

Memory management

Module 5

Lecture

Operating systems 2019

1DT044, 1DT096 and 1DT003

Human user



Human user



Human user



Process A



Process B



Process B



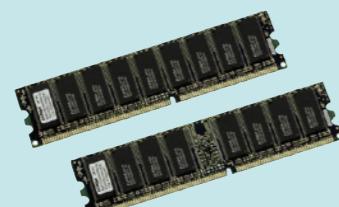
• • • • •

Process Z

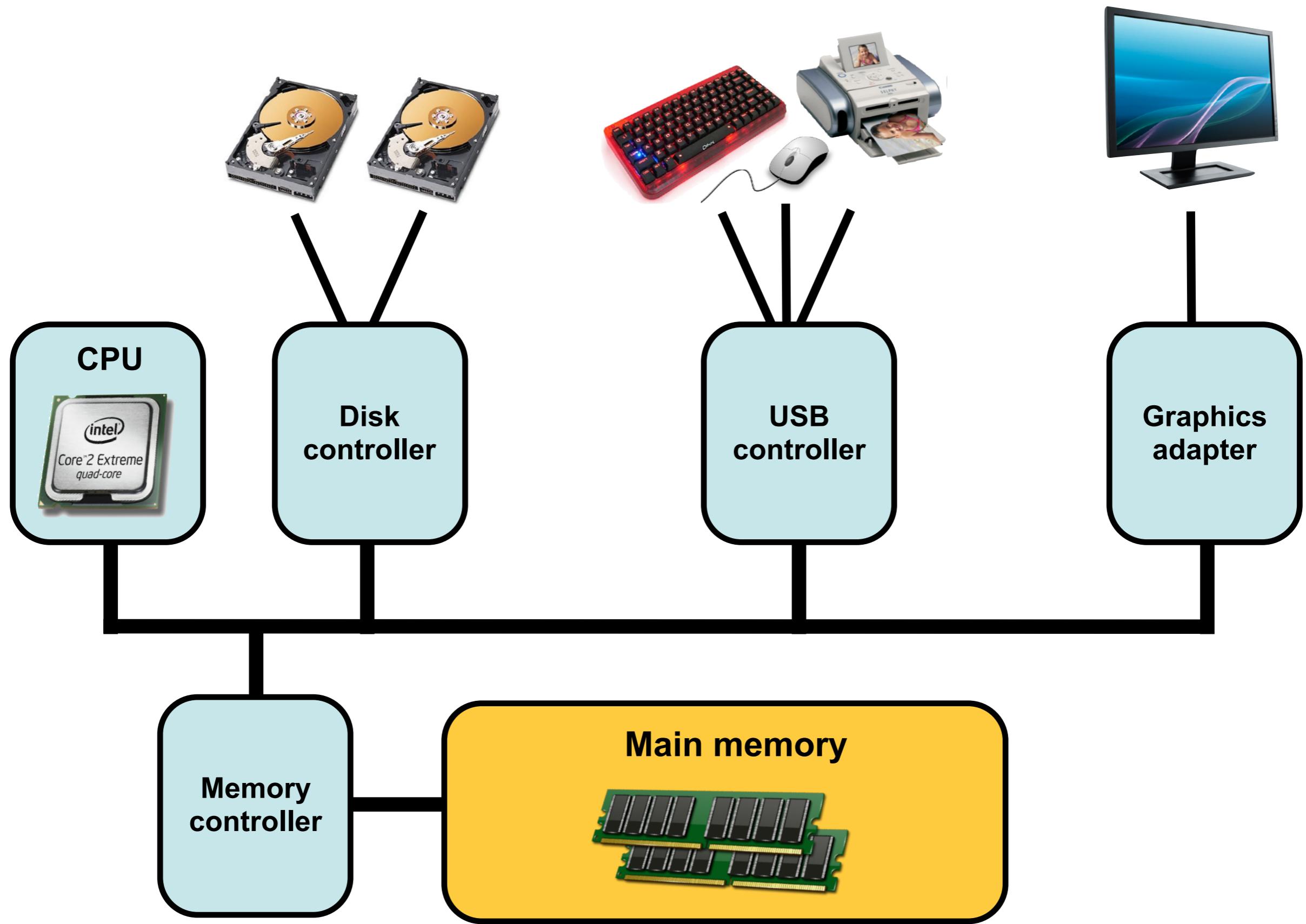


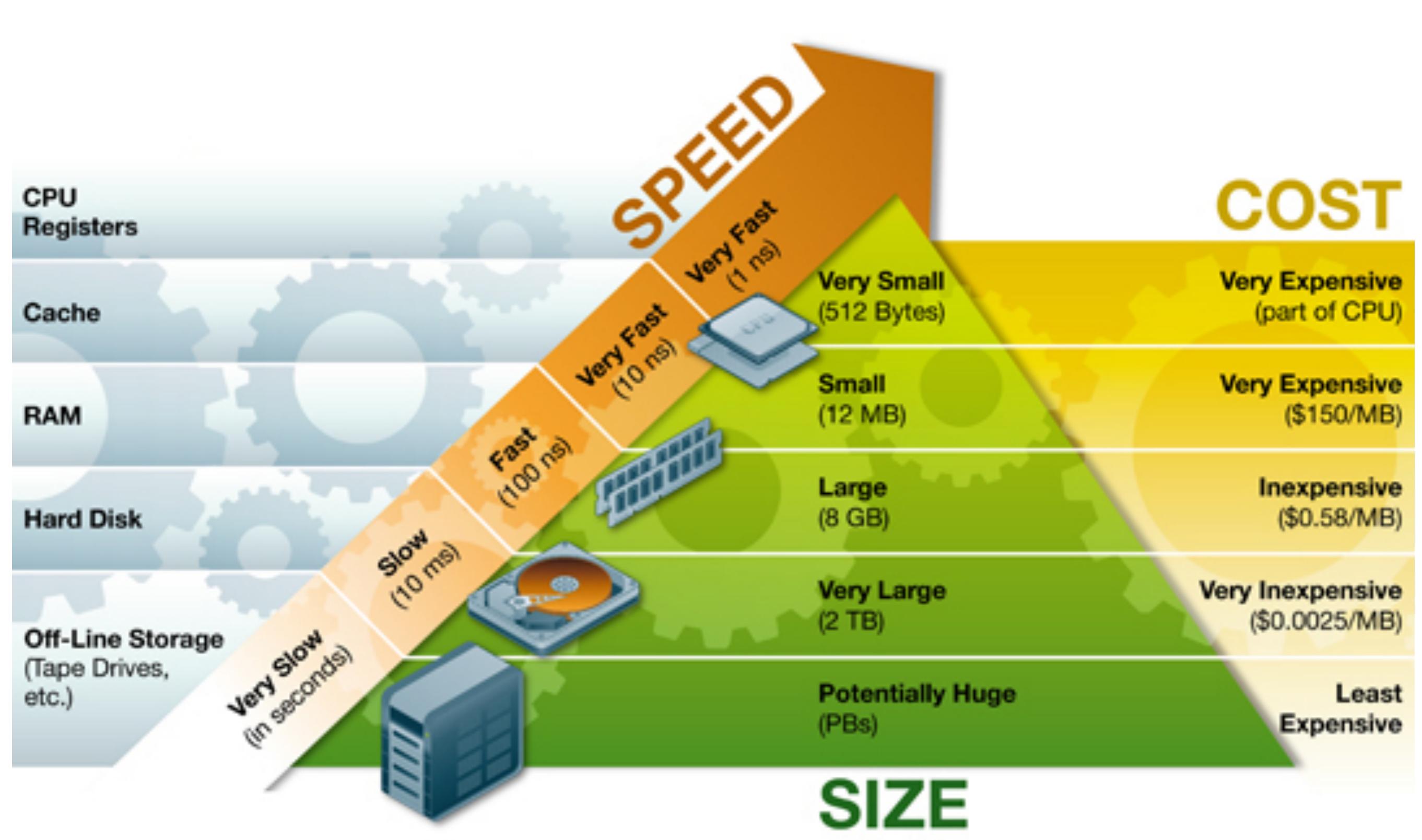
The operating systems **controls the hardware** and **coordinates** its use among the various application programs for the various user. An operating system provides an **environment for the execution of programs**.

Computer hardware



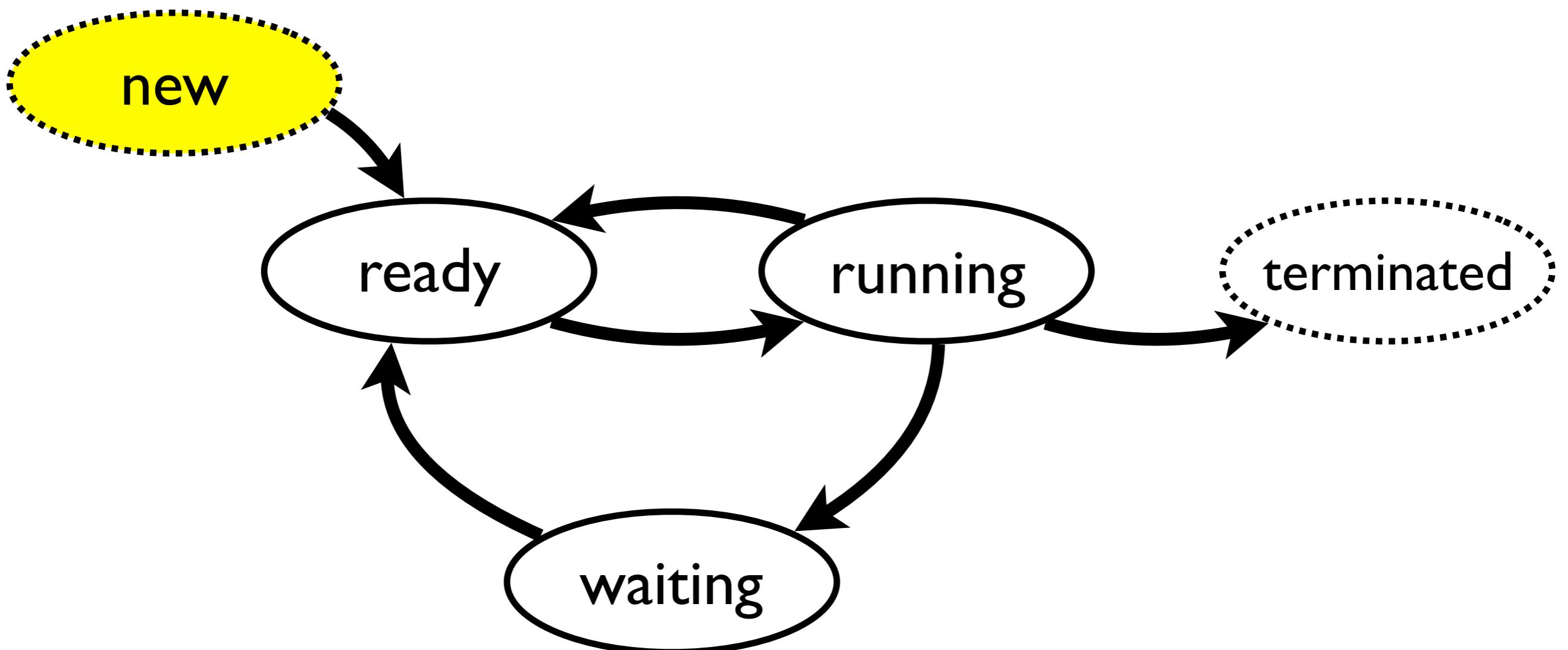
A typical computer system





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)

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

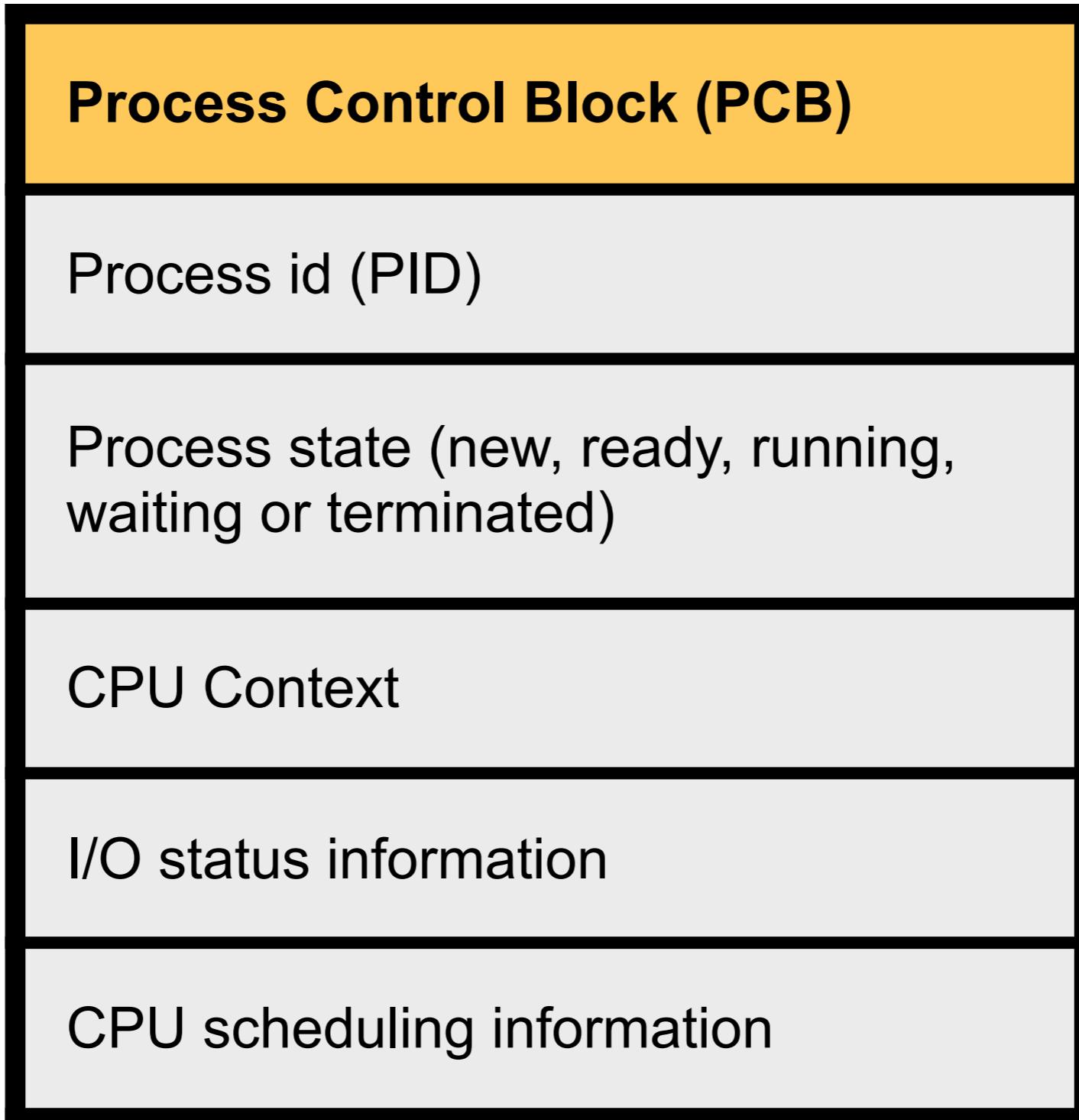
2018-01-21

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

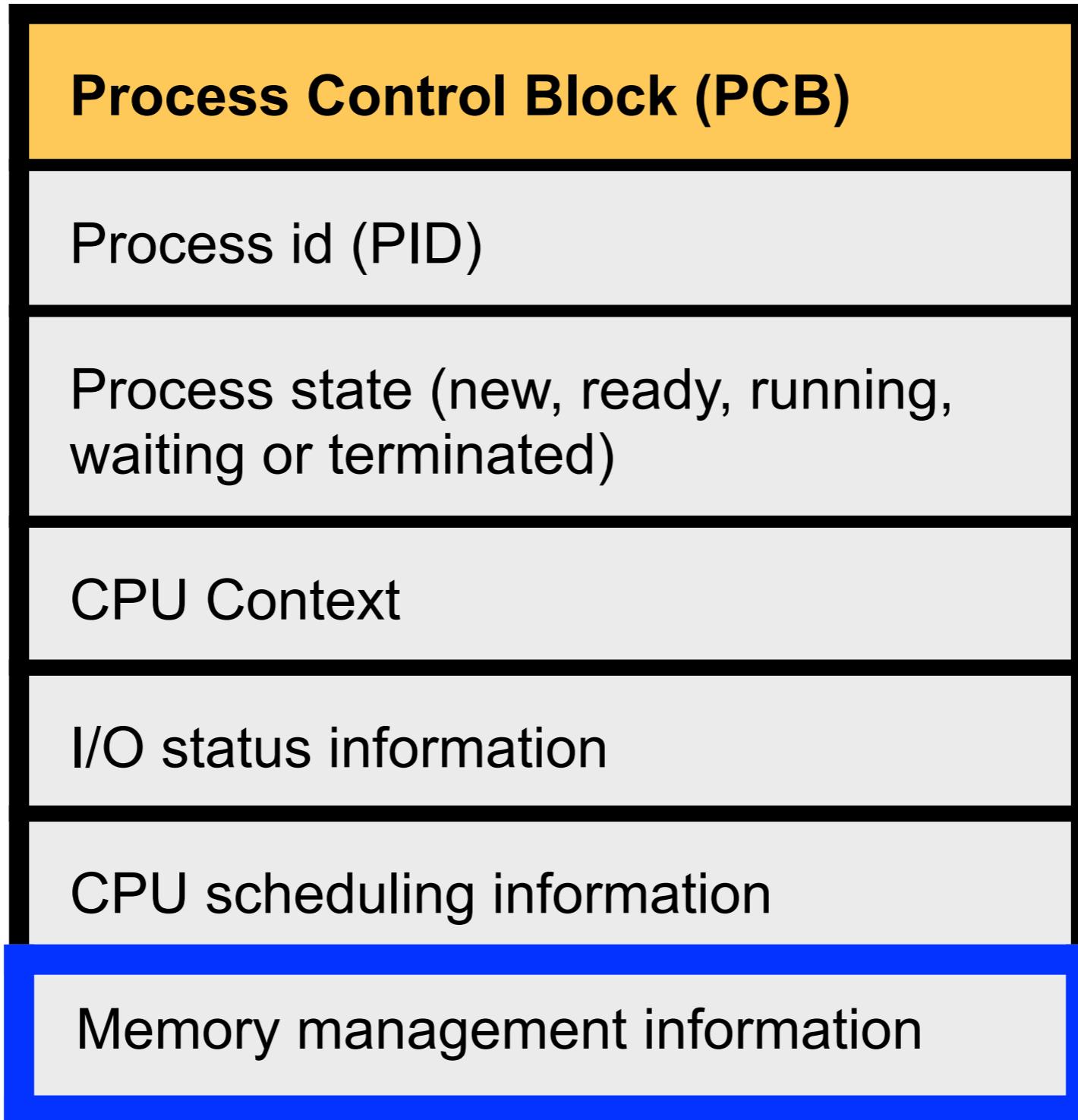
Process control block (PCB)

Must allocate
memory for a
new PCB

Example of information stored in the PCB



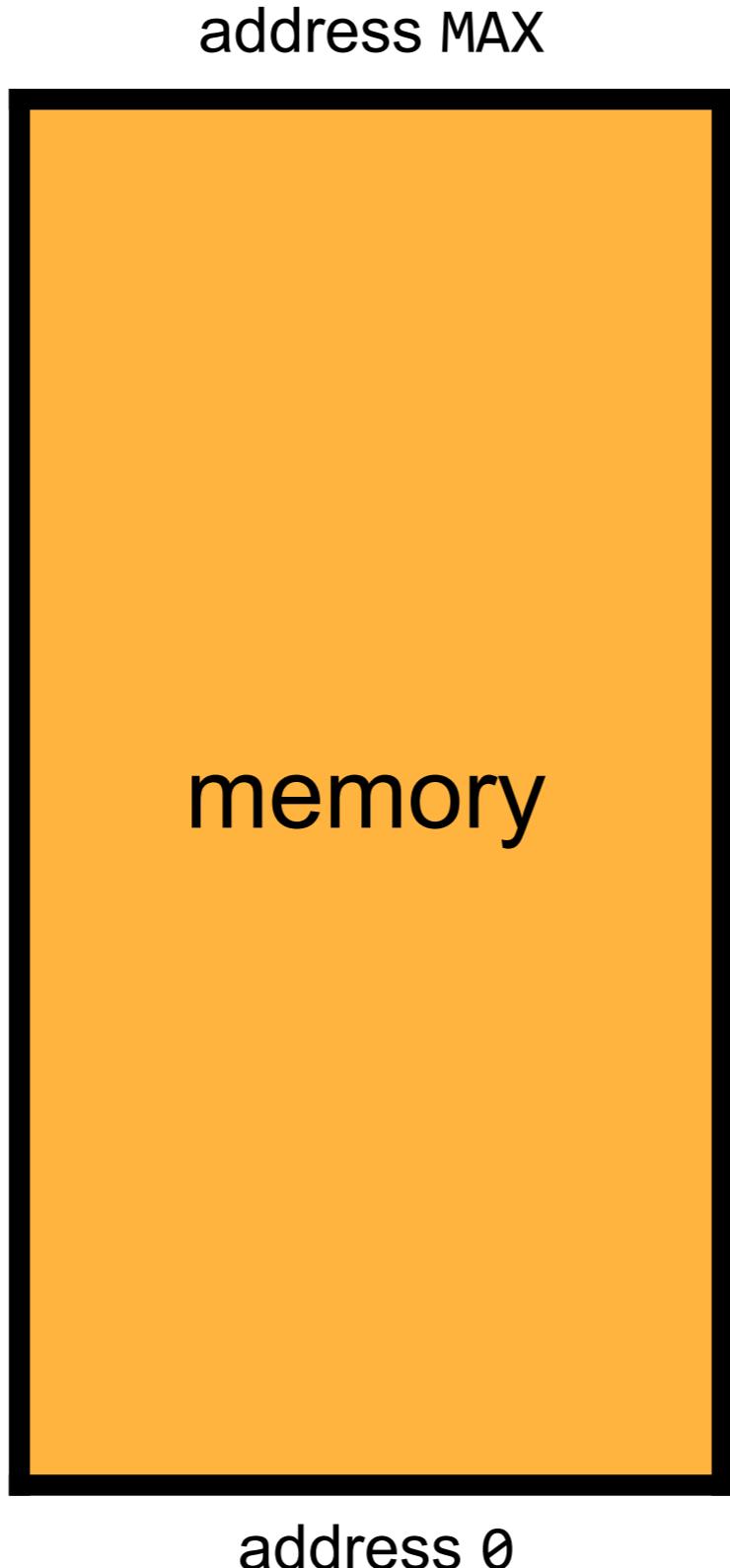
Example of information stored in the PCB



Process memory space

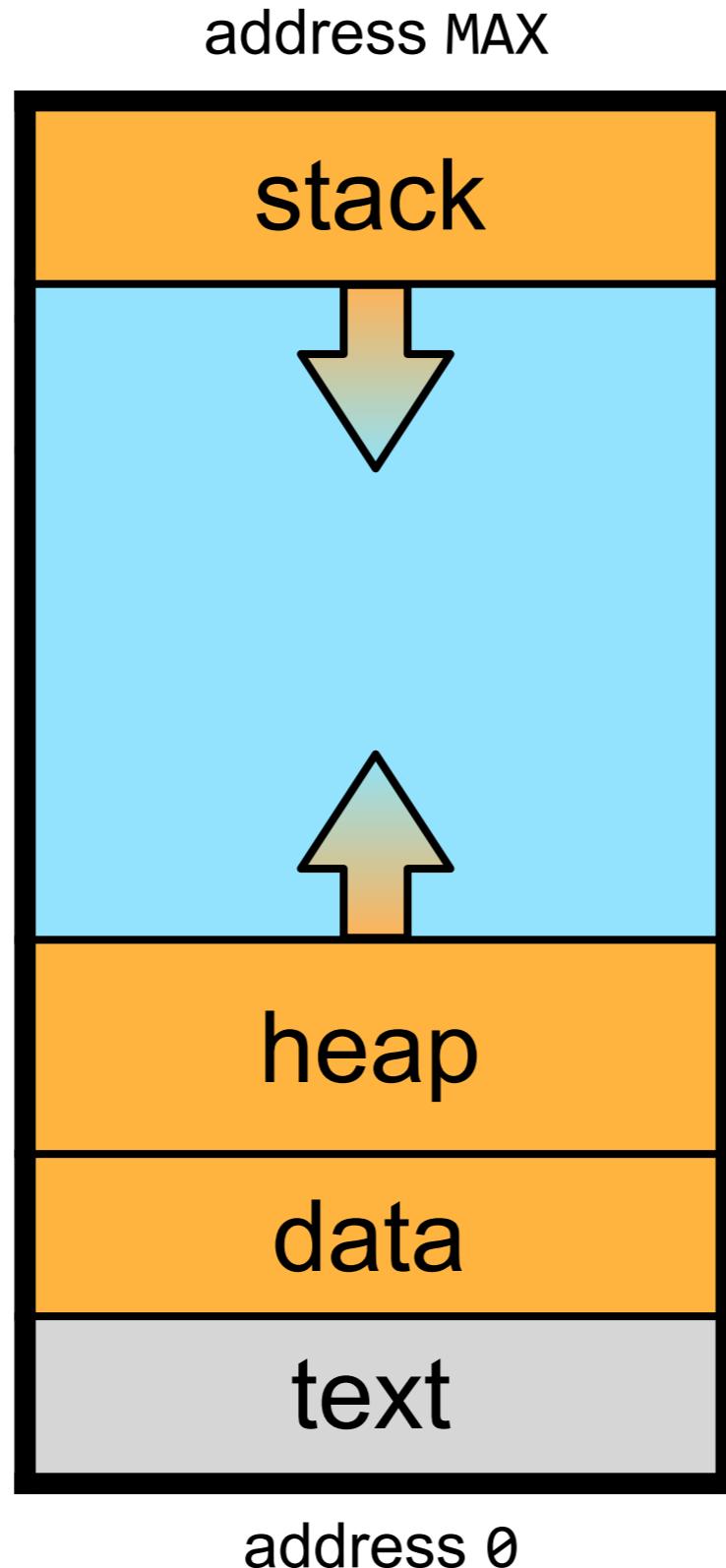
In addition to a PCB, the operating system must allocate a new **memory space** for each new process. This memory space is often referred to as the process **memory image**.

