Operating System

Chapter 8

Contents

1.Introduction

Why do we need to study Computer Architecture?

- Understanding Computer Functionality
- Hardware Design and Optimization
- Software Development
- Compatibility and Portability
- Troubleshooting and Debugging
- System Administration
- Security
- Innovations and Advances
- Career Opportunities
- Academic Pursuits

Why do we need an OS?

- **Convenience :** It becomes easier for a programmer to code
- **Resource Management :**
  - Allocating memory for the user programs
  - Scheduling time for the user programmers

Computer Hardware and Software Structure

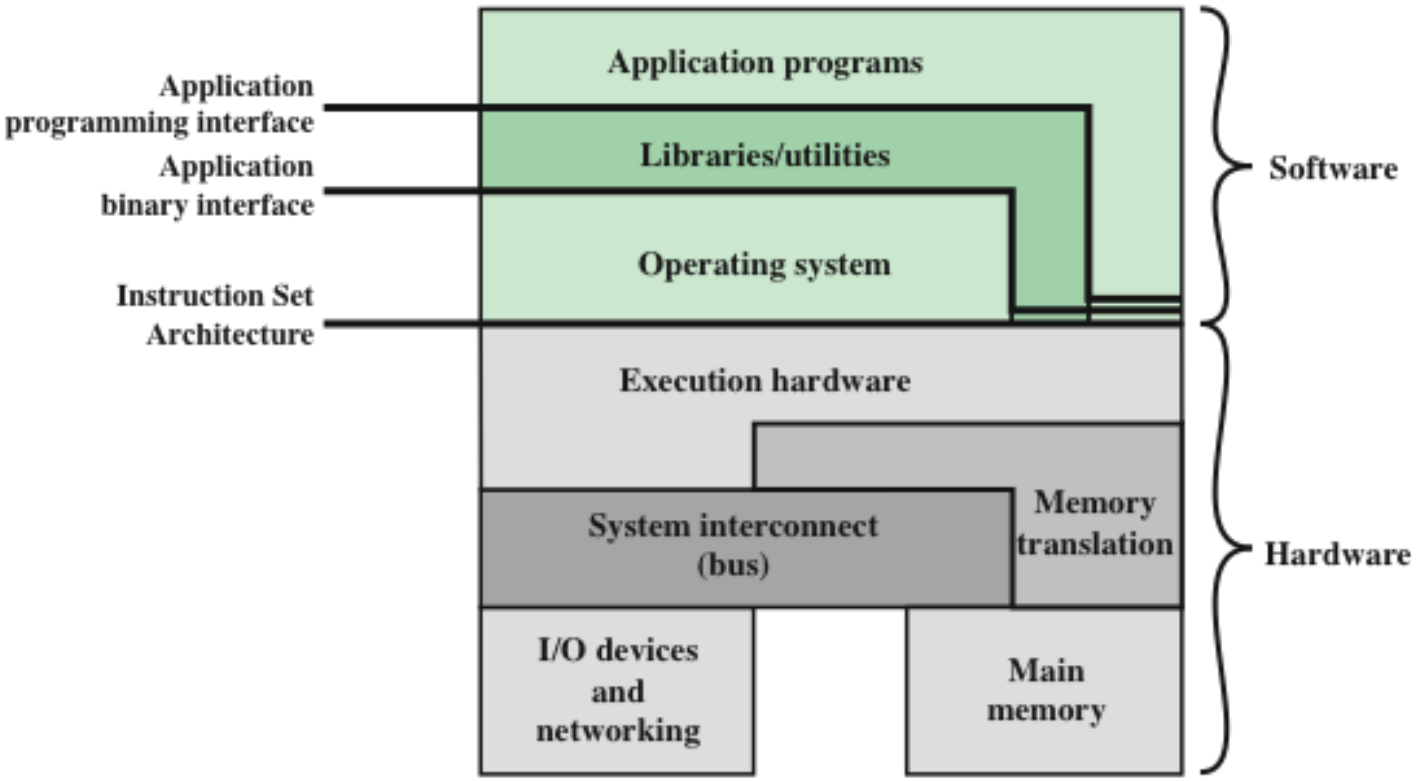


Figure 8.1 Computer Hardware and Software Structure

The services provided by the Operating System

- **Helping in writing codes -** OS has different utility functions. These functions can be called by our programs . For example i/o functions.
- **Helping in program execution –** Allocating RAM, I/O devices.
- **Controlled access to files** – Helps in determining which program has permission to access a file.
- **Error Detection and Response –** Detecting an error and taking appropriate measures.
- **Interrupt :** Contains functions for handling different interrupts.
- **Accounting –** Keeping log of CPU usage, error logs.

The OS as a resource manager:

A program for running other programs:

- OS functions the same way as **any other computer program.**
- It gives **control to the processor** for **running other programs**, the processor **returns back the control to it**.
- **Timer:**
  - Whenever the control is released to another programmer (PC points to the start of that program)
  - A timer is started
  - Once the timer becomes zero
  - The control is returned back to the OS (i.e the PC points to the OS code)

The OS as a resource manager:

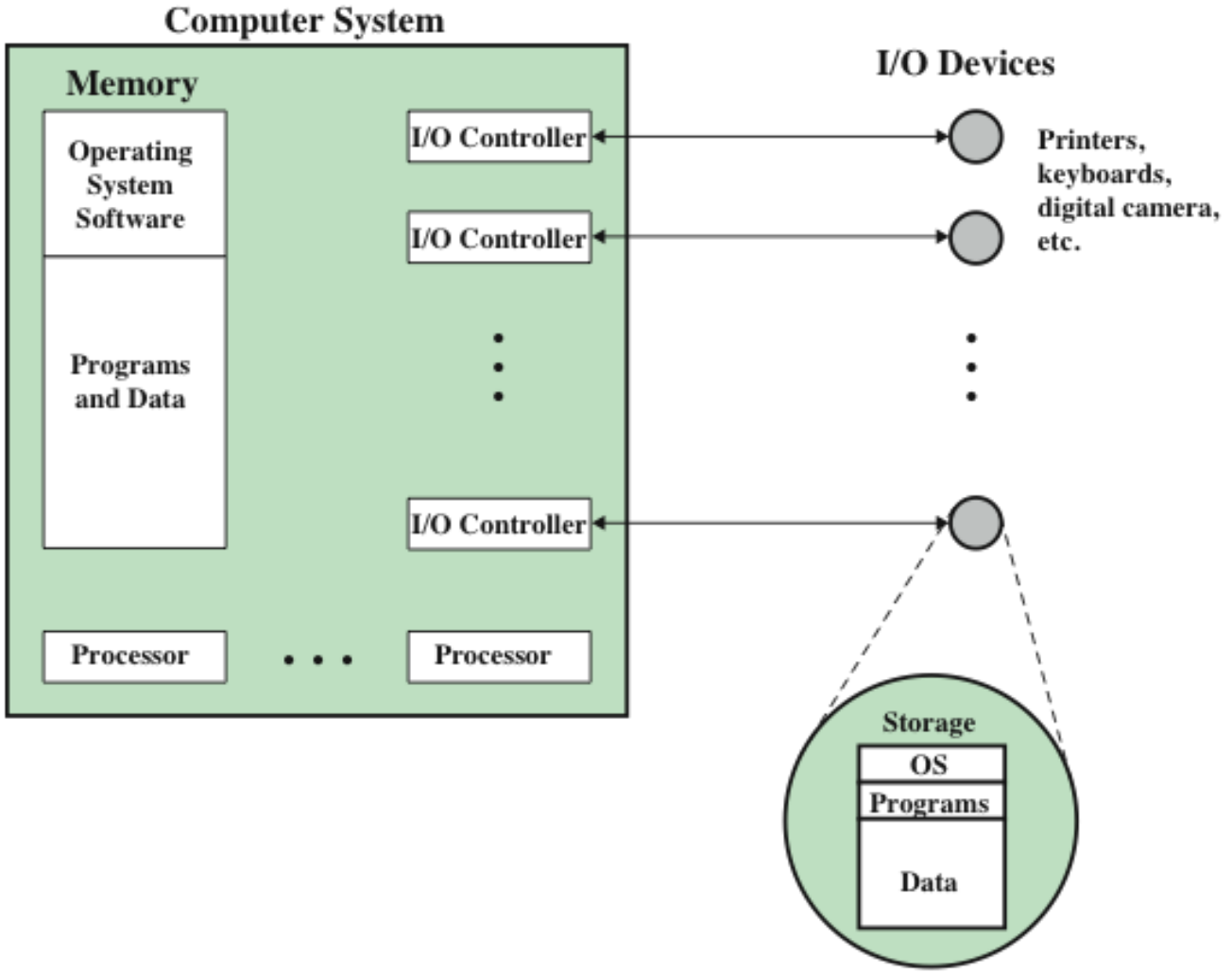


Figure 8.2 The Operating System as Resource Manager

Types of Operating Systems

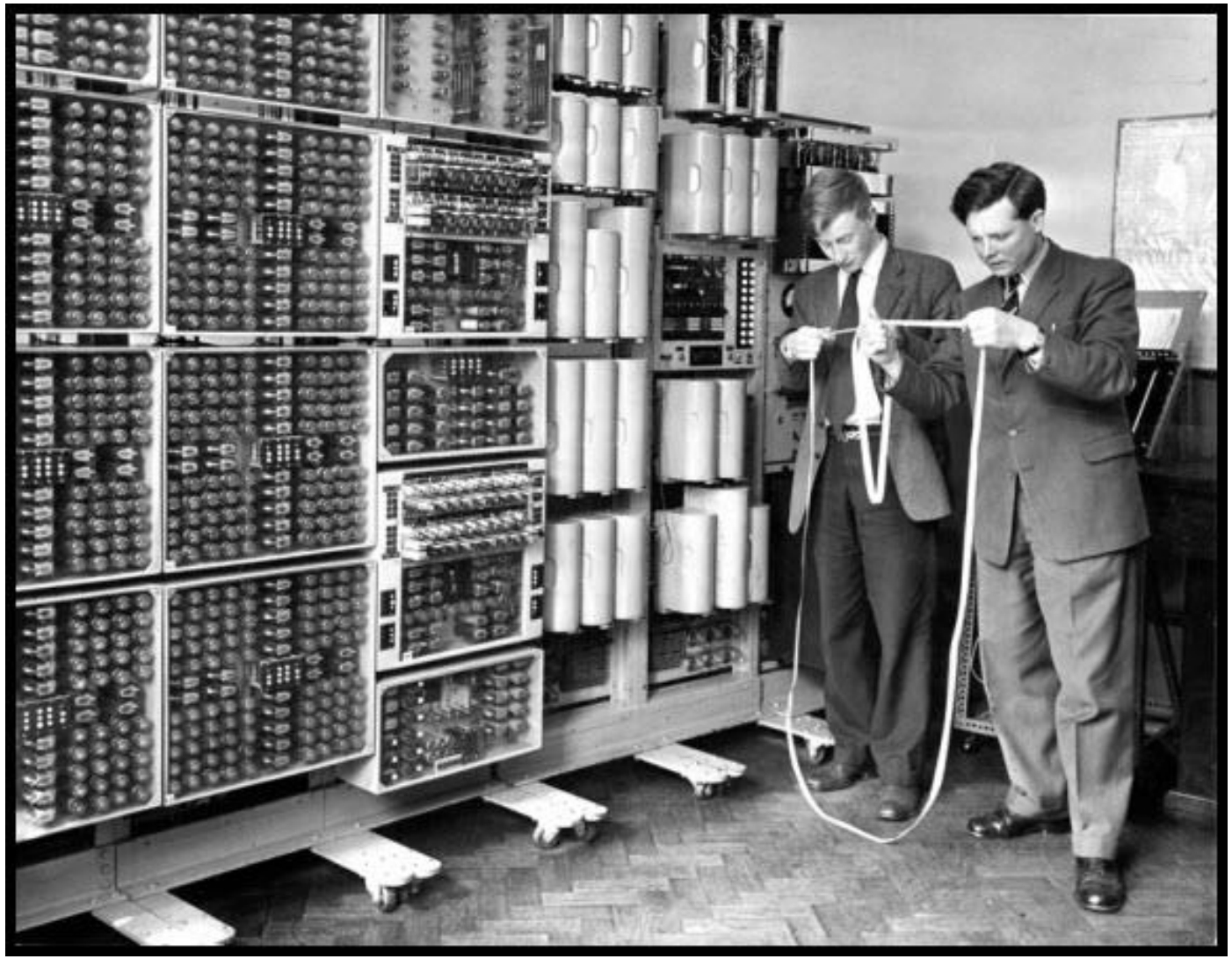
- Interactive system
  - The user/programmer interacts directly with the computer to request the execution of a job or to perform a transaction
  - User may, depending on the nature of the application, communicate with the computer during the execution of the job
- Batch system
  - Opposite of interactive
  - The user's program is batched together with programs from other users and submitted by a computer operator
  - After the program is completed results are printed out for the user

Types of Operating Systems

- Multiprogramming
  - Keep processor as busy as possible
  - Work on more than one program at a time
  - Processor switch rapidly among the programs

- Uniprogramming
  - Works only one program at a time

Early Systems



Early Systems

- From the late 1940s to the mid-1950s the programmer interacted directly with the computer hardware – there was no OS
  - Processors were run from a console consisting of display lights, toggle switches, some form of input device and a printer
- Problems:
  - Scheduling
    - Sign-up sheets were used to reserve processor time
      - This could result in wasted computer idle time if the user finished early
      - If problems occurred the user could be forced to stop before resolving the problem
  - Setup time
    - A single program could involve
      - Loading the compiler plus the source program into memory
      - Saving the compiled program
      - Loading and linking together the object program and common functions

2.1.Scheduling

Process

Process : basically refers to a running program and the components that are helping it to run

- A program in execution
- Registers being used
- Program counters
- Other meta info (process id, running time, children ids, etc)

Meta data maintained for a process by the OS

- **Identifier** – every process has a unique id
- **State** – we will see this soon – each process has a state
- **Priority** – some of the processes have higher priorities
  (e.g – device driver programs)
- **Program Counter** – the program counter value of that process
- **Starting** and **ending location** of the process code
- **Register data** – the register values
- **I/O status** - Was the process waiting for some input
- **Log** – Other information (running time, no. of executions etc.)

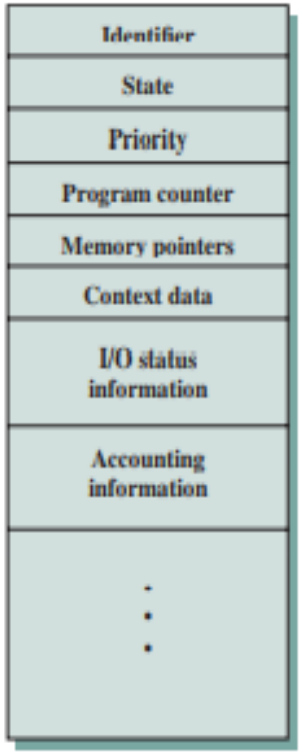
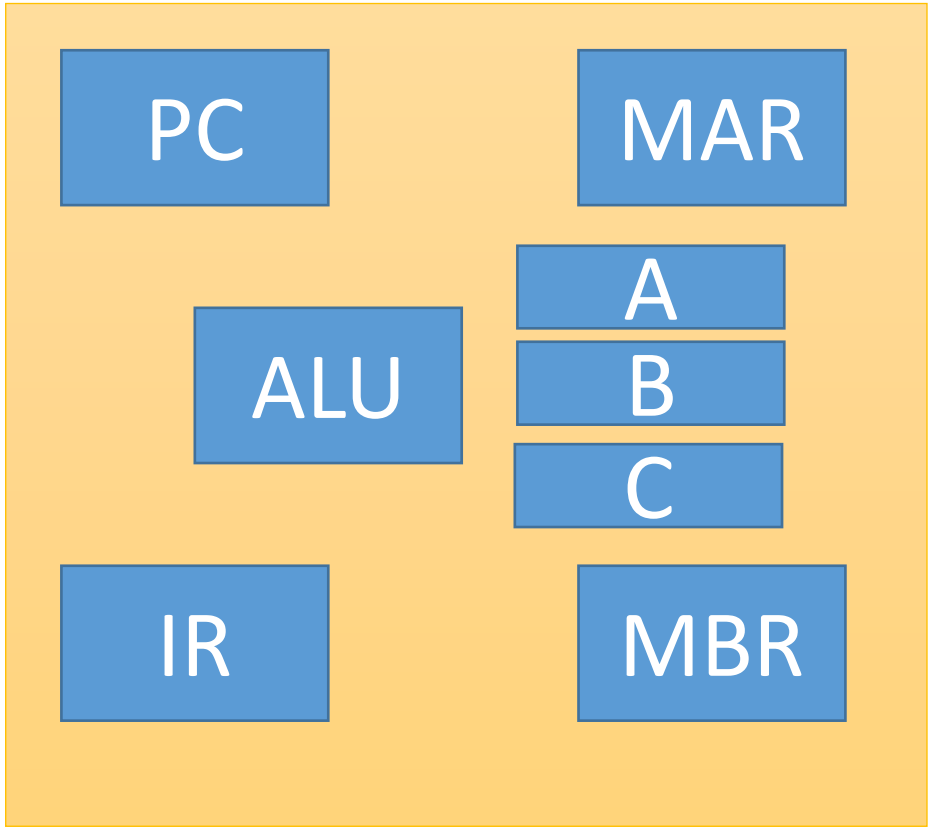


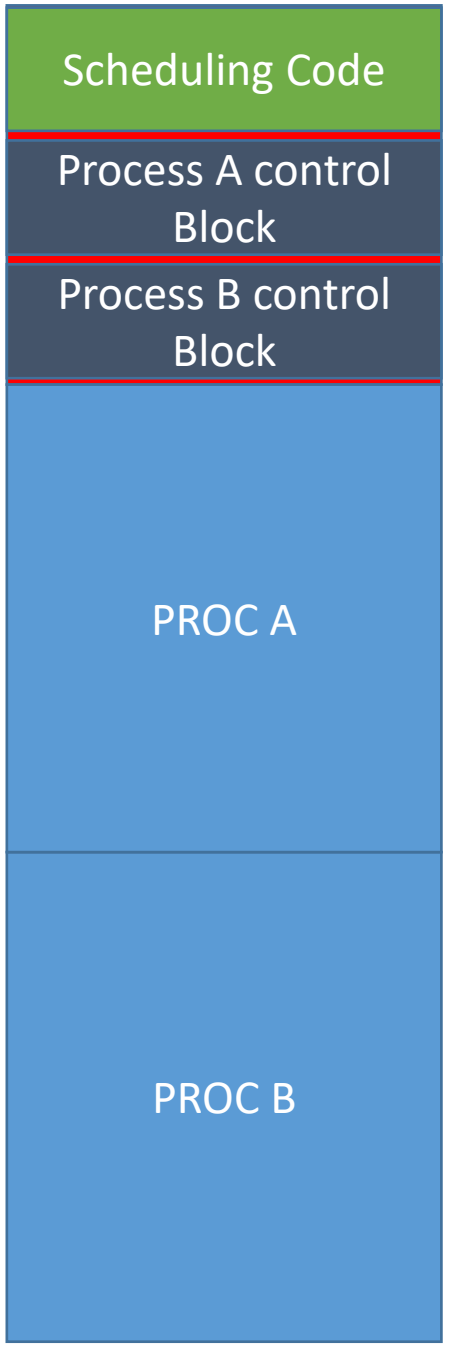
Figure 8.8 Process Control Block

(The memory of the OS where all the info of a process is stored is called the **process control block**.)

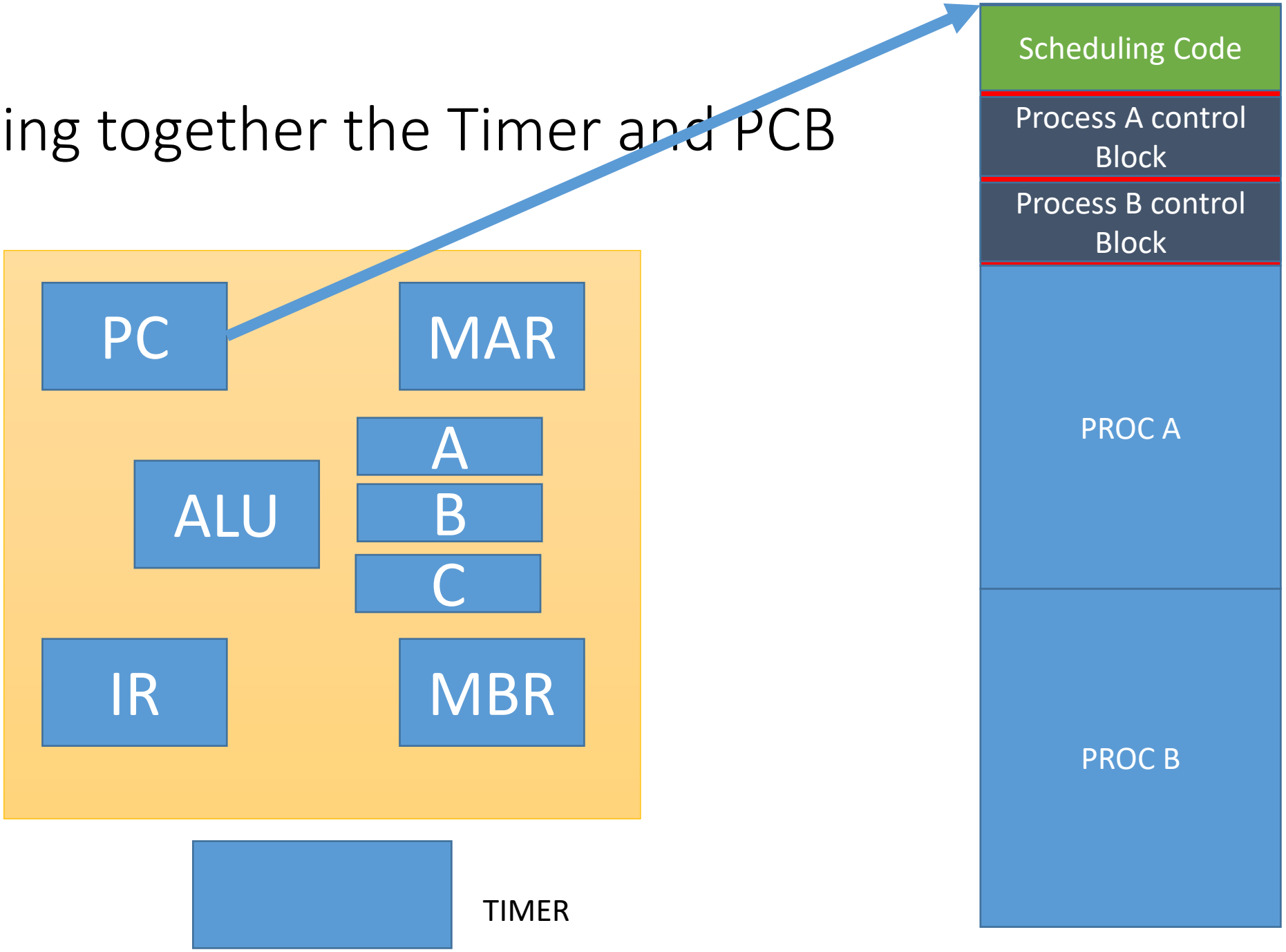
Putting together the Timer and PCB



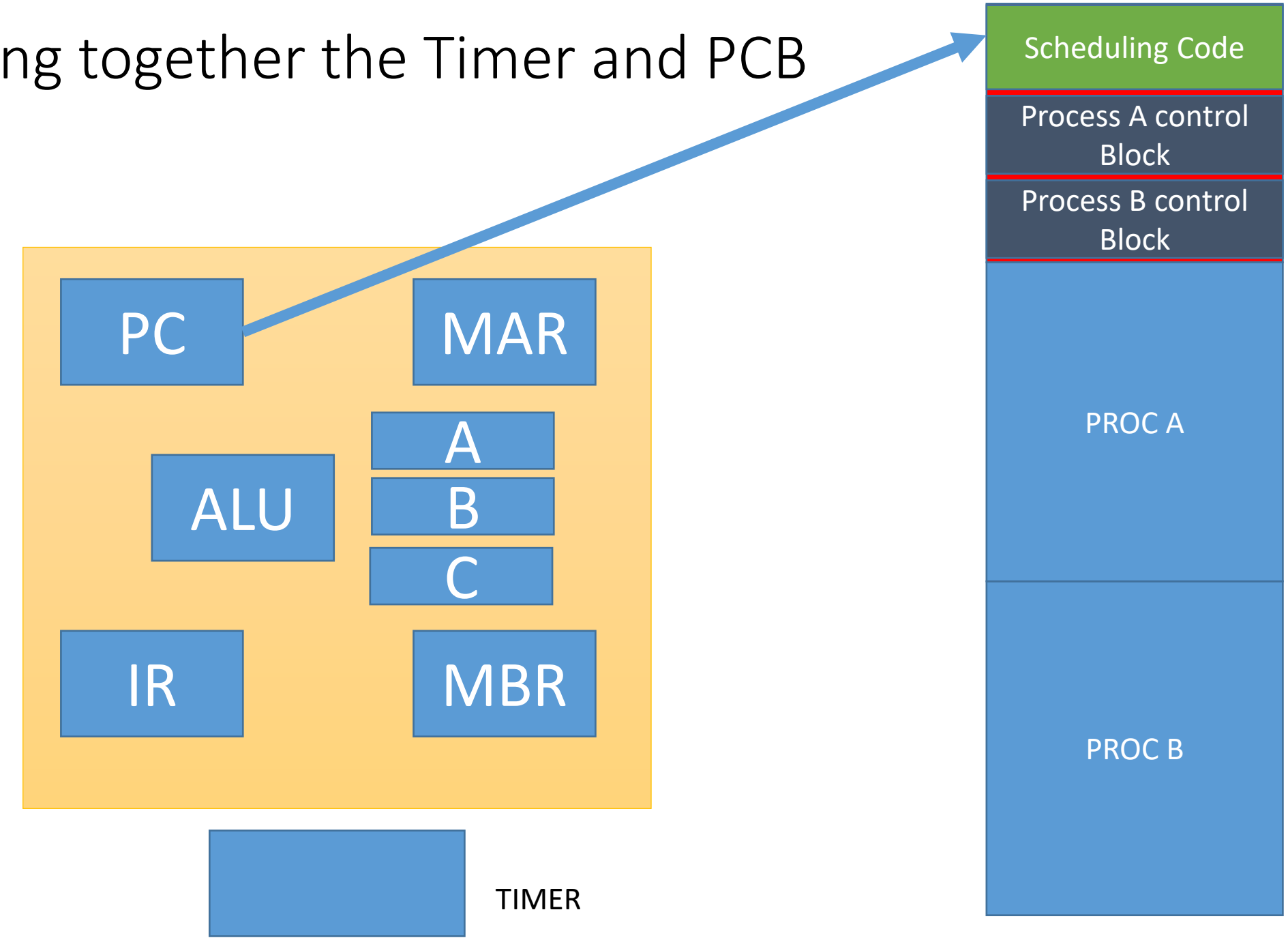
TIMER



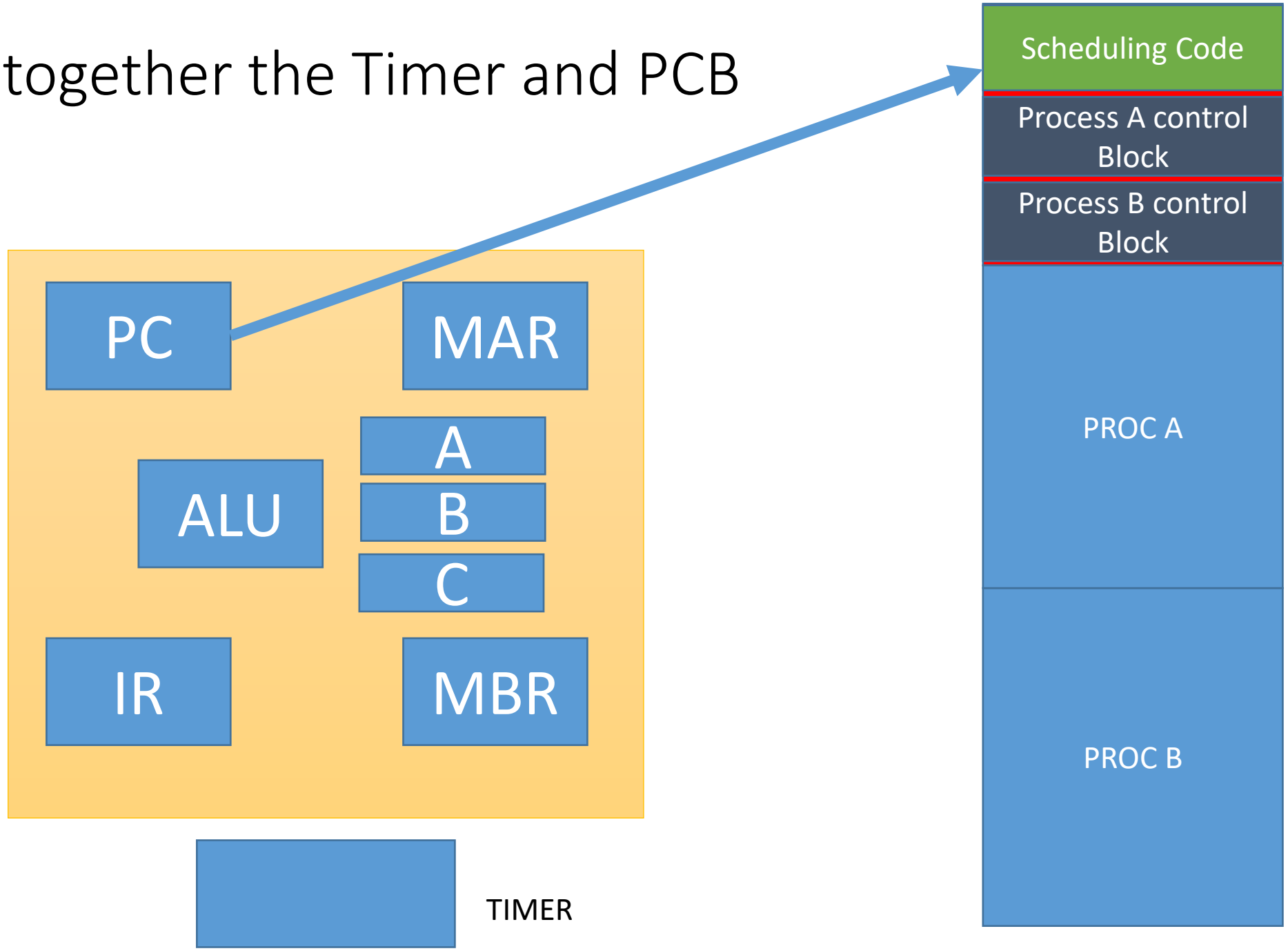
Putting together the Timer and PCB



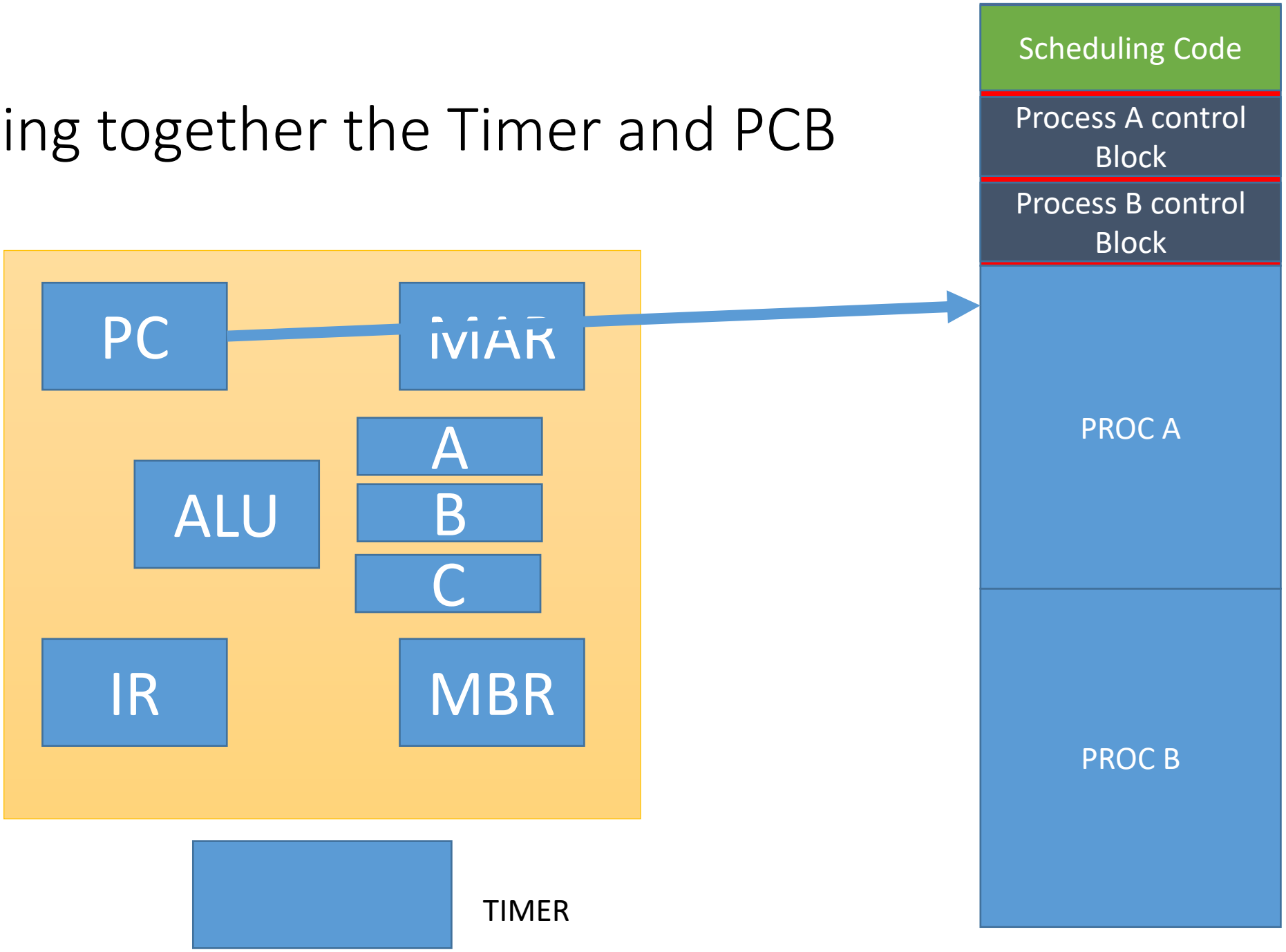
Putting together the Timer and PCB



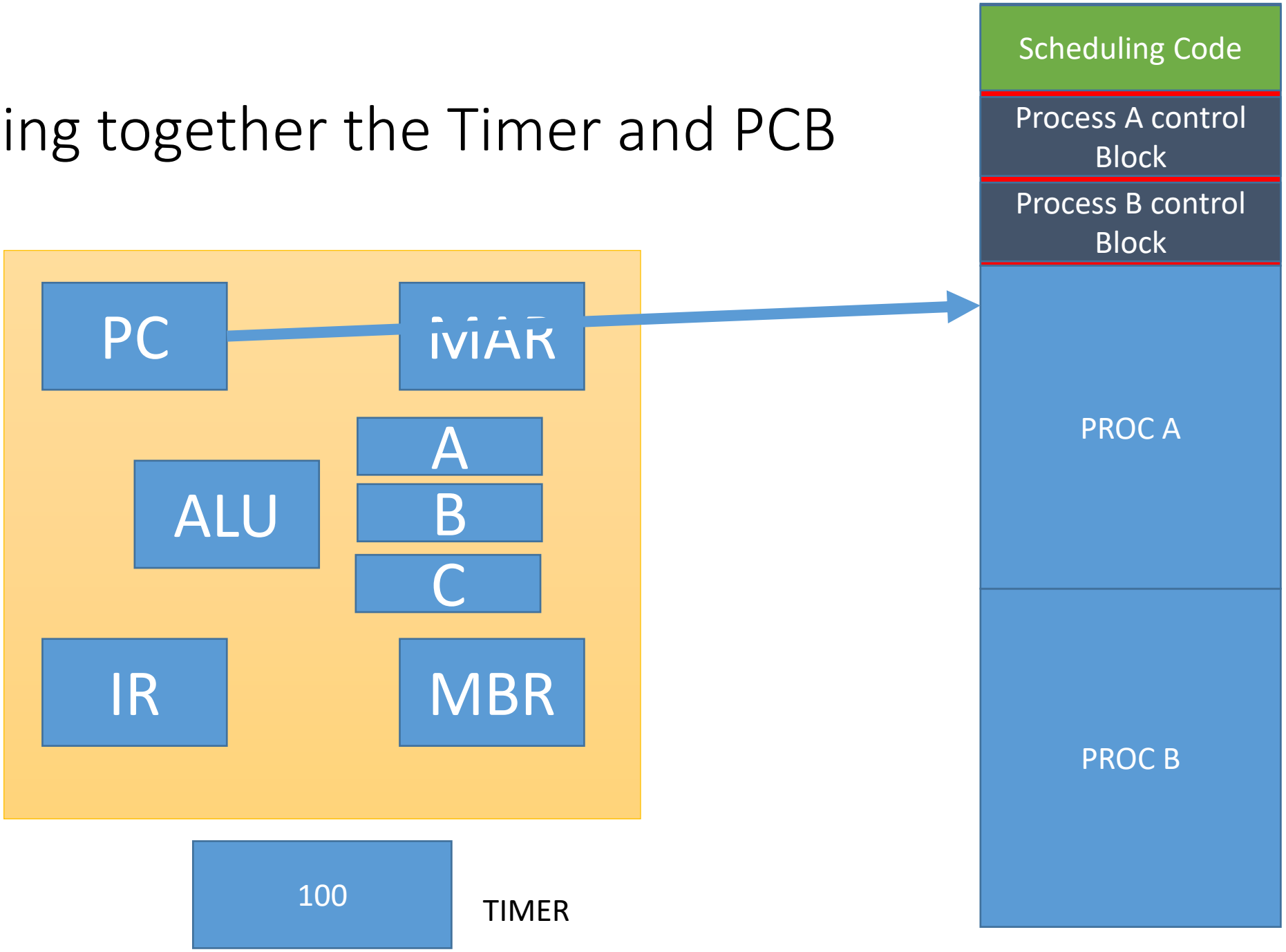
Putting together the Timer and PCB



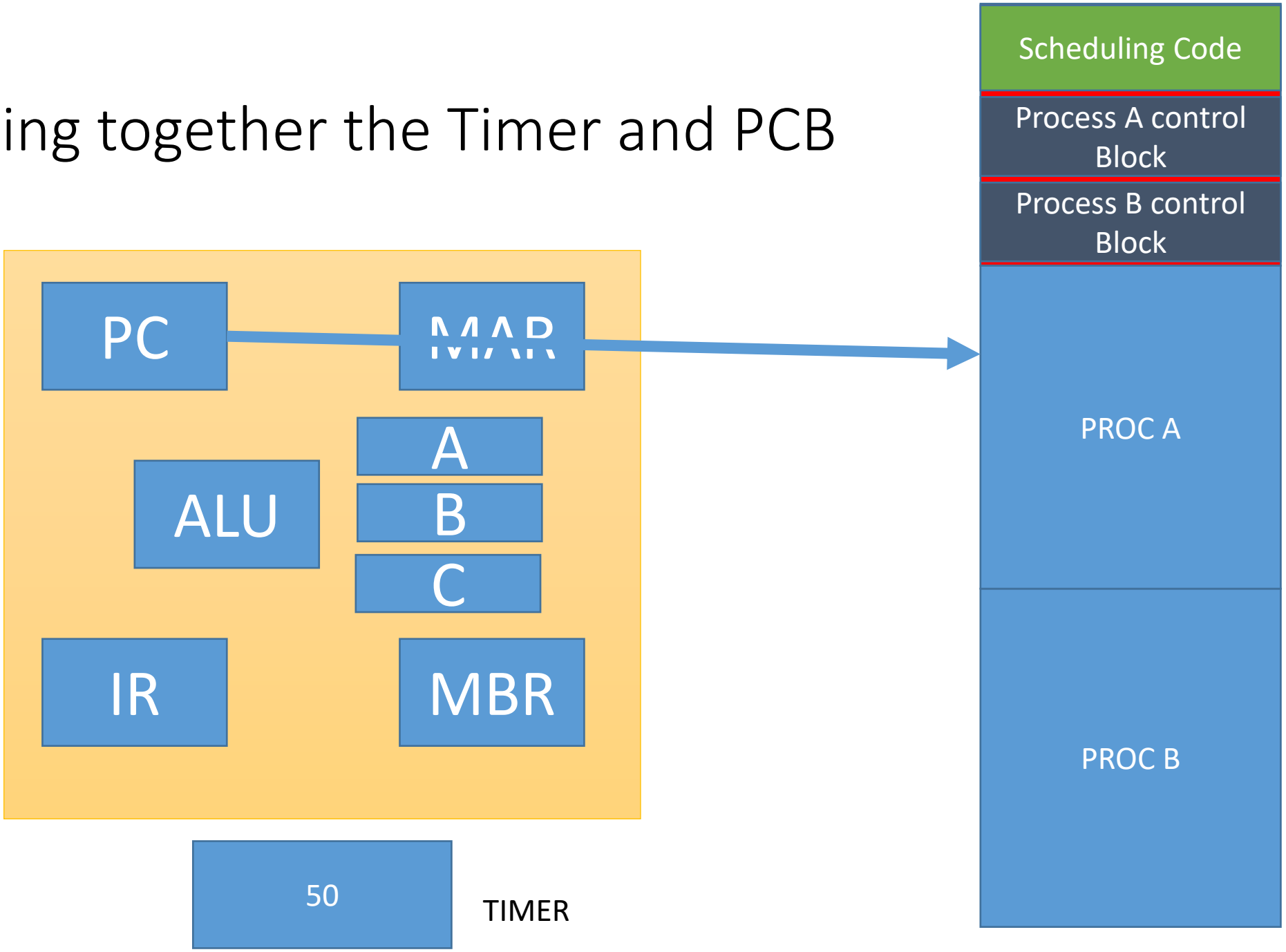
Putting together the Timer and PCB



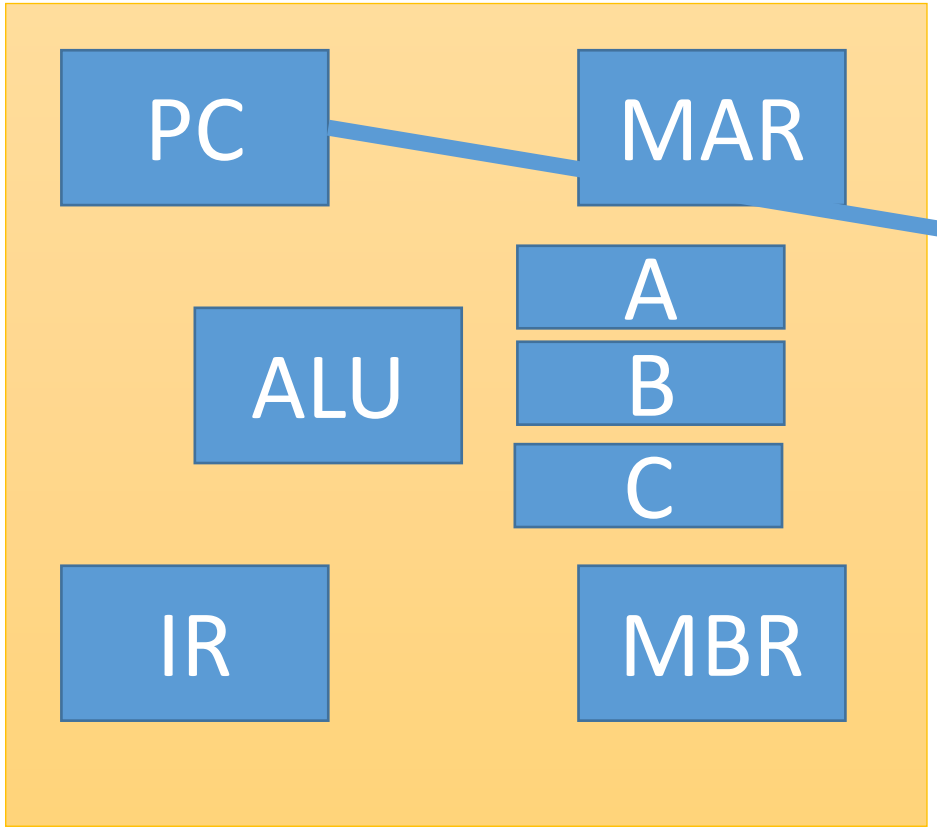
Putting together the Timer and PCB



Putting together the Timer and PCB



Putting together the Timer and PCB

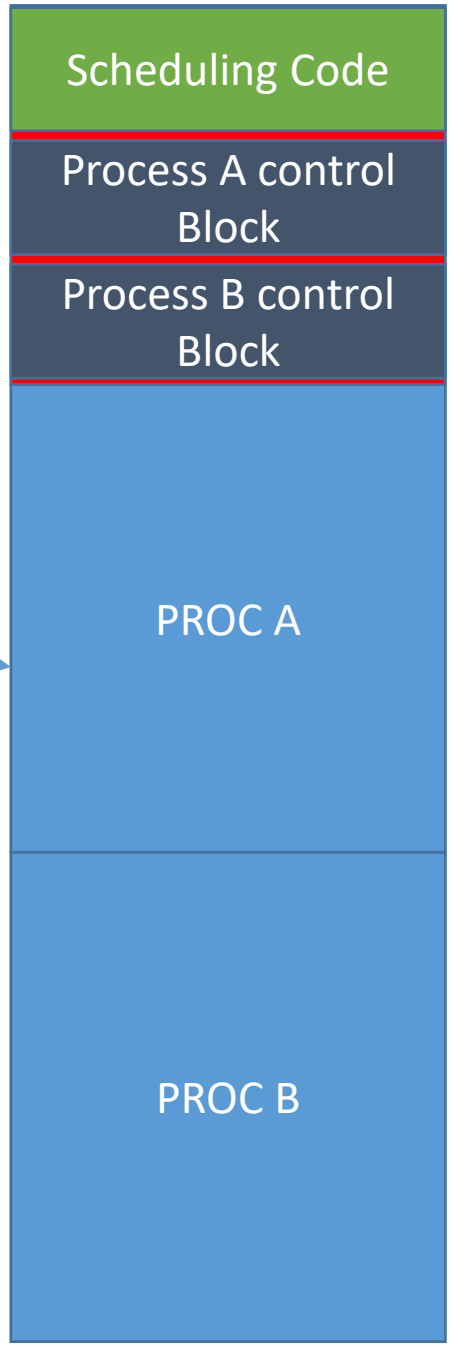


TIMER

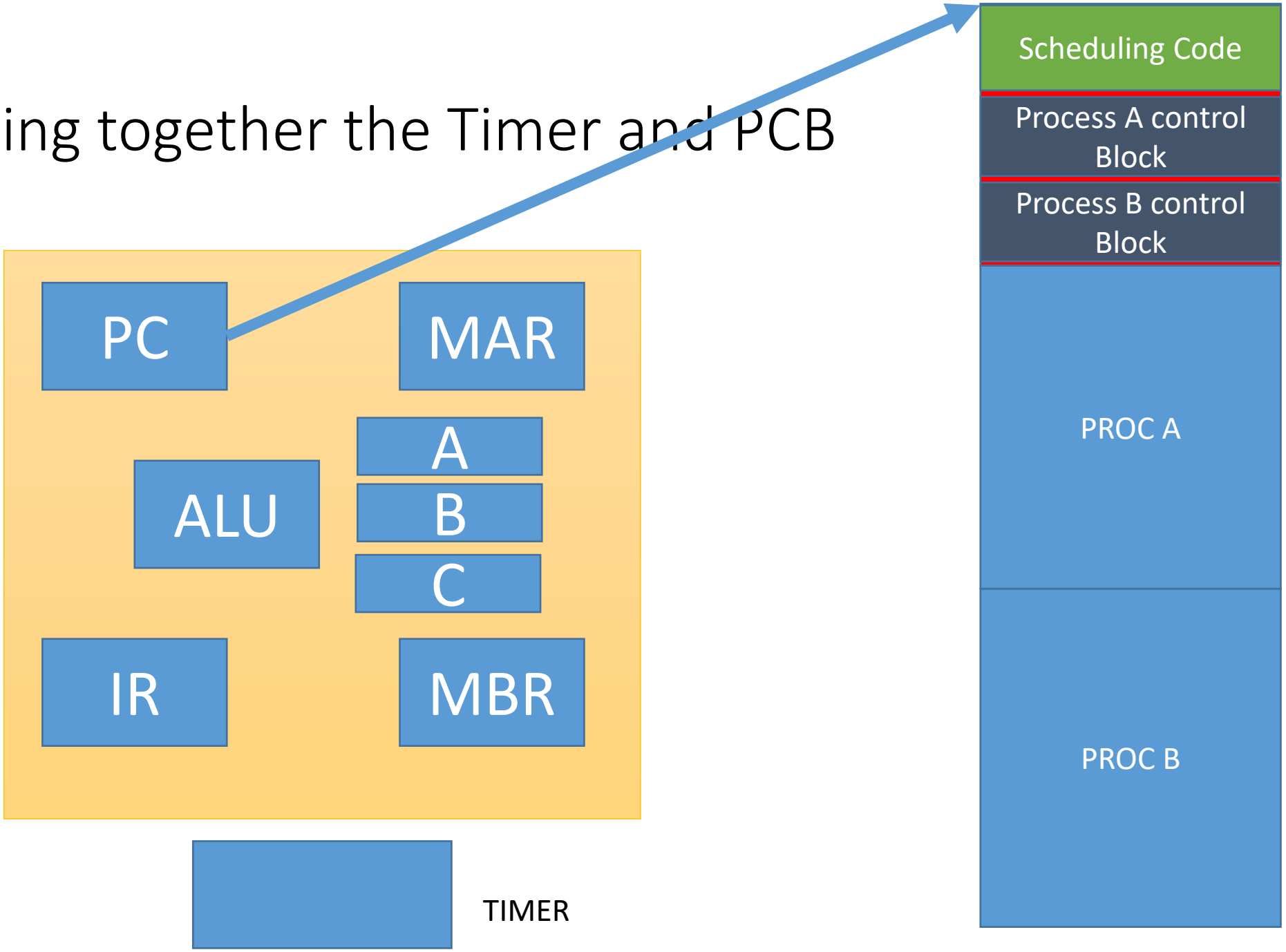
Hence all the **required info for rerunning** the PROC A from same place (PC, register values) are stored in the **process control block**

The execution **time** for PROC A is **up**

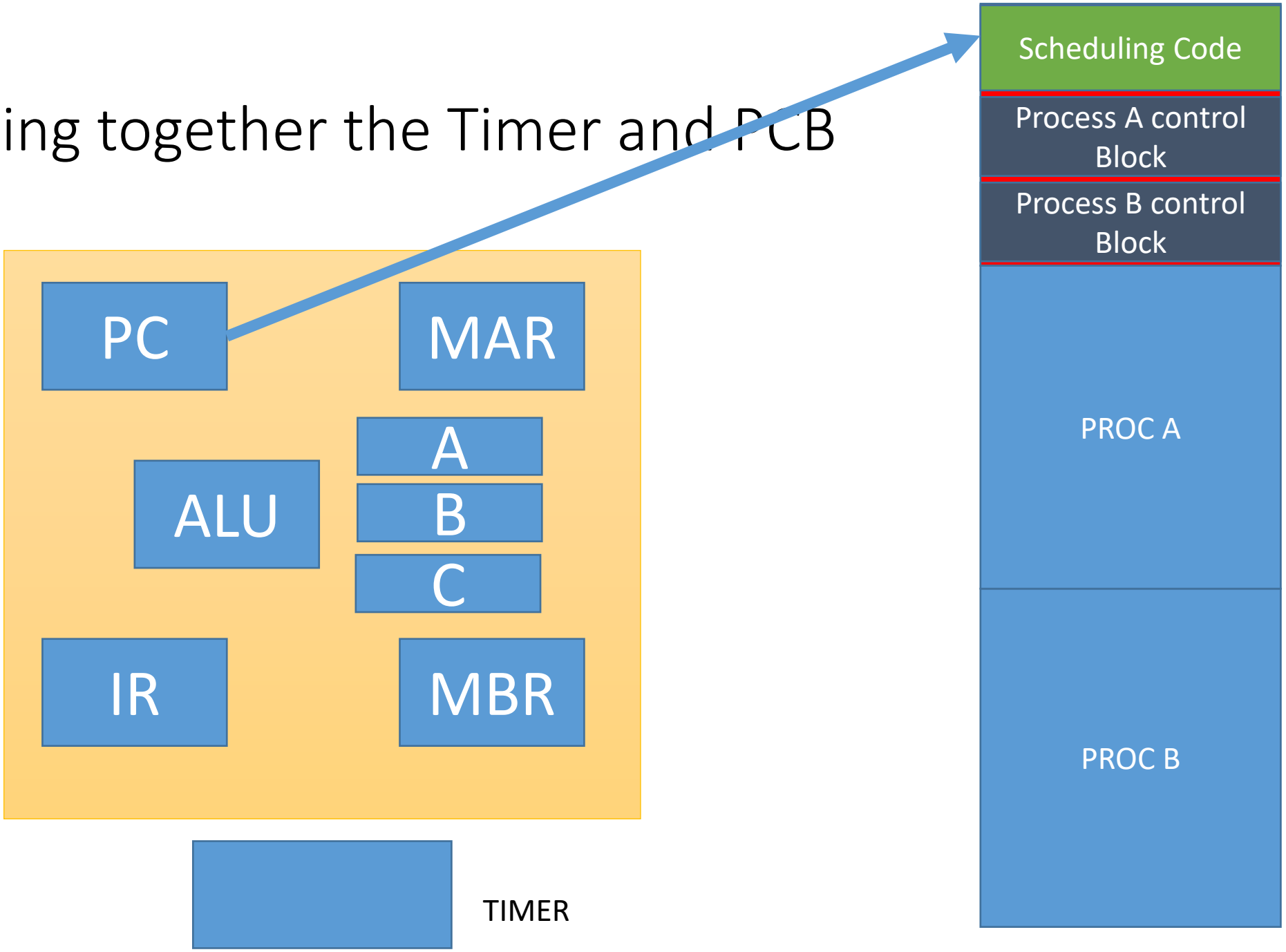
However it will be **brought in later on** if the OS decides to



