CSCI 509

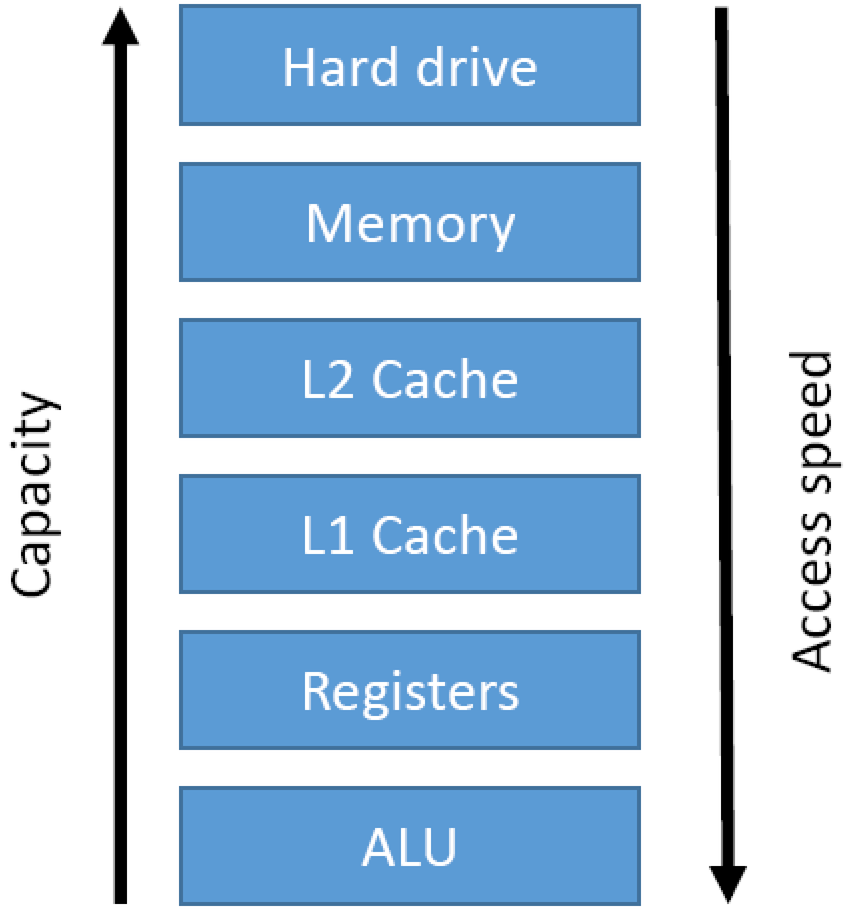
OPERATING SYSTEMS INTERNALS

PART 3: MEMORY MANAGEMENT

Chapter 9: Main Memory

MEMORY HIERARCHY

- When we talk about accessing memory, which memory are we talking about?
- Addresses refer to what part of memory?



Hard drive

Memory

L2 Cache

L1 Cache

Registers

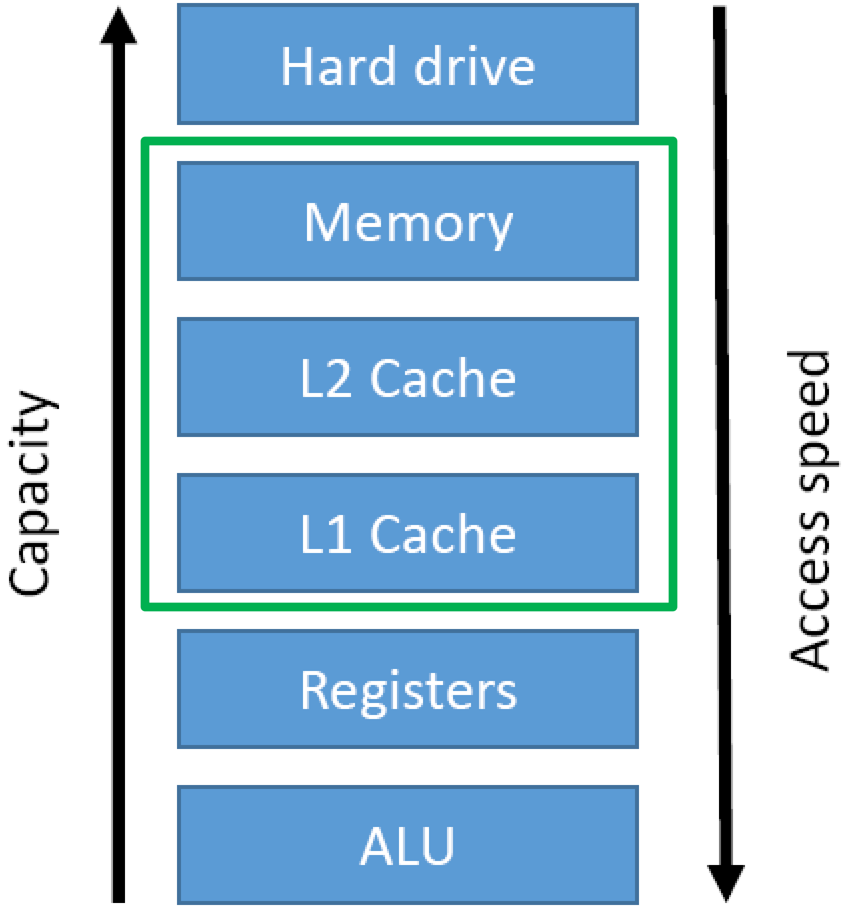
ALU

Capacity

Access speed

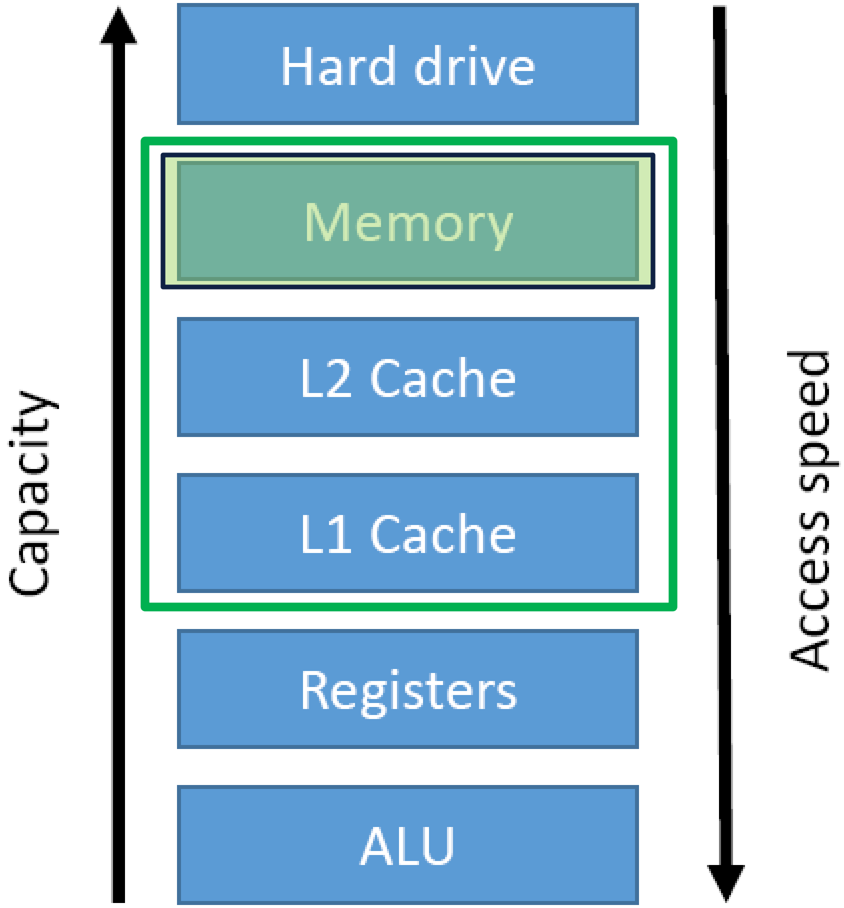
MEMORY HIERARCHY

- Hard drives are not considered part of memory, rather part of storage.



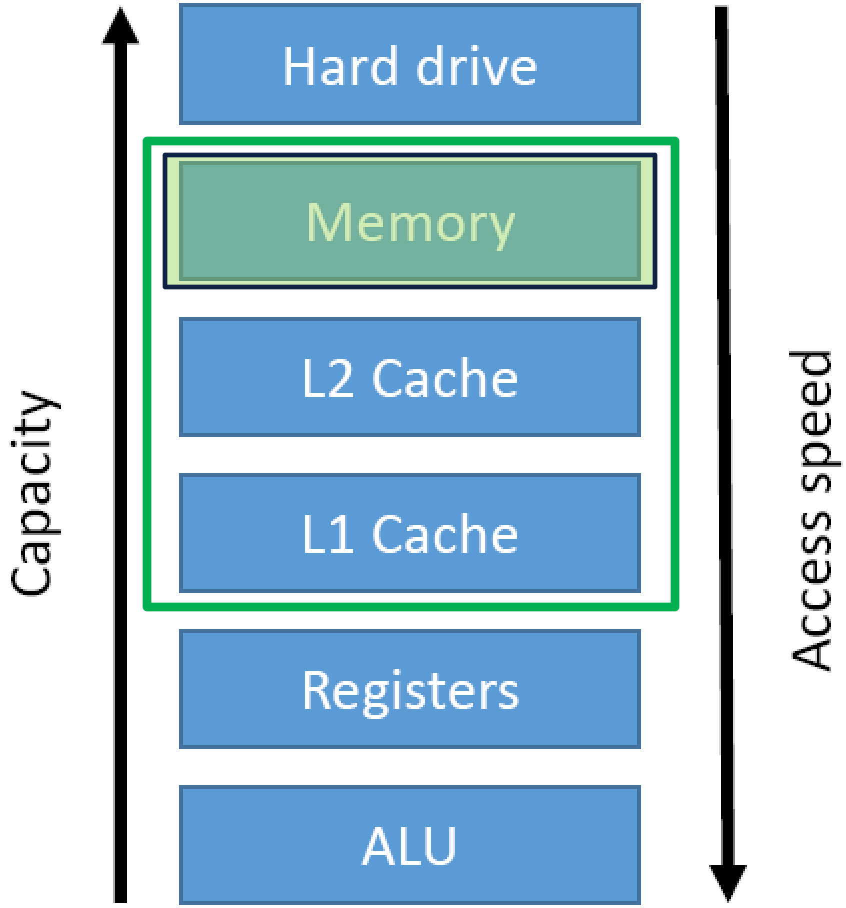
MEMORY HIERARCHY

- Hard drives are not considered part of memory, rather part of storage.
- A process/data is in memory if it's in main memory.



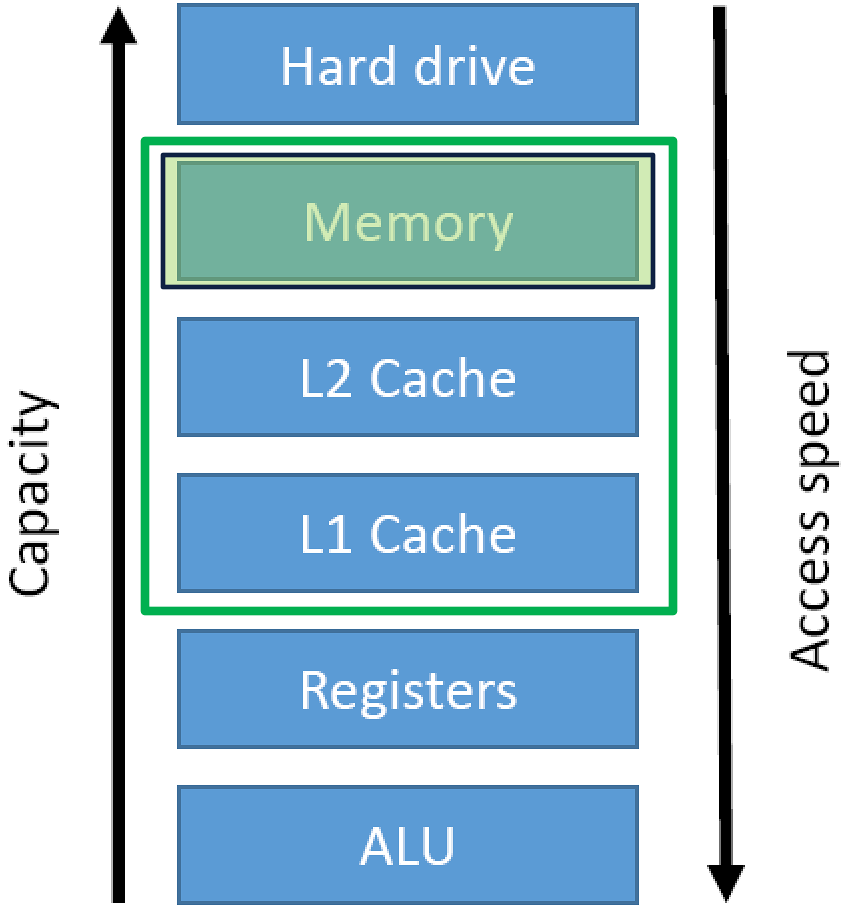
MEMORY HIERARCHY

- Hard drives are not considered part of memory, rather part of storage.
- A process/data is in memory if it's in main memory.
- Addresses refer to memory locations in Main Memory (RAM)
- L1 and L2 can be considered an extension of Main Memory.



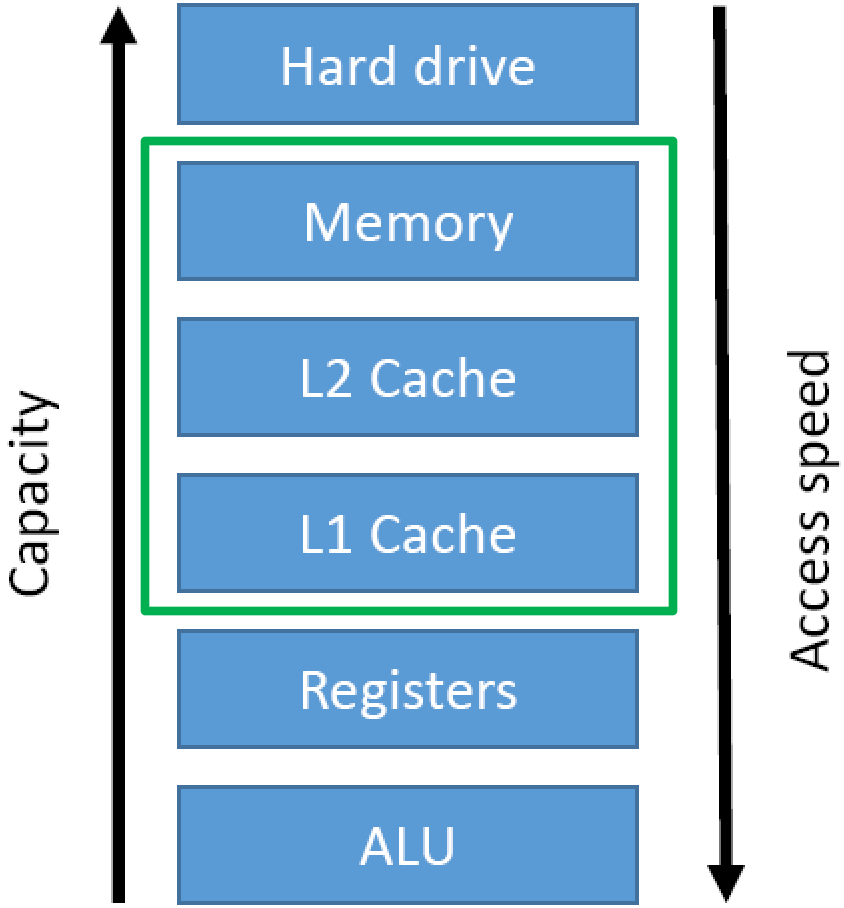
MEMORY HIERARCHY

- Hard drives are not considered part of memory, rather part of storage.
- A process/data is in memory if it's in main memory.
- Addresses refer to memory locations in Main Memory (RAM)
- L1 and L2 can be considered an extension of Main Memory.
- Main memory would also contain a copy of L1 and L2 Cache
- However, the Main memory copy may not be updated, it will be flagged non the less.



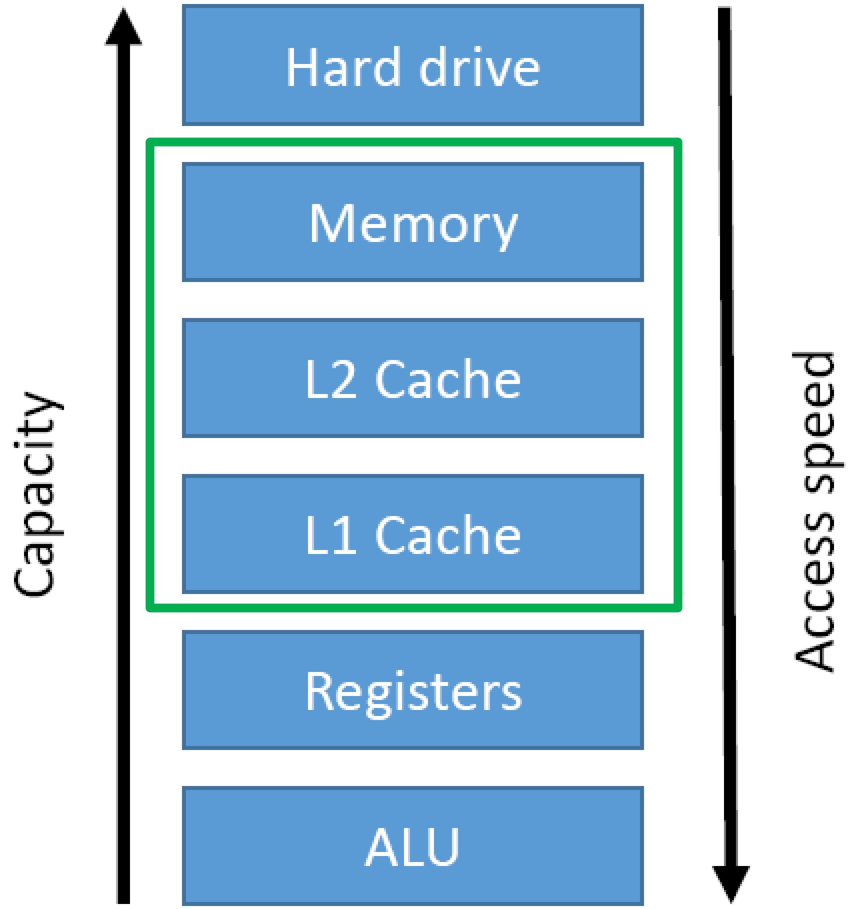
PROCESS MEMORY

- Each Process has its own memory space.
- Why?



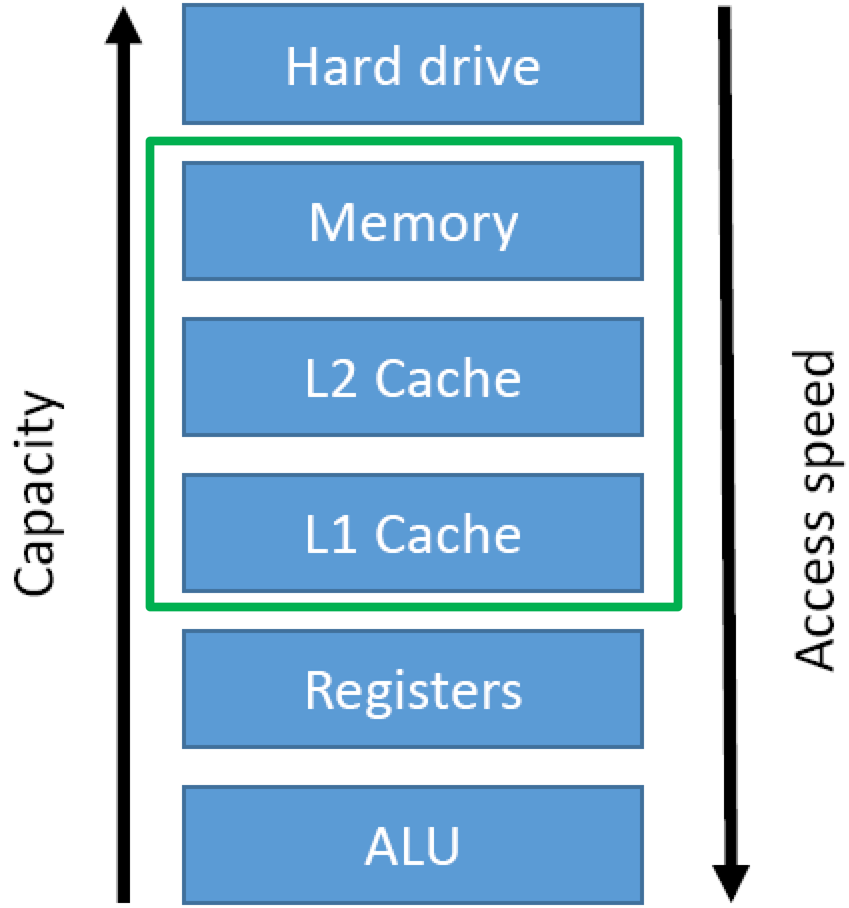
PROCESS MEMORY

- Each Process has its own memory space.
- Why?
  - Protection
  - Modularity



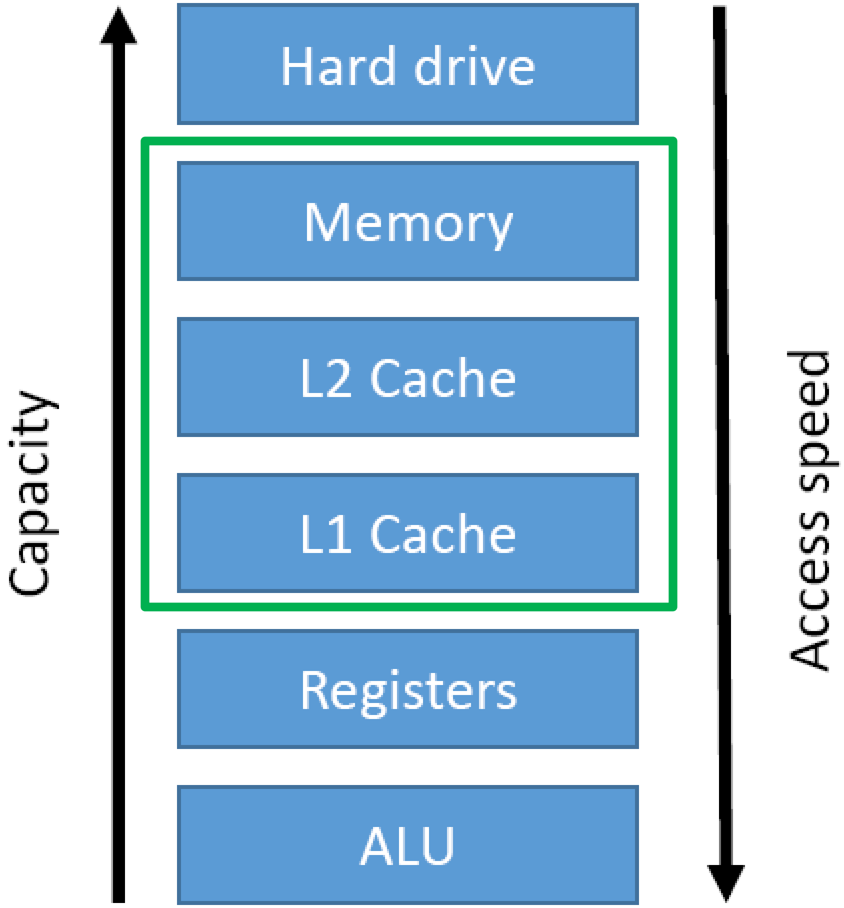
PROCESS MEMORY

- Each Process has its own memory space.
- Why?
  - Protection
  - Modularity
- Process A should not be able to access the memory of Process B.



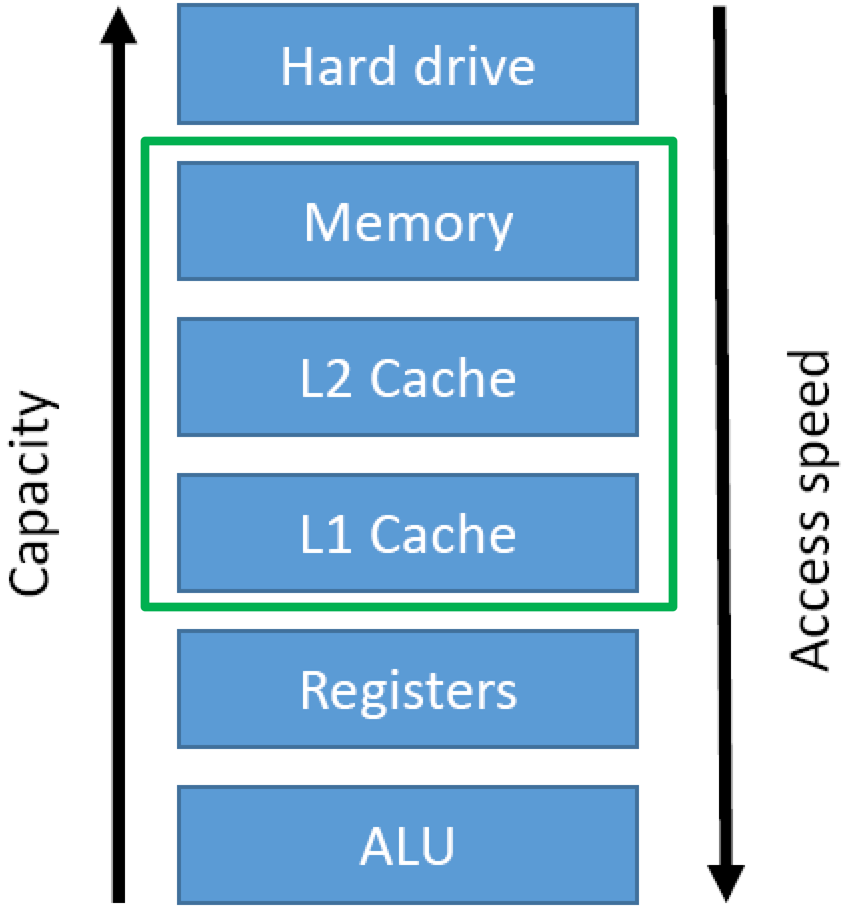
PROCESS MEMORY

- Each Process has its own memory space.
- Why?
  - Protection
  - Modularity
- Process A should not be able to access the memory of Process B.
- Exceptions?



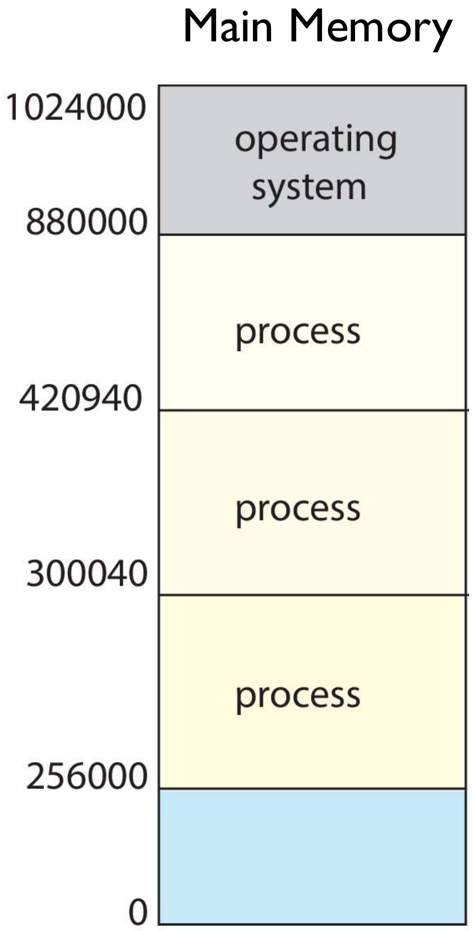
PROCESS MEMORY

- Each Process has its own memory space.
- Why?
  - Protection
  - Modularity
- Process A should not be able to access the memory of Process B.
- Exceptions?
- Except for shared memory segments, which are handled by OS).



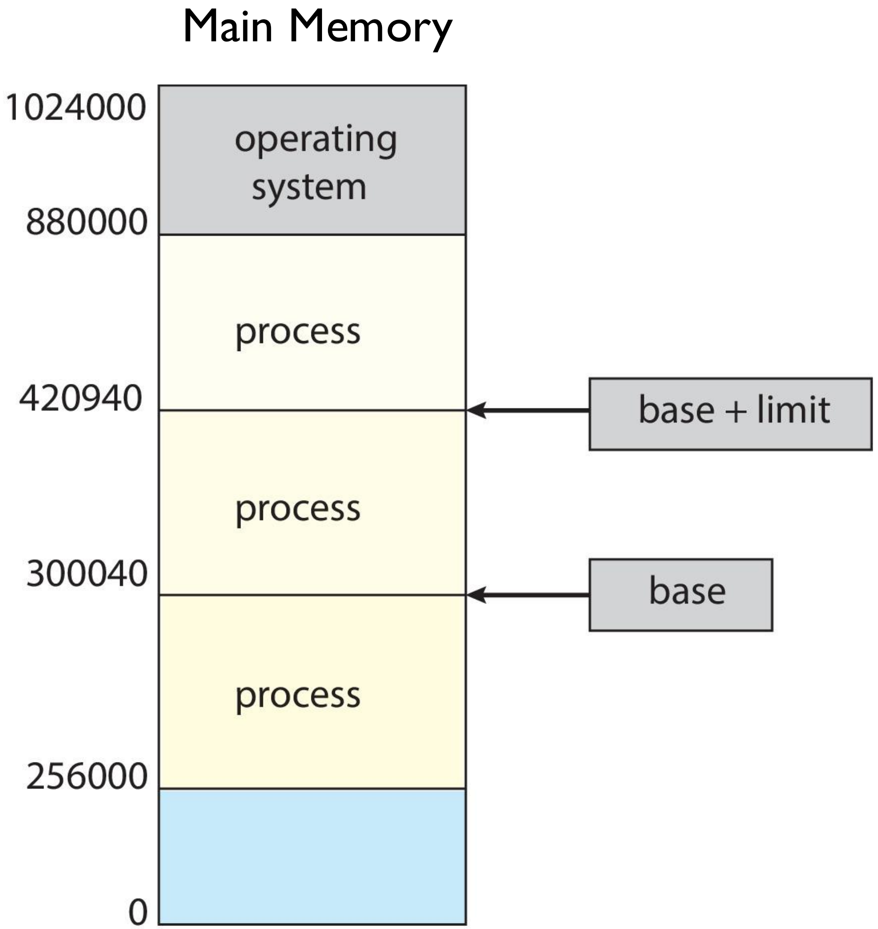
PROCESS MEMORY BOUNDARIES

Q: How to enforce memory protection and process memory?



PROCESS MEMORY BOUNDARIES

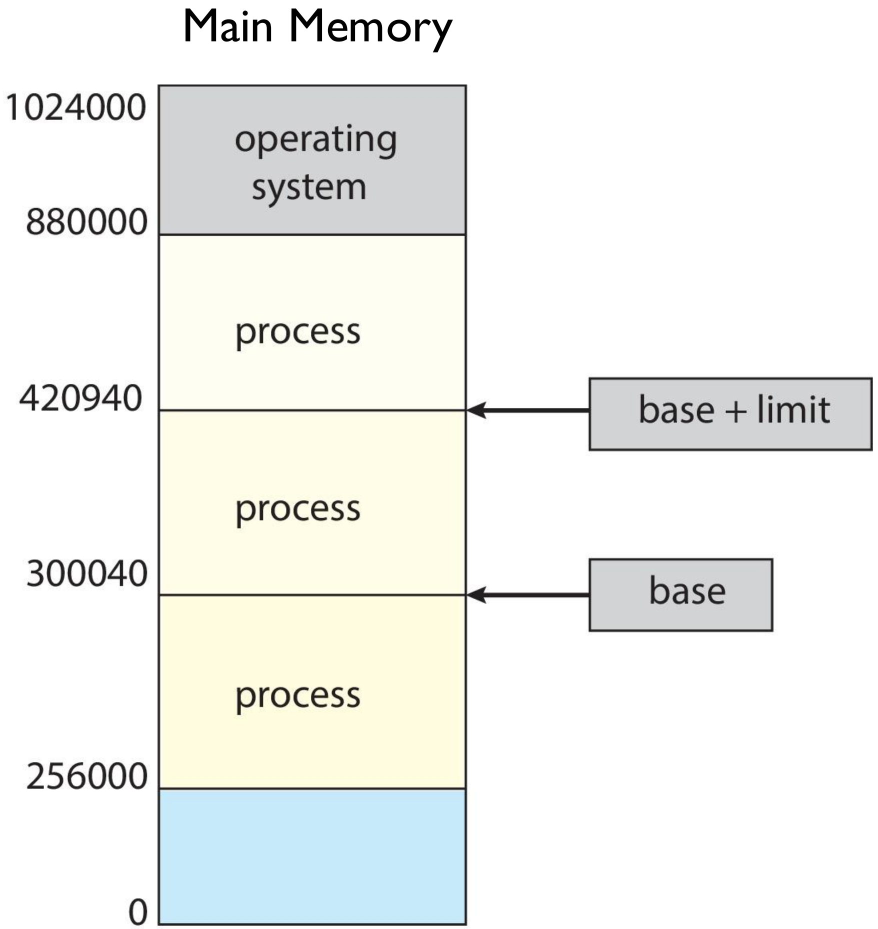
Two register values, **base** and **limit**, are used to hold the starting and end memory addresses of a process.



PROCESS MEMORY BOUNDARIES

Two register values, **base** and **limit**, are used to hold the starting and end memory addresses of a process.

Q: Who has access to the base and limit registers?

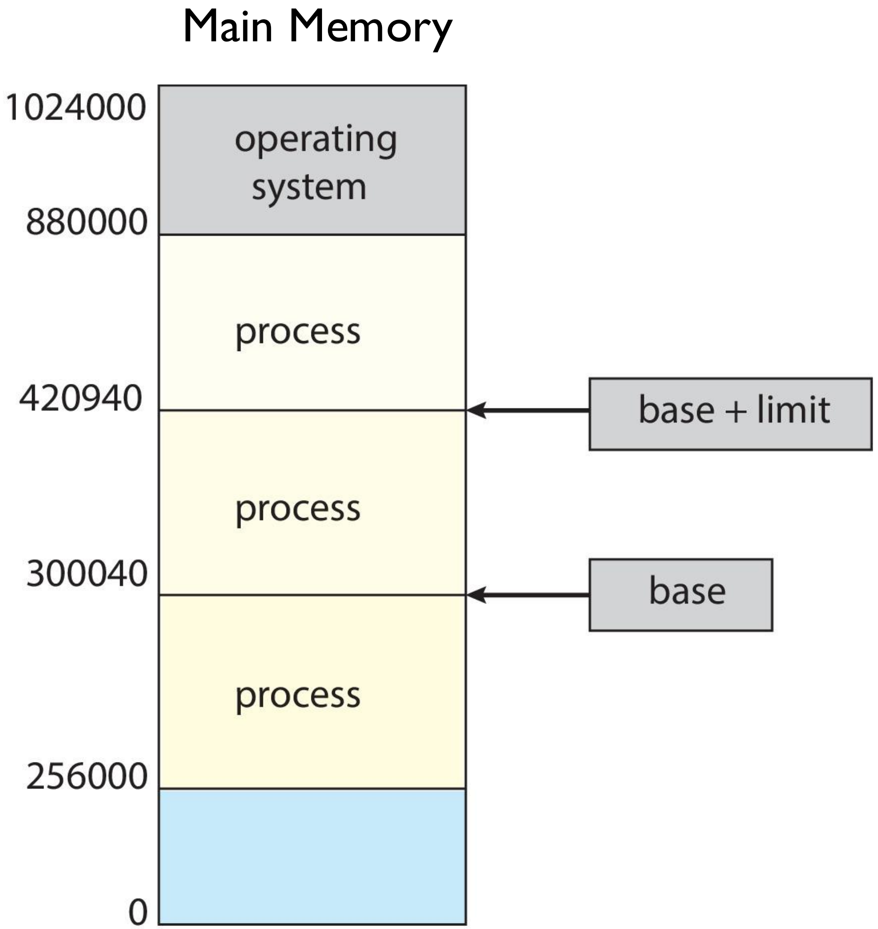


PROCESS MEMORY BOUNDARIES

Two register values, **base** and **limit**, are used to hold the starting and end memory addresses of a process.

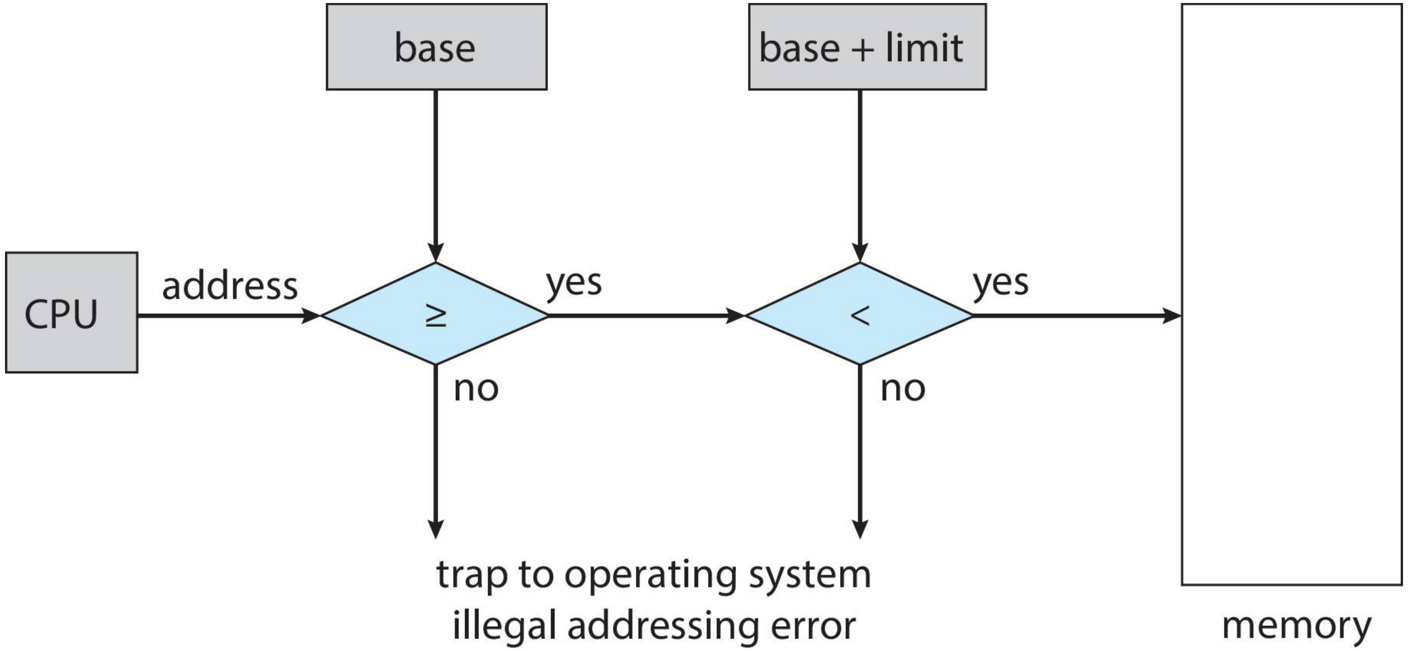
Q: Who has access to the base and limit registers?

