Operating Systems I

Steven Hand

Michaelmas / Lent Term 2005/06

16 lectures for CST IA

Handout 1 of 2

Course Aims

This course aims to:

- provide you with a general understanding of how a computer works,
- explain the structure and functions of an operating system,
- illustrate key operating system aspects by concrete example, and
- prepare you for future courses. . .

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

- describe the fetch-execute cycle of a computer
- understand the different types of information which may be stored within a computer memory
- compare and contrast CPU scheduling algorithms
- explain the following: process, address space, file.
- distinguish paged and segmented virtual memory.
- discuss the relative merits of Unix and NT. . .

Course Outline

- Part I: Computer Organisation
  - Computer Foundations
  - Operation of a Simple Computer.
  - Input/Output.
- Part II: Operating System Functions.
  - Introduction to Operating Systems.
  - Processes & Scheduling.
  - Memory Management.
  - I/O & Device Management.
  - Protection.
  - Filing Systems.
- Part III: Case Studies.
  - Unix.
  - Windows NT.

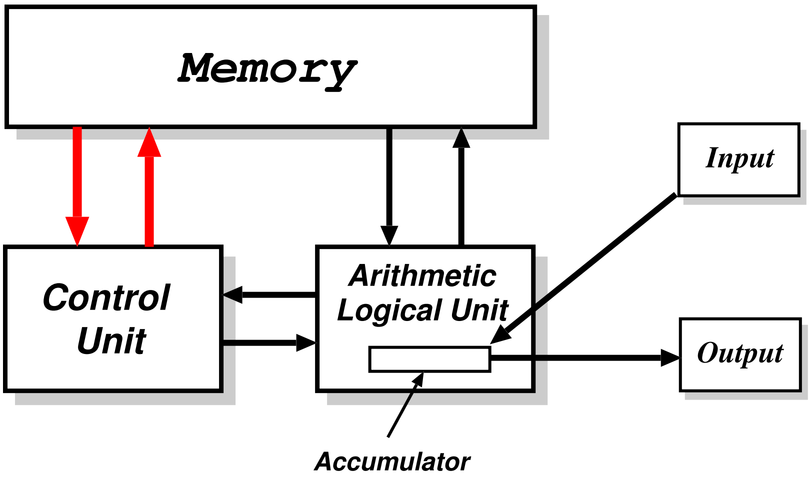
Recommended Reading

- Tannenbaum A S
  Structured Computer Organization (3rd Ed)
  Prentice-Hall 1990.
- Patterson D and Hennessy J
  Computer Organization & Design (2rd Ed)
  Morgan Kaufmann 1998.
- Bacon J [and Harris T]
  Concurrent Systems or *Operating Systems*
  Addison Wesley 1997, Addison Wesley 2003
- Silberschatz A, Peterson J and Galvin P
  Operating Systems Concepts (5th Ed.)
  Addison Wesley 1998.
- Leffler S J
  The Design and Implementation of the 4.3BSD UNIX Operating System.
  Addison Wesley 1989
- Solomon D and Russinovich M
  Inside Windows 2000 (3rd Ed) or *Windows Internals (4th Ed)*
  Microsoft Press 2000, Microsoft Press 2005

A Chronology of Early Computing

- (several BC): abacus used for counting
- 1614: logarithms disovered (John Napier)
- 1622: invention of the slide rule (Robert Bissaker)
- 1642: First mechanical digital calculator (Pascal)
- Charles Babbage (U. Cambridge) invents:
  - 1812: "Difference Engine"
  - 1833: "Analytical Engine"
- 1890: First electro-mechanical punched card data-processing machine (Hollerith, later IBM)
- 1905: Vacuum tube/triode invented (De Forest)
- 1935: the relay-based *IBM 601* reaches 1 MPS.
- 1939: *ABC* — first electronic digital computer (Atanasoff & Berry, Iowa State University)
- 1941: *Z3* — first programmable computer (Zuse)
- Jan 1943: the *Harvard Mark I* (Aiken)
- Dec 1943: *Colossus* built at 'Station X', Bletchley Park (Newman & Wynn-Williams, et al).

The Von Neumann Architecture



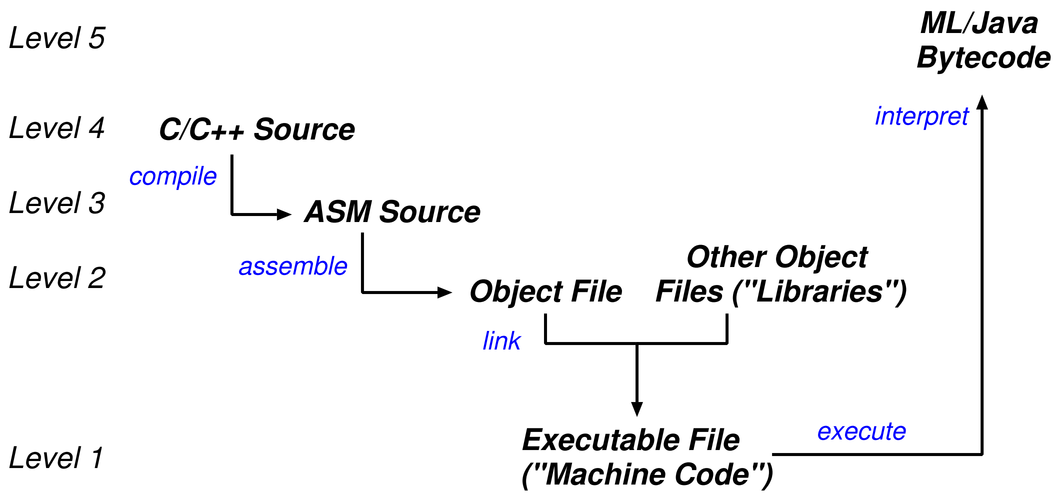
- 1945: *ENIAC* (Eckert & Mauchley, U. Penn):
  - 30 tons, 1000 square feet, 140 kW,
  - 18K vacuum tubes, 20×10-digit accumulators,
  - 100KHz, circa 300 MPS.
  - Used to calculate artillery firing tables.
  - (1946) blinking lights for the media. . .
- But: “programming” is via plugboard ⇒ v. slow.
- 1945: von Neumann drafts “EDVAC” report:
  - design for a stored-program machine
  - Eckert & Mauchley mistakenly unattributed

Further Progress. . .

- 1947: “point contact” transistor invented (Shockley, Bardeen & Brattain, Bell Labs)
- 1949: *EDSAC*, the world’s first stored-program computer (Wilkes & Wheeler, U. Cambridge)
  - 3K vacuum tubes, 300 square feet, 12 kW,
  - 500KHz, circa 650 IPS, 225 MPS.
  - 1024 17-bit words of memory in mercury ultrasonic delay lines.
  - 31 word “operating system” (!)
- 1954: *TRADIC*, first electronic computer without vacuum tubes (Bell Labs)
- 1954: first silicon (junction) transistor (TI)
- 1959: first integrated circuit (Kilby & Noyce, TI)
- 1964: IBM System/360, based on ICs.
- 1971: Intel 4004, first micro-processor (Ted Hoff):
  - 2300 transistors, 60 KIPS.
- 1978: Intel 8086/8088 (used in IBM PC).
- $\sim$ 1980: first VLSI chip ($>$ 100,000 transistors)

Today: $\sim$ 200M transistors, $\sim 0.09\mu$, ~ 3 GHz.

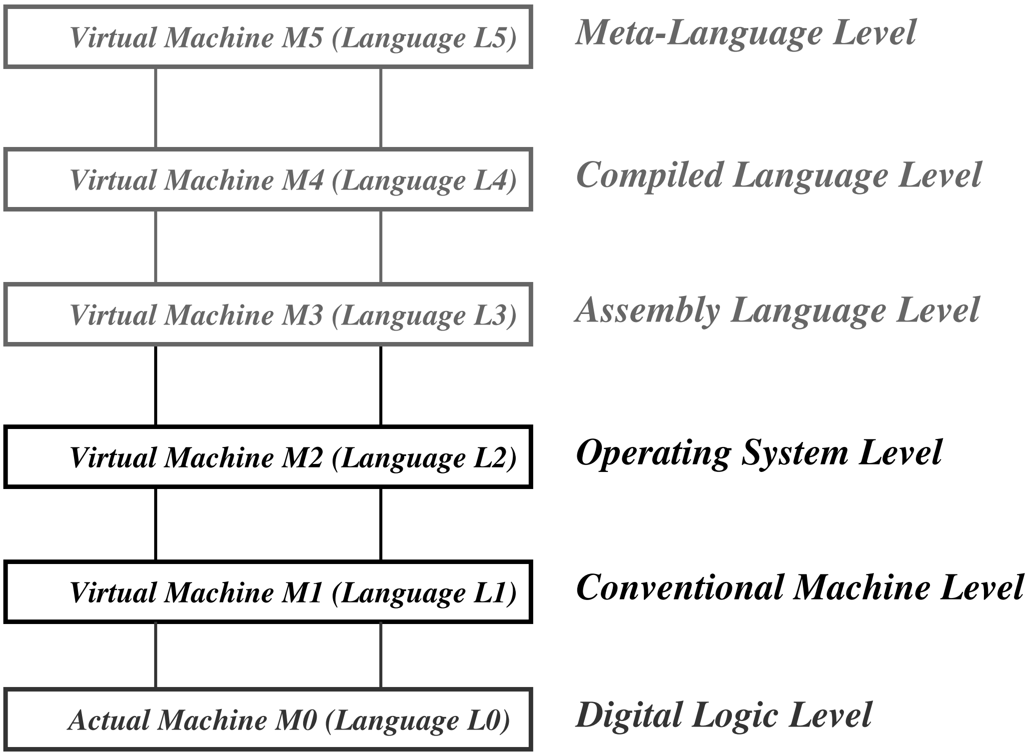
Languages and Levels



- Modern machines all programmable with a huge variety of different languages.
- e.g. ML, java, C++, C, python, perl, FORTRAN, Pascal, scheme, . . .
- We can describe the operation of a computer at a number of different *levels*; however all of these levels are *functionally equivalent*
  — i.e. can perform the same set of tasks
- Each level relates to the one below via either

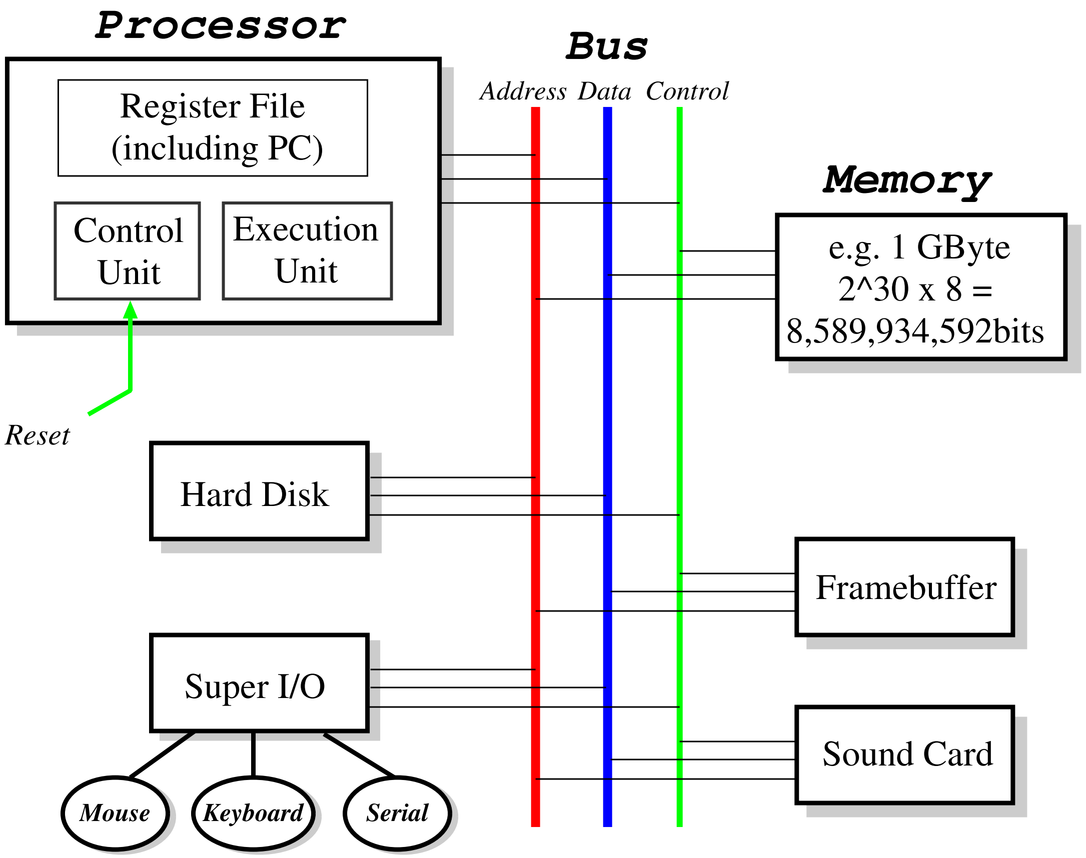
  a. translation, or
  b. interpretation.

Layered Virtual Machines


Virtual Machine M5 (Language L5)
Meta-Language Level
Virtual Machine M4 (Language L4)
Compiled Language Level
Virtual Machine M3 (Language L3)
Assembly Language Level
Virtual Machine M2 (Language L2)
Operating System Level
Virtual Machine M1 (Language L1)
Conventional Machine Level
Actual Machine M0 (Language L0)
Digital Logic Level

- In one sense, there is a set of different machines M_0, M_1, . . . M_n, each built on top of the other.
- Can consider each machine M_i to understand only machine language L_i.
- Levels 0, -1 pot. done in Dig. Elec., Physics. . .
- This course focuses on levels 1 and 2.
- NB: all levels useful; none “the truth”.

A (Simple) Modern Computer



- Processor (CPU): executes programs.
- Memory: stores both programs & data.
- Devices: for input and output.
- Bus: transfers information.

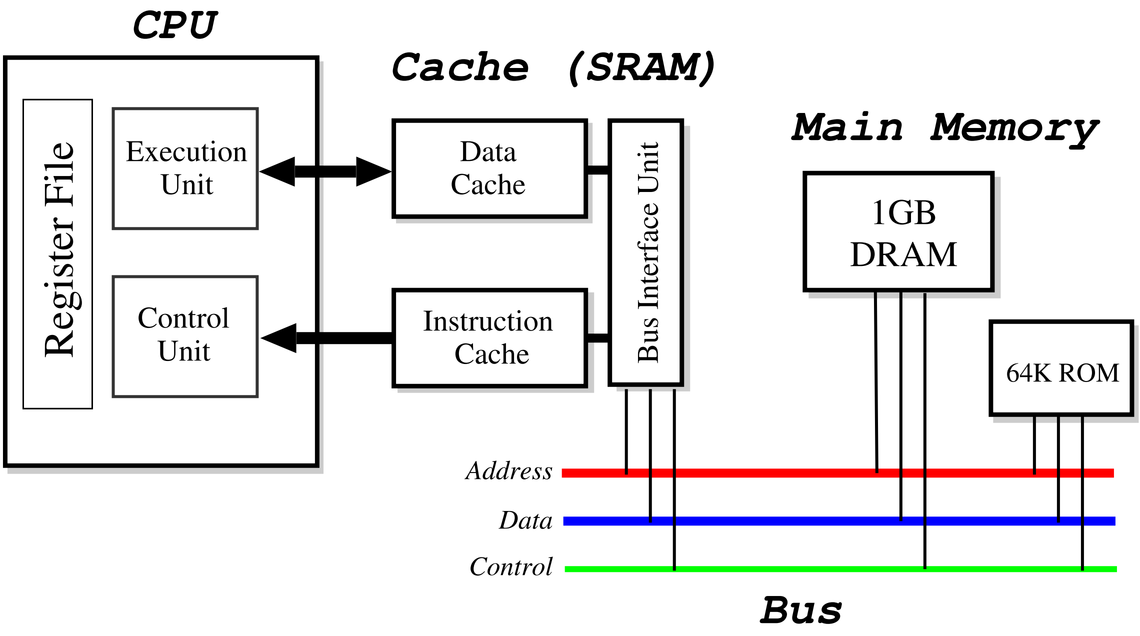
Registers and the Register File

R0	0x5A	R8	0xEA02D1F
R1	0x102034	R9	0x1001D
R2	0x2030ADCB	R10	0xFFFFFFFF
R3	0x0	R11	0x102FC8
R4	0x0	R12	0xFF0000
R5	0x2405	R13	0x37B1CD
R6	0x102038	R14	0x1
R7	0x20	R15	0x20000000

Computers all about operating on information:

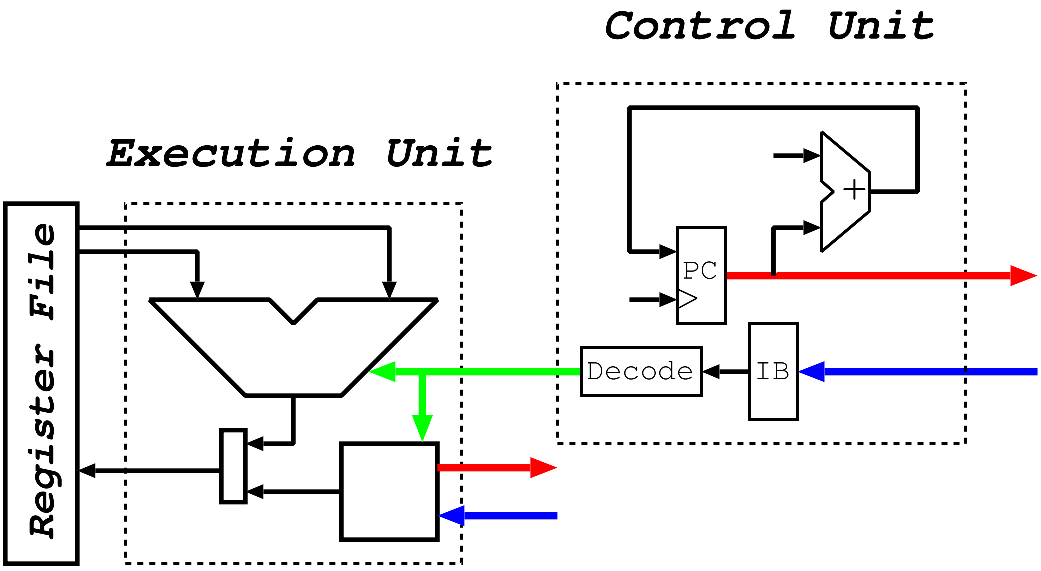
- information arrives into memory from input devices
- memory is a essentially large byte array which can hold any information we wish to operate on.
- computer *logically* takes values from memory, performs operations, and then stores result back.
- in practice, CPU operates on *registers*:
  - a register is an extremely fast piece of on-chip memory, usually either 32- or 64-bits in size.
  - modern CPUs have between 8 and 128 registers.
  - data values are *loaded* from memory into registers before being operated upon,
  - and results are *stored* back again.

Memory Hierarchy



- Use *cache* between main memory and register: try to hide delay in accessing (relatively) slow DRAM.
- Cache made from faster SRAM:
  - more expensive, so much smaller
  - holds copy of subset of main memory.
- Split of instruction and data at cache level ⇒ “Harvard” architecture.
- Cache ↔ CPU interface uses a custom bus.
- Today have $\sim$ 1MB cache, $\sim$ 1GB RAM.

The Fetch-Execute Cycle

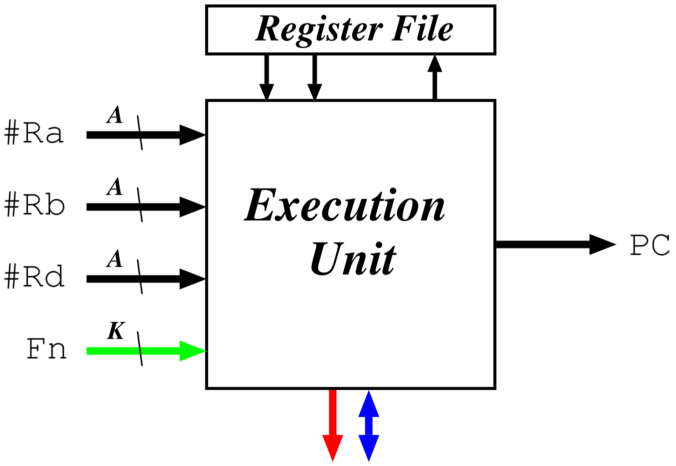


- A special register called *PC* holds a memory address; on reset, initialised to 0.
- Then:

1. Instruction *fetched* from memory address held in PC into instruction buffer (IB).
2. Control Unit determines what to do: *decodes* instruction.
3. Execution Unit *executes* instruction.
4. PC updated, and back to Step 1.

- Continues pretty much forever. . .

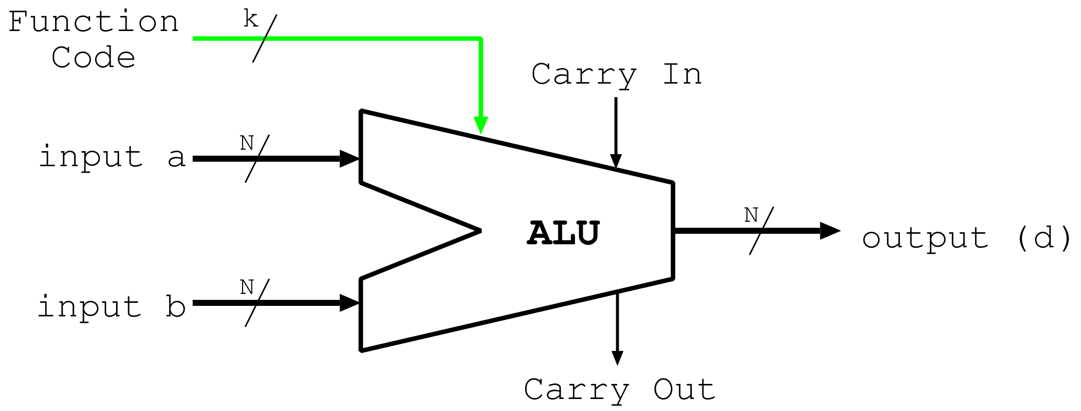
Execution Unit



- The "calculator" part of the processor.
- Broken into parts (*functional units*), e.g.
  - Arithmetic Logic Unit (ALU).
  - Shifter/Rotator.
  - Multiplier.
  - Divider.
  - Memory Access Unit (MAU).
  - Branch Unit.
- Choice of functional unit determined by signals from control unit.

Arithmetic Logic Unit

An N-bit ALU



- Part of the execution unit.
- Inputs from register file; output to register file.
- Performs simple two-operand functions:
  - a + b
  - a - b
  - a AND b
  - a OR b
  - etc.
- Typically perform *all* possible functions; use function code to select (mux) output.

Number Representation

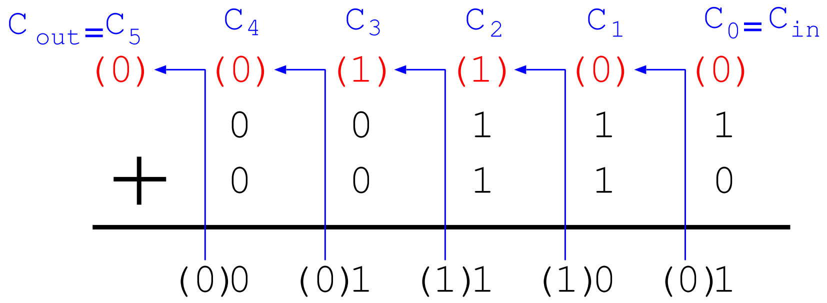
0000_2	0_{16}	0110_2	6_{16}	1100_2	C_{16}
0001_2	1_{16}	0111_2	7_{16}	1101_2	D_{16}
0010_2	2_{16}	1000_2	8_{16}	1110_2	E_{16}
0011_2	3_{16}	1001_2	9_{16}	1111_2	F_{16}
0100_2	4_{16}	1010_2	A_{16}	10000_2	10_{16}
0101_2	5_{16}	1011_2	B_{16}	10001_2	11_{16}

- a n-bit register $b_{n-1}b_{n-2}\ldots b_1b_0$ can represent 2^n different values.
- Call b_{n-1} the *most significant bit* (msb), b_0 the *least significant bit* (lsb).
- Unsigned numbers: treat the obvious way, i.e. val $= b_{n-1}2^{n-1} + b_{n-2}2^{n-2} + \cdots + b_12^1 + b_02^0$, e.g. $1101_2 = 2^3 + 2^2 + 2^0 = 8 + 4 + 1 = 13$.
- Represents values from 0 to $2^n - 1$ inclusive.
- For large numbers, binary is unwieldy: use hexadecimal (base 16).
- To convert, group bits into groups of 4, e.g. $1111101010_2 = 0011|1110|1010_2 = 3EA_{16}$.
- Often use "$0x$" prefix to denote hex, e.g. $0x107$.
- Can use dot to separate large numbers into 16-bit chunks, e.g. $0x3FF.FFFF$.