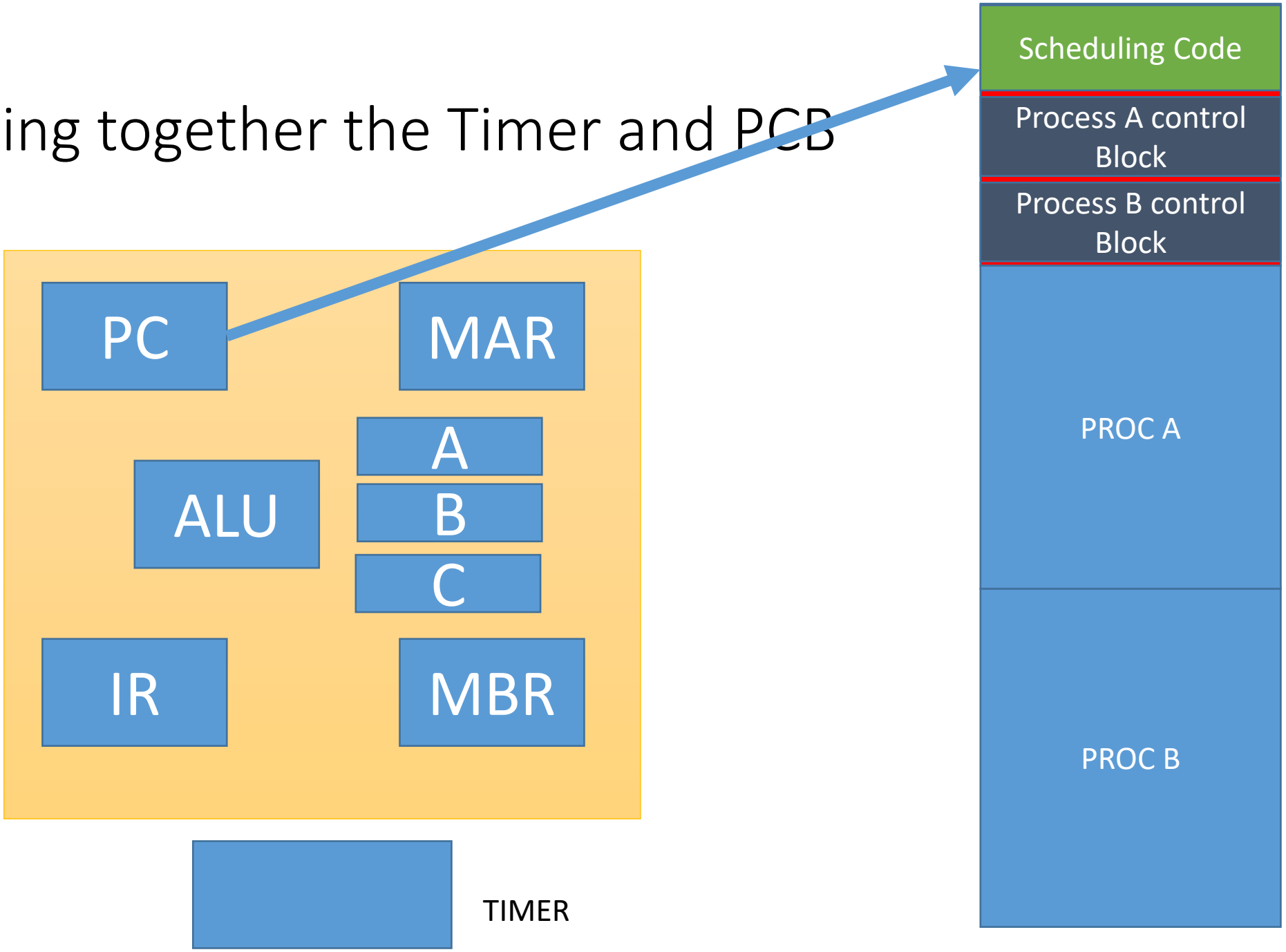
Putting together the Timer and PCB



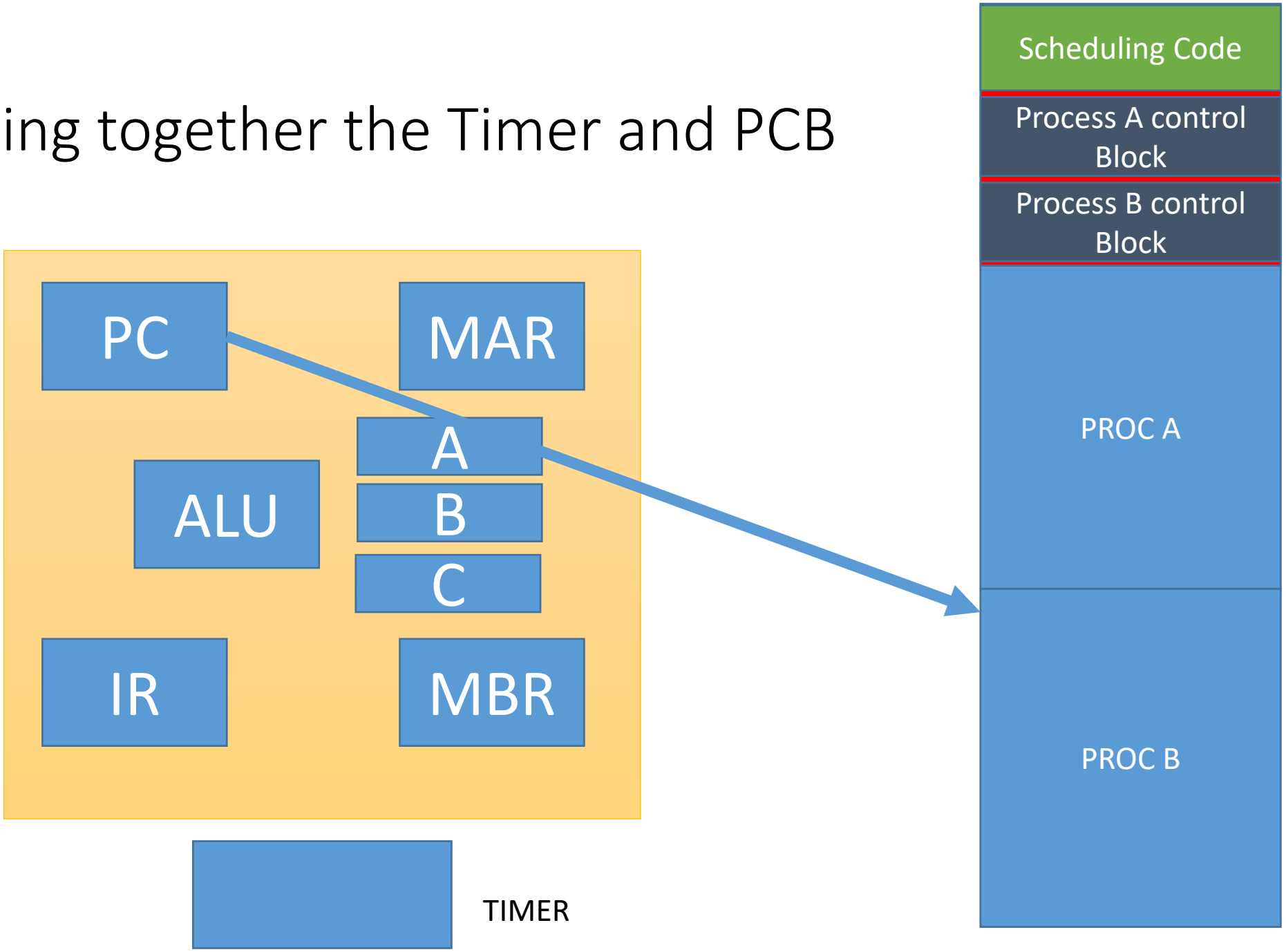
Putting together the Timer and PCB



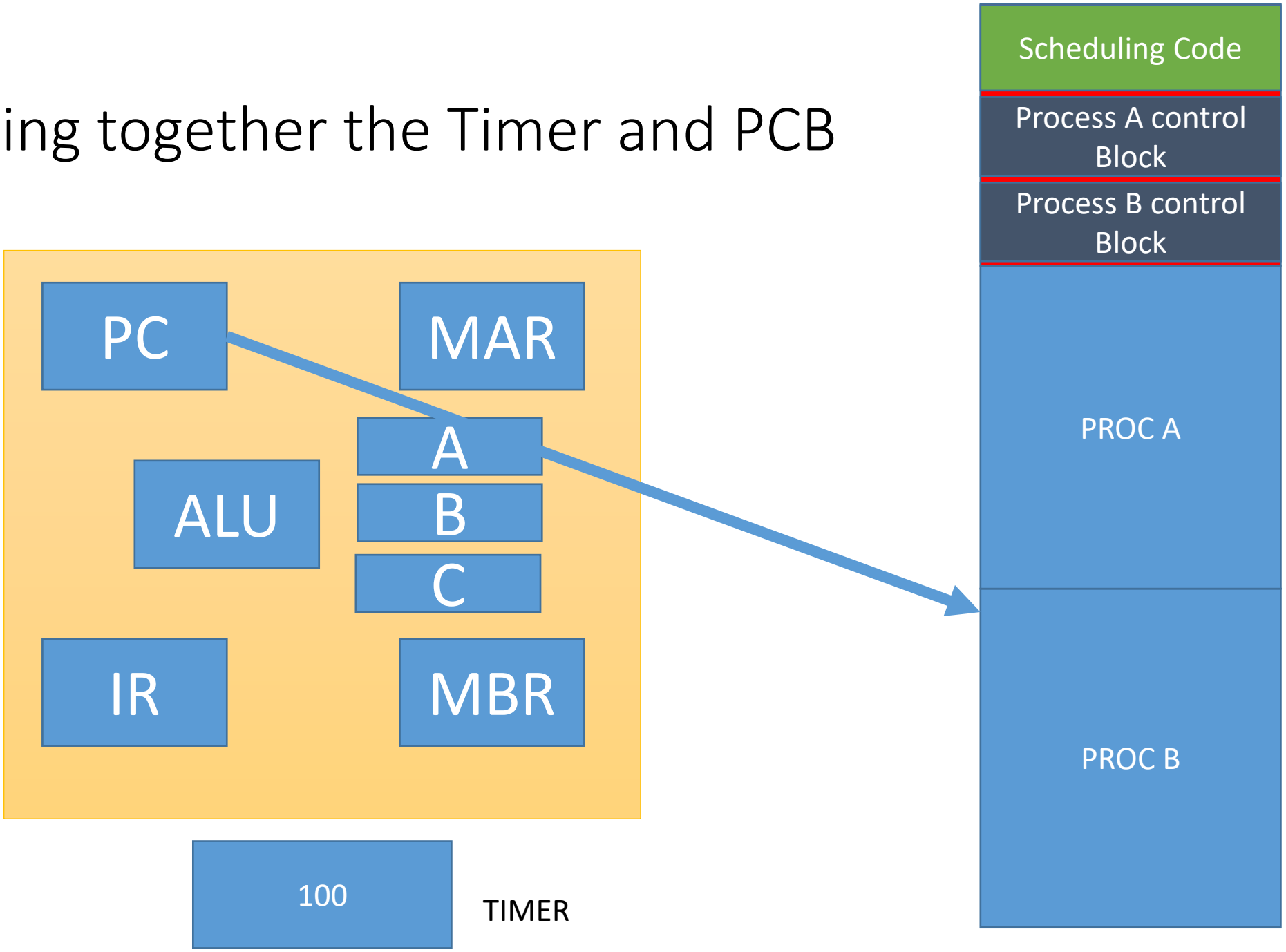
Putting together the Timer and PCB



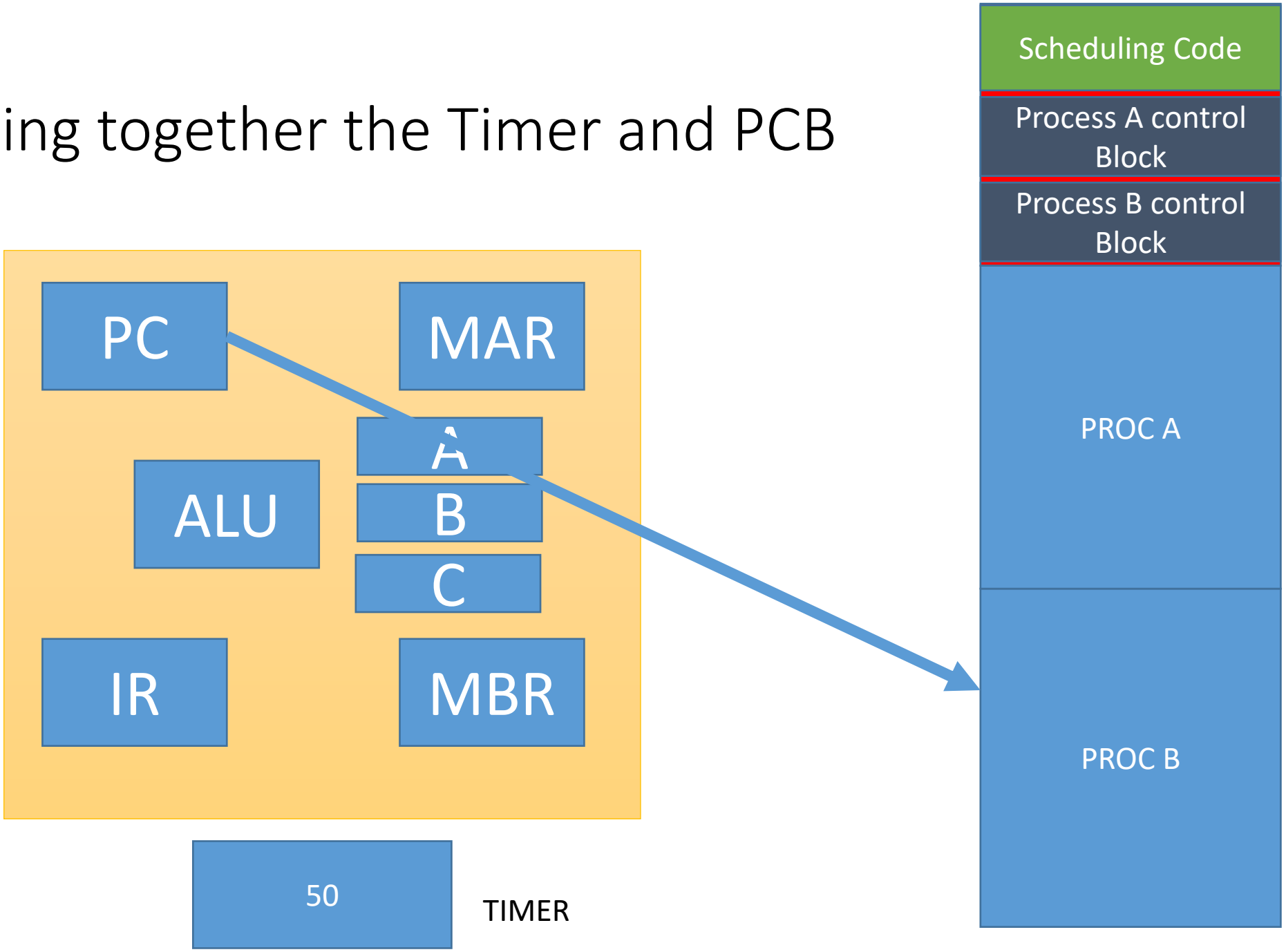
Putting together the Timer and PCB



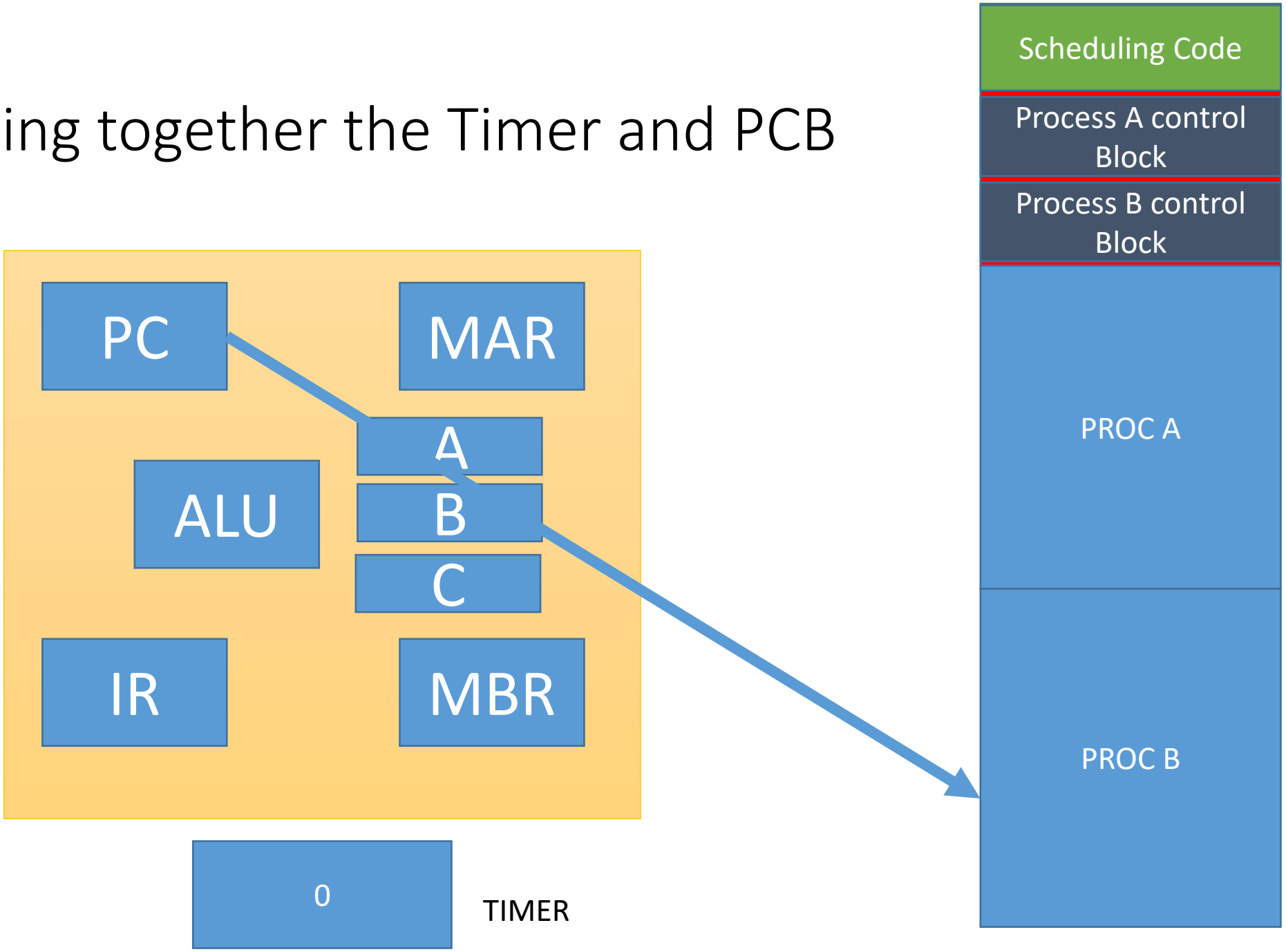
Putting together the Timer and PCB



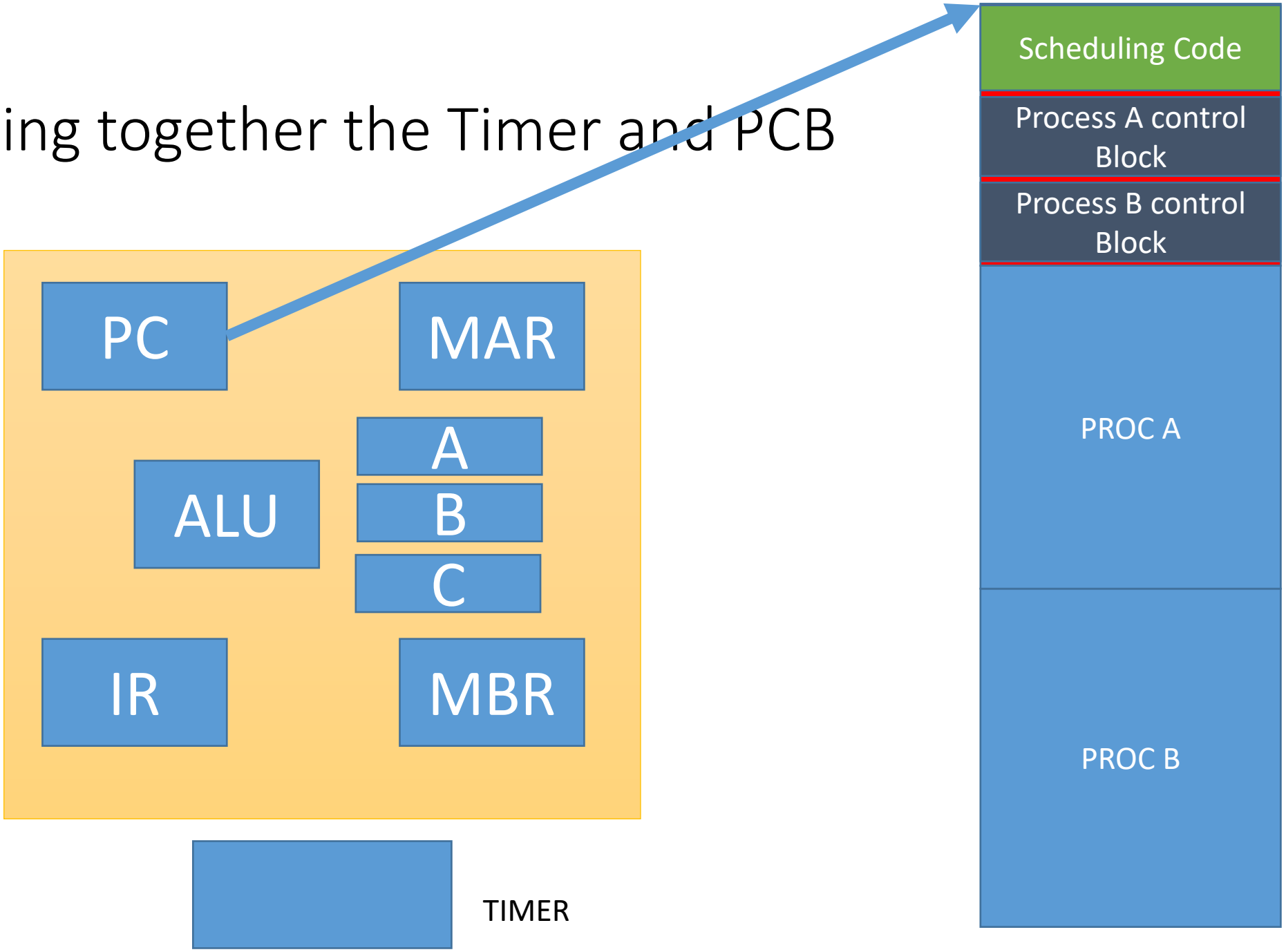
Putting together the Timer and PCB



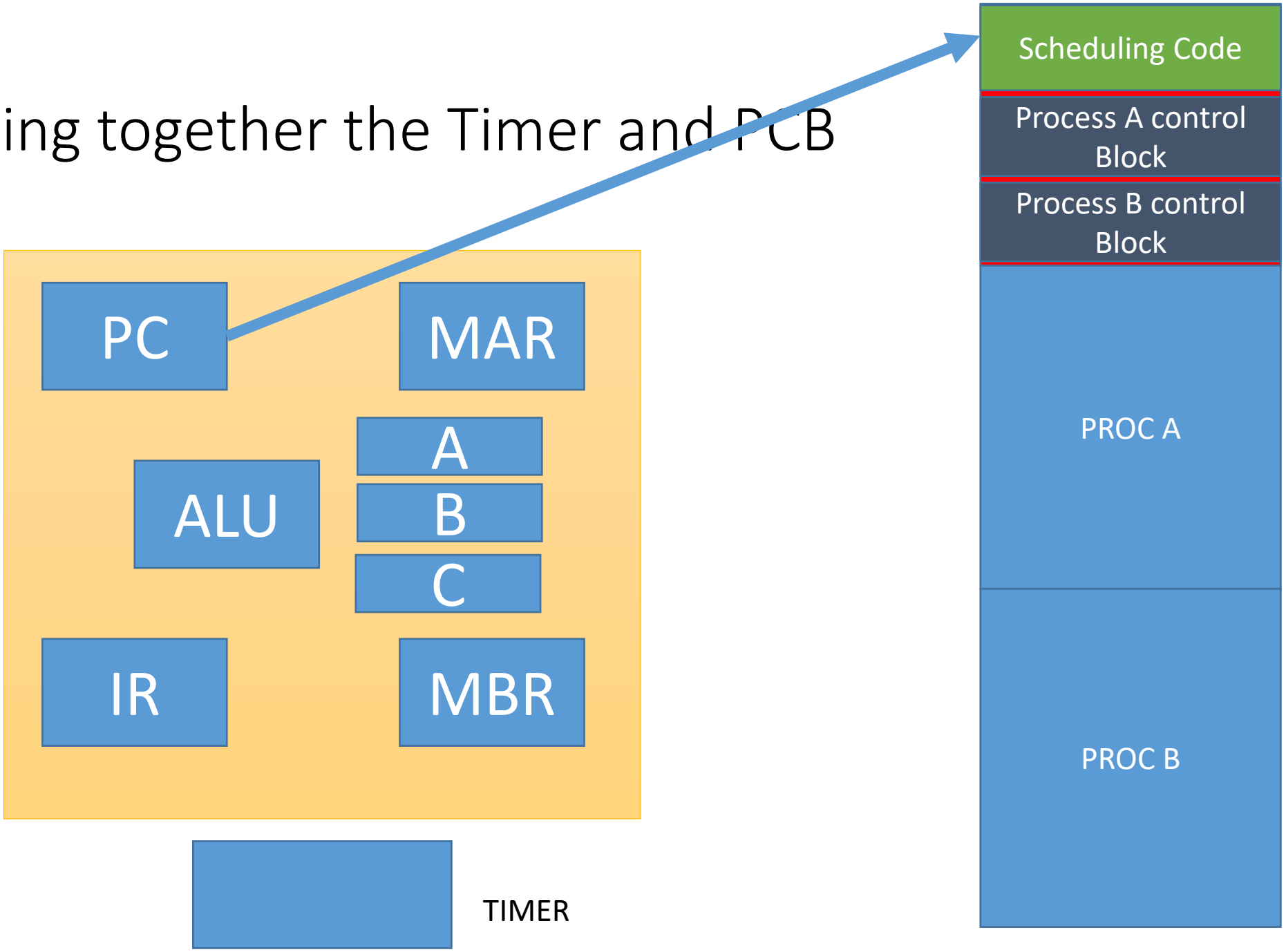
Putting together the Timer and PCB



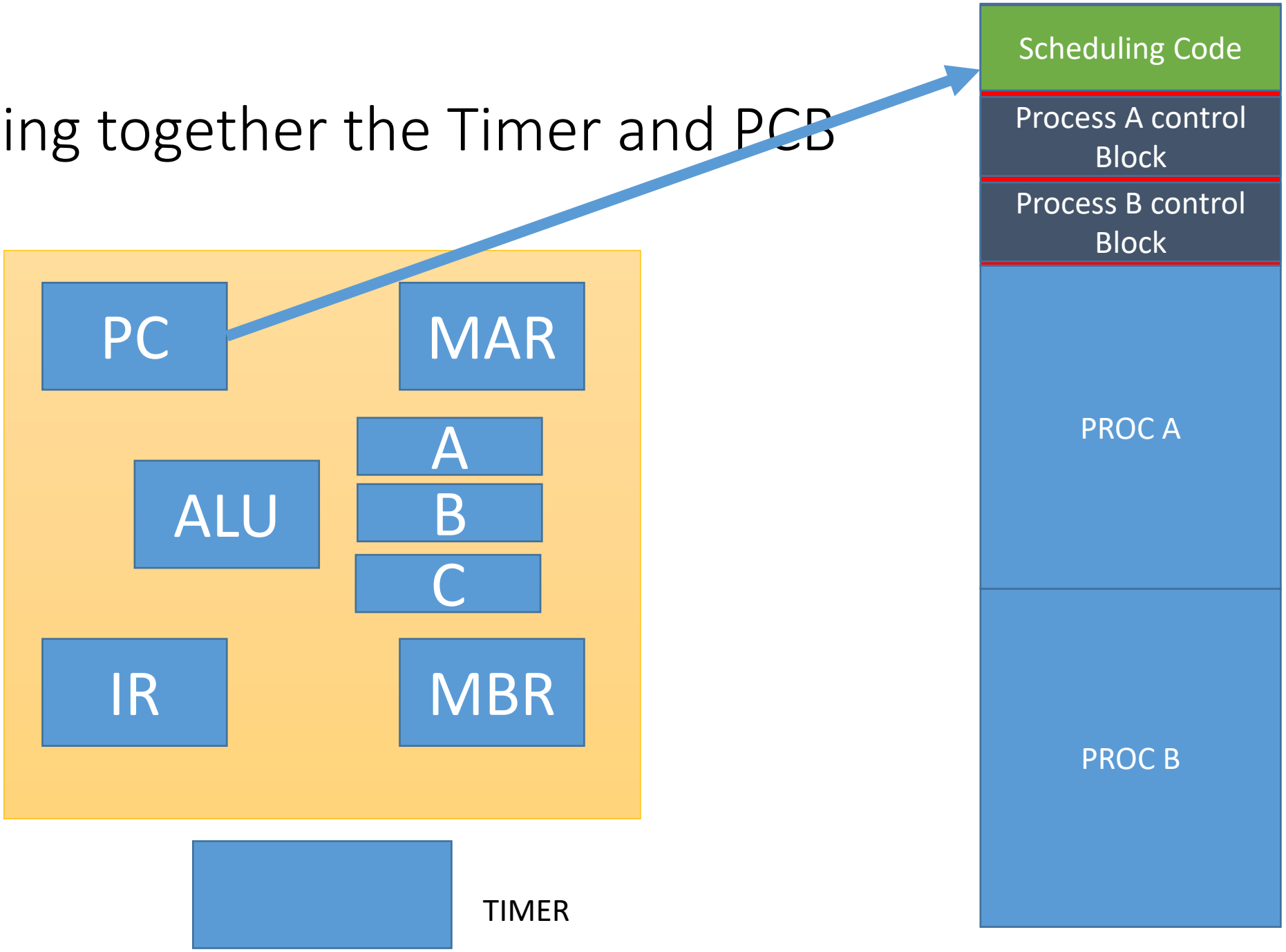
Putting together the Timer and PCB



Putting together the Timer and PCB



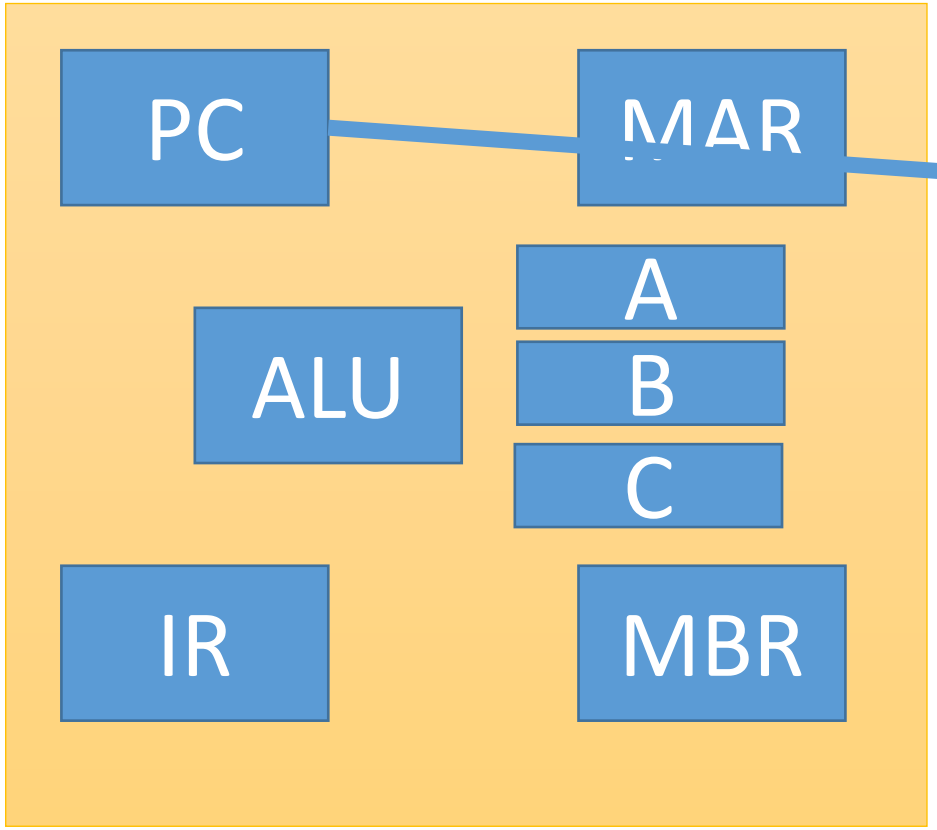
Putting together the Timer and PCB



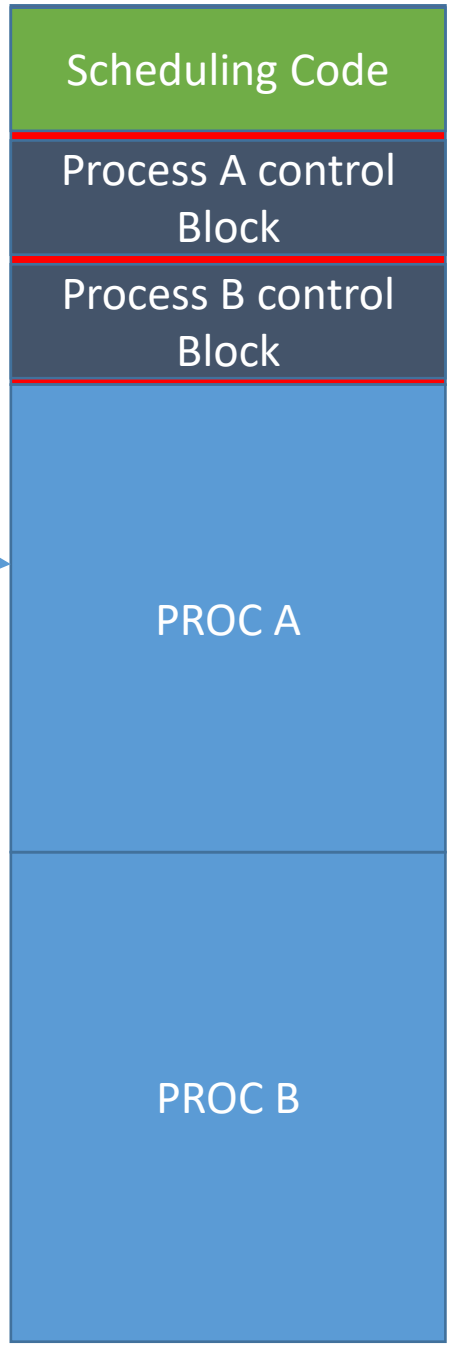
Putting together the Timer and PCB

The **Proc** A Has been **rescheduled** to run **again**

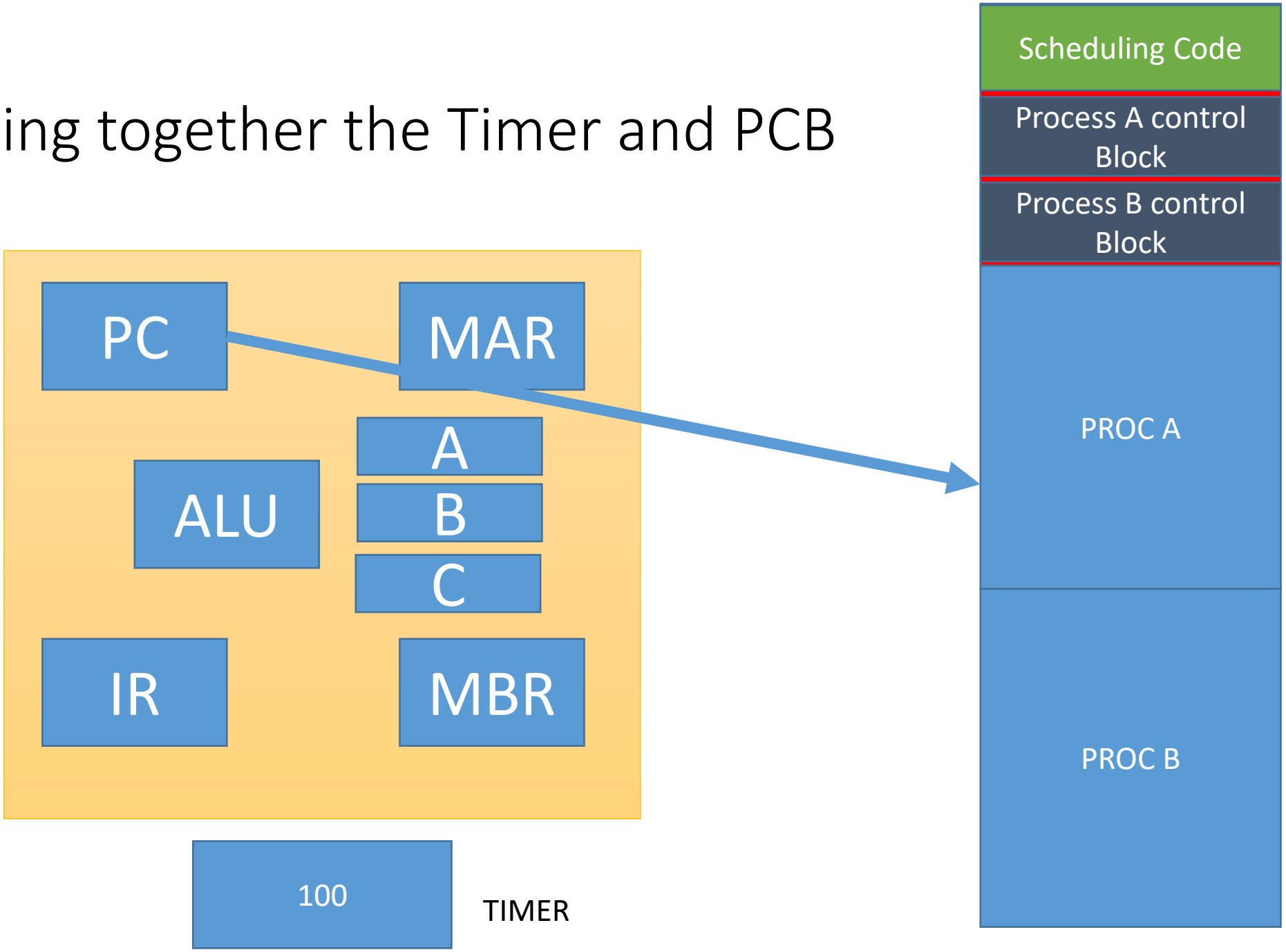
Since all its information , Were **preserved**, it is starting From the **same place** where it was before



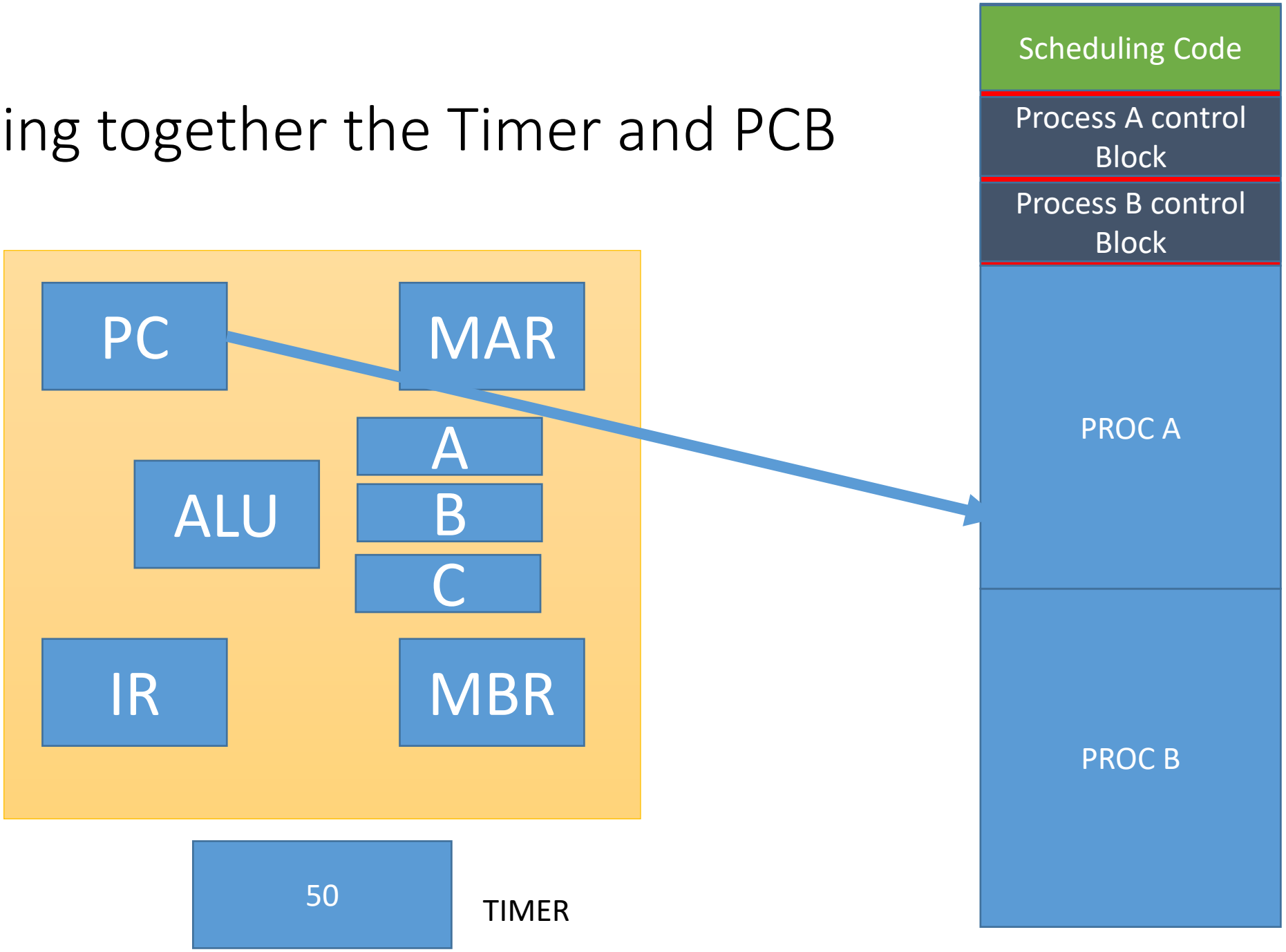
TIMER



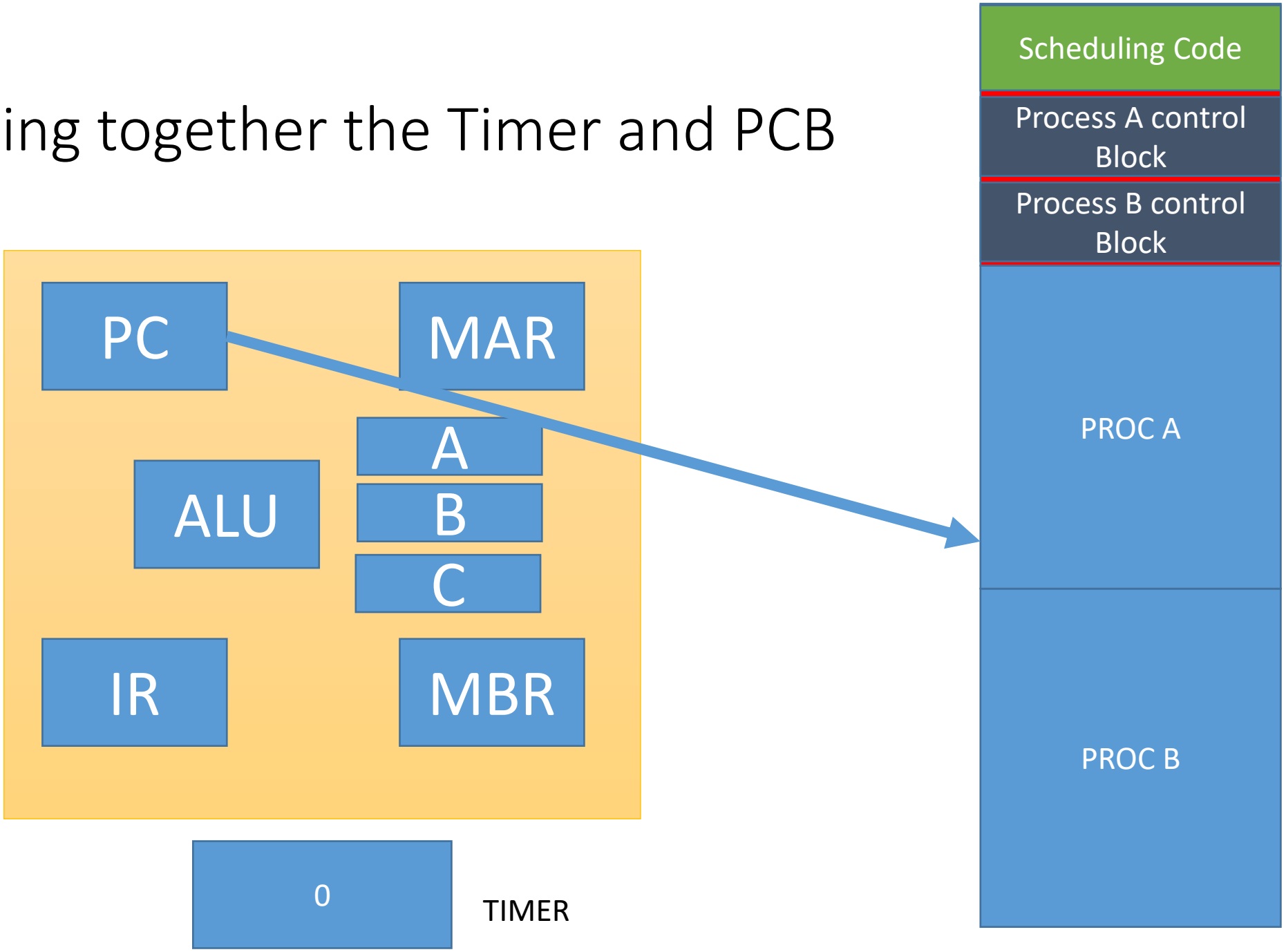
Putting together the Timer and PCB



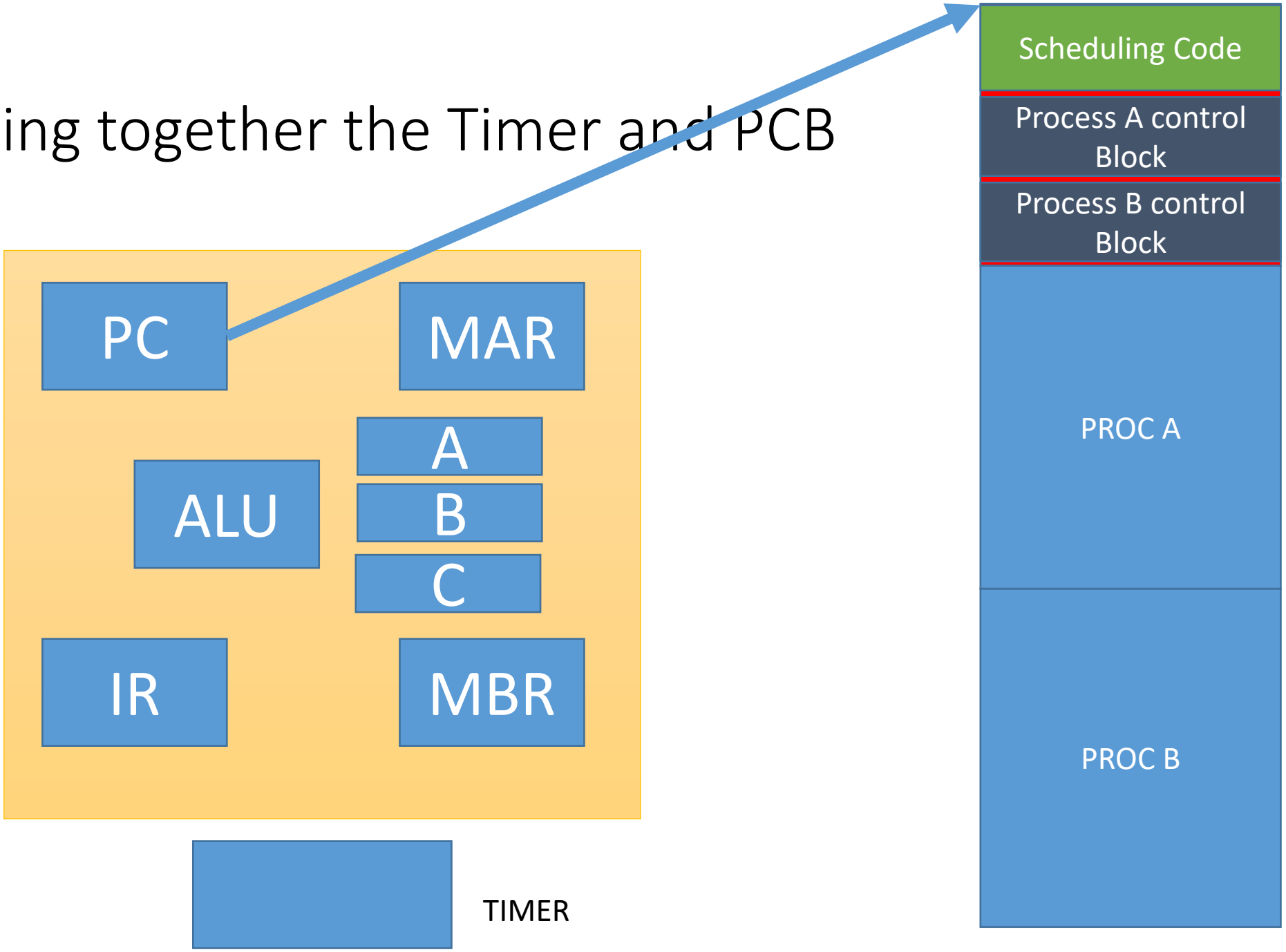
Putting together the Timer and PCB



Putting together the Timer and PCB



Putting together the Timer and PCB



2.2.Different States of a process

Different States of a Process

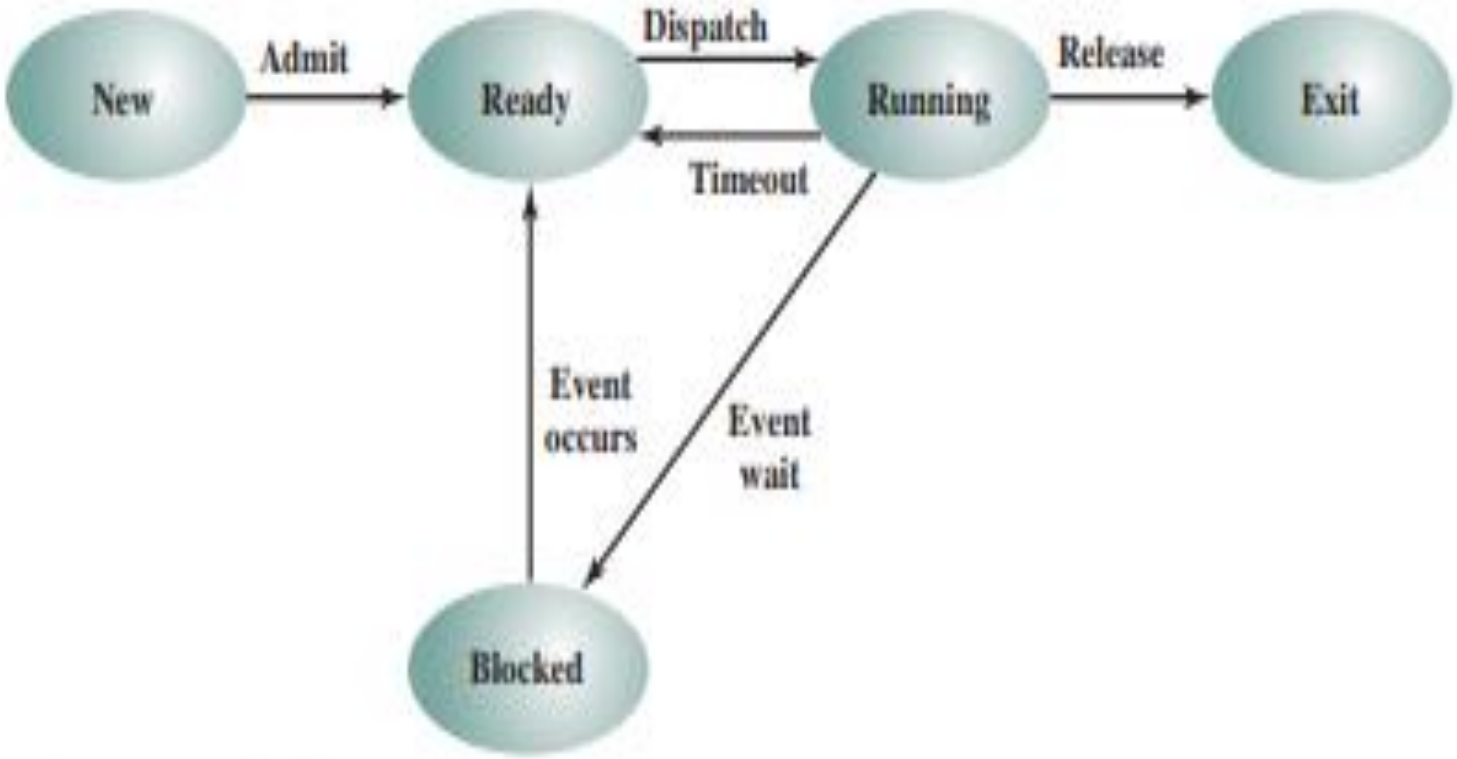


Figure 8.7 Five-State Process Model

New- program clicked to be initiated.
Ready-The process's code is in the memory along with meta data in PCB
Running-The process is being executed in processor
Blocked- Not running, but it may run later on, its necessary information are stored before switching to another processor.
Exit-The process has terminated, soon its code and meta data will be evicted.

