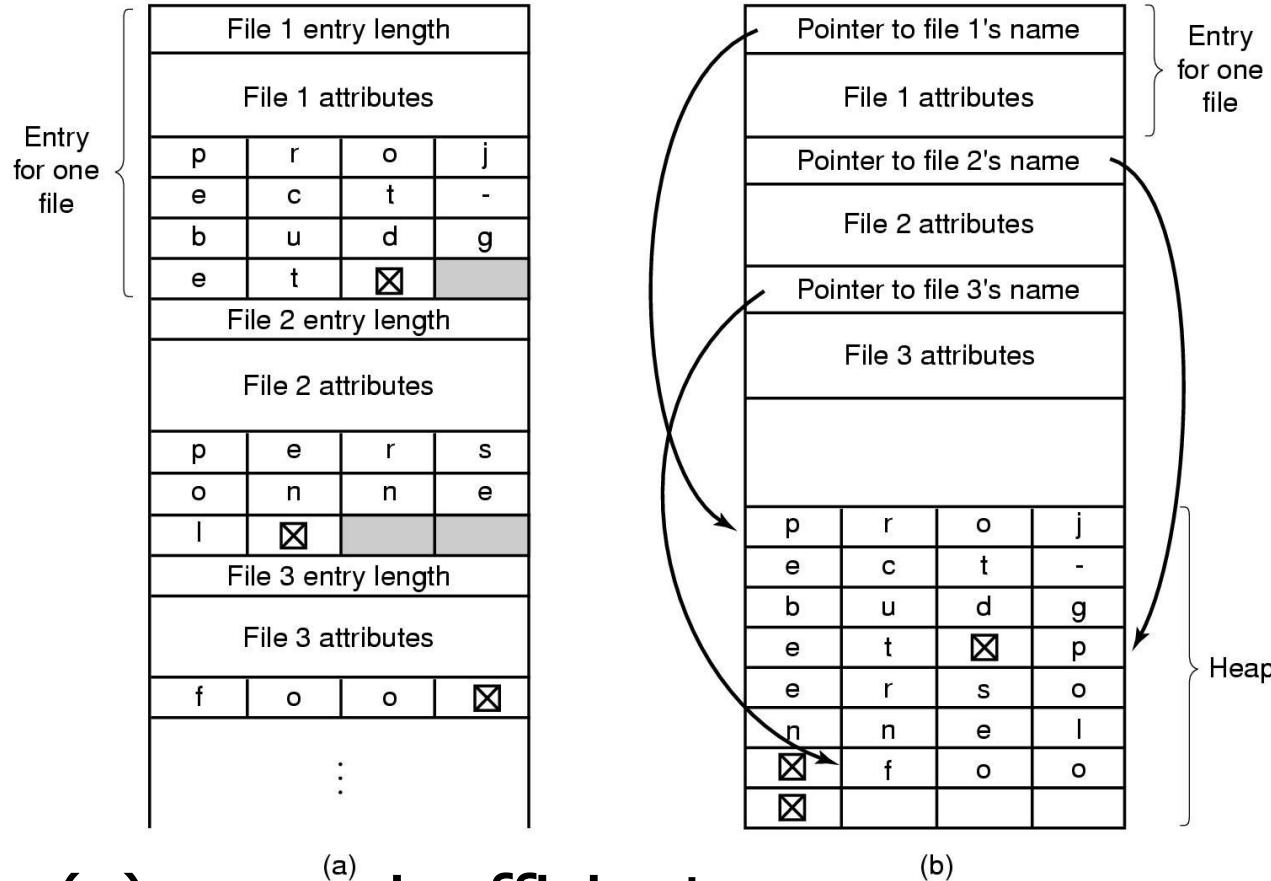


(a) Table-based directory (Windows)

- fixed size entries
- disk addresses and attributes in directory entry

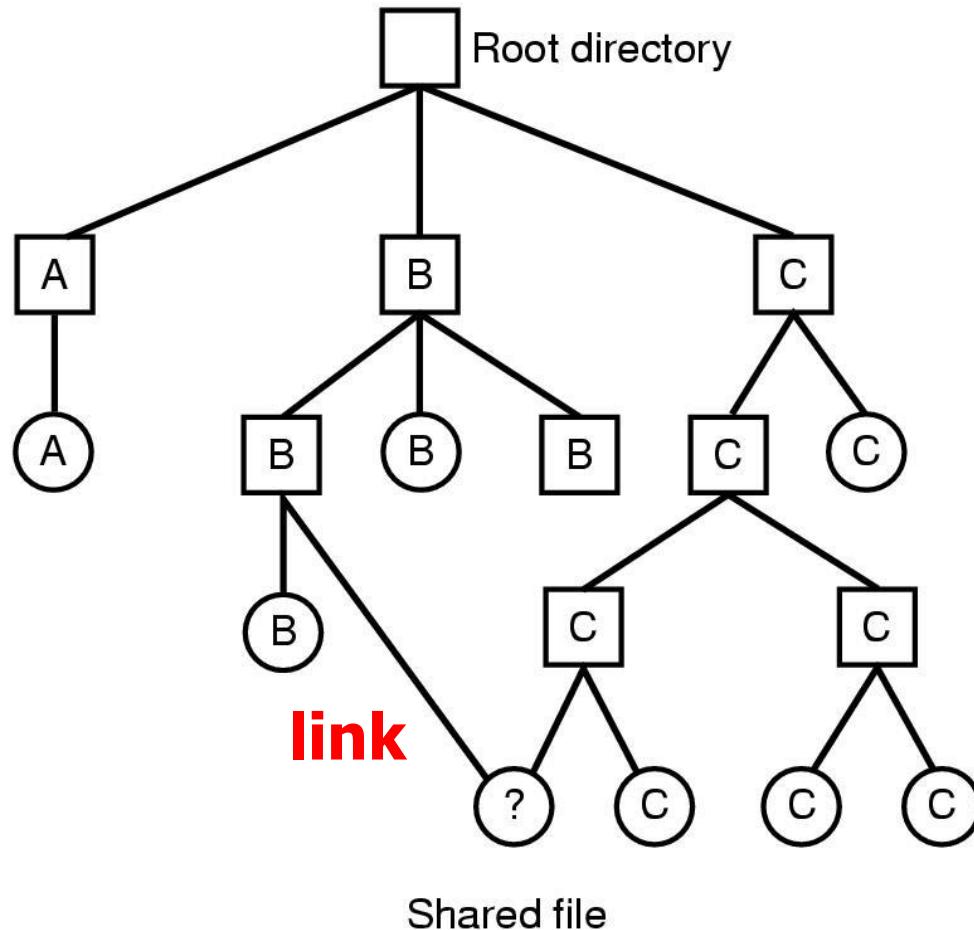
(b) Indirect directory (UNIX i-nodes)

Implementing directories: name storage



- In-line (a) space-inefficient
- Names in heap (b): complex
- Performance enhancements:
 - Hash table per directory (for large directories)
 - Cache frequent searches

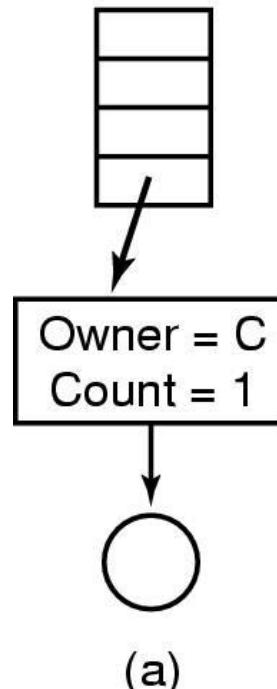
Shared files (linking)



- Now a Directed Acyclic Graph (DAG) instead of a tree
- Care needed when reading directories (cycles!)

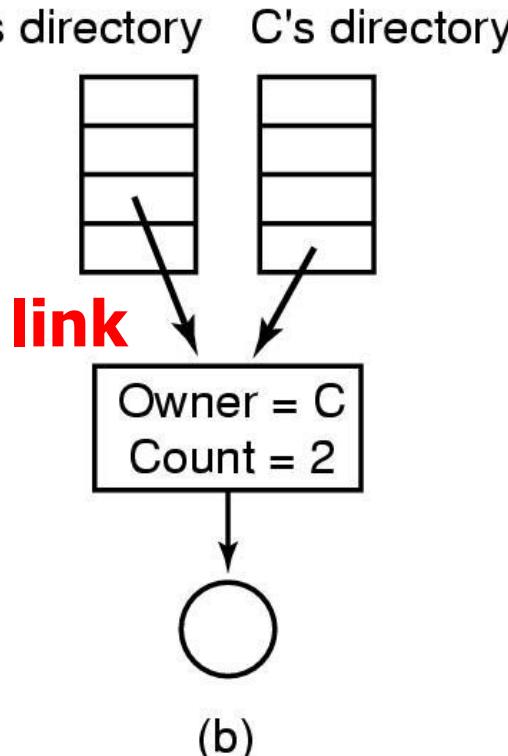
Shared files (2): hard links

C's directory



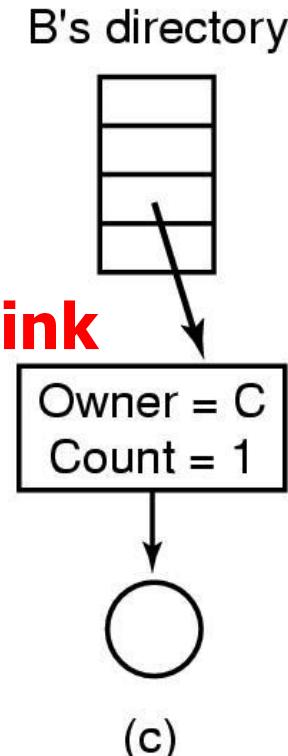
(a)

B's directory



(b)

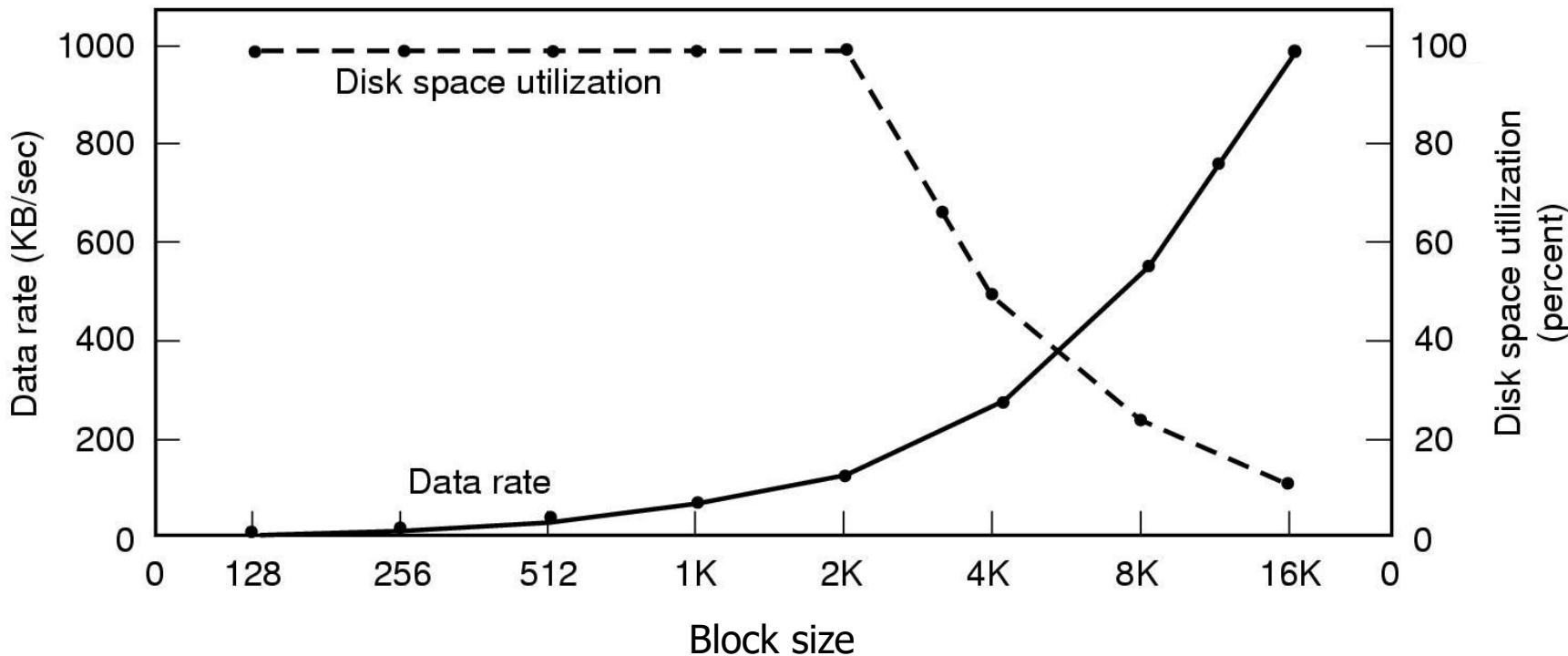
C's directory



(c)

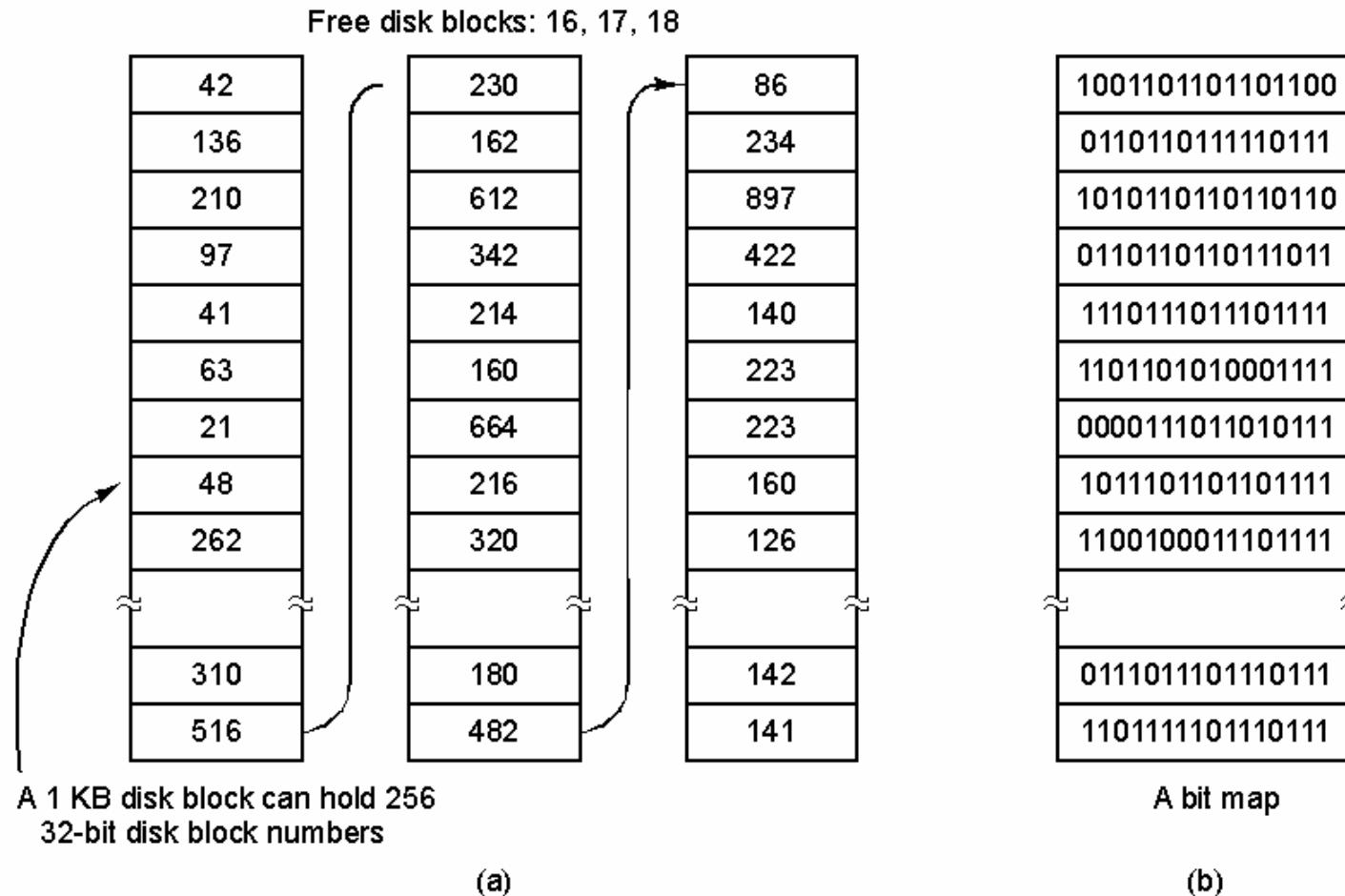
- Link is another directory entry to same i-node
- Deleting directory entry creates ownership issues
- Alternative: link is itself a file (breaks link on delete)
 - UNIX soft links
 - Windows shortcuts

Disk space management



- All files 2KBytes (typical average file size)
- Trade-off between utilisation and speed
 - Speed becoming more important

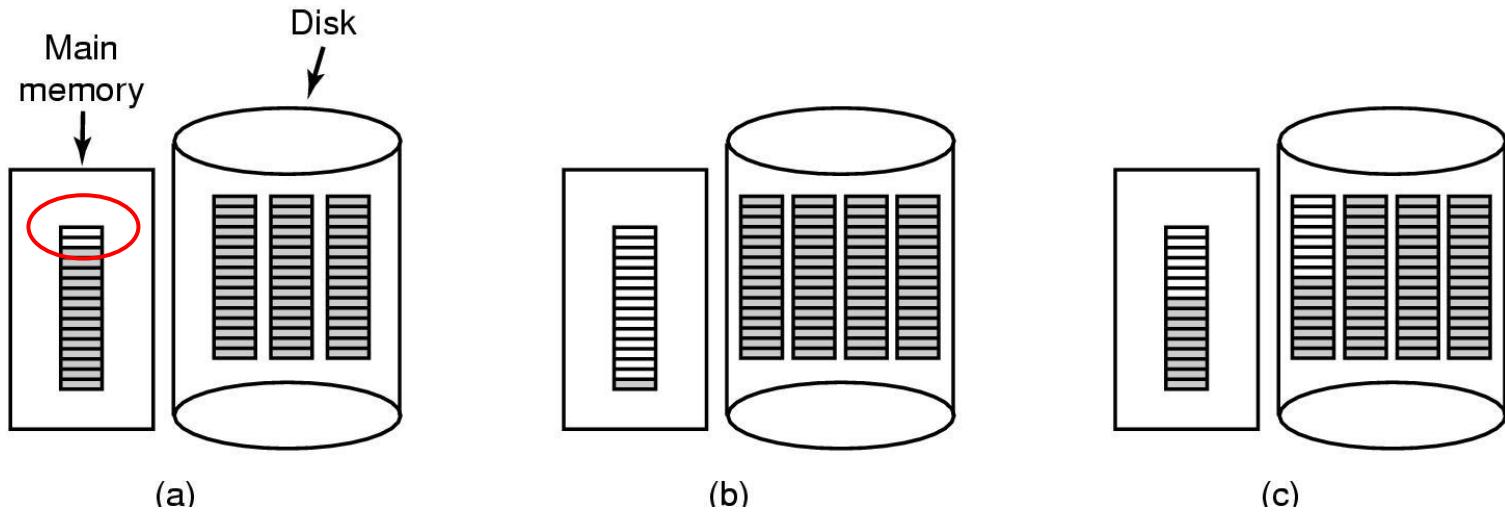
Free space management



- (a) Free blocks linked list – stored in free disk
(b) Bit map - one bit per free block

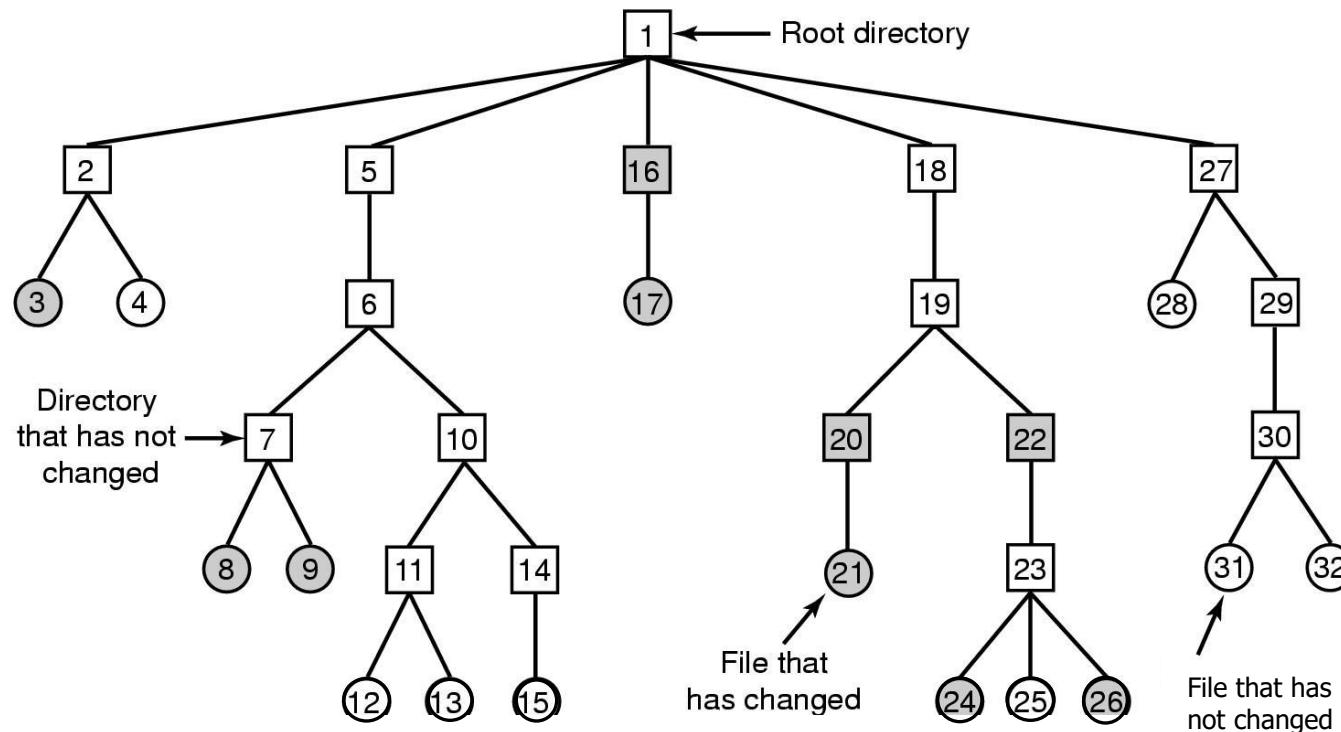
Optimising free space management

Room for 2
more entries



- a) In-memory free space list has two slots left
- b) Deleting a 3-block file creates a new free-space block
 - Creating a new 3-block file requires *disk read*
- c) Alternative strategy: retain half-full free block (split)

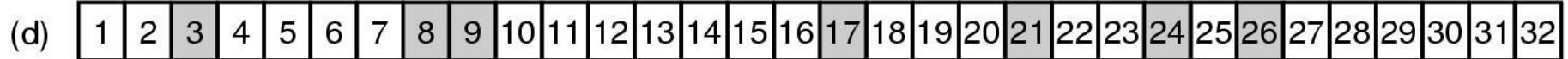
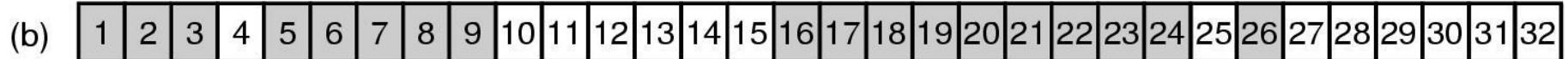
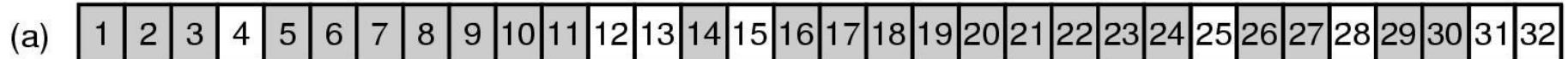
File system reliability: backups



Incremental backup strategy (logical backup)

- Periodic dump of all files
- Incrementally back up changed files
 - Include parent directories to enable clean restore

Logical backup algorithm



- a) Mark i-nodes of all modified files and *all* directories
- b) Unmark directories with no modified descendants
- c) Backup the modified directories
- d) Backup the modified files

File system reliability: scans

BLOCK NUMBER															
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	0	1	0	1	1	1	0	0	1	1	1	0	0	
Blocks in use															
0	0	1	0	1	0	0	0	0	1	1	0	0	0	1	1
Free blocks															

(a)

BLOCK NUMBER															
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	0	1	0	1	1	1	0	0	1	1	1	0	0	
Blocks in use															
0	0	0	0	1	0	0	0	0	1	1	0	0	0	1	1
Free blocks															

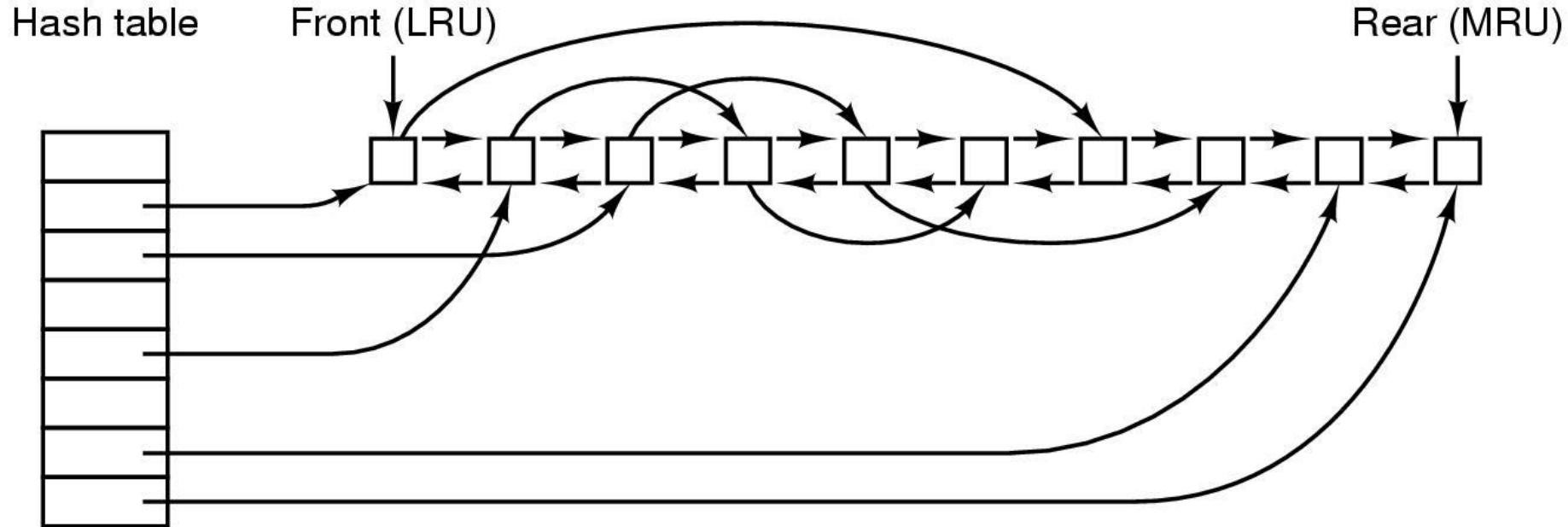
(b)

BLOCK NUMBER															
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	0	1	0	1	1	1	0	0	1	1	1	0	0	
Blocks in use															
0	0	1	0	2	0	0	0	0	1	1	0	0	0	1	1
Free blocks															

BLOCK NUMBER															
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	0	1	0	1	2	1	1	1	0	0	1	1	1	0
Blocks in use															
0	0	1	0	1	0	0	0	0	1	1	0	0	0	1	1
Free blocks															

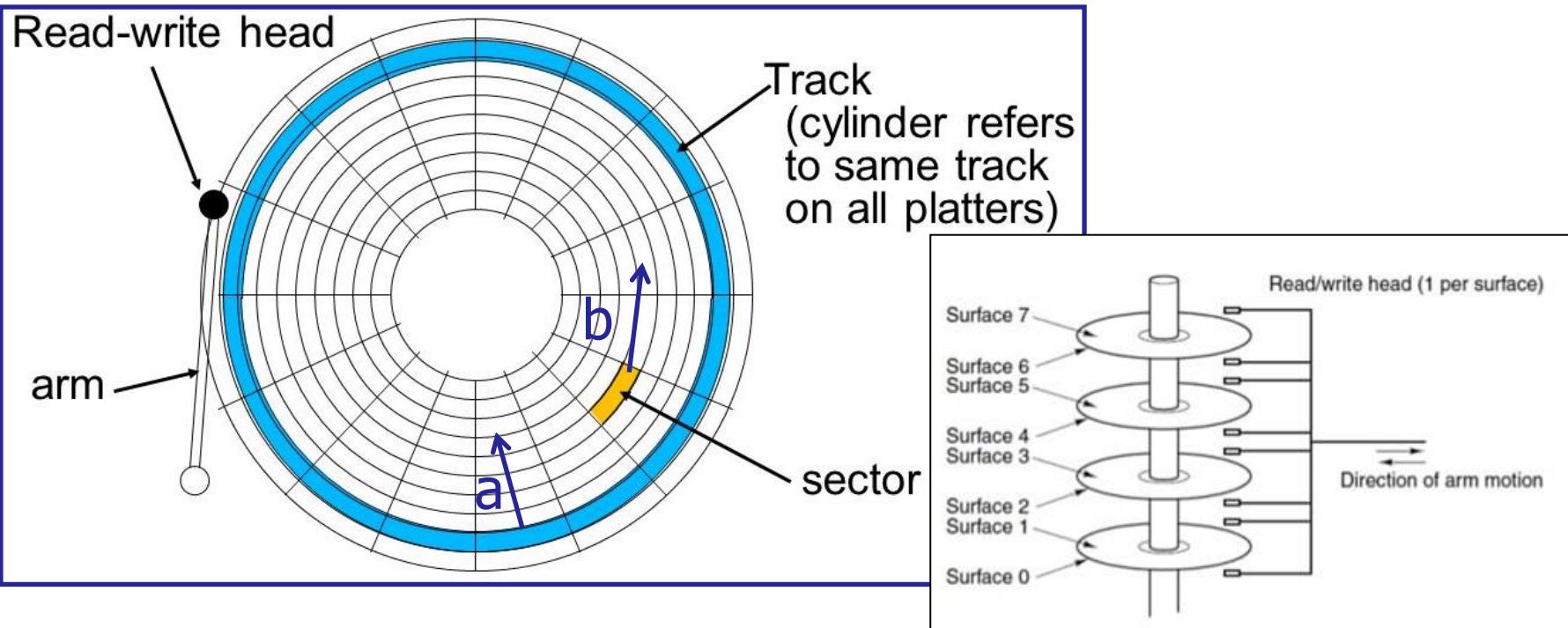
- Block in use/free counts checked for consistency
 - (a) consistent: Free = NOT(in use)
 - (b) missing block: **rebuild free list**
 - (c) duplicate block in free list: **rebuild free list**
 - (d) duplicate data block: **duplicate the block (*partial fix*)**

File system performance: block caching



- Hash table with least-recently-used collision list
- Variations to maintain file system integrity (crash-resistance):
 - Essential blocks written immediately (e.g. i-nodes)
 - Periodically flush the cache (UNIX: *update* process calls *sync* every 30s)
 - Immediately write all updates to disk (Windows/MS-DOS)
 - Maintain a journal of pending updates (NTFS)

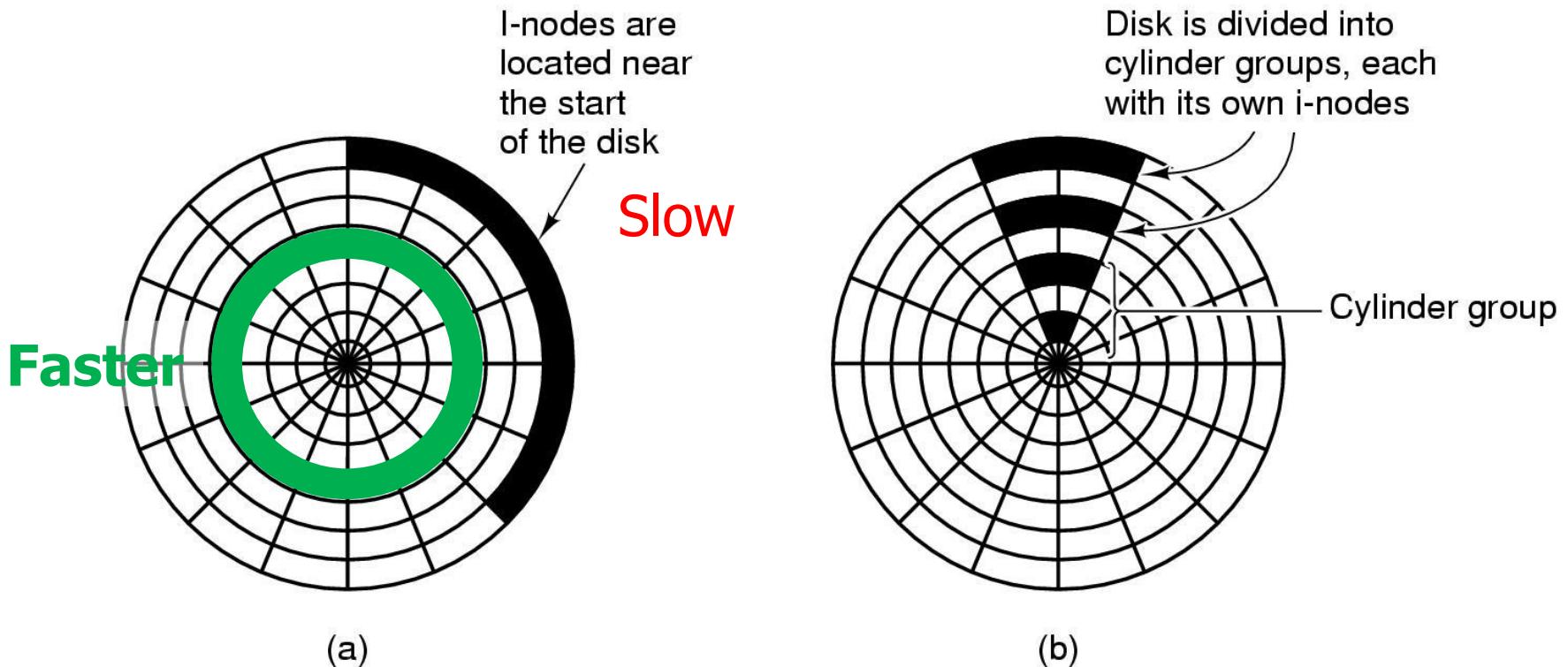
File system performance: disk drives



Disk time dominated by *seeks*:

- 5-10 msec to move between tracks (cylinders) [a]
- 5-10 msec to rotate to correct sector [b]
- Read speed: 50-160MB/second (1KB block takes < 0.02msec)

Minimising disk seeks

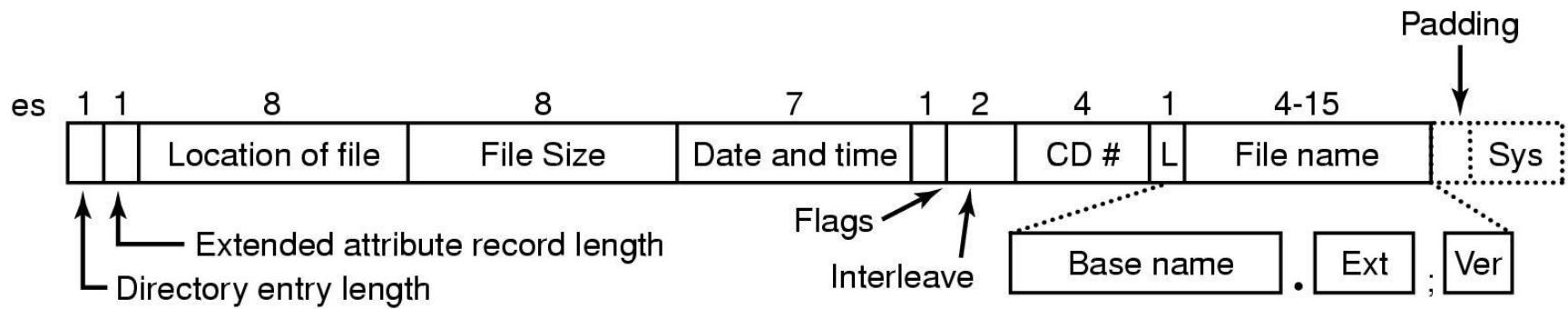


- Allocate blocks in chunks (e.g. two blocks per free chunk)
- Allocate blocks physically close together (same cylinder)
- Place i-nodes in the middle of the disk
- Interleave i-nodes and file blocks in *cylinder groups*

Log-structured file systems

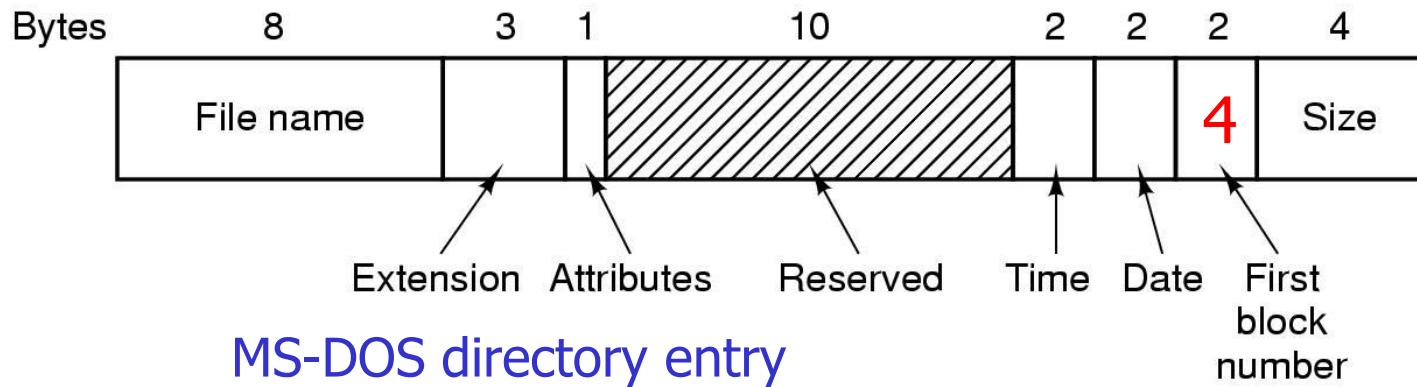
- Takes advantage of CPU and memory advances
 - disk caches can also be larger
 - increasing number of read requests from cache
 - most disk accesses will be writes
- Structure the entire disk as a log
 - Goal: maximise consecutive writes
 - have all writes initially buffered in memory
 - periodically write these to the *end* of the disk log
 - when file opened, locate i-node, then find blocks
 - i-node map (in memory)
 - Periodic cleaning to deal with deletes

CD-ROM file systems

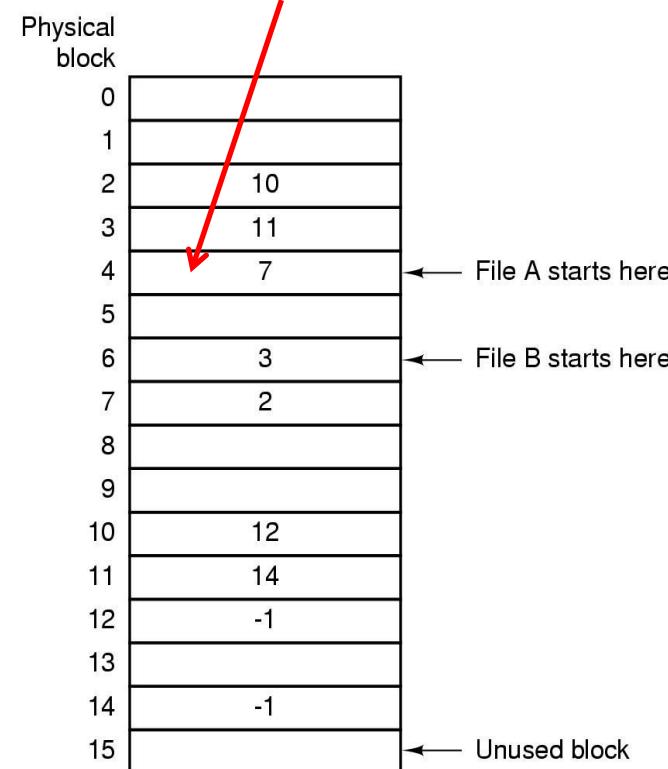


The ISO 9660 directory entry
(no subdirectories)

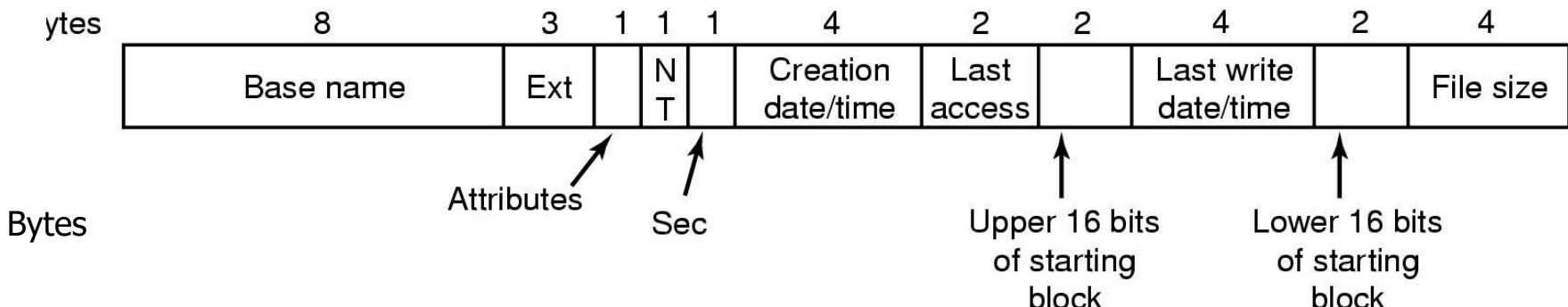
The MS-DOS file system



Block size	FAT-12	FAT-16	FAT-32
0.5 KB	2 MB		
1 KB	4 MB		
2 KB	8 MB	128 MB	
4 KB	16 MB	256 MB	1 TB
8 KB		512 MB	2 TB
16 KB		1024 MB	2 TB
32 KB		2048 MB	2 TB



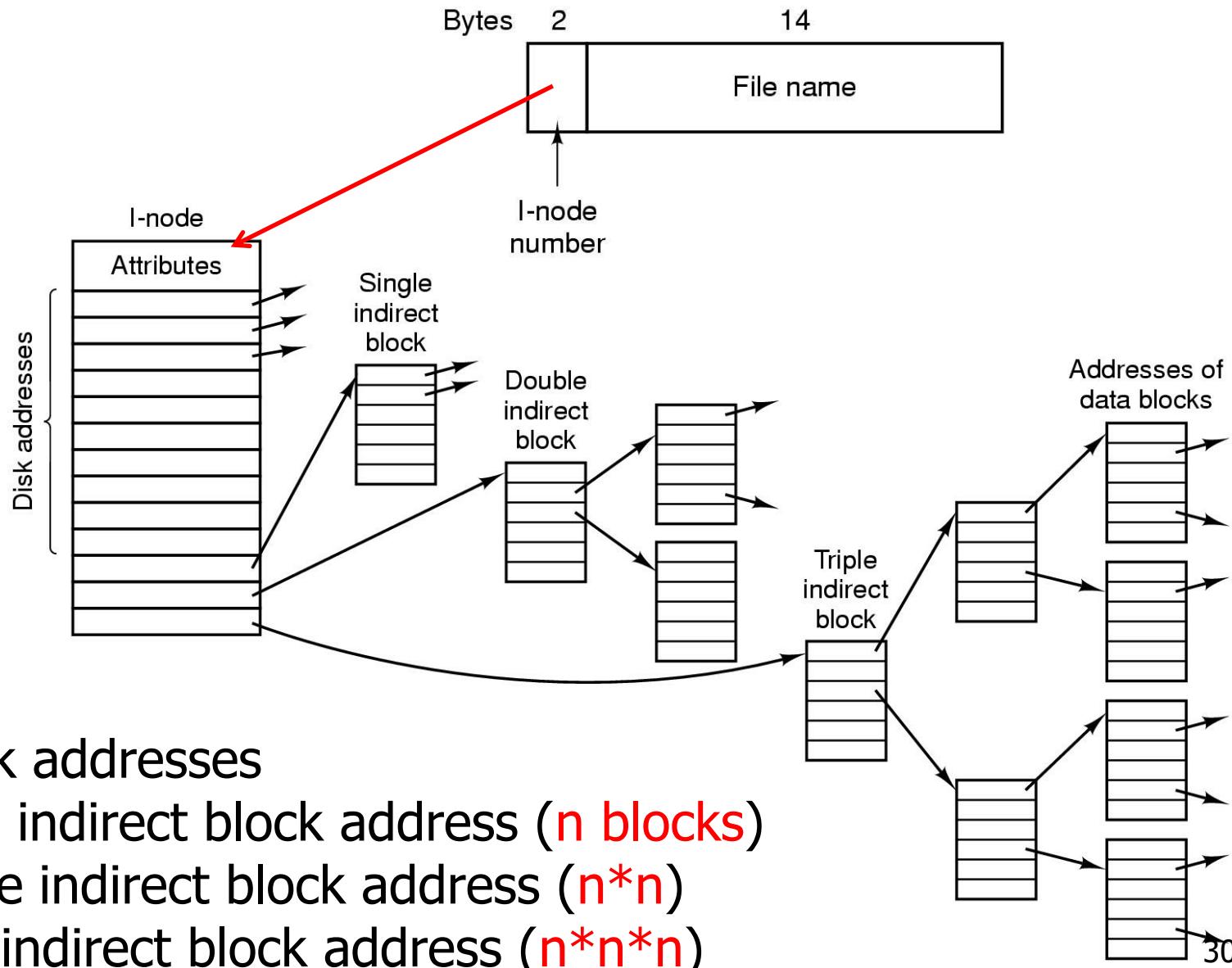
Windows file system - names



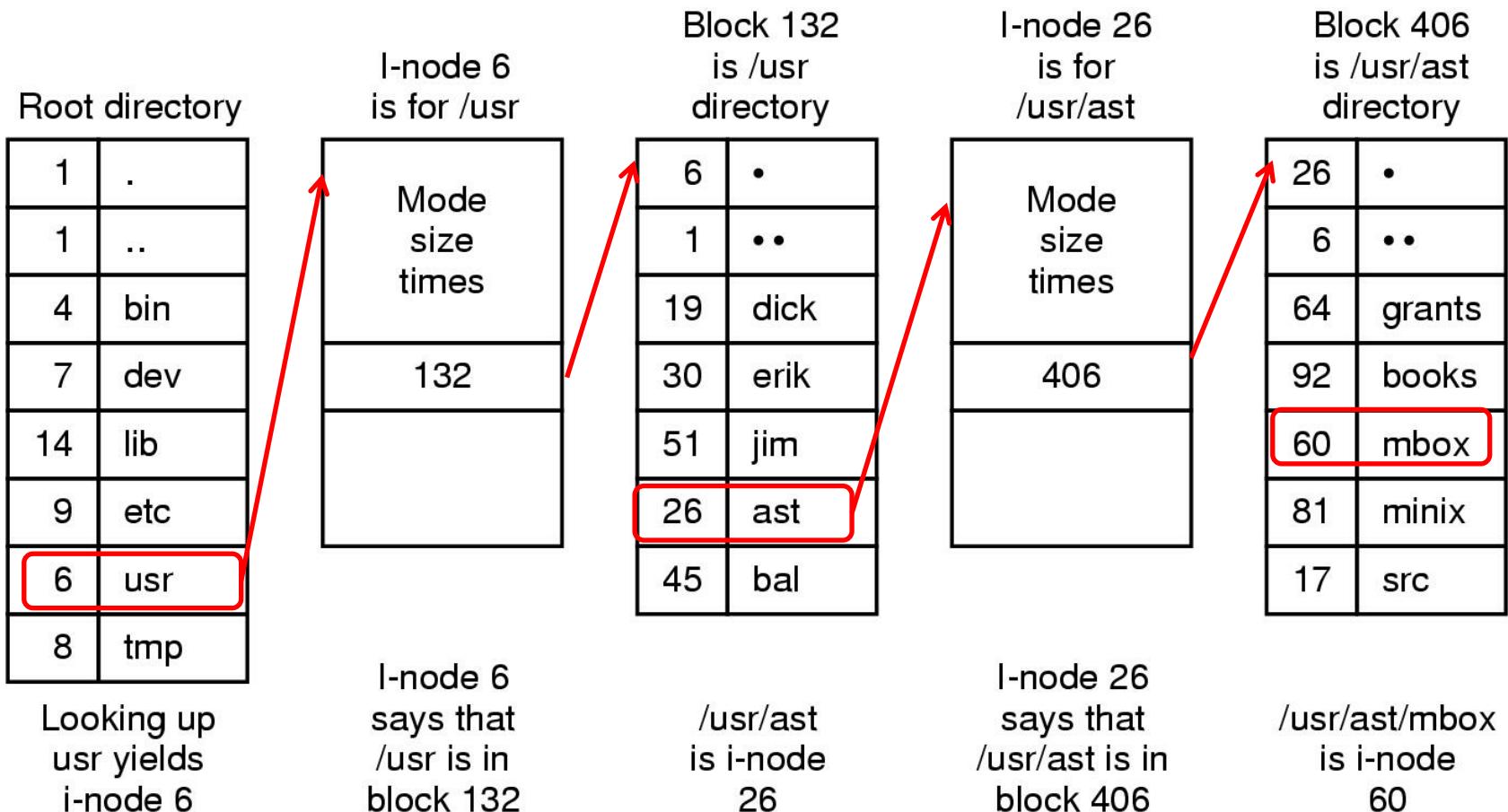
68	d o g	A	0	C	K							0	
3	o v e	A	0	C	K	t	h	e	I	a	0	z	y
2	w n f o	A	0	C	K	x	j	u	m	p	0	s	
1	T h e q	A	0	C	K	u	i	c	k	b	0	r	o
T	H E Q U I ~ 1	A	N	T	S	Creation time	Last acc	Upp	Last write		Low	Size	

Long name stored in a sequence of secondary directory entries

The UNIX file system



UNIX file lookup



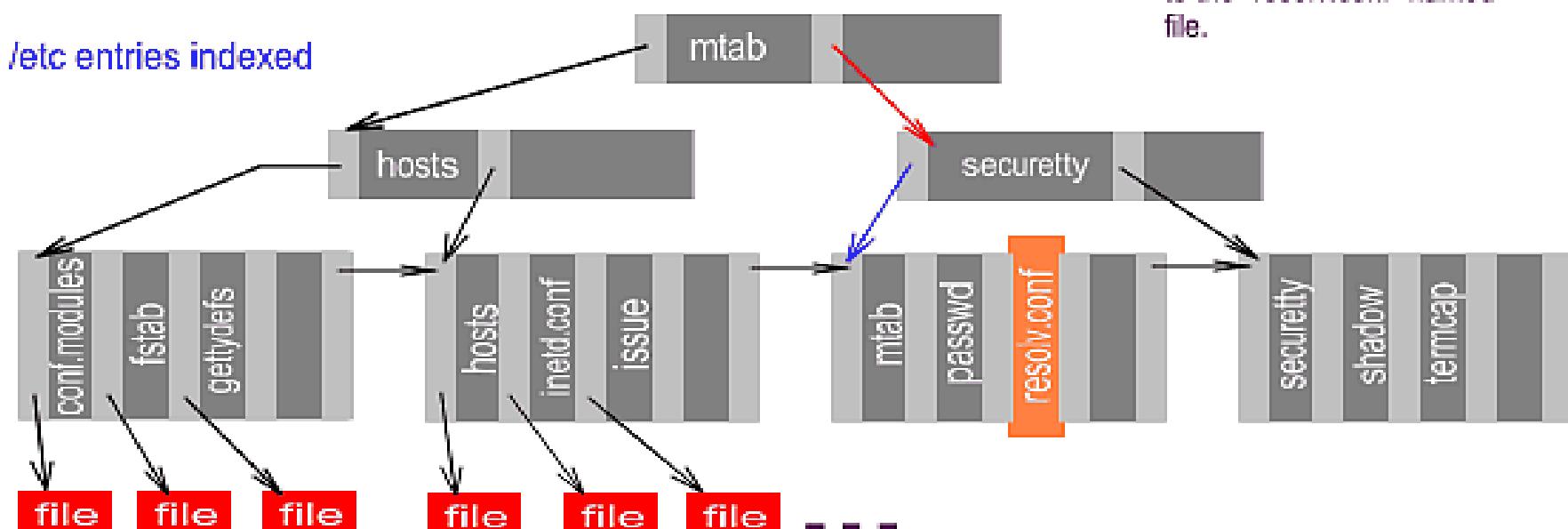
The steps in looking up `/usr/ast/mbox`

Btree file directories (Apple)

(1) to locate the file `resolv.conf` we begin at the tree's root, scan sequentially, and find that there is no key greater than `resolv.conf`, so we use the last pointer (in red)

(2) got directed to another internal node.
Let's do the same. Scan through the node's keys and realise that "security" is a greater key than "resolv.conf". We use the accompanying pointer (in blue).

(3) we got the final leaf node. Now it's time to scan sequentially throughout the ascending ordered keys of the node. Finally, found the desired key, we should use the accompanying pointer to the "resolv.conf" named file.