Number Representation (2)

- What about *signed* numbers? Two main options:
- Sign & magnitude:
  - top (leftmost) bit flags if negative; remaining bits make value.
  - e.g. byte $10011011_2 \rightarrow -0011011_2 = -27$.
  - represents range $-(2^{n-1}-1)$ to $+(2^{n-1}-1)$, and the bonus value -0 (!).
- 2's complement:
  - to get $-x$ from x, invert every bit and add 1.
  - e.g. $+27 = 00011011_2 \Rightarrow$
    $-27 = (11100100_2 + 1) = 11100101_2$.
  - treat $1000\ldots000_2$ as -2^{n-1}.
  - represents range -2^{n-1} to $+(2^{n-1}-1)$
- Note:
  - in both cases, top-bit means "negative".
  - both representations depend on n;
- In practice, all modern computers use 2's complement. . .

Unsigned Arithmetic



```
C out=C5    C4     C3     C2     C1     C0=Cin
   (0)     (0)    (1)    (1)    (0)    (0)
             0      0      1      1      1
  +          0      0      1      1      0
  ---------------------------------------
          (0)0   (0)1   (1)1   (1)0   (0)1
```

- (we use 5-bit registers for simplicity)
- Unsigned addition: C_n means “carry”:

```
   00101      5          11110    30
 + 00111      7       +  00111     7
-------------        --------------
0  01100     12       1  00101     5
-------------        --------------
```

- Unsigned subtraction: $\overline{C_n}$ means “borrow”:

```
   11110     30          00111     7
 + 00101    -27       +  10110   -10
-------------        --------------
1  00011      3       0  11101    29
-------------        --------------
```

Signed Arithmetic

- In signed arithmetic, carry no good on its own. Use the *overflow* flag, $V = (C_n \oplus C_{n-1})$.
- Also have *negative* flag, $N = b_{n-1}$ (i.e. the msb).
- Signed addition:

```
   00101     5           01010    10
 + 00111     7       +   00111     7
-------------        --------------
0  01100    12       0   10001   -15
-------------        --------------
   0                     1
```

- Signed subtraction:

```
   01010    10           10110   -10
 + 11001    -7       +   10110   -10
-------------        --------------
1  00011     3       1   01100    12
-------------        --------------
   1                     0
```

- Note that in overflow cases the sign of the result is always wrong (i.e. the N bit is inverted).

Arithmetic & Logical Instructions

- Some common ALU instructions are:

Mnemonic		C/Java Equivalent
`and`	$d \leftarrow a, b$	`d = a & b;`
`xor`	$d \leftarrow a, b$	`d = a ^ b;`
`bis`	$d \leftarrow a, b$	`d = a | b;`
`bic`	$d \leftarrow a, b$	`d = a & (~b);`
`add`	$d \leftarrow a, b$	`d = a + b;`
`sub`	$d \leftarrow a, b$	`d = a - b;`
`rsb`	$d \leftarrow a, b$	`d = b - a;`
`shl`	$d \leftarrow a, b$	`d = a << b;`
`shr`	$d \leftarrow a, b$	`d = a >> b;`

  Both d and a *must* be registers; b can be a register or a (small) constant.

- Typically also have `addc` and `subc`, which handle carry or borrow (for multi-precision arithmetic), e.g.

```
add  d0, a0, b0    // compute "low" part.
addc d1, a1, b1    // compute "high" part.
```

- May also get:
  - Arithmetic shifts: `asr` and `asl`(?)
  - Rotates: `ror` and `rol`.

Conditional Execution

- Seen C, N, V; add Z (zero), logical NOR of all bits in output.
- Can predicate execution based on (some combination) of flags, e.g.

  ```
  subs d, a, b    // compute d = a - b
  beq proc1      // if equal, goto proc1
  br  proc2      // otherwise goto proc2
  ```

  Java equivalent approximately:

  ```
    if (a==b) proc1() else proc2();
  ```

- On most computers, mainly limited to branches.
- On ARM (and IA64), everything conditional, e.g.

  ```
  sub   d, a, b    # compute d = a - b
  moveq d, #5      # if equal, d = 5;
  movne d, #7      # otherwise d = 7;
  ```

  Java equiv: `d = (a==b) ? 5 : 7;`

- “Silent” versions useful when don’t really want result, e.g. `tst`, `teq`, `cmp`.

Condition Codes

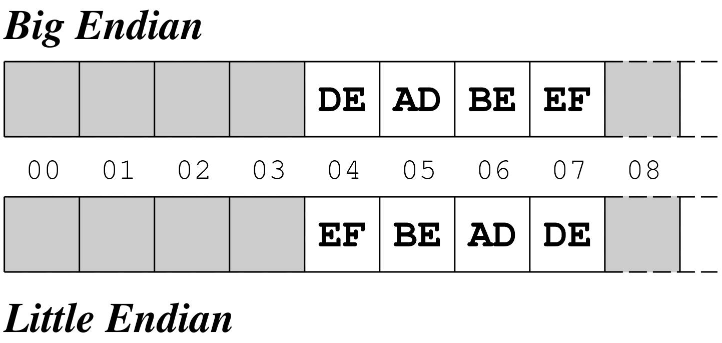
Suffix	Meaning	Flags
`EQ`, `Z`	Equal, zero	Z == 1
`NE`, `NZ`	Not equal, non-zero	Z == 0
`MI`	Negative	N == 1
`PL`	Positive (incl. zero)	N == 0
`CS`, `HS`	Carry, higher or same	C == 1
`CC`, `LO`	No carry, lower	C == 0
`VS`	Overflow	V == 1
`VC`	No overflow	V == 0
`HI`	Higher	C == 1 `&&` Z == 0
`LS`	Lower or same	C == 0 `||` Z == 1
`GE`	Greater than or equal	N == V
`GT`	Greater than	N == V `&&` Z == 0
`LT`	Less than	N `!=` V
`LE`	Less than or equal	N `!=` V `||` Z == 1

- `HS`, `LO`, etc. used for unsigned comparisons (recall that $\overline{C}$ means “borrow”).

- `GE`, `LT`, etc. used for signed comparisons: check both N and V so always works.

Loads & Stores

- Have variable sized values, e.g. bytes (8-bits), words (16-bits), longwords (32-bits) and quadwords (64-bits).
- Load or store instructions usually have a suffix to determine the size, e.g. '`b`' for byte, '`w`' for word, '`l`' for longword.
- When storing > 1 byte, have two main options: big endian and little endian; e.g. storing longword `0xDEADBEEF` into memory at address `0x4`.



Big Endian

00	01	02	03	04	05	06	07	08
				DE	**AD**	**BE**	**EF**	
				EF	**BE**	**AD**	**DE**	

Little Endian

If read back a *byte* from address `0x4`, get `0xDE` if big-endian, or `0xEF` if little-endian.

- Today have x86 & Alpha little endian; Sparc & 68K, big endian; Mips & ARM either.

Addressing Modes

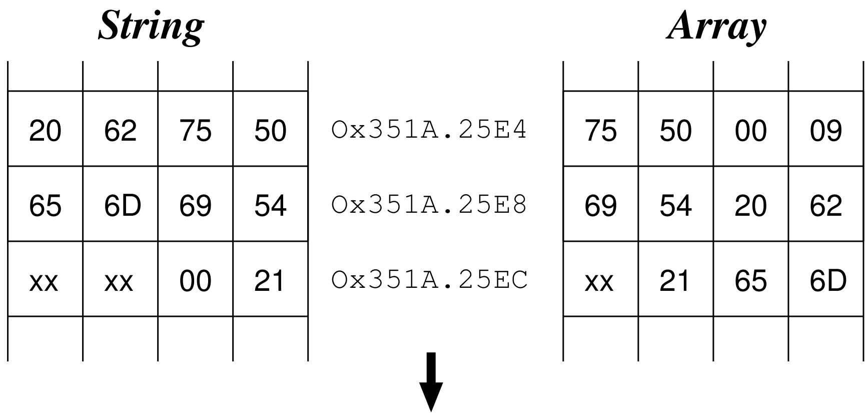
- An *addressing mode* tells the computer where the data for an instruction is to come from.
- Get a wide variety, e.g.

Register:	`add  r1, r2, r3`
Immediate:	`add  r1, r2, #25`
PC Relative:	`beq  0x20`
Register Indirect:	`ldr  r1, [r2]`
” + Displacement:	`str  r1, [r2, #8]`
Indexed:	`movl r1, (r2, r3)`
Absolute/Direct:	`movl r1, $0xF1EA0130`
Memory Indirect:	`addl r1, ($0xF1EA0130)`

- Most modern machines are *load/store* ⇒ only support first five:
  - allow at most one memory ref per instruction
  - (there are very good reasons for this)
- Note that CPU generally doesn't care *what* is being held within the memory.
- i.e. up to *programmer* to interpret whether data is an integer, a pixel or a few characters in a novel.

Representing Text

- Two main standards:
  1. ASCII: 7-bit code holding (English) letters, numbers, punctuation and a few other characters.
  2. Unicode: 16-bit code supporting practically all international alphabets and symbols.
- ASCII default on many operating systems, and on the early Internet (e.g. e-mail).
- Unicode becoming more popular (esp UTF-8!).
- In both cases, represent in memory as either *strings* or *arrays*: e.g. "Pub Time!"



String					*Array*			
20	62	75	50	`0x351A.25E4`	75	50	00	09
65	6D	69	54	`0x351A.25E8`	69	54	20	62
xx	xx	00	21	`0x351A.25EC`	xx	21	65	6D

- `0x49207769736820697420776173203a2d28`

Floating Point (1)

- In many cases want to deal with very large or very small numbers.
- Use idea of "scientific notation", e.g. $n = m \times 10^e$
  - m is called the $mantissa$
  - e is called the $exponent$.

  e.g. $C = 3.01 \times 10^8$ m/s.
- For computers, use binary i.e. $n = m \times 2^e$, where m includes a "binary point".
- Both m and e can be positive or negative; typically
  - sign of mantissa given by an additional $sign$ bit.
  - exponent is stored in a $biased$ $(excess)$ format.

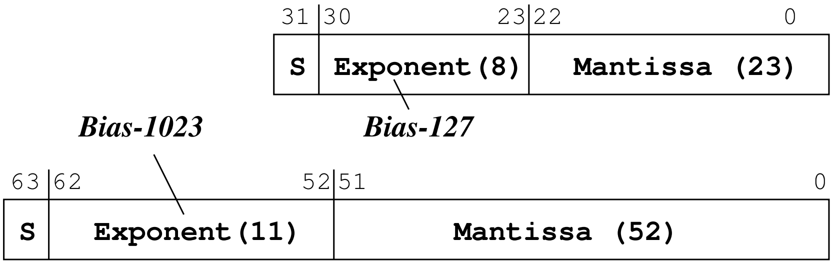
$\Rightarrow$ use $n = (-1)^s m \times 2^{e-b}$, where $0 \leq m < 2$ and b is the bias.

- e.g. 4-bit mantissa & 3-bit bias-3 exponent allows positive range $[0.001_2 \times 2^{-3}, 1.111_2 \times 2^4]$

$=$ [$(\frac{1}{8})(\frac{1}{8})$, $(\frac{15}{8})16$], or [$\frac{1}{64}$, 30]

Floating Point (2)

- In practice use IEEE floating point with *normalised* mantissa $m = 1.xx\ldots x_2$ $\Rightarrow$ use $n = (-1)^s((1+m) \times 2^{e-b})$,
- Both single (`float`) and double (`double`) precision:



- IEEE fp reserves $e = 0$ and $e = \max$:
  - ± 0 (!): both e and m zero.
  - $\pm\infty$: $e = \max$, m zero.
  - NaNs : $e = \max$, m non-zero.
  - *denorms* : $e = 0$, m non-zero
- Normal positive range $[2^{-126}, \sim 2^{128}]$ for single, or $[2^{-1022}, \sim 2^{1024}]$ for double.
- NB: still only $2^{32}/2^{64}$ values — just spread out.

Data Structures

- Records / structures: each field stored as an offset from a *base address*.
- Variable size structures: explicitly store addresses (*pointers*) inside structure, e.g.

```
datatype rec = node of int * int * rec
         |     leaf of int;

val example = node(4, 5, node(6, 7, leaf(8)));
```

Imagine `example` is stored at address `0x1000`:

Address	Value	Comment
0x0F30	0xFFFF	Constructor tag for a leaf
0x0F34	8	Integer 8
⋮		
0x0F3C	0xFFFE	Constructor tag for a node
0x0F40	6	Integer 6
0x0F44	7	Integer 7
0x0F48	0x0F30	Address of inner node
⋮		
0x1000	0xFFFE	Constructor tag for a node
0x1004	4	Integer 4
0x1008	5	Integer 5
0x100C	0x0F3C	Address of inner node

Instruction Encoding

- An instruction comprises:

  a. an *opcode*: specify what to do.
  b. zero or more *operands*: where to get values

  e.g. `add r1, r2, r3` ≡

1010111	001	010	011

- Old machines (and x86) use *variable length* encoding motivated by low code density.

- Most modern machines use fixed length encoding for simplicity. e.g. ARM ALU operations.

31 28	27 26	25	24 21	20	19 16	15 12	11 0
Cond	**00**	*I*	*Opcode*	*S*	*Ra*	*Rd*	*Operand 2*

`and r13, r13, #31` = `0xe20dd01f` =

1110	00	1	0000	0	1101	1101	000000011111

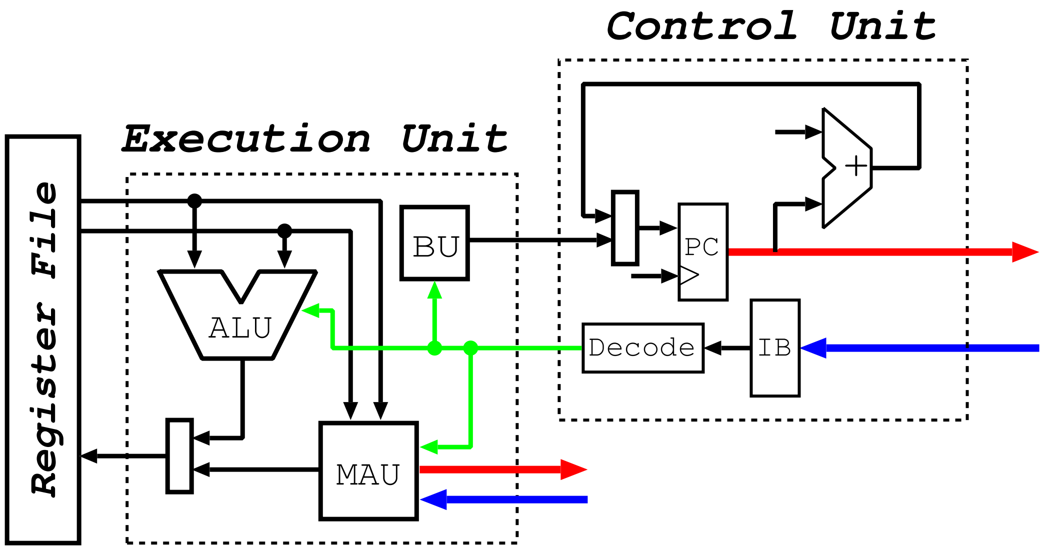
`bic r3, r3, r2` = `0xe1c33002` =

1110	00	0	1110	0	0011	0011	000000000010

`cmp r1, r2` = `0xe1510002` =

1110	00	0	1010	1	0001	0000	000000000010

Fetch-Execute Cycle Revisited

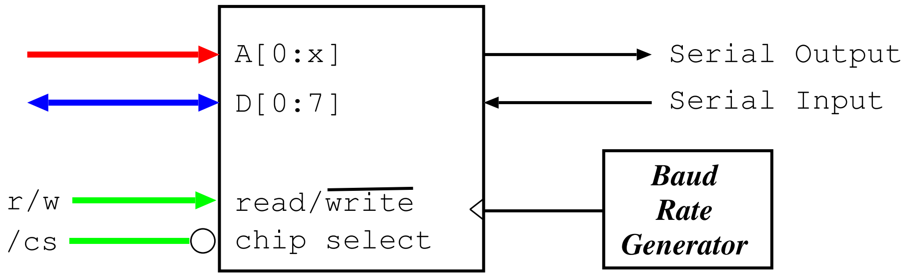


1. CU fetches & decodes instruction and generates (a) control signals and (b) operand information.
2. Inside EU, control signals select functional unit ("instruction class") and operation.
3. If ALU, then read one or two registers, perform operation, and (probably) write back result.
4. If BU, test condition and (maybe) add value to `PC`.
5. If MAU, generate address ("addressing mode") and use bus to read/write value.
6. Repeat *ad infinitum*.

Input/Output Devices

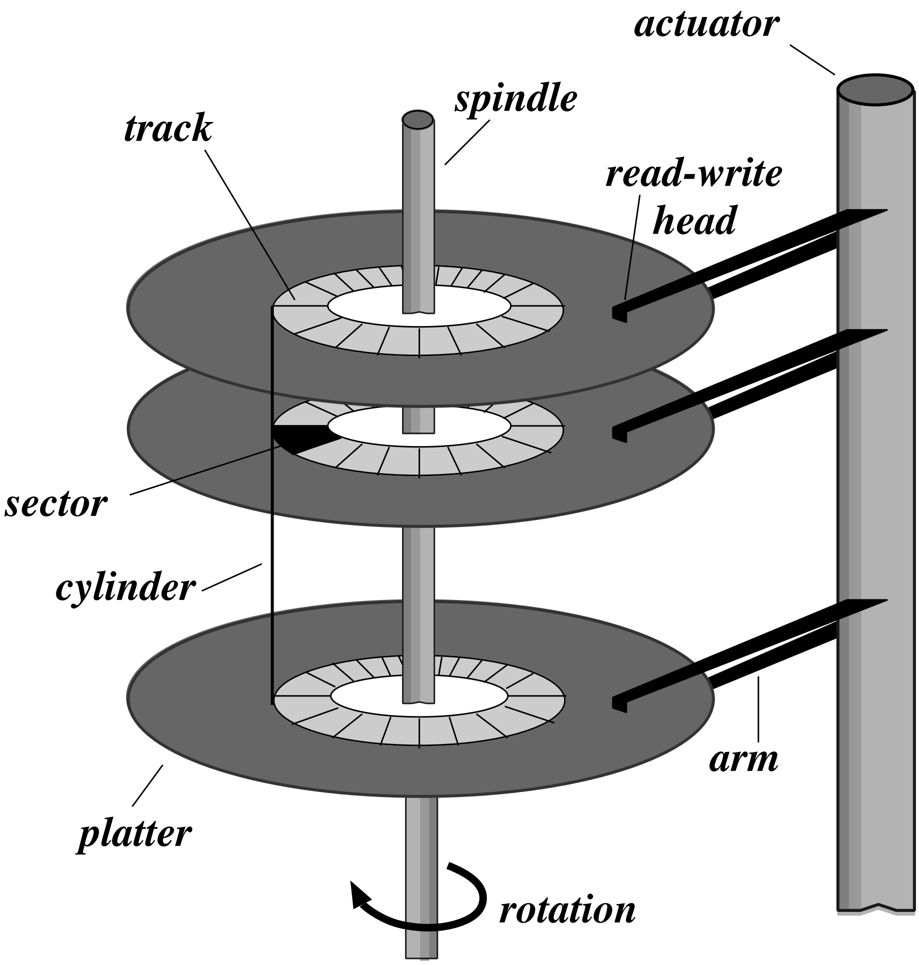
- Devices connected to processor via a *bus* (e.g. ISA, PCI, AGP).
- Includes a wide range:
  - Mouse,
  - Keyboard,
  - Graphics Card,
  - Sound card,
  - Floppy drive,
  - Hard-Disk,
  - CD-Rom,
  - Network card,
  - Printer,
  - Modem
  - etc.
- Often two or more stages involved (e.g. IDE, SCSI, RS-232, Centronics, etc.)

UARTs



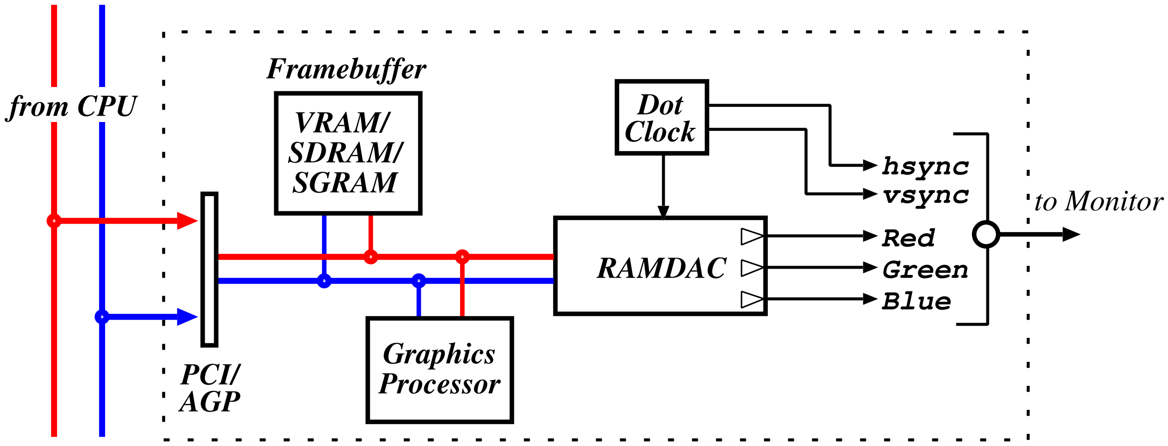
- Universal Asynchronous Receiver/Transmitter:
  - stores 1 or more bytes internally.
  - converts parallel to serial.
  - outputs according to RS-232.
- Various baud rates (e.g. 1,200 – 115,200)
- Slow and simple. . . and very useful.
- Make up “serial ports” on PC.
- Max throughput $\sim$ 14.4KBytes; variants up to 56K (for modems).

Hard Disks



- Whirling bits of (magnetized) metal. . .
- Rotate 3,600 – 12,000 times a minute.
- Capacity $\sim$ 250 GBytes ($\approx 250 \times 2^{30} bytes$).

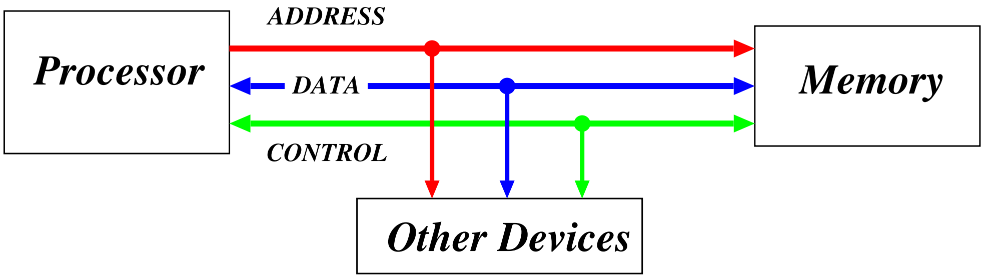
Graphics Cards



- Essentially some RAM (framebuffer) and some digital-to-analogue circuitry (RAMDAC).
- RAM holds array of $pixels$: picture elements.
- Resolutions e.g. 640x480, 800x600, 1024x768, 1280x1024, 1600x1200.
- Depths: 8-bit (LUT), 16-bit (RGB=555, 24-bit (RGB=888), 32-bit (RGBA=888).
- Memory requirement $= x \times y \times$ depth, e.g. 1280x1024 @ 16bpp needs 2560KB.

⇒ full-screen 50Hz video requires 125 MBytes/s (or $\sim$ 1Gbit/s).

Buses



- Bus = collection of *shared* communication wires:

  ✓ low cost.

  ✓ versatile / extensible.

