Memory management

Module 5

Lecture

Operating systems 2020

1DT003, 1DT044 and 1DT096

karl.marklund@it.uu.se

Course timelines

Period 3				Modul 4			Modul 5			Modul 6			Modul 7			Modul 8	
W	D	Date	Bivillkor	OS	OSPP	DSP	OS	OSPP	DSP	OS	OSPP	DSP	OS	OSPP	DSP	OSPP	DSP
9	Mo	2020-02-24		T		L (KM)											
	Tu	2020-02-25			T	L (KM)											
	We	2020-02-26					L (KM)										
	Th	2020-02-27		S						L (KM)							
	Fr	2020-02-28			S	W			L (KM)								
10	Mo	2020-03-02				2 x L (LÅ)	W										
	Tu	2020-03-03		C		T		W									
	We	2020-03-04							L (LÅ)								
	Th	2020-03-05		Tr	C + Tr							L (LÅ)	W				
	Fr	2020-03-06		Sr	Sr	T + S			W					W			
11	Mo	2020-03-09				C	S										W
	Tu	2020-03-10				CL		S									
	We	2020-03-11			Cr + Ch				S		Pc	W	W				
	Th	2020-03-12	Avlyst dag (DV/IT)	Cr + Ch						Pc				W			
	Fr	2020-03-13	Annan tenta (DV/IT)				Sr										
12	Mo	2020-03-16	Avlyst dag (DV/F)			Sr + Tr											W
	Tu	2020-03-17	Annan tenta (DV/F)			CLr+Cr+Ch			Sr			S					
	We	2020-03-18	Avlyst dag														
	Th	2020-03-19	Tentamen										E	E			E
	Fr	2020-03-20						Sr		Pcr	Pcr + Dpr				L (LÅ)		

Background

Human user

Non-interactive process with no human user

Human user

Human user

Process A

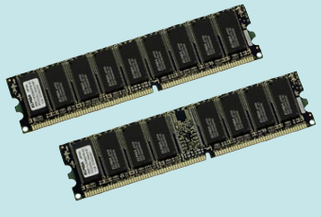
Process B

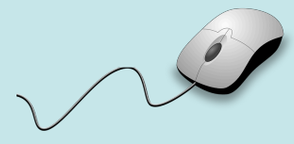
Process B

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

CPU

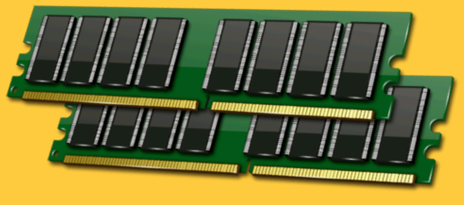
Disk controller

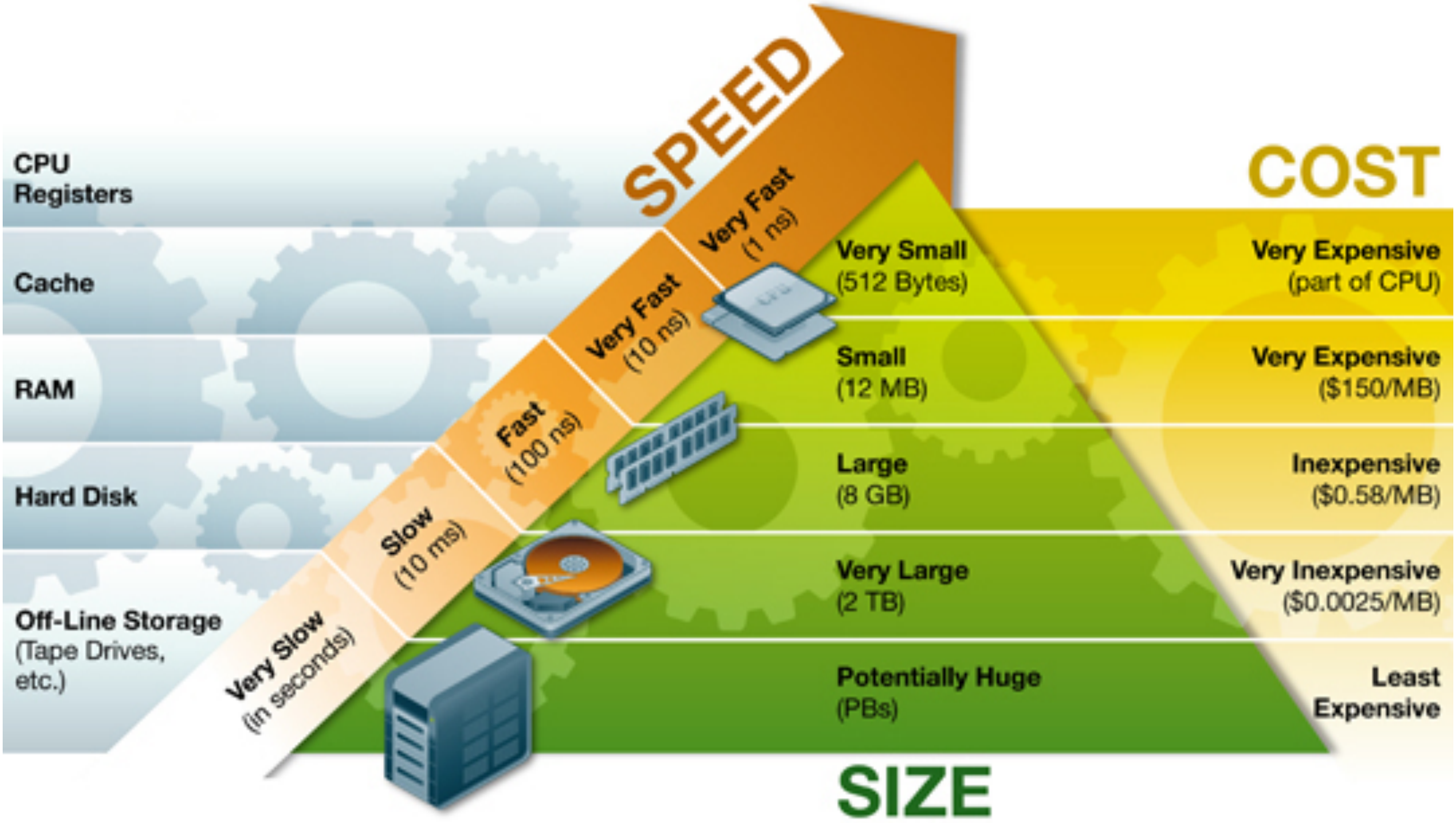
USB controller

Graphics adapter

Memory controller

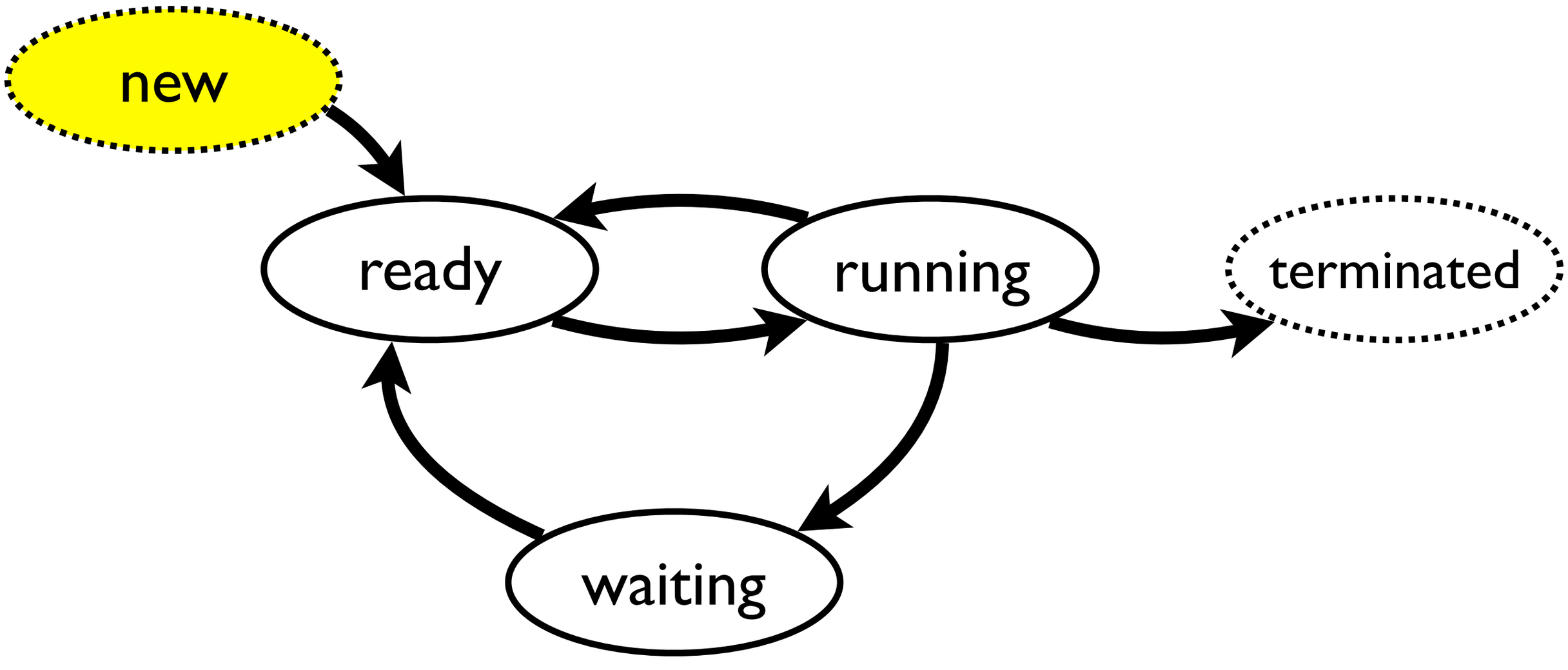
Main memory




CPU
Registers
Cache
RAM
Hard Disk
Off-Line Storage
(Tape Drives,
etc.)
SPEED
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)
COST
Very Expensive
(part of CPU)
Very Expensive
($150/MB)
Inexpensive
($0.58/MB)
Very Inexpensive
($0.0025/MB)
Least
Expensive
SIZE

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

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

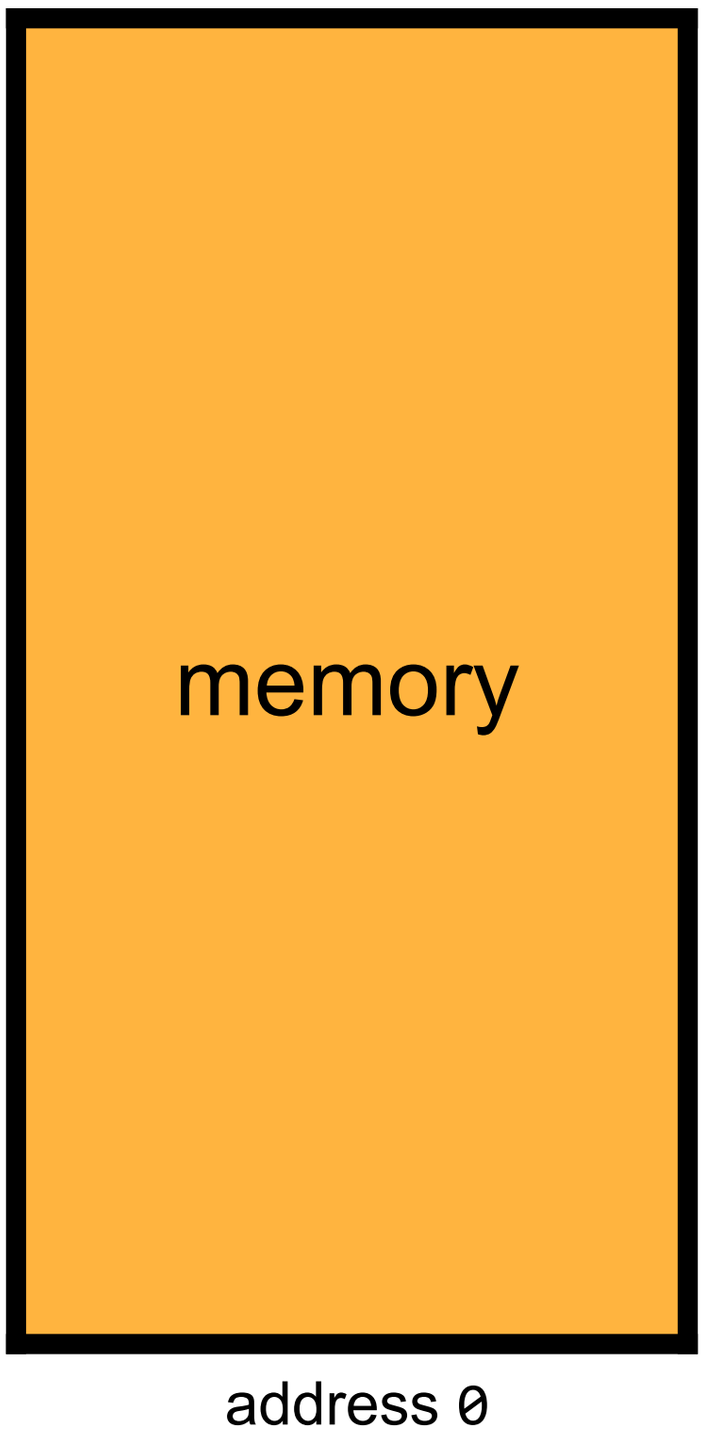
Example of information stored in the PCB

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

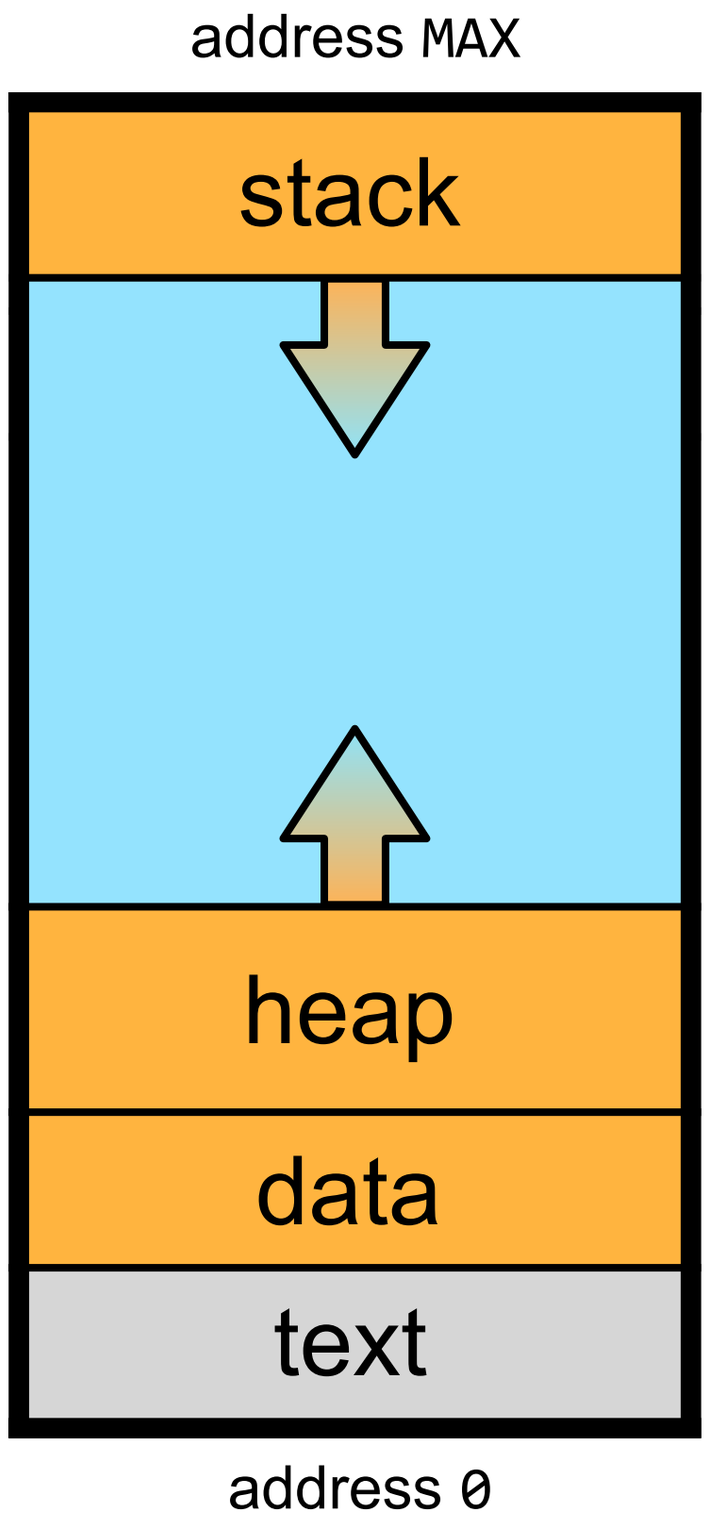
In addition to a PCB, the operating system must allocate a new memory space for each new process. This memory space is often referred to as the process **memory image**.

Static memory allocation



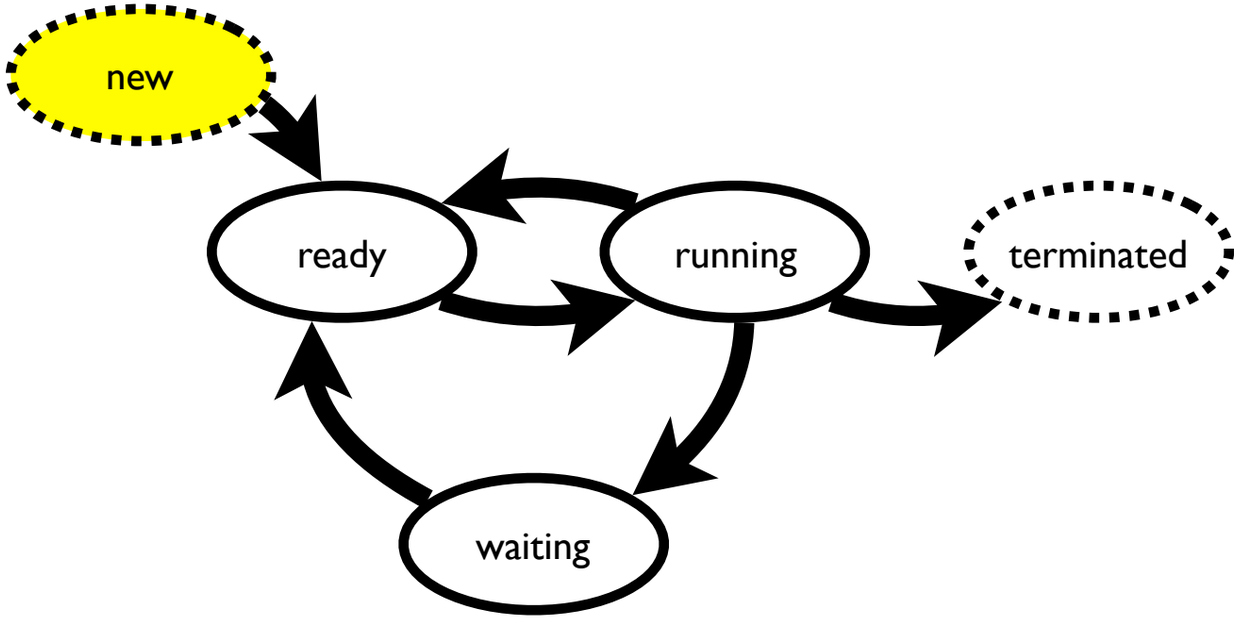
The operating system must allocates a blob of memory for the new process memory image.

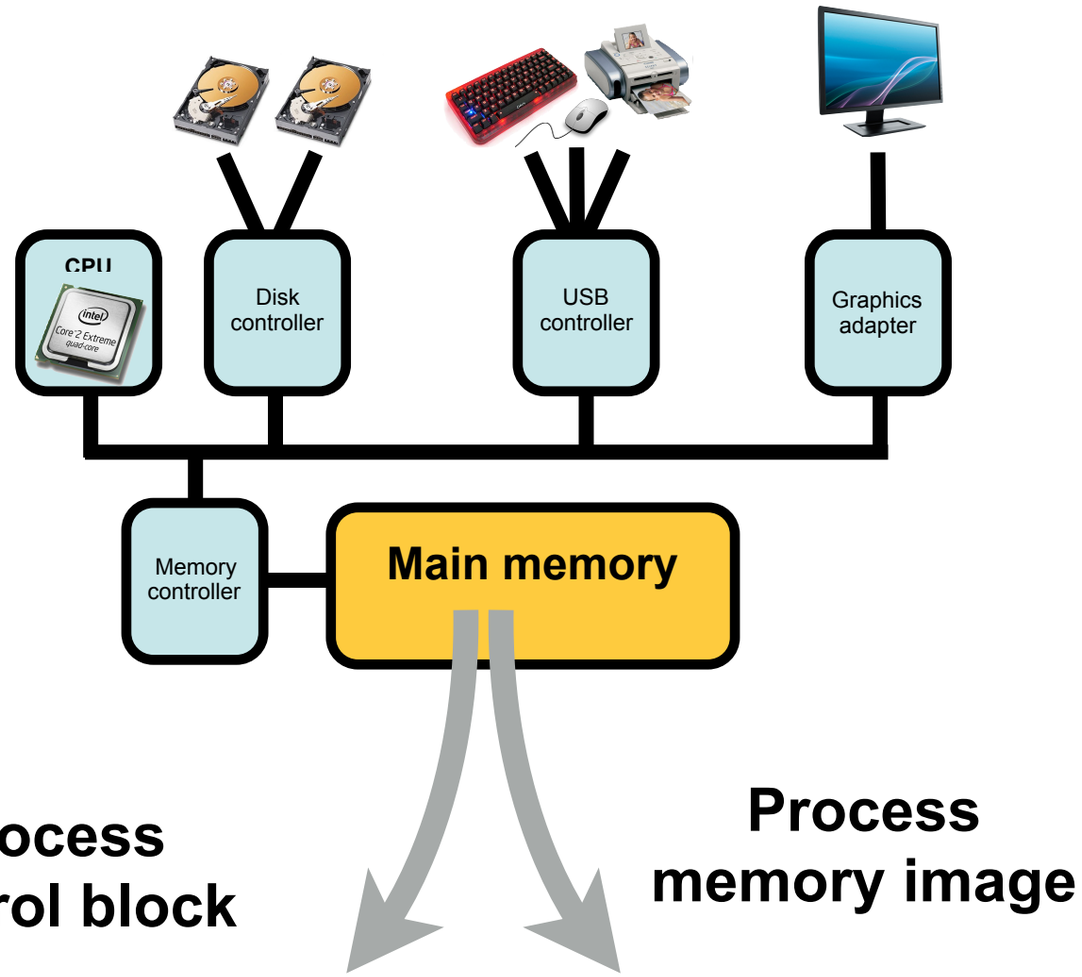
Static memory allocation



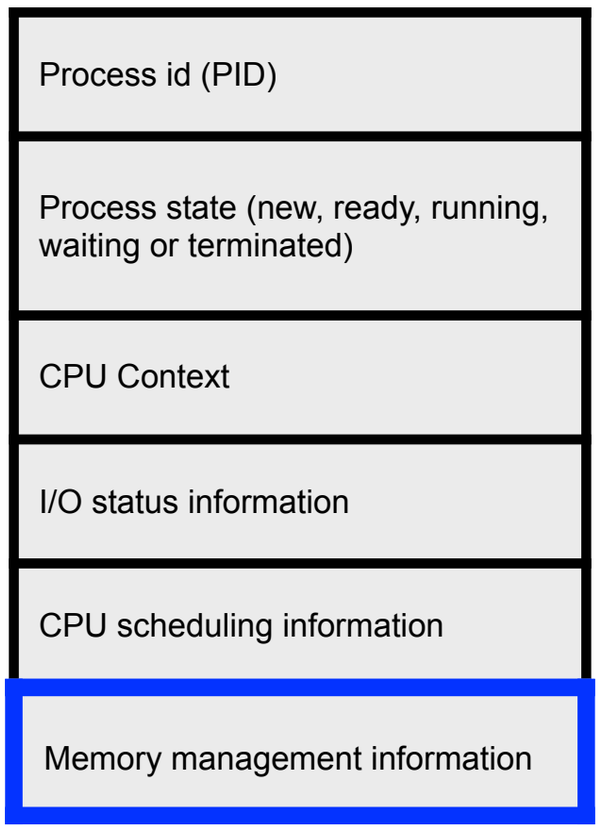
The allocated memory image is divided into the following segments:

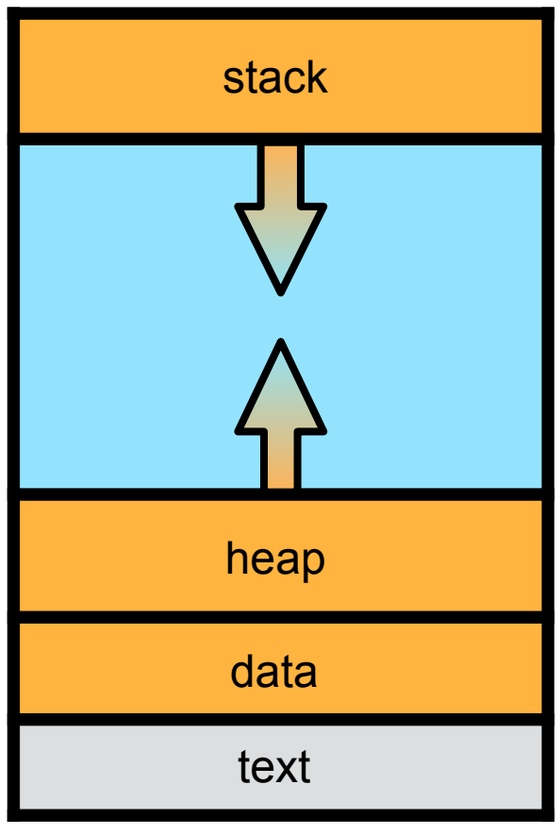
- text
- data
- heap
- stack





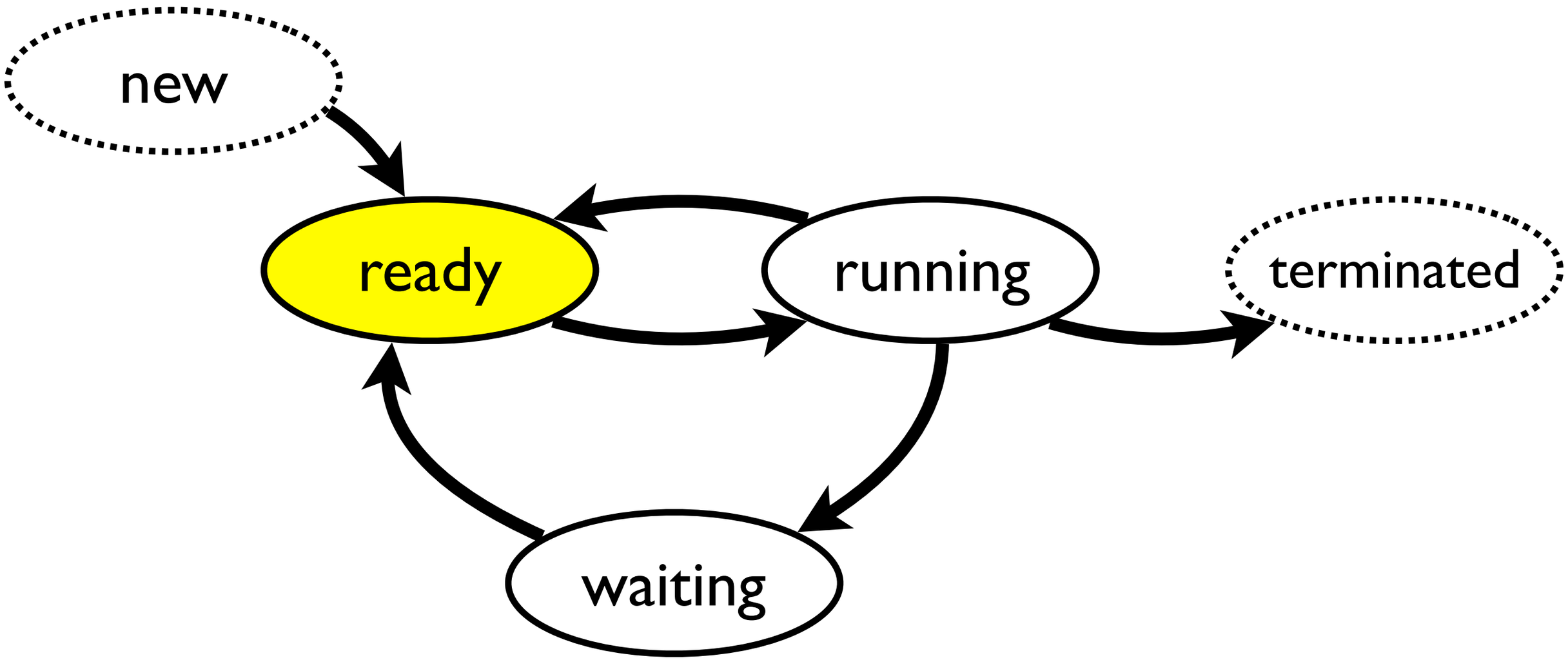
For new each process, a PCB and a process image must be allocated in memory (RAM).





