CMPUT 379

Operating Systems Concepts

Pawel Gburzynski
321 Athabasca Hall
Office Hours: MWF 15:00-16:00

Credits (sort of)

These slides do not follow any textbook in particular; however, some examples, figures, graphs may have been borrowed (sometimes subliminally) from several books, including:

A. Siberschatz, G. Galvin, P. Gagne. Operating System Concepts, Wiley & Sons.

A. Tanenbaum. Modern Operating Systems, Prentice Hall.

… and others. I have also taken advantage of miscellaneous stuff found on the Web.

This mark in the right bottom corner indicates that the slide contains dynamic effects, and its static (paper) variant may (but doesn't have to) be a bit confusing.

A simple definition

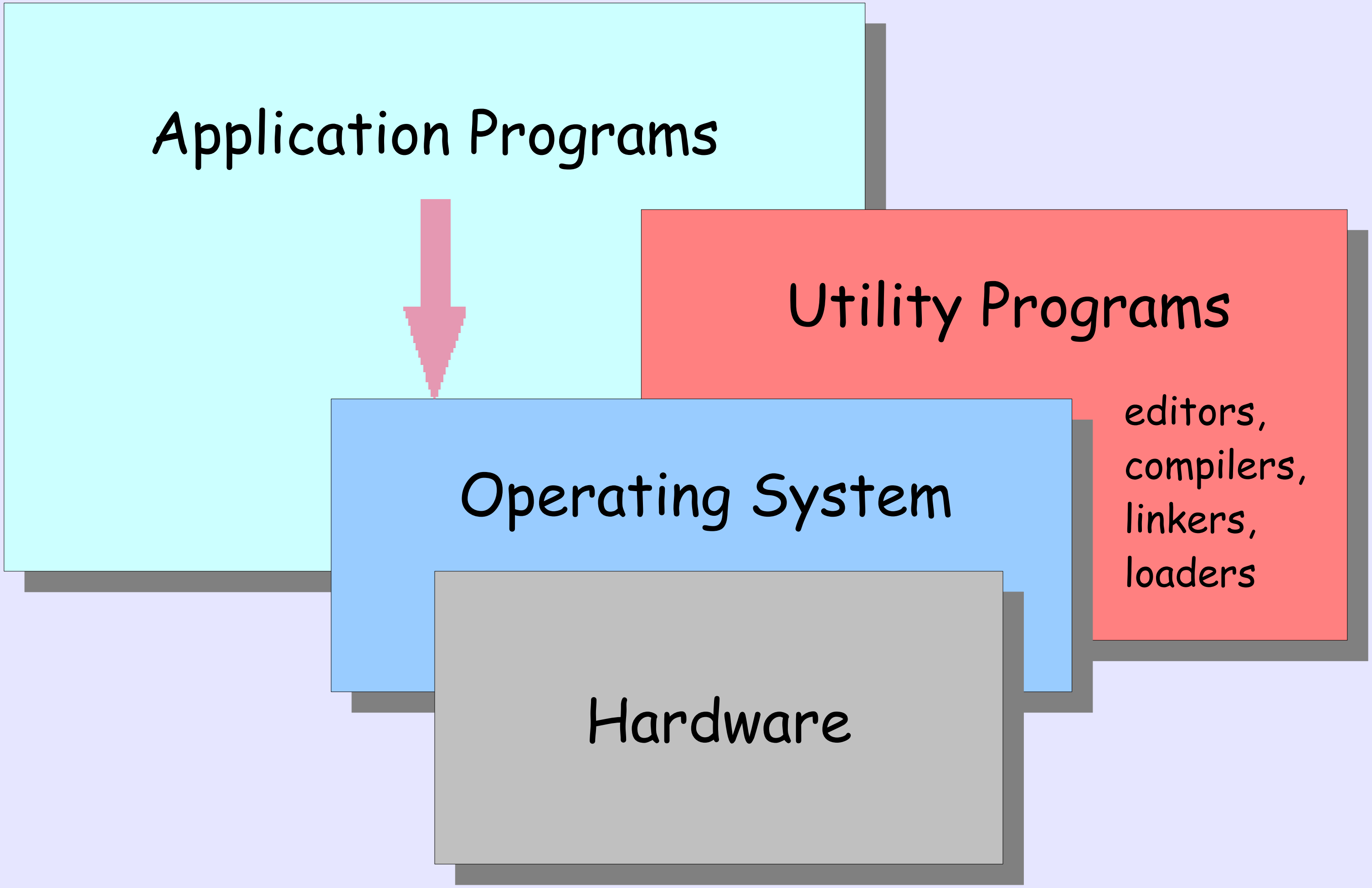
I am assuming that you have written some programs and run them on some computers.

An operating system is a bunch of software layers separating your programs from the computer hardware.

We shall start from a short trip through the layers separating a program from the OS.

Our question is “what does it mean for a program to actually execute on a modern computer”?

An obvious picture



Application Programs

Utility Programs

editors,
compilers,
linkers,
loaders

Operating System

Hardware

Here is a sample program

program.c

```
int read_something (void);
int do_something (int);
void write_something (const char*);

int some_global_variable;
static int some_local_variable;
main () {
        int some_stack_variable;
        some_stack_variable = read_something ();
        some_global_variable = do_something (some_stack_variable);
        write_something ("I am done");
}
```

Here's the rest of it:

extras.c

```
#include <stdio.h>
extern int some_global_variable;
int read_something (void) {
        int res;
        scanf ("%d", &res);
        return res;
}
int do_something (int var) {
        return var + var;
}
void write_something (const char* str) {
        printf ("%s: %d\n", str, some_global_variable);
}
```

A reminder: basic storage classes in C

```
int some_global_variable;
    globally visible by all modules (common, bss)

static int some_local_variable;
   global/local: visible by all functions within the current module,
   but not outside the module (data)

main () {
      int some_stack_variable;
              allocated on the stack, visible only within the block
              (called auto for automatic)
   ...
}
```

How we compile and run the program

```
gcc program.c extras.c
./a.out
```

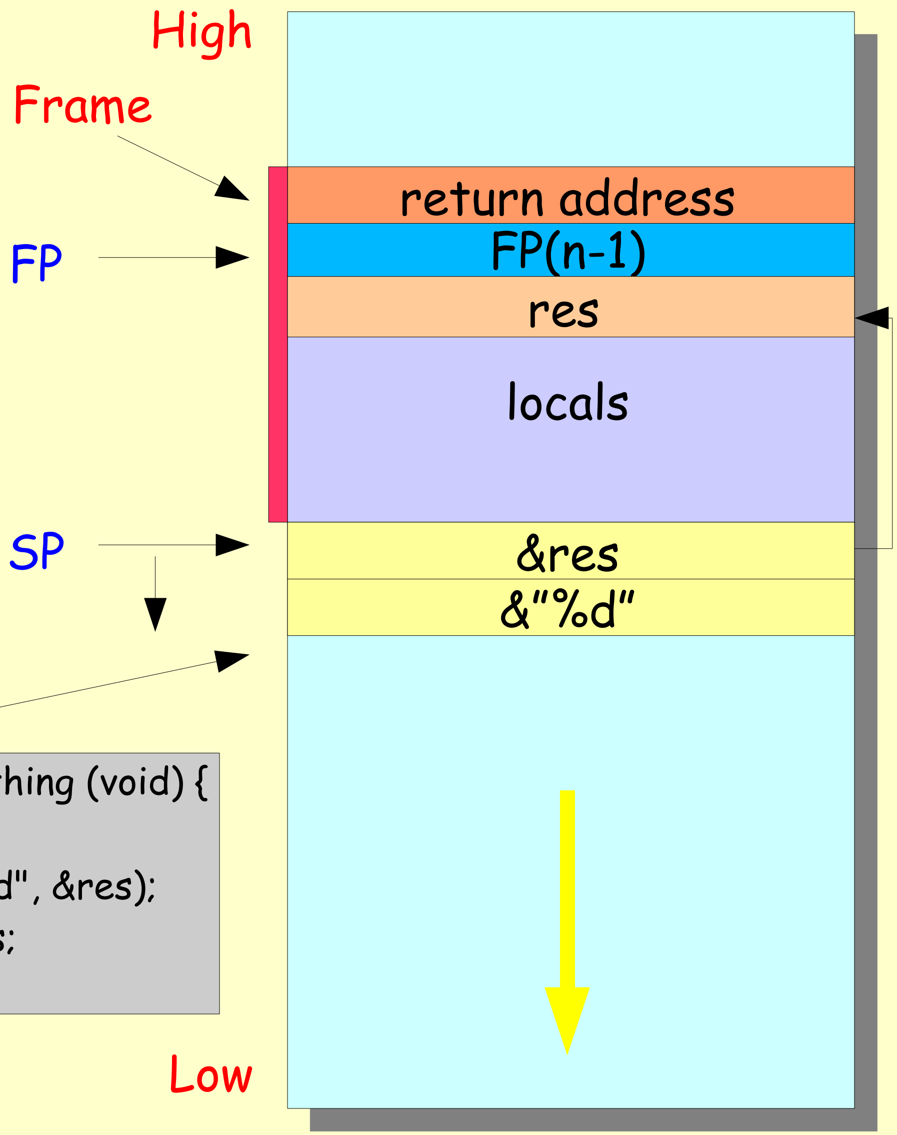
or we can compile the modules independently:

```
gcc -c program.c               => program.o
gcc -c extras.c                => extras.o
gcc program.o functions.o -o executable
./executable
```

gcc -S extras.c produces extras.s:

```
        .file   "extras.c"
        .version "01.01"
gcc2_compiled.:
        .section .rodata
.LC0:
        .string "%d"
.text
        .align 4
.globl read_something
        .type    read_something,@function
read_something:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        leal    -4(%ebp), %eax
        pushl   %eax
        pushl   $.LC0
        call    scanf
        movl    -4(%ebp), %eax
        leave
        ret
....
```

```
int read_something (void) {
        int res;
        scanf ("%d", &res);
        return res;
}
```



program.o looks like this:

```
000000 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
000010 01 00 03 00 01 00 00 00 00 00 00 00 00 00 00 00
000020 04 01 00 00 00 00 00 00 34 00 00 00 00 00 28 00
000030 0b 00 08 00 55 89 e5 83 ec 08 e8 fc ff ff ff 89
000040 c0 89 45 fc 83 ec 0c ff 75 fc e8 fc ff ff ff 83
000050 c4 10 89 c0 a3 00 00 00 00 83 ec 0c 68 00 00 00
000060 00 e8 fc ff ff ff 83 c4 10 c9 c3 90 00 00 00 00


......


0003b0 6c 5f 76 61 72 69 61 62 6c 65 00 6d 61 69 6e 00
0003c0 72 65 61 64 5f 73 6f 6d 65 74 68 69 6e 67 00 64
0003d0 6f 5f 73 6f 6d 65 74 68 69 6e 67 00 73 6f 6d 65
0003e0 5f 67 6c 6f 62 61 6c 5f 76 61 72 69 61 62 6c 65
0003f0 00 77 72 69 74 65 5f 73 6f 6d 65 74 68 69 6e 67
000400 00 00 00 00 11 00 00 00 02 0a 00 00 1f 00 00 00
000410 02 0b 00 00 27 00 00 00 01 0c 00 00 2f 00 00 00
000420 01 05 00 00 34 00 00 00 02 0d 00 00
00042c
```

ELF format

SIGNATURE

```
000 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
    --  E  L  F 32 LE FW ---------------------------
```

ELF HEADER

```
010 01 00 03 00 01 00 00 00 00 00 00 00 00 00 00 00
     0001  0003    00000001    00000000    00000000
           target architecture                 program header (executable module only)
     relocatable               starting address for execution (executable module only)
     module

020 04 01 00 00 00 00 00 00 34 00 00 00 00 00 28 00
       00000104    00000000  0034  0000  0000  0028
      SHT offset   CPU flags  hdr    no PHT   SHT entry size
                             length

030 0b 00 08 00
    000b  0008
          index (into SHT) of the string section containing section names
    number of entries in SHT
```

SHT = Section Header Table

PHT = Program Header Table

ELF contents: SHT (0-1)

Section 0 (NULL)

```
104             00 00 00 00 00 00 00 00 00 00 00 00
110 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
120 00 00 00 00 00 00 00 00 00 00 00 00
```

Entry size = 28; 104 + 28 = 12c

28 bytes = a words

Section 1 (.text)

```
12c 1f 00 00 00 = 0000001f   name pointer (index into a string section)
    01 00 00 00 = 00000001   section type (PROGBITS = belongs to the program)
    06 00 00 00 = 00000006   flags: in program's memory + executable
    00 00 00 00 = 00000000   address in memory (unknown because module is relocatable)
    34 00 00 00 = 00000034   offset to section in ELF file
    38 00 00 00 = 00000038   size 34 + 38 = 6c
    00 00 00 00 = 00000000   extra link (none - typically points to a related section)
    00 00 00 00 = 00000000   extra info (none)
    04 00 00 00 = 00000004   address alignment (word boundary)
    00 00 00 00 = 00000000   entry size (if the section consists of fixed-size entries)
```

ELF contents: SHT (2-3)

```
Section 2 (.rel.text)
154 1b 00 00 00
    09 00 00 00    type: list of relocation points
    00 00 00 00
    00 00 00 00
    04 04 00 00    offset = 404
    28 00 00 00    size = 28
    09 00 00 00    pointer to the symbol table (index into SHT)
    01 00 00 00    pointer to the section to which relocations apply
    04 00 00 00
    08 00 00 00    this section consists of entries 8 bytes each

Section 3 (.data)
17c 25 00 00 00
    01 00 00 00    PROGBITS again (a piece of program)
    03 00 00 00    in memory + writeable
    00 00 00 00
    74 00 00 00    offset (irrelevant if size 0)
    00 00 00 00    size is zero (our one static variable has been optimized out)
    00 00 00 00
    00 00 00 00
    04 00 00 00    word alignment
    00 00 00 00
```

ELF contents: SHT (4-7)

Section 4 (.bss)

```
1a4 2b 00 00 00
    08 00 00 00    NOBITS (no space to fill)
    03 00 00 00    writeable
    00 00 00 00
    74 00 00 00    offset (ignored)
    04 00 00 00    one word
    00 00 00 00
    00 00 00 00
    04 00 00 00
    00 00 00 00
```

Section 6 (.note.GNU-stack)

```
1f4 38 00 00 00
    01 00 00 00    PROGBITS
    00 00 00 00    no memory, read only
    00 00 00 00
    7e 00 00 00    irrelevant
    00 00 00 00    no preallocated memory
    00 00 00 00
    00 00 00 00
    01 00 00 00
    00 00 00 00
```

Section 5 (.rodata)

```
1bc 30 00 00 00
    01 00 00 00    PROGBITS
    02 00 00 00    memory (read only)
    00 00 00 00
    74 00 00 00
    0a 00 00 00
    00 00 00 00
    00 00 00 00
    01 00 00 00    looks like strings
    00 00 00 00
```

Section 7 (.comment)

```
21c 48 00 00 00
    01 00 00 00    PROGBITS
    00 00 00 00    no memory
    00 00 00 00
    7e 00 00 00
    33 00 00 00    but present in the file
    00 00 00 00
    00 00 00 00
    01 00 00 00
    00 00 00 00
```

ELF contents: SHT (8-a)

Section 8 (.shstrtab)

```
244 11 00 00 00
    03 00 00 00    STRTAB (strings, not for program – contains section names, see header)
    00 00 00 00
    00 00 00 00
    b1 00 00 00
    51 00 00 00
    00 00 00 00
    00 00 00 00
    01 00 00 00
    00 00 00 00
```

Section 9 (.symtab)

```
26c 01 00 00 00
    02 00 00 00    SYMTAB
    00 00 00 00
    00 00 00 00
    bc 02 00 00    offset = 02bc
    e0 00 00 00    size = e0
    0a 00 00 00    linked to section a =>
    09 00 00 00    first non-local symbol
    04 00 00 00
    10 00 00 00    entry size = 10
```

Section a (.strtab)

```
294 09 00 00 00
    03 00 00 00    STRTAB
    00 00 00 00
    00 00 00 00
    9c 03 00 00    offset = 039c
    65 00 00 00    length = 65
    00 00 00 00
    00 00 00 00
    01 00 00 00
    00 00 00 00
```

ELF contents: section body 8, 5, 1

Section 8 (STRTAB .shstrtab)

```
0b1    00 2e 73 79 6d 74 61 62 00 2e 73 74 72 74 61
       --  .  s  y  m  t  a  b --  .  s  t  r  t  a
0c0 62 00 2e 73 68 73 74 72 74 61 62 00 2e 72 65 6c
     b --  .  s  h  s  t  r  t  a  b --  .  r  e  l
0d0 2e 74 65 78 74 00 2e 64 61 74 61 00 2e 62 73 73
     .  t  e  x  t --  .  d  a  t  a --  .  b  s  s
0e0 00 2e 72 6f 64 61 74 61 00 2e 6e 6f 74 65 2e 47
    --  .  r  o  d  a  t  a --  .  n  o  t  e  .  G
0f0 4e 55 2d 73 74 61 63 6b 00 2e 63 6f 6d 6d 65 6e
     N  U  -  s  t  a  c  k --  .  c  o  m  m  e  n
100 74 00
     t --
```

Section 5 (PROGBITS .rodata)

```
074             49 20 61 6d 20 64 6f 6e 65 00
                 I     a  m     d  o  n  e --
```

Section 1 (PROGBITS .text)

```
034             55 89 e5 83 ec 08 e8 fc ff ff ff 89
040 c0 89 45 fc 83 ec 0c ff 75 fc e8 fc ff ff ff 83
050 c4 10 89 c0 a3 00 00 00 00 83 ec 0c 68 00 00 00
060 00 e8 fc ff ff ff 83 c4 10 c9 c3 90
```

ELF contents: sections a and 7

Section a (STRTAB .strtab)

```
39c                                     00 70 72 6f
                                            p  r  o
3a0 67 72 61 6d 2e 63 00 73 6f 6d 65 5f 6c 6f 63 61
     g  r  a  m  .  c -- s  o  m  e  _  l  o  c  a
3b0 6c 5f 76 61 72 69 61 62 6c 65 00 6d 61 69 6e 00
     l  _  v  a  r  i  a  b  l  e -- m  a  i  n --
3c0 72 65 61 64 5f 73 6f 6d 65 74 68 69 6e 67 00 64
     r  e  a  d  _  s  o  m  e  t  h  i  n  g __ d
3d0 6f 5f 73 6f 6d 65 74 68 69 6e 67 00 73 6f 6d 65
     o  _  s  o  m  e  t  h  i  n  g __ s  o  m  e
3e0 5f 67 6c 6f 62 61 6c 5f 76 61 72 69 61 62 6c 65
     _  g  l  o  b  a  l  _  v  a  r  i  a  b  l  e
3f0 00 77 72 69 74 65 5f 73 6f 6d 65 74 68 69 6e 67
     _  w  r  i  t  e  _  s  o  m  e  t  h  i  n  g
    00
```

Section 7 (PROGBITS .comment)

```
07e                                           00 47
                                                  G
080 43 43 3a 20 28 47 4e 55 29 20 33 2e 33 2e 33 20
     C  C  :     (  G  N  U  )     3  .  3  .  3
090 32 30 30 34 30 34 31 32 20 28 52 65 64 20 48 61
     2  0  0  4  0  4  1  2     (  R  e  d     H  a
0a0 74 20 4c 69 6e 75 78 20 33 2e 33 2e 33 2d 37 29
     t     L  i  n  u  x     3  .  3  .  3  -  7  )
0b0 00
```

ELF contents: sections 9 and 2

Section 9 (.symtab)

```
2bc 00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
    ==============  ==============  ==============  ==  ==  ======
    01  00  00  00  00  00  00  00  00  00  00  00  04  00  f1  ff
    ==============  ==============  ==============  ==  ==  ======

         name           value            size        info   SHTIDX
    00  00  00  00  00  00  00  00  00  00  00  00  03  00  01  00
    ==============  ==============  ==============  ==      ======
    00  00  00  00  00  00  00  00  00  00  00  00  03  00  03  00
    00  00  00  00  00  00  00  00  00  00  00  00  03  00  04  00
    00  00  00  00  00  00  00  00  00  00  00  00  03  00  05  00
    0b  00  00  00  00  00  00  00  04  00  00  00  01  00  04  00
    00  00  00  00  00  00  00  00  00  00  00  00  03  00  06  00
    00  00  00  00  00  00  00  00  00  00  00  00  03  00  07  00

    1f  00  00  00  00  00  00  00  3d  00  00  00  12  00  01  00
    ==============  ==============  ==============  ==      ======
         main                       code size      global    .text
                                                  function
    24  00  00  00  00  00  00  00  00  00  00  00  10  00  00  00
    ==============  ==============  ==============  ==      ======
       read_...                                    global
    33  00  00  00  00  00  00  00  00  00  00  00  10  00  00  00
    40  00  00  00  04  00  00  00  04  00  00  00  11  00  f2  ff
    55  00  00  00  00  00  00  00  00  00  00  00  10  00  00  00
```

Each entry consists of:

- 3 words
- 2 bytes
- 1 half-word

These correspond to sections – those that describe the contents of the program's memory.

Section 2 (.rel.text)

```
404  07  00  00  00  02  0a  00  00
     ==============  ==============
         offset      relocation type
     17  00  00  00  02  0b  00  00
     ==============  ==============
     21  00  00  00  01  0c  00  00
     29  00  00  00  01  05  00  00
     2e  00  00  00  02  0d  00  00
```

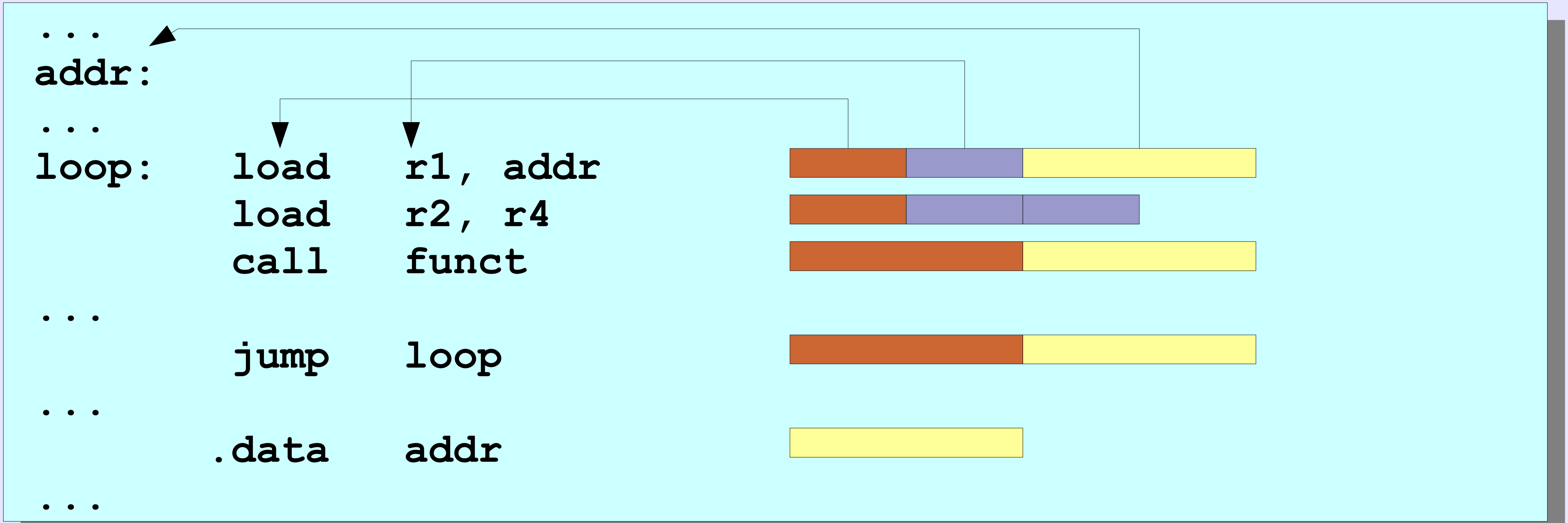
Section .text (disassembled)

```
int read_something (void);
int do_something (int);
void write_something (const char*);

int some_global_variable;
static int some_local_variable;

main () {
       int some_stack_variable;
       some_stack_variable = read_something ();
       some_global_variable = do_something (some_stack_variable);
       write_something ("I am done");
}
```

```
  0: 55                pus
  1: 89 e5             mov
  3: 83 ec 08          sub
  6: e8 fc ff ff ff    cal
  b: 89 c0             mov
  d: 89 45 fc          mov
 10: 83 ec 0c          sub
 13: ff 75 fc          pushl  0xfffffffc(%ebp)
 16: e8 fc ff ff ff    call   17 <main+0x17>
 1b: 83 c4 10          add    $0x10,%esp
 1e: 89 c0             mov    %eax,%eax
 20: a3 00 00 00 00    mov    %eax,0x0
 25: 83 ec 0c          sub    $0xc,%esp
 28: 68 00 00 00 00    push   $0x0
 2d: e8 fc ff ff ff    call   2e <main+0x2e>
 32: 83 c4 10          add    $0x10,%esp
 35: c9                leave
 36: c3                ret
```

Copyright © Pawel Gburzynski

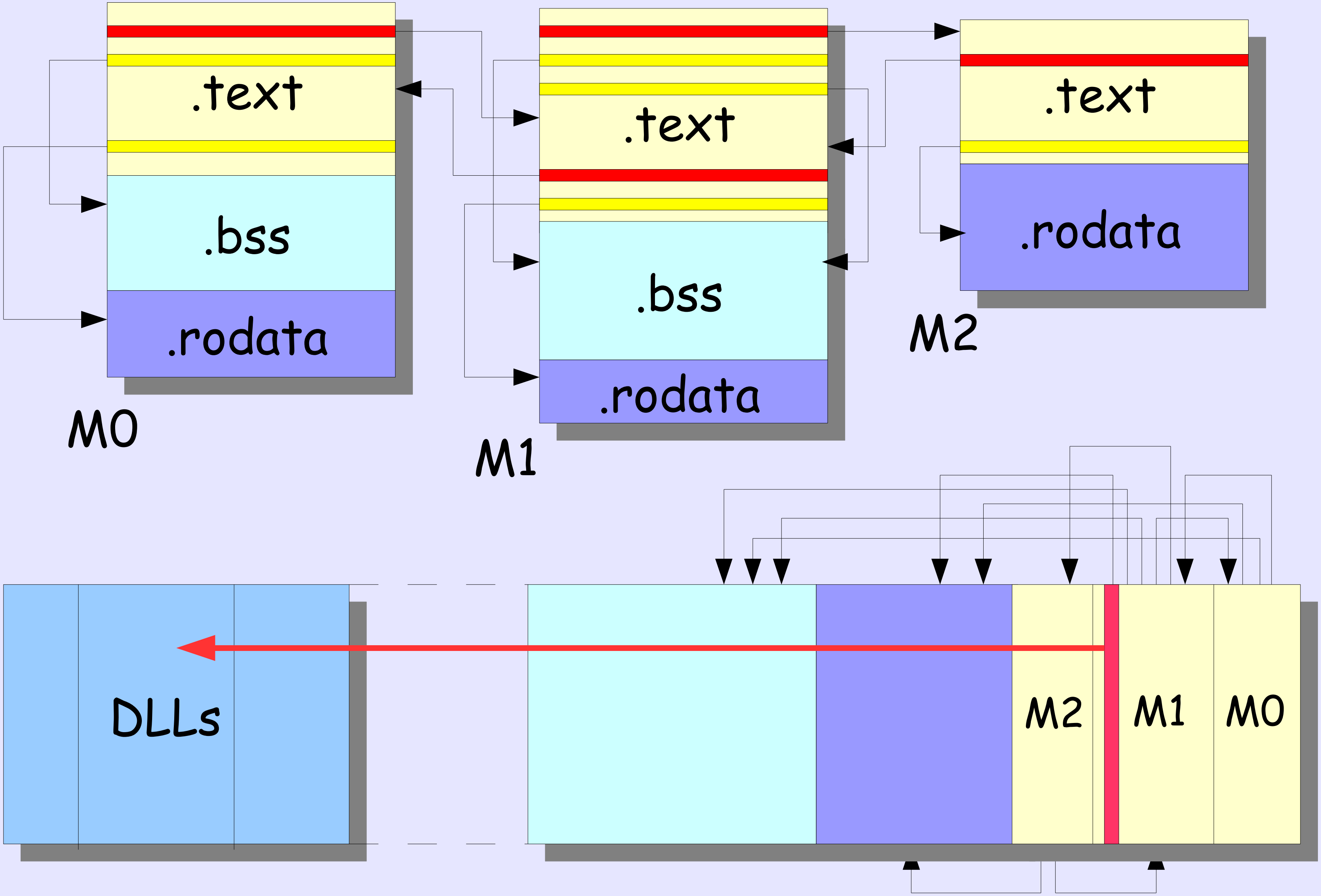
Relocatability issues



```
...
addr:
...
loop:   load   r1, addr
        load   r2, r4
        call   funct
...
        jump   loop
...
       .data   addr
...
```

An address doesn't have to be located outside the current module to be essentially unknown. The exact location of the module in memory is only known when the complete program is loaded for execution.

The role of the linker



computer memory

Classes of executable code

Absolute: it can only be sensibly executed when loaded in a specific memory location

Relocatable: some relocation information is left in the executable, such that when the program is loaded, it can still be relocated to the actual location

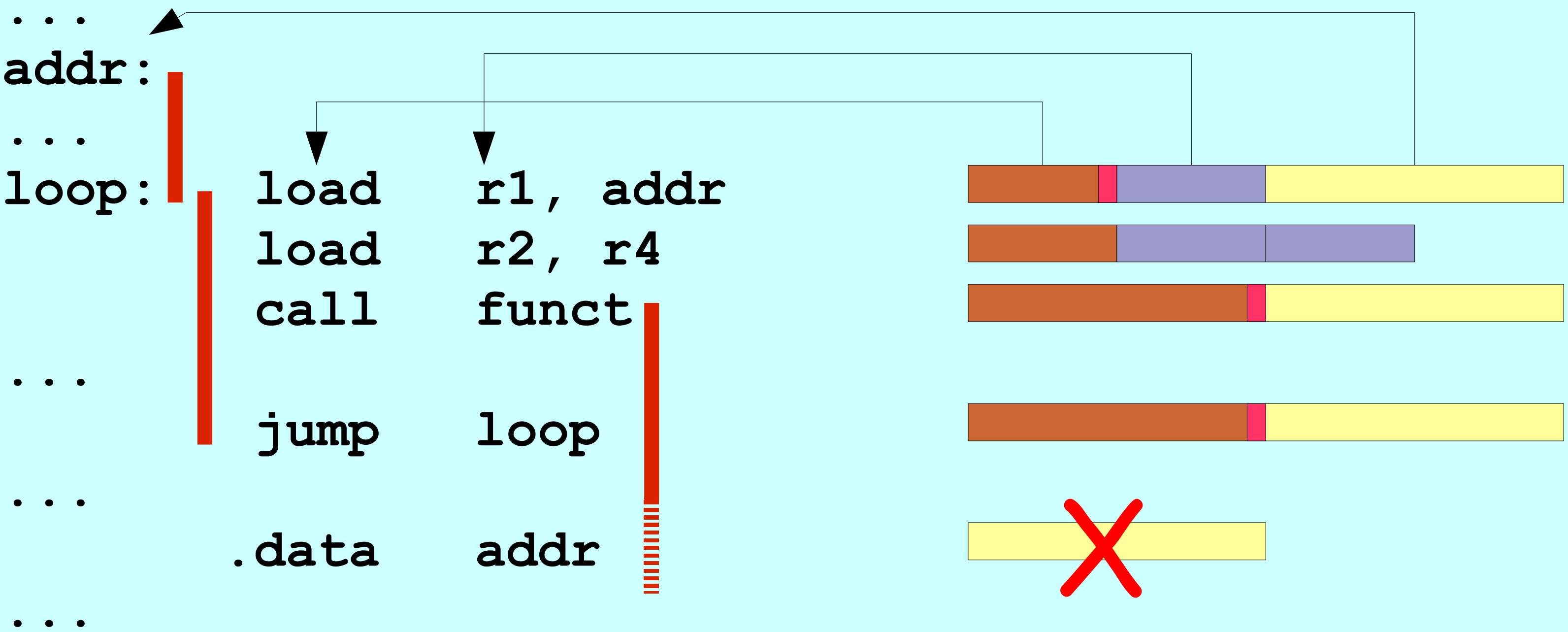
Position-independent: the program executes correctly, no matter where it is loaded

Movable: the program can be loaded anywhere, it can also be moved to a new location in the middle of its execution without impairing its integrity

PIC: Position Independent Code

Most contemporary CPU architectures implement PC-relative addressing modes, which typically look like this:

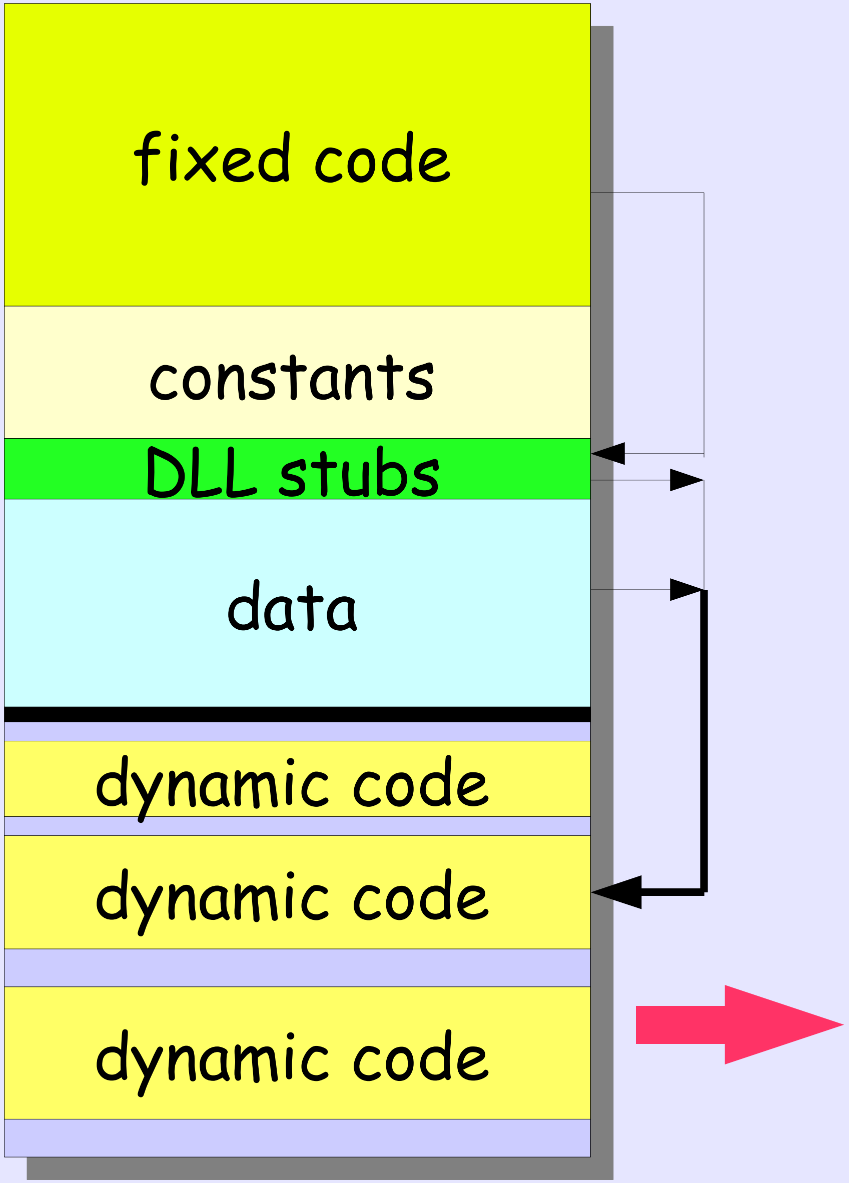
operand = Mem (PC + offset)



```
 ...
addr:
 ...
loop:   load    r1, addr
        load    r2, r4
        call    funct
 ...
        jump    loop
 ...
       .data    addr
 ...
```

PIC has many uses in various scenarios where it is convenient to be able to load the code in any location without having to worry about relocation (DLLs, for one thing).

Illustration



Note that the dynamic code can operate on non-PIC data supplied by the static caller, e.g., via function parameters.

It may be convenient to be able to put those pieces wherever there is room available

The executable

SIGNATURE (same as before)

```
000 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
    --  E  L  F 32 LE FW ---------------------------
```

ELF HEADER

```
010 02 00 03 00 01 00 00 00 90 83 04 08 34 00 00 00
     0002  0003    00000001    08048390    00000034
       ▲    target architecture   ▲        program header pointer
       |                          |
     executable                starting address for execution
     module

020 c8 2a 00 00 00 00 00 00 34 00 20 00 06 00 28 00
       00002ac8    00000000  0034  0020  0006  0028
      SHT offset   CPU flags  hdr     PHT    SHT entry size
                             length  params

030 1f 00 1c 00
    001f  001c
     ▲    index (into SHT) of the string section containing section names
     |
    number of entries in SHT
```

SHT = Section Header Table
PHT = Program Header Table

objdump

Once you know what to expect in an ELF file, there is a way to display its contents in a more friendly format – see man objdump.

objdump -h executable (list section headers)

```
executable:     file format elf32-i386
Sections:        (a bit confusing, numbering ignores the NULL section and is in decimal)
Idx Name          Size      VMA       LMA       File off  Algn
  0 .interp       00000013  080480f4  080480f4  000000f4  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
......
 11 .text         000001d0  08048390  08048390  00000390  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
......
 13 .rodata       0000001d  08048580  08048580  00000580  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 14 .data         00000010  080495a0  080495a0  000005a0  2**2
                  CONTENTS, ALLOC, LOAD, DATA
......
 21 .bss          00000020  080496b4  080496b4  000006b4  2**2
                  ALLOC
......
```

/lib/ld-linux.so.2

The essence

The executable includes many sections that at first sight appear exotic. The meaning of some of them may become clearer later in the course.

The obvious ones include:

.text	the program code
.rodata	constants, mostly character strings
.data	dynamic global initialized data
.bss	dynamic global uninitialized data

Some "exotic" sections relate to the idea of dynamic linking. More generally, a program may need specific DLLs and/or require assistance of "interpreters" to create its memory image, as its fragments become dynamically needed.

The .text section

```
Section b (.text)       = 2ac8 + c * 28
2ca8    80 00 00 00 = 00000080    name pointer
        01 00 00 00 = 00000001    section type (PROGBITS)
        06 00 00 00 = 00000006    flags: alloc + exec
        90 83 04 08 = 08048390    address in memory
        90 03 00 00 = 00000390    offset in ELF file
        d0 01 00 00 = 000001d0    size
        00 00 00 00 = 00000000    extra link (none)
        00 00 00 00 = 00000000    extra info (none)
        01 00 00 00 = 00000001    address alignment (byte)
        00 00 00 00 = 00000000    entry size
```

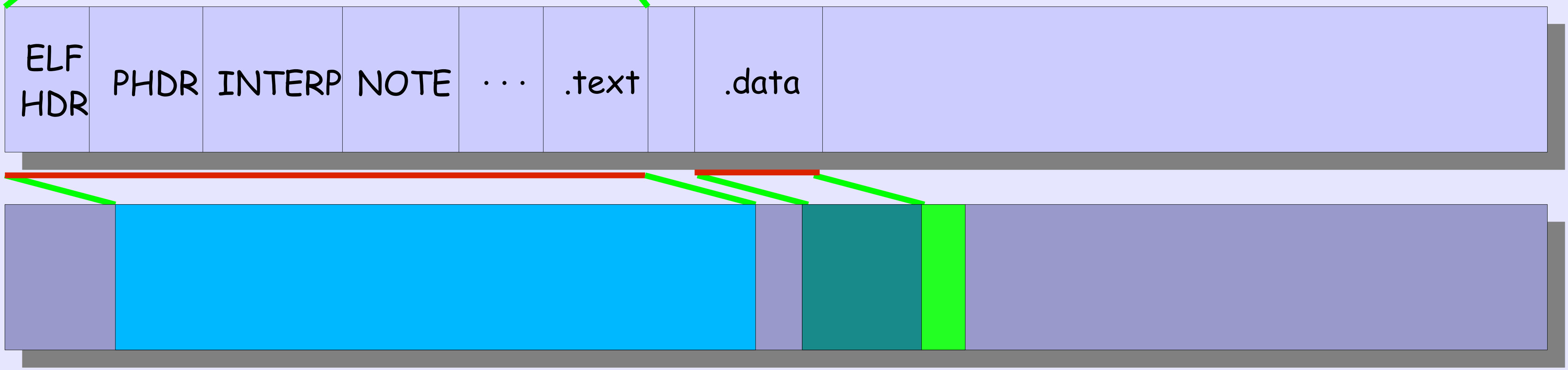
Sections representing memory chunks, like .text, .data, .rodata, are simply pieces to be written into the respective locations in memory to build the program's executable image.

Some allocatable sections, like .bss, have no image in the file: size and location are the only attributes that matter.

Things are a bit more complicated:

Program Header: *(this is what in fact describes how the program is loaded)*

```
    PHDR off    0x00000034 vaddr 0x08048034 paddr 0x08048034 align 2**2
         filesz 0x000000c0 memsz 0x000000c0 flags r-x
  INTERP off    0x000000f4 vaddr 0x080480f4 paddr 0x080480f4 align 2**0
         filesz 0x00000013 memsz 0x00000013 flags r--
    LOAD off    0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
         filesz 0x0000059d memsz 0x0000059d flags r-x
    LOAD off    0x000005a0 vaddr 0x080495a0 paddr 0x080495a0 align 2**12
         filesz 0x00000114 memsz 0x00000134 flags rw-
 DYNAMIC off    0x000005ec vaddr 0x080495ec paddr 0x080495ec align 2**2
         filesz 0x000000c8 memsz 0x000000c8 flags rw-
    NOTE off    0x00000108 vaddr 0x08048108 paddr 0x08048108 align 2**2
         filesz 0x00000020 memsz 0x00000020 flags r--
```



How do we interface with the OS?

```
 80484d0 <read_something>:
 80484d0:       55                      push   %ebp
 80484d1:       89 e5                   mov    %esp,%ebp
 80484d3:       83 ec 08                sub    $0x8,%esp
 80484d6:       83 ec 08                sub    $0x8,%esp
 80484d9:       8d 45 fc                lea    0xfffffffc(%ebp),%eax
 80484dc:       50                      push   %eax
 80484dd:       68 92 85 04 08          push   $0x8048592
 80484e2:       e8 59 fe ff ff          call   8048340 <_init+0x38>
 80484e7:       83 c4 10                add    $0x10,%esp
 80484ea:       8b 45 fc                mov    0xfffffffc(%ebp),%eax
 80484ed:       c9                      leave
 80484ee:       c3                      ret
```

```
08048320 <.plt>:        (stands for Procedure Linkage Table)
 8048320:    ff 35 c8 95 04 08      pushl  0x80495c8
 8048326:    ff 25 cc 95 04 08      jmp    *0x80495cc
 804832c:    00 00                  add    %al,(%eax)
 804832e:    00 00                  add    %al,(%eax)
 8048330:    ff 25 d0 95 04 08      jmp    *0x80495d0
 8048336:    68 00 00 00 00         push   $0x0
 804833b:    e9 e0 ff ff ff         jmp    8048320 <_init+0x18>
 8048340:    ff 25 d4 95 04 08      jmp    *0x80495d4
 8048346:    68 08 00 00 00         push   $0x8
 804834b:    e9 d0 ff ff ff         jmp    8048320 <_init+0x18>
.....
```

Lazy linkage

```
08048320 <.plt>:        (stands for Procedure Linkage Table)
 8048320:    ff 35 c8 95 04 08       pushl   0x80495c8
 8048326:    ff 25 cc 95 04 08       jmp     *0x80495cc
 804832c:    00 00                   add     %al,(%eax)
 804832e:    00 00                   add     %al,(%eax)
 8048330:    ff 25 d0 95 04 08       jmp     *0x80495d0
 8048336:    68 00 00 00 00          push    $0x0
 804833b:    e9 e0 ff ff ff          jmp     8048320 <_init+0x18>
 8048340:    ff 25 d4 95 04 08       jmp     *0x80495d4
 8048346:    68 08 00 00 00          push    $0x8
 804834b:    e9 d0 ff ff ff          jmp     8048320 <_init+0x18>
.....
```

```
Contents of section .got:    (global offset table)
 80495c4 080495ec 00000000 00000000 08048336
 80495d4 08048346 08048356 08048366 08048376
 80495e4 08048386 00000000
```

This address will be replaced on the first call by the actual address of the called function. Just by looking at the executable, we have no clue where it will be located. These days, it is a standard practice to put all system functions in DLLs.

Steps involved

When the program is loaded, the loader identifies:

- the “interpreter” responsible for dynamic linkage
- the “DLLs” needed by the program

this information is available in the ELF headers

All calls to DLL functions are turned into lazy linkage pointers involving a call through the .ptl section, and an offset in the .got section.

The initial setting of the offset triggers a jump to the interpreter with parameters that allow it to identify the DLL and the called function.

Before calling the function, the interpreter replaces the offset in .got with the direct pointer to the function, so that the next call will be straightforward (or as straightforward as it can get under the circumstances ☺). .got it?

Now for the easy way

gcc -o executable -static program.o extras.o

Executable size: dynamically linked = 14221 bytes
statically linked = 1691082 bytes

```
8048218 <read_something>:
8048218:    55                    push   %ebp
8048219:    89 e5                 mov    %esp,%ebp
804821b:    83 ec 08              sub    $0x8,%esp
804821e:    83 ec 08              sub    $0x8,%esp
8048221:    8d 45 fc              lea    0xfffffffc(%ebp),%eax
8048224:    50                    push   %eax
8048225:    68 f2 e4 08 08        push   $0x808e4f2
804822a:    e8 cd 04 00 00        call   80486fc <scanf>
804822f:    83 c4 10              add    $0x10,%esp
8048232:    8b 45 fc              mov    0xfffffffc(%ebp),%eax
8048235:    c9                    leave
8048236:    c3                    ret
```

```
080486fc <scanf>:
80486fc:    55                    push   %ebp
.....
8048711:    e8 ba 34 01 00        call   805bbd0 <_IO_vfscanf>
.....
```

Just a bit more of this

scanf is a rather complicated function. It does lots before getting to the point where a true system action is needed. The most relevant action related to the task at hand is:

```
080651c0 <__libc_read>:
 80651c0:    53                  push   %ebx
 80651c1:    8b 54 24 10         mov    0x10(%esp,1),%edx
 80651c5:    8b 4c 24 0c         mov    0xc(%esp,1),%ecx
 80651c9:    8b 5c 24 08         mov    0x8(%esp,1),%ebx
 80651cd:    b8 03 00 00 00      mov    $0x3,%eax
 80651d2:    cd 80               int    $0x80
 80651d4:    5b                  pop    %ebx
 80651d5:    3d 01 f0 ff ff      cmp    $0xfffff001,%eax
 80651da:    0f 83 70 da fe ff   jae    8052c50 <__syscall_error>
 80651e0:    c3                  ret
```

The single statement responsible for passing control to the system is int. This is an implementation of what we refer to as a **system call**.

CPU states

Problem (user) state

Some instructions (called privileged instructions) are illegal. To execute them the CPU must switch to the ...

System (supervisor) state

Privileged instructions are those machine instructions that directly reference fundamental resources:

- ➔ peripheral devices
- ➔ memory mapping
- ➔ CPU state

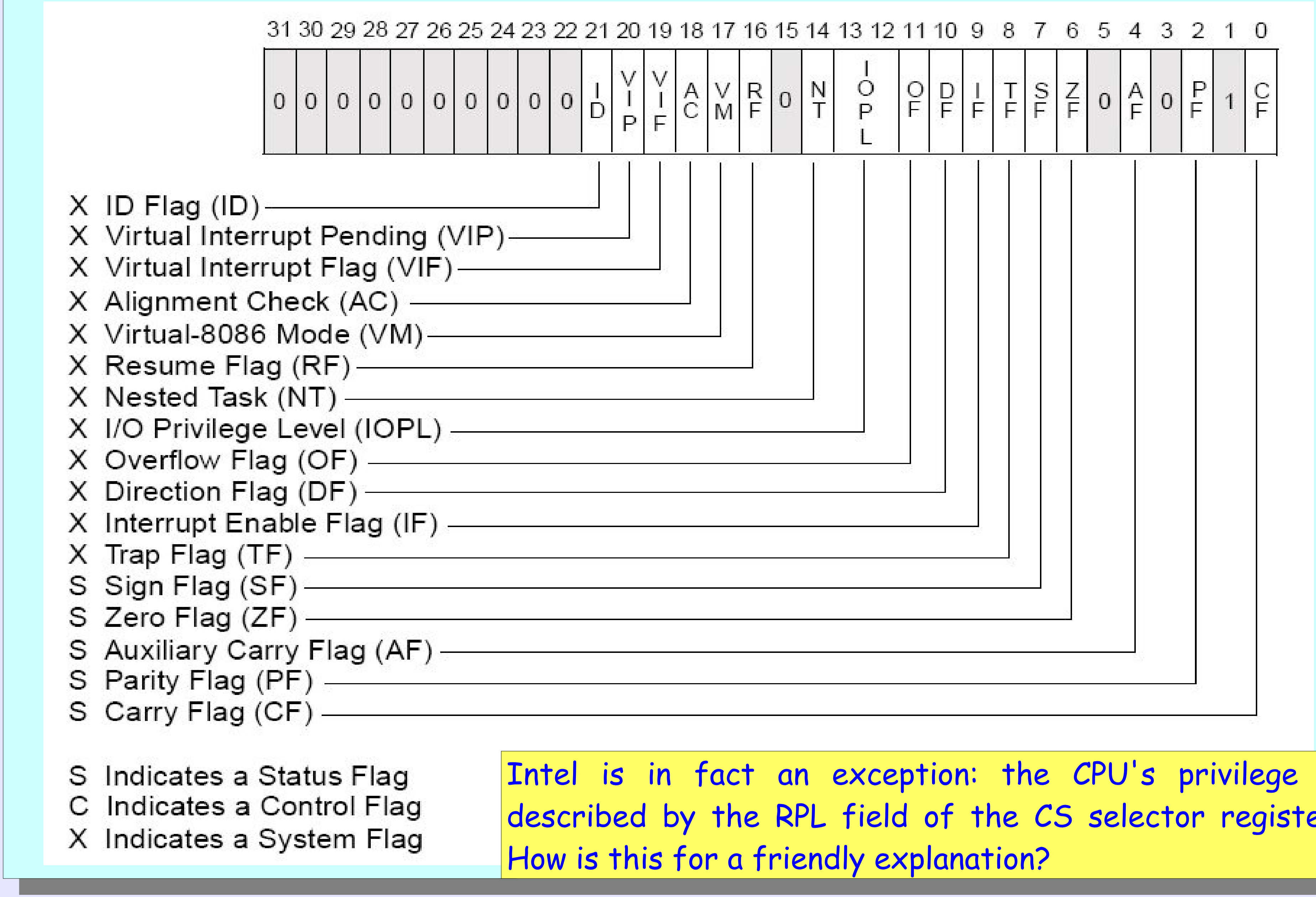
State transition

Programs execute in problem state. They can affect the state of the CPU in a very restricted way by performing supervisor calls. This is typically done by executing a special (non-privileged) machine instruction.

By executing a supervisor call the program invokes a function and, at the same time, assumes the system/supervisor state. The restriction consists in the fact that the function is located within the system area and cannot be declared by the user.

The CPU state is usually (should I say, historically) described by the contents of a special register called PSW, PSR, or whatever. We will be calling it PSW (Program Status Word).

PSW (EFLAGS) on the Intel



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13 12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	ID	VIP	VIF	AC	VM	RF	0	NT	IOPL	OF	DF	IF	TF	SF	ZF	0	AF	0	PF	1	CF

X ID Flag (ID)
X Virtual Interrupt Pending (VIP)
X Virtual Interrupt Flag (VIF)
X Alignment Check (AC)
X Virtual-8086 Mode (VM)
X Resume Flag (RF)
X Nested Task (NT)
X I/O Privilege Level (IOPL)
X Overflow Flag (OF)
X Direction Flag (DF)
X Interrupt Enable Flag (IF)
X Trap Flag (TF)
S Sign Flag (SF)
S Zero Flag (ZF)
S Auxiliary Carry Flag (AF)
S Parity Flag (PF)
S Carry Flag (CF)

S Indicates a Status Flag
C Indicates a Control Flag
X Indicates a System Flag

Intel is in fact an exception: the CPU's privilege is described by the RPL field of the CS selector register. How is this for a friendly explanation?

Fossils are more educational

```
63                                     31         24                       0
+--------+----+-+-+-+-+-+----------------+--+--+----+------------------------+
|SYS MASK|MKEY|A|M|W|P| Interrupt Code |IL|CC|PMSK|   Instruction Address  |
+--------+----+-+-+-+-+-+----------------+--+--+----+------------------------+
```

IBM 360 PSW

15															0
T		S			I	I	I				X	N	Z	V	C

MC68000 SR

Why the two modes?

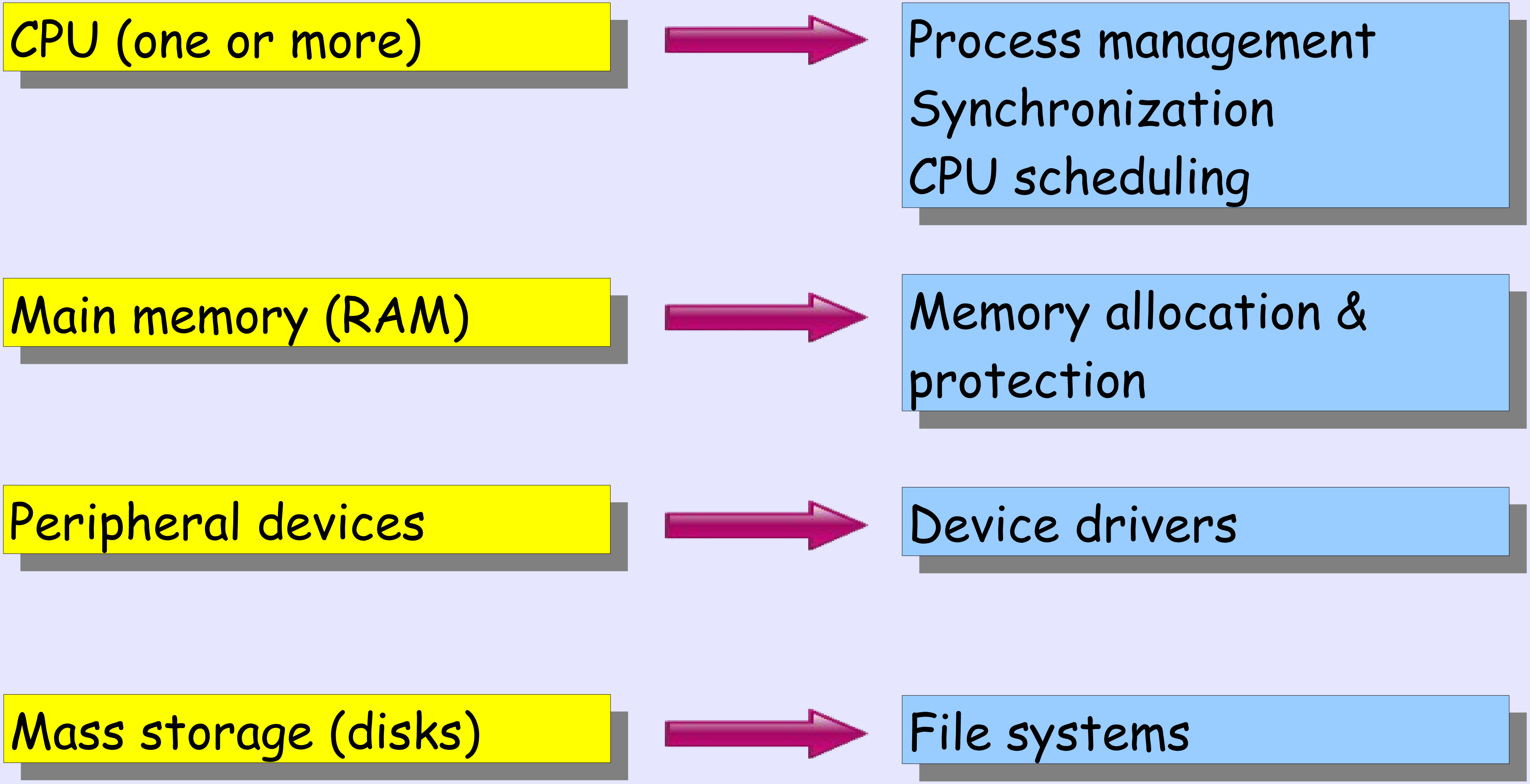
- The system needs to protect itself from the user program. We don't want it to crash when the program misbehaves.

- The system needs to protect multiple programs against themselves. One program should not be able to damage another program, e.g., belonging to a different user.

- The system wants to be in charge of the resources, such that it can present their consistent logical view to the programs (and to the user). This is impossible if the programs are allowed to access those resources directly.

Resources

According to some point of view, operating systems are all about resource management:



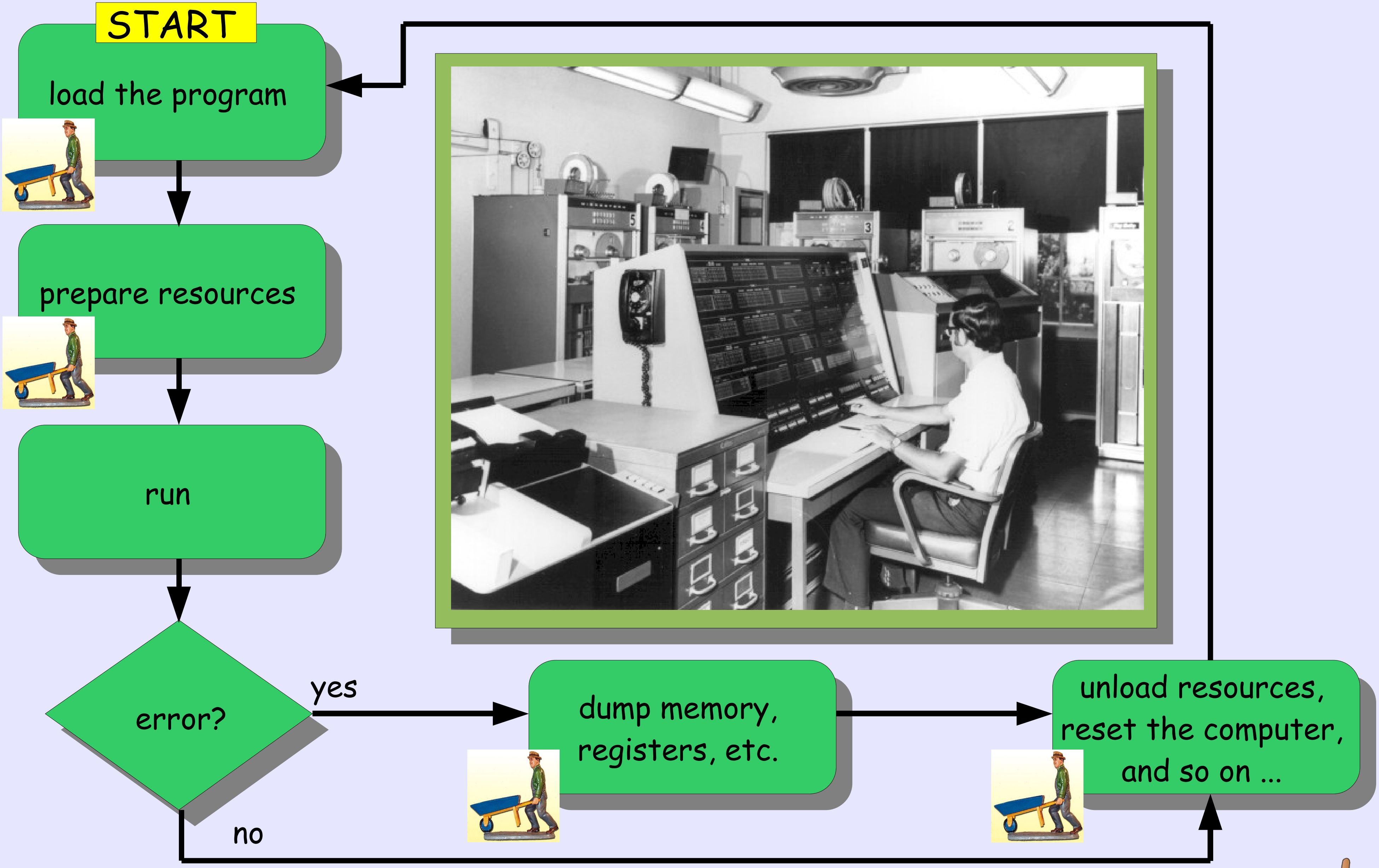
CPU (one or more) → Process management, Synchronization, CPU scheduling

Main memory (RAM) → Memory allocation & protection

Peripheral devices → Device drivers

Mass storage (disks) → File systems

Early computing



START

load the program

prepare resources

run

error?

yes

no

dump memory, registers, etc.

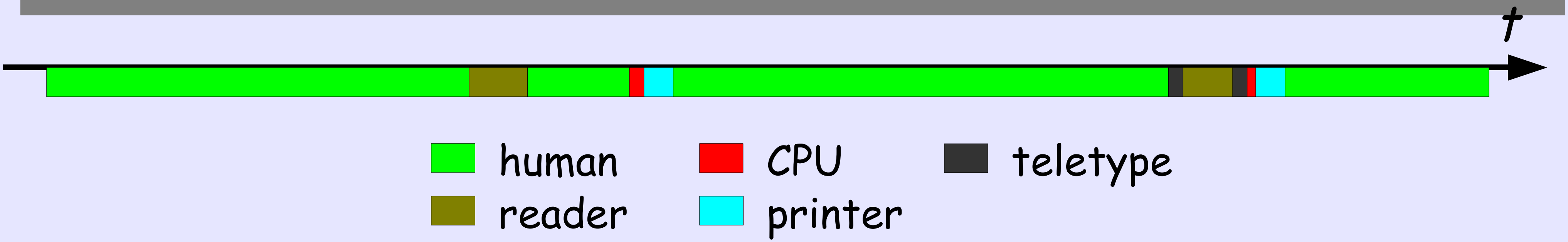
unload resources, reset the computer, and so on ...

Performance issues

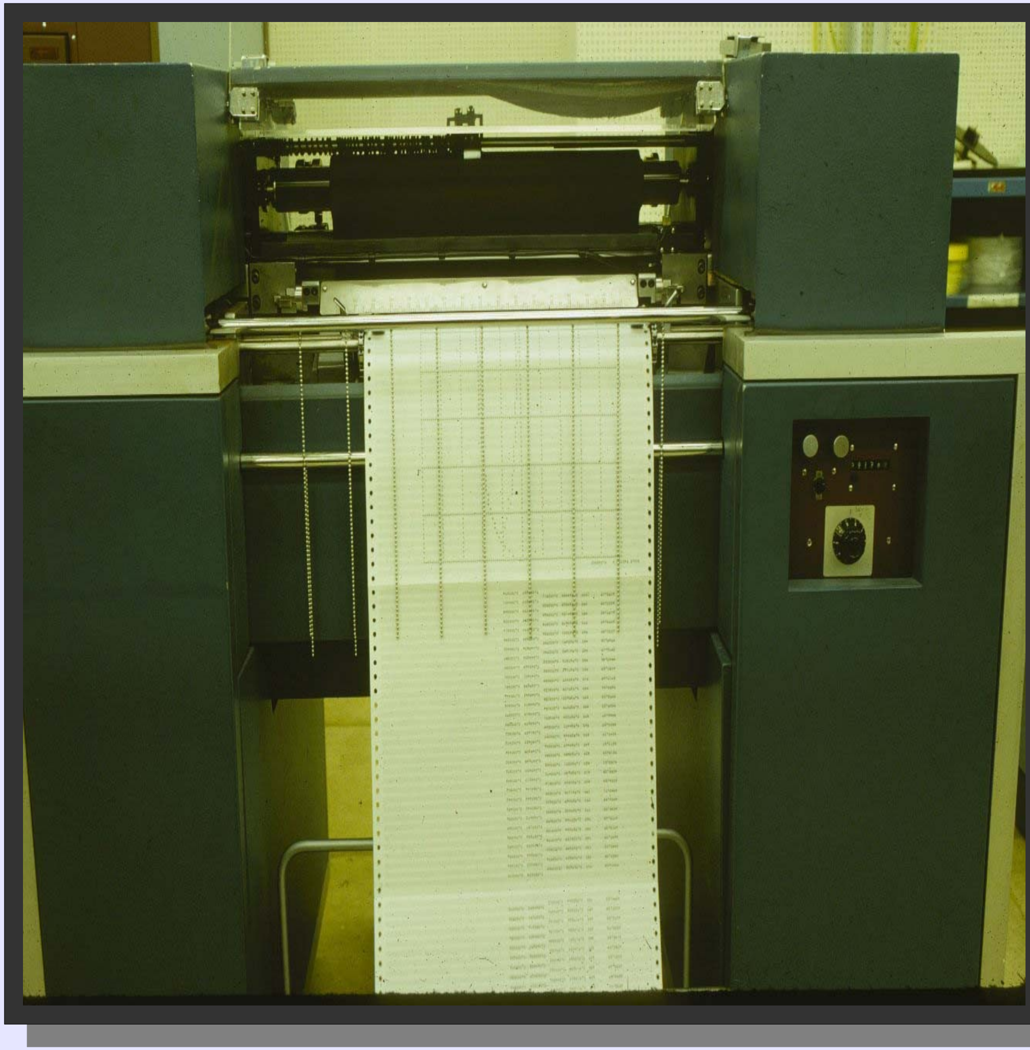
CPU and memory were very expensive, yet there was a lot of demand for processing power.

The human factor would considerably reduce the system's performance. Its effective throughput was a small fraction of what it could be, if the CPU were able to do something useful all the time.

The human factor was only a part of the problem. With only one program in the system, only one device at a time could be kept busy.



Some old peripheral devices



line printer

punched card reader

Two ideas that changed the world

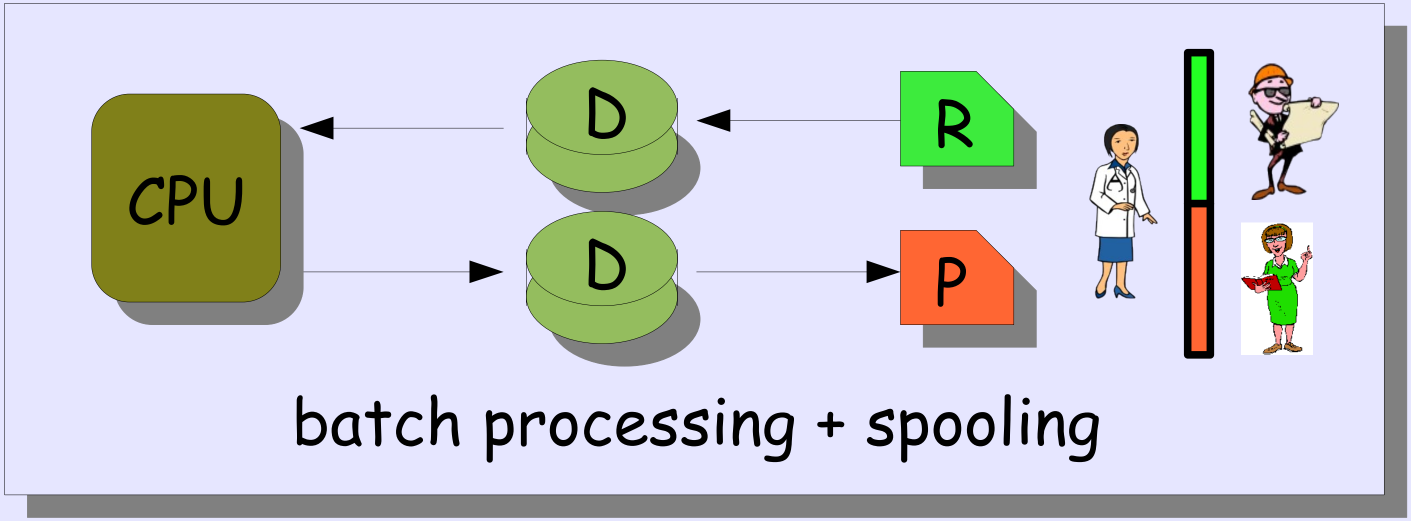
(of computing)

Batch processing, i.e.,

remove the clumsy human being from the computer room

Multiprogramming, i.e.,

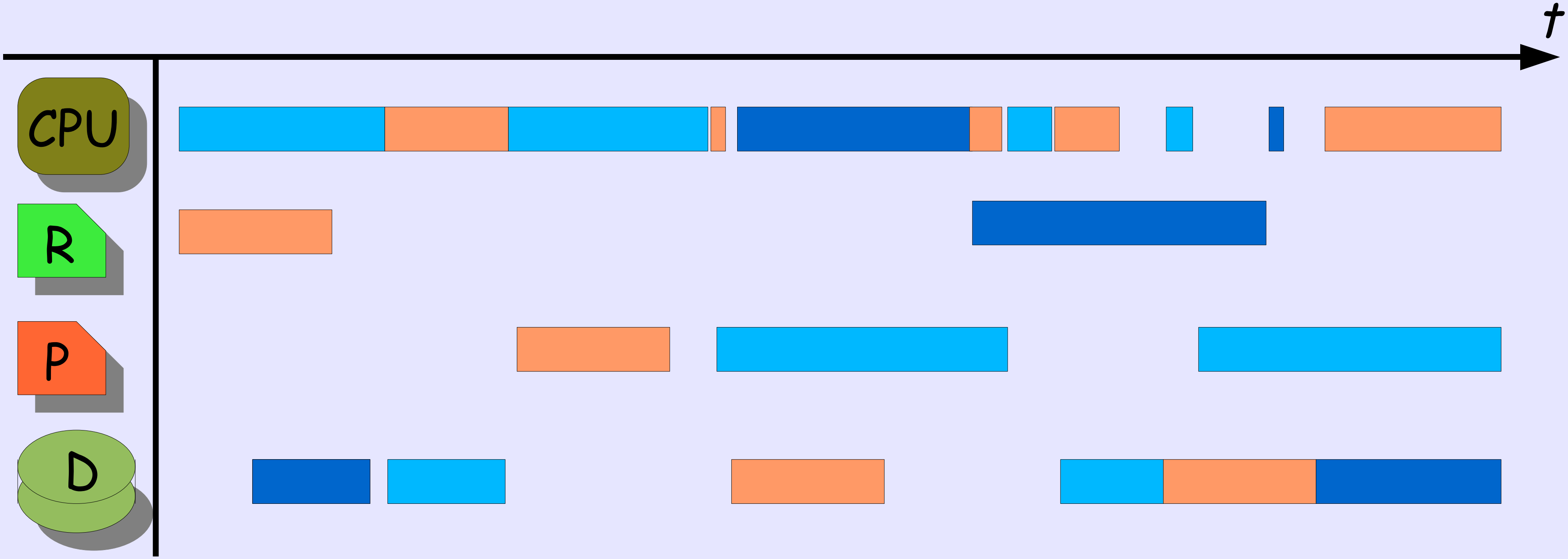
find something to do for as many hardware components as possible



batch processing + spooling

Multiprogramming

As an idea aimed at improving the efficiency of computer components, multiprogramming only makes sense if the different components can operate autonomously.



Channels

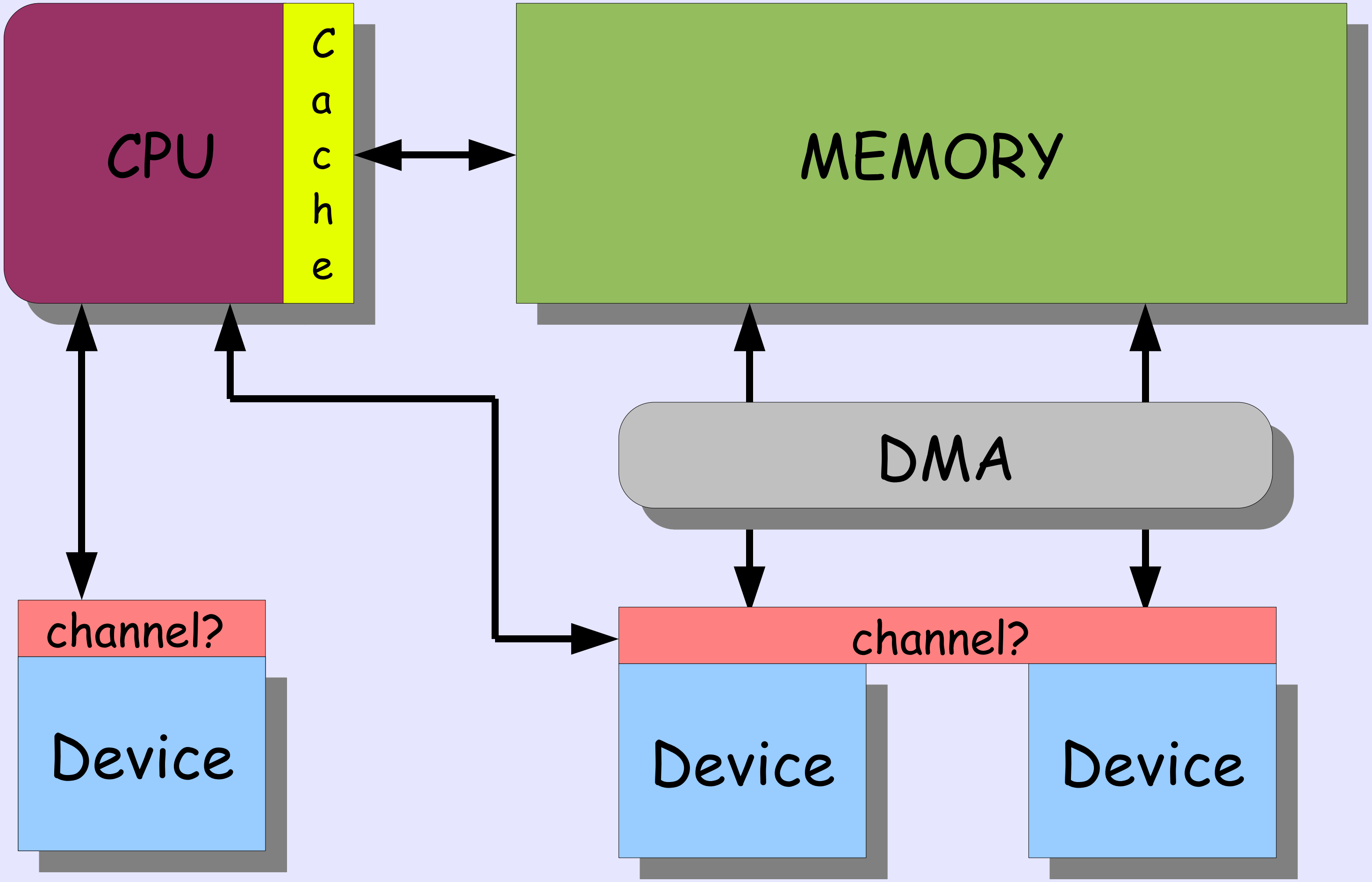
Conceptually, we envision a special type of processor associated with an independent peripheral device. Historically, such a processor is called a channel.

Sometimes, the channel is built into the device.

Sometimes, several devices share a single channel that supervises them collectively. Example: a SCSI controller servicing multiple disks. Depending on the organization (complication) of the channel, there may be restrictions on the degree of parallelism and independence.

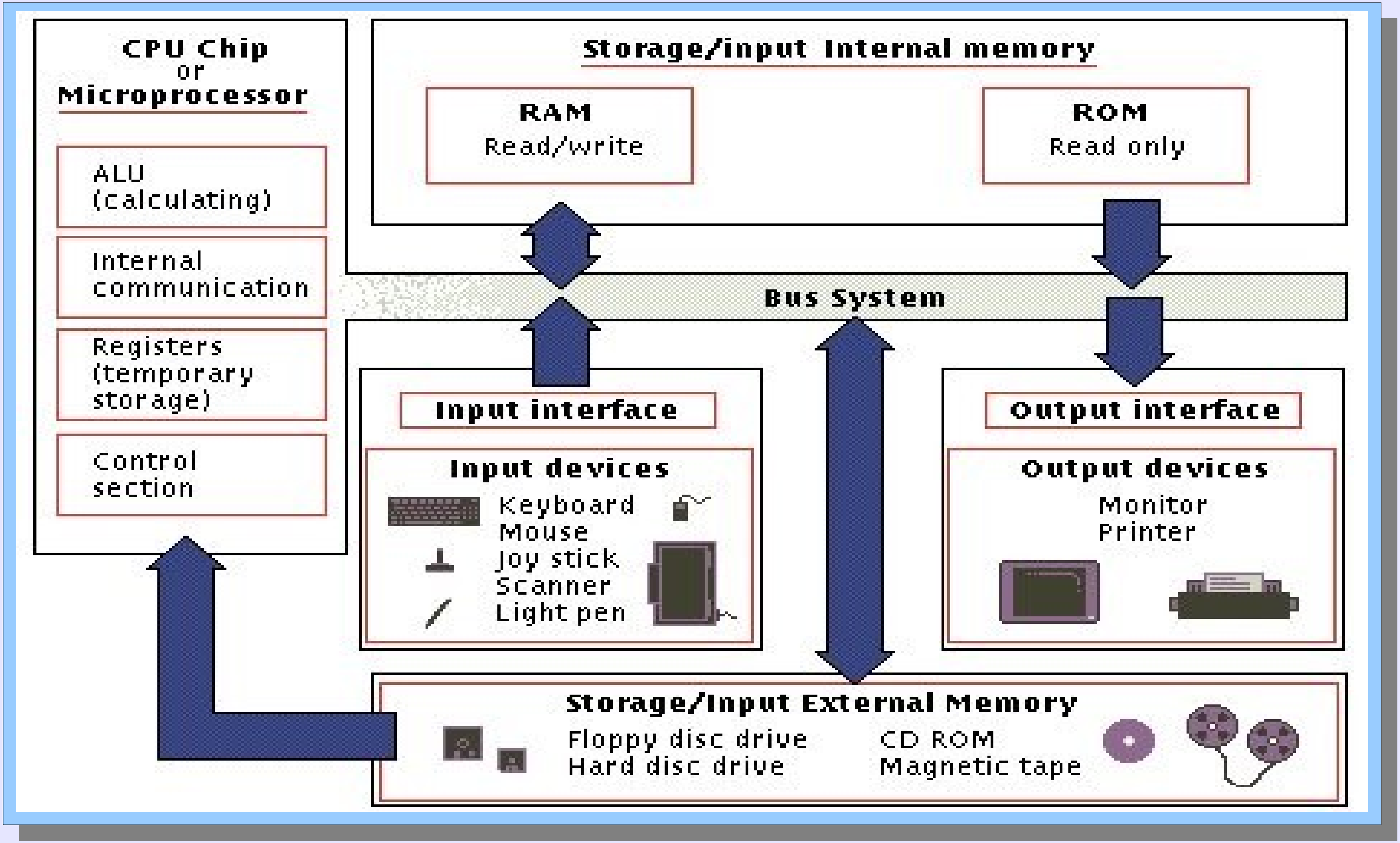
Sometimes, the device is truly dumb. Not too many of those these days, except in the microcontroller world.

Computer organization



This isn't how computers are physically wired, but how they appear to the OS.

Typical physical structure



A brief summary

- The OS provides a layer of interfaces between user programs and (physical) resources.
- The OS is responsible for presenting a consistent structure of logical resources being mapped into physical resources. This involves fair and safe sharing and partitioning (because we have multiprogramming).
- The OS should be able to handle multiple programs in a way that preserves their independence.
- Efficiency is an important objective.
- Owing to the independent operation of I/O channels, even if there is only one CPU, the OS should still be able to control multiple activities in parallel.

Activities in the system

Typically, during its lifetime, your program can be preempted and then resumed, possibly many times. The system can handle other activities while running your program. How does that work?

There are computers with multiple CPUs. Then, different activities could be handled by different CPUs. But most computers have only one CPU (or two, like in “dual core,” CPUs) each. So there must be a way to multiplex the CPU(s) among the multiple activities.

The proper view is to consider the CPU (or the multiple CPUs) as a resource that the multiple activities in the system compete for.

This view is general. Operating Systems are about implementing fair and efficient strategies for handling competitions for resources. The competitors are called **processes**.

Types of activities

Processes

Identifiable activities that can be stopped and resumed. They are described by special data structures and characterized by well-known sets of attributes. One such attribute is the process status.