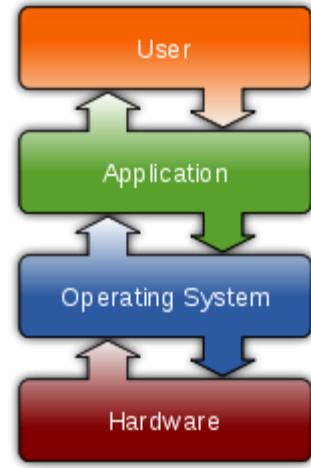


- Path length to all leaf nodes is the same
- Ordered keys for fast search
- Grow "upwards": split when too many keys

ENCE360

Operating

Systems



Input/Output

MOS Ch 5

Introduction

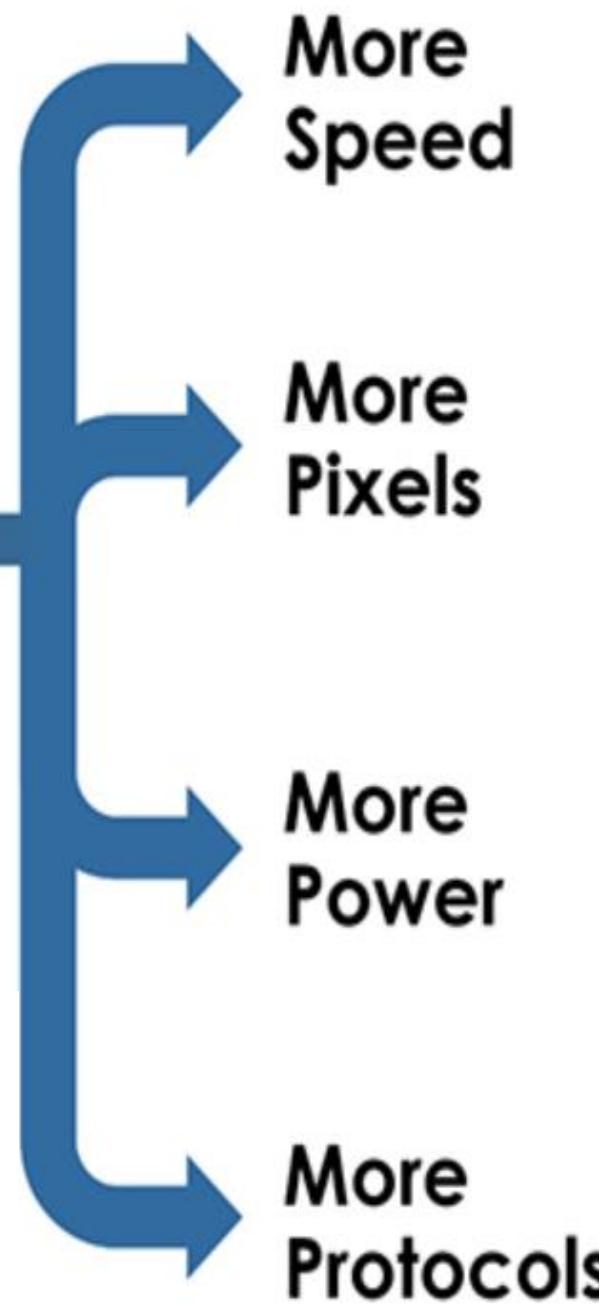
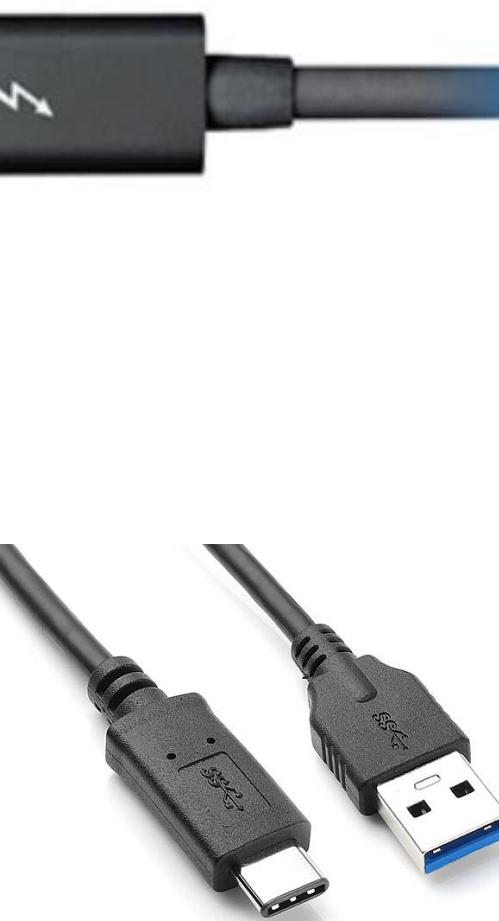
- One OS function is to control devices
 - significant fraction of code (80-90% of Linux)
- Want all devices to be simple to use
 - Convenience: treat all the same
 - E.g. file-like: stdin/stdout, pipe, re-direct
- Want to optimize access to device
 - efficient
 - devices have very different needs

Input/output challenges

- Wide variety of peripherals
 - Third party hardware and software
 - Delivering different amounts of data
 - At different speeds
 - In different formats
- All slower than CPU and RAM
- Need I/O subsystem to handle

Throughput	Type of Service
10 bps	keyboard
64 kbps	Integrated Services Digital
2.4 Mbps	Bluetooth 3.0
480 Mbps	USB 2.0
1 Gbps	Gigabit Ethernet
1 Gbps	IEEE 802.11ac WiFi
3.2 Gbps	FireWire 3.0
10 Gbps	USB 3.1
16 Gbps	eSATA 3.2
31 GBps	PCI Express 4.0
40 Gbps	USB-C Thunderbolt 3

USB-C



40 Gbps



Two 4k



Up to 100w



2021: devices are getting faster

GPUs: Nvidia GeForce GTX TITAN Z

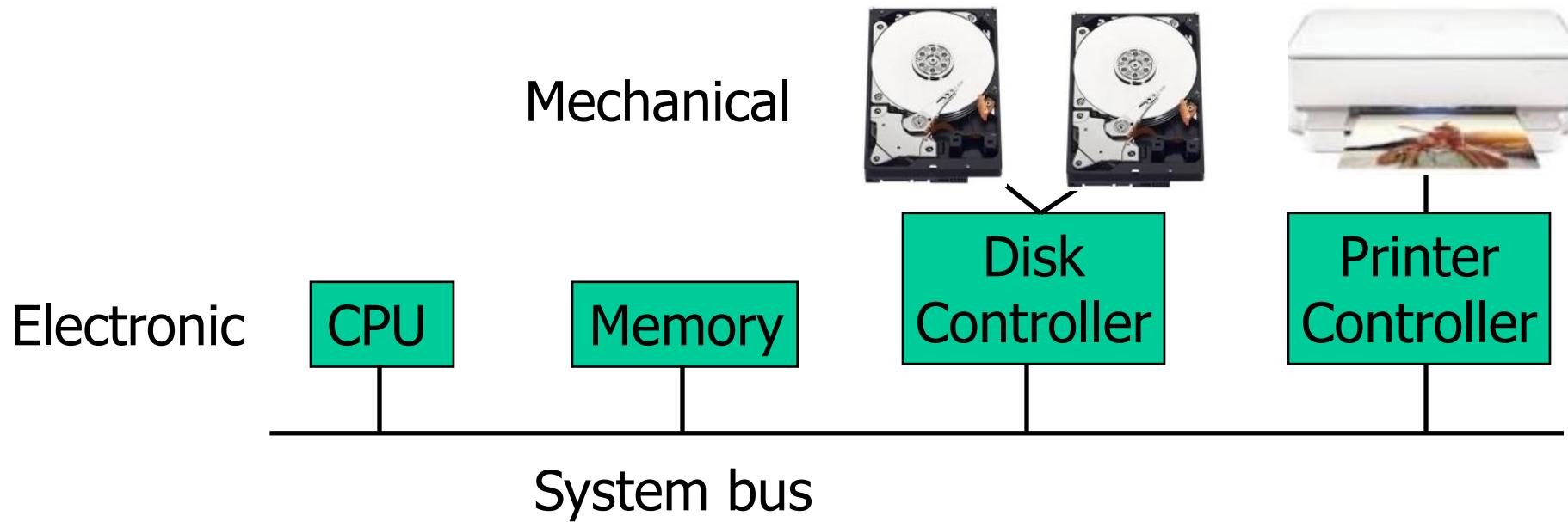
- 5,760 CUDA cores
 - Equivalent to ~100 CPUs
- 12 GB of GDDR5 memory
 - I/O speed can dominate



I/O hardware device types

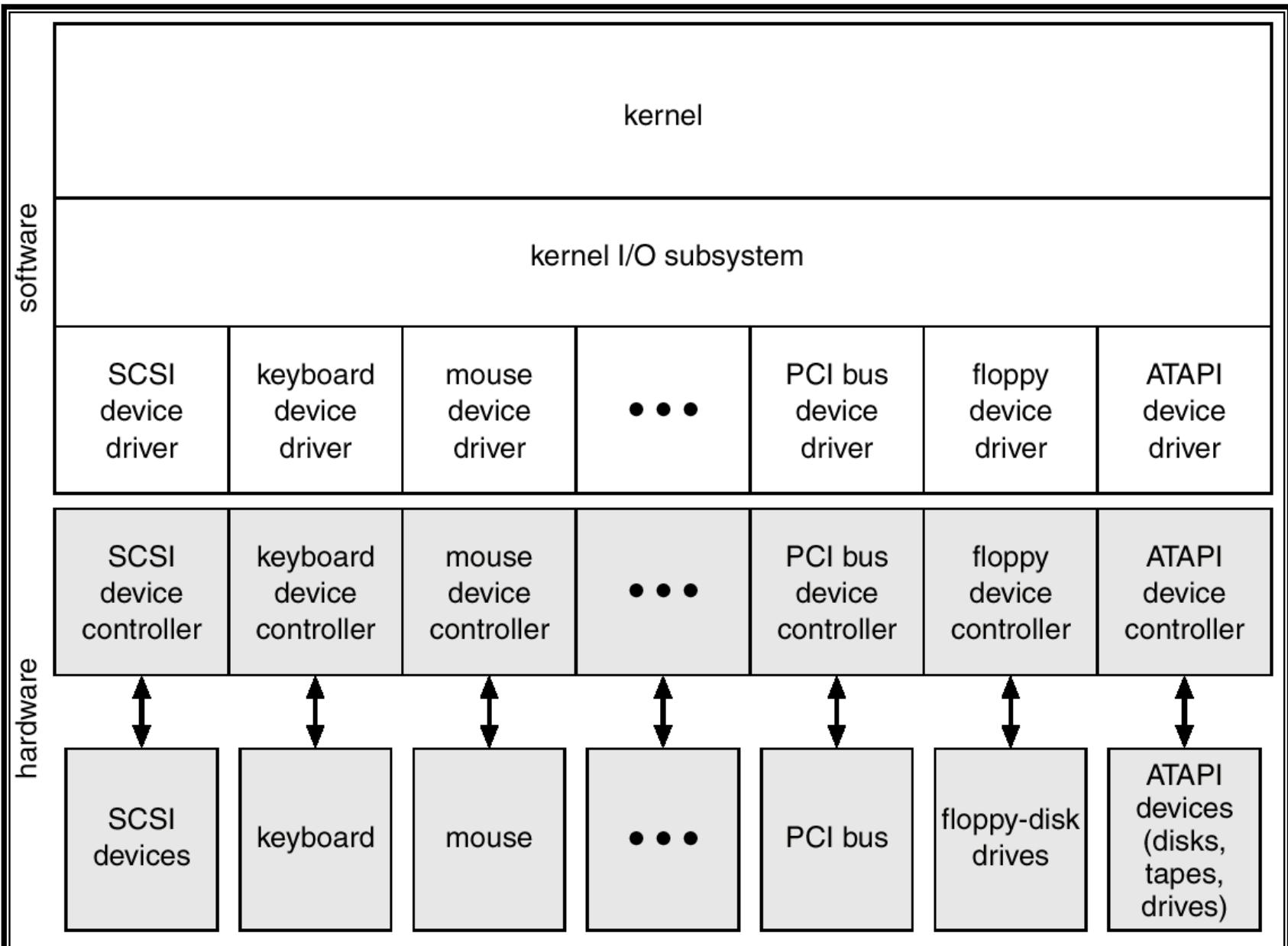
- **Block**
 - Data stored in fixed-sized blocks (512B to 32KB)
 - Blocks addressed and read independently
 - Examples: disk, CD-ROM, USB stick
- **Character**
 - Generates or receives a character stream
 - Examples: printer, network, mouse
- **other**
 - Example: clocks (just generate interrupts)

Hardware: devices and controllers



- Mechanical interface is very low-level
 - Stream of bytes (preamble, data, ECC)
- OS deals with **device controller**, e.g. for disk drive:
 - Convert serial stream to blocks of bytes
 - Perform error correction
 - Copy to main memory
- On device or part of computer

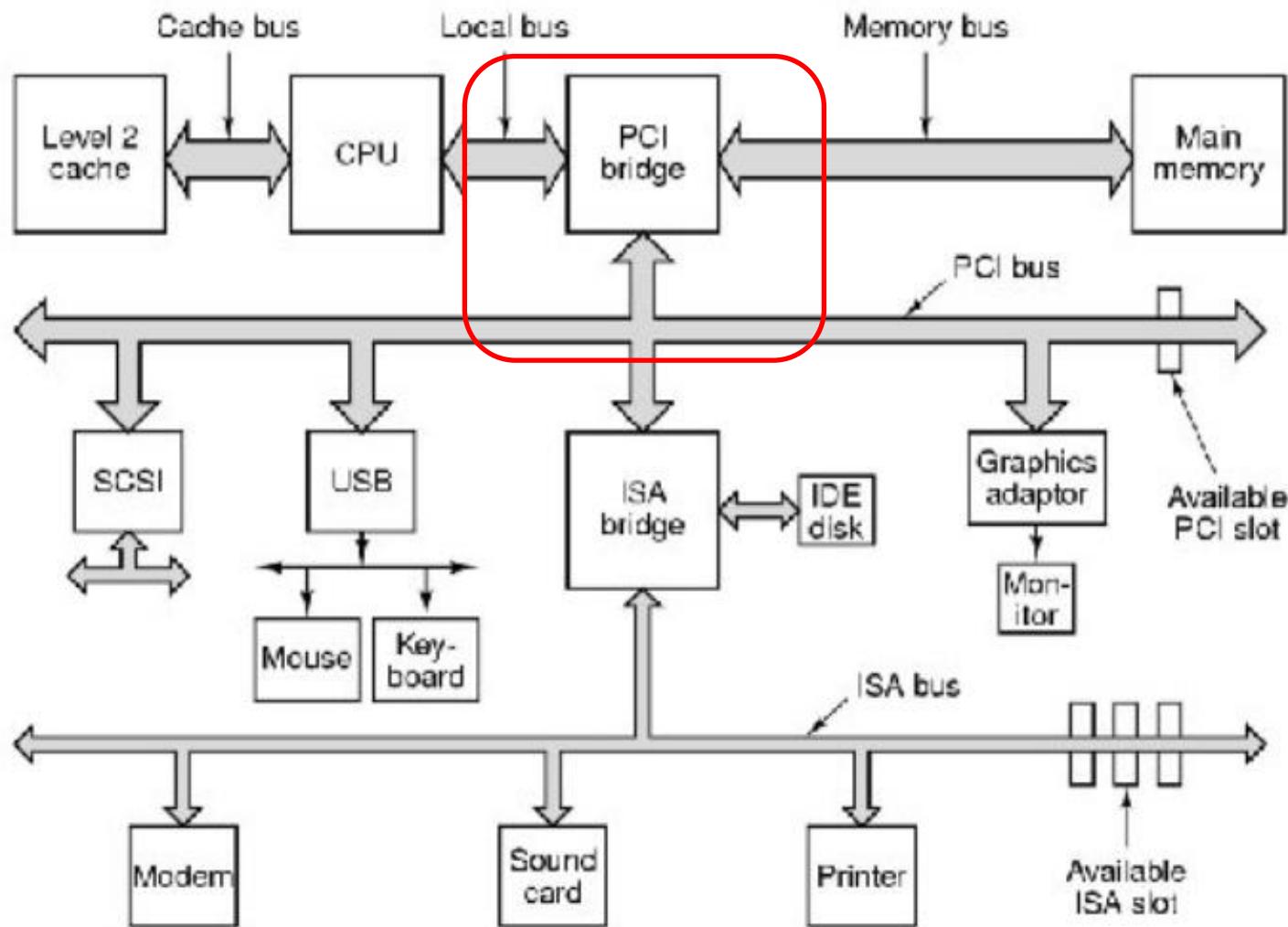
Kernel I/O structure



Memory mapped I/O

- Direct I/O: CPU accesses controller registers and data buffer separately from memory
 - IN REG, PORT
 - OUT PORT, REG
- Memory-mapped I/O: map I/O ports to memory addresses
 - Can perform I/O using standard languages (e.g. C)
 - Can perform same operations (e.g. TEST instruction)
 - Easy to control access
- Some drawbacks:
 - Caching needs to be disabled
 - Requires second bus (for speed)

Example: Pentium architecture



- PCI bridge diverts I/O requests based on address range

Bus architectures

- Serial bus:
 - single pair of wires for unidirectional data transmission
 - e.g. USB ports, keyboard/mouse, network connections
- Parallel bus:
 - multiple wires for high-speed bi-directional data transfers
 - e.g. disk drives, graphics cards

Bus acronyms

- EISA - Extended Industry Standard Architecture
- ATA – Advanced Technology Attachment (for IBM AT PC)
uses IDE connector (Integrated Drive electronics)
- PCMCIA
 1. PC Card Means Confusing The Industry Again
 2. People Cannot Memorize Computer Industry Acronyms
 3. **Personal Computer Memory Card International Association**
 4. Plastic Connectors May Crack If Adjusted
- SCSI – Small Computer System Interface
- AGP – Accelerated Graphics Port
- PCI – Peripheral Component Interconnect
- SATA - Serial ATA
- eSATA – External SATA
- USB – Universal Serial Bus

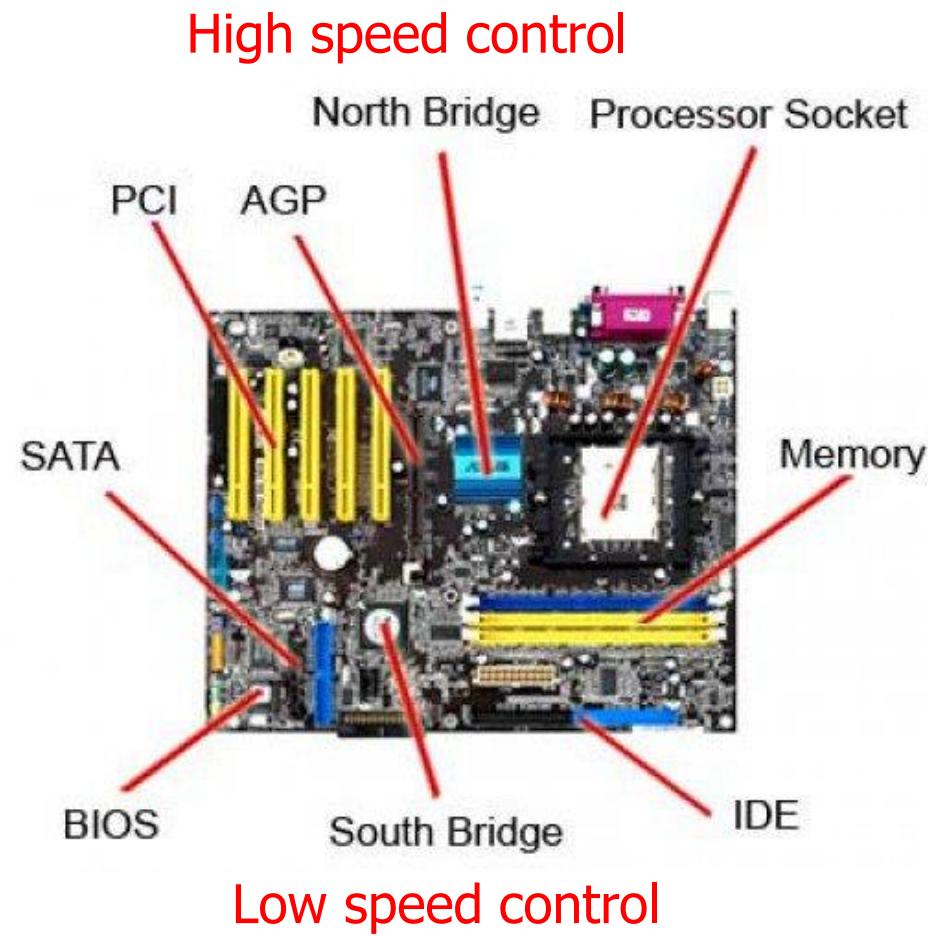
• Parallel Bus Interfaces

history:

- EISA 33 MBps
- ATA 167 MBps
- SCSI 320 MBps
- Express Card (was PC Card, which was PCMCIA) 400 MBps

now:

- AGP 2.1 GBps
- PCI Express 4.0 31GBps (2017)



• Serial Bus Interfaces

- eSATA: 16 Gbps
- USB-C Thunderbolt 3: 40 Gbps

SATA: Serial ATA

- **SATA 1.0** 1.5 Gbit/s (First generation)
- **SATA 2.0** 3 Gbit/s (Second generation)
- **SATA 3.0** 6 Gbit/s (Third generation)
- **SATA 3.2 16 Gbit/s** (SATA Express)
- **eSATA:** special connector specified for external devices
- **eSATAp:** power over eSATA or eSATA/USB Combo

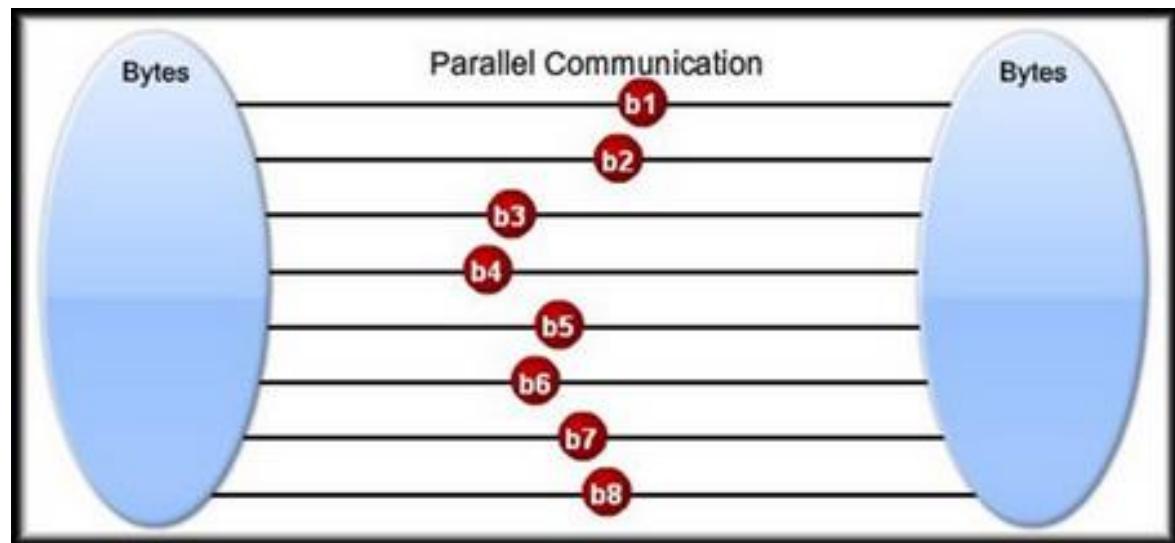
- Solid State Disk drives are close to saturating the SATA 3 Gbps limits
- Ten channels of fast flash can actually reach such a high bandwidth that a move from SATA 3 to SATA 3.2 would benefit the flash read speeds.

Parallel SCSI: **S**mall **C**omputer **S**ystems **I**nterface

- Parallel interface with 8, 16, 32 or 64 bit data lines
- Daisy chained disks
 - 7 for standard SCSI
 - 15 for wide SCSI
- Devices are independent
- Devices communicate with each other and host
- SATA is simpler (cheaper) so now more common

Serial now beats parallel interfaces – why?

- **Parallel:**
 - at 1GHz: bit “travels” 300mm per tick
 - at 1000GHz: 0.3mm per tick
 - clock skew: different bits arrive at different times depending on cable length



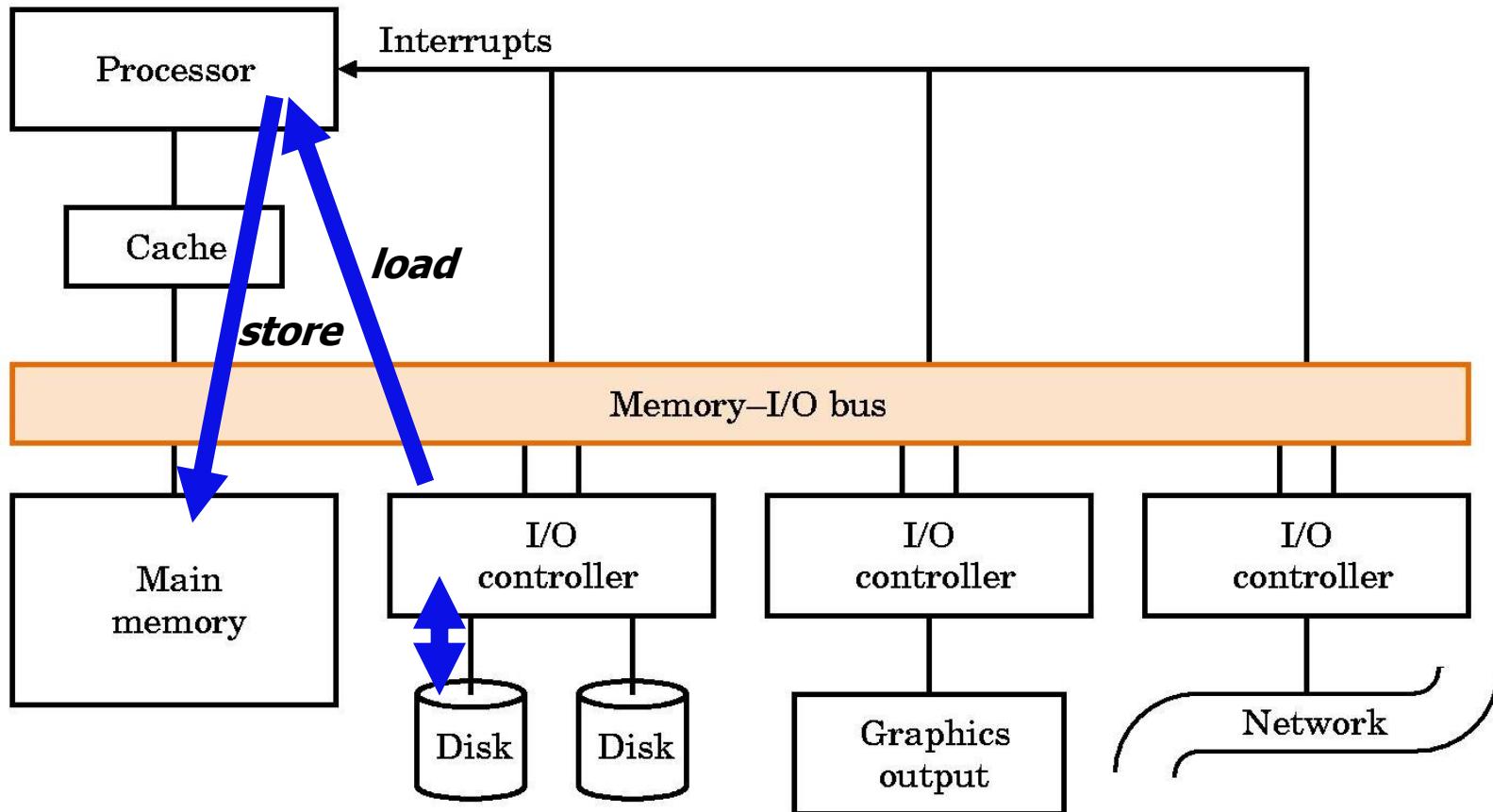
- **Serial:**
 - send one bit at a time
 - no frequency limit
 - e.g. Serial SCSI supports higher data rates than Parallel SCSI

Architectures: FireWire vs USB

- **FireWire:** "Peer-to-Peer" architecture
 - peripherals negotiate bus conflicts to determine which device can best control a data transfer
- **Hi-Speed USB:** "Master-Slave" architecture
 - computer handles all arbitration functions and dictates data flow to, from and between the attached peripherals
 - Adds overhead that slows data flow control

CPU-based data transfer

- CPU determines either through polling or interrupts that the device is ready
- Transfers one word of data
- Polls or waits for interrupt
- Transfers next word...

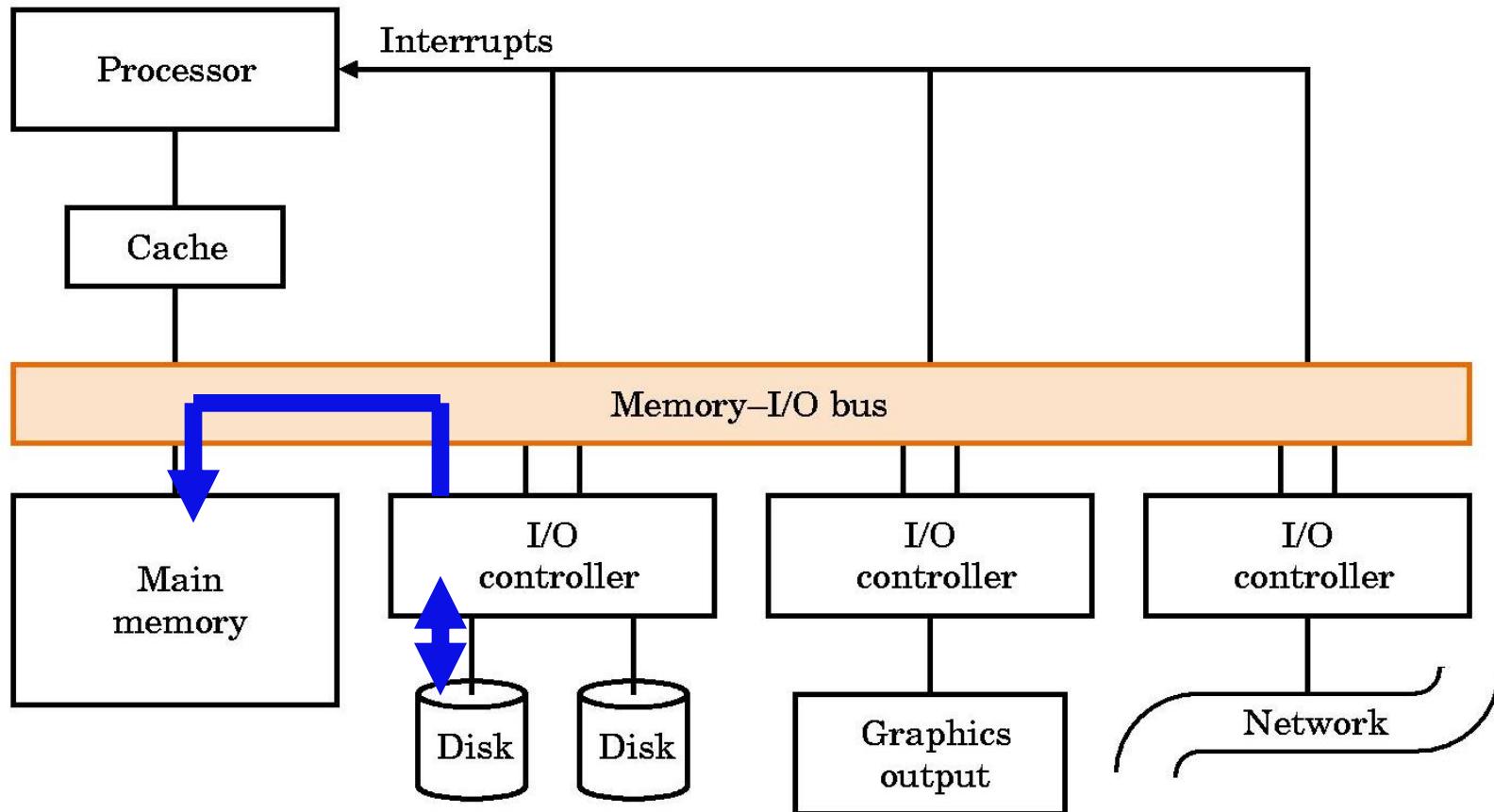


Direct Memory Access (DMA)

- Interrupt driven and programmed I/O require active CPU intervention
 - **CPU involved/interrupted per block**
 - **CPU is tied up**
 - **Transfer rate is limited**
- Versus direct Memory Access (DMA)
 - One operation per I/O *request*

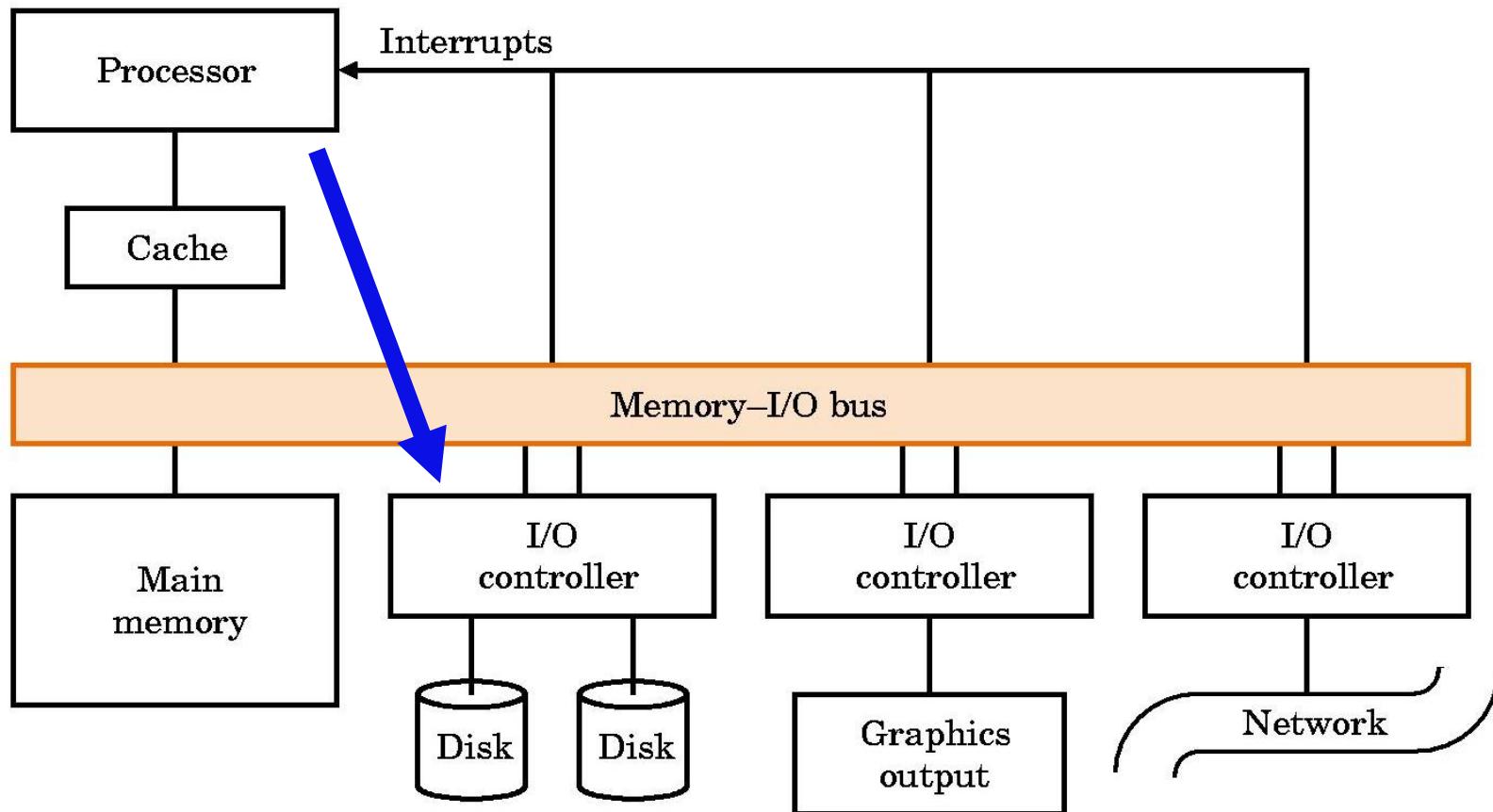
Direct Memory Access (DMA)

- Device controller or DMA controller (extra hardware) accesses memory directly
- CPU sets up the controller and is then released to continue processing



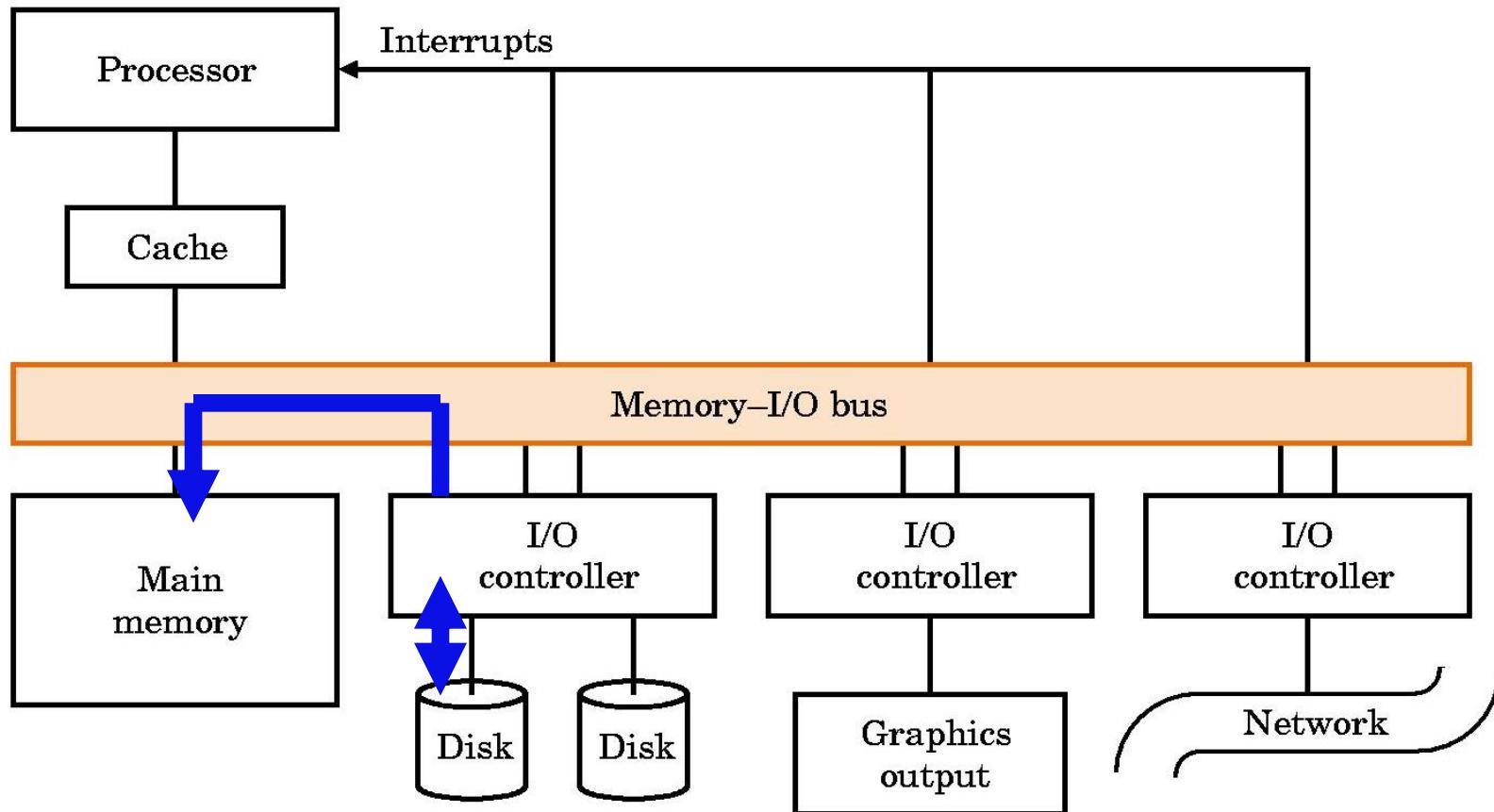
DMA operation (1): initialise

- Processor provides to the DMA device the I/O number of bytes, memory address, and the operation to be performed



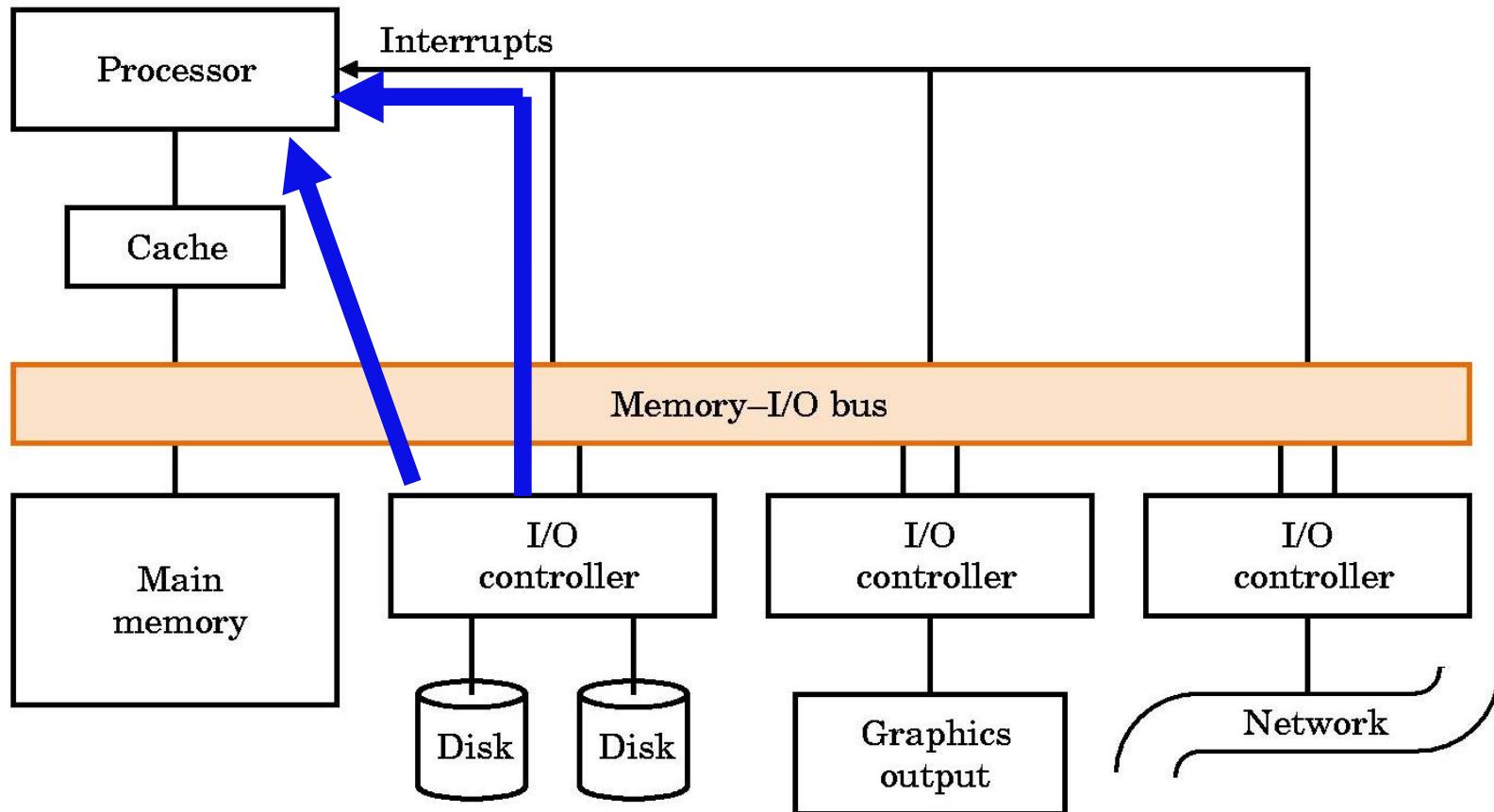
DMA operation (2): transfer data

- The device performs the transfer over the memory-I/O bus to or from memory

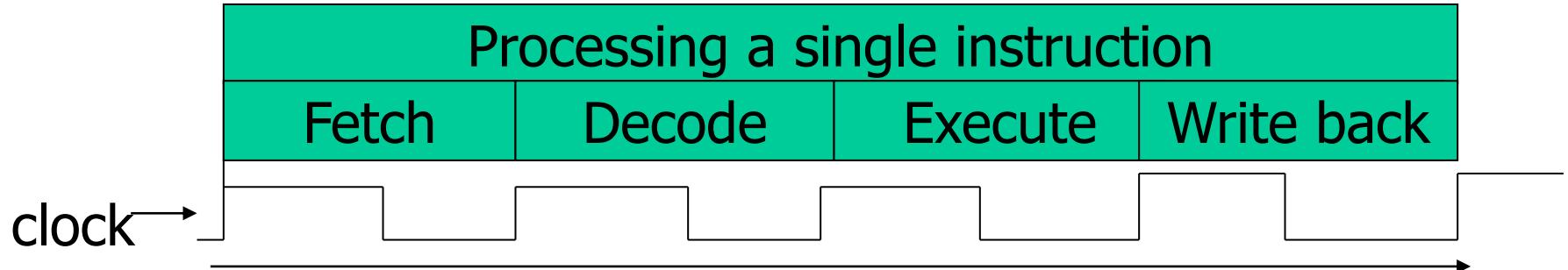


DMA operation (3): done

- Device interrupts the processor
- Processor checks the status registers



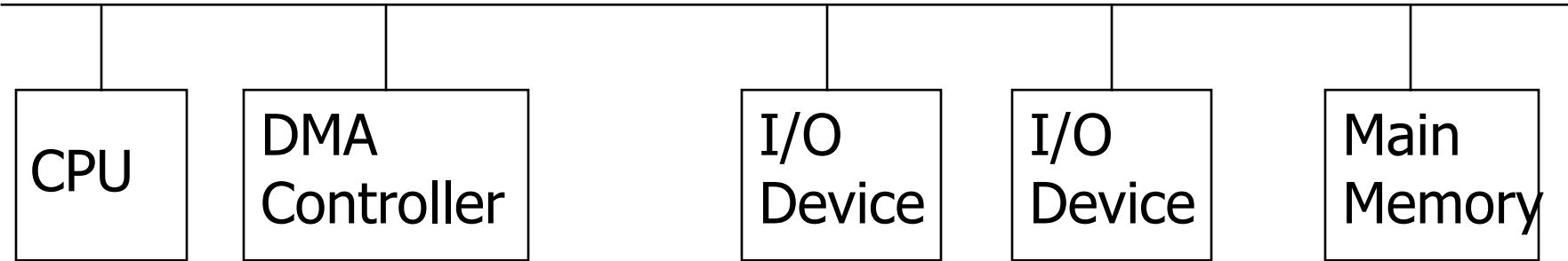
DMA cycle stealing



- DMA controller takes over bus for a *single clock cycle* and transfers one word of data
- Not an interrupt
 - CPU suspended just before it accesses bus
 - i.e. before an operand or data fetch or a data write
 - No context switch
- Slows down CPU a small amount
 - < 1 instruction vs multiple for an interrupt/context switch

DMA hardware configurations (1)

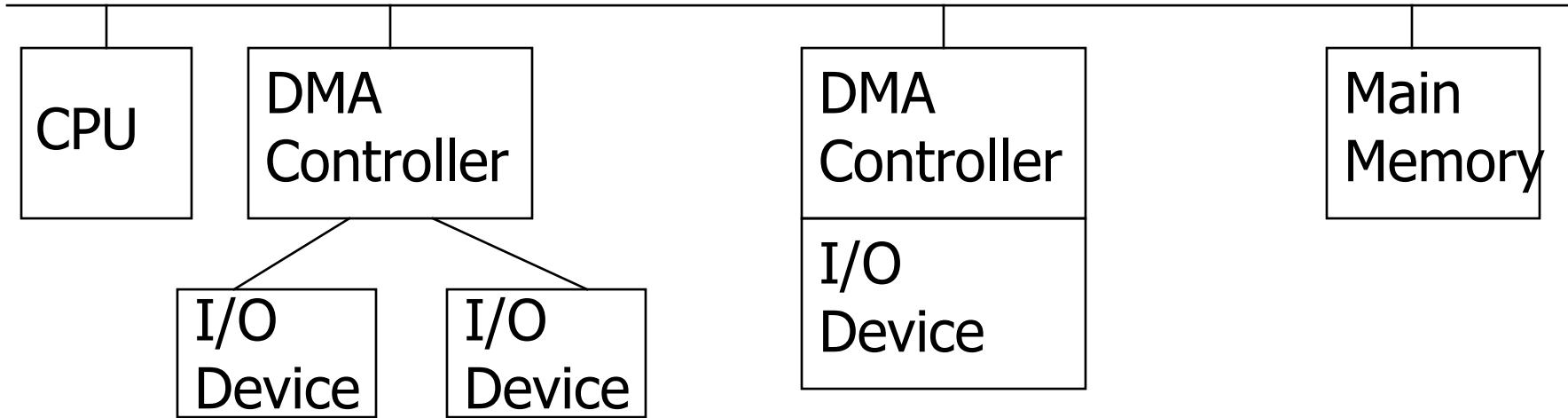
DMA shares bus with I/O units



- Single Bus, detached DMA controller
- Each transfer uses bus twice
 - **I/O → DMA → memory**
- CPU is suspended twice

DMA hardware configurations (2)

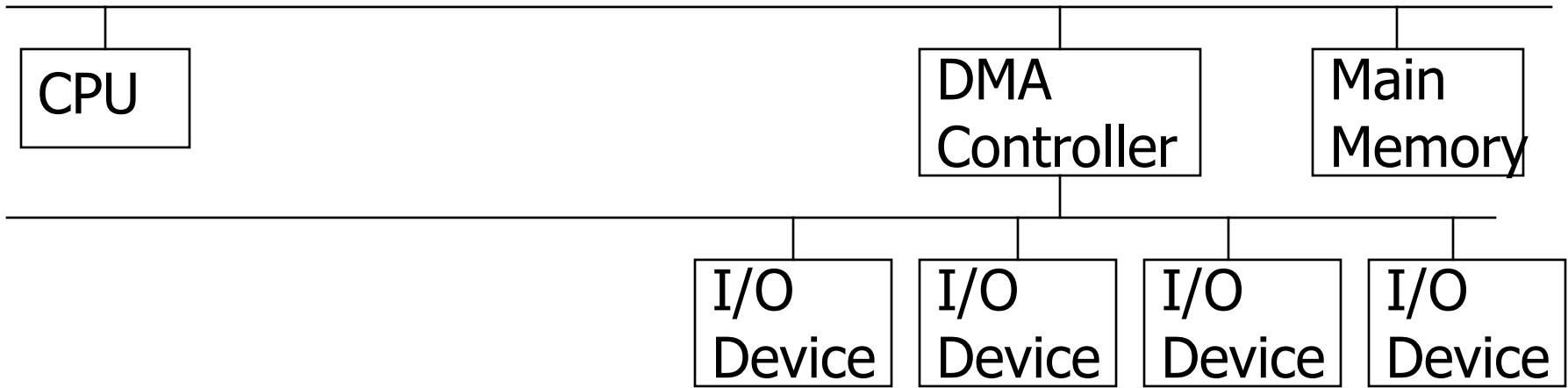
DMA shares bus only with other DMA, CPU, memory



- **Single Bus, Integrated** DMA controller
- Controller may support >1 device
- Each transfer uses bus **once**
 - **DMA → memory**
- **CPU is suspended once**

DMA hardware configurations (3)

DMA has a separate I/O bus

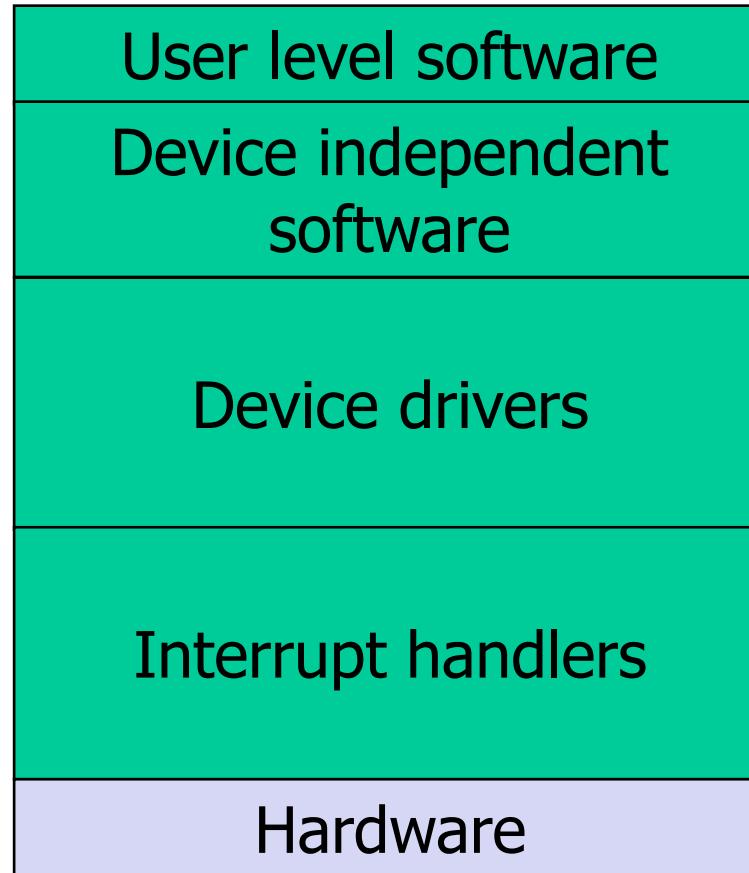


- Separate I/O Bus
- Bus supports all DMA enabled devices
- Each transfer uses bus once
 - **DMA → memory**
- CPU is suspended once

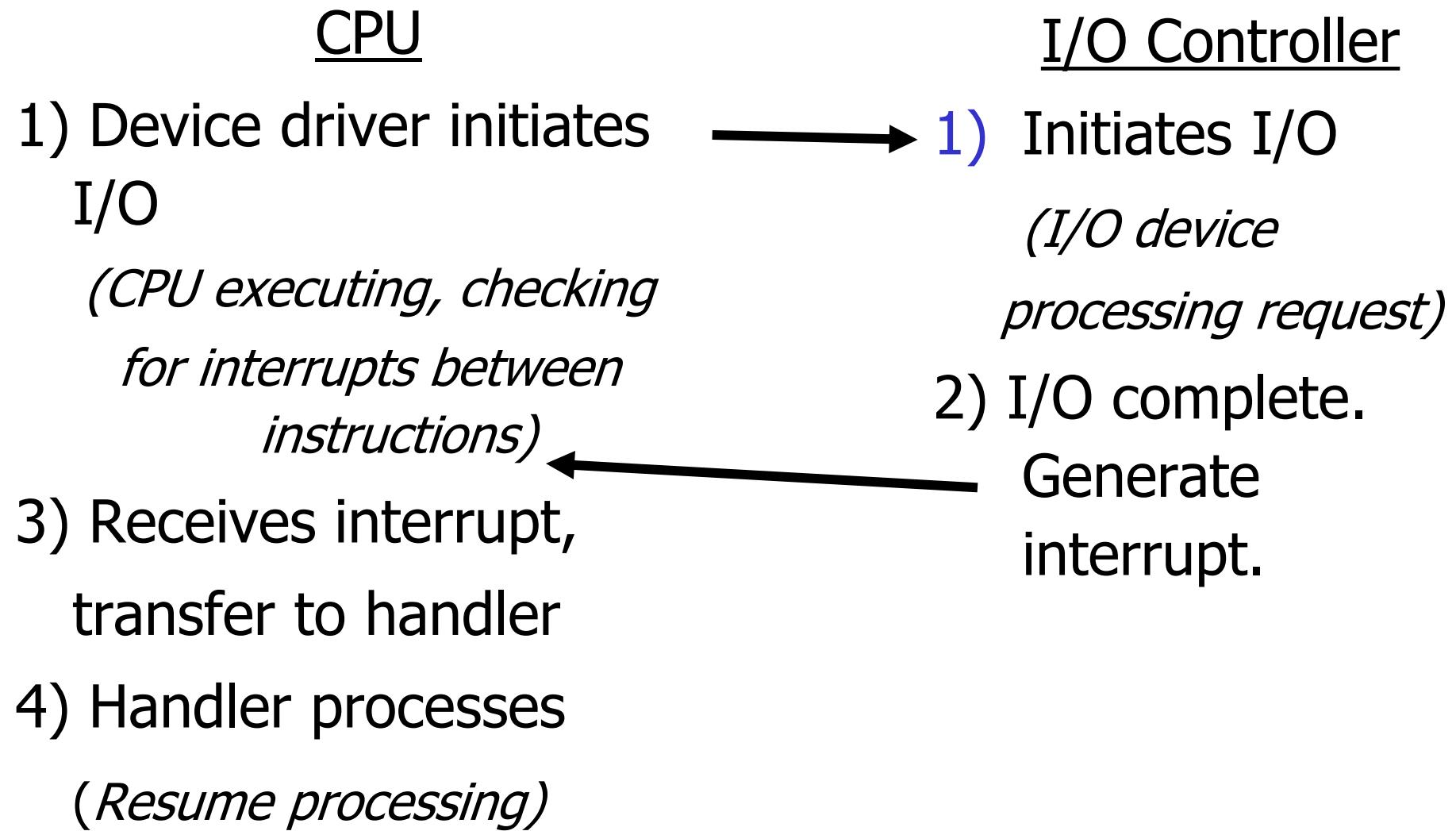
I/O data transfer summary

- Very Old: programmed I/O
 - Controller reads from device
 - OS polls controller for data
- Old: interrupt-driven I/O
 - Controller reads from device
 - Controller interrupts OS
 - OS copies data to memory
- Modern: direct memory access
 - Controller reads from device
 - Controller copies data to memory
 - Controller interrupts OS

I/O software layers



Interrupt handlers



Device interrupt handlers

- Make interrupt handler as small as possible
 - **Split into two pieces**
- **First part:** “Top half”:
 - Interrupts disabled
 - Does minimal amount of work
 - Interrupts re-enabled
 - Main processing deferred until later
- **Second part** does most of the work
 - Windows: “**deferred procedure call**” (DPC)
 - Linux: “**bottom half**” handler
- Implementation-specific
 - 3rd party vendors (part of the driver)

Device drivers

- Device-specific code
 - includes interrupt handler
- Accept abstract requests
 - ex: “read block n”
- See that they are executed by device controller
 - Interact with registers
 - Issue commands
 - Wait for interrupts
- Perform error checking
- Pass data to device-independent software

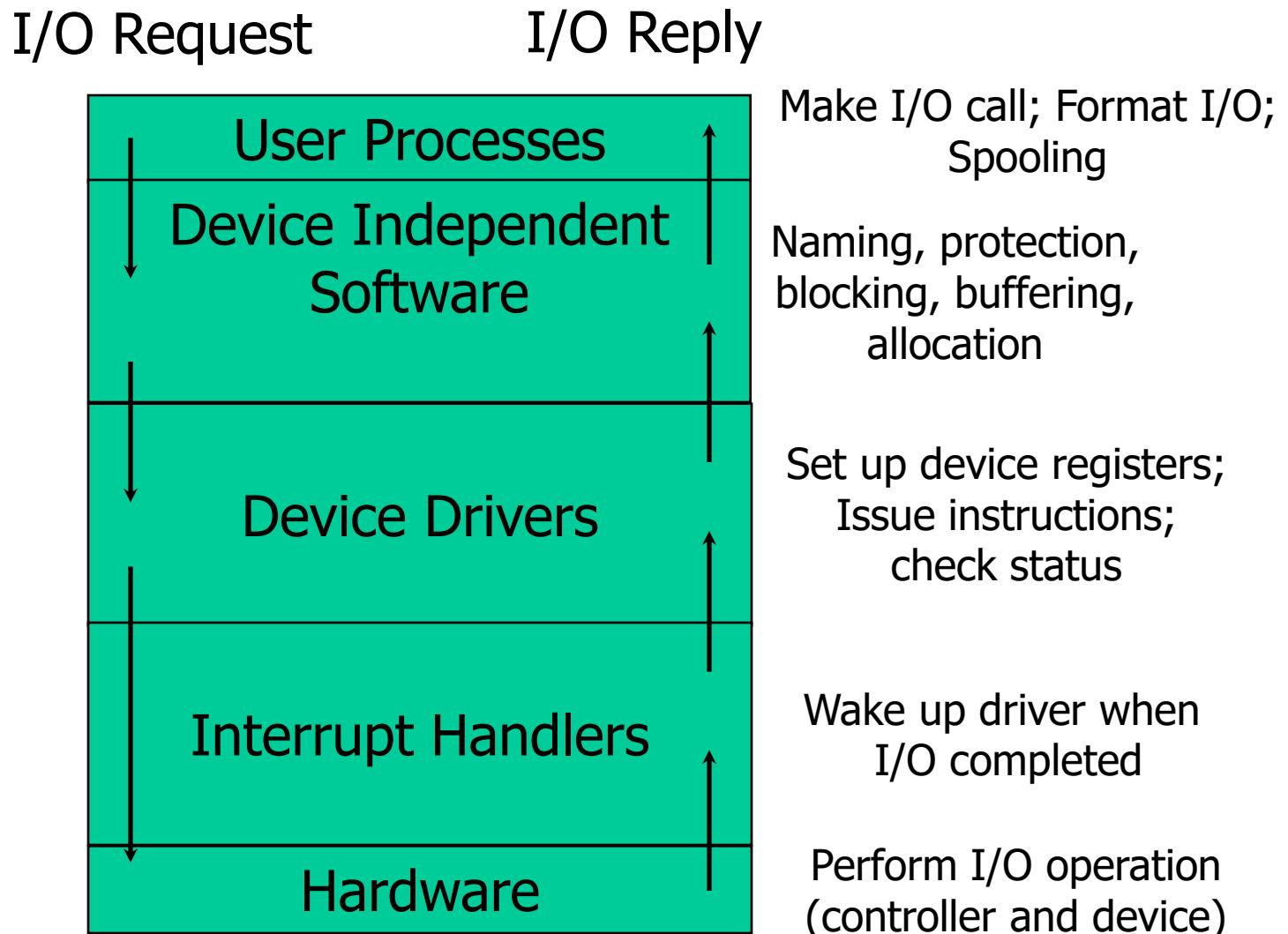
Device-independent I/O Software

- I/O functions common to all devices:
 - uniform interface (API) to rest of OS
 - naming and device allocation/protection
 - device-independent block size
 - buffering
 - error reporting
- Exact boundary is system-dependent
 - sometimes inside drivers for efficiency

User-space I/O software

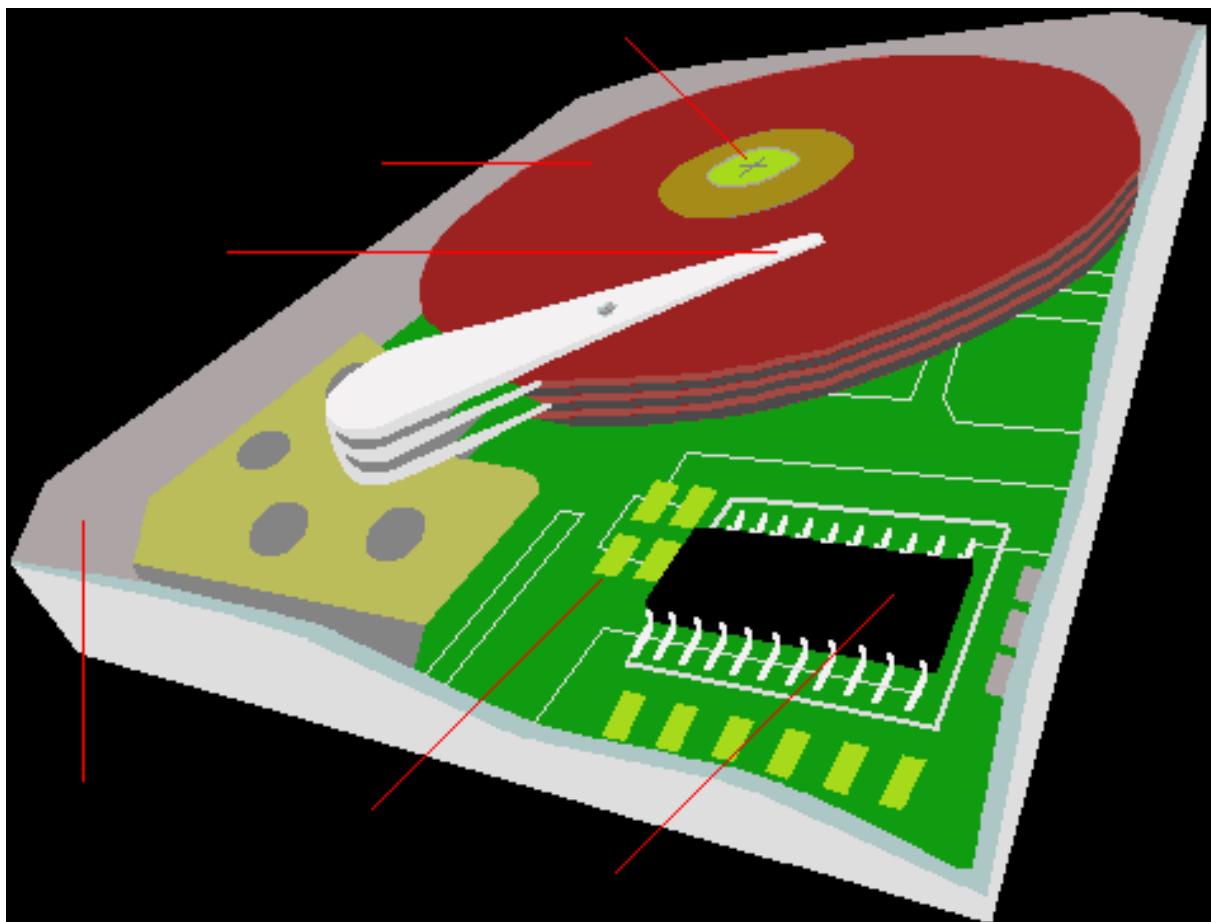
- Library routines, e.g. `n = write(fd, buffer, bytes);`
 - put parameters in place for system call
 - formatting, e.g. `printf()`, `scanf()`, `gets()`
- Spooling
 - spool directory
 - daemon (program or process that sits idly in the background until it is invoked to perform its task - from Greek mythology meaning "guardian spirit")
 - e.g.: print spoolers, e-mail handlers, network transfers, schedulers

I/O software layers summary



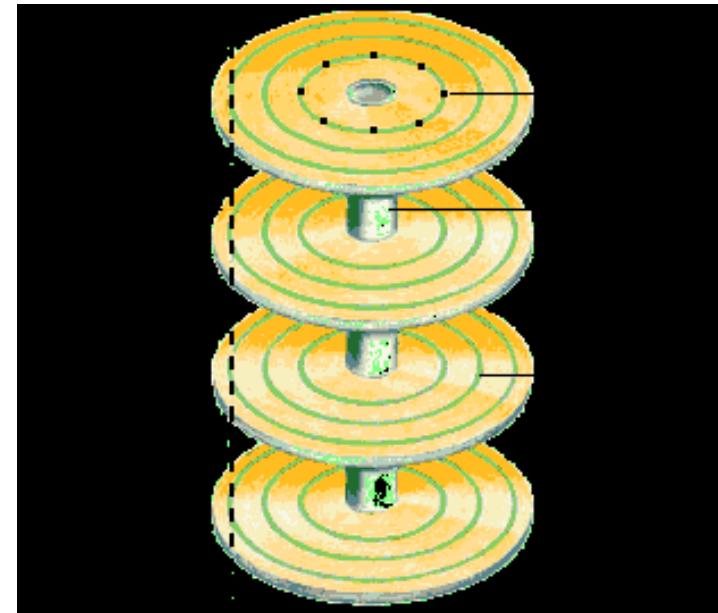
Hard disk drives (HDD)

- Controller often on disk
- Cached to speed access



HDD

- Platters
- Tracks
- Sectors
- Disk arms all move together
 - Cylinders (track x platter)

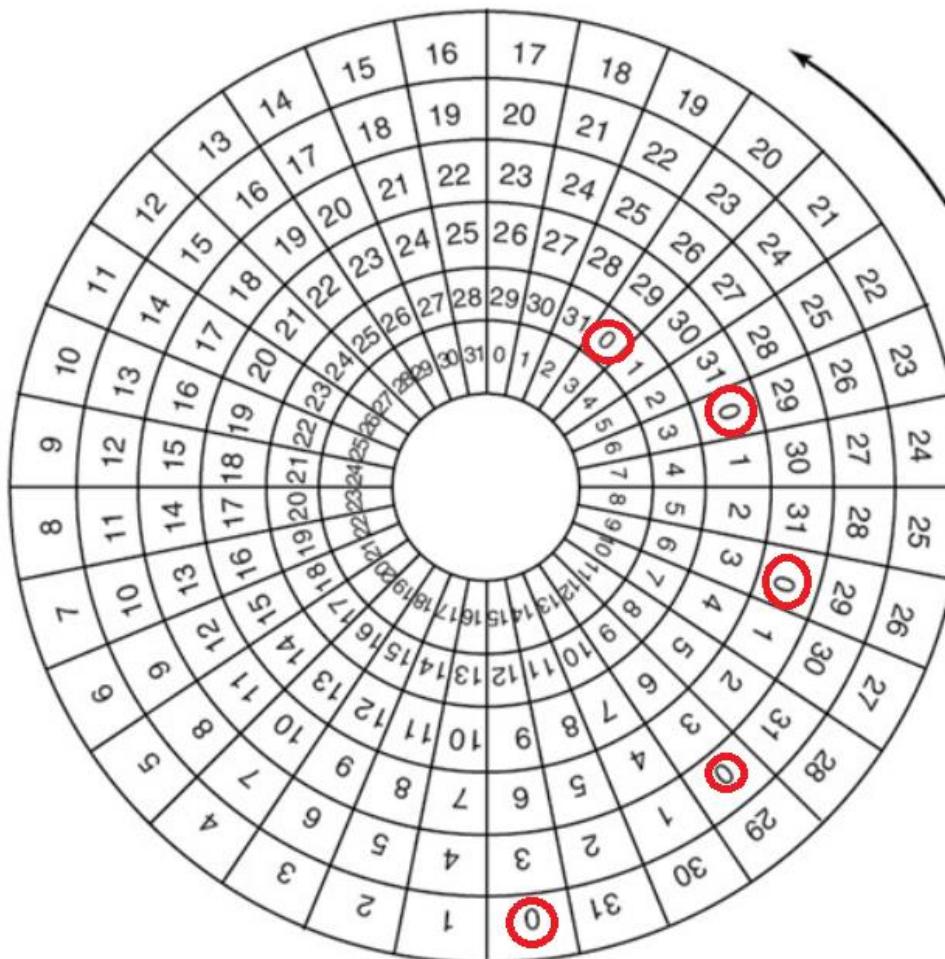


E.g. 4TB Hitachi Ultrastar 7K4000 3.5" HDD

- five 800GB platters spinning at 7,200 rpm
- 16K cylinders (~8 billion sectors, 4KB per sector)
- 16 heads with 8 mS seek time
- 64MB buffer
- SATA 6Gb/s interface
- 2 million hours MTBF (228 years)

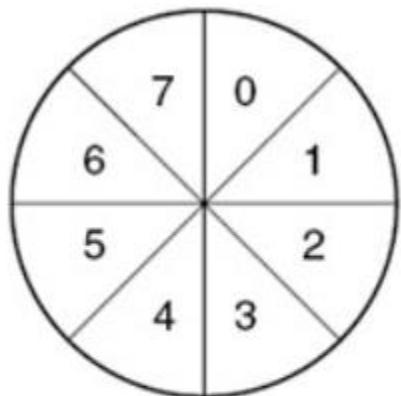
Optimisation 1: cylinder skew

- Allows for time taken to switch cylinders

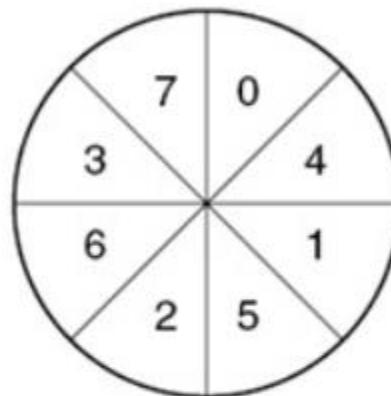


Optimisation 2: sector interleaving

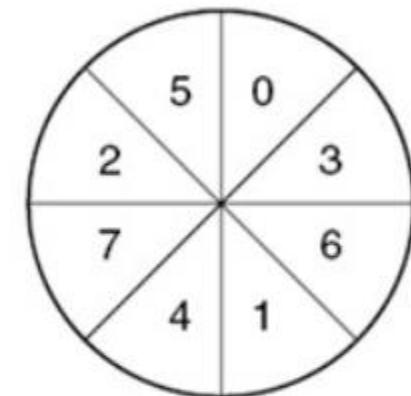
- Allows for time taken to flush buffer



No interleave



single interleave

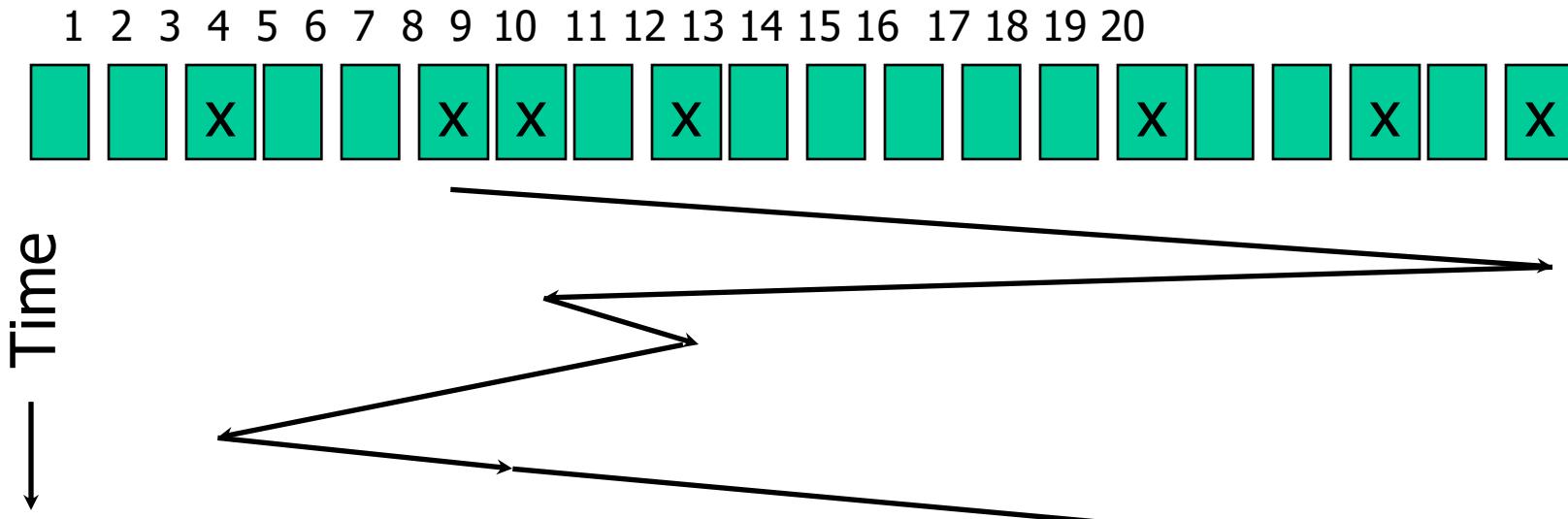


double interleave

Optimisation 3: disk arm Scheduling

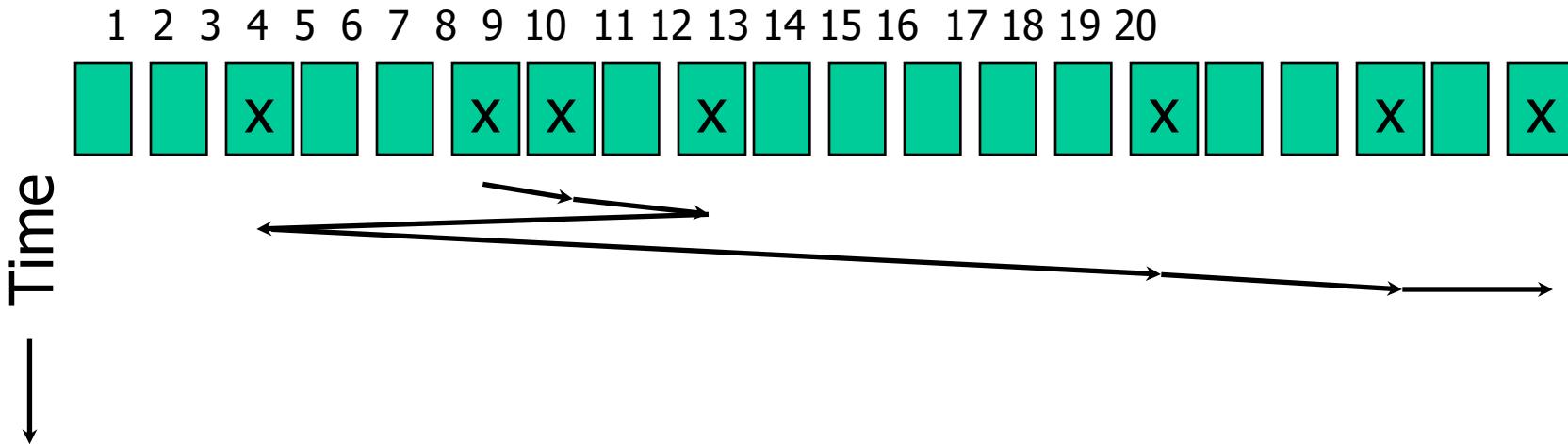
- Read time:
 - seek time (move arm to cylinder)
 - rotational delay (time for sector to rotate under head)
 - transfer time (time to read bits off disk)
- Seek time dominates
- How does disk arm scheduling affect seek?