Static memory allocation



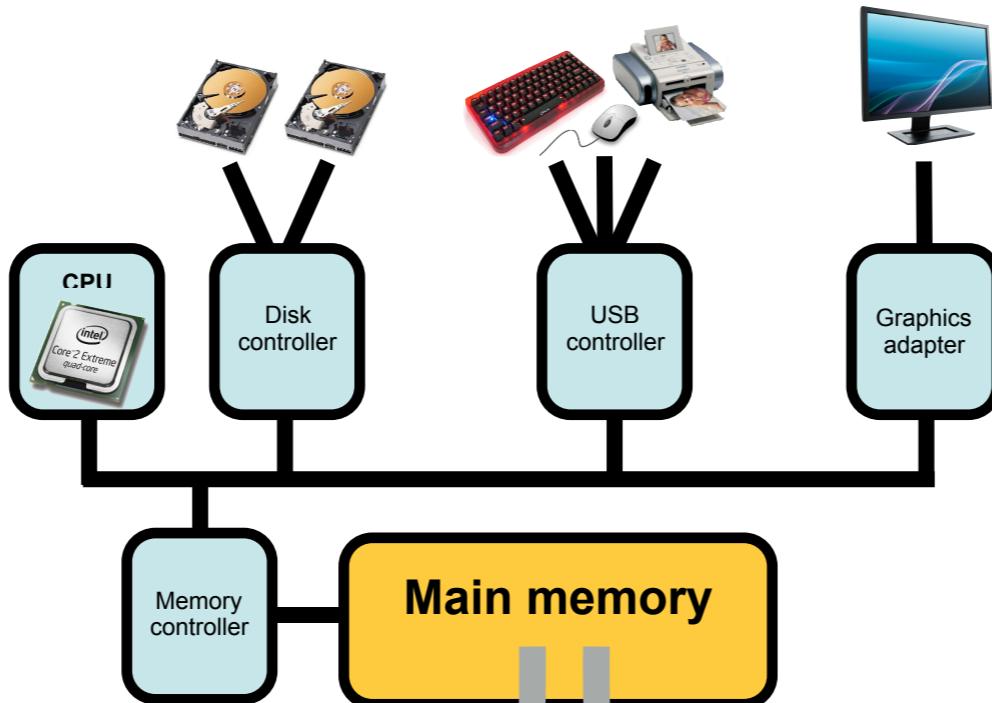
The operating system must allocates a blob of memory for the new process memory image.

Static memory allocation



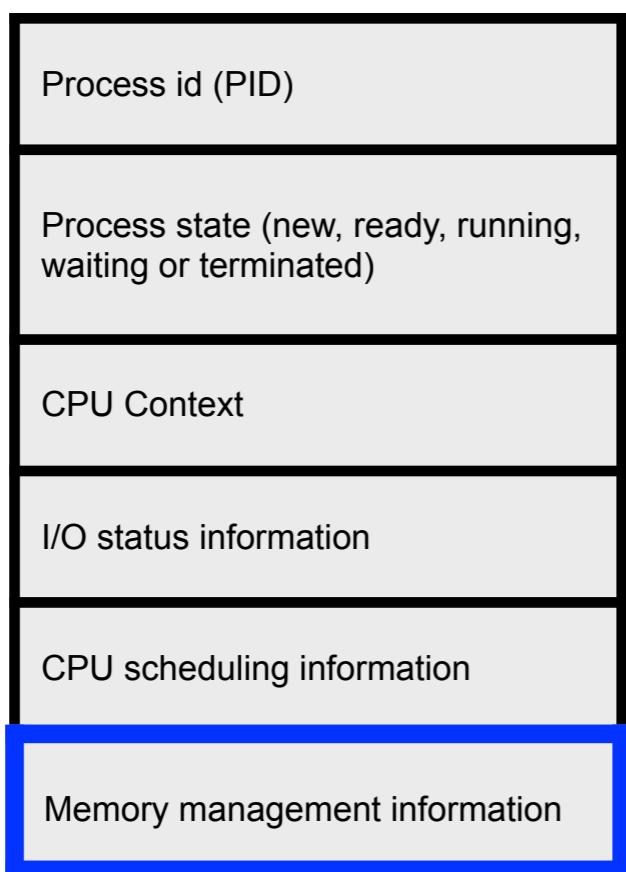
The allocated memory image is divided into the following segments:

- ▶ text
- ▶ data
- ▶ heap
- ▶ stack

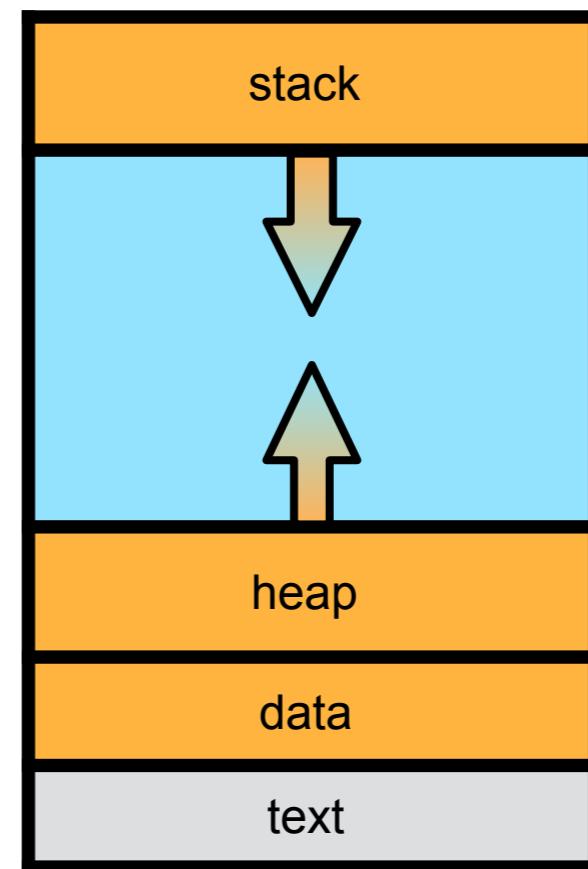


For each process, a PCB and a process image must be allocated in memory (RAM).

Process control block

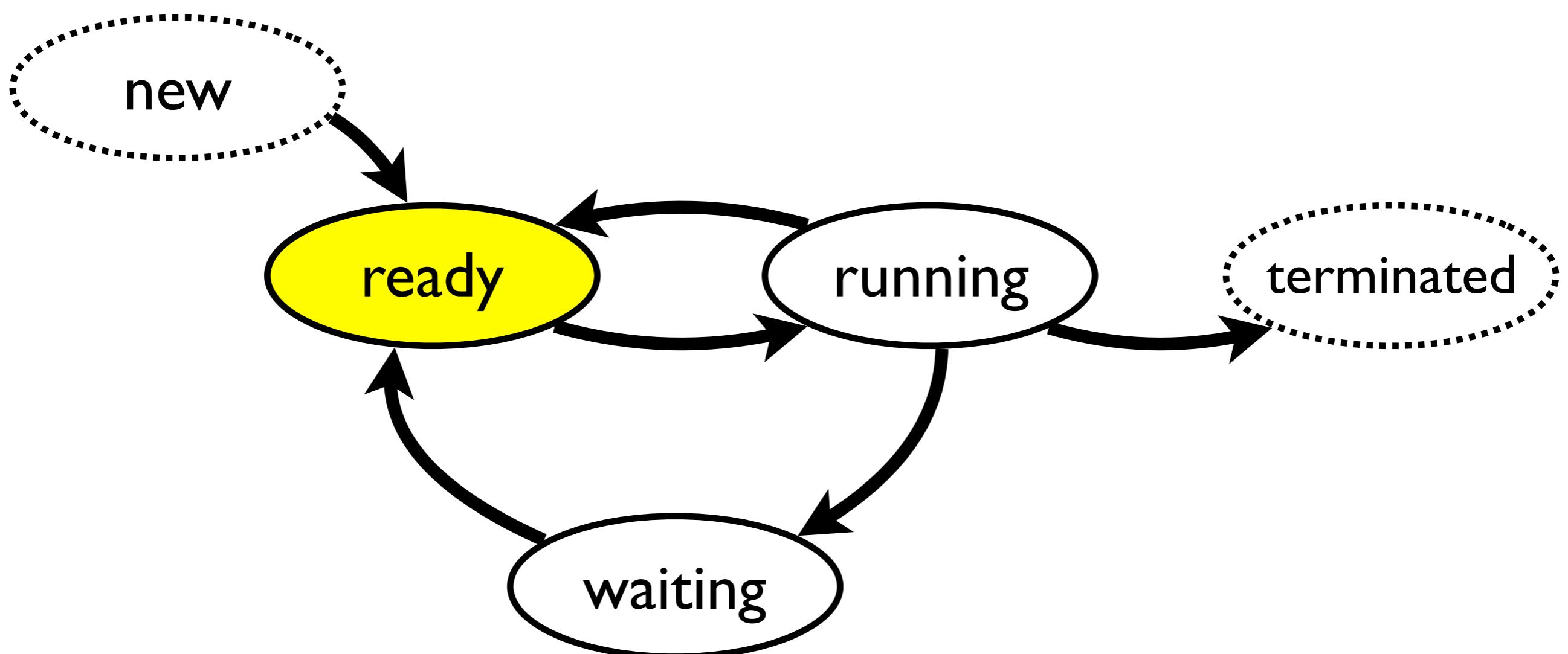


Process memory image



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.

Human user



Human user



Human user



Process A



Process B



Process B



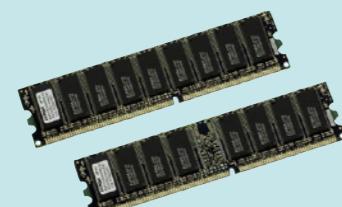
• • • • •

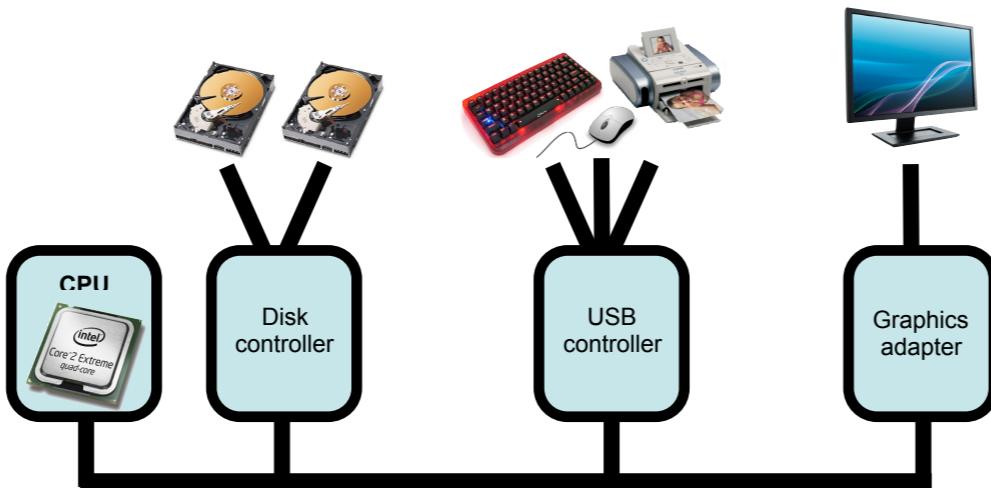
Process Z



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**.

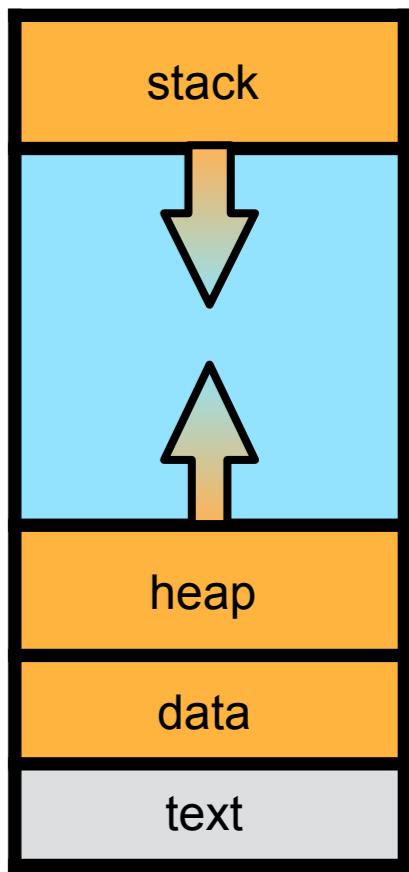
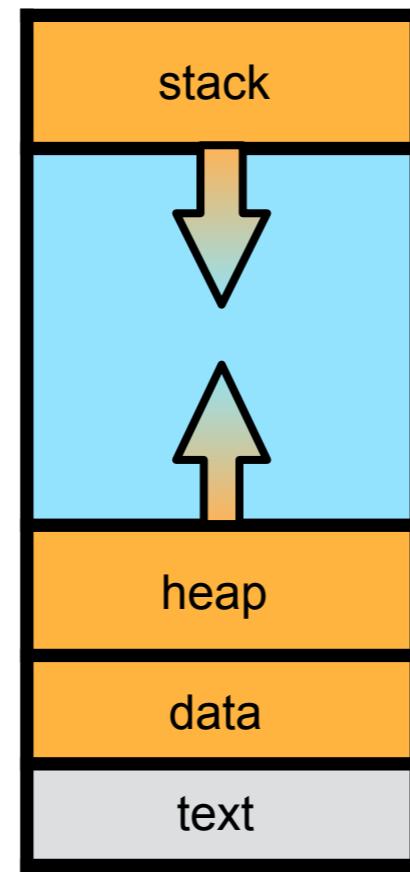
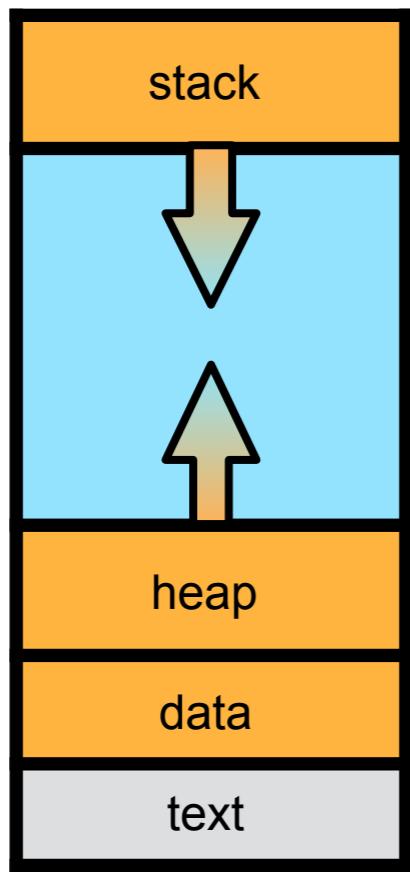
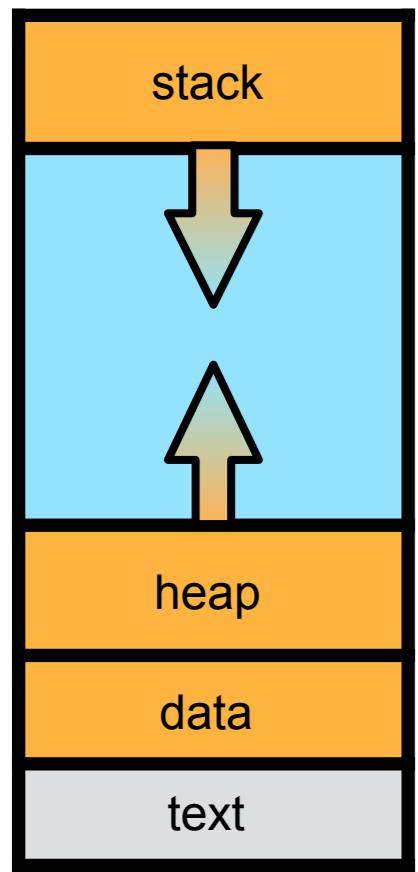
Computer hardware





1

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



Human user



Non-interactive
process with no
human user

Human user



Human user



Process A



Process B



Process B



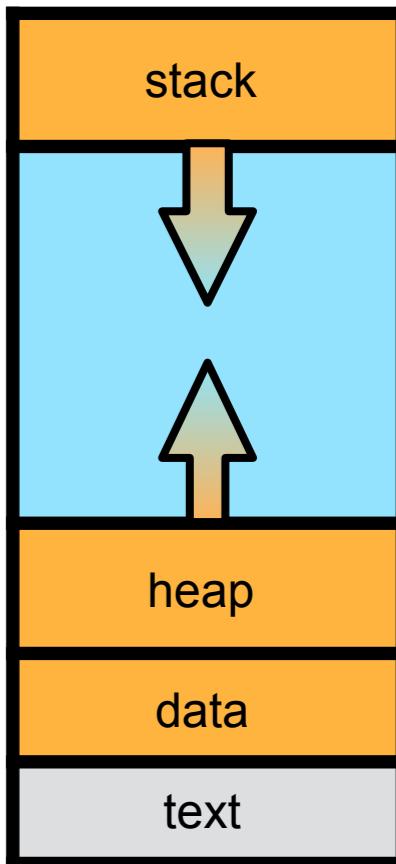
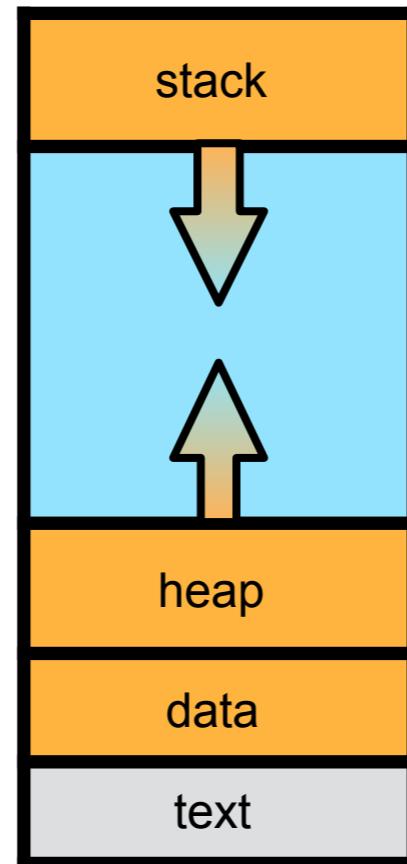
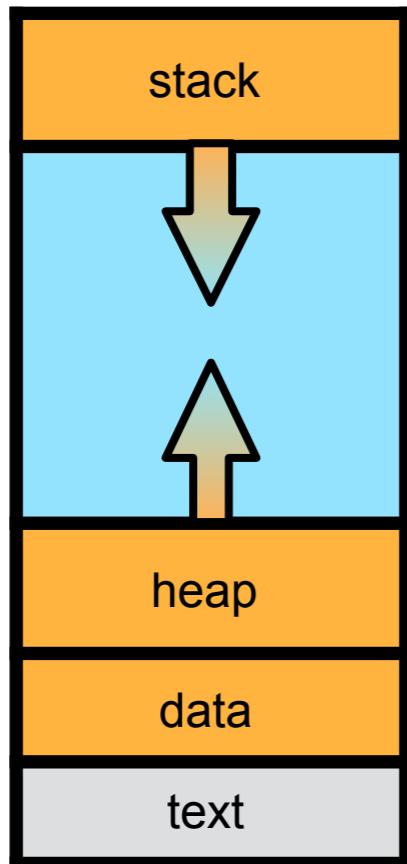
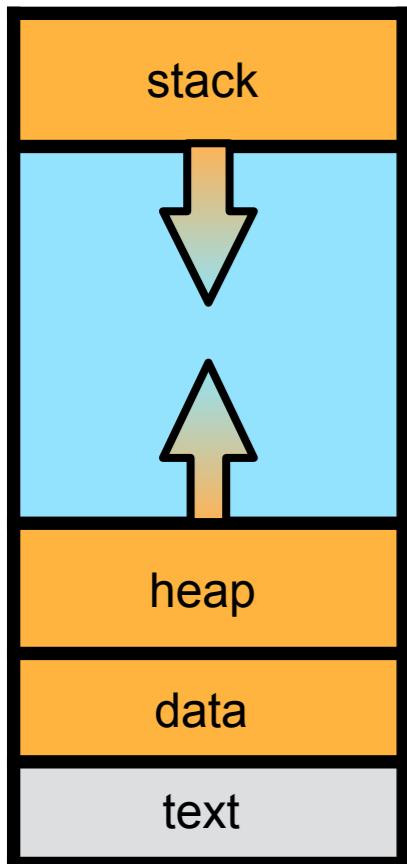
• • • • •

Process Z



2

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



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.