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.



- An **executable** 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

Non-interactive process with no human user

Human user

Human user

Process A

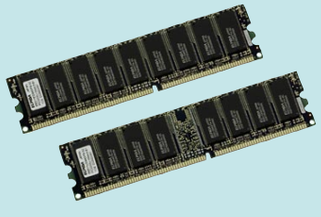
Process B

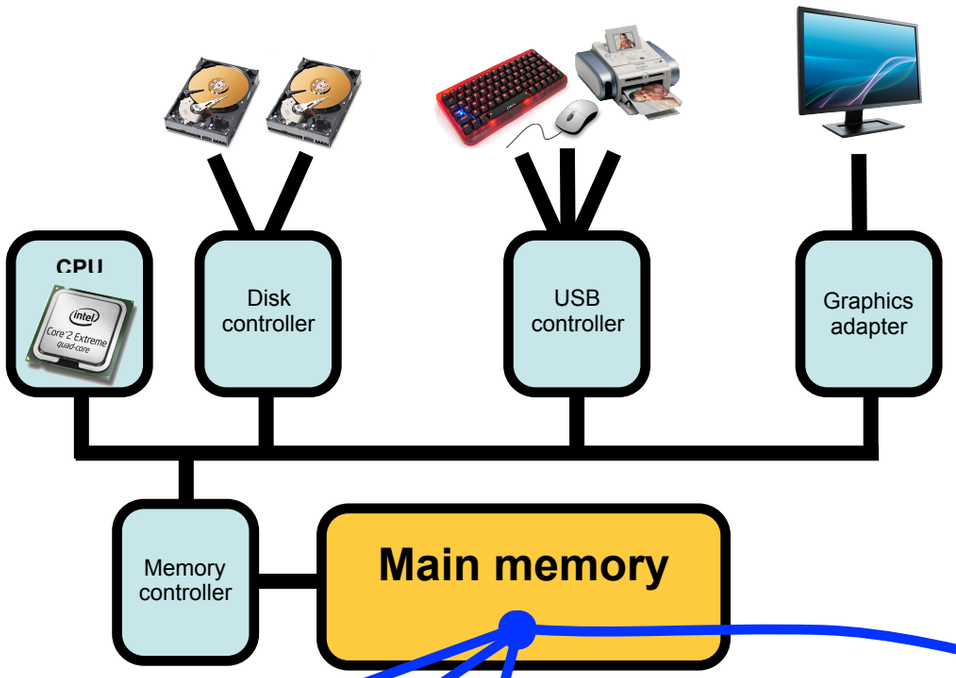
Process B

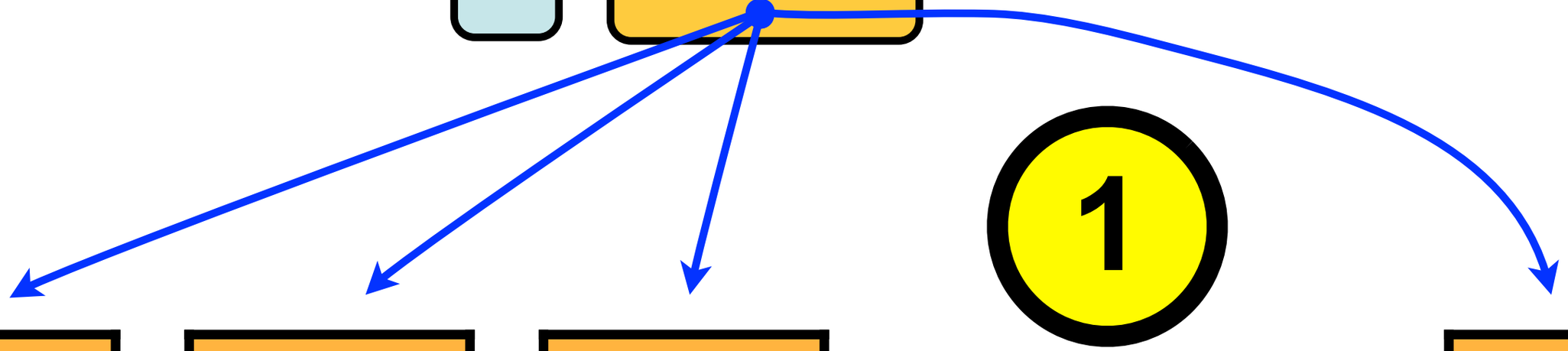
Process Z

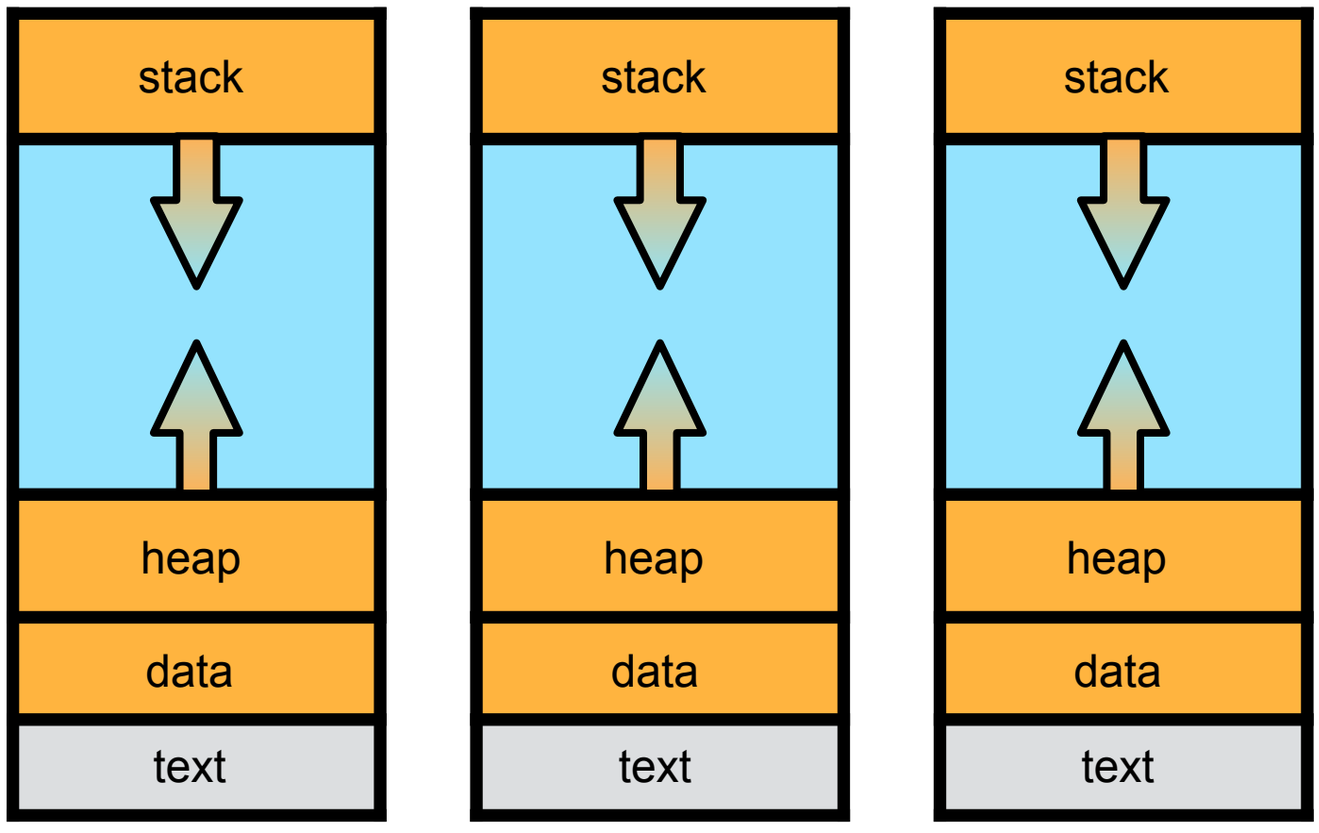
The operating systems **controls** the **hardware** and **coordinates** its use among the various application programs for the various user. An operating system provides an **environment for the execution of programs**.

Computer hardware

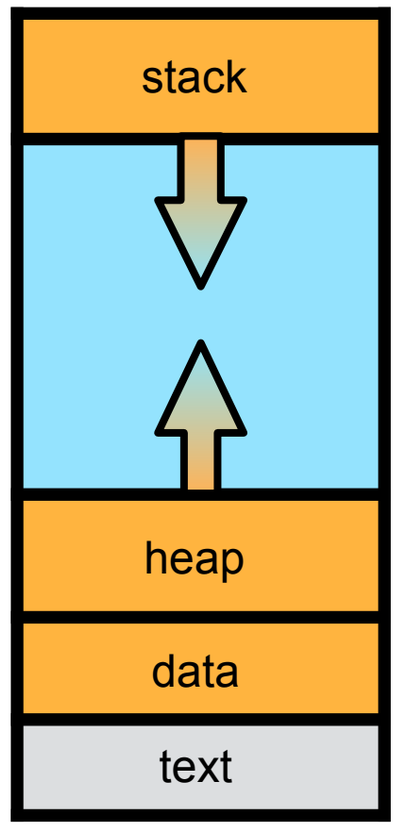








How can the **available memory** be allocated by the various processes?



Human user

Non-interactive process with no human user

Human user

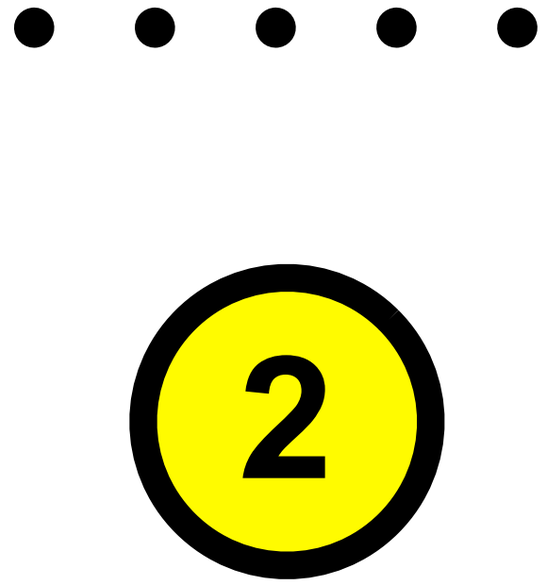
Human user

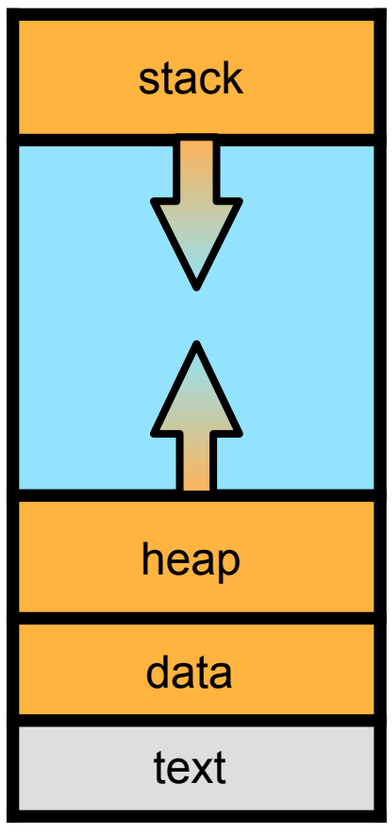
Process A

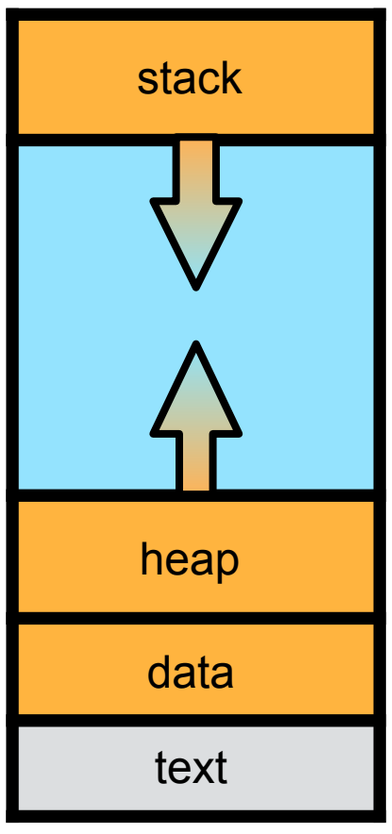
Process B

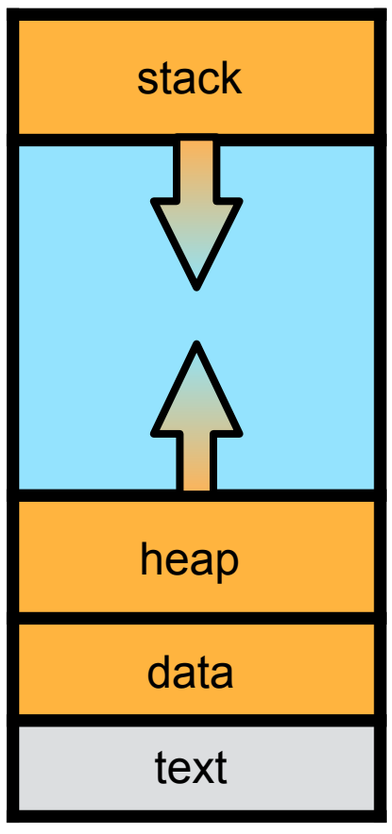
Process B

Process Z

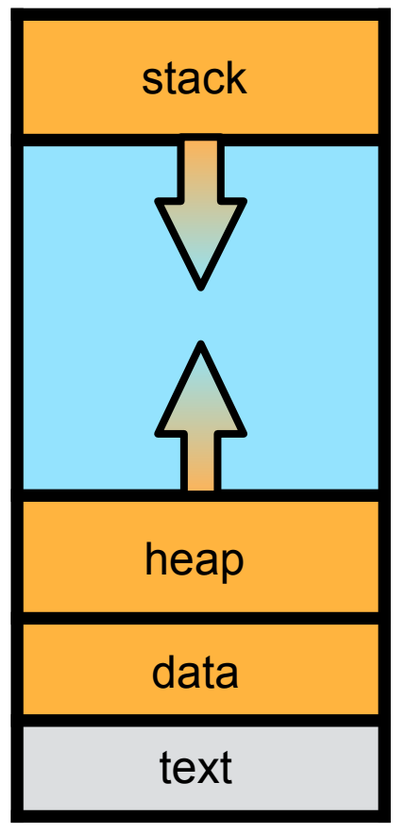








Must enforce **memory protection** between process images.



Single contiguous allocation

Single contiguous 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.

Source: https://en.wikipedia.org/wiki/Memory_management_(operating_systems)#Single_contiguous_allocation 2018-02-21

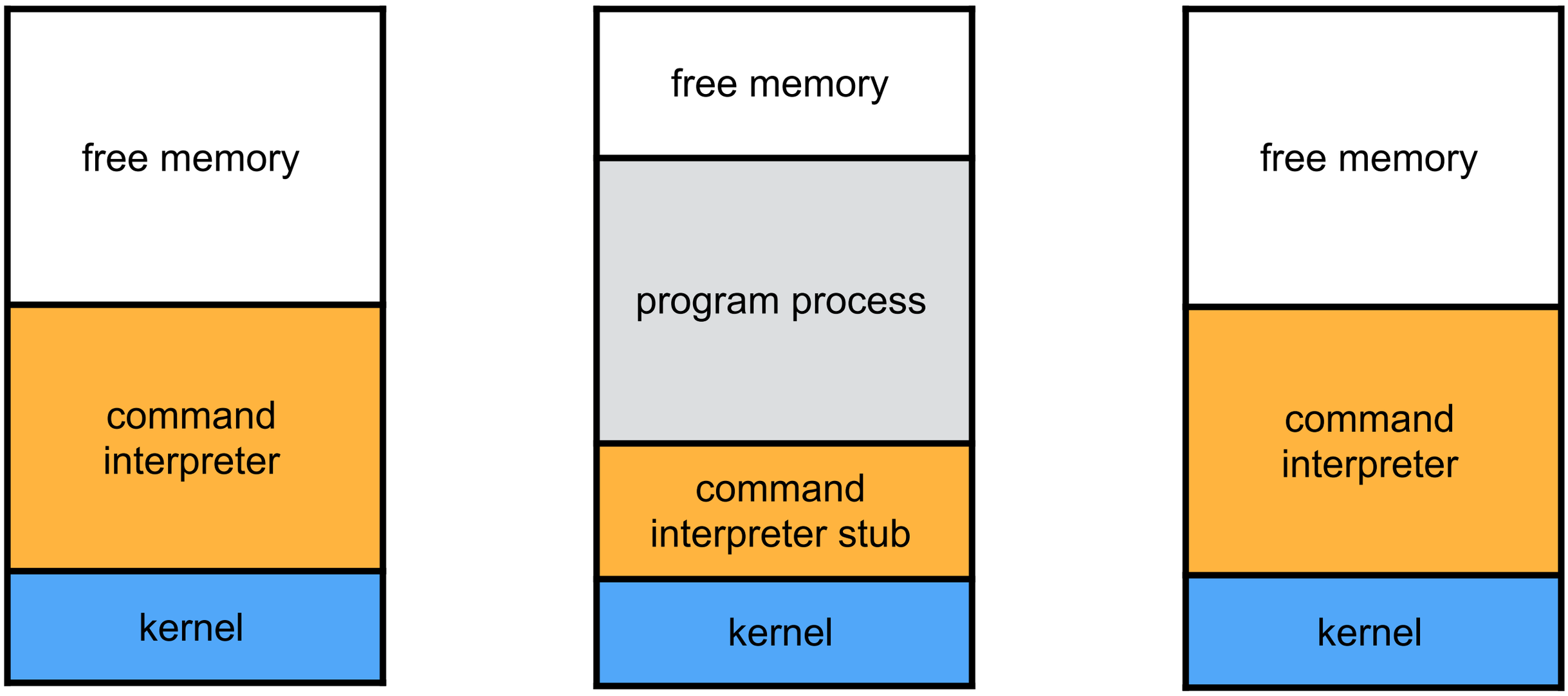
Single contiguous allocation

- **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.

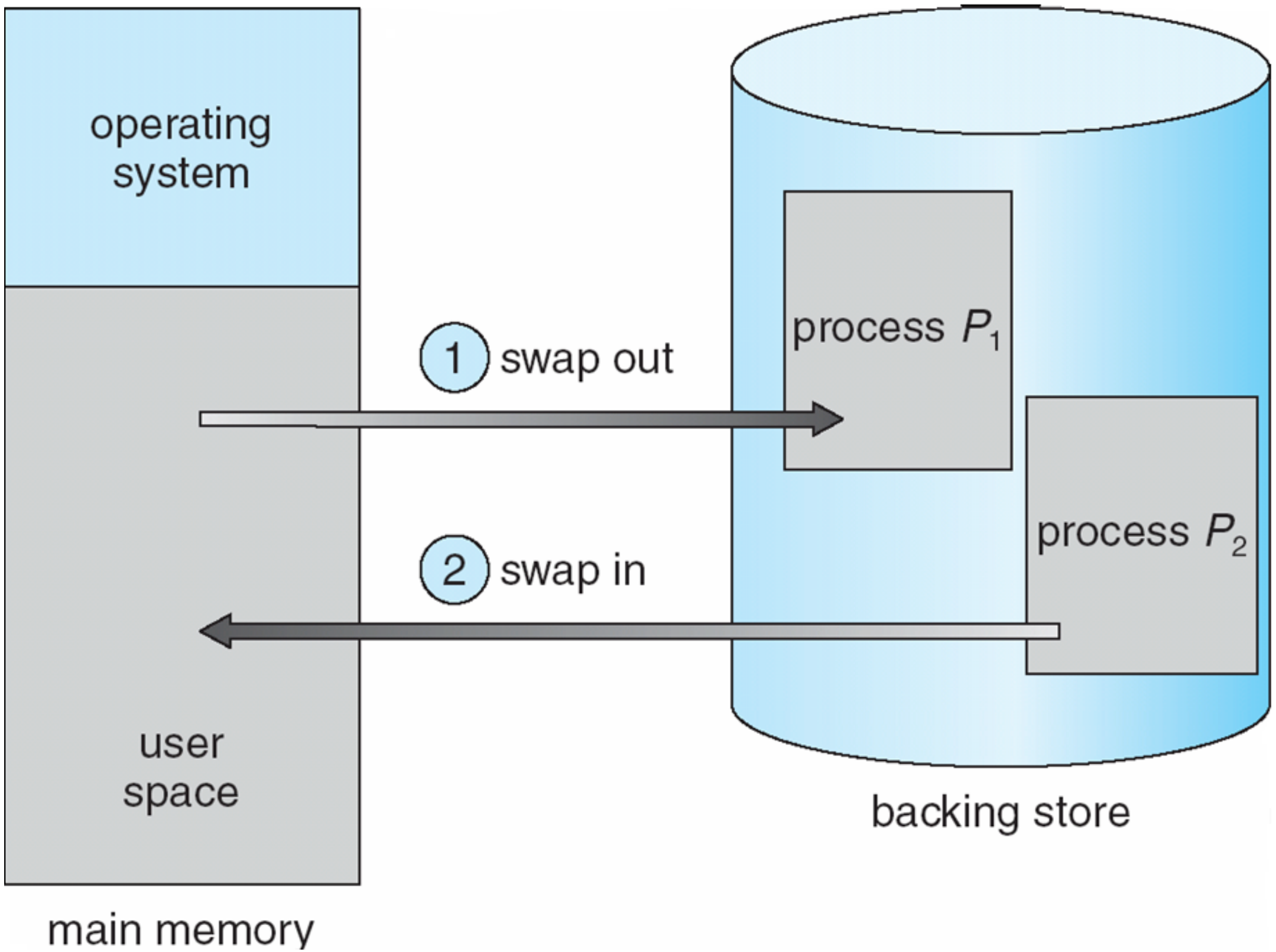
Source: https://en.wikipedia.org/wiki/MS-DOS 2018-02-21

- 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.


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

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.

Sources

http://www.webopedia.com/TERM/M/memory_resident.html 2012-02-16

http://en.wikipedia.org/wiki/Resident_set_size 2013-02-15

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).