First come first served (FCFS)



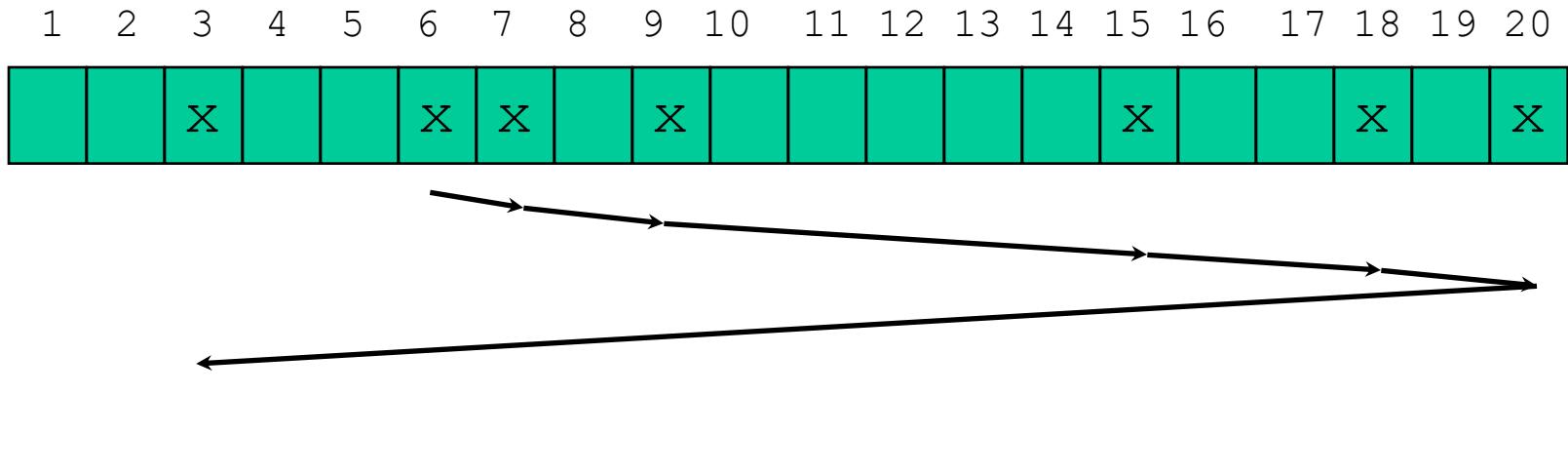
- Service requests in order that they arrive
- $14+13+2+6+3+12+3=53$ cylinder moves
- Little can be done to optimize
 - One option: cache data passing under the head(s)

Shortest Seek First (SSF)



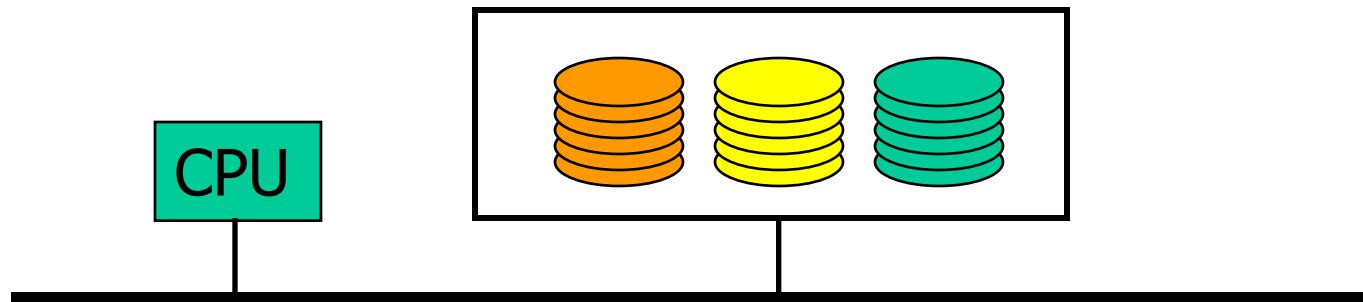
- $1+2+6+12+3+2 = 26$
- Suppose many requests?
 - Stays in middle
 - Starvation!

Elevator (SCAN)



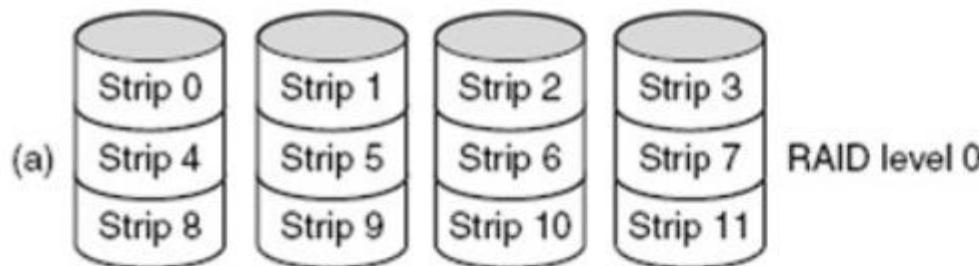
- $1+2+6+3+2+17 = 31$
- Usually a little worse avg seek time than SSF
 - But is more fair, avoids starvation
- C-SCAN (Circular SCAN) has less variance
- Note, seek getting faster, rotational not
 - Someday, change algorithms

Redundant array of independent disks (RAID)

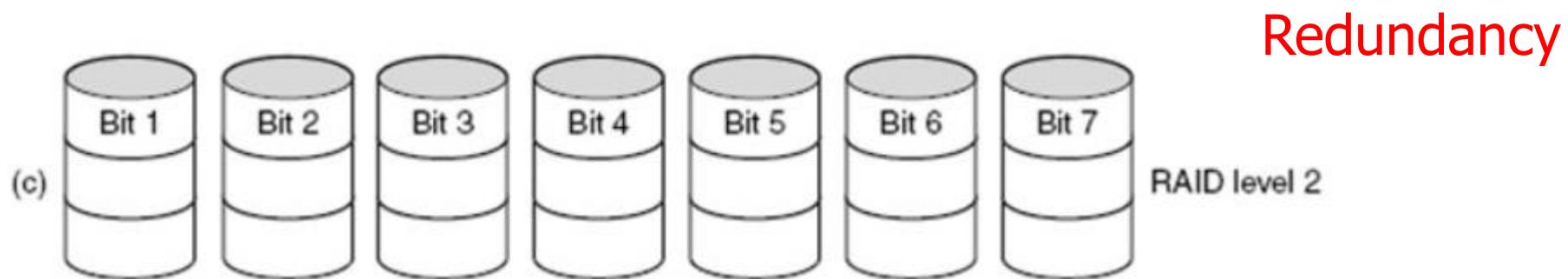
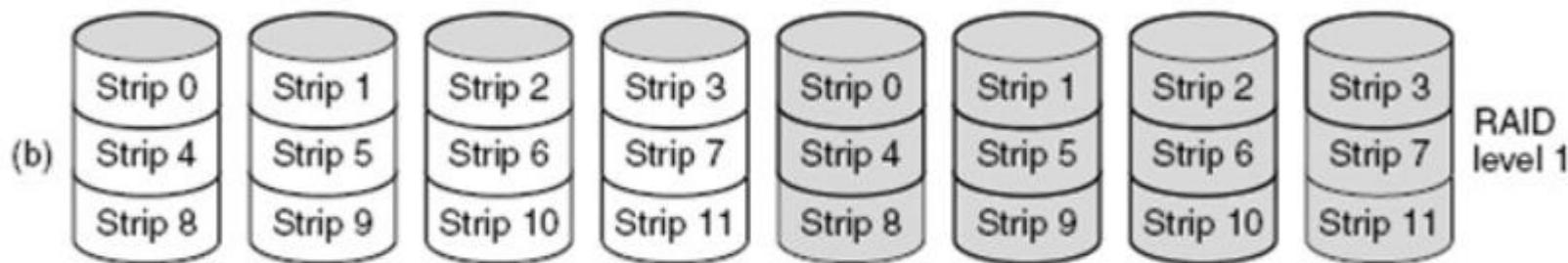


- Typical layout: SCSI controller and disk pack (7 or 15 disks)
- For speed
 - read/write data from multiple drives in parallel
- For fault-tolerance
 - Example: 38 disks: 32 bit word + 6 check bits
 - Spread across multiple disks
 - Redundant copies

Some RAID configurations



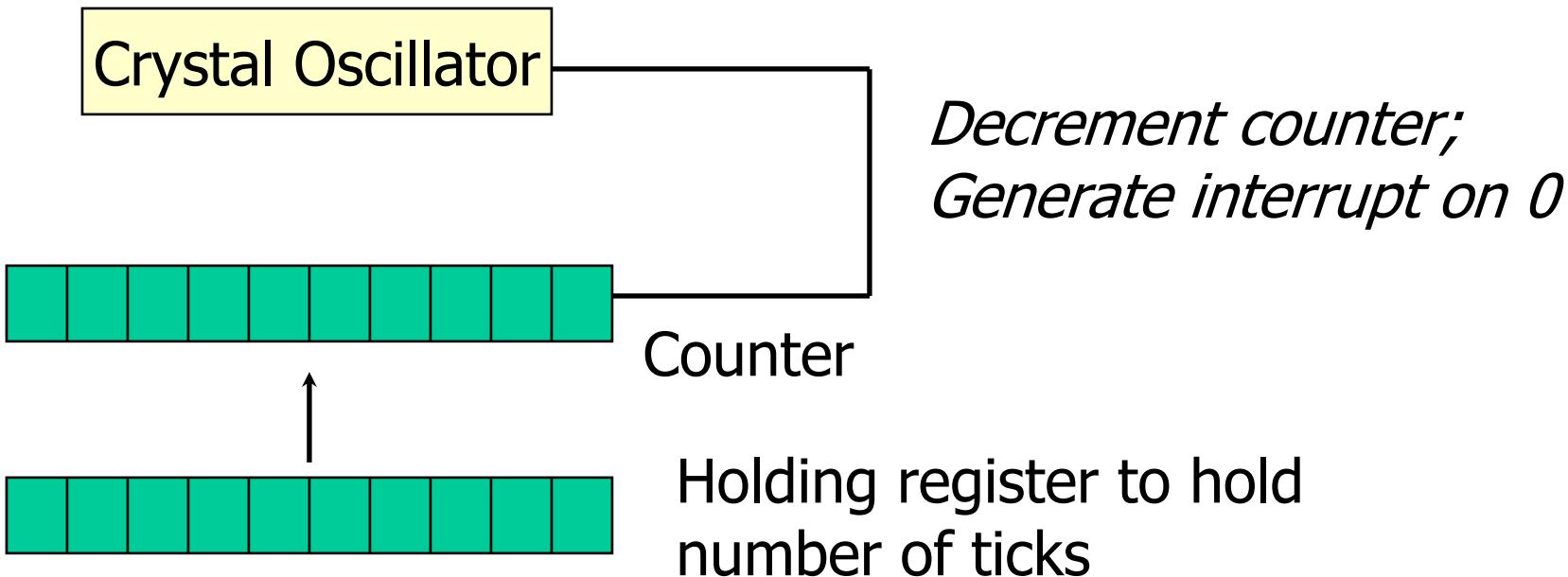
Data sectors “striped”
across RAID disks



Super-parallel (requires synchronised disks)

Clock hardware

- Oscillator generates GHz “ticks”
- Can step down using programmable counters:
 - Time of day (ticks, seconds+ticks, ticks since boot)
 - Timer intervals (scheduler, accounting, alarms)



Performance

- I/O is a major factor in system performance:
 - Demands CPU to execute device driver, kernel I/O code
 - Context switches due to interrupts
 - Data copying
 - Network traffic especially stressful

Improving Performance

- Minimise (by using large transfers, smart controllers, polling) :
 - context switches
 - data copying (multiple buffering)
 - Interrupts
- Use DMA
- Balance CPU, memory, bus, and I/O performance for highest throughput

Input/Output

Example Exam Questions

Example Exam Questions

What percentage of the Linux OS is dedicated to controlling devices?

What are the fastest and slowest I/O devices and what speed?

Example Exam Questions

What percentage of the Linux OS is dedicated to controlling devices?

80-90%

What are the fastest and slowest I/O devices and what speed?

Keyboard @ 10bps,
USB-C Thunderbolt 3 @ 40 Gbps

Example Exam Questions

What information does the CPU pass to the DMA controller?

Example Exam Questions

What information does the CPU pass to the DMA controller?

- Read/Write instruction
- Device address
- Starting address of memory block for data
- Amount of data to be transferred

Example Exam Questions

Describe DMA Transfer Cycle Stealing

Example Exam Questions

Describe DMA Transfer Cycle Stealing

- DMA controller takes over bus for a cycle
- Transfer of one word of data
- Not an interrupt - CPU does not switch context
- CPU suspended just before it accesses bus
 - i.e. before an operand or data fetch or a data write
- Slows down CPU but not as much as CPU doing transfer

Example Exam Questions

Describe three DMA configurations and how many BUS cycles does each use?

Example Exam Questions

Describe three DMA configurations and how many BUS cycles does each use?

- DMA shares bus with I/O units – 2 cycles
- DMA shares bus only with other DMA, CPU, memory – 1 cycle
- DMA has a separate I/O bus – 1 cycle

Example Exam Questions

Why is an Interrupt Handler split into two parts, what are the two parts called in Linux and what do they do?

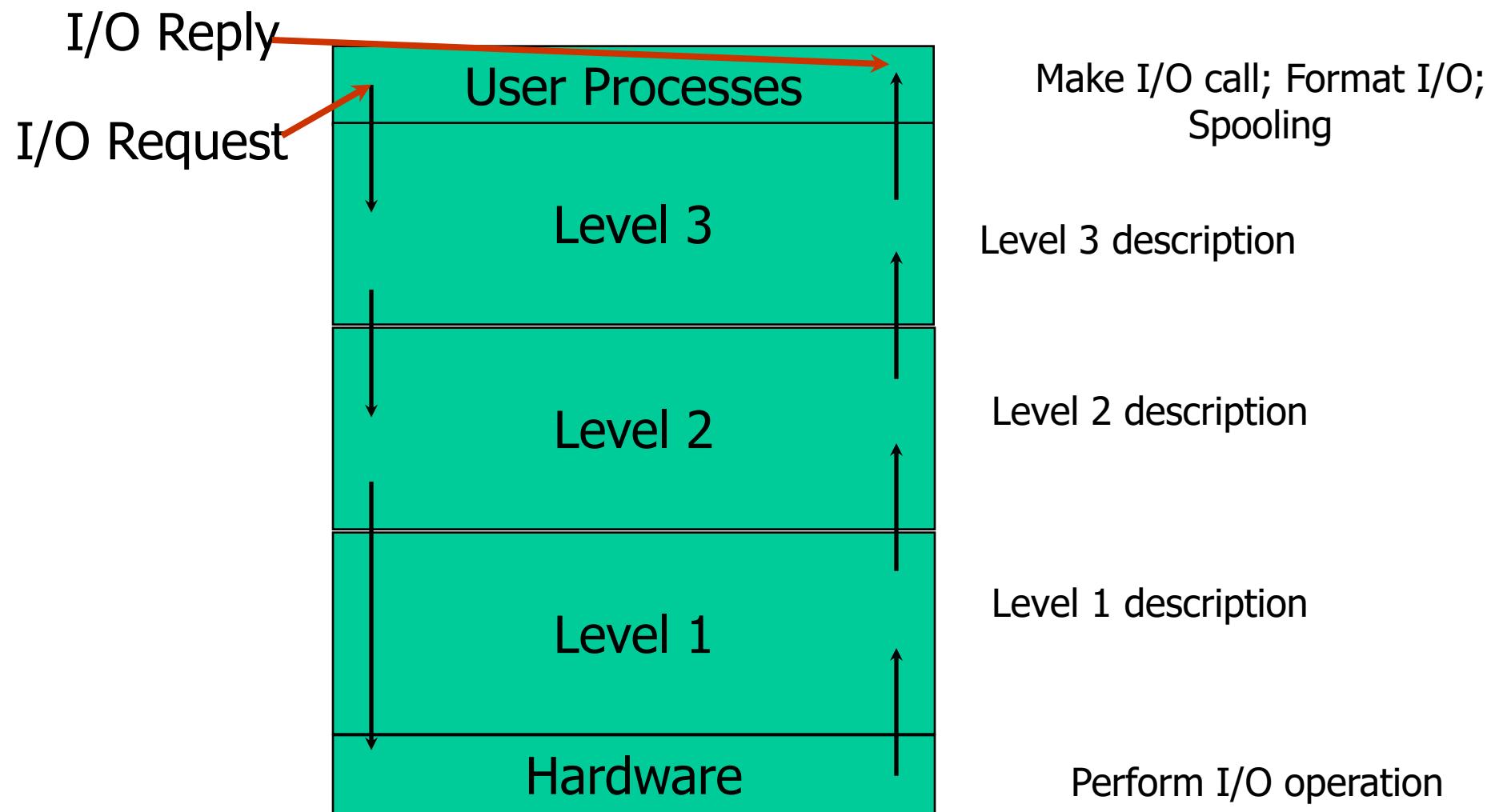
Example Exam Questions

Why is an Interrupt Handler split into two parts, what are the two parts called in Linux and what do they do?

- Within Interrupt Handler, all Interrupts are Blocked
- Interrupt Handlers must finish up quickly so as not to keep interrupts blocked for long
- Top half: the function actually responds to the interrupt – within the Interrupt Handler
- Bottom-half (Linux) / DPC (Windows): a routine that is deferred by the top half to be executed later, at a safer time - during bottom-half, all interrupts are enabled

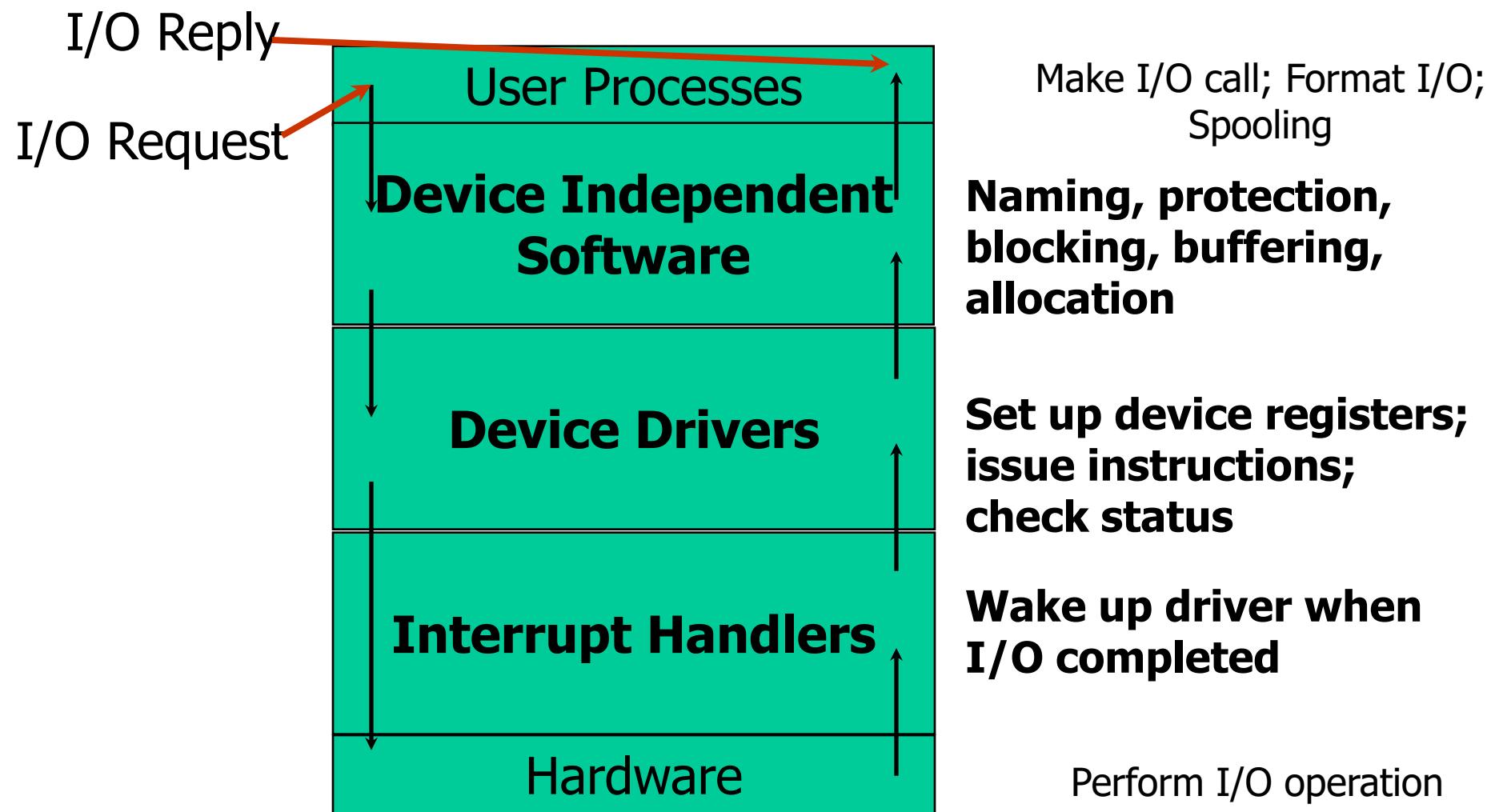
Example Exam Questions

Label levels one to three
with a brief (few words) description of each level.



Example Exam Questions

Label levels one to three
with a brief (few words) description of each level.



• Parallel Bus Interfaces

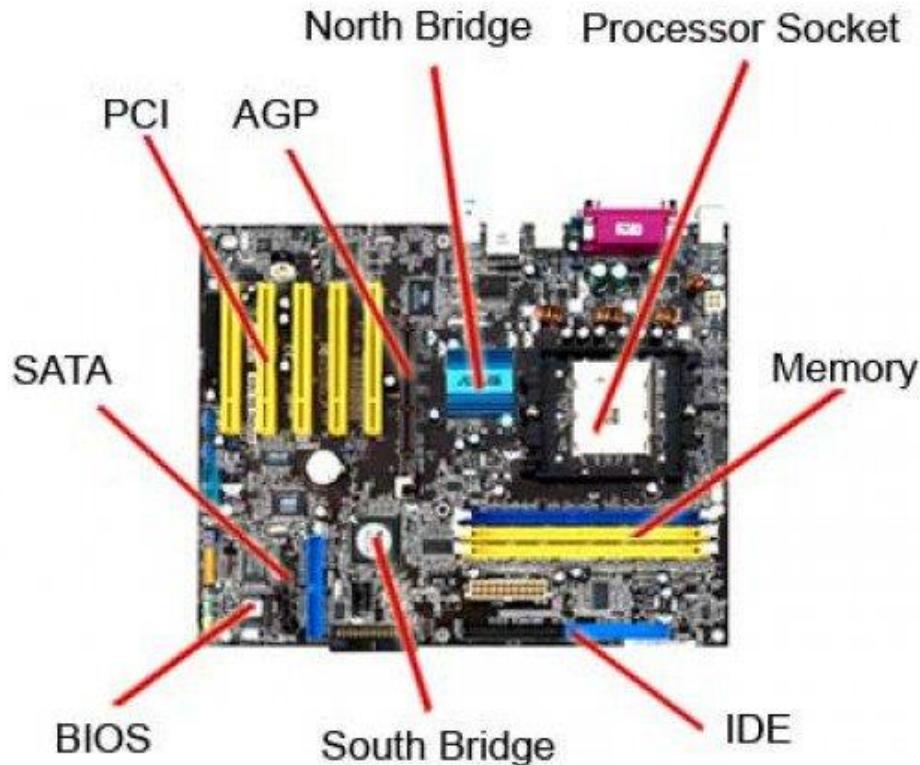
history:

- EISA 33 MBps
- ATA 167 MBps
- SCSI 320 MBps
- Express Card (was PC Card, which was PCMCIA) 400 MBps

now:

- AGP 2.1 GBps
- ~~PCI Express 4.0 31GBps (2017)~~

High speed control



Low speed control

* Multi-lane serial,
all based on PCI Express

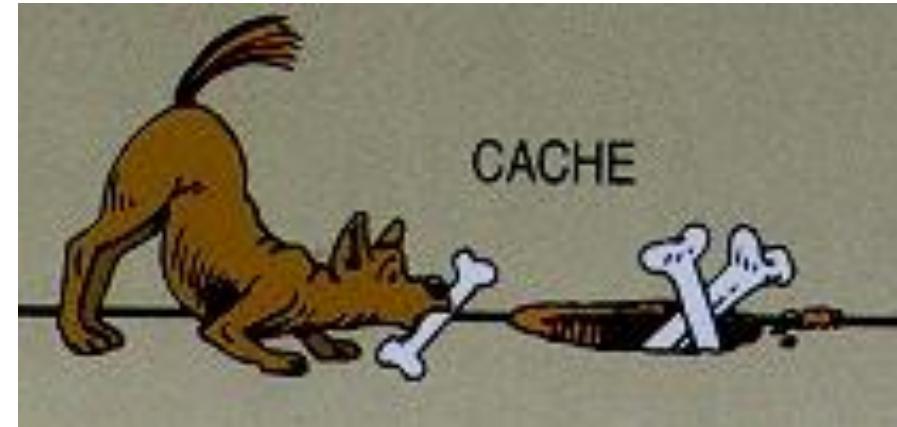
• Serial Bus Interfaces*

- ~~eSATA SATAe~~: 16 Gbps
- USB-C Thunderbolt 3: 40 Gbps
- PCI Express 6.0 7.8-126 GB/s (x16)

ENCE360

Operating

Systems

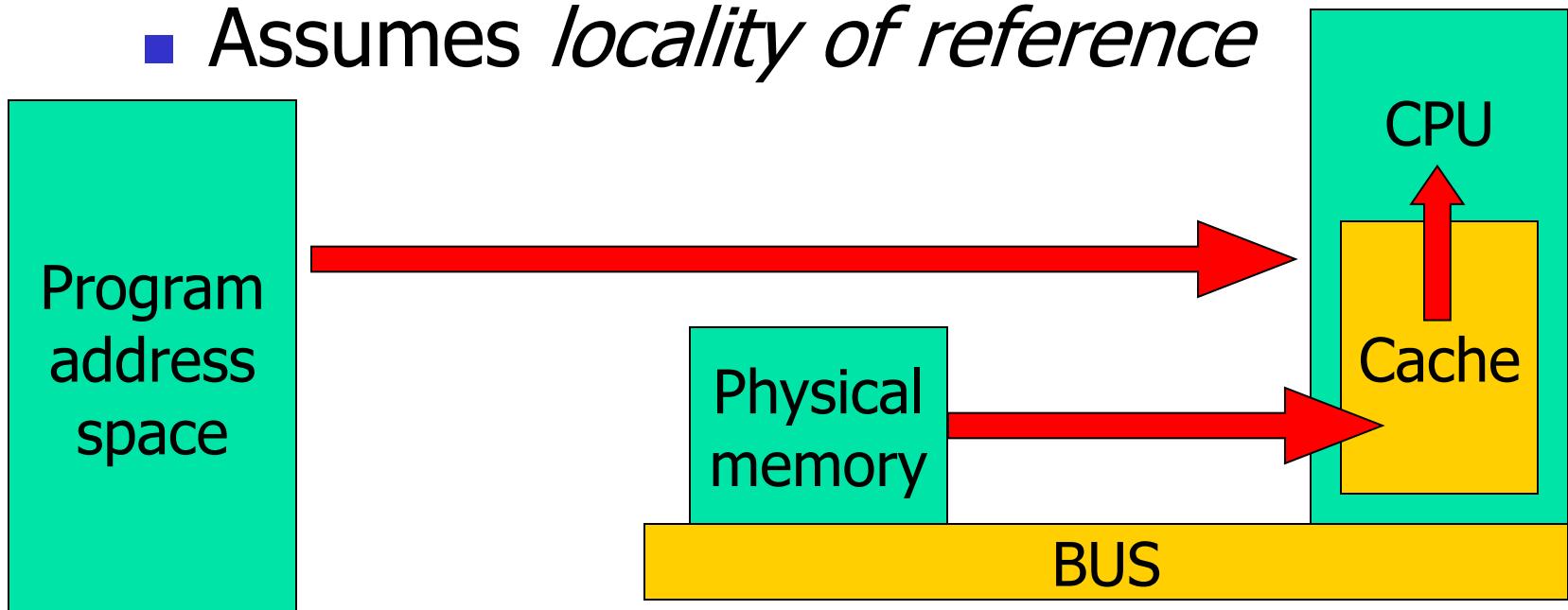


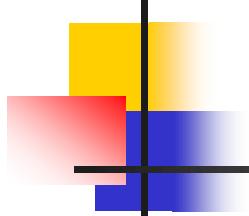
Memory Management

*Caching for
Faster memory*

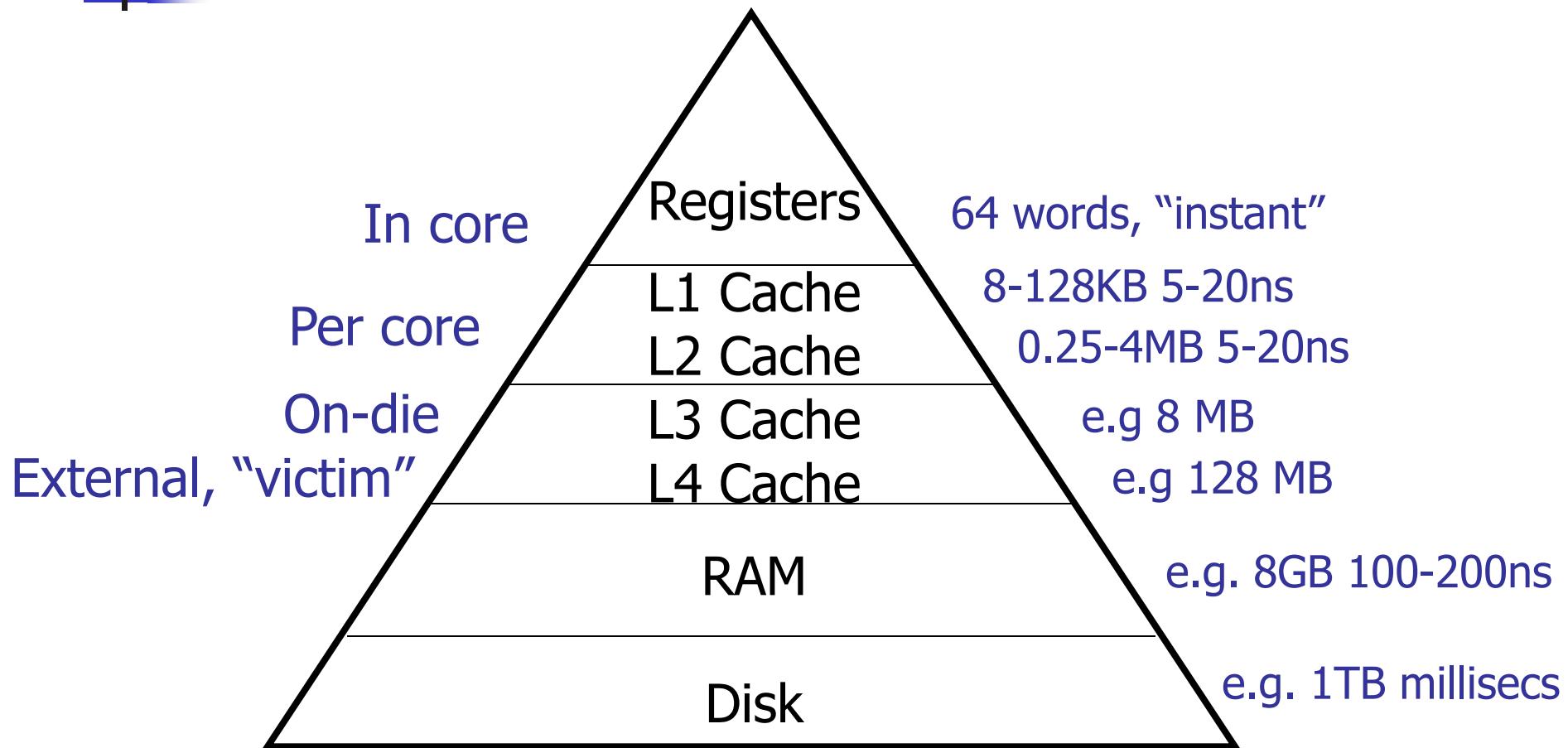
Caching

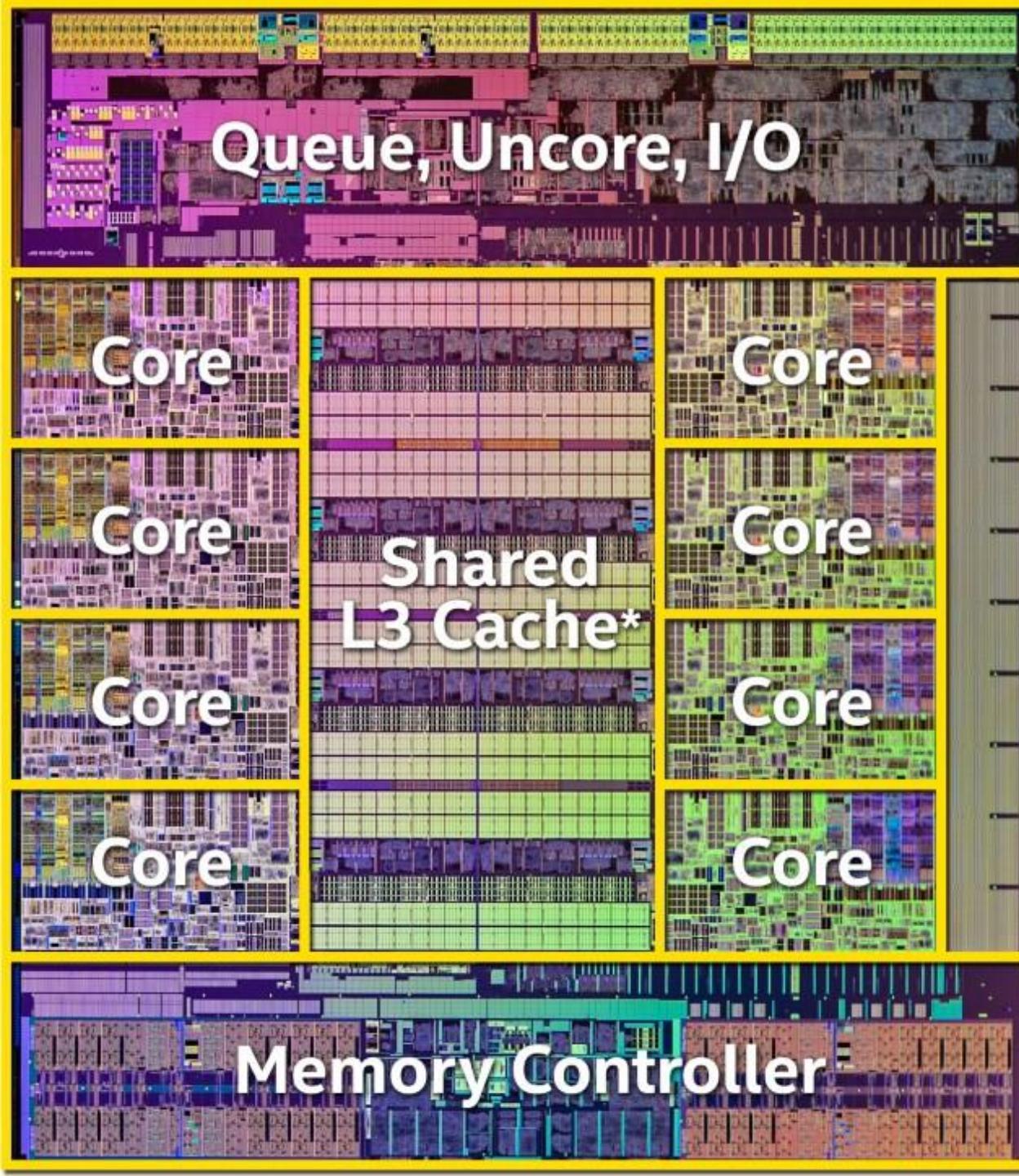
- Memory *much* slower than processor
- Put faster “cache” close to CPU
- Assumes *locality of reference*





Memory hierarchy





Queue, Uncore, I/O

Core

Core

Core

Core

Shared
L3 Cache*

Core

Core

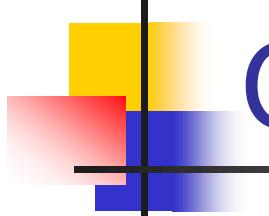
Core

Core

Memory Controller

Intel
Haswell
(2013)

L1, L2 caches in
cores



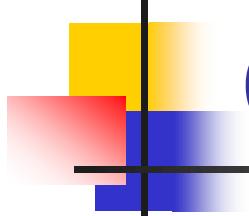
Cache entries

- Cache *line*: fixed length (4 to 64 bytes)
- Valid bit
- Cache tag: identifies address range
- Line contents:

Valid?	Dirty?	Tag	Data 1	Data 2	Data 3	Data 4
--------	--------	-----	--------	--------	--------	--------

- Address (bits):

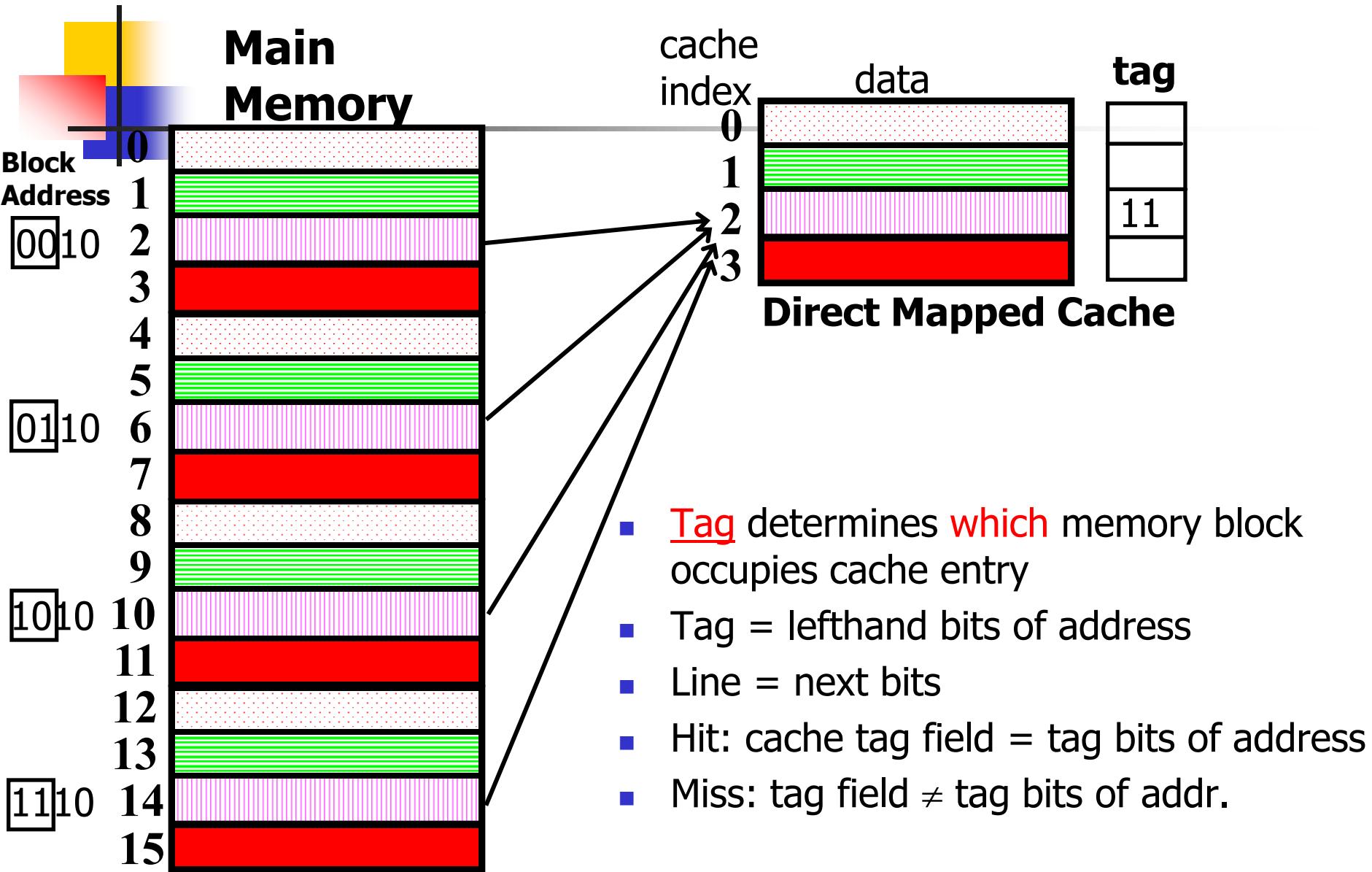
Tag	(Line)	Byte offset
-----	--------	-------------

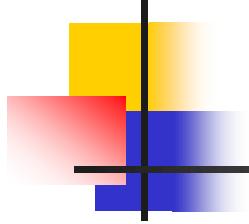


Cache flavours

- Direct mapping
 - Address indicates where it is (fast!)
- Associative
 - Any cache line for any address
 - Address identified by tag
- Set associative
 - Combination of the two

Direct mapping





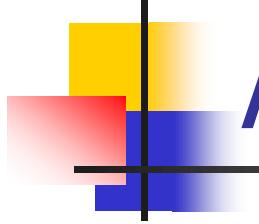
Direct mapping pros and cons

- Pros:

- FAST! Direct access to the cache line
 - Simple

- Cons:

- Can be poor use of space
 - Potential for thrashing

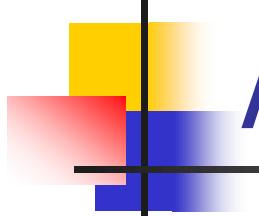


Associative caches

- Cache entry is “associated” with address by the tag
 - Search all entries, OR
 - Comparator circuit for each cache line

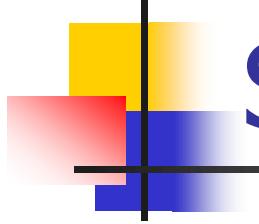
Tag	Byte
001101	01

Tag	D0	D1	D2	D3
000010				
101001				
001101				
001110				



Associative cache

- Pros:
 - Maximises cache use
 - Minimises collisions
- Cons
 - Slow (compare each entry), OR
 - Complex/expensive (comparator hardware)



Set-associative

- Combination of direct and associative
- Direct-mapped set of smaller associative caches
- *N-way* cache: n associative entries per direct address map
 - Typically 2, 4 or 8-way

Direct Mapped Cache Fill

Main
Memory

Index
0
1
2
3
4
5
...

Cache Memory
Index 0
Index 1
Index 2
Index 3

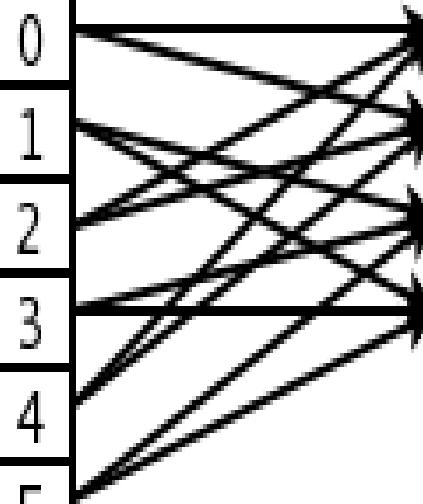


2-Way Associative Cache Fill

Main
Memory

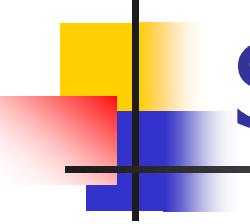
Index
0
1
2
3
4
5
...

Cache Memory
Index 0, Way 0
Index 0, Way 1
Index 1, Way 0
Index 1, Way 1



Each location in main memory can be
cached by just one cache location.

Each location in main memory can be
cached by one of two cache locations.



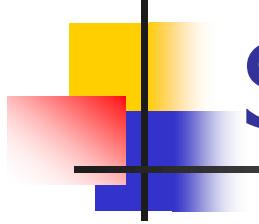
Set-associative example

	V	Tag	Data		V	Tag	Data
7		0011				0101	
6		1010				1100	
5		1101				0001	
4		0101				0010	
3		1111				1001	
2		0000				0101	
1		1010				1111	
0		1101				0000	
Entry A				Entry B			

e.g. 8 line cache, 4 bytes per cache line: tag=4 bits, line=3 bits, byte=2 bits

1000 011 01 – look in line 3 for 1000, MISS

0001 101 11 – look in line 5 for 0001, found in entry B. Read, return byte 3



Set-associative pros and cons

- Pros

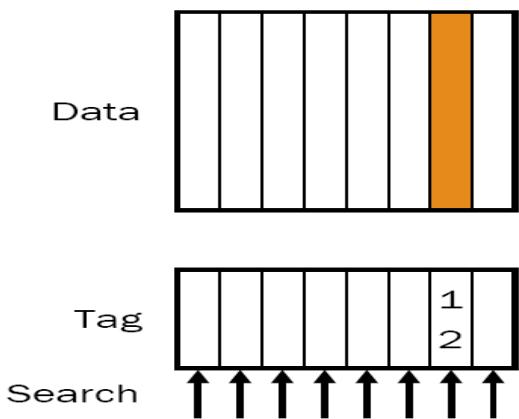
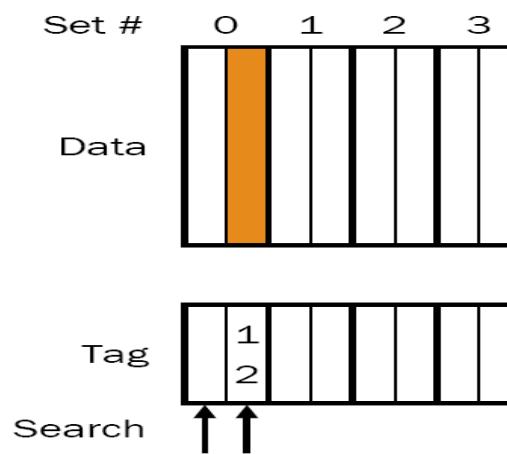
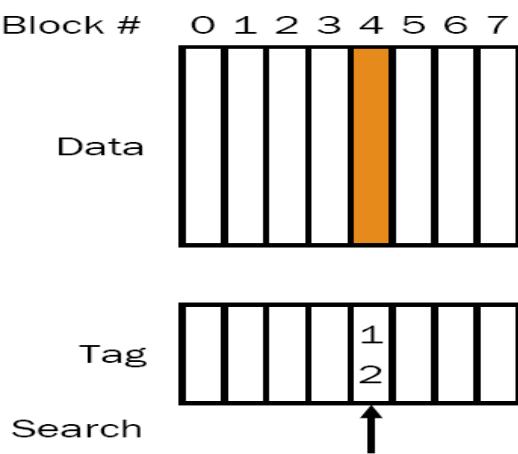
- Higher hit rate
 - (8-way = 60% lower miss rate)
- Easier to build / faster than fully-associative

- Cons

- Slower access (than direct mapped)

Direct Mapped (D) vs Fully Associative (A) vs Set Associative (SA) Caching

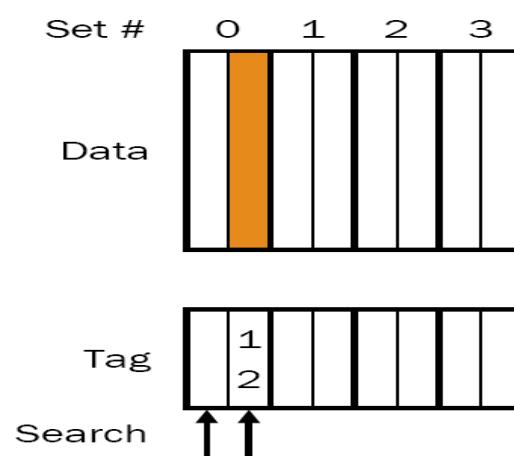
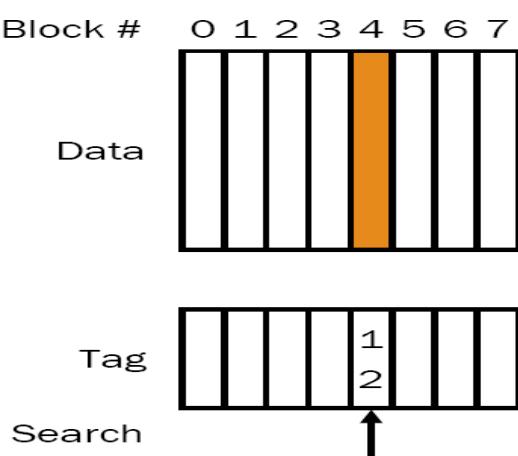
- **D:** each address can be stored in one cache location
 - address = memory block (tag) + cache line (+ offset in line)
- 2 way **SA:** each address stored in 2 cache locations
 - address = memory block (tag) + set number (+ offset in line)
- **A:** any address can be stored in any cache location
 - address = memory block (tag) (+ offset in line)



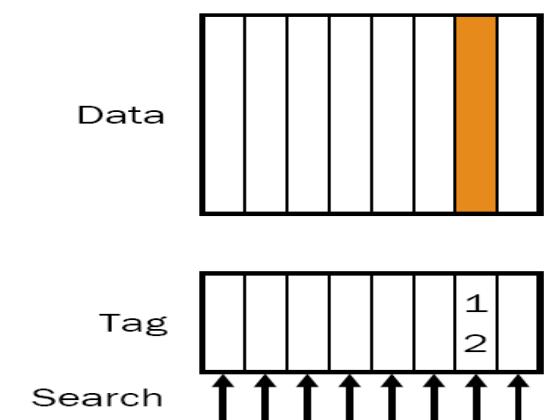
Direct/Set/Associate Relationship

- Conceptually all are set-associative:

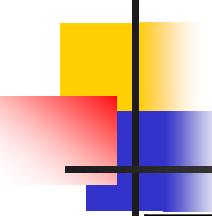
- Set size = 1: **direct mapping**
- Set size = number of cache entries: **fully associative**
- $1 < \text{Set size} < \text{number of entries}$: **set-associative**



Set number = line



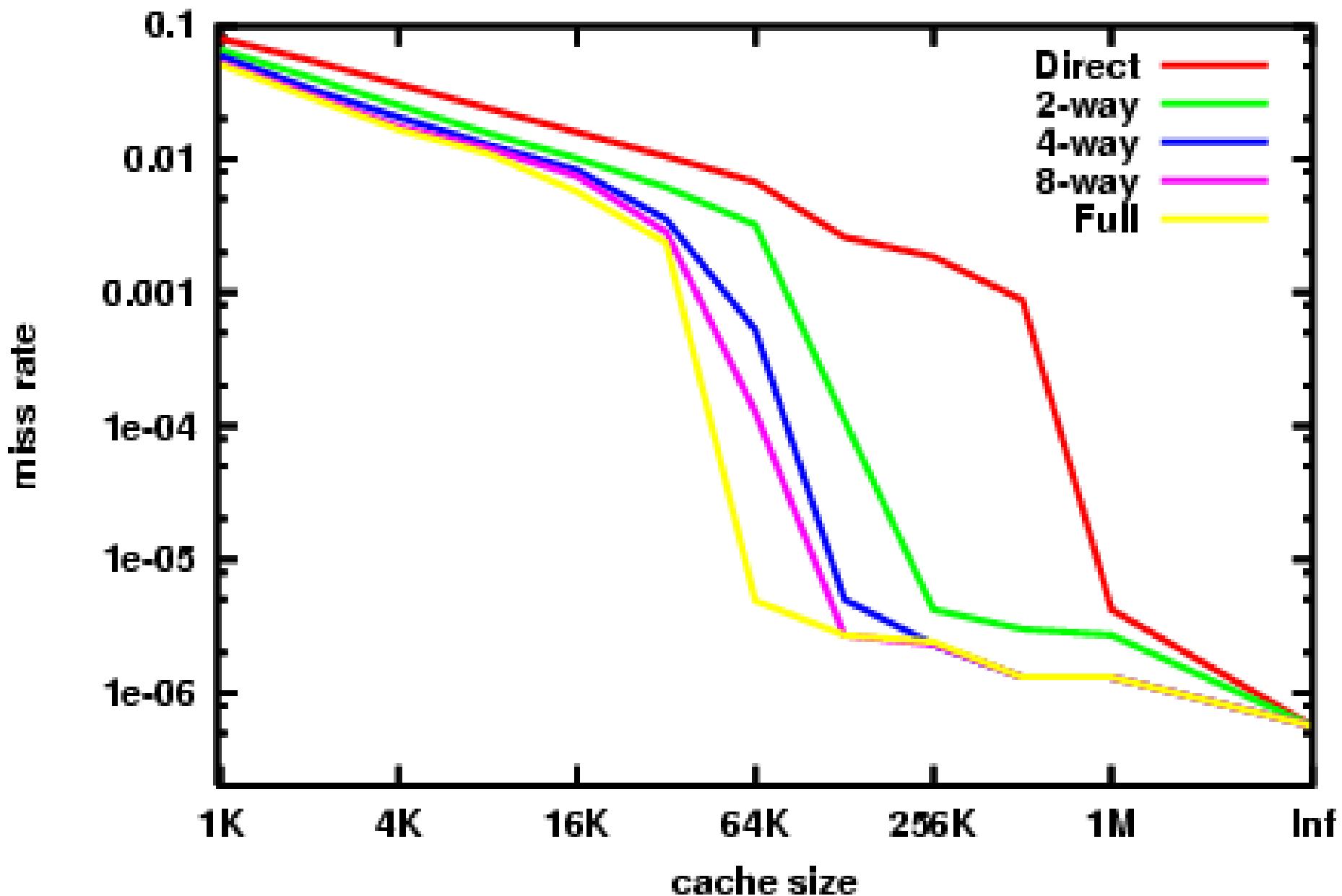
No set number



Cache Performance tradeoff

Cache type	Hit ratio	Search speed
Direct mapped	Good	Best
Fully associative	Best	Moderate
N-way set associative, $N > 1$	Very good, better as N Increases	Good, worse as N increases*

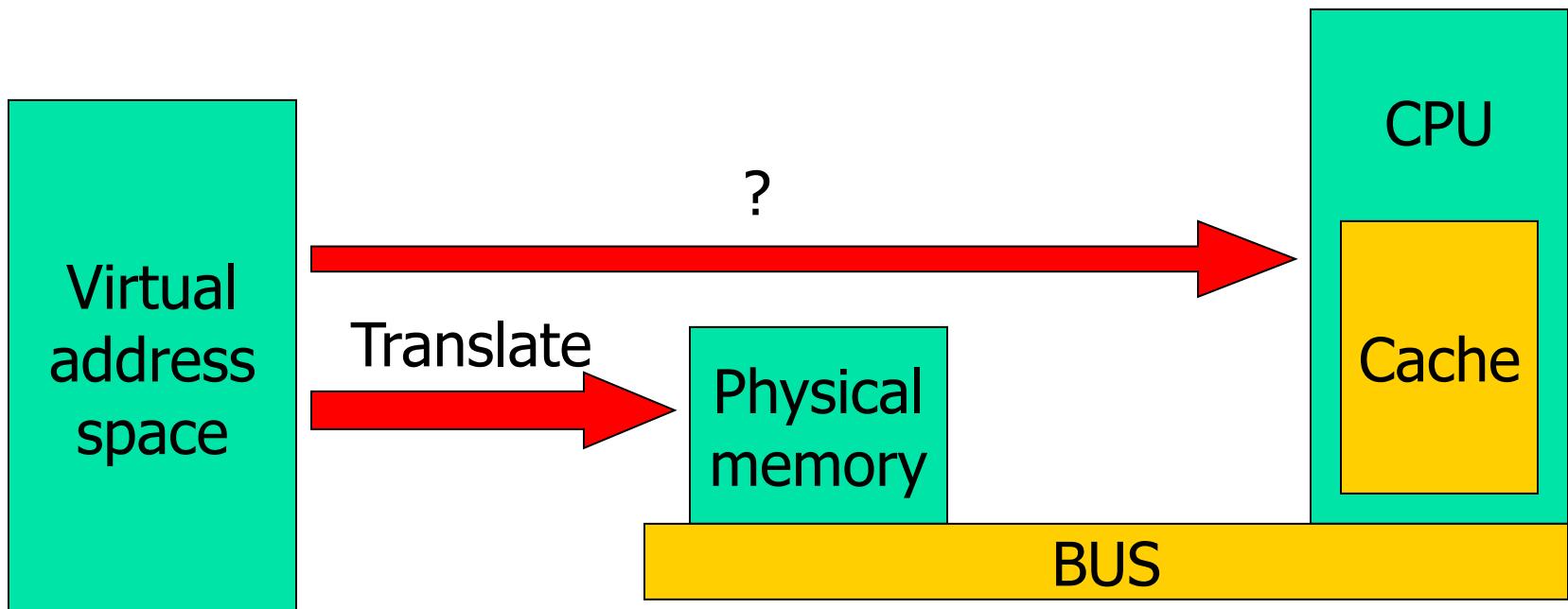
- Direct mapped and set-associative most common
- E.g. Haswell processors:
 - L1, L2: 8-way 64 Bytes per line
 - L3: direct mapped 64 Bytes per line



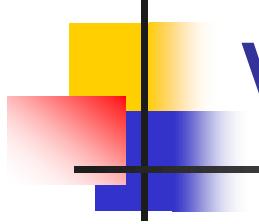
Miss rate versus cache size on the Integer portion of SPEC CPU2000

A clever compiler can take cache collisions into account, placing instructions and data in memory.