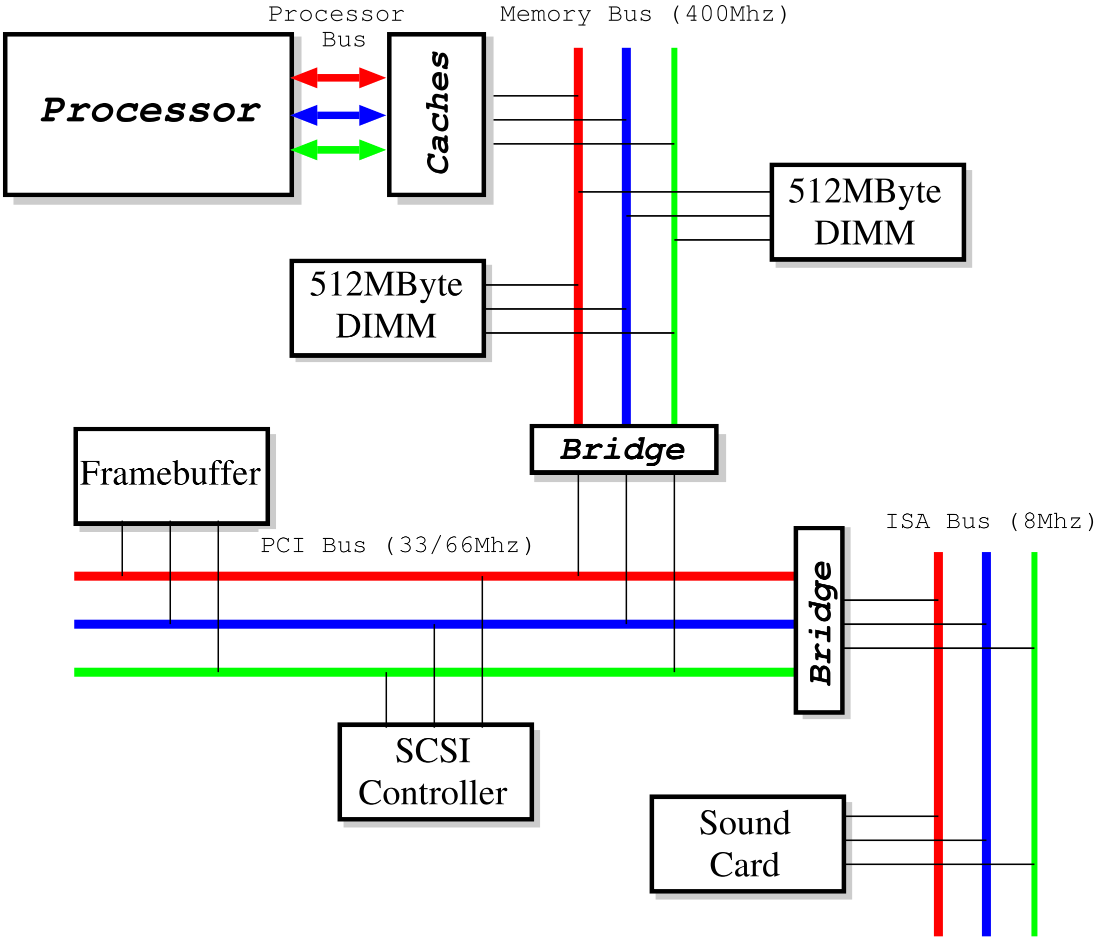
  ✗ potential bottle-neck.

- Typically comprises address lines, data lines and control lines (+ power/ground).

- Operates in a *master-slave* manner, e.g.

  1. master decides to e.g. read some data.
  2. master puts addr onto bus and asserts 'read'
  3. slave reads addr from bus and retrieves data.
  4. slave puts data onto bus.
  5. master reads data from bus.

Bus Hierarchy



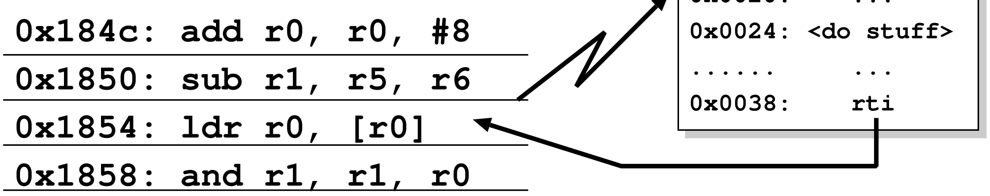
- In practice, have lots of different buses with different characteristics e.g. data width, max #devices, max length.
- Most buses are *synchronous* (share clock signal).

Interrupts

- Bus reads and writes are *transaction* based: CPU requests something and waits until it happens.
- But e.g. reading a block of data from a hard-disk takes $\sim 2ms$, which is $\sim 5,000,000$ clock cycles!
- *Interrupts* provide a way to decouple CPU requests from device responses.

  1. CPU uses bus to make a request (e.g. *writes* some special values to a device).
  2. Device goes off to get info.
  3. Meanwhile CPU continues doing other stuff.
  4. When device finally has information, raises an *interrupt*.
  5. CPU uses bus to read info from device.

- When interrupt occurs, CPU *vectors* to handler, then *resumes* using special instruction, e.g.

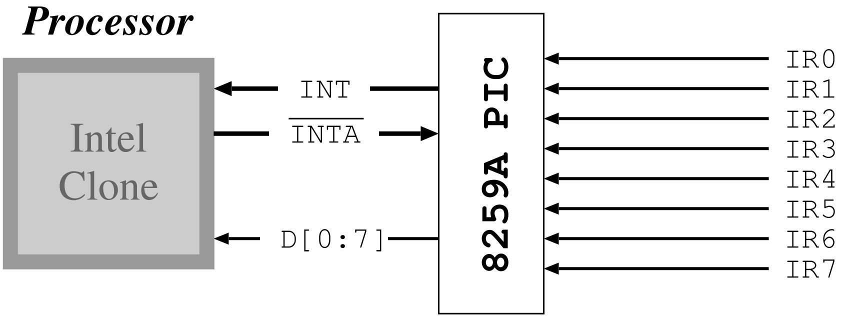


```
0x184c: add r0, r0, #8
0x1850: sub r1, r5, r6
0x1854: ldr r0, [r0]
0x1858: and r1, r1, r0
```

```
0x0020:     ...
0x0024: <do stuff>
......      ...
0x0038:     rti
```

Interrupts (2)

- Interrupt lines ($\sim 4 - 8$) are part of the bus.
- Often only 1 or 2 pins on chip ⇒ need to encode.
- e.g. ISA & x86:



1. Device asserts `IRx`.
2. PIC asserts `INT`.
3. When CPU can interrupt, strobes `INTA`.
4. PIC sends interrupt number on `D[0:7]`.
5. CPU uses number to index into a table in memory which holds the addresses of handlers for each interrupt.
6. CPU saves registers and jumps to handler.

Direct Memory Access (DMA)

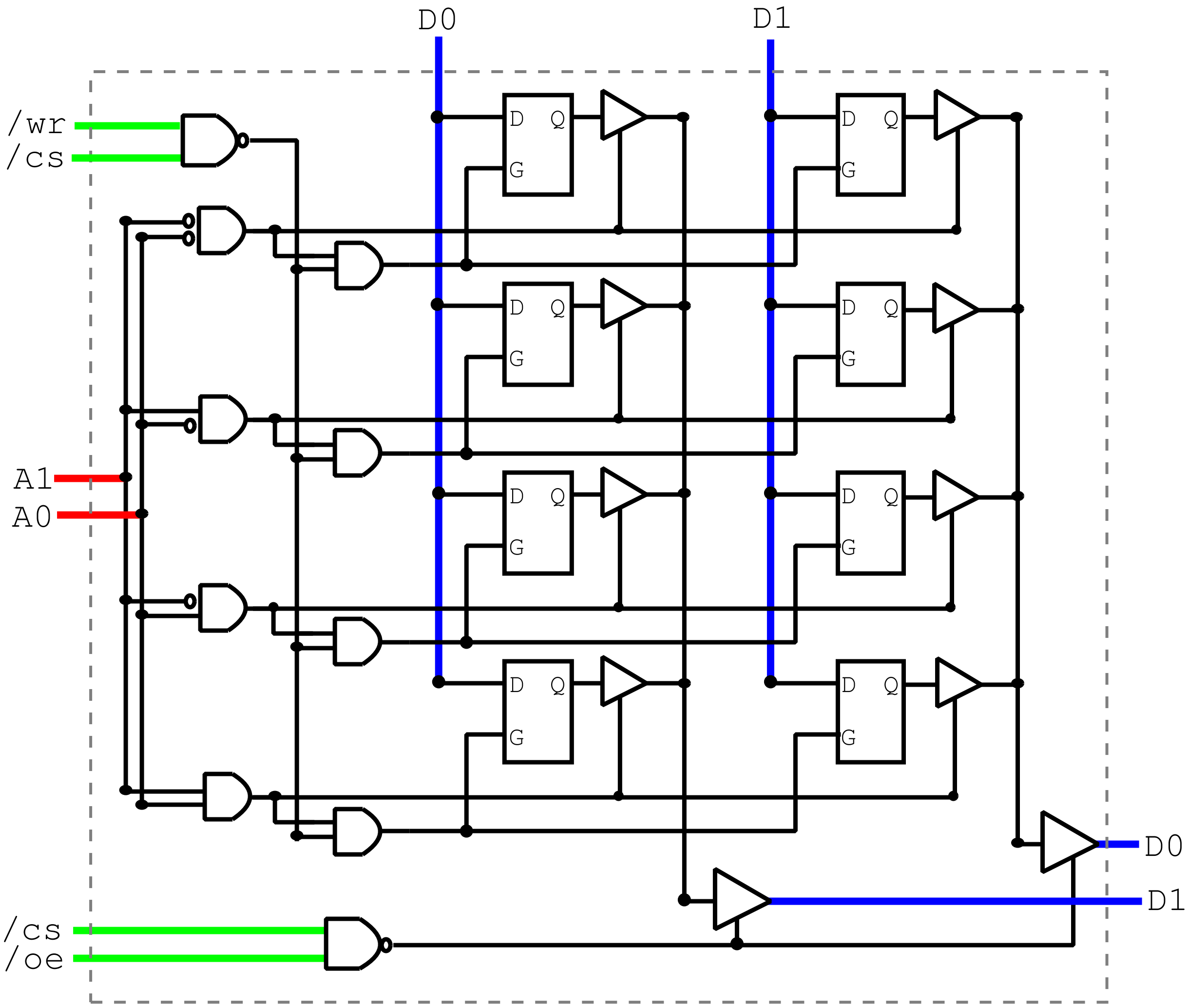
- Interrupts good, but even better is a device which can read and write processor memory *directly*.
- A generic DMA "command" might include
  - source address
  - source increment / decrement / do nothing
  - sink address
  - sink increment / decrement / do nothing
  - transfer size
- Get one interrupt at end of data transfer
- DMA channels may be provided by devices themselves:
  - e.g. a disk controller
  - pass disk address, memory address and size
  - give instruction to read or write
- Also get "stand-alone" programmable DMA controllers.

Summary

- Computers made up of four main parts:
  1. Processor (including register file, control unit and execution unit),
  2. Memory (caches, RAM, ROM),
  3. Devices (disks, graphics cards, etc.), and
  4. Buses (interrupts, DMA).
- Information represented in all sorts of formats:
  - signed & unsigned integers,
  - strings,
  - floating point,
  - data structures,
  - instructions.
- Can (hopefully) understand all of these at some level, but gets pretty complex.

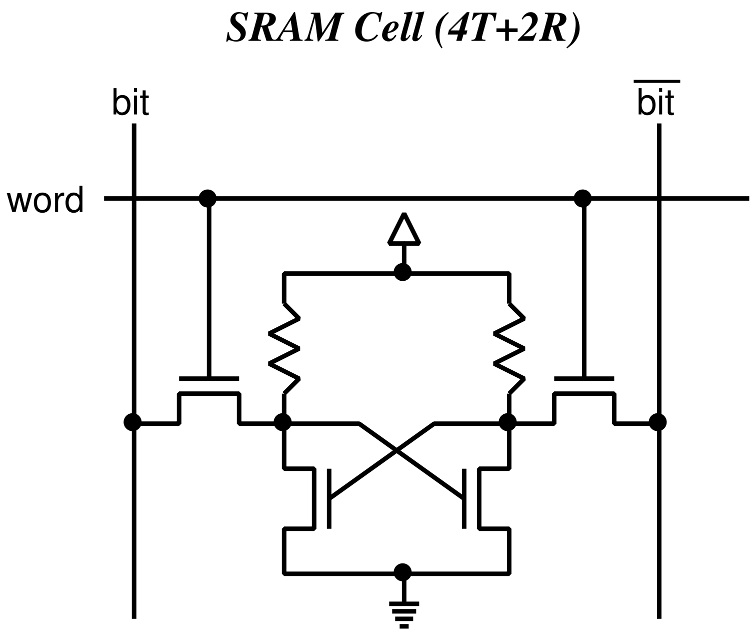
⇒ to be able to actually *use* a computer, need an operating system.

Static RAM (SRAM)



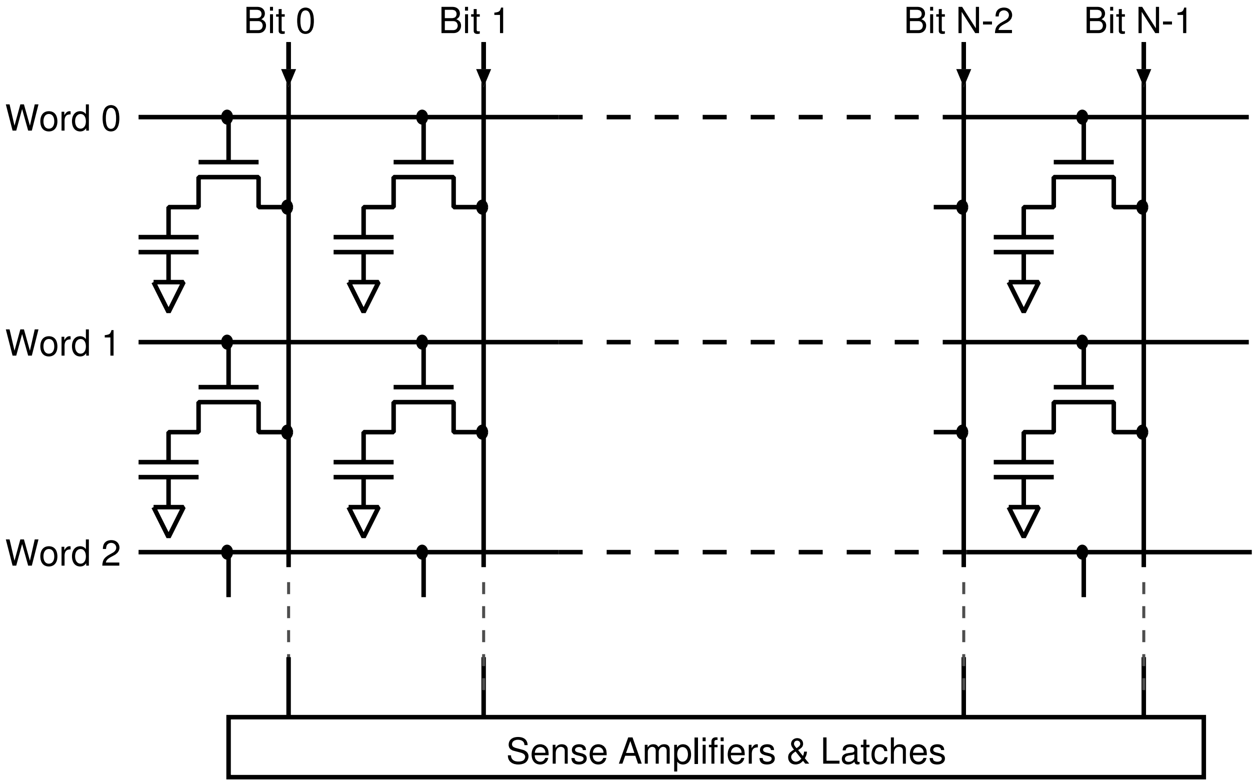
- Relatively fast (currently $5 - 20ns$).
- Logically an array of (transparent) D-latches
- In reality, only cost ~ 6 transistors per bit.

SRAM Reality



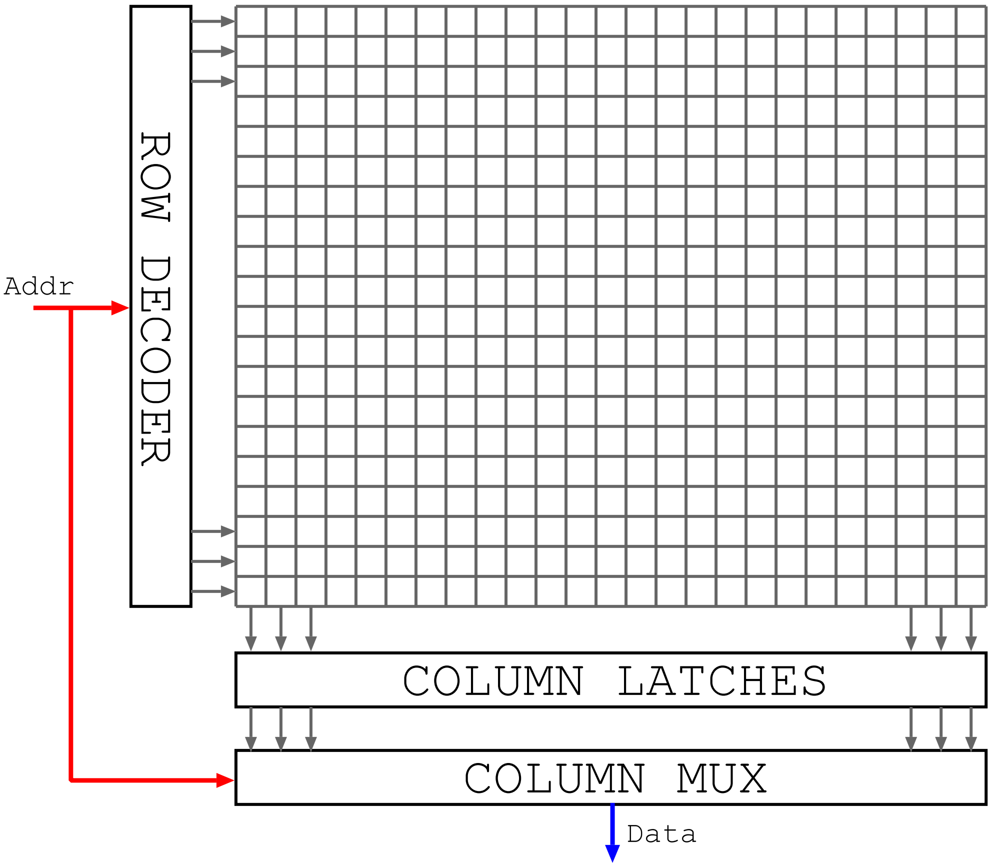
- State held in cross-coupled inverters.
- More common is $6T$ cell; use depletion mode transistors as load resistors.
- To read:
  - precharge `bit` and $\overline{\texttt{bit}}$,
  - strobe `word`,
  - detect difference (sense amp).
- To write:
  - precharge either `bit` (for “1”) or $\overline{\texttt{bit}}$ (for “0”),
  - strobe `word`.

Dynamic RAM (DRAM)



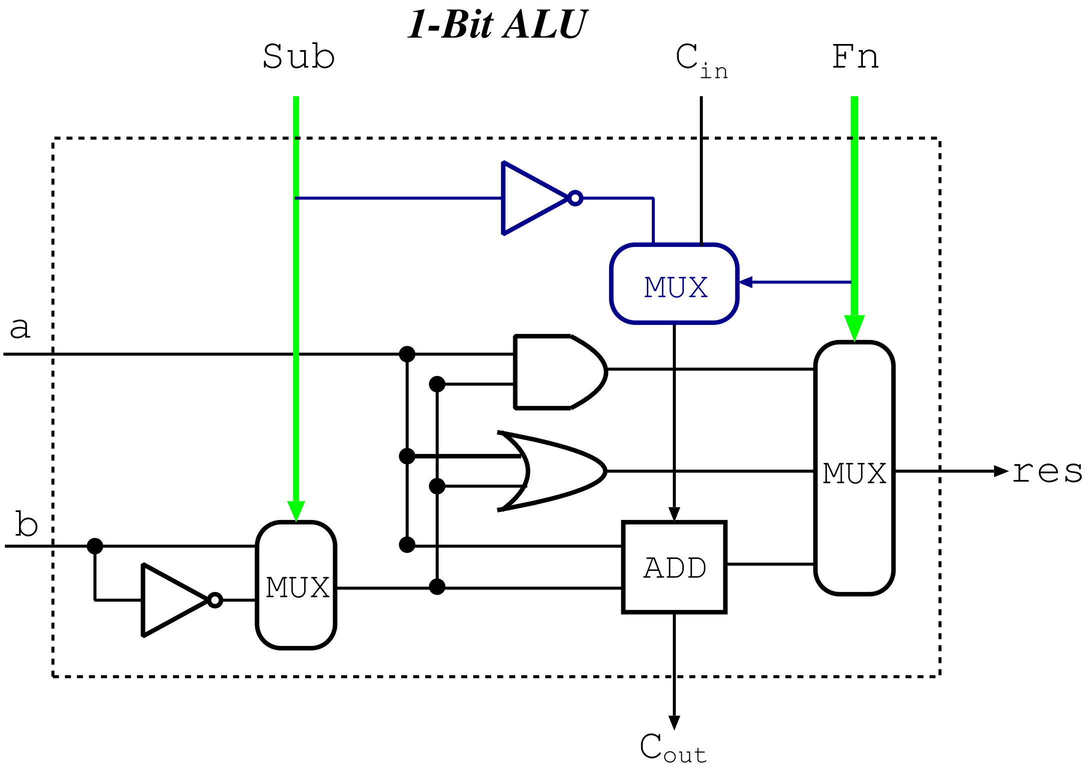
- Use a single transistor to store a bit.
- Write: put value on bit lines, strobe word line.
- Read: pre-charge, strobe word line, amplify, latch.
- “Dynamic”: refresh periodically to restore charge.
- Slower than SRAM: typically $50ns - 100ns$.

DRAM Decoding



- Two stage: row, then column.
- Usually share address pins: RAS & CAS select decoder or mux.
- FPM, EDO, SDRAM faster for same row reads.

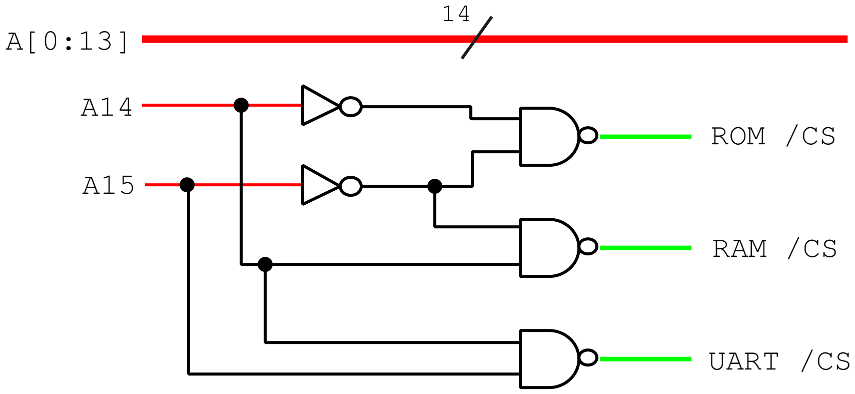
A 1-bit ALU Implementation



- Eight possible functions (4 types):
  1. a AND b, a AND $\overline{b}$.
  2. a OR b, a OR $\overline{b}$.
  3. $a+b$, $a+b$ with carry.
  4. $a-b$, $a-b$ with borrow.
- To make n-bit ALU bit, connect together (use carry-lookahead on adders).

