Operating Systems

Julian Bradfield

jcb+os@inf.ed.ac.uk

JCMB-2610

Course Aims

- general understanding of structure of modern computers
- purpose, structure and functions of operating systems
- illustration of key OS aspects by example

Course Outcomes

By the end of the course you should be able to

- describe the general architecture of computers
- describe, contrast and compare differing structures for operating systems
- understand and analyse theory and implementation of: processes, resource control (concurrency etc.), physical and virtual memory, scheduling, I/O and files

In addition, during the practical exercise and associated self-study, you will:

- become familiar (if not already) with the C language, gcc compiler, and Makefiles
- understand the high-level structure of the Linux kernel both in concept and source code
- acquire a detailed understanding of one aspect (the scheduler) of the Linux kernel

Course Outline

This outline is subject to modification during the course.

- Introduction; history of computers; overview of OS (this lecture)
- Computer architecture (high-level view); machines viewed at different abstraction levels
- Basic OS functions and the historical development of OSes
- Processes (1)
- Processes (2) – threads and SMP
- Scheduling (1) – cpu utilization and task scheduling
- Concurrency (1) – mutual exclusion, synchronization
- Concurrency (2) – deadlock, starvation, analysis of concurrency

- Memory (1) – physical memory, early paging and segmentation techniques
- Memory (2) – modern virtual memory concepts and techniques
- I/O (1) – low level I/O functions
- I/O (2) – high level I/O functions and filesystems
- Case studies: one or both of: the Windows NT family; IBM's System/390 family – N.B. you will be expected to study Linux during the practical exercise and in self-study.
- Other topics to be determined, e.g. security.

Assessment

The course is assessed by a written examination (75%), one practical exercise (15%) and an essay (10%).

The practical exercise will run through weeks 3–7, and will involve understanding and modifying the Linux kernel. The final assessed outcome is a relatively small part of the work, and will not be too hard; most of the work will be in understanding C, Makefiles, the structure of a real OS kernel, etc. This is essential for real systems work!

The essay will be due at the end of teaching semester, and will be on a topic (or topics) to be decided, either a more extensive investigation of something covered briefly in lectures, or a study of something not covered.

Textbooks

There are many very good operating systems textbooks, most of which cover the material of the course (and more).

I shall be (very loosely) following
W. Stallings *Operating Systems: Internals and Design Principles* (4th edition), Prentice-Hall, 2001.

Another book that can as well be used is
A. Silberschatz and P. Galvin *Operating Systems Concepts* (5th edition), Addison-Wesley, 1998.

Most of the other major OS texts are also suitable.

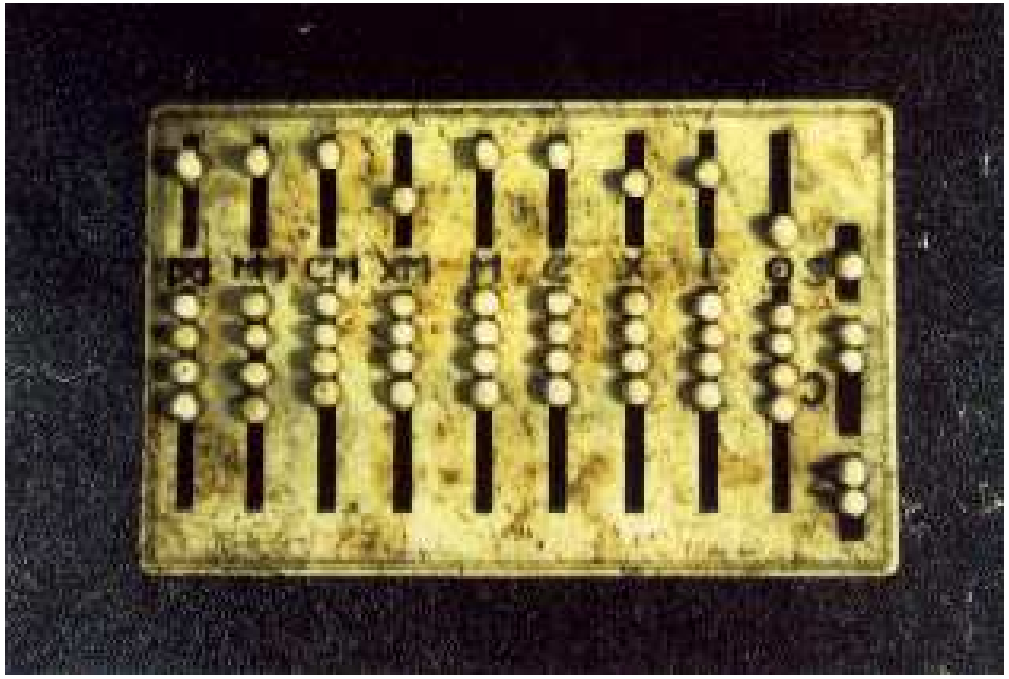
Acknowledgement

I should like to thank Dr Steven Hand of the University of Cambridge, who has provided me with many useful figures for use in my slides, and allowed me to use some of his slides as a basis for some of mine.

A brief and selective history of computing . . .

Computing machines have been increasing in complexity for many centuries, but only recently have they become complex enough to require something recognizable as an operating system. Here, mostly for fun, is a quick review of the development of computers.

The abacus – some millennia BP.



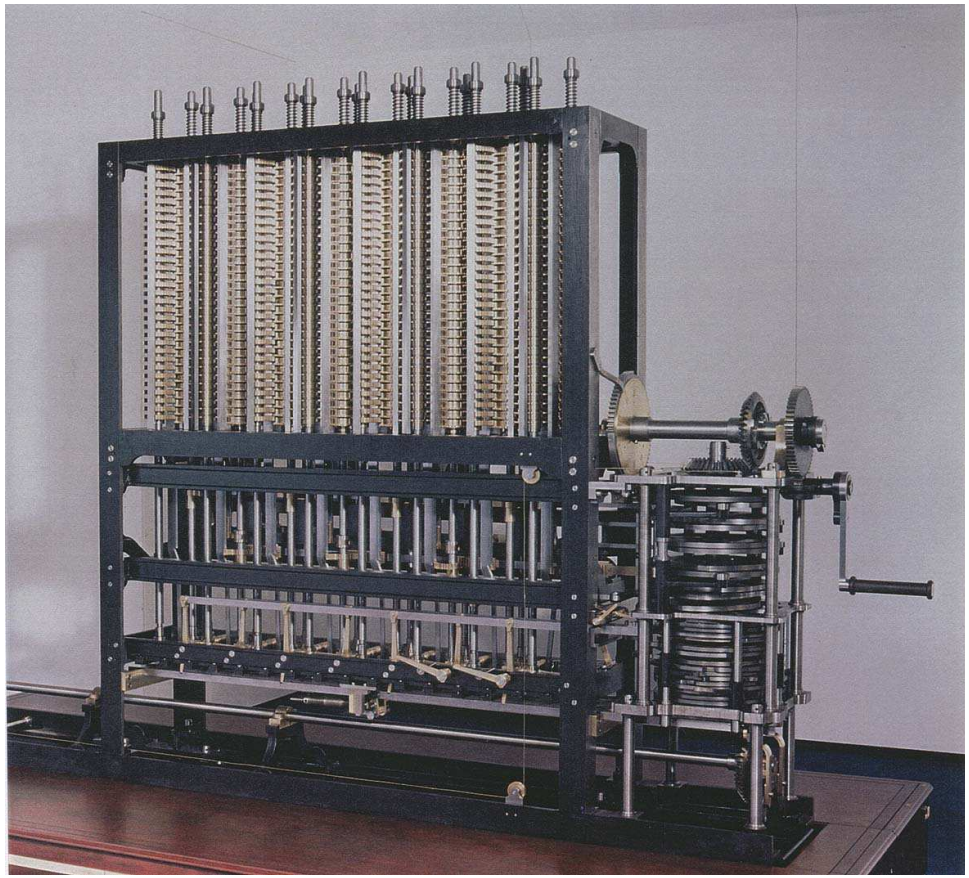
[Association pour le musée international du calcul de l'informatique et de l'automatique de Valbonne Sophia Antipolis (AMISA)]

Logarithms (Napier): the slide rule – 1622 Bissaker

First mechanical digital calculator – 1642 Pascal

[original source unknown]

The Difference Engine, [The Analytical Engine] – 1812, 1832 Babbage / Lovelace.



[Science Museum ??]

Analytical Engine (never built) anticipated many modern aspects of computers. See http://www.fourmilab.ch/babbage/.

Electro-mechanical punched card – 1890 Hollerith ($\rightarrow$ IBM)

Vacuum tube – 1905 De Forest

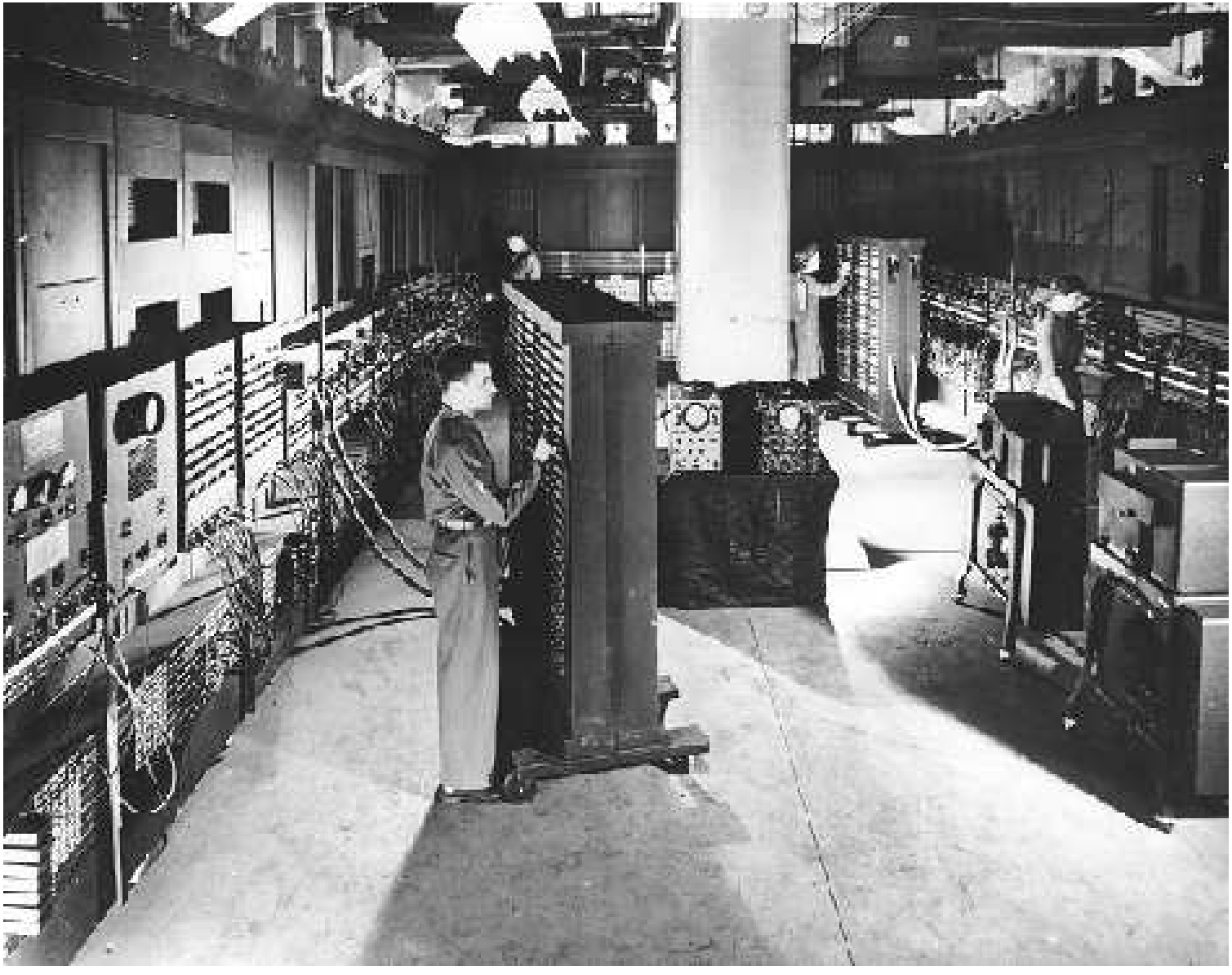
Relay-based IBM 610 hits 1 MultiplicationPS – 1935

ABC, 1st electronic digital computer – 1939 Atanasoff / Berry

Z3, 1st programmable computer – 1941 Zuse

Colossus, Bletchley Park – 1943

ENIAC – 1945, Eckert & Mauchley

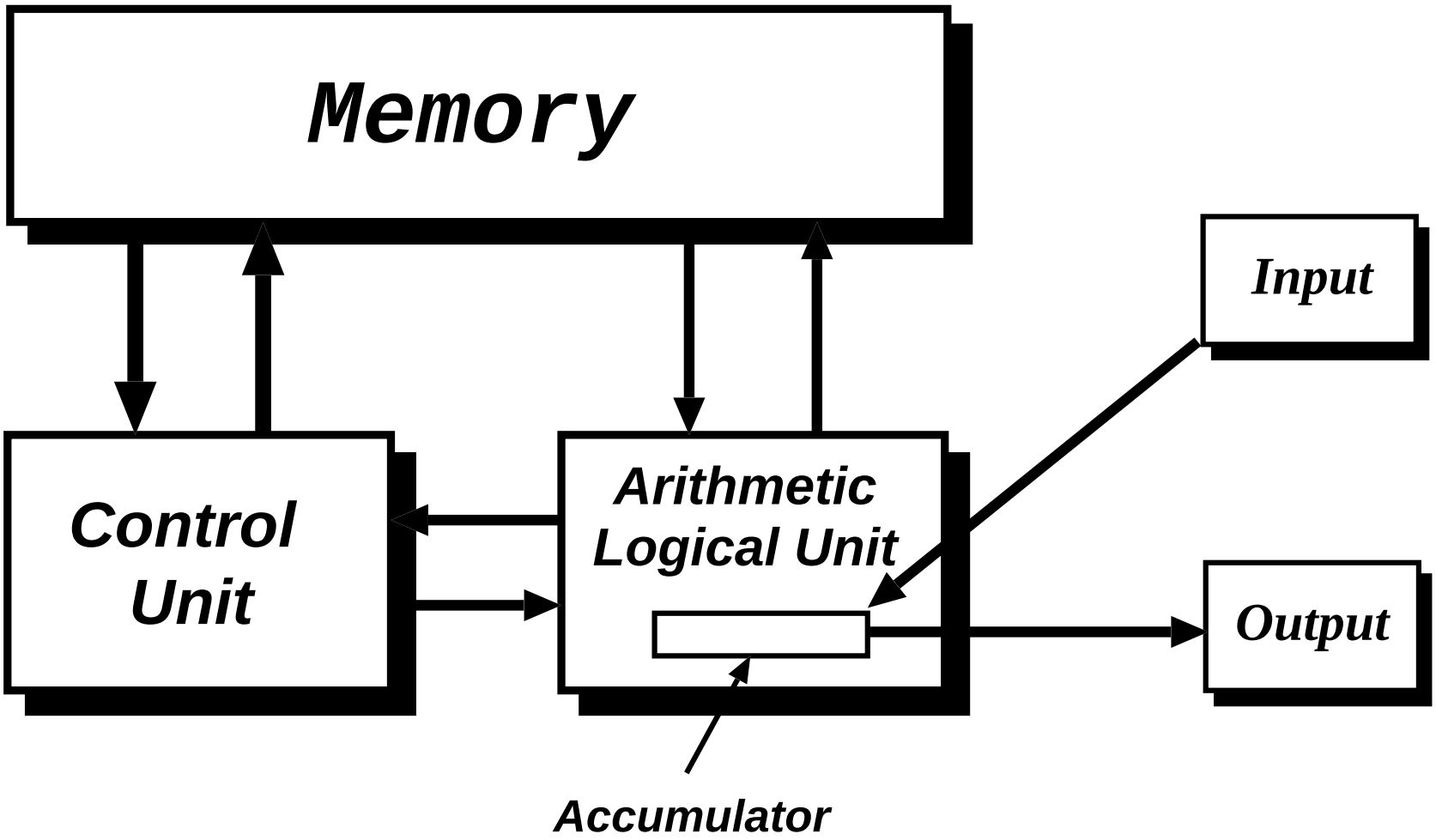


[University of Pennsylvania]

- 30 tons, 1000 sq feet, 140 kW
- 18k vacuum tubes, 20 10-digit accumulators
- 100 kHz, around 300 M(ult)PS
- in 1946 added blinking lights for the Press!

Programmed by a plugboard, so very slow to change program.

The Von Neumann Architecture



In 1945, John von Neumann drafted the EDVAC report, which set out the architecture now taken as standard.

the transistor – 1947 (Shockley, Bardeen, Brattain)

EDSAC, 1st stored program computer – 1949 (Wilkes)

- 3k vacuum tubes, 300 sq ft, 12 kW
- 500kHz, ca 650 IPS
- 1K 17-bit words of memory (Hg ultrasonic delay lines)
- operating system of 31 words
- see `http://www.dcs.warwick.ac.uk/~edsac/` for a simulator

TRADIC, 1st valve-free computer – 1954 (Bell Labs)

first IC – 1959 (Kilby & Noyce, TI)

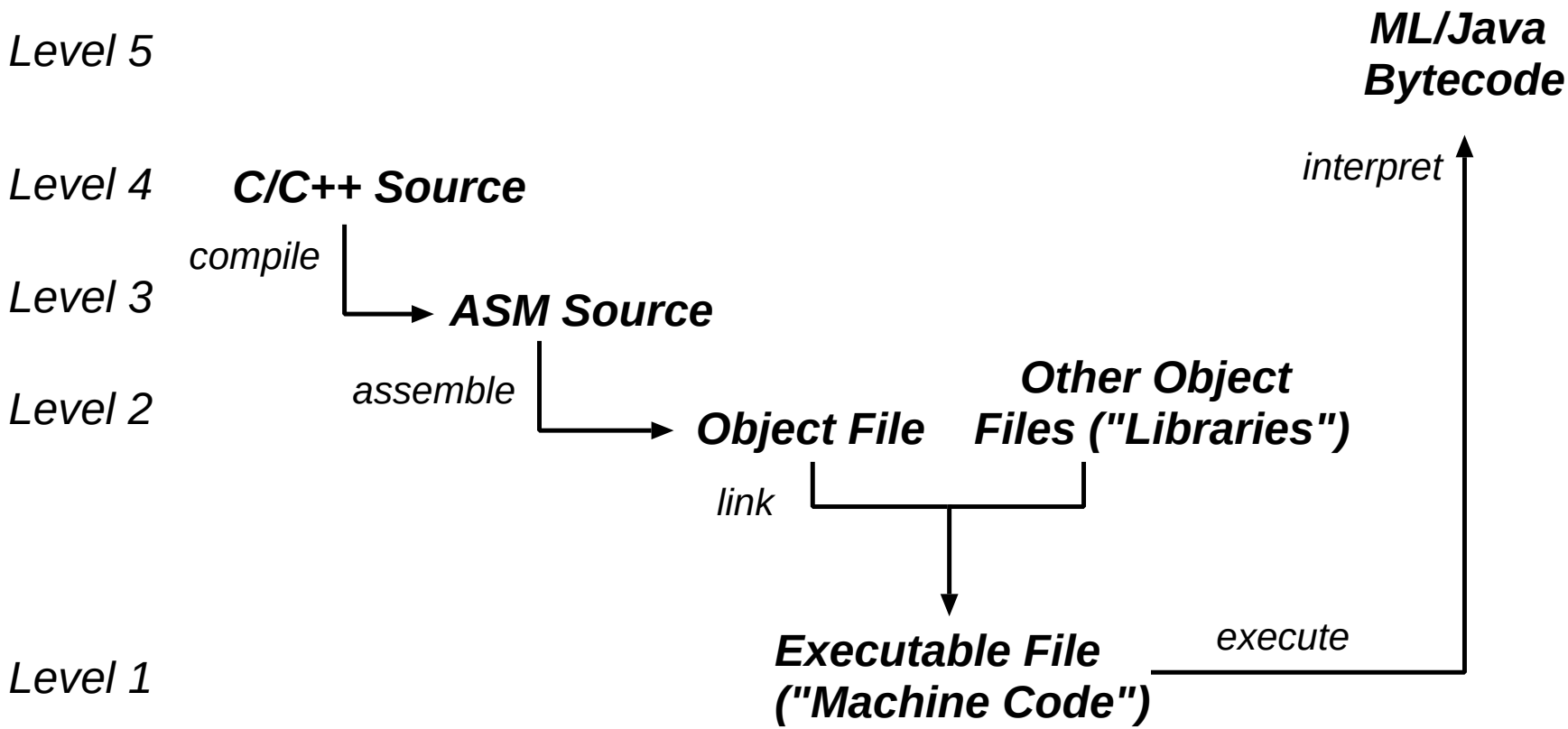
IBM System/360 – 1964. Direct ancestor of today's zSeries, with continually evolved operating system.