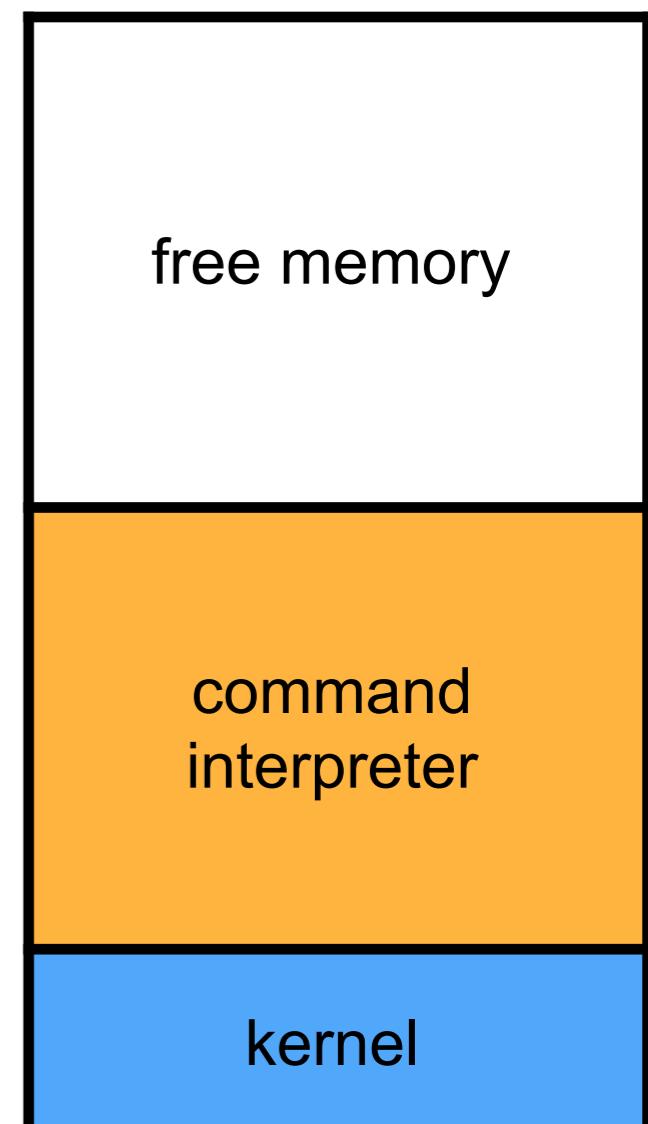
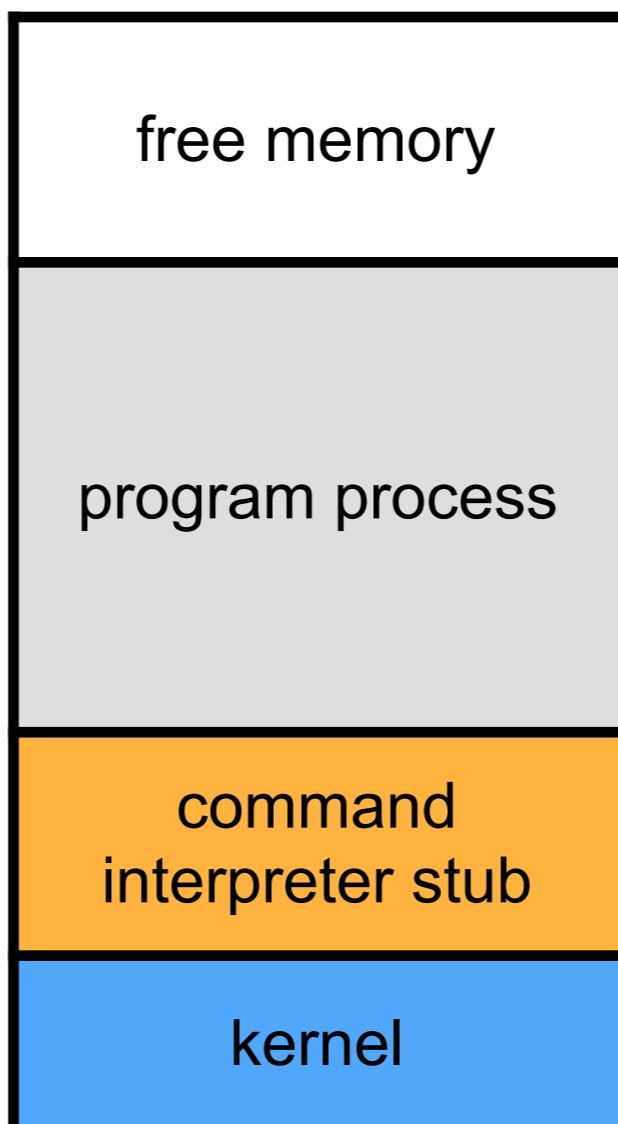
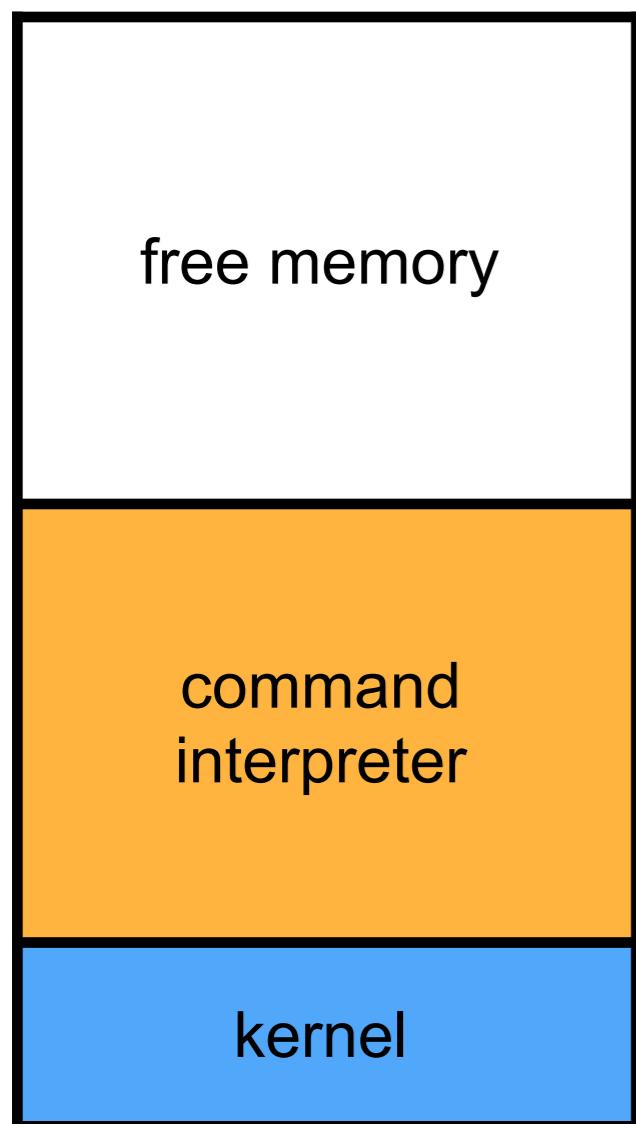
Single contiguous allocation

- ▶ 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.

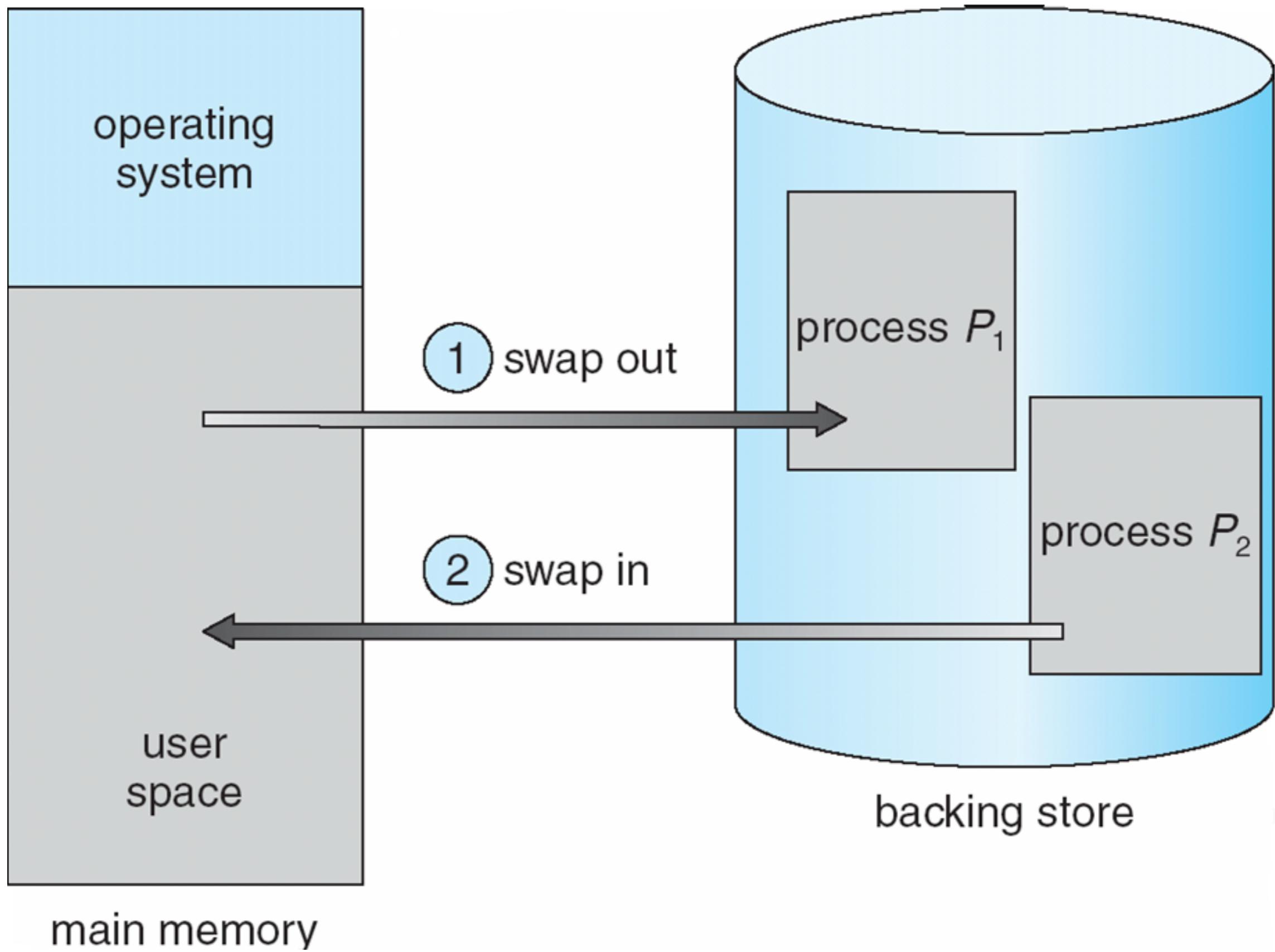
- ▶ MS-DOS is a single-tasking operating system. It has a command interpreter that is invoked when the computer is started.
- ▶ MS-DOS simply loads a program into memory, writing over most of the command interpreter to give the program as much memory as possible.
- ▶ When the program terminates, the 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.



Memory resident

Certain programs, can be marked as being memory resident, which means that the operating system is not permitted to swap them out to a storage device; they will always remain in memory.

The programs and data used most frequently are the ones that should be memory resident.

This includes central portions of the operating system and special programs, such as calendars and calculators, that you want to be able to access immediately.

The **resident set size** is the portion of a process's memory that is held in RAM. The rest of the memory exists in swap or the filesystem (never loaded or previously unloaded parts of the executable).

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.

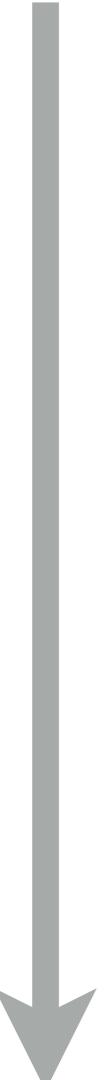
Main memory usually divided into two partitions: one for the operating system and one for all the processes in the system.

low
address

Resident **operating system**,
usually held in low memory with
interrupt vector.

User **processes** held in high
memory.

high
address



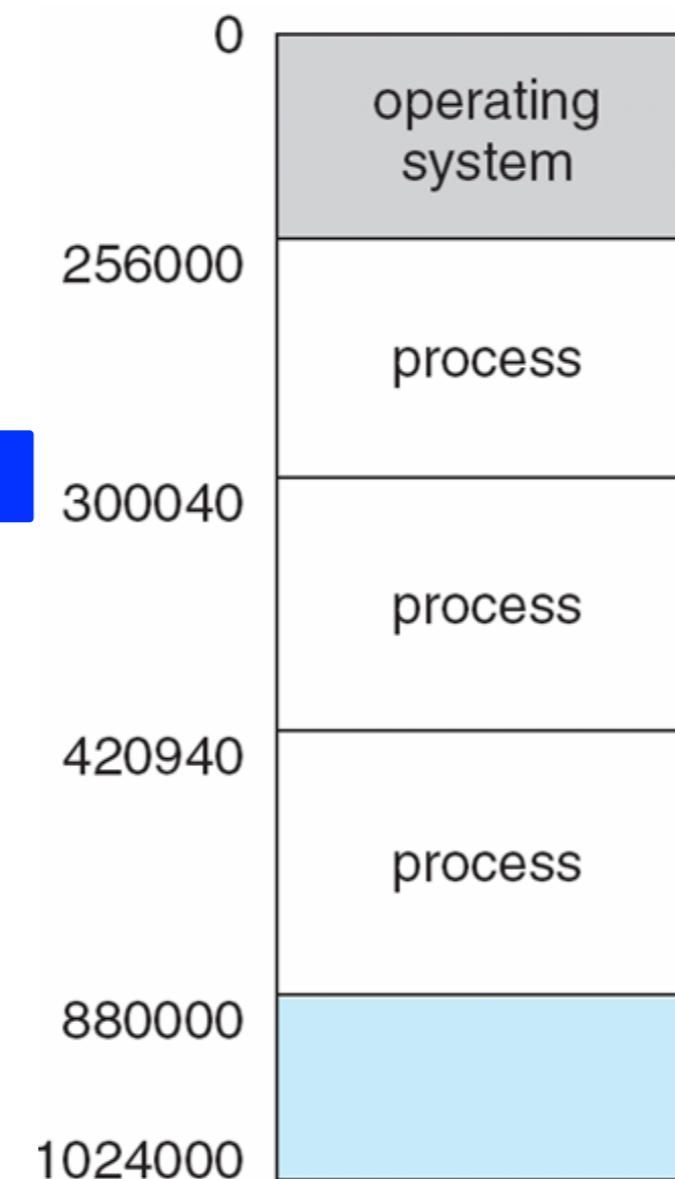
Logical address space

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

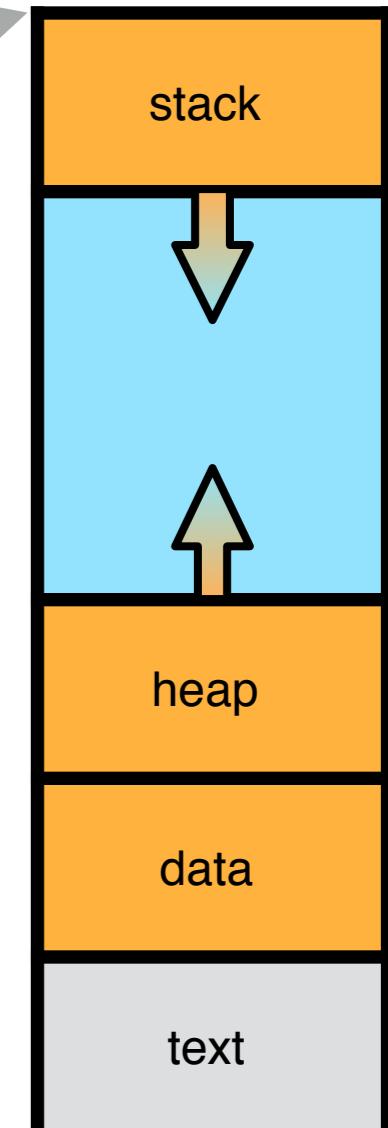
Process control block

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

Main memory



Process memory image



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

low
addresses

Main memory

Physical memory
addresses



CPU



high
addresses

Logical and physical memory addresses

Currently executing process

Logical memory addresses

Address

0



Address
Max



CPU

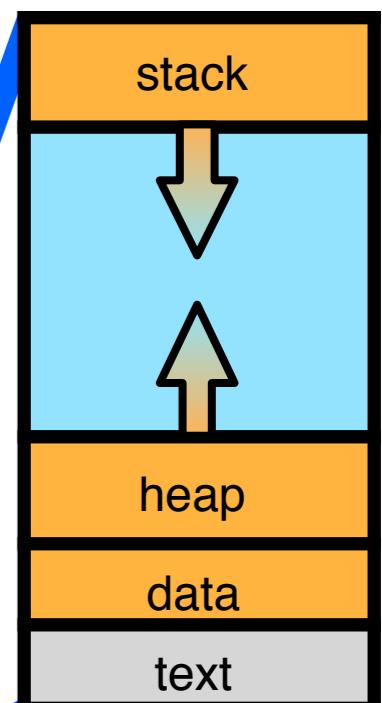
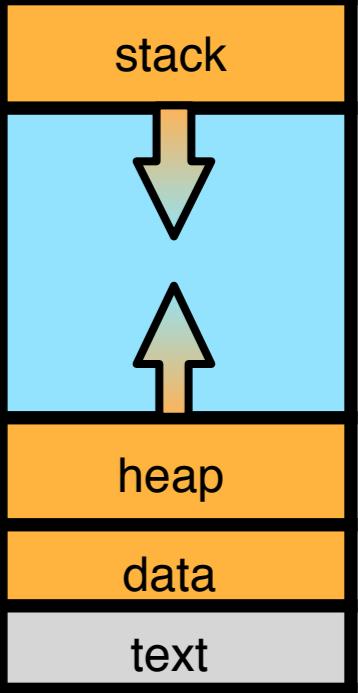
Main memory

Physical memory
addresses

P1

P2

P3

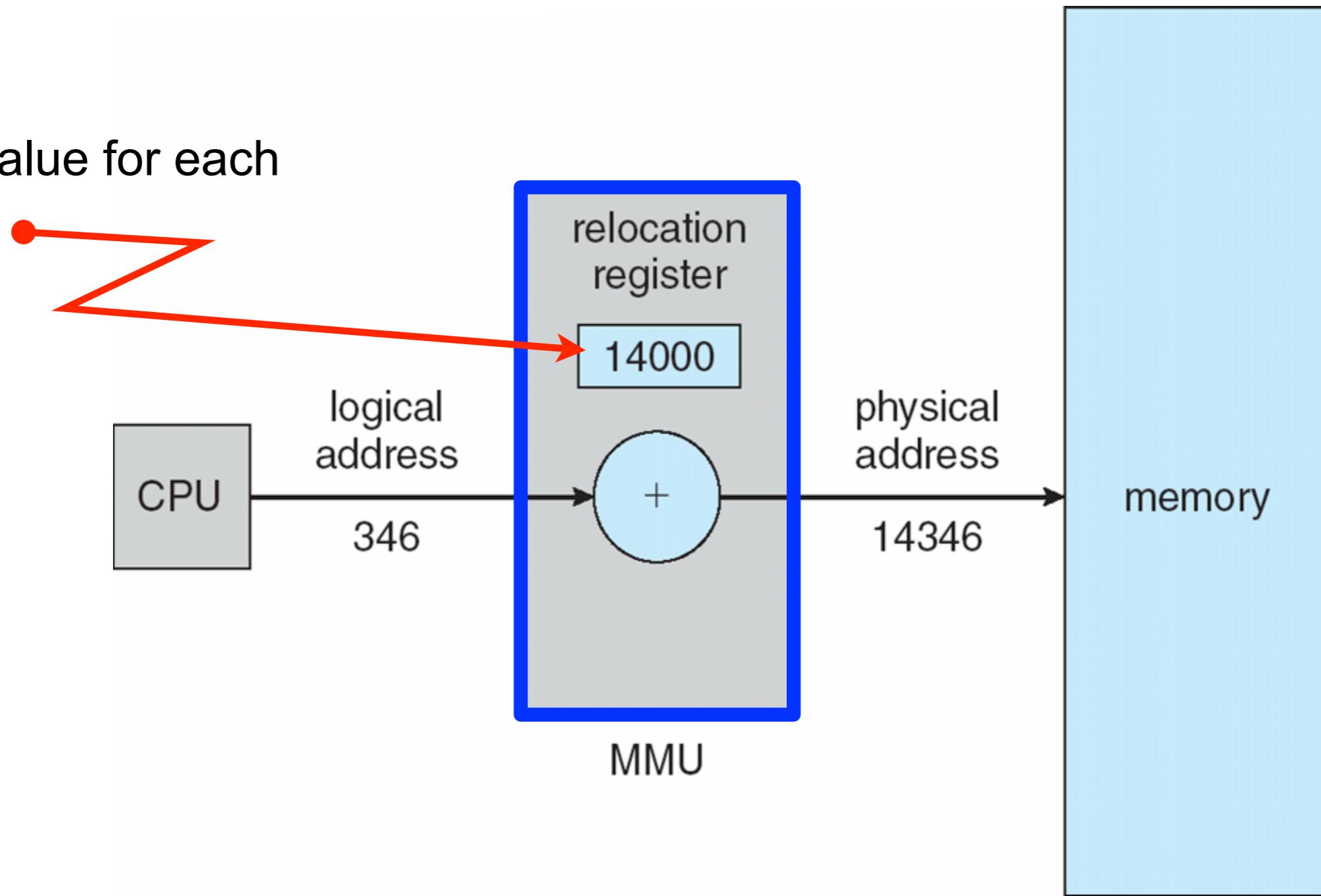


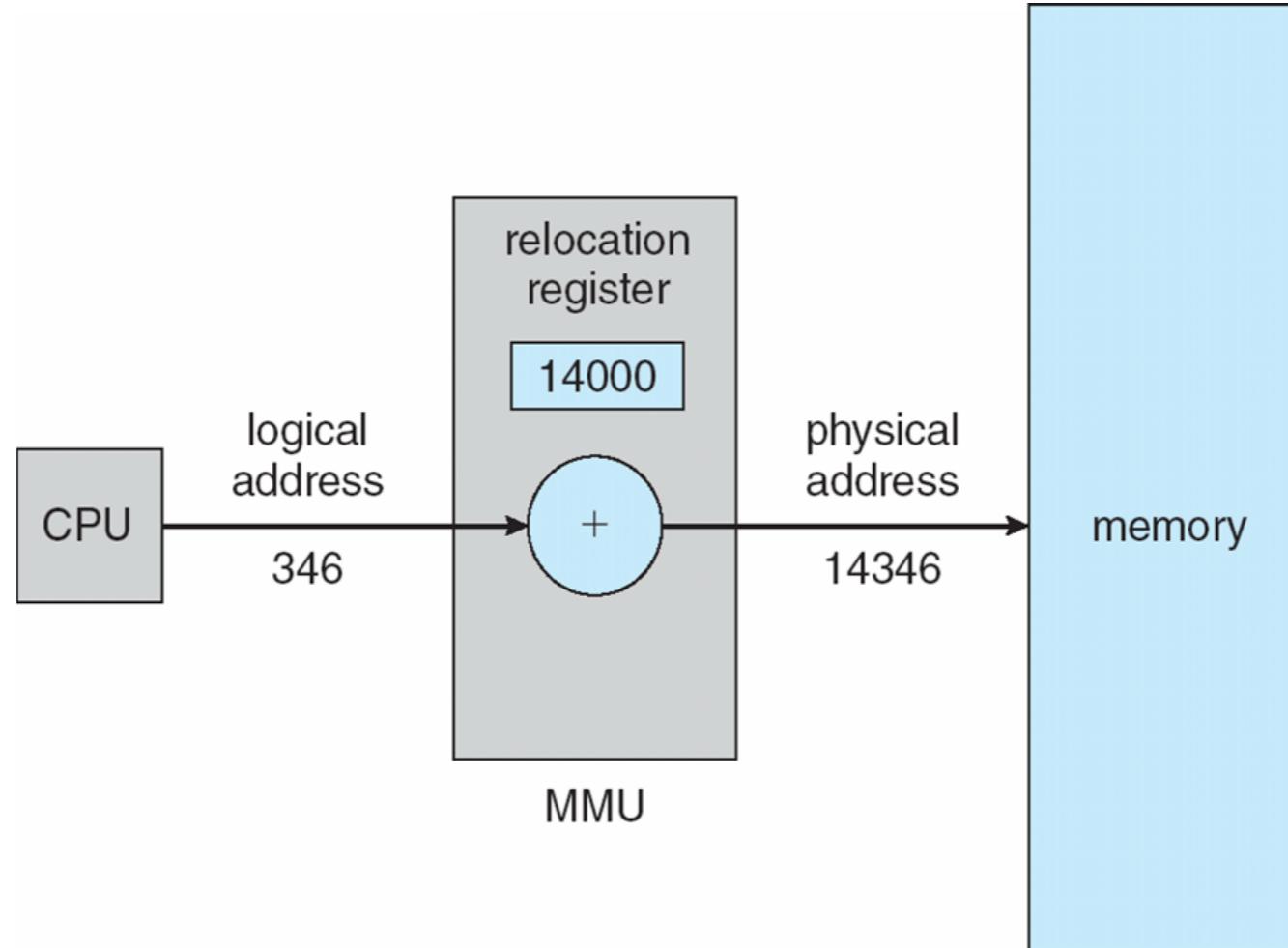
Memory management unit (MMU)