Accessing Memory

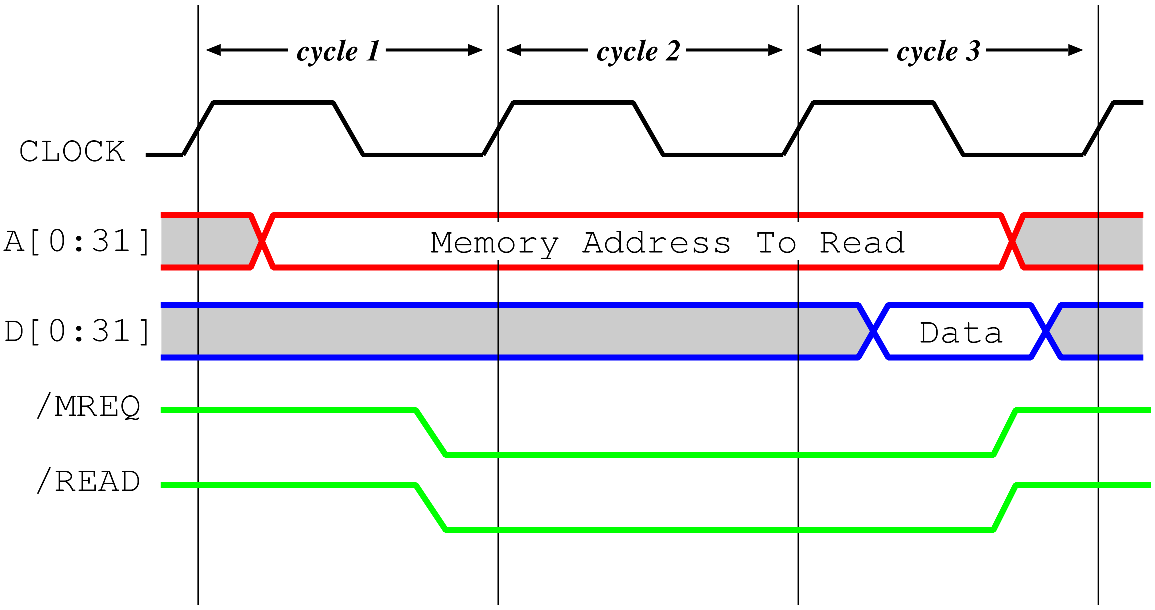
- To load/store values need the *address* in memory.
- Most modern machines are *byte addressed*: consider memory a big array of 2^A bytes, where A is the number of address lines in the bus.
- Lots of things considered "memory" via address decoder, e.g.



- Typically each device decodes only a subset of low address lines, e.g.

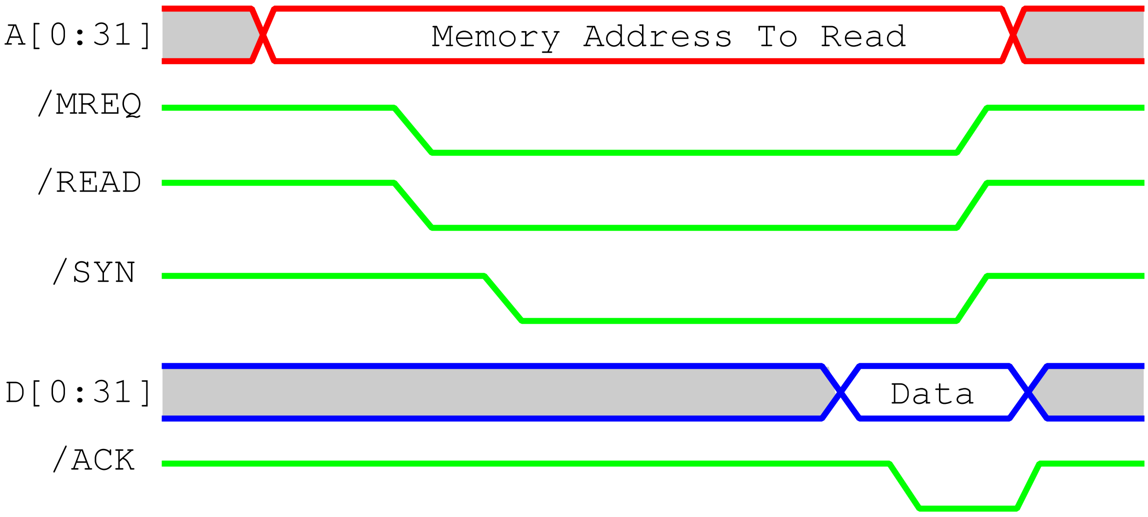
Device	Size	Data	Decodes
ROM	1024 bytes	32-bit	A[2:9]
RAM	16384 bytes	32-bit	A[2:13]
UART	256 bytes	8-bit	A[0:7]

Synchronous Buses



- Figure shows a read transaction which requires three bus cycles.
  1. CPU puts address onto address lines and, after settle, asserts control lines.
  2. Memory fetches data from address.
  3. Memory puts data on data lines, CPU latches value and then deasserts control lines.
- If device not fast enough, can insert *wait states*.
- Faster clock/longer bus can give *bus skew*.

Asynchronous Buses

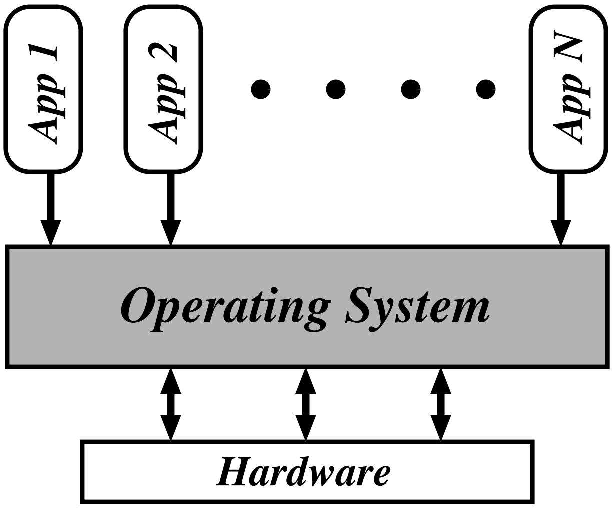


- Asynchronous buses have no shared clock; instead work by *handshaking*, e.g.
  - CPU puts address onto address lines and, after settle, asserts control lines.
  - next, CPU asserts /SYN to say everything ready.
  - as soon as memory notices /SYN, it fetches data from address and puts it onto bus.
  - memory then asserts /ACK to say data is ready.
  - CPU latches data, then deasserts /SYN.
  - finally, Memory deasserts /ACK.
- More handshaking if mux address & data. . .

What is an Operating System?

- A program which controls the execution of all other programs (applications).
- Acts as an intermediary between the user(s) and the computer.
- Objectives:
  - convenience,
  - efficiency,
  - extensibility.
- Similar to a government. . .

An Abstract View



- The Operating System (OS):
  - controls all execution.
  - multiplexes resources between applications.
  - abstracts away from complexity.
- Typically also have some *libraries* and some *tools* provided with OS.
- Are these part of the OS? Is IE4 a tool?
  - no-one can agree. . .
- For us, the OS ≈ the *kernel*.

In The Beginning. . .

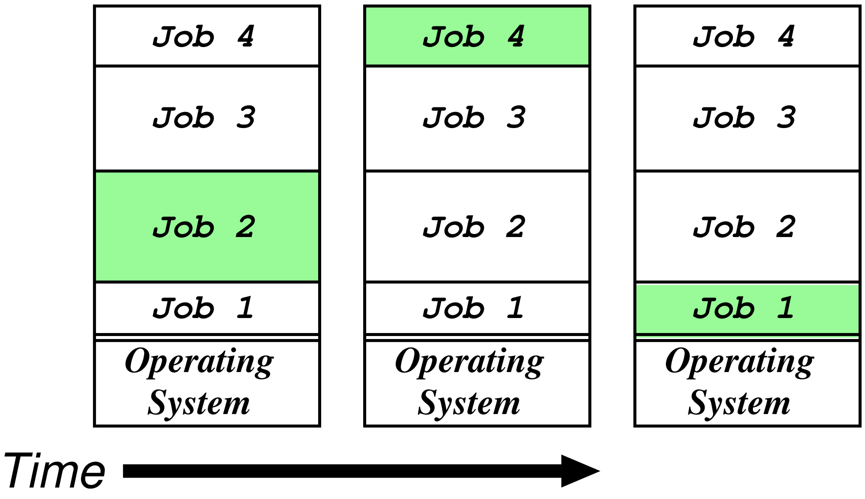
- 1949: First stored-program machine (EDSAC)
- to $\sim$ 1955: "Open Shop".
  - large machines with vacuum tubes.
  - I/O by paper tape / punch cards.
  - user = programmer = operator.
- To reduce cost, hire an *operator*:
  - programmers write programs and submit tape/cards to operator.
  - operator feeds cards, collects output from printer.
- Management like it.
- Programmers hate it.
- Operators hate it.

$\Rightarrow$ need something better.

Batch Systems

- Introduction of tape drives allow *batching* of jobs:
  - programmers put jobs on cards as before.
  - all cards read onto a tape.
  - operator carries input tape to computer.
  - results written to output tape.
  - output tape taken to printer.
- Computer now has a *resident monitor*:
  - initially control is in monitor.
  - monitor reads job and transfer control.
  - at end of job, control transfers back to monitor.
- Even better: *spooling systems*.
  - use interrupt driven I/O.
  - use magnetic disk to cache input tape.
  - fire operator.
- Monitor now *schedules* jobs. . .

Multi-Programming

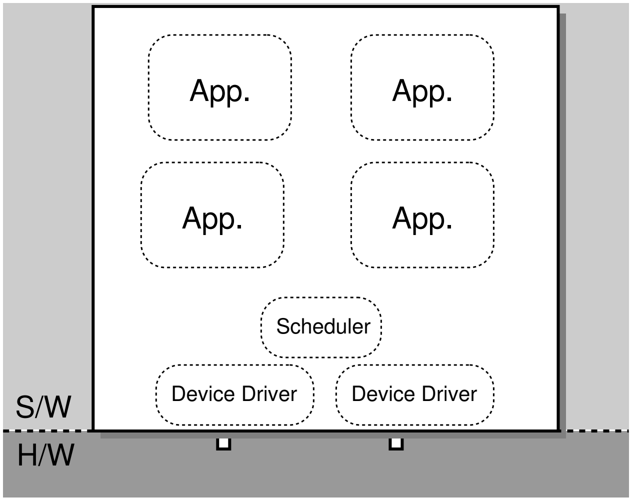


- Use memory to cache jobs from disk ⇒ more than one job active simultaneously.
- Two stage scheduling:
  1. select jobs to load: *job scheduling*.
  2. select resident job to run: *CPU scheduling*.
- Users want more interaction ⇒ *time-sharing*:
- e.g. CTSS, TSO, Unix, VMS, Windows NT. . .

Today and Tomorrow

- Single user systems: cheap and cheerful.
  - personal computers.
  - no other users ⇒ ignore protection.
  - e.g. DOS, Windows, Win 95/98, . . .
- RT Systems: power is nothing without control.
  - hard-real time: nuclear reactor safety monitor.
  - soft-real time: mp3 player.
- Parallel Processing: the need for speed.
  - SMP: 2–8 processors in a box.
  - MIMD: super-computing.
- Distributed computing: global processing?
  - Java: the network is the computer.
  - Clustering: the network is the bus.
  - CORBA: the computer is the network.
  - .NET: the network is an enabling framework. . .

Monolithic Operating Systems



- Oldest kind of OS structure ("modern" examples are DOS, original MacOS)
- Problem: applications can e.g.
  - trash OS software.
  - trash another application.
  - hoard CPU time.
  - abuse I/O devices.
  - etc. . .
- No good for fault containment (or multi-user).
- Need a better solution. . .

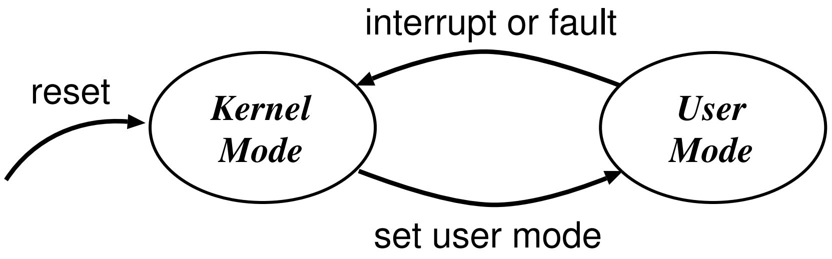
Dual-Mode Operation

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

⇒ provide $hardware$ support to differentiate between (at least) two modes of operation.

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

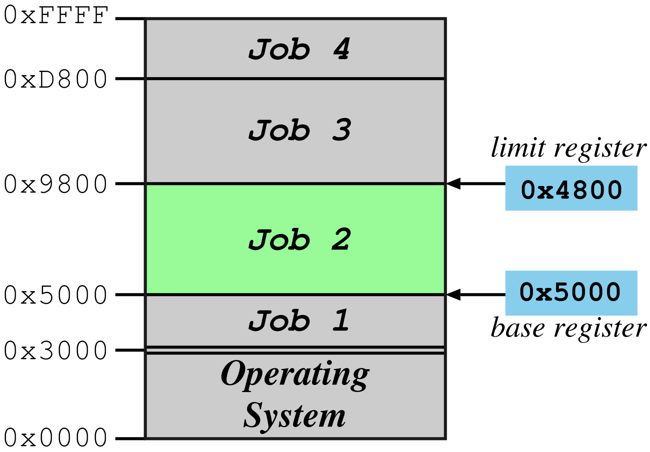
- Hardware contains a mode-bit, e.g. 0 means kernel, 1 means user.



- Make certain machine instructions only possible in kernel mode. . .

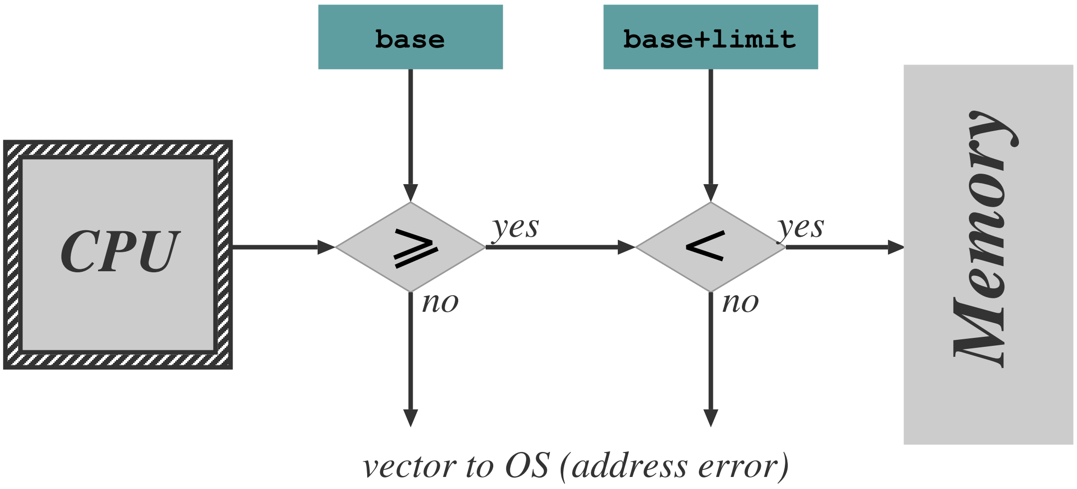
Protecting I/O & Memory

- First try: make I/O instructions privileged.
  - applications can't mask interrupts.
  - applications can't control I/O devices.
- But:
  1. Application can rewrite interrupt vectors.
  2. Some devices accessed via *memory*
- Hence need to protect memory also. . .
- e.g. define a *base* and a *limit* for each program.



- Accesses outside allowed range are protected.

Memory Protection Hardware



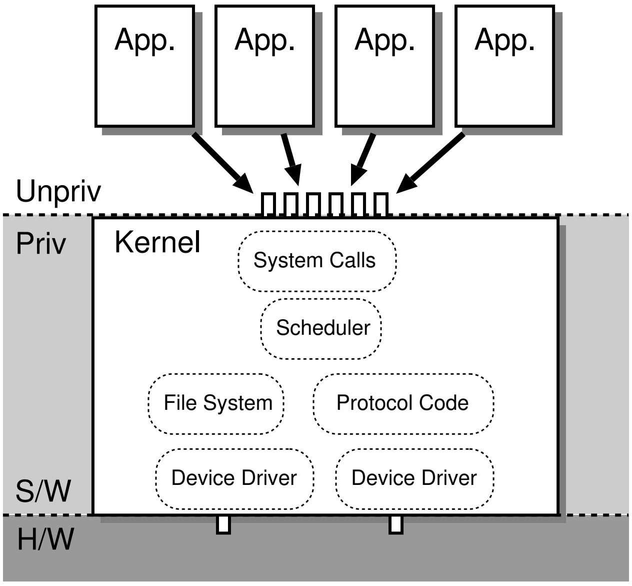
- Hardware checks every memory reference.
- Access out of range ⇒ vector into operating system (just as for an interrupt).
- Only allow *update* of base and limit registers in kernel mode.
- Typically disable memory protection in kernel mode (although a bad idea).
- In reality, more complex protection h/w used:
  - main schemes are *segmentation* and *paging*
  - (covered later on in course)

Protecting the CPU

- Need to ensure that the OS stays in control.
  - i.e. need to prevent any given application from 'hogging' the CPU the whole time.

  ⇒ use a *timer* device.
- Usually use a *countdown* timer, e.g.
  1. set timer to initial value (e.g. `0xFFFF`).
  2. every *tick* (e.g. $1\mu s$), timer decrements value.
  3. when value hits zero, interrupt.
- (Modern timers have programmable tick rate.)
- Hence OS gets to run periodically and do its stuff.
- Need to ensure only OS can load timer, and that interrupt cannot be masked.
  - use same scheme as for other devices.
  - (viz. privileged instructions, memory protection)
- Same scheme can be used to implement time-sharing (more on this later).

Kernel-Based Operating Systems



- Applications can't do I/O due to protection

  ⇒ operating system does it on their behalf.

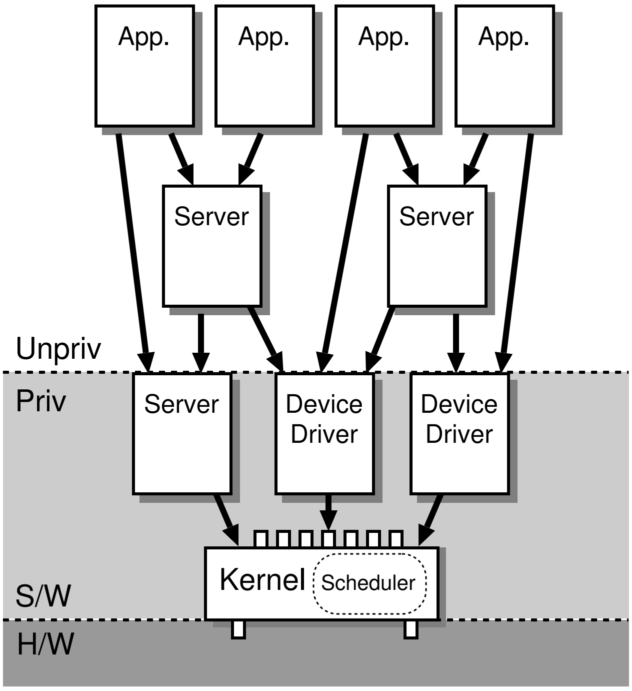
- Need secure way for application to invoke operating system:

  ⇒ require a special (unprivileged) instruction to allow transition from user to kernel mode.

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

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

Microkernel Operating Systems



- Alternative structure:
  - push some OS services into *servers*.
  - servers may be privileged (i.e. operate in kernel mode).
- Increases both *modularity* and *extensibility*.
- Still access kernel via system calls, but need new way to access servers:

  ⇒ interprocess communication (IPC) schemes.

Kernels versus Microkernels

So why isn't everything a microkernel?

- Lots of IPC adds overhead

  ⇒ microkernels usually perform less well.

- Microkernel implementation sometimes tricky: need to worry about synchronisation.
- Microkernels often end up with redundant copies of OS data structures.

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

- e.g. linux is "kernel", but has kernel modules and certain servers.
- e.g. Windows NT was originally microkernel (3.5), but now (4.0 onwards) pushed lots back into kernel for performance.
- Still not clear what the best OS structure is, or how much it really matters. . .

Operating System Functions

- Regardless of structure, OS needs to *securely multiplex resources*, i.e.

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

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

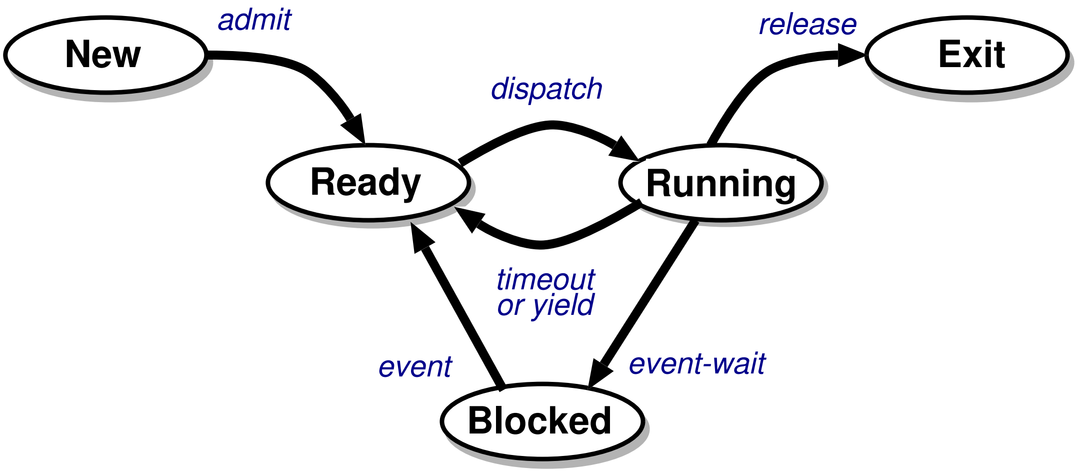
  - share CPU (in time) and provide each application with a virtual processor,
  - allocate and protect memory, and provide applications with their own virtual address space,
  - present a set of (relatively) hardware independent virtual devices, and
  - divide up storage space by using filing systems.

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

Process Concept

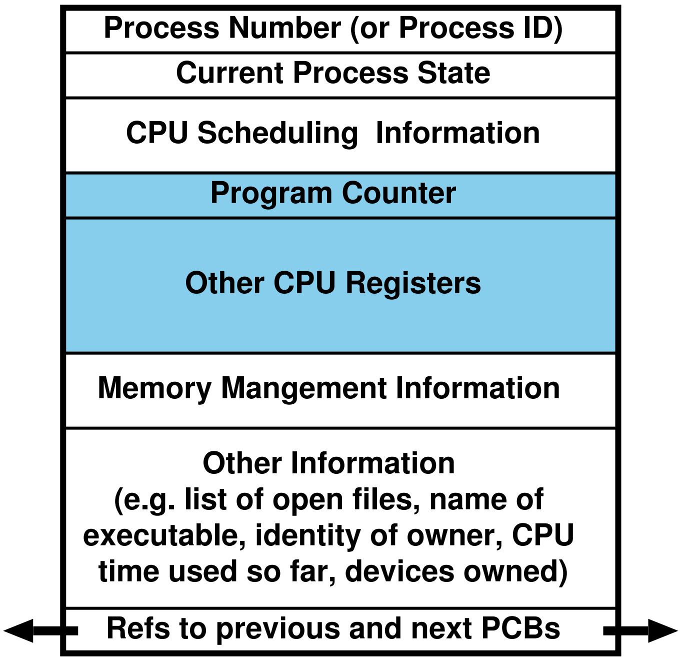
- From a user's point of view, the operating system is there to execute programs:
  - on batch system, refer to *jobs*
  - on interactive system, refer to *processes*
  - (we'll use both terms fairly interchangeably)
- Process $\neq$ Program:
  - a program is *static*, while a process is *dynamic*
  - in fact, a process $\triangleq$ "a program in execution"
- (Note: "program" here is pretty low level, i.e. native machine code or *executable*)
- Process includes:
  1. program counter
  2. stack
  3. data section
- Processes execute on *virtual processors*

Process States



- As a process executes, it changes $state$:
  - *New*: the process is being created
  - *Running*: instructions are being executed
  - *Ready*: the process is waiting for the CPU (and is prepared to run at any time)
  - *Blocked*: the process is waiting for some event to occur (and cannot run until it does)
  - *Exit*: the process has finished execution.
- The operating system is responsible for maintaining the state of each process.