Only the kernel (in kernel mode) can access the values of these two registers

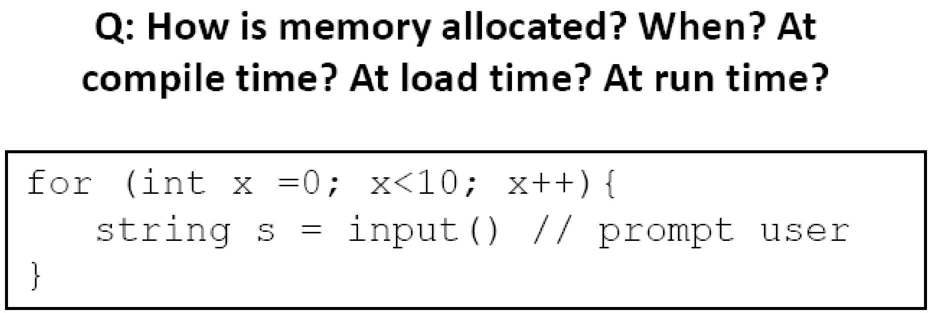


MEMORY PROTECTION

- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user.
- The instructions to loading the base and limit registers are privileged.



ADDRESS IN CODE



Q: How is memory allocated? When? At compile time? At load time? At run time?

```
for (int x =0; x<10; x++){
   string s = input() // prompt user
}
```

ADDRESS IN CODE

Q: How is memory allocated? When? At compile time? At load time? At run time?

```
for (int x =0; x<10; x++){
   string s = input() // prompt user
}
```

Q: Which of these are known about prior execution?

Q: Do the OS know how big an `int` is … how much space is required to store an `int`?

Q: Do the OS know what data the user will input?

ADDRESS IN CODE

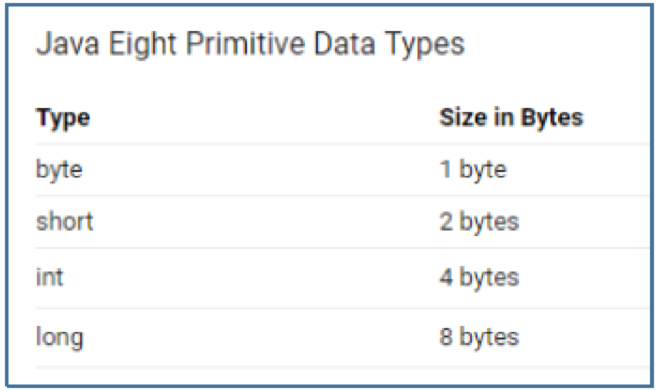
Q: How is memory allocated? When? At compile time? At load time? At run time?

```
for (int x =0; x<10; x++){
   string s = input() // prompt user
}
```

Q: Which of these are known about prior execution?

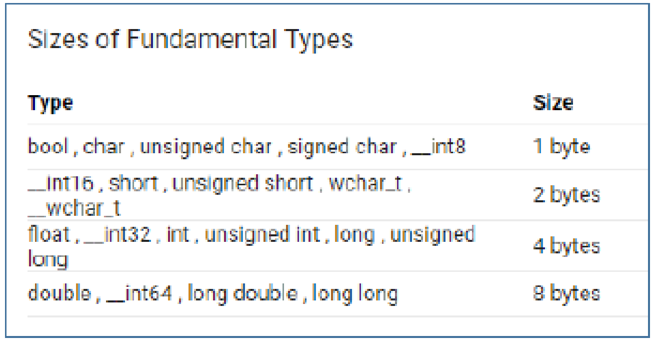
Q: Do the OS know how big an `int` is … how much space is required to store an `int`?

Q: Do the OS know what data the user will input?



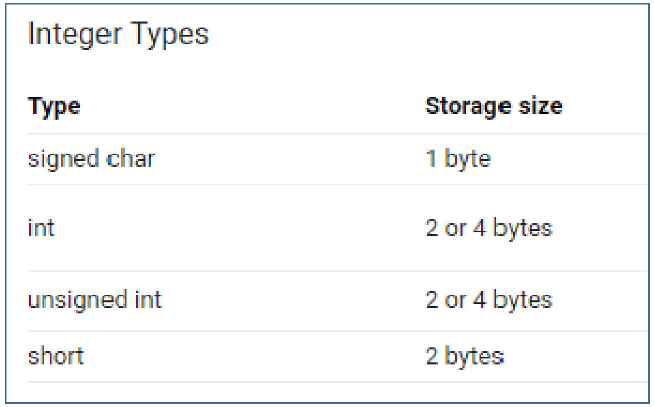
Java Eight Primitive Data Types

Type	Size in Bytes
byte	1 byte
short	2 bytes
int	4 bytes
long	8 bytes



Sizes of Fundamental Types

Type	Size
bool , char , unsigned char , signed char , __int8	1 byte
__int16 , short , unsigned short , wchar_t , __wchar_t	2 bytes
float , __int32 , int , unsigned int , long , unsigned long	4 bytes
double , __int64 , long double , long long	8 bytes

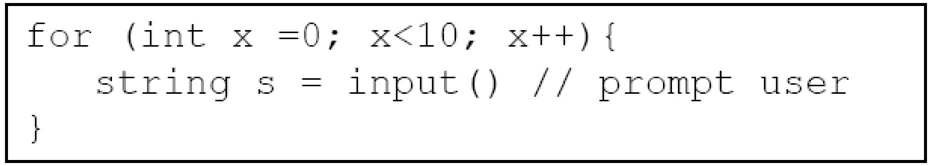


Integer Types

Type	Storage size
signed char	1 byte
int	2 or 4 bytes
unsigned int	2 or 4 bytes
short	2 bytes

ADDRESS BINDING

When you write a code, you don’t worry about memory addresses … but ultimately everything has to be bound to physical memory,.

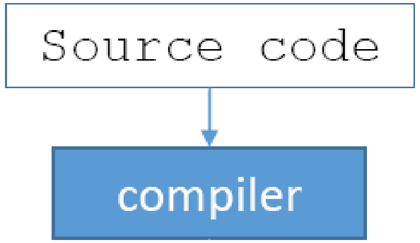


```
for (int x =0; x<10; x++){
   string s = input() // prompt user
}
```

PROGRAM LOADING

`Source code`

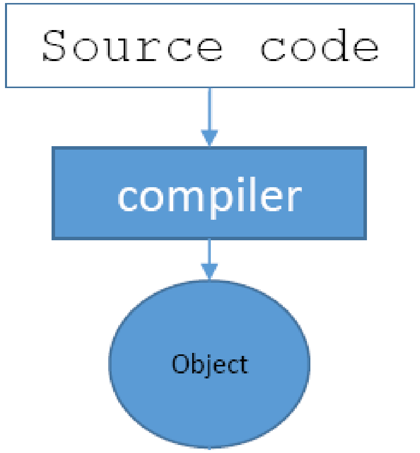
PROGRAM LOADING



Source code

compiler

PROGRAM LOADING

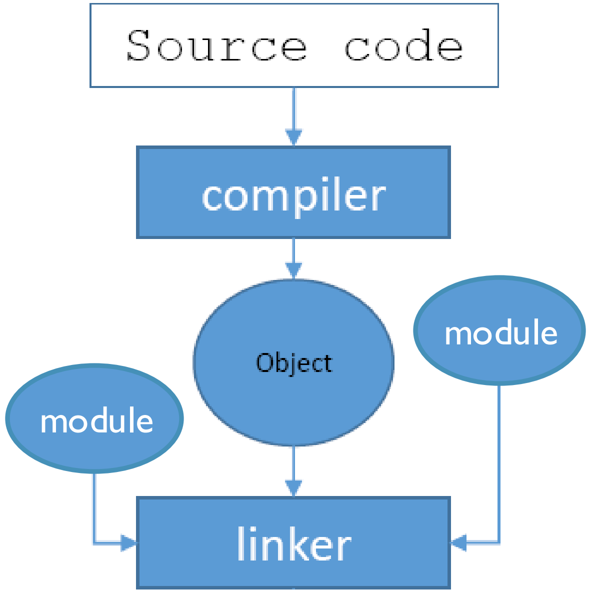


Source code

compiler

Object

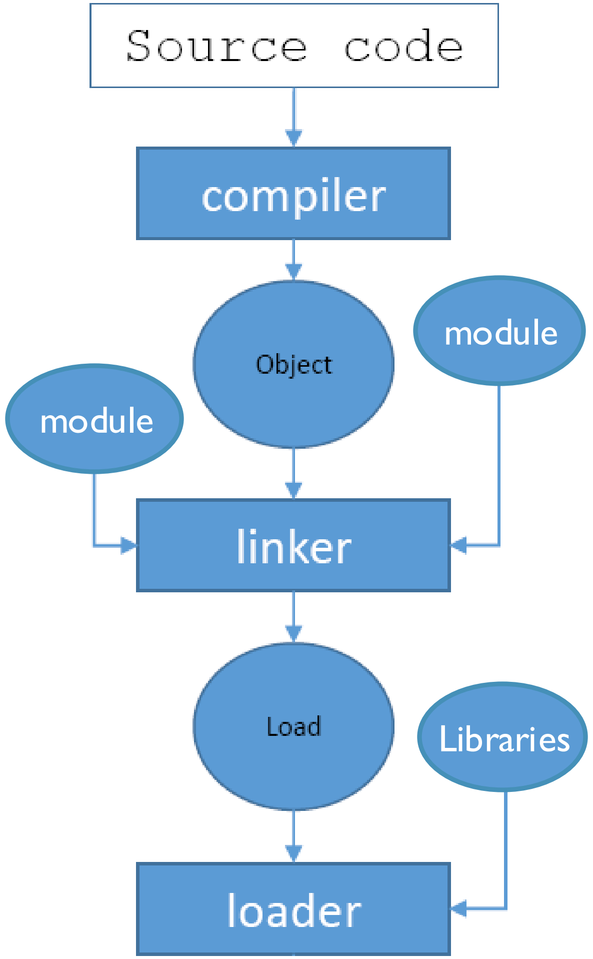
PROGRAM LOADING



- Usually a Linker is needed to incorporate other object modules which the source code is dependent on.

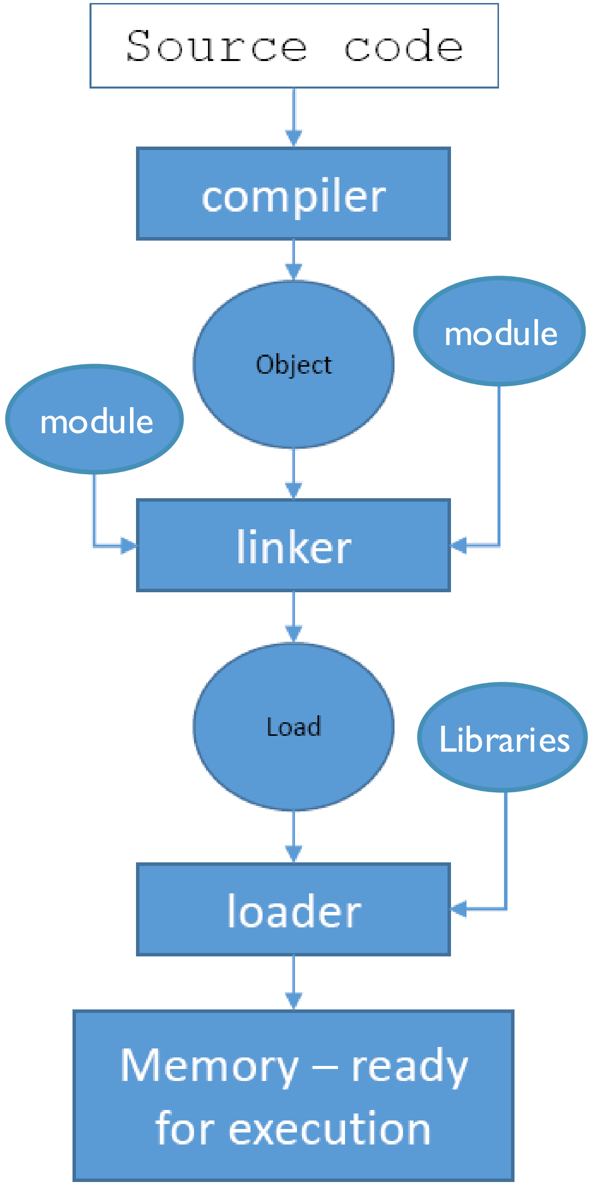
PROGRAM LOADING

- Library calls might be needed as well



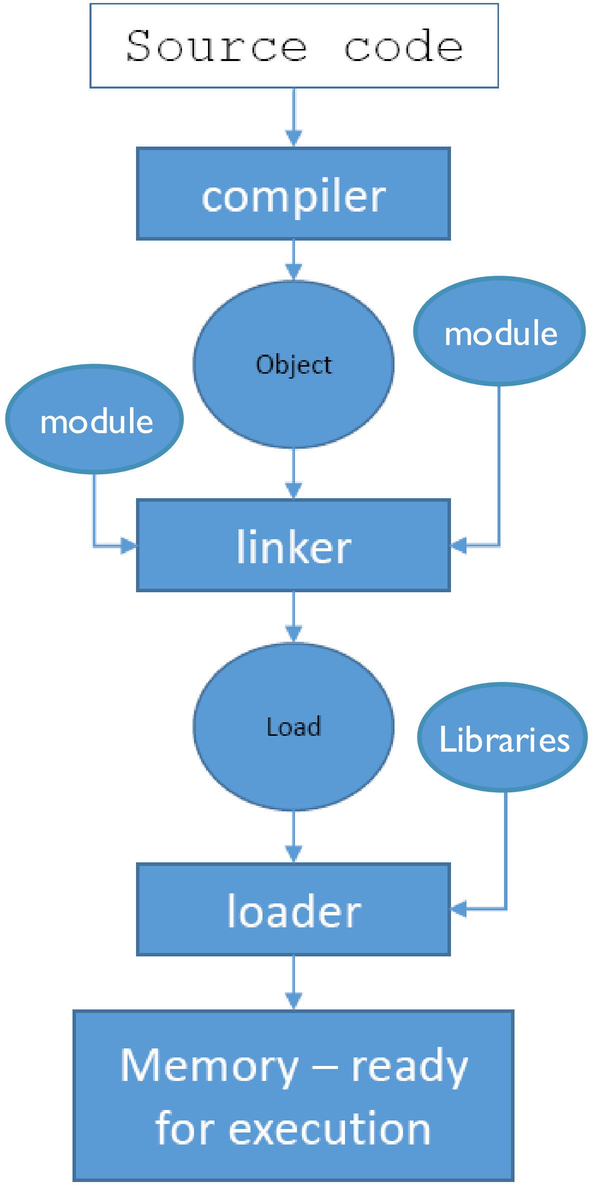
PROGRAM LOADING

- After loading, the program is in memory and ready for execution.



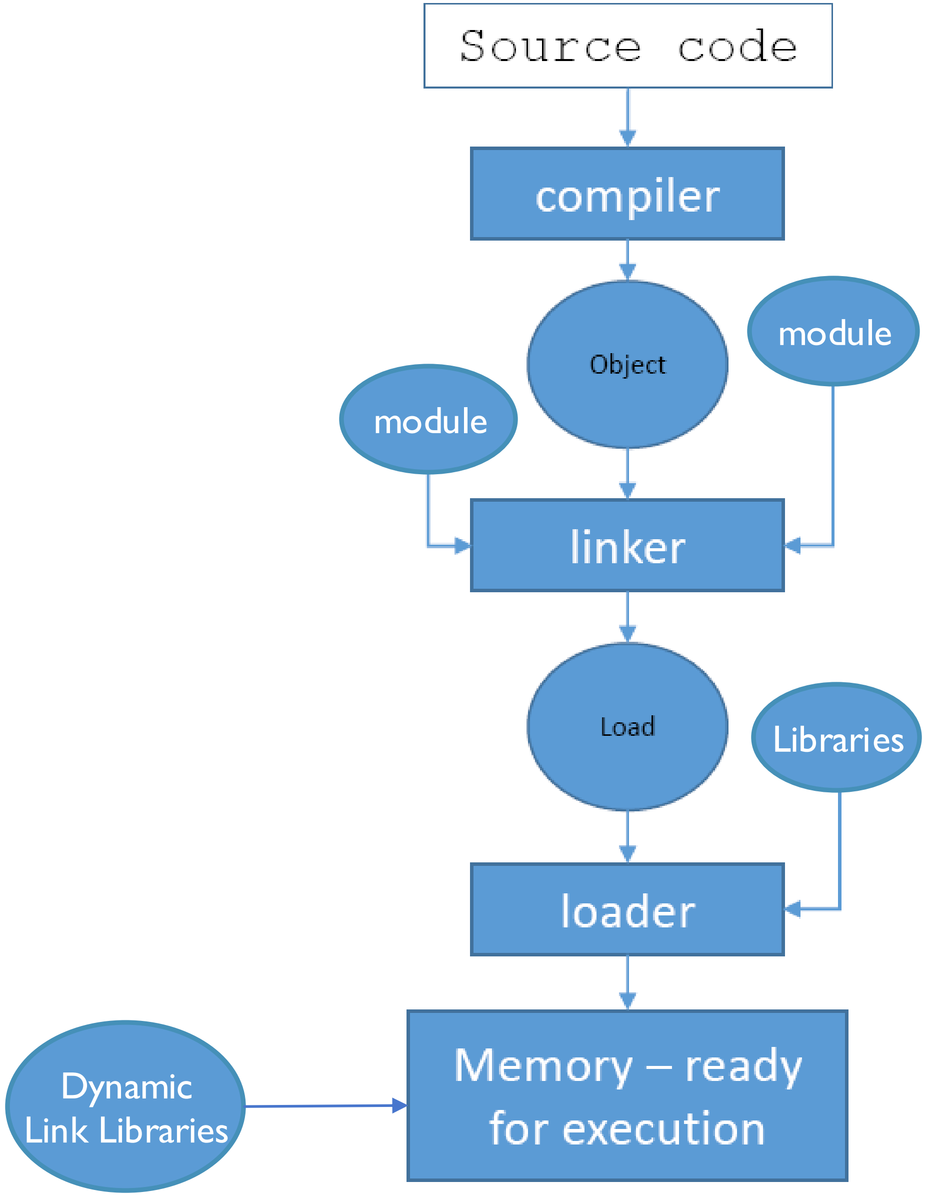
PROGRAM LOADING

- Can you load libraries during execution?



PROGRAM LOADING

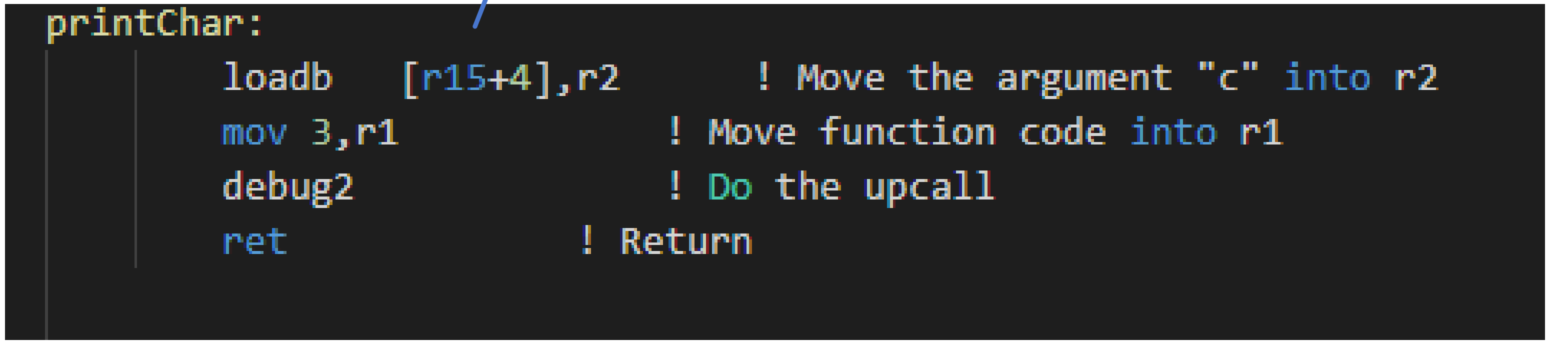
- Can you load libraries during execution?
- Yes: Dynamic Link Libraries; used extensively in modern operating systems.



ADDRESS BINDING

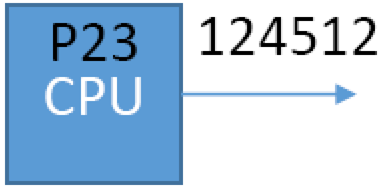
- With address binding done at runtime … the address run by each instruction is a logical address, not a physical one!

[r15+4] is not a real/physical memory address ..

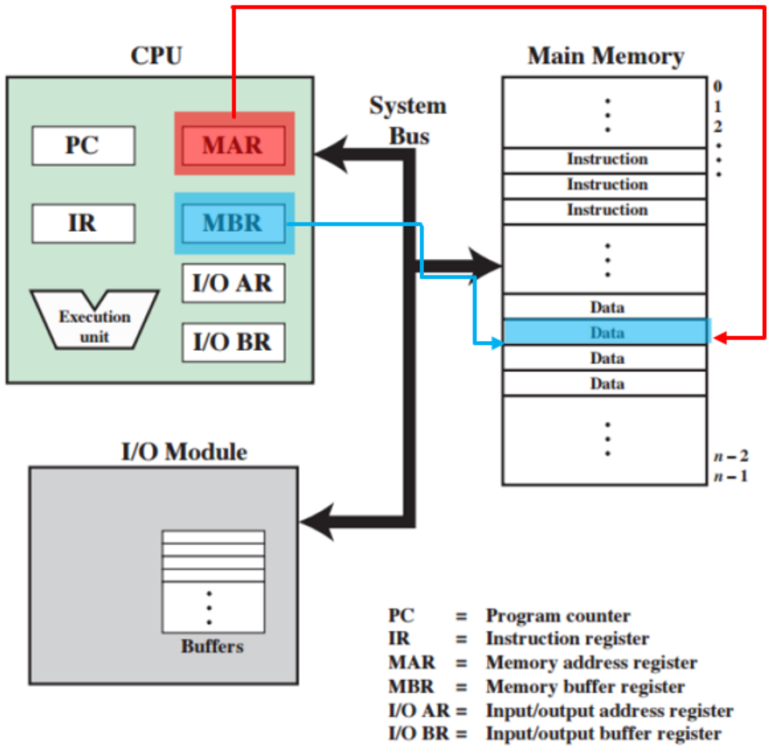


```
printChar:
        loadb   [r15+4],r2      ! Move the argument "c" into r2
        mov 3,r1                ! Move function code into r1
        debug2                  ! Do the upcall
        ret                 ! Return
```

LOGICAL TO PHYSICAL ADDRESS

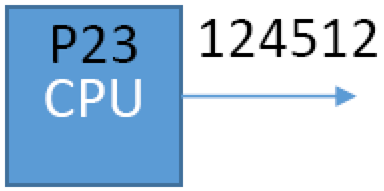


The address generated by a CPU is called a logical address, and most often referred to as virtual address

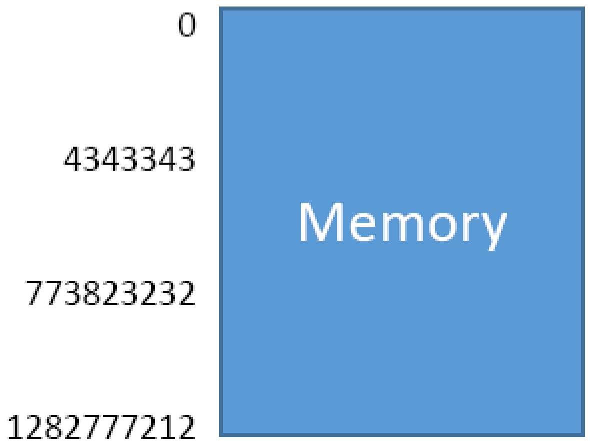


PC	=	Program counter
IR	=	Instruction register
MAR	=	Memory address register
MBR	=	Memory buffer register
I/O AR	=	Input/output address register
I/O BR	=	Input/output buffer register

LOGICAL TO PHYSICAL ADDRESS



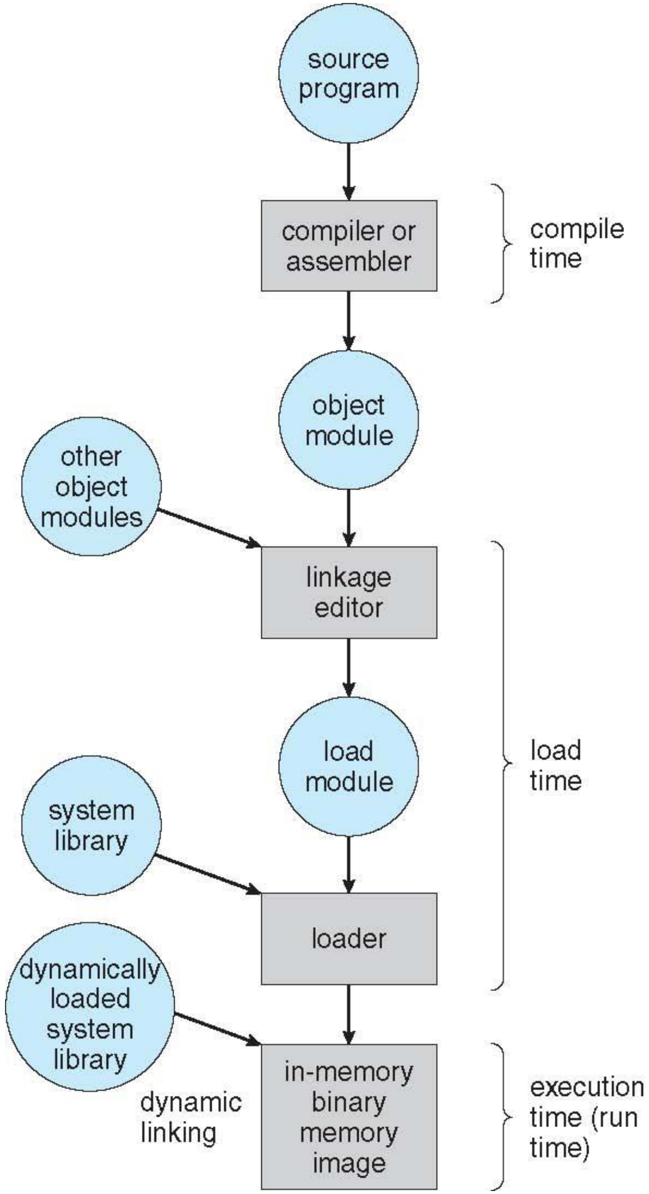
The address generated by a CPU is called a <u>logical</u> address, and most often referred to as <u>virtual</u> address



The address that the memory unit (memory) reasons about is referred to as <u>physical</u> address

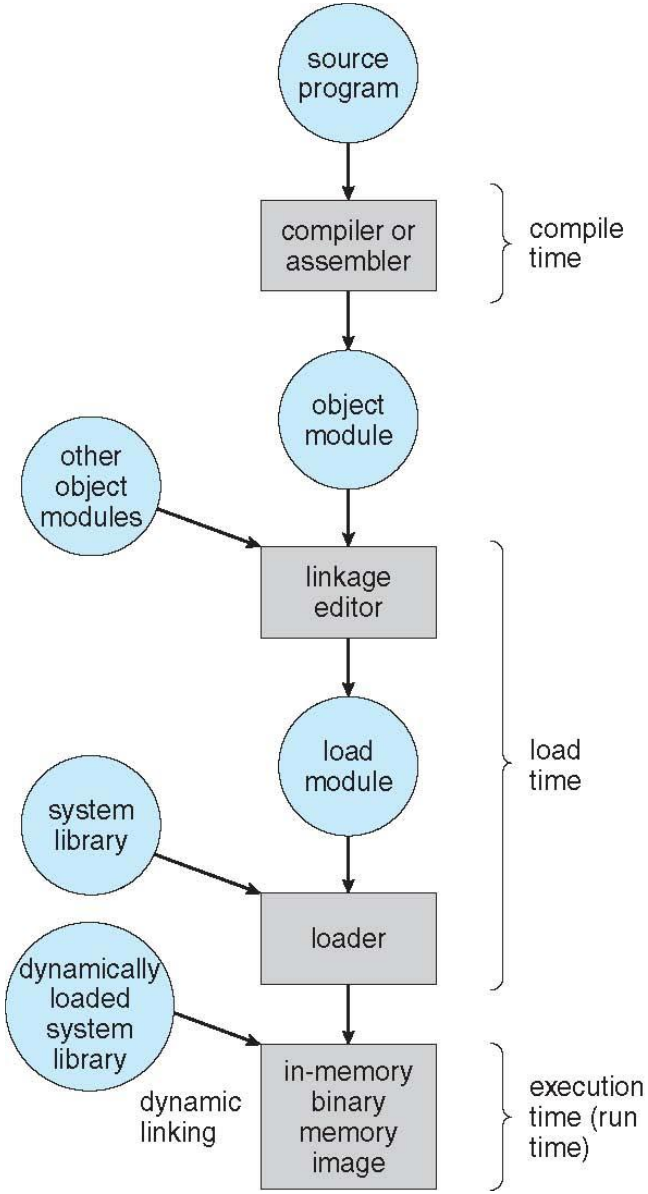
ADDRESS BINDING

- When do we bind source address values to physical memory?



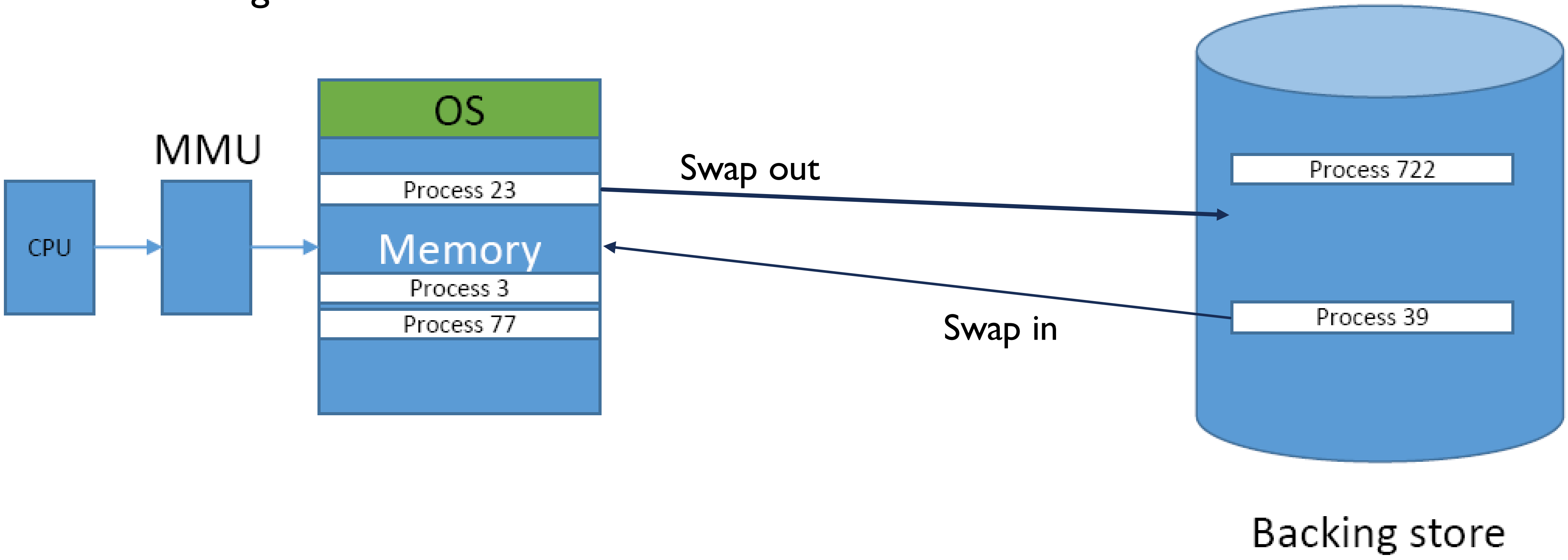
ADDRESS BINDING

- Address binding of instructions and data to memory addresses can happen at three different stages
  - **Compile time**: If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
  - **Load time**: Must generate **relocatable code** if memory location is not known at compile time
  - **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another
    - Need hardware support for address maps (e.g., base and limit registers)



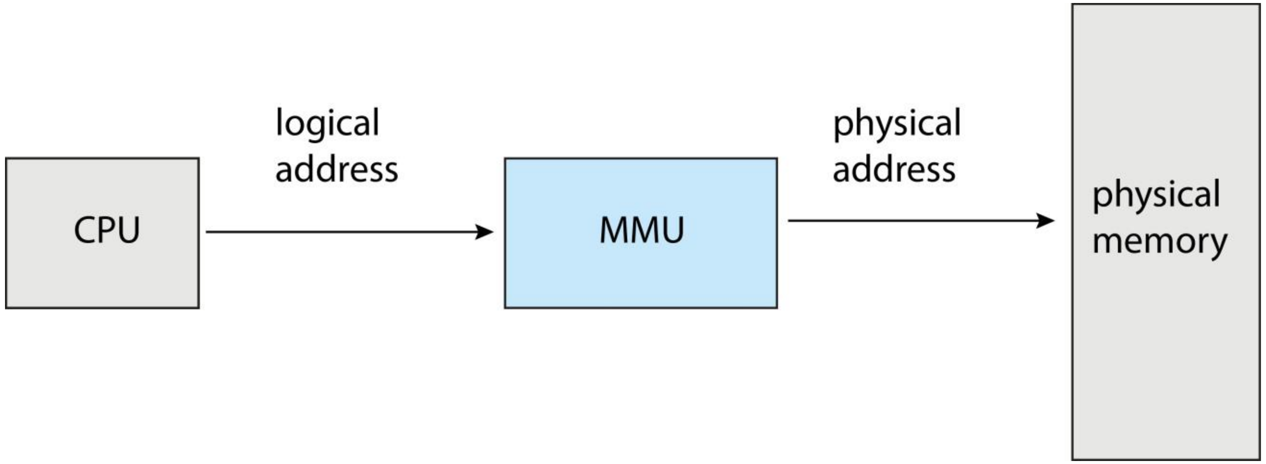
SWAPPING

- When process are swapped back, they don't necessary sit in the same place in memory!
- This is another reason why runtime address binding is beneficial.

