Process Control Block (PCB)

Process Control Block (PCB)

The process control block (PCB) is a data structure in the operating system kernel containing the information needed to manage a particular process.

Source https://en.wikipedia.org/wiki/Process_control_block 2018-01-21

In brief, the PCB serves as the repository for any information that may vary from process to process.

Example of information stored in the PCB

Process Control Block (PCB)
Process id (PID)
Process state (new, ready, running, waiting or terminated)
CPU Context
I/O status information
Memory management information
CPU scheduling information

Context switch

Context switch (1)

In computing, a context switch is the process of storing the state of a process or of a thread, so that it can be restored and execution resumed from the same point later. This allows multiple processes to share a single CPU, and is an essential feature of a multitasking operating system.

Context switch (2)

It is not enough to save and restore the CPU context. Other information, for example the process state must also be saved and restored.

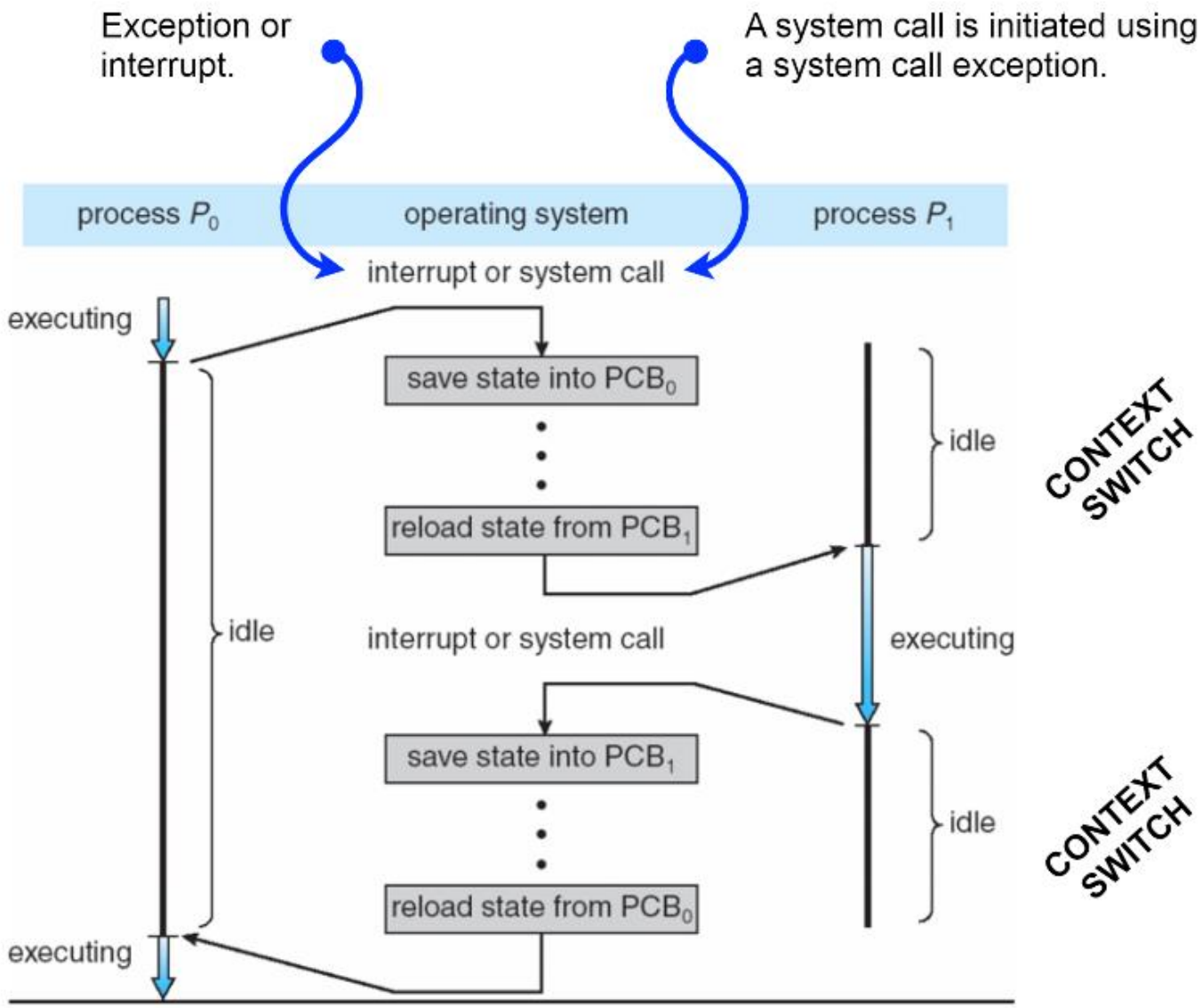
Context switch (3)

When CPU switches to another process, the system must save the state of the old process and restore the state for the new process via a **context switch**.

Context of a process is represented in the PCB.

Context-switch time is **overhead**; the system does no useful work while switching.

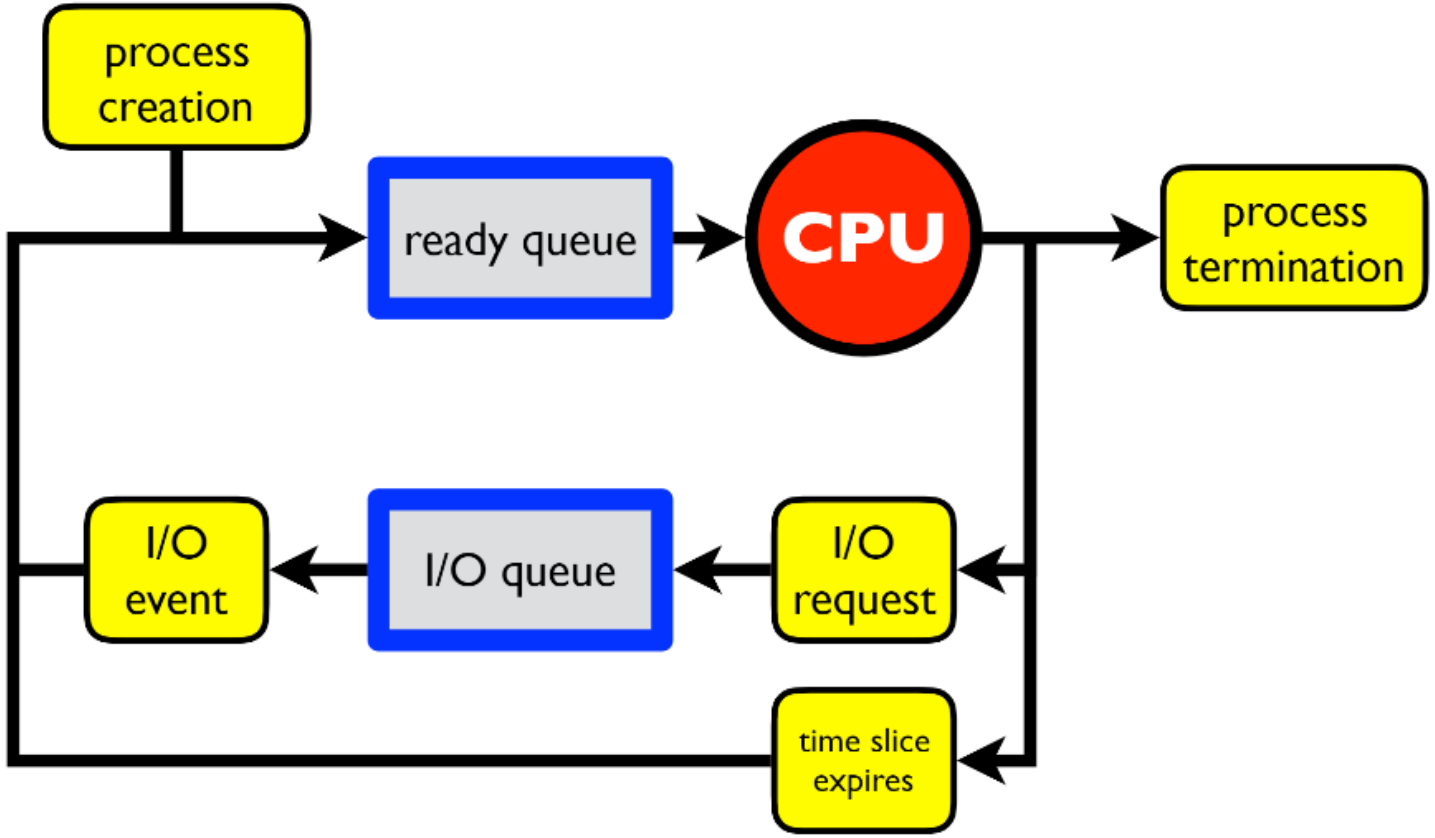
Time for context switch dependent on **hardware support**.


Exception or interrupt.
A system call is initiated using a system call exception.
process P_0
operating system
process P_1
interrupt or system call
executing
save state into PCB_0
reload state from PCB_1
idle
CONTEXT SWITCH
idle
interrupt or system call
executing
save state into PCB_1
reload state from PCB_0
idle
CONTEXT SWITCH
executing

Process queues

Process queues

We have already seen two examples of process queues: the ready queue and the I/O queue.



Process queues

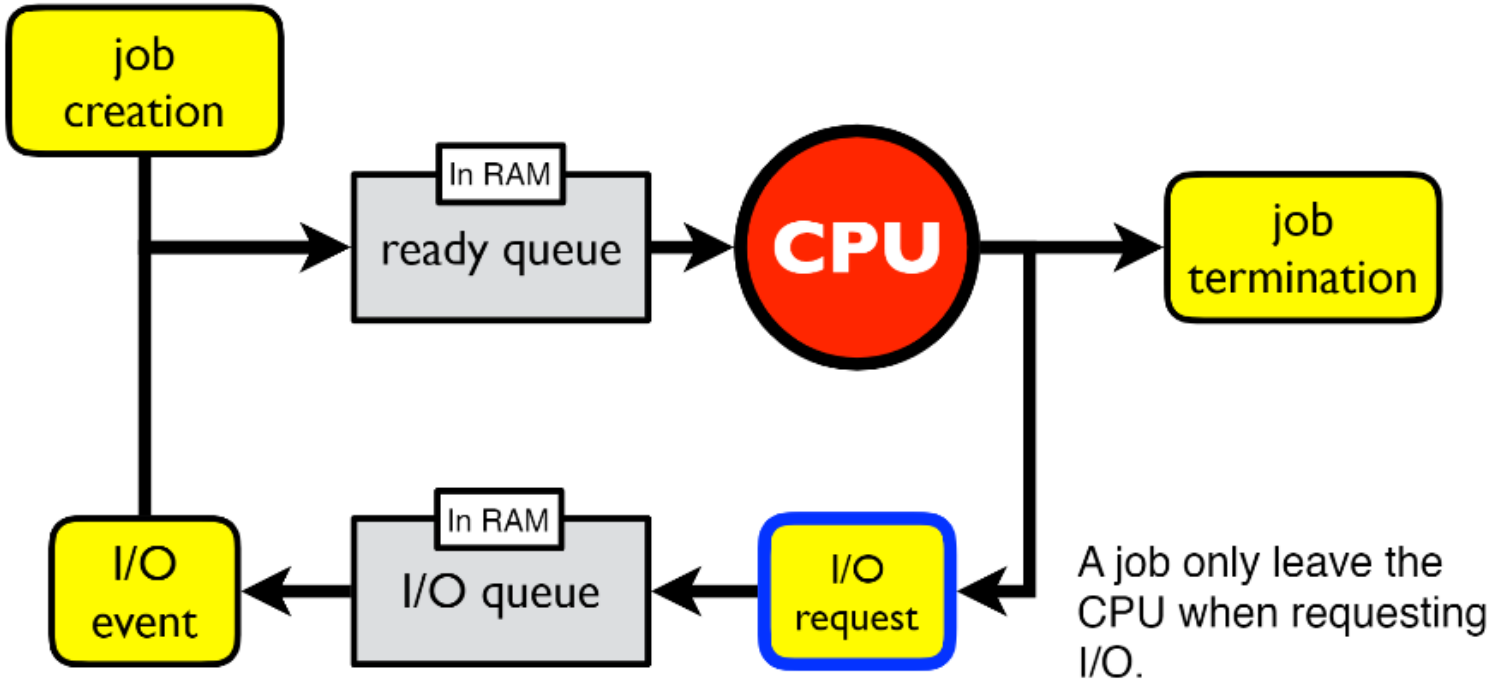
In general we may use one queue for each I/O device.

Ready queue	Set of all processes residing in main memory, ready and waiting to execute
Device queue	Set of processes waiting for an I/O device

CPU Scheduling

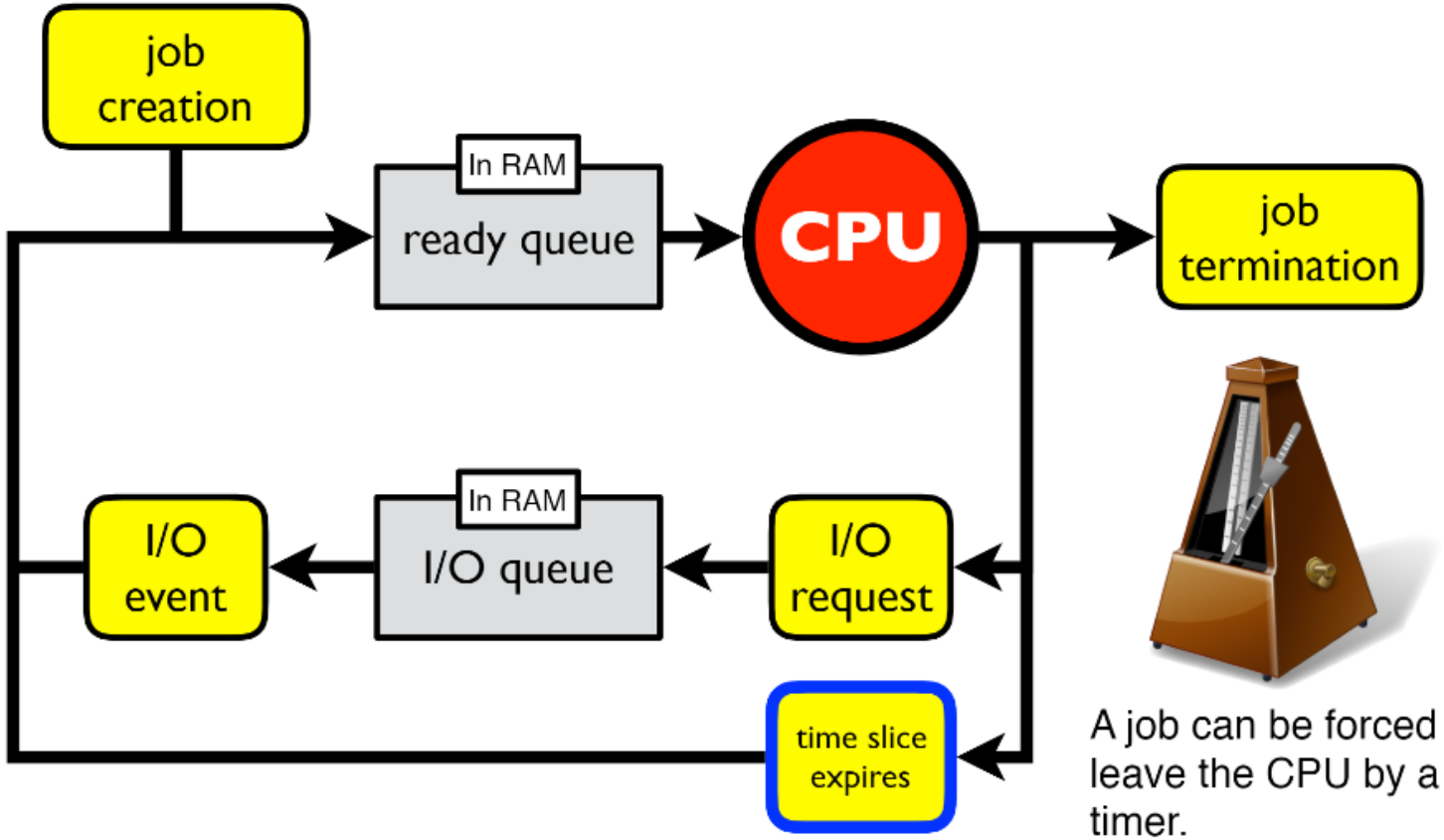
Multiprogramming

A schematic view of multiprogramming



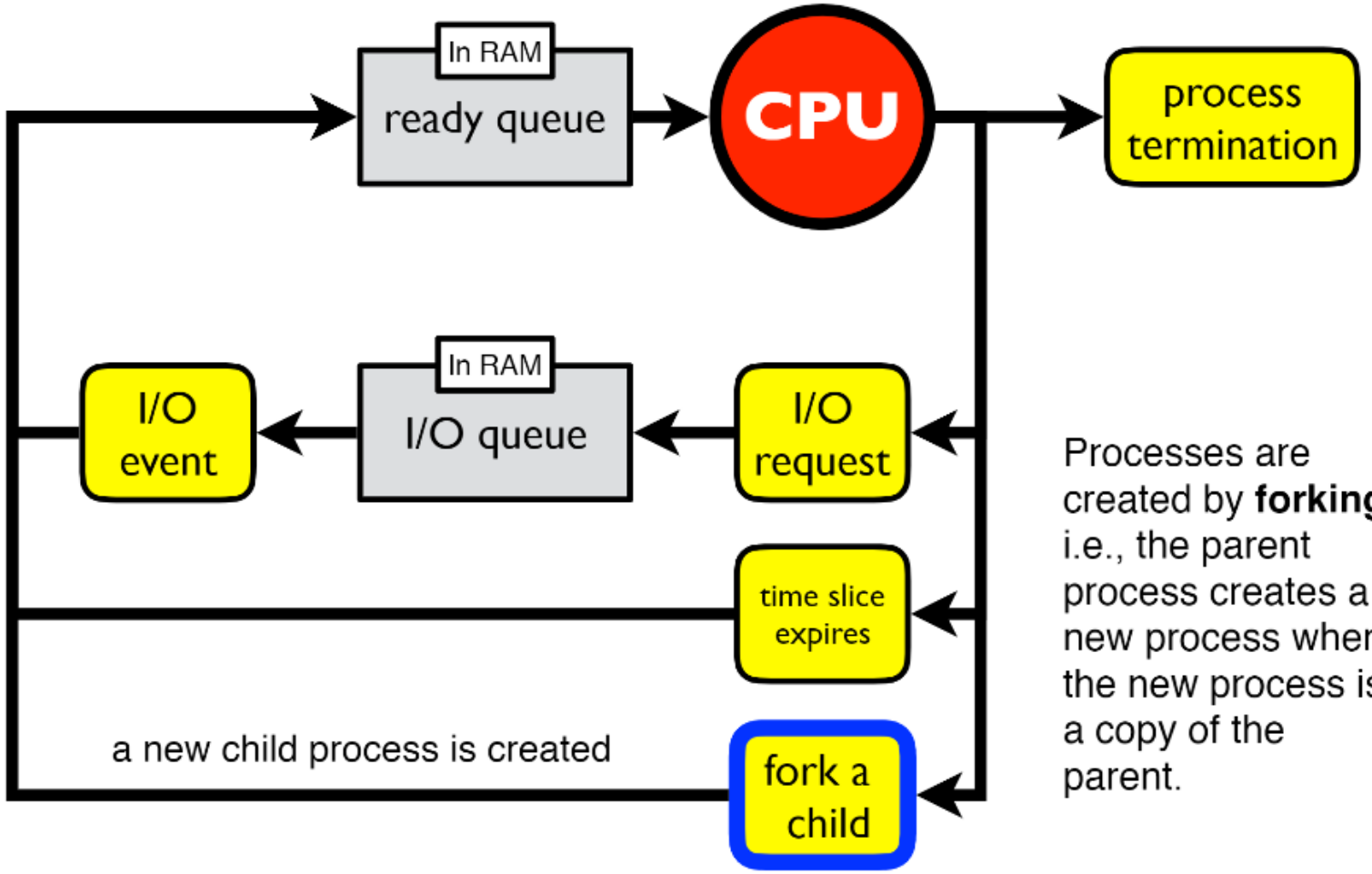
Multitasking

A schematic view of multitasking



A job can be forced to leave the CPU by a timer.

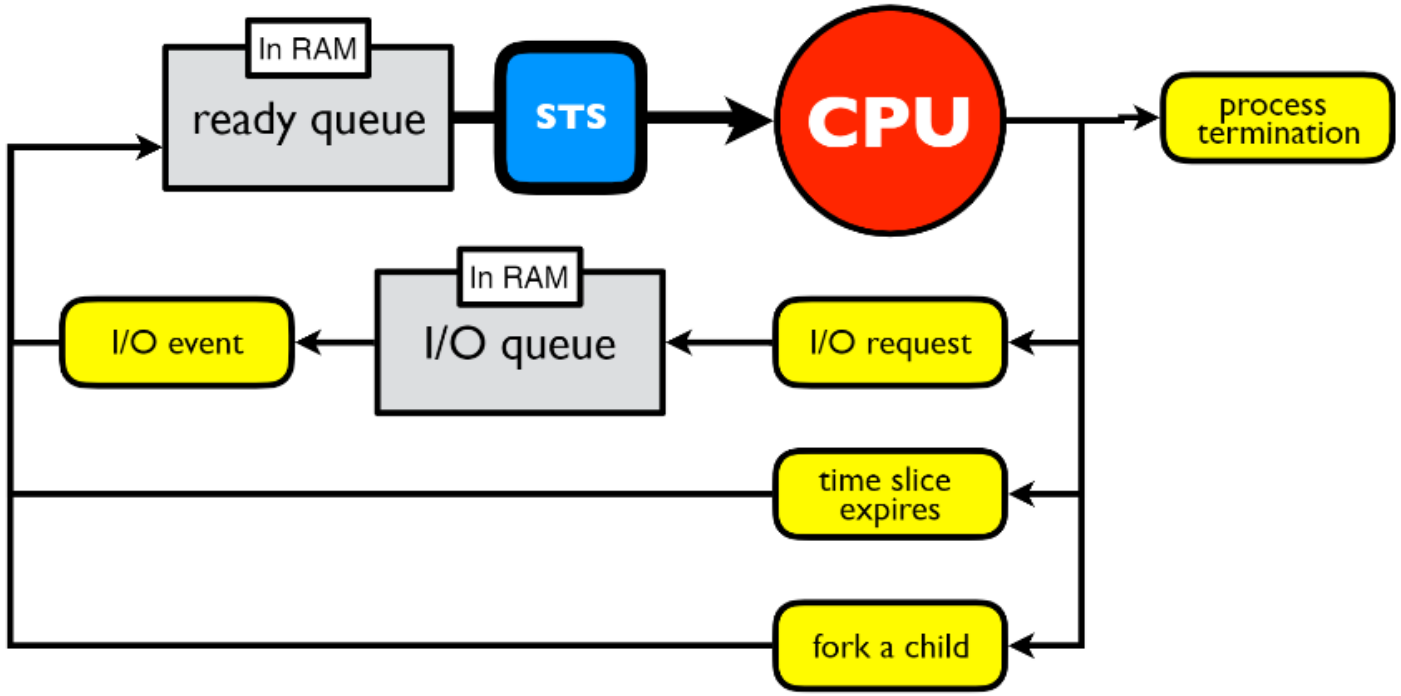
Process creation



Processes are created by **forking**, i.e., the parent process creates a new process where the new process is a copy of the parent.