Interrupt service routines (+ system calls)

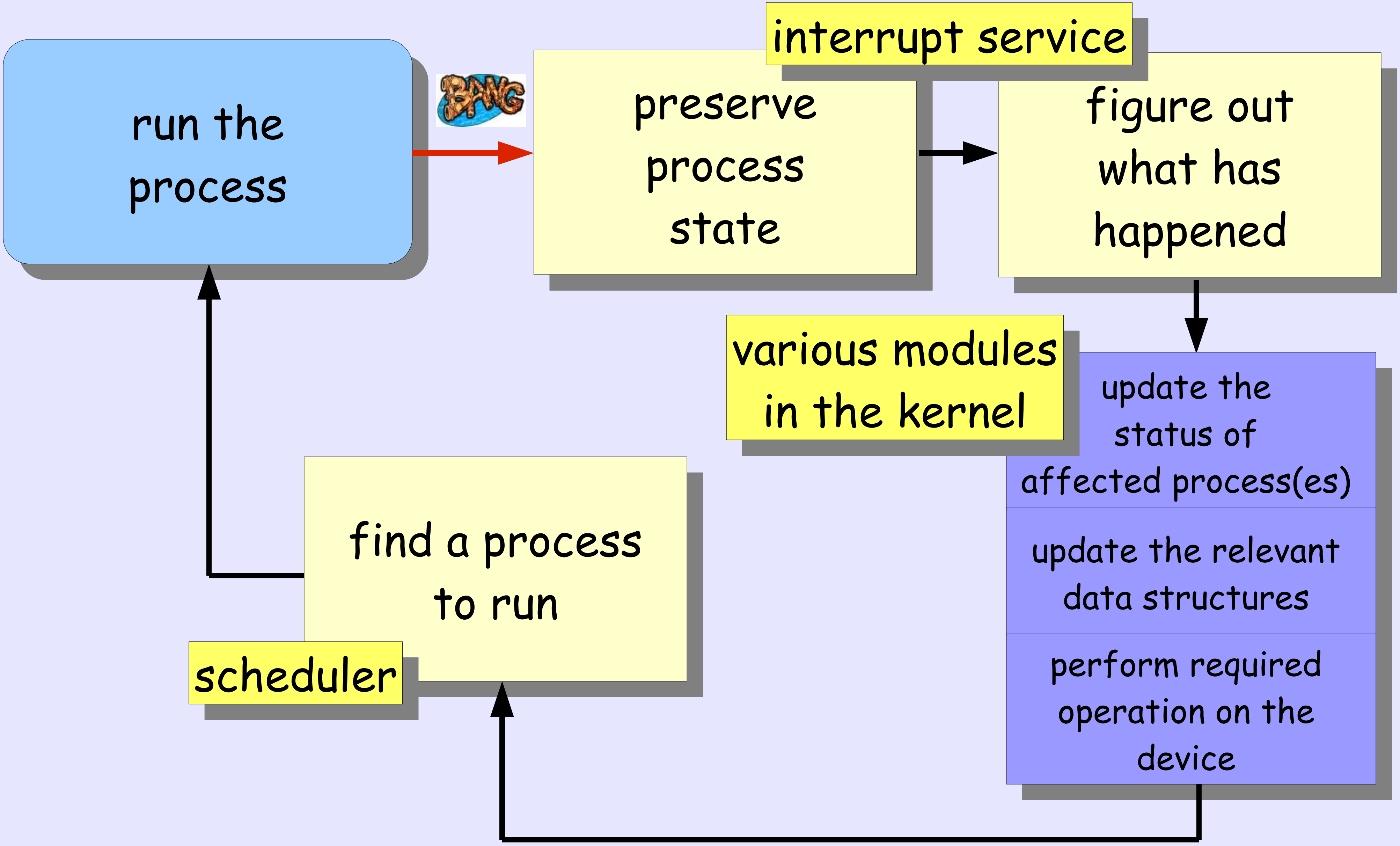
Activities to handle events in the system. Everything that happens in the system that may affect the status of a process boils down to the execution of an interrupt service routine.

Other?

Nope! The entire Kernel is in fact a bunch of interrupt service routines, sometimes in a minor disguise.

Kernel == whatever remains when you remove processes.

OS operation cycle



run the process

preserve process state

interrupt service

figure out what has happened

various modules in the kernel

update the status of affected process(es)

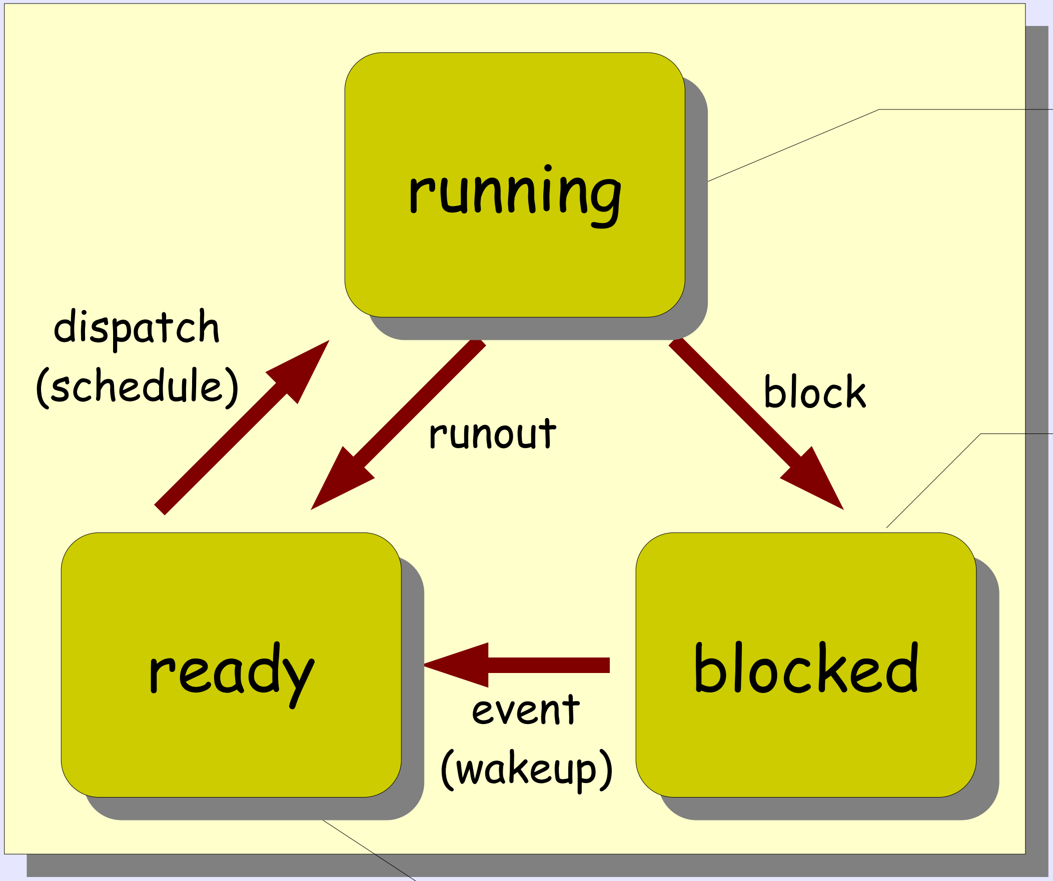
update the relevant data structures

perform required operation on the device

find a process to run

scheduler

Basic process states

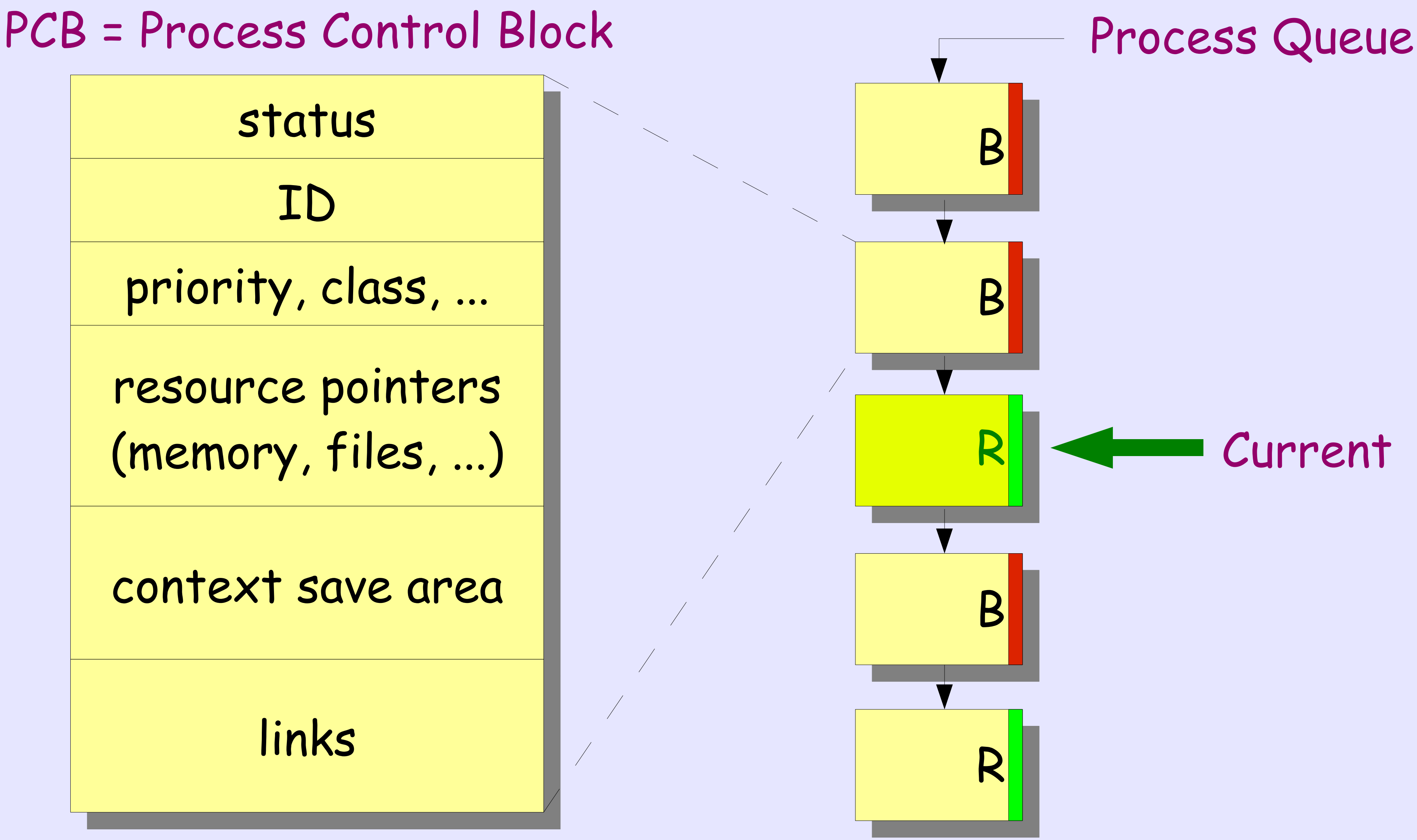


The process currently has the CPU

The process cannot proceed until something (external) happens

The process is ready to use the CPU whenever it becomes available

System data structures



PCB = Process Control Block

status

ID

priority, class, ...

resource pointers (memory, files, ...)

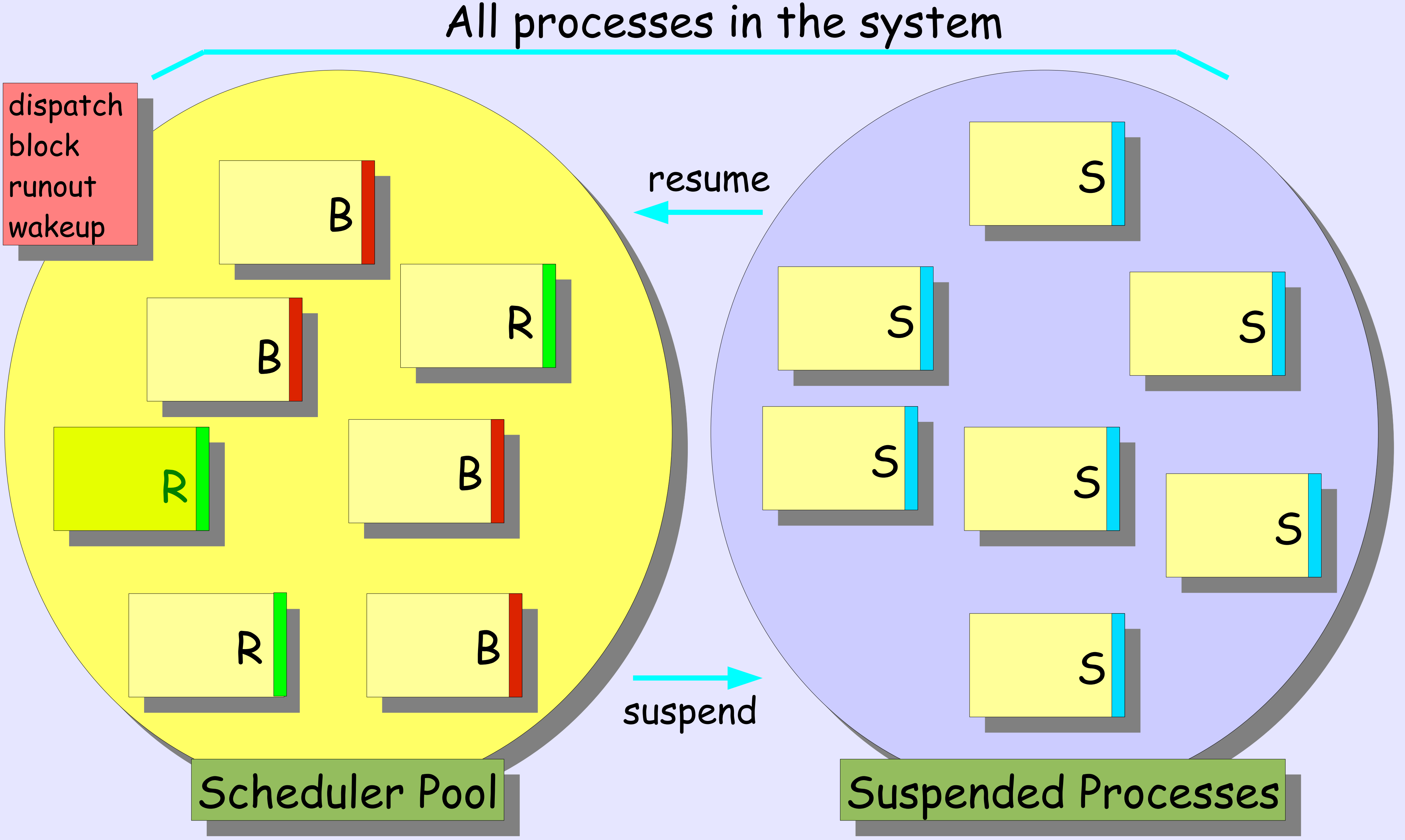
context save area

links

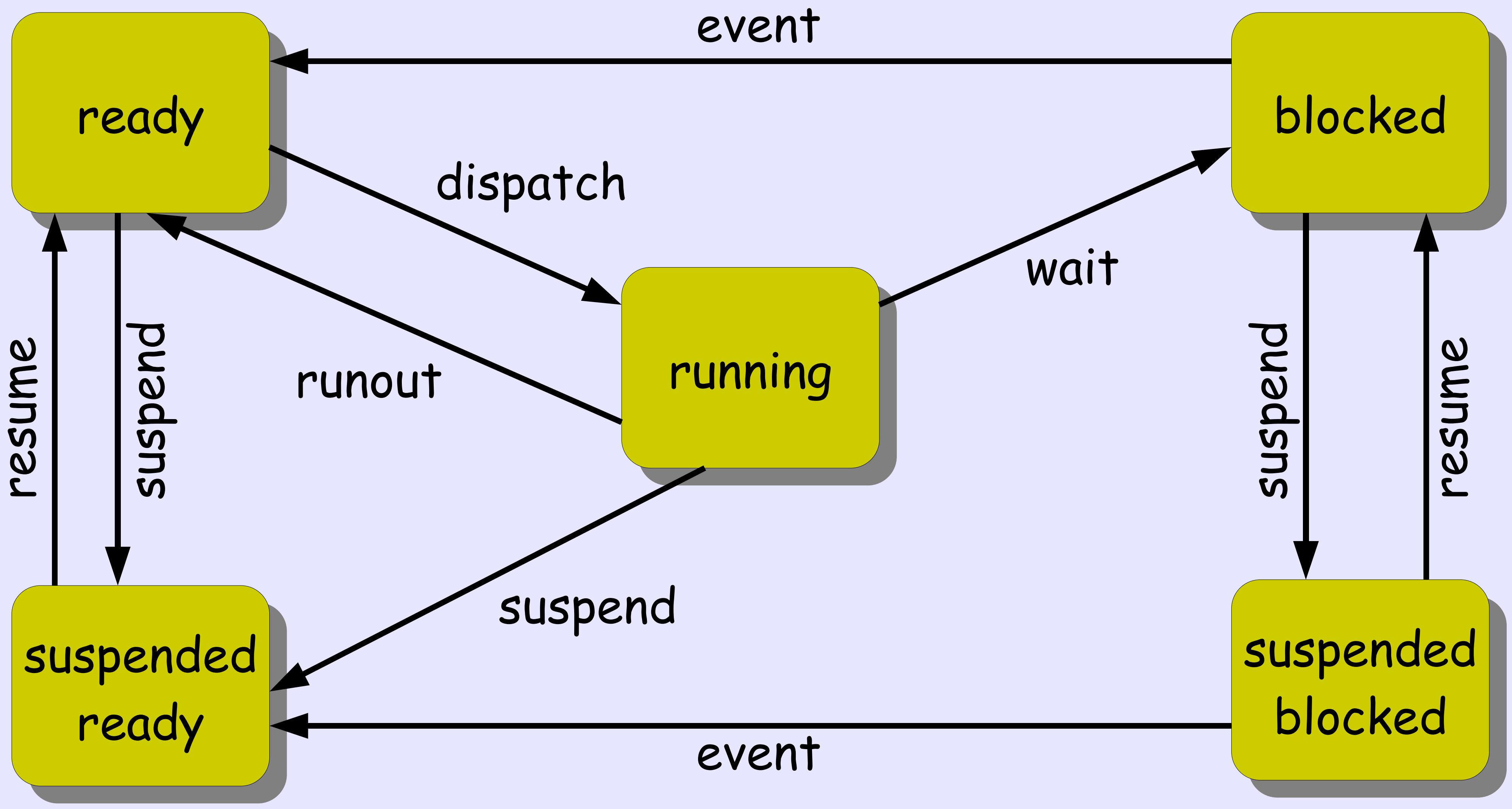
Process Queue

Current

Another view

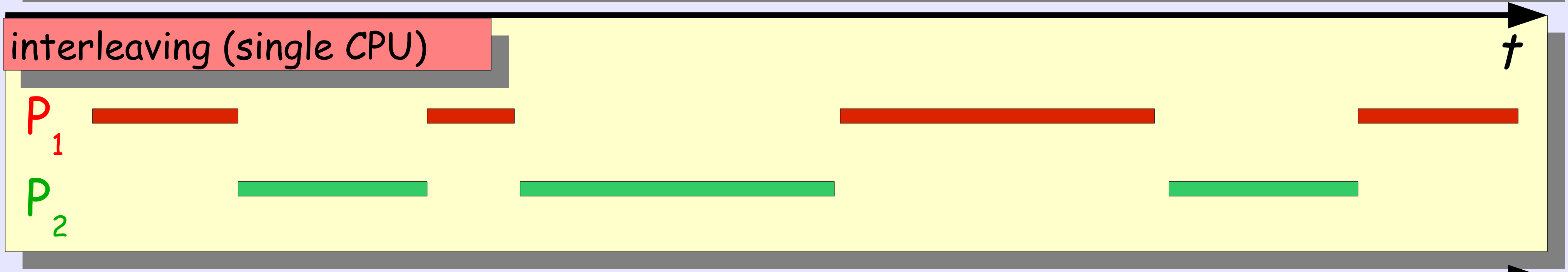


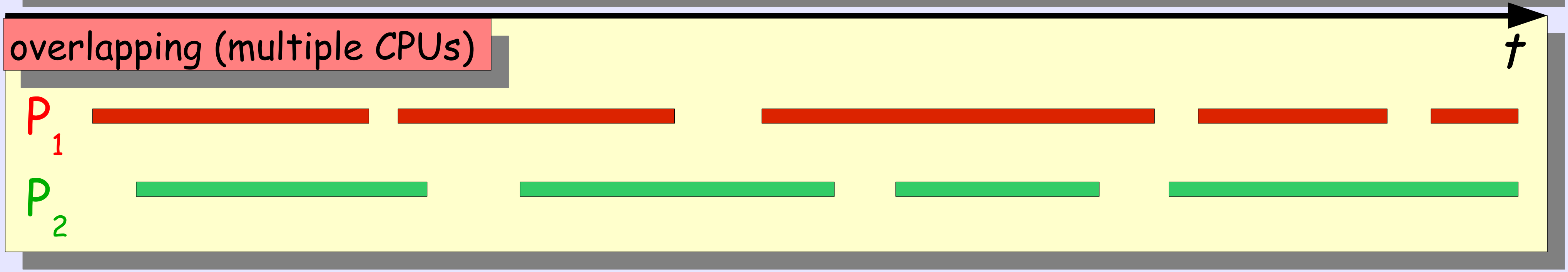
A more complete transition diagram



Process synchronization

Suppose that we have a number of concurrent processes (for example two).





Generally, unless we employ some mechanism, we have no way of telling in which way the instructions of the two processes will be interwoven in time. Sometimes it makes a difference.

Example

```
reader_1 (void) {
    int k;
    while (1) {
        fscanf (d1, “%d”, &k);
        sum += k;
    }
}
```

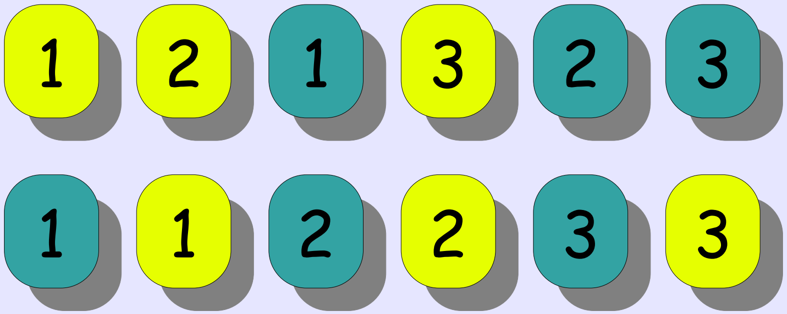
```
reader_2 (void) {
    int k;
    while (1) {
        fscanf (d2, “%d”, &k);
        sum += k;
    }
}
```

```
int sum = 0;
FILE *d1, *d2;
...
main () {
    ...
    start (reader_1);
    start (reader_2);
    ...
}
```

Example (cntd)

sum = 0

reader_1		reader_2	
0	read k (=1)	0	read k (=2)
1	load sum	1	load sum
2	add k to sum	2	add k to sum
3	store sum	3	store sum



sum = 2

sum = 1

Mutual exclusion

Eliminating unfriendly configurations of interleaved instructions:

```
reader_1 (void) {
    int k;
    while (1) {
       fscanf (d1, "%d", &k);
       beware ("sum");
       sum += k;
       worry_no_more ("sum");
    }
}
```

```
reader_2 (void) {
    int k;
    while (1) {
       fscanf (d2, "%d", &k);
       beware ("sum");
       sum += k;
       worry_no_more ("sum");
    }
}
```

Note that it makes sense to parameterize the operations (there may be many independent "critical sections").

How to do it?

Let us give it a try assuming no special features in the underlying system.

Prerequisites:

- Processes execute concurrently (it makes no difference whether they are interleaving or overlapping). We never know which one will go next and for how long.
- Consistency of basic operations is guaranteed in hardware, i.e., if two processes try to store something in the same location at the same time, the location is going to end up with one thing or the other but not with garbage.

The generic problem

```
process1 ( ... ) {
    ...
    some irrelevant startup
    ...
    while (true) {
        ...
        whatever
        ...
        enter_section(...);
        critical operations
        exit_section(...);
        ...
        whatever
        ...
    }
}
```

```
process2 ( ... ) {
    ...
    some irrelevant startup
    ...
    while (true) {
        ...
        whatever
        ...
        enter_section(...);
        critical operations
        exit_section(...);
        ...
        whatever
        ...
    }
}
```

Attempt 1

```
process1 ( ... ) {
    ...
    while (true) {
       ...
       while (entered);
       entered = 1;
       critical operations
       entered = 0;
       ...
    }
}
```

```
process2 ( ... ) {
    ...
    while (true) {
       ...
       while (entered);
       entered = 1;
       critical operations
       entered = 0;
       ...
    }
}
```

```
    entered = 0;
    start (process1);
    start (process2);
```

Attempt 2

```
process1 ( ... ) {
    ...
    while (true) {
        ...
        while (pnum == 2);
        critical operations
        pnum = 2;
        ...
    }
}
```

```
process2 ( ... ) {
    ...
    while (true) {
        ...
        while (pnum == 1);
        critical operations
        pnum = 1;
        ...
    }
}
```

```
    pnum = 1;
...
```

Exclusion “as such” is easy to achieve. But we would expect some flexibility and “decency” from what we would call a solution.

Postulates

- **No "almost working" solutions.** If you think that some interleaving scenarios are not very likely, think again.
- **Simplicity is an asset.** Most critical sections are short, and the operations guarding them should not overshadow their cost.
- **No locked-in patterns of behavior.** Do not assume that B will always follow A or anything like that.
- **Determinism.** The amount of waiting time should not depend on circumstances other than for how long the section is going to be actually busy (no *indefinite postponement*).
- **Generalizability.** *N* processes should not be much more difficult to handle than 2.

Attempt 3

```
process1 ( ... ) {
   ...
   while (true) {
      ...
      while (p2inside);
      p1inside = true;
      critical operations
      p1inside = false;
      ...
   }
}
```

```
process2 ( ... ) {
   ...
   while (true) {
      ...
      while (p1inside);
      p2inside = true;
      critical operations
      p2inside = false;
      ...
   }
}
```

```
   p1inside = p2inside = false;
...
```

Attempt 4

```
process1 ( ... ) {
    ...
    while (true) {
      ...
      p1trying = true;
      while (p2trying);
      critical operations
      p1trying = false;
      ...
    }
}
```

```
process2 ( ... ) {
    ...
    while (true) {
      ...
      p2trying = true;
      while (p1trying);
      critical operations
      p2trying = false;
      ...
    }
}
```

```
    p1trying = p2trying = false;
...
```

Attempt 5

```
process1 ( ... ) {
    ...
    while (true) {
        ...
        p1trying = true;
        while (p2trying) {
            p1trying = false;
            wait_a_little ();
            p1trying = true;
        }
        critical operations
        p1trying = false;
        ...
    }
}
```

```
process2 ( ... ) {
    ...
    while (true) {
        ...
        p2trying = true;
        while (p1trying) {
            p2trying = false;
            wait_a_little ();
            p2trying = true;
        }
        critical operations
        p2trying = false;
        ...
    }
}
```

indefinite postponement

```
    p1trying = p2trying = false;
...
```

Attempt 6: Dekker's Algorithm

```
process1 ( ... ) {
    ...
    while (true) {
       ...
       p1trying = true;
       while (p2trying) {
          if (turn == 2) {
             p1trying = false;
             while (turn == 2);
             p1trying = true;
          }
       }
       critical operations
       turn = 2;
       p1trying = false;
       ...
    }
}
```

```
process2 ( ... ) {
    ...
    while (true) {
       ...
       p2trying = true;
       while (p1trying) {
          if (turn == 1) {
             p2trying = false;
             while (turn == 1);
             p2trying = true;
          }
       }
       critical operations
       turn = 1;
       p2trying = false;
       ...
    }
```

```
    p1trying = p2trying = false;
    turn = 1;

...
```

Attempt 7: Peterson's Algorithm

```
process1 ( ... ) {
    ...
    while (true) {
        ...
        p1trying = true;
        cookie = 2;
        while (p2trying &&
                cookie == 2);
        critical operations
        p1trying = false;
        ...
    }
}
```

```
process2 ( ... ) {
    ...
    while (true) {
        ...
        p2trying = true;
        cookie = 1;
        while (p1trying &&
                cookie == 1);
        critical operations
        p2trying = false;
        ...
    }
}
```

```
    p1trying = p2trying = false;
    cookie = ...;
...
```

Attempt 8: Hyman's Algorithm

```
process1 ( ... ) {
    ...
    while (true) {
        ...
        p1trying = true;
        while (turn != 1) {
            while (p2trying);
            turn = 1;
        }
        critical operations
        p1trying = false;
        ...
    }
}
```

```
process2 ( ... ) {
    ...
    while (true) {
        ...
        p2trying = true;
        while (turn != 2) {
            while (p1trying);
            turn = 2;
        }
        critical operations
        p2trying = false;
        ...
    }
}
```

```
    p1trying = p2trying = false;
    turn = 1;
...
```

This was pretty much just for fun

The best algorithms are relatively simple, but they only work for two processes. Generalizations are possible but:

- *N* must be known a priori
- the algorithms become awfully complicated

Real-life solutions are based on stronger prerequisites than variable sharing:

- the OS can provide requisite tools for processes (available as system calls)
- some hardware tools can be used in desperate situations

Suppose there is a single CPU

```
process1 ( ... ) {
    ...
    while (true) {
        ...
        dont_preempt ();
        critical operations
        preempting_ok ();
        ...
    }
}
```

```
process2 ( ... ) {
    ...
    while (true) {
        ...
        dont_preempt ();
        critical operations
        preempting_ok ();
        ...
    }
}
```

Neat system calls, eh? Unfortunately:

- not safe: the process assumes full charge of a fundamental resource, i.e., the CPU.
- cannot be parameterized (one global critical section)

A better way ...

... would be to use these operations internally. For example, the OS could implement a system call:

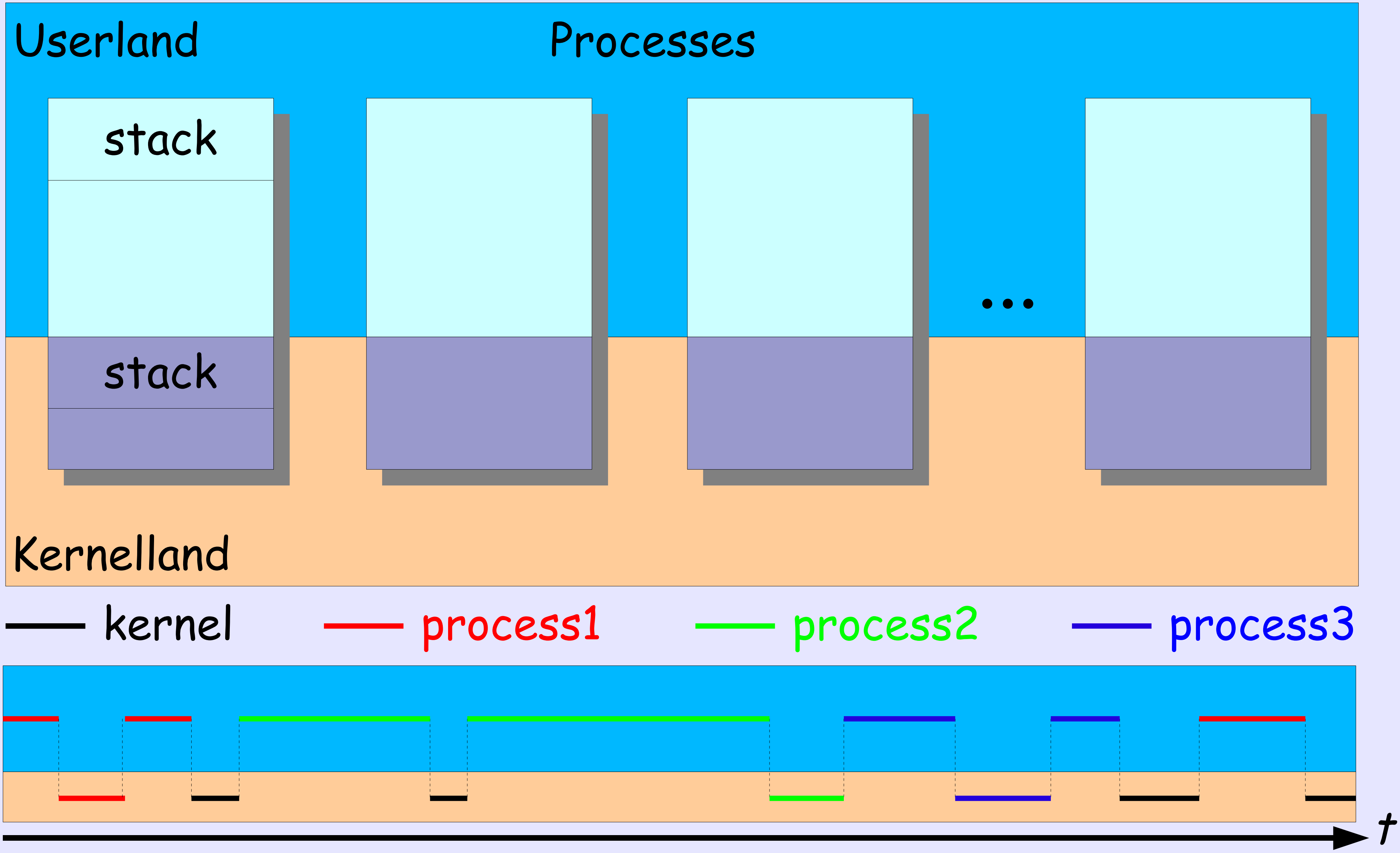
```
bool section_busy (int *g) {
    int result;
    dont_preempt ();
    result = *g;
    *g = true;
    preempting_ok ();
    return result;
};
```

```
   ...
while (section_busy (&flag));
    critical operations
flag = false;
   ...
```

- works for any number of processes
- appears simple and trivially parameterizeable

- busy waiting; couldn't the system do something useful instead of spinning futilely at 'while (enter_section (&flag));' ?

Userland versus Kernelland



Taxonomy of interrupts

•Internal Interrupts (sometimes called traps or exceptions). They are caused by events occurring within the CPU, i.e., by explicit actions (or misbehavior) of programs, e.g.,

- system call
- division by zero
- illegal memory reference

•External Interrupts. They are caused by events occurring outside the CPU, e.g.,

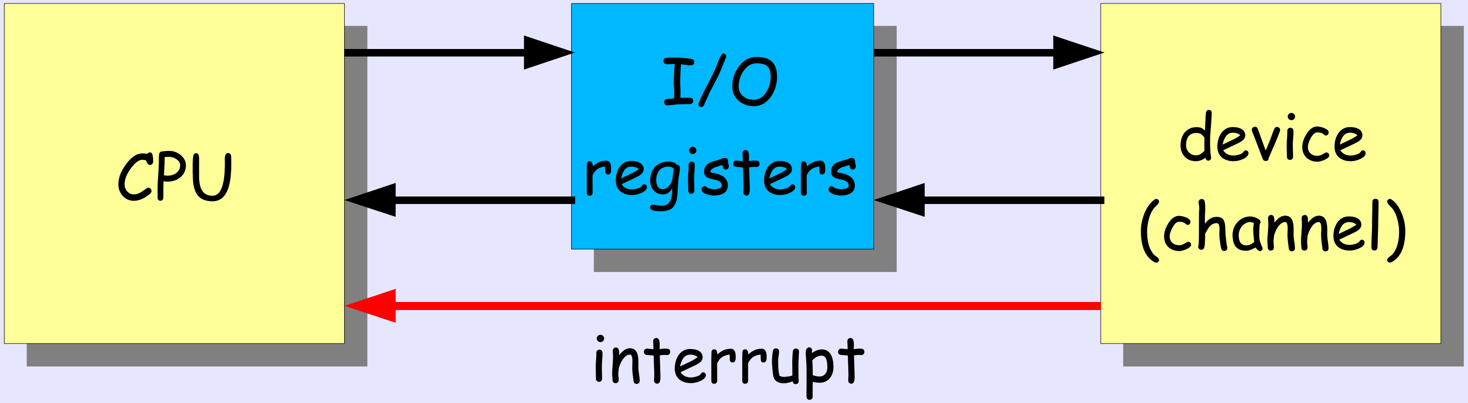
- I/O notifications
- timers going off
- reset button

External interrupts ...

... are asynchronous (w.r.t. the CPU activities). What does it mean?

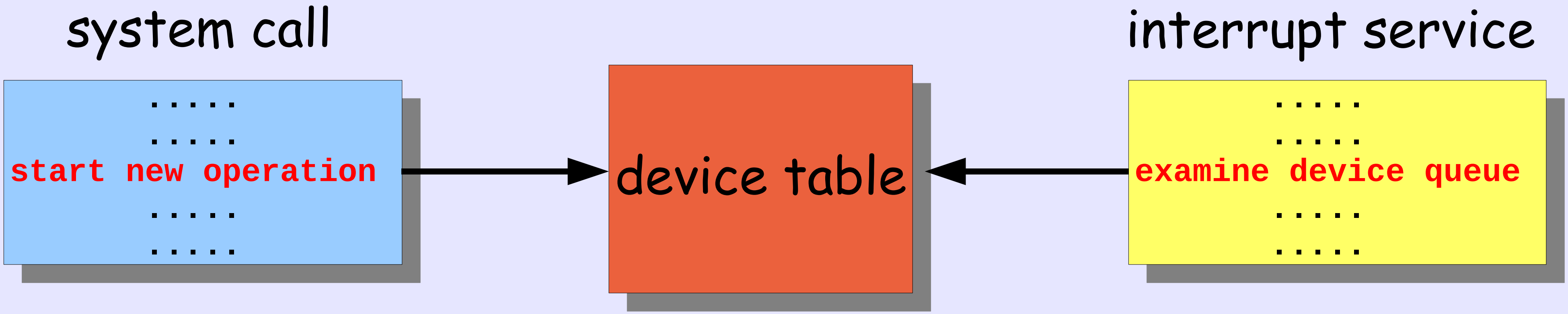
Remember that I/O devices are in fact idependent processors capable of sustaining non-trivial activities in parallel with the CPU.

Look at it this way: external interrupts provide handles whereby external devices can execute CPU code.



In principle, the I/O registers would suffice, but then the CPU would have to poll them for status change.

Critical sections revisited



Critical sections do not only apply to processes. The kernel has to deal with mutual exclusion problems involving interrupts.

Note that:

- I/O interrupts occur spontaneously
- naive solutions for critical sections won't work (imagine "busy waiting" in an interrupt)

Consequently, some hardware mechanism is needed.

External interrupts are maskable

This means that the CPU can request to hold an external interrupt until it explicitly resumes its reception.

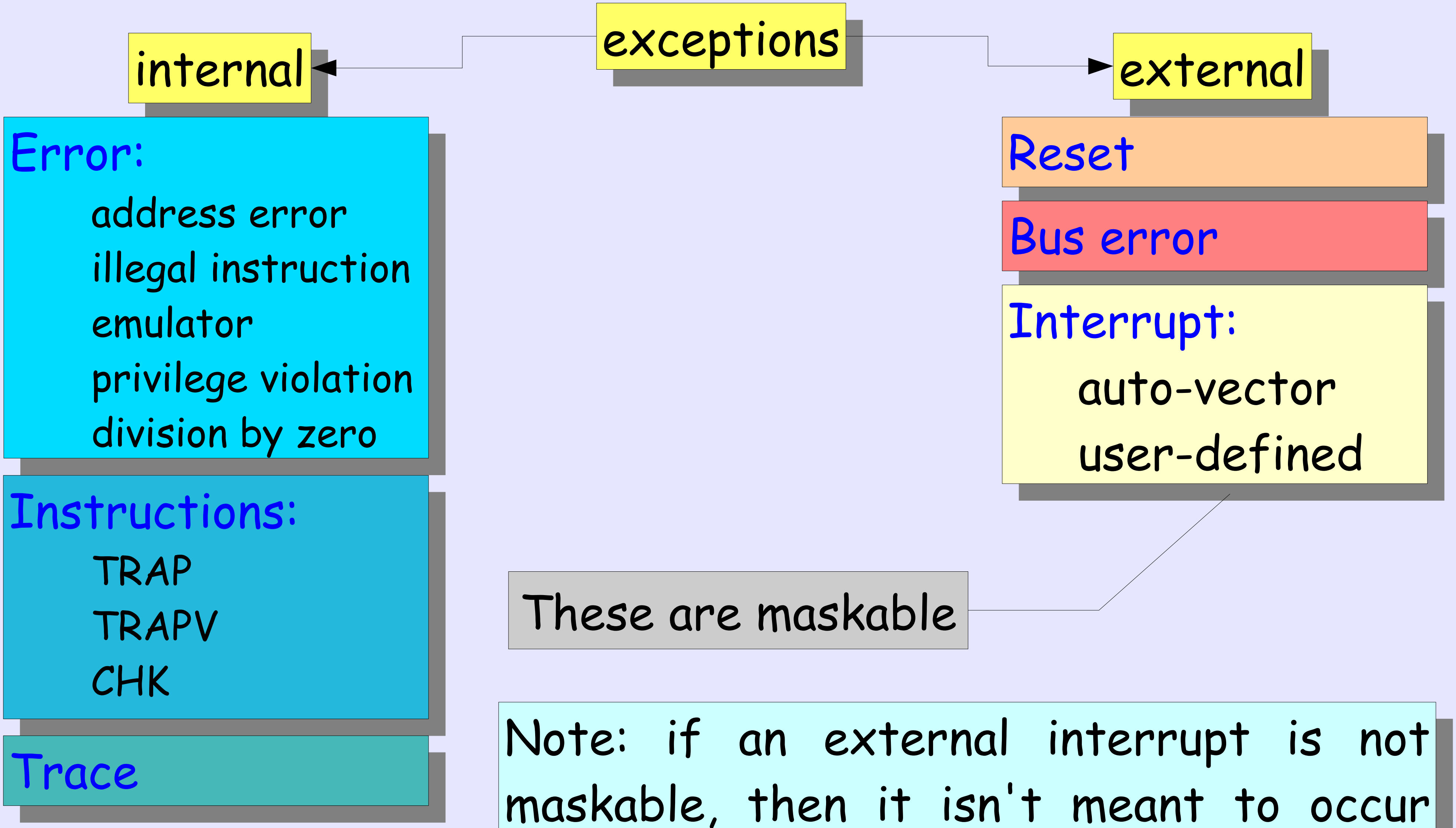
An interrupt occurring while it is masked remains pending and will be accepted when unmasked.

Note that some internal interrupts may also be maskable, e.g., integer overflow, floating point underflow (on some machines).

In contrast to external interrupts, a masked internal interrupt is irretrievably lost if it occurs. The philosophy of masking is different in the two cases:

external interrupts => hold on for a moment
internal interrupts => I don't care about these events

MC 68000 interrupts



exceptions

internal

Error:

- address error
- illegal instruction
- emulator
- privilege violation
- division by zero

Instructions:

- TRAP
- TRAPV
- CHK

Trace

external

Reset

Bus error

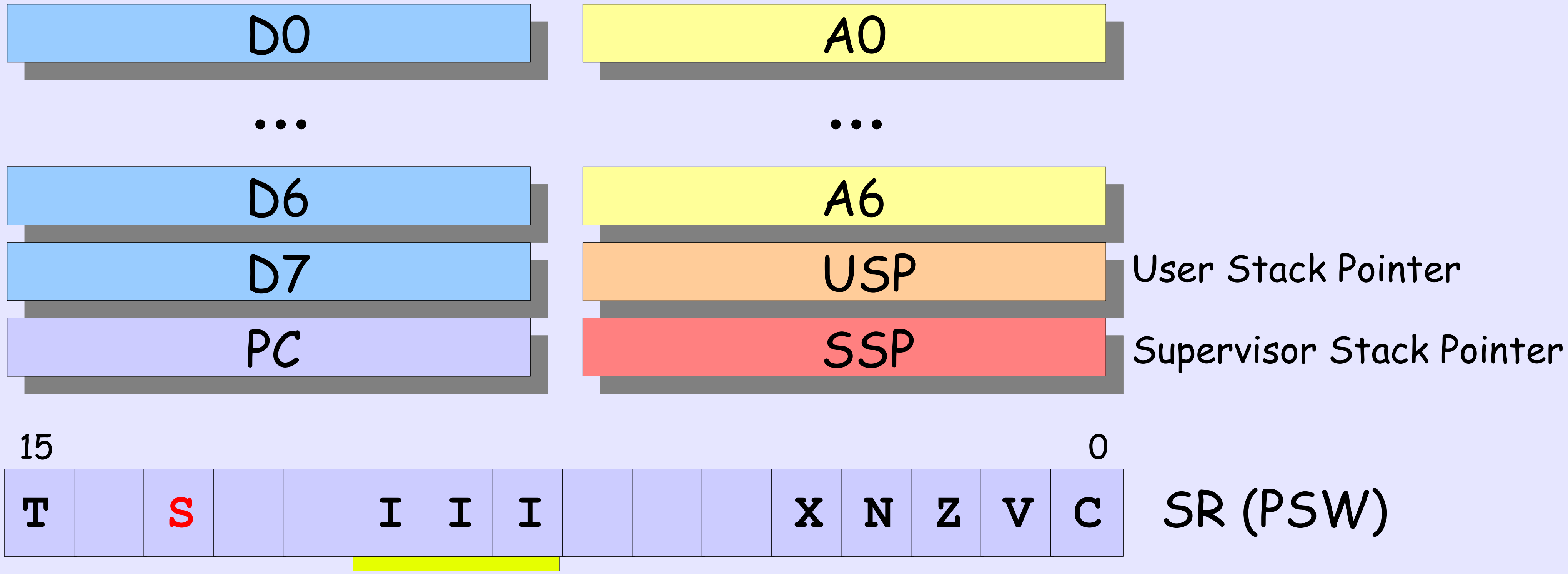
Interrupt:

- auto-vector
- user-defined

These are maskable

Note: if an external interrupt is not maskable, then it isn't meant to occur during normal operation.

MC 68000 CPU



D0

...

D6

D7

PC

A0

...

A6

USP User Stack Pointer

SSP Supervisor Stack Pointer

SR (PSW)

15															0
T		S			I	I	I				X	N	Z	V	C

External interrupts are grouped into 7 levels numbered 1-7. The three I bits indicate the current maximum level of masked interrupts. Level 7 is not maskable and is not to be used for "normal" interrupts.

MC 68000 “Exception” Service

Supervisor Stack Pointer is assumed as the effective SP

PC is pushed onto the stack

SR is pushed onto the stack

The S bit in SR is set

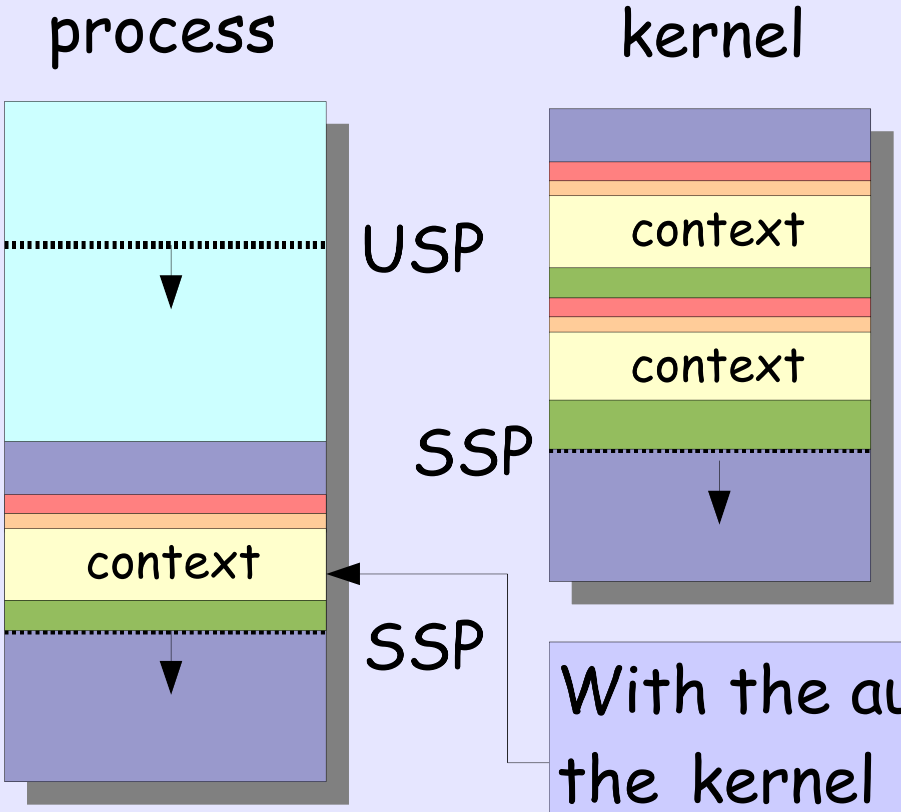
Crossing the Userland boundary

If this is an (external) interrupt, the I bits in SR are set to its level (effectively masking this level and all levels below)

The CPU determines the so called “vector number” based on the exception type and fetches from the memory location pointed by it the address of the service function. This address is loaded into PC.

Safely crossing the Userland boundary

The interrupt acceptance mechanism must allow the OS to preserve the state of the interrupted activity. Note that besides processes, interrupt service routines can be interruptible as well.



An interrupted interrupt service routine looks like a function that has (unwillingly) called another function. The new routine must preserve the context of the interrupted one.

With the automatic switch of the stack pointer, the kernel counterpart of the process's stack becomes a natural context save area.

Intel 486/586

Interrupts (exceptions):

- faults, i.e., program errors
- aborts, i.e., unrestartable faults
- traps, i.e., program tracing
- software interrupt (supervisor call)
- true interrupts, i.e., external ones

All true interrupts, except one left just in case, are maskable with a single bit (called IF) in EFLAGS (Intel's PSW). There are two (privileged) machine instructions: STI (interrupt enable) and CLI (interrupt disable) that set/clear IF.

Similar to MC68000, SP is switched upon an interrupt.

Also, IF is automatically cleared, so by default only one interrupt is accepted at a time.

Another view of mutual exclusion

```
process1 ( ... ) {
    ...
    enter_section (A) ;
    critical operations
    exit_section (A) ;
    ...
}
```

```
process2 ( ... ) {
    ...
    enter_section (A) ;
    critical operations
    exit_section (A) ;
    ...
}
```

```
enter_section (A)
```

```
process1 ( ... ) {
    ...
    enter_section (A) ;
    ...
}
```

```
process2 ( ... ) {
    ...
    exit_section (A) ;
    ...
}
```

This looks like a wait/wakeup mechanism, i.e., event passing.

Implementing mutual exclusion

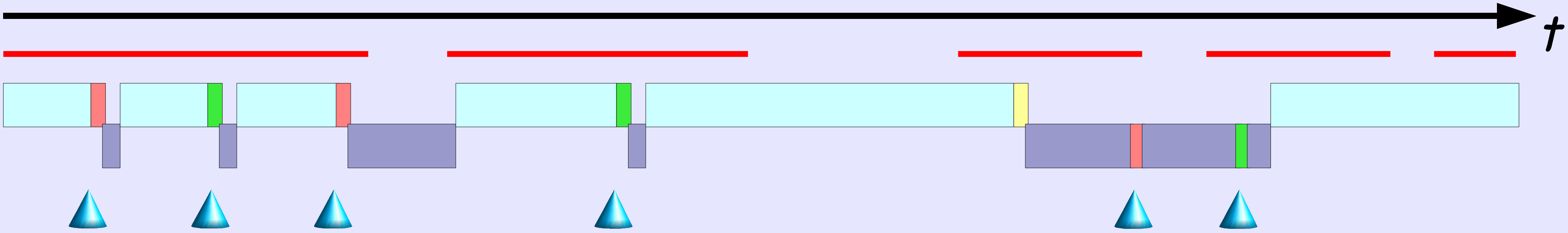
System calls (may be different names for the same thing):

enter_section (Id) exit_section (Id)

wait_signal (Id) trigger_signal (Id)

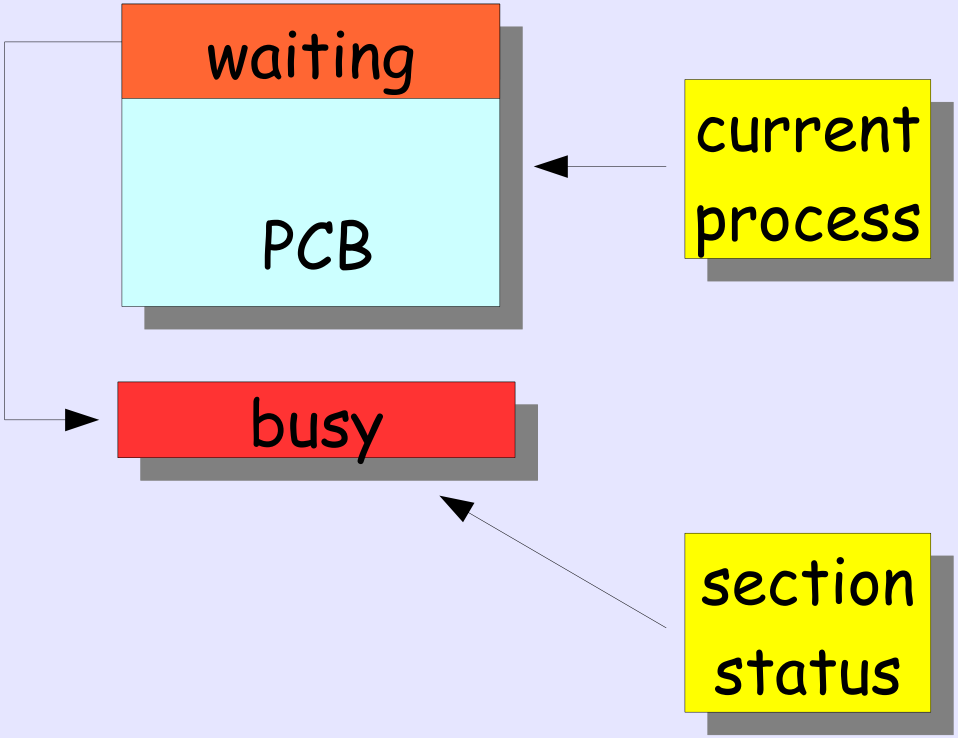
We can believe that the system should know how to do it right. After all, it manages the processes, so it can make sure that they do not preempt each other in certain circumstances.

Even if you do not see too many of such operations in normal programs, remember that they may be needed internally by some (otherwise unrelated) system calls.



In the Kernelland

- enter system call
- do not preempt
- check section status
- mark as waiting
- preempting OK
- reschedule



waiting

PCB

current process

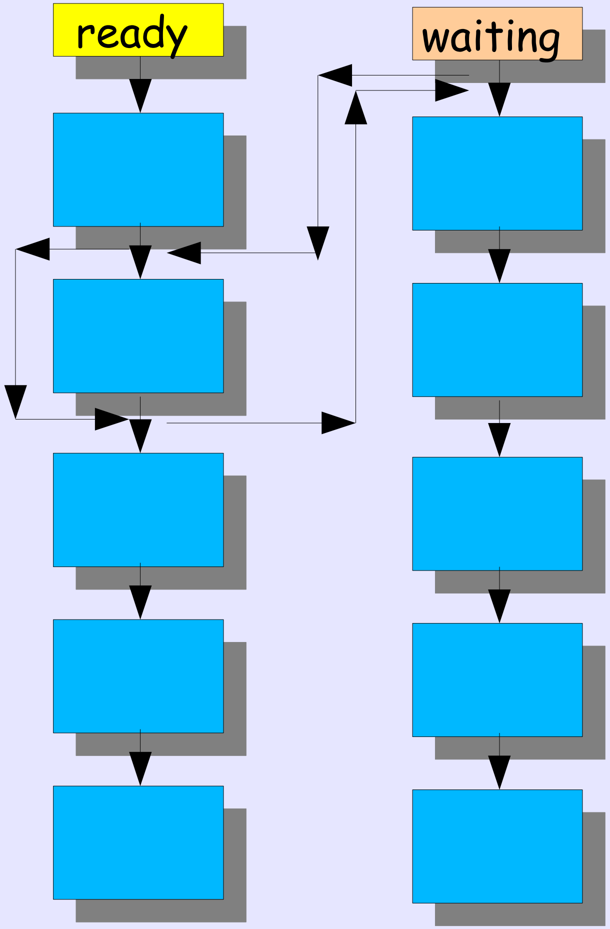
busy

section status

Implementing "do not preempt" is easy: the kernel may simply make sure that no rescheduling will occur for a while. To make their lives easier, many systems do not allow rescheduling to occur at all while a process is in the Kernelland.

The issue is a bit trickier, though

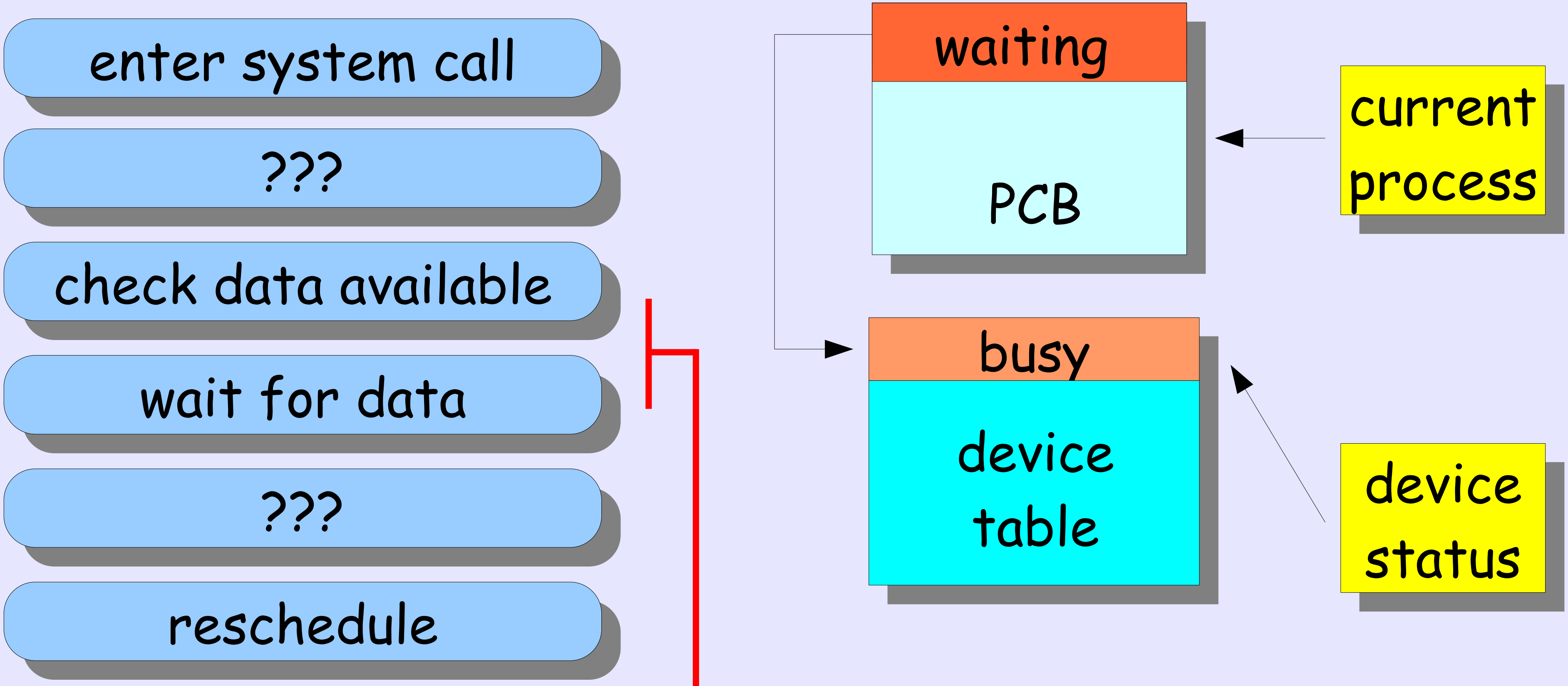
For example, the action of marking the process as waiting may involve moving its PCB across different queues:



There would be no problem, if the Kernelland were only accessible to system calls (we assume a single CPU). Unfortunately, interrupts can also get in there.

Suppose that the waiting queue is examined by interrupt service routines. Even if we somehow manage to avoid inconsistencies in the pointers, various unfriendly scenarios are possible.

Another look at the same thing



enter system call

???

check data available

wait for data

???

reschedule

waiting

PCB

current process

busy

device table

device status

Things may become tricky when the data shows up while we are doing this. The device status may be changed by an interrupt service routne.

Interrupt masking …

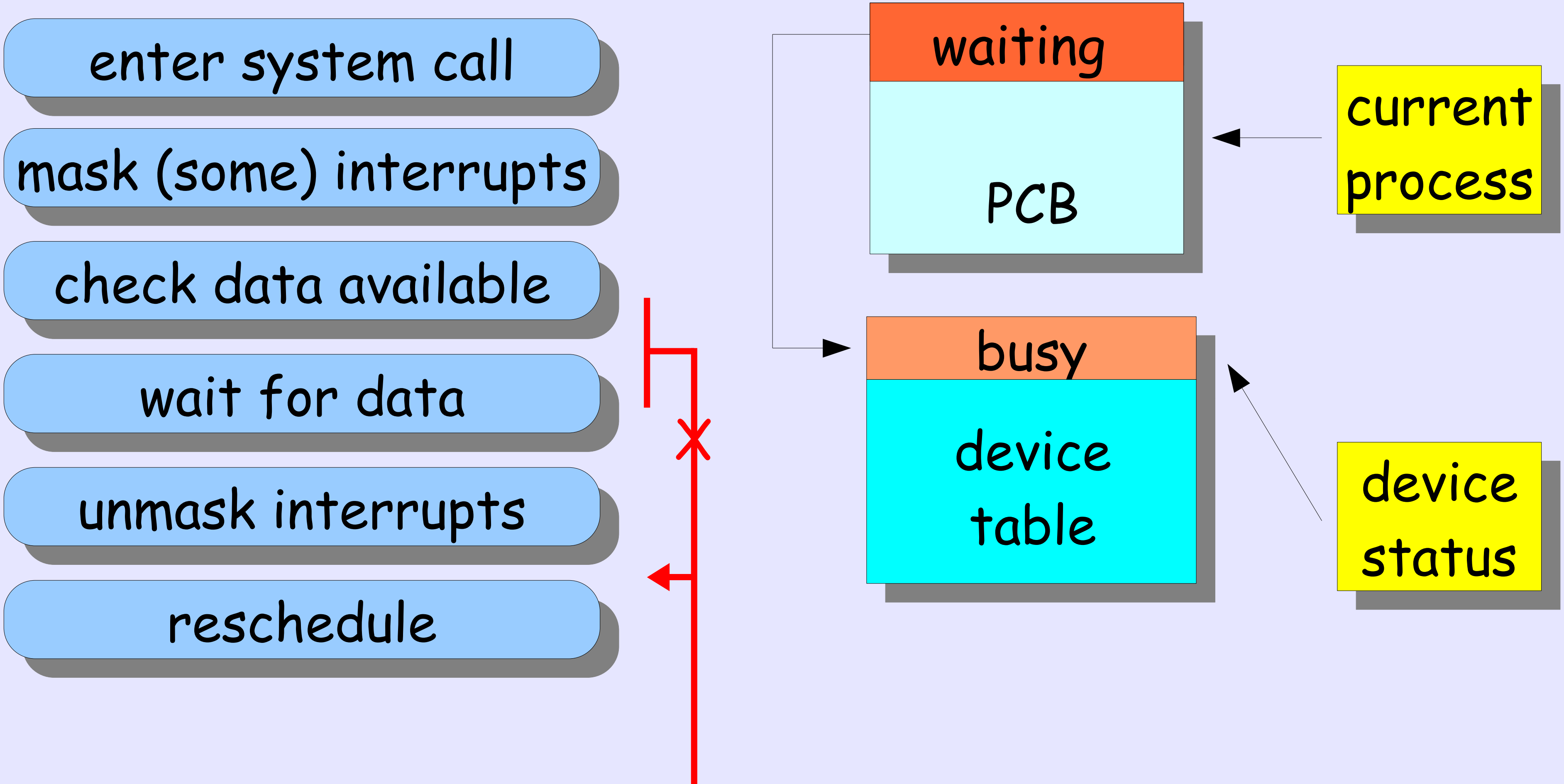
… is the most powerful built-in synchronization mechanism available to the kernel.

The CPU is executing some code (activity), be it a piece of user program or something in the Kernelland, including an interrupt service routine.

The only way for this activity to lose the CPU, that is beyond its control, is when an (external) interrupt occurs.

Remember that internal interrupts do not occur unless the activity generates them (purposely, we hope).

Once again



enter system call

mask (some) interrupts

check data available

wait for data

unmask interrupts

reschedule

waiting

PCB

current process

busy

device table

device status

Nothing wrong can happen now. The interrupt, should it occur here, will remain pending until it is safe for it to be accepted.

High-level tools

Semaphore is a structure consisting of a counter and two operations:

```
class Semaphore {
     int counter = 1;
     void P(void), V(void);
};
```

```
class CSemaphore {
     int counter = N ;
     void P(void), V(void);
};
```

S->P():

```
if (counter > 0)
     counter = counter - 1;
else
     wait until allowed to proceed;
```

S->V():

```
if (a process is waiting for S)
     let it proceed;
else
     counter = counter + 1;
```

This is a "counting" semaphore. No more than *N* processes can be within the critical section at the same time.

Monitors

A monitor is a structure that consists of (guarded) data, some functions (methods), and some events. The critical section is hidden in the implementation of the methods. Note: this is just a convenient high-level way of talking about things.

There are two types of methods: regular ones (nothing special here) and entryprocedures. The idea is that only one entryprocedure (of the same monitor) can be active at any given time.

Entryprocedures can wait for and trigger events. This is their way of coordinating access to the guarded data from multiple processes (activities).

Example

```
monitor Counter {
   int Total = 0;
   entryprocedure increment (int val) {
      Total = Total + val;
   }
   int value (void) {
      return Total;
   }
};
```

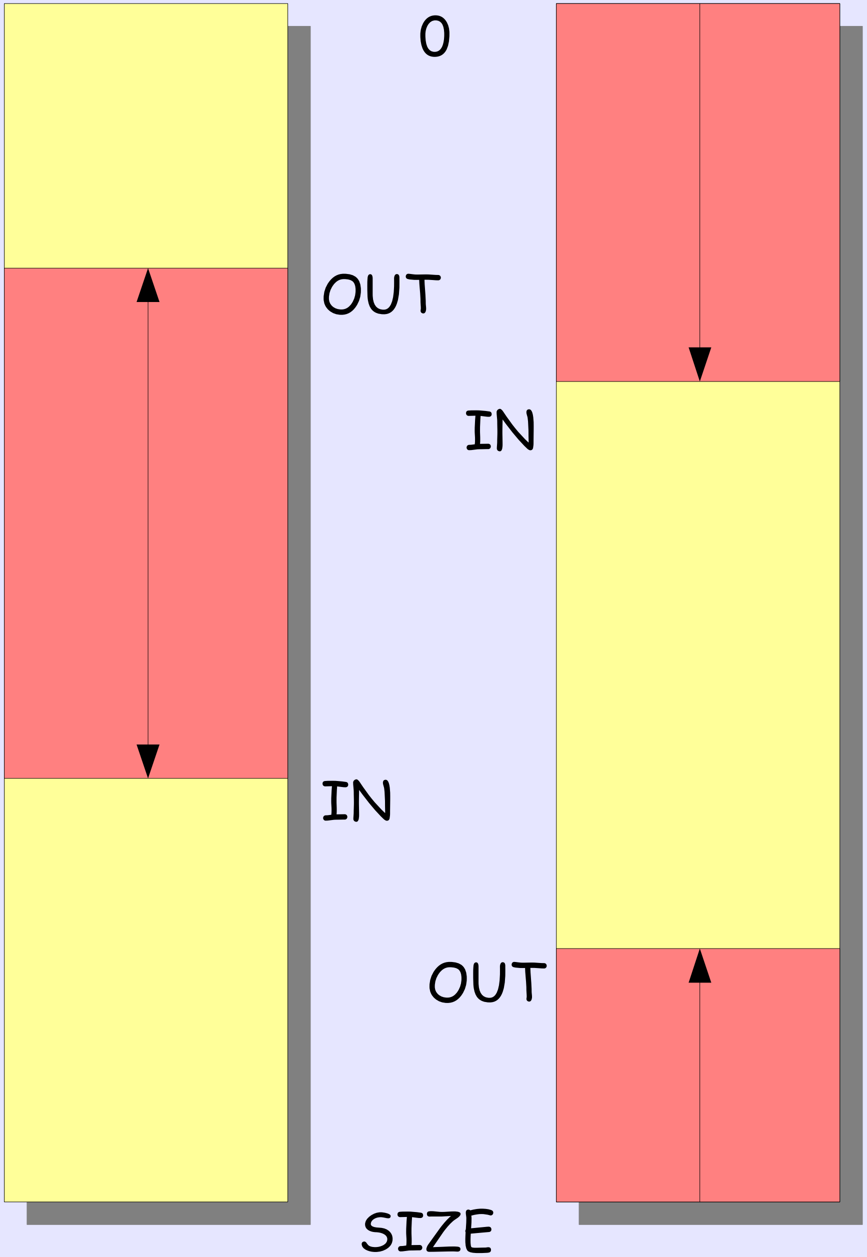
```
Counter CNT;
...
    start (reader_1);
    start (reader_2);
...
```

Note: of course, we do not solve anything just by using new keywords and writing a piece of pseudo-code in a non-existent language. The concept needs an implementation to be useful.

```
reader_1 (void) {
    int k;
    while (1) {
       fscanf (d1, “%d”, &k);
       CNT.increment (k);
    }
}
```

```
reader_2 (void) {
    int k;
    while (1) {
       fscanf (d2, “%d”, &k);
       CNT.increment (k);
    }
}
```

Another example: circular buffer



```
monitor CBuffer {
    int IN = 0, OUT = 0;
    ITEM Buf[SIZE];
    event Room, NonEmpty;
    entryprocedure put (ITEM it) {
       if ((IN+1)%SIZE == OUT)
          Room.wait ();
       Buf[IN] = it;
       IN = (IN+1)%SIZE;
       NonEmpty.trigger ();
    };
    ITEM entryprocedure get () {
       if (OUT == IN)
          NonEmpty.wait ();
       res = Buf[OUT];
       OUT = (OUT+1)%SIZE;
       Room.trigger ();
       return res;
    }
};
```

Event passing

We have already seen one event passing mechanism. Any direct mutual exclusion mechanism will do, e.g.,

```
producer(...) {
  ITEM tosend;
  ...
  while (1) {
    ...
    tosend = produce();
    Empty.P();
    buffer = tosend;
    Full.V();
    ...
  }
};
```

```
consumer(...) {
  ITEM next;
  ...
  while (1) {
    ...
    Full.P();
    next = buffer;
    Empty.V();
    consume(next);
    ...
  }
};
```

```
...
Semaphore Empty, Full;
...
Full.P();
start (producer);
start (consumer);
...
```

Is it really needed?

```
producer(...) {
  ITEM tosend;
  ...
  while (1) {
    ...
    tosend = produce();
    while (Full);
    buffer = tosend;
    Full = true;
    ...
  }
};
```

```
...
bool Full;
...
Full = false;
start (producer);
start (consumer);
...
```

```
consumer(...) {
  ITEM next;
  ...
  while (1) {
    ...
    while (!Full);
    next = buffer;
    Full = false;
    consume(next);
    ...
  }
};
```

Formally, we don't need mutual exclusion to handle this scenario (remember Attempt 2?). The advantage of semaphores (as an event passing tool) is in the avoidance of busy waiting.

Signal passing semantics

One more example: semaphore implemented as a monitor:

```
monitor Semaphore {
  bool busy = false;
  event Free;
  entryprocedure P() {
    if (busy)
      Free.wait();
    busy = true;
  };
  entryprocedure V() {
    busy = false;
    Free.trigger();
  };
};
```

This demonstrates that different mutual-exclusion tools are formally interchangeable. Indeed, a very basic tool, e.g, of the dont_preempt flavor, can be used as a basis for implementing everything else. A blocking mechanism that avoids busy waiting is another useful feature.

The signals here have a specific semantics, which we will call DNQ, for Do Not Queue. A signal is ignored if nobody is waiting for it.

Other semantics?

The semantics offered by semaphores (like in the producer-consumer scenario, is QOS, i.e., Queue One Signal.

Queue Multiple Signals (QMS)? Not very useful (although not completely useless). Counting semaphores implement something like that. Certainly, N must be bounded to make sense.

This one is often handy (and used a lot in the Kernel): DNQ-CA, which stands for Do Not Queue, Check Again.
Its advantage is that there is no need for any specific place to store the signal, i.e., the signal object need not be represented (like a semaphore, for example).

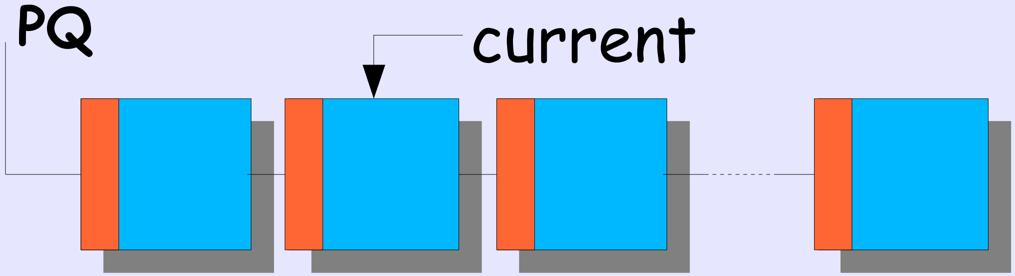
Imagine a Kernelland mechanism ...

... for awaiting and triggering events. It may look like this:

```
void waitforevent(int eid){
  current->Status = eid;
}
```

This is called when the system returns to the Userland, and "current" may have to (or must) be changed:

```
void schedule(){
  PCB *p;
  for (p=PQ, 1; p=p->Next)
    if (p->Status == 0){
      current = p;
      userland ();
    }
}
```



Status==0	READY
Status!=0	BLOCKED, waiting for eid.

```
void trigger(int eid){
  PCB *p;
  for (p=PQ, p!=NULL; p=p->Next)
    if (p->Status == eid)
      p->Status = 0;
}
```

How to use this for mutual exclusion

```
void access(resource *r){
  while (1){
    mask();
    if (!r->busy) {
      r->busy = true;
      unmask ();
      return;
    }
    waitforevent ((int)r);
    unmask ();
    schedule ();
  }
}
```

```
void free(resource *r){
  r->busy = false;
  trigger ((int)r);
}
```

We can say that access effectively executes as an entryprocedure of the monitor guarding the resource.

What is the semantics of signals (as per trigger)? All processes waiting for a given signal are awakened and they have to "check it again". This is what I meant by DNQ-CA.

I/O wakeup

Note: both access and free can be used by processes (in the Kernelland). External interrupts cannot use access, although they can use free. **This makes sense.**

```
void read_data(device *d, userbuf *b){
  while (1){
    mask();
    if (d->data_available){
      move_to_userland (r->data, b);
      d->data_available = false;
      unmask();
      return;
    }
    waitforevent ((int)d);
    unmask ();
    schedule ();
  }
}
```

```
interrupt endIO(){
  device d;
  d = which_device();
  d->data_available = true;
  trigger ((int)d);
 ...
  schedule ();
}
```

A few words about multiple CPUs

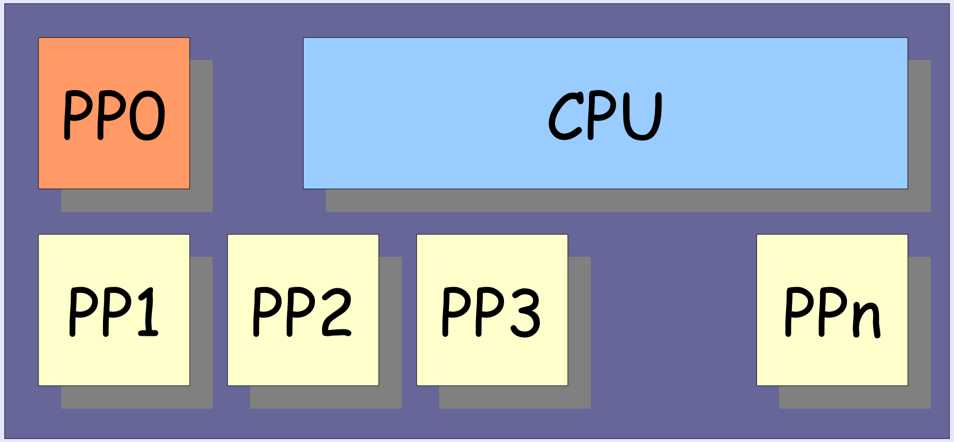
Symmetric systems:

All CPUs have identical rights. They grab processes for execution from the same global pool.

Non-symmetric systems:

Different processors have different preassigned roles.

Example: Scope/NOS for CDC 6000 series:



PP0 (peripheral processor 0) runs the monitor (recall the name), which coordinates the activities of all the other processors.

SMP: Symmetric Multi-Processors

Bad news: interrupt masking is no longer the ultimate remedy. Different CPUs can receive interrupts independently.

Good news: some useful hardware tools can help.

More good news: busy waiting is no longer unconditionally harmful.

Test and Set

```
T&S     F,V,R
```

"Atomically: R ← F; F ← V"

Compare to this

```
while 1 {
    T&S Lock,1,Prev;
    if (Prev == 0) break;
}
```

→

```
bool section_busy (int *g) {
    int result;
    dont_preempt ();
    result = *g;
    *g = true;
    preempting_ok ();
    return result;
};
```

A digression

In what circumstances may busy waiting make sense on a single-CPU system?

The answer: if the event is to be delivered by a device, and the expected waiting time is shorter than the combined overhead of putting the process to sleep and waking it up.

Such cases are rather rare, although when we get to the multiple CPU scenario and replace "a device" by "another processor", we will get a more general and very true statement.

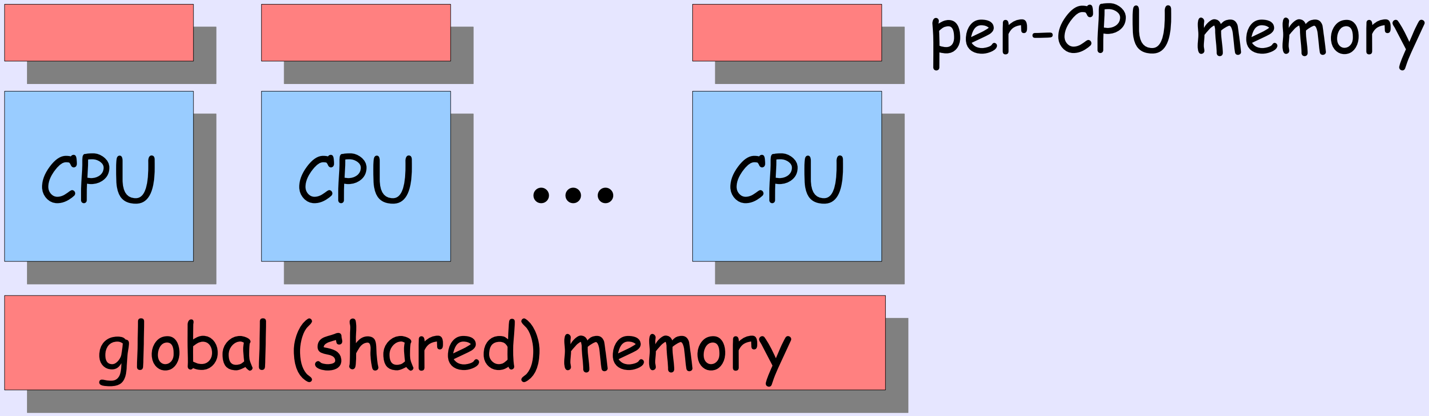
The moral: short critical sections on multiple-CPU systems can be sensibly resolved with busy waiting (spin locks). In fact, that may be the most economical way of handling them.

Adapting a single-CPU OS to SMP

Here is a crude but simple and effective way:

Make sure that only one CPU at a time can be in the Kernelland (e.g., a spin lock).

Account for multiple current processes (the current pointer must belong to per-CPU data).



The Kernel uses effectively only one CPU (at a time), but multiple processes can run in parallel. This may be enough to enjoy the performance gains.

Lockless programming

Consider a conditional hardware variant of T&S (called check and swap) that is atomic and works like this:

```
bool cas (*int loc, int old, int new) {
  if (*loc == old) {
    *loc = new;
    return true;
  } else
    return false;
}
```

... and recall our old problem of calculating

```
sum = sum + k;
```

```
while (1) {
  old = sum;
  new = sum + k;
  if (cas (*sum, old, new))
    break;
}
```

Lockless programming is about synchronizing processes without using locks. It is tricky, so solutions focus on ways to access standard data structures (e.g., queues) used in IPC.