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

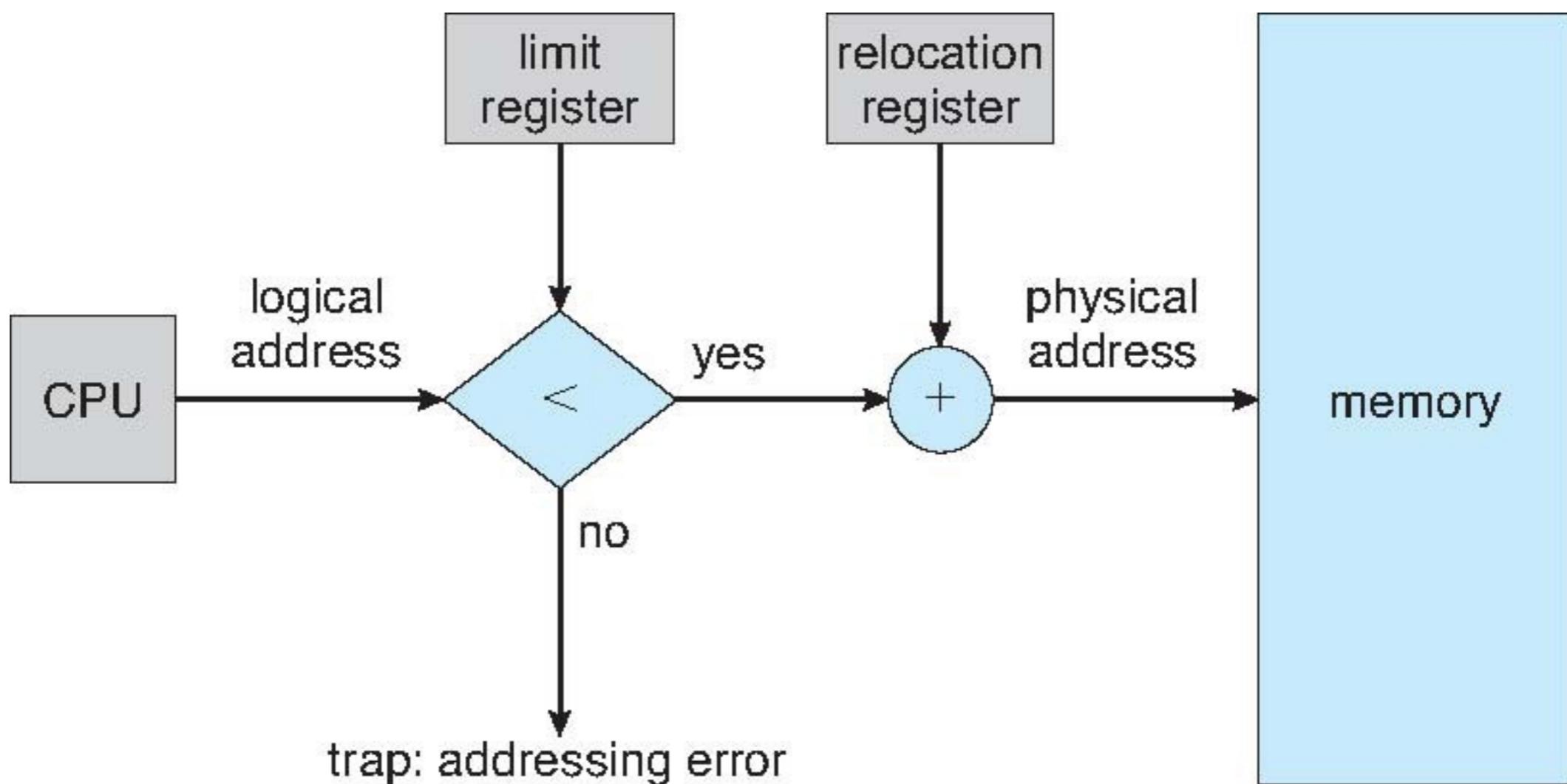
Separate value for each process.



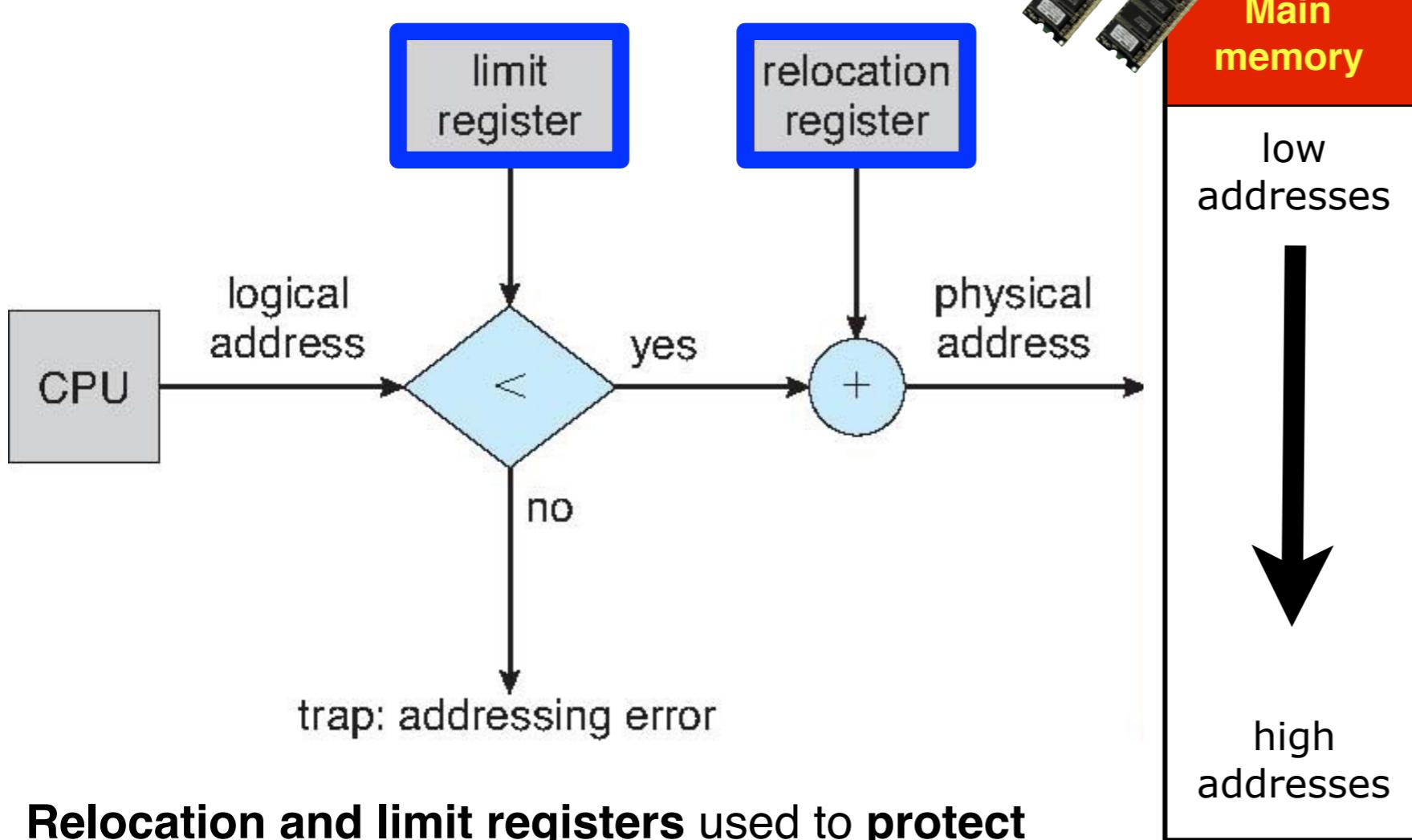


- ▶ Dynamic relocation using a **relocation register**.
- ▶ MMU maps **logical addresses to physical addresses**.
- ▶ The value in the relocation register is added to every address generated by a user process at the time it is sent to memory.
- ▶ The user program deals with logical addresses; it never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

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



Contiguous Allocation



Relocation and limit registers used to protect

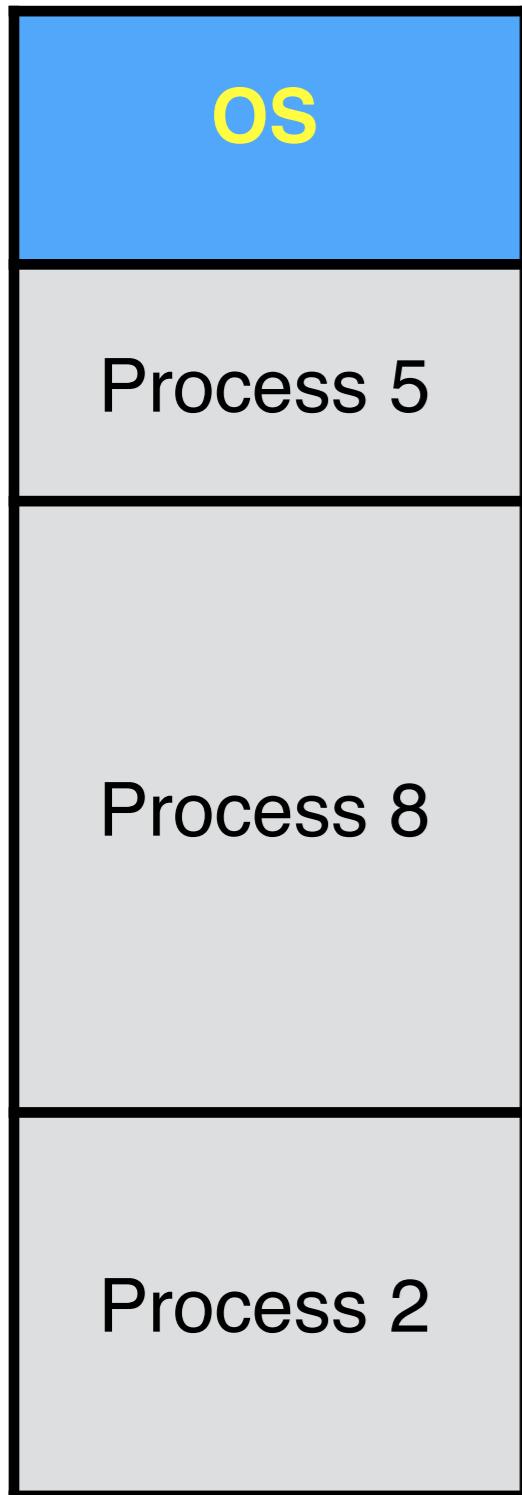
Main memory usually divided into two partitions:

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

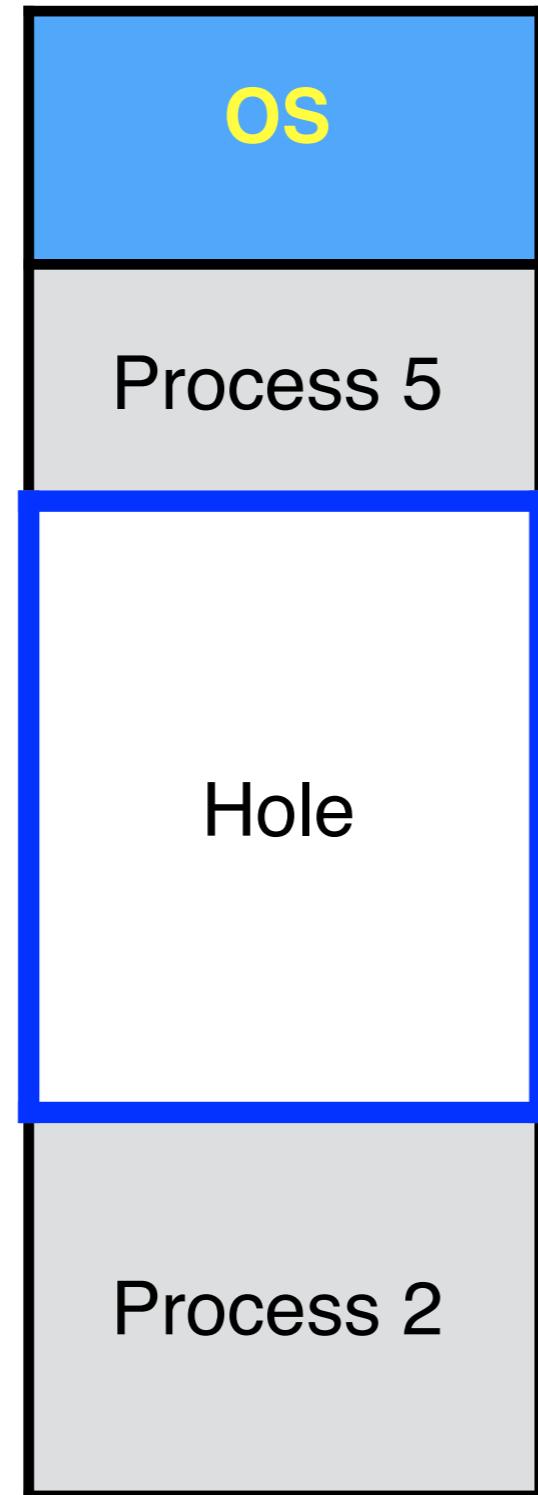
- ★ user processes from each other
- ★ operating-system code and data from user processes
 - ▶ Relocation register contains value of smallest physical address
 - ▶ Limit register contains range of logical addresses – each logical address must be less than the limit register
 - ▶ MMU maps logical address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

**Dynamic
memory
management**

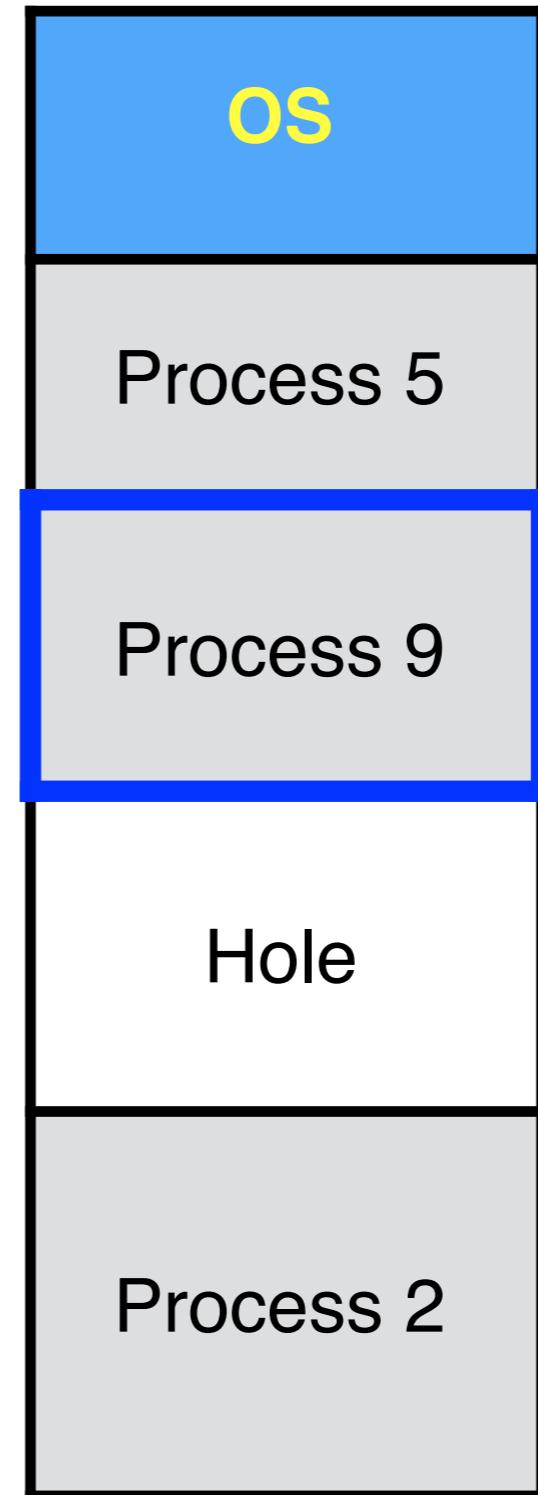
Three process occupies all available memory.



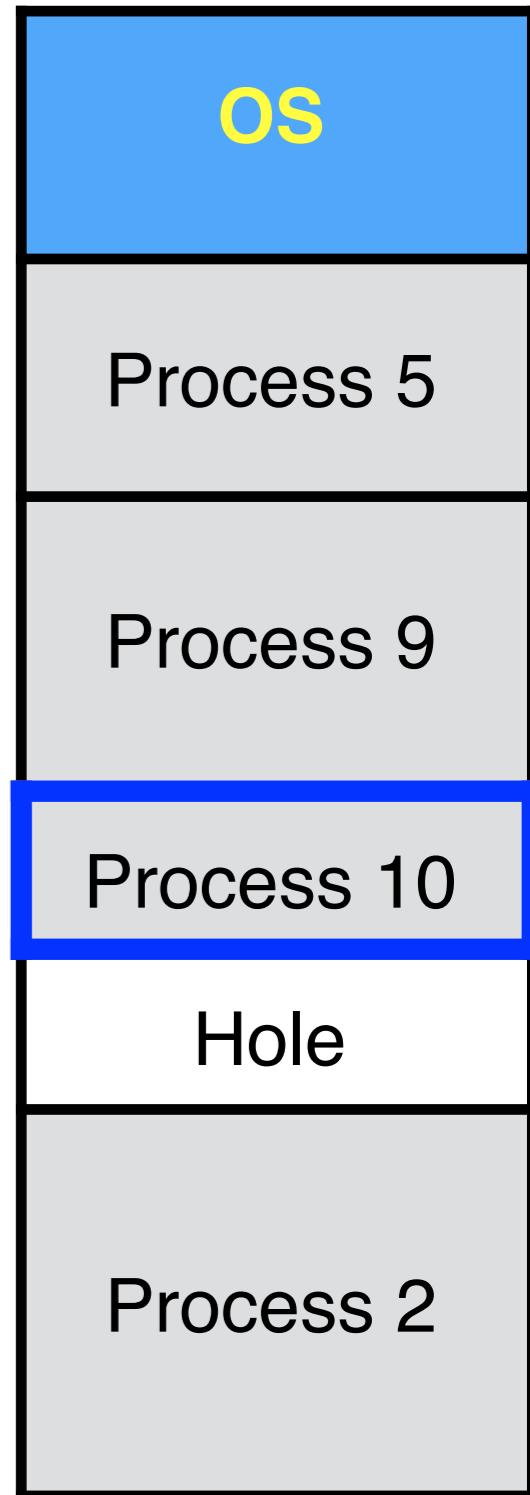
Process 8 terminates and leaves a free hole.

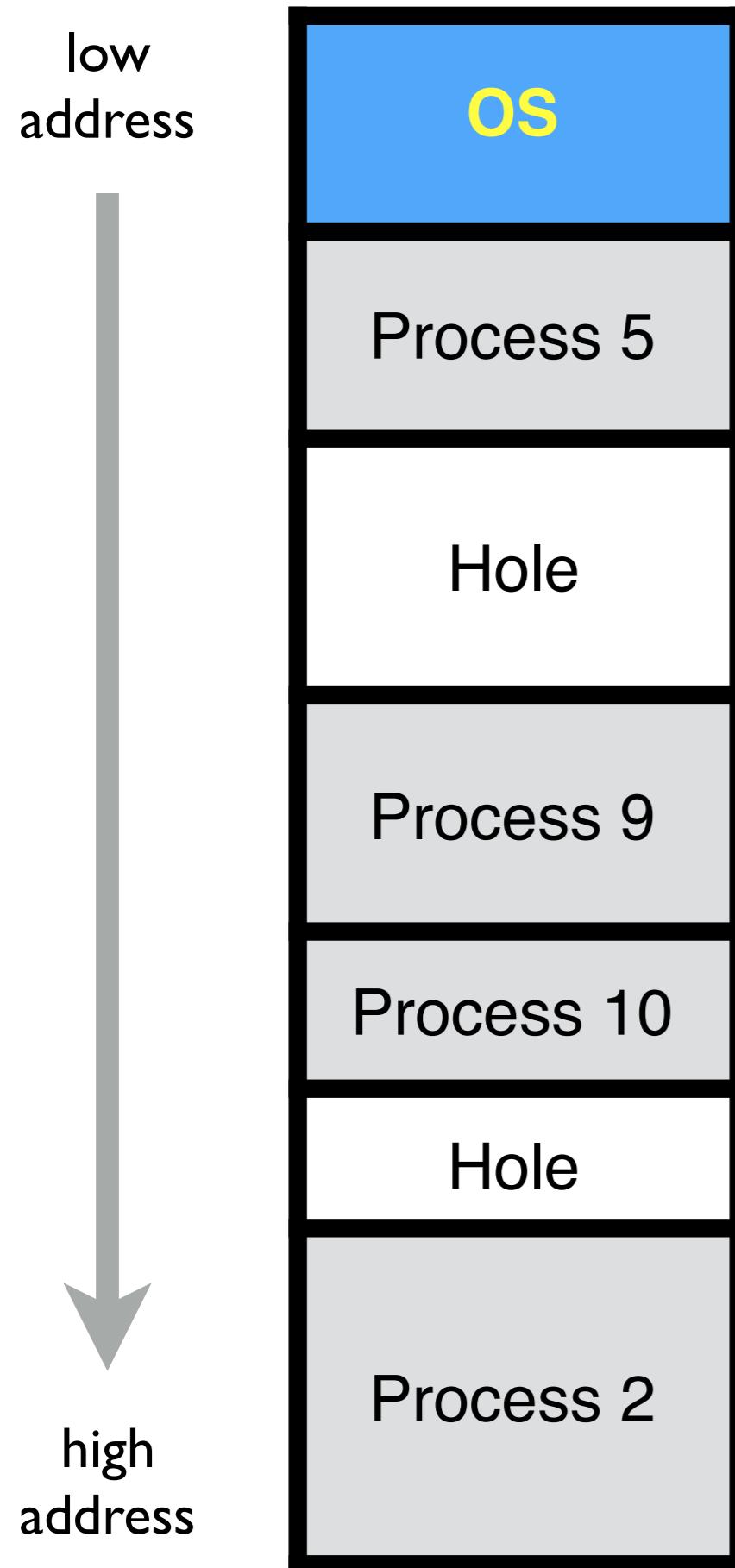


Process 9 allocates part of the hole left by process 8 and the hole of free memory shrinks.