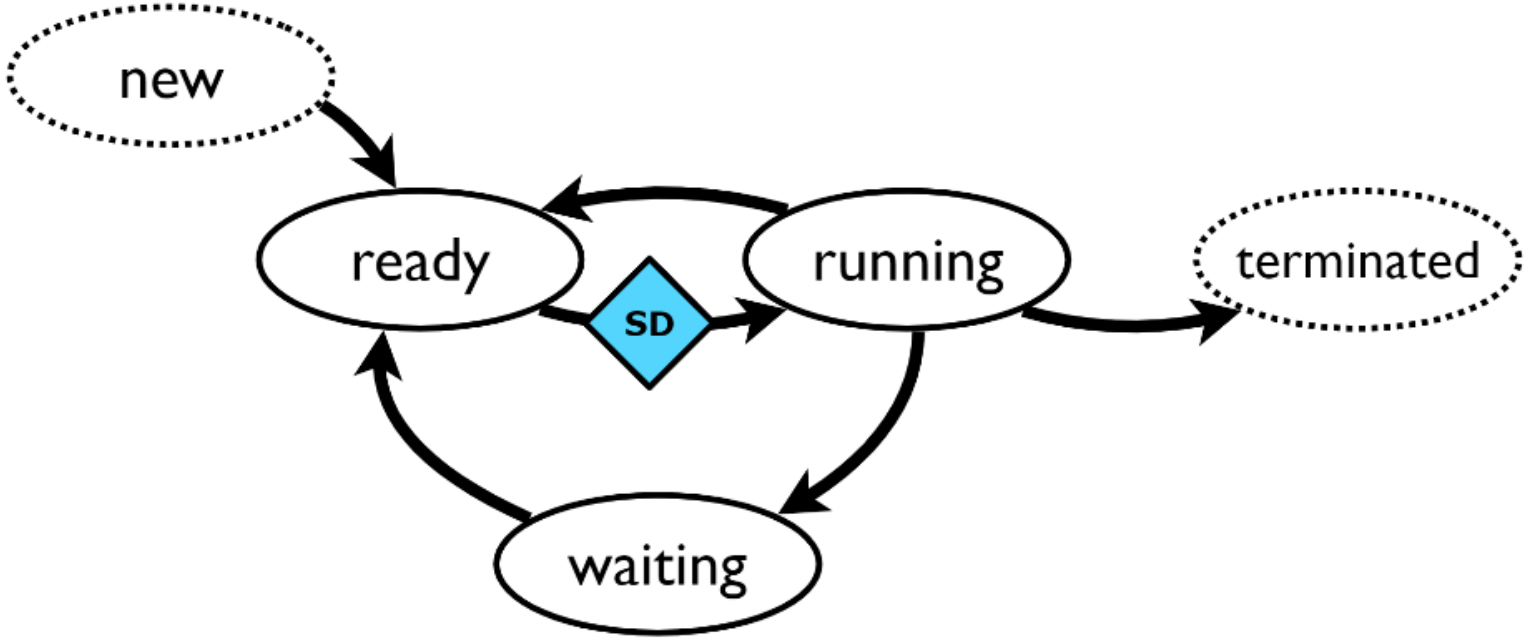
Short-term scheduler

The Short-term scheduler (STS), aka CPU scheduler, selects which process in the in memory ready queue that should be executed next and allocates CPU.



Scheduler dispatch

The CPU scheduler selects one process from among the processes in memory that are READY to execute. The scheduler dispatcher then gives the selected process control of the CPU. This action is called scheduler dispatch (SD).

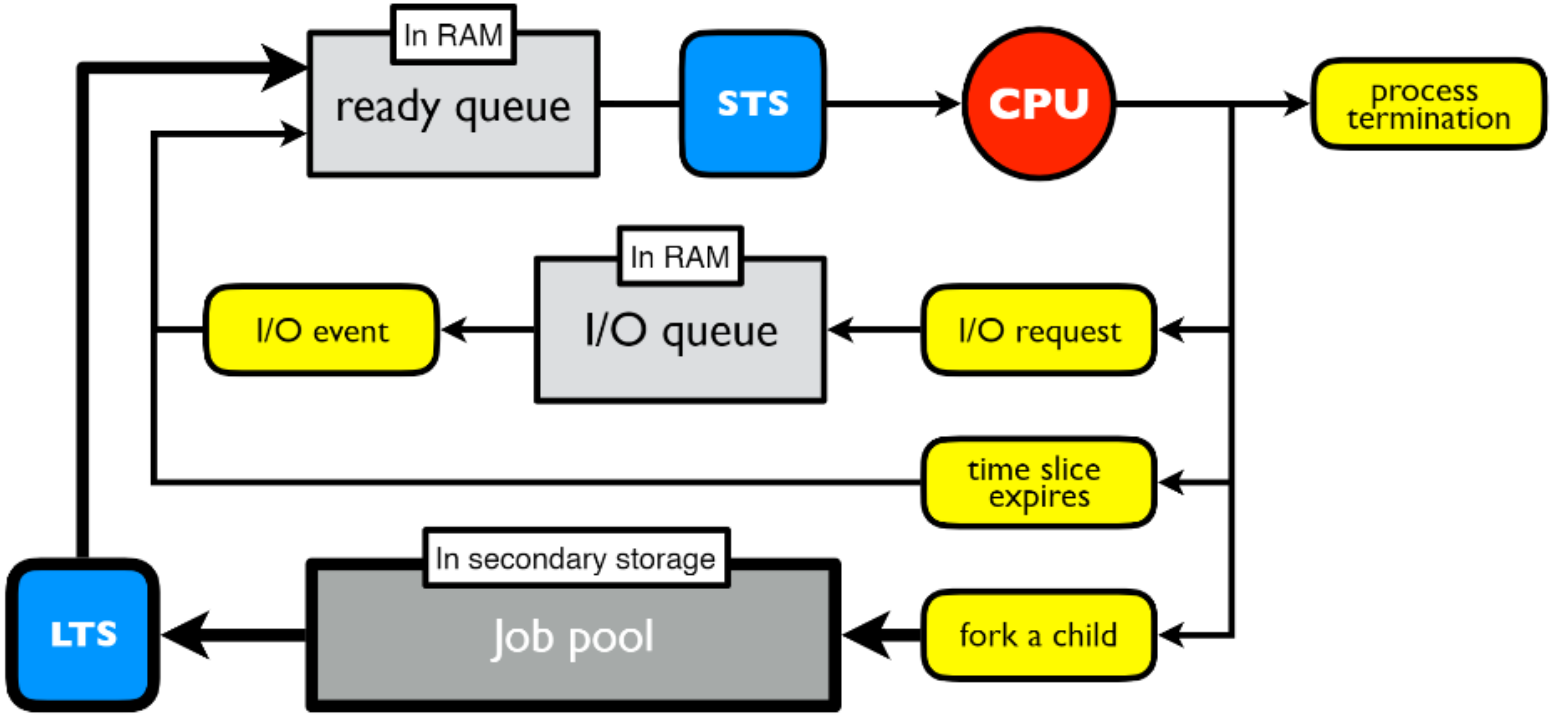


Example of information stored in the PCB

Process Control Block (PCB)
Process id (PID)
Process state (new, ready, running, waiting or terminated)
CPU Context
I/O status information
Memory management information
CPU scheduling information

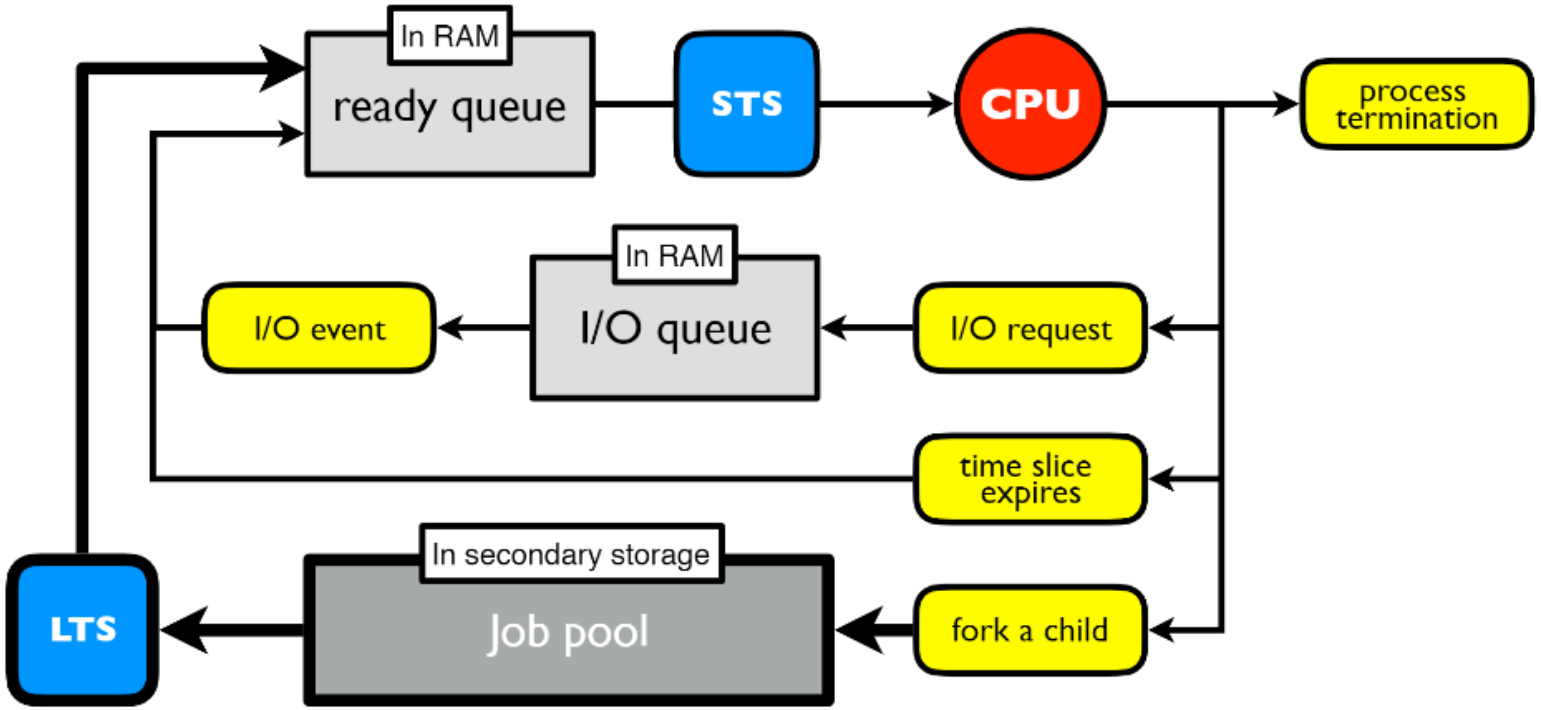
Long-term scheduler (1)

The Long-term scheduler (LTS) (aka job scheduler) decides whether a new process should be brought into the ready queue in main memory or delayed. When a process is ready to execute, it is added to the job pool (on disk). When RAM is sufficiently free, some processes are brought from the job pool to the ready queue (in RAM).



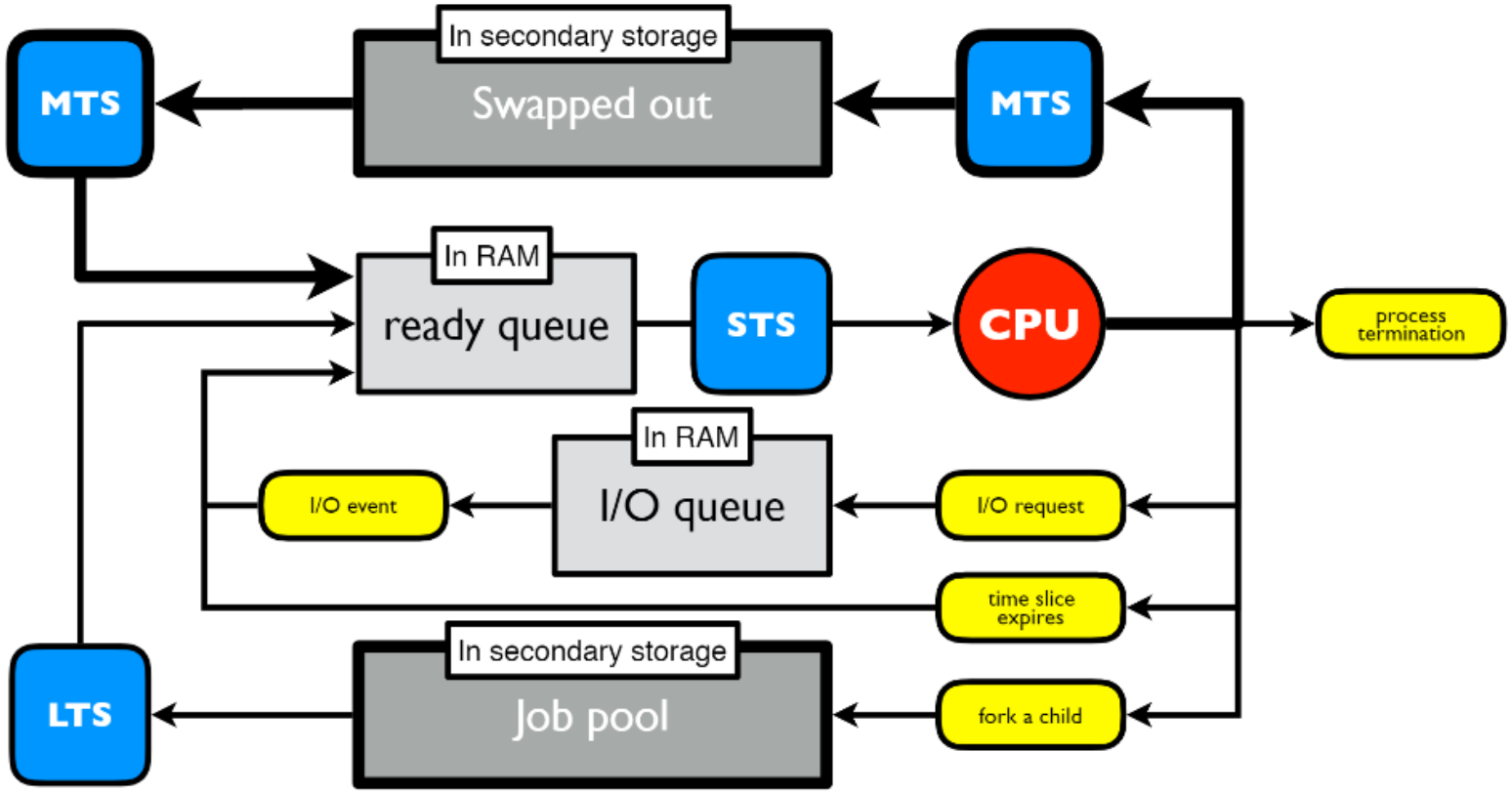
Long-term scheduler (2)

On some systems, the long-term scheduler may be absent or minimal. For example, time-sharing systems such as UNIX and Microsoft Windows systems often have no long-term scheduler but simply put every new process in memory for the short-term scheduler.



Medium-term scheduler

The medium-term scheduler (MTS) temporarily removes processes from main memory and places them in secondary storage and vice versa, which is commonly referred to as "swapping in" and "swapping out".



Classification of processes

In general, processes can be classified by the following characteristics.

- ★ Interactive
- ★ Batch
- ★ Real-Time
- ★ I/O Bound
- ★ CPU Bound

Interactive

Interactive processes interact constantly with their human users.

- Spend a lot of time waiting for keypresses and mouse operations.
- When input is received, the process must be woken up quickly, or the user will find the system to be unresponsive.
- Typically, the average delay must fall between 50 and 150 ms. The variance of such delay must also be bounded, or the user will find the system to be erratic.

Typical examples:

- Command shells and interpreters, text editors, graphical applications and games.

Batch

Batch processes do not interact with human users.

- ★ Do not need to be responsive.
- ★ Often run in the background.
- ★ Often penalised by the scheduler.

Typical examples:

- ★ Compilers, database search engines, and scientific computations.

Real-time

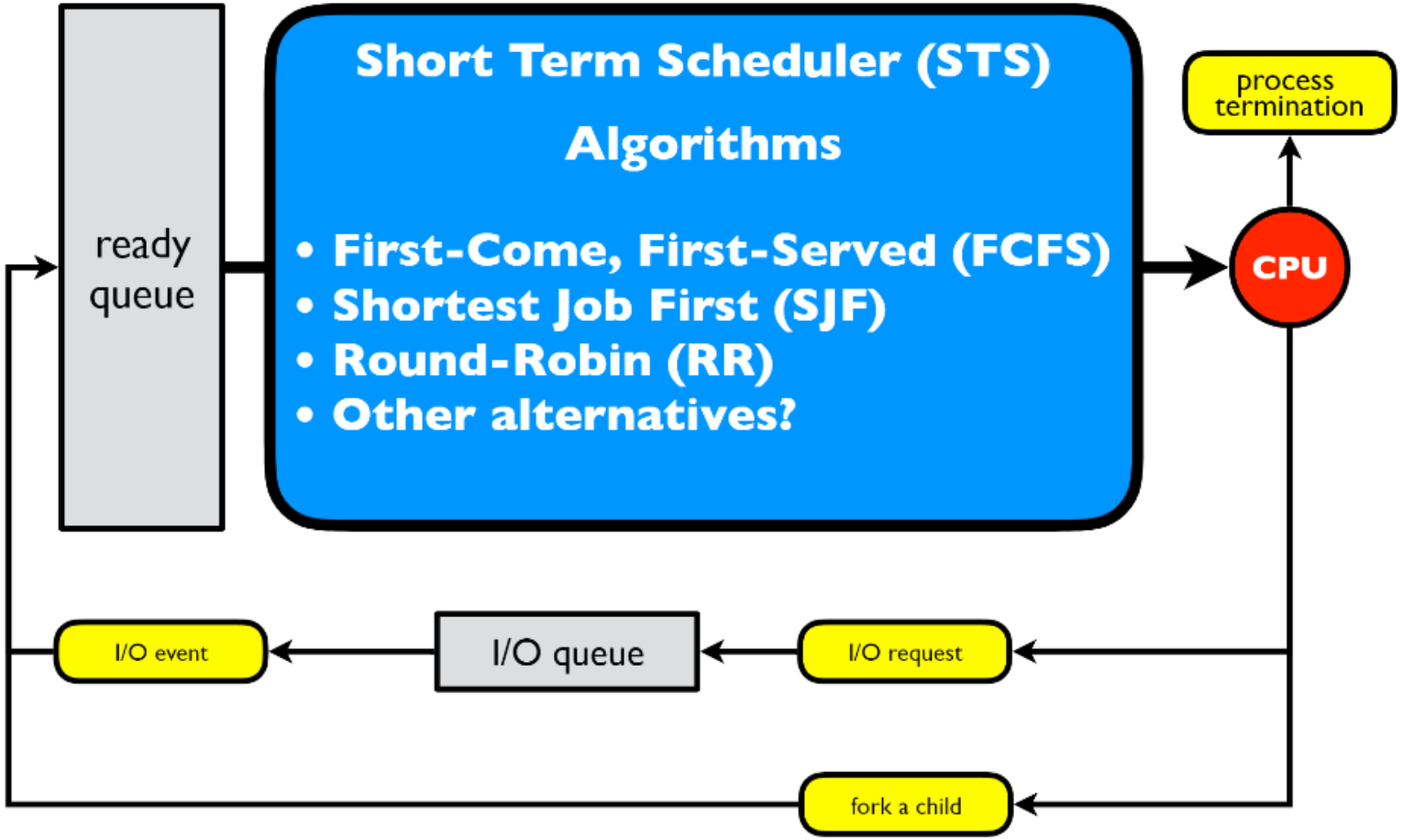
Real-time processes have very strong scheduling requirements.

- ★ Such processes should never be blocked by lower-priority processes.
- ★ Should have a short response time.
- ★ Most important, response time should have a minimum variance.

Typical examples:

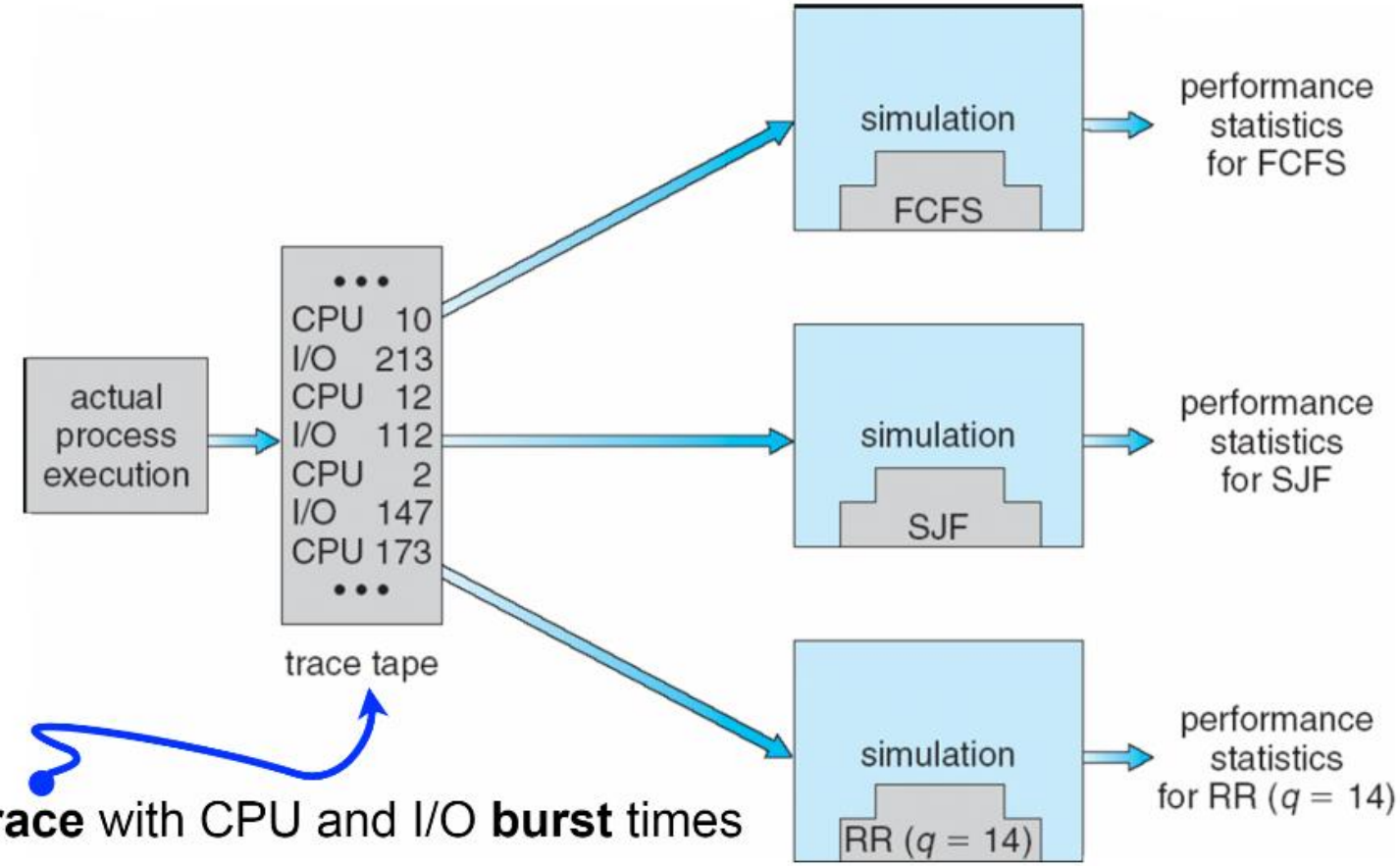
- ★ Video and sound applications, robot controllers, and programs that collect data from physical sensors.

Scheduling algorithms



Evaluation of CPU schedulers by simulation

To evaluate different CPU scheduling algorithms, data obtained from instrumented executables can be used as input in simulations.