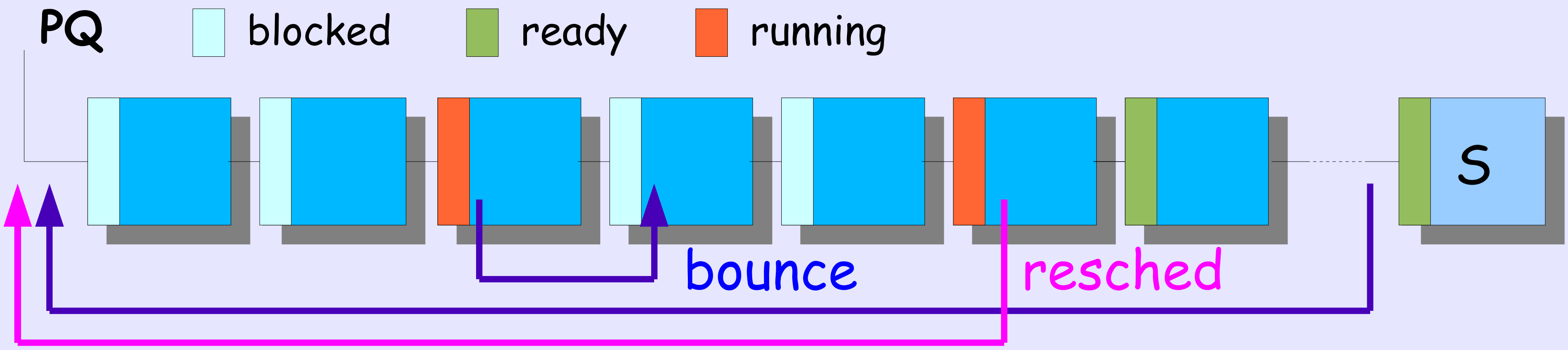
Another example

Basic synchronization tools on some SMP system (of a sentimental value to your instructor): OOPS for MERA-400.

```
lock (int *v) {
   while (T&S (v))
      bounce ();
}
```

```
unlock (int *v) {
    *v = 0;
     resched ();
}
```

A trivial observation: busy waiting does not hurt too much if the spinning CPU couldn't possibly do anything useful.



Examining the process queue

The process queue is a critical data structure with two types of entries:

Soft entry:

Multiple copies of the CPU scheduler are allowed to soft-enter the process queue looking for processes to run. The only problem is to make sure that two CPUs do not take the same process for execution.

Hard entry:

The queue is being reorganized. Absolute mutual exclusion, no soft entries allowed.

Picking up processes for execution

Status = 0000000000000000 READY

Status = 0000000000000001 RUNNING

Status = xxxxxxxxxxxxxxx 1 BLOCKED

event identifier

```
int T&S(int *s, int b){
  int k = *s;
  *s = *s | b;
  return k;
}
```

remember this?

```
void schedule(){
  PCB *p;
  for (p=PQ, 1; p=p->Next)
    if (p->Status == 0){
      current = p;
      userland ();
    }
}
```

```
void schedule(){
  PCB *p;
  for (p=PQ, 1; p=p->Next)
    if (T&S(p->Status,1) == 0){
      current = p;
      userland ();
    }
}
```

Process queue locks

```
int hard, soft, soft_lock;
```

```
softentry () {
  while (T&S(hard,1));
  while (T&S(soft_lock,1));
  soft++;
  soft_lock = 0;
  hard = 0;
}
```

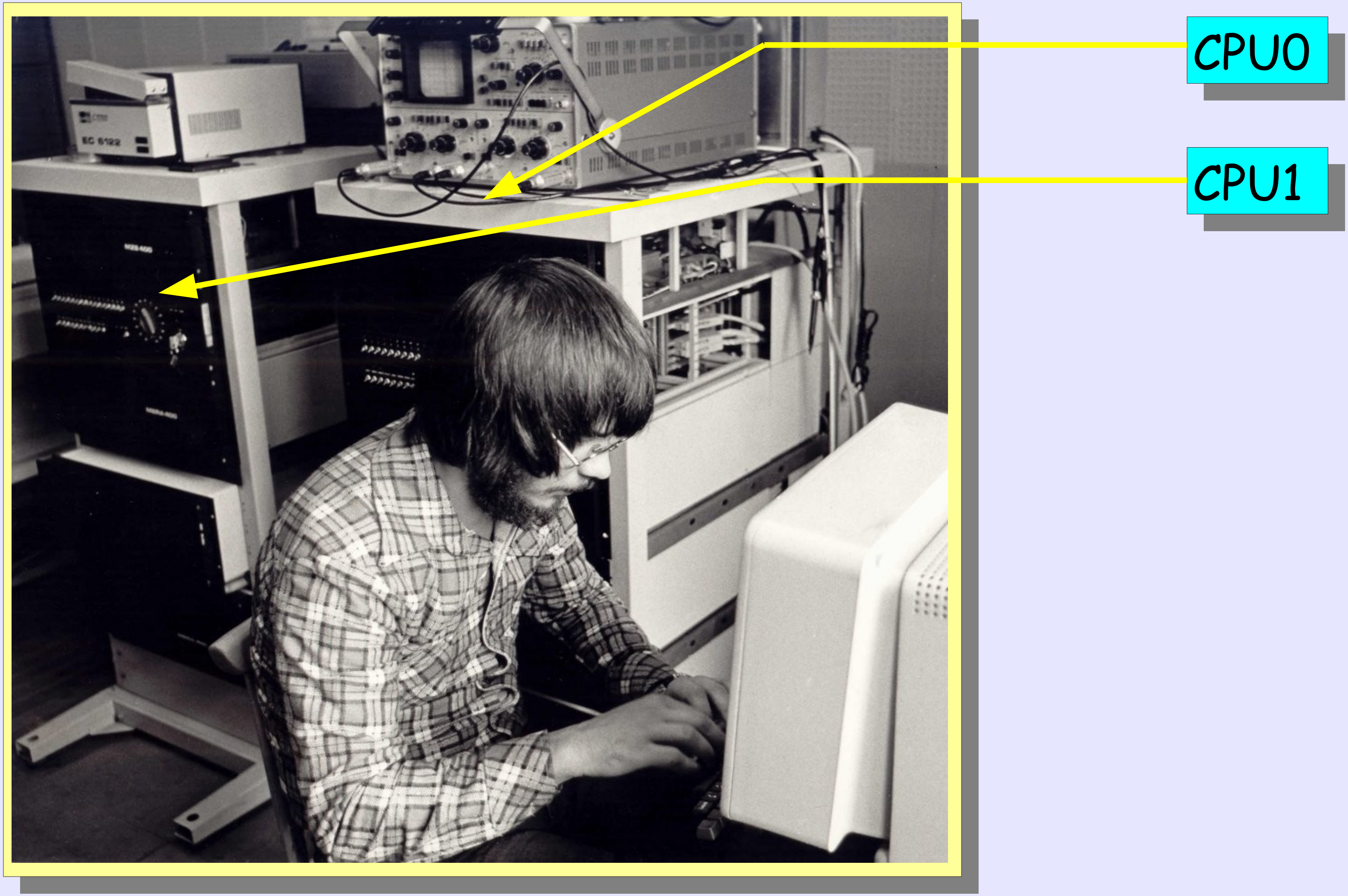
```
hardentry () {
  while (T&S(hard,1));
  while (soft);
}
```

```
softexit () {
  while (T&S(soft_lock,1));
  soft--;
  soft_lock = 0;
}
```

```
hardexit () {
  hard = 0;
}
```

soft_lock is not needed if **++** and **--** are atomic.

The machine (1983)



Message passing

Shared memory + locks is not the only possible IPC paradigm. Message passing has the following advantages:

Arguably, message passing is conceptually simpler and easier to handle. Shared memory brings about its known bag of tricky synchronization issues.

Natural block/wakeup mechanism: a blocking wait for message reception looks like a simple read operation; dispatching a message looks like a write.

Applicable in those circumstances when the communicating processes are physically unable to share memory, e.g., across the Internet.

Consequently, it makes sense to use it for local communication: it will make programs easily portable to networked systems.

Paradigms

Message structure:

- Datagrams: independent packets of data (of a limited length)
- Streams (pipes): strings of bytes (no limit on length)

Delivery/pickup:

send/receive operations

- Messages addressed directly to processes (direct mechanism)
- Mailboxes/special objects where messages are deposited

Reliability:

- Buffering? What happens when the buffer is filled
- Can a message get lost (e.g., in a loosely distributed system)?
- Priority data: is it possible to bypass the buffer?

UNIX message passing tools

Direct mechanism: SIGNALS

```
kill (int pid, int signal);
```

... sends a simple message directly to the indicated process. There is no receive operation: the process receives the signal asynchronously.

This mechanism is also used by the Kernel to send processes "messages" about their misbehavior. A program that generates an error (say, illegal memory reference or division by zero) receives a pertinent signal.

A program can prepare itself for a signal reception. An unprepared process receiving a signal is terminated (aborted).

Signal reception

signal (int signal, SIGARG func);

The indicated function is called (asynchronously) when the signal is received. This is similar to an interrupt. Generally, reliable reception of signals from other processes is difficult.

Signals are stored as binary flags in a special field in the PCB and are checked when the process returns to the Userland. Interrupting a blocked system call by a signal may be tricky.

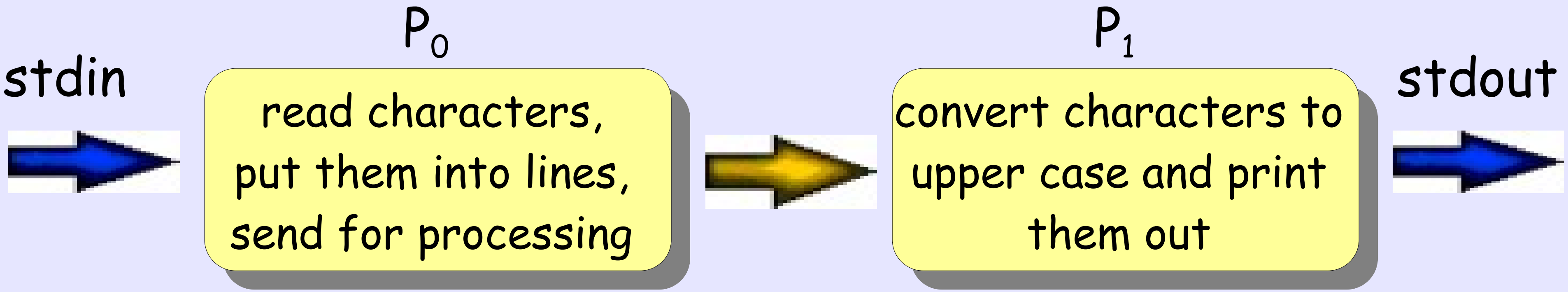
Some signals cannot be caught (SIGSTOP, SIGKILL); some signals are ignored by default (SIGCHLD).

signal (int signal, SIG_IGN);

says that the signal should be ignored (if possible).

Pipes: an example

Here is a two-process program that reads characters from the standard input, converts them to upper case, and writes them to standard output. As a distributed processing problem, it is blatantly stupid, but we want to see the communication structure, not the application's complexity.



stdin

P_0

read characters, put them into lines, send for processing

P_1

convert characters to upper case and print them out

stdout

Example, cntd.

```
#include <stdio.h>
#include <unistd.h>
#include <ctype.h>

int pfd [2];
```

```
main () {

    if (pipe (pfd) < 0) {
       perror ("Cannot create pipe");
       exit (99);
    }

    if (fork ())
       do_process1 ();
    else
       do_process0 ();
}
```

Example, cntd.

```
#define MAX_LINE_LENGTH 256
```

```
do_process0 () {
  int fill, c;
  char line [MAX_LINE_LENGTH];
  close (pfd [0]);
  fill = 0;
  while ((c = getchar())!= EOF){
    if (c == '\n') {
      line [fill++] = c;
      write (pfd[1],&line,fill);
      fill = 0;
    } else if (fill < MAX_LINE_LENGTH-1)
      line [fill++] = c;
  }
  if (fill) {
    line [fill++] = '\n';
    write (pfd [1], &line, fill);
  }
  close (pfd [1]);
}
```

```
do_process1 () {
  char c;
  close(pfd [1]);
  while(read(pfd[0],&c,1)>0){
    if(islower(c))
      c = toupper (c);
    putchar(c);
  }
}
```

Extensions of pipes

FIFOs:

Named pipes: look like open-ended files, and can be used by unrelated processes.

Sockets:

Local: kind of like named pipes, but offering a few more options regarding the message structure and operations.

Network: pipes that work across machines connected to networks, including the Internet.

You use network sockets whenever you do web browsing, e-mail, and all that stuff.

Details in the labs. Also, see assignment 2.

Deadlock

A process is said to be in the state of deadlock when it is waiting for something that will provably never happen.

Trivial examples:

```
...                                   ...
 wait (nonexistentevent);              while (1);
...                                   ...
```

We are not interested in such cases: they result from bugs in sequential programming. The interesting cases of deadlock involve activities of multiple processes. Each process by itself is OK. The processes get deadlocked because of some “collaboration” problems.

Another example

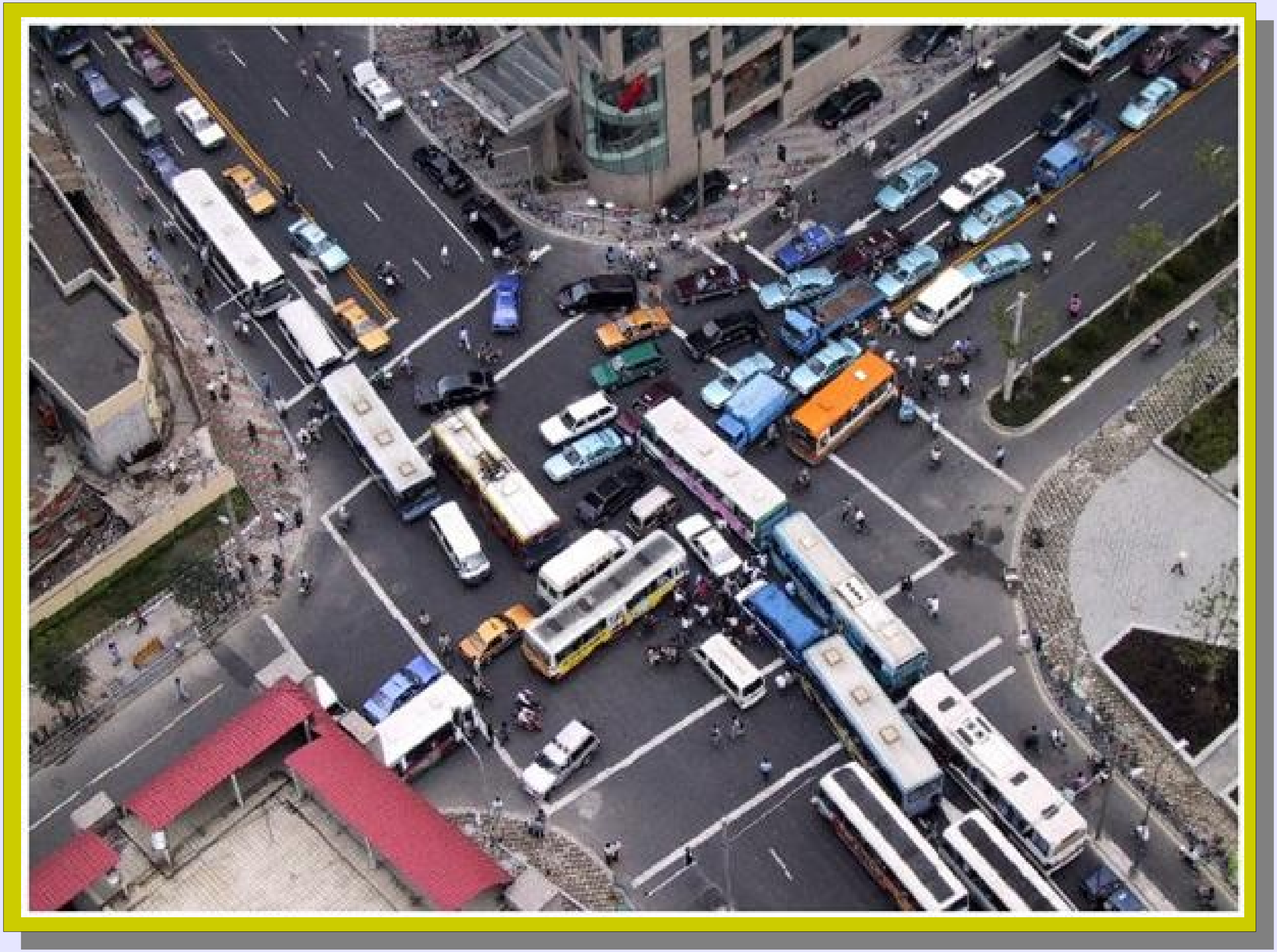
Each of the two processes has to grab two critical sections to accomplish something. This often happens, e.g., in database access:

```
process1 (...) {
  ...
  Sem1->P();
  ...
  Sem2->P();
    do critical operations
  Sem2->V();
  Sem1->V();
  ...
}
```

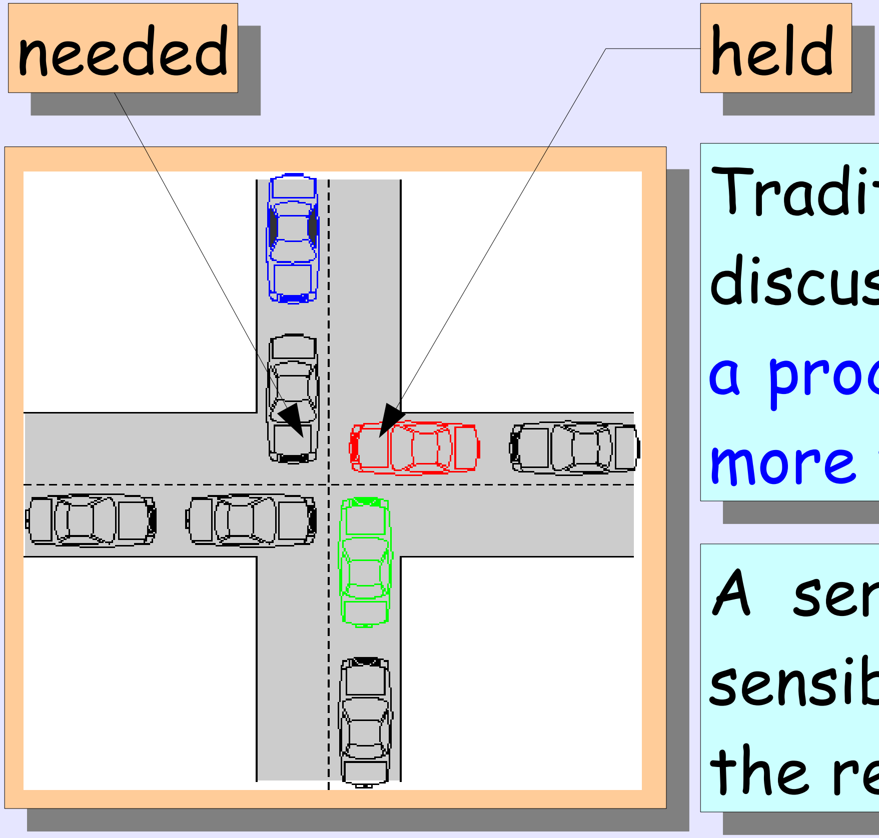
```
process2 (...) {
  ...
  Sem2->P();
  ...
  Sem1->P();
    do critical operations
  Sem1->V();
  Sem2->V();
  ...
}
```

The problem will not occur, if the processes grab the locks in the same order.

Yet another example



Resource allocation



Traditionally, the deadlock problem is discussed in terms of resource allocation: a process holds some resources and needs more to complete.

A serious problem occurs if there is no sensible (non-destructive) way to remove the resources from the process.

By stretching our idea of a resource, we can define all (non-trivial) kinds of deadlock as resource allocation problems. Remember: critical sections are resources.

How to avoid deadlock?

Ich will schreiben!
Geben Sie mir Papier!
Geben Sie mir den Stift!

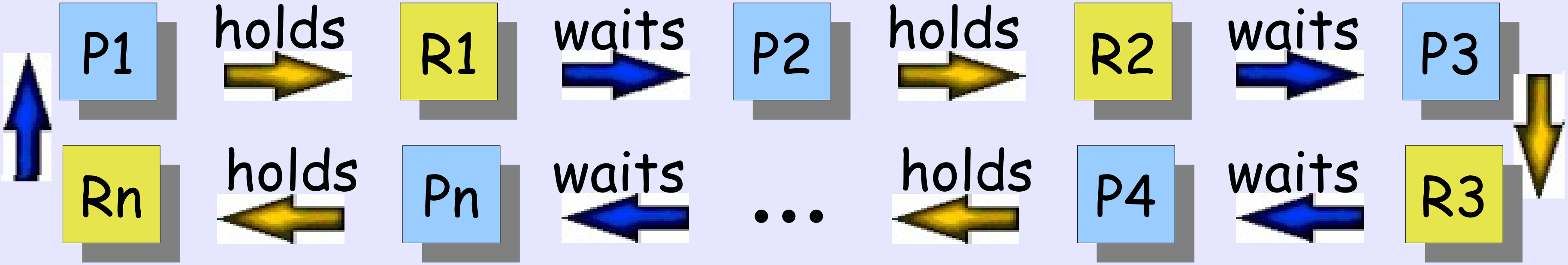
provide enough resources for everybody, such that they will never get into problems

teach the "processes" to be courteous and yield

teach them how to fight if other parties are not courteous

impose a protocol (restrictions) on the resource allocation scheme, e.g., traffic signs

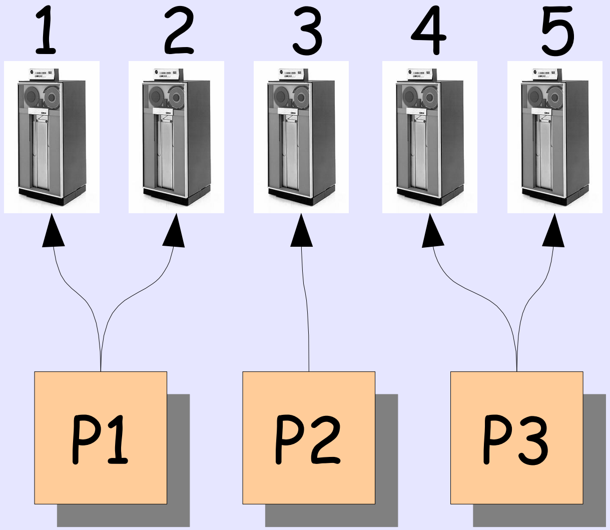
Deadlocks are like vicious circles:



P1 holds R1 waits P2 holds R2 waits P3

Rn holds Pn waits ... holds P4 waits R3

... but the circles need not be explicit. For example, assume that a system has 5 tape drives and there are three processes:



Now suppose that each of the three processes needs one more drive to complete. We may have a deadlock. Note that a process need not say which particular drive it needs. In most cases it doesn't care.

Resource types

Preemptible:

Can be taken from the process and later returned without affecting the process's integrity.

Non-preemptible:

Once a process acquires the resource, it must remain allocated to the process until explicitly released.

Examples:

CPUs are generally(?) preemptible. A single process never wants more than one, but exotic distributed applications may require a number of CPUs to complete.

Tape drives, floppy disks, USB storage keys, CD/RW drives, are non-preemptible resources.

Memory may be preemptible or not, depending on the system.

Necessary conditions for a deadlock

Mutual Exclusion

Resources are not shared. Processes holding them require mutual exclusion.

Hold+Wait

Processes are allowed to hold resources while waiting for additional resources.

No Preemption

Resources cannot be removed from processes and later returned unless the process releases them explicitly.

Circle

One can identify a circular chain of processes and resources, with each process holding a resource needed by the previous process and requesting a resource held by the next process.

Denying the necessary conditions

Processes cannot request resources incrementally. You must receive everything you are ever going to need at the beginning.

Denies Hold+Wait.

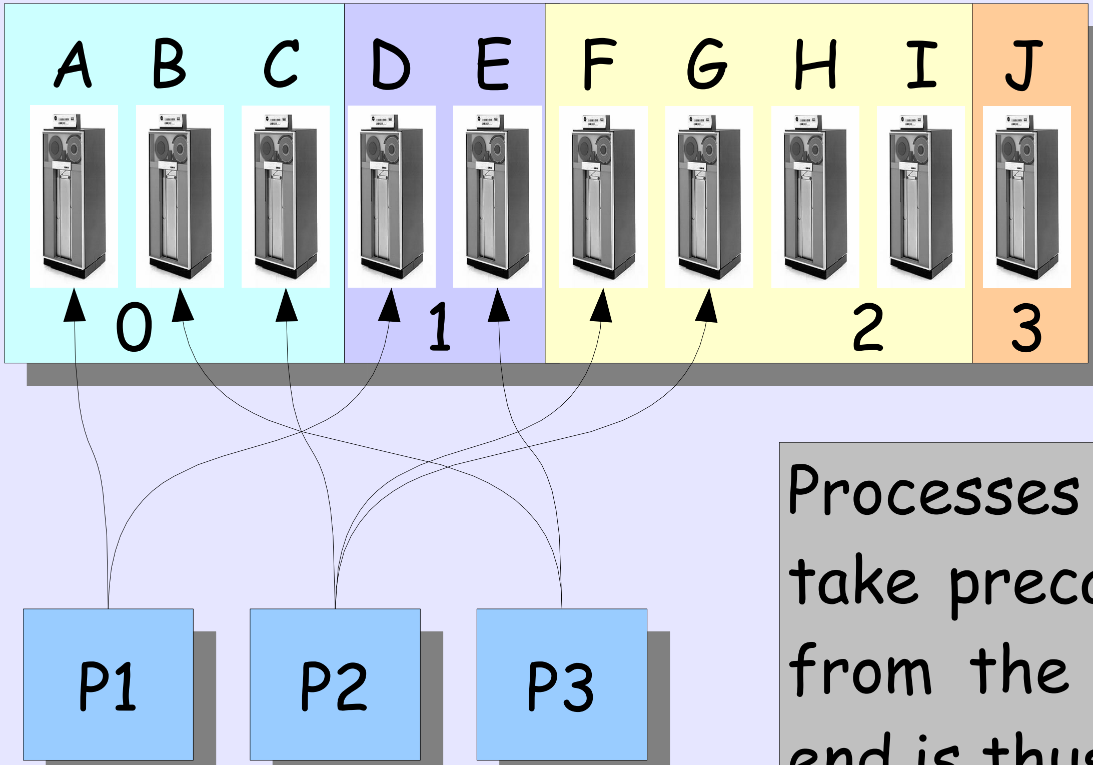
If your request cannot be fulfilled immediately, you have to release everything, and re-request everything back together with the new item.

Denies Hold+Wait + No Preemption. A bit more flexible. This approach is popular for acquiring multiple locks, e.g., in databases.

You must request resources in a certain order. This order is followed by everybody.

Denies Circles.

Ordering resources



A process holding a resource from a given class can only request resources from a higher class.

Processes that need a lot of drives must take precaution not to start requesting from the high end. Note that the low end is thus going to be popular.

This is OK because those processes that need little can start from the high end and thus hope for a shorter waiting time. This exemplifies one of the various policing strategies that used to be popular in mainframe batch systems.

Banker's algorithm

Suppose that there is only one resource type with M units. These are the rules:

Each process (job) declares in advance the maximum number of units that it will be needing at any given time. This declaration cannot be more than M.

The system aborts a process that violates its declaration.

A process may request an arbitrary number of units at once, as long as the request does not violate its declaration. The process may have to wait, but the system guarantees that as long as processes eventually complete (and thus release what they've been holding), all processes will happily execute to successful completion.

Safe vs. unsafe state

The system makes sure to keep the resource allocation in a safe state. It grants a request only if the resulting state is safe, meaning that there is a way to allocate whatever is left to the processes as to execute them all to completion.

Example:

?

Process	Max	Current
P0	10	5
	4	
P2	9	3

M=12

Left

4

Note that unsafe state does not imply deadlock! The process's Max declaration may have applied to some moment in the past, and the process may not request that many items any more.

The general case

`n`	the number of processes (jobs)
`m`	the number of resource types
`Av[j]`	the number of resource units of type `j` still available
`Max[i,j]`	the maximum demand of process `i` for resource `j`
`Al[i,j]`	current allocation of resource `j` to process `i`
`N[i,j]`	remaining need of process `i` for resource `j`

We have:

$$N[i,j] = Max[i,j] - Al[i,j]$$

$$Av[j] = T[j] - \sum_i Al[i,j]$$

where `T[j]` is the total number of units of resource `j` available in the idle system.

The allocation procedure

```
void allocate (int j, int k) {
  if (k > N[current,j])
    abort (current);
  while (1) {                   LOCK
    if (k <= Av[j]) {
      Av[j] -= k;
      Al[current,j] += k;
      N[current,j] -= k;
      if (safe ()) {
        grant (j, k);           UNLOCK
        return;
      }
      Av[j] += k;
      Al[current,j] -= k;
      N[current,j] += k;
    }
    wait (RESOURCE_RELEASE);    UNLOCK
    schedule ();
  }
};
```

Safety check

```
bool safe () {
  int W[m], i, j; bool F[n], alldone;
  for (j=0; j<m; j++) W[j]=Av[j];
  for (j=0; i<n; j++) F[i]=false;
Restart:
  alldone = true;
  for (i=0; i<n; i++) {
    if (!F[i]){
      for (j=0; j<m; j++)
        if (N[i,j] > W[j]) break;
      if (j == m) {
        for (j=0; j<m; j++) W[j] += Al[i,j];
        F[i] = true;
        goto Restart;
      }
      alldone = false;
    }
  }
  return alldone;
}
```

One more example

Pcs		A	B	C	A	B	C	A	B	C
		allocation			maximum			need		
0		2	0	0	6	4	4	4	4	4
1	?	1	1	1	3	4	5	2	3	4
2		5	4	3	8	7	5	3	3	2
3	●	2	2	1	2	3	2	0	1	1

A	B	C
left		
2	3	2

Is this one safe?

Note that in some circumstances the safety assessment may be conservative and unnecessarily restrictive. For example, process 2 (which appears to be a resource hog), may not need anything any more. It may terminate in the very next while returning all its resources to the free pool. There is only so much we can do with the static nature of the "maximum" declarations.

Deadlock detection

You see a stale configuration of processing, each of them waiting for some resource, and each of them holding some resource. You ask yourself "are they deadlocked or just temporarily stuck on something"?

`n`	the number of processes (that appear stuck)
`m`	the number of resource types
`Av[j]`	the number of resource units of type `j` still available
`Al[i,j]`	current allocation of resource `j` to process `i`
`Rq[i,j]`	the number of units of resource `j` requested by `i`

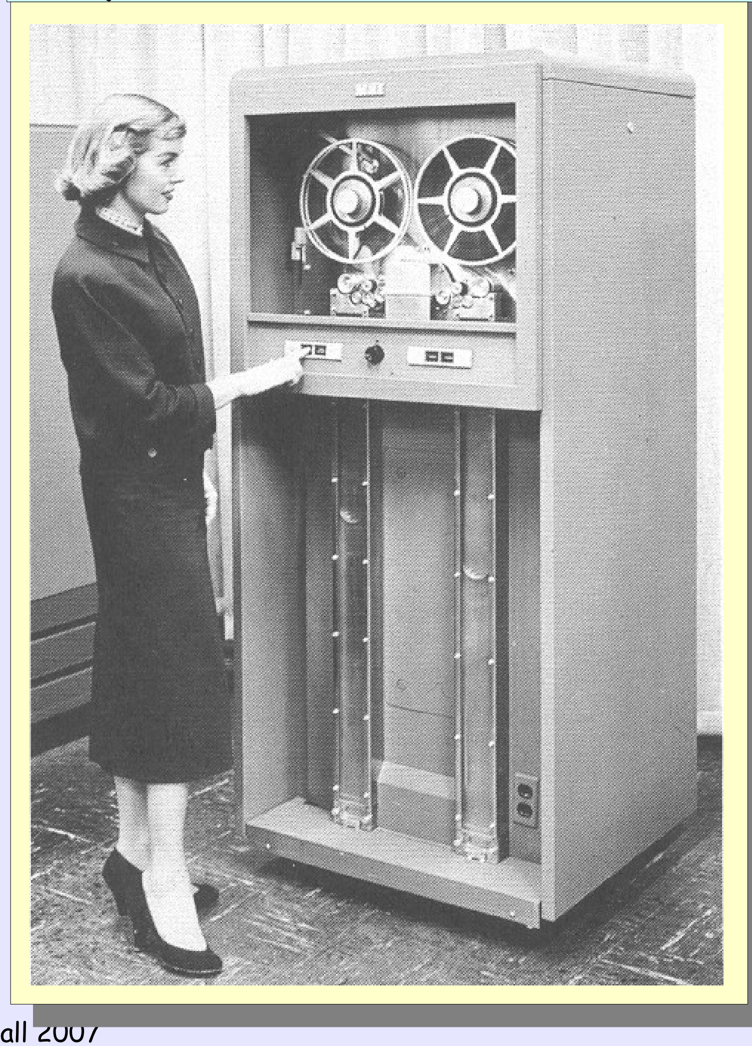
Note: this is only superficially similar to the Banker's algorithm. We do not require the processes to declare anything in advance. Here we simply want to check if the stuck processes can make progress at all.

Deadlock detection algorithm

```
int deadlock () {
  int W[m], i, j; bool F[n], alldone;
  for (j=0; j<m; j++) W[j]=Av[j];
  for (i=0; i<n; i++) F[i]=false;
Restart:
  alldone = true;
  for (i=0; i<n; i++) {
    if (!F[i]) {
      for (j=0; j<m; j++)
        if (Rq[i,j] > W[j]) break;
      if (j == m) {
        for (j=0; j<m; j++) W[j] += Al[i,j];
        F[i] = true;
        goto Restart;
      }
      alldone = false;
    }
  }
  return !alldone;
}
```

Is deadlock prevention important?

These days there are practically no physical resources that can be involved in deadlock scenarios (the abominable tape drives can only be seen in old movies).



The popular realistic cases of deadlock result practically exclusively from synchronization, i.e., acquiring multiple locks in the wrong order.

Remember: always acquire multiple locks in the same order in all processes (or threads).

Sometimes you don't know whether the other processes follow your protocol. What then?

One more useful operation

`S->P();`
`S->V();`
+
`S->T();`

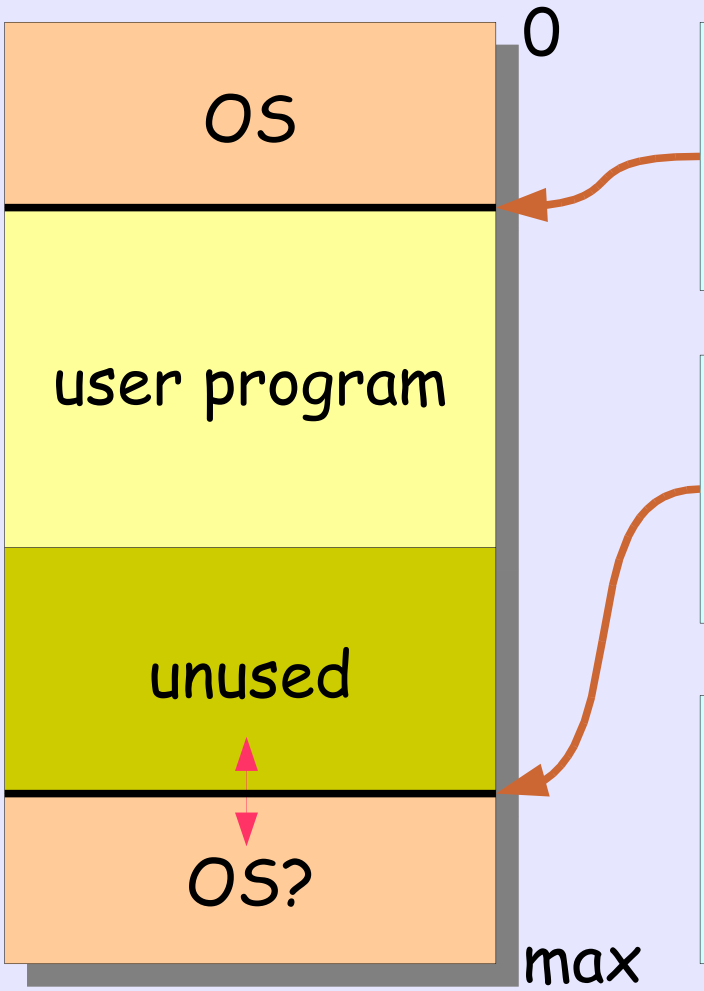
Try the semaphore, return immediately (true or false).

```
void getTwoLocks (semaphore *sem1, semaphore *sem2) {

  while (1) {
    sem1->P();
    if (sem2->T())
      return;
    sem1->V();
    sem2->P();
    if (sem1->T())
      return;
    sem2->V();
  }
}
```

Memory allocation (management)

(a historic perspective)

In a single-programming system, the issue is rather simple, e.g., we may have something like this:



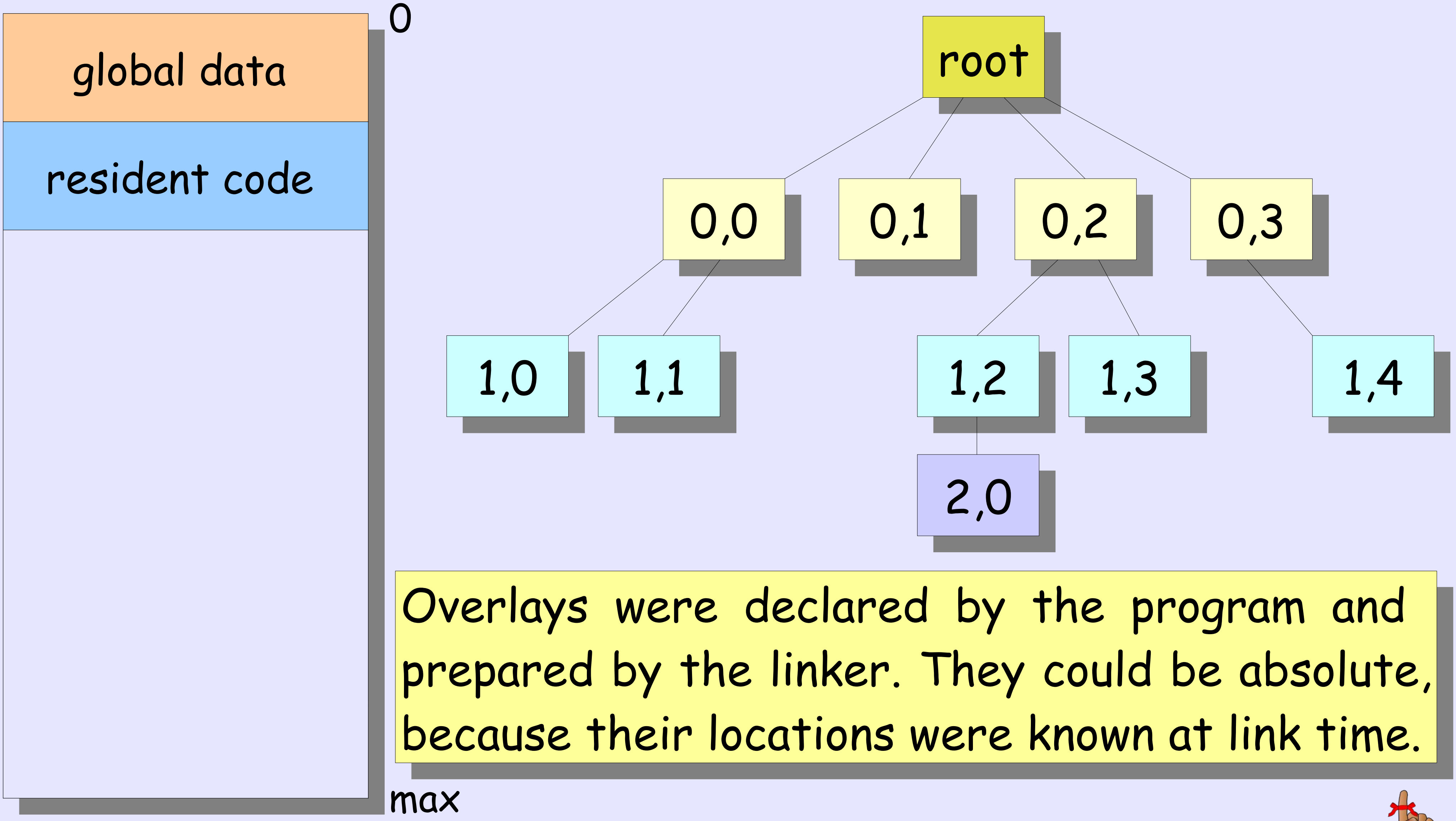
This boundary is fixed: the user program is always loaded into the same memory location (so it can be absolute).

This boundary may vary: the system may keep here its heap, i.e., the dynamic data structures.

Sometimes the system area is protected from the user program, but it isn't really that important.

Overlays

An early idea aimed at running large programs in small memory:



Overlays were declared by the program and prepared by the linker. They could be absolute, because their locations were known at link time.

Memory layouts of early programs...

... were rather simple, e.g., it wasn't obvious that:

- stacks are generally useful
- code and data should be separated

The prevailing programming languages were:

Fortran	for efficiency (number crunching)
Cobol	for convenience (data processing)

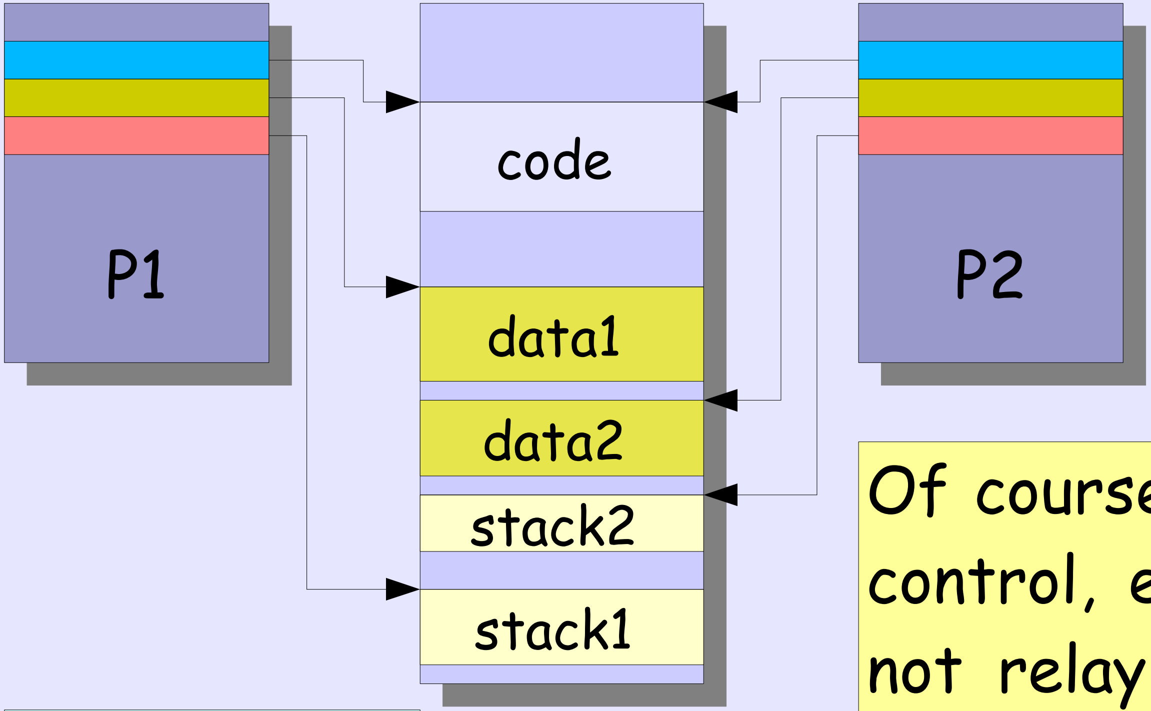
They are (were) both horribly static. Neither of them allowed recursion (at least in their official versions). Consequently, nobody cared about a built-in stack. Few of the popular CPU architectures supported it.

Consequently, the preferred view of the memory area occupied by a program was a single, continuous, mostly static, chunk.

A digression

Q: What is the single most important advantage of separating code from data?

A: Reentrancy, i.e., the ability to share code among multiple processes.



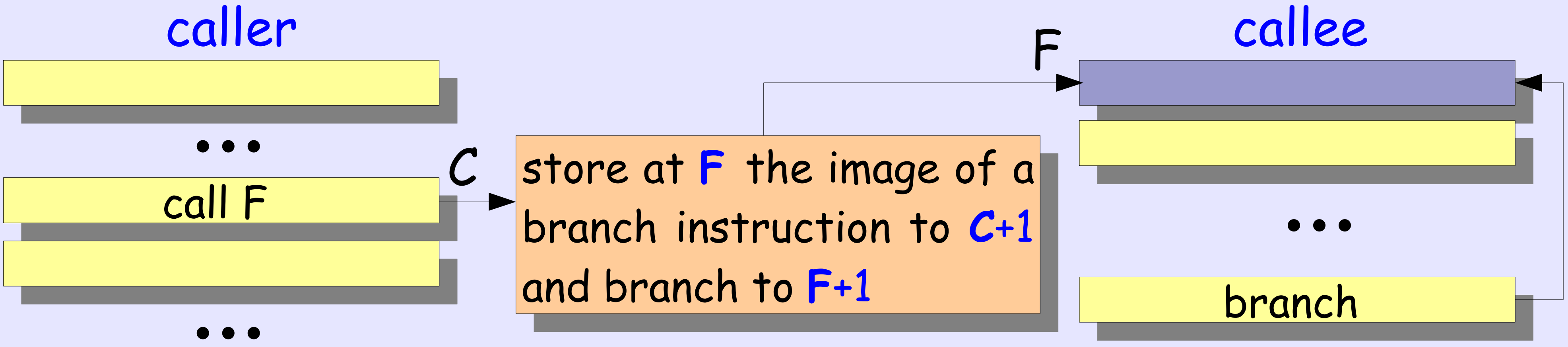
Prerequisites: 1) being able to enforce the read-only status of the code area, 2) having separate pointers to code and data.

Of course, the mechanics of program control, e.g., subroutine calls, should not relay on storing anything in the code area.

Recall **fork** !

Example

Hardware mechanism for subroutine call on CDC 6000 machines:


caller
...
call F
...
C
store at F the image of a branch instruction to C+1 and branch to F+1
F
callee
...
branch

The code image of the called function is modified.

There is a single static place (per function) where the return address is stored. Thus, there is no way to call the function recursively, unless you implement your own stack by hand.

All memory references issued by a program went to a single preallocated chunk of physical memory, so it wasn't easy to share.

Sometimes one misses the old days...

One reason for modifying code on the fly was efficiency. Its legitimacy can be disputed, but ...

```
void crunch (int N, double M) {
    for (int i = 0; i < N; i++)
        if (some_global_option)
            M [i] = 1.0/M [i];
        else
            M [i] = M [i]/M_PI;
}
```

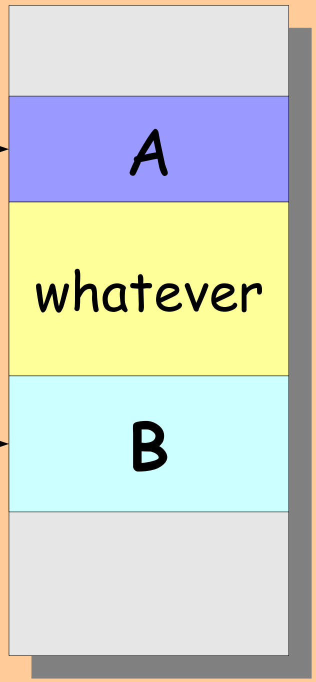
```
void crunch (int N, double M) {
    for (int i = 0; i < N; i++)
            M [i] = 1.0/M [i];
}
```

Such modifications were popular among library functions that would auto-tune themselves to global options, hardware parameters, or self-measured perceived performance.

Trampolines

Even these days the purists have to yield sometimes. Recall slide 31 (Lazy Linkage). In some cases, a dynamically created function (typically put onto and run from the stack) is demonstrably the most efficient way to accomplish something tricky.

```
int A (int u, int v) {
    int c;
    int B (int w) {
        return w + c;
    };
    ...
    return B (v);
}
```



B is a nested function (such functions are legal in gcc, for example). Note that whenever called, it must know where A's local variables are.

Everything is fine until we want to have pointers to such functions.

Look at this:

```
int A (int u, int v) {
    int c;
    int B (int w) {
        return w + c;
    };
    int (*fp) (int);
    fp = B;
    c = u;
    return C (fp, v);
}
```

```
int C (int (*fp) (int), int q) {
    return (*fp) (q);
}
```

B is blind without an extra pointer that tells it where c can be found. Thus, it needs such a pointer whenever run. So how to represent pointers to functions like B? Note that the same issue concerns pointers to member functions in C++.

But note this: in whatever context B is legitimately called, its A must be known (syntactically). The pointer to A's stack frame is a "momentary constant" of that context.

The most efficient way out ...

... is to have on A's stack a trivial chunk of code. This is what it does:

- ➔ Load the pointer to A's frame into the place (a register) when the called function expects it;
- ➔ Jump to the function according to its bare pointer (provided in some register).

Note that this is faster than, e.g., having a wrapper (library) function that would need its own stack frame, etc.

Such chunks of code (called trampolines or thunks) are used in some other places, notably, for invoking signal handlers (slide 116).

Goals of memory allocation schemes

Efficiency and flexibility combined with the full dynamics of multiprogramming. Ideally, if there is enough free memory to run a new program, the program should be able to run, regardless of where the free memory is physically located (and how fragmented it is).

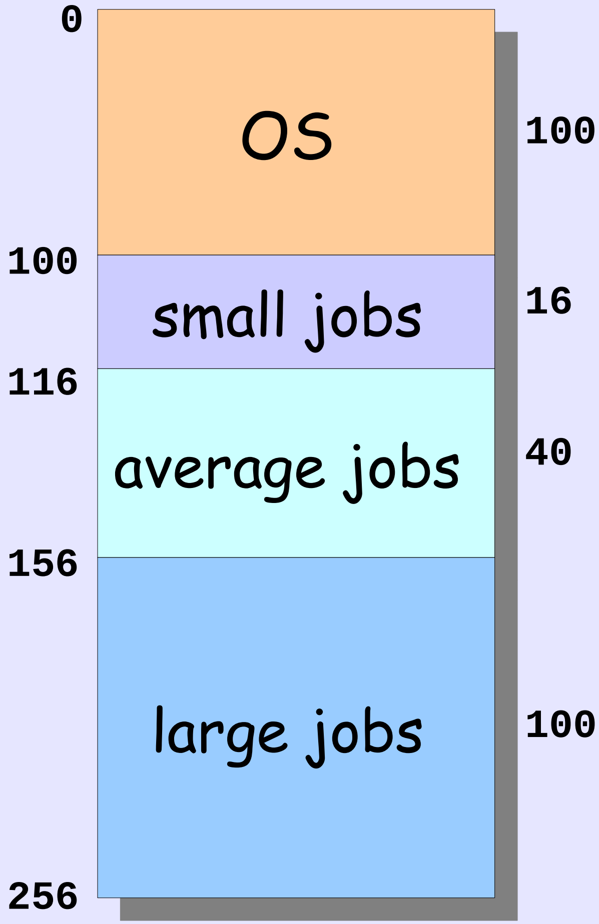
Separation of concerns. The system deals with physical memory, whereas programs want to see logical memory. A process's view of its logical address space should not depend on the physical organization of memory.

Protection coexisting with collaboration opportunities. Programs should not be able to interfere, except in controlled and agreed upon collaboration scenarios.

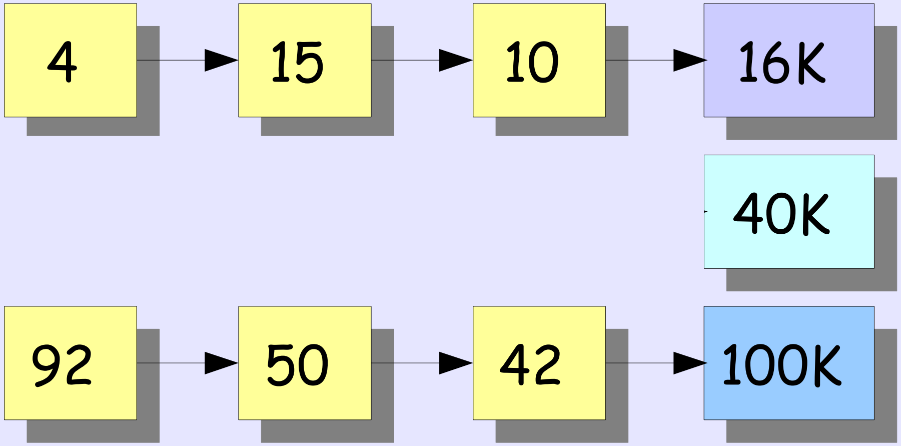
Fixed partitions

E.g., regions in the MFT version of OS/360

The physical memory was divided into fixed and rigid chunks, for example, 256KB of memory could be partitioned like this:



Allocation strategies? Best fit?



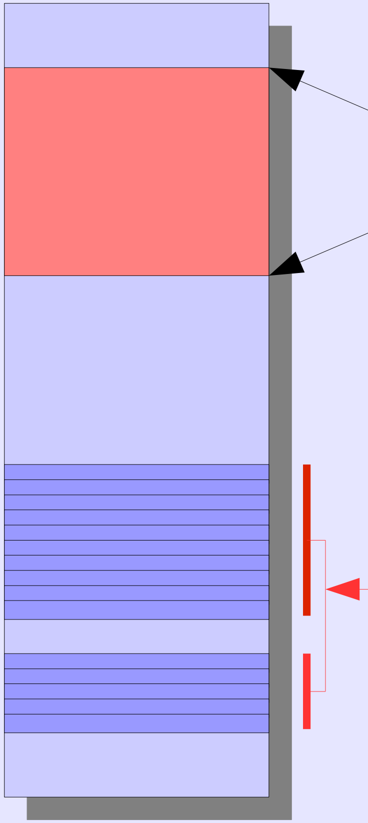
What to do if a queue to a large partition becomes empty? Should the processes be allowed to migrate up?

There were some obvious problems

A job prepared to run in one partition may not be able to run in another one. Different partitions start at different addresses. Should relocatability be postulated for all jobs?

Protection tools are needed to eliminate interference, e.g.,

Fence registers (becoming part of the context info),



from

to

or

from

length

or storage access keys, as on IBM 360:

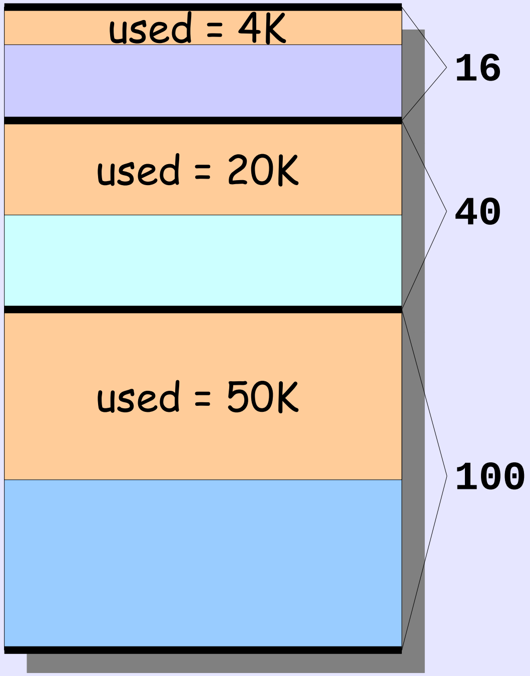
```
63                                     31          24                        0
+--------+----+-+-+-+-+----------------+--+--+----+------------------------+
|SYS MASK|MKEY|A|M|W|P| Interrupt Code |IL|CC|PMSK|   Instruction Address  |
+--------+----+-+-+-+-+----------------+--+--+----+------------------------+
```

IBM 360 PSW

... offering a finer granularity of protection (individual 2K frames) and enabling memory sharing.

Fragmentation...

... i.e., the problem haunting the early naive memory allocation strategies.



total free = 82K

Internal fragmentation, i.e., the unused leftovers of partitions.

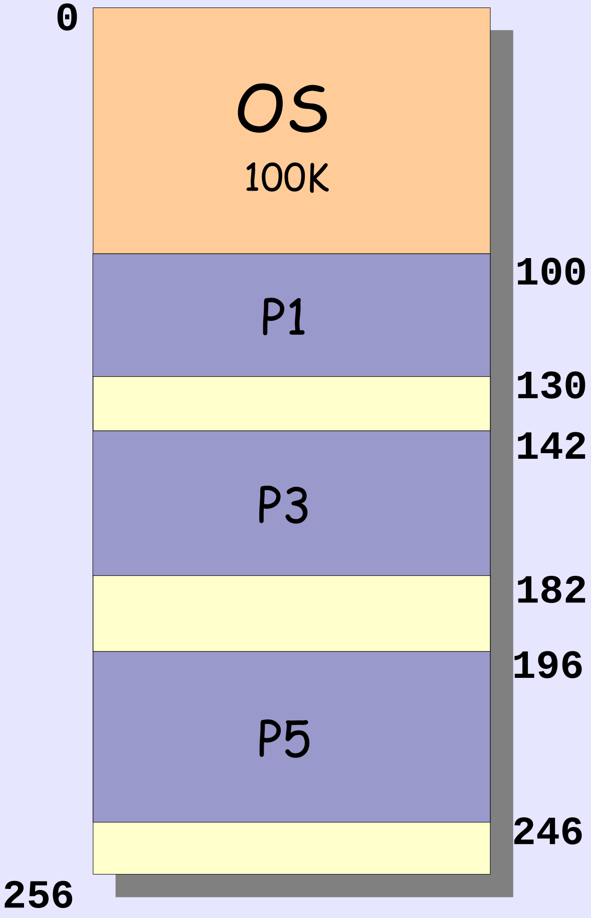
External fragmentation, i.e., the fact that the partitions exist at all. For example, we cannot run a 128KB program, even when all partitions are empty.

Can the explicit partitions be eliminated and replaced with a more dynamic scheme, e.g., adaptable to the actual configuration of jobs to run?

Variable partitions ...

... e.g., MVT on IBM/360

The OS creates partitions automatically as jobs come (and go):



Now, if it happens that P4 finishes next, we will have two holes. Clearly, this approach appears to be prone to internal fragmentation. Is there a way to deal with it?

Gap filling strategies

Is it possible to statistically reduce the impact of fragmentation?

Best Fit

The program is put into the tightest hole that can accommodate it. This way we are trying to create as small holes as possible.

Worst Fit

The program is put into the largest hole. This way we are trying to create large holes that can possibly still accommodate other programs.

First Fit

Don't bother. Put the program into the first hole big enough.

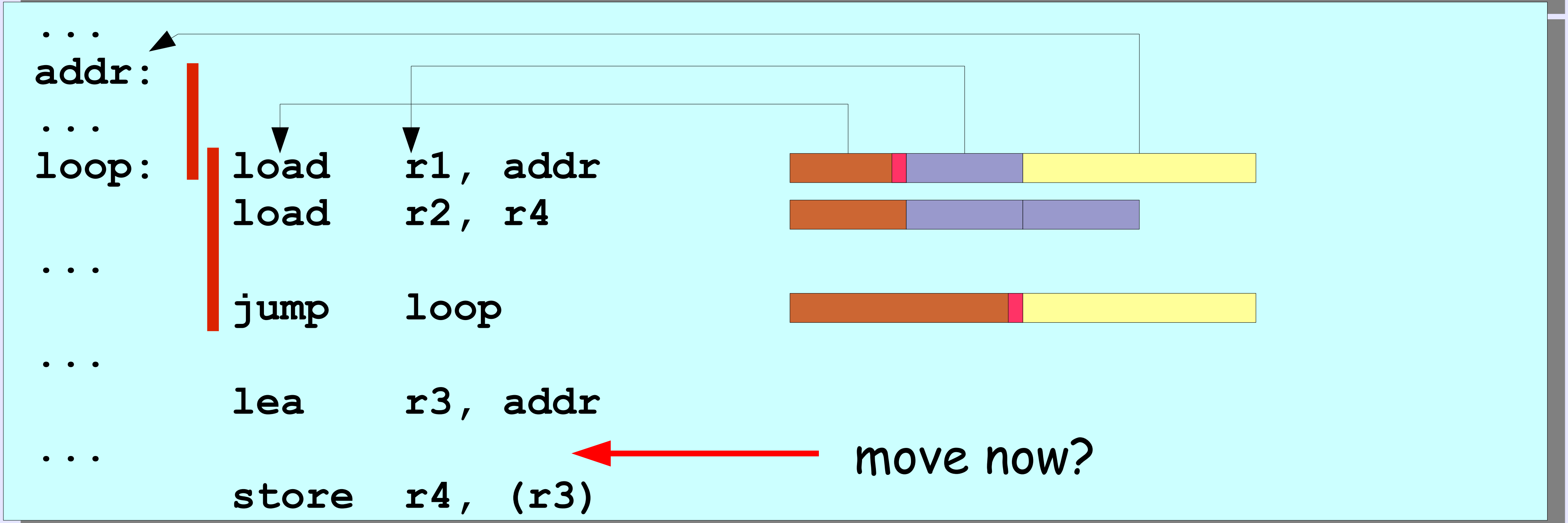
The bottom line is that trying to be smart does not help a lot. In a system like this, serious fragmentation is bound to occur.

Compaction would be nice

Can we move programs in memory to reclaim a single continuous free chunk?

Certainly, not just like that! Few programs are movable. [Recall our discussion: relocatability, position independence, movability].

Position independence may not be such a big deal these days. But movability is something quite different.



```
 ...
addr:
 ...
loop:   load    r1, addr
        load    r2, r4
 ...
        jump    loop
 ...
        lea     r3, addr
 ...
        store   r4, (r3)
```

move now?

Compaction requires some hardware

For example, CDC 6000 series (SCOPE, NOS):

All memory addresses issued by the CPU are translated using two special (protected) registers named RA and FL:

```
if (issued address >= FL)
    exception;
else
    actual address = issued address + RA;
```

The two registers are reloaded (as part of the process context) whenever the CPU is assigned to a process.

This mechanism adds a bit to the complexity of memory references, but it accomplishes one important goal: we can begin to differentiate between logical address space and physical memory.

Compaction with RA + FL ...

... accomplishes: protection, separation of concerns, flexibility (understood as eliminating fragmentation) ...

... but it falls short of solving all our problems. For example, how to implement memory sharing?

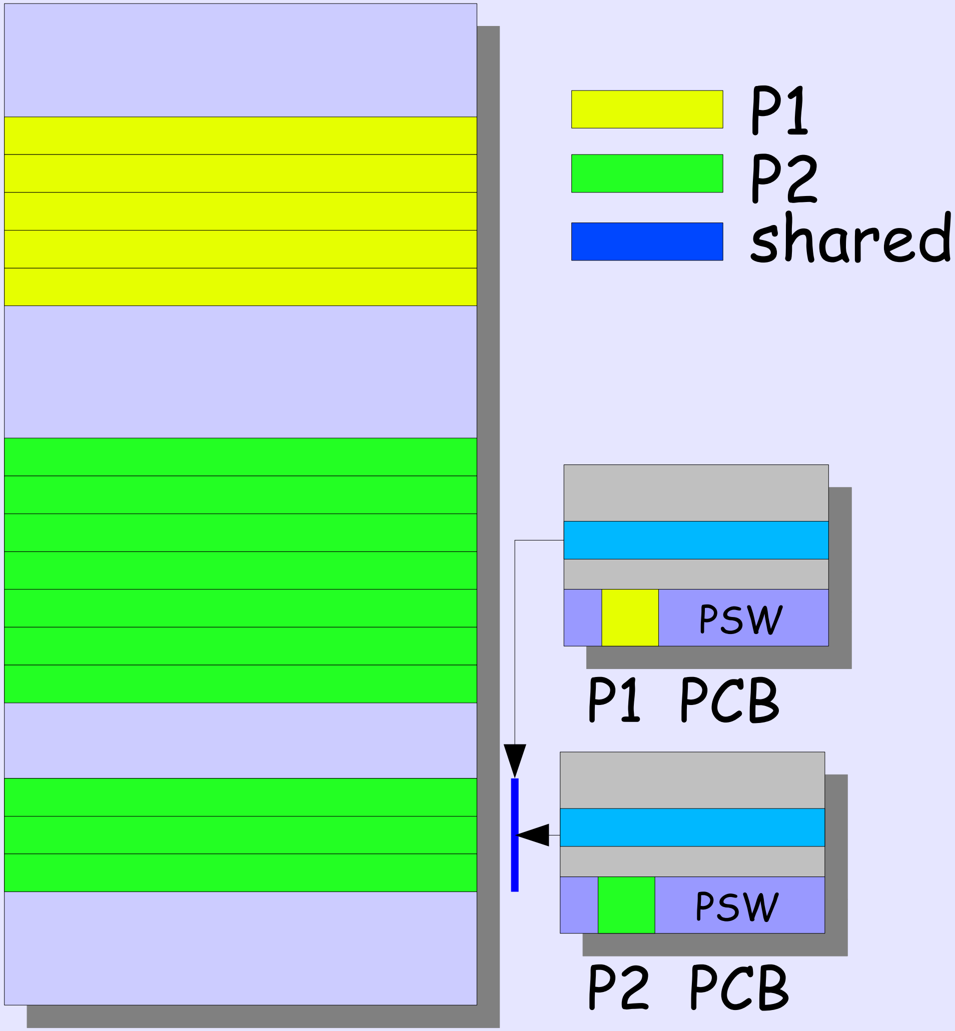
No way! The best we can do is to emulate it via system calls, i.e., some kind of message passing.

One more problem is the fact that what a program gets is one continuous chunk of memory with exactly the same accessibility attributes. Everything is writeable.

Well, there is no memory sharing, so the concept of, say, reentrant and write-protected code is completely useless anyway.

Memory sharing ...

... was in fact simpler on systems without hardware relocation. For example, you could have it on the 360:



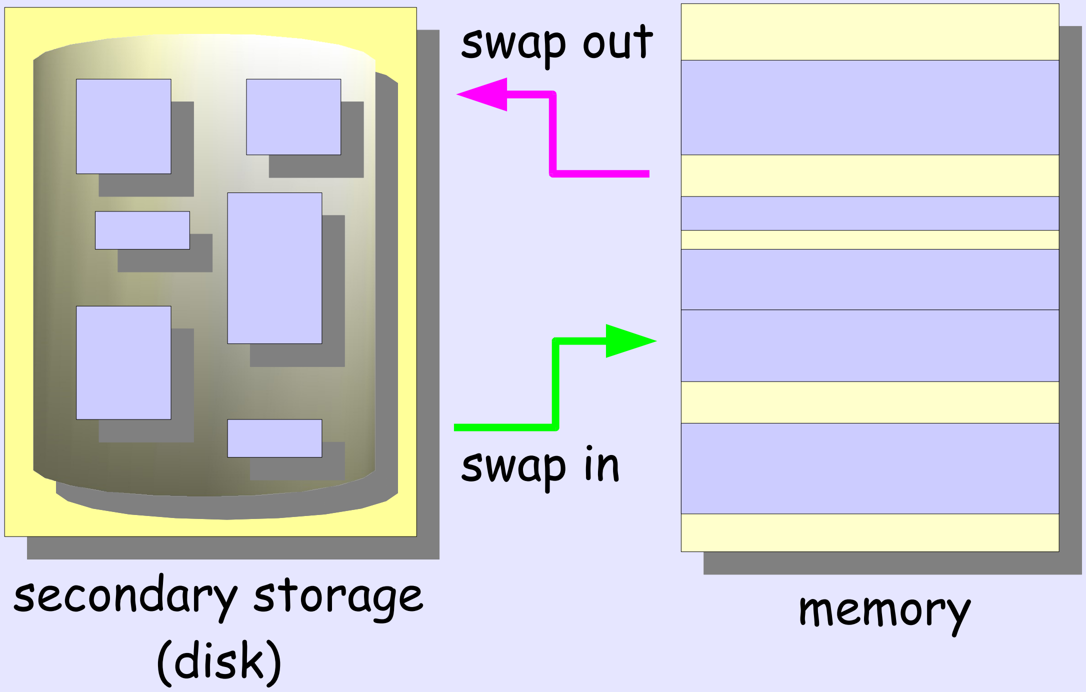
The shared area is described in the PCBs of P1 and P2.

P1 is given the CPU: the system resets the storage access keys of the shared frames to yellow.

P2 is given the CPU: the system resets the storage access keys of the shared frames to green.

Swapping ...

... i.e., turning memory into a preemptible resource. Programs can be suspended and swapped out, then swapped in later. Particularly useful with movable partitions.



The pool of programs in execution can be larger than what can fit into memory at once.

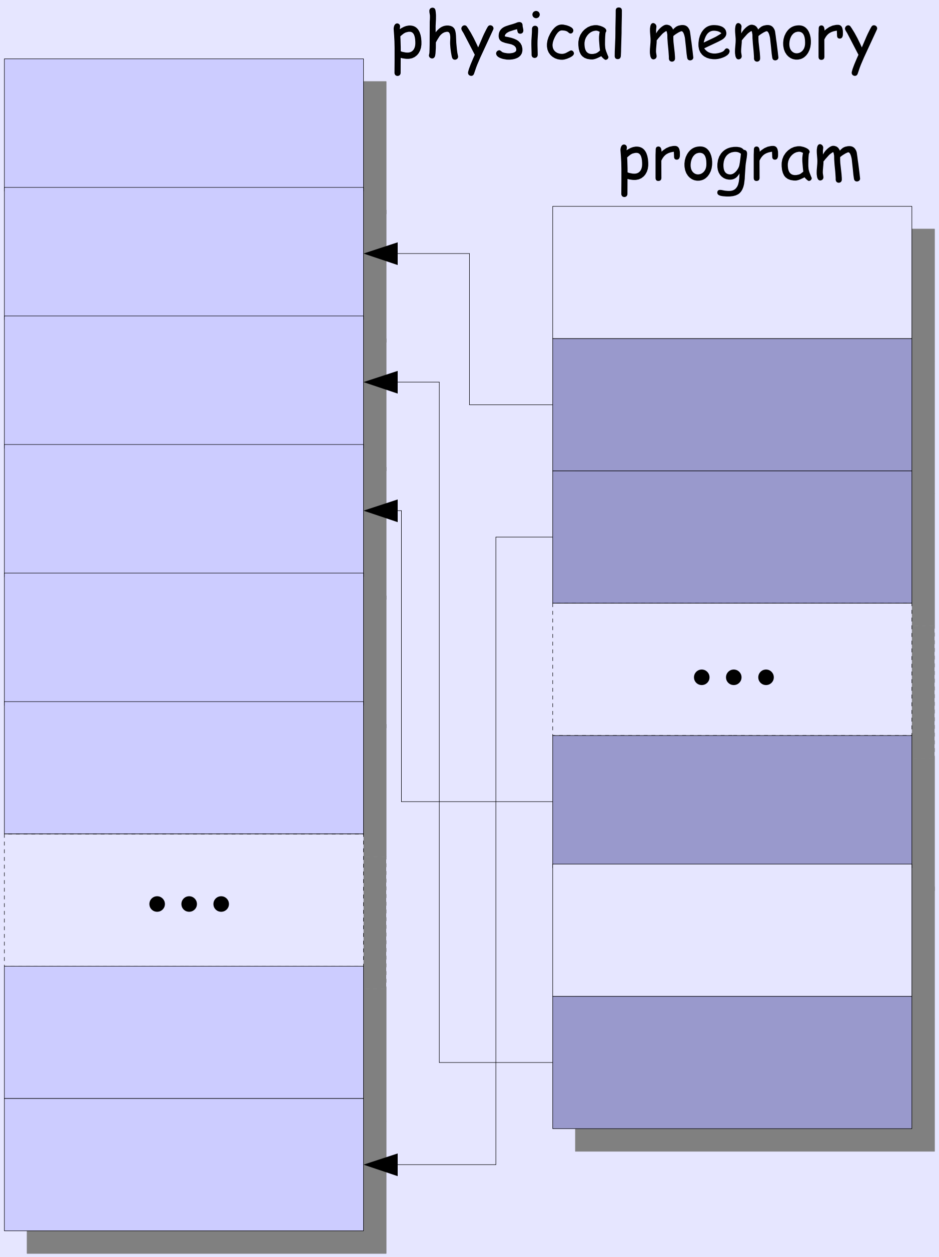
There is one more level where the programs compete for system's attention, i.e., one more level of scheduling.

The system is always able to accommodate an important program without having to kill any of those currently in execution.

When to swap out?

- The program has exceeded its inter-swap interval, and there are other programs waiting to be swapped in (fair processing).
- The program has reached a stage when it has to wait for a longish time (for something “truly external”). Beware that programs blocked on I/O may not be good candidates for swapping out. Why? When does a program's memory become a non-preemptible resource?
- A high priority program requires immediate execution and there is no other way to accommodate it.
- With some partition systems, e.g., ones with compaction, a program can grow or shrink. When a program wants to grow too much, the system may have to swap it out until the situation becomes more favorable.

Paging



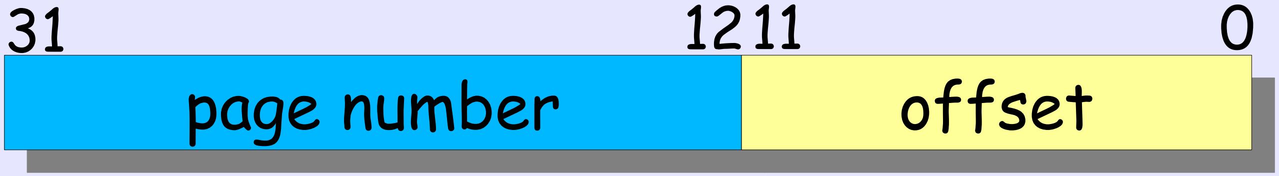
Physical memory is divided into fixed-size blocks called frames. The logical address space of a program is divided into chunks of the same size called pages. There is a mechanism for mapping any page into any frame.

Some parts of the logical space may be unmapped.

- Typical page sizes:
  - Intel x86: 4096 bytes
  - MC68451: 256 bytes minimum
  - IBM-370: 2048 or 4096 bytes

Pages and frames

The page size is always a power of two. This simplifies the interpretation of a memory address. For example, for 4K pages,

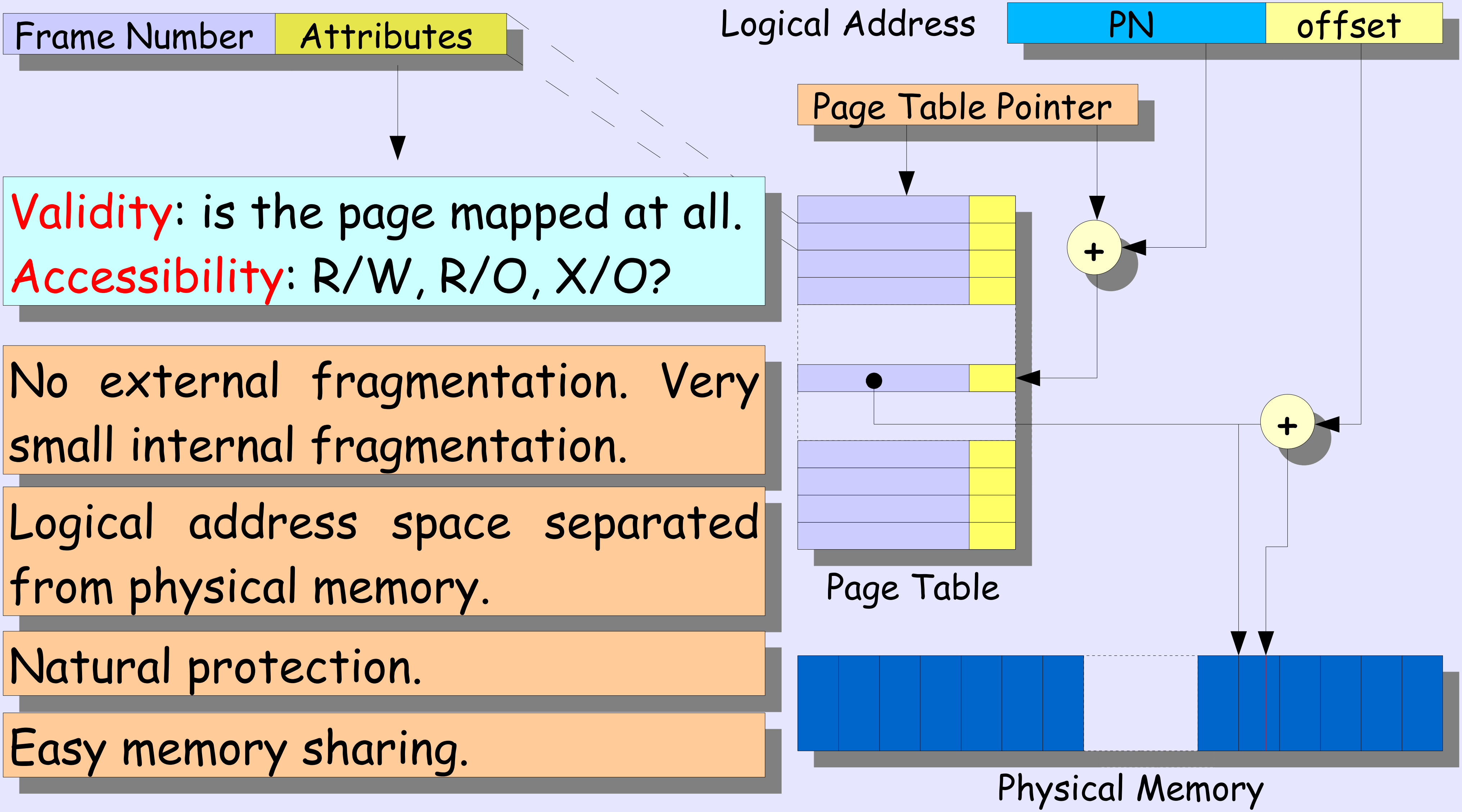


This operation is carried out in hardware, essentially at every memory reference (at least in principle).

Typically, the mapping of pages into frames is described in a special data structure called the page table. The page table becomes part of the process context. It describes the complete memory layout of the process.

Details depend on the architecture. The page table is usually pointed to by a special protected register of the CPU called the page table pointer.

Page tables



Validity: is the page mapped at all.
Accessibility: R/W, R/O, X/O?

No external fragmentation. Very small internal fragmentation.

Logical address space separated from physical memory.

Natural protection.

Easy memory sharing.

Address translation ...

... is optional, i.e., it can be switched on or off by the system (by setting/clearing some bit in some protected register). There are situations when the system must operate in physical memory, e.g.,

Accessing special locations, e.g., corresponding to registers of peripheral devices.

Supplying addresses (buffers) to peripheral devices as parameters of physical I/O operations.

Setting up the page tables.

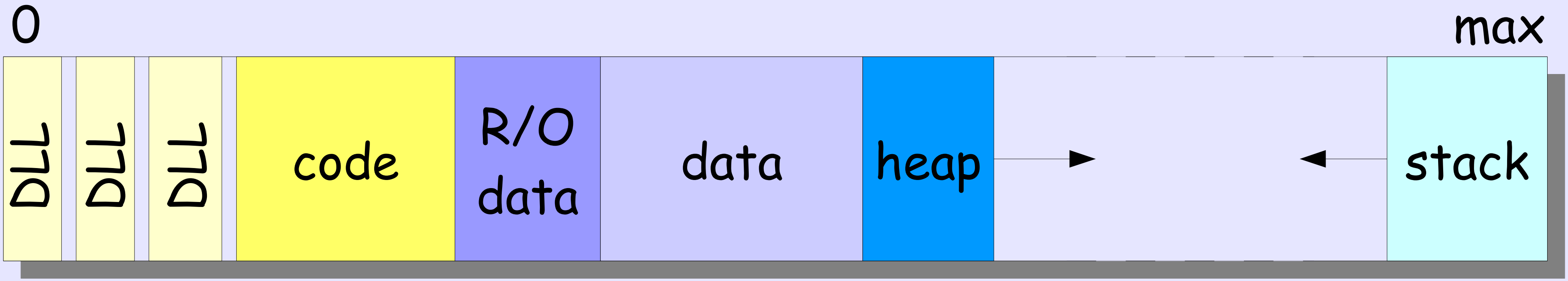
Note: generally, peripheral devices cannot take advantage of address translation. Why?

Address translation is context-relative (different processes see different logical address spaces). When a peripheral device is running, the CPU context usually describes an unrelated process.

Multi-level tables

The actual (used) portion of a program's logical address space need not be continuous. Different chunks can be assigned different accessibility attributes.