Putting the bits and bytes of scheduling together

The data structure maintained by OS:

OS maintains some queues.

Long term queue - List of programs that are **waiting to enter the system**. (say the user has clicked on 4 different programs)

This queue is used by the **high level scheduler.** Based on this queue, it will allocate memory for process codes and hence bring them into the **ready** state.

Short term queue - Consists of all the processes in the **ready state**.

The short term scheduler will pick which process to run next.

Intermediate queue: A blocked process which was running in the memory, but blocked due to some operations and is no longer in the memory (we will see this in detail **later**)

I/O Queue - Multiple processes may **request** for an **input/output device** at a time. Hence these requests are inserts into a queue. From which later on the access is given.

Different States of a Process

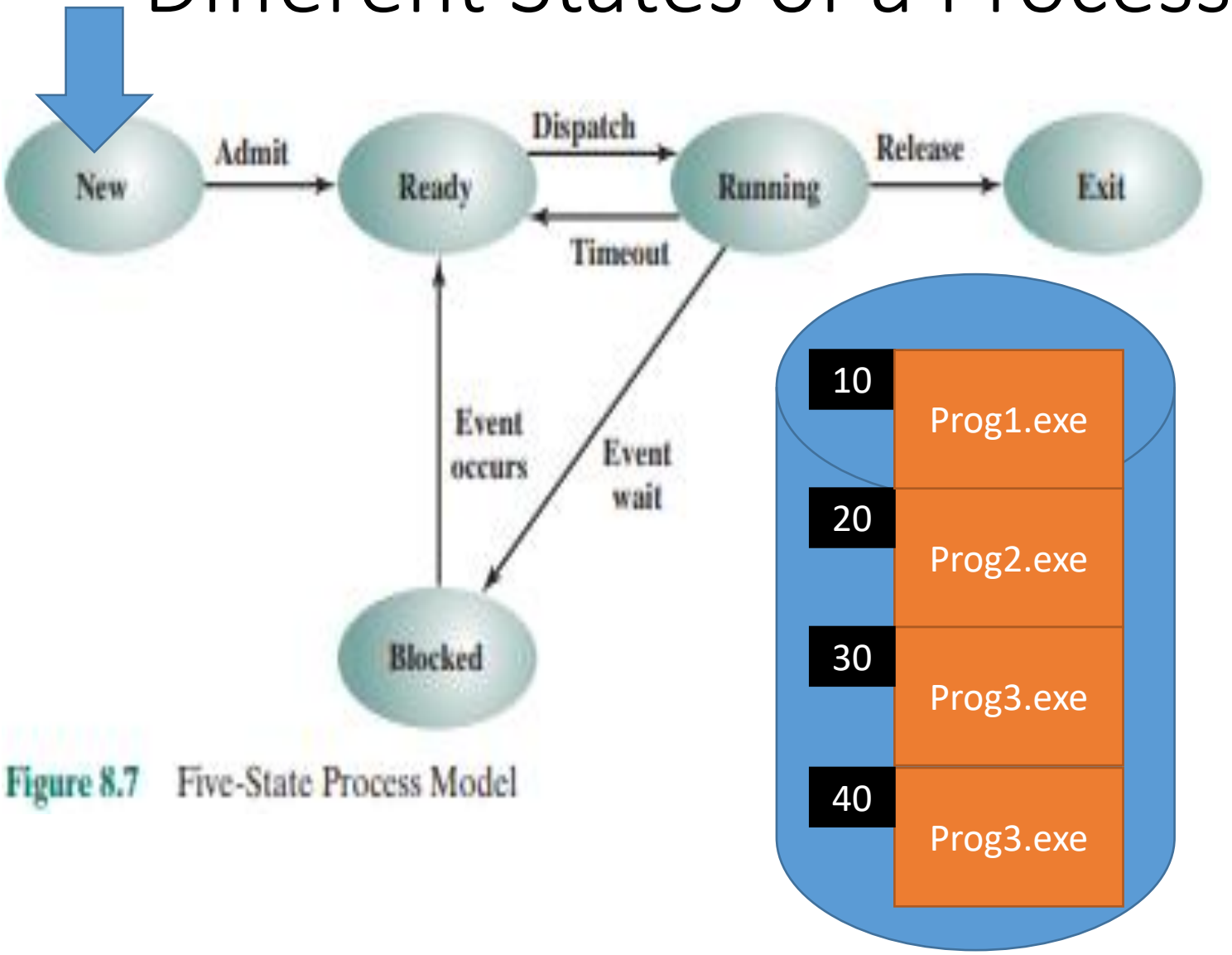
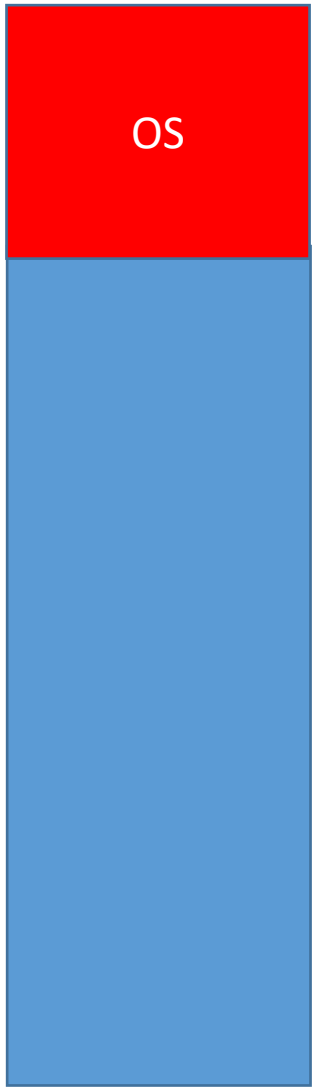


Figure 8.7 Five-State Process Model

Suppose the user wants to run **prog1.exe**



Different States of a Process

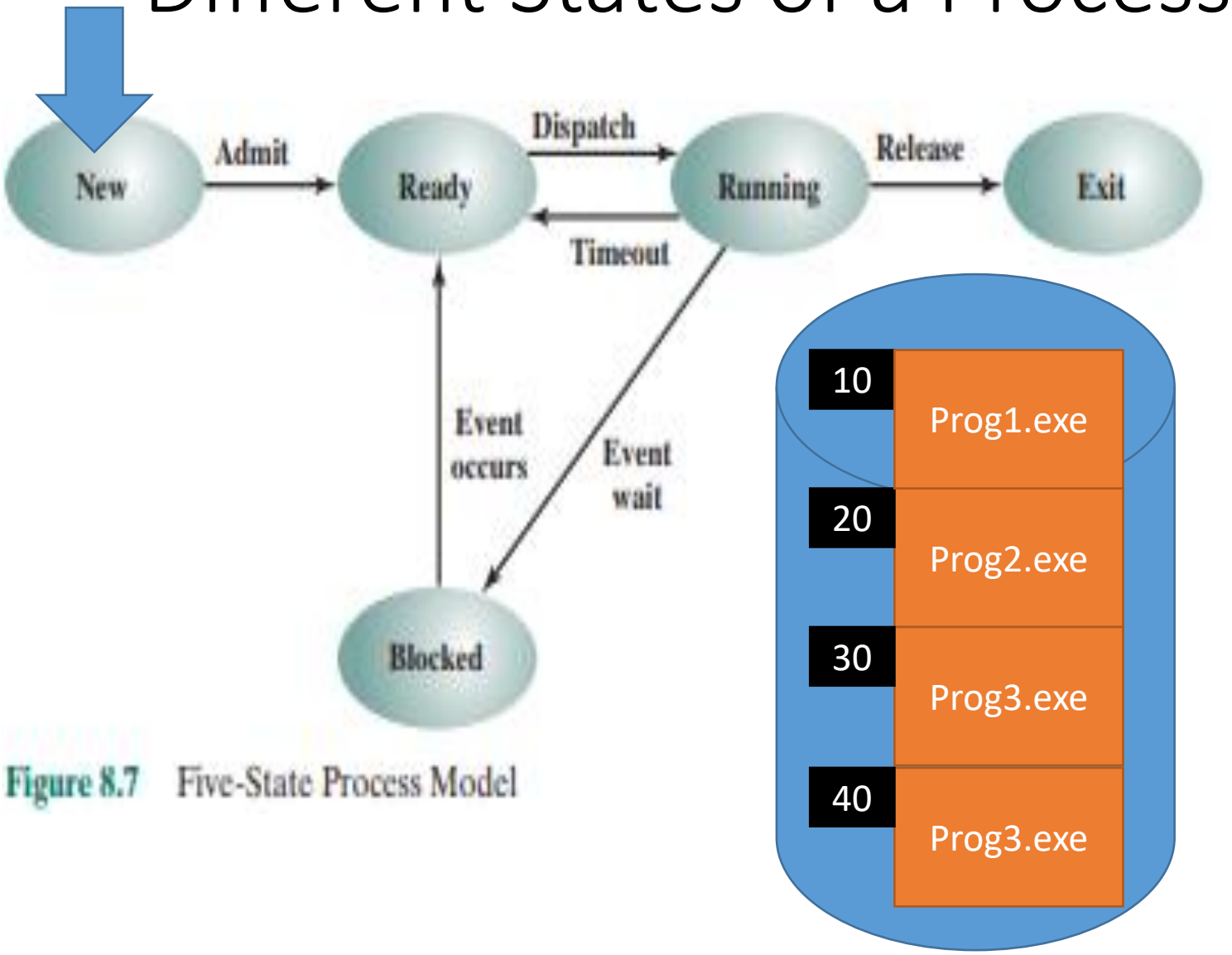
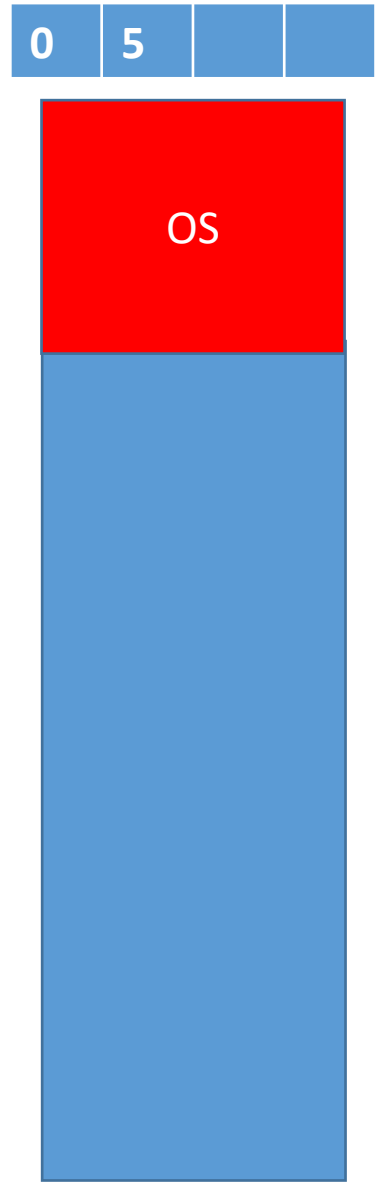


Figure 8.7 Five-State Process Model

Its address will be provided to the **long term queue** of the OS.



Different States of a Process

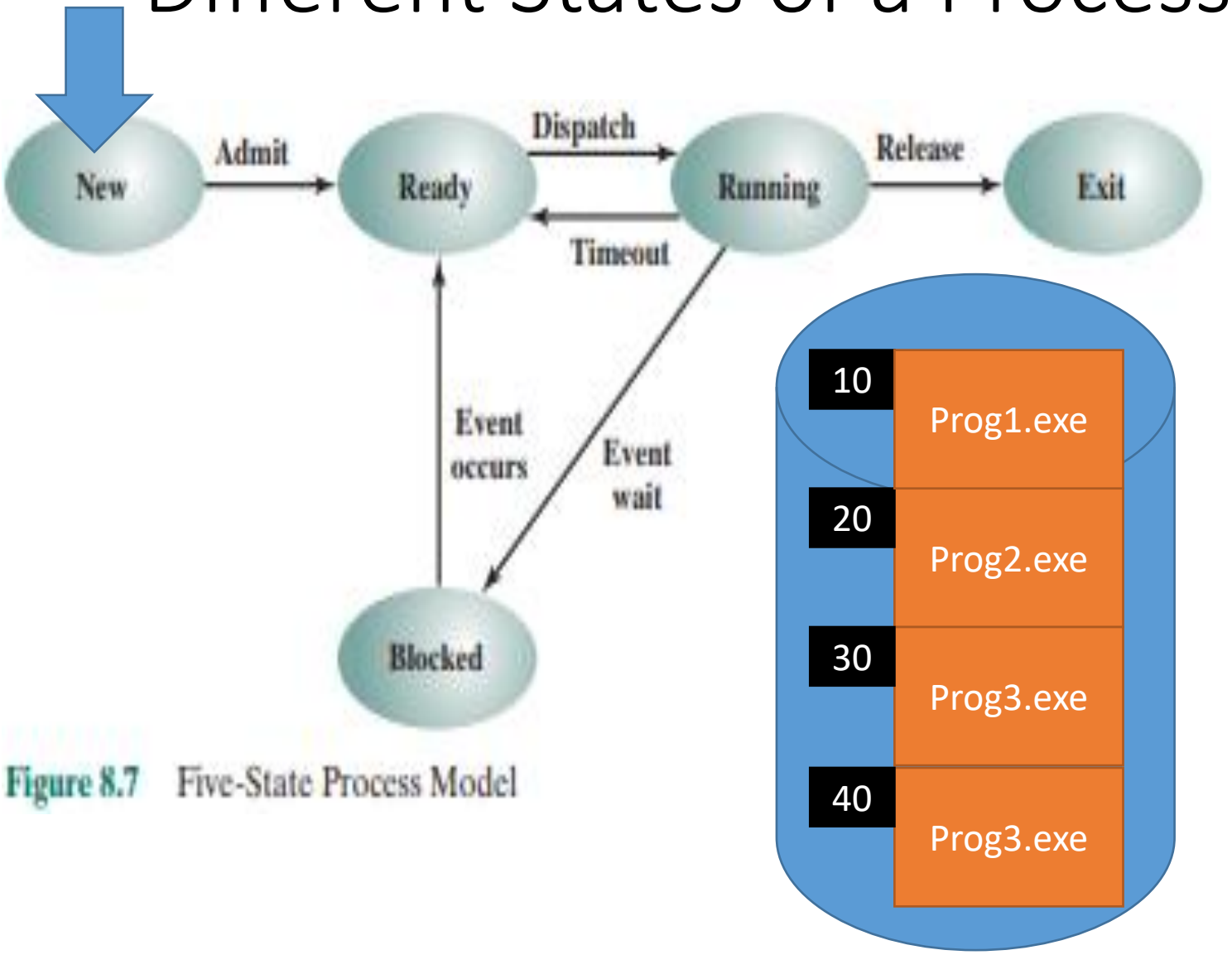
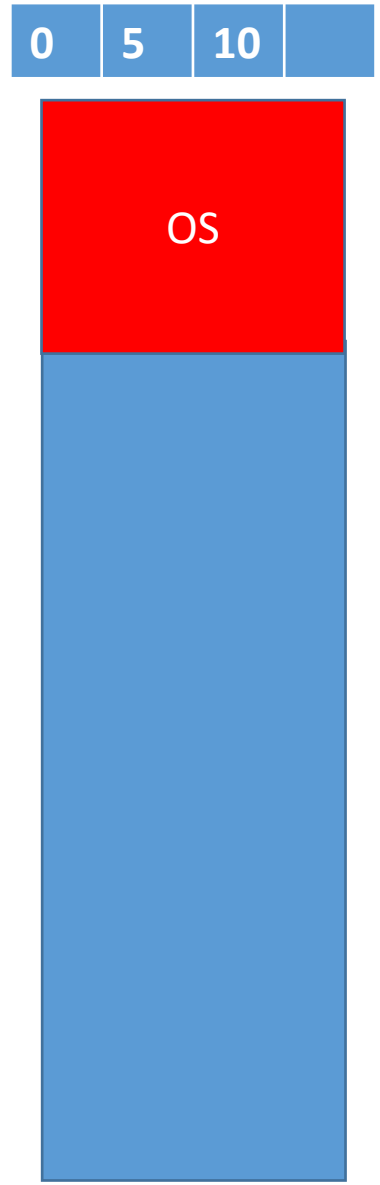


Figure 8.7 Five-State Process Model

Its address will be provided to the **long term queue** of the OS.



Different States of a Process

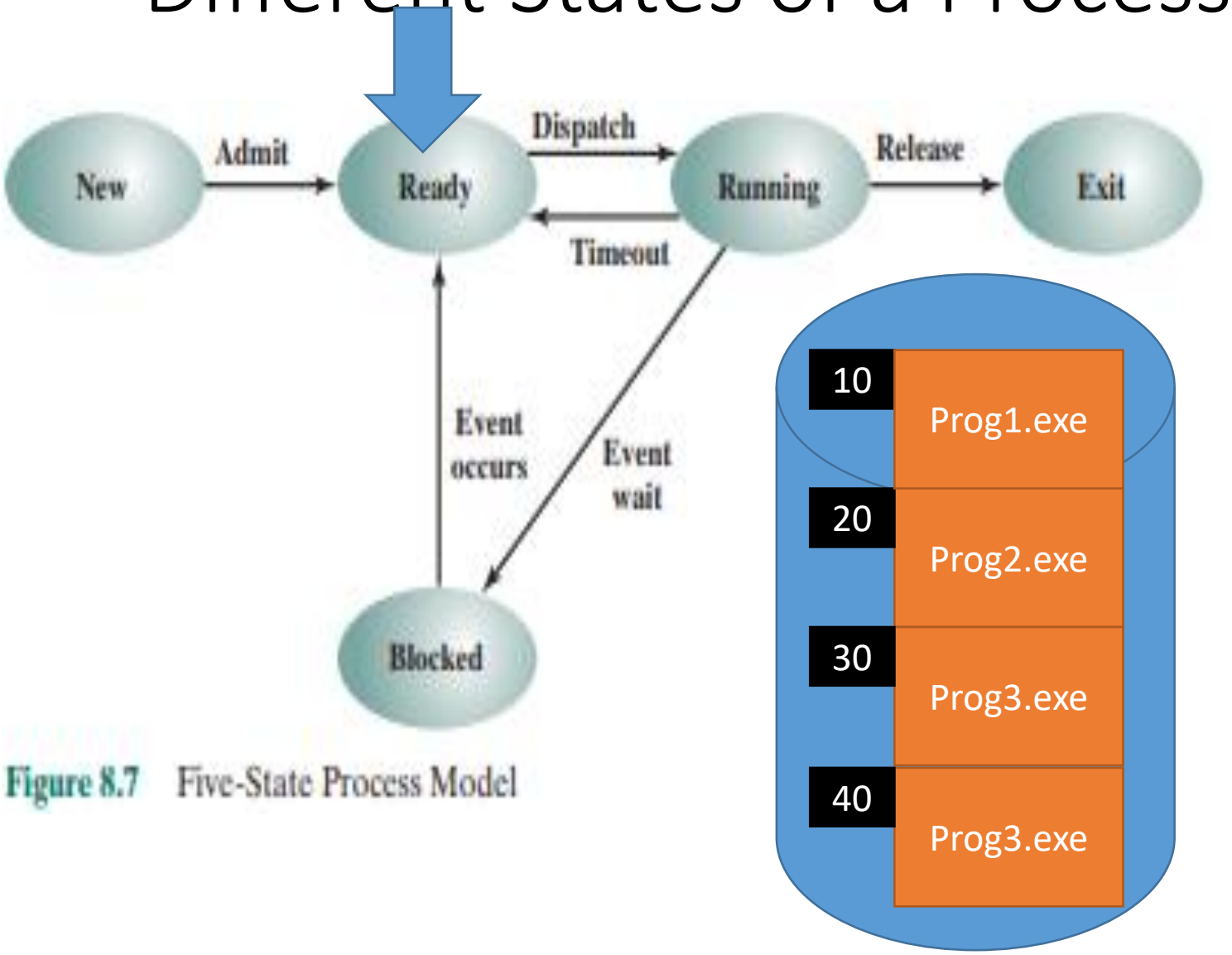
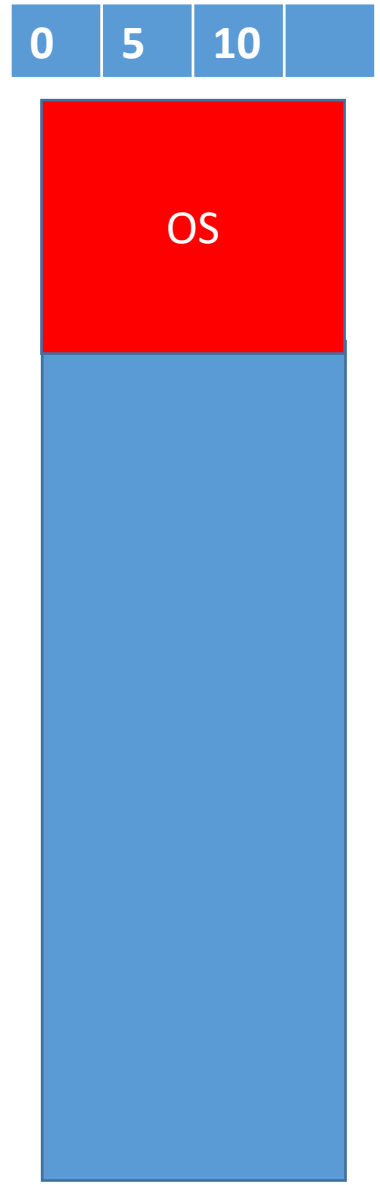


Figure 8.7 Five-State Process Model

Its address will be provided to the **long term queue** of the OS.



Different States of a Process

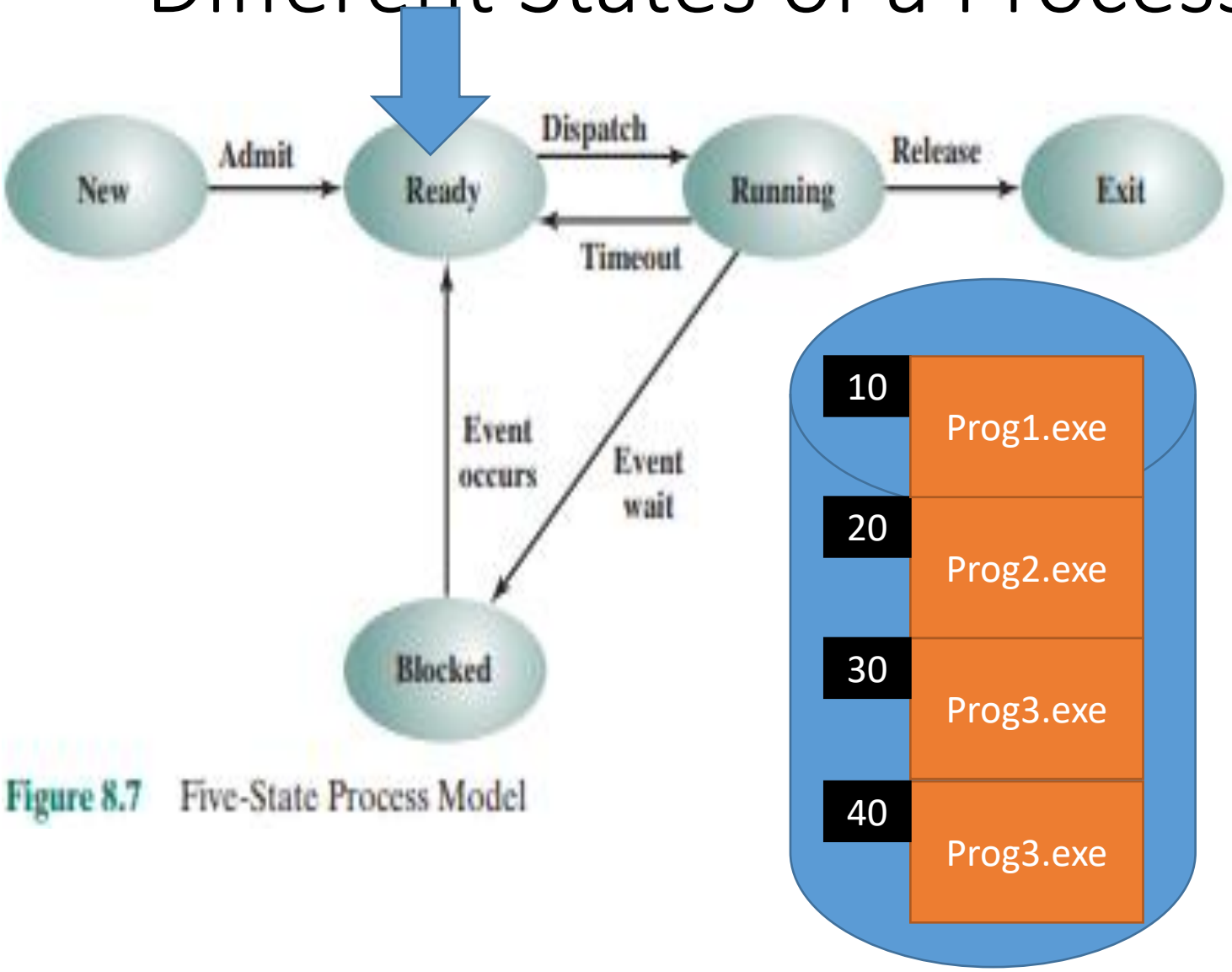
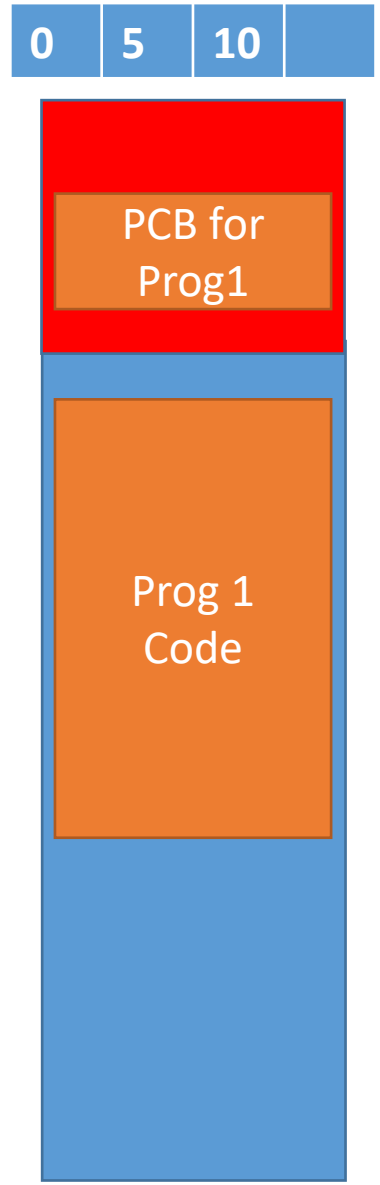


Figure 8.7 Five-State Process Model

Now its code and metadata
Are provided into the RAM



Different States of a Process

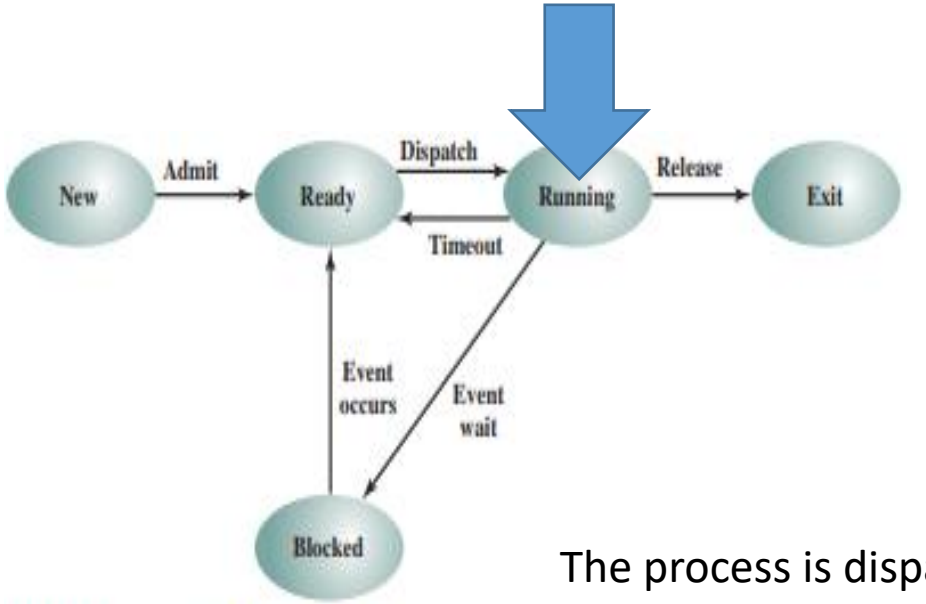
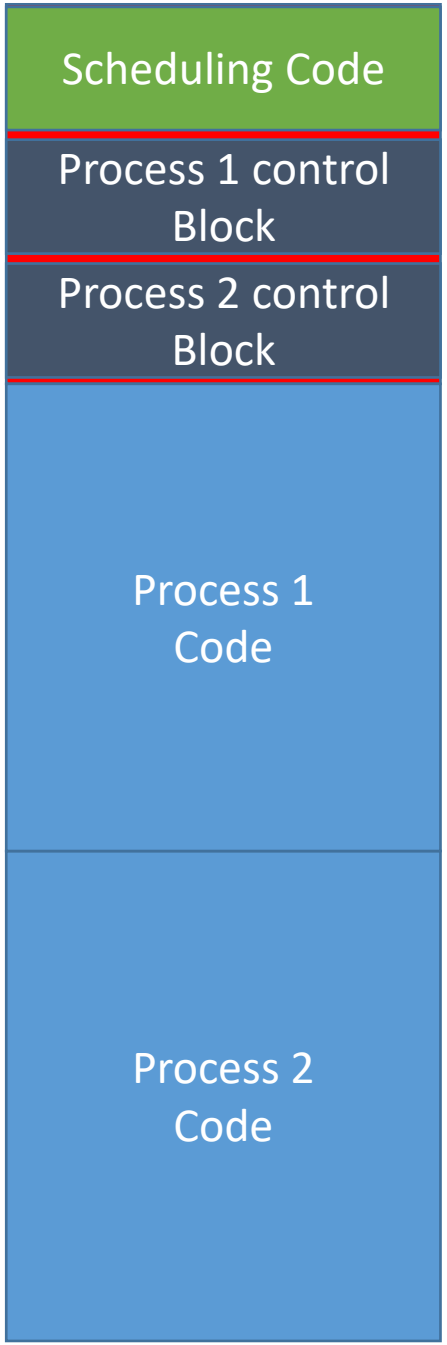


Figure 8.7 Five-State Process Model

The process is dispatched by the scheduler.

i.e the PC begins to take in instruction from it



Different States of a Process

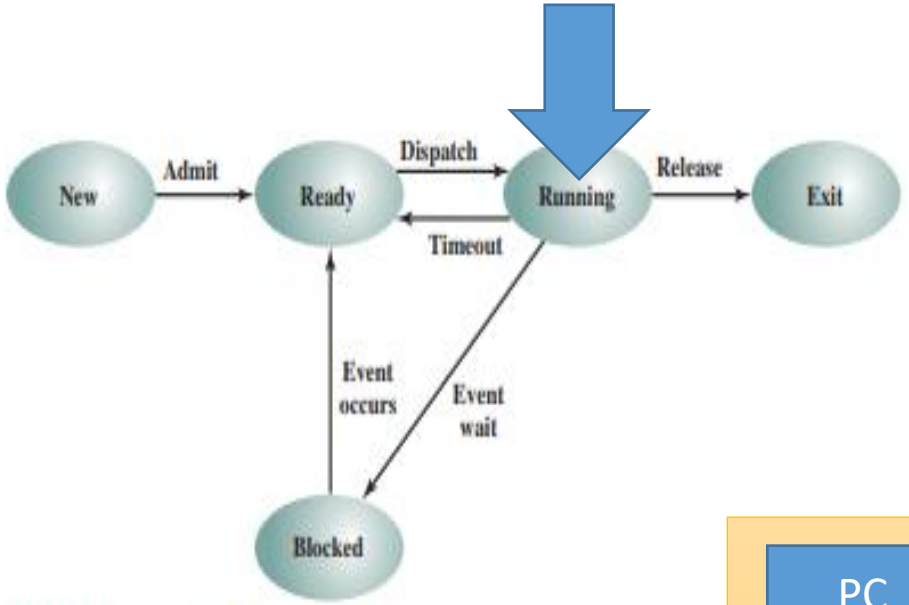
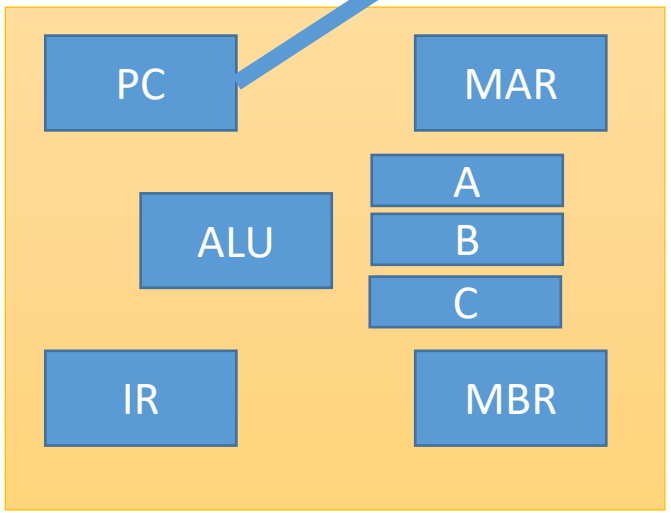
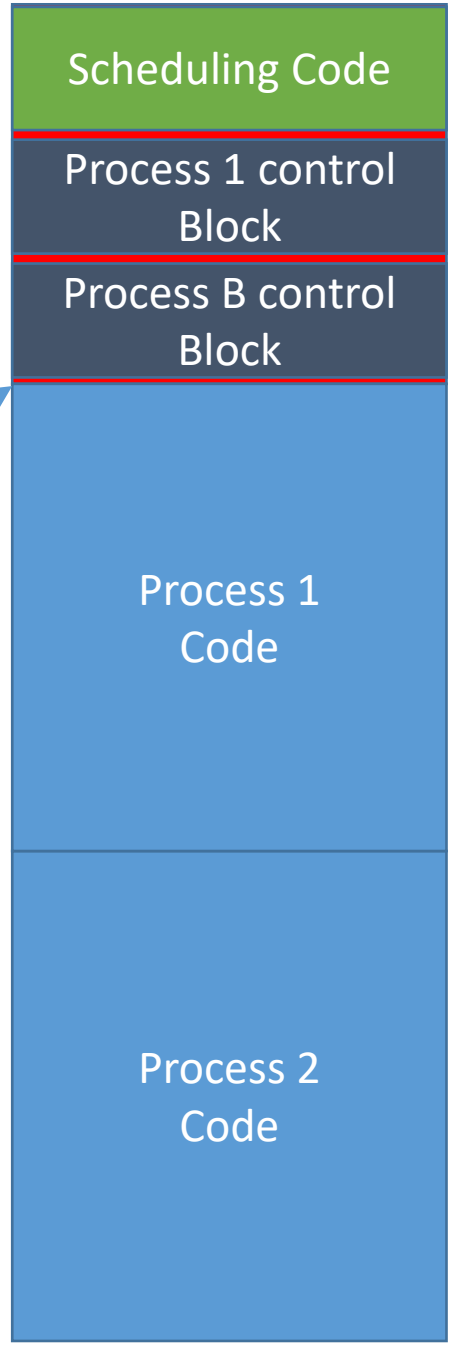


Figure 8.7 Five-State Process Model



TIMER



Different States of a Process

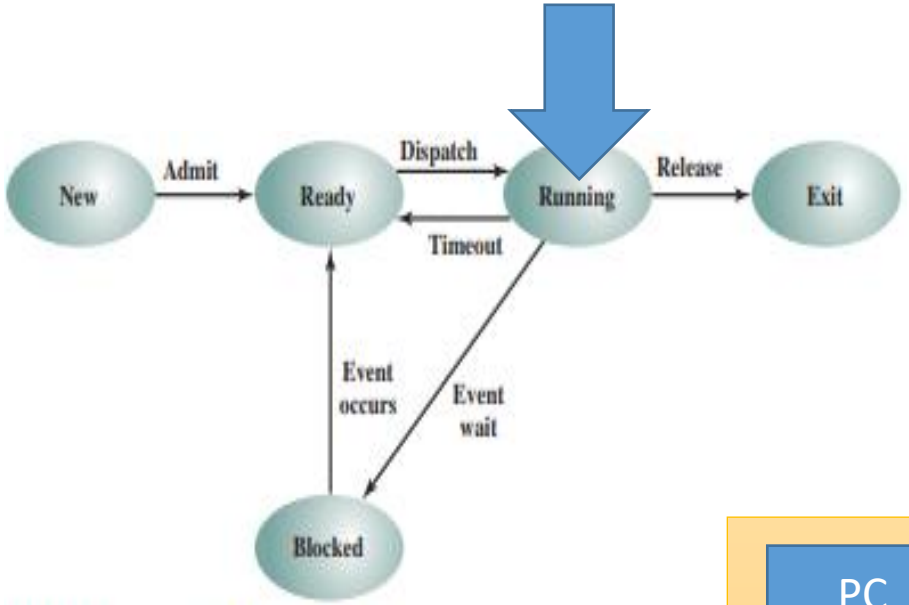
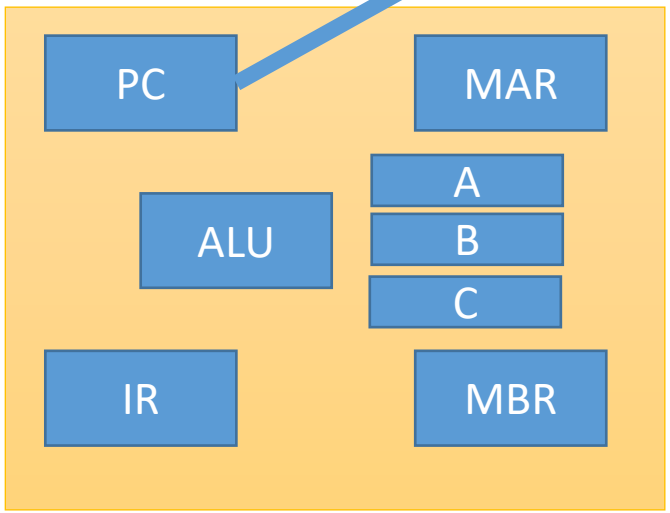
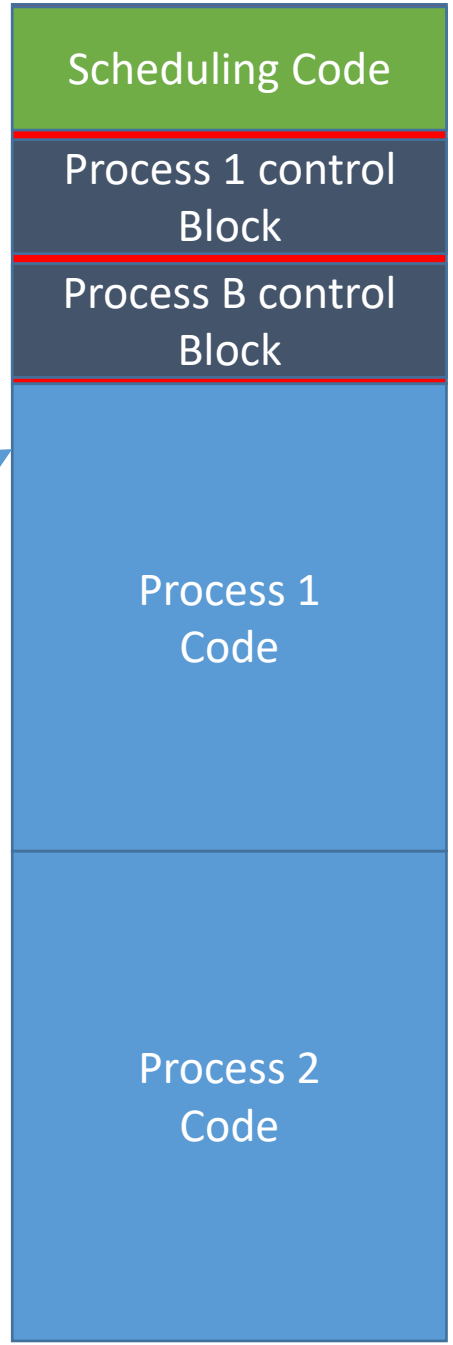


Figure 8.7 Five-State Process Model



TIMER



Different States of a Process

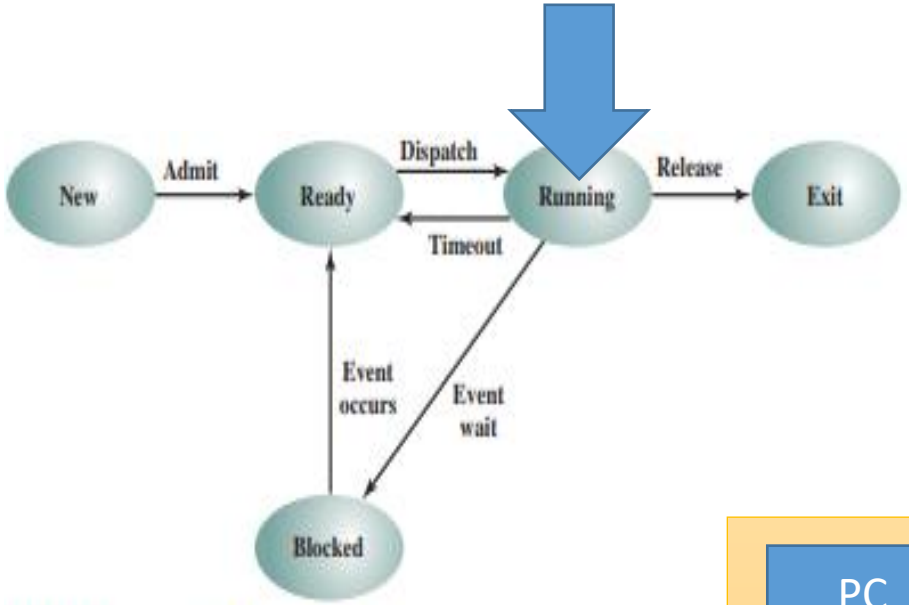
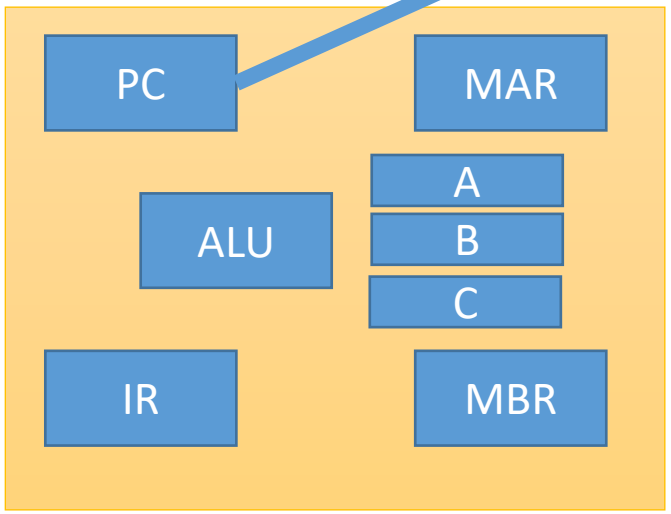
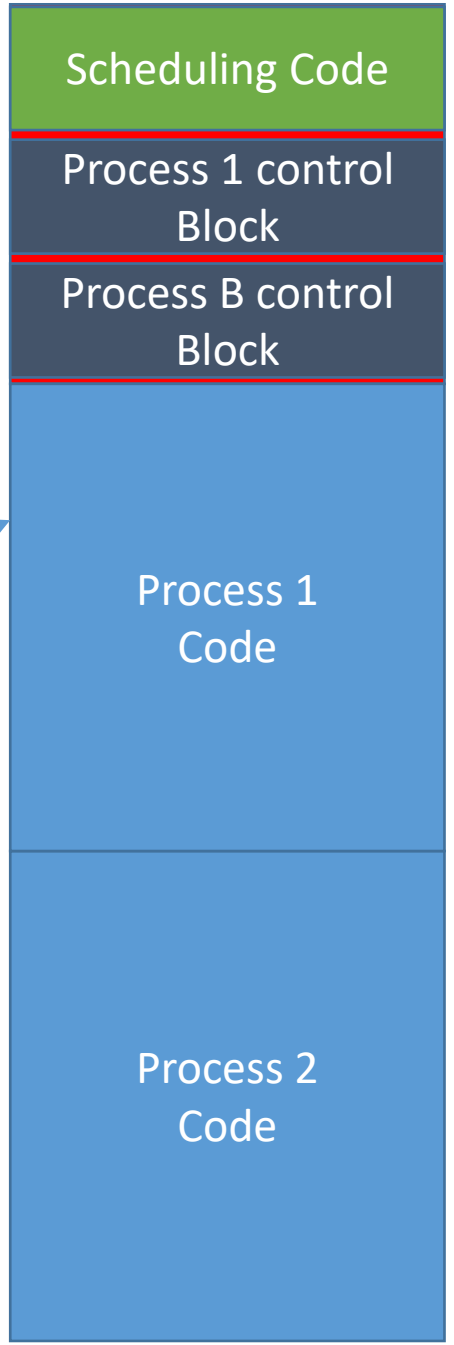


Figure 8.7 Five-State Process Model

Time OUT



TIMER



Different States of a Process

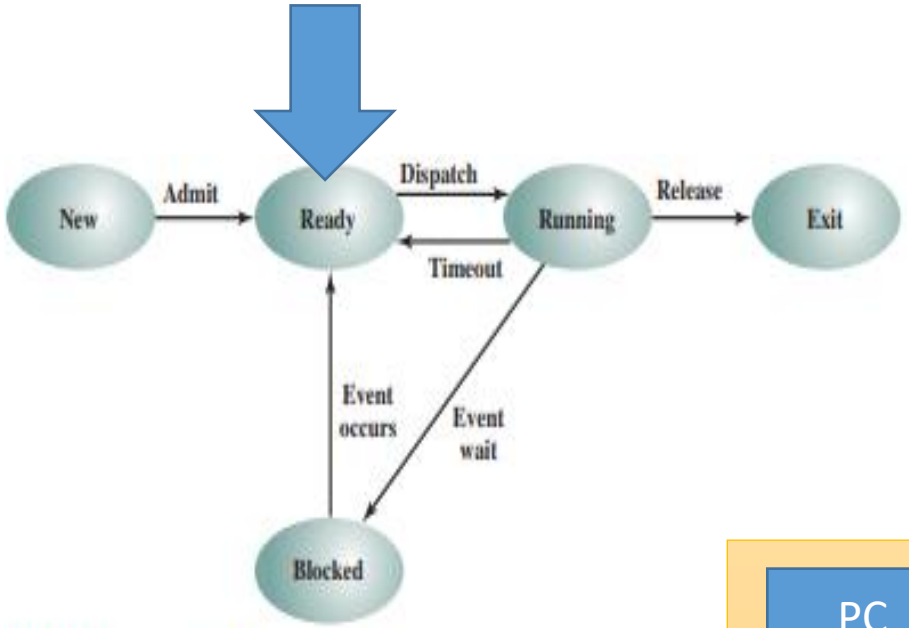
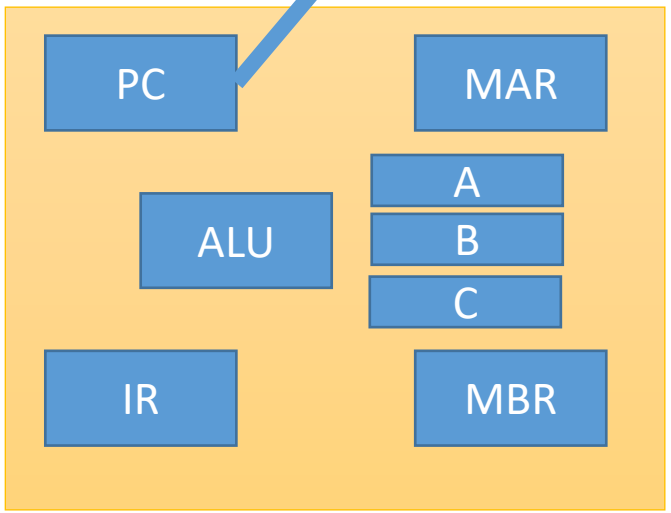
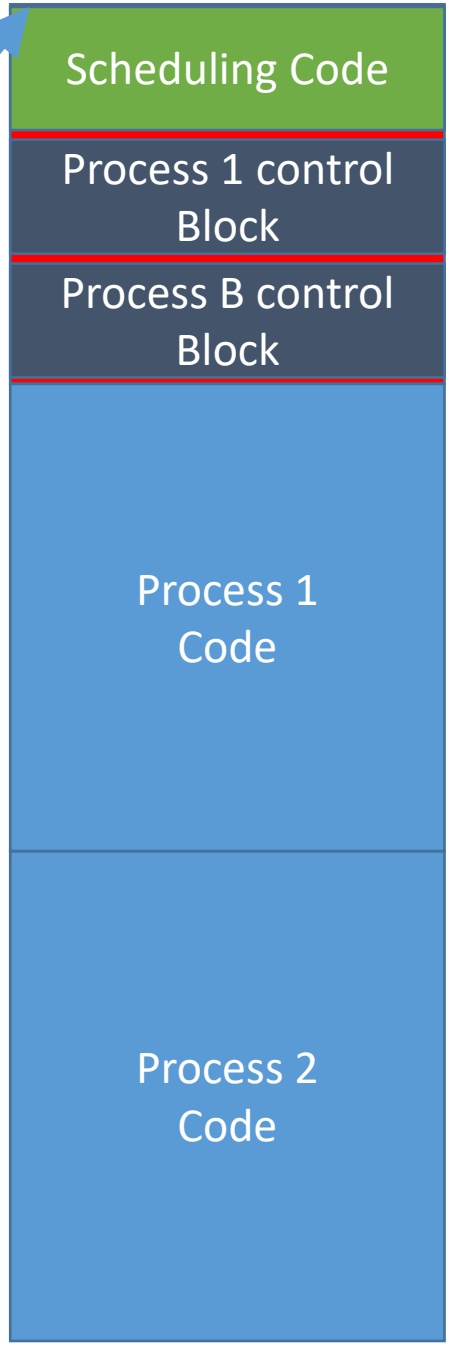