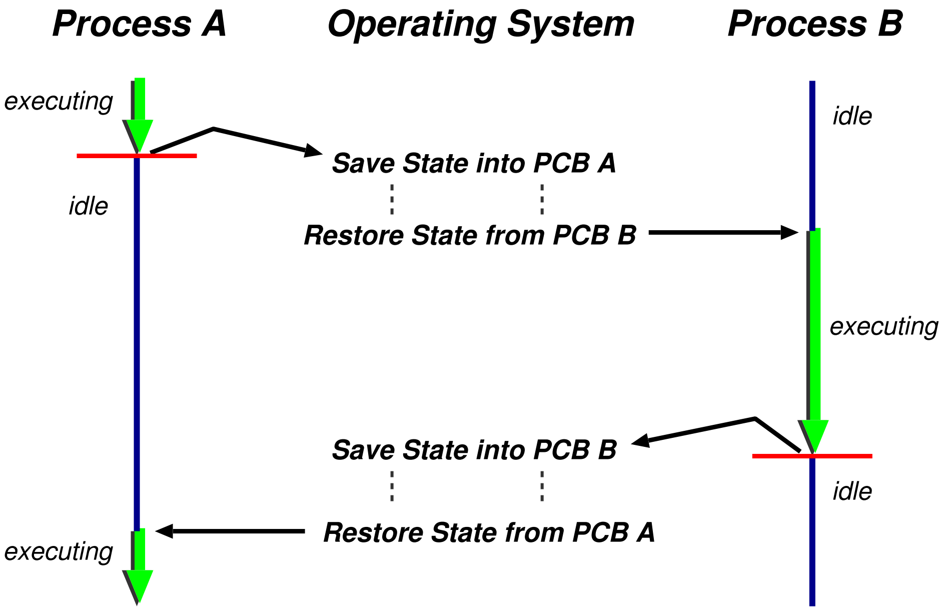
Process Control Block


Process Number (or Process ID)
Current Process State
CPU Scheduling Information
Program Counter
Other CPU Registers
Memory Mangement Information
Other Information
(e.g. list of open files, name of
executable, identity of owner, CPU
time used so far, devices owned)
Refs to previous and next PCBs

OS maintains information about every process in a data structure called a *process control block* (PCB):

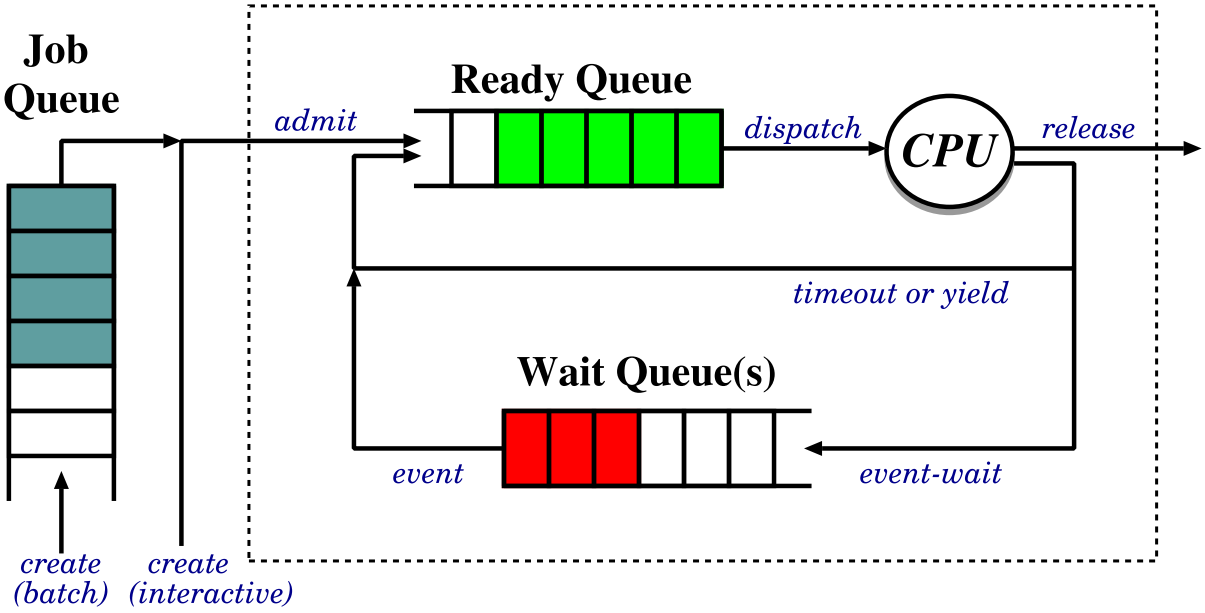
- Unique process identifier
- Process state (*Running*, *Ready*, etc.)
- CPU scheduling & accounting information
- Program counter & CPU registers
- Memory management information
- . . .

Context Switching



- *Process Context* = machine environment during the time the process is actively using the CPU.
- i.e. context includes program counter, general purpose registers, processor status register, . . .
- To switch between processes, the OS must:
  - a) save the context of the currently executing process (if any), and
  - b) restore the context of that being resumed.
- Time taken depends on h/w support.

Scheduling Queues



- Job Queue: batch processes awaiting admission.
- Ready Queue: set of all processes residing in main memory, ready and waiting to execute.
- Wait Queue(s): set of processes waiting for an I/O device (or for other processes)
- Long-term & short-term schedulers:
  - *Job scheduler* selects which processes should be brought into the ready queue.
  - *CPU scheduler* selects which process should be executed next and allocates CPU.

Process Creation

- Nearly all systems are *hierarchical*: parent processes create children processes.
- Resource sharing:
  - parent and children share all resources.
  - children share subset of parent's resources.
  - parent and child share no resources.
- Execution:
  - parent and children execute concurrently.
  - parent waits until children terminate.
- Address space:
  - child duplicate of parent.
  - child has a program loaded into it.
- e.g. Unix:
  - `fork()` system call creates a new process
  - all resources shared (child is a clone).
  - `execve()` system call used to replace the process' memory space with a new program.
- NT/2K/XP: `CreateProcess()` system call includes name of program to be executed.

Process Termination

- Process executes last statement and asks the operating system to delete it (**exit**):
  - output data from child to parent (**wait**)
  - process' resources are deallocated by the OS.
- Process performs an illegal operation, e.g.
  - makes an attempt to access memory to which it is not authorised,
  - attempts to execute a privileged instruction
- Parent may terminate execution of child processes (**abort**, **kill**), e.g. because
  - child has exceeded allocated resources
  - task assigned to child is no longer required
  - parent is exiting ("cascading termination")
  - (many operating systems do not allow a child to continue if its parent terminates)
- e.g. Unix has `wait()`, `exit()` and `kill()`
- e.g. NT/2K/XP has `ExitProcess()` for self and `TerminateProcess()` for others.

Process Blocking

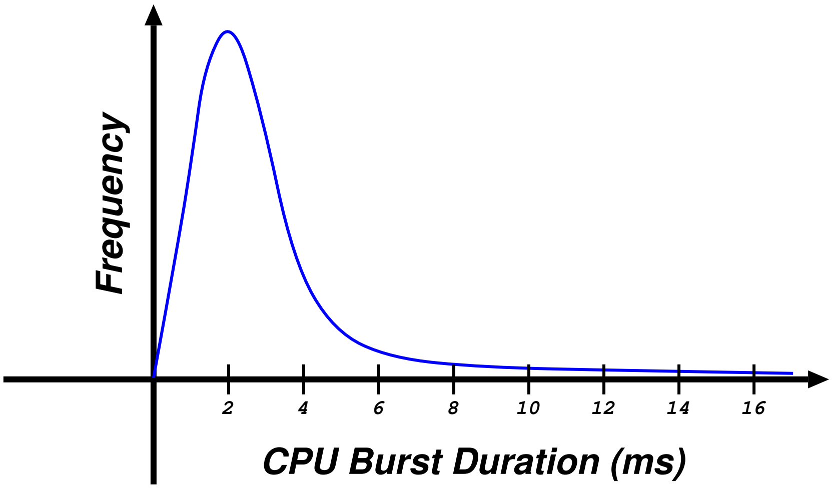
- In general a process blocks on an *event*, e.g.
  - an I/O device completes an operation,
  - another process sends a message
- Assume OS provides some kind of general-purpose blocking primitive, e.g. `await()`.
- Need care handling *concurrency* issues, e.g.

```
if(no key being pressed) {
   await(keypress);
   print("Key has been pressed!\n");
}
// handle keyboard input
```

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

- (This is a *big* area: lots more detail next year.)
- In this course we'll generally assume that problems of this sort do not arise.

CPU-I/O Burst Cycle



- CPU-I/O Burst Cycle: process execution consists of a *cycle* of CPU execution and I/O wait.
- Processes can be described as either:
  1. I/O-bound: spends more time doing I/O that than computation; has many short CPU bursts.
  2. CPU-bound: spends more time doing computations; has few very long CPU bursts.
- Observe most processes execute for at most a few milliseconds before blocking

⇒ need multiprogramming to obtain decent overall CPU utilization.

CPU Scheduler

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

- There are a number of occasions when we can/must choose a new process to run:
  1. a running process blocks (`running` → `blocked`)
  2. a timer expires (`running` → `ready`)
  3. a waiting process unblocks (`blocked` → `ready`)
  4. a process terminates (`running` → `exit`)
- If only make scheduling decision under 1, 4 ⇒ have a *non-preemptive* scheduler:

  ✔ simple to implement

  ✘ open to denial of service

  – e.g. Windows 3.11, early MacOS.
- Otherwise the scheduler is *preemptive*.

  ✔ solves denial of service problem

  ✘ more complicated to implement

  ✘ introduces concurrency problems. . .

Idle system

What do we do if there is no ready process?

- halt processor (until interrupt arrives)
  - ✔ saves power (and heat!)
  - ✔ increases processor lifetime
  - ✘ might take too long to stop and start.
- busy wait in scheduler
  - ✔ quick response time
  - ✘ ugly, useless
- invent idle process, always available to run
  - ✔ gives uniform structure
  - ✔ could use it to run checks
  - ✘ uses some memory
  - ✘ can slow interrupt response

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

Scheduling Criteria

A variety of metrics may be used:

1. CPU utilization: the fraction of the time the CPU is being used (and not for idle process!)
2. Throughput: # of processes that complete their execution per time unit.
3. Turnaround time: amount of time to execute a particular process.
4. Waiting time: amount of time a process has been waiting in the ready queue.
5. Response time: amount of time it takes from when a request was submitted until the first response is produced (in time-sharing systems)

Sensible scheduling strategies might be:

- Maximize throughput or CPU utilization
- Minimize average turnaround time, waiting time or response time.

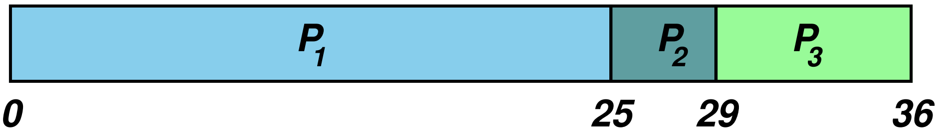
Also need to worry about $fairness$ and $liveness$.

First-Come First-Served Scheduling

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

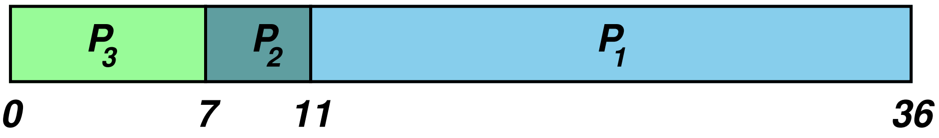
Process	Burst Time
P_1	25
P_2	4
P_3	7

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



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

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



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

- First case poor due to *convoy effect*.

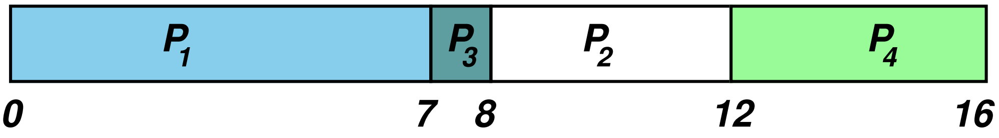
SJF Scheduling

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

- Associate with each process the length of its next CPU burst.
- Use these lengths to schedule the process with the shortest time (FCFS can be used to break ties).

For example:

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



- Waiting time for P_1=0; P_2=6; P_3=3; P_4=7;
- Average waiting time: $(0 + 6 + 3 + 7)/4 = 4$.

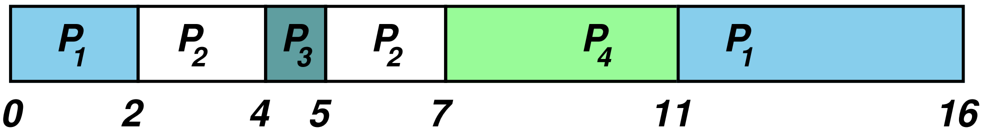
SJF is optimal in that it gives the minimum average waiting time for a given set of processes.

SRTF Scheduling

- SRTF = Shortest Remaining-Time First.
- Just a preemptive version of SJF.
- i.e. if a new process arrives with a CPU burst length less than the *remaining time* of the current executing process, preempt.

For example:

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



- Waiting time for P_1=9; P_2=1; P_3=0; P_4=2;
- Average waiting time: $(9 + 1 + 0 + 2)/4 = 3$.

What are the problems here?

Predicting Burst Lengths

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

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

- If we expand the formula we get:

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

  where τ_0 is some constant.
- Choose value of α according to our belief about the system, e.g. if we believe history irrelevant, choose $\alpha \approx 1$ and then get $\tau_{n+1} \approx t_n$.
- In general an exponential averaging scheme is a good predictor if the variance is small.

Round Robin Scheduling

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

- Process at the front of the ready queue is allocated the CPU for (up to) one quantum.
- When the time has elapsed, the process is preempted and appended to the ready queue.

Round robin has some nice properties:

- Fair: if there are n processes in the ready queue and the time quantum is q, then each process gets $1/n^{th}$ of the CPU.
- Live: no process waits more than $(n-1)q$ time units before receiving a CPU allocation.
- Typically get higher average turnaround time than SRTF, but better average *response time*.

But tricky choosing correct size quantum:

- q too large $\Rightarrow$ FCFS/FIFO
- q too small $\Rightarrow$ context switch overhead too high.

Static Priority Scheduling

- Associate an (integer) priority with each process
- For example:

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

- Then allocate CPU to the highest priority process:
  - ‘highest priority’ typically means smallest integer
  - get preemptive and non-preemptive variants.
- e.g. SJF is a priority scheduling algorithm where priority is the predicted next CPU burst time.
- Problem: how to resolve ties?
  - round robin with time-slicing
  - allocate quantum to each process in turn.
  - Problem: biased towards CPU intensive jobs.
    - ∗ per-process quantum based on usage?
    - ∗ ignore?
- Problem: starvation. . .

Dynamic Priority Scheduling

- Use same scheduling algorithm, but allow priorities to change over time.
- e.g. simple aging:
  - processes have a (static) *base priority* and a dynamic *effective priority*.
  - if process starved for k seconds, increment effective priority.
  - once process runs, reset effective priority.
- e.g. computed priority:
  - first used in Dijkstra's THE
  - time slots: . . . , t, $t + 1$, . . .
  - in each time slot t, measure the CPU usage of process j: u^j
  - priority for process j in slot $t + 1$: $p^j_{t+1} = f(u^j_t, p^j_t, u^j_{t-1}, p^j_{t-1}, \ldots)$
  - e.g. $p^j_{t+1} = p^j_t/2 + ku^j_t$
  - penalises CPU bound → supports I/O bound.
- today such computation considered acceptable. . .

Memory Management

In a multiprogramming system:

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

⇒ must share memory between OS and k processes.

The memory magagement subsystem handles:

1. Relocation
2. Allocation
3. Protection
4. Sharing
5. Logical Organisation
6. Physical Organisation

The Address Binding Problem

Consider the following simple program:

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

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

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

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

Then the address binding problem is:

what values do we give Rx and Ry ?

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

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

Address Binding and Relocation

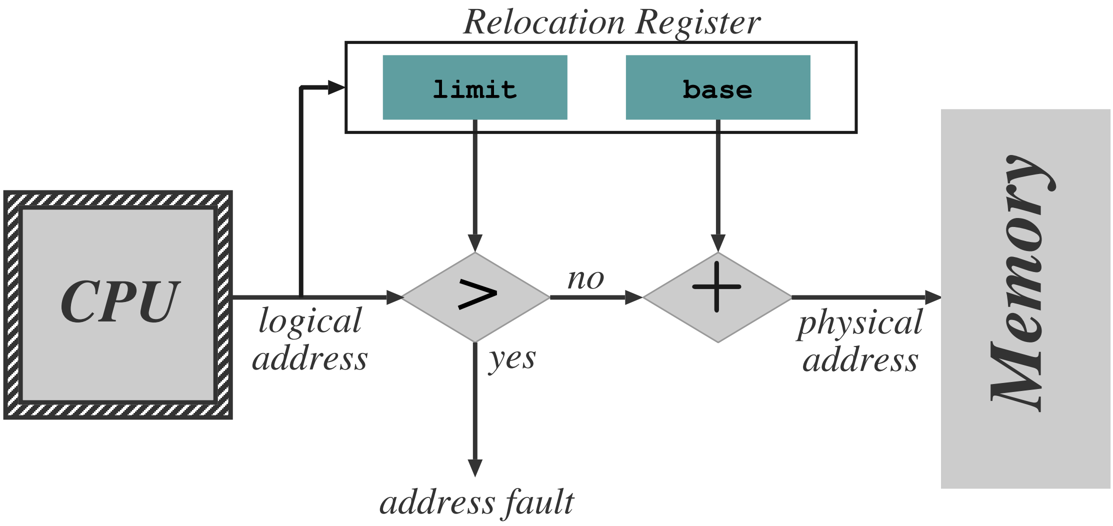
To solve the problem, we need to translate between "program addresses" and "real addresses".

This can be done:

- at compile time:
  - requires knowledge of absolute addresses
  - e.g. DOS .com files
- at load time:
  - when program loaded, work out position in memory and update code with correct addresses
  - must be done every time program is loaded
  - ok for embedded systems / boot-loaders
- at run-time:
  - get some hardware to automatically translate between program and real addresses.
  - no changes at all required to program itself.
  - most popular and flexible scheme, providing we have the requisite hardware (MMU).

Logical vs Physical Addresses

Mapping of logical to physical addresses is done at run-time by Memory Management Unit (MMU), e.g.



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

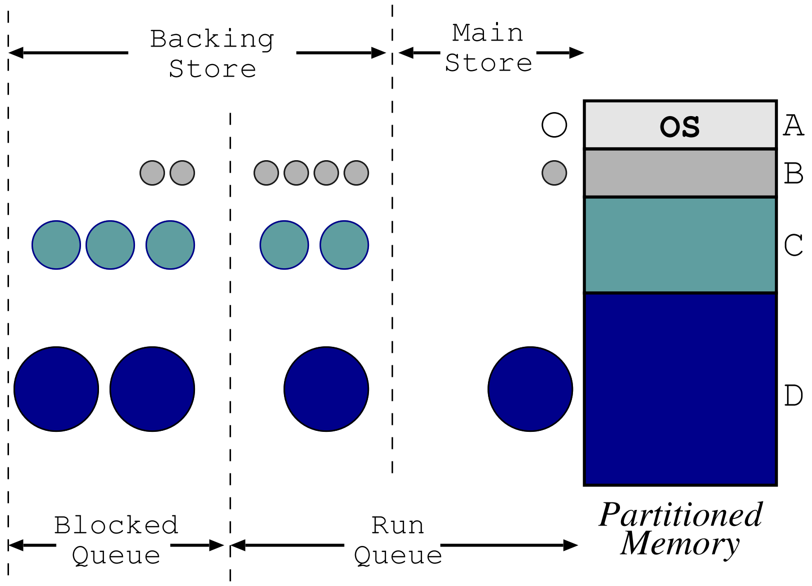
Contiguous Allocation

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

Where do we put processes in memory?

- OS typically must be in low memory due to location of interrupt vectors
- Easiest way is to statically divide memory into multiple fixed size partitions:
  - bottom partition contains OS, remaining partitions each contain exactly one process.
  - when a process terminates its partition becomes available to new processes.
  - e.g. OS/360 MFT.
- Need to protect OS and user processes from malicious programs:
  - use base and limit registers in MMU
  - update values when a new processes is scheduled
  - NB: solving both relocation and protection problems at the same time!

Static Multiprogramming

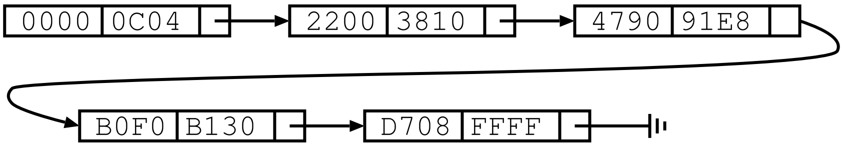


- partition memory when installing OS, and allocate pieces to different job queues.
- associate jobs to a job queue according to size.
- swap job back to disk when:
  - blocked on I/O (assuming I/O is slower than the backing store).
  - time sliced: larger the job, larger the time slice
- run job from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

Dynamic Partitioning

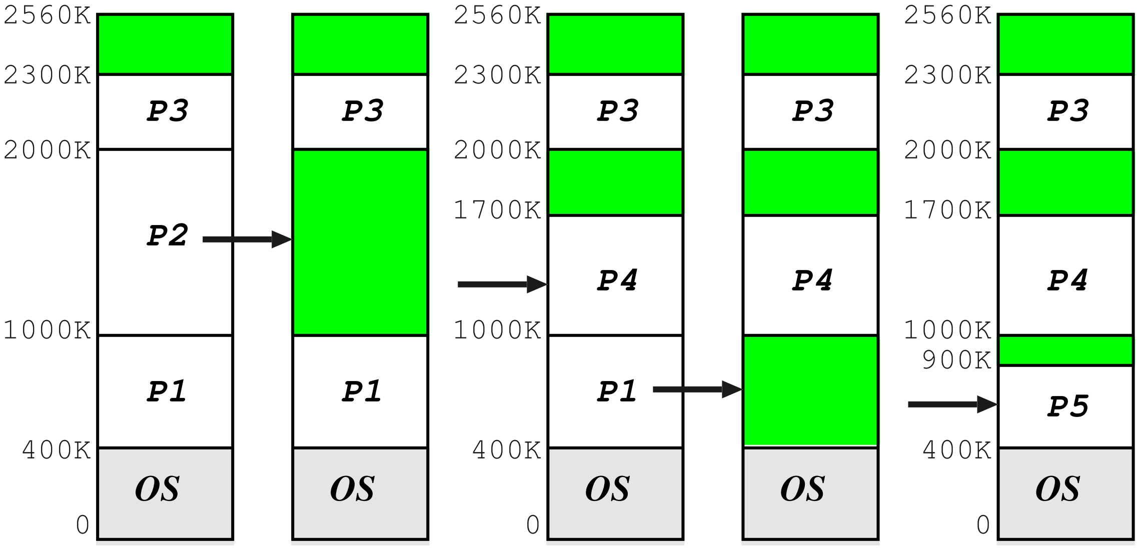
Get more flexibility if allow partition sizes to be dynamically chosen (e.g. OS/360 MVT) :

- OS keeps track of which areas of memory are available and which are occupied.
- e.g. use one or more *linked lists*:



- When a new process arrives the OS searches for a hole large enough to fit the process.
- To determine which hole to use for new process:
  - **first fit**: stop searching list as soon as big enough hole is found.
  - **best fit**: search entire list to find "best" fitting hole (i.e. smallest hole large enough)
  - **worst fit**: counterintuitively allocate largest hole (again must search entire list).
- When process terminates its memory returns onto the free list, coalescing holes where appropriate.