Complication: virtual memory

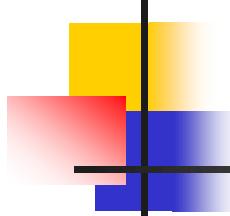


Max 64 bit number = 18, 446,744, 073,709, 551,615
As a memory address, access 18 million Terabytes of RAM
(= 18,000 Petabytes = 18 Exabytes)



Virtual memory

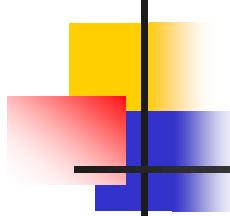
- Programmer access *virtual* addresses
- **Hardware** translates into *physical* addresses
 - Memory management unit (MMU)
- *Bus* retrieves the data from physical memory
- *Bus transaction* moves 1 or more bytes of data in from memory



Cache position

- Physical vs Virtual addressing:
 - Physical:
 - address translated first (slower)
 - *Context* not required (unique)
 - Virtual:
 - In parallel with address translation
 - But: context required (non-unique)
 - Hybrid, e.g.: virtual lookup, physical tag
 - Entry(s) fetched from cache using virtual address
 - Later checked for hit (once address translated)



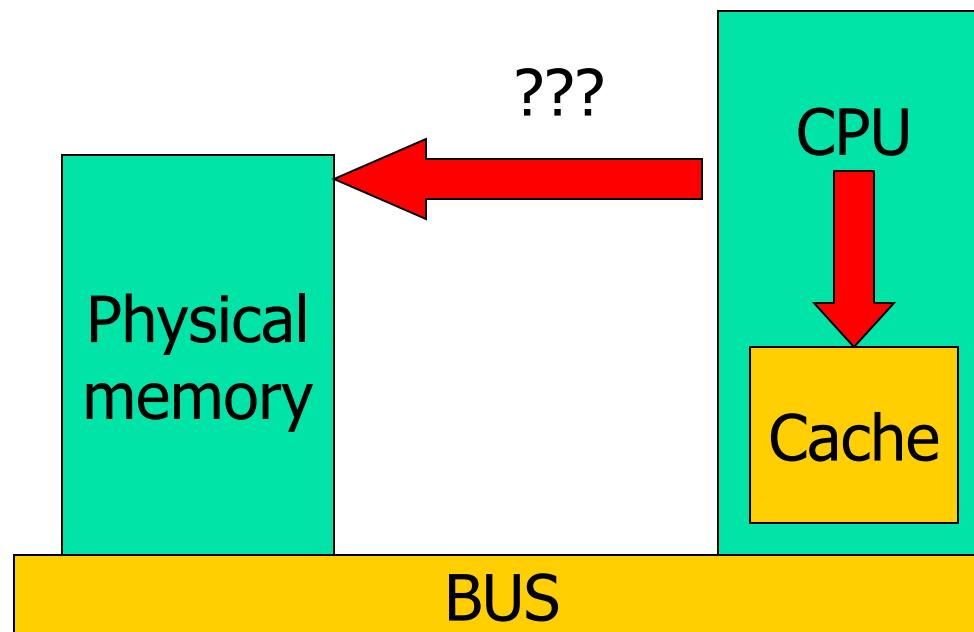


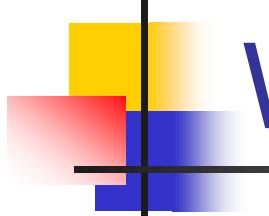
Replacement policy

- Cache miss “evicts” an old entry
- Which cache entry to evict?
 - Direct mapping: no issue
 - Associative/set associative:
 - RANDOM
 - FIFO
 - Least recently used (LRU)
 - Least *frequently* used

And then it all went horribly wrong...

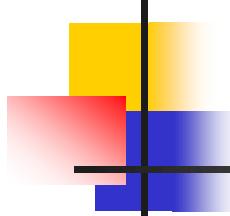
- Policy needed for writing to memory
 - Consistency – cache and memory “agree”
 - Speed – multiple writes





Write policies

- **Writing data in cache:**
 - **Write-through:** always write to memory (and cache)
 - **Copy back/write-back:**
 - write to memory when evicting (even if unchanged)
 - **Write-deferred:**
 - Mark updated cache entry as “dirty” (if changed)
 - Write out dirty entry on eviction
- **Writing data not read:**
 - **Write allocation (write before read)**
 - Bring missed entry into cache
 - (vs just write in memory)



Prefetching

- Load next n lines of memory into cache
 - e.g. video
- May write over needed data
- Memory traffic increases
- Processor may be idle during fetch
- Clever hardware can make worthwhile
 - Specialist caches e.g. instruction trace cache

Optimising cache hits (1): Data structure reorganisation

- Maximise cache hits by keeping data accesses contiguous:

```
typedef struct{  
  
    float x,y,z;  
  
    int a,b;  
  
} Vertex;  
  
Vertex Vertices [NumVs] ;
```

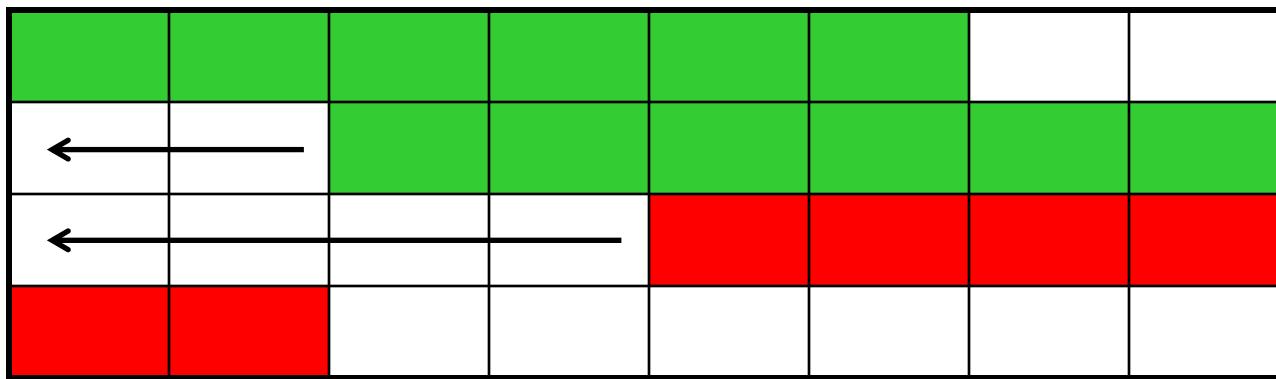


```
typedef struct{  
  
    float x [NumVs] ;  
  
    float y [NumVs] ;  
  
    float z [NumVs] ;  
  
    int a [NumVs] ;  
  
    int b [NumVs] ;  
  
} VerticesList;  
  
VerticesList Vertices;
```

All values of x now contiguous

Optimising cache hits (2): Data Alignment

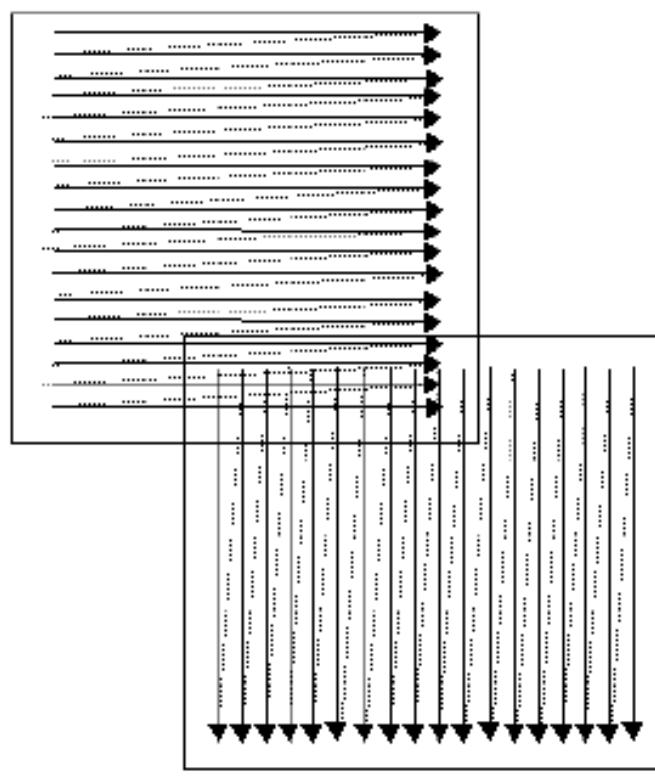
- Try to keep data within 1 line of cache:



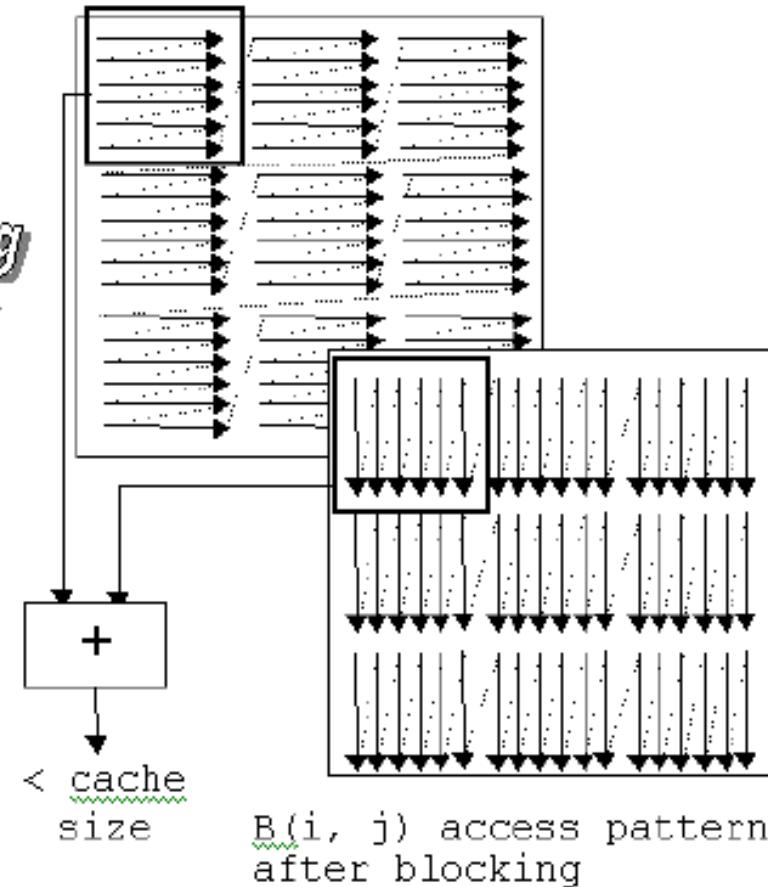
- Organise data to fit on a cache line
 - maximises cache hit rate

Optimising cache hits (3): Loop Blocking

$A(i, j)$ access pattern



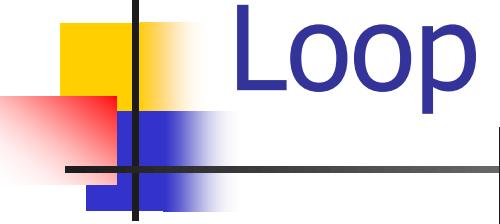
$A(i, j)$ access pattern
after blocking



$B(i, j)$ access pattern

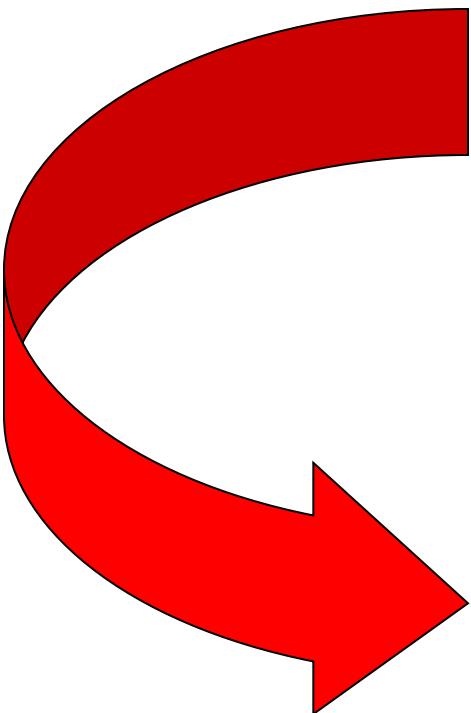
$B(i, j)$ access pattern
after blocking

Organise data access “span” to fit in cache

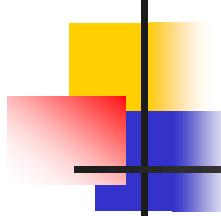


Loop Blocking (2)

```
float A[MAX, MAX], B[MAX, MAX]
...
for (i=0; i< MAX; i++) {
    for (j=0; j< MAX; j++) {
        A[i,j] = A[i,j] + B[j, i];
    }
}
```

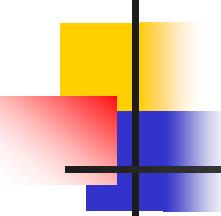


```
float A[MAX, MAX], B[MAX, MAX];
for (i=0; i< MAX; i+=block_size) {
    for (j=0; j< MAX; j+=block_size) {
        for (ii=i; ii<i+block_size; ii++) {
            for (jj=j; jj<j+block_size; jj++) {
                A[ii,jj] = A[ii,jj] + B[jj, ii];
            }
        }
    }
}
```



Summary

- Most modern processors/operating systems use caches, usually at least two levels (typically four)
- Cache design is complex
- Design affects performance
- Need to know cache design to optimise code



Example Exam Question

The following is a fragment of a **16KByte, 2-way associative L1 cache with a line size of 16 bytes**, an update policy of write-deferred, write-allocate, and a replacement policy of least-recently-used (LRU).

Set	Tag	Valid?	Dirty?	Tag	Valid?	Dirty?
47	101	Y	N	111	N	N
46	010	Y	N	110	Y	Y
45	011	Y	Y	000	N	N
44	111	Y	Y	101	Y	N
43	010	Y	N	111	Y	N

For each of the following consecutive memory operations (in binary) for a 64KByte virtual memory space, indicate the **number of memory transfers** assuming that the bus width is 4 bytes, and show the **state of the “valid” and “dirty” bits** of the corresponding cache entry after the operation.

Show your working.

- (a) [2 marks] WRITE: 1010001011110100
- (b) [2 marks] READ: 1100001011100111
- (c) [2 marks] READ: 1100001011110000
- (d) [2 marks] WRITE: 0000001011111111
- (e) [2 marks] READ: 0100001011011010

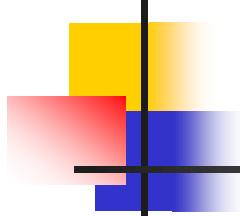
Solution

64KByte address space = 16 bit address
16KByte, 2-way set-associative 16 bytes

- 16 byte line/block size = **4** bits to address each byte
- 2-way associative means 32 bytes per set (2×16)
- 16KB cache @ 32 bytes per set = 512 sets (**9** bits)
($16\text{KB}/32\text{B} = 2^{14}/2^5 = 2^9$ sets = 512 sets)
- 9 bits to represent set number leaves **3** bits for tag
- low **4** bits = **offset** (to address each byte in a line/block)
- next **9** bits = **set number** (cache entry number)
- high **3** bits = **tag** (block number in memory)

3 bit 9 bit 4 bit
tag set num offset

(a) **WRITE** **101** **000101111** **0100**



Solution

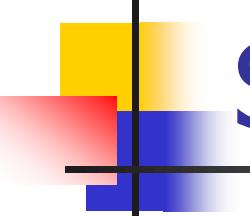
3 bits tag	9 bits set num	4 bits offset
---------------	-------------------	------------------

(a) **WRITE** **101** **000101111** **0100**

- Set number 101111_2 = set 47_{10} (decimal)
- Tag = 101 = Tag matched = Cache hit – that line/block now becomes dirty:

Set	Tag	Valid?	Dirty?	Tag	Valid?	Dirty?
47	101	Y	N	111	N	N
46	010	Y	N	110	Y	Y
45	011	Y	Y	000	N	N
44	111	Y	Y	101	Y	N
43	010	Y	N	111	Y	N

- Valid = Y, **Dirty = Y**
- No memory transfers because write-deferred (not write through)



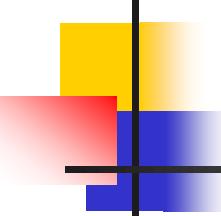
Solution

(b) READ: **110 000101110 0100**

- Set number 101110_2 = set 46_{10}
- Tag = 110 = Tag matched = Cache hit:

Set	Tag	Valid?	Dirty?	Tag	Valid?	Dirty?
47	101	Y	Y	111	N	N
46	010	Y	N	110	Y	Y
45	011	Y	Y	000	N	N
44	111	Y	Y	101	Y	N
43	010	Y	N	111	Y	N

- No memory transfers required – cache remains the same
- Valid = Y, Dirty = Y



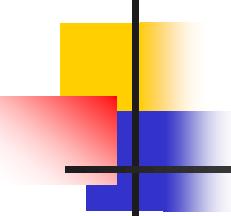
Solution

(c) READ **110 000101111 0000**

- Set number $101111_2 = \text{set } 47_{10}$
- Tag = 110 = No tag matched = Cache miss as no tag 110 is found on set 47 – so need to bring one line/block into the cache to replace an invalid (empty) entry (or a least recently used entry):

Set	Tag	Valid?	Dirty?	Set	Tag	Valid?	Dirty?
47	101	Y	Y	111	N	N	
46	010	Y	N	110	Y	Y	
45	011	Y	Y	000	N	N	
44	111	Y	Y	101	Y	N	
43	010	Y	N	111	Y	N	

- 4 memory transfers because must read in 16 bytes on bus width of 4
- **Valid = Y, Dirty = N**



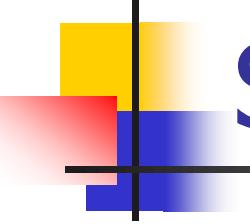
Solution

(d) WRITE 000 000**101111 1111**

- Set number $101111_2 = \text{set } 47_{10}$
- Tag = 000 = No tag matched = Cache miss as no tag 000 is found on set 47 – so need to bring one line/block into the cache to replace the least recently used entry (write-allocate):

Set	Tag	Valid?	Dirty?	Set	Tag	Valid?	Dirty?
47	101	Y	Y		110	Y	N
46	010	Y	N		110	Y	Y
45	011	Y	Y		000	N	N
44	111	Y	Y		101	Y	N
43	010	Y	N		111	Y	N

- 8 memory transfers: dirty & valid so write it out – 4 memory transfers and then bring in the new line/block – 4 memory transfers
- Valid = Y, Dirty = Y



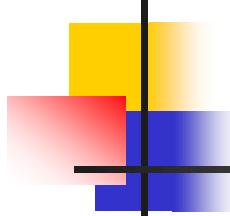
Solution

(e) READ **010 000101101 1010**

- Set number $101101_2 = \text{set } 45_{10}$
- Tag = 010 = No tag matched = Cache miss as no tag 010 is found on set 45 – so need to bring one line/block into the cache to replace an invalid (empty) entry or a least recently used entry:

Set	Tag	Valid?	Dirty?	Set	Tag	Valid?	Dirty?
47	000	Y	Y	110	Y	N	
46	010	Y	N	110	Y	Y	
45	011	Y	Y	000	N	N	
44	111	Y	Y	101	Y	N	
43	010	Y	N	111	Y	N	

- 4 memory transfers because must read in 16 bytes on bus width of 4
- **Valid = Y, Dirty = N**



Solution

- (a) Write set 47, tag 101 : 0 transfers
- (b) Read set 46, tag 110 : 0 transfers
- (c) Read set 47, tag 110 : 4 transfers
- (d) Write set 47, tag 000 : 8 transfers
- (e) Read set 45, tag 010 : 4 transfers

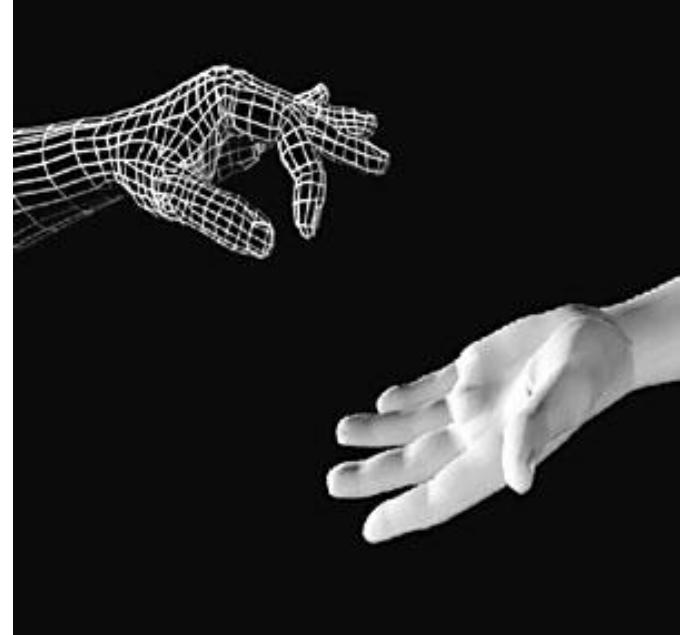
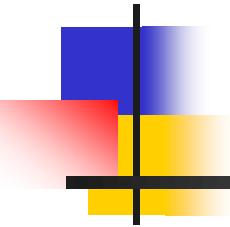
Set	Tag	Valid?	Dirty?	Set	Tag	Valid?	Dirty?
47	000	Y	Y	110	110	Y	N
46	010	Y	N	110	110	Y	Y
45	011	Y	Y	010	010	Y	N
44	111	Y	Y	101	101	Y	N
43	010	Y	N	111	111	Y	N

Total of 16 transfers

ENCE360

Operating

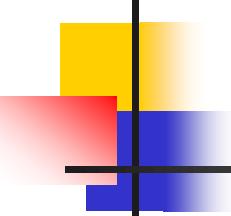
Systems



Memory management

Virtual memory

MOS (3rd ed.) Chapter 3

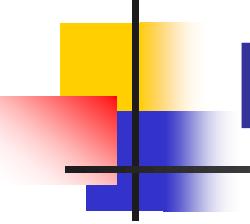


In the beginning...

- Single process
- Physical addressing
- Spatial separation between OS and user program
- Very little safeguards

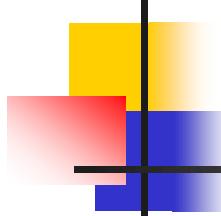
&0000	to	&3FFF	ROM
&4000	to	&57FF	Screen Memory
&5800	to	&5AFF	Screen Memory (Colour Data)
&5B00	to	&5BFF	Printer Buffer
&5C00	to	&5CBF	System Variables
&5CC0	to	&5CCA	Reserved
&5CCB	to	&FF57	Available Memory (between PROG and RAMTOP)
&FF58	to	&FFFF	Reserved

ZX Spectrum memory map



Problem 1: multiple programs

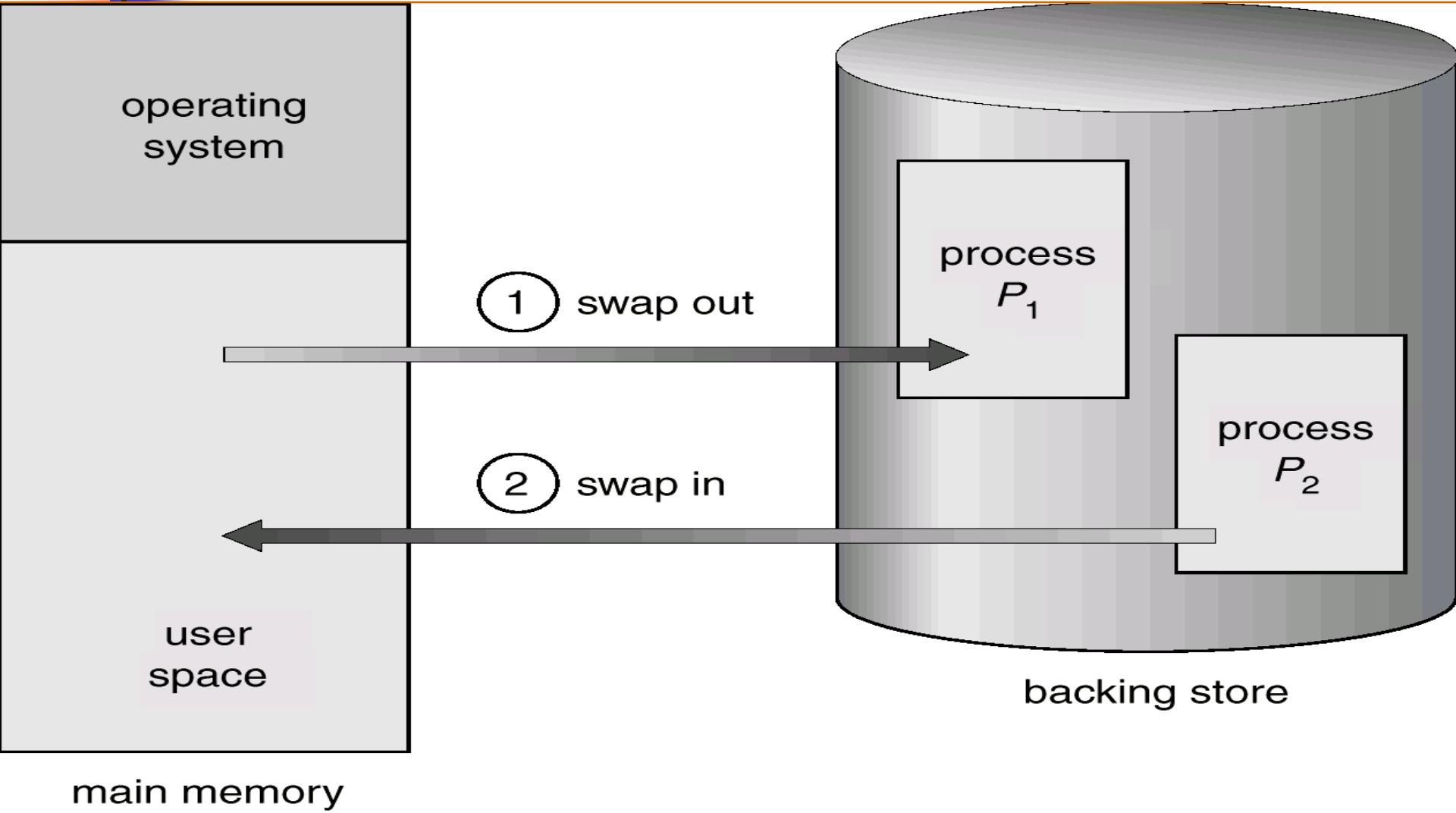
- More than one process running at a time
- Protection needed between address spaces
- Protection from operating system
- Address space small

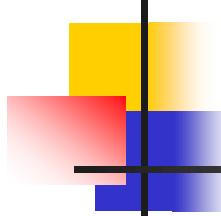


Solution 1: monoprogramming

- OS fixed into low addresses
- Programs compiled/loaded with *fixed* address space
 - Per program, or
 - Per user
- Poor solution:
 - Only certain combinations of programs can be run
 - External fragmentation (wasted memory)
 - Operating system *not protected*

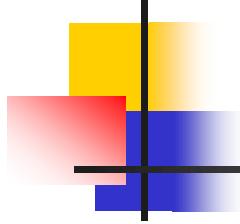
Solution 2: swapping





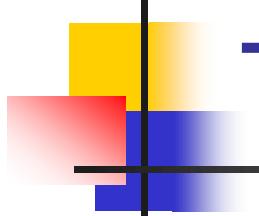
Solution 3: base-limit address

- *Virtual* address space of 0 to $P_{\text{limit}} - 1$
- Processor given P_{base} and P_{limit} when this process switched into context
- Hardware (CPU) converts:
 - Address $A_p = A_v + P_{\text{base}}$
 - Trap if $A_p \geq P_{\text{limit}}$



Base-limit problems

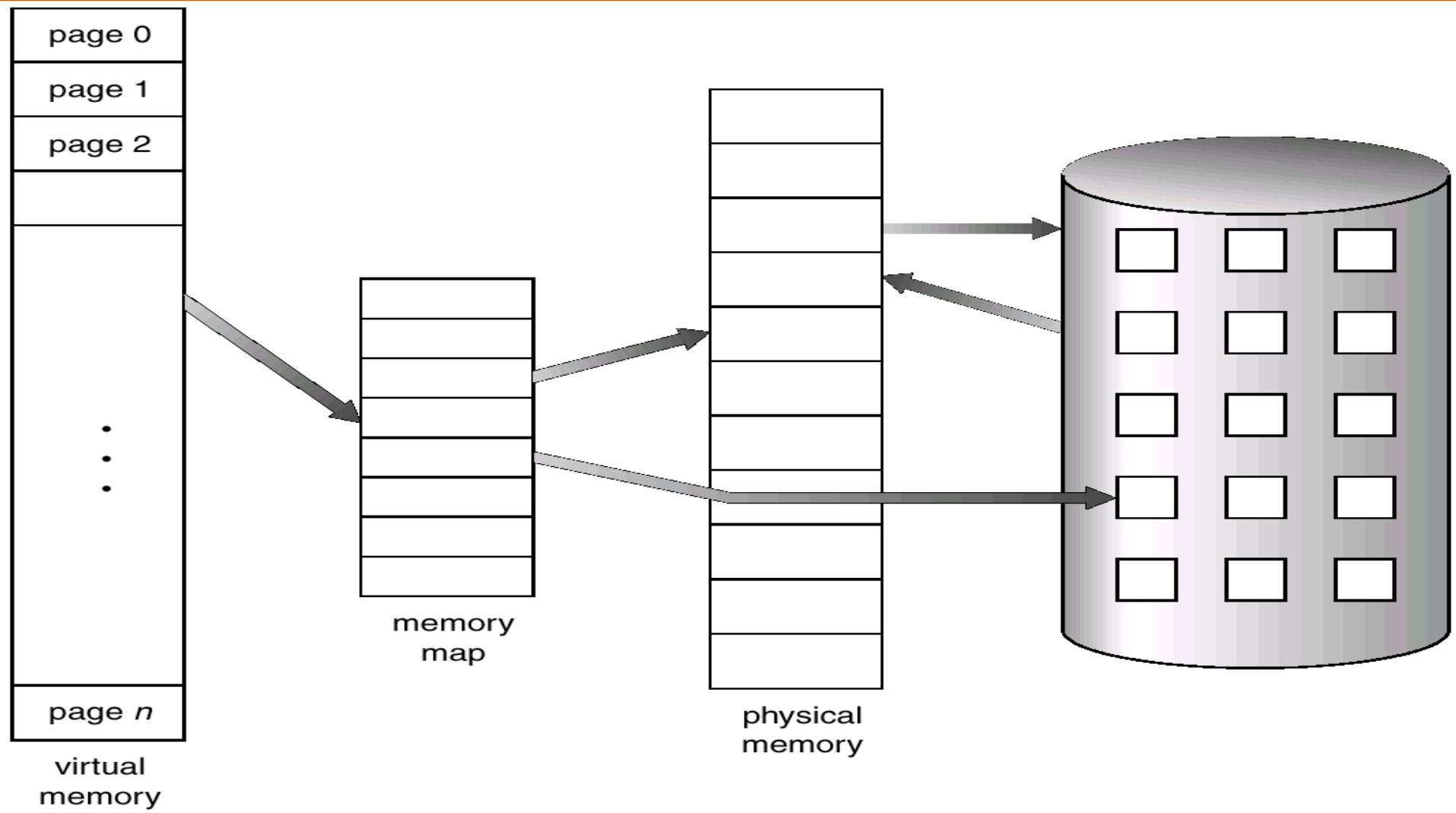
- Wasteful:
 - entire process loaded (vs locality)
 - External fragmentation
- P_{limit} dictates total address space size
 - Write small programs, OR
 - Manage memory yourself (write out intermediate results)

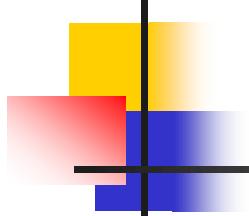


The “Manchester” solution

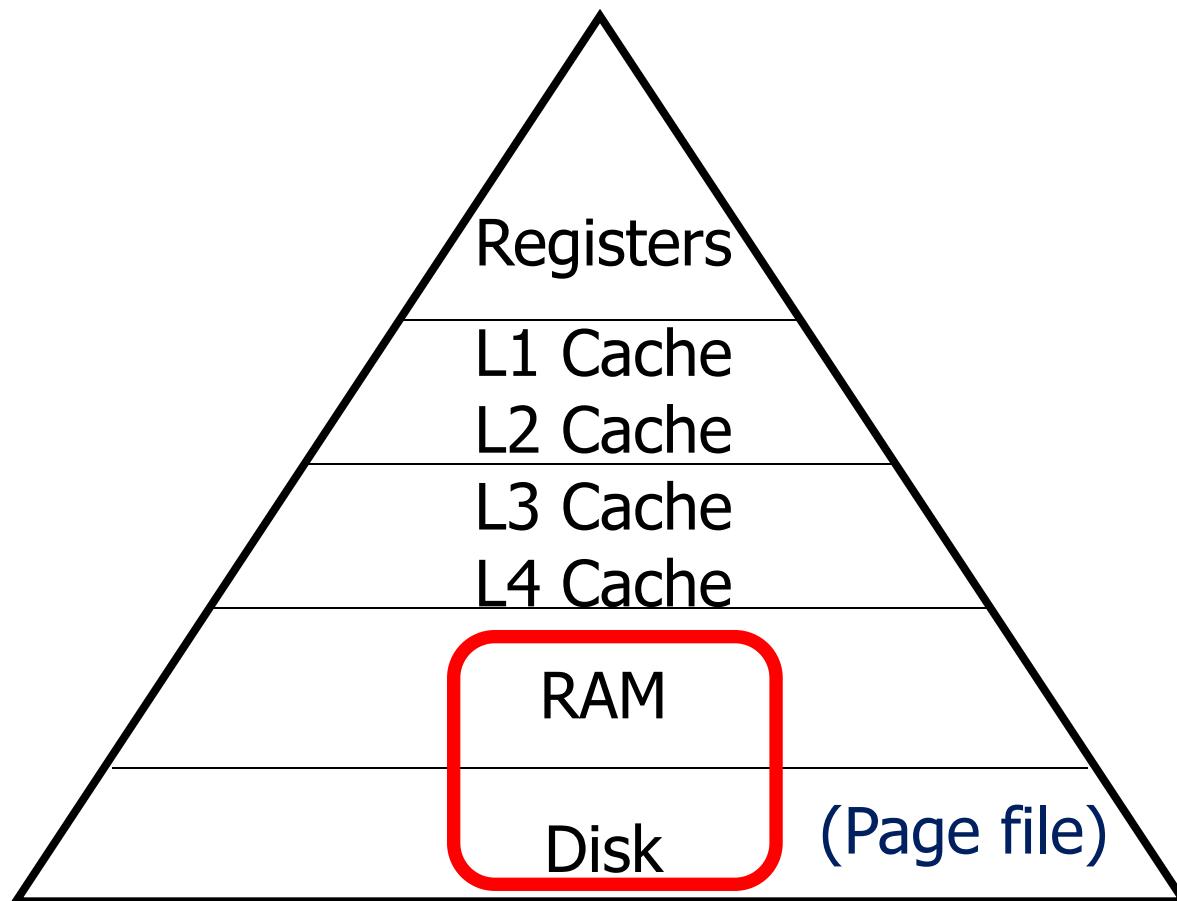
- Divorce virtual address from physical memory
- Allow arbitrary address contiguous space
 - (e.g. 32bits - 4GB)
 - (e.g. 64bits - 18 million Terabytes (18 Exabytes))
- Virtual address space *does not really exist anywhere*
 - Physical memory mapped to currently used portion of address space
 - The rest resides somewhere else (typically on disk)

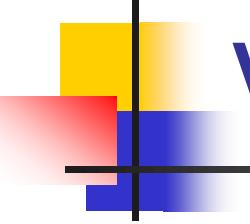
Virtual addressing





Memory hierarchy

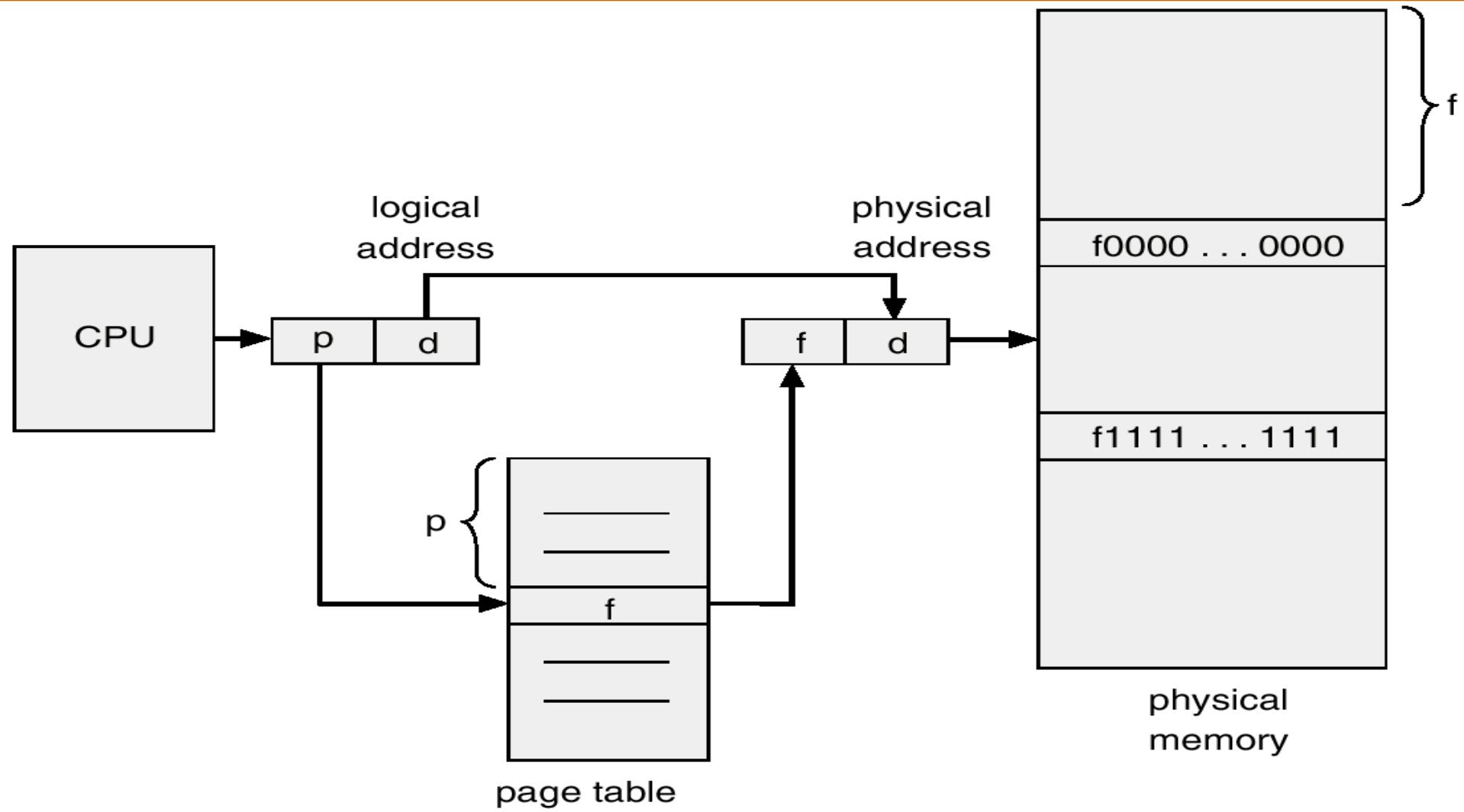


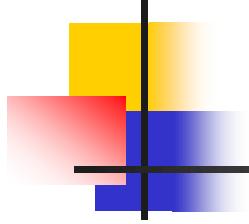


Virtual memory paging

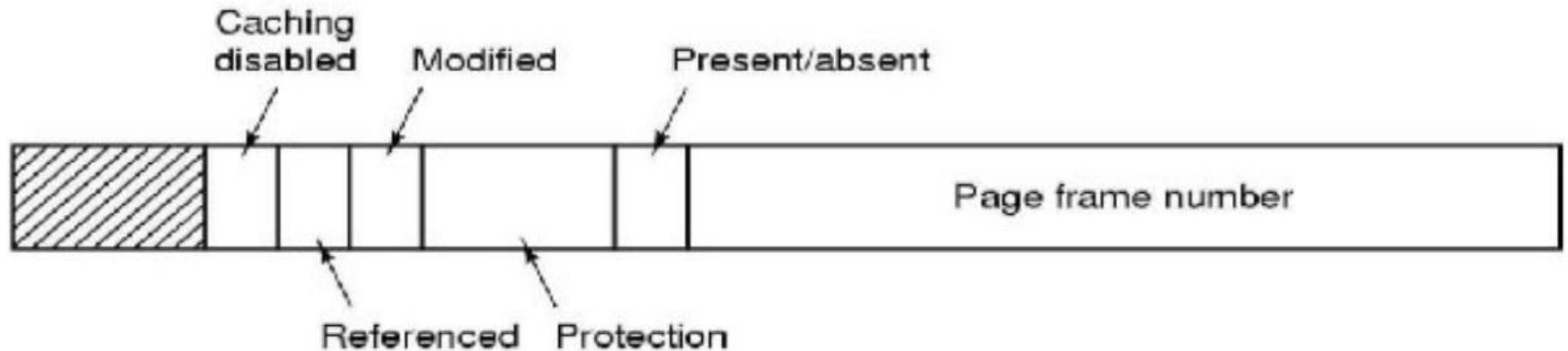
- Memory divided up into pages of 512 to 8192 bytes
- MMU: virtual page mapped to physical using *page table*:
 - Specialised direct-mapped cache (no tag)
 - Cache index is *virtual page number*
 - Data in cache is *physical page number*
 - Add page number to low bytes of virtual address (byte number) to get physical address
 - Treats physical memory as a fully-associative cache of the virtual address space (page table instead of tags)

Linear page table



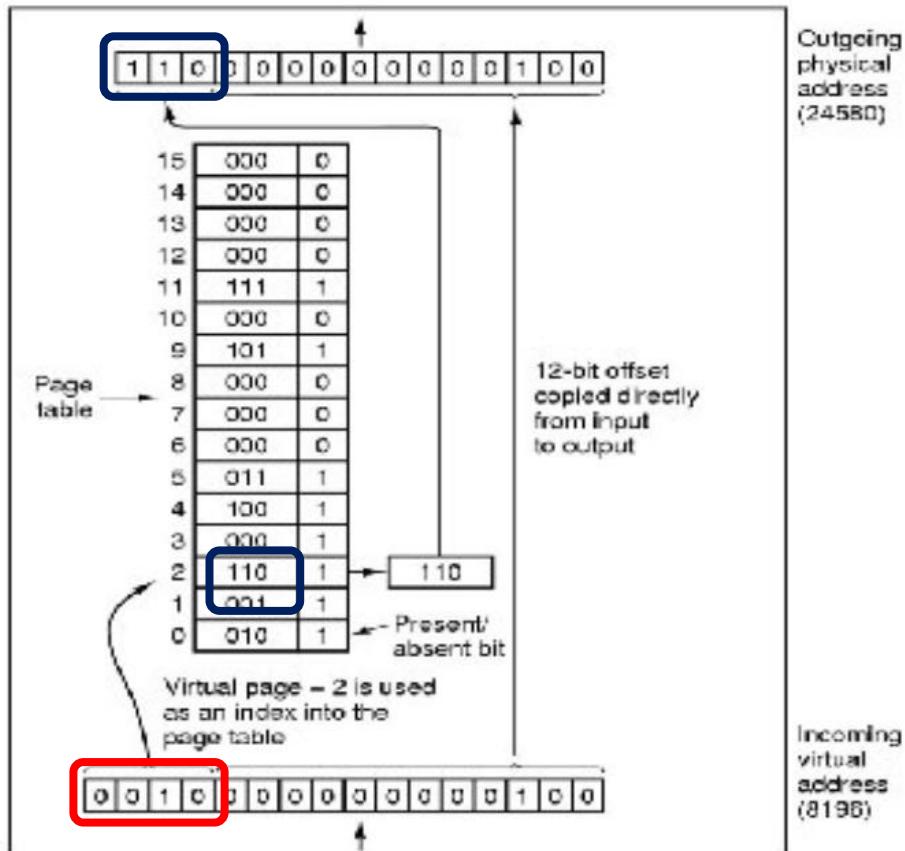


Page table entry (PTE) contents

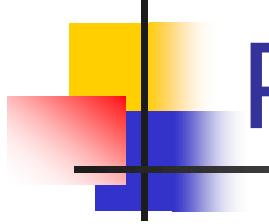


- Usually 32 bits
- Looked up in hardware
- OS records where other pages can be found

Address translation



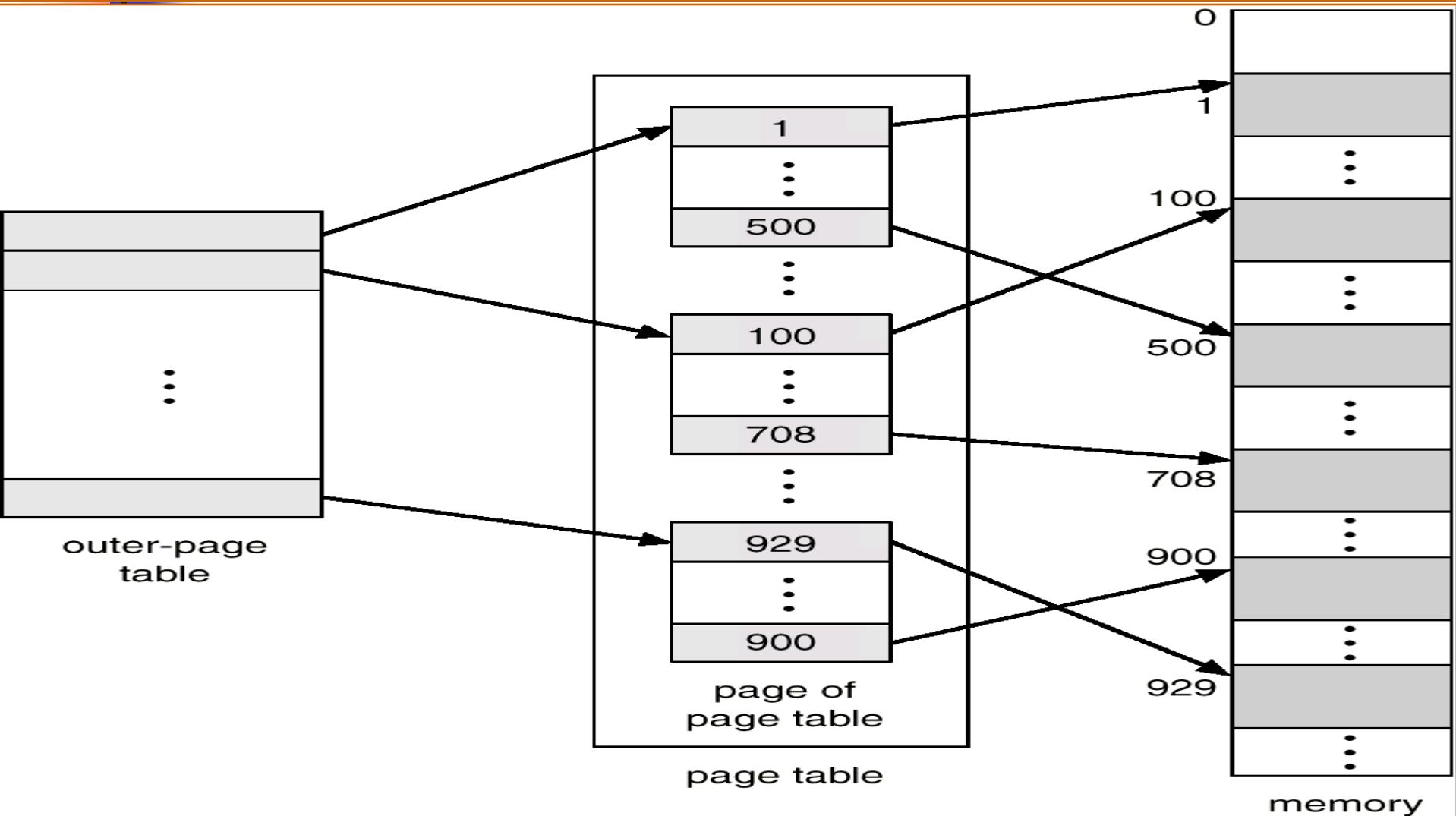
- High-order virtual address indexes page table (**virtual page number**)
- Page table entry contains physical page number
- Substituted to give physical address

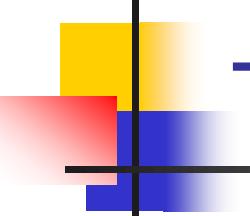


Problem: large address spaces

- Linear page table not feasible for big address spaces
- Use a *tree*: “hierarchical page table”
 - Most virtual pages are empty/not used
 - Divide virtual space into page ranges
 - Add “Page table descriptor” nodes
 - Include nodes for resident pages only
 - Locality of reference assumed

Hierarchical page table

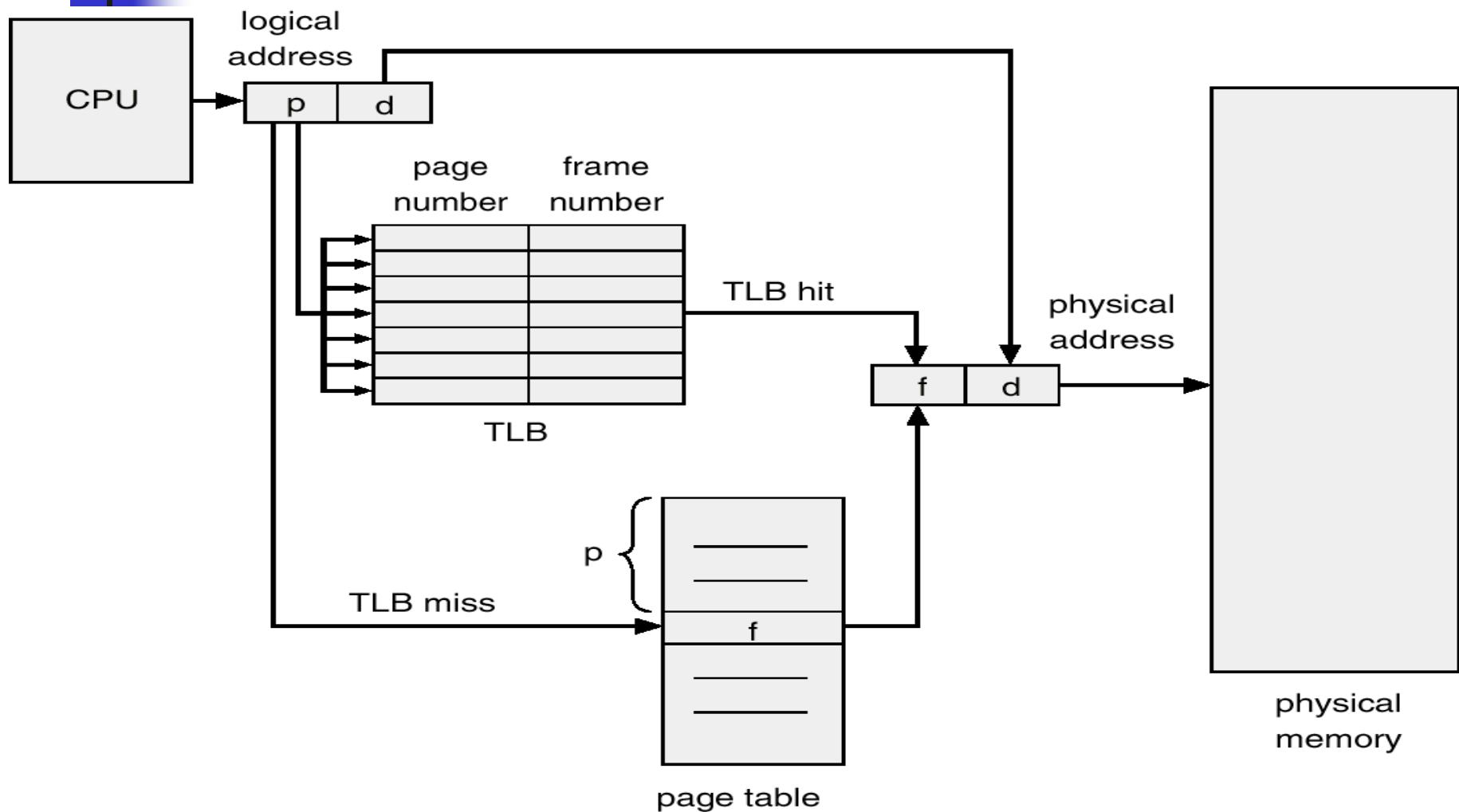


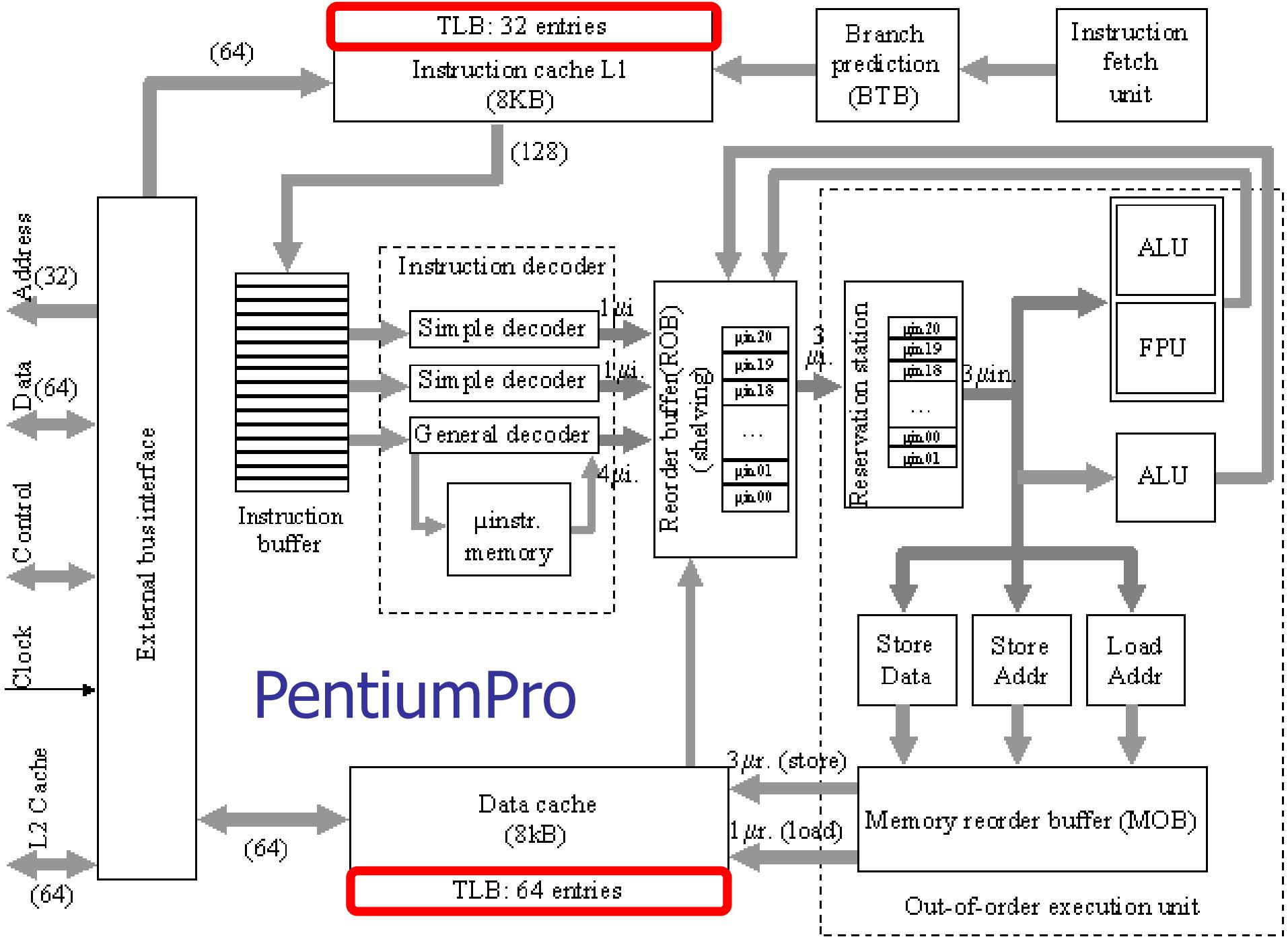


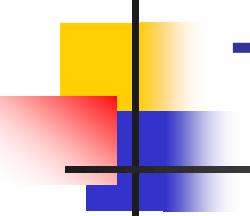
Translation Lookaside Buffer (TLB)

- Cache of translations, stored in CPU/MMU (often the same)
- Look in TLB first. If failed, go to page table
- Needs to know context:
 - Invalidate on context switch (common) OR
 - Store context too (e.g. process id, other)
 - Usually split instruction/data
- Small (8-64 entries), fully associative (parallel search), hard wired
 - Typically 99% hit rate

Paging with TLB

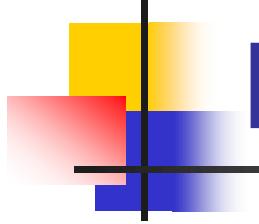






TLB extensions

- UltraSPARC: translation storage buffer (TSB)
 - If TLB (on CPU) miss, go to TSB (direct mapped)
 - TSB controlled by OS, helped by hardware
 - If TSB lookup fails, go to page table (no hardware help)
- OS has to help!
 - Manage TSB according to CPU requirements
 - Manage page table however it sees fit



Problem: huge address spaces

Virtual addressing means programs can get very large

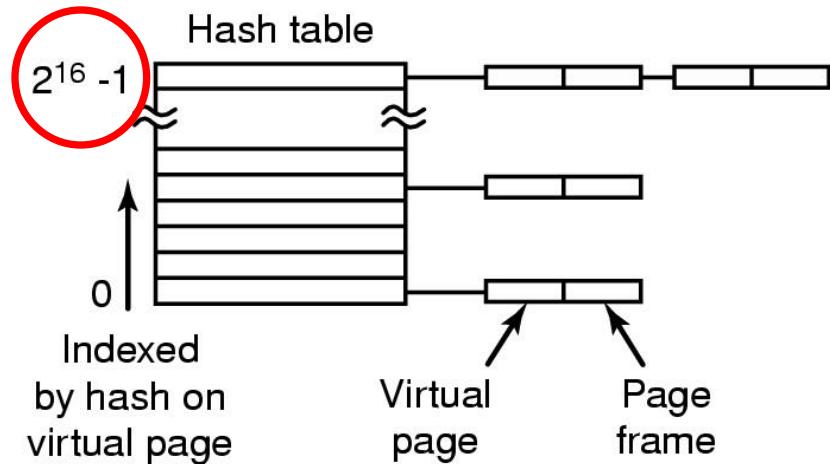
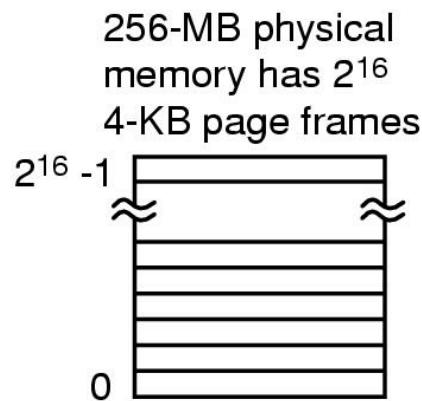
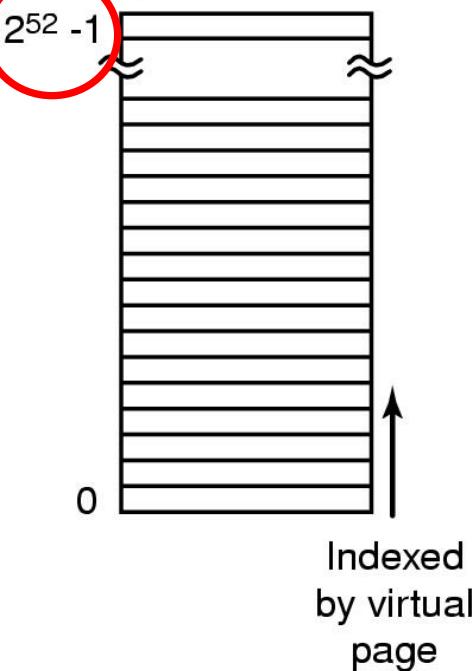
- Hierarchical page table becomes too large

Solution: Invert the page table

- One page table for all processes
 - One entry per *physical* page
- If TLB misses, scan inverted page table
- If not there (so not in memory) page fault occurs
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
 - Solution: use a hash table (by virtual page number)

Inverted page table

Traditional page table with an entry for each of the 2^{52} pages

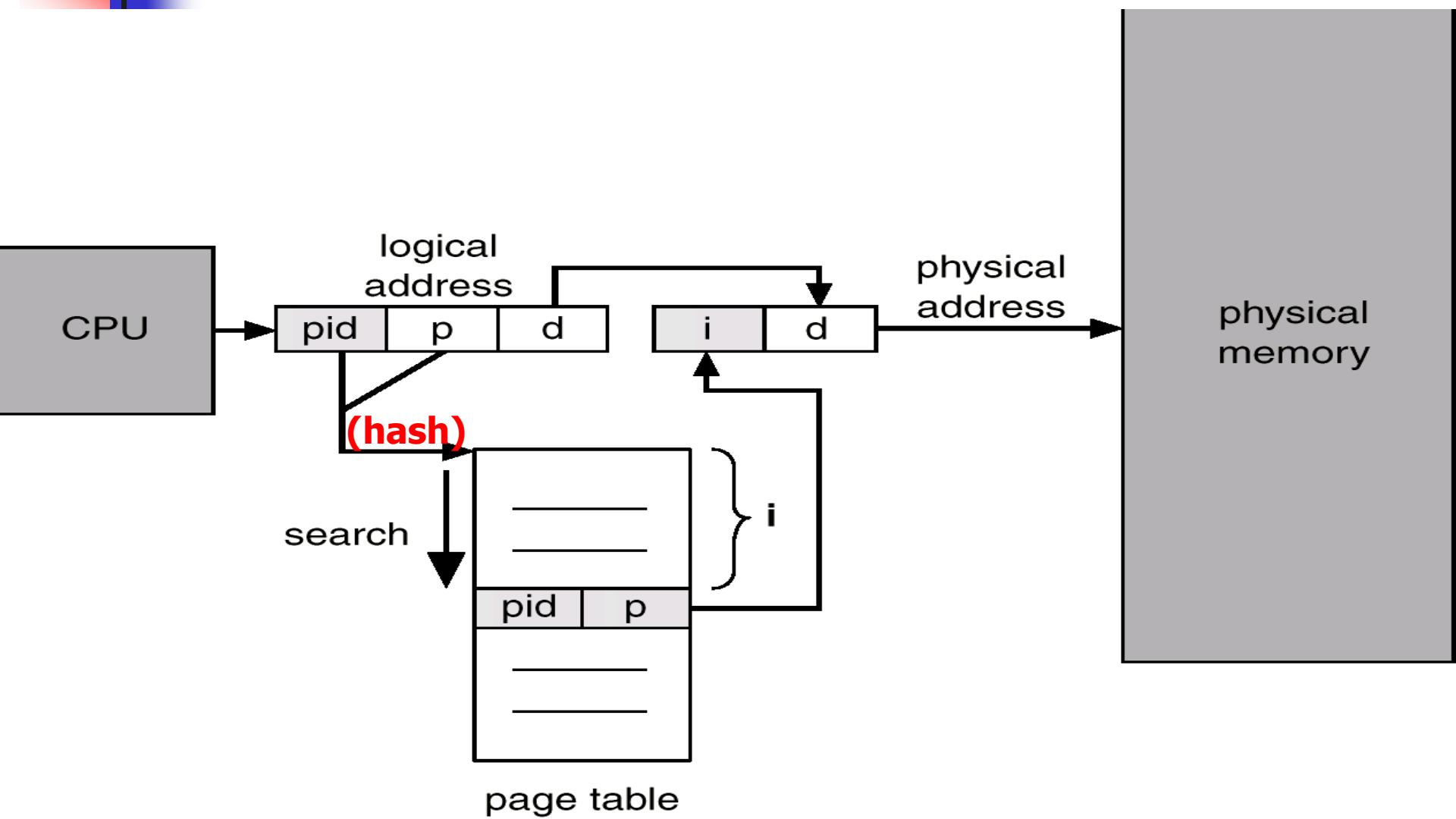


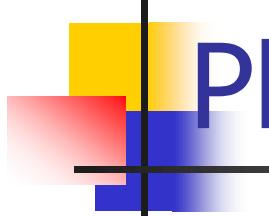
Traditional, inverted and inverted/hashed page tables

Inverted page table lookup

one single global inverted page table for all processes

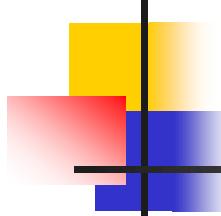
(pid=process ID, p=virtual page number, i=physical frame address, d=offset)





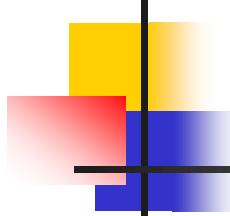
Physical address space division

- Some memory should not be cached (e.g. device registers)
- Some memory should not be mapped to virtual addresses (e.g. kernel)
- E.g. MIPS R2000
 - 2GB cached, mapped user segment
 - 512MB cached, unmapped kernel segment
 - 512MB uncached, unmapped kernel segment
 - 1GB cached, mapped kernel segment



Virtual address segmentation

- Virtual address space needs managing by OS
 - Protection of various memory regions
 - Size of different regions (dynamic)
- Solution: “segmentation”
 - Separate process virtual memory into independent logical units:
 - Program
 - Variables
 - Stack
 - Heap, etc.
 - Multiple virtual address spaces
 - Handled by the hardware (MMU)

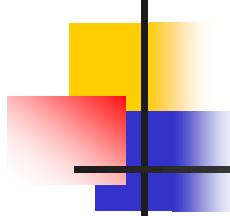


Page faults

- Only a subset of the virtual address space can fit in memory (working set)
- Access to non-resident address (page) signals a “page fault” exception
- Instruction “rolled back”
- Page fetched from disk, instruction executed again
 - Other work (processes) done during the wait

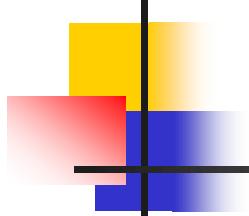
Page fault handling

1. Hardware traps to kernel (assembler routine)
2. General registers saved, OS called:
 1. Determine which virtual page needed
 2. Check validity of address, seeks free page frame
 - If selected frame is dirty, write it to disk
 3. Bring new page in from disk (from *pagefile*)
 - Page tables updated
 4. Faulting process is reset:
 1. Instruction backed up (unwound) to when it began
 2. Process scheduled
 5. Return to assembler
3. Registers restored
4. Program continues...