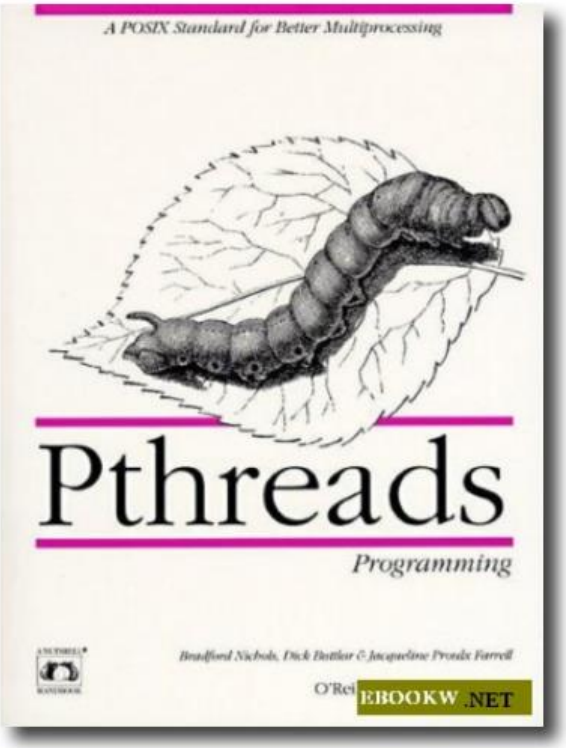


Trace with CPU and I/O **burst** times

Threads

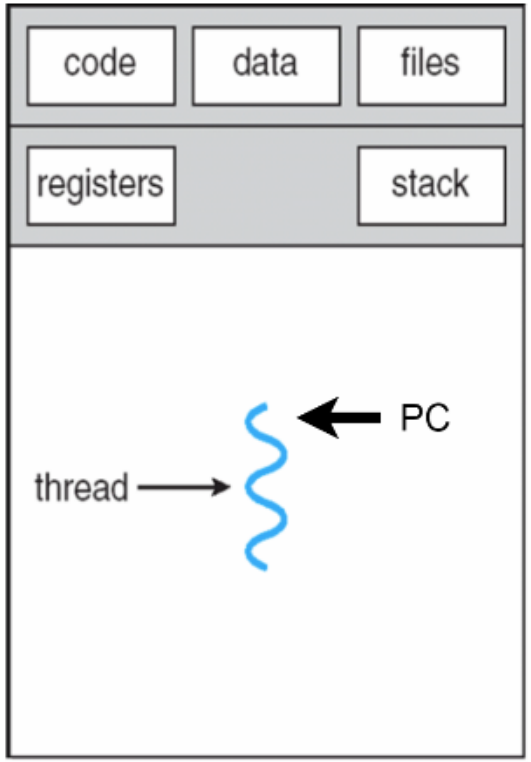
Pthreads

A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization.



- ★ May be provided either as user-level or kernel-level.
- ★ API specifies behaviour of the thread library, implementation depends on the specific library.
- ★ Common in UNIX operating systems (Solaris, Linux, Mac OS X).

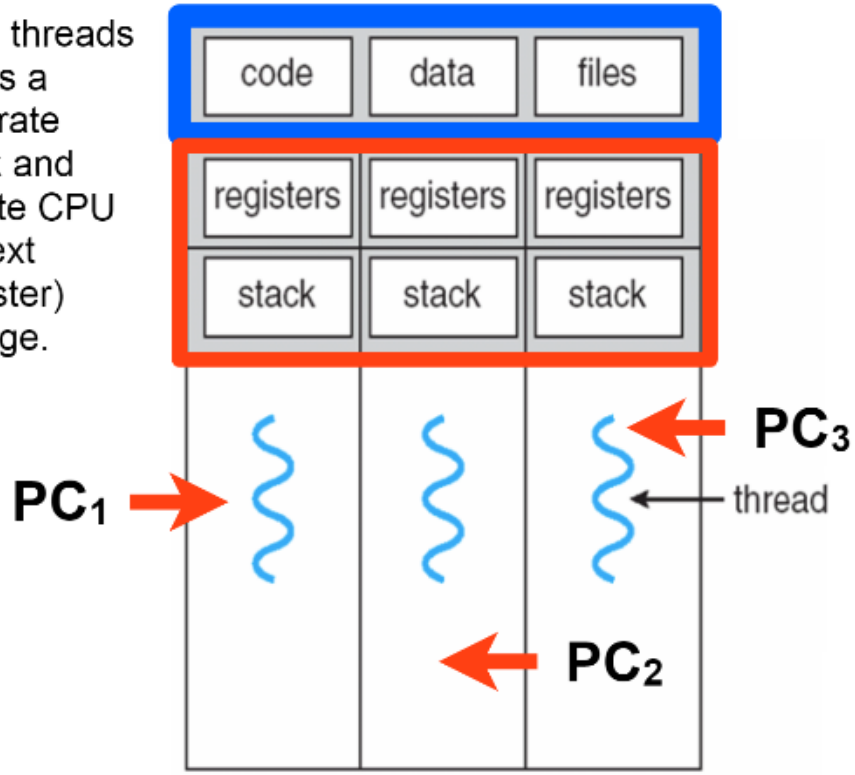
Single and multithreaded processes



single-threaded process

Threads share code, data and open files.

Each threads needs a separate stack and private CPU context (register) storage.



multi-threaded process

Note: this is an example of a process with three threads. Generally, a process may have n > 0 threads.

Parallelism ≠ Concurrency

In parallel systems, two tasks are **actually** performed **simultaneously**.

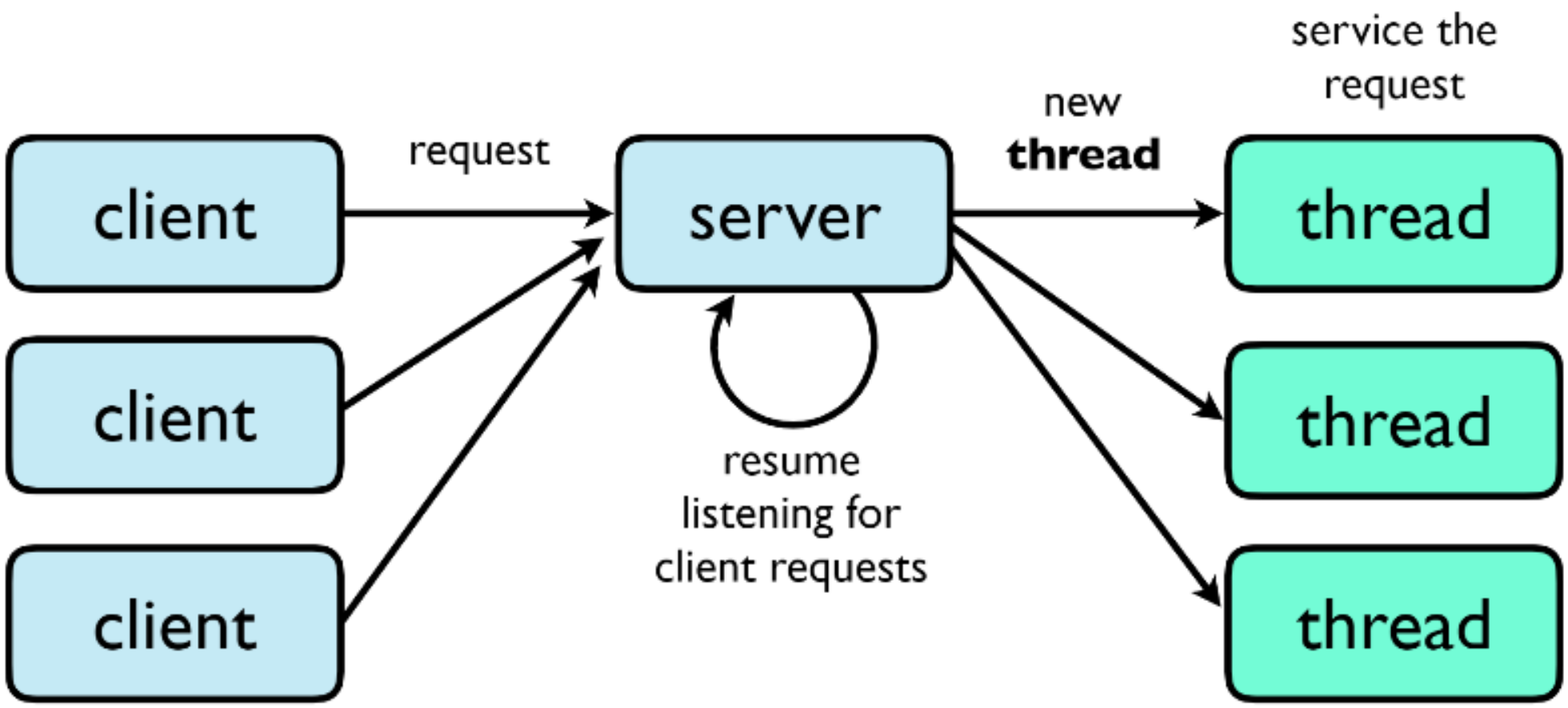
Parallelism is when tasks **literally** run at the **same time**, eg. on a multicore processor.

Two tasks are concurrent if the **order** in which the two tasks are executed in time is **not predetermined**.

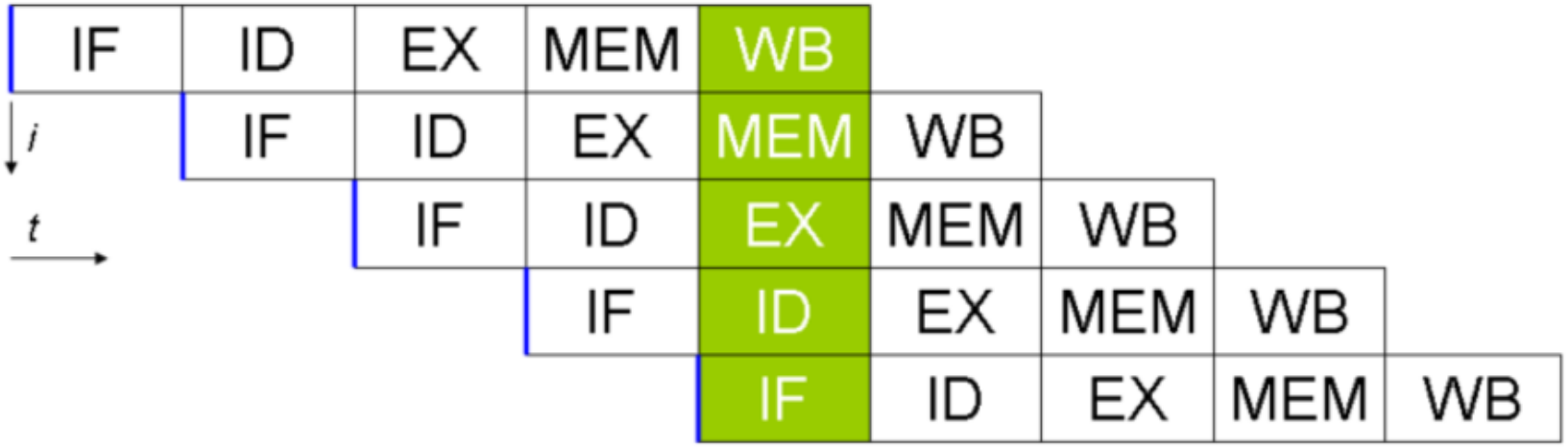
The **appearance** of several tasks executing at once.

Tasks are actually split up into chunks that share the processor with chunks from other tasks by interleaving the execution in a time-slicing way.

Tasks can start, run, and complete in **overlapping** time periods.

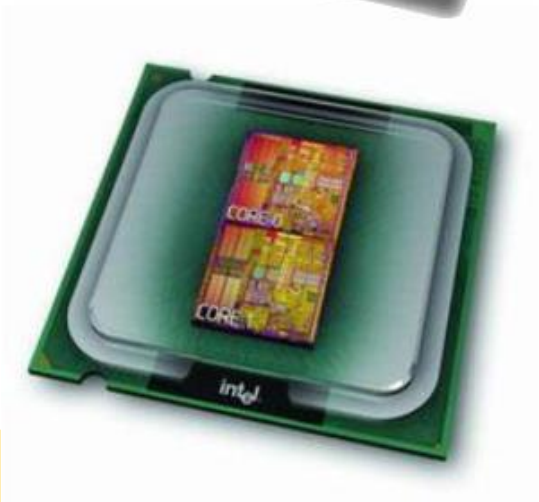

client
client
client
request
server
new
thread
service the
request
thread
thread
thread
resume
listening for
client requests

To improve efficiency, processors commonly utilize **pipelines** internally.



- ★ Allows several instructions to be processed together.
- ★ Instructions are still consumed into the pipeline one at a time.

Multi-Core CPUs



- ★ A multi-core processor is composed of two or more independent cores.
- ★ One can describe it as an integrated circuit which has two or more individual processors (called cores in this sense).

To make full use of multicore computers, programmers will need to learn how to use concurrent programming.

Concurrent execution of threads

On a single core CPU

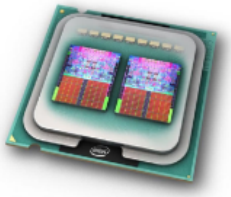
single core	T_1	T_2	T_3	T_4	T_1	T_2	T_3	T_4	T_1	...

time →

Threads take turn executing on the single CPU core. By switching fast enough between the threads they appear to be executing "at the same time".

concurrent ≠ parallel

On a dual core CPU



core 1	T_1	T_3	T_1	T_3	T_1	...
core 2	T_2	T_4	T_2	T_4	T_2	...

time →

Remark 3

At most two threads can execute in parallel (truly at the same time) on a dual core CPU. On every core, threads take turn executing just as on a single core CPU.

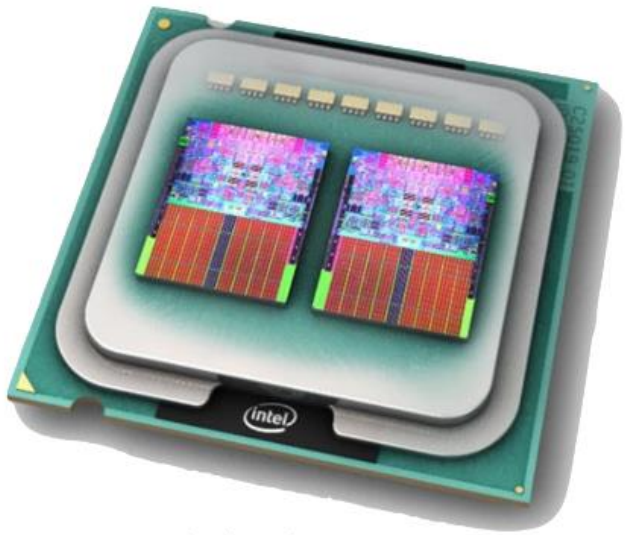
Challenges when using threads

Challenges when using threads

Multi-core give us more computing power - do programmers use threads to utilize this?

Challenges

- ★ Dividing activities
- ★ Balance
- ★ Data splitting
- ★ Data dependency
- ★ Testing and debugging
- ★ Use of shared resources



Intel quad core

Dividing activities

Finding areas in an application that can be divided into separate, concurrent tasks and potentially run in parallel on individual cores.

Balance

If tasks can be found to executed concurrently, programmers also must ensure the tasks to perform equal work of equal value.

Using a separate execution core for a task that don’t contribute much value to the overall process may not be worth the cost.

Data splitting

Just as applications are divided into separate tasks, the data accessed and manipulated by the tasks must be divided to run on separate cores.

Data dependency

The data accessed by the tasks must be examined for dependencies between two or more tasks. In instances where one task depends on data from another, programmers must ensure that the execution of the tasks is synchronized to accommodate the data dependency.

Testing and debugging

When a program is running concurrently on multiple cores, there are many different execution paths. Testing and debugging such concurrent programs is inherently more difficult than testing and debugging single-threaded applications.

Programming using shared memory with threads

shared state

```
#define N 10000

int  BALANCE  0;
```

Thread A

```
for (int i = 0; i < N; i++) {
   BALANCE++;
}
```

Thread B

```
for (int i = 0; i < N; i++) {
   BALANCE--;
}
```

BALANCE == ?

Interleaving of executing threads

Thread A (increment)				Thread B (decrement)		
OP	Operands	$t0	BALANCE	OP	Operands	$t0
			0			
lw	$t0,	0	0			
addi	$t0, $t0, 1	1	0			
sw	$t0, BALANCE	1	1			
			1	lw	$t0, BALANCE	1
			1	addi	$t0, $t0, -1	0
			0	sw	$t0, BALANCE	0

If the instructions are executed in this order, the increment and decrement cancel each other and the resulting **BALANCE** is **0**.

Interleaving of executing threads

Thread A (increment)				Thread B (decrement)		
OP	Operands	$t0	BALANCE	OP	Operands	$t0
			0			
lw	$t0, BALANCE	0	0			
			0	lw	$t0, BALANCE	0
addi	$t0, $t0, 1	1	0			
sw	$t0, BALANCE	1	1			
			1	addi	$t0, $t0, -1	-1
			-1	sw	$t0, BALANCE	-1

Both threads tries to access and update the shared memory location **BALANCE** concurrently. Updates are not atomic and the result depends on the particular order in which the data accesses take place. In this example the resulting **BALANCE** is **-1**.

Race condition

A race condition or race hazard is the behaviour of an electronic, software or other system where the output is dependent on the sequence or timing of other uncontrollable events.

It becomes a bug when events do not happen in the intended order.

The term originates with the idea of two signals racing each other to influence the output first.

shared state

```
#define N 10000

int  BALANCE  0;
```

Thread A

```
for (int i = 0; i < N; i++) {
   BALANCE++;
}
```

Thread B

```
for (int i = 0; i < N; i++) {
   BALANCE--;
}
```

BALANCE == ?

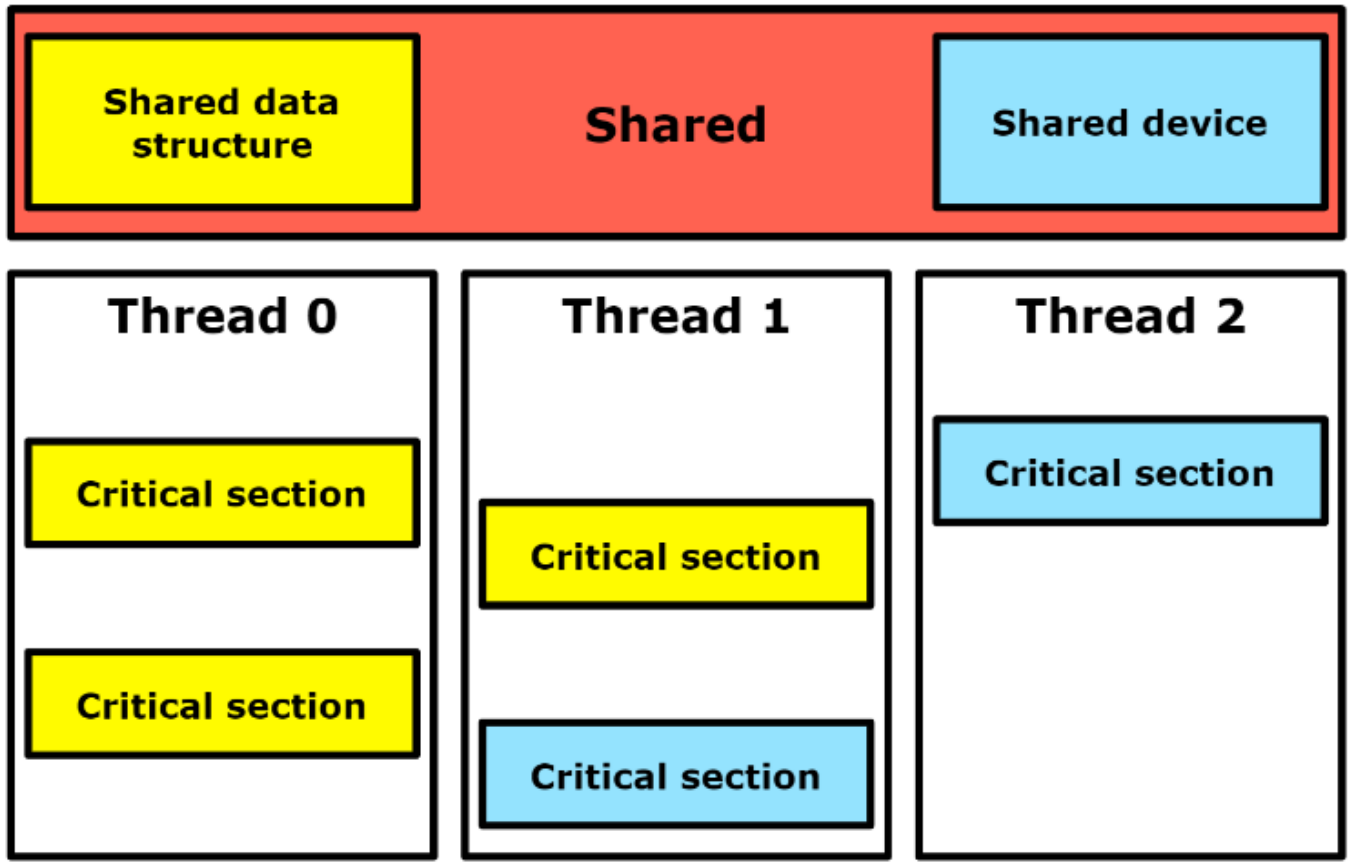
Thread A (increment)				Thread B (decrement)		
OP	Operands	$t0	BALANCE	OP	Operands	$t0
			0			
lw	$t0, BALANCE	0	0			
			0	lw	$t0, BALANCE	0
addi	$t0, $t0, 1	1	0			
sw	$t0, BALANCE	1	1			
			1	addi	$t0, $t0, -1	-1
			-1	sw	$t0, BALANCE	-1

Data race

- ★ Two threads may read and write the same variable **BALANCE** without synchronization.

Race condition

- ★ The final value of the variable **BALANCE** depends on how the threads are scheduled.

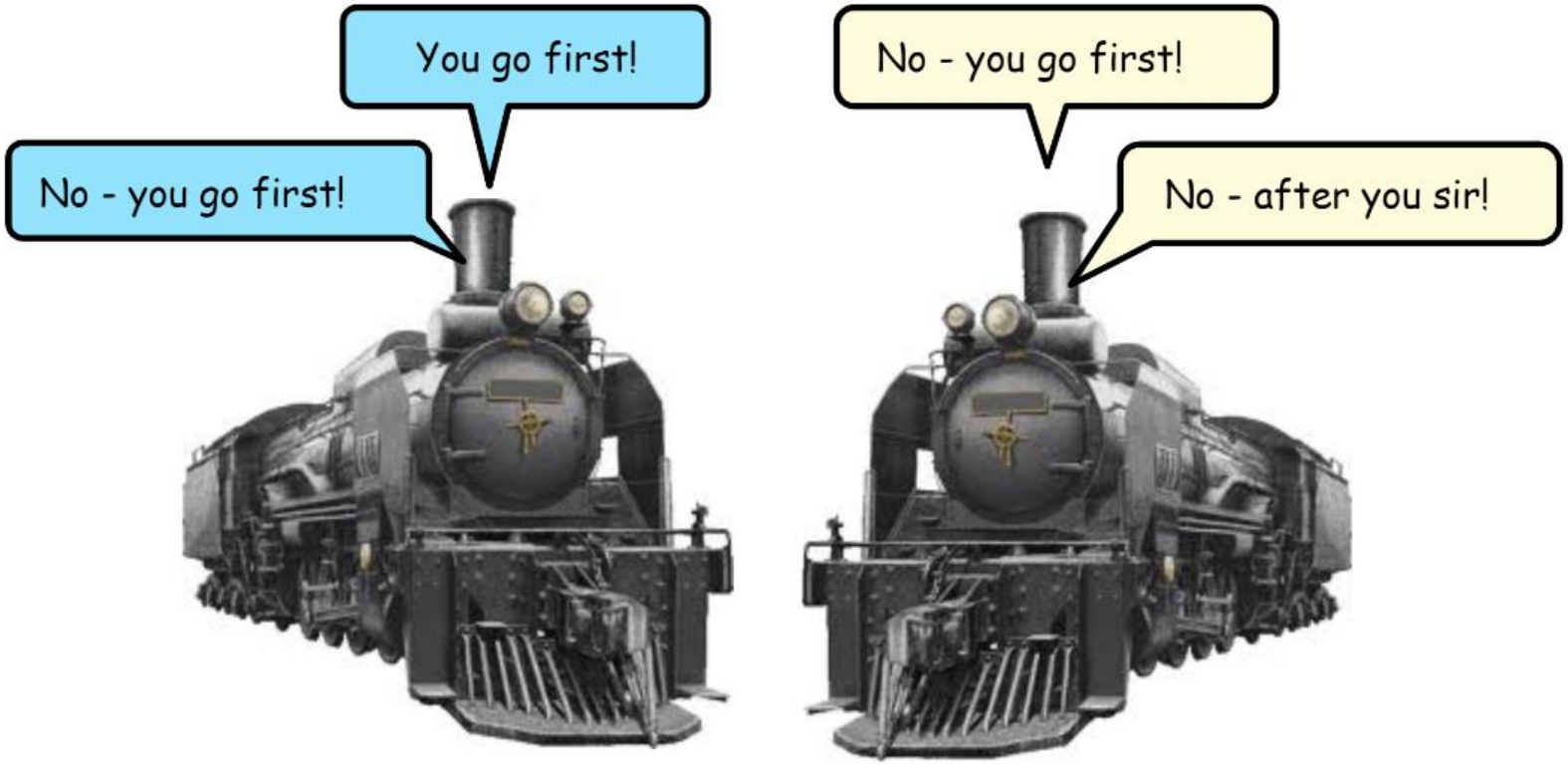


Three threads accessing a shared data structure (yellow) and a shared device (blue).

