Computer Systems

Computer Systems

Dr. Steffen van Bakel (Huxley 425)

Lecture time	Monday:	9:00	-	9:50 am	(145)
		10:00	-	10:50 pm	(145)
	Thursday:	17:00	-	17:50 am	(145)
Tutorial	Thursday:	11:00	-	11:50 am	(145)

Questions (outside off lectures or tutorial) best via email; reply will go (anonymised) back to the whole class.

Course Objectives

- Study characteristics of real-time systems.
- Concurrent programming concepts; concurrency versus parallelism and synchronisation.
- What is an operating system, what defines it and distinguishes it from other features of the computer.
- Why it is needed, what does it provide.
- Common concepts that underlie operating systems and real-time systems.
- Detailed investigation of the implementation of main OS functions through the operation system Minix.

Books

Don't depend on only the contents of these slides!

There are several good books around, that all more or less cover the topics of this course. A good choice is:

- Tanenbaum, Modern Operating Systems, Prentice Hall 1992.

(Recommended Book)

The Minix lab will be using

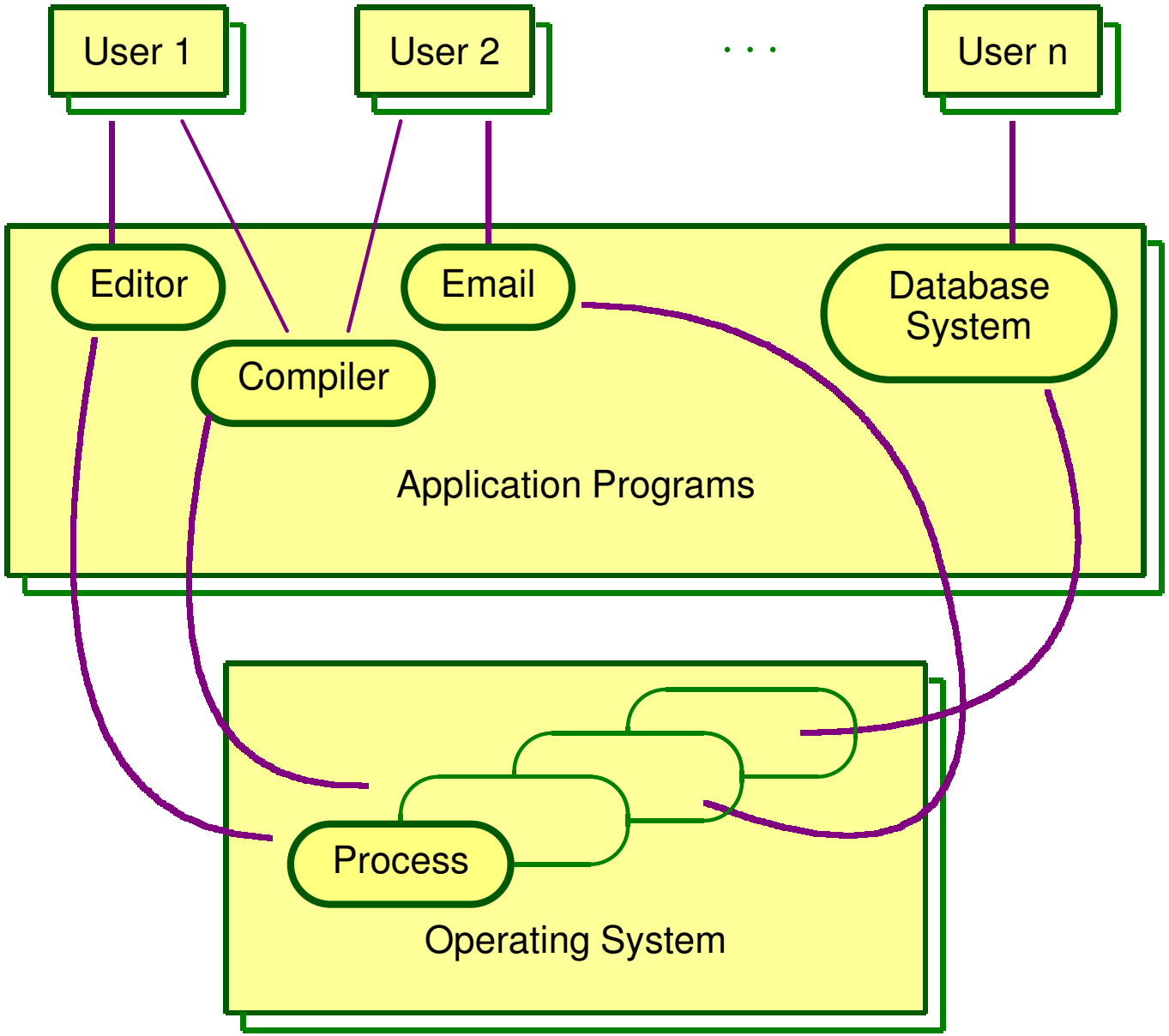
- Tanenbaum - Woodhull, Operating Systems, Design and Implementation (3rd ed.), Prentice Hall 1997.

which discusses almost all the material of the course as well.

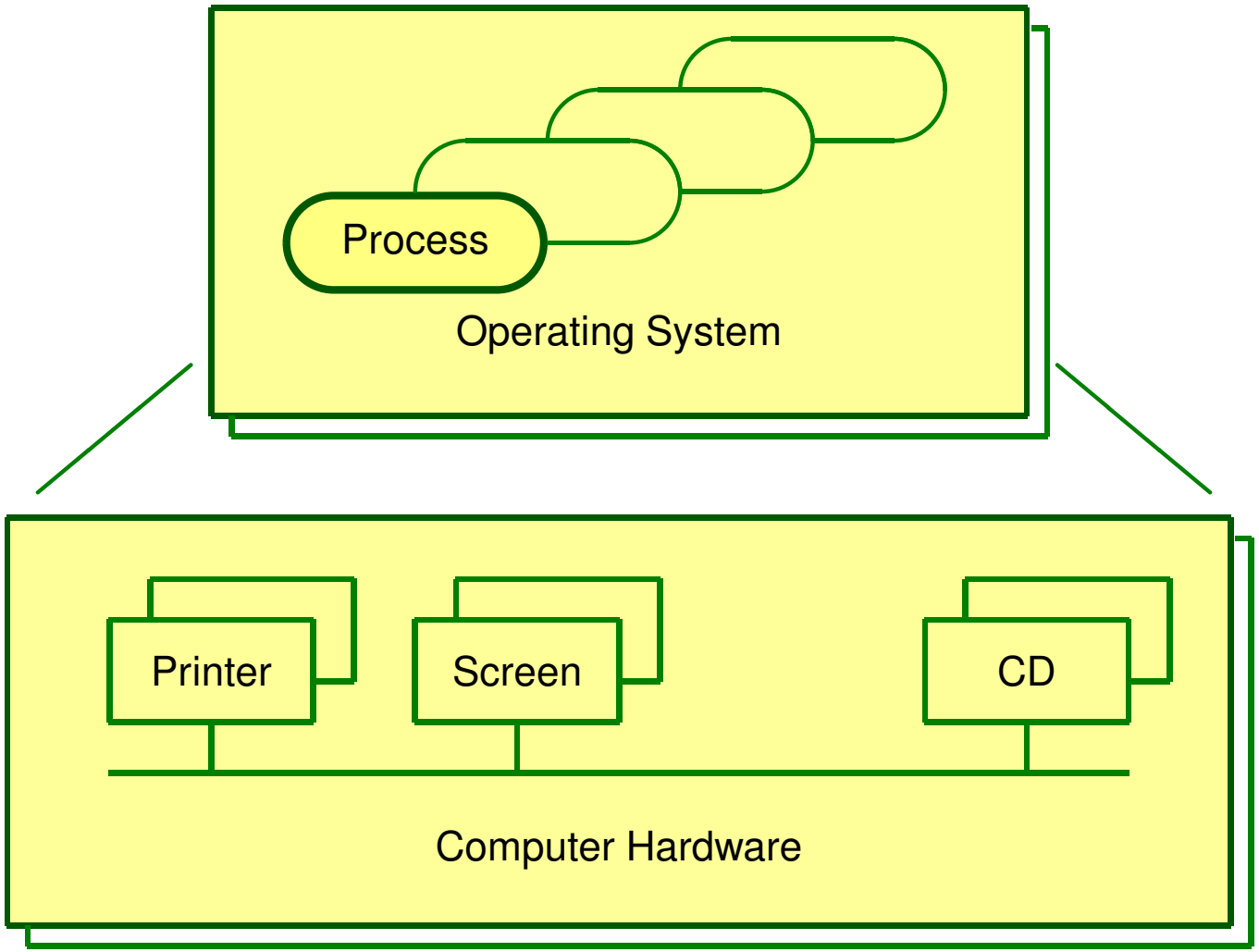
All sources (`.c` files, etc) for the lab all are available online, so you do not need to buy the book for that.

Introduction

Modern Computer System (top)



Modern Computer System (bottom)



Resource Management

- Making efficient use of the (limited) resources that are available: Time; Space; Money.
- Sharing resources among multiple users: Schedule access; Fair allocation; Prevent interference.

Microcomputers are cheap, and can easily be used for one single tasks: dedicated; Organisers, Mobile phones, Cars, Washing machines, ...

Large mainframes are expensive, and are therefore often shared by many users (e.g.: DoC 2012 batch server: IBM system x3750 m4, 512GB RAM, 8TB disk; it has 32 cores, hyper-threading making it appear to have 64: approx. £7,700.50).

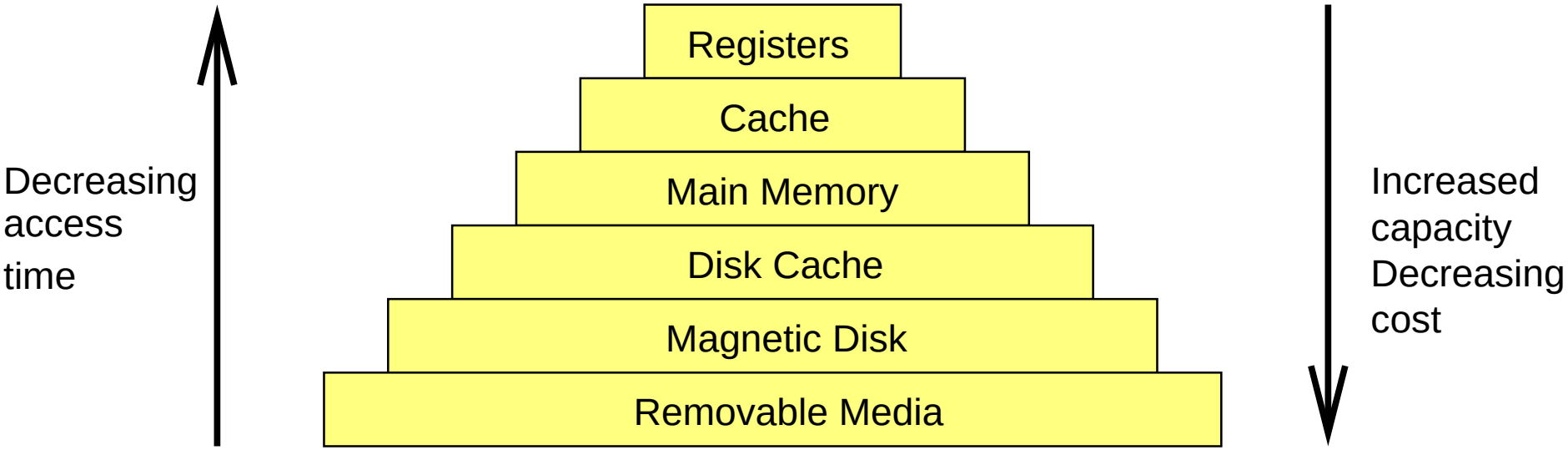
Resources

- Processors; divide a number and/or time.
- Memory; **RAM**, Cache, Disk, ...
- Input / Output Devices; Terminals (1 per user), Printers (more users), ...
- Communications Devices; Networks, Servers, Satellite dishes, ...
- Internal devices; Clocks, Timers.
- Long-term storage (files); Disks, Tapes, CD, ...
- Software; Compilers, Editors, E-mail, Database, ...
- Processes - programs in execution.

Memory

Design considerations:

- How much ?
- How fast ?
- How expensive ?
- How volatile ?



Memory (2)

Volatile:

Registers – Speed: 9 ns
- Capacity: 32-128 bytes

L1 cache – Speed: 9 ns
- Capacity: 128 KB

L2 cache – Speed: 20 ns
- Capacity: 3 MB

Main memory – Speed: 70 ns
- Capacity: 500 GB

Non-volatile:

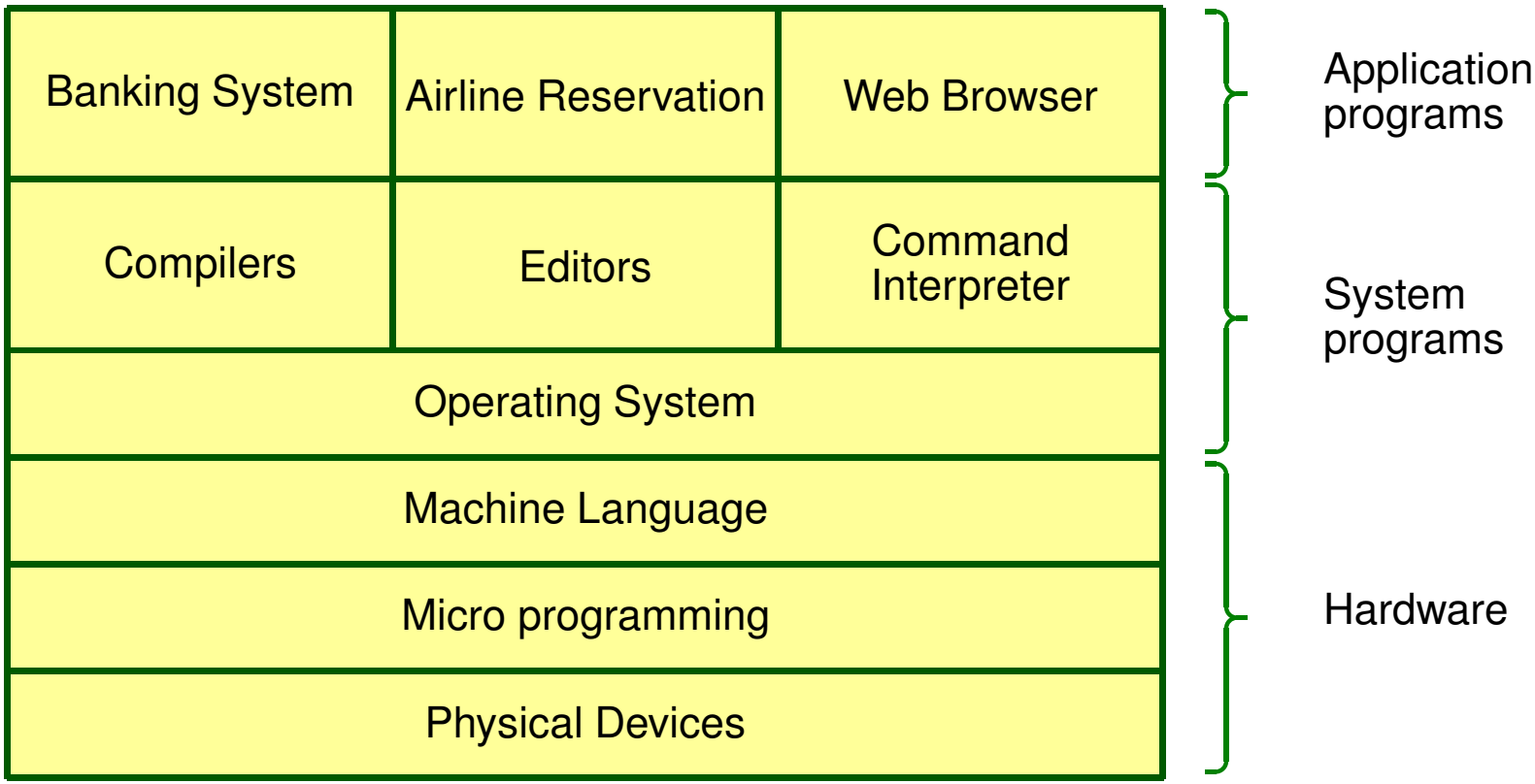
Magnetic disks – Speed: 10 ms
- Capacity: 20 GB

Optical disks (CD, DVD) – Speed: $>$ 100 ms
- Capacity: up to several GB

Removable media – Speed: $\gg$ 100 ms
- Capacity: up to several TB

Layered Machine

O.S. converts raw hardware into a usable computer system. There are various levels (not all parallel) of (abstract) machines in a computer.



Programming Levels

Micro programming Often fixed into the **CPU**, the primitive language of the computer, that interprets the basic instructions like **`add`**, **`move`**, . . . , and fetches the right arguments for each.

Machine language Set of instructions that **CPU**'s *Micro program* can execute.

Device instructions Every device comes with its own instruction set, like **`seek`**, **`move`**, and parameters like **`tracknr`**, **`headnr`**, for disks. Instructions must be loaded in specific addresses in main memory (memory ports), and will then be executed by controller of device, which is often another **CPU**.

Operating System Functions

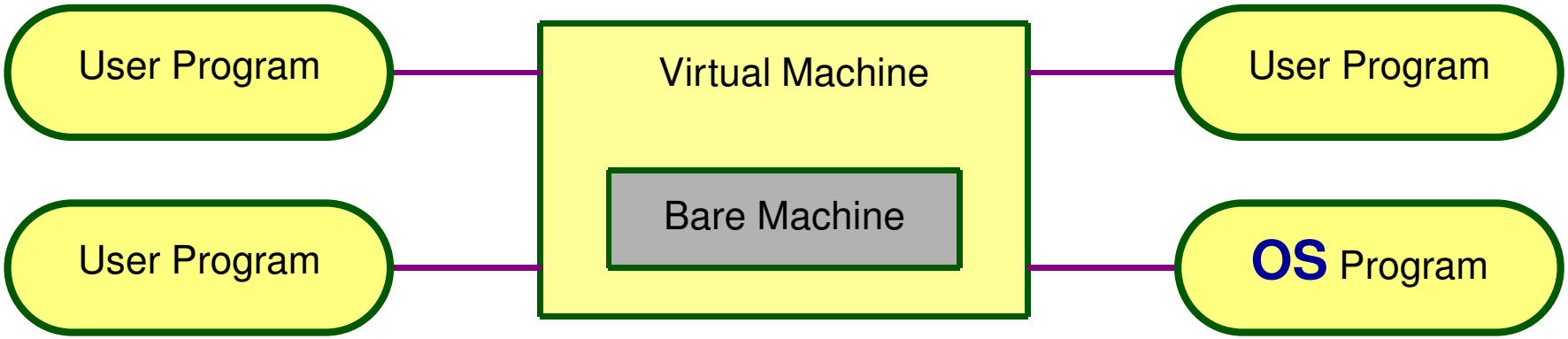
Operating System A language that offers various parts of the computer as independent entities. It organises the data for the user in files; using calls like **`read`**, **`write`**, **`open`** it is possible to access that data without having accurate information on where a file is located.

These instructions are implemented as procedures, normally grouped in a *system call library*, written in a combination of the *machine language* (perhaps using a compiler), and the *device instruction* sets.

Command interpreters A program that is normally distinct from the **OS**. It interprets the commands typed by the user, by executing programs that are associated to the instructions.

Application programs These form a group of programs, that are (almost) independent of the machine, and are often exchangeable by others; banking system, games, LaTeX, ...

Provision of a Virtual Machine



The details of the machine should be *kept hidden* from users and user programs.

All access to hardware resources should be the *exclusive right* of the **OS**, and request from users should be as *abstract* as possible, i.e. users should not need to know *how* a file is stored on disk.

Extended / Virtual Machine Facilities

Simplified I/O Device independence; open a file on floppy, tape or hard-disk is *one* operation.

Virtual Memory Larger than real or partitioned.

Filing System Long term storage, on disk or tape, accessed by symbolic names.

Program Interaction and communication semaphores, locks, monitors, *messages*.

Network communication Packaging a file to send, routing.

Protection Prevent programs accessing resources not allocated to them.

Program Control User interaction with programs; command language, shells.

Accounting & Management Information

OS *Characteristics*

Concurrency Several simultaneous parallel activities:

- overlapped **I/O** & computation;
- multiple user and **OS** programs running in parallel.

Important issues are:

- Switch activities at arbitrary times:
  - to guarantee fairness;
  - prompt replies.
- Synchronise actions:
  - avoid long waiting cycles;
  - accurate error handling.
- Protection from interference:
  - each process its own space;
  - re-entrant code.

OS *Characteristics (2)*

Non-determinacy Results from (unforeseen) events occurring in unpredictable order. Examples are: timer interrupt, terminal input (users need to think), program error, faults in the network, disk errors, ...

We are in the real world, therefore **OS** must expect and cater for non-determinacy

Sharing To manage sharing of data, programs and hardware, some issues are:

- Resource allocation; economical and fair use of memory, **CPU** time, disk space, ...
- Simultaneous access to resources; disks, **RAM**, code, network, **CPU**, ...
- Mutual exclusion; guarantee that the risky parts are protected, and no un-authorised access is possible.
- Protection against corruption.

OS *Characteristics (3)*

Long term storage Filing systems like disks, tapes, ... should take care of:

- Easy access to files by user defined names; directory structure, links, shared disks;
- Access controls; read, write, remove, execute, or copy permission;
- Protection against failure through Backups:
  daily / weekly / monthly, and
  partial or complete;
- Storage management for easy expansion etc: add disks without need for re-compilation of **OS**.

Parallel Operating Systems

Tightly coupled – multiple processors;
- shared resources, i.e. memory, disk and **I/O** devices;
- communication via shared memory.

Increased throughput – speed-up ratio with n processors is not n but less than n;
- increased overheads for management of multiple processors;
- contention for resources, i.e. memory or disk access.

Increased reliability – fault tolerance;
- grace degradation.

Parallel Operating Systems (2)

Symmetric multiprocessing – each processor runs a copy of the operating system;

- many processes (up to maximum) can run at the same time without performance deterioration;
- **OS** easy to design, similar to uniprocessor **OS**;
- no load balancing, i.e. processor no. 1 might be overloaded while processor no. 2 might be idle.

Asymmetric multiprocessing – master processor performs scheduling;

- slave processors perform tasks allocated by master server;
- often used for massive parallel systems;
- load balancing possible;
- **OS** is more complicated to design.

- Dedicated slave processors becoming more common, i.e. processors managing disk system.

Distributed Operating Systems

- *Loosely* coupled:
  - independent machines, i.e. processor, memory, disk;
  - communication via bus, network, phone lines.
- Resource sharing.
- Computation speed-up.
- Reliability.
- Communications.

Real-time Operating Systems

- Special purpose operating systems.
- Scheduling: maximise turnaround time.

Hard real-time systems – real-time tasks are guaranteed to complete in fixed time with support for deadlines;
- limited or secondary storage;
- applications: control systems for industry, medical equipment or engines.

Soft real-time systems – real-time tasks are guaranteed to have priority, and no support for deadlines;
- secondary storage;
- applications: Multimedia, Augmented and Virtual Reality, Remote Sensing.

- Components of soft-real functionality can be found in modern **OS** like UNIX or Windows NT.

Basic Concepts

Computer Systems Overview

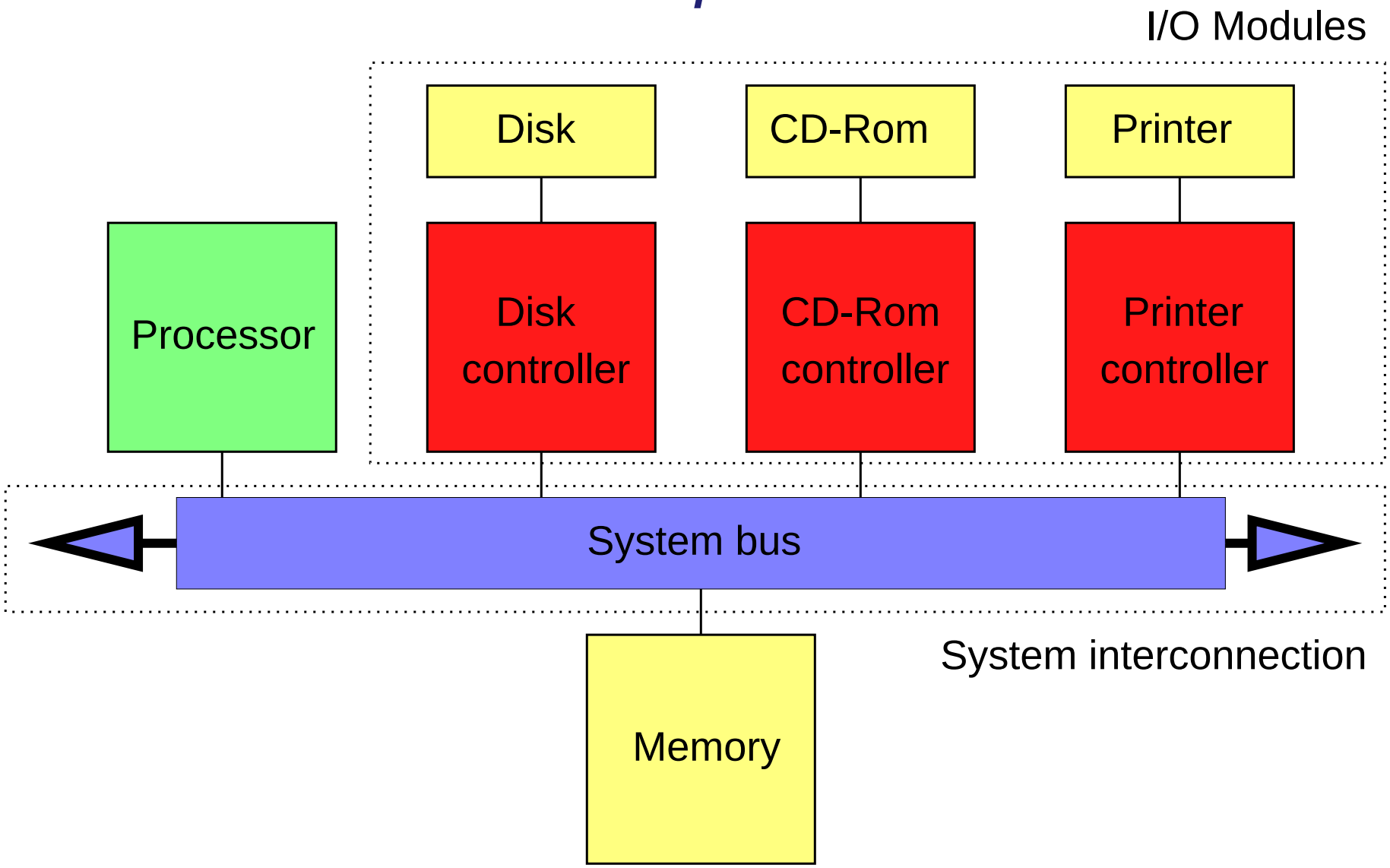
Processor CPU – controls computer hardware;
- executes instructions and programs.

Memory – stores data;
- stores programs.

I/O modules – read and write from **I/O** devices;
- **I/O** controller;
- **I/O** devices.

System interconnection
- connects the different hardware components via bus;
- provides communication between hardware components.

Computer


I/O Modules
Disk
CD-Rom
Printer
Processor
Disk controller
CD-Rom controller
Printer controller
System bus
System interconnection
Memory

CPU

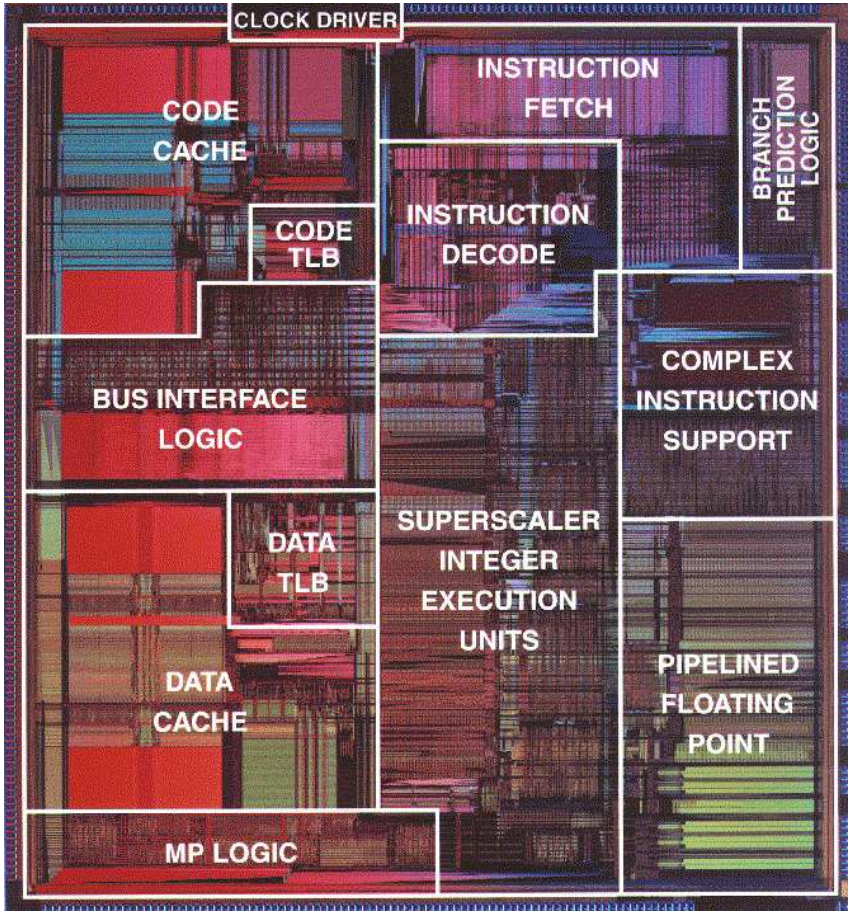
Registers – general purpose registers;
– control and status registers.

Cache – data cache;
– instruction cache.

Fetch Unit

Decode Unit

Execute Unit – integer execution unit;
– floating point execution unit.



Example: Intel Pentium III

Registers

- Registers are very fast memory locations *inside* the **CPU**.
- General purpose registers:

  Data Registers to calculate with;

  Address Registers contain addresses of memory locations of data and instructions.

  Can be modified explicitly by user (assembler) programs.
- Control and status registers:

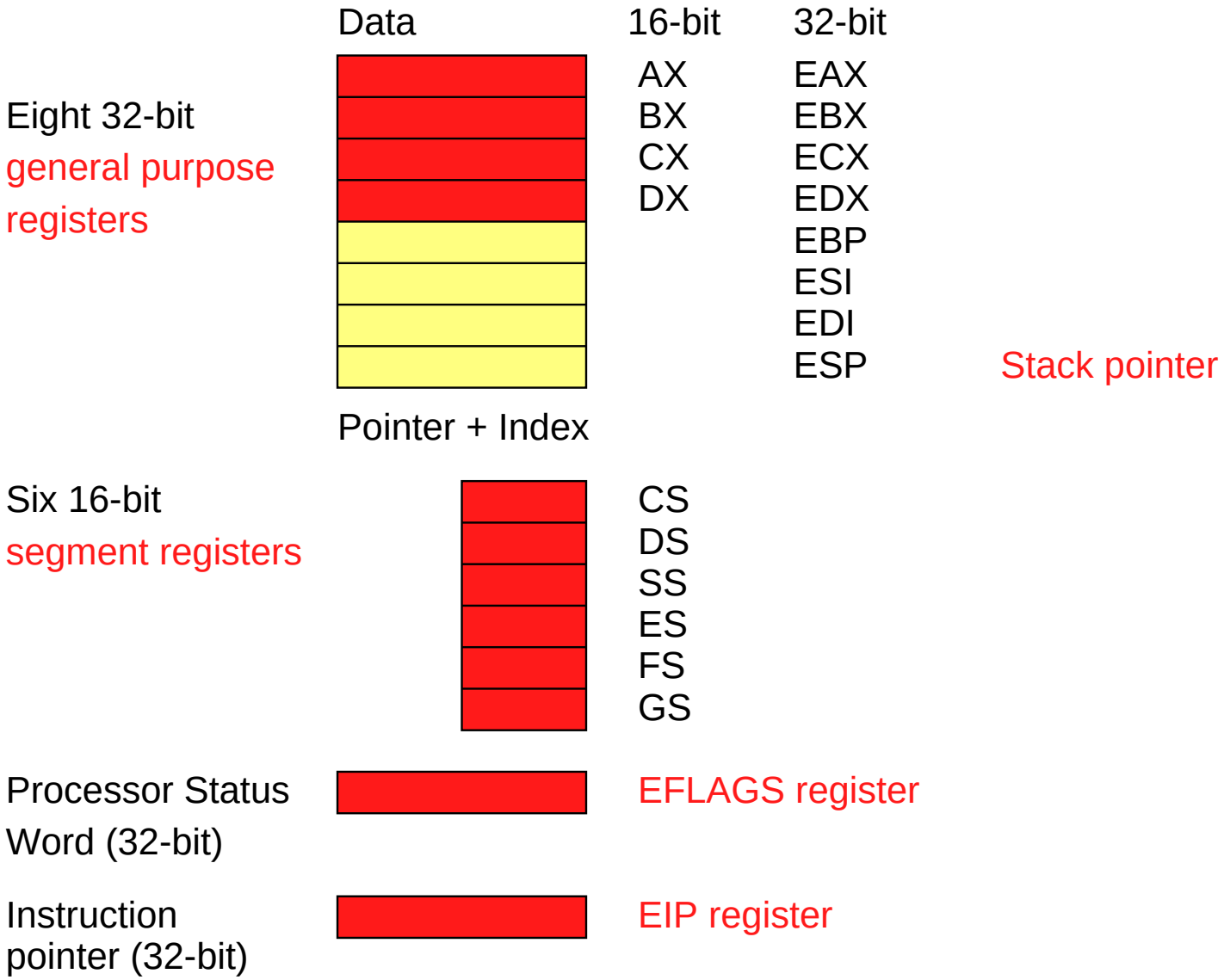
  Program Counter special register which contains the address of the next instruction to be fetched;

  Instruction register special register which holds the most recently fetched instruction;

  Process status word special register which contains control and status information about the **CPU**.

  These are modified implicitly by running user program.

Intel Pentium



Intel Pentium (2)

Segment registers – `CS` contains code segment;

- `SS` contains stack segment;
- `DS`, `ES`, `FS`, `GS` contain data segment.

Different data segments for static, dynamic, shared and exported data segments.

`EFLAGS` contains a number of flags indicating the state of the **CPU**:

Flag	Purpose	Bit
`CF`	carry flag	0
`PF`	parity flag	2
`ZF`	zero flag	6
`SF`	sign flag	7
`IF`	interrupt enable flag	9
`OF`	overflow flag	11
`VIF`	virtual interrupt flag	19
`VIP`	virtual interrupt pending	20

Input / Output Communication

- **I/O** communication allows to read and write data from devices including:
  - input devices: Keyboard, Mouse, etc;
  - output devices: Display, Printer, etc;
  - input and output devices: Disk, Network card, Modem, etc.
- **I/O** communication techniques:
  - programmed **I/O**;
  - interrupt driven **I/O**;
  - Direct Memory Access (**DMA**).
- **I/O** communication instructions:
  - test instructions;
  - control instructions;
  - read and write instructions.

Why interrupts?

For key-strokes: do you wait (*poll*, i.e. actively testing the input) for key-strokes to arrive?

For printing: do you let the **CPU** print character-by-character, or send a block and wait for it to finish?

What if another instruction arrives before you've finished processing the first?

- we introduce *interrupts*: this is a mechanism that, on an event, 'grabs' the **CPU**: i.e. the current program's execution is 'interrupted', and the **CPU** deals with the event;
- using interrupts, we need not poll for **I/O**, the **CPU** will be forced to deal with it, and the **CPU**'s time is not wasted with waiting;
- we need to guarantee that the original program can continue 'undamaged', i.e. is not improperly influenced by the interruption.

Interrupts are rumoured to be invented by Edsger W. Dijkstra.

Intermezzo on Dijkstra

“ While he had programmed extensively in machine code in the 1950s, he was known for his low opinion of the `GOTO` statement in computer programming, writing a paper in 1965, and culminating in the 1968 article ”A Case against the `GO TO` Statement”, regarded as a major step towards the widespread deprecation of the `GOTO` statement and its effective replacement by structured control constructs, such as the while loop.

This article was retitled by editor Niklaus Wirth to ”`GO TO` Statement Considered Harmful”, which introduced the phrase ”considered harmful” in computing. This methodology was also called structured programming, the title of his 1972 book, coauthored with C.A.R. Hoare and Ole-Johan Dahl.

He worked on the compiler for Algol’60, one of the first compilers supporting recursion, and strongly opposed the teaching of BASIC. ”

(taken off Wikipedia)

I/O *Function Overview*

Programmed I/O – **CPU** issues **I/O** command to **I/O** module;
- **CPU** 'busy-waits' for the operation to be completed before proceeding.

Interrupt-driven I/O – best with process structure;
- **CPU** issues **I/O**; if no need to wait on **I/O**, process does not halt and continues to execute subsequent instructions;
- interrupted by **I/O** module when finished;
- if need to wait on **I/O** it suspends, and other job is processed;
- **CPU** will be *interrupted* by **I/O** module when finished.

Direct Memory Access (DMA) – a **DMA** module controls exchange of data between memory and **I/O** module;
- **I/O** module reads or writes directly into **RAM**;
- **CPU** sends request for data block to **DMA**; interrupted only after entire block has been transferred.

DMA is found in most systems.

Programmed vs Interrupt Driven **I/O**

Programmed – after **CPU** starts **I/O**, it enters loop and tests status of **I/O** device; **CPU** exits loop if status of **I/O** devices shows that the read or write request has been completed;

- advantage: simple to program (used up to Windows NT).
- disadvantages:
  - **CPU** must manage **I/O** itself;
  - **CPU** wastes time for busy waiting;
  - **CPU** speed limits **I/O** transfer rate: **I/O** goes through **CPU**.

  This gives a severe performance degradation: **CPU** is millions of times faster than the device.

Interrupt driven – after **CPU** starts **I/O**, it continues;

- **I/O** device will interrupt **CPU** when request is complete;
- advantages:
  - **CPU** can do something else;
  - **CPU** wastes no time for busy waiting.

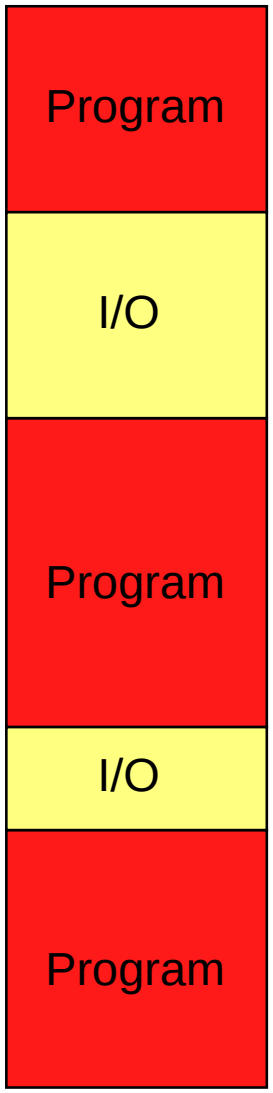
Direct Memory Access

- **CPU** issues read or write instructions specifying the memory address where to read from or write to the data.
- **CPU** can do something else.
- **DMA** module manages transfer directly from **I/O** device to memory; it 'steals' the bus: suspends the **CPU**.
- **DMA** module will interrupt **CPU** when request is complete.

Advantages **I/O** does not go via **CPU**.

Disadvantages
- Competition between **CPU** and **DMA** module for bus usage.
- **DMA** module has priority, **CPU** may execute more slowly during **DMA** transfer.

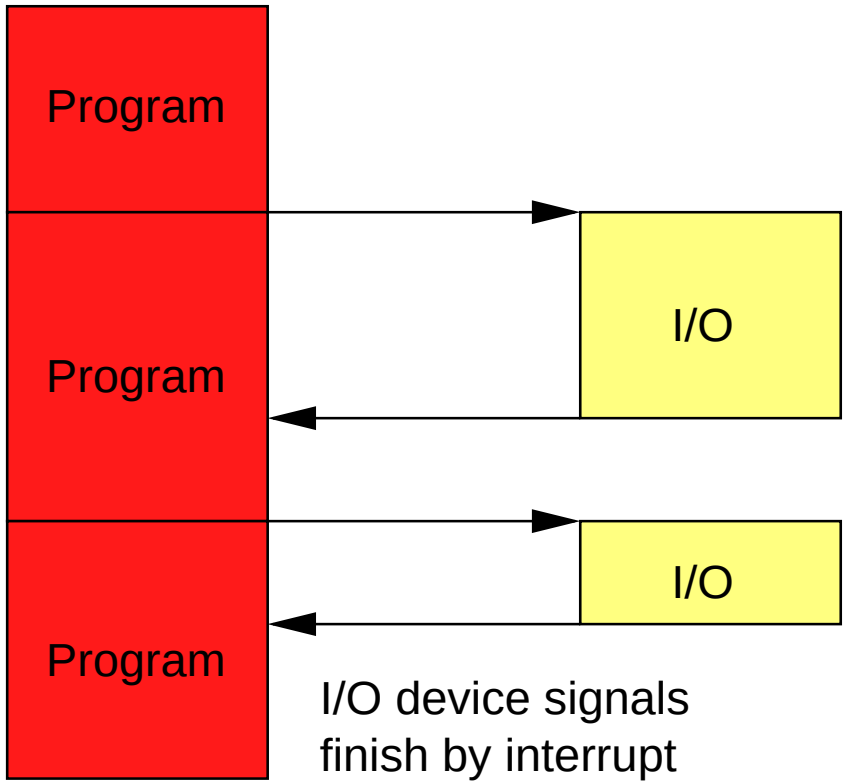
Effect of Interrupts



Program is busy waiting for I/O device to finish

Program is busy waiting for I/O device to finish

Programmed interrupts



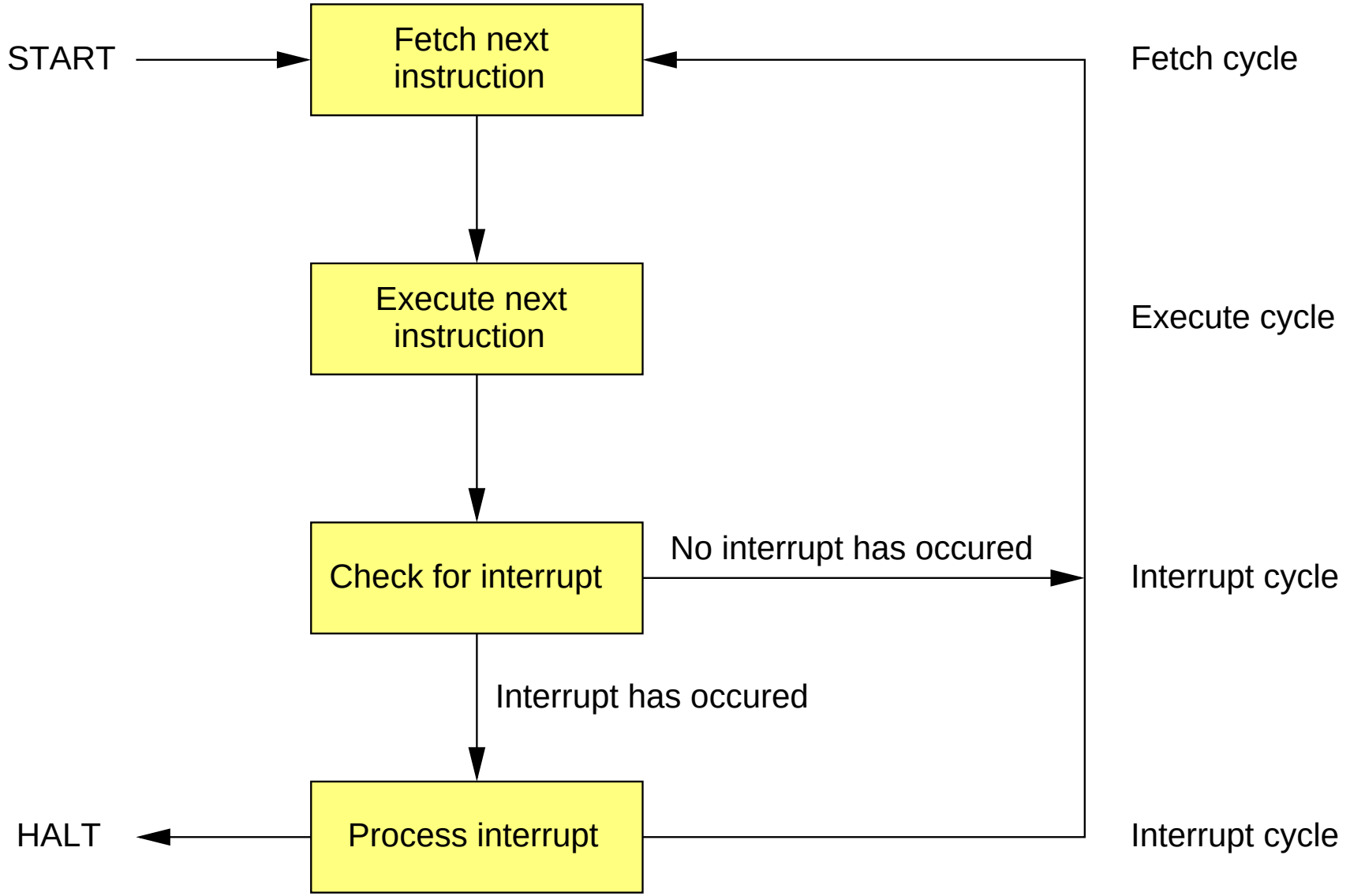
I/O device signals finish by interrupt

Interrupt driven

Instruction Cycle (Classic)

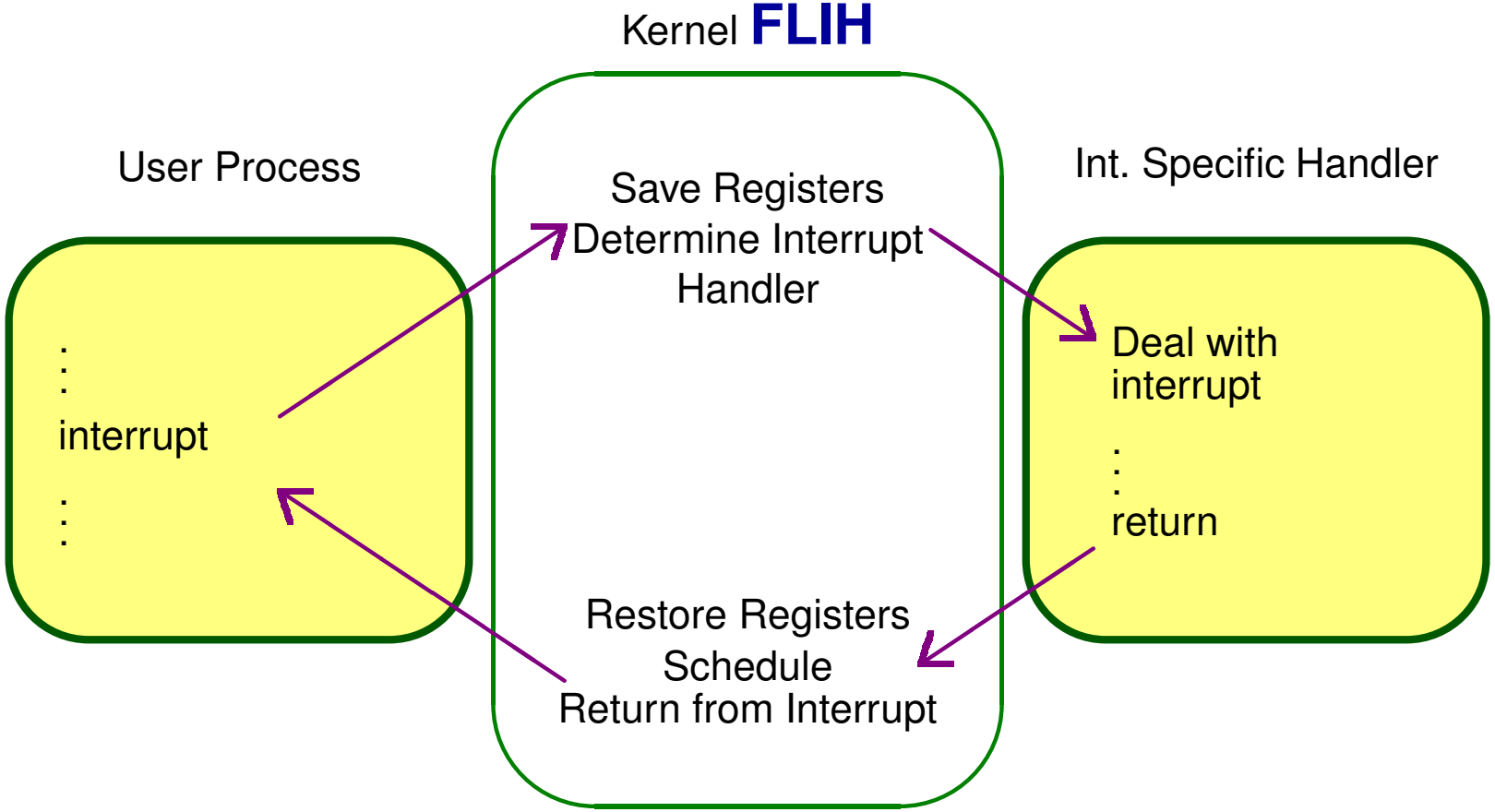
- Instructions in a *high-level* language (i.e. C++, Java, or Haskell) are broken into a set of *machine* instructions (i.e. *assembler*).
- Machine instructions are executed by **CPU** as a set of *fetch* and *execute* microsteps:
  - fetch an instruction from memory;
  - determine its type (`move`, or `add`, ...);
  - fetch the operands;
  - execute the instruction;
  - check for interrupts.
- If an interrupt has been found, the **CPU** *automatically*
  - pushes `PC` and `PSW` on the stack;
  - switches to kernel mode;
  - uses device number to index memory to find handler;
  - switches `PC` to start address of handler.

Fetch and Execute Cycle Flow Diagram



*First Level Interrupt Handler (**FLIH**)*

- Save registers of current process in **`PCB`**.
- Determine source of interrupt.
- Initiate service of interrupt; call **OS** interrupt handler.



FLIH *(2)*

- The **FLIH** is *hardware dependent* and therefore implemented in assembler.
- **FLIH** must run with interrupts disabled.
- Enable / disable interrupts by priority level in **`PSW`**.
- Interrupt handler may call kernel to unblock another process (e.g. one that was waiting for **I/O**) which could result in a different process rather than interrupted one eventually being run by scheduler.
- Handler could be a kernel procedure to service a normal kernel call (send a message, delay process).

Second Level Interrupt Handling

This is software, specified in the **OS** code (written in **`C`** or **`C++`**).

- In the handler, the state of the **CPU** (its *registers* etc) is saved, and the actions are performed that deal with properly servicing the interrupt.
- After dealing with the interrupt, interrupts are re-enabled, and normally the original, interrupted computation must be resumed: registers are restored to their original values, and **`PC`** is reset.
- Care is needed:
  - if another interrupt arrises (perhaps from another device), simultaneously or shortly afterward (during the handling of this interrupt), it is handled properly: no interrupts should get lost;
  - if another interrupt occurs while processing one, the saved **`PC`** is not overwritten.

Interrupts: How

The **CPU** has two special features: a bit, the *interrupt flag*, and a group of bits, the *interrupt mask*, with a bit for each kind of device.

- When a device wants to notify the **CPU**, it raises the flag in the **PSW** (it has direct access); it is allowed to do that if *its* bit in the *interrupt mask* is low.
- In each instruction cycle, the **CPU** tests for interrupts by checking (this is part of the microprogramming) the flag; if an interrupt needs dealing with, it:
  - saves the current **PC** value (part of the micro program; to be able to resume whatever it was doing);
  - switches off interrupts (do not bother me now, I'm busy!), by raising bits in the mask; and
  - jumps to the routine that specifies how to deal with the interrupt (the *interrupt handler*).

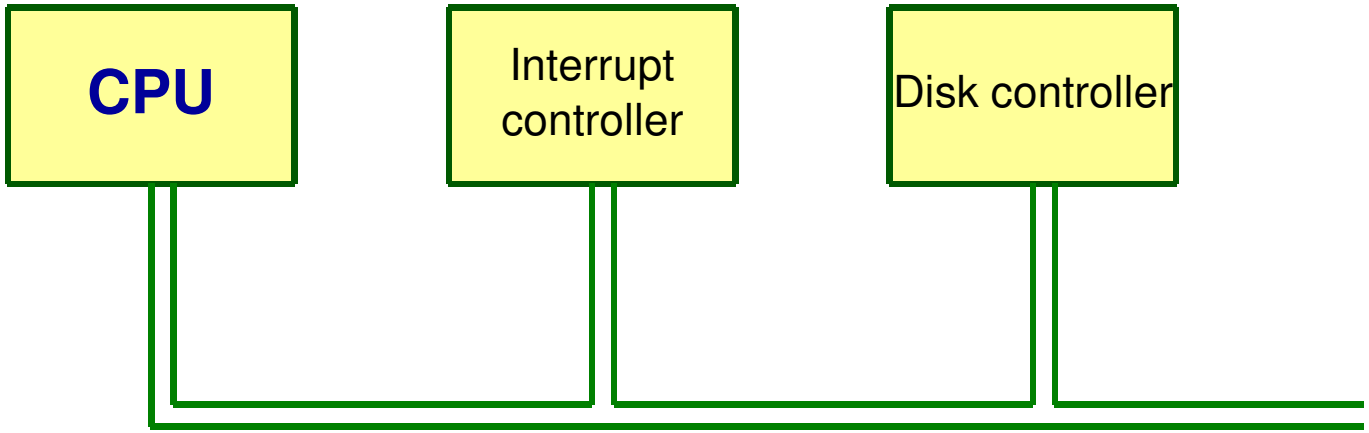
Fetch and Execute Cycle

(This is microprogramming)

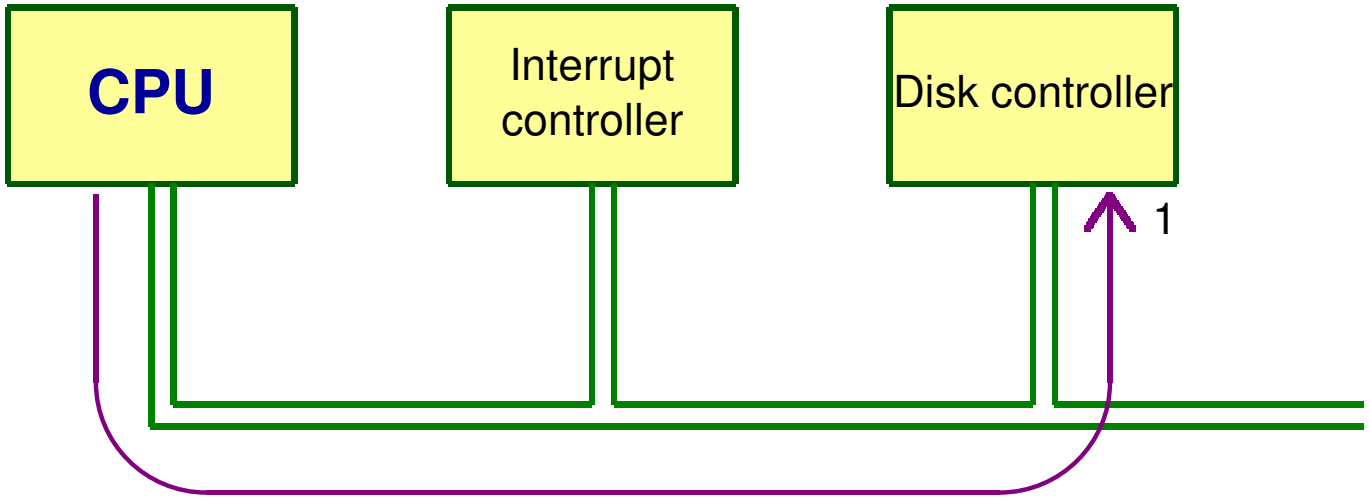
```
register IR;                /* Instruction register */
register PC;                /* Program Counter */
register PSW;               /* Processor status word */

for(;;) {
  IR = fetch(PC) ;          /* Fetch instruction from address in PC */
  PC += 1;                  /* Increment PC */
  decode(IR);               /* Decode fetched instruction in IR */
  execute(IR);              /* Execute fetched instruction in IR */
  if (PSW[IF])              /* Check if interrupt pending flag is set { */
    PSW[IF] = FALSE;        /* Unset interrupt pending flag */
    push(PSW);              /* Push PSW on control stack */
    push(PC);               /* Push PC on control stack */
    PC = FLIH;              /* Load address of interrupt handler */
  }
}
```

Interrupts

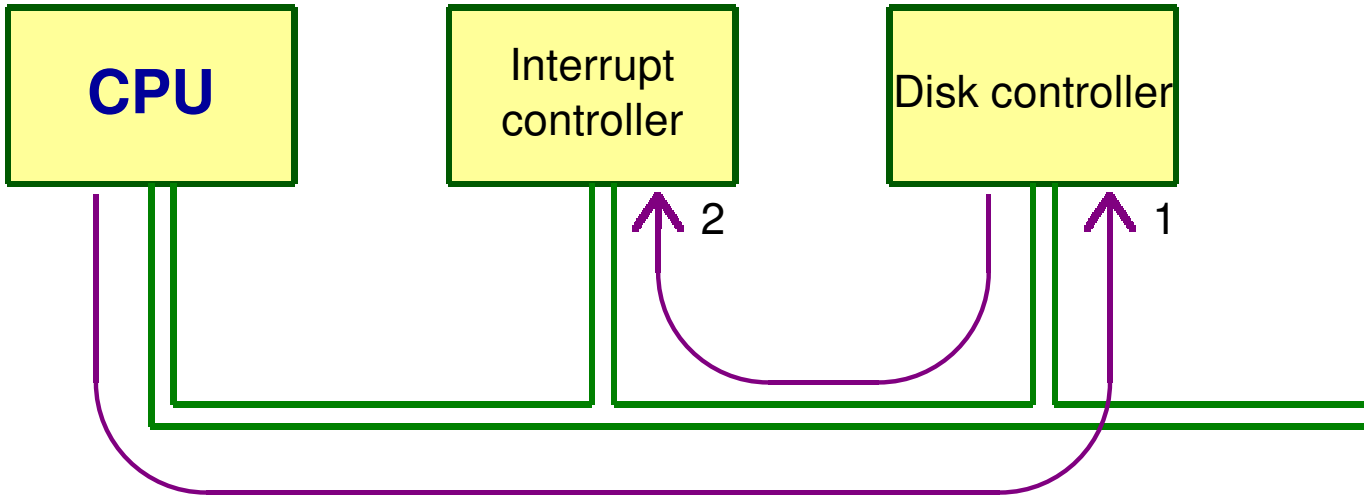


Interrupts



1 Driver writes instruction into disk controller;

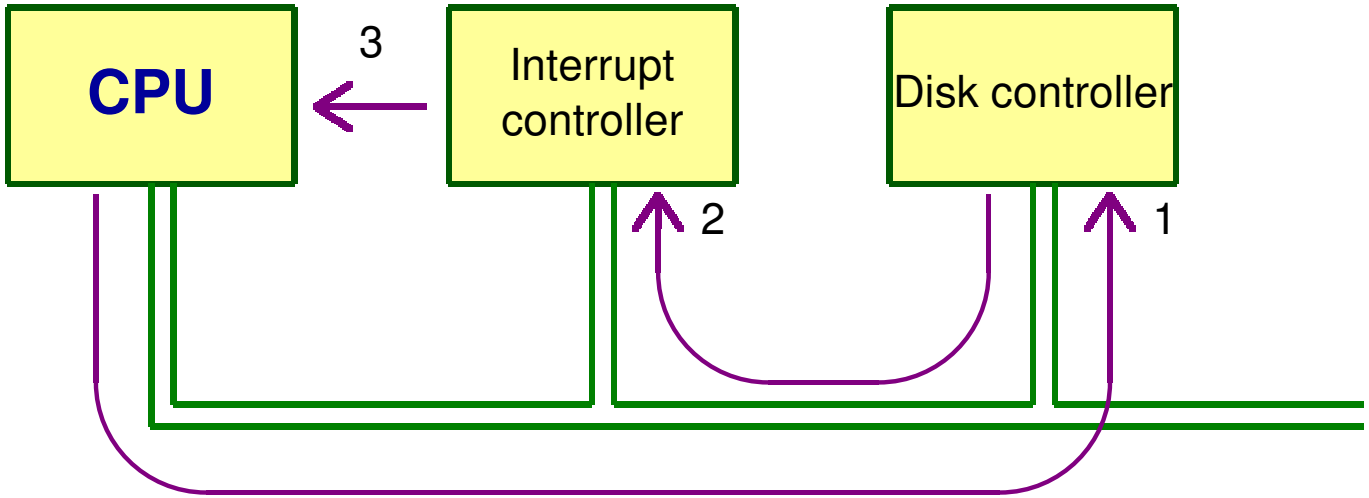
Interrupts



1 Driver writes instruction into disk controller;

2 On finishing, disk controller signals interrupt controller;

Interrupts

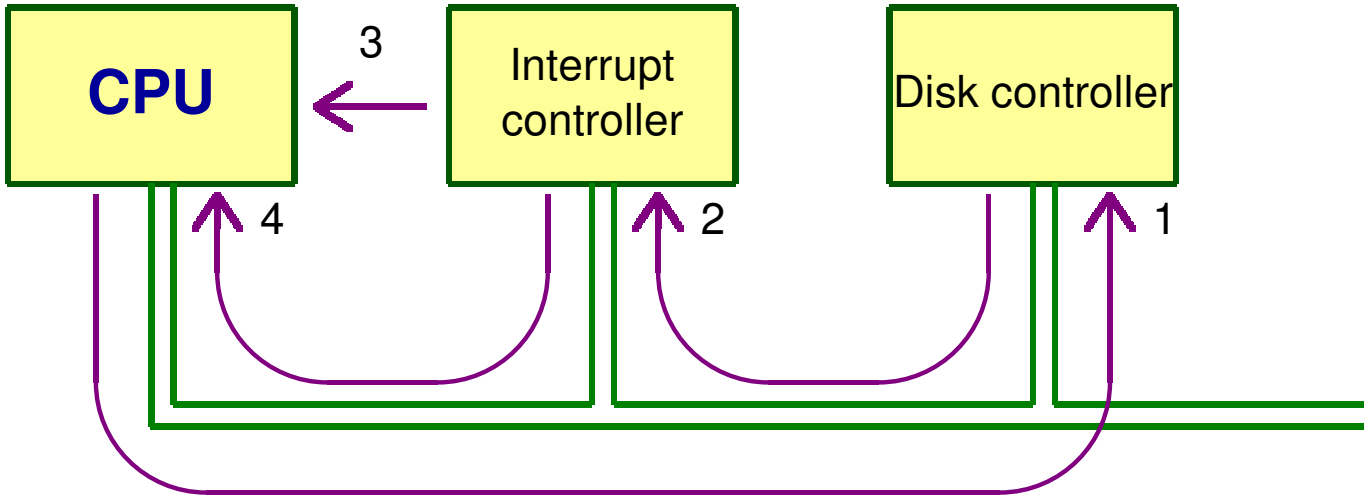


1 Driver writes instruction into disk controller;

2 On finishing, disk controller signals interrupt controller;

3 Raise a bit on **CPU**, informing of interrupt;

Interrupts



1 Driver writes instruction into disk controller;

2 On finishing, disk controller signals interrupt controller;

3 Raise a bit on **CPU**, informing of interrupt;

4 Put number of device on the bus.

Problems with Interrupts

Processing interrupts is hazardous; **CPU** has to save *at least* the **PC**, and perhaps all registers. But where?