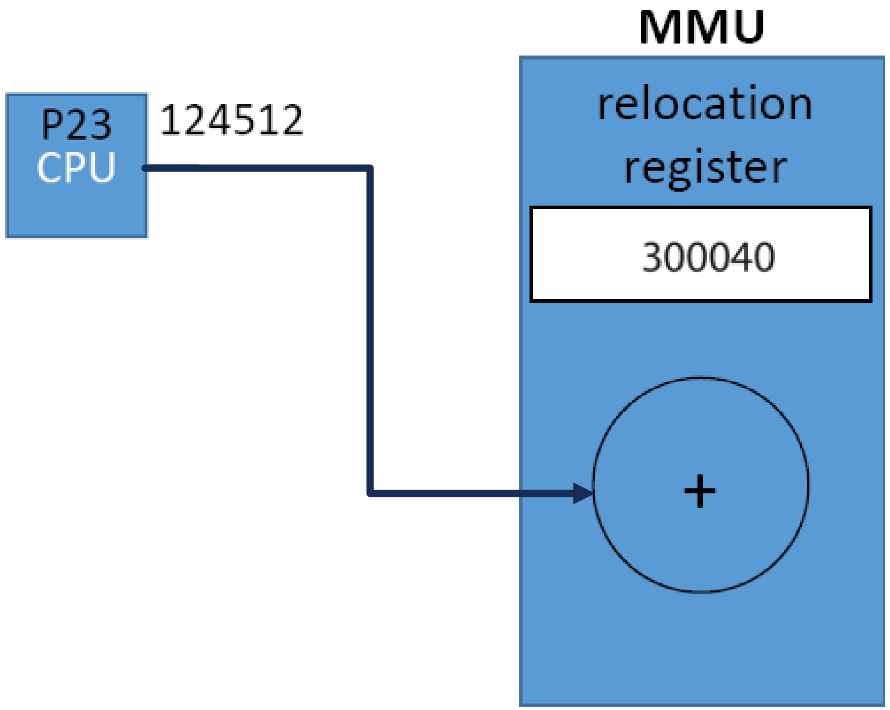


ADDRESS TRANSLATION

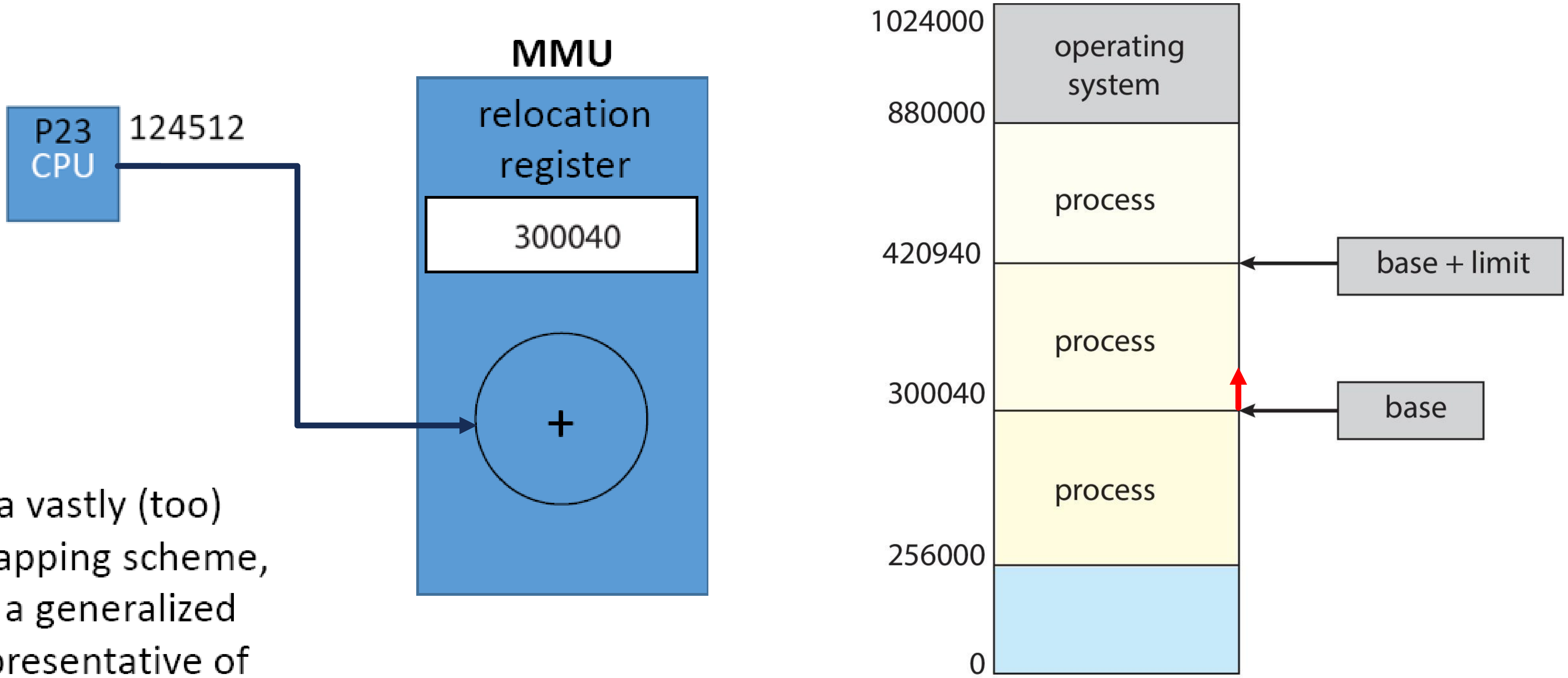
- Hardware device that at run time maps virtual to physical address



MEMORY MANAGEMENT UNIT

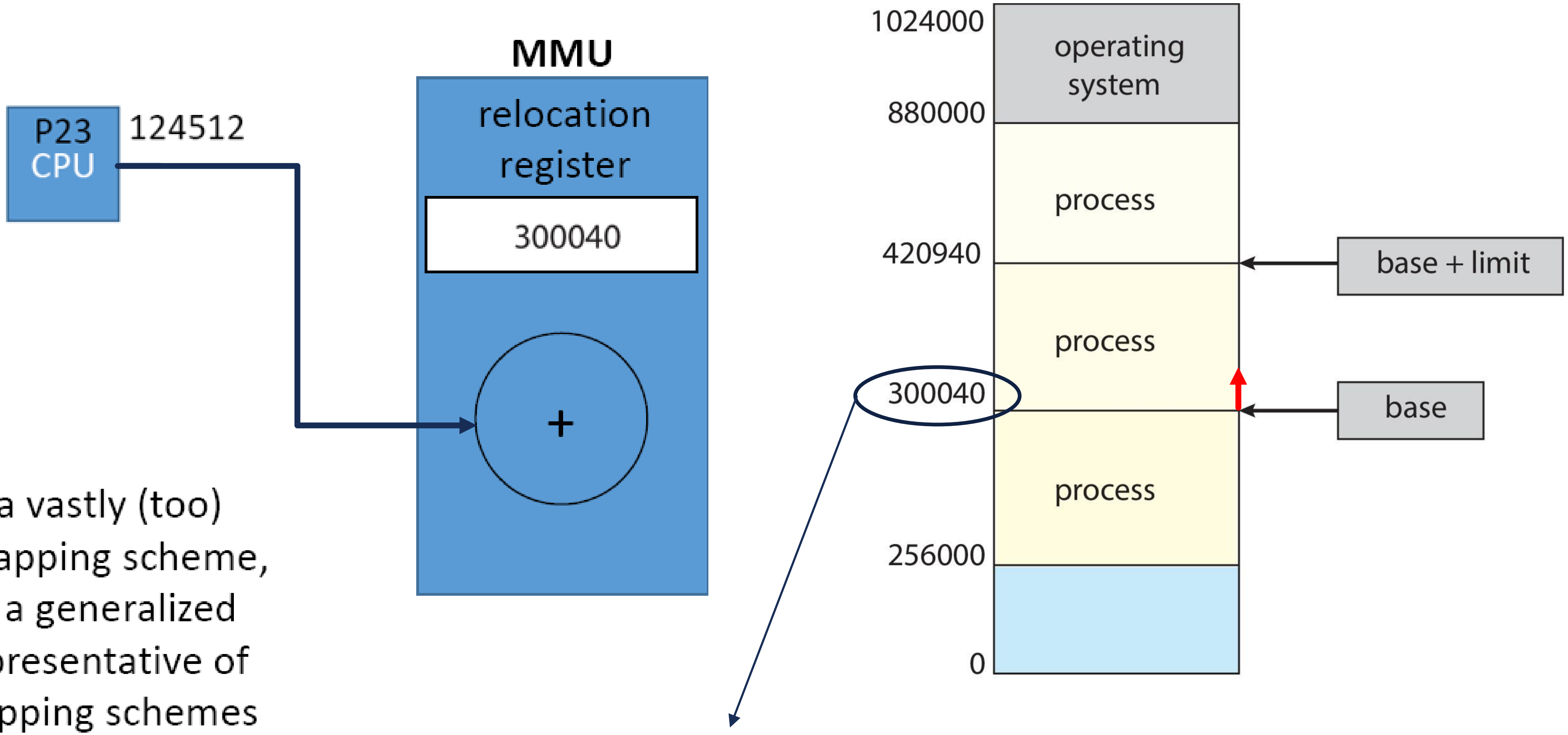


MEMORY MANAGEMENT UNIT



This is a vastly (too) simple mapping scheme, but it is a generalized form representative of most mapping schemes

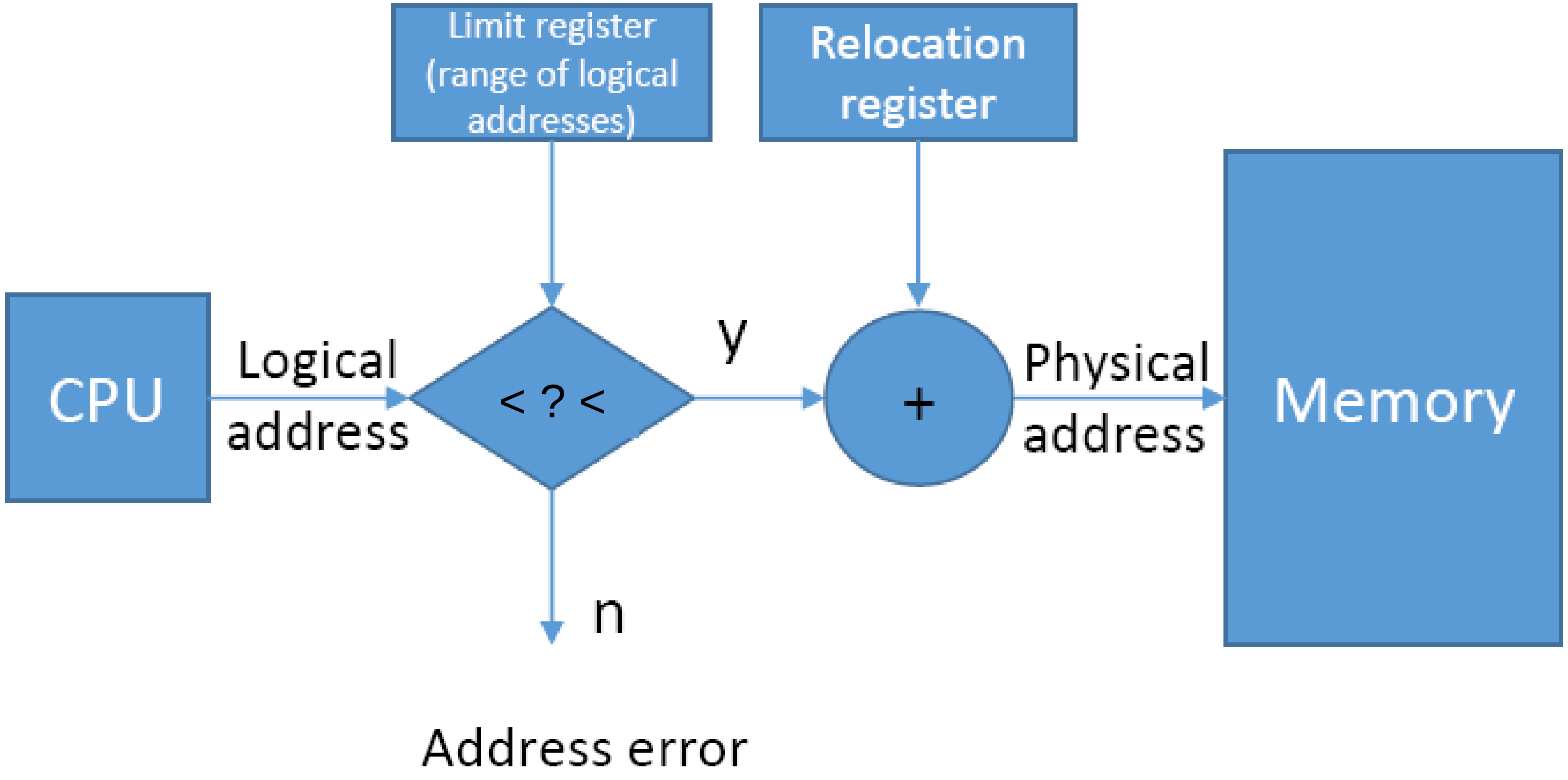
MEMORY MANAGEMENT UNIT



This is a vastly (too) simple mapping scheme, but it is a generalized form representative of most mapping schemes

This could be logical address of '0'

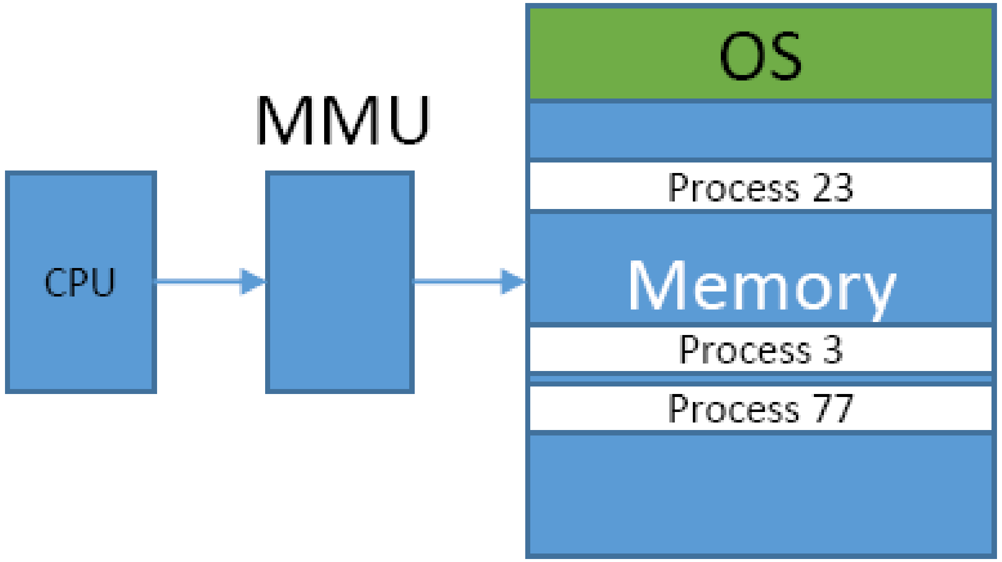
MEMORY PROTECTION AND TRANSLATION



MEMORY FORM

Regardless of how the MMU maps a logical/virtual address to a physical one …

Q: What is one assumption we've made about how processes fit into memory?

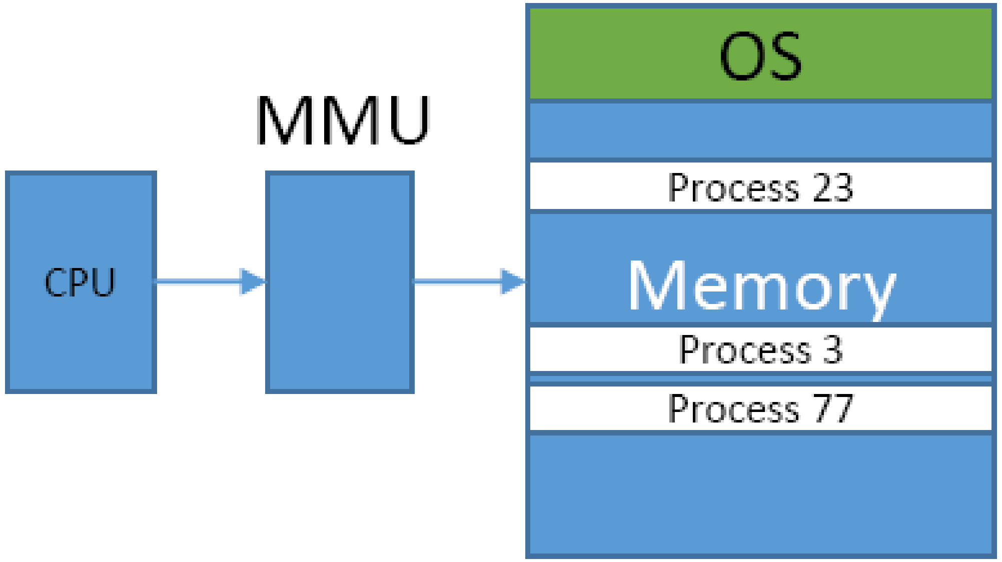


MEMORY ALLOCATION

Regardless of how the MMU maps a logical/virtual address to a physical one …

Q: What is one assumption we've made about how processes fit into memory?

<u>Contiguous</u> memory allocation



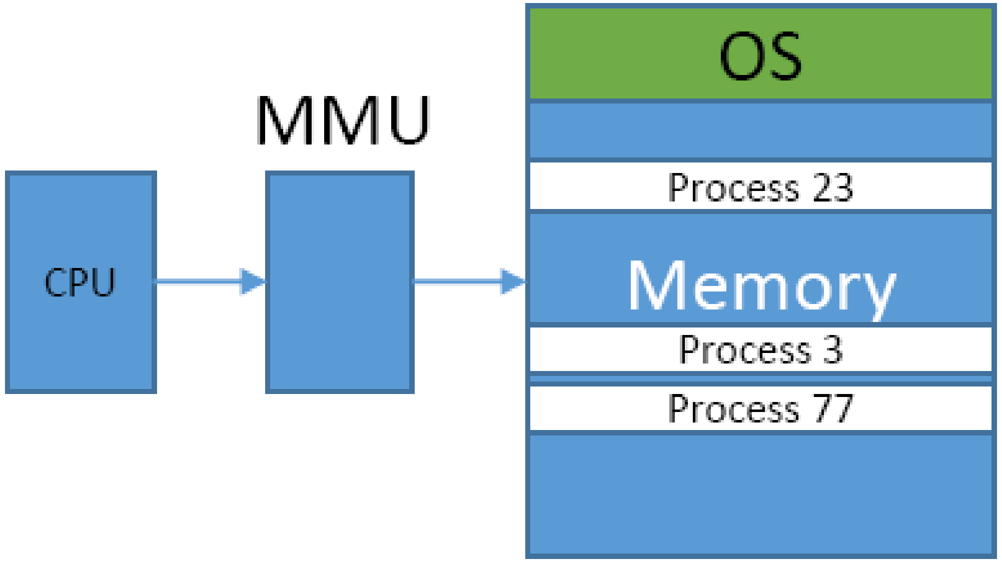
MEMORY ALLOCATION

Regardless of how the MMU maps a logical/virtual address to a physical one …

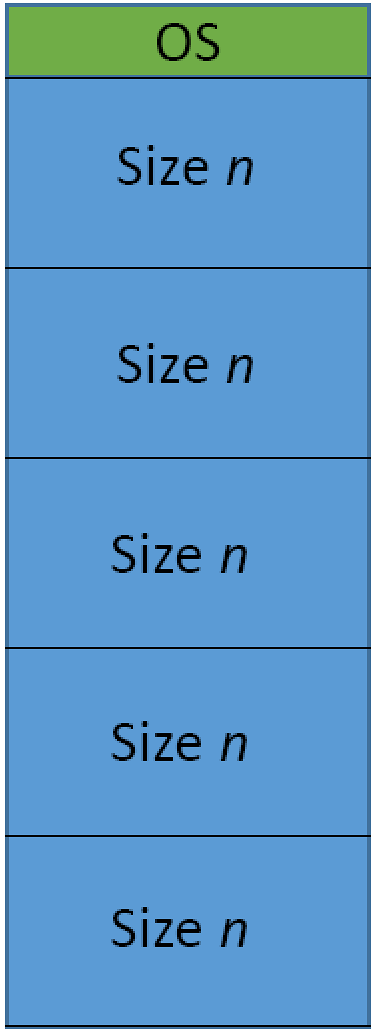
Q: What is one assumption we've made about how processes fit into memory?

<u>Contiguous</u> memory allocation

- Memory Allocation algorithms
- Memory fragmentation

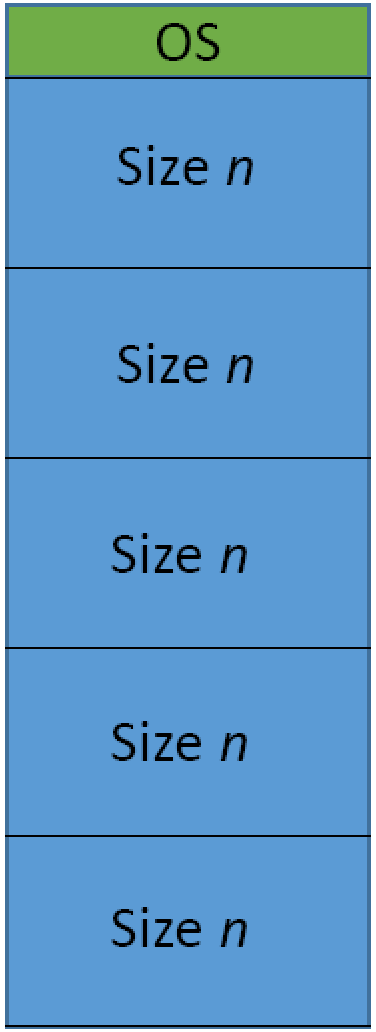


MEMORY ALLOCATION



The earliest partition methods relied on fixed-sized partitions of size *n*

MEMORY ALLOCATION



OS

Size *n*

Size *n*

Size *n*

Size *n*

Size *n*

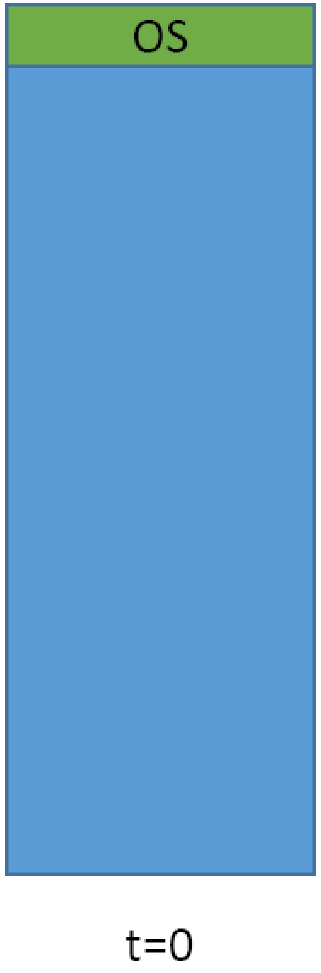
The earliest partition methods relied on fixed-sized partitions of size *n*

Assume a Process of size K

Q: What are the advantages of this approach?

Q: What are the disadvantages of this approach?

MEMORY ALLOCATION



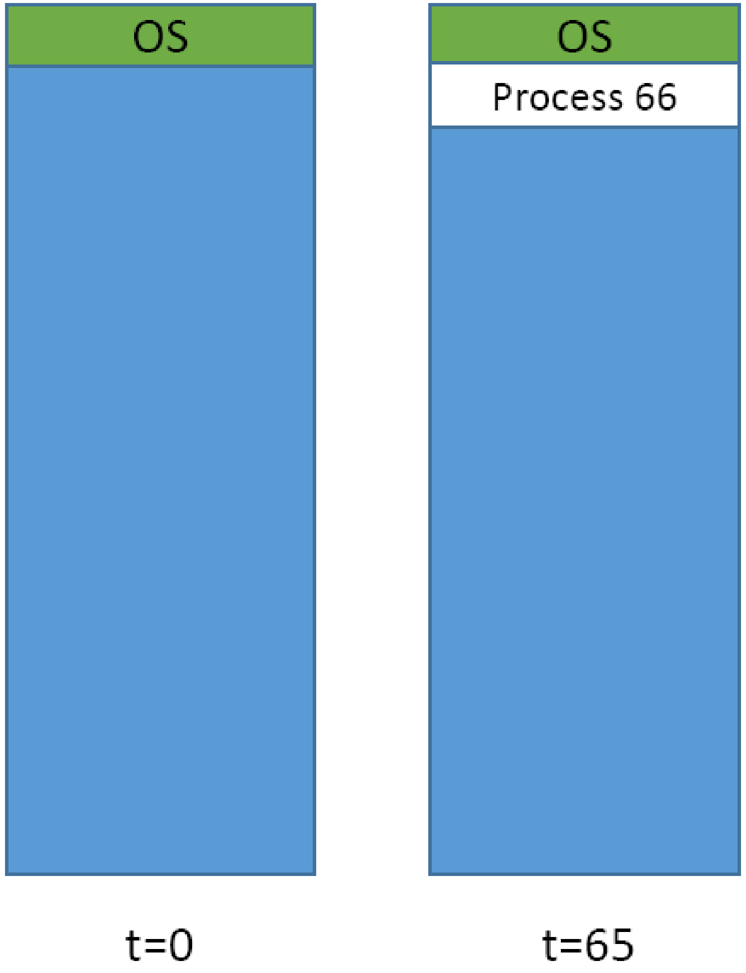
Assuming variable partition size
We just allocate enough memory for the process

At t=0, no process is in memory
... and process 66 is ready

Process 66
Size : 7

Q: Where should it be placed in memory?

MEMORY ALLOCATION

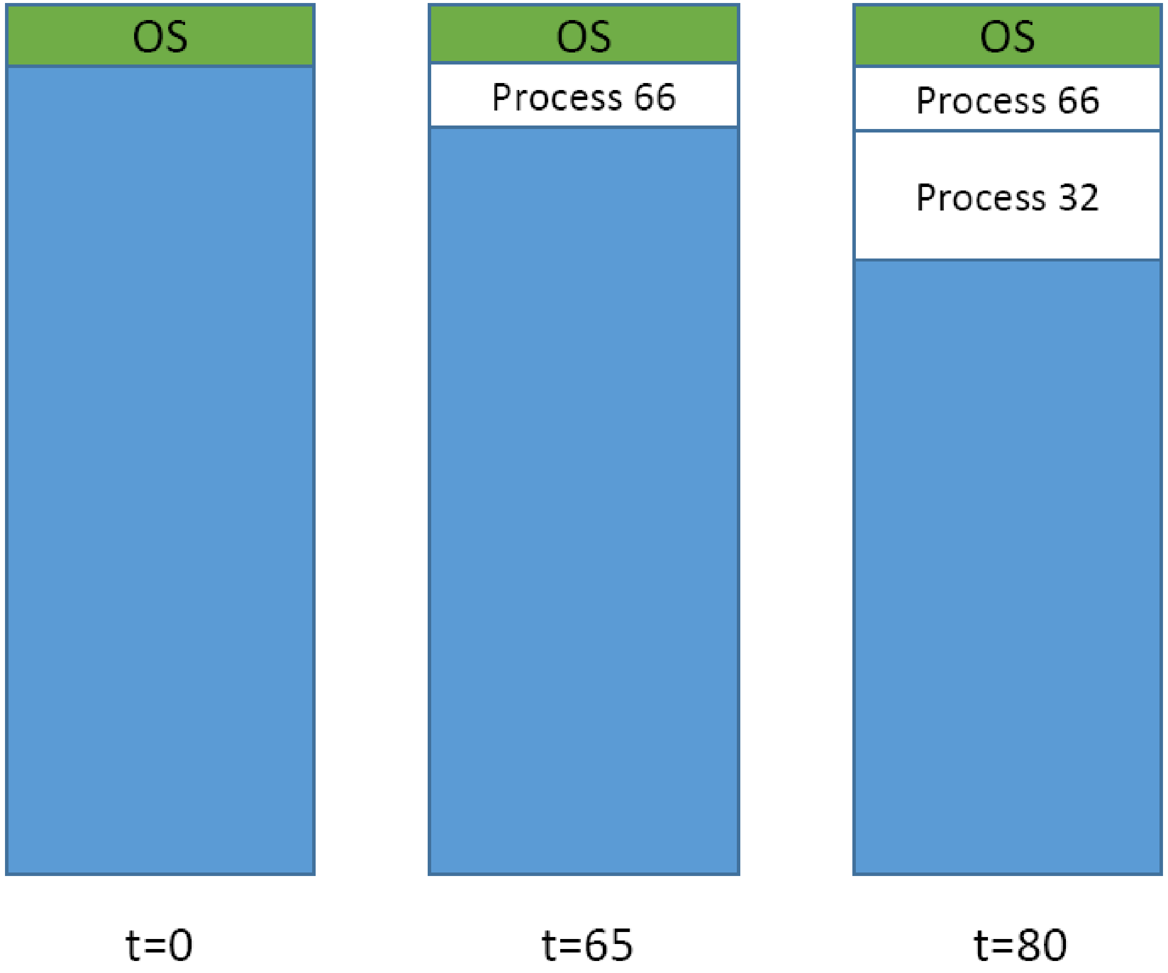


At t=65, process 66 is still running, and another process is ready …

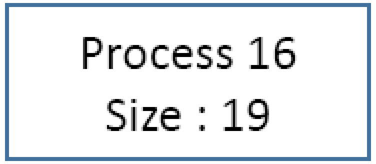
Process 32
Size : 12

Q: Where should it be placed?

MEMORY ALLOCATION

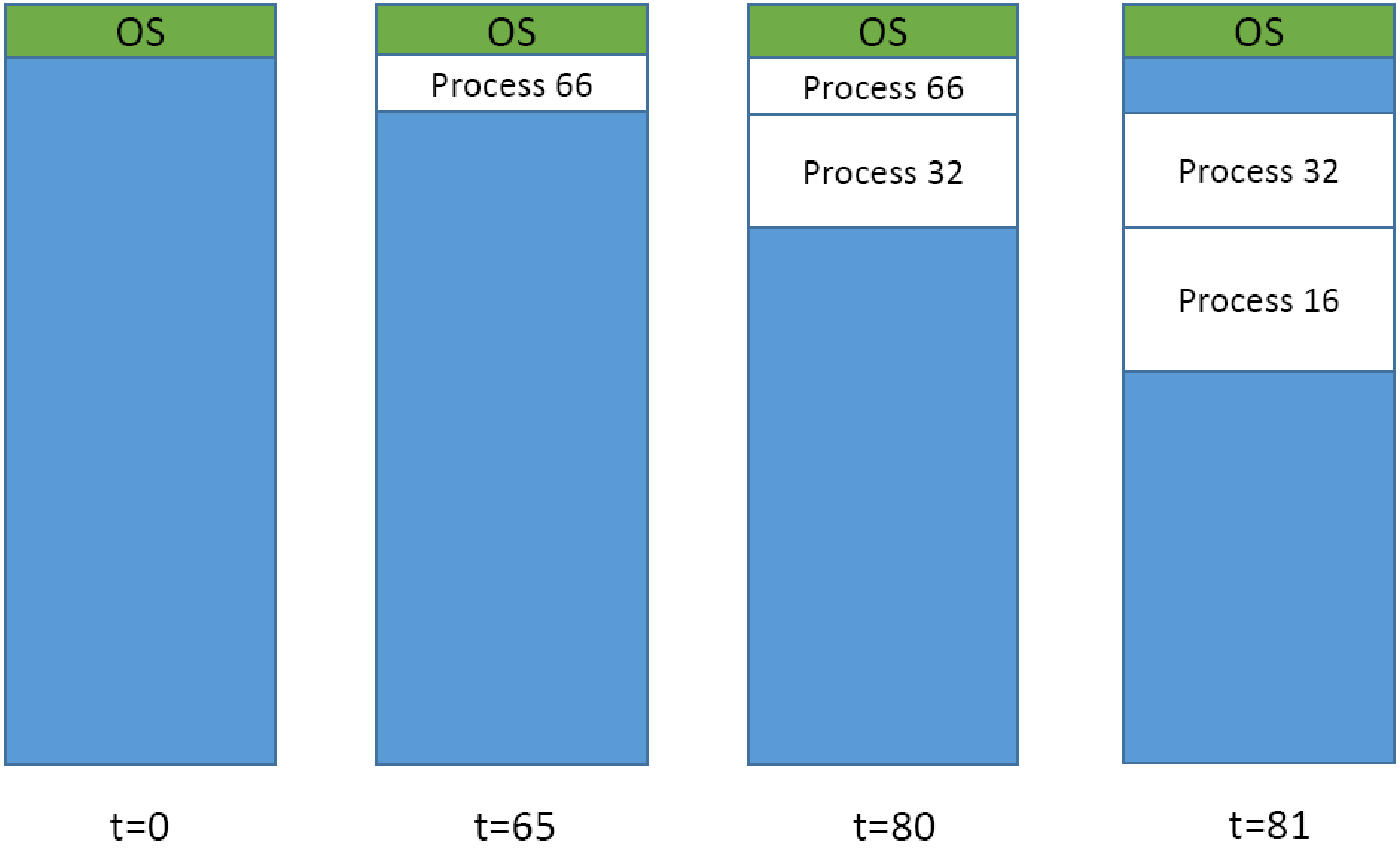


At t=80, process 66 completes, and at t=81, process 16 is ready

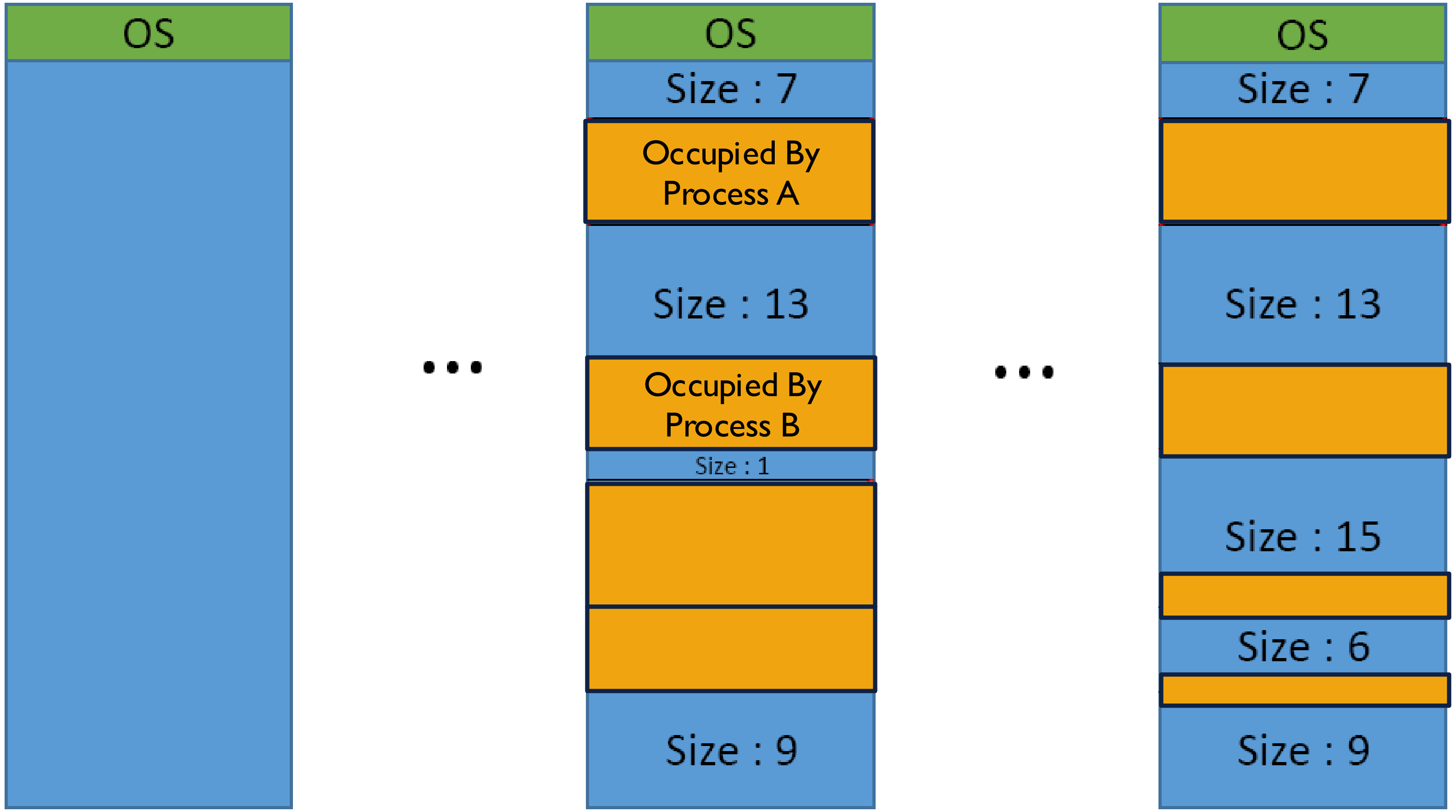


Q: Where should it be placed?

MEMORY ALLOCATION

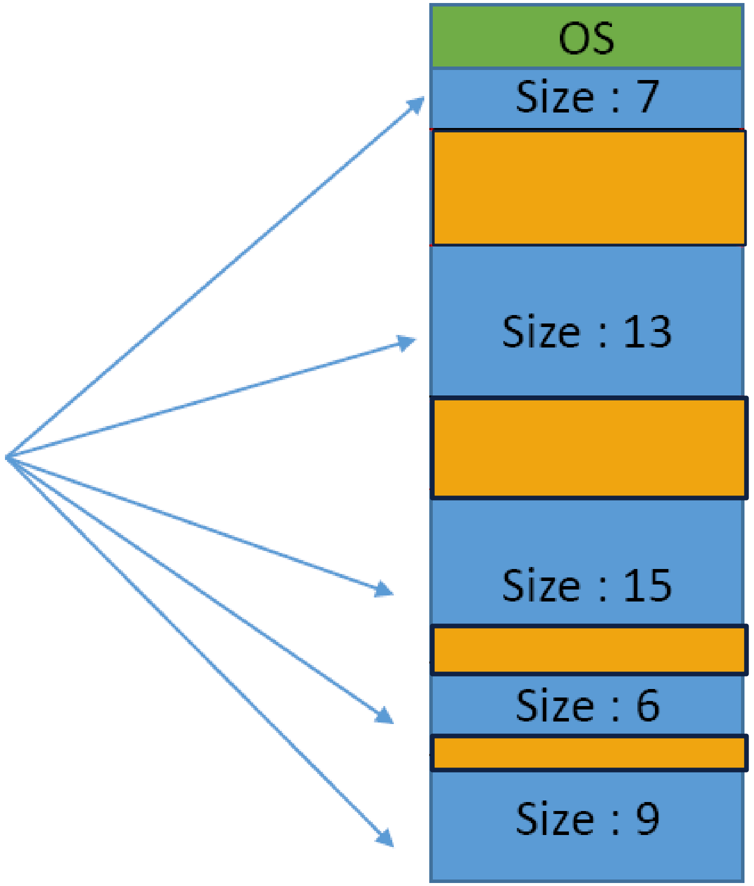


MEMORY ALLOCATION



MEMORY HOLES

Available slots are called memory holes

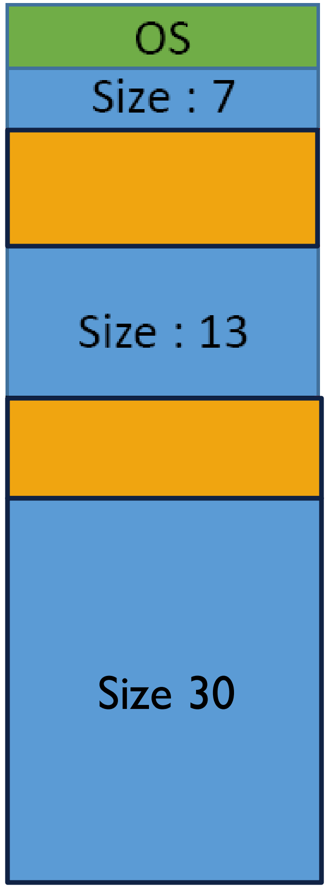


MEMORY ALLOCATION STRATEGY

Process 77
Size : 8

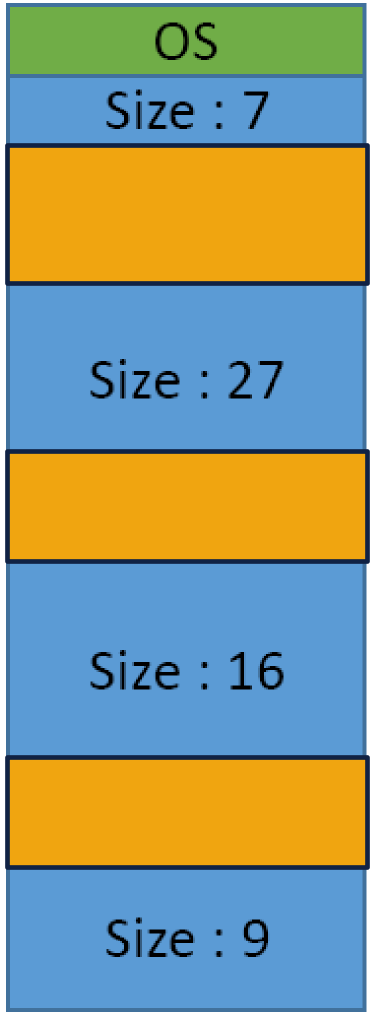
At time t, process 77 is ready, and the OS tries to place it into memory …

Task : enumerate three possible allocation strategies … where should Process 77 be placed?



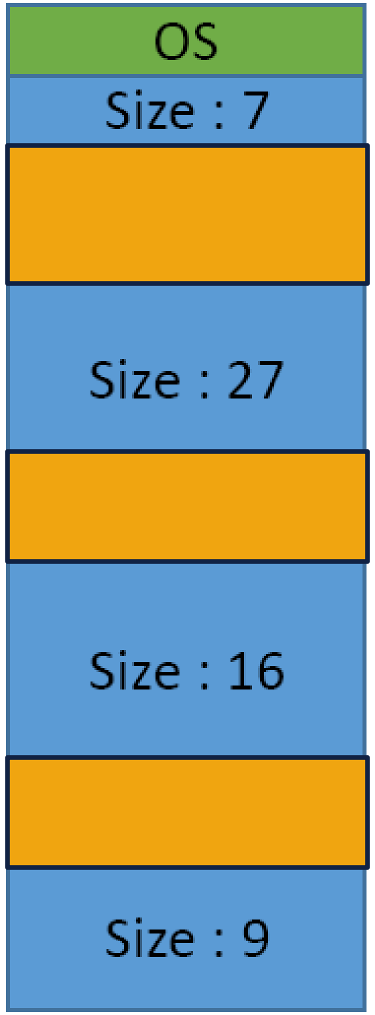
MEMORY ALLOCATION STRATEGY

- Best Fit: Place the process into the memory hole closest to the size required.



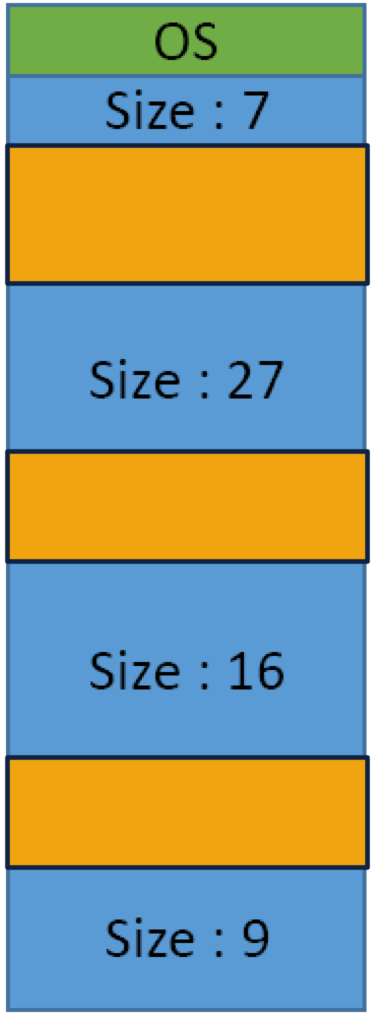
MEMORY ALLOCATION STRATEGY

- Best Fit: Place the process into the memory hole closest to the size required.
- Worst Fit: Place the process into the largest memory hole.