Mutual exclusion

The requirement of ensuring that no two concurrent processes or threads are in their critical section at the same time.

- ★ A basic requirement in concurrency control, to prevent race conditions.
- ★ First identified and solved by **Edsger W. Dijkstra** in his seminal 1965 paper titled *Solution of a problem in concurrent programming control*, and is credited as the first topic in the study of concurrent algorithms.

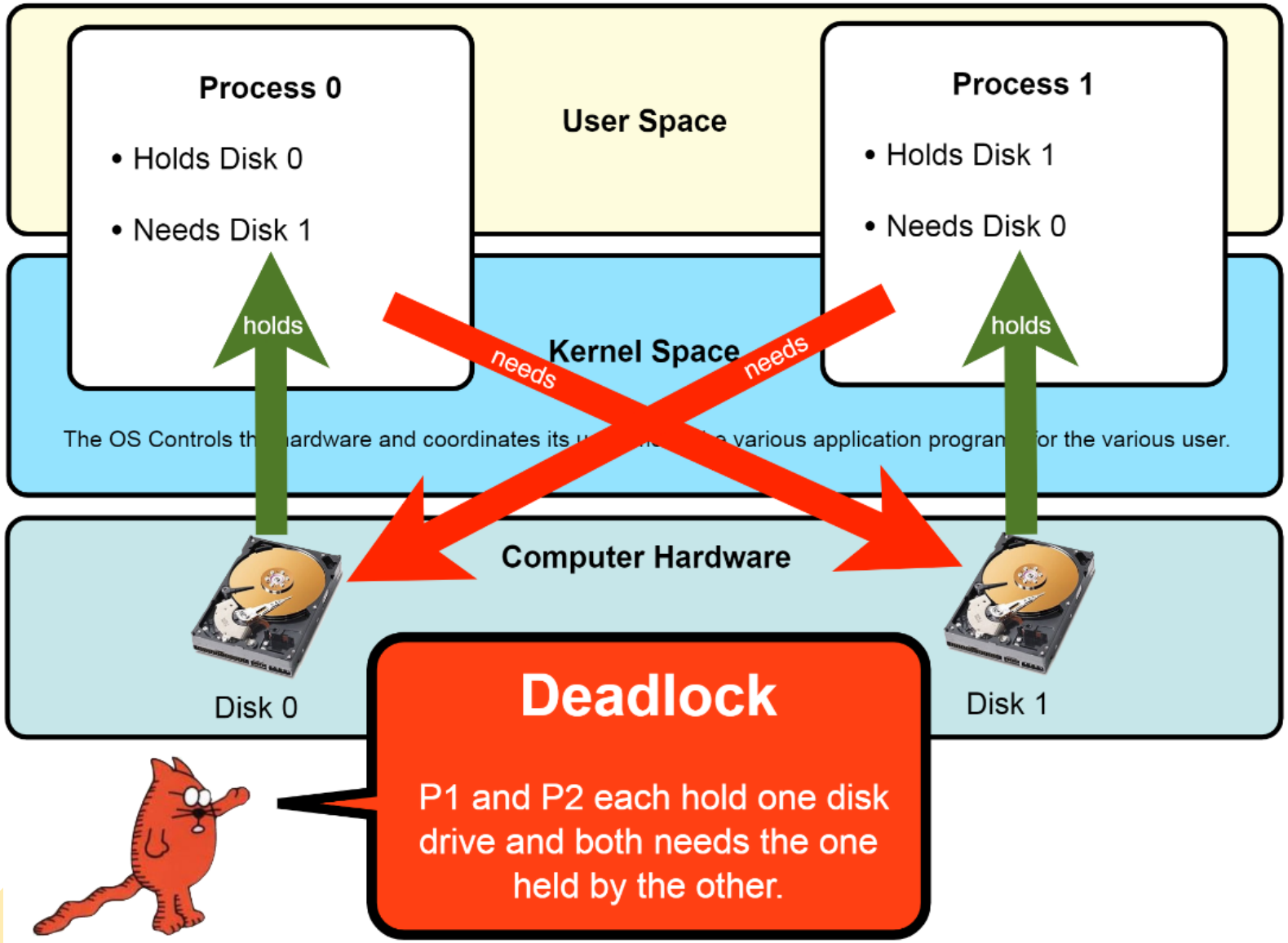


When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone.

An illogical statute passed by the Kansas legislature in early 20th century.

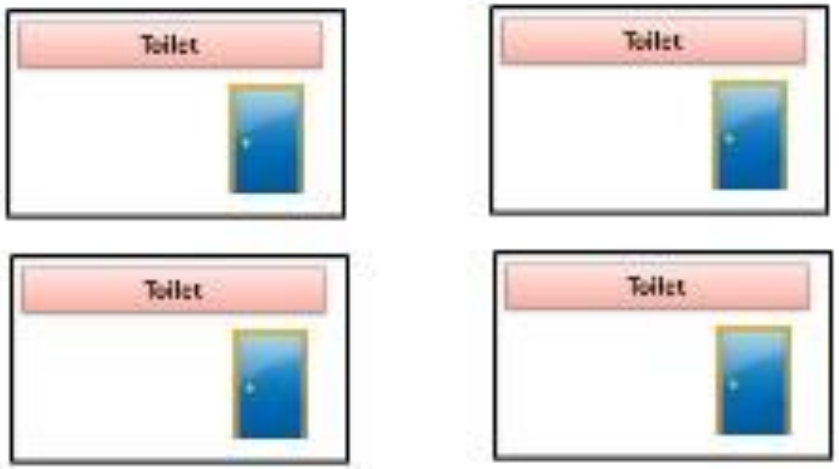
Deadlock

In concurrent computing, a deadlock is a state in which each member of a group is waiting for some other member to take action, such as sending a message or more commonly releasing a lock.


Process 0
• Holds Disk 0
• Needs Disk 1
User Space
Process 1
• Holds Disk 1
• Needs Disk 0
holds
needs
needs
holds
Kernel Space
Computer Hardware
Disk 0
Disk 1
Deadlock
P1 and P2 each hold one disk drive and both needs the one held by the other.

MUTEX

SEMAPHORE



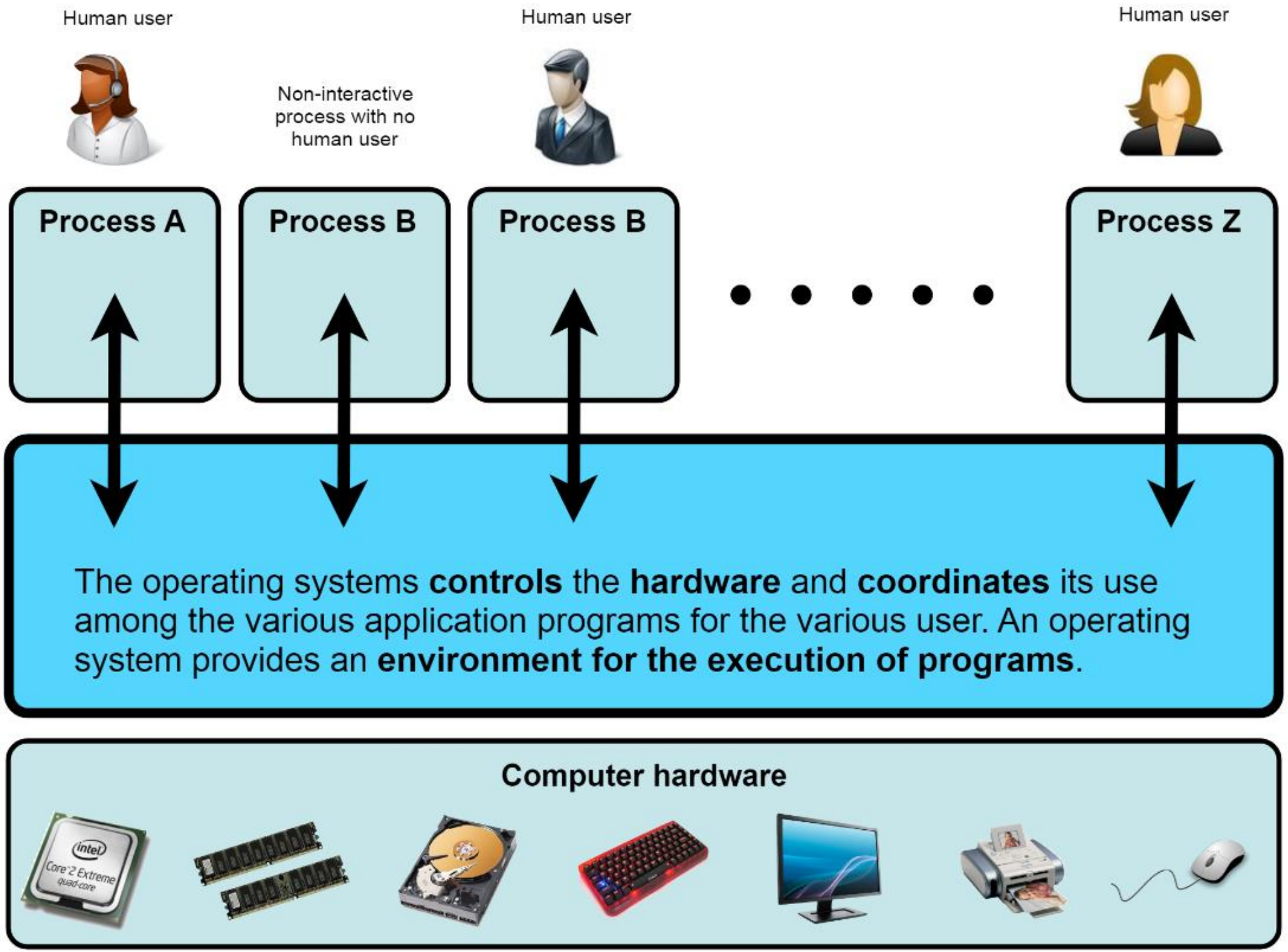
four identical keys

Thread
Mutex
A mutex enables the item to be accessed by only one thread at a time
Image Credits: CodeGym

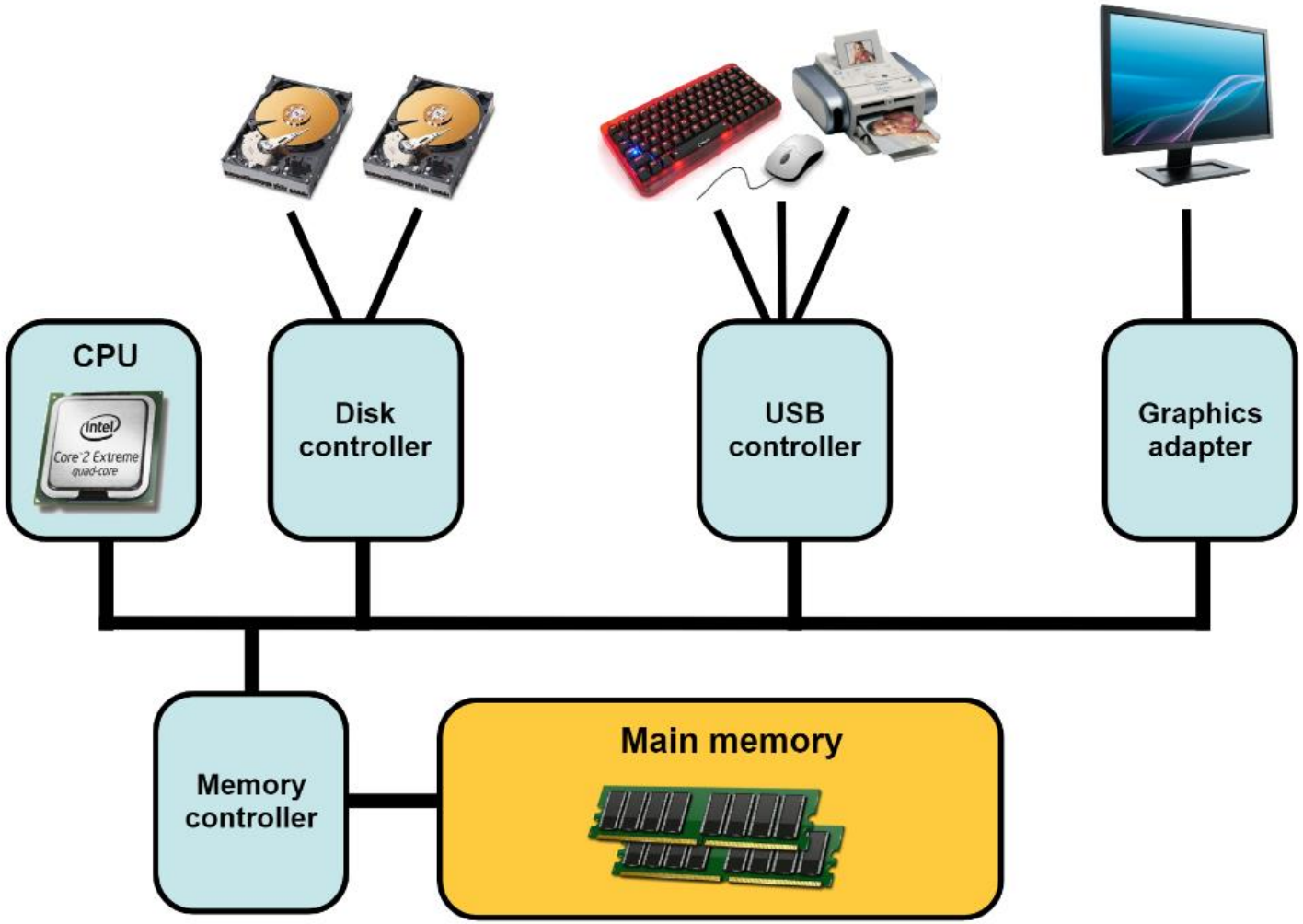
Dining philosophers

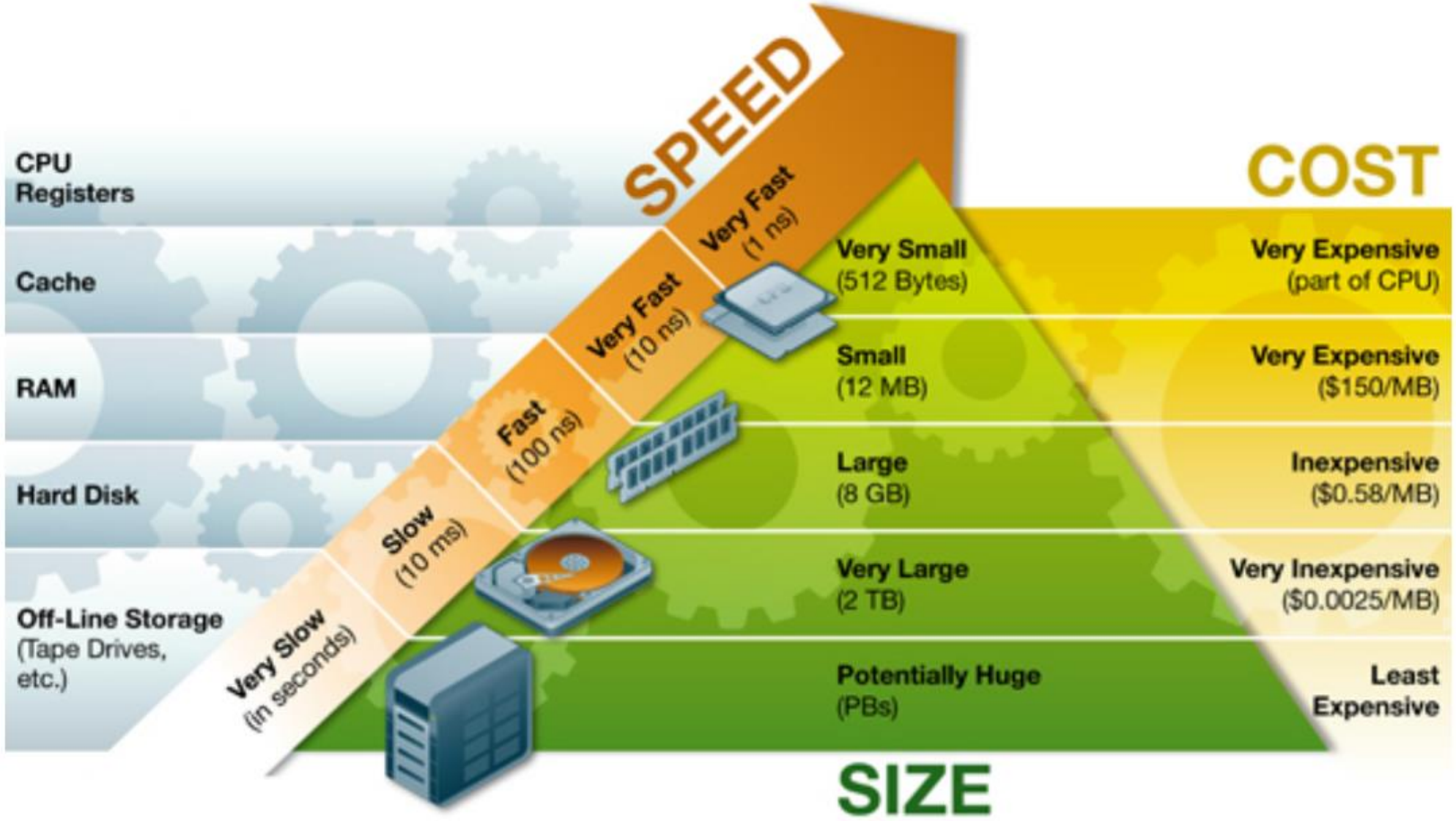
An **example problem** often used in concurrent algorithm design to illustrate synchronization issues and techniques for resolving them.

MEMORY MANAGMENT


Human user
Human user
Human user
Non-interactive process with no human user
Process A
Process B
Process B
Process Z
The operating systems **controls** the **hardware** and **coordinates** its use among the various application programs for the various user. An operating system provides an **environment for the execution of programs**.
Computer hardware

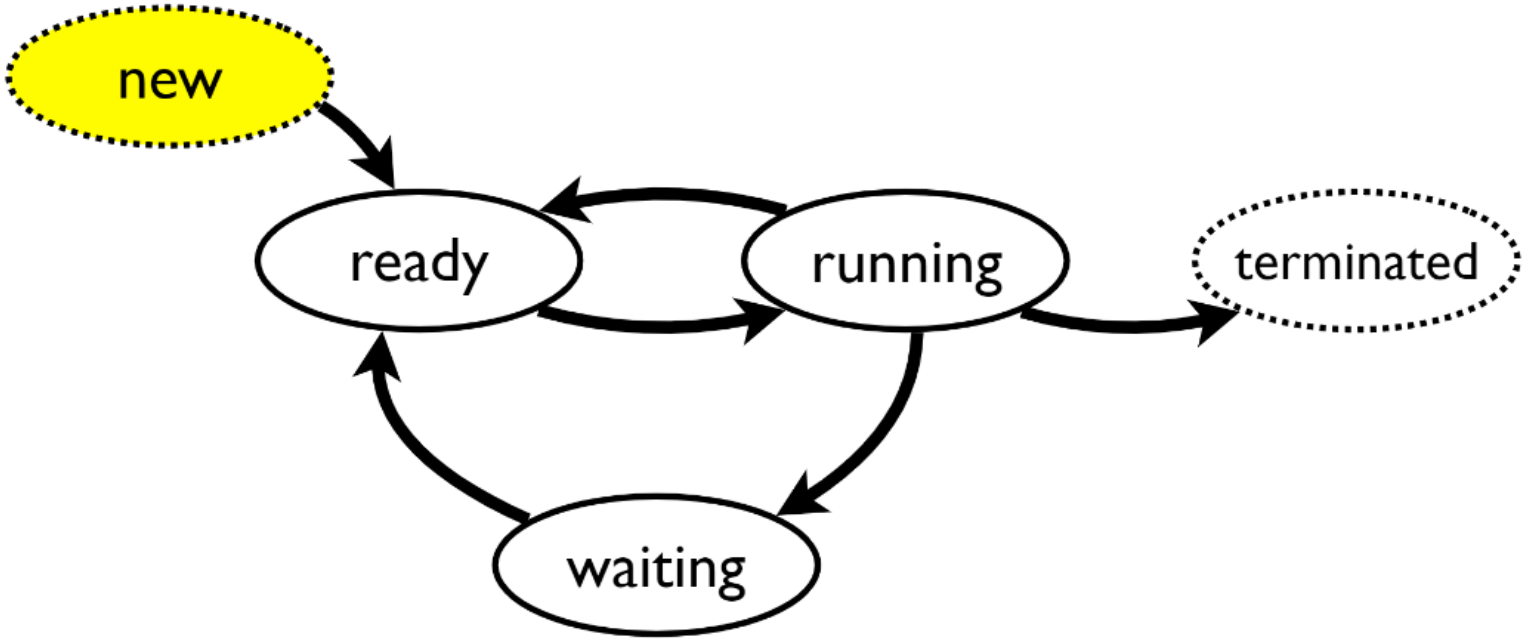
A typical computer system




SPEED
COST
SIZE
CPU Registers
Cache
RAM
Hard Disk
Off-Line Storage (Tape Drives, etc.)
Very Fast (1 ns)
Very Fast (10 ns)
Fast (100 ns)
Slow (10 ms)
Very Slow (in seconds)
Very Small (512 Bytes)
Small (12 MB)
Large (8 GB)
Very Large (2 TB)
Potentially Huge (PBs)
Very Expensive (part of CPU)
Very Expensive ($150/MB)
Inexpensive ($0.58/MB)
Very Inexpensive ($0.0025/MB)
Least Expensive

Process creation

To run a program, the operating system must **fetch** the program **executable from disk** and place the program in a **new process** in **main memory** for it to be run.



Process Control Block (PCB)

The operating system must allocate memory for the process control block.