- In internal registers on the **CPU** (so not the normal registers), that the **OS** can access as well; this blocks nested interrupts, long periods of blocked interrupts.
- On stack: whose stack?

  User Stack ∗ stack pointer **SP** might not be legal.
  ∗ **SP** might point to end of page; *page fault*.

  Kernel stack ∗ safer, since **SP** very likely legal, and page fixed (pinned) in memory;
  ∗ requires switching to kernel mode (time consuming).

Interrupts vs. Exceptions

- *Interrupts* are *asynchronous* events:

  Hardware clock or **I/O** devices (disk-, network controller);

  Software `int`errupt instruction.

- *Exceptions* are *synchronous*:

  Traps execution is continued after instruction which caused the exception (overflow, division by zero);

  Faults execution is continued before instruction which caused the exception (memory faults);

  Aborts execution is aborted.

- Interrupts and exceptions are implemented by hardware: modified fetch and execute cycle.
- Processing of interrupts and exceptions is essentially identical.

Vectored Interrupts (Intel)

Kernel sets up vector table of interrupt handlers. When device generates an interrupt it also produces the vector number to identify itself. (Trap instruction is a software interrupt.) Then the current **PC** and **PSW** are put on stack, and the new **PC** and **PSW** are loaded from the vector table.

(This is done by microprogramming.)

Vector Table

Vectored Interrupts (2)

If all interrupt handling in the Kernel is done by assembly language procedures, then interrupt handlers.

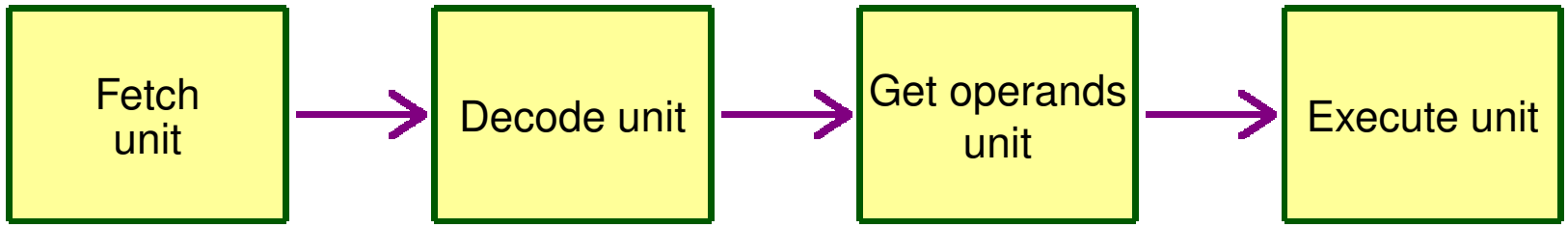
1. save and restore interrupted program's context;
2. perform device dependent processing.

Note that then there is no software **FLIH**.

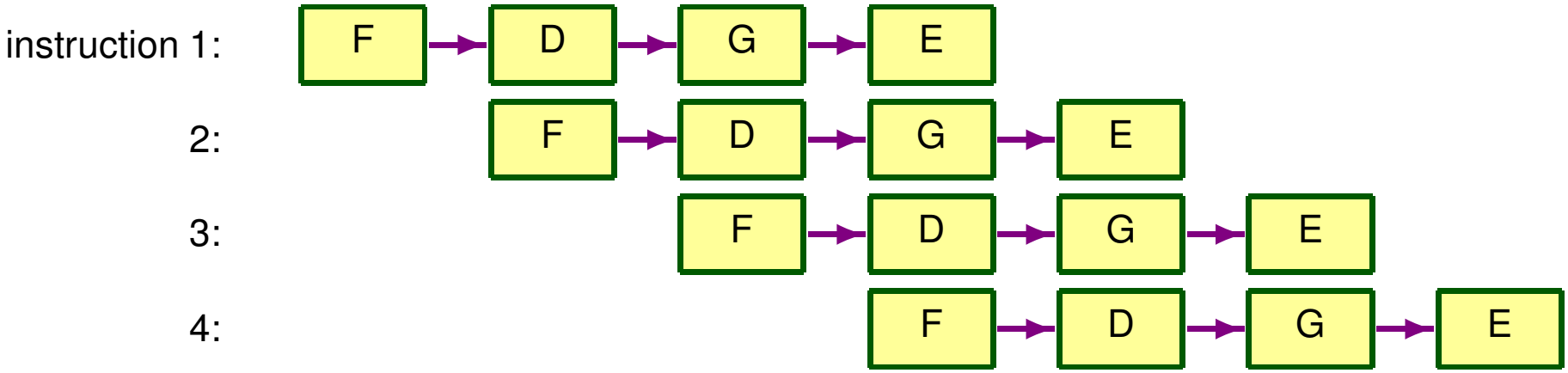
But interrupt handlers should be written in high level language and be part of device handler process - particularly for real-time systems. This requires a **FLIH** in kernel to perform actions common to all interrupt handling.

Pipeline **CPU**

Many **CPU**s nowadays have a *pipelined* approach:

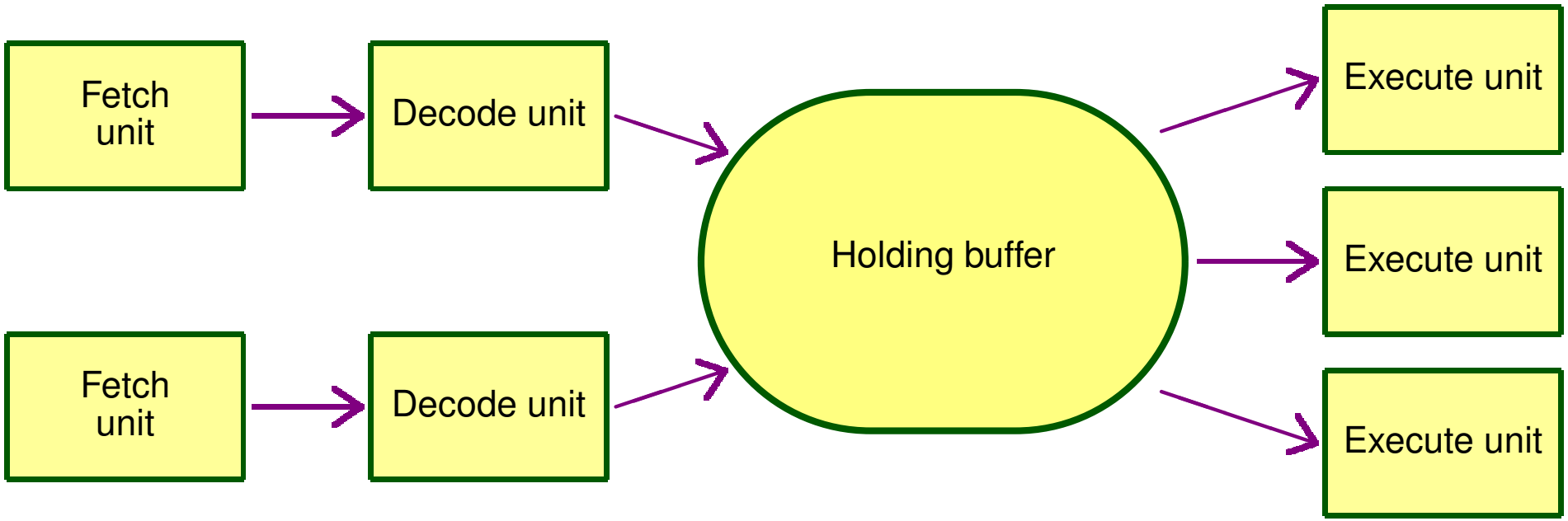


which allows for speed-up of the **CPU**:



Example: RISC, Intel Pentium, . . .

Superscalar **CPU**



Example: Intel Pentium Pro, . . .

- Multiple execution units (arithmetic, boolean, . . .).
- Instructions are executed out of order.
- Hardware checks correctness.
- **OS** and interrupt handling much more complex.

Precise Interrupt

We need an interrupt that leaves the machine in a well-defined state:

- the **PC** is saved in a known place;
- all instructions before the **PC** have fully executed;
- no instruction beyond the **PC** has been executed;
- the execution state of the instruction at the **PC** is known.

Note that:

- instructions after the **PC** may have started, but effect on registers and such need to be reversed;
- instruction at **PC** could have started, but need not;
- often, for **I/O**-interrupt, it hasn't;
- for **trap** of page fault, **PC** points to the instruction that caused the interrupt.

Imprecise Interrupt

Makes life very unpleasant for **OS** programmer.

- Large internal state is dumped unto stack.
- **OS** has to find out what has happened.
- Fast **CPU**, but slow interrupt handling.

Chip designers make some interrupts, like **I/O**, precise but others, like error traps, imprecise; not bad, because restart of process not likely.

A bit can be present that makes all interrupts precise; slows the machine down considerably because of log maintenance.

Interrupt Processing: Intel Pentium

- 256 different interrupts via interrupt description table (**IDT**)
- **IDT** stores **`EIP`** and **`CS`** for the interrupt specific handler:

  No. 0 divide error;

  No. 4 overflow error;

  No. 6 invalid instruction;

  No. 12 stack fault;

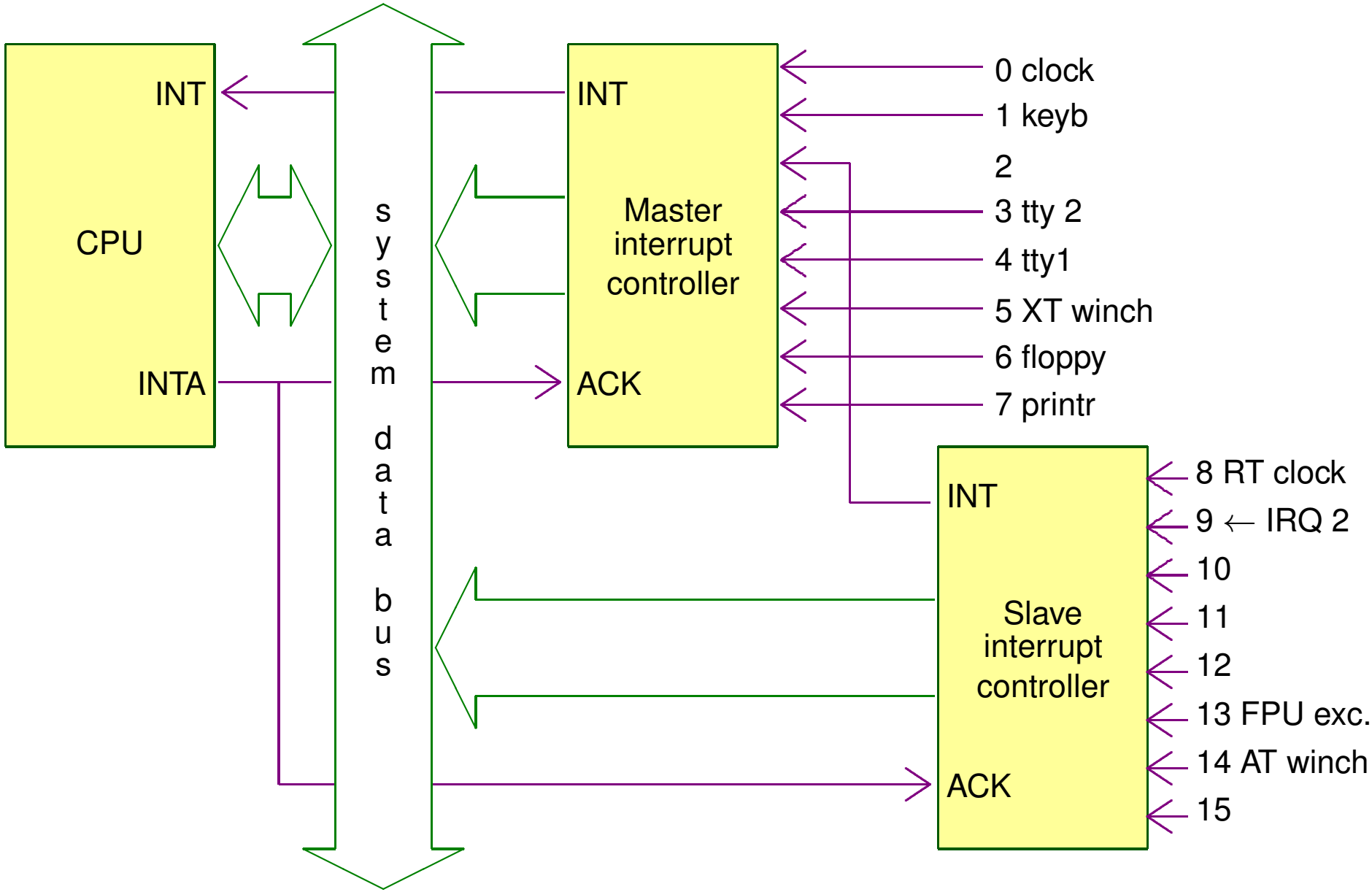
  No. 14 page fault; (*later ...*)

  No. 16 floating point error;

  No. 32-255 user defined, external interrupts (**I/O** devices).

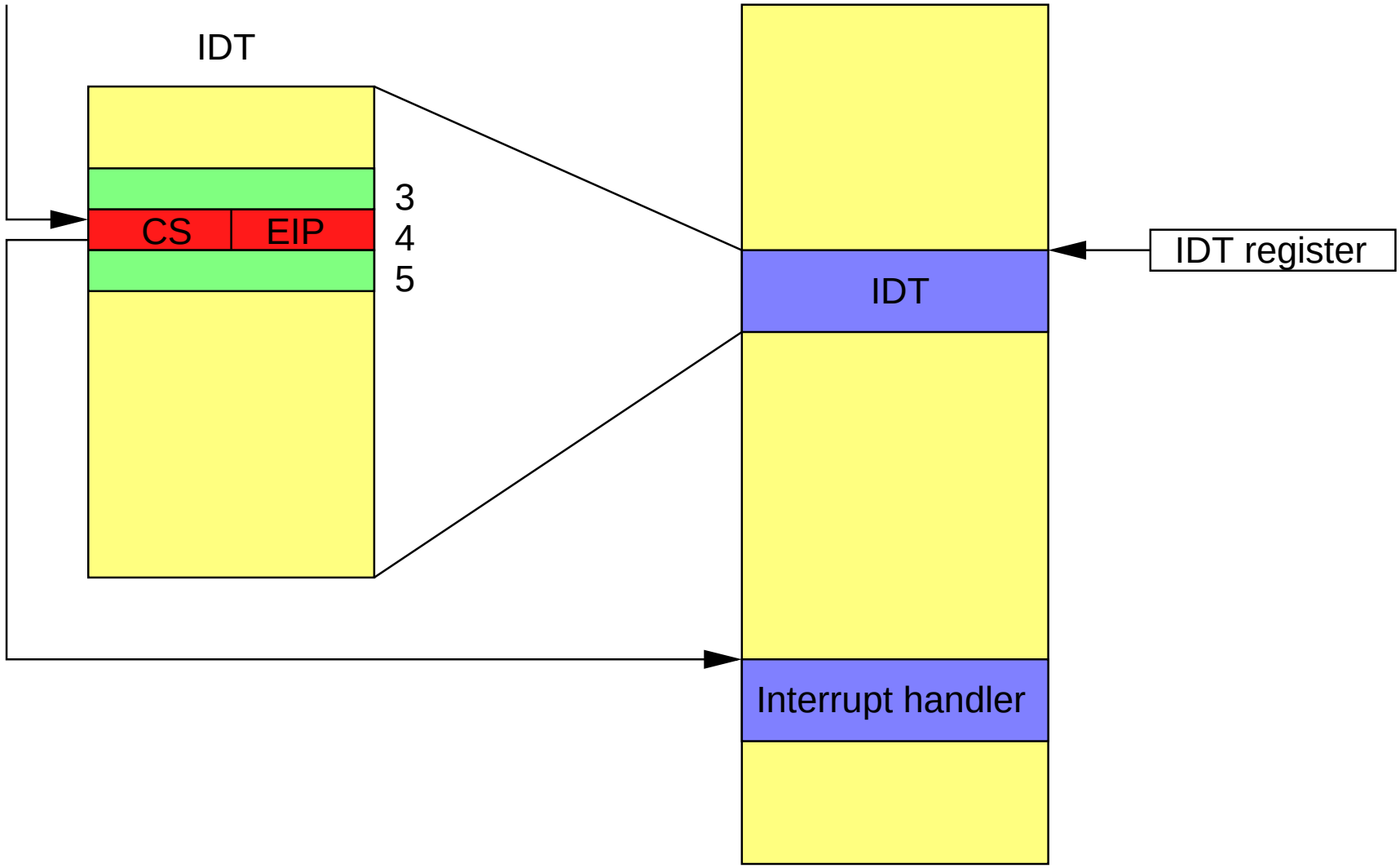
- Interrupts can be generated by **`INT#n`**.
- Return from interrupt using **`IRET`**.
- Interrupts can be enabled using **`STI`**.
- Interrupts can be disabled using **`CLI`**.

Interrupts Pentium (2)



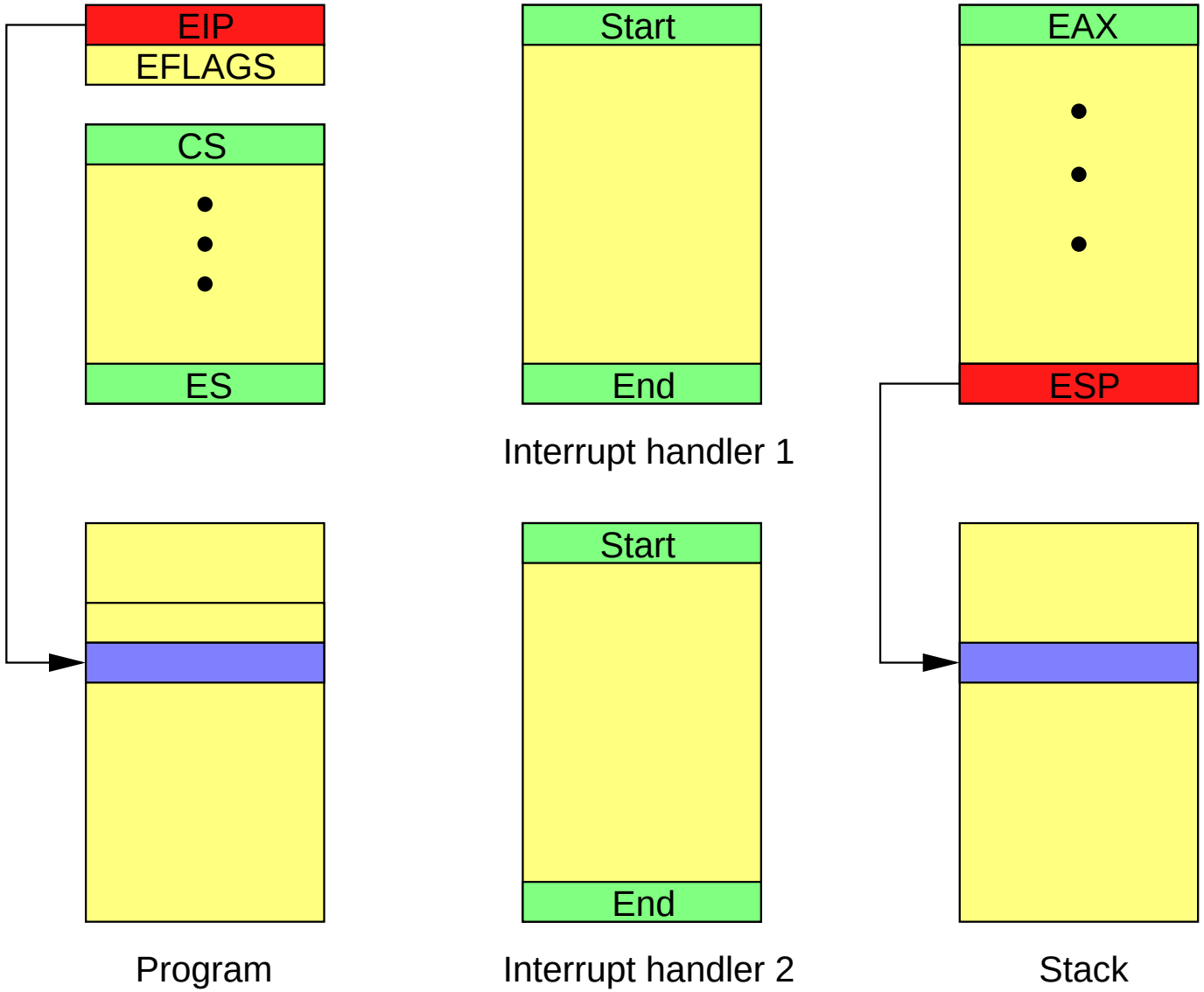
Intel Pentium (2)

interrupt no. 4



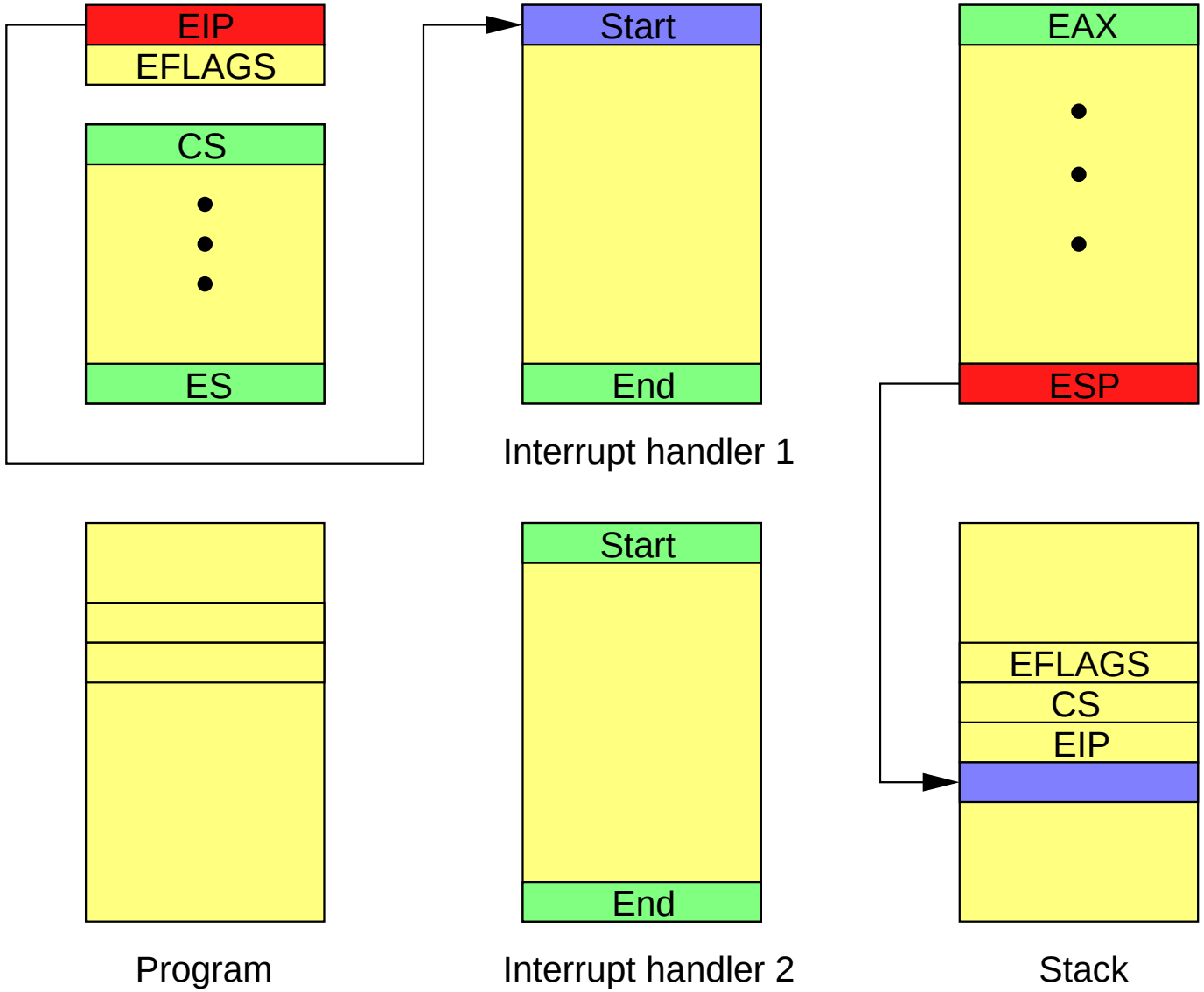
Interrupt Processing

Before interrupt:



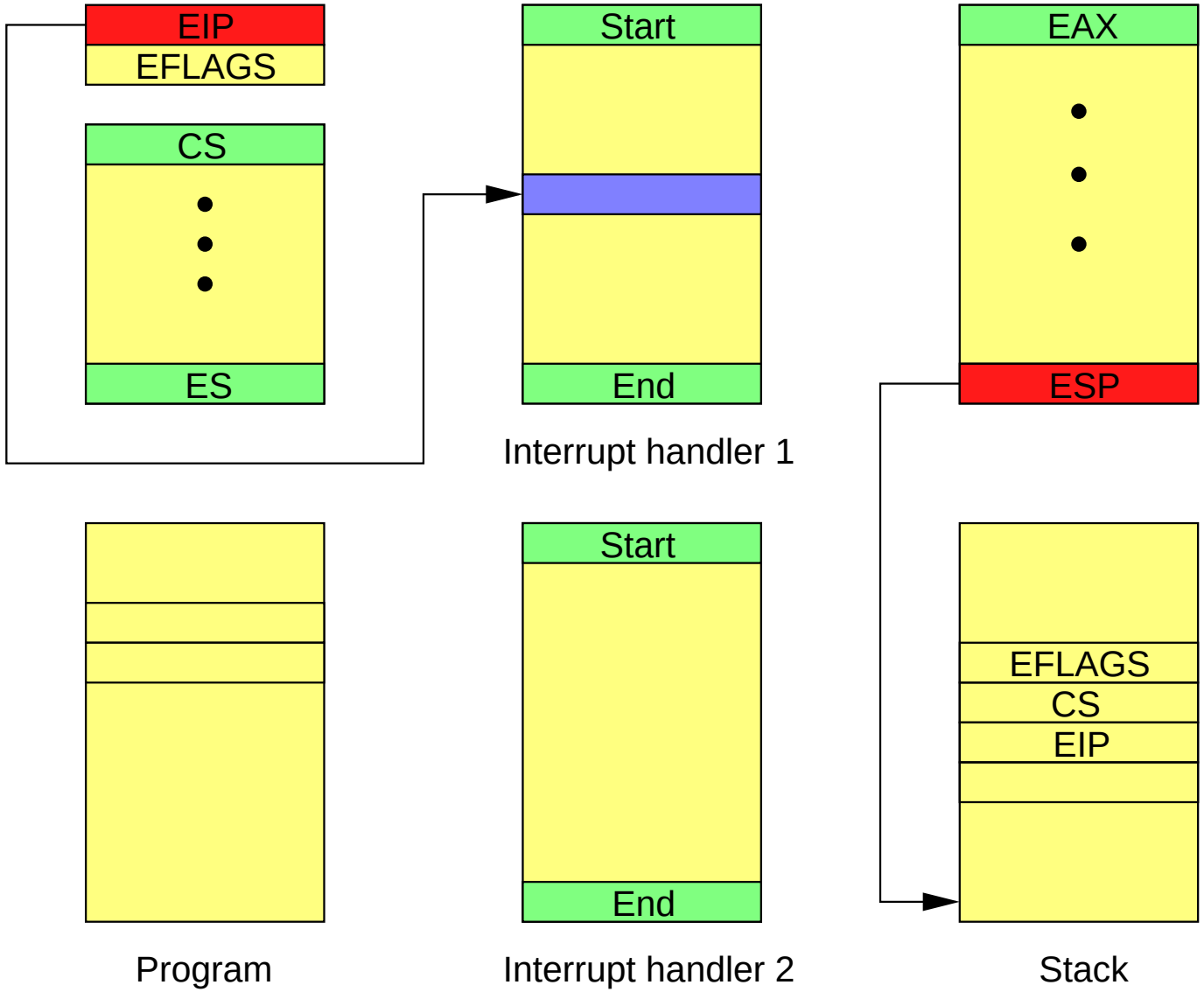
Interrupt Processing (2)

After interrupt 1 has occurred:



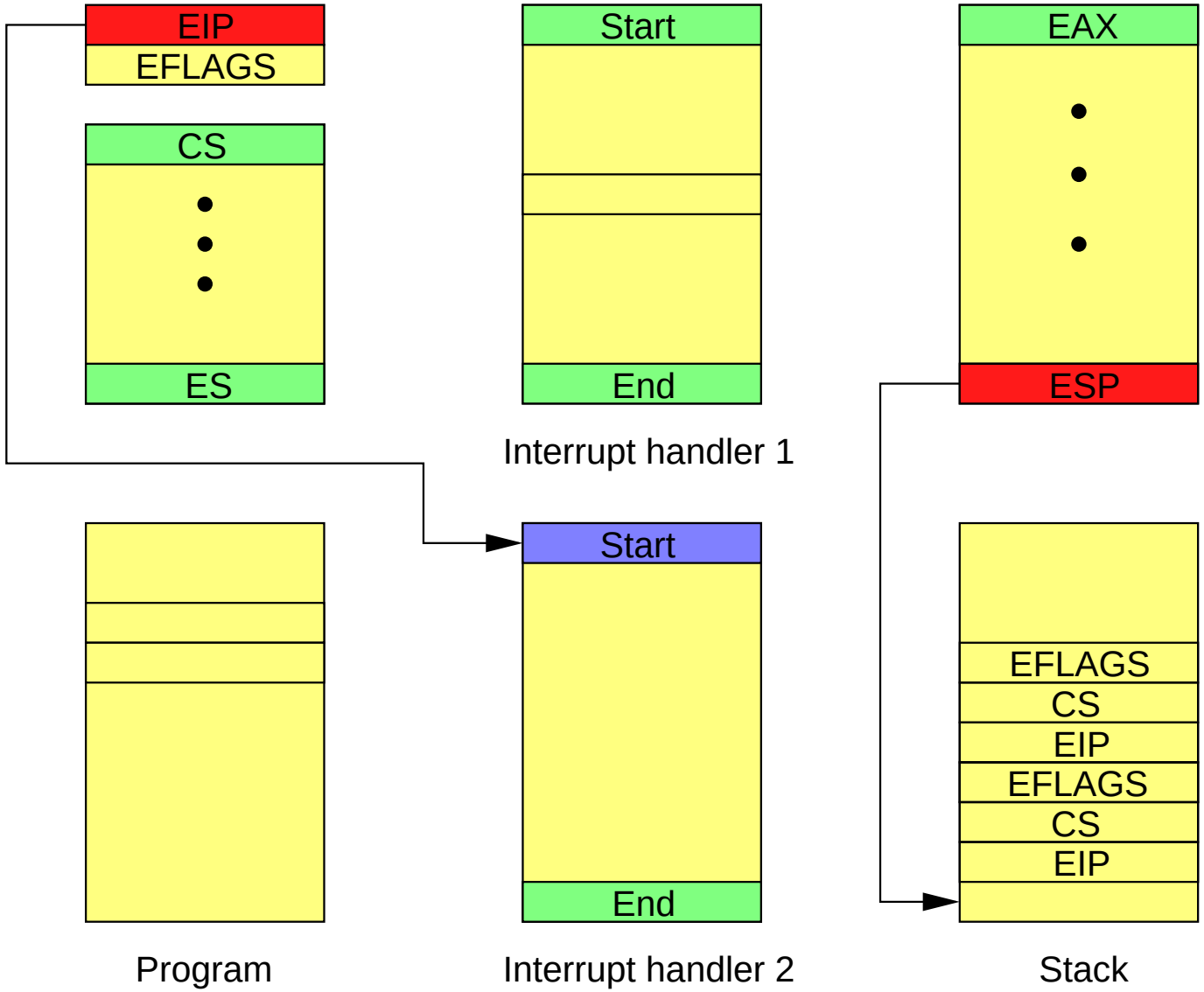
Interrupt Processing (3)

Before interrupt 2 occurs:



Interrupt Processing (4)

After interrupt 2 has occurred:



Interrupts in Minix

Device Send electrical signal to interrupt controller

Controller **1** Interrupt **CPU**.

2 Send digital id of interrupting device.

Kernel **1** Save registers.

2 Execute driver software to read **I/O** device.

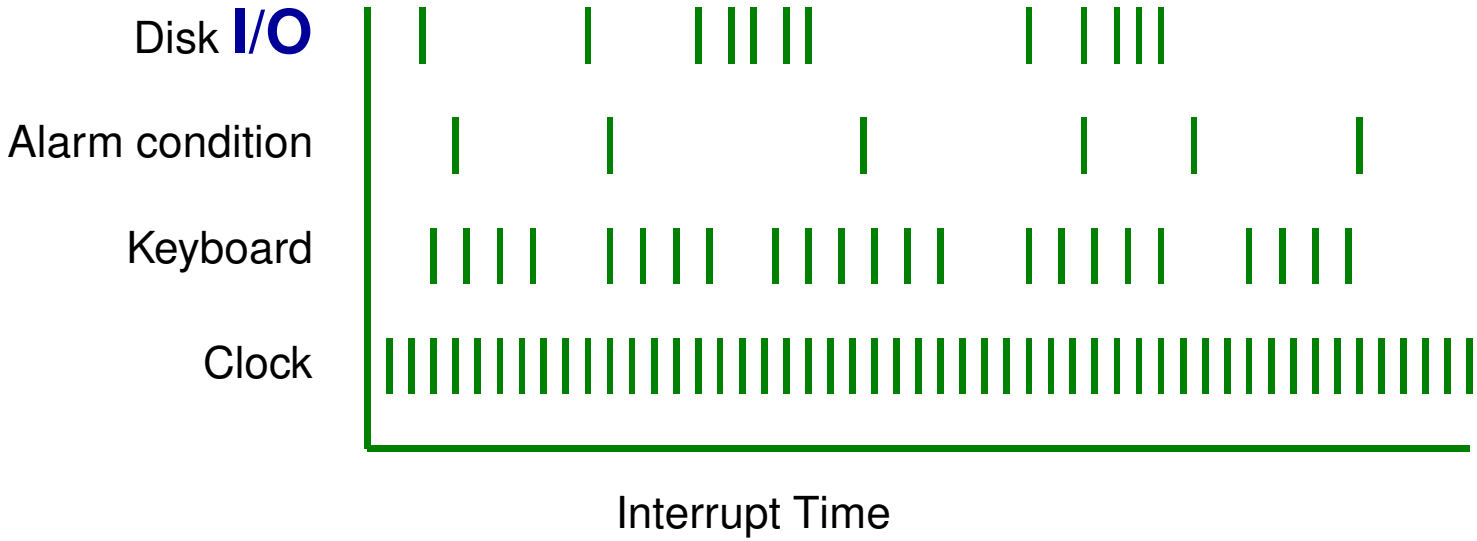
3 Send message.

4 Restart a process (not necessarily interrupted one).

Processes

Non - Determinism

Operating Systems and Real-Time systems are non-deterministic in that they must respond to events (**I/O**) which occur in an unpredictable order, and at any time



Avoid **I/O** *Delays*

An **I/O** action is normally started by sending information to a special address in the machine's memory (**DMA**); the result of the action is reported in another address.

After initiating an **I/O** action, the **CPU** can repeatedly check the reply address to see if the execution went well: but this is a *busy-wait* solution, which wastes **CPU**-cycles.

It is better to have the **CPU** start another action, and have the **I/O** device *interrupt* the **CPU** when the **I/O** action has finished, after which the **CPU** can continue the initial program.

The requires a concept of *halt* and *restart* of program execution: we need to store the *state* of a program, so that we can start it later as if it was not halted at all.

Processes

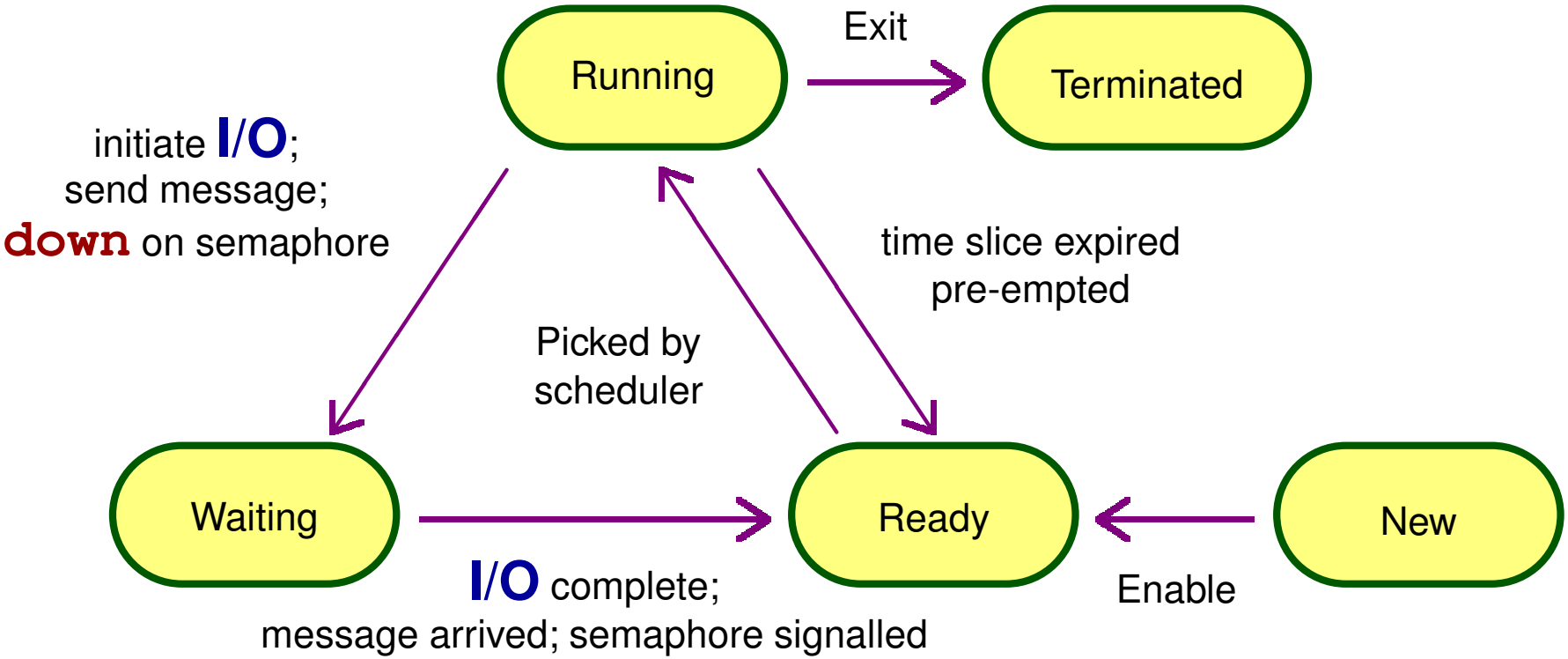
An abstraction of a running program, characterised by

- *code*,
- *input*,
- *output*, and
- *state*.

Used in situations where a program can get blocked (waiting for **I/O**, e.g.) and another program gets the **CPU** instead; the blocked process can get restarted at a later, therefore, all information concerning the process, needed to restart safely, should be stored.

For each process, all this data is stored in a *process descriptor*, or *process control block* (**PCB**, discussed later), that is kept in the *process table*

Process States



New the process is being created.
Running executing on a processor.
Ready runnable and waiting for processor.
Waiting blocked or waiting for an event.
Terminated process is being deleted.

Scheduler

Events (or interrupts) cause process switches.

- The way an **OS** switches between processes cannot be pre-determined, since the events which cause the switches are not deterministic.
- The interleaving of instructions, executed by a **CPU**, from a set of processes is non-deterministic.

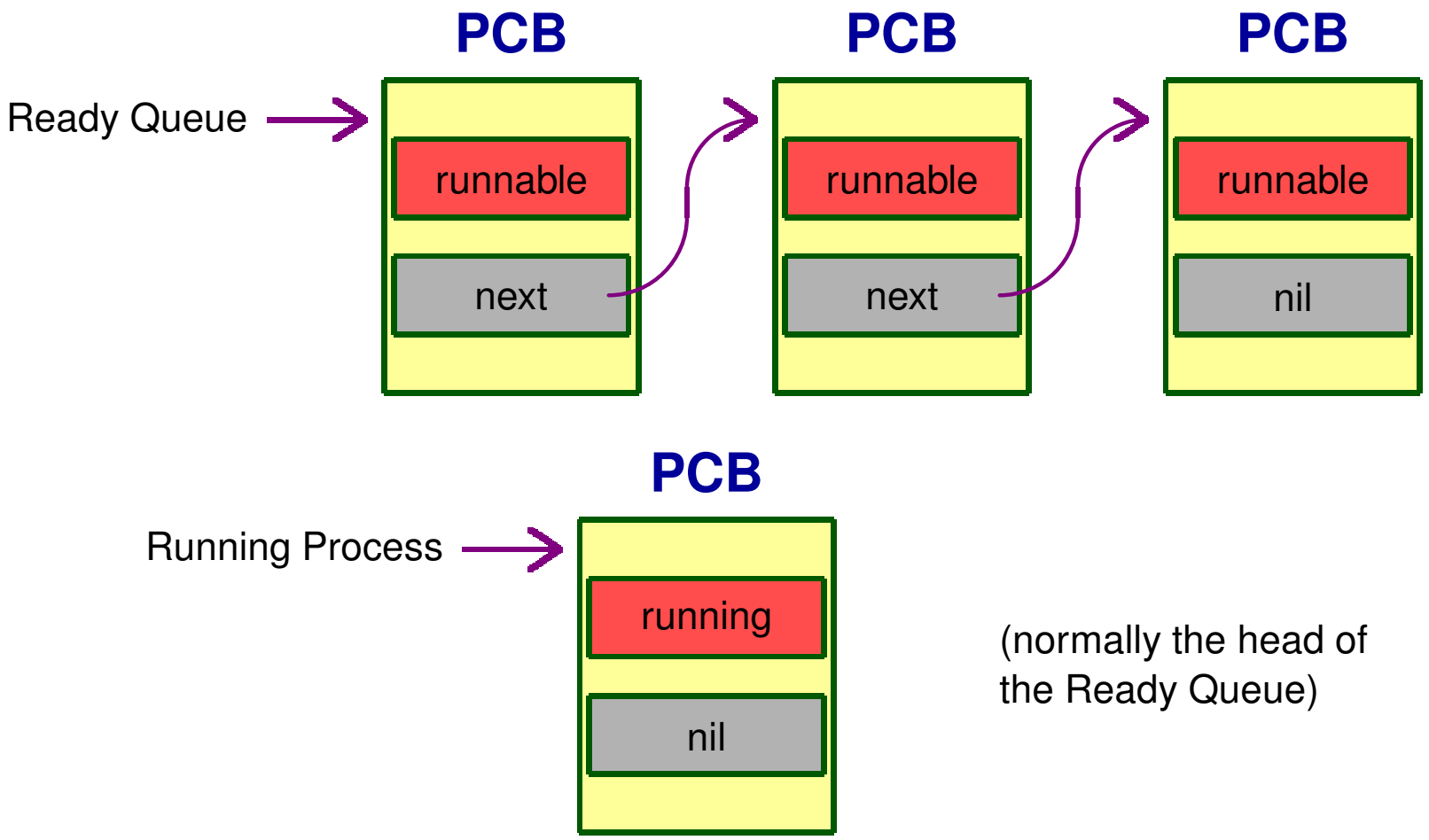
The scheduler is that part of the **OS** that decides which of the runnable processes runs first. Some criteria for *scheduling* are:

Fairness Each process gets its fair share of **CPU**.
Efficiency Keep **CPU** busy 100 percent of the time.
Response time Minimise for interactive users.
Turnaround Minimise the time users must wait for output.
Throughput Maximise the number of jobs processed.

Some of these criteria are in conflict.

Scheduler (2)

It basically changes the *state* of a process; it picks a process from the *Ready Queue* and makes it the *Running Process*



Scheduler's Task

The scheduler

- Allocates *processes* to *processors*.
- Selects highest priority *ready* process (from head of Ready Queue) and moves it to the *running* state, i.e., allows it to start executing on the processor.
- Gets invoked after every entry to the kernel.

Current process continues unless:

- Kernel call moved it into waiting state (e.g. waiting on **I/O**).
- Error trap occurred (e.g. memory protection violation).
- Time slice expired.
- A higher priority process is made ready.

Scheduling Strategies

Different strategies exist in selecting the next running process:

- Preemptive scheduling: time-slices.
- Run to completion: non-preemptive.
- Round Robin: last run process to tail of queue.
- Priority scheduling: new priority = 1 / (fraction of time-slice used).
- Multiple queues.
- Shortest job first.
- Guaranteed scheduling: allotted time = 1 / (# procs).
- Lottery scheduling.
- Real time scheduling.
- Two level scheduling: process on disk or in RAM.

Scheduler Lay-out

```
void Scheduler (){
  /* current is a pointer to the PCB of the process running when the
   * scheduler was invoked. */
  if ( current->state != running ||
       current->priority < ready_queue->priority)
  {                  /* choose new process */
    if (current->state == running) {
      current->state = ready;
      schedule(current);
            /* add current to ready_q in priority order */}
    current = ready_q;
    ready_q = ready_q->next;
  }
  load registers from PCB or stack;
  Put PC and PSW on stack;
  return from interrupt;   /* load PC and PSW from stack */
}
```

Assembler Level

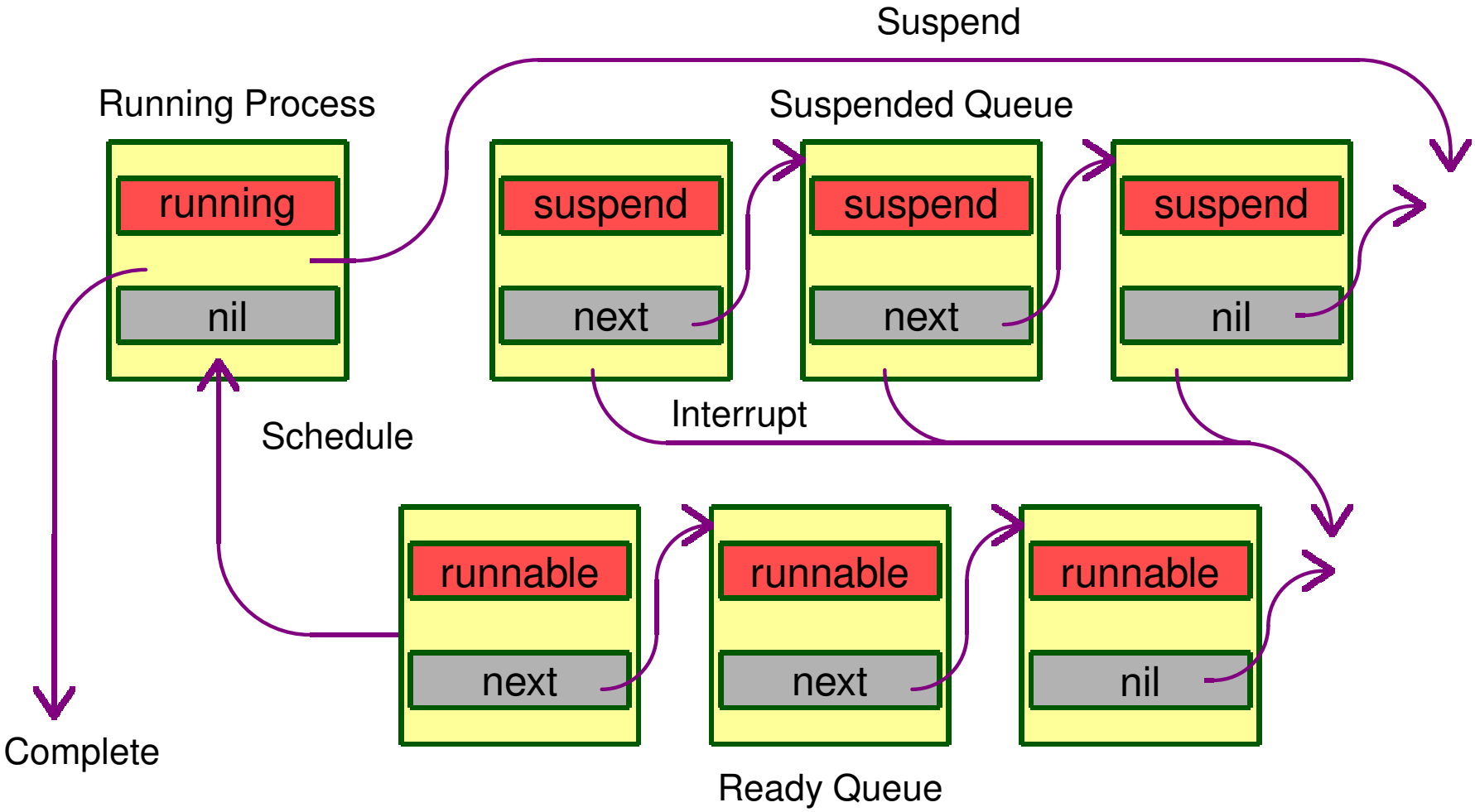
Low Level Scheduling

Natural Break / Run to completion Process runs until it completes or suspends itself by:

- Performing a call to the kernel e.g. `receive(message)` ; it then typically waits for interrupt to occur, and a message from the device handler to arrive.
- Cause error trap, and then the control switches to an exception handler.
- Process completes and leaves system; the current process is resumed after interrupt.

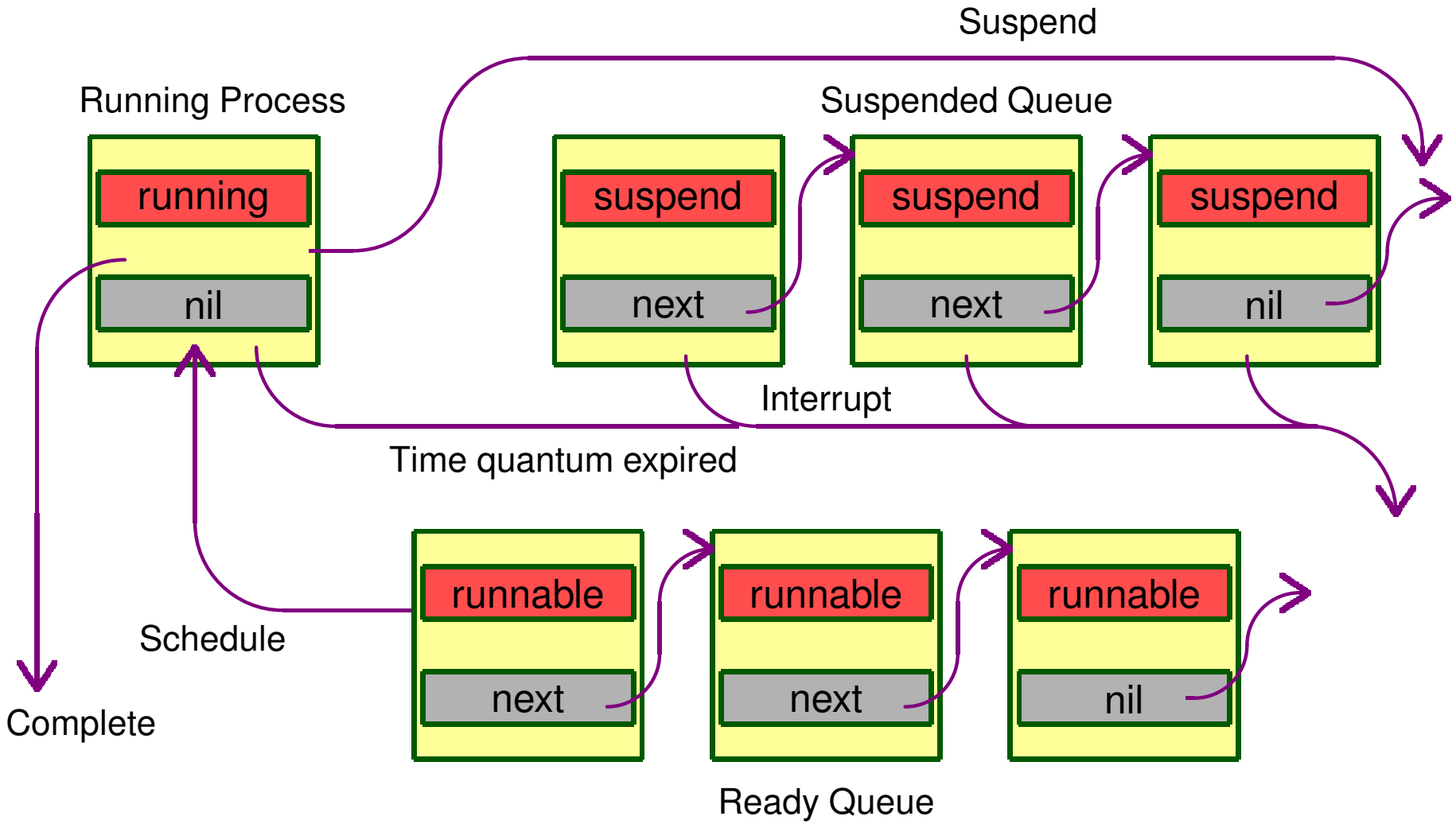
Round Robin Process Queue

Runnable process inserted at end of ready queue.



R.R.Q. with Time slice

Process runs until it suspends itself or time quantum expires.



Small quantum: more overhead; large quantum: poor response.

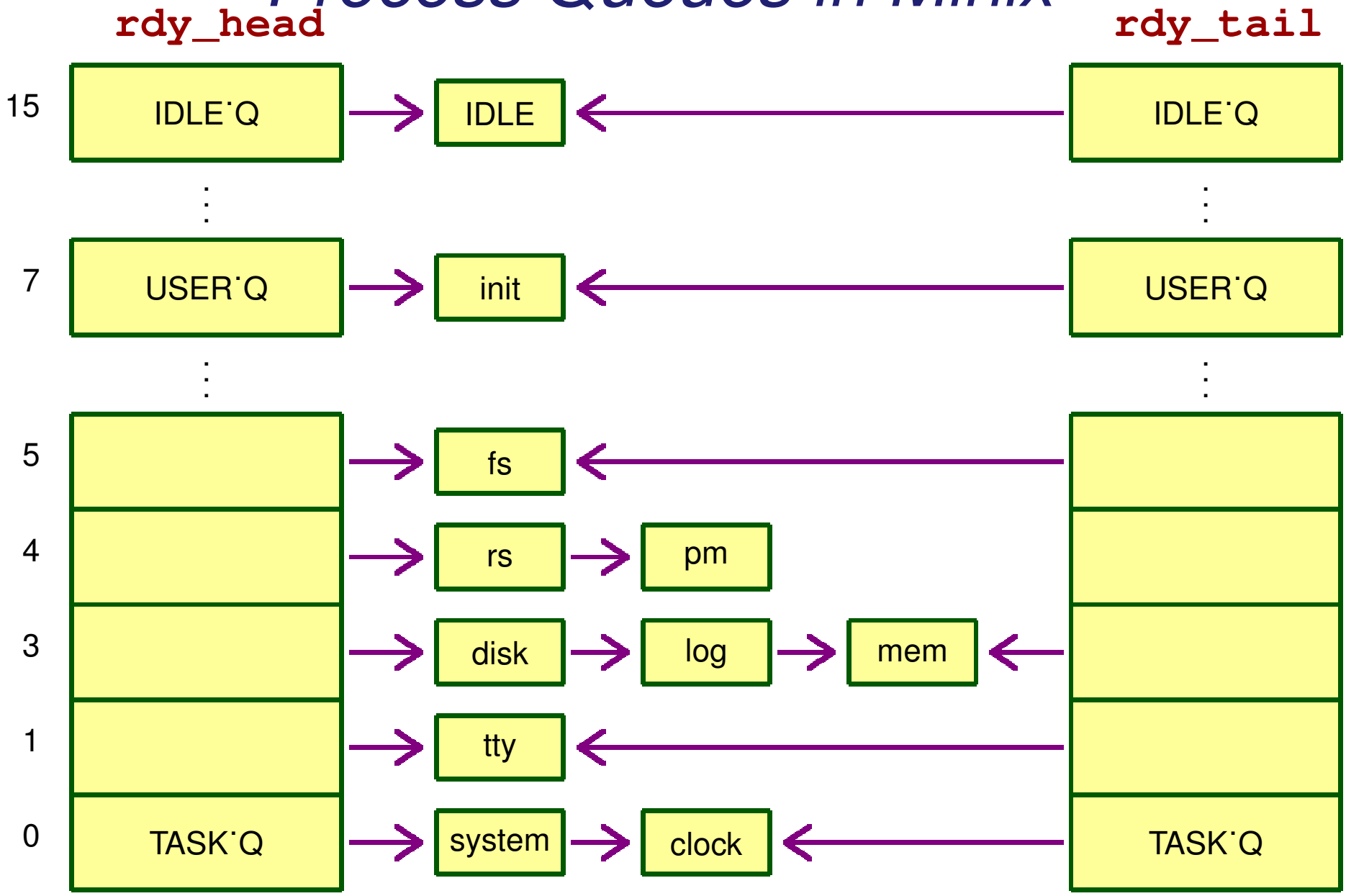
Processes, Drivers, Servers, and Tasks in Minix

- Minix uses a priority scheduler, with up to 16 levels.
- If a process or driver keeps on using up its time slot, it gets a lower priority.
- The tasks Clock and System run in Kernel space and get no **PID**.
- System processes get a longer time quantum, but it is still finite.
- User processes blocked on **I/O** get restarted with what remains of their quantum.
- The process Reincarnation Server starts up the device drivers: this has as advantage that it will be notified if one of them crashes (or even blocks), so can (kill it and) start it up again (*unique for Minix*).
- The process Process Manager deals with **`fork`**, **`exec`**, etc.
- The process Init starts up system for real. It runs the **`/etc/rc`** script, that starts drivers and services not on the boot image: all user processes are (grand) children of Init, itself a child of RS.

Minix Start-up Components

Component	Description	Loaded by
kernel	Kernel, clock, and system tasks	(in boot image)
pm	Process Manager	"
fs	File System	"
rs	(Re)starts servers and drivers	"
memory	RAM disk driver	"
log	Buffers log output	"
tty	Console and keyboard driver	"
driver	Disk (at, bios, or floppy)	"
init	parent of all user processes	"
floppy	Floppy driver (if booted from hd)	`/etc/rc`
is	Information server (for debug dumps)	"
cmos	Reads CMOS clock to set time	"
random	Random number generator	"
printer	Printer driver	"

Process Queues in Minix



Minix initial start-up queues.