Sources	http://www.webopedia.com/TERM/M/memory_resident.html	2012-02-16
	http://en.wikipedia.org/wiki/Resident_set_size	2013-02-15

Partitioned allocation

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

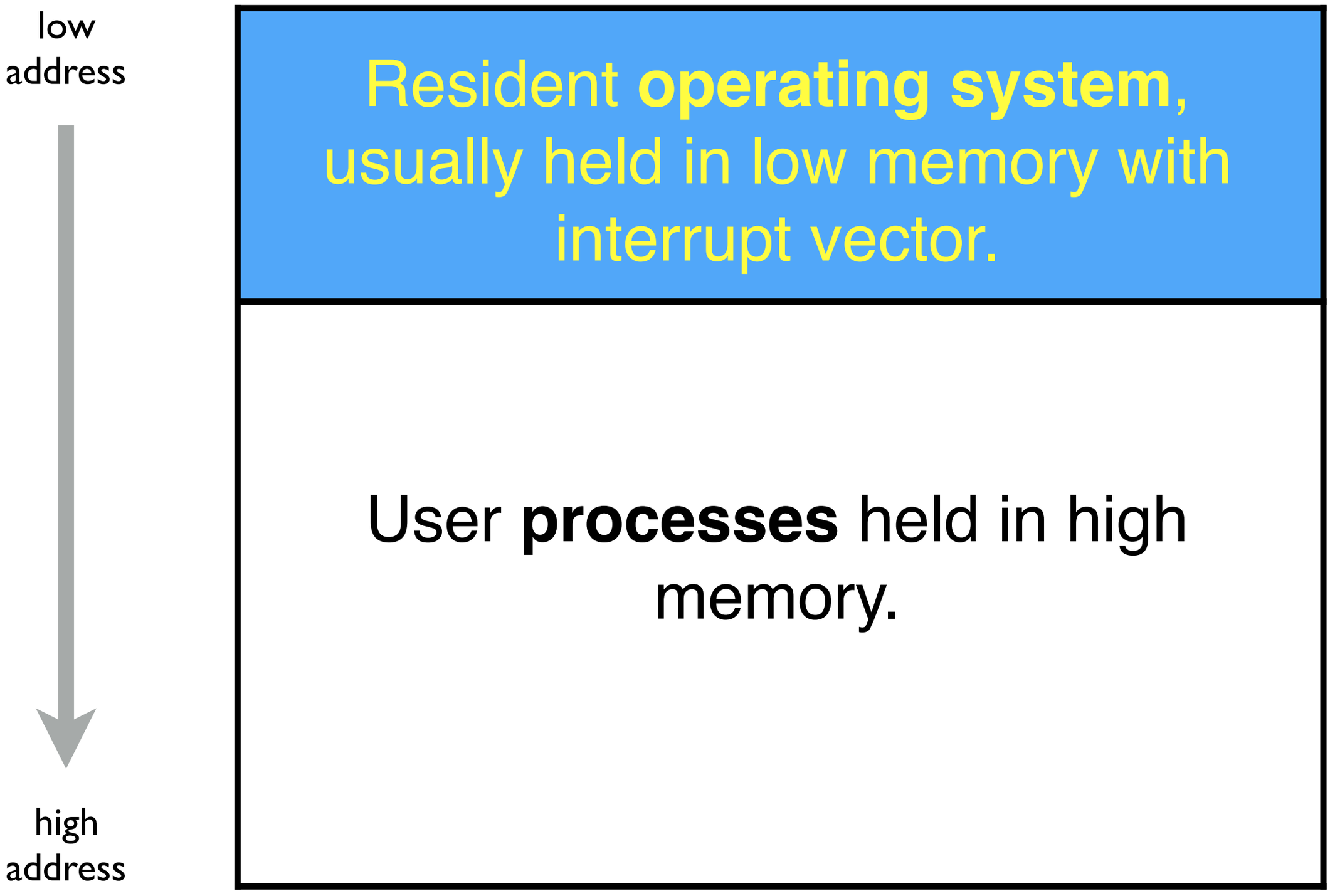
Source: https://en.wikipedia.org/wiki/Memory_management_(operating_systems)#Partitioned_allocation 2018-02-21

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.

Source: https://en.wikipedia.org/wiki/Memory_management_(operating_systems)#Partitioned_allocation 2018-02-21

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

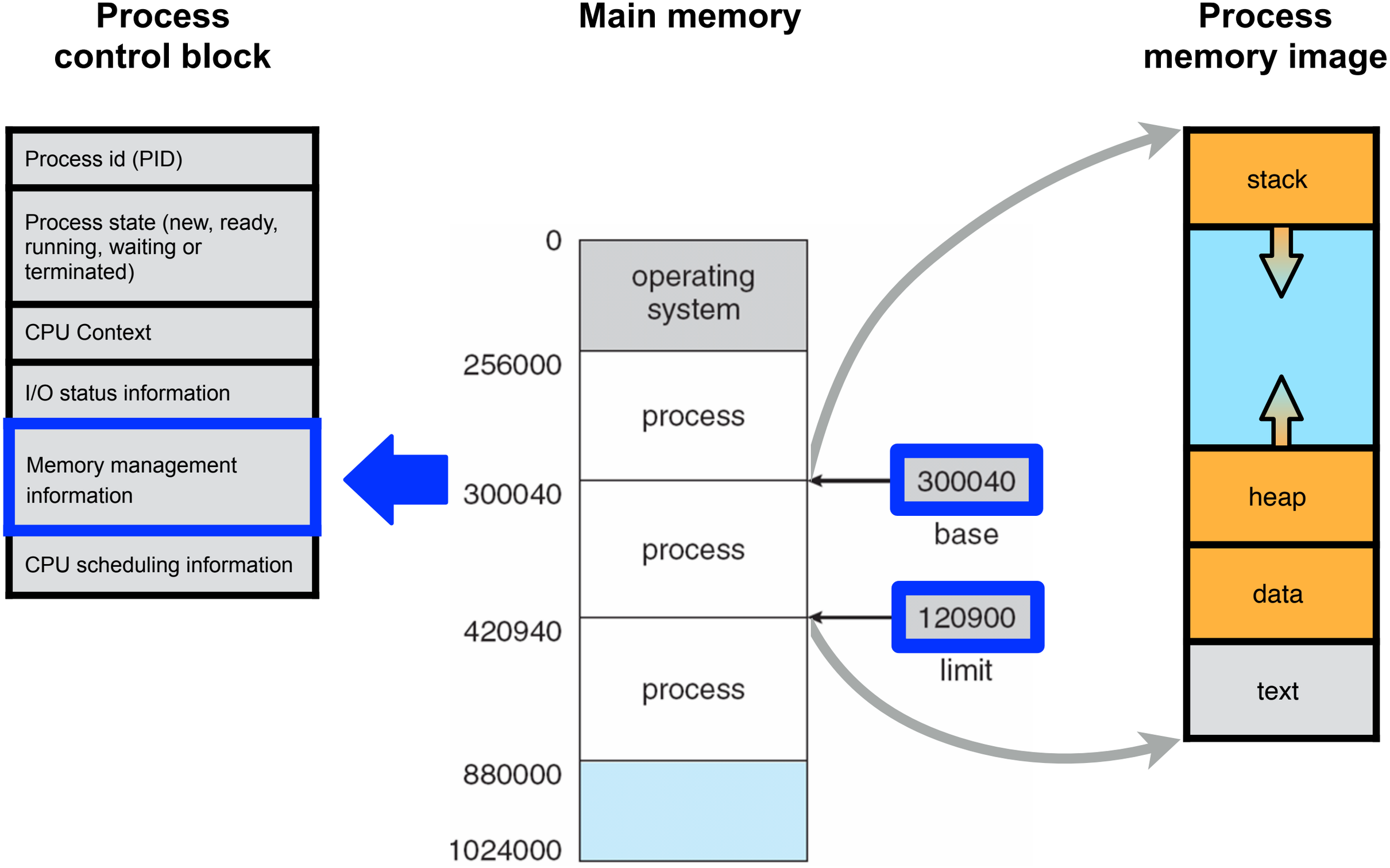


Logical address space

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

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

2018-02-21



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

Currently executing process

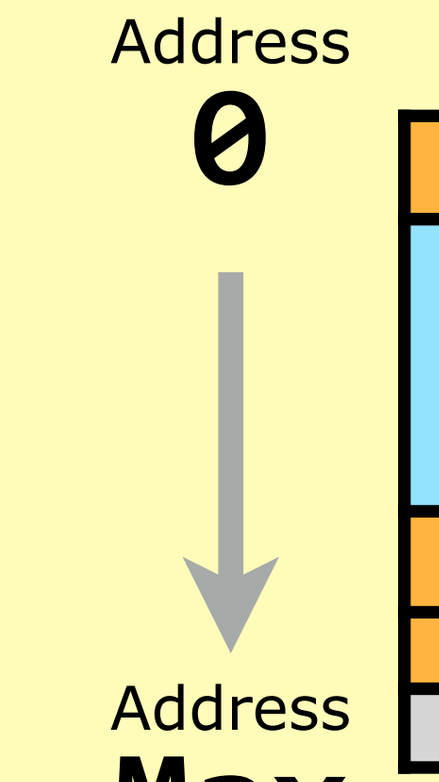
Logical memory addresses

Address

0

Address

Max



CPU

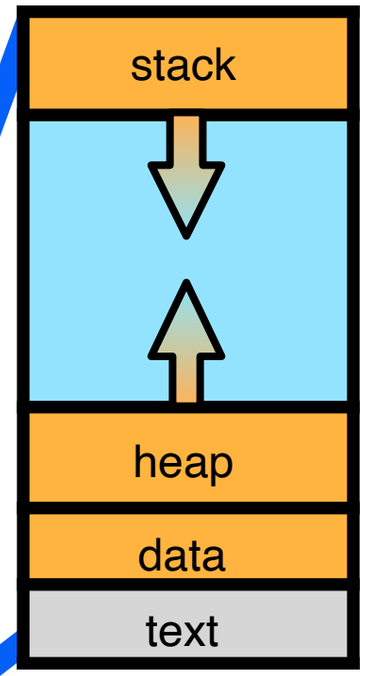
Main memory

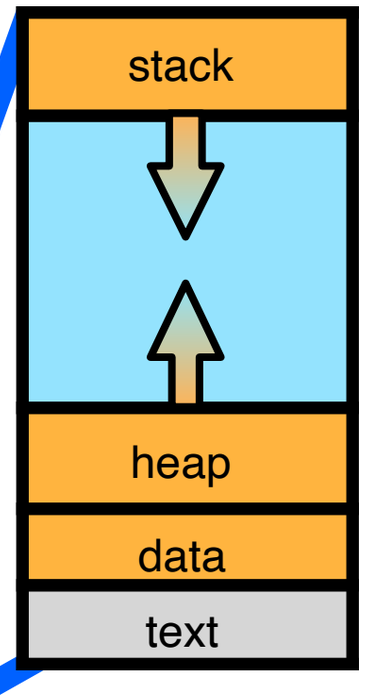
Physical memory addresses

P1

P2

P3



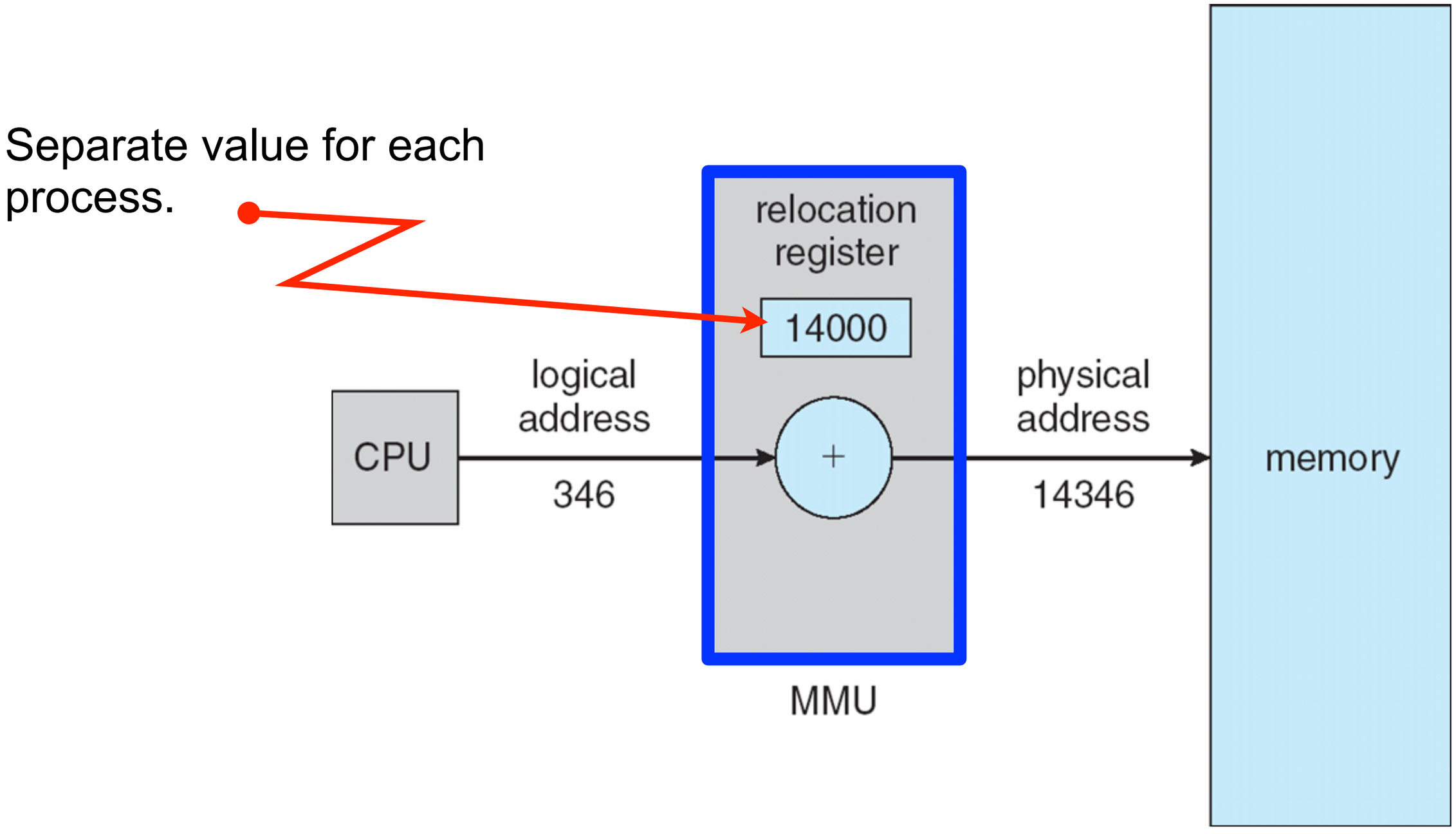


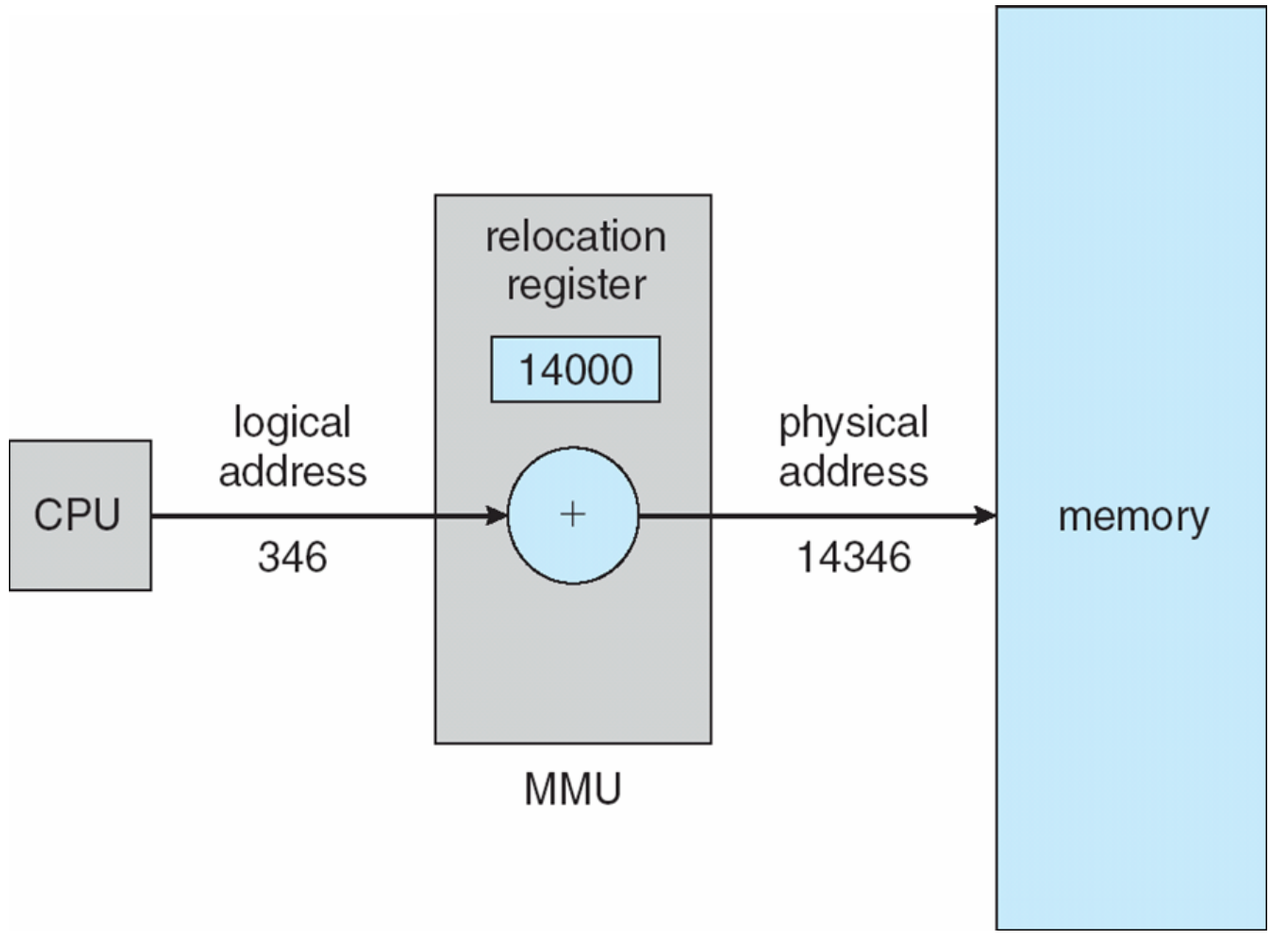
Memory management unit (MMU)

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**.

Source: https://en.wikipedia.org/wiki/Memory_management_unit 2018-02-21

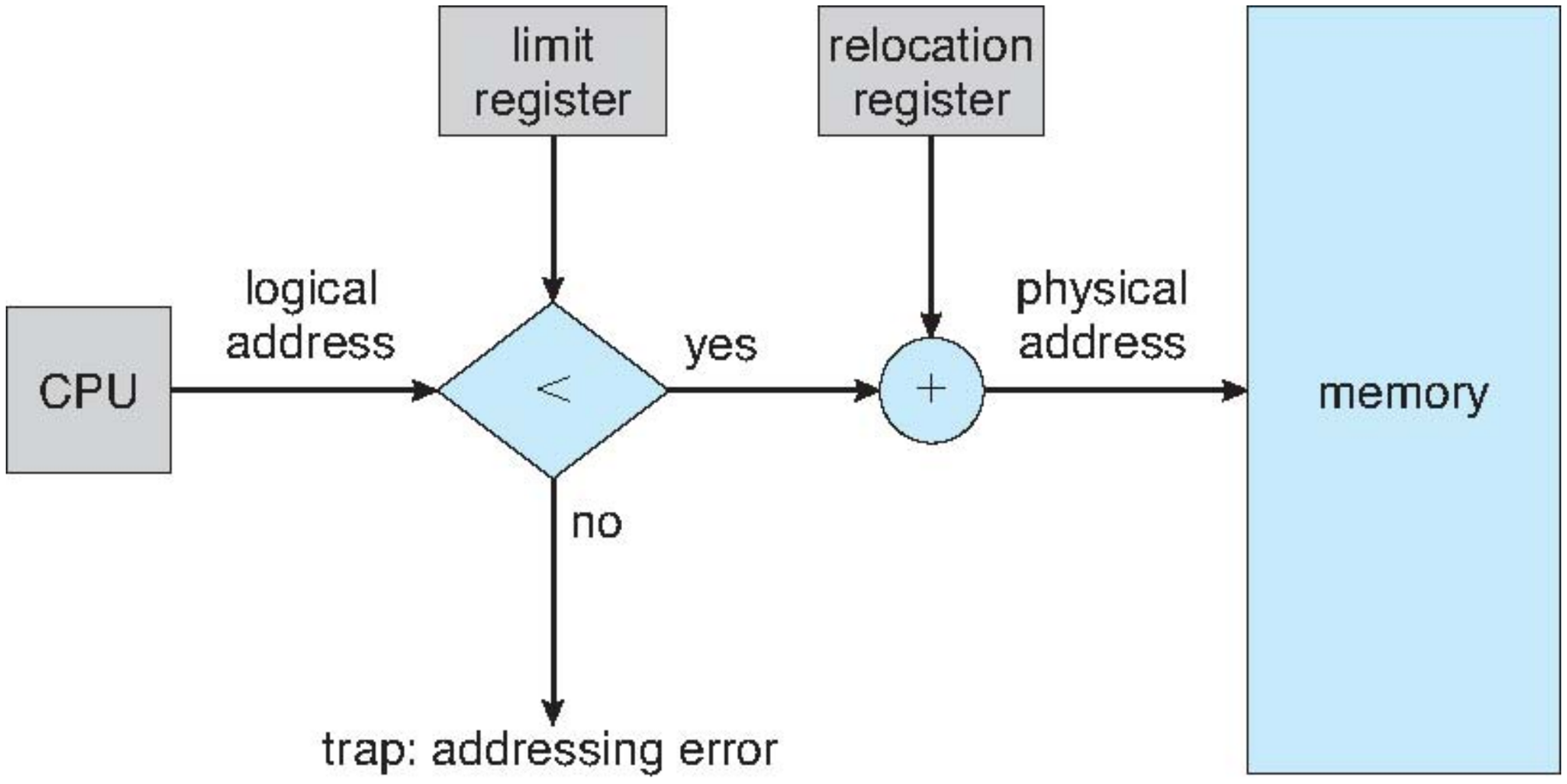
A simple MMU for partitioned contiguous allocation



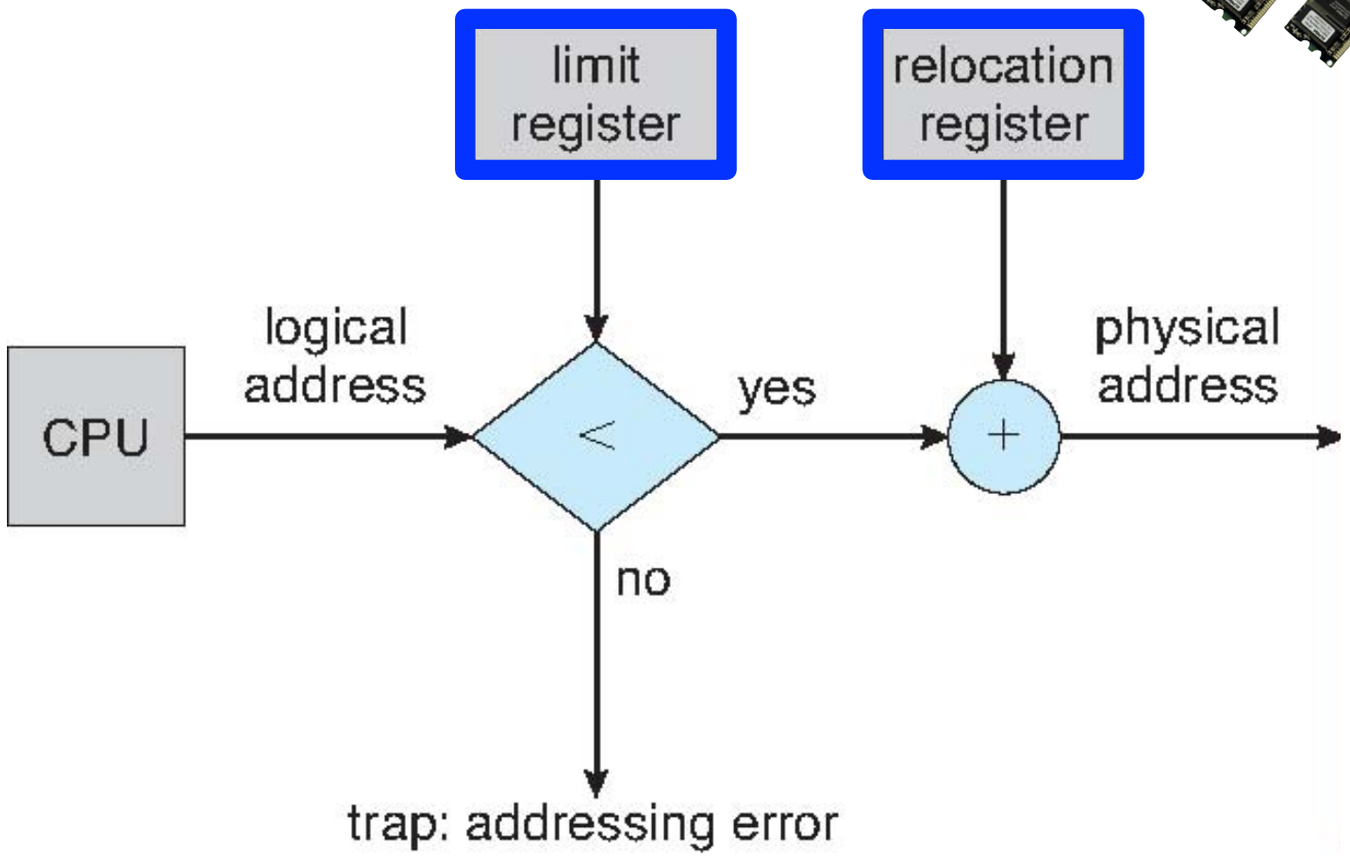


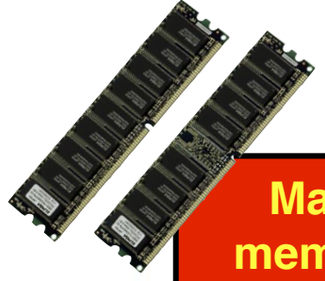
- Dynamic relocation using a **relocation register**.
- MMU **maps logical addresses to physical addresses.**
- The value in the relocation register is added to every address generated by a user process at the time it is sent to memory.
- The user program deals with logical addresses; it never sees the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

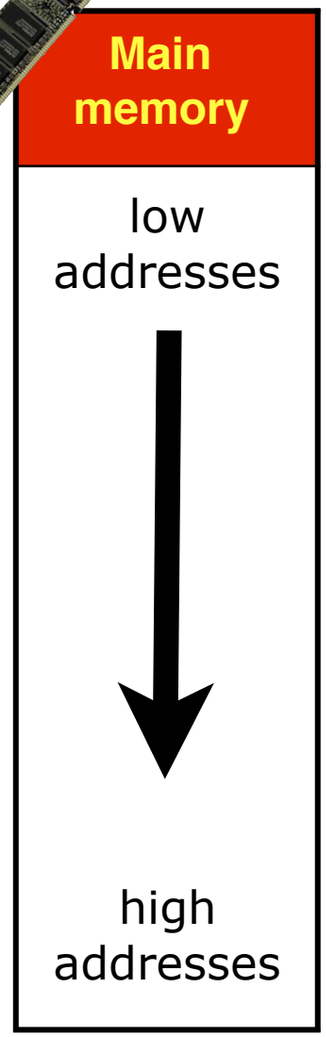
Must check if the address is within the boundaries of the process memory image.