Process control block (PCB)

Must allocate memory for a new PCB

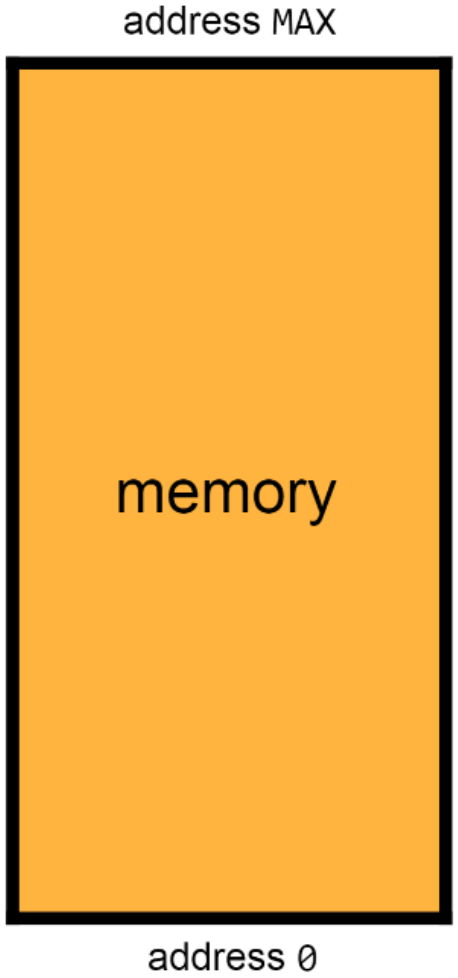
Example of information stored in the PCB

Process Control Block (PCB)
Process id (PID)
Process state (new, ready, running, waiting or terminated)
CPU Context
I/O status information
CPU scheduling information
Memory management information

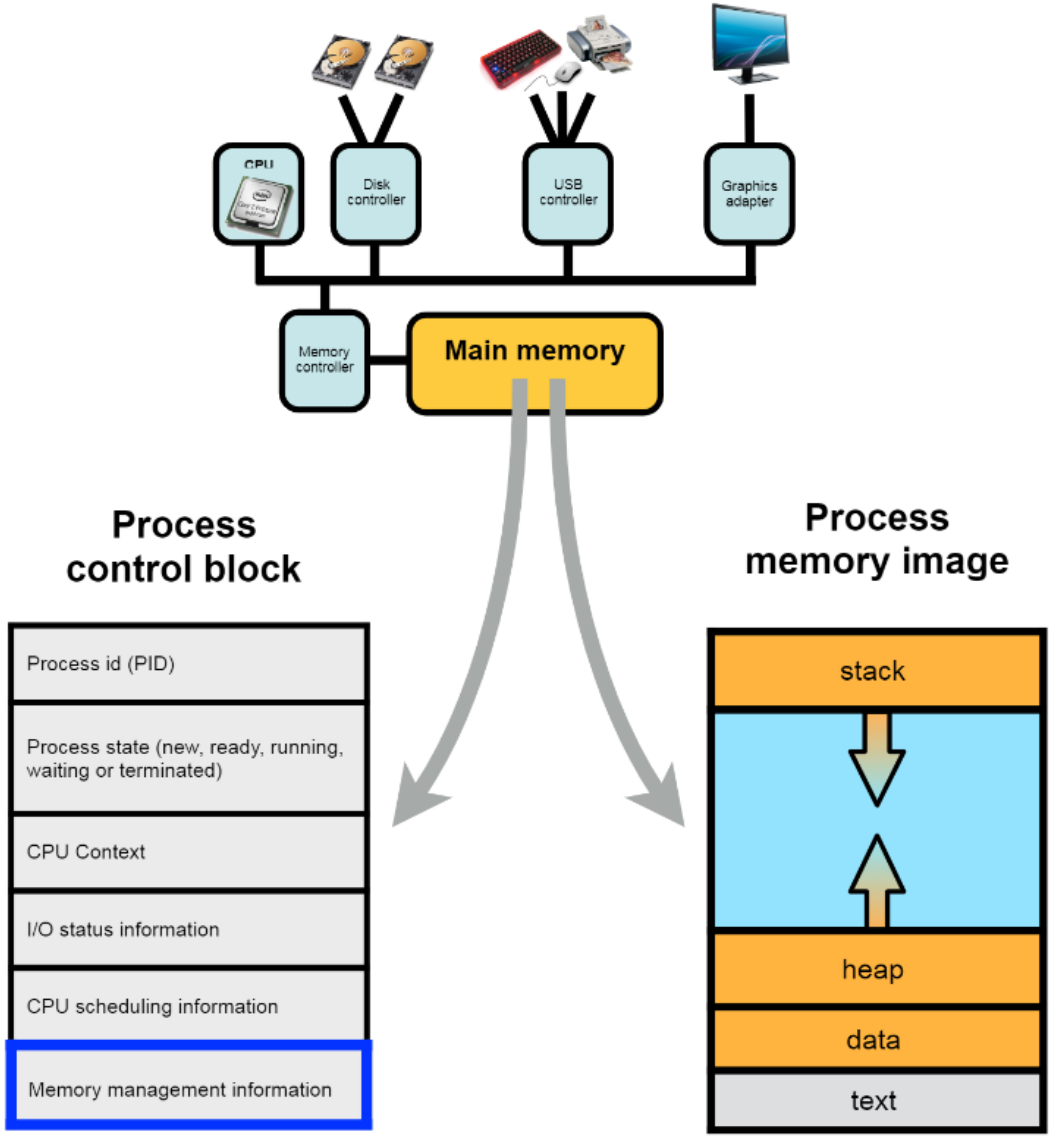
Process memory space

In addition to a PCB, the operating system must allocate a new **memory space** for each new process. This memory space is often referred to as the process **memory image**.

Static memory allocation

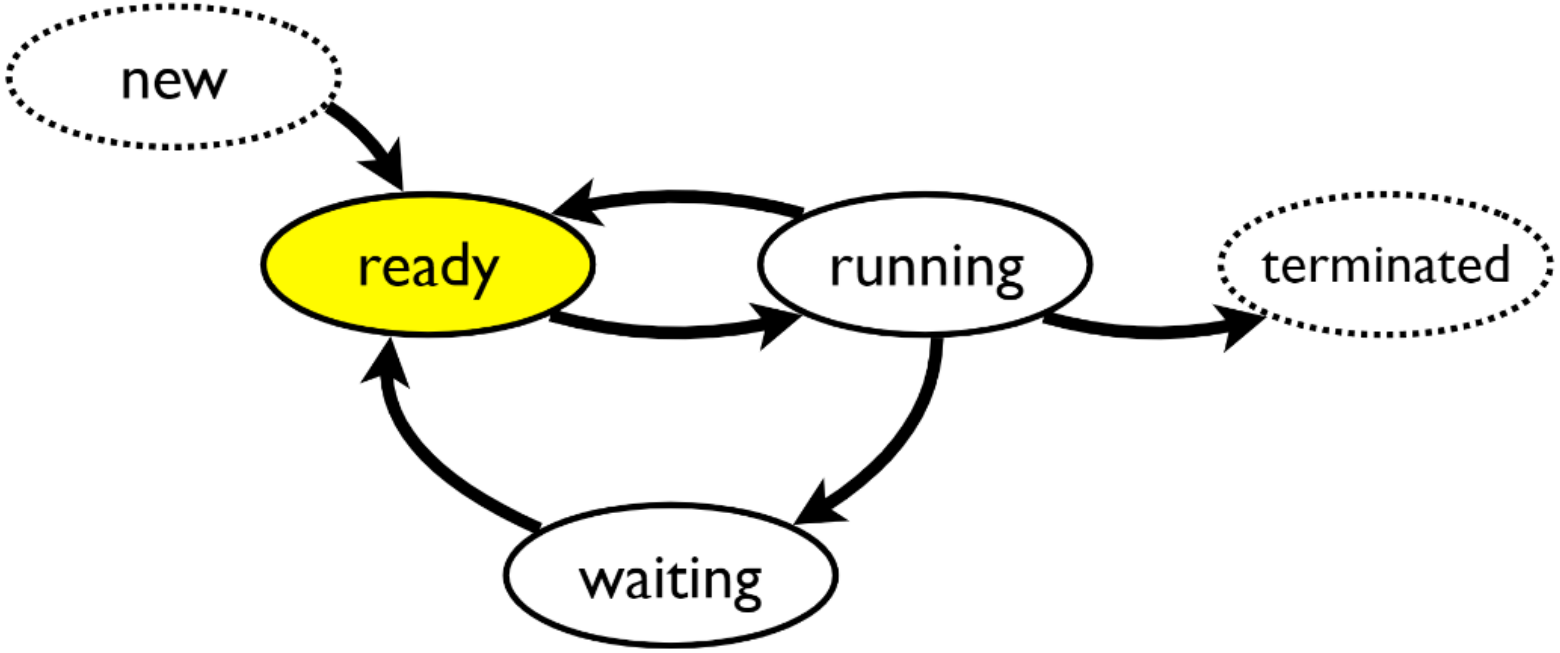


The operating system must allocates a blob of memory for the new process memory image.

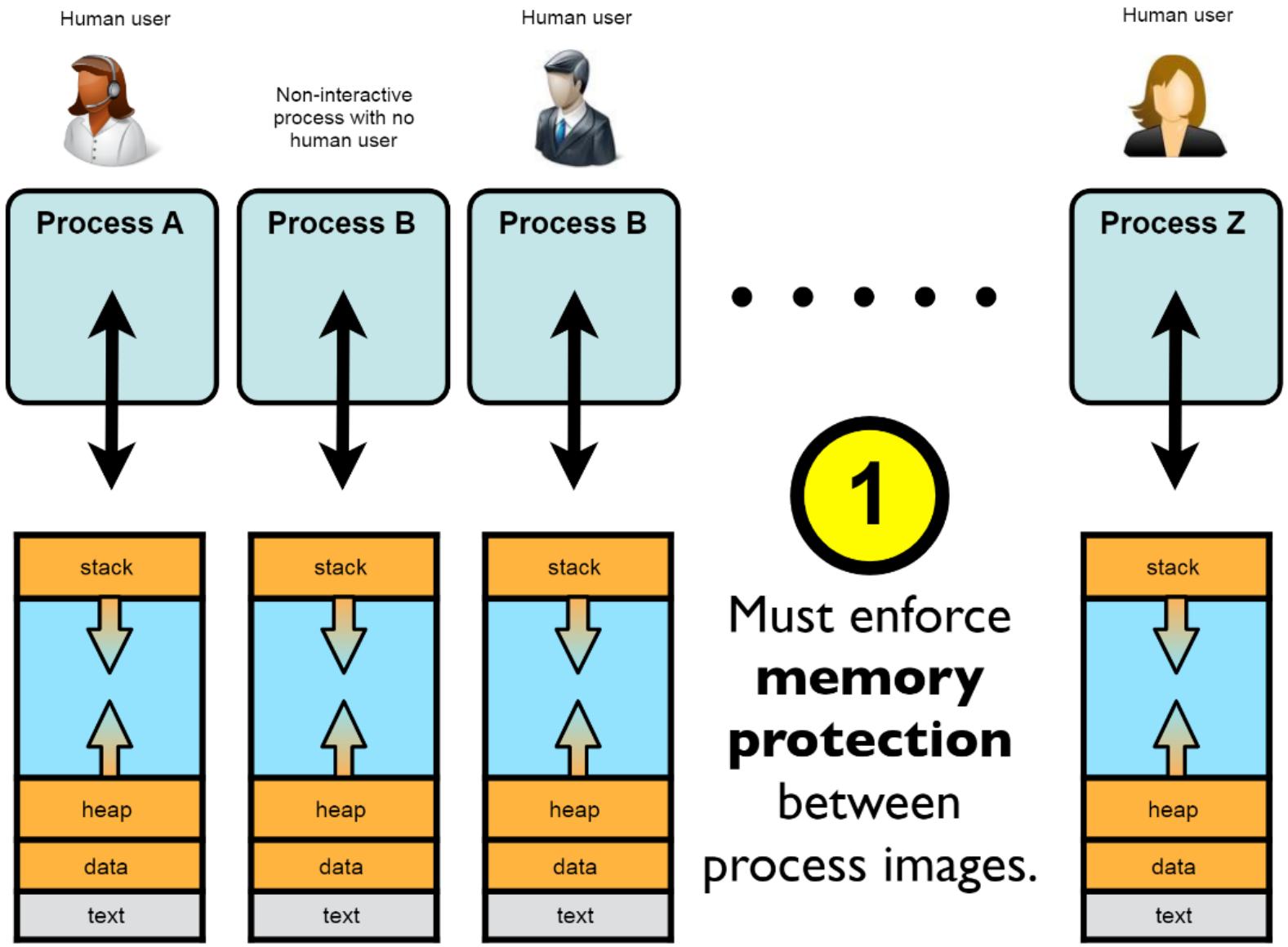

CPU
Disk controller
USB controller
Graphics adapter
Memory controller
Main memory
Process control block
Process id (PID)
Process state (new, ready, running, waiting or terminated)
CPU Context
I/O status information
CPU scheduling information
Memory management information
Process memory image
stack
heap
data
text

Ready

After the operating system has allocated a PCB and memory image for the new process it is ready to execute and changes state from new to ready.

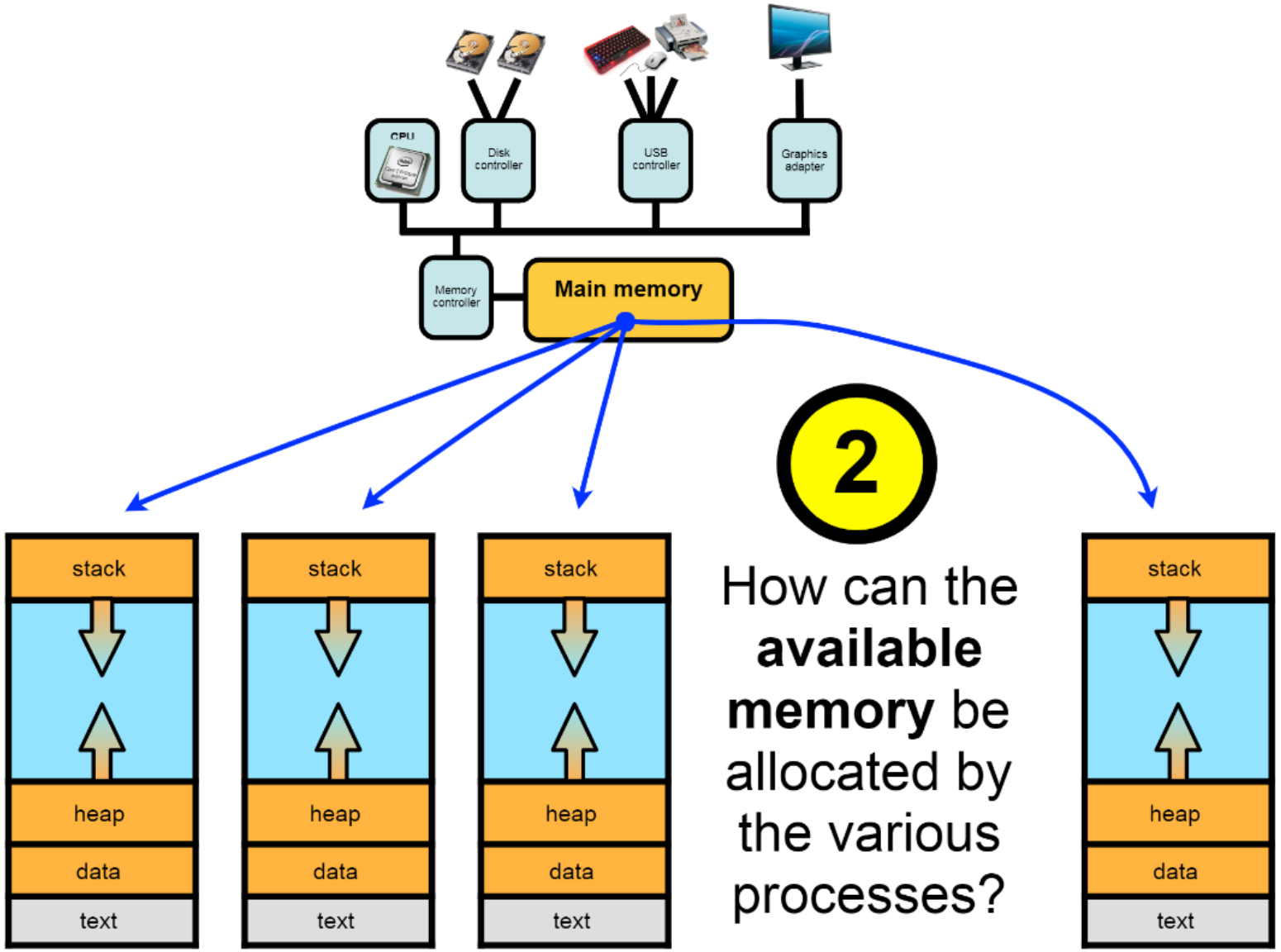


- A **program** must be brought (from disk) into memory and placed within a **process** for it to be run.
- Main **memory** and **registers** are the only storage CPU can access directly.
- **Register** access in one CPU clock (or less).
- **Main memory** can take many cycles.
- **Cache** sits between main memory and CPU registers.
- Must enforce **memory protection** between processes.



1

Must enforce **memory protection** between process images.



2

How can the **available memory** be allocated by the various processes?

Single contiguous allocation

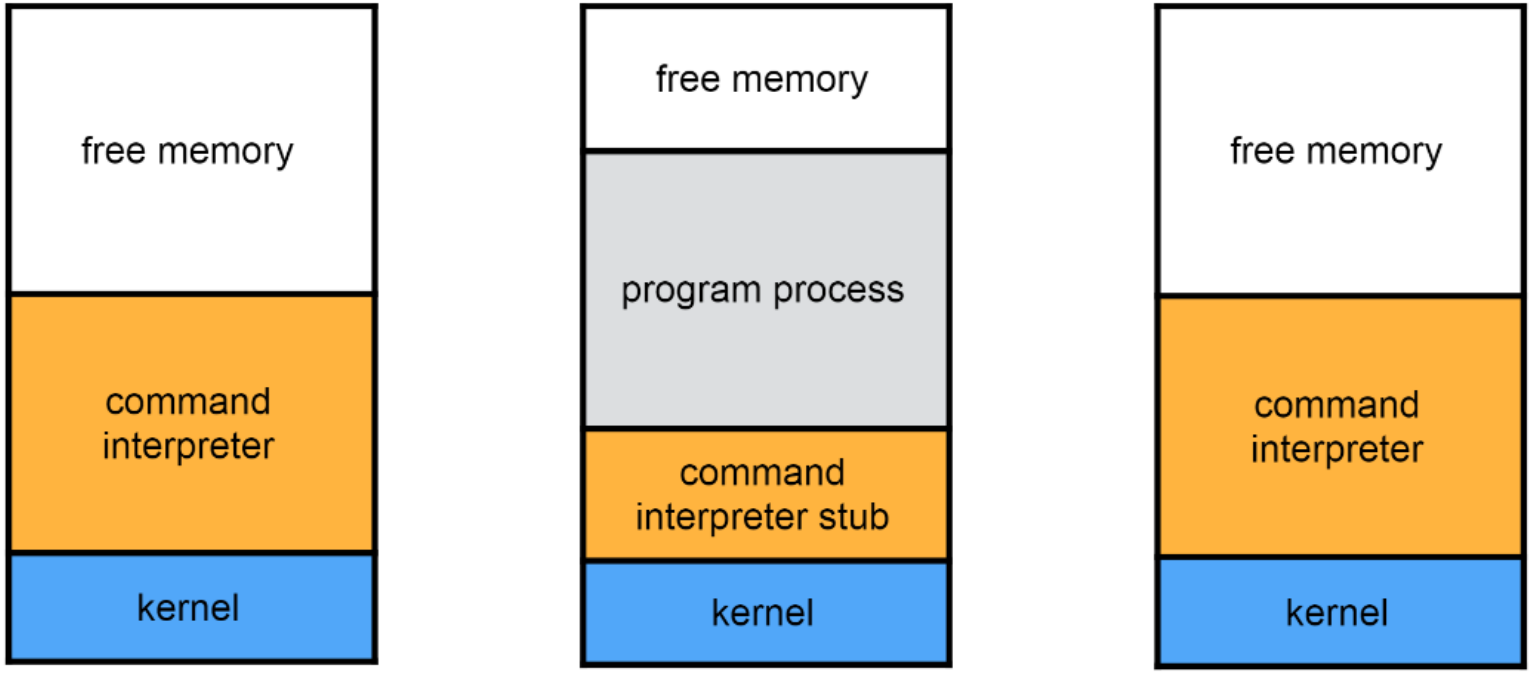
Single allocation is the simplest memory management technique. All the computer's memory, usually with the exception of a small portion reserved for the operating system, is available to the single application.

Single contiguous allocation

- **MS-DOS** (released 1981) is an example of a system which allocates memory in this way.
- An **embedded system** running a single application might also use this technique.
- A system using single contiguous allocation may still **multitask** by **swapping** the contents of memory to switch among users. Sometimes the term **single-tasking** is used instead of multi-tasking for such systems.

- MS-DOS (acronym for Microsoft Disk Operating System) is a operating system for x86-based personal computers mostly developed by Microsoft and initially **released 1981**.
- During its lifetime, several competing products were released for the x86 platform, and MS-DOS went through eight versions, until **development ceased in 2000**.
- MS-DOS is a **single-tasking** operating system.

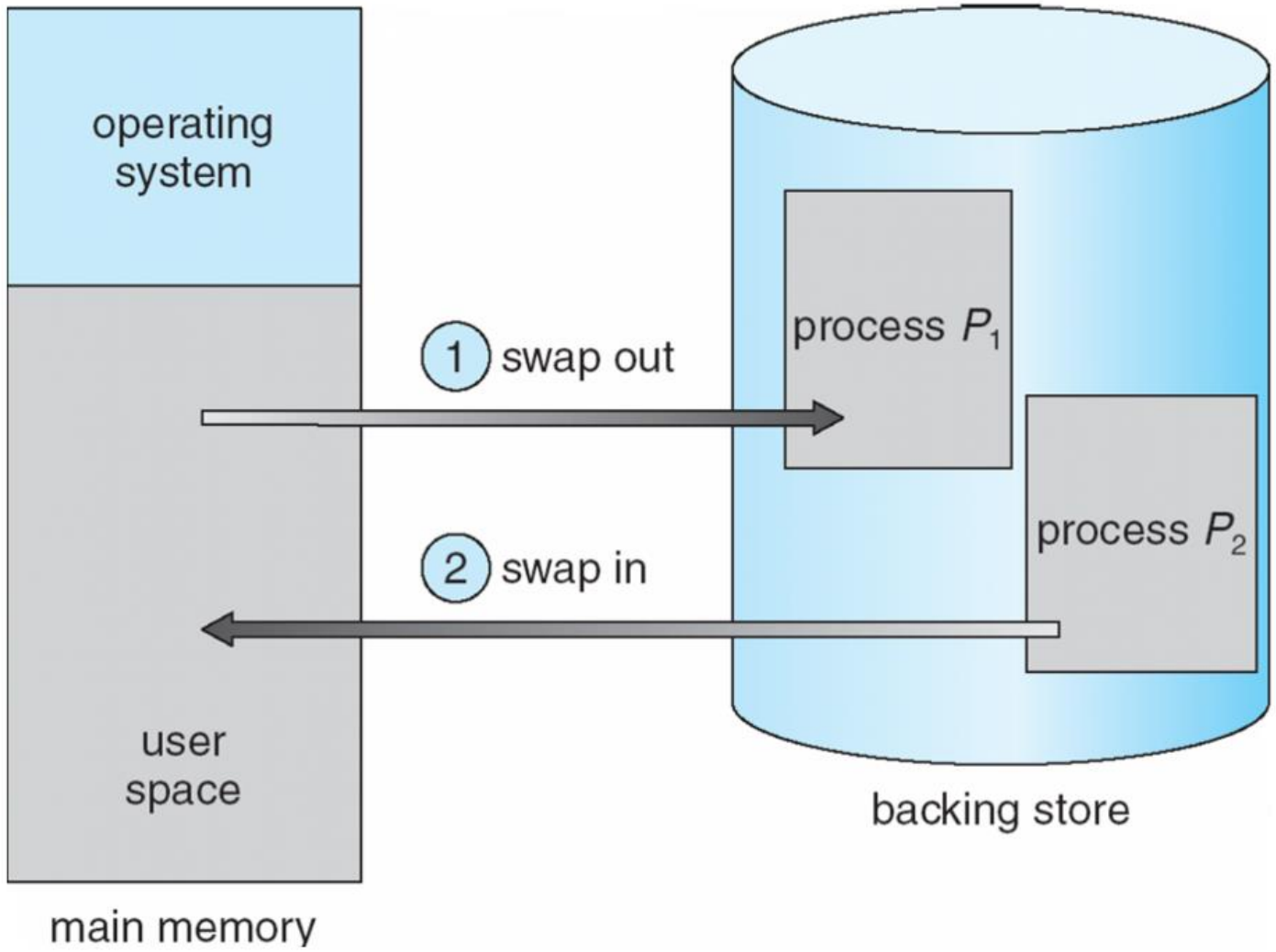
- MS-DOS is a single-tasking operating system. It has a command interpreter that is invoked when the computer is started.
- MS-DOS simply loads a program into memory, writing over most of the command interpreter to give the program as much memory as possible.
- When the program terminates, the small portion of the command interpreter that was not overwritten resumes execution. Its first task is to reload the rest of the command interpreter from disk.