Minix Scheduler

```
PRIVATE void sched(rp, queue, front)
register struct proc *rp;       /* process to be scheduled  */
int *queue;                     /* return: queue to use  */
int *front;                     /* return: front or back  */
 {
 /* This function determines the scheduling policy. It is called whenever a
  * process must be added to one of the scheduling queues to decide where to
  * insert it. As a side-effect the process' priority may be updated.  */
    static struct proc *prev_ptr = NIL_PROC;  /* previous without time  *
    int time_left = (rp->p_ticks_left > 0);  /* quantum fully consumed
    int penalty = 0;                /* change in priority  */
     /* Check whether the process has time left. Otherwise give a new quantum
      * and possibly raise the priority. Processes using multiple quantums
      * in a row get a lower priority to catch infinite loops in high priority
      * processes (system servers and drivers).  */
    if ( !time_left)  {                     /* quantum consumed ?  */
      rp->p_ticks_left = rp->p_quantum_size;  /* give new quantum  */
      if (prev_ptr == rp) penalty ++;  /* catch infinite loops  */
      else penalty --;                 /* give slow way back  */
      prev_ptr = rp;                   /* store ptr for next  */
     }
```

Minix Scheduler (2)

```
    /* Determine the new priority of this process. The bounds are determined
     * by IDLE's queue and the maximum priority of this process. Kernel task
     * and the idle process are never changed in priority.     */
    if (penalty != 0 && !iskernelp(rp))  {
      rp->p_priority += penalty;        /* update with penalty   */
      if (rp->p_priority < rp->p_max_priority)   /* check upper bound
        rp->p_priority=rp->p_max_priority;
      else if (rp->p_priority > IDLE_Q-1)   /* check lower bound   */
        rp->p_priority = IDLE_Q-1;
     }

    /* If there is time left, the process is added to the front of its queue,
     * so that it can immediately run. The queue to use simply is always the
     * process' current priority.     */
    queue = rp->p_priority;
    front = time_left;
}
```

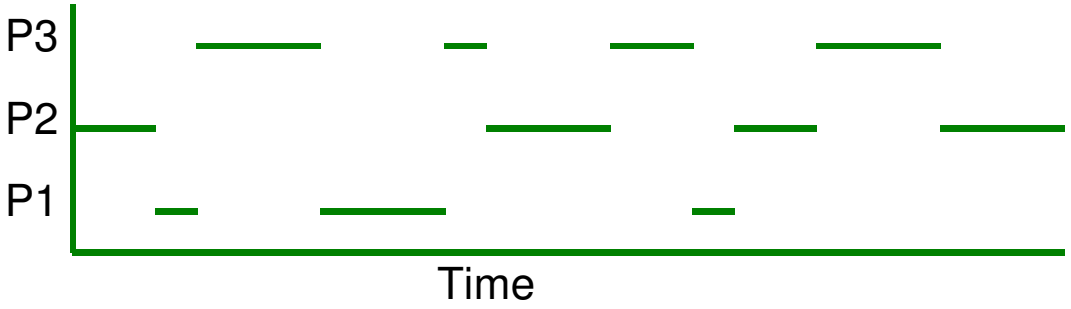
Minix Scheduler (3)

```
PRIVATE void pick_proc()
 {
 /*  Decide who to run now. A new process is selected by setting 'next_ptr'.
 *  When a billable process is selected, record it in 'bill_ptr', so that the
 *  clock task can tell who to bill for system time.      */
  register struct proc *rp;                  /*  process to run   */
  int q;                                     /*  iterate over queues  */
   /*  Check each of the scheduling queues for ready processes. The number of
    *  queues is defined in proc.h, and priorities are set in the task table.
    *  The lowest queue contains IDLE, which is always ready.      */
  for (q=0; q < NR_SCHED_QUEUES; q++)   {
    if ( (rp = rdy_head[q]) != NIL_PROC)   {
      next_ptr = rp;                         /*  run process 'rp' next   */
      if (priv(rp)->s_flags & BILLABLE)
        bill_ptr = rp;                       /*  bill for system time   */
      return;
     }
    }
 }
```

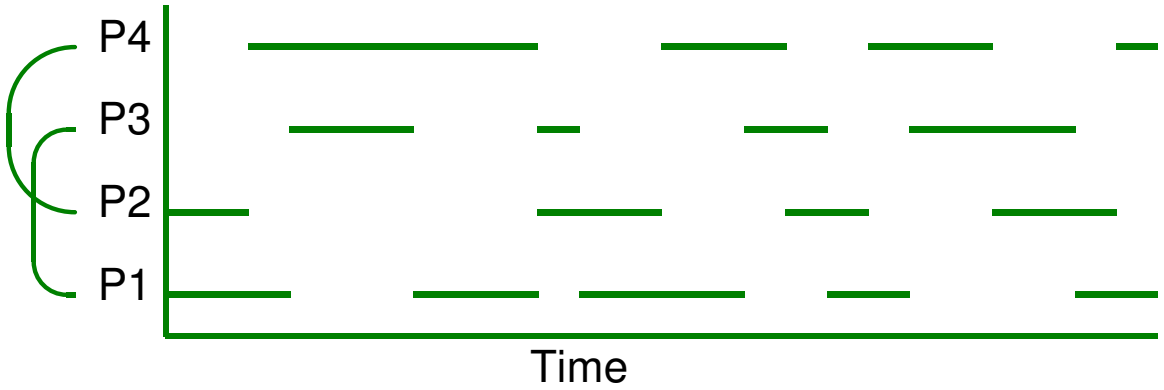
Concurrent Processes

Activation of one or more processes at the same time.

Apparent Concurrency A single hardware processor which is switched between processes by interleaving; this gives the illusion of concurrent execution. Also called *pseudo concurrency*.



Real Concurrency Multiple hardware processors; usually less processors than processes



Concurrency Levels

In general, concurrency can be provided at 3 levels:

Operating System The **OS** can support multi-programming like for example, in Unix, Mach, OSX, Windows NT. All these systems use the concept of *processes* to model multi-programming.

High Level language tasks Several high level programming languages exist that allow explicit concurrency. They are typically used for implementing parallel applications, real-time systems and **OS**. Examples: Ada, Regis, Monitors package in Modula 2.

Threads Those are a kind of lightweight process, and can exist within a context with full-scale processes. It is a unit of **CPU** utilisation, and is identified by the `PC`, registers, and stack. Protection is not supplied by the kernel and threads may share code and data with other threads in process. They are light, because the switching overhead is minimal.

Shared vs Non-shared

Shared Data Used in multiprocessor systems that have shared memory. Access to the shared data is controlled by synchronisation primitives (like, for example, *semaphores*, *monitors*), and processes communicate through the shared data.

Non-shared Data Used in distributed systems that are, for example, interconnected by networks. The communication primitives deal with synchronisation, and communication is established through message passing or remote procedure call.

Expressing Concurrent Execution

The overall design for a system with processes can be:

Static The number of processes is fixed and known at compile time.

Dynamic Processes can be created at any time, and their number is determined by a program.

Nested Processes can be defined within processes.

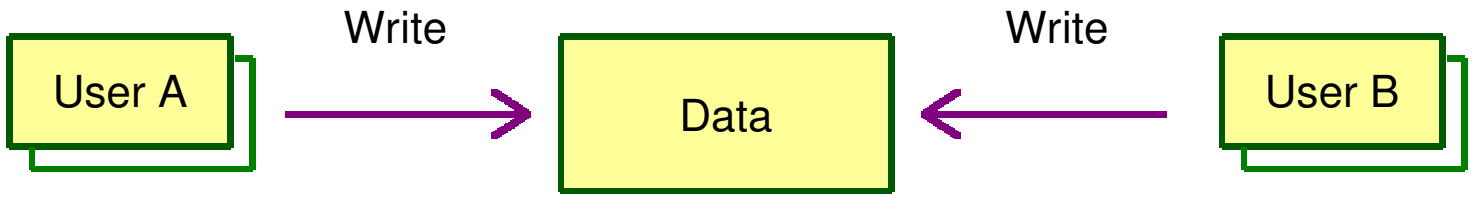
Flat Processes are defined at outermost level of program; each program is at execution time exactly one process.

Coarse grained This expresses that processes are on the average large, and are relatively few.

Fine Grained This expresses that processes are on the average light weight, and many simple processes can exist that contain only a few statements.

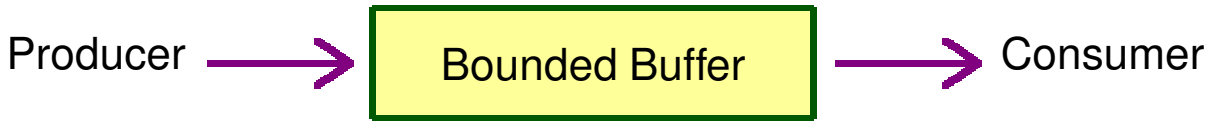
Process Interactions

1 Shared Objects.



Mutual exclusion in critical section.

2 Synchronisation of Action. Same time or defined order.



3 Exchange of Information (printing / mail)

- mutual exclusion;
- shared memory;
- distribution.

To safely use shared objects, synchronisation is necessary.

Concurrent Programming

A combination of notations, language constructs, system facilities and techniques for:

- expressing potential parallelism
- solving the resulting synchronisation and communication problems.

The techniques are independent of whether concurrency is real (supported by parallel hardware) or apparent.

It enhances expressiveness and elegance of solutions to practical problems, and for efficiency:

- Provides programming abstractions for dealing with ”real world” parallelism as occurs for:
  - Operating Systems;
  - Real-Time Systems.
- Leads to efficient implementations through interleaving on uni-processors and real parallel execution on multi-processors.

Concurrency Issues

The main issues in managing concurrency safely are:

1. How to express correctly that two programs run concurrently.
2. How to make sure that processes synchronise their actions in the right way (perhaps by formulating constraints on ordering of events).
3. How to let processes communicate (use shared data or messages).

Synchronisation and communication are interdependent.

Synchronisation

Mutual Exclusion

- The term *critical section* of a process refers to a section of code in which the process may change a resource (data such as variables, tables or files) which it shares with other processes. A synchronisation mechanism is required at the entry and exit of the critical section to ensure the coherency, consistency and integrity of shared resources.
- The term *mutual exclusion* refers to a collection of techniques for sharing resources so that the use of those resources among different processes does not lead to conflicts or unwanted interactions. Mutual exclusion ensures that if a process enters in (is in) its critical section, no other process can be in its critical section. To achieve mutual exclusion, each process must request permission to enter its critical section.

Requirements for Mutual Exclusion

- Given a set of processes with critical sections for the same resource, only one process at a time is allowed to enter its critical section.
- A process that halts in its critical section must do so without interfering with other processes.
- When no process is inside its critical section, any process requesting permission to enter its critical section must be allowed to enter its critical section without delay.
- No process requiring access to its critical can be delayed infinitely (no deadlock or starvation).
- No assumptions are made about relative speed of processes.
- A process may remain inside its critical section only for finite time.

Synchronisation Primitives

A *critical region* is a part of a program where it accesses a shared resource. The safeness criterion is that only *one* process is allowed entry to *its* critical region at a time.

To avoid race conditions, synchronisation primitives between processes are necessary:

- Locks.
- Guard variables.
- Semaphores.
- Messages.

The `tsl` *Instruction*

Binary indicator controls access to shared resource, **`lock`**.

`lock = 1` resource is unavailable (locked).
`lock = 0` resource is free (lock open).

The '**`test and set lock`**' instruction **`tsl`** does (indivisible).

- Stores the value from **`lock`** into a register.
- Writes **`1`** in **`lock`**.

We can program entering and leaving a critical section as follows:

```
enter_cr:
   tsl reg,lock     | copy lock to reg and set lock to 1
   cmp reg,#0       | was lock 0?
   jne enter_cr     | if it was not 0, lock was set, so loop
   ret              | return to caller

leave_cr:
   move lock,#0     | store a 0 in lock
   ret              | return to caller
```

Guard Variable

```
int guard = 0;

process P-i {
  for (;;) {
      entry protocol
    while (guard == 1)  {}
                    /* wait if other process is in CS */
    guard = 1;
      critical section
    guard = 0;
      exit protocol
  }
}
```

Busy-wait Waste of **CPU** time.

Race condition When `P-1` sets `guard` to `1` and then `P-2` immediately does the same, both processes will go into their critical section.

Two Guard Variables

```
process P-i {
  global int enter-i = FALSE;

  for (;;) {
      entry protocol
    enter-i = TRUE;          /* announce intent to enter */
    while (enter-(3-i)) {}/* wait if other process is in CS */
      critical section
    enter-i = FALSE;
      exit protocol
  }
}
```

Busy-wait

Deadlock When `P-1` sets `enter-1` to `TRUE` and then `P-2` immediately sets `enter-2` to `TRUE`, both processes will remain in a loop.

Strict Alternation

```
int turn = 1;    /* Process 1 can go first */

process P-i {
  for (;;) {
      entry protocol
    while (turn != i) {}
                   /* wait if other process is in CS */
      critical section
    turn = 3-i;
      exit protocol
  }
}
```

Busy-Wait

Synchronisation What if `P-1` is very slow and `P-2` very fast?

Peterson's Solution

```
int turn = 1;          /* Process 1 can go first */

process P-i {
  for (;;) {
    global int enter-i = FALSE;
      entry protocol
    enter-i = TRUE; /* announce intent to enter */
    turn = 3-i;     /* set priority to P2 */
    while (enter-(3-i) & turn == 3-i) {}
        /* wait if other process is in CS */
      critical section
    enter-i = FALSE;
      exit protocol
  }
}
```

This idea breaks the possible deadlock. Concurrent assignment to `turn` results in one of the processes having priority.

Semaphores

- Idea: processes will cooperate by means of *signals*:
  - a process will stop waiting for a specific signal;
  - a process will continue if it has received a specific signal.
- Semaphores are *special variables*, accessible via the following public, atomic operations:

  `down(s)` receive a signal via semaphore **`s`**;

  `up(s)` transmit a signal via semaphore **`s`**;

  `init(s,i)` initialise semaphore **`s`** with value **`i`**.
- Semaphores have two private components:
  - a counter;
  - a list of processes – unless implemented by busy-waiting.
- Concept of semaphores is due to Dijkstra: Dijkstra, E. *Cooperating Sequential Processes*, 1965.

Semaphores (2)

Non-negative integer; updated by operations **down** and **up**.

```
down(s) ::= if (s>0)
               s = s-1;
            else suspend process on s

up(s)   ::= if   ( processes waiting on s)
               resume one of these processes
            else s = s + 1;
```

Binary Semaphore: Only takes values 0 and 1 (a lock!):

```
/* s == 1 */  up(s)  /* s == 1 unchanged */
```

General Semaphore: Takes values 0, 1, 2 . . . n. Can be used as a counter, e.g. to count the number of items in a buffer. The initial value is 1 for mutual exclusion, i.e. only 1 process can be in its critical section at a time.

Semaphores (3)

- The initial value of a semaphore counter indicates how many processes can access shared data at the same time.
- The current value of a semaphore counter indicates:

  `s.count` $\geq$ **`0:`** how many processes can execute **`down(s)`** without being blocked;

  `s.count` $<$ **`0:`** how many blocked processes are in **`s.queue`**.
- The queue of blocked processes is usually implemented as a FIFO queue.
- Advantages:
  - can be used to implement synchronisation between a number of processes;
  - can be used to avoid busy-waiting of processes.

Mutual Exclusion

```
process A
   ...
   down(s)
      critical section
   up(s)
end
```

```
process B
   ...
   down(s)
      critical section
   up(s)
end
```

```
main() {
   var s:Semaphore
   ...
   init(s, 1) /* initialise semaphore */
   ...
      start processes A and B in random order
   ...
}
```

Synchronisation

`Process A` must execute its critical section before **`process B`** can execute its critical section

```
process A
   ...
      critical section
   up(s)
end
```

```
process B
   ...
   down(s)
      critical section
end
```

```
var s:Semaphore
...
init(s, 0) /* initialise semaphore */
...
   start Process A and B in random order
...
```

Deadlock

Deadlock and starvation of processes are still possible:

```
process A
   ...
   down(s1)
   ...
   down(s2)
   ...
end
```

```
process B
   ...
   down(s2)
   ...
   down(s1)
   ...
end
```

```
var s1,s2:Semaphore
...
   start Process A and B in random order
...
```

Producer / Consumer

Producer/Consumer problem: Client/server applications

- Producer constraints: items can only be deposited in buffer if
  - there is space in buffer;
  - mutual exclusion is ensured.
- Consumer constraints: items can only be fetched from buffer if
  - there are items in buffer;
  - mutual exclusion is ensured.
- Buffer constraints:
  - buffer can hold between *0* and *N* items.

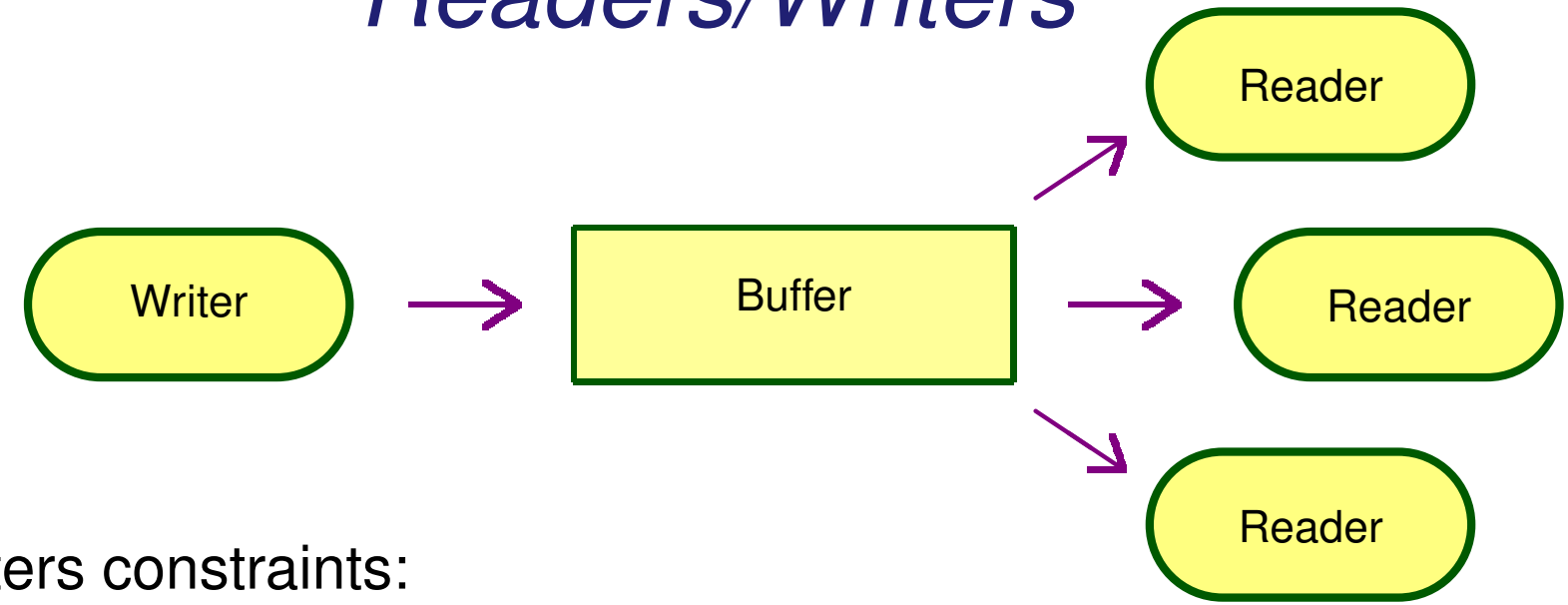
Producer / Consumer (2)

```
var item, space, mutex:  Semaphore
init(item, 0) /* Semaphore to ensure buffer is not empty */
init(space, N) /* Semaphore to ensure buffer is not full */
init(mutex, 1) /* Semaphore to ensure mutual exclusion */
```

```
procedure producer()
  loop
    produce item
    down(space)
    down(mutex)
    deposit item
    up(mutex)
    up(item)
  end loop
end procedure
```

```
procedure consumer()
  loop
    down(item)
    down(mutex)
    fetch item
    up(mutex)
    up(space)
    consume item
  end loop
end procedure
```

Readers/Writers



- Writers constraints:
  - items can only be written if no other process is writing;
  - items can only be written if no other process is reading.
- Readers constraints:
  - items can only be read if no other process is writing;
  - items can be read if there are other processes reading.
- Buffer constraints:
  - buffer can hold an arbitrary number of items.

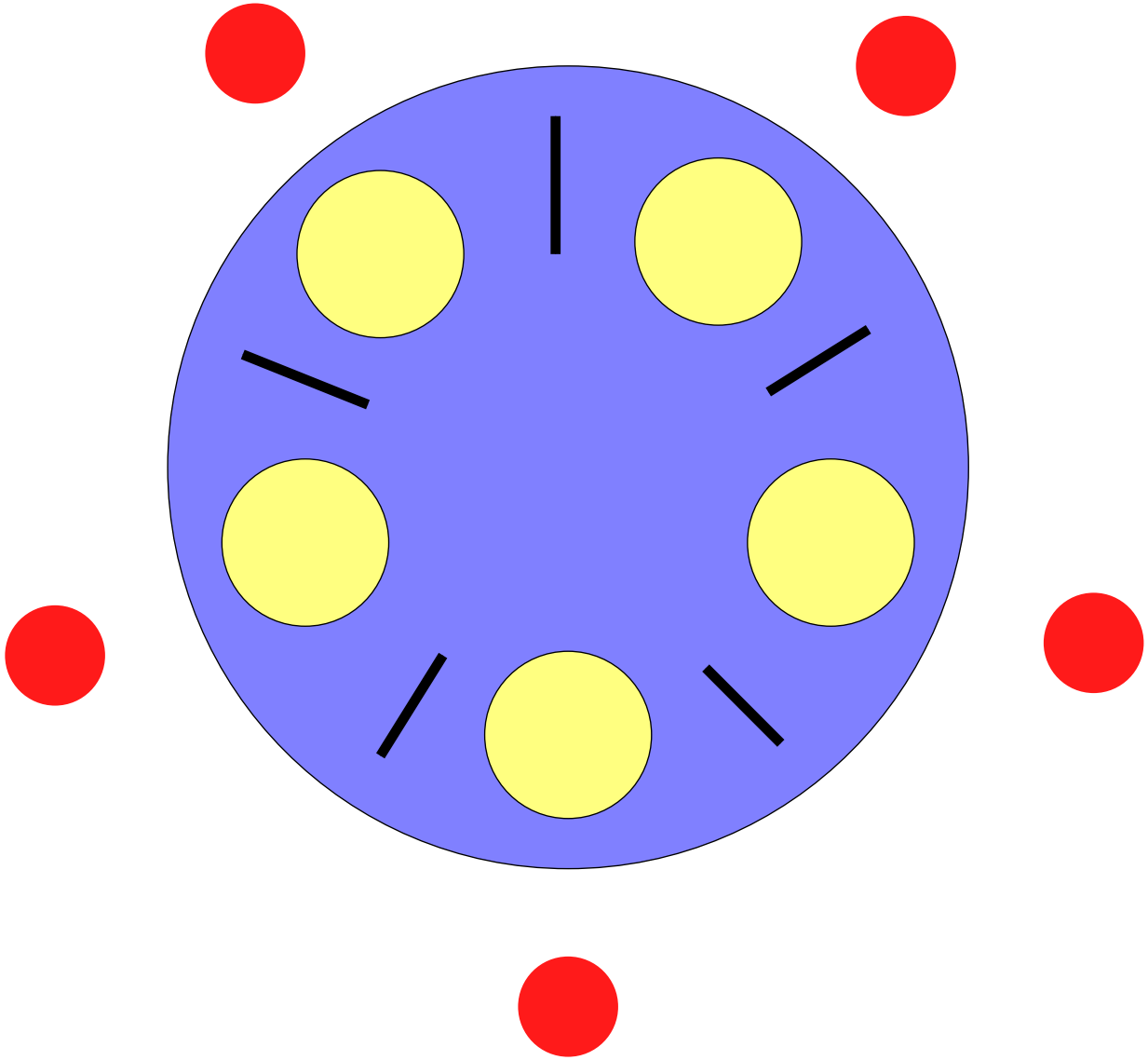
Readers/Writers (3)

```
semaphore mutex, wrt;
int read_cnt = 0;
init(mutex, 1);
init(wrt, 1);


process writer()
  loop
    produce item
    down(wrt);
    write item
    up(wrt);
  end loop
end producer
```

```
process reader()
  loop
    down(mutex)
    read_cnt += 1;
    if (read_cnt == 1)
      down(wrt);
    up(mutex);
    read item
    down(mutex);
    read_cnt -= 1
    if (read_cnt == 0)
      up(wrt);
    up(mutex);
    consume item
  end loop
end producer
```

Dining Philosophers



Dining Philosophers (2)

```
var chopstick:  array [0..4] of Semaphore
procedure philosopher(i:int)
  loop
    down(chopstick[i])
    down(chopstick[i+1 mod 5])
    eat
    up(chopstick[i])
    up(chopstick[i+1 mod 5])
    think
  end loop
end philosopher
```

Comments – Semaphores are difficult to use, understand and verify (rather unstructured).

- Synchronisation dispersed in processes. Small mistakes are disastrous.
- No explicit textual association between semaphore and the data item to which it is synchronising access.

Messages

Message Passing

A mechanism for Inter-Process Communication (IPC), transferring information between processes through sending and receiving information using a fixed interface, packaged in *messages*.

Suitable for networked or distributed systems.

Message passing is controlled by the **OS** kernel, using system functionality that is not is available to user processes; they can only **`send`** and **`receive`** messages, and are oblivious to *how* messages are passed, or that they are put on hold, and do not need to acknowledge receipt.

The three main issues are:

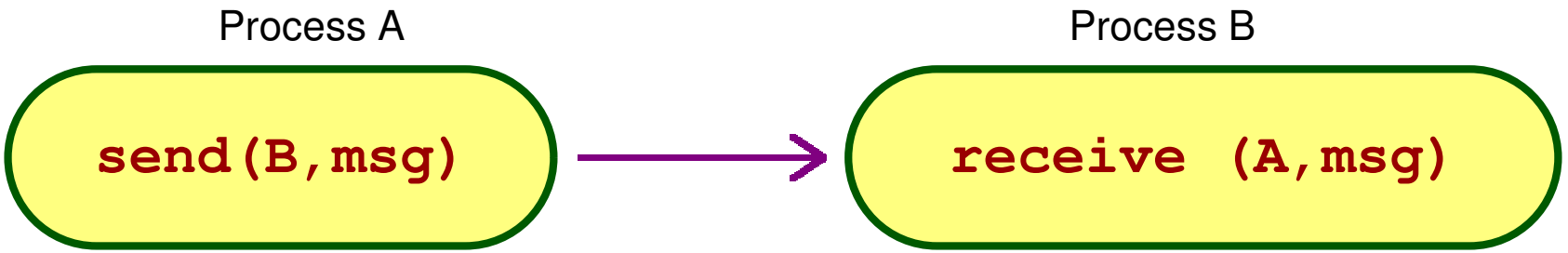
- How can one process pass information to another.
- Make sure that process do not get in each other's way.
- Proper sequencing when dependencies are present.

Simple Message Primitives

We have two main procedures:

`send(P,msg)` (executed by the **OS**) Send message **`msg`** to process **`P`**. If process **`P`** is busy, i.e. has not reached **`receive`** yet, the message is queued by the **OS**. The sending process is not *blocked* (*suspended*) by the **OS** waiting for the message to arrive, so it can continue after sending. This is called an *Asynchronous Send*.

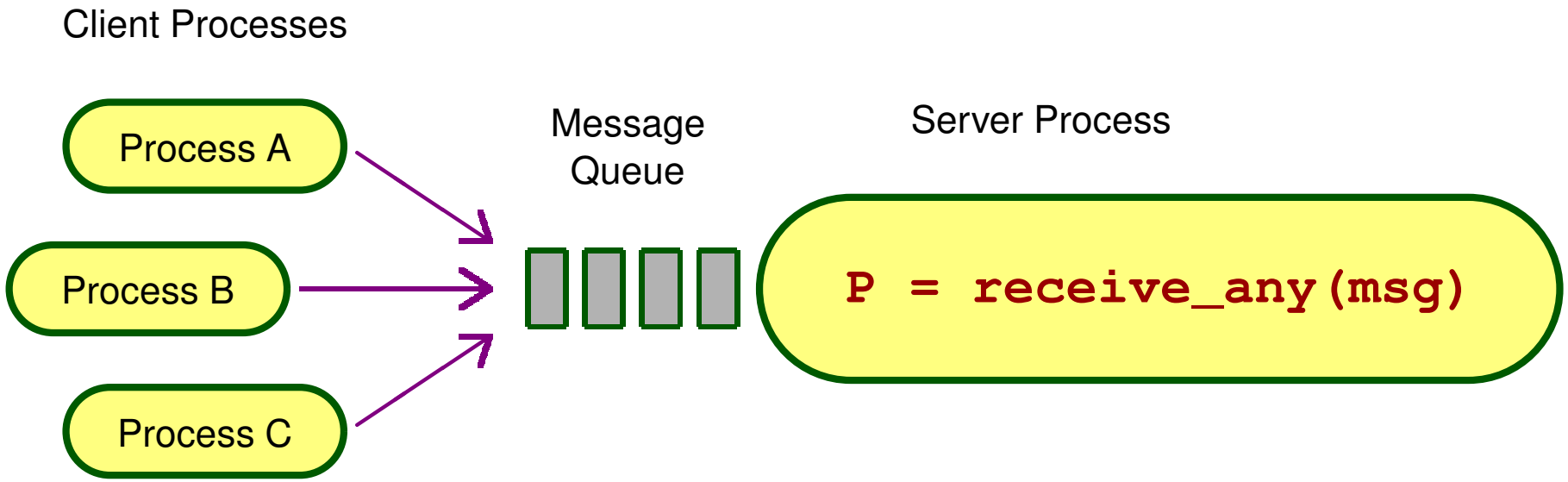
`receive(P,msg)` (executed by the **OS**) Receive message from process **`P`** into variable **`msg`**. If no message has arrived from **`P`**, i.e. there is no message in the queue, the receiving process is blocked.



Servers

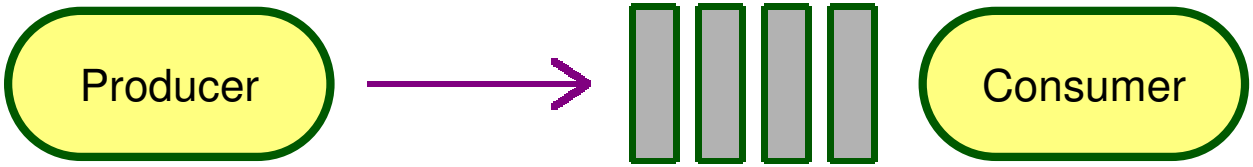
We'd like to deal with the possibility of potentially multiple senders. Since it is then in general not clear what the source of message is, this asks for a different **`receive`** primitive.

`source = receive_any(msg)` Receives a message from any process into variable **`msg`**. The name of the sending process is returned in **`source`**.



Producer - Consumer

For example, we can implement the synchronisation between a producer and a consumer by:




```
process Producer (void) {
  for (;;) {              /* does not wait for items to be consumed */
    produce(item);        /* generate something to put in buffer */
    build_message(m,item); /* construct message to send */
    send(Consumer,m);     /* send item to Consumer */
} }

process Consumer (void) {
  for (;;) {  /* does not care how many items are in the queue */
    receive(Prod,m);      /* get message containing item */
    extract_item(m,item); /* extract item from message */
    consume(item);        /* do something with the item */
} }
```

Problems with Asynchronous Sends

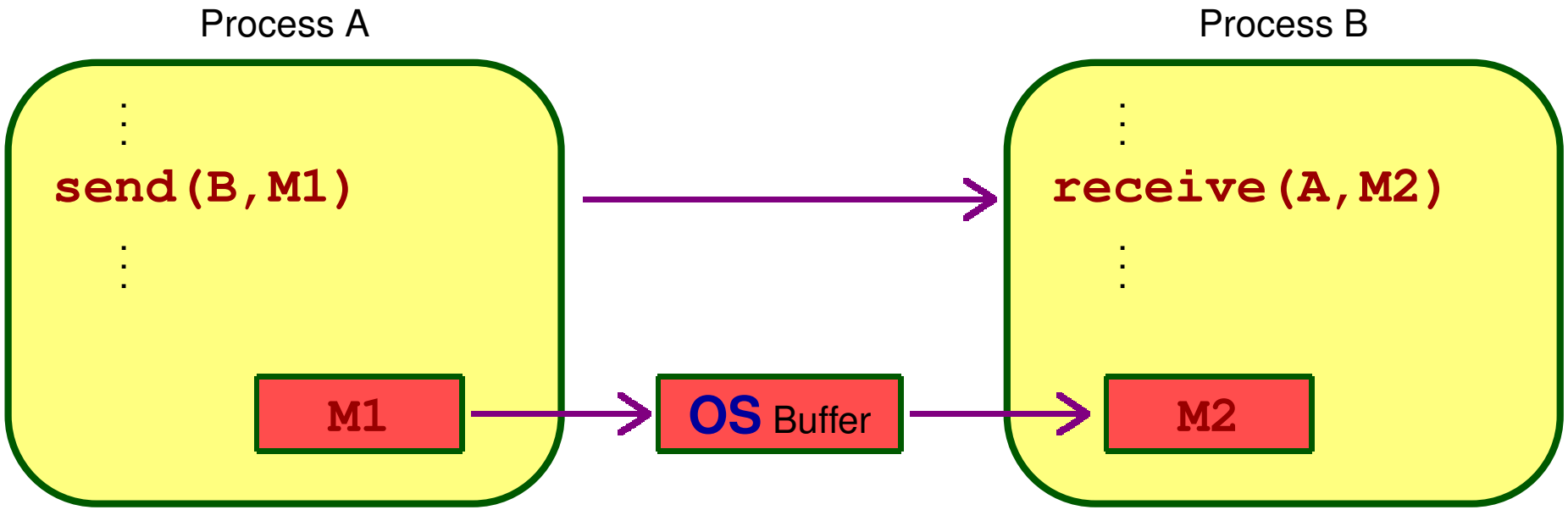
An *Asynchronous Send* increases possible parallelism between sender and receiver, but has some hazards:

- Sender continues so can access message variable after sending, but before it is received.
- If message variable was declared within high level language procedure, it can exit before message received, destroying the data.

So the **OS** must provide buffer space.

Problems with Asynchronous Sends (2)

In fact, the **OS** copies message from sender's variable into **OS** buffer as part of the implementation of **`send`**. This message is then queued from the **OS** buffer to the receiver. When receiving process executes **`receive`**, the **OS** copies the message out of the buffer into the receiver's variable and releases the **OS** buffer.

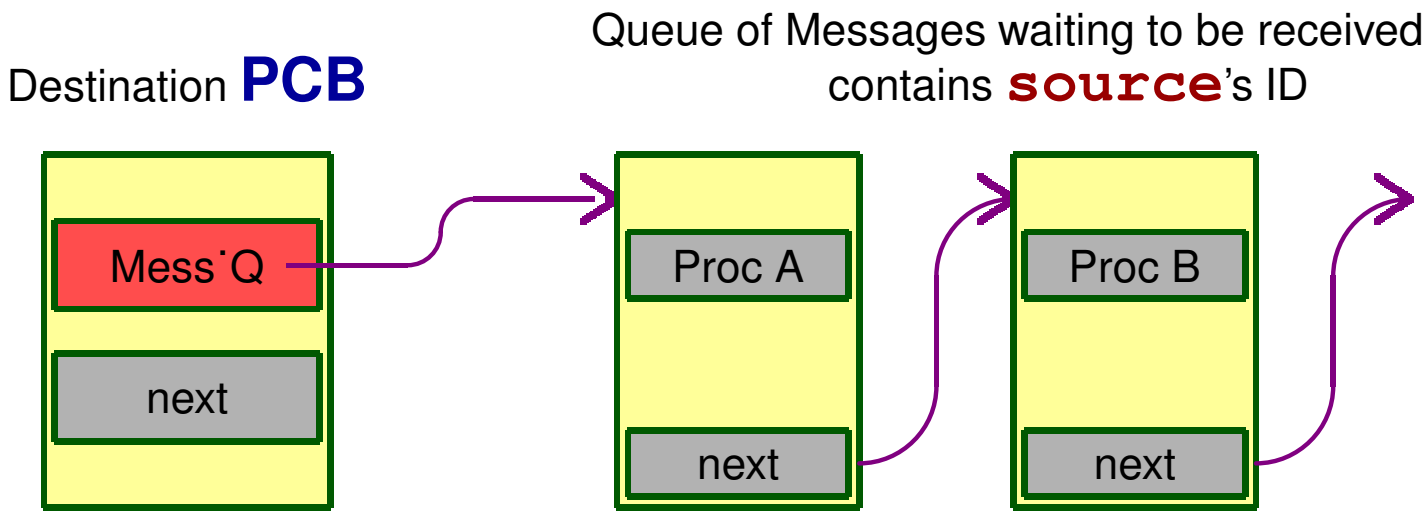


Problems with Asynchronous Sends (3)

- If the producer runs faster than the consumer, the **OS** may run out of buffers. What to do then?
  - Block Producer?
  - Abort Producer?
  - Crash System?
- Messages may be of any length, but the management of variable sized buffers in **OS** increases complexity.
- Fixed Length Messages: implemented in some **OS** to reduce **OS** buffer management complexity.

  If sender has long a message to send, it passes a pointer to variable in sender's space rather than the message itself.

Receive in Asynchronous Message Passing



receive is now implemented by:

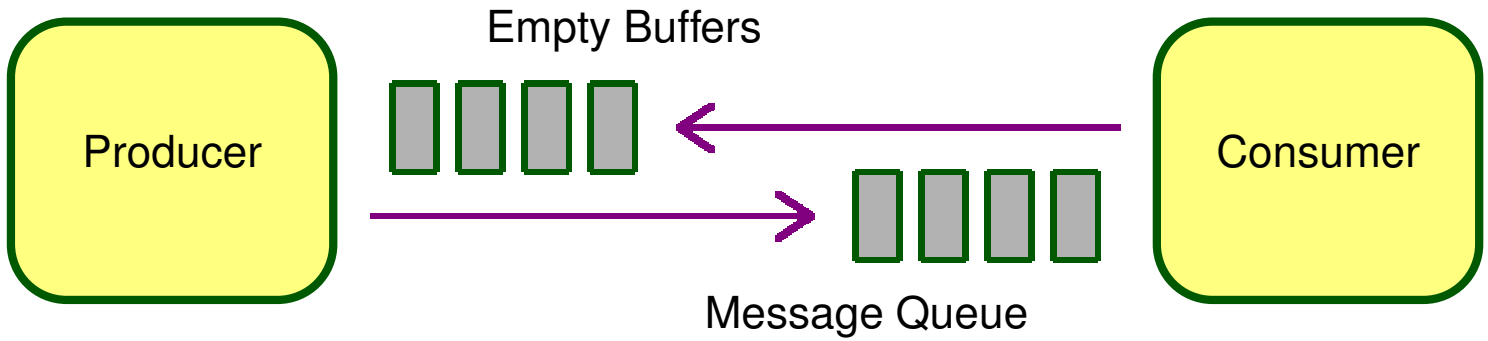
```
void receive(source,mess)
{
  if (mess from source in receiver.Mess-Q) {
     remove mess from receiver.Mess_Q
     return buffer to kernel;
  }
  else state = waiting; /* for a message from source */
}
```

Asynchronous Send

On sending a message, the kernel must check if the destination process is ready to receive. If so, it delivers the message, and reactivates the receiver; if not, the message is buffered, so that the sender *can* continue.

```
void send(destination,mess)
{
  if (destination waiting for message from current) {
    copy mess from sender space to destination space;
    ready(destination);
  }
  else if (getbuff(buffer) == FAIL)
                    /* get buffer from kernel buffer pool */
      error(nobuffers);
    else {
      buffer = mess; /* Save message in buffer */
      insert buffer at end of receiver.Mess_Q;
    }
  }   /* sender continues */
}
```

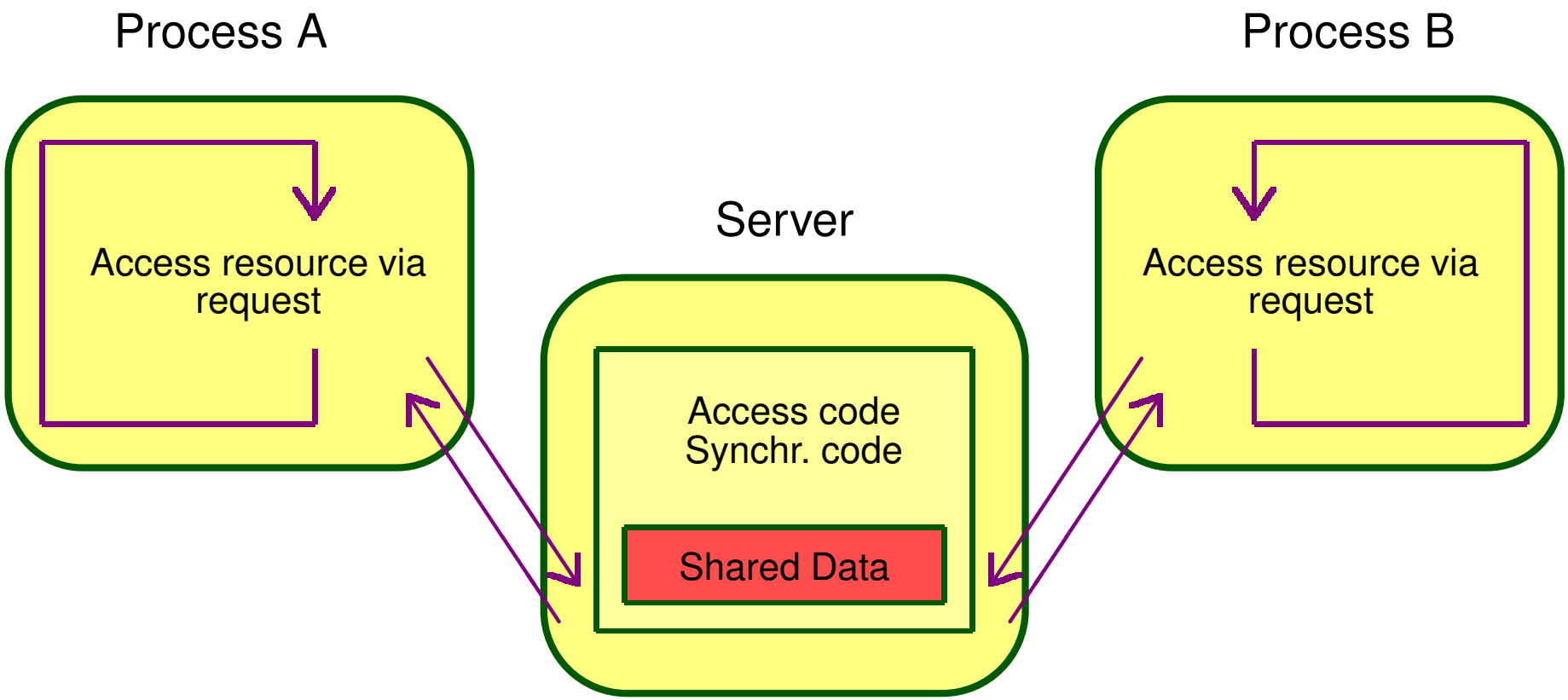
Producer - Consumer with Flow Control




```
process Producer {
  for (;;) {
    produce(item); receive(Consumer,mess);
    build(mess,item); send(Consumer,mess);
} }

process Cons (void) {
  #define N 100; message messages[N]; message mess;
  for (i=0;i<N;i++)  send(Producer,messages[i]);
  for (;;) {
    receive(Producer,mess); extract(mess,item);
    send(Producer,mess); consume(item);
} }
```

Server Paradigm



- Data access and synchronisation code within manager. This is a sequential program that receives requests, and sends replies.
- Manager is a separate compilation unit and syntactic unit, c.f. abstract data type or monitor.

Server Paradigm (2)

Advantages – modularity.

- clear interfaces (messages).
- mutual exclusion within manager.

Disadvantage overhead of message passing.

Implementation scheme

```
process Server (void) {
  % process_id client;
  % request Service_Request;
  % reply Service_Reply;
  for (;;) {
    client = receive_any(request);
      perform service request
    send(client,reply);
} }
```

Server process always has exclusive access to its own data, so mutual exclusion is automatic.

Example: Readers/Writers to Database

Database accessible for all processes, but access under control of *one* process, the database manager.

```
process Reader {
  for (;;) {
    send(DB_Manager, start_read);
    receive (DB_Manager,message);   /* wait for ok */
      read from database
    send(DB_Manager, end_read);
      use data
} }
process Writer {
  for (;;) {
      generate data
    send(DB_Manager, start_write);
    receive (DB_Manager, message);   /* wait for ok */
      write database
    send(DB_Manager, end_write);
      use data
} }
```

DB_Manager

```
process DB_Manager {
  int reader_count = 0;
  int writing = FALSE;
  for (;;) {
    source = receive_any(message);
    switch (message) {
    case start_read:
      if (!writing) {
        send(source,OK);
        reader_count += 1; }
      else      /* Writer busy, queue Reader */
        add_to_queue (source, reader_Q);
    case end_read:
      reader_count -= 1;
      if (reader_count == 0 & !empty(writer_Q)) {
        /* First Writer on queue can write */
        source = remove_from_queue(writer_Q);
        send(source,OK);
        writing = TRUE;
      }
```

DB_Manager (2)

```
    case start_write:
      if (reader_count==0 & !writing)  {
        send(source,OK);
        writing = TRUE;
      }
      else add_to_queue(source, writer_Q);  /* DB in use */
    case end_write:
      writing = FALSE;
      if (empty(reader_Q) & !empty (writer_Q)) {
        source = remove_from_queue(writer_Q);
        send(source,OK);
        writing = TRUE;
      }
      else while (!empty(reader_Q)) {
             /* all queued Readers continue */
        source = remove_from_queue(reader_Q);
        send(source,OK);
        reader_count += 1;
  } } }
```

Synchronous Receive

Using this approach, the sender blocks until the message is received by the receiver. So the **OS** does not let the sending process run after **`send`** until the receiving process has executed **`receive`** for this message; the **`receive`** primitive remains unchanged.

Notice that:

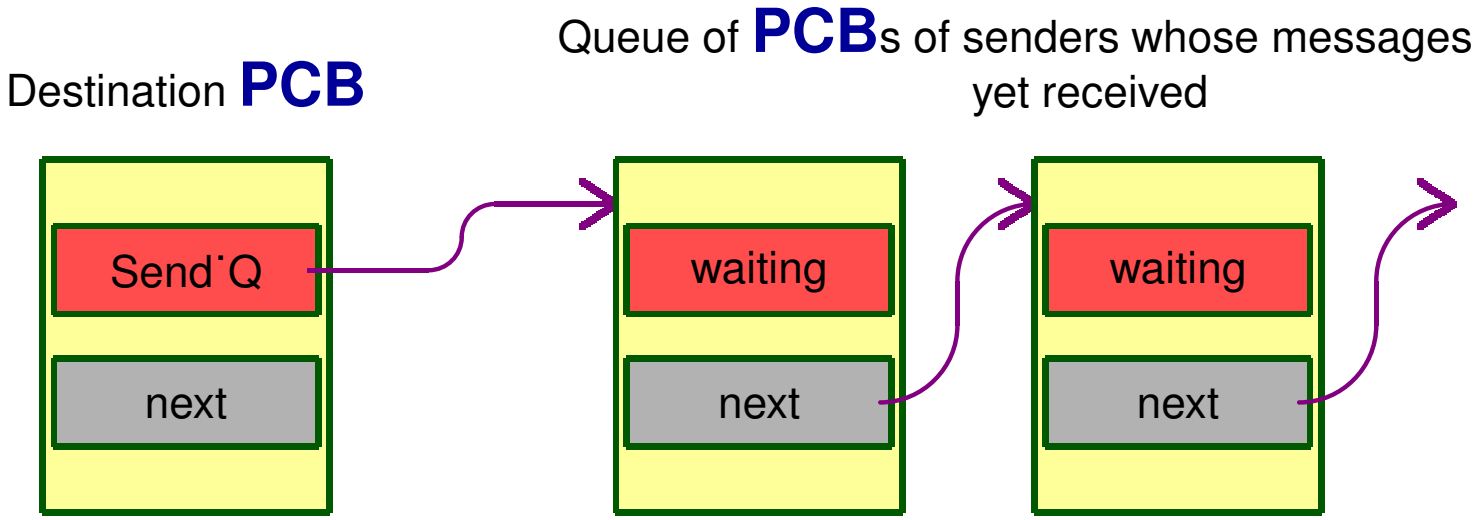
- No **OS** buffering is required.
- More efficient, since the message is copied only *once* (up to *twice* in Asynchronous Send).
- This solution allows for less parallelism.
- A slow receiver slows the sender down.

Asynchronous Send No receivers: messages queued.

Synchronous Send No receivers: senders queued.

For both: receiver is queued if no message is available.

Synchronous Receive: Queue




```
void receive(source,message) {
  if (source in sender.queue) {
     remove PCB from sender.queue;
     copy message from source 's to receiver's space;
     ready(source);   /* put in Ready Queue
        * Message is received, so source can continue */
  }
  else current.state = waiting; /* for source */
}
```

Synchronous Send

On sending a message, the kernel must check if the destination process is waiting to receive. If so, it delivers the message, and reactivates the receiver; if not, the sending process is taken of the Ready Q, attached to Sender Q, so the sender *cannot* continue.

```
void send(destination,mess)
{
  if (destination waiting for message from sender)  {
    copy mess from sender's to destination's space;
    ready(destination);  /* unblock receiver
      * blocked on receive and put in Ready Queue.
      * Message is received, so sender can continue */
  }
  else {
    unready(sender);
    sender.state = waiting; /* put sender to sleep, waiting for
      * destination to accept message from queue */
    insert current at end of destination's Sender Queue;
  }
}
```

Send Message in Minix (in `proc.c`)

```
PRIVATE int mini_send(caller_ptr, dst, m_ptr, flags)
register struct proc *caller_ptr;   /* who is trying to send?    */
int dest;                           /* to whom is message being sent?    */
message *m_ptr;                     /* pointer to message buffer   */
unsigned flags;                     /* system call flags   */
 {
 /* Send a message from  caller_ptr to  dest. If  dest is blocked waiting
  * for this message, copy the message to it and unblock  dest. If  dest is
  * not waiting at all, or is waiting for another source, queue  caller_ptr.    */
   register struct proc *dest_ptr = proc_addr(dst);
   register struct proc **xpp;

    /* Check if  dst is blocked waiting for this message. The destination's
     *  SENDING flag may be set when its  SENDREC call blocked while sending.    */
   if ((dest_ptr->p_flags & (RECEIVING | SENDING)) == RECEIVING
         && (dest_ptr->p_getfrom == ANY ||
         dest_ptr->p_getfrom == caller_ptr->p_nr))  {
       /* Destination is indeed waiting for this message.    */

            ⋮
```

Send Message in Minix (2)

```
CopyMess(caller_ptr->p_nr, caller_ptr, m_ptr, dst_ptr,
        dst_ptr->p_messbuf);
   dest_ptr->p_flags &=  ~RECEIVING;  /* deblock destination  */
   if ((dst_ptr->p_rts_flags &=  ~RECEIVING) == 0) enqueue(dst_ptr)
    } else if ( ! (flags & NON_BLOCKING))  {
    /* Destination is not waiting. Block and queue caller.    */
      caller_ptr->p_messbuf = m_ptr;
      if (caller_ptr->p_rts_flags == 0) dequeue(caller_ptr);
      caller_ptr->p_rts_flags |= SENDING;
      caller_ptr->p_sendto = dst;

       /* Process is now blocked. Put in on the destination's queue.    */
      xpp = &dst_ptr->p_caller_q;                 /* find end of list  */
      while (*xpp != NIL_PROC) xpp = &(*xpp)->p_q_link;
      *xpp = caller_ptr;                          /* add caller to end  */
      caller_ptr->p_q_link = NIL_PROC;            /* mark new end of list  */
    } else  {
      return(ENOTREADY);
    }
   return(OK);
}
```

Receive Message in Minix

```
PRIVATE int mini_receive(caller_ptr, src, m_ptr, flags)
register struct proc *caller_ptr;   /* process trying to get message   */
int src;                            /* which message source is wanted (or ANY)   */
message *m_ptr;                     /* pointer to message buffer   */
unsigned flags;                     /* system call flags   */
 {   /* A process or task wants to get a message. If a message is already queued,
  *  acquire it and deblock the sender. If no message from the desired source
  *  is available block the caller, unless the flags don't allow blocking.     */
   register struct proc **xpp;
   register struct notification **ntf_q_pp;
   message m;
   int bit_nr;
   sys_map_t *map;
   bitchunk_t *chunk;
   int i, src_id, src_proc_nr;
    /* Check to see if a message from desired source is already available.
     * The caller's SENDING flag may be set if SENDREC couldn't send. If it is
     * set, the process should be blocked.     */
   if (!(caller_ptr->p_rts_flags & SENDING))   {
        /* Check if there are pending notifications, except for SENDREC.   */
```

omitted from notes

Receive Message in Minix (2)

```
/* Check caller queue. Use pointer pointers to keep code simple.     */
  xpp = &caller_ptr->p_caller_q;
  while (*xpp != NIL_PROC)  {
     if (src == ANY || src == proc_nr(*xpp))  {
          /* Found acceptable message. Copy it and update status.     */
        CopyMess((*xpp)->p_nr, *xpp, (*xpp)->p_messbuf,
           caller_ptr, m_ptr);
        if (((*xpp)->p_rts_flags &= ~SENDING) == 0) enqueue(*xpp)
        *xpp = (*xpp)->p_q_link;          /* remove from queue   */
        return(OK);                       /* report success   */
      }
     xpp = &(*xpp)->p_q_link;             /* proceed to next   */
 }  }
 /* No suitable message is available or the caller couldn't send in  SENDREC.
  * Block the process trying to receive, unless the flags tell otherwise.     */
if ( ! (flags & NON_BLOCKING))  {
   caller_ptr->p_getfrom = src;
   caller_ptr->p_messbuf = m_ptr;
   if (caller_ptr->p_rts_flags == 0) dequeue(caller_ptr);
   caller_ptr->p_rts_flags |= RECEIVING;
   return(OK);
 } else  {return(ENOTREADY);  } }
```

Additional Message Primitives

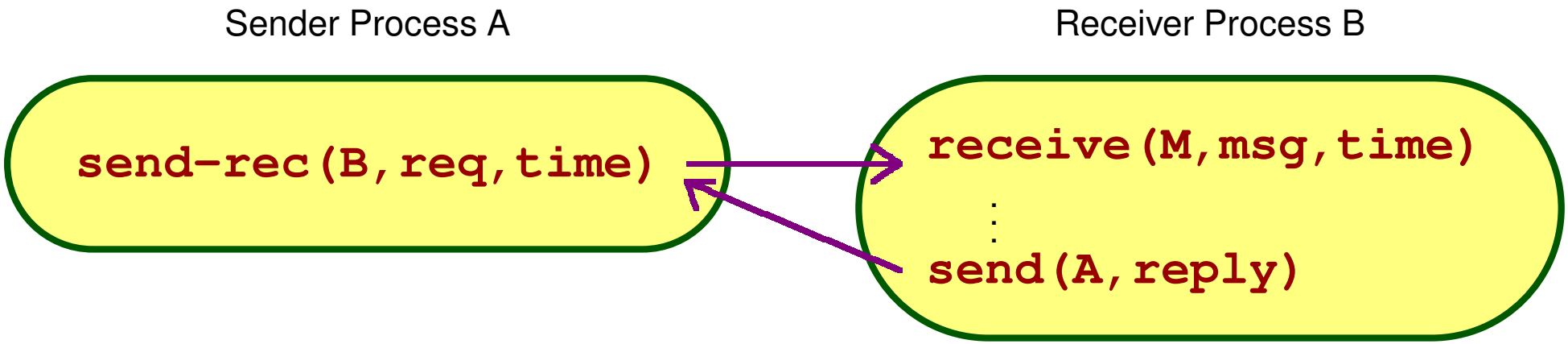
Time-outs Can be used for synchronous **`send`** and **`receive`** to limit the amount of time spent on waiting.
Looks like: **`receive(P,msg,time)`**.

Conditional Receive Allows the choice between receiving a message only if one is waiting; else continue with other things. This option permits the receiver to check for a message without being blocked. This correspond to *polling for input* instead of *waiting for interrupt* as is the normal case; it's a busy-wait.

Indirect Naming May be used instead of naming the process from which a message is received or to which a message is sent. Sender does not know name of receiver and receiver does not know name of sender.

Request Reply

- Bi-directional communication that uses a combined **`send`** and **`receive`** (reply) mechanism.
- Synchronous - sender blocks while waiting for reply.
- Time-out may be used to limit time for which sender (or receiver) is blocked.
- Receiver continues immediately after sending reply, as can be guaranteed that the message is received immediately; the sender was waiting for it.



Selective Receive

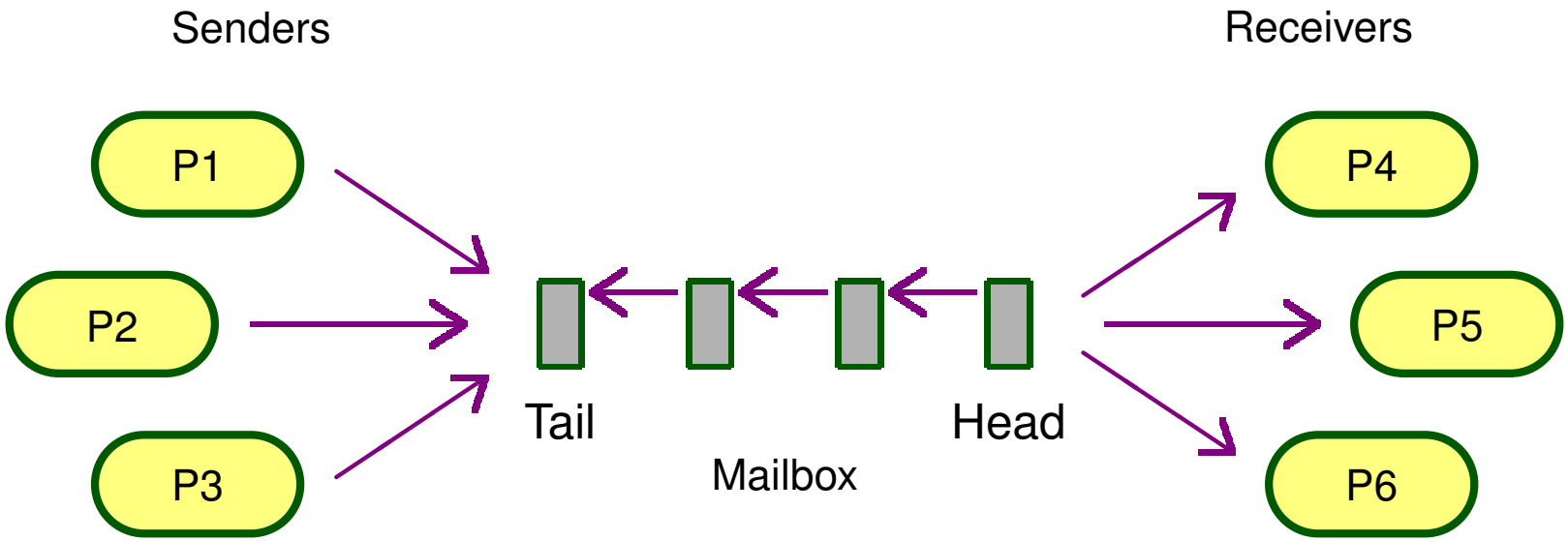
Select messages from multiple sources. as an alternative to **`receive_any`**; e.g. Ada, Conic.

```
SELECT
  WHEN G1 receive(p1,msg1)
    DO ... END
OR
  WHEN G2 receive(p2,msg2)
    DO ... END
OR
  DELAY (50)
END
```

Other alternatives: **`ELSE`**, **`TERMINATE`**

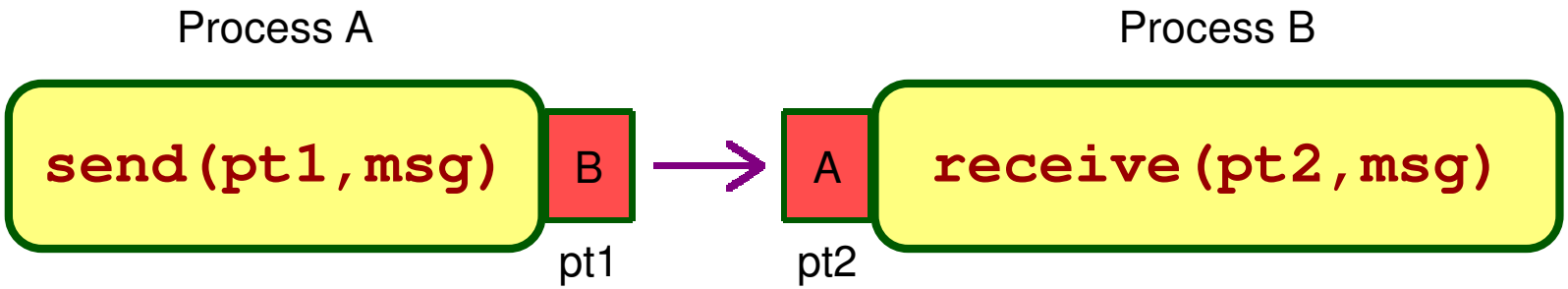
Mailbox

An **OS** data structure (a queue) independent of both sender and receiver. Queues either processes or messages depending on which type of communication primitive is being used. Permits multiple senders and receivers. May be bounded or unbounded queue.