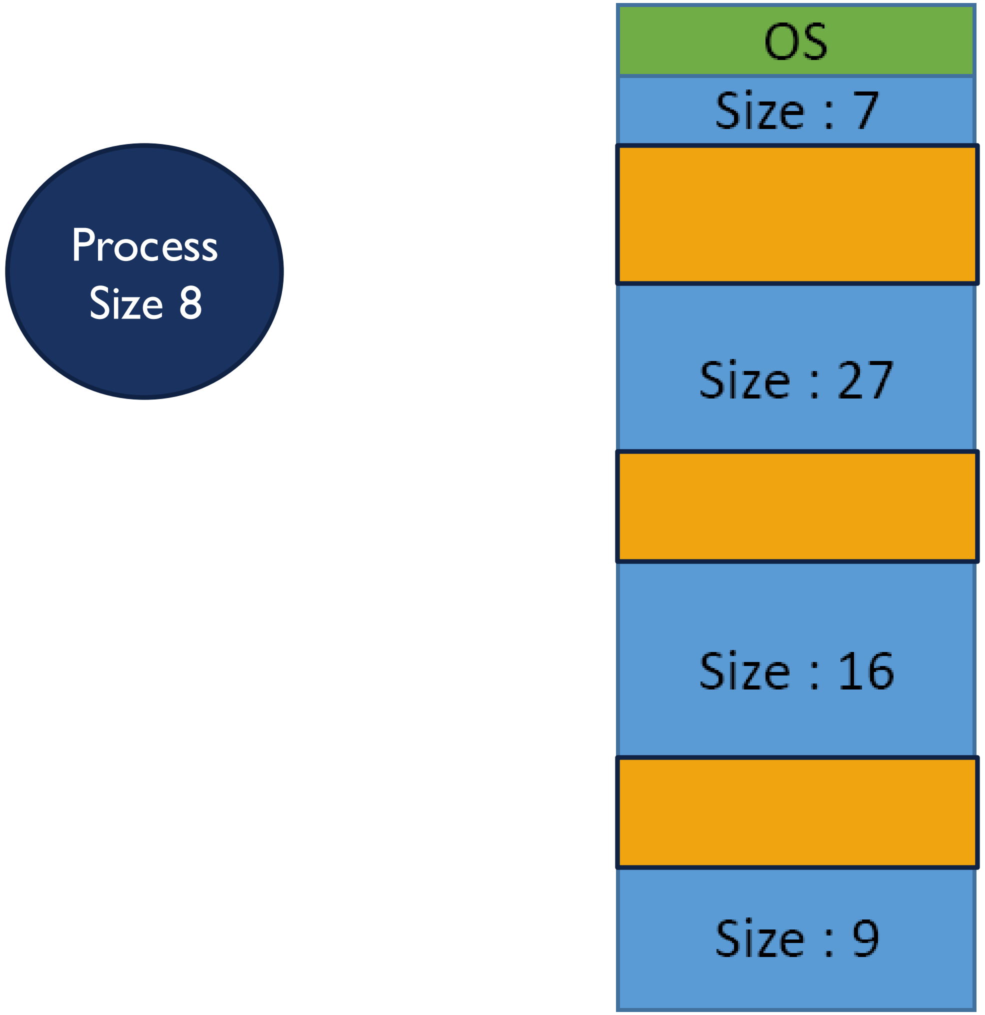


MEMORY ALLOCATION STRATEGY

- Best Fit: Place the process into the memory hole closest to the size required.
- Worst Fit: Place the process into the largest memory hole.
- First Fit: Place the process into the first memory space that can hold it.
- What are the advantages of each?

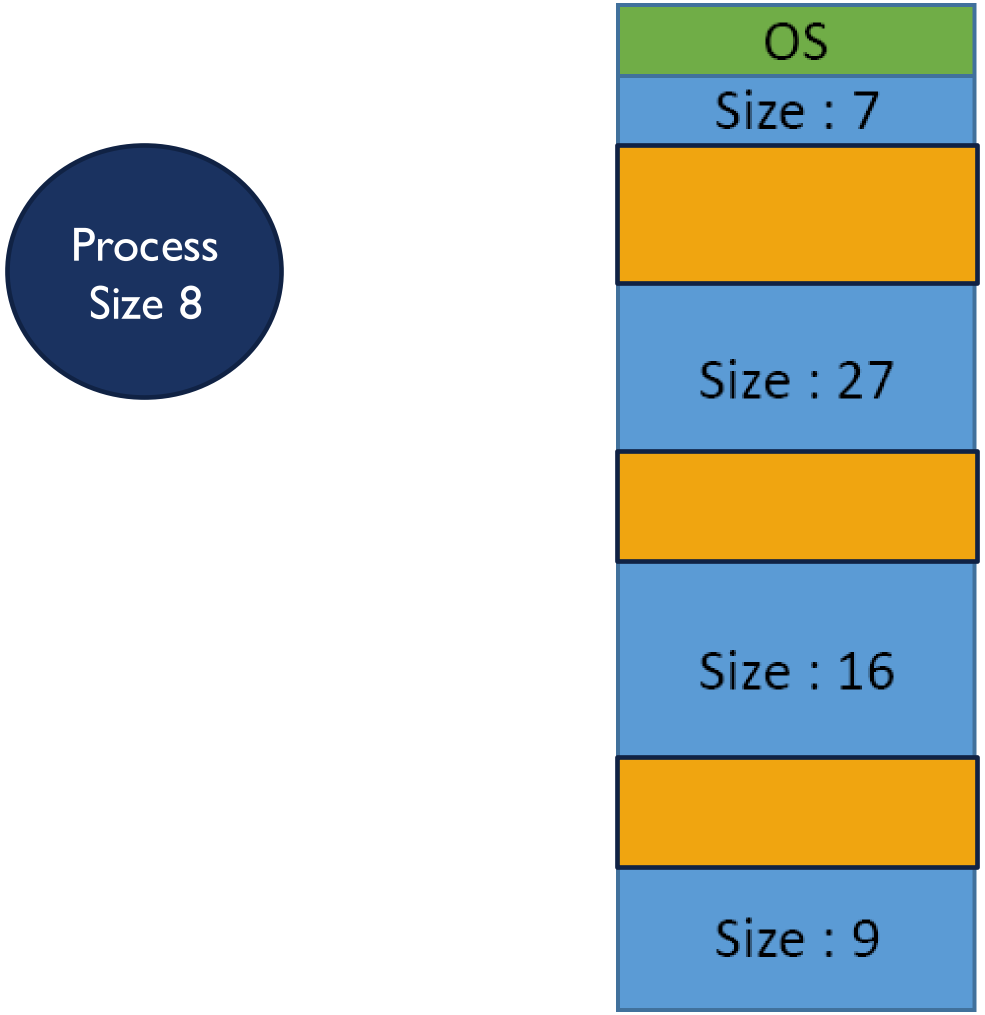


MEMORY ALLOCATION ALGORITHM



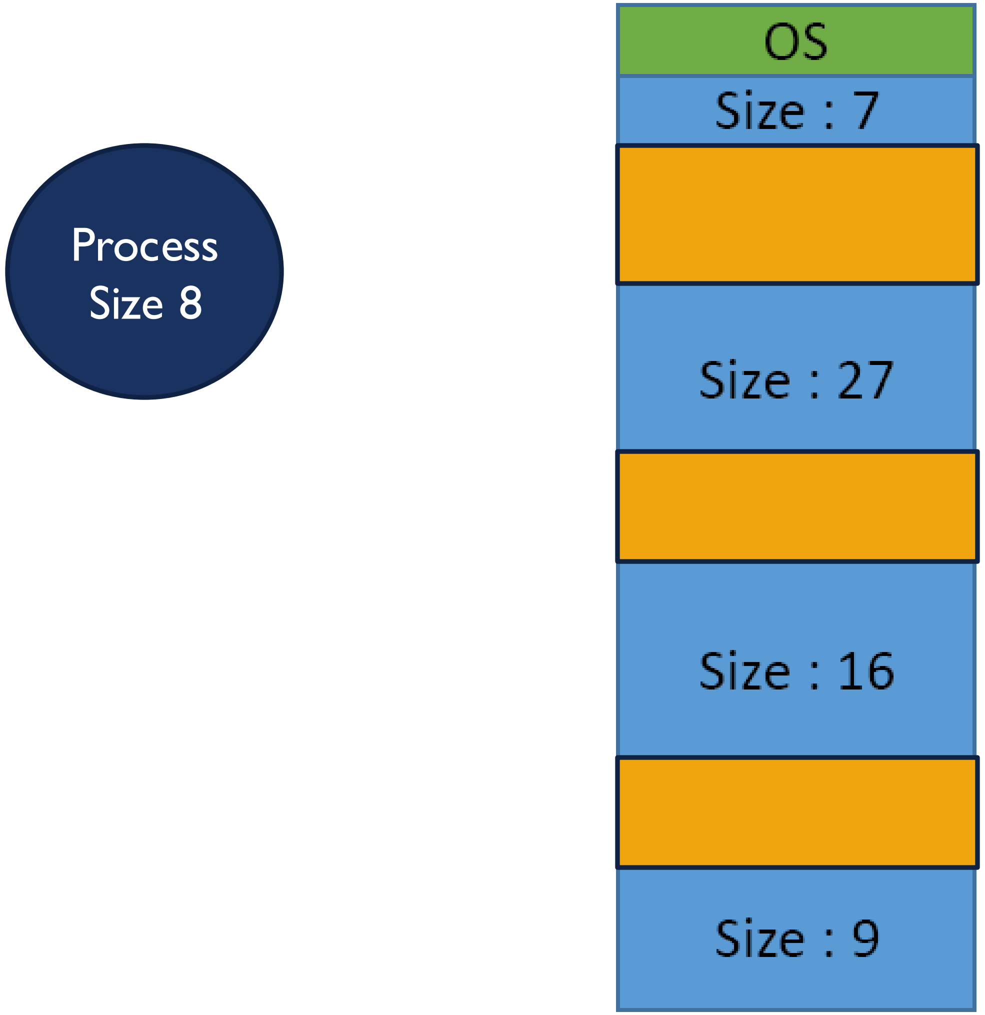
FIRST FIT

- First Fit: Place the process into the first memory space that can hold it.



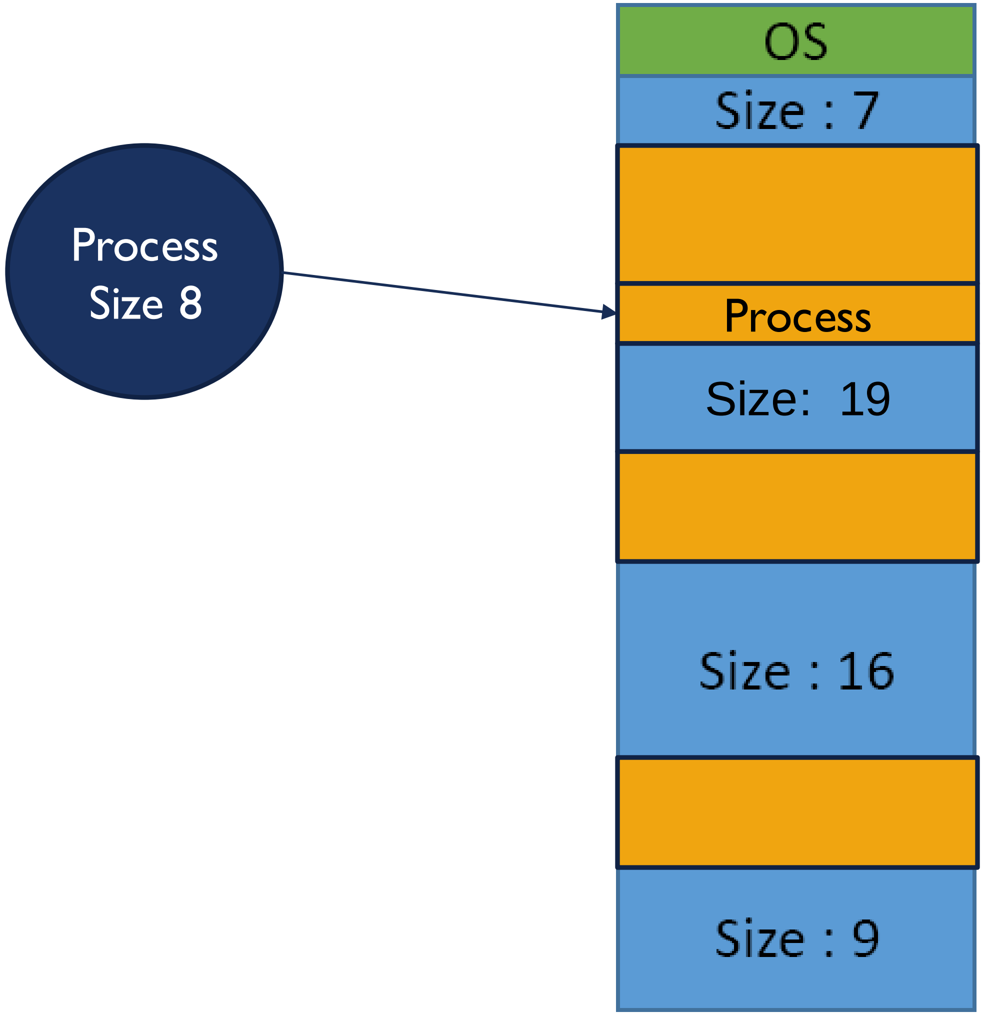
FIRST FIT

- First Fit: Place the process into the first memory space that can hold it.
- Fastest: No need to search through all memory.

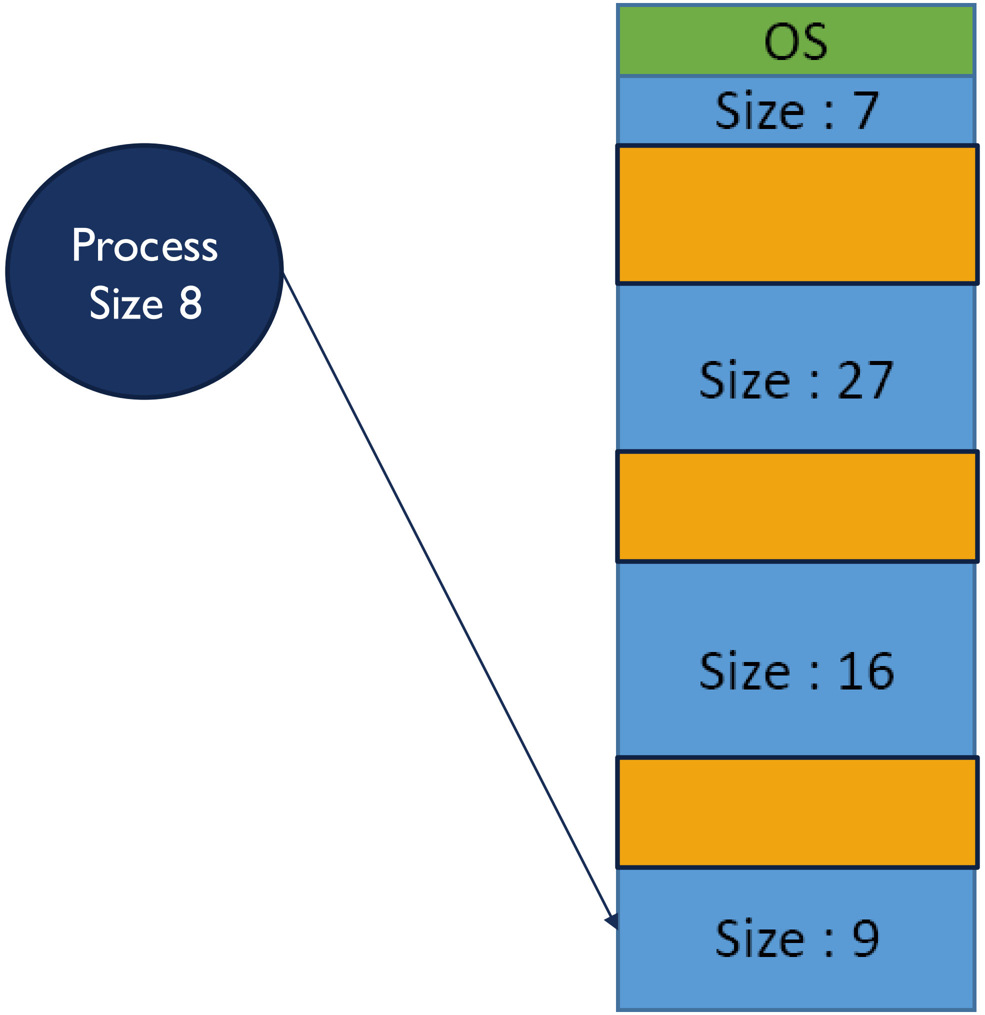


FIRST FIT

- First Fit: Place the process into the first memory space that can hold it.
- Fastest: No need to search through all memory.

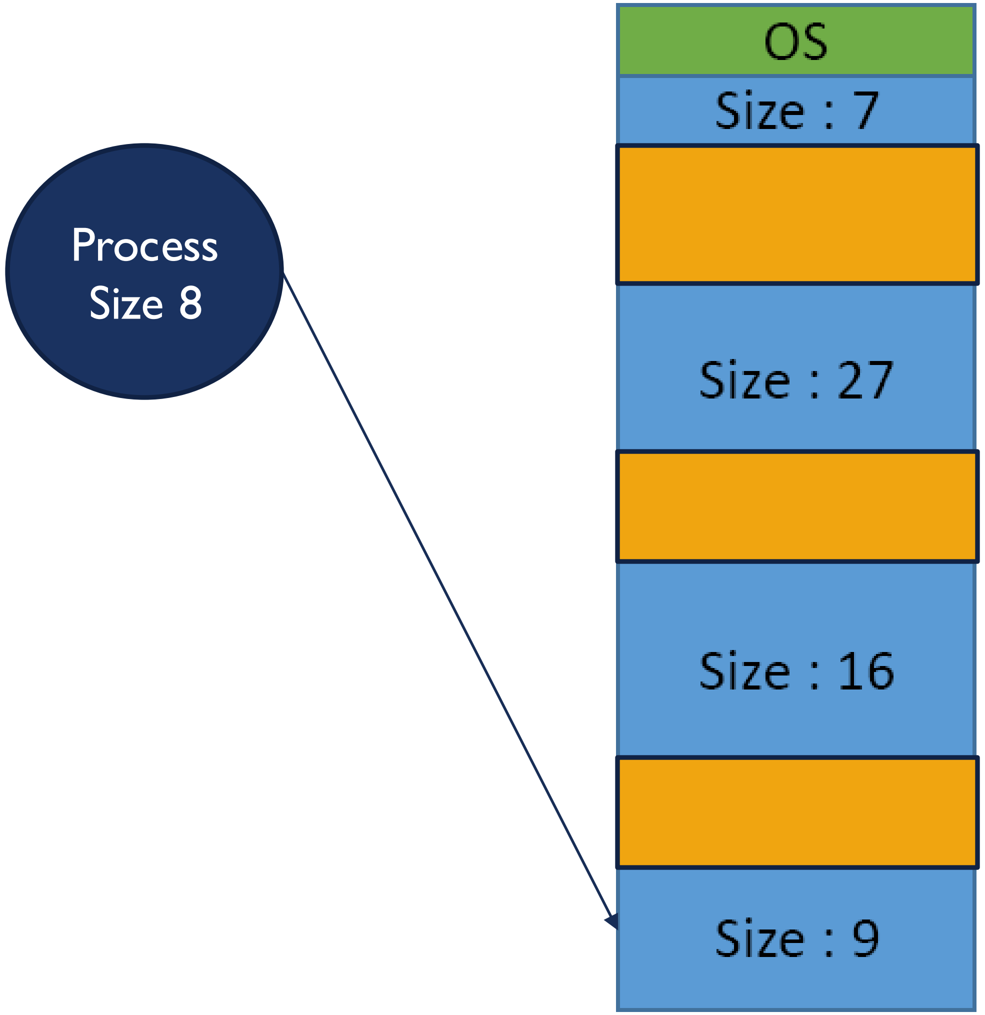


BEST FIT



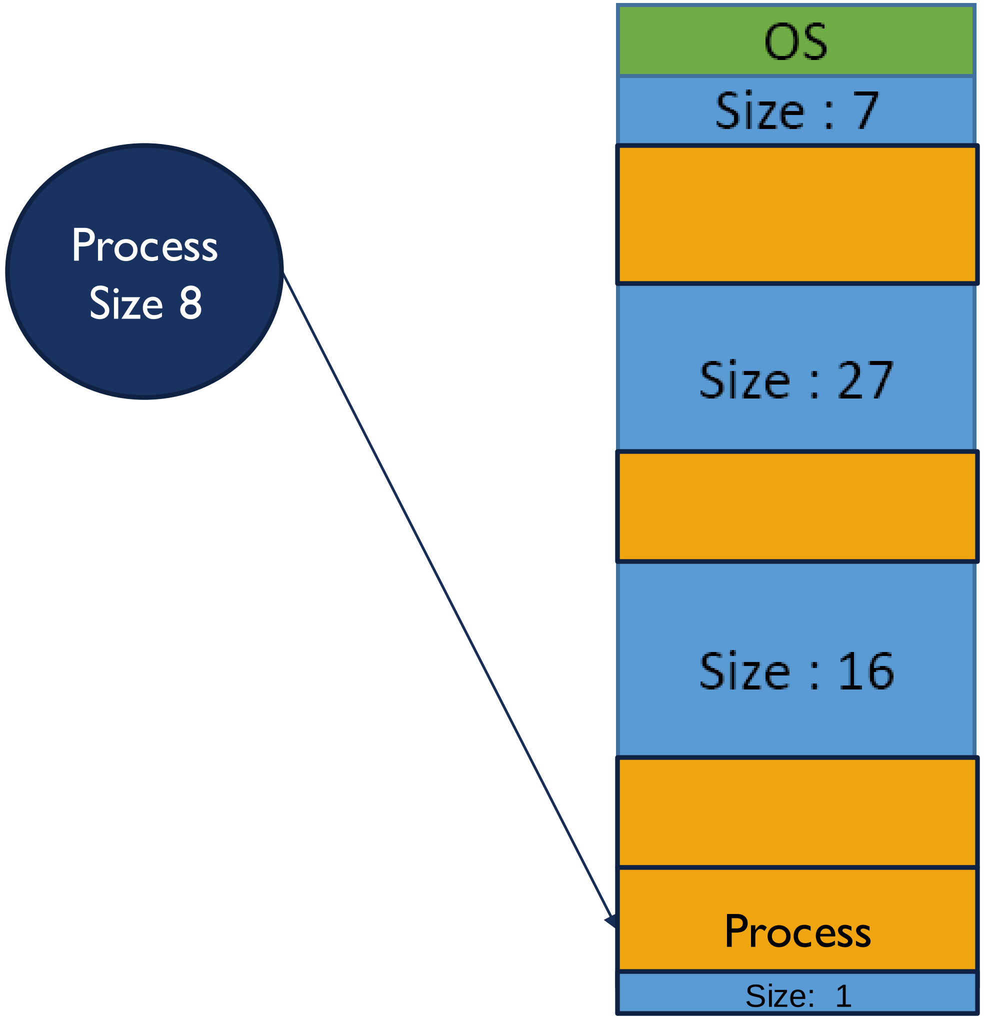
BEST FIT

- Best Fit: Smallest hole left behind

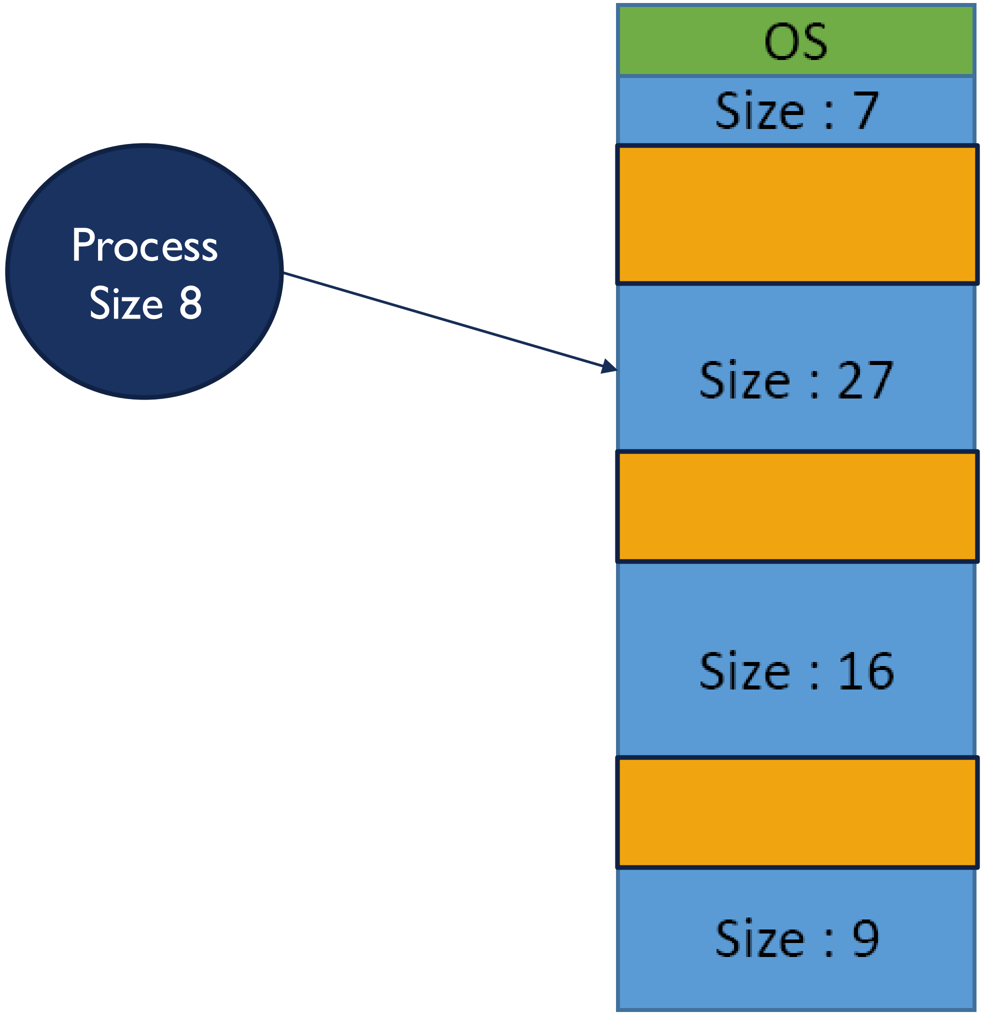


BEST FIT

- Best Fit: Smallest hole left behind
- Smallest hole = less wasted memory space.



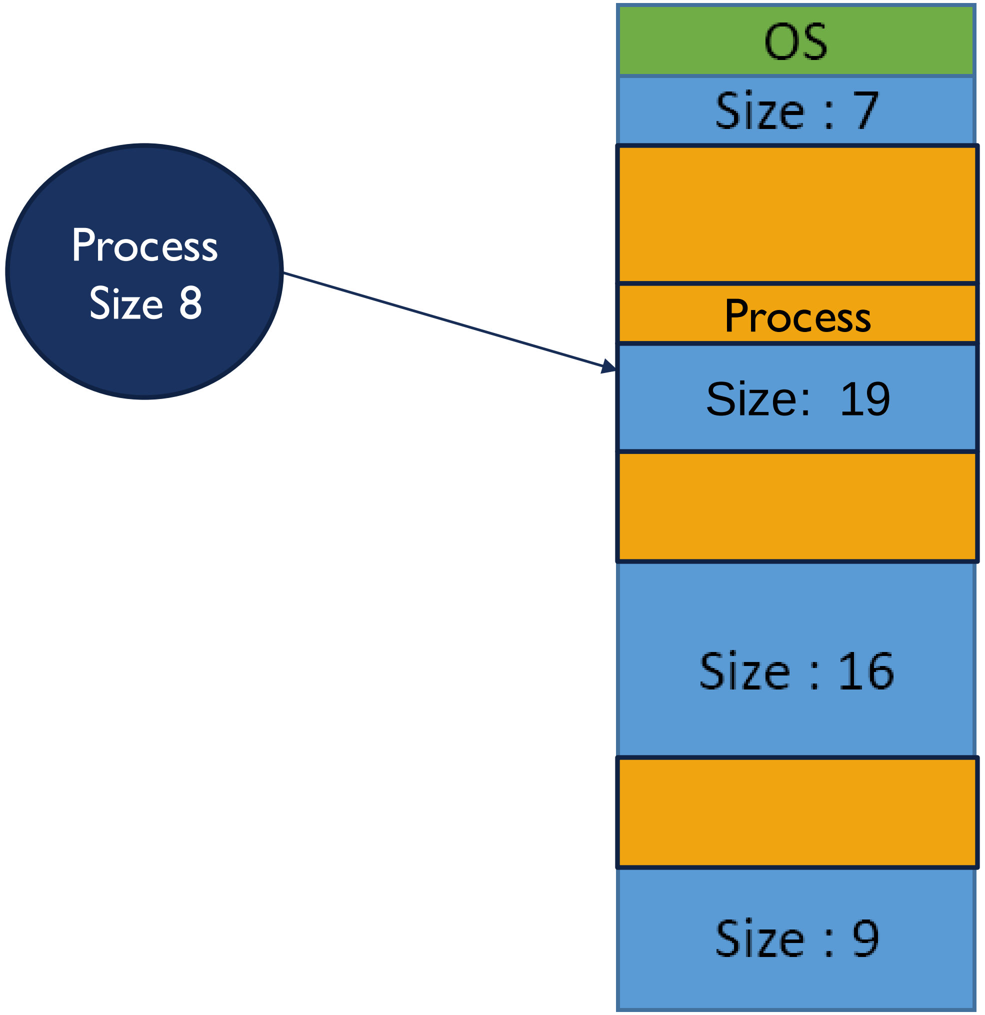
WORST FIT



- Worst Fit: Largest hole left behind.
- Why would that be good?

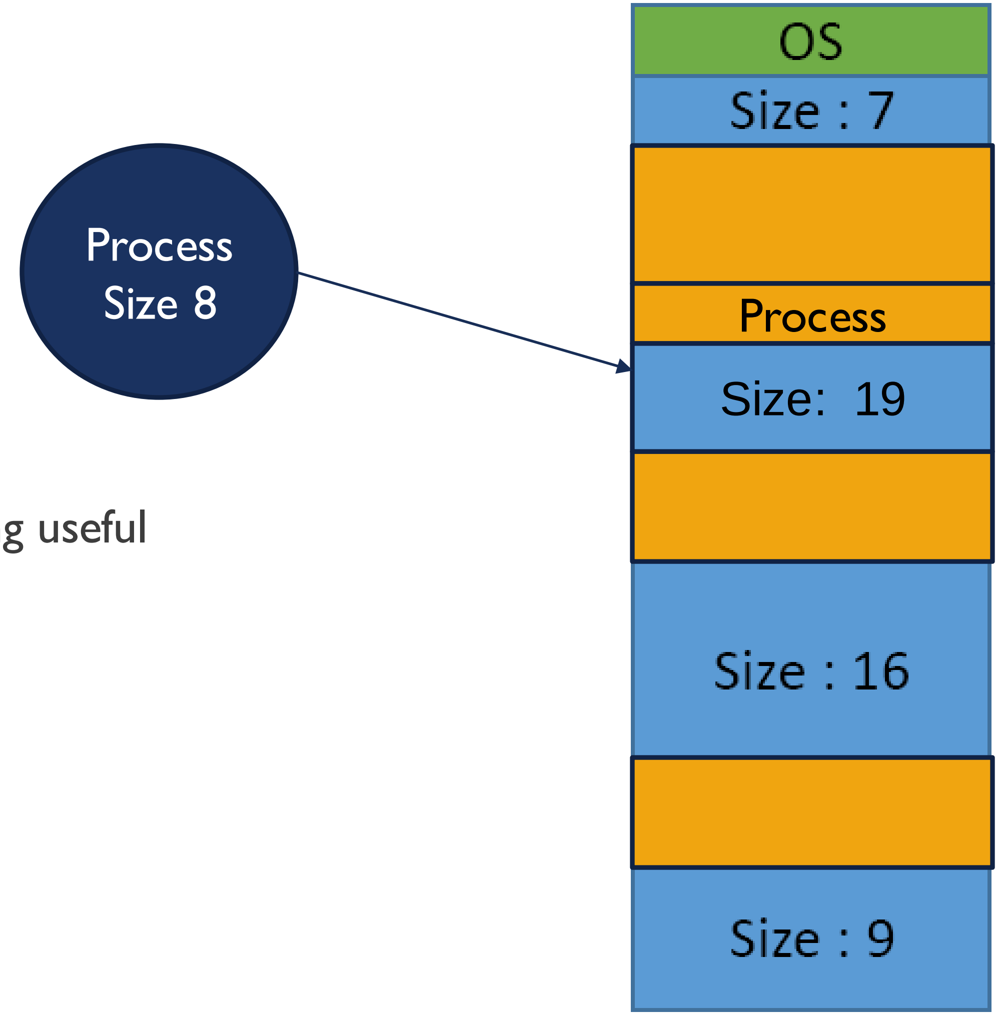
WORST FIT

- Worst Fit: Largest hole left behind.
- Why would that be good?



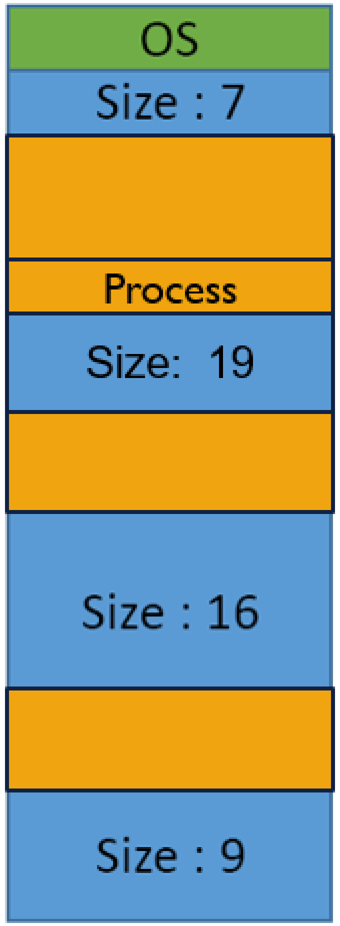
WORST FIT

- Worst Fit: Largest hole left behind.
- Why would that be good?
- Large holes have a higher change of being useful again.



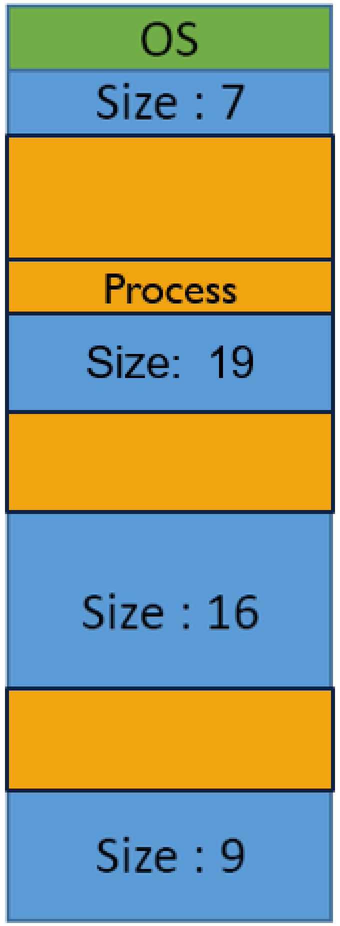
EXTERNAL FRAGMENTATION

- Statics showed that the best algorithms are Best Fit and First Fit.
- No matter what algorithm we use, we goanna end up with holes that are too small to use …



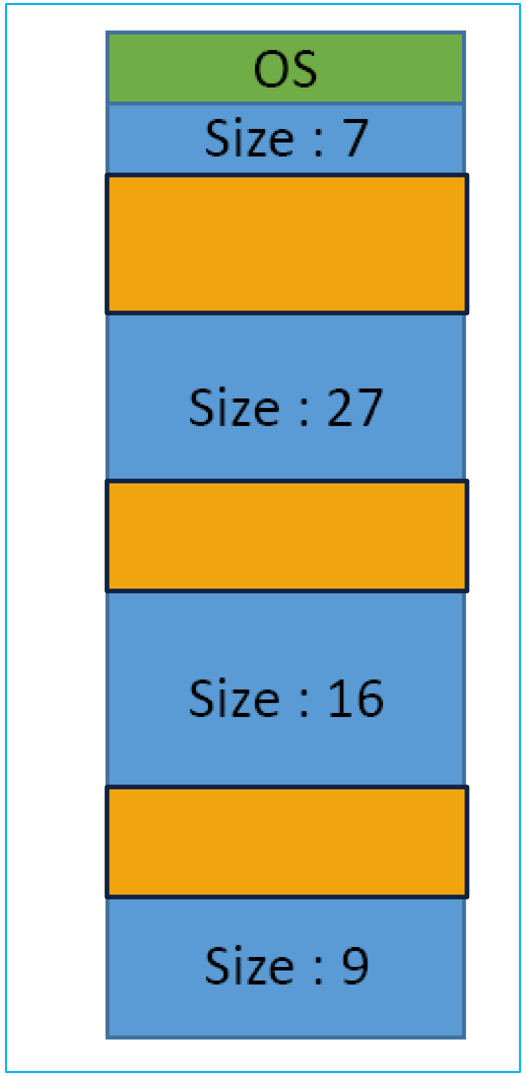
EXTERNAL FRAGMENTATION

- Statics showed that the best algorithms are Best Fit and First Fit.
- No matter what algorithm we use, we goanna end up with holes that are too small to use …
- Roughly 1/3 of the memory is unusable because holes being too small.
- This is what we External Fragmentation



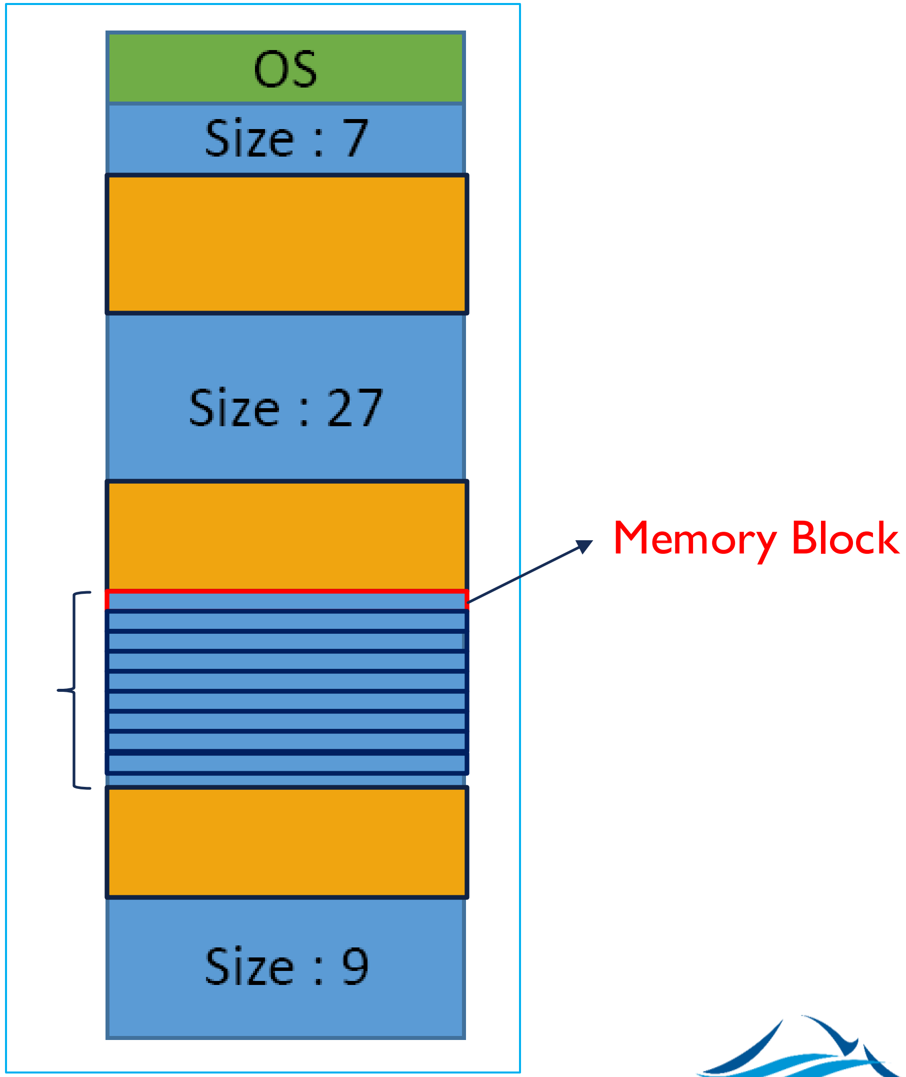
INTERNAL FRAGMENTATION

To simplify hardware, Memory is assigned in blocks.