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

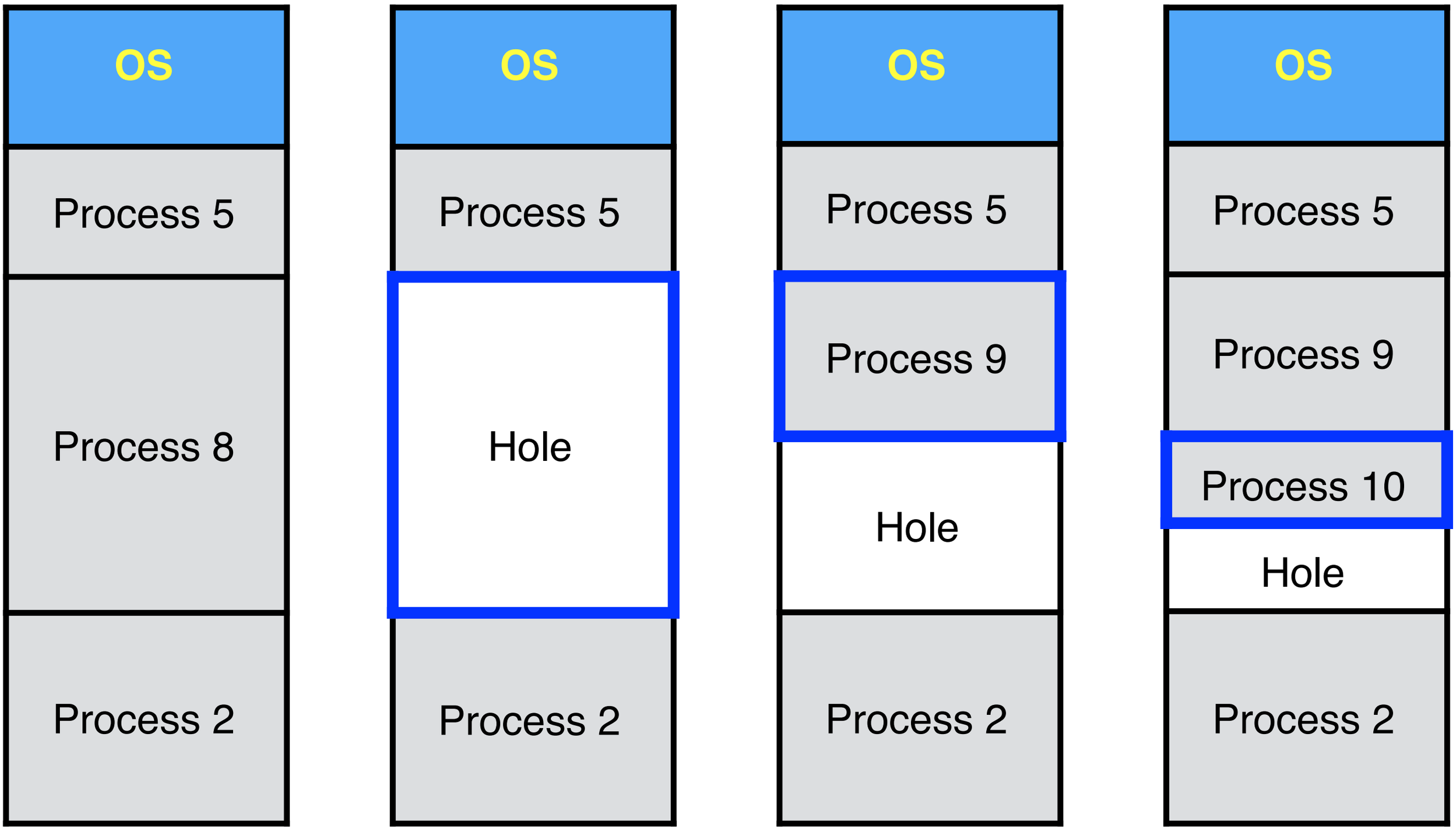
Dynamic memory management

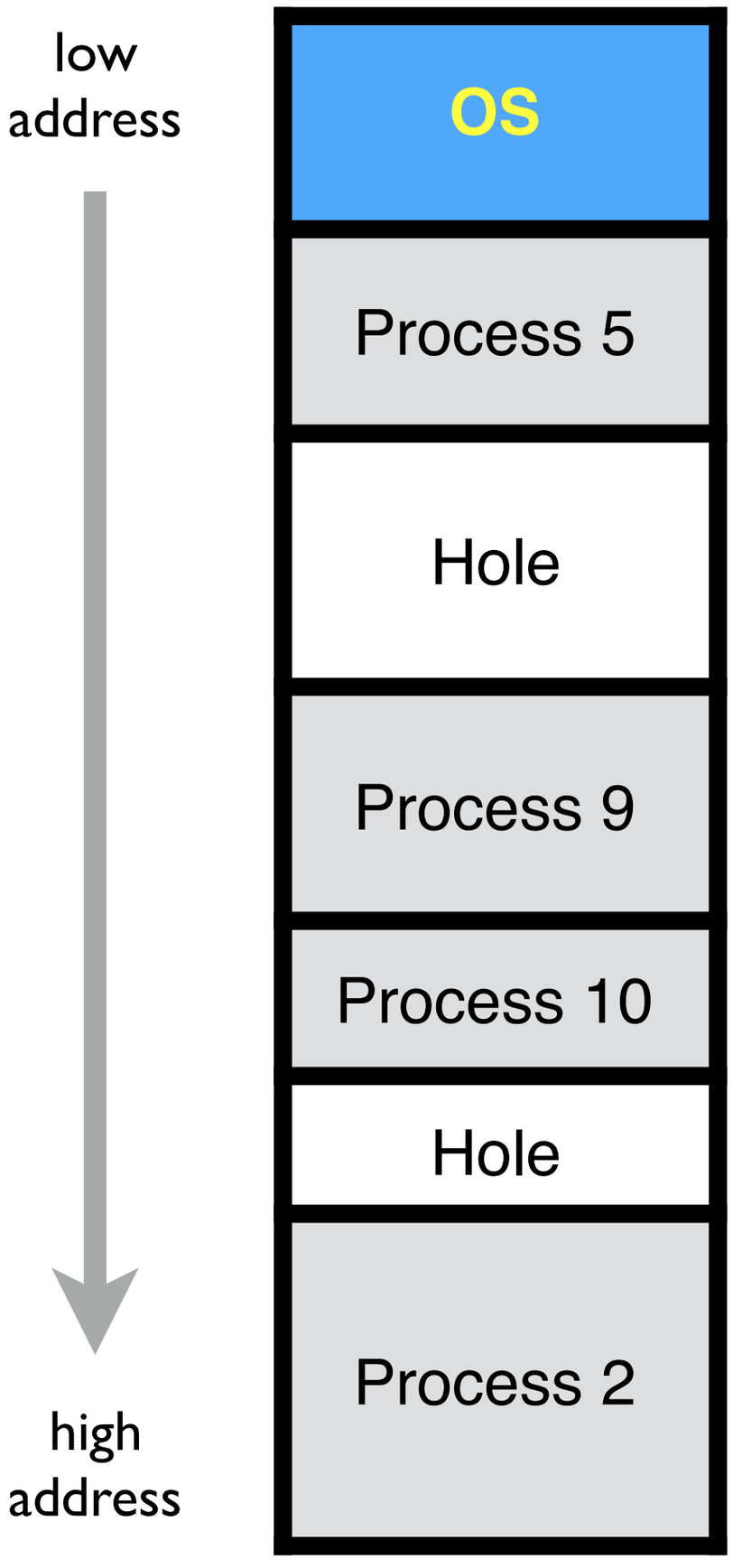
Three process occupies all available memory.

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.

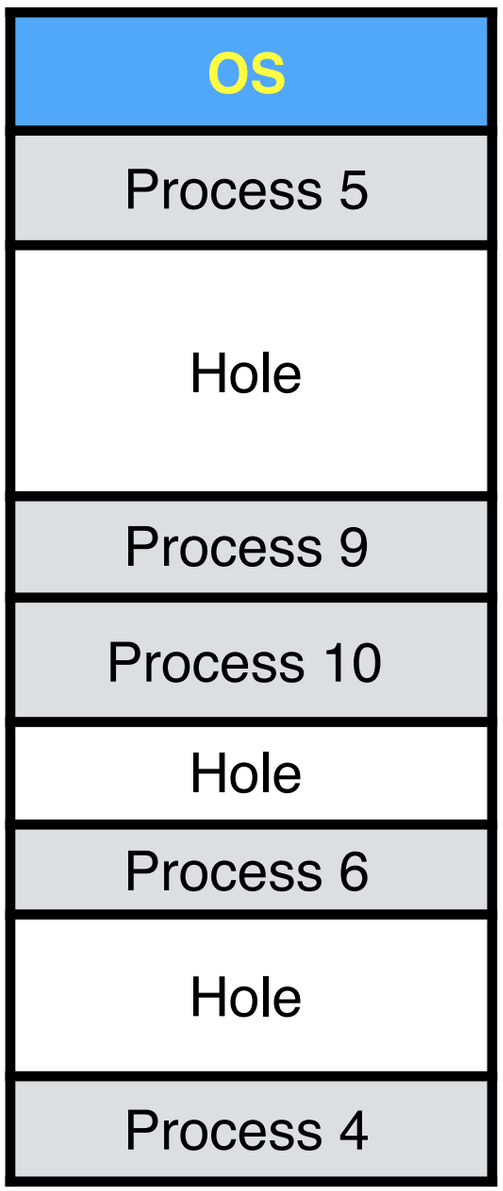
When a process arrives, it is **allocated memory from a hole large enough** to accommodate it.

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

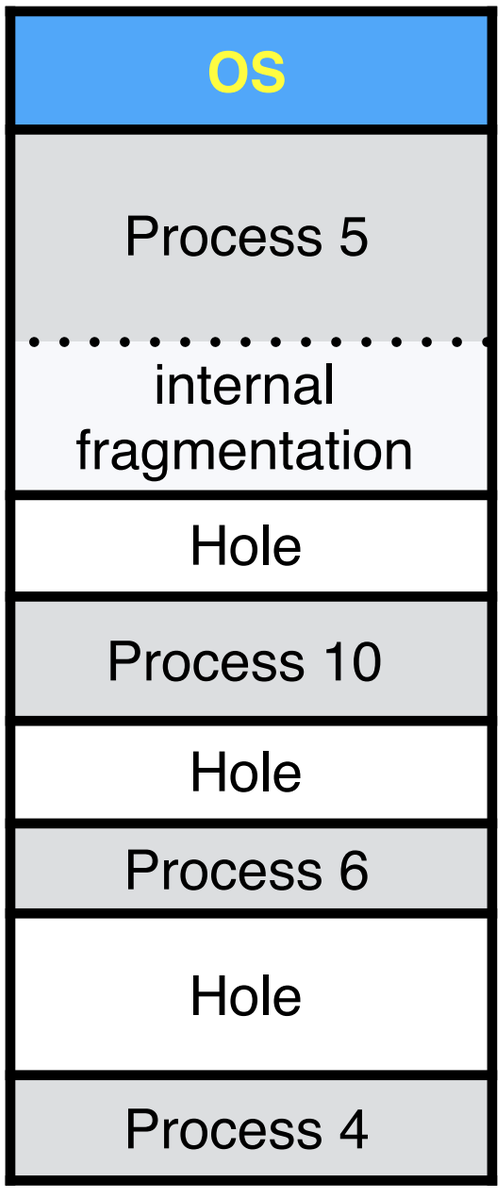
Simulations show that first-fit and best-fit performs better than worst-fit in terms of speed and storage utilization.

Fragmentation

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

Fragmentation

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.

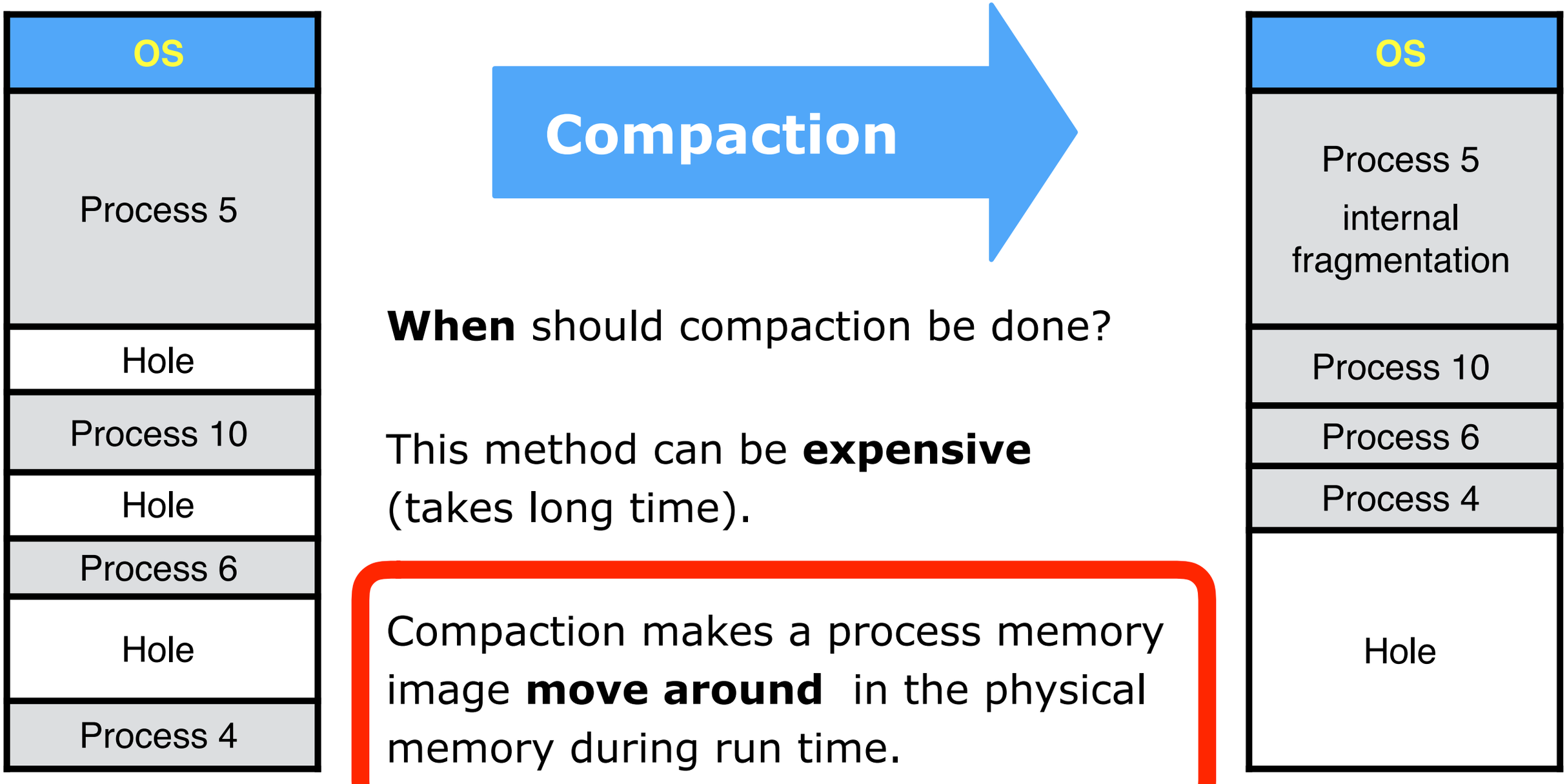
What to do about external fragmentation?

Compaction

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

Compaction

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



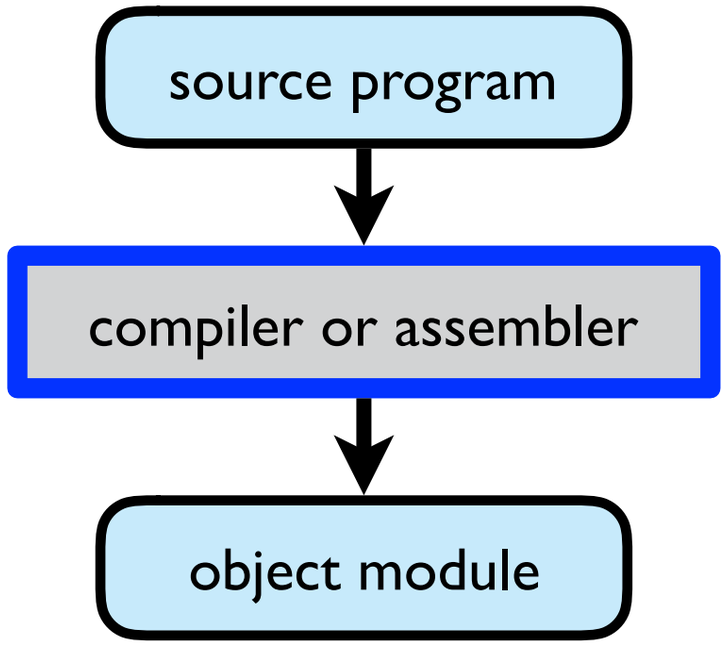
When should compaction be done?

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

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

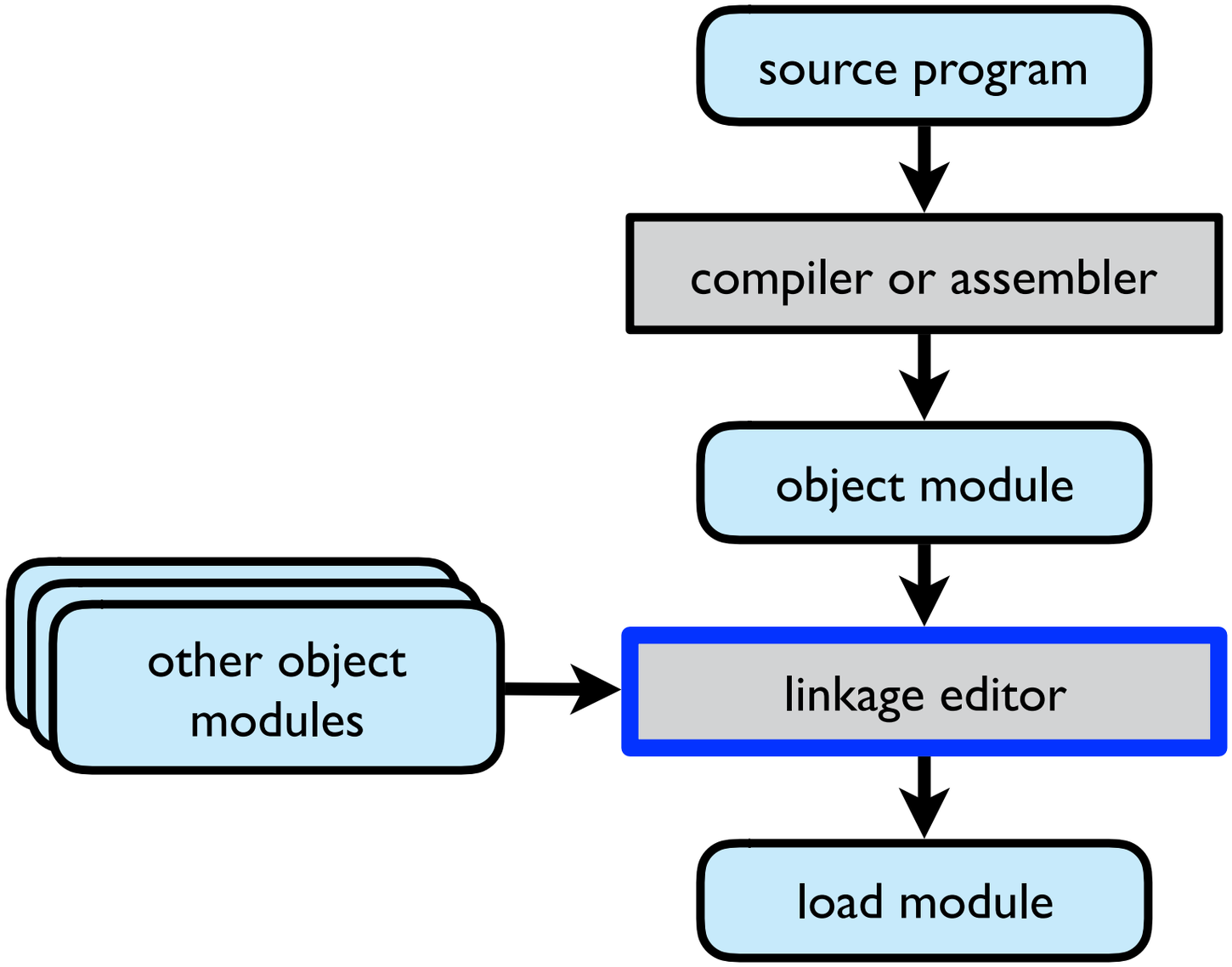
Address binding

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



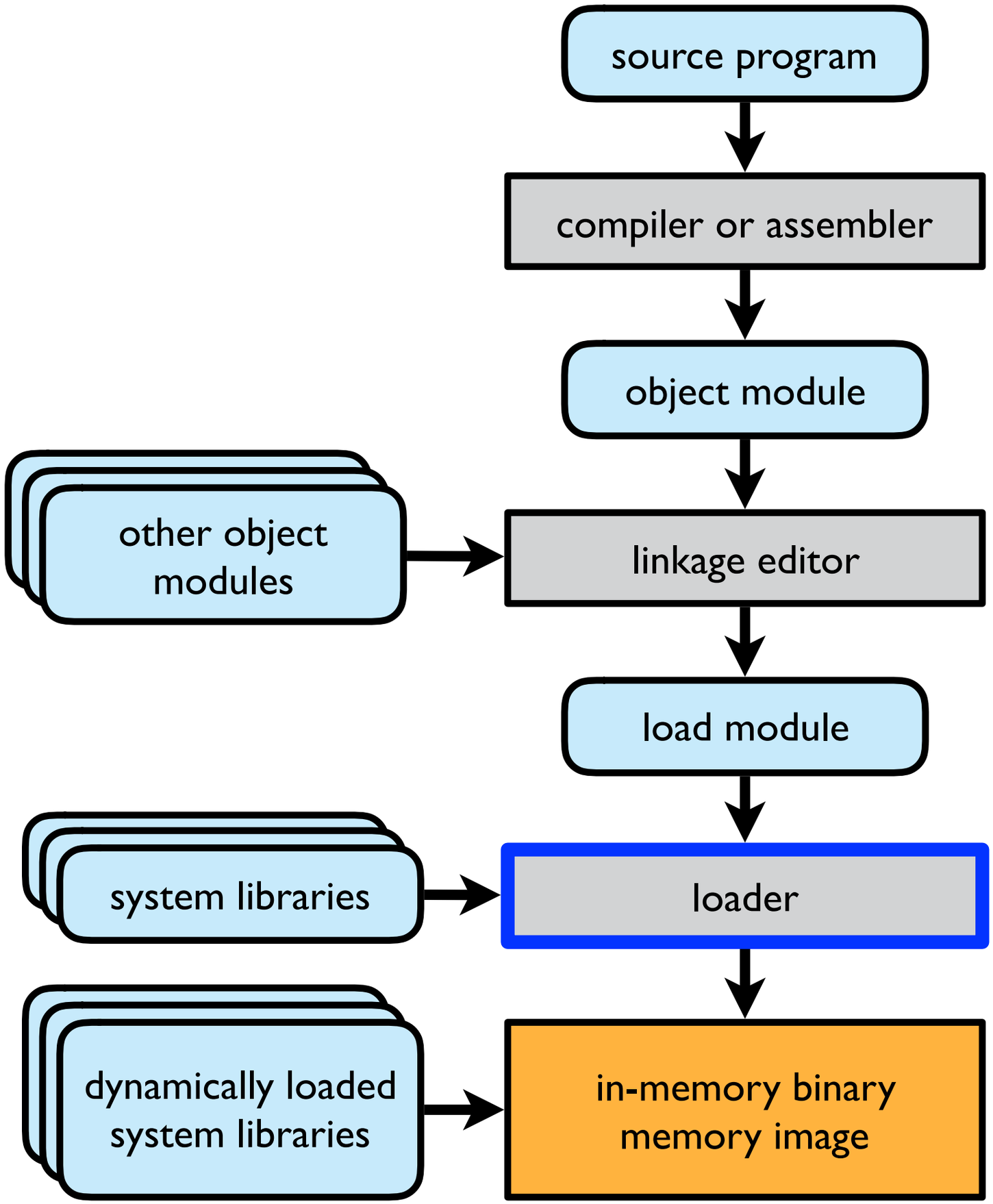
A **compiler** is computer software that transforms computer code written in one programming language (the source language) into another programming language (the target language).