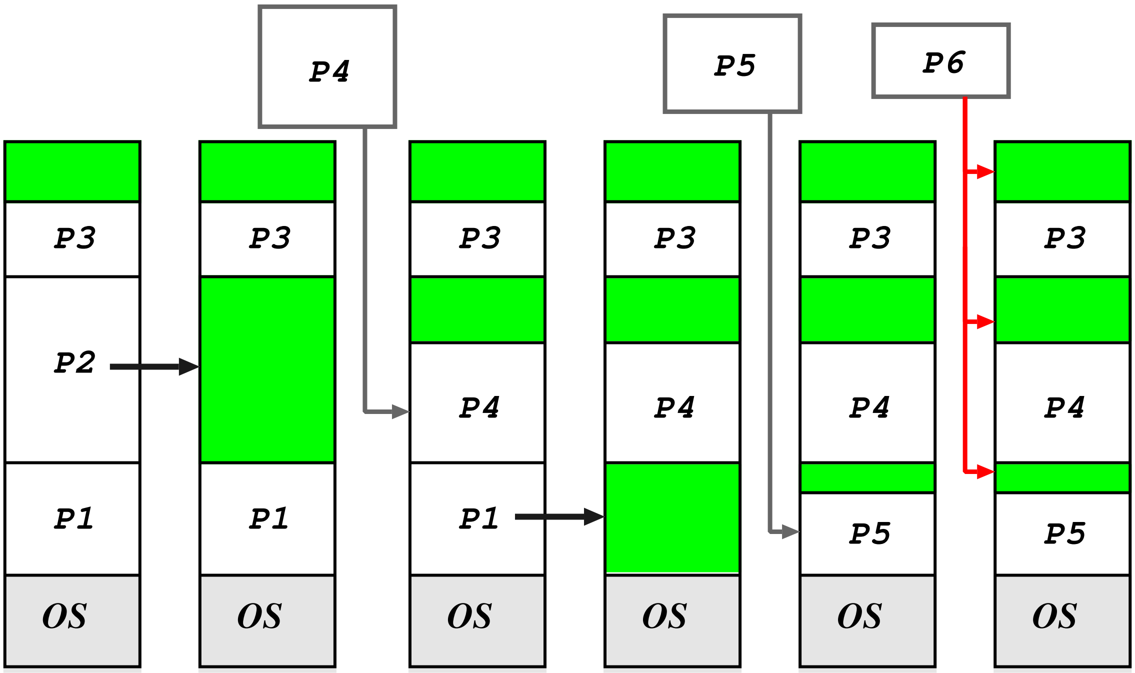
Scheduling Example



- Consider machine with total of 2560K memory.
- Operating System requires 400K.
- The following jobs are in the queue:

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

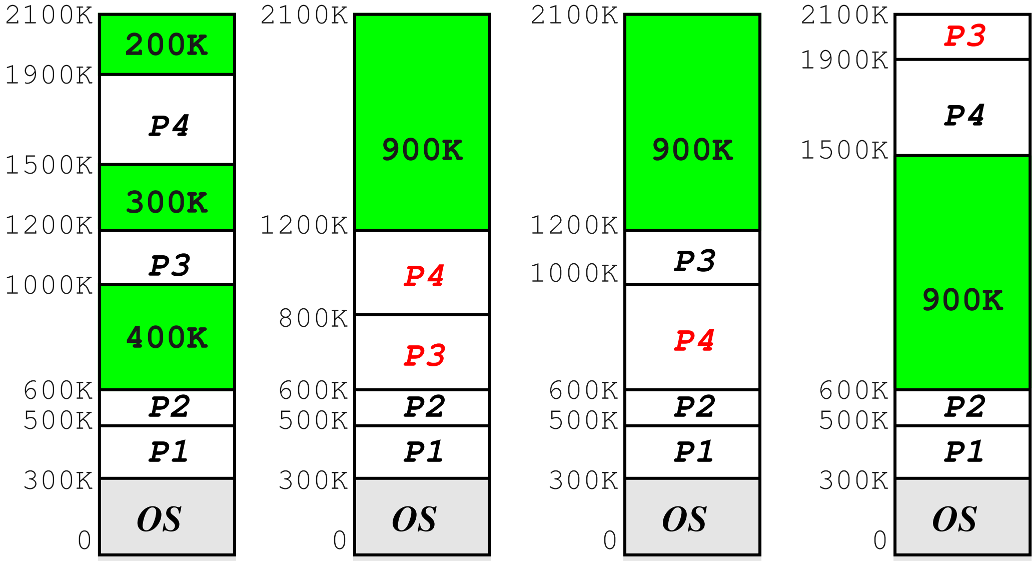
External Fragmentation



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

Solution: compact holes periodically.

Compaction



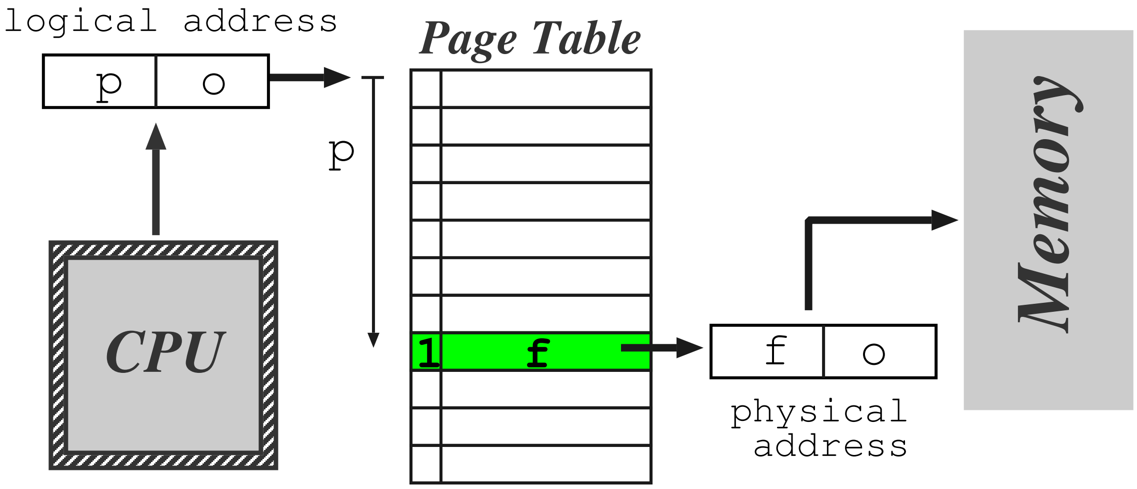
Choosing optimal strategy quite tricky. . .

Note that:

- Require run-time relocation.
- Can be done more efficiently when process is moved into memory from a swap.
- Some machines used to have hardware support (e.g. CDC Cyber).

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

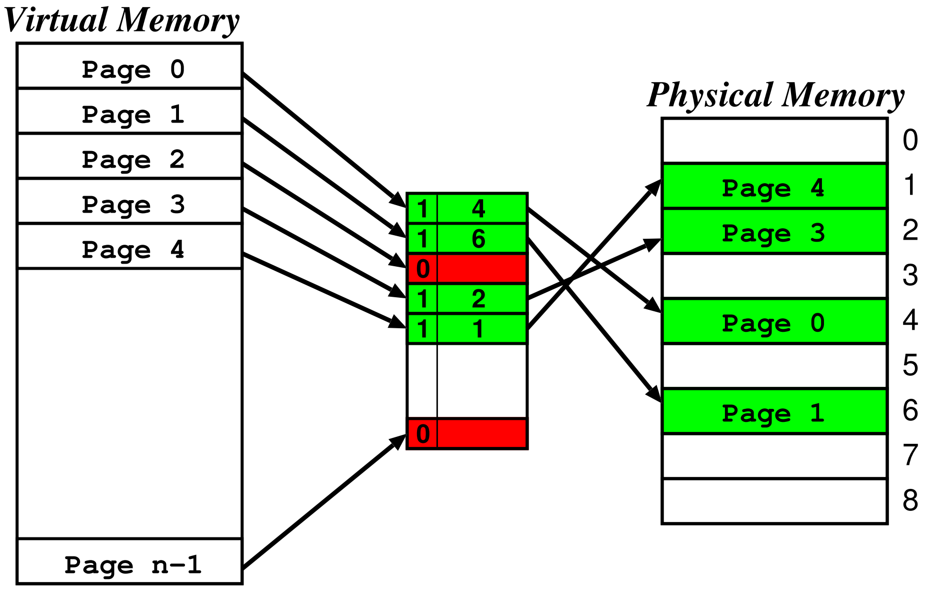
Paged Virtual Memory



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

- divide physical memory into relatively small blocks of fixed size, called *frames*
- divide logical memory into blocks of the same size called *pages* (typical value is 4K)
- each address generated by CPU is composed of a page number p and page offset o.
- MMU uses p as an index into a *page table*.
- page table contains associated frame number f
- usually have $|\texttt{p}| \gg |\texttt{f}| \Rightarrow$ need valid bit.

Paging Pros and Cons



- ✔ memory allocation easier.
- ✘ OS must keep page table per process
- ✔ no external fragmentation (in physical memory at least).
- ✘ but get **internal fragmentation**.
- ✔ clear separation between user and system view of memory usage.
- ✘ additional overhead on context switching

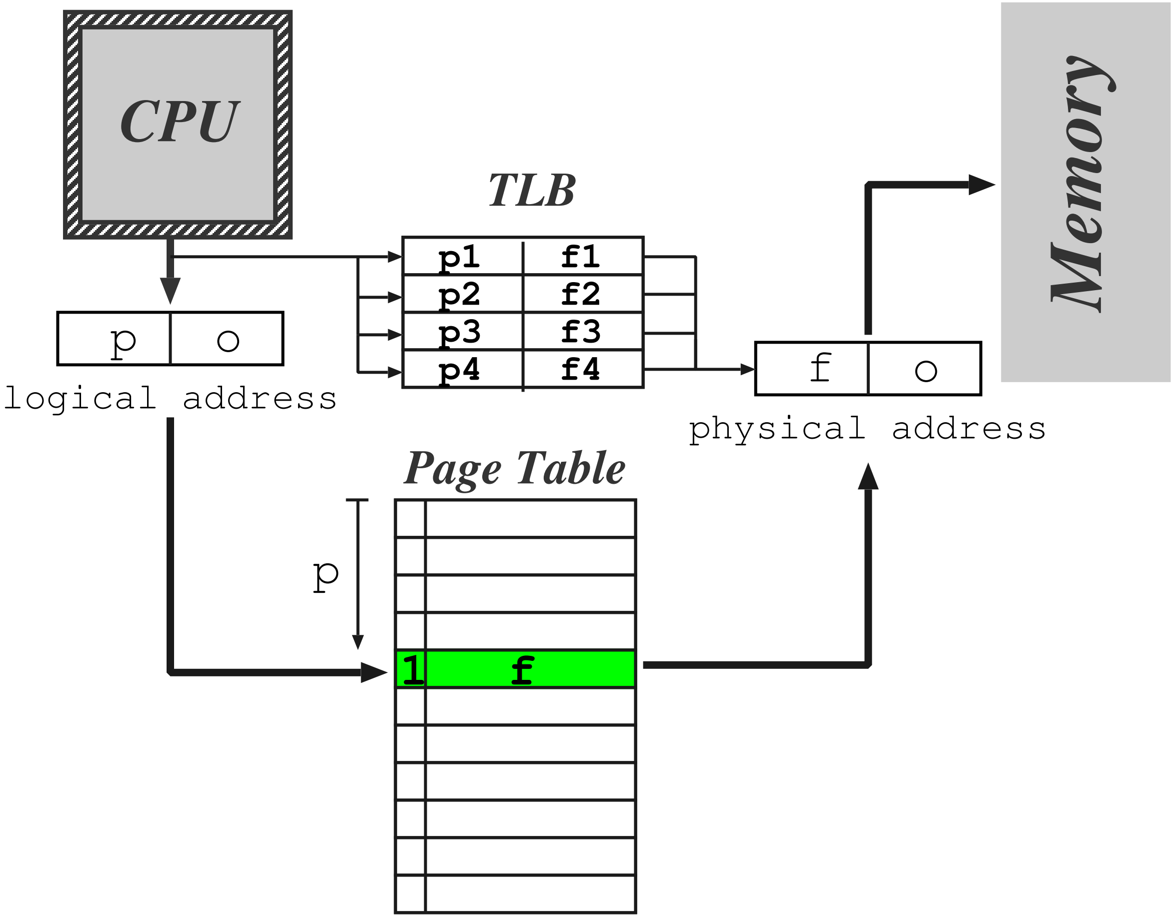
Structure of the Page Table

Different kinds of hardware support can be provided:

- Simplest case: set of dedicated relocation registers
  - one register per page
  - OS loads the registers on context switch
  - fine if the page table is small. . . but what if have large number of pages ?
- Alternatively keep page table in memory
  - only one register needed in MMU (page table base register (PTBR))
  - OS switches this when switching process
- Problem: page tables might still be very big.
  - can keep a page table length register (PTLR) to indicate size of page table.
  - or can use more complex structure (see later)
- Problem: need to refer to memory *twice* for every 'actual' memory reference. . .

  ⇒ use a translation lookaside buffer (TLB)

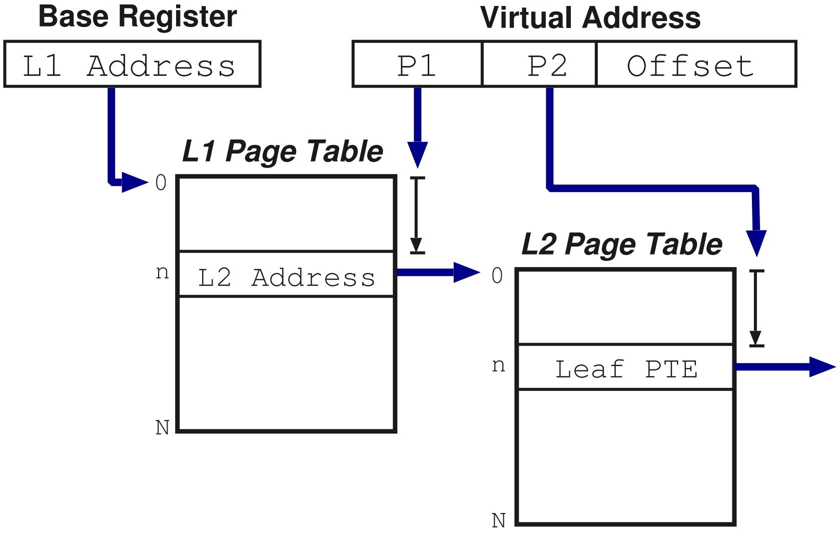
TLB Operation



- On memory reference present TLB with logical memory address
- If page table entry for the page is present then get an immediate result
- If not then make memory reference to page tables, and update the TLB

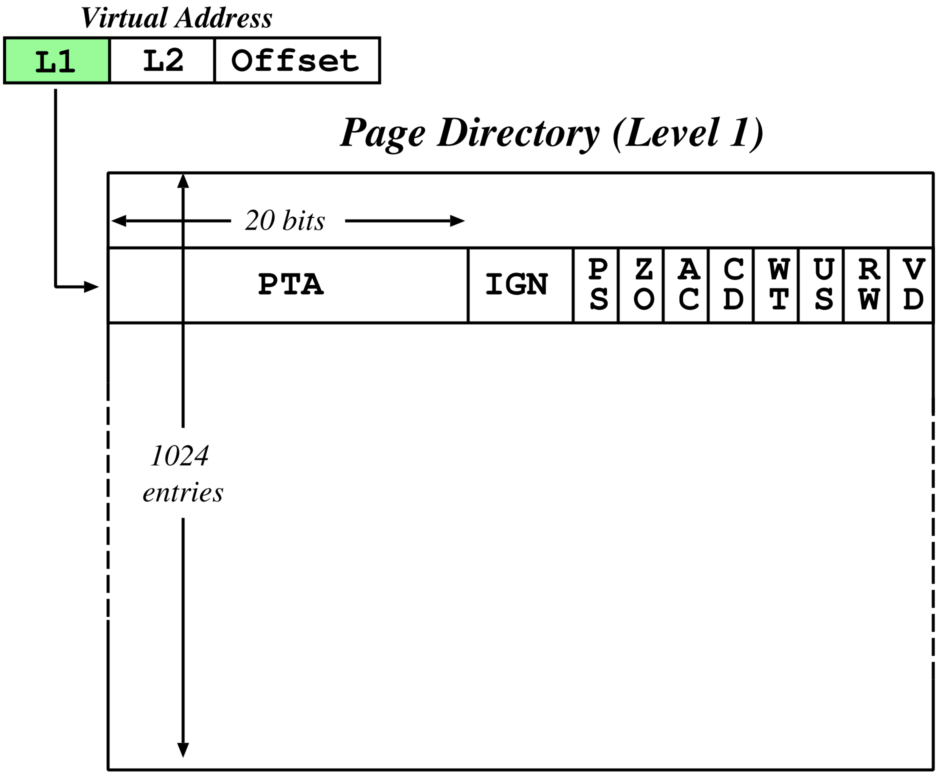
Multilevel Page Tables

- Most modern systems can support very large $(2^{32}, 2^{64})$ address spaces.
- Solution – split page table into several sub-parts
- Two level paging – page the page table



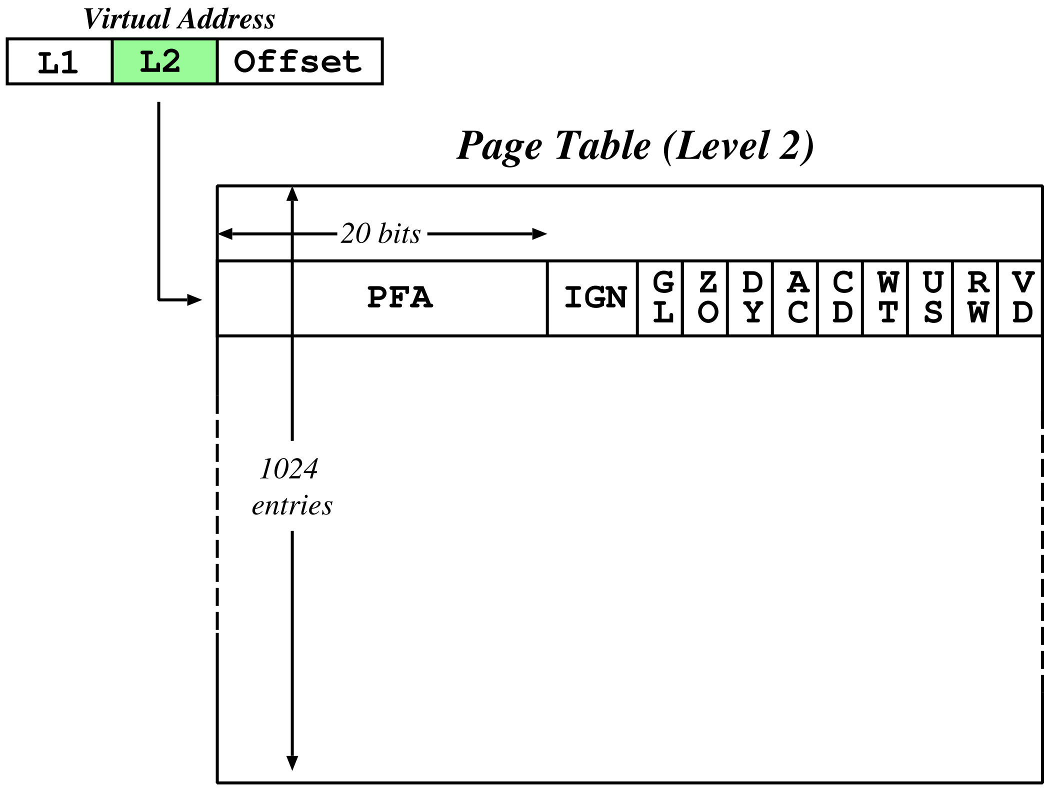
- For 64 bit architectures a two-level paging scheme is not sufficient: need further levels.
- (even some 32 bit machines have > 2 levels).

Example: x86



- Page size 4K (or 4Mb).
- First lookup is in the *page directory*: index using most 10 significant bits.
- Address of page directory stored in internal processor register (`cr3`).
- Results (normally) in the address of a *page table*.

Example: x86 (2)



- Use next 10 bits to index into page table.
- Once retrieve page frame address, add in the offset (i.e. the low 12 bits).
- Notice page directory and page tables are exactly one page each themselves.

Protection Issues

- Associate protection bits with each page – kept in page tables (and TLB).
- e.g. one bit for read, one for write, one for execute.
- May also distinguish whether may only be accessed when executing in *kernel mode*, e.g.

Frame Number	**K**	**R**	**W**	**X**	**V**

- At the same time as address is going through page hardware, can check protection bits.
- Attempt to violate protection causes h/w trap to operating system code
- As before, have *valid/invalid* bit determining if the page is mapped into the process address space:
  - if invalid ⇒ trap to OS handler
  - can do lots of interesting things here, particularly with regard to sharing. . .

Shared Pages

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

- binaries: editor, compiler etc.
- libraries: shared objects, dlls.

So how does this work?

- Implemented as two logical addresses which map to one physical address.
- If code is *re-entrant* (i.e. stateless, non-self modifying) it can be easily shared between users.
- Otherwise can use *copy-on-write* technique:
  - mark page as read-only in all processes.
  - if a process tries to write to page, will trap to OS fault handler.
  - can then allocate new frame, copy data, and create new page table mapping.
- (may use this for lazy data sharing too).

Requires additional book-keeping in OS, but worth it, e.g. over 40Mb of shared code on my linux box.

Virtual Memory

- Virtual addressing allows us to introduce the idea of *virtual memory*:
  - already have valid or invalid page translations; introduce new “non-resident” designation
  - such pages live on a non-volatile *backing store*
  - processes access non-resident memory just as if it were ‘the real thing’.
- Virtual memory (VM) has a number of benefits:
  - *portability*: programs work regardless of how much actual memory present
  - *convenience*: programmer can use e.g. large sparse data structures with impunity
  - *efficiency*: no need to waste (real) memory on code or data which isn’t used.
- VM typically implemented via *demand paging*:
  - programs (executables) reside on disk
  - to execute a process we load pages in *on demand*; i.e. as and when they are referenced.
- Also get *demand segmentation*, but rare.

Demand Paging Details

When loading a new process for execution:

- create its address space (e.g. page tables, etc)
- mark PTEs as either "invalid or "non-resident"
- add PCB to scheduler.

Then whenever we receive a *page fault*:

1. check PTE to determine if "invalid" or not
2. if an invalid reference ⇒ kill process;
3. otherwise 'page in' the desired page:
    - find a free frame in memory
    - initiate disk I/O to read in the desired page
    - when I/O is finished modify the PTE for this page to show that it is now valid
    - restart the process at the faulting instruction

Scheme described above is *pure* demand paging:

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

Page Replacement

- When paging in from disk, we need a free frame of physical memory to hold the data we're reading in.
- In reality, size of physical memory is limited ⇒
  - need to discard unused pages if total demand for pages exceeds physical memory size
  - (alternatively could swap out a whole process to free some frames)
- Modified algorithm: on a page fault we
  1. locate the desired replacement page on disk
  2. to select a free frame for the incoming page:
     (a) if there is a free frame use it
     (b) otherwise select a *victim page* to free,
     (c) write the victim page back to disk, and
     (d) mark it as invalid in its process page tables
  3. read desired page into freed frame
  4. restart the faulting process
- Can reduce overhead by adding a 'dirty' bit to PTEs (can potentially omit step 2c above)
- Question: how do we choose our victim page?

Page Replacement Algorithms

- First-In First-Out (FIFO)
  - keep a queue of pages, discard from head
  - performance difficult to predict: no idea whether page replaced will be used again or not
  - discard is independent of page use frequency
  - in general: pretty bad, although very simple.
- Optimal Algorithm (OPT)
  - replace the page which will not be used again for longest period of time
  - can only be done with an oracle, or in hindsight
  - serves as a good comparison for other algorithms
- Least Recently Used (LRU)
  - LRU replaces the page which has not been used for the longest amount of time
  - (i.e. LRU is OPT with -ve time)
  - assumes past is a good predictor of the future
  - Q: how do we determine the LRU ordering?

Implementing LRU

- Could try using *counters*
  - give each page table entry a time-of-use field and give CPU a logical clock (counter)
  - whenever a page is referenced, its PTE is updated to clock value
  - replace page with smallest time value
  - problem: requires a search to find min value
  - problem: adds a write to memory (PTE) on every memory reference
  - problem: clock overflow
- Or a *page stack*:
  - maintain a stack of pages (doubly linked list) with most-recently used (MRU) page on top
  - discard from bottom of stack
  - requires changing 6 pointers per [new] reference
  - very slow without extensive hardware support
- Neither scheme seems practical on a standard processor ⇒ need another way.