INTERNAL FRAGMENTATION

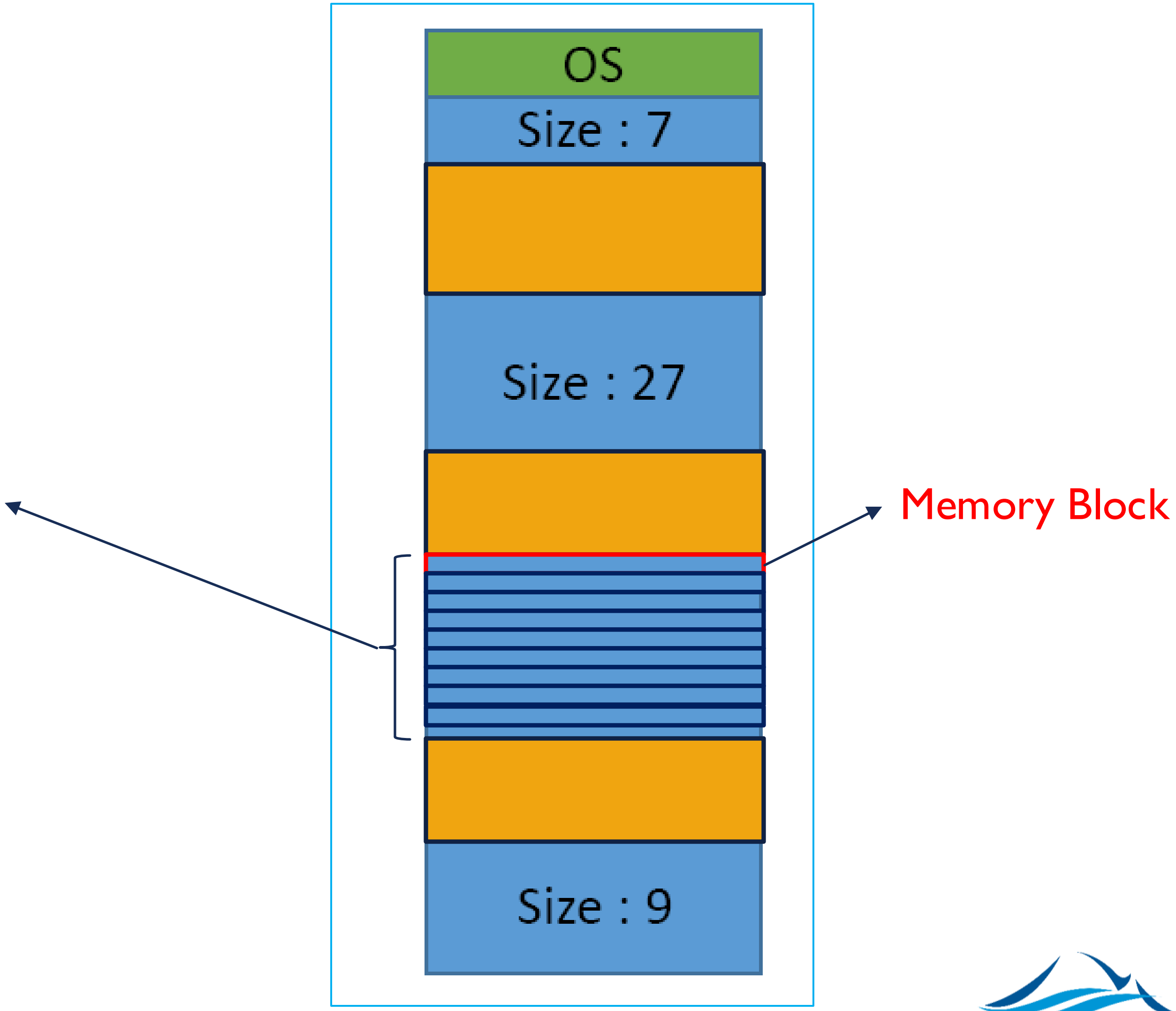
To simplify hardware, Memory is assigned in blocks.



INTERNAL FRAGMENTATION

To simplify hardware, Memory is assigned in blocks.

Assume a block size of 64 KB, If a process needs 96 KB of memory. How many blocks the OS will allocate to it?

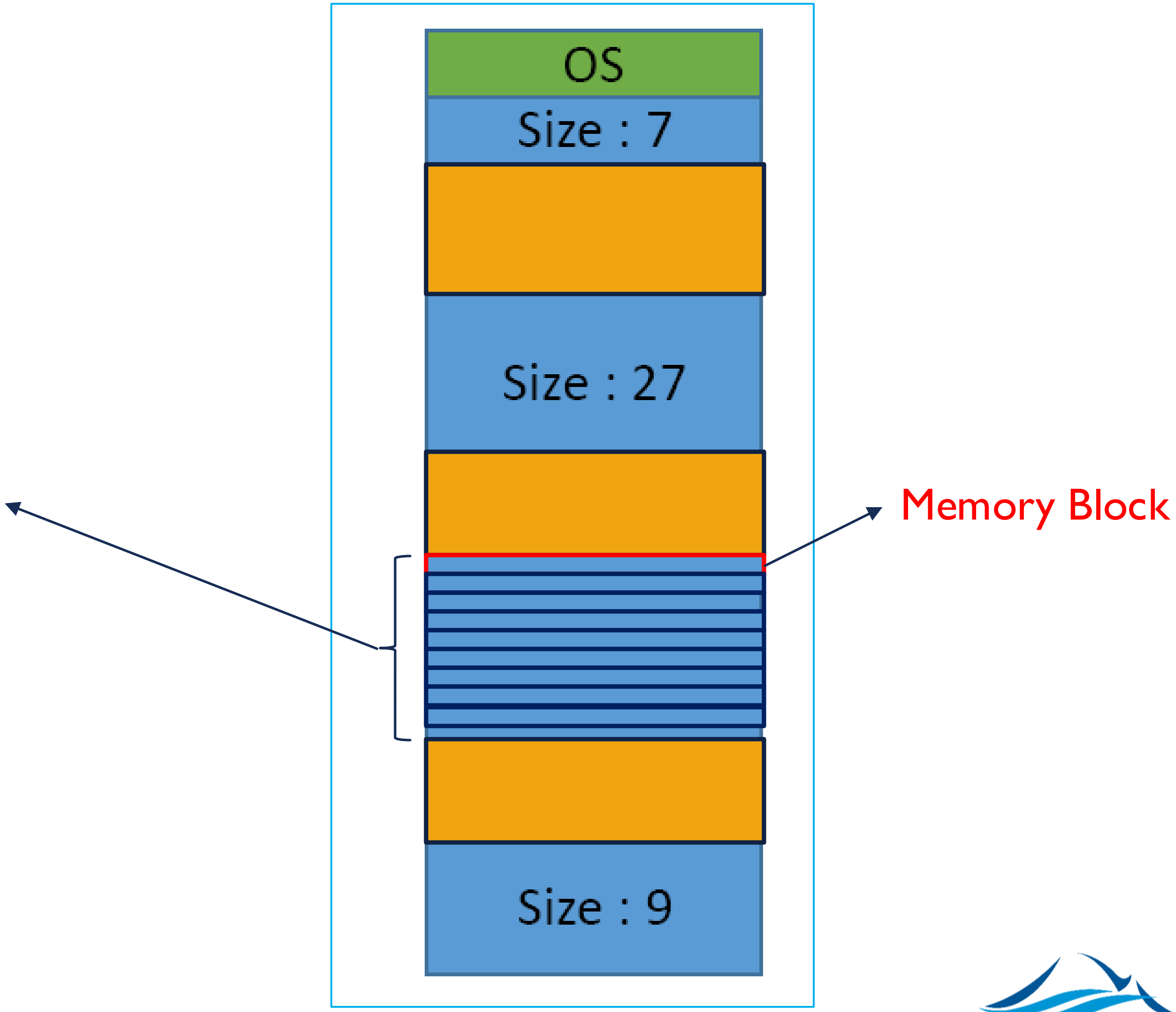


INTERNAL FRAGMENTATION

To simplify hardware, Memory is assigned in blocks.

Assume a block size of 64 KB, If a process needs 96 KB of memory. How many blocks the OS will allocate to it?

Two blocks but half of block 2 is never used and is essentially wasted ...



INTERNAL FRAGMENTATION

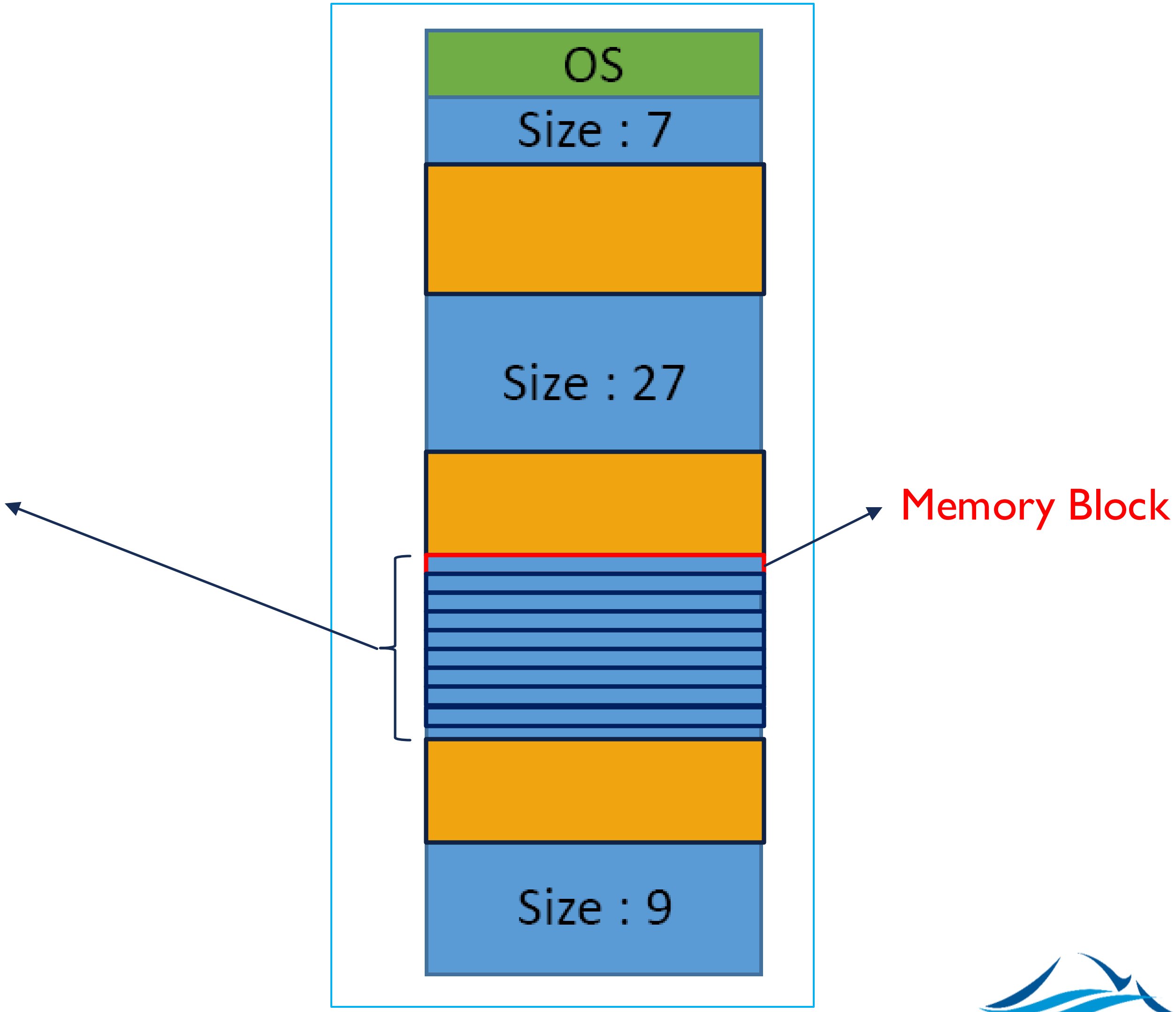
To simplify hardware, Memory is assigned in blocks.

Assume a block size of 64 KB, If a process needs 96 KB of memory. How many blocks the OS will allocate to it?

Two blocks but half of block 2 is never used and is essentially wasted ...

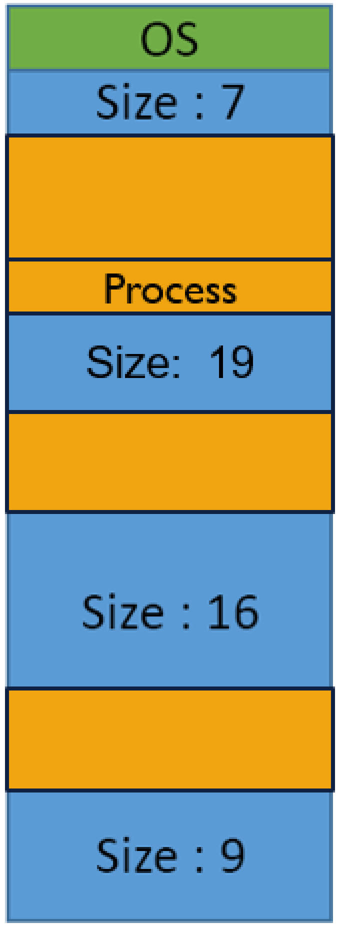
This is known as internal fragmentation.

There is not much to do about internal fragmentation except for using smaller block sizes.



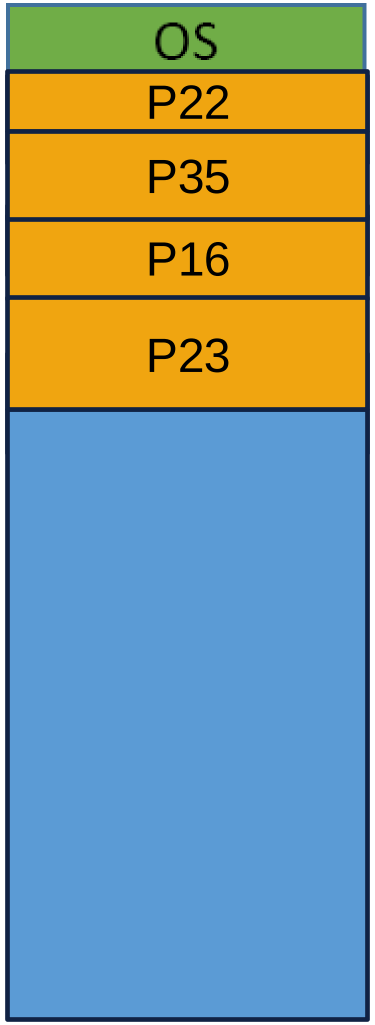
EXTERNAL FRAGMENTATION

- How to solve the problem of holes too small for new processes?



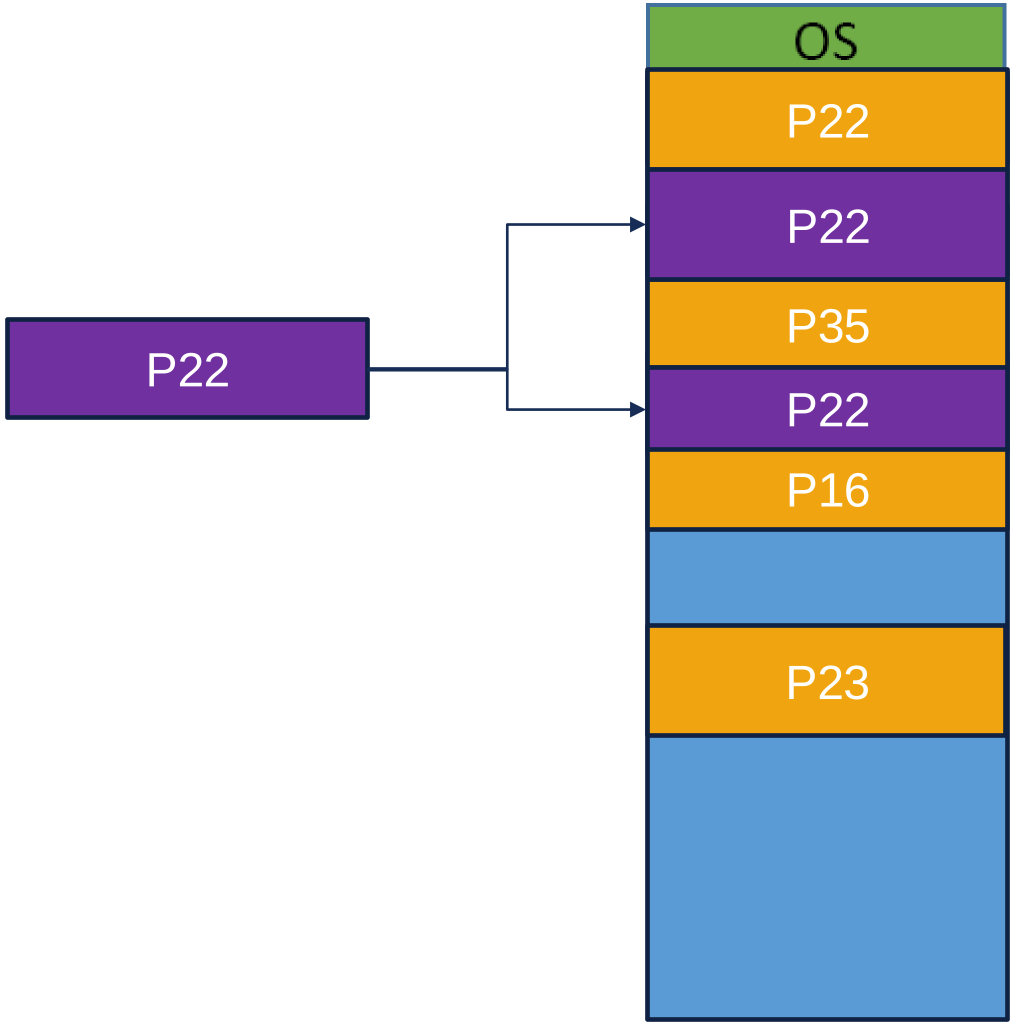
MEMORY COMPACTION

- How to solve the problem of holes too small for new processes?
- Rearranging processes and "compacting" them.

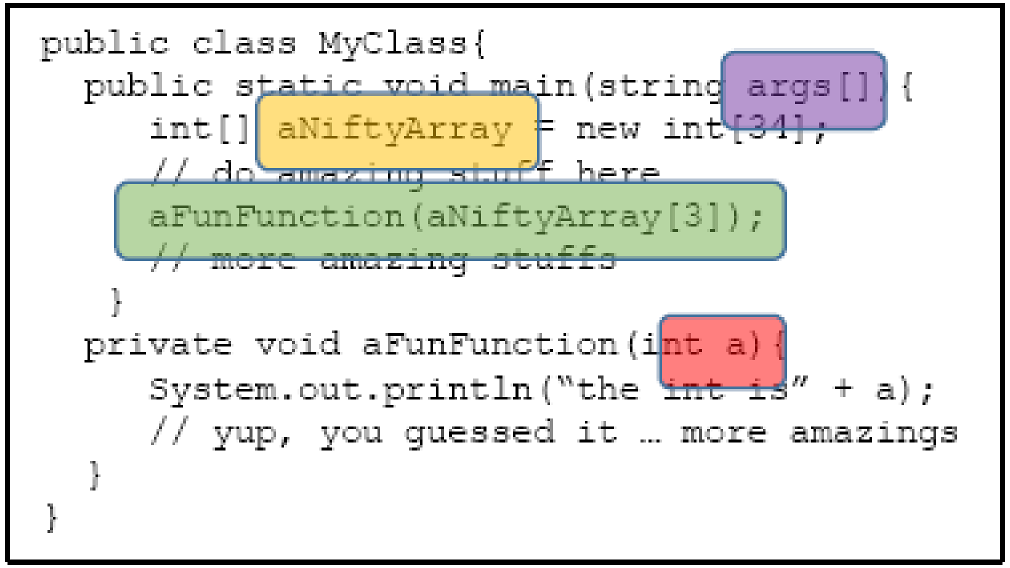


MEMORY SEGMENTATION

- How to solve the problem of holes too small for new processes?
- Rearranging processes and "compacting" them.
- Or using non contiguous memory for the process!

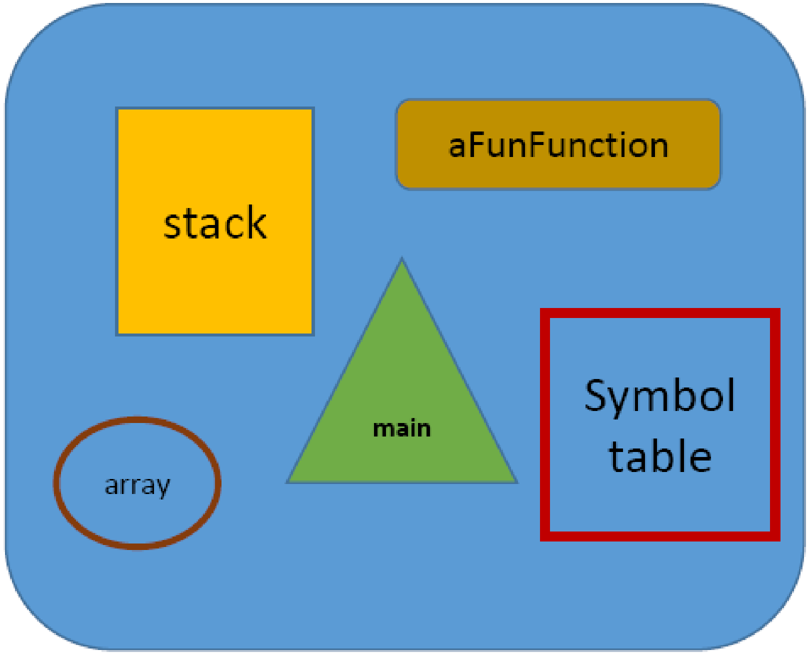


MEMORY SEGMENTATION



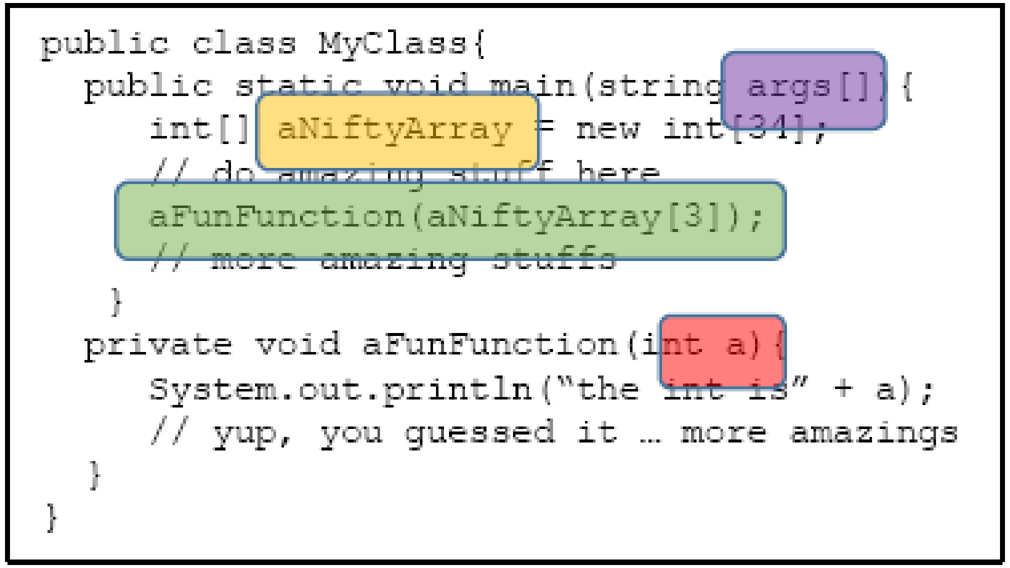
```
public class MyClass{
  public static void main(string args[]){
     int[] aNiftyArray = new int[34];
     // do amazing stuff here
     aFunFunction(aNiftyArray[3]);
     // more amazing stuffs
   }
  private void aFunFunction(int a){
     System.out.println("the int is" + a);
     // yup, you guessed it … more amazings
  }
}
```

As programmers, we don't treat memory as contiguous.

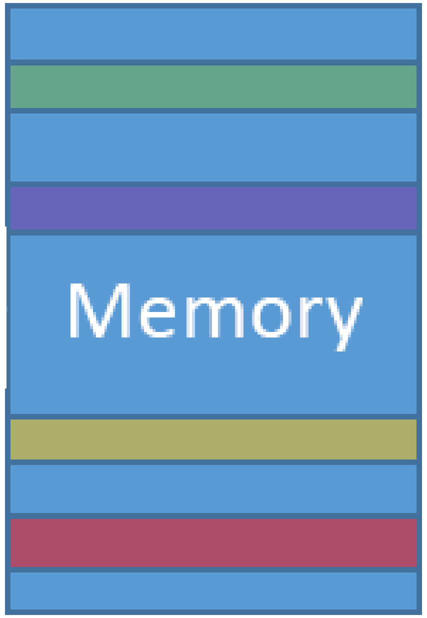


Logical address space (Programmer's perspective)

MEMORY SEGMENTATION



```
public class MyClass{
  public static void main(string args[]){
     int[] aNiftyArray = new int[34];
     // do amazing stuff here
     aFunFunction(aNiftyArray[3]);
     // more amazing stuffs
   }
  private void aFunFunction(int a){
     System.out.println("the int is" + a);
     // yup, you guessed it … more amazings
  }
}
```

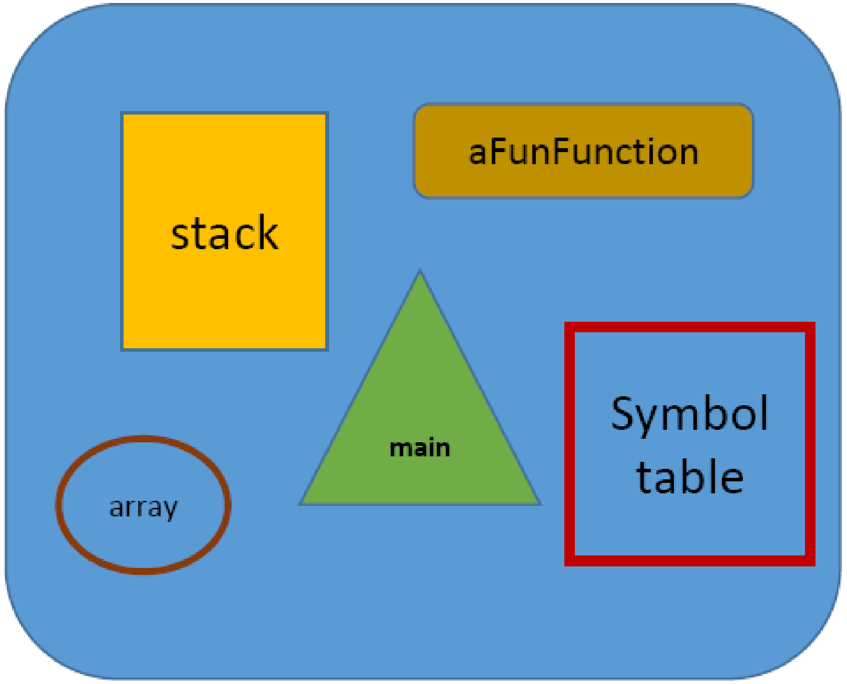


As programmers, we don't treat memory as contiguous.

Memory segmentation: allocate memory in segments representing logical entities.

SEGMENT ADDRESS

Two tuple : <*segment num, offset*>

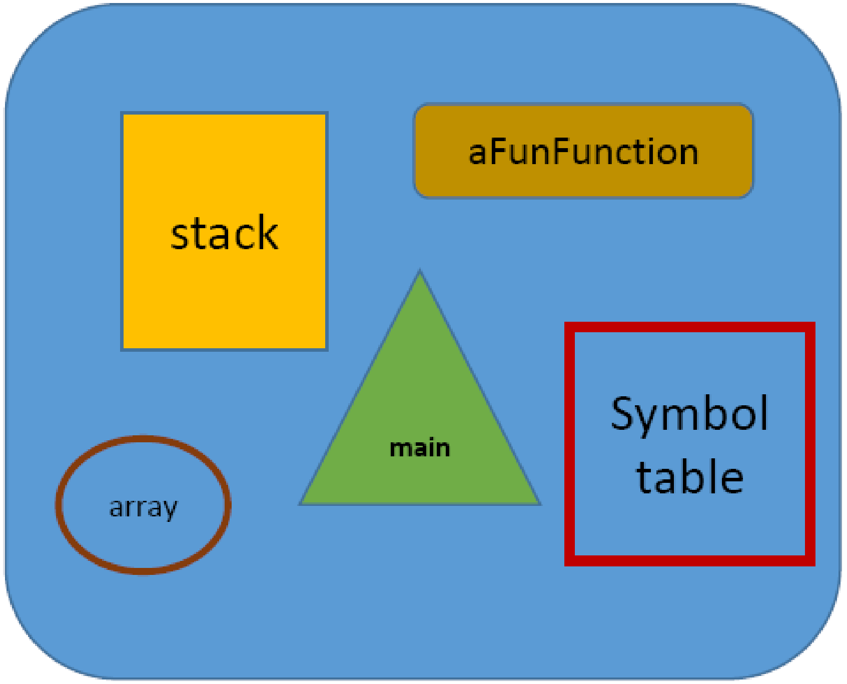


Logical address space
(Programmer's perspective)

SEGMENT ADDRESS

If each piece of code/statement can be uniquely identified via a 2D address (segment num, offset) …

Two tuple : <*segment num, offset*>



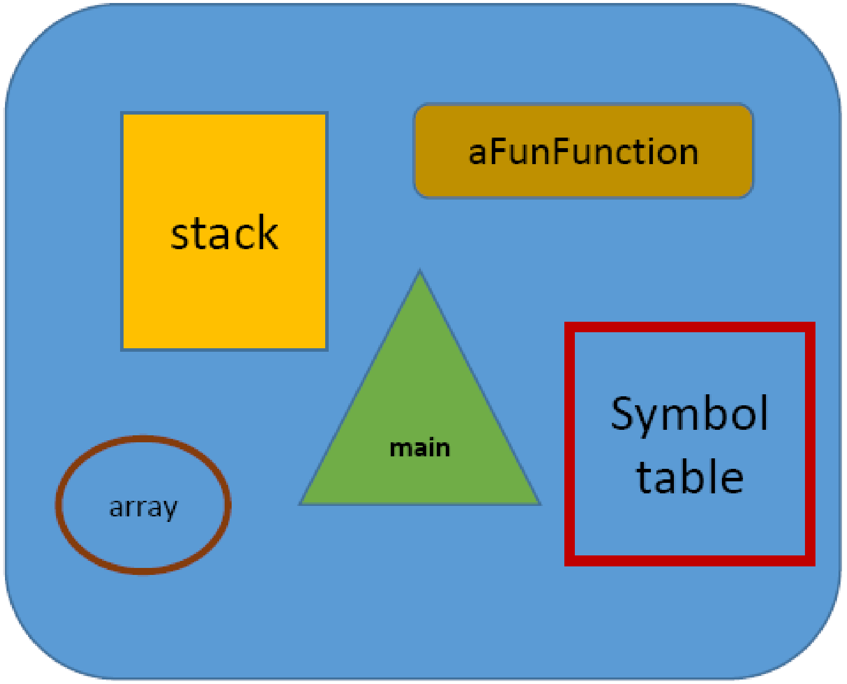
Logical address space
(Programmer's perspective)

SEGMENT ADDRESS

Two tuple : *<segment num, offset>*

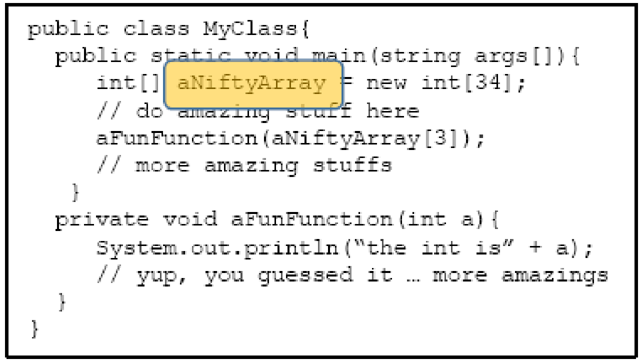
If each piece of code/statement can be uniquely identified via a 2D address (segment num, offset) …

Q: How does the OS map from the 2D address space (programmer's perspective) to the 1D physical address space?