Intel 4004, 1st μ-processor – 1971 (Ted Hoff)

Intel 8086, IBM PC – 1978

VLSI ($>$ 100k transistors) – 1980

Levels of (Programming) Languages



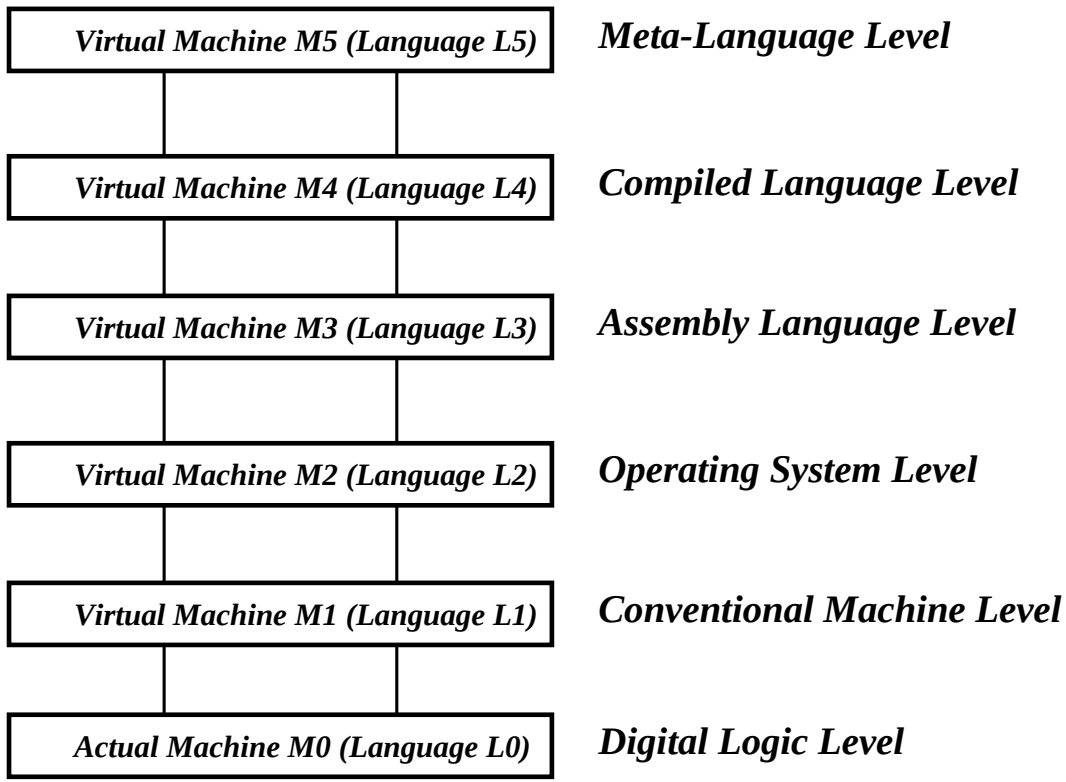
(Modern) Computers can be programmed at several levels.

Level relates to lower via either translation/compilation or interpretation.

Similarly operation of a computer can be described at many levels.

Exercise: justify (or attack) the placing of bytecode in Level 5 in the diagram.

Layered Virtual Machines

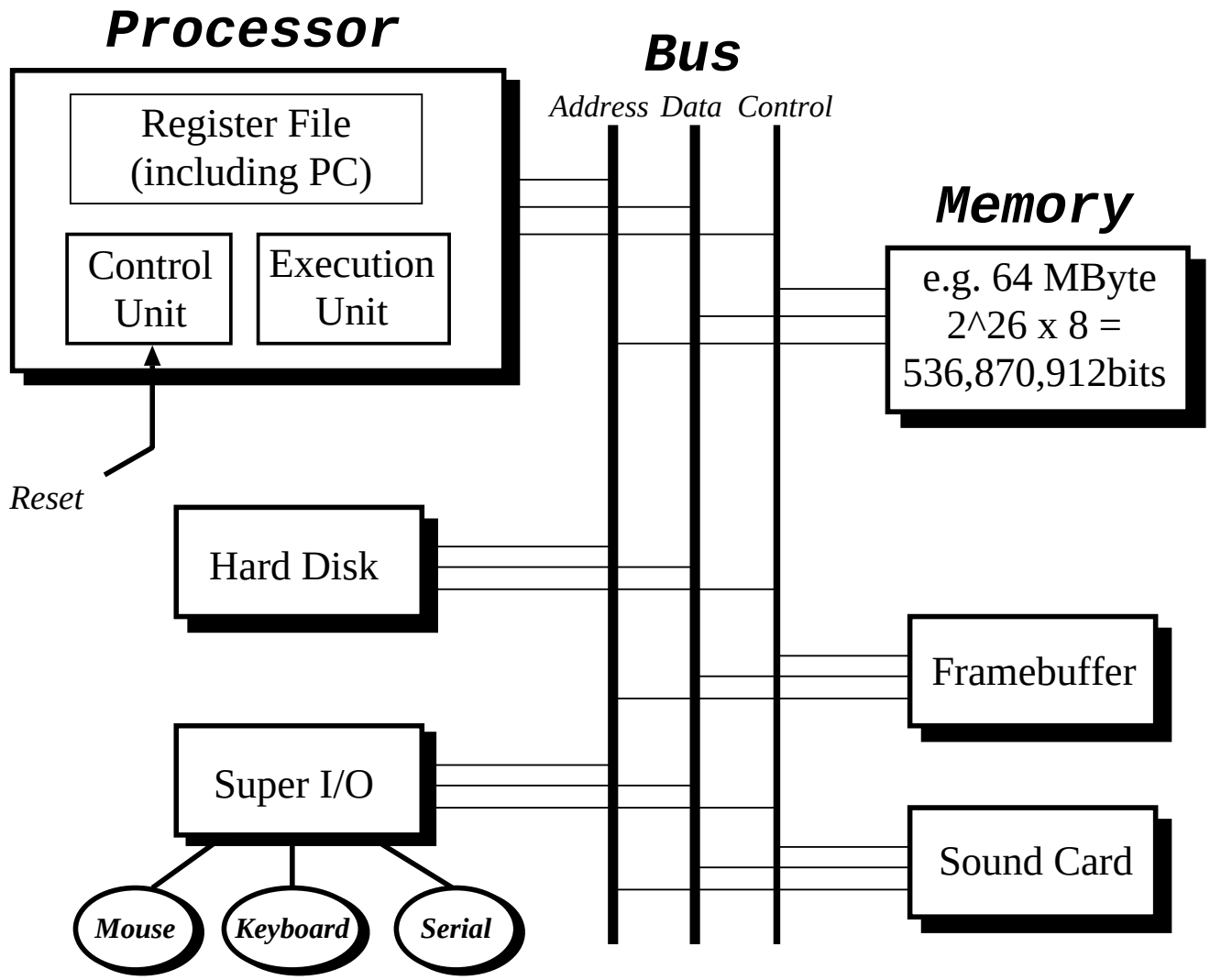


Think of a virtual machine in each layer built on the lower VM; machine in one level understands language of that level.

This course considers mainly levels 1 and 2.

Exercise: Operating Systems are often written in assembly language or C or higher. What does it mean to say level 2 is below levels 3 and 4?

Quick Review of Computer Architecture



(Please revise Inf2!)

Registers

(Very) fast on-chip memory.

Typically 32 or 64 bits; nowadays from 8 to 128 registers is usual.

Data is loaded from memory into registers before being operated on.

Registers may be purely internal and not visible to the programmer, even at machine code level.

Most processors distinguish data and control registers: bits in a control register have special meaning to CPU.

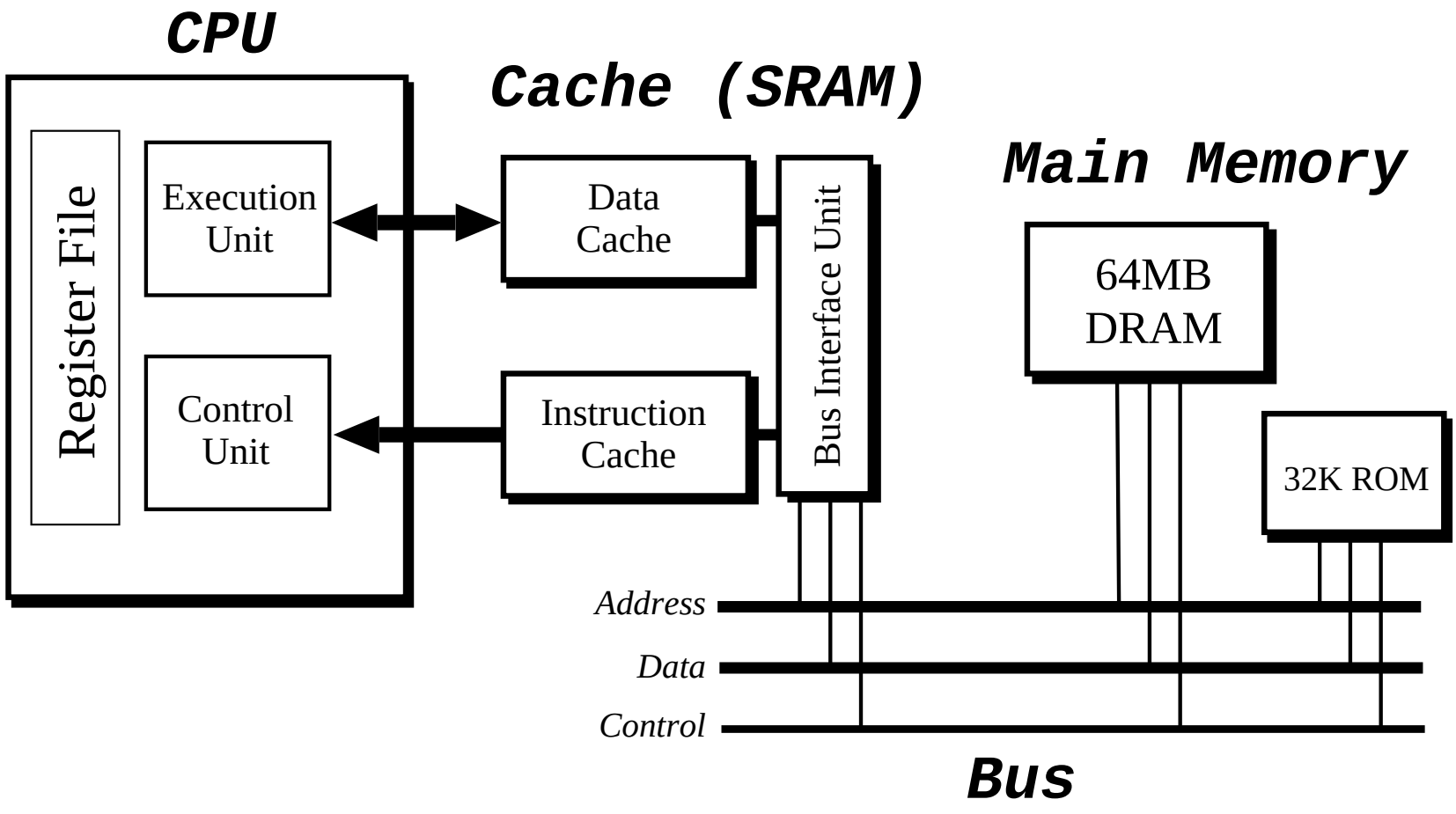
Intel Pentium has:

- eight 32-bit general purpose registers
- six 16-bit segment registers (for address space management)
- two 32-bit control registers, including Program Counter (called EIP by Intel)

IBM z/Architecture has:

- sixteen 64-bit general registers
- sixteen 64-bit floating point registers
- one 32-bit floating point control register
- sixteen 64-bit control registers
- sixteen 32-bit access registers (for address space management)
- one Program Status Word (PC)

Memory Hierarchy

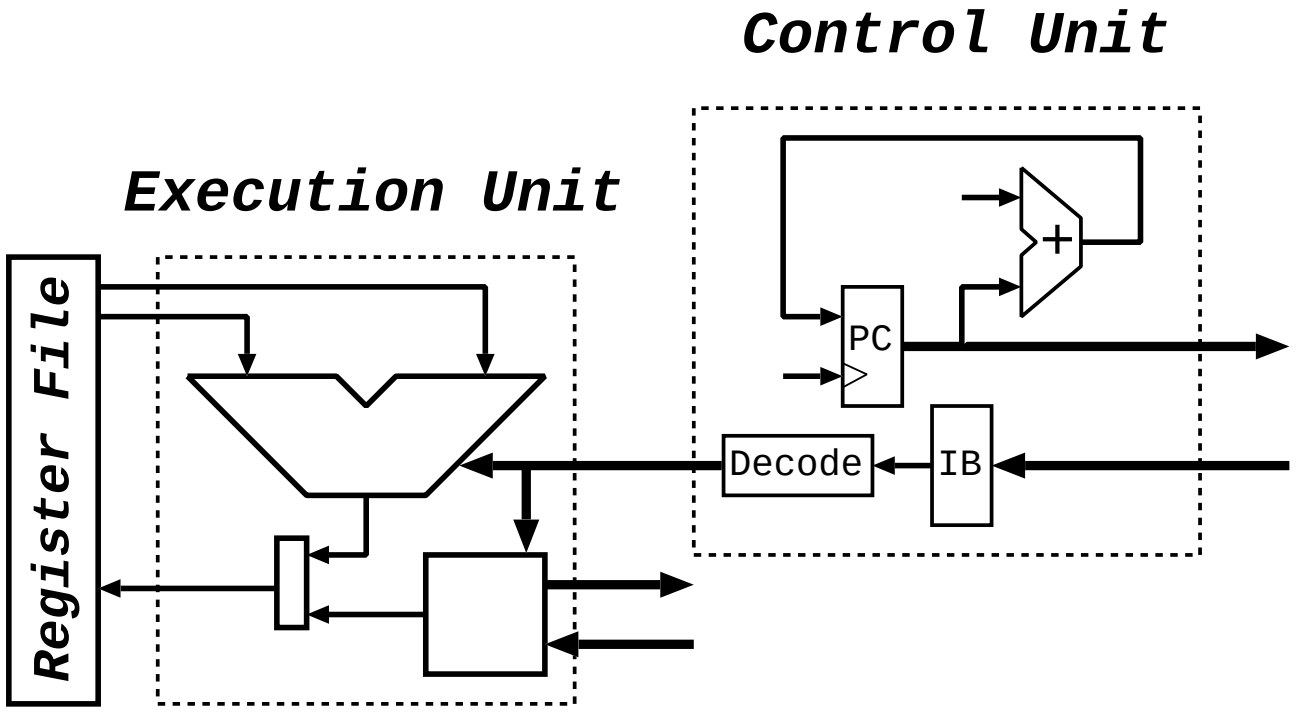


The cache is fast, expensive memory sitting between CPU and main memory – cache $\leftrightarrow$ CPU via special bus.

May have several levels of cache.

The OS has to be aware of the cache and control it, e.g. when switching address spaces.

The Fetch–Execute Cycle



PC initialized to fixed value on CPU reset. Then repeat (until halt):

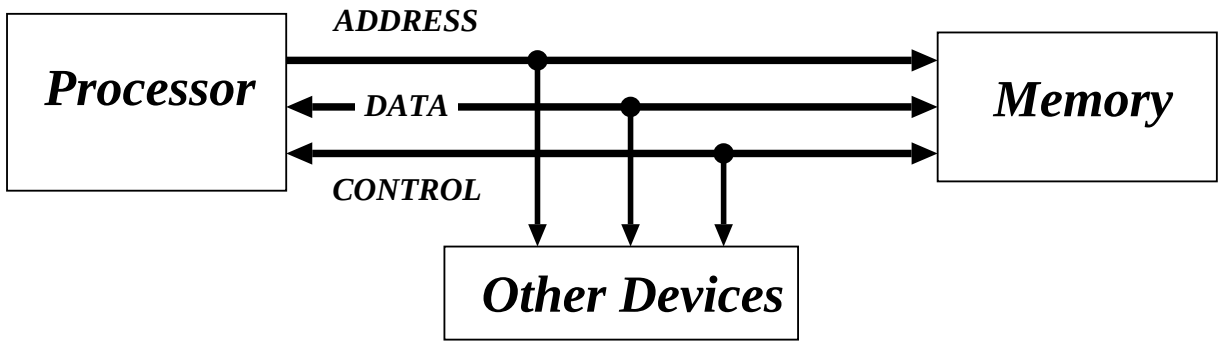
1. instruction is *fetched* from memory address in *PC* into instruction buffer
2. Control Unit *decodes* instruction
3. Execution Unit *executes* it
4. *PC* is updated: explicitly by jumps, implicitly otherwise

Input/Output Devices

We'll consider these later in the course. For now, note that:

- I/O devices typically connected to CPU via a *bus* (or via a chain of buses and bridges)
- wide range of devices, e.g.: hard disk, floppy, CD, graphics card, sound card, ethernet card, modem
- often with several stages and layers
- all of which are very slow compared to CPU.

Buses



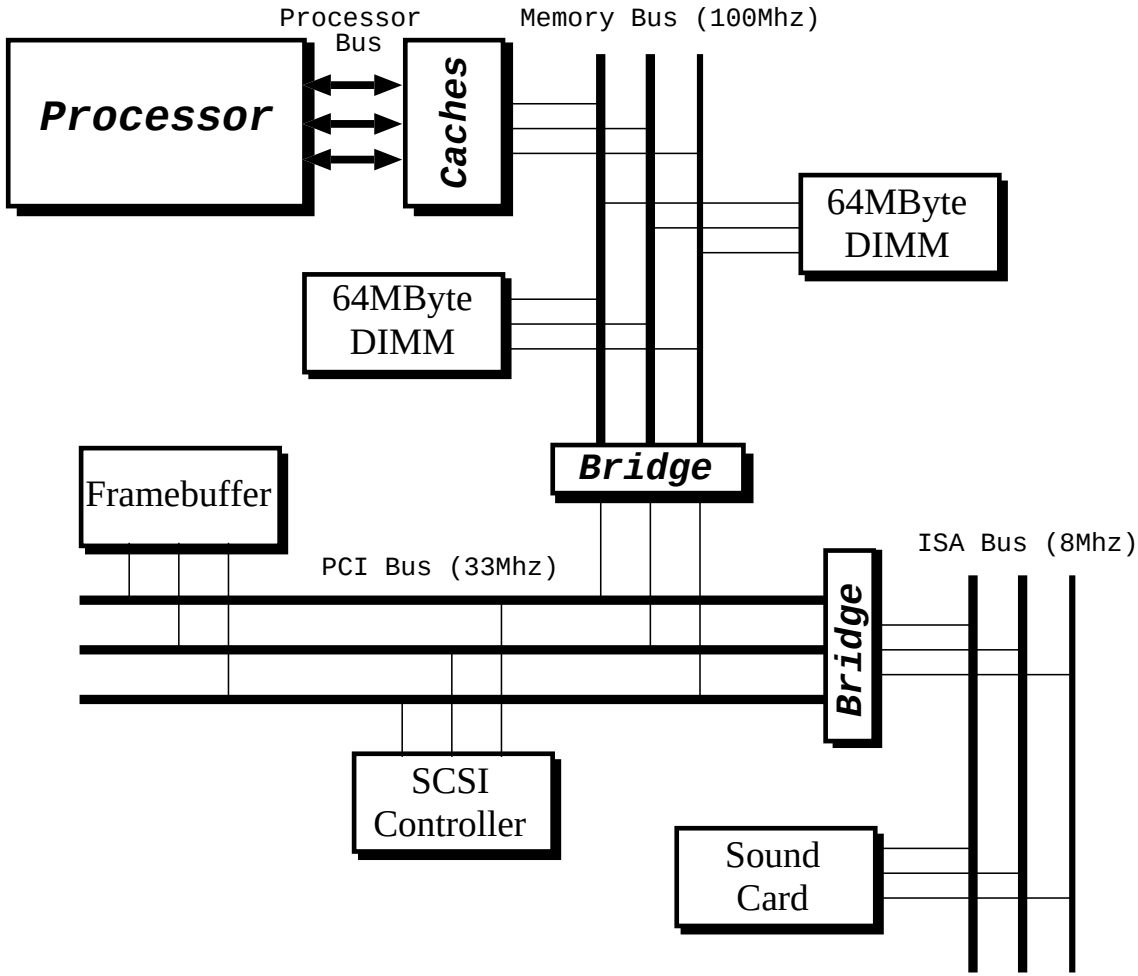
A bus is a group of 'wires' shared by several devices (e.g. CPU, memory, I/O). Buses are cheap and versatile, but can be a severe performance bottleneck (e.g. PC-card hard disks).