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) 2018-02-21



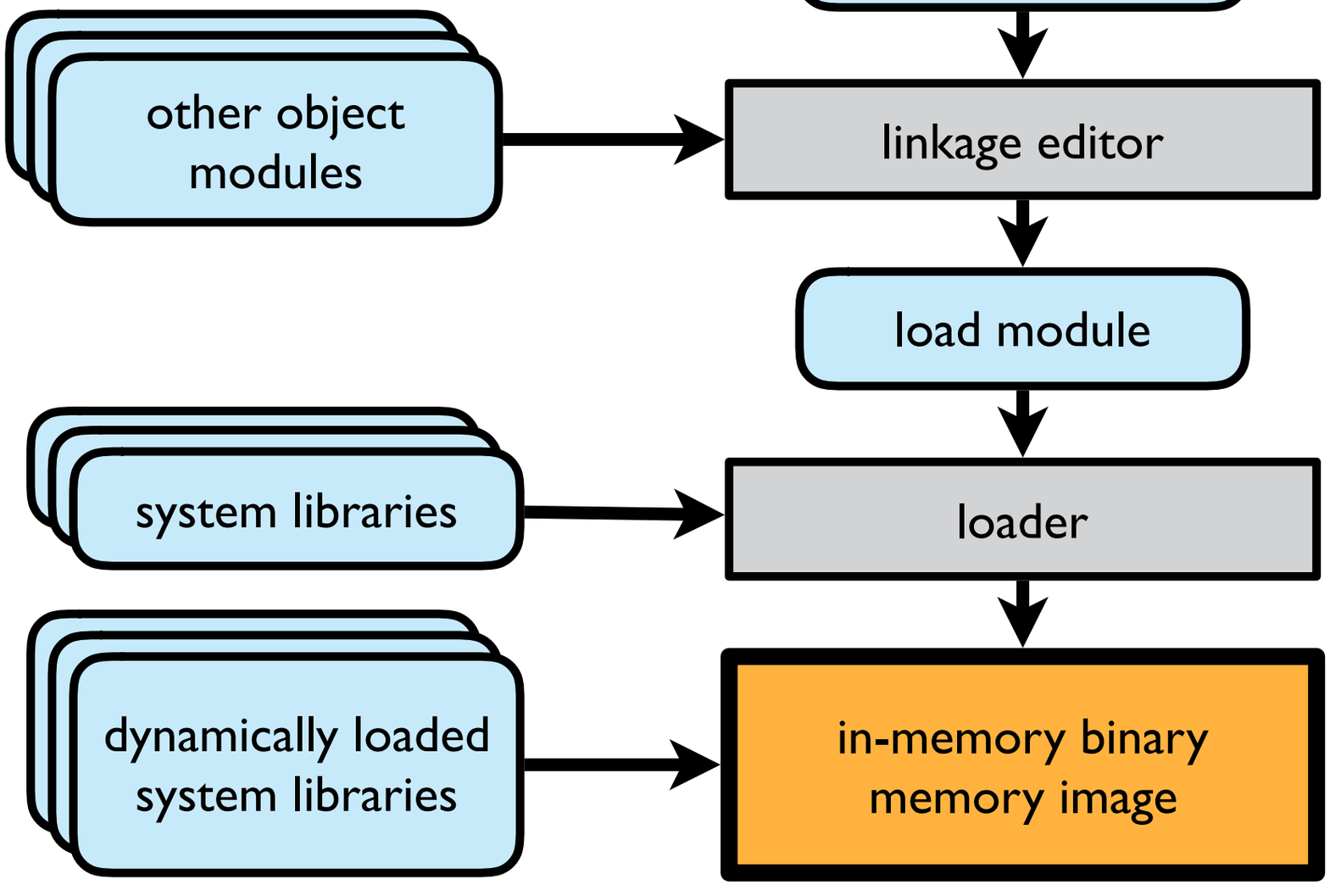
The **loader** reads the contents of the executable file containing the program instructions (and static data) into memory.

Once loading is complete, the operating system starts the program by passing control to the loaded program code.

Source: https://en.wikipedia.org/wiki/Loader_(computing) 2018-02-21

Binding of instructions and data to physical memory addresses

Address binding can take place at the following stages.

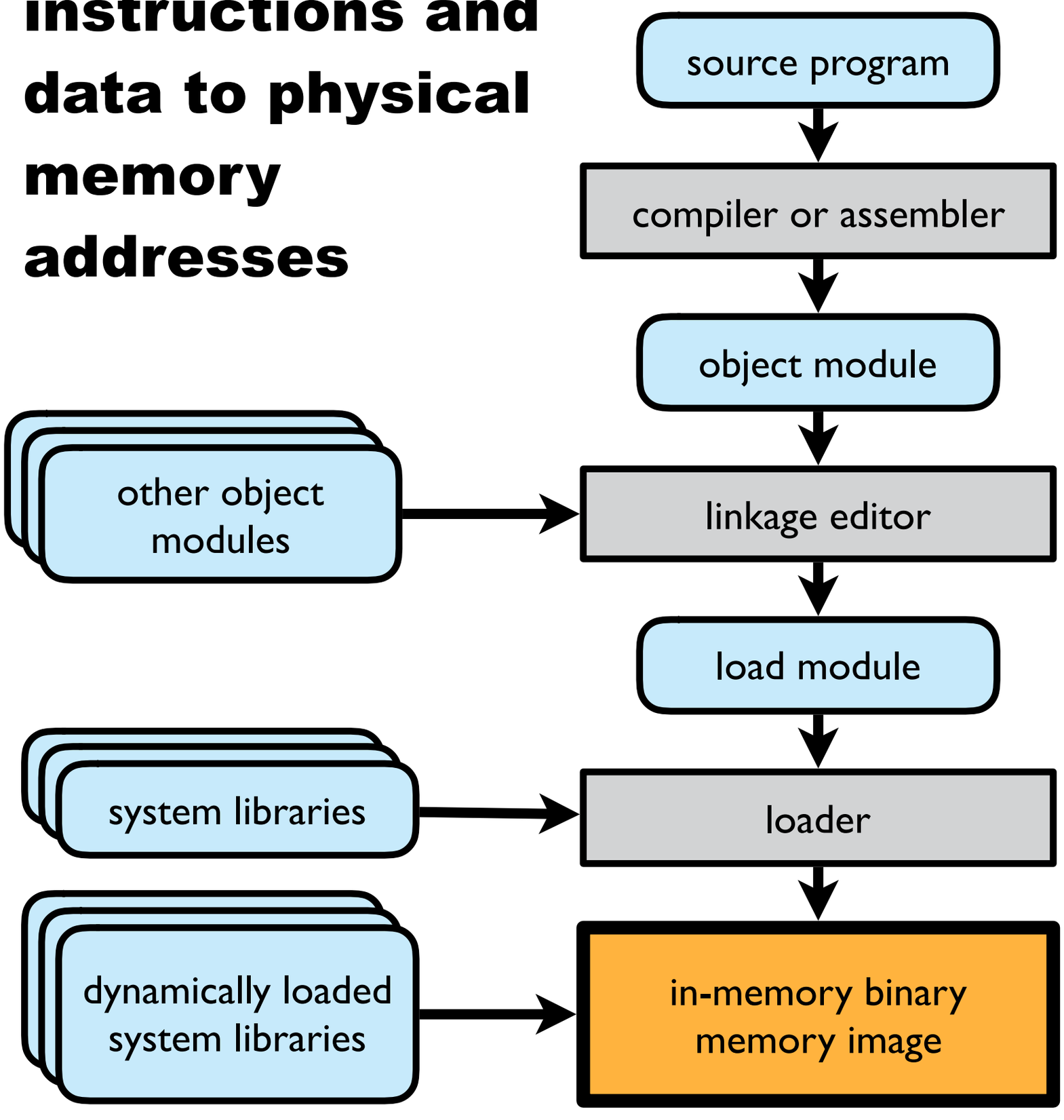


Compile time

Load time

Run time

Binding of instructions and data to physical memory addresses

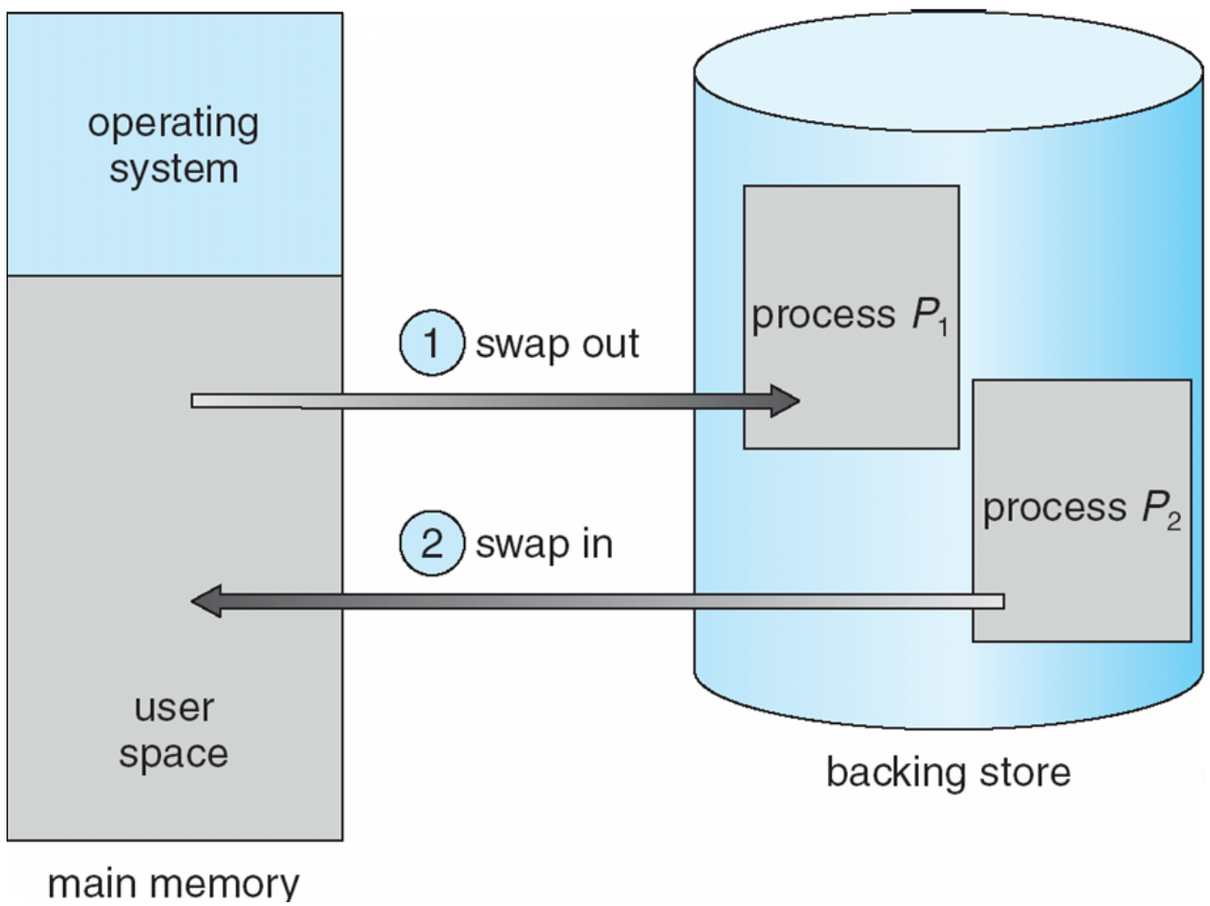


If **binding** delayed until **run time** the process can be moved during its execution.

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

Run time

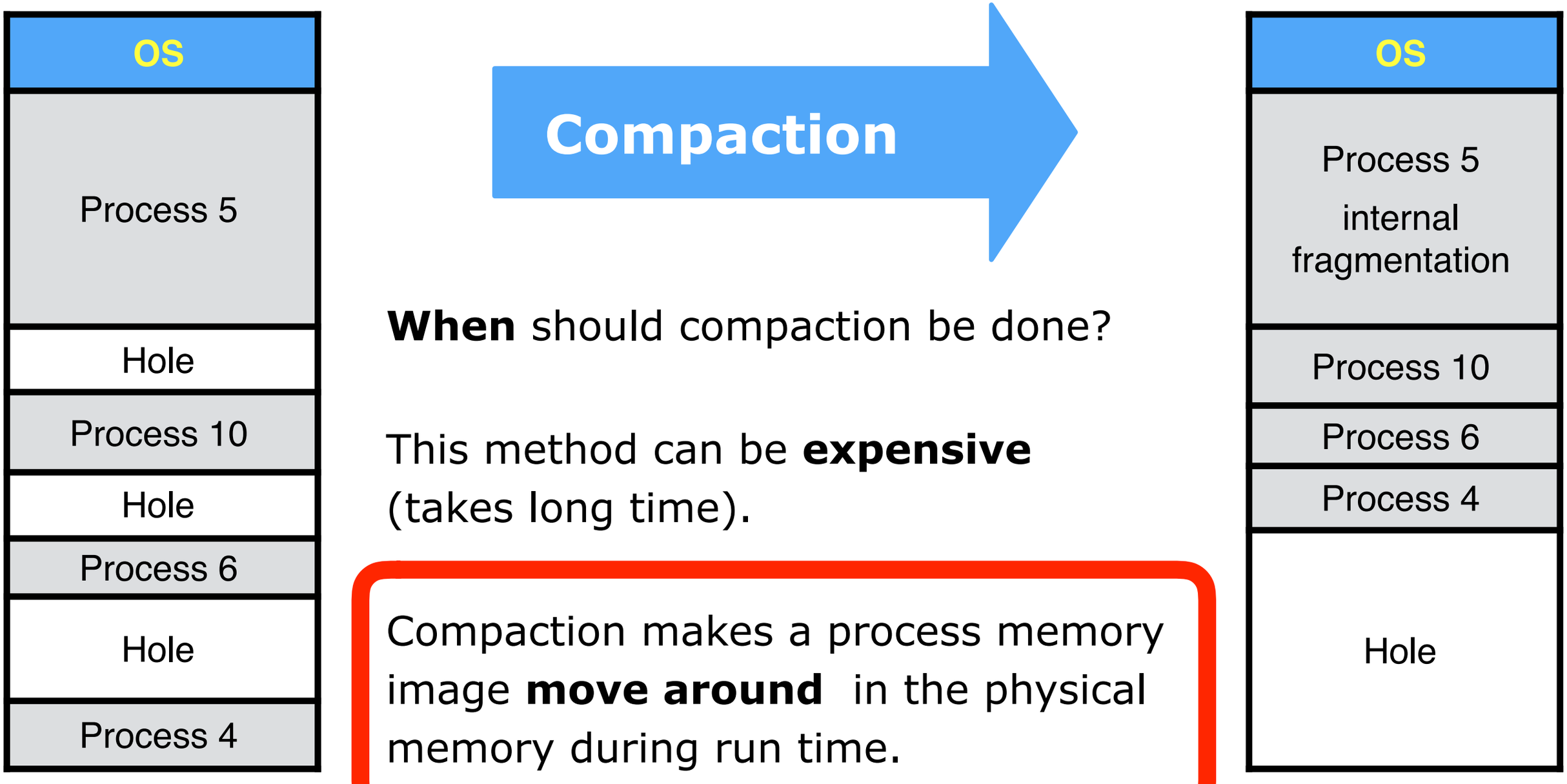
Swapping



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

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.

Paging

Paged memory management

Paged allocation divides the computer's primary memory into **fixed-size** units called page **frames**, and the program's logical (or 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.

Source: https://en.wikipedia.org/wiki/Memory_management_(operating_systems)#Paged_memory_management 2018-02-22

Assume we need to allocate a 4 byte logical memory space.

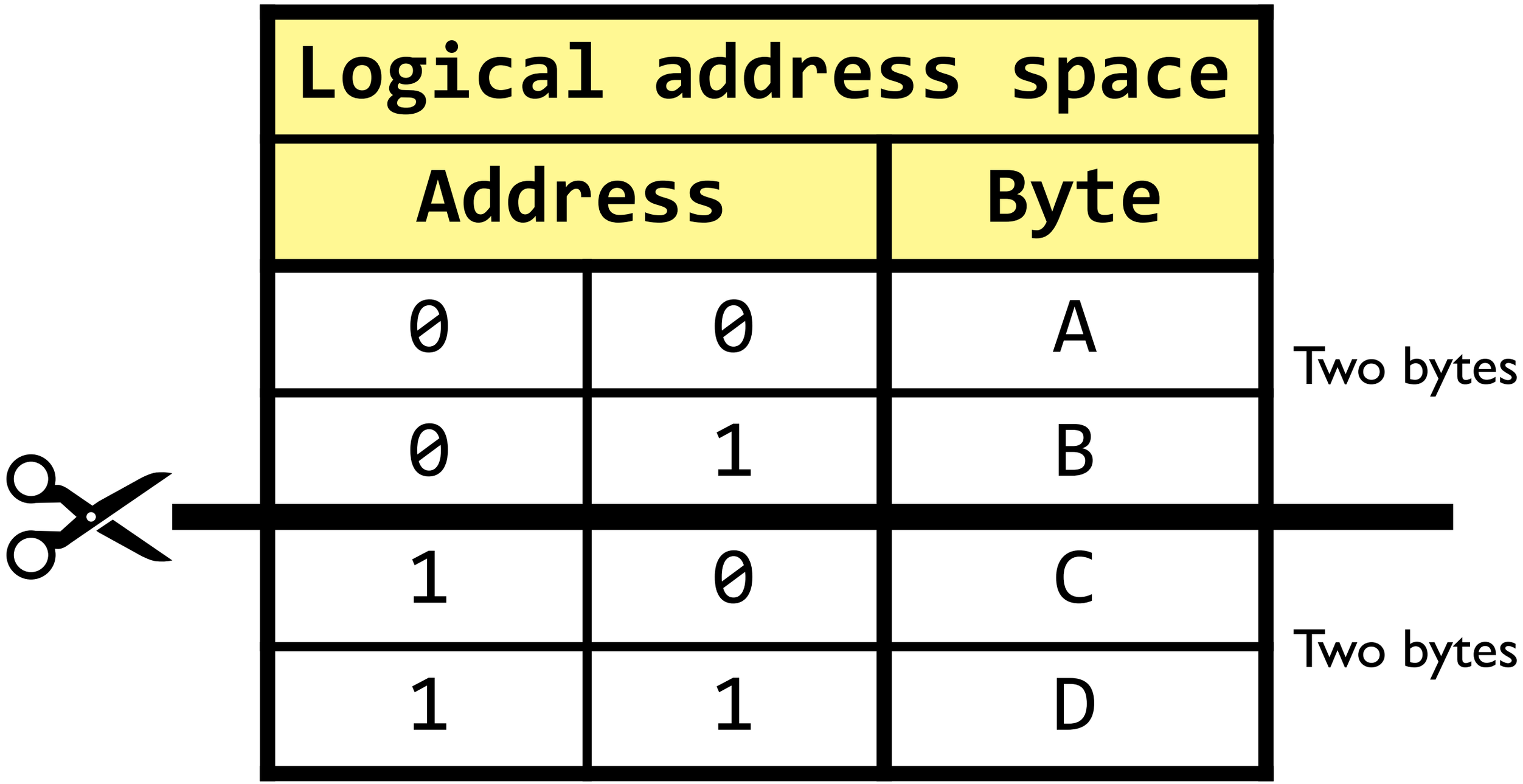
In the logical address space we store the values A, B, C, D.

We address each byte with a two bit binary address.

Logical address space		
Address		Byte
0	0	A
0	1	B
1	0	C
1	1	D

Due to external fragmentation, we might not be able to find four consecutive (contiguous) bytes of physical memory. But **what if** we can find holes of two consecutive bytes in the physical memory?

If we cut the logical address space in two halves, each half will be two bytes.



Let's call each half a **page**.

Now the **first bit** of the logical binary address uniquely identifies the **page**.

The **second bit** of the logical address numbers the bytes within each page, we call this the page **offset**.

Logical address space			
Address		Byte	
Page	Offset		
0	0	A	Page 0
0	1	B	
1	0	C	Page 1
1	1	D	

Assume we have a 8 byte physical memory. We need 3 bits to address the physical memory ($_2\log(8) = 3$). The lowest address = 000 and the highest address 111.

Divide the physical memory into **frames** of the same size (two bytes) as the pages. Number of frames = 8/2 = 4. We number the frames in binary: 00, 01, 10, 11.

The two most significant bits will be the **frame** number.

Physical address space			
Address			Byte
Frame		Offset	
0	0	0	W
0	0	1	X
0	1	0	
0	1	1	
1	0	0	Y
1	0	1	Z
1	1	0	
1	1	1	

The least significant bit will be the frame **offset**.

Frame 0	Already allocated to another process.
Frame 1	Free hole.
Frame 2	Already allocated to another process.
Frame 3	Free hole.

We use a **page table** to map a logical page number to an empty physical frame.

Logical address space		
Address		Byte
Page	Offset	
0	0	A
0	1	B
1	0	C
1	1	D

Page table	
Page	Frame
0	?
1	?

Physical address space			
Address			Byte
Frame		Offset	
0	0	0	W
0	0	1	X
0	1	0	
0	1	1	
1	0	0	Y
1	0	1	Z
1	1	0	
1	1	1	

In this example, page 0 is placed in frame 01 and page 1 is placed in frame 11.

Logical address space		
Address		Byte
Page	Offset	
0	0	A
0	1	B
1	0	C
1	1	D

Page table	
Page	Frame
0	01
1	11

Physical address space			
Address			Byte
Frame		Offset	
0	0	0	W
0	0	1	X
0	1	0	A
0	1	1	B
1	0	0	Y
1	0	1	Z
1	1	0	C
1	1	1	D

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

If we double the size of the logical address space from 4 bytes to 8 bytes, we could keep the 2 byte page/frame size and double the number of pages from 2 to 4 (A) or keep using 2 pages and double the page/frame size from 2 to 4 byes (B).

A

Logical address space		
Address		
Page		Offset
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

The two most significant bits of the logical address identifies the page number and the least significant bit is the offset.

B

Logical address space		
Address		
Page	Offset	
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

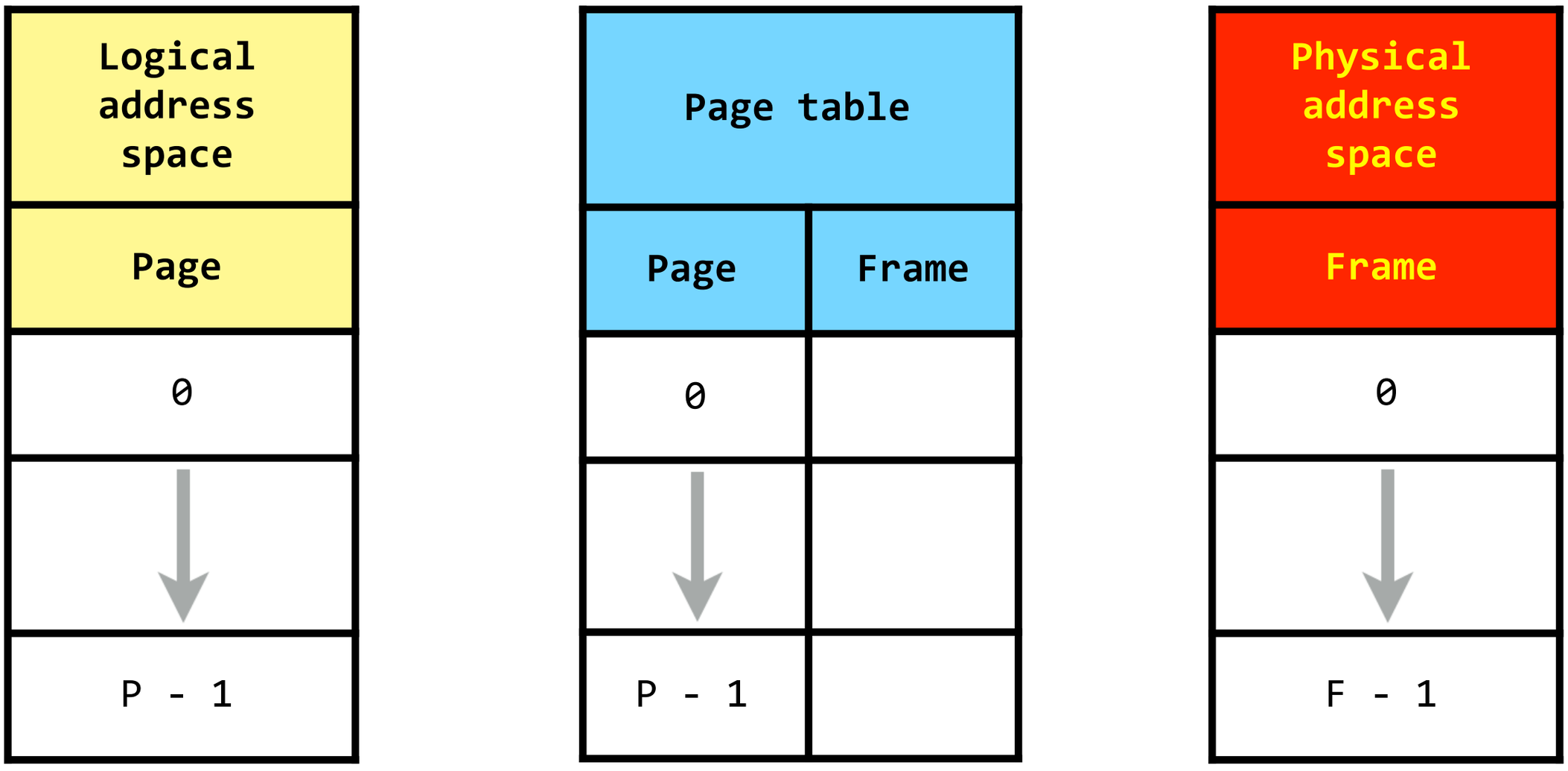
The most significant bit of the logical address identifies the page number and the two least significant bits is the offset.

We can generalize this method for arbitrary sized logical address spaces and arbitrary sized physical address spaces.

Let **P** = number of pages and **F** = number of frames.

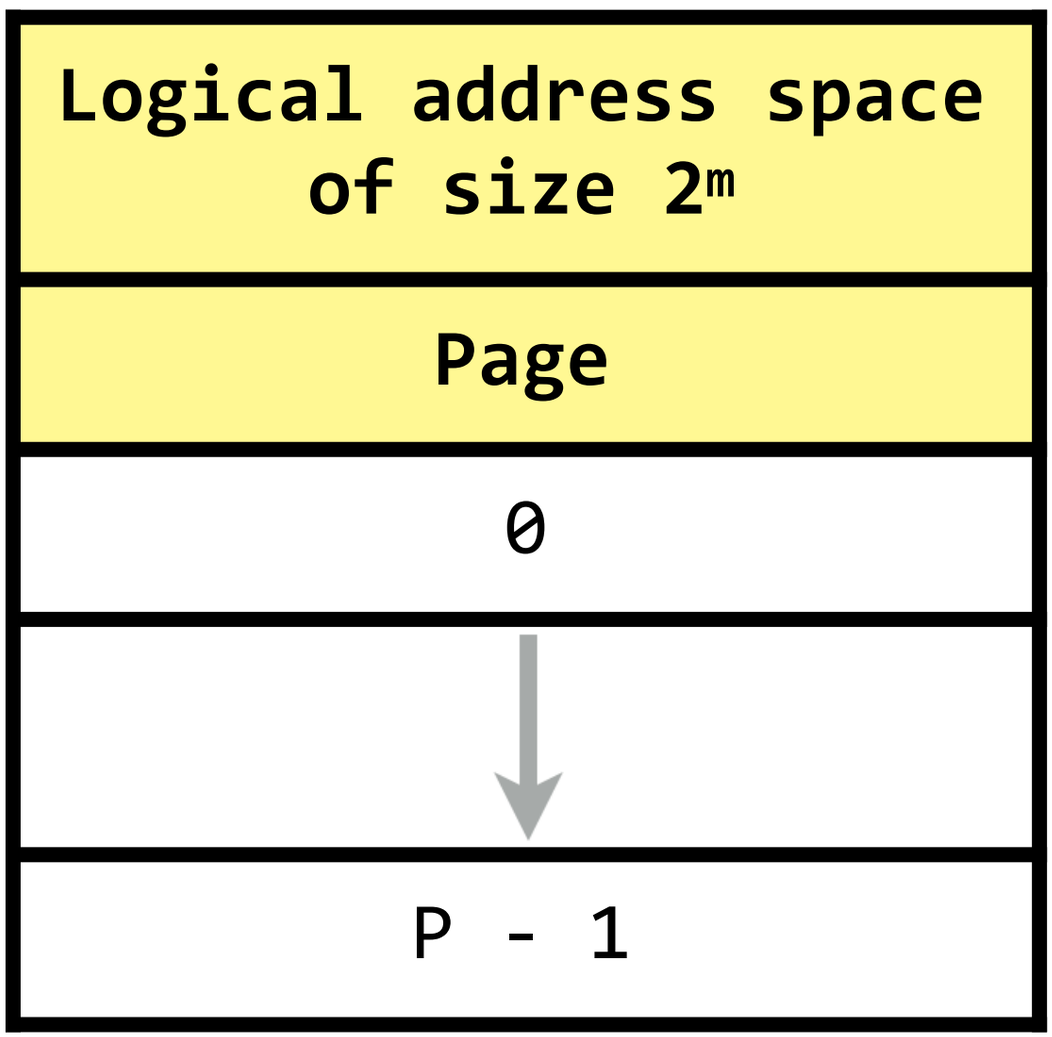
The size of both the logical address space and the physical address space will be a multiple of the page/frame size.