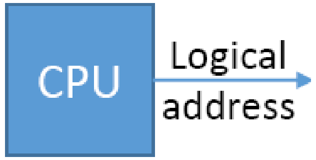


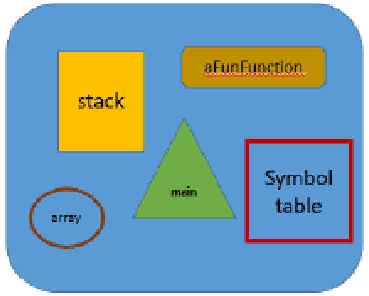
Logical address space (Programmer's perspective)

SEGMENT TABLE



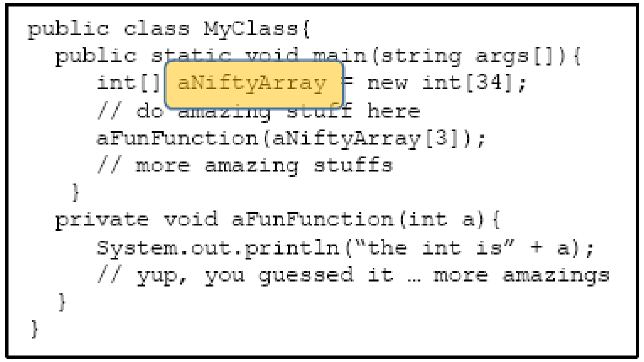
```
public class MyClass{
  public static void main(string args[]){
     int[] aNiftyArray = new int[34];
     // do amazing stuff here
     aFunFunction(aNiftyArray[3]);
     // more amazing stuffs
   }
  private void aFunFunction(int a){
     System.out.println("the int is" + a);
     // yup, you guessed it … more amazings
  }
}
```





The process executing on a CPU generates an address … for example, the process needs the address of the variable `aNiftyArray`

SEGMENT TABLE

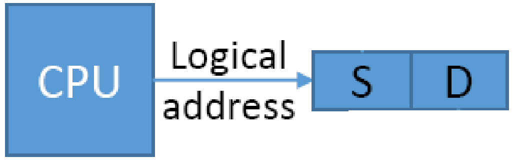


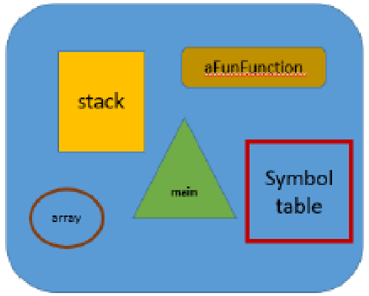
```
public class MyClass{
  public static void main(string args[]){
     int[] aNiftyArray = new int[34];
     // do amazing stuff here
     aFunFunction(aNiftyArray[3]);
     // more amazing stuffs
   }
  private void aFunFunction(int a){
     System.out.println("the int is" + a);
     // yup, you guessed it … more amazings
  }
}
```

The logical address is made up of 2 parts

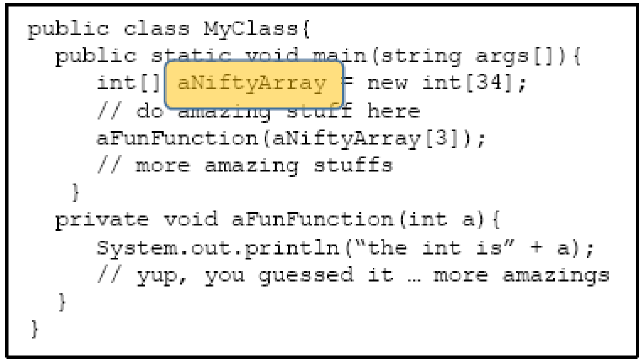
<segment num, offset>

<S, D>

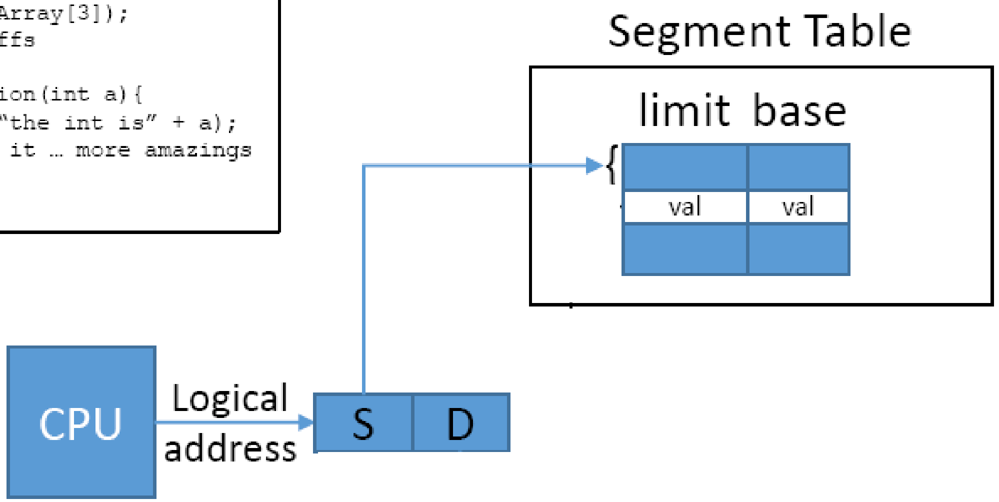


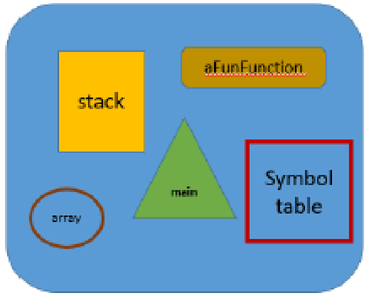


SEGMENT TABLE



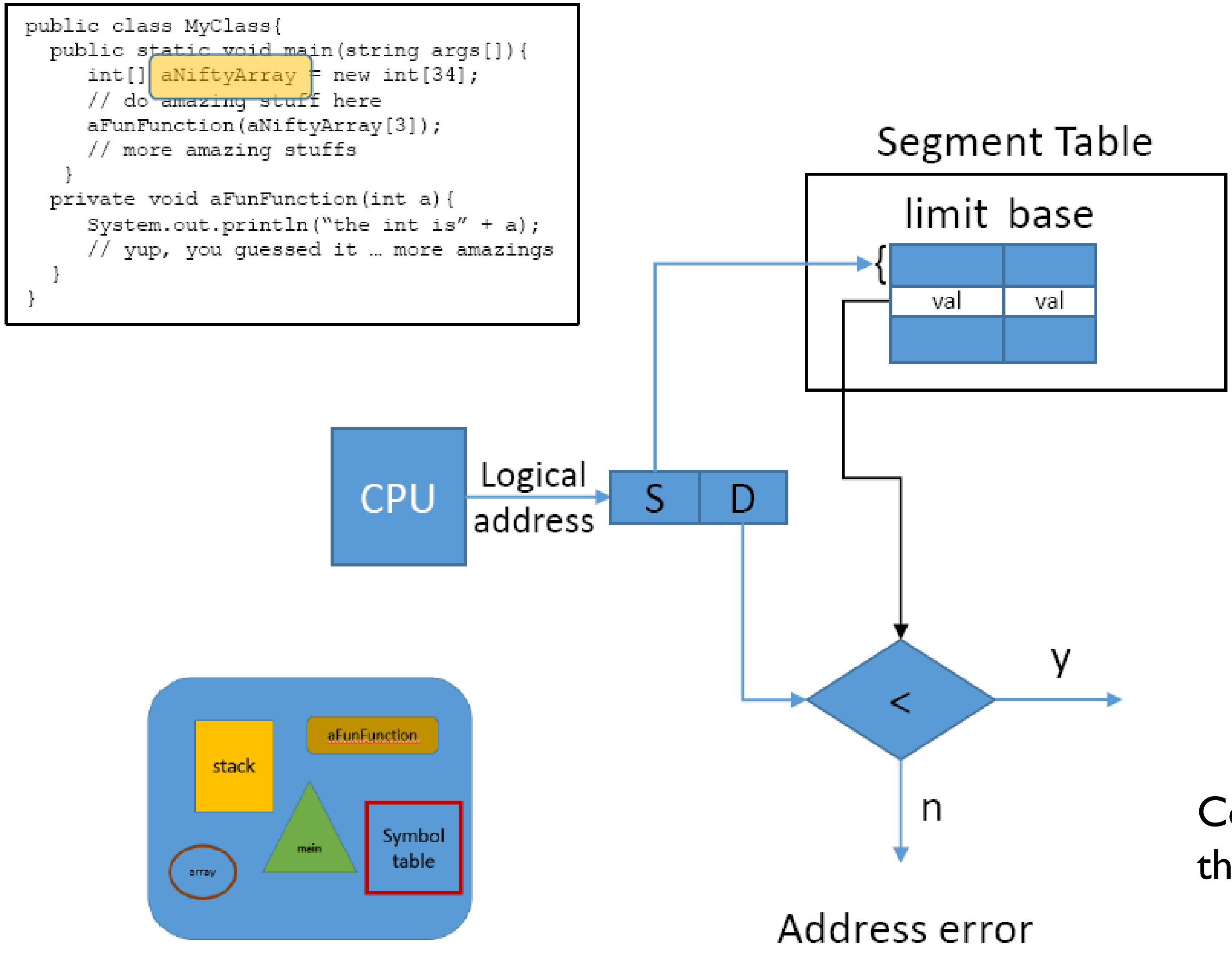
```
public class MyClass{
  public static void main(string args[]){
     int[] aNiftyArray = new int[34];
     // do amazing stuff here
     aFunFunction(aNiftyArray[3]);
     // more amazing stuffs
   }
  private void aFunFunction(int a){
     System.out.println("the int is" + a);
     // yup, you guessed it … more amazings
  }
}
```





- Base: value of physical starting address
- Limit: size of segment

SEGMENT TABLE

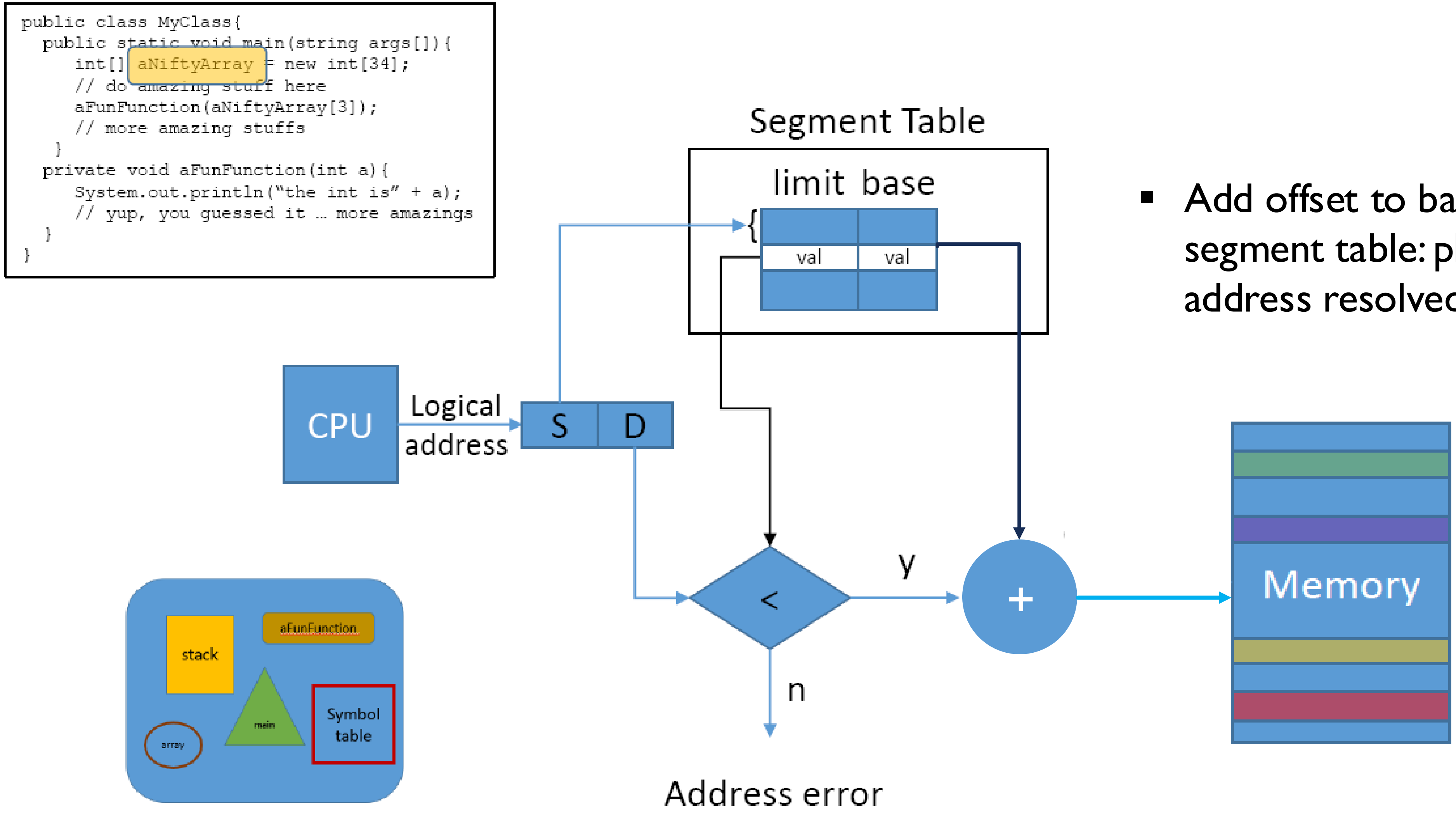


```
public class MyClass{
  public static void main(string args[]){
     int[] aNiftyArray = new int[34];
     // do amazing stuff here
     aFunFunction(aNiftyArray[3]);
     // more amazing stuffs
   }
  private void aFunFunction(int a){
     System.out.println(“the int is” + a);
     // yup, you guessed it … more amazings
   }
}
```

- Base: value of physical starting address
- Limit: size of segment

Comparing ‘D’, the offset, with the limit.

SEGMENT TABLE



```
public class MyClass{
  public static void main(string args[]){
     int[] aNiftyArray = new int[34];
     // do amazing stuff here
     aFunFunction(aNiftyArray[3]);
     // more amazing stuffs
   }
  private void aFunFunction(int a){
     System.out.println(“the int is” + a);
     // yup, you guessed it … more amazings
  }
}
```

- Add offset to base value from segment table: physical address resolved.

SEGMENT TABLE

Segment table

base	limit
4000	2000
760	120
56500	300
43000	700

Q: Is the Segment table valid?

A	B
C	D

A : Yes

B : No

SEGMENT TABLE

Segment table

	base	limit
0	1400	250
1	2600	3500
2	200	300
3	60000	5
4	17000	1800

I. <3,200>
II. <0,50>
III. <3,60000>
IV. <1,200>
V. <4,4>

Q: For the segment table shown right, which of the choices I-V of logical addresses would result in an address error?

A B
C D

A : I only
B : II and III only
C : I and III only
D : IV and V only

SEGMENT TABLE

Segment table

	base	limit
0	1400	250
1	2600	3500
2	200	300
3	60000	5
4	17000	1800

I. <3,200>
II. <0,50>
III. <3,60000>
IV. <1,200>
V. <4,4>

Q: For the segment table shown right, which of the choices I-V of logical addresses would result in an address error?

A	B
C	D

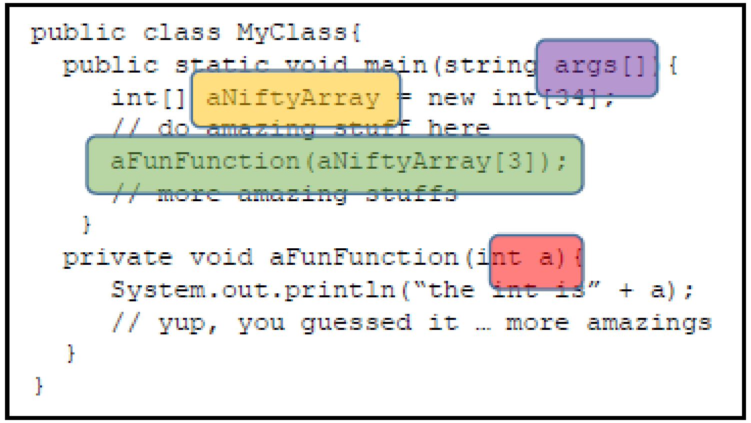
A : I only

B : II and III only

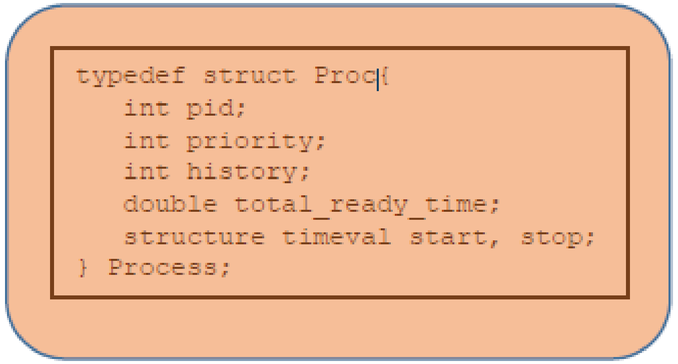
[✓ C : I and III only]

D : IV and V only

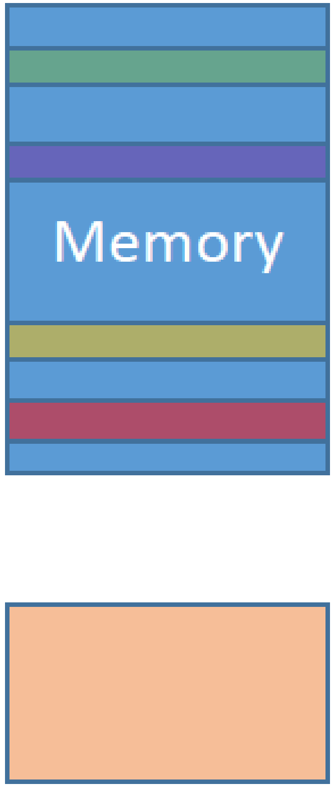
FRAGMENTATION



```
public class MyClass{
  public static void main(string args[]){
     int[] aNiftyArray = new int[34];
     // do amazing stuff here
     aFunFunction(aNiftyArray[3]);
     // more amazing stuffs
   }
  private void aFunFunction(int a){
     System.out.println(“the int is” + a);
     // yup, you guessed it … more amazings
  }
}
```



```
typedef struct Proc{
   int pid;
   int priority;
   int history;
   double total_ready_time;
   structure timeval start, stop;
} Process;
```

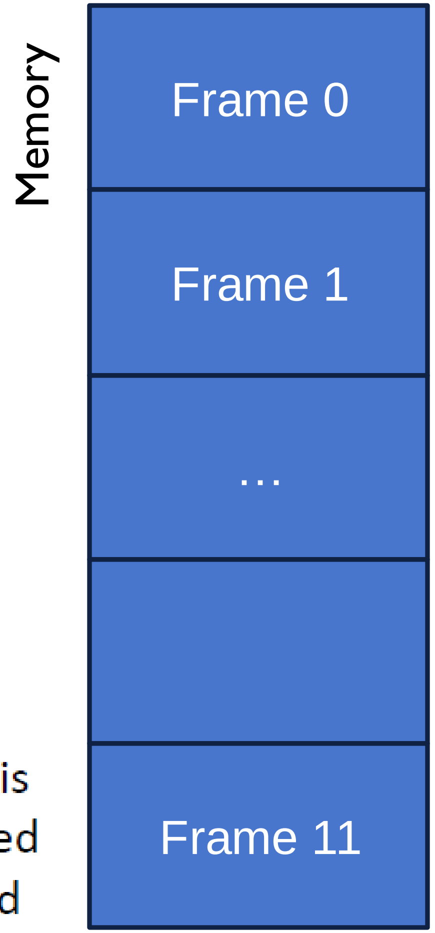


Q: Where might the HUGE orange data structure be placed into memory?

Segmentation can’t completely eliminate fragmentation.

Solution: Paging

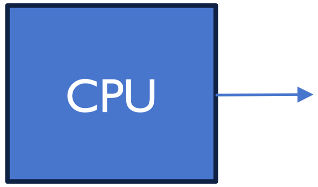
PAGING

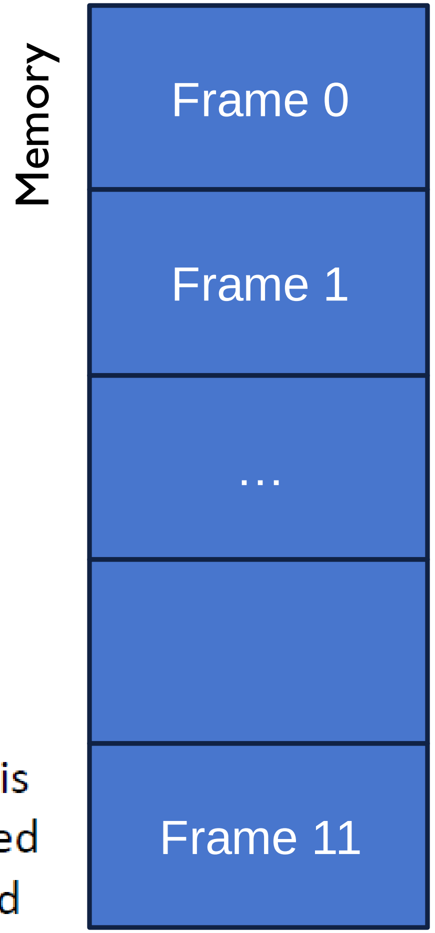


Physical memory is broken up into fixed sized blocks called **frames**

PAGING

The logical memory space is broken up into blocks of the same size, named **pages**

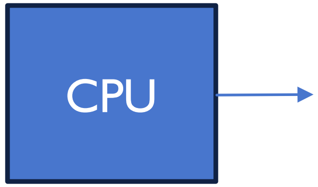




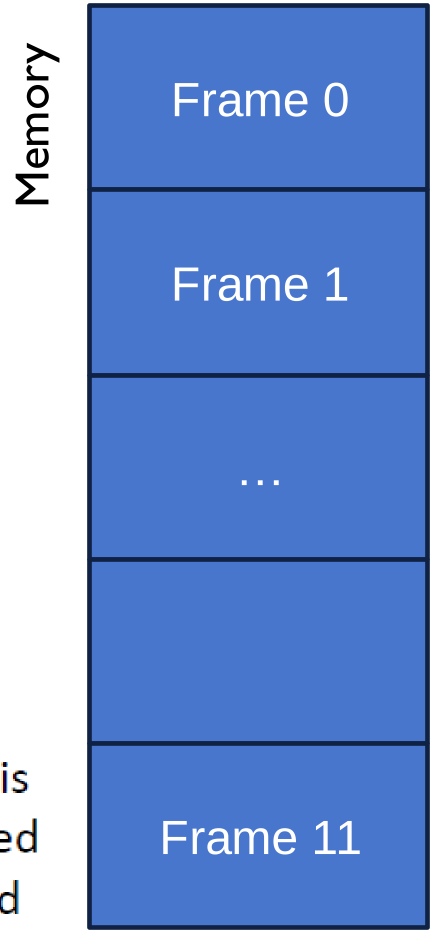
Physical memory is broken up into fixed sized blocks called **frames**

PAGING

The logical memory space is broken up into blocks of the same size, named **pages**



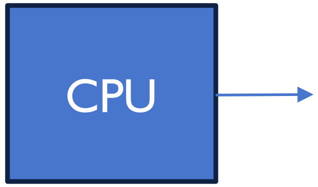
- The logical address space is totally separate from the physical address space … the “data” page can residue in any frame.



Physical memory is broken up into fixed sized blocks called **frames**

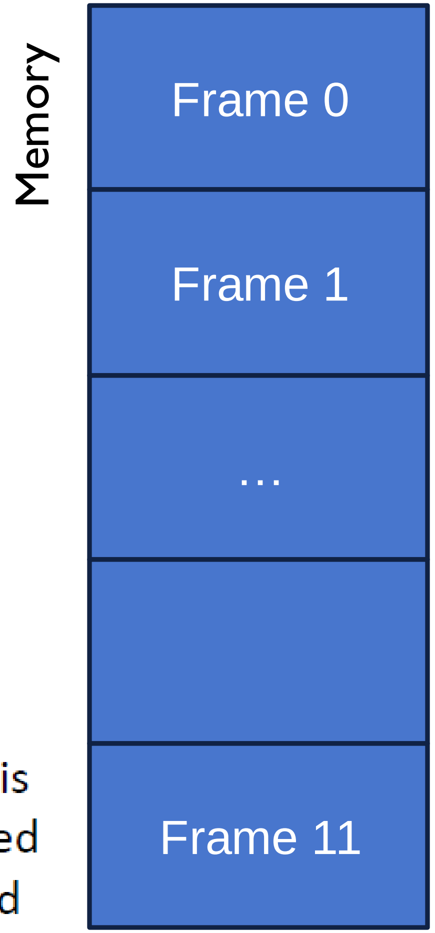
PAGING

The logical memory space is broken up into blocks of the same size, named **pages**

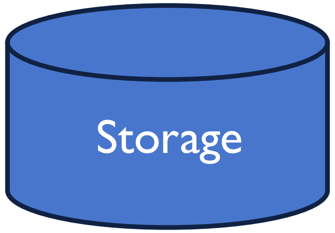


- The logical address space is totally separate from the physical address space … the "data" page can residue in any frame.
- Not all pages need to reside in memory, some can reside on disk/secondary storage!

Physical memory is broken up into fixed sized blocks called **frames**

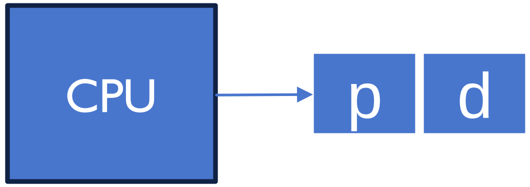


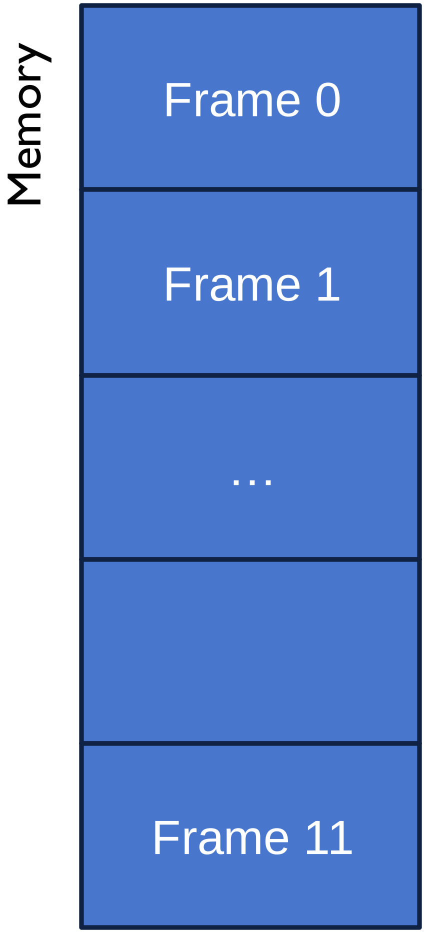
A very large capacity backing store (aka disk) is available for storing data.

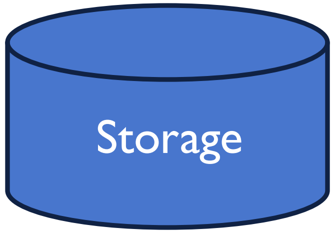


Huge Capacity

PAGING





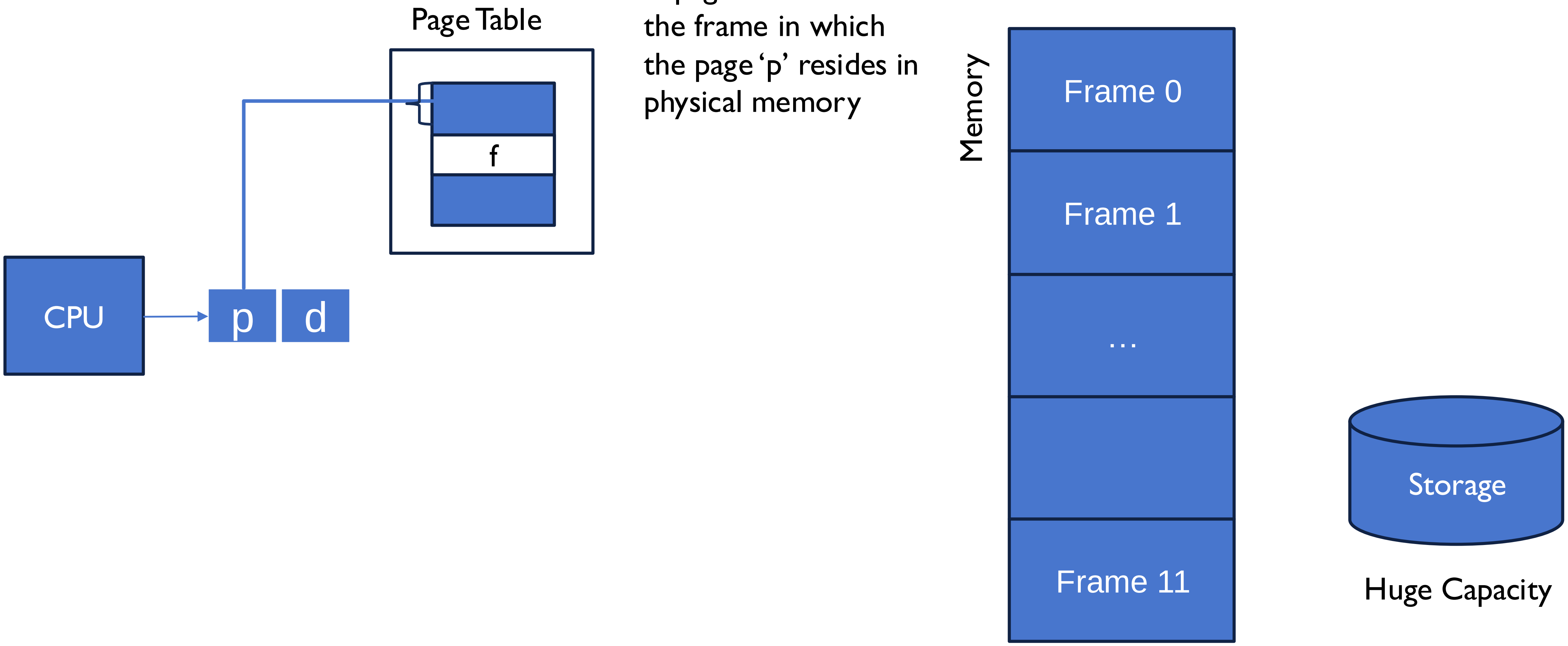


Each address generated by the CPU has a page number (p) and a page offset (d) value

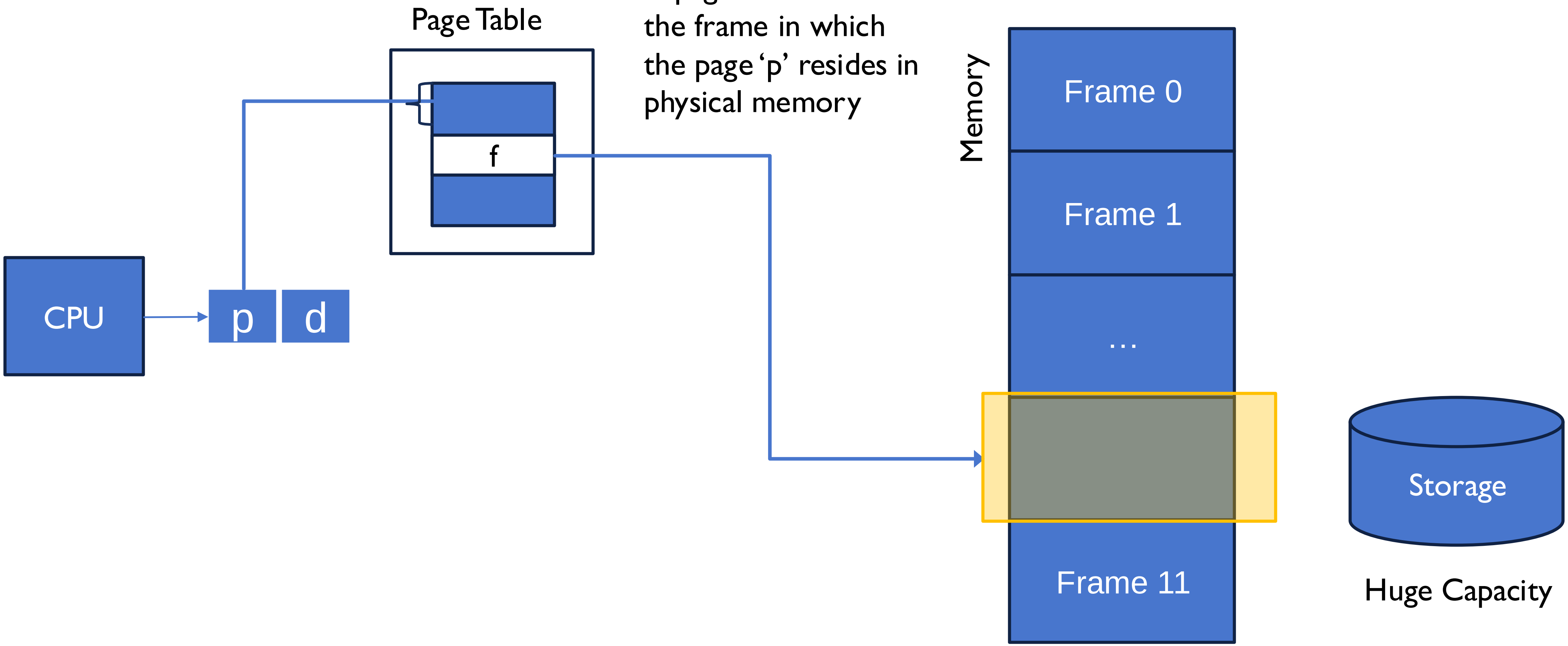
Huge Capacity

PAGING

A page table indicates the frame in which the page 'p' resides in physical memory

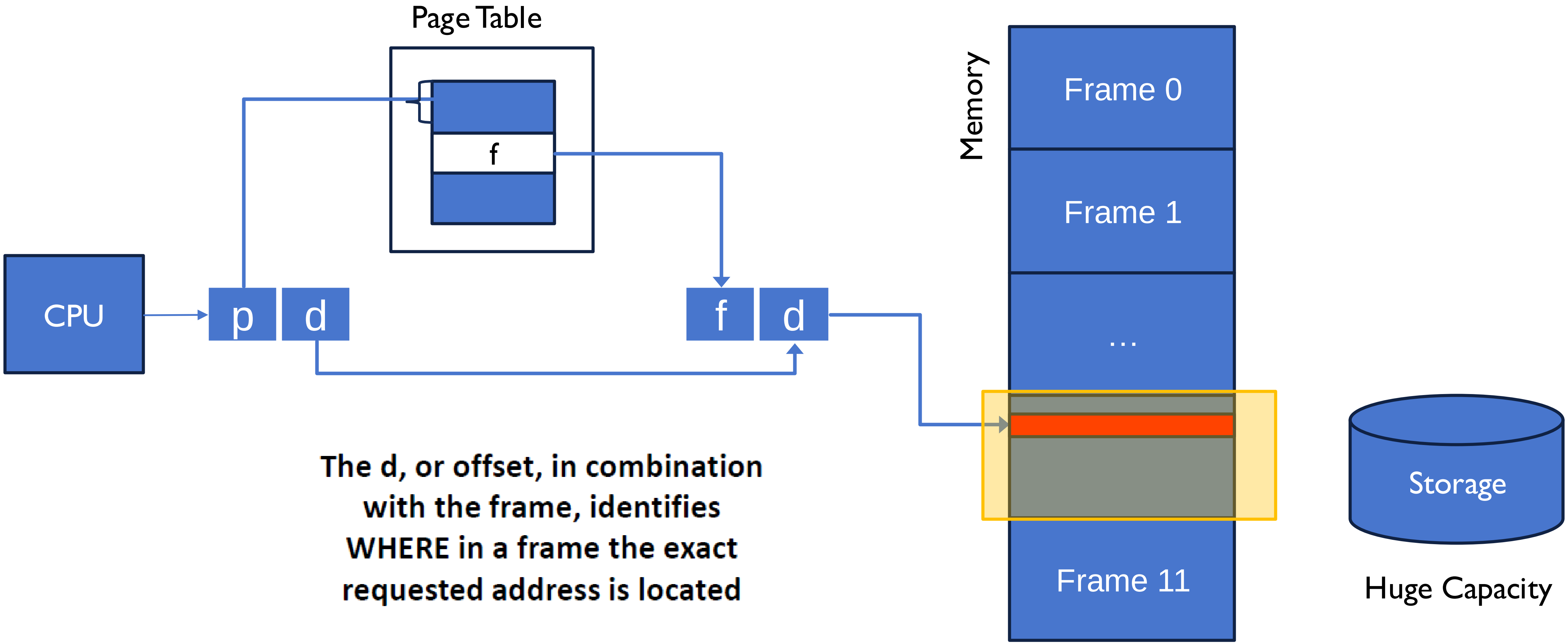


PAGING



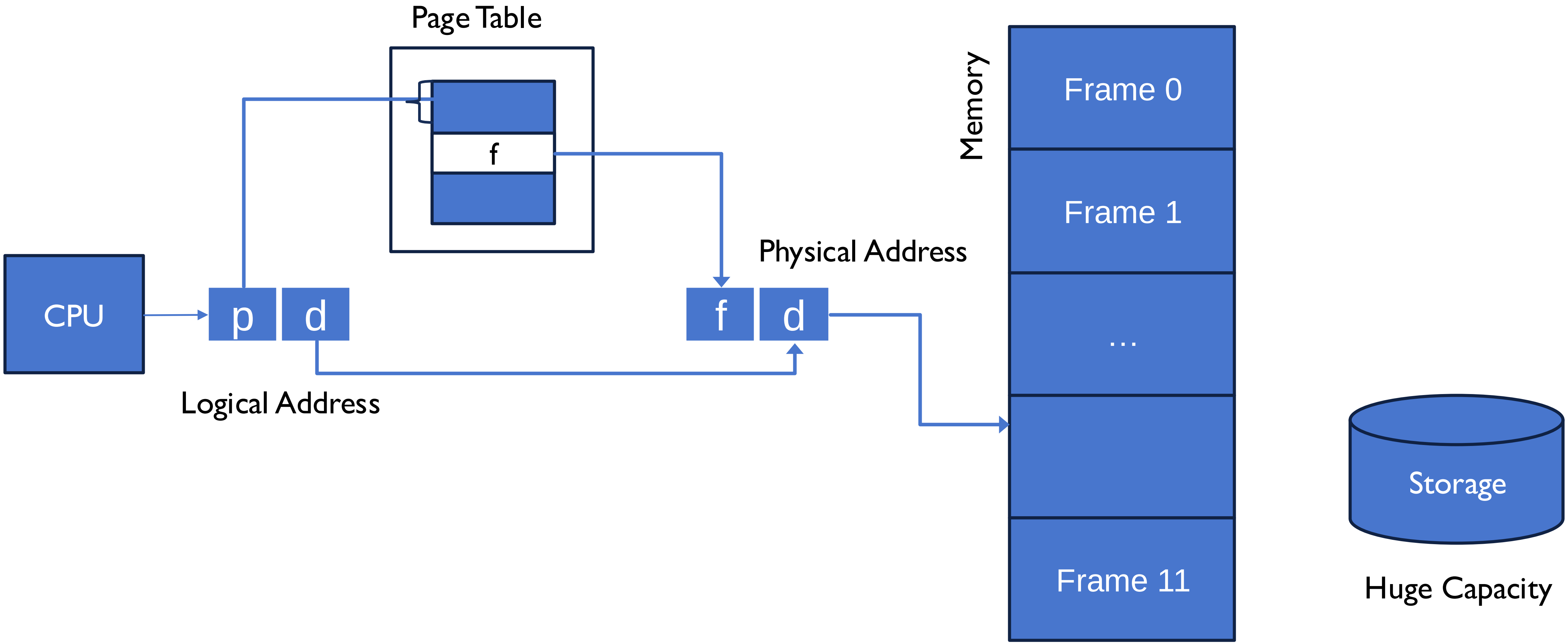
A page table indicates the frame in which the page ‘p’ resides in physical memory

PAGING

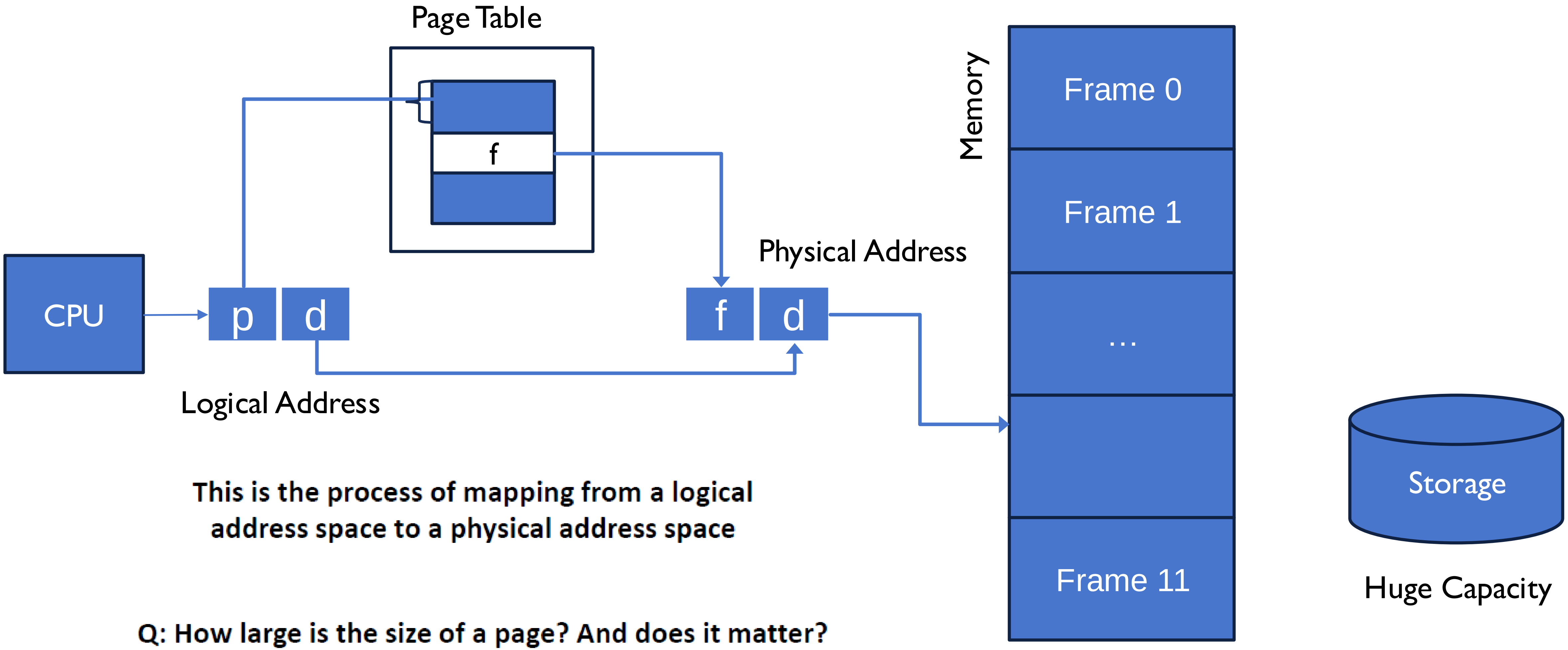


The d, or offset, in combination with the frame, identifies WHERE in a frame the exact requested address is located

PAGING


Page Table
f
CPU
p
d
Logical Address
f
d
Physical Address
Memory
Frame 0
Frame 1
…
Frame 11
Storage
Huge Capacity

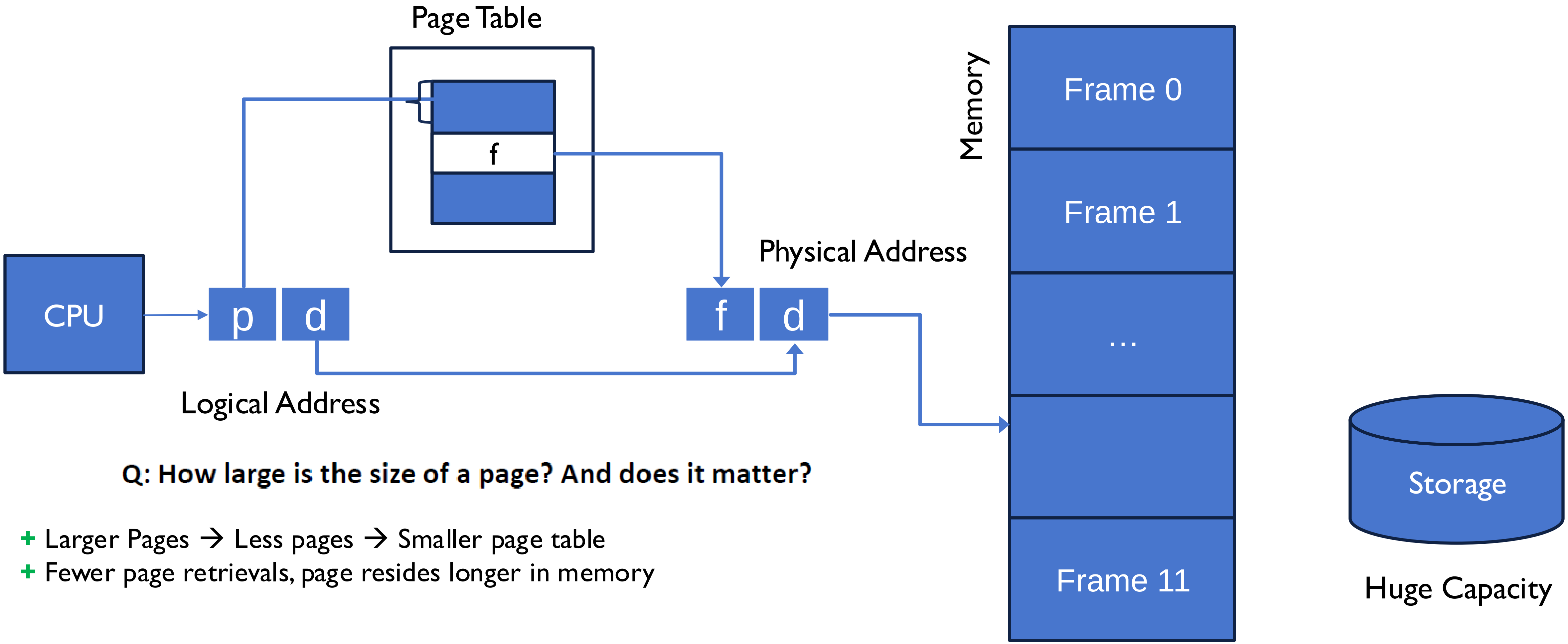
PAGING



This is the process of mapping from a logical address space to a physical address space

Q: How large is the size of a page? And does it matter?