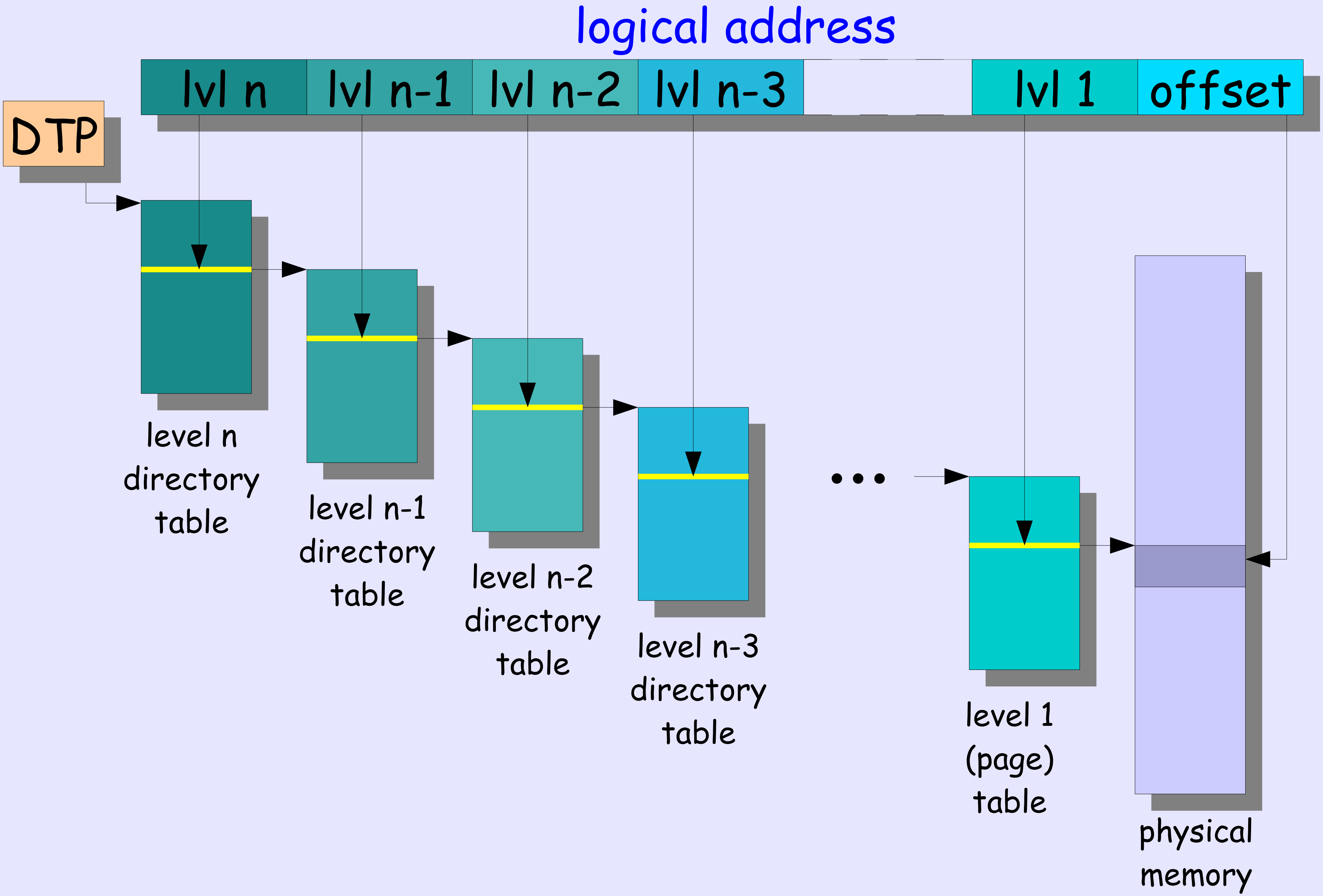
Thus, it makes sense to assume that the logical address space has the same layout for all programs (large and small), e.g.,



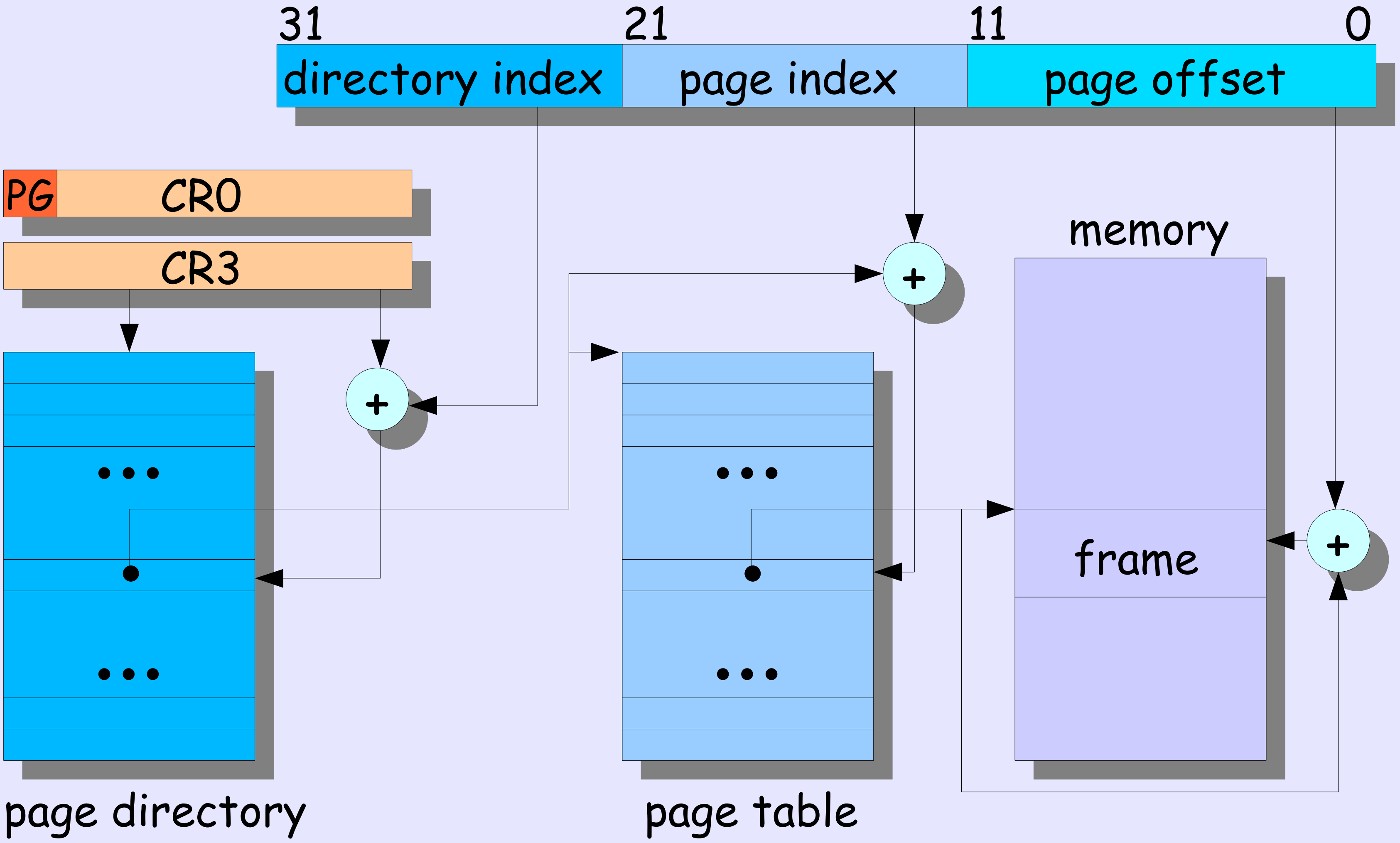
This is doable in principle with a page table (with lots of invalid entries), but the page table may become huge.

For example: 32-bit address = 4GB address space; with 4KB pages, we need 1M page table entries (4 bytes each?) = 4MB.

Generally, we may have ...



Example: 32-bit Pentium



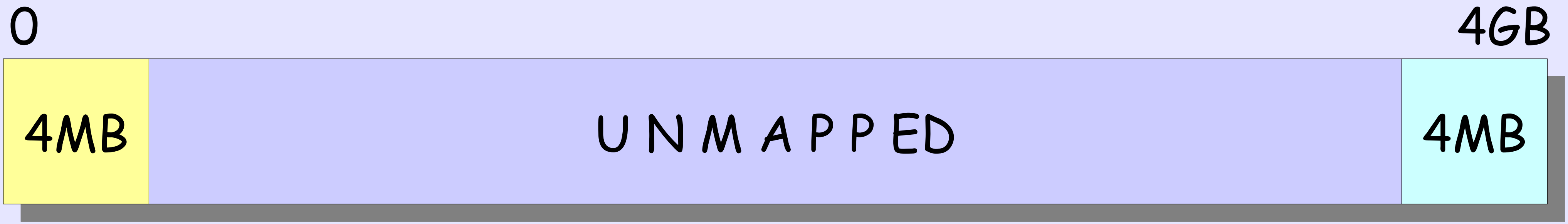
PDEsz = 4B, PTEsz = 4B, Psz = 4KB, PDsz = 4KB, PTsz = 4KB

What does it buy us?

Quite a bit of complication for one thing. But suppose that a program wants a little something at the beginning of the address space + a little something at the very end.

With one level, we have: PTS = PTEsz x (4G/Psz) = 4MB of tables.

With two levels, we have: PDsz = 4KB, PTsz = 4KB. Total = 3 x 4KB = 12 KB of tables. With this much, we can map a 4MB piece at location 0 and a 4MB piece at the end (e.g., for the stack).

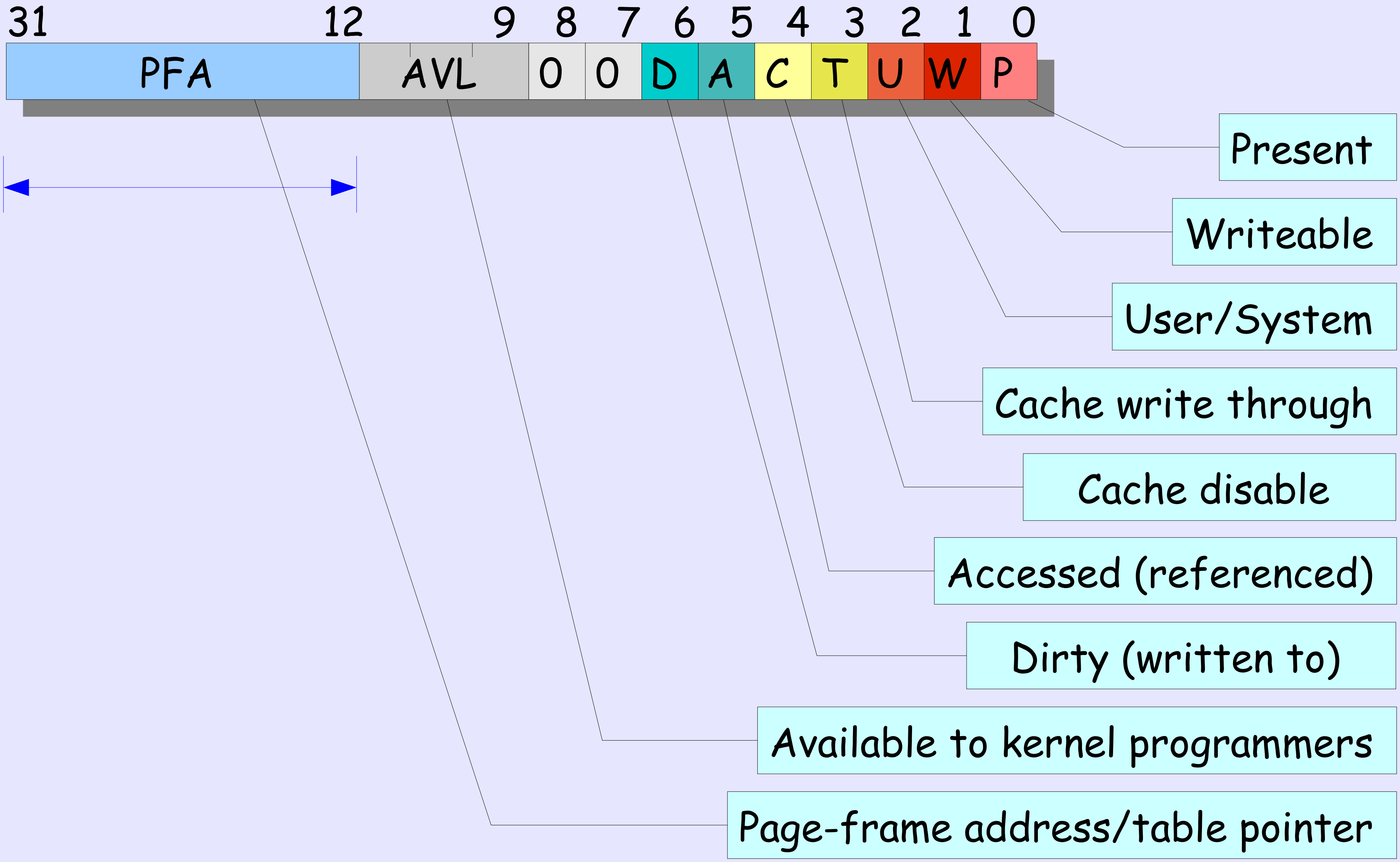


4MB = 1K pages

1 table (PD or PT) = 1K entries

1 entry = 4 bytes

PDE/PTE format



31 12	9	8	7	6	5	4	3	2	1	0
PFA	AVL	0	0	D	A	C	T	U	W	P

Present

Writeable

User/System

Cache write through

Cache disable

Accessed (referenced)

Dirty (written to)

Available to kernel programmers

Page-frame address/table pointer

A few notes

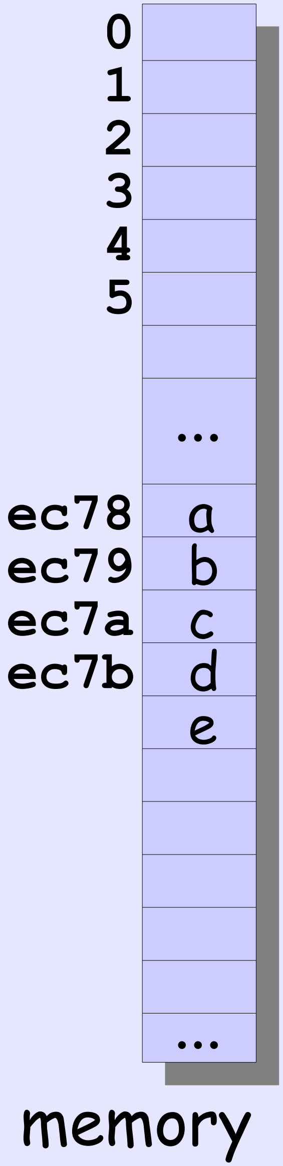
The size of PFA determines the maximum amount of physical memory that the system can use.

Attributes can be specified on a per-page or per-directory basis. Generally, it makes sense to associate accessibility attributes with chunks larger than pages.

Sometimes, in two-level hierarchies, the directories are called segments. This reflects the fact that they provide a coarser grain of allocation – one that better corresponds with logical segments of programs.

With all these levels of indirection, the cost of a memory reference is bound to skyrocket, unless something is done. This is taken care of by some extra hardware called TLB.

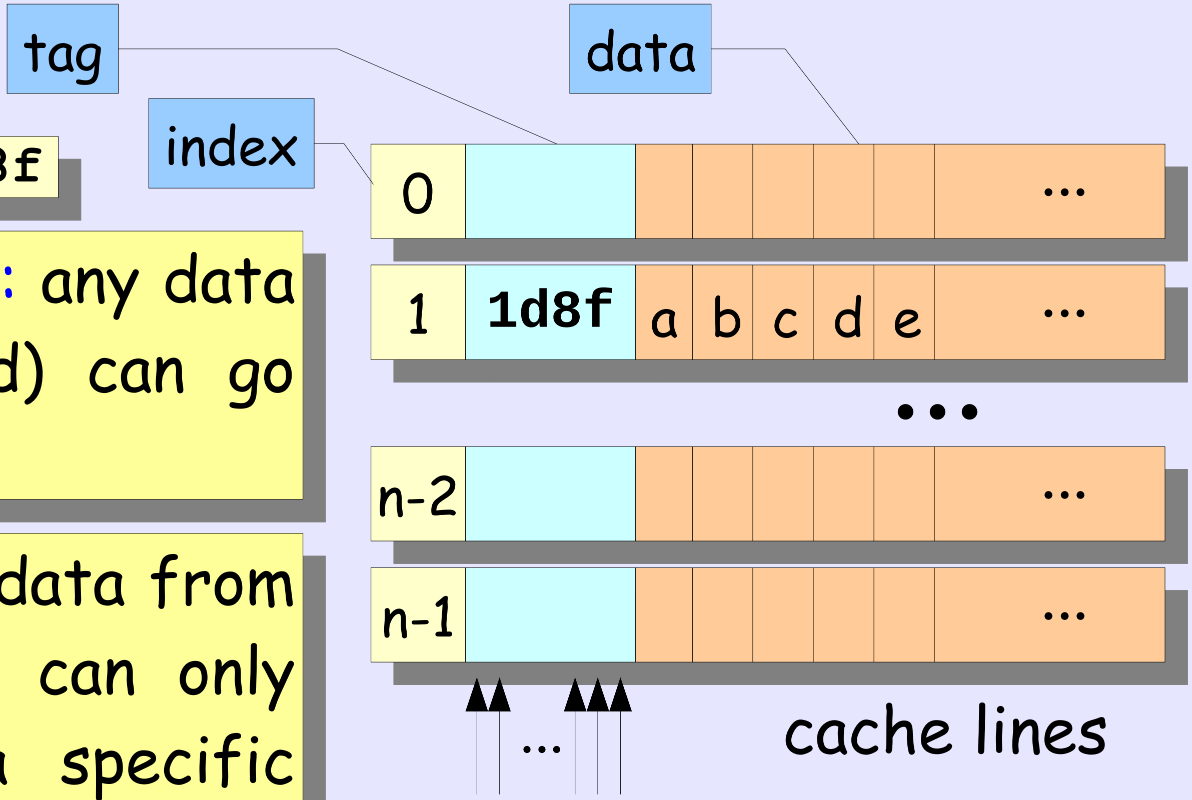
A bit on caches



`ec78 >> 3 == 1d8f`

Full associativity: any data (properly aligned) can go to any line.

Direct mapping: data from a given location can only be stored in a specific line.



For example, suppose that index = (addr >> 3) & 0xff. Then, location ec78 is always mapped to line number 8f (143) and can never be mapped elsewhere.

Caches are characterized by ...

The number of lines and line size (the product == cache size).

Associativity rules: fully associative, direct mapped, something in between.

Replacement policy: how the lines are recycled. Note that the associativity rules may restrict the replacement policy.

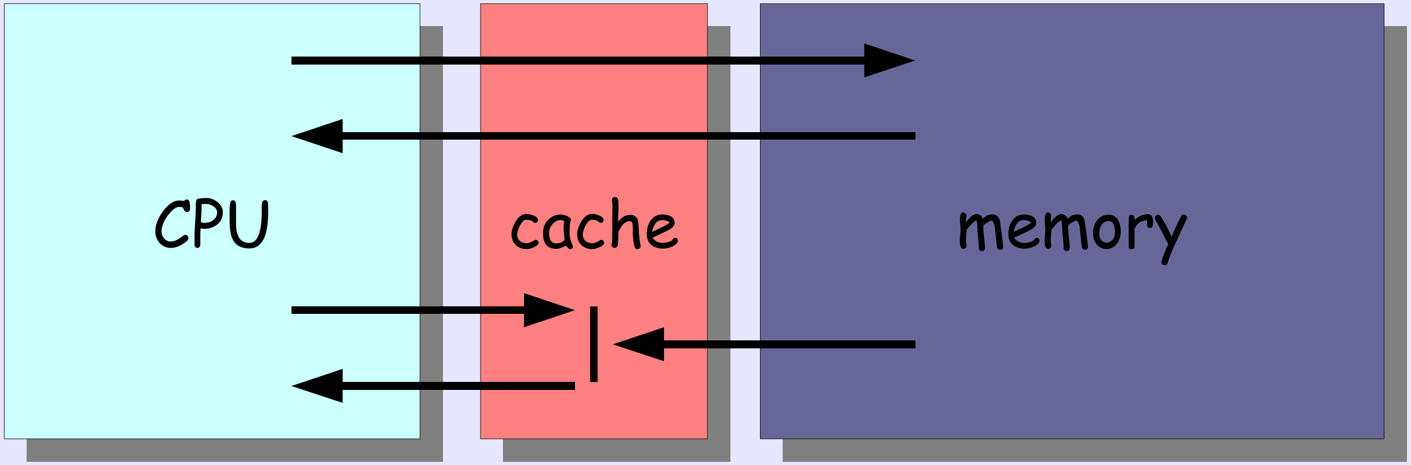
Write policy: write-through == every write to the cache results in a memory write; write-back == the data is written back when the line is reloaded and the old data is dirty (has been modified).

Note that the write policy is not without meaning in the context of multiple CPUs. Some memory references (e.g., Test & Set) may bypass the cache to ensure access coherence.

Spin locks and memory cache

A spin lock typically looks like this:

```
while (T&S (&flag));
```



Test & Set usually bypasses the cache and occupies the bus. Sometimes this may be a better way:

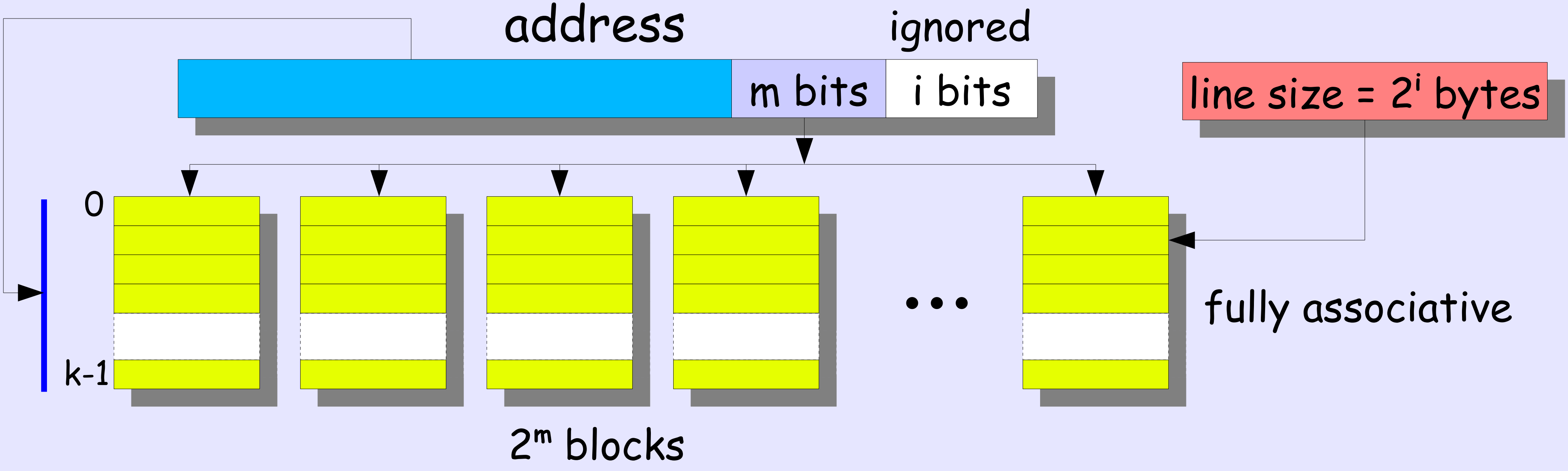
```
void spinlock (int *flag) {
  while (1) {
    if (!T&S (flag)) break;
    while (*flag != 0);
  }
}
```

Associativity types

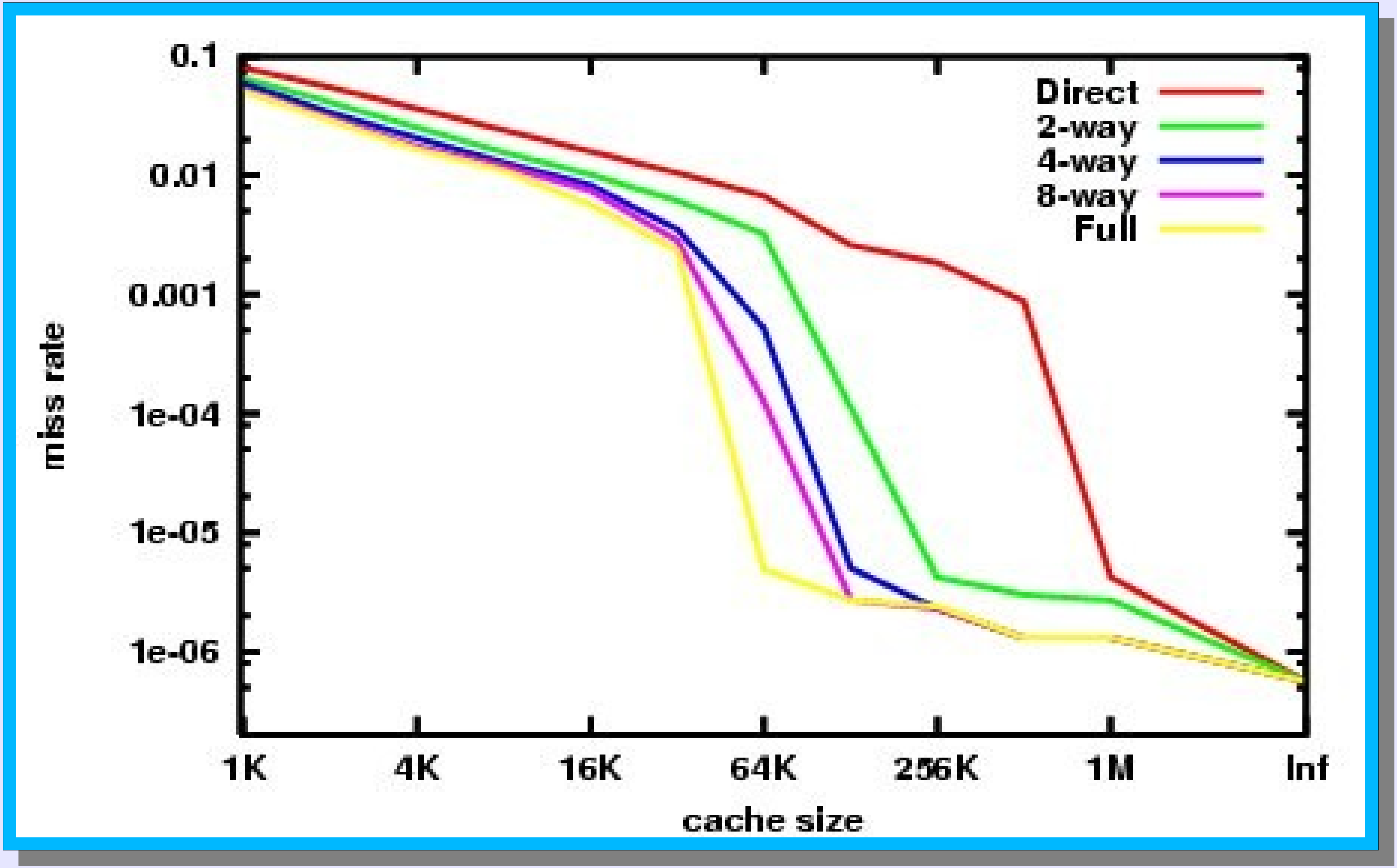
Full associativity is nice but expensive to implement. Direct mapping is simple, but gives lousy performance.

In practice, we often see some compromises between the two associativity extremes, e.g., k-way caches.

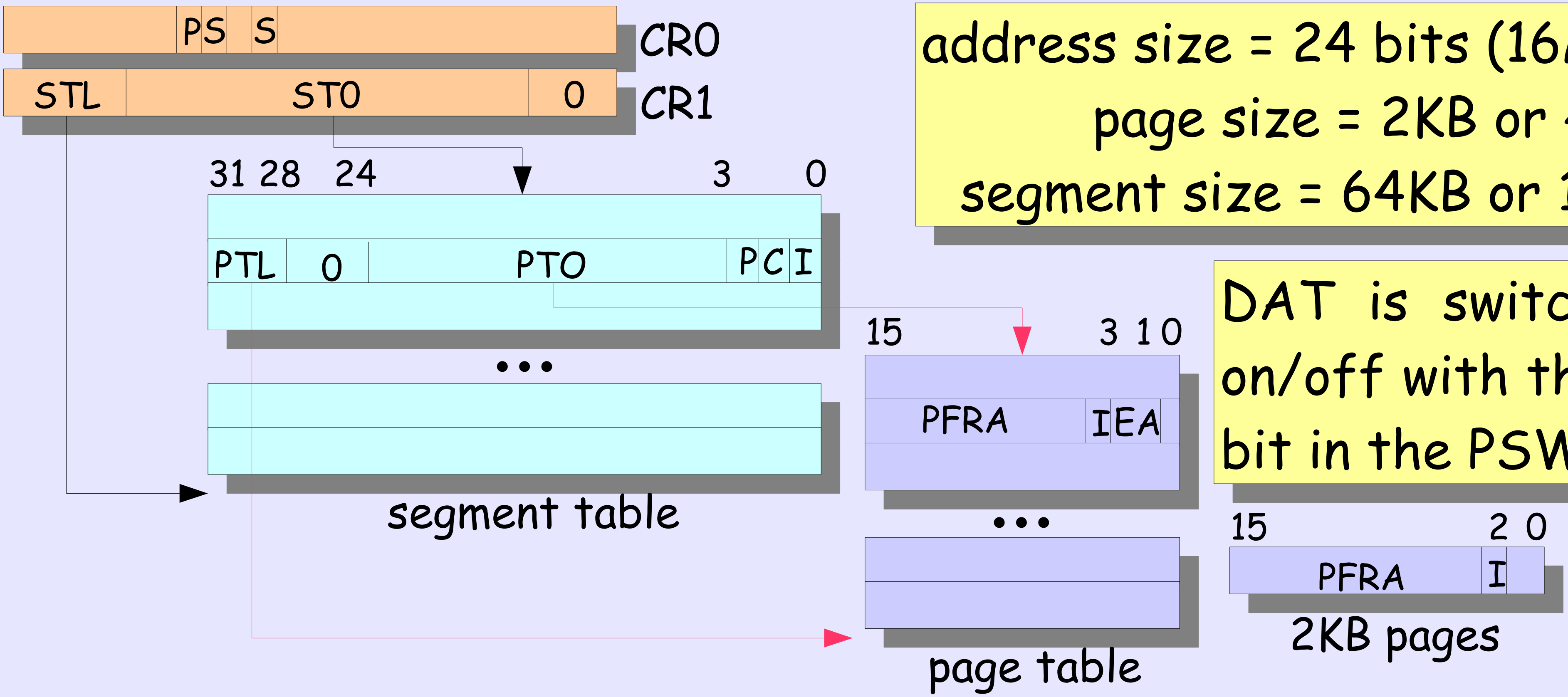
A cache is k-way, if any given memory location can be cached in any of k specific lines. Usually, k is a power of two.



Associativity tradeoffs (example)



Now for some paleontology: IBM 370

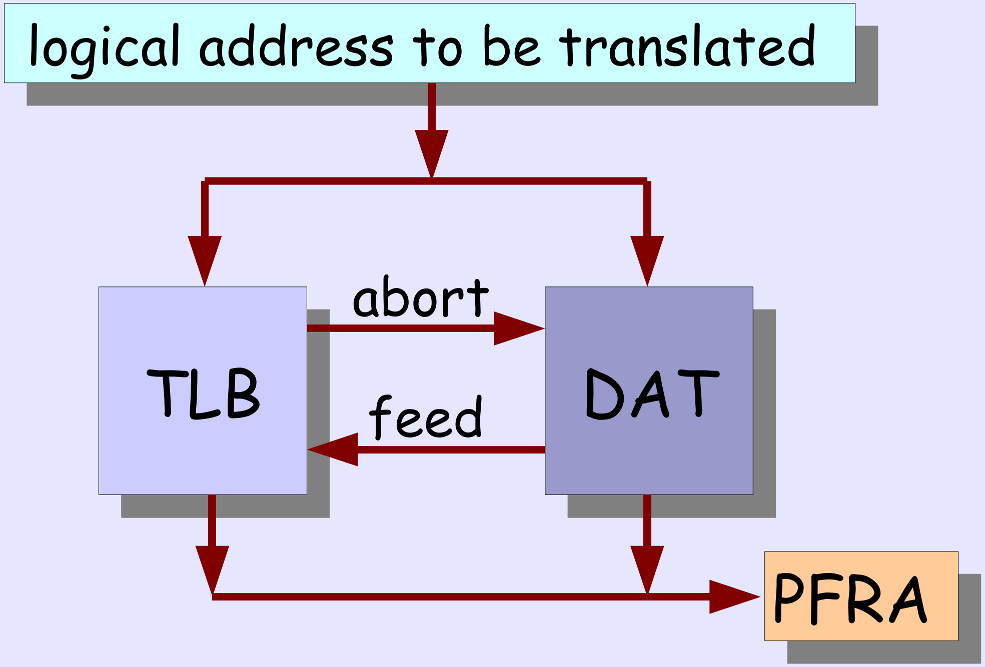


address size = 24 bits (16MB)
page size = 2KB or 4KB
segment size = 64KB or 1MB

DAT is switched on/off with the T bit in the PSW.

PTL/STL : length in increments of 1/16 of the maximum size
P : protected (read-only), I : invalid, C : common
PFRA : Page/Frame Real Address
EA : Extended Address (added later)

TLB on the 370: the mother of all TLBs

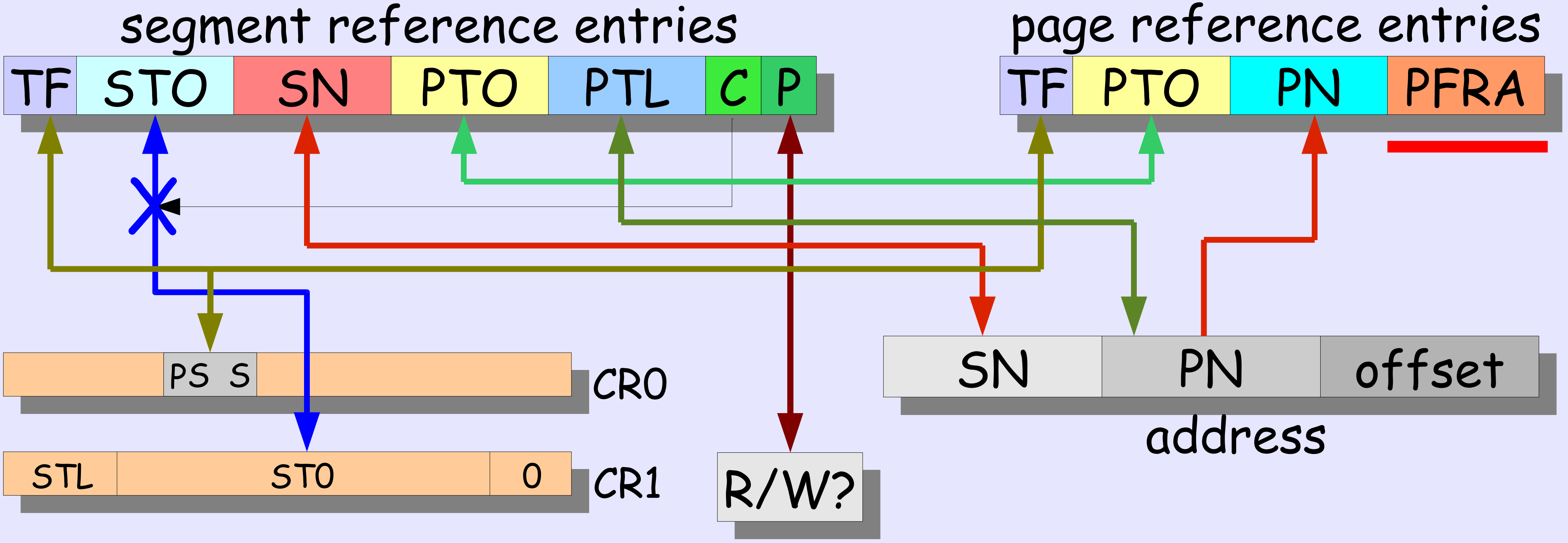


Two translation mechanism are operating in parallel. DAT is authoritative and always produces an answer, albeit slow. TLB is much faster, but it may fail.

TLB stores some information returned by DAT. If the next memory reference can be resolved by TLB, the answer will be produced quickly.

TLB on the 370 ...

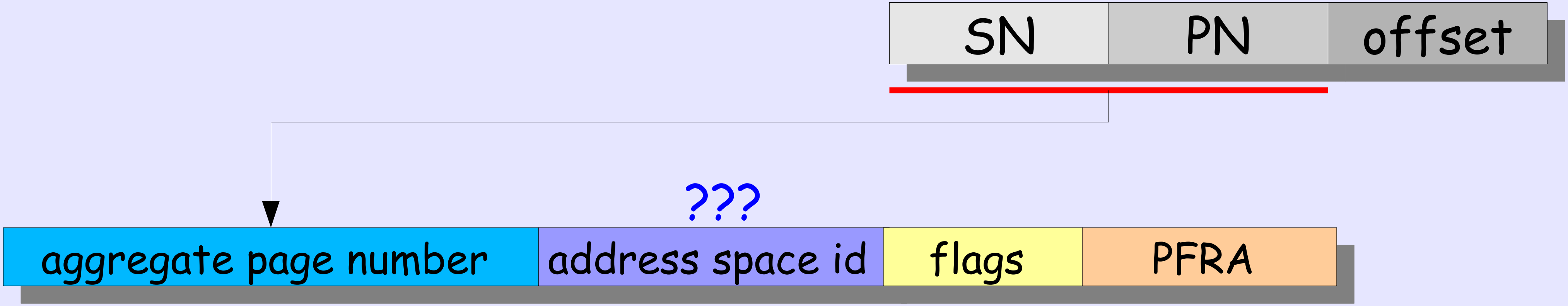
... is a specialized cache capable of storing a limited number of entries of two types:



One more little twist: If C is set, the STO matching is not verified. This accounts for the so-called common segments.

In principle, it can be simpler

An entry can only store information about pages, ignoring the segments:



This is in fact how most TLBs are implemented today. Note that the associativity string (APN) is longer than in the previous case. One can argue that using two entry types (segment/page) improves the accuracy of the TLB – for the same expense understood as the total amount of storage.

Coherence of TLB caches

Entries in the TLB may become obsolete (and incorrect) when the actual mapping changes, e.g., a page is unmapped or remapped. This tends to happen, as we shall shortly see, quite often.

Upon a context switch, completely new tables become active. There are two options:

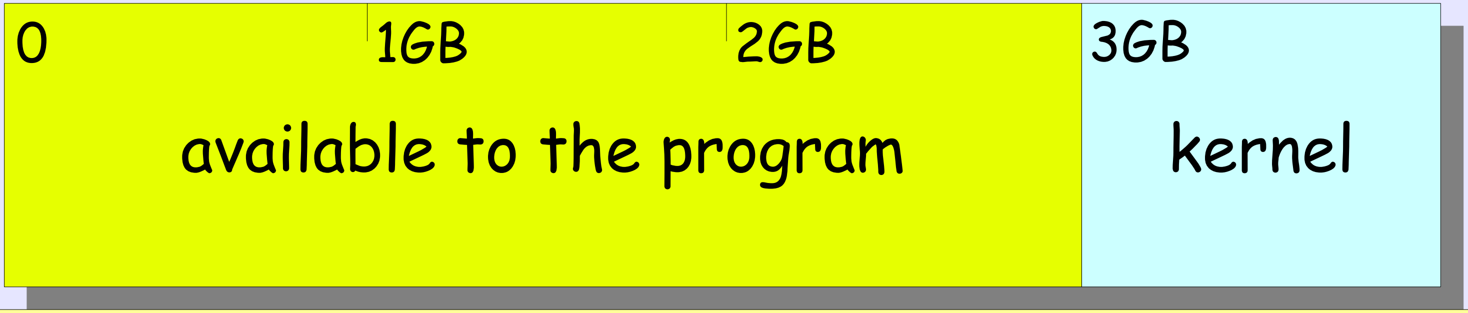
Completely invalidate the TLB. This happens on the (vanilla) Pentium. Writing to CR3 automatically invalidates the TLB.

Retain all entries, but make it possible to distinguish different address spaces (the 370 approach).

The latter is more efficient in the face of frequent context switches. Additionally, the entries pertaining to the common segment are never invalidated (as long as its mapping doesn't change).

More about the common segment

A popular approach to organizing the logical address space of a process is to include the kernel memory in it, e.g., in older Linux:

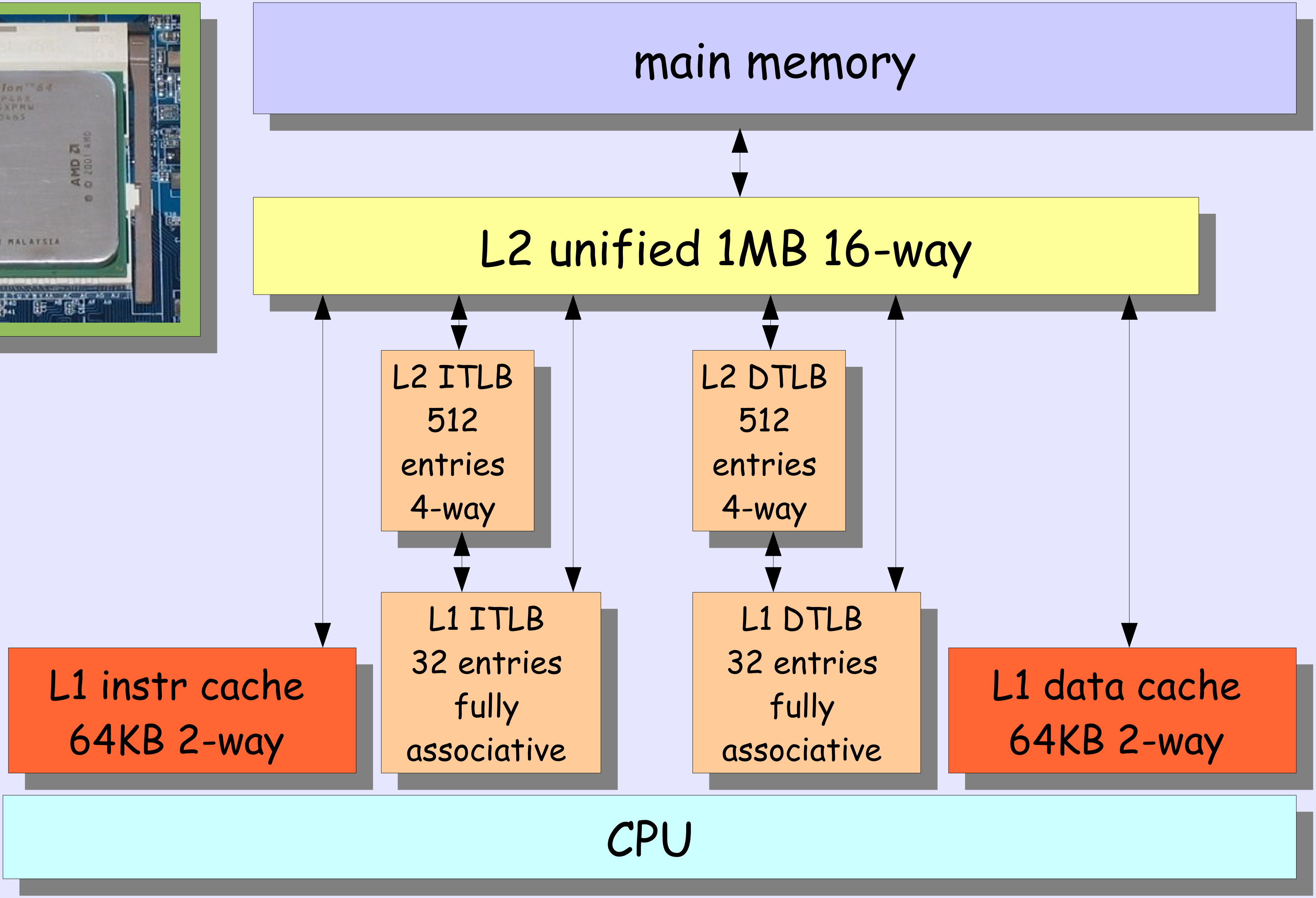


We not only execute the kernel on the process's stack, but also within the process's address space. Isn't that nice and simple?

The kernel part is in fact a common segment: it occurs in all address spaces (always in the same location).

These days, as 4GB of physical memory is not such a big deal, we have reached the limit of the 32-bit address range. Thus, “wasting” 1 GB for the kernel has become an issue. Consequently, modern (32-bit) systems tend to depart from this approach.

The K8 cache structure for Athlon 64



main memory

L2 unified 1MB 16-way

L2 ITLB 512 entries 4-way

L2 DTLB 512 entries 4-way

L1 instr cache 64KB 2-way

L1 ITLB 32 entries fully associative

L1 DTLB 32 entries fully associative

L1 data cache 64KB 2-way

CPU

The dynamics of address spaces

By now, we should be comfortable with the idea that the logical address space of a process contains holes (often very large).

Suppose that a program gets loaded into memory.

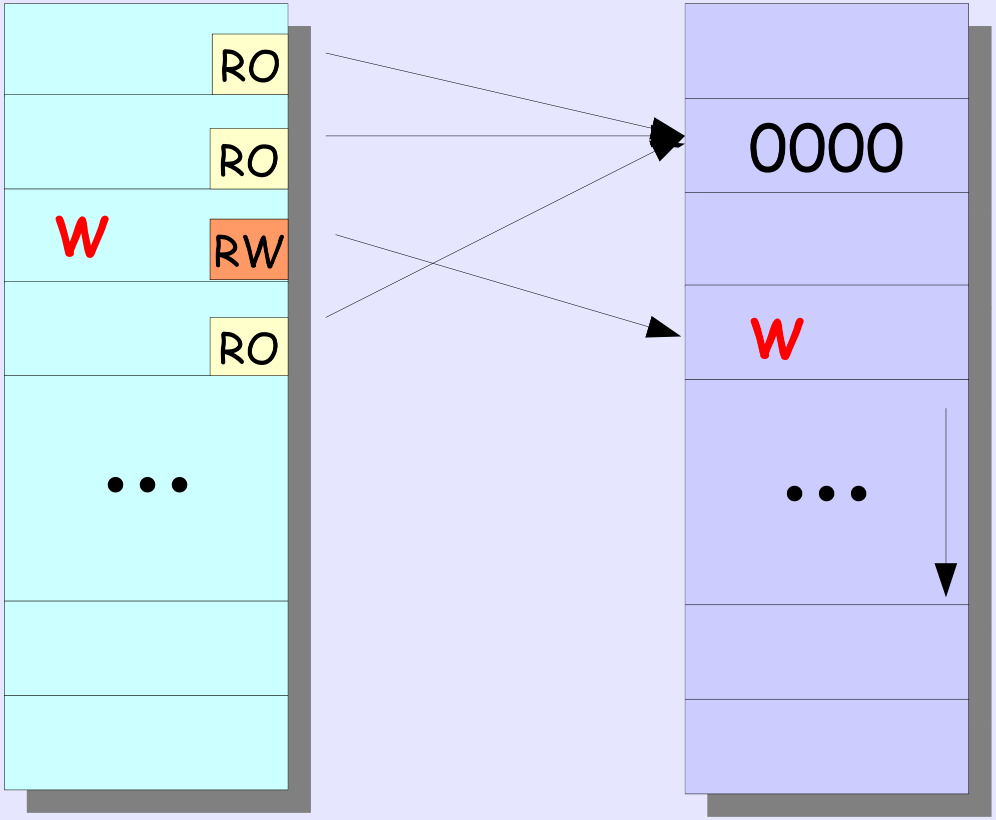
Do we have to load it entirely all at once? Well, we know about DLLs, but other than that?

Do we have to provide room for all the program's data in advance? For example, what about the stack? How can we know how much of it the program is going to need?

Can we somehow figure it out automatically (without forcing the program to tell us anything) when the program is going to use some memory, so that we can bring it in as needed?

A sample trick

Suppose that a program declares a huge array filled with zeros.



If the array happens to be constant, we can trivially save by using only one frame for the entire array. What if it isn't?

Mark all the pages as read only.

Whenever the program writes something to the array, the system will receive an exception (formally an error).

The system will automatically correct it, i.e., copy the zero frame to a new frame, mark it as read/write, and resume the program.

This is in fact how uninitialized static data area (.data) is set up.

This is called COW (Copy On Write)

Recall fork. This is how the operation is carried out:

The child inherits exactly the address space of the parent. No memory is copied!

All memory that is not to be actually shared is marked as R/O + COW.

For as long as the two processes do not write to some area, they are going to share the frames. This is why fork is so simple and inexpensive physically, while being so powerful as a concept.

This hints at a more general idea. Let us call it CRAPE for "Creatively Responding to Apparent Program Errors."

The program errors we have in mind are called page faults.

More tricks?

When a program pushes something onto a nonexistent page of its stack, the system allocates a blank new frame and restarts the program.

No need to load the entire program at once. As the program runs through its code, the system receives an exception whenever the program references a new (so far untouched) page. Then it brings the new code fragment into memory and restarts the program.

No need to allocate the entire data at once – by the same token.

Suppose that the system wants to emulate something (e.g., a device) as a bunch of memory locations. It can easily intercept all references to a given page range and fool the program into thinking that is actually reads from and writes to memory.

Memory mapped stuff, typically files

Here is a standard system call available on UNIX systems:

```
void *mmap (void *start, size_t length, int prot,
                      int flags, int fd, off_t offset);
```

It allows you to map a file fragment into memory. In a nutshell, this is how a program is loaded for execution. Recall this:

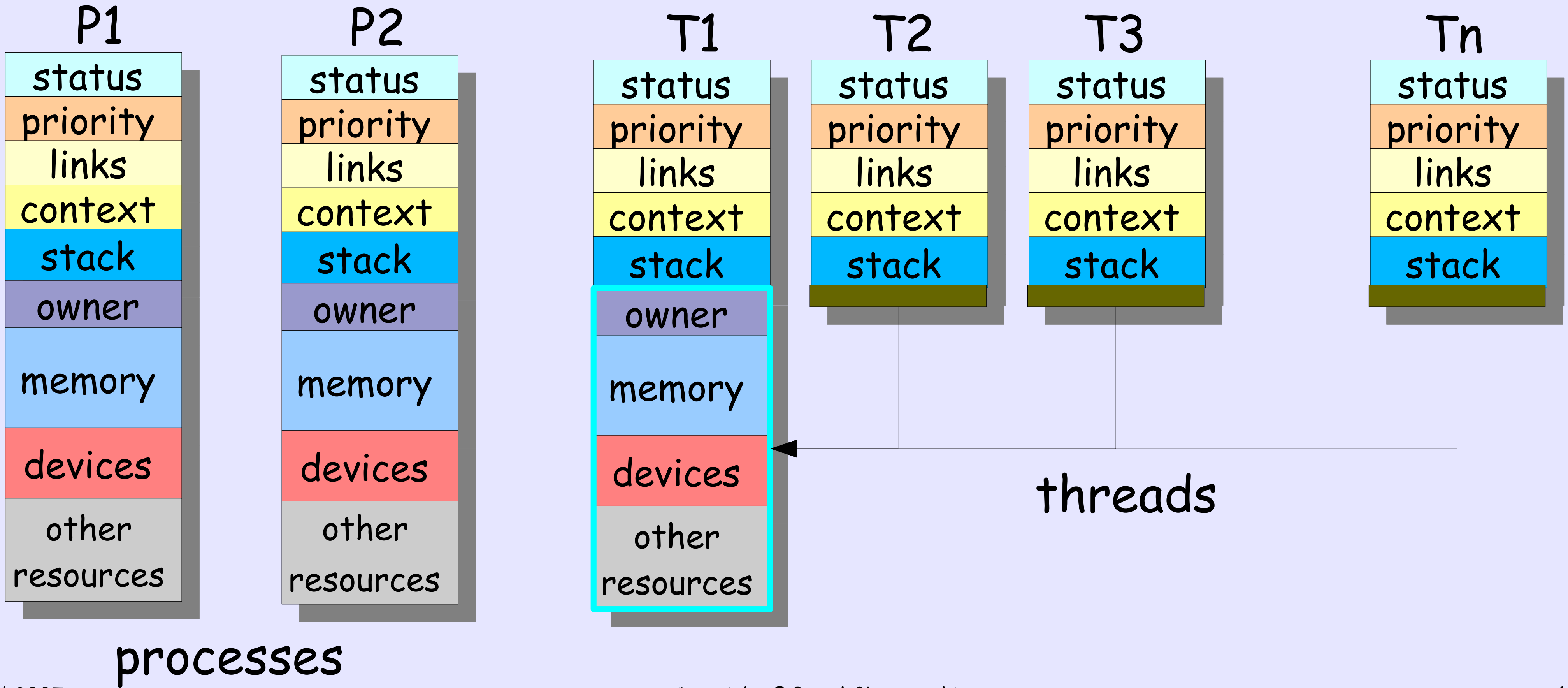
```
    PHDR off    0x00000034 vaddr 0x08048034 paddr 0x08048034 align 2**2
         filesz 0x000000c0 memsz 0x000000c0 flags r-x
  INTERP off    0x000000f4 vaddr 0x080480f4 paddr 0x080480f4 align 2**0
         filesz 0x00000013 memsz 0x00000013 flags r--
    LOAD off    0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
         filesz 0x0000059d memsz 0x0000059d flags r-x
    LOAD off    0x000005a0 vaddr 0x080495a0 paddr 0x080495a0 align 2**12
         filesz 0x00000114 memsz 0x00000134 flags rw-
 DYNAMIC off    0x000005ec vaddr 0x080495ec paddr 0x080495ec align 2**2
         filesz 0x000000c8 memsz 0x000000c8 flags rw-
    NOTE off    0x00000108 vaddr 0x08048108 paddr 0x08048108 align 2**2
          filesz 0x00000020 memsz 0x00000020 flags r--
```

Let us call this idea lazy loading. The program's components get loaded gradually as they are (implicitly) demanded.

Processes versus threads

Threads are strongly related processes that typically share all memory (and most other resources) except for the stack.

Some systems do not differentiate much between processes and threads. Some others economize on internal data structures.



Prerequisites for CRAPE

In order for CRAPE to work, the respective program faults must be 100% correctable. This means that:

The fault (an internal interrupt) should return enough information for the system to identify the exact source of the problem.

The partial effects of the last instruction (the one that has triggered the fault) must be accountable for.

ADD A,B,C

(A) + (B) -> C

fetch instruction

fetch (A)

fetch (B)

compute

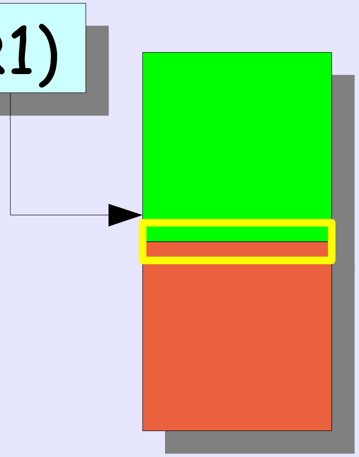
store at C

LOAD R1,R2,OFF(R1)

fetch instruction

load R1

→ load R2



If R1 is overwritten before the fault, the instruction cannot be restarted.

Basically there are two approaches

- Before executing a potentially unrestartable instruction, check if all the data it needs are in memory. If not, trigger an exception before starting to execute the instruction.

- While executing the instruction, keep track of its partial effects. There are two alternative solutions regarding what to do after a fault:

Undo the partial effects before triggering the fault.

Save the partially done state, and when the instruction is restarted – just keep going.

Simple instructions cause no problems

... assuming that the CPU has been designed with CRAPE in mind.

```
LOAD  R1-R7, OFFSET(R1)
```

If the range of the data block is well contained, say, no more than one page, it makes sense to check beforehand.

Alternatively, the CPU can load the data into intermediate (internal) registers first and only move it to the target registers at the successful completion.

```
MOVE (R2)+,-(R3)
```

```
load (R2); R2++; R3--; store; R2--; R3++; fault
```

Here the CPU can keep track of the increments/decrements and undo them before triggering the fault (PDP 11/45).

Instructions that operate on large data

MVCL rd, rs

move characters long (IBM 370)

When the instruction is interrupted, the registers reflect its partial state of execution, so it will be continued when re-executed.

rd	destination address	
rd+1	destination length	
rs	source address	
rs+1	pad	source length

MOVC3 len, src, dst

move characters 3-arg (Vax 11/780)

Here, the arguments can be general, e.g., address constants. When the instruction is interrupted:

R0 = remaining count, R1 -> next src byte, R3 -> next dst byte

The PSW has a special flag == "instruction partially done". When set, the instruction takes arguments from the registers instead of using the "original" ones.

On the Intel ...

... instructions like those are implemented via prefixing, e.g.,

```
mov ecx, length
mov esi, src
mov edi,dst
repe movsb
```

The prefix (repe) means: repeat until ecx reaches zero (decrementing ecx after each execution).

Although repe movsb is assembled as a single instruction, it can be safely interrupted, as the respective registers keep track of its partial execution.

Contemporary CPUs avoid too fancy instructions in terms of the complications of their arguments (not functionality), such that the extent of changes before a fault can be contained and rendered harmless from the viewpoint of CRAPE.

Virtual memory

We have argued that swapping is useful. More, it is indispensable:

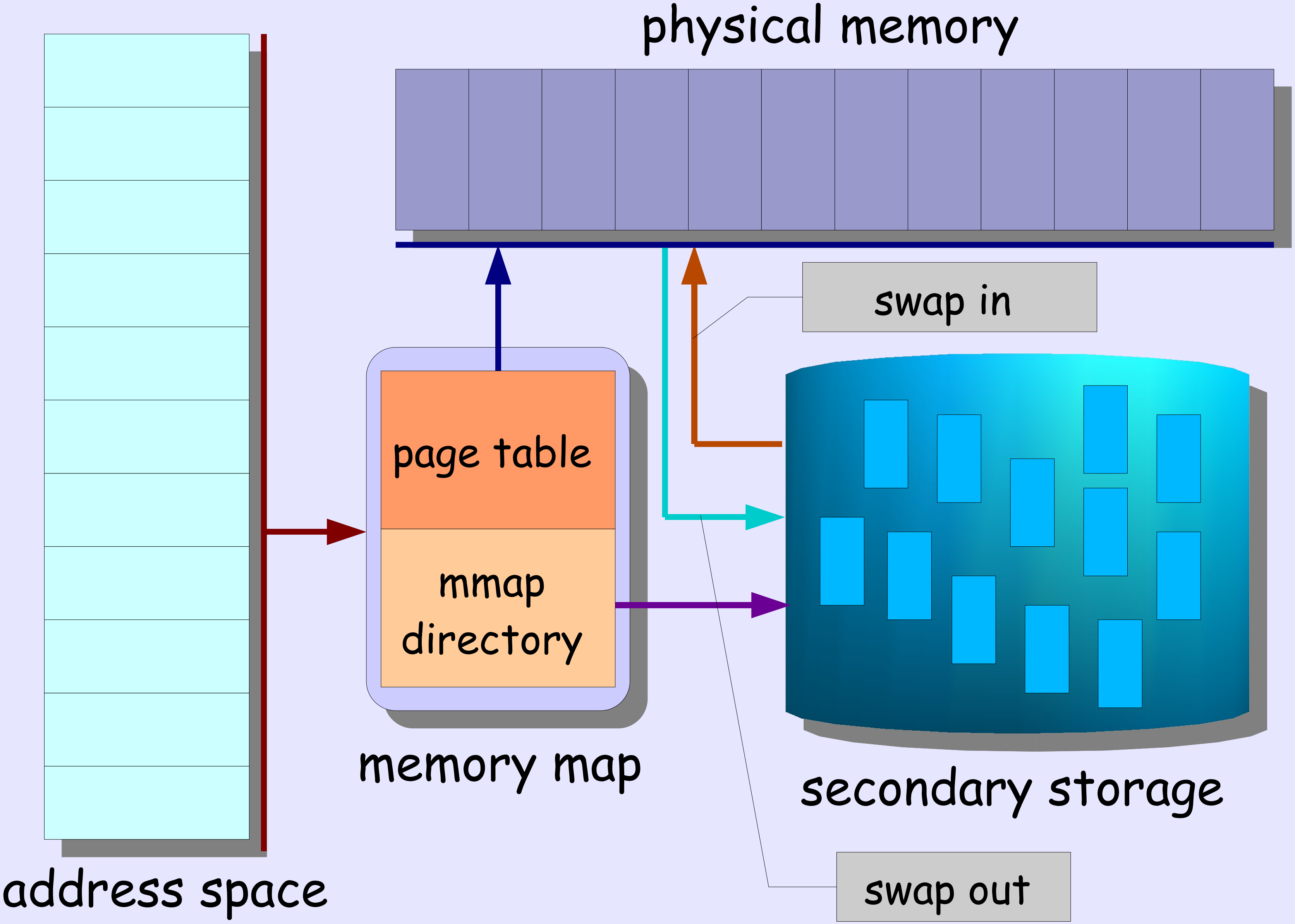
Physical memory is never tied up to the current configuration of processes in execution. Priority jobs can always be admitted.

Physical memory is made preemptible. Thus, memory-related deadlocks can be avoided.

In a partition-based system, a process can be either completely swapped out or completely swapped in. A partially swapped-in state is useless (from the viewpoint of the process).

With CRAPE, we can be more flexible, can't we? For example, lazy loading upon (implicit) demand looks like a partially swapped-out state. The process can run even though some of its stuff is not in memory.

The general picture



Swapping out

Note that the static components of the program's address space, i.e., code + constants, are already "swapped out" before the program starts. They are never going to change.

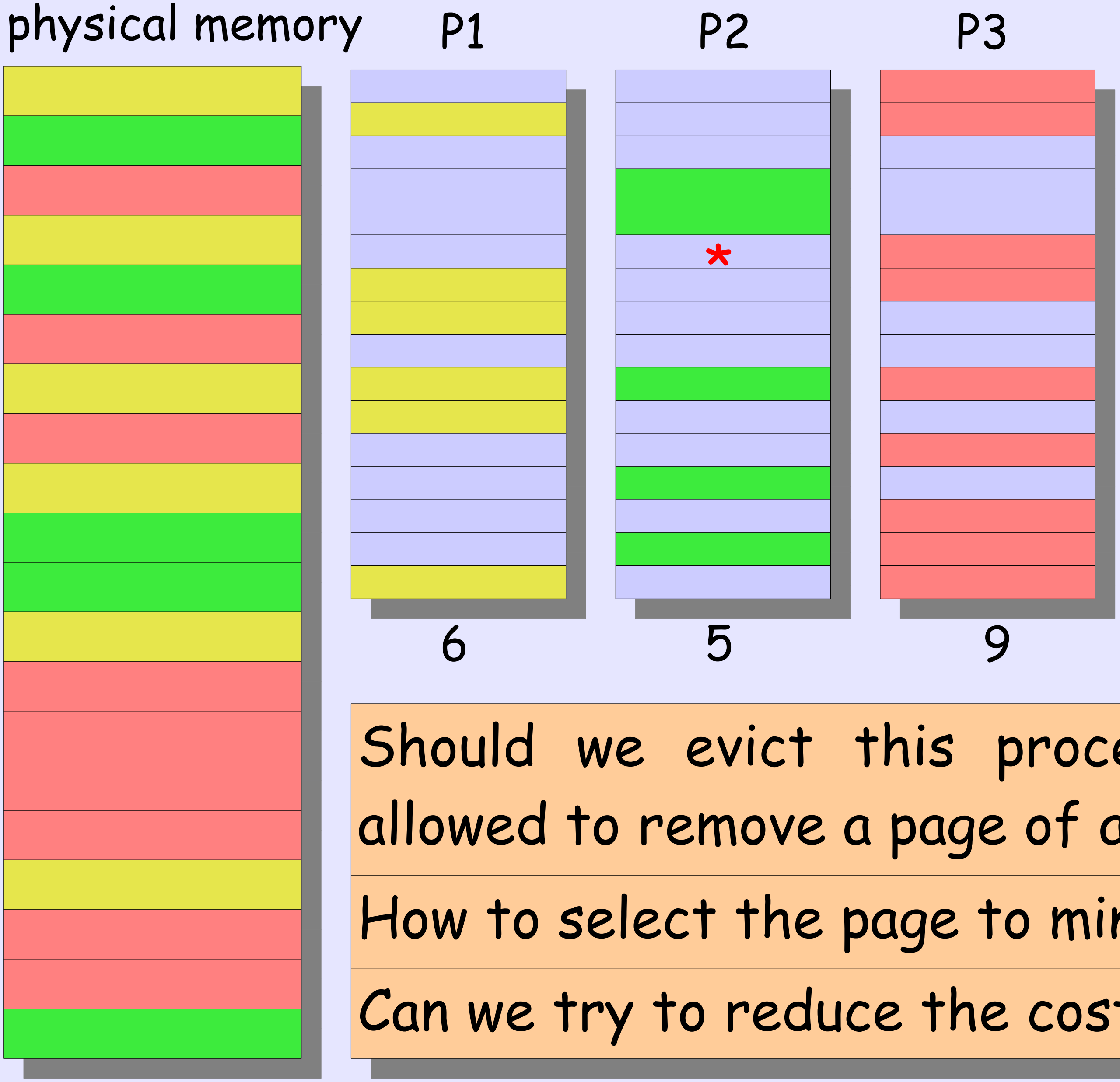
Thus, only the dynamic components, i.e., data + stack, will need a special swap area in secondary storage. Also, only these components will ever have to be physically swapped out.

The idea behind the whole scheme is this:

A partially swapped out program may still be able to run. It may not be needing some parts of its allocated memory area at this very moment.

Thus, we may be able to run more programs in whatever physical memory we have than it would seem at first sight.

Per-page decisions can be tricky



When a program demands a new page, the system may have to find room for it, i.e., decide which page (of which process) should be evicted.

Should we evict this process's page, or are we allowed to remove a page of another process?

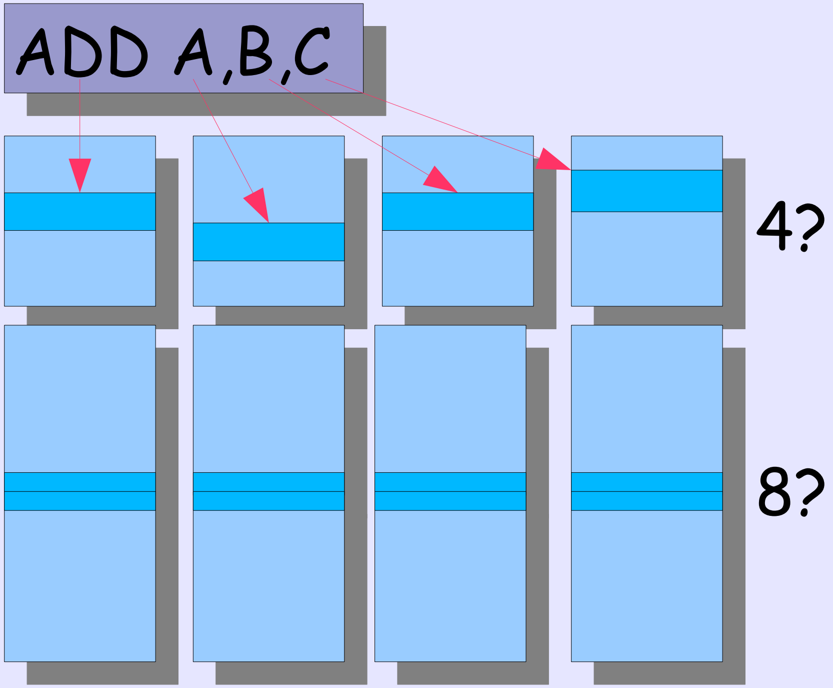
How to select the page to minimize the damage?

Can we try to reduce the cost of the eviction?

How much memory does a process need?

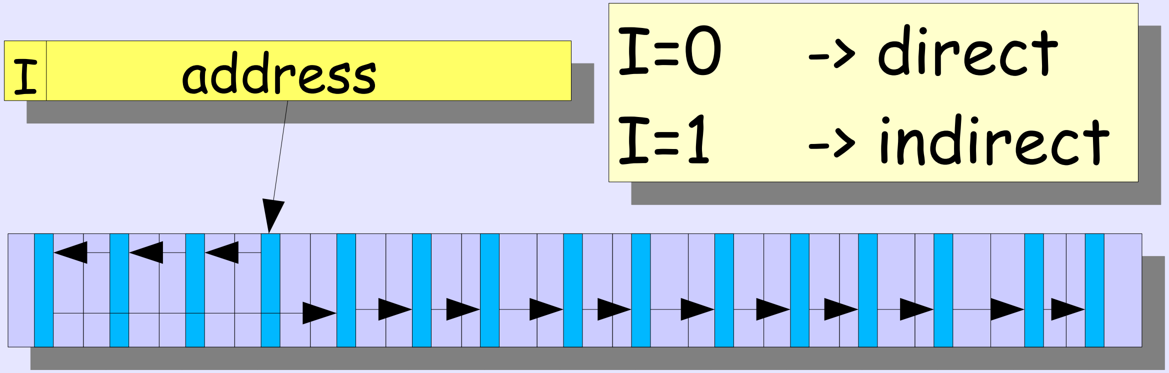
The system may come up with some idea and assume that the process will be able to live within the confines. This imposes another restriction on the degree of fanciness of machine instructions.

What is the maximum number of pages that may be needed simultaneously by the most fancy machine instruction?



ADD A,B,C

How unfriendly can it get? Indirect addressing on NOVA-3: an address could point to another address:



I=0 -> direct

I=1 -> indirect

Page replacement policies...

... are about selecting the victim page, i.e., the one that should be evicted to make room for the newly demanded page. They can be:

Global: the victim page is sought among all processes.

Local: only the pages of the faulting process are considered.

The goal: to find the page with the highest likelihood of not being immediately needed.

The obvious thing to maximize is the time during which the victim page will not be demanded back. This interval determines the page fault rate in the system, which shouldn't be too high.

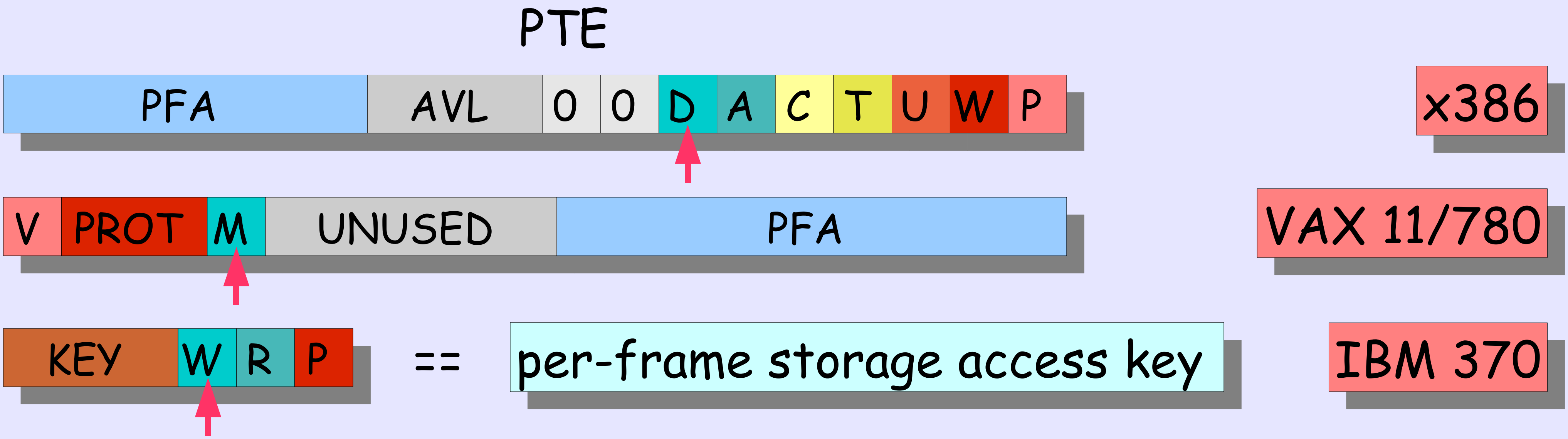
Unfortunately, no matter what we do, it is all just a guessing game. But the guess can be educated.

Suppose we have found a victim

There are two possibilities: it can be clean or dirty. In the former case, it doesn't have to be sent back to secondary storage.

How does the system know?

Some logical segments are always clean: code, constants (.rodata). But a data page can be clean as well: typically at least 80% of data references are for reading rather than writing.



PTE

PFA	AVL	0	0	D	A	C	T	U	W	P

x386

V	PROT	M	UNUSED	PFA

VAX 11/780

KEY	W	R	P

== per-frame storage access key

IBM 370

Some ideas:

Random Replacement

The victim is selected at random. This one is easy, but not very smart. So how about this one:

Random Replacement+

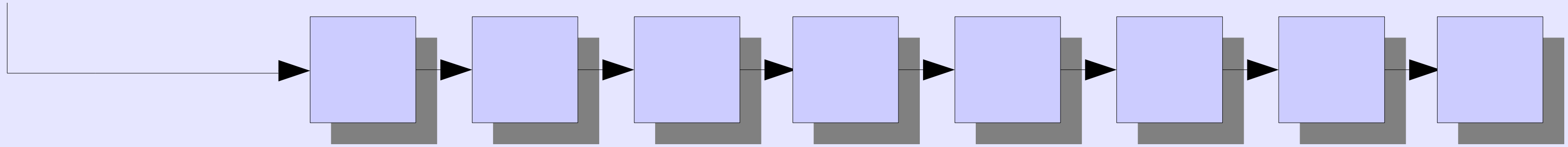
The victim is selected at random from among the clean pages. If all pages are dirty, the victim is selected at random from the dirty pages.

This kind of approach (giving priority to clean pages) is generally stupid. Some segments are always clean, so an approach like this will tend to victimize out all code/constant pages after a while.

The moral. First focus on finding a page that is likely not to be needed for a long time. Then look at its clean/dirty status. Not the other way around!

FIFO

Select the oldest page, i.e., one that has been in memory for the longest time. This one is reasonably easy to implement:



The system maintains a list of pages. Upon a fault, if a page has to be evicted,

The first page from the list is used as the victim.

A description of the new page is added at the end.

This is a cheap constant-time (O(1)) action performed upon every page fault.

The rationale makes sense: a page that has been in memory for a long time, is likely to have been explored and thus not needed.

However ...

FIFO has a painful disadvantage: it does not account for "permanently needed pages".

```
double sum;
int i;
double HugeArray [N];
    ...
sum = 0;
for (i = 0; i < N; i++)
  sum += HugeArray [i];
   ...
```

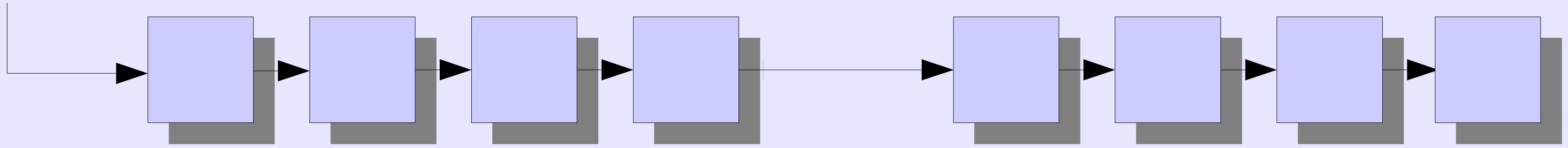
`i` and `sum` are permanently needed, even though the traversed and explored fragments of `HugeArray` may be not.

This is rather typical. In any computing problem there are data (like counters, accumulators) that are needed all the time.

Remember: old need not mean useless! Some respect for tradition is a good thing.

LRU (Least Recently Used)

Victimize the page that has not been used for the longest time.



This is not difficult conceptually. We may have a list as before,

Whenever a page is referenced, its description is removed from its current place ...

... and appended at the end.

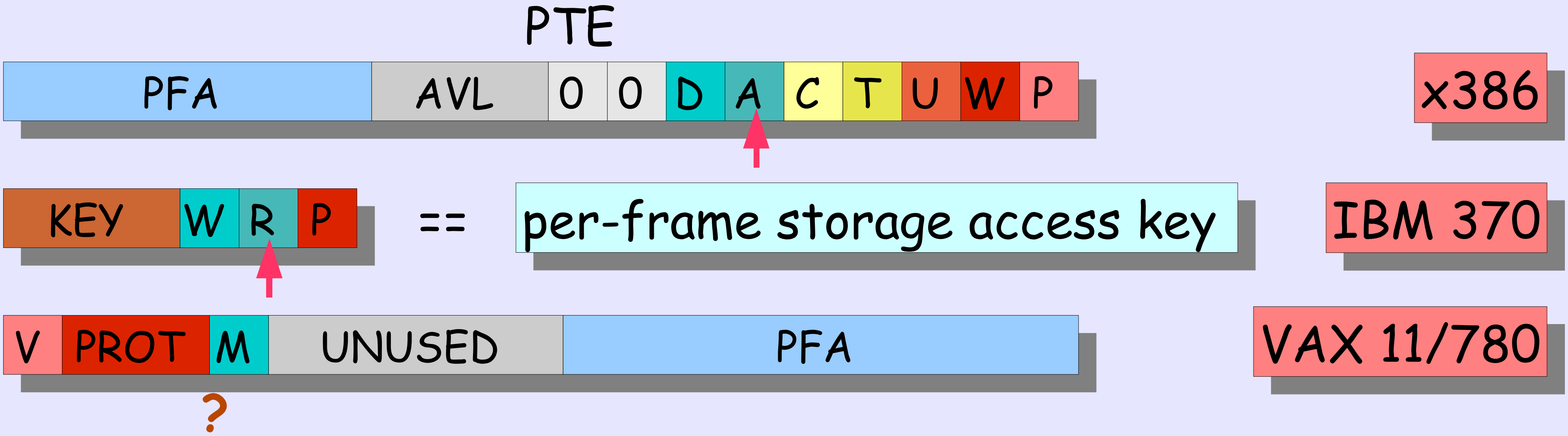
When we need a victim, the head points to the page that hasn't been used for the longest time.

Unfortunately, this calls for an expensive action to be performed after every memory reference.

This is why LRU can only be approximated.

How to approximate LRU?

The system needs a way to tell whether a page has been referenced within some time interval.



PTE

PFA	AVL	0	0	D	A	C	T	U	W	P	x386

KEY	W	R	P	==	per-frame storage access key		IBM 370

V	PROT	M	UNUSED	PFA		VAX 11/780

?

If there's no 'referenced' bit, CRAPE comes to the rescue:

Instead of clearing the referenced bit, mark the page as invalid without evicting it.

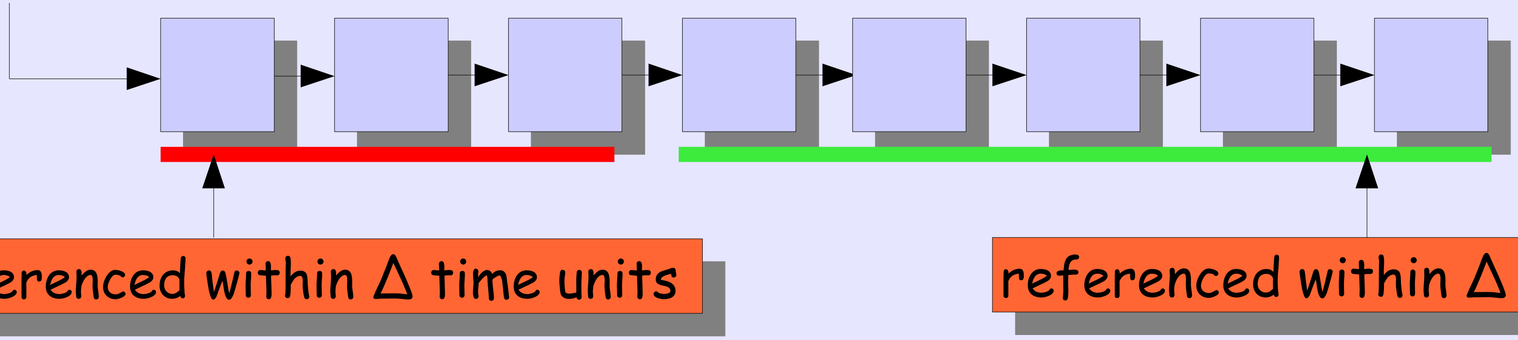
On a page fault, mark the page as valid and note the fact that it has been referenced.

Here's one idea

Remember the page list. We couldn't afford to shuffle it at every memory reference, but perhaps we can at some decent intervals.

Every fixed interval, say Δ time units, the system goes through the list.

If it finds a page whose 'R' bit is set, it moves the page to the end and clears the 'R' bit.