A bus typically has address lines, data lines and control lines.

Operated in master–slave protocol: e.g. to read data from memory, CPU (master) puts address on bus and asserts 'read'; memory (slave) retrieves data, puts data on bus; CPU reads from bus.

In some cases, may need initialization protocol to decide which device is the bus master; in others, it's pre-determined.

Bus Hierarchy



Most computers have many different buses, with different functions and characteristics.

Interrupts

Devices much slower than CPU; can't have CPU wait for device. Also, external events may occur.

Interrupts provide suitable mechanism. Interrupt is (logically) a signal line into CPU. When asserted, CPU jumps to particular location (e.g. on x86, on interrupt (IRQ) n, CPU jumps to address stored in nth entry of table pointed to by IDTR control register).

The jump saves state; when the *interrupt handler* finishes, it uses a special return instruction to restore control to original program.

Thus, I/O operation is: instruct device and continue with other tasks; when device finishes, it raises interrupt; handler gets info from device etc. and schedules requesting task.

In practice (e.g. x86), may be one or two interrupt pins on chip, with interrupt controller to encode external interrupts onto bus for CPU.

Direct Memory Access (DMA)

DMA means allowing devices to write *directly* (i.e. via bus) into main memory.

E.g., CPU tells device 'write next block of data into address x'; gets interrupt when done.

PCs have basic DMA; IBM mainframes' 'I/O channels' are a sophisticated extension of DMA (CPU can construct complex programs for device to execute).

So what is an Operating System *for*?

An OS must . . .

handle relations between CPU/memory and devices (relations between CPU and memory are usually in CPU hardware);

handle allocation of memory;

handle sharing of memory and CPU between different logical tasks;

handle file management;

ever more sophisticated tasks . . .

... in Windows, handle most of the UI graphics. (Is this OS business?)

Exercise: In Stallings, or on the Web, find the Brown/Denning hierarchy of OS functions. Discuss the ordering of the hierarchy, paying particular attention to levels 5 and 6. Which levels does the Linux kernel handle? And Windows 2000?

(*kernel*: the single (logical) program that is loaded at boot time and has primary control of the computer.)

In the beginning. . .

Earliest 'OS' simply transferred programs from punched card reader to memory.

Everything else done by lights and switches on front panel.

Job scheduling done by sign-up sheets.

User (= programmer = operator) had to set up entire job (e.g.: load compiler, load source code, invoke compiler, etc) programmatically.

I/O directly programmed.

First improvements

Users write programs and give tape or cards to *operator*.

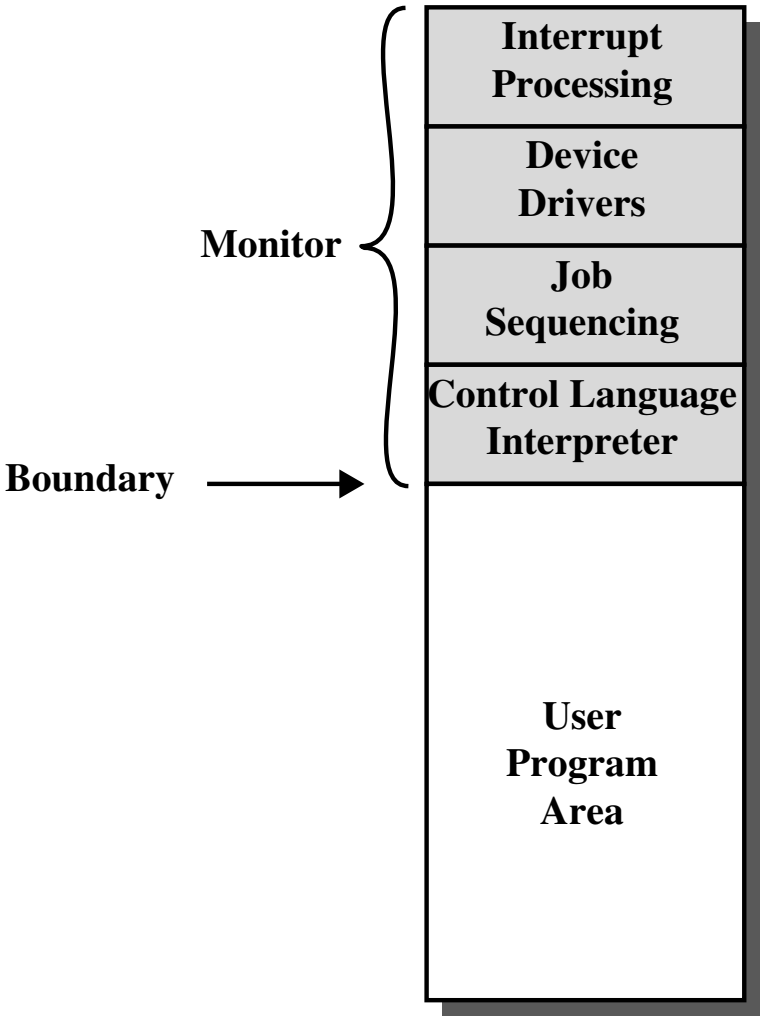
Operator feeds card reader, collects output, returns it to users.

(Improvement for user – not for operator!)

Start providing standard card libraries for linking, loading, I/O drivers, etc.

Early batch systems

Late 1950s–early 1960s saw introduction of *batch systems* (General Motors, IBM; standard on IBM 7090/7094).



- *monitor* is simple resident OS: reads jobs, transfers control to program, receives control back from program at end of task.
- *batches* of jobs can be put onto one tape and read in turn by monitor – reduces human intervention.
- monitor permanently resident: user programs must be loaded into different area of memory

Protecting the monitor from the users

Having monitor co-resident with user programs is asking for trouble. Desirable features, needing hardware support, include:

- memory protection: user programs should not be able to . . . write to monitor memory,
- timer control: . . . or run for ever,
- privileged instructions: . . . or directly access I/O (e.g. might read next job by mistake) or certain other machine functions,
- interrupts: . . . or delay the monitor's response to external events

Making good use of resource – multiprogramming

Even in the 60s, I/O was very slow compared to CPU. So jobs would waste most (typically $> 75\%$) of the CPU cycles waiting for I/O.

Multiprogramming introduced: monitor loads several user programs; when one is waiting for I/O, run another.

Multiprogramming means the monitor must:

- manage memory among the various tasks
- schedule execution of the tasks

Multiprogramming OSes introduced early 60s – Burroughs MCP (1963) was early (and advanced) example.

In 1964, IBM introduced System/360 hardware architecture. Family of architectures, still going strong (S/360 $\rightarrow$ S/370 $\rightarrow$ S/370-XA $\rightarrow$ ESA/370 $\rightarrow$ ESA/390 $\rightarrow$ z/Architecture). Simulated/emulated previous IBM computers.

Early S/360 OSes not very advanced: DOS single batch; MFT ran fixed number of tasks. In 1967 MVT ran up to 15 tasks.

Using batch systems was (and is) pretty painful. E.g. on MVS, to assemble, link and run a program:

```
//USUAL     JOB   A2317P,'MAE BIRDSALL'
//ASM       EXEC  PGM=IEV90,REGION=256K,          EXECUTES ASSEMBLER
//             PARM=(OBJECT,NODECK,'LINECOUNT=50')
//SYSPRINT  DD    SYSOUT=*,DCB=BLKSIZE=3509  PRINT THE ASSEMBLY LIS
//SYSPUNCH  DD    SYSOUT=B                   PUNCH THE ASSEMBLY LIS
//SYSLIB    DD    DSNAME=SYS1.MACLIB,DISP=SHR    THE MACRO LIBRARY
//SYSUT1    DD    DSNAME=&&SYSUT1,UNIT=SYSDA,    A WORK DATA SET
//             SPACE=(CYL,(10,1))
//SYSLIN    DD    DSNAME=&&OBJECT,UNIT=SYSDA,  THE OUTPUT OBJECT MO
//             SPACE=(TRK,(10,2)),DCB=BLKSIZE=3120,DISP=(,PASS)
//SYSIN     DD    *                              IN-STREAM SOURCE C
            .
            code
            .
/*
```

```
//LKED      EXEC  PGM=HEWL,                          EXECUTES LINKAGE ED
//             PARM='XREF,LIST,LET',COND=(8,LE,ASM)
//SYSPRINT  DD    SYSOUT=*                           LINKEDIT MAP PRINT
//SYSLIN    DD    DSNAME=&&OBJECT,DISP=(OLD,DELETE) INPUT OBJECT MO
//SYSUT1    DD    DSNAME=&&SYSUT1,UNIT=SYSDA,    A WORK DATA SET
//             SPACE=(CYL,(10,1))
//SYSLMOD   DD    DSNAME=&&LOADMOD,UNIT=SYSDA,   THE OUTPUT LOAD MO
//             DISP=(MOD,PASS),SPACE=(1024,(50,20,1))
//GO        EXEC  PGM=*.LKED.SYSLMOD,TIME=(,30), EXECUTES THE PROGR
//             COND=((8,LE,ASM),(8,LE,LKED))
//SYSUDUMP  DD    SYSOUT=*                           IF FAILS, DUMP LIS
//SYSPRINT  DD    SYSOUT=*,                          OUTPUT LISTING
//             DCB=(RECFM=FBA,LRECL=121)
//OUTPUT    DD    SYSOUT=A,                          PROGRAM DATA OUTPU
//             DCB=(LRECL=100,BLKSIZE=3000,RECFM=FBA)
//INPUT     DD    *                                  PROGRAM DATA INPUT
            .
            data
            .
/*
//
```

Time-sharing

Allow interactive terminal access to computer, with many users sharing.

Early system (CTSS, Cambridge, Mass.) gave each user 0.2s of CPU time; monitor then saved user program state, loaded state of next scheduled user.

IBM's TSS for S/360 was similar – and a software engineering disaster. Major motivation for development of SE!

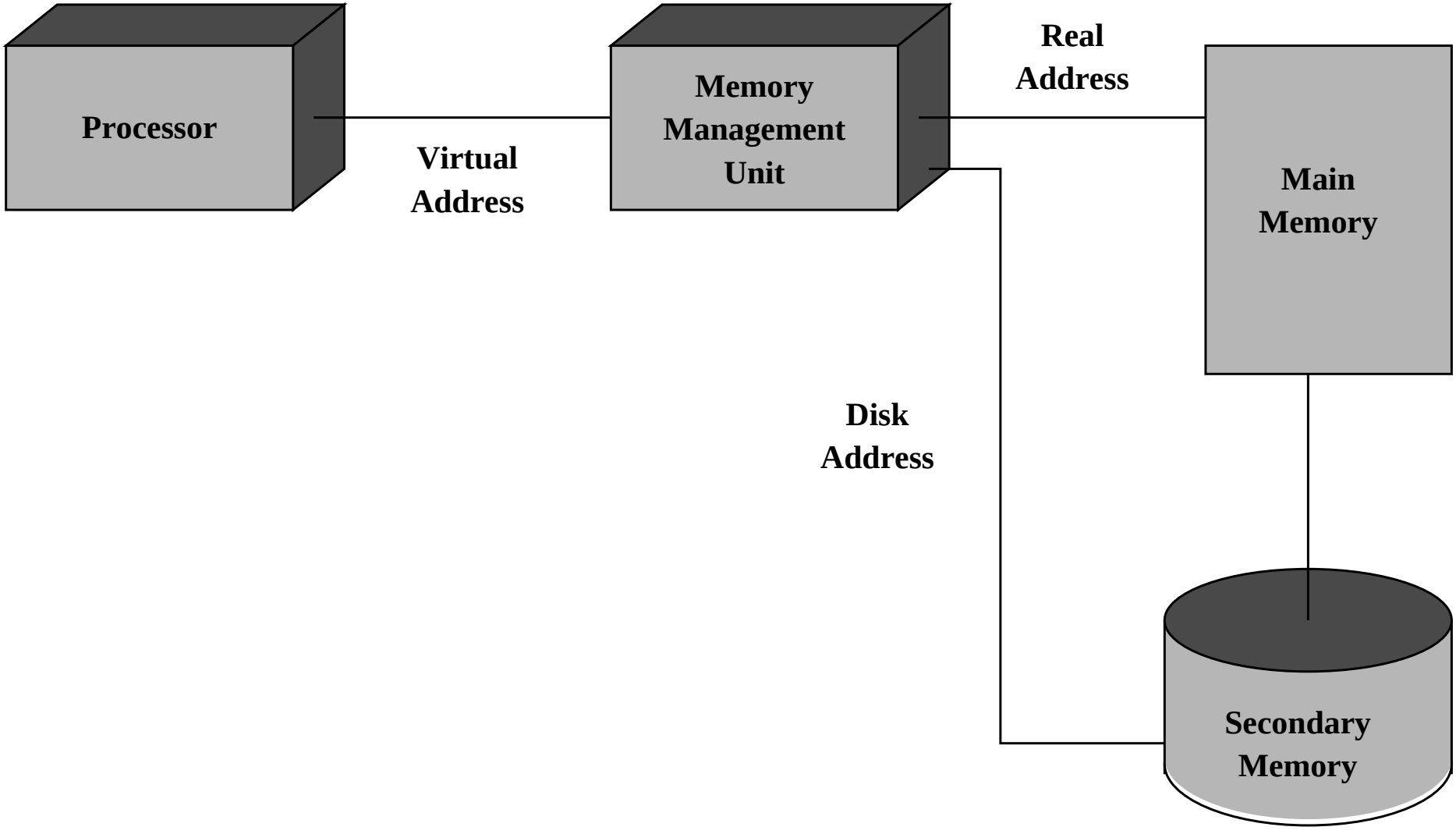
Virtual Memory

Multitasking, and time-sharing in particular, much easier if all tasks are resident, rather than being swapped in and out of memory.

But not enough memory! *Virtual memory* decouples memory as seen by the user task from physical memory. Task sees virtual memory, which may be anywhere in real memory, and can be paged out to disk.

Hardware support required: all memory references by user tasks must be translated to real addresses – and if the virtual page is on disk, monitor called to load it back in real memory.

In 1963, Burroughs had virtual memory. IBM only introduced it to mainframe line with S/370 in 1972.



Virtual Memory Addressing

The Process Concept

With virtual memory, becomes natural to give different tasks their own independent *address space* or view of memory. Monitor then schedules *processes* appropriately, and does all *context-switching* (loading of virtual memory control info, etc.) transparently to user process.

Note on terminology. It's common to use 'process' for task with independent address space, espec. in Unix setting, but this is not a universal definition. Tasks sharing the same address space are called 'tasks' (IBM) or 'threads' (Unix). But some older OSes without virtual memory called their tasks 'processes'.

Communication between processes becomes a major issue (studied later); as does control of resources.

Modes of CPU operation

To protect OS from users, all modern CPUs operate in more than one *privilege level*

- S/370 family has *supervisor* and *problem* states
- Intel x86 has *rings 0,1,2,3*.

Transition to a higher privilege level only allowed via tightly controlled mechanisms. E.g. IBM SVC (supervisor call) or Intel INT are like software interrupts: change to supervisor mode and jump to pre-determined address.

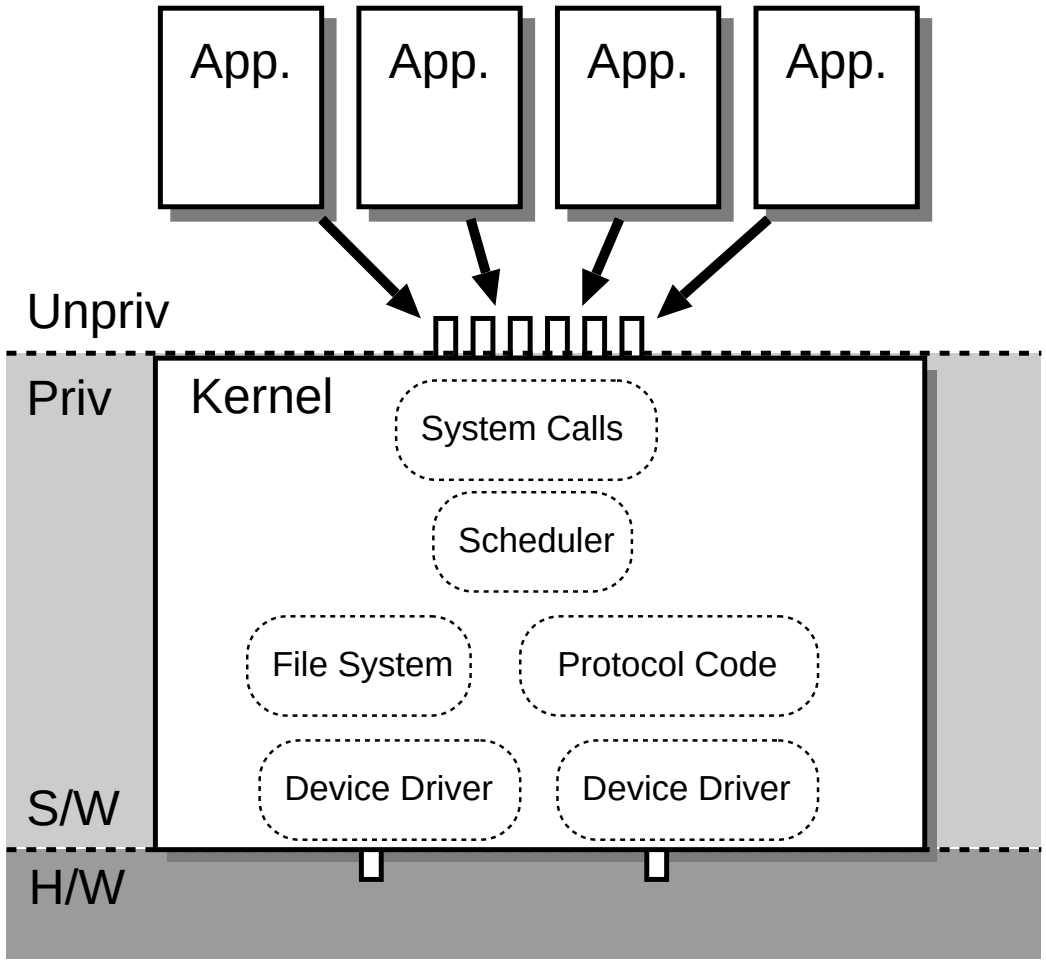
CPU instructions that can damage system are restricted to supervisor state: e.g. virtual memory control, I/O.