The hole of free memory gets smaller and smaller





Hole – block of available free memory.

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

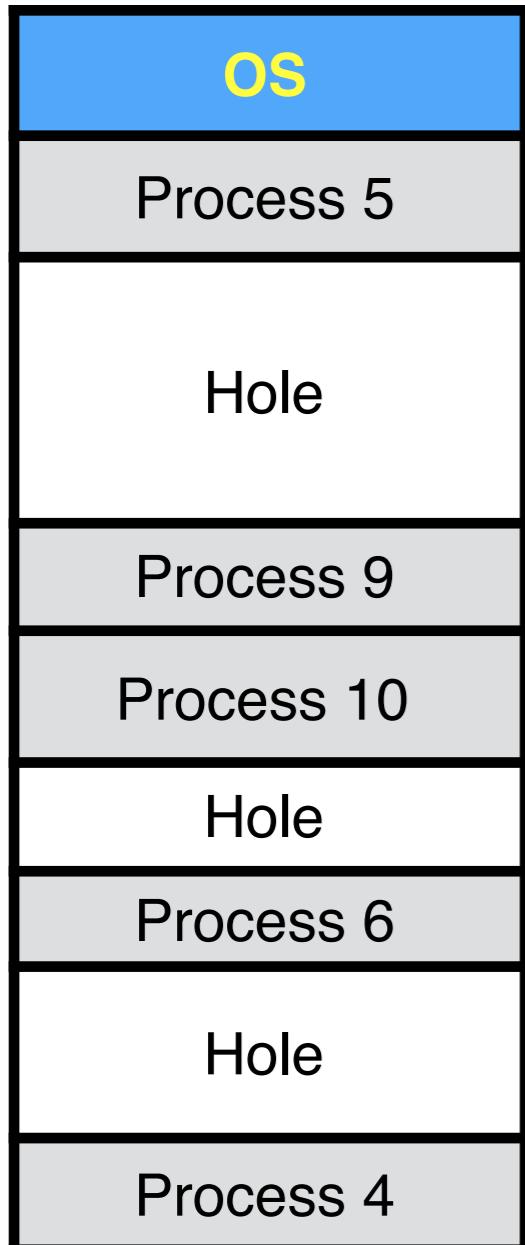
When a process arrives, it is **allocated memory from a hole large enough** to accommodate it.

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**?

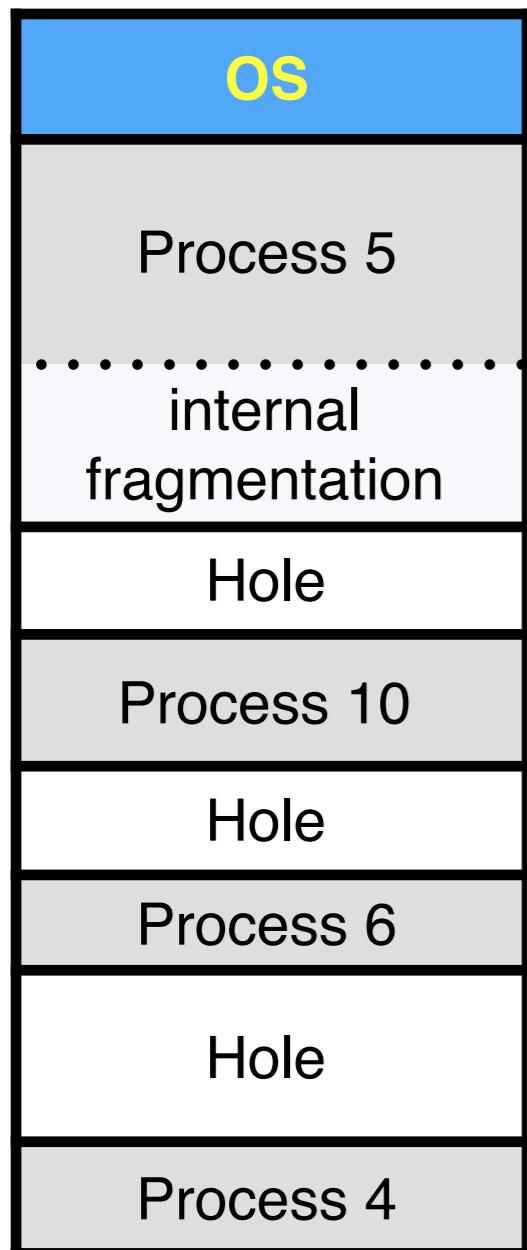


- ▶ **First-fit:** Allocate the first hole that is big enough
- ▶ **Best-fit:** Allocate the smallest hole that is big enough; must search entire list, unless ordered by size
 - ▶ Produces the smallest leftover hole
- ▶ **Worst-fit:** Allocate the largest hole; must also search entire list
 - ▶ Produces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

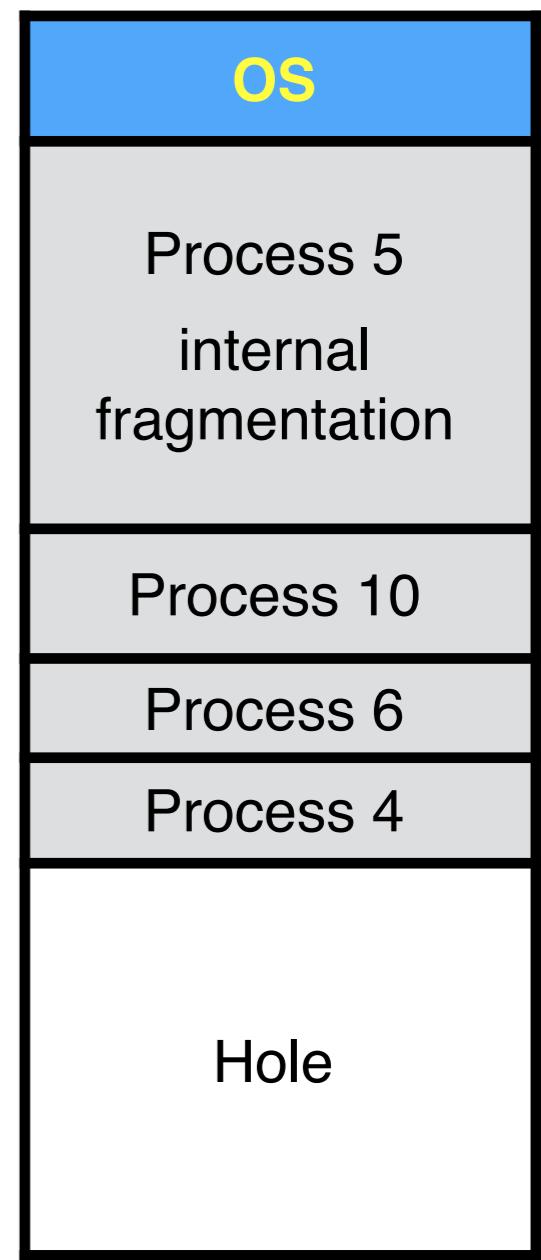
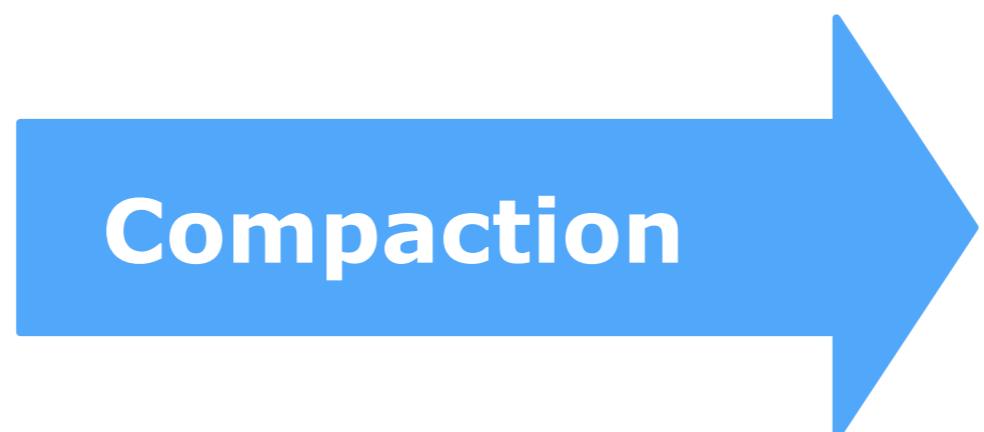
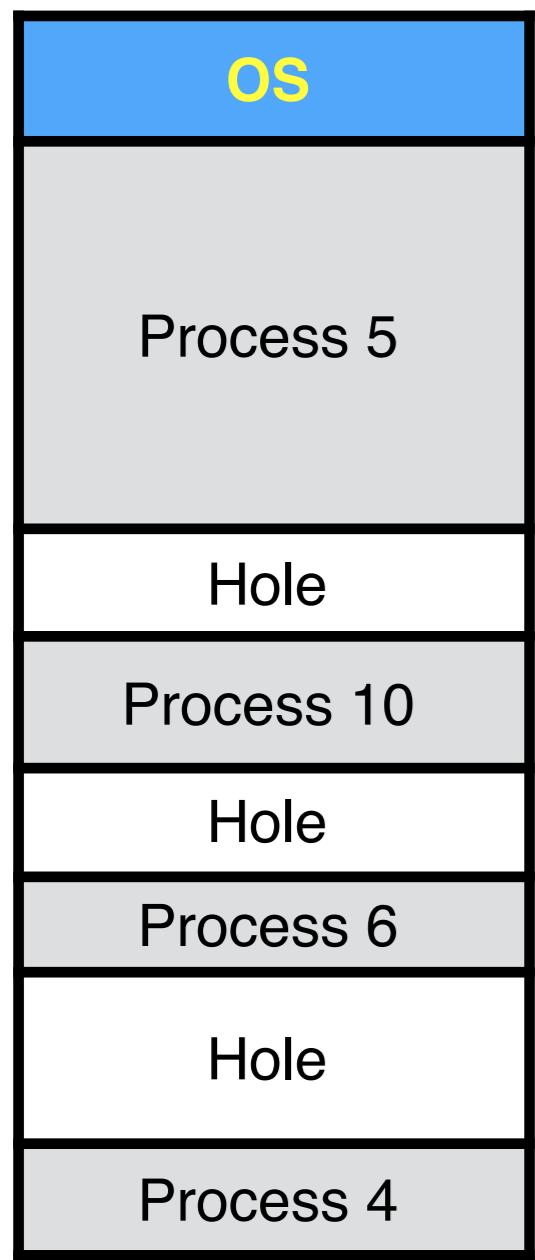


- ▶ **Fragmentation** leads to storage space being "wasted", and the term also refers to the wasted space itself.
- ▶ **Internal fragmentation:** allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.
- ▶ **External fragmentation:**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

**What to do
about external
fragmentation?**

Compaction

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

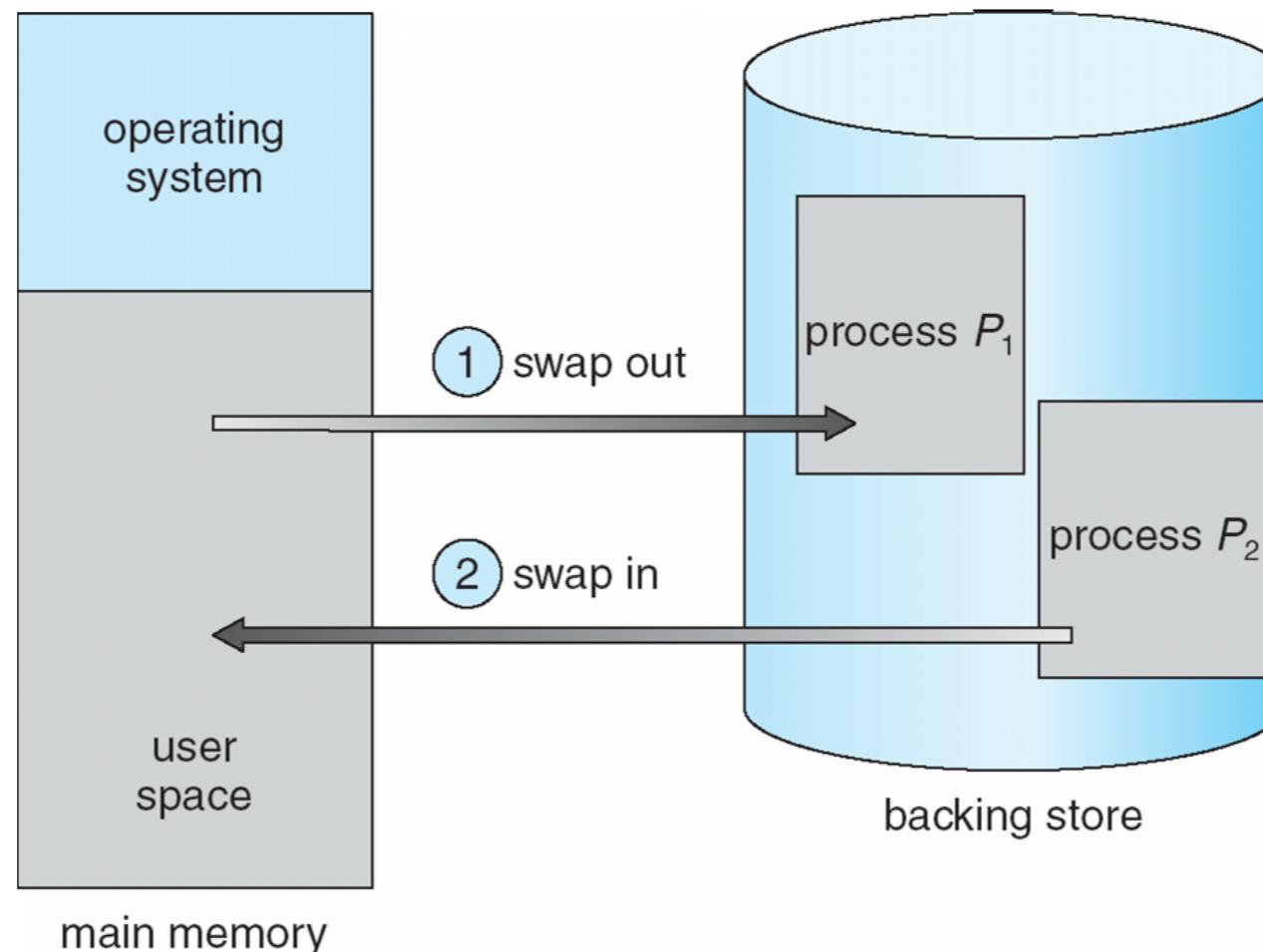


When should compaction be done?

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

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

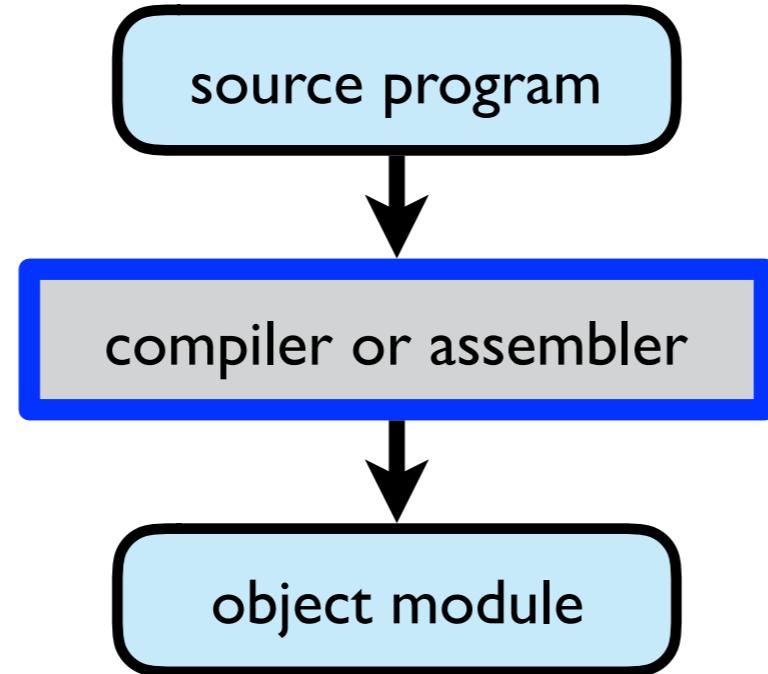
Swapping



If swapping is used, a process may move around in the physical memory during run time.

Address binding

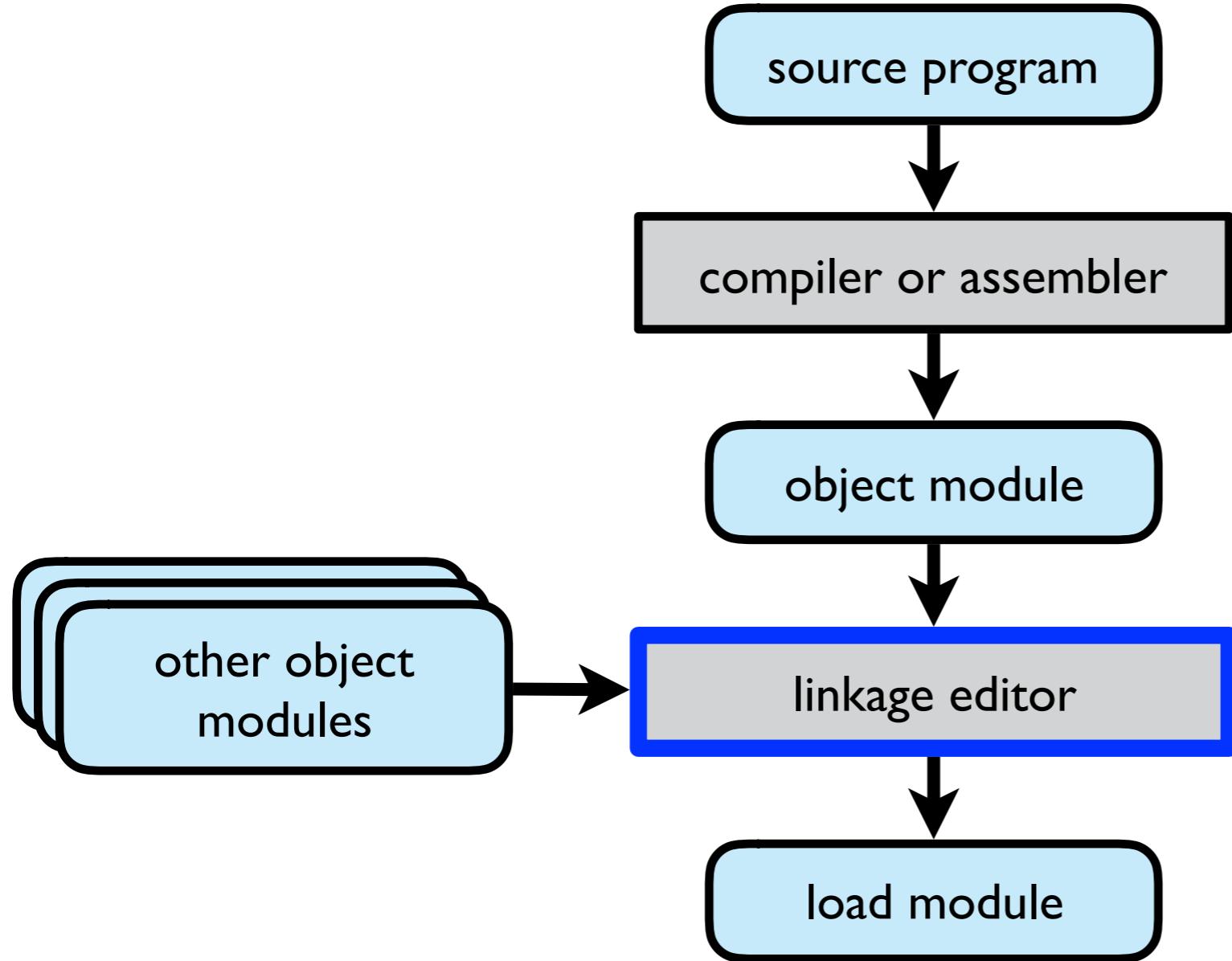
Address binding is the process of mapping the program's logical (or virtual addresses) to corresponding physical memory addresses.



A **compiler** is computer software that transforms computer code written in one programming language (the source language) into another programming language (the target language).