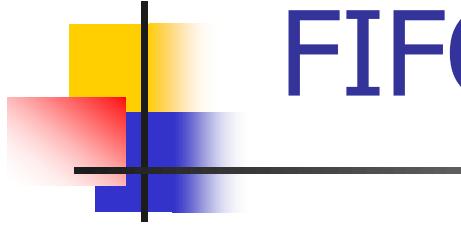
Page replacement algorithms (PRA)

- Page fault forces choice
 - Make room for incoming page: which page must be removed?
 - Modified pages must first be saved
 - Better not to choose an often used page
- Optimal strategy: replace page needed at the farthest point in future
 - Impossible!
 - Use fast approximation



Not Recently Used PRA

- Each page has Reference bit, Modified bit
 - bits are set when page is referenced, modified
 - Reference bit periodically reset to 0
- Pages are classified as:
 1. not referenced, not modified
 2. not referenced, modified
 3. referenced, not modified
 4. referenced, modified
- NRU removes page at random from lowest class set

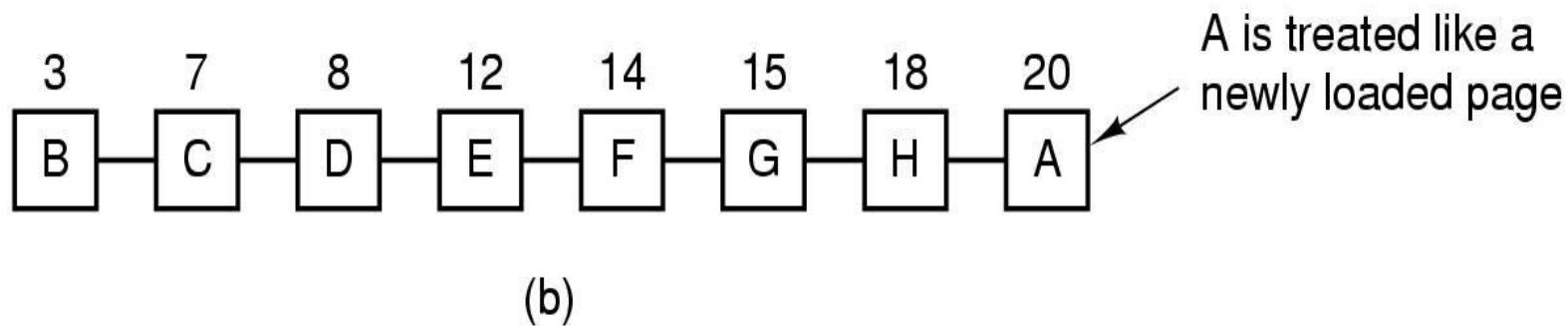
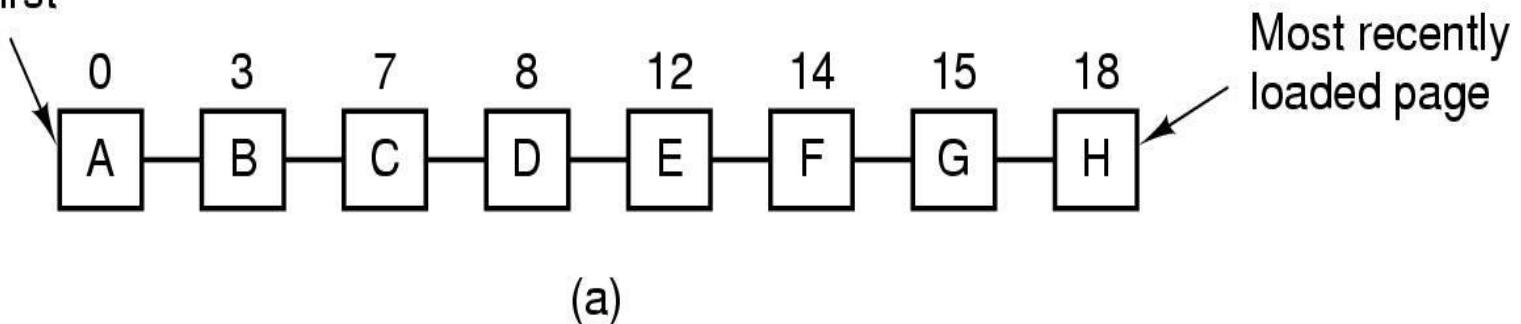


FIFO PRA

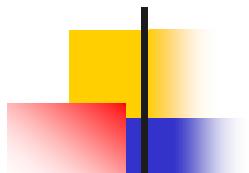
- Maintain a linked list of all pages
 - in order they came into memory
- Page at beginning of list replaced
- Disadvantage
 - page in memory the longest may be often used

“Second Chance” PRA

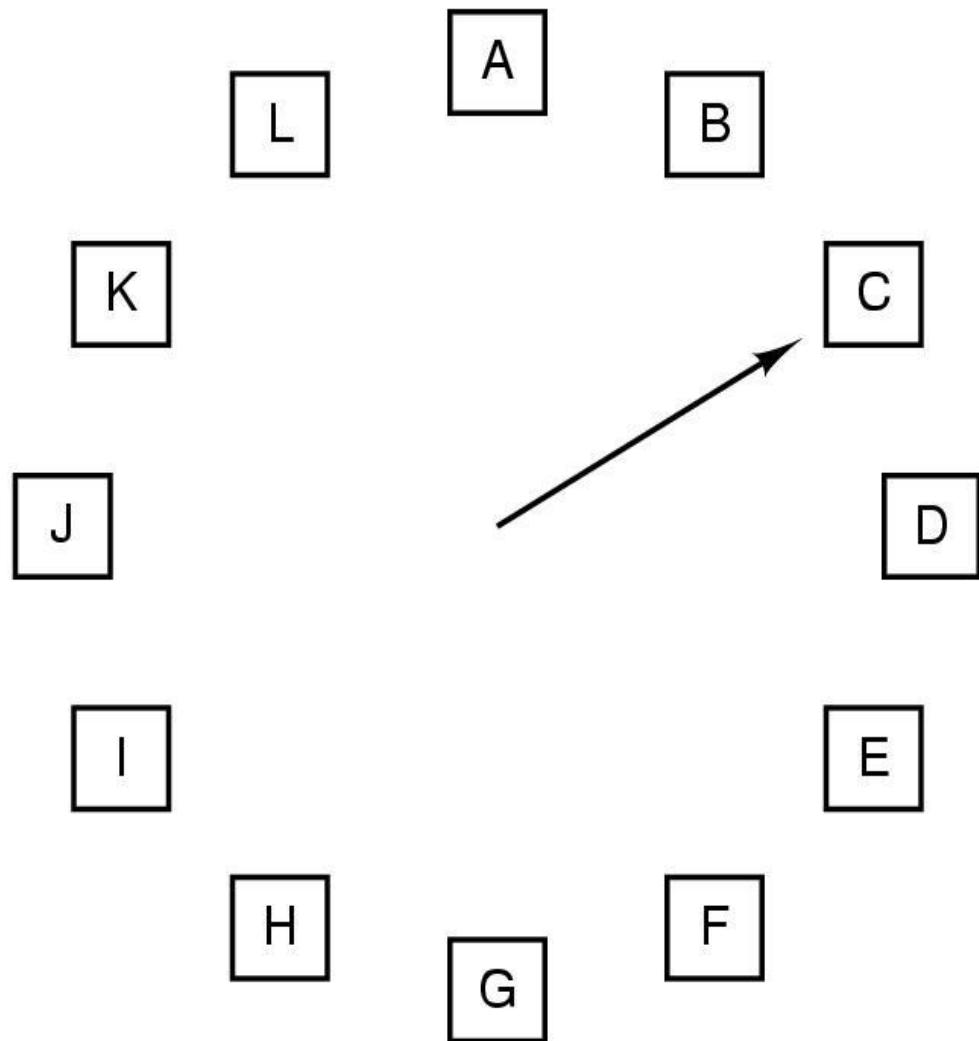
Page loaded first



- Combines NRU and FIFO
- Pages sorted in FIFO order
- If fault occurs at $t=20$ but A has *Referenced* bit set:
 - A sent to back of FIFO as though just loaded
 - Repeat on next entry until *Referenced*=0



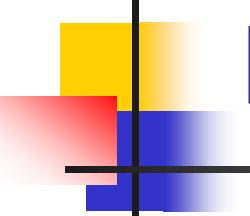
Clock PRA



When a page fault occurs,
the page the hand is
pointing to is inspected.
The action taken depends
on the R bit:

R = 0: Evict the page

R = 1: Clear R and advance hand



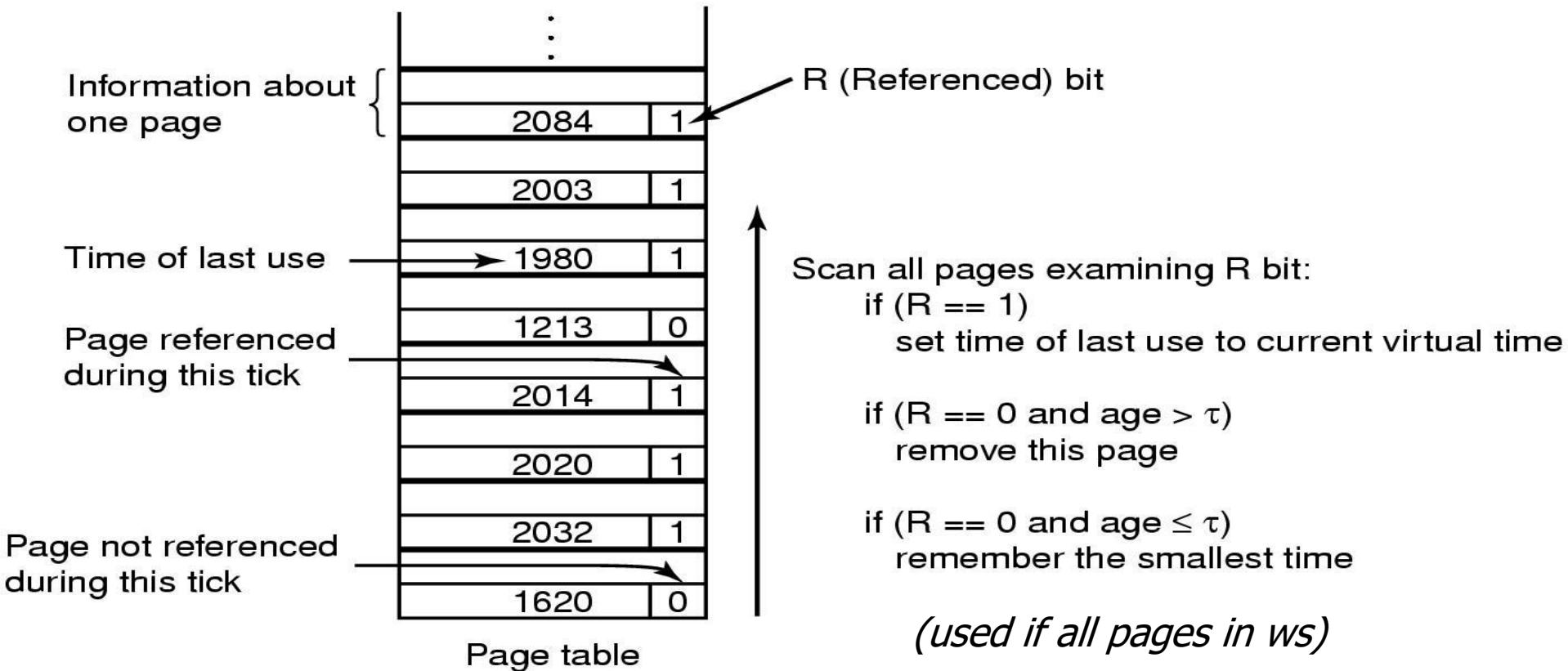
Least Recently Used (LRU) PRA

- Assume pages used recently will be used again soon
 - throw out page that has been unused for longest time
- Must keep a linked list of pages
 - most recently used at front, least at rear
 - update this list every memory reference !!
- Alternatively keep counter in each page table entry
 - choose page with lowest value counter
 - periodically zero the counter

Working Set PRA

2204

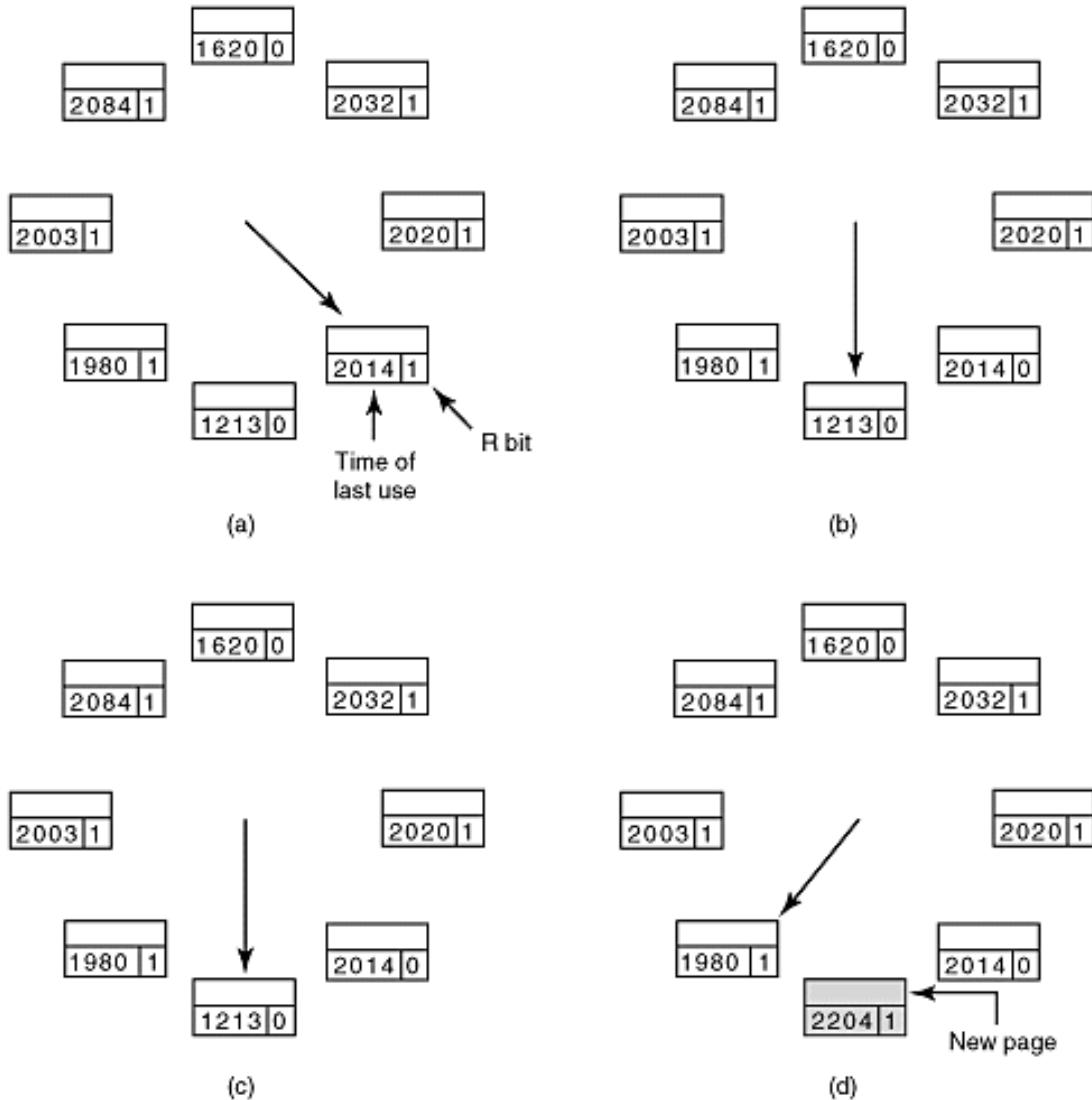
Current virtual time



- The working set is the set of pages used by the k most recent memory references (approximately)
 - Per *process*
 - Keeping WS in memory reduces thrashing (on switch)

WSClock PRA

2204 Current virtual time



a) Check current page entry for:

- a) Referenced = 0
- b) Age > t

b) Referenced = 1:

- a) Set Referenced=0
- b) advance hand

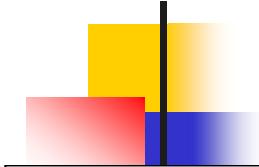
c) Check current page entry for:

- a) Referenced = 0
- b) Age > t

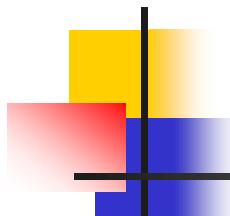
d) Match:

- a) Evict/replace page
- b) Set Referenced=1

Page replacement algorithms summary

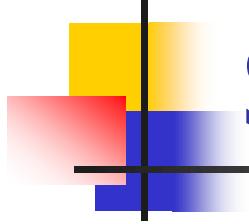


Algorithm	Comment
Optimal	Not implementable, but useful as a benchmark
NRU (Not Recently Used)	Very crude
FIFO (First-In, First-Out)	Might throw out important pages
Second chance	Big improvement over FIFO
Clock	Realistic
LRU (Least Recently Used)	Excellent, but difficult to implement exactly
NFU (Not Frequently Used)	Fairly crude approximation to LRU
Aging	Efficient algorithm that approximates LRU well
Working set	Somewhat expensive to implement
WSClock	Good efficient algorithm



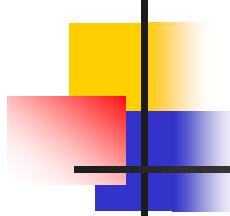
Cleaning policy

- Keep free space in page table for faster page loading:
 - Background process, paging “daemon”
 - Periodically frees page table entries
- When too few frames are free
 - selects pages to evict using a replacement algorithm
 - Marks as free but doesn't clear content (reserves)
 - Able to be restored up until actually evicted
- Can use same circular list (clock)
 - additional pointer (hand)



Swapping

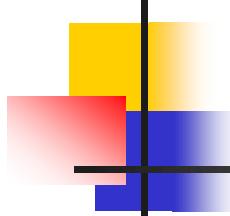
- Separate to paging
- Entire context (process) unloaded from memory, stored on disk
- Totally OS controlled
- Reduces page thrashing but *huge* performance hit for offending process



Page size

Small page size:

- Advantages
 - less internal fragmentation
 - better fit for various data structures, code sections
 - less unused program in memory
- Disadvantages
 - programs need many pages, larger page tables



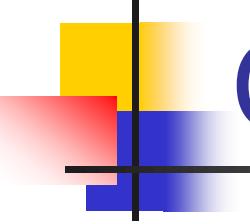
Page size

- Tradeoff between speed and efficiency:
 - Larger page size = more read per disk access
 - Smaller page size means less wastage
- Depends on *use*:
 - Scattered memory addresses: smaller page size is better
 - Contiguous memory addresses: larger page
 - Run simulations to optimise size

Operating system involvement

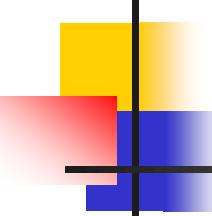
Four times when OS involved with paging:

1. Process creation
 - determine program size
 - create page table
2. Process execution
 - MMU reset for new process
 - TLB flushed
3. Page fault time
 - determine virtual address causing fault
 - swap target page out, needed page in
4. Process termination time
 - release page table, pages



OS designer issues

- Replacement strategies
 - Pages
 - TLB entries
- Memory division/allocation considerations
 - Sharing
 - Caching
 - Controlling thrashing
- Page/Swap file management

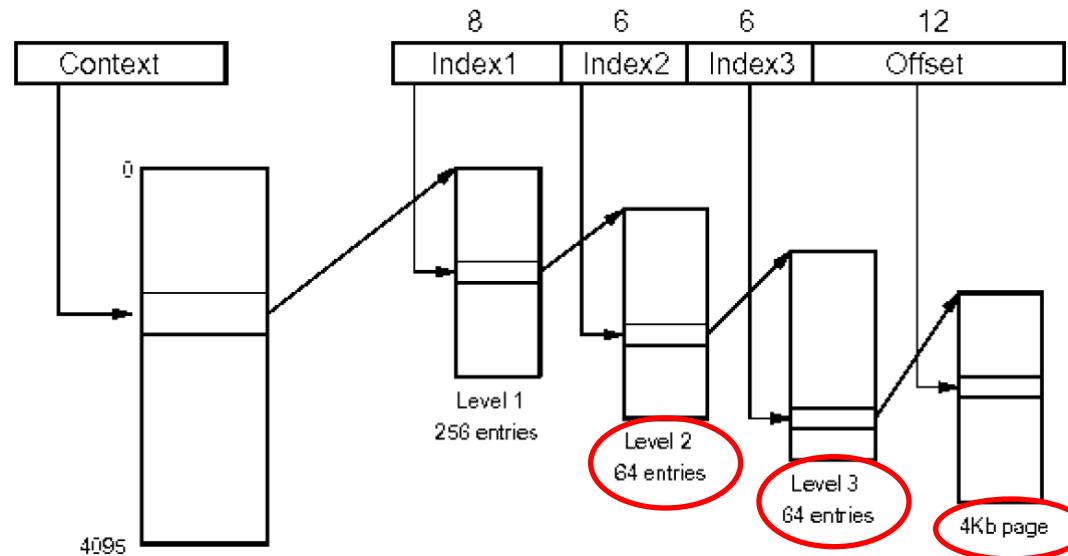


Summary

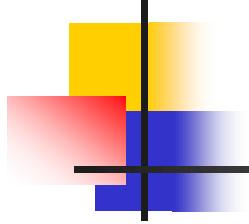
- Paging allows memory to do more:
 - Exceed physical memory size
 - Run multiple processes efficiently
- Complex set of caches etc to keep speed up
- Hardware and OS both heavily involved
- Like caching, OS can be optimised to minimise page faulting

Example exam question

The **32-bit** SPARC architecture uses a **3-level page table** as illustrated below.



Calculate **how many bytes are required for the page table** of a process on a SPARC system with a **5MByte** text segment, a **20MByte** data segment, and a **16MByte** stack. Assume that the text segment starts at 0x0, that the data segment follows the text segment, and that the stack grows down from 0xFFFFFFFF. **Level 1 nodes are 1024 bytes** each, **level 2 and 3 nodes are 256 bytes** each. *Show your working.*



Solution (1)

Advantages:

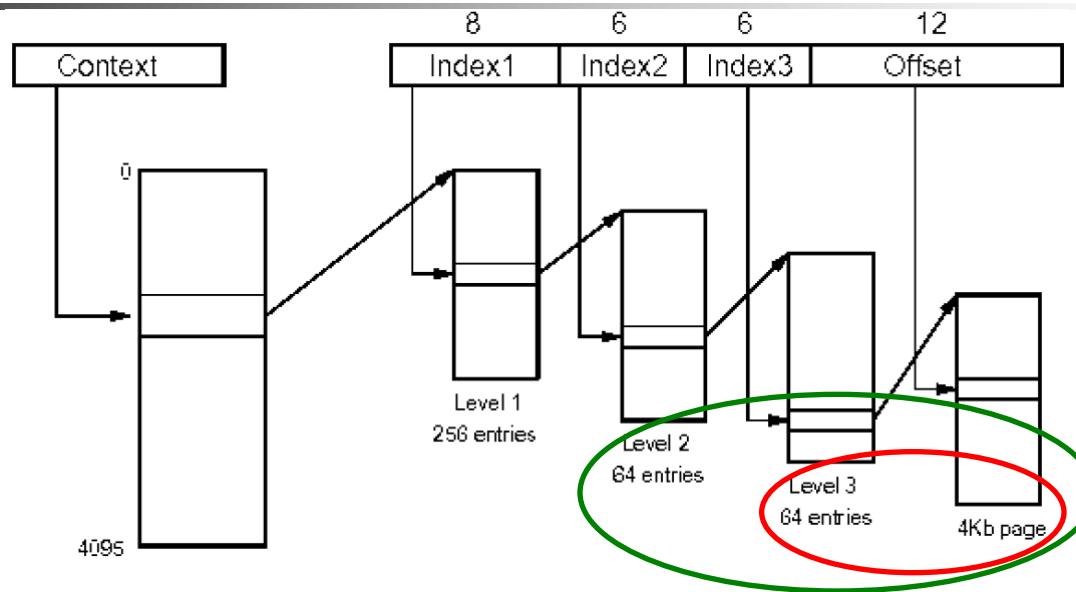
- Takes up less space than a 2 or 1 level page table
- Not all of the address space needs to be in memory at once, only the pages that are required.

Disadvantages:

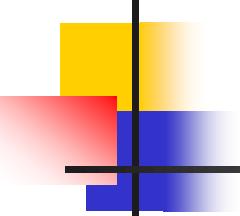
- Slower to look up than a 2 or 1 page table.
- Can become large and complex. (A better option may be to invert the page table so there is one entry for each physical frame.)

Solution (2)

How much memory can a L1, L2, L3 page/node address?



- A level 3 page addresses $64 \times 4\text{KB}$ pages
= **256KB** (in the red ellipse above)
- A level 2 page addresses $64 \times 64 \times 4\text{KB}$ pages
= **16MB** (in the green ellipse)
- A level 1 page addresses all 32bit virtual memory (**4GB**)



Solution (3)

How many pages/nodes are needed?

Level 1 pages

Need only *one* level 1 page/node

Level 2 pages

For **5MB text + 20MB data** (from 0x0 to 25MB)

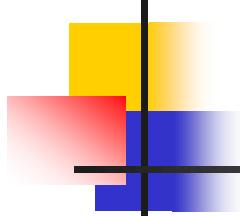
Need $25\text{MB}/16\text{MB}$ (L2 page) rounded up

= two level 2 pages (which can access **16MB** each)

For **16MB stack**

Need only one level 2 (**16MB**) page

Need total of *three* level 2 pages/nodes



Solution (4)

How many pages/nodes are needed?

Level 3 pages

For 5MB text + 20MB data

Need 25MB/**256KB**(L3 page)

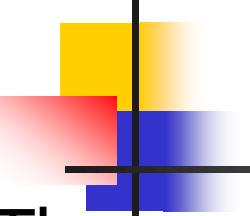
= 100 level 3 pages (which can access **256KB** each)

For 16MB stack

Need 16MB/**256KB**(L3 page)

= 64 level 3 pages

Need total of 164 level 3 pages/nodes



Solution (5)

What is the total memory used?

The question stated that “Level 1 nodes are **1024** bytes each, level 2 and 3 nodes are **256** bytes each”.

Page table size (total memory used):

1 L1 node + 3 L2 nodes + 164 L3 nodes

$$1 \times \mathbf{1024} + 3 \times \mathbf{256} + 164 \times \mathbf{256} \text{ bytes}$$

$$= 1024 + 768 + 41984 \text{ bytes}$$

$$= \mathbf{43,776 \text{ bytes}}$$

ENCE360

Operating Systems



Distributed systems:

Message passing (MPI)

Distributed CPUs: message passing

```
#define N 100                                     /* number of slots in the buffer */

void producer(void)
{
    int item;
    message m;

    while (TRUE) {
        item = produce_item();
        receive(consumer, &m);
        build_message(&m, item);
        send(consumer, &m);
    }
}

void consumer(void)
{
    int item, i;
    message m;

    for (i = 0; i < N; i++) send(producer, &m); /* send N empties */
    while (TRUE) {
        receive(producer, &m);
        item = extract_item(&m);
        send(producer, &m);
        consume_item(item);
    }
}
```

Receive blocks until a message is received

N empty messages act as a buffer

/* message buffer */

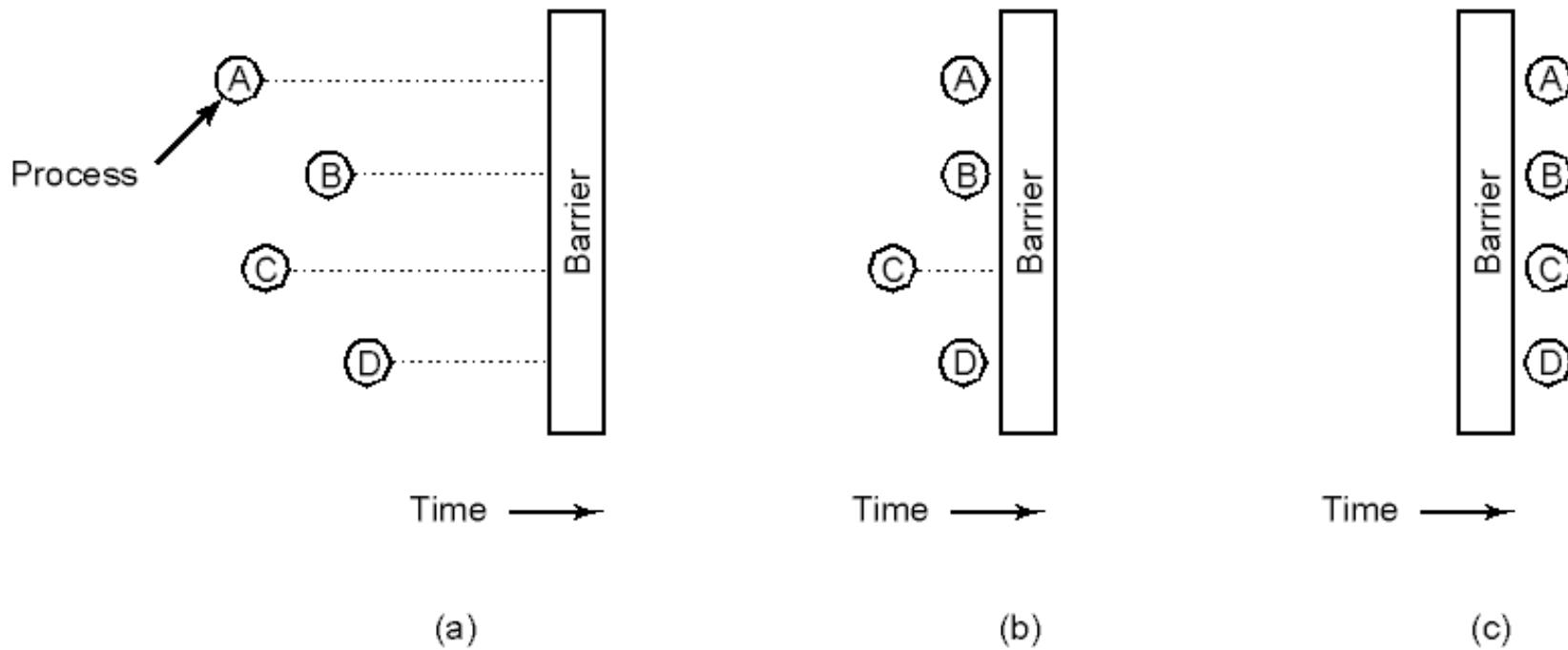
/* generate something to put in buffer */

/* wait for an empty to arrive */

/* construct a message to send */

/* send item to consumer */

Multi-process synchronisation: barriers



■ Synchronises distributed iterative algorithms

- Blocks progress until all processes have completed
- Example: large matrix operations (e.g. image analysis)

Message Passing Interface (MPI)

- MPI is a:
 - language-independent communications protocol used to program distributed computers.
 - message-passing application programmer interface, together with protocol and semantic specifications for how its features must behave in any implementation.
- High performance, scalable, portable.
- Point-to-point and collective communication

Message Passing Interface (MPI)

- API: specific set of routines directly callable from C, C++, Fortran (and any language able to interface with such libraries, including C#, Java or Python)
- Advantages:
 - portability - implemented for almost every distributed memory architecture, similar across languages
 - Speed
 - ▶ Optimised for the hardware on which it runs
 - ▶ Can be distributed over multiple network nodes
 - Complementary to threads (can use both)
 - ▶ E.g. cluster of multi-core nodes

Message Passing Interface (MPI)

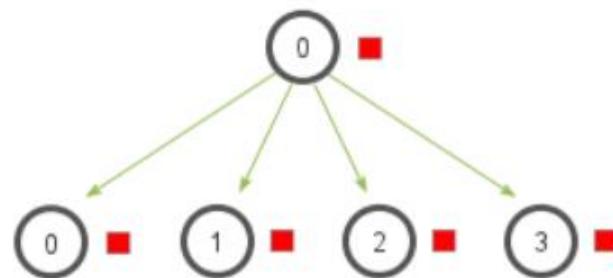
- Invocation (run 5 copies of hello_world):
 - `mpirun -n 5 -hostfile remote_hosts hello_world`
- Code structure: similar to process forking:
 - Same code run for all processes
 - Check the “rank” to decide what to do (0=root)
- MPI function parameters: if the data type is a standard one (int, char, double, etc), you can use predefined MPI datatypes (`MPI_INT`, `MPI_CHAR`, `MPI_DOUBLE`, etc)

Message Passing Interface (MPI)

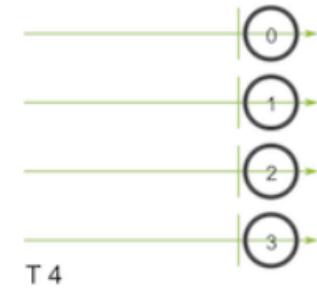
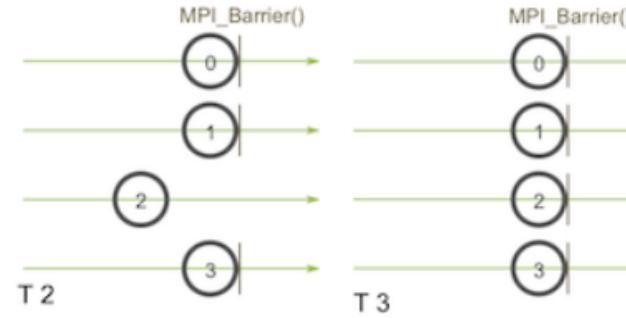
■ Main functions:

- Point to point: Send and Recv
- One to many: Bcast and Barrier (synchronise)

MPI_Bcast

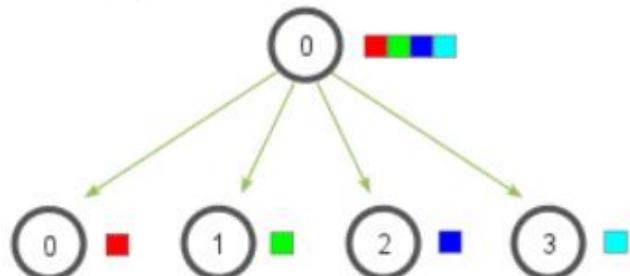


MPI_Barrier()

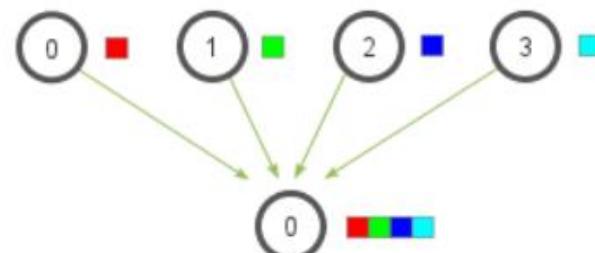


- Distribute and collate: Scatter and Gather

MPI_Scatter



MPI_Gather



```
int main(int argc, char *argv[])
{
    char idstr[32];
    char buff[BUFSIZE];
    int numprocs;
    int myid;
    int i;
    MPI_Status stat;
    /* MPI programs start with MPI_Init; all 'N' processes exist thereafter */
    MPI_Init(&argc, &argv);
    /* find out how big the SPMD world is */
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    /* and this processes' rank is */
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);

    /* At this point, all programs are running equivalently, the rank
       distinguishes the roles of the programs in the SPMD model, with
       rank 0 often used specially... */
    if (myid == 0)
    {
        printf("%d: We have %d processors\n", myid, numprocs);
        for (i=1; i<numprocs; i++)
        {
            sprintf(buff, "Hello %d! ", i);
            MPI_Send(buff, BUFSIZE, MPI_CHAR, i, TAG, MPI_COMM_WORLD);
        }
        for (i=1; i<numprocs; i++)
        {
            MPI_Recv(buff, BUFSIZE, MPI_CHAR, i, TAG, MPI_COMM_WORLD, &stat);
            printf("%d: %s\n", myid, buff);
        }
    }
    else
    {
        /* receive from rank 0: */
        MPI_Recv(buff, BUFSIZE, MPI_CHAR, 0, TAG, MPI_COMM_WORLD, &stat);
        sprintf(idstr, "Processor %d ", myid);
        strncat(buff, idstr, BUFSIZE-1);
        strncat(buff, "reporting for duty", BUFSIZE-1);
        /* send to rank 0: */
        MPI_Send(buff, BUFSIZE, MPI_CHAR, 0, TAG, MPI_COMM_WORLD);
    }

    /* MPI programs end with MPI_Finalize; this is a weak synchronization point */
    MPI_Finalize();
    return 0;
}
```

Send/Recv example: hello_world

Scatter and Gather

```
MPI_Scatter(  
    void* send_data,  
    int send_count,  
    MPI_Datatype send_datatype,  
    void* recv_data,  
    int recv_count,  
    MPI_Datatype recv_datatype,  
    int root,  
    MPI_Comm communicator)
```

```
MPI_Gather(  
    void* send_data,  
    int send_count,  
    MPI_Datatype send_datatype,  
    void* recv_data,  
    int recv_count,  
    MPI_Datatype recv_datatype,  
    int root,  
    MPI_Comm communicator)
```

■ Scatter:

- If id=root: divides up send_data and sends a portion to each process
- All processes (including root): receive data in recv_data for processing

■ Gather:

- All processes (including root): send send_data to the root process
- Root process: receive all messages in recv_data

```
if (world_rank == 0) {
    rand_nums = create_rand_nums(elements_per_proc * world_size);
}

// Create a buffer that will hold a subset of the random numbers
float *sub_rand_nums = malloc(sizeof(float) * elements_per_proc);

// Scatter the random numbers to all processes
MPI_Scatter(rand_nums, elements_per_proc, MPI_FLOAT, sub_rand_nums,
            elements_per_proc, MPI_FLOAT, 0, MPI_COMM_WORLD);

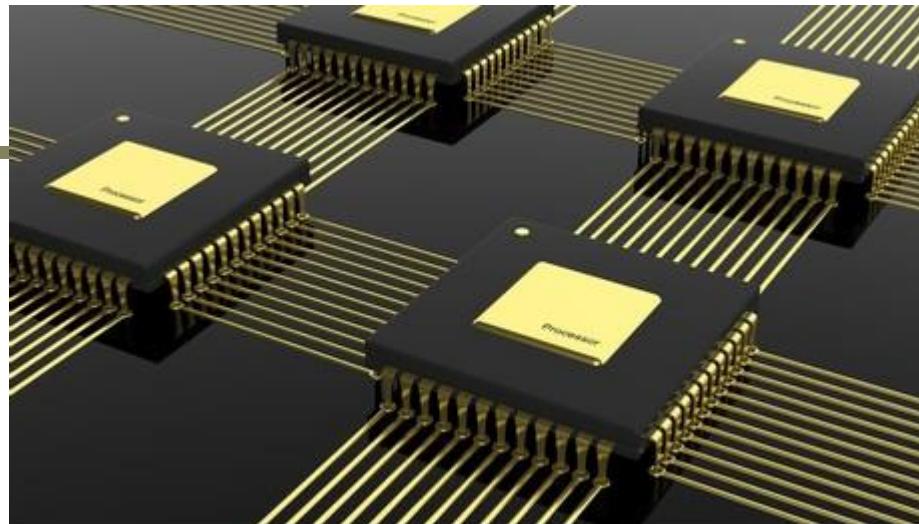
// Compute the average of your subset
float sub_avg = compute_avg(sub_rand_nums, elements_per_proc);
// Gather all partial averages down to the root process
float *sub_avgs = NULL;
if (world_rank == 0) {
    sub_avgs = malloc(sizeof(float) * world_size);
}
MPI_Gather(&sub_avg, 1, MPI_FLOAT, sub_avgs, 1, MPI_FLOAT, 0,
           MPI_COMM_WORLD);

// Compute the total average of all numbers.
if (world_rank == 0) {
    float avg = compute_avg(sub_avgs, world_size);
}
```

Scatter/gather example: average

ENCE360

Operating Systems



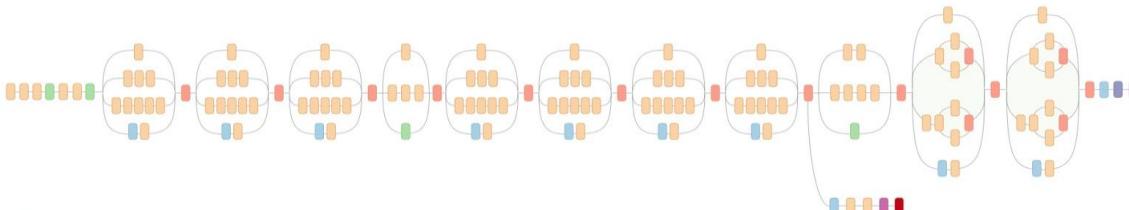
shutterstock.com • 251423881

Multiprocessor systems

MOS (3rd ed.) CH. 8

Why multiprocessing?

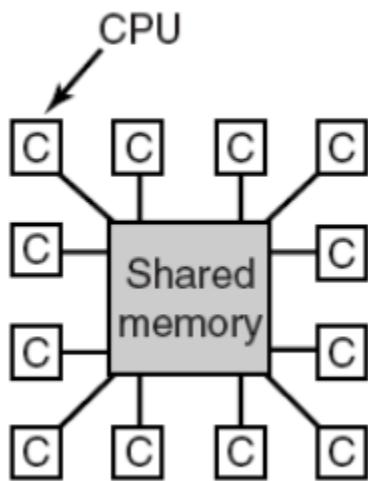
- More speed!
 - 10GHz clock: can travel 2cm between cycles
 - 100GHz clock: 2mm
 - 1000GHz clock: 0.2mm!
 - Solution: more CPUs
- Rapid growth in high-CPU, parallelisable algorithms (e.g. deep learning)
- Internet turns the world's PCs into a giant cluster (virtual supercomputer)
- Sharing of resources (I/O devices, files)



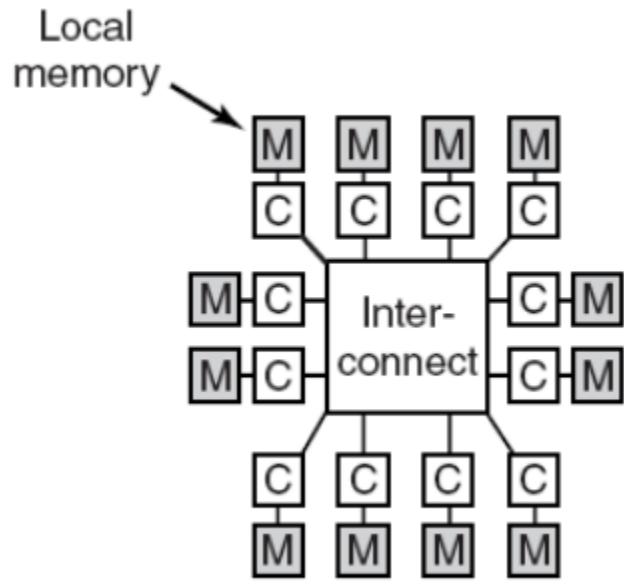
Legend:
Orange: Convolution
Blue: AvgPool
Green: MaxPool
Red: Concat
Purple: Dropout
Pink: Fully connected
Yellow: Softmax



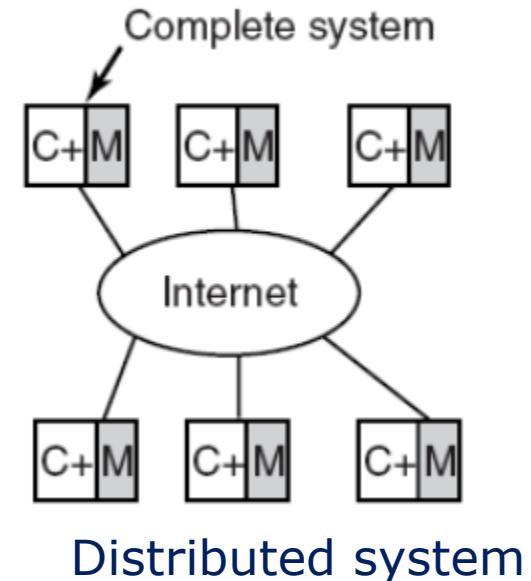
Types of multiprocessors



Multiprocessor



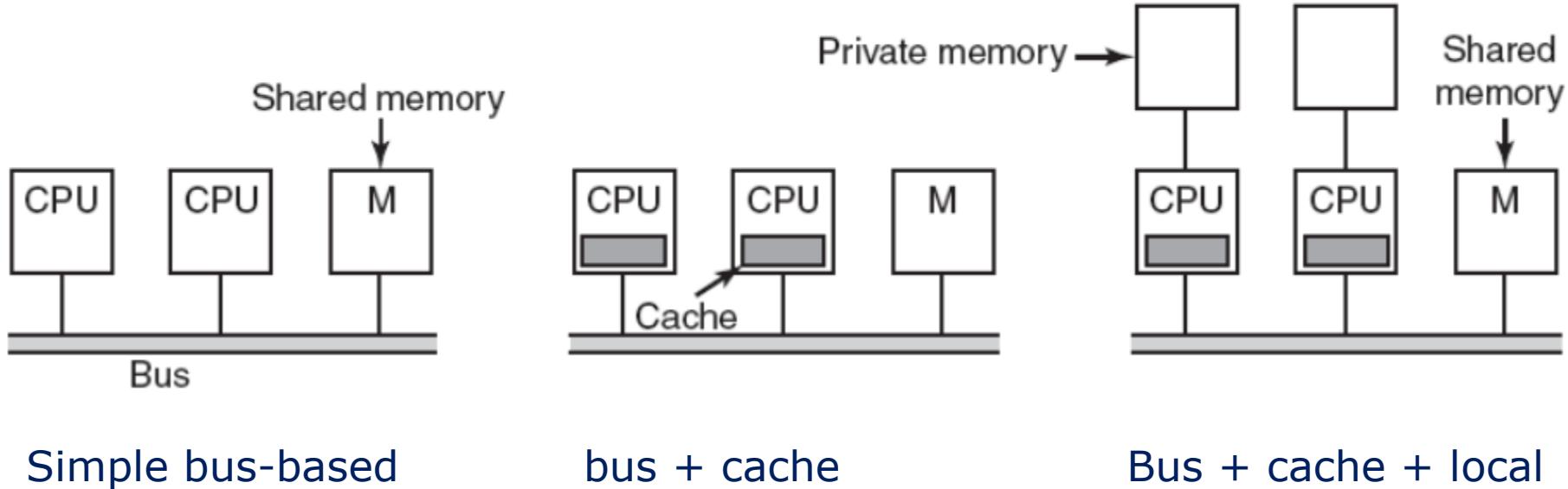
Multicomputer (cluster)



Distributed system

- Multiprocessor: multiple CPUs (or cores) in a single machine
 - Memory shared between cores
- Multicomputer: multiple nodes connected by interconnect hardware
 - Nodes can have > 1 CPU
 - Each CPU has its own memory
 - Multi-word messages transmitted over fast interconnect
- Distributed system: independent computers communicate via internet

Shared memory multiprocessors



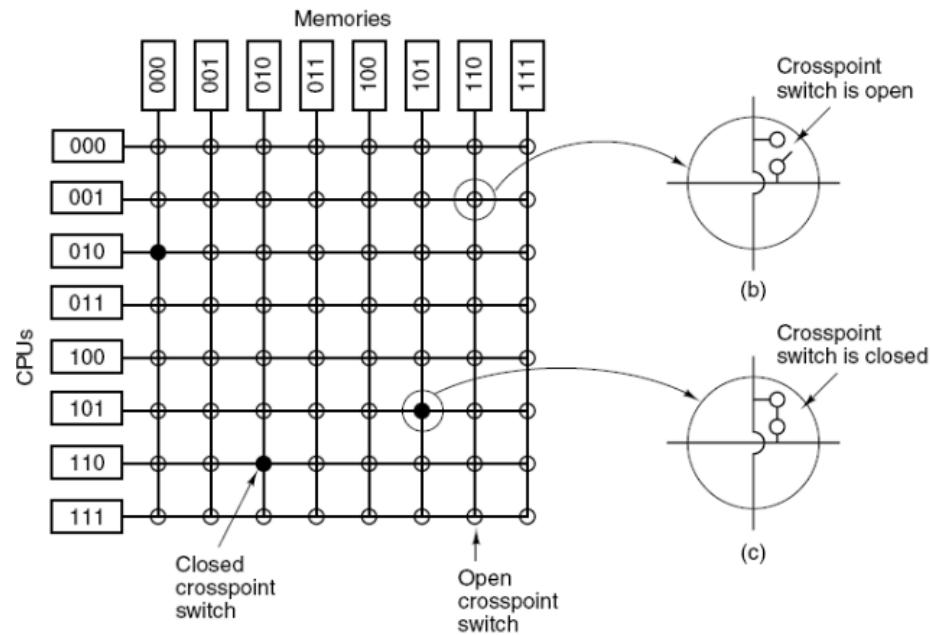
Simple bus-based

bus + cache

Bus + cache + local

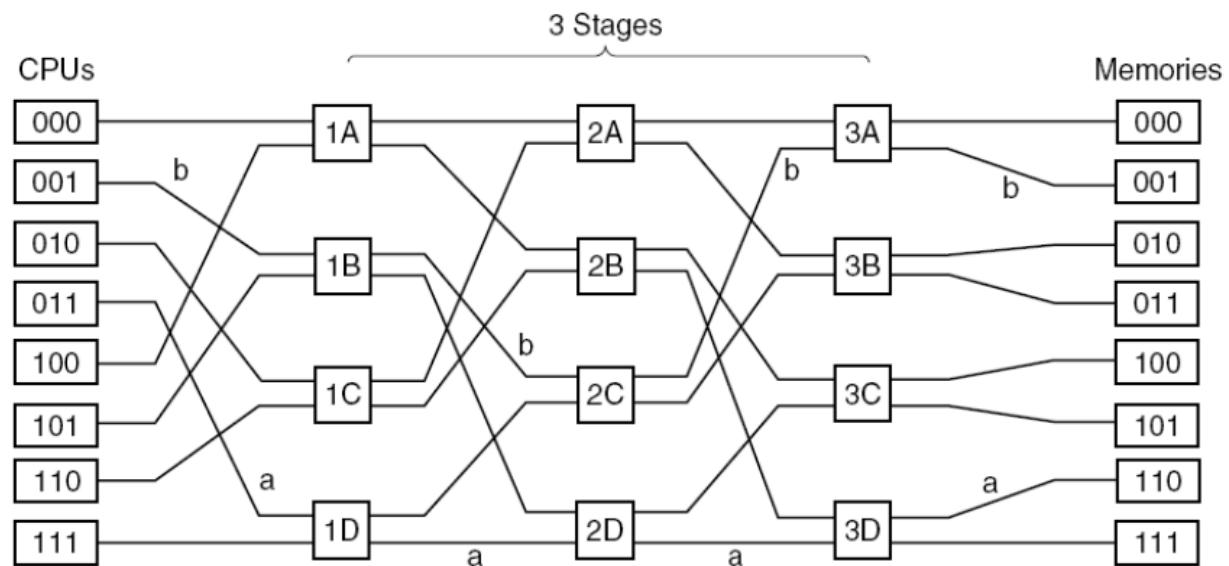
- Memory sharing. Two flavours:
 - Uniform memory access (UMA): memory independent of CPUs
 - Non-uniform (NUMA): CPUs share their (local) memory
- Two methods of connecting (UMA) memory:
 - Bus-based
 - Switched

Memory switching networks

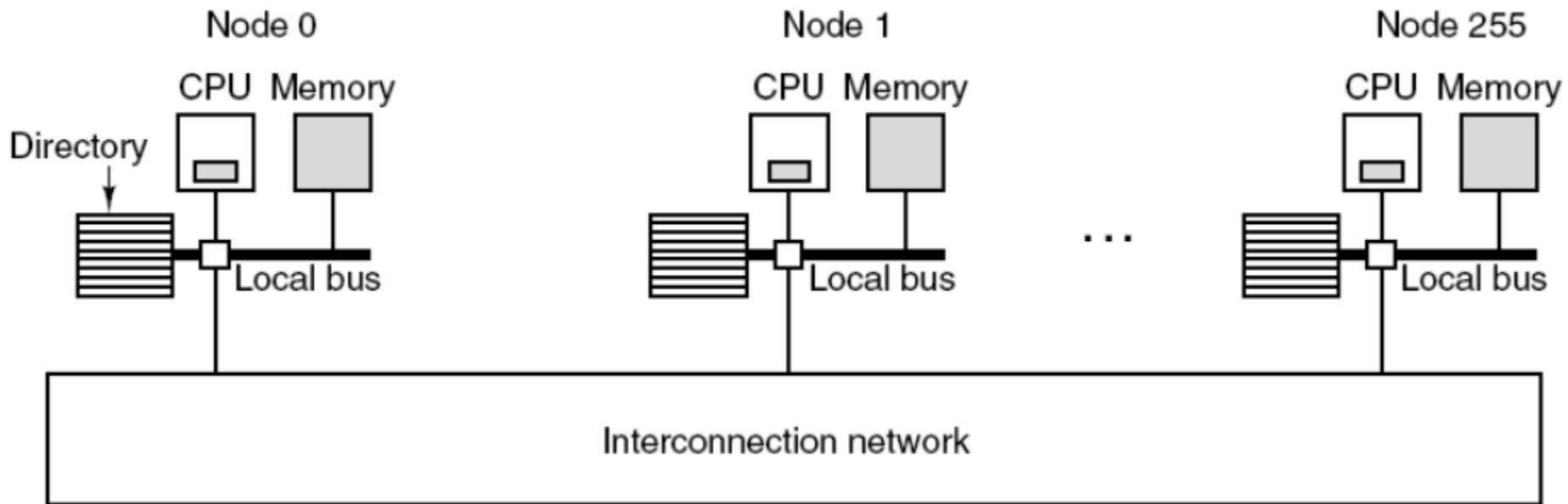


Crossbar switch

Omega network

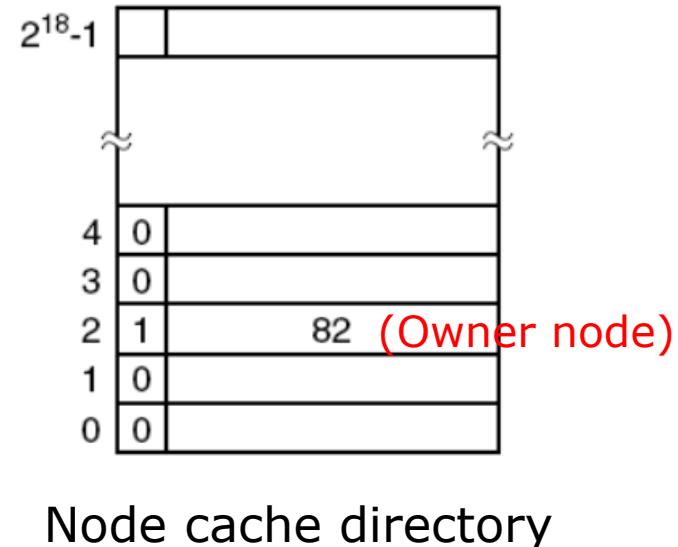
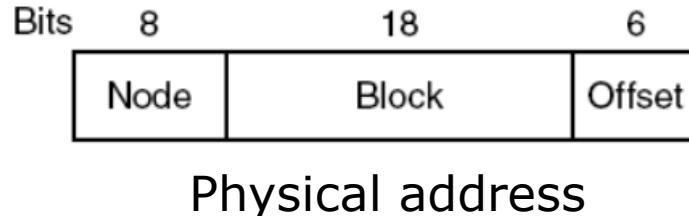