Ports or Sockets

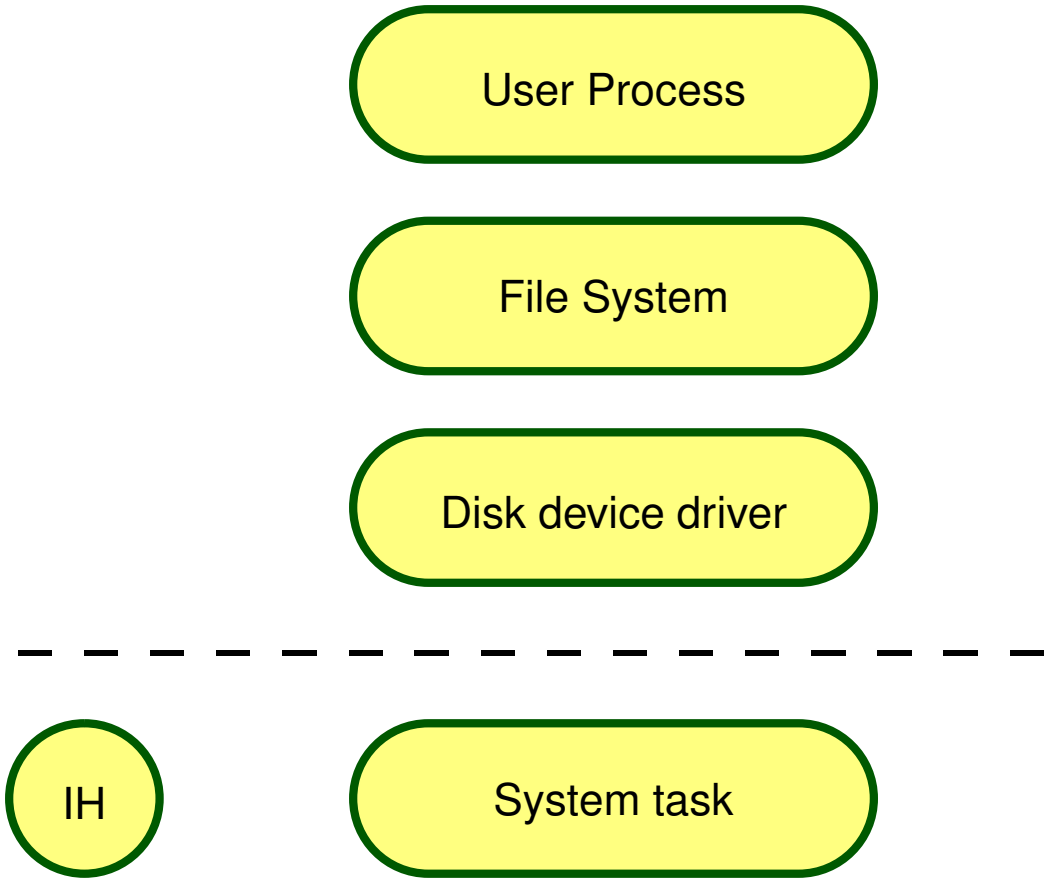
Data structure, maintained by kernel, within a process, associated to another process. The port holds name of source or destination, and is used to pass messages through. Its name is system-wide unique, and a process can have more than one port.



Needs a primitive for binding, **`bind(pt1,B)`**, when binding between ports and processes is not permanent; places name of process in **`pt1`** data structure. The **OS** picks up the name when the message is sent.

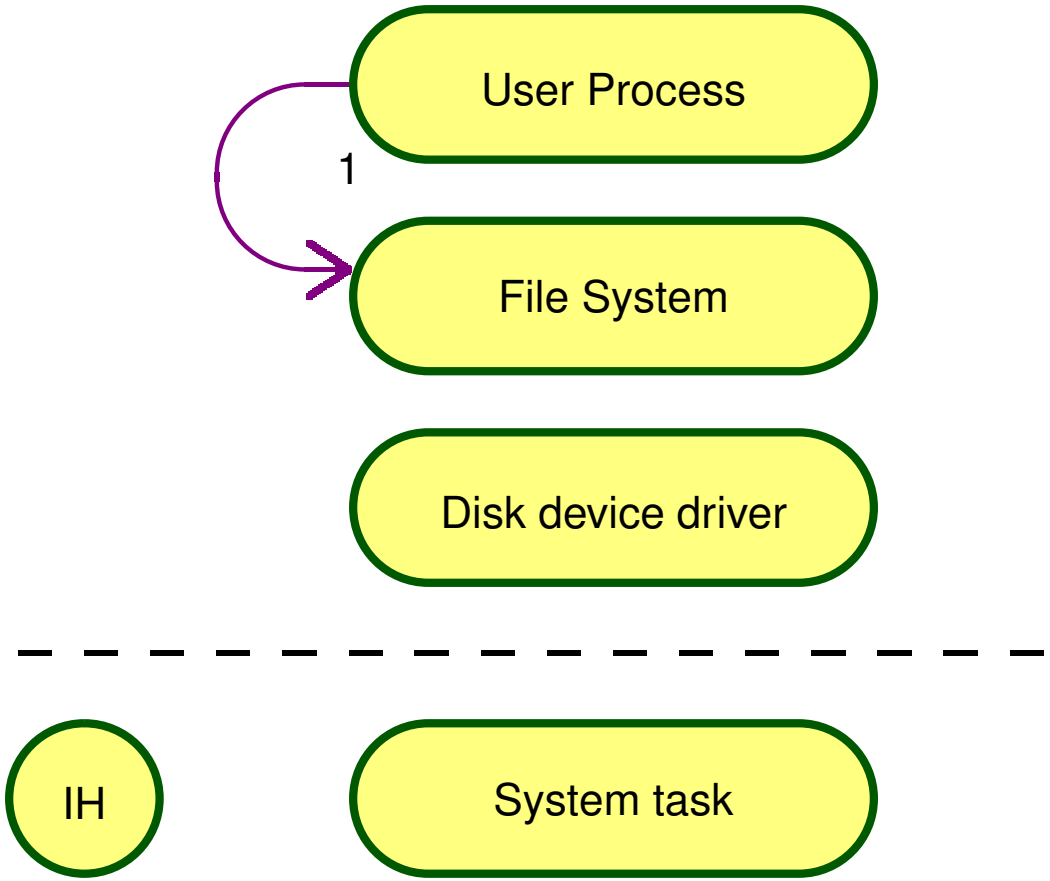
Minix I/O *System Structuring*

- Four *independent* processes; IH runs in context of kernel.
- Uses message passing; reading a file from disk results in 11 sent messages:



Minix I/O *System Structuring*

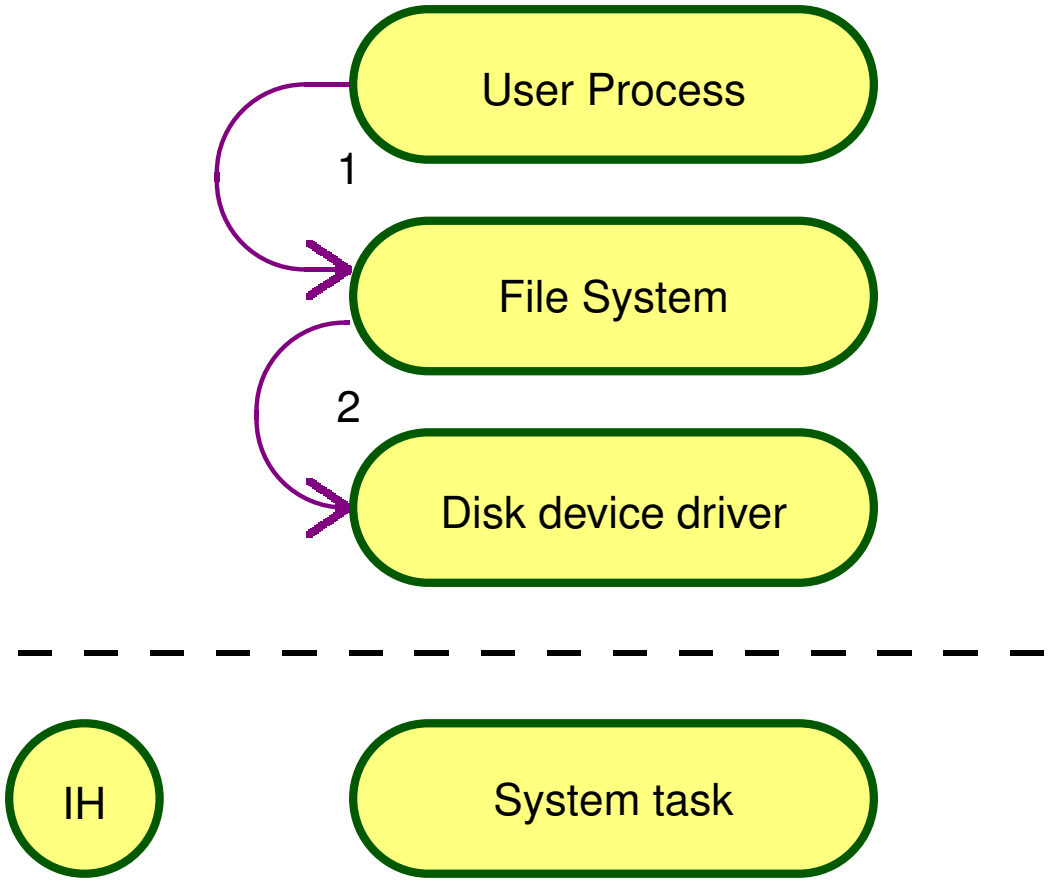
- Four *independent* processes; IH runs in context of kernel.
- Uses message passing; reading a file from disk results in 11 sent messages:



1: I/O Request

Minix I/O *System Structuring*

- Four *independent* processes; IH runs in context of kernel.
- Uses message passing; reading a file from disk results in 11 sent messages:

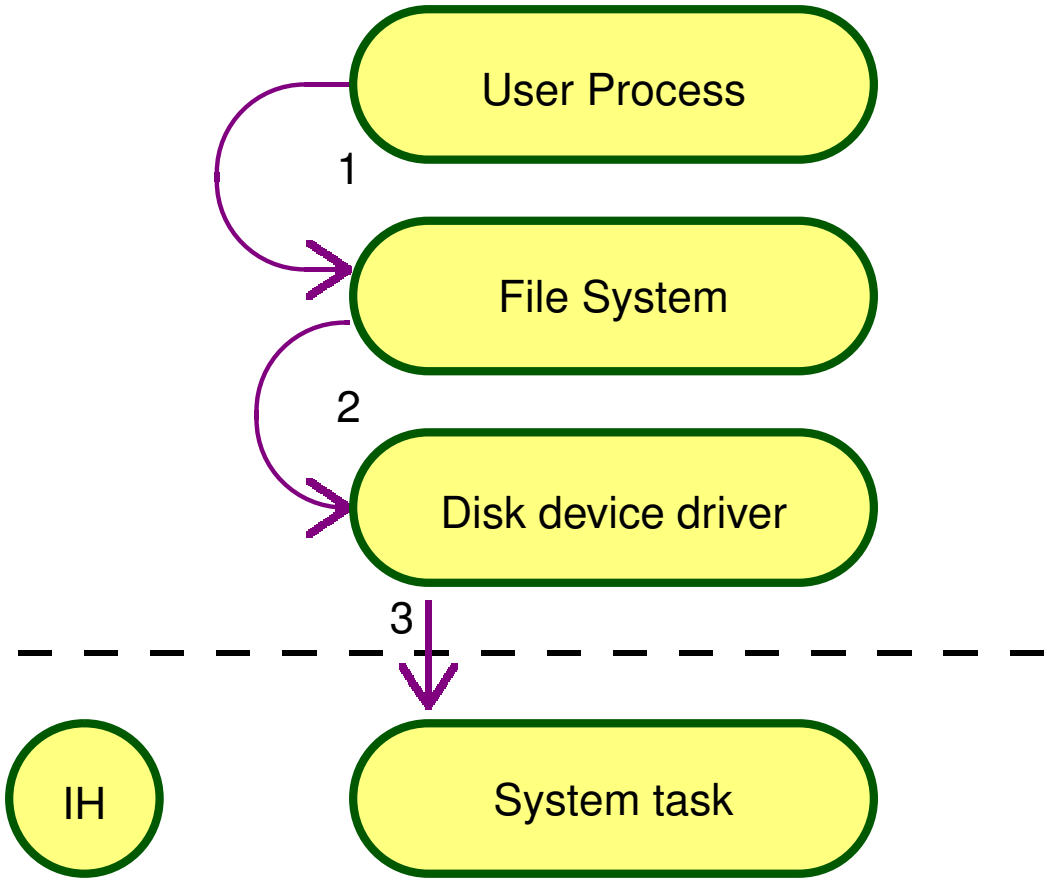


1: I/O Request

2: Forward request

Minix **I/O** *System Structuring*

- Four *independent* processes; IH runs in context of kernel.
- Uses message passing; reading a file from disk results in 11 sent messages:



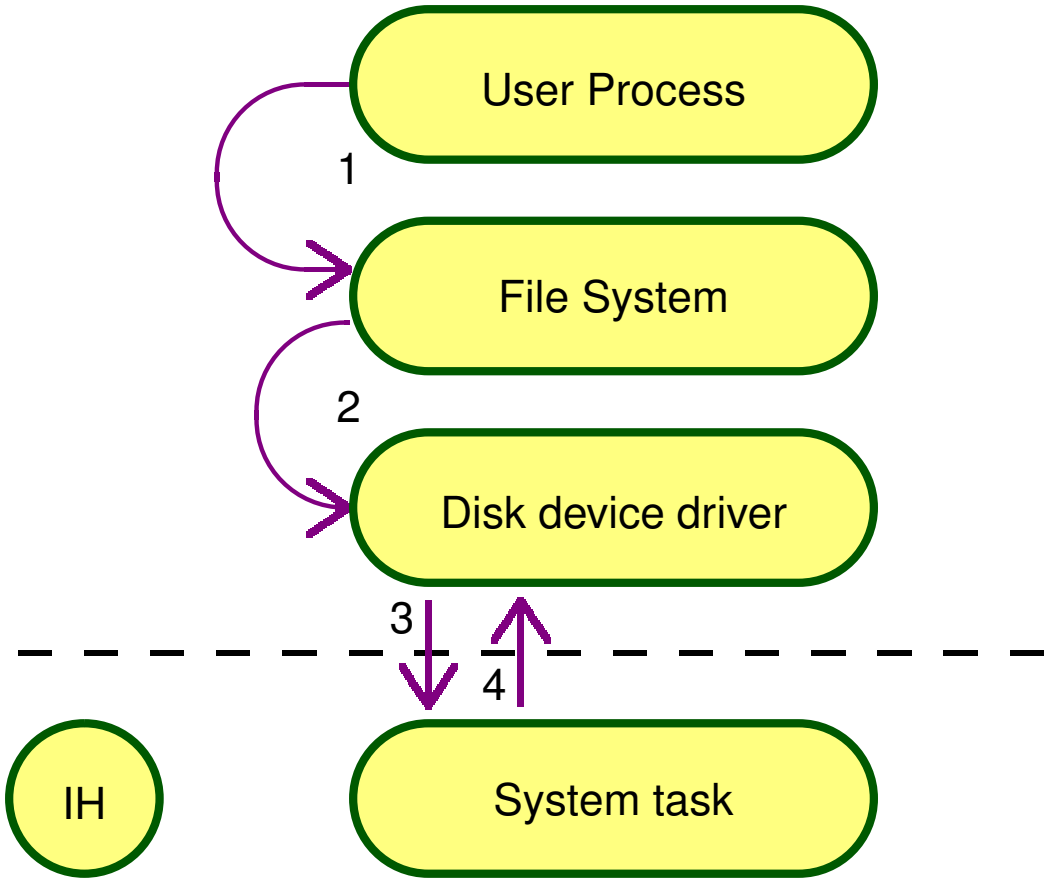
1: **I/O** Request

2: Forward request

3: Forward Request

Minix **I/O** *System Structuring*

- Four *independent* processes; IH runs in context of kernel.
- Uses message passing; reading a file from disk results in 11 sent messages:



1: **I/O** Request

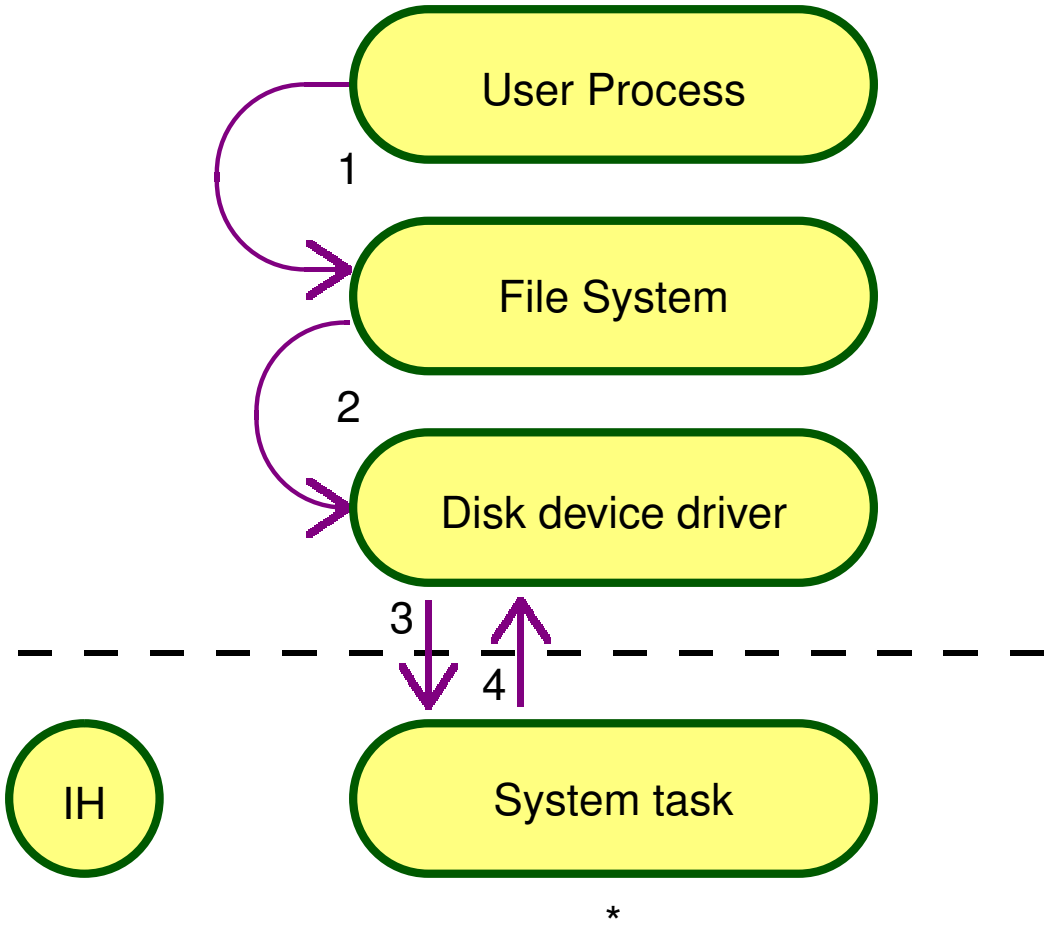
2: Forward request

3: Forward Request

4: Acknowledge

Minix **I/O** *System Structuring*

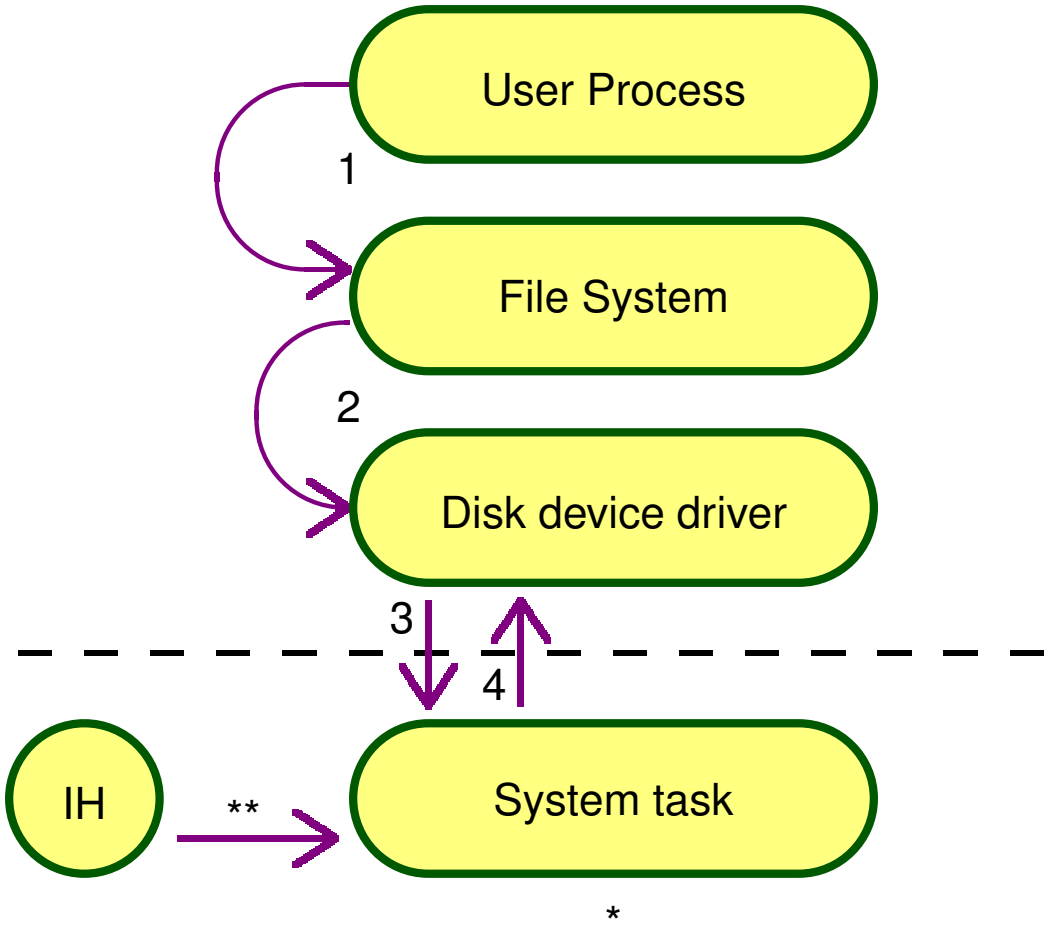
- Four *independent* processes; IH runs in context of kernel.
- Uses message passing; reading a file from disk results in 11 sent messages:



1: **I/O** Request
2: Forward request
3: Forward Request
4: Acknowledge
*: **I/O** is initiated by the task

Minix I/O *System Structuring*

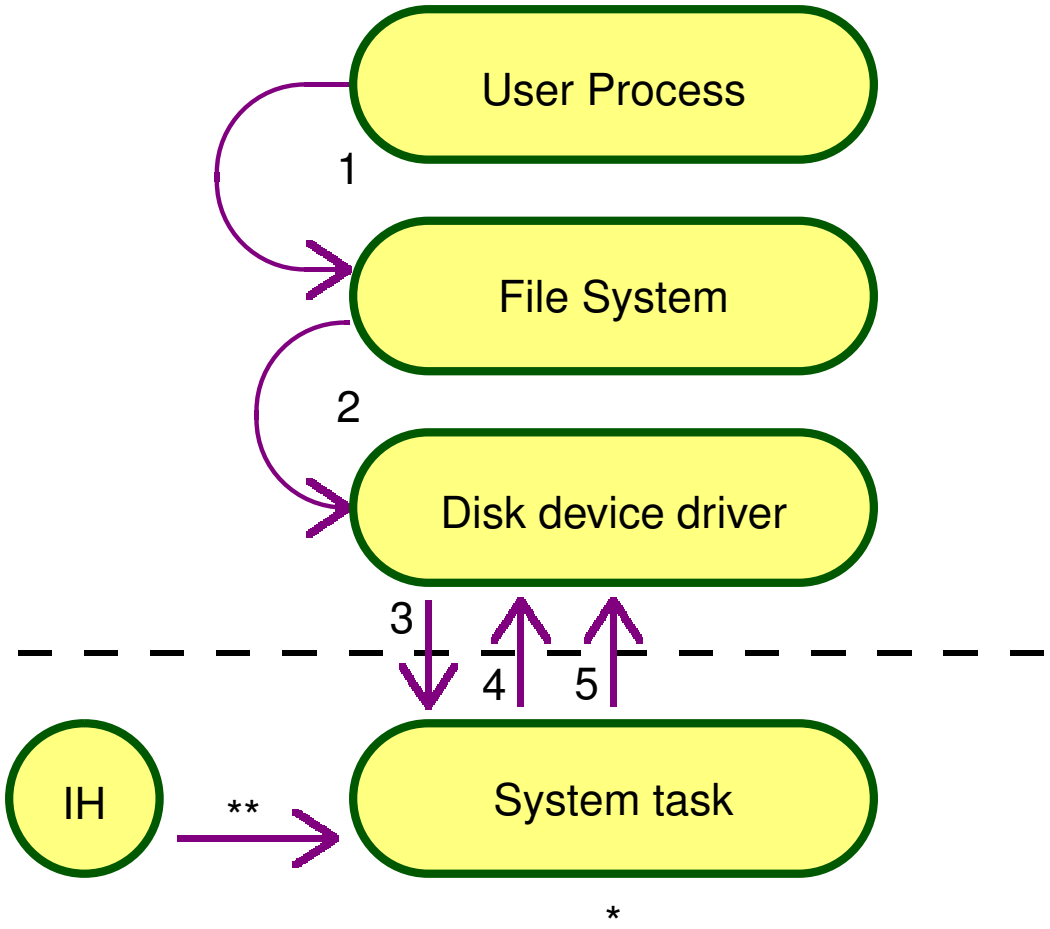
- Four *independent* processes; IH runs in context of kernel.
- Uses message passing; reading a file from disk results in 11 sent messages:



1: I/O Request
2: Forward request
3: Forward Request
4: Acknowledge
*: I/O is initiated by the task
**: I/O complete

Minix **I/O** *System Structuring*

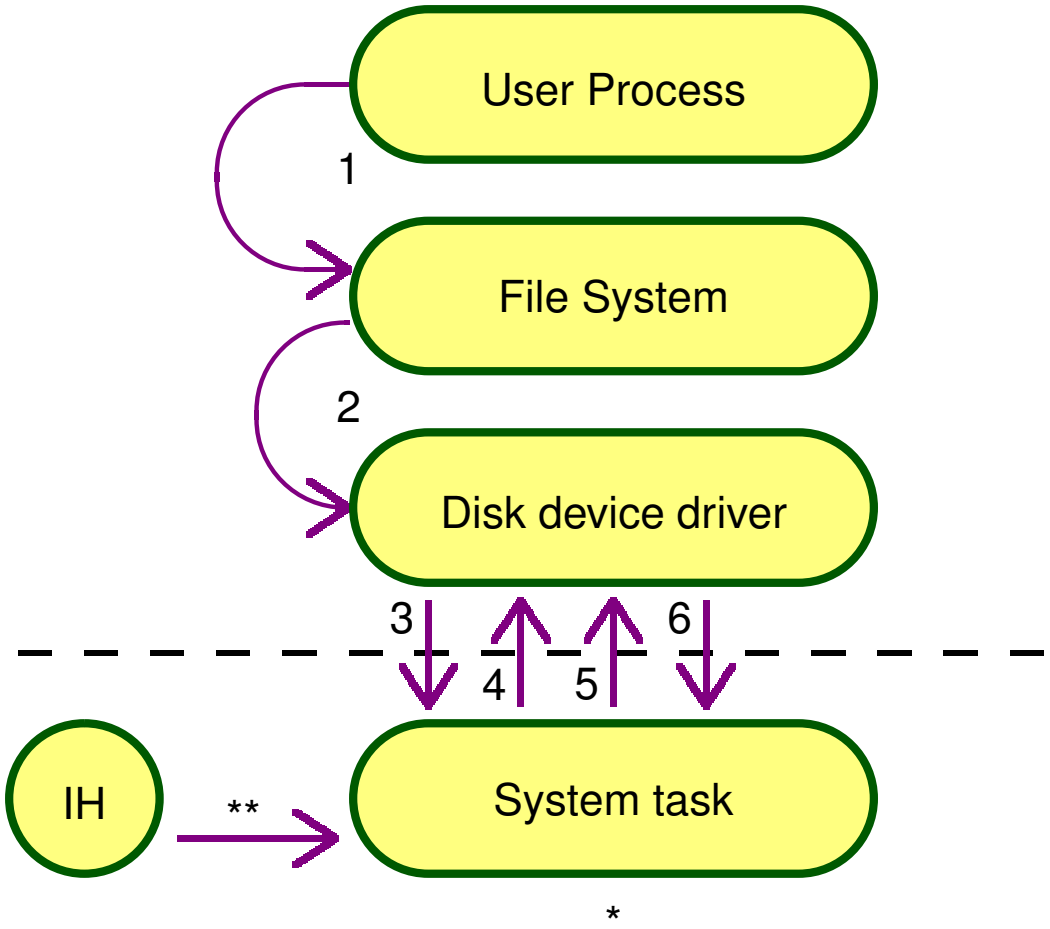
- Four *independent* processes; IH runs in context of kernel.
- Uses message passing; reading a file from disk results in 11 sent messages:



1: **I/O** Request
2: Forward request
3: Forward Request
4: Acknowledge
*: **I/O** is initiated by the task
: **I/O complete
5: Inform

Minix I/O *System Structuring*

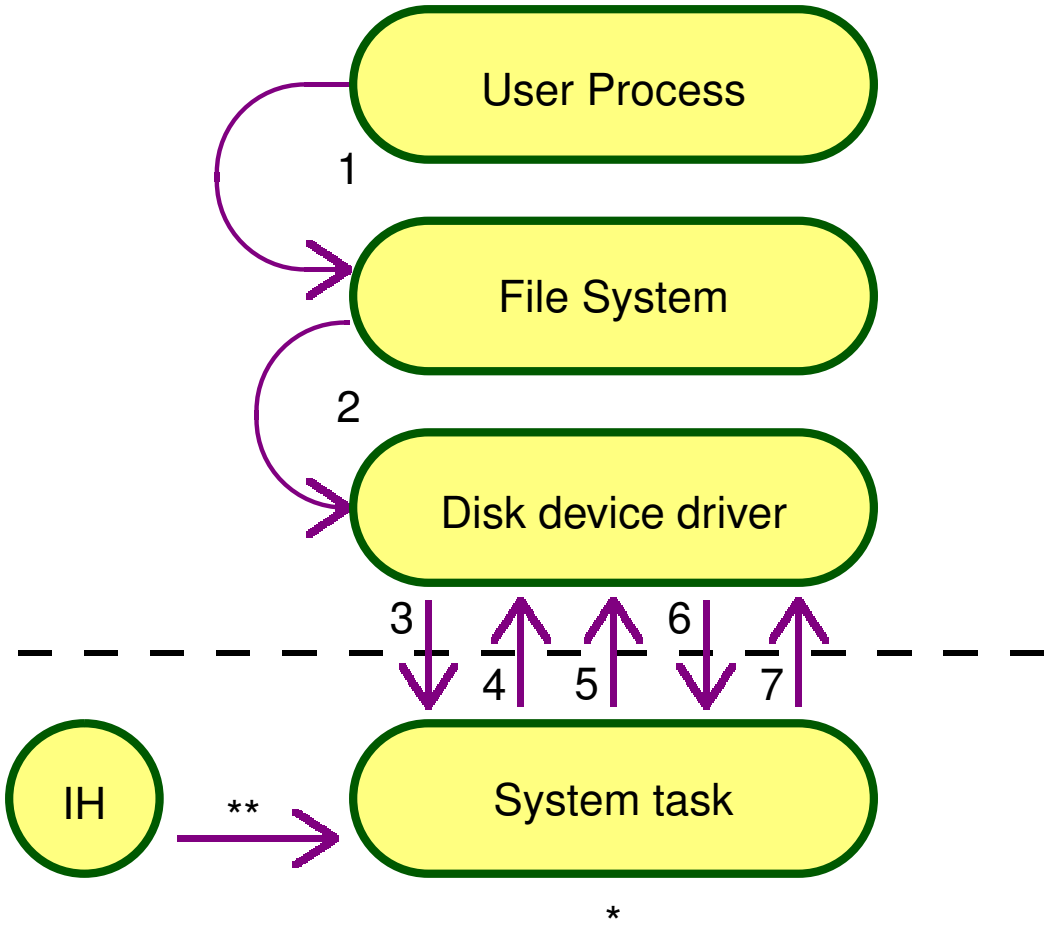
- Four *independent* processes; IH runs in context of kernel.
- Uses message passing; reading a file from disk results in 11 sent messages:



1: I/O Request
2: Forward request
3: Forward Request
4: Acknowledge
*: I/O is initiated by the task
**: I/O complete
5: Inform
6: Request copy data to FS cache

Minix I/O *System Structuring*

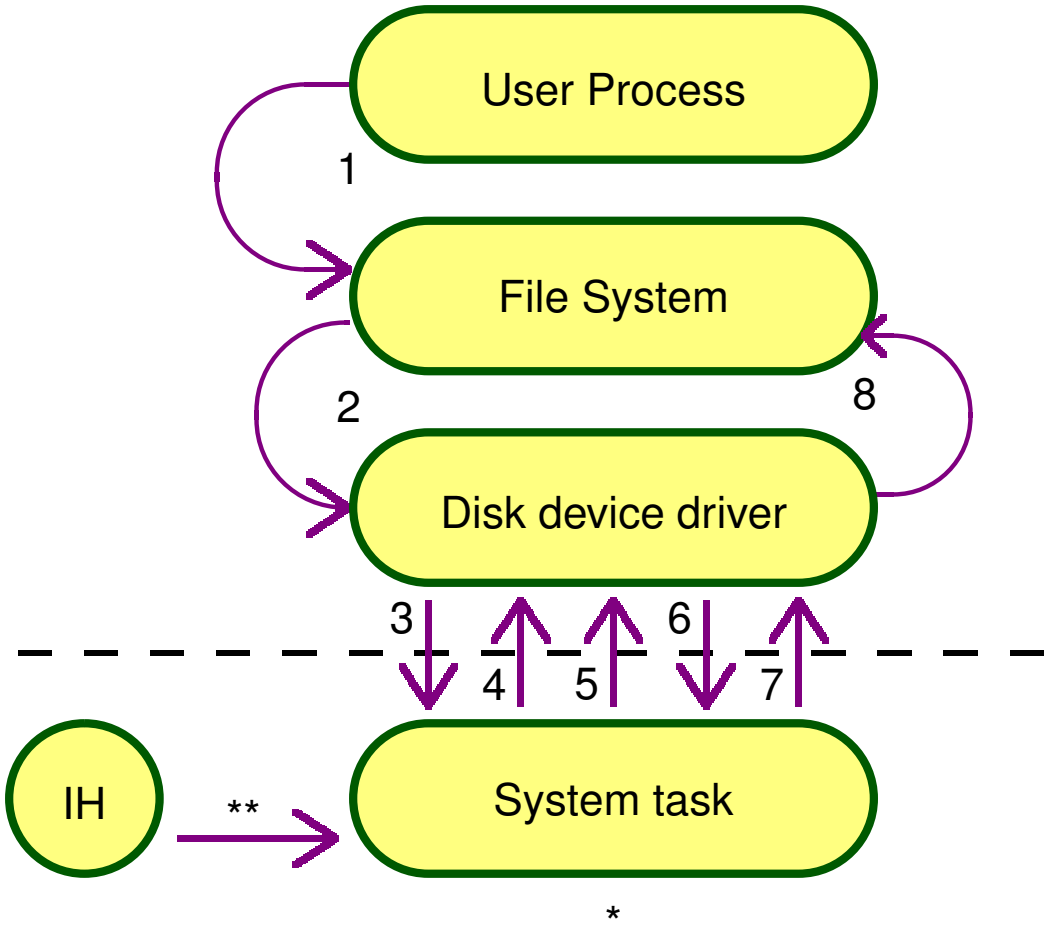
- Four *independent* processes; IH runs in context of kernel.
- Uses message passing; reading a file from disk results in 11 sent messages:



1: I/O Request
2: Forward request
3: Forward Request
4: Acknowledge
*: I/O is initiated by the task
**: I/O complete
5: Inform
6: Request copy data to FS cache
7: Give data

Minix I/O *System Structuring*

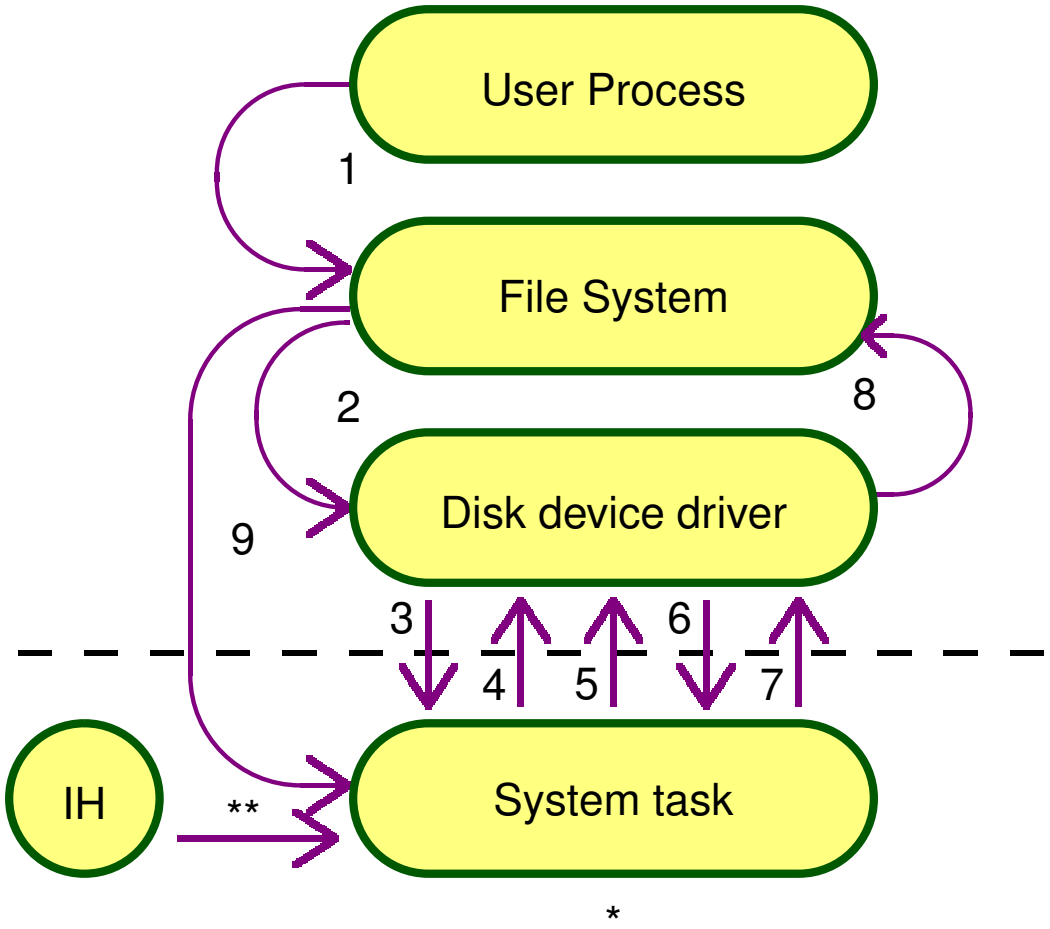
- Four *independent* processes; IH runs in context of kernel.
- Uses message passing; reading a file from disk results in 11 sent messages:



1: I/O Request
2: Forward request
3: Forward Request
4: Acknowledge
*: I/O is initiated by the task
**: I/O complete
5: Inform
6: Request copy data to FS cache
7: Give data
8: Inform FS: data ready

Minix I/O *System Structuring*

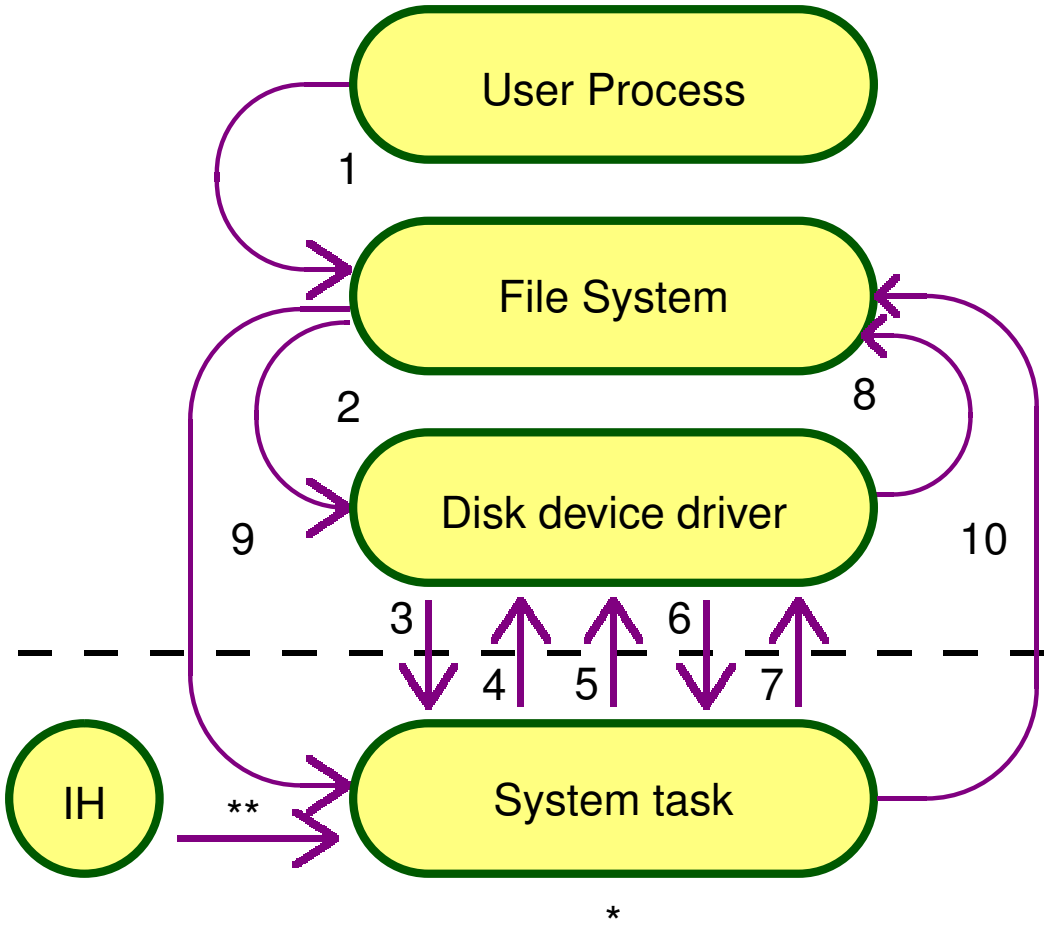
- Four *independent* processes; IH runs in context of kernel.
- Uses message passing; reading a file from disk results in 11 sent messages:



1: I/O Request
2: Forward request
3: Forward Request
4: Acknowledge
*: I/O is initiated by the task
**: I/O complete
5: Inform
6: Request copy data to FS cache
7: Give data
8: Inform FS: data ready
9: Ask data

Minix I/O *System Structuring*

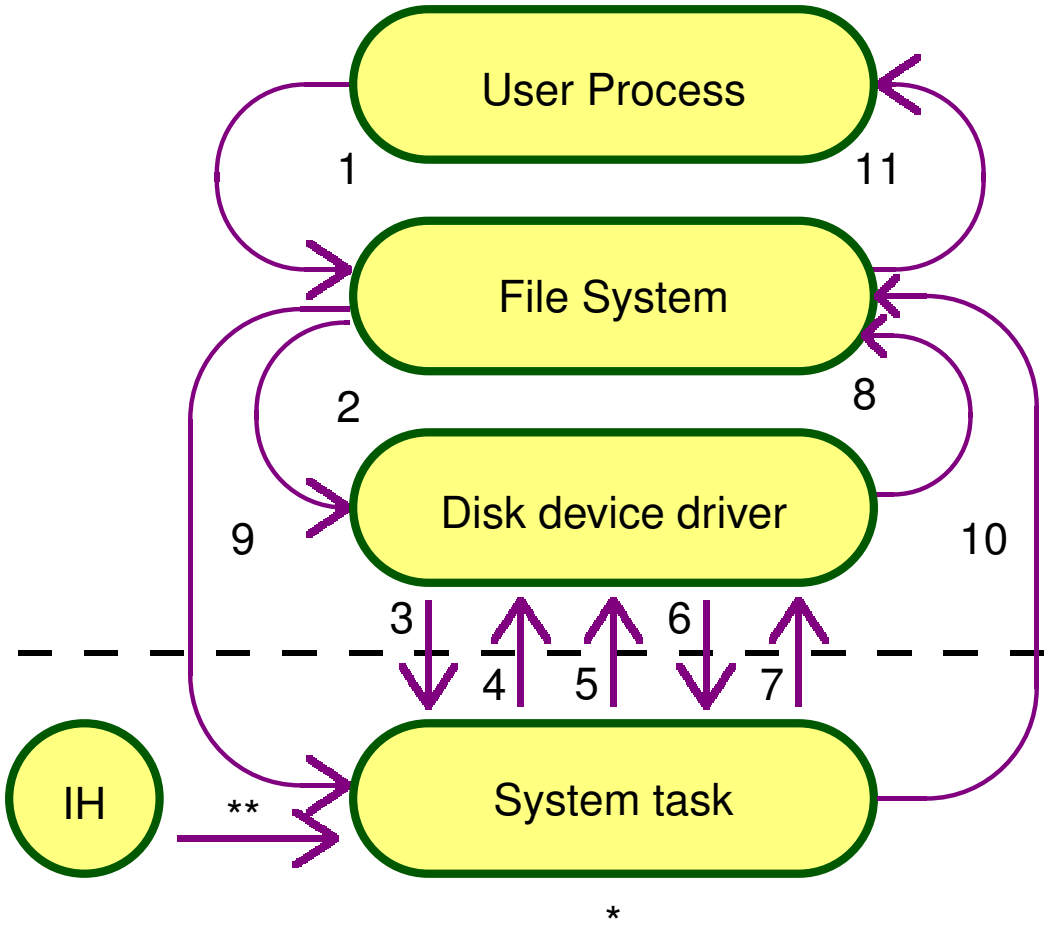
- Four *independent* processes; IH runs in context of kernel.
- Uses message passing; reading a file from disk results in 11 sent messages:



1: I/O Request
2: Forward request
3: Forward Request
4: Acknowledge
*: I/O is initiated by the task
**: I/O complete
5: Inform
6: Request copy data to FS cache
7: Give data
8: Inform FS: data ready
9: Ask data
10: Supply data

Minix I/O *System Structuring*

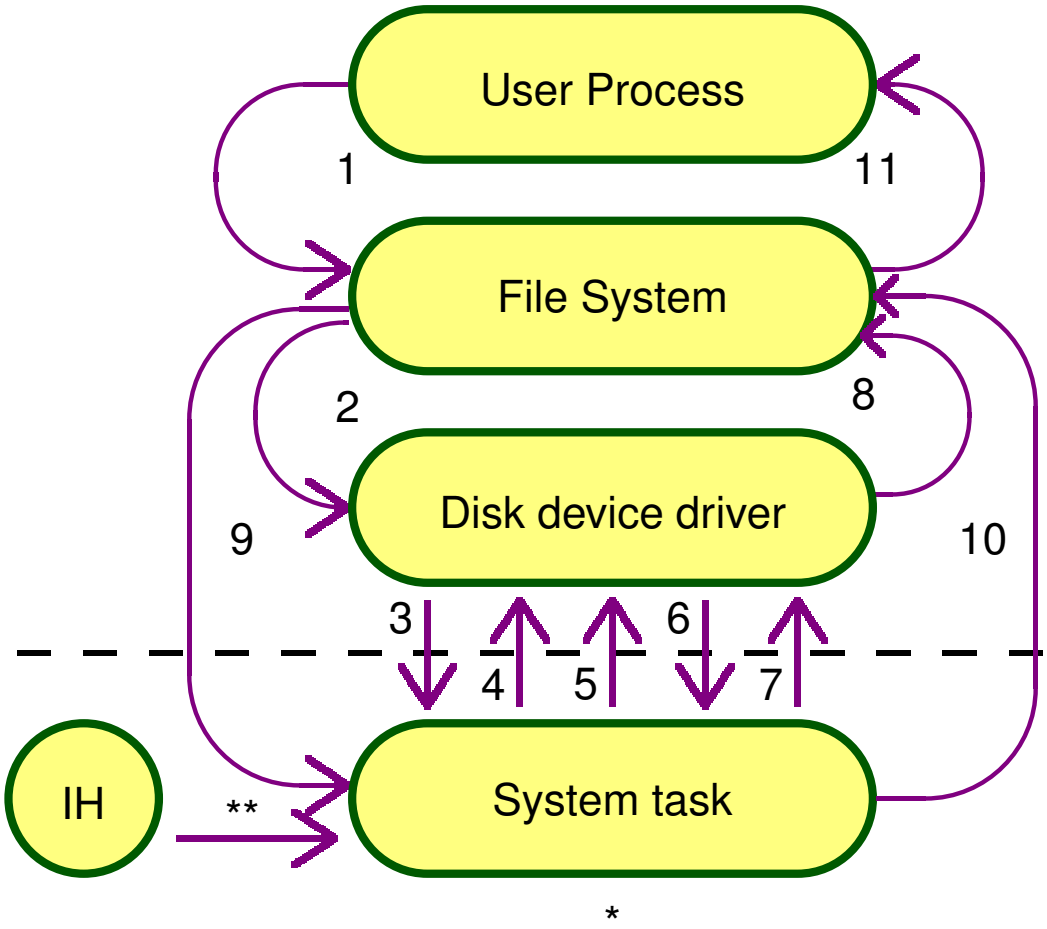
- Four *independent* processes; IH runs in context of kernel.
- Uses message passing; reading a file from disk results in 11 sent messages:



1: I/O Request
2: Forward request
3: Forward Request
4: Acknowledge
*: I/O is initiated by the task
**: I/O complete
5: Inform
6: Request copy data to FS cache
7: Give data
8: Inform FS: data ready
9: Ask data
10: Supply data
11: Reply to user process

Minix I/O *System Structuring*

- Four *independent* processes; IH runs in context of kernel.
- Uses message passing; reading a file from disk results in 11 sent messages:



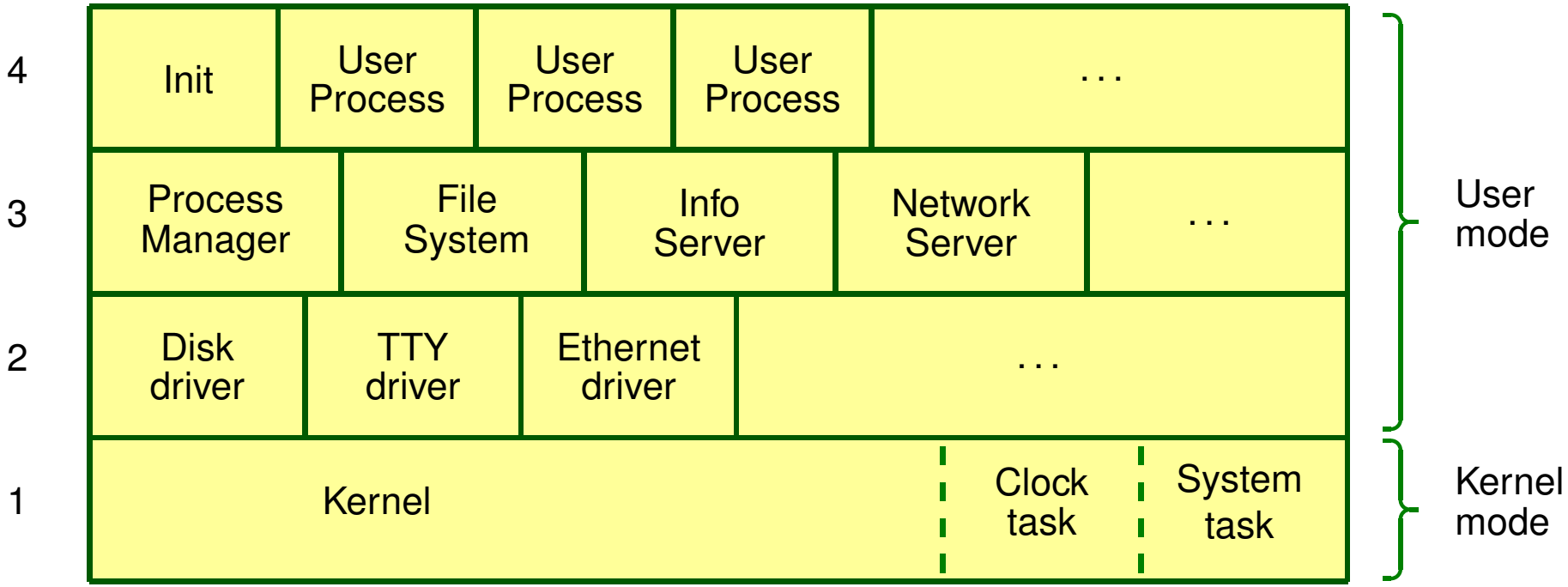
1: I/O Request
2: Forward request
3: Forward Request
4: Acknowledge
*: I/O is initiated by the task
**: I/O complete
5: Inform
6: Request copy data to FS cache
7: Give data
8: Inform FS: data ready
9: Ask data
10: Supply data
11: Reply to user process

- It has a *single* File System process, which gives a bottleneck.

Minix I/O System Structuring (2)

```
process Device_Handler () {
  initialise data structures and device;
  while (TRUE)  {
    receive (FS,ReqMessage);
                /* wait for request to come in from FS */
      switch (ReqMessage.type) {
                case read:   initiate requested read operation;
                case write:  initiate requested write operation;
                case ioctl:  initiate requested control operation;
      };
      receive (IntMes);   /* wait for I/O completion */
      check for errors;
      set up IntMes with status to send to FS;
      send (FS,IntMes)
    }
  }
}
```

Process Structure of Minix



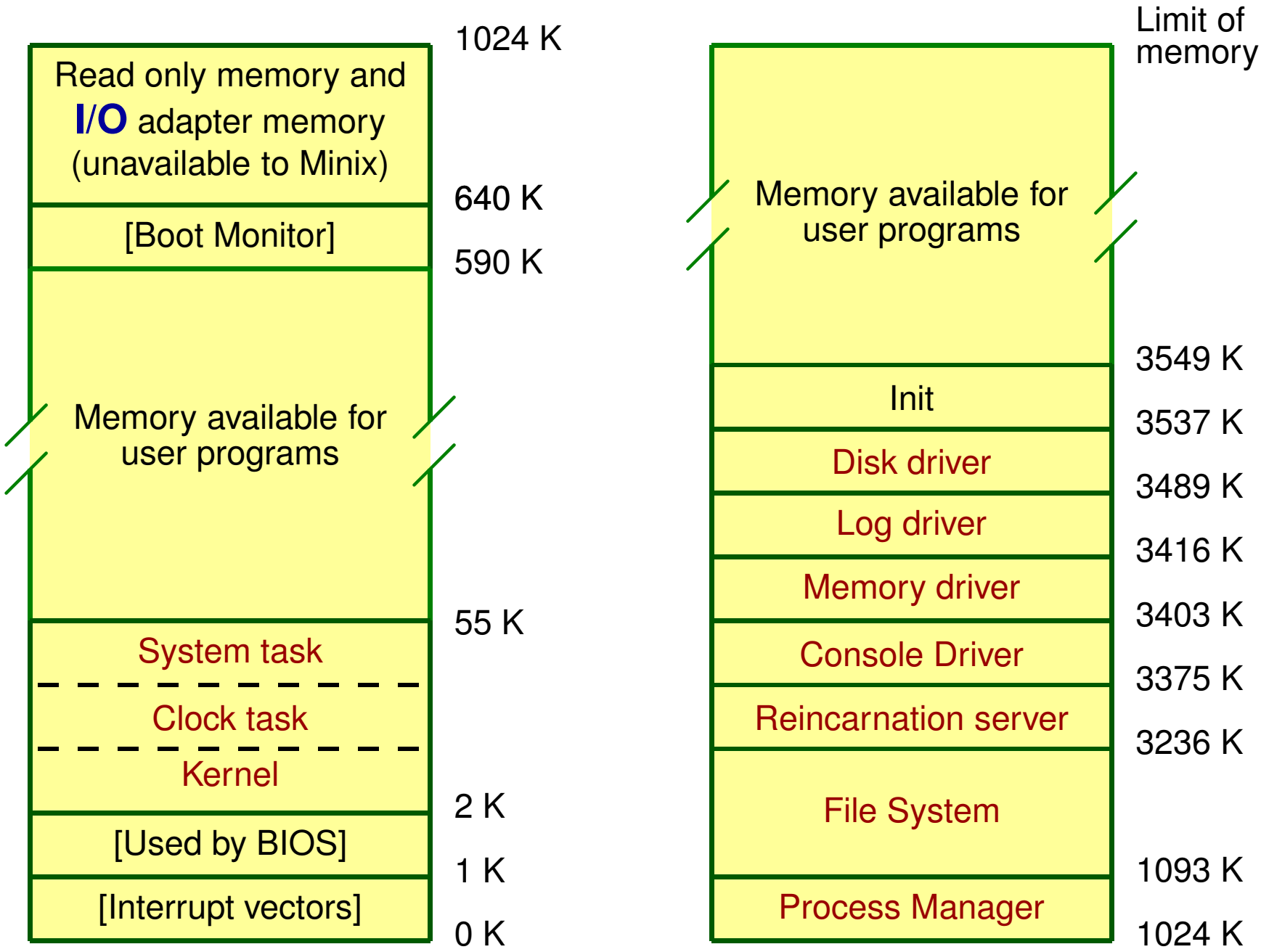
4 User processes.

3 Server processes.

2 Device drivers.

1 Kernel.

Memory Lay-out of Minix



Example: Mach

A system built as an extension of Accent, used in NeXTStep / Openstep / Apple's OS X (Darwin). It provides mailboxes, but calls them ports. Mailbox provides message buffers (default size 8).

Mach supports six basic abstractions for message passing.

task Execution environment, unit of resource allocation.

thread The basic unit of execution.

port A one-way communication channel implemented as a message queue managed by the kernel.

ports set A group of ports, treated as a logical unit.

message A collection of typed data objects used in communication between threads.7

memory object An object usually residing in secondary storage.

Mach (2)

The main outline of the message system is:

- Ports are data structures maintained by the kernel.
- Ports have a system-wide unique name.
- A port is not bound permanently to a single task.
- A port can have many senders but only one receiver.
- A task must have send rights to send to a port.
- A task must have receive rights to receive from a port.
- A task can have a number of ports at a time.
- Messages are of variable length.

Mach (3)

Ports can be on remote computers - works both within a single machine and across network. Use message passing to interact with the Kernel.

Message contains:

- destination port.
- reply port.
- size.
- operation.
- typed data.
- ports.
- pointer to data segments.

Mach (4)

Some system calls related to the message mechanism.

`port_alloc` create port and allocate buffer space.

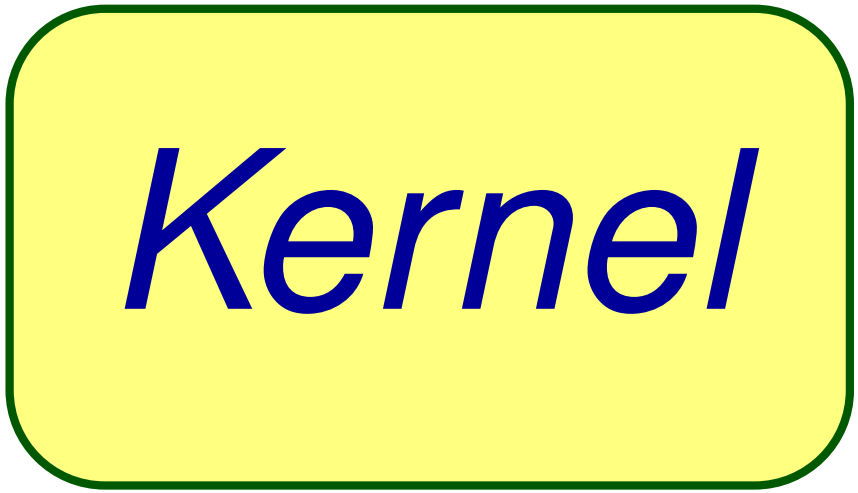
`msg_send` asynchronous send of message port. If mailbox is full, sender can:

a. wait indefinitely until there is room;
b. wait at most a number of milli-secs;
c. immediate return;
d. use temporary Kernel buffer (1 buffer per sending thread) - thread then continues.

`rcv_msg` blocking receive. Can be applied to a set of ports.

`rpc_port` sender blocks waiting for a reply on a reply port sent with the request.

(see also https://developer.apple.com/library/mac/#documentation/Darwin/Conceptual/KernelProgramming/Mach/Mach.html).


Kernel

Main Program in **OS**

```
void Kernel ()
{
   disable interrupts;
   initialise data structures;
   create initial processes;
   Schedule process;
   /*  Selects highest priority process, enables interrupts
    *  and runs it, at priority defined by its PSW */
TRAP:  /*  after interrupt or kernel call */
   save registers of current process;
   determine source of interrupt;
   service interrupt or call;
}
```

First Level
Interrupt Handler

Note: Servicing interrupt or call may change state of current or another process and result in a re-schedule: i.e. a different process will be chosen to run by scheduler.