Source: https://en.wikipedia.org/wiki/Logical_address

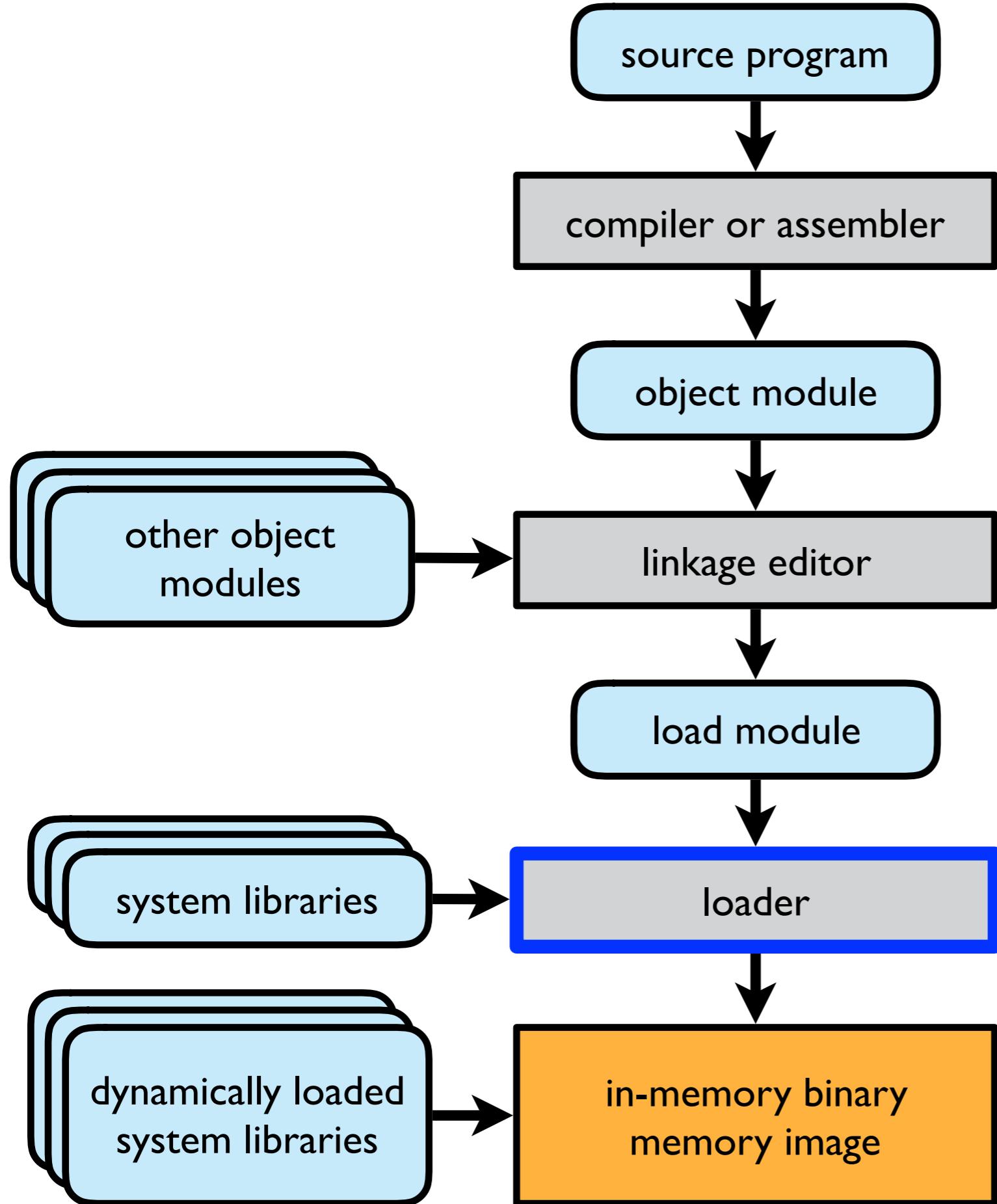
2018-02-21



The **linker** or link editor is a computer program that takes one or more object files generated by a compiler and combines them into single executable file, library file or another object file.

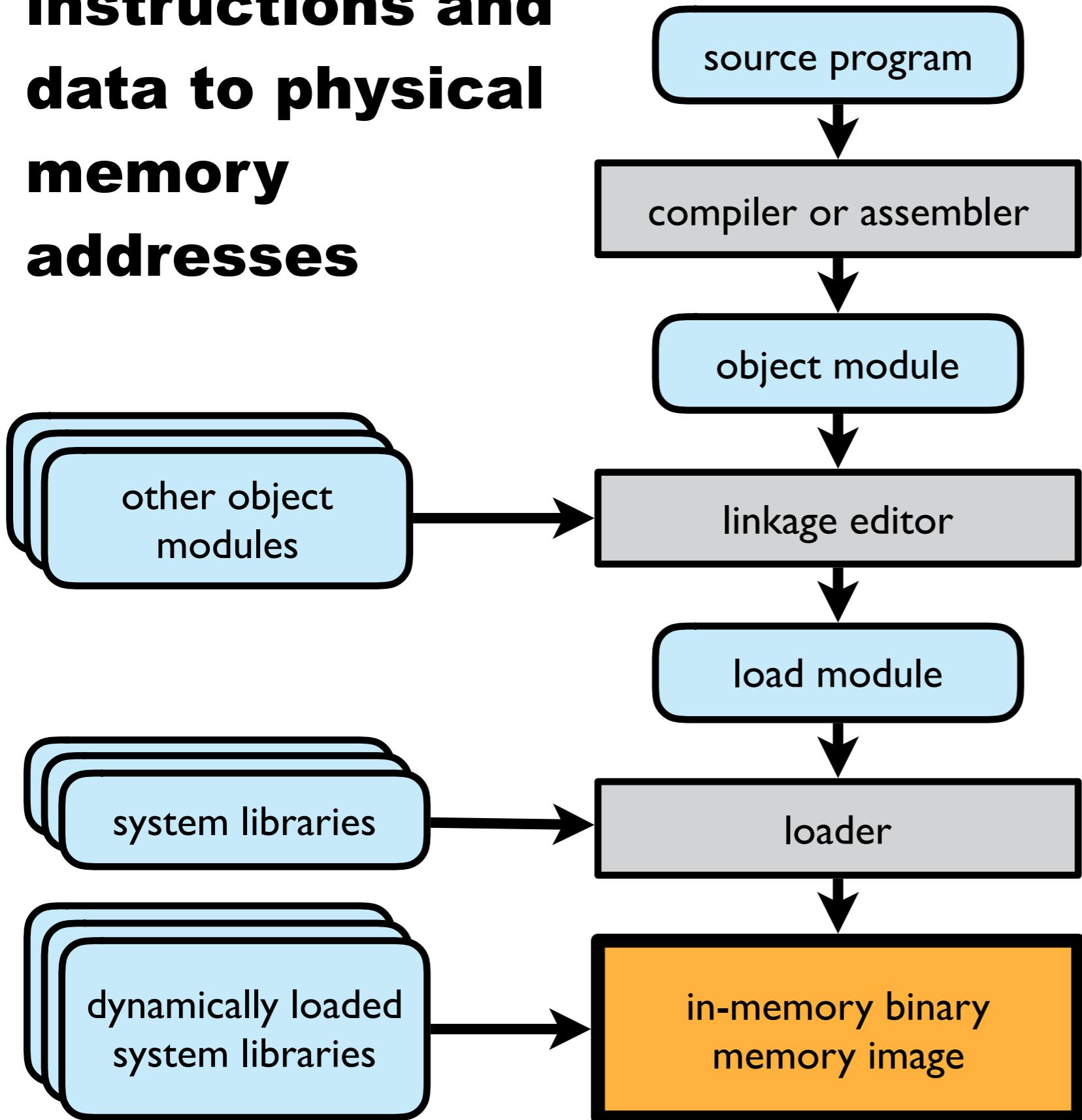
Source: [https://en.wikipedia.org/wiki/Linker_\(computing\)](https://en.wikipedia.org/wiki/Linker_(computing))

2018-02-21



The **loader** reads the contents of the executable file containing the program instructions into memory. Once loading is complete, the operating system starts the program by passing control to the loaded program code.

Binding of instructions and data to physical memory addresses



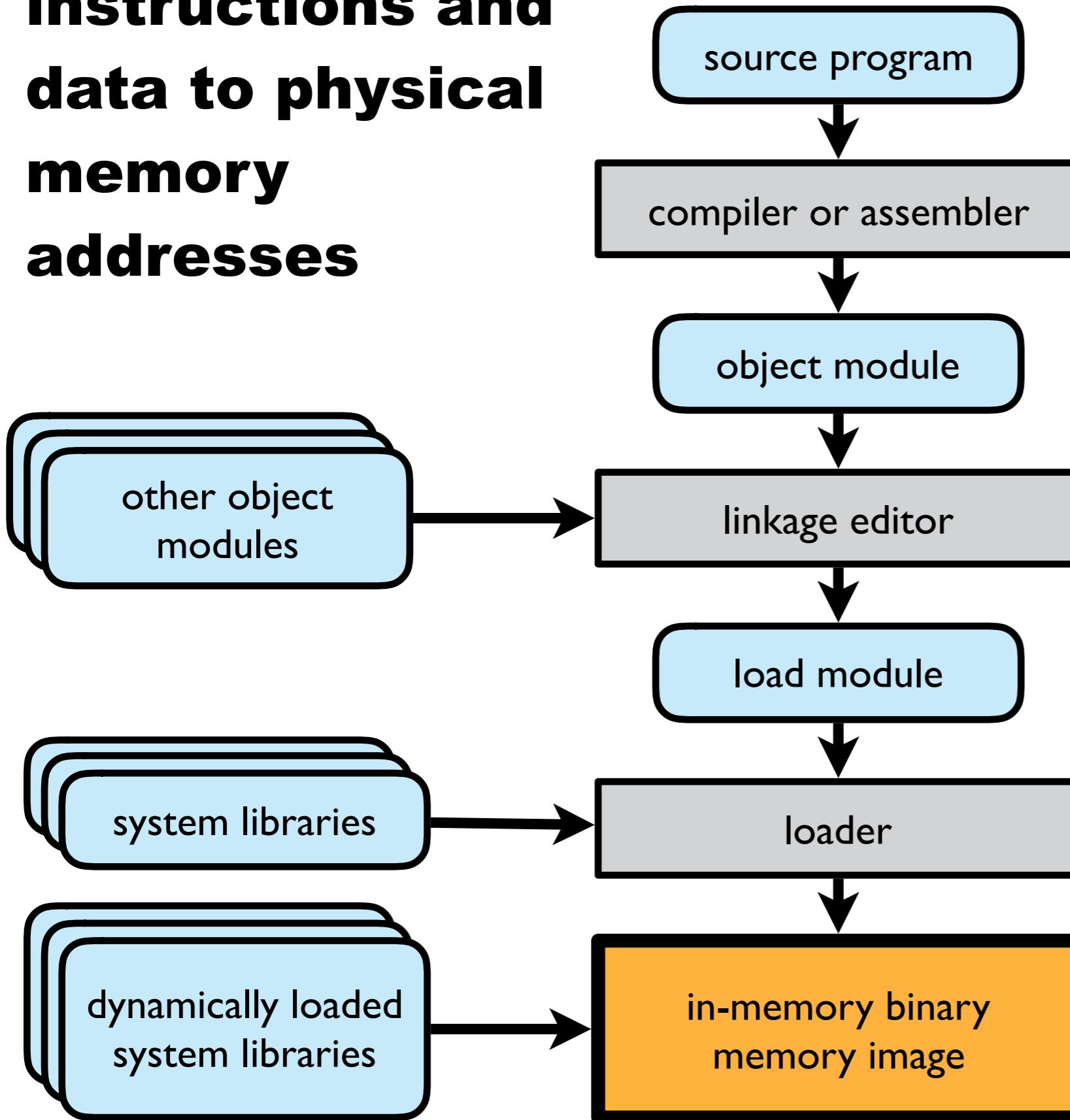
Address binding can take place at the following stages.

Compile time

Load time

Run time

Binding of instructions and data to physical memory addresses



If binding delayed until run time the process can be moved during its execution.

Need hardware support for address maps (e.g., base and limit registers)

Run time

Paging

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.



Decimal hens and eggs

1000 hens, each with a 3 digit decimal number 000, 001, ..., 999 and 100 baskets numbered 00, 01, ..., 99.



In which basket should eggs from hen 372 be placed? If we use a simple **direct mapped** scheme, put egg from hen 372 in basket $372 \bmod 100 = 72 = 372 \bmod 10^2 = 2$ last digits of $372 = 72$.

Binary hens and eggs

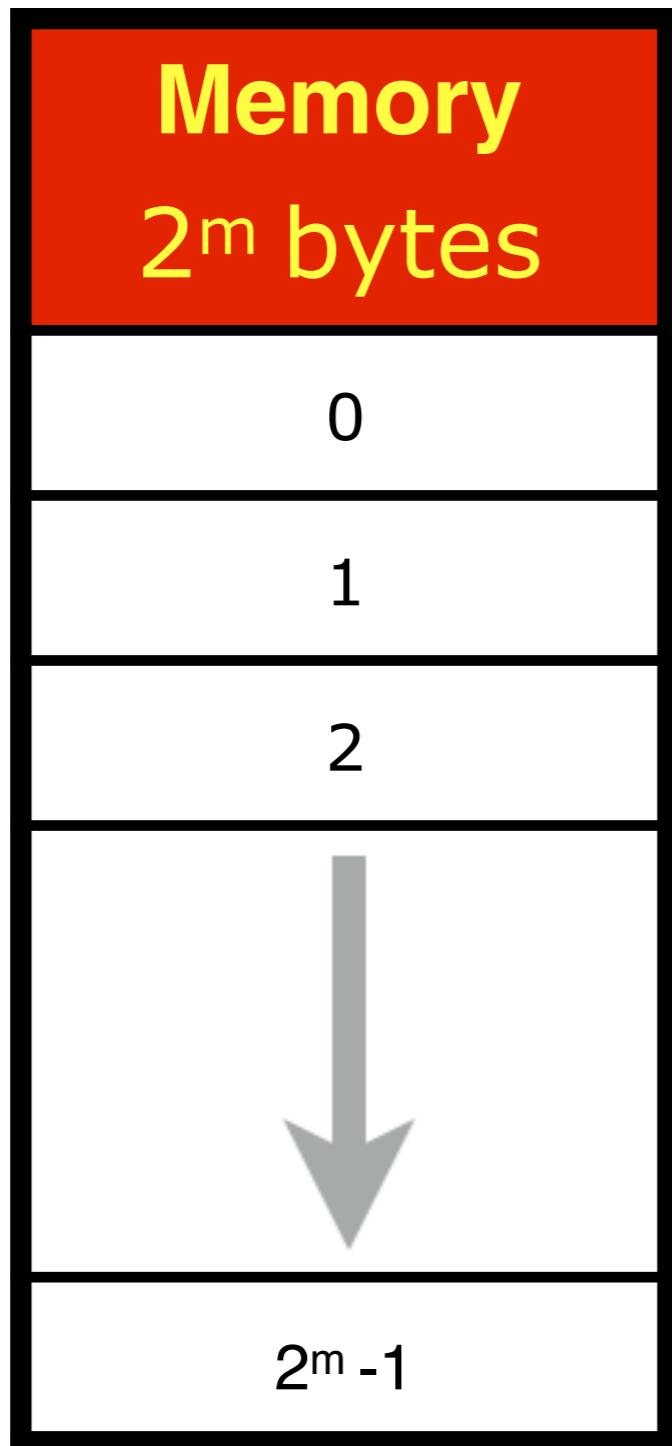
32 hens, each with a 5 bit binary number 00000, 00001, ..., 11111 and 8 baskets numbered 000, 001, ..., 111.



In which basket should eggs from hen 10110 be placed? If we use a simple **direct mapped** scheme, put egg from hen 10110 in basket $101100 \bmod 8 = 10110 \bmod 2^3 = 3$ last bits of 10110 = 110.

Addressing the memory

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



To address 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
 $0, 1, \dots, [2^{(m-n)}] - 1$

Pick a frame (address translation)

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

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

To address 2^m locations, we need a **m** bit address.

A =

m bit address

To which of the ($m - n$) frames should an address be mapped?

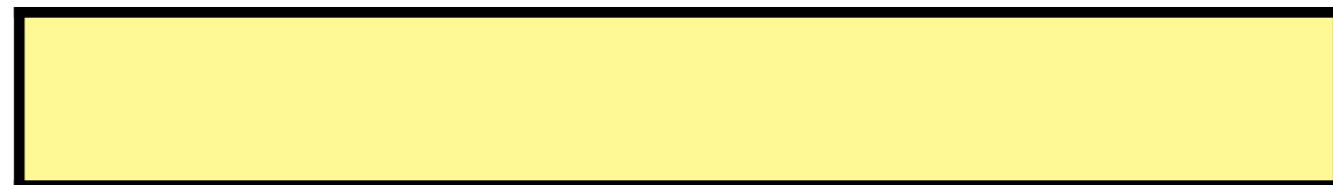
- ▶ Could map address A to frame $A \bmod (m-n)$, i.e., use the $(m-n)$ least significant bits of address A as frame number.
- ▶ But what if we use the $(n-m)$ most significant bits instead?

Pick a frame (address translation)

How can we map a logical address as seen by the CPU to a physical frame of size 2^n in memory of size 2^m ?

m bit logical address seen by the CPU

A =



Frame size = $2^n \Leftrightarrow m-n$ frames.

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

frame number

???

A =

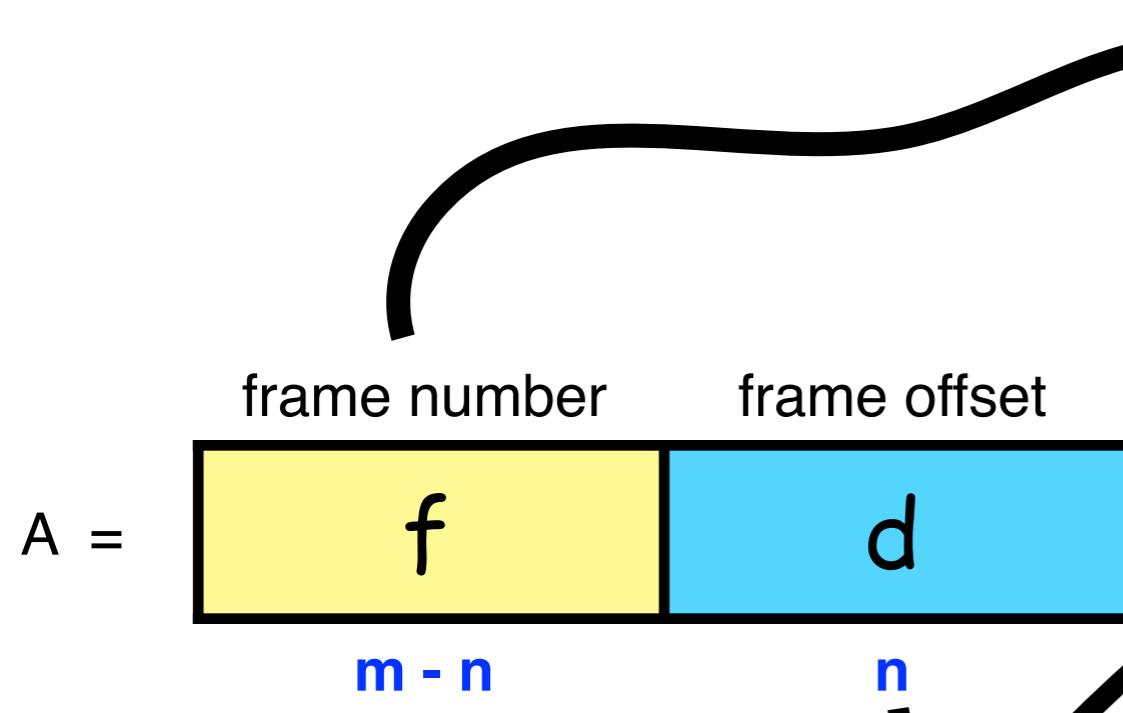


What does the n least significant bits mean?

| 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.



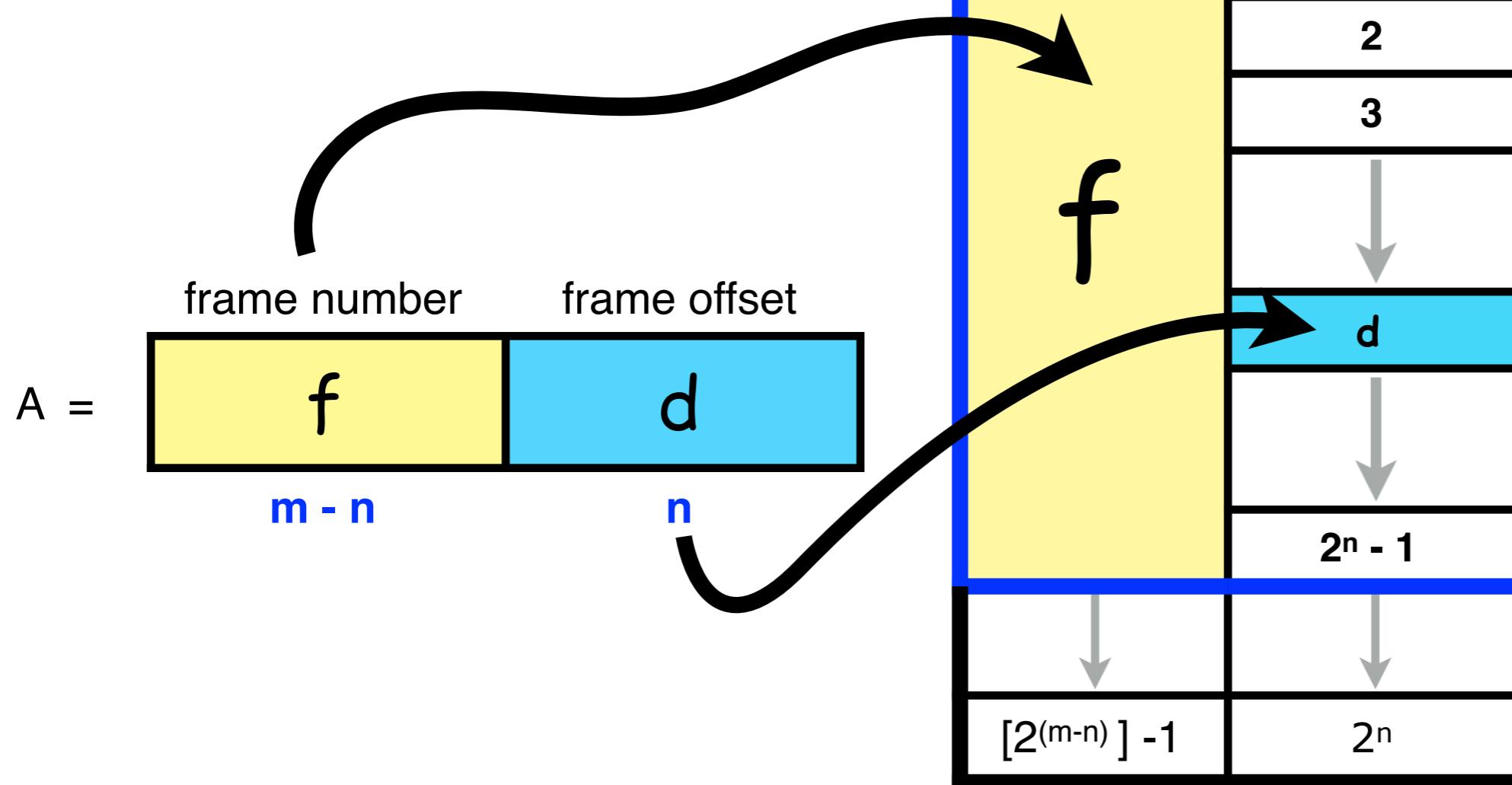
| Memory | |
|---------------------------|----------------|
| 2 ^m bytes | |
| Frame | Size (bytes) |
| 0 | 2 ⁿ |
| 1 | 2 ⁿ |
| 2 | 2 ⁿ |
| 3 | 2 ⁿ |
| ... | ... |
| [2 ^(m-n)] - 1 | 2 ⁿ |

Each frame is 2^n bytes.

We number the bytes in a frame $0, 1, 2, \dots, 2^n - 1$

Byte d within frame f .