NUMA multiprocessors



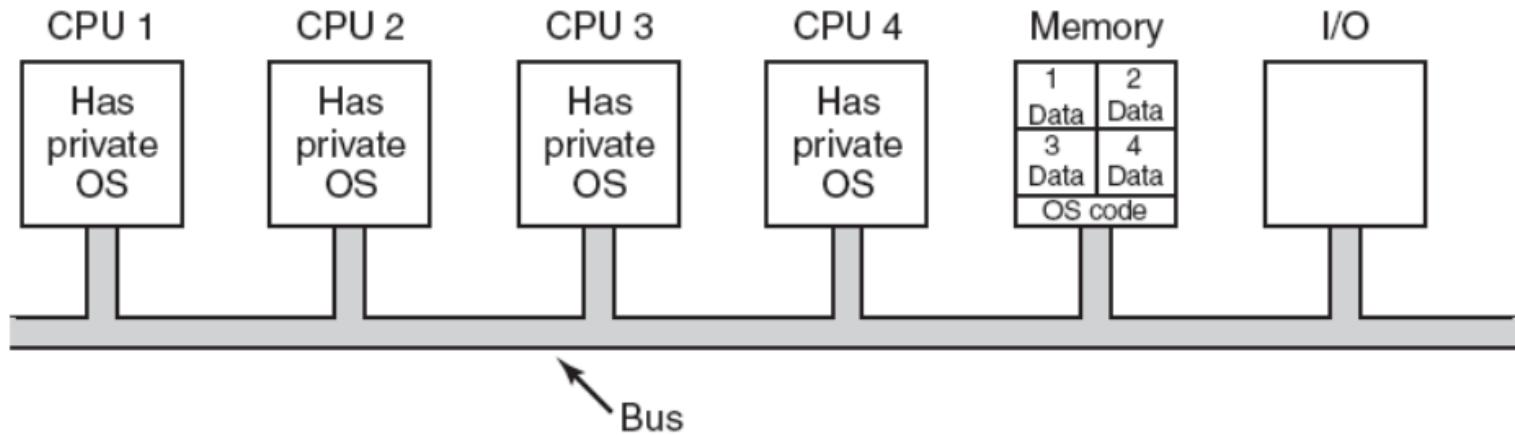
- Single physical address space statically divided between processors
- Local memory accessed quickly
- Remote memory accessed more slowly via LOAD/STORE requests
- Caching optional:
 - NC-NUMA: no cache
 - CC-NUMA: “cache-coherent” (no cache inconsistency)

Directory-based CC-NUMA



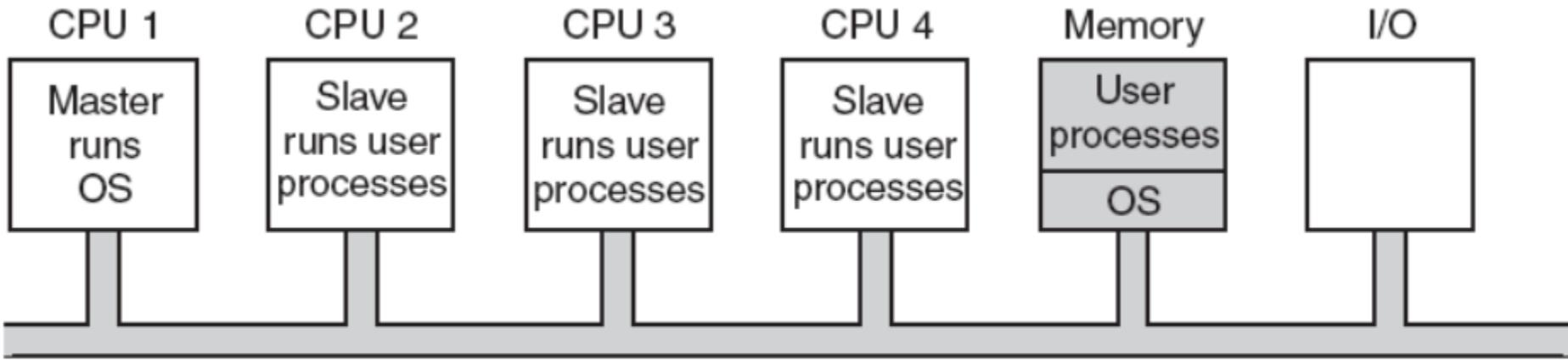
- Cache and memory distributed over the processor nodes
- Each node maintains a directory of where each cache line resides (not cached, cached locally, cached at node n)
- Memory request routed to the node responsible for that address range
 - Cached locally: return the value
 - Not cached: read from memory, send to requester, record the owning node
 - Cached *elsewhere*: record the new owning node, request old owner to pass on the value and invalidate cache

Operating system (1): private OS per CPU



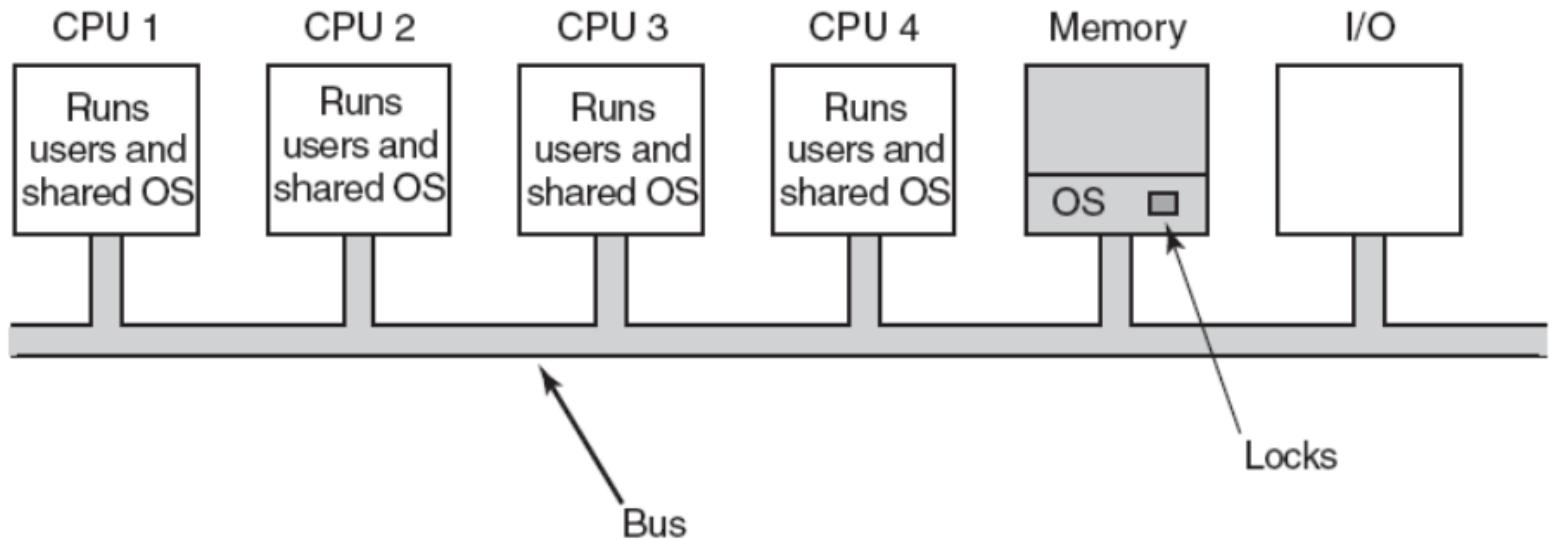
- Statically divided memory
 - Share static (read-only) OS code etc
 - Can reserve shared space for inter-process communication
 - Can allocate unevenly (e.g. one CPU dedicated to larger jobs)
- Very little resource sharing/communication
 - User has all processes allocated to one CPU (no load balancing)
 - Memory can't be shared (external fragmentation)
 - Private I/O caches (e.g. disk writes) requires coordination
- Rarely used any more

Operating system (2): master/slave



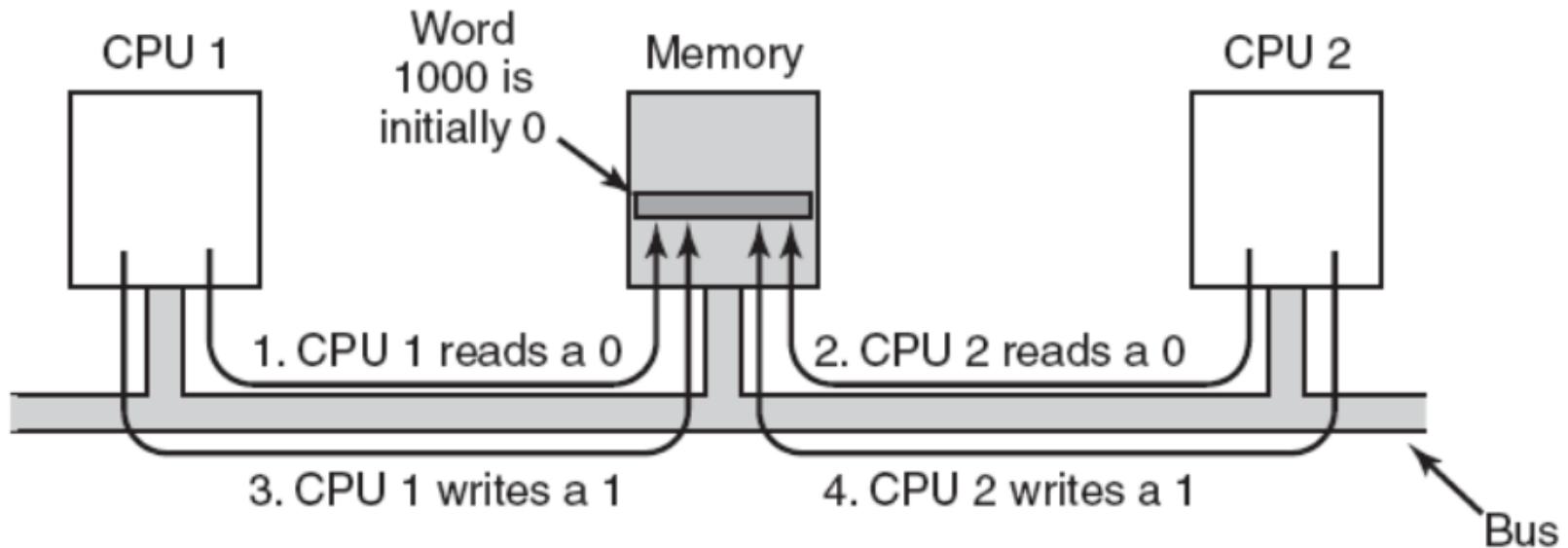
- OS runs on master only
 - Handles system calls for all CPUs
- Solves most problems with previous model:
 - Centralised load balancing (CPU job lists)
 - Dynamic memory allocation
 - Single copy of I/O buffers etc
- Problem: master is a bottleneck (10% time in system calls = 10 slaves max)

Operating system (3): SMP



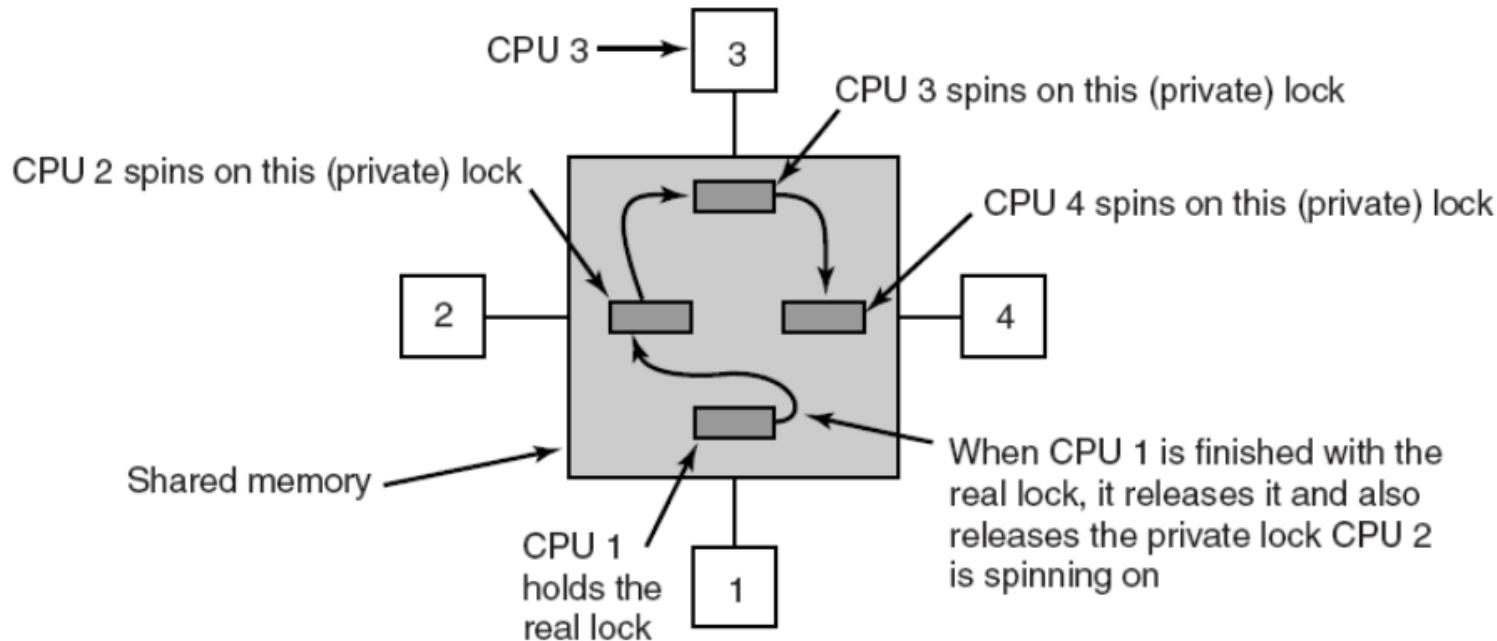
- Symmetric multiprocessing (SMP): one OS, multiple CPUs
 - OS data tables shared
 - CPUs all run user processes and OS (trap system calls)
- OS resources locked by mutexes
 - OS split into critical regions
 - Care needed to avoid deadlocks

Multiprocessor synchronisation



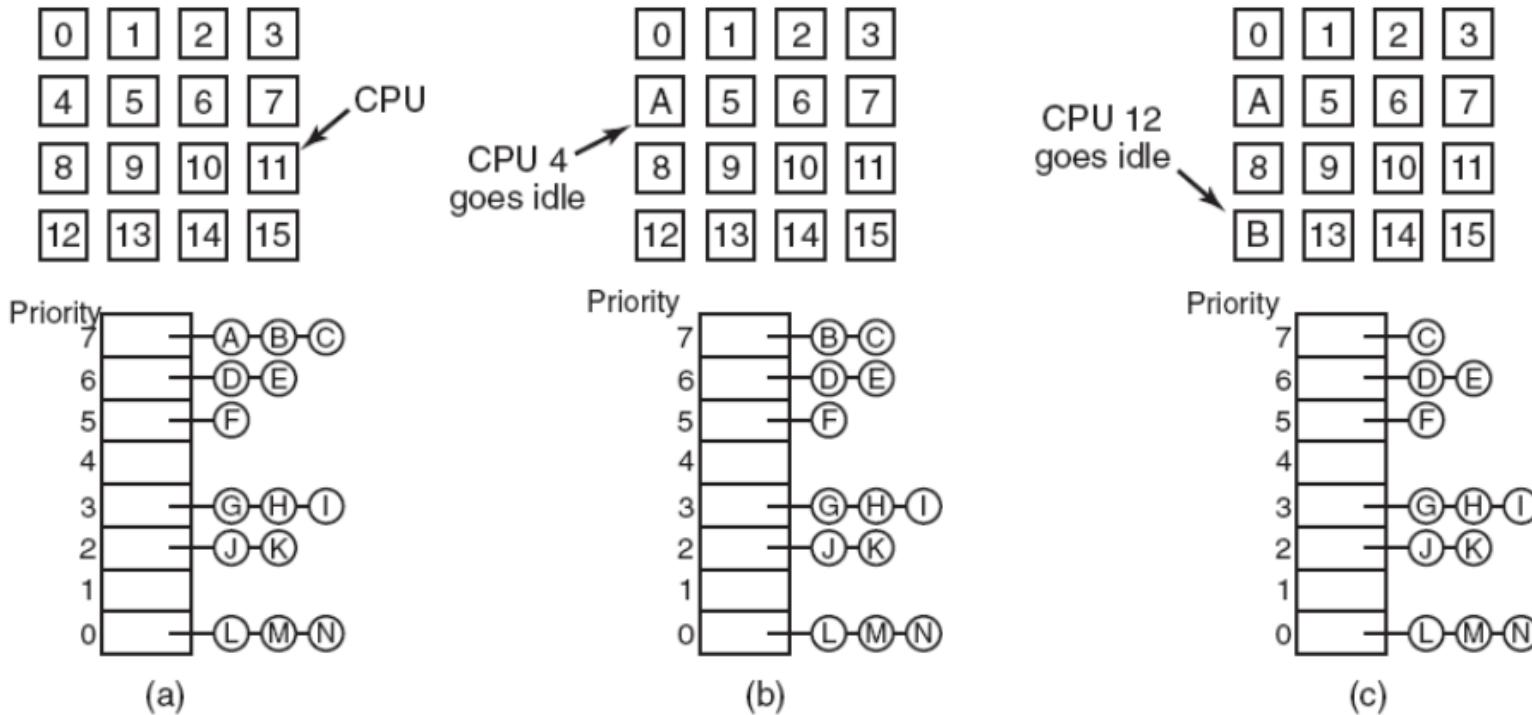
- Atomicity (e.g. Test and Set Lock) fails: interrupts local to CPU
 - Bus locking (preferred)
 - Peterson's protocol
- Problem: CPU implements spin lock waiting for the bus
 - Wastes CPU time spinning
 - Polling places heavy load on bus (or memory)
 - ▶ NOT fixed by caching (cached line thrashes between two CPUs)

Reducing thrashing



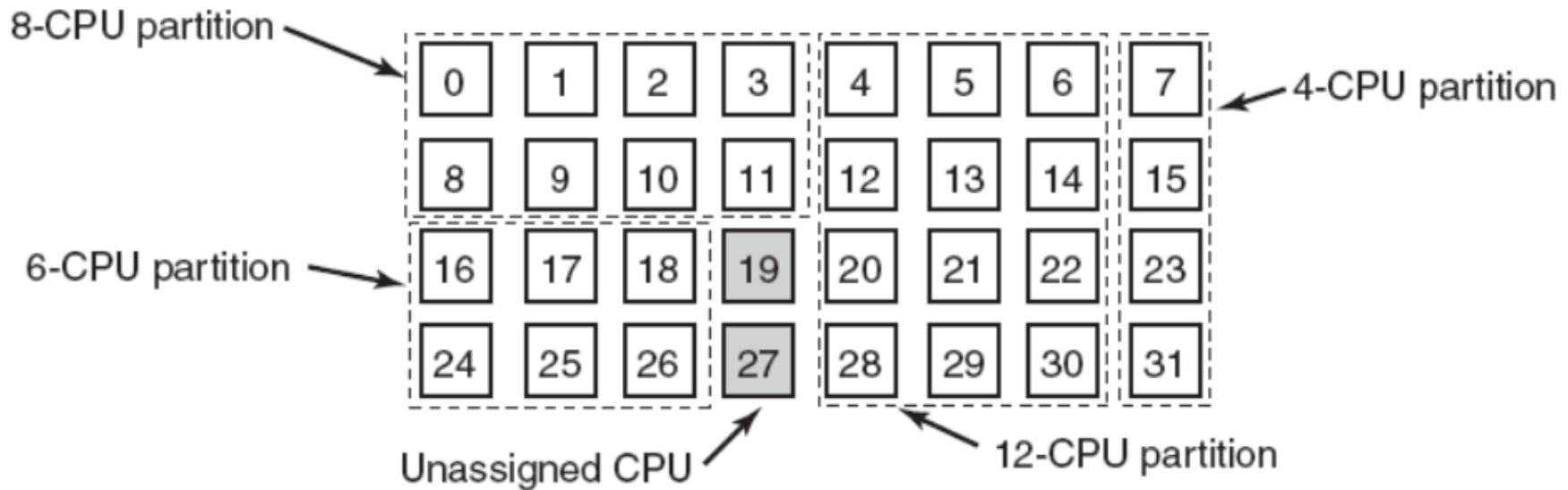
- Option 1: test the lock (read) first
 - Reads locally cached copy until it changes (invalidates)
- Option 2: exponential backoff (delay between polls, double on each fail)
- Option 3: multiple lock copies
 - Requesting CPU creates new lock in a linked list (queue)
 - Releasing CPU unlocks the next lock in the queue
- Option 4: switch processes (depends on switch versus lock wait time)

Process (thread) time scheduling



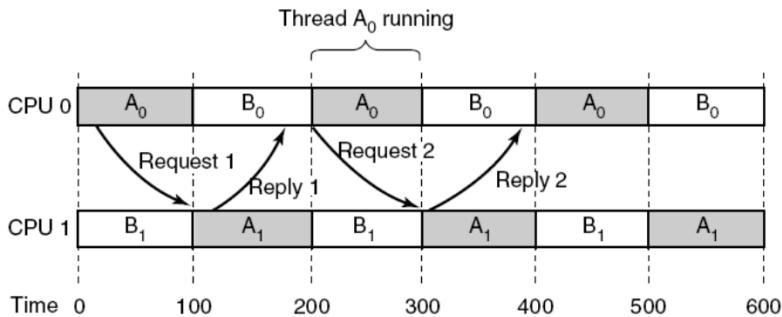
- Scheduling thread bursts across multiple CPUs: centralised process list
 - Next thread of highest priority scheduled (same as single-CPU)
- Smart scheduling: thread holding spin locks flagged for more time
- Affinity scheduling: allocate threads (not bursts) to CPUs to maximise cache use
 - CPU schedules its own allocated processes (threads)
 - Idle CPU can “steal” thread from another CPU

Process (thread) space scheduling

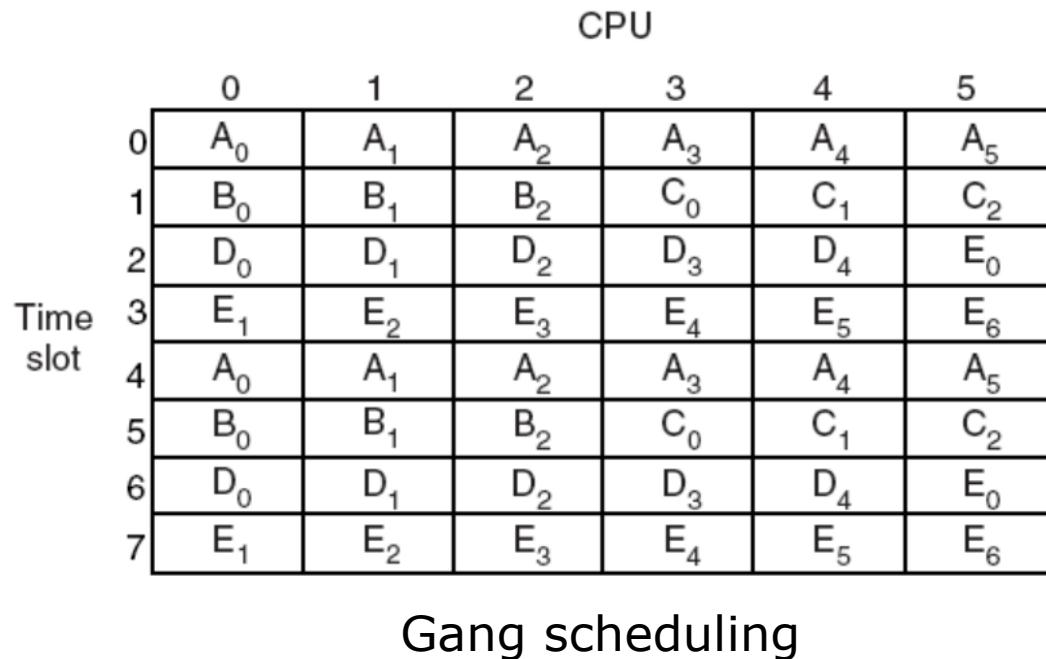


- Suitable for groups of related threads (e.g. *make* parallel compilation)
- No multiprogramming: one thread per CPU
- Threads allocated to available CPUs on creation'
 - Block if not enough CPUs free (alternative: dynamically request threads)
- Problem: I/O block wastes entire CPU

Space and time: gang scheduling



Space scheduling:
communication delays



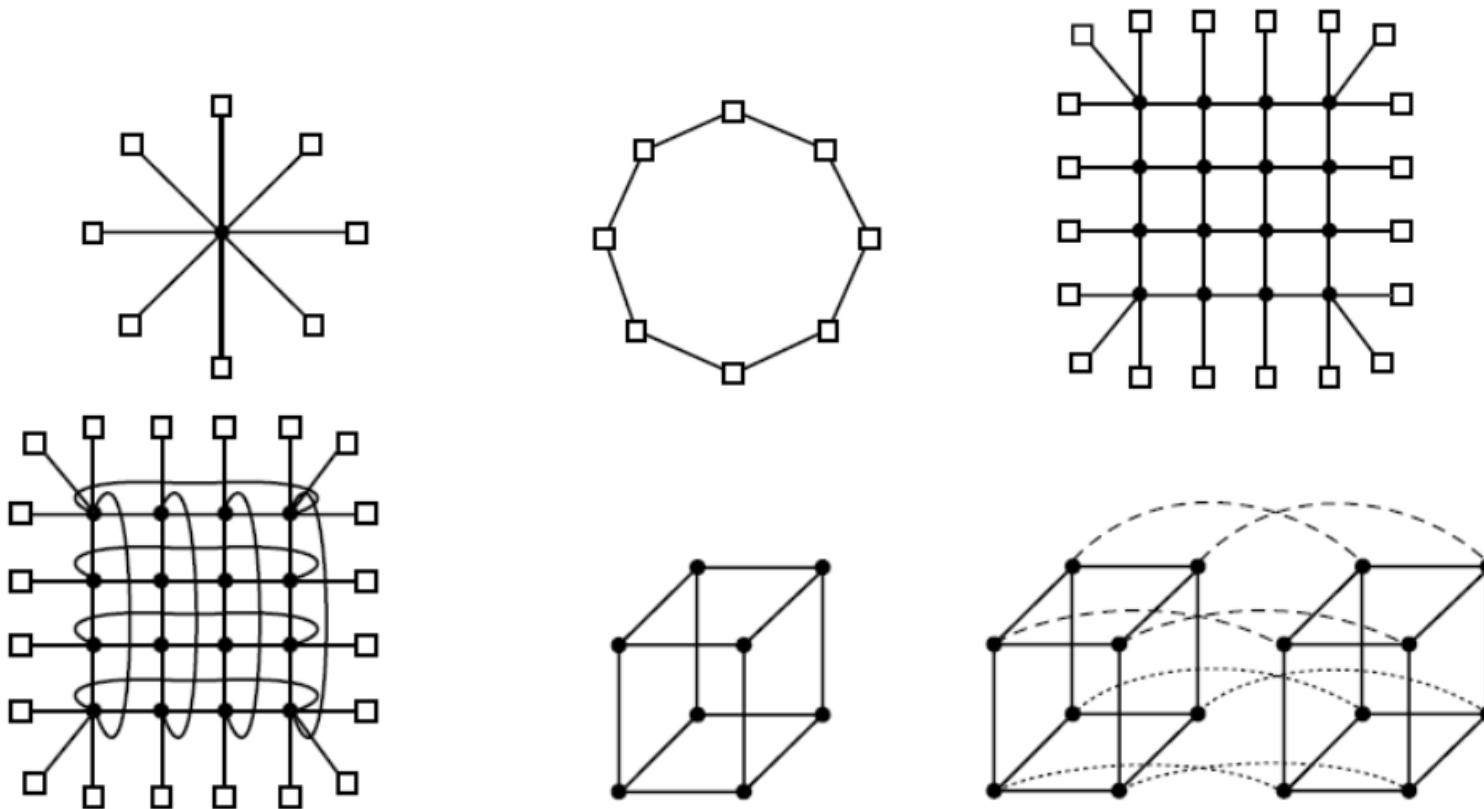
- Reduce idle time: add timesharing to space-scheduled processors
 - Problem: inter-process communication requires same time schedule
- Solution: *gang* scheduling
 - Statically (space) schedule threads every t cycles
 - I/O blocks until end of quantum only
 - Synchronised scheduling of related threads (“gang”)

Multicomputers



- Tightly coupled independent CPUs (no shared memory)
- Example: National eScience Infrastructure (NeSI)
 - Maui: Cray XC50 Massively Parallel Capability Computer
 - ▶ 464 2.4GHz 40-core processors (18,560 nodes total)
 - ▶ Memory: 96-192GB per node (66.8TB total)
 - Mahuika: Cray CS400 Capacity High Performance Computing cluster
 - ▶ 234 2.1GHz 36-core processors (8,424 nodes total)
 - ▶ Memory: 128GB per node (30TB total)
- Key issue: inter-processor communication

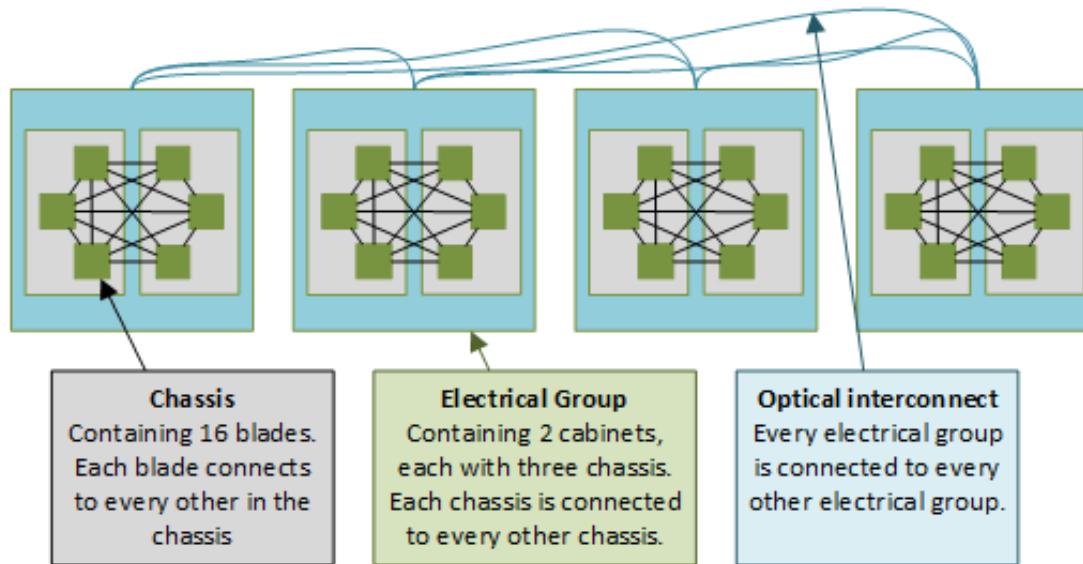
Interconnect topologies



- Various topologies: single switch, ring, grid, double torus, cube, hypercube
- Tradeoff between distance (time) and number of connections. Example: 1024 nodes:
 - Grid: 32x32, diameter = 64
 - 10d hypercube: $\log_2(1024) = 10$

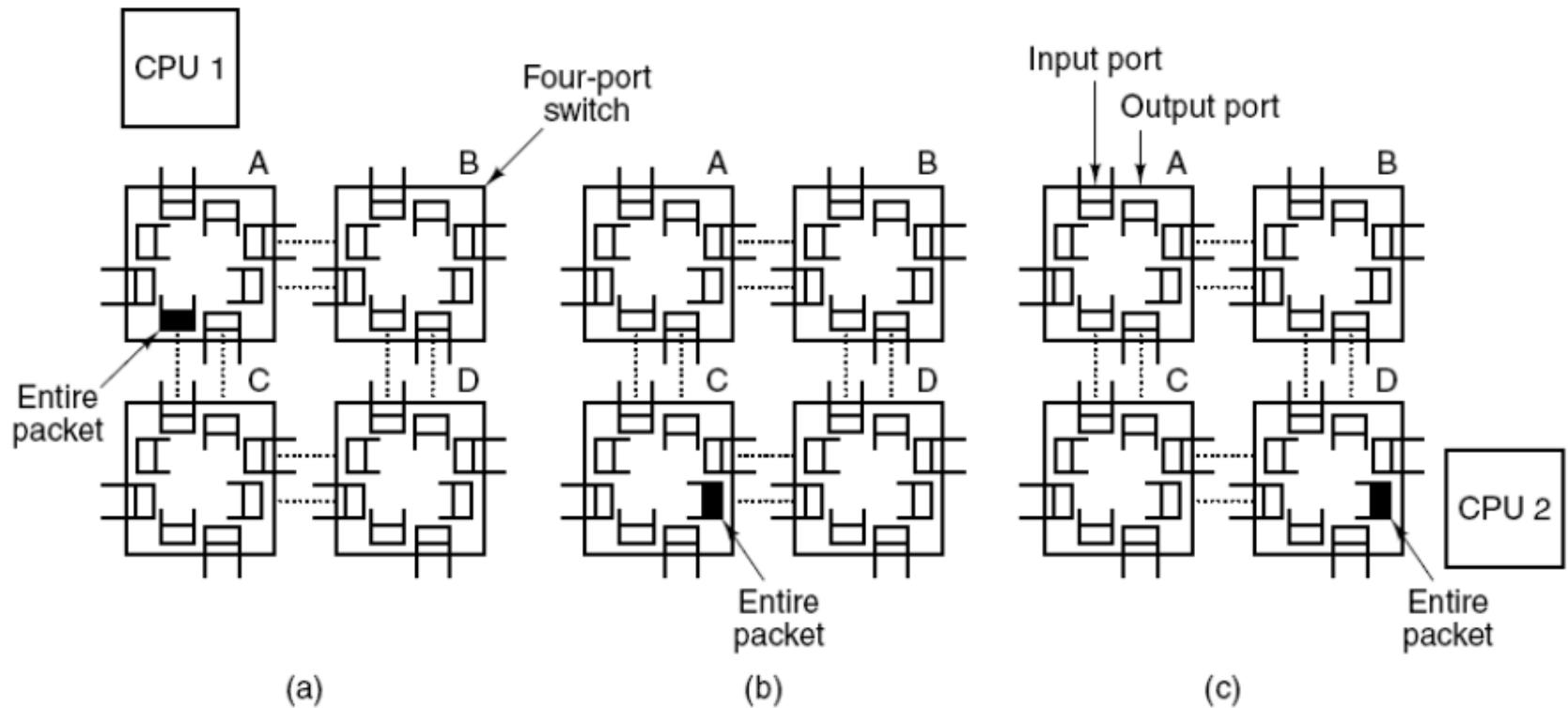
NeSI's Maui: “dragonfly” topology

Aries interconnect for an 8 cabinet Cray XC40



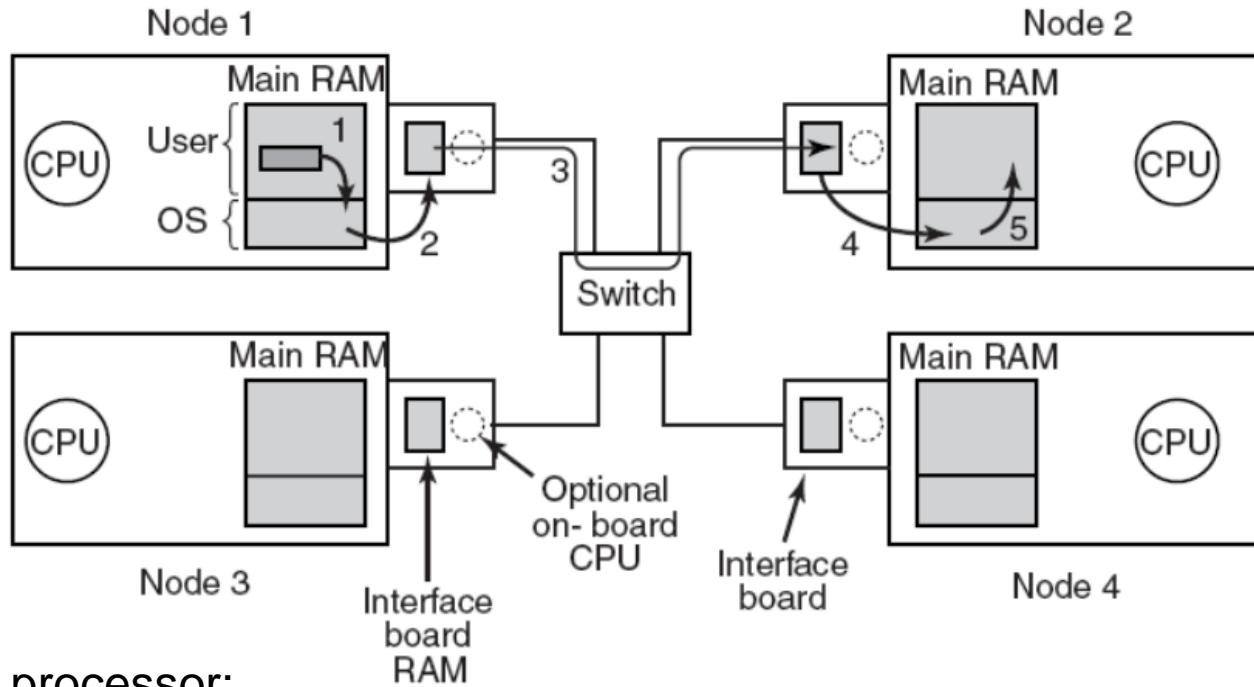
- Hierarchical topology
- Four (two-CPU) nodes per blade, 16 blades per chassis
 - Blades, chassis and electrical groups (6 chassis) each fully connected (fast)
- Electrical groups fully connected via optical (slower) interconnects
- Locality: very fast for small node groups, slower for very large ones
 - Divide threads into related groups (c.f. *space scheduling*)

Message routing



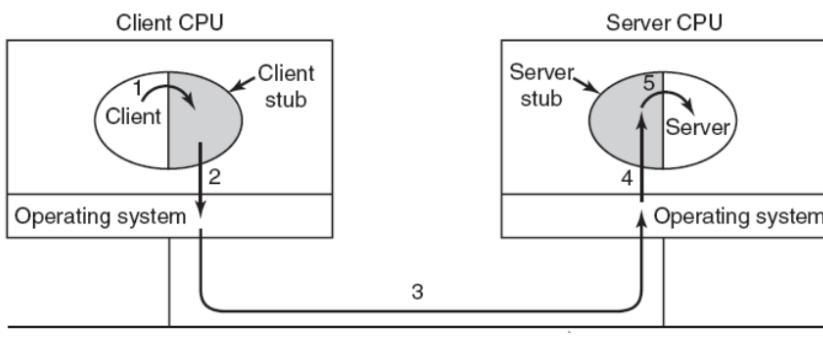
- Store-and-forward: each switch buffers a packet, then forwards it on
- Alternative: circuit switching
 - First switch establishes path to the destination
 - Bits “pumped” to destination; no buffering en route
 - “wormhole” routing: start sending sub-packets before route completed

Network interface

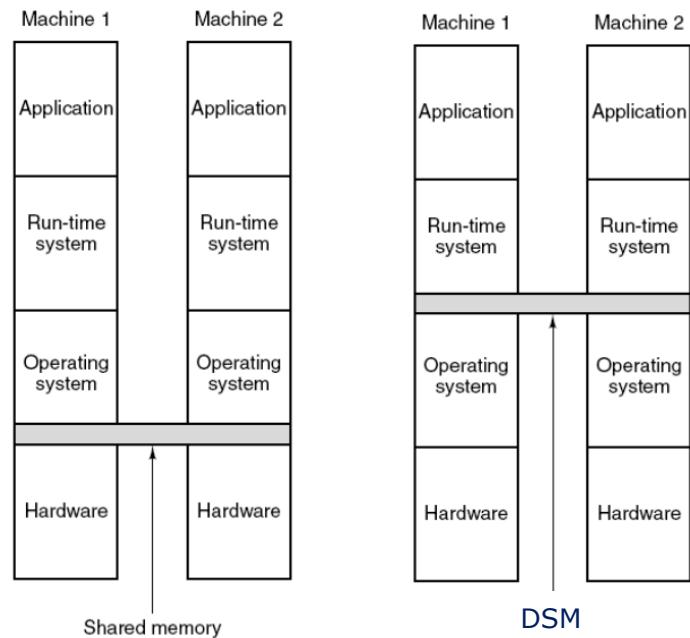


- Network processor:
 - Dedicated RAM: buffer between synchronous/asynchronous data access
 - Optional CPU: handles retries, multicasting, compression, protection
 - ▶ Requires synchronisation with main CPU
- Problem: multiple (5) data copies – slows down transfer rate
 - Map interface buffer to *user* space (one process, or partition the buffer)
 - Separate network interface for kernel (e.g. for remote file access)

User-level communication software



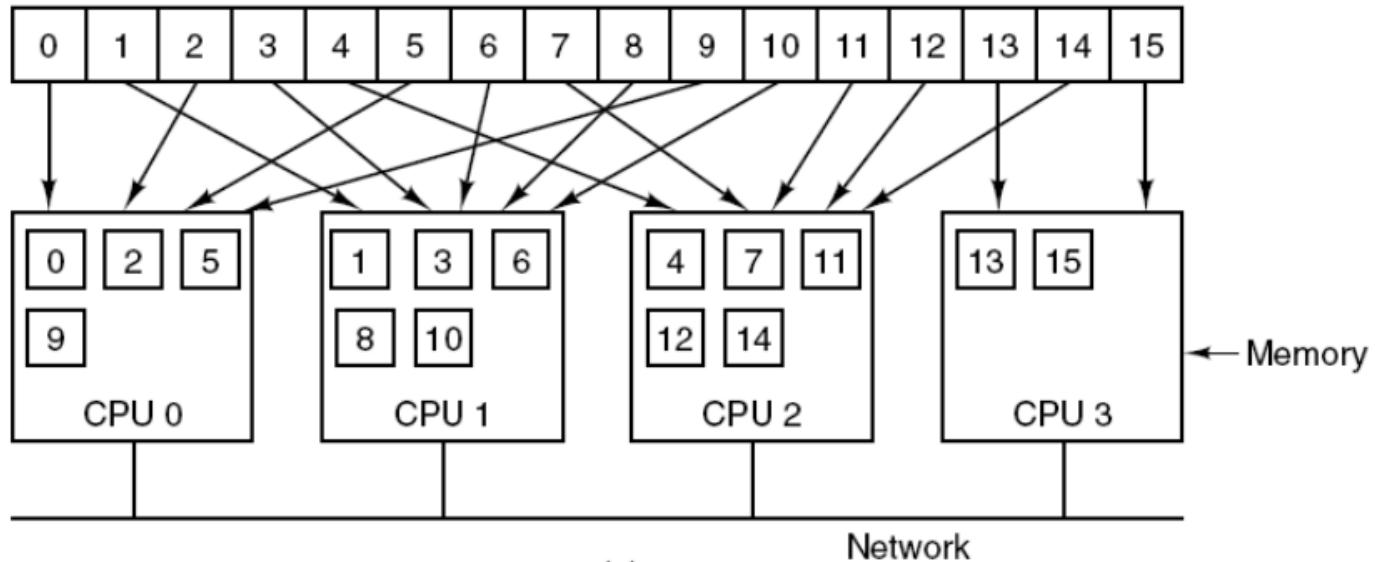
Remote procedure call (RPC)



- Message passing: send and receive (e.g. MPI)
- Remote procedure calls: execute code on remote CPU
 - Stub hides marshalling/unmarshalling of messages
 - Limitations to datatypes (static structures) and functions (strongly typed)
- Distributed shared memory (DSM)
 - Emulation of true shared memory

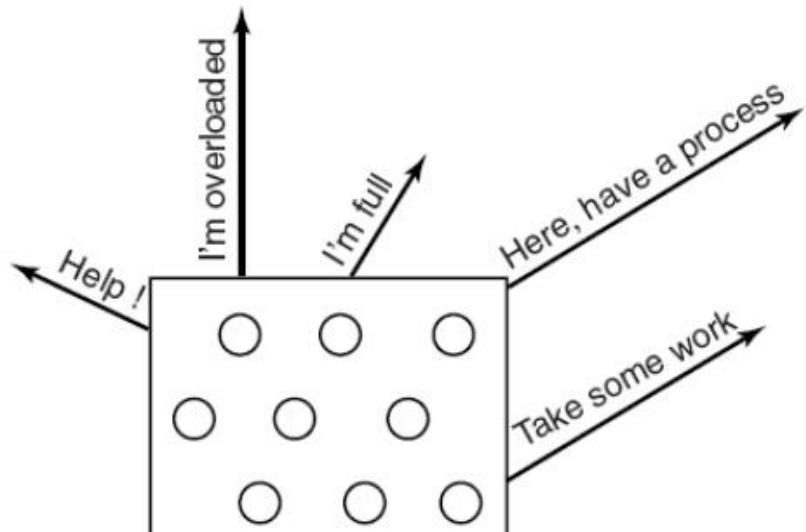
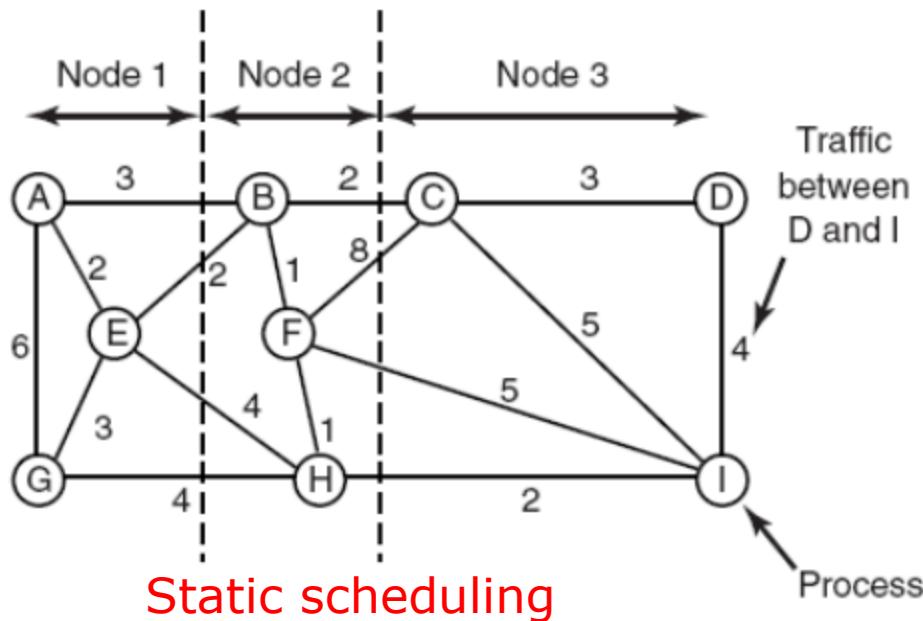
Distributed shared memory tuning

Globally shared virtual memory consisting of 16 pages



- DSM emulates shared memory across a network
 - Page not found locally triggers a fault, fetches from another node
 - Problem: SLOW!
- Solution: minimise page transfers
 - Replicate pages, invalidate other copies on write or journal the changes
 - Increase the page size
 - ▶ Problem: false sharing (unrelated variables thrash between nodes)
 - ▶ Compiler optimises by grouping related variables

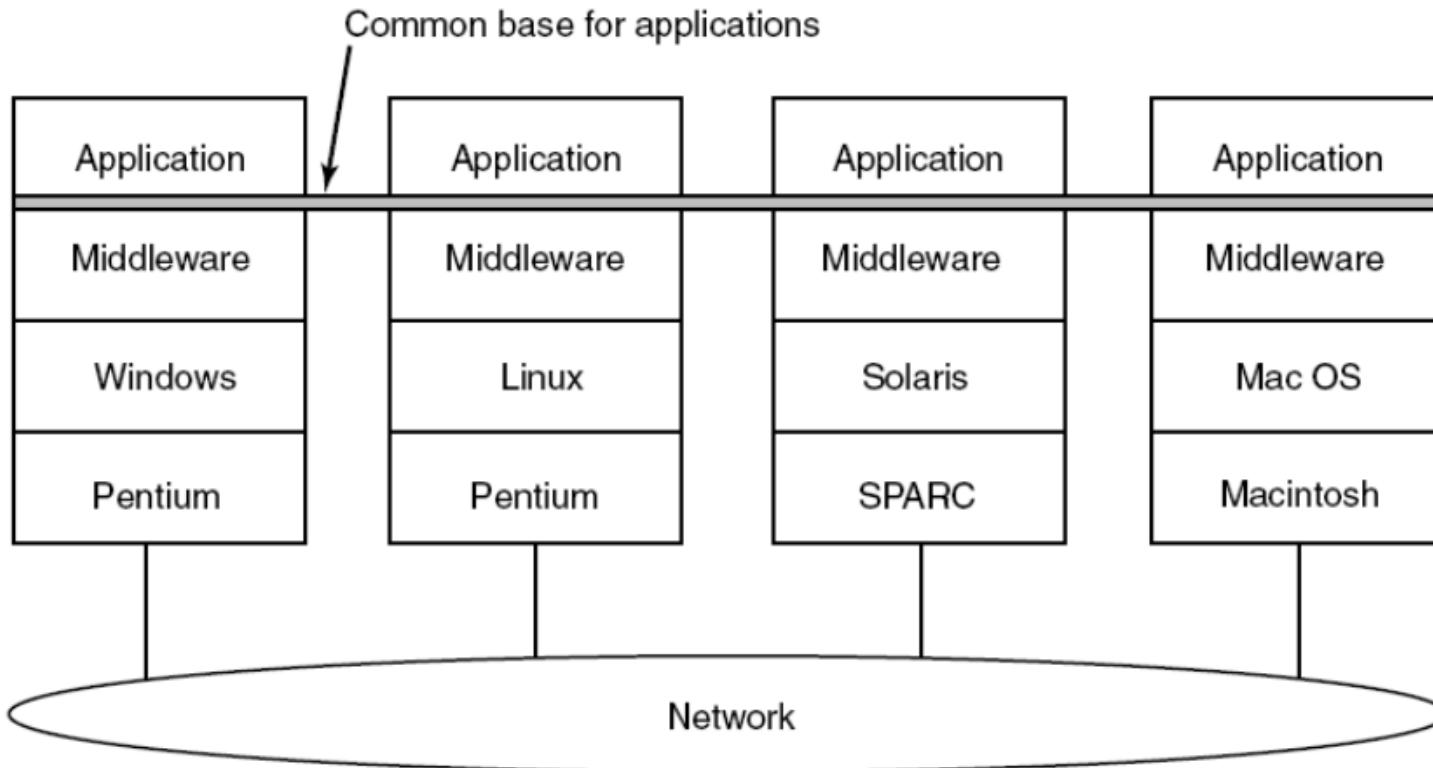
Process scheduling



Dynamic scheduling

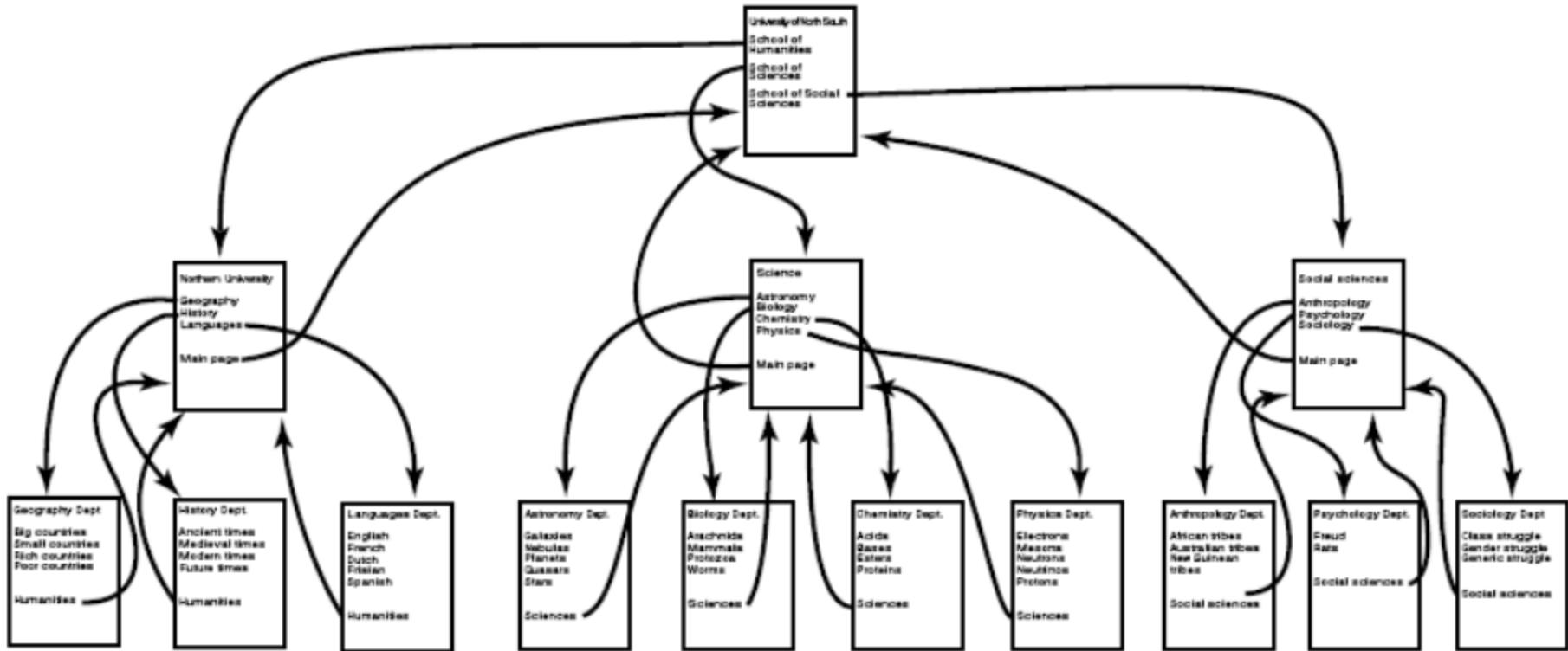
- Goals: minimise wasted CPU cycles/communication, fairness to users
- Static scheduling
 - Graph-theoretic approach
 - Goal: minimise network traffic between nodes
- Dynamic scheduling: hand off process from overworked to underworked nodes
 - a) Overloaded node asks for help
 - b) Underworked node asks for work

Distributed systems



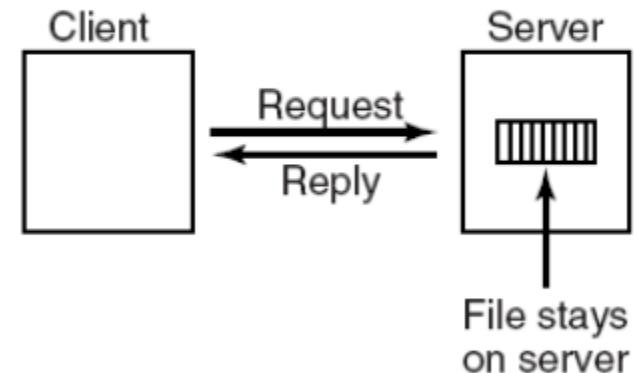
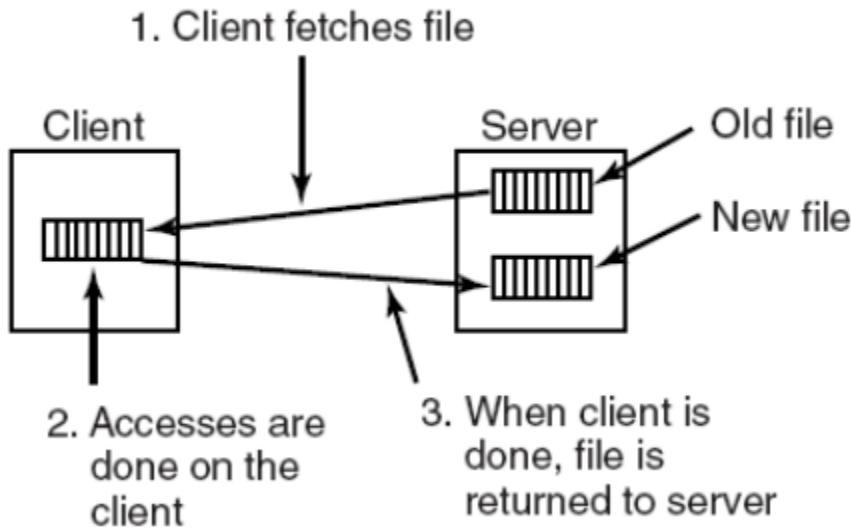
- Loosely coupled multi-computer system:
 - Connected by internet (very slow)
 - Different operating systems
 - Non-dedicated machines
- Most of the previous techniques don't apply
 - Computers cooperate via middleware (high-level communication abstraction)

Document-based: the World Wide Web



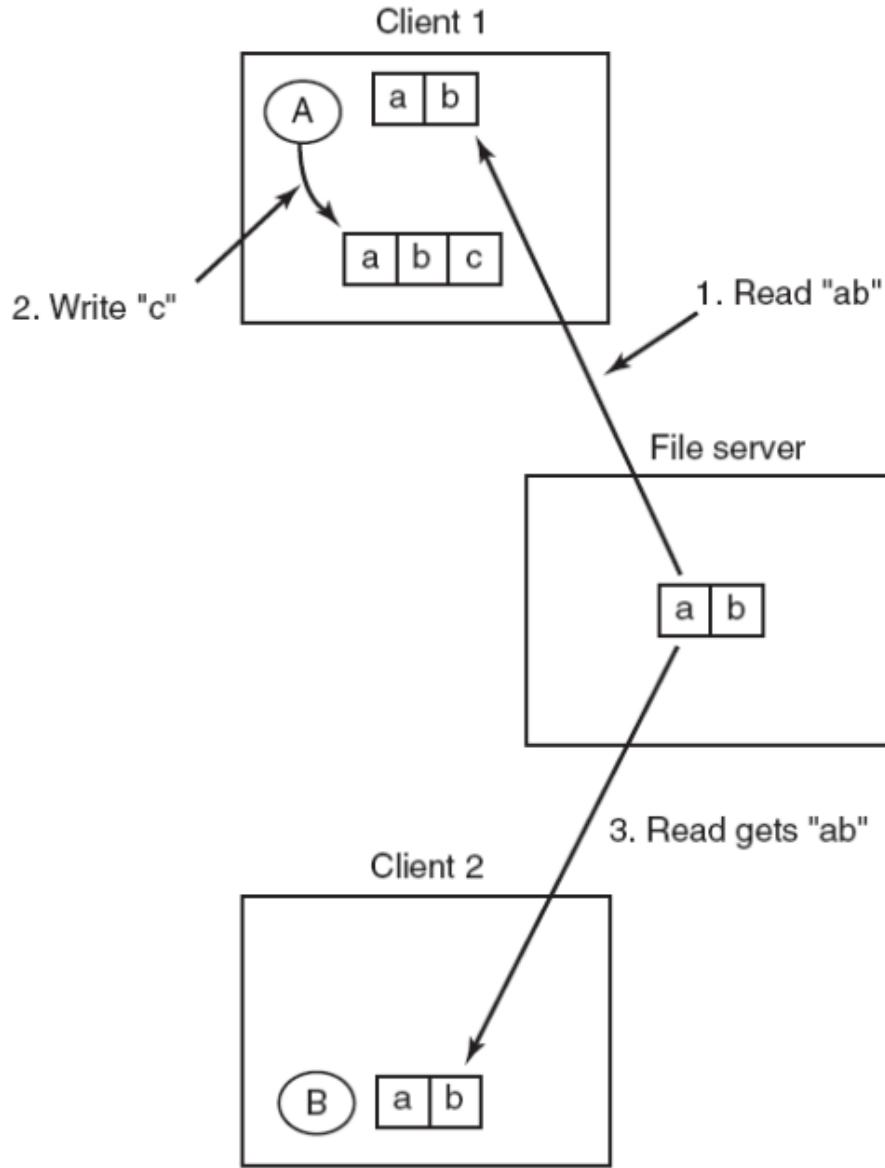
- Memory abstraction: web page
 - In cache: read locally
 - Not in cache/out of date: retrieve from another node
- Location-dependent naming

File-based: global file system



- Files spread over multiple nodes
- Two models of remote access:
 - Upload/download: download a copy from the remote node, upload if any changes
 - ▶ Inefficient for random access
 - ▶ Synchronisation issues when multiple writers
 - Remote access: reads and writes performed by remote node (e.g. RPC)
 - ▶ Potentially slower if a lot of I/O operations