Kernel Mode

Kernel entry switches processor into *privileged mode* for access to:

- kernel general purpose registers;
- Program Status Word;
- memory management registers;
- input / output instructions on some processors;
- enable and disable of interrupts;
- special instructions, like *halt*.

Note: some system processes may need to execute in privileged mode (e.g. device handlers).

Kernel Design

Kernel is the main component of **OS**; it forms a bridge between *applications* and the actual *data processing* done at the hardware level.

The kernel's responsibilities include managing the *system's resources* (the communication between hardware and software components); it provides the *lowest-level abstraction layer* for the resources (like processors and **I/O** devices) through inter-process communication mechanisms and system calls.

Requirements:

- kernel should contain minimal functionality;
- should be implementable in high level language, and therefore keep use of assembly language to minimum.

In some system the Input / Output layer is part of the kernel (e.g. Unix).

Kernel Functions

Process Management – manage process data;

- manage process context;
- stop, start, create, delete process.

First Level Interrupt Handling – provision of facilities for interrupt handling to device handlers.

Multiplex processor amongst processes – scheduler.

Provide process interaction mechanisms – semaphores and / or message passing.

Timer functions – delay for a period.

- time and date.
- accounting: run time measurements.

Interrupt handling

Kernel Entry

Kernel invoked as a result of:

External Interrupt from external device or clock.

Kernel Call from a running process:

- communication primitive, delay etc.

Internal Interrupt or error trap:

- divide by zero, memory protection violation;
- page fault;
- illegal or privileged instruction.

All these events result in an automatic switch on the **CPU**, and are dealt with by *interrupt handlers*. These are **OS**-dependent; interrupt handling itself (i.e. which handler to activate) is done by hardware.

Kernel Entry Mechanisms

Access procedures for each kernel operation available to processes that set up parameters needed for call and invoke kernel.

Procedure Calls Kernel resides in shared address space (or ROM). Used if no privileged mode.

- kernel procedures linked into process code;
- single entry point; parameter indicates operation.

Special Instruction – Supervisor Call (`SVC`), or trap;

- similar to hardware interrupt;
- register parameter determines operation.

Message Passing e.g. Mach, Minix Each process has default ports for sending and receiving messages to and from kernel. (Implemented via traps.)

Process Representation

Process Control Block (**PCB**) or Process Descriptor contains:

Context – program counter (**`PC`**);
- processor status word (**`PSW`**);
- stack pointer;
- registers (maybe on stack);
- memory addressing registers.

Process state – running / waiting / ready / sending / receiving.

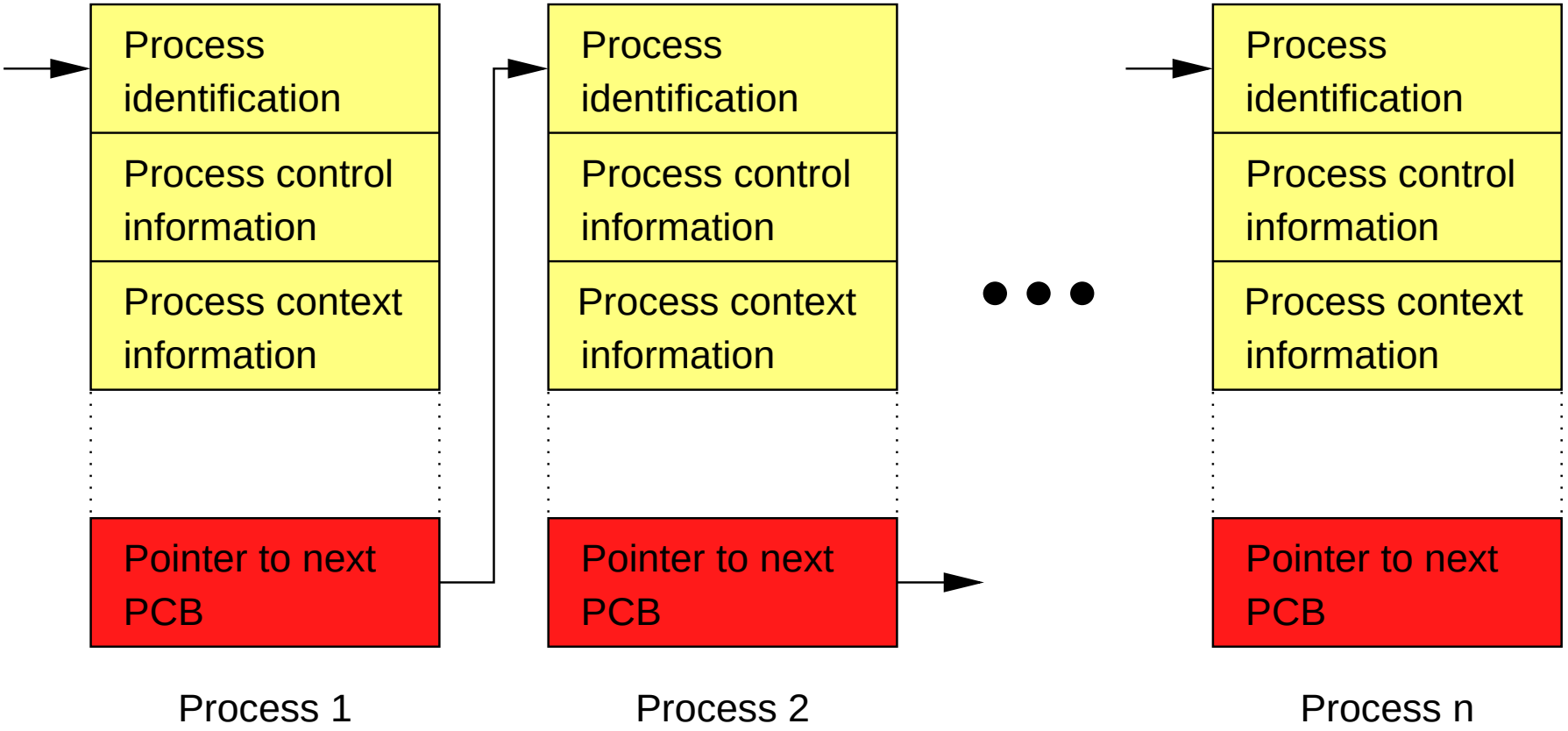
Priority

Process identifier

(Parent identifier)

Pointers to other PCB To represent various process queues.

Process Representation (2)



Process queues (ready Q, waiting Q, sending Q) are linked lists.

Process Representation (3)

Accounting information – start time;
- **CPU** ticks used.

Resources allocated – memory allocated (base, limit, page-, segment tables);
- root directory;
- open files;
- **I/O** devices.

Pointers for queues

- Context saved in **PCB** when process *stops* running.
- Context loaded into processor registers when process *starts* running.

Kernel maintains a number of arrays of **PCB**s.

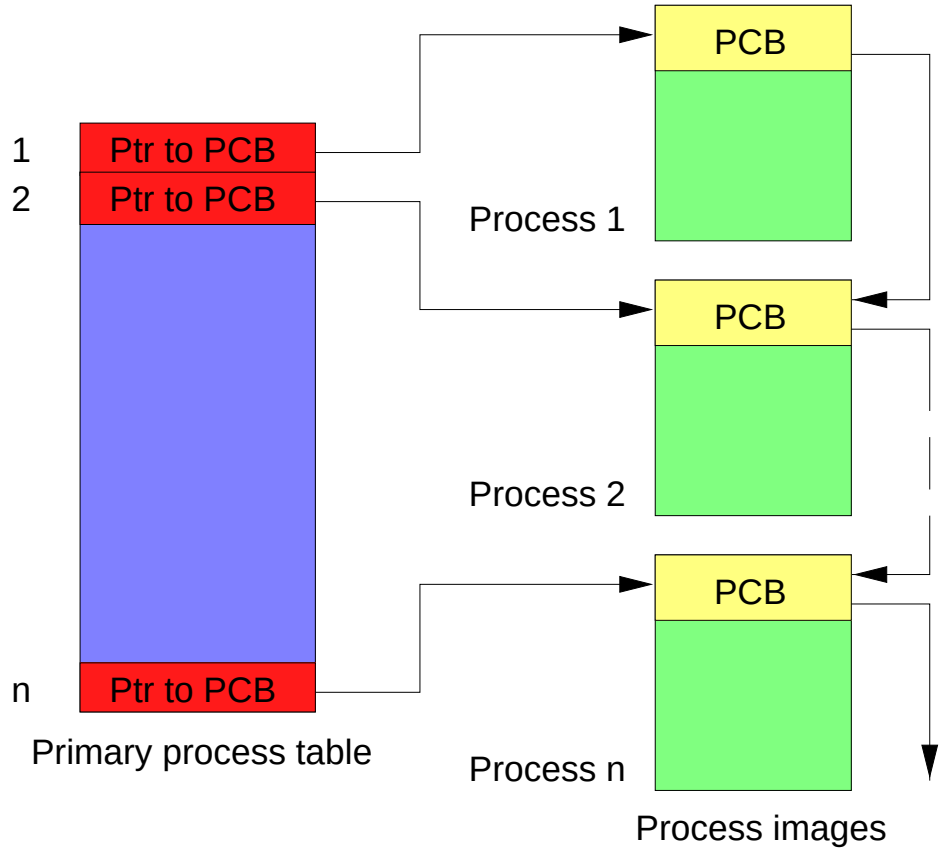
Process Data

- Address space, a list of memory locations which process can read or write.
  - the code / core image / executable file;
  - the data;
  - the stack;
  - the files it is using;
  - the processes it has created so far;
  - data stored for inter-process communication;
  - signals;
  - owner.
- Set of registers, including program counter (**PC**), stack pointer (**SP**), and other information needed to run the program.

Process Identification

- **ID** of process (**PID**):
  - **PID** is unique numerical identifier;
  - **PID** may be used as index to primary process table;
  - **PID** may be used to cross-reference process tables.
- **ID** of parent process (**PPID**);
- **ID** of owner of process (the user who created the process).

Organisation of **ID**s **OS** is dependent.



Process Context

Process context – if the process is being executed, the process context is in the **CPU**'s registers (it does *not* get updated while the process runs);

– if the process is being stopped, the process context is saved to resume execution later.

General purpose registers – data registers;

– pointer and index registers.

Control and status registers – instruction pointer;

– processor status word, i.e. overflow flag, sign flag, interrupt enable/disable flag, etc (hardware dependent).

Example Intel Pentium:

– general purpose registers: **`EAX`**, **`EBX`**, **`ECX`**, **`EDX`**, **`EBP`**, **`ESI`**, **`EDI`**, **`ESP`**;

– control and status registers: **`EIP`** and **`EFLAGS`**;

– segment registers: **`CS`**, **`DS`**, **`SS`**, **`ES`**, **`FS`**, **`GS`**.

Process Control Information

- Status information:
  - process state;
  - events and signals.
- Scheduling information:
  - scheduling priority (static and dynamic);
  - scheduling policy.
- Resource information:
  - open files;
  - open directories;
  - dynamically allocated memory;
  - **I/O** devices.

Also this is **OS** dependent.

Process Management Operations

Process Creation Data needed:

- process name;
- code + start address;
- data + stack.

Kernel creates a **PCB**, assigns memory, makes process runnable.

Process Deletion – stop process;

- clean up - outstanding **I/O**, messages etc;
- recover resources - memory;
- erase **PCB**.

Process control – halt process - make un-runnable;

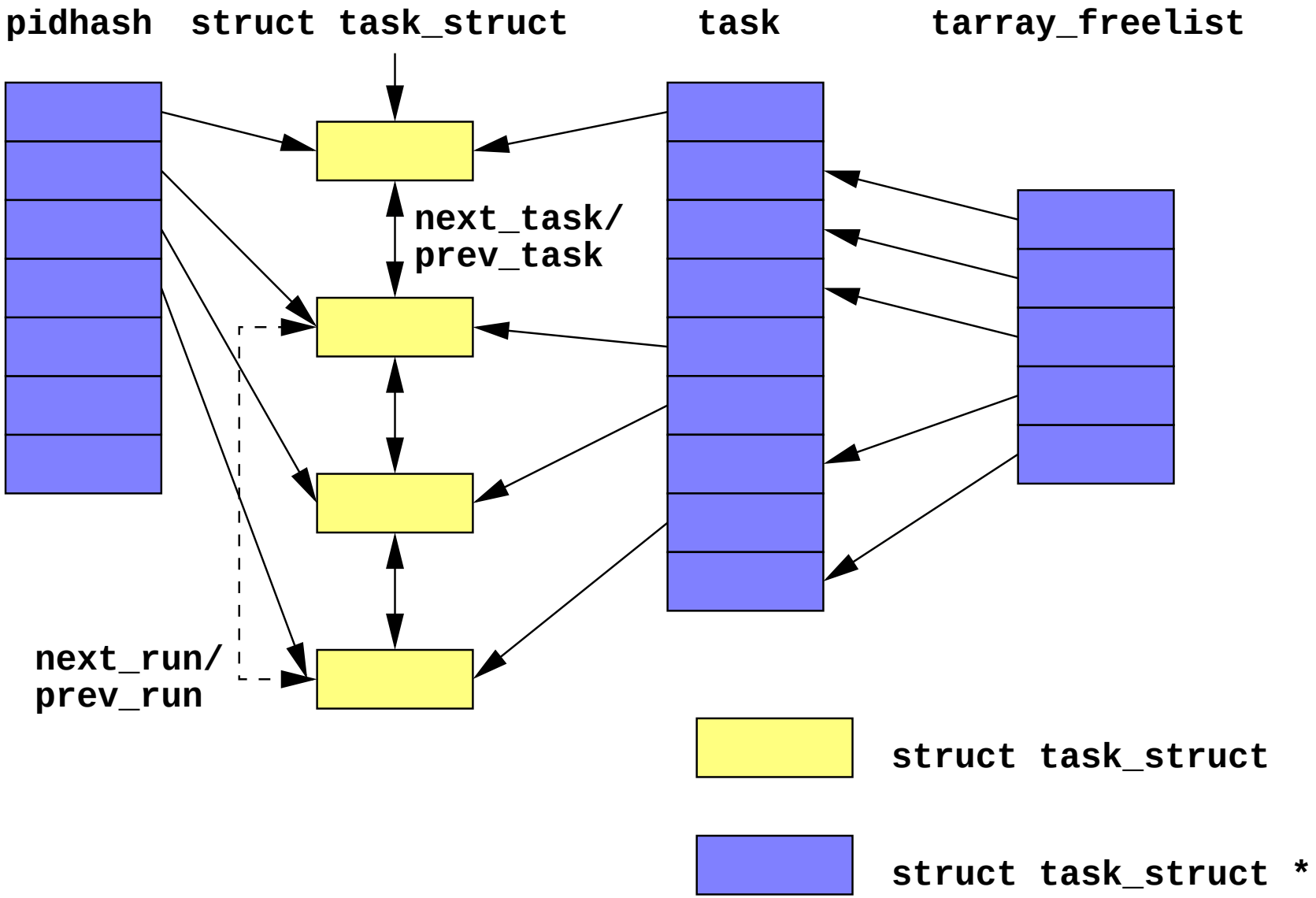
- continue process - make runnable.

Change Priority application dependent scheduling.

Linux: Process Description

- Linux represents each process as a **`struct task`**;
- Linux has two circular lists:
  - **`task list`** is a list of all tasks;
  - **`run list`** is a list of all tasks ready to run.
- Every task is represented by an entry into the array **`task`**:
  - The array size is fixed and limits the maximum number of tasks;
  - The array index does not correspond to the process **ID**.
- A list of free entries in the array **`task`** is maintained by the linked list **`tarray_freelist`**.
- A hash table **`pidhash`** allows the conversion of process **ID**s to a pointer to the corresponding **`struct task`**.

Linux: Process Description (2)



Minix Process Descriptor (in `kernel/proc.h`)

```
struct proc  {
   struct stackframe_s p_reg;   /* process' registers saved in stack frame   */
#if (CHIP == INTEL)
   reg_t p_ldt_sel;             /* selector in  gdt with  ldt base and limit  *
   struct segdesc_s p_ldt[2+NR_REMOTE_SEGS];
                                /*   CS,  DS and remote segments   */
#endif
#if (CHIP == M68000)
    /* M68000 specific registers and FPU details go here.     */
#endif
   proc_nr_t p_nr;              /* number of this process (for fast access)   */
   struct priv *p_priv;         /* system privileges structure   */
   char p_rts_flags;            /*   SENDING,  RECEIVING, etc.     */
   char p_misc_flags;           /* Flags that do suspend the process   */
   char p_priority;             /* current scheduling priority   */
   char p_max_priority;         /* maximum scheduling priority   */
   char p_ticks_left;           /* number of scheduling ticks left   */
   char p_quantum_size;         /* quantum size in ticks   */
          :
```

Minix Process Descriptor (2)

```
   struct mem_map p_memmap[NR_LOCAL_SEGS];
                                    /* memory map ( T, D, S) */
   clock_t p_user_time;             /* user time in ticks */
   clock_t p_sys_time;              /* sys time in ticks */
   struct proc *p_nextready;        /* pointer to next ready process */
   struct proc *p_caller_q;         /* head of list of procs wishing to send */
   struct proc *p_q_link;           /* link to next proc wishing to send */
   message *p_messbuf;              /* pointer to passed message buffer */
   proc_nr_t p_getfrom;             /* from whom does process want to receive? */
   proc_nr_t p_sendto;              /* to whom does process want to send? */
   sigset_t p_pending;              /* bit map for pending kernel signals */
   char p_name[P_NAME_LEN];         /* name of the process, including 0 */
#if DEBUG_SCHED_CHECK
   int p_ready, p_found;
#endif
 }
```

Minix Process Organisation (kernel/proc.h)

```
/* Bits for the runtime flags. A process is runnable iff p_rts_flags == 0. */
#define P_SLOT_FREE     001     /* set when slot is not in use */
#define NO_MAP          002     /* keeps unmapped forked child from running */
#define SENDING         004     /* set when process blocked trying to send */
#define RECEIVING       010     /* set when process blocked trying to receive */
#define SIGNALED        0x10    /* set when new kernel signal arrives */
#define SIG_PENDING     0x20    /* unready while signal being processed */
#define P_STOP          0x40    /* set when process is being traced */
#define NO_PRIV         0x80    /* keep forked system process from running */

/* Misc flags */
#define MF_VM 0x01              /* Process uses VM */

/* Scheduling priorities for p_priority. Values must start at zero (highest
 * priority) and increment. Priorities of the processes in the boot image
 * can be set in table.c. IDLE must have a queue for itself, to prevent low
 * priority user processes to run round-robin with IDLE. */
#define NR_SCHED_QUEUES 16      /* MUST equal minimum priority + 1 */
#define TASK_Q          0       /* highest, used for kernel tasks */
#define MAX_USER_Q      0       /* highest priority for user processes */
#define USER_Q          7       /* default (should correspond to nice 0) */
#define MIN_USER_Q      14      /* minimum priority for user processes */
#define IDLE_Q          15      /* lowest, only IDLE process goes here */
```

Minix Process Organisation (2)

```
/* Magic process table addresses.    */
#define BEG_PROC_ADDR   (&proc[0])
#define BEG_USER_ADDR   (&proc[NR_TASKS])
#define END_PROC_ADDR   (&proc[NR_TASKS + NR_PROCS])

#define NIL_PROC        ((struct proc *) 0)
#define NIL_SYS_PROC    ((struct proc *) 1)
#define cproc_addr(n)   (&(proc + NR_TASKS)[(n)])
#define proc_addr(n)    (pproc_addr + NR_TASKS)[(n)]
#define proc_nr(p)      ((p)->p_nr)

#define isokprocn(n)    ((unsigned)((n) + NR_TASKS) <
                             NR_PROCS + NR_TASKS)
#define isemptyn(n)     isemptyp(proc_addr(n))
#define isemptyp(p)     ((p)->p_rts_flags == SLOT_FREE)
#define iskernelp(p)    iskerneln((p)->p_nr)
#define iskerneln(n)    ((n) < 0)
#define isuserp(p)      isusern((p)->p_nr)
#define isusern(n)      ((n) >= 0)
```

Minix Process Descriptor Organisation (3)

```
/* The process table and pointers to process table slots. The pointers allow
 * faster access because now a process entry can be found by indexing the
 *  pproc_addr array, while accessing an element  i requires a multiplication
 * with  sizeof(struct proc) to determine the address.    */

EXTERN struct proc proc[NR_TASKS + NR_PROCS];
                            /* process table  */
EXTERN struct proc *pproc_addr[NR_TASKS + NR_PROCS];
EXTERN struct proc *rdy_head[NR_SCHED_QUEUES];
                            /* ptrs to ready list headers  */
EXTERN struct proc *rdy_tail[NR_SCHED_QUEUES];
                            /* ptrs to ready list tails  */
```

Unix Process Creation

- System call **`fork`** creates a (child) process identical to calling process (parent), by allocating a new **PCB**, and filling it with the parent's **PCB**: they are identical, but for the **PID** and the **PPID**.

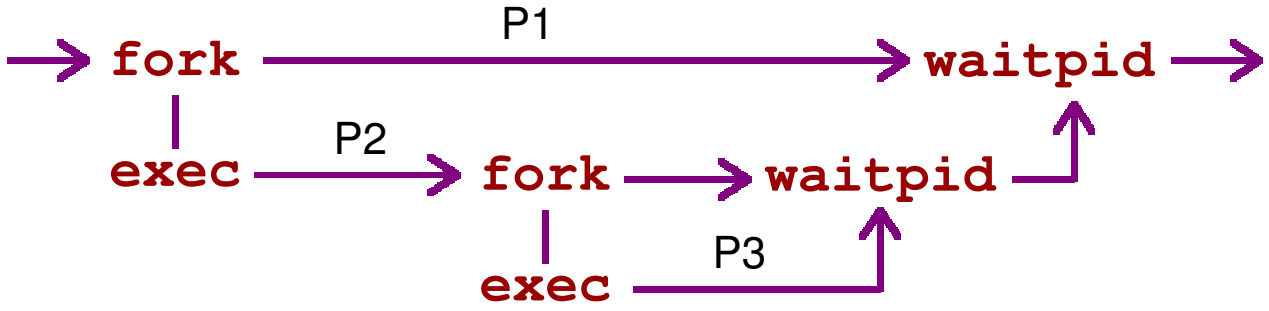
  The child process will inherit the resources of the parent process and will be executed concurrently with the parent process.

  To create a different process the child must call **`exec`**(ute), which will replace the code and reset the registers. For process synchronisation, to join, a process can call **`waitpid`**.
  - For the parent process **`fork`** returns the **PID** of the child;
  - For the child process **`fork`** returns **`0`**.
- System call **`exit`** terminates a process implicitly or explicitly:
  - it returns an exit status to the parent process;
  - it does not yet free the resources of the process.
- Parent and child processes are executing concurrently.

Use of `fork` *and* `exec`

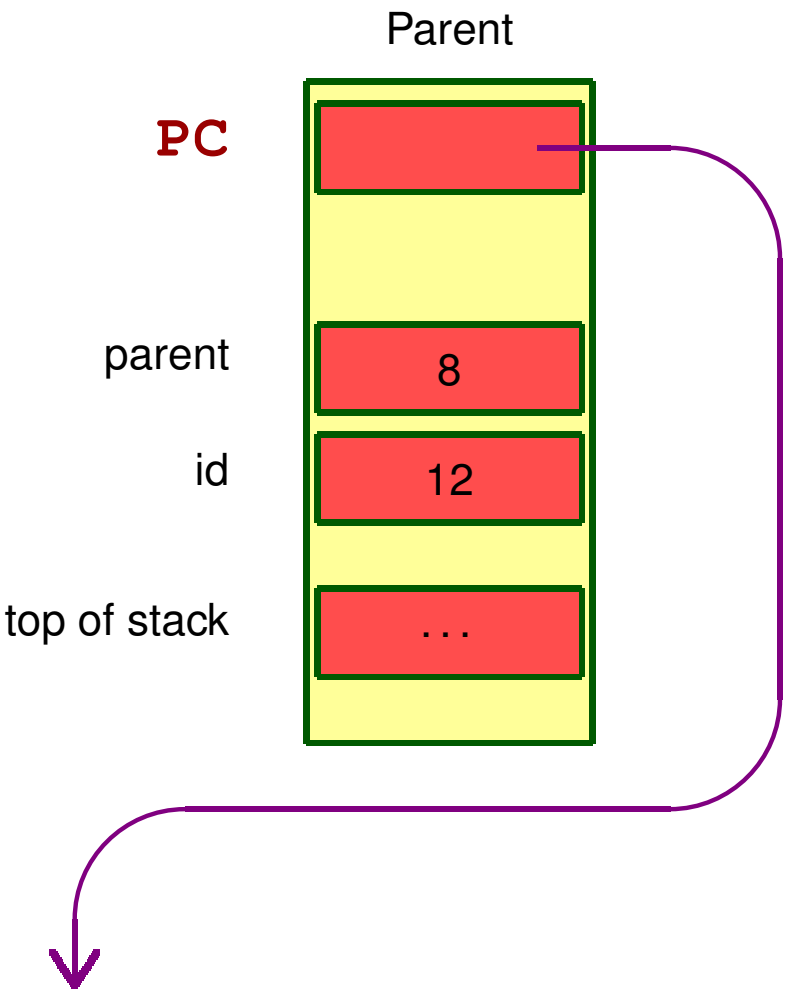
A command interpreter (shell) could do:

```
while (TRUE)  {                          /* repeat forever */
  read_command (command,parameters);
  if (fork () != 0)                      /* fork off child process */
    waitpid(-1, &status, 0);             /* Parent code */
  else                                   /* Child code */
    execve (command, parameters,0);
                                         /* execute command */
}
```



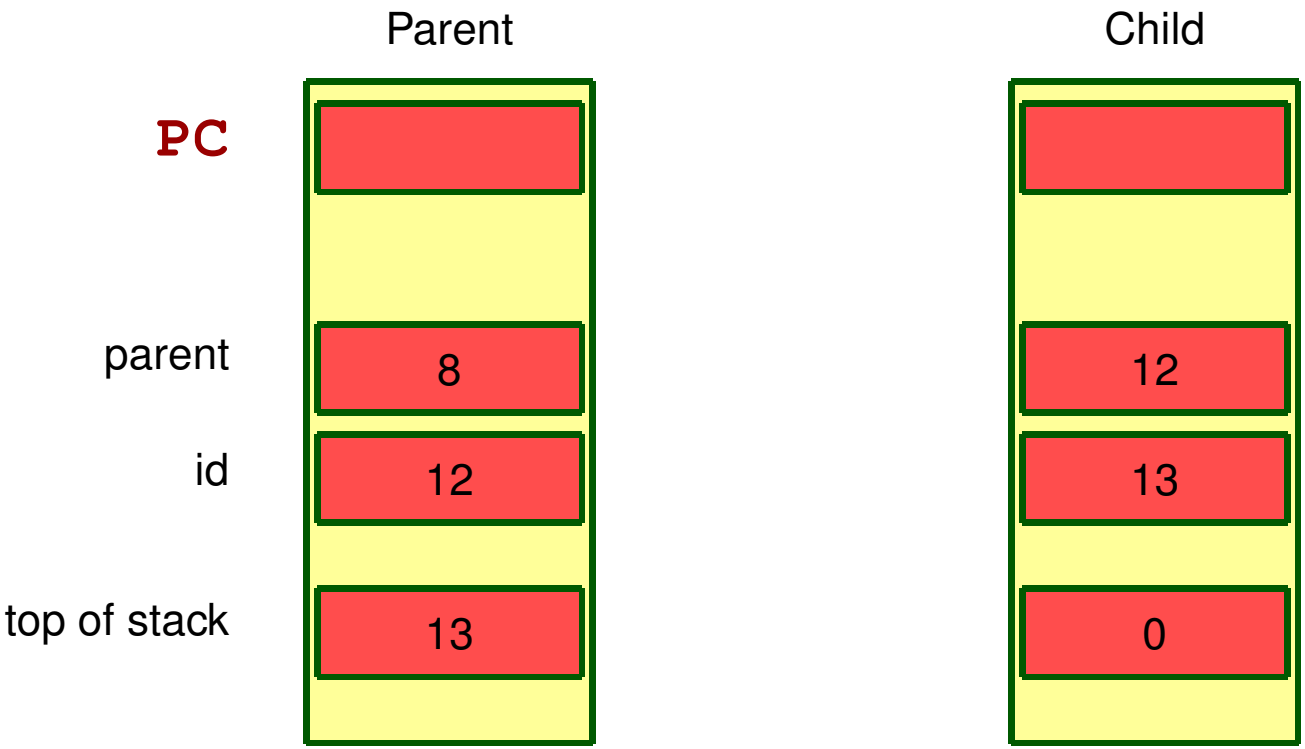
When a process dies, the system checks if a (parent) process is waiting on `waitpid`.

fork



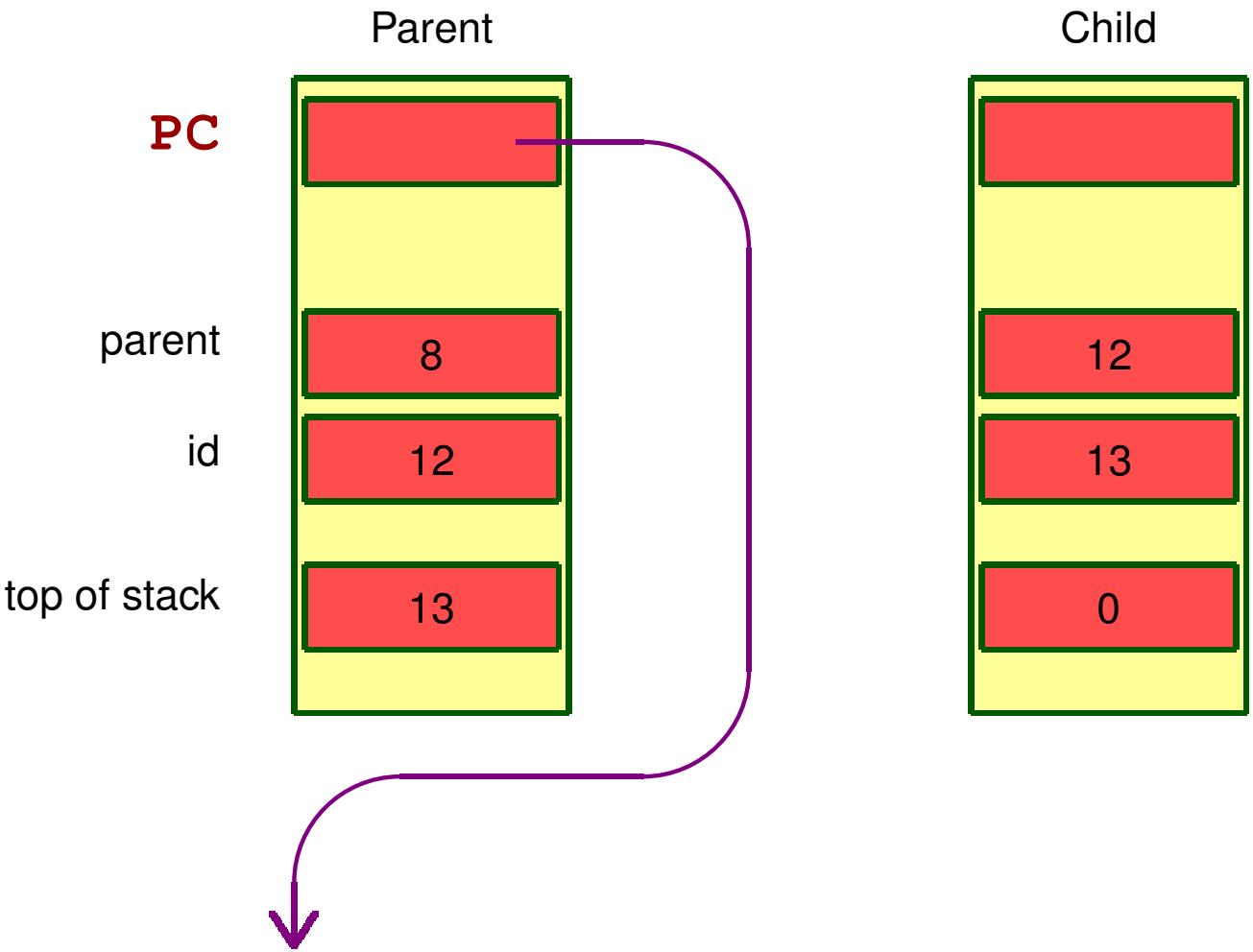

```
if fork() != 0 { parent code } else { child code }
```

fork



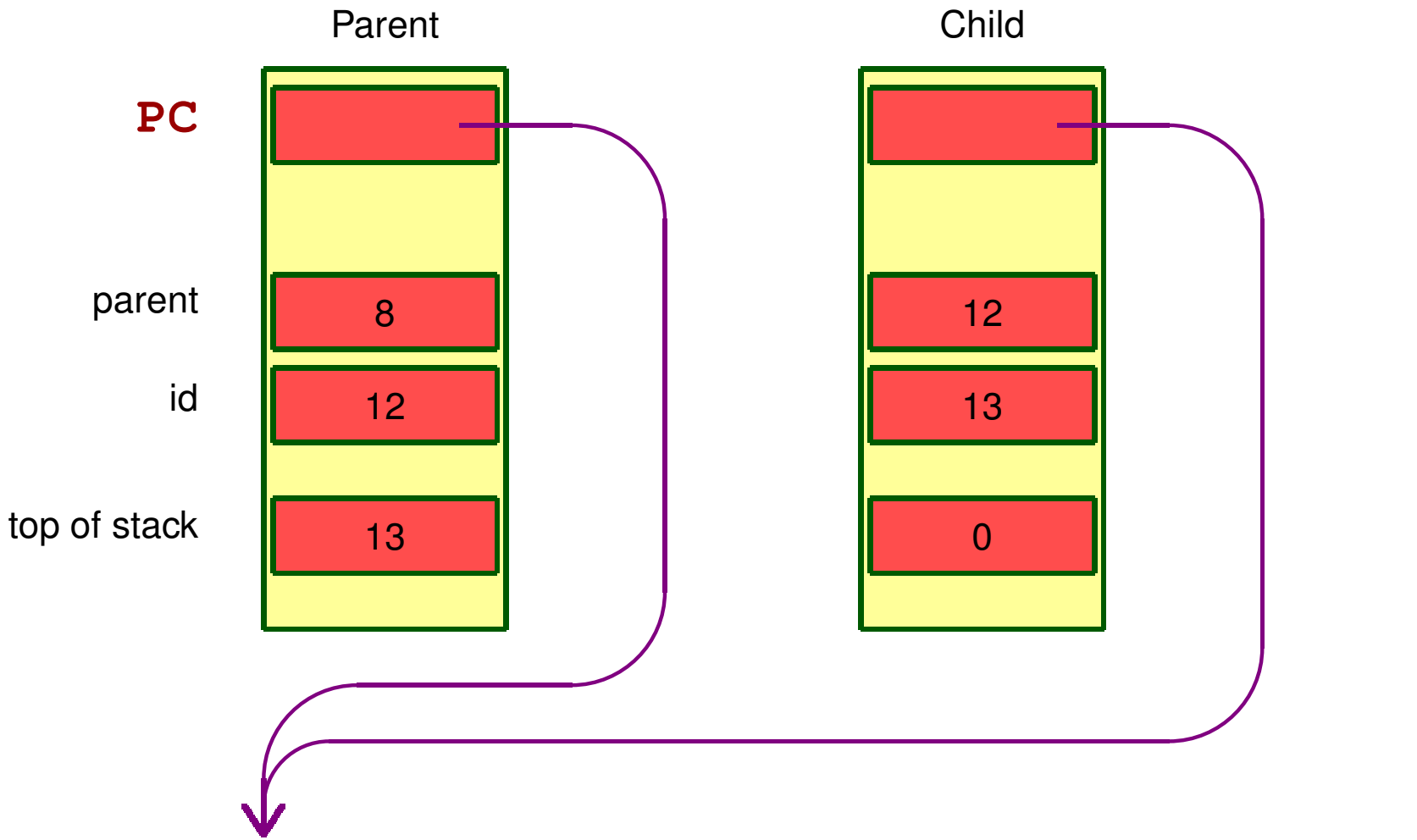

```
if fork() != 0 { parent code } else { child code }
```

fork



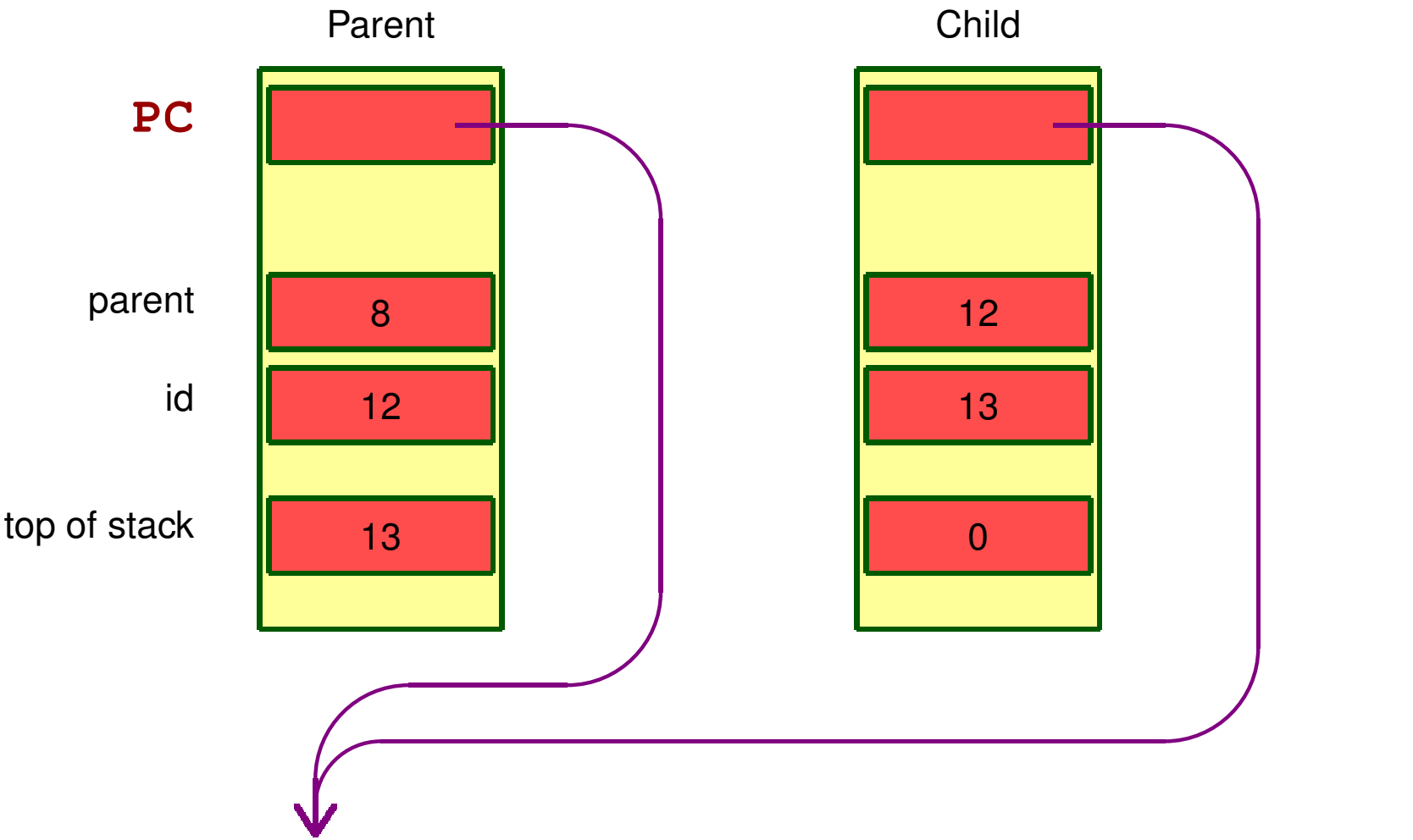

```
if fork() != 0 { parent code } else { child code }
```

fork



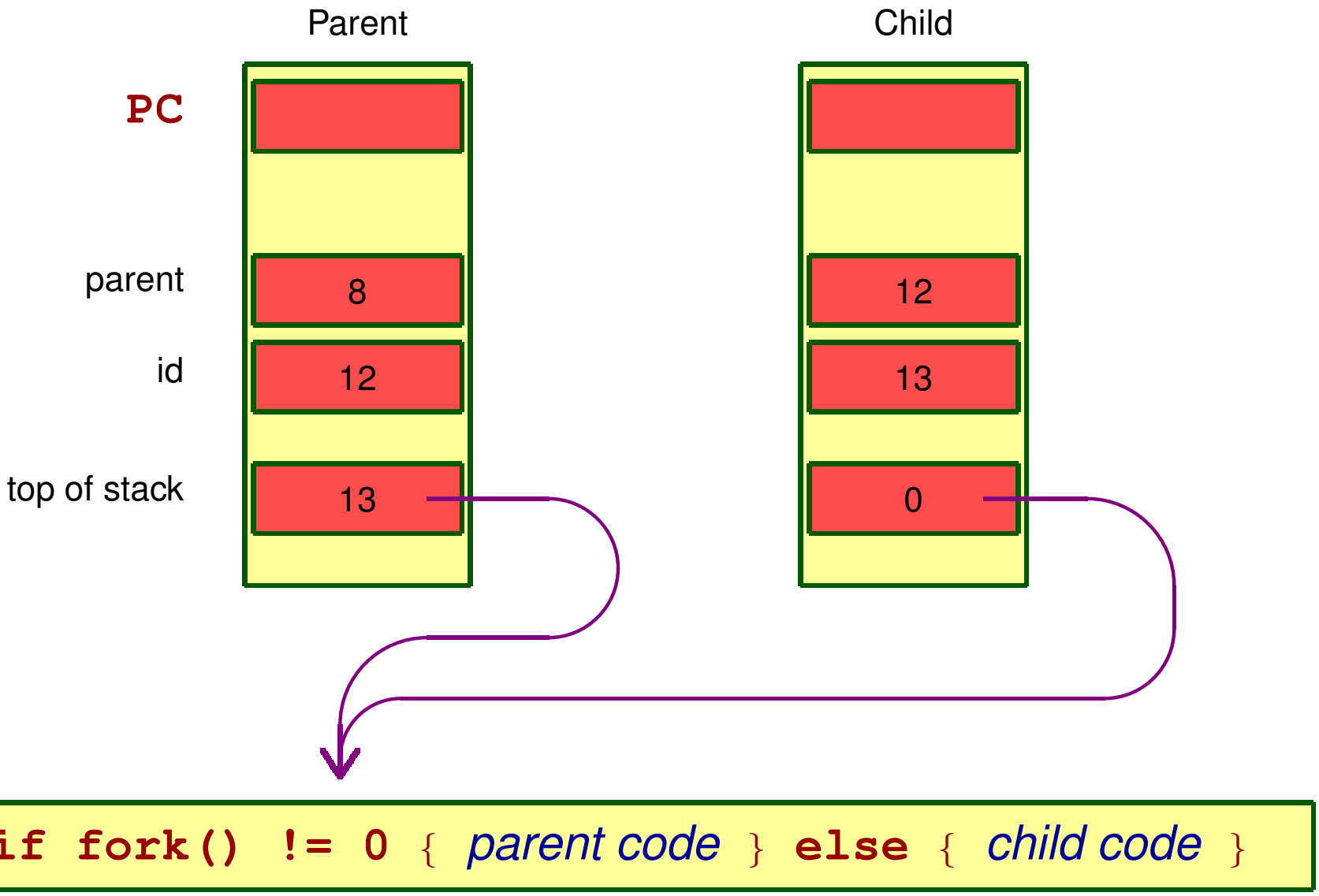

```
if fork() != 0 { parent code } else { child code }
```

fork



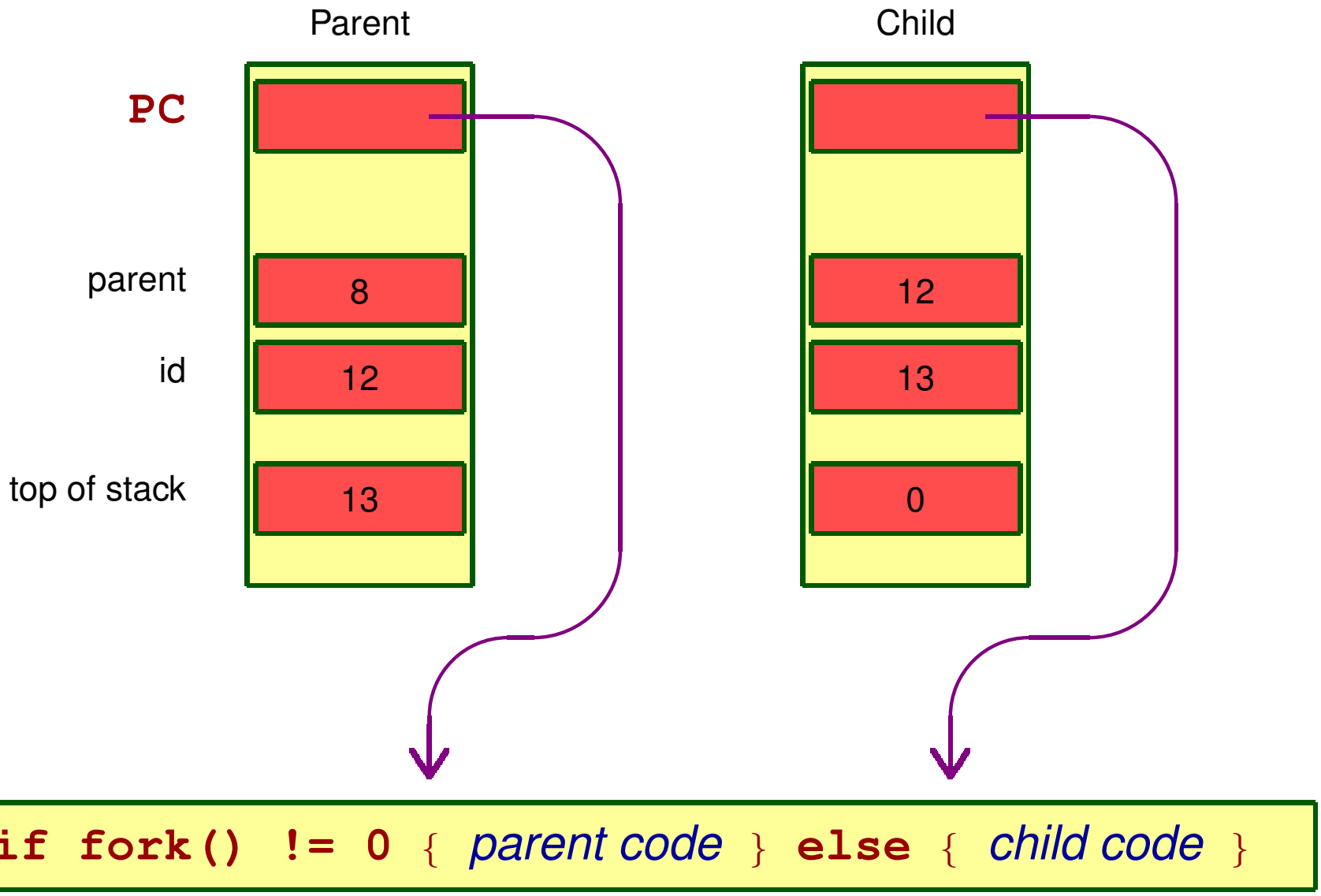

```
if fork() != 0 { parent code } else { child code }
```

fork




```
if fork() != 0 { parent code } else { child code }
```

fork


Parent
Child
PC
parent
id
top of stack
8
12
13
12
13
0
if fork() != 0 { parent code } else { child code }

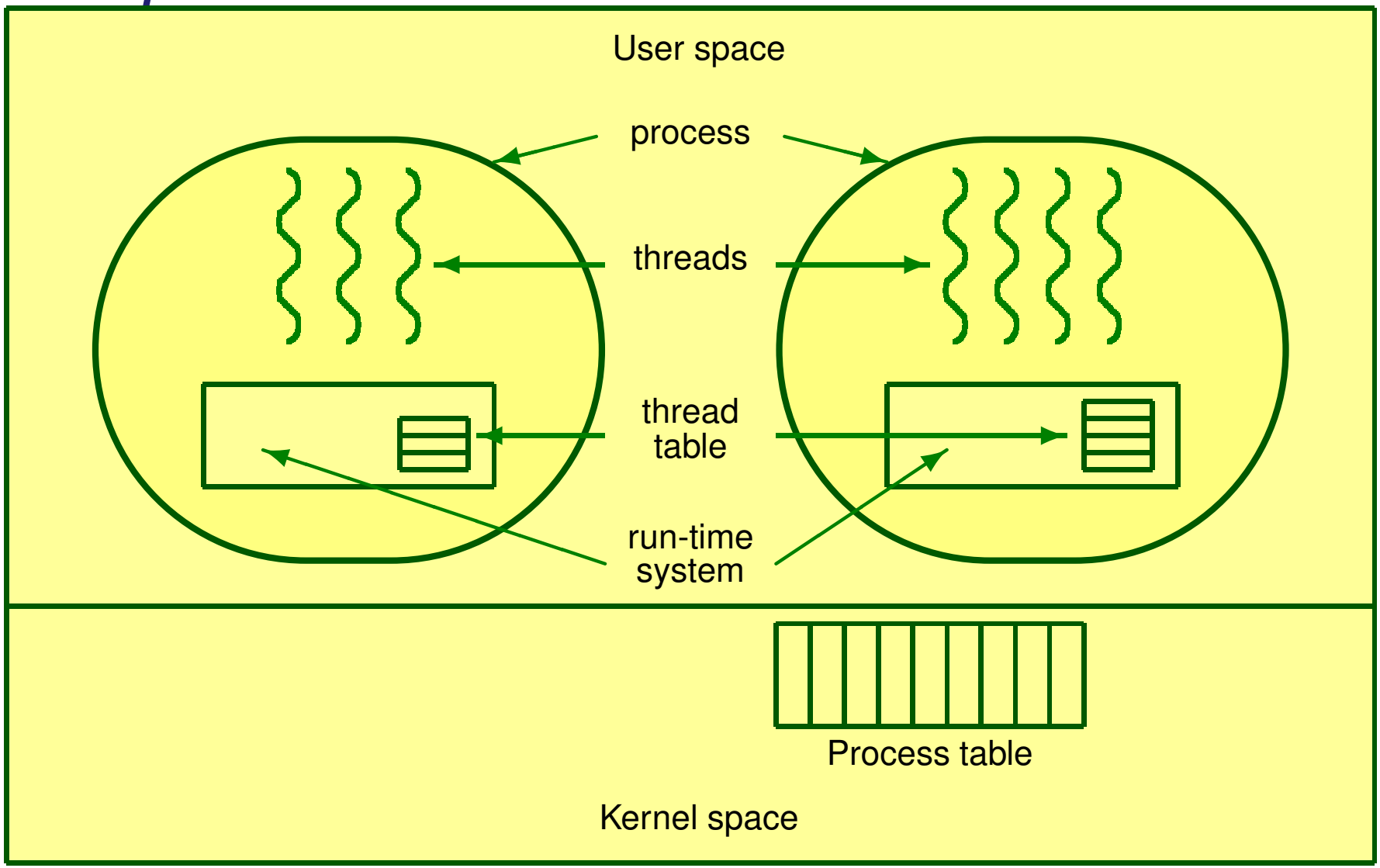
Threads

Threads are a kind of lightweight process, and can exist within a context with full-scale processes. It is a unit of **CPU** utilisation, and is identified by the **PC**, registers, and stack.

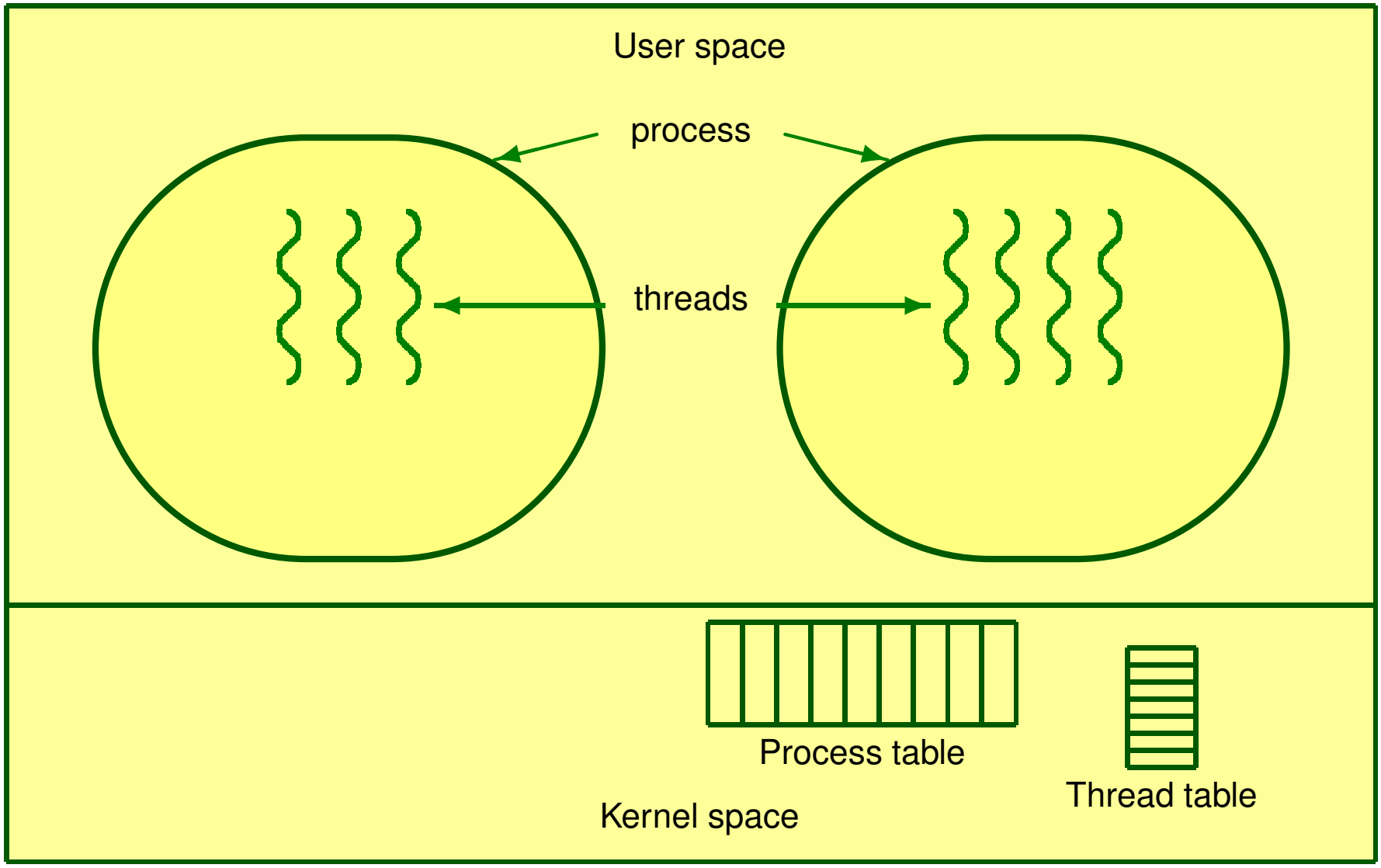
Protection is not supplied by the kernel and threads may share code and data with other threads in process. They are light, because the switching overhead is minimal (switching discussed later).

On systems that recognise threads, switching is performed by the kernel; otherwise, scheduling code is added to the process that contains the threads.

Implementation Threads: in Process



Implementation Threads: in **OS**



System Calls and I/O

System Calls

A means of having access to objects that **OS** manages. On machine with special hardware protection, these instructions are the only way to manipulate those objects.

We distinguish:

Run-time Library Collection of (compiled) *procedures* that can be linked-in to a `.o` file.

Library Routine Procedures that are standard, and could be written by the user directly; can be seen as abbreviations (like `atoi`).

System Call Requests for special services to the **OS**, like access to a privileged resource, normally not available to the user; typically asking for **I/O** (like `printf`).

Process System Calls

load process into memory.
execute process; replace core image.
stop / **resume** process.
create process; make entry in process table.
get / **set** process attributes.
send / **receive** message.
allocate memory before loading.
free memory after killing process.
getpid get process identifier (number).
getpgrp get group id.

Signals

wait for a fixed amount of time, for an event to occur, or getting a signal of an event, like the death of another process.
pause suspend the process until the next signal.
sigaction define action to take on signals.
sigpending suspend process.
kill process; stop its execution, and remove entry.

Files and Devices

File Manipulation `create` / `delete` file in/from secondary memory.

- `open` a file, give access to its contents.
- `close` a file.
- `read` / `write` from an opened file.
- `re-position` within a file.
- `get` / `set` file attributes.

Device Manipulation `request` / `release` device (c.f. `open`, `close`).

- `read` / `write` / `control` (e.g. re-position, reset etc).
- `get` / `set` device attributes.

Maintenance

Information maintenance – **`get`** / **`set`** date.

- **`get`** / **`set`** system data.
- **`get`** process attributes (e.g. **ID**, run-time, size).
- **`set`** process attributes (e.g. priority).

Communication maintenance – **`open`** / **`close`** communication connection.

- **`send`** / **`receive`** message.
- **`get`** connection status.