Figure 8.7 Five-State Process Model

The process is in ready state again



TIMER



Different States of a Process

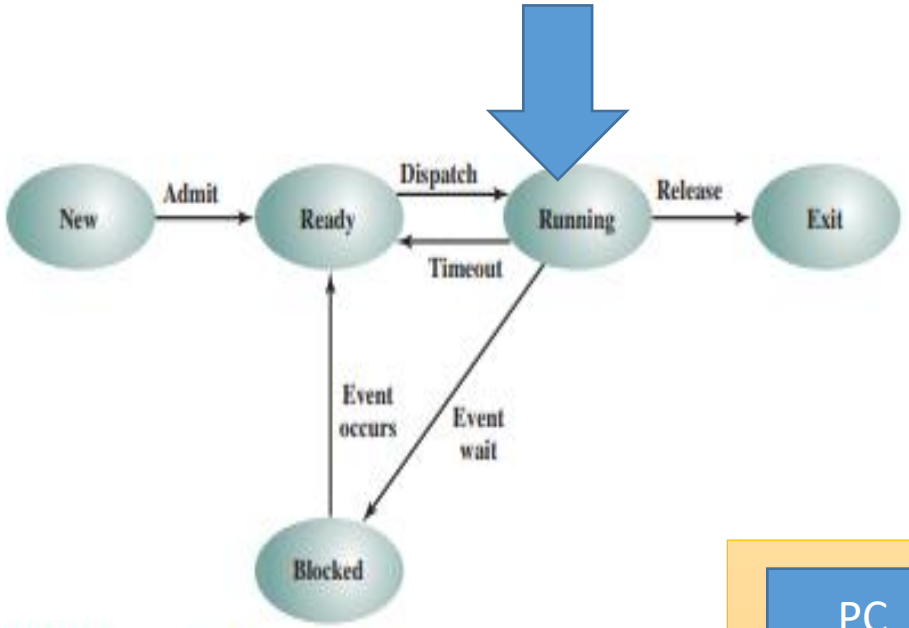
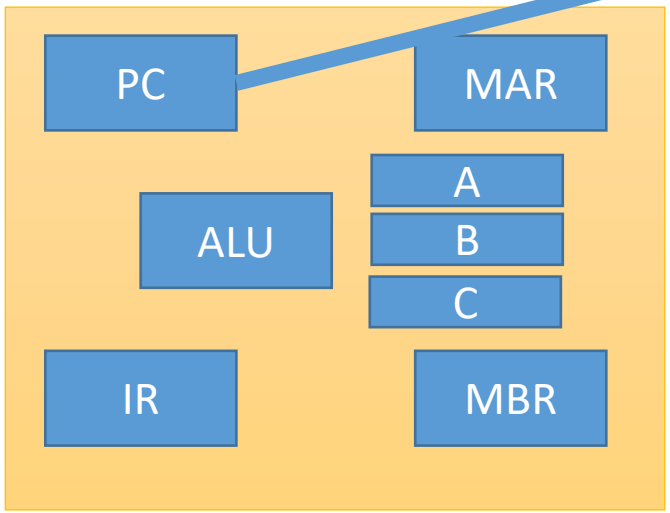
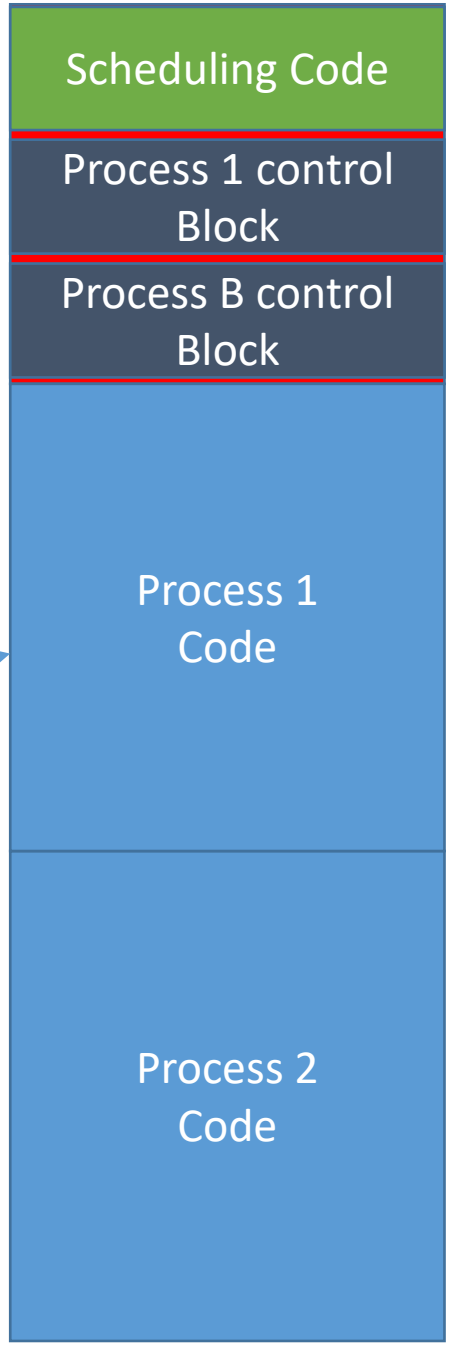


Figure 8.7 Five-State Process Model

Lets assume, that the scheduler decided to run this process again



TIMER



Different States of a Process

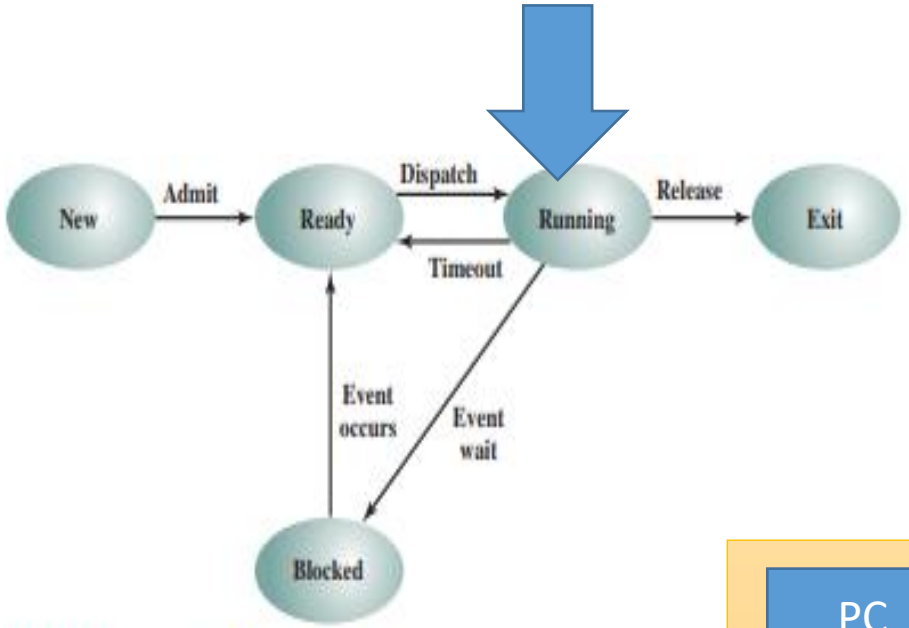
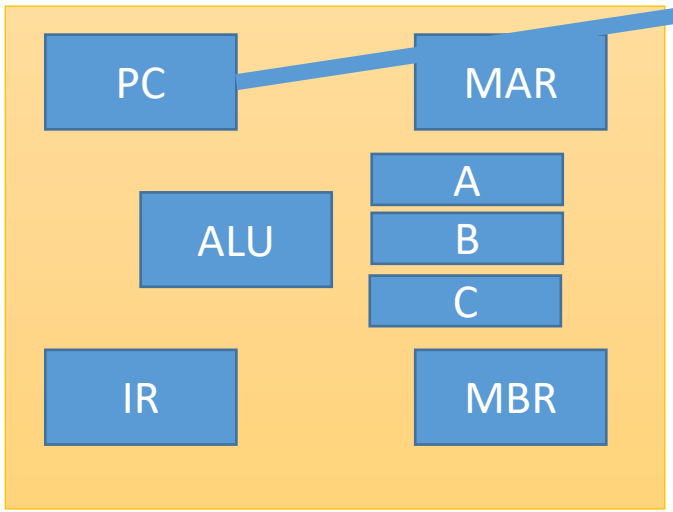
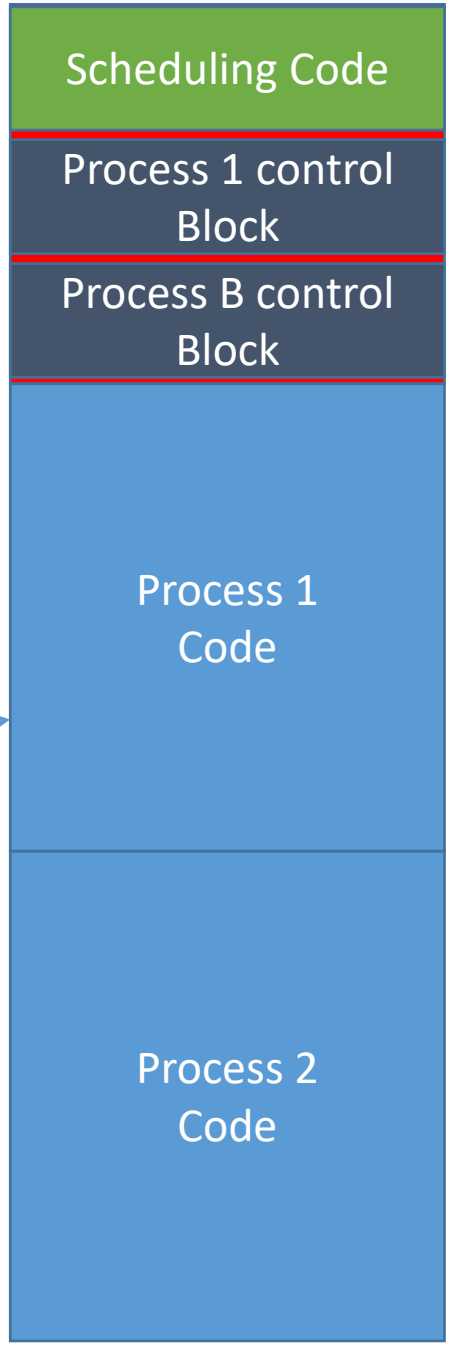


Figure 8.7 Five-State Process Model

Lets assume, that the scheduler decided to run this process again



TIMER



Different States of a Process

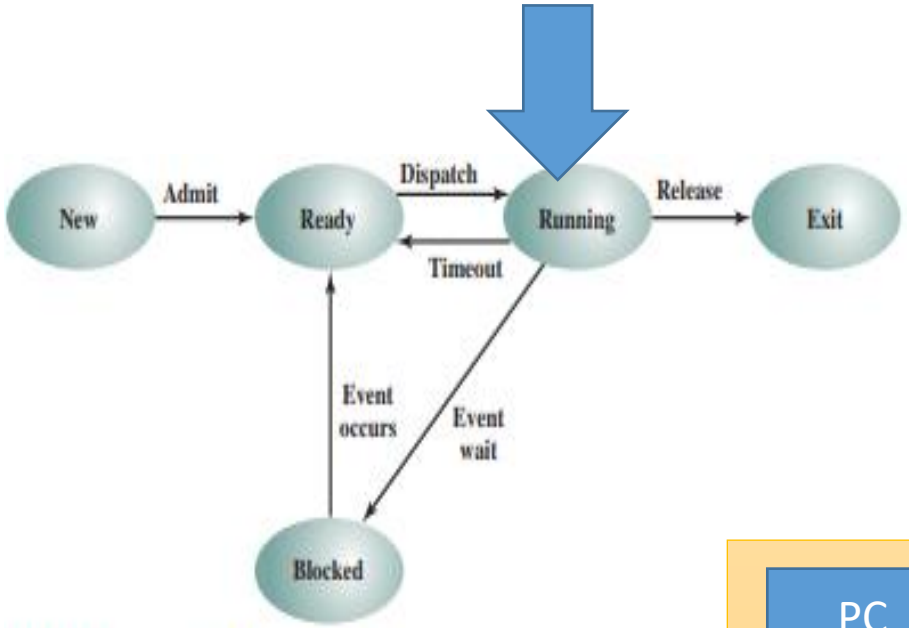
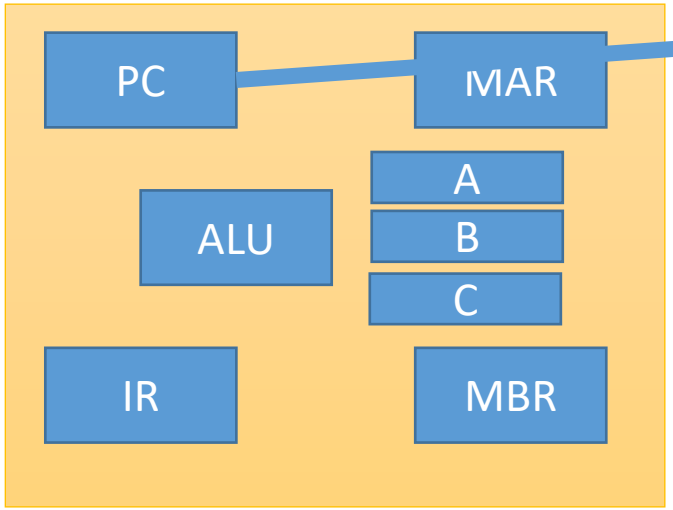
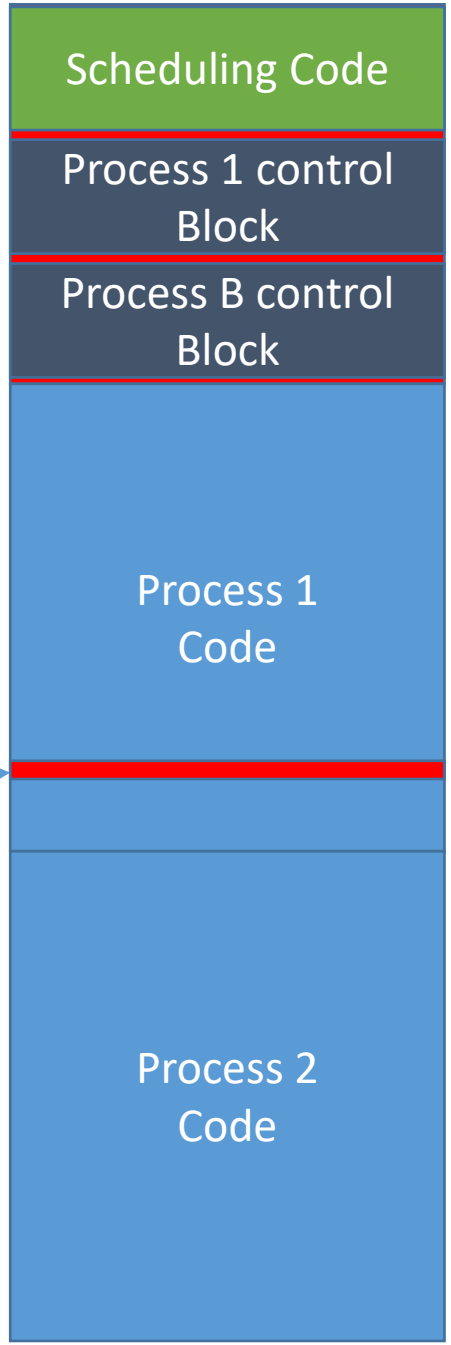


Figure 8.7 Five-State Process Model

Lets assume, that the scheduler decided to run this process again



TIMER



Different States of a Process

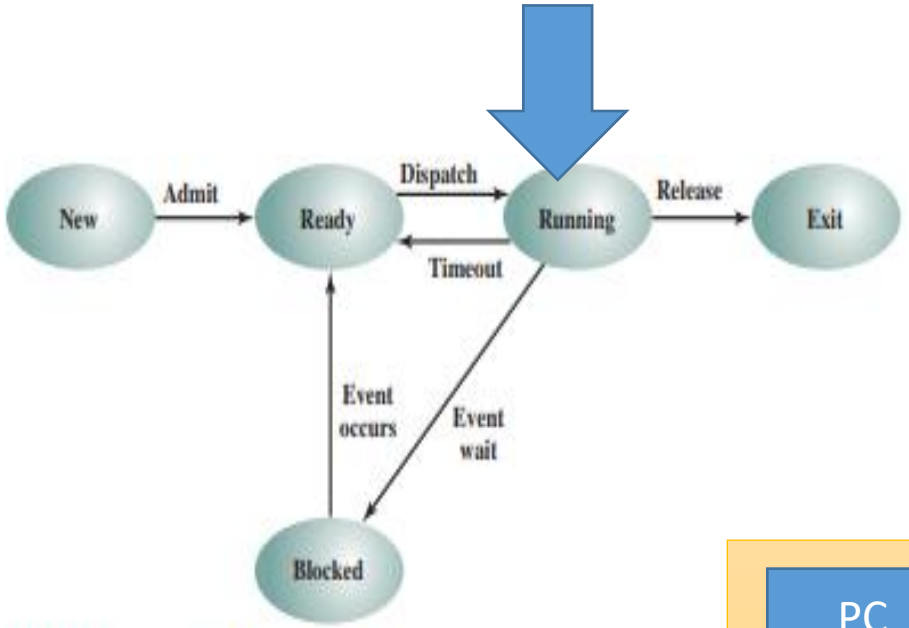
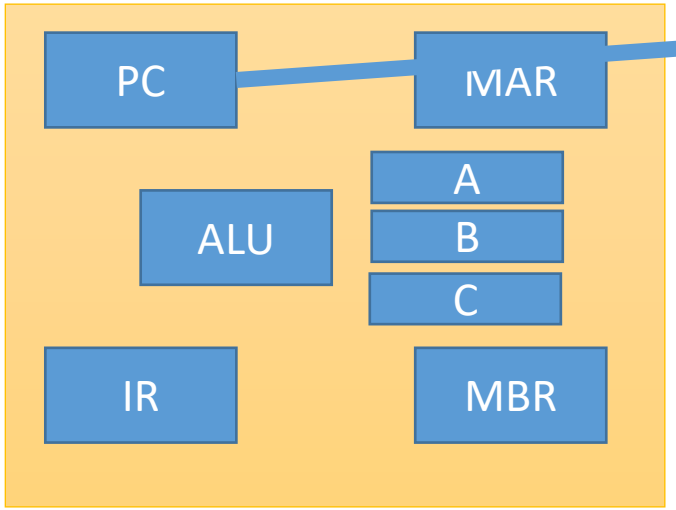
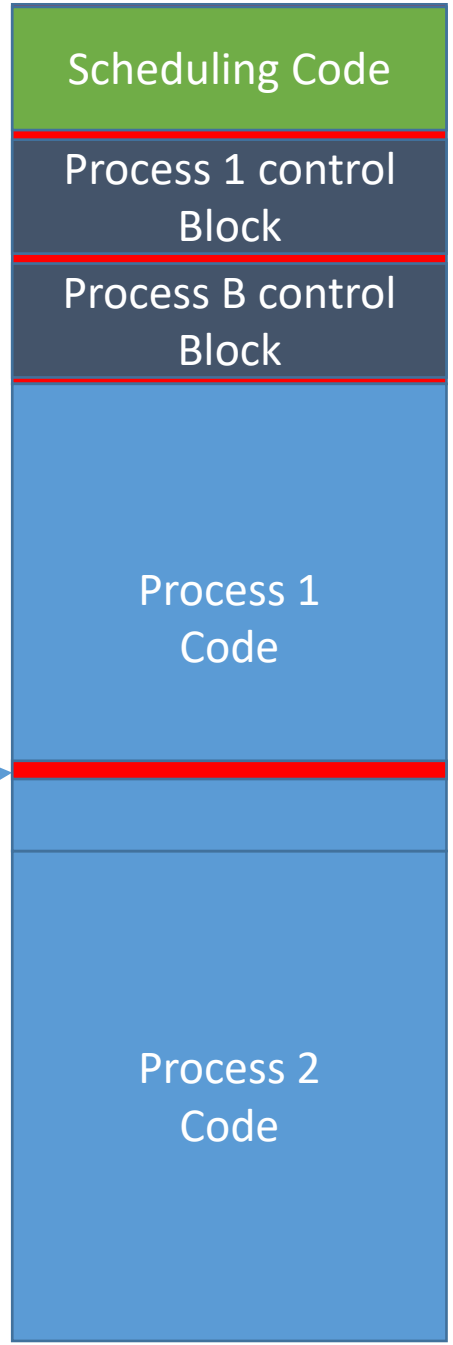


Figure 8.7 Five-State Process Model



TIMER



This is a line
Of code that
Requires **action**
To be taken
By the **user**

Different States of a Process

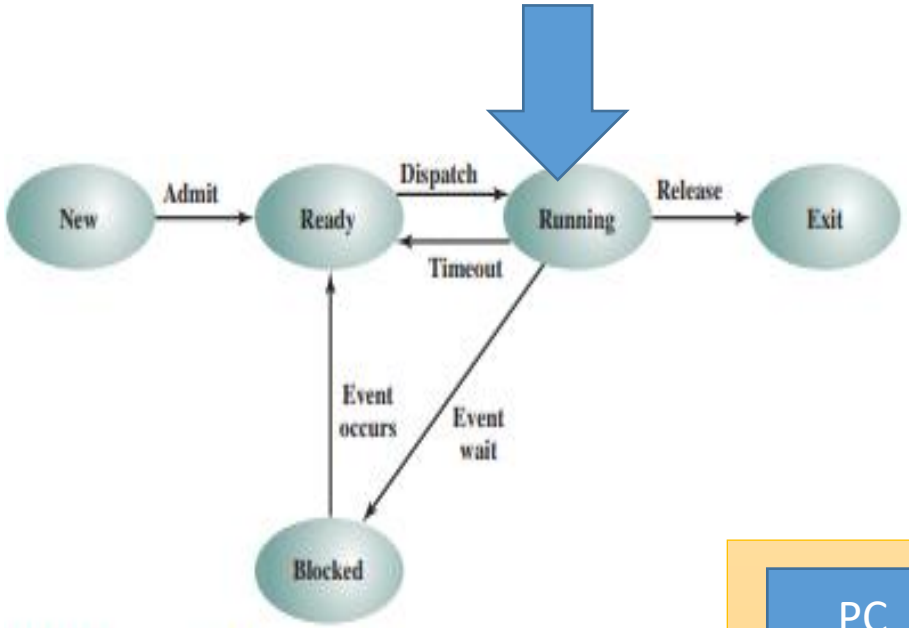
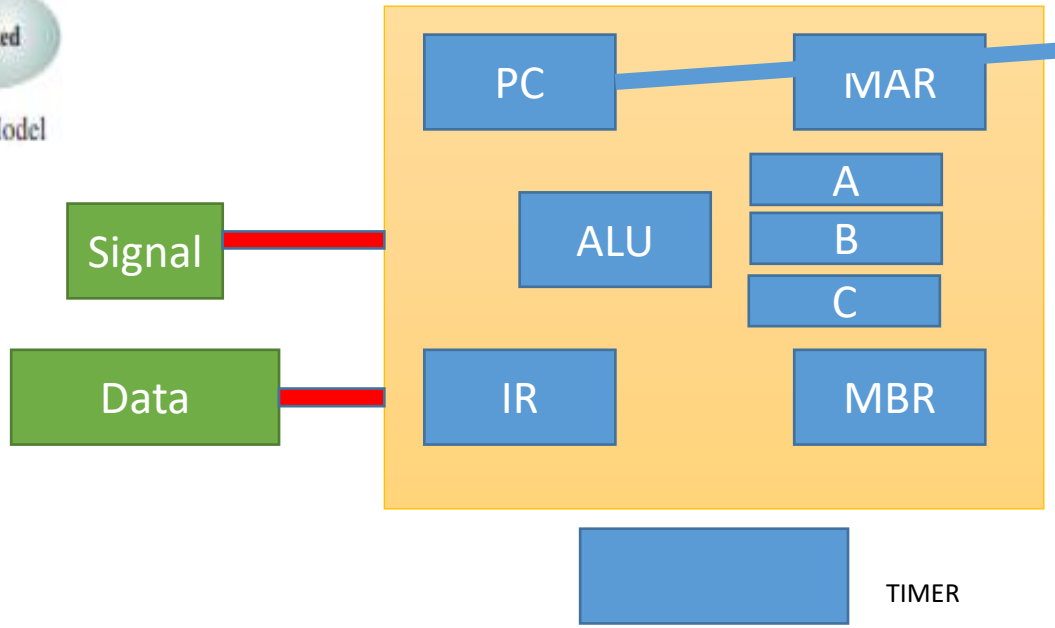
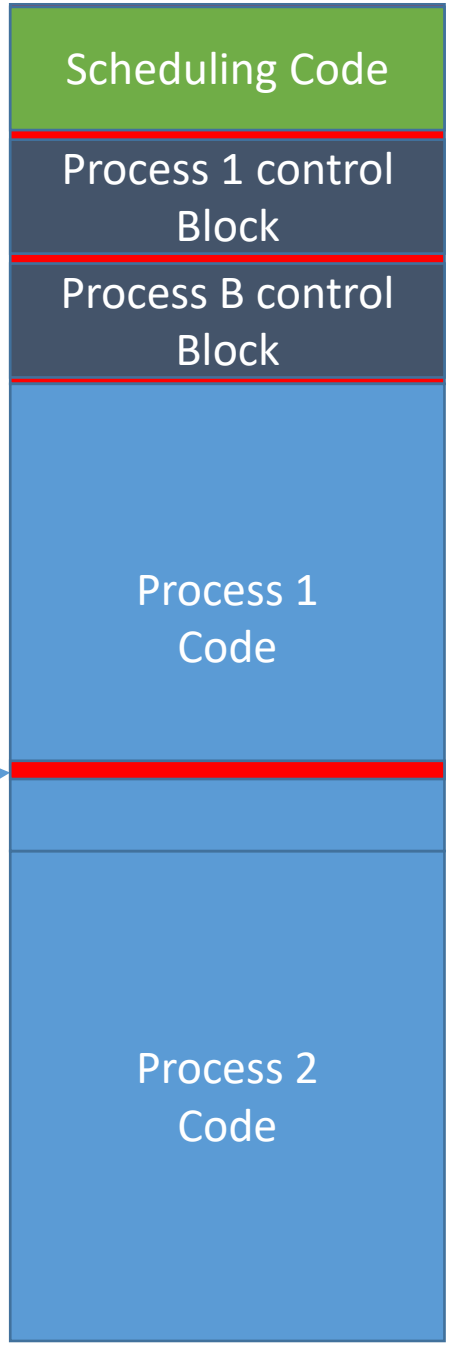


Figure 8.7 Five-State Process Model





This is a line
Of code that
Requires action
To be taken
By the user

Different States of a Process

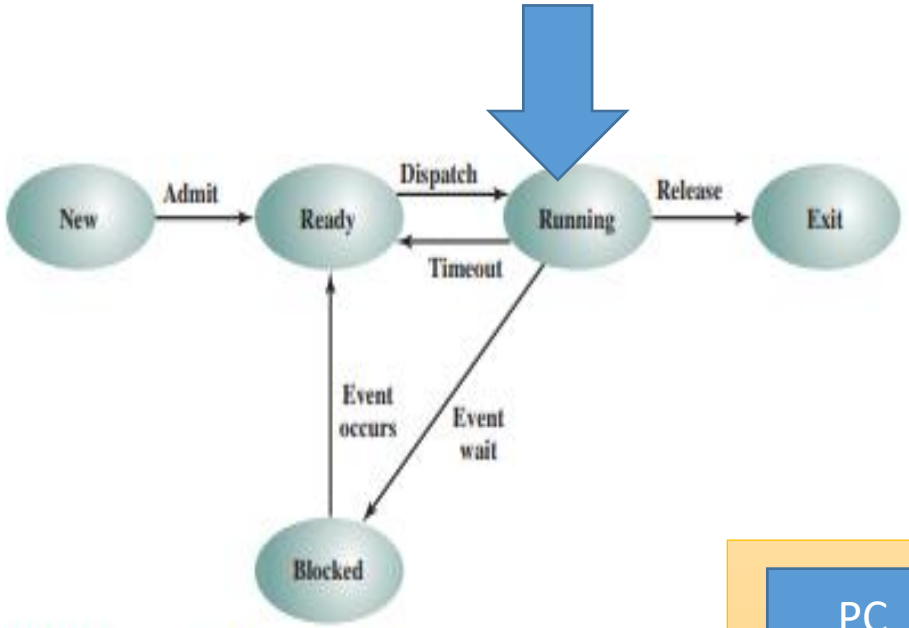
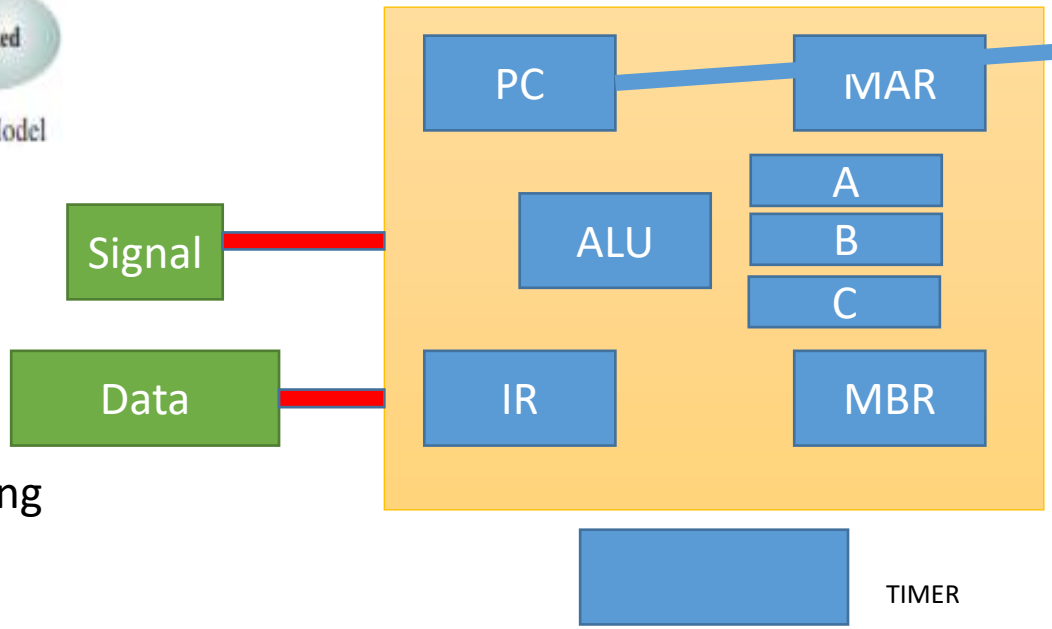
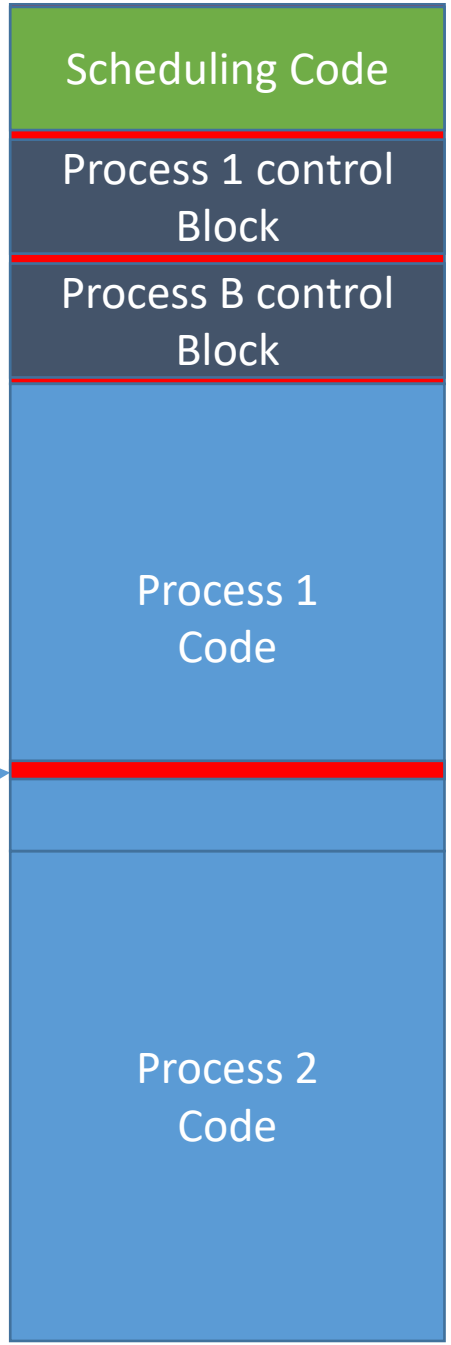


Figure 8.7 Five-State Process Model

Let us assume
That the device
Connected is a
Sensor which
Provides **data** along
With a **signal**





This is a line
Of code that
Requires **action**
To be taken
By the **user**

Which basically,
Means that the
Processor will
Have to **wait**
Before executing the
Next line.

Different States of a Process

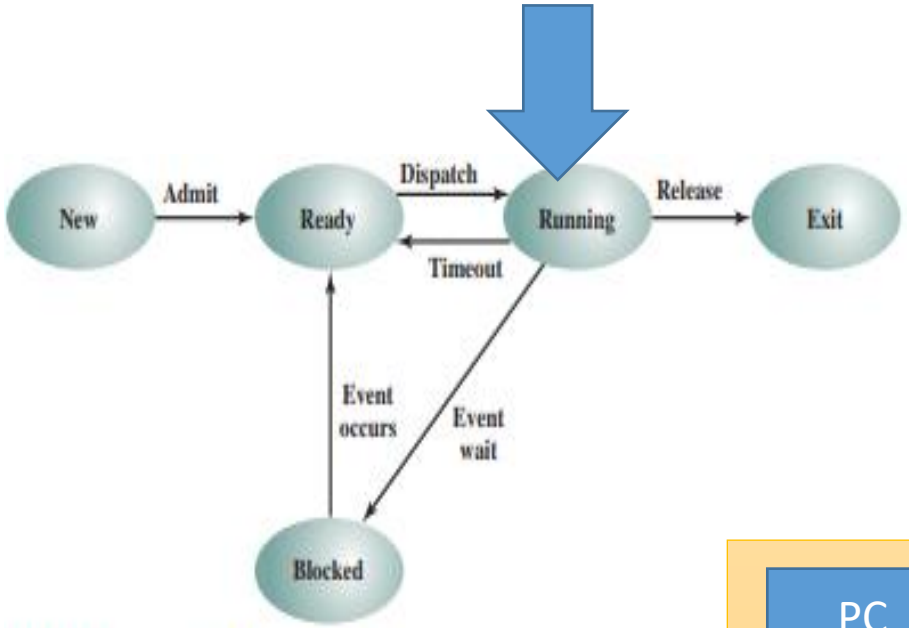
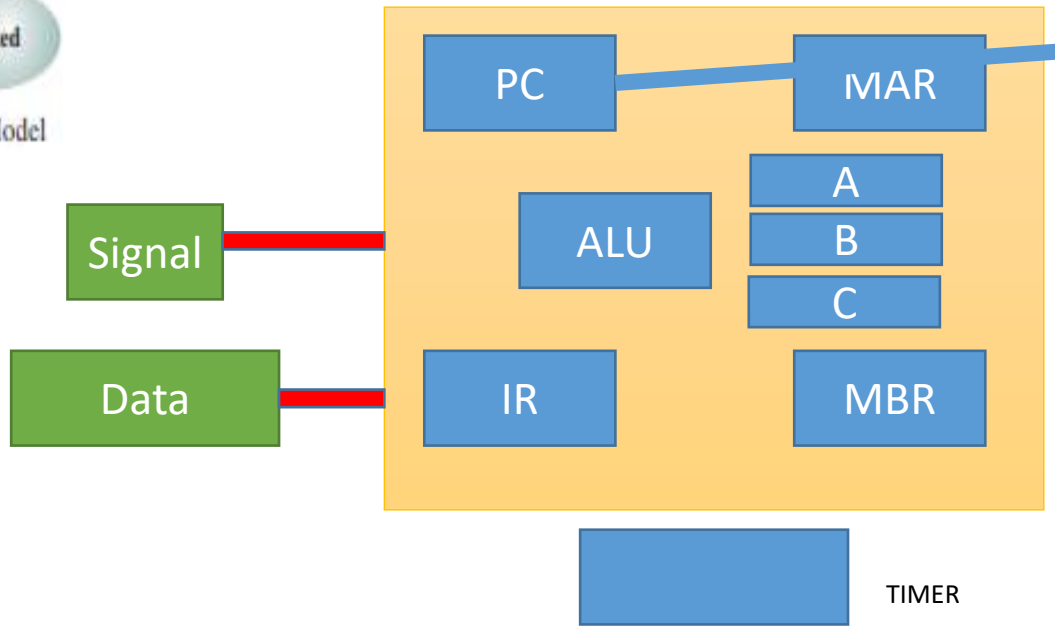
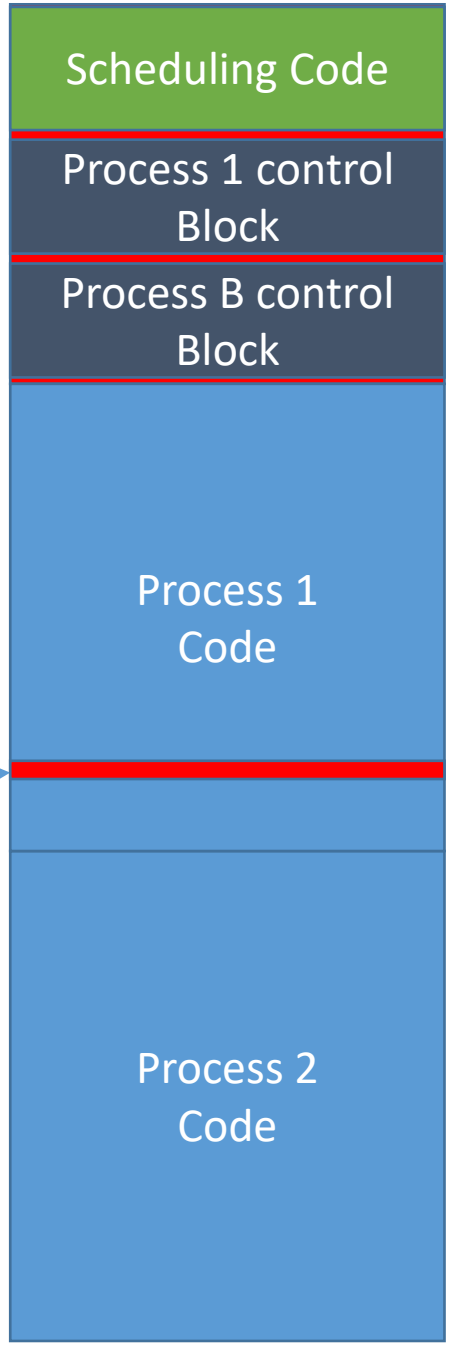


Figure 8.7 Five-State Process Model





Time runs **Out** for this Instruction!

Different States of a Process

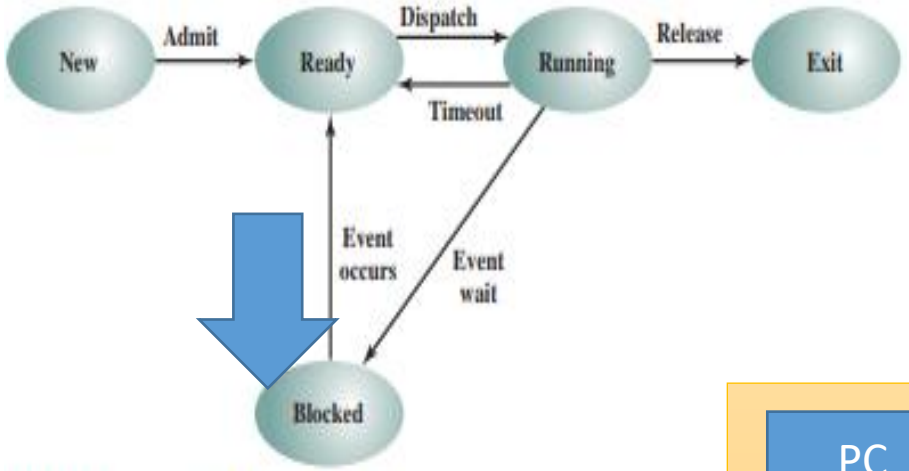
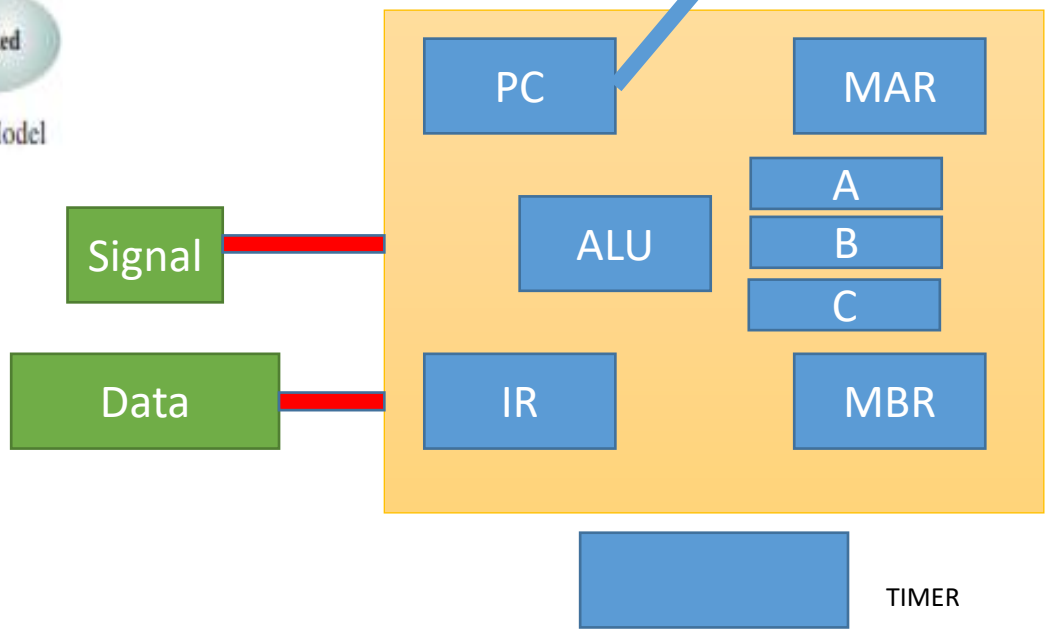
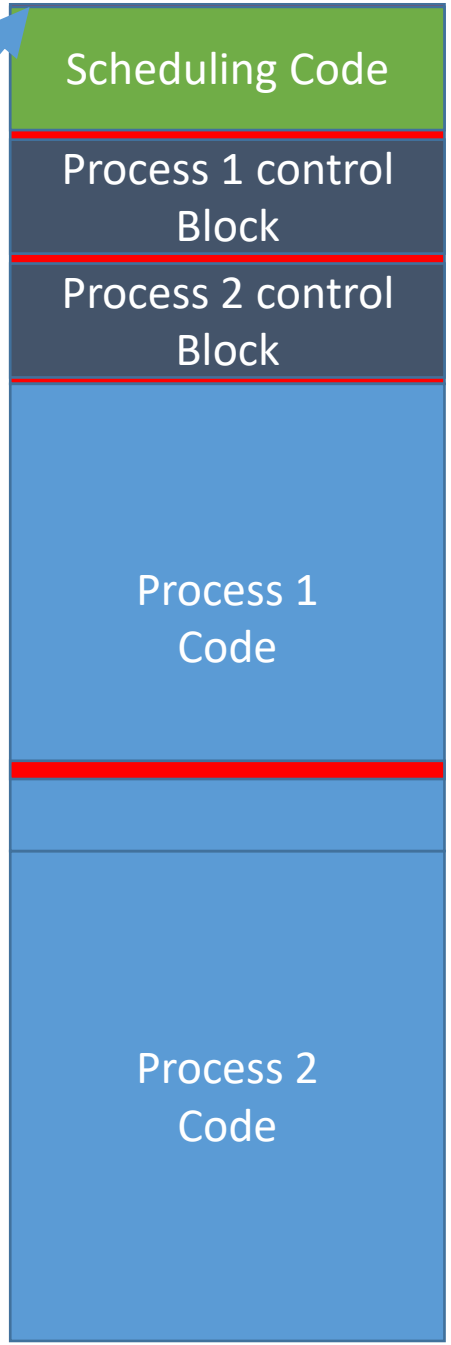


Figure 8.7 Five-State Process Model





Time runs **Out** for this Instruction!

Since it was Waiting for an Event, the OS Marks it as blocked

Different States of a Process

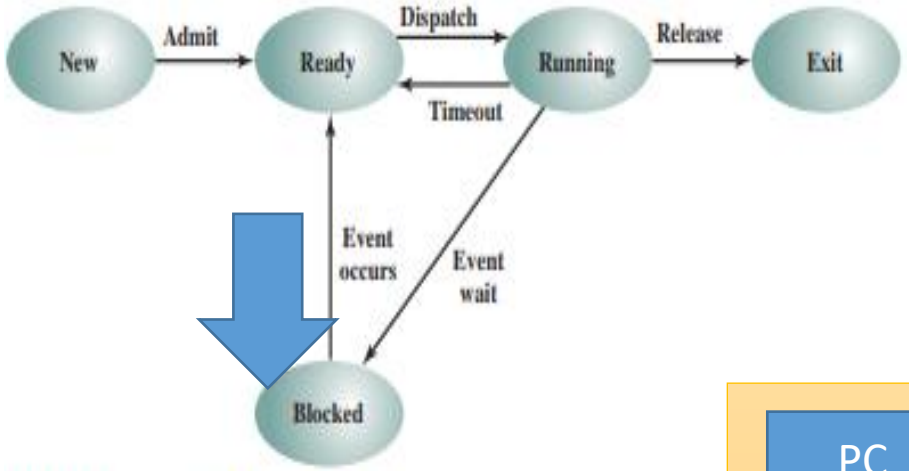
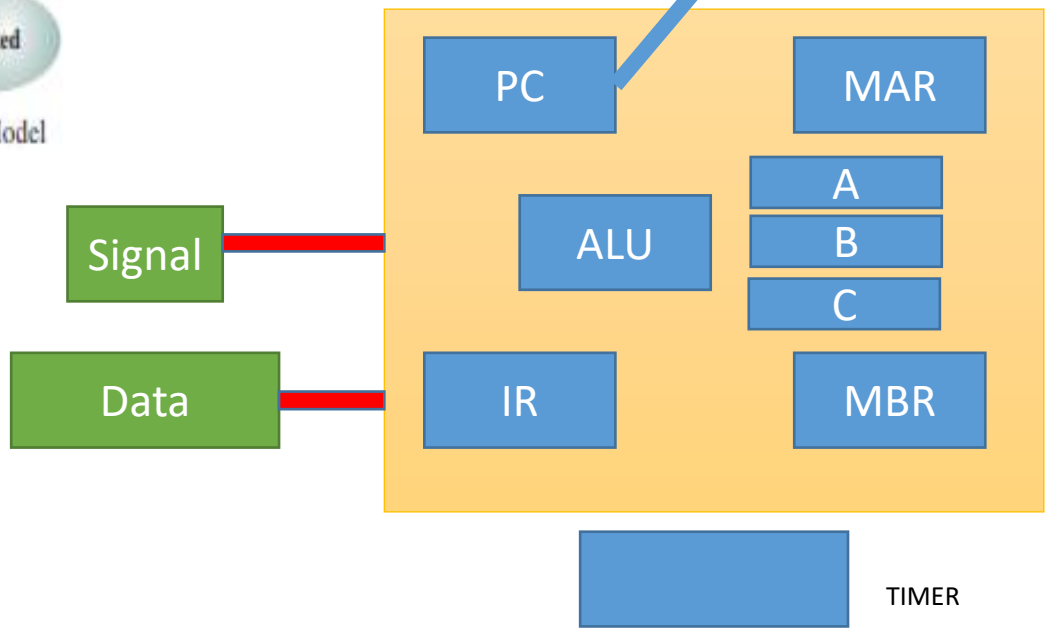
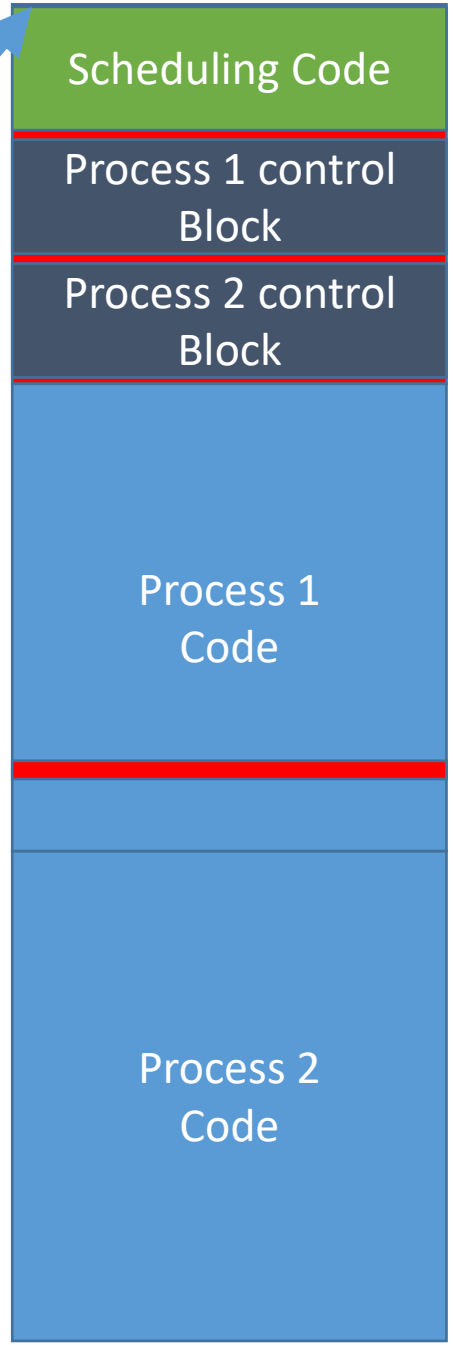


Figure 8.7 Five-State Process Model





The scheduler will
Schedule process 1
Again only if it
Gets the **input**
And the **signal**

Different States of a Process

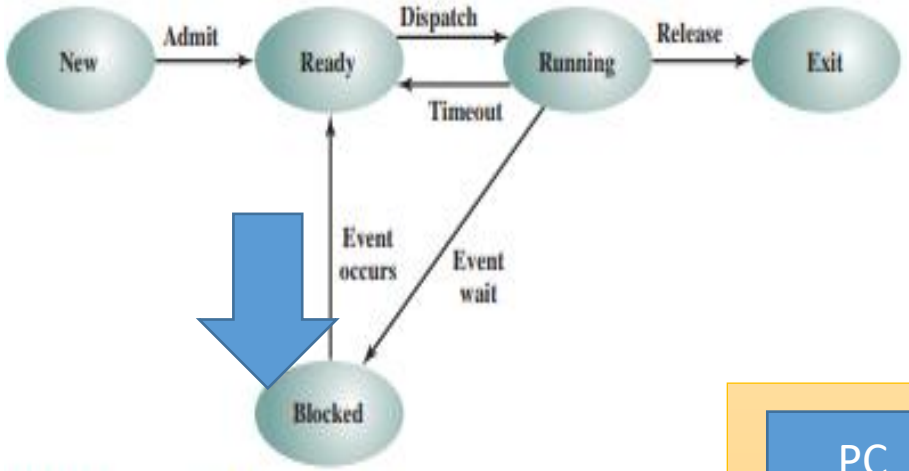
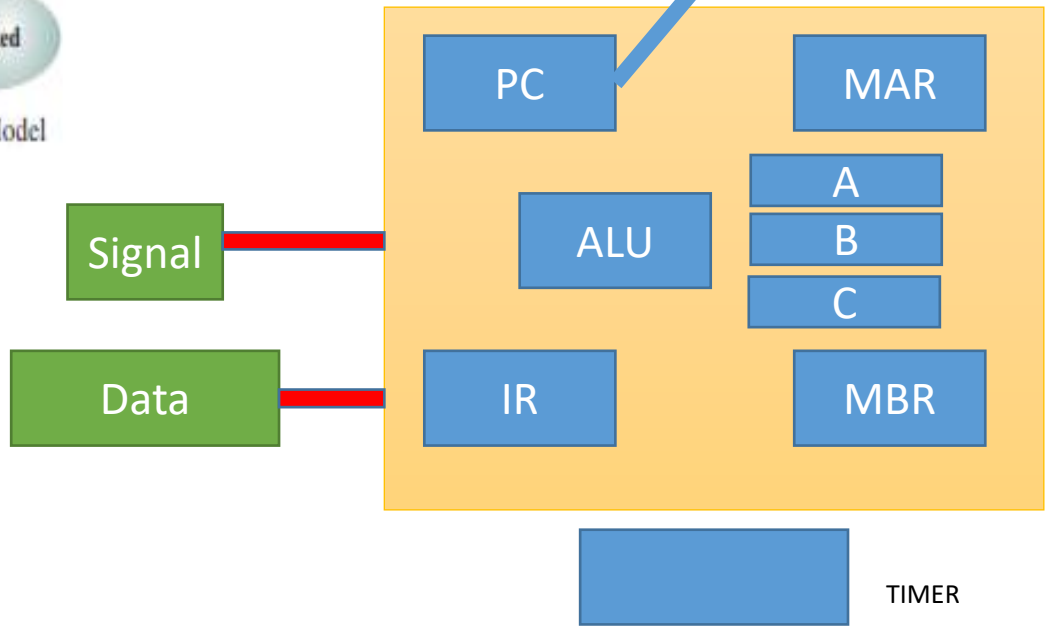
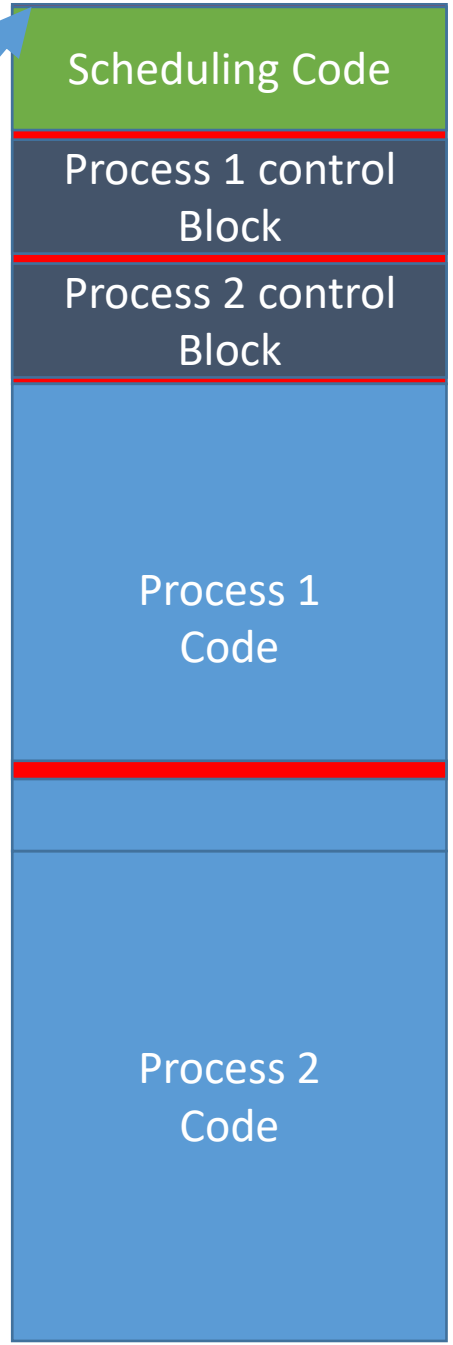