Swapping

A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution.

- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images.
- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed.
- Major part of swap time is **transfer time**; total transfer time is directly proportional to the amount of memory swapped
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows).
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk.


operating system
1 swap out
process P_1
process P_2
2 swap in
user space
backing store
main memory

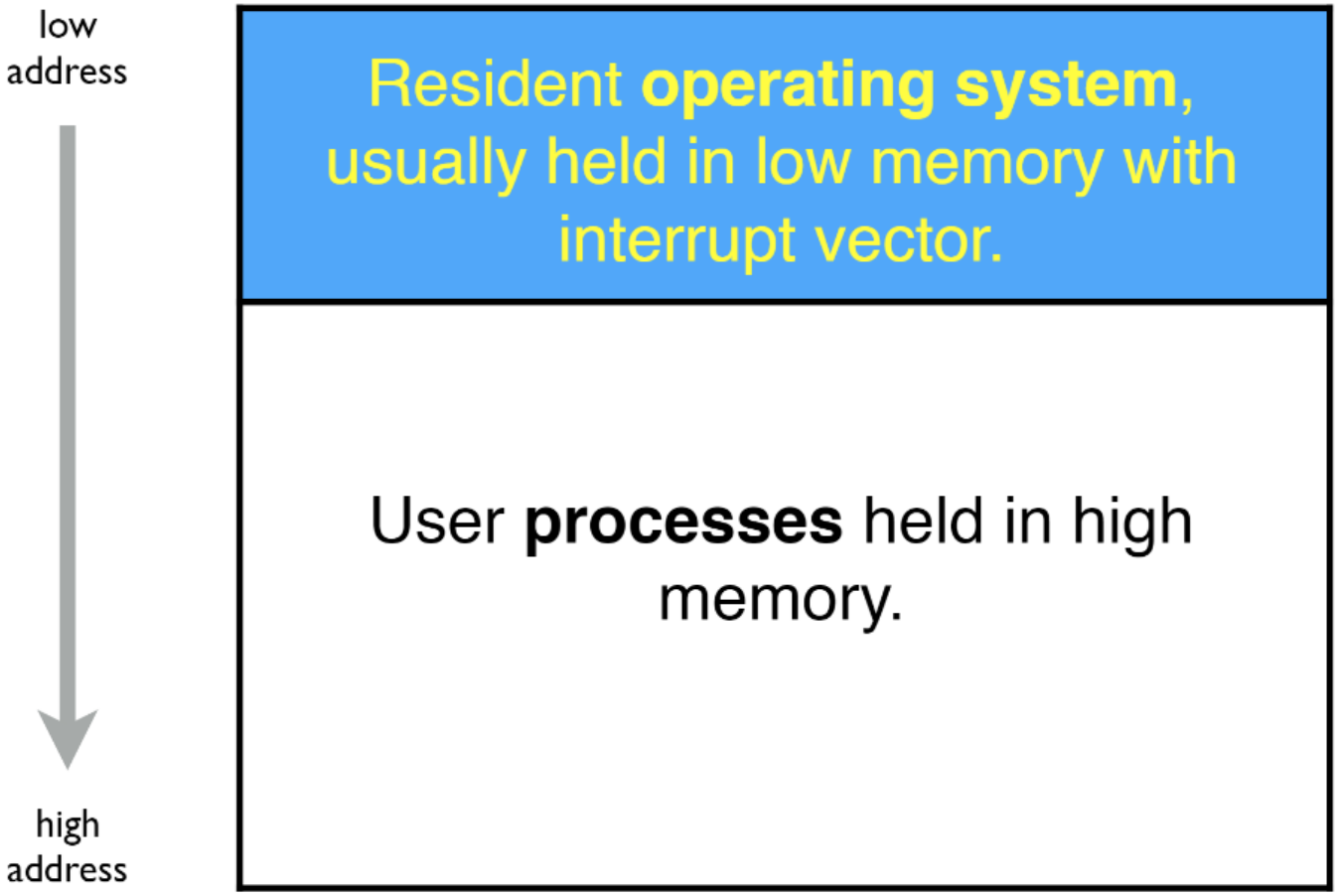
Partitioned allocation

Partitioned allocation divides primary memory into multiple memory partitions, usually contiguous areas of memory.

Partitioned allocation

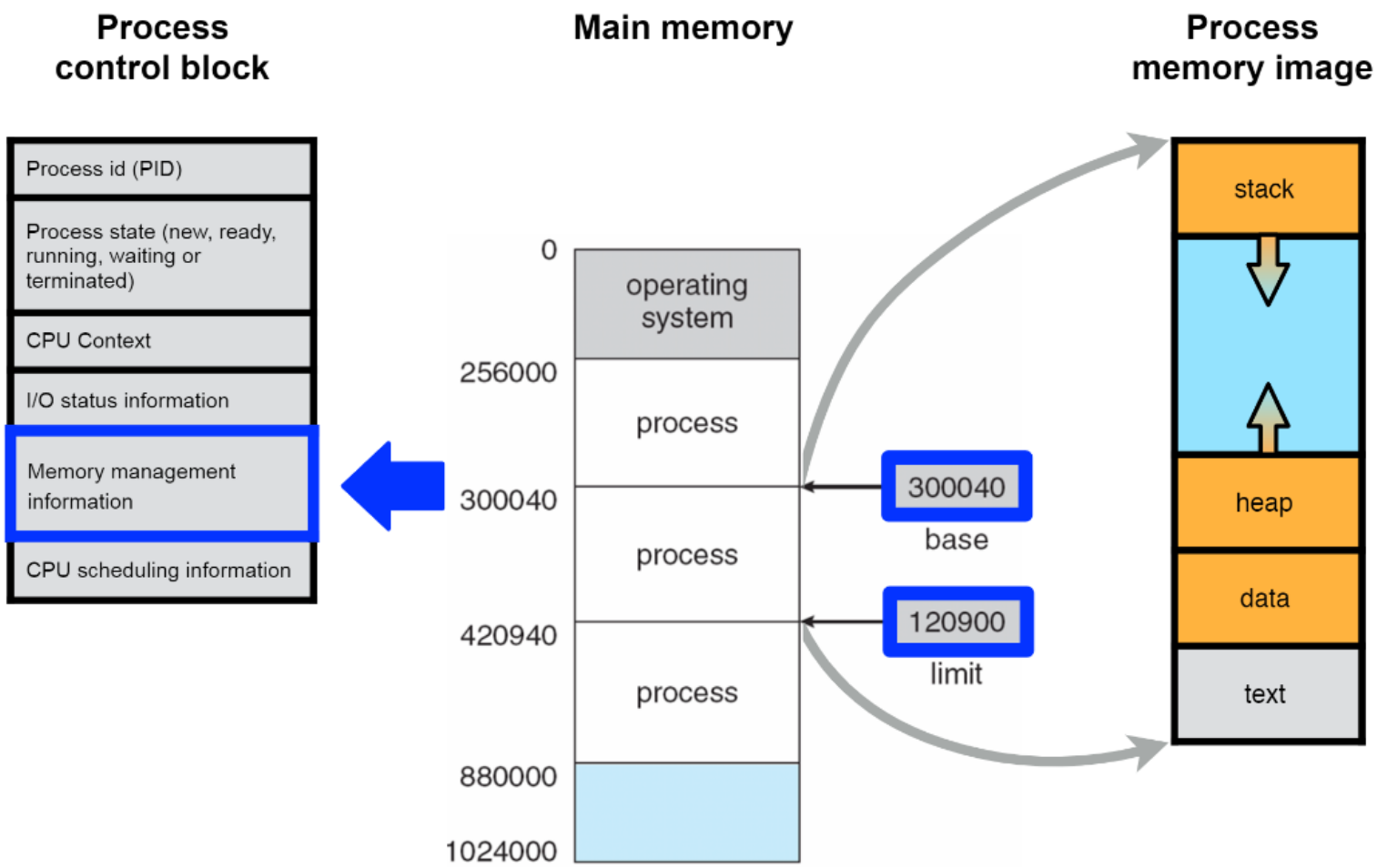
- Partitioned allocation divides primary memory into multiple memory partitions, **usually contiguous** areas of memory.
- Each partition might contain all the information for a specific job or task.
- Memory management consists of allocating a partition to a job when it starts and unallocating it when the job ends.

Main memory usually divided into two partitions: one for the operating system and one for all the processes in the system.



Logical address space

A logical address is the address at which a memory cell appears to reside from the perspective of an executing application program.



A pair of **base** and **limit** registers **define** a process **logical address space**.

CPU

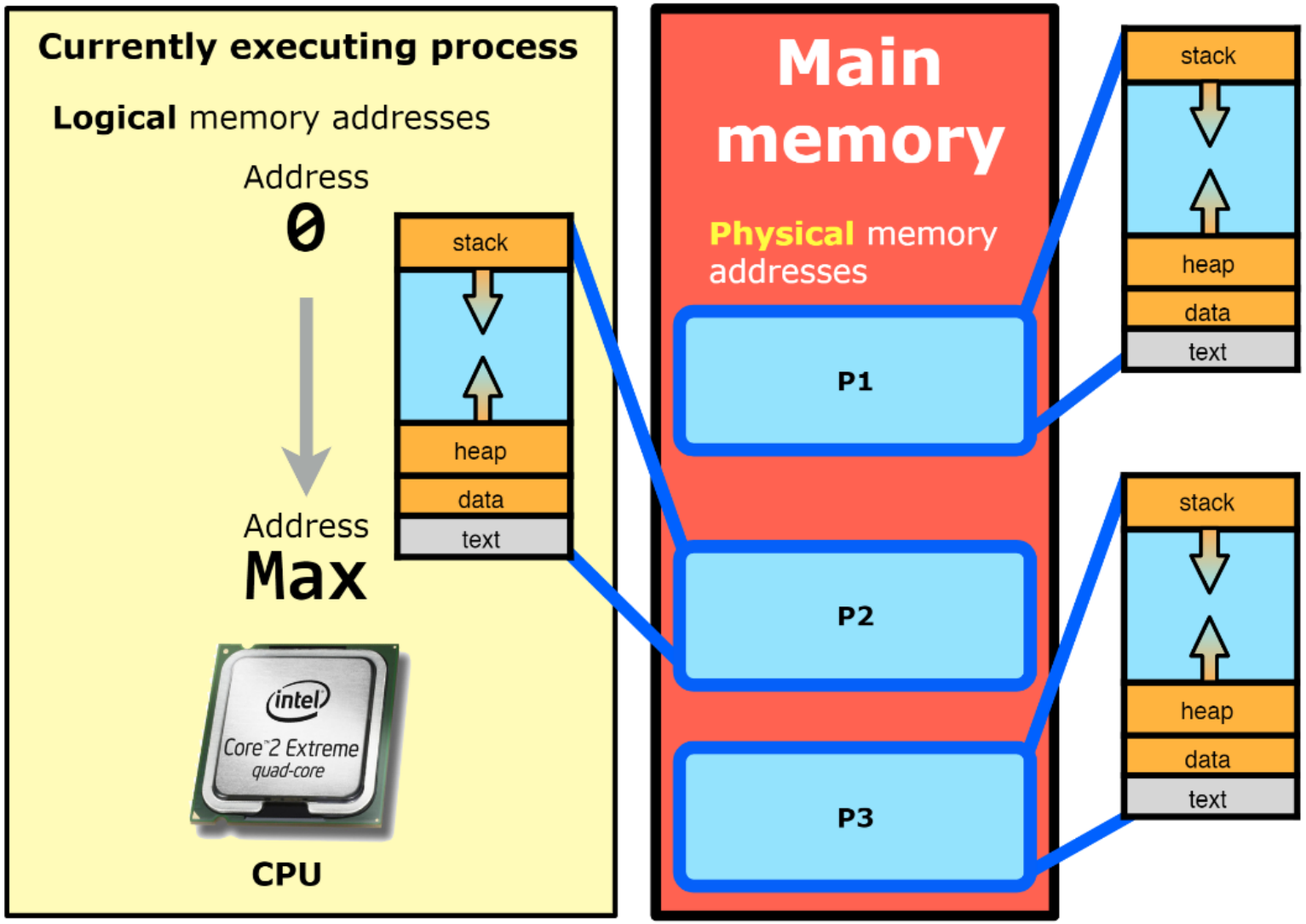
Main memory

Physical memory addresses

low addresses

high addresses

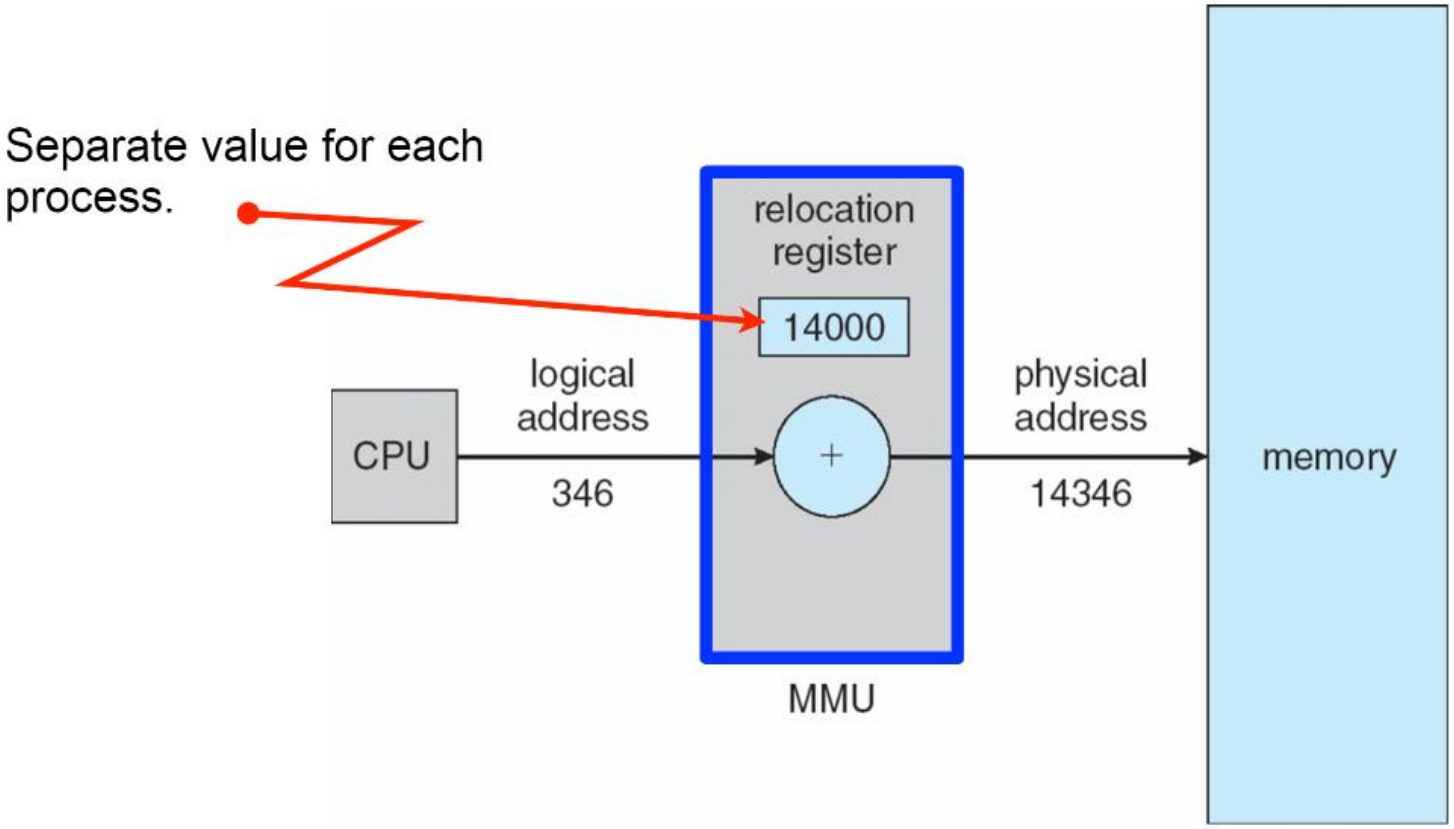
Logical and physical memory addresses

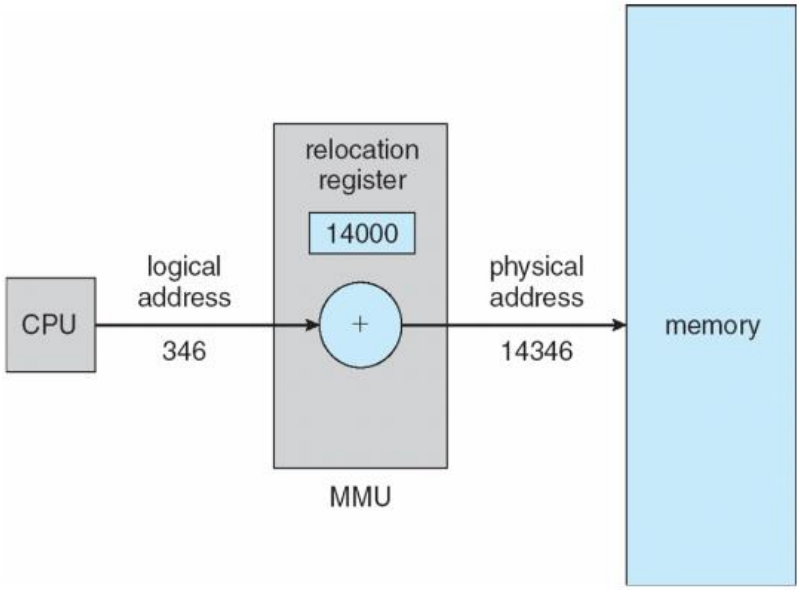


Memory management unit (MMU)

The MMU is a computer hardware unit having all memory references passed through itself, primarily performing the **translation** of **logical memory addresses to physical addresses**.

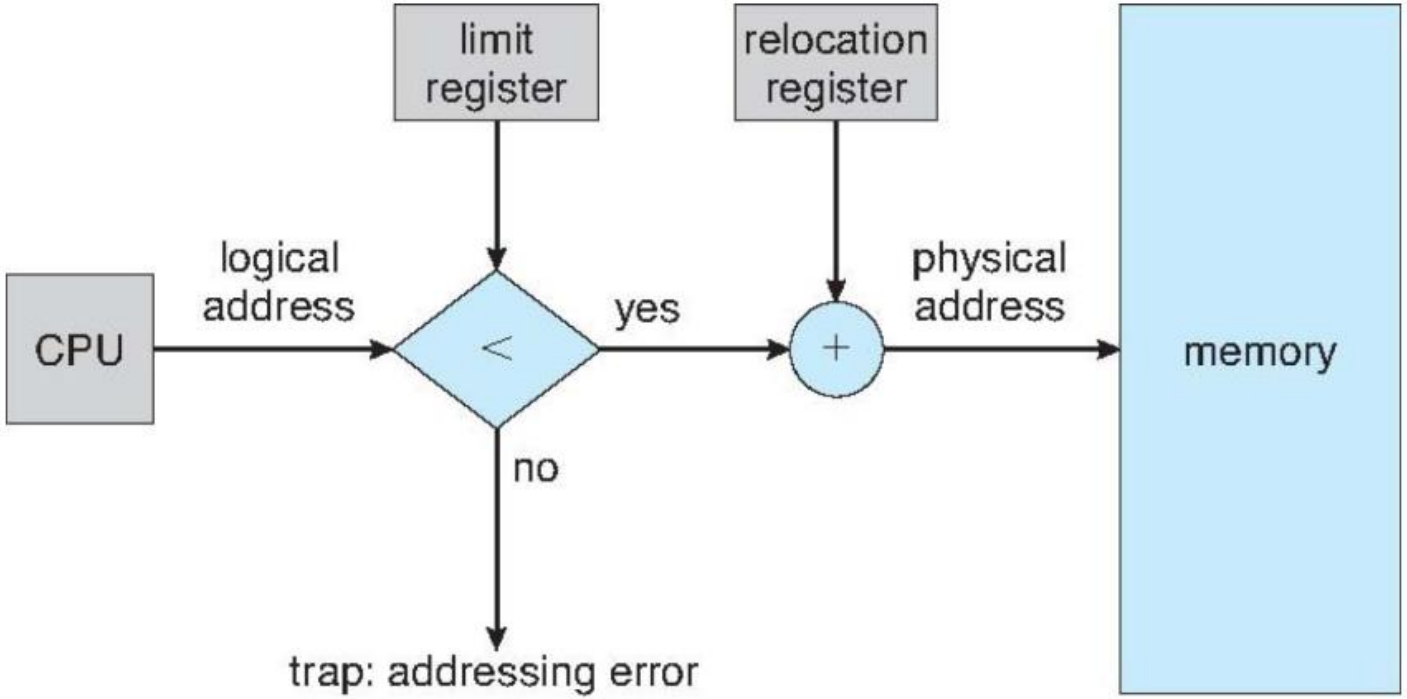
A simple MMU for partitioned contiguous allocation



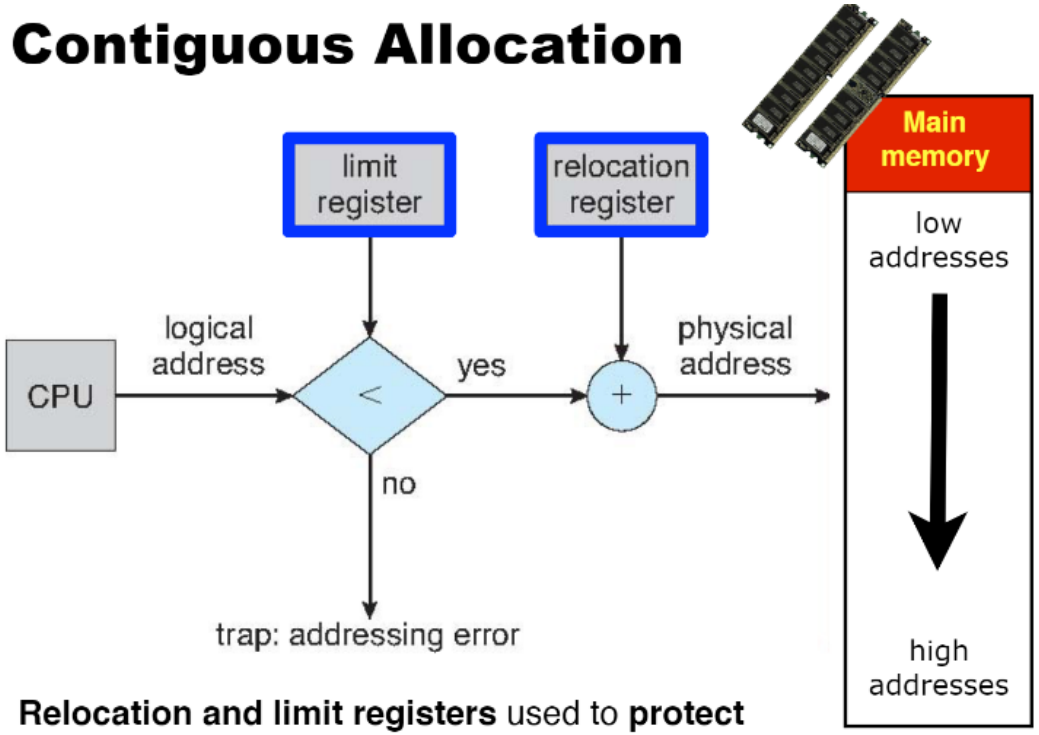


- Dynamic relocation using a **relocation register**.
- MMU **maps logical addresses to physical addresses.**
- The value in the relocation register is added to every address generated by a user process at the time it is sent to memory.
- The user program deals with logical addresses; it never sees the real physical addresses.