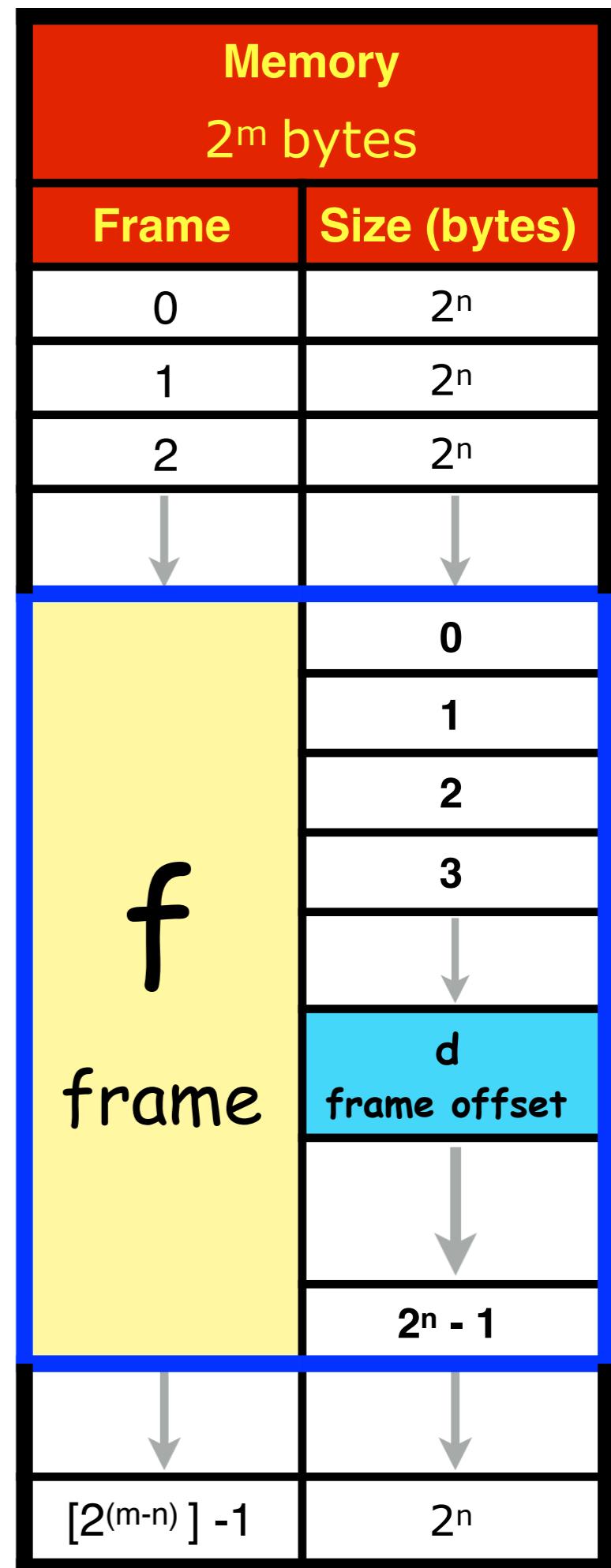
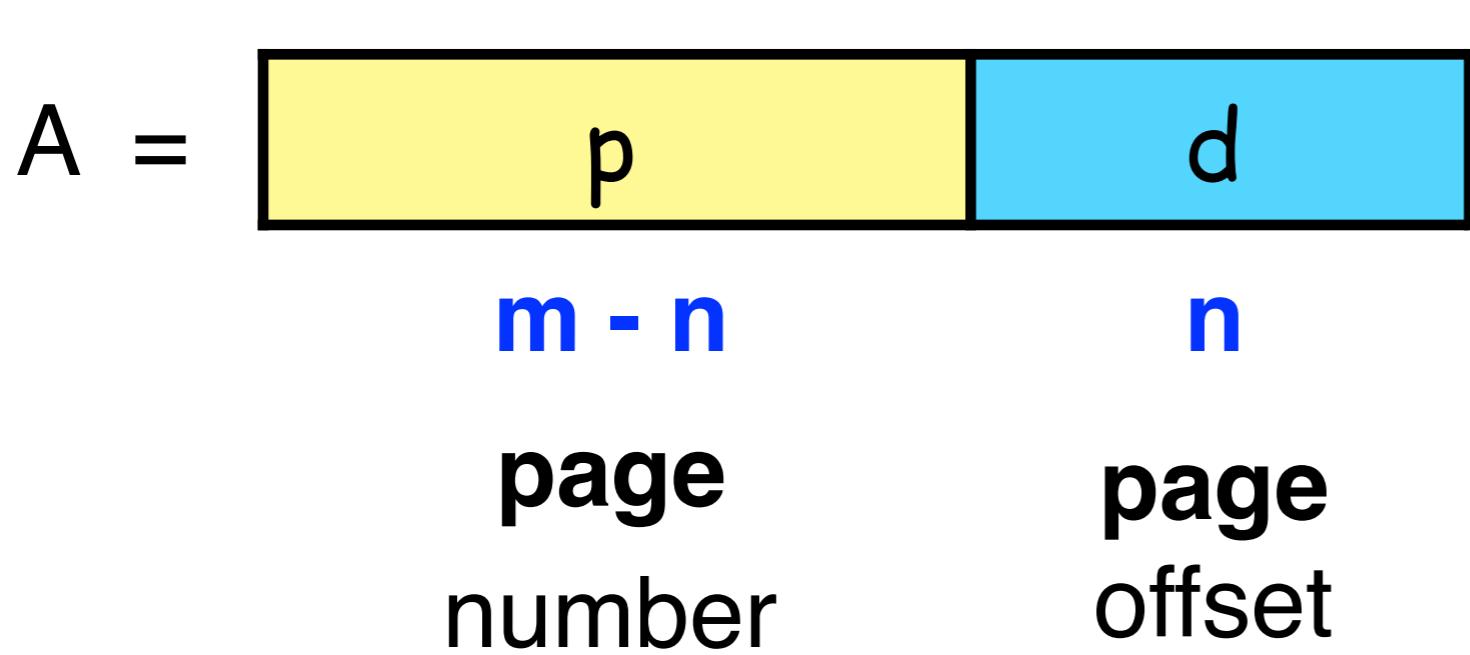
- ▶ Using this scheme, a logical address will always be mapped to the same physical frame.
- ▶ The physical memory must still be allocated as a sequence of contiguous frames.



Pages and frames

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

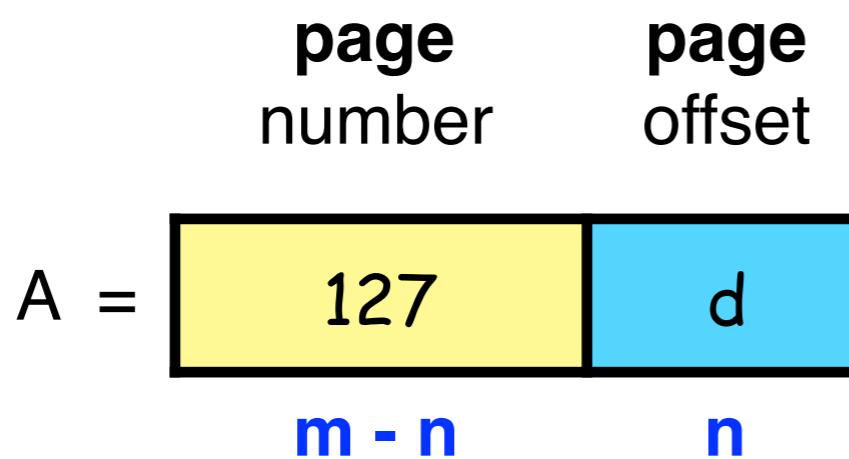
- ▶ **Logical** address space divided into fixed sized **pages**.
- ▶ **Physical** memory divided into **frames** of the same fixed size as the pages.



Page table

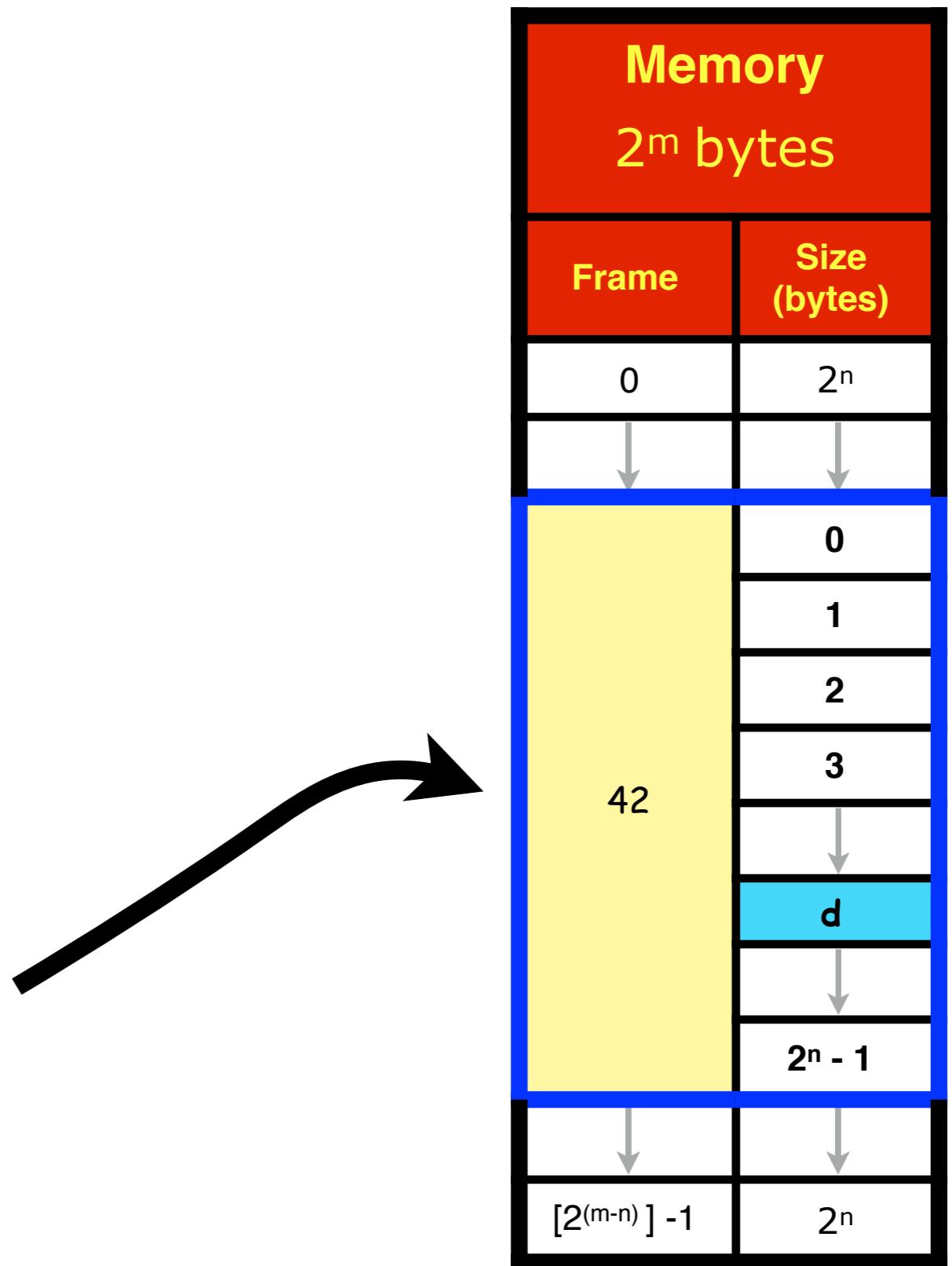
A page lookup table maps logical pages to physical frames.

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



Page table

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



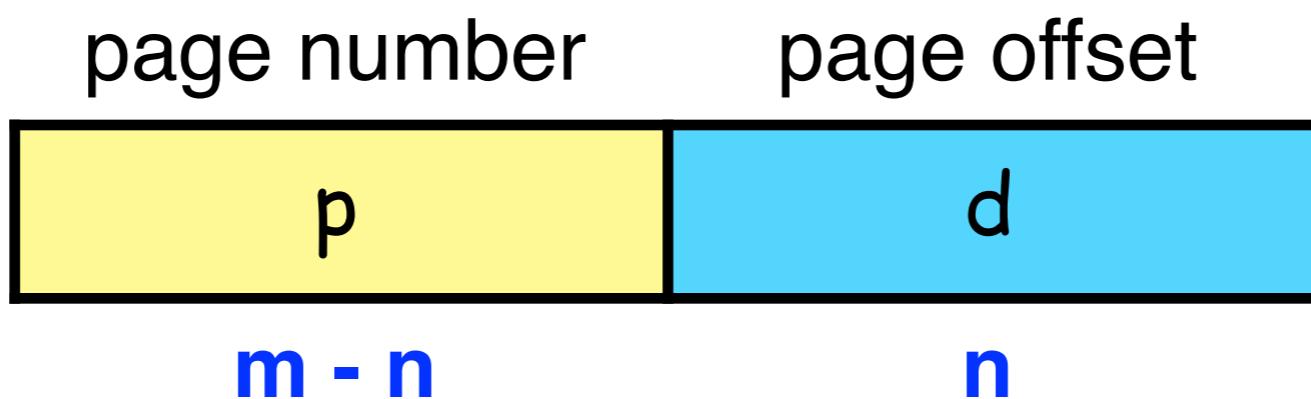
Paged memory management

- ▶ 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**.

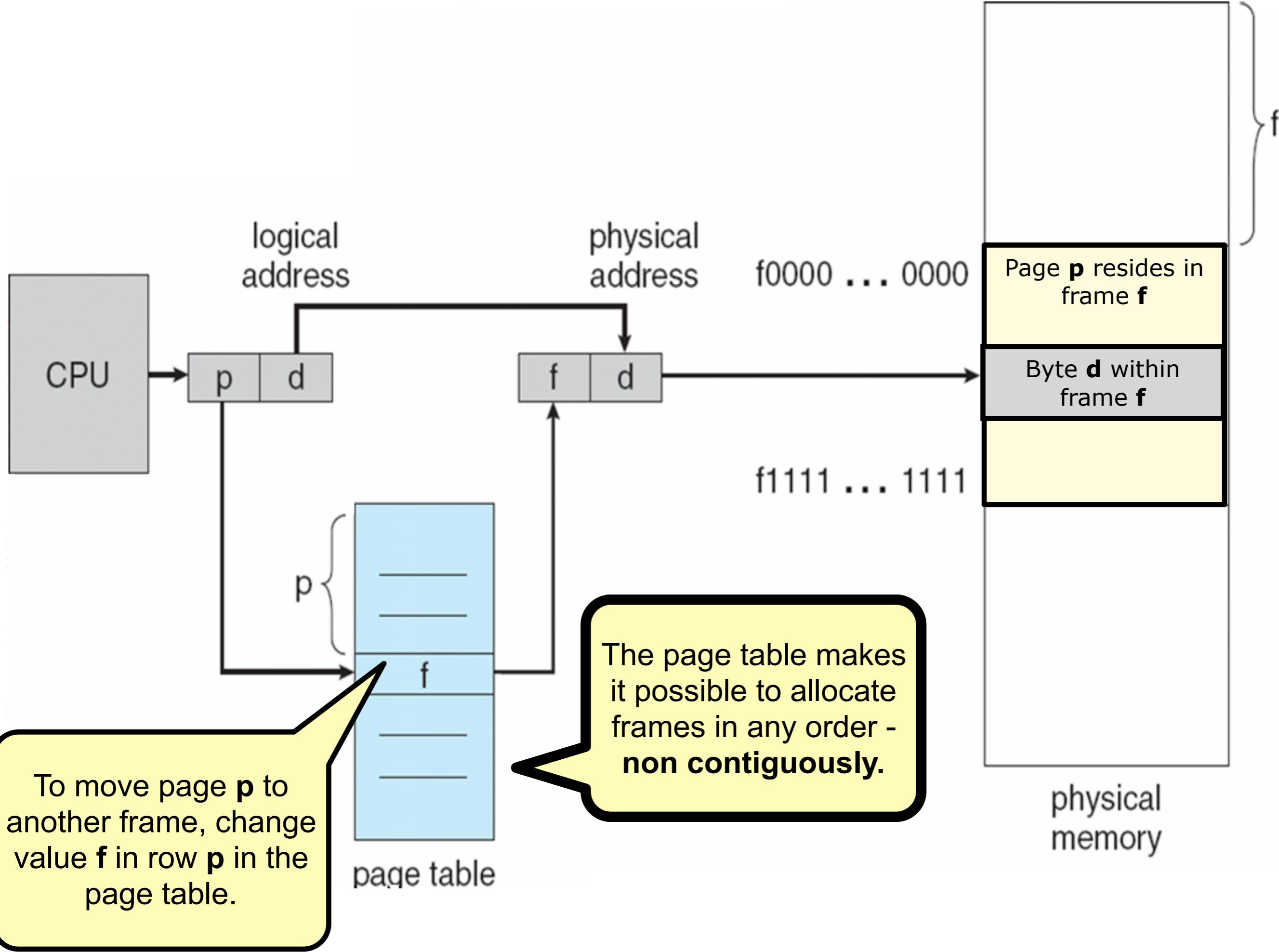
Address translation scheme

Logical address generated by CPU is divided into:

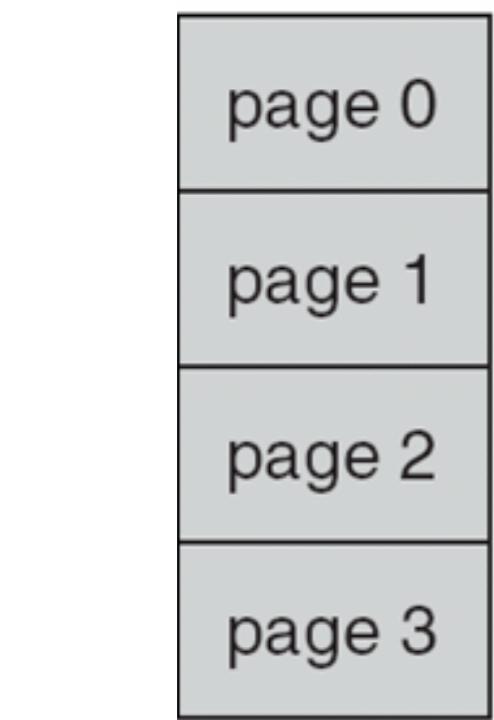
- ▶ **Page number (p)** – used as an index into a page table which contains base address of each frame in physical memory
- ▶ **Page offset (d)** – combined with base address to define the physical memory address that is sent to the memory unit.



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



Currently executing process



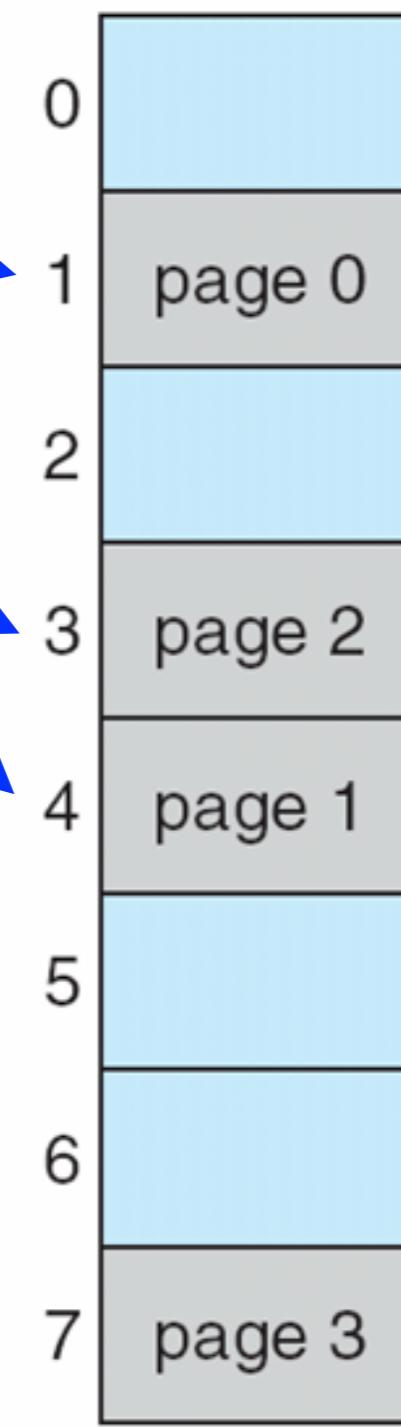
logical
memory



CPU

Physical memory

frame
number



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

How many frames?

$$(32 \text{ bytes}) / [(4 \text{ bytes})/\text{frame}] = 8 \text{ frames}$$

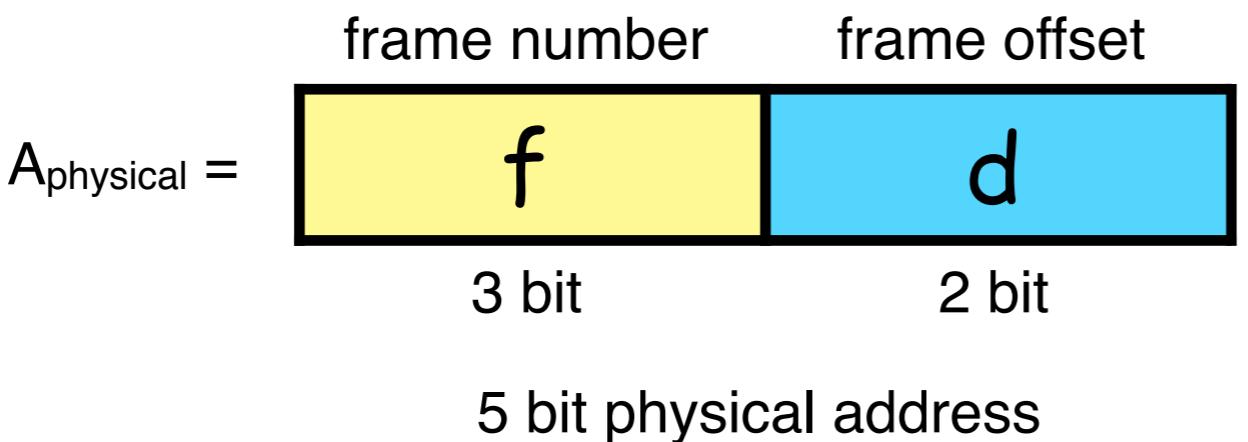
How large physical address?