Figure 8.7 Five-State Process Model





The scheduler will Schedule process 1 Again only if it Gets the input And the signal

Every time before Scheduling a process, It will **check whether**, The **blocked process Events occurred** ot **not**

Different States of a Process

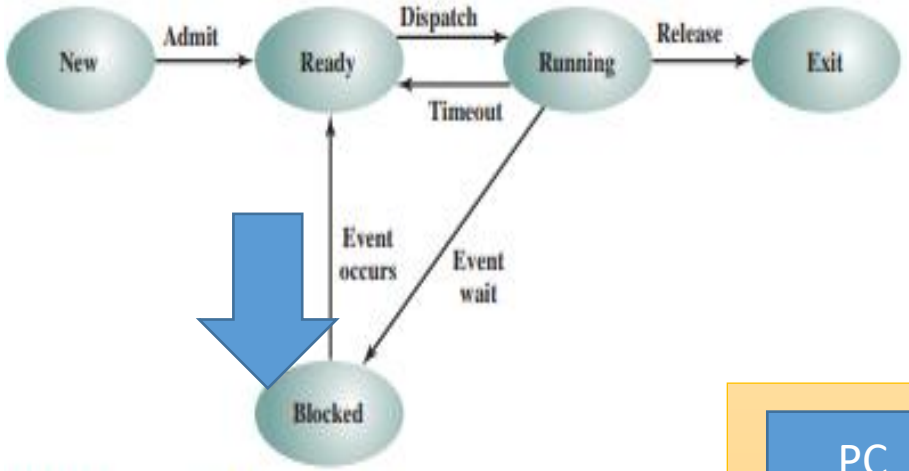
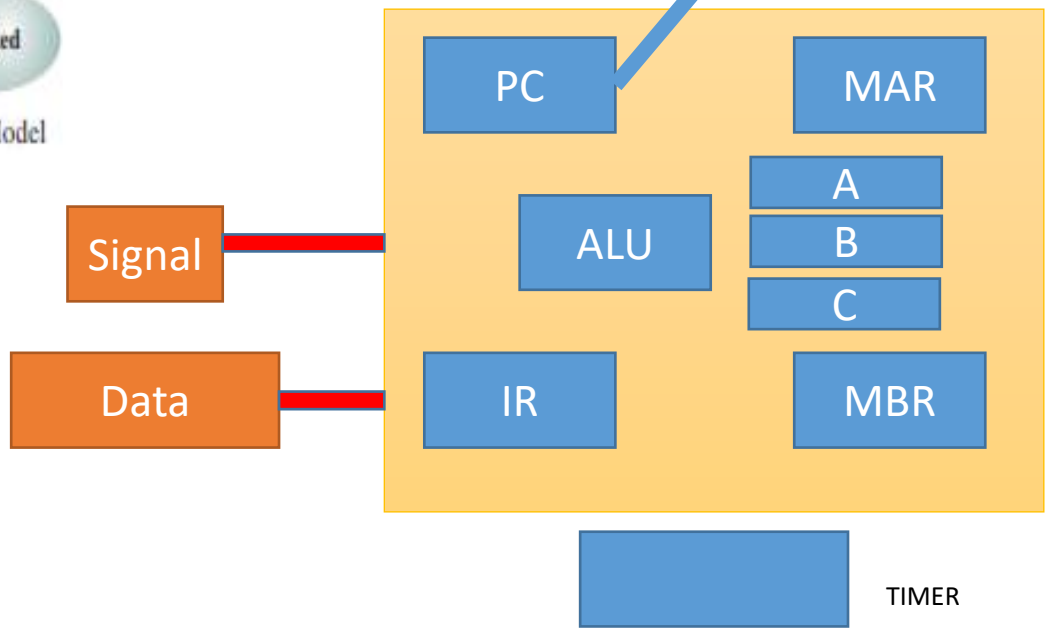
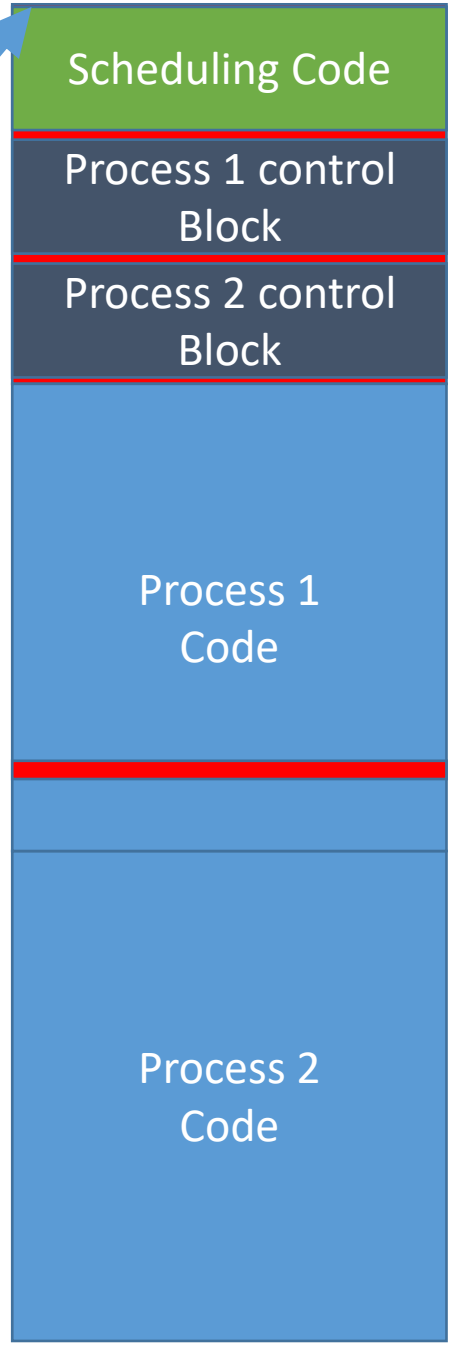


Figure 8.7 Five-State Process Model

If the **particular Event occurs**





The scheduler will Schedule process 1 Again only if it Gets the input And the signal

Every time before Scheduling a process, It will check whether, The blocked process Events occurred ot not

Different States of a Process

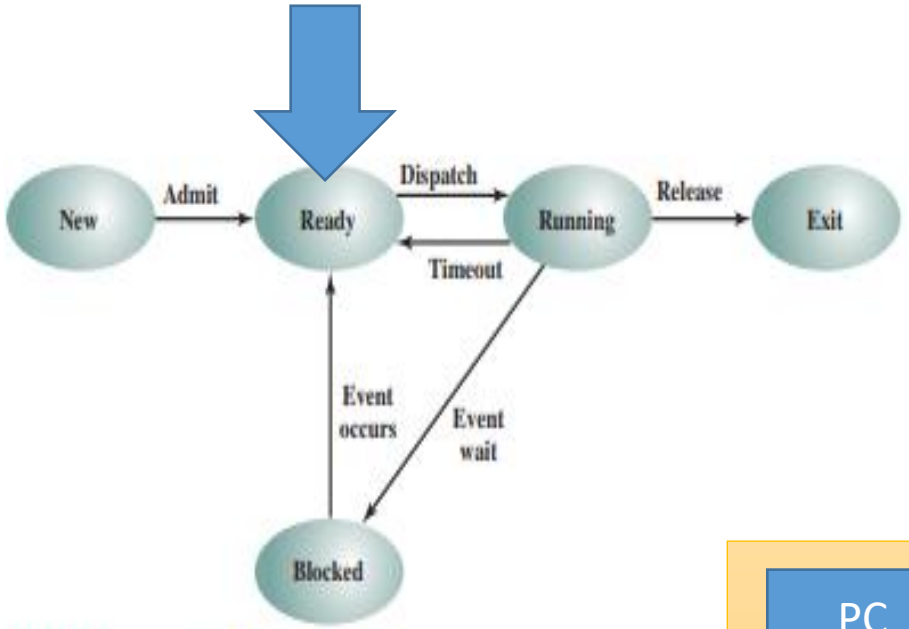
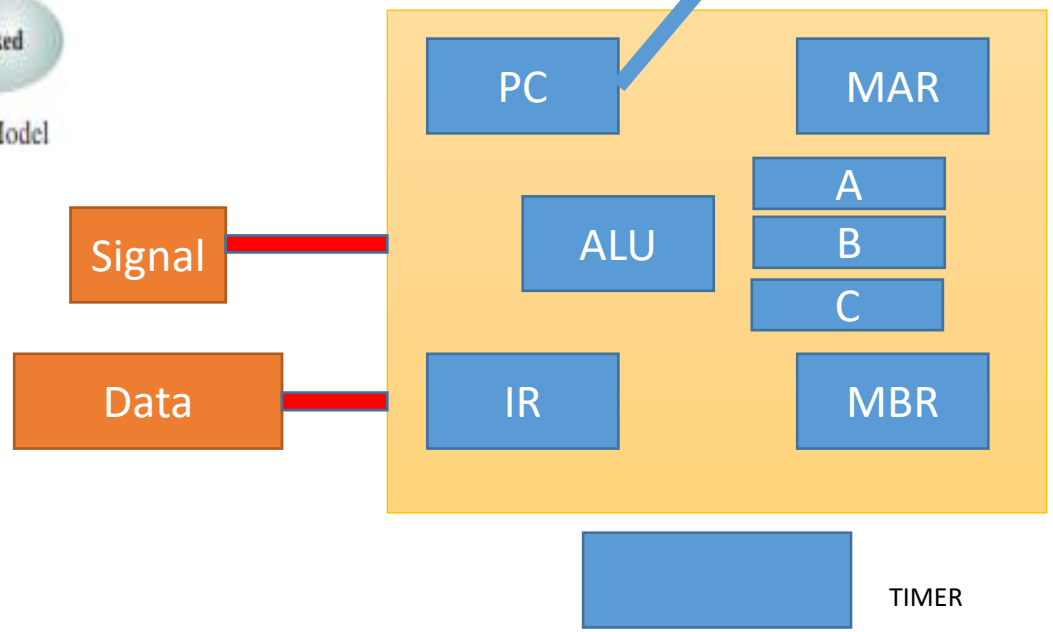
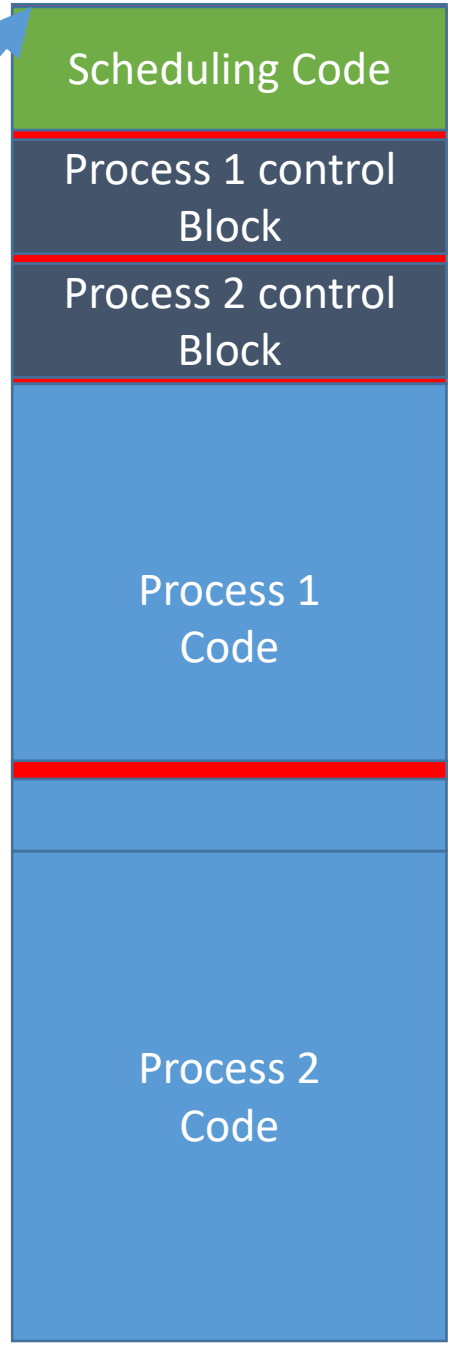


Figure 8.7 Five-State Process Model

If the particular
Event occurs

It will be **marked**
As ready again





The scheduler will
Schedule process 1
Again only if it
Gets the input
And the signal

Every time
before
Scheduling a
process,
It will check
whether,
The blocked
process
Events occurred
ot not

Different States of a Process

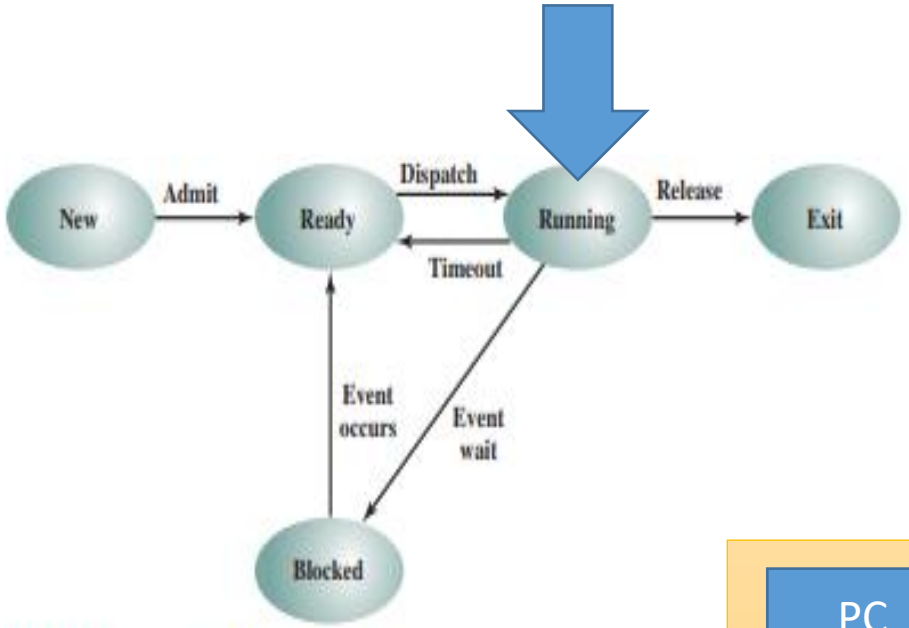
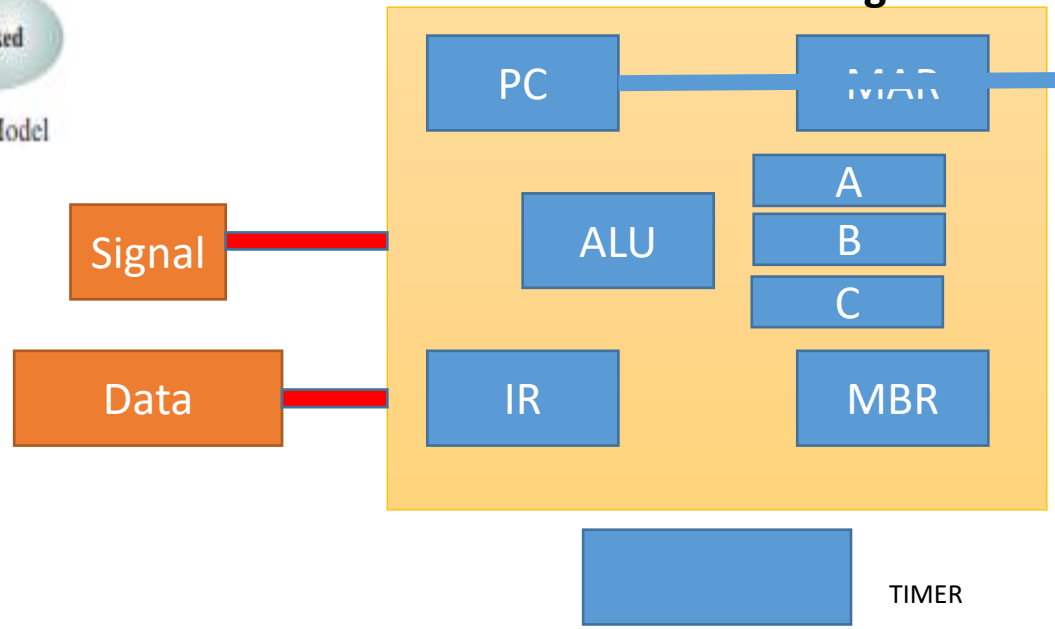


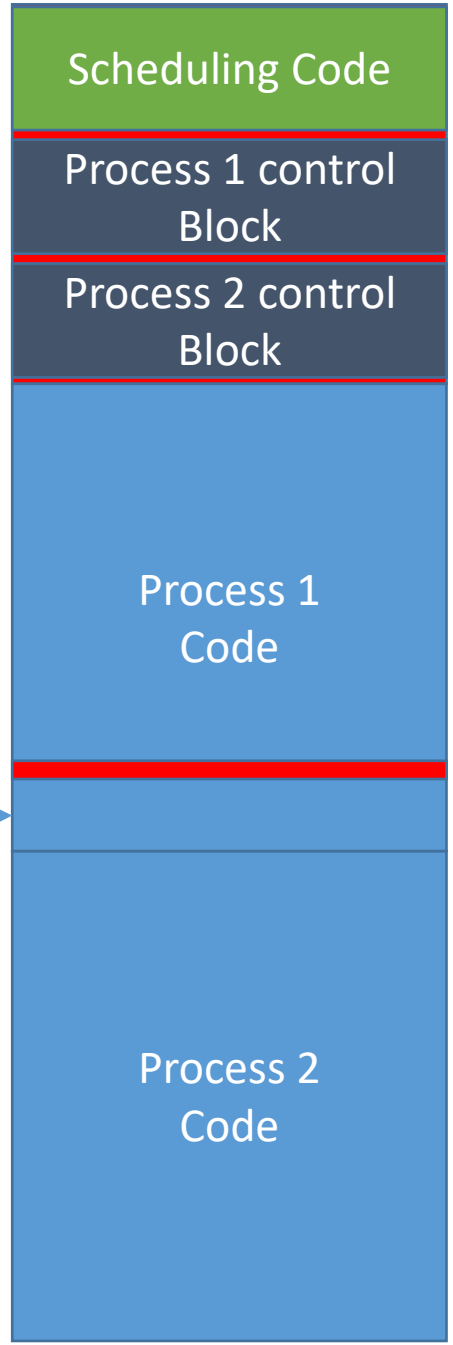
Figure 8.7 Five-State Process Model

If the particular
Event occurs

It will be marked
As ready again



And can be **Scheduled** to run **again**



The scheduler will
Schedule process 1
Again only if it
Gets the input
And the signal

Every time
before
Scheduling a
process,
It will check
whether,
The blocked
process
Events occurred
ot not

Different States of a Process

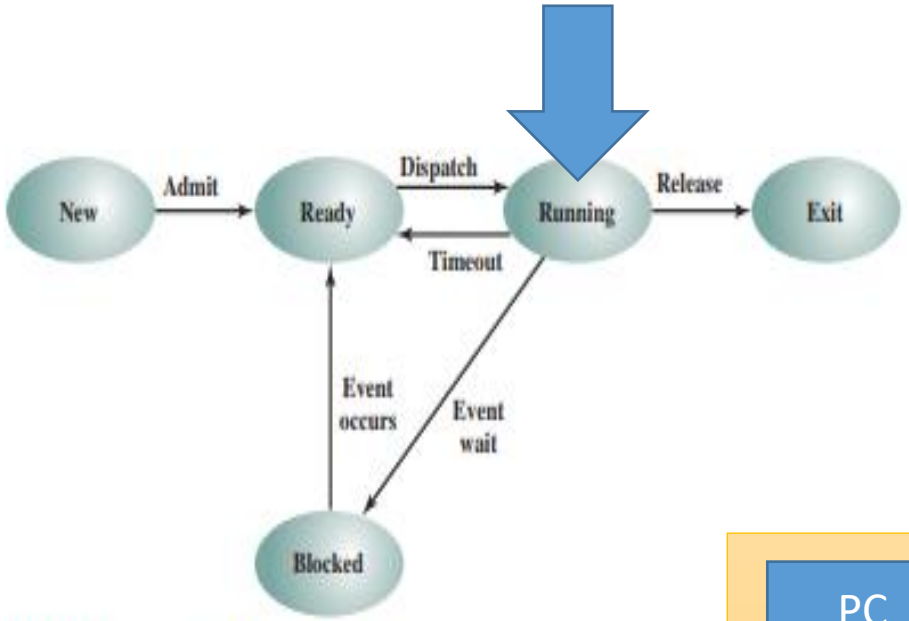
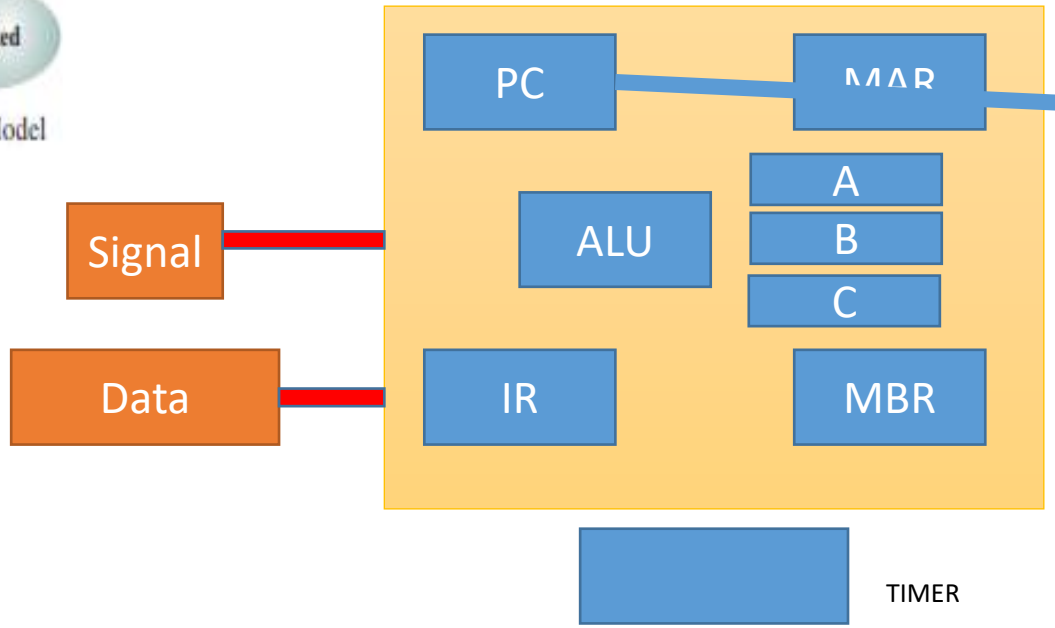
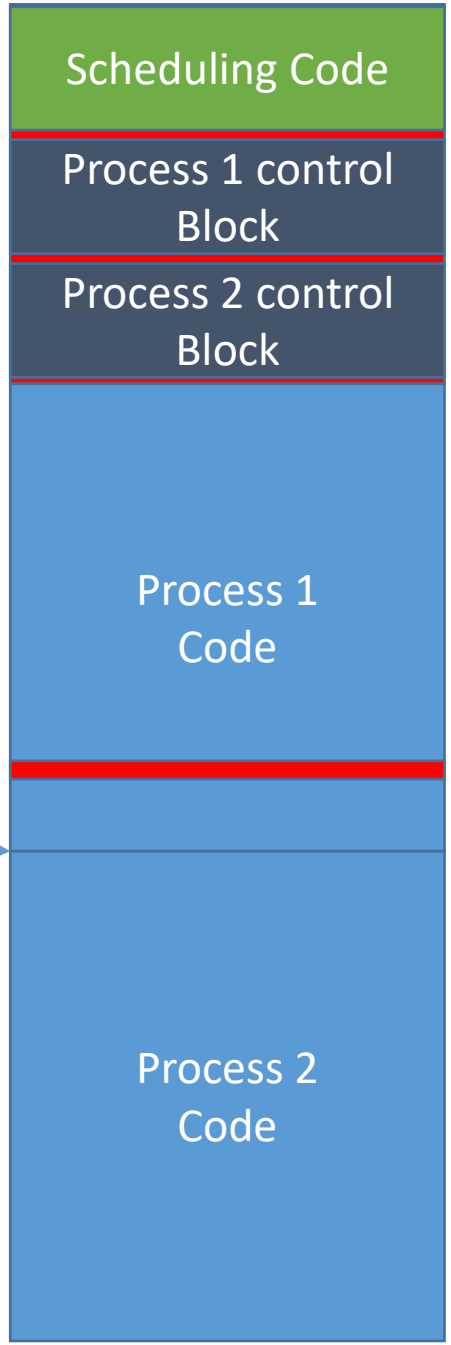


Figure 8.7 Five-State Process Model

Once its **final Line of code** Has been executed





Different States of a Process

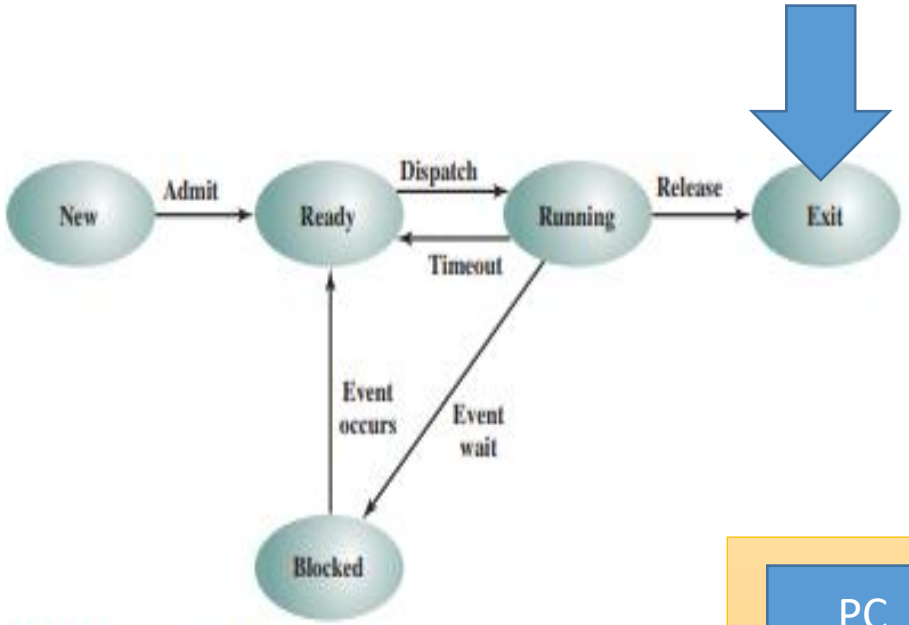
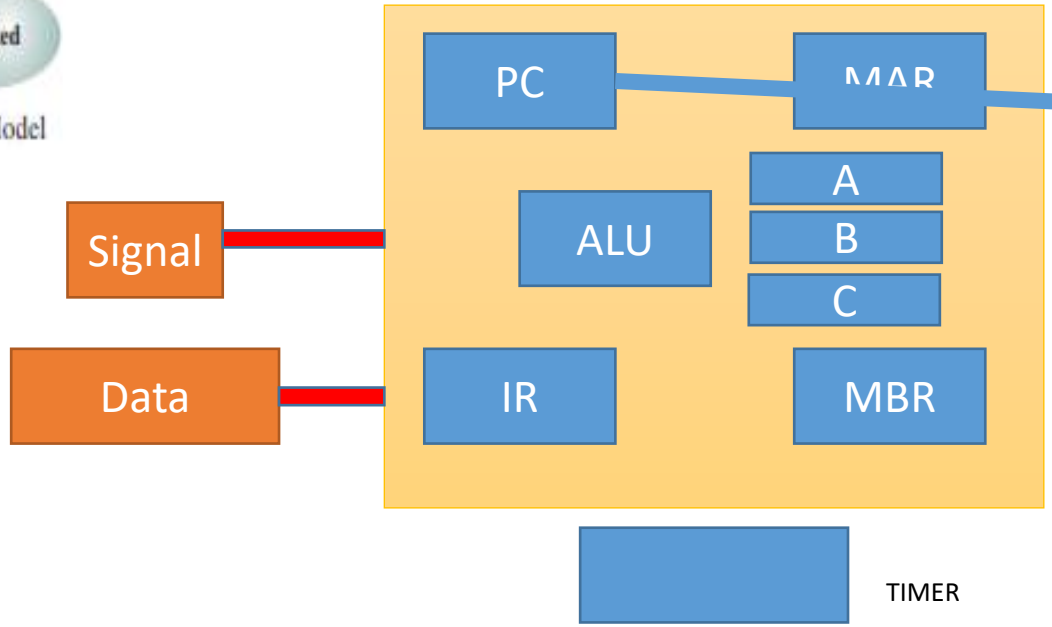
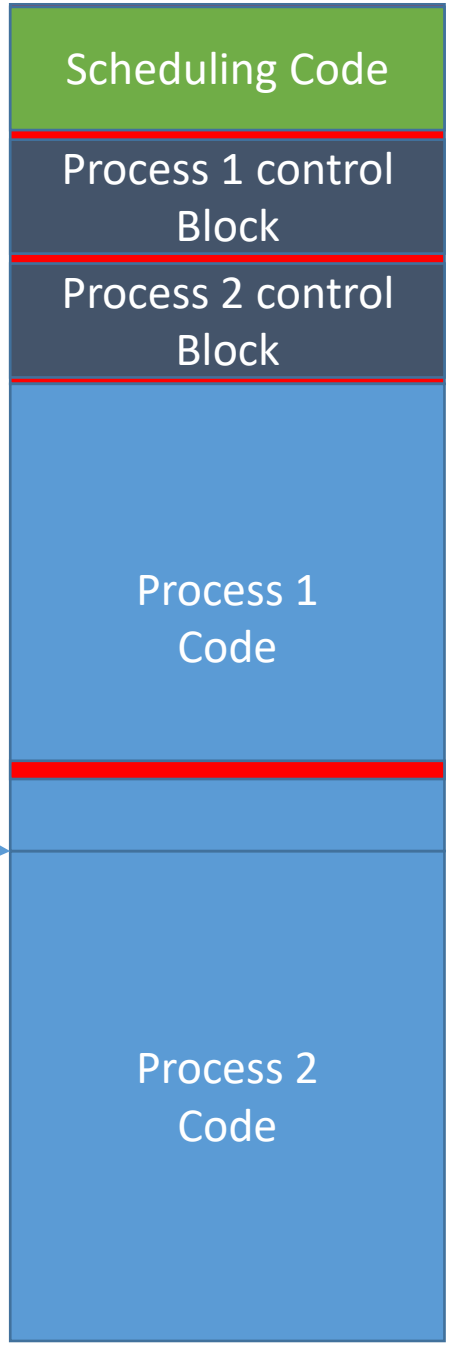


Figure 8.7 Five-State Process Model

Once its final Line of code Has been executed





The memory having the code and PCB can be declared as **free memory**, and can Be **overwritten** by **other processes**.

Different States of a Process

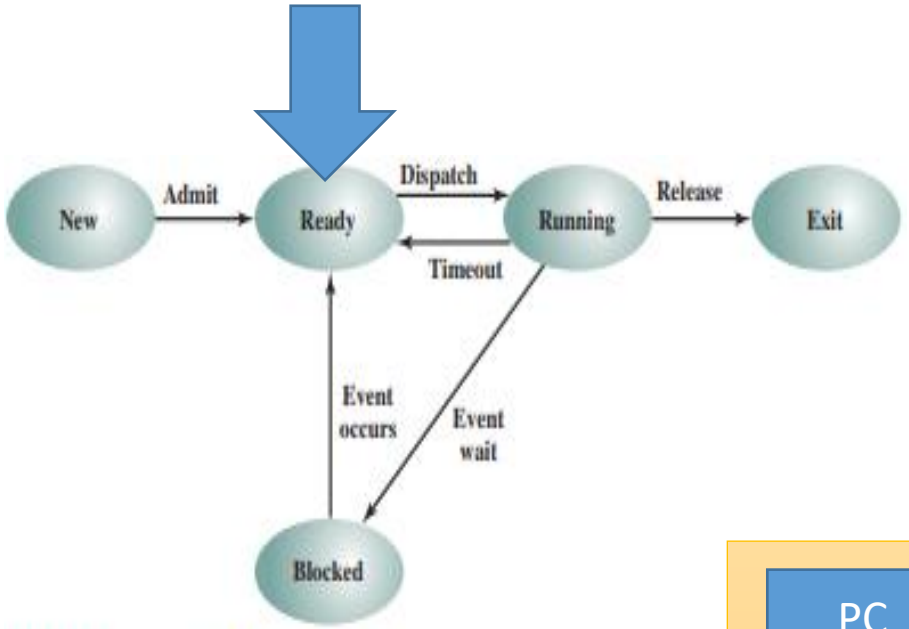
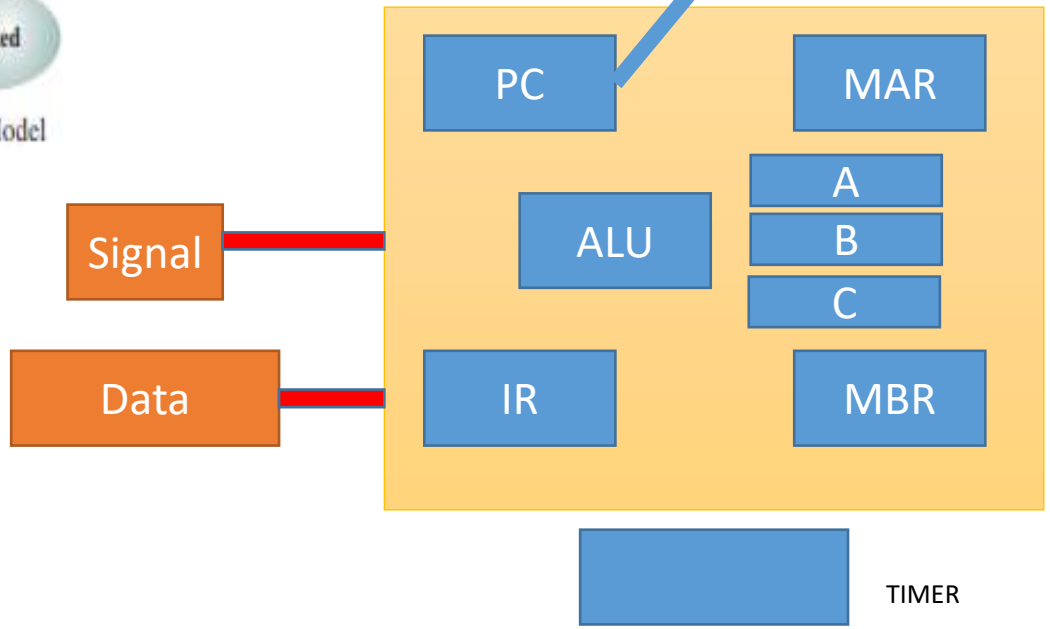
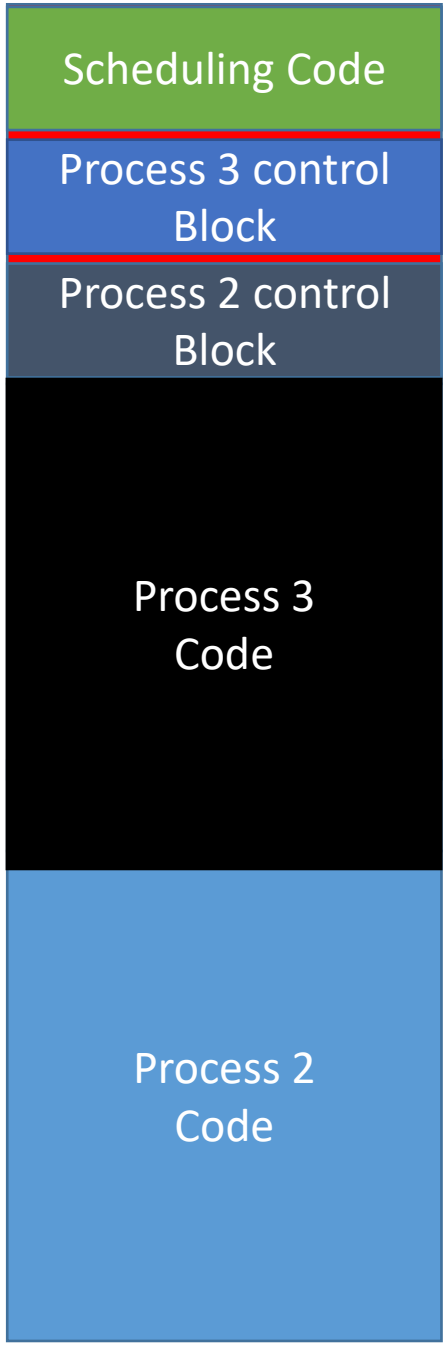


Figure 8.7 Five-State Process Model

Once its final Line of code Has been executed





The memory having the code and PCB can be declared as free memory, and can Be overwritten by other processes.

This state of the process is known as **Exit state**.

Different Types of Scheduling

Long Term Scheduling

- Basically determining which instructions will be brought from the New state to the Ready state.
- i.e. from the Storage (Hard Disc) to the Memory

Short Term Scheduling

- Determines which process will be getting the processor
- i.e. which process will go from ready state to running state

3.1 Memory Management and its necessities

Memory Management

- If multiple programs were not being run simultaneously.
- And there was a lot of memory space.
- We could reserve just one part of the memory for OS and the other part for the processes.

- But multiple processes, finite memory -> efficient memory management required.
- A situation may occur where all the processes are blocked

Memory Management

- If **multiple** programs were **not** being run simultaneously.
- And there was a **lot** of **memory space**.
- We could reserve just one part of the memory for **OS** and the other part for the **processes**.

- But multiple processes, finite memory -> efficient memory management required.
- A situation may occur where all the processes are blocked

Memory Management

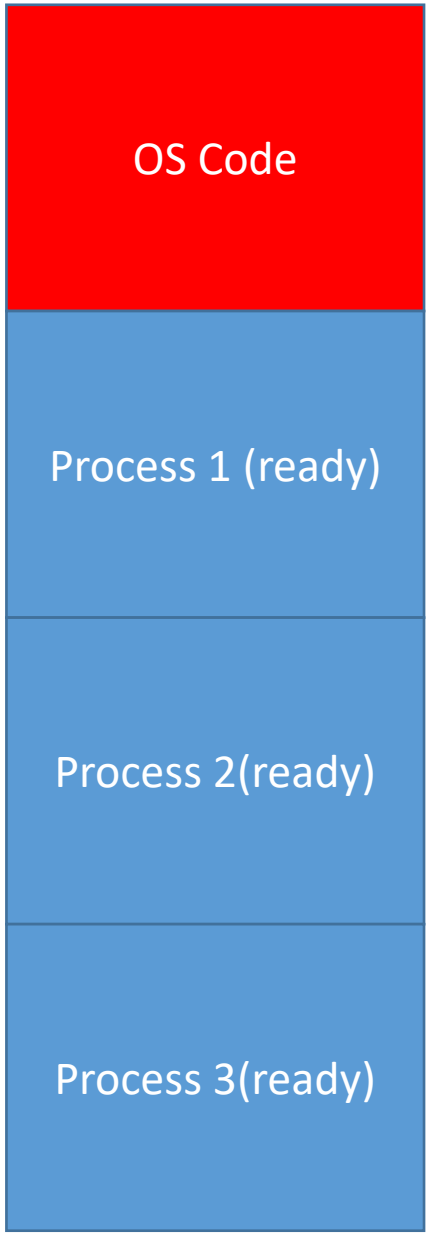
- If multiple programs were not being run simultaneously.
- And there was a lot of memory space.
- We could reserve just one part of the memory for OS and the other part for the processes.

- But multiple processes, **finite memory** -> **efficient** memory management required.
- A situation may occur where all the **processes are blocked**

Memory Management

- If multiple programs were not being run simultaneously.
- And there was a lot of memory space.
- We could reserve just one half of the memory for OS and the other half for the process.

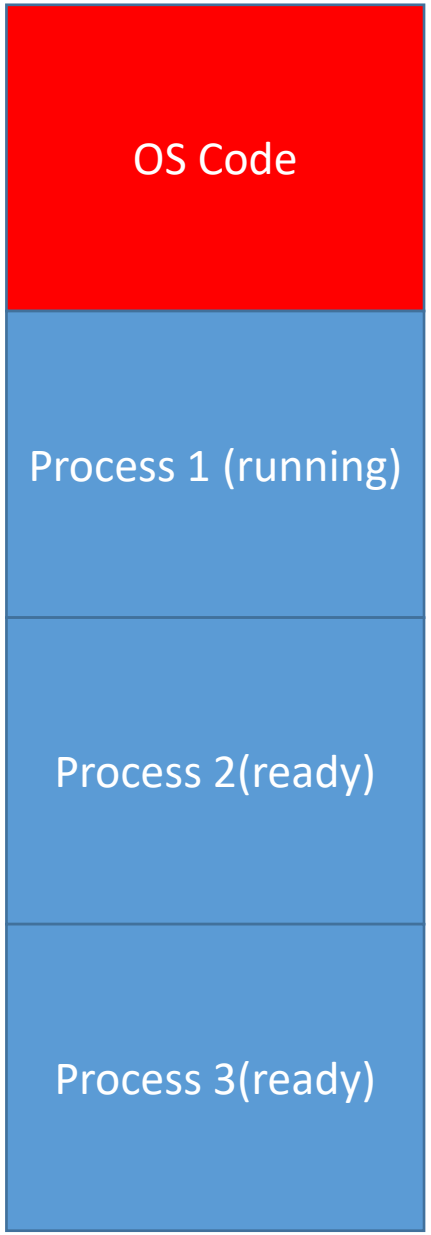
- But multiple processes, finite memory -> efficient memory management required.
- A situation may occur where all the processes are blocked



Memory Management

- If multiple programs were not being run simultaneously.
- And there was a lot of memory space.
- We could reserve just one half of the memory for OS and the other half for the process.

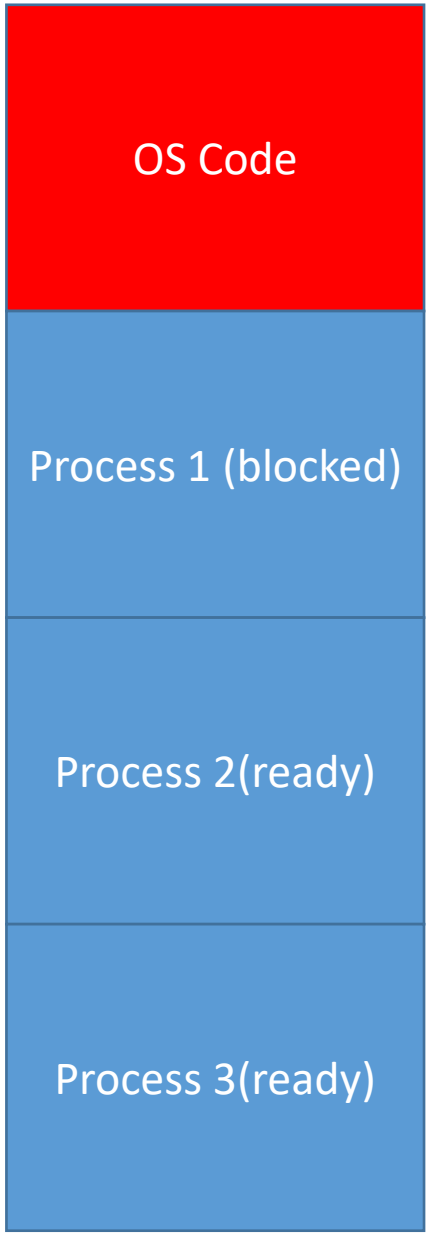
- But multiple processes, finite memory -> efficient memory management required.
- A situation may occur where all the processes are blocked



Memory Management

- If multiple programs were not being run simultaneously.
- And there was a lot of memory space.
- We could reserve just one half of the memory for OS and the other half for the process.

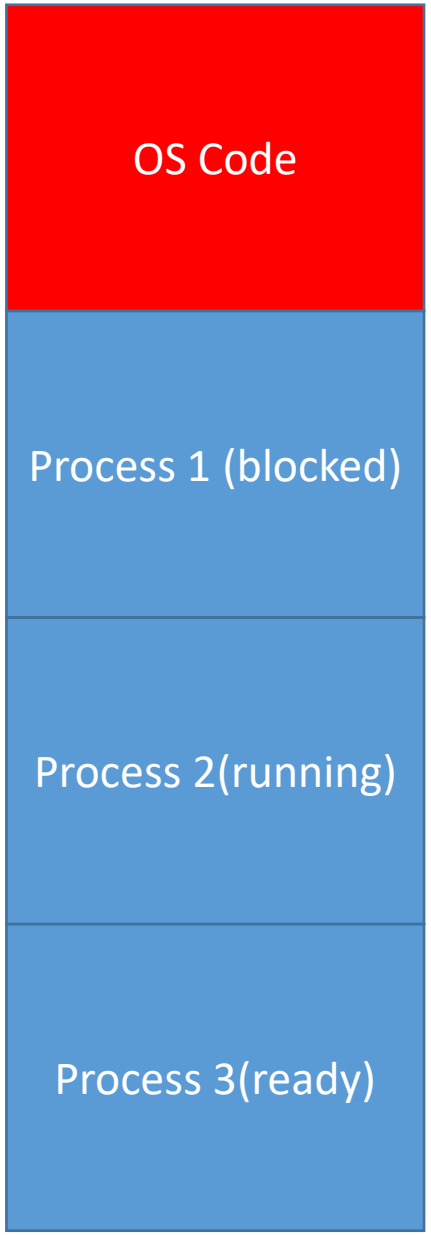
- But multiple processes, finite memory -> efficient memory management required.
- A situation may occur where all the processes are blocked



Memory Management

- If multiple programs were not being run simultaneously.
- And there was a lot of memory space.
- We could reserve just one half of the memory for OS and the other half for the process.

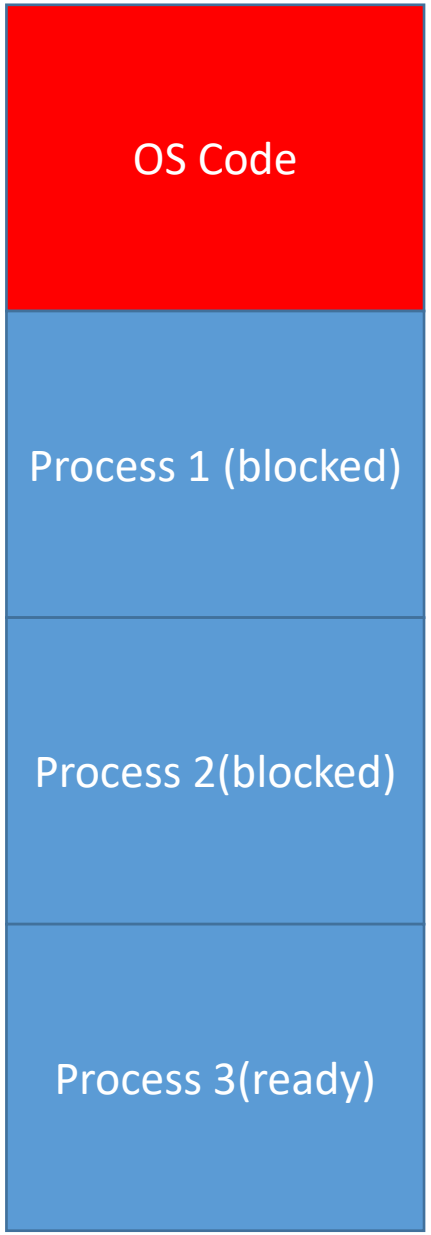
- But multiple processes, finite memory -> efficient memory management required.
- A situation may occur where all the processes are blocked



Memory Management

- If multiple programs were not being run simultaneously.
- And there was a lot of memory space.
- We could reserve just one half of the memory for OS and the other half for the process.

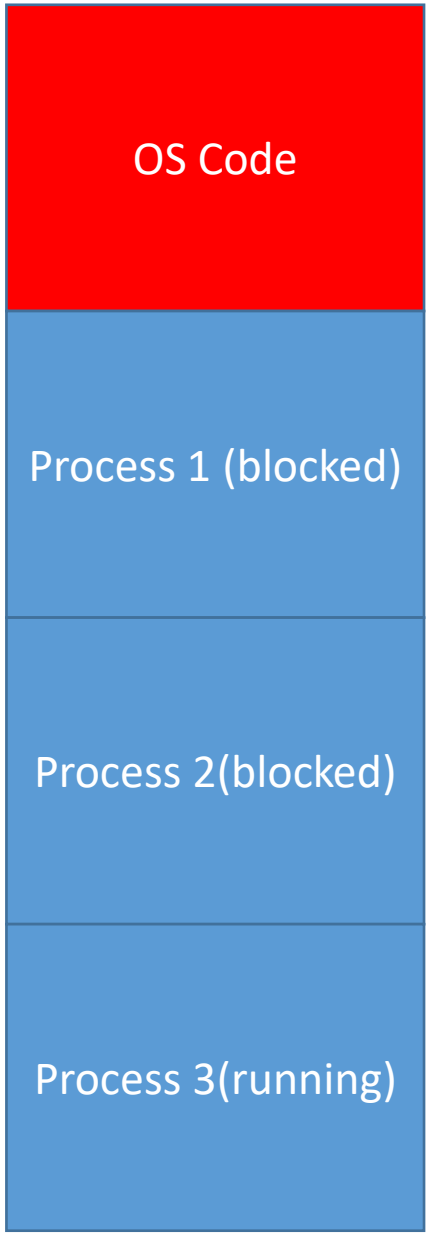
- But multiple processes, finite memory -> efficient memory management required.
- A situation may occur where all the processes are blocked



Memory Management

- If multiple programs were not being run simultaneously.
- And there was a lot of memory space.
- We could reserve just one half of the memory for OS and the other half for the process.

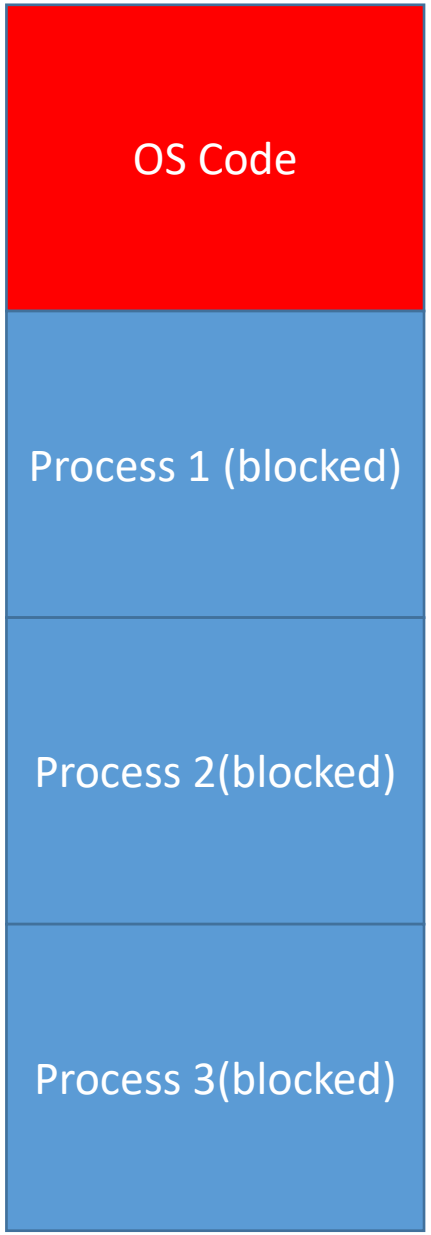
- But multiple processes, finite memory -> efficient memory management required.
- A situation may occur where all the processes are blocked



Memory Management

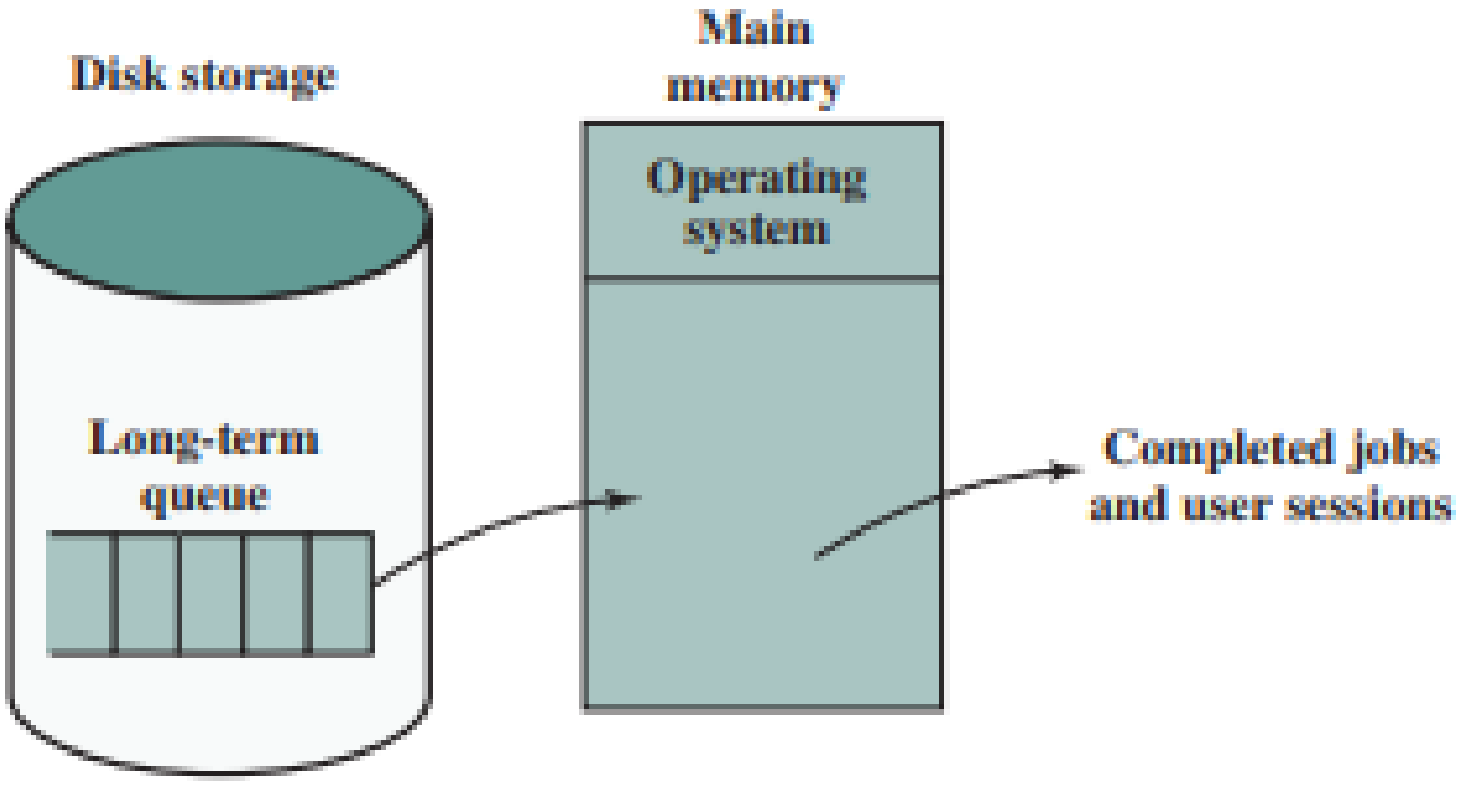
- If multiple programs were not being run simultaneously.
- And there was a lot of memory space.
- We could reserve just one half of the memory for OS and the other half for the process.