System Calls in Minix

Caller

1 Put message pointer and destination of message into **CPU** registers.

2 Execute software interrupt mechanism.

Kernel

1 Save registers.

2 Send and / or receive message.

3 Restart a process (not necessarily calling one).

The Procedure `sys_call` *in Minix*

Each system-call in Minix eventually calls `sys_call`.

```
PUBLIC int sys_call(call_nr, src_dst, m_ptr)
int call_nr;                    /* system call number and flags  */
int src_dst;                    /* src to receive from or dst to send to  */
message *m_ptr;                 /* pointer to message in the caller's space  */
 {
 /* System calls are done by trapping to the kernel with an INT instruction.
  * The trap is caught and sys_call() is called to send or receive a message
  * (or both). The caller is always given by 'proc_ptr'.    */
  register struct proc *caller_ptr = proc_ptr;  /* get pointer to caller
  int function = call_nr & SYSCALL_FUNC;  /* get system call function  */
  unsigned flags = call_nr & SYSCALL_FLAGS;  /* get flags  */
  int mask_entry;               /* bit to check in send mask  */
  int group_size;               /* used for deadlock check  */
  int result;                   /* the system call's result  */
  vir_clicks vlo, vhi;          /* virtual clicks containing message to send  */

    /* Check if the process has privileges for the requested call. Calls to the
     * kernel may only be SENDREC, because tasks always reply and may not block
     * if the caller doesn't do receive().    */
```

The Procedure `sys_call` in Minix (2)

```
if (!(priv(caller_ptr)->s_trap_mask & (1 << function)) ||
  (iskerneln(src_dst) && function != SENDREC
    && function != RECEIVE))  {
  return(ETRAPDENIED);                  /* trap denied by mask or kernel  */
 }
 /* Require a valid source and/ or destination process, unless echoing.    */
if (src_dst != ANY && function != ECHO)  {
  if (!isokprocn(src_dst) || isemptyn(src_dst))  {
    return(EDEADSRCDST);  }
   }
  .
  .
 /* If the call is to send to a process, i.e., for SEND, SENDREC or NOTIFY,
  * verify that the caller is allowed to send to the given destination.    */
if (function & CHECK_DST)  {
  if (! get_sys_bit(priv(caller_ptr)->s_i PC_to, nr_to_id(src_dst
    return(ECALLDENIED);                /* call denied by PC mask  */
   }
 }
 /* Now check if the call is known and try to perform the request. The only
  * system calls that exist in MINIX are sending and receiving messages.
```

The Procedure `sys_call` in Minix (3)

```
 * - SENDREC: combines SEND and RECEIVE in a single system call
 * - SEND: sender blocks until its message has been delivered
 * - RECEIVE: receiver blocks until an acceptable message has arrived */
switch(function) {
   case SENDREC:    /* A flag is set so that notifications cannot interrupt SENDREC.
      priv(caller_ptr)->s_flags |= SENDREC_BUSY;  /* fall through */
   case SEND:
      result = mini_send(caller_ptr, src_dst, m_ptr, flags);
      if (function == SEND || result != OK) {
         break;                                  /* done, or SEND failed */
       }                                         /* fall through for SENDREC */
   case RECEIVE:
      if (function == RECEIVE)
         priv(caller_ptr)->s_flags &= SENDREC_BUSY;
      result = mini_receive(caller_ptr, src_dst, m_ptr, flags);
      break;
   default:
      result = EBADCALL;                         /* illegal system call */
    }  /* Now, return the result of the system call to the caller.    */
   return(result);
}
```

Example: the **`read`** *system call*

First: **`lib/posix/_read.c`**:

```
#include <lib.h>
#define read _read
#include <unistd.h>

PUBLIC ssize_t read(fd, buffer, nbytes)
int fd;
void *buffer;
size_t nbytes;
 {
   message m;

   m.m1_i1 = fd;
   m.m1_i2 = nbytes;
   m.m1_p1 = (char *) buffer;
   return(_syscall(FS, READ, &m));
 }
```

read (2)

Second: lib/other/syscall.c:

```
#include <lib.h>

PUBLIC int _syscall(who, syscallnr, msgptr)
int who, syscallnr;
register message *msgptr;
 {
   int status;

   msgptr->m_type = syscallnr;
   status = _sendrec(who, msgptr);
   if (status != 0)  {
       /* 'sendrec' itself failed.    */
      msgptr->m_type = status;
    }
   if (msgptr->m_type < 0)  {
      errno = -msgptr->m_type;
      return(-1);
    }
   return(msgptr->m_type);
 }
```

read (3)

Third: in src/lib/i386/rts/_iPC.s:

```
SEND = 1
RECEIVE = 2
SENDREC = 3

SYSVEC = 33

! all message passing routines save ebp, but destroy eax and ecx.
__sendrec:
        push    ebp
        mov     ebp, esp
        push    ebx
        mov     eax, SRC_DST(ebp)    ! eax = dest-src
        mov     ebx, MESSAGE(ebp)    ! ebx = message pointer
        mov     ecx, SENDREC         ! _sendrec(srcdest, ptr)
        int     SYSVEC               ! trap to the kernel
        pop     ebx
        pop     ebp
        ret
```

read (4)

Fourth: src/kernel/i8259.c:

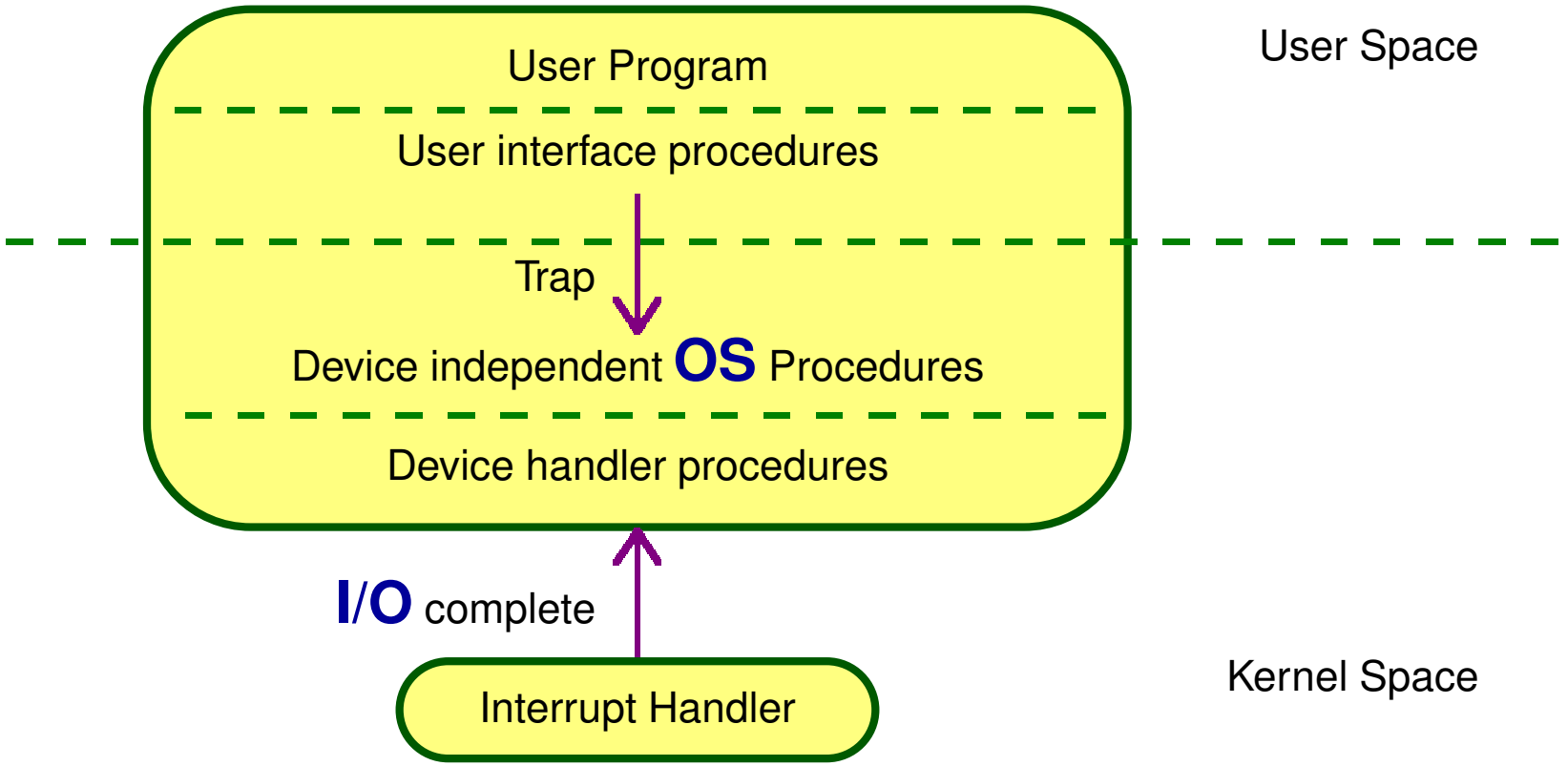
```
        set_vec(SYS_VECTOR, s_call);      /*  SYS_VECTOR = 33 */
```

Fifth: src/kernel/mpx386.s:

```
_s_call:
_p_s_call:
   :
   :                ! now set up parameters for sys_call()
        push ebx          ! pointer to user message
        push eax          ! src/dest
        push ecx          ! SEND/RECEIVE/BOTH
        call _sys_call    ! sys_call(function, src_dest, m_ptr)
                          ! caller is now explicitly in proc_ptr
        mov AXREG(esi), eax ! sys_call MUST PRESERVE si
! Fall into code to restart proc/task running.
_restart:
        ! Restart the current process or the next process ...
```

So read eventually results in calling sys_call ...

Unix: Single Monolithic Process



Kernel and User mode are different parts of same process. Interface between **I/O** components not very clean.

More efficient, as procedure calls do not involve context switch for communication between **I/O** components.

Device Management & Input / Output

Objectives – Provide uniform simple view of **I/O**: hide complexity of device handling, give uniform naming and error handling;

- Device independence from:
  - device type (e.g. terminal, disk file or magnetic tape);
  - device instance (e.g. which terminal).
- Fair access to shared devices;
- Allocation of dedicated devices;
- Exploit parallelism of **I/O** devices for multiprogramming.

Device variations – Speed differences;

- Synchronous or asynchronous operation;
- Unit of data transfer (character or block);
- Character Codes (translate to internal code if necessary);
- Operations supported (e.g. **`read`**, **`write`**, **`seek`**);
- Error conditions;
- Sharable (disks, files), or single user (printer, CD-ROM).

I/O *Layers*

User Interface – **OS I/O** library procedures to set up parameters (device independent);

- Access virtual devices as files (Unix), streams (Dec 10), data sets (IBM);
- Operations - **`open`**, **`close`**, **`read`**, **`write`**, **`seek`**;
- Synchronous or asynchronous.

Device Independent Layer – Implements device independence;

- Allocation of dedicated devices;
- Protection - user access validation;
- Map virtual device to real device (naming and switching);
- Validate requests against device characteristics;
- Buffering for performance and block size independence;
- Error reporting.

I/O *Layers (2)*

Device Handler – Handles *one* device type (*major*), but may control multiple devices of the same type (*minor*);

- Implements block read or write;
- Access device registers;
- Initiate operations;
- Schedule requests;
- Handle errors.

Interrupt Handler – Process each interrupt;

- Block devices: on transfer completion, signal device handler;
- Character devices: when character transferred, process next character.

Devices

Device	Data rate	Type	Operation
Clock	7.5 bytes/sec	???	???
Keyboard	10 bytes/sec	char	r
Mouse	100 bytes/sec	char	r
56k Modem	7 KB/sec	char	r, w
Telephone channel	8 KB/sec	char	r
Dual ISDN	16 KB/sec	char	r, w
Laser Printer	100 KB/sec	char	w
Scanner	400 KB/sec	char	r
Classic Ethernet	1.25 MB/sec	char	r, w
USB (universal serial bus)	1.5 MB/sec	char	r, w
Digital Camcorder	4 MB/sec	char	r, w

Devices (2)

IDE disk	5 MB/sec	block	r, w, s
40x CD-ROM	6 MB/sec	block	r, s
Fast Ethernet	12.5 MB/sec	char	r, w
ISA bus	16.7 MB/sec	char	r, w
EIDE (ATA-2) disk	16.7 MB/sec	block	r, w, s
FireWire (IEEE 1394)	50 MB/sec	char	r, w
XGA Monitor	60 MB/sec	char	r, w, s
SONET OC-12 network	78 MB/sec	char	r, w
SCSI Ultra 2 disk	80 MB/sec	block	r, w, s
GigaBit Ethernet	125 MB/sec	char	r, w
PCI Bus	528 MB/sec	char	r, w
Sun Gigaplane XB backplane	20GB/sec	char	r, w

Terminal

- Terminal consists of: keyboard & display.
- Character device which supports reading/writing (38400 bps).
- Each keystroke produces an interrupt; interrupt handler delivers input character to device handler.
- Handler handles character depending of terminal settings:

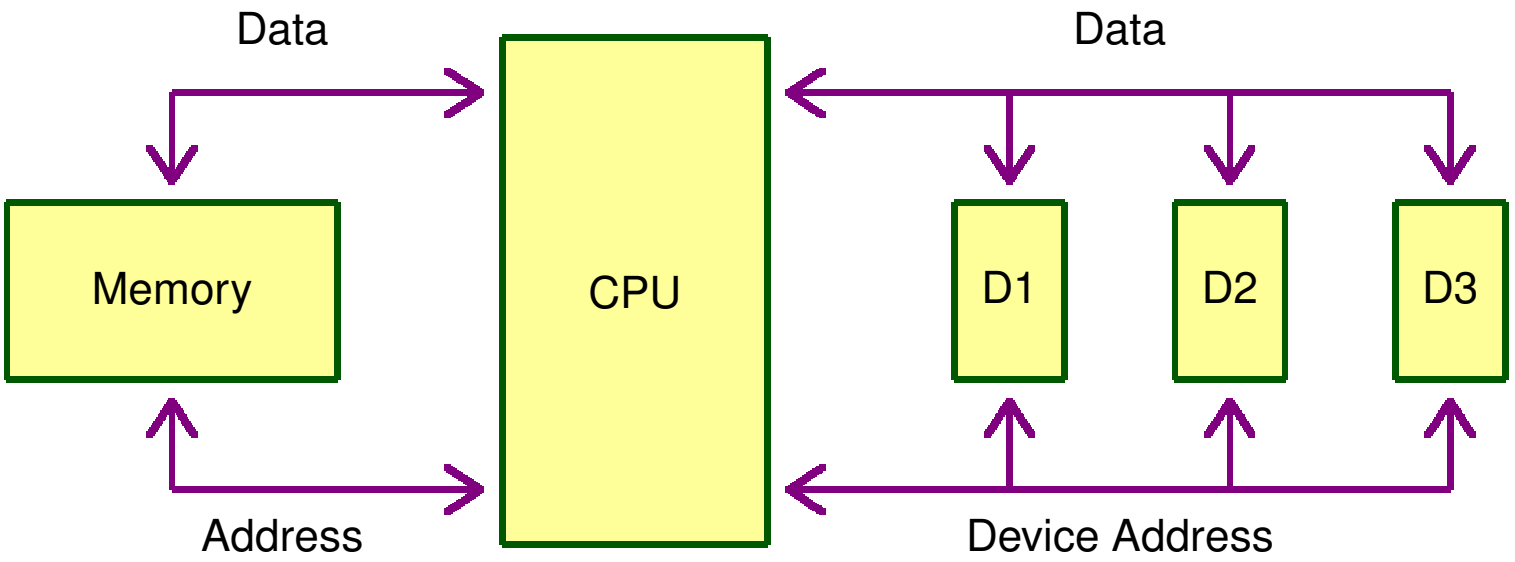
  Canonical mode (or cooked mode) characters are delivered to the device independent layer after handling all intraline editing: line-oriented, useful for normal applications;

  Non-canonical mode (or raw mode) characters are delivered to the device independent layer immediately: character-oriented, useful for applications like editors;

  – manages output on terminal (echoing).

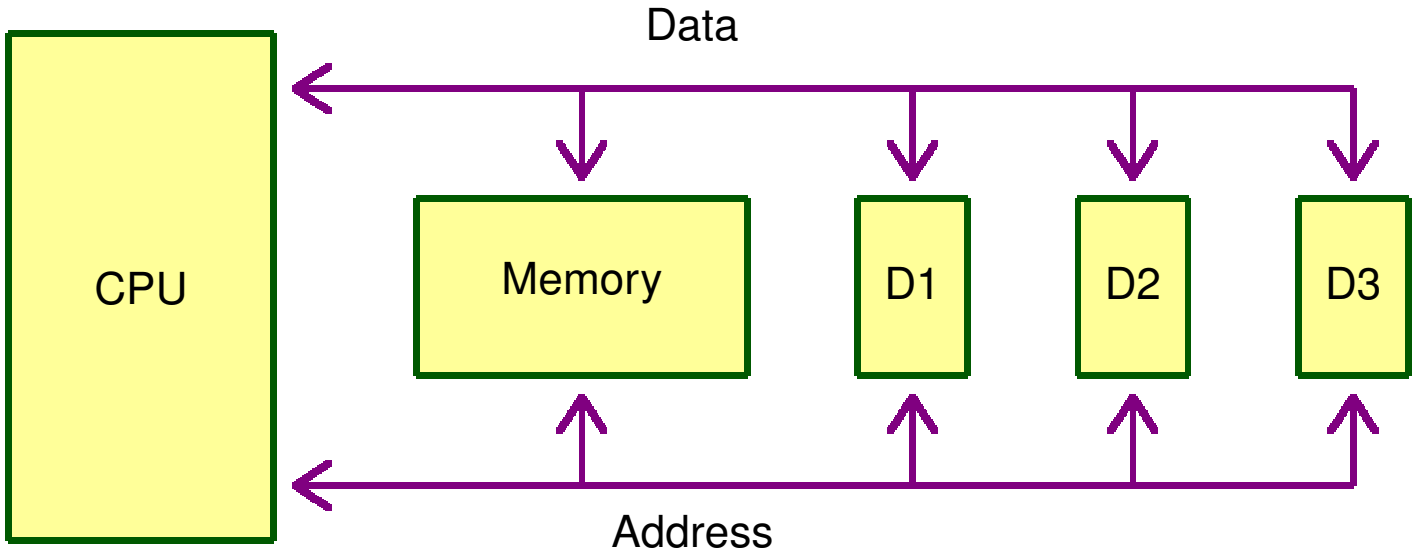
- Terminal device handler also handles output character depending of terminal settings: handles line breaks. handles other special characters (tabs, etc).

Separate I/O *Bus*



- Comes with special assembly language **I/O** instructions, like `IN reg, dev` and `OUT reg, dev`.
- Use assembly code procedures called by high level language or code insert.

Memory Mapped I/O *(DMA)*



- Device addressed as memory location.
- In high level language declare variable with

```
address = device address
VAR stat[177560B] : BITSET
rbuf [177562B] : CHAR
```

- Use high level language statements which reference memory.
- More flexible.

Buffering

Unbuffered I/O Data is transferred direct from user space to / from device.

Each read or write causes physical **I/O**, which implies that **FS** and device handler are used for each transfer.

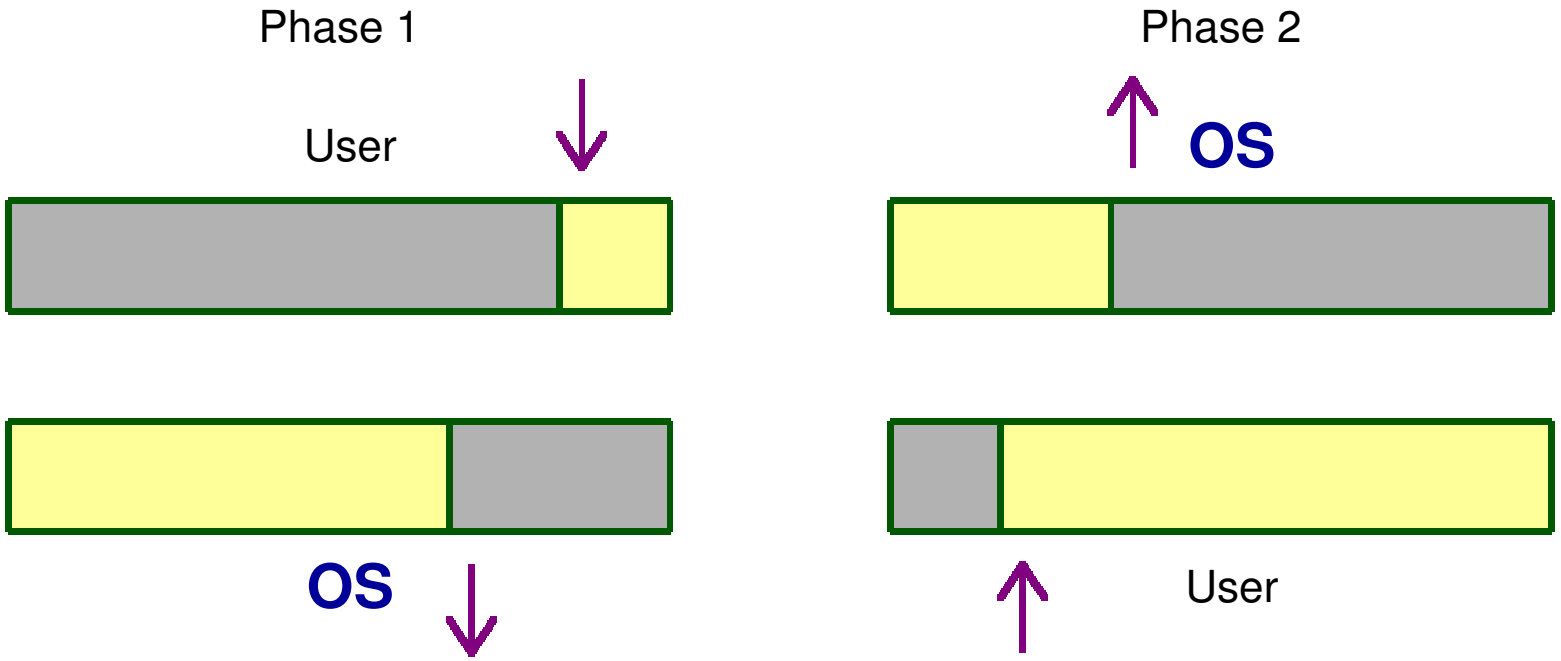
This implies a high process switching overhead (e.g. per character).

Buffered I/O Used to smooth peaks in **I/O** traffic and cater for differences in data transfer unit between devices:

Output user's output data is transferred to **OS** output buffer: user continues and is only suspended when buffer full'

Input **OS** reads ahead for user. Read normally satisfied from buffer: user blocked if buffer empty.

Double Buffering



More generally, can use bounded buffer.

If data passed by value in message, we use implicit buffering.

Spooling

Blocking user when opening a file to an allocated, non-shareable device causes delays and bottlenecks.

Solution: Spool to intermediate medium - disk file (e.g. line printer (lpr) spooler).

Only Spooler Process can access the printer itself.

1. user opens **`/dev/lpr`**.
2. **FS** opens **`/var/spool/printer/userN`**.
3. user writes to **`printer`** file, **FS** diverts output to disk file.
4. user closes **`/dev/lpr`**.
5. **FS** closes **`/var/spool/printer/userN`**.
6. **FS** sends filename '**`userN`**' to spooler.
7. Spooler eventually opens **`/var/spool/printer/userN`** for reading.
8. Spooler copies file to printer.
9. Spooler closes and deletes **`/var/spool/printer/userN`**.

Device Allocation

Dedicated For example *magnetic tape*, *terminal*, *printer*:

- allocated for long period;
- only allocated to authorised processes;
- **OS** Policy decision when to allocate - possible deadlock;
- simple policy - `open` fails if already opened (alternatively, queue `open` requests).

Shared For example: disks, window terminals. (**OS** provides filing system for disks - see later).

Spooled For example *printers*:

- provides sharing of non-sharable devices;
- reduces **I/O** time, which gives greater throughput;
- printer output saved to disk file; that file is printed later by the spooler daemon;
- printer is only allocated to the spooler daemon, no normal process is allowed direct access.

Minix **I/O** *User Interface*

- Minix **I/O** User Interface is very similar to Linux/Unix.
- Devices are accessible via *special files*, like:
  - **`/dev/console`**
  - **`/dev/printer`**
  - **`/dev/modem`**
  - **`/dev/mouse`**
- Device independence:

```
cp prog.c /dev/printer     cp prog.c /dev/mouse ?
cp prog.c /dev/console     cp /dev/lpd prog.c   ?
```

- **I/O** classes:
  - character (unstructured) files and devices;
  - block (structured) devices;
  - pipes (message) ipc;
  - socket (message) network interface.

Minix Device Handler Messages

Request Message

`type int` Operation requested: **`read`**, **`write`**, **`position`** heads, **`rewind`**, etc (device specific);
`device int` minor device number;
`proc int` user process requesting **I/O**;
`count int` byte count (number of bytes to transfer) is **`ioctl`** code;
`position long` position on device (e.g. disk address);
`address char*` buffer address of data in user process.

Reply

`type int` always reply;
`proc int` user process number form request, needed to identify relevant **`fd`** data structure in File System;
`reply status` number of bytes transferred or error number.

Clocks

Clocks tell time of day and block **CPU**-hogging (in multi-programmed computers); not to be confused with **CPU**-cycle synchroniser.

Clock is a device: has hardware and software.

Old Hardware – tied to a 110 or 220 volt power line;
- causes an interrupt on every voltage cycle (50 or 60 Hz).

New Hardware – crystal oscillator, a counter and holding register;
- crystal generates accurate signal (100s MHz - 1000 MHz);
- `signal` starts counter, when down to 0 then interrupt;
- interrupt frequency: count down and restart gives a *clock tick*; interrupt frequency controlled by software.

Driver maintains time of day; prevents processes from running longer than allowed; accounting for **CPU** usage; alarm system calls from user process; watchdog times for system parts (stop disk rotation on lap-top if not used); profiling, monitoring and statistics gathering.

Minix Device Handler Task

```
process Device_Handler () {
   initialise data structures and device;
   while (TRUE)  {
      receive (FS,ReqMessage); /* I/O request comes from FS */
      switch (ReqMessage.type) {
         case read;  process requested read operation;
         case write; process requested write operation;
         case ioctl; process requested control operation;
      };
      receive (IntMes);   /* wait for I/O completion */
      check for errors;
      set up IntMes with status to send to FS;
      send (FS,IntMes)
} }
```

Some errors handled by *task* (e.g. retry after disk read failure or communication errors). Others handled by the *File System* (e.g. printer out of paper). Send message to error handler which prints message to operator. File System reports ‘hard’ failures to users.

An example task: the printer driver

```
/* This file contains the printer driver. It is a fairly simple driver,
 * supporting only one printer. The valid messages and their parameters are:
 *
 * DEV_OPEN: initializes the printer
 * DEV_CLOSE: does nothing
 * HARD_INT: interrupt handler has finished current chunk of output
 * DEV_WRITE: a process wants to write on a terminal
 * CANCEL : terminate a previous incomplete system call immediately */

PUBLIC void main(void)
 {
    /* Main routine of the printer task. */
   message pr_mess;              /* buffer for all incoming messages */
   struct sigaction sa;
   int s;

    /* Install signal handlers. Ask PM to transform signal into message. */
   sa.sa_handler = SIG_MESS;
   sigemptyset(&sa.sa_mask);
   sa.sa_flags = 0;
   if (sigaction(SIGTERM,&sa,NULL)<0)
      panic("PRN","sigaction failed", errno);
```

Printer driver (2)

```
while (TRUE)  {
   receive(ANY, &pr_mess);
   switch(pr_mess.m_type)  {
      case DEV_OPEN:
         do_initialize();                    /* initialize  */
       /* fall through  */
      case DEV_CLOSE:
         reply(TASK_REPLY, pr_mess.m_source, pr_mess.PROC_NR, OK);
         break;
      case DEV_WRITE:      do_write(&pr_mess); break;
      case DEV_STATUS:     do_status(&pr_mess); break;
      case CANCEL:         do_cancel(&pr_mess); break;
      case HARD_INT:       do_printer_output(); break;
      case SYS_SIG:        do_signal(&pr_mess); break;
      case DEV_PING:       notify(pr_mess.m_source); break;
      default:
         reply(TASK_REPLY, pr_mess.m_source,
               pr_mess.PROC_NR, EINVAL);
       }
    }
 }
```

Printer driver (3)

```
PRIVATE void do_write(m_ptr)
register message *m_ptr;   /* pointer to the newly arrived message   */
 {
    /* The printer is used by sending  DEV_WRITE messages to it. Process one.    */
   register int r = SUSPEND;
   int retries;
   int status;
    /* Reject command if last write is not yet finished, the count is not
    * positive, or the user address is bad.     */
   if (writing) r = EIO;
   else if (m_ptr->COUNT <= 0) r = EINVAL;
    /* Reply to  FS, no matter what happened, possible  SUSPEND caller.     */
   reply(TASK_REPLY, m_ptr->m_source, m_ptr->PROC_NR, r);
```

Printer driver (4)

```
/* If no errors occurred, continue printing with SUSPENDed caller.
   * First wait until the printer is online to prevent stupid errors.  */
  if (SUSPEND == r)  {
     caller = m_ptr->m_source;
     proc_nr = m_ptr->PROC_NR;
     user_left = m_ptr->COUNT;
     orig_count = m_ptr->COUNT;
     user_vir = (vir_bytes) m_ptr->ADDRESS;
     writing = TRUE;
     retries = MAX_ONLINE_RETRIES + 1;
     while (--retries > 0)  {
        sys_inb(port_base + 1, &status);
        if ((status & ON_LINE))  {  /* printer online!   */
           prepare_output();
           do_printer_output();
           return;  }
        tickdelay(30);                    /* wait before retry  */
      }
      /* If we reach this point, the printer was not online in time.  */
     done_status = status;
     output_done();
}  }
```

Printer driver (5)

```
PRIVATE void output_done()
 {   /* Previous chunk of printing is finished. Continue if OK and more.
      * Otherwise, reply to caller ( FS).  */
   register int status;

   if (!writing) return;                    /* probably leftover interrupt  */
      if (done_status != OK)  {             /* printer error occurred  */
         status = EIO;
         if ((done_status & ON_LINE) == 0)
            printf("Printer is not on line \n");
         else if ((done_status & NO_PAPER))  {
            printf("Printer is out of paper \n");
            status = EAGAIN;
          } else
            printf("Printer error, status is 0x%02X \n",
                                 done_status);
          /* Some characters have been printed, tell how many.    */
         if (status == EAGAIN && user_left < orig_count)
            status = orig_count - user_left;
         oleft = 0;                         /* cancel further output  */
       }
```

Printer driver (6)

```
else if (user_left != 0)  {    /* not yet done, continue!    */
   prepare_output(); return;
 }
   else status = orig_count;  /* done! report back to FS  */
   revive_pending = TRUE;
   revive_status = status;
   notify(caller);
 }

PRIVATE void reply(code, replyee, process, status)
int code;                     /*  TASK_REPLY or REVIVE  */
int replyee;                  /* destination for message (normally FS)  *
int process;                  /* which user requested the printing  */
int status;                   /* number of chars printed or error code  */
 {
   /* Send a reply telling FS that printing has started or stopped.    */
  message pr_mess;
  pr_mess.m_type = code;      /*  TASK_REPLY or REVIVE  */
  pr_mess.REP_STATUS = status;  /* count or EIO  */
  pr_mess.REP_PROC_NR = process;  /* which user does this pertain to  */
  send(replyee, &pr_mess);    /* send the message  */
 }
```

Printer driver (7)

```
PRIVATE void do_status(m_ptr)
register message *m_ptr;    /* pointer to the newly arrived message   */
 {
   if (revive_pending)  {
      m_ptr->m_type = DEV_REVIVE;        /* build message   */
      m_ptr->REP_PROC_NR = proc_nr;
      m_ptr->REP_STATUS = revive_status;

      writing = FALSE;                   /* unmark event   */
      revive_pending = FALSE;            /* unmark event   */
    } else m_ptr->m_type = DEV_NO_STATUS;
   send(m_ptr->m_source, m_ptr);         /* send the message   */
 }

PRIVATE void do_cancel(m_ptr)
register message *m_ptr;                 /* pointer to the newly arrived messa
 {
   if (writing && m_ptr->PROC_NR == proc_nr)   {
      oleft = 0;                         /* cancel output by interrupt handler
      writing = FALSE;
      revive_pending = FALSE;
    }
   reply(TASK_REPLY, m_ptr->m_source, m_ptr->PROC_NR, EINTR);
 }
```

Printer driver (8)

```
PRIVATE void do_initialize()
 {   /* Set global variables and initialize the printer.     */
   static int initialized = FALSE;
   if (initialized) return;
   initialized = TRUE;
    /* Get the base port for first printer.     */
   sys_vircopy(SELF, BIOS_SEG, LPT1_IO_PORT_ADDR,
         SELF, D, (vir_bytes) &port_base, LPT1_IO_PORT_SIZE);
   sys_outb(port_base + 2, INIT_PRINTER);
   tickdelay(1);                  /* easily satisfies Centronics minimum   */
                                  /* was 2 millisecs; now is  17 millisecs   */
   sys_outb(port_base + 2, PR_SELECT);
   irq_hook_id = 0;
   sys_irqsetpolicy(PRINTER_IRQ, 0, &irq_hook_id);
   sys_irqenable(&irq_hook_id);
 }

PRIVATE void do_printer_output()
 {   /* This function does the actual output to the printer. This is called on
     * a  HARD_INT message sent from the generic interrupt handler that 'forwards'
     * interrupts to this driver. The generic handler did not reenable the
     * printer IRQ yet!     */
```

Printer driver (9)

```
PRIVATE void prepare_output()
 {  /* Start next chunk of printer output. Fetch the data from user space.  */
   register int chunk;
   if ( (chunk = user_left) > sizeof obuf) chunk = sizeof obuf;
   if (OK!=sys_datacopy(proc_nr, user_vir, SELF,
         (vir_bytes) obuf, chunk))  {
      done_status = EFAULT;
      output_done();
      return;
    }
   optr = obuf;
   oleft = chunk;
 }
PRIVATE void do_signal(m_ptr)
message *m_ptr;                 /* signal message  */
 {
   int sig;
   sigset_t sigset = m_ptr->NOTIFY_ARG;
    /* Expect a  SIGTERM signal when this server must shutdown.  */
   if (sigismember(&sigset, SIGTERM)) exit(0);
    /* Ignore all other signals.  */
 }
```

File System

Objectives

A file is a *named* collection of data of arbitrary size.

- To provide long term, non-volatile, on-line storage for, e.g. programs, data, text.
- To allow the sharing of information of programs such as editors, compilers, utilities.
- To allow concurrent access to shared databases such as airline reservation information.
- To provide users with a convenient means of organising and accessing data, by allowing the use of symbolic names, performing an automatic back-up, etc.

Filing System User Functions

`create` Create empty file: allocate space and enter name and location into directory.

`delete` Deallocate space and invalidate or remove directory entry.

`open` Search directory for file name, check access validity and set pointers to file.

`close` Remove pointers to file.

`read` Access file, update current position pointers.

`write` Access file, update pointers.

`reposition` Set current position to a particular value.

`truncate` Erase contents but keep all other attributes.

`rename` Change name.

Filing System User Functions (2)

Read attributes: creation date, size. Write attributes: protection.

Support Functions:

- Management of disk space (usually in terms of blocks).
- Symbolic name to physical disk address.
- Locking of files for exclusive access.
- Performance optimisation via caching and read ahead.
- Protection against system failure (back-up, recovery, archive).
- Protection against unauthorised user access, but allow sharing between co-operating users.

Layered File Systems

Application Programs High level language interface to files. Access methods: *sequential*, *random*, or *indexed sequential*.

Logical File System Directory Level Provides mapping from symbolic names to file location i.e. supports directory system (access control). Usually a *tree-structured* directory system, e.g. Unix, Apple, Windows.

File Organisation Module Allocation and management of file space, free space etc.

Basic File System Translates block address to physical disk address (eg. *drive*, *cylinder*, *surface*, *sector*).

I/O Control Device drivers and interrupt handlers which transfer physical blocks of data between memory and disk.

Devices Disks on which files are stored.

Note: Archive services usually run as application programs.

File Access Methods

Sequential – file blocks processed in order;
- a **`read`** reads the next block and updates pointer;
- a **`write`** appends to end of file;
- can rewind (to start) or skip blocks.

Direct Access (Random access) – file viewed as numbered sequence (array) of blocks;
- can read / write blocks in *random* order.

Indexed Sequential – records are arranged in logical sequence according to *key* contained in each record;
- maintain index from key to record;
- can access sequentially in key order or random.

Partitioned – file of sequential sub-files (libraries).

Unit of access can be record, byte or arbitrary block.

File Attributes

A subset of the following file attributes may be held within a particular directory system

Basic Information **file name** symbolic name, unique within directory;

file type eg text, binary, executable, directory;

file organisation eg sequential, random etc;

creator program which created file and can be used to open file (like Mac style double click).

Address Information **volume** disk drive, partition;

start address cylinder, track, block number;

size used

size allocated

File Attributes (2)

Access Control Information **owner** person who controls the file, often the creator;

authentication password;

permitted actions eg *read*, *write*, *delete* for owner or others.

Usage Information **creation** date and time;

last modify date and time sometimes includes user **ID**;

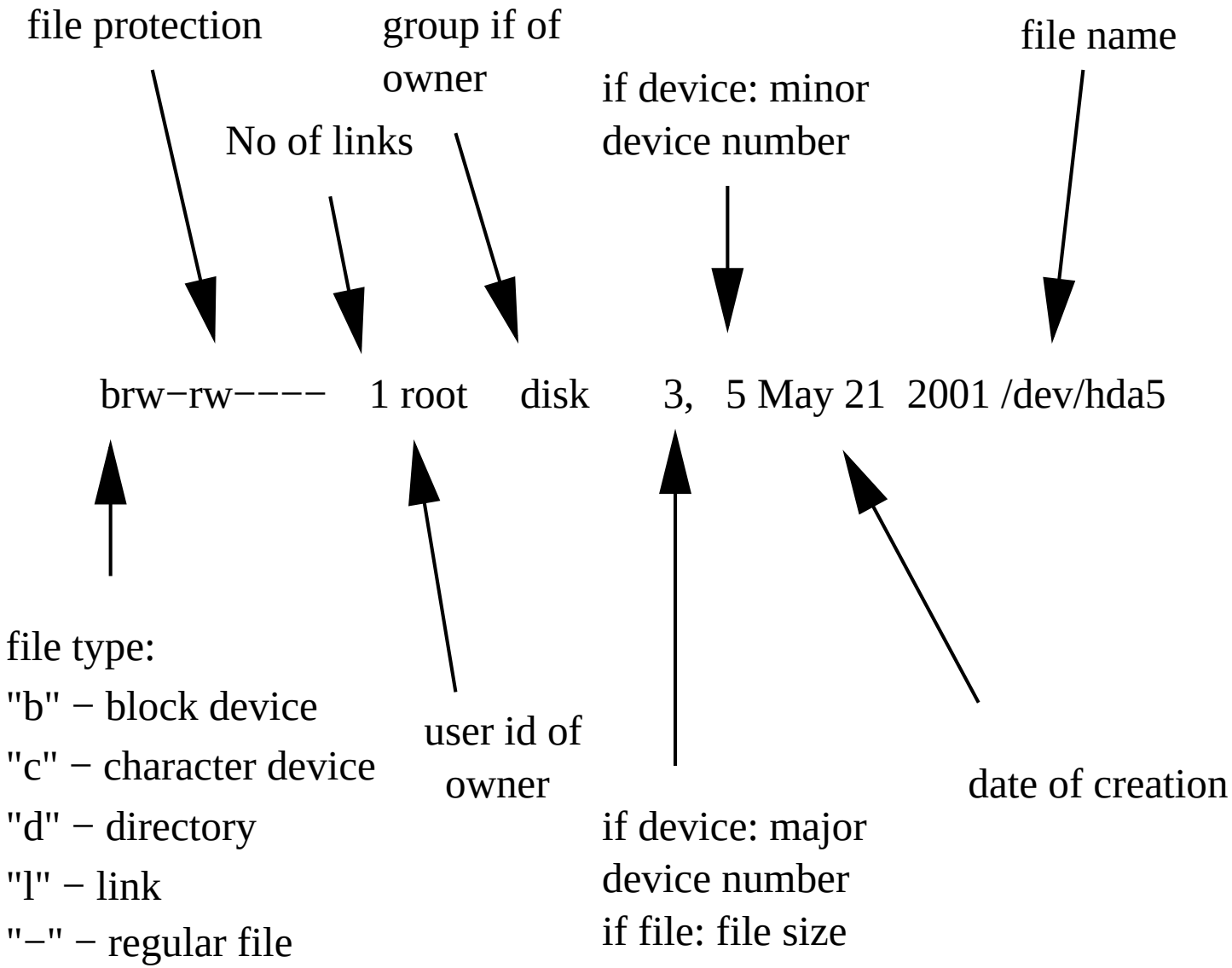
last read date and time sometimes includes user **ID**;

archive date and time

expiry date when the file will be deleted;

access activity counts number of reads / writes, that managers can use to control file system.

File Attributes (Unix)



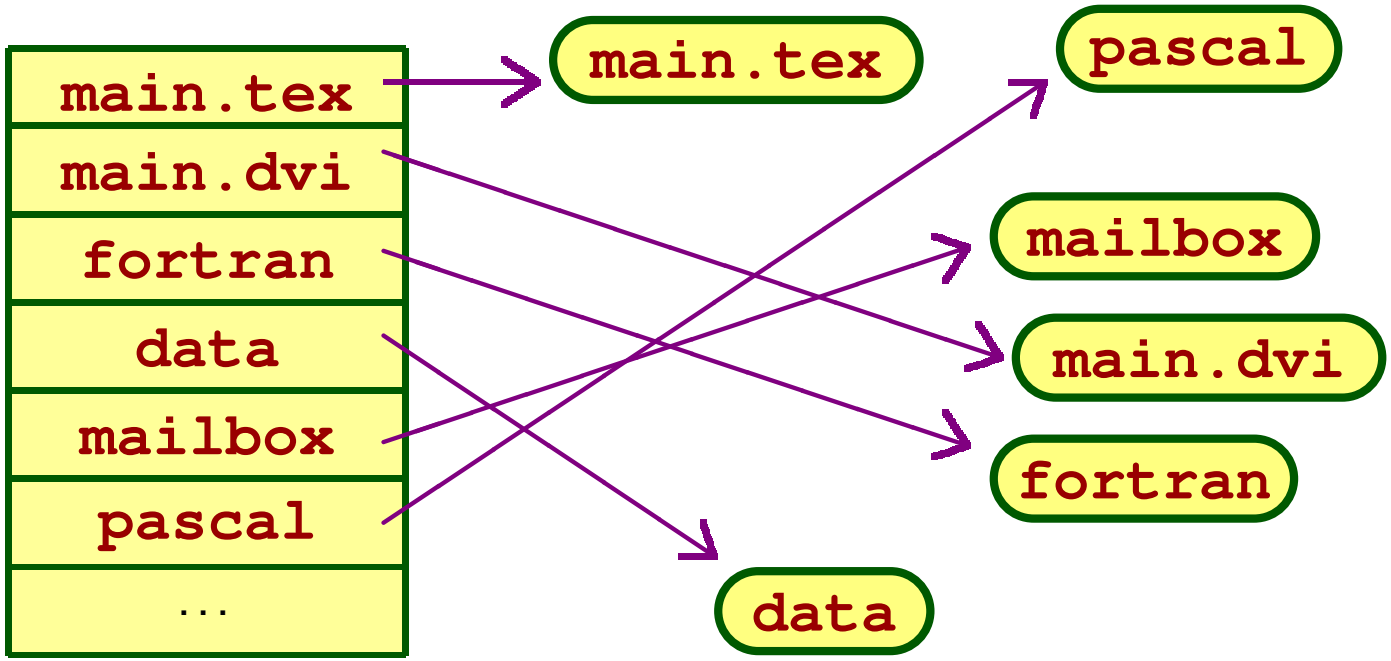
stat

File attributes can be accessed using system call **stat** (from **include/sys/stat.h**).

```
struct stat {
  dev_t st_dev;          /* major/minor device number */
  ino_t st_ino;          /* i-node number */
  mode_t st_mode;        /* file mode, protection bits, etc. */
  short int st_nlink;    /* # links; */
  uid_t st_uid;          /* uid of the file's owner */
  short int st_gid;      /* gid; */
  dev_t st_rdev;
  off_t st_size;         /* file size */
  time_t st_atime;       /* time of last access */
  time_t st_mtime;       /* time of last data modification */
  time_t st_ctime;       /* time of last file status change */
}
```

One level directory

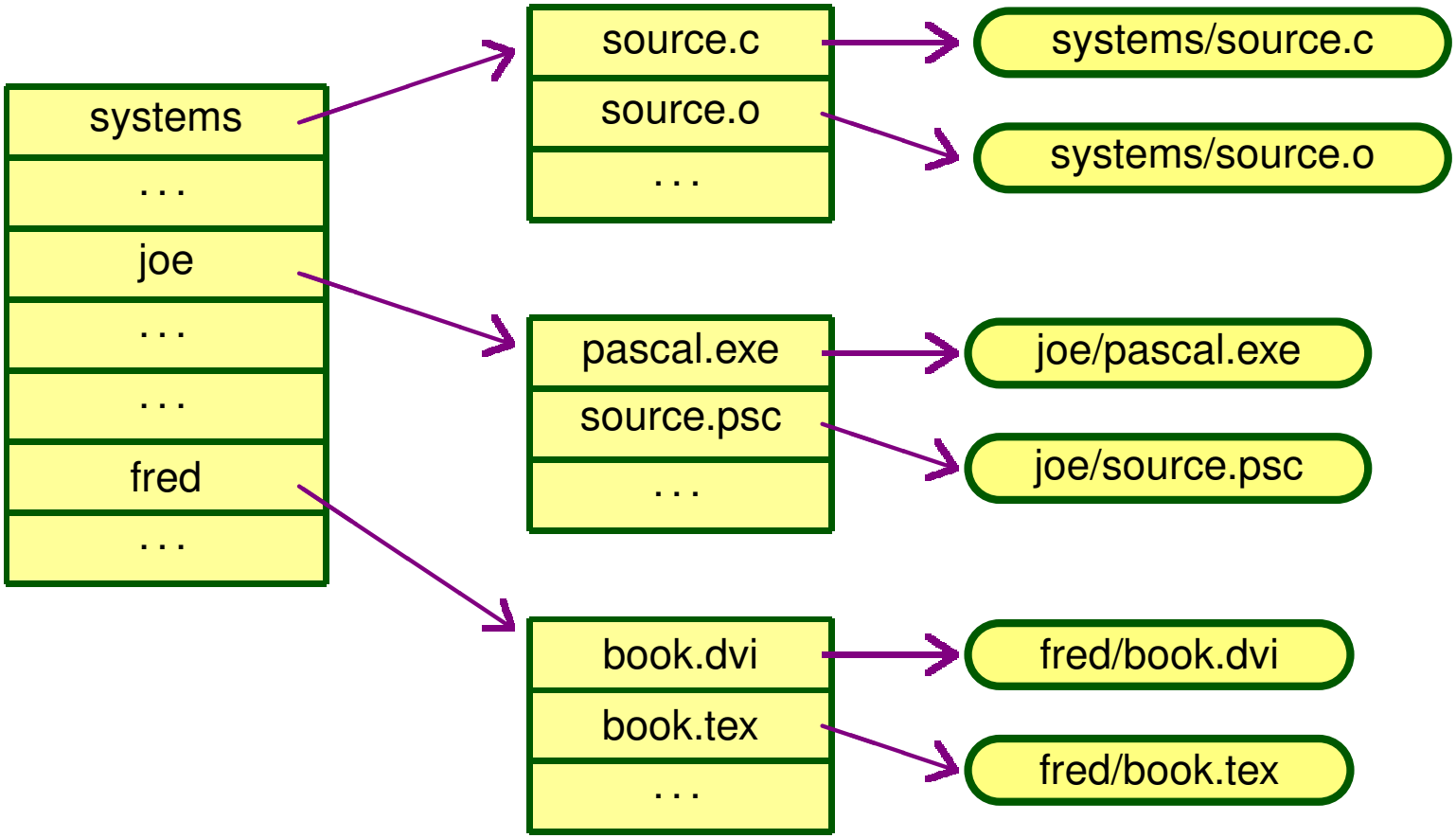
A *directory* maps symbolic names to logical disk locations, or to a physical disk address.



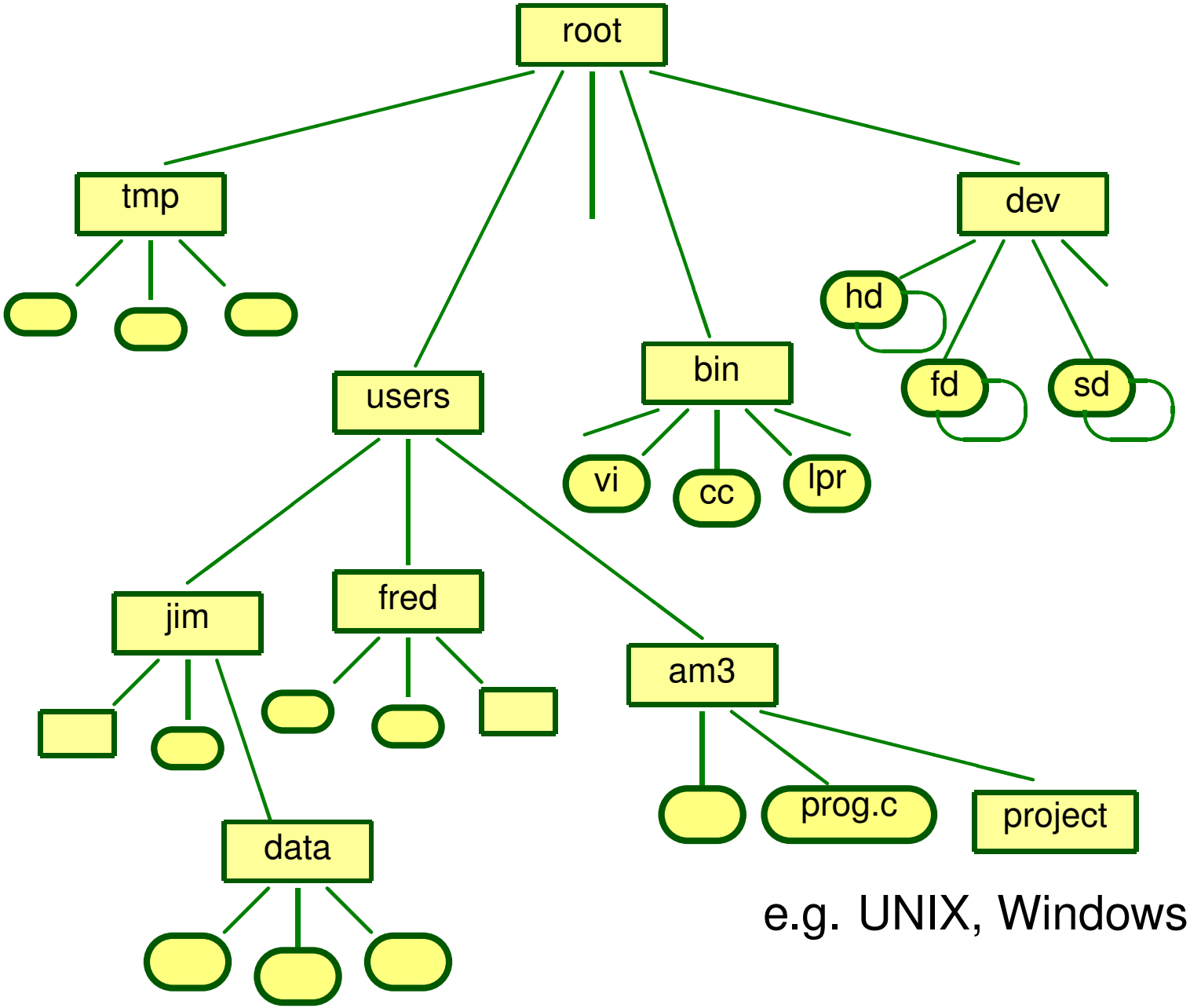
Usually simple system holding subset of file attributes - basic address and access control information.

Two-Level Directory Structure

- Master File Directory (MFD): one entry per user.
- User File Directory (UFD): one entry per file.



Multi-Level (Tree) Directory Structure



e.g. UNIX, Windows, Mac

Directory Operations

Open / Close directory.

Search Find file using pattern matching on a string, wildcard characters, e.g. `abc???.pas`, `fred.*`.

Create / Delete files or directories.

Link Create a link to a file.

Unlink Remove a link to a file.

Change directory Opens new directory as current one.

List or display files in directory.

Read Attributes of a file.

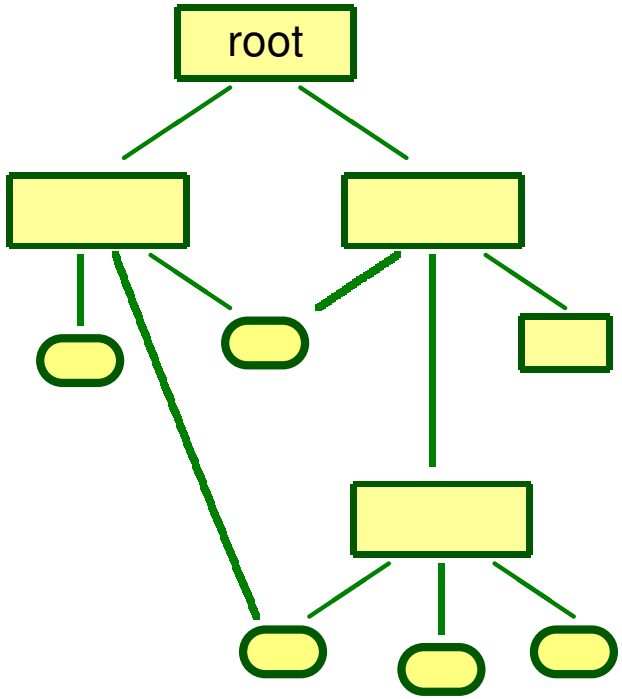
Write attributes Change attributes of a file.

Mount Creates a link in a directory to a directory on another disk or remote server.

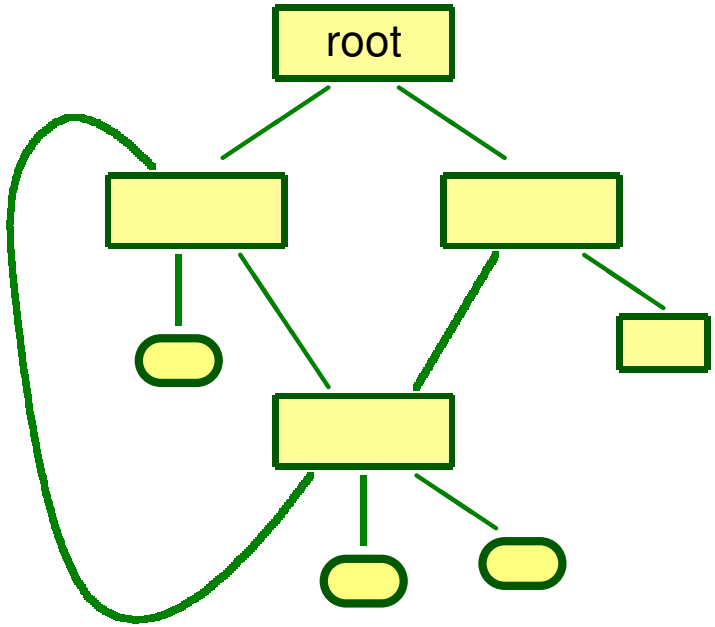
Links

Reference to a directory in another part of the tree or a file in another directory. Allows alternative names for same file. (Alias in OSX).

Hard link references address of file (only for files in Unix);
Symbolic link references full path name of file/directory.



Acyclic Graph Directory



Cyclic Graph Directory

Problems with links

```
lrwxrwxrwx 1 svb    29 Sep 26 00:44 OSmac.sty ->
                       ../../../TeX/styles/OSmac.sty
lrwxrwxrwx 1 svb    32 Sep 26 00:43 Slides.sty ->
                       /Users/svb/TeX/styles/Slides.sty
-rw------- 2 svb 68865 Sep 25 21:23 concurrency.tex
```

File deletion Search for links and remove them:

Leave links cause exception when used; e.g. symbolic links, Mac alias;

Keep link count with file only delete file when count equals 0, e.g. hardlinks.

Looping Directory traversal algorithms may loop, eg. archiving utilities.

Cross-Device Links It is not possible to have a hard link to a file on a *mounted* file system; only a symbolic link will do.

Minix per Process File System Data

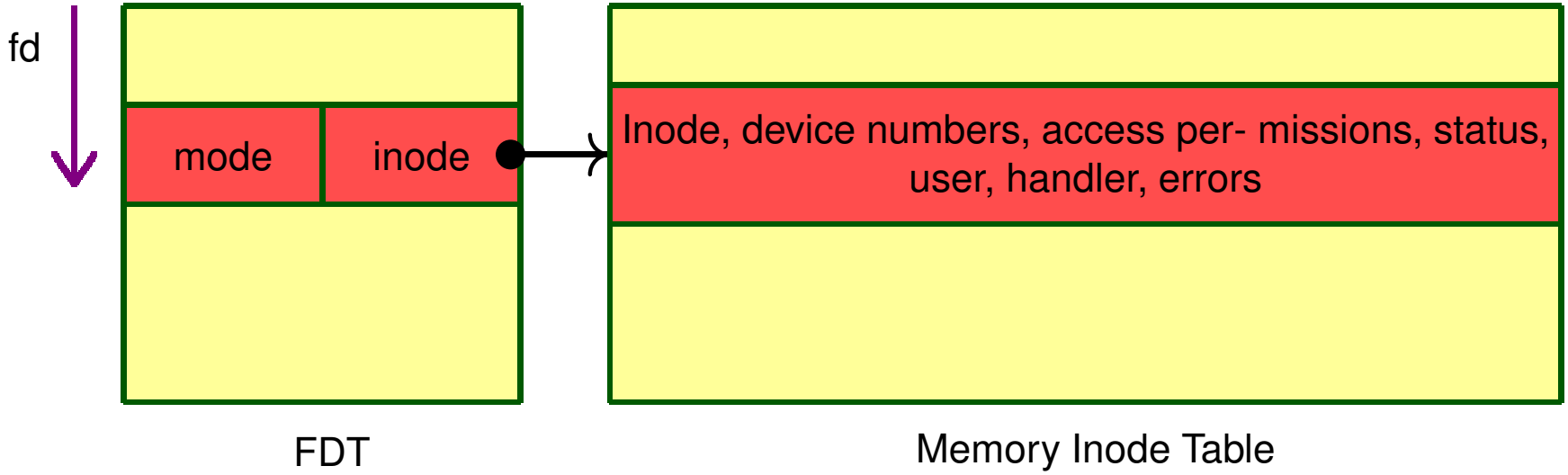
- Sharing model does not work in Unix / Minix as spawned process inherits its parent's open files, even the current position pointer.
- When file is **`open`**ed or **`create`**d, the call returns the file descriptor (**`fd`**).
- File position table:

  mode indicates if file is opened to read, write, etc;
  count number of processes that have opened the file;
  position offset from start of file in bytes; gets updated by **`read`**, **`write`**, or **`seek`** (not used in device files);
  pointer to open inode entry.

- File system holds following information for each process:
  - pointer to inode of *current root directory*, modified by **`chroot`**;
  - pointer to inode of *current working directory*, **`chdir`**;
  - *user* and *group* identifiers;
  - *process* identifier;
  - table of pointers to *shared file position info*, indexed by **`fd`**.

File Descriptor Table

Each process has its *File Descriptor Table*.

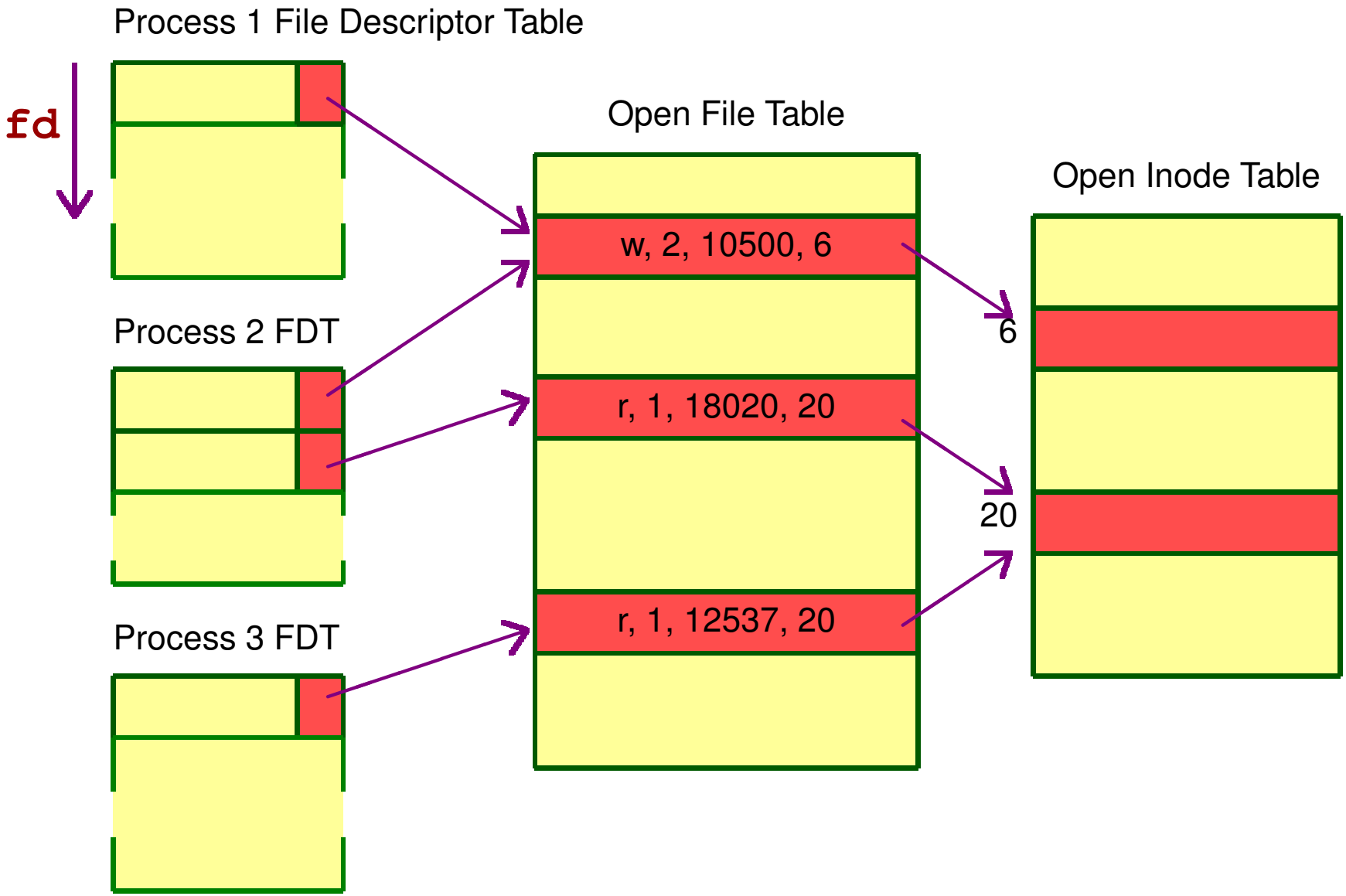


Each process has 3 file descriptors when created.

file descriptor	name
0	stdin
1	stdout
2	stderr

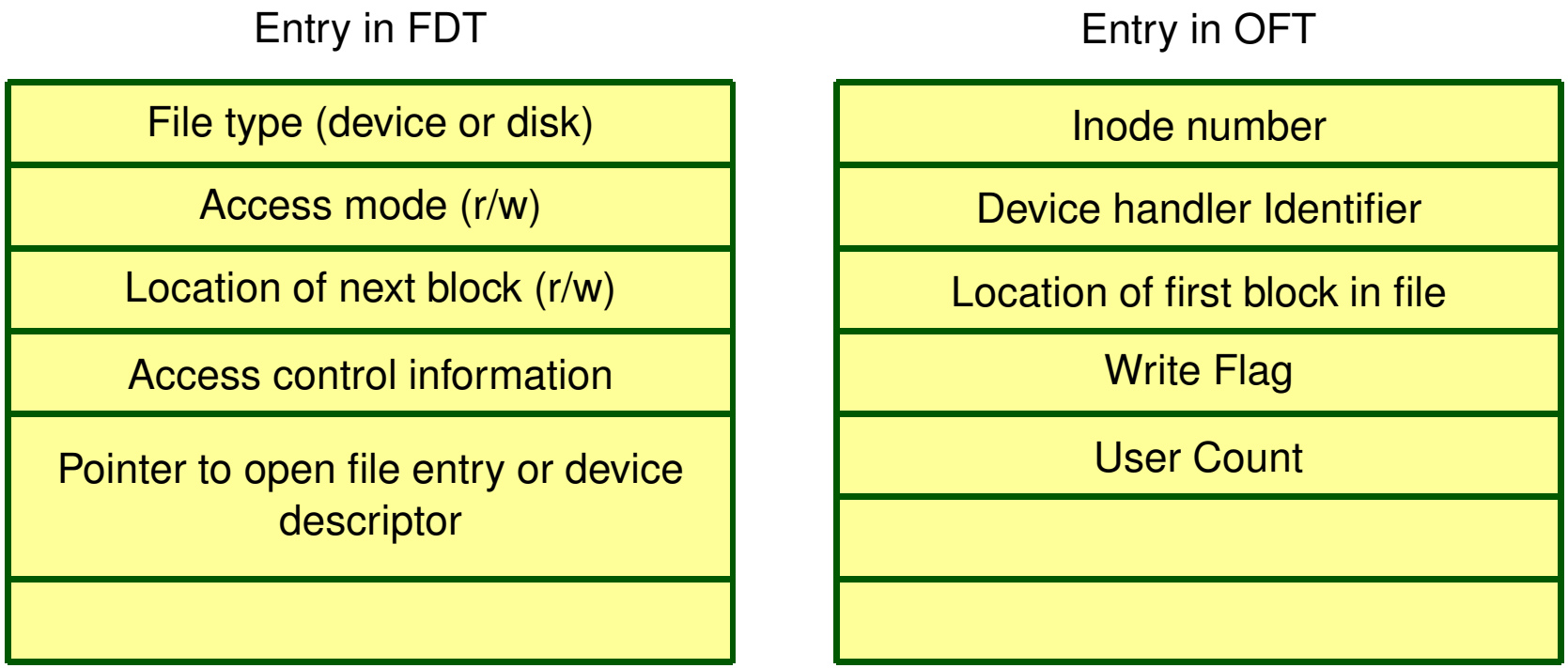
Minix/Unix File Sharing

Remember that **`fork`** copies a **PCB**, so an entry to an open file with pointer can be shared.