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

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

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

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



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

32 byte physical memory and 4 byte logical pages

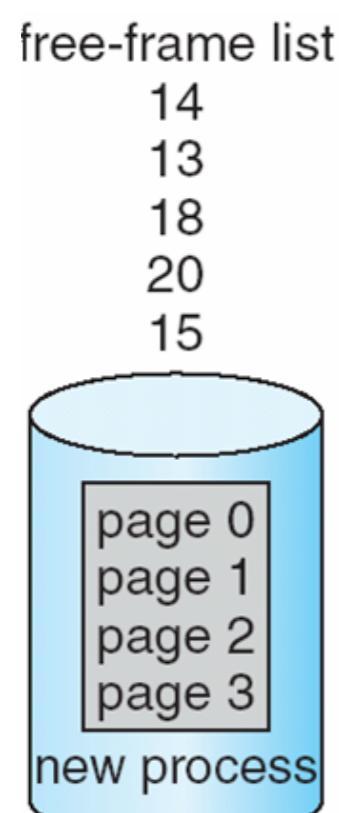
| Logical memory | |
|----------------|--------------|
| Page | Byte content |
| 0 | a |
| | b |
| | c |
| | d |
| 1 | e |
| | f |
| | g |
| | h |
| 2 | i |
| | j |
| | k |
| | l |
| 3 | m |
| | n |
| | o |
| | p |

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

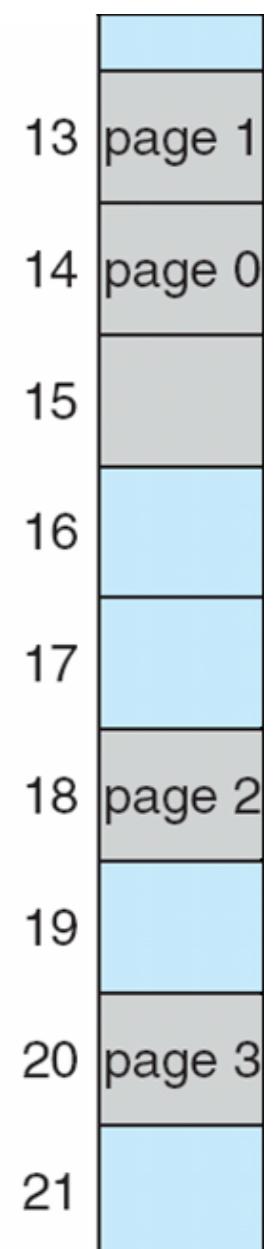
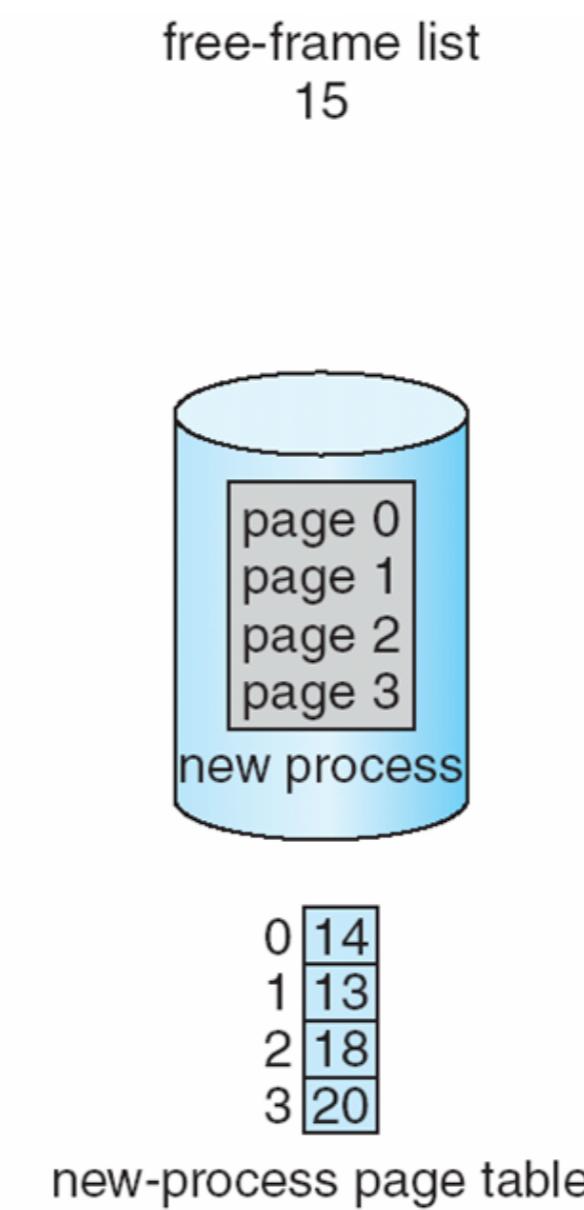
| Physical Memory 32 bytes | | | |
|-----------------------------|-------|--------|---------|
| byte | frame | offset | content |
| 0 | 0 | 0 | |
| 1 | | 1 | |
| 2 | | 2 | |
| 3 | | 3 | |
| 4 | | 0 | i |
| 5 | | 1 | j |
| 6 | | 2 | k |
| 7 | | 3 | l |
| 8 | 1 | 0 | m |
| 9 | | 1 | n |
| 10 | | 2 | o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

List of free frames

Before allocation

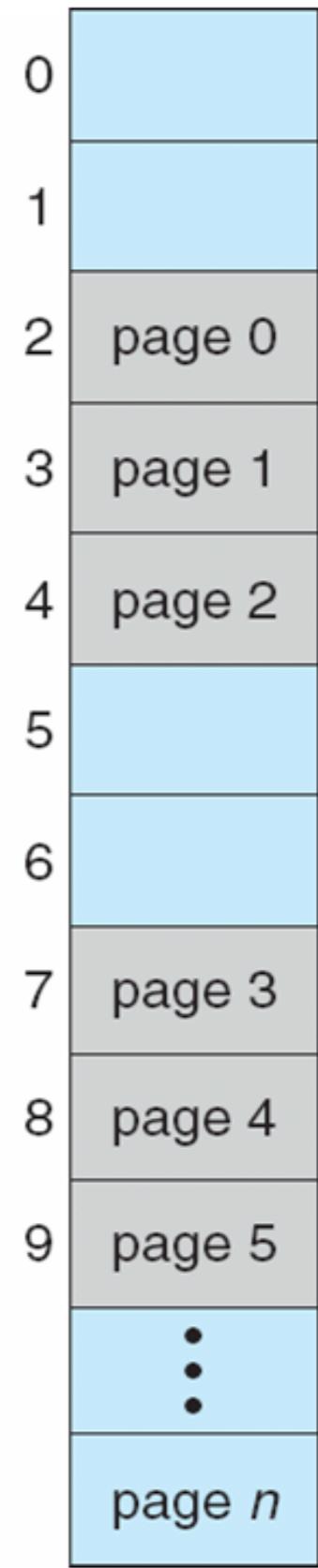
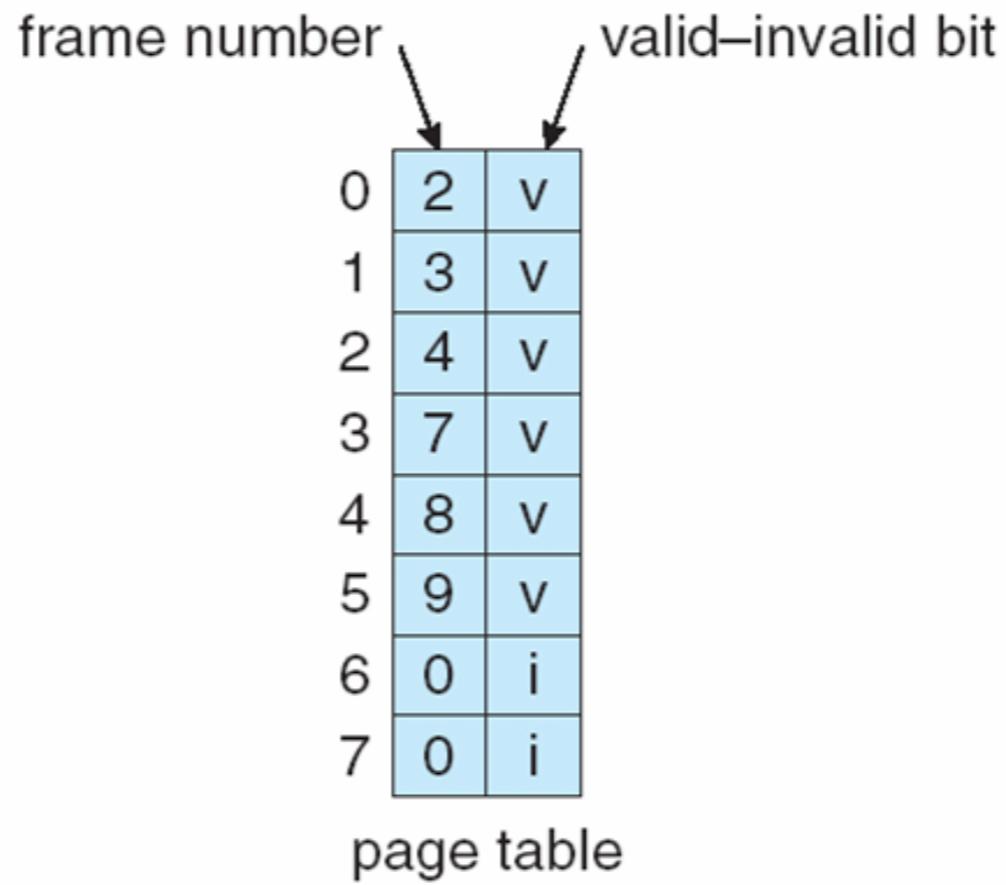
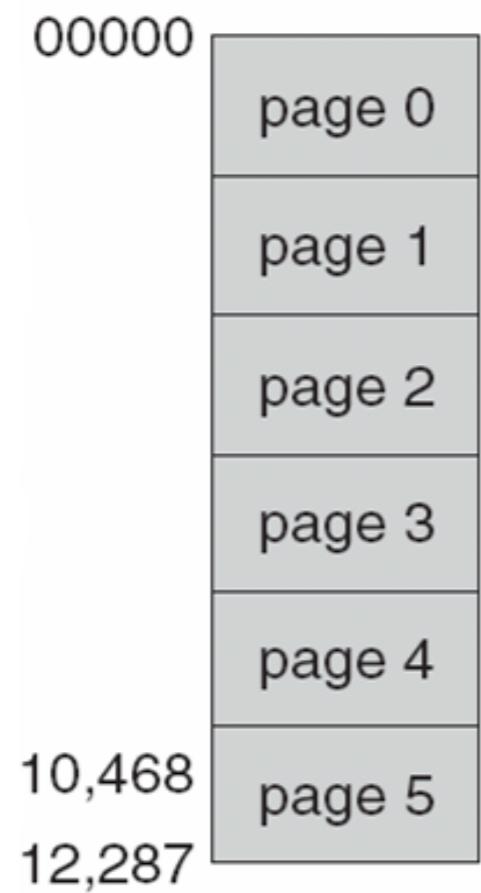


After allocation



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.

Shared pages

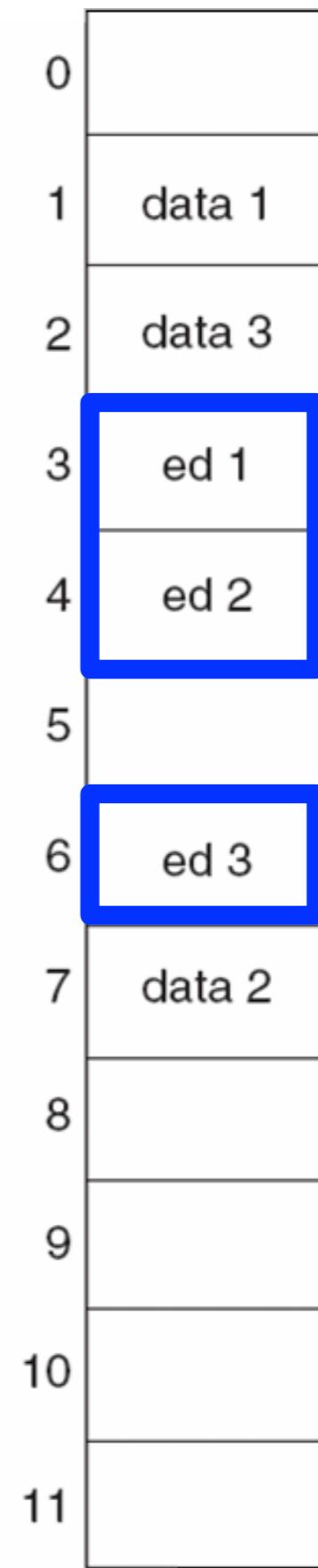
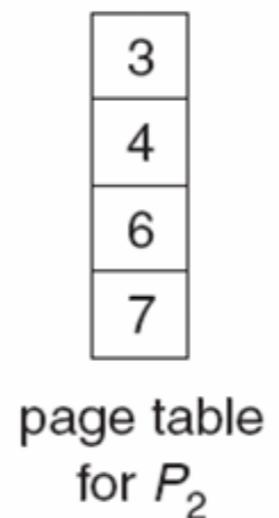
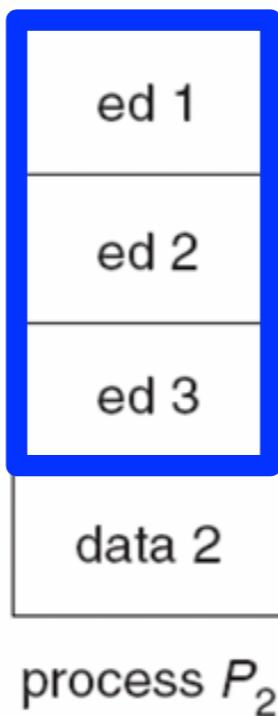
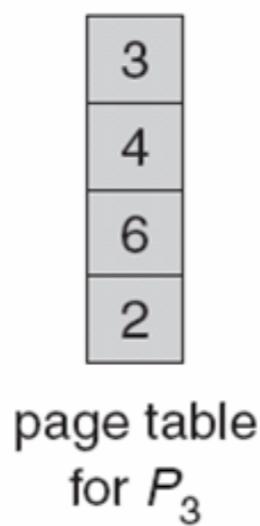
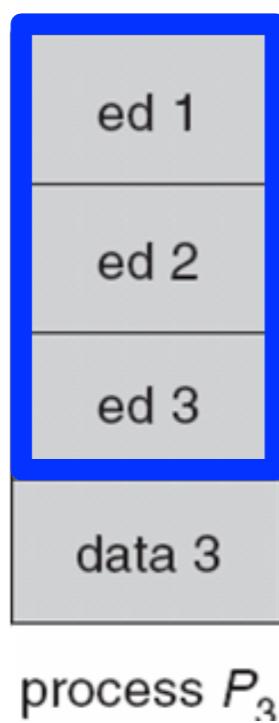
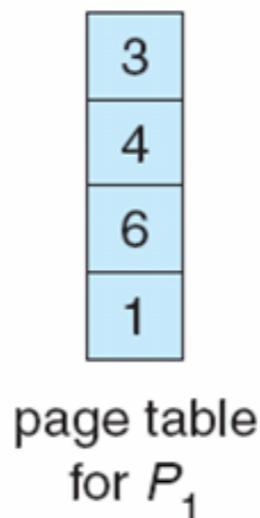
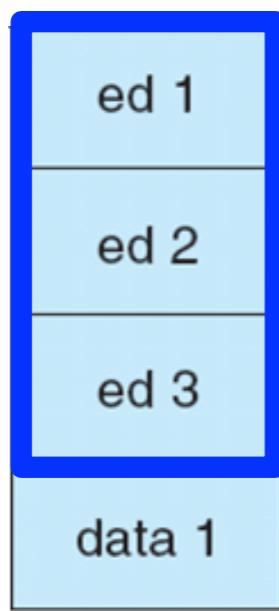
Paging makes it possible for processes to share parts of their memory space with each other.

Shared code

- ▶ One copy of **read-only** (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
- ▶ Shared code must appear in same location in the logical address space of all processes.

Private code and data

- ▶ Each process keeps a separate copy of the code and data.
- ▶ The pages for the private code and data can appear anywhere in the 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.

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

$\alpha \rightarrow 1$ when $\# \text{ TLB misses} \rightarrow 0$

$\alpha \rightarrow 0$ when $\# \text{ TLB hits} \ll \# \text{ TLB misses}$

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.

Effective access time (3)

Effective access time (EAT).

$$\text{EAT} = P(\text{TLB hit}) \cdot (1 + \varepsilon)\alpha + P(\text{TLB miss}) \cdot \frac{(2 + \varepsilon)(1 - \alpha)}{2 + \varepsilon - \alpha}$$

↓

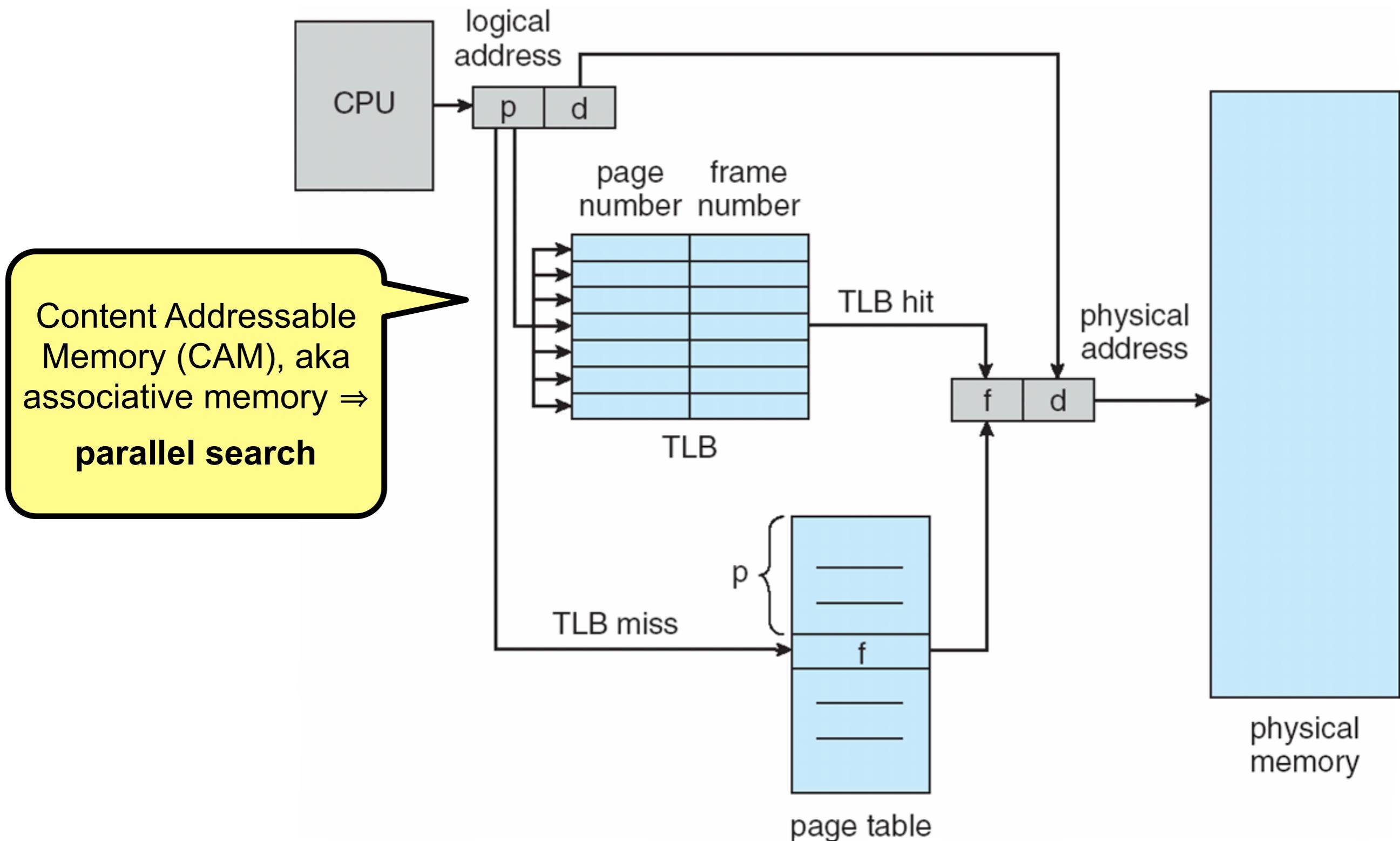
▶ ε for TLB lookup
▶ 1 ms for memory access

↓

▶ ε for TLB
▶ 1 ms for in memory page table lookup access
▶ 1 ms for memory access

Paging hardware with TLB

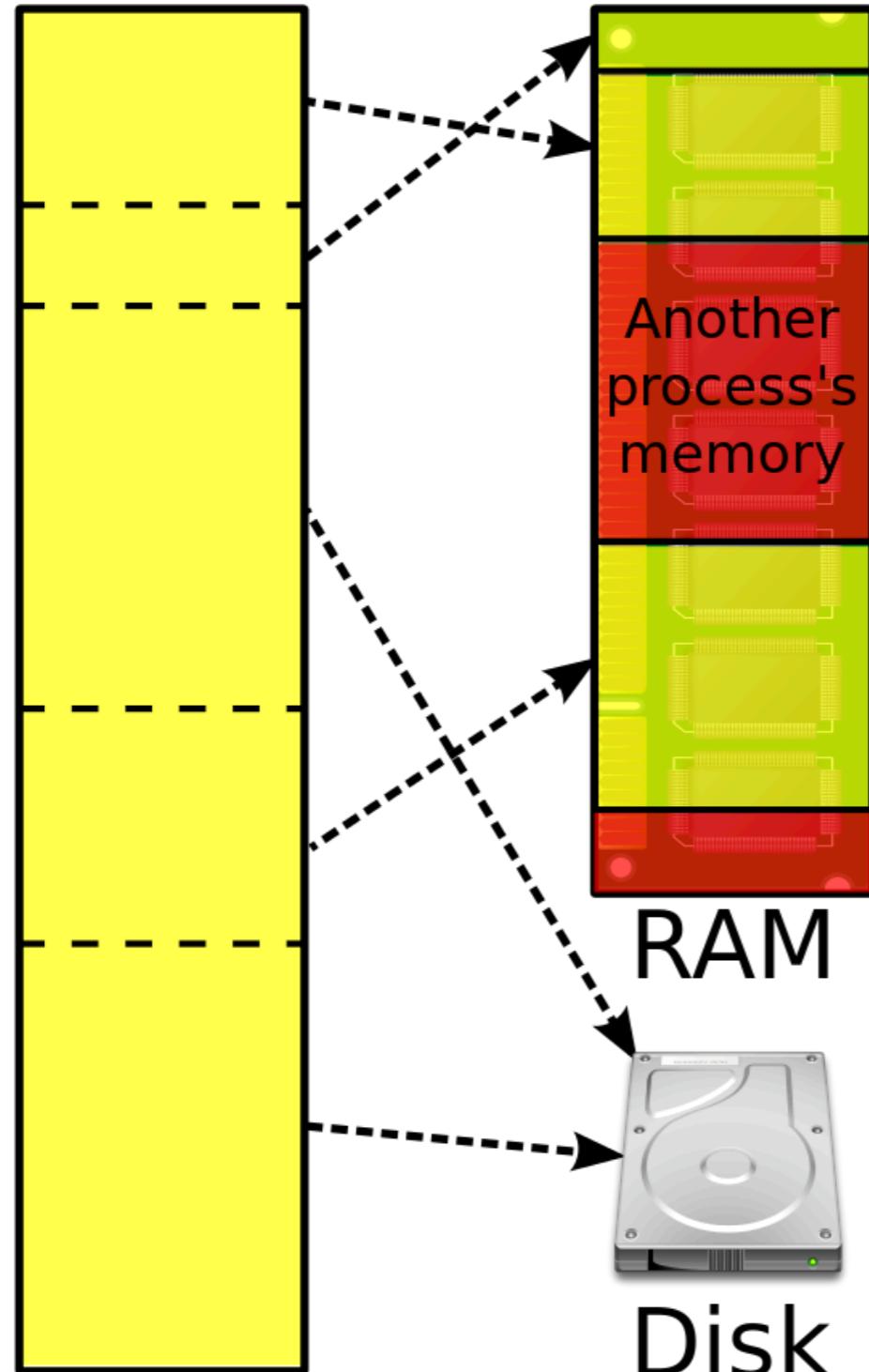
A **translation lookaside buffer** (TLB) is a **cache** that memory management **hardware** uses to improve virtual address translation speed.



Virtual memory

Virtual memory provides an idealized abstraction of the physical memory which creates the illusion of a larger virtual memory than the physical memory.

Virtual memory
(per process)



Virtual memory combines active RAM and inactive memory on disk to form a **large range of virtual contiguous addresses**.

Using **paging** a process can conceptually use more memory than might be physically available.

Pages are **swapped in** from disk to RAM and **swapped out** from RAM dynamically.

Virtual memory is an integral part of a modern computer architecture; implementations usually require **hardware support**, typically in the form of a memory management unit built into the CPU.