- But multiple processes, finite memory -> efficient memory management required.
- A situation may occur where all the processes are blocked



3.2 Swapping

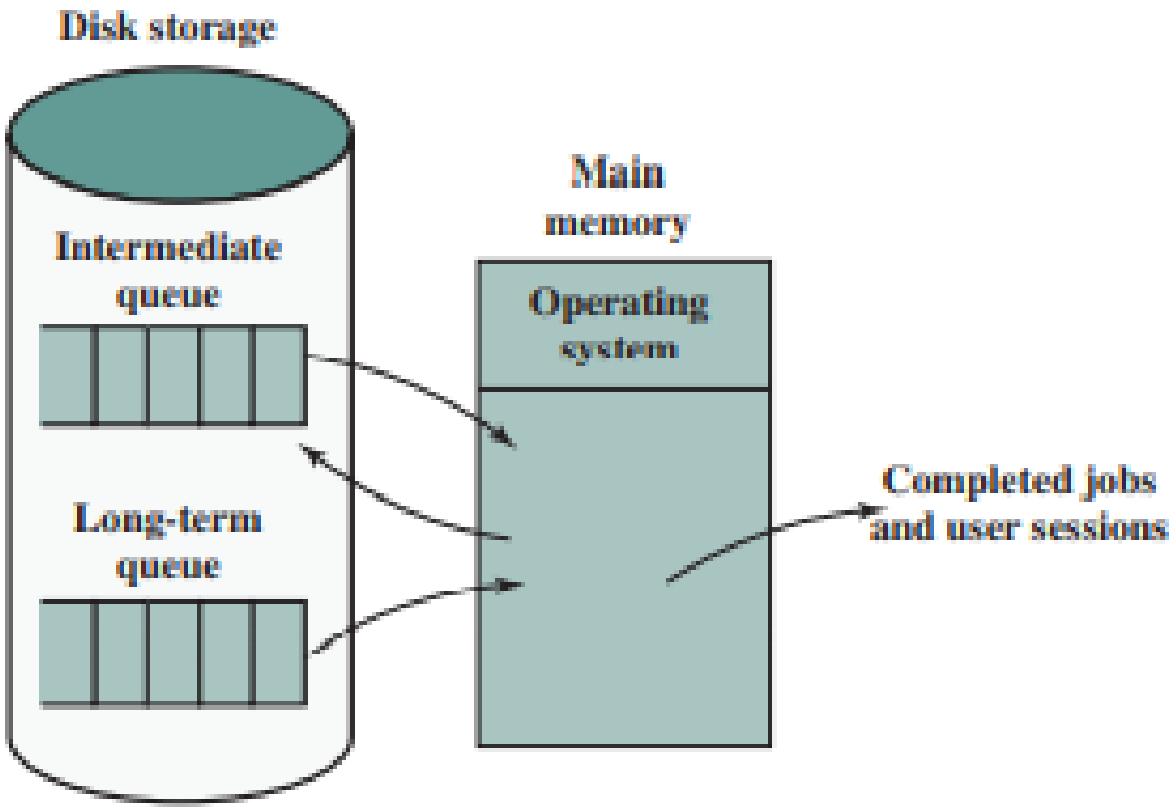
Solution: Swapping



(a) Simple job scheduling

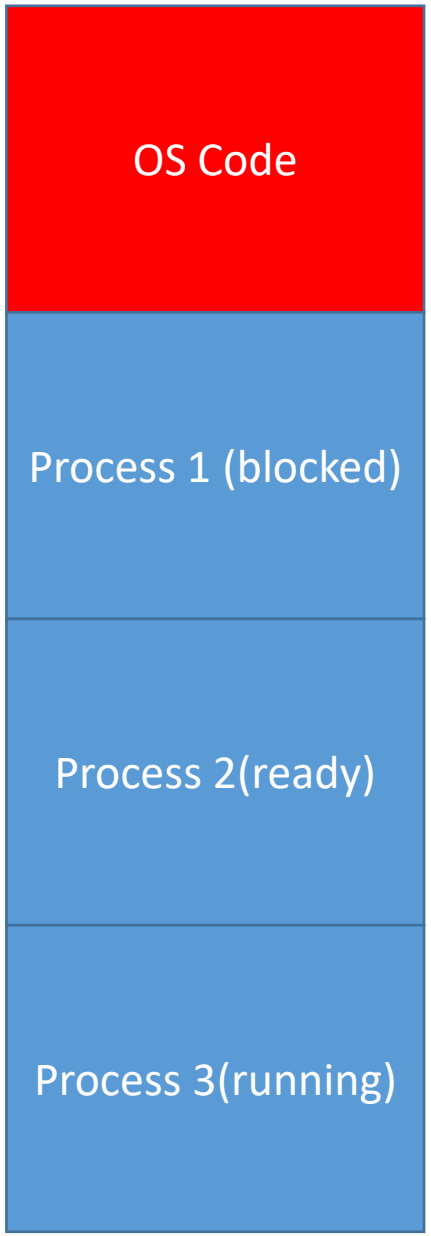
WE ARE AQUAINTED WITH THIS SCENARIO

Solution: Swapping (Cont.)

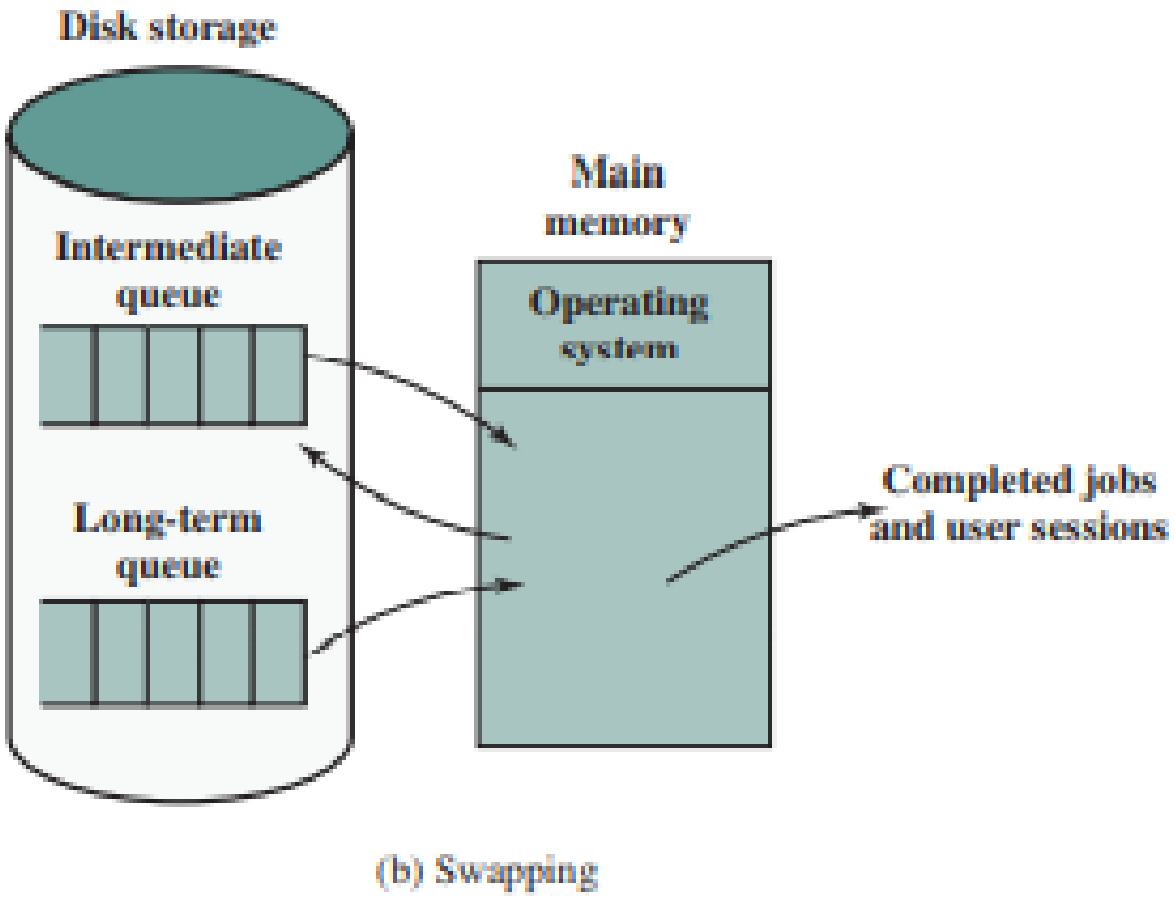


(b) Swapping

Figure 8.12 The Use of Swapping



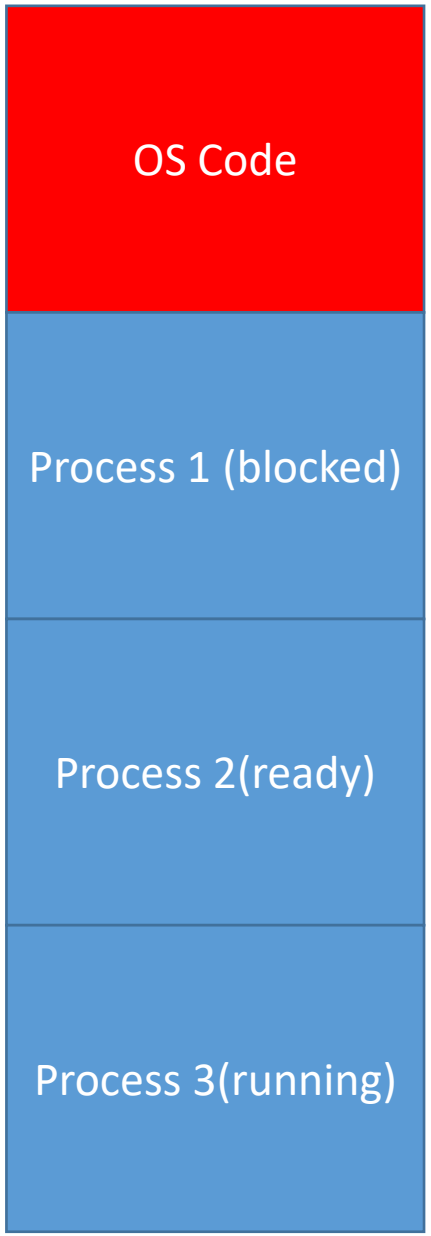
Solution: Swapping (Cont.)



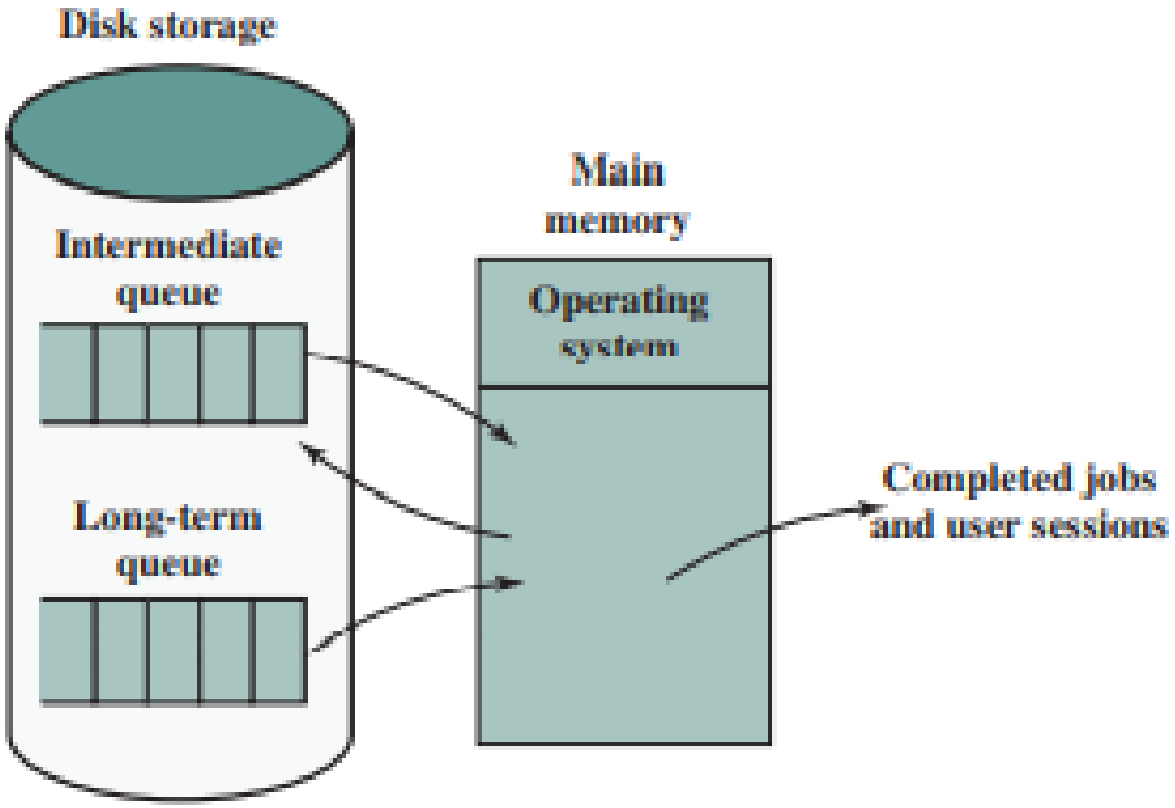
(b) Swapping

Figure 8.12 The Use of Swapping

This process is
Occupying space.
And will be **idle** for
A **long time** compared
To the non-blocked
Processes.



Solution: Swapping (Cont.)

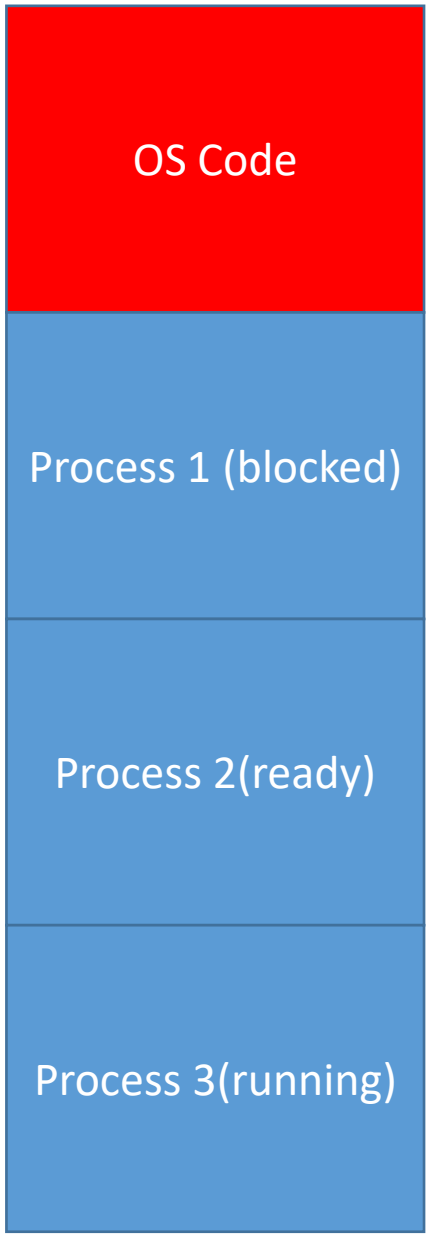


(b) Swapping

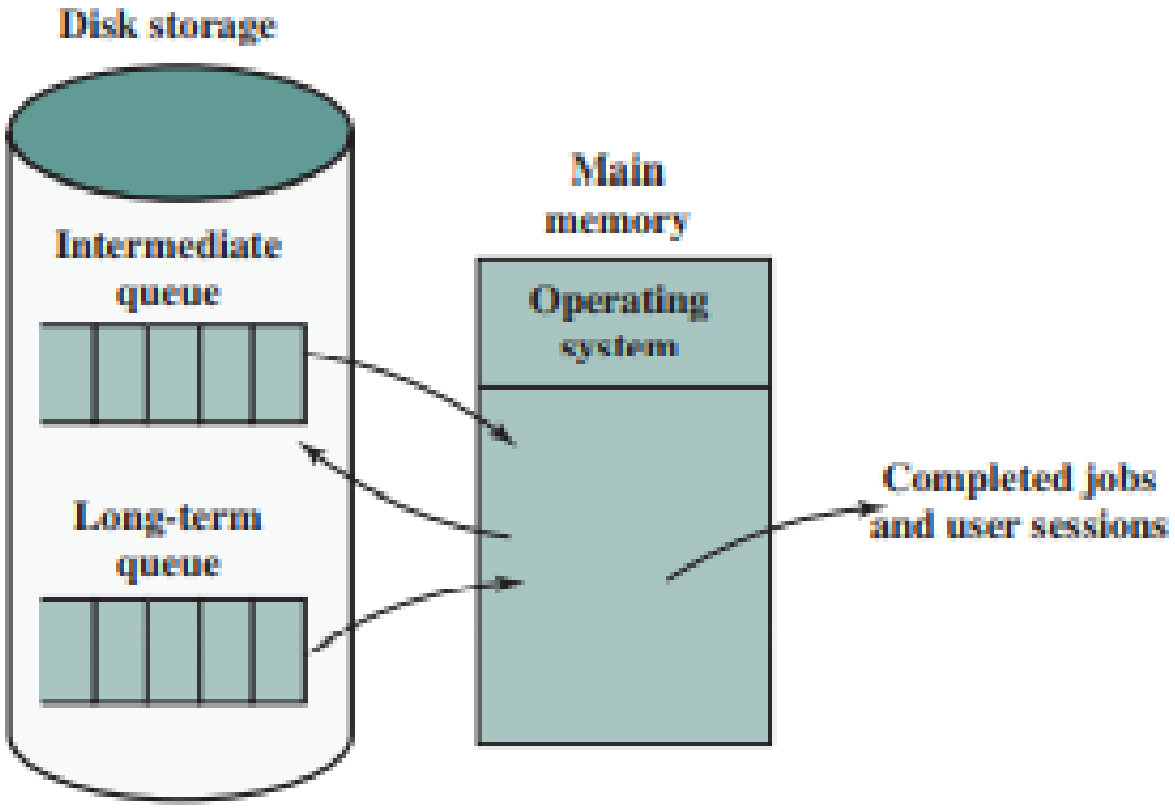
Figure 8.12 The Use of Swapping

This process is
Occupying space.
And will be idle for
A long time compared
To the non-blocked
Processes.

We can insert
Another process
In place of this one
Using the addresses in
The **long term queue**



Solution: Swapping (Cont.)

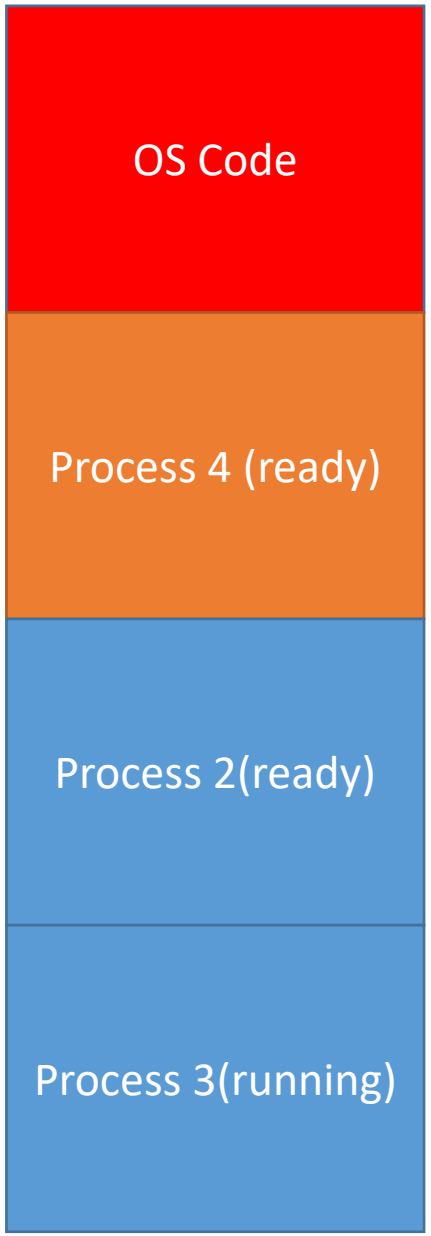


(b) Swapping

Figure 8.12 The Use of Swapping

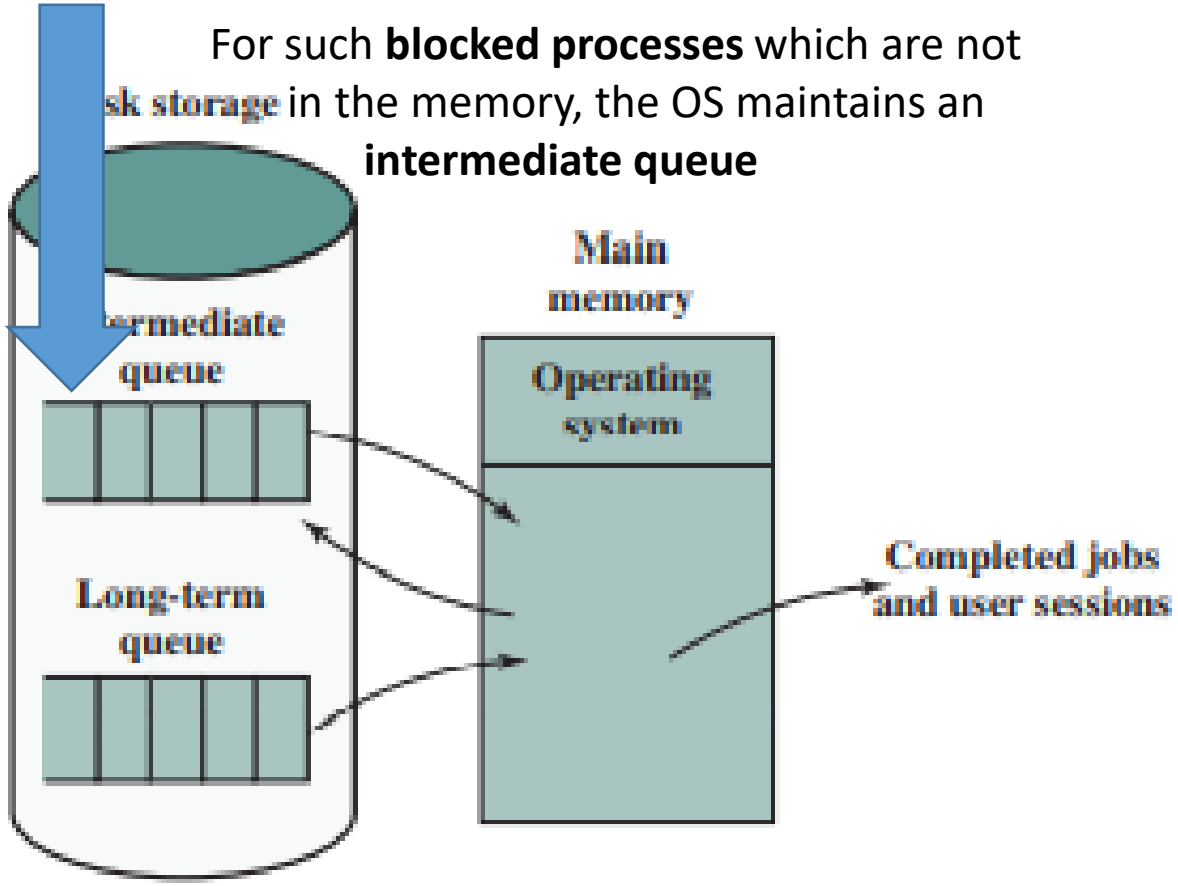
This process is
Occupying space.
And will be idle for
A long time compared
To the non-blocked
Processes.

We can insert
Another process
In place of this one
Using the addresses in
The long term queue



Solution: Swapping (Cont.)

For such **blocked processes** which are not in the memory, the OS maintains an **intermediate queue**

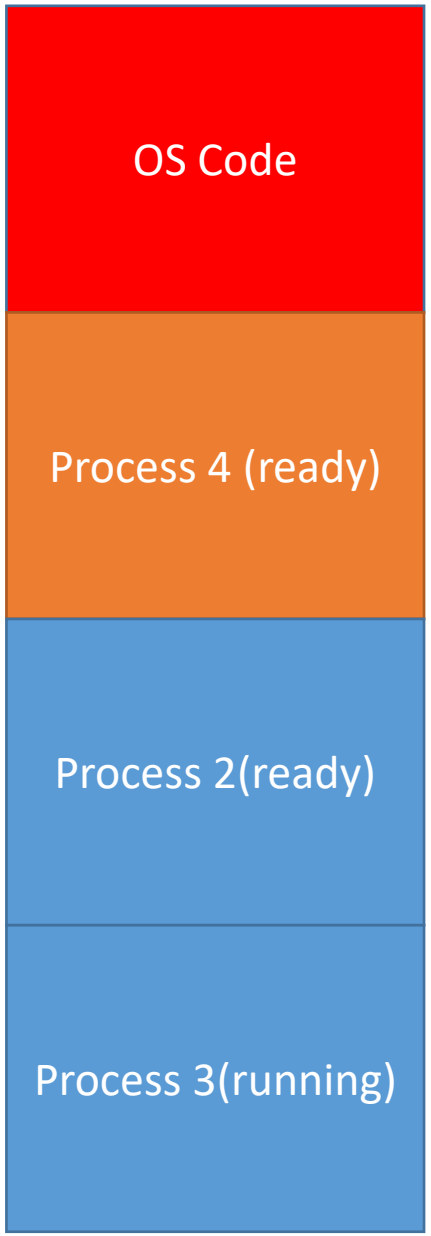


(b) Swapping

Figure 8.12 The Use of Swapping

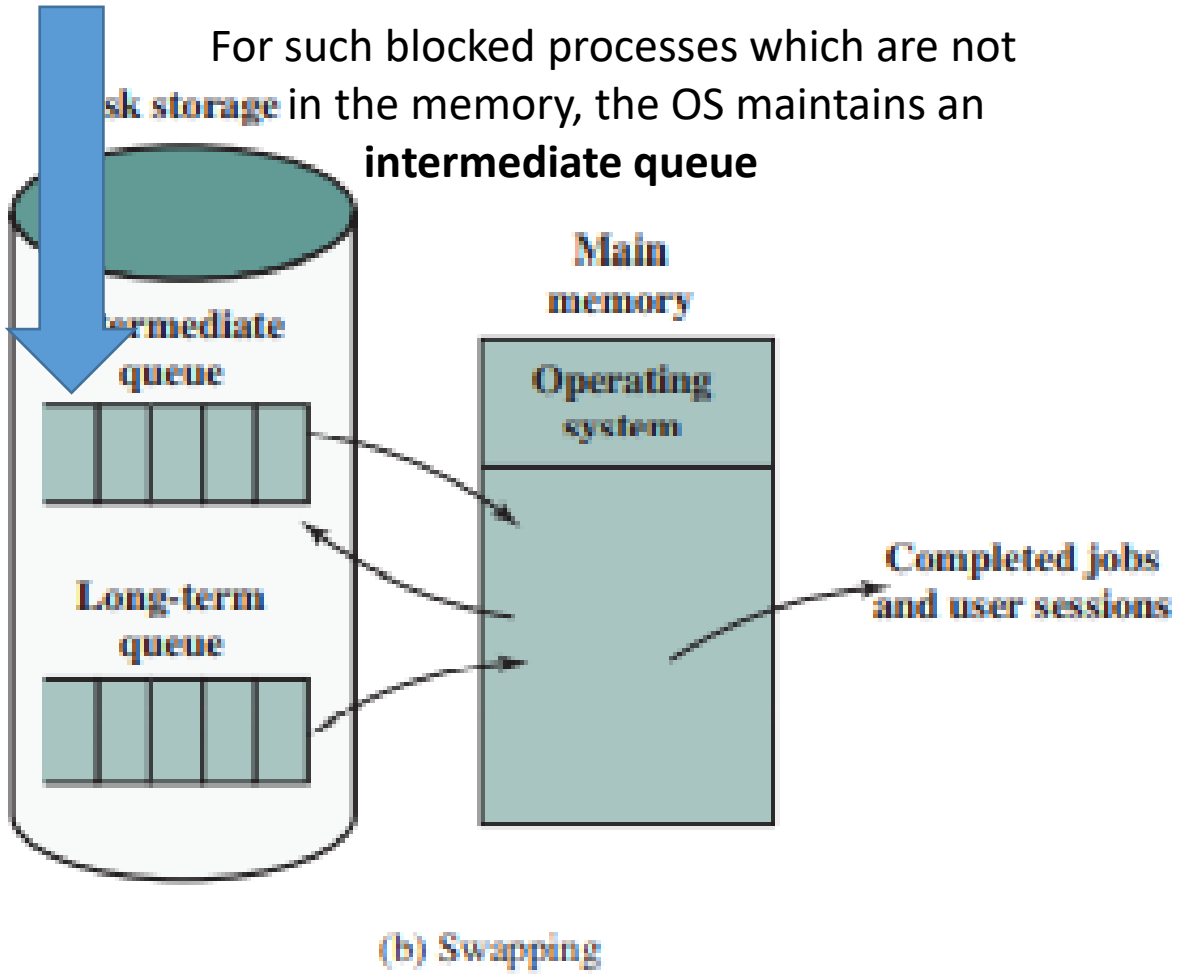
This process is
Occupying space.
And will be idle for
A long time compared
To the non-blocked
Processes.

We can insert
Another process
In place of this one
Using the addresses in
The long term queue



Solution: Swapping (Cont.)

For such blocked processes which are not
sk storage in the memory, the OS maintains an
intermediate queue

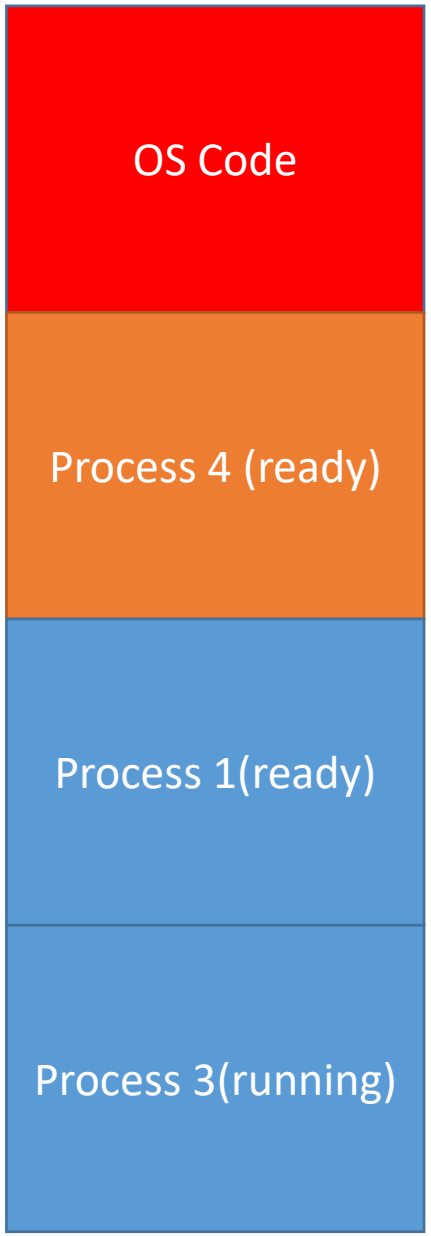


(b) Swapping

Figure 8.12 The Use of Swapping

This process is
Occupying space.
And will be idle for
A long time compared
To the non-blocked
Processes.

We can insert
Another process
In place of this one
Using the addresses in
The long term queue



Lets
Assume
Process
2
ended

Later on, If the OS's scheduler finds
the process to be **unblockable**, then it will
simply load its code into the memory.
Remove its info from the **intmdt** queue and mark it as **ready**.

- Swapping is I/O operation so there is a chance of making things worse
- But disk I/O is comparatively faster than other I/O (printer or tape etc.)
- Usually improve performance

3.3 Partitioning

Partitioning

- We have understood swapping.
- Basically it is referring to swapping the blocked process codes in the RAM with the new process codes in the disc.
- (Though it is not an actual swap as no write operation is done in the DISC)
- However, every time the OS tries to bring in a **new process**, it will have to be **careful** about whether or not its code is becoming **overlapped** with another process's code.
- There is **no organized approach** for this so far.
- Hence we introduce the concept of Partitioning .

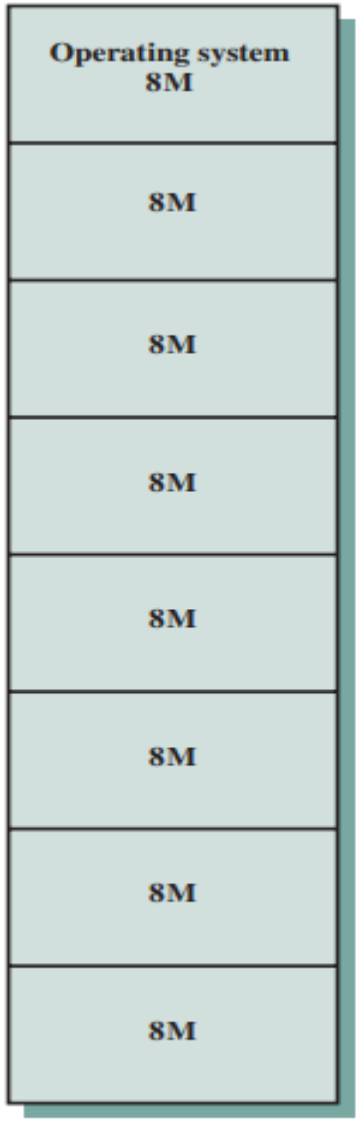
Partitioning

- Equal sized partitions
- Variable sized partitions

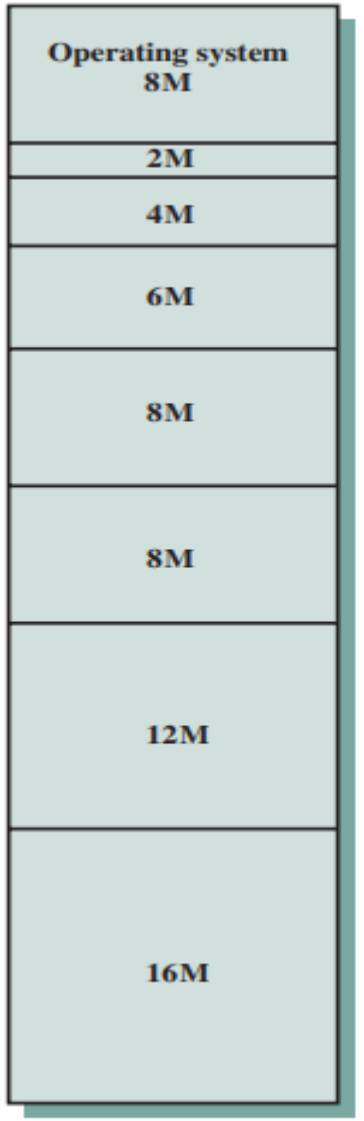
Now it is not **required to check**
Other processes,

The OS just needs to keep info about
Whether a **partition is filled up or not**.

Then while placing a new process, it
Just has to place the **codes** in the
Unoccupied partitions.



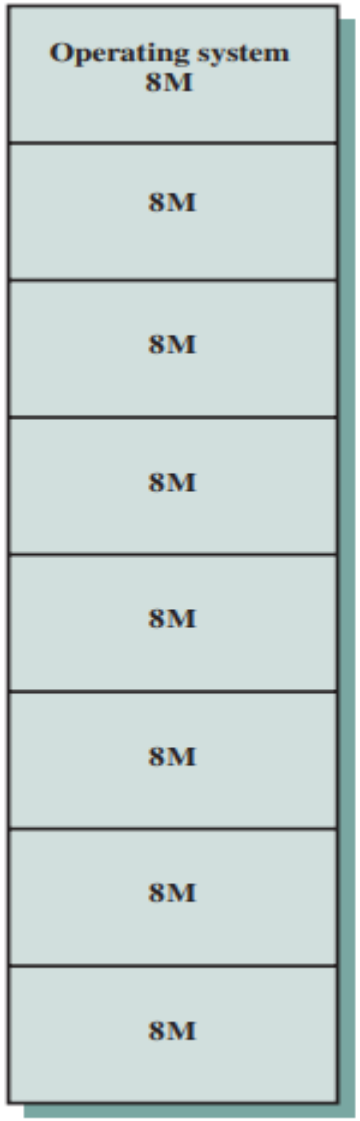
(a) Equal-size partitions



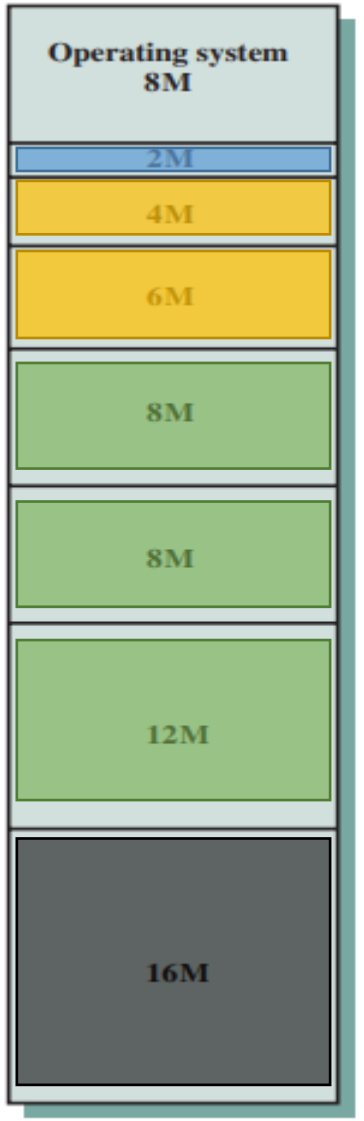
(b) Unequal-size partitions

Partitioning

- Equal sized partitions
- Variable sized partitions



(a) Equal-size partitions



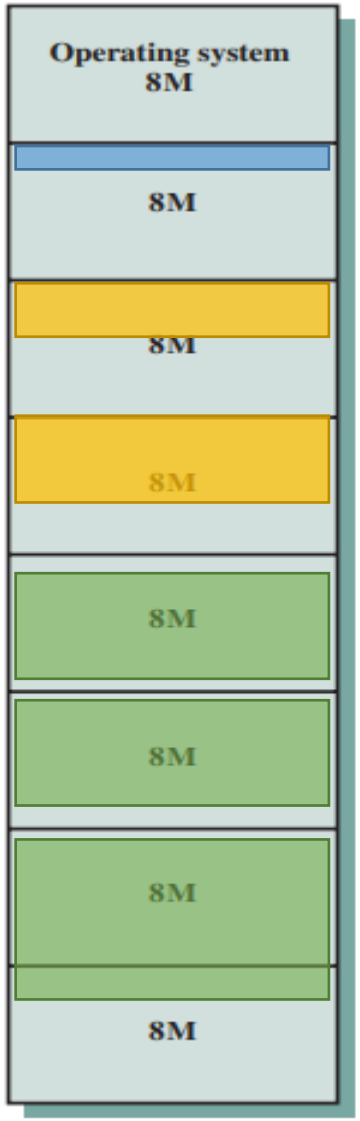
(b) Unequal-size partitions

Partitioning

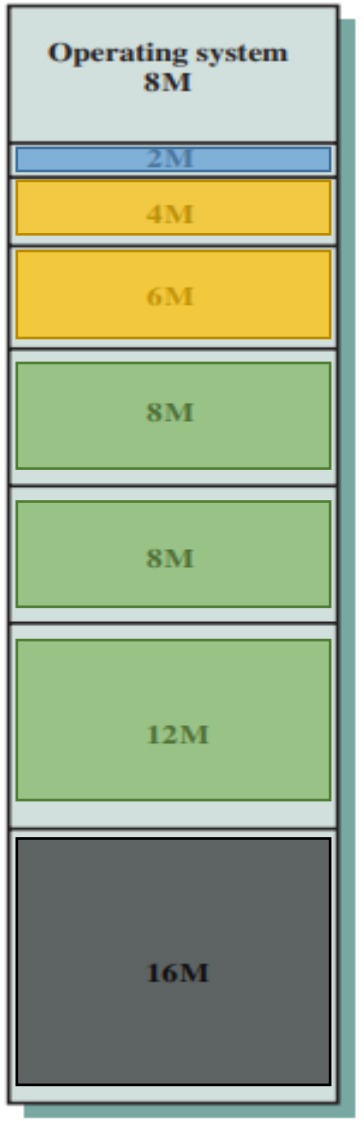
- Equal sized partitions
- Variable sized partitions

16 M

Variable sized partitions are **better**



(a) Equal-size partitions

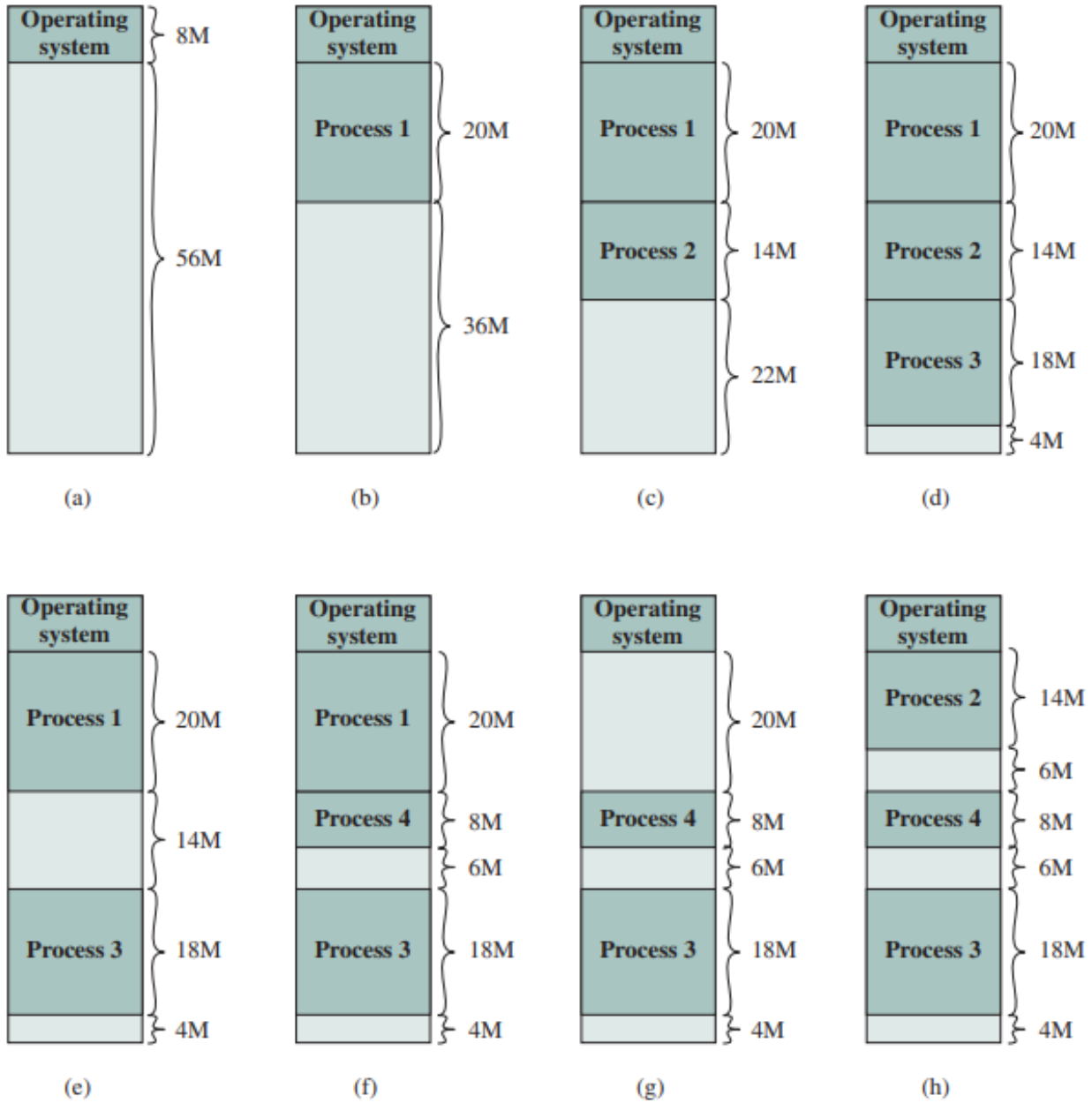


(b) Unequal-size partitions

Partitioning

- Issues with **variable sized partitions**

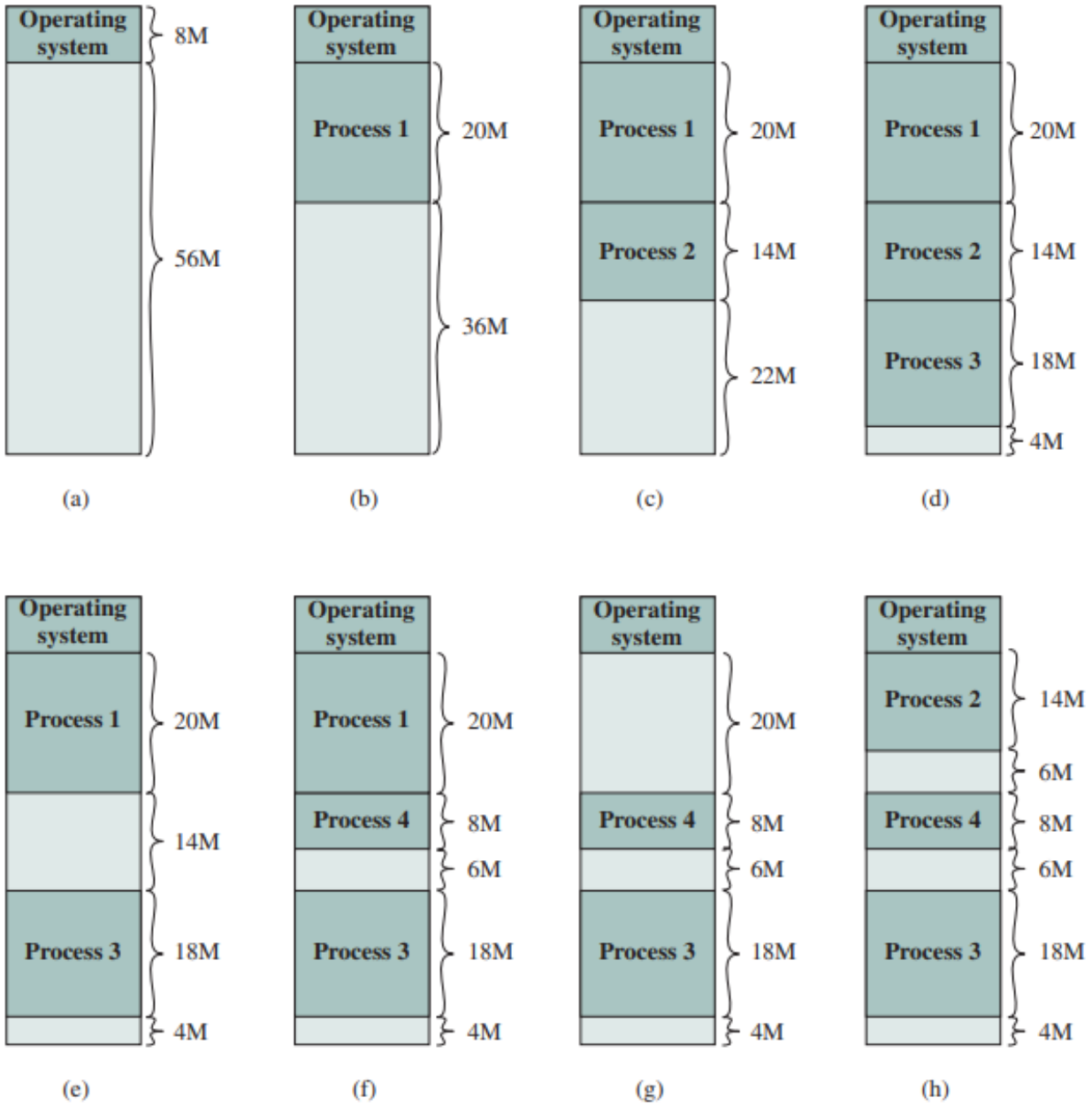
- **FRAGMENTATION**
- **Leads to small holes in memory**
- **Compaction is required but time consuming**



Partitioning

- **SOLUTION**:

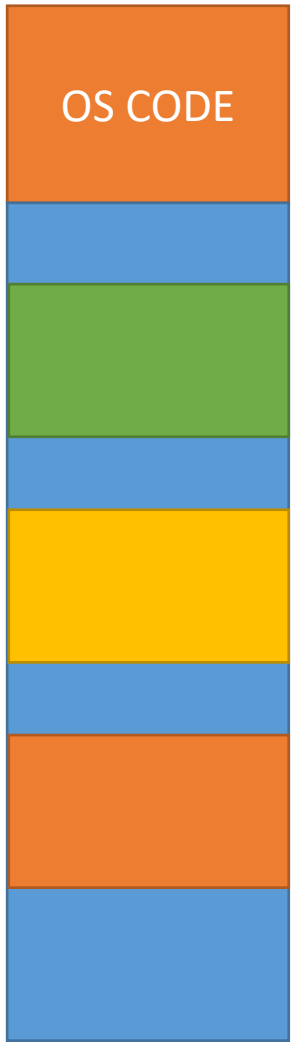
Dynamic variable partitions with **compaction**



Partitioning

- SOLUTION:

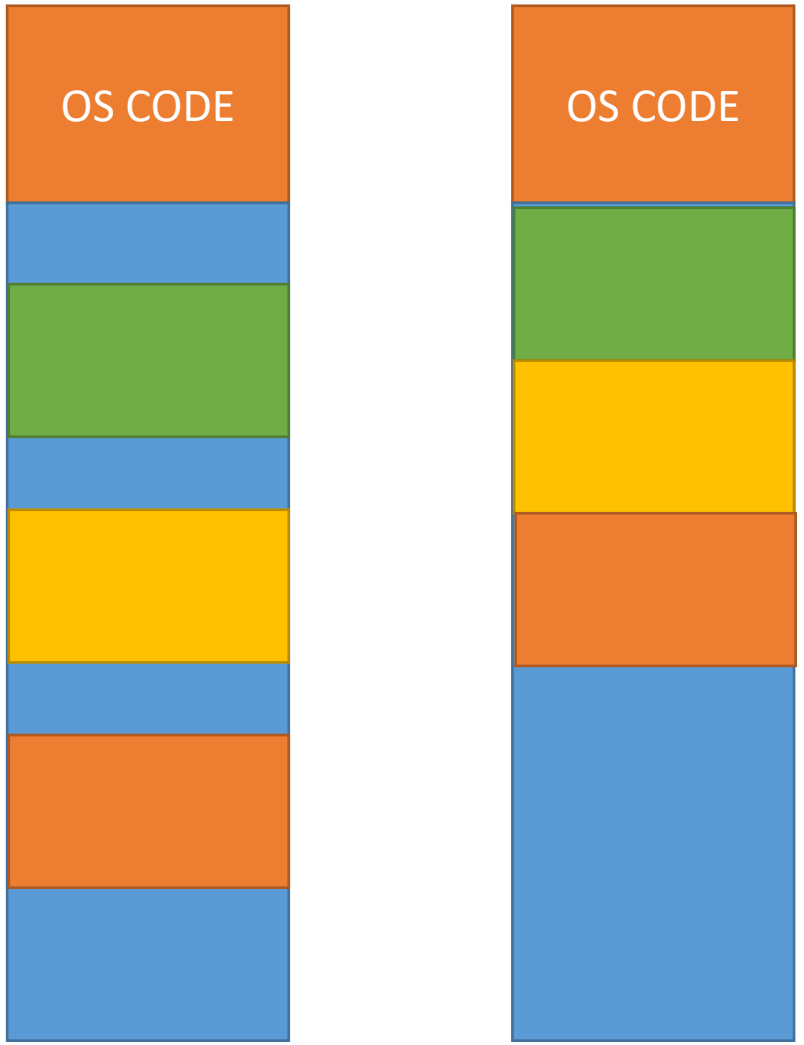
Dynamic variable partitions with **compaction**



Partitioning

- SOLUTION:

Dynamic variable partitions with **compaction**

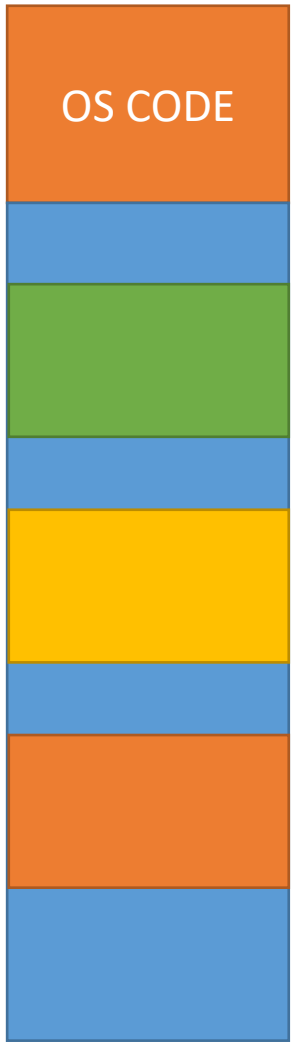


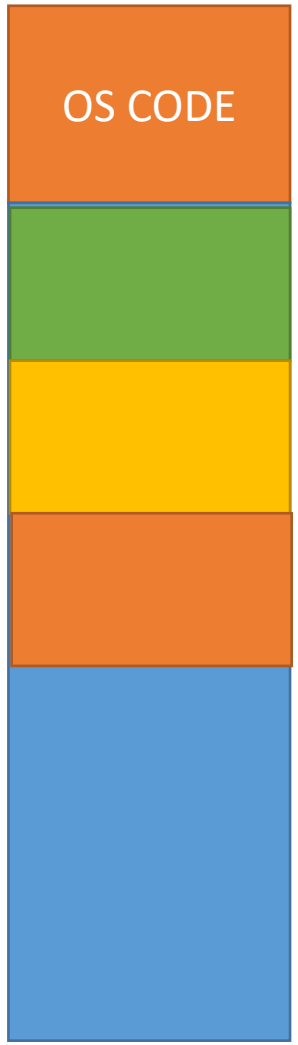
Partitioning

- SOLUTION:

Dynamic variable partitions with **compaction**

Compaction is requiring several memory read write operations





Note that :
We need to use **Displacement addressing** Here for the process codes as the physical Processes can be placed at any part of the memory.