When allocating a new page we can use any free frame. The physical memory can be allocated non-contiguous on a page basis while the logical address space appears contiguous.



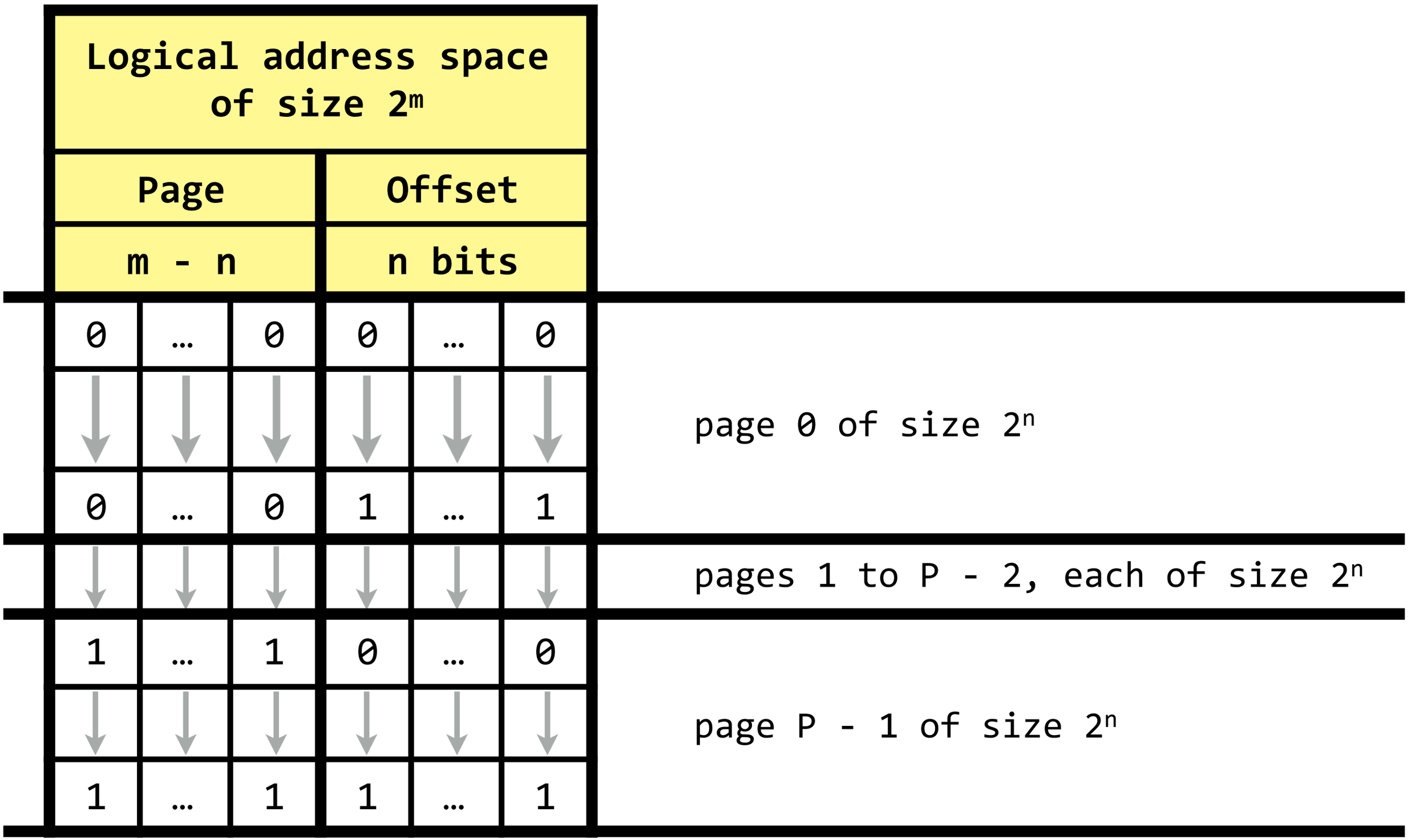
For a logical address space of size 2^m and a page/frame size of 2^n, how many pages do we have?

$$P = \frac{2^m}{2^n} = 2^{m-n}$$



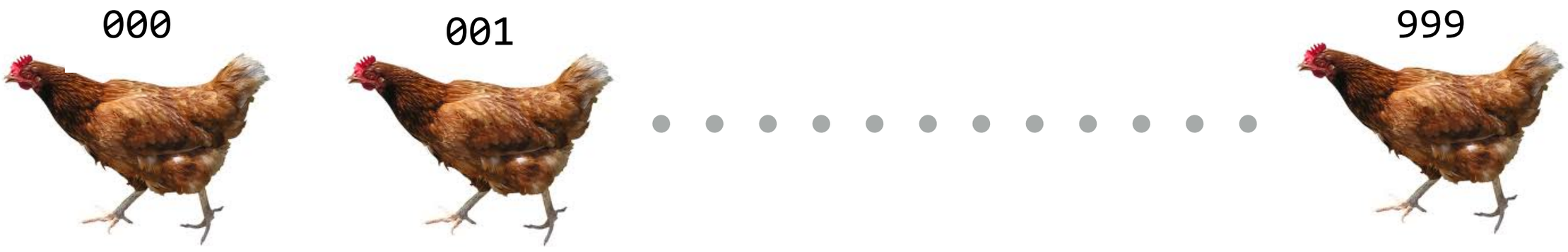
The **m - n most significant** bits of the logical address will identify the **page** number.

The **n least significant** bits of the logical address will be the page/frame **offset**.



Decimal hens and eggs

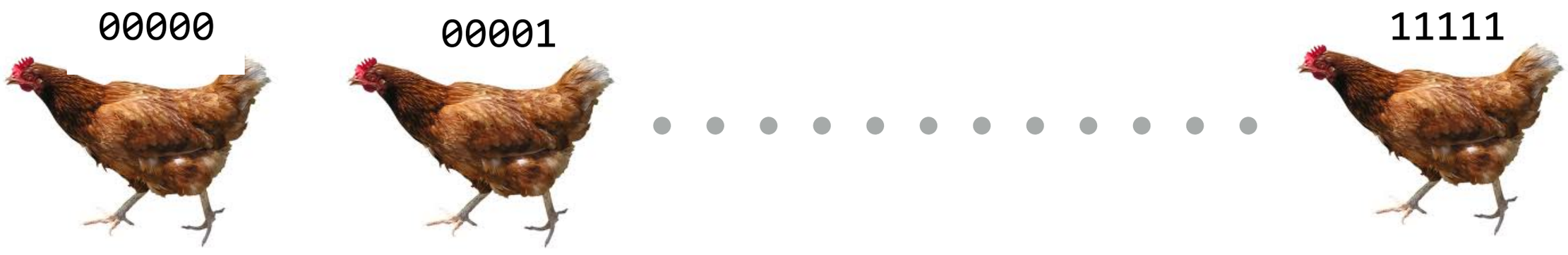
1000 hens, each with a 3 digit decimal number 000, 001, ..., 999 and 100 baskets numbered 00, 01, ..., 99.



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 mod 100 = 72 = 372 mod 10^2 = 2 last digits of 372 = 72.

Binary hens and eggs

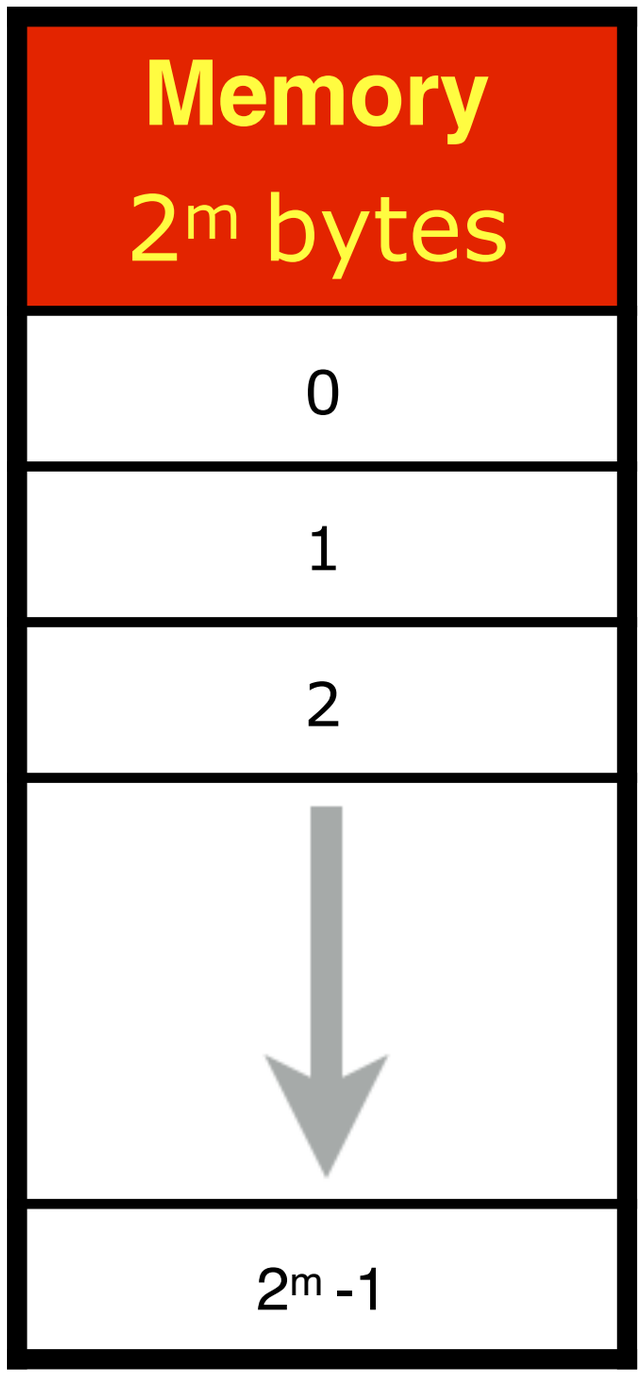
32 hens, each with a 5 bit binary number 00000, 00001, ..., 11111 and 8 baskets numbered 000, 001, ..., 111.



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 mod 8 = 10110 mod 2^3 = 3 last bits of 10110 = 110.

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.

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?

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

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?

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 mod (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)

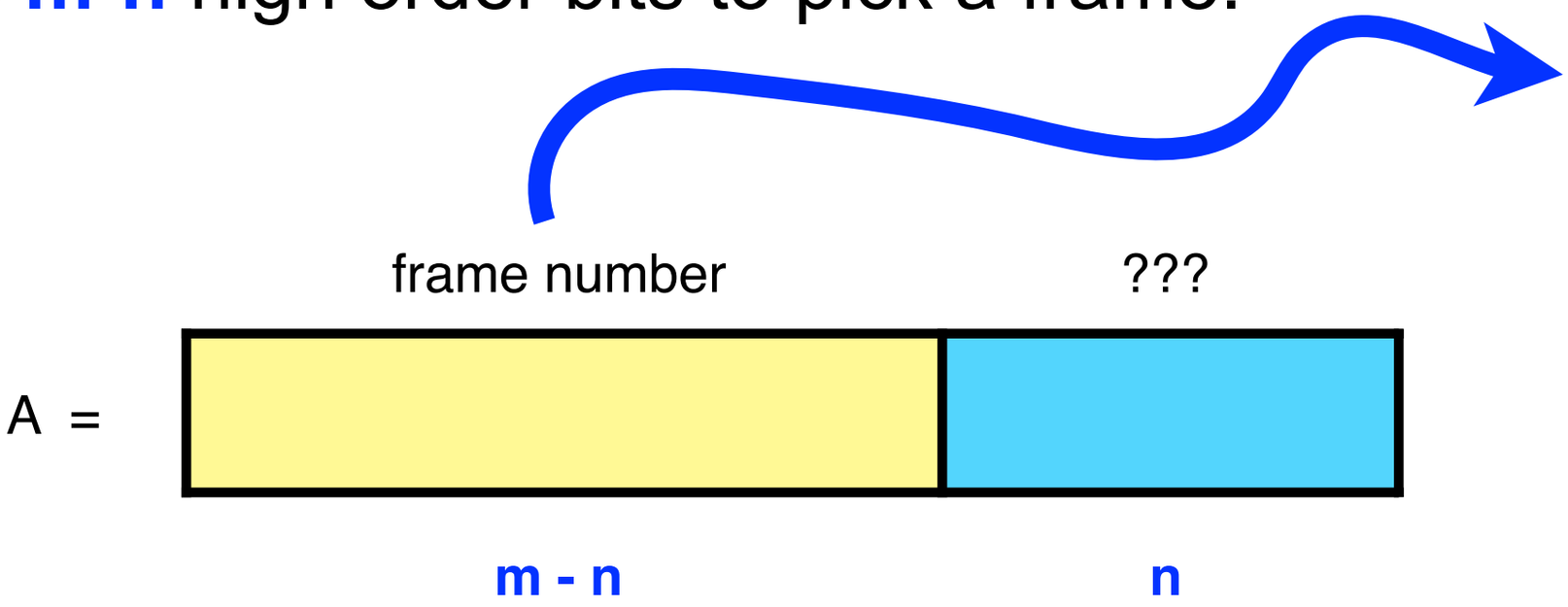
How can we map a logical address as seen by the CPU to a physical frame of size 2^n in memory of size 2^m?

m bit logical address seen by the CPU

A =

Frame size = 2^n ⇔ **m-n** frames.

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

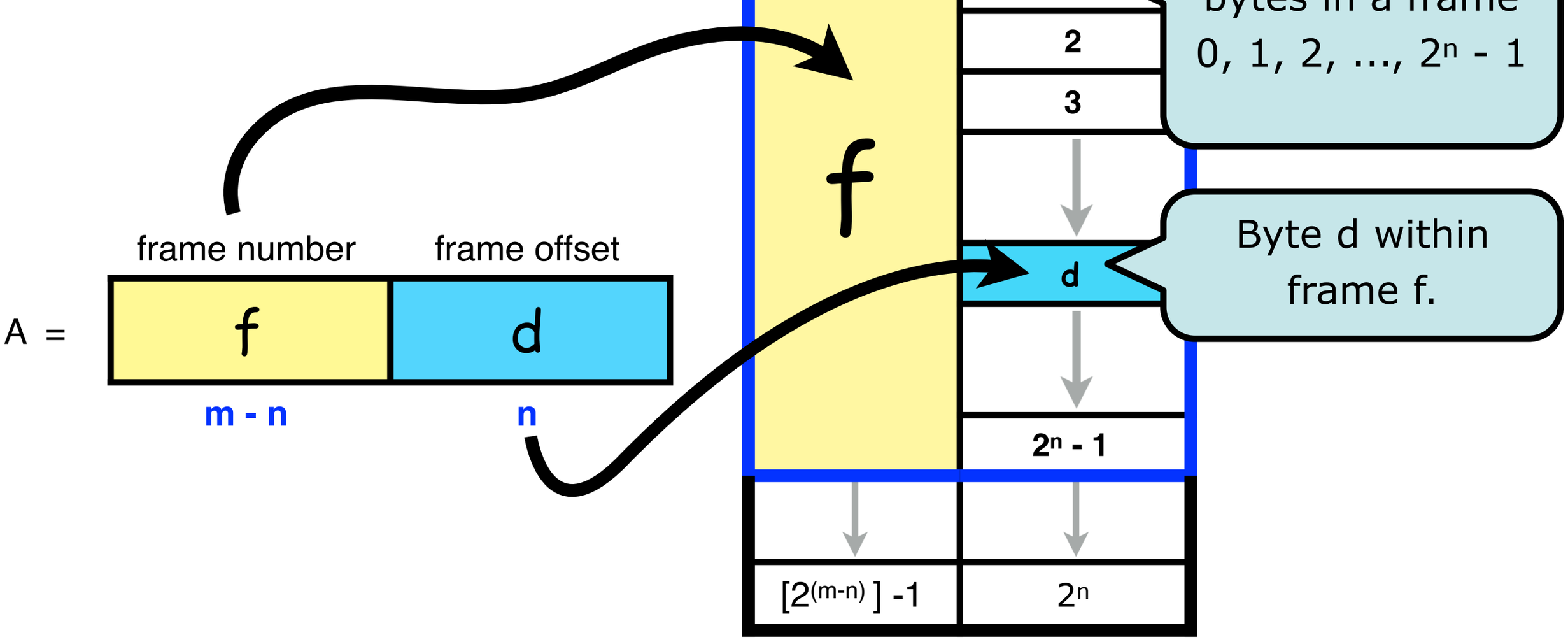


What does the **n** least significant bits mean?

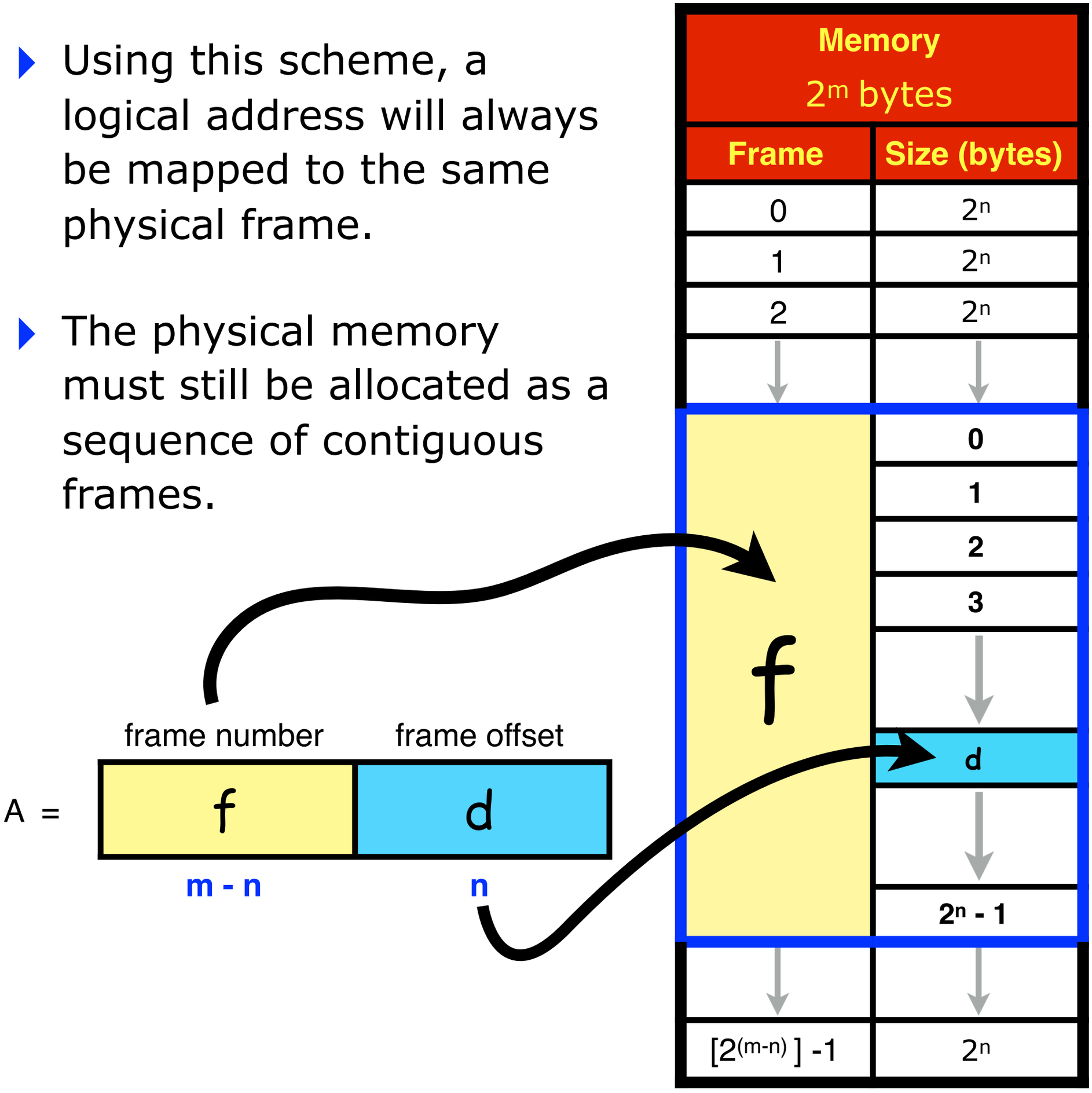
Memory 2^m bytes	
Frame	**Size (bytes)**
0	2^n
1	2^n
2	2^n
↓	↓
$[2^{(m-n)}] - 1$	2^n

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

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



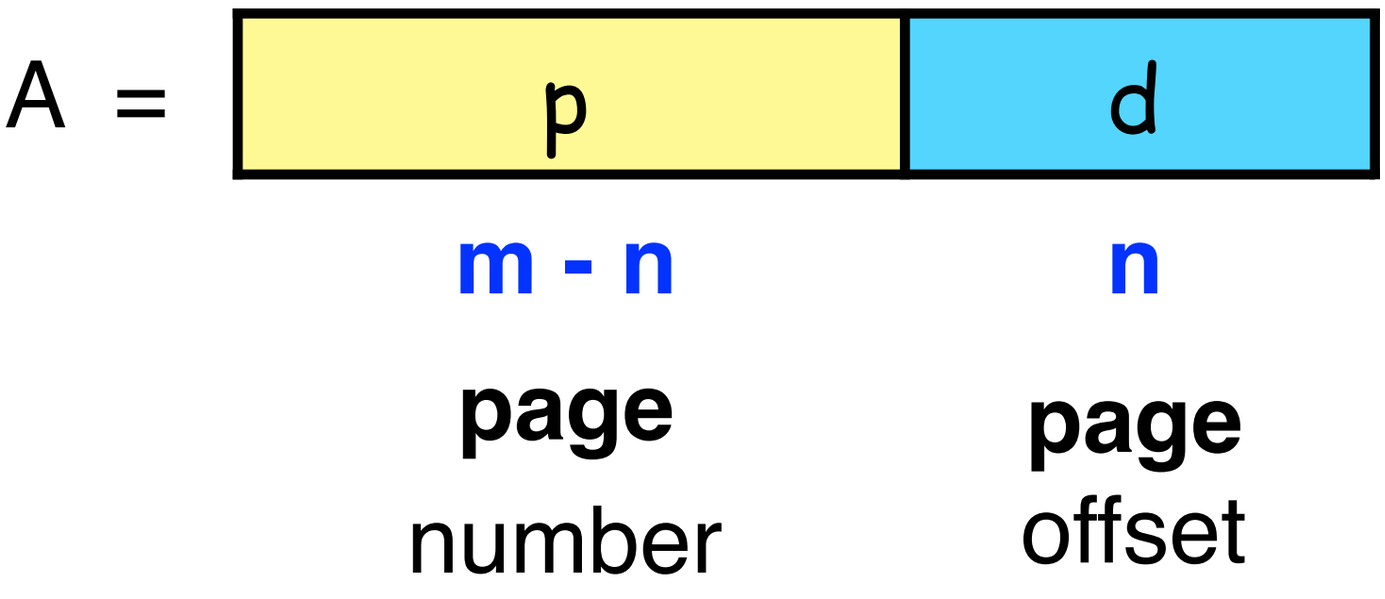
- Using this scheme, a logical address will always be mapped to the same physical frame.
- The physical memory must still be allocated as a sequence of contiguous frames.