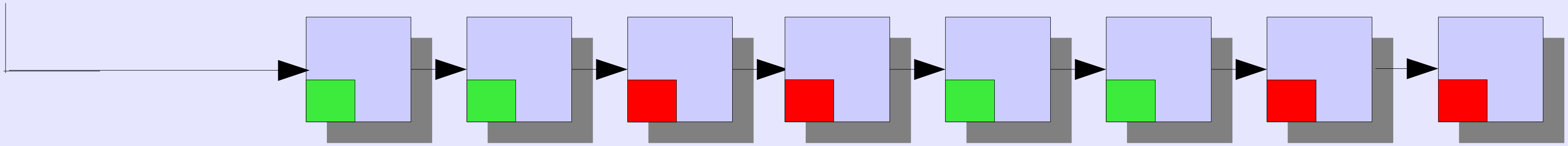
not referenced within Δ time units

referenced within Δ time units

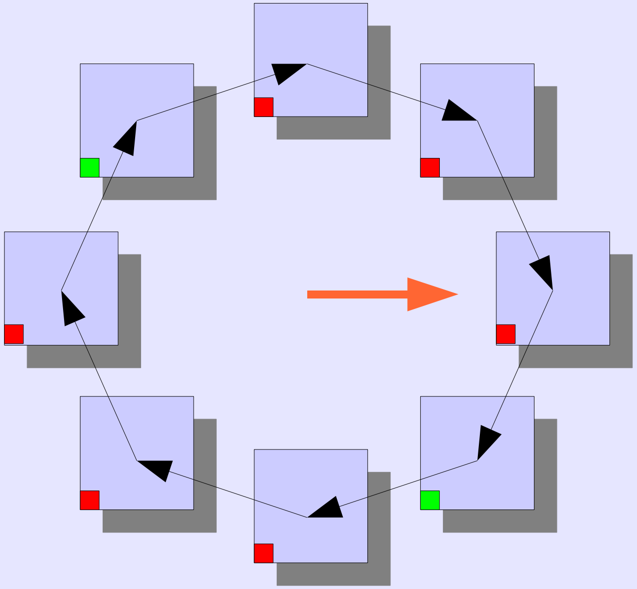
This looks like some approximation, but it isn't very smart. What if the last page gets referenced after having been checked and just before the fault? We can do better.

Second chance

It starts like FIFO. The pages are put into a list:



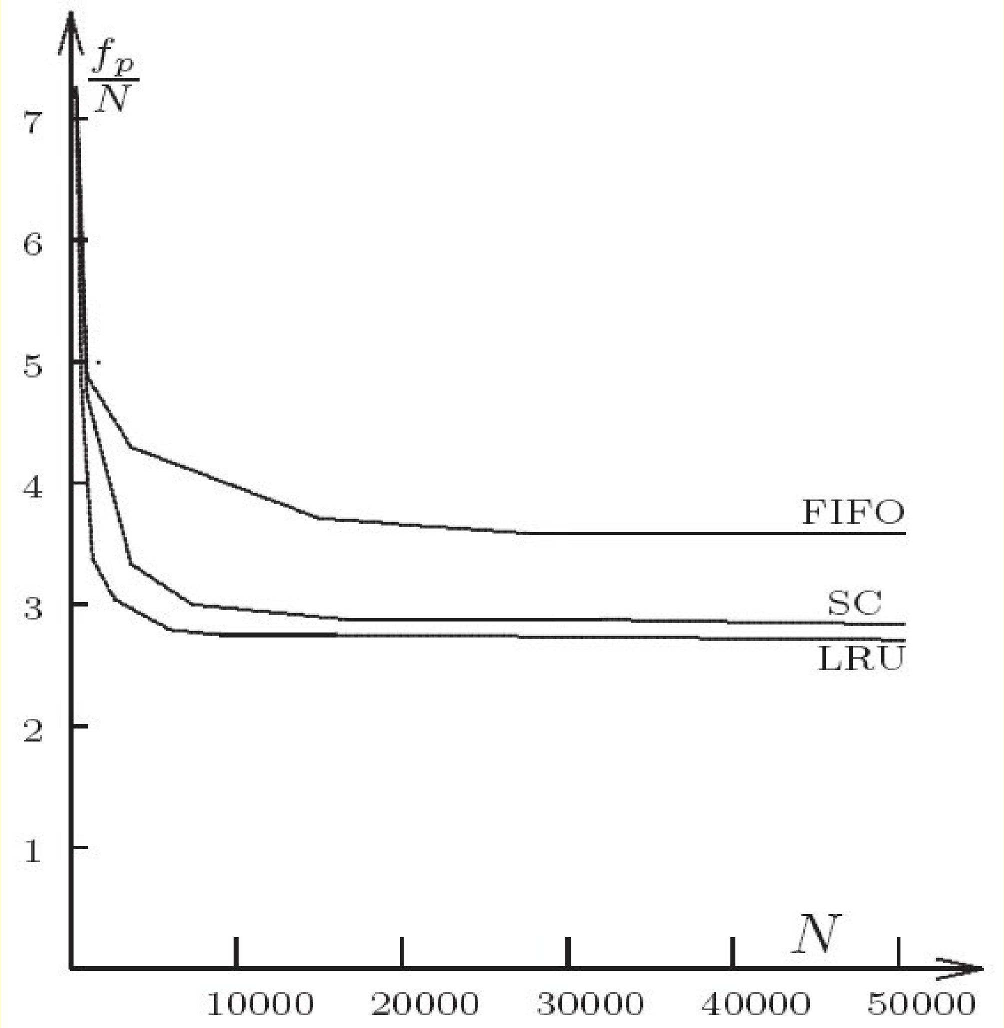
The twist: if the 'R' bit of the victim candidate is 1, the candidate is moved to the end of the list, its 'R' bit cleared, and the next page from the list is tried.



The page to be victimized (by FIFO) gets a second chance to prove its relevance.

Note that, in contrast to pure FIFO, second chance will not victimize permanently needed pages.

Performance



Heapsort executed in 25% of required memory.

Second chance is a very good (especially for its cheap price) approximation of LRU.

In some variations, it is the most popular policy being used in actual systems.

However, the policy is not all. With multiprogramming, it is also important how to divide the available physical frames among multiple processes.

Some theory

A memory reference string of a program is the list of memory locations consecutively referenced by that program, e.g.,

0x80034224	0x80034
0x80034228	0x80034
0x8003422a	0x80034
0x8003422c	0x80034
0x80039910	0x80039
0x80039922	0x80039
0x80039923	0x80039
0x80039924	0x80039
0x80039925	0x80039
0x8006fffc	0x8006f
0x80070000	0x80070
0x80070004	0x80070
0x80070008	0x80070
0x8007100c	0x80071
0x8003d010	0x8003d
0x8003d014	0x8003d
0x80034017	0x80034
0x80034018	0x80034
0x80034019	0x80034
0x800382ae	0x80038
0x80070000	0x80070
0x8006fffc	0x8006f
0x8006fffa	0x8006f

A page reference string is the list of pages consecutively referenced by a program, with subsequent references to the same page merged into one.

```
0x80034
0x80039
0x8006f
0x80070
0x80071
0x8003d
0x80034
0x80038
0x80070
0x8006f
```

Then, the actual page numbers are irrelevant, in particular, all these page reference strings are equivalent:

34	39	6f	70	71	3d	34	38	70	6f
1	2	3	4	5	6	1	7	4	3
99	98	97	96	95	94	99	93	06	97

Monotonic strategies

Suppose that the system has admitted a program for execution. It says: You run with a local replacement strategy. Here is your real memory, k frames total. This is how much I can afford you at the moment. Needless to say, I don't care how grandiose ideas you may have regarding the size of you virtual memory.

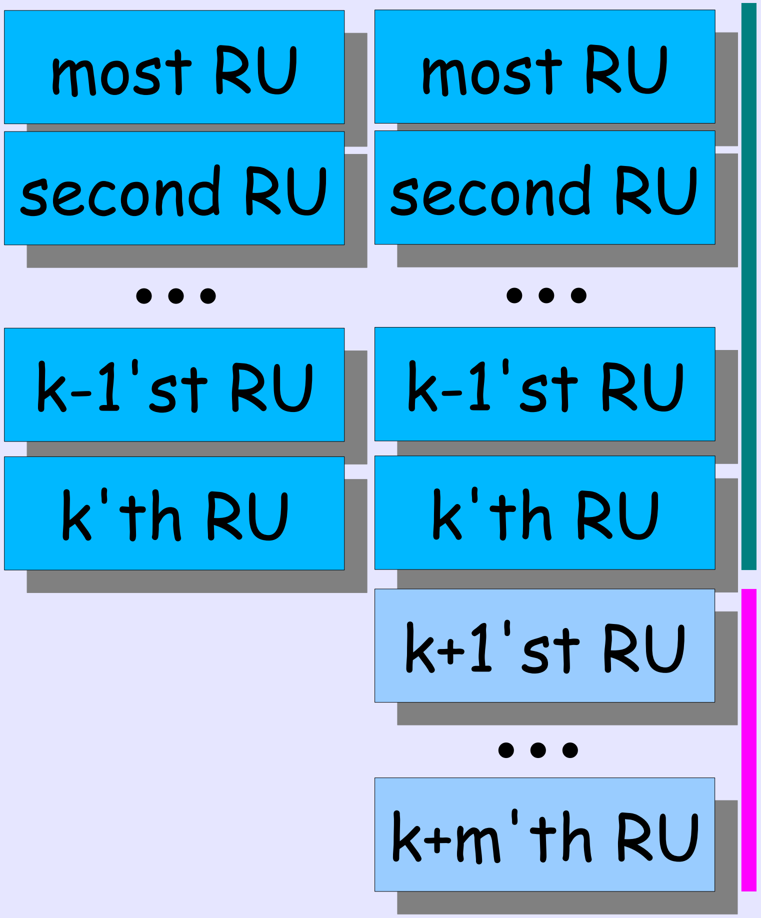
The program runs and generates faults. The system sees that there are too many of those for comfort.

So it says: OK, OK, I have managed to scramble a few more frames. So now I will give you a bit more. I hope this will make your life easier.

It is natural to assume that more physical memory (for the same program) will translate into fewer page faults.

Definition

A strategy is monotonic, if no program (under a local variant of the strategy) will ever generate more faults when run in more physical memory.



most RU

second RU

•••

k-1'st RU

k'th RU

most RU

second RU

•••

k-1'st RU

k'th RU

k+1'st RU

•••

k+m'th RU

Example: LRU is monotonic.

This is because the set of k+m most recently used pages includes k most recently used pages.

A program running in k+m frames cannot generate more faults then when run in k frames: at any time, it has the same frames as before + something extra.

Belady's anomaly

Consider FIFO and this page reference string:

3 frames

	1	2	3	4	1	2	5	1	2	3	4	5
	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟩	🟩	🟥	🟥	🟩
-	1	2	3	4	1	2	5	5	5	3	4	4
-	-	1	2	3	4	1	2	2	2	5	3	3
-	-	-	1	2	3	4	1	1	1	2	5	5

4 frames

	🟥	🟥	🟥	🟥	🟩	🟩	🟥	🟥	🟥	🟥	🟥	🟥
-	1	2	3	4	4	4	5	1	2	3	4	5
-	-	1	2	3	3	3	4	5	1	2	3	4
-	-	-	1	2	2	2	3	4	5	1	2	3
-	-	-	-	1	1	1	2	3	4	5	1	2

Thus, FIFO is not monotonic!

A page fault can mean:

0. An error (an actual illegal memory reference)

1. A CRAPE trick that only requires some change of flags, etc.

2. Request for a brand new page, e.g., the stack has grown a bit.

3. Request for a static page (code) that must be fetched from the executable file (or a mmap'ped file).

4. Request for a data page that was once swapped out and is now demanded back.

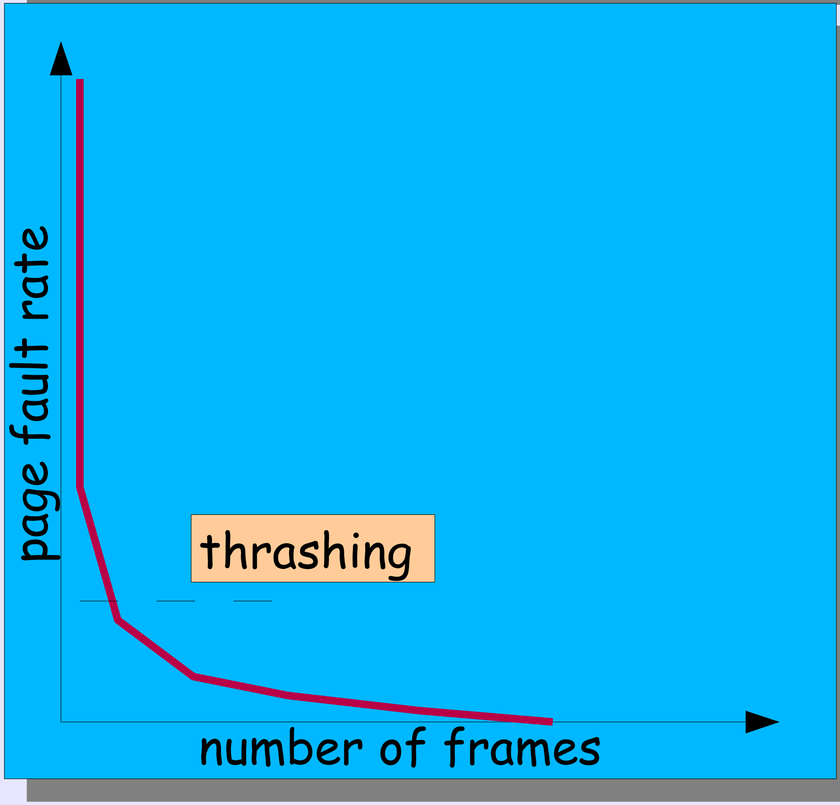
1. No need to do any page transfers at all => no blocking.

2, 3, 4. A frame may have to be evicted. This may require sending it back to the swap area => possible blocking.

3,4. A page has to be brought in => mandatory blocking. Possibly one more I/O transfer to send the evicted page back to the swap area.

How much memory?

Most programs need very little physical memory to run efficiently, compared to their total memory used over the entire lifetime.



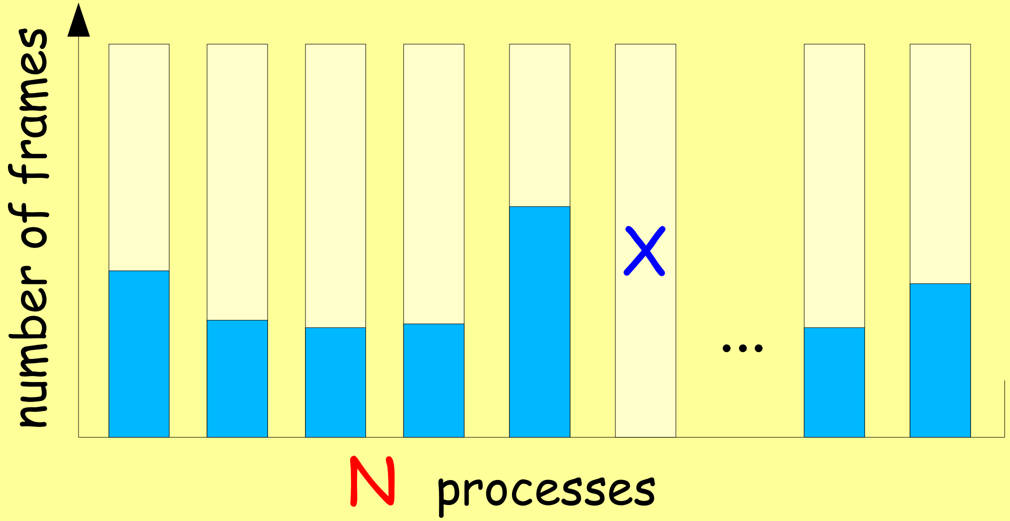
The phenomenon captured in curves like this is called locality.

The sharp upturn at the vertical axis is called thrashing.

Thrashing means that a program's performance does not degrade gracefully as the amount of physical memory available to it is reduced.

In other words, the partitioning of memory among multiple programs in execution is not a trivial issue. The problem is that different programs exhibit different degrees of locality.

The optimization problem



The system would like to fill its entire physical memory with processes (as much as possible, anyway).

What are the objectives?

Utilization: memory -> 100% , CPU -> 100% , paging device -> 100%

No program thrashes

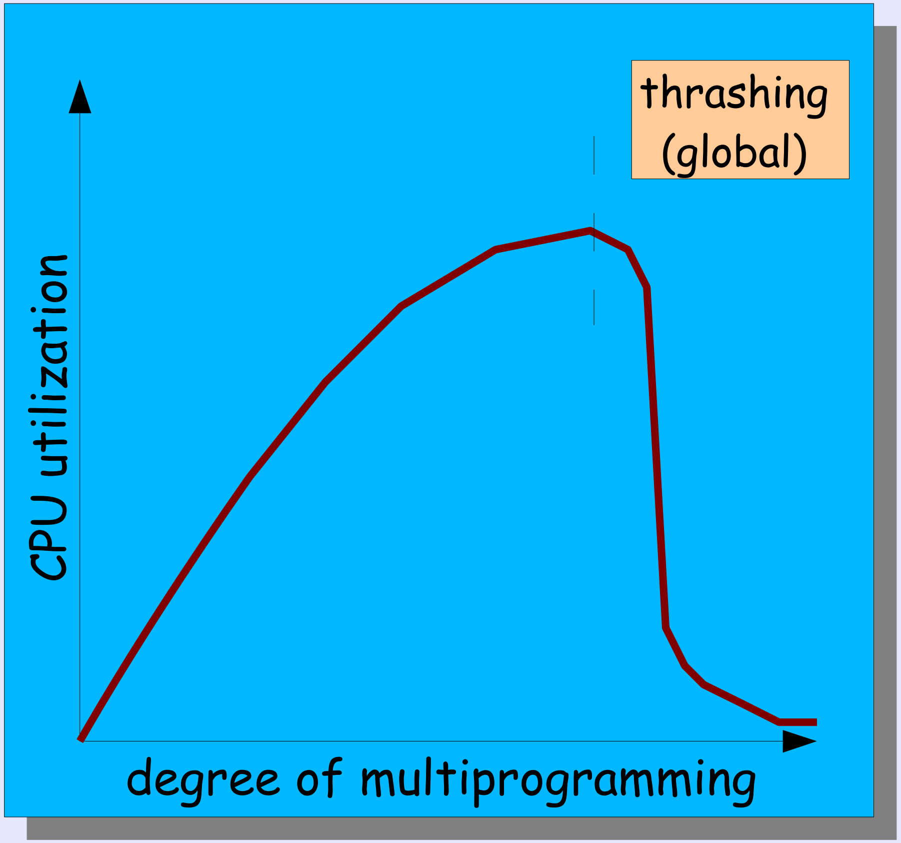
No program has more than it needs

In other words “fairness and neatness”: all programs generate page faults close to the same “optimum” rate f_{opt}.

Global vs. local thrashing

If one program thrashes while others run (perhaps more than) comfortably, then we have unfair memory allocation.

Sometimes programs thrash no matter how you try.



This means that there are simply too many of them.

Note that a program may start small and grow, i.e., its locality pattern can change for the worse.

In such a situation, complete swapping out is the only way to eliminate global thrashing.

Optimum page fault frequency

The paging device has some “average” processing time of a page fault, say t_f. Note that different faults may have different actual processing times. The predominant component is the time to carry out a disk I/O transfer.

Let f denote the observed page fault frequency. Note that it is physically impossible for f to be steadily greater than $1/\ t_f$. This is because the number of processes and the queue to the paging device are finite.

The paging device's view (it wants to be 100% busy) is that f_{opt} should actually be close to $1/\ t_f$. Now, t_f is essentially equal to the cost of a disk transfer (occasionally two), thus:

$$f_{opt} \approx \frac{1}{t_{io} \times (1 + \alpha)}$$

where t_{io} is the average time to perform a disk transfer, and α is the dirty factor.

So how much memory?

Assume that time is measured in memory references, why not!?

$$\Delta = \frac{1}{f_{opt}}$$

is the number of memory references that a program should be allowed to carry out successfully before it hits a page fault.

This gives us a recipe for memory allocation:

Each program should have exactly as many frames as to be able to perform exactlyΔ memory references before hitting a page fault.

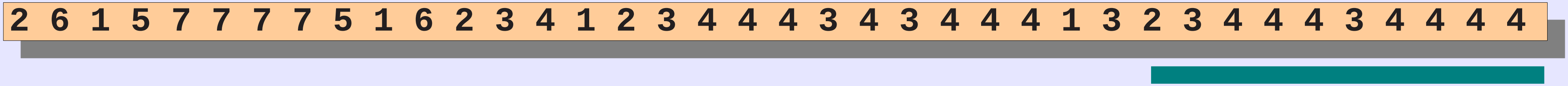
Needless to say, this is all wishful thinking (the future is unpredictable), but at least we can formally define the target.

And, of course, the best we can do is to monitor the program's past behavior and quietly assume that whatever the program is doing, it is likely to be doing that for a while.

The working set model

The working set of a given program at time ***t***, denoted ***W(t,Δ)***, is defined as the collection of pages referenced by the program within the last Δ memory references preceding ***t***.

Assume Δ = 10



2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4 3 4 4 4 4

2 3 4

Of course, Δ is always much higher than 10. For example, a typical average disk access/transfer time is, say, 20 msec, so let us suppose that Δ is 35 msec. This may translate into well over 1M memory references on a contemporary CPU.

A simple (?) solution:

Allocate to each process its working set (and nothing more).

Let N be the number of processes in execution, M be the total amount of physical memory available to processes, and $|W_i\ (t,\Delta)|$ denote the size of the working set of process P_i at time t .

$$D_t = \sum_{i=0}^{N-1} |W_i(t,\Delta)|$$

gives the total demand for frames at time t.

$D_t < M$

the system may be able to admit a new program for execution

$D_t > M$

the system cannot fulfill the demands of the present processes and may have to consider a complete swap-out of some.

Unfortunately ...

... there is no way for the system to know exactly the working set size of a process.

Even if the system could magically know that size, it wouldn't make sense to reconsider program admission or swapping-out after every memory reference.

Thus, this is all just theory. But it *gives* us ideas and hints for a practical implementation.

For example, the system can monitor the page fault frequency and respond to its fluctuations.

$f > f_{opt} + \epsilon$ add more memory to the program

$f < f_{opt} - \epsilon$ take some memory away from the program

In any case, it makes no sense to be jumpy with drastic decisions.

The triple point...

CPU 100% / paging device 100% / memory 100%

... is the Holy Grail of computer performance – it may be difficult to reach. But a serious "computing system" (not too many of those these days) should make it its goal. If you cannot come close,

- → you may not have enough jobs to run
- → your system may be misconfigured

Note that generally it doesn't make sense to buy:

- → more memory than you need
- → a faster CPU than you need
- → a faster (and larger) disk than you need

Remember this! Don't throw your money away!

How it all looks in real life

The theory is OK, but its practical embodiments tend to be somewhat confusing to the uninitiated. Here are the reasons:

Pages from processes' address spaces are not the only type of stuff stored in memory (file cache is another important stuff).

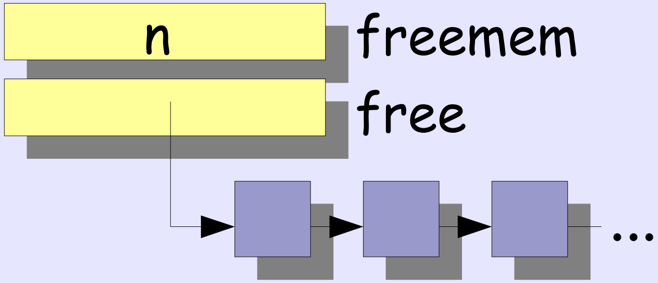
It makes sense to separate reclaiming free memory from servicing page faults. Why?

Creative strategies of optimizing disk i/o operations (e.g., read ahead) add finesse to our options.

We shall start from an overview of some simple generic (but realistic) approach, which corresponds roughly to old BSD UNIX, and then comment on a few more modern refinements.

An example

The system maintains a list of free pages:



The page fault service function doesn't execute any replacement policy:

```
page *pagein () {
   while (1) {
       mask ();
       if (free != NULL)
          break;
       waitforevent (&freemem);
       unmask ();
       schedule ();
    }
    p = free;
    free = p->next;
    freemem--;
    unmask ();
    return p;
}
```

The system tries hard to avoid a situation when free == NULL. This is never meant to happen: the check is just a precaution.

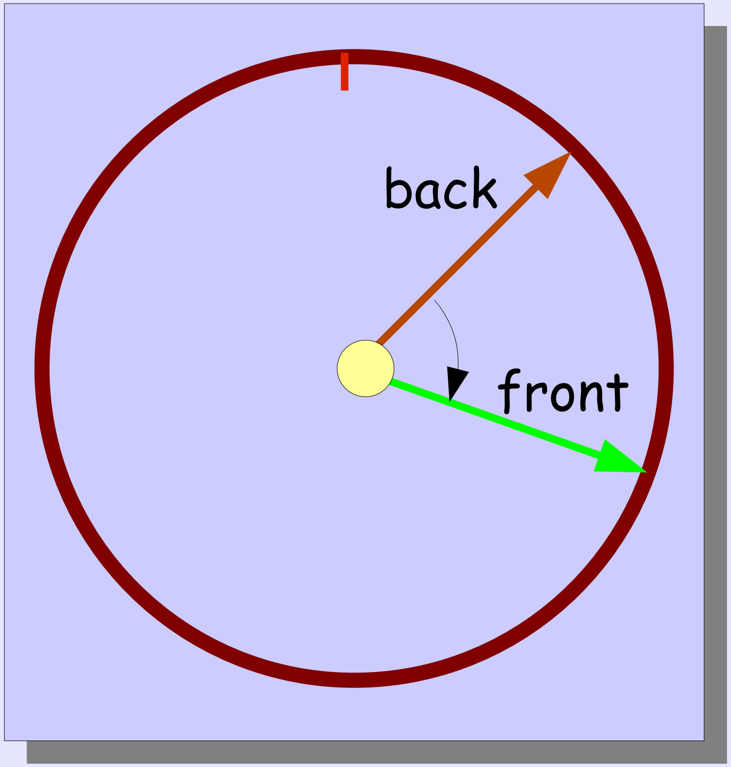
The free list is maintained by a special system process called page daemon, which makes sure that the list never shrinks below a certain minimum.

The page daemon

```
void pagedaemon () {
    while (1) {
       if (freemem >= lotsfree) {
          delay (some_standard_longish_interval);
          continue;
       }

       if (freemem > desperate || !try_to_swap_out ())
          victimize_some_page ();

       delay (victimize_interval (freemem – desperate));
    }
```

Conceptually, the shortest victimize_interval (which gets shorter as freemem gets closer to desperate) corresponds to f_{opt}. If free memory continues to shrink at this interval, it means thrashing.

The victimizer



The front hand clears the “referenced” bits, and the back hand looks at them again. If the back hand finds a page with the referenced bit cleared, that page is victimized.

Originally, there was a single hand, and the strategy looked even closer to second chance.

Note that the victimizer only starts when freemem < lotsfree, which means that it may start in an old stale configuration. Thus, the system should refrain from victimizing any pages until the back hand looks at something that has been actually given a (second) chance.

A more involved scheme (Linux)

Three lists of pages implementing an approximate LRU order:

active_list:	page->age >= 0
inactive_dirty_list:	page->age == 0
inactive_clean_list:	page->age == 0

page->age is a confusing name. It is more like relevance. page->age == 0 means "not needed".

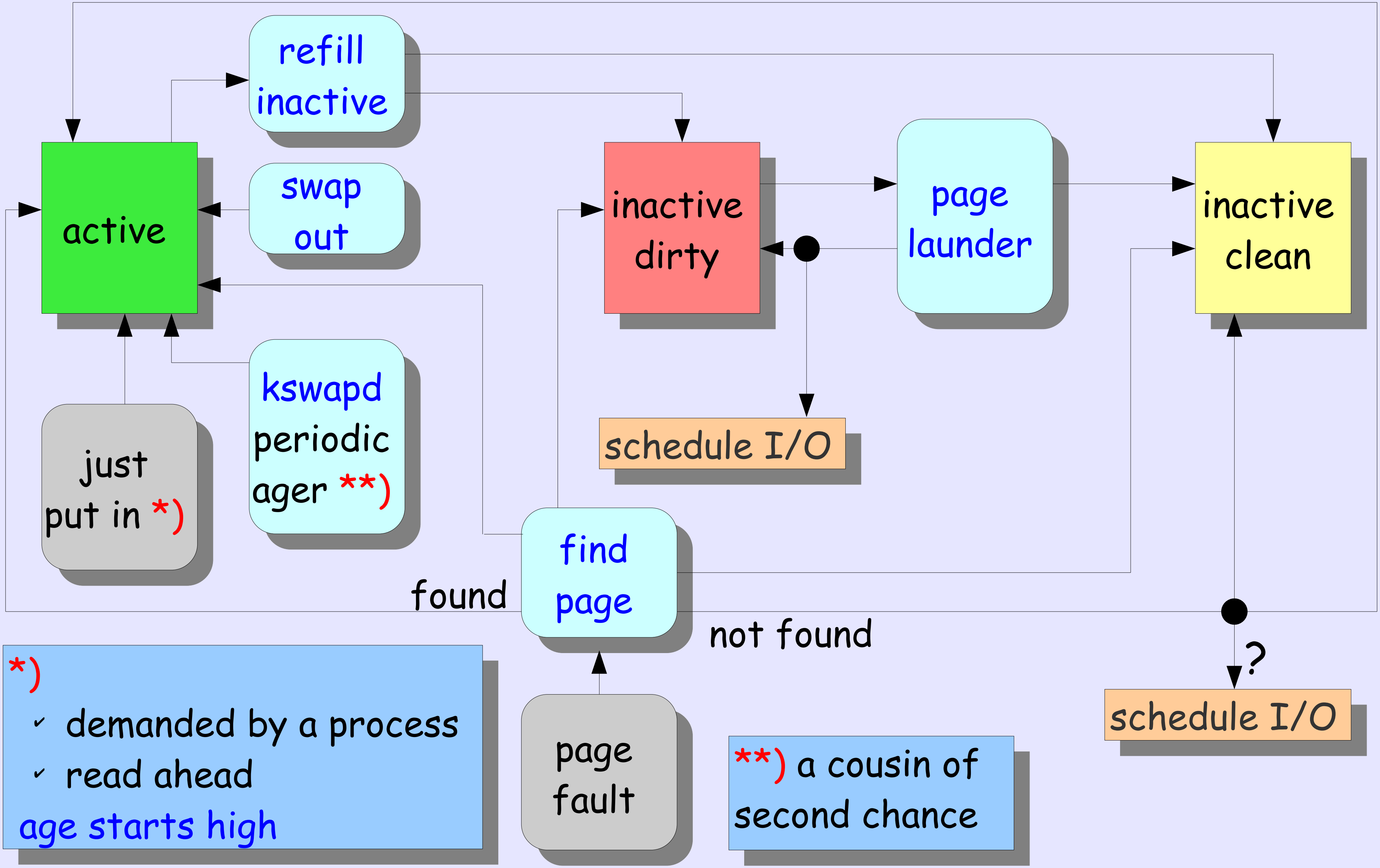
active_list is for pages appearing "recent". Typically, an active page is mapped into a process's address space, but not necessarily.

inactive (dirty or clean) means "not mapped into any process's address space."

inactive_dirty does not imply that the page is in fact dirty. It only means that the page was dirty when put into the list.

inactive_clean does mean that the page is clean. This part is easy.

Linux page cache



*)

- demanded by a process
- read ahead

age starts high

**) a cousin of second chance

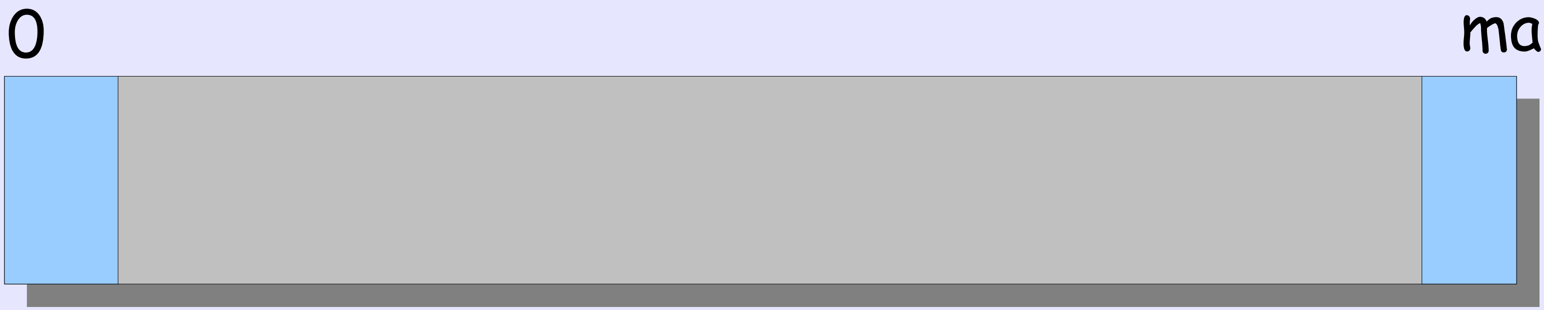
A few closing remarks

Really huge logical address spaces, e.g., 64-bit addressing. How to implement page tables for them?

Multilevel tables? How many levels?

Exercise:

Let A be the number of address bits (e.g., 64), let l be the number of bits in a page offset (e.g., 12), let k be the number of levels. What is the overhead on describing an "extremely hollow" address space like this?



A few simple calculations

The number of address bits per level is: $u=(A-l)/k$

The number of entries in a table is thus: $2^u=2^{(A-l)/k}$

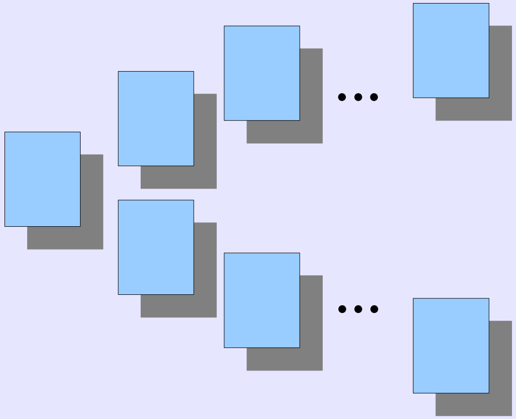
The size of a table is: $E\times 2^u$

where ***E*** is the size of one entry (e.g., 8 for 64-bit addresses)

The total number of all tables is : $1+2\times(k-1)$

The overhead:

$$O=E\times(1+2\times(k-1))\times 2^{(A-l)/k}$$



Some examples

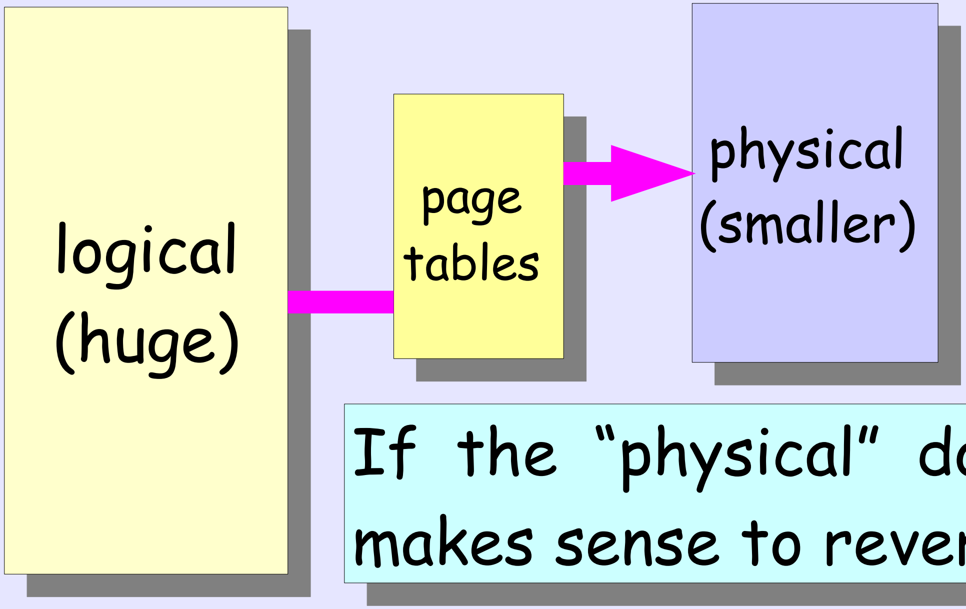
$A=64$, $l=12$, $E=8$	
k	*overhead*
1	3.6×10^{16}
2	1.6×10^{9}
4	470K
6	35K
8	10K
10	5K
12	3.5K

$A=32$, $l=12$, $E=4$	
k	*overhead*
1	4.2×10^{6}
2	12K
4	896

$A=64$, $l=14$, $E=8$	
k	*overhead*
5	72K
4	320K
6	28K

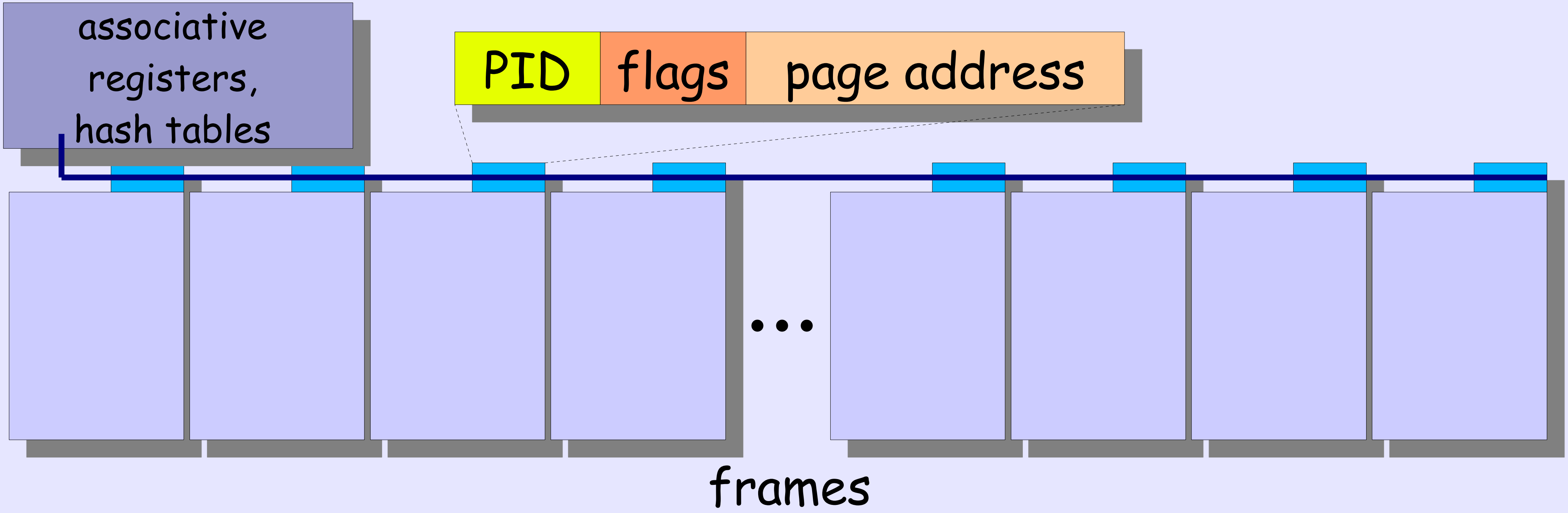
Of course, not all possibilities are realistic, but the trade-off is clear. Now, would you really want to have 5-level tables?

Inverted page tables



Page tables are large because the domain is large: they have to account somehow for every page from a large population.

If the “physical” domain is smaller, then perhaps it makes sense to reverse the mapping.



No tables at all?

The worst problem with inverted page tables is that they make memory sharing difficult.

Remember TLB. It resolves the vast majority of all references without resorting to the tables at all. So this is the idea:

Make TLB misses perceptible by the kernel (as correctable faults), and let the system take care of filling in the entries.

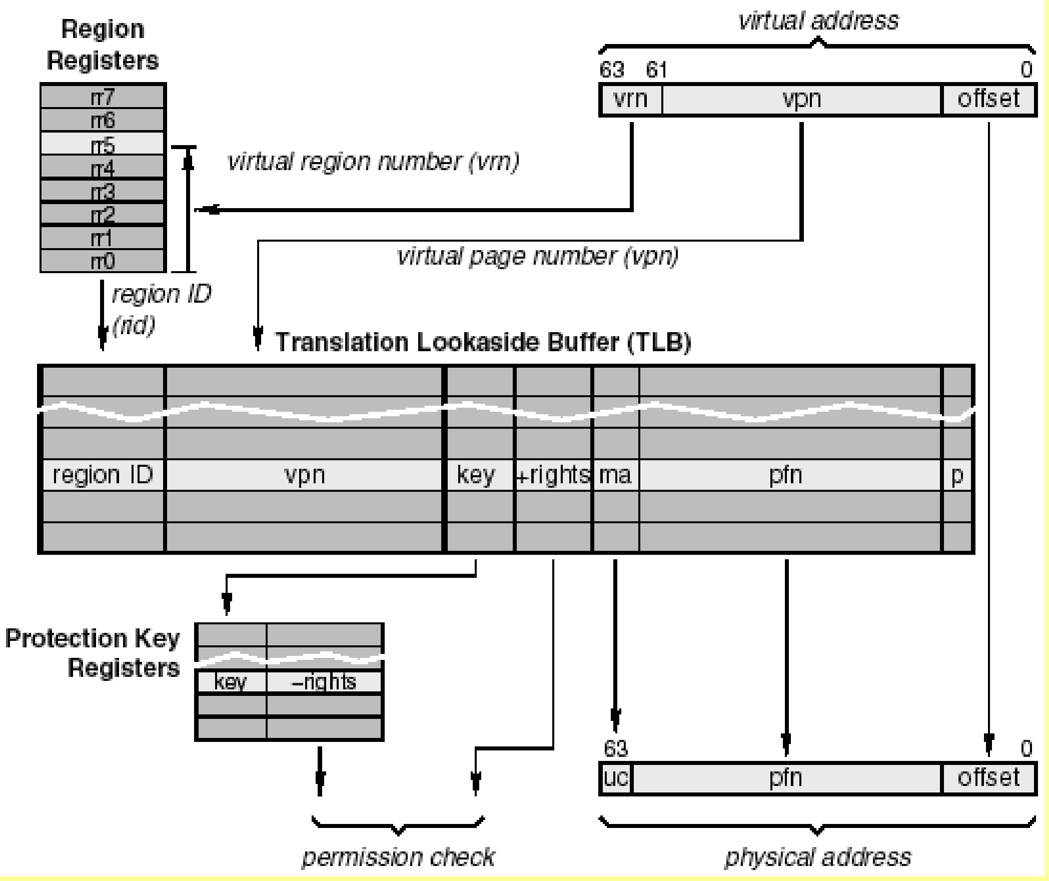
The system can implement the mapping the way it pleases. Whatever data structures it uses, they are not tied up to any mapping hardware, so they can be more sophisticated and compact e.g., hash tables, dictionaries.

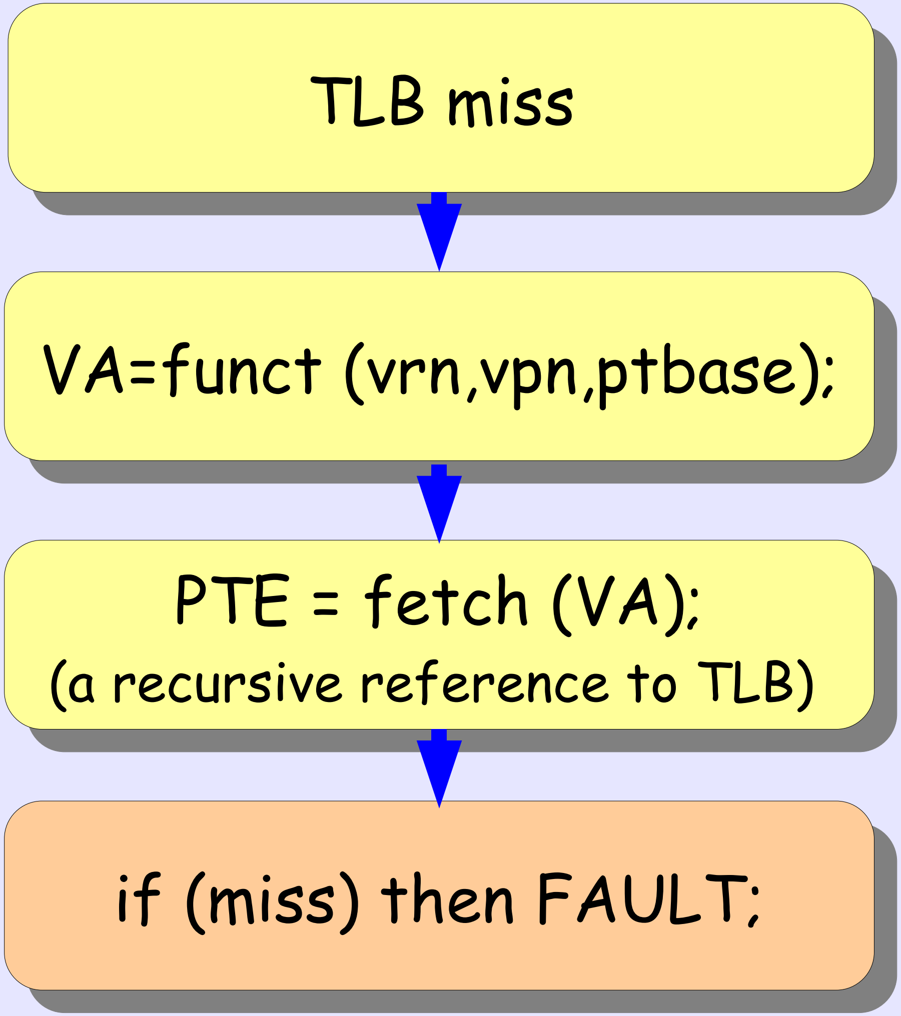
The cost of forcing the hardware to go through five levels of tables may be comparable to the cost of doing something smarter entirely in software.

Sometimes hardware can assist in that

VHPT walker on IA-64: optional hardware assist for implementing virtual page tables.

TLB on IA-64




TLB miss

VA=funct (vrn,vpn,ptbase);

PTE = fetch (VA);
(a recursive reference to TLB)

if (miss) then FAULT;

Superpages

Some contemporary architectures allow you (as an option) to have pages of variable size. This means that TLB entries can describe physical memory chunks of different sizes, e.g.,

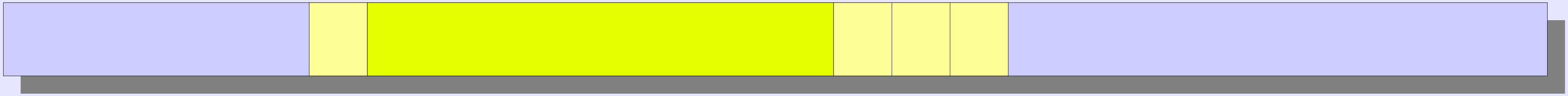
PFRA	SID	LEN	FLAGS

A large continuous range of physical memory can be described with much fewer entries. This means:

reduced size of tables

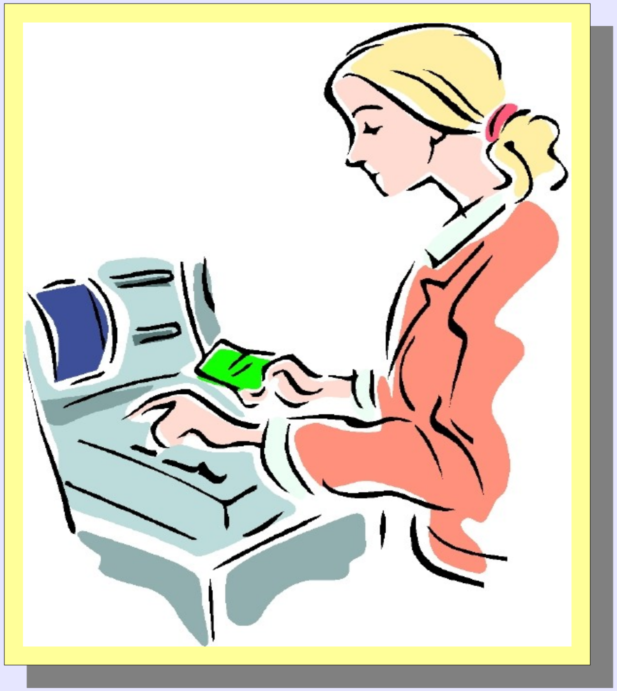
more TLB hits

More burden for the OS: promoting/demoting pages, fragmentation.



Scheduling

customers



server

Jobs (processes)	Admission	The system
CPU bursts	CPU scheduler	CPU
Processes	Swapper	CPU queue
I/O requests	Disk scheduler	Disk

Basic performance criteria

Customer

Turnaround time. How much time it takes on the average to handle a customer. How long am I going to wait?

Server

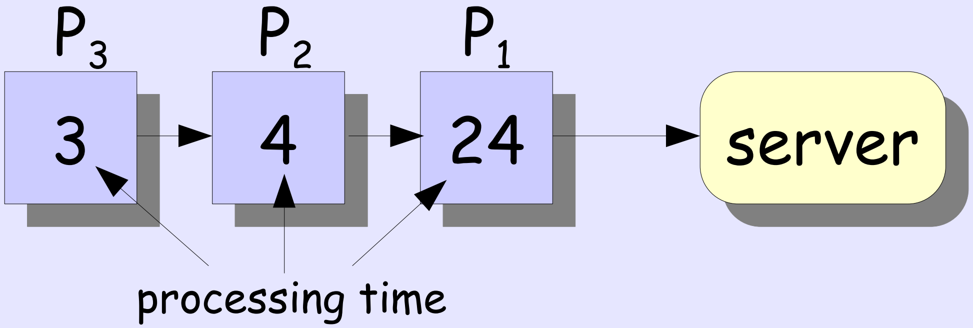
Throughput. How many customers can I handle per time unit?

Customer satisfaction? Maximize the number of customers that are happy with my service.

Turnaround time and throughput are always well defined (numerically). The measure of customer satisfaction depends on what the customer specifically needs.

Throughput is often uninteresting because the service is work-conserving, i.e., the ordering of customers doesn't affect their service time.

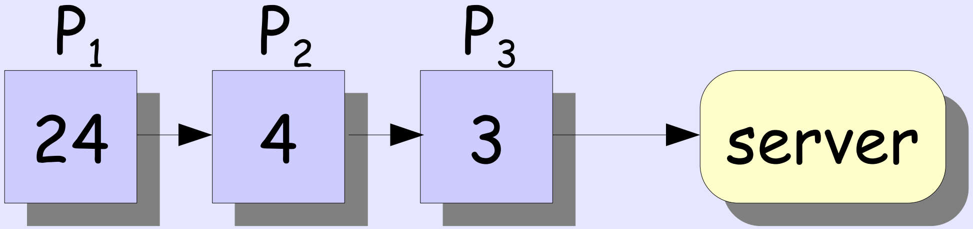
SJF vs. FCFS



Turnaround time:

P1	24
P2	28
P3	31

Average:
(24+28+31/3)
= 27.67



Turnaround time:

P1	31
P2	7
P3	3

Average:
(31+7+3/3)
= 13.67

SJF: Shortest Job First. Give priority to the customers with the shortest processing time. This policy guarantees the shortest possible average turnaround time.

Note that throughput is the same in both cases: 3/31 .

In OS, we talk about ...

Long-term scheduling. This is about admitting jobs for execution. Decisions made once per job's lifetime. Relevant for batch systems.

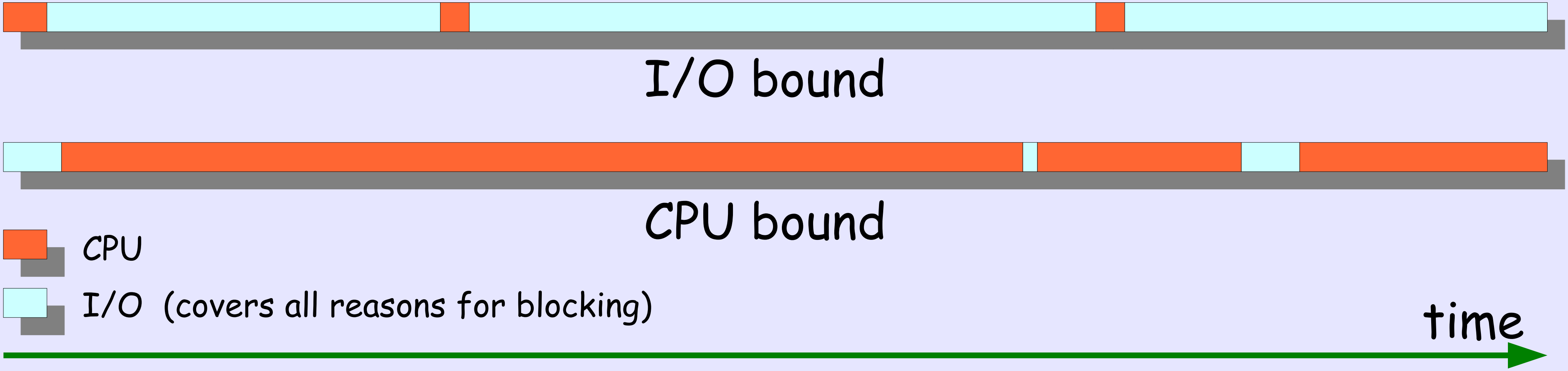
Intermediate-term. Swapping in/out. May be done several times per job's lifetime, not more often than every few seconds.

Short-term. This is CPU scheduling. Zillions times per process's lifetime, may be hundreds of times per second.

General criteria are similar. Remember: in the absence of a better description of the client's preference, it always makes sense to minimize the turnaround time – if only to avoid ...

... the convoy effect:

Short term scheduling



CPU bursts are the customers. We care about the turnaround time of CPU bursts.

SJF says that processes with short CPU bursts (i.e., I/O bound processes) should be given priority in acquiring the CPU.

Easy to say. How can we know what the duration of a process's next CPU burst is going to be?

The system may try to guess

Extrapolating the duration of the next CPU burst from the past history.

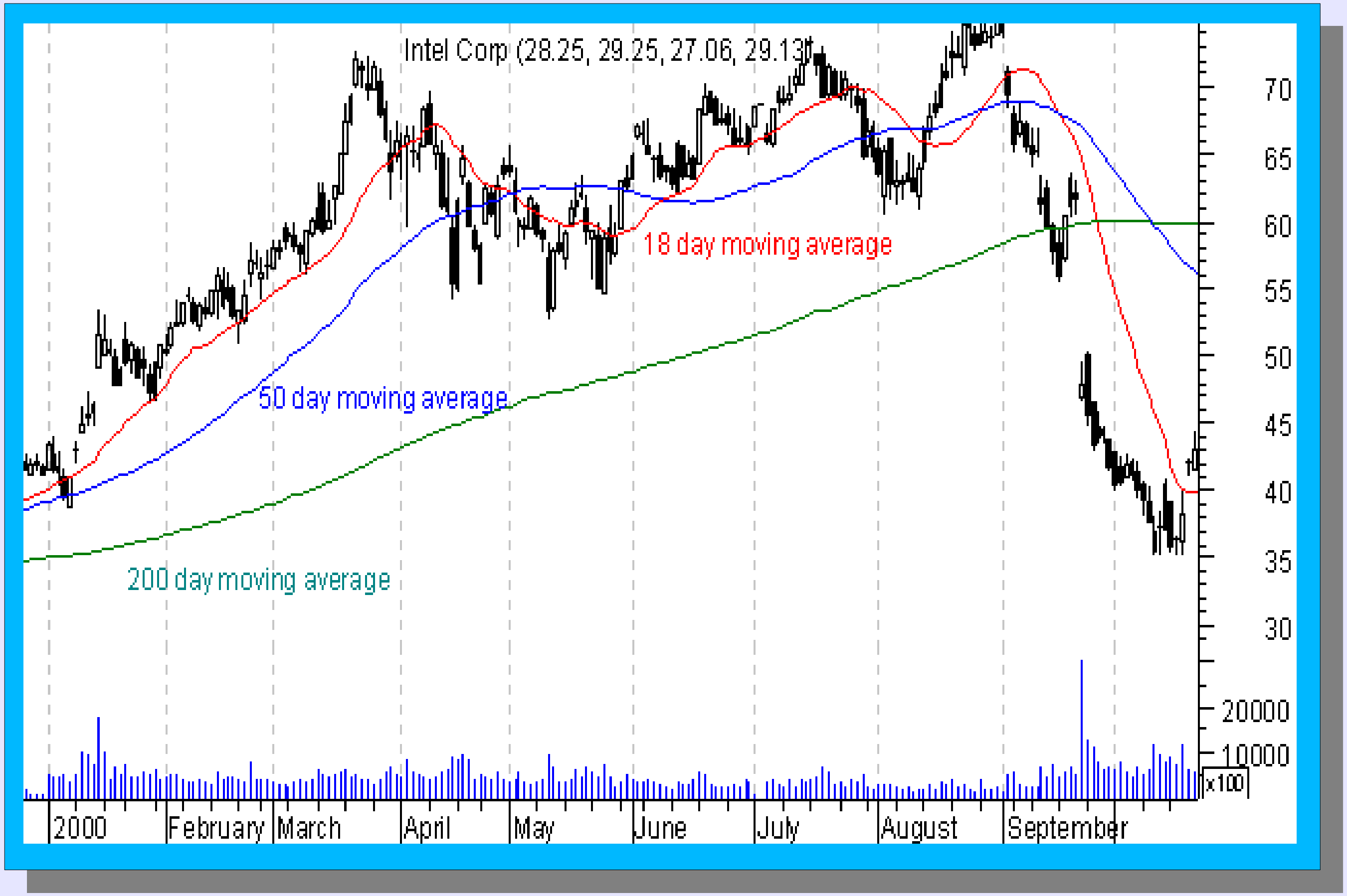
Exponential Moving Average. A standard way to calculate a snapshot dynamic average of (potentially) infinitely many samples, such that old samples are aged out.

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

t_k	k-th sample
τ_k	k-th average
α	recency coefficient (0-1]

$\alpha \to 1$	history matters little, recent samples prevail
$\alpha \to 0$	history matters a lot, resistant to fluctuations

Example: tracing trends in stock prices



d -sample EMA

$$\alpha = \frac{2}{1+d}$$

For CPU scheduling, systems often assume that the duration of the next CPU burst will be the EMA of past durations. The value of α used is typically in the range 0.5-0.8.

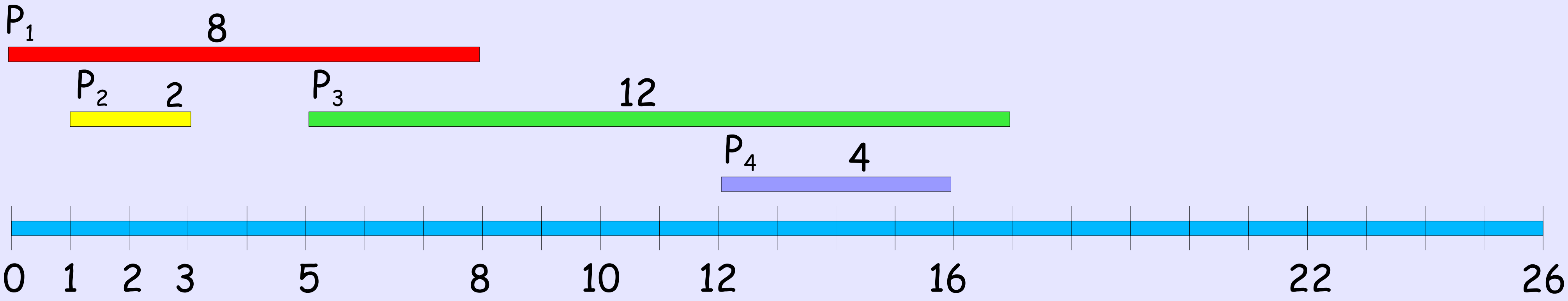
Preemptive SJF

What if the process doesn't want to exhibit an I/O burst as predicted?

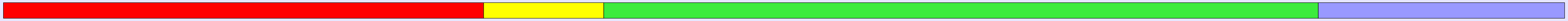
Obviously, a real-life policy must be preemptive, e.g.,

- The CPU is assigned to the process with the shortest estimated duration of CPU burst.
- If the current process outruns the estimate, it can be preempted.
- When a new process becomes ready, and its estimated duration of CPU burst is less than the estimated leftover duration of the current process's burst, the current process is preempted by the new process.

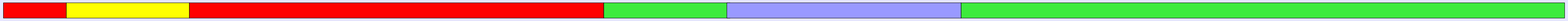
Example



Non preemptive SJF: average turnaround = (8+9+17+14)/4 = 12



Preemptive SJF: average turnaround = (10+2+21+4)/4 = 9.25



Thus, preemptive SJF gives a better turnaround time. This isn't surprising, as shorter jobs, not being known in advance, cannot be noticed (and properly accounted for) by the non-preemptive variant when earlier (and longer) jobs are started.

Priority

Generally, scheduling schemes may be based on the notion of priority associated with a process.

For example, with SJF, the priority is:

$$P = \frac{1}{\tau}$$

where τ is the predicted duration of the next CPU burst.

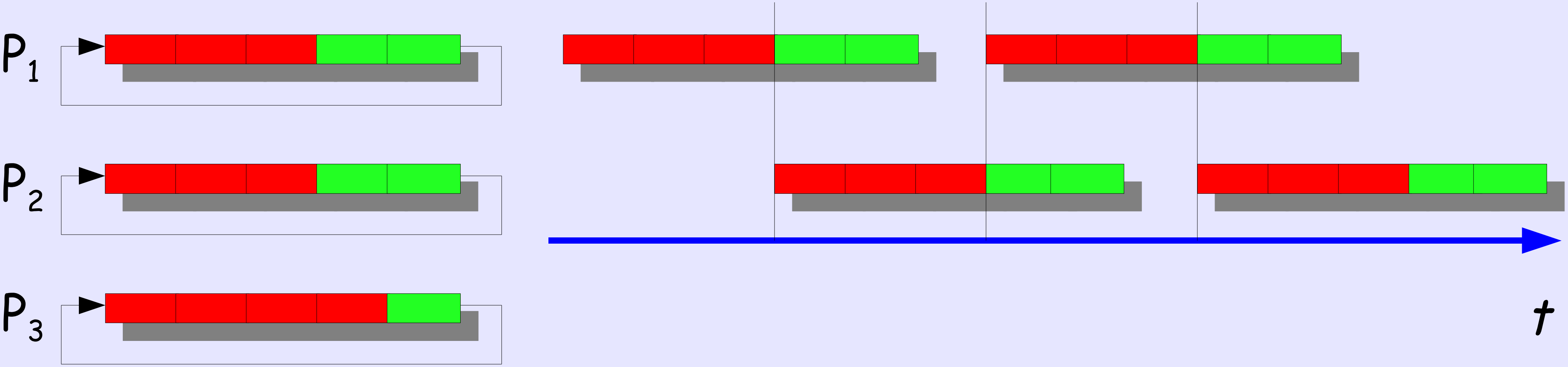
In a real system, the priority may be a function of several factors.

Internal factors: measurable (internal) properties of the process, e.g., CPU burst length, memory size, objective importance of the proces.

External factors: attributes assigned by humans, e.g., subjective importance.

Starvation

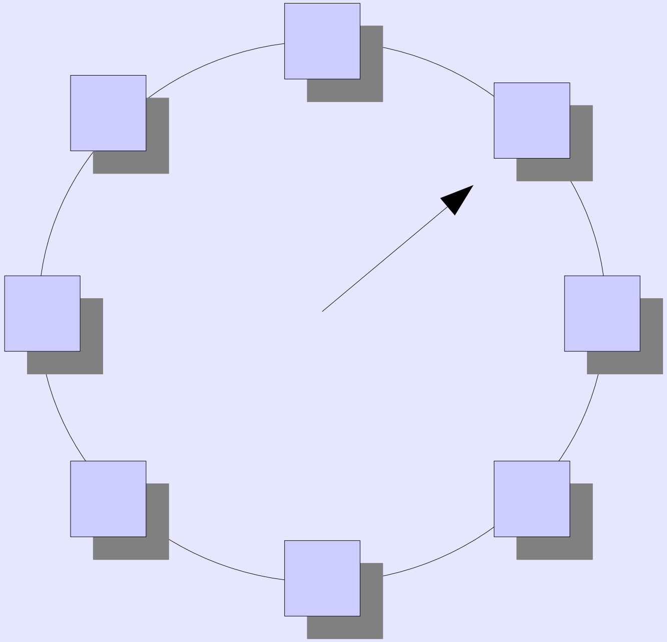
A scheduling policy may be starvation prone, i.e., a constant availability of high-priority jobs may result in a starvation of low-priority jobs. For example, SJF (preemptive or not) is starvation prone:



The standard way to eliminate starvation is to add a time component to the priority, e.g., the age factor.

Response time

Average turnaround time is not the only performance criterion of interest. For an interactive job (with a human user sitting in front of a terminal), the primary criterion is the (expected) waiting time for the program's reaction.



Round Robin

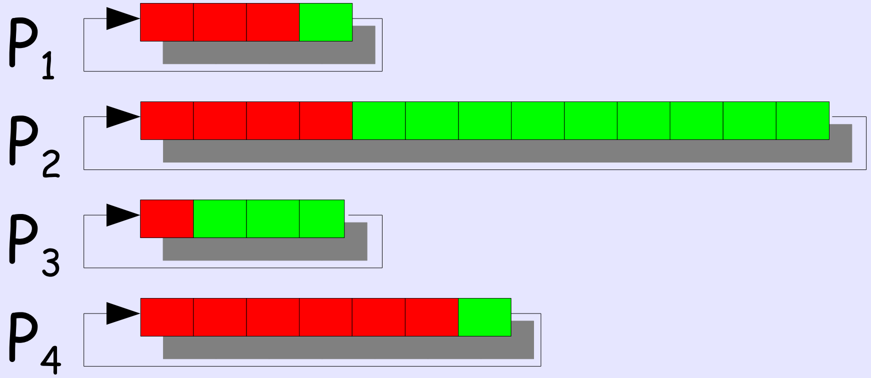
The system defines a time slice. A process can lose the CPU in two ways:

It hits an I/O burst, i.e., its CPU burst runs out naturally.

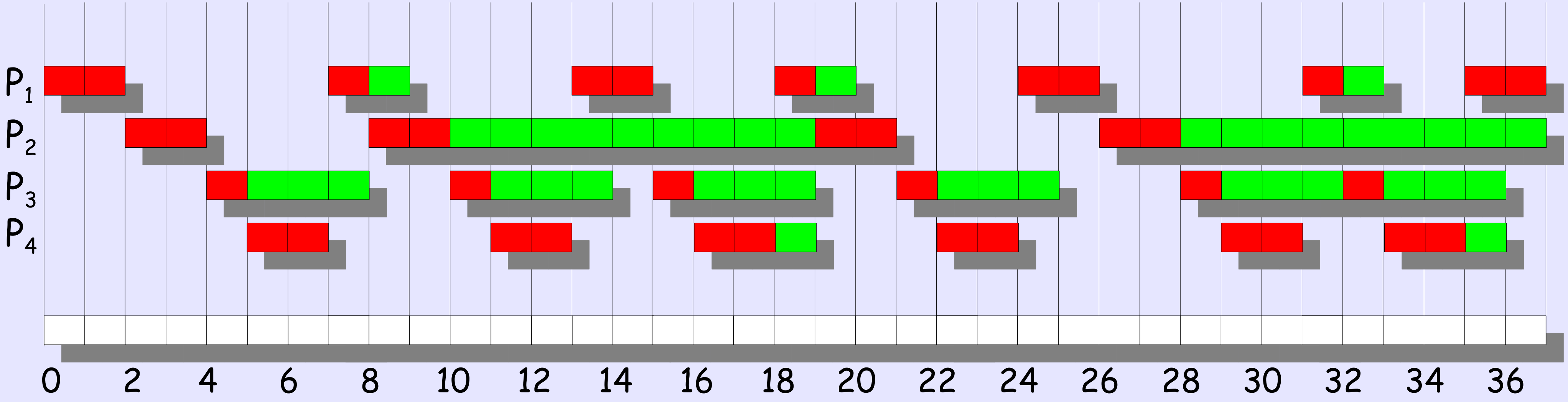
Its time slice expires.

The time slice can be the same for all; also, some processes may be getting a larger share of the CPU – reflecting their higher importance.

Round Robin: example



Time Slice = 2



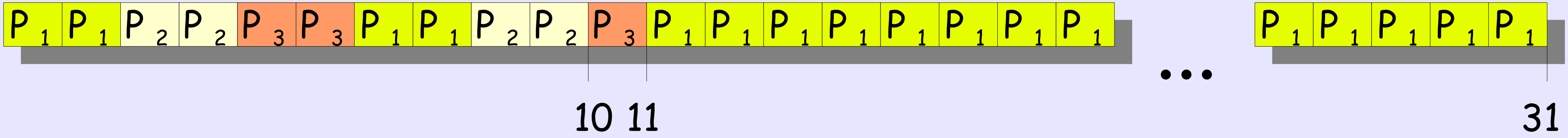
Note: with SJF (preemptive or not) P_2 and P_4 would starve.

Round Robin ...

... doesn't give the best turnaround time.

P_1	24
P_2	4
P_3	3

Time Slice = 2



P_1 P_1 P_2 P_2 P_3 P_3 P_1 P_1 P_2 P_2 P_3 P_1 P_1 P_1 P_1 P_1 P_1 P_1 P_1 ... P_1 P_1 P_1 P_1 P_1

10 11 31

Average turnaround: (10+11+31)/3 = 17.33. We know that we can do better than this, i.e., SJF yields 13.67.

So what does it minimize?

Round Robin ...

... guarantees that if a process "has something to say," it won't have to wait for more than

$$(N-1) \times S$$

where *N* is the number of processes and *S* is the time slice.

Theoretically, the best response time is achieved when *S* is the shortest possible, i.e., single instruction time.

Practically, the context switch overhead imposes a limit on how small *S* can be.

Typically, *S* is between a millisecond and a tenth of a second.

How to schedule kernel processes?

What is the best scheduling policy for high priority, extremely I/O bound processes?

FPPS: Fixed Priority Preemptive Scheduling

Processes are rigidly ordered in some fashion. The first process that is ready gets the CPU.

We know for a fact that the CPU bursts are always short. There is no need to guess or estimate.

Therefore, fixed (high) priority is OK.

Also, the exact ordering of those processes (their actual priority) doesn't really matter, as long as they are ahead of everything else.

A digression: types of kernel processes

What do we need kernel processes (threads) for?

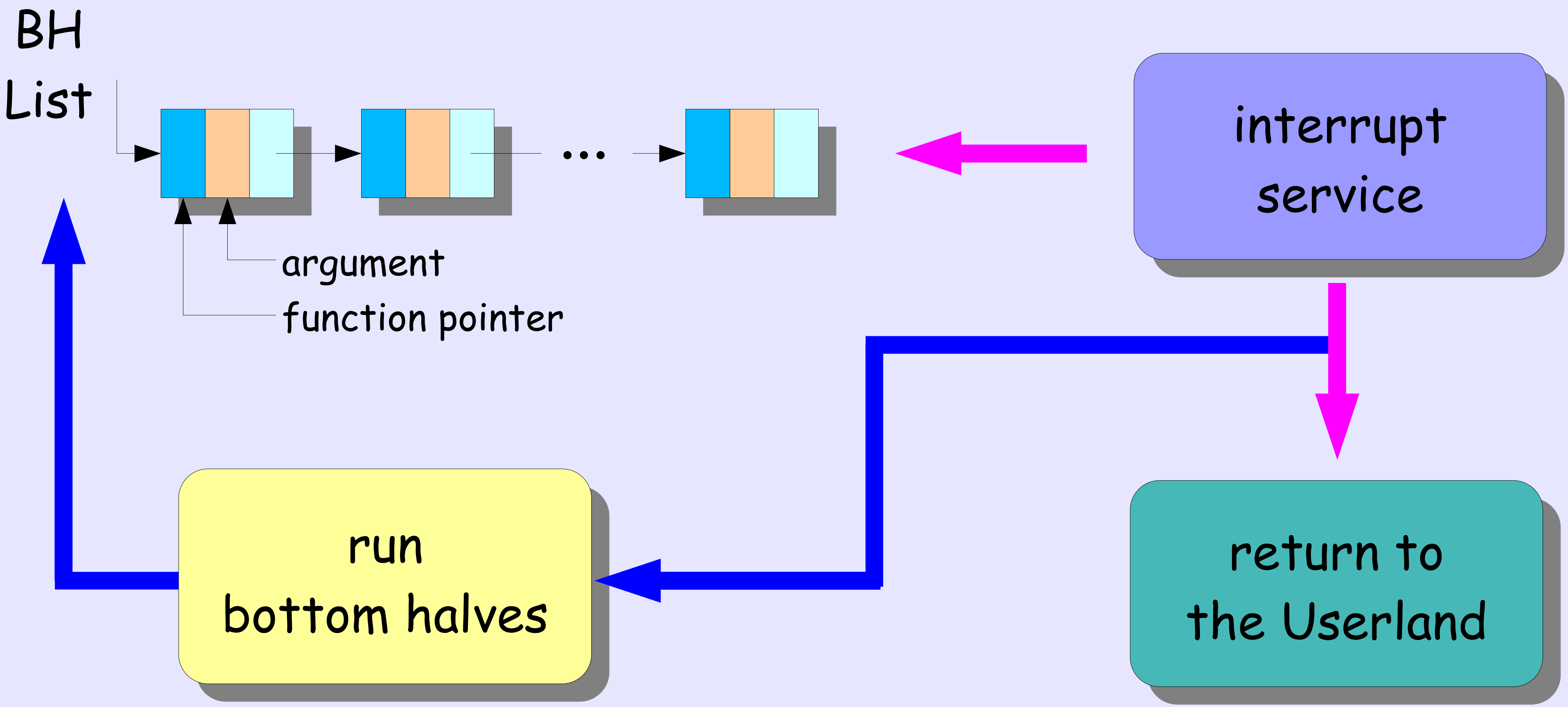
To perform actions that cannot be properly performed from an interrupt service routine. For example:

- something that does I/O on its own and looks like a thread (paging-related daemons, file-system components, especially for networked file systems)
- something that contains complicated actions of lower priority than interrupts; while those actions are triggered by interrupts, it makes better sense to carry them out elsewhere

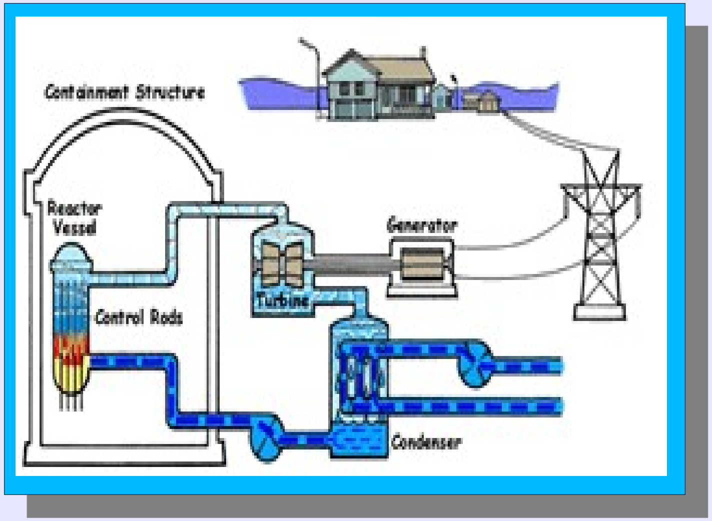
Note that interrupts may block other interrupts, and thus impair the system's responsiveness, if they are "too heavy."

Bottom halves

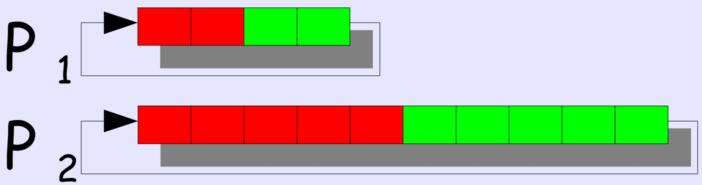
In the latter case, if the action looks like a single burst, it need not be implemented as a process.



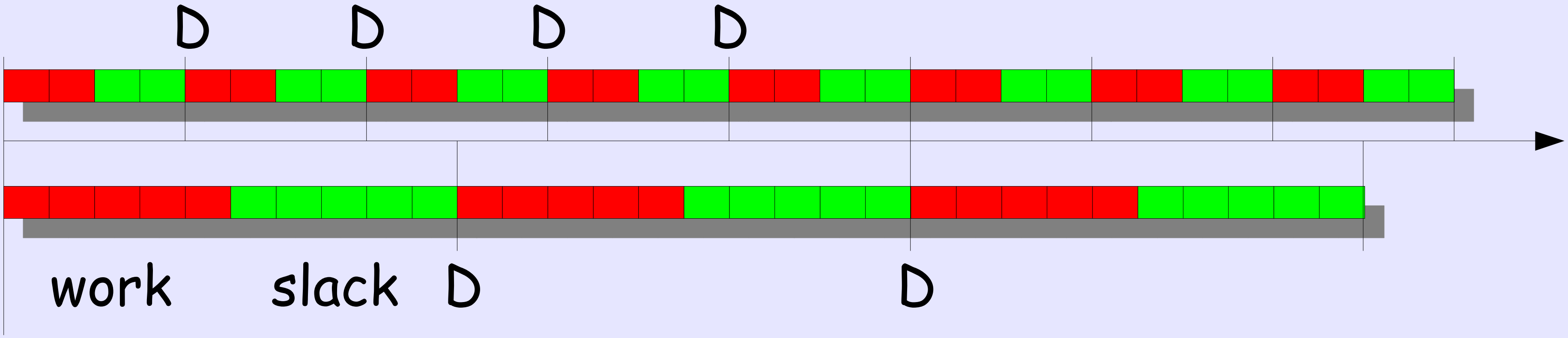
Hard deadlines



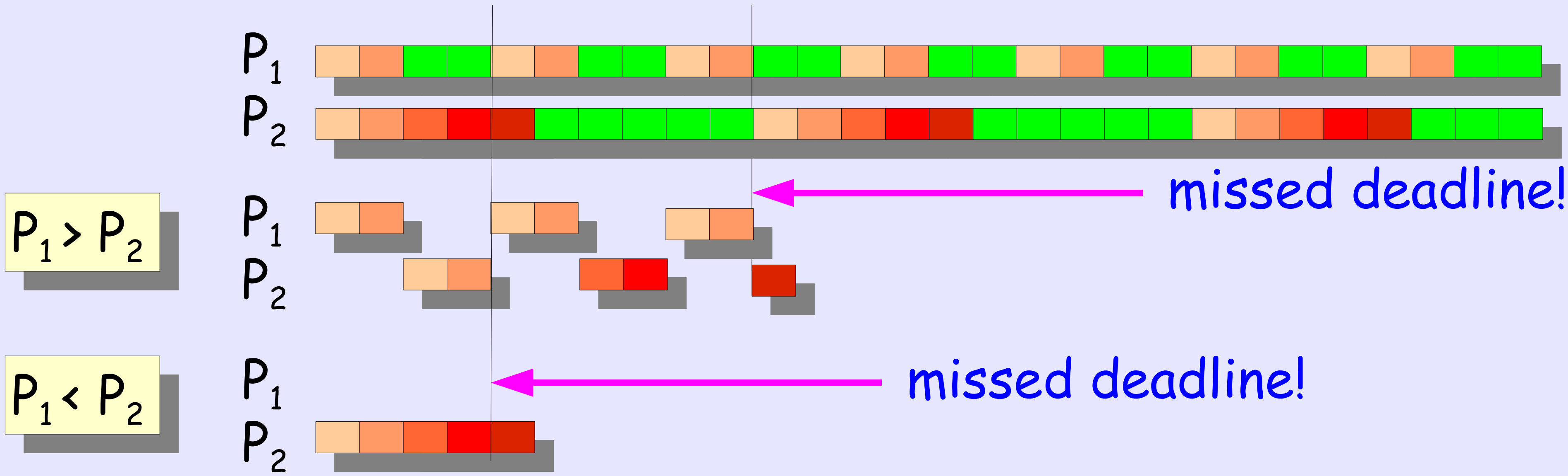
In some (real-time) applications, average performance is irrelevant. What matters is the obedience of hard deadlines.



P_1 starts at 40 msec intervals, has enough work to run for 20 msec, and MUST complete before its next ready time. P_2 starts at 100 msec intervals, has enough work to run for 50 msec, and ...



FPPS?