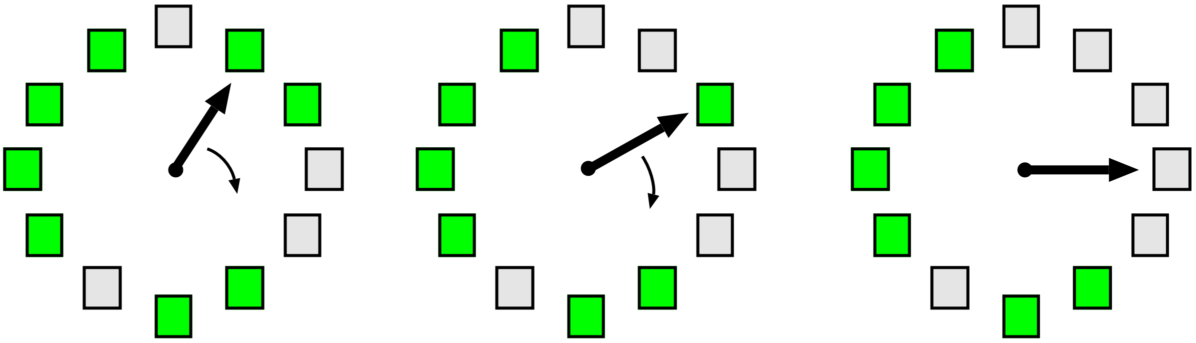
Approximating LRU (1)

- Many systems have a *reference bit* in the PTE which is set by h/w whenever the page is touched
- This allows *not recently used* (NRU) replacement:
  - periodically (e.g. 20ms) clear all reference bits
  - when choosing a victim to replace, prefer pages with clear reference bits
  - if also have a *modified bit* (or *dirty bit*) in the PTE, can extend MRU to use that too:

Ref?	Dirty?	Comment
no	no	best type of page to replace
no	yes	next best (requires writeback)
yes	no	probably code in use
yes	yes	bad choice for replacement

- Or can extend by maintaining more history, e.g.
  - for each page, the operating system maintains an 8-bit value, initialized to zero
  - periodically (e.g. 20ms) shift reference bit onto high order bit of the byte, and clear reference bit
  - select lowest value page (or one of) to replace

Approximating LRU (2)

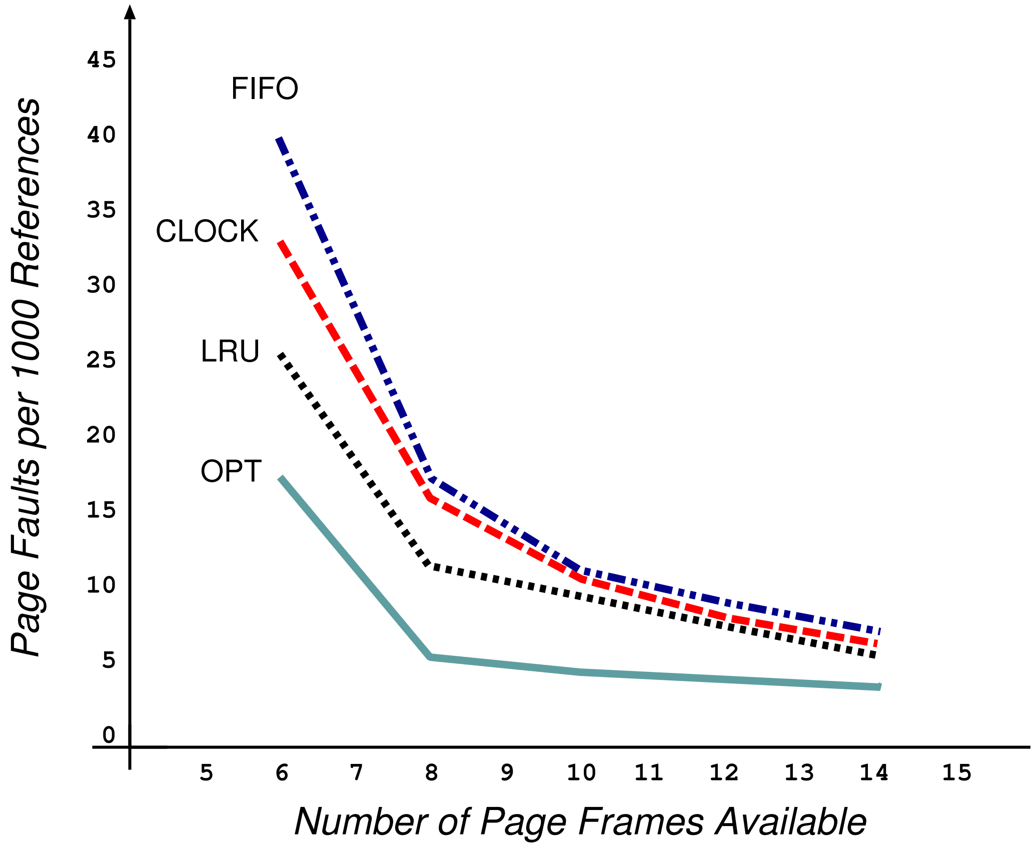


- Popular NRU scheme: *second-chance FIFO*
  - store pages in queue as per FIFO
  - before discarding head, check its reference bit
  - if reference bit is 0, discard, otherwise:
    - reset reference bit, and
    - add page to tail of queue
    - i.e. give it "a second chance"
- Often implemented with a circular queue and a current pointer; in this case usually called *clock*.
- If no h/w provided reference bit can emulate:
  - to clear "reference bit", mark page no access
  - if referenced ⇒ trap, update PTE, and resume
  - to check if referenced, check permissions
  - can use similar scheme to emulate modified bit

Other Replacement Schemes

- Counting Algorithms: keep a count of the number of references to each page
  - LFU: replace page with smallest count
  - MFU: replace highest count because low count $\Rightarrow$ most recently brought in.
- Page Buffering Algorithms:
  - keep a min. number of victims in a free pool
  - new page read in before writing out victim.
- (Pseudo) MRU:
  - consider access of e.g. large array.
  - page to replace is one application has *just finished with*, i.e. most recently used.
  - e.g. track page faults and look for sequences.
  - discard the k^{th} in victim sequence.
- Application-specific:
  - stop trying to second guess what's going on.
  - provide hook for app. to suggest replacement.
  - must be careful with denial of service. . .

Performance Comparison



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

- want to minimise area under curve
- FIFO can exhibit Belady's anomaly (although it doesn't in this case)
- getting frame allocation right has major impact. . .

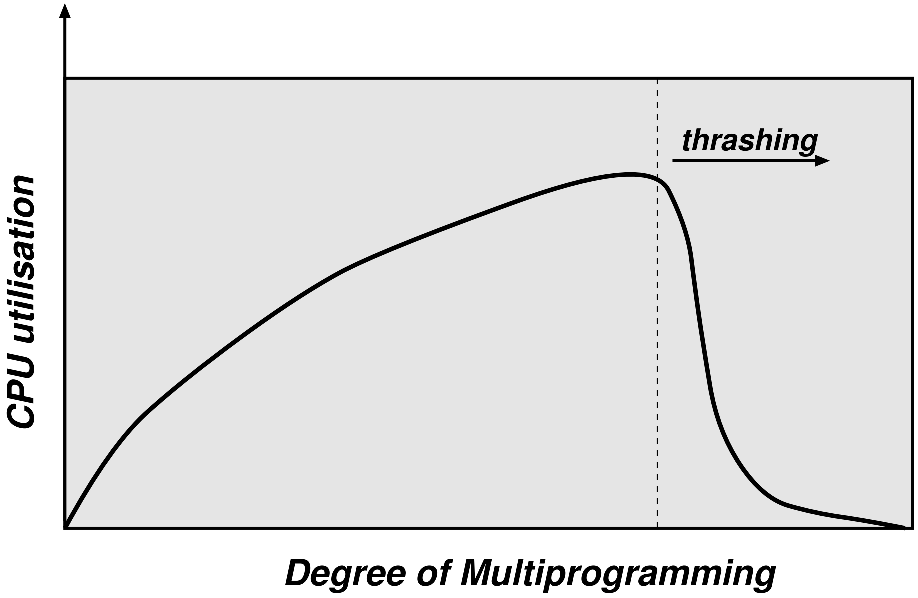
Frame Allocation

- A certain fraction of physical memory is reserved per-process and for core OS code and data.
- Need an *allocation policy* to determine how to distribute the remaining frames.
- Objectives:
  - Fairness (or proportional fairness)?
    - ∗ e.g. divide m frames between n processes as m/n, with remainder in the free pool
    - ∗ e.g. divide frames in proportion to size of process (i.e. number of pages used)
  - Minimize system-wide page-fault rate?
    (e.g. allocate all memory to few processes)
  - Maximize level of multiprogramming?
    (e.g. allocate min memory to many processes)
- Most page replacement schemes are *global*: all pages considered for replacement.

⇒ allocation policy implicitly enforced during page-in:

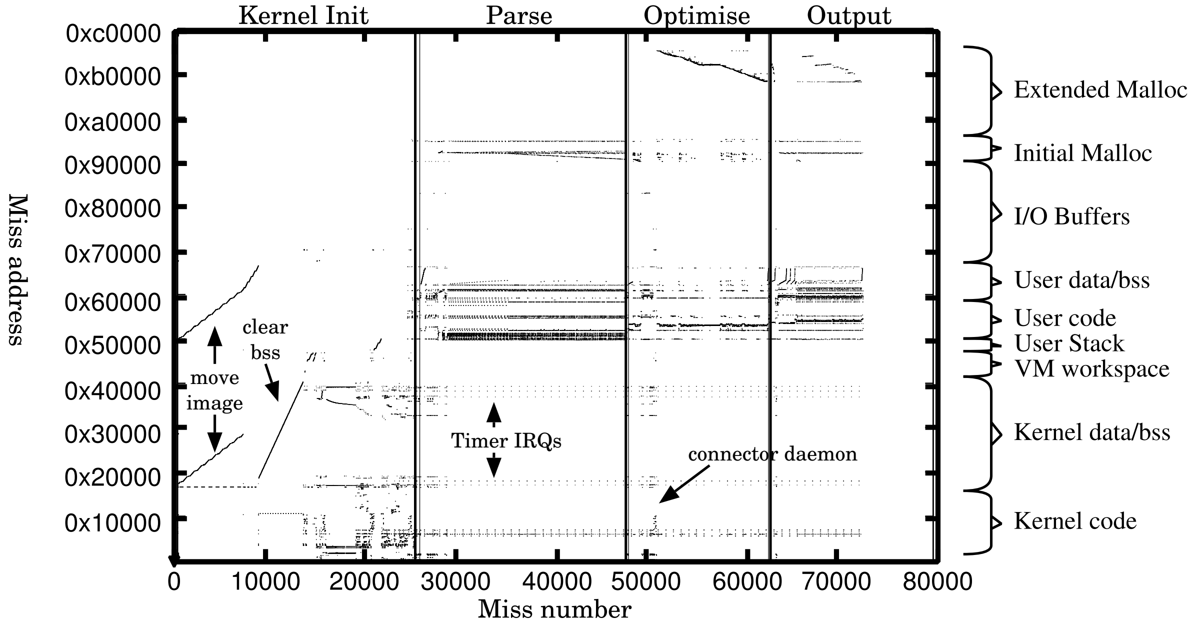
- allocation succeeds iff policy agrees
- 'free frames' often in use ⇒ steal them!

The Risk of Thrashing



- As more processes enter the system, the frames-per-process value can get very small.
- At some point we hit a wall:
  - a process needs more frames, so steals them
  - but the other processes need those pages, so they fault to bring them back in
  - number of runnable processes plunges
- To avoid thrashing we must give processes as many frames as they “need”
- If we can’t, we need to reduce the MPL (a better page-replacement algorithm will not help)

Locality of Reference



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

- procedure being executed
- . . . sub-procedures
- . . . data access
- . . . stack variables

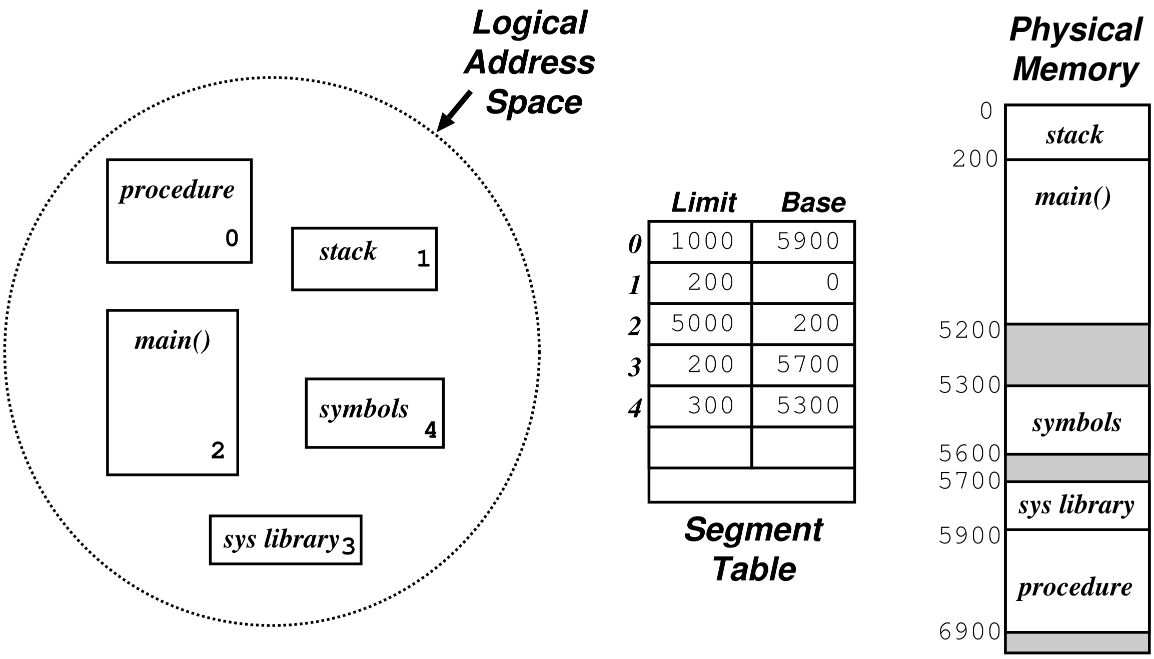
Note: have locality in both space and time.

Avoiding Thrashing

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

- define the *Working Set* (Denning, 1967)
  - set of pages that a process needs in store at "the same time" to make any progress
  - varies between processes and during execution
  - assume process moves through $phases$
  - in each phase, get (spatial) locality of reference
  - from time to time get $phase\ shift$
- Then OS can try to prevent thrashing by maintaining sufficient pages for current phase:
  - sample page reference bits every e.g. 10ms
  - if a page is "in use", say it's in the working set
  - sum working set sizes to get total demand D
  - if $D > m$ we are in danger of thrashing $\Rightarrow$ suspend a process
- Alternatively use page fault frequency (PFF):
  - monitor per-process page fault rate
  - if too high, allocate more frames to process

Segmentation



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

Segment Table

- User prefers to view memory as a set of segments of no particular size, with no particular ordering
- Segmentation supports this user-view of memory — logical address space is a collection of (typically disjoint) segments.
- Segments have a name (or a number) and a length — addresses specify segment and offset.
- Contrast with paging where user is unaware of memory structure (all managed invisibly).

Implementing Segments

- Maintain a segment table for each process:

Segment	Access	Base	Size	Others!

- If program has a very large number of segments then the table is kept in memory, pointed to by ST base register STBR
- Also need a ST length register STLR since number of segs used by different programs will differ widely
- The table is part of the process context and hence is changed on each process switch.

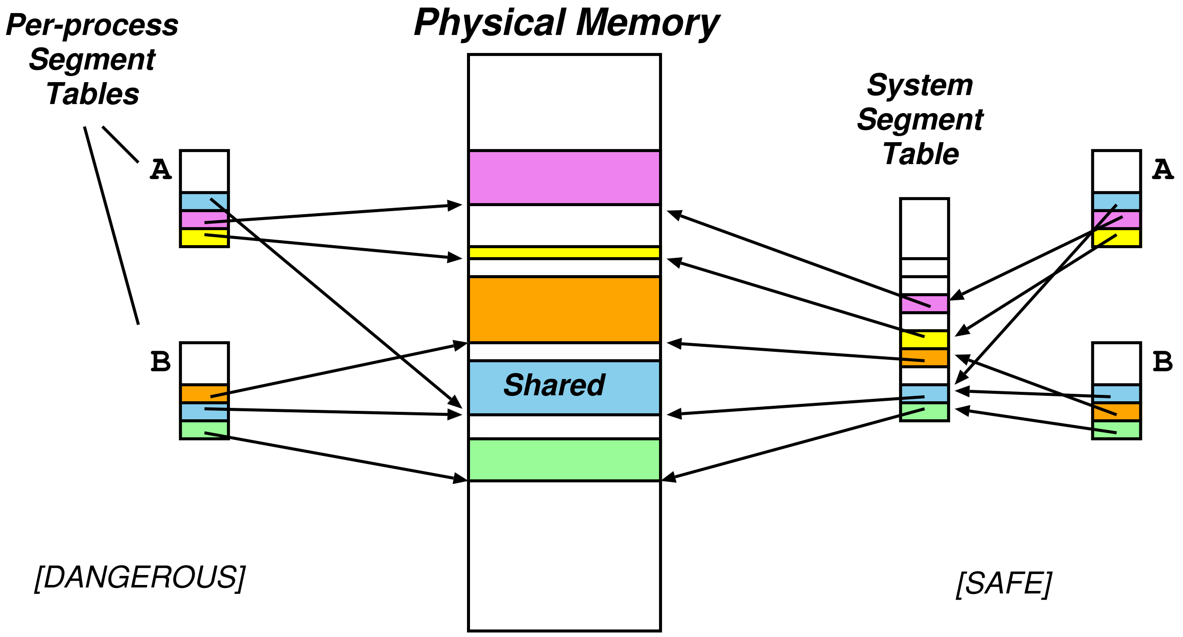
Algorithm:

1. Program presents address (s, d).
   Check that $s <$ STLR. If not, fault
2. Obtain table entry at reference $s+$ STBR, a tuple of form (b_s, l_s)
3. If $0 \leq d < l_s$ then this is a valid address at location (b_s, d), else fault

Sharing and Protection

- Big advantage of segmentation is that protection is per segment; i.e. corresponds to logical view.
- Protection bits associated with each ST entry checked in usual way
- e.g. instruction segments (should be non-self modifying!) thus protected against writes etc.
- e.g. place each array in own seg ⇒ array limits checked by hardware
- Segmentation also facilitates sharing of code/data
  - each process has its own STBR/STLR
  - sharing is enabled when two processes have entries for the same physical locations.
  - for data segments can use copy-on-write as per paged case.
- Several subtle caveats exist with segmentation — e.g. jumps within shared code.

Sharing Segments



Sharing segments:

- wasteful (and dangerous) to store common information on shared segment in each process segment table
- assign each segment a unique System Segment Number (SSN)
- process segment table simply maps from a Process Segment Number (PSN) to SSN

External Fragmentation Returns. . .

- Long term scheduler must find spots in memory for all segments of a program.
- Problem now is that segs are of variable size $\Rightarrow$ leads to fragmentation.
- Tradeoff between compaction/delay depends on average segment size
- Extremes: each process 1 seg — reduces to variable sized partitions
- Or each byte one seg separately relocated — quadruples memory use!
- Fixed size small segments $\equiv$ paging!
- In general with small average segment sizes, external fragmentation is small.

Segmentation versus Paging

	logical view	allocation
Segmentation	✔	✘
Paging	✘	✔

⇒ try combined scheme.

- E.g. *paged segments* (Multics, OS/2)
  - divide each segment s_i into $k = \lceil l_i/2^n \rceil$ pages, where l_i is the limit (length) of the segment.
  - have page table per segment.

  ✘ high hardware cost / complexity.

  ✘ not very portable.

- E.g. *software segments* (most modern OSs)
  - consider pages $[m, \ldots, m+l]$ to be a segment.
  - OS must ensure protection / sharing kept consistent over region.

  ✘ loss in granularity.

  ✔ relatively simple / portable.

Summary (1 of 2)

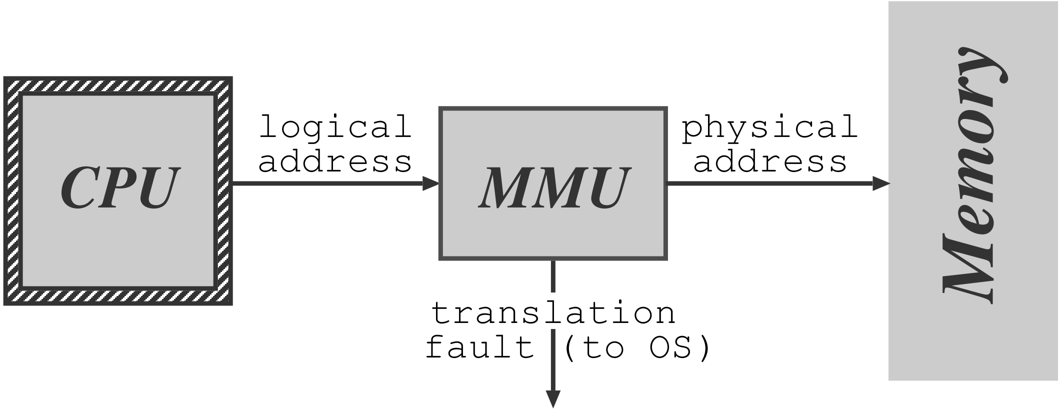
Old systems directly accessed [physical] memory, which caused some problems, e.g.

- Contiguous allocation:
  - need large lump of memory for process
  - with time, get [external] fragmentation

  ⇒ require expensive compaction
- Address binding (i.e. dealing with *absolute* addressing):
  - "`int x; x = 5;`" → "`movl $0x5, ????`"
  - compile time ⇒ must know load address.
  - load time ⇒ work every time.
  - what about swapping?
- Portability:
  - how much memory should we assume a "standard" machine will have?
  - what happens if it has less? or more?

Can avoid lots of problems by separating concept of *logical* (or virtual) and *physical* addresses.

Summary (2 of 2)



Run time mapping from logical to physical addresses performed by special hardware (the MMU).

If we make this mapping a *per process* thing then:

- Each process has own *address space*.
- Allocation problem split:
  - virtual address allocation easy.
  - allocate physical memory ‘behind the scenes’.
- Address binding solved:
  - bind to logical addresses at compile-time.
  - bind to real addresses at load time/run time.

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

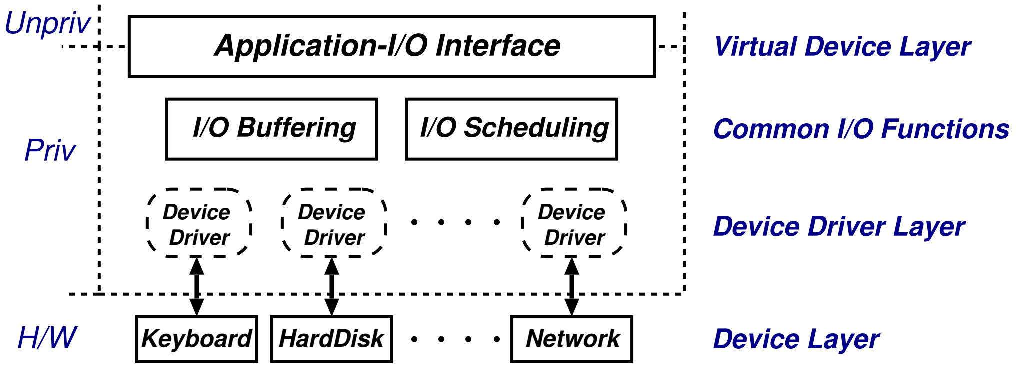
I/O Hardware

- Wide variety of ‘devices’ which interact with the computer via I/O, e.g.
  - Human readable: graphical displays, keyboard, mouse, printers
  - Machine readable: disks, tapes, CD, sensors
  - Communications: modems, network interfaces
- They differ significantly from one another with regard to:
  - Data rate
  - Complexity of control
  - Unit of transfer
  - Direction of transfer
  - Data representation
  - Error handling

⇒ difficult to present a uniform I/O system which hides all the complexity.

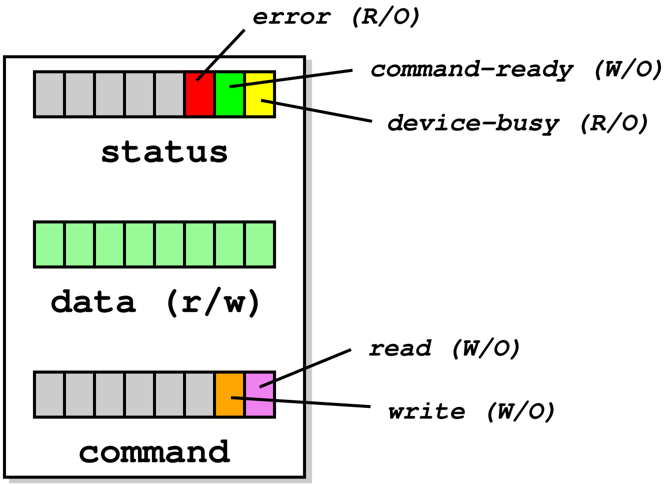
I/O subsystem is generally the ‘messiest’ part of OS.

I/O Subsystem



- Programs access *virtual devices*:
  - terminal streams not terminals
  - windows not frame buffer
  - event stream not raw mouse
  - files not disk blocks
  - printer spooler not parallel port
  - transport protocols not raw ethernet
- OS deals with processor–device interface:
  - I/O instructions versus memory mapped
  - I/O hardware type (e.g. 10's of serial chips)
  - polled versus interrupt driven
  - processor interrupt mechanism

Polled Mode I/O



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

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

- What's the problem here?