Memory Protection

Virtual memory itself allows user's memory to be isolated from kernel memory and other users' memory. Both for historical reasons and to allow user/kernel memory to be appropriately shared, many architectures have separate protection mechanisms as well:

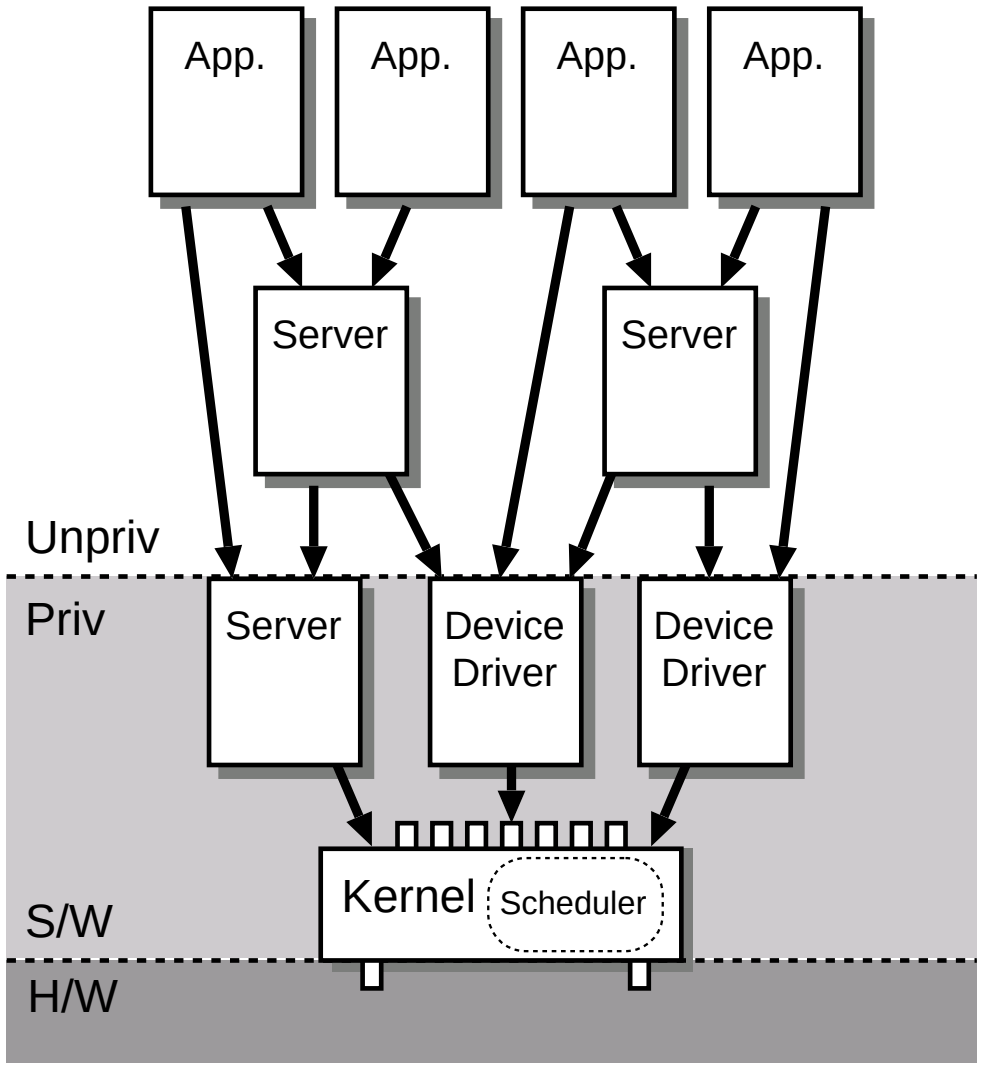
- A frame or page may be read or write accessible only to a processor in a high privilege level;
- In S/370, each frame of memory has a 4-bit storage key, and each task runs with a particular key.
- the virtual memory mechanism may be extended with permission bits; frames can then be shared.
- combination of all the above may be used.

OS structure – traditional



All OS function sits in the kernel. Some modern kernels are very large – tens of MLoC. Bug in any function can crash system...

OS structure – microkernels



Small core, which talks to (maybe privileged) components in separate servers.

Kernel vs Microkernel

Microkernels:

- increase modularity
- increase extensibility

but

- have more overhead (due to IPC)
- can be difficult to implement (synchronization)
- often keep multiple copies of OS data structures

Modern real (rather than CS) OSes are hybrid:

- Linux is monolithic, but has modules that are dynamically (un)loadable
- Windows NT was orig. microkernel-ish, but for performance has put stuff back into kernel.

See GNU Hurd (based on MACH microkernel) ...

Processes – what are they?

Recall that a process is 'a program in execution'; may have own view of memory; sees one processor, although it's sharing it with other processes – running on *virtual processor*.

To switch between processes, we need to track:

- its memory, including stack and heap
- the contents of registers
- program counter
- its *state*

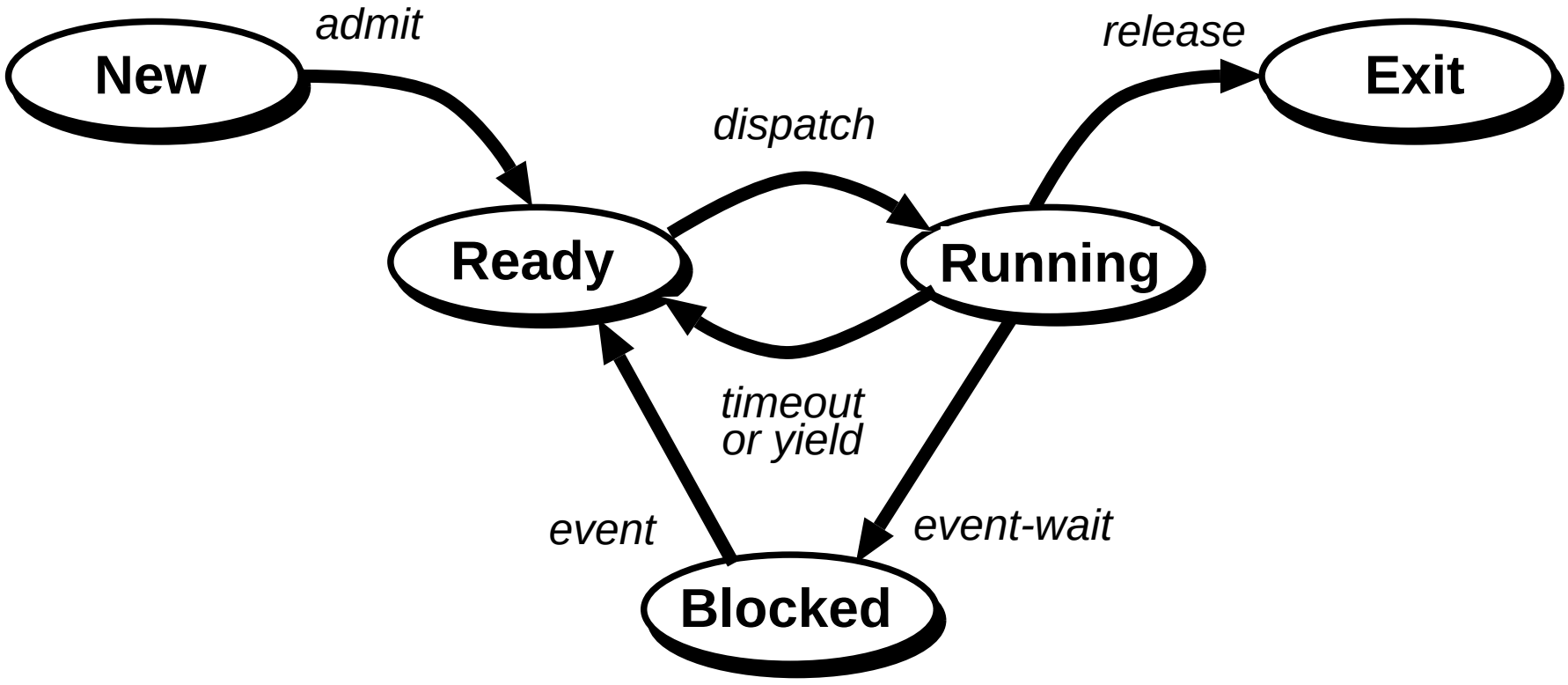
Process States

State is an abstraction used by OS. One standard analysis has five states:

- *New*: process being created
- *Running*: process executing on CPU
- *Ready*: not on CPU, but ready to run
- *Blocked*: waiting for an event (and so not runnable)
- *Exit*: process finished, awaiting cleanup

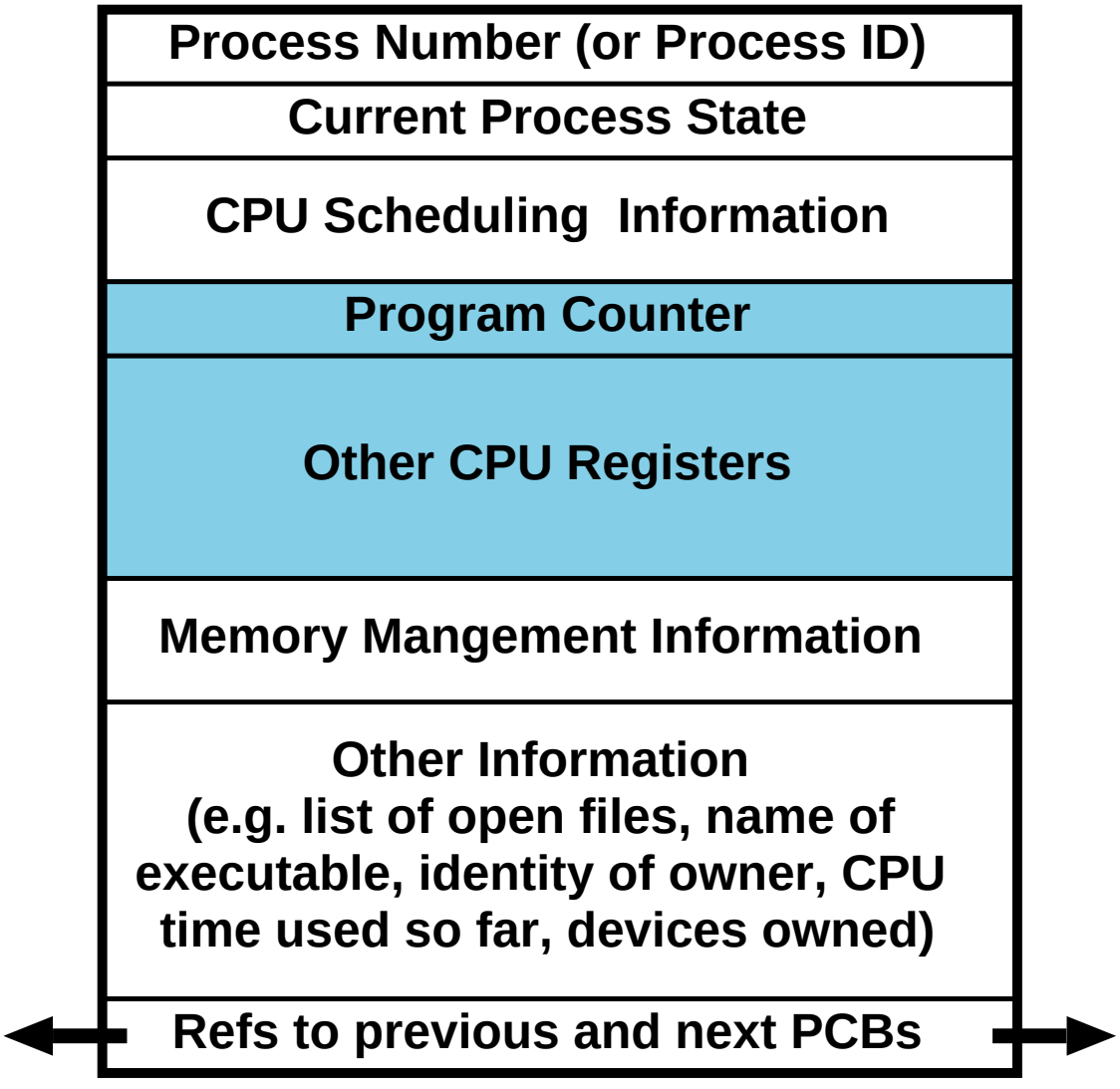
Exercise: find out what process states Linux uses. How do they correspond to this set?

State of process is maintained by OS. Transitions between states happen as follows:



- *admit:* process control set up, move to run queue
- *dispatch:* scheduler gives CPU to runnable process
- *timeout/yield:* running process forced to/volunteers to give up CPU
- *event-wait:* process needs to wait for e.g. I/O
- *event:* event occurs – wake up process and tell it
- *release:* process terminates, release resources

Process Control Block

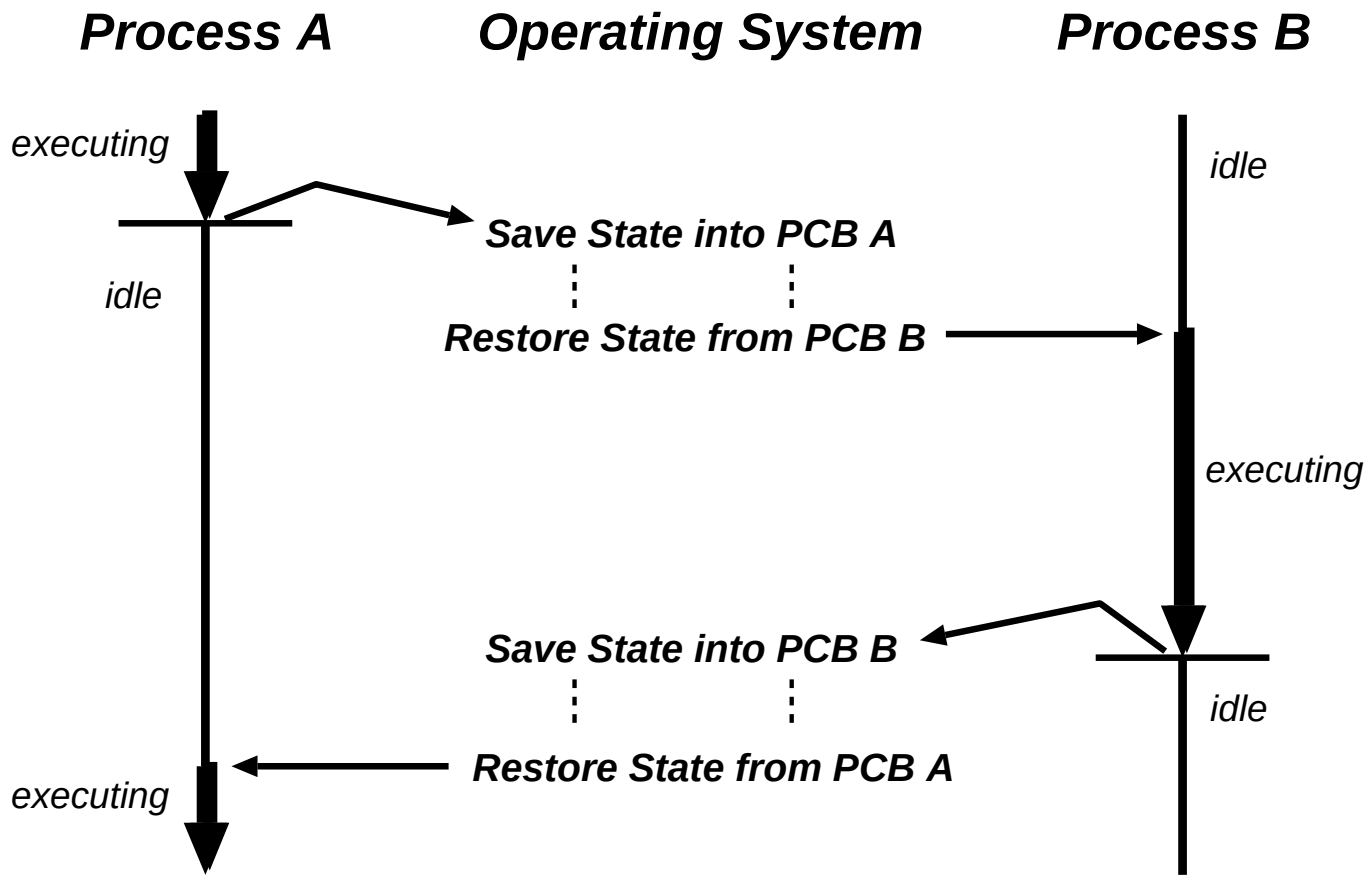


PCB contains all necessary information:

- unique process ID
- process state
- PC and other registers (when not running)
- memory management info
- scheduling and accounting info
- ...

Context Switching

PCB allows OS to switch process *contexts*:



Time-consuming, so modern CPUs provide H/W support. (About 80 pages in IBM ESA/390 manual – complex, sophisticated, rarely used mechanisms.)

Kernel Context?

In what context does the kernel execute?

- in older OSes, kernel is seen as single program in real memory
- in modern OSes, kernel may execute in context of user process
- parts of OS may even be processes (in some sense)

For example, in both Unix and OS/390, I/O is dealt with by kernel code running in context of user process, but master scheduler is independent of user processes.

(Using advanced features of S/390, the OS/390 kernel may be executing in the context of several user processes.)

Scheduling

When do processes move from Ready to Running? This is the job of the *scheduler*. We will look at this in detail later.

Creating Processes (1)

How, why, when are processes created?

- By the OS when a job is submitted or a user logs on.
- By the OS to perform background service for user (e.g. printing).
- By explicit request from user program (spawn, fork).

In Unix, create a new process (and address space) for every program executed: e.g. shell does `fork()` and child process does `execve()` to load program. N.B. `fork()` creates a full copy of the calling process.

In WinNT, `CreateProcess()` creates new process and loads program.

In OS/390, users create subtasks only for explicit concurrent processing, and all subtasks share same address space. (For new address space, submit batch job...)

Creating Processes(2)

When a process is created, the OS must

- assign unique identifier
- allocate memory space: both kernel memory for control structures, and user memory
- initialize PCB and (maybe) memory management tables
- link PCB into OS data structures
- initialize remaining control structures
- for WinNT, OS/390: load program
- for Unix: make child process a copy of parent

Modern Unices don't *actually* copy; they share and do *copy-on-write*.

Ending Processes

Processes may

- terminate voluntarily (Unix `exit()`)
- perform illegal operation (privileged instruction, access non-existent memory, etc.)
- be killed by user (Unix `kill()`) or OS because
  - allocated resources exceeded
  - task functionality no longer needed
  - parent terminating (in some OSes) ...

On termination, the OS must:

- deal with pending output etc.
- release all system resources held by process
- unlink PCB from OS data structures
- reclaim all user and kernel memory

Processes and Threads

Processes

- own resources such as address space, i/o devices, files
- are units of scheduling and execution

These are logically distinct. Some old OSes (MVS) and most modern OSes (Unix, Windows) allow many *threads* (or lightweight processes [some Unices] or tasks [IBM]) to execute concurrently in one process (or address space [IBM]).

Everything previously said about scheduling applies to threads; but process-level context is shared by the thread contexts. All threads in one process share system resources. Hence

- creating threads is quick (ca. 10 times quicker than processes)
- ending threads is quick
- switching threads within one process is quick
- inter-thread communication is quick and easy (have shared memory)

Thread Operations

Thread state similar to process state. Basic operations similar:

- *create:* thread spawns new thread, specifying instruction pointer or routine to call. OS sets up thread context: registers, stack space, ...
- *block:* thread waits for event. Other threads may execute.
- *unblock:* event occurs, thread become ready.
- *finish:* thread completes; context reclaimed.

Real Threads vs Thread Libraries

Threads can be implemented as part of the OS; e.g. Linux, OS/390, Windows.

If the OS does not do this (or in any case), threads can be implemented by user-space libraries:

- thread library implements mini-process scheduler (entirely in user space), e.g.
- context of thread is PC, registers, stacks etc., saved in
- thread control block (stored in user process's memory)
- switching between threads can happen voluntarily, or on timeout (user level timer, rather than kernel timer)

Advantages include:

- context switching very fast - no OS involvent
- scheduling can be tailored to application
- thread library can be OS-independent

Disadvantages:

- if thread makes blocking system call, entire process is blocked. There are ways to work round this. Exercise: How?
- user-space threads don't execute concurrently on multiprocessor systems.

MultiProcessing

There is always a desire for faster computers. One solution is to use several processors connected together. Following taxonomy is widely used:

- Single Instruction Single Data stream (SISD): normal setup, one processor, one instruction stream, one memory.
- Single Instruction Multiple Data stream (SIMD): a single program executes in lockstep on several processors. E.g. vector processors (used for large scientific applications).
- Multiple Instruction Single Data stream (MISD): not used.
- Multiple Instruction Multiple Data stream (MIMD): many processors each executing different programs on different data.

Within MIMD systems, processors may be *loosely coupled*, for example, a network of separate computers with communication links; or *tightly coupled*, for example processors connected via single bus to shared memory.

Symmetric MultiProcessing – SMP

With shared memory multiprocessing, where does the OS run?