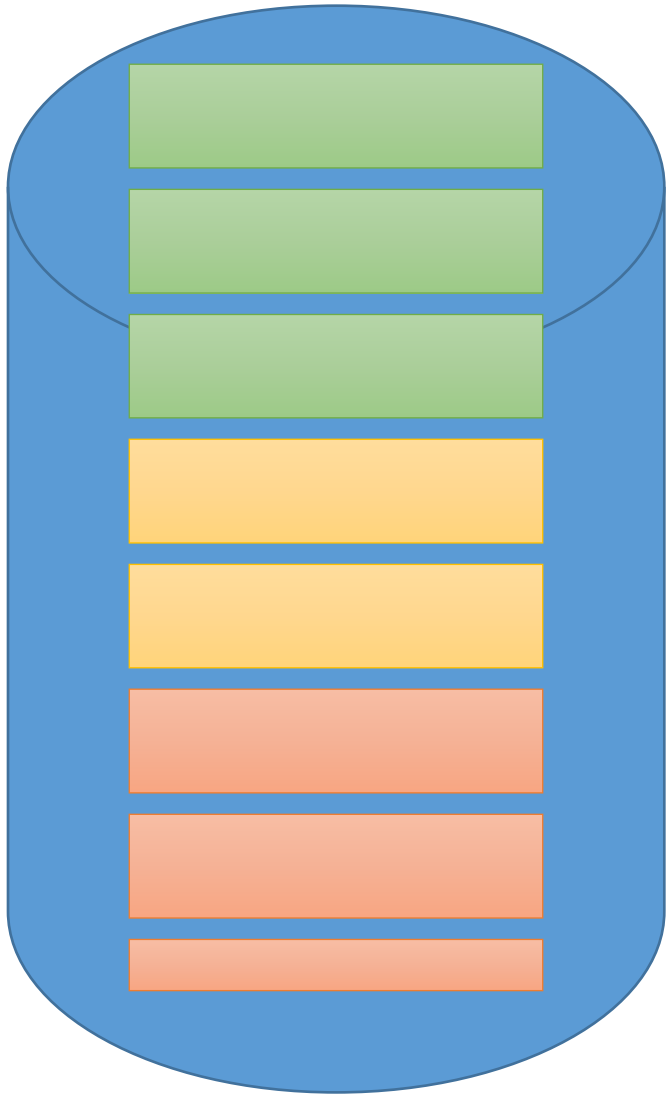
3.4 Paging and Page Tables

A more efficient approach : Paging

- Both of the fixed and unfixed **partitioning** mechanisms are **inefficient**. May result in waste of a lot of space.
- Compaction requires several **memory read write operations**.

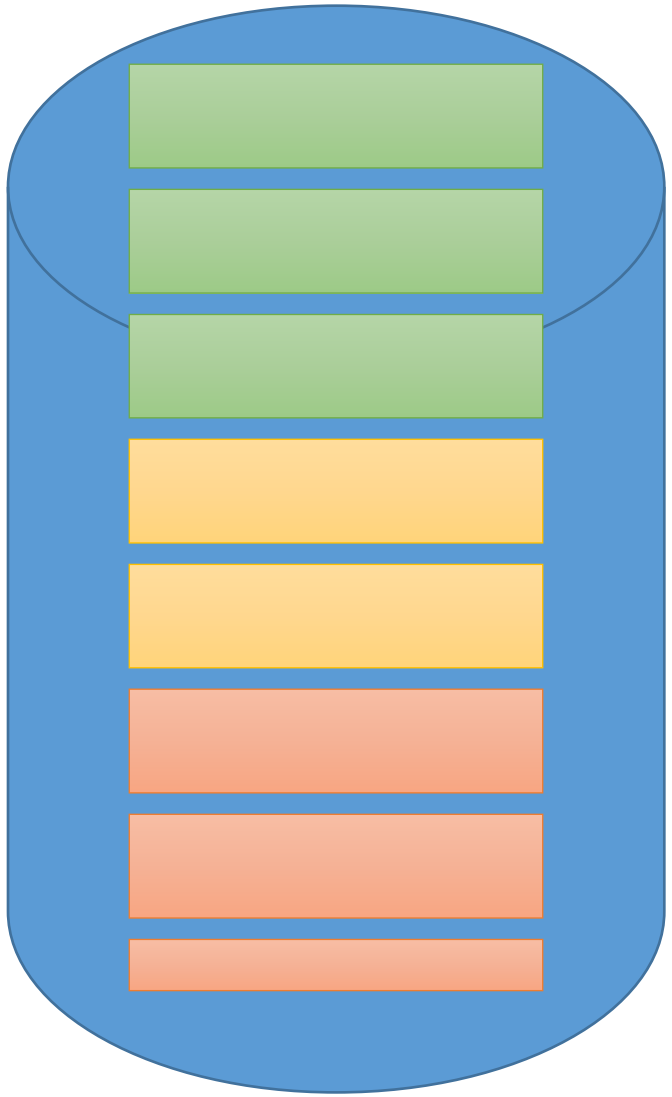
- Thus we try another mechanism called **Paging**.

A more efficient approach : Paging



Programs with **fixed sized pages**

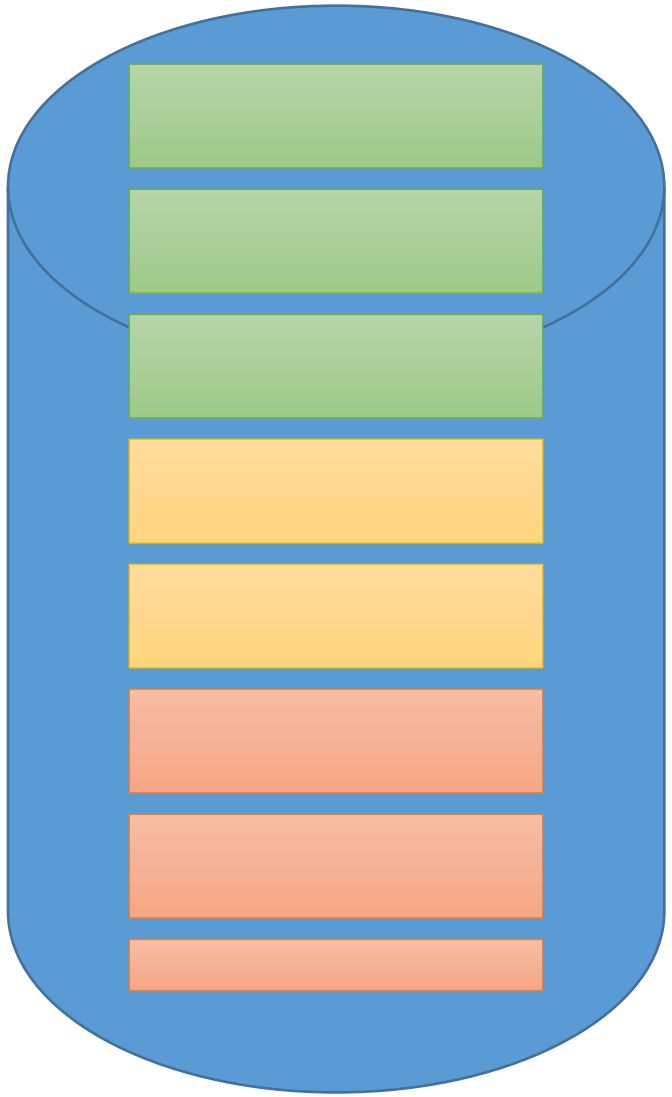
A more efficient approach : Paging

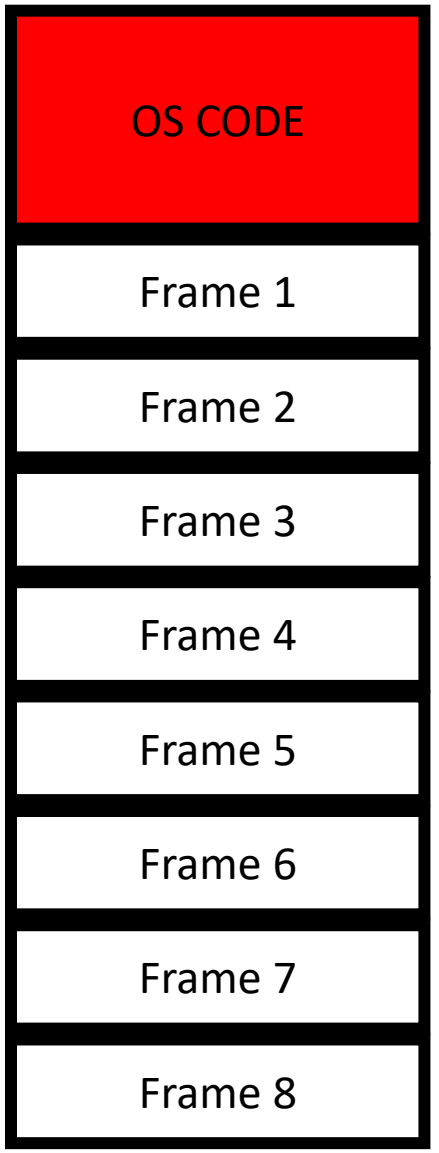


Programs with fixed sized pages

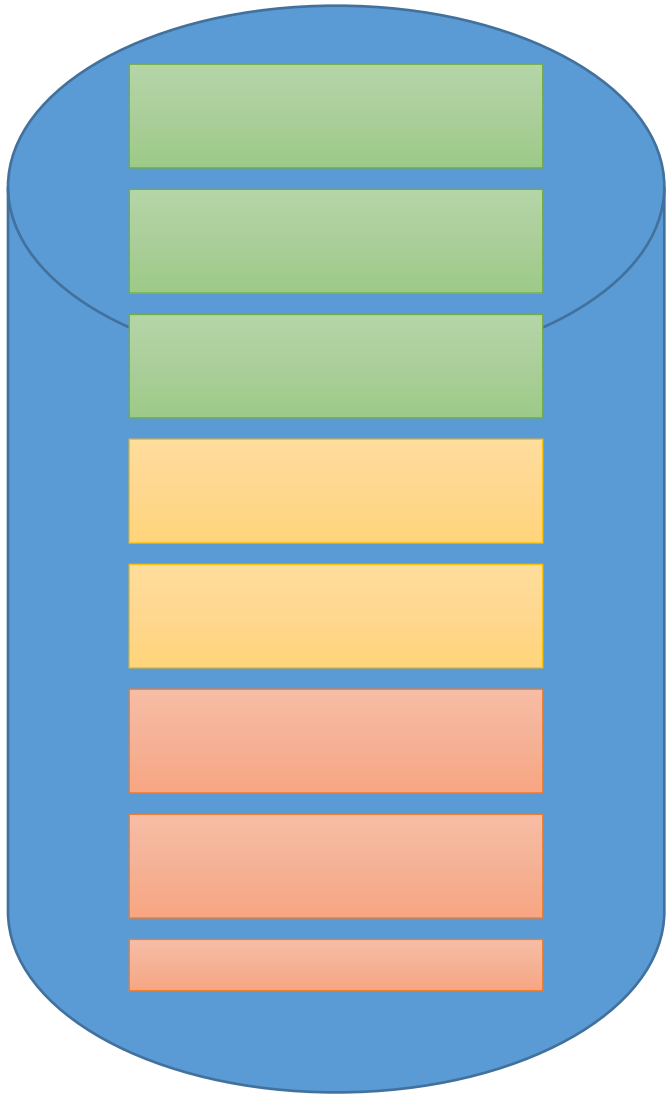
Note that some programs may have **Partial page** sizes.

A more efficient approach : Paging



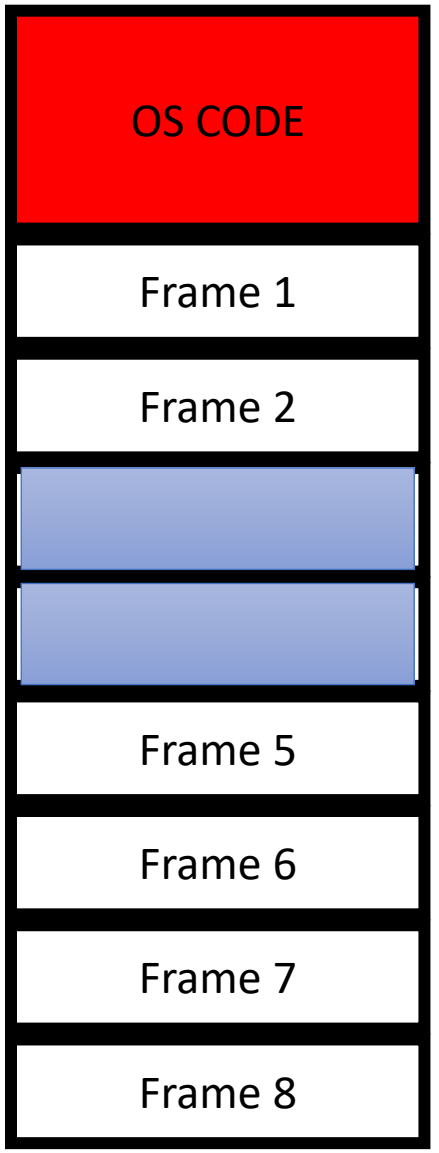


A more efficient approach : Paging

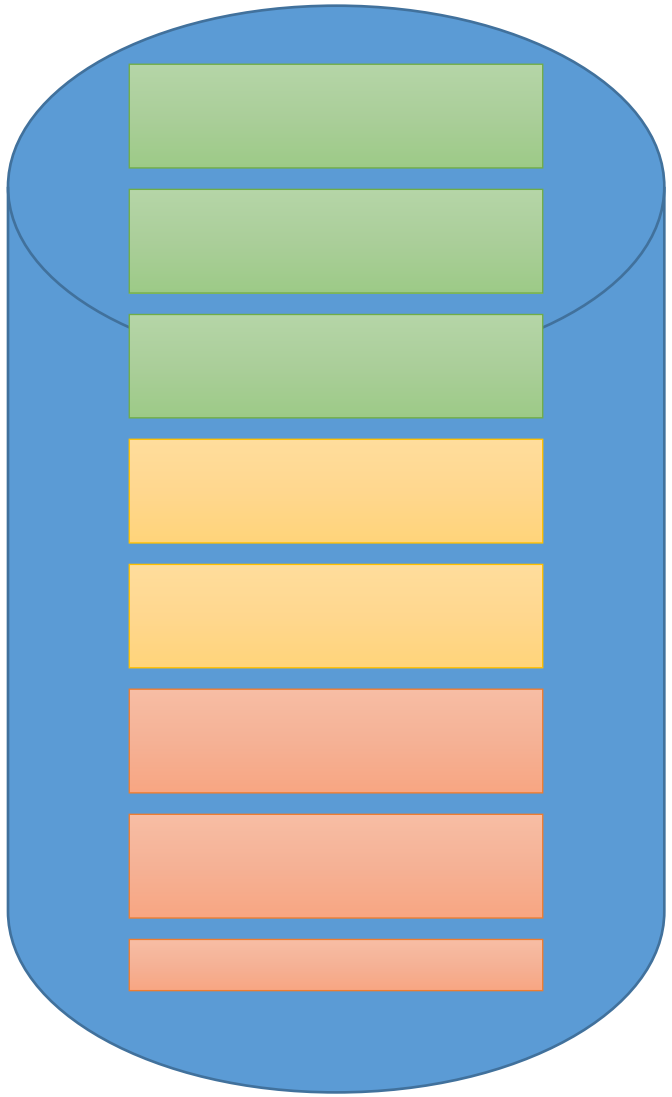


Avail able chunks of memory referred to as frames

Assume that This program With **two pages** Was already there

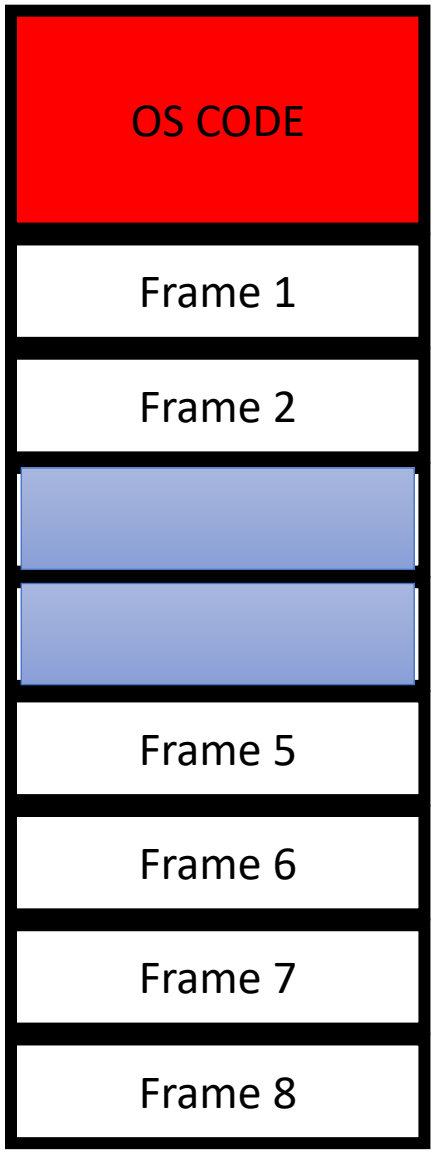


A more efficient approach : Paging

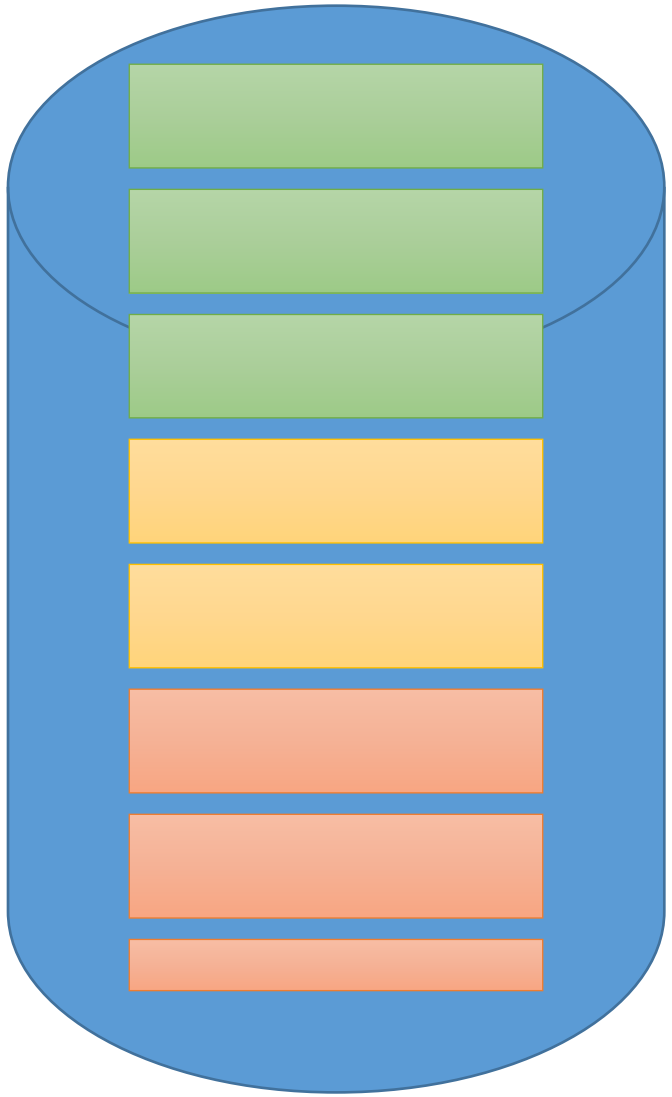


We want to **bring** in
This program

Assume that
This program
With two pages
Was already there

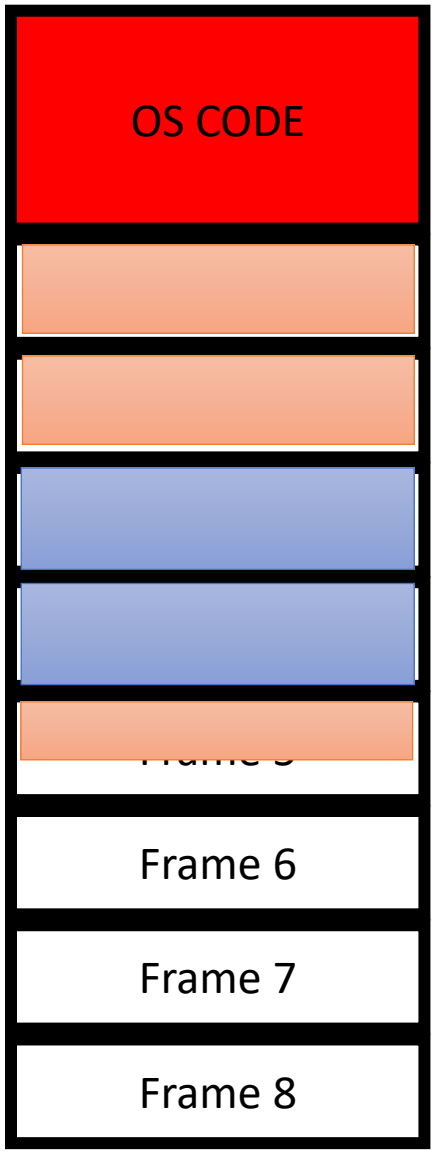


A more efficient approach : Paging

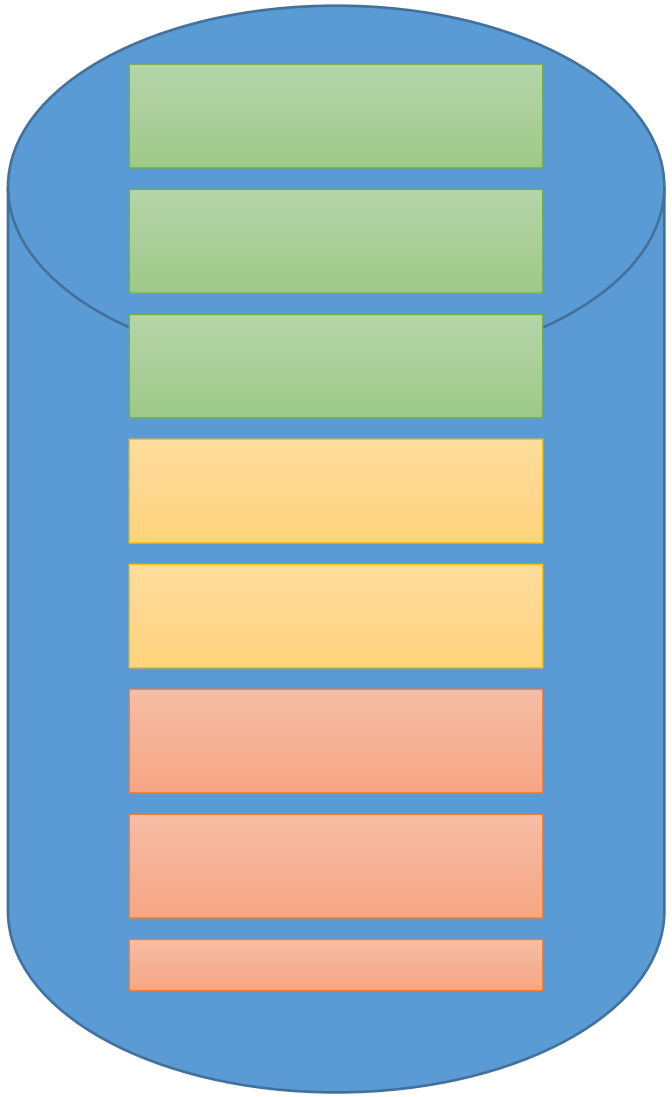


We want to bring in
This program

Assume that
This program
With two pages
Was already there

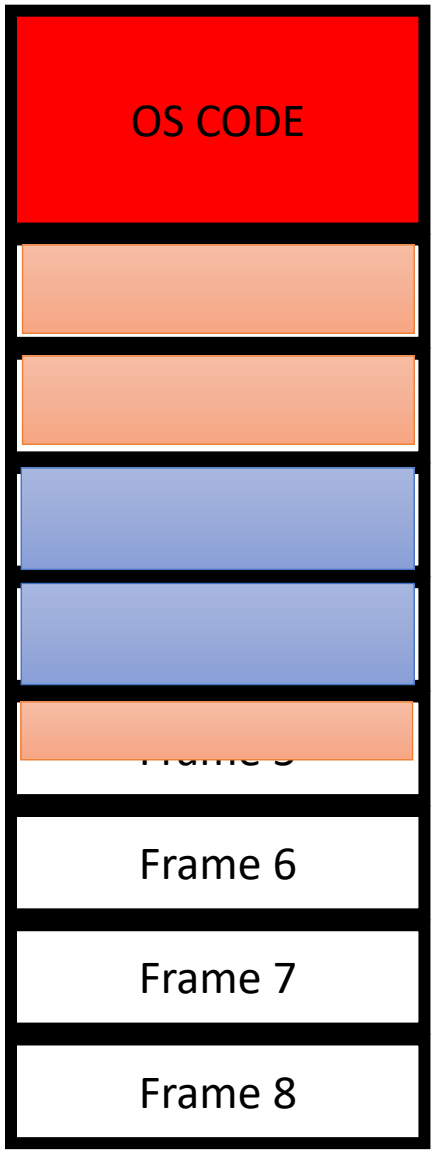


A more efficient approach : Paging



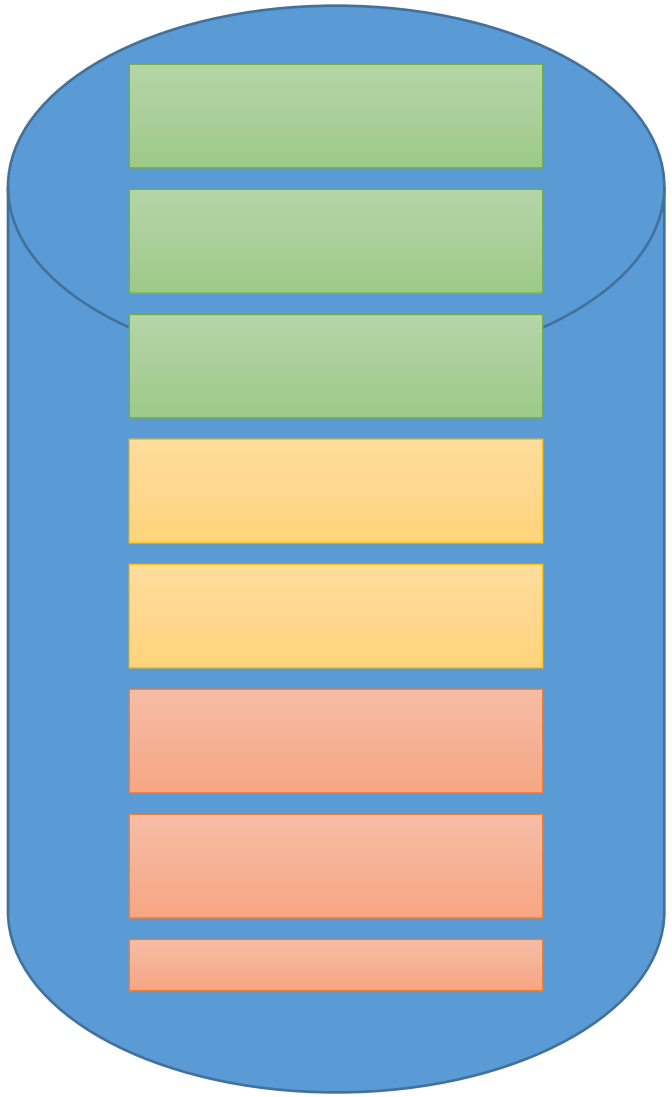
We want to bring in
This program

Assume that
This program
With two pages
Was already there



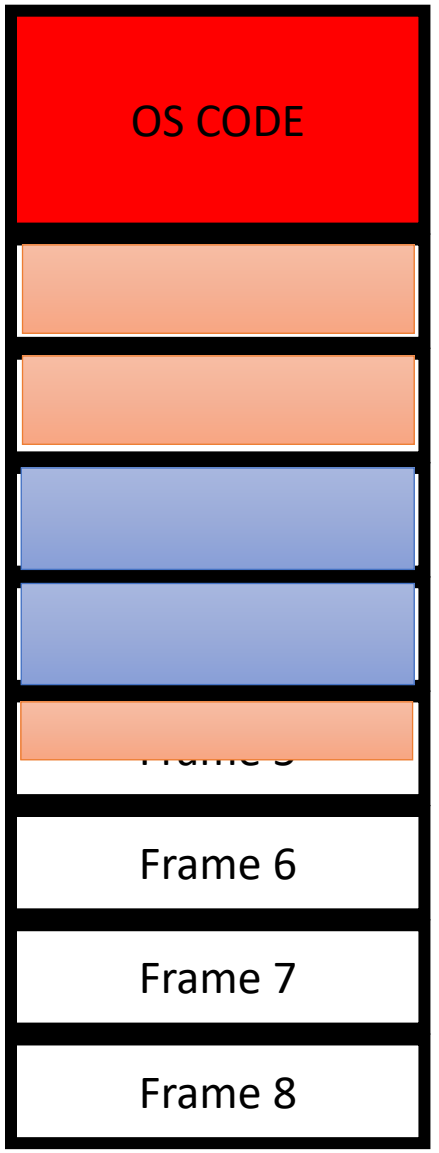
Very **little wastage**

A more efficient approach : Paging



We want to bring in
This program

Assume that
This program
With two pages
Was already there



But observe that
The code is no longer
Present in a
Sequential manner

Very little wastage

Page table

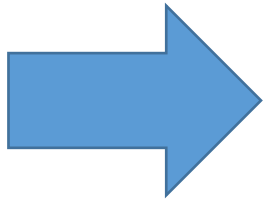
- A **mapping** is required to be maintained.
- For keeping track :
  - Which **page** of a program is in which **frame**
  - We need to use a **new kind** of **addressing** mode.
  - Now when a processor gets an instruction with an address it will be in the form **(page no, relative address)**
  - it will change it into the format **(frame, relative address)** using the page table.
- For **each process,** OS will maintain a **page table**.

Example

```
CMP AX,BX
JGE #100
MOV AX, BX
ADD AX, BX
LOAD AX, #200
....
...
...
...
```

Example

CMP AX,BX
JGE #100
MOV AX, BX
ADD AX, BX
LOAD AX, #200
….
…
…
…



The
Program
Is divided
Into pages

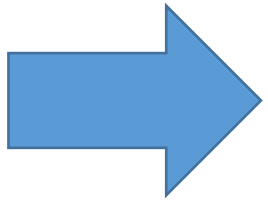
CMP AX,BX
JGE #100
MOV AX, BX
ADD AX, BX
LOAD AX, #200

…
..
…
…
..
…

…
..
…
…
..
…

Example

CMP AX,BX
JGE #100
MOV AX, BX
ADD AX, BX
LOAD AX, #200
….
…
…
…



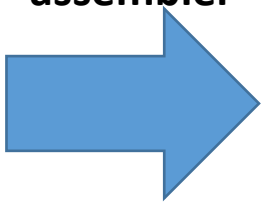
The
Program
Is divided
Into pages

CMP AX,BX
JGE #100
MOV AX, BX
ADD AX, BX
LOAD AX, #200

…
..
…
…
..
…

…
..
…
…
..
…

Observe that The addresses are written In a new format By the assembler

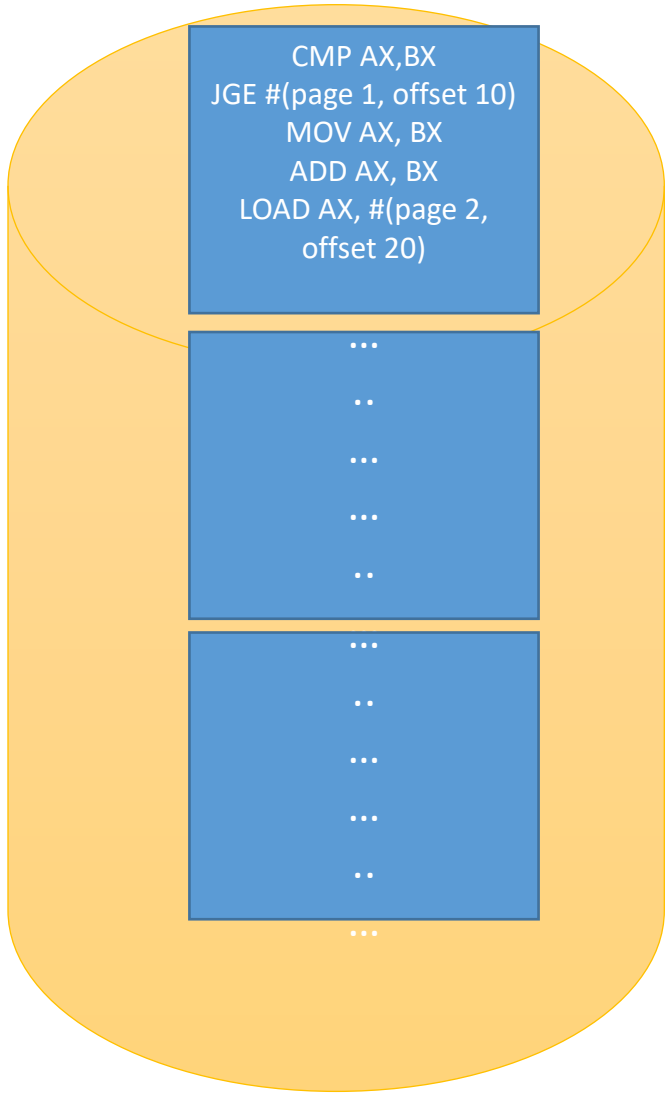


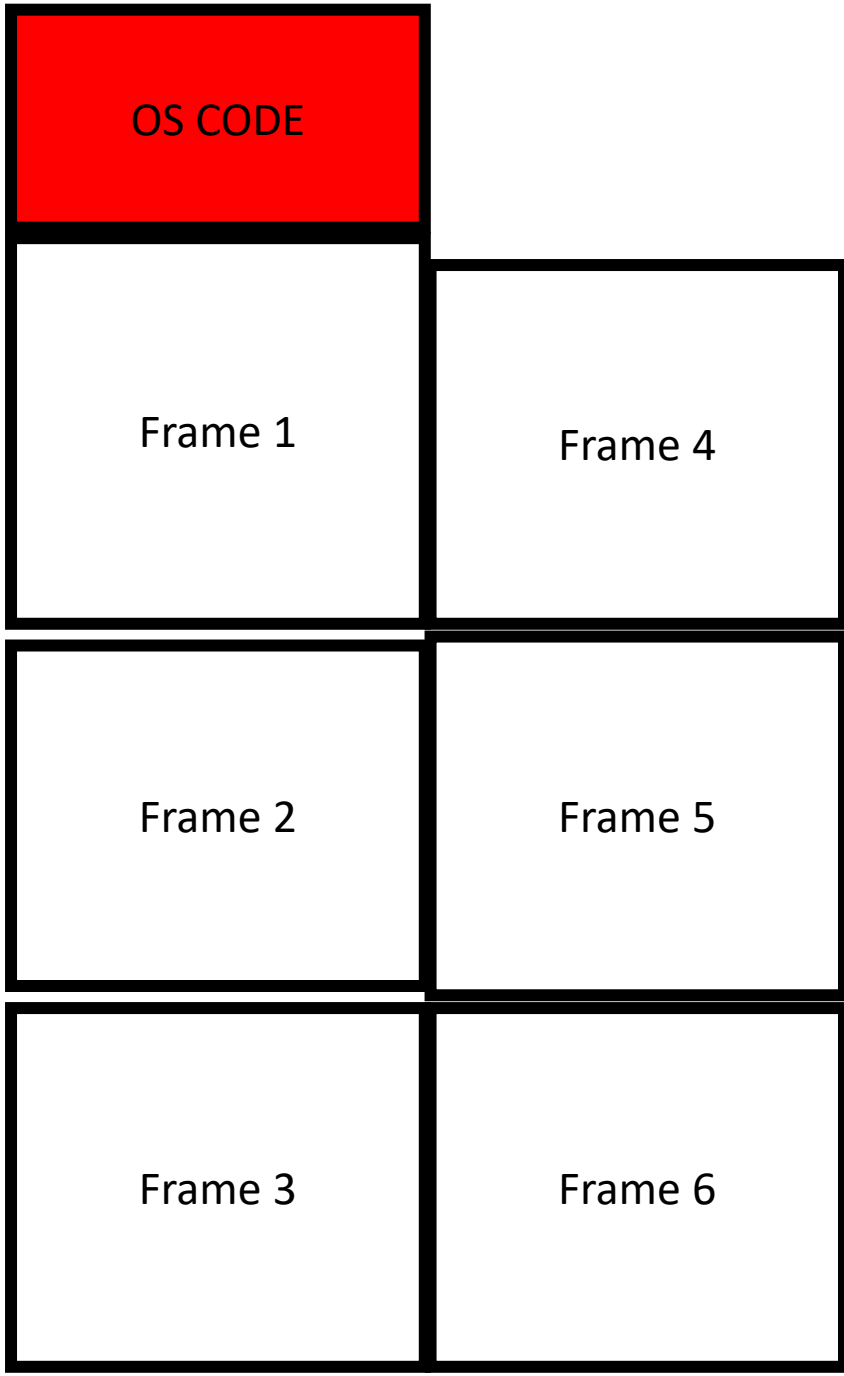
CMP AX,BX
JGE #(page 1, offset 10)
MOV AX, BX
ADD AX, BX
LOAD AX, #(page 2, offset 20)