Interrupts Revisited

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

- at end of each instruction, processor checks interrupt line(s) for pending interrupt
- if line is asserted then processor:
  - saves program counter,
  - saves processor status,
  - changes processor mode, and
  - jump to well known address (or its contents)
- after interrupt-handling routine is finished, can use e.g. the `rti` instruction to resume.

Some more complex processors provide:

- multiple levels of interrupts
- hardware vectoring of interrupts
- mode dependent registers

Interrupt-Driven I/O

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

- Interrupt handler (processor-dependent) may:
  - save more registers.
  - establish a language environment.
  - demultiplex interrupt in software.
  - invoke appropriate interrupt service routine (ISR)
- Then ISR (device- not processor-specific) will:
  1. for programmed I/O device:
     - transfer data.
     - clear interrupt (sometimes a side effect of tx).
  1. for DMA device:
     - acknowledge transfer.
  2. request another transfer if there are any more I/O requests pending on device.
  3. signal any waiting processes.
  4. enter scheduler or return.

Question: who is scheduling who?

Device Classes

Homogenising device API completely not possible
⇒ OS generally splits devices into four *classes*:

1. Block devices (e.g. disk drives, CD):
   - commands include `read`, `write`, `seek`
   - raw I/O or file-system access
   - memory-mapped file access possible
2. Character devices (e.g. keyboards, mice, serial):
   - commands include `get`, `put`
   - libraries layered on top to allow line editing
3. Network Devices
   - varying enough from block and character to have own interface
   - Unix and Windows/NT use *socket* interface
4. Miscellaneous (e.g. clocks and timers)
   - provide current time, elapsed time, timer
   - `ioctl` (on UNIX) covers odd aspects of I/O such as clocks and timers.

I/O Buffering

- Buffering: OS stores (a copy of) data in memory while transferring between devices
  - to cope with device speed mismatch
  - to cope with device transfer size mismatch
  - to maintain “copy semantics”
- OS can use various kinds of buffering:
  1. single buffering — OS assigns a system buffer to the user request
  2. double buffering — process consumes from one buffer while system fills the next
  3. circular buffers — most useful for bursty I/O
- Many aspects of buffering dictated by device type:
  - character devices ⇒ line probably sufficient.
  - network devices ⇒ bursty (time & space).
  - block devices ⇒ lots of fixed size transfers.
  - (last usually major user of buffer memory)

Blocking v. Nonblocking I/O

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

1. Blocking: process suspended until I/O completed
   - easy to use and understand.
   - insufficient for some needs.
2. Nonblocking: I/O call returns as much as available
   - returns almost immediately with count of bytes read or written (possibly 0).
   - can be used by e.g. user interface code.
   - essentially application-level "polled I/O".
3. Asynchronous: process runs while I/O executes
   - I/O subsystem explicitly signals process when its I/O request has completed.
   - most flexible (and potentially efficient).
   - . . . but also most difficult to use.

Most systems provide both blocking and non-blocking I/O interfaces; fewer support asynchronous I/O.

Other I/O Issues

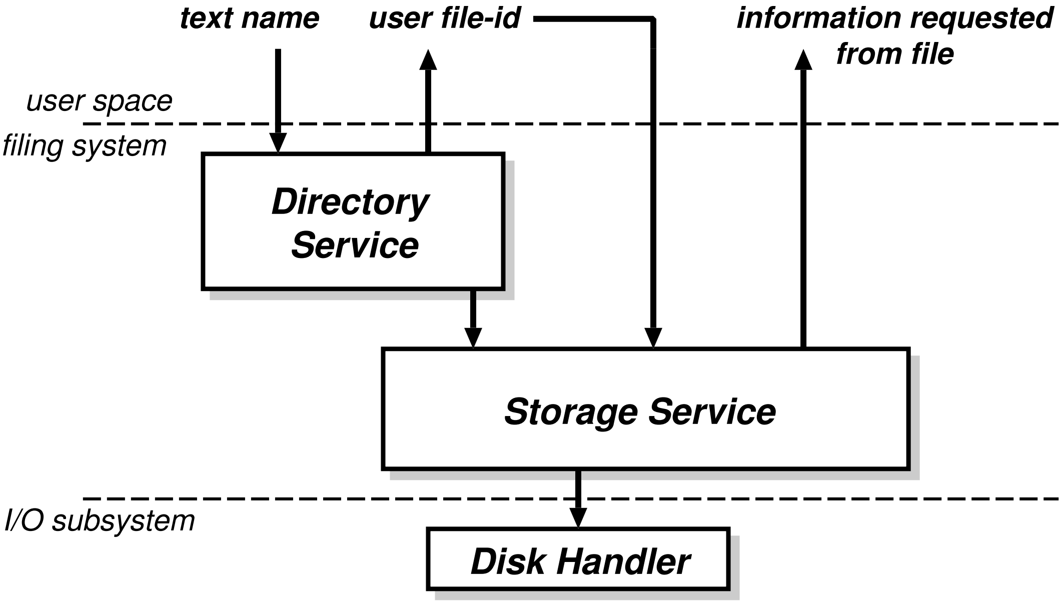
- Caching: fast memory holding copy of data
  - can work with both reads and writes
  - key to I/O performance
- Scheduling:
  - e.g. ordering I/O requests via per-device queue
  - some operating systems try fairness. . .
- Spooling: queue output for a device
  - useful if device is "single user" (i.e. can serve only one request at a time), e.g. printer.
- Device reservation:
  - system calls for acquiring or releasing exclusive access to a device (care required)
- Error handling:
  - e.g. recover from disk read, device unavailable, transient write failures, etc.
  - most I/O system calls return an error number or code when an I/O request fails
  - system error logs hold problem reports.

I/O and Performance

- I/O a major factor in system performance
  - demands CPU to execute device driver, kernel I/O code, etc.
  - context switches due to interrupts
  - data copying
  - metwork traffic especially stressful.
- Improving performance:
  - reduce number of context switches
  - reduce data copying
  - reduce # interrupts by using large transfers, smart controllers, polling
  - use DMA where possible
  - balance CPU, memory, bus and I/O performance for highest throughput.

Improving I/O performance is one of the main remaining systems challenges. . .

File Management



Filing systems have two main components:

1. Directory Service
   - maps from names to file identifiers.
   - handles access & existence control
2. Storage Service
   - provides mechanism to store data on disk
   - includes means to implement directory service

File Concept

What is a file?

- Basic abstraction for non-volatile storage.
- Typically comprises a single contiguous logical address space.
- Internal structure:
  1. None (e.g. sequence of words, bytes)
  2. Simple record structures
     - lines
     - fixed length
     - variable length
  3. Complex structures
     - formatted document
     - relocatable object file
- Can simulate last two with first method by inserting appropriate control characters.
- All a question of who decides:
  - operating system
  - program(mer).

Naming Files

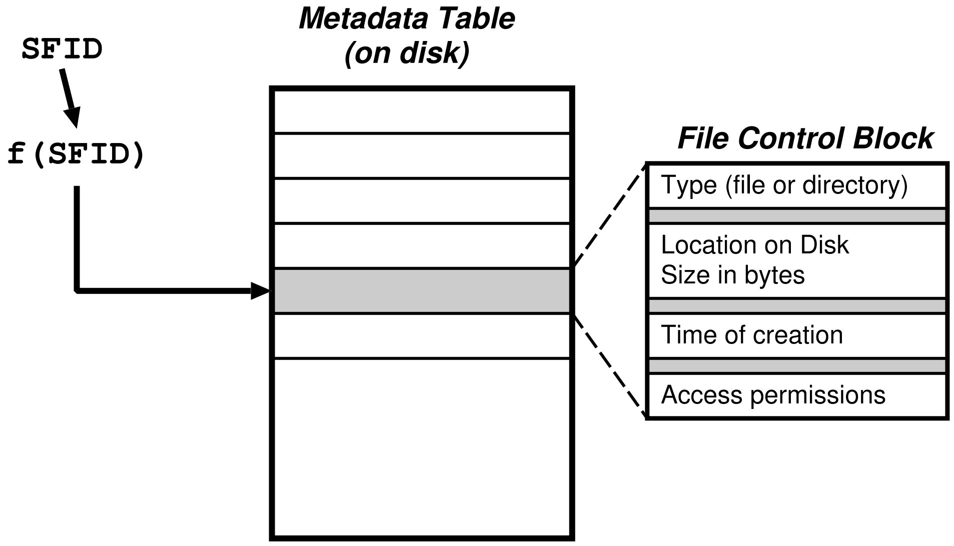
Files usually have at least two kinds of ‘name’:

1. System file identifier (SFID):
   - (typically) a unique integer value associated with a given file
   - SFIDs are the names used within the filing system itself
2. “Human” name, e.g. `hello.java`
   - What users like to use
   - Mapping from human name to SFID is held in a *directory*, e.g.

Name	**SFID**
hello.java	12353
Makefile	23812
README	9742

   - Directories also non-volatile ⇒ must be stored on disk along with files.
3. Frequently also get user file identifier (UFID).
   - used to identify *open* files (see later)

File Meta-data



In addition to their contents and their name(s), files typically have a number of other attributes, e.g.

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

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

Directory Name Space (I)

What are the requirements for our name space?

- Efficiency: locating a file quickly.
- Naming: user convenience
  - allow two (or more generally N) users to have the same name for different files
  - allow one file have several different names
- Grouping: logical grouping of files by properties (e.g. all Java programs, all games, . . .)

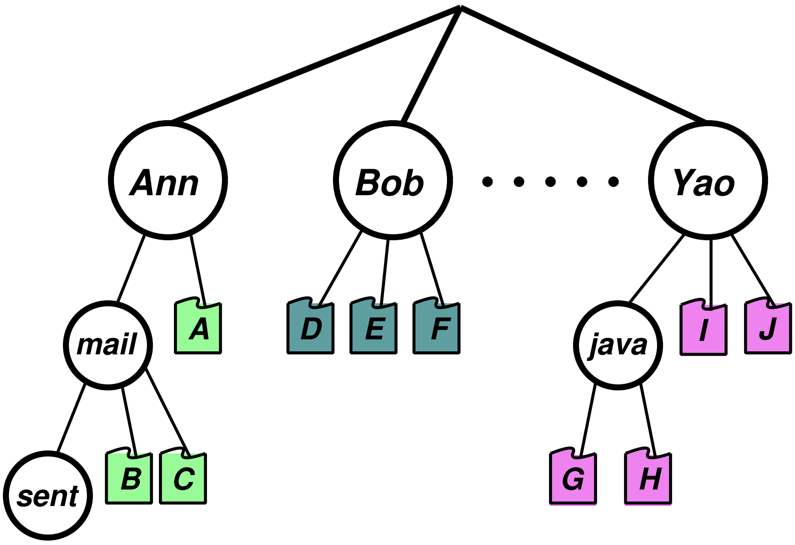
First attempts:

- Single-level: one directory shared between all users

  ⇒ naming problem
  ⇒ grouping problem

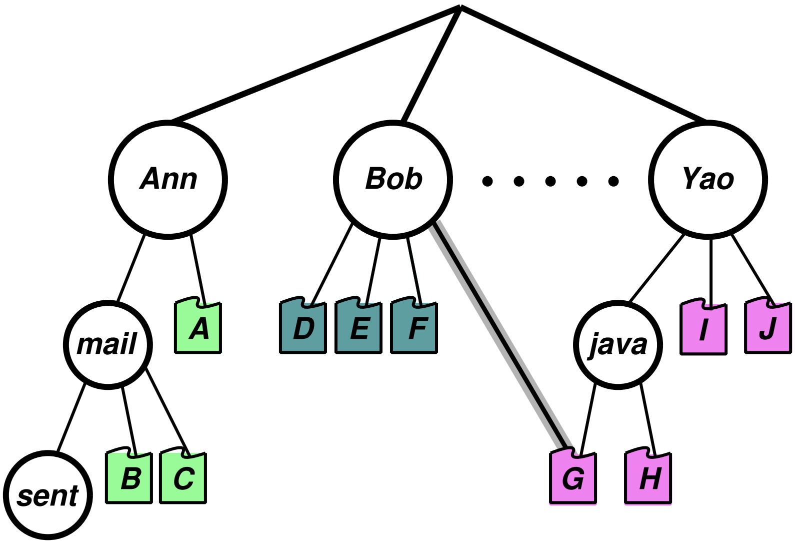
- Two-level directory: one directory per user
  - access via *pathname* (e.g. `bob:hello.java`)
  - can have same filename for different user
  - but still no grouping capability.

Directory Name Space (II)



- Get more flexibility with a general *hierarchy*.
  - directories hold files or [further] directories
  - create/delete files relative to a given directory
- Human name is full path name, but can get long: e.g. /usr/groups/X11R5/src/mit/server/os/4.2bsd/utils.c
  - offer relative naming
  - login directory
  - current working directory
- What does it mean to delete a [sub]-directory?

Directory Name Space (III)



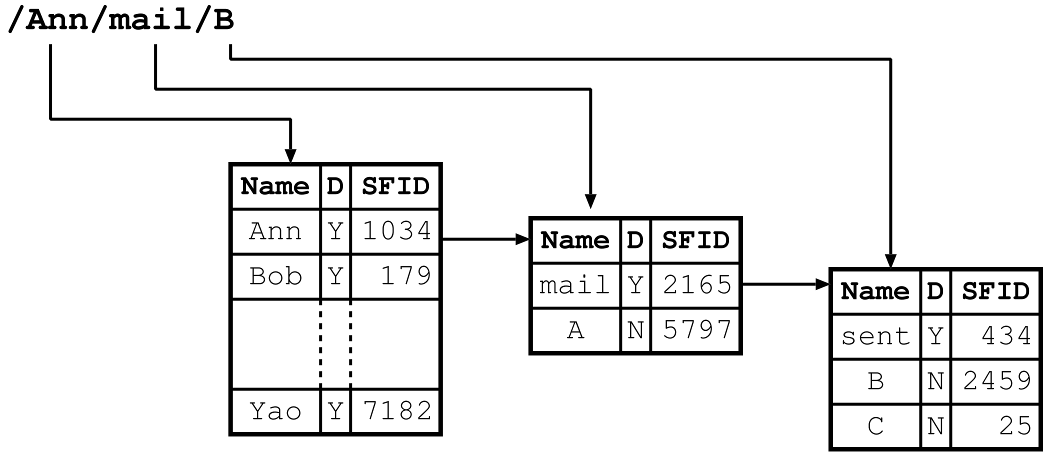
- Hierarchy good, but still only one name per file.

⇒ extend to directed acyclic graph (DAG) structure:

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

- Problem: dangling references
- Solutions:
  - back-references (but variable size records)
  - reference counts.
- Problem: cycles. . .

Directory Implementation



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

Name	D	SFID
mail	Y	2165
A	N	5797

Name	D	SFID
sent	Y	434
B	N	2459
C	N	25

- Directories are non-volatile ⇒ store as "files" on disk, each with own SFID.
- Must be different *types* of file (for traversal)
- Explicit directory operations include:
  - create directory
  - delete directory
  - list contents
  - select current working directory
  - insert an entry for a file (a "link")

File Operations (I)

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

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

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

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

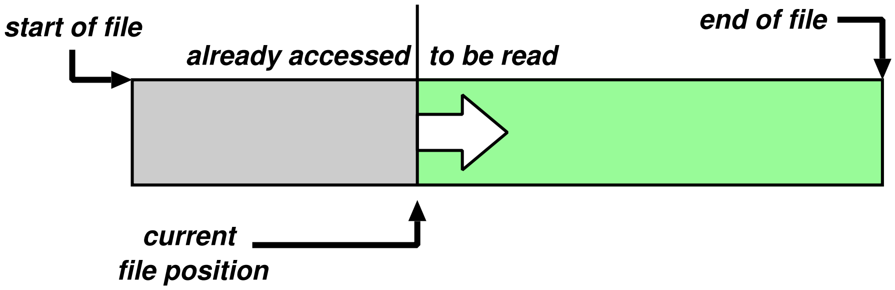
- Once have `UFID` can read, write, etc.

    – various modes (see next slide)

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

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

File Operations (II)



- Associate a *cursor* or *file position* with each open file (viz. UFID), initialised to start of file.
- Basic operations: *read next* or *write next*, e.g.
  - `read(UFID, buf, nbytes)`, or
  - `read(UFID, buf, nrecords)`
- Sequential Access: above, plus `rewind(UFID)`.
- Direct Access: *read* N or *write* N
  - allow "random" access to any part of file.
  - can implement with `seek(UFID, pos)`
- Other forms of data access possible, e.g.
  - append-only (may be faster)
  - indexed sequential access mode (ISAM)

Other Filing System Issues

- Access Control: file owner/creator should be able to control what can be done, and by whom.
  - access control normally a function of directory service ⇒ checks done at file *open* time
  - various types of access, e.g.
    - read, write, execute, (append?),
    - delete, list, rename
  - more advanced schemes possible (see later)
- Existence Control: what if a user deletes a file?
  - probably want to keep file in existence while there is a valid pathname referencing it
  - plus check entire FS periodically for garbage
  - existence control can also be a factor when a file is renamed/moved.
- Concurrency Control: need some form of *locking* to handle simultaneous access
  - may be mandatory or advisory
  - locks may be shared or exclusive
  - granularity may be file or subset

Protection

Require protection against unauthorised:

- release of information
  - reading or leaking data
  - violating privacy legislation
  - using proprietary software
  - covert channels
- modification of information
  - changing access rights
  - can do sabotage without reading information
- denial of service
  - causing a crash
  - causing high load (e.g. processes or packets)
  - changing access rights

Also wish to protect against the effects of errors:

- isolate for debugging
- isolate for damage control

Protection mechanisms impose controls on access by *subjects* (e.g. users) on *objects* (e.g. files).

Protection and Sharing

If we have a single user machine with no network connection in a locked room then protection is easy.

But we want to:

- share facilities (for economic reasons)
- share and exchange data (application requirement)

Some mechanisms we have already come across:

- user and supervisor levels
  - usually one of each
  - could have several (e.g. MULTICS rings)
- memory management hardware
  - protection keys
  - relocation hardware
  - bounds checking
  - separate address spaces
- files
  - access control list
  - groups etc

Design of Protection System

- Some other protection mechanisms:
  - lock the computer room (prevent people from tampering with the hardware)
  - restrict access to system software
  - de-skill systems operating staff
  - keep designers away from final system!
  - use passwords (in general challenge/response)
  - use encryption
  - legislate
- ref: Saltzer + Schroeder Proc. IEEE Sept 75
  - design should be public
  - default should be no access
  - check for current authority
  - give each process minimum possible authority
  - mechanisms should be simple, uniform and built in to lowest layers
  - should be psychologically acceptable
  - cost of circumvention should be high
  - minimize shared access

Authentication of User to System (1)

Passwords currently widely used:

- want a long sequence of random characters issued by system, but user would write it down
- if allow user selection, they will use dictionary words, car registration, their name, etc.
- best bet probably is to encourage the use of an algorithm to remember password
- other top tips:
  - don't reflect on terminal, or overprint
  - add delay after failed attempt
  - use encryption if line suspect
- what about security of password file?
  - only accessible to login program (CAP, TITAN)
  - hold scrambled, e.g. UNIX
    - only need to write protect file
    - need irreversible function (without password)
    - maybe 'one-way' function
    - however, off line attack possible

      ⇒ really should use *shadow passwords*

Authentication of User to System (2)

E.g. passwords in UNIX:

- simple for user to remember

```
arachnid
```

- sensible user applies an algorithm

```
!r!chn#d
```

- password is DES-encrypted 25 times using a 2-byte per-user 'salt' to produce a 11 byte string
- salt followed by these 11 bytes are then stored

```
IML.DVMcz6Sh2
```

Really require unforgeable evidence of identity that system can check:

- enhanced password: challenge-response.
- id card inserted into slot
- fingerprint, voiceprint, face recognition
- smart cards

Authentication of System to User

User wants to avoid:

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

Partial solution in old days for directly wired terminals:

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

But, today micros used as terminals ⇒

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

Anyway, still have the problem of comms lines:

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

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

Mutual suspicion

- We need to encourage lots and lots of suspicion:
  - system of user
  - users of each other
  - user of system
- Called programs should be suspicious of caller (e.g. OS calls always need to check parameters)
- Caller should be suspicious of called program
- e.g. Trojan horse:
  - a ‘useful’ looking program — a game perhaps
  - when called by user (in many systems) inherits all of the user’s privileges
  - it can then copy files, modify files, change password, send mail, etc. . .
  - e.g. Multics editor trojan horse, copied files as well as edited.
- e.g. Virus:
  - often starts off as Trojan horse
  - self-replicating (e.g. ILOVEYOU)

Access matrix

Access matrix is a matrix of subjects against objects.

Subject (or principal) might be:

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

Objects are things like:

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

Matrix is large and sparse ⇒ don't want to store it all.

Two common representations:

1. by object: store list of subjects and rights with each object ⇒ *access control list*
2. by subject: store list of objects and rights with each subject ⇒ *capabilities*

Access Control Lists

Often used in storage systems:

- system naming scheme provides for ACL to be inserted in naming path, e.g. files
- if ACLs stored on disk, check is made in software ⇒ must only use on low duty cycle
- for higher duty cycle must cache results of check
- e.g. Multics: open file = memory segment.
  On first reference to segment:
  1. interrupt (segment fault)
  2. check ACL
  3. set up segment descriptor in segment table
- most systems check ACL
  - when file opened for read or write
  - when code file is to be executed
- access control by program, e.g. Unix
  - exam prog, RWX by examiner, X by student
  - data file, A by exam program, RW by examiner
- allows arbitrary policies. . .

Capabilities

Capabilities associated with active subjects, so:

- store in address space of subject
- must make sure subject can't forge capabilities
- easily accessible to hardware
- can be used with high duty cycle
  e.g. as part of addressing hardware
  - Plessey PP250
  - CAP I, II, III
  - IBM system/38
  - Intel iAPX432
- have special machine instructions to modify (restrict) capabilities
- support passing of capabilities on procedure (program) call

Can also use *software* capabilities:

- checked by encryption
- nice for distributed systems

Password Capabilities

- Capabilities nice for distributed systems but:
  - messy for application, and
  - revocation is tricky.
- Could use timeouts (e.g. Amoeba).
- Alternatively: combine passwords and capabilities.
- Store ACL with object, but key it on capability (not implicit concept of “principal” from OS).
- Advantages:
  - revocation possible
  - multiple “roles” available.
- Disadvantages:
  - still messy (use ‘implicit’ cache?).

Covert channels

Information leakage by side-effects: lots of fun!

At the hardware level:

- wire tapping
- monitor signals in machine
- modification to hardware
- electromagnetic radiation of devices

By software:

- leak a bit stream as:

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

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

In general, guarding against covert channels is prohibitively expensive.

(only usually a consideration for military types)

Glossary and Acronyms: A–H

AGP	Advanced Graphics Port
ALU	Arithmetic/Logic Unit
API	Application Programming Interface
ARM	a 32-bit RISC microprocessor
ASCII	American Standard Code for Information Interchange
Alpha	a 64-bit RISC microprocessor
BSD	Berkeley Software Distribution (Unix variant)
BU	Branch Unit
CAM	Content Addressable Memory
COW	Copy-on-Write
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
DMA	Direct Memory Access
DOS	1. a primitive OS (Microsoft) 2. Denial of Service
DRAM	Dynamic RAM
FCFS	First-Come-First-Served (see also FIFO)
FIFO	First-In-First-Out (see also FCFS)
FS	File System
Fork	create a new copy of a process
Frame	chunk of physical memory (also *page frame*)
HAL	Hardware Abstraction Layer

Glossary and Acronyms: I–N

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

Glossary and Acronyms: 0–Sp

OS	Operating System
OS/2	a PC operating system (IBM & Microsoft)
PC	1. Program Counter 2. Personal Computer
PCB	1. Process Control Block 2. Printed Circuit Board
PCI	Peripheral Component Interface
PIC	Programmable Interrupt Controller
PTBR	Page Table Base Register
PTE	Page Table Entry
Page	chunk of virtual memory
Poll	[repeatedly] determine the status of
Posix	Portable OS Interface for Unix
RAM	Random Access Memory
ROM	Read-Only Memory
SCSI	Small Computer System Interface
SFID	System File ID
Shell	program allowing user-computer interaction
Signal	event delivered from OS to a process
SJF	Shortest Job First
SMP	Symmetric Multi-Processor
Sparc	a 32 bit RISC processor (Sun)

Glossary and Acronyms: SR–X

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