Master–slave: The kernel runs on one CPU, and dispatches user processes to others. All I/O etc. is done by request to the kernel on the master CPU. Easy, but inefficient and failure prone.

Symmetric: The kernel may execute on any CPU. Kernel may be multi-process or multi-threaded. Each processor may have its own scheduler. Much more flexible and efficient – but much more complex. This is SMP.

Exercise: Why is this MIMD, and not MISD?

SMP OS design considerations

- cache coherence: several CPUs, one shared memory. Each CPU has its own cache. What happens when CPU 1 writes to memory that CPU 2 has cached? This problem is usually solved by hardware designers, not OS designers.
- re-entrancy: several CPUs may call kernel simultaneously. Kernel code must be written to allow this.
- scheduling: genuine concurrency between threads. Also between kernel threads.
- memory: must maintain virtual memory consistency between processors (since each CPU has VM hardware support).
- fault tolerance: single CPU failure should not be catastrophic.

Scheduling

Scheduling happens over several time-scales and at several levels.

- Batch scheduling, long-term: which jobs should be started? Depends on, e.g., estimated resource requirements, tape drive requirements, ...
- medium term: some OSes *suspend* or *swap out* processes to ameliorate resource contention. This is a medium term (seconds to minutes) procedure. We won't discuss it. Exercise: read up in the textbooks on suspension/swapout – which modern OSes do it?
- process scheduling, short-term: which process gets the CPU next? How long does it get?

We will consider mainly short-term scheduling here.

Scheduling Criteria

To schedule effectively, need to decide criteria for success! For example,

- good utilization: minimize the amount of CPU idle time
- good utilization: job throughput
- fairness: jobs should all get a 'fair' share of CPU ...
- priority: ... unless they're high priority
- response time: fast (in human terms) response to interactive input
- real-time: hard deadlines, e.g. chemical plant control
- predictability: avoid wild variations in user-visible performance

Balance very system-dependent: on PCs, response time is important, utilization irrelevant; in large financial data centre, throughput is vital.

Non-preemptive Policies

In a non-preemptive policy, once a job gets the CPU, it keeps it until it yields or needs I/O etc. Such policies are often suitable for long-term scheduling; not often used now for short-term. (Obviously poor for interactive response!)

- first-come-first-served: (FCFS, FIFO, queue) – what it says. Favours long and CPU-bound processes over short or I/O-bound processes. Not often appropriate; but used as sub-component of priority systems.
- shortest process next: (SPN) – dispatch process with shortest expected processing time. Improves overall performance, favours short jobs. Poor predictability. How do you estimate expected time? For batch jobs (long-term), user can estimate; for short-term, can build up (weighted) average CPU residency over time as process executes. E.g. exponentially weighted averaging.
- and others ...

Preemptive Policies

Here we interrupt processes after some time (the *quantum*).

- round-robin: when the quantum expires, running process is sent to back of ready queue. Good for general purposes. Tends to favour CPU-bound processes – can be refined to avoid this. How big should the quantum be? 'Slightly greater than the typical interaction time.' (How fast do you type?) Recent Linux kernels have base quantum of around 50ms.
- shortest remaining time: (SRN) – preemptive version of SPN. On quantum expiry, dispatch process with shortest expected running time. Tends to starve long CPU-bound processes. Estimation problem as for SPN.

- feedback: use dynamically assigned priorities:
  - new process starts in queue of priority 0 (highest);
  - each time it's pre-empted, goes to back of next lower priority queue;
  - dispatch first process in highest occupied queue.

  This tends to starve long jobs, esp. in interactive context. Possible solutions:
  - increase quantum for lower priority processes
  - raise priority for processes that are starved

Scheduling evaluation: Suggested Reading

In your favourite OS textbook, read the chapter on basic scheduling. Study the section(s) on evaluation of scheduling algorithms. Ensure that you understand the principles of queueing analysis and simulation modelling for evaluating scheduler algorithms.

Suitable material is in Stallings, ch 9 (esp. section 9.2) and Silberschatz et al. ch 6 (esp. section 6.6).

Questions to the newsgroup.

Multiprocessor Scheduling

Scheduling for SMP systems involves:

- assigning processes to processors
- deciding on multiprogramming on each processor
- actually dispatching processes

processes to CPUs: Do we assign processes to processors statically (on creation), or dynamically? If statically, may have idle CPUs; if dynamically, complexity of scheduling is increased – esp. in SMP, where kernel may be executing concurrently on several CPUs.

multiprogramming: Do we need to multiprogram on each CPU? 'Obviously, yes.' But if there are many CPUs, and the application is parallel at the thread level, may be better (for response time) not to.

SMP scheduling: Dispatching

For process scheduling, performance analysis and simulation indicate that the differences between scheduling algorithms are much reduced in a multi-processor system. There may be no need to use complex systems: FCFS, or slight variant, may suffice.

For thread scheduling, situation is more complex. SMP allows many threads within a process to run concurrently; but because these threads are typically interacting frequently (unlike different user processes), it turns out that performance is sensitive to scheduling. Four main approaches:

- load sharing: idle processor selects ready thread from whole pool
- gang scheduling: a *gang* of related threads are simultaneous dispatched to a set of CPUs
- dedicated CPUs: static assignment of threads (within program) to CPUs
- dynamic scheduling: involve the application in changing number of threads; OS shares CPUs among applications 'fairly'.

Load sharing is simplest and most like uniprocessing environment. As for process scheduling, FCFS works well. But it has disadvantages:

- the single pool of TCBs must be accessed with mutual exclusion – may be bottleneck, esp. on large systems
- preempted threads are unlikely to be rescheduled to same CPU; loses benefits of CPU cache (hence Linux, e.g., refines algorithm to try to keep threads on same CPU)
- program wanting all its threads running together is unlikely to get it – if threads are tightly coupled, could severely impact performance.

Most systems use load sharing, but with refinements or user-specifiable parameters to address some of the disadvantages. Gang scheduling or dedicated assignment may be used in special purpose (e.g. parallel numerical and scientific computation) systems.

Real-Time Scheduling

Real-time systems have deadlines. These may be *hard*: necessary for success of task, or *soft*: if not met, it's still worth running the task. Deadlines give RT systems particular requirements in:

- determinism: need to acknowledge events (e.g. interrupt) within predetermined time
- responsiveness: and take appropriate action quickly enough
- user control: hardness of deadlines and relative priorities is (almost always) a matter for the user, not the system
- reliability: systems must 'fail soft'. `panic()` is not an option! Better still, they shouldn't fail.

RTOSes typically do not handle deadlines as such. Instead, they try to respond quickly to tasks' demands. This may mean allowing preemption almost everywhere, even in small kernel routines.

Suggested reading: read the section on real-time scheduling in Stallings (section 10.2).

Exercise: how does Linux handle real-time scheduling?

Concurrency

When multiprogramming on a uniprocessor, processes are *interleaved* in execution, but concurrent in the abstract. On multiprocessor systems, processes really concurrent. This gives rise to many problems:

- resource control: if one resource, e.g. global variable, is accessed by two processes, what happens? Depends on order of executions.
- resource allocation: processes can acquire resources and block, stopping other processes.
- debugging: execution becomes non-deterministic (for all practical purposes).

Concurrency – example problem

Suppose a server, which spawns a thread for each request, keeps count of the number of bytes written in some global variable `bytecount`.

If two requests are served in parallel, they look like

serve request$_1$	serve request$_2$
`tmp1 = bytecount + thiscount1;`	`tmp2 = bytecount + thiscount2;`
`bytecount = tmp1;`	`bytecount = tmp2;`

Depending on the way in which threads are scheduled, `bytecount` may be increased by `thiscount1`, `thiscount2`, or (correct) `thiscount1 + thiscount2`.

Solution: control access to shared variable: protect each read–write sequence by a *lock* which ensures *mutual exclusion*. (Remember Java `synchronized`.)

Mutual Exclusion

Allow processes to identify *critical sections* where they have exclusive access to a resource. The following are requirements:

- mutual exclusion must be enforced!
- processes blocking in noncritical section must not interfere with others
- processes wishing to enter critical section must eventually be allowed to do so
- entry to critical section should not be delayed without cause
- there can be no assumptions about speed or number of processors

A requirement on clients, which may or may not be enforced, is:

- processes remain in their critical section for finite time

Implementing Mutual Exclusion

How do we do it?

- via hardware: special machine instructions
- via OS support: OS provides primitives via system call
- via software: entirely by user code

Of course, OS support needs internal hardware or software implementation. How do we do it in software?

We *assume* that mutual exclusion exists in hardware, so that memory access is atomic: only one read or write to a given memory location at a time. (True in almost all architectures.) (Exercise: is such an assumption necessary?)

We will now try to develop a solution for mutual exclusion of two processes, P_0 and P_1. (Let $\hat{i}$ mean $1 - i$.)

Exercise: is it (a) true, (b) obvious, that doing it for two processes is enough?

Mutex – first attempt

Suppose we have a global variable `turn`. We could say that when P_i wishes to enter critical section, it loops checking `turn`, and can proceed iff `turn` $= i$. When done, flips `turn`. In pseudocode:

```
while ( turn != i ) { }
/* critical section */
turn = î;
```

This has obvious problems:

- processes busy-wait
- the processes must take strict turns

although it does enforce mutex.

Mutex – second attempt

Need to keep state of each process, not just id of next process.

So have an array of two boolean flags, `flag[i]`, indicating whether P_i is in critical. Then P_i does:

```
while ( flag[î] ) { }
flag[i] = true;
/* critical section */
flag[i] = false;
```

This doesn't even enforce mutex: P_0 and P_1 might check each other's flag, then both set own flags to true and enter critical section.

Mutex – third attempt

Maybe set one's own flag before checking the other's?

```
flag[i] = true;
while ( flag[î] ) { }
/* critical section */
flag[i] = false;
```

This does enforce mutex. (Exercise: prove it.)

But now both processes can set flag to true, then loop for ever waiting for the other! This is *deadlock*.

Mutex – fourth attempt

Deadlock arose because processes insisted on entering critical section and busy-waited. So if other process's flag is set, let's clear our flag for a bit to allow it to proceed:

```
flag[i] = true;
while ( flag[î] ) {
  flag[i] = false;
  /* sleep for a bit */
  flag[i] = true;
}
/* critical section */
flag[i] = false;
```

OK, but now it is *possible* for the processes to run in exact synchrony and keep deferring to each other – *livelock*.

Mutex – Dekker's algorithm

Ensure that one process has priority, so will not defer; and give other process priority after performing own critical section.

```
flag[i] = true;
while ( flag[î] ) {
  if ( turn == î ) {
    flag[i] = false;
    while ( turn == î ) { }
    flag[i] = true;
  }
}
/* critical section */
turn = î;
flag[i] = false;
```

Optional Exercise: show this works. (If you have lots of time.)

Mutex – Peterson's algorithm

Peterson came up with a much simpler and more elegant (and generalizable) algorithm.

```
flag[i] = true;
turn = î;
while ( flag[î] && turn == î ) { }
/* critical section */
flag[i] = false;
```

Compulsory Exercise: show that this works. (Use textbooks if necessary.)

Mutual Exclusion: Using Hardware Support

On a uniprocessor, mutual exclusion can be achieved by preventing processes from being interrupted. So just disable interrupts! Technique used extensively inside many OSes. Forbidden to user programs for obvious reasons. Can't be used in long critical sections, or may lose interrupts.

This doesn't work in SMP systems. A number of SMP architectures provide special instructions. E.g. S/390 provides TEST AND SET, which reads a bit in memory and then sets it to 1, atomically as seen by other processors. This allows easy mutual exclusion: have shared variable `token`, then process grabs token using test-and-set.

```
while ( test-and-set(token) == 1 ) { }
/* critical section */
token = 0;
```

This is still busy-waiting. Deadlock is possible: low priority process grabs the token, then high priority process pre-empts and busy waits for ever.

Semaphores

Dijkstra provided the first general-purpose abstract technique for OS and programming language control of concurrency.

A *semaphore* is a special (integer) variable *s*, which can be accessed **only** by the following operations:

- `init(s,n)`: create the semaphore and initialize it to the non-negative value *n*.
- `wait(s)`: the semaphore value is decremented. If the value is now negative, the calling process is blocked.
- `signal(s)`: the semaphore is incremented. If the value is non-positive, one process blocked on `wait` is unblocked.

It is traditional, following Dijkstra, to use `P` (*proberen*) and `V` (*verhogen*) for `wait` and `signal`.

Types of semaphore

A semaphore is called *strong* if waiting processes are released FIFO; it is *weak* if no guarantee is made about the order of release. Strong semaphores are more useful and generally provided; henceforth, all semaphores are strong.

A *binary* or *boolean* semaphore takes only the values 0 and 1: `wait` decrements from 1 to 0, or blocks if already 0; `signal` unblocks, or increments from 0 to 1 if no blocked processes.

Recommended Exercise: Show how to use a private integer variable and two binary semaphores in order to implement a general semaphore. (Please think about this *before* looking up the answer!)

Implementing Semaphores

How do we implement a semaphore? Need an integer variable and queue of blocked processes, protected against concurrent access.

Use any of the mutex techniques discussed earlier. So what have we bought by implementing semaphores?

Answer: the mutex problem (and the associated busy-waiting) are confined inside just two (or three) system calls. User programs do not need to busy-wait; only the OS busy-waits, and only during the (short) implementation of semaphore operations.

Using Semaphores

A semaphore gives an easy solution to user level mutual exclusion, for any number of processes. Let s be a semaphore initialized to 1. Then each process just does:

```
wait(s);
/* critical section */
signal(s);
```

Exercise: what happens if s is initialized to m rather than 1?

The Producer–Consumer Problem

General problem occurring frequently in practice: a *producer* repeatedly puts items into a buffer, and a *consumer* takes them out. Problem: make this work, without delaying either party unnecessarily. (Note: can't just protect buffer with a mutex lock, since consumer needs to wait when buffer is empty.)

Can be solved using semaphores. Assume buffer is an unlimited queue. Declare two semaphores: `init(n,0)` (tracks number of items in buffer) and `init(s,1)` (used to lock the buffer).

Producer loop

```
datum = produce();
wait(s);
append(buffer,datum);
signal(s);
signal(n);
```

Consumer loop

```
wait(n);
wait(s);
datum = extract(buffer);
signal(s);
consume(datum);
```

Exercise: what happens if the consumer's `wait` operations are swapped?

Monitors

Because solutions using semaphores have `wait` and `signal` separated in the code, they are hard to understand and check.

A *monitor* is an 'object' which provides some *methods*, all protected by a blocking mutex lock, so only one process can be 'in the monitor' at a time. Monitor local variables are only accessible from monitor methods.

Monitor methods may call:

- `cwait(`*c*`)` where *c* is a *condition variable* confined to the monitor: the process is suspended, and the monitor released for another process.
- `csignal(`*c*`)`: some process suspended on *c* is released and takes the monitor.

Unlike semaphores, `csignal` does nothing if no process is waiting.

What's the point? The monitor enforces mutex; and all the synchronization is inside the monitor methods, where it's easier to find and check.

This version of monitors has some drawbacks; there are refinements which work better.

The Readers/Writers Problem

A common situation is to have a resource which may be *read* by many processes at once, but any read must block a write; and which can be written by only one process at once, blocking all other access.

This can be solved using semaphores. There are design decisions: do readers have priority? Or writers? Or do they all go into a common queue?

Suggested Reading: read about the problem in your OS textbook (e.g. Stallings 5.8).

Examples include:

- Unix file locks: many Unices provide read/write locking on files. See man fcntl on Linux.
- The OS/390 ENQ system call provides general purpose read/write locks.
- The Linux kernel uses 'read/write semaphores' internally. See `lib/rwsem-spinlock.c`.

Message Passing

Many systems provide *message passing* services. Processes may `send` and `receive` messages to and from each other.

`send` and `receive` may be blocking or non-blocking when there is no receiver waiting or no message to receive. Most usual is non-blocking `send` and blocking `receive`.

If message passing is *reliable*, it can be used for mutex and synchronization:

- simple mutex by using a single message as a *token*
- producer/consumer: producer sends data as messages to consumer; consumer sends null messages to producer to acknowledge consumption.

Message-passing is implemented using fundamental mutex techniques.

Deadlock

We have already seen deadlock. In general, *deadlock* is the *permanent* blocking of two (or more) processes in a situation where each holds a resource the other needs, but will not release it until after obtaining the other's resource:

Process P

```
acquire(A);
acquire(B);
release(A);
release(B);
```

Process Q

```
acquire(B);
acquire(A);
release(B);
release(A);
```

Some example situations are:

- `A` is a disk file, `B` is a tape drive.
- `A` is an I/O port, `B` is a memory page.

Another instance of deadlock is message passing where two processes are each waiting for the other to send a message.

Preventing Deadlock

Deadlock requires three facts about system policy to be true:

- resources are held by only one process at a time
- a resource can be held while waiting for another
- processes do not unwillingly lose resources

If any of these does not hold, deadlock does not happen. If they are true, deadlock may happen if

- a circular dependency arises between resource requests

The first three can to some extent be prevented from holding, but not practically so. However, the fourth can be prevented by ordering resources, and requiring processes to acquire resources in increasing order.

Avoiding Deadlock

A more refined approach is to deny resource requests that might lead to deadlock. This requires processes to declare in advance the maximum resource they might need. Then when a process does request a resource, analyse whether granting the request might result in deadlock.

How do we do the analysis? If we grant the request, is there sufficient resource to allow one process to run to completion? And when it finishes (and releases its resources), can we run another? And so on. If not, we should deny (block) the original request.

Suggested Reading: Look up *banker's algorithm*. (E.g. Stallings 6.3).

Deadlock Detection

Even if we don't use deadlock avoidance, similar techniques can be used to detect whether deadlock *currently exists*. What can we do then?

- kill all deadlocked processes (!)
- selectively kill deadlocked processes
- forcibly remove resources from some processes (what does the process do?)
- if checkpoint-restart is available, roll back to pre-deadlock point, and hope it doesn't happen next time (!)

Memory Management

The OS needs memory; the user program needs memory. In multiprogramming world, each user process needs memory. They each need memory for:

- code (instructions, text): the program itself
- static data: data compiled into the program
- dynamic data: heap, stack

Memory management is the problem of providing this. Key requirements:

- relocation: moving programs in memory
- allocation: assigning memory for processes
- protection: preventing access to other processes' memory...
- sharing: ...except when appropriate
- logical organization: how memory is seen by process
- physical organization: and how it is arranged in hardware

Relocation and Address Binding

When we load the contents of a static variable into a register, where is the variable in memory? When we branch, where do we branch to?

If programs are always loaded at same place, can determine this at compile time.

But in multiprogramming, can't predict where program will be loaded. So, compiler can tag all memory references, and make them relative to start of program. Then *relocating loader* loads program at location X, say, and adds X to all memory addresses in program. Expensive. And what if program is swapped out and brought back elsewhere?

Writing relocatable code

One way round: provide hardware instructions that access memory relative to a *base register*, and have programmer use these. Program loader then sets base register, but nothing else.