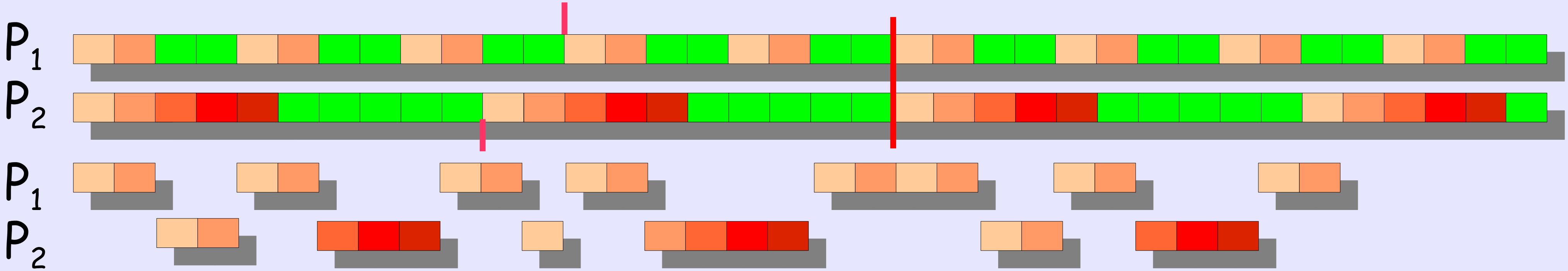
$P_1 > P_2$

missed deadline!

$P_1 < P_2$

missed deadline!

Deadline Scheduling: run the process with the closest deadline.



Multimode systems

SJF: batch (or background) jobs. Turnaround time is the measure that matters. Aging may be added to avoid starvation.

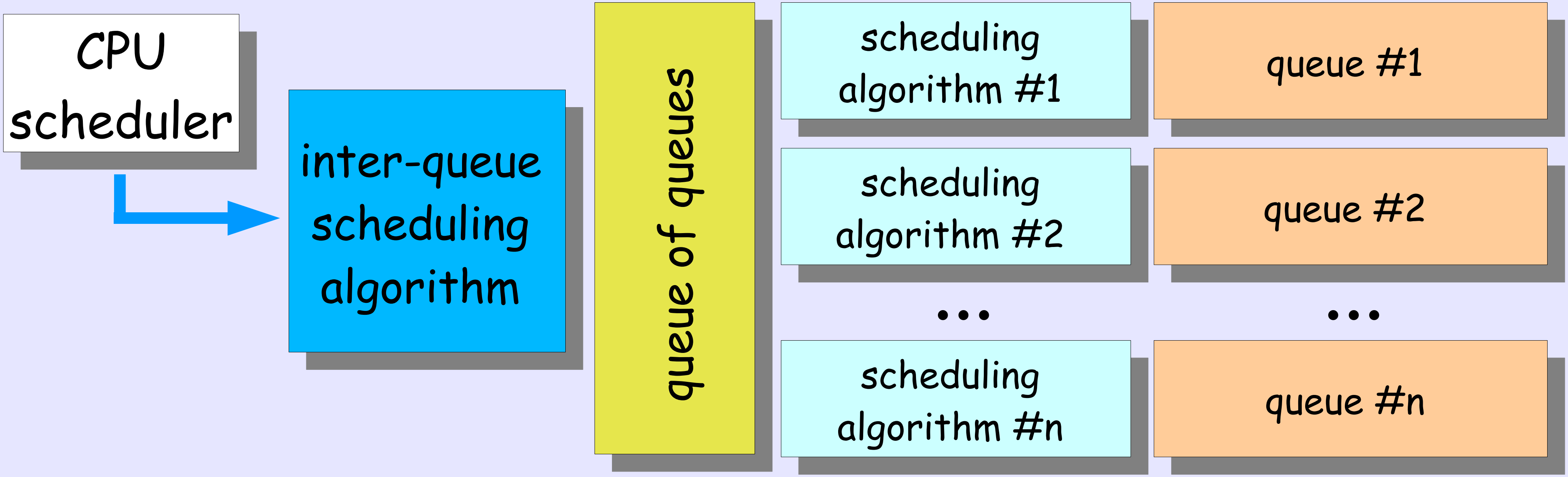
Round-Robin: interactive jobs. Response time is the objective.

FPPS: kernel jobs. They should have a very high priority, and their relative importance doesn't really matter.

Deadline Scheduling: real-time jobs with hard constraints. Obedience of deadlines is the objective.

How do different types of jobs coexist within a single system?

Multi-level queues



CPU scheduler

inter-queue scheduling algorithm

queue of queues

scheduling algorithm #1

queue #1

scheduling algorithm #2

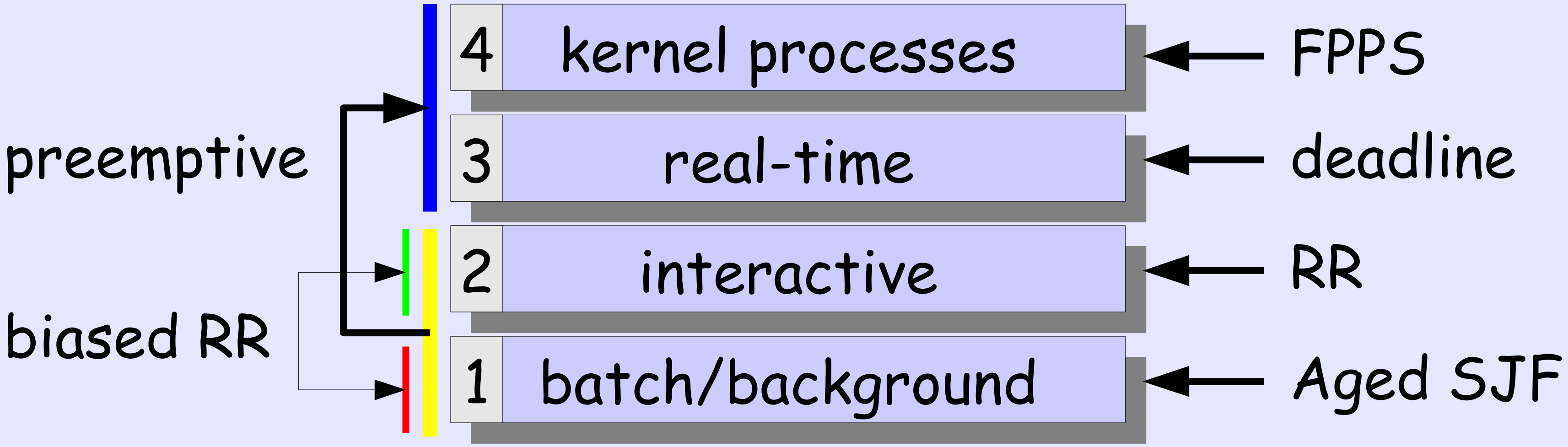
queue #2

...

...

scheduling algorithm #n

queue #n



4 kernel processes ← FPPS

3 real-time ← deadline

2 interactive ← RR

1 batch/background ← Aged SJF

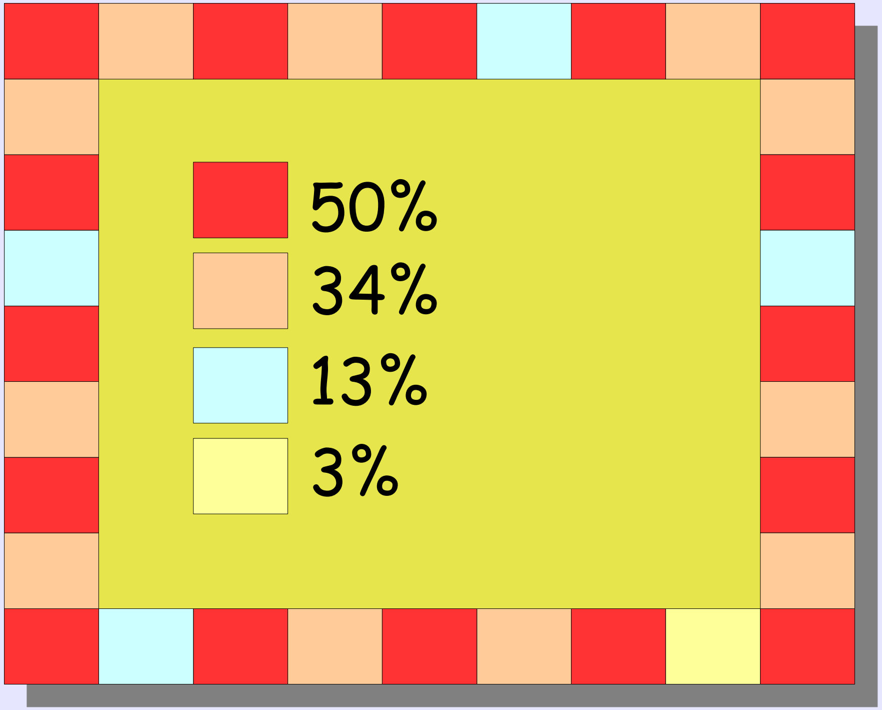
preemptive

biased RR

Feedback queues

Combining the best of RR and SJF. Imagine four RR queues:

4	slice = 8
3	slice = 16
2	slice = 32
1	slice = 64



When a process shows up, it is put in queue 4.

If a process uses up its entire slice, it is demoted to the next (lower-priority) queue, unless it already is at the bottom.

If a process uses less than half its slice, it is promoted to the next higher-priority queue, unless it already is at the top.

A bit on Real-Time Systems

Standard system:

Capacity:	high throughput, statistically good utilization of all equipment and low overhead
Responsiveness:	low average turnaround time, low average response time
Overload:	fair performance degradation

Real-time system:

Capacity:	“scheduleability,” number of context switches per second, number of interrupts per second
Responsiveness:	worst-case delay, low variance, deterministic behavior
Overload:	stability, unaffected performance for critical tasks

Primary sources of problems

Scheduling policies, e.g., aging introduces nondeterminism, rigid priorities are starvation-prone, deadline scheduling requires extra specification and is not for everybody.

Context switch, e.g., extra actions upon context switch may reduce capacity and increase nondeterminism.

Synchronization tools and paradigms, e.g., critical sections in the kernel may impair scheduleability (context switches and interrupts).

Memory management, e.g., sophisticated contemporary techniques of memory mapping introduce a lot of nondeterminism.

Examples of non-RT paradigms:

Non-preemptibility of processes in the Kernelland (many standard UNIX systems work this way).

Processes cannot be rescheduled until they leave the Kernelland. Usually, they spend very little time in there, but who can tell the bound?

Virtual memory, including lazy loading, also DLLs.

A program running happily at full speed may suddenly have to wait for I/O for no good reason.

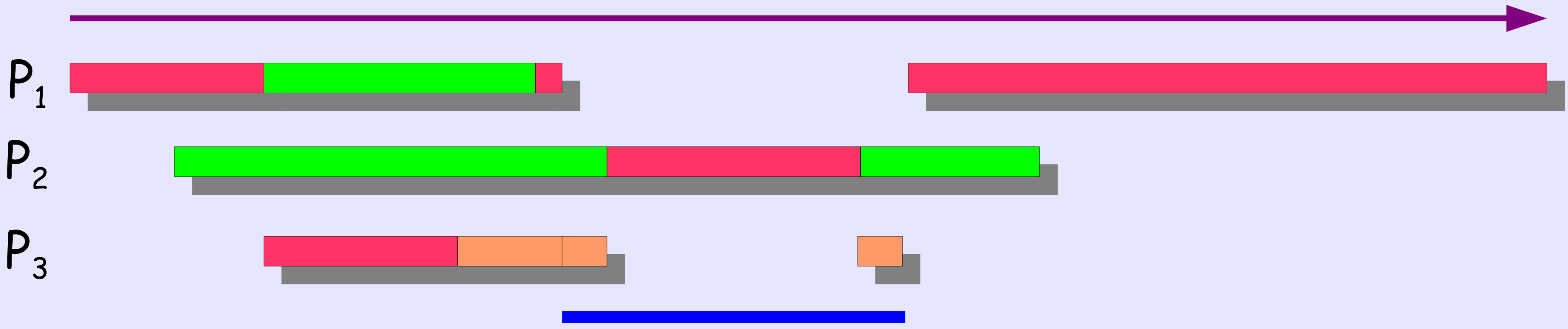
Disk inodes for representing devices (as on UNIX systems).

Access to an otherwise extremely fast device may stall because its inode has disappeared from the cache.

Synchronization issues

Excessive use of interrupt masking in the kernel may make it difficult to produce a decent bound on interrupt latency.

The priority inversion problem:



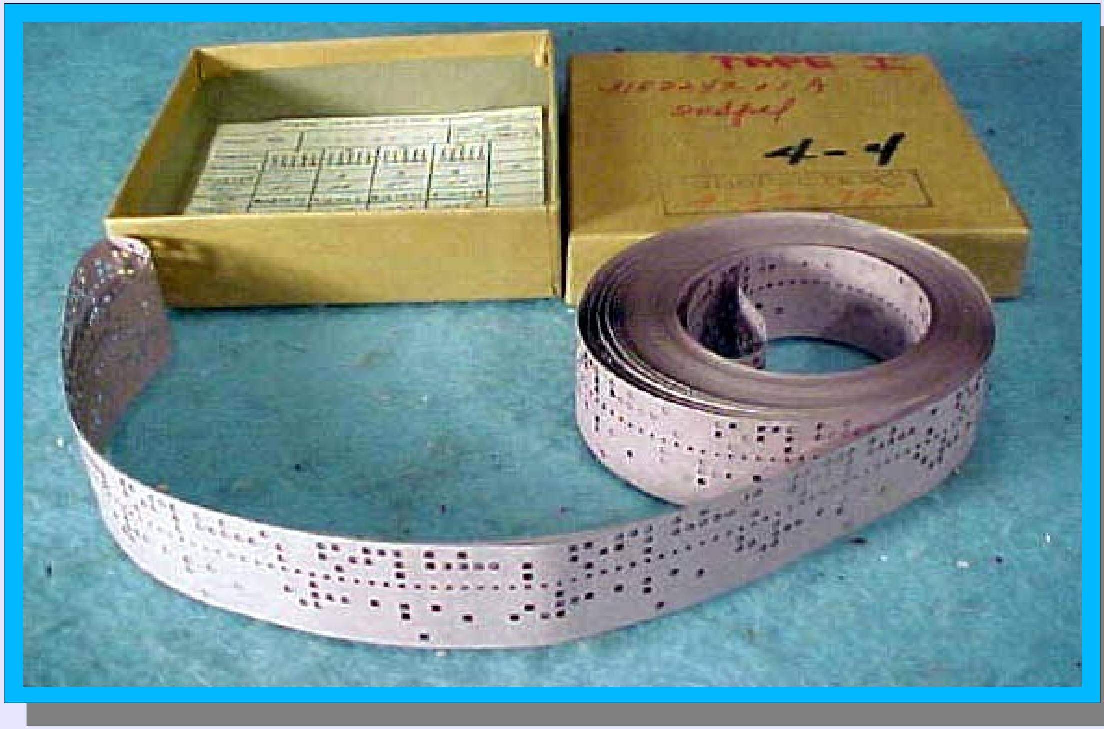
Within this area, P_1 has been effectively demoted to the priority level of P_3 .

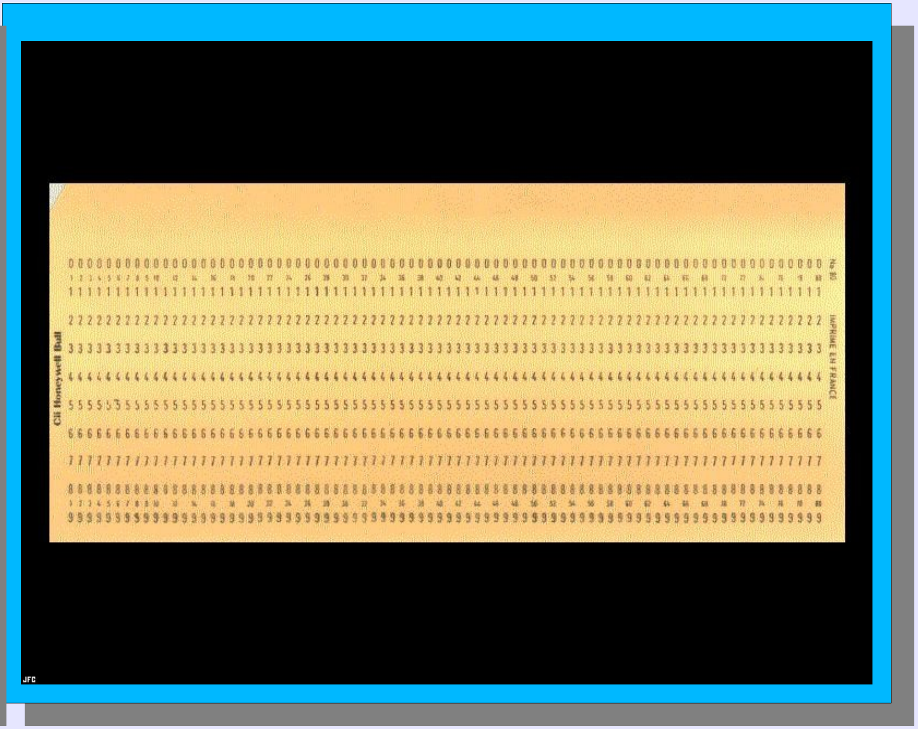
A process holding a critical section should be temporarily promoted to the highest priority of a process waiting for it.

File systems

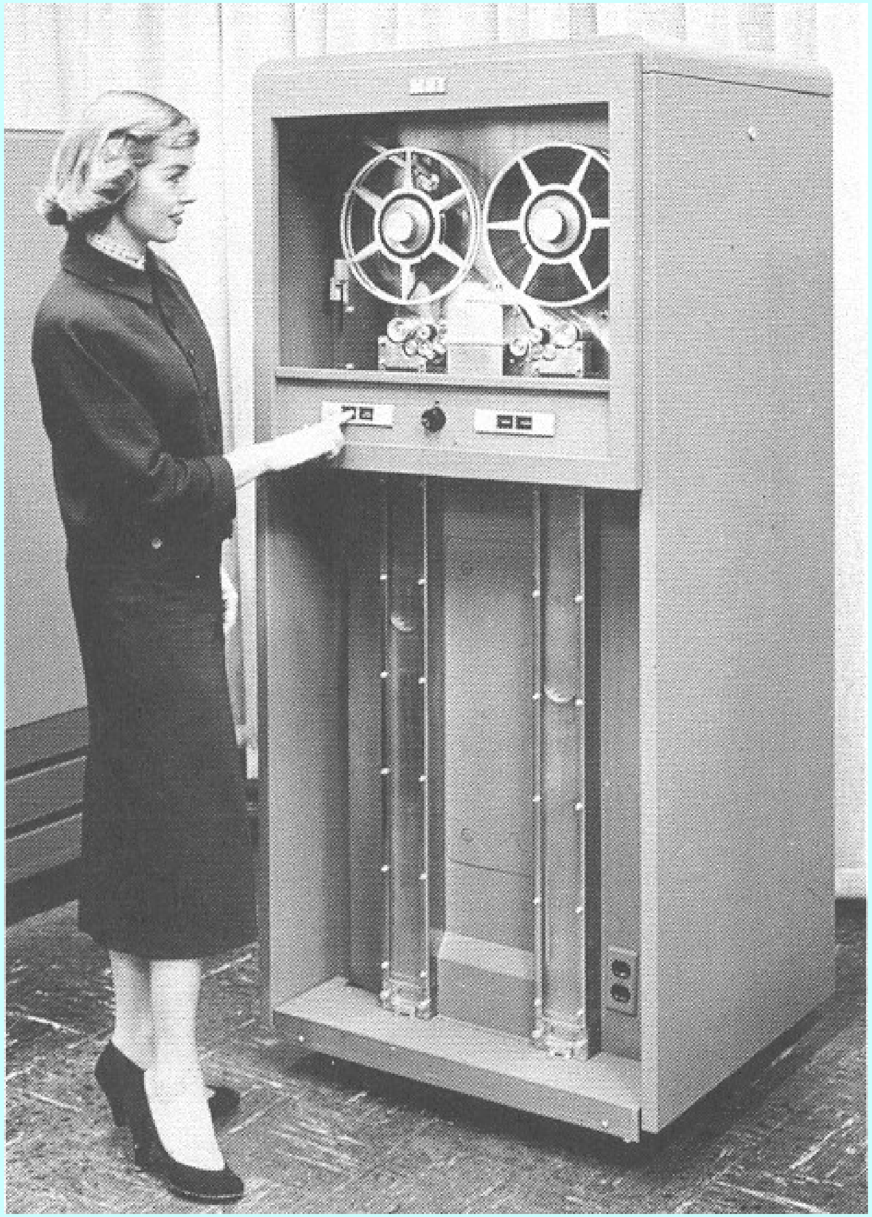
Do we want a definition? OK, here is one. A file is an identifiable collection of information, typically stored on a non-volatile medium for safe archival and retrieval.

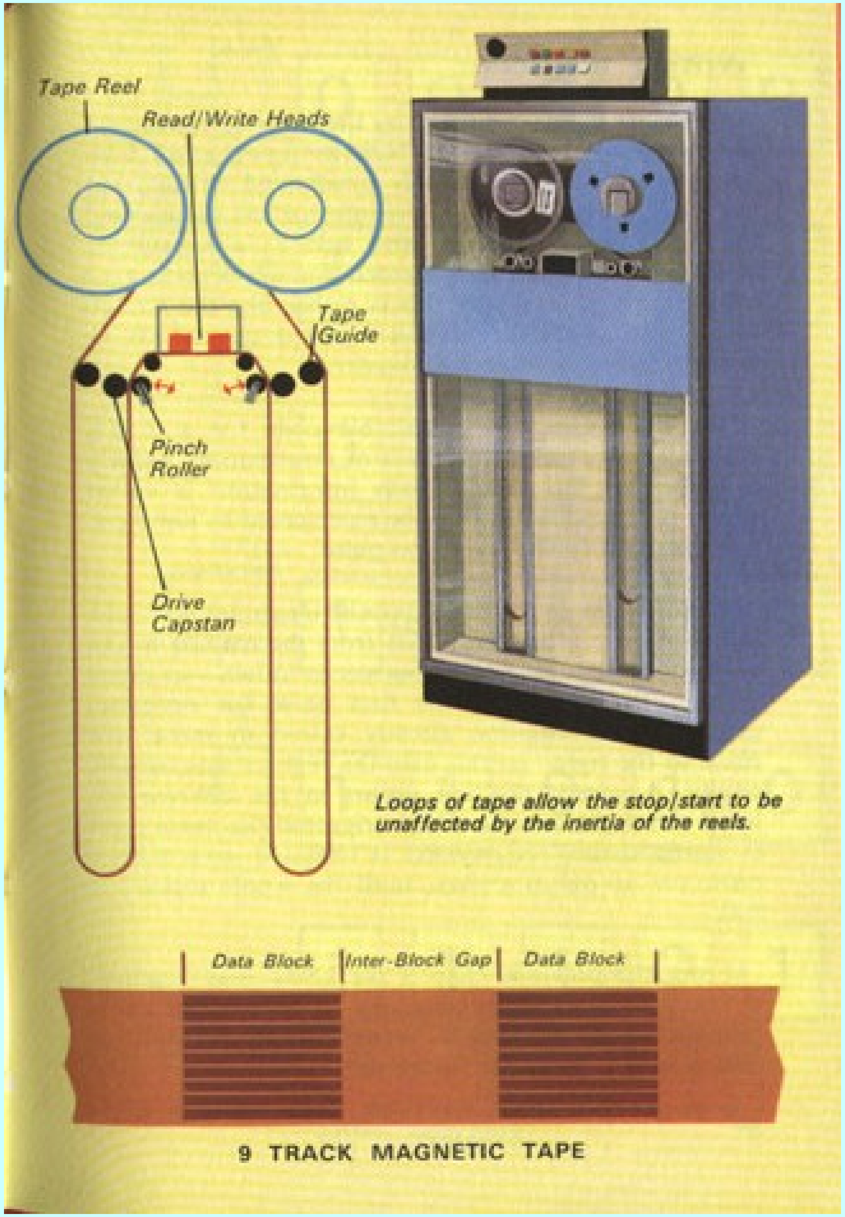
To make it interesting, let us look at files from a historical perspective. What were the early media for storing stuff?





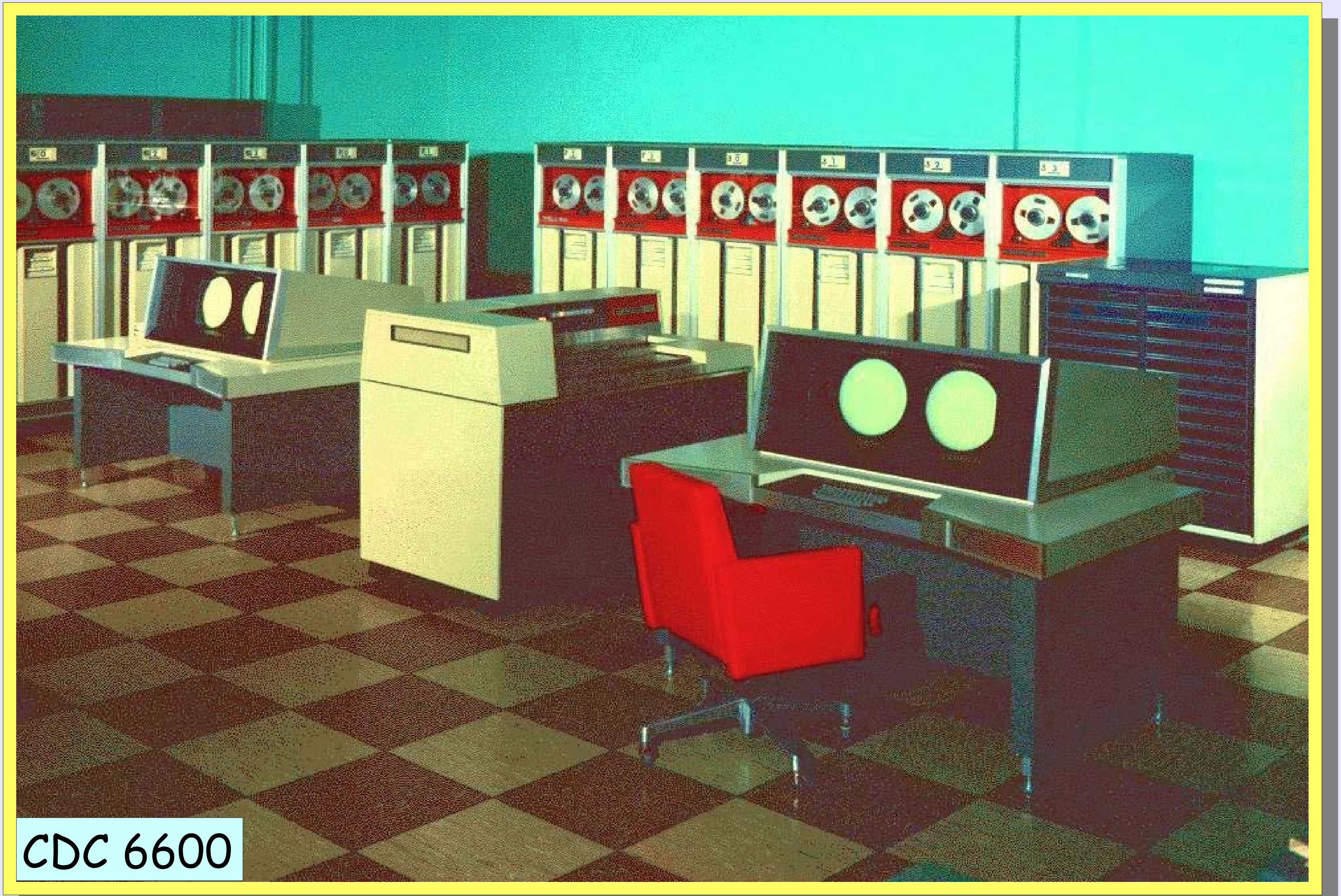
Let us start from magnetic tapes





Loops of tape allow the stop/start to be unaffected by the inertia of the reels.

Early systems heavily relied on them



CDC 6600

A simple tape-based file system

Each file is stored on a separate tape.

Tapes are labeled, e.g. with stickers.

When a program references a tape, it flashes a message on the console.

```
1.W JANUS
    ############............... LP0 PRT
2.D NRIPGZY **
    ATTACH, OLDPL, ID=PAWEL, MR=1.
3.R IMM01A9 AB
    FTN,I.
4.M QUARK3Y AC
    SIMULATOR.
5.T HOZA0HY AC
    REQUEST TAPEFILE,NT. RING, VSN=CERN4332.
    ASSIGN TAPE AND TYPE N.GO.
6.R POLI0FU AE
    UPDATE PL=MYFILE.
```

Problems

Poor utilization of tapes. Either every tape must be large enough to accommodate the largest conceivable file, or we have to deal with tapes of different sizes and force the program to specify the needed size when it creates a file.

Poor protection. What if the operator mounts the wrong tape by mistake?

Files are sequential. No easy way to access distant fragments of a file. Difficult to update a file "in place."

Inconvenience. A lot of manual labor.

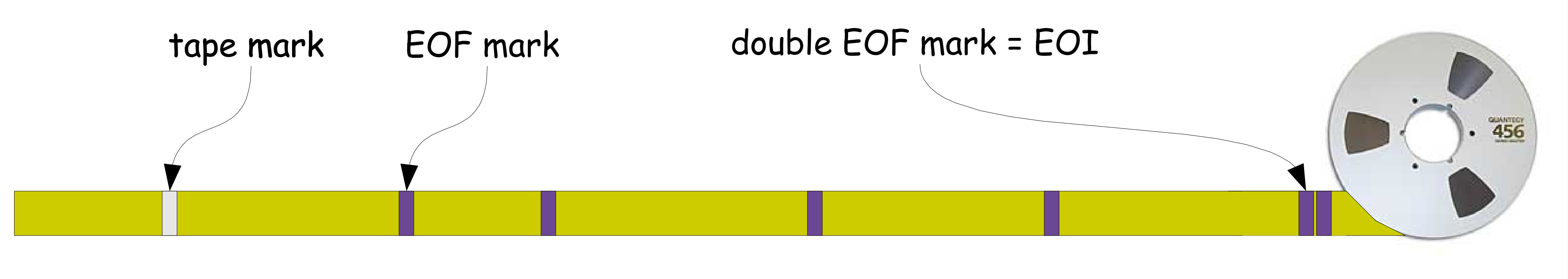
Some improvements

Protection methods:

Physical: RING ON/OFF. No write allowed without the ring.

Logical: VSN label on tape. The system checks whether the VSN specified by the program matches the one on tape. Operator can override.

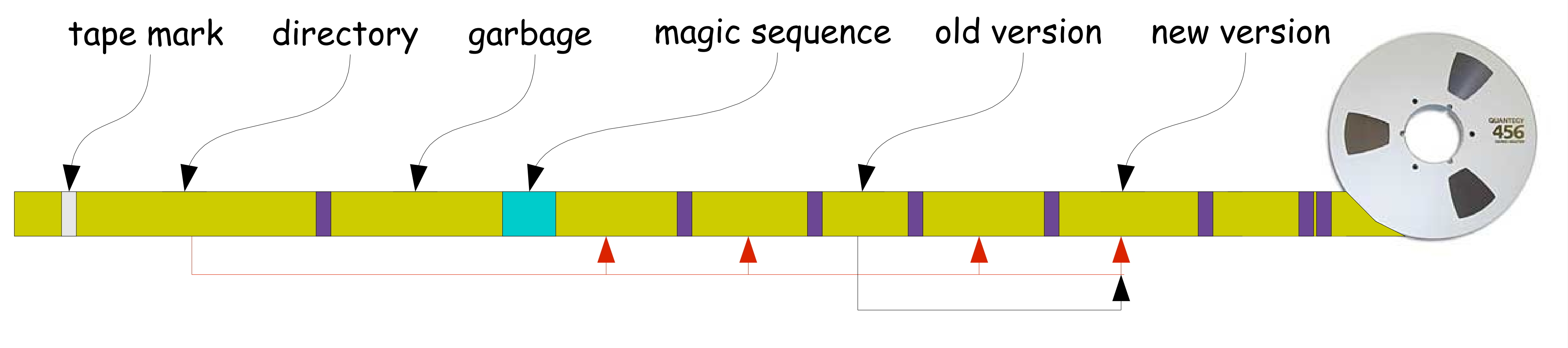
Improving utilization by storing multiple files on one tape:



Operations: skipf (tape, n), skipb (tape, n), skipeoi (tape)

More improvements

Directories on tape:



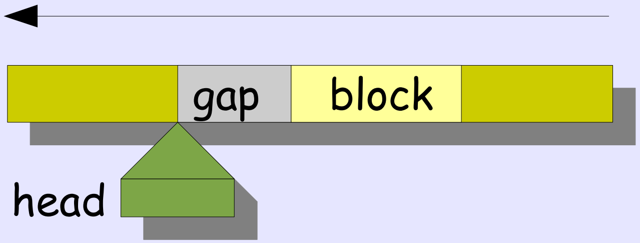
The problem is the inherent non-determinism: if you write the same information twice, starting from exactly the same location, you may end up in two (slightly) different places.

In mid-70's, some of those archival systems became “advanced” and semi-friendly. Tapes were the second safest (after punched cards) storage medium, offering a lot of capacity – by the standards of those days.

Blocking

Practically all storage media require information to be blocked, but not always for the same reason.

Tapes:



A written record must be preceded by a gap needed to absorb the tape's inertia.

Tape density: 1600 bpi
Gap size: 1/4''

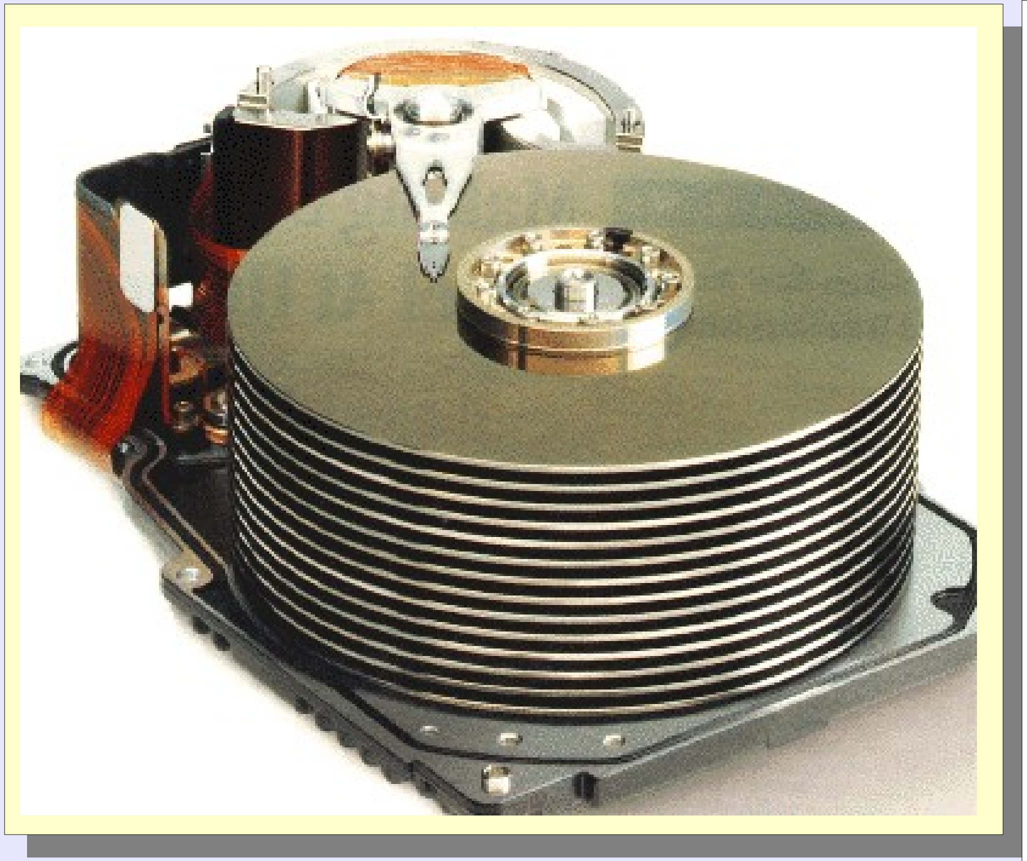
80 byte records: 1/20'' (9 tracks)
Utilization: (1/20)/(1/20+1/4)=1/6

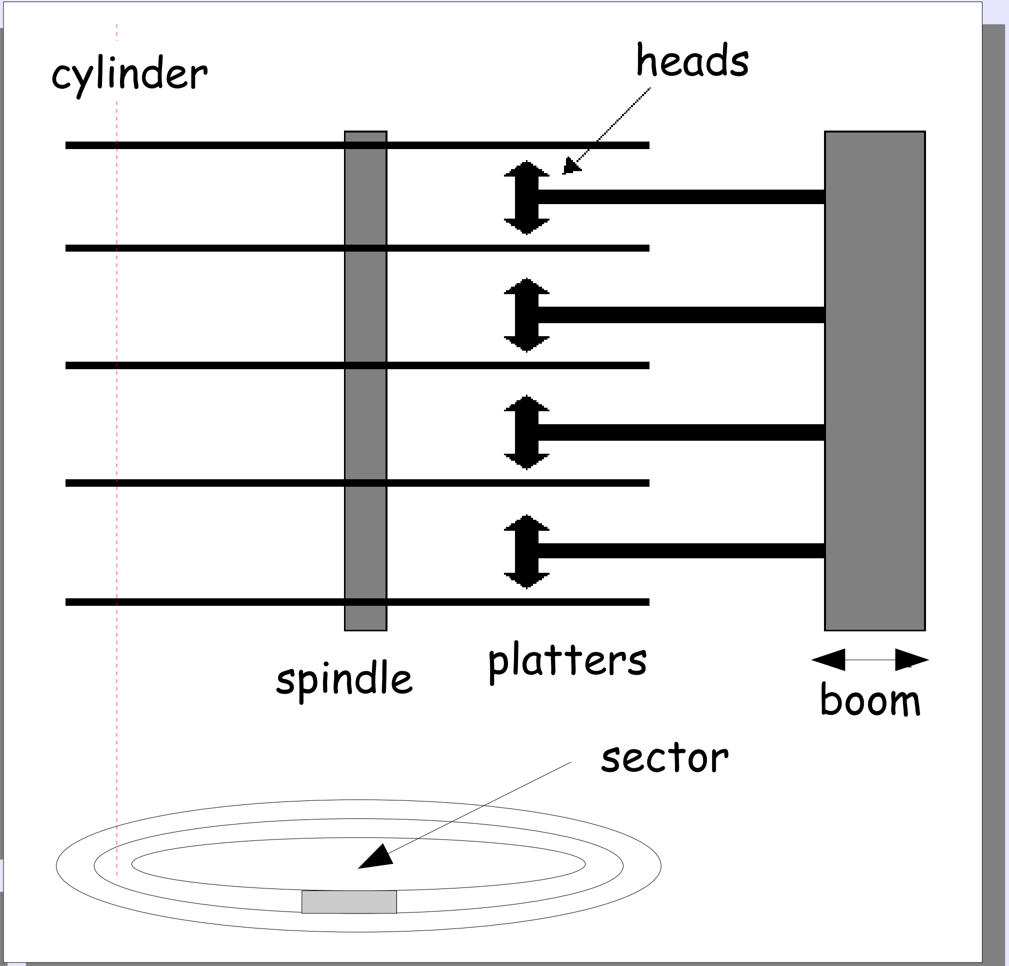
800 byte records: 1/2'', Utilization: (1/2)/(1/2+1/4)=2/3

Large blocks = better tape utilization.

Disks

Information on disks is also blocked. A single physical block (sector) is the minimum unit of information that can be transferred (and addressed).





disk -> cylinder -> head -> sector

Surface mapping

It makes sense to view a disk as a continuous range (array) of sectors. For example, assume that:

P = the number of platters, so $2P$ = the number of heads

C = the number of cylinders

T = the number of sectors per track

Then the total number of sectors $S = 2P \times C \times T$. A number s between 0 and $S-1$ can be mapped into the disk surface in the following way:

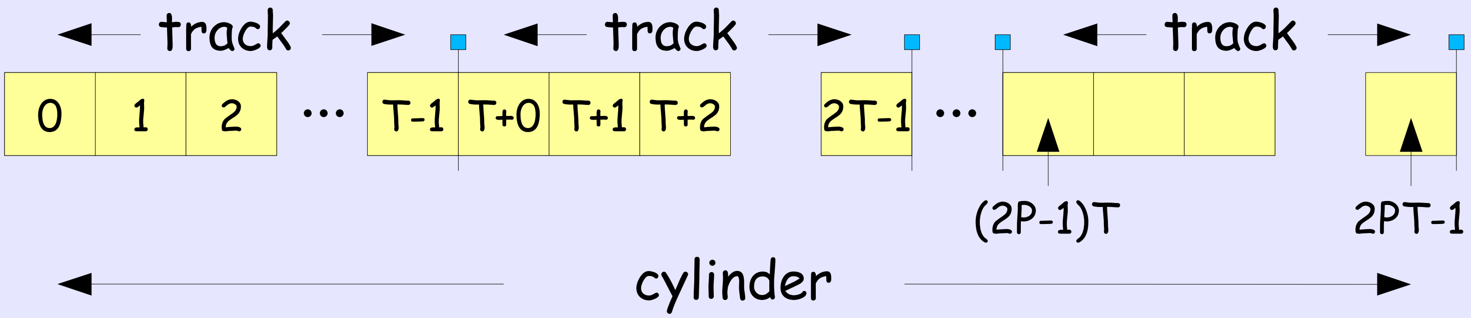
$c = s / (2P \times T)$ is the cylinder number

$u = s \bmod T$ is the sector number within track

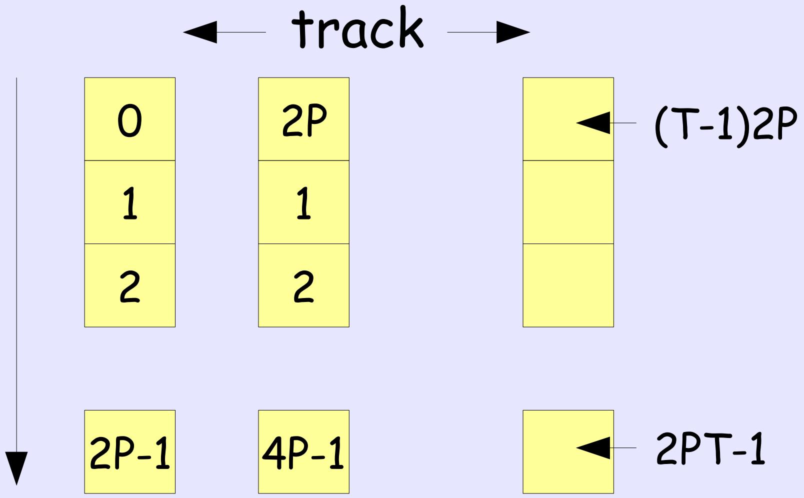
$h = (s/T) \bmod 2P$ is the head number

Possible mappings

The mapping from the previous slide can be “unwound” as follows:

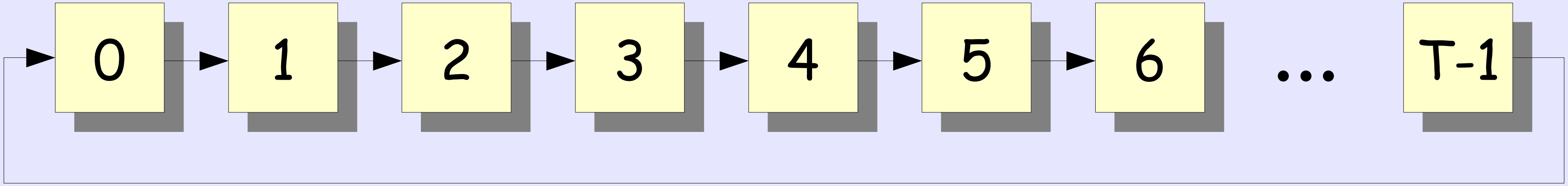


We can also go like this:

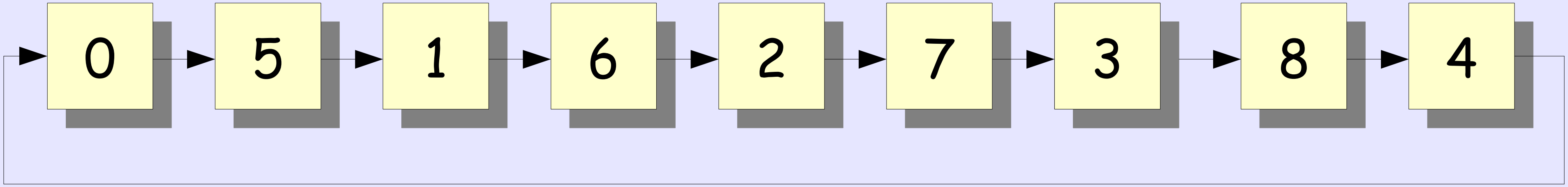


If the controller allows for that, we may be able to read all the sectors from all platter sides at once, at the cost of reading one sector.

Hopscotch



Sometimes the system would prefer to have some breathing space between consecutive sectors. In such a case, it may make sense to number them like this:



It looks nice is the track length is odd (more generally, if it doesn't have too many divisors).

Design issues in file systems

The user-visible part:

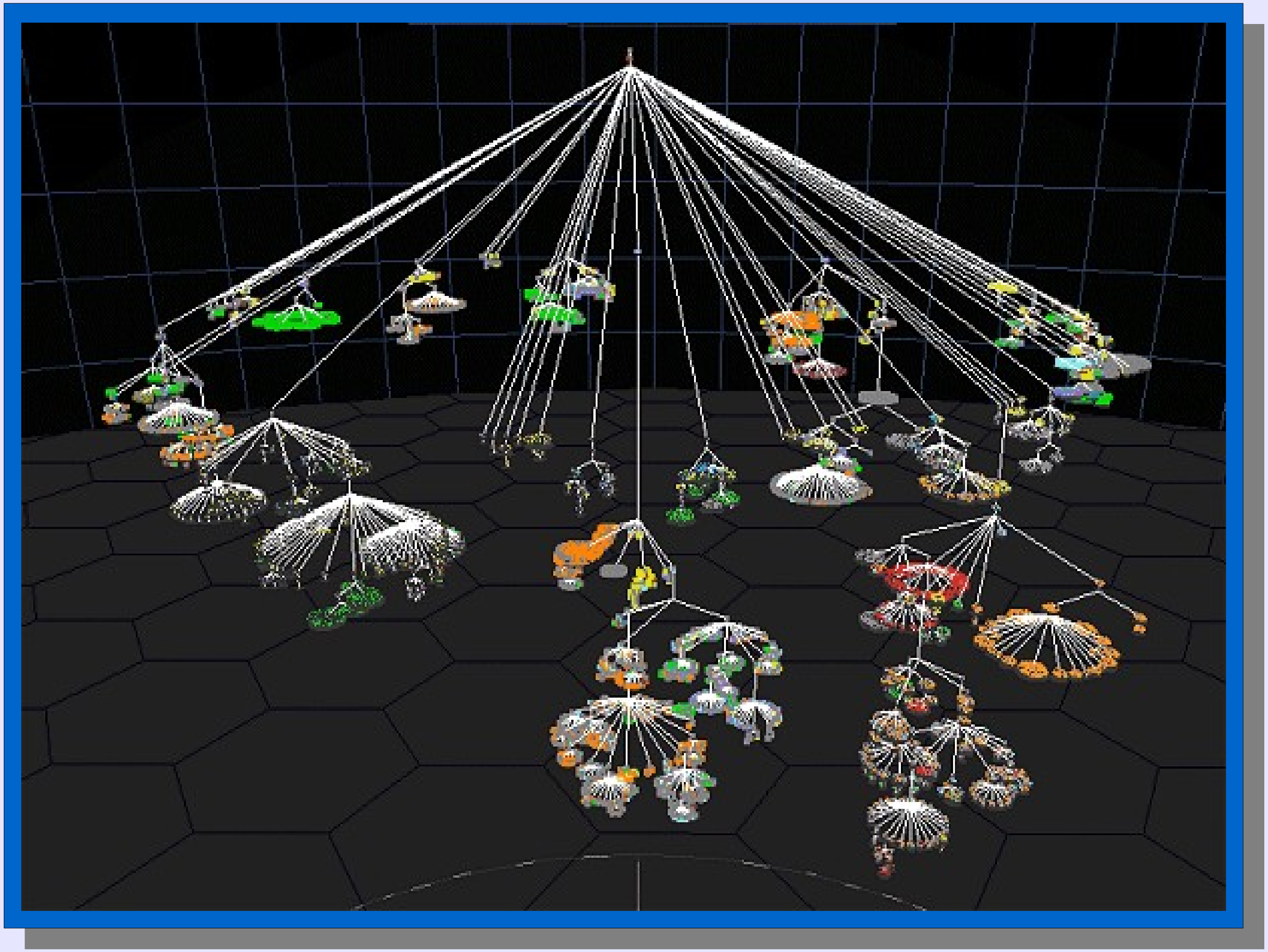
- file identification and location (hierarchy)
- operations on files (access methods)
- efficiency
- reliability, resilience, integrity

Internals:

- organization of disk storage
- caching
- integration with VM
- reliability, resilience, integrity

File identification and location

These days, practically all file systems follow essentially the same paradigm, an unlimited tree-like hierarchy:



This is a starlight visualization of a hierarchical file system: see http://starlight.pnl.gov/

Surprisingly, it took people quite a while to invent this “wheel” of all contemporary file systems.

Example: SCOPE for CDC 6000

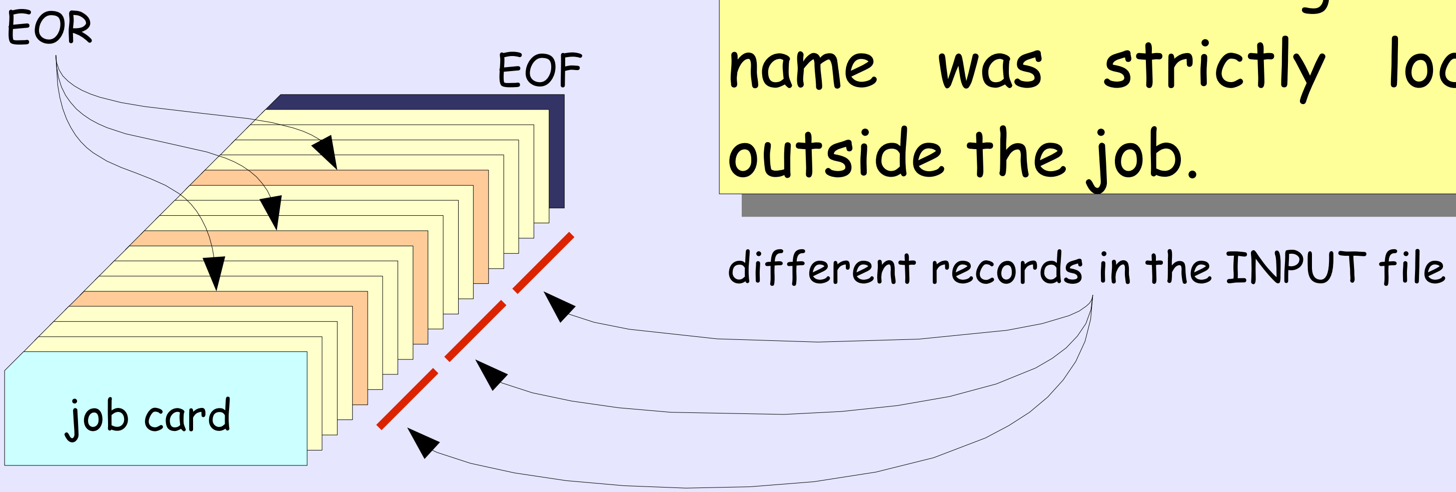
There were two types of files: local and permanent. A local file only survived until the end of the job. Each job received two standard files:

INPUT

OUTPUT

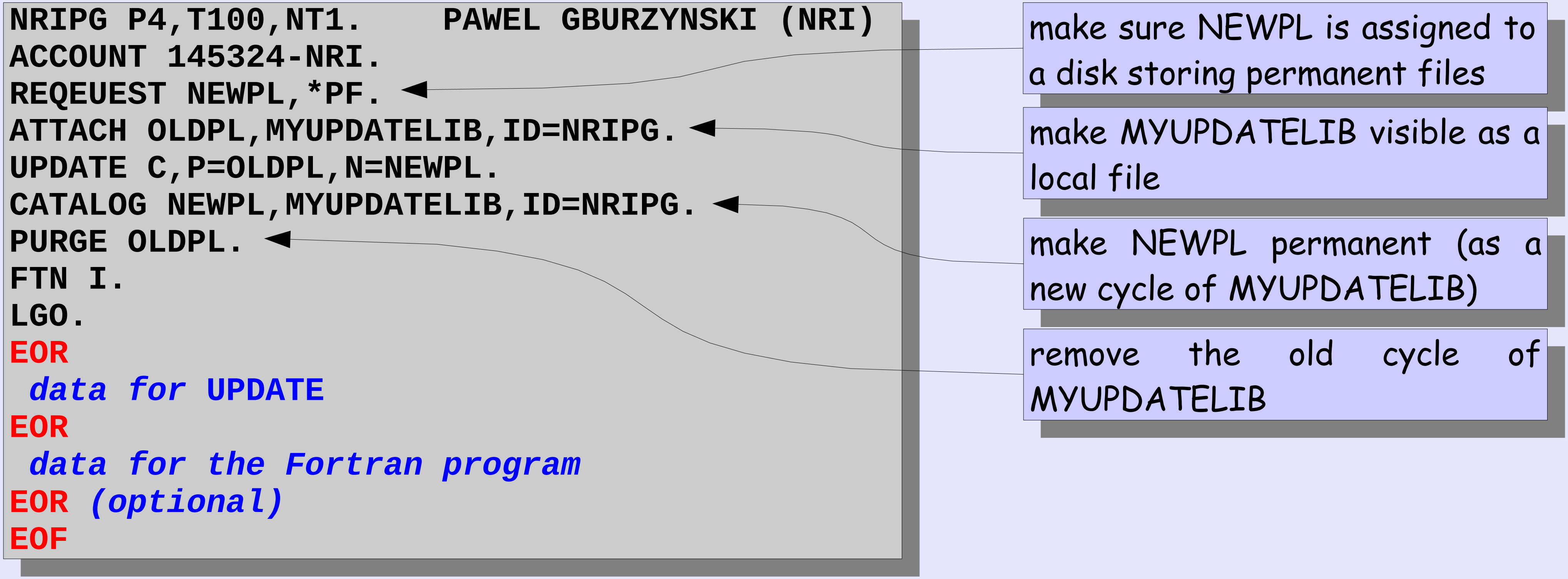
It was possible to DISPOSE (of) a local file, e.g., assigning it to a particular device.

The name of a local file was up to 7 characters starting with a letter. That name was strictly local, i.e., invisible outside the job.



Permanent files

If a file was supposed to stay after the job ended, you had to make it permanent:



```
NRIPG P4,T100,NT1.     PAWEL GBURZYNSKI (NRI)
ACCOUNT 145324-NRI.
REQEUEST NEWPL,*PF.
ATTACH OLDPL,MYUPDATELIB,ID=NRIPG.
UPDATE C,P=OLDPL,N=NEWPL.
CATALOG NEWPL,MYUPDATELIB,ID=NRIPG.
PURGE OLDPL.
FTN I.
LGO.
EOR
 data for UPDATE
EOR
 data for the Fortran program
EOR (optional)
EOF
```

make sure NEWPL is assigned to a disk storing permanent files

make MYUPDATELIB visible as a local file

make NEWPL permanent (as a new cycle of MYUPDATELIB)

remove the old cycle of MYUPDATELIB

The directory of permanent files was essentially flat. The role of ID was to provide for some sense of privacy, although it was quite illusory.

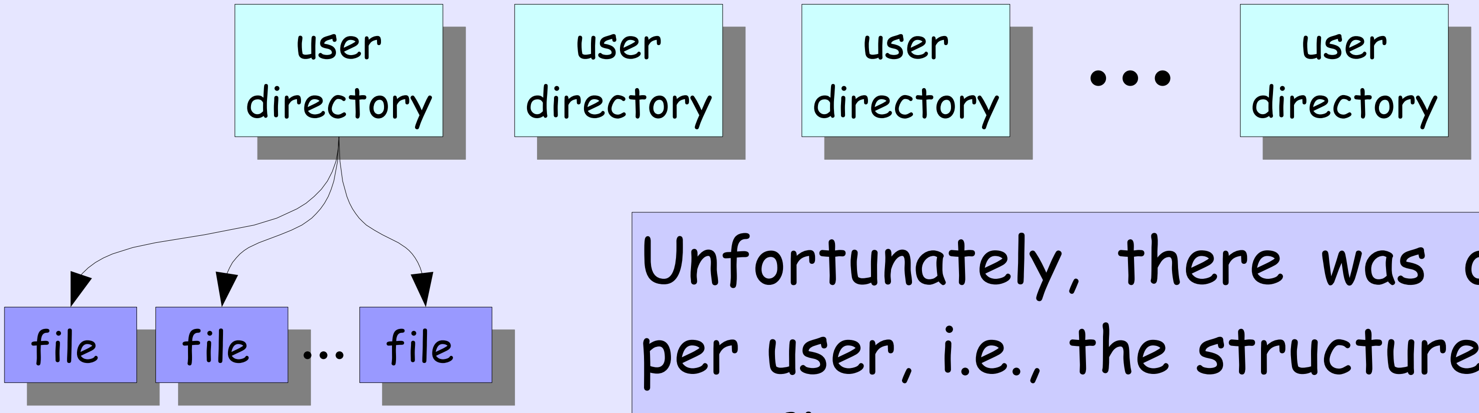
Early hierarchies

Note that on a batch system, the notion of a user is less clear than in an interactive (time-sharing) system.

With the advent of time sharing, it became natural to:

- make files permanent by default
- make them private by default as well

For example, under MTS (present on campus until early 90's):



Unfortunately, there was only one level per user, i.e., the structure of user files was flat.

File access methods

How would you like to perceive the information stored in a file?

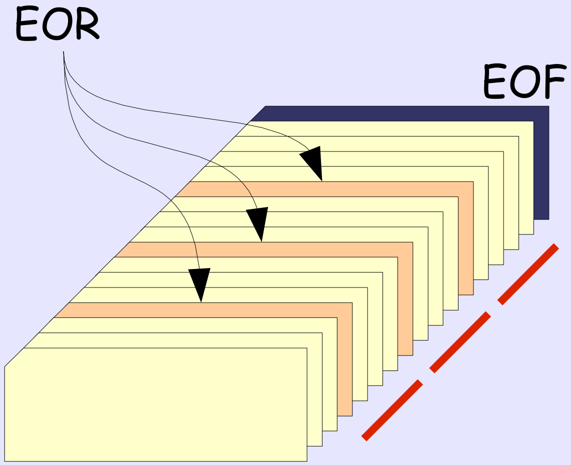
sequential versus direct access

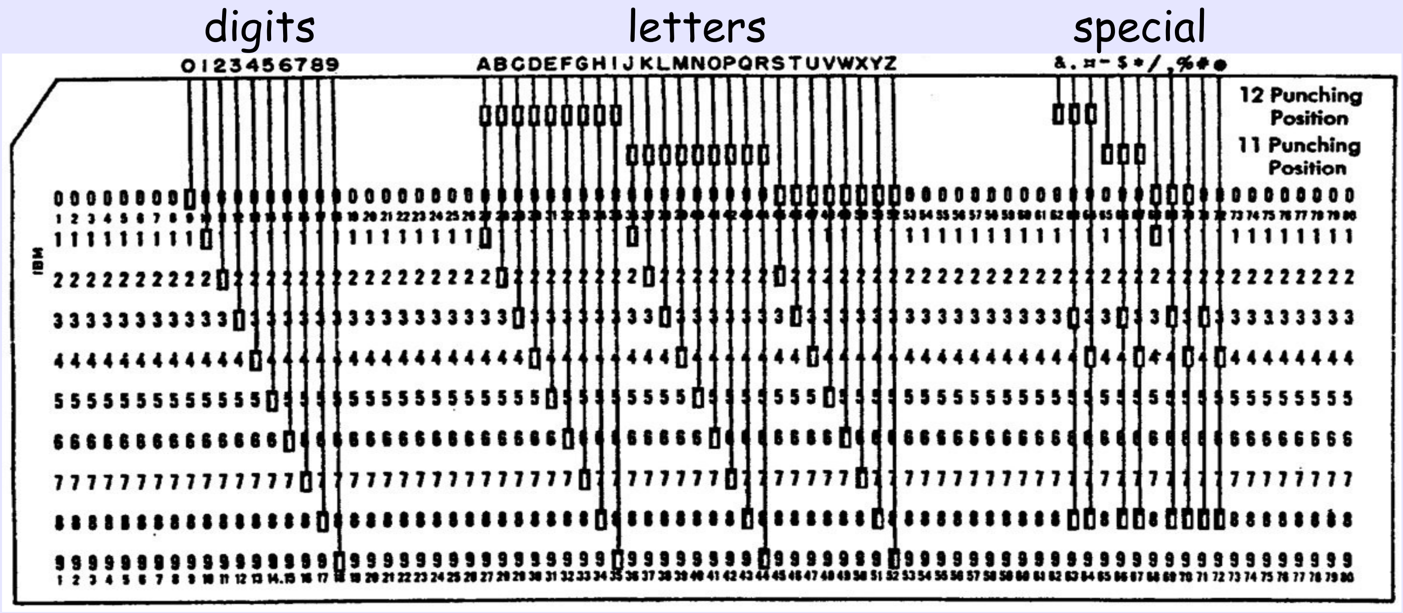
text versus binary

formatted versus unformatted

Punched cards + the lack of an agreed upon standard for storing text files have influenced these issues for a long time, and quite unnecessarily so.

The role of early files was to approximate decks of cards:





Some exotic file attributes

information type: character/binary

record length:

- 80-character fixed
- variable with special markers at the end
- variable with the length field stored in the record

indexing:

- by record number
- by a key

character encoding: (lots of messy internal codes)

extensibility: how the file is allowed to grow/shrink

Example: indexed sequential files

```
00100          FUNCTION IFIBB (N)
00110          K=1
00112          IFIBB=0
00120          L=2
00130          DO 10 I=1,N
00140          IFIBB=K
00150          K=L
00160          L=IFIBB+K
00170    10    CONTINUE
00300          RETURN
00302          END
```

```
write   00112          IFIBB=0
write   00300          RETURN
write   00302          END
write   00120          L=2
```

Indexed sequential files ...

... were very popular on early (and not so early) timesharing systems as a standard representation of text for editing, especially in combination with various "intelligent" terminals (advanced by IBM).

A file like that was immediately accessible as a straightforward sequential file (with the ordering of records determined by the number sequence).

At the same time, the very same file was directly editable.

Note that these days, your favorite text editor (emacs, VI) does not operate on the original file. It reads the file into an internal "buffer" and then copies the buffer (saves your changes) onto the original.

Direct access files

In addition to implementing indexed-sequential files, systems tried to cater to databases, which required various types of direct-access files. Typical operations:

```
write (file, key, record);
read (file, key, record_buffer);
```

where key was numerical or textual, fixed or variable length. Similarly, the record length was fixed or variable, and so on – think of all the combinations.

You may know (e.g., from 204 and 291) that there are different ways to implement record indexing, depending on the application. Early systems often tried to implement as much of them as possible internally.

These days, e.g., on UNIX systems ...

... there is just one type of files. A file is a general-purpose sequence of bytes.

Whenever a file is opened, its position is set to byte number zero.

Whenever *k* bytes are read or written, the file position is advanced by *k* .

Random access is implemented by making it possible to change the file position explicitly.

```
off_t   lseek ( int file_descriptor, off_t offset, int whence);
```

SEEK_SET	*offset* bytes from the beginning of file
SEEK_CUR	*offset* bytes from current position
SEEK_END	*offset* bytes from the end of file

Examples

```
lseek (myfd, 0, SEEK_SET);
```

sets the position to 0, i.e., rewinds the file

```
lseek (myfd, -64, SEEK_END);
```

sets the position to 64 bytes before the last byte of the file

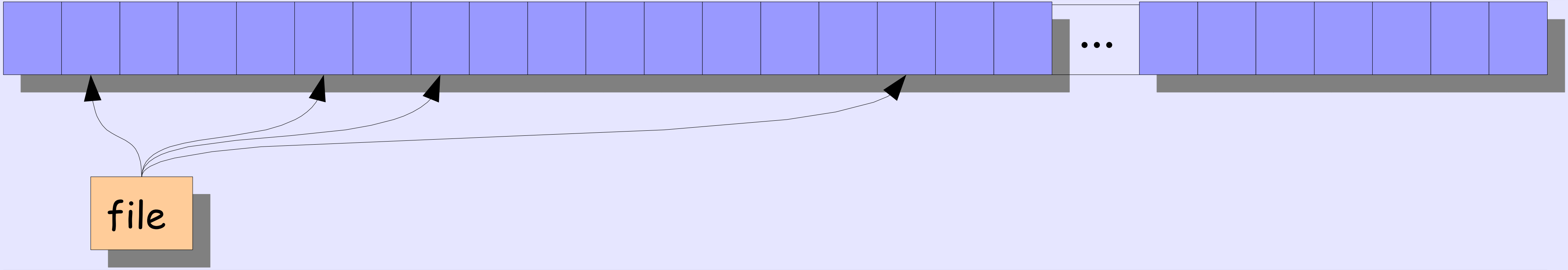
```
lseek (myfd, 16, SEEK_CUR);
```

advances the position 16 bytes forward

Note that seeking behind the last byte actually present in the file is legal. If you do that and write something, you will create a hole. No storage will be formally allocated to the hole, and if you read those bytes, you will get zeros.

Representing files

A disk can be envisioned as a simple array of sectors. Every sector is directly accessible, independently of other sectors.

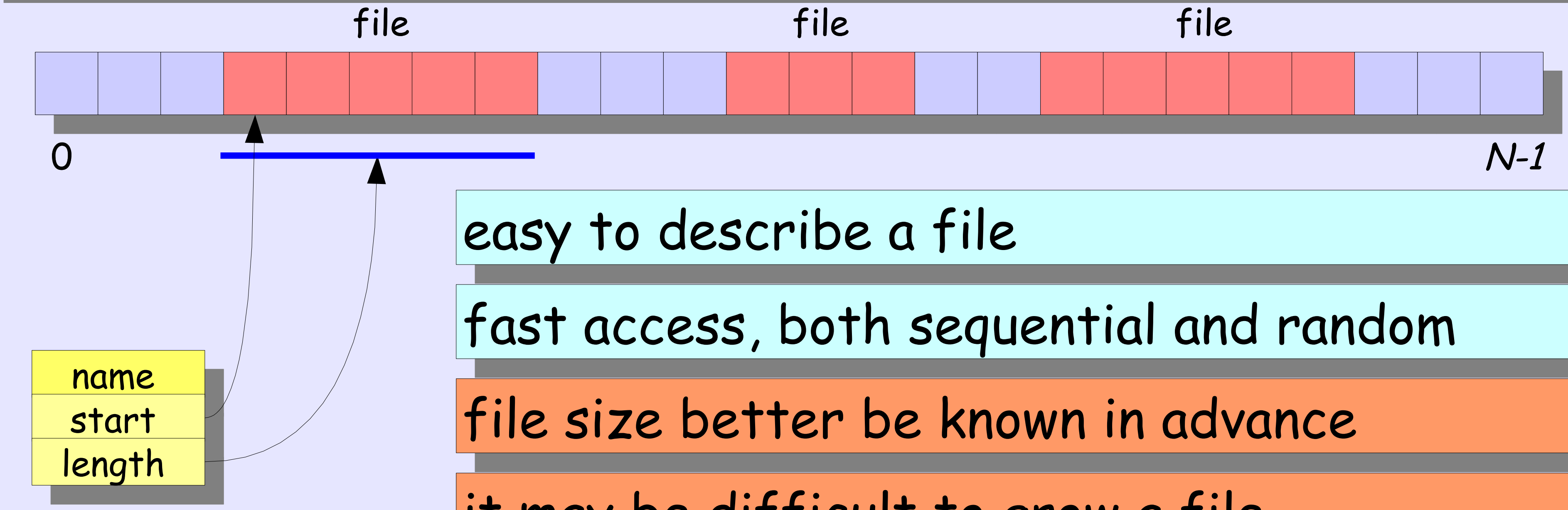


Files are sequences (or sets) of sectors described in some data structures. Ideally, those data structures should facilitate:

- fast location of subsequent sectors for sequential access
- fast location of any sectors for random access
- efficient file growth and shrinkage
- efficient usage of disk storage – no fragmentation
- reliability: low risk of serious damage from a minor malfunction

A few simple ideas

Continuous allocation: a file occupies a continuous range of sectors on disk.



- easy to describe a file
- fast access, both sequential and random
- file size better be known in advance
- it may be difficult to grow a file
- fragmentation

Naïve as it appears, this approach is quite popular in some applications. A program may receive a disk partition to implement its own "file" covering a consecutive range of sectors.

Disk partitions

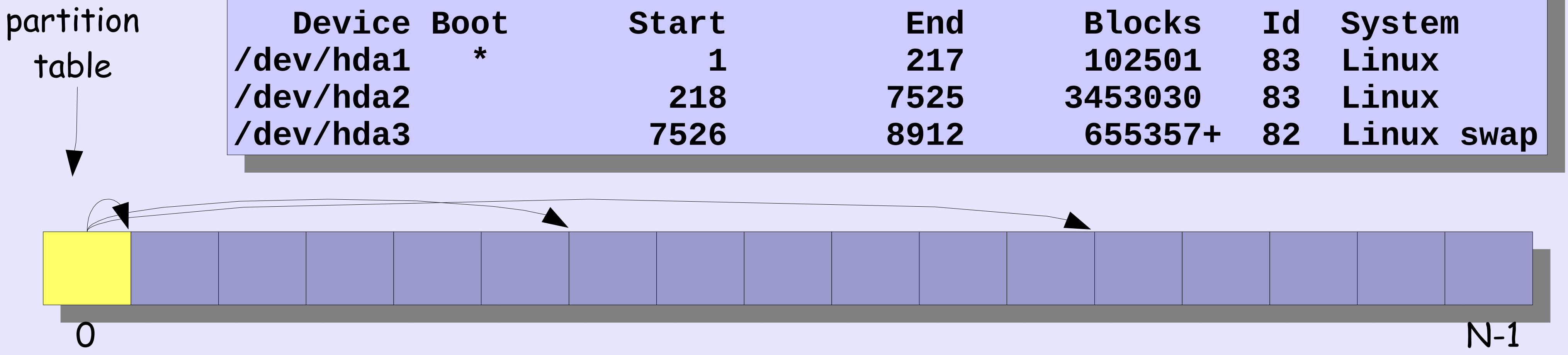
Typically, a disk is partitioned (sometimes trivially, i.e., into a single partition) before a file system is put on it.

You may want to have several file systems on the disk (possibly of different types).

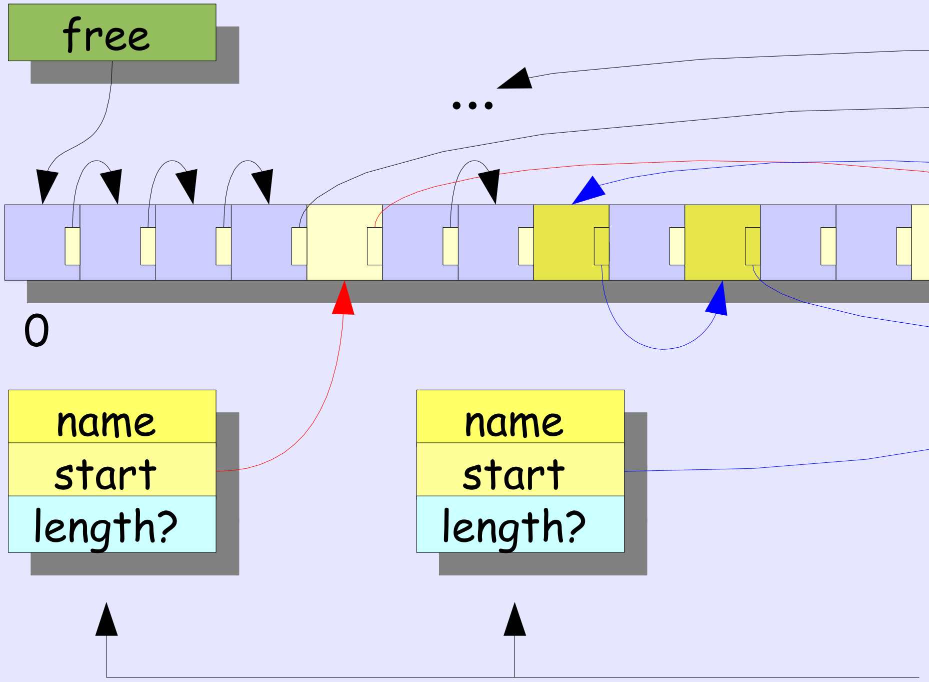
Some programs (e.g., high-performance databases) may prefer to operate on raw partitions and implement their own filesystems.

```
/sbin/fdisk  /dev/hda
```

```
Disk /dev/hda: 4311 MB, 4311982080 bytes
15 heads, 63 sectors/track, 8912 cylinders
Units = cylinders of 945 * 512 = 483840 bytes

   Device Boot       Start          End        Blocks   Id  System
/dev/hda1   *            1          217        102501   83  Linux
/dev/hda2              218         7525       3453030   83  Linux
/dev/hda3             7526         8912        655357+  82  Linux swap
```



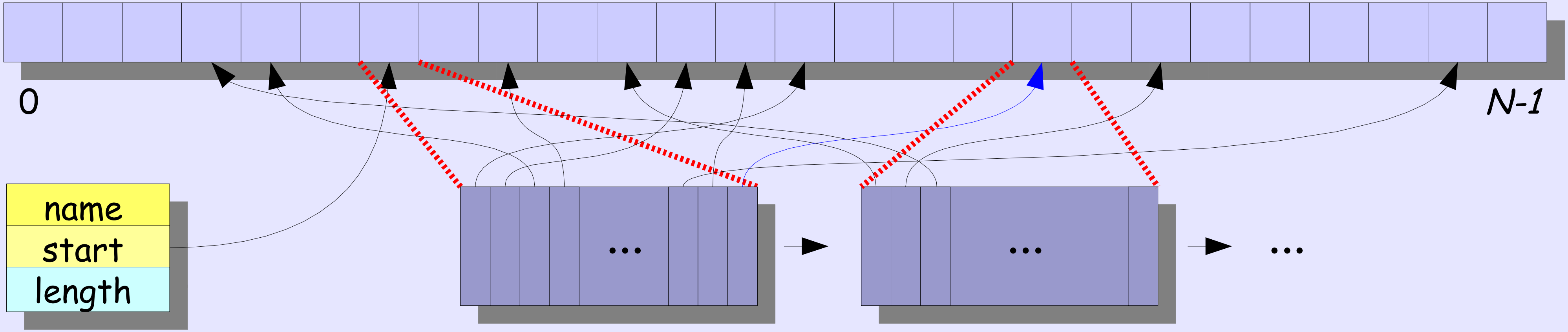
Linked allocation



We may need room to store these, e.g. in a reserved range of sectors at the beginning of the partition

One obvious problem: random access is not much fun. To read a given block, you have to read all the blocks in front of it.

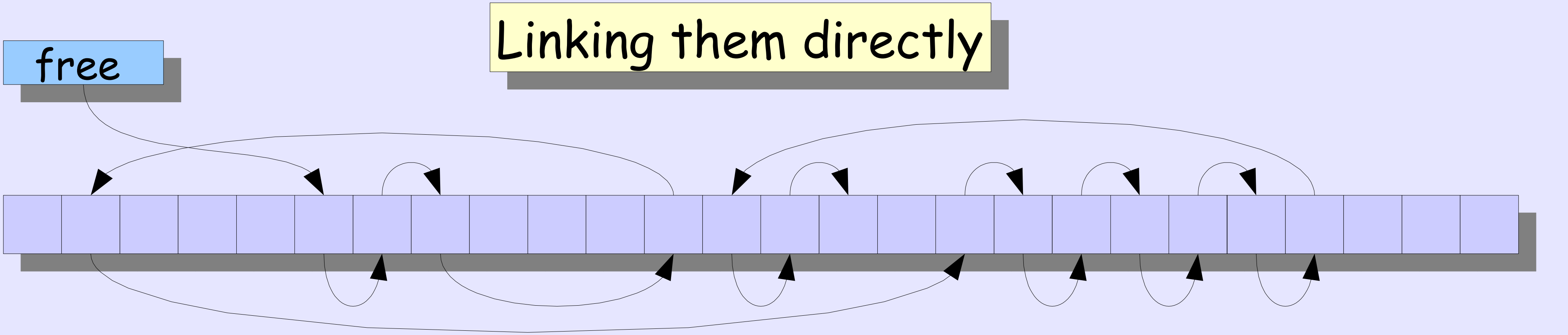
Separating links from blocks



This is better and may even be acceptable. Note that the allocation block size is flexible (within reason), i.e., it can be any multiple of the sector size. Suppose the block size is 512 bytes and pointer size is 32 bits.

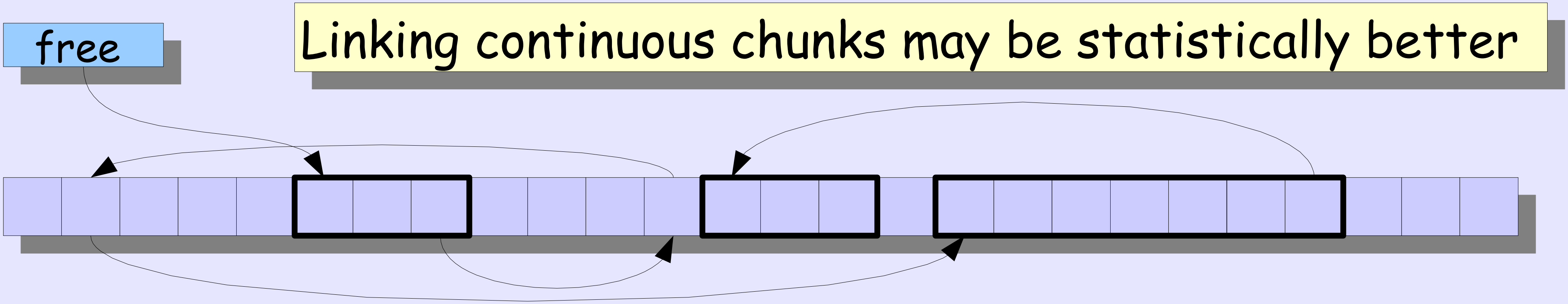
maximum disk (partition) size:	4G x 512 = 2TB
overhead:	1/127 < 1%
random access cost:	N/(512 x 127) extra reads

Locating free blocks


Linking them directly

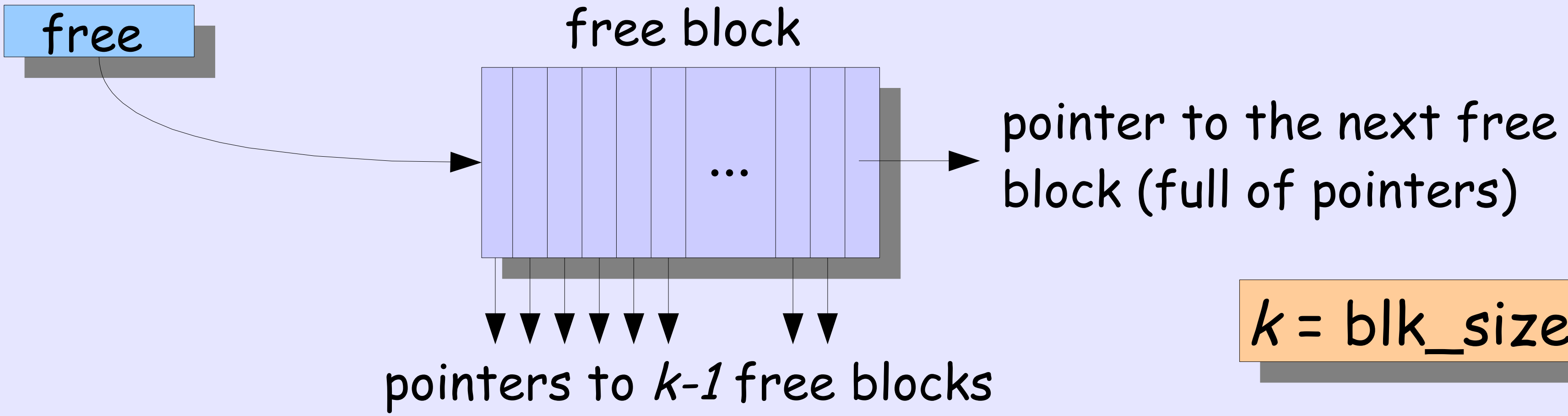
Note that this way, as a new file is being written, we effectively need two I/O operations per each block:

- to update the free list
- to write the actual data block.


Linking continuous chunks may be statistically better

Even better

A free block may contain pointers to many other free blocks, not just the next one on the free list.


free
free block
...
pointer to the next free block (full of pointers)
pointers to *k-1* free blocks

k = blk_size/ptr_size

The system keeps one block of pointers in memory. When it runs out of pointers, it uses the block itself (which is now empty) and reads the next block of pointers from the free list.