Must check if the address is within the boundaries of the process memory image.



Contiguous Allocation



Main memory usually divided into two partitions:

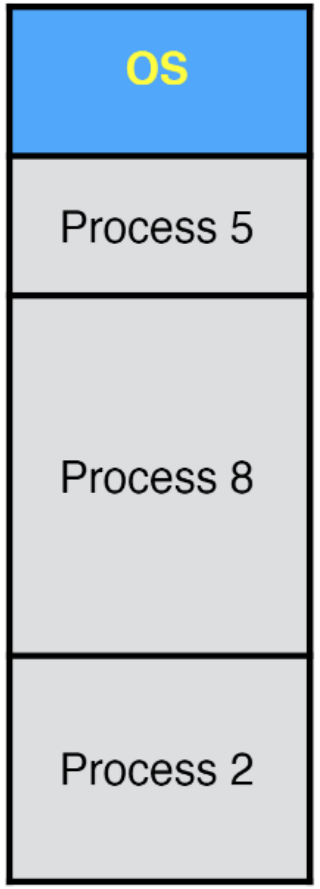
- **Resident operating system**, usually held in low memory with **interrupt vector**.
- **User processes** then held in high memory

Relocation and limit registers used to **protect**

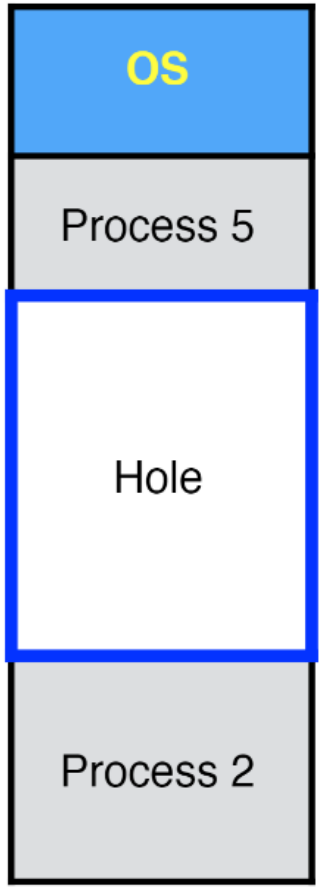
- user processes from each other
- operating-system code and data from user processes
  - **Relocation register** contains value of smallest physical address
  - **Limit register** contains range of logical addresses – each logical address must be less than the limit register
  - **MMU** maps logical address dynamically

Dynamic memory management

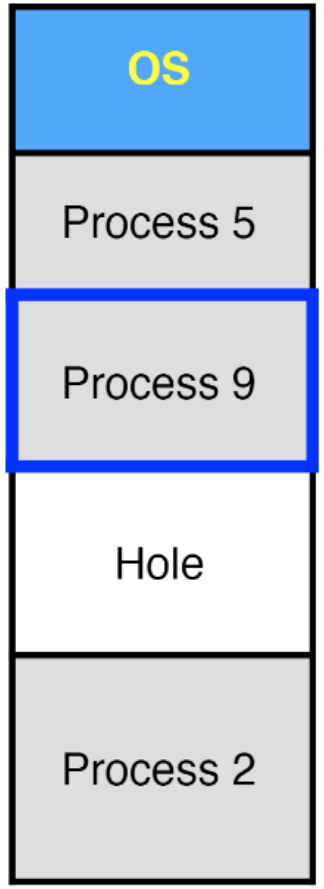
Three process occupies all available memory.



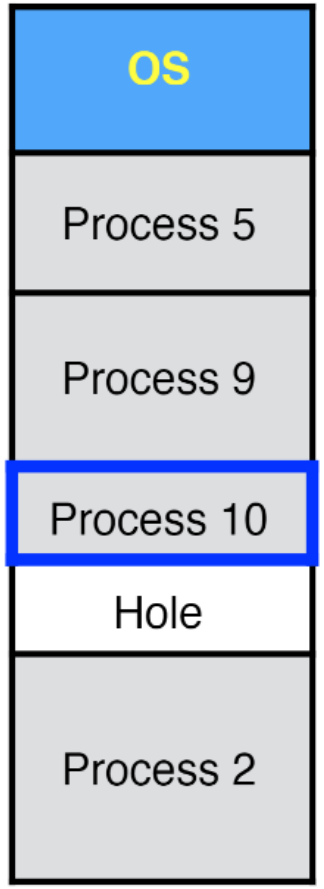
Process 8 terminates and leaves a free hole.

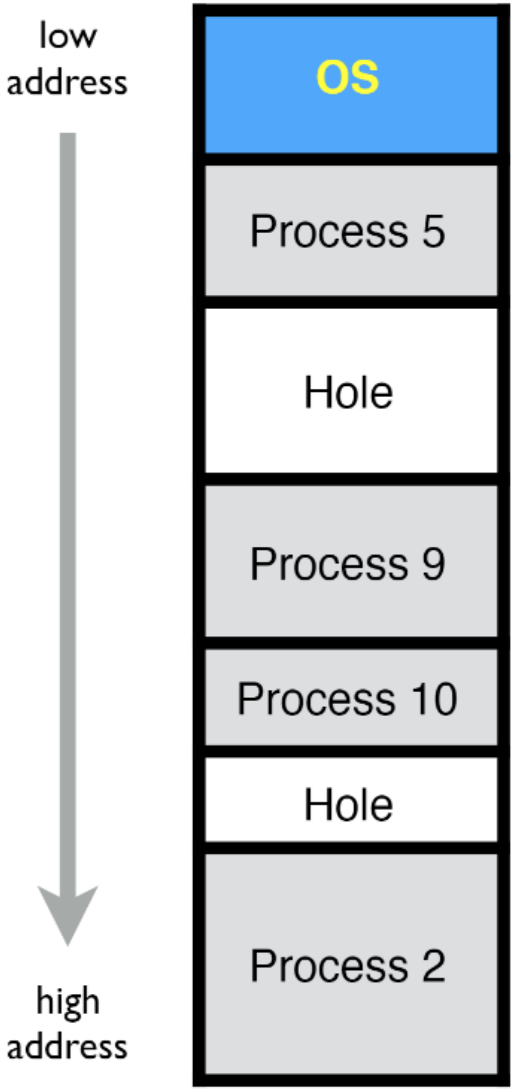


Process 9 allocates part of the hole left by process 8 and the hole of free memory shrinks.



The hole of free memory gets smaller and smaller





Hole – block of available free memory.

Holes of various size are **scattered** throughout memory.

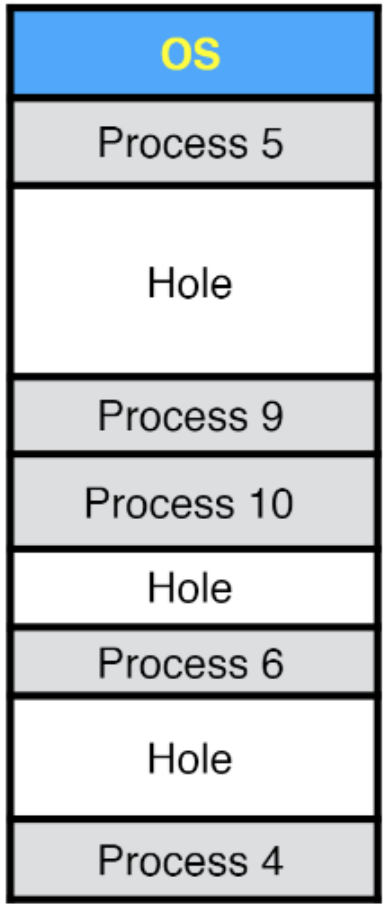
When a process arrives, it is **allocated memory from a hole large enough** to accommodate it.

Operating system maintains information about:

- Allocated partitions.
- Free partitions (holes).

Dynamic storage allocation problem

How to **satisfy** a request of **size n** from a **list** of **free holes**?

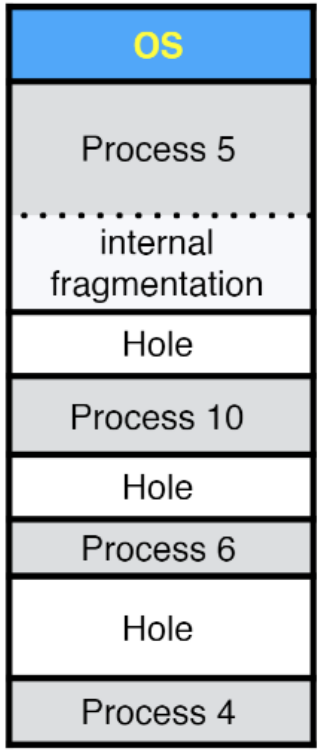


- **First-fit:** Allocate the first hole that is big enough
- **Best-fit:** Allocate the smallest hole that is big enough; must search entire list, unless ordered by size
  - Produces the smallest leftover hole
- **Worst-fit:** Allocate the largest hole; must also search entire list
  - Produces the largest leftover hole

Simulations show that first-fit and best-fit performs better than worst-fit in terms of speed and storage utilization.

Fragmentation

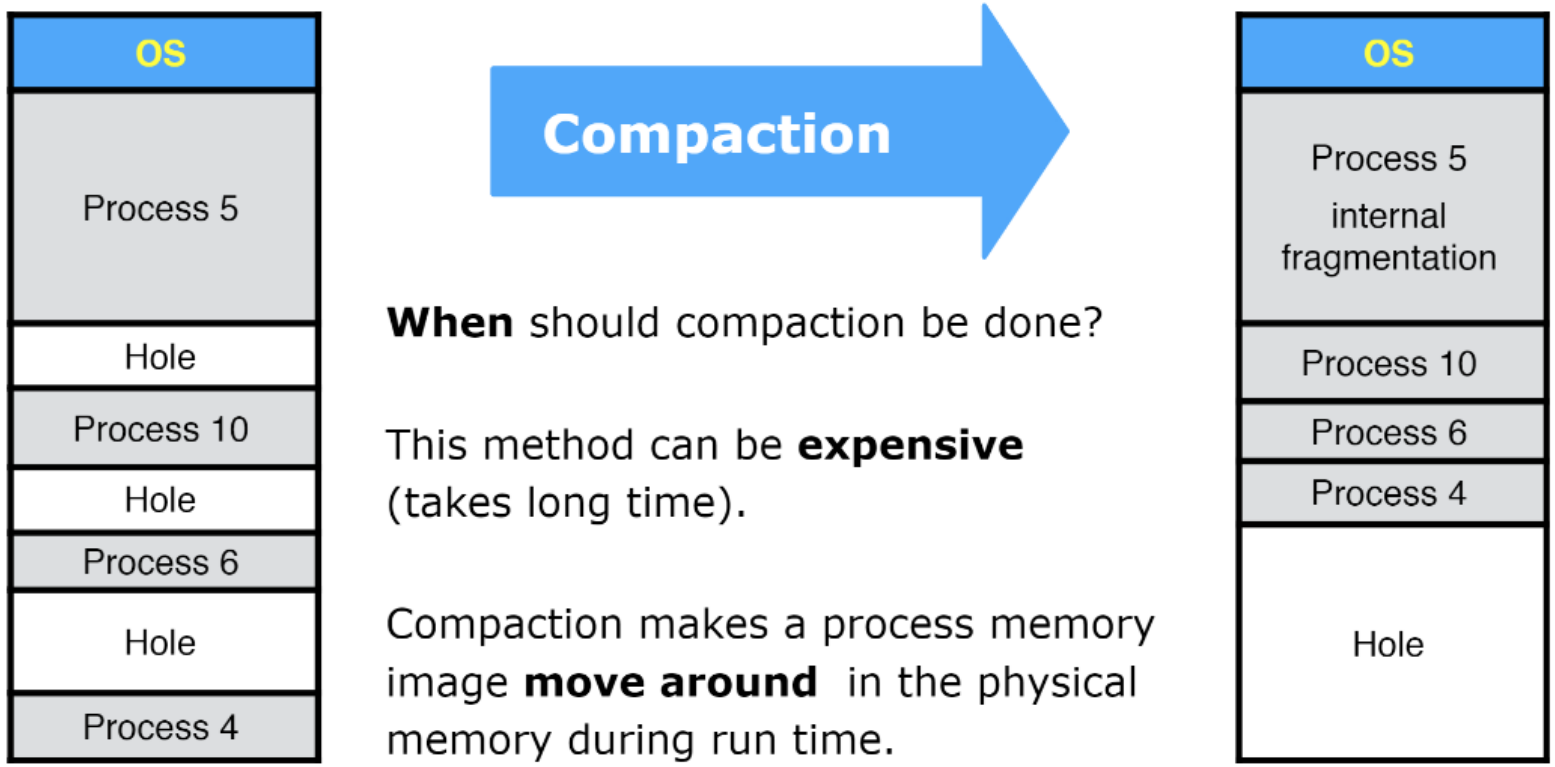
Fragmentation is a phenomenon in which storage space is used inefficiently, reducing capacity and often performance.



- **Fragmentation** leads to storage space being "wasted", and the term also refers to the wasted space itself.
- **Internal fragmentation:** allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.
- **External fragmentation:** total memory space exists to satisfy a request, but it is not contiguous.

Compaction

Move all processes to one end of the address space producing one large hole at the other end of the address space.



When should compaction be done?

This method can be **expensive** (takes long time).

Compaction makes a process memory image **move around** in the physical memory during run time.

Paged memory management

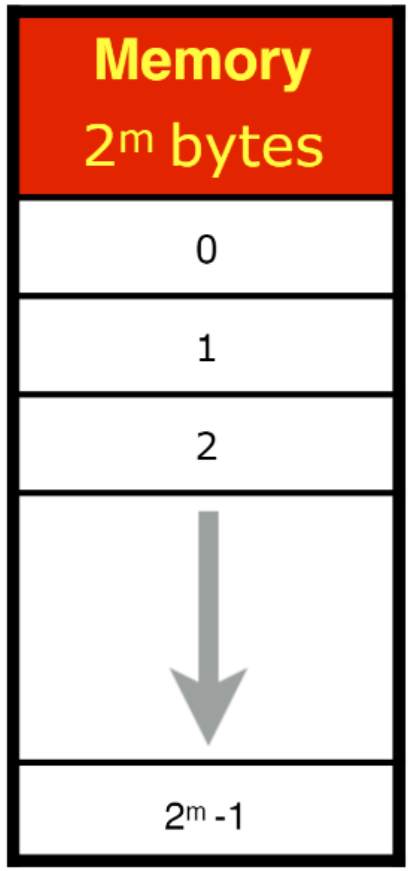
Paged allocation divides the computer's primary memory into **fixed-size** units called page **frames**, and the program's virtual address space into **pages** of the **same size**.

The hardware memory management unit **maps pages to frames**.

The **physical** memory can be **allocated non-contiguous** on a page basis while the logical address space appears contiguous.

Addressing the memory

To address the memory, how many bits do we need?



To address $\mathbf{2^m}$ locations,
we need a **m** bit address.

A =

m bit address

Frames

Physical memory of size **2^m bytes** divide into frames of size **2^n bytes** each.

Memory 2^m bytes	
Frame	**Size (bytes)**
0	2^n
1	2^n
2	2^n
↓	↓
$[2^{(m-n)}] -1$	2^n

We number the frames 0, 1, ...
How many frames do we get?

$$\frac{\text{Size of memory}}{\text{Size of frame}} = \frac{2^m}{2^n} = 2^{m-n}$$

We number the frames
0, 1, ..., $[2^{(m-n)}] -1$

Pick a frame (address translation)

How can we map a logical address as seen by the CPU to a physical frame in memory?

m bit logical address seen by the CPU

A =

Use **m-n** high order bits to pick a frame.

frame number ???

A =

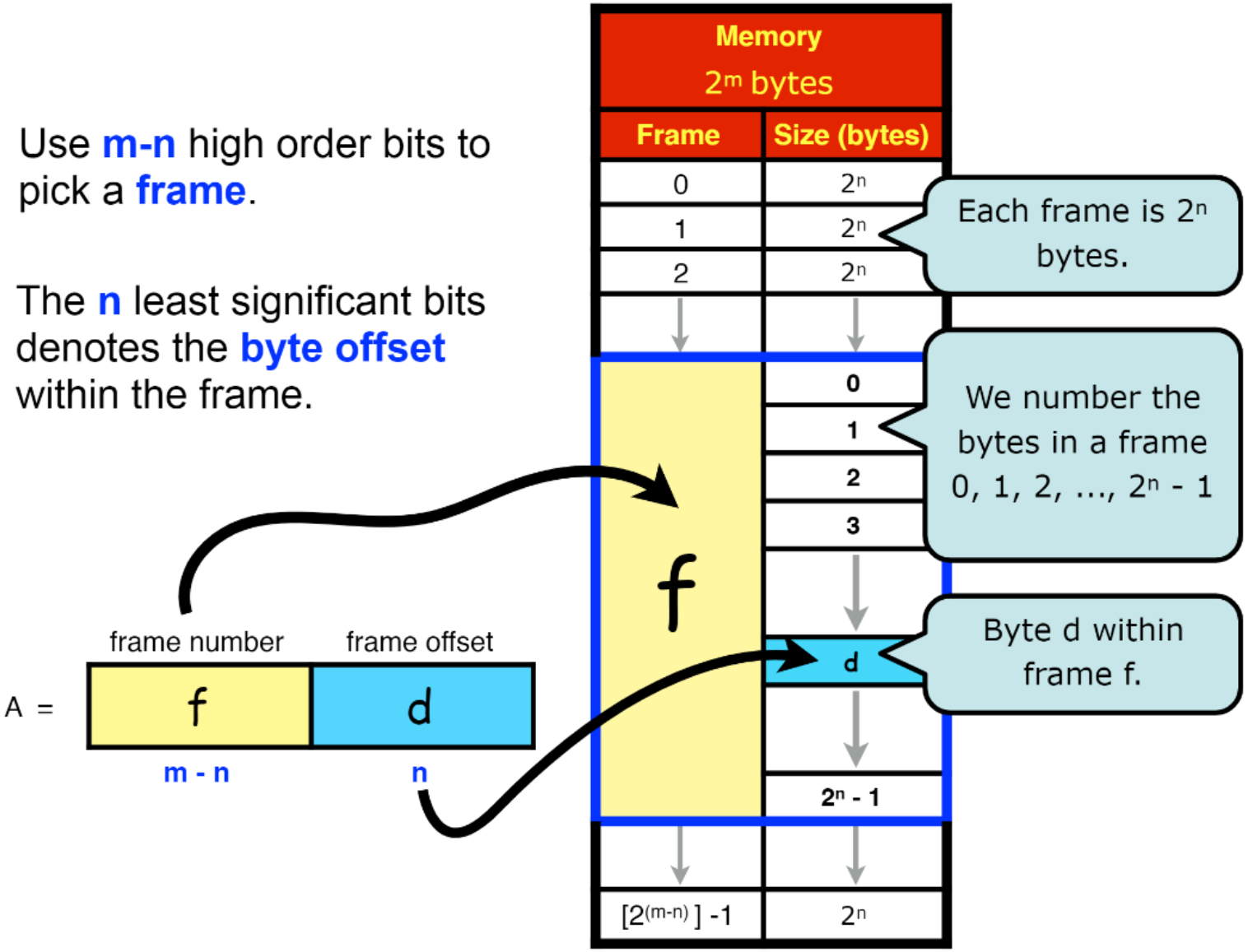
m - n **n**

What does the **n** least significant bits mean?

Memory 2^m bytes	
Frame	**Size (bytes)**
0	2^n
1	2^n
2	2^n
↓	↓
$[2^{(m-n)}] - 1$	2^n

Use **m-n** high order bits to pick a **frame**.

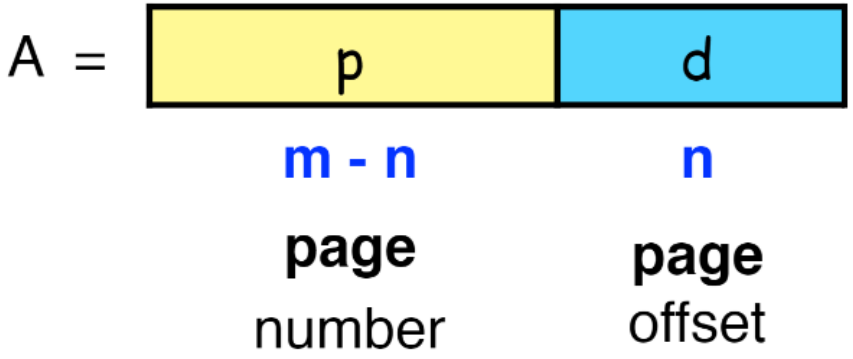
The **n** least significant bits denotes the **byte offset** within the frame.

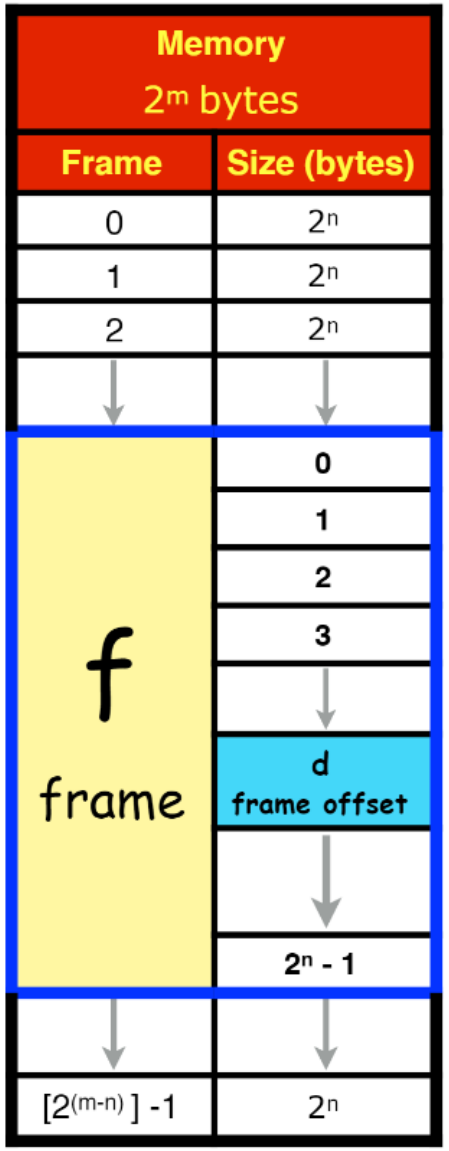


Pages and frames

A solution that allows for non-contiguous allocation of physical frames.

- **Logical** address space divided into fixed sized **pages**.
- **Physical** memory divided into **frames** of the same fixed size as the the pages.





Page table

A page lookup table maps logical pages to physical frames.