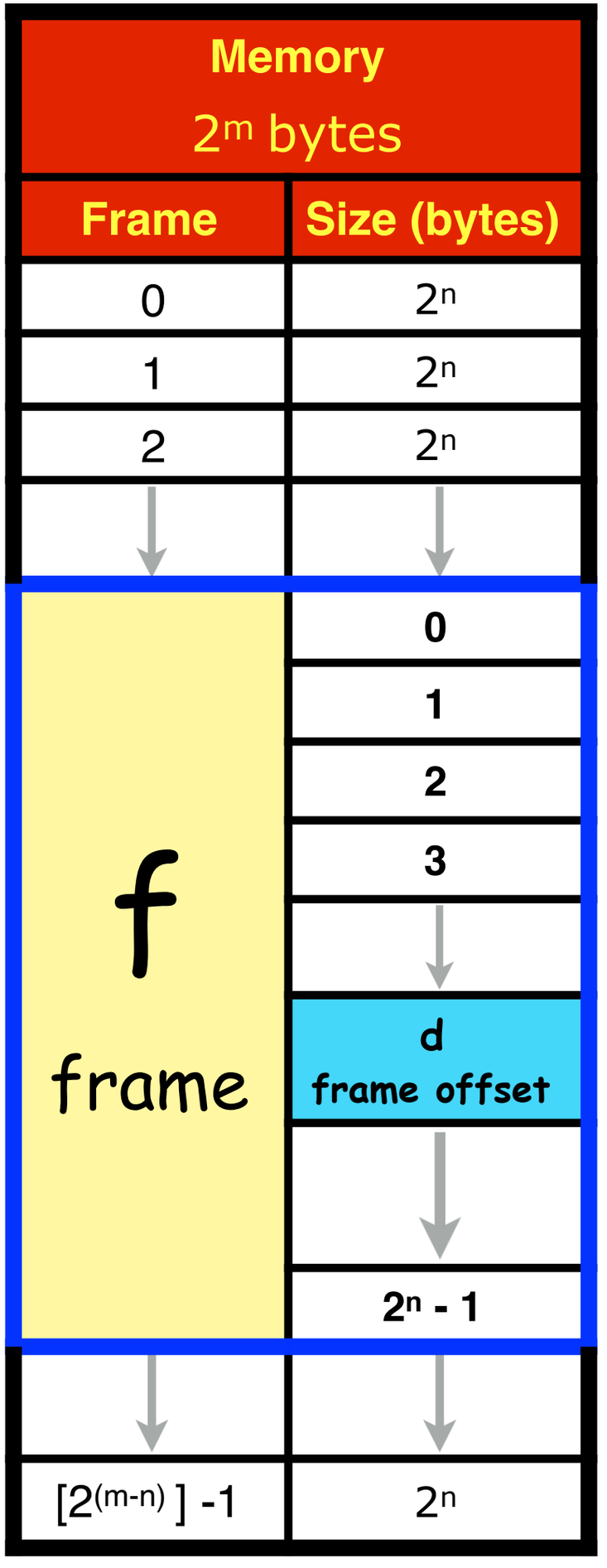


Pages and frames

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

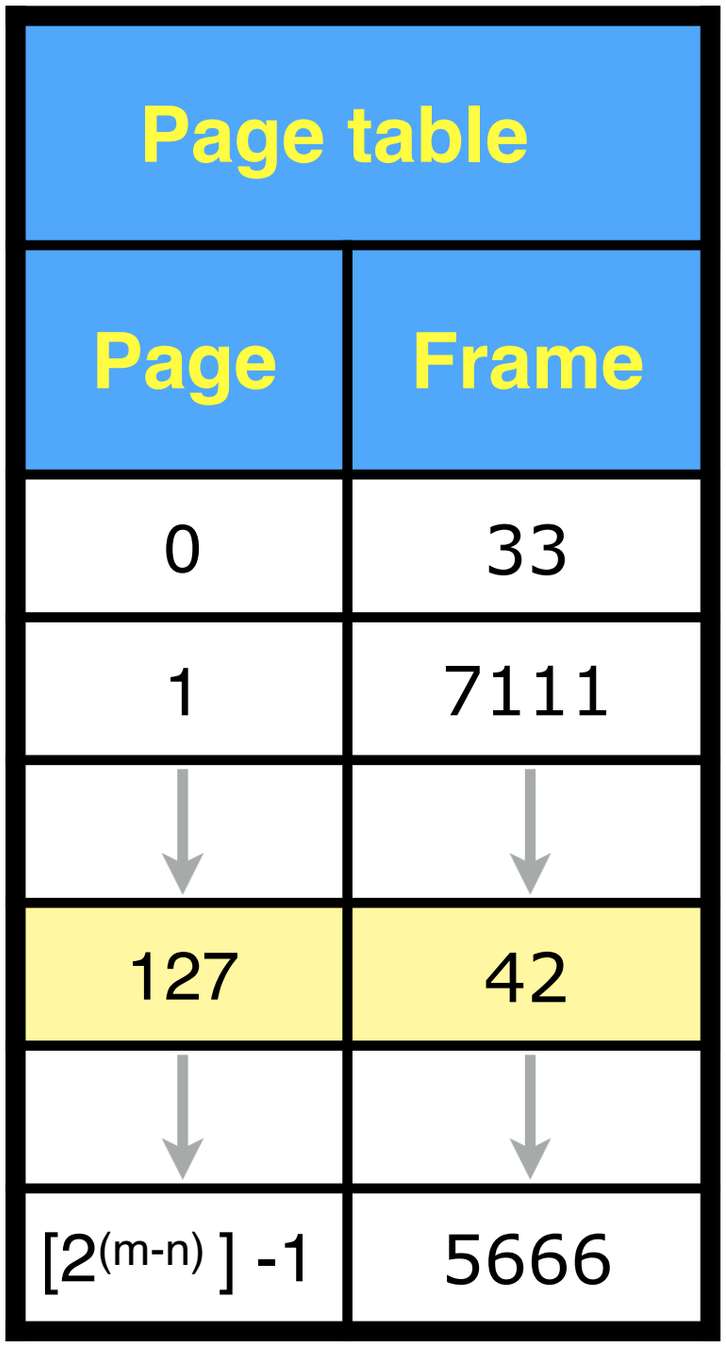
- **Logical** address space divided into fixed sized **pages**.
- **Physical** memory divided into **frames** of the same fixed size as the the pages.

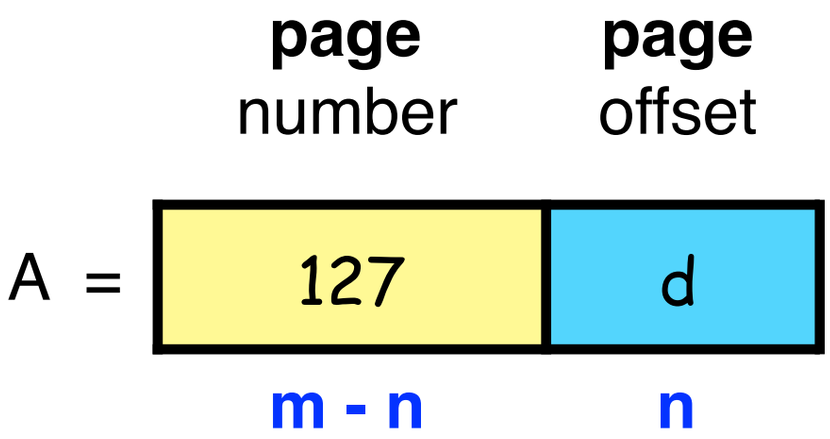


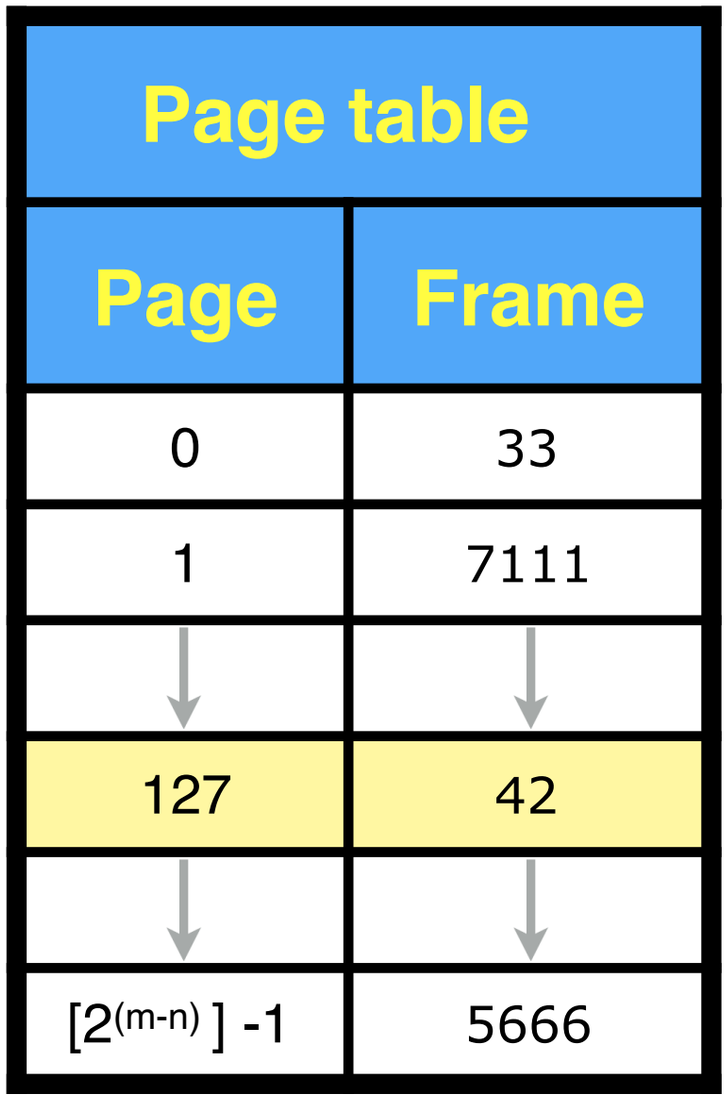


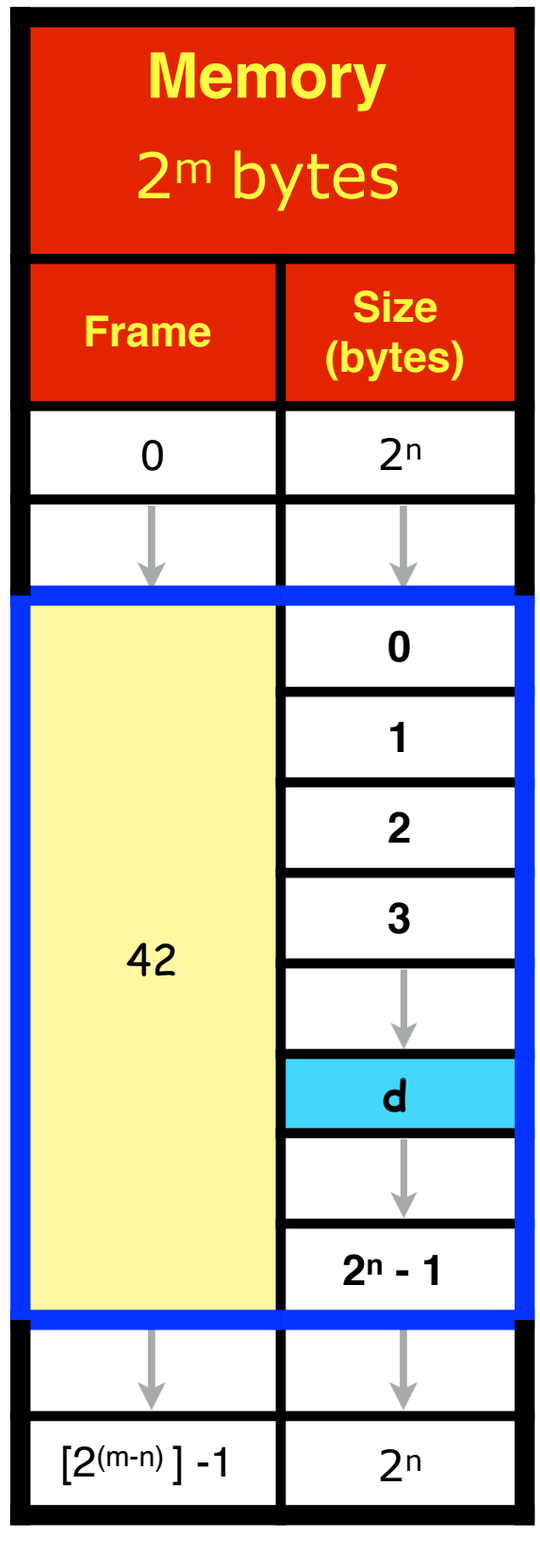
Page table

A page lookup table maps logical pages to physical frames.




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
2m bytes
Frame
Size
(bytes)
0
2n
42
0
1
2
3
d
2n - 1
[2(m-n)] -1
2n

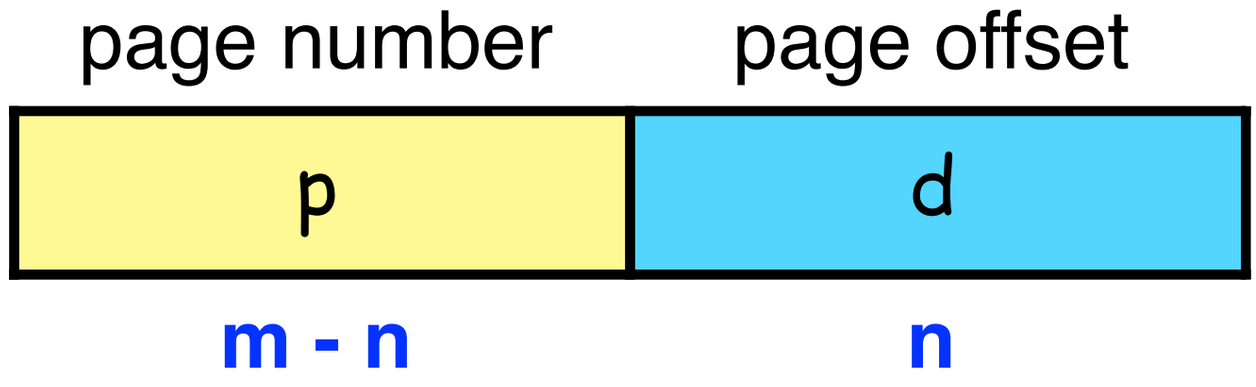
Paged memory management

- **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**.

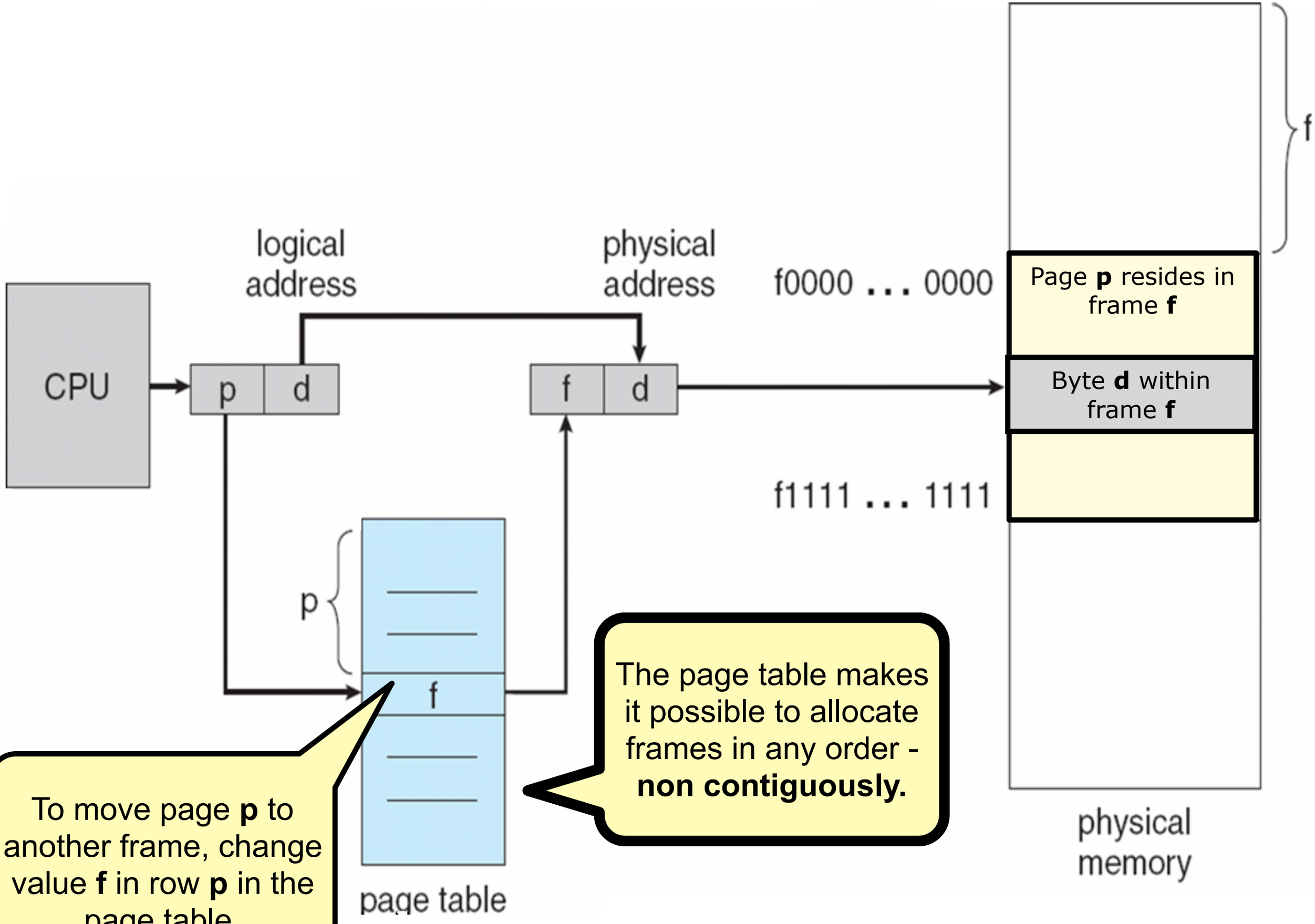
Address translation scheme

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
f1111 ... 1111
Page **p** resides in frame **f**
Byte **d** within frame **f**
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

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

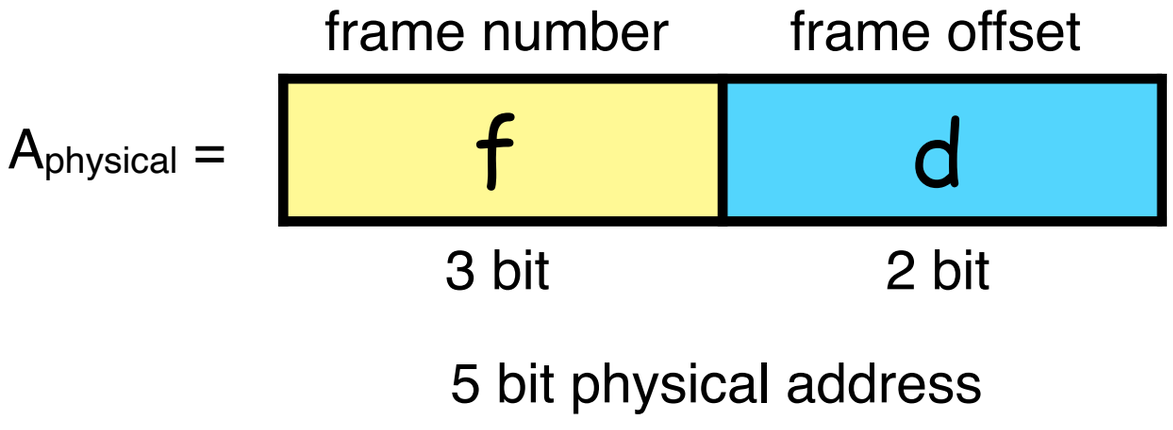
How large physical address?

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



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

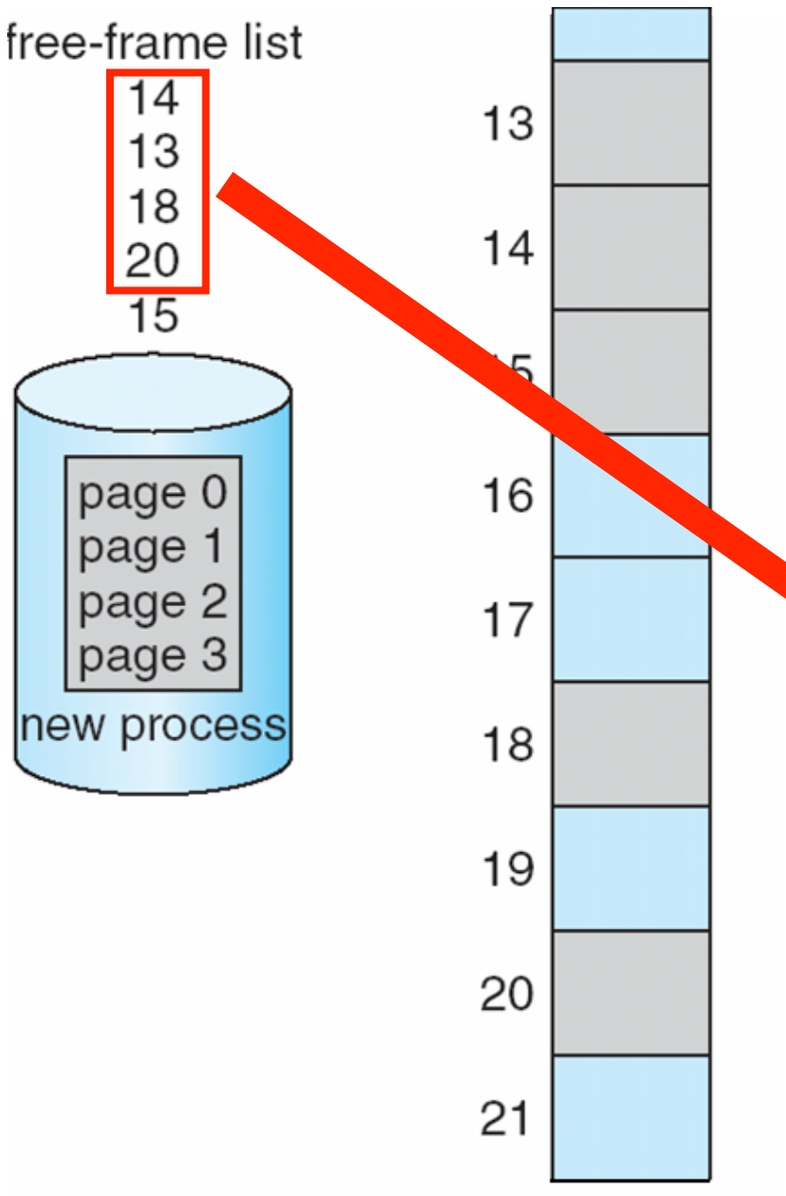
Logical memory	
Page	**Byte content**
0	a
	b
	c
	d
1	e
	f
	g
	h
2	i
	j
	k
	l
3	m
	n
	o
	p

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

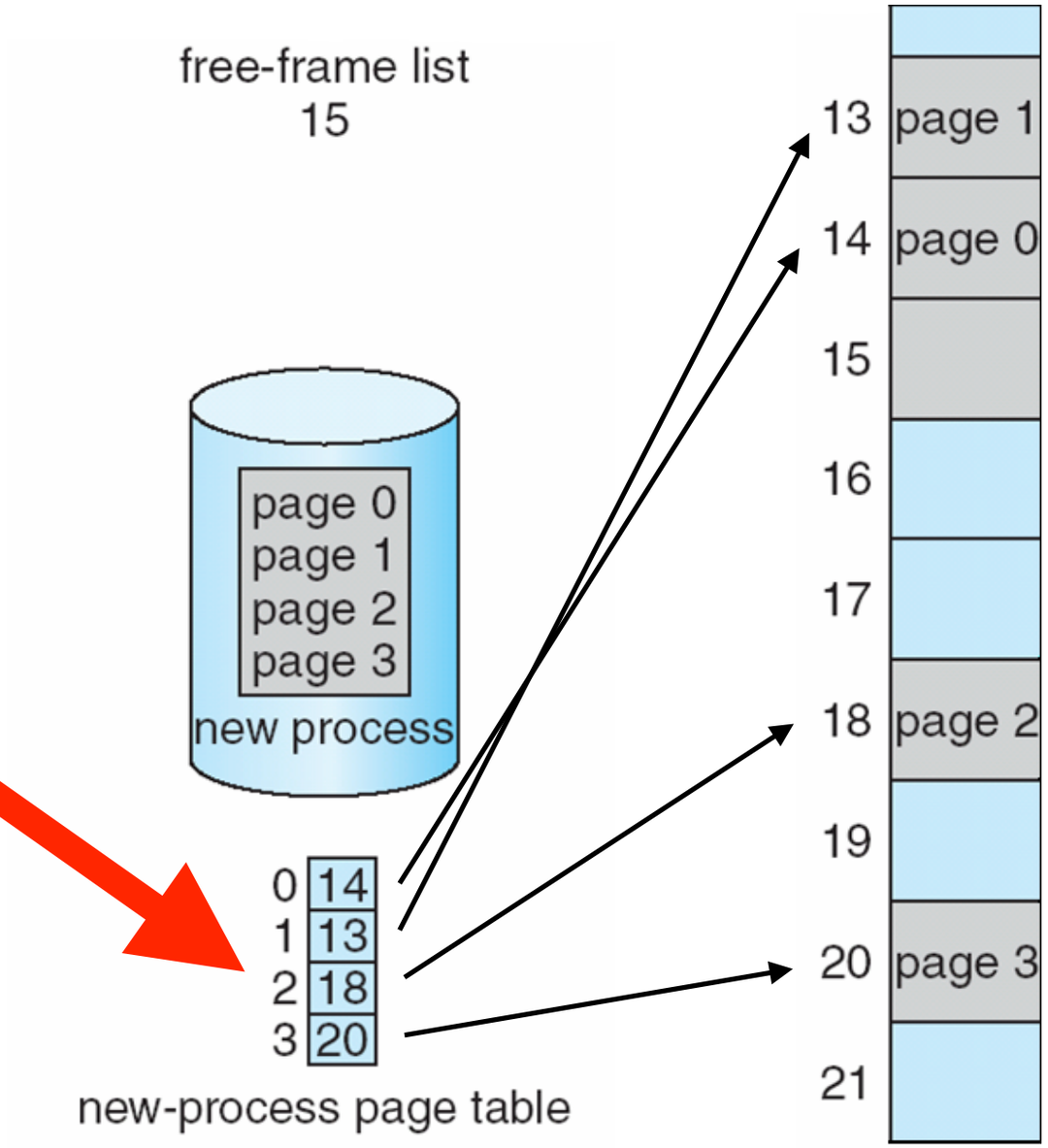
Physical Memory 32 bytes			
byte	**frame**	**offset**	**content**
0	0	0	
1		1	
2		2	
3		3	
4	1	0	i
5		1	j
6		2	k
7		3	l
8	2	0	m
9		1	n
10		2	o
11		3	p
12	3	0	
13		1	
14		2	
15		3	
16	4	0	
17		1	
18		2	
19		3	
20	5	0	a
21		1	b
22		2	c
23		3	d
24	6	0	e
25		1	f
26		2	g
27		3	h
28	7	0	
29		1	
30		2	
31	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