As free blocks are returned, the system fills up the current block, then flushes it to the disk, and opens a new one storing in it the pointer to the just flushed block.

Bit maps

Each allocatable block has one status bit in the bit map. If the status bit is 1, it means that the block is empty (or perhaps full).

The bit maps must be eventually written to the disk, but the system can cache them in memory. Otherwise, a block acquisition/release would always require an extra I/O operation to update a bit in the map.

We can see two tradeoffs:

Block size

smaller blocks -> less fragmentation, but more complexity and higher access time

Sync intervals

delayed updates -> less overhead, but a higher likelihood of a disastrous damage

What is the best block size?

file size	#files	%files	%cumm	space	%space	%cumm
0	147479	1.2	1.2	0.0	0.0	0.0
1	3288	0.0	1.2	0.0	0.0	0.0
2	5740	0.0	1.3	0.0	0.0	0.0
4	10234	0.1	1.4	0.0	0.0	0.0
8	21217	0.2	1.5	0.1	0.0	0.0
16	67144	0.6	2.1	0.9	0.0	0.0
32	231970	1.9	4.0	5.8	0.0	0.0
64	282079	2.3	6.3	14.3	0.0	0.0
128	278731	2.3	8.6	26.1	0.0	0.0
2097152	16140	0.1	99.9	23230.4	9.0	63.5
4194304	7221	0.1	100.0	20850.3	8.0	71.5
8388608	2475	0.0	100.0	14042.0	5.4	77.0
16777216	991	0.0	100.0	11378.8	4.4	81.3
33554432	479	0.0	100.0	11456.1	4.4	85.8
67108864	258	0.0	100.0	12555.9	4.8	90.6
134217728	61	0.0	100.0	5633.3	2.2	92.8
268435456	29	0.0	100.0	5649.2	2.2	95.0
536870912	12	0.0	100.0	4419.1	1.7	96.7
1073741824	7	0.0	100.0	5004.5	1.9	98.6
2147483647	3	0.0	100.0	3620.8	1.4	100.0

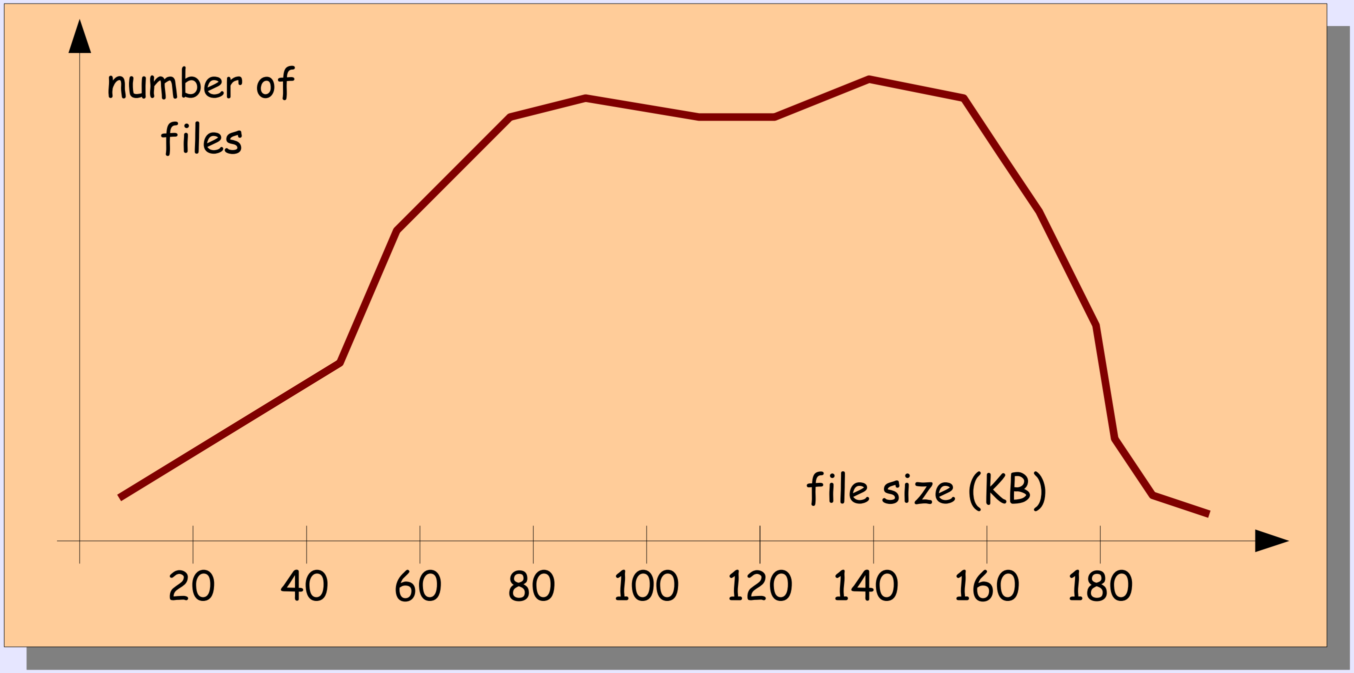
- the average file size is 22k
- pick a file at random: it is probably smaller than 2k
- pick a byte at random: it is probably in a file larger than 512k
- 89% of files take up 11% of the disk space
- 11% of files take up 89% of the disk space

These statistics of file sizes from a UNIX system were collected in 1993.

Clearly, there are many small files, quite a few of them being trivially small.

These trends tend to change with time

In an old batch system, these statistics might look as follows:



with very few small files (they were pointless in batch systems), and the typical file size around 100KB or so.

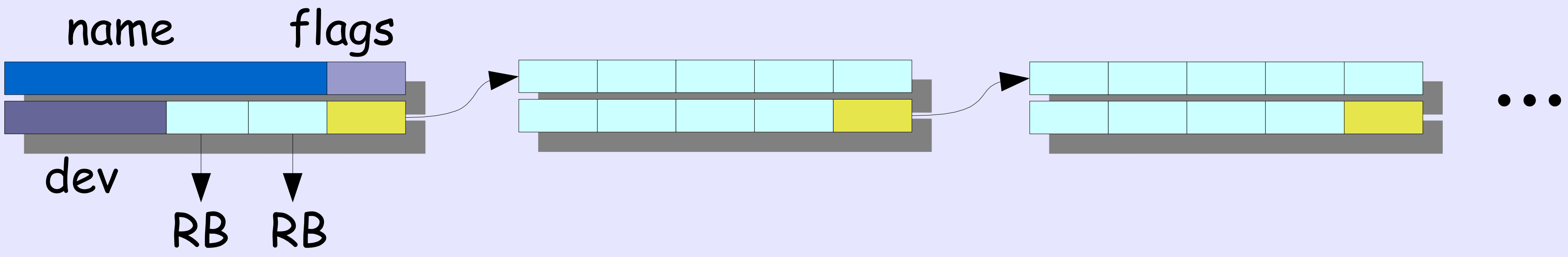
A file system designed for such an environment did not have to worry too much about fragmentation.

A case study: SCOPE 3.x.x

Sector (called PRU) size:	480 bytes (64 60-bit words)
Block (called RB) size:	70 sectors (PRU) = 33,600 bytes

A 1MB file (considered rather large) takes only 32 blocks.

A bit map for a 300MB disk (considered humongous) is merely 9400 bits, which translates into 157 60-bit words.



Nine RB's are described with 2 words, i.e., 120 bits. The complete set of links for a 300MB disk takes about (9400/9)x2 = 2089 words. Note that all those links can be kept (cached) in memory.

Note that…

… with the concept of local files, which are not expected to survive the end of job, the links need not ever be written to the disk! Thus, the whole structure of the "local" file system is kept in memory! The disk is just a raw collection of blocks.

Unfortunately, this idea is completely worthless for a contemporary system.

For a 2KB file, the fragmentation is (33,600-2048)/33,600=94% !

Disk are much larger. This also affects, e.g., the pointer size.

There are no jobs, and "local" files make no sense.

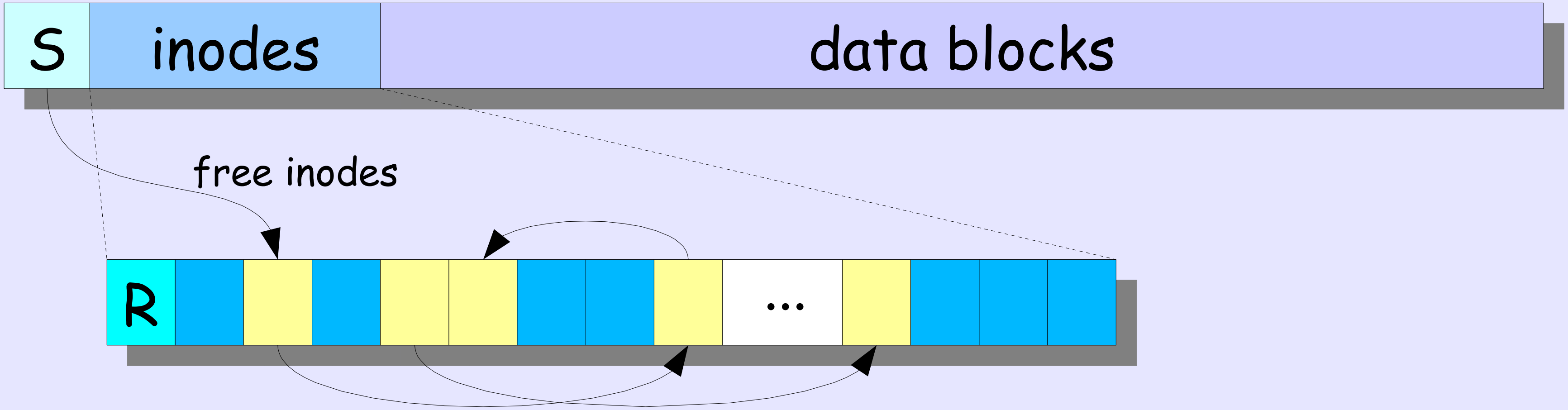
For example, a 10MB file, with 1K blocks and 4-byte pointers needs 40KB of pointer space alone.

Also, a bit map for a modest 40GB disk takes 40Mbits = 5MB.

The UNIX file system

Let us start from the somewhat aged original. Later, we will comment on contemporary modifications and departures.

The first block of the partition is called the superblock. It contains the parameters of the file system, e.g., the actual block size and the boundary between inodes and data blocks.



The inodes part is an array of entries of fixed size. Each entry describes one file (or is empty).

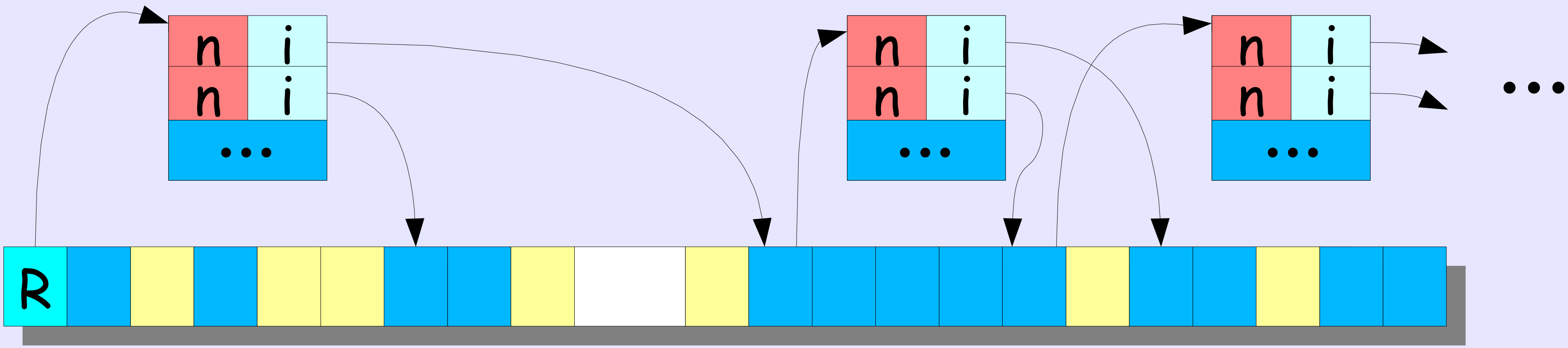
File identification

Inodes directly represent files. If we didn't mind the inconvenience, we could identify files by inode numbers. The system in fact does it internally. And you also can, if you are root.

The familiar tree-like hierarchy is imposed on the inodes by the following two properties:

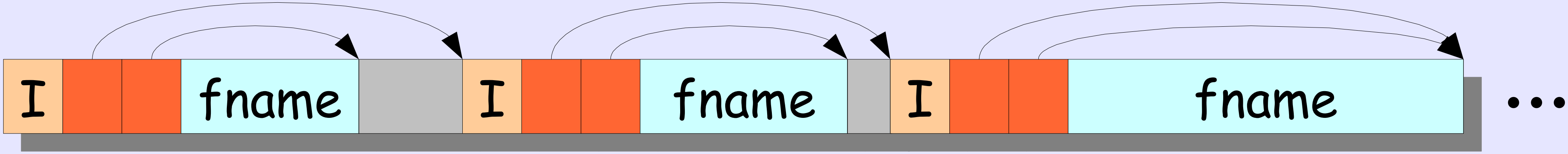
A file pointed to by an inode can be a directory. This is a special file that associates names with some inode numbers.

The root directory of the file system is stored in the file described by inode 0.



Directory structure

A directory entry contains precisely two items of information: file name + inode number. The entries are linked:

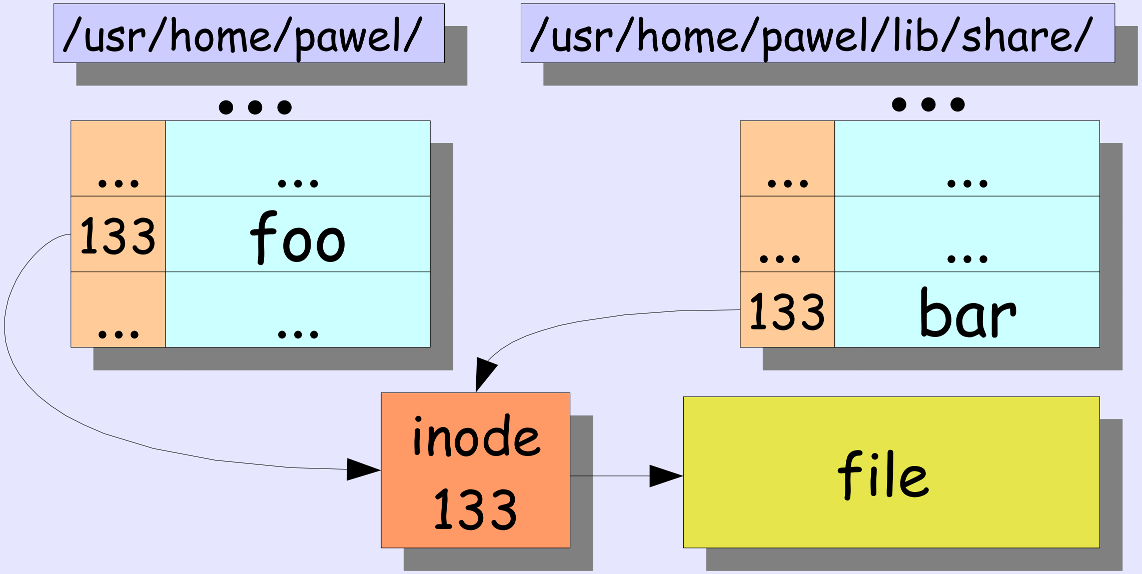


Note that some fragmentation problems may arise as files are created, destroyed, and the directory entries are recycled.

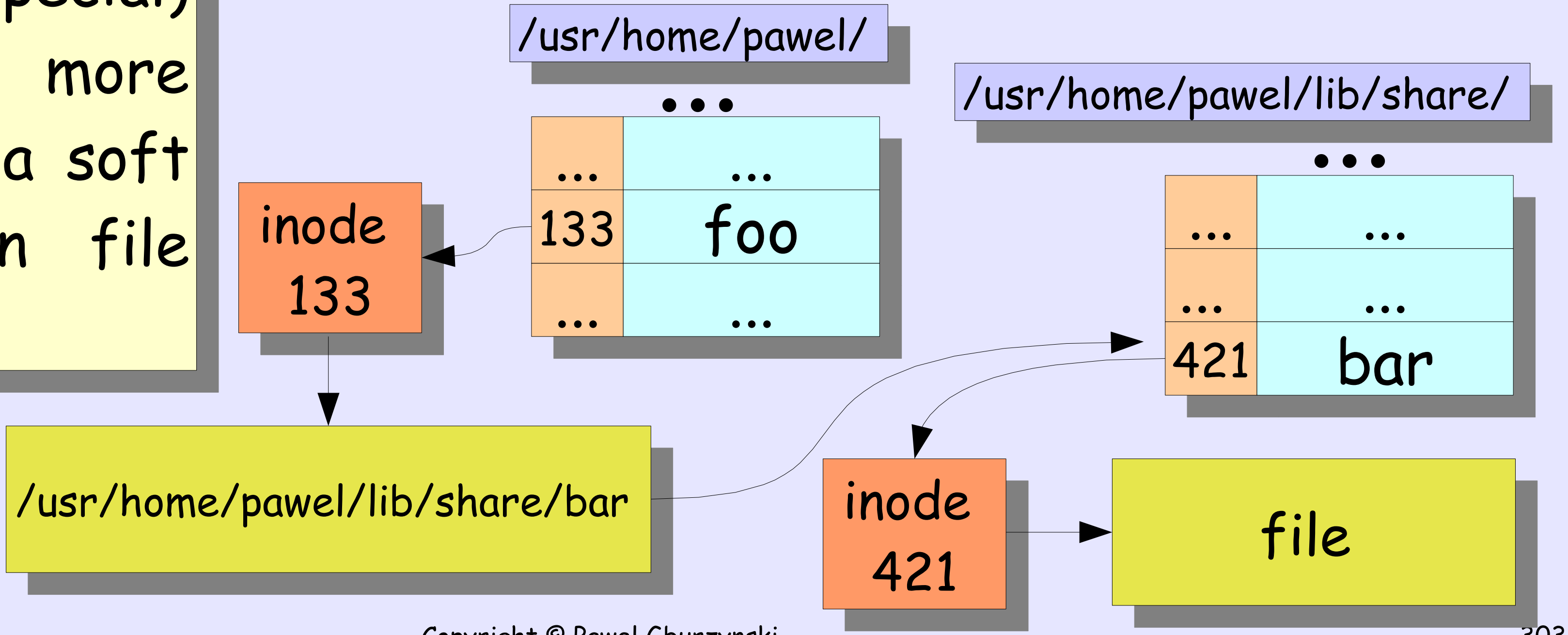
The important point is that a directory entry contains no information about the file. All that is kept in the inode. This brings about a few interesting features.

Links

Hard links: the same file is accessible under two different names, possibly at distant places in the hierarchy.



Soft links: the link is stored in a (special) file. This is more flexible, e.g., a soft link can span file systems.

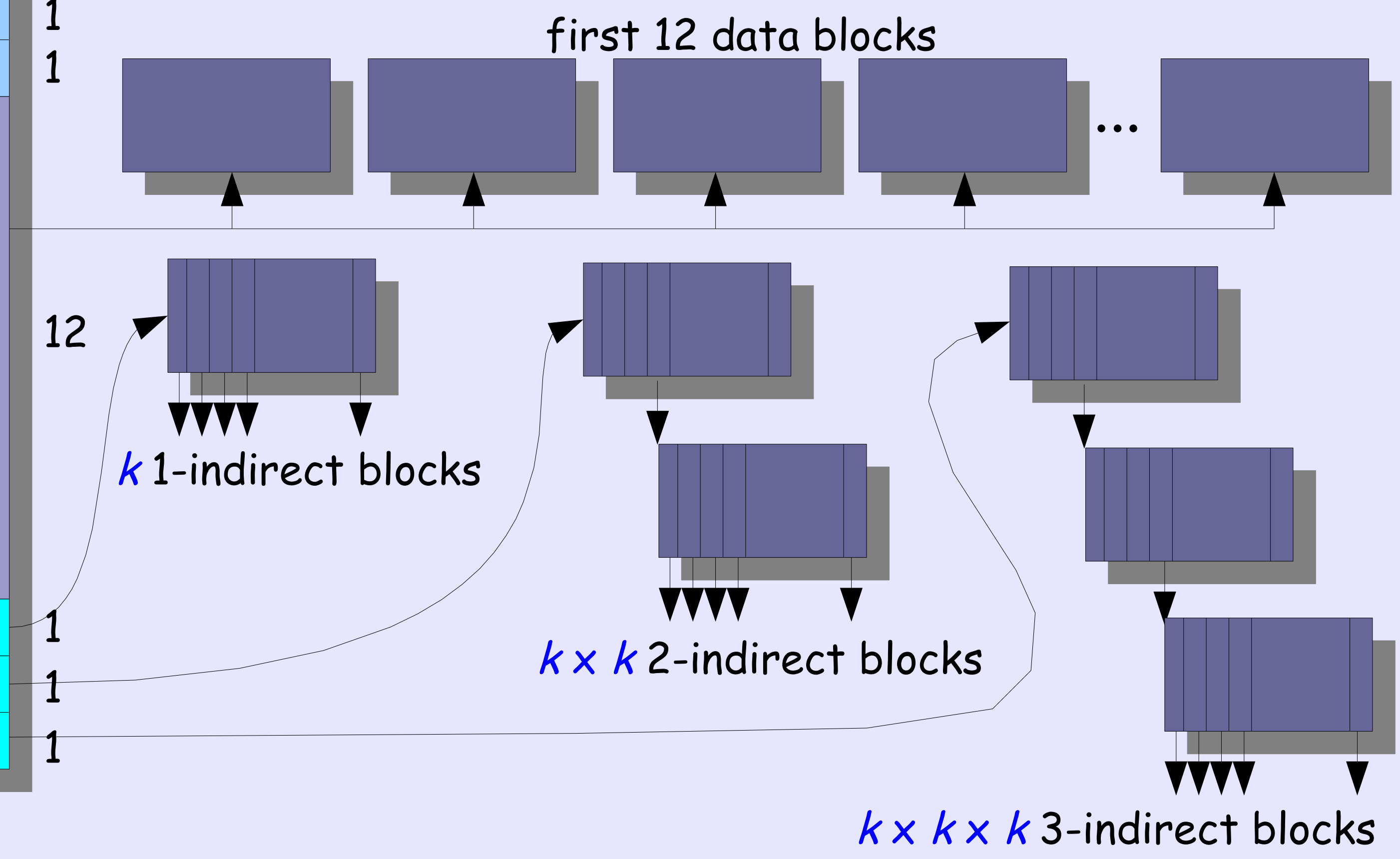


Copyright © Pawel Gburzynski

Inode layout + block list

reference count	1
mode	1
ownership: group & user	2
time stamps: created, modified, accessed	3
size	1
block count	1
direct block pointers	12
single indirect pointer	1
double indirect pointer	1
triple indirect pointer	1

```
ls -l
drwxr-xr-x 3 pawel users 4.0k Oct 25 13:30 379
drwx------ 3 pawel users 4.0k Nov 16 03:39 ARCHIVE
drwxr-xr-x 2 pawel users 4.0k Sep  5  2004 BIN
-rw-r--r-- 1 pawel users 5.9k Aug  8 16:44 Config
lrwxrwxrwx 1 pawel users   26 Apr 21  2005 FTP ->
                                    /usr/menaik3/ftp/pub/pawel
-rw------- 1 pawel users 3.3k Sep 29  2004 Mailbox
```



Maximum file size

Assume 512-byte blocks, with $k = 128$. We get:

$F_{max} = 12 \times 512 + 128 \times 512 + 128^2 \times 512 + 128^3 \times 512 > 1GB$

This doesn't look like infinity any more. Note that by switching to 1K blocks (still decent), we multiply this number by $2^4 = 16$.

Hollow files are nicely accommodated. For example, consider this:

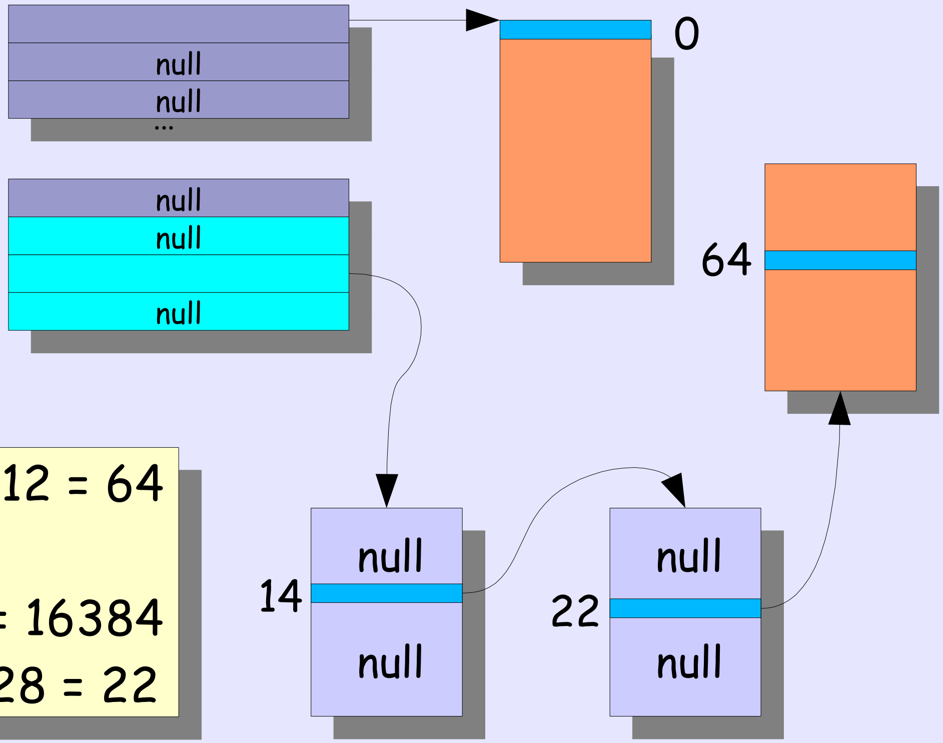
```
...
fd = open (“junk”, O_CREAT |
           O_WRONLY | O_TRUNC, 0660);
write (fd, &byte, 1);
lseek (fd, 1000000, SEEK_SET);
write (fd, &byte, 1);
close (fd);
...
```

1000001 - 12 x 512 = 993857 1000000 % 512 = 64

993857 / 512 = 1941.... = 1942 blocks

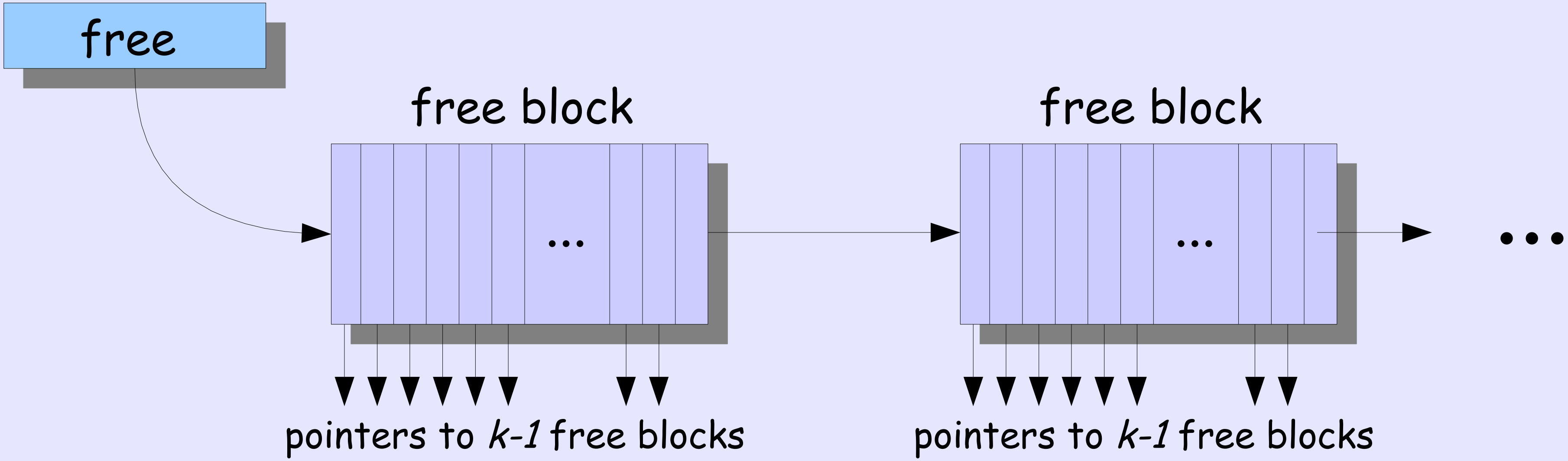
1942 - 128 = 1814 128 x 128 = 16384

1814 / 128 = 14.17... 1814 % 128 = 22



Free space management

The original system used a free list, like this one:



with the free pointer stored in the superblock (of course).

When the filesystem was mounted, the first free block was read into memory, providing immediate access to (up to) 127 + 1 free blocks.

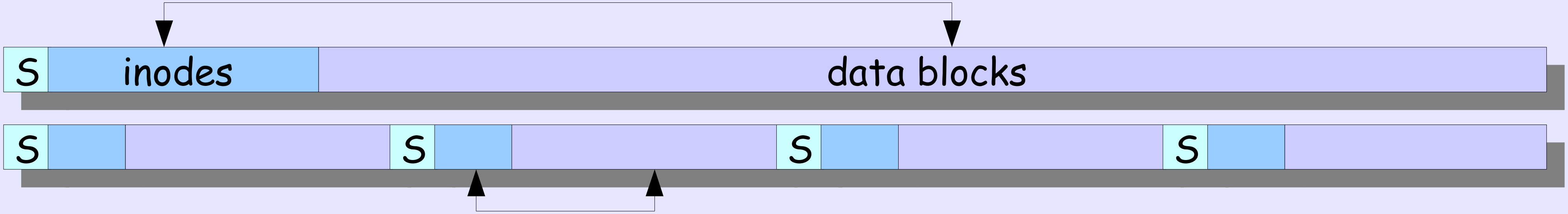
Modern enhancements

Dual block size

Low overhead for large files, low fragmentation for small ones.

Grouping

Splitting inodes into chunks and replicating the superblock.



Bitmaps

... instead of free lists. They are also grouped.

Journaling

Neat tricks to practically eliminate the possibility of any damage when somebody pulls the proverbial plug.

Fragmentation/overhead

Small blocks are truly needed for low fragmentation. Here are some statistics for a typical UNIX mix of files:

Used (MB)	Waste (%)	Blocking
775.2	0.0	Data, no separation
807.8	4.2	Data only, 512B
828.7	6.9	Data + inodes, 512B
866.5	11.8	Data + inodes, 1024B
948.5	22.4	Data + indoes, 2048B
1128.3	45.6	Data + inodes, 4096B

On the other hand, larger blocks reduce the overhead on disk transfers, and thus improve the performance of heavily I/O bound programs, like databases. Typically, by switching from 512B blocks to 4KB blocks, you cut the time component of I/O overhead by half.

Dual block size

1 block = *k* fragments, e.g., 4KB blocks, each consisting of four 1K fragments.

Bit maps are used for representing block status. Individual bits are applied to fragments rather than blocks, for example:

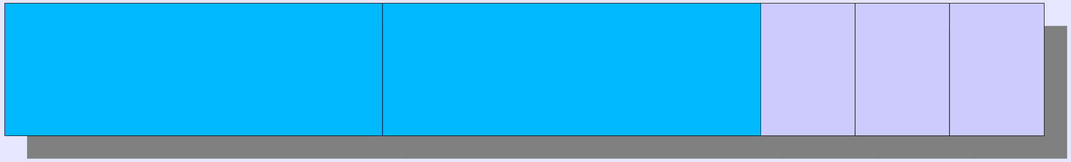
blocks	0	1	2	3
fragments	0-3	4-7	8-11	12-15
map bits	0000	0011	1100	1111

Suppose 1 means free. Only block 3 is free. Block 0 is used entirely. Blocks 1 and 2 are fragmented. They cannot be used as blocks, but their unused fragments are available for allocation to small files,

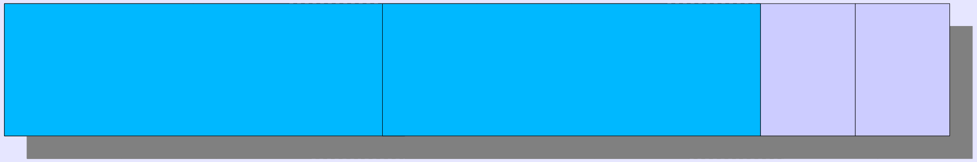
The trick

There is no difference between a large file and a small one (in its representation on the disk). Any file consists of a number of blocks + possibly a number of fragments (never more than k - 1) at the end.

For example, assume that the file length is 11,000 bytes. The file takes two blocks and three fragments.



11,000 = 2 x 4096 + 2 x 1024 + 760



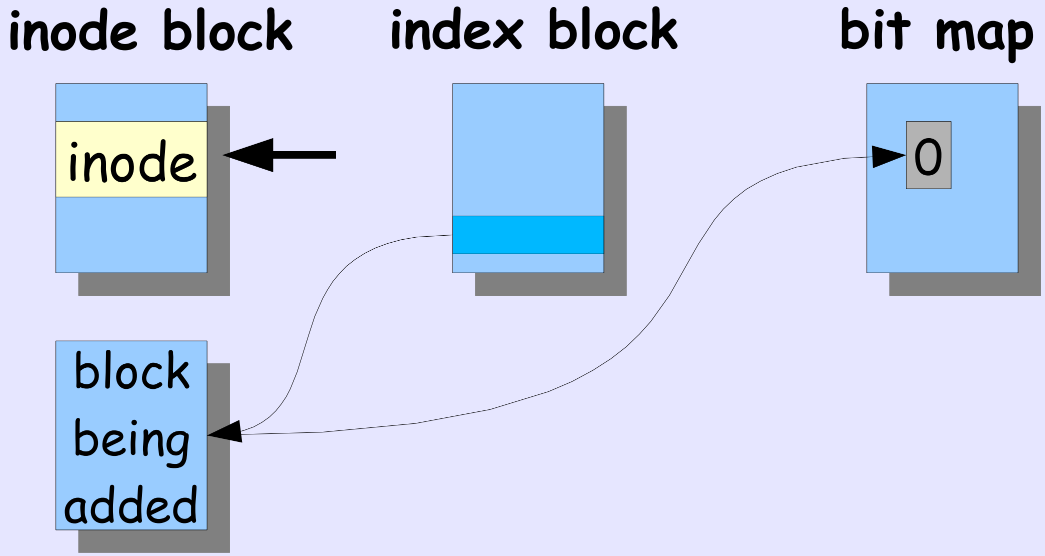
...

The important fact is that just by looking at the file size we know which pointers point to blocks and which point to fragments.

Copying can be avoided, if the system allocates consecutive fragments from full blocks.

Journaling

Suppose that a block is to be added to a file. These are the steps involved in the operation:



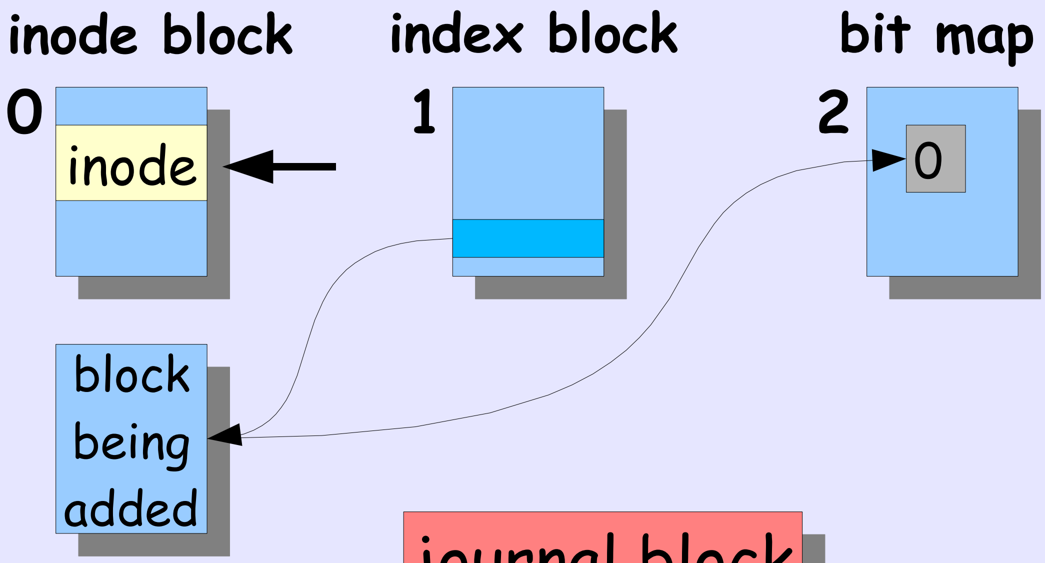
Add the block pointer to the index block

Change the block status in the bitmap

Update size in the inode

Three blocks must be written to disk to complete the operation, i.e., to make the disk structure fully consistent with the changes. If the system crashes after the first write but before the completion of the last one, we may be in trouble.

Similar to transaction roll-back in DB



journal block

To modify:

Block 1, bytes ... should be ...

Block 2, bytes ... should be ...

Block 0, bytes ... should be ...

Store the journal entry

Add the block pointer to the index block

Change the block status in the bitmap

Update size in the inode

Remove the journal entry

System consistency does not depend upon a success of multiple writes to the disk. After every single write, the system either is consistent, or is able to complete the unfinished transaction.

Extents

I told you that the simple idea of continuous allocation is too valuable to be discarded as naive. Perhaps we shouldn't use it directly, but it may make sense to build files of continuous chunks (block ranges) called extents.

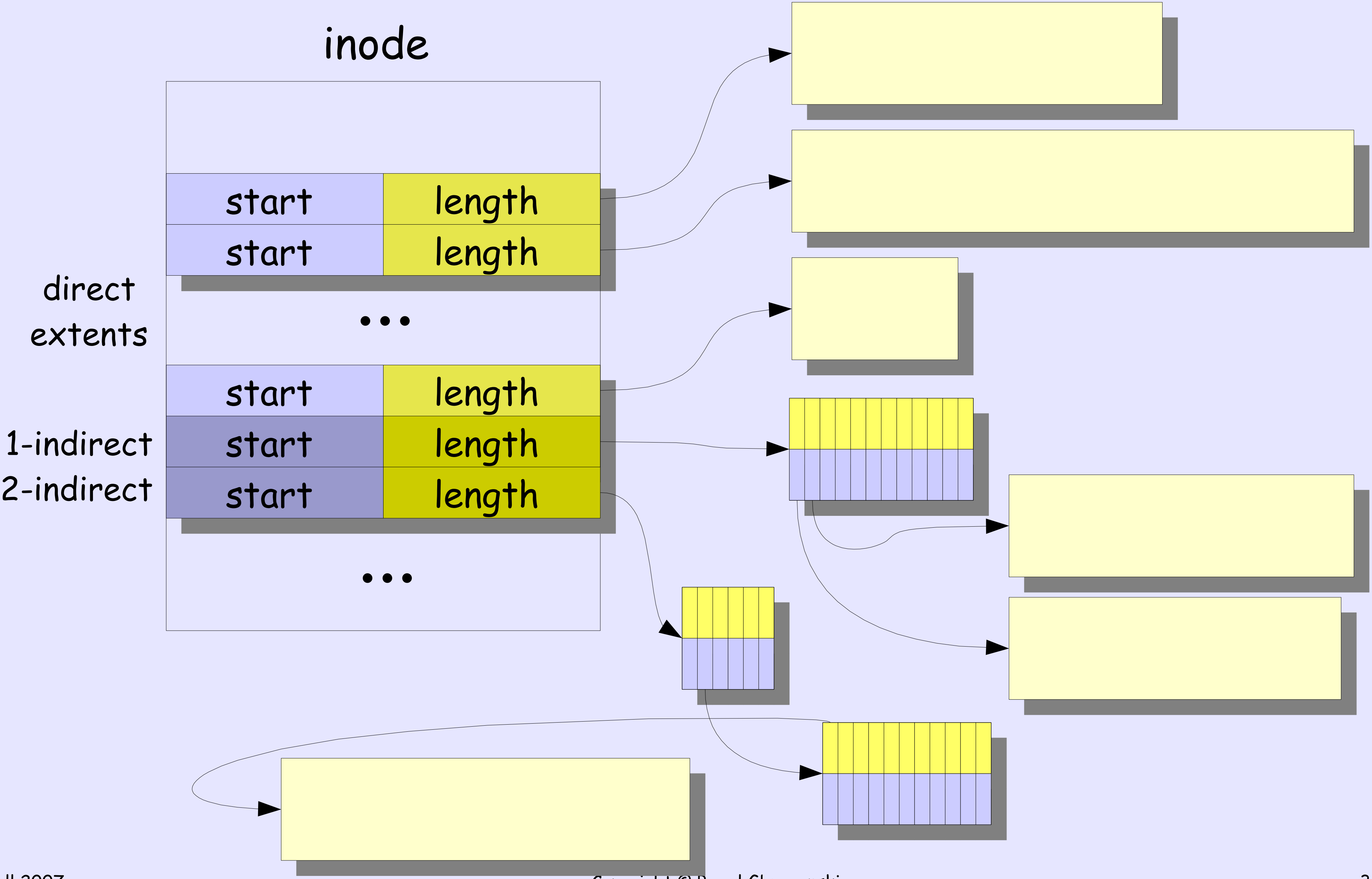
Statistically, even a large file may need just a few extents.

If long extents tend to be frequent, a file's description will tend to be short.

This means that access time to the file will tend to improve – both for sequential and random operation.

Also, describing free storage with extents may be more efficient (fewer things to keep track of, faster location of free space).

For example


inode
start
length
start
length
direct
extents
...
start
length
1-indirect
start
length
2-indirect
start
length
...

Comments

Holes cause no problem: they are taken care of by empty extents with start == NULL.

One problem is the complication of random access to distant blocks.

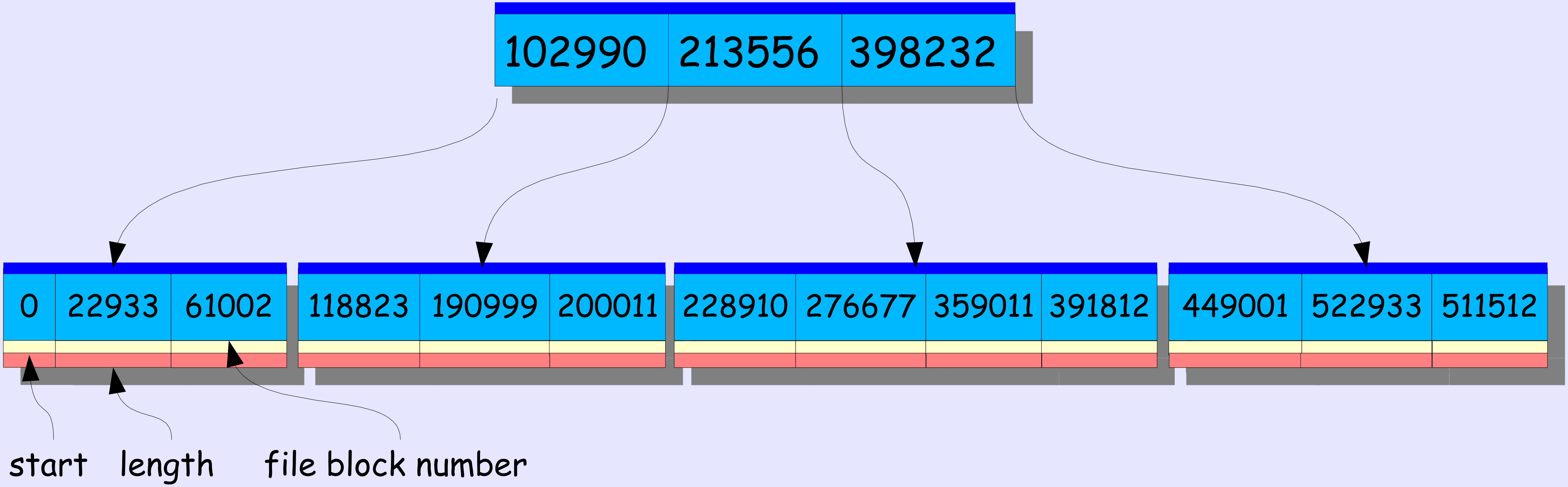
But we hope that the extents will tend to be large, so there will be few of them, so traversal of index extents will be infrequent.

Caching will also help, assuming that the "random references" are not completely random.

Besides, there are other (more efficient from the viewpoint of random access) ways of representing series of extents than the straightforward extrapolation of UNIX indexes and index blocks.

B-Trees

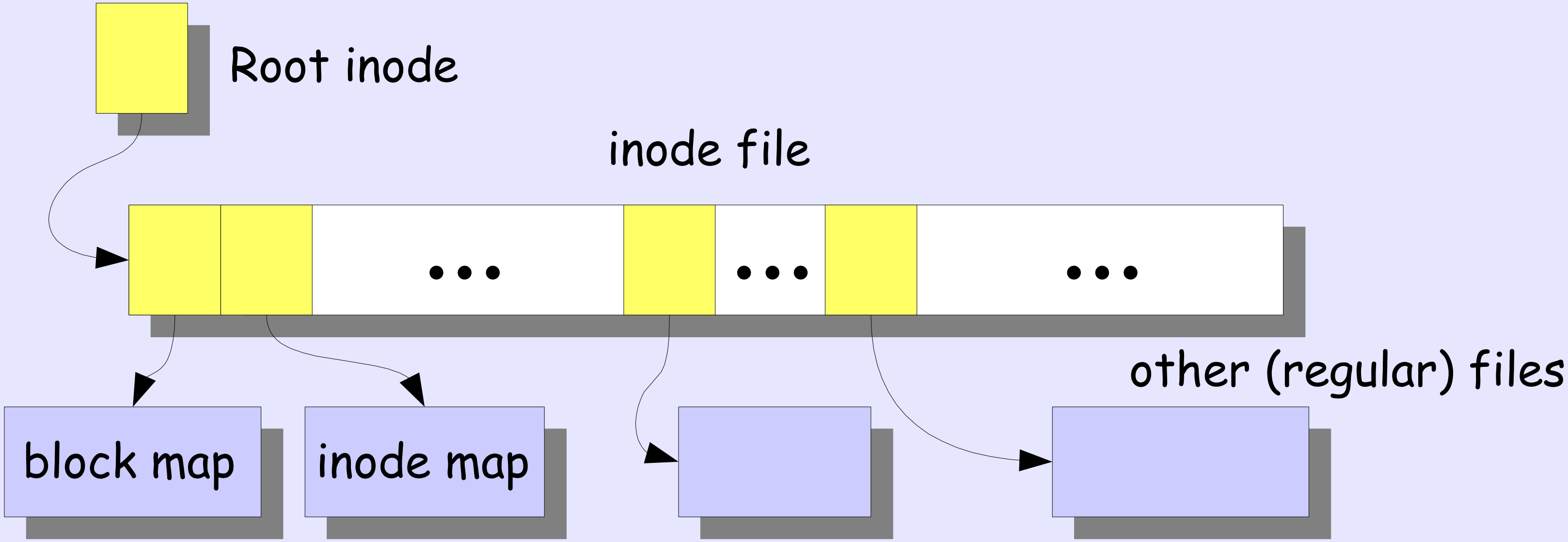
You know what they are if you have taken a database course. It used to be a prerequisite for 379.



A file can be viewed as a database of extents with block numbers used as keys.

Other ideas: WAFL

Write Anywhere File Layout. All metadata is kept in files:

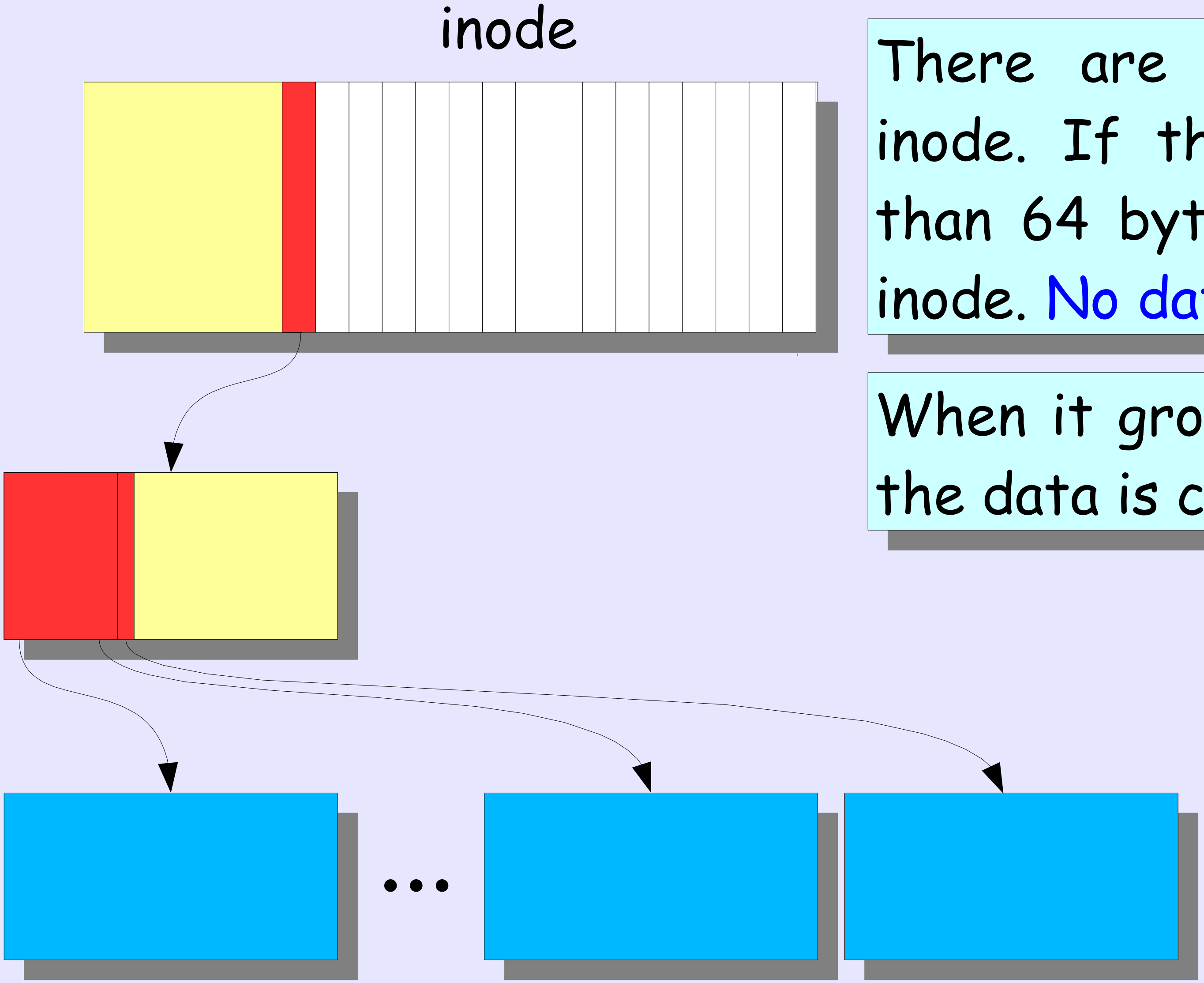


The entire file system forms a tree of blocks rooted at the single (Root) inode. This means that:

No blocks are dedicated to inodes, and there is no need to draw any lines.

An arbitrary number of filesystems can coexist on the same partition.

All pointers in inode are the same level



There are 16 pointers in the inode. If the file is not larger than 64 bytes, it is kept in the inode. No data blocks are needed!

When it grows beyond 64 bytes, the data is copied to a block.

Whenever the maximum file size at the current level is exceeded, the inode pointers are copied to a new index block and a new level is started.

Snapshots

One neat feature of WAFL is the possibility to create quick "snapshots" of the entire filesystem. To create a snapshot, you:

1. duplicate the root inode
2. enter a COW mode

A bit map entry for a block consists of an entire (32-bit) word. Here's is how such an entry may evolve:

Time	Map entry	Description
T1	...000000	Free
T2	...000001	Allocated to active FS
T3	...000011	Snapshot 1 created
T4	...000111	Snapshot 2 created
T5	...000110	Deleted from active FS
T6	...000110	Snapshot 3 created
T7	...000100	Snapshot 1 deleted
T8	...000000	Snapshot 2 deleted

The block becomes free:

Note that when an index block is marked as belonging to several snapshots, all blocks that fall under it are implicitly assumed to belong to the same snapshots.

Disk scheduling

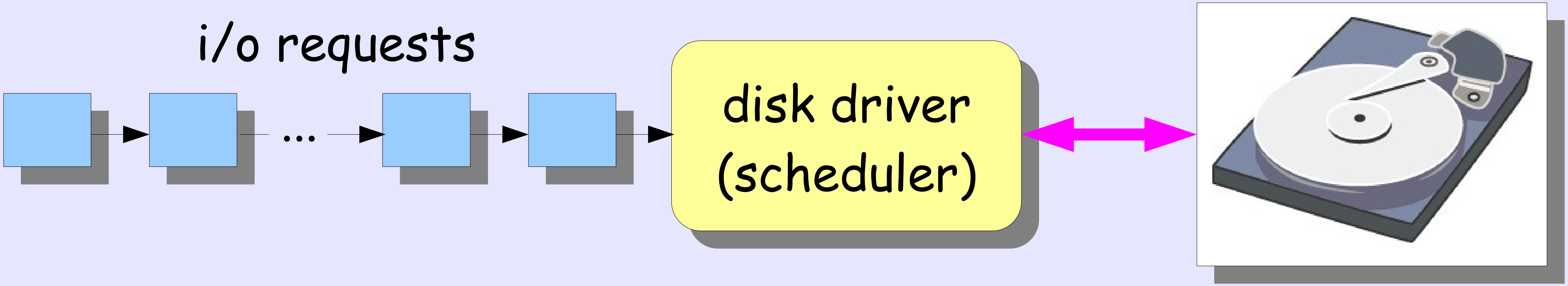
Disk I/O requests may arrive from several places at pretty much the same time (in bursts), occasionally faster that they can be handled by the device. This is quite typical because of:

multiprogramming: multiple programs doing I/O concurrently

paging: programs don't have to do explicit I/O to trigger I/O

caching: I/O can happen independently of programs' intentions

various performance enhancing tricks (like read-ahead) contribute to this picture



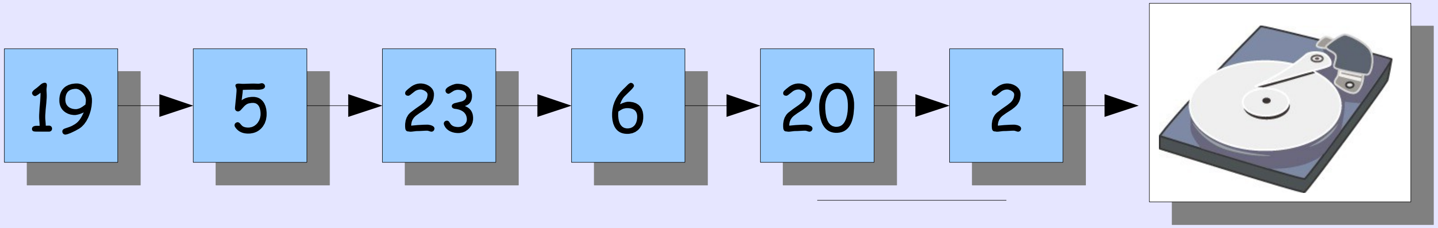
i/o requests

...

disk driver (scheduler)

What is there to optimize?

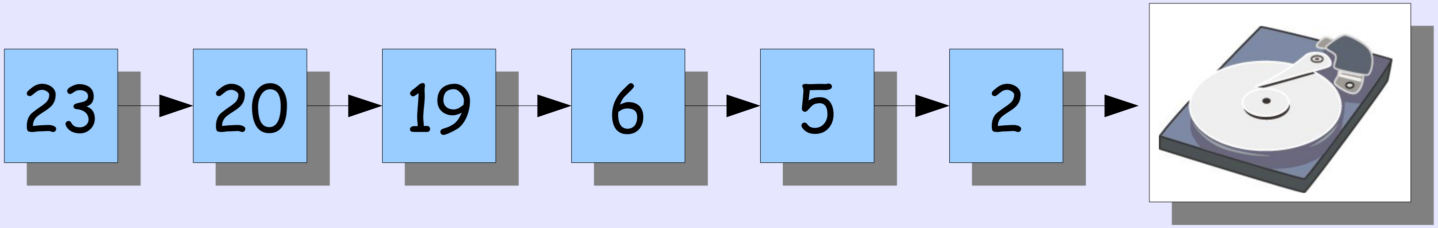
The seek time, of course. Suppose you have a disk with 24 cylinders. Here is a sample sequence of I/O requests:



19 → 5 → 23 → 6 → 20 → 2 →

Total seek distance (TSD) =

(20-2)+(20-6)+(23-6)+(23-5)+(19-5) = 81.



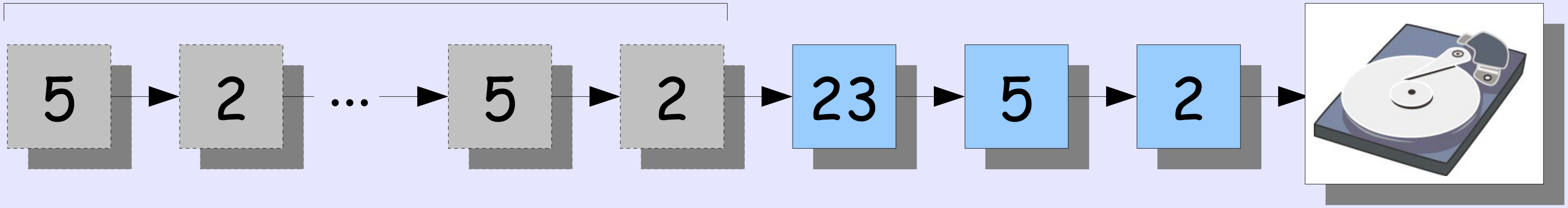
23 → 20 → 19 → 6 → 5 → 2 →

TSD = (5-2)+(6-5)+(19-6)+(20-19)+(23-20) = 21.

SSF: Shortest Seek First

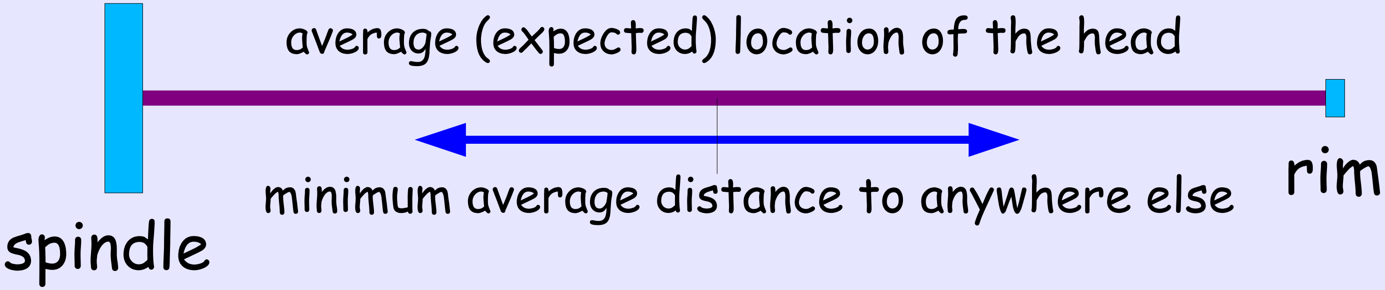
Having completed processing a request, select the next request with the shortest seek distance from your current position.

Similar to SJF, this strategy is simple and optimal (w.r.t. TSD), while being starvation prone.



5 → 2 … → 5 → 2 → 23 → 5 → 2 →

Even if such scenarios do not persist, the cylinders located close to the middle of the surface will tend to receive better service than those located closer to the "edges":



average (expected) location of the head

minimum average distance to anywhere else

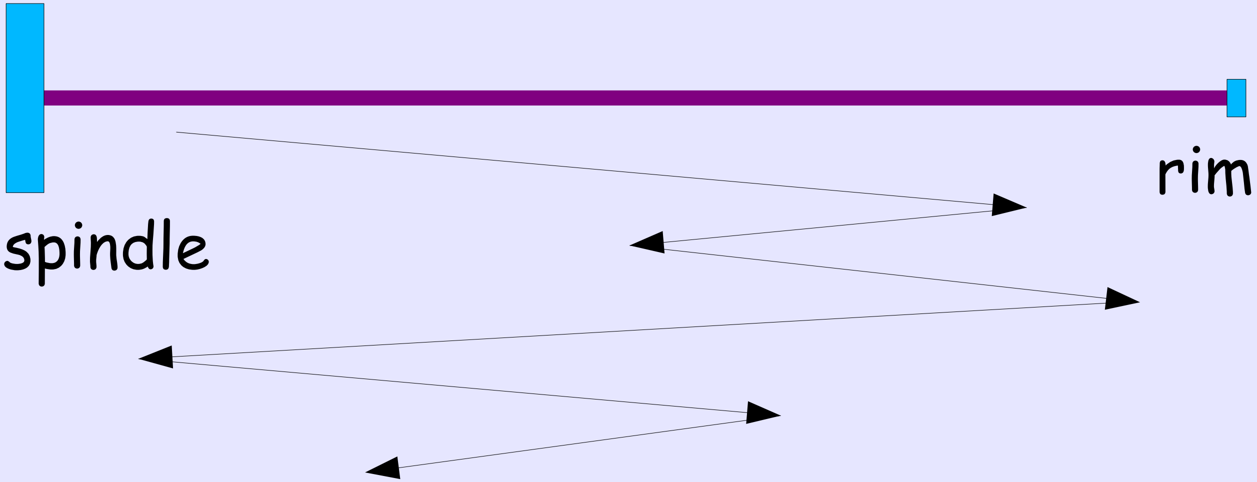
spindle

rim

LOOK (aka Elevator) Strategy

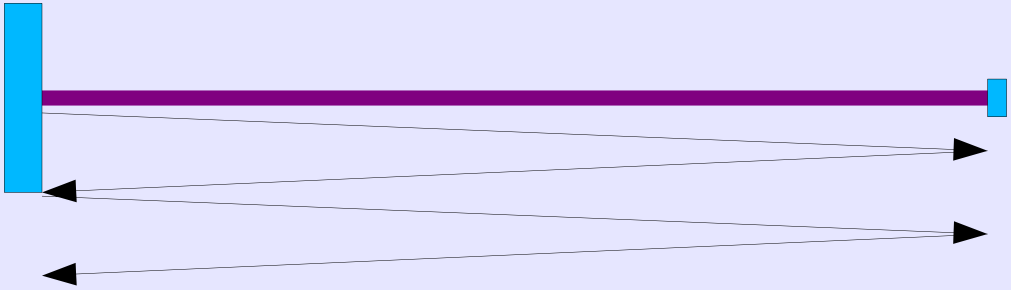
Once you start going in a given direction, do not reverse unless there are no more requests to be serviced along the way.

LOOK is the most popular and widely used strategy. It is not optimal (as SSF), but decent and reasonably fair. There are still some rudimentary fairness issues remaining with SCAN, which lead fairness nuts towards more fair strategies.

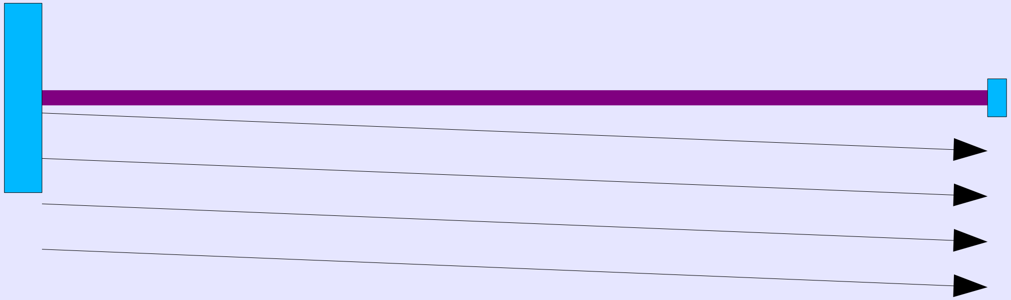


The middle range of cylinders is still favored. If you are located there, your waiting time will likely be shorter than if you are closer to an edge.

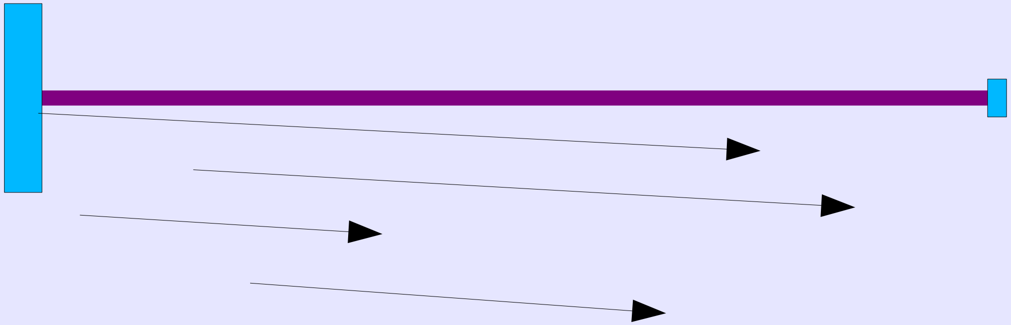
Some (more fair) variations



SCAN: keep going towards the edge even if there are no outstanding requests there (in case they show up).



C-SCAN: service requests while going in one direction only; the return move is idle.



C-LOOK: service requests while going in one direction only; however, you don't have to reach the rim, if there are no more requests towards it.

Other issues

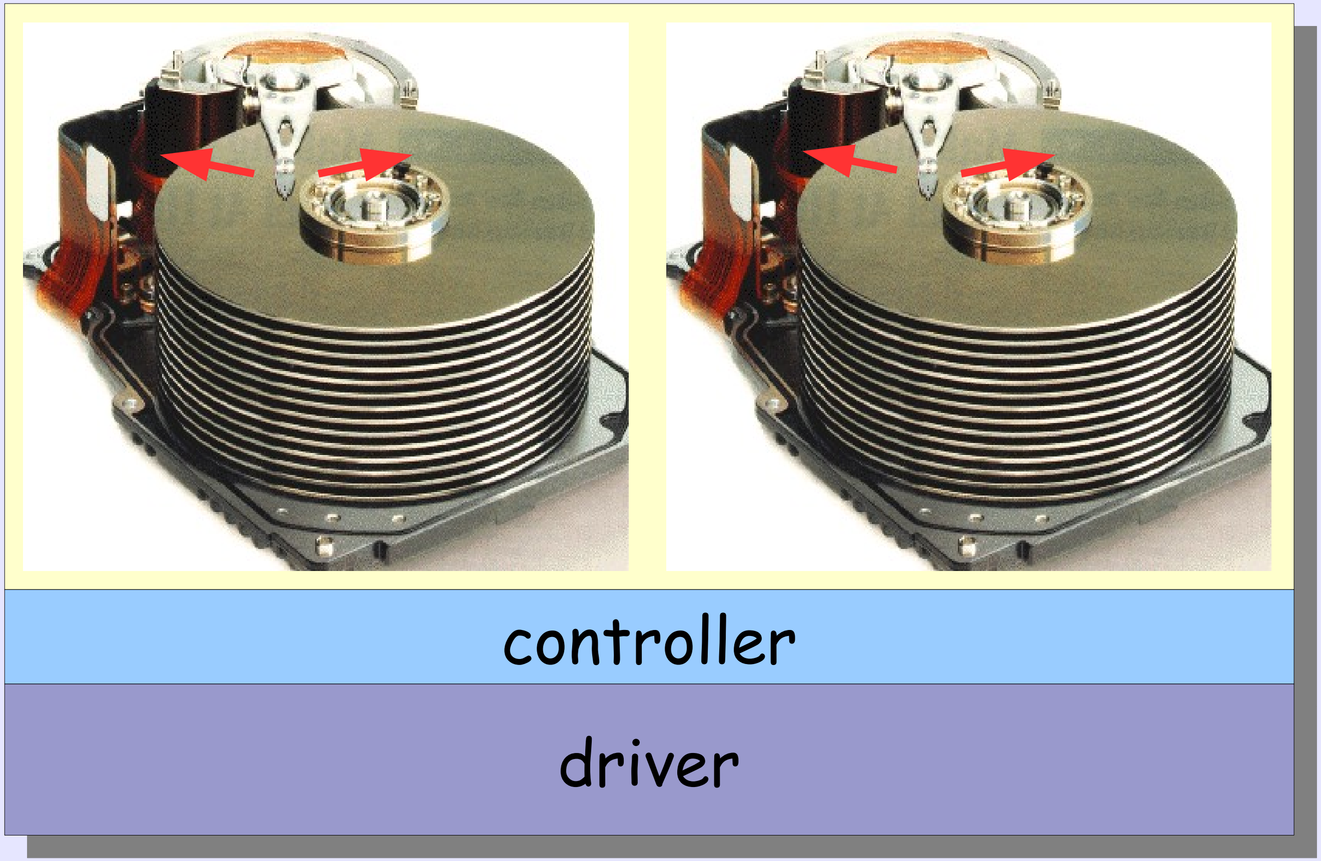
Optimizing other components of the disk access cost:

- The file system will try to allocate continuous ranges of blocks to files.
- The hopscotch approach to numbering consecutive sectors/blocks.
- Reading ahead.
- Merging multiple requests.

Multiple disks in high performance systems:

- Separation of positioning from transfer.
- Disk arrays: for performance and reliability.

Multiple disks: independent positioning



controller

driver

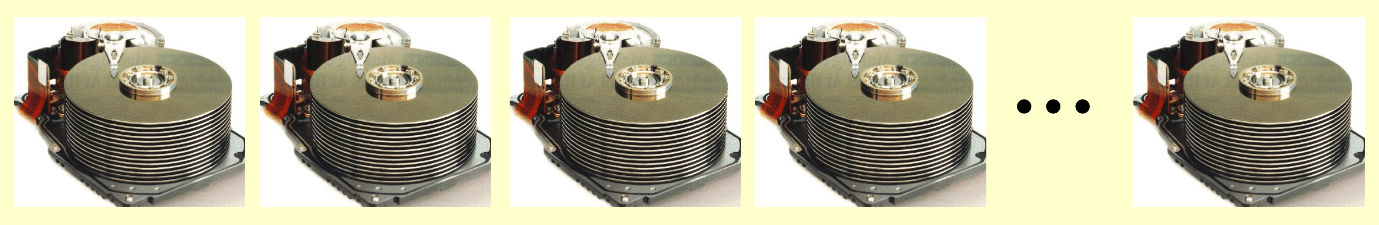
Transaction = seek + wait + transfer. It can be split into two independent parts:

Seek (i.e., position the heads at cylinder N).

Carry out the actual transfer.

The controller may be capable of sustaining one actual data transfer at a time (this occupies a high speed channel, which may be an expensive shared resource), while being able to position heads on multiple disks in parallel.

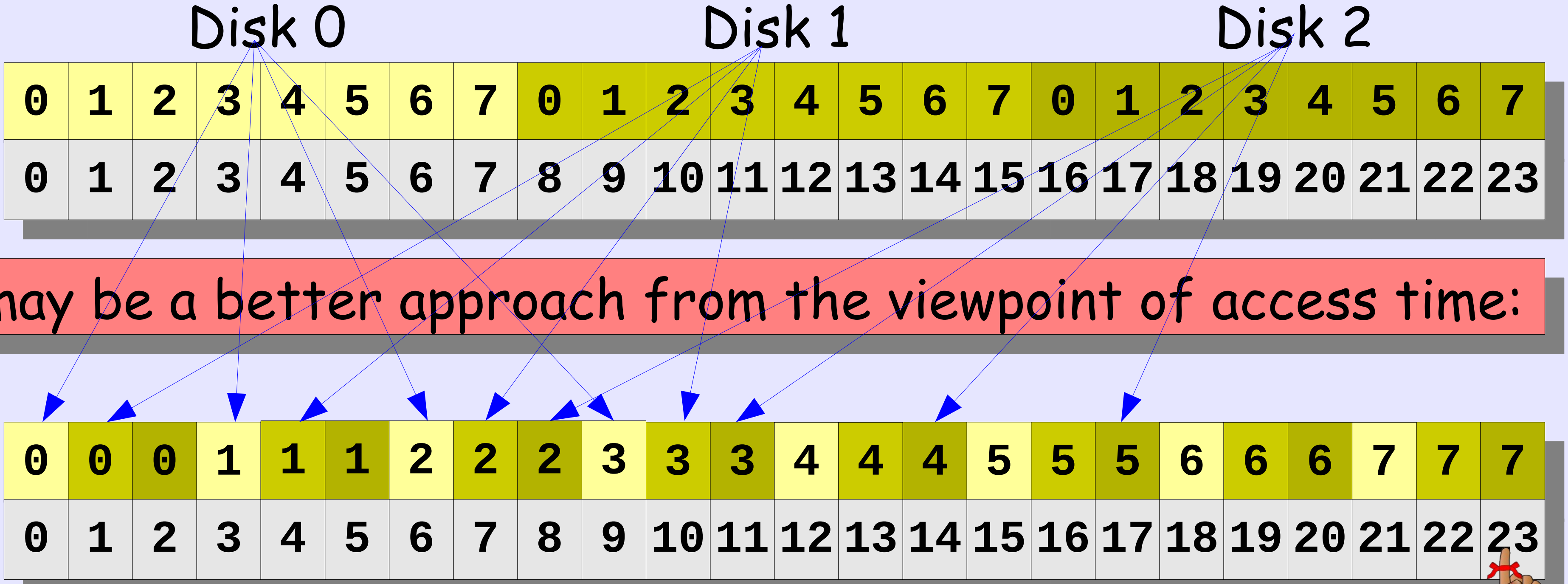
Multiple disks, perhaps lots of them



How to best take advantage of them?

What are the options for a single file system spanning them all?

JBOD: Just a Bunch Of Disks. View them all as a single large virtual disk (a virtual partition?).



0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23

This may be a better approach from the viewpoint of access time:

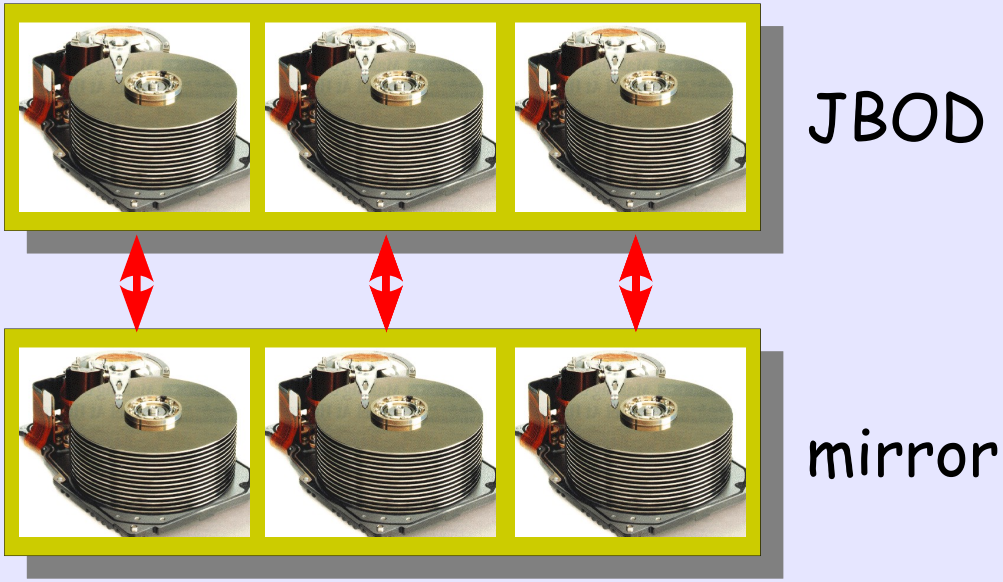
0	0	0	1	1	1	2	2	2	3	3	3	4	4	4	5	5	5	6	6	6	7	7	7
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23

Redundant Arrays of Independent Disks

With JBOD, if a single disk fails, the whole file system becomes essentially useless. It is as if a hole suddenly popped up in your partition.

RAID level 1

Create an exact copy (a mirror) on two or more disks. The multiple disks are for redundancy and reliability rather than larger size. If you have a lot of disks, you can group them, e.g.,



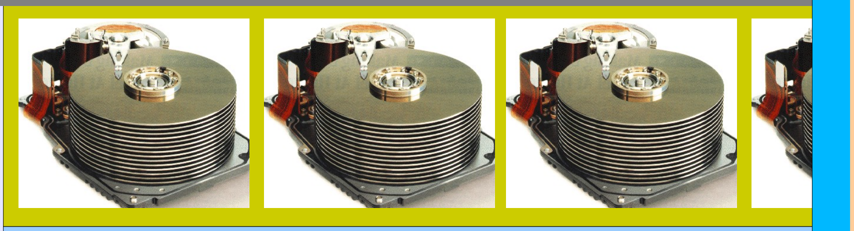
Note that either set can be used for reading (alternately) to reduce average access time.

Writing goes to the basic set, and then the mirror is updated.

Other RAID levels

RAID level 2

Data is striped at the bit leve
disks. Hamming codes can be
recovery from a single or even c



con

```
0111 1001
1101 1010
...
```

0110100 100110

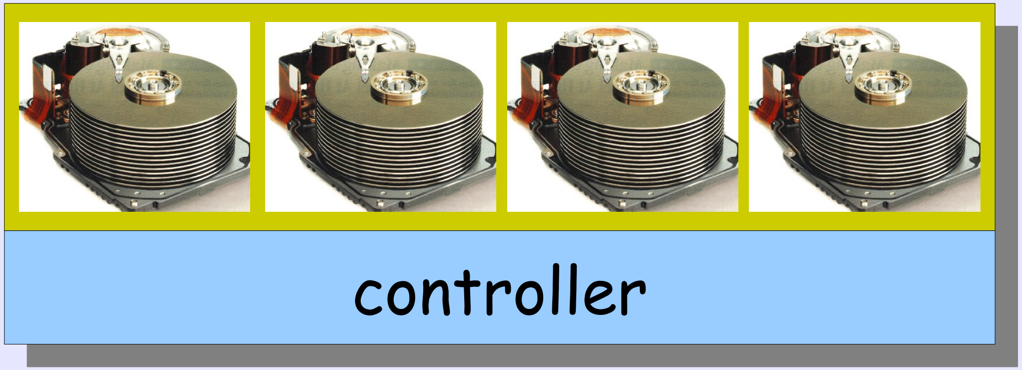
Each byte in a bl
two nibbles, and e
of the 4/7 Hamm
to reconstruct th

0000	=>	0 0 0 0 0 0 0
0001	=>	0 0 0 0 1 1 1
0010	=>	0 0 1 1 0 0 1
0011	=>	0 0 1 1 1 1 0
0100	=>	0 1 0 1 0 1 0
0101	=>	0 1 0 1 1 0 1
0110	=>	0 1 1 0 0 1 1
0111	=>	0 1 1 0 1 0 0
1000	=>	1 0 0 1 0 1 1
1001	=>	1 0 0 1 1 0 0
1010	=>	1 0 1 0 0 1 0
1011	=>	1 0 1 0 1 0 1
1100	=>	1 1 0 0 0 0 1
1101	=>	1 1 0 0 1 1 0
1110	=>	1 1 1 1 0 0 0
1111	=>	1 1 1 1 1 1 1

Here is the most popular one

RAID level 5

Block-level striping with parity data distributed over all disks.



0	1	~~2~~	P0
3	4	~~P3~~	5
6	P6	~~7~~	8
P9	9	~~10~~	11

Writing a block triggers a recalculation of the parity block in the corresponding row.

Suppose one disk fails.

The data blocks in the missing column can be recalculated from the remaining blocks + the parity block.

P = XOR (… … …)

7 = XOR (6 P6 8)

Reading ahead (for file operations)

When a file is opened, the system tentatively sets the read ahead option for the file and assumes some window size specifying how much extra stuff should be read ahead after each actual read.