E.g. In S/390, typical instruction is

```
L R13,568(R12)
```

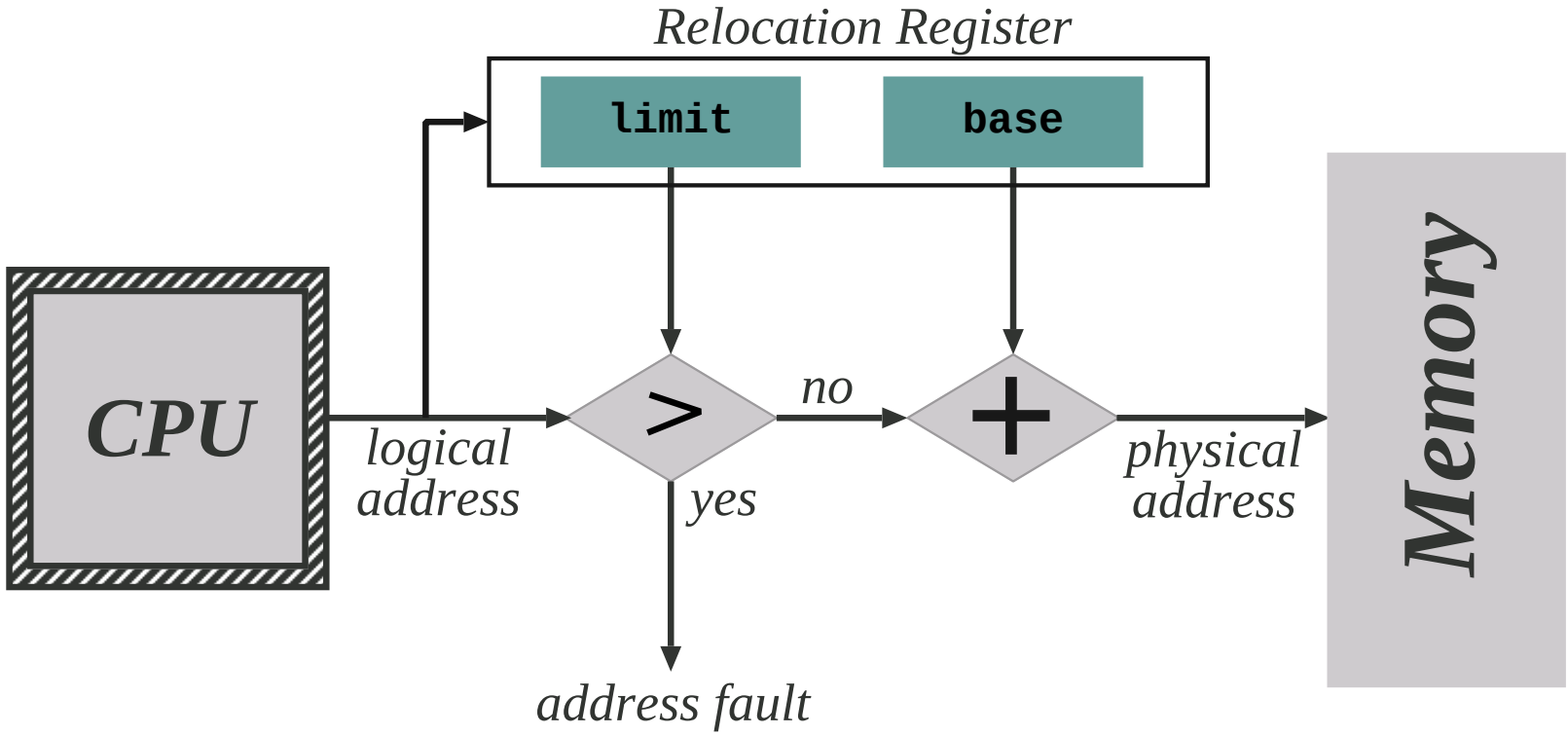
meaning 'load register 13 with value in address (contents of register 12 plus 568)'. Programmer (or assembler/compiler) makes all memory refs of this form; programmer or OS loads R12 with appropriate value.

This requires explicit programming: why not have hardware and OS do it?

Segmentation

A *segment* is a portion of memory starting at an address given in a *base register* B. The OS loads a value b into B. When program refers to memory address x, hardware transparently translates it to $x + b$.

To achieve protection, can add *limit register* L. OS loads L with length of segment l. Then if $x > l$, raise *address fault* (exception). (origin of Unix error message 'Segmentation fault'.)



Partitioning

Segmentation allows programs to be put into any available chunk of memory. How do we *partition* memory between various processes?

- fixed partitioning: divide memory into fixed chunks. Disadvantage: small process in large chunk is wasteful. Example: OS/MFT.
- dynamic partitioning: load process into suitable chunk; when exits, free chunk, maybe merge with neighbouring free chunks. Disadvantage: *(external) fragmentation* – memory tends to get split into small chunks. May need to *swap out* running process to make room for higher priority new process. How do we choose chunks?
  - first fit: choose first big enough chunk
  - next fit: choose first big enough chunk after last allocated chunk
  - best fit: choose chunk with least waste

  First fit is generally best: next fit fragments a bit more; best fit fragments a lot.

Partitioning – the Buddy System

Compromise between fixed and dynamic.

- Memory is maintained as a binary tree of blocks of sizes 2^k for $L \leq k \leq U$ suitable upper and lower bounds.
- When process of size s, $2^{i-1} < s \leq 2^i$, comes in, look for free block of size 2^i. If none, find (recursively) block of size 2^{i+1} and split it in two.
- When blocks are freed, merge free sibling nodes ('buddies') to re-create bigger blocks.

Variants on the buddy system are still used, e.g. in allocating memory within the Linux kernel. (I.e. memory for use by the kernel.)

Multiple Segments

Can extend segmentation to have multiple segments per program:

- hardware/OS provide different segments for different types of data, e.g. text (code), data (static data), stack (dynamic data). (How do you tell what sort of address is being used?)
- hardware/OS provides multiple segments at user request.
  - logical memory address viewed as pair (s, o)
  - process has *segment table*: look up entry s in table to get base and limit b_s, l_s
  - translate as normal to $o + b_s$ or raise fault if $o + b_s > l_s$

Exercise: look up how segmentation is done on the Intel x86 architecture.

Segmentation has some advantages:

- may correspond to user view of memory.
- importantly, protection can be done per segment: each segment can be protected against, e.g., read, write, execute.
- makes sharing of code/data easy. (But better to have a single list of segment descriptors, and have process segment tables point into that, than to duplicate information between processes.)

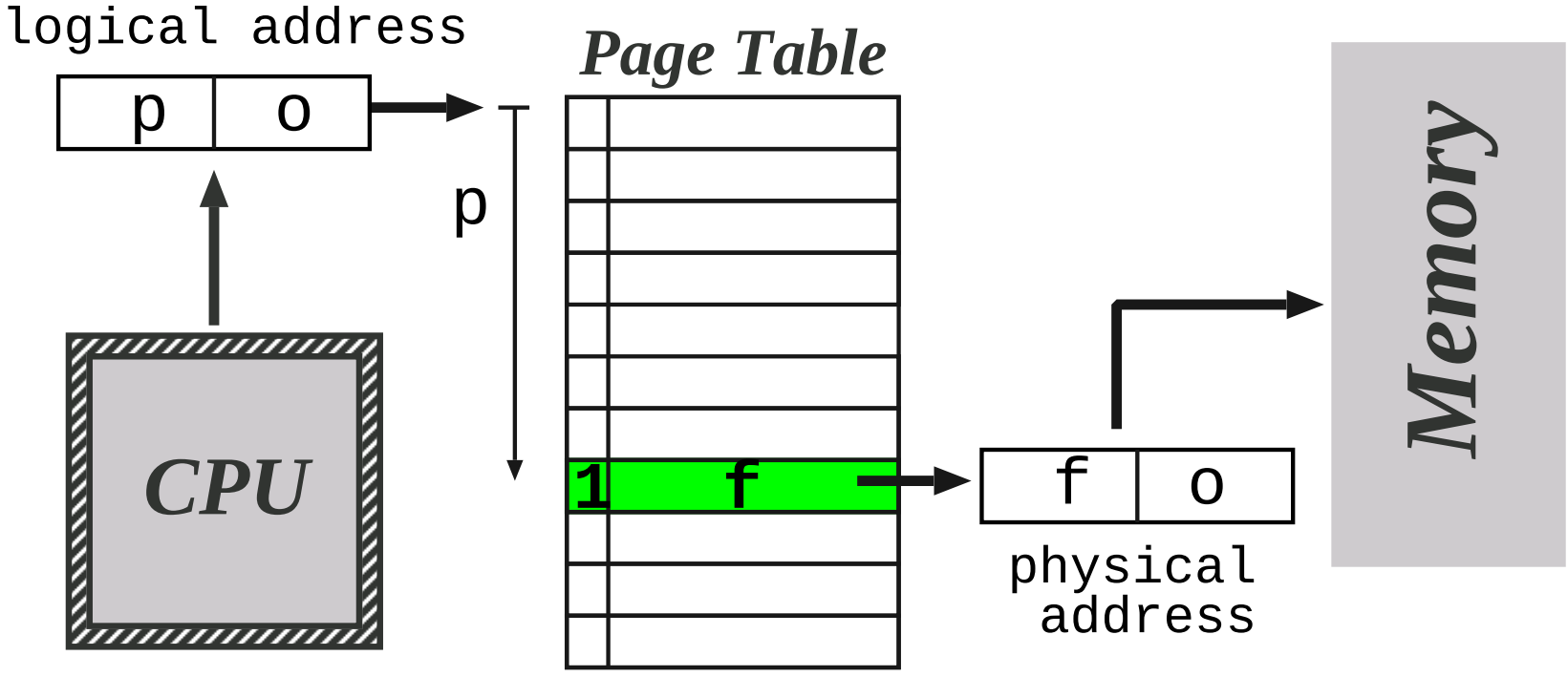
and some disadvantages:

- variable size segments leads to external fragmentation again;
- may need to *compact* memory to reduce fragmentation;
- small segments tend to minimize fragmentation, but annoy programmer.

Paging

Small segments reduce fragmentation; variable size segments introduce various problems. A special case would be to have many small fixed-size segments always provided – invisibly to the programmer. This is *paging*.

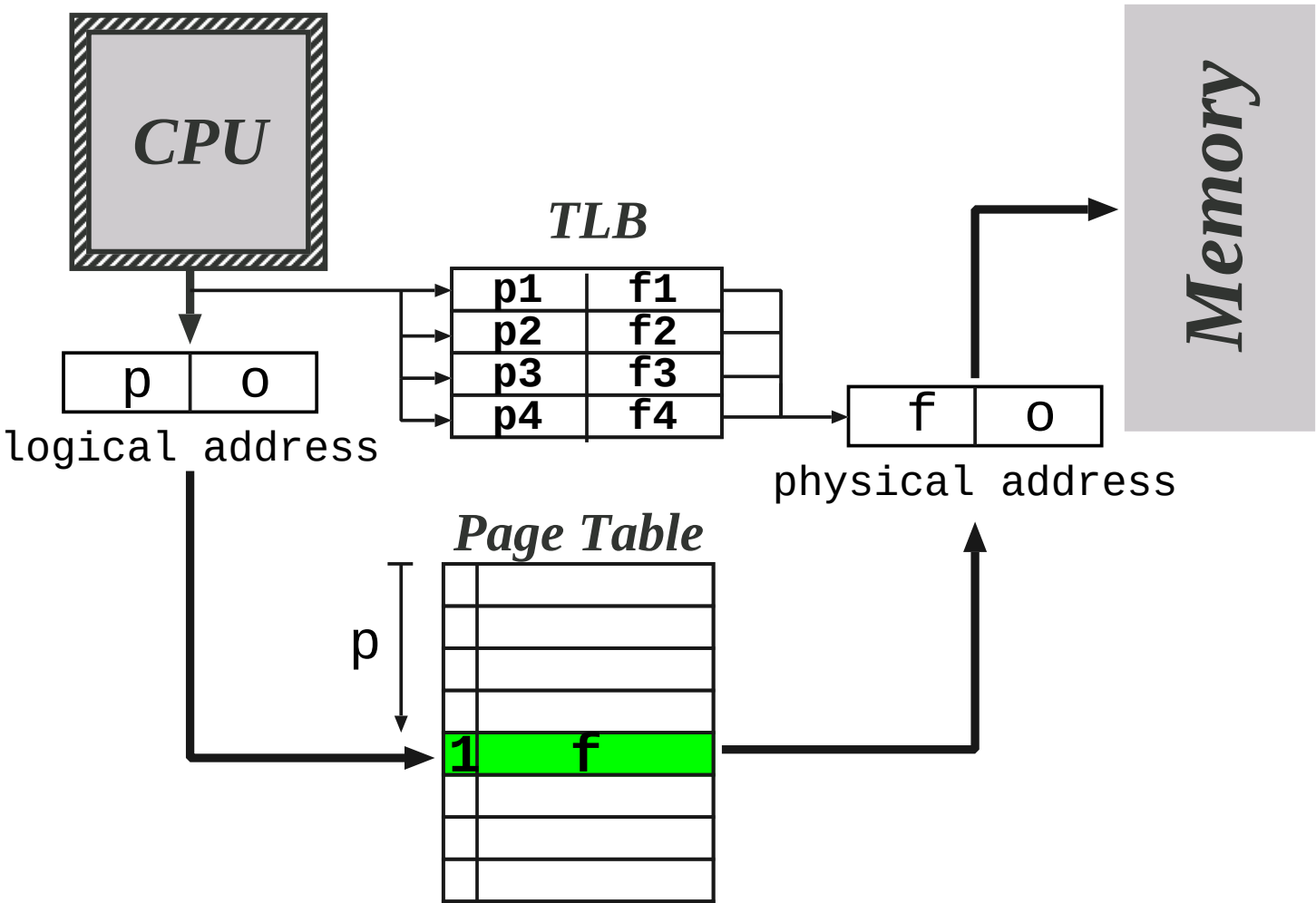
Virtual storage is divided in *pages* of fixed size (typically 4KB). Each page is mapped to a *frame* of real storage, by means of a *page table*.



Page table entry includes *valid bit*, since not all pages may have frames. Start and length of page table are held in control registers, as for segmentation. May also include protection via *protection bit(s)* in page table entry, e.g. read, write, execute, supervisor-mode-only, etc.

Translation Lookaside Buffer

With paging (or segmentation), each logical memory reference needs two (or more) physical memory references. A *translation lookaside buffer (TLB)* is a special cache for keeping recently used paging information. TLB is *associative* cache mapping page address directly to frame address.



Like all caches, the TLB introduces a coherency problem.

When the process context switches, active page table changes: must flush TLB.

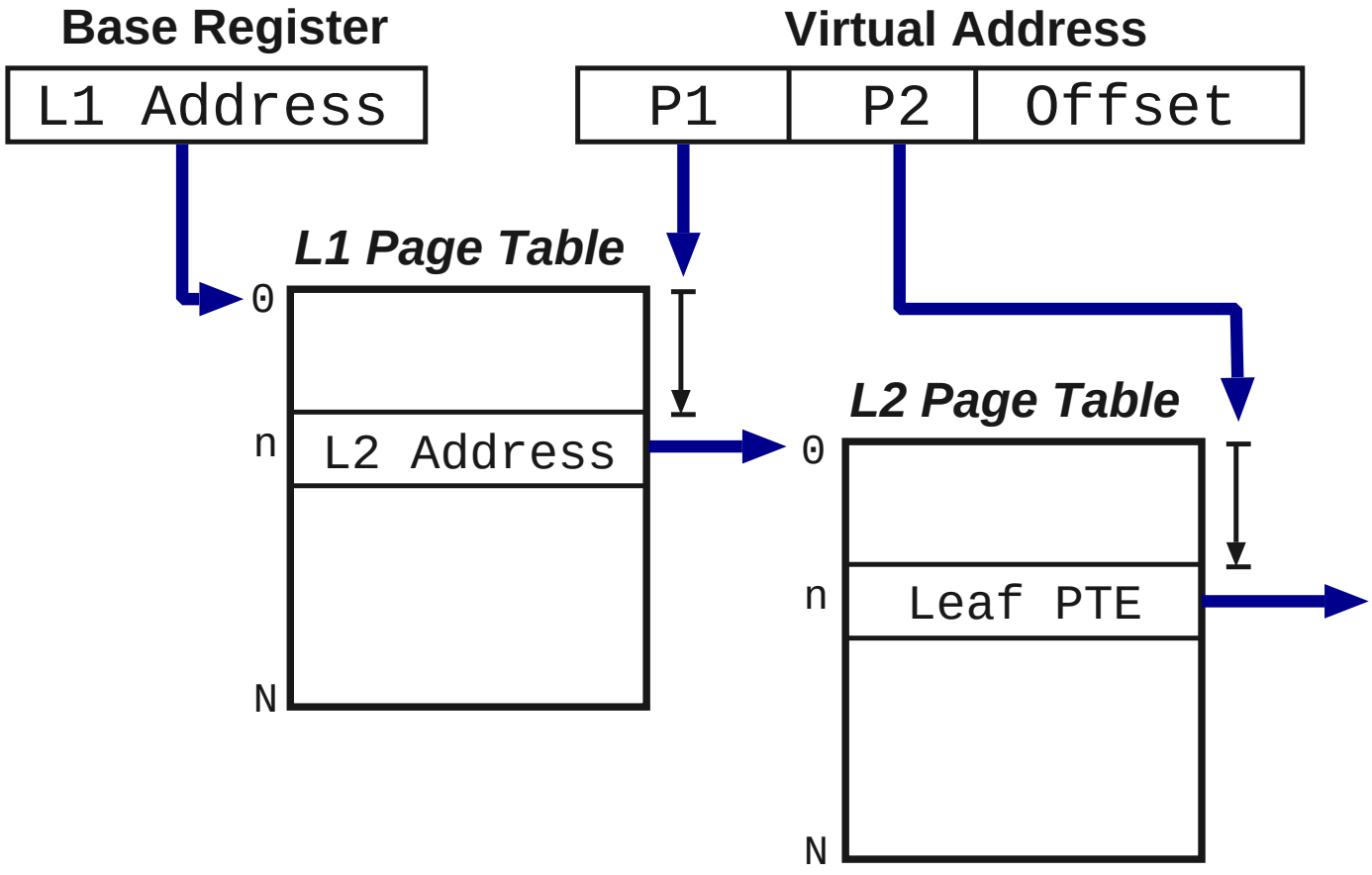
When page is freed, must invalidate entry in TLB.

Note that TLB also caches protection bits; changes in protection bits must invalidate TLB entry.

Multi-level Paging

Modern systems have address space of at least 2^{31} bytes, or 2^{19} 4K pages. That's a page table several megabytes long: one for each process...

Modern systems have two (or more) levels of page table:



Sharing Pages

Memory can be shared by having different pages map to the same frame.

For code, need *re-entrant* code: stateless, not self-modifying.

Otherwise, use *copy-on-write*:

- mark the pages read-only in each process (using protection bits in page table);
- when process writes, generates protection exception;
- OS handles exception by allocating new frame, copying shared page, and updating process's page table

Virtual Memory

Pages do not have to be in real memory all the time! We can store them on disk when not needed.

- initialize process's page table with invalid entries;
- on first reference to page, get exception: handle it, allocate frame, update page table entry;
- when real memory gets tight, choose some pages, write them to disk, invalidate them, and free the frames for use elsewhere;
- when process refers to page on disk, get exception; handle by reading in from disk (if necessary paging out some other page).

OS often uses frame-address portion of invalid page table entry to keep its location on disk.

Hardware support for VM usually includes:

- modified bit for page: no need to write out page if not changed since last read in;
- referenced bit or counter: unreferenced pages are first candidates for freeing.

Architectures differ where this happens:

- On Intel, modified and reference bits are part of page table entry.
- On S/390, they are part of *storage key* associated with each real frame.

Exercise: What, if any, difference does this make to the OS memory management routines?

Combined Paging and Segmentation: S/390

The concepts of paging and segmentation can be combined.

In S/390, they are intertwined, and can be seen as a 2-level paging system.

- Logical address is 31 bits:
- first 11 bits index into current segment table
- next 8 bits index into page table;
- remaining bits are offset.

Page tables can be paged out, by marking their entries invalid in the segment table.

For normal programming, there is only one segment table per process. Other segment tables (up to 16) are used by special purpose instructions for moving data between address spaces.

Combined Paging and Segmentation: Intel

Intel has full blown segmentation and independent paging.

- Logical address is 16-bit segment id and 32-bit offset.
- Segment id indexes into segment table; but
- segment id portion of logical address is found via a segment register;
- which is usually implicit in access type (`CS` register for instruction accesses, `DS` for data, `SS` for stack, `ES` for string data), but can be specified to be in any of six segment registers (there are exceptions).
- Segment registers are part of task context. (Task context stored in special system segments!)
- May be single global segment table; may also have task-specific segment tables.

The result of segment translation is 32-bit *linear address*.

Completely independently, the linear address goes through a two-level paging system.

- Segment related info (e.g. segment tables) can be paged out; so can second-level page tables.
- There is no link between pages and segments: segments need not lie on page boundaries.
- Pages can be 4KB, or 4MB.
- Page table register is part of task context, stored in task segment (!).

Paging Policies

In such a virtual memory system, the OS has to decide when to page in and out. What are the criteria?

- minimize number of page faults: avoid paging out pages that will be soon need
- minimize disk i/o: avoid reclaiming dirty (modified) pages

Fetch Policies

When should a page be brought back into main memory from disk?

- demand paging: when referenced. The locality principle suggests this should work well after an initial burst of activity.
- prepaging: try to bring in pages ahead of demand, exploiting characteristics of disks to improve efficiency.