List of free frames

Before allocation

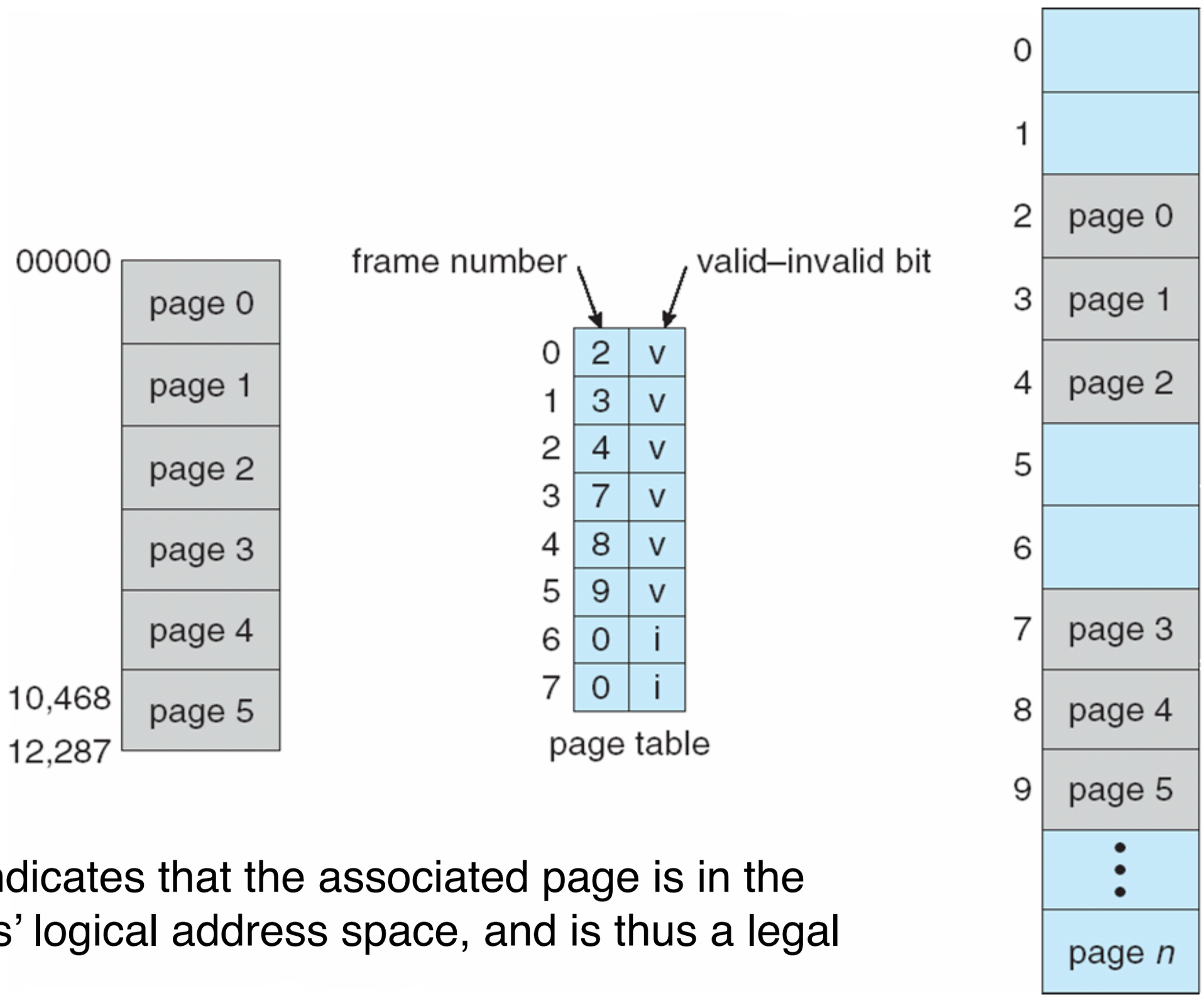


After allocation



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

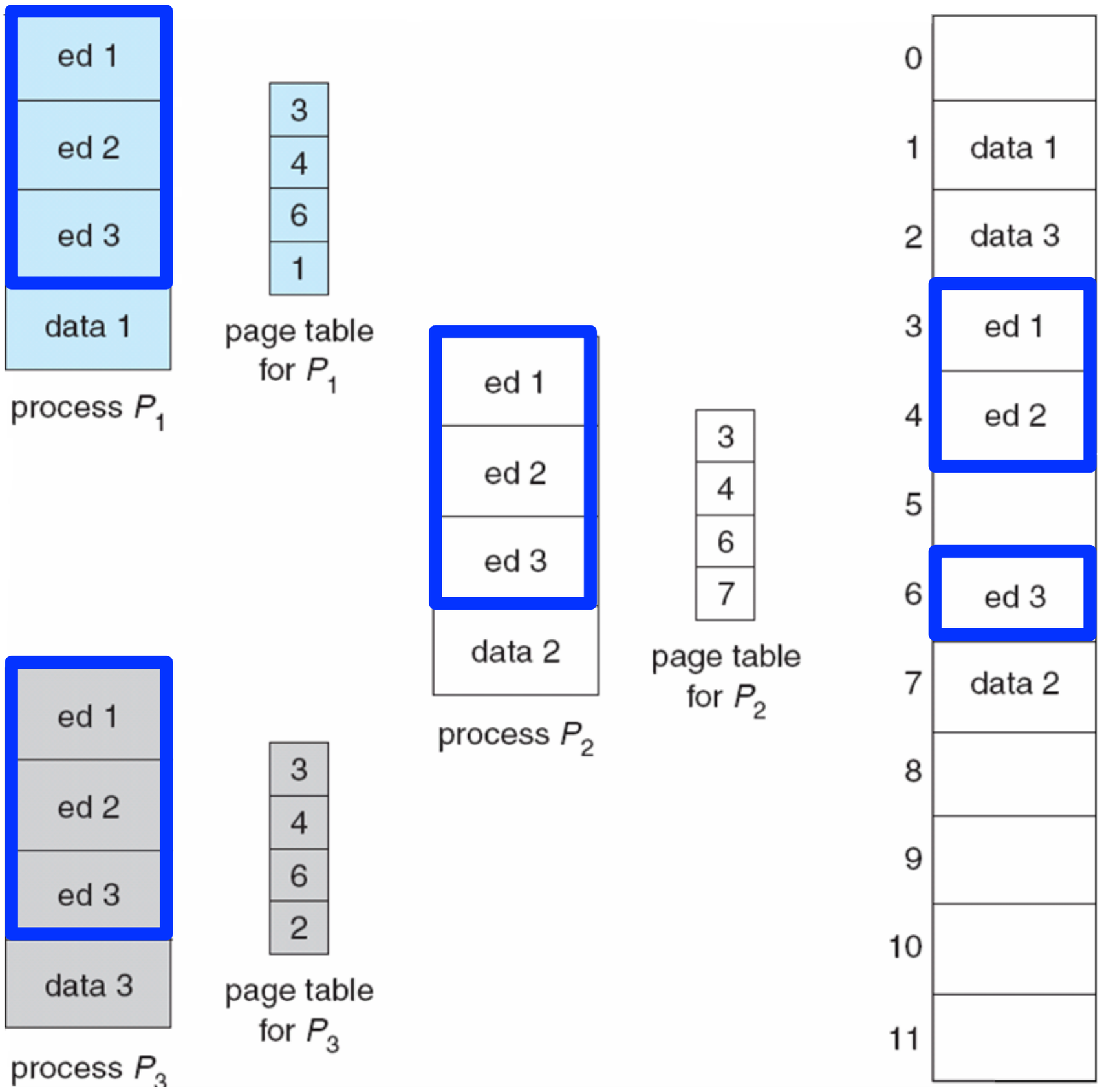
Paging makes it possible for processes to share parts of their memory space with each other.

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.


ed 1
ed 2
ed 3
data 1
process P_1
3
4
6
1
page table for P_1
ed 1
ed 2
ed 3
data 2
process P_2
3
4
6
7
page table for P_2
ed 1
ed 2
ed 3
data 3
process P_3
3
4
6
2
page table for P_3
0
1 data 1
2 data 3
3 ed 1
4 ed 2
5
6 ed 3
7 data 2
8
9
10
11

fork()
&
copy-on-write

Paging makes it possible for the child to initially share pages with its parent.

On Linux

```
$> man fork
```

Under Linux, **`fork()`** is implemented using **copy-on-write** pages, so the only penalty that it incurs is the time and memory required to duplicate the parent's page tables, and to create a unique task structure for the child.

Copy-on-write finds its main use in sharing the virtual memory of operating system processes, in the implementation of the **fork** system call.

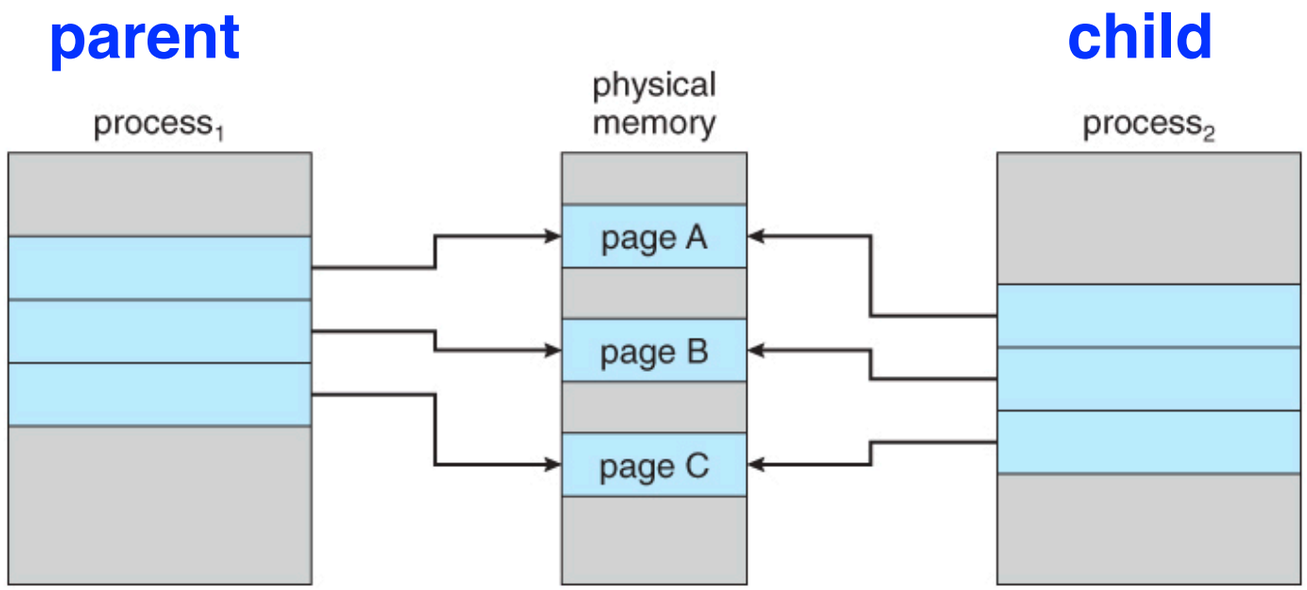
Typically, the process does not modify any memory and immediately executes a new process, replacing the address space entirely.[1]

Thus, it would be wasteful to copy all of the process's memory during a fork, and instead the **copy-on-write** technique is used.

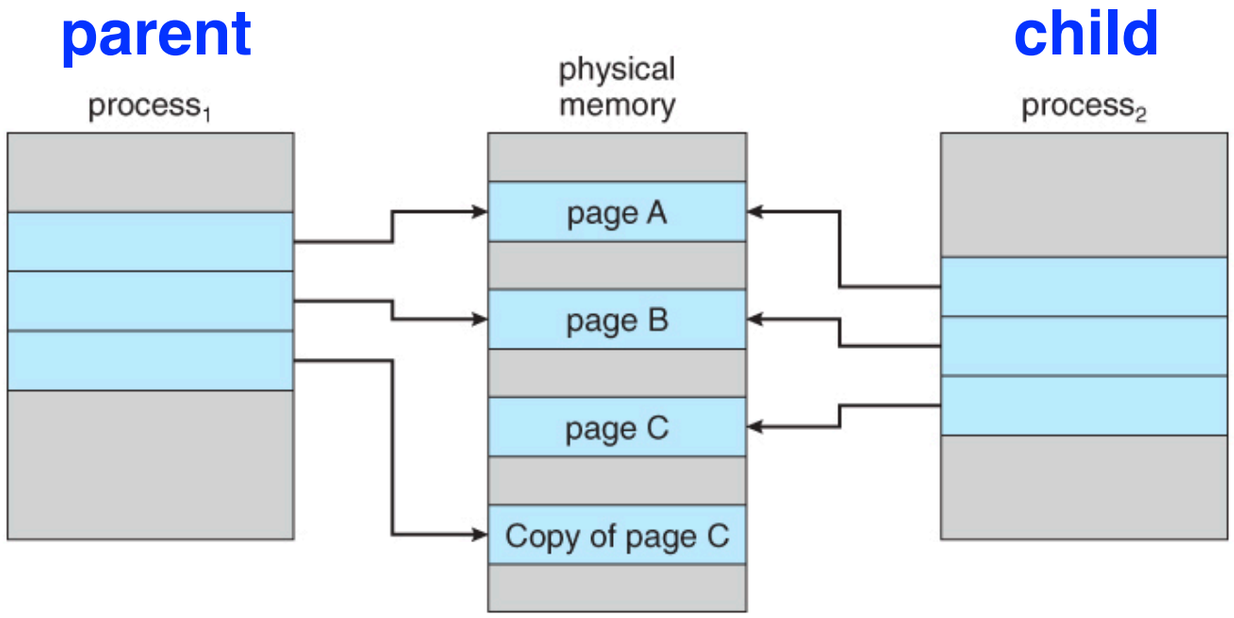
Source: https://en.wikipedia.org/wiki/Copy-on-write 2020-03-07

1) Typically the child immediately replaces the memory image using **`exec()`**.

After **`fork()`** the parent and child shares all pages.



If the parent modifies page C after **`fork()`**, a copy-on-write of page C is triggered.



Obviously only pages that can be modified even need to be labeled as copy-on-write. Code [text] segments can simply be shared.

Some systems provide an alternative to the `fork()` system call called a virtual memory fork, `vfork()`.

In this case the parent is suspended, and the child uses the parent's memory pages. This is very fast for process creation, but requires that the child not modify any of the shared memory pages before performing the `exec()` system call.

In essence this addresses the question of which process executes first after a call to fork, the parent or the child. With `vfork()`, the parent is suspended, allowing the child to execute first until it calls `exec()`, sharing pages with the parent in the meantime.

The `vfork()` function has the same effect as `fork()`, except that the behavior is undefined if the process created by `vfork()` either modifies any data other than a variable of type `pid_t` used to store the return value from `vfork()`, or returns from the function in which `vfork()` was called, or calls any other function before successfully calling `_exit()` or one of the `exec()` family of functions.

Source: http://man7.org/linux/man-pages/man2/vfork.2.html 2020-03-07

Both Linux and macOS implements the `vfork()` system call.

Translation lookaside buffer

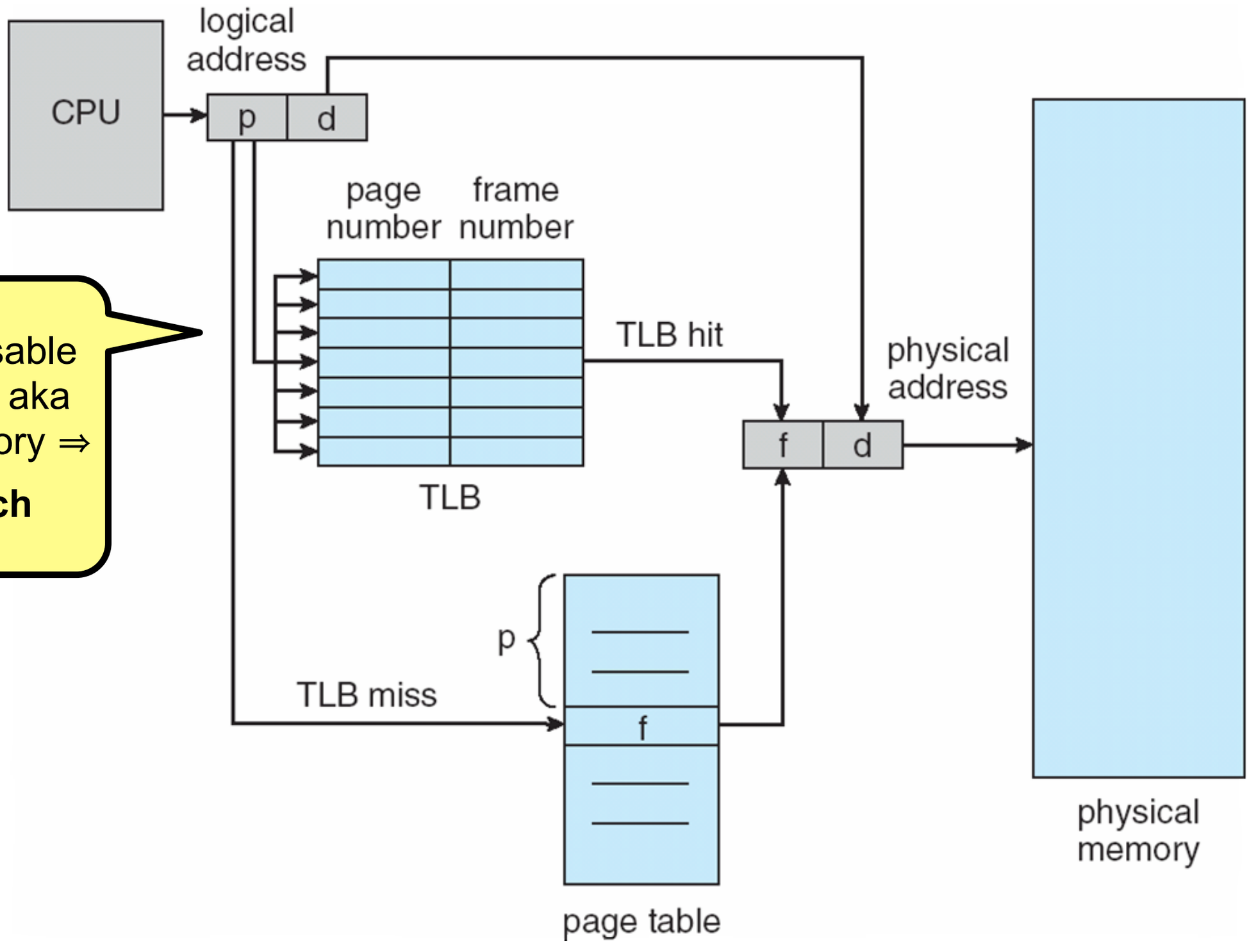
A translation lookaside buffer (TLB) is a memory **cache** that is used to reduce the time taken to access a user memory location.

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

Source: https://en.wikipedia.org/wiki/Translation_lookaside_buffer 2018-02-22

Paging hardware with TLB

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



Content Addressable Memory (CAM), aka associative memory ⇒

parallel search

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

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

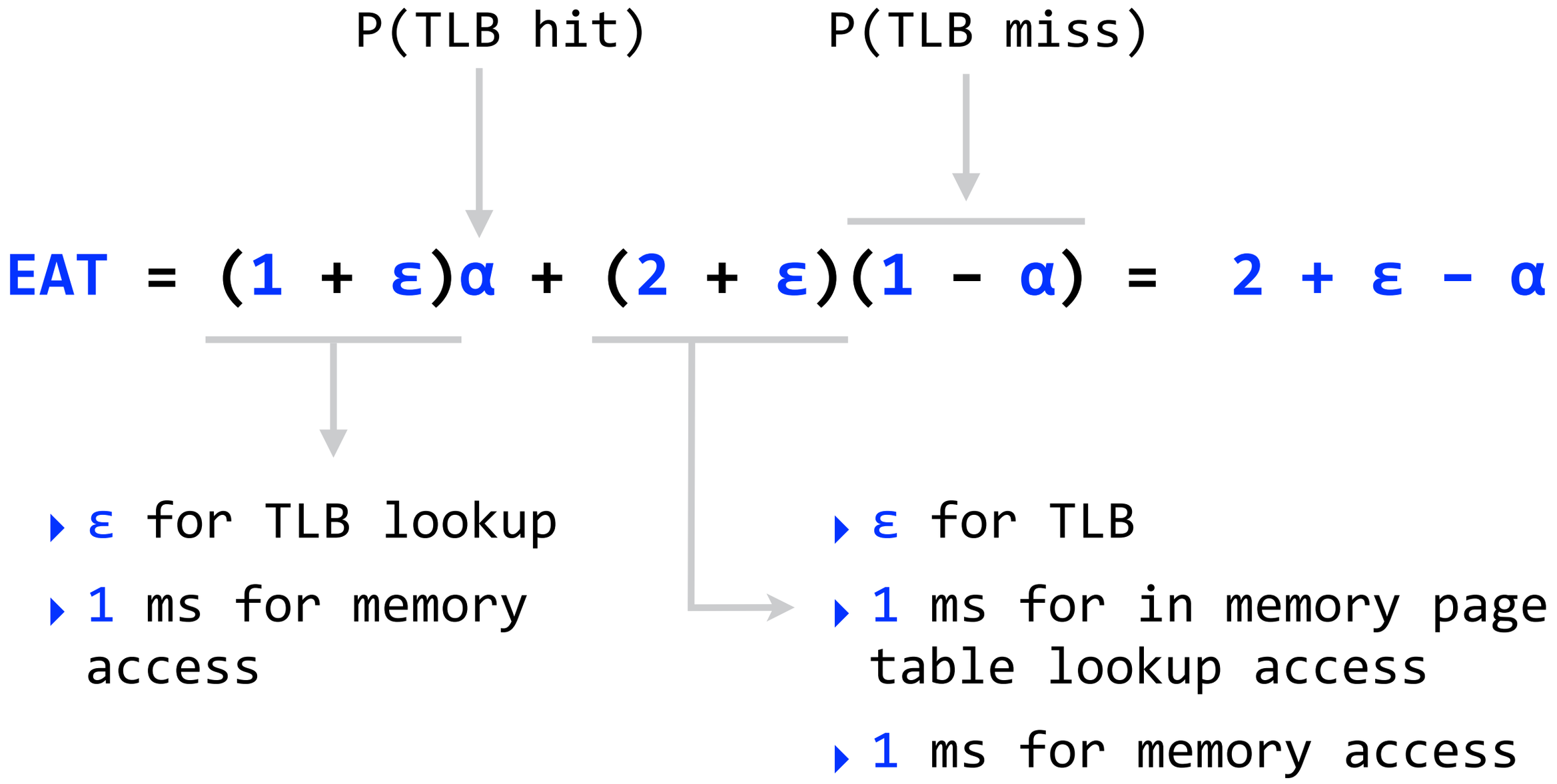
$$P(\text{TLB miss}) = (1-\alpha)$$

Assume the following memory access times.

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

Effective access time (3)

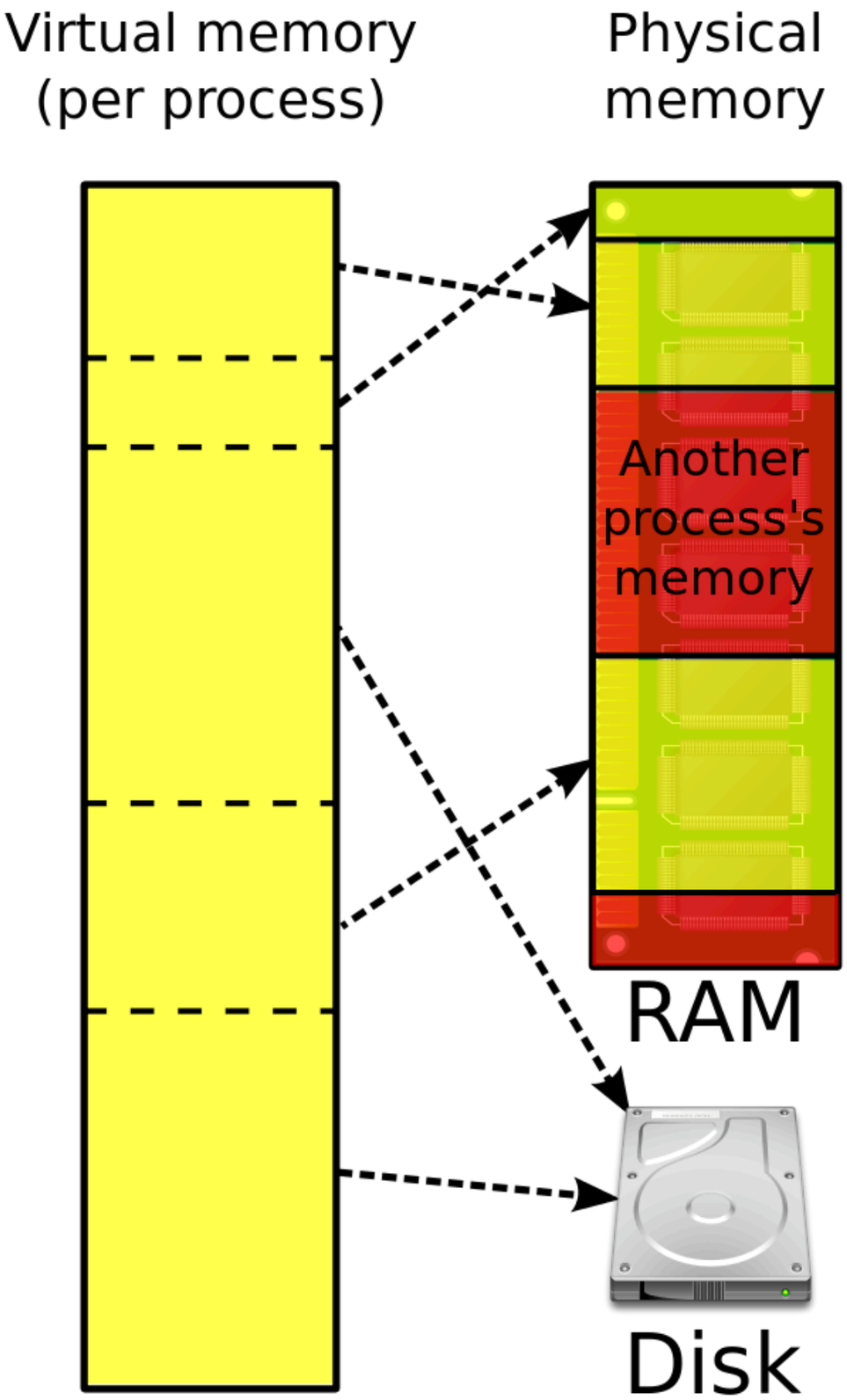
Effective access time (EAT).



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.

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



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.

Source https://en.wikipedia.org/wiki/Virtual_memory 2018-02-22

Learn more in the following self study material

Page table implementation

Module 5 self study material

Operating systems 2020

1DT003, 1DT044 and 1DT096

Segmentation

Module 5 self study material

Operating systems 2020

1DT003, 1DT044 and 1DT096