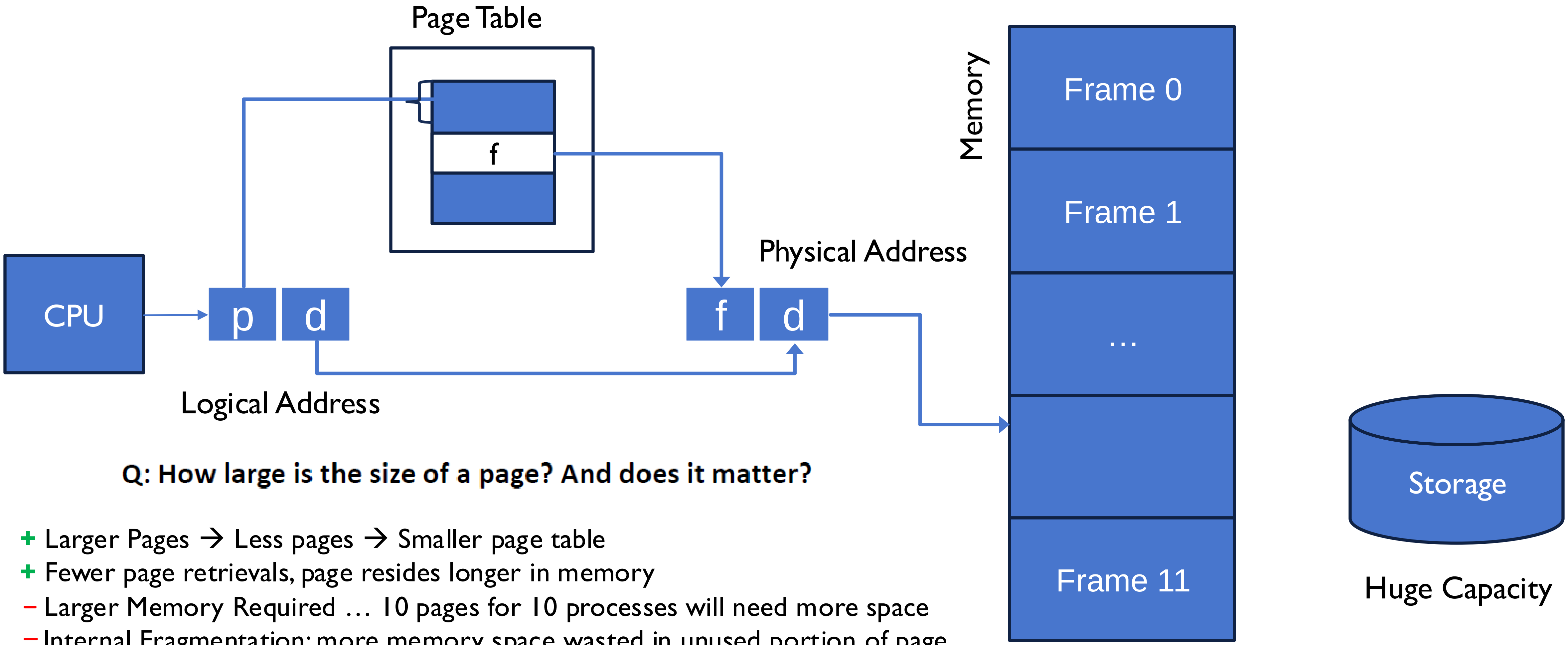
PAGING



Q: How large is the size of a page? And does it matter?

+ Larger Pages → Less pages → Smaller page table
+ Fewer page retrievals, page resides longer in memory

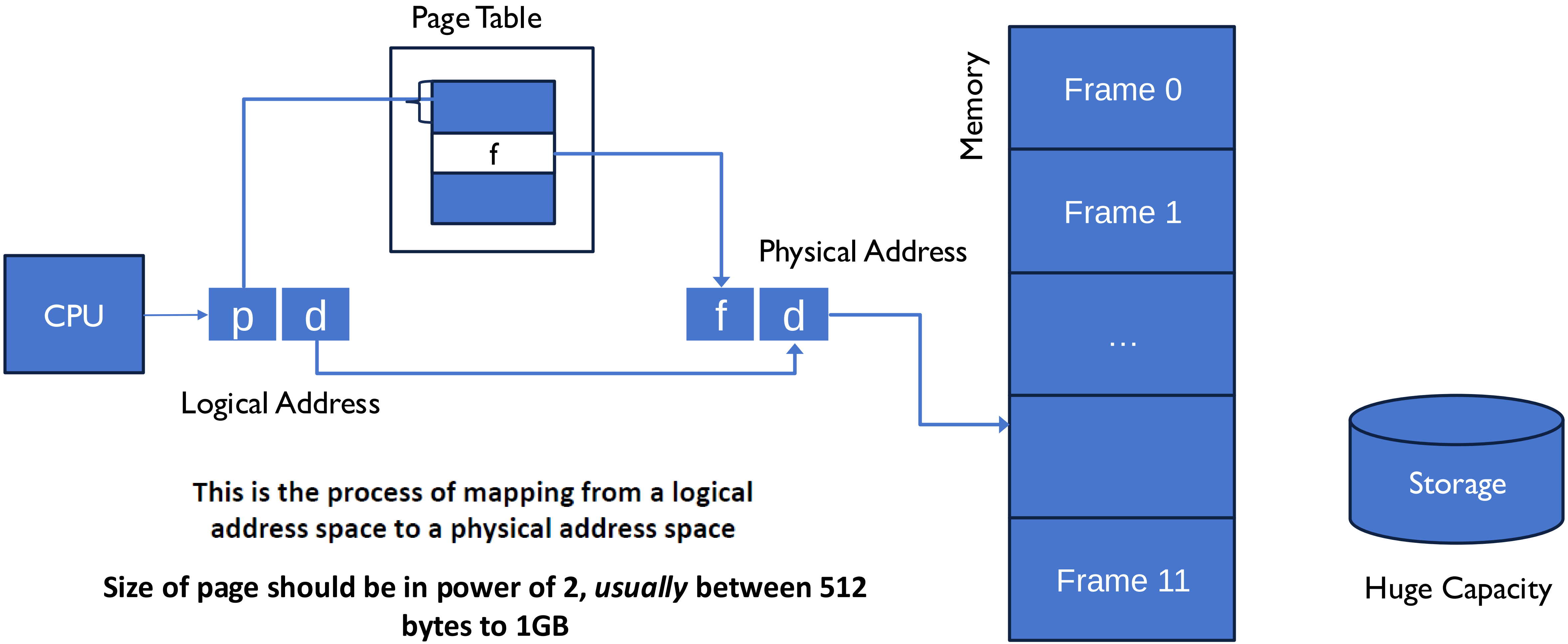
PAGING



Q: How large is the size of a page? And does it matter?

+ Larger Pages → Less pages → Smaller page table
+ Fewer page retrievals, page resides longer in memory
− Larger Memory Required … 10 pages for 10 processes will need more space
− Internal Fragmentation: more memory space wasted in unused portion of page

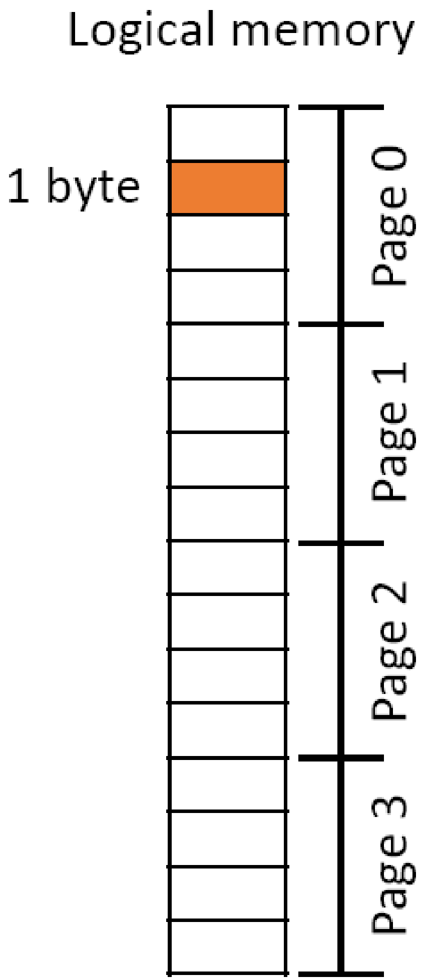
PAGING



This is the process of mapping from a logical address space to a physical address space

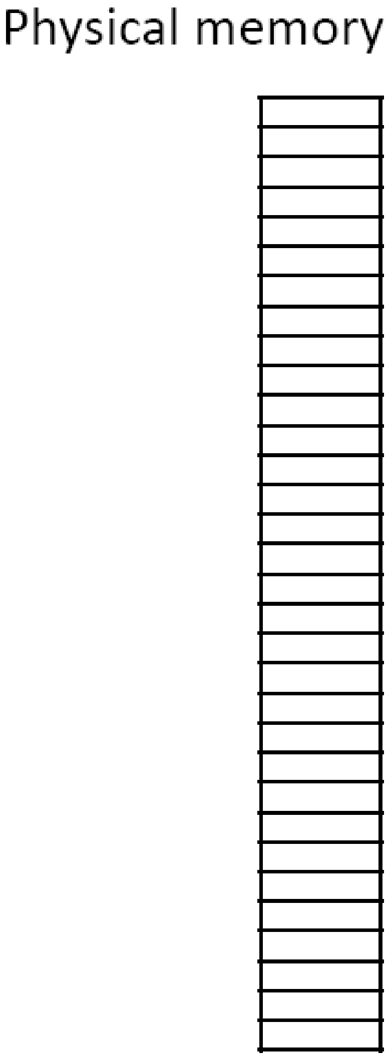
Size of page should be in power of 2, *usually* between 512 bytes to 1GB

PAGING

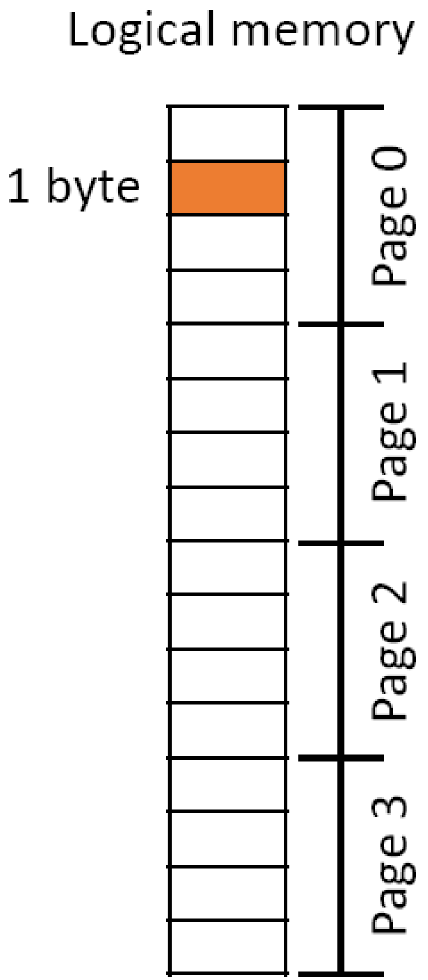


Size of logical address space : 16

Page size = 4 bytes



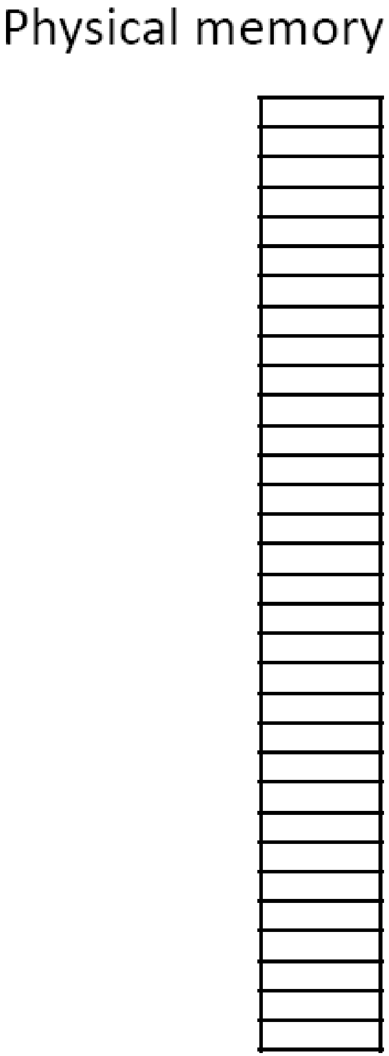
PAGING



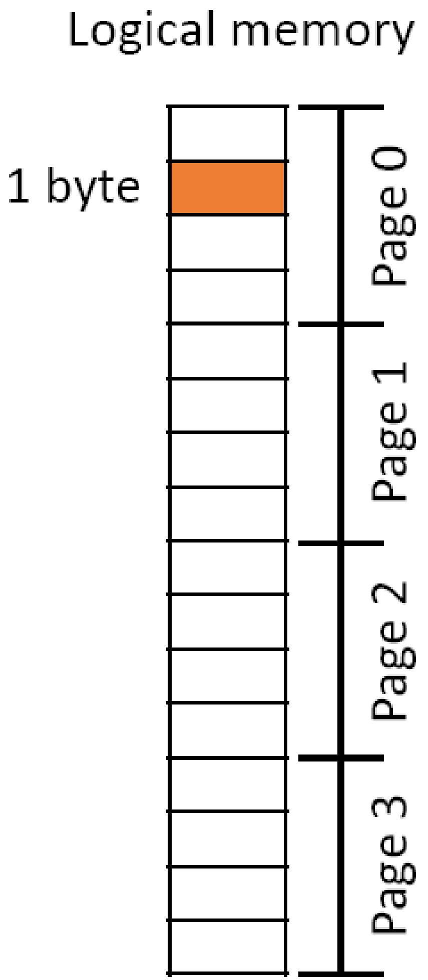
Size of logical address space : 16

Page size = 4 bytes

Q: How many bytes should each frame be?

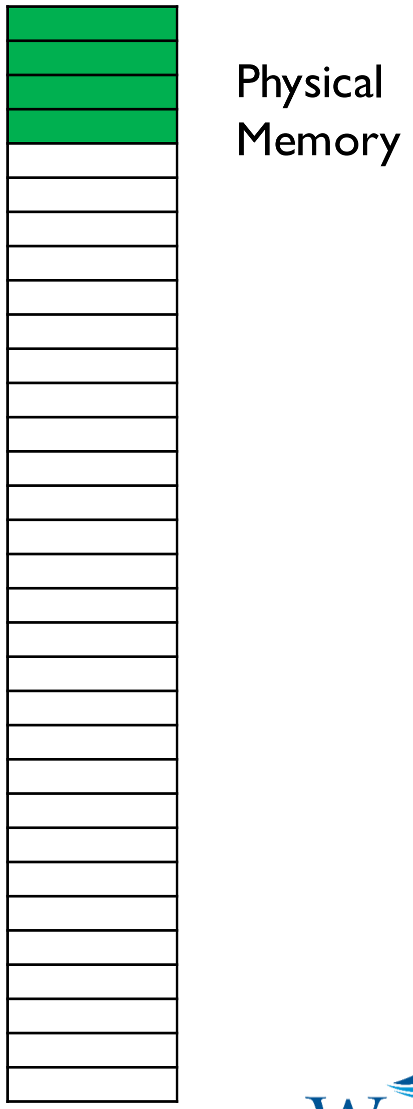


PAGING

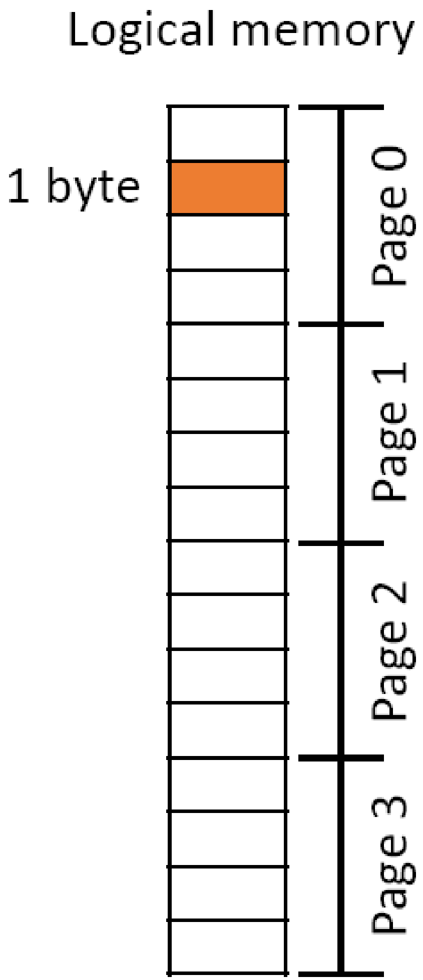


Size of logical address space : 16
Page size = 4 bytes

Q: How many bytes should each frame be?



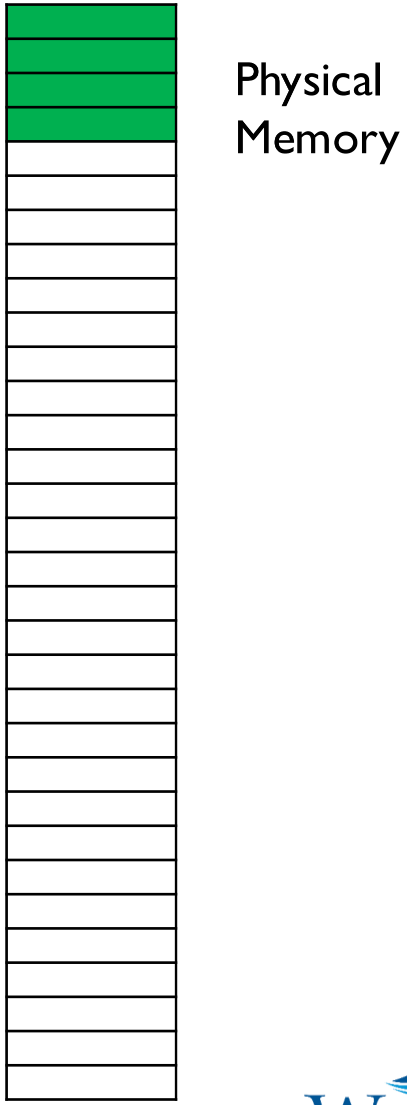
PAGING



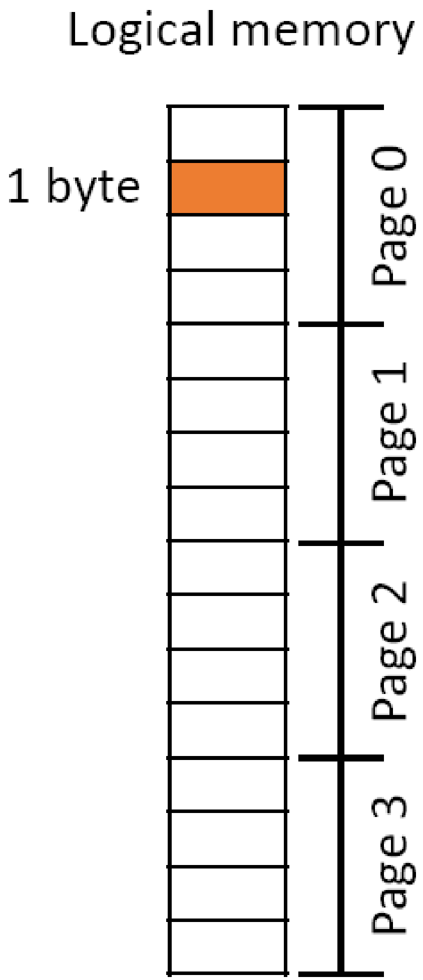
Size of logical address space : 16
Page size = 4 bytes

Q: How many bytes should each frame be?

Worksheet Q2: How many bits do we need for the logical address?



PAGING



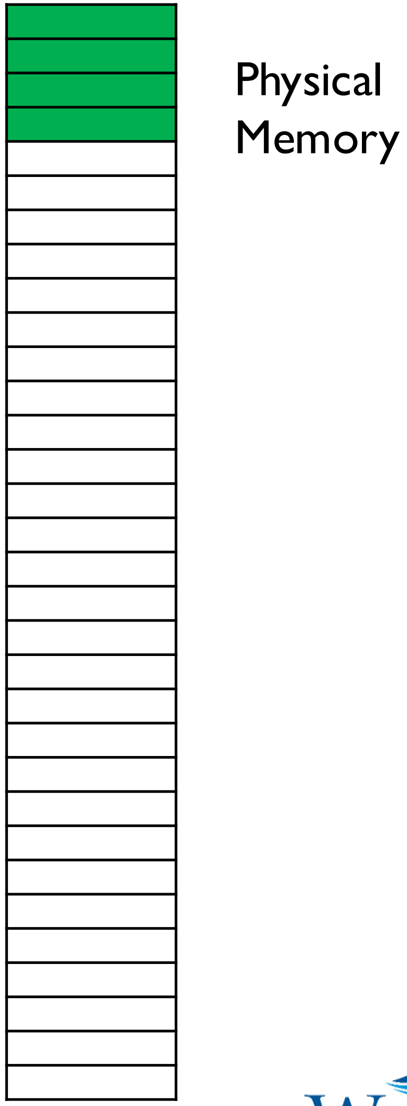
Size of logical address space : 16

Page size = 4 bytes

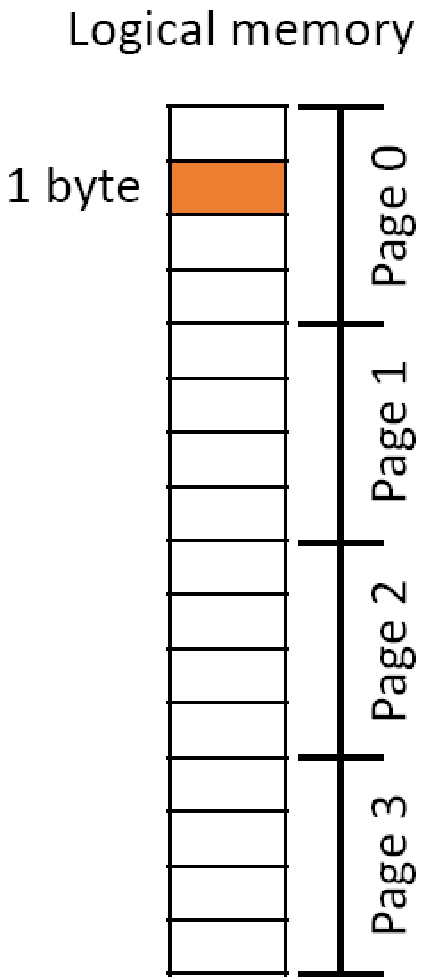
Worksheet Q2: How many bits do we need for the logical address?

m: total number of bits needed

n: number of bits needed for offset bits

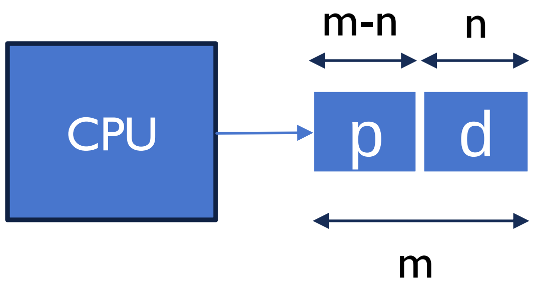


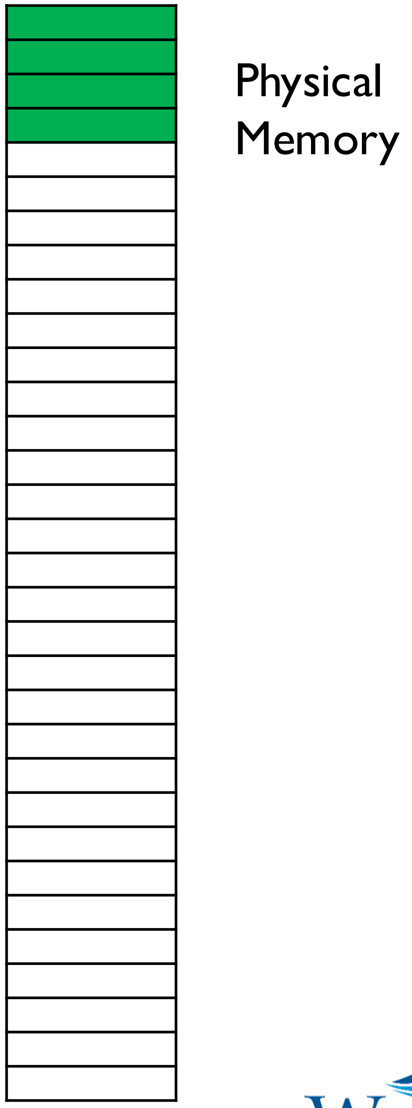
PAGING



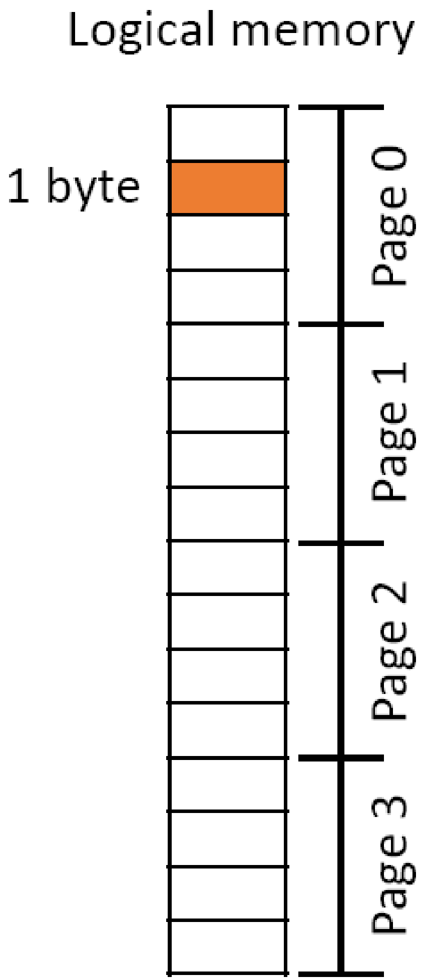
Size of logical address space : 16

Page size = 4 bytes





PAGING

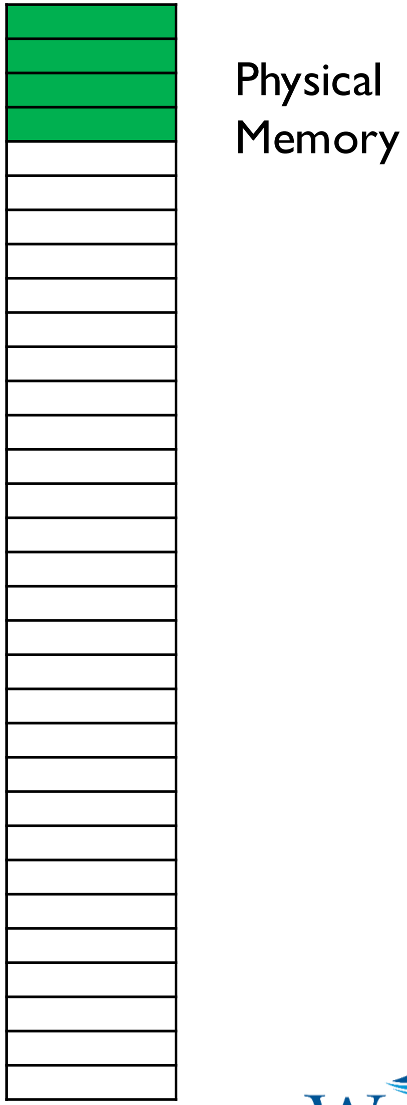


Size of logical address space : 16
Page size = 4 bytes

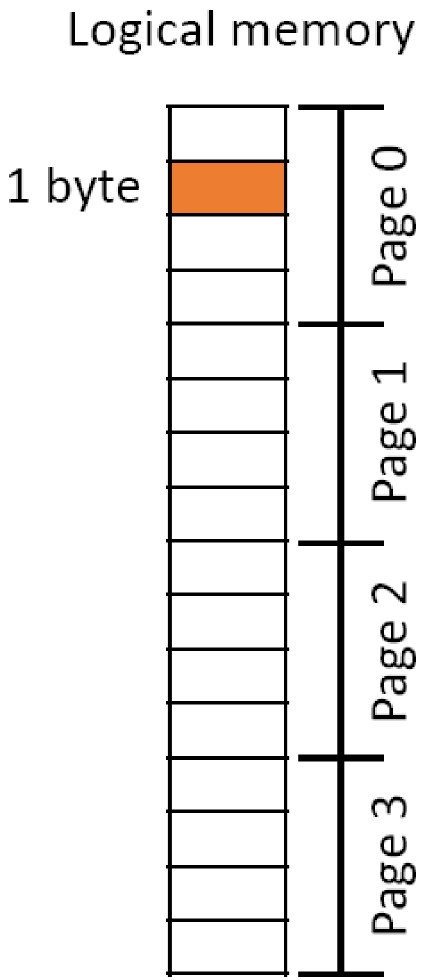
Worksheet Q2: How many bits do we need for the logical address?

m: total number of bits needed
n: number of bits needed for offset bits

Q: 16 is what power of 2? → $m = \log_2(16) = 4$



PAGING



Size of logical address space : 16

Page size = 4 bytes

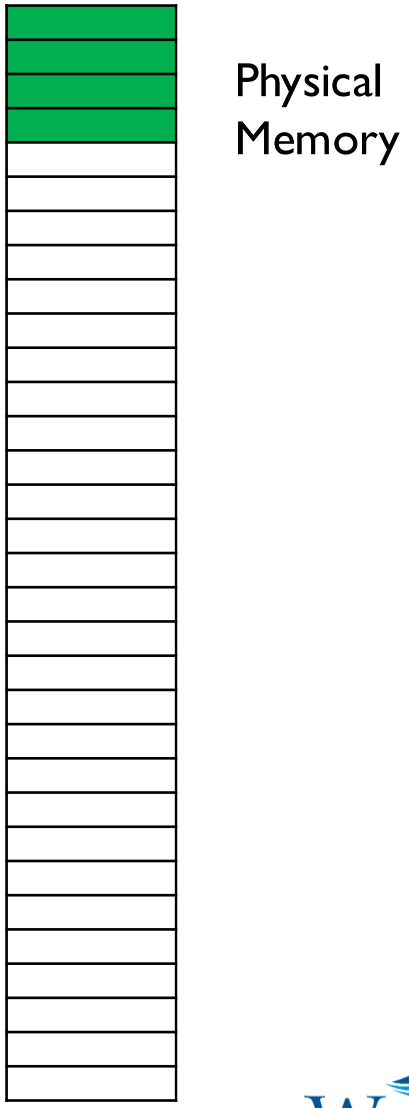
Worksheet Q2: How many bits do we need for the logical address?

m: total number of bits needed

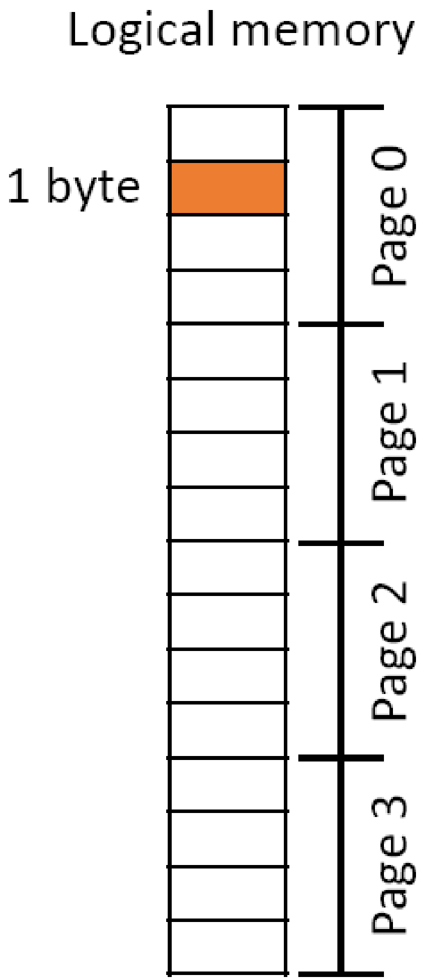
n: number of bits needed for offset bits

Q: 16 is what power of 2? → $m = \log_2(16) = 4$

Q: 4 is what power of 2? → $n = \log_2(4) = 2$



PAGING



Size of logical address space : 16

Page size = 4 bytes

Worksheet Q2: How many bits do we need for the logical address?

m: total number of bits needed

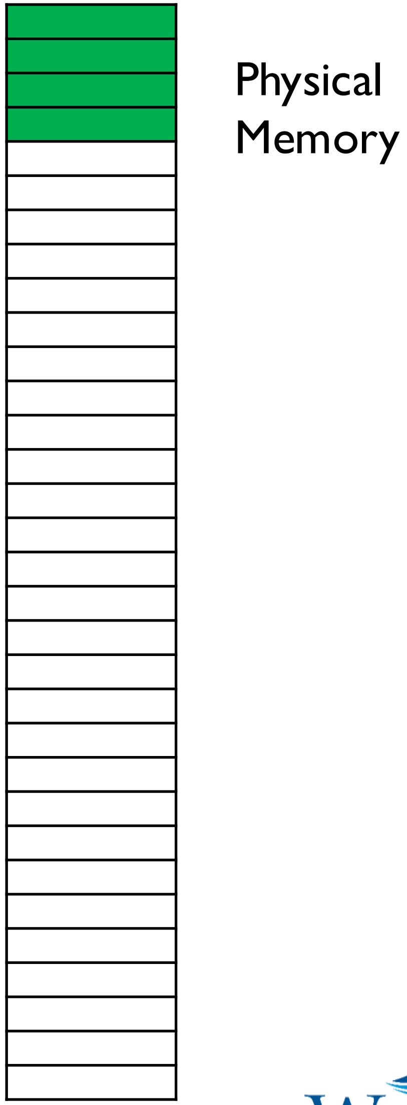
n: number of bits needed for offset bits

Q: 16 is what power of 2? → $m = \log_2(16) = 4$

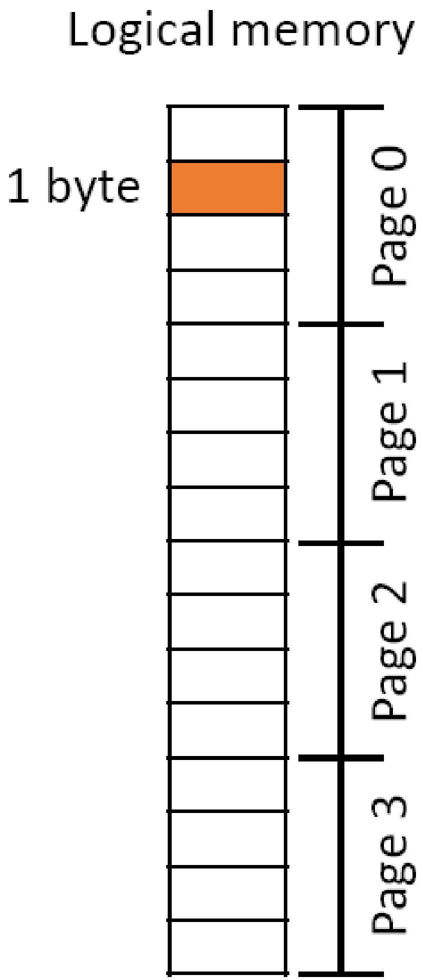
Q: 4 is what power of 2? → $n = \log_2(4) = 2$

Size of logical address space : 2^m

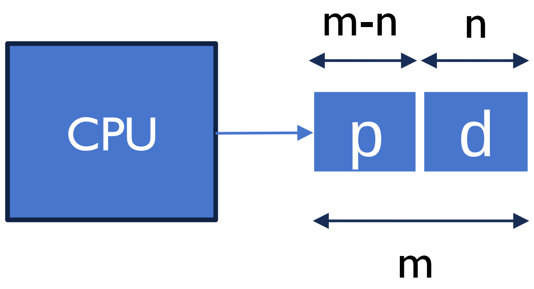
Page size : 2^n bytes



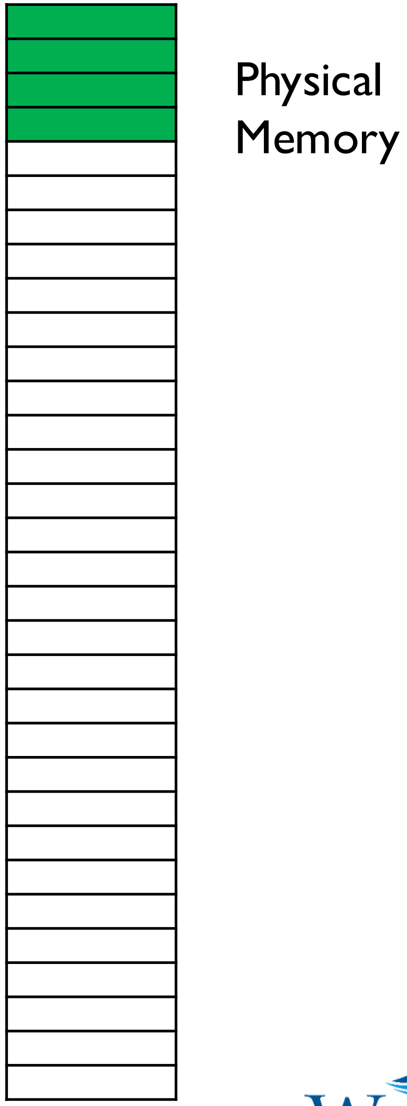
PAGING



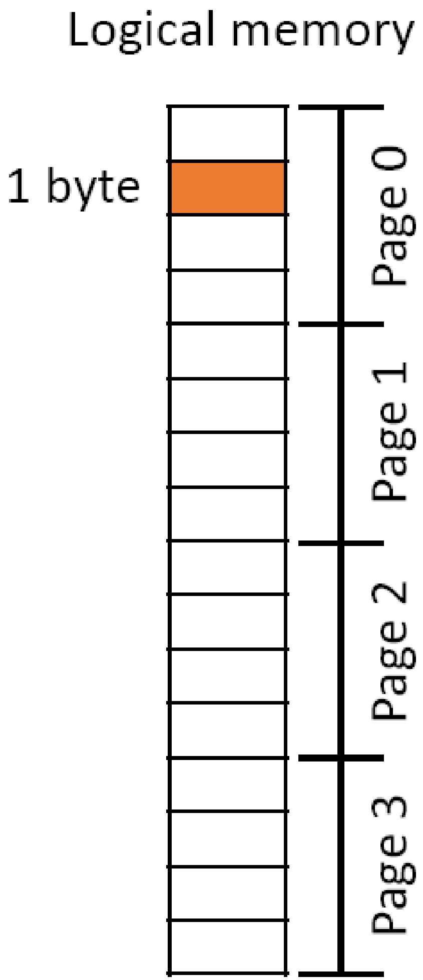
Size of logical address space : 16
Page size = 4 bytes



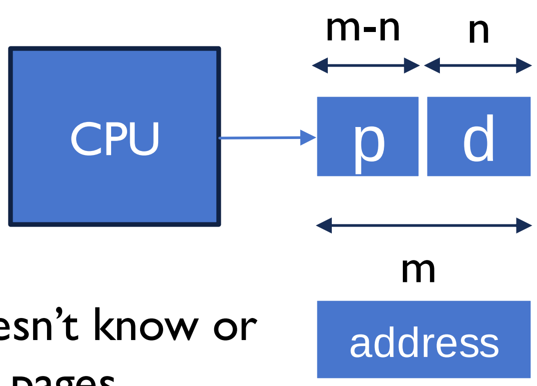
When page # and page size are both a power of 2 … logical address becomes contiguous!