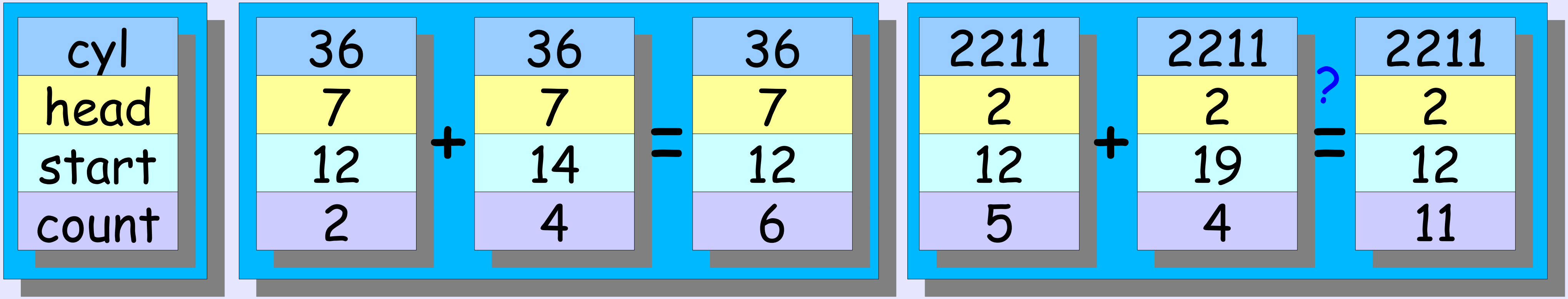
If the subsequent read operation falls within the window, the system assumes that the read ahead option makes sense and it should remain on.

Otherwise, the system assumes that the file is being read at random, and the read-ahead option is switched off.

It's a pity we cannot have write-ahead, isn't it? ☺

Plugging

Whenever a new I/O request is added to the disk queue, the driver tries to merge it with another request already queued there.

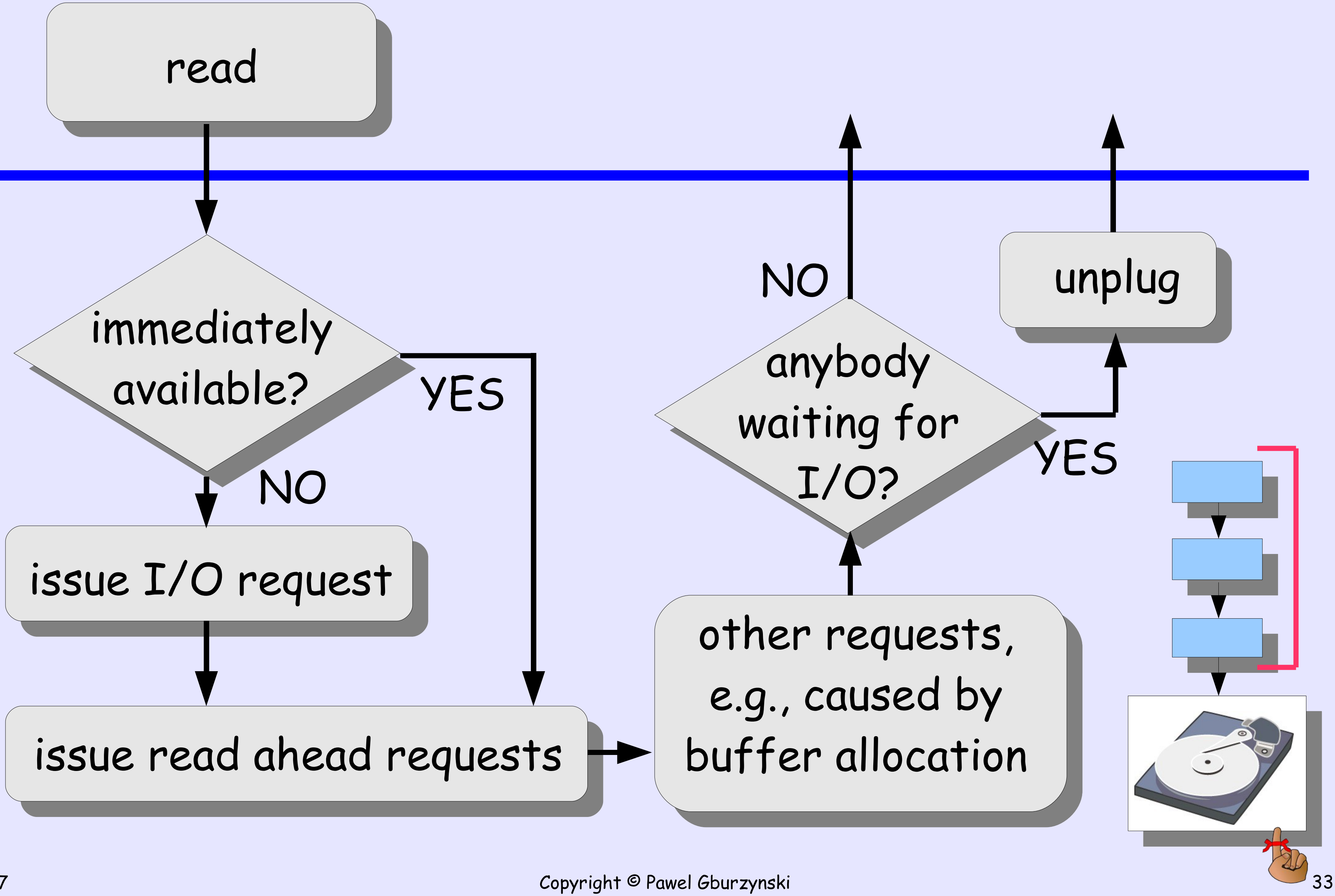


cyl	36		36		36	2211		2211		2211
head	7		7		7	2		2	?	2
start	12	+	14	=	12	12	+	19	=	12
count	2		4		6	5		4		11

What you are able to do depends on the capabilities of the controller.

Sometimes it makes sense to hold a request at the front of the queue waiting for merging opportunities.

How it works



read

immediately available?

YES

NO

issue I/O request

issue read ahead requests

other requests, e.g., caused by buffer allocation

anybody waiting for I/O?

NO

YES

unplug

Security and protection

Threats:

- Unauthorized access to information
- Unauthorized alteration or destruction of information
- Unauthorized acquisition of service
- Denial of service to legitimate users (and other pranks)

Penetration techniques:

- Bribery
- Tricking legitimate users (betting on their recklessness)
- Stealing or guessing authentication information (passwords)
- Electronic eavesdropping
- Trap doors (security holes present in systems)
- Trojan horses, worms, viruses

Authentication

There are obvious tradeoffs between the security of an authentication scheme and its cost and convenience.

Electronic badges, smart cards, tokens, keys, etc., may be secure in principle, but they can be forgotten, lost, replicated ...

Biometric techniques may be very secure, but they are expensive and not always reliable.

Passwords are the least cumbersome and expensive. No wonder they are so popular.

They should be difficult to guess and easy to remember. This is a contradiction, especially if you have to (easily) remember a whole lot of independent (not mutually derivable) secure passwords.

Protecting passwords

Passwords must be known to the system in some form, so that they can be verified when entered for authentication. Early systems used to store them in plain form in (protected) files.

These days passwords are stored in encrypted form, e.g.,

/etc/passwd

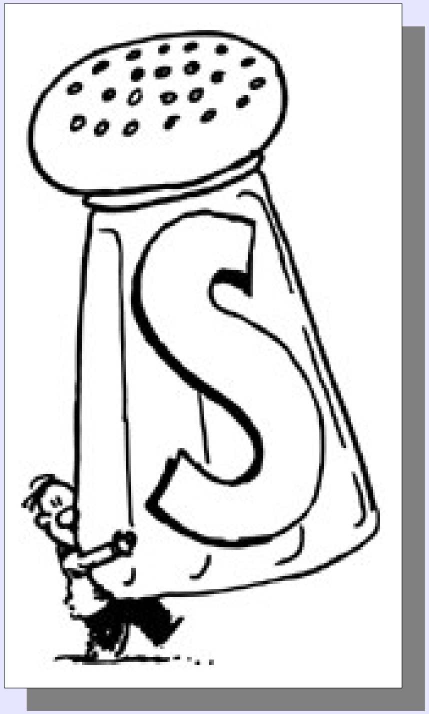
```
root:x:0:0:root:/root:/bin/bash
pawel:x:16089:16089::/home/pawel:/bin/bash
pg:x:0:0:Root alias for Pawel:/root:/bin/bash
sfmowner:x:16110:16110::/home/sfmowner:/bin/bash
cvs:x:16120:16120::/home/cvs:/bin/bash
```

/etc/shadow

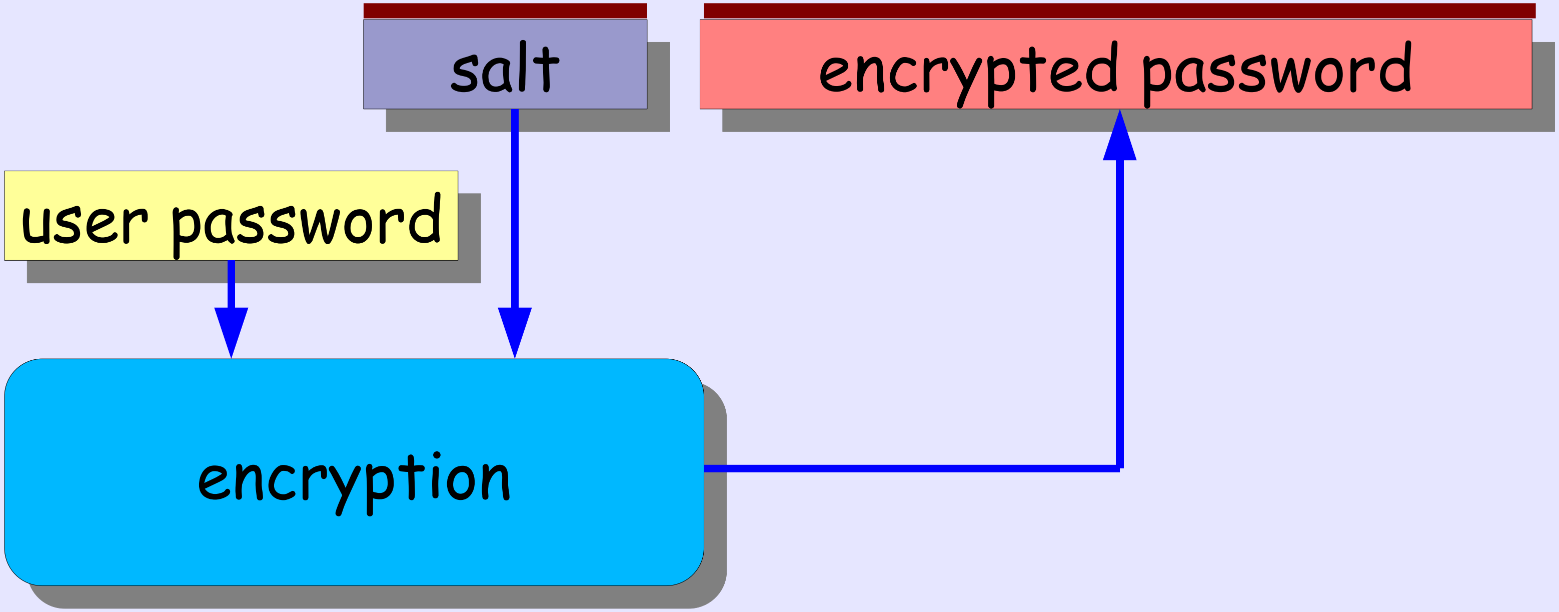
```
root:$1$G6ax6/Fm$jTJzaanahzNZSnuHt8F6c.:12696:0:99999:7:::
pg:$1$n3skVUoL$hbETVgTOFk999Tu/PGSeB/:12696:0:99999:7:::
pawel:$1$SP7HB6Ul$nJlOT01/HIkrjCSo8rvPZ/:12697:0:99999:7:::
sfmowner:$1$lTtApwqj$h7iaryTYdn/YZmLRXYffA1:12712:0:99999:7:::
cvs:$1$fNucVi1O$Gsx0GMOUFdLYab6Qd1dUr/:12839:0:99999:7:::
```

The salt

Encrypted passwords are "salted" to prevent systematic attacks based on pre-encrypted databases of passwords.



```
pawel:$1$SP7HB6Ul$nJlOT01/HIkrjCSo8rvPZ/
```



salt

encrypted password

user password

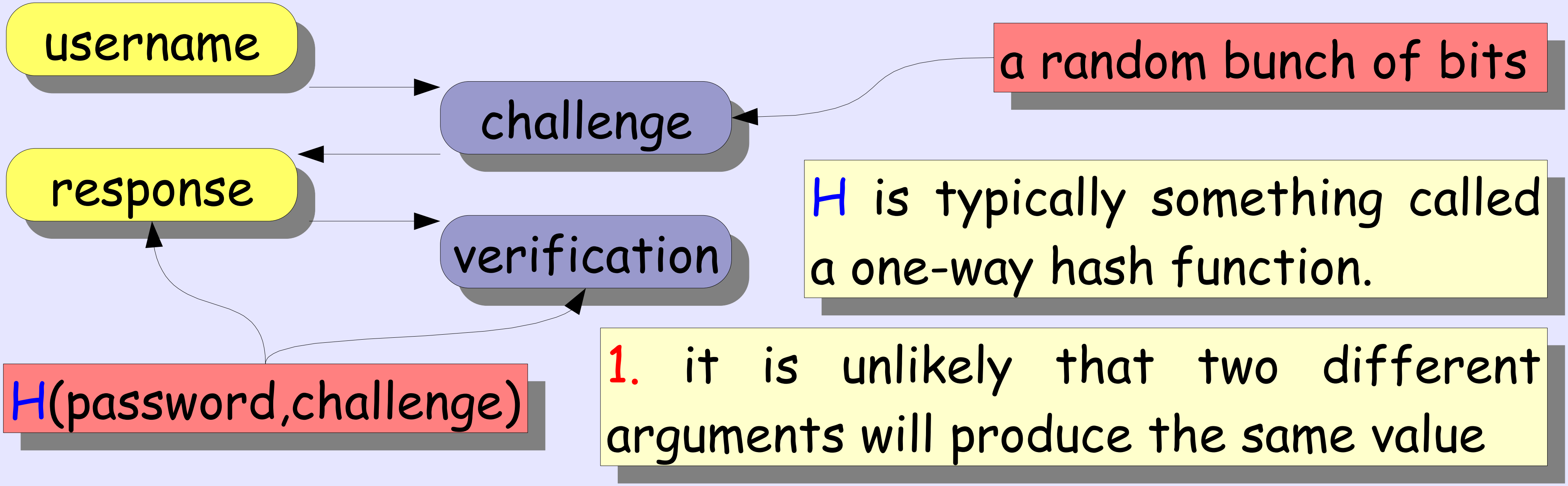
encryption

Salt is generated as a random string when the password is created. Its role is to make sure that the same plaintext password maps into a whole range of possible ciphertexts.

Password exposure

A password must be typed at some point. This makes it susceptible to exposure.

The situation is much worse when the computer is separated from the user by a network.



username

challenge

a random bunch of bits

response

verification

H(password,challenge)

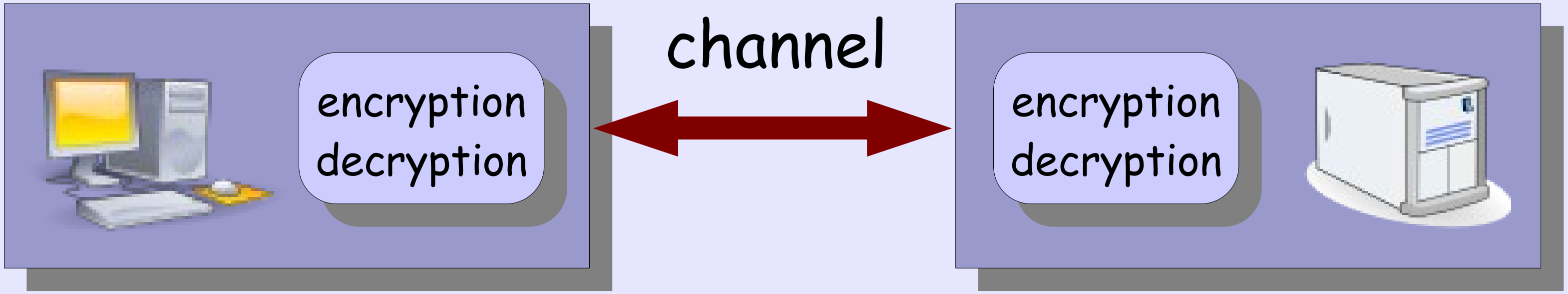
H is typically something called a one-way hash function.

1. it is unlikely that two different arguments will produce the same value

2. it is virtually impossible to reverse H, i.e., determine the argument from the value

Copyright © Pawel Gburzynski

Another option is ssh



Traditional encryption/decryption requires a shared key. This may pose a problem in some circumstances, e.g., you may want to access your account from a foreign (untrusted) machine.

Unfortunately, some (rudimentary) familiarity with cryptography is needed to understand the issues involved in secure operation of computers, not only those interconnected via networks. And which ones aren't these days?!

So embrace yourselves for a crash course...

Traditional cryptography

encryption	decryption
$C = E_K(P)$	$P = D_K(C)$

K is the key, P is the plaintext, C is the ciphertext. E and D are often identical (or almost identical).

Attack types (a good cryptosystem should be resistant to them):

Ciphertext-only. Samples of C are available.

Known plaintext. Samples of P and corresponding C are available.

Chosen plaintext-ciphertext. The attacker can obtain C for specific P.

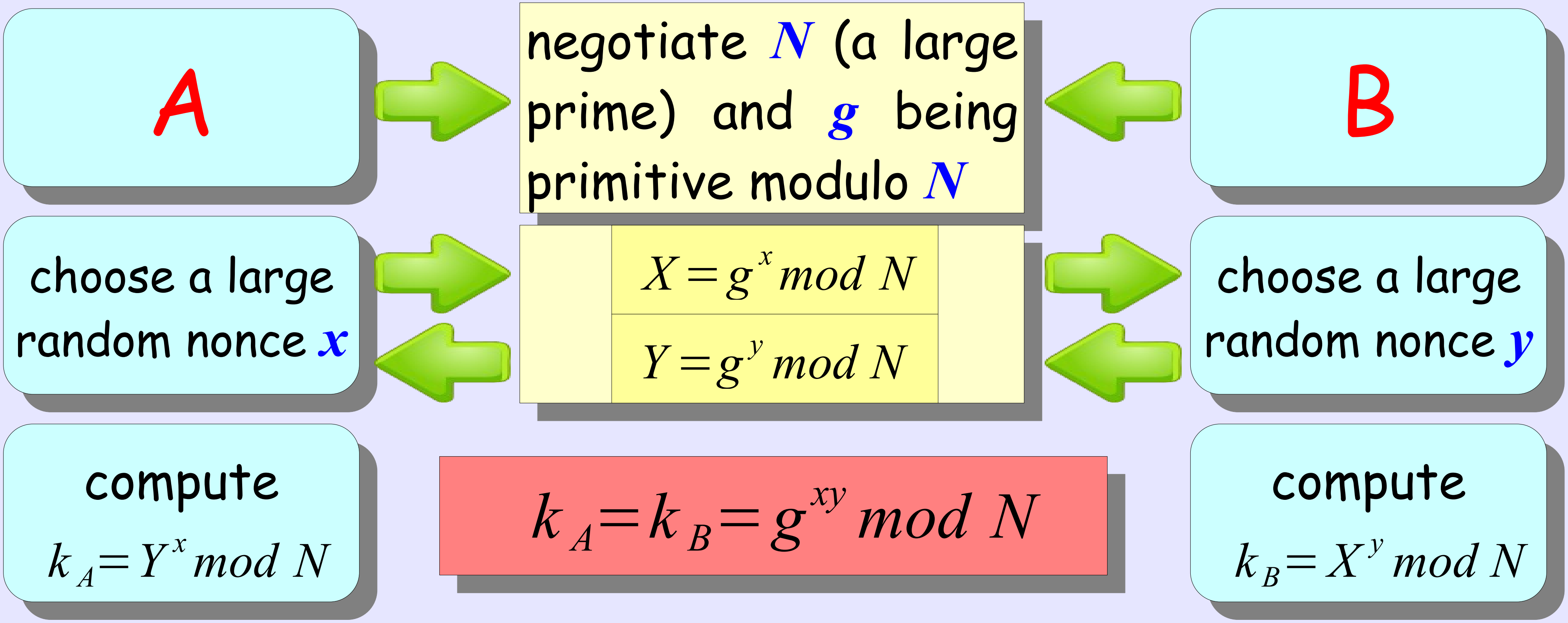
Adaptive chosen plaintext-ciphertext. The attacker has easy access to the encryption device and can choose subsequent P based on the outcome of previous encryptions.

Related key. The attacker can additionally study how the modification of K affects the relationship between P and C.

Key exchange

With traditional cryptography, two parties who want to communicate must agree on a common ***K***. Needless to say, this is tricky if they have never met before.

Diffie-Hellman



A

negotiate ***N*** (a large prime) and ***g*** being primitive modulo ***N***

B

choose a large random nonce ***x***

$X = g^x \mod N$

$Y = g^y \mod N$

choose a large random nonce ***y***

compute

$k_A = Y^x \mod N$

$k_A = k_B = g^{xy} \mod N$

compute

$k_B = X^y \mod N$

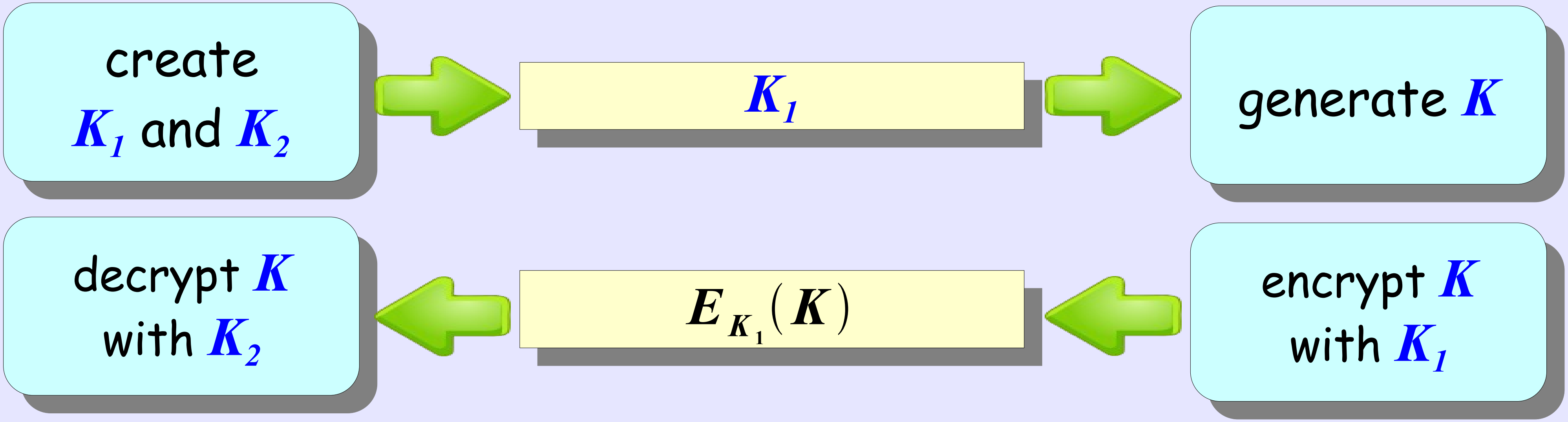
Public-key cryptography

encryption	decryption
$C = E_{K_1}(P)$	$P = D_{K_2}(C)$

K_1 and K_2 are different. The knowledge of one key (+ some P and C) will not let you derive the other key.

Public-key cryptosystems (RSA is the most popular one) are too costly for on-line encryption of large amounts of information. But they can be used for lots of other very useful things.

Key exchange



create K_1 and K_2

K_1

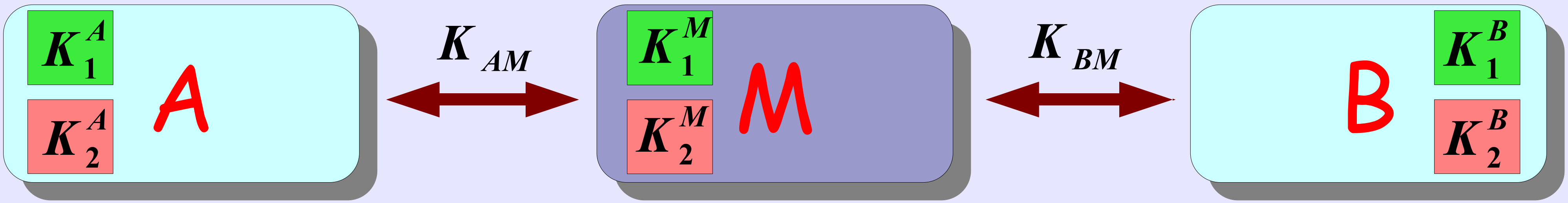
generate K

encrypt K with K_1

$E_{K_1}(K)$

decrypt K with K_2

MIM (the notorious Man In the Middle)



public

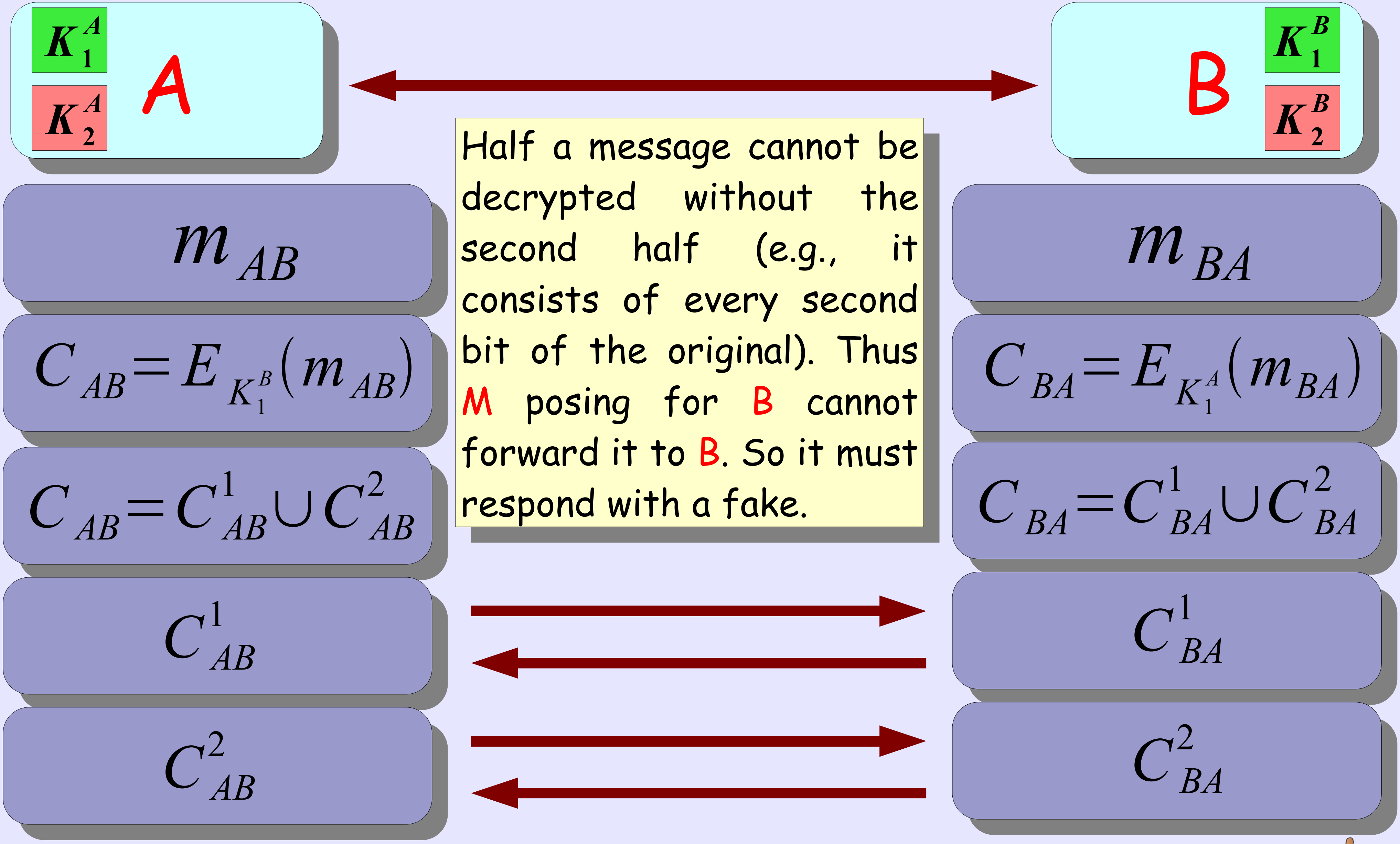
private

A and B set up two “secure” sessions with M thinking that they talk to each other. M can intercept all data. It can also alter data or inject its own data into the session.

Note that the problem can be avoided if A (or B) knows with absolute certainty that the public key K_1^B (or K_1^A) it is holding in fact belongs to B (or A).

But this gets us back to square 1, i.e., how to avoid a trip with the suitcase handcuffed to your wrist?

This may work sometimes



K_1^A K_2^A A

K_1^B K_2^B B

m_{AB}

$C_{AB} = E_{K_1^B}(m_{AB})$

$C_{AB} = C^1_{AB} \cup C^2_{AB}$

C^1_{AB}

C^2_{AB}

m_{BA}

$C_{BA} = E_{K_1^A}(m_{BA})$

$C_{BA} = C^1_{BA} \cup C^2_{BA}$

C^1_{BA}

C^2_{BA}

Half a message cannot be decrypted without the second half (e.g., it consists of every second bit of the original). Thus M posing for B cannot forward it to B. So it must respond with a fake.

Digital signatures

K_1 - private key, ***K_2*** - public key, ***D*** – document to sign. You send to your friend (possibly along with ***D***): $E_{K_1}(D)$

You publish ***K_2*** . Anybody can see that you have signed ***D***. Nobody can forge your signature unless they steal your private key.

Signatures can be shortened by using message digests. A message digest is a one-way hash of the document ***H(D)***, which is typically much shorter (and usually fixed size).

Then, your signature of ***D*** becomes: $E_{K_1}(H(D))$

Establishing trust

Why should you trust my public key? Or anybody's public key for that matter, including your bank?

Easy: I can have my public key signed by somebody whom you trust.

But how can you trust that somebody's public key?

Perhaps somebody else (whom you trust) has signed it?

Will it ever end?

Nope. You must obtain some trust the standard way, i.e., somebody must bring it to you in a suitcase handcuffed to their wrist.

Such suitcases are called certificates.

A sample certificate

```
-----BEGIN CERTIFICATE-----
MIIDiDCCAvGgAwIBAgIDIWeSMA0GCSqGSIb3DQEBBAUAMIHOMQswCQYDVQQGEwJa
QTEVMBMGA1UECBMMV2VzdGVybiBDYXBlMRIwEAYDVQQHEwlDYXBlIFRvd24xHTAb
BgNVBAoTFFRoYXd0ZSBDb25zdWx0aW5nIGNjMSgwJgYDVQQLEx9DZXJ0aWZpY2F0
aW9uIFNlcnZpY2VzIERpdmlzaW9uMSEwHwYDVQQDExhUaGF3dGUgUHJlbWl1bSBT
ZXJ2ZXIgQ0ExKDAmBgkqhkiG9w0BCQEWGXByZW1pdW0tc2VydmVyQHRoYXd0ZS5j
b20wHhcNMDUwNjIzMTcxOTQwWhcNMDYwNjI1MDU0ODM3WjCBmTELMAkGA1UEBhMC
Q0ExEDAOBgNVBAgTB0FsYmVydGExETAPBgNVBAcTCEVkbW9udG9uMR4wHAYDVQQK
ExVVbml2ZXJzaXR5IG9mIEFsYmVydGExKDAmBgNVBAsTH0RlcGFydG1lbnQgb2Yg
Q29tcHV0aW5nIFNjaWVuY2UxGzAZBgNVBAMTEnNmbS5jcy51YWxiZXJ0YS5jYTCB
nzANBgkqhkiG9w0BAQEFAAOBjQAwgYkCgYEAvzBWCVrekWLC1JvKw3qndSGc9ErI
0o5B6RWljOCw4PZUm4W9oXR2nrZxNAWHObBvyU91X+o7Z75v1pG/E6Ooqc49tXQB
t2lEbgARwP5Z1jSirRiUKdFSFSDskov5Bhy2UYuUQ/nb5dYzSsK3QotChgO6h0hL
3rKHwLosAIVma0sCAwEAAaOBpjCBozAdBgNVHSUEFjAUBggrBgEFBQcDAQYIKwYB
BQUHAwIwQAYDVR0fBDkwNzA1oDOgMYYvaHR0cDovL2NybC50aGF3dGUuY29tL1Ro
YXd0ZVByZW1pdW1TZXJ2ZXJDQS5jcmwwMgYIKwYBBQUHAQEEJjAkMCIGCCsGAQUF
BzABhhZodHRwOi8vb2NzcC50aGF3dGUuY29tMAwGA1UdEwEB/wQCMAAwDQYJKoZI
hvcNAQEEBQADgYEAVNzUzgkJmQVWzNgf+EuFpFi+x4uKovVBSreJd/Od+Go28y2R
OHapjZX4udO2GJRPISqCS2/3Vq56pHzPMA3ltVVwk1UzK3OHuHn9NklVZSR4+2xL
WlZIWFtnjyKoYYOUqV/G1COaj2OGyzgSAORYSfnGW/UL7aGL0rUD9d4zdQo=
-----END CERTIFICATE-----
```

Its contents are not encrypted. You can read them, e.g., with this command: openssl x509 -text < sfm.crt

... and you will see:

```
Version: 3 (0x2)
Serial Number: 2189202 (0x216792)
Signature Algorithm: md5WithRSAEncryption
Issuer: C=ZA, ST=Western Cape, L=Cape Town, O=Thawte Consulting ...
Validity
   Not Before: Jun 23 17:19:40 2007 GMT
   Not After : Jun 25 05:48:37 2008 GMT
   Subject: C=CA, ST=Alberta, L=Edmonton,
        O=University of Alberta, OU=Department of Computing Science,
        CN=sfm.cs.ualberta.ca
 ...
Subject Public Key Info:
   Public Key Algorithm: rsaEncryption
   RSA Public Key: (1024 bit)
   Modulus (1024 bit):
        00:bf:30:56:09:5a:de:91:62:c2:d4:9b:ca:c3:7a:
        a7:75:21:9c:f4:4a:c8:d2:8e:41:e9:15:a5:8c:e0:
 ...
Signature Algorithm: md5WithRSAEncryption
        54:dc:d4:ce:09:09:99:05:56:cc:d8:1f:f8:4b:85:a4:58:be:
        c7:8b:8a:a2:f5:41:4a:b7:89:77:f3:9d:f8:6a:36:f3:2d:91:
 ...
```

Why should you trust the certificate?

It was signed by one of recognized certifying authorities. They had to verify that the certificate belonged to me before signing.

But how do you know that the certificate has been actually signed by THEM?

Because your browser (or whatever software interprets the certificate) comes equipped with their public key. That key was put there by whoever set up the software.

Sometimes an authority has no ready key implanted in the customer's software. Then their key must be signed by another authority who has. Or perhaps the key of that signing authority has been signed by another authority ...

Penetration types

Get access to a machine on which you have no account at all:

- break-in over the network through a pre-existing security hole (trapdoor)
- spread a virus that will create a trapdoor for you

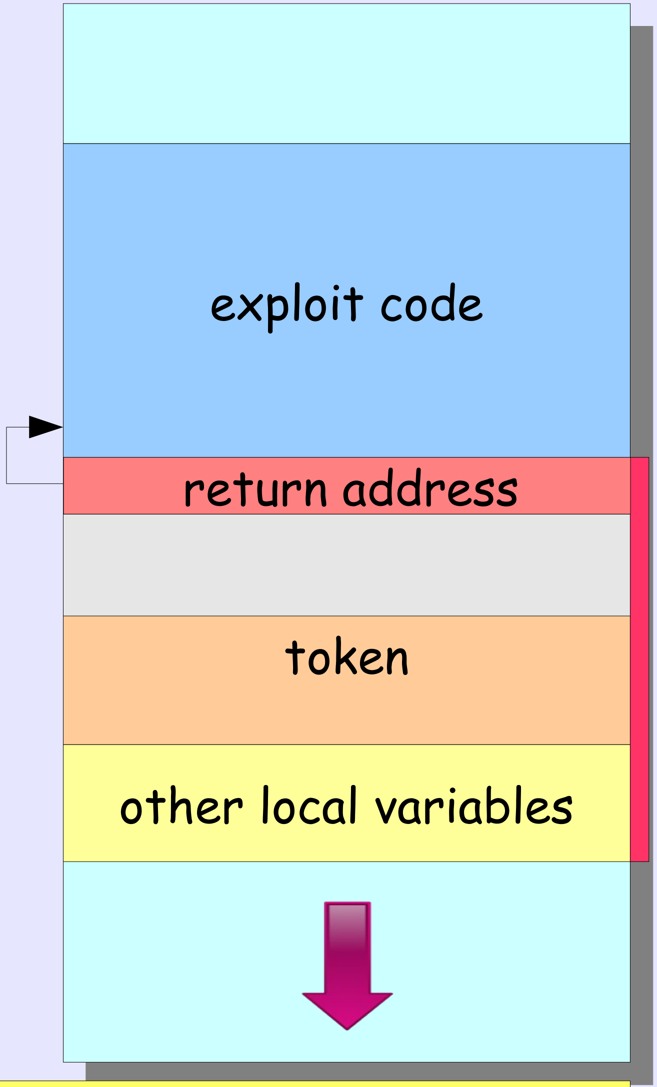
Become root (the superuser) on a machine to which you have access as a non-privileged user:

- your non-privileged access can be legitimate to begin with
- you have established non-privileged access by breaking into a regular user's account (see above)

Note that on Windows the vast majority of all users have administrator privilege (i.e., everybody runs as root). Isn't that stupid?

Buffer overflow

```
bool trim (const char *name) {
  char token [24];
   ...
  strcpy (token, name);
   ...
}
```



The program itself may have a very limited functionality, but the exploit can force it to do anything, e.g.,

- ➔ setup a socket to your machine
- ➔ make stdin/stdout point to the socket
- ➔ execute sh

Note that you can do a lot from a very short exploit, as it can access all the libraries linked to the exploited program.

Web-related attacks

Browser attacking the server: fooling the server into executing a script supplied by the user.

Server attacking the browser: running a script in the browser that:

- compromises confidential data (e.g., harvests e-mail addresses)
- alters the user's files
- crashes or overloads the system

CGI scripts run by web servers are a never-ending source of problems.

```
...
$username = $form (“username”);
print 'finger $username';
...
```

Username: `foo; rm -rf *`

Becoming superuser

Let us use UNIX for example. Becoming superuser on Windows is far from posing a remote challenge. On UNIX, the superuser is called root.

/etc/passwd

```
root:x:0:0:root:/root:/bin/bash
pawel:x:16089:16089::/home/pawel:/bin/bash
pg:x:0:0:Root alias for Pawel:/root:/bin/bash
sfmowner:x:16110:16110::/home/sfmowner:/bin/bash
cvs:x:16120:16120::/home/cvs:/bin/bash
```

User ID (UID)

Group ID (GID)

Root is user ID 0. The actual user name is irrelevant. Root can:

- freely access all files and devices
- freely access all memory (which is available as a special device)
- change system parameters

Superprograms on UNIX

Some programs (not only on UNIX) must be privileged, i.e., they must be able to do something that a regular user program cannot do, even if they are run by a regular user, e.g.,

passwd, chsh must be able to modify the password file

su must be able to start a shell as a different user

ping, traceroute must be able to open raw sockets

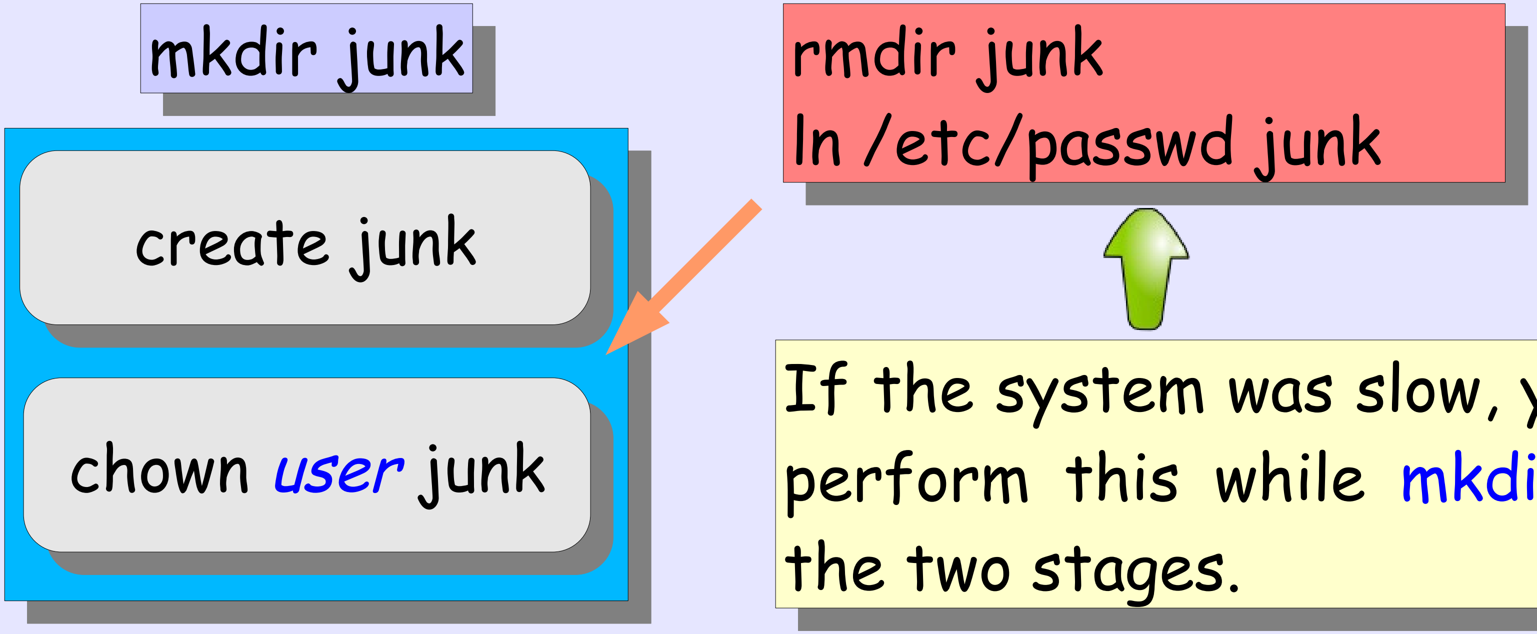
```
ls -l /usr/sbin
...
-rwxr-xr-x  1 root root      92612 Mar  2  2004 aspell
-rwxr-xr-x  1 root root       2044 Mar  2  2004 aspell-import
-rwsr-xr-x  1 root root      38936 May  5  2004 at
-rwxr-xr-x  1 root root       6076 Mar 30  2004 atktopbm
...
```

's' stands for SETUID on execution. The program will be run as if it were called by the owner of its executable.

Tricking SETUID programs

... is a particularly exciting endeavor. Looking for a buffer overflow is an obvious idea (lots of accomplishments in this area have been noted). But there are other ways.

On old UNIX systems, mkdir was a SETUID program.



mkdir junk

create junk

chown *user* junk

rmdir junk
ln /etc/passwd junk

If the system was slow, you could try to perform this while mkdir was between the two stages.

Of course, you had to be prepared to try more than once. Needless to say, you didn't do it by hand.

A few observations

The seriousness of a security hole is determined by what can be gained by exploiting it, not by its “size”.

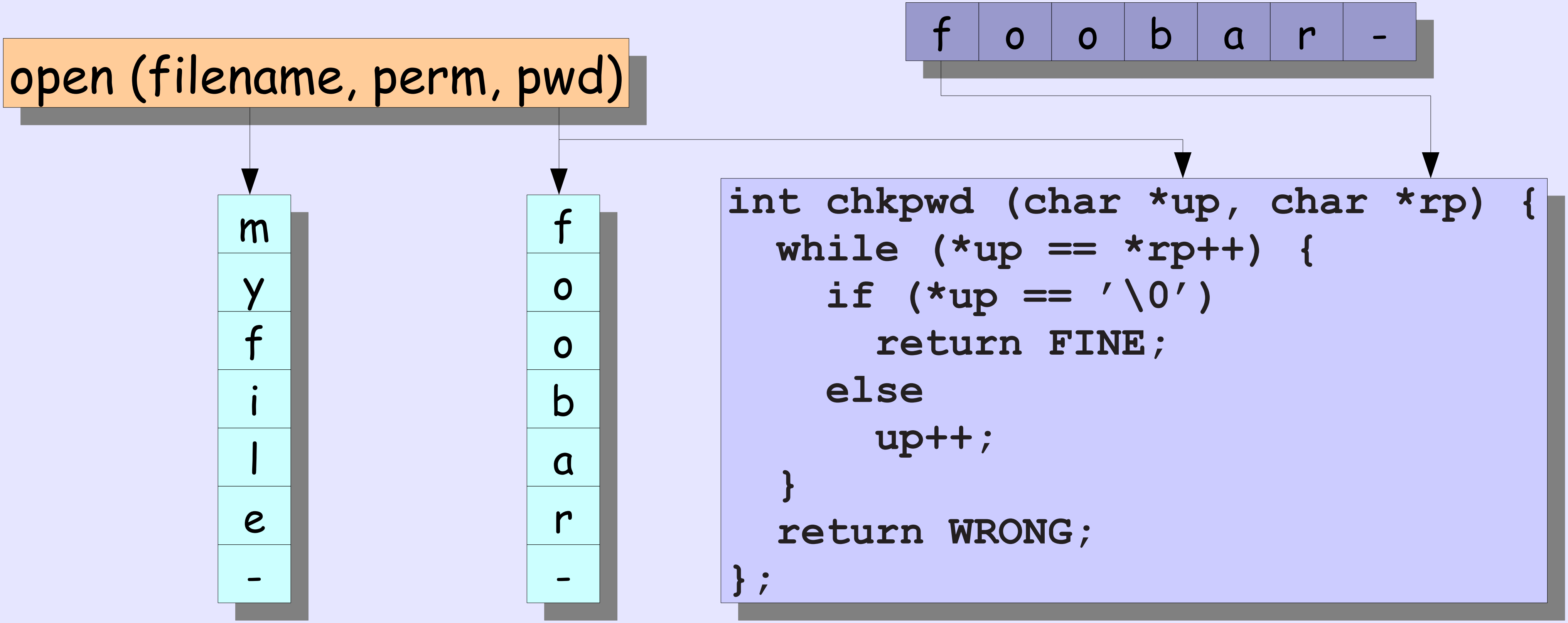
It is considerably more difficult to prove (or at least become convinced about) the impossibility of something than to demonstrate that something exists. Consequently writing a working program is incomparably easier than writing a secure program.

Securing systems consisting of several programs is NOT equivalent to securing individual programs. It requires completely new insights.

Illustration

A certain system (call it TENEX) for a certain mainframe machine (call it DEC 10) used passwords to protect files.

The system used VM. Programs were able to monitor their page faults in order to give a performance feedback to the users.



open (filename, perm, pwd)

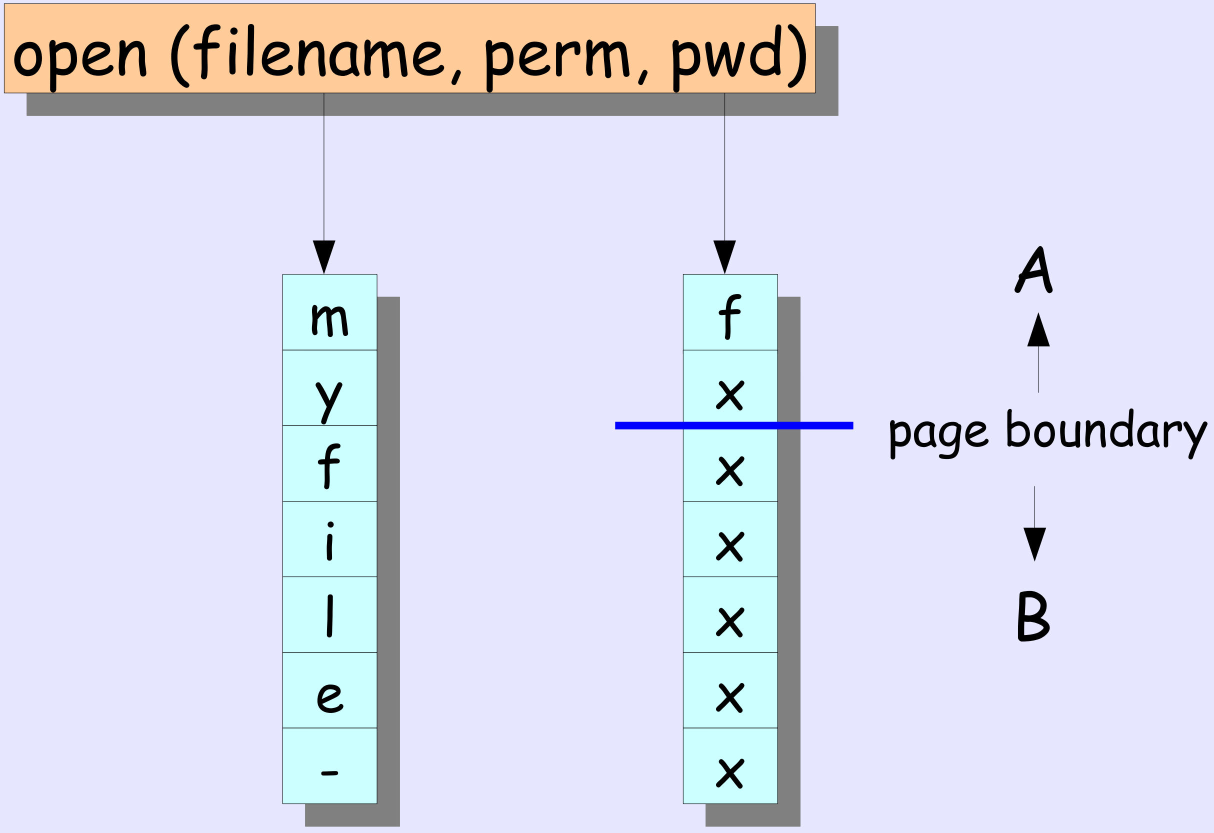
f	o	o	b	a	r	-

m	y	f	i	l	e	-

f	o	o	b	a	r	-

```
int chkpwd (char *up, char *rp) {
  while (*up == *rp++) {
    if (*up == '\0')
      return FINE;
    else
      up++;
  }
  return WRONG;
};
```

The security hole



open (filename, perm, pwd)

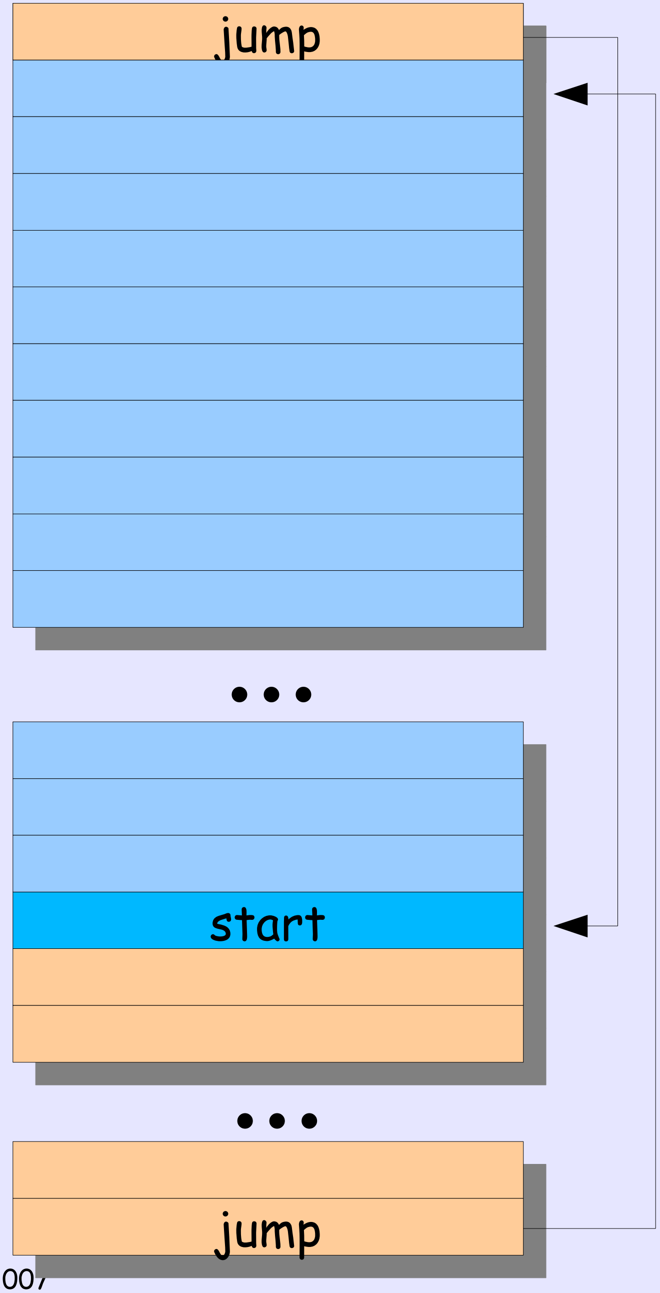
Make sure page A is present while B is not.

If you get a fault for B, you have guessed the first character.

Then shift the string one byte up and try again for the next character.

The complexity of guessing the password has been reduced from N^K to $N \times K$.

Viruses and worms



Viruses are programs that reproduce themselves by attaching to other programs.

They spread when binary (executable) programs are shared by users.

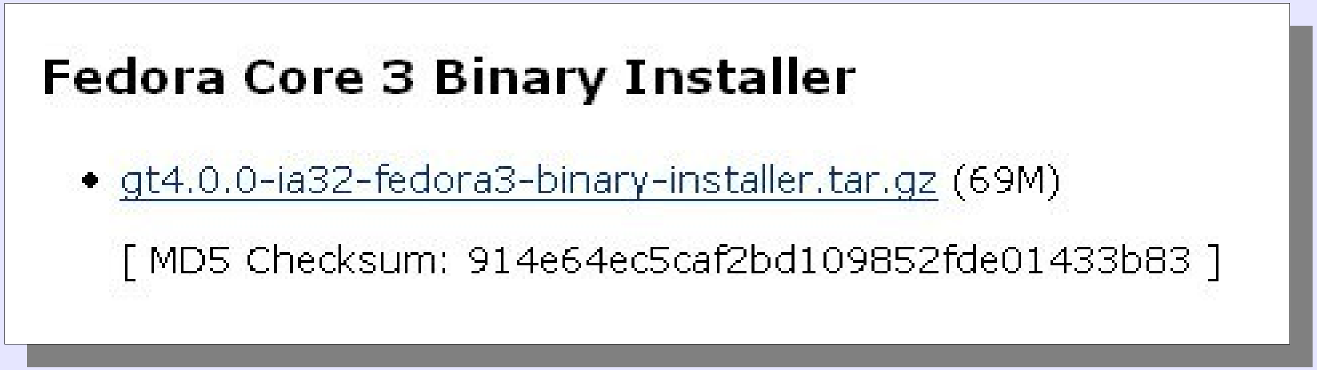
The amount of havoc they can create depends on the permissions of the infected program. Unfortunately, on Windows, many programs have infinite permissions.

Worms are programs that spread by means other than infection. People often confuse them with viruses.

Some issues in virology

Detection

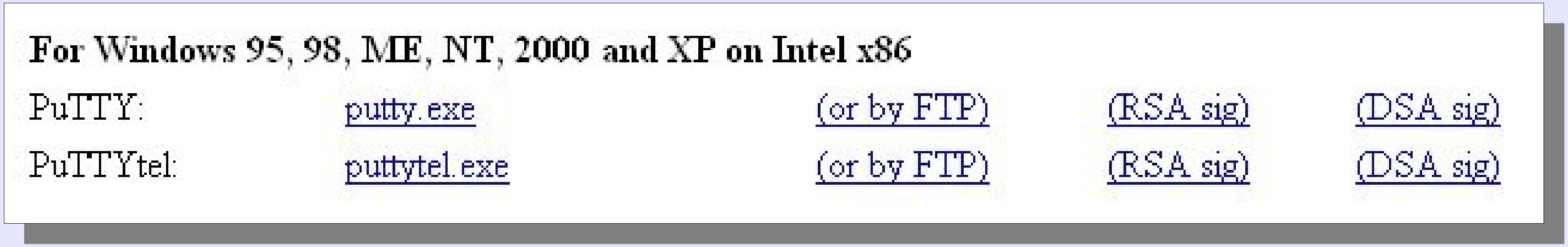
- Signatures for clean executables. These days, when you download code from a respectable site, they provide you with checksums or signatures.
- Virus detectors based on databases of footprints of known viruses.



Fedora Core 3 Binary Installer

- gt4.0.0-ia32-fedora3-binary-installer.tar.gz (69M)

  [MD5 Checksum: 914e64ec5caf2bd109852fde01433b83]



For Windows 95, 98, ME, NT, 2000 and XP on Intel x86

PuTTY:	putty.exe	(or by FTP)	(RSA sig)	(DSA sig)
PuTTYtel:	puttytel.exe	(or by FTP)	(RSA sig)	(DSA sig)

Advanced viruses

Delayed action – to delay detection until after the virus has spread. This will maximize the effect.

Hiding from the infected program (staged infection).

Morphic code – to complicate the footprint.

Cryptographic attacks: data-napping, safe "collection of ransom."

Non-traceability of the attacker.

Accessibility/protection in OS: domains

An operating system deals with objects, e.g., processes, files, devices, memory segments, sockets, which should be protected.

Protection rules specify who is allowed to perform what operations on which objects.

A domain is set of pairs <*object*, *rights*>, where *rights* is a subset of operations applicable to the *object*.

Here is a sample domain:

/home/pawel/myfile	rwx
/home/yannis/hisfile	r
process 11452	kill
semaphore S10	V

When we say that a process executes in some domain, it means that the process is allowed to perform a specific set of operations on a specific set of objects.

Domains on UNIX

In traditional UNIX, the domain of a process is determined by its UID and GID. Given these two numbers, you can tell what the process can do with all relevant objects in the system.

```
ls -l
drwxr-xr-x 3 pawel users  4.0k Oct 25 13:30 379
drwx------ 3 pawel users  4.0k Nov 16 03:39 ARCHIVE
drwxr-xr-x 2 pawel users  4.0k Sep  5  2004 BIN
-rw-r--r-- 1 pawel yannis 5.9k Aug  8 16:44 Config
lrwxrwxrwx 1 pawel users    26 Apr 21  2005 FTP -> /usr/menaik3/ftp/pub/pawel
-rw------- 1 pawel users  3.3k Sep 29  2004 Mailbox
```

This form of expressing accessibility is common to all objects that are represented by inodes, i.e., files, devices, sockets, fifos, symbolic links.

Other objects, e.g., shared memory segments, semaphores, are also tagged with UIDs/GIDs of their owners.

Domain switching

Normally, it should be impossible for a process to change its domain at will. The whole idea of protection would make no sense.

On UNIX (and on all systems) one obvious way for a process to temporarily upgrade its domain is to issue a system call.

SETUID programs run in different domains than the users (programs) invoking them.

UNIX rules for explicit domain switching are a bit tricky in general, although simple in principle.

In principle:

A non-root process cannot change its UID/GID. A root process can change them to whatever it pleases.

More specifically

Each process has three sets of UID/GID attributes:

effective

the one being used for assessing the process's access rights

real

the one reflecting the actual user running the process

saved

which may reflect the process's past UID/GID

For a typical process, the three pairs of attributes are identical. If this is a non-root process, it cannot change them at all.

The rules

Suppose you have a SETUID process. When it starts, its

effective ID (euid, egid)

reflects the ID of the owner of the executable file

real ID (ruid, rgid)

reflects the ID of the user running the program

saved ID (suid, sgid)

is the same as effective ID

The process can:

```
seteuid (ruid);
setegid (rgid);
```

to temporary relax the privilege, and then do

```
seteuid (suid);
setegid (sgid);
```

to regain it

Representing domains in the system

Conceptually, we can envision a matrix:

	File1	File2	Sem1	Sem2	Prt1	Prcs1	Prcs2	D1	D2	D3
D1	RW	RX	PV	PV		KSR	SR		E	
D2	R		P	PV	W	S				E
D3	RWX	X	V			SR	R			

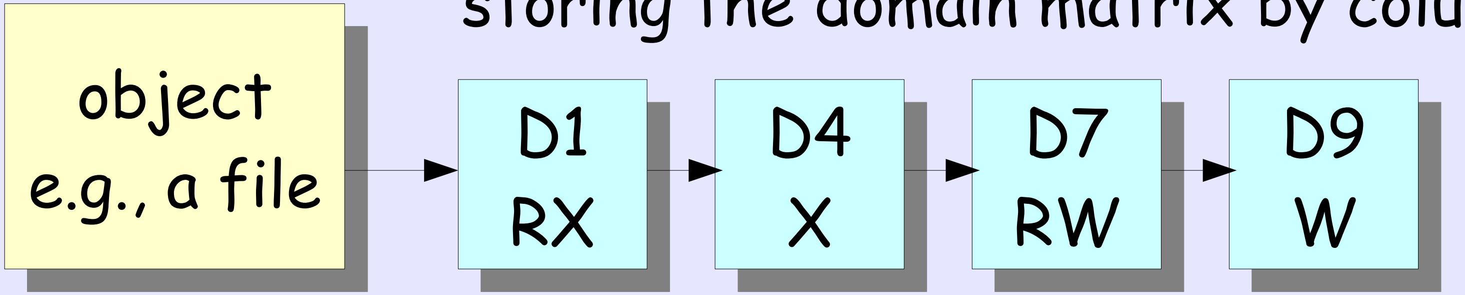
The rules for domain switching can be included in the matrix: domains themselves are objects, and the operation of entering a domain is subject to protection rules.

Nobody sane would actually try to store domain matrices as matrices because they tend to be quite sparse (and large).

The two possible ways of storing them correspond to two paradigms in describing access rights in operating systems.

Way 1: Access Control Lists

storing the domain matrix by columns



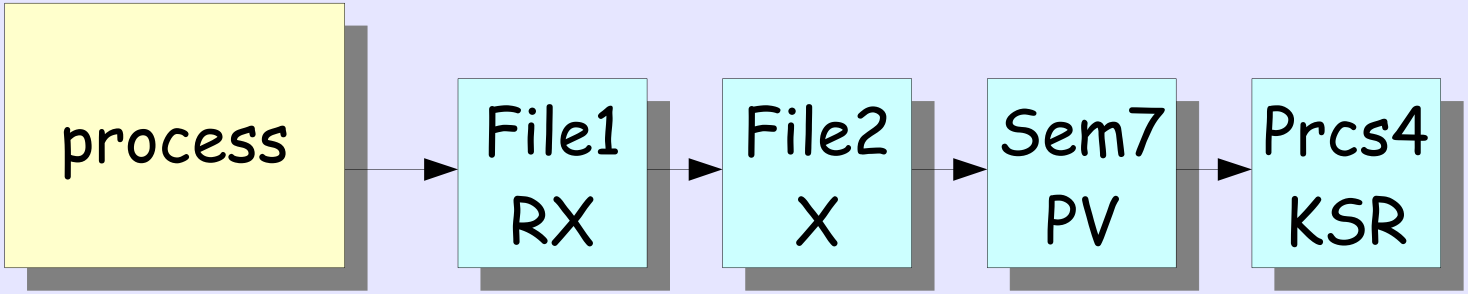
On UNIX, we have a poor man's variant of ACLs compressed to 9 (or so) bits.

For each file, there are just four possible domains: owner, group, world, and the superuser, with the last one being trivial and needing no description.

For each non-trivial domain there are three operations: read, write, execute. The SETUID feature adds no new operation. It assigns a domain to a process created from the executable file.

Way 2: Capabilities (by rows)

With each process, we associate a list of capabilities, i.e., “tickets” allowing the process to access some objects for some purpose.



Capabilities are more natural in some environments, notably in distributed systems, where keeping track of domains is complicated.

Capabilities are always well defined because one capability refers to one specific object.

How to represent capabilities?

➔ Let the kernel store them (so that processes cannot tamper with them) and make them available via verifiable pointers/descriptors.

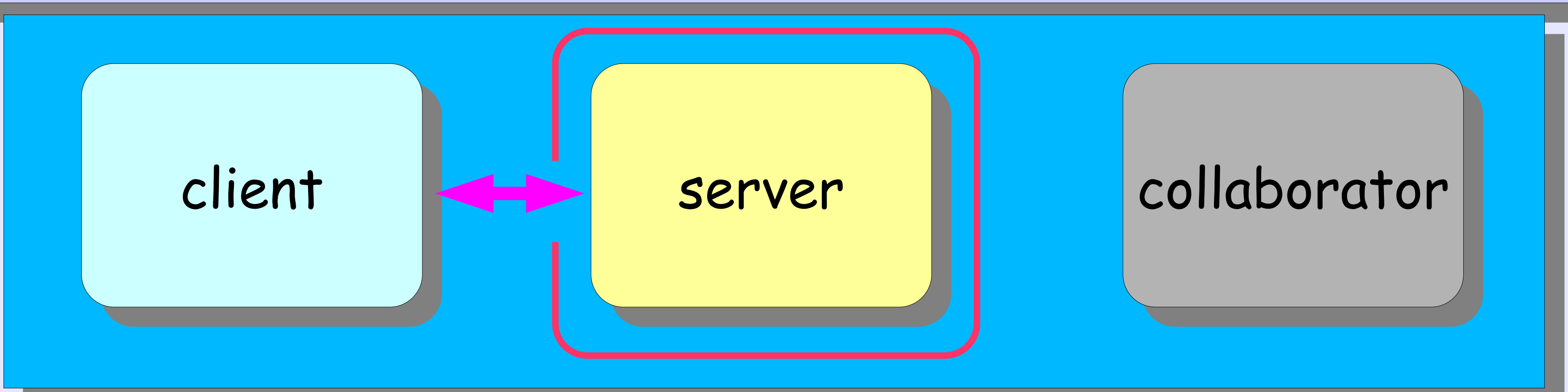
Note that processes would like to be able to pass capabilities, e.g., to their children. For example, UNIX file descriptors can be viewed as capabilities of sorts.

However, this approach is not possible in truly distributed systems, e.g., ones that do not share memory at all.

➔ Let the processes store capabilities by themselves, but make them tamper-resistant via cryptographic means.

Covert channels

Can inter-process communication be treated as a protectable operation?



- The server is not allowed to write to a file that the collaborator may have access to.
- The server is not allowed to write to a socket, mailbox, whatever that the collaborator may access.
- The server is not allowed to touch any object that the collaborator could possibly use as a means of getting any message from the server.

Unfortunately ...

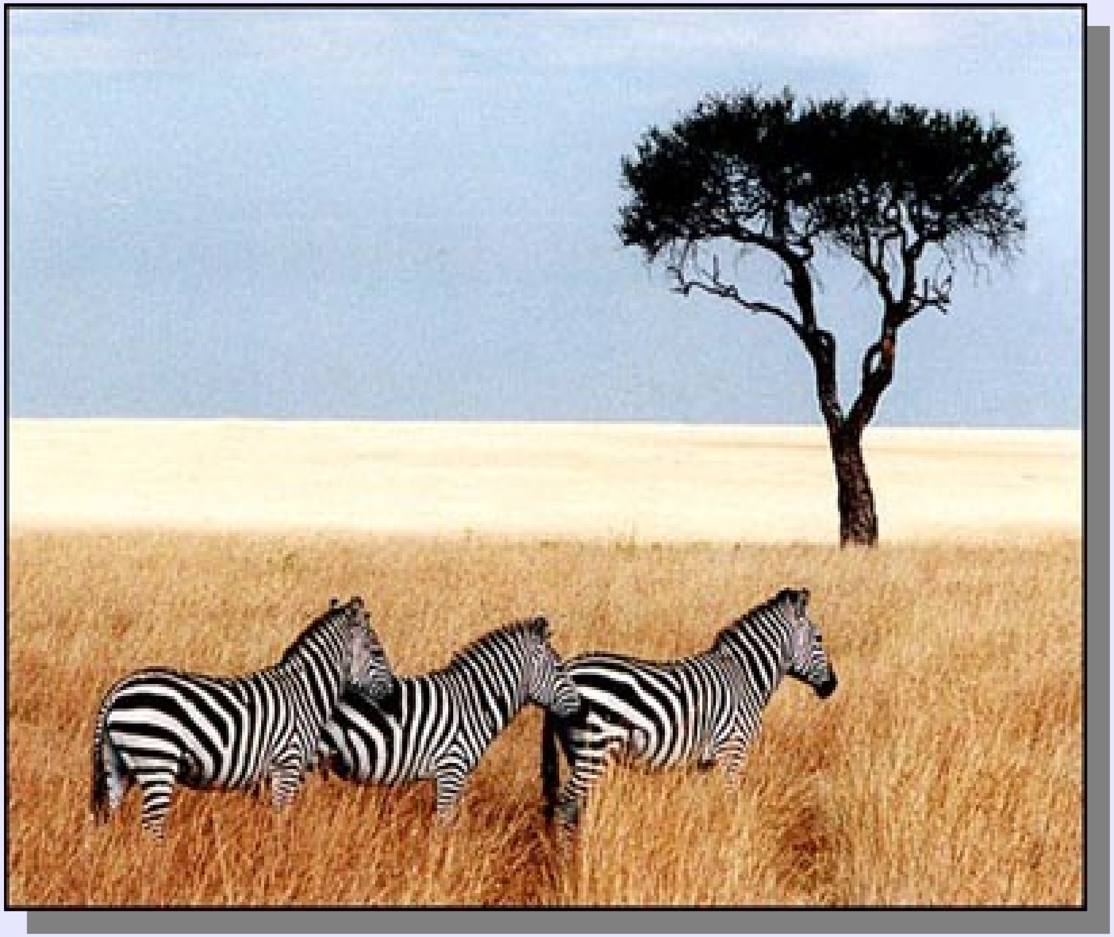
... the problem cannot be solved by formally restricting the domains of the two conspirators. The server always can:

- exhibit a specific pattern of CPU activity that will be perceived by the collaborator as increased/decreased load;
- exhibit a specific pattern of I/O activity (this will affect the I/O service perceived by the collaborator);
- allocate completely innocent resources (e.g., recall the abominable tape drives) according to some patterns;

In fact, any way of influencing the system's performance that can be perceived at all by the collaborator will work.

The channel can be slow and noisy, but as long as it is not random, it can be used to pass reliable messages.

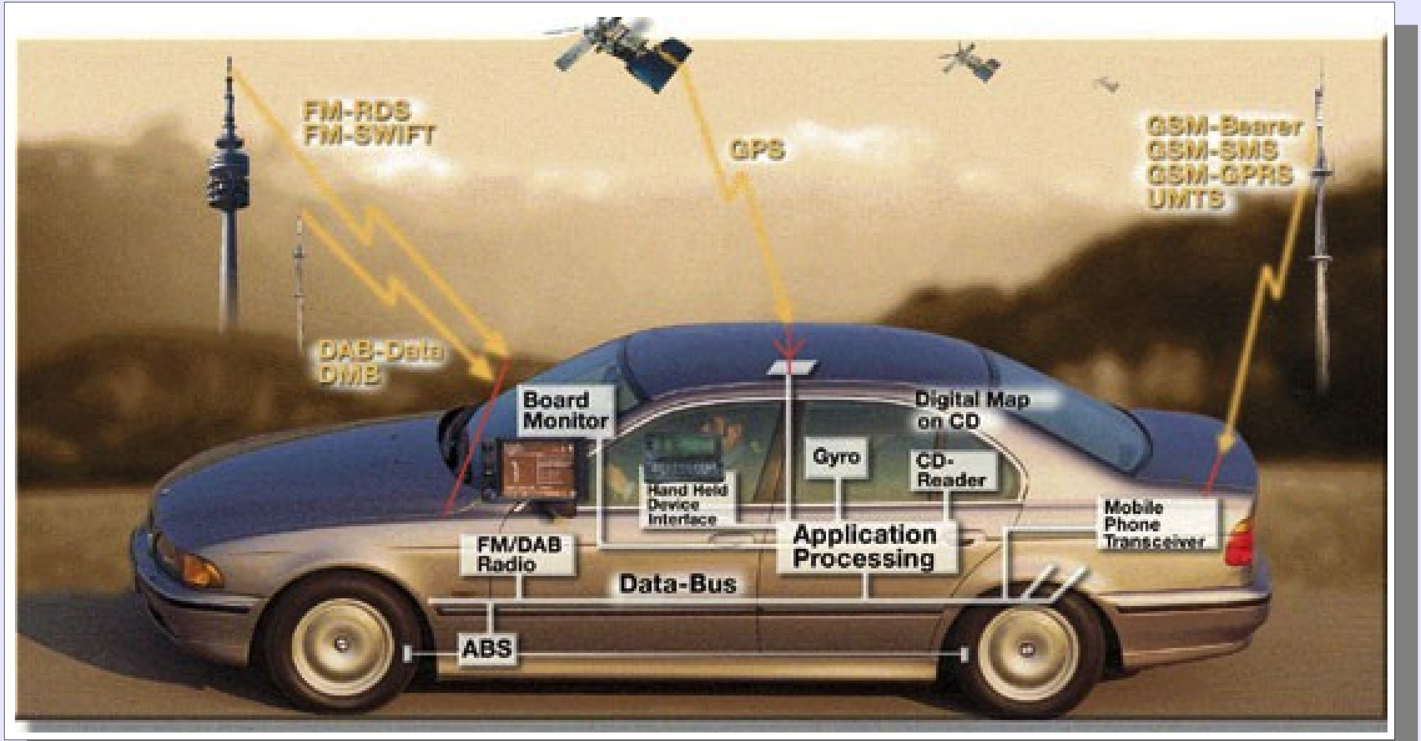
Steganography

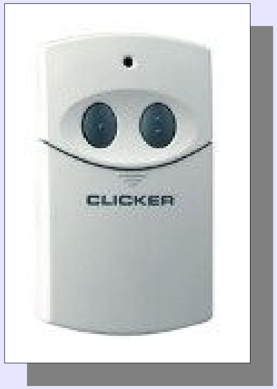


This image of zebras (well, not exactly the resized copy on this slide) encodes a few plays by Shakespeare.

Embedded systems

i.e., operating systems for various appliances, gadgets, sensing devices, etc.





Specifics

Typically cater to a single (possibly multi-threaded) application. Sometimes to a few (fixed) applications.

Protection may not be important or even relevant.

Mass storage (as we know it, i.e., disks) is seldom present. Its role is taken over by flash memory.

Some "file systems" may still be needed.

No virtual memory.

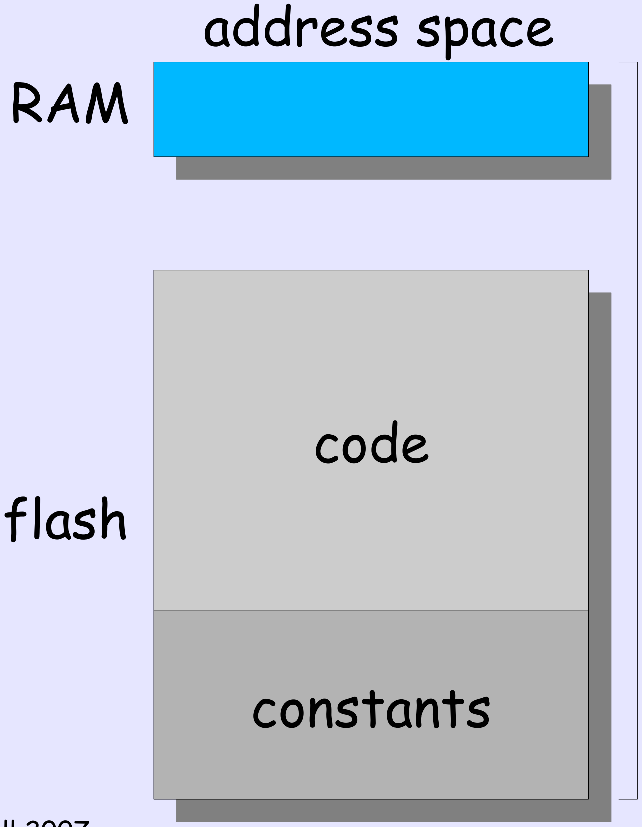
Strongly I/O bound (reactive) behavior.

Small footprint, sometimes must be tiny, e.g., less than 4K of RAM. Code executed directly from flash.

Very small operating systems

Q: How do you program a system that can execute in 2K of RAM? What is your most serious problem, if you want to have multiple "processes"?

A: the stack.



To describe a process, we need:

- room for the PCB
- room for the stack
- room for the context save area (in the PCB or on the stack)

Even though the size of a single stack may be trivial, we have to come up with some allocation scheme for several of them.

TinyOS approach

There are no such things as processes. There are two types of activities:

- hardware event handlers (i.e., pieces of code called to service interrupts)
- so-called tasks, which are chunks of code that can be triggered (scheduled) by hardware event handlers

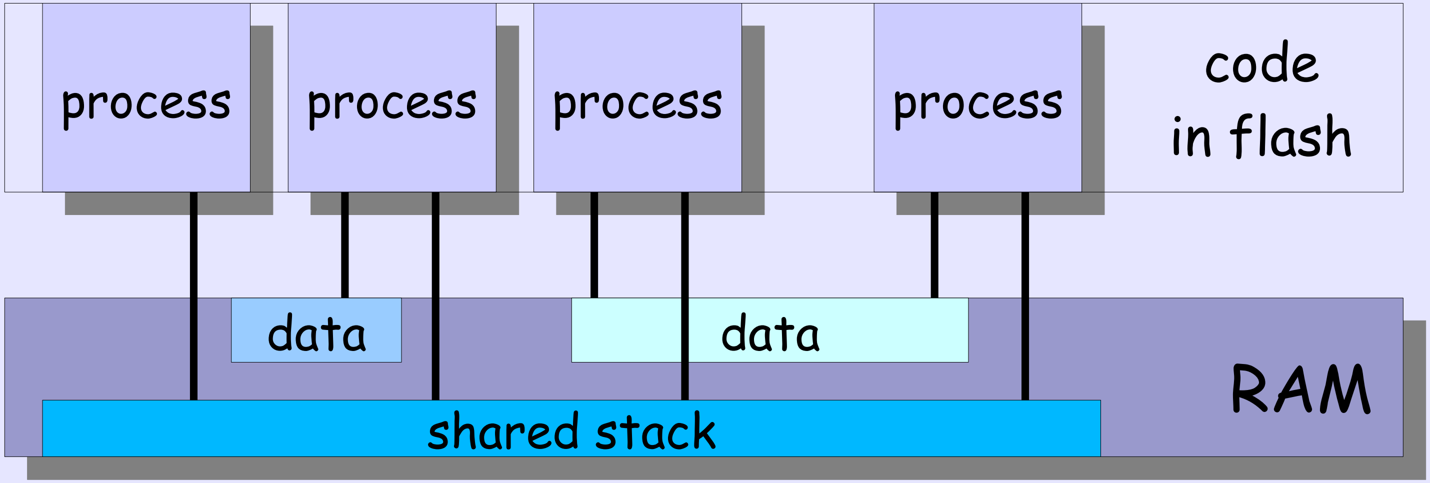
The idea is that tasks are not preemptible by other tasks, i.e., a task starts, runs to the end, and disappears.

Hardware event handlers can preempt tasks.

Owing to the fact that at most one "task" is alive at a time, everything executes on a single stack. Also, there is no need to save the context, except for preserving registers at an interrupt.

PicOS approach

Tasks are preemptible at certain boundaries, of which they are aware. This makes it possible for them to share a single stack.



20 bytes of RAM per PCB

128 bytes of the (global and shared) stack goes a long way

Sample process code

```
process (sniffer, sess_t)
     char c;
     entry (RC_TRY)
          data->packet = tcv_rnp (RC_TRY, efd);
          data->length = tcv_left (packet);
     entry (RC_PASS)
          if (data->user != US_READY) {
               wait (&data->user, RC_PASS);
               delay (1000, RC_LOCKED);
               release;
          }
          c = 1;
          ion (LEDS, CONTROL, &c, LEDS_SET);
          ...
     entry (RC_ENP)
          tcv_endp (data->packet);
          signal (&data->packet);
          proceed (RC_TRY);
endprocess (1)
```

automatic variable, does not survive preemption

private (permanent) data of the process

blocking, i.e., waiting for something to happen or become available

explicit yield

unconditional transition to a different state (not quite equivalent to **goto**)

A process is an automaton (FSM), whose transition function is determined by the events the process wants to perceive.

IPC

```
process (coordinator, bufstr_t)
  entry (MN_START)
    data->mon = fork (monitor, data->buf);
    for (int i = 0; i < data->N; i++) {
      fork (consumer, data->buf);
      fork (producer, data->buf);
    }
  entry (MN_WAIT)
    if (!running (consumer) || !running (producer)) {
      killall (producer);
      killall (consumer);
      kill (data->mon);
      terminate ;
    }
    joinall (consumer, MN_WAIT);
    joinall (producer, MN_WAIT);
endprocess (1)
```

PicOS on MSP430