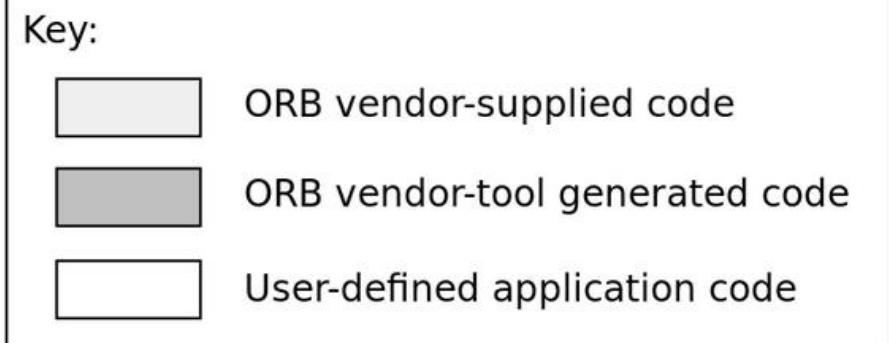
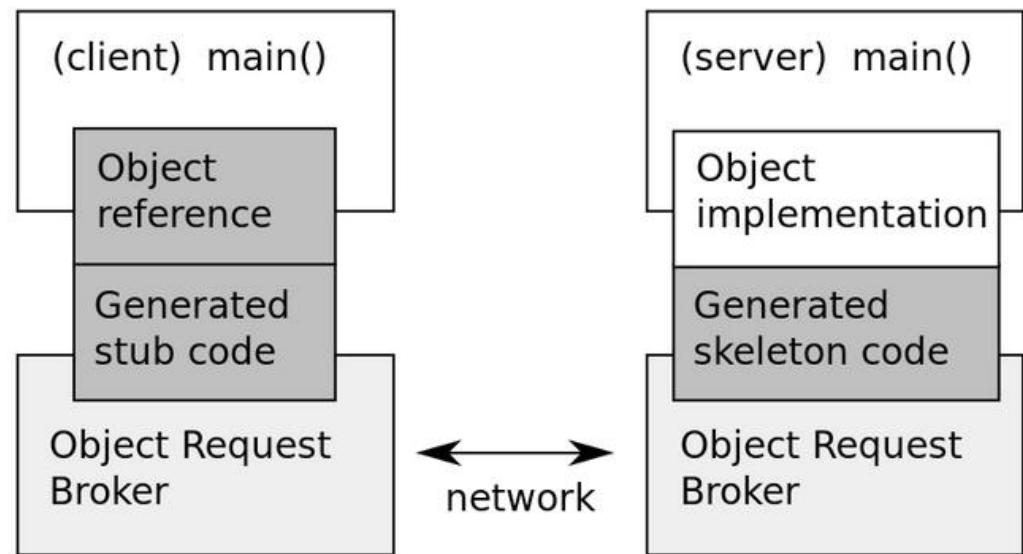
Distributed file system issues



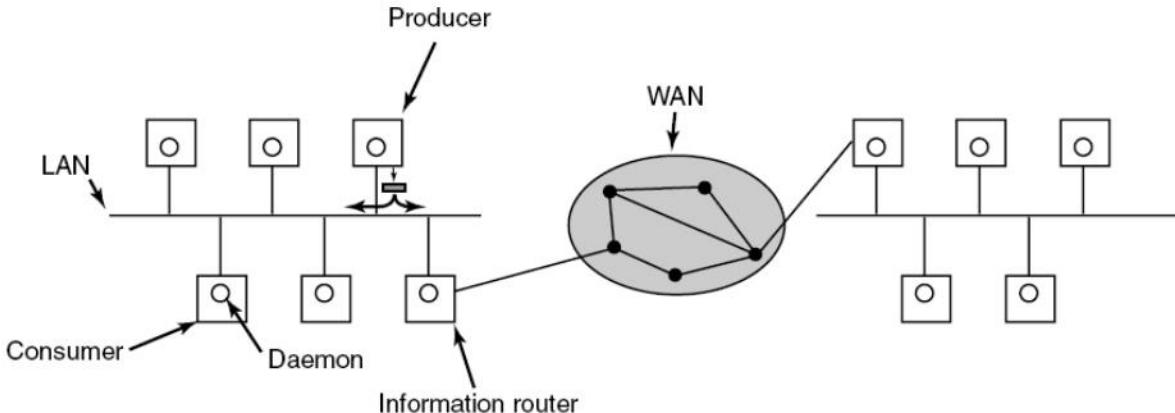
- File sharing semantics
 - Sequential consistency: propagate all writes immediately
 - Session consistency: propagate on file close
 - ▶ Other readers see old copy
 - ▶ Multiple writers compete for the final copy
- Naming:
 - Machine+path: easy to implement
 - Location-independent: allows load/space balancing
- Directory structures:
 - Remote mounting: different machines see different file systems
 - Global view: harder to implement

Object-based: CORBA

- Object Request Broker (ORB) routes requests to servers
- Interface definition (IDL) compiled into client stub
- Objects generated on server, returning a reference
- Communication details completely hidden
 - Type, location of server
 - Language of server code
 - Compression
 - Load balancing (object splitting, method forwarding)



Coordination-based middleware



- Processes swap messages via a global pool (Shared, virtual, associative memory)
- Linda (1983): read and write to a shared *tuple space*
 - Data objects are arbitrary tuples of values
 - Retrieved by pattern matching
- Publish/subscribe: publishers broadcast messages to subscribers on the network
- Jini (Apache River): network-centric model
 - Devices communicate via a *JavaSpace*
 - Jini device requests a lookup service, receives registration *code*
 - ▶ Supplies *proxy code* for others to use to access it
 - Messages passed through *JavaSpace* via templates (like Linda)

Summary

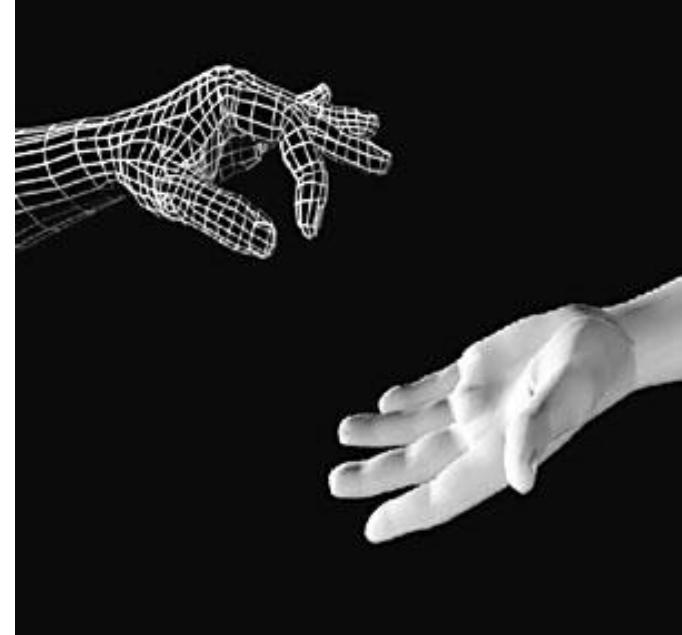
Item	Multiprocessor	Multicomputer	Distributed System
Node configuration	CPU	CPU, RAM, net interface	Complete computer
Node peripherals	All shared	Shared exc. maybe disk	Full set per node
Location	Same rack	Same room	Possibly worldwide
Internode communication	Shared RAM	Dedicated interconnect	Traditional network
Operating systems	One, shared	Multiple, same	Possibly all different
File systems	One, shared	One, shared	Each node has own
Administration	One organization	One organization	Many organizations

- Operating concept extended to inter-process communication
- Concepts and approaches depend on level of coupling:
 - Multiprocessor: shared memory
 - Multicomputer: message passing
 - Distributed system: high-level abstraction (files, objects, requests)
- Tuning decisions dominated by cost of communication

ENCE360

Operating

Systems

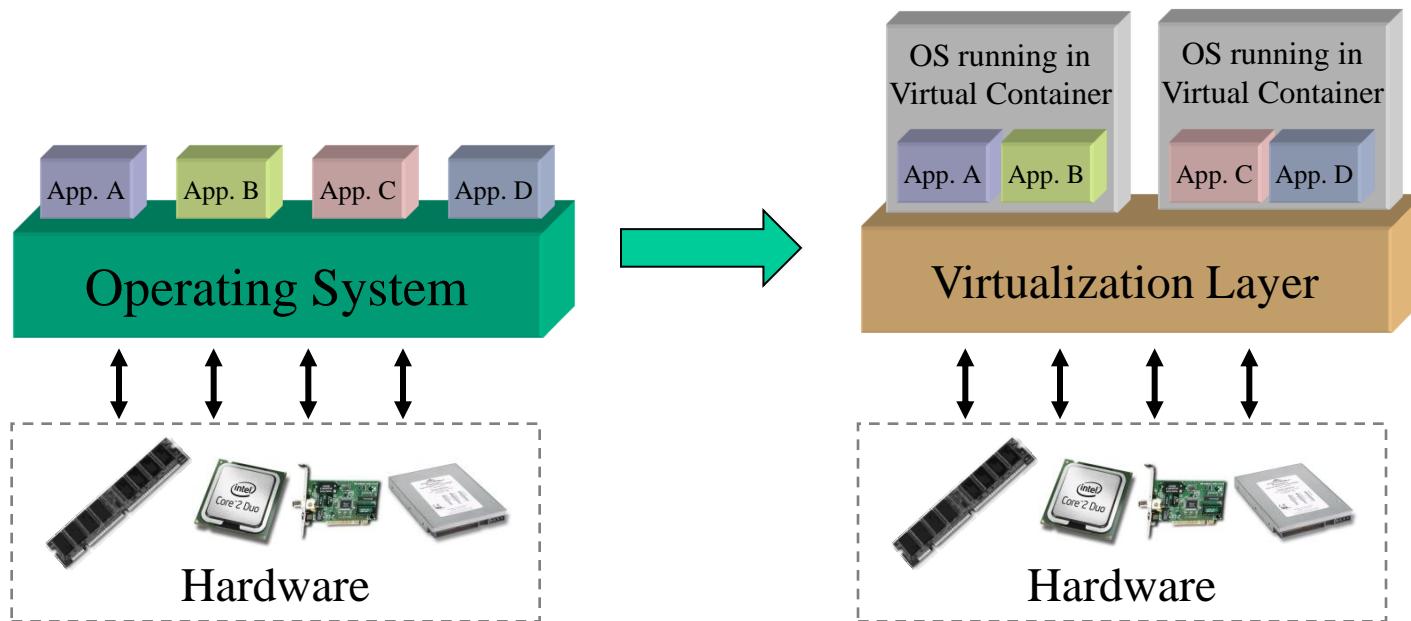


Virtualization

MOS (3rd ed.) CH 8

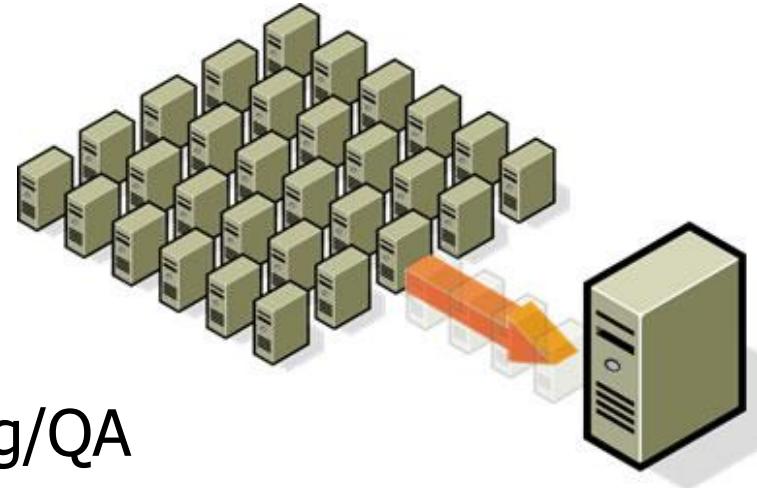
What is virtualization?

- Multiple “virtual computers” on a physical machine
- Each has own copy of the operating system
 - May be different operating systems (e.g. Linux and Windows)



Virtualization: Why?

- Server consolidation
- Application consolidation
- Virtual hardware
- Sandboxing and debugging/testing/QA
- Load balancing
- Redundancy and failover
- Multiple execution environments (legacy)
- Software migration (mobility)
- Software as an appliance





Oracle VM VirtualBox Manager

Details Snapshots

 Windows-7
Running Ubuntu-8.10
Powered Off 01_test_1
Saved 02_test
Powered Off 02_test_2
Powered Off Windows-XP-Dev
Saved Fedora-14
Saved ubuntu
Powered Off Solaris
Powered Off MSDOS (Snapshot 2)
Powered Off Win-XP-SP3
Powered Off ee
Powered Off

General

Name: Windows-7
OS Type: Windows 7

System

Base Memory: 1024 MB
Boot Order: Floppy, CD/DVD-ROM,
Hard Disk
Acceleration: VT-x/AMD-V, Nested
Paging

Display

Video Memory: 128 MB
Remote Desktop Server: Disabled

Storage

Audio

Host Driver: CoreAudio
Controller: ICH AC97

Network

Adapter 1: Intel PRO/1000 MT Desktop (NAT)

Serial Ports

USB

Device Filters: 0 (0 active)

Preview



Windows-7 [Running]

DE German (Germany) ? Help

Donnerstag
27

Januar 2011

15:40

02.02.2011



Left X

Virtualization gotchas

- CPUs control what user and kernel mode can do:
 - **Sensitive** instructions can only be run in kernel mode
 - **Privileged** instructions trap when executed in user mode
 - Virtualisation requires all sensitive instructions to be privileged (not always the case)
- Virtual machines:
 - *think* they are running in **kernel** mode
 - are actually running in **user** mode (virtual kernel)
- Other resources also need to be shared
 - Memory
 - I/O devices

Virtualization Requirements

Popek and Goldberg: Formal Requirements for Virtualizable Third Generation Architectures – 1974:

- Properties for a Virtual Machine Monitor
 - Equivalence: A program running under the VMM should exhibit a behavior essentially identical to that demonstrated when running on an equivalent machine directly.
 - Resource control: The VMM must be in complete control of the virtualized resources.

- Formal analysis described through 2 theorems:

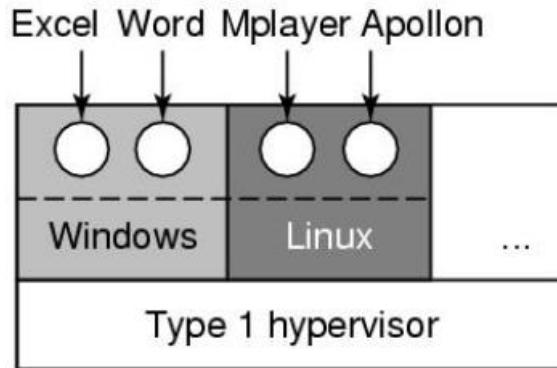
Theorem 1:

A VMM may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions. Non privileged instructions must instead be executed natively.

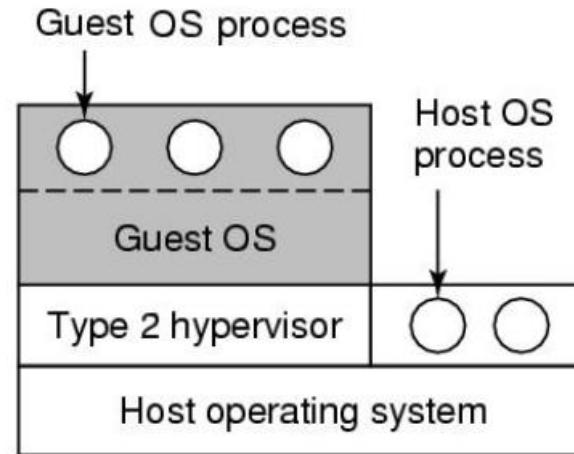
Theorem 2:

A [computer] is recursively virtualizable if it is virtualizable and a VMM without any timing dependencies can be constructed for it.

Hypervisors



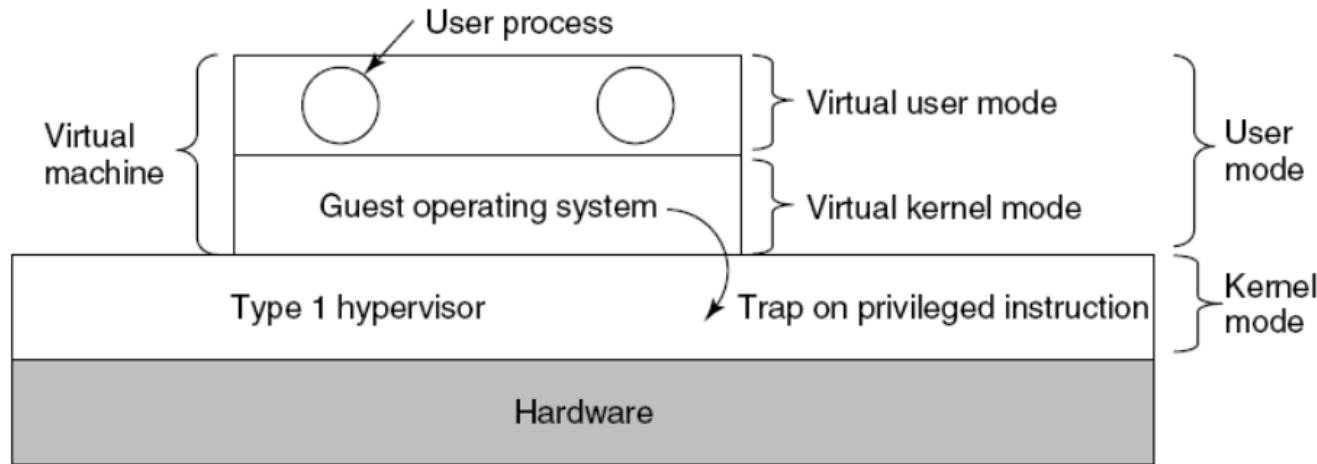
(a)



(b)

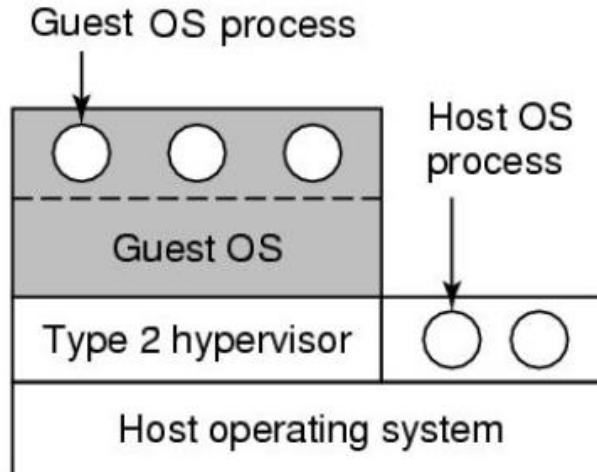
- AKA a Virtual Machine Monitor (VMM)
- Type 1 hypervisor:
 - runs on bare machine (hardware-assisted)
- Type 2 hypervisor:
 - runs on another OS (full virtualisation)
 - May include hardware assistance

Type 1 hypervisor



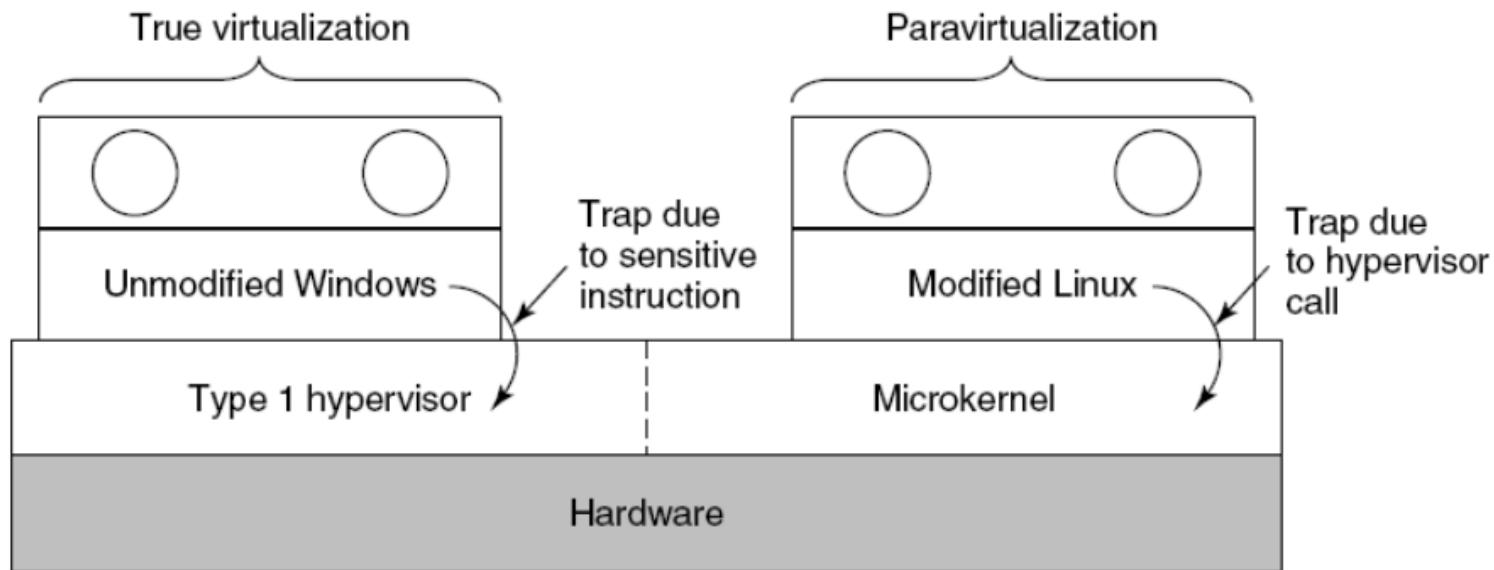
- Hypervisor runs on hardware
- Requires VM extensions (e.g. Intel Virtualization Technology – Intel-VT) that trap to the hypervisor:
 - Virtual machine runs in a container
 - Hardware traps sensitive instructions and traps to the hypervisor
 - Hypervisor emulates the actual call

Type 2 hypervisors (e.g. VMware)



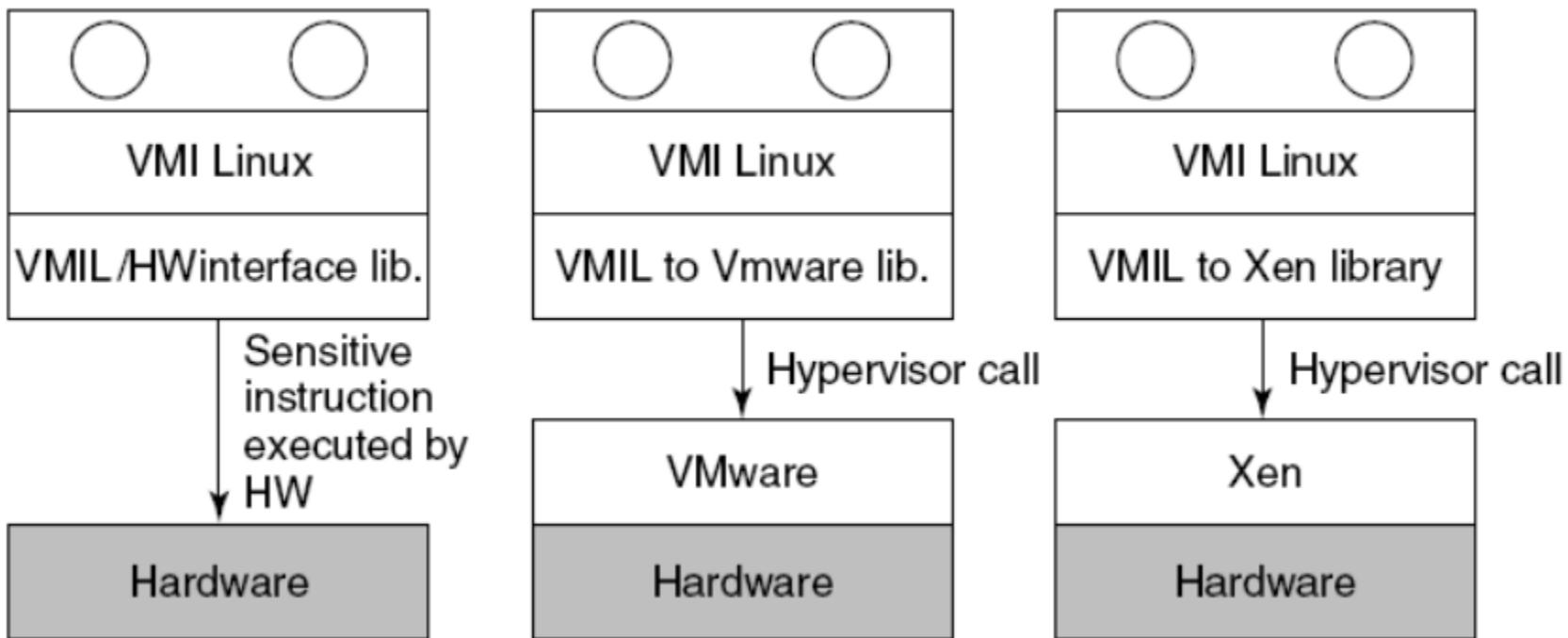
- Runs on top of host operating system
 - Doesn't require hardware support (but *may* use it)
- Sensitive instructions: **binary translation** (on execute):
 - Scan binary for *basic blocks* (no branching)
 - Replace all sensitive instructions with call to hypervisor trap function (emulates the real system call)
 - Cached non-sensitive blocks run as fast as native code
 - Can be faster than hardware assisted (fewer traps)

Paravirtualisation



- Operating system modified (source code)
 - All sensitive instruction calls replaced with hypervisor calls
 - Guest OS is VM-aware, can act accordingly
 - Goal: OP-independent *Virtual Machine Interface* (VMI)

VMI Linux

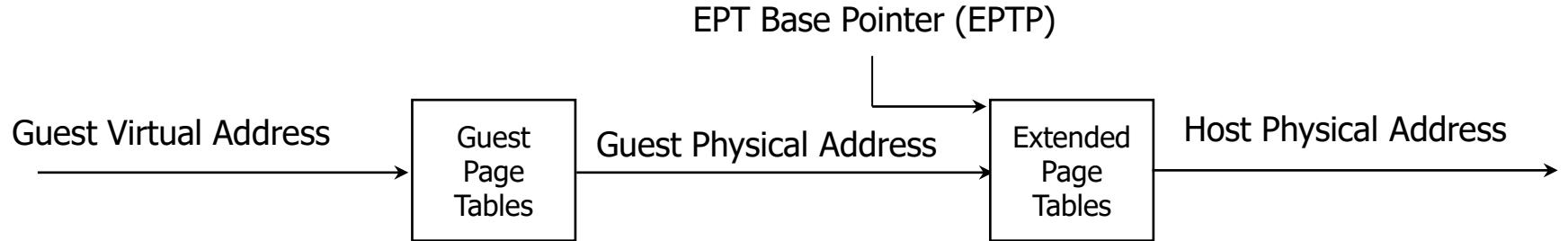


- Hypervisors supply a “VMI ROM”
- VMIL libraries provide system interface:
 - If VMI ROM present, use it
 - Otherwise native calls to hardware
- Almost zero performance overhead

Virtualised memory (1): page table shadowing

- Naïve solution: two translations performed for every memory address
 - Guest OS virtual -> guest OS physical (VMM virtual)
 - VMM virtual -> physical
- *Shadow page tables*: VMM traps translations
 - Guest OS translates virtual address to (virtual) physical address
 - *Hypervisor OS* performs further translation
 - Trap every reference or update to page tables
 - Maintain “shadow” page tables (one per guest VM)
 - Map guest OS virtual address to a real physical address and pass back to guest OS
- Significant overhead incurred

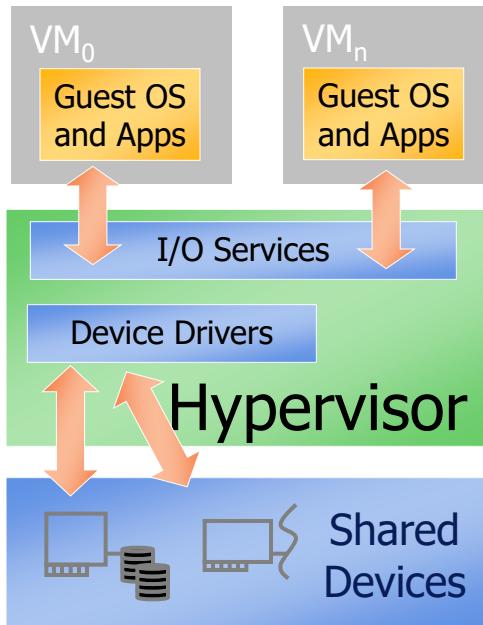
Virtualised memory (2): Second Level Address Translation (SLAT)



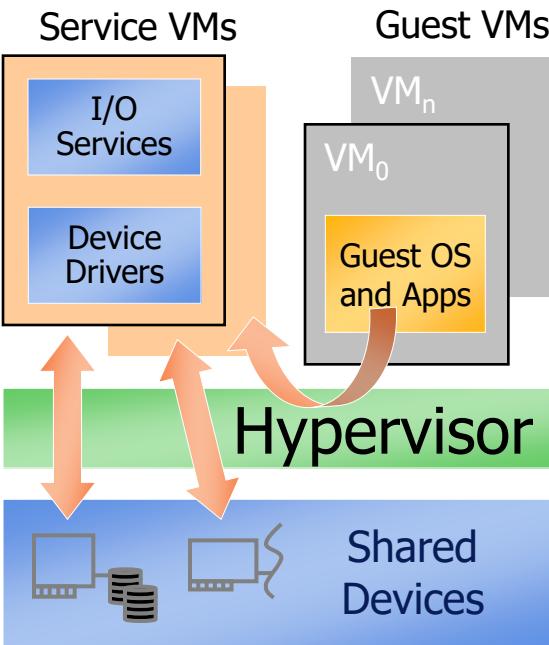
- E.g. Extended Page Table (EPT – Intel)
- A new page-table structure, under the control of the VMM
 - Defines mapping between guest- and host-physical addresses
 - EPT base pointer points to the EPT page tables
 - EPT (optionally) activated on VM entry, deactivated on VM exit
- Hardware performs end-to-end (guest virtual to physical) mapping
 - No VM exits (traps) due to guest page faults
 - Guest has full control over its own page tables

I/O Virtualization

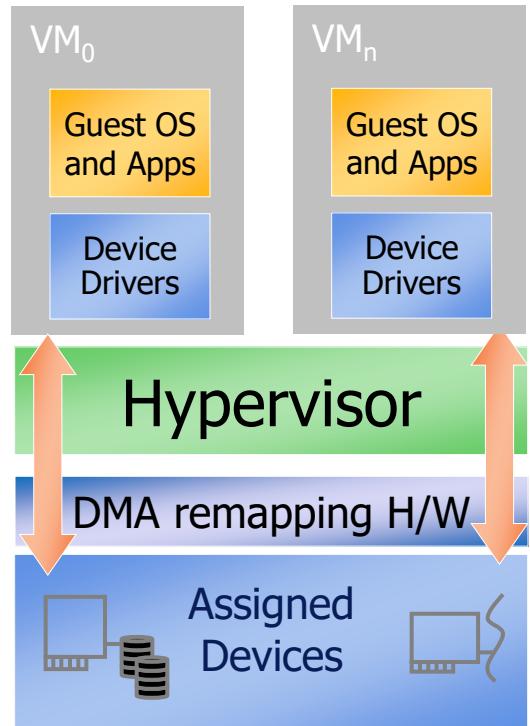
Monolithic model



Service model



Pass-through model



DMA hardware maps address

- Hypervisor programmes DMA with address *domain* conversions
- DMA remaps addresses on the fly
- E.g. Intel VT-d

Modern virtualisation benefits (1)



Secure Multiplexing

- Run multiple VMs on single physical host
- Processor hardware isolates VMs, e.g. MMU

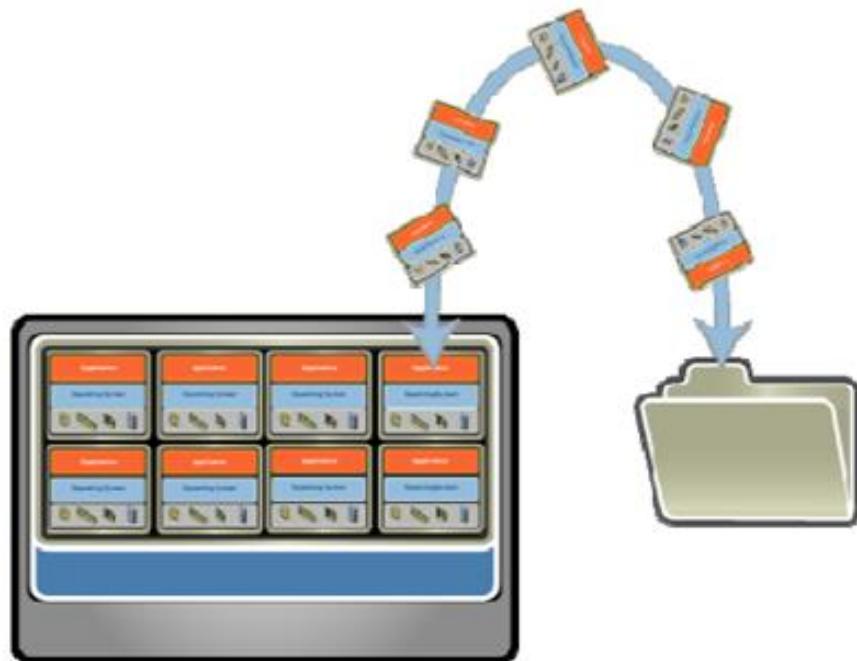
Strong Guarantees

- Software bugs, crashes, viruses within one VM cannot affect other VMs

Performance Isolation

- Partition system resources
- Example: VMware controls for reservation, limit, shares

Modern virtualisation benefits (2)



Entire VM is a File

- OS, applications, data
- Memory and device state

Snapshots and Clones

- Capture VM state on the fly and restore to point-in-time
- Rapid system provisioning, backup, remote mirroring

Easy Content Distribution

- Pre-configured apps, demos
- Virtual appliances

ENCE360

Operating

Systems



Performance optimisation

Optimising cache hits (1): Data structure reorganisation

- Maximise cache hits by keeping data accesses contiguous:

```
typedef struct{  
  
    float x,y,z;  
  
    int a,b;  
  
} Vertex;  
  
Vertex Vertices [NumVs] ;
```

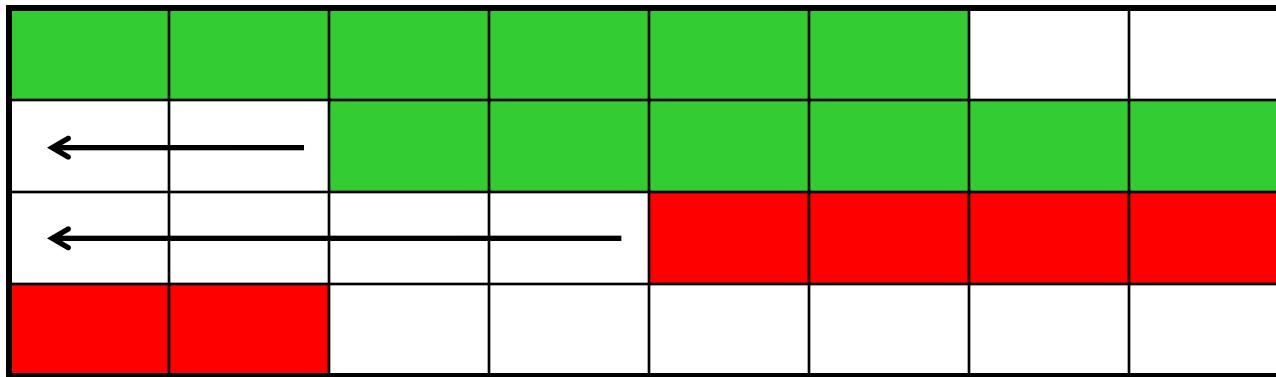


```
typedef struct{  
  
    float x [NumVs] ;  
  
    float y [NumVs] ;  
  
    float z [NumVs] ;  
  
    int a [NumVs] ;  
  
    int b [NumVs] ;  
  
} VerticesList;  
  
VerticesList Vertices;
```

All values of x now contiguous

Optimising cache hits (2): Data Alignment

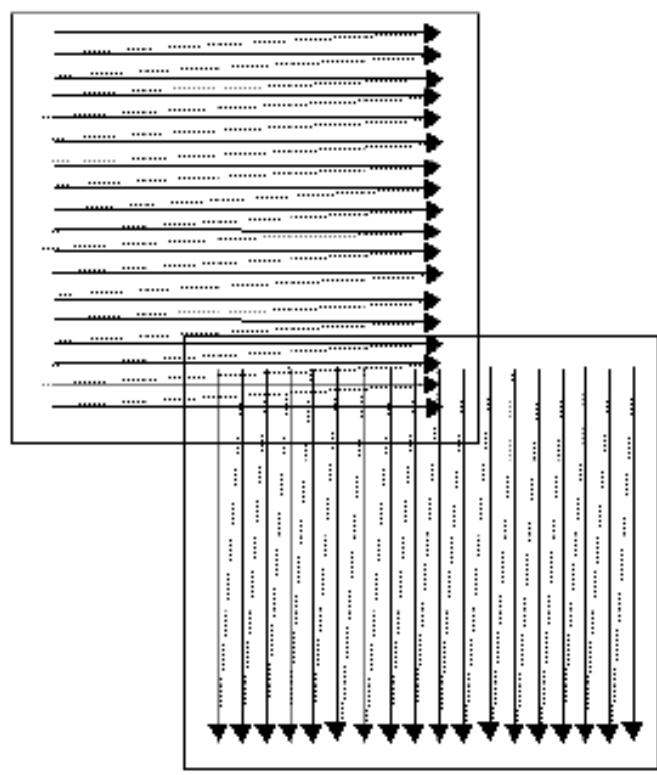
- Try to keep data within 1 line of cache:



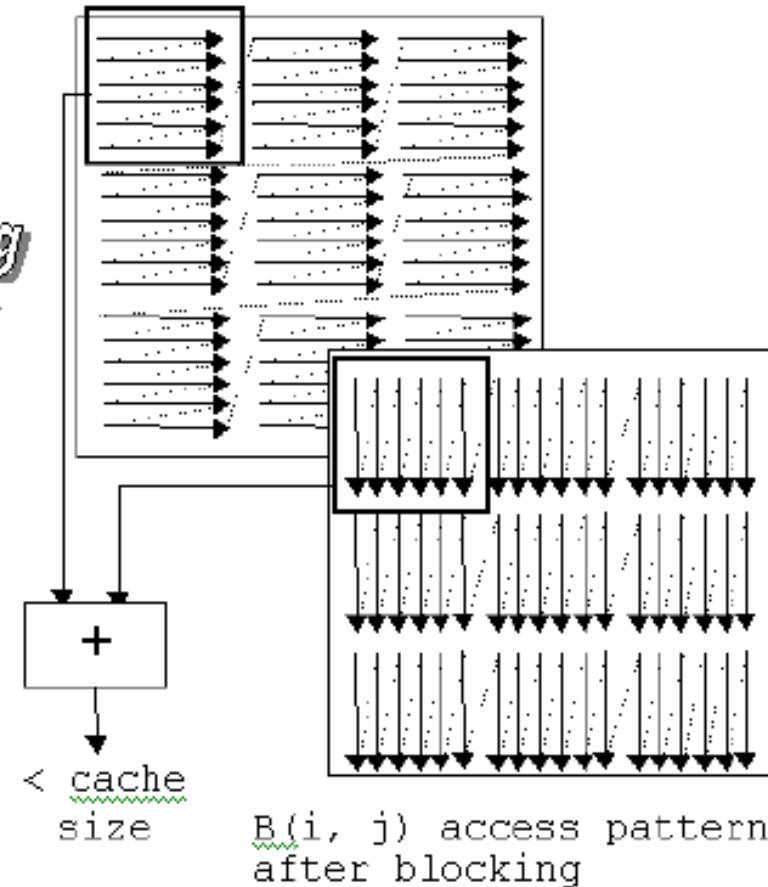
- Organise data to fit on a cache line
 - maximises cache hit rate

Optimising cache hits (3): Loop Blocking

$A(i, j)$ access pattern



$A(i, j)$ access pattern
after blocking

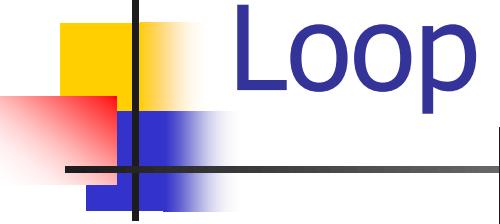


$B(i, j)$ access pattern

< cache
size

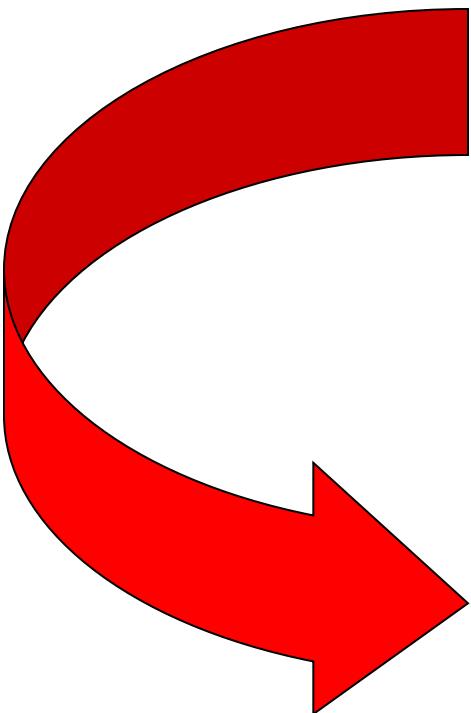
$B(i, j)$ access pattern
after blocking

Organise data access “span” to fit in cache

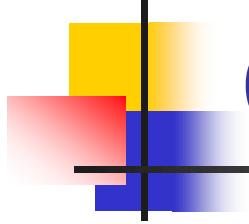


Loop Blocking (2)

```
float A[MAX, MAX], B[MAX, MAX]
...
for (i=0; i< MAX; i++) {
    for (j=0; j< MAX; j++) {
        A[i,j] = A[i,j] + B[j, i];
    }
}
```



```
float A[MAX, MAX], B[MAX, MAX];
for (i=0; i< MAX; i+=block_size) {
    for (j=0; j< MAX; j+=block_size) {
        for (ii=i; ii<i+block_size; ii++) {
            for (jj=j; jj<j+block_size; jj++) {
                A[ii,jj] = A[ii,jj] + B[jj, ii];
            }
        }
    }
}
```

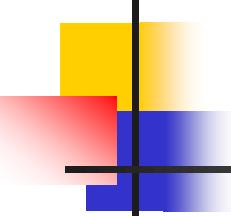


Code reordering: strip mining

- Combine loops and iterate over cache size:

```
typedef struct _VERTEX {
float x, y, z, nx, ny, nz, u, v;
} Vertex_rec;

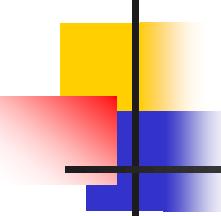
main() {
    Vertex_rec v[Num];
    for (i=0; i<Num; i++) {
        transform(v[i]);
    }
    for (i=0; i<Num; i++) {
        lighting(v[i]);
    }
}
```



Strip mining 2

- Combine loops and iterate over cache size:

```
main()
{
    Vertex_rec v[Num];
    for (i=0; i < Num; i+=strip_size) {
        for (j=i; j < min(Num, i+strip_size); j++) {
            transform(v[j]);
        }
        for (j=i; j < min(Num, i+strip_size); j++) {
            lighting(v[j]);
        }
    }
}
```

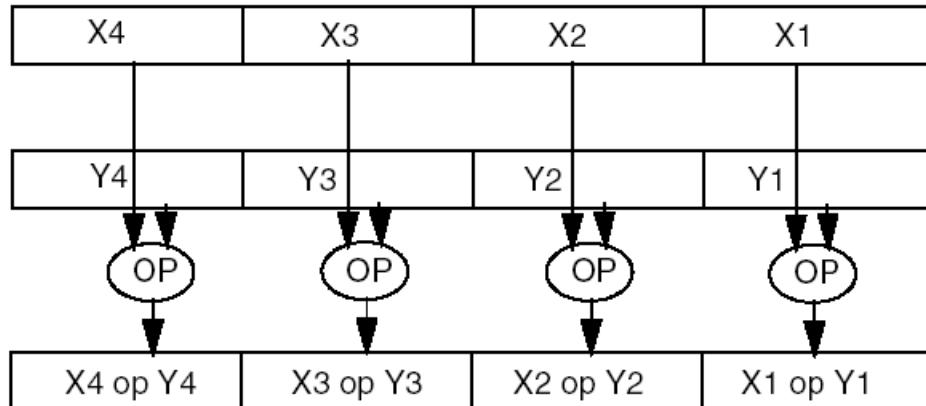


Loop unrolling

- Unroll loops until branching is less than 10% of execution time
- Unroll hot (inner) loops to 16 or less iterations
- Reduces the number of conditional branch/jump instructions
- Maximises the use of Branch Target Predictor (BTB)

Normal loop	After loop unrolling
<pre>int x; for (x = 0; x < 100; x++) { delete(x); }</pre>	<pre>int x; for (x = 0; x < 100; x += 5) { delete(x); delete(x + 1); delete(x + 2); delete(x + 3); delete(x + 4); }</pre>

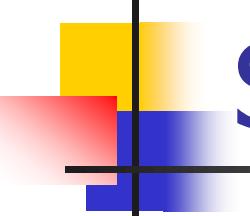
Vectorisation: SIMD



E.g. SSE 128 bit registers:

- Two 64-bit floats OR
- Two 64 bit ints OR
- Eight 16 bit ints OR
- Sixteen bytes

- SIMD = Single Instruction Multiple Data:
 - Added special registers that hold multiple data items in one register (AVX512: 512 bit registers)
 - Added instructions carry out a single operation in parallel
 - E.g. Intel MMX, SSE, AVX/AVX2/AVX512



SIMD example

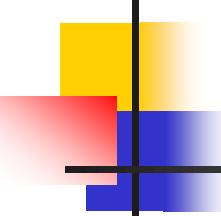
```
vec_res.x = v1.x + v2.x;  
vec_res.y = v1.y + v2.y;  
vec_res.z = v1.z + v2.z;  
vec_res.w = v1.w + v2.w;
```

Four sequential instructions

```
movaps xmm0, [v1] ;xmm0 = v1.w | v1.z | v1.y | v1.x  
addps xmm0, [v2] ;xmm0 = v1.w+v2.w | v1.z+v2.z | v1.y+v2.y | v1.x+v2.x  
movaps [vec_res], xmm0 ;xmm0
```

Parallel execution

- Vector of four values moved into SIMD register
- All four values added simultaneously
- Result moved to memory in one instruction



SIMD: how?

- Automatic vectorisation (e.g. Intel C++ compiler):
 - Build statement dependency graph
 - Cluster into strongly connected components (SCCs)
 - Split SCCs (repeat loops if necessary) and vectorise where possible
- Compiler hints **#pragma vector always**
- Write your own:
 - Intrinsics

```
__m512 __mm512_add_ps  (__m512 a, __m512 b);           // calculates a + b for two vectors of 16 floats  
__m512 __mm512_fmadd_ps(__m512 a, __m512 b, __m512 c); // calculates a*b + c for three vectors of 16 floats
```

- Assembler

```
movaps xmm0, [v1] ;xmm0 = v1.w | v1.z | v1.y | v1.x  
addps xmm0, [v2] ;xmm0 = v1.w+v2.w | v1.z+v2.z | v1.y+v2.y | v1.x+v2.x  
movaps [vec_res], xmm0 ;xmm0
```

Vectorisation

```
for (i = 0; i < 1024; i++)
    c[i] = a[i] * b[i];
```

Strip mine

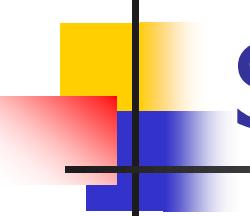
```
for (i = 0; i < 1024; i += 4)
    for (j = 0; j < 4; j++)
        c[i+j] = a[i+j] * b[i+j];
```

Loop distribution

```
for (i = 0; i < 1024; i += 4)
{
    for (j = 0; j < 4; j++) tA[j] = A[i+j];
    for (j = 0; j < 4; j++) tB[j] = B[i+j];
    for (j = 0; j < 4; j++) tC[j] = tA[j] * tB[j];
    for (j = 0; j < 4; j++) C[i+j] = tC[j];
}

for (i = 0; i < 1024; i += 4)
{
    vA = vec_ld(&A[i]);
    vB = vec_ld(&B[i]);
    vC = vec_mul(vA, vB);
    vec_st(vC, &C[i]);
}
```

Vector codes



SIMD gotchas

- Code must be loop-independent, straight-line code

```
for (i = 1; i < end; i++)  
    f[i] = f[i-1] + b[i-1];
```

Loop dependence

```
for (i = 0; i < end; i++)  
    c[idxC[i]] = a[i] + b[i];
```

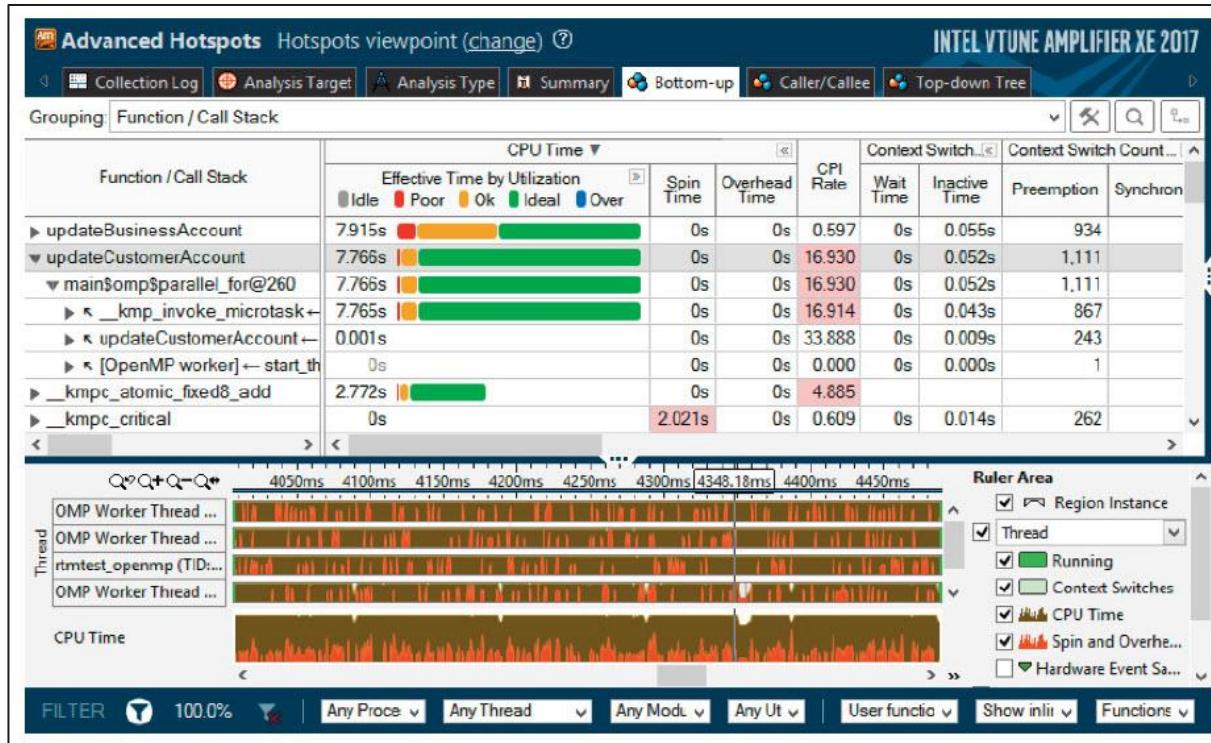
Loop *index* dependence

```
for (i = 0; i < CalcEnd(); i++)  
{  
    if (DoJump())  
        i += CalcJump();  
    c[i] = a[i] + b[i];  
}
```

Non straight-line code

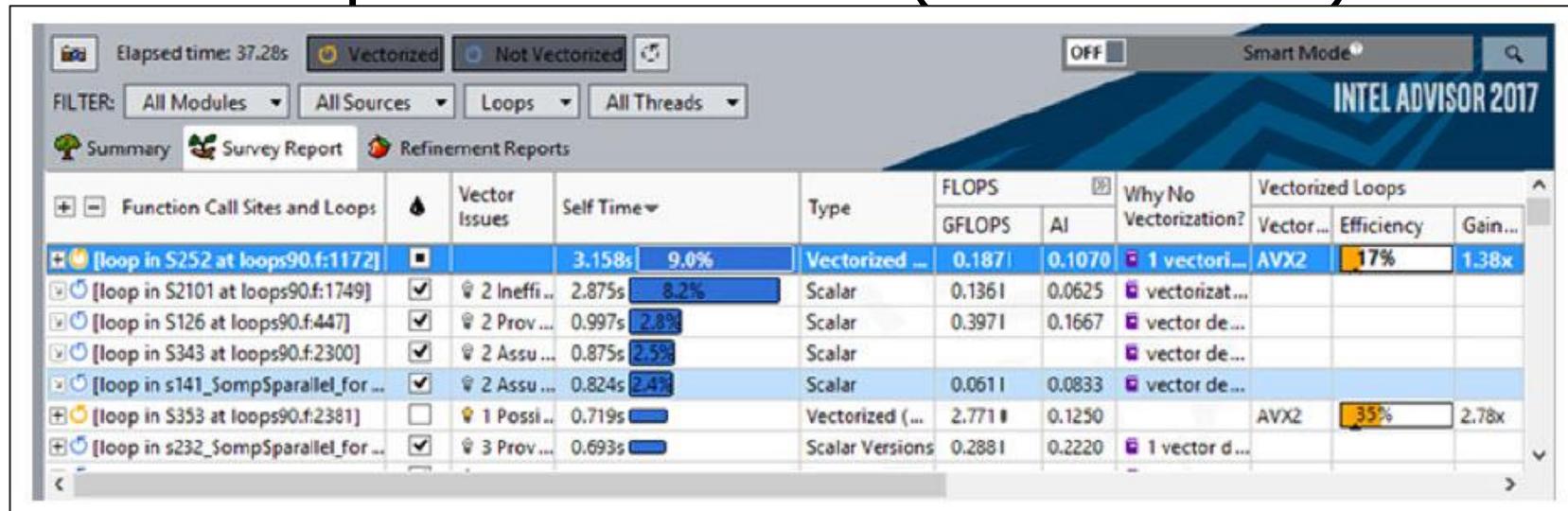
Optimisation approach (Intel)

1. Measure baseline performance
2. Determine hotspots (e.g. Vtune)

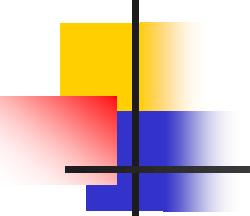


Optimisation approach (2)

3. Determine candidate (hot) loops
4. Measure potential benefits (Intel Advisor)



5. Implement recommendations
 - Code reorganisation, vectorisation hints, threading



Intel Integrated Performance Primitives (IPP)

- Optimised function libraries (including SSE) for:

Vector/Matrix Mathematics	Video, Audio Decode/Encode
Signal Processing	Cryptography
Computer Vision	Ray Tracing/Rendering
Speech Recognition	String Processing
Data Compression	

```
pBuffer = ippsMalloc_8u(bufSize);
```

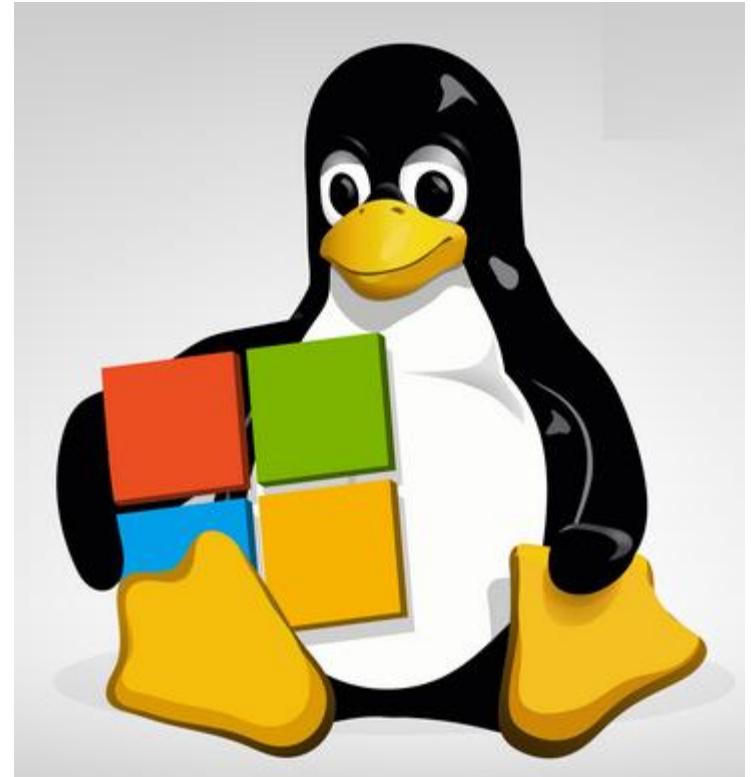
Example: blur an image

```
/* Filter the image */
if (status >= ippStsNoErr) status =ippiFilterBoxBorder_8u_C3R(pSrc,
srcSize, maskSize, ippBorderRep1, NULL, pBuffer);
```

ENCE360

Operating

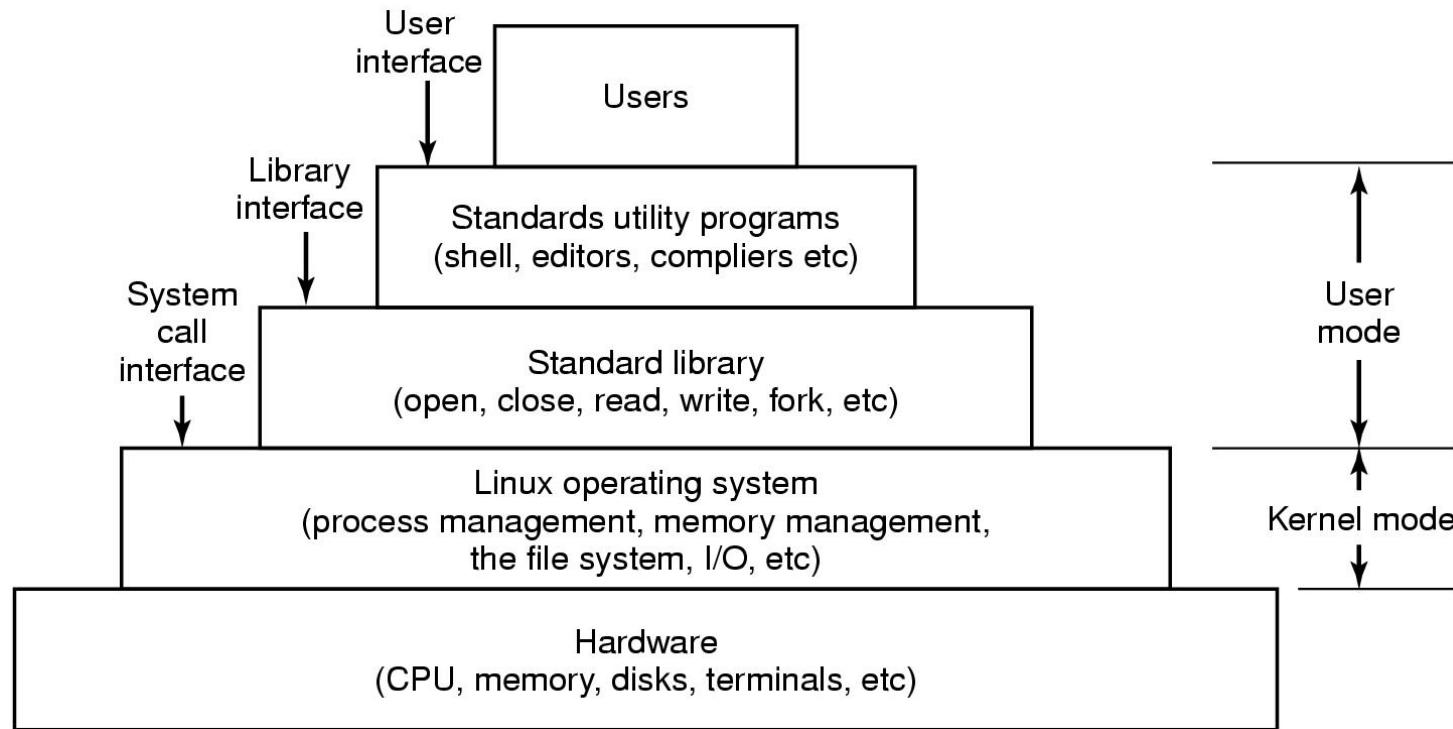
Systems



Linux v Windows

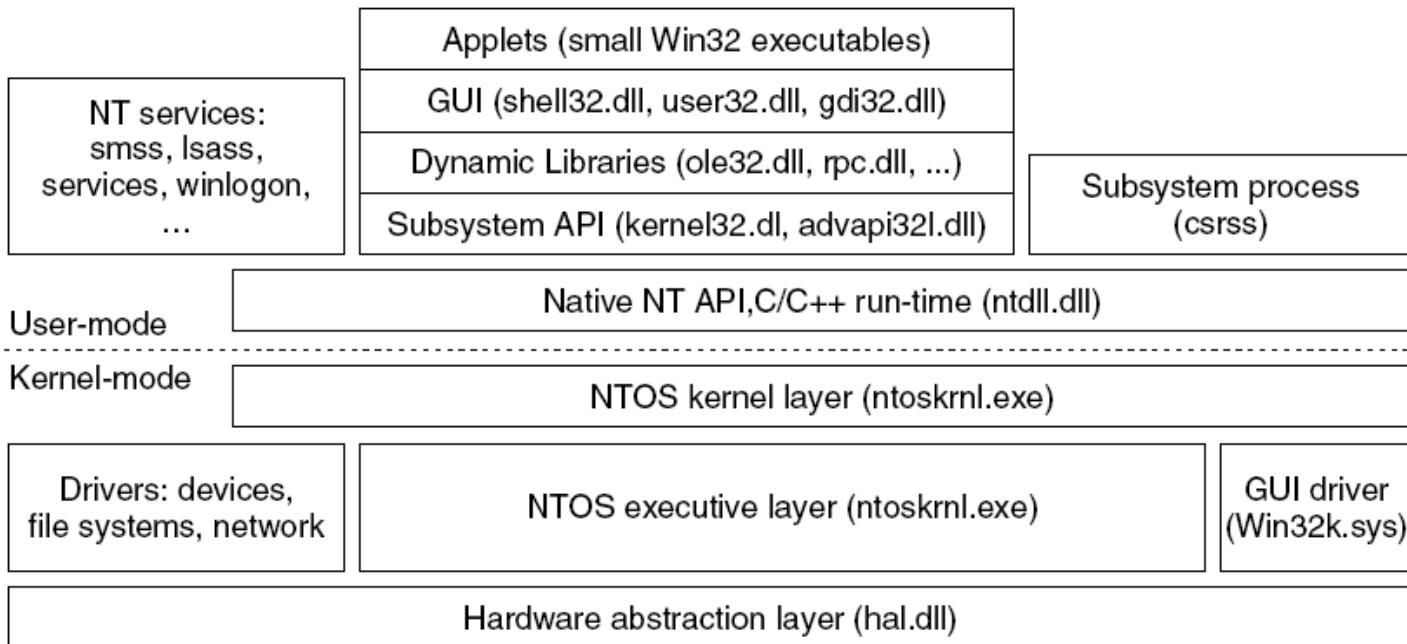
MOS (3rd ed.) CH 10,11

Philosophy: Linux



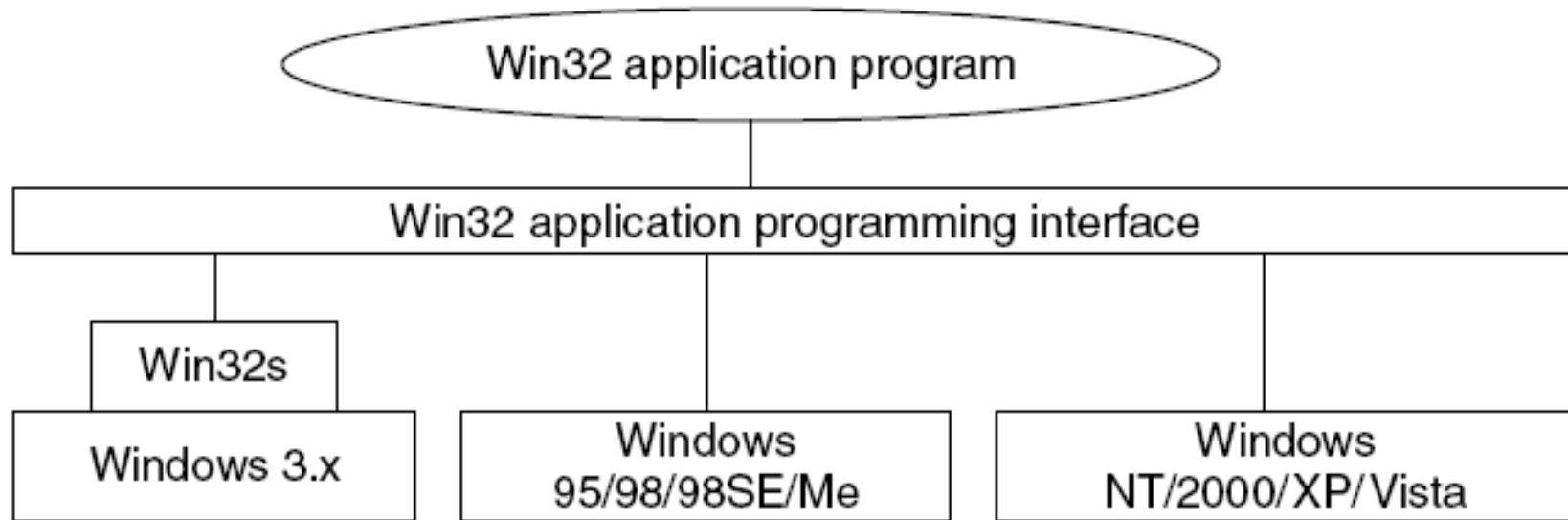
- Monolithic
- Clear separation between kernel and user space
 - Standard library provides trap to kernel
- Windowing via user-mode X system