Open File Table

The Open File Table is a list maintained for the **OS** for all the files in use by the processes.



To implement the readers/writers problem, do: if user count is 0, and Write Flag is false, then permit the file to be opened in writemode; else, if the writeflag is Write Flag then permit open in readmode, and increment the User Count.

Consistency File Semantics

Unix Semantics – writes immediately visible to all users;

- child process can share current location pointer in file: parent reads block 510 (for file), child then reads block 511.

Unix keeps a *single file image* for every file.

Session Semantics – writes to file not visible to other users;

- when file is closed, changes are made visible to new sessions, but are not seen by current sessions.

This requires to maintain multiple copies of a file.

Immutable Files Read-only sharing.

Locks Sharing of files requires the capability for locking to give exclusive access to prevent others reading while the file is being updated. Lock can be on:

- whole file;
- individual record of a file.

Locks

- Advisory file locking. This is a mechanism *provided* by the OS, but not *enforced* by the OS. Processes must look for locks on file before requesting access.
- Implements multiple locks over *disjoint* regions of a single file.
- Use shared lock table, with each entry containing:
  - lock type: reading or writing;
  - **PID** of process that holds the lock;
  - inode number of file;
  - offset of first byte of lock region;
  - offset of last byte of lock region.

Disk Organisation

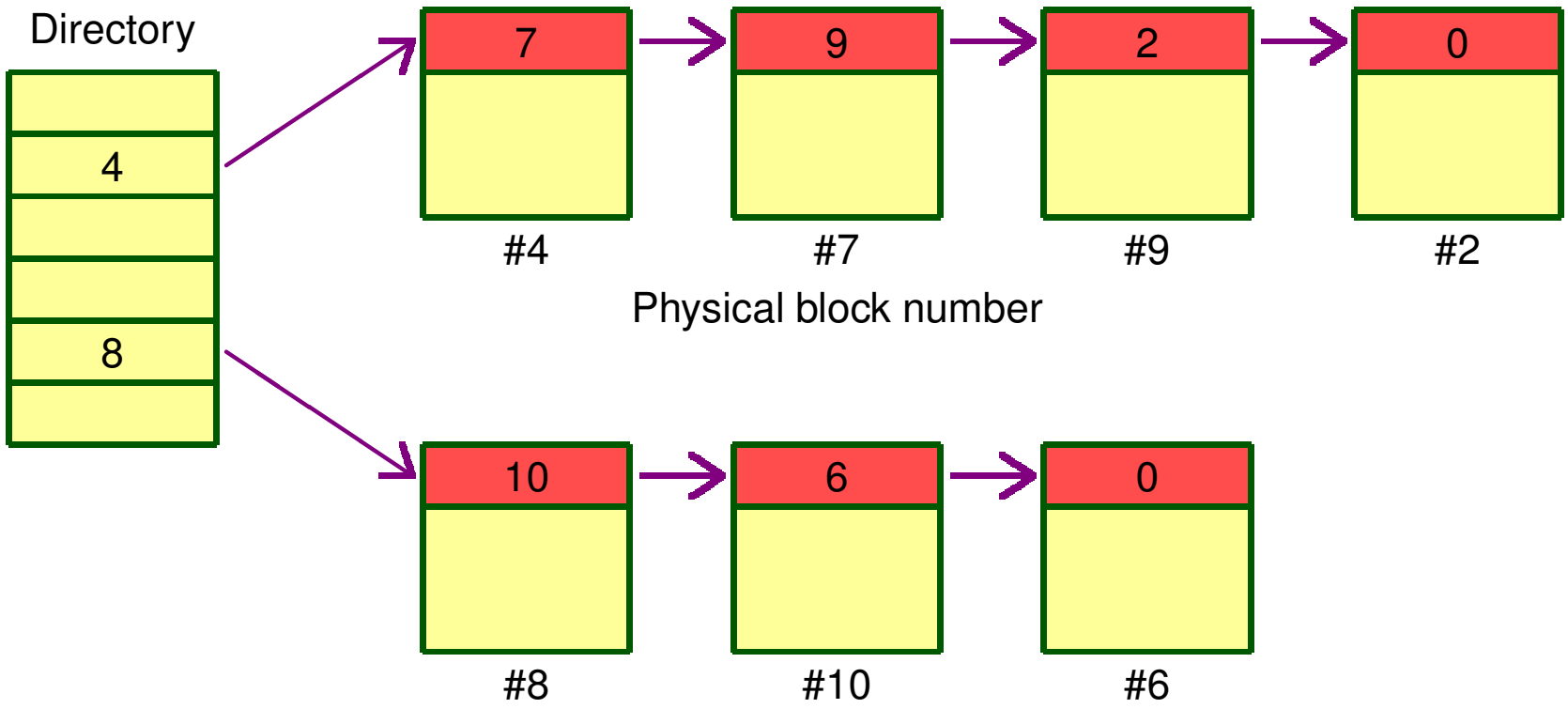
Size of a file is naturally variable which calls for *dynamic space management*. Generally, space is allocated in *blocks* that are (typically) 512 to 4096 bytes in size. Various methods exist for accessing the blocks belonging to a file.

Blocks are grouped in a list; number used for indexing.

Block Linkage (chaining)

- Search chain to find required logical block.
- Disadvantage: Only suitable for sequential file access, since many disk accesses are required to find the *n*-th file element.
- Wastes pointer space in each block.

Block Linkage



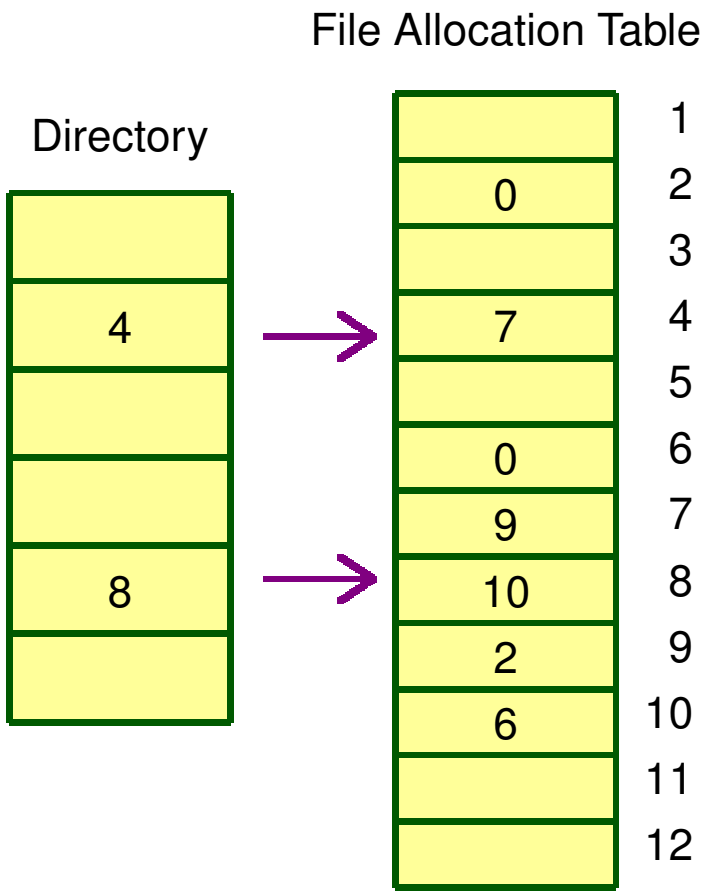
Better to allocate blocks in cluster e.g. 4 contiguous blocks and only then use pointer to next cluster.

File Allocation Table (File Map)

The allocation of disk blocks to files is recorded in a *file allocation table* stored in one or more disk blocks (e.g. MS-DOS (FAT-block) and OS/2).

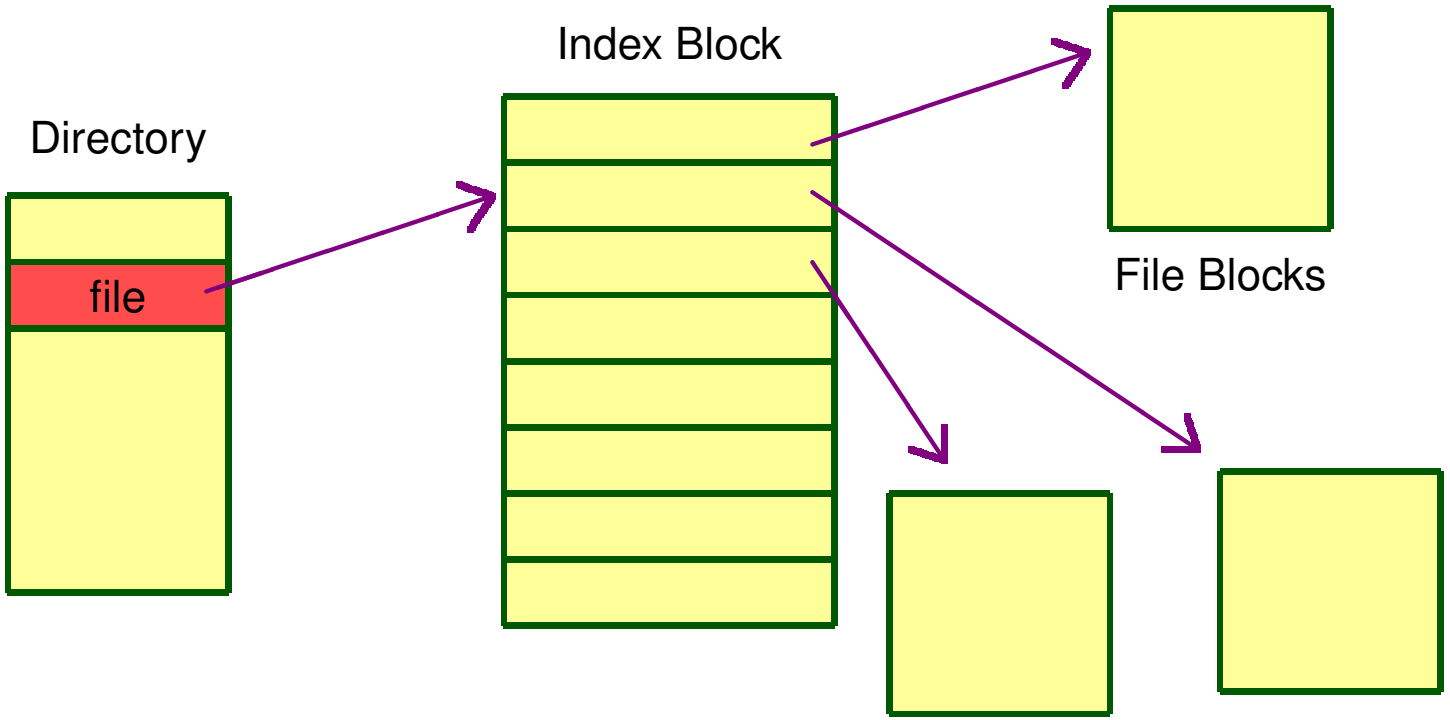
- Sequential Access only.
- Hold all (or part of) the *file allocation table* in store to reduce the number of disk accesses. So a maximum of two disk accesses per file block access is needed if the blocks for the file are referenced on one file map block.

Index FAT using physical block number. Each entry holds block number of next physical block in file.



Second entry is stored in blocks 4,7,9, and 2; fifth in 8, 10, and 6

Index Blocks

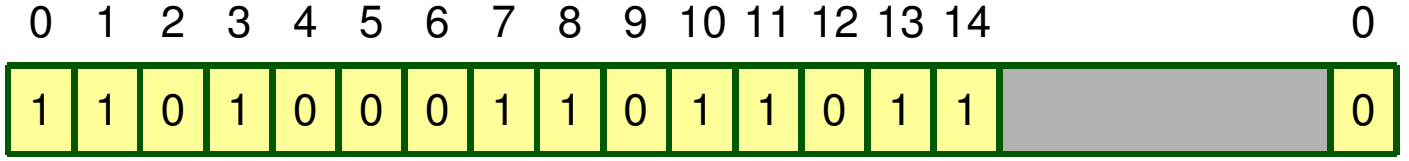


- After first access the index block can be kept in memory.
- Good for *random access*.
- A whole index block at a time must be allocated even for small files: *larger disk space overheads*.
- Difficult to insert and delete blocks.

Free Space Management

Free space file Use methods described above to keep a file of free blocks. Linked lists can be slow for allocating multiple blocks. Using a block of pointers (c.f. index block) permits fast allocation of multiple free blocks.

Bitmap Array of bits (1 per disk block).



Note: A 500 Mbyte disk requires 500K bits for 1Kbyte blocks, i.e. 62.5 Kbytes, so feasible to hold in main store for efficiency, but system crashes may result in inconsistencies between map in store and map on disk.

For a 500 Gb disk, this is 62.5Mb; when choosing 4Kb blocks, this becomes 8Mb, still doable.

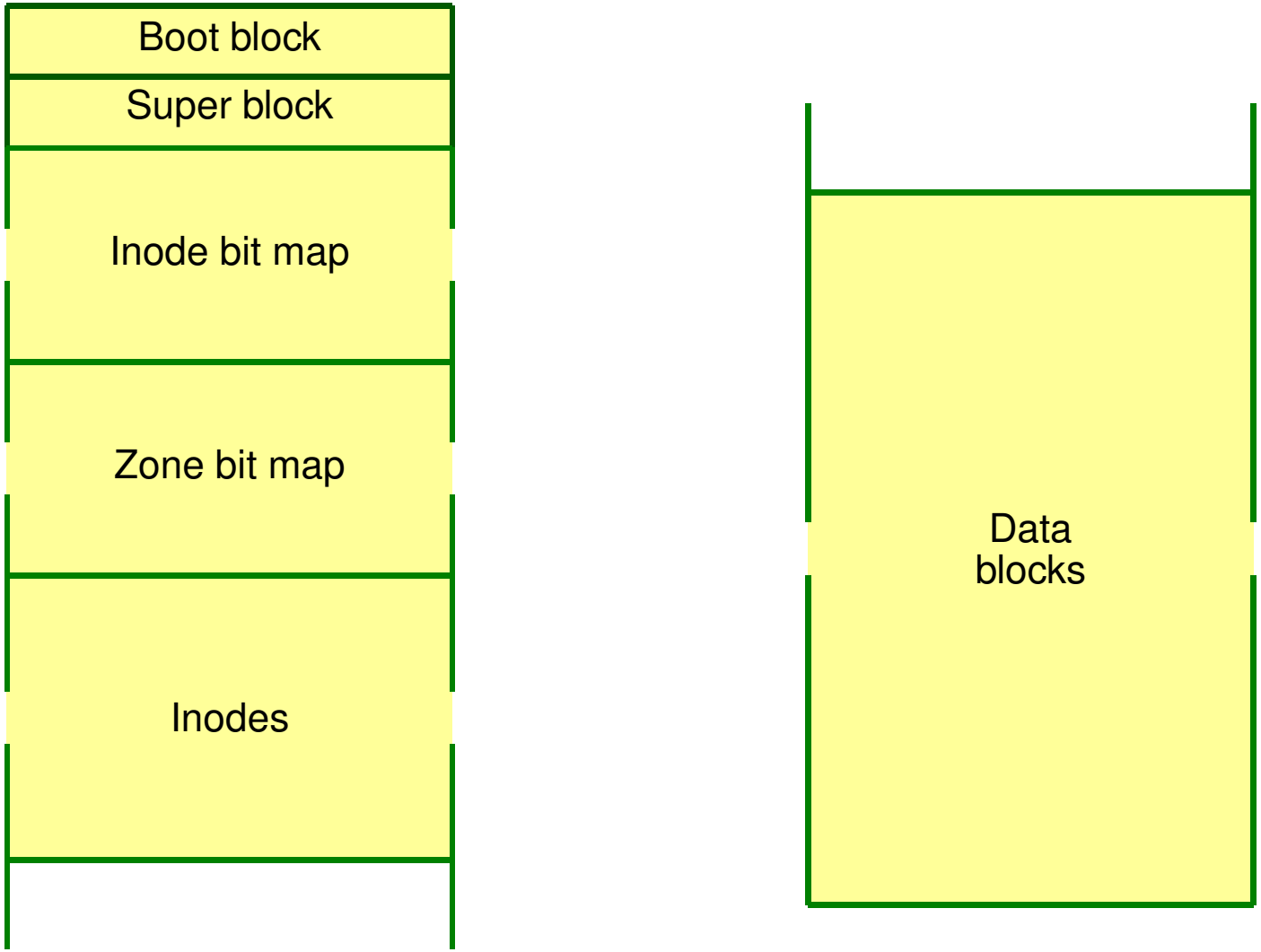
Criteria For Choosing Disk Block Size

- Too large: waste space for small files.
- Too small: waste space on large numbers of pointers and increase file transfer time as seek time to access each block of file is greater than the block transfer time.
- Memory buffer space - space needed for current index block and block being read or written.

Typical Block sizes:

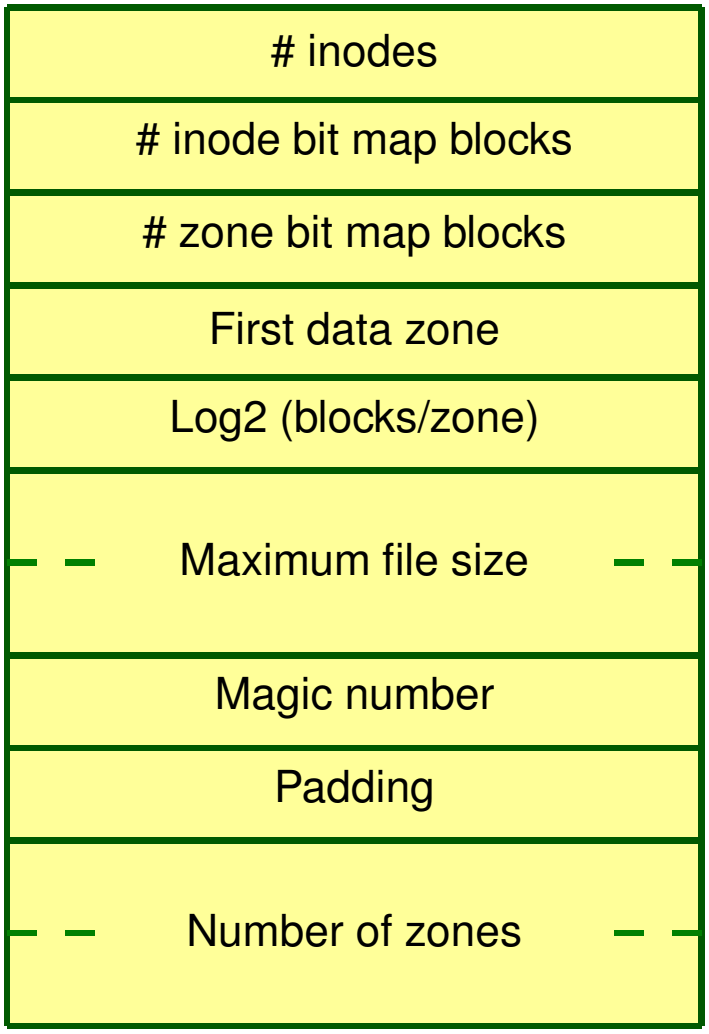
- Unix: 1K for small blocks and 8K for large blocks.
- Minix has 1K blocks allocated in zones of 2, 4, 8, 16, ... blocks, depending on the disk size.

Minix File System: Disk Lay-out

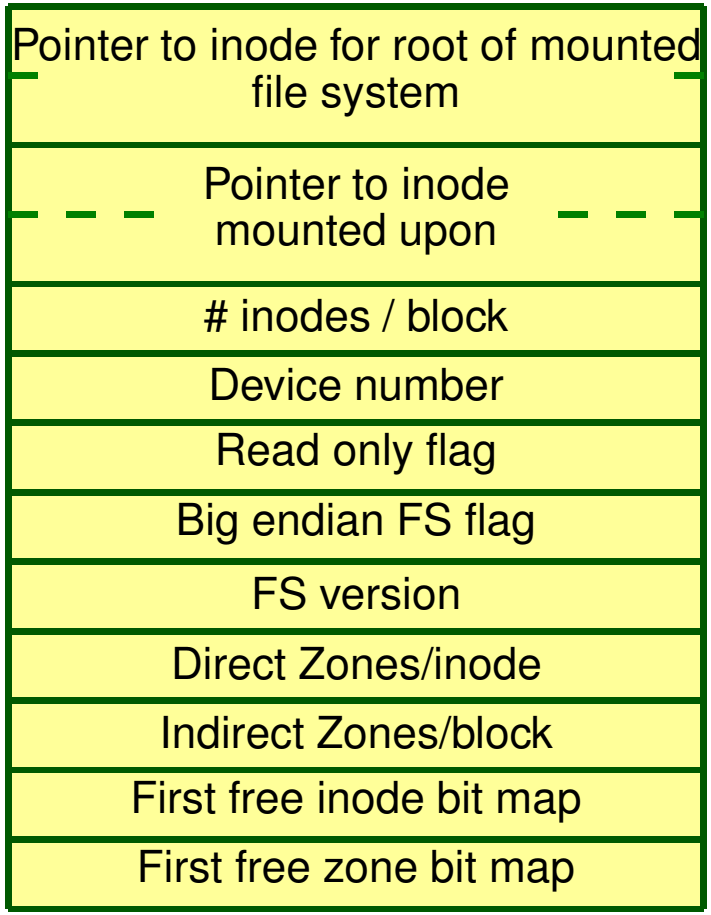


Minix File System Lay-out

Super Block Information on disk and in-memory

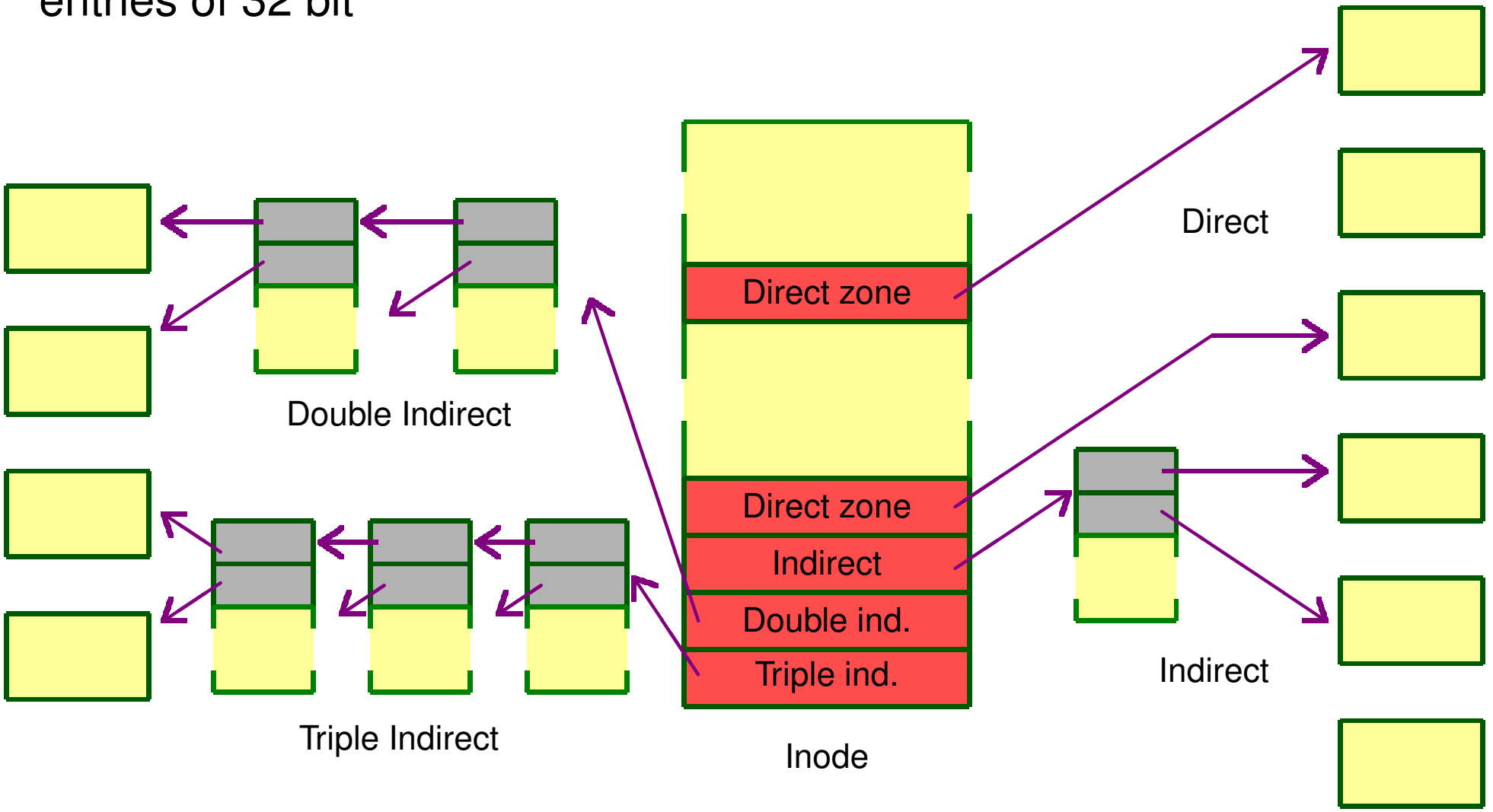


Super Block Information in-memory only

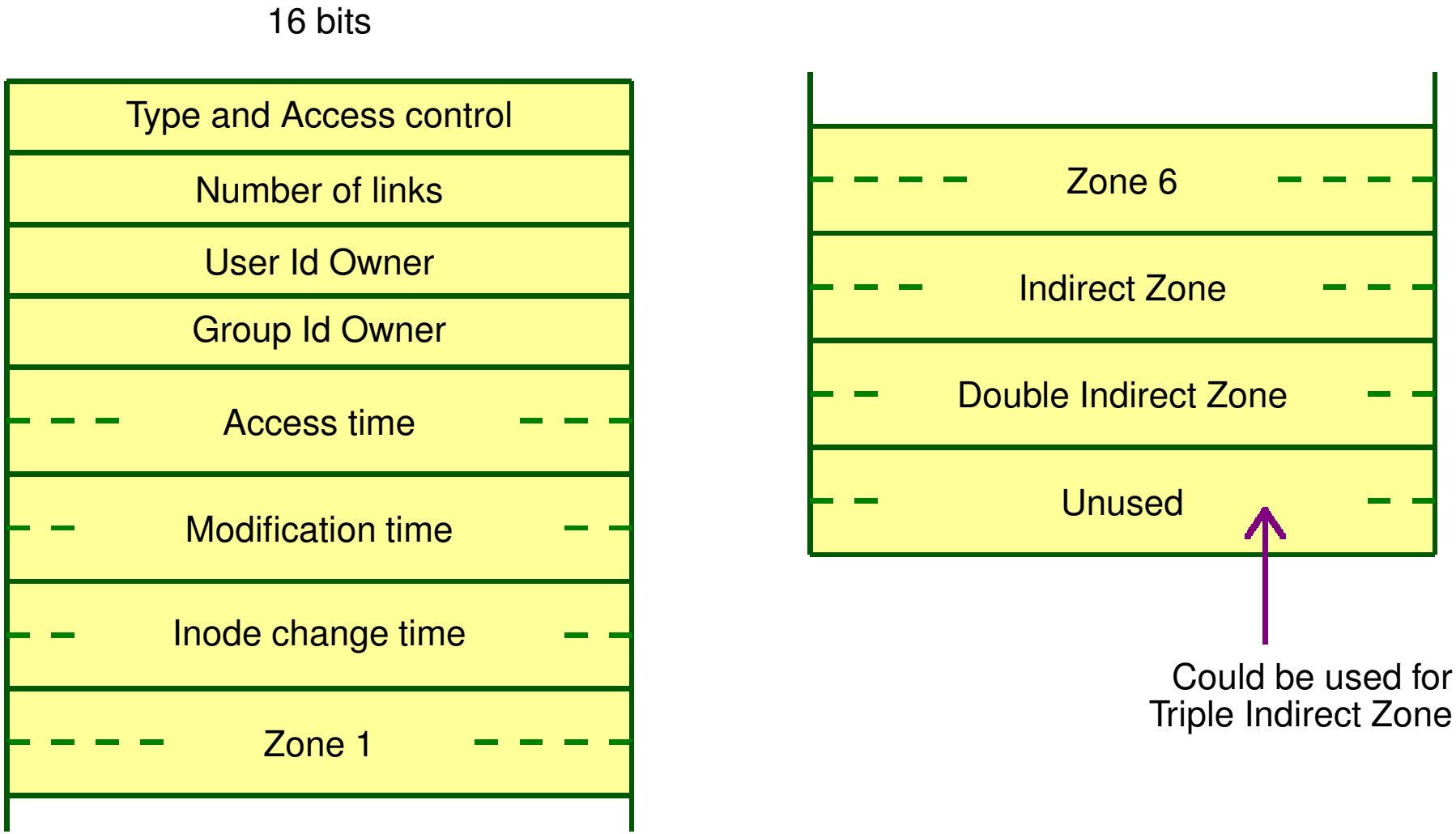


Minix/Unix disk space allocation

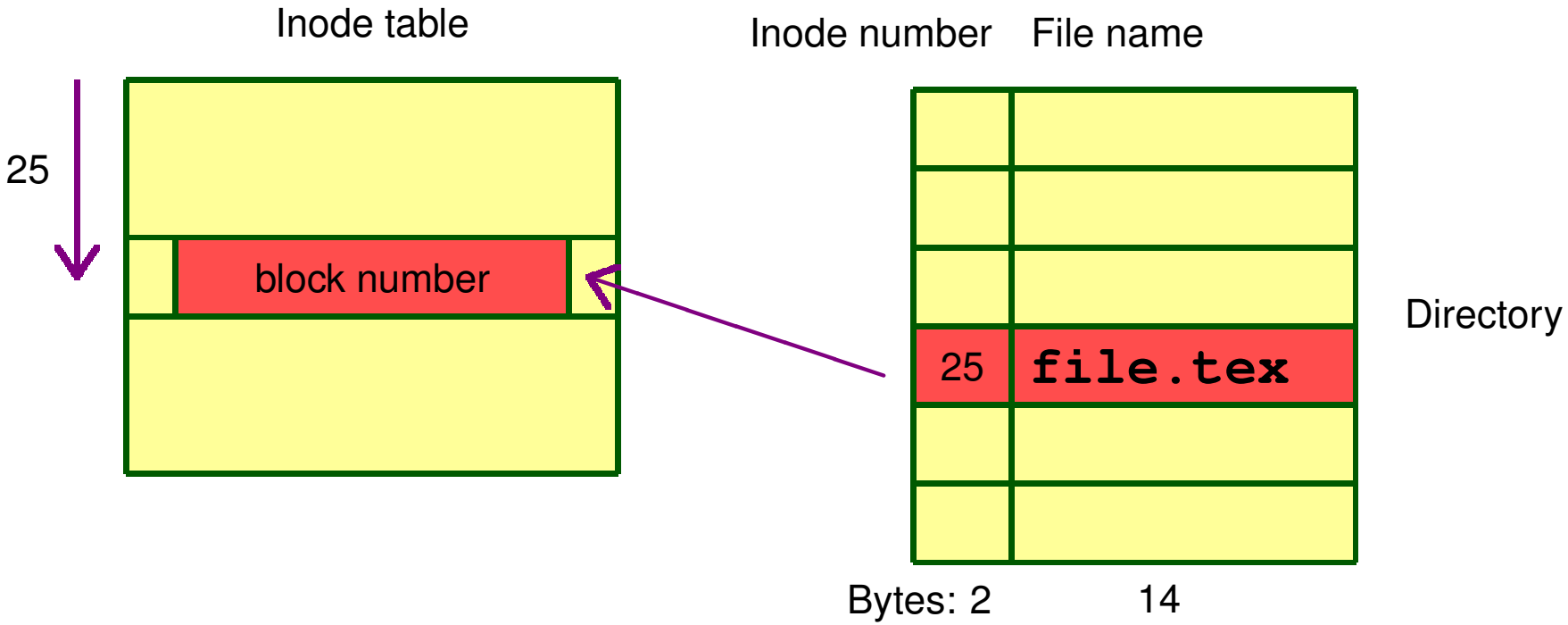
Minix uses 1Kbyte disk blocks. Each block of pointers contains 256 entries of 32 bit



Minix Inode


16 bits
Type and Access control
Number of links
User Id Owner
Group Id Owner
Access time
Modification time
Inode change time
Zone 1
Zone 6
Indirect Zone
Double Indirect Zone
Unused
Could be used for
Triple Indirect Zone

Minix Directory

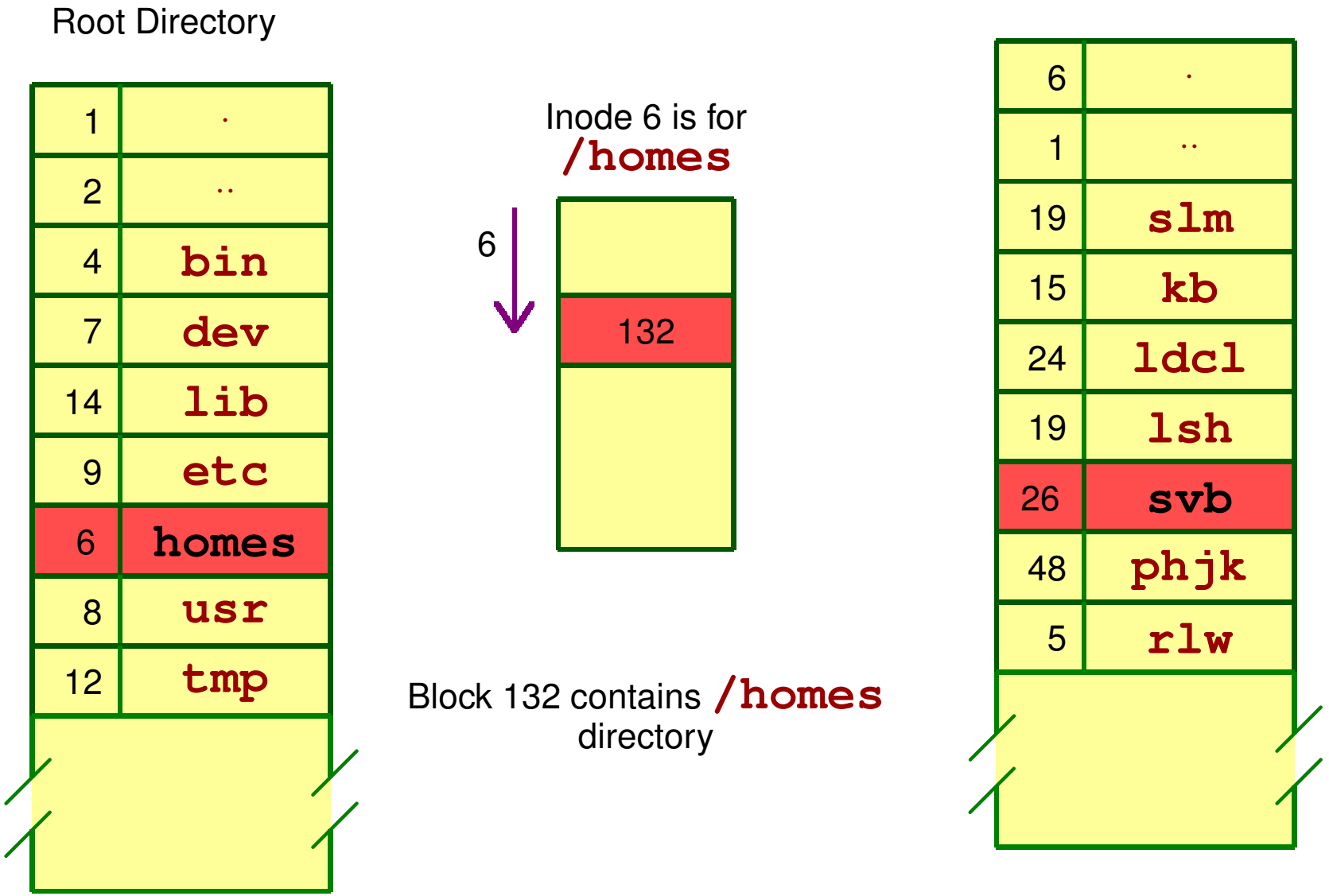


When file is opened, **OS** opens the inode table, and an (inode) entry is created in memory. This is as the inode on disk, but adds:

- inode number (for rewrite);
- number of processes which have opened file;
- major / minor device number for devices.

Directory Lookup

Looking for the file **/homes/svb/.email**.

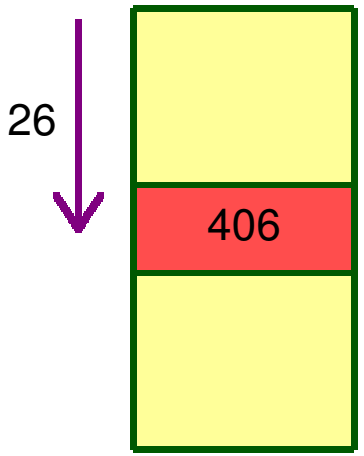


Directory Lookup (2)

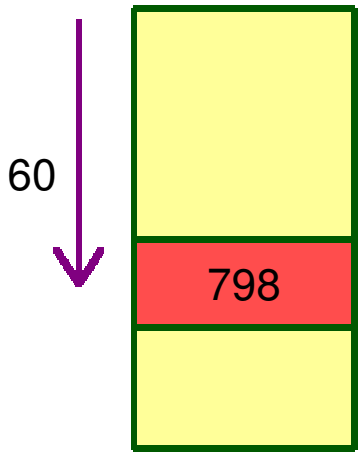
Block 406 contains
/homes/svb directory

26	.
6	..
13	Apps
23	bin
10	Lab
17	Linux
22	NeXT
60	.email
27	Public

Inode 26 is for
/homes/svb



Inode 60 is for
/homes/svb/.email



Block 798 contains inode of
/homes/svb/.email

Minix File Server Task lay-out

```
process fs ()  {
   initialise data structures;
    for (;; )  {
      source = receive_any (msg);
      switch(&msg.mode)  {
       case open:   get device descriptor;
          check access permissions and mode;
           if (device free )  allocate file descriptor
          Insert required information into descriptor;
          send (source, fd);
          break;
       case close:  ...  ;
       case read:   use fd  to find file descriptor;
          check request validity;
          set up devicemsg;
          send (device, &devicemsg);
          break;
       case write:  ...  ;
       case I/O_complete:   if error report it;
          find descriptor for device;
          send (userprocess, status);
    }
}
```

Unix I/O Calls

`fd = create (filename,permission)` **`fd`** is an index to a file descriptor, **`permission`** is used for access control. It opens the file for reading/writing.

`fd = open (filename,mode)` **`mode`** is either **`0`**, **`1`**, or **`2`** for **`read`**, **`write`**, or **`read/write`**

`close (fd)` Close a file or device.

`numbytesread = read (fd,buffer,numbytes)`

Read **`numbytes`** from file or device referenced by **`fd`** into memory **`buffer`**. Returns the number of bytes actually read in **`numbytesread`**.

Unix I/O calls (2)

`numbyteswritten = write(fd,buffer,numbytes)` Write **`numbytes`** to file referenced by **`fd`** from memory **`buffer`**. Returns the number of bytes actually written in **`numbyteswritten`**.

`fd = mknod (devname,ioclass,permission)`

Creates new special file.

`pipe (&fd[0])`

Creates a pipe; **`fd`** is an array of two integers: **`fd[0]`** is for reading, **`fd[1]`** for writing.

`newfd = dup (oldfd), dup2 (oldfd, newfd)`

Duplicate a file descriptor.

`ioctl (fd,operation,&termios)`

Used to control devices; **`&termios`** is an array of control characters.

The command `cat`

```
#include <stdlib.h>
#define BUFSIZE 512
int main(int argc, char **argv)
 {
   int fd, n, stdin, stdout, stderr;
   char buffer[BUFSIZE];

   stdin = 0;   /* Standard input always corresponds to fd = 0   */
   stdout = 1;   /* Standard output always corresponds to fd = 1   */
   stderr = 2;   /* Standard error always corresponds to fd = 2   */

   fd = open(argv[1], O_RDONLY);   /* Open file   */
   if (fd < 0)   {
      write(stderr, "Can't open file", 15);
    } else   {
      do   {
         n = read(fd, buffer, BUFSIZE);
         if (n < 0) write(stderr, "Error while reading", 19);
         else write(stdout, buffer, n);
       } while (n > 0);
    }
   close(fd);   /* Close file   */
 }
```

Redirection

Standard output `date > /dev/printer`: output of program goes to the printer.

Standard input `sort < file`: outputs the sorted version of `file` on the screen.

Both `sort < file1 > file2`: redirects input of program to `file1` and output to `file2`.

Standard error `myprog > file1 >& /dev/console`: redirects standard of program to `file1`, standard error to `console`.

Pipes `cat myprog.c | sort`: redirects output of the `cat` program to the input of the `sort` program.

File System Call

For the system call

```
fd = open(read,"/homes/svb/.email")
```

the Run-time system must perform the following function:

```
int open (int mode, char *name)
{
  message fmsg;
  fmsg.type = open;
  fmsg.name = name;     /*  pass pointer to name string */
  fmsg.mode = mode;
  fd = send_receive (FS,&fmsg);
  if (fd < 0)
    error(fd);
  else
    return(fd);
}
```

File System Task

```
process fs () {/* This process has id FS */
  fs_init();
  while (TRUE)  {
    source = receive_any(&msg);
    switch(msg.type) {
    case open:
      look up directory entry to get inode
      if (file already open)
        use inode table entry;
      else create new entry in memory inode table;
      if (shared position)
        use entry;
      else create new position entry;
      allocate file descriptor table entry;
      send (source, index of descriptor);
      break;
    case ...
  }
}
```

Archiving

Information on secondary storage can be lost through hardware or software failures: *backup* and *recovery*.

Periodic (total) Dump Time consuming; recovery of individual files by scanning entire dump tape.

Incremental Dump Only save files which have been *updated*, *changed* or *created* since the last dump. Set *flag* is set in directory entry on modification; global flag for the directory. Dumper (low priority process) checks directories for *flags set*, dumps modified file and *clears* the flag. Performed frequently; generates large number of dumps. (Combine periodic and incremental dumps.)

Recovery Work *backwards* through *incremental dump tapes* restoring any dumped file *except* one already restored. Finally use last Periodic Dump.

Information per File

- *inode* number;
- *major* and *minor* device number;
- *access control* permissions;
- current *status* (allocated / free / out of service);
- current *user* (if allocated);
- *device handler* process identifier;
- *error handling* procedures (who to send what messages).

Memory Management

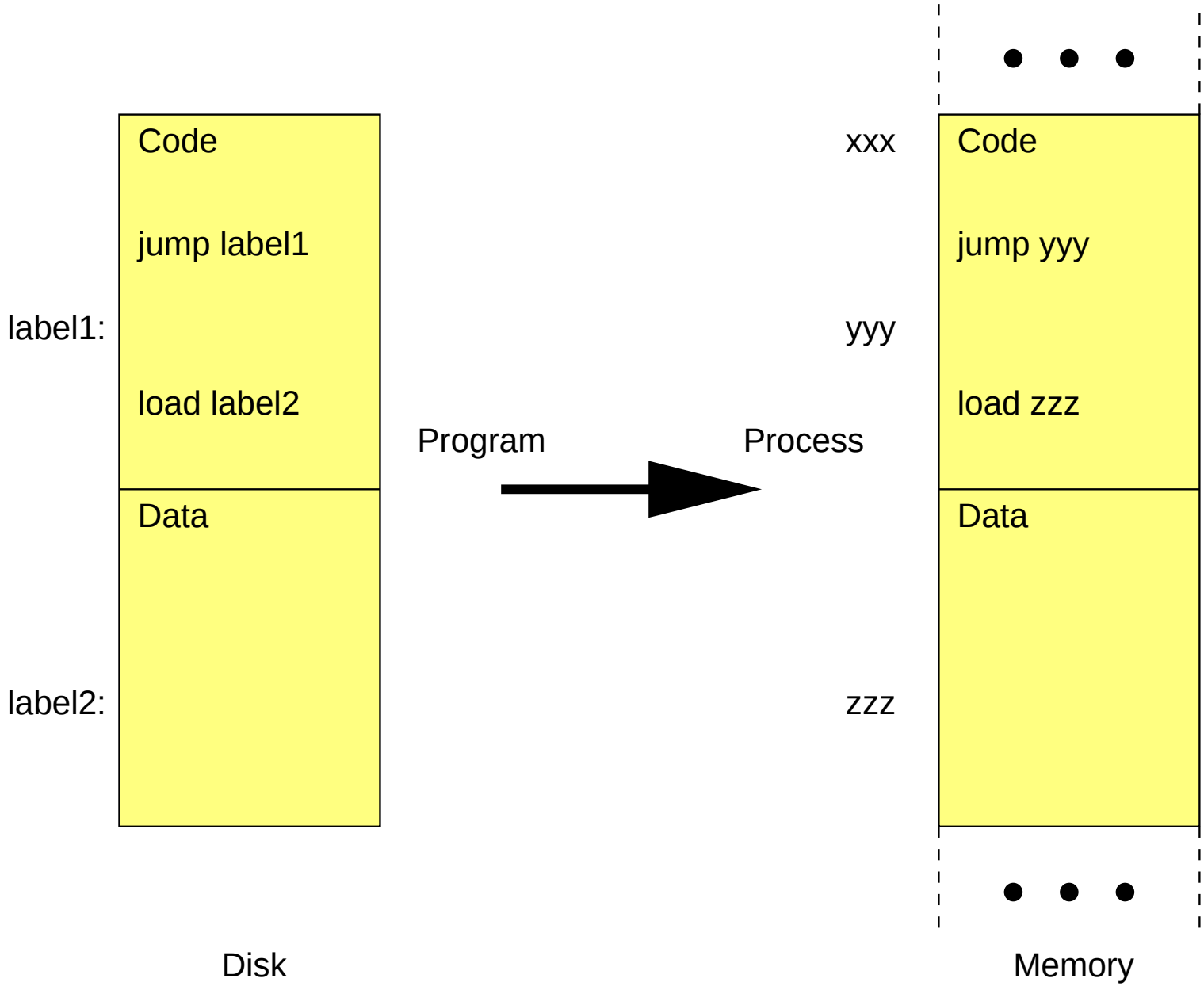
Memory Management issues

- Memory address binding:
  - compile time;
  - load time;
  - run time;
  - relocation.
- Memory partitioning:
  - fixed partitioning;
  - dynamic partitioning;
  - buddy system.
- Virtual memory.
- Paging.
- Segmentation.

Relocation and Address Binding

- A program is loaded from disk into memory; executed as a process.
- Symbolic memory addresses of program (for both instructions and code) must be translated into physical memory addresses of process.
- Address binding requires knowledge about the location of process in memory.
- Address binding may be necessary at load time or run time:
  - processes may be placed into different memory location at load time;
  - processes may be relocated from one memory location to another memory location at run time, i.e. due to swapping.

Relocation and Address Binding (2)



Address Binding: Compile Time

When the program is created – during compilation all symbolic memory addresses are translated to absolute addresses (i.e. to physical memory addresses);

– during loading and execution, no further intervention from **OS** is needed.

Advantages – programs can be loaded and executed without modifications.

Disadvantages – programs can not be loaded at arbitrary locations in memory;

– programs can not be dynamically relocated during run time.

Programmer or compiler requires knowledge about the internal memory management schemes of the **OS**.

Address Binding: Load Time

When the program is loaded in memory – during compilation all symbolic memory addresses are translated to relative addresses (i.e. to the beginning of the program);

- during loading all relative memory references are replaced by absolute memory references (i.e. by adding the starting address of the program in memory);
- during execution no further intervention from **OS** is needed.

Advantages programs can be loaded at arbitrary locations in memory.

Disadvantages programs can not be dynamically relocated during run time.

Programmer or compiler requires no knowledge about the internal memory management schemes of the **OS**.

Address Binding: Execution Time

At run time, i.e. while the process is executed – during compilation all symbolic memory addresses are translated to relative addresses (i.e. to the beginning of the program);

- during loading no intervention from **OS** is needed;
- during execution each encountered relative memory references is replaced by absolute memory references (i.e. by adding the starting address of the program in memory).

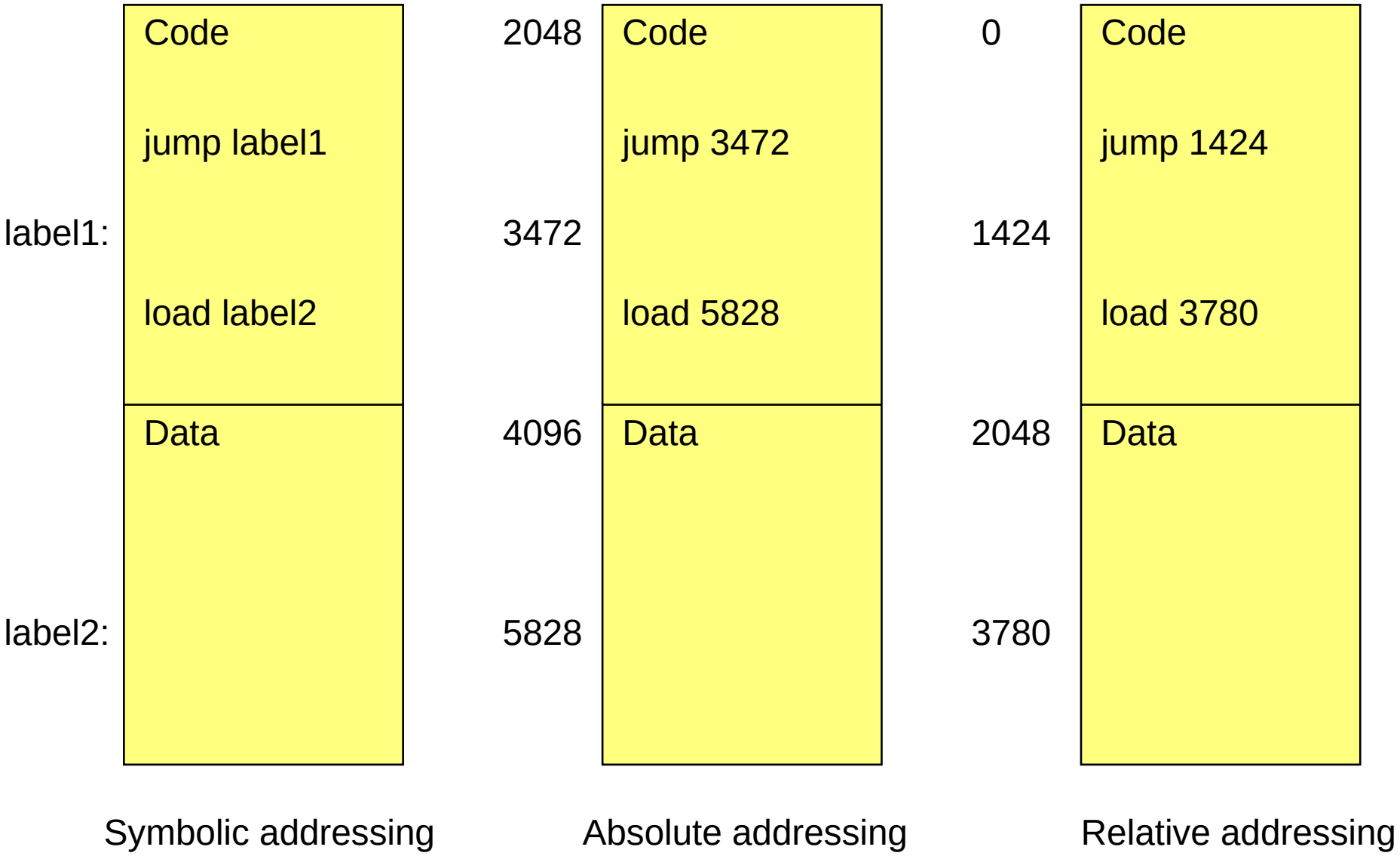
Advantages – programs can be loaded at arbitrary locations in memory;

- programs can be dynamically relocated during run time.

Disadvantages – requires hardware support for fast address translation.

Programmer, compiler and loader require no knowledge about the internal memory management schemes of the **OS**.

Memory Addressing



Memory Addressing (2)

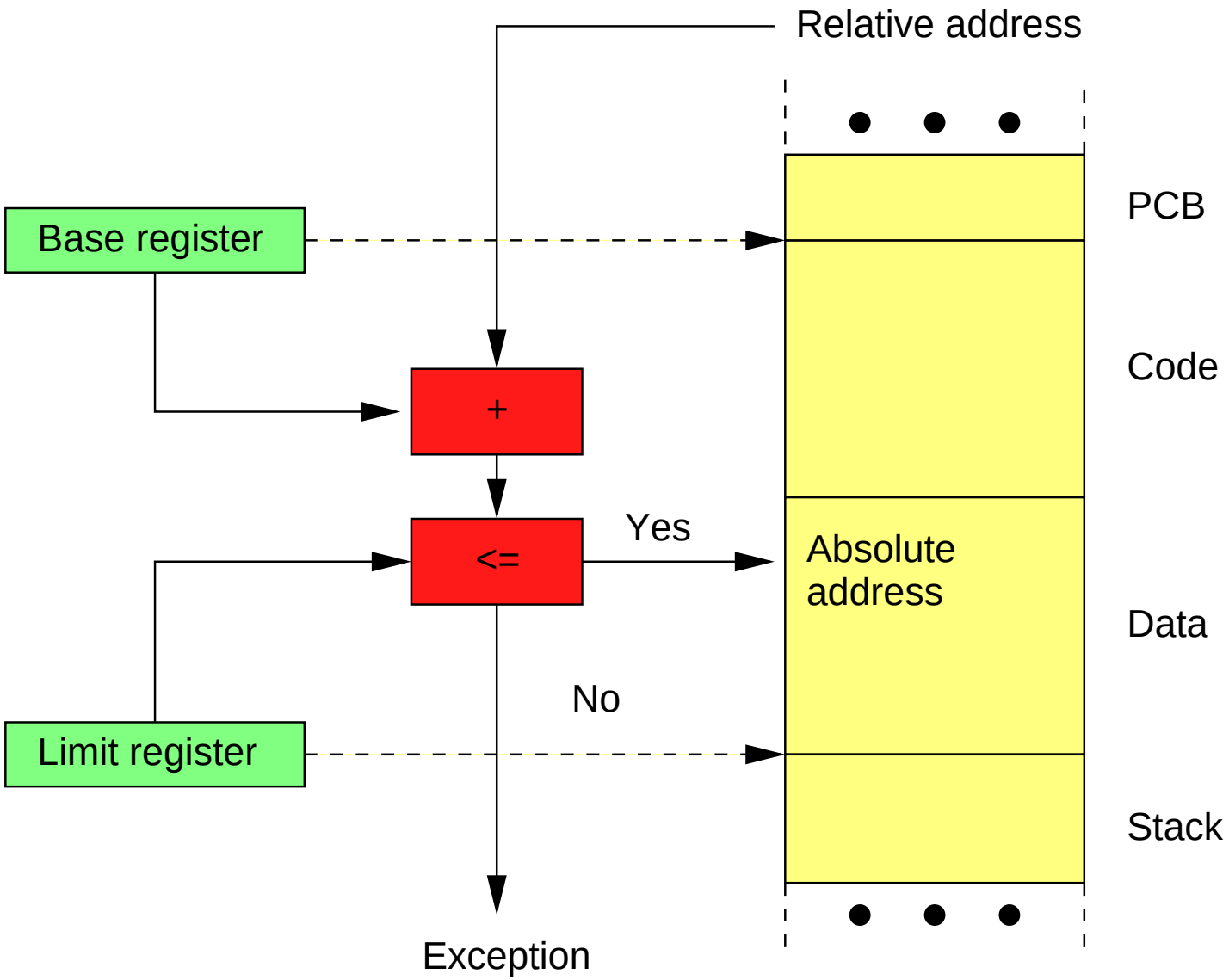
Symbolic addresses – must be translated to relative or absolute addresses at compile time.

Relative addresses – can be generated at compile time;
- must be translated to absolute addresses at load or run time;
- program which contains relative addresses is *relocatable*.

Absolute addresses – can be generated at compile time;
- can be generated at load time;
- can be generated at run time (requires hardware).

Virtual addresses – used for virtual memory;
- discussed later.

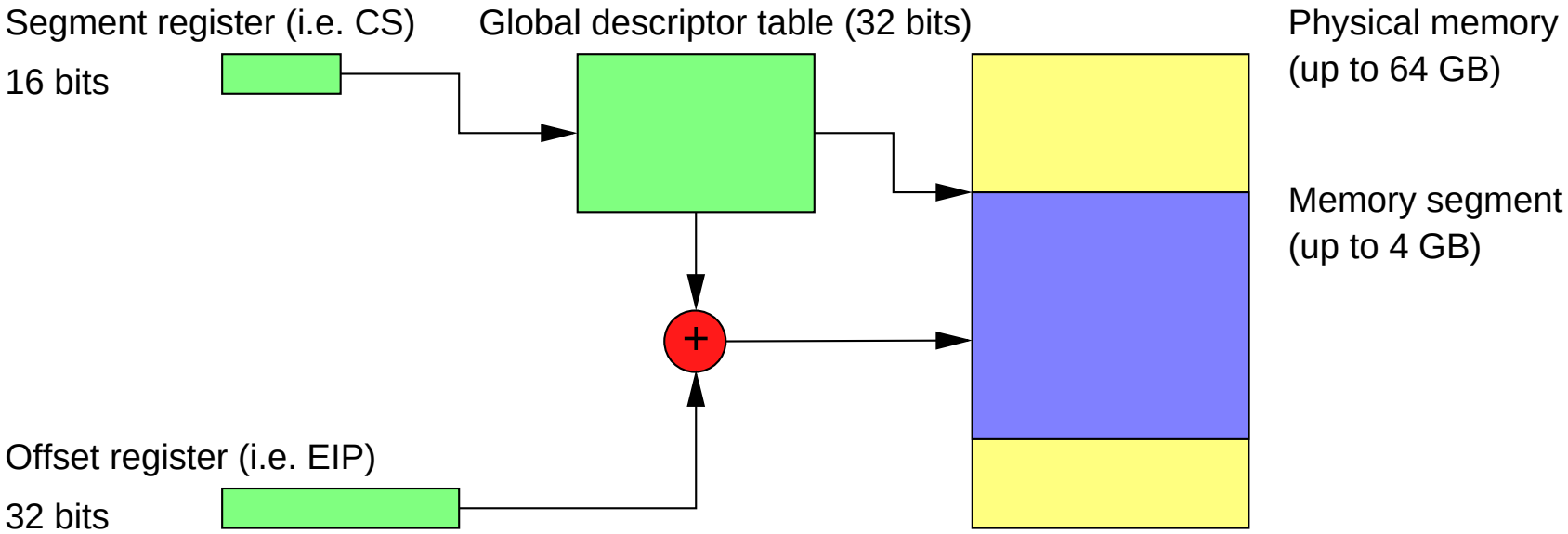
Memory Addressing (3)



Intel Pentium

EIP contains the offset in the current code segment (pointed at by **CS** register) for the next instruction to be executed. The **EIP** register can only be modified by instructions such as **JMP**, **CALL** or **RET**.

To fetch the next instruction the **EIP** and **CS** register are needed.



Memory Partitioning

- Memory is split into two areas:
  - area for resident part of **OS**;
  - area for resident part of user processes (can be divided into partitions).

Single partition – allows only one user process in memory;
  - simple memory management but inefficient.

Multiple partitions – allows more than one user process in memory;
  - more complex memory management but more efficient.

Partitioning schemes – fixed partitioning;
  - dynamic partitioning.

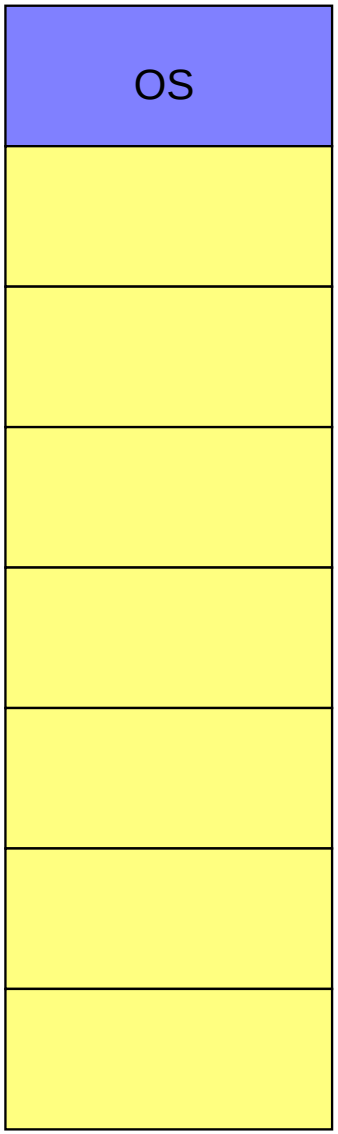
Problems – internal fragmentation;
  - external fragmentation.