Prepaging was not shown to be effective, and has been little, if at all, used.

Recently it has become a live issue again with a study suggesting it can now be useful.

http://www.cs.amherst.edu/~sfkaplan/research/prepaging/

Replacement Policy

When memory runs out, and a page is brought in, which page gets paged out?

Aim: page out the page with the longest time until its next reference. (This provably minimizes page faults.) In the absence of clairvoyance, we can try:

- LRU – least recently used: choose the page with longest time since last reference. This is almost optimal – but would have very high overhead, even if hardware supported it.
- FIFO – first in, first out: simple, but pages out heavily used pages. Performs poorly.
- clock policy: attempts to get some of the performance of LRU without the overhead. See next page.

Clock Replacement Policy

Makes use of the 'use' (accessed) bit provided by most hardware.

Put frames in a circular list $0, \ldots, n-1$. Have an index i. When looking for a page to replace, do:

```
increment i;
while (frame i used) {
  clear use bit on frame i;
  increment i; }
return i;
```

Hence doesn't choose page unless it has been unreferenced for one complete pass through storage. Clock algorithm performs reasonably well, about 25% worse than LRU.

Enhance to reduce I/O: scan only unmodified frames, without clearing use bit. If this fails, scan modified frames, clearing use bit. If this fails, repeat from beginning.

Page Caching

Many systems (including Linux) use a clock-like algorithm with the addition of caches or buffers:

- When a page is replaced, it's added to the end of the free page list if clear, or the modified page list if dirty.
- The actual frame used for the paged-in page is the *head* of the free page list.
- If no free pages, or when modified list gets beyond certain size, write out modified pages and move to free list.

This means that

- pages in the caches can be instantly restored if referenced again;
- I/O is batched, and therefore more efficient.

Linux allows you to tune various parameters of the paging caches. It also has a background kernel thread that handles actual I/O; this also 'trickles out' pages to keep a certain amount of memory free most of the time, to make allocation fast.

Resident Set Management

In the previous schemes, when a process page faults, some other process's page may be paged out. An alternative view is to manage independently the *resident set* of each process.

- allocate a certain number of frames to each process (on what criteria?)
- after a process reaches its allocation, if it page faults, choose some page of that process to reclaim
- re-evaluate resident set size (RSS) from time to time

How do we choose the RSS? The *working set* of a process over time Δ is the set of pages referenced in the last Δ time units. Aim to keep the working set in memory (for what Δ?).

Working sets tend to be stable for some time (locality), and change to a new stable set every so often ('interlocality transitions').

Actually tracking the working set is too expensive. Some approximations are

- page fault frequency: choose threshold frequency f. On page fault:
  - if (virtual) time since last fault is $< 1/f$, add one page to RSS; otherwise
  - discard unreferenced pages, and shrink RSS; clear use bits on other pages

  Works quite well, but poor performance in interlocality transitions
- variable-interval sampled working set: at intervals,
  - evaluate working set (clear use bits at start, check at end)
  - make this the initial resident set for next interval
  - add any faulted-in pages (i.e. shrink RS only between intervals)
  - the interval is every Q page faults (for some Q), subject to upper and lower virtual time bounds U and L.
  - Tune Q, U, L according to experience...

Input/Output

I/O is the messiest part of most operating systems.

- dealing with wildly disparate hardware
- with speeds from 10^2 to 10^9 bps
- and applications from human communication to data storage
- varying complexity of device interface (e.g. line printer vs disk)
- data transfer sizes from 1 byte to megabytes
- in many different representations and encodings
- and giving many idiosyncratic error conditions

Uniformity is almost impossible.

I/O Techniques

The techniques for I/O have evolved (and sometimes unevolved):

- direct control: CPU controls device by reading/writing data. lines directly
- polled I/O: CPU communicates with hardware via built-in controller; busy-waits for completion of commands.
- interrupt-driven I/O: CPU issues command to device, gets interrupt on completion
- direct memory access: CPU commands device, which transfers data directly to/from main memory (DMA controller may be separate module, or on device).
- I/O channels: device has specialized processor, interpreting special command set. CPU asks device to execute entire I/O program.

Terminology warning: Stallings uses 'programmed I/O' for 'polled I/O'; but the PIO (programmed I/O) modes of PC disk drives are (optionally but usually) interrupt-driven.

Programmed/Polled I/O

Device has *registers*, accessible via system bus. For output:

- CPU places data in *data* register
- CPU puts `write` command in *command* register
- CPU busy-waits reading *status* register until `ready` flag is set

Similarly for input, where CPU reads from data register.

Interrupt-driven I/O

Recall basic interrupt technique from earlier lecture.

Interrupt handler is usually split into a device-independent prologue (sometimes called the 'interrupt handler') and a device-dependent body (sometimes called the 'interrupt service routine'). Prologue saves context (if required), does any interrupt demuxing; body does device-specific work, e.g. acknowledge interrupt, read data, move it to user space.

ISRs need to run fast (so next interrupt can be handled), but may also need to do complex work; therefore often schedule non-urgent part to run later. (Linux 'bottom halves' (2.2 and before) or 'tasklets' (2.4), MVS 'service request blocks').

DMA

A DMA controller accesses memory via system bus, and devices via I/O bus. To use system bus, it *steals cycles*: takes mastery of the bus for a cycle, causing CPU to pause.

CPU communicates (as bus master) with DMA controller via usual bus technique: puts address of memory to be read/written on data lines, address of I/O device on address lines, read/write on command lines.

DMA controller handles transfer between memory and device; interrupts CPU when finished.

Note: DMA interacts with paging! Can't page out a page involved in DMA. Solutions: either lock page into memory, or copy to buffer in kernel memory and use that instead.

I/O Channels

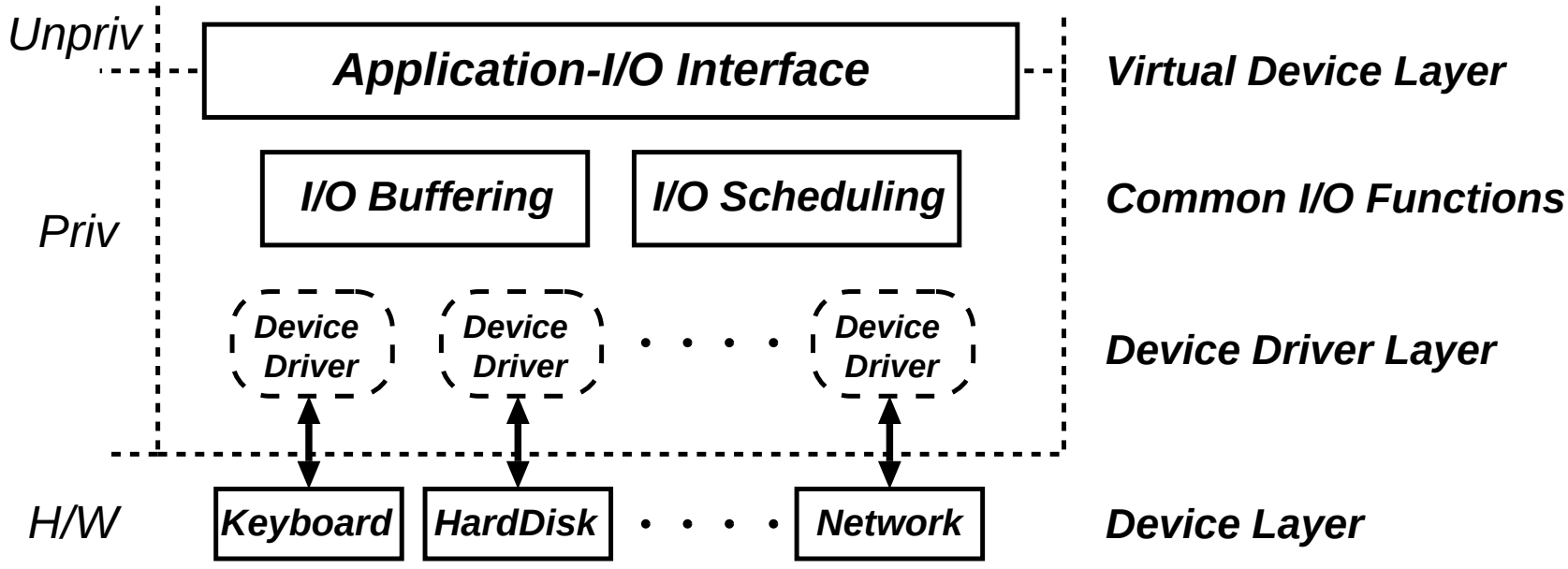
IBM mainframe peripherals have always had sophisticated controllers called 'channels'.

Operating system builds *channel program* (with commands including data transfer, conditionals and loops) in main memory, and issues `START SUBCHANNEL` instruction. Channel executes entire program before interrupting CPU.

Channels and devices are themselves organized into a complex communication network to achieve maximum performance.

IBM mainframe disk drives (DASD (direct access storage device) volumes) are themselves much more sophisticated than PC disks, with built-in facilities for structured (key, value) records and built-in searching on keys: designed particularly for database applications.

Taming I/O programming



So far as possible, confine device-specific code to small, low layer, and write higher-level code in terms of abstract device classes.

Many systems classify devices into broad classes:

- character: terminals, printers, keyboards, mice, ... typically transfer data byte at a time, don't store data.
- block: disk, CD-ROM, tape, ... transfer data in blocks (fixed or variable size), usually store data
- network: ethernet etc, tend to have mixed characteristics and need idiosyncratic control
- other: clocks etc.

Unix has the 'everything is a file' philosophy: devices appear as (special) files. If read/write makes sense, you (application programmer) can do it; device-specific functions available via `ioctl` system call on device file.

But somebody still has to write the device driver!

Disk Basics

Disks are the main storage medium, and their physical characteristics give rise to special considerations.

- A typical modern disk drive comprises several *platters*, each a thin disk coated with magnetic material.
- A comb of *heads* is on a movable arm, with one head per surface.
- If the heads stay still, they access circles on the spinning platters. One circle is called a *track*; the set of tracks is called a *cylinder*.
- Often, tracks are divided into fixed length *sectors*.

Consequently, to access data in a given sector, need to:

- move head assembly to right cylinder (around 4 ms on modern disks)
- wait for right sector to rotate beneath head (around 5 ms in modern disks)

Disk scheduling is the art of minimizing these delays.

Disk Scheduling

If I/O requests from all programs are executed as they arrive, can expect much time to be wasted in seek and rotation delays. Try to avoid this. How?

If we don't know the current disk position, all we can do is use expected properties of disk access. Because of locality, *LIFO* may actually work quite well. If we do know current position, can do intelligent scheduling:

- SSTF (shortest service time first): do request with shortest seek time.
- SCAN: move the head assembly from out to in and back again, servicing requests for each cylinder as it's reached. Avoids starvation; is harmed by locality.
- C-SCAN: scan in one direction only, then flip back. Avoids bias towards extreme tracks.
- FSCAN, N-step-SCAN: avoid long delays by servicing only a quota (N-step-SCAN) of requests per cylinder, or (FSCAN) only those requests arrived before start of current scan.

RAID

Disks are slow, and store critical information. *RAID* (redundant array of independent (orig. inexpensive) disks) is a suite of techniques for improving failure resistance and performance.

Basic idea is to view several ('small', cheap) physical disks as one logical volume. There are seven levels of RAID.

- Level 0: data are *striped* across n disks. Data is divided into strips; first n logical strips placed in first physical strip of each disk. Thus n consecutive logical strips can be read in parallel. Choice of strip size depends on application: high transfer rate (small strips $\longrightarrow$ high parallelism within one request), or high I/O request rate (large strips $\longrightarrow$ several different requests in parallel).
- Level 1: data are *mirrored* (duplicated) on each disk. Protects against disk failure; no overhead, instant recovery.

- Level 2: data are striped in small (byte or word) strips across some disks, with an error checksum (Hamming code) striped across other disks. Overkill; not used.
- Level 3: same, but using only parity bits, stored on other disk. If one disk fails, data can be read with on-the-fly parity computation; then failed disk can be regenerated. Has write overhead.
- Level 4: large data strips, as for level 0, with extra parity strip on other disk. Write overhead again, bottleneck on parity disk.
- Level 5: as level 4, but distribute parity strip across disks, avoiding bottleneck.
- Level 6: data striping across n disks, with two different checksums on two other disks (usually one simple parity check, one more sophisticated checksum). Designed for very high reliability requirements.

File Organization

Unix users are used to the idea of a file as an unstructured stream of bytes. This is not universally the case. Structural hierarchy is often provided at OS level:

- field: basic element of data. May be typed (string, integer, etc.). May be of fixed or variable length. Field name may be explicit, or implicit in position in record.
- record: collection of related fields, relating to one entity. May be of fixed or variable length, and have fixed or variable fields.
- file: collection of records forming a single object at OS and user level. (Usually) has a name, and is entered in directories or catalogues. Usual unit of access control.
- database: collection of related data, often in multiple files, satisfying certain design properties. Usually not at OS level. See database course.

Layers of Access to File Data

As usual, access is split into conceptual layers:

- device drivers: already covered
- physical I/O: reading/writing blocks on disk. Already covered.
- basic I/O system: connects file-oriented I/O to physical I/O. Scheduling, buffering etc. at this level.
- logical I/O: presents the application programmer with a (hopefully uniform) view of files and records.
- access methods: provide application programmer with routines for indexed etc. access to files.

File Organization

Within the file, how are records structured and accessed? May be concern of application program only (e.g. Unix), or built in to operating system (e.g. OS/390).

- byte stream: unstructured stream of bytes. Only native Unix type.
- pile: unstructured sequence of variable length records. Records and fields need to be self-identifying; can be searched only exhaustively.
- (fixed) sequential: sequence of fixed-length records. Can store only value of fields. One field may be *key*. Search is sequential; if records ordered by key, need not be exhaustive (but then problems in update).
- indexed sequential: add an *index file*, indexing key fields by position in main file, and *overflow file* for updates. Access much faster; update handled by adding to (sequential) overflow file. Every so often, merge overflow file with main file.
- indexed: drop the sequential ordering; use one exhaustive index plus auxilary indexes.
- hashed / direct: hash key value directly into offset within file. (Again, use overflow file for hash value clashes.)

Directories and Catalogues

How is a file found on disk? Usually have special files (in known location) listing other files with location.

Many systems have hierarchical *directories*:

- directories list files, including other directories.
- file is located by *path* through directory tree, e.g. `/group/teaching/cs3/os/Modules/worker.c`
- directory entry may contain *file metadata* (owner, permissions, access/mod times etc.), or this may be stored with file.
- (usually) directories can only be accessed via system calls, not by normal user I/O routines

Unix Files and Directories

In Unix:

- files are unstructured byte sequences
- metadata (including pointers to data) is stored in an *inode*
- directories link names to inodes (and that's all)
- hence file permissions are entirely unrelated to directory permissions
- inodes may be listed in multiple directories
- inodes (and file data) are automatically freed when no directory links to it
- the *root directory* of a filesystem is found in a fixed inode (number 2 in Linux filesystems).

OS/390 Data Set Organization

is complex. To simplify:

- files may have any of the formats mentioned above, and others
- files live on a disk *volume*, which has a *VTOC* giving names and some metadata (e.g. format) for files on the disk
- files from many volumes can be put in *catalogs*
- and a *filename prefix* can be associated with a catalog via the *master catalog*. E.g. the file `JCB.ASM.SOURCE` will be found in the catalog associated with `JCB`
- catalogs also contain additional metadata (security etc.), depending on files
- the master catalog is defined at system boot time from the VTOC of the system boot volume.

Access Control

Often files are shared between users. Access rights may be restricted. Types of access include

- knowledge of existence (e.g. seeing directory entry)
- execute (for programs)
- read
- write
- write append-only
- change access rights
- delete

Access Control Mechanisms

include

- predefined permission bits, e.g. Unix read/write/execute for owner/group/other users.
- access control lists giving specific rights to specific users or groups
- capabilities granted to users over files (see Computer Security)

Blocking

How are the *logical* records packed into *physical* blocks on disk? Depending on hardware, block size may be fixed, or variable. (PC/Unix disks are fixed; S/390 DASDs allow varying block sizes.)

- fixed blocking: pack constant number of fixed-length records into block
- variable, spanning: variable-length records, packed without regard to block boundaries. May need implicit or explicit continuation pointers when spanning blocks. Some records need two I/O operations.
- variable, non-spanning: records don't span blocks; just waste space at end of block

Choices are made on performance criteria.

File Space Allocation

How do we allocate physical blocks to files? This is very similar to the memory allocation problem, but with more options (since OS can manipulate complex data structures, unlike memory hardware).

- contiguous allocation: makes file I/O easy and quick. But: fragmentation; need for compaction. (If space is not pre-allocated (own problems!), may need dynamic compaction.
- chained allocation: allocate blocks as and when needed, and chain them together in one list per file. Easy, no fragmentation problems, but file may be scattered over disk, $\rightarrow$ v. inefficient I/O; direct access is slow.
- indexed allocation: file has *index* of blocks or sequences of blocks allocated to it. E.g. file `foo` is on blocks 3,4,5,78,79,80. Most popular method; direct access and sequential access; avoids fragmentation, has some contiguity.

Similar approaches to organizing *free space* on disk, though many systems just use *bitmap* of block allocation.

The Windows NT family

The successor to the old Windows 9? family. Started after (failure of) OS/2.

- Started 1989. New codebase; microkernel based architecture.
- NT3.1 released 1993; poor quality.
- NT3.5 released 1994 (3.51 in 1995); more or less usable.
- NT4.0 released 1996; matched W95 look'n'feel. For performance, some functions (esp. graphics) put back into kernel.
- Windows 2000 (NT 5.0); adds features for distributed processing; Active Directory (distributed directory service).
- Windows XP: no really significant OS-side changes. Terminal servers allow multiple users on one workstation (cf. Linux virtual consoles).
- Windows Vista: still NT, but many components extensively re-worked. Interesting techniques include machine-learning based paging. Many security techniques, including the Trusted Platform Module.

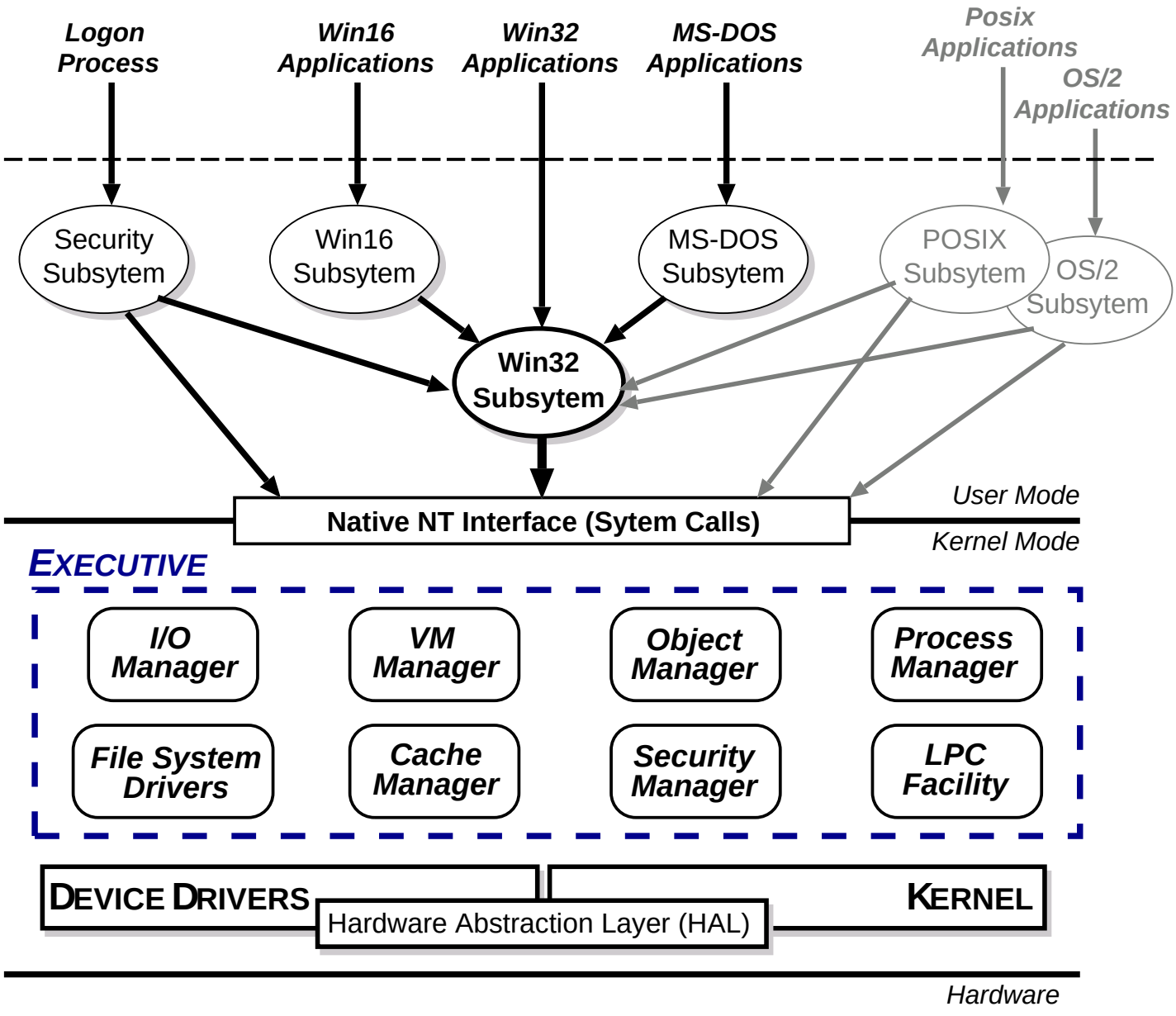
NT Design Principles

Design goals include:

- portability – not just Intel;
- security – for commercial and military use;
- POSIX compliance – to 'ease transition from Unix'...
- SMP support;
- extensibility;
- internationalization and localization;
- backwards compatibility (to Windows 9?, 3.1, even MS-DOS)

Accordingly, NT is micro-kernel based, modular/layered, and written in high-level language. (C and C++).