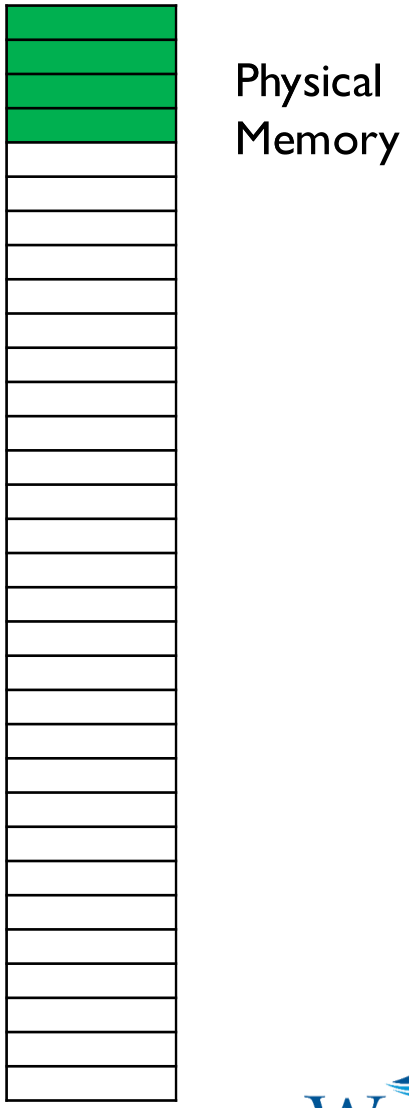
PAGING



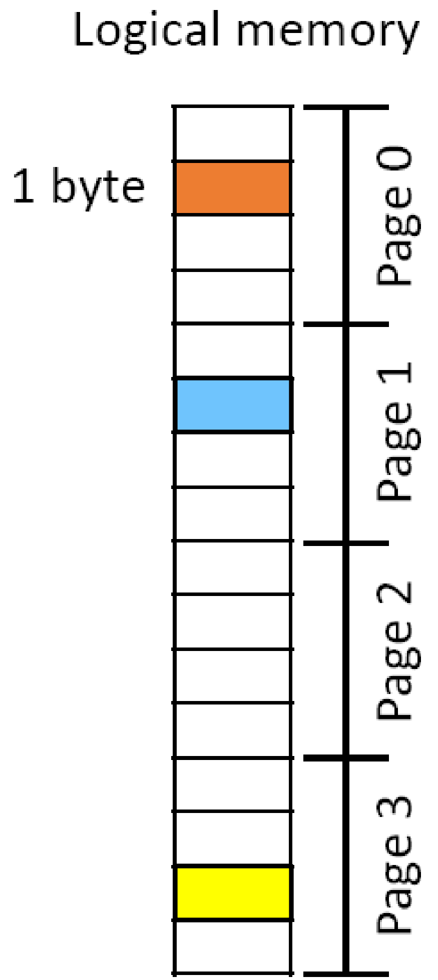
Size of logical address space : 16
Page size = 4 bytes



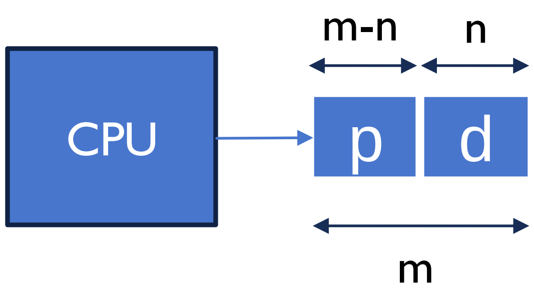
Software doesn't know or worry about pages …

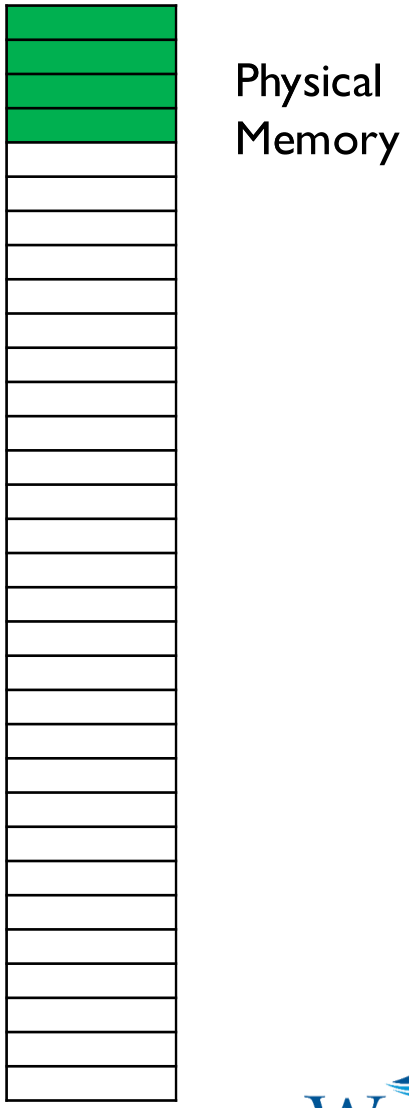


PAGING

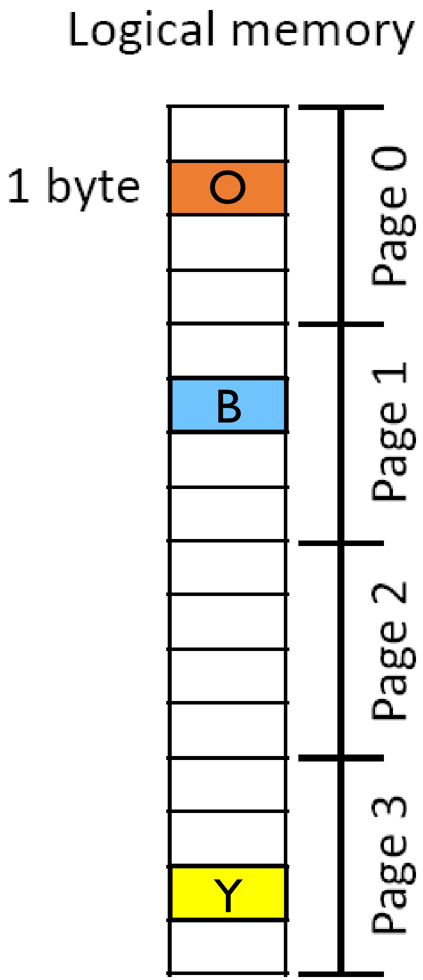


- What does this mean?
- Program doesn’t see pages, it sees contiguous memory …





PAGING

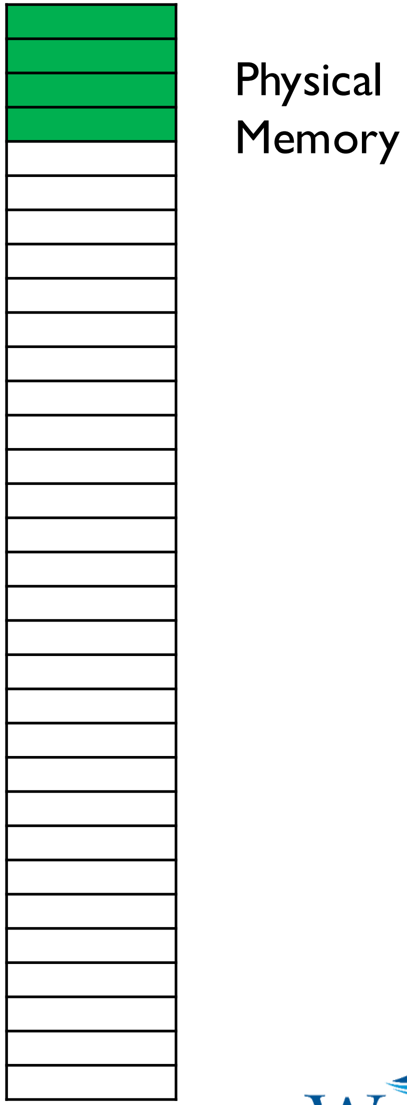


m = 4 = total size of logical address space

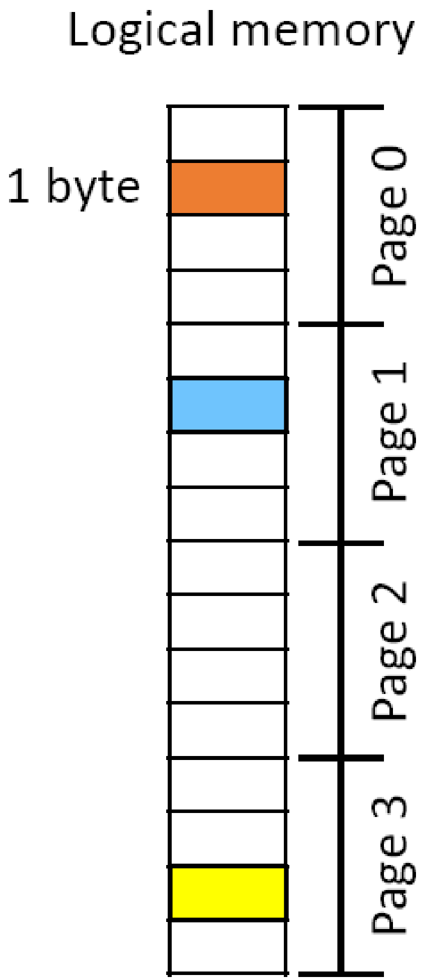
n = 2 = size of page

Worksheet Q3

Q: What is the address of the orange byte?
Q: What is the address of the blue byte?
Q: What is the address of the yellow byte?



PAGING



m = 4 = total size of logical address space

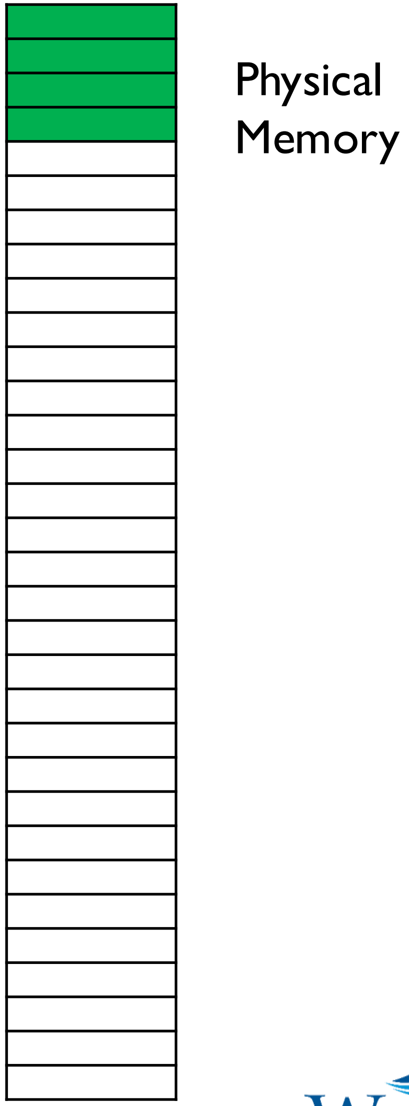
n = 2 = size of page

Q: What is the address of the orange byte?
Q: What is the address of the blue byte?
Q: What is the address of the yellow byte?

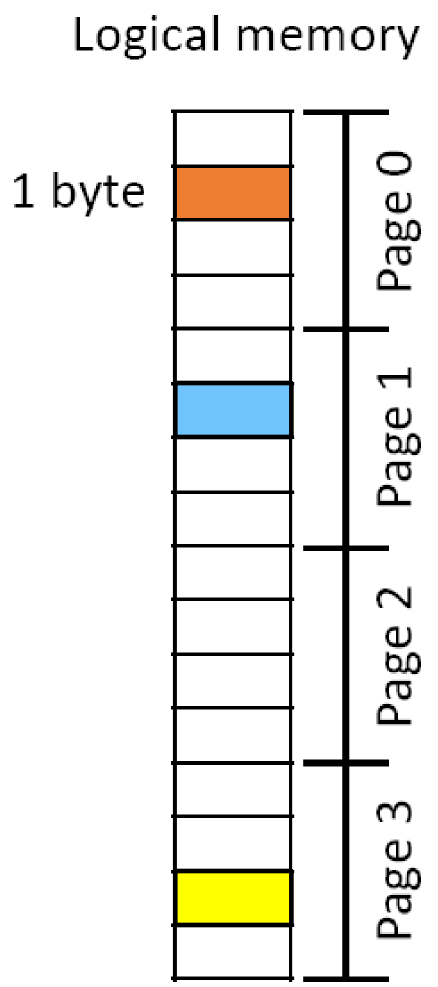
🟧 0001

🟦 0101

🟨 1110



PAGING

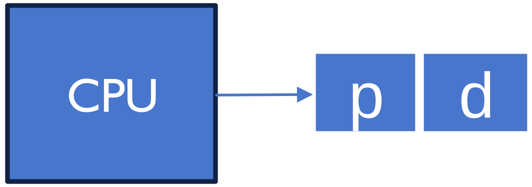


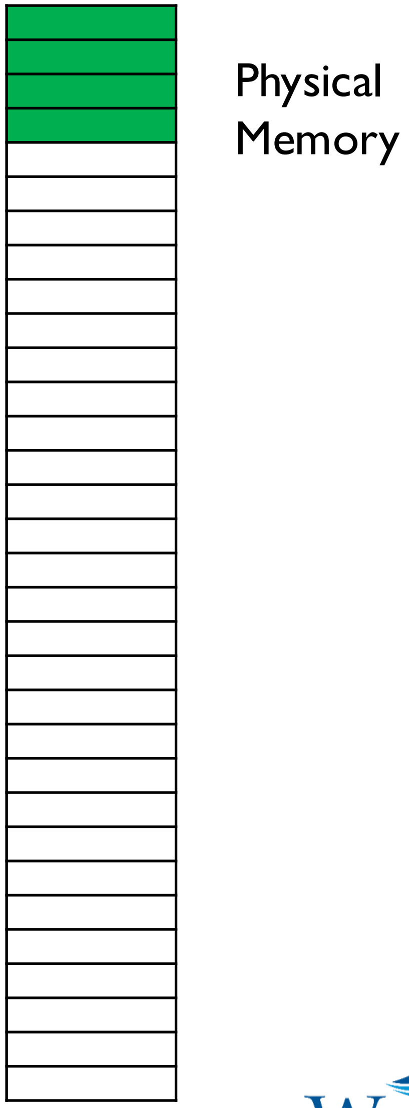
m = 4 = total size of logical address space

n = 2 = size of page

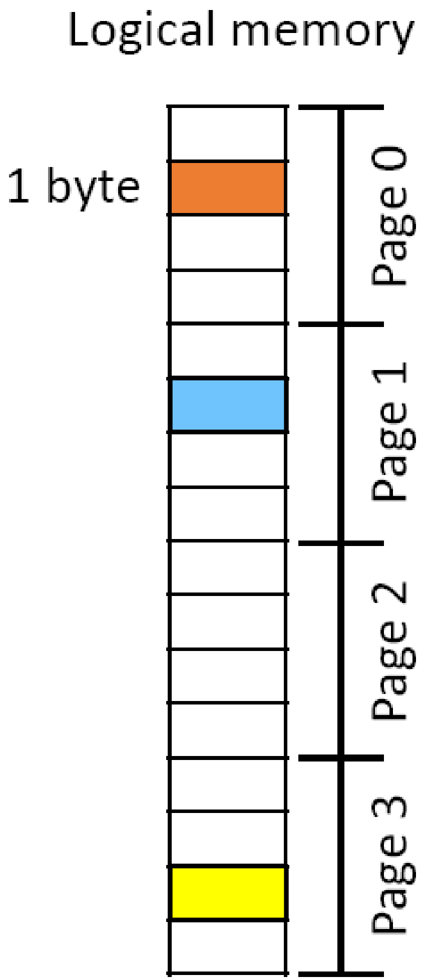
Q: What is the address of the orange byte?
Q: What is the address of the blue byte?
Q: What is the address of the yellow byte?

🟧	0001
🟦	0101
🟨	1110





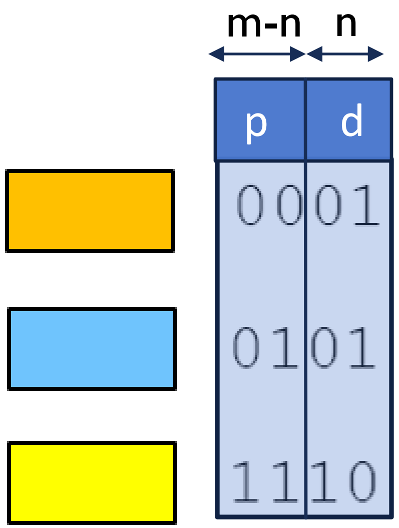
PAGING



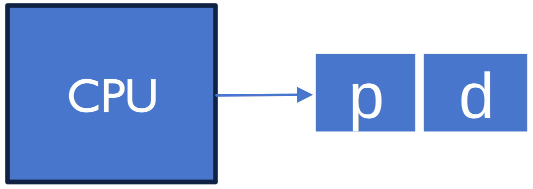
m = 4 = total size of logical address space

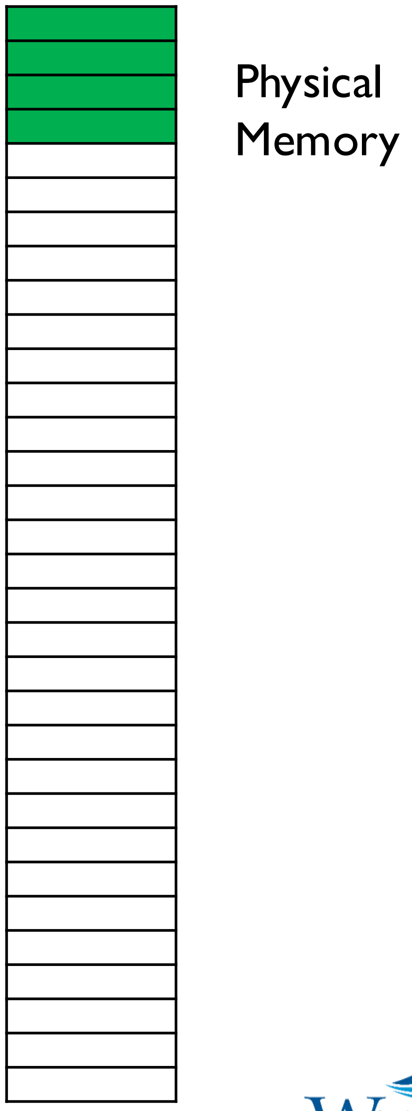
n = 2 = size of page

Q: What is the address of the orange byte?
Q: What is the address of the blue byte?
Q: What is the address of the yellow byte?



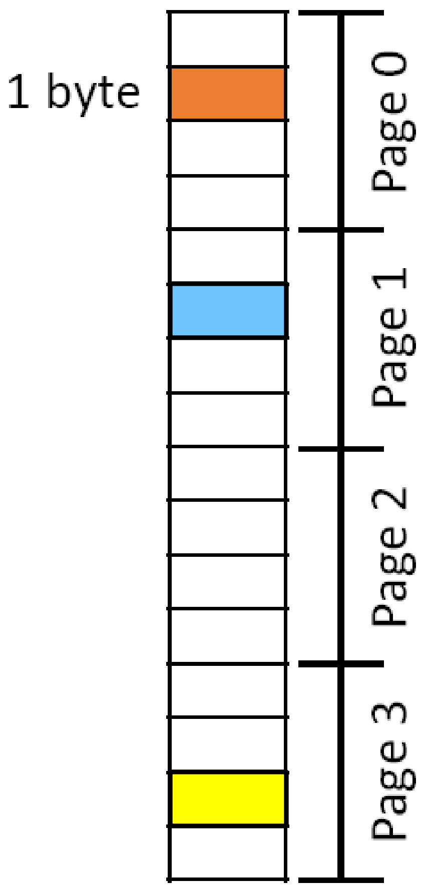
	m-n	n
	p	d
🟧	00	01
🟦	01	01
🟨	11	10





PAGING

Logical memory

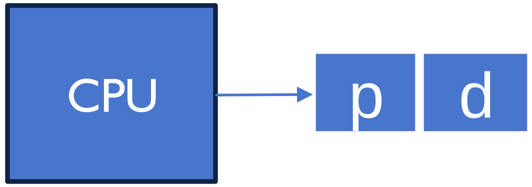


m = 4 = total size of logical address space

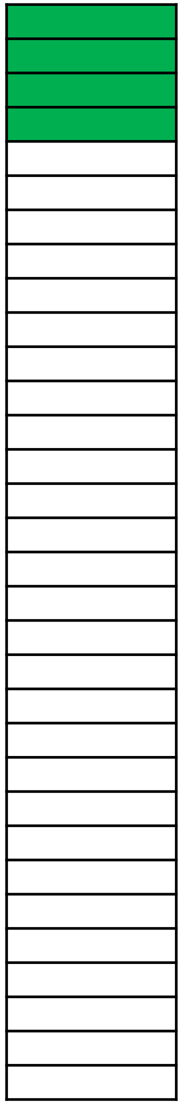
n = 2 = size of page

Q: What is the address of the orange byte?
Q: What is the address of the blue byte?
Q: What is the address of the yellow byte?

	m-n	n
	p	d
🟧	00	01
🟦	01	01
🟨	11	10



Contiguous logical memory can naturally be treated as page, offset <p,d>

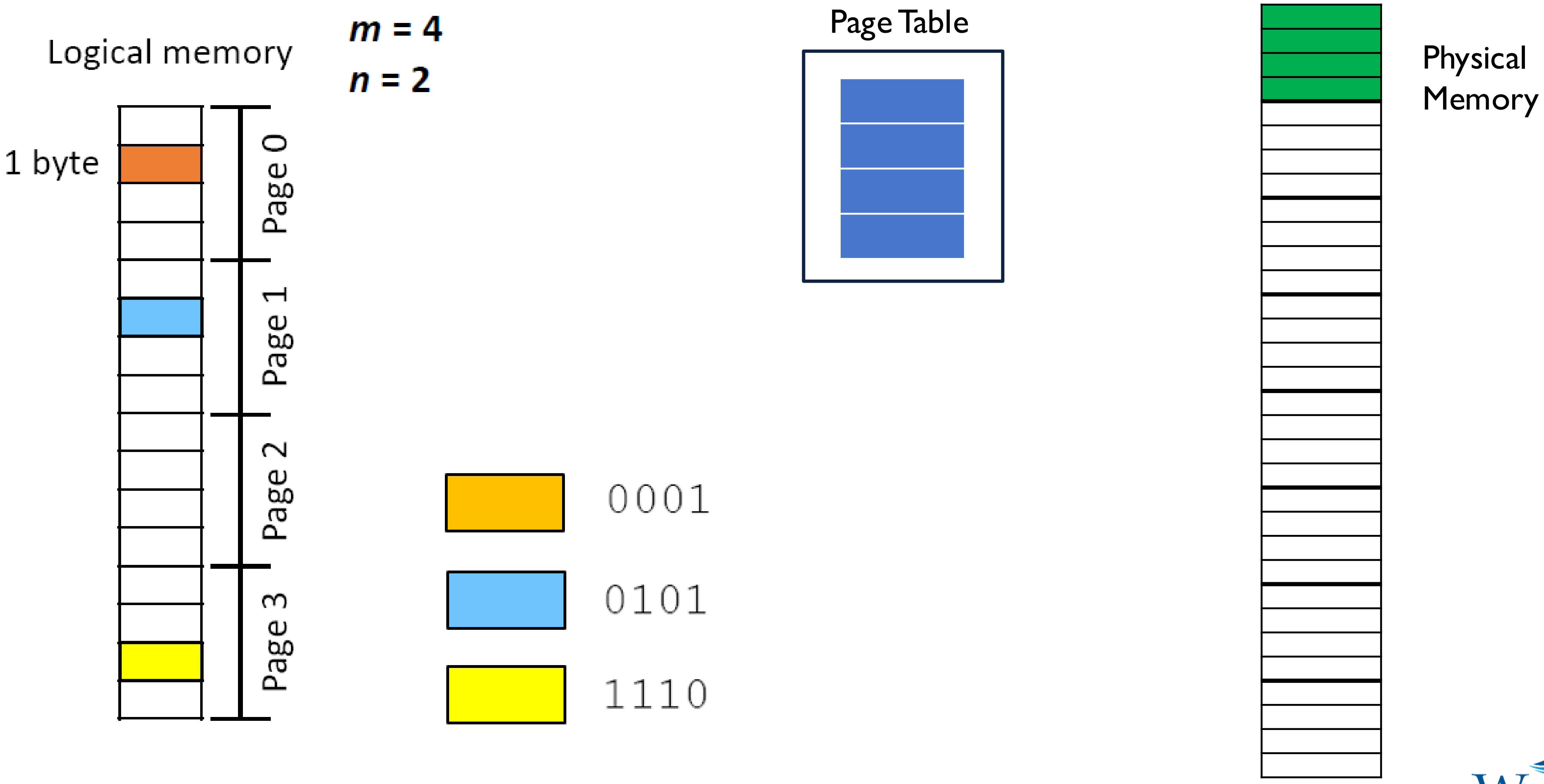


Physical Memory

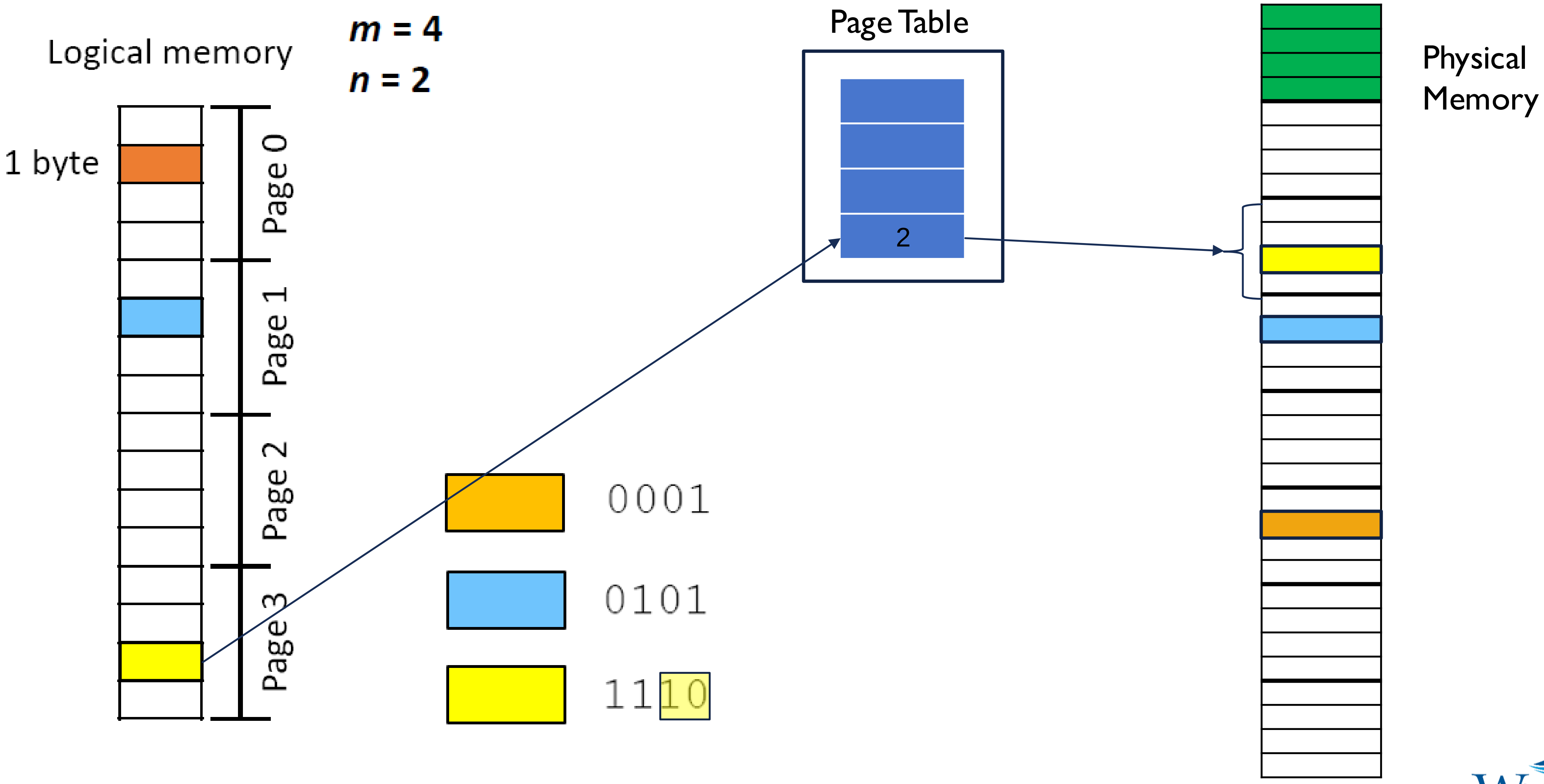
PAGING

$m = 4$

$n = 2$


Logical memory
1 byte
Page 0
Page 1
Page 2
Page 3
Page Table
Physical Memory
0001
0101
1110

PAGING



$m = 4$

$n = 2$

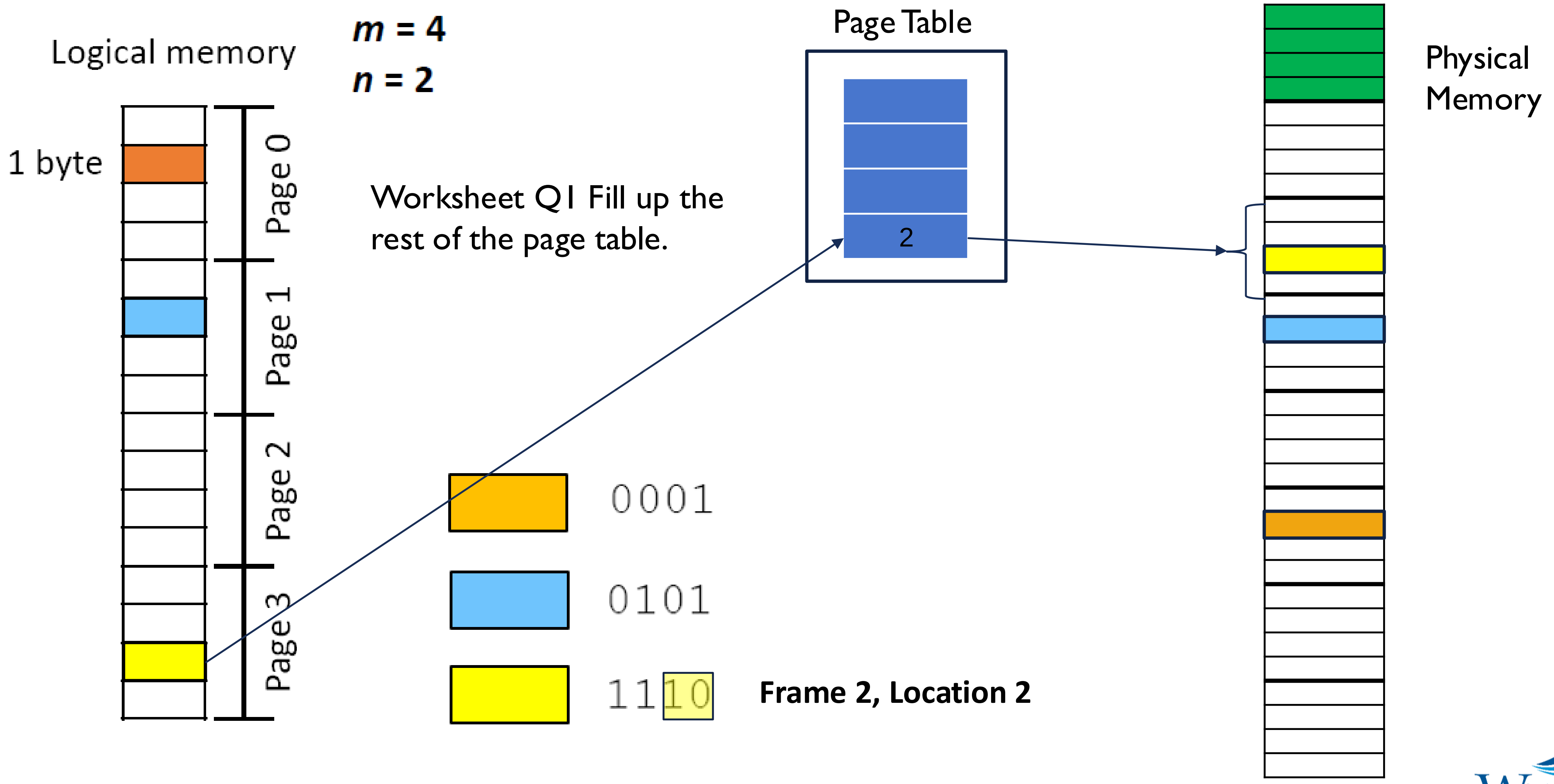
Logical memory
1 byte
Page 0
Page 1
Page 2
Page 3
Page Table
2
Physical Memory
0001
0101
1110

PAGING

***m* = 4**

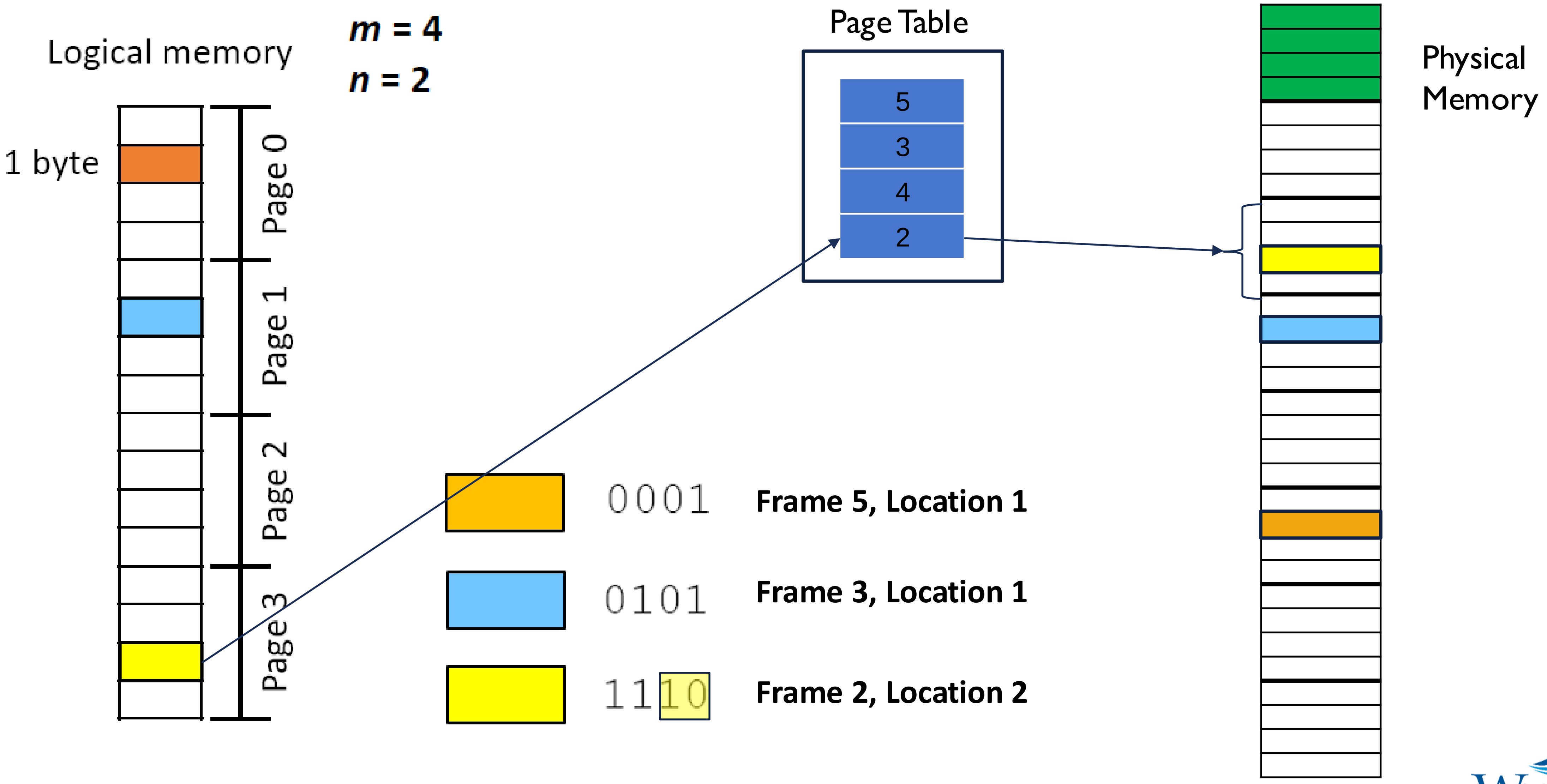
***n* = 2**

Worksheet Q1 Fill up the rest of the page table.



Frame 2, Location 2

PAGING



Logical memory

m = 4

n = 2

1 byte

Page 0

Page 1

Page 2

Page 3

Page Table

5

3

4

2

Physical Memory

🟧 0001 **Frame 5, Location 1**

🟦 0101 **Frame 3, Location 1**

🟨 1110 **Frame 2, Location 2**