Fixed Partitioning

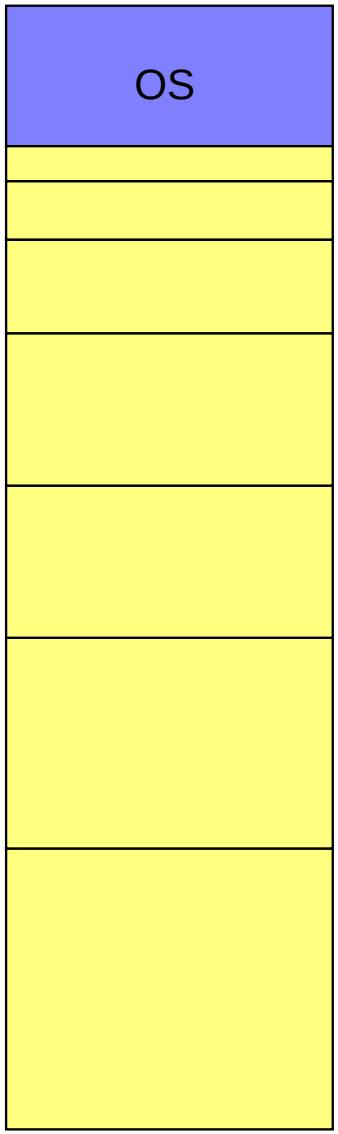
- Memory divided into fixed number of partitions with fixed size:
  - each partition is either allocated or not allocated;
  - each partition which is not allocated forms a *hole*;
  - holes may be scattered in memory.
- **OS** maintains list of allocated and unallocated partitions:
  - if a process is created, the **OS** allocates a partition large enough to accommodate it;
  - if a process is terminated, the **OS** frees the partition associated with this process.
- Partitions can be:
  - equal-sized;
  - unequal-sized.
- Processes occupy exactly one partition: if a partition is larger than the process, the remaining memory is wasted: internal fragmentation.

Fixed Partitioning (2)

Equal-size partitions



8 MB
8 MB
8 MB
8 MB
8 MB
8 MB
8 MB



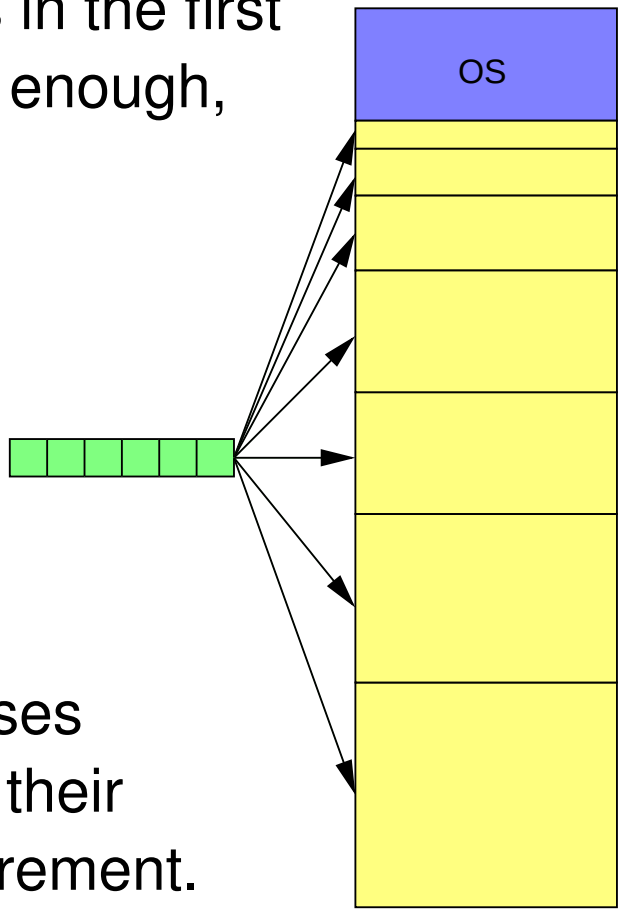
8 MB
2 MB
4 MB
6 MB
8 MB
8 MB
12 MB
16 MB

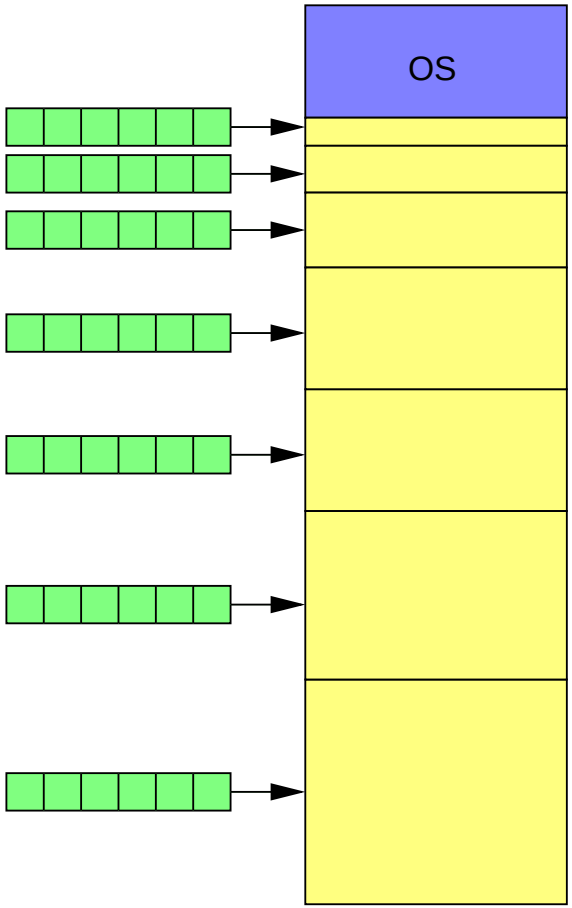
Unequal-size partitions

Fixed Partitioning (3)

Now either:

- run processes in the first slot that is big enough, or





- queue processes depending on their memory requirement.

Dynamic Partitioning

- Memory is divided into a variable number of partitions with variable size:
  - at the beginning the entire memory forms a single partition;
  - as time goes on new partitions are created and deleted dynamically.
- **OS** maintains a list allocated and unallocated memory partitions:
  - if a process is created, the **OS** allocates a partition of the requested size;
  - if a process terminates, the **OS** frees the partition associated with this process.

Dynamic Partitioning (2)

- **OS** uses one of the following methods to satisfy memory requests:

  First fit allocate the first hole that is big enough;

  Best fit allocate the smallest hole that is big enough: **OS** must search entire list of holes, unless list is ordered: produces the smallest left-over hole;

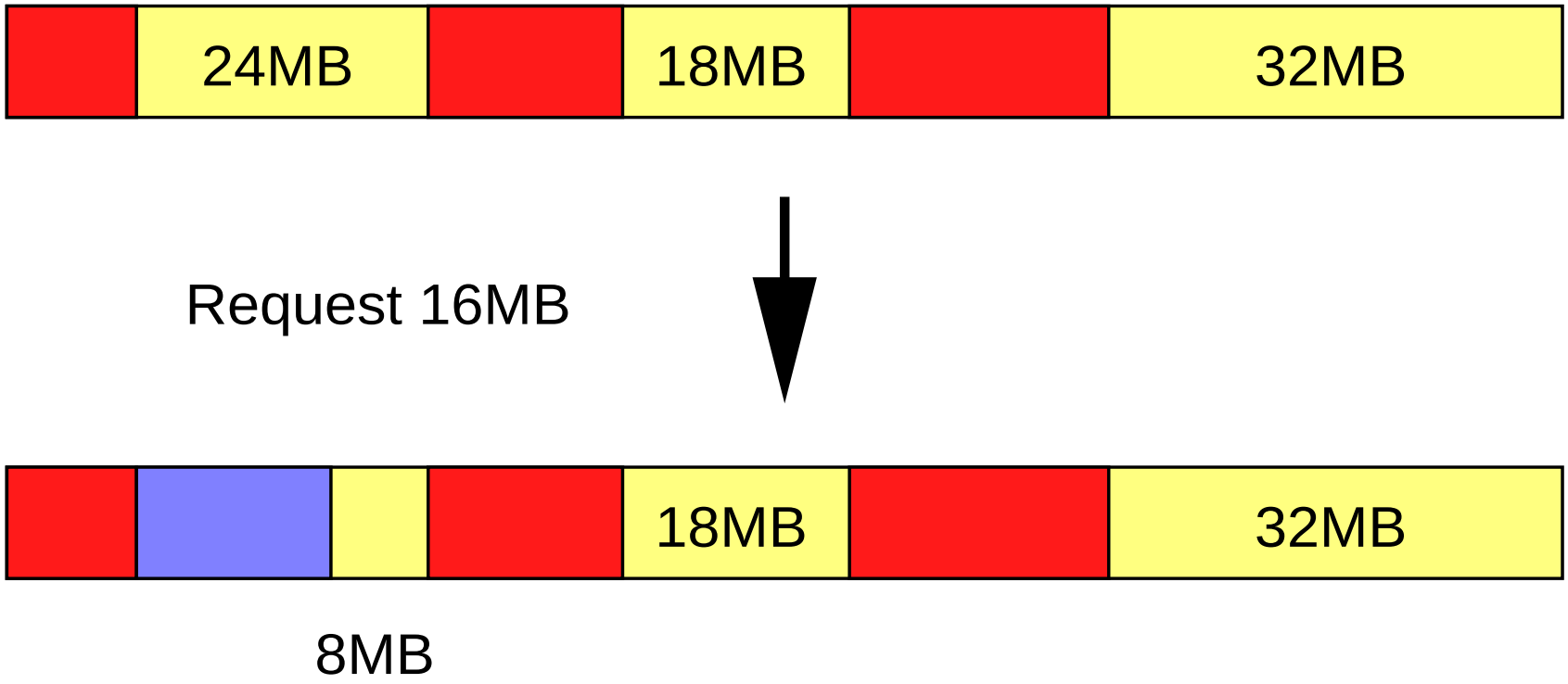
  Worst-fit allocate the largest hole that is available: **OS** must search entire list of holes, unless list is ordered: produces the largest left-over hole.

- First fit and best fit are better than worst-fit in terms of memory utilisation.

First Fit

- **OS** maintains a list of holes $M_1, \cdots, M_n$ in order of increasing base address: $M_1 > \cdots > M_n$

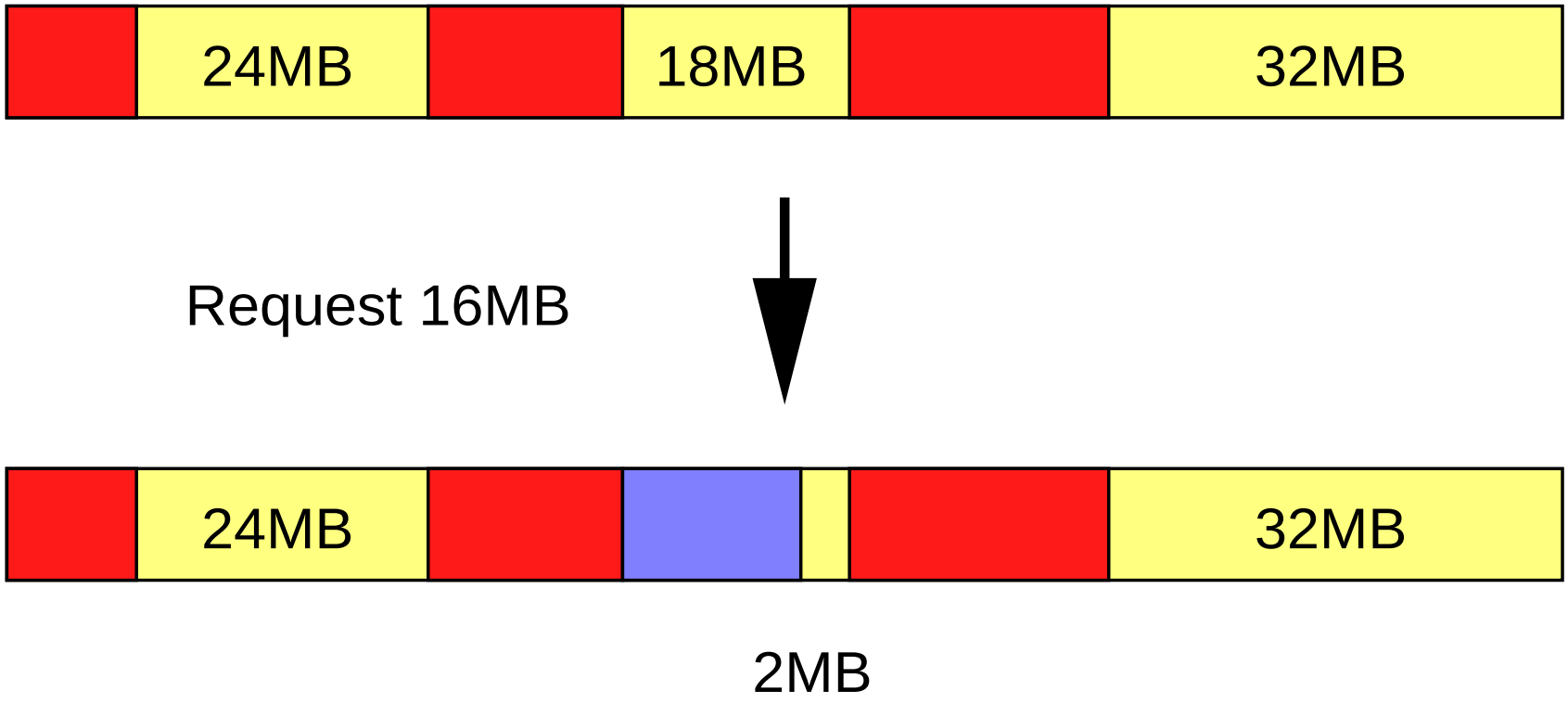
$$L = \text{(24MB, 18MB, 32MB)}$$



Best Fit

- **OS** maintains a list of holes $M_1, \cdots, M_n$ in order of increasing size $M_1 \leq \cdots \leq M_n$:

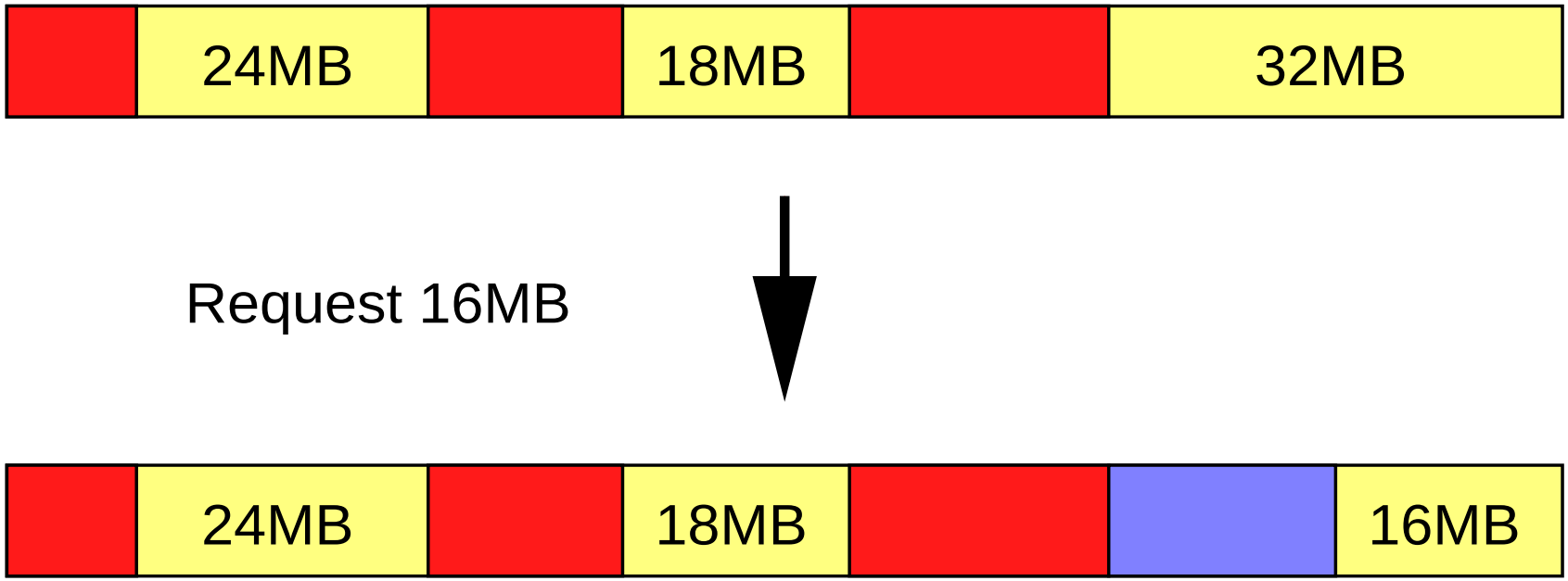
$$L = \text{(18MB, 24MB, 32MB)}$$



Worst Fit

- **OS** maintains a list of holes $M_1, \cdots, M_n$ in order of decreasing size $M_1 > \cdots > M_n$:

$$L = \text{(32MB, 24MB, 18MB)}$$



Dynamic Partitioning

Fragmentation and Compaction

Fixed partitioning – each process occupies an entire partition;

- if the process is smaller than the partition, memory is wasted: internal fragmentation.

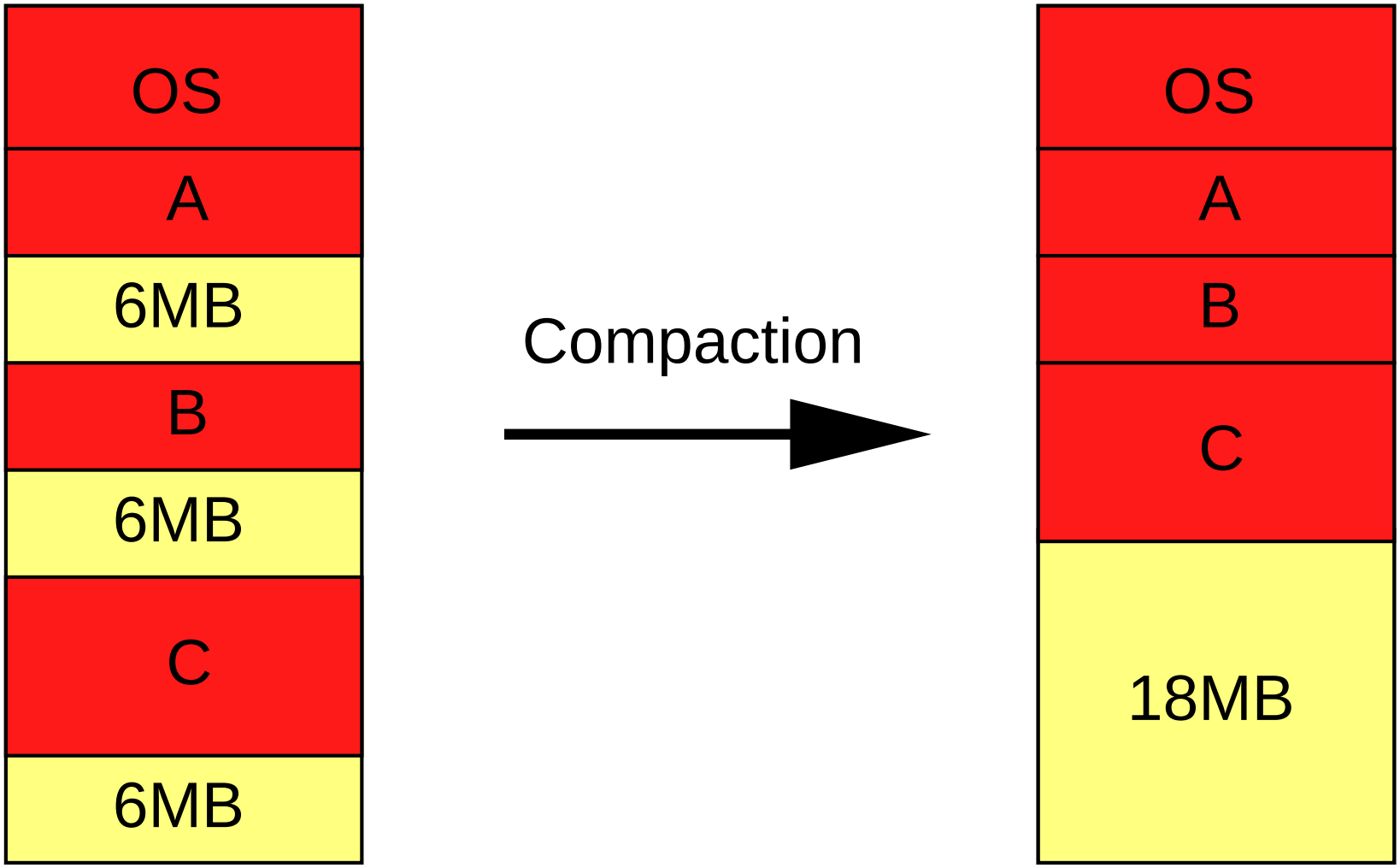
Dynamic partitioning – each process occupies a partition exactly the size of the process;

- if the process terminates it leaves a hole of unused memory: as time goes on more and more smaller holes appear: external fragmentation.

Compaction – can reduce external fragmentation;

- collect all free memory holes and merge into a single hole;
- requires dynamic relocation because memory addresses of process can change during execution.

Fragmentation and Compaction (2)



Buddy System

Problems – fixed partitioning leads to internal fragmentation;
– dynamic partitioning leads to external fragmentation.

Solution Buddy system: compromise between fixed and dynamic partitioning schemes.

- Memory is divided into buddies:
  – each buddy is a block of size 2^k ($min \leq k \leq max$);
  – 2^{min} is size of the smallest block which can be allocated;
  – 2^{max} is size of the largest block which can be allocated: total amount of available memory.
- Buddy system can lead to internal fragmentation, but internal fragmentation is minimised because:
  – buddies can have variable size;
  – buddies can be generated dynamically.

Buddy System (2)

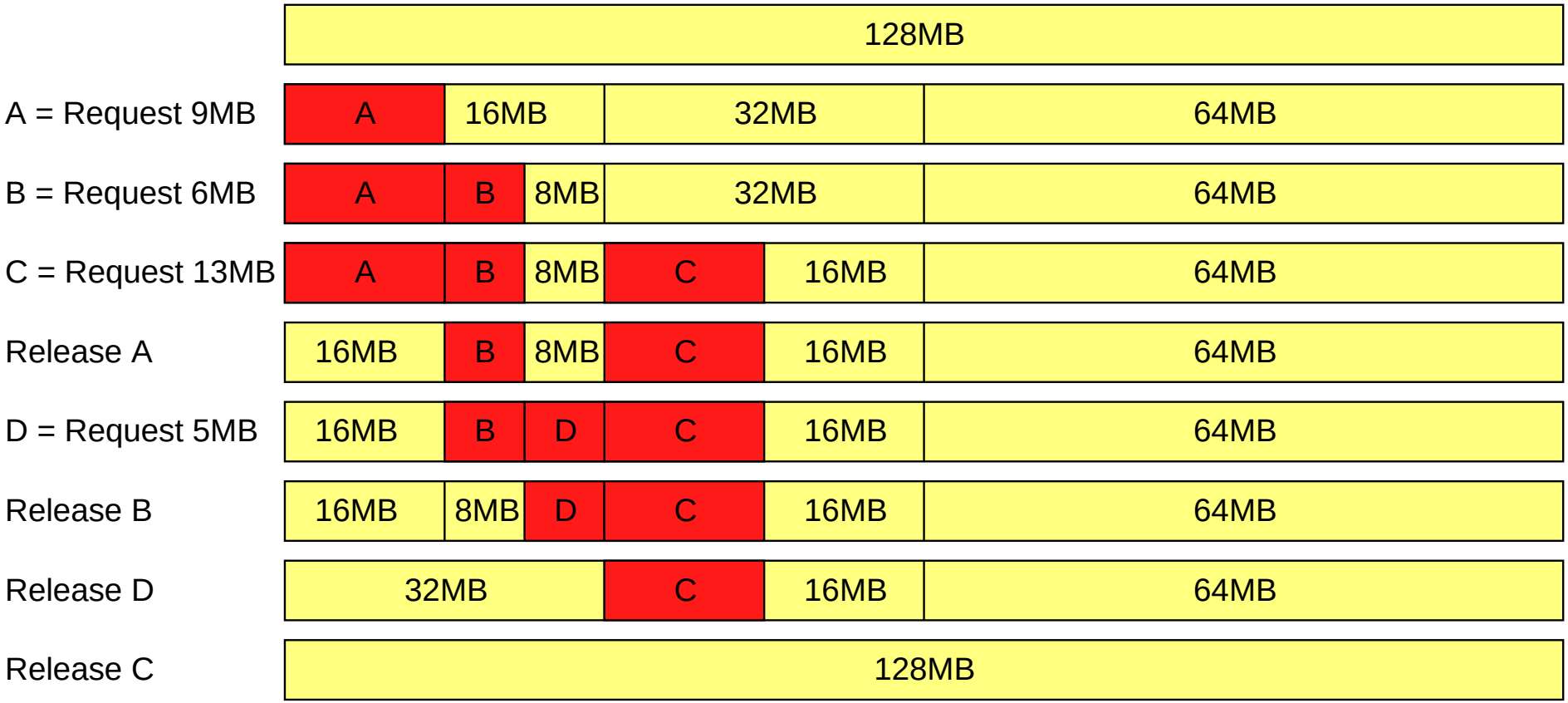
- Start with entire memory: a single block of size $2^i = 2^{max}$.
- When a request of size s is made:
  - if $2^{i-1} < s \leq 2^i$ then allocate entire block of size 2^i;
  - else split block in two new buddies of size 2^{i-1};
  - if $2^{i-2} < s \leq 2^{i-1}$ then allocate one of the two buddies'
  - otherwise repeat process by splitting one of the two buddies.
- Buddies are repeatedly subdivided until buddy of size greater or equal to s is generated.
- Buddies can be merged if two adjacent buddies become free.
- **OS** maintains a number of lists with available buddies:
  - list L_i contains a list of available buddies of size 2^i;
  - if a pair of available buddies occurs in list L_i they are merged and moved as a single available buddy into list L_{i+1}.

Buddy System (3)

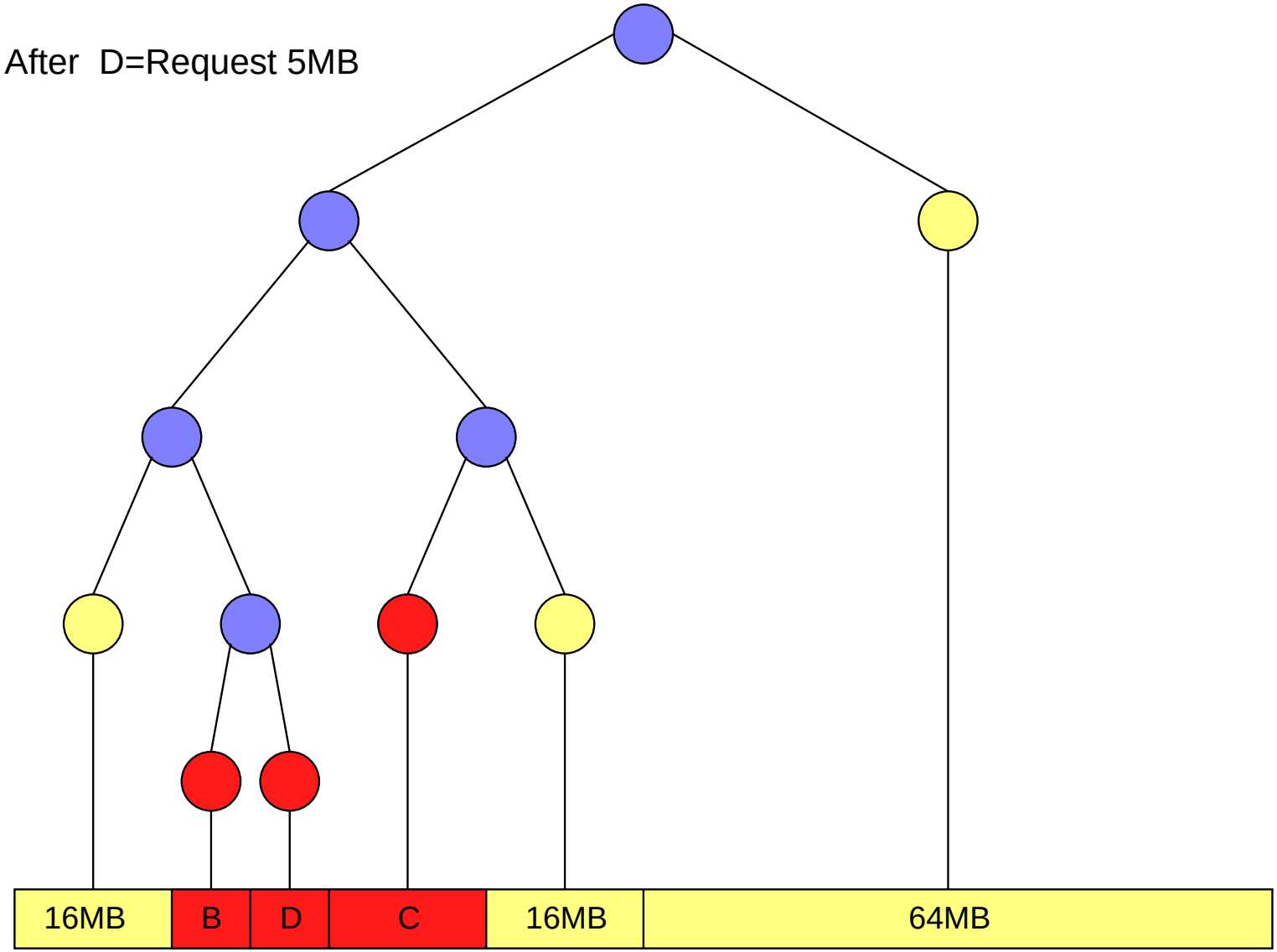
The **OS** performs the following steps if a request for memory of size s comes in, with $2^{i-1} < s \leq 2^i$:

```
procedure get_hole(i)
  if (i = kmax)
    return failure
  if (list Li is empty)
    get hole(i+1)
    split hole into buddies
    put buddies on list Li
  take first hole on list Li
end get_hole
```

Buddy System (4)



Buddy System (5)



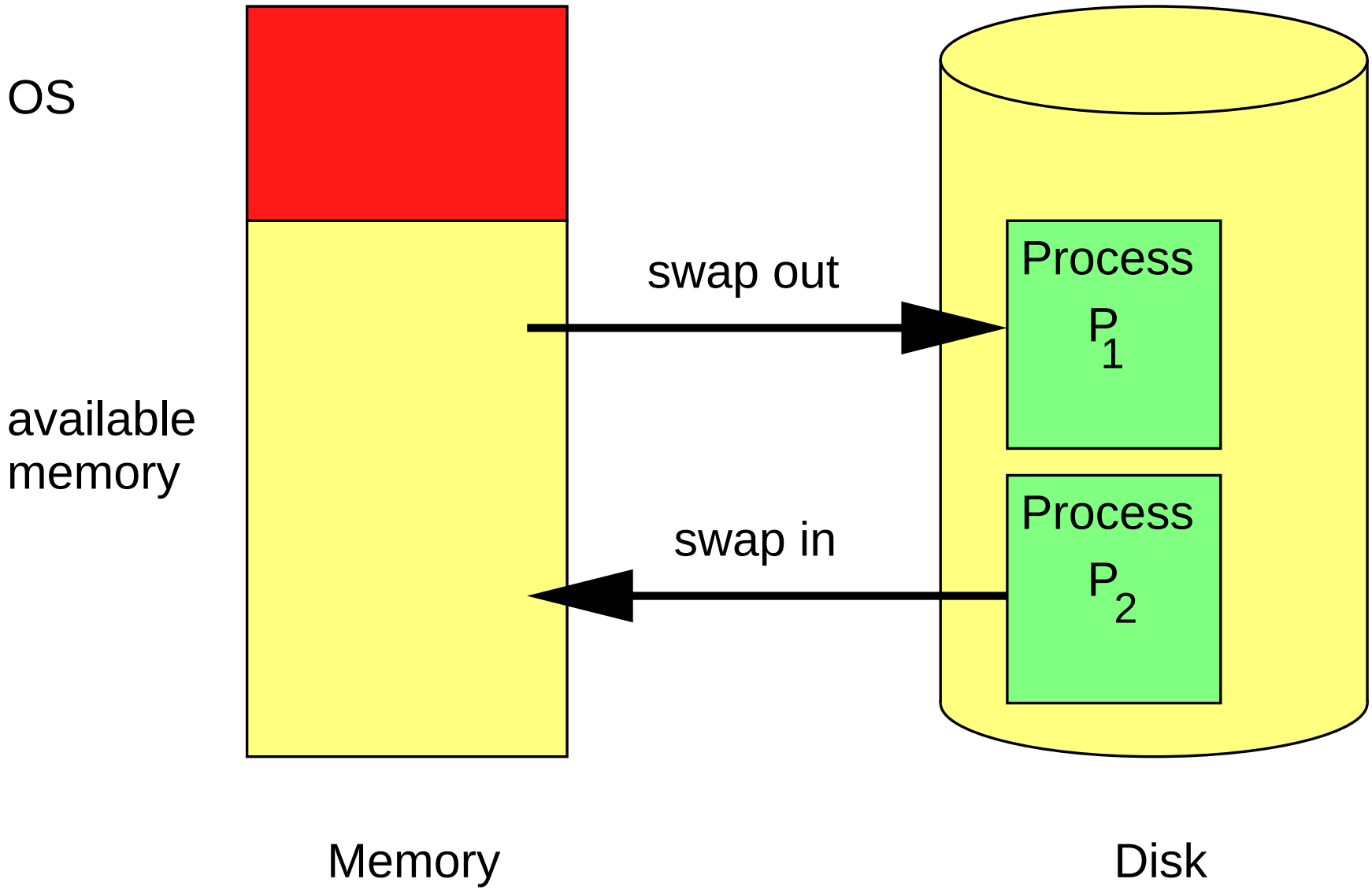
Swapping

- A process can be temporarily swapped out of memory to a backing store, and later on brought back into memory for continued execution.
- Backing store is usually on a fast hard disk which is large enough for copies of the memory images of all processes.
- Backing store provides access functions:
  - to swap out process images;
  - to swap in process images.

Advantages The number processes is not limited by the amount of memory but by the amount of backing store memory.

Disadvantages The size of the largest processes is still limited by the amount of memory.

Swapping (2)


OS
available memory
swap out
swap in
Process P_1
Process P_2
Memory
Disk

Paging

Problems Traditionally,

- the entire process must be resident in memory when executing;
- the entire process must be smaller than the physical memory;
- the entire process must occupy a single contiguous memory area.

Solution – use virtual memory;

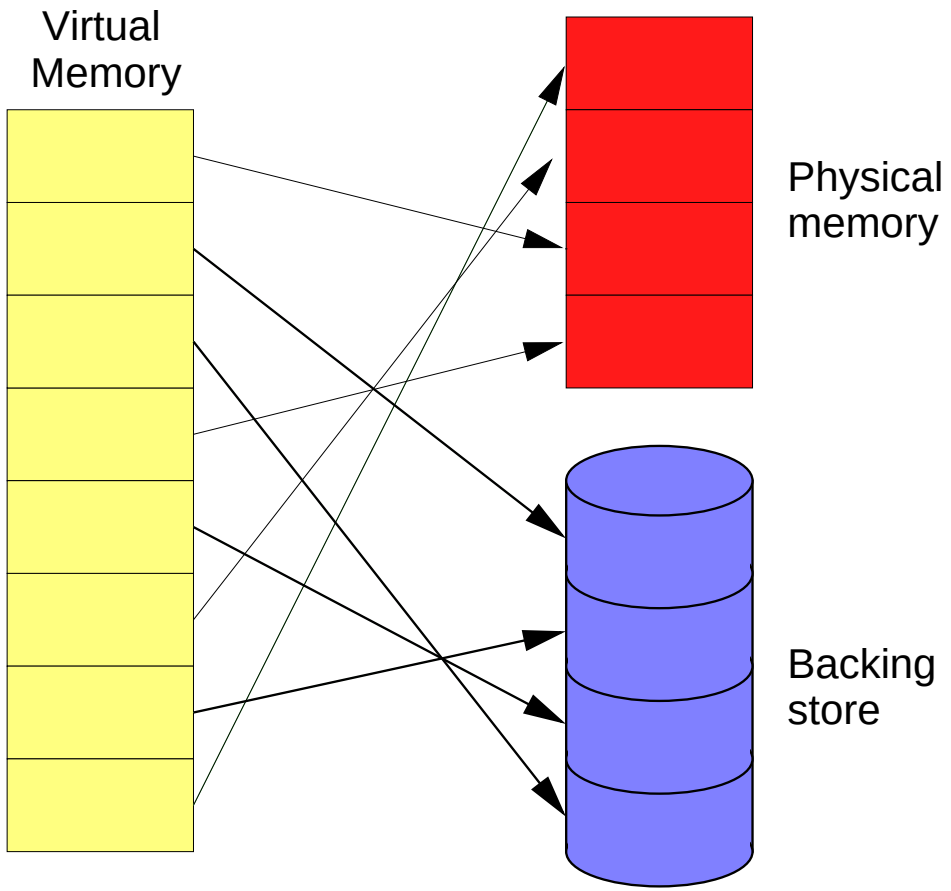
- map contiguous virtual memory to discontiguous physical memory chunks;
- map contiguous virtual memory onto physical memory and backing store.

- Virtual memory can be larger than physical memory.
- Physical memory does not need to contain entire process.

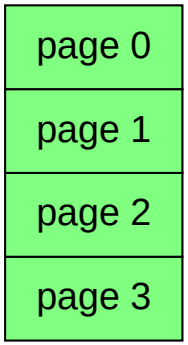
This requires a change in the hardware, and in the microprogramming.

Paging (2)

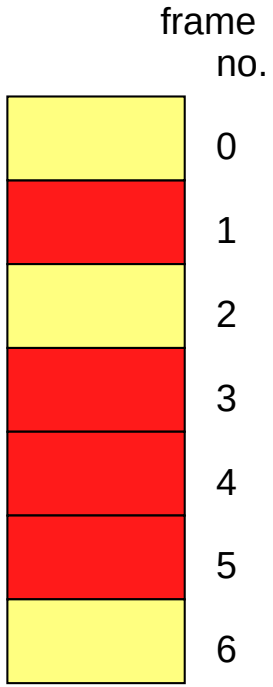
- Physical memory is divided into fixed size blocks called *frames*.
- Virtual memory is divided into fixed size blocks of the same size called *pages*.
- Size of frames and pages is power of 2, i.e. 2^i usually between 512 and 8192 bytes.
- Paging requires:
  - address translation;
  - page fault management;
  - page replacement.



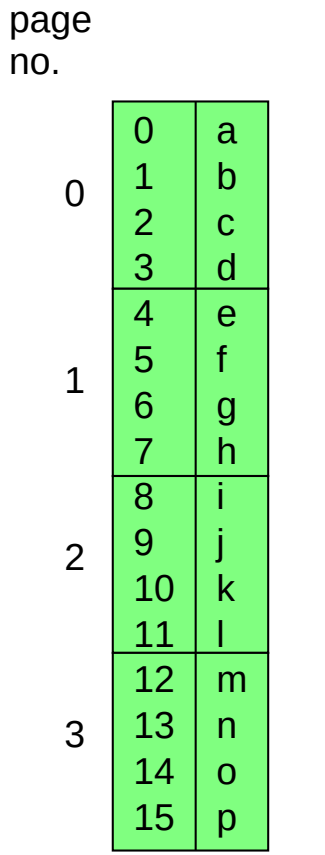
Paging (3)



page	frame
0	1
1	4
2	3
3	5

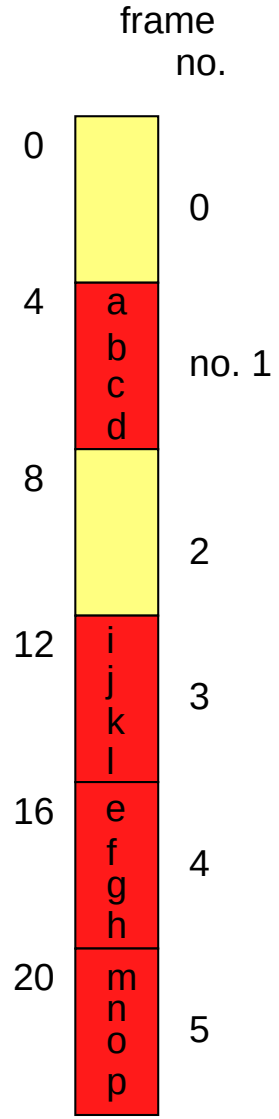


physical memory



Virtual address

page	frame
0	1
1	4
2	3
3	5



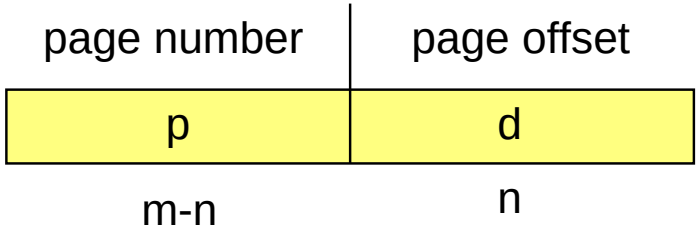
Physical address

Paging (4)

- Any virtual address generated by the processor is divided into:

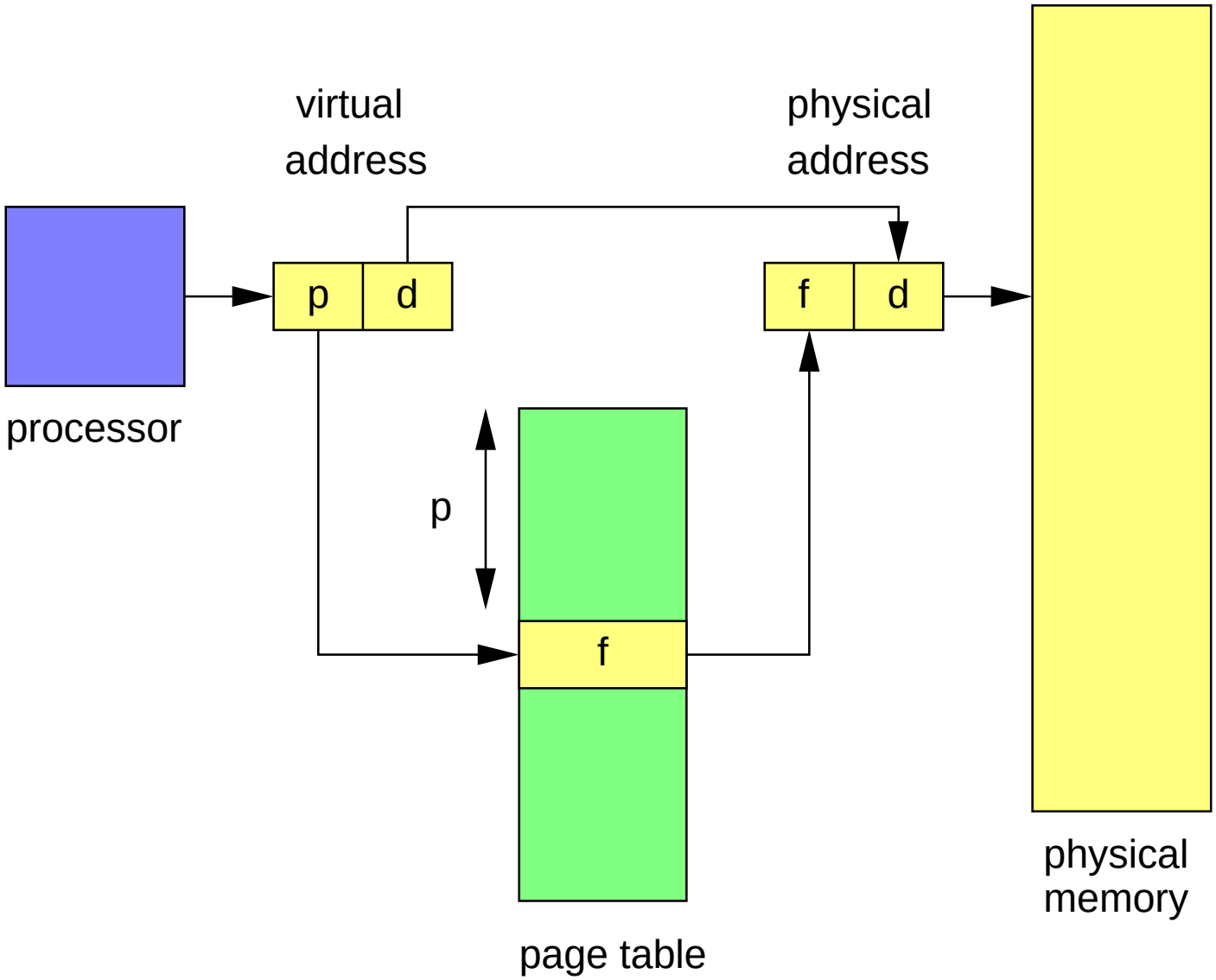
  Page number (p) used as an index into a page table which contains the base address of each page in physical memory;

  Page offset (d) used in combination with the base address to define the physical memory address that is sent to the memory unit.



- n is the number of bits per offset: 2^n is the page size.
- m is the number of bits per memory address: 2^m is the size of the virtual memory.

Paging (5)


virtual
address
physical
address
p
d
f
d
processor
p
f
page table
physical
memory

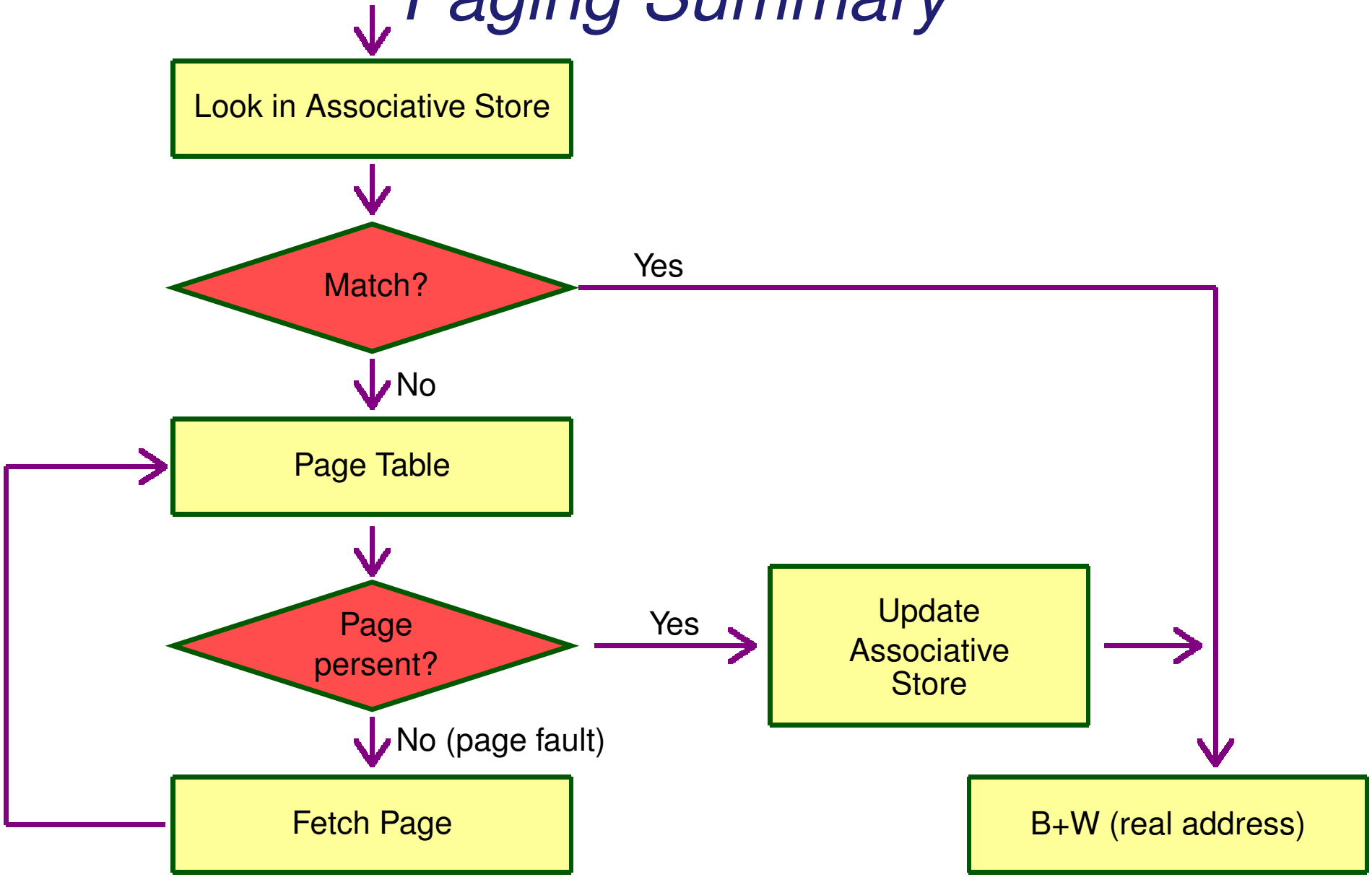
Paging (6)

- Each process has its own page table:
  - ensures protection;
  - allows relocation.

  Memory access is transparent to programmer and compiler.
- Each entry in the page table contains the base address of the page in memory plus:

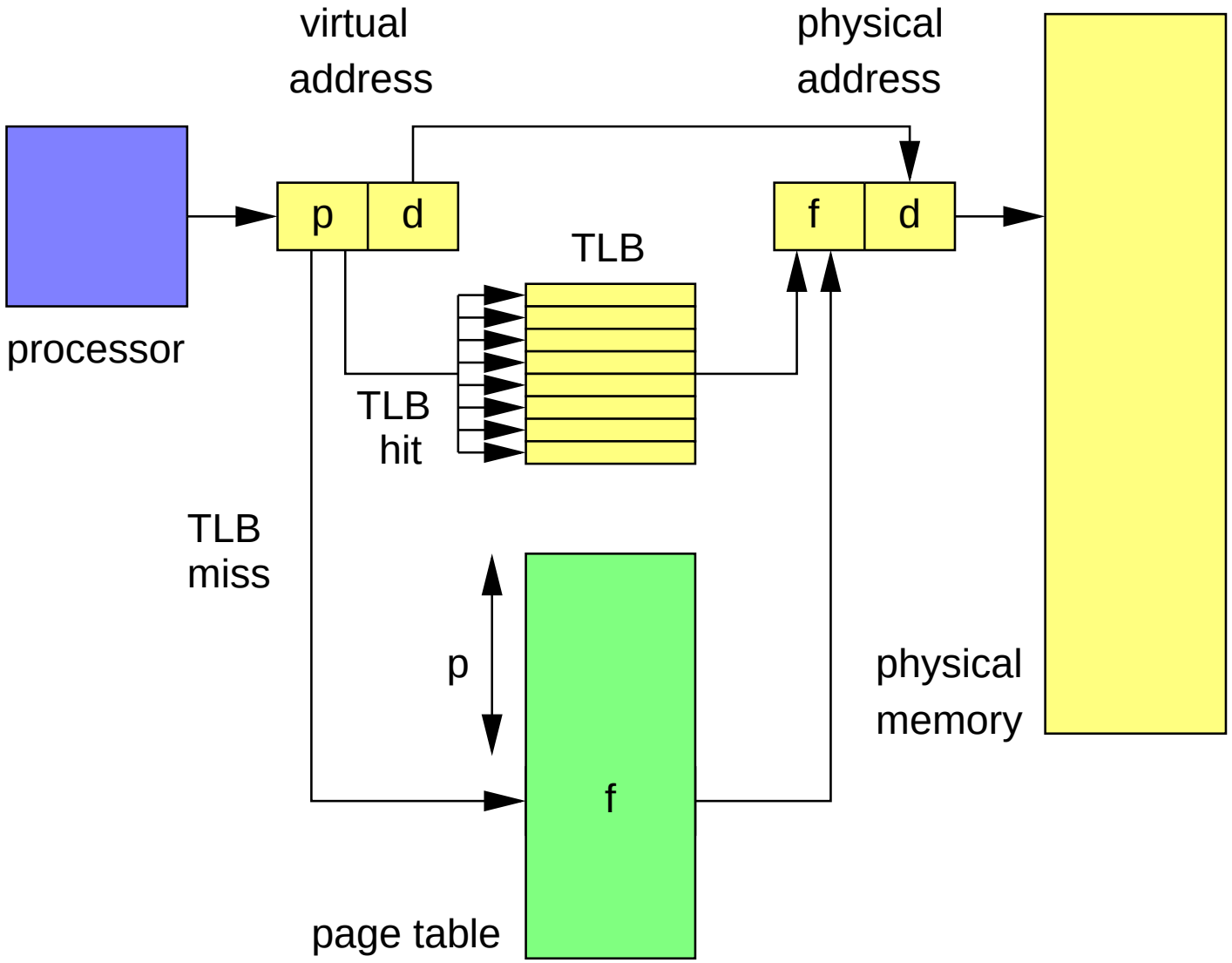
  valid bit used to indicate whether the page is in memory;
  write bit used to indicate whether the page has been modified while in memory.
- Small page size:
  - less internal fragmentation;
  - more efficient memory use.
- Large page size:
  - less overhead for address translation;
  - faster memory access.

Paging Summary



Hardware Support

Reduced access time via *translation look-aside buffer* (TLB):

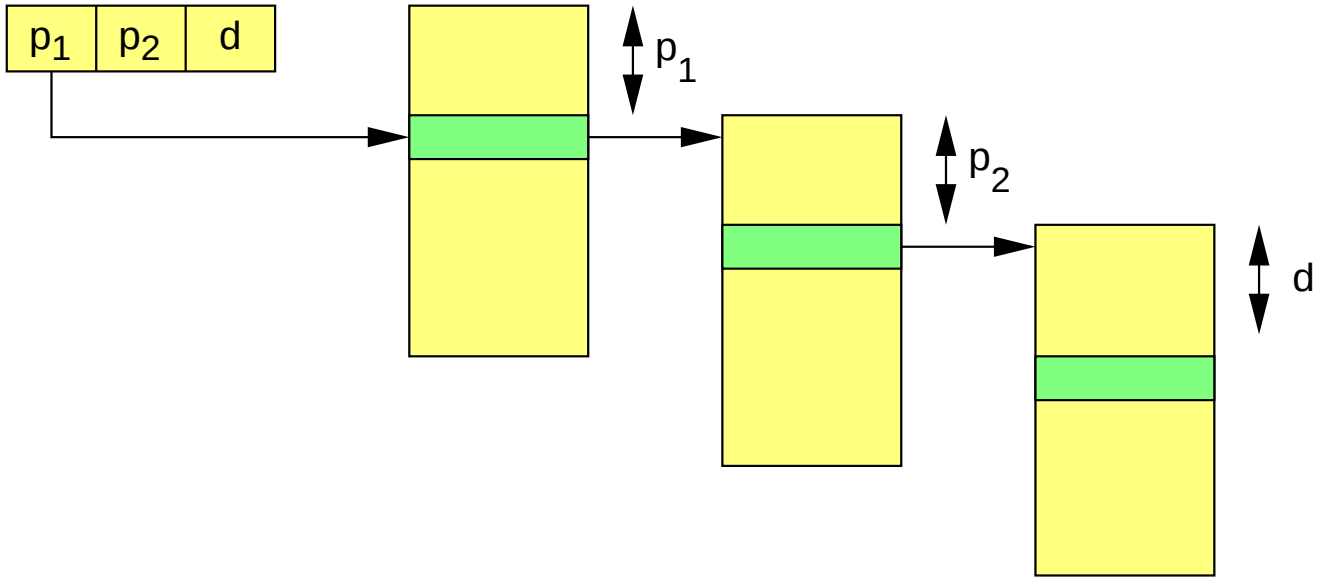


Multi Level

- Modern hardware supports a large address space, i.e. between 2^{32} and 2^{64}.

Problem page tables can become very large, i.e. for a page size of 2^{12} and an address space of 2^{32} there are 2^{20} page entries.

Solution use multi-level paging:



Paging: Access Times

- Access time for single-level paging:
  - time for memory access: 100 ns;
  - time for TLB access: 20 ns;
  - If the hit ratio is 80%, the time for translation is: effective access time = 0.80 $\times$ 120 + 0.20 $\times$ 220 = 140 ns: 40% slower than without paging;
  - If the hit ratio is 98%, the time for translation is: effective access time = 0.98 $\times$ 120 + 0.02 $\times$ 220 = 122 ns: 22% slower than without paging.
- Intel Pentium has two level paging and TLB hit ratio of 98%.

Demand Paging

- Page is loaded into memory only when it is needed:
  - less **I/O**;
  - less memory;
  - faster response;
  - more processes.
- If page is in memory (if valid flag is set), continue.
- If page is not in memory:
  - if frames are available load referenced page into memory;
  - if frames are not available choose another page in memory:
    - save page to backing store (if write flag is set);
    - load referenced page.
- Each access to a page which is not in memory produces an exception: page fault.
- Need strategy for replacement.

Optimal Algorithm

- Strategy: replace page that will not be used for the longest time.
- Example: page requests 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5.
- With 4 frames:

Frame 1	1	4
Frame 2	2	–
Frame 3	3	–
Frame 4	4	5

6 page faults

- Optimal algorithm is not known.
- Optimal algorithm can be used to evaluate performance of other algorithms.

First-in, First-out (FIFO)

- Example: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- With 3 frames:

Frame 1	1	4	5
Frame 2	2	1	3
Frame 3	3	2	4

9 page faults.

- With 4 frames:

Frame 1	1	5	4
Frame 2	2	1	5
Frame 3	3	2	–
Frame 4	4	3	–

10 page faults.

Problem: increasing number of frames can increase number of page faults.

Least-recently Used (LRU)

- Strategy: replace page that has not been used for the longest time.
- Example: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5.
- With 4 frames:

Frame 1	1	–	5
Frame 2	2	–	–
Frame 3	3	5	4
Frame 4	4	3	–

8 page faults.

- Implementation:
  - each page has a counter: every time the page is referenced the counter is updated with current value of the clock;
  - to replace a page choose a page which has the lowest counter, i.e. has not been referenced for a long time.

Segmentation

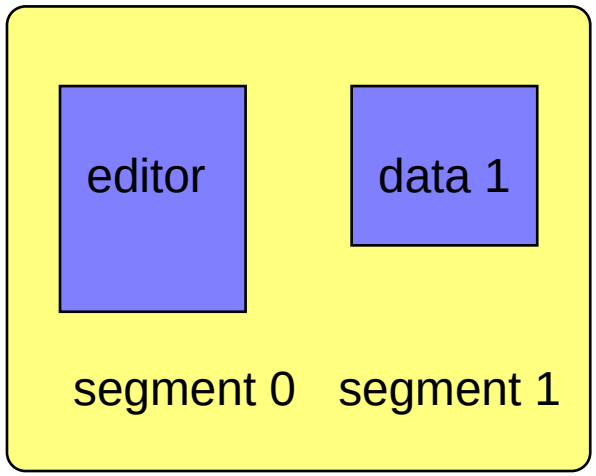
- Memory management scheme that supports system view of memory: paging.
- Memory management scheme that supports user view of memory: segmentation.
- Each logical part of the program has its segment:
  - private instructions;
  - shared instructions;
  - private data;
  - shared data;
  - local data;
  - global data;
  - stack.

Segmentation (2)

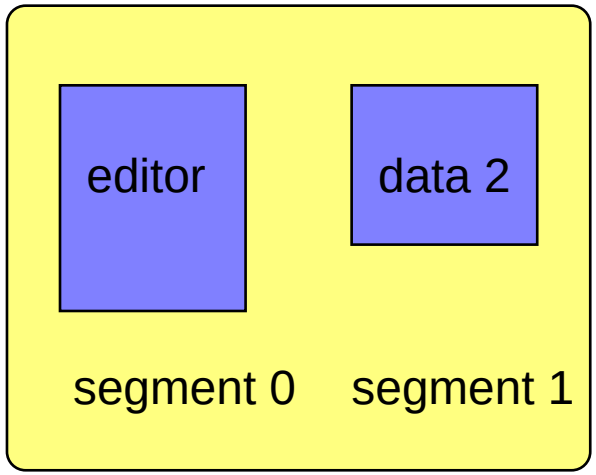
- Virtual addresses consist of two-dimensional user addresses:
  - segment address;
  - offset address.
- Segment table maps two-dimensional user address to one-dimensional physical memory address. Each entry has:

  base – the physical address of start of memory segment;
  limit – the physical address of end of memory segment.
- Table base register: physical address of segment table start;
- Table limit register: physical address of segment table end.
- Segmentation simplifies:
  - relocation;
  - sharing;
  - allocation.

Segmentation (3)



	base	limit
0	A	B
1	C	D



	base	limit
0	A	B
1	E	F