Philosophy: Windows



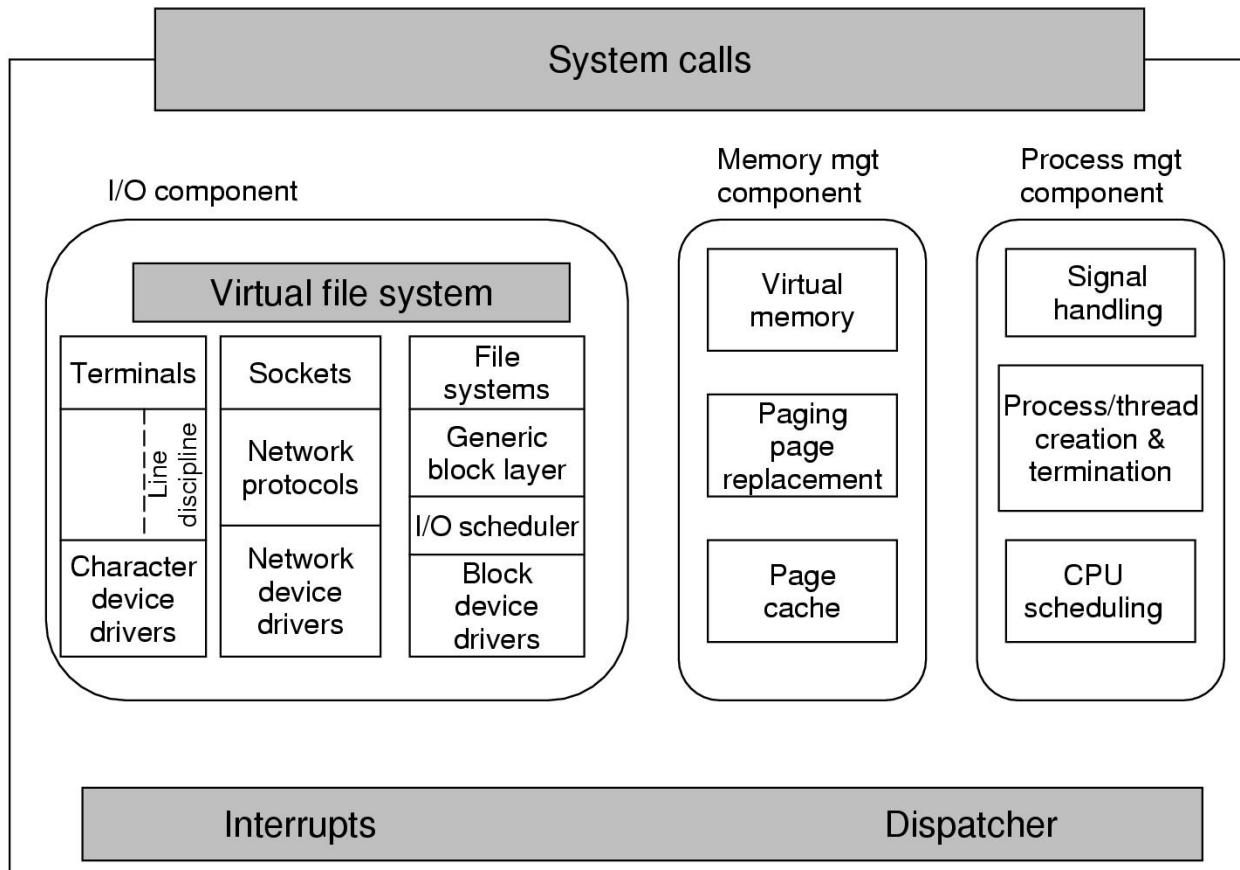
- Higher functionality as part of OS
- Monolithic, layered design
 - Dynamic libraries access subsystems via APIs
 - Native API wraps system calls
- Windowing subsystem in kernel

Windows portability



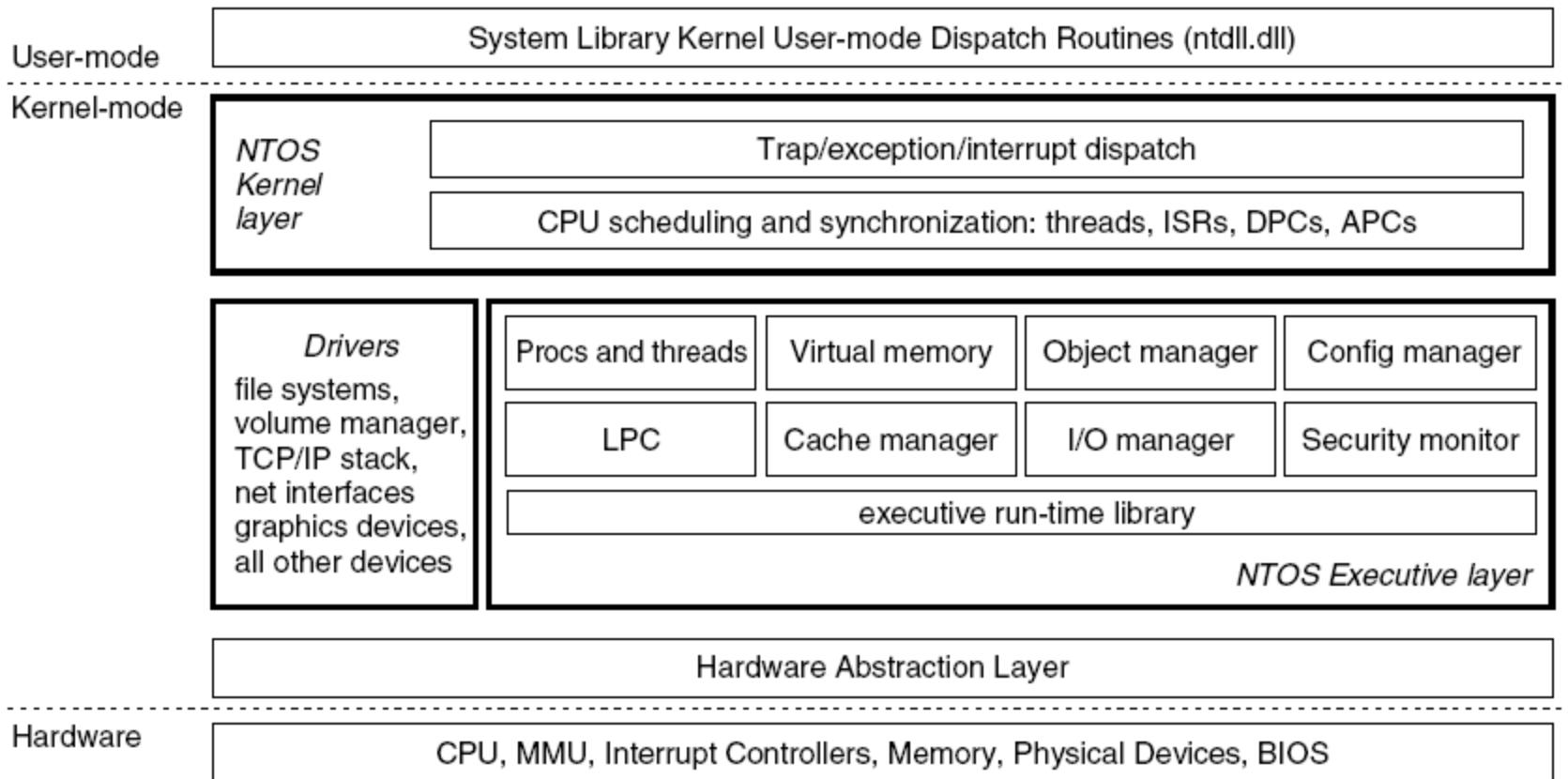
- Win32 API provides backward compatibility
 - Originally one of several (IBM OS/2, POSIX)
- Hardware abstraction layer

Kernel structure: Linux



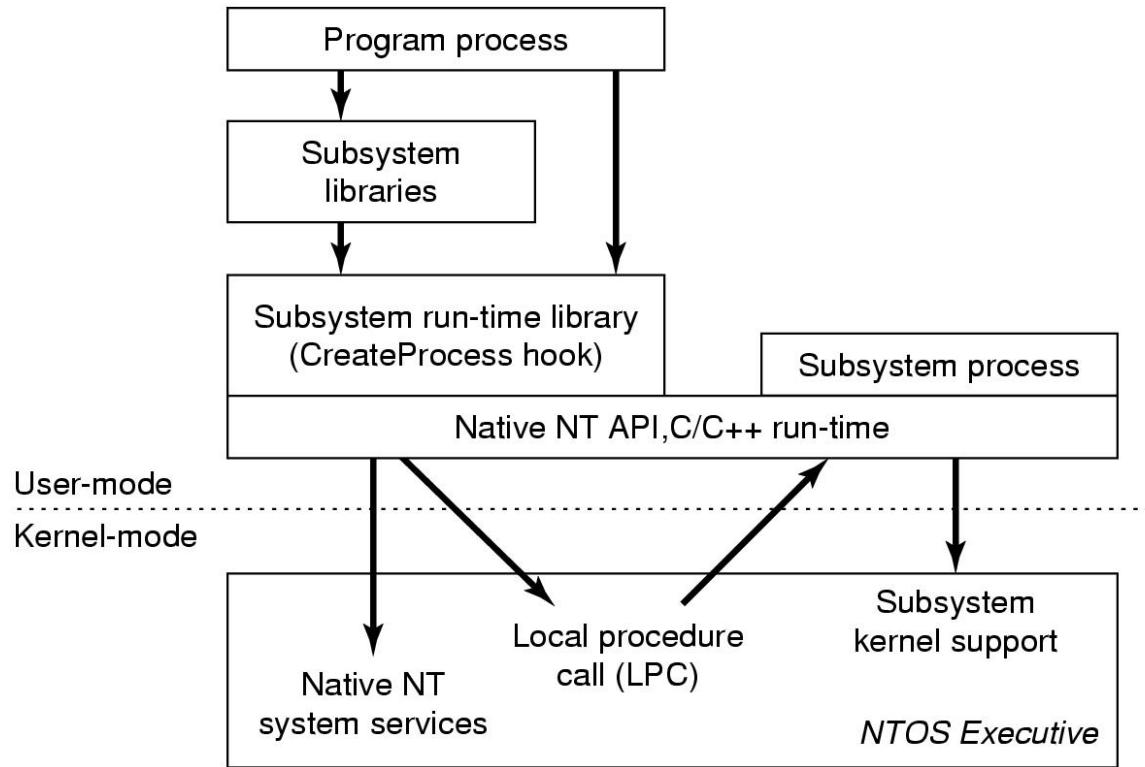
- Monolithic kernel sits on hardware
- Divided into *components*
- Generic abstract routines for common functions
(e.g. I/O block handler)

Kernel structure: Windows



- NTOS kernel layer: handle user system calls
- Hardware abstraction layer: hide hardware specifics

Windows subsystems

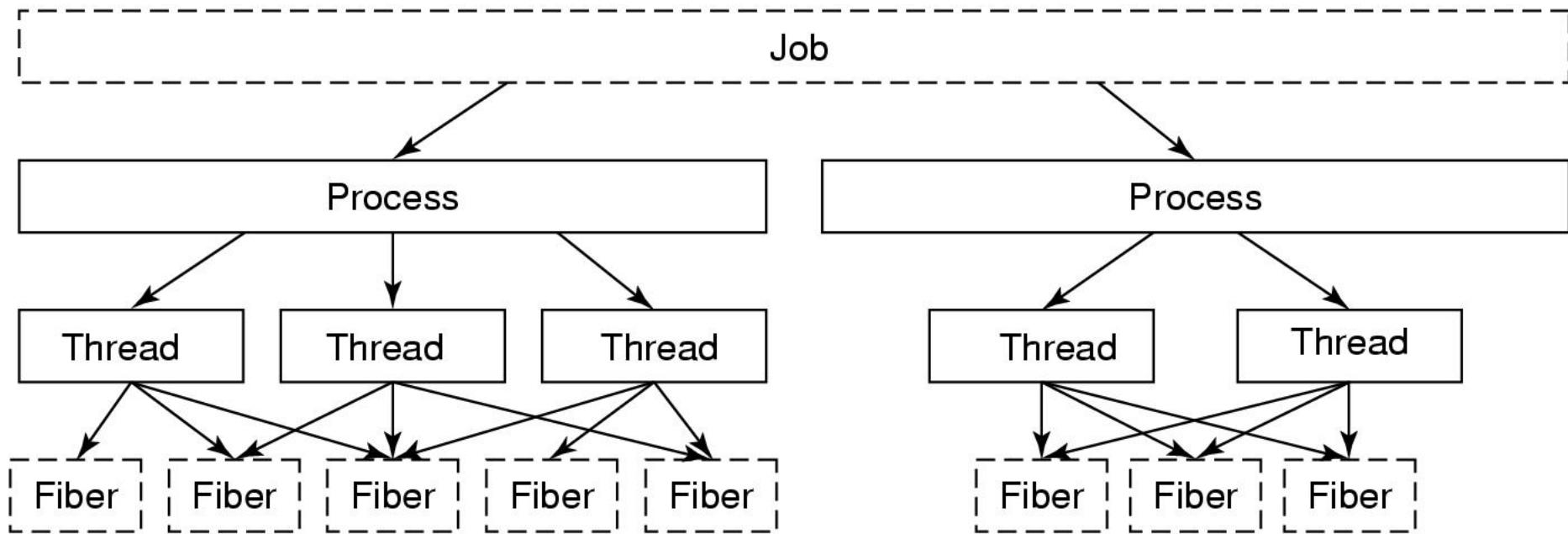


- Functionality provided by subsystem processes
 - Started on loading the program (if needed)
- Communicate via Local Procedure Calls (LPC)

Processes and threads: Linux

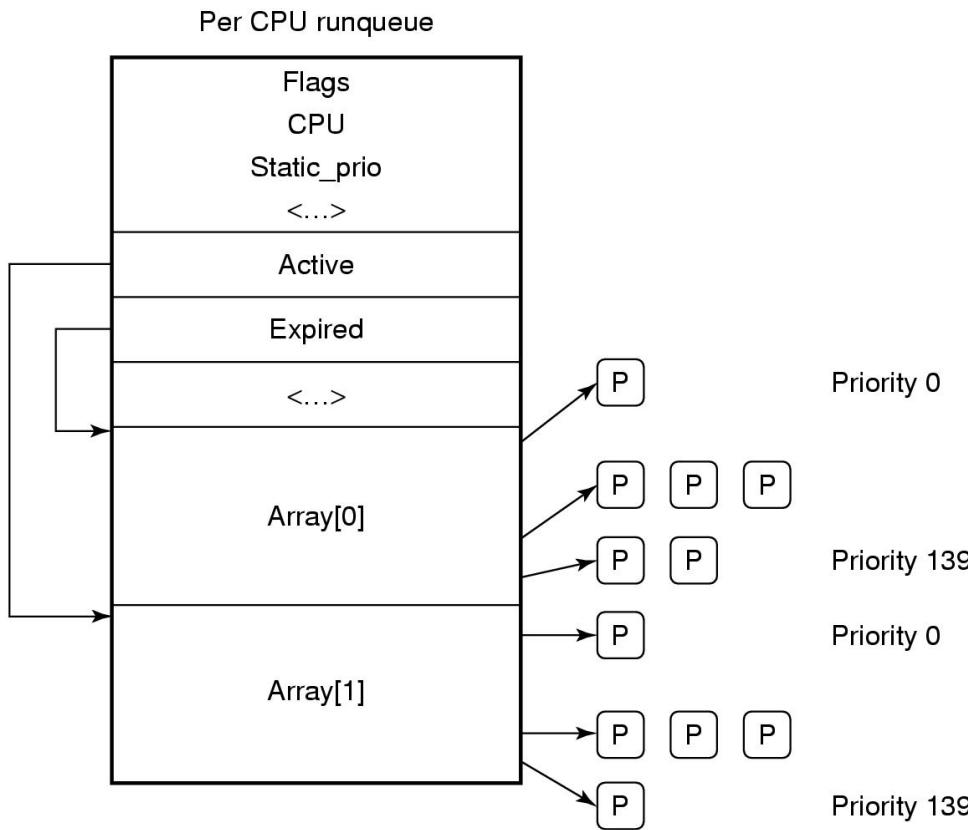
- Task: internal representation of a runnable entity
- Process: created by forking from another process
 - Forked process contains copy of parent address space
 - Memory **copy on write**: pages shared until they differ
- Thread (kernel): two or more processes that share resources
- Clone mechanism allows control over sharing via *sharing_flags*:
 - VM: same address space (thread) vs own (process)
 - FS: share (or not) file system (umask, root, working dirs)
 - FILES: share vs copy file descriptors
 - SIGHAND: share vs copy signal table
 - PID: old vs new process id
 - PARENT: same (sibling) vs child
- Pthreads: user threads library (like Windows fibers)

Processes and threads: Windows



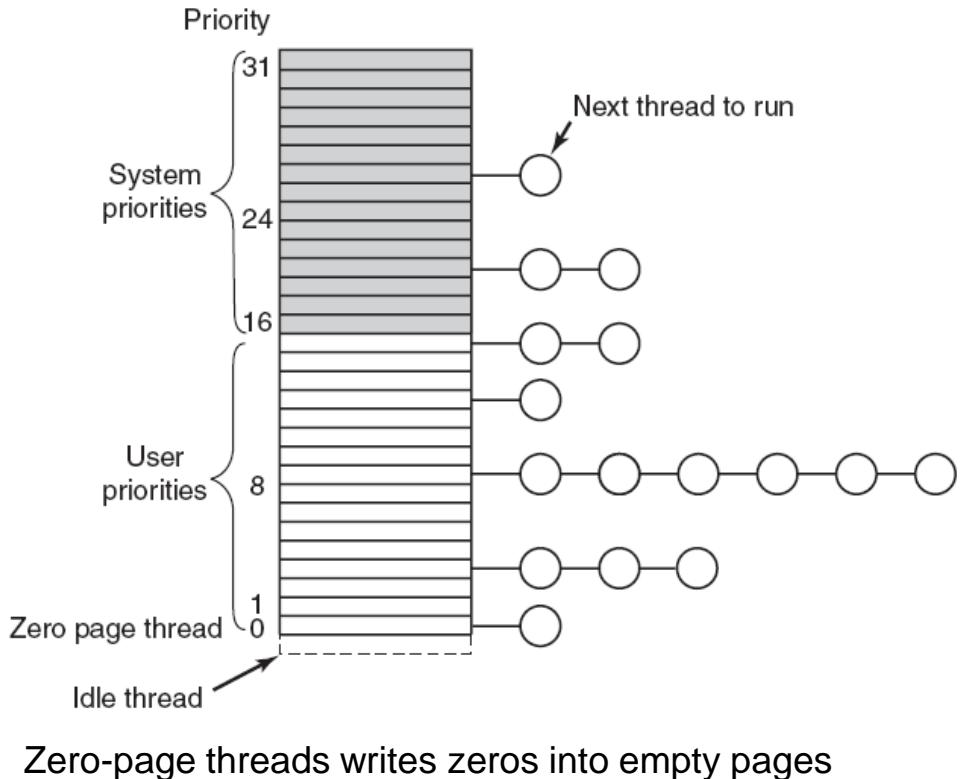
- Job: arbitrary collection of processes (parent, rarely used)
- Process: resource collection; has no parent process
- **Thread: schedulable (runnable) object**
- Fiber: user-mode thread (many-to-many; rarely used)

Scheduling: Linux



- Thread-based
- Three priority classes:
 - Real-time FIFO (highest)
 - Real-time round-robin
 - Timesharing (lowest)
- Active/expired:
 - Tasks initially active
 - Move to expired when time expires
 - Swap pointers when all expired
 - Avoids starvation
- Priority-based quanta
- Dynamic bonus:
 - Penalised on wakeup
 - Pre-empted/expired rewarded

Scheduling: Windows

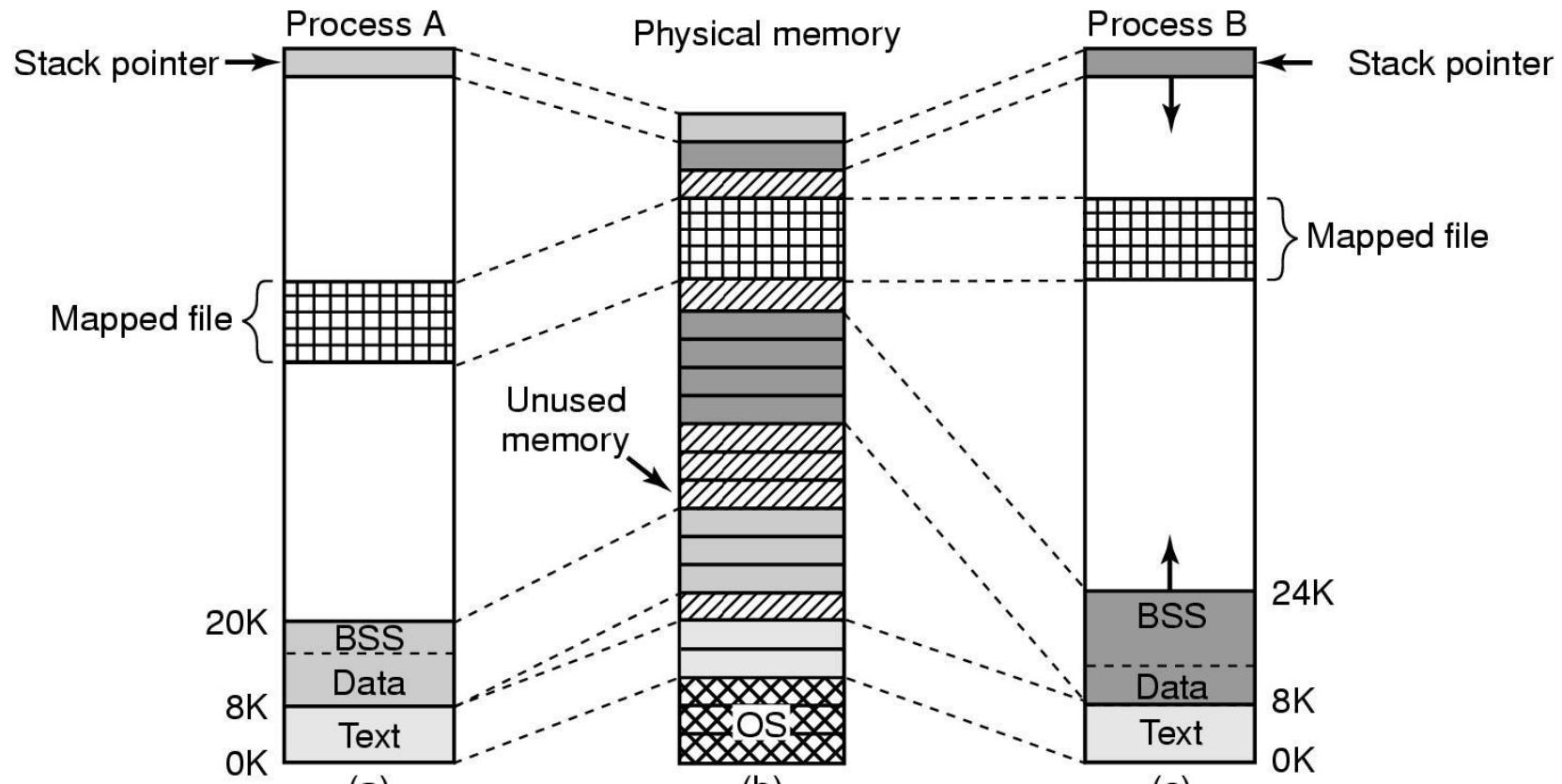


- Threads individually scheduled
 - Round-robin by priority
 - Thread runs scheduling code
 - *Ideal processor* maintained
- Dynamic:
 - Boost I/O completion
 - Boost on event
 - Degrade on quantum completion
 - Boost “old” waiting threads to 15 for 2 quanta (inversion prevention)
- Fixed quantum:
 - Client: 20ms default
 - Server: 180ms default

Lightweight synchronisation

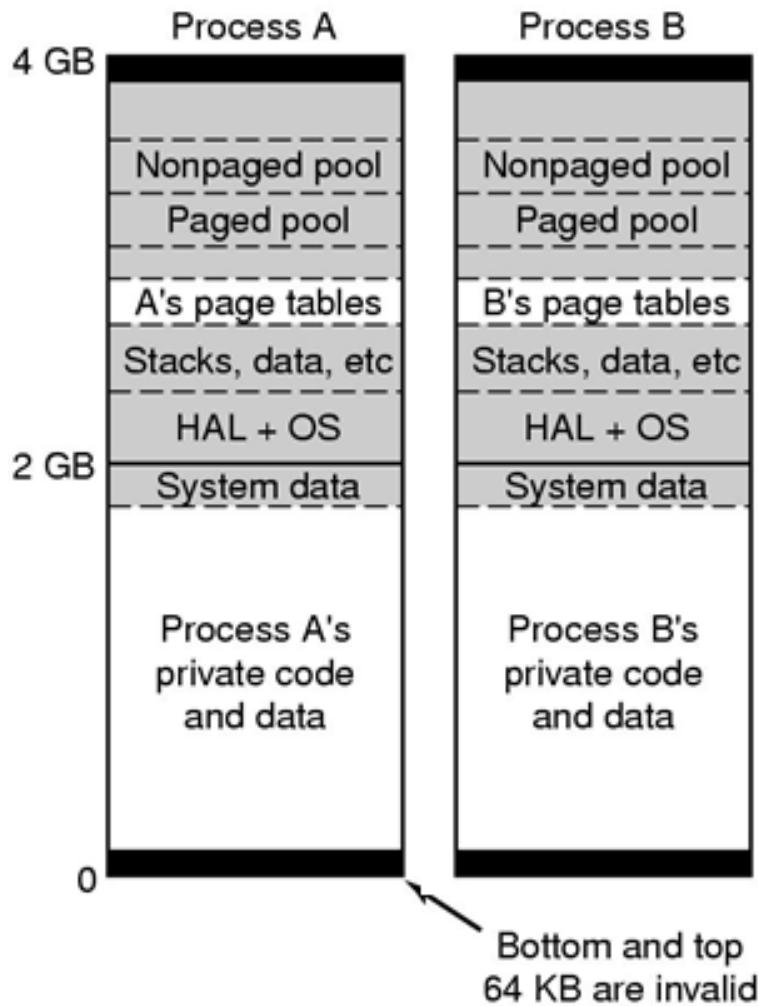
- Linux: *futexes* (fast user-space mutex)
 - Futex_var in user space
 - Wait queue in kernel space
 - Processes can test and set without kernel trap unless contention:
 - Place process on wait queue
 - Wake waiting processes
- Windows: *critical sections*
 - Implemented in process address space
 - Optimise use of spin locks + kernel calls for multiprocessors

Virtual memory: Linux



- Text, data and stack segments
- Shared text segments
- Shared memory-mapped files

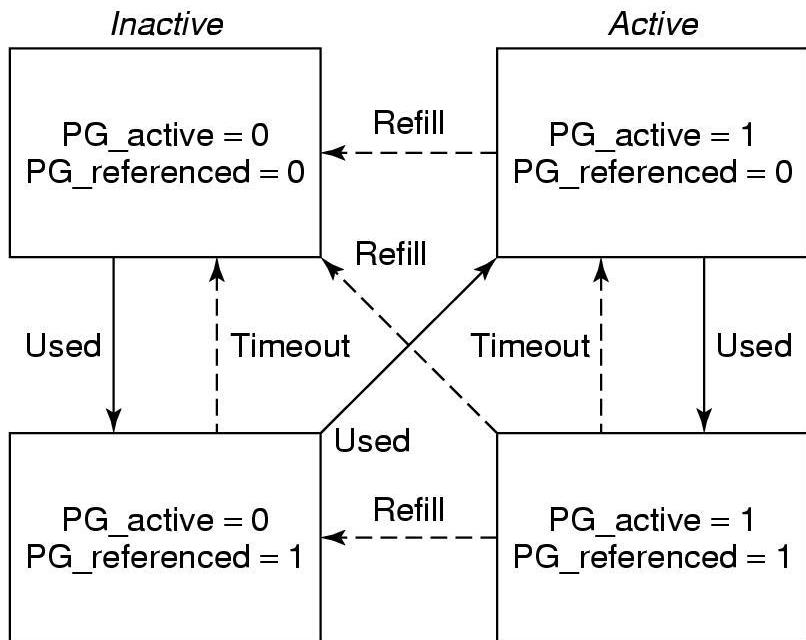
Virtual memory: Windows



- Kernel address space shared across processes
 - Avoids memory map switch
- Top and bottom invalid
 - Traps bad pointers (e.g. 0, -1)
- Three page states:
 - Invalid (unused)
 - Committed: in use
 - Reserved: can't be claimed for other purposes

Page replacement: Linux

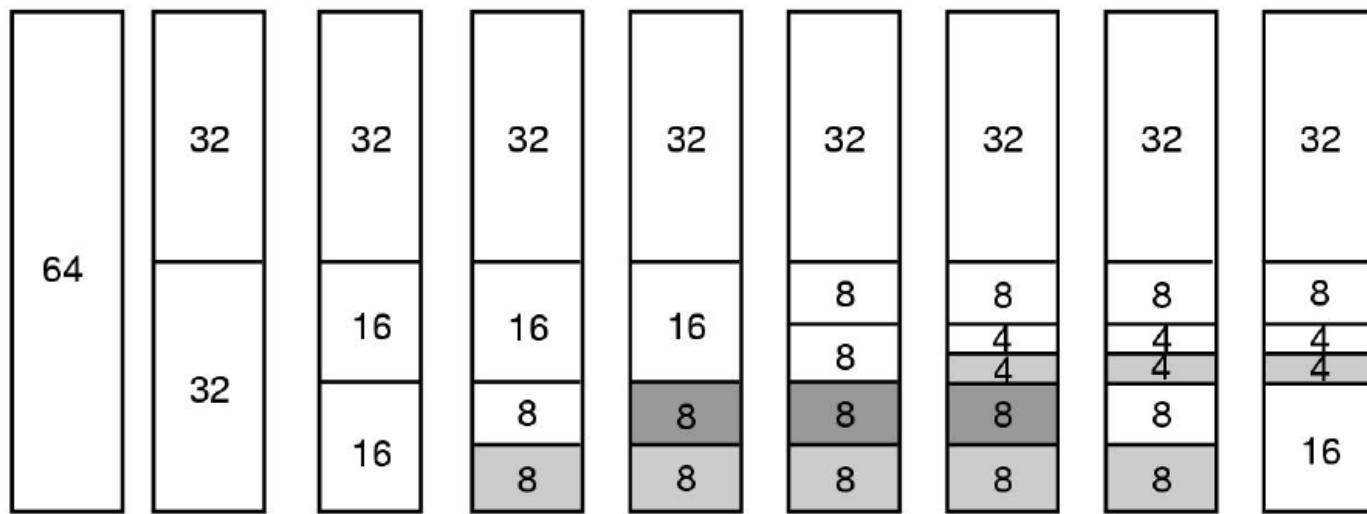
- Page Frame Reclamation Algorithm (PFRA)
 - Daemon maintains free page list
- Page types:
 - Unreclaimable: can't discard
 - Swappable: write back
 - Syncable: write if dirty
 - Discardable: reclaim
- Clock algorithm:
 - First pass: reset *referenced*
 - Second pass: update state based on *referenced*
 - If run out of inactive: refill from active pages
 - Add to free list from (inactive, not referenced)
- *Pdflush*: write back very old dirty pages (cleaner)



Page replacement: Windows

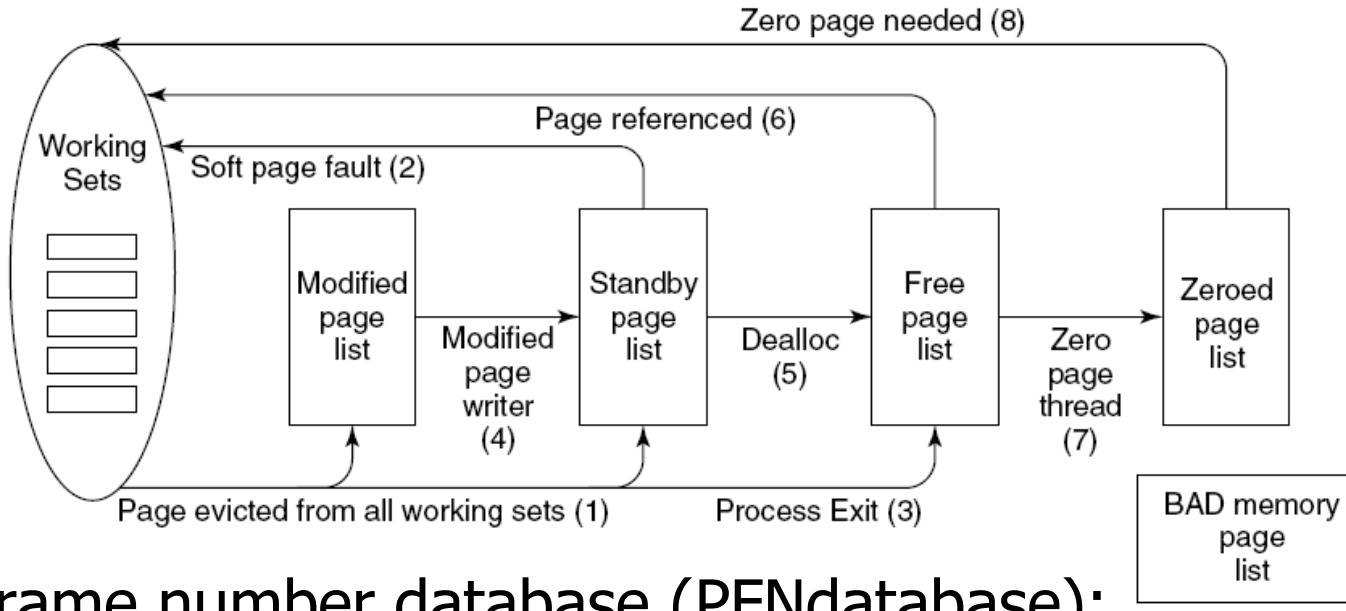
- Fundamental unit: **process working set**
 - Minimum and maximum (soft) limits
 - Memory normally allowed to grow unbounded
 - Clock algorithm used to update working sets (bounded)
- Working set manager runs every second:
 - **Lots of memory:** scan and update pages
 - Reset access bits and compute age
 - Estimate number of unused (recently) pages
 - **Memory getting low:** stop growing working set
 - Identify processes with significant unused pages
 - Replace oldest pages when needed
 - **Memory running out:** trim working sets
 - Remove oldest pages

Physical memory allocation: Linux



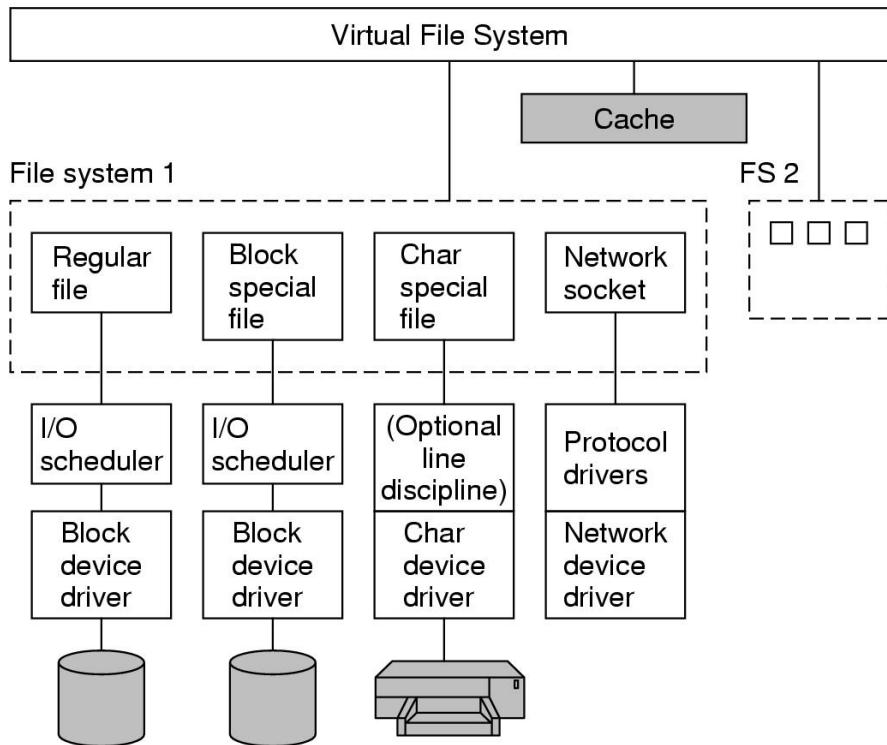
- Buddy algorithm allocates *contiguous* physical memory
 - Divide by 2 until minimum required size reached
 - Merge adjacent halves to regain larger spaces
 - Maintain lists of block addresses by size
- Slab algorithm manages partial chunks
 - 65 pages = 128 page chunk, in two slabs (65 pages, used, 63 empty)
- Object caches track kernel object slabs
- Vmalloc allocates non-contiguous memory (large units only)

Physical memory management: Windows



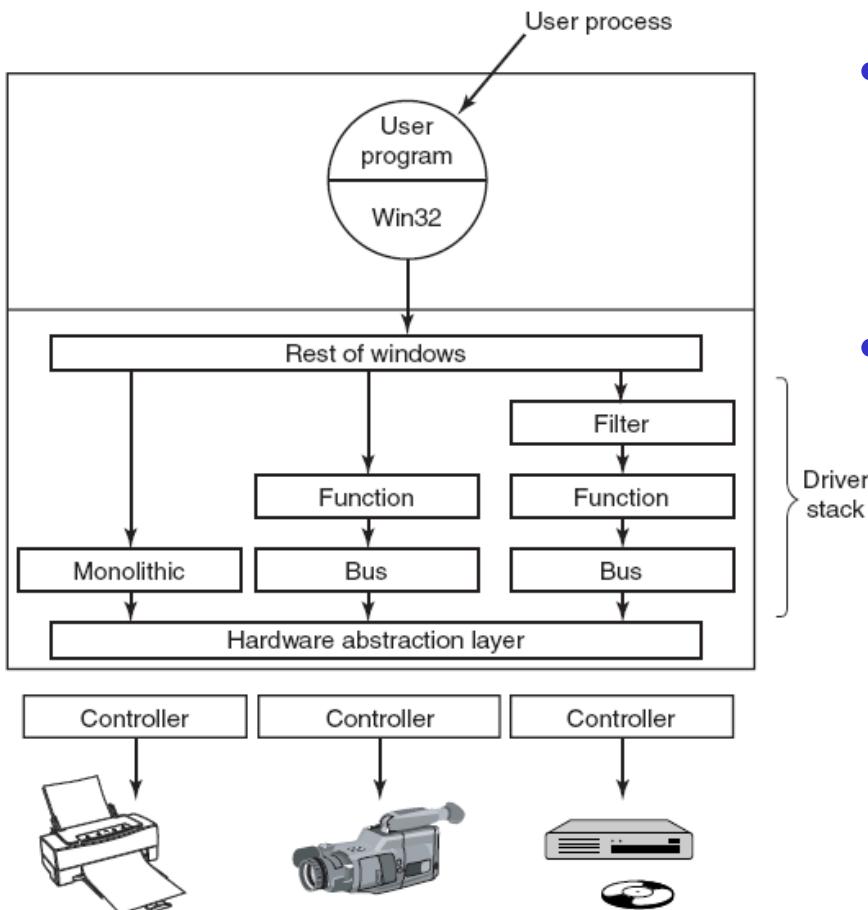
- Page frame number database (PFNdatabase): non-WS pages available for re-use
 - Modified pages: waiting to be written out
 - Standby pages: mapped but not used recently
 - Free pages: no longer in use
 - Zeroed pages: wiped pages (e.g. for stack)
- Linux equivalent: *page cache*
- Extensive prefetching (standby): read-ahead and *superfetch*

I/O processing: Linux



- Drivers split in half:
 - Top half runs in caller context (=> Linux)
 - Bottom half runs in kernel context (=> device)
- Cache holds 1000's of blocks
 - Write to disk when cache too large
 - Flush every 30 seconds
- Scheduling: elevator algorithm
 - Contiguous operations merged
 - Deadlines of 0.5/5s (R/W)
- Drivers can be loaded at runtime (loadable modules)

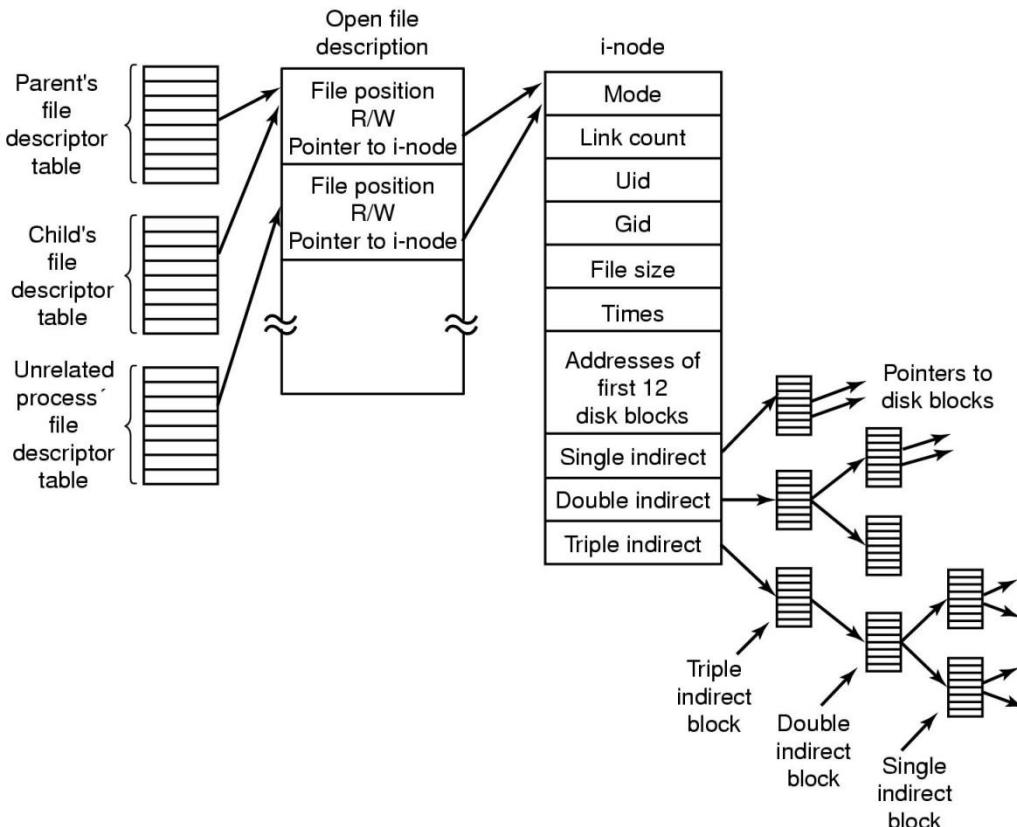
I/O processing: Windows



- Stackable, abstract drivers
 - vs Linux: monolithic
- Deferred, asynchronous processing:
 - ISR responds to interrupt, queues lower-priority deferred procedure call (DPC) – current context
 - DPC queues an asynchronous procedure call (APC) - initiator's context

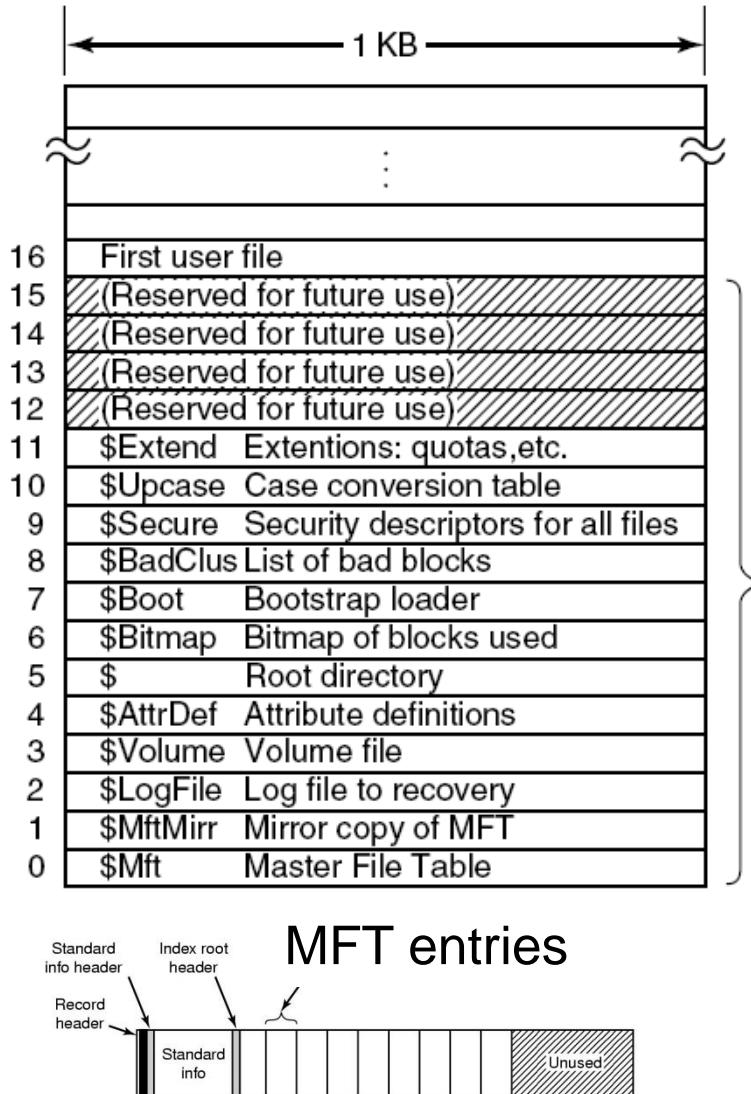
(Linux 2.6 introduced softirq and tasklets – similar to DPC)

File system: Linux

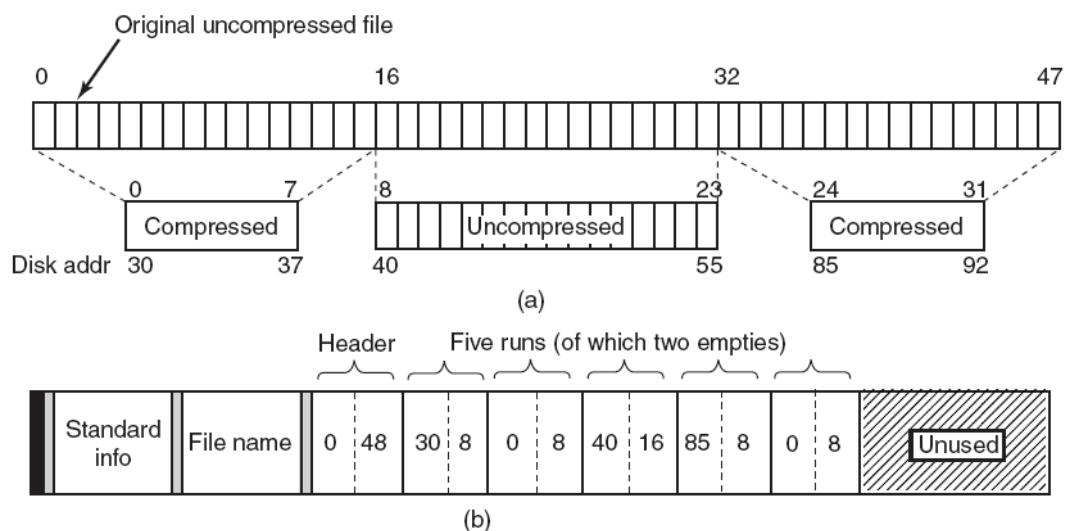


- Hierarchical i-node table
- Open file descriptor table: allows processes to share file descriptor (e.g. posn)
- Journaling:
 - Describes all writes
 - Circular buffer
 - Hierarchical:
 - Log records
 - Atomic operations
 - Transactions
 - Log flushed by transaction
 - Metadata versus all
- *Virtual* file system
 - File system-independent abstraction

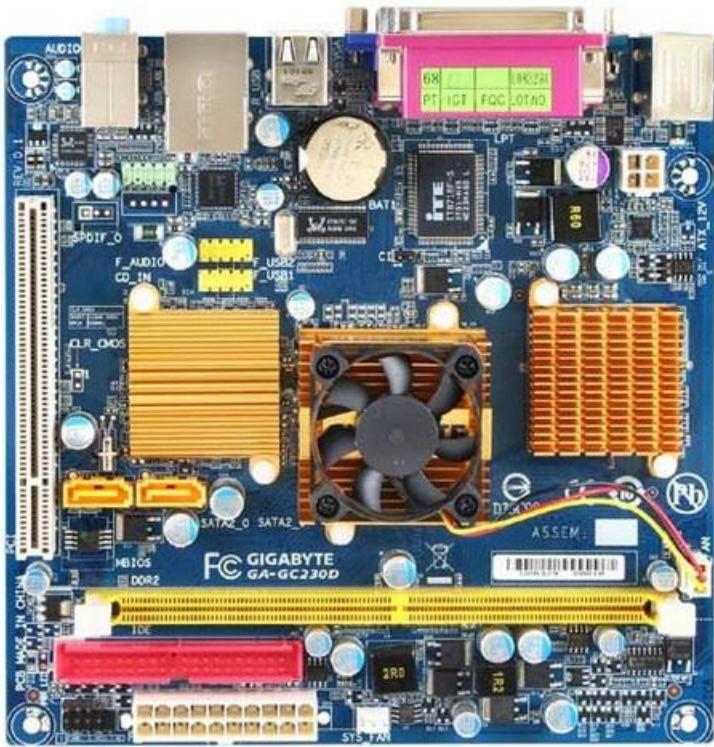
File system: Windows NTFS



- Linear Master File Table
 - Fixed 1KB + overflow records
- Multi-stream files
- B+ tree index for large directories
- File compression
 - 16 block runs for fast seeking



System on a chip (SoC) vs CPU the battle for the future of computing



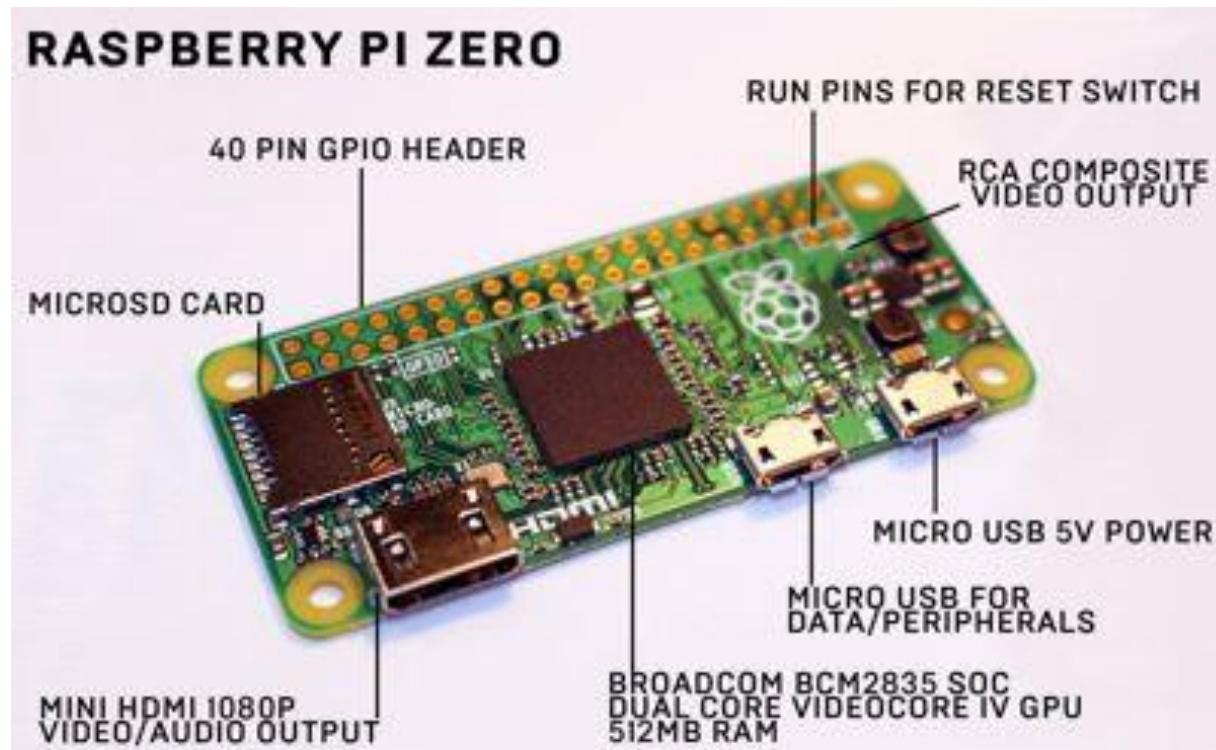
CPU computer



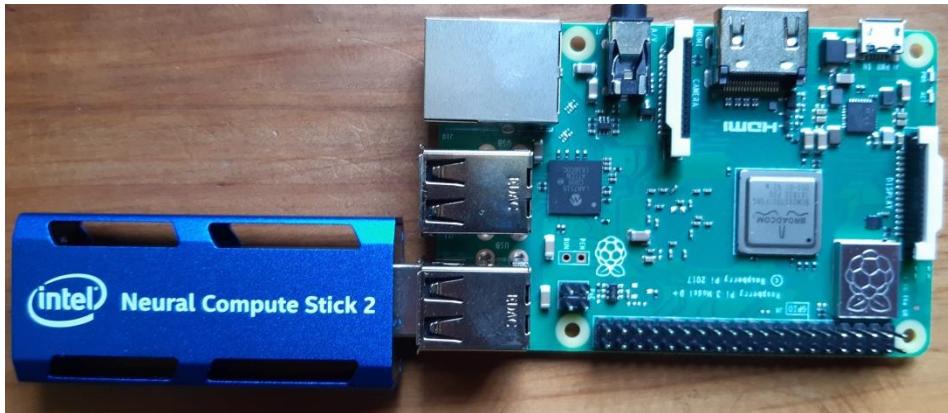
SoC computer (iPad)

SoC: \$5 Raspberry Pi Zero

- **1GHz** ARM11 core, GPU
 - 512MB RAM
 - 64GB micro card



Computer vision in the field



Raspberry Pi and
Neural Compute Stick:
computer vision in
the field

| **Automatic Classifier**
Artificial Intelligence classifying animals

Blurring the lines

Microsoft Windows 11

- announced June 2021
- built for desktop and mobile devices - easier for developers to create native applications that run across devices

Apple macOS 12 Monterey

- announced June 2021 – introduces Shortcuts, Universal Control (single keyboard and mouse can interact with multiple Macs and iPads at once)

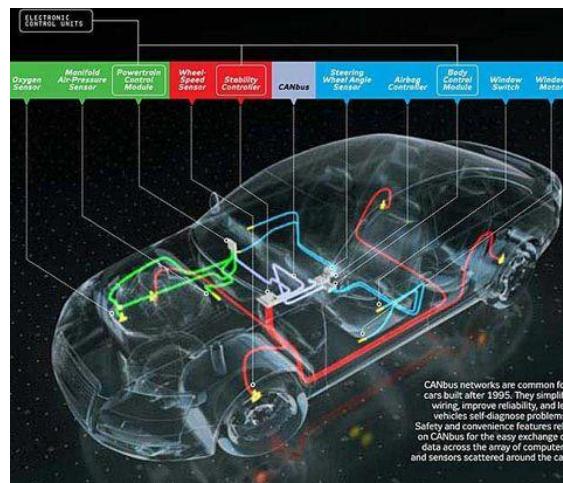
Apple iOS 15 announced June 2021

Google Android 12 available late 2021

Linux distributions: one for every user

Overall: Ubuntu, Server: CentOS, Gaming: Fedora Games Spin,
Lightweight: Lubuntu, for Programmers: Fedora, Beginner-Friendly:
Manjaro (or Mint), Best-Looking: elementary OS, for Windows Users:
Robolinux, for Kids: Sugar on a Stick (SoS) ...

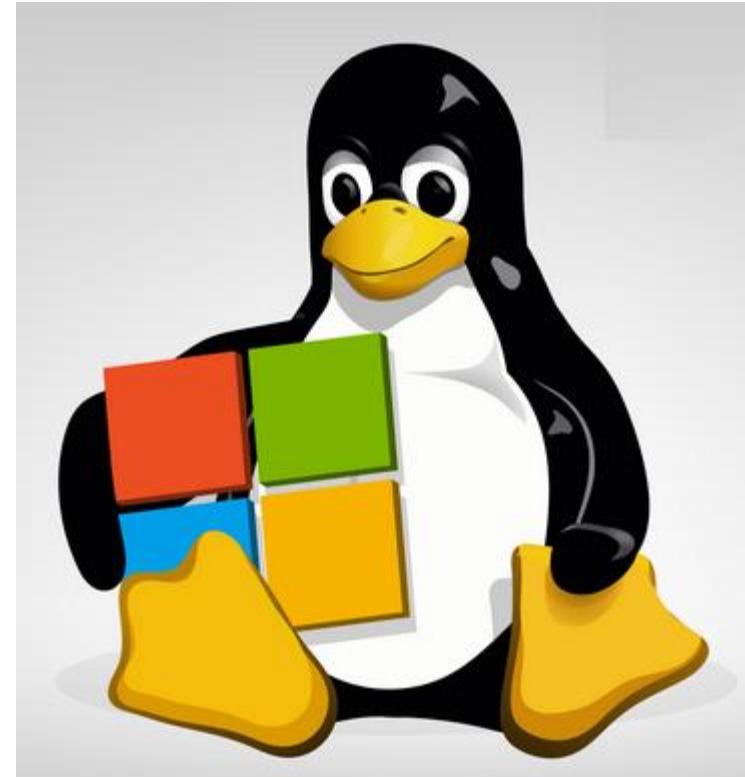
Ubiquitous operating systems



ENCE360

Operating

Systems



*Good luck with
your exams!*