NT Family General Structure



HAL, Kernel, Executive, Subsystems

The *Hardware Abstraction Layer* converts all hardware-specific details into an abstract interface.

The *(micro)kernel* handles process and thread scheduling, interrupt/exception handling, SMP synchronization, and recovery. It is object-oriented. It is non-pageable and non-preemptible.

The *executive* comprises a number of modules, running in kernel mode but in thread context (cf. Linux kernel threads).

Subsystems are user-mode processes providing native NT facilities to other operating system APIs: Win32, POSIX, OS/2, Win3.1, MS-DOS.

Processes and Threads

NT has processes that own resources, including an address space, and threads that are the dispatchable units, as in the general description earlier in the course.

Process/thread services are provided by the executive's *process manager*. General purpose; no built-in restrictions to parent/child relationships.

Scheduling is priority-based; processes returning from I/O get boosted, and then decay each quantum. Special hacks for GUI response.

Quantum is around 20ms on Workstation, or double/triple for 'foreground task'. Longer quantum on server configurations.

Memory Management

NT has paged virtual memory. Page reclamation is local to process; resident set size is managed according to global demand.

The executive *virtual memory manager* provides VM services to processes, such as sharing and memory-mapped files.

In Vista, the system tries to pre-load pages that are likely to be needed – it even tracks application usage by time of day in order to load them before they're called.

Object Management

Almost everything is an *object*. The executive *object manager* provides object services:

- hierarchical namespace for named objects (via directory objects);
- access control lists
- naming domains – mapping existing namespaces to object namespace
- symbolic links
- *handles* to objects (used by processes etc.)

I/O Management

The executive *I/O manager* supervises dispatching of basic I/O to device drivers.

- asynchronous request/response model – application queues request, is signalled on completion
- device drivers and file system drivers can be stacked (cf. Solaris streams)
- cache manager provides general caching services
- network drivers include distributed system support (W2K and XP)

NT supports the old FAT-32 filesystem, but also the modern NT filesystem.

- an NTFS *volume* occupies a partition, a disk, or multiple disks
- an NTFS *file* is structured: a file has *attributes*, including (possibly multiple) data attributes and security attributes
- files located via *MFT (master file table)*
- NTFS is a *journalling* file system

OS/390 – MVS

See earlier for evolution. MVS is the basic operating system component of OS/390. MVS comprises BCP (Basic Control Program) and JES2 (Job Entry Subsystem).

MVS design objectives (1972–date!):

- performance
- reliability
- availability
- compatibility

in the large system environment.

A large multiprocessor MVS cluster is resilient. The system recovers from, and reconfigures round, almost all failures, including hardware failures. (99.999% uptime is claimed; some installations are said to have stayed up for more than a decade.)

MVS Components

- supervisor: main OS functions
- Master Scheduler: system start and control, communication with operator; master task in system
- Job Entry Subsystem (JES2): entry of batch jobs, handling of output
- System Management Facility (SMF): accounting, performance analysis
- Resource Measurement Facility: records data about system events and usage, for use by SMF and others
- Workload Manager: manages workload according to installation goals
- Timesharing Option (TSO/E): provides interactive timesharing services
- TCAM, VTAM, TCP/IP: telecoms and networking
- Global Resource Serialization: resource control across clusters

and many others

Supervisor

- Dispatcher: main scheduler (in OS sense)
- Real Storage Manager: manages real memory, decides on page in/out/reclaim, determines resident set sizes etc.
- Auxiliary Storage Manager: handles page/swap in/out
- Virtual Storage Manager: address space management and virtual allocation (calls RSM to get real memory)
- System Resources Manager: supervisor component of Workload Manager: advises/instructs above components

Job Entry Subsystem

handles processing of batch jobs:

- read from 'card reader' or TSO `SUBMIT` command
- convert JCL to internal form
- start execution
- spool output; hold for later inspection, and/or print

JES2 is a basic system; JES3 provides more advanced job scheduling facilities.

Address spaces

A virtual address space includes:

- nucleus: important control blocks (in ptic CVT); most OS routines
- Fixed Link Pack Area: non-pageable (e.g. for performance) shared libraries etc.
- private area: includes address-space-local system data, and user programs and data
- Common Service Area: data shared between all tasks
- System Queue Area: page tables etc.
- Pageable Link Pack Area: shared libraries permanently resident in virtual memory

Note that (unlike Linux) most system data is mapped into all address spaces.

Address spaces are created for each *started task* (operator `START` command), job, and TSO user.

There are also *data spaces*: extra address spaces solely for user data.

Recap of other aspects

We have already described many aspects of S/390 and OS/390. To recap:

- Memory is paged on demand, with a two-level paging system.
- The task is the basic dispatchable unit. One address space may have many tasks. The service request is a small unit, dispatchable to any address space
- There is a general resource control mechanism `ENQ/DEQ`.
- SMP is supported in hardware.
- I/O is highly sophisticated, offloaded from CPU. Throughput and transaction rates can be very high. (E.g. 10 000s of concurrent users accessing terabyte databases; sustained I/O of tens of MB/s on a single CPU.) Intra-cluster data transfer at GB/s.

VM – Virtual Machine Operating System

The S/390 hardware is easy to virtualize. VM is an S/390 OS exploiting this.

- VM provides each user with a (configurable) virtual S/390 machine to which they have full access.
- The VM CP (control program) gives each user a virtual console with operator-like commands, at which
- they may start CMS (Conversational Monitor System), a single-user operating system for interactive work;
- or they may load a S/390 operating system: MVS, Linux/390, or even VM.
- VM is a fully paging virtual memory OS, so
- IBM OSes can be adjusted to allow communication with VM *hypervisor*, so only one OS does the paging etc. (maybe VM, maybe guest)

A single S/390 machine can reasonably host 10–20 thousand virtual machines running CMS. Some production shops run 3000 virtual Linux machines on one S/390 under VM.

Security Overview

Security requirements are sometimes divided into categories:

- Confidentiality: data (or even its existence) should protected from disclosure to unauthorized entities.
- Integrity: data should not be modified by unauthorized entities.
- Availability: data should be available to authorized entities
- Authenticity: all entities should be identified, so that all operations are attributable. Also, nowadays,
- Non-repudiation: no entity should be able to deny doing any action that it did do.

Attacks

Many attacks are obvious: cut power lines, intercept phone lines, etc.

Some are less obvious: *traffic analysis* may reveal critical information. *Data aggregation* may extract sensitive information from several apparently innocent sources.

Social engineering attacks often work.

Protection

Different degrees and granularities of protection can be provided, with increasing difficulty, by operating sytems.

- no protection: fine, if the system is contained by physical security.
- isolation of tasks: different tasks have separate address spaces, filespaces, etc., with no communication.
- public/private: allow object owners to make them public (accessible to other processes) or private.
- sharing via access lists: OS enforces user-specified access restrictions given in ACLs
- . . . via capabilities: or with dynamically created access capabilities, which may
- limit uses: constrain detailed use: printing, viewing, copying etc.

Techniques

- User identification
  - Passwords
  - One-time passwords
  - Biometrics
- Confidentiality
  - OS facilities
  - Encryption
- Authenticity and non-repudiation
  - Cryptographic signing

Malicious Software

Two main entry routes for malicious software:

- exploiting bugs in system software, e.g. buffer overflow attacks
- exploiting users, e.g. most email viruses

Malicious software includes worms, viruses, logic bombs, trojan horses, trap doors. Prevention via:

- rigorous access control on need-to-know basis
- reviews of potentially exploitable code
- user education

Detection via

- signature scanning
- sandboxed execution
- performance and system behaviour analysis

Cracking and counter-cracking

The ultimate desideratum of a cracker is complete privileged access to a system. Attacks may involve several levels of indirection:

- break directly into a privileged network server, suborn an operator, etc.
- break a user account, then exploit weakness in OS to get root
- break into a trusted but more vulnerable machine, use as relay

Cracking user accounts

Nowadays, by far the easiest way is by tricking the user into opening an executable attachment or running a download. With foolishly designed mail systems, you may not even need to trick the user. Counter-measures, in increasing order of severity:

- educate users
- scan mail for known viruses
- modify local mail programs etc. to stop them executing attachments
- prohibit (by modifying OS if necessary) execution of any program not digitally signed or otherwise known to be trusted

More traditional techniques:

- guessing (or brute force searching) passwords.
  Counter: draconian password policies (problems with this?).
- faking login screens (this would be very easy on DICE).
  Counter: train users to use "secure attention" before log on, if available.

Trapdoors and Backdoors

The cracker (either as an insider, or by cracking another machine) inserts a trapdoor into a privileged program, allowing root access or whatever.
Classic paper by Ken Thompson 1984:

- modify the C compiler (`cc`) so that it recognizes when it is compiling the `login` program: it then inserts a routine to allow a master password for any account;
- and if it recognizes it is compiling itself, it inserts these two routines.

Moral: you can't trust a system built with somebody else's tools, even if the source is clean.
Main counter: well trusted sources. But ...

In autumn 2003, somebody attempted to insert a trapdoor into the master copy of the Linux kernel. They added what looked like an obscure test with a typo to a system call, which would actually allow anybody to become root by passing appropriate arguments. Fortunately,

- It was caught by clash detection in version control.
- The source tree they modified was not actually the master (although is a source used by many people).

Attacking Servers and Services

Many Unix machines run (or ran) servers such as FTP, Web, etc. Many of these must provide services on behalf of all users; they therefore [!] run as root. Windows machines similarly run several services in privileged modes.

Buffer overflow vulnerabilities in privileged servers give direct entry for the cracker. Numerous examples, on both Unix and Windows.

Don't run unnecessary servers; use firewall to block access to all except trusted servers.

The Internet Worm of 3 November 1988

Robert Morris, Jr., released the first worm that seriously disrupted the Internet. It used many techniques. For attack:

- buffer overflow problem in the Unix `finger` service
- exploiting intentional 'trapdoor' in Unix mail servers compiled in debugging mode – very many production servers were!
- password guessing
- Unix remote execution allowed easy spread

It also had defensive capabilities:

- changes its name to `sh`
- `forks()` to change process id frequently
- avoids leaving files around
- obfuscates data in memory to hinder analysis

Breaking root

Once the cracker has user access, the next step is to become superuser, get admin status, etc. Usual vulnerabilities are easier to exploit when already on the system. May also be other possibilities, for example:

- MVS has a notion of authorized program which can do privileged things
- one MVS system had a home-grown user command processor (= shell) which ran authorized (so it could invoke authorized programs)
- and which ran in storage key 8 (normal user storage key) . . .
- MVS I/O standard access methods allow user to intercept READ calls on open files, so
- user could install read hook on files being used by shell

Intrusion detection, concealment and rootkits

Modern systems do a lot of monitoring to try to detect suspicious activity: changed files, unusual processes. Therefore, after successful crack, need to avoid detection. Often install modified copies of `ls`, `ps` etc.
Modern Linux rootkits try even harder: install kernel modules which modify kernel code of system calls so that certain processes and files are ignored - even clean `ls` or `ps` will not show them. Also modify load average to ignore your password cracker, etc. etc.

Multilevel Security

Military and governmental security desires have driven much work on secure operating systems.

Multilevel security is the concept of processing information with differing degrees of sensitivity on the same system. E.g. British official scheme is unclassified, restricted, confidential, secret, top secret in a hierarchy, refined by *codewords* restricting material to specified users. (E.g. the famous 'top/most secret ultra' was the 'top/most secret' classification, with the 'ultra' codeword identifying the Enigma decrypts.) A user cleared to one level should not see higher level material.

Applications also in commerce, banking, comms (telephone billing system should read but not write switching data).

The Bell–LaPadula Security Policy Model

A security policy model describes concisely and accurately what constraints security places on information flow.

The BLP model is two basic principles:

- No Read Up (simple security property): a process running at one level may not read data at a higher level;
- No Write Down (*-property): a process running at one level may not write data at a lower level.

The second policy prevents viruses etc. from copying sensitive information down to a lower security level, as well as stopping humans accidentally doing so.

(How is declassification achieved?)

Maintaining the Security Policy

The design principle is to mediate all data transfers through a small, verifiable OS component, the *reference monitor*. This, combined with the hardware it uses (and the human), forms the *Trusted Computing Base (TCB)*. Application programs do not have to be checked!

Special architectures are required for highest security, but this is now too expensive/inconvenient. Most current 'secure' operating systems run on commodity hardware, and are 'hardened' versions of commodity OSes.

Evaluation Criteria

The U.S. Department of Defense designed the hugely influential *Orange Book* criteria for evaluating the security of systems. Classification scheme A1, B3, B2, B1, C2, C1, D. Only one general purpose computer (Honeywell SCOMP: purpose-built hardware and OS) ever achieved an A1 rating, which required formal verification. Various Unices and MVS reached B2 (structured mandatory access controls, etc. etc.).

Now superseded by the *Common Criteria* based on Orange Book, British and European criteria.

The Common Criteria for Security Assurance Evaluation

0. Inadequate Assurance
1. Functionally Tested. Provides analysis of the security functions, using a functional and interface specification of the TOE, to understand the security behaviour. The analysis is supported by independent testing of the security functions.
2. Structurally Tested. Analysis of the security functions using a functional and interface specification and the high level design of the subsystems of the TOE. Independent testing of the security functions, evidence of developer "black box" testing, and evidence of a development search for obvious vulnerabilities.

3 Methodically Tested and Checked. The analysis is supported by "grey box" testing, selective independent confirmation of the developer test results, and evidence of a developer search for obvious vulnerablitities. Development environment controls and TOE configuration management are also required.

4 Methodically Designed, Tested and Reviewed. Analysis is supported by the low-level design of the modules of the TOE, and a subset of the implementation. Testing is supported by an independent search for obvious vulnerabilities. Development controls are supported by a life-cycle model, identification of tools, and automated configuration management.

5 Semiformally Designed and Tested. Analysis includes all of the implementation. Assurance is supplemented by a formal model and a semiformal presentation of the functional specification and high level design, and a semiformal demonstration of correspondence. The search for vulnerabilities must ensure relative resistance to penetration attack. Covert channel analysis and modular design are also required.

6 Semiformally Verified Design and Tested. Analysis is supported by a modular and layered approach to design, and a structured presentation of the implementation. The independent search for vulnerabilities must ensure high resistance to penetration attack. The search for covert channels must be systematic. Development environment and configuration management controls are further strengthened.

7 Formally Verified Design and Tested. The formal model is supplemented by a formal presentation of the functional specification and high level design showing correspondence. Evidence of developer "white box" testing and complete independent confirmation of developer test results are required. Complexity of the design must be minimised.

Single Level Security – the Easy Way

Many workers with security clearances want to use 'standard' computers (i.e. Windows) in a simple (non-MLS) way. How is data protected? Usually by a software add-on that maintains the hard drive in an encrypted state.

The U.K.'s CESG provides a Windows product (Kilgetty) which combines a dongle with a password to maintain encryption. It is considered adequate to protect two levels: e.g. a PC with Kilgetty used for Top Secret material can be treated as merely Confidential when it's switched off. (This is actually a rather strong statement of confidence in the encryption.)

Of course, the native classification is the highest level ever used since installation.

Steganography – Hiding the Existence of Data

In many contexts, even revealing the existence of sensitive data may be dangerous – whether because an enemy army may be able to draw inferences, or because you're a political activist under investigation by the security police. (In several countries, mere use of crypto is a crime; in the U.K., you can be (legally) forced to decrypt any encrypted material.)

Steganography is the science of hiding data.

The simplest (don't use it) trick is to hide data in an image, by using the least significant bit of each pixel to carry the data.

Stegfs: a Steganographic File System for Linux

Theory by Anderson, Needham, Shamir; implementation by McDonald and Kuhn.

Stegfs provides a crypto-based secure filesystem with multiple levels of security. If you have only the level 3 password (say), you can't tell that there is a level 4, let alone that there is any level 4 data.

If the machine is running at level 3, the OS doesn't know about level 4 data, so it may write over it...

...so StegFS maintains several copies of data in dispersed blocks, in the hope that one of them will survive until you next enter the relevant level - every so often, you should enter the highest security level and run a maintenance procedure to regenerate the multiple copies.

Some current topics

Recent OS research and development tends to centre around distributed systems and large scale networks, but not exclusively.

What follows is a personal selection of some recent topics presented at the Workshop on Hot Topics in Operating Systems in 2005.

Are Virtual Machine Monitors Microkernels Done Right?

by Steven Hand, Andrew Warfield, Keir Fraser, Evangelos Kotsovinos (Cambridge), Dan Magenheimer (HP Labs)

This paper pours some oil on the monolithic/microkernel flames by arguing that virtual machine monitors, because they need to achieve both strict isolation of guest machines, and good performance, provide most of the good things about microkernels while eliminating some of the bad things.

The claimed bad things include: allowing user space activity to mess up the system (e.g. paging); needing lots of IPC.

The good things include extensibility, security and manageability.

Stupid File Systems Are Better

by Lex Stein (Harvard)

These days, storage is often virtualized (RAID, logical volumes, etc.). We also have fancy filesystems that try to optimize the layout of files on disk (recall lecture). This paper does some experiments and suggests that the fancy layout is a waste of time in a virtualized world – a random layout is often as fast, and is much more consistent as disks are moved between different virtualization strategies.

patch (1) Considered Harmful

by Marc E. Fiuczynski (Princeton); Robert Grimm (NYU); Yvonne Coady (Victoria); David Walker (Princeton)

Patches often patch many different files; and patch works on uninterpreted lines of text. This makes patches effectively unmaintainable except with a lot of hand work. This paper proposes a new style of patch language, which knows about the semantic structure of the code.

Treating Bugs As Allergies: A Safe Method for Surviving Software Failures

by Feng Qin, Joseph Tucek and Yuanyuan Zhou (UIUC)

The idea here is, when you hit a bug, roll the program back to a checkpoint, then make some modification to the execution environment (e.g. tweak the memory allocator to allocate somewhere else), and try again. Firstly, changing the environment may prevent the bug from manifesting; secondly, it should provide additional diagnostics.

The question is, what to change...

Singularity

by lots of people at Microsoft Research

"What would an OS look like if it were designed from the ground up for dependability?"

Singularity uses strongly typed safe languages (no more C!) to ensure that programs satisfy its requirements. It has software isolated processes – instead of memory protection (the whole system runs in supervisor mode in a single address space!), language safety and a trusted compiler ensure that no process can access data belonging to another process or the kernel.

All communication is by message passing along typed channels.

Dynamic loading of code is not possible – extensions must run in new SIPs. But SIPs are cheap to create compared to traditional processes.