Page table	
Page	**Frame**
0	33
1	7111
↓	↓
127	42
↓	↓
$[2^{(m-n)}] - 1$	5666

	page number	page offset
A =	127	d
	m - n	n

Page table	
Page	**Frame**
0	33
1	7111
↓	↓
127	42
↓	↓
$[2^{(m-n)}] - 1$	5666

Memory 2^m bytes	
Frame	**Size (bytes)**
0	2^n
↓	↓
42	0
	1
	2
	3
	↓
	d
	↓
	$2^n - 1$
↓	↓
$[2^{(m-n)}] - 1$	2^n

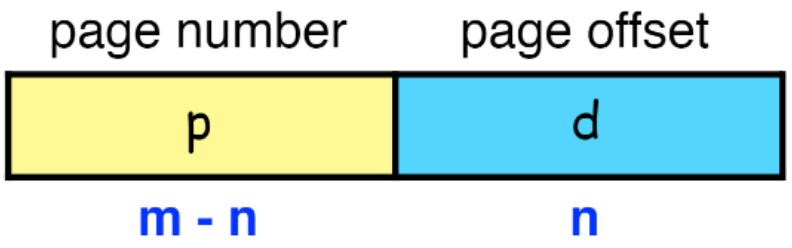
Paged memory management

- **Logical address space** of a process can be **noncontiguous**; process is **allocated physical memory whenever** the latter is **available**.
- Divide **physical memory** into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 8,192 bytes).
- Divide **logical memory** into blocks of same size called **pages**.
- Keep track of all **free frames**.
  - To run a program of size n pages, need to find n free frames and load program.
- Set up a **page table** to translate logical to physical addresses.
- **Internal fragmentation** possible.
- No **external fragmentation**.

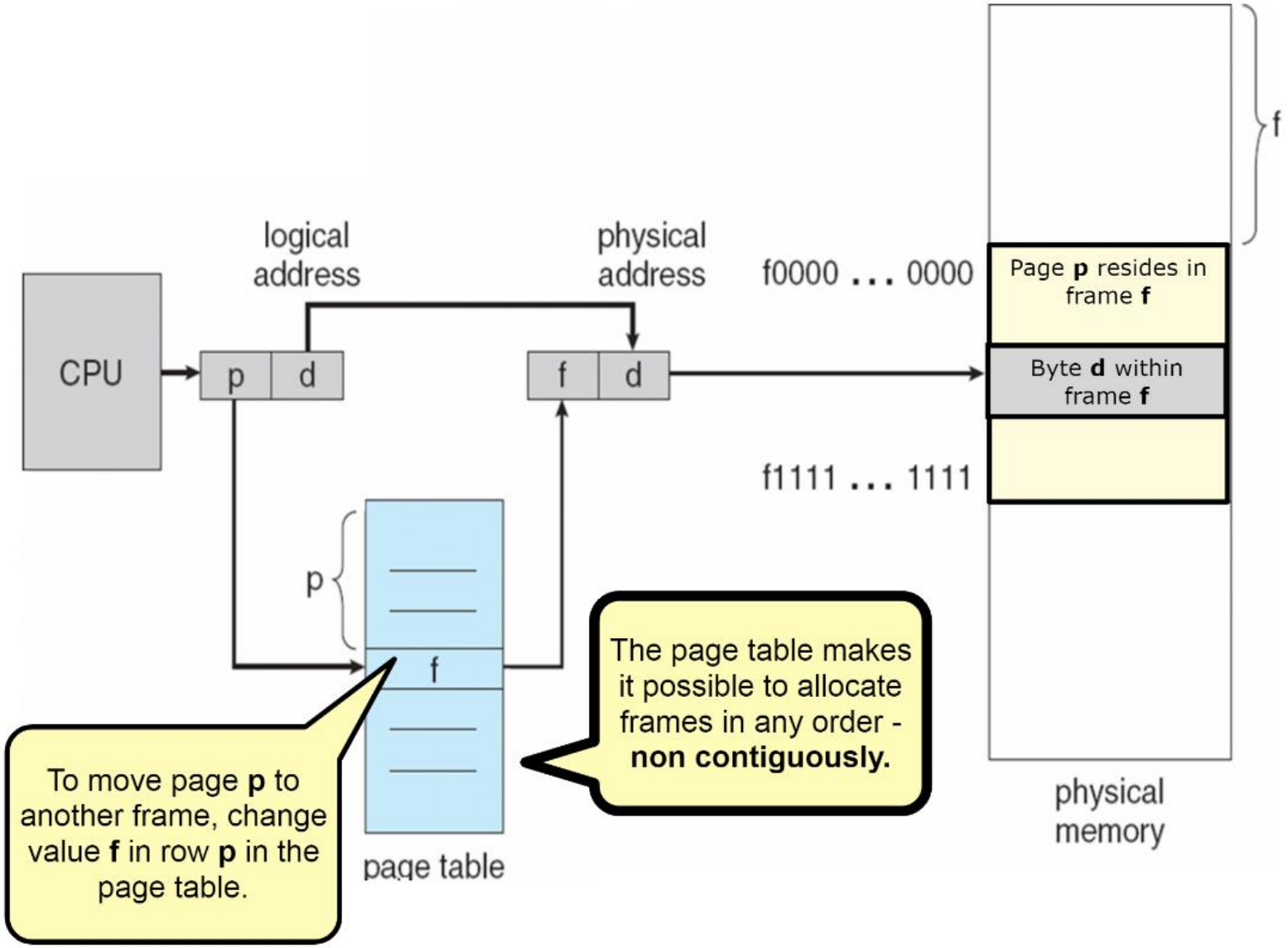
Address translation scheme

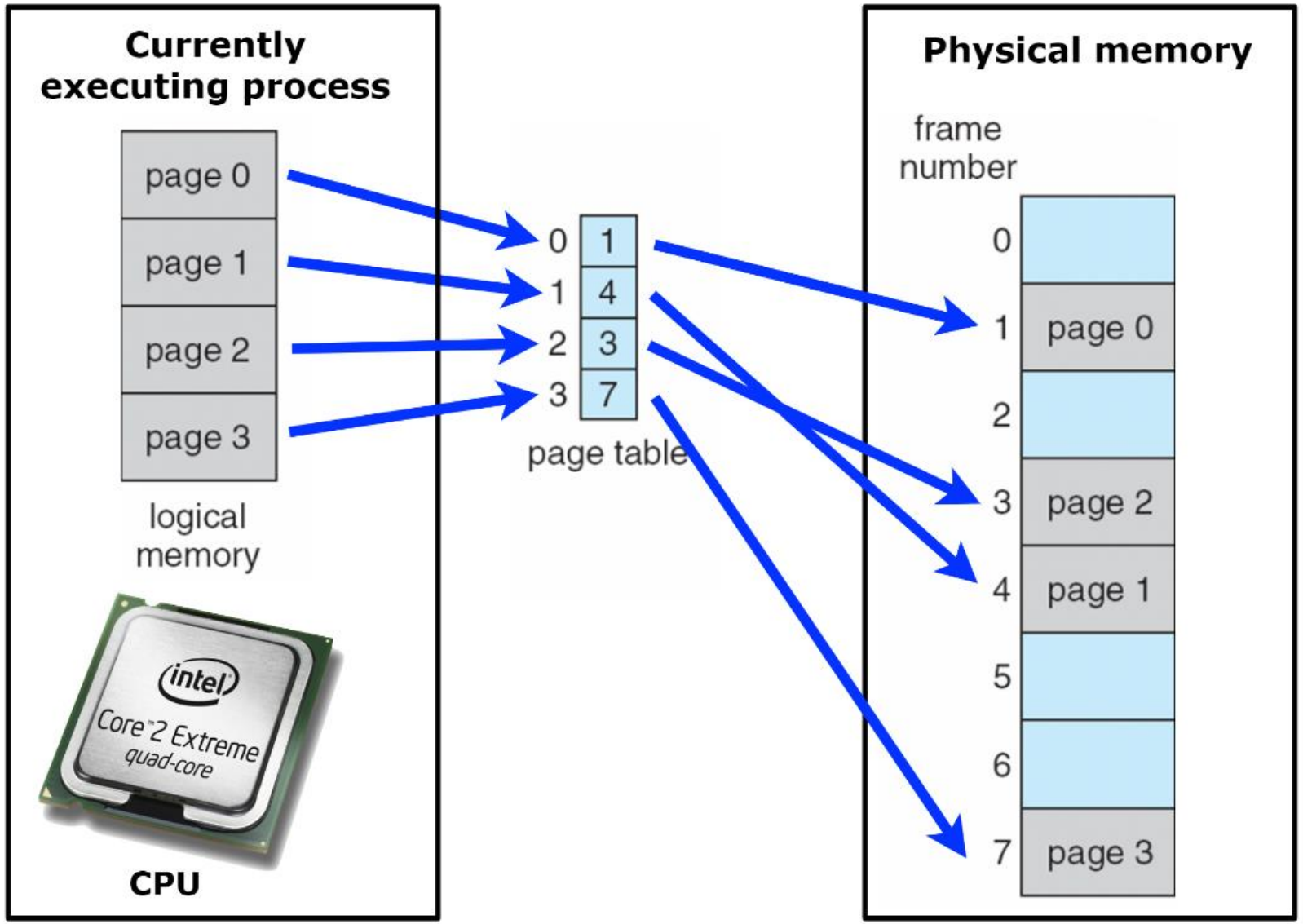
Logical address generated by CPU is divided into:

- **Page number** (p) – used as an index into a page table which contains base address of each frame in physical memory
- **Page offset** (d) – combined with base address to define the physical memory address that is sent to the memory unit.



For given logical address space 2^m and page size 2^n


logical address
physical address
CPU
p
d
f
d
f0000 . . . 0000
Page p resides in frame f
Byte d within frame f
f1111 . . . 1111
f
p
f
page table
physical memory
To move page p to another frame, change value f in row p in the page table.
The page table makes it possible to allocate frames in any order - non contiguously.


Currently executing process
page 0
page 1
page 2
page 3
logical memory
intel
Core 2 Extreme
quad-core
CPU
0 1
1 4
2 3
3 7
page table
Physical memory
frame number
0
1 page 0
2
3 page 2
4 page 1
5
6
7 page 3

Example: 32 byte physical memory and 4-byte pages.

How many frames?

(32 bytes)/[(4 bytes)/frame)] = 8 frames

How large physical address?

$_2\log(32) = {}_2\log(2^5) = 5$ bit address

How is the physical address divided into frame number and frame offset?

$_2\log(8) = {}_2\log(2^3) = 3$ bits for frame number

$_2\log(4) = {}_2\log(2^2) = 2$ bits for frame offset

	frame number	frame offset
$A_{physical}$ =	f	d
	3 bit	2 bit

5 bit physical address

Physical memory 32 bytes		
byte	**frame**	**offset**
0	0	0
1		1
2		2
3		3
4	1	0
5		1
6		2
7		3
8	2	0
9		1
10		2
11		3
12	3	0
13		1
14		2
15		3
16	4	0
17		1
18		2
19		3
20	5	0
21		1
22		2
23		3
24	6	0
25		1
26		2
27		3
28	7	0
29		1
30		2
31		3

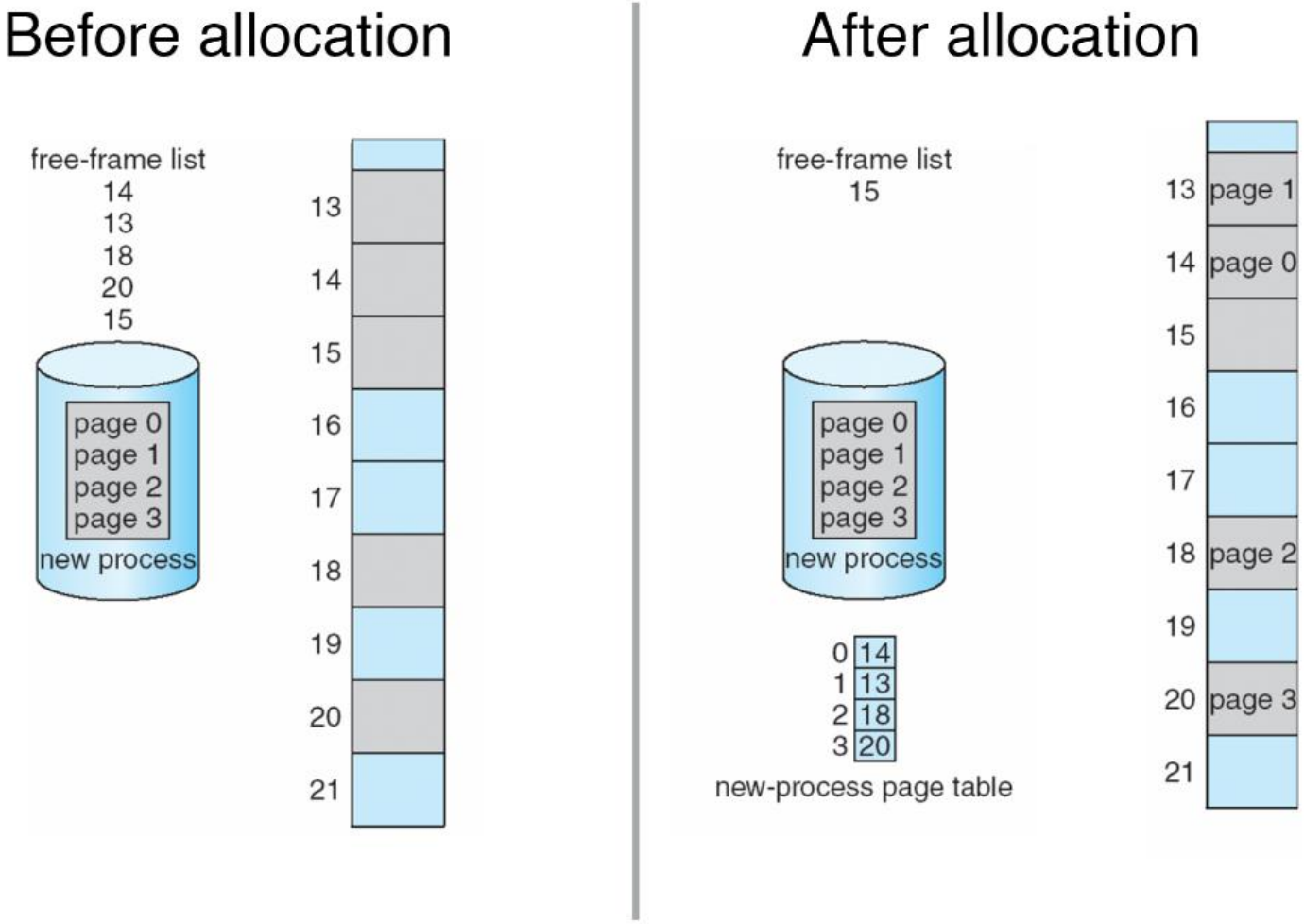
32 byte physical memory and 4 byte logical pages

Logical memory	
Page	Byte content
0	a
	b
	c
	d
1	e
	f
	g
	h
2	i
	j
	k
	l
3	m
	n
	o
	p

Page table	
p	f
0	5
1	6
2	1
3	2

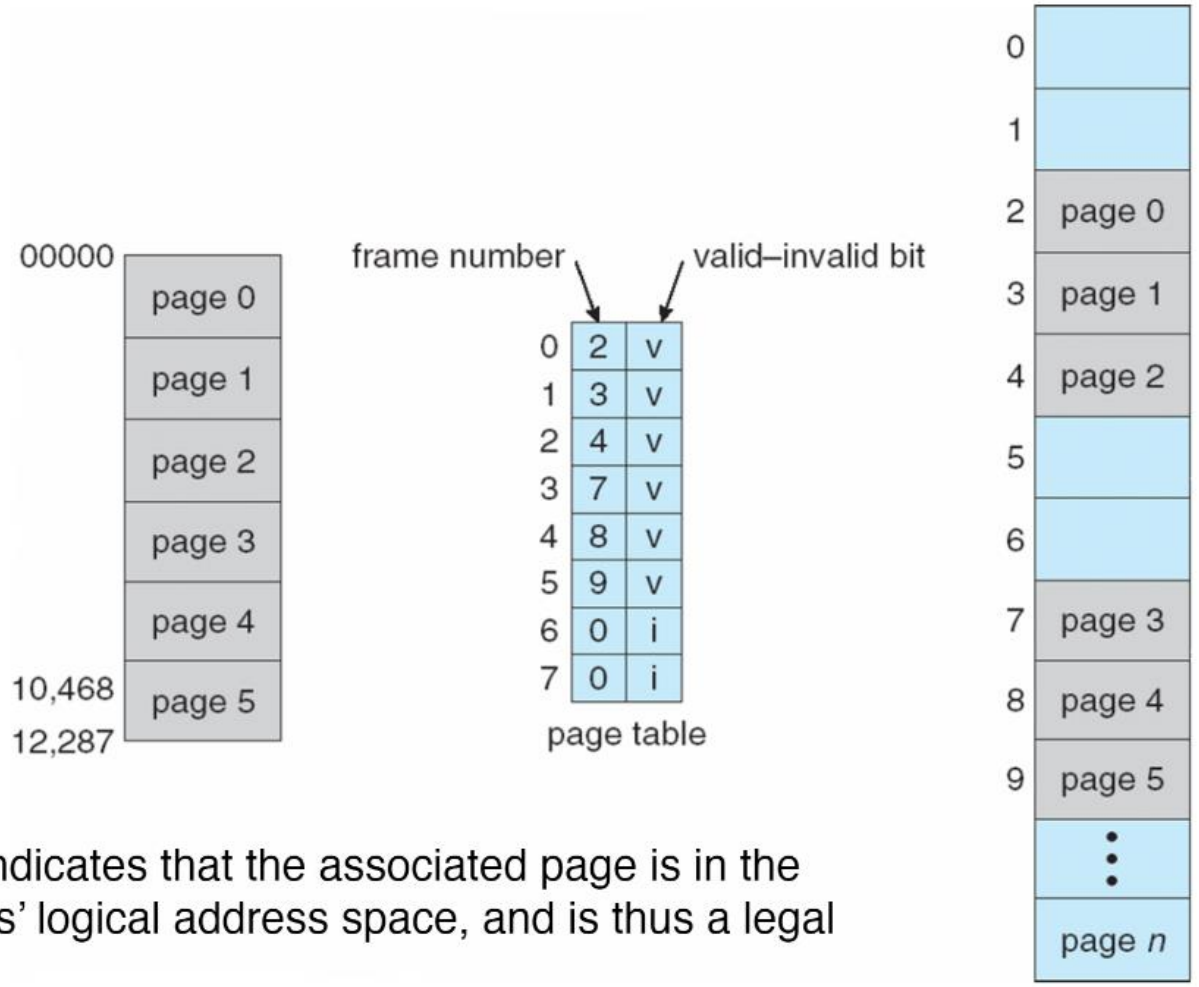
Physical Memory 32 bytes			
byte	frame	offset	content
0	0	0	
1		1	
2		2	
3		3	
4	1	0	i
5		1	j
6		2	k
7		3	l
8	2	0	m
9		1	n
10		2	o
11		3	p
12	3	0	
13		1	
14		2	
15		3	
16	4	0	
17		1	
18		2	
19		3	
20	5	0	a
21		1	b
22		2	c
23		3	d
24	6	0	e
25		1	f
26		2	g
27		3	h
28	7	0	
29		1	
30		2	
31		3	

List of free frames



Memory protection

Memory protection between processes implemented by associating a **valid bit** with each entry in the per process page table.



- **valid** indicates that the associated page is in the process' logical address space, and is thus a legal page.
- **invalid** indicates that the page is not in the process' logical address space.

Translation lookaside buffer

A translation lookaside buffer (TLB) is a memory **cache** that is used to reduce the time taken to access a user memory location.

The TLB stores the recent translations of virtual memory to physical memory.

Effective access time (1)

Use the hit ratio and the relative access times to measure the performance.

$$\propto = hit\ ratio = \frac{\#\ TLB\ hits}{\#\ TLB\ hits + \#\ TLB\ misses}$$

$$\propto \rightarrow 1 \quad when\ \#\ TLB\ misses \rightarrow 0$$

$$\propto \rightarrow 0 \quad when\ \#\ TLB\ hits << \#\ TLB\ misses$$

Effective access time (2)

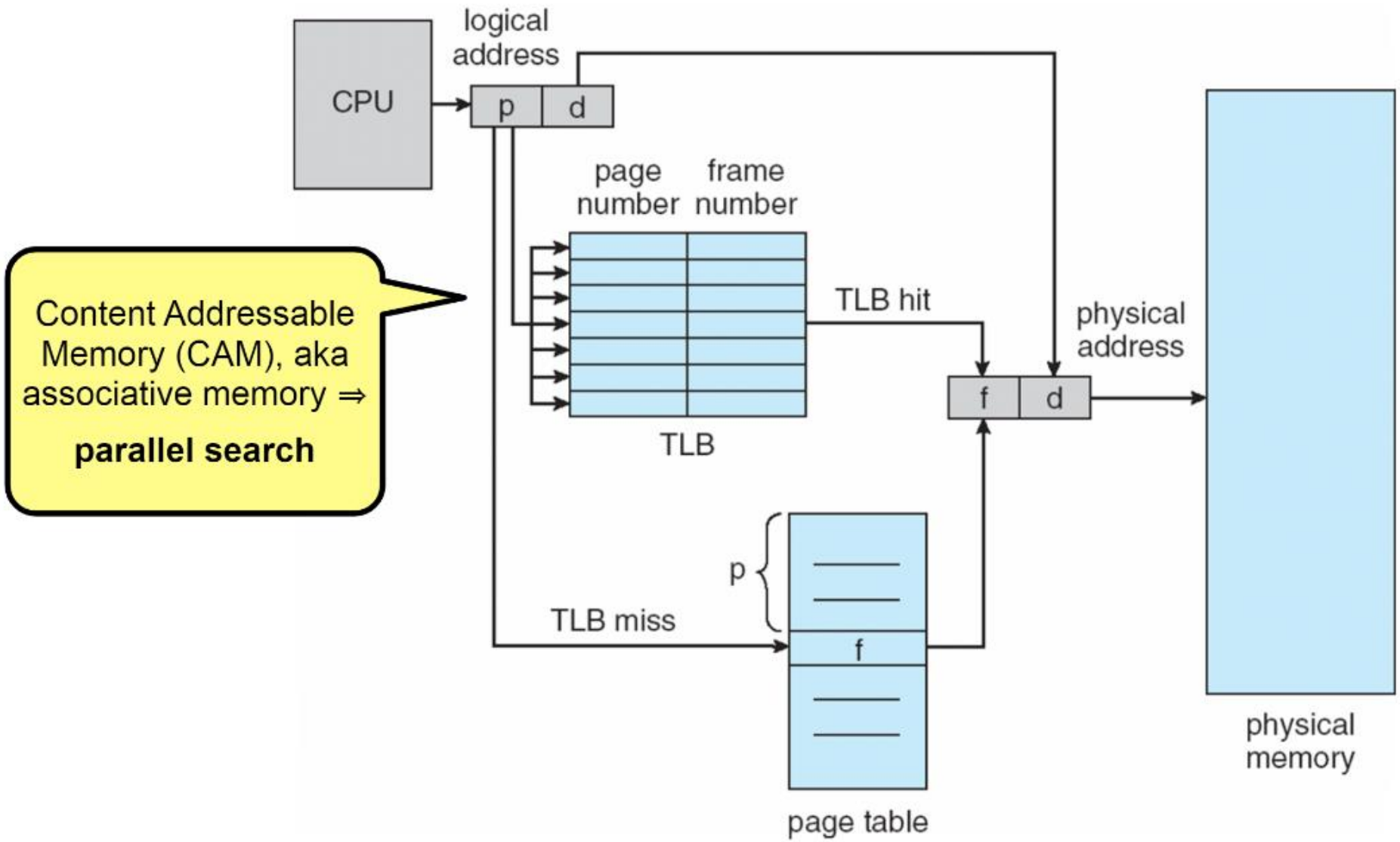
Access probabilities

$$P(\text{TLB hit}) = \alpha$$

$$P(\text{TLB miss}) = (1-\alpha)$$

Assume the following memory access times.

- Associative TLB lookup = ε time unit.
- Memory cycle time = 1 ms.

Paging hardware with TLB

A **translation lookaside buffer** (TLB) is a **cache** that memory management **hardware** uses to improve virtual address translation speed.