…
..
…
…
..
…

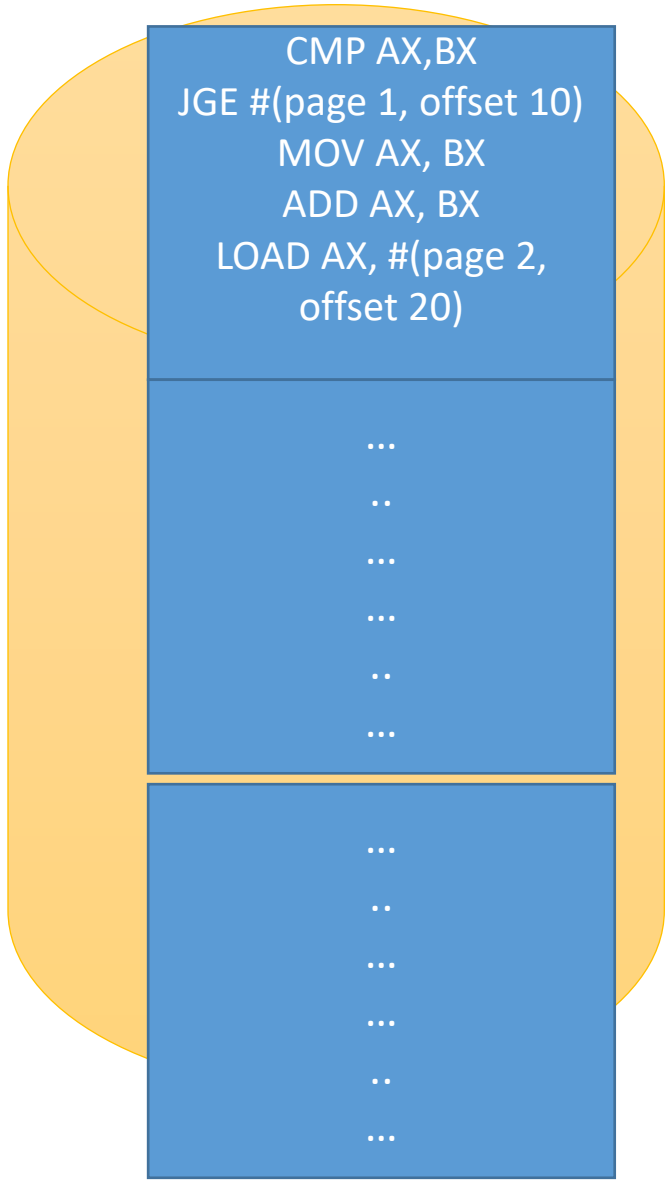
…
..
…
…
..
…

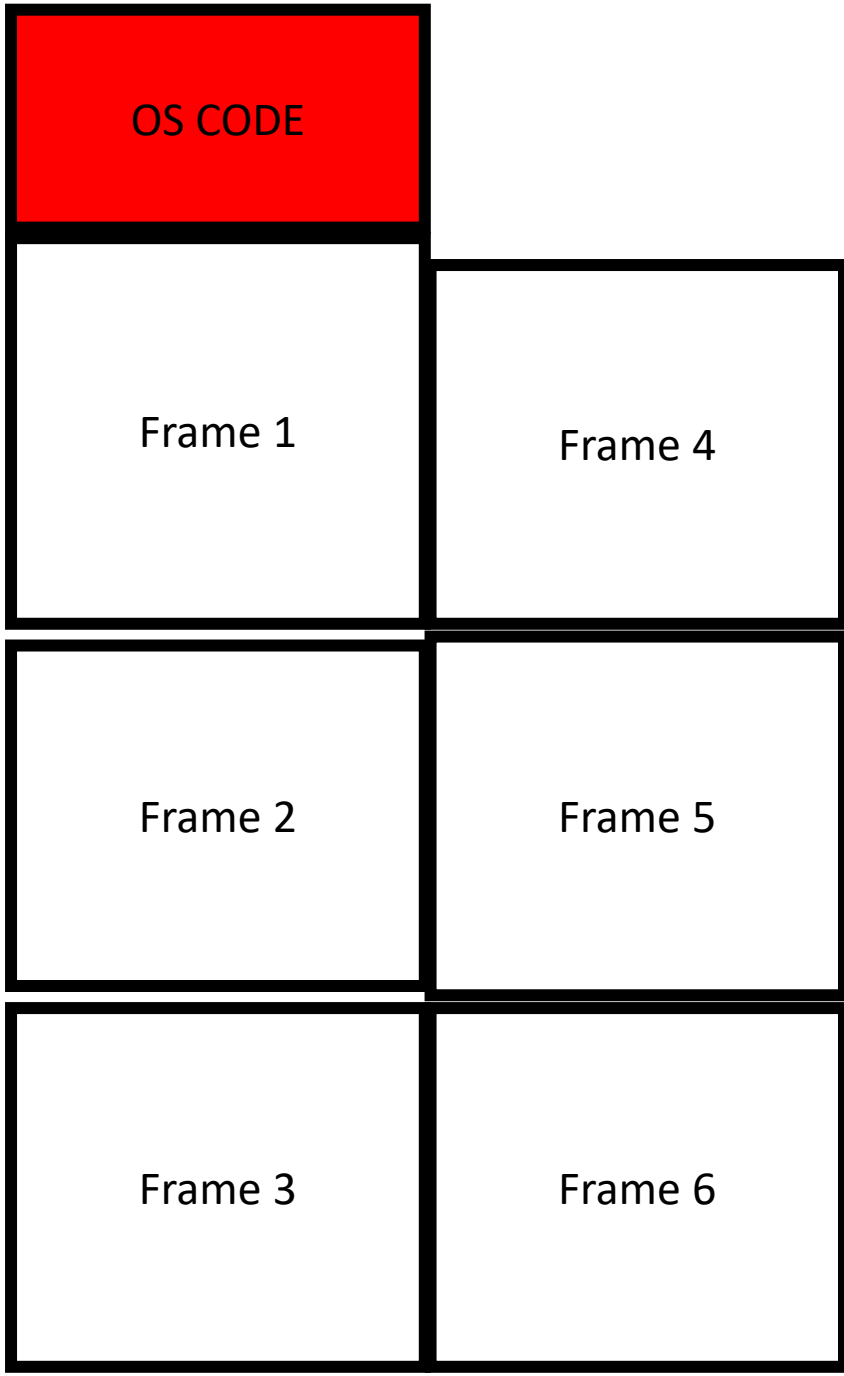
A more efficient approach : Paging



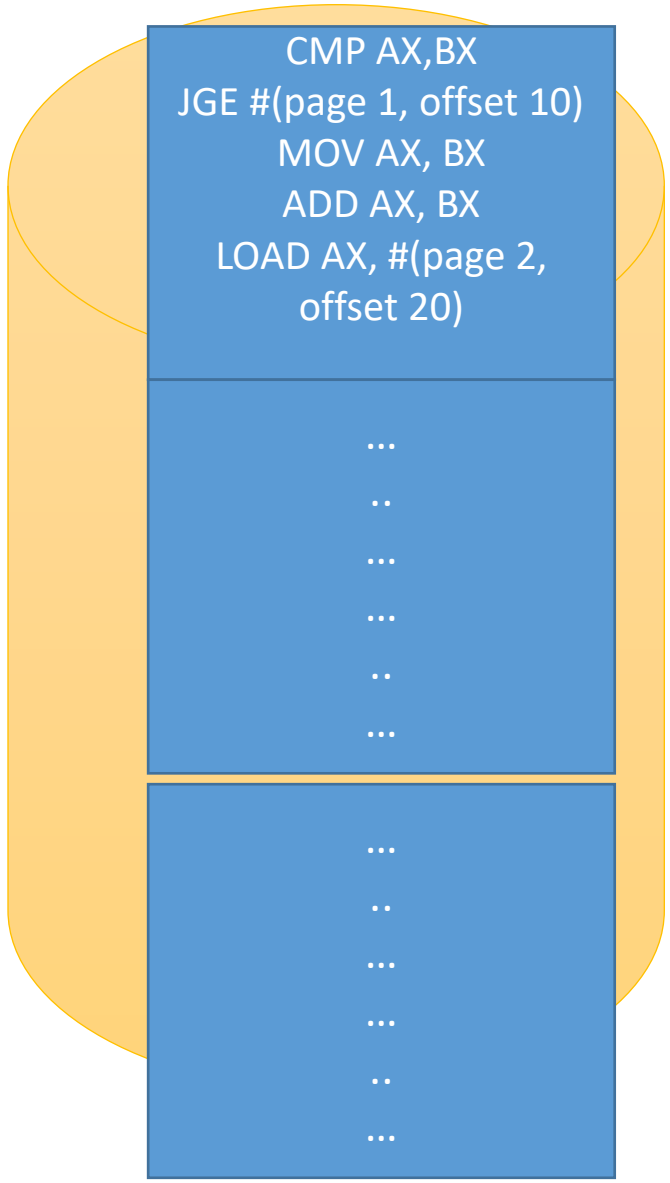


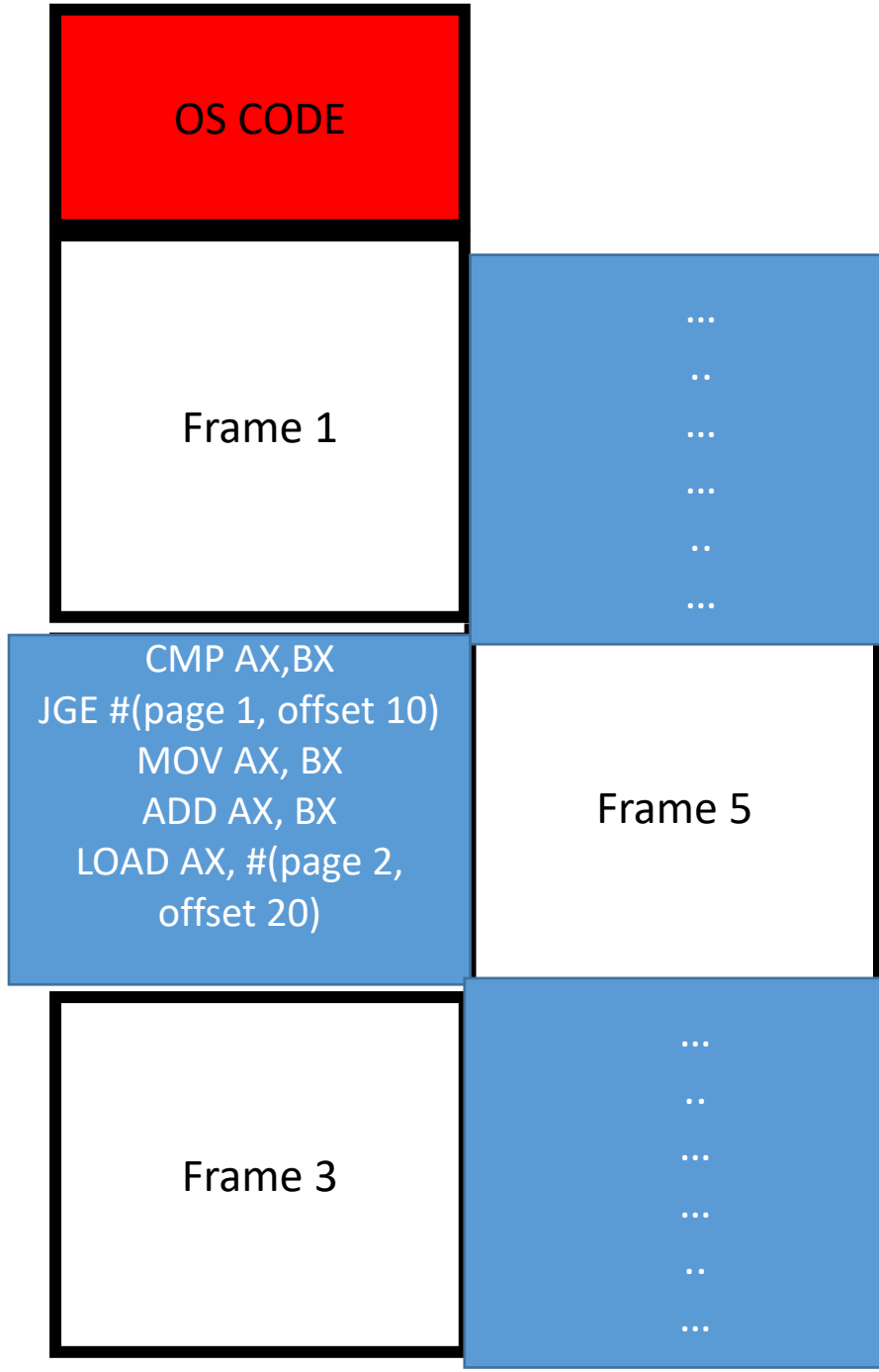
A more efficient approach : Paging



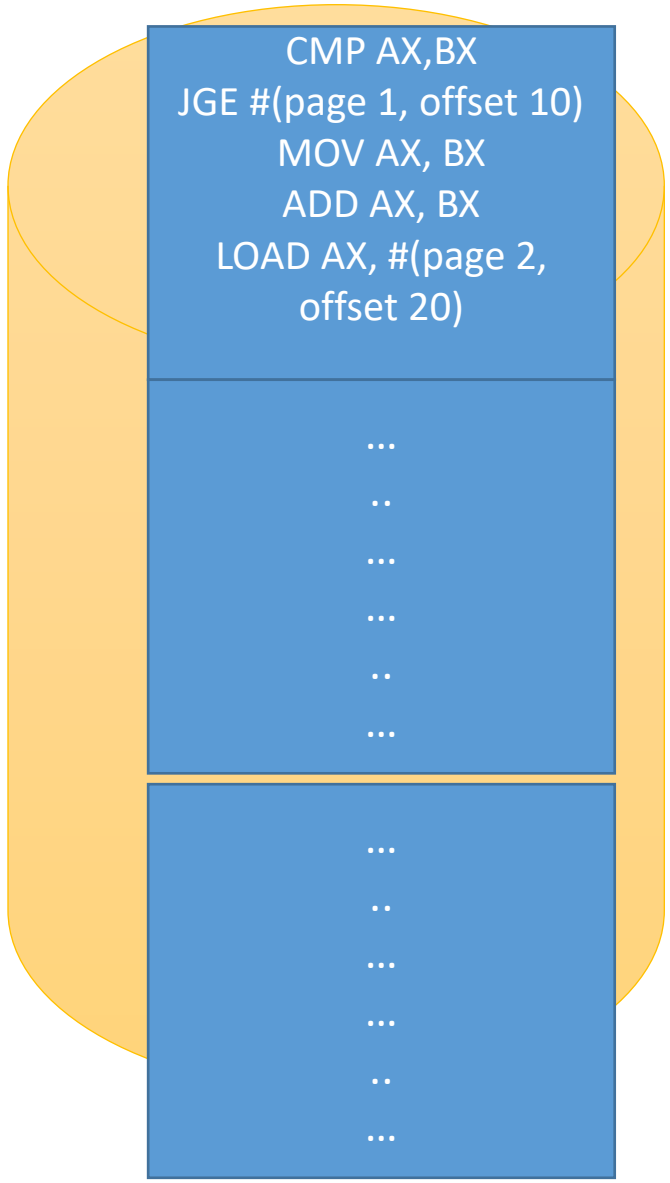


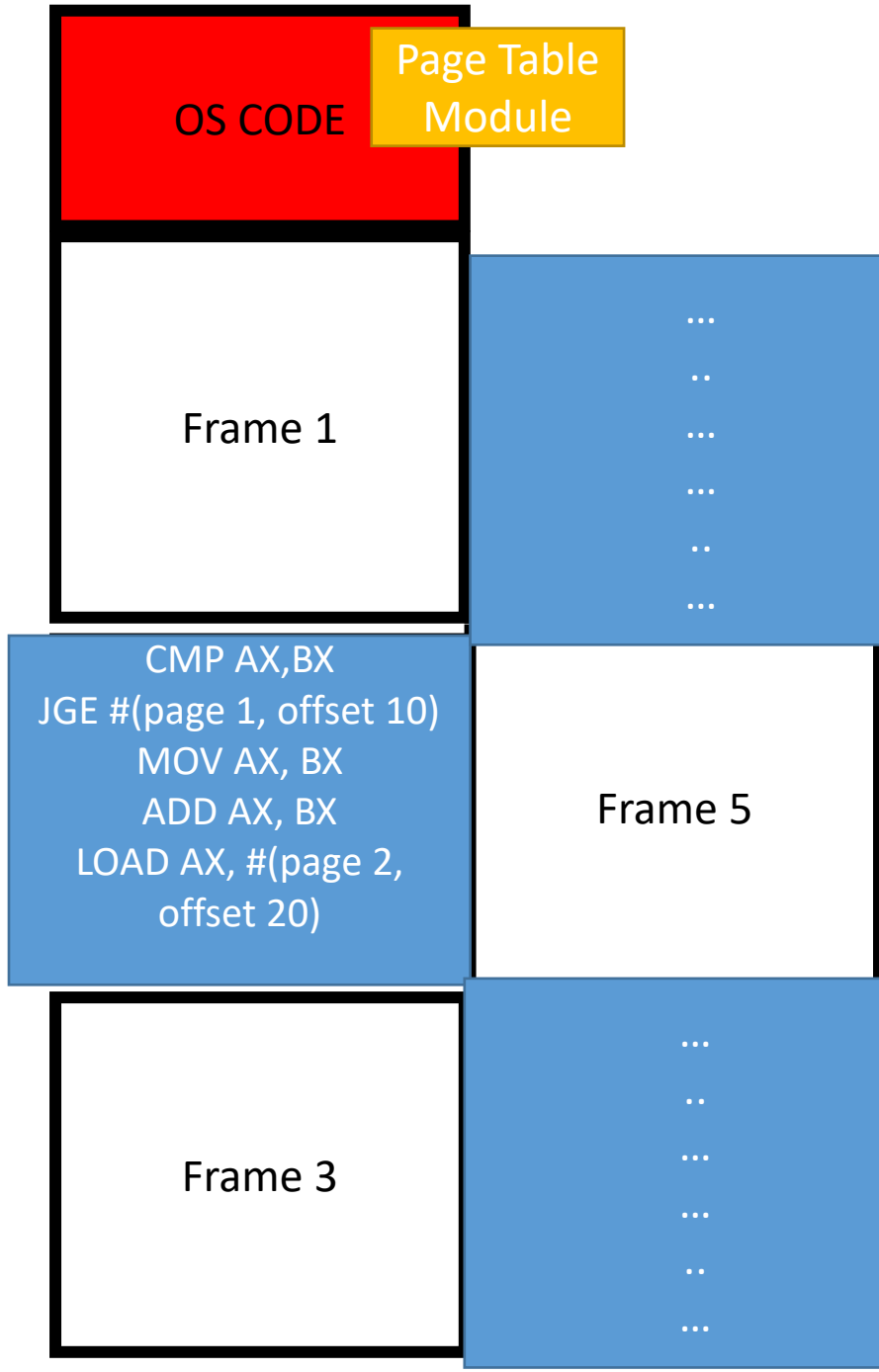
A more efficient approach : Paging



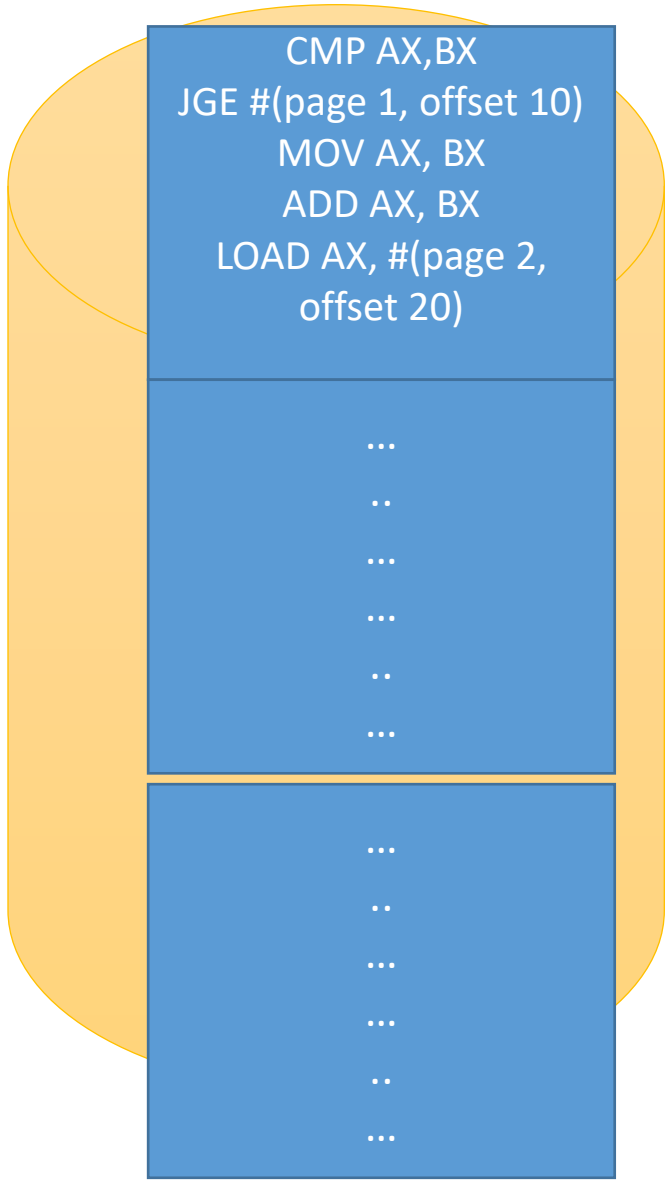


A more efficient approach : Paging

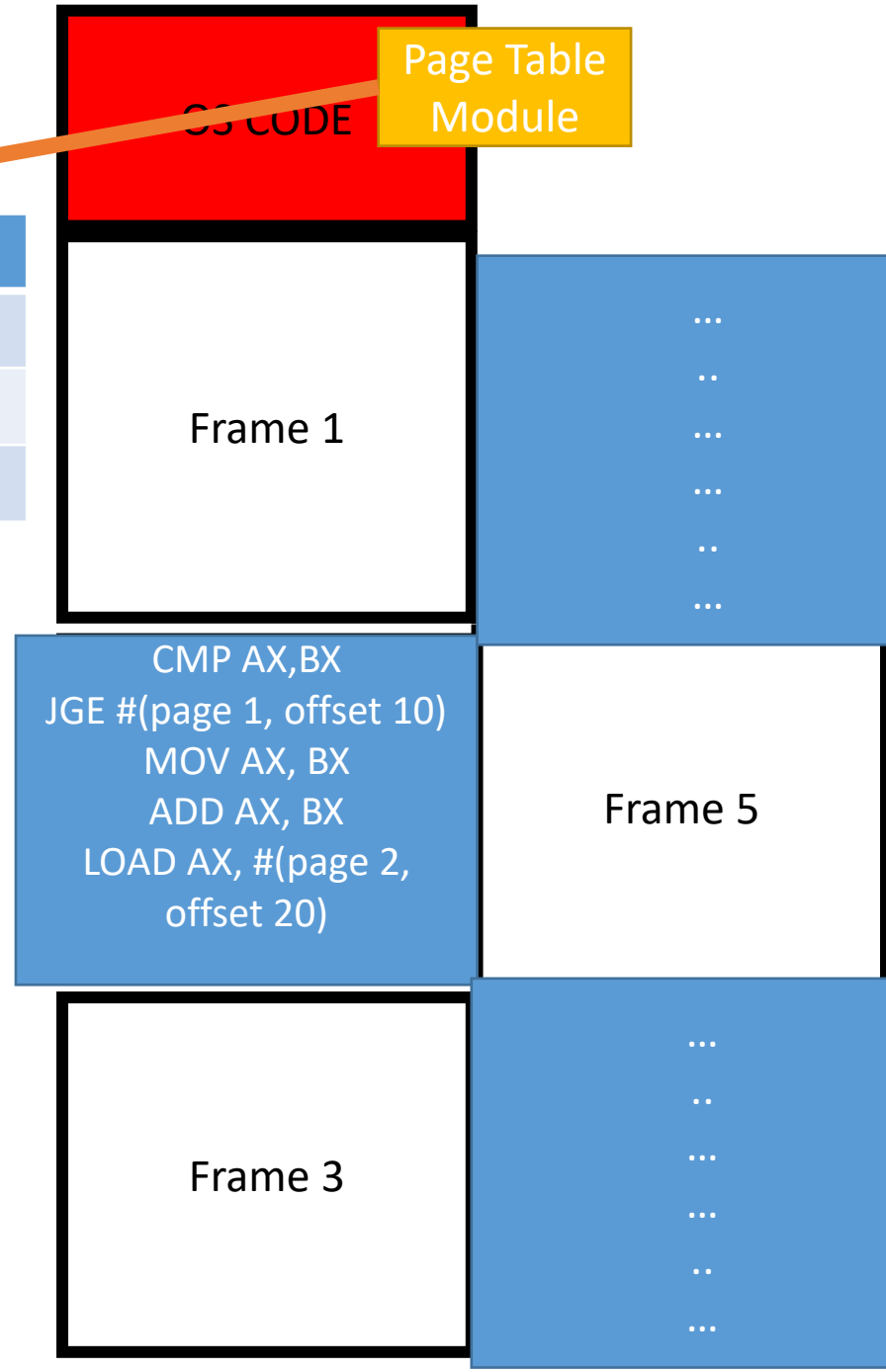




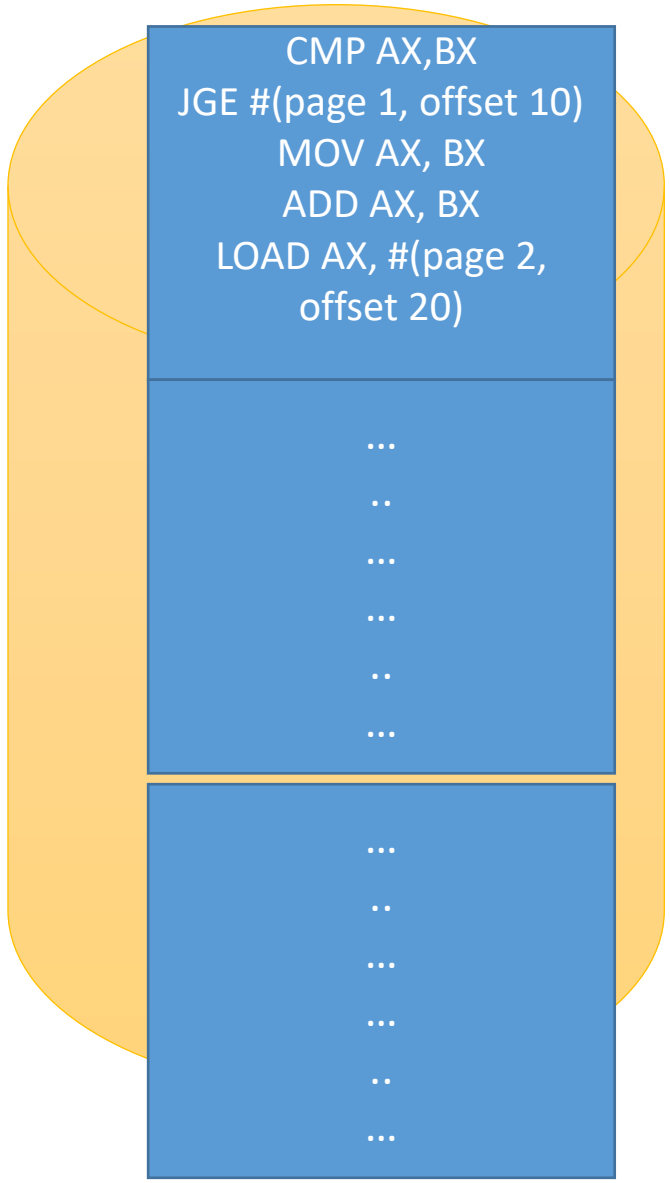
A more efficient approach : Paging



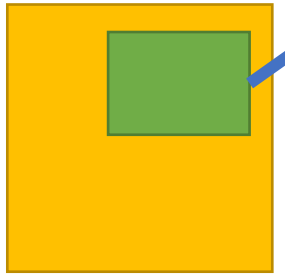
Page	Frame
Page 1	Frame 2
Page 2	Frame 4
Page 3	Frame 6

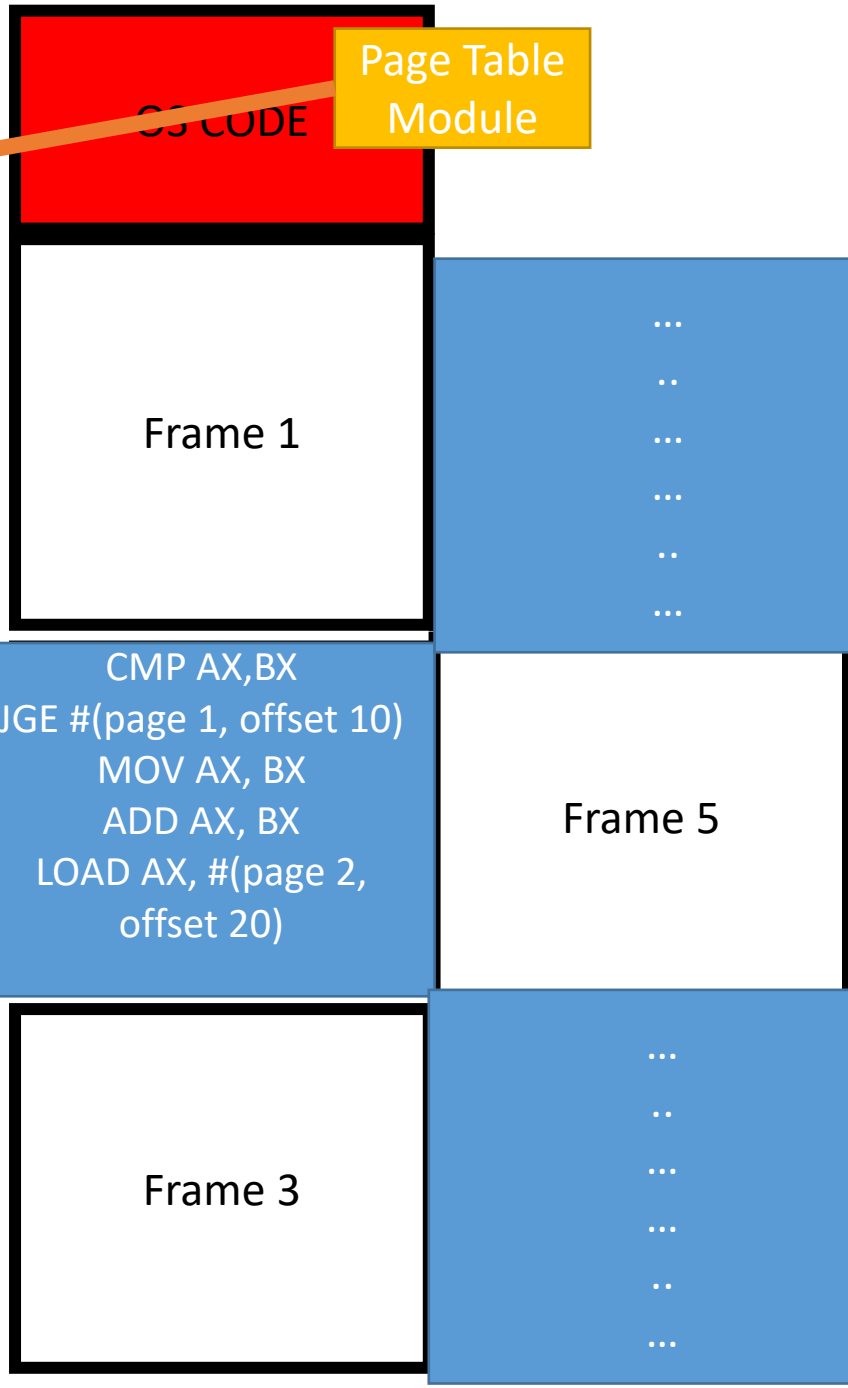


A more efficient approach : Paging

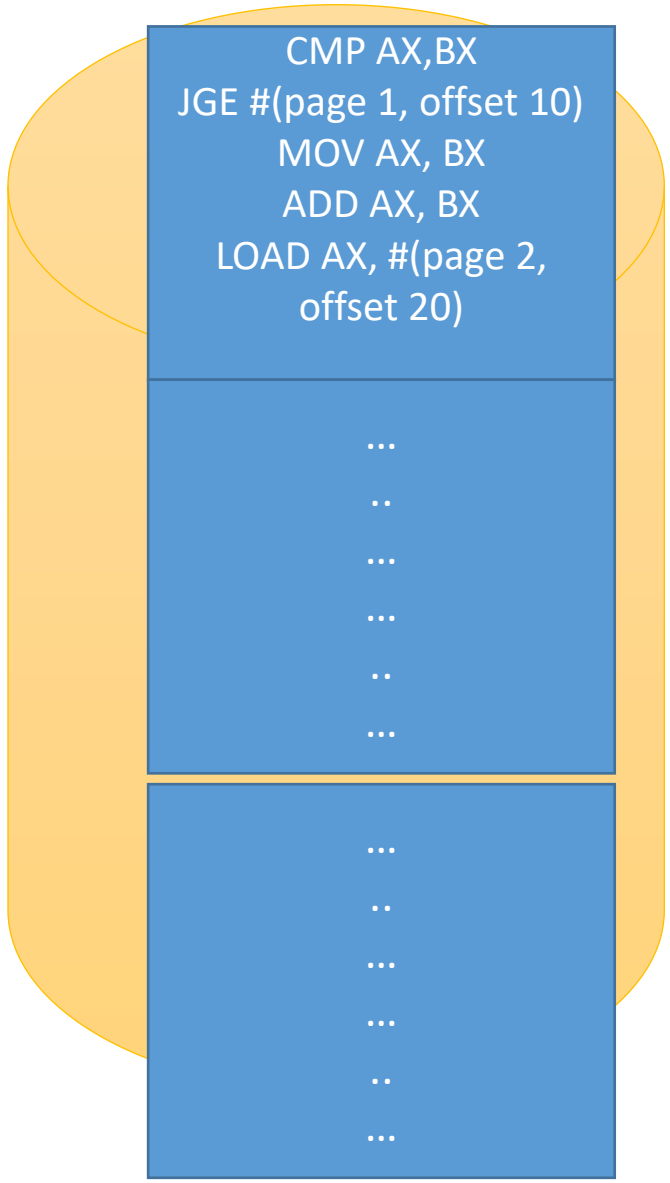


Page	Frame
Page 1	Frame 2
Page 2	Frame 4
Page 3	Frame 6

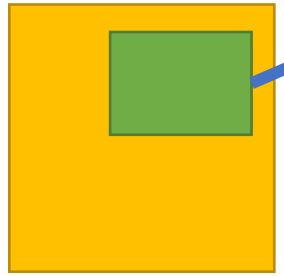


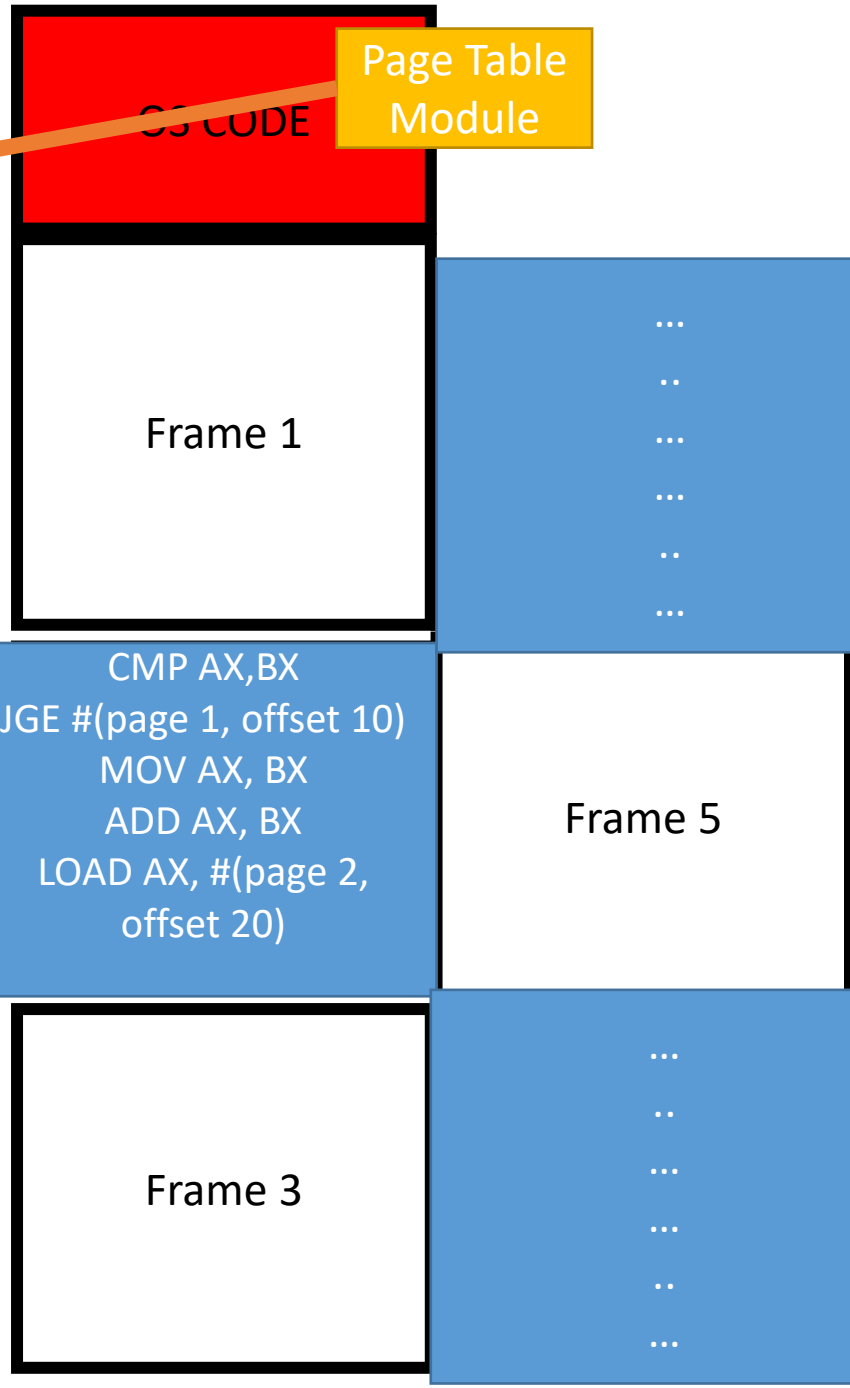


A more efficient approach : Paging

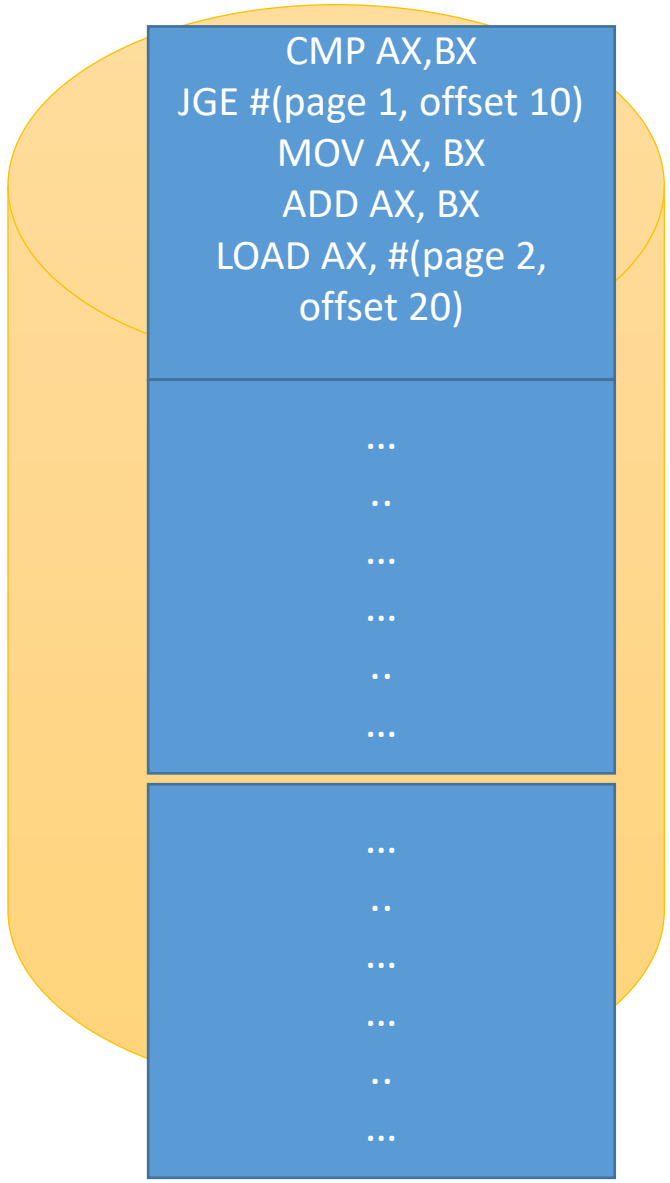


Page	Frame
Page 1	Frame 2
Page 2	Frame 4
Page 3	Frame 6

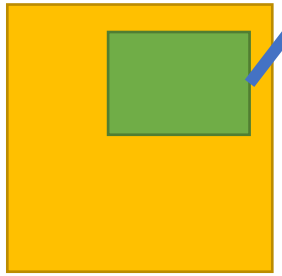


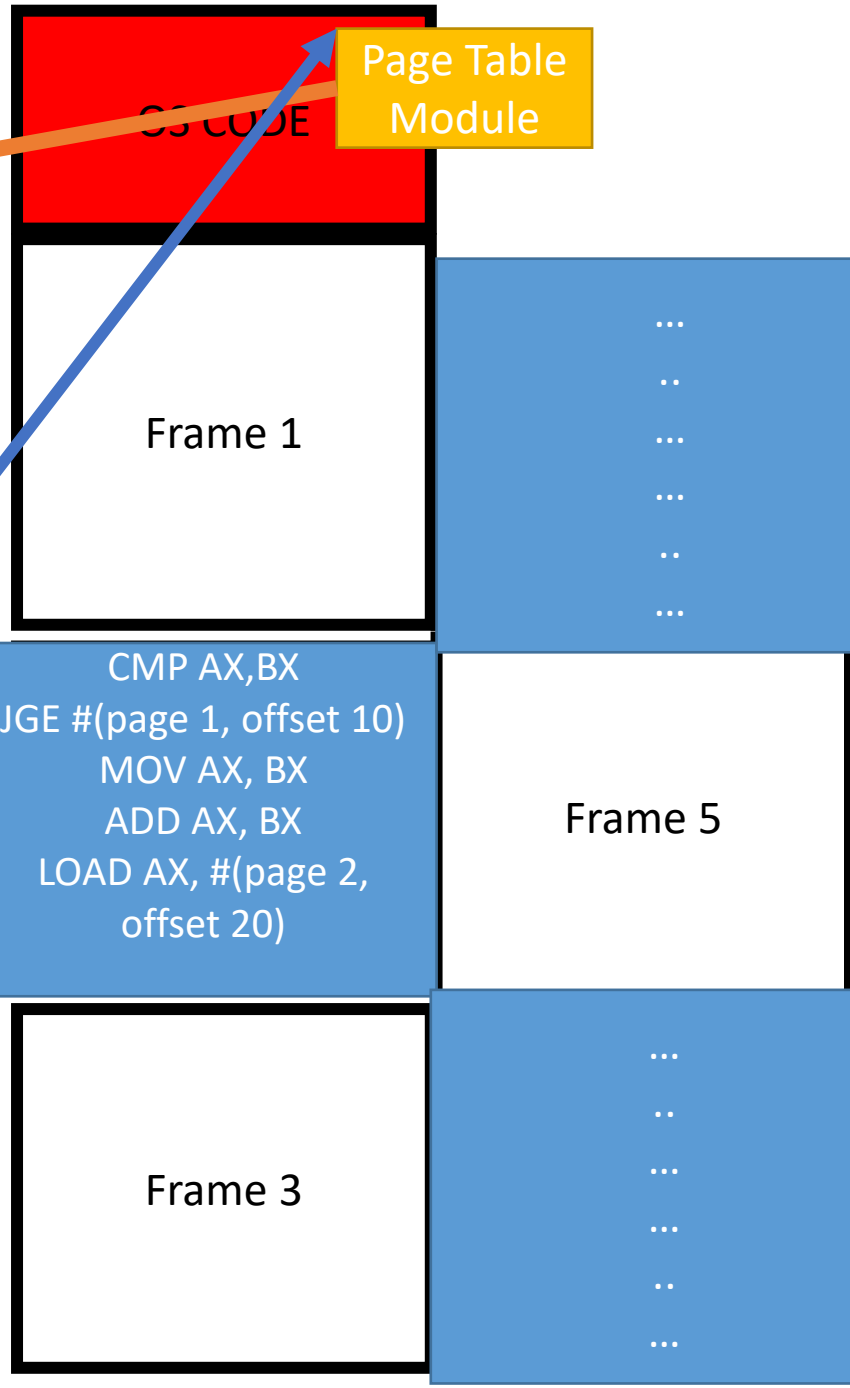


A more efficient approach : Paging

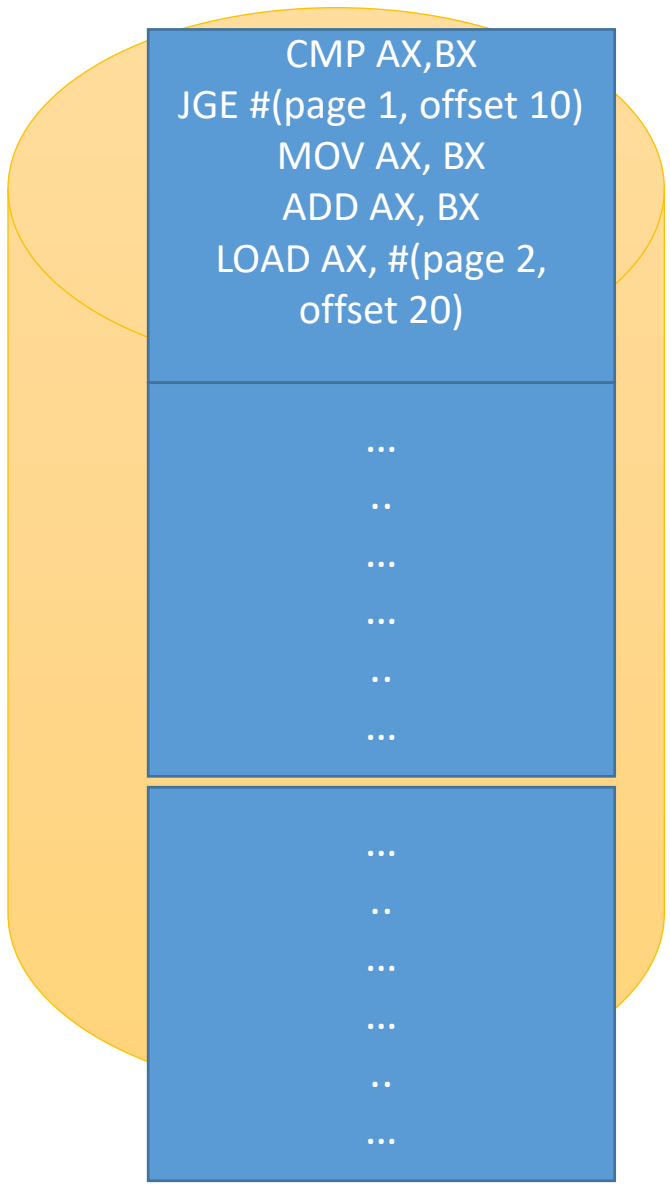


Page	Frame
Page 1	Frame 2
Page 2	Frame 4
Page 3	Frame 6

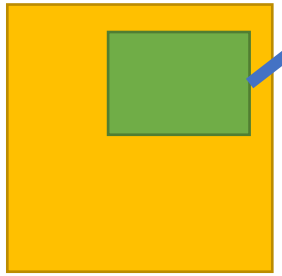


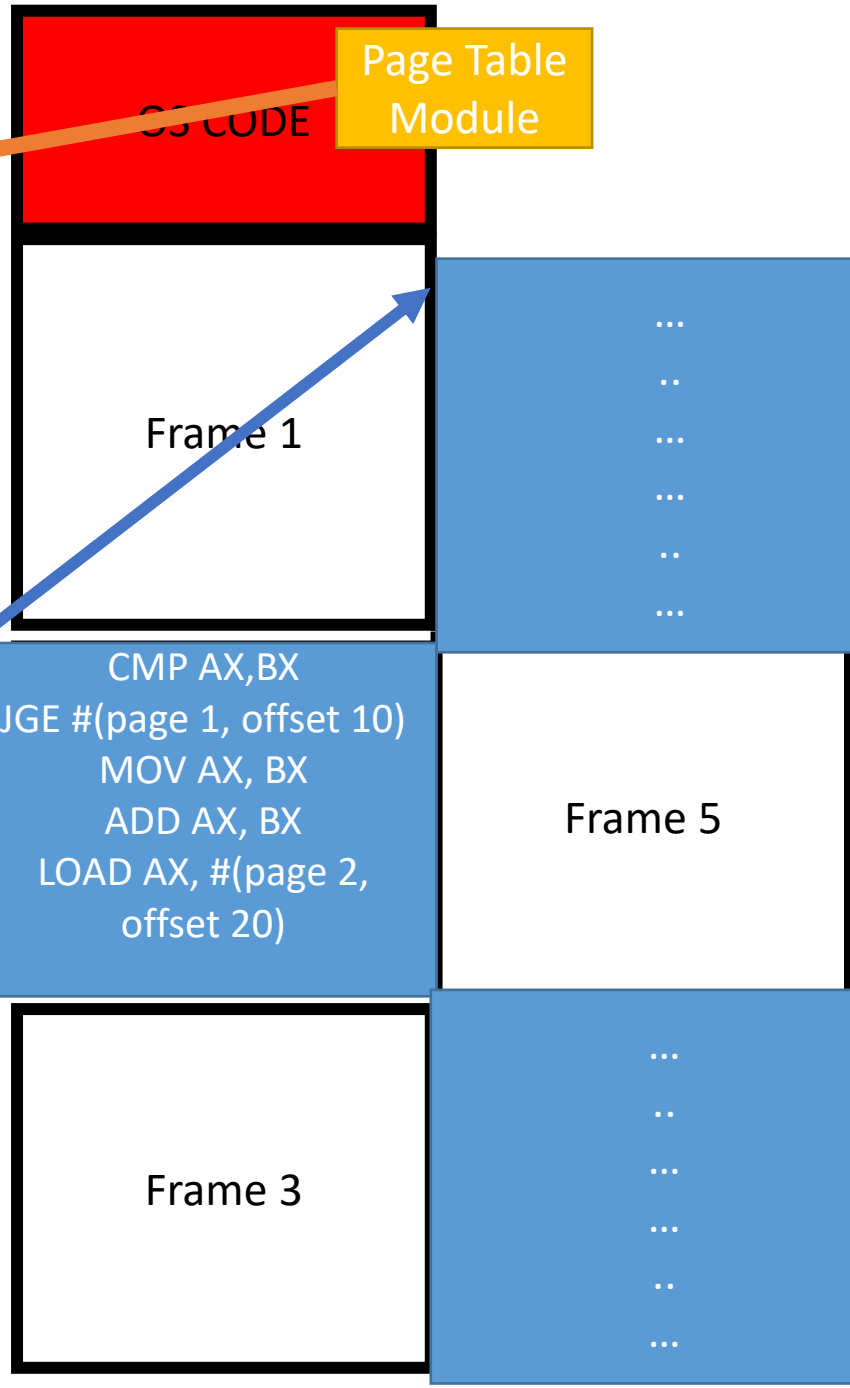


A more efficient approach : Paging



Page	Frame
Page 1	Frame 2
Page 2	Frame 4
Page 3	Frame 6





Demand Paging

- It is not wise to bring in **all the pages of a program** (as we did with the 3 pages of the previous program)
- We should bring in a program's page only when it is **required**.
- **Principle of locality** : A program's execution is confined to some particular pages (because of loops, functions, conditions).
- That is the **execution time of few pages** are way more compared to the **other pages** of the program

4.1 Demand Paging

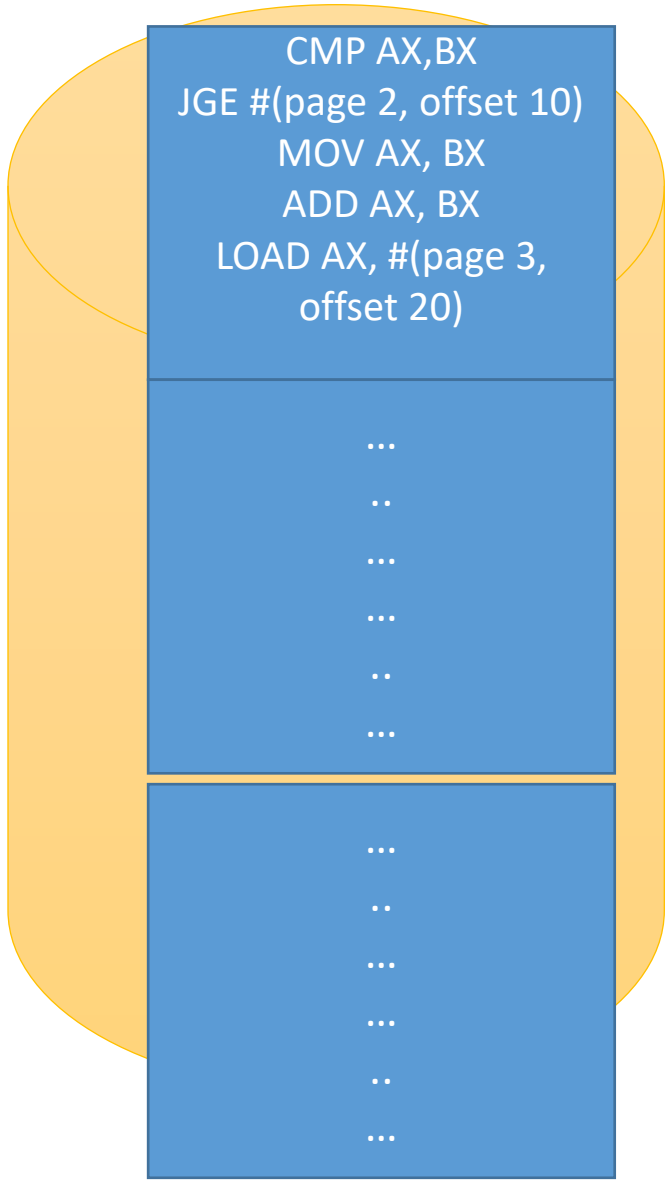
Demand Paging

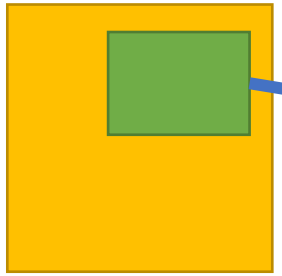
Three terms related to Demand Paging

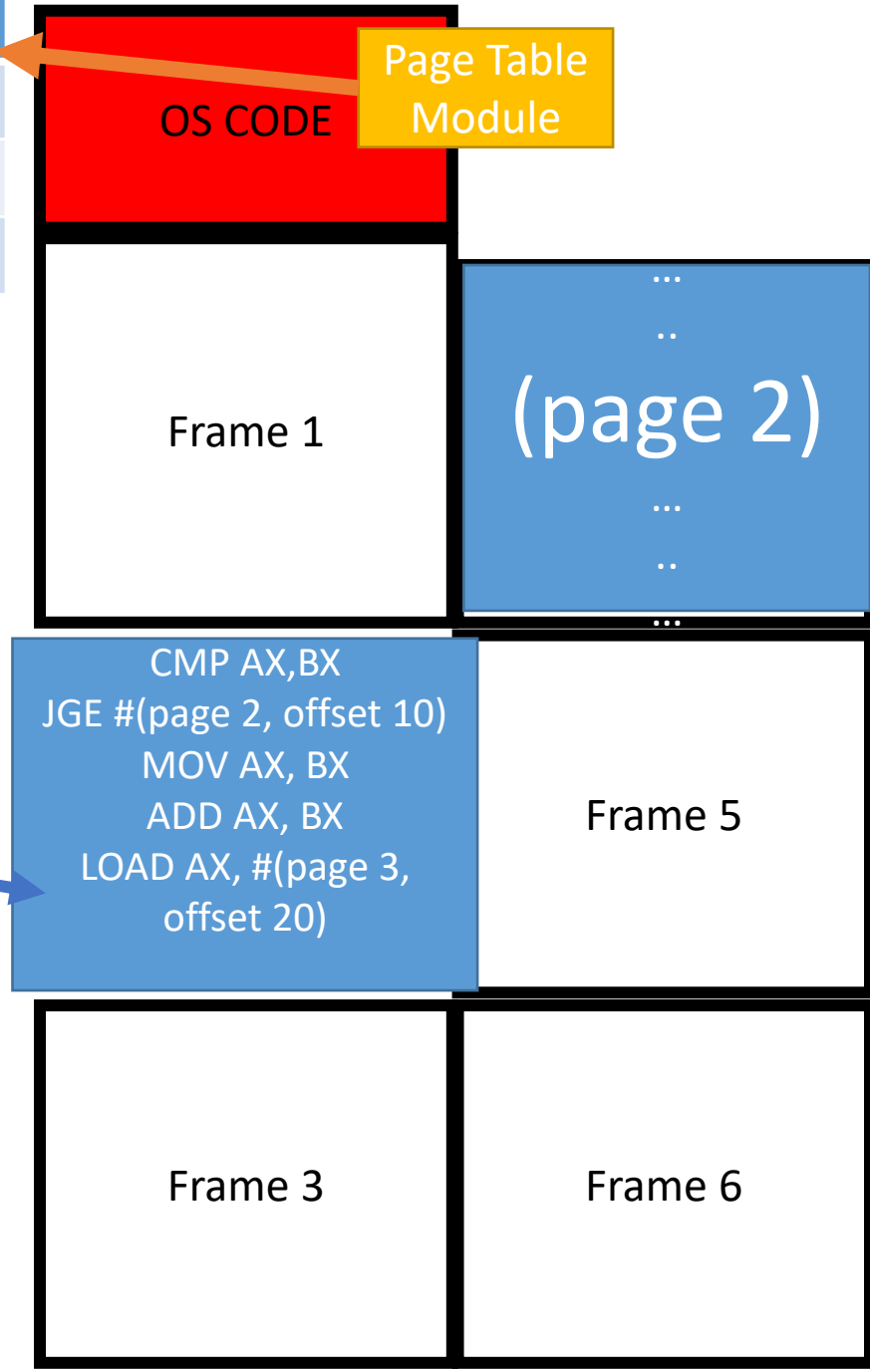
- Page Fault
- Replacement Algorithm
- Thrashing

Demand Paging

Page	Frame
Page 1	Frame 2
Page 2	Frame 4

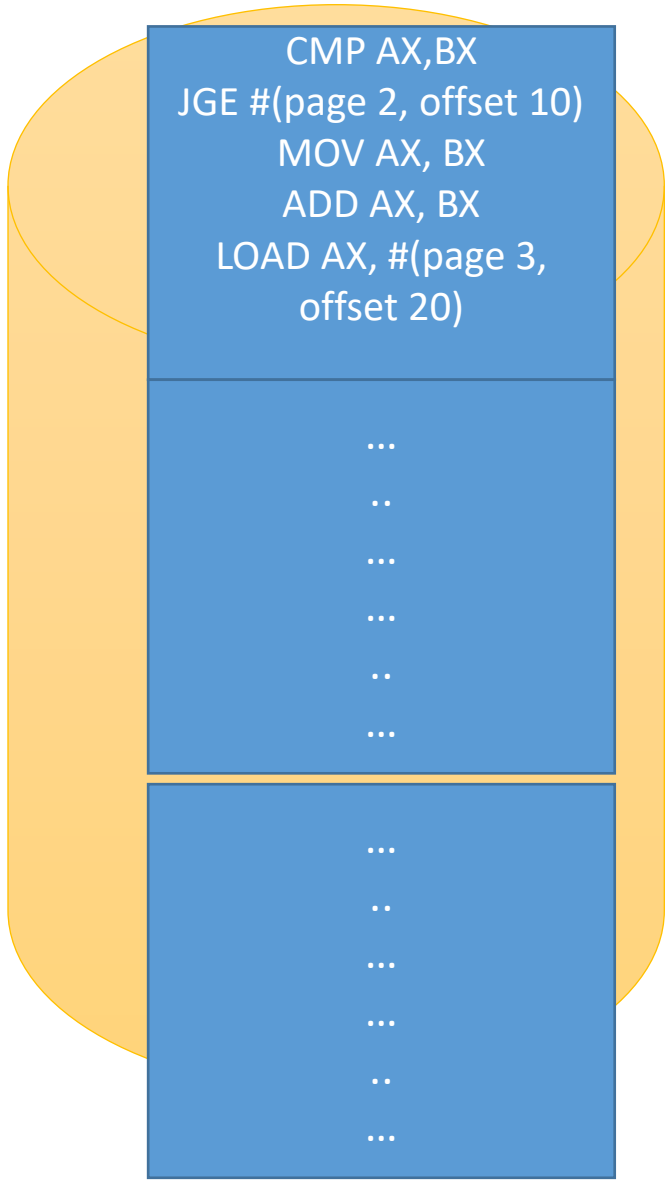




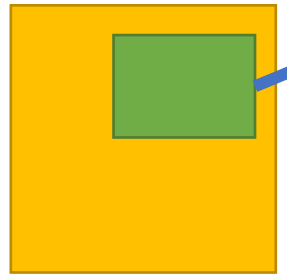


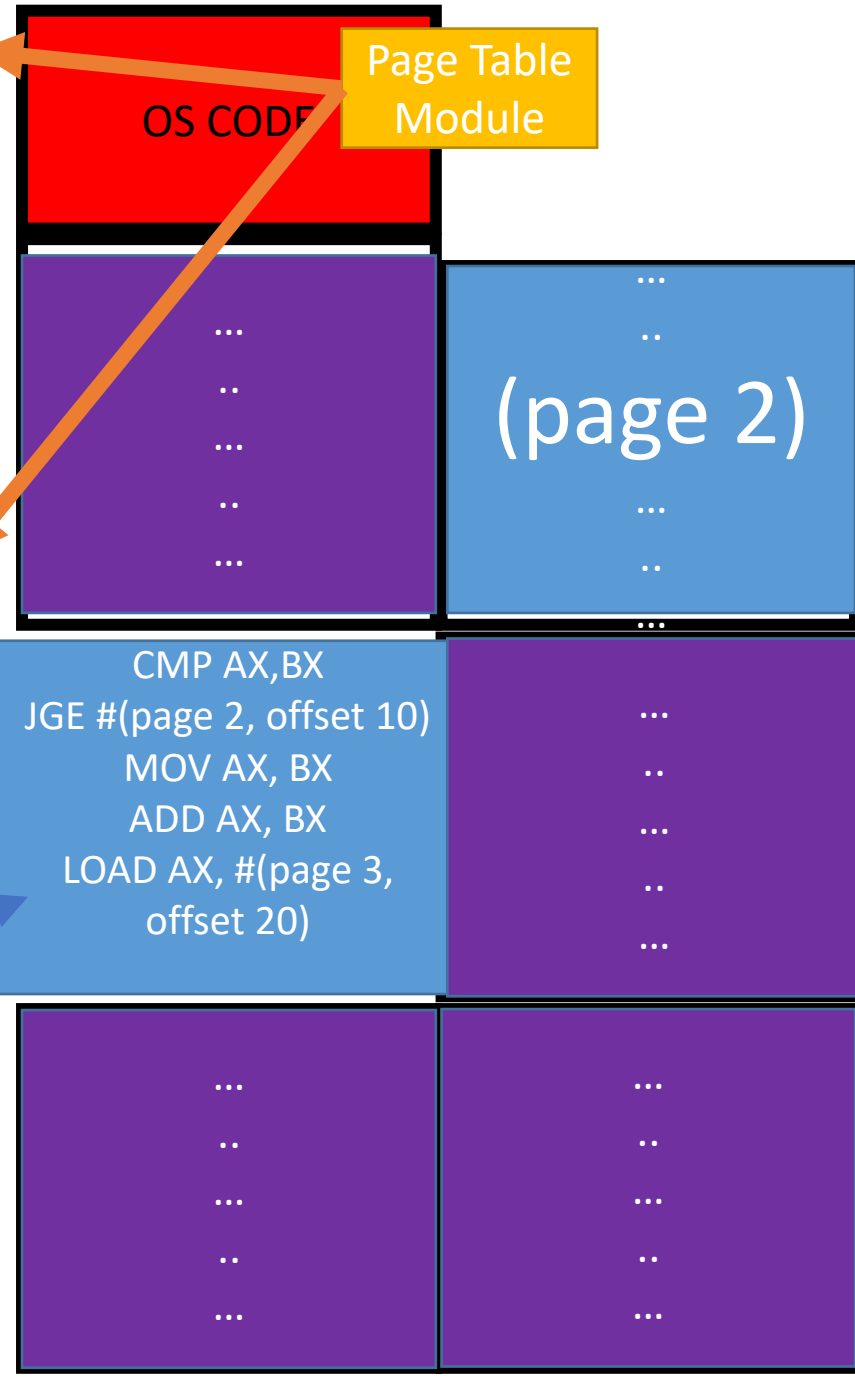
Demand Paging

Page	Frame
Page 1	Frame 2
Page 2	Frame 4



Page	Frame
Page 1	Frame 1
Page 2	Frame 3
Page 3	Frame 5
Page 4	Frame 6





Demand Paging

Page	Frame
Page 1	Frame 2
Page 2	Frame 4

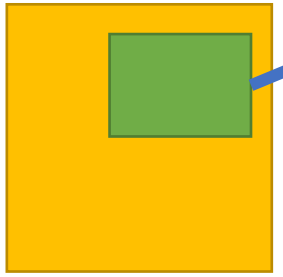
CMP AX,BX
JGE #(page 2, offset 10)
MOV AX, BX
ADD AX, BX
LOAD AX, #(page 3, offset 20)

...
..
...
...
..
...

...
..
...
...
..
...

Page	Frame
Page 1	Frame 1
Page 2	Frame 3
Page 3	Frame 5
Page 4	Frame 6

Observe that this instruction is Requiring data from a page that is not In the memory (**page 3**)



OS CODE

Page Table Module

...
..
...
..
...

...
..
(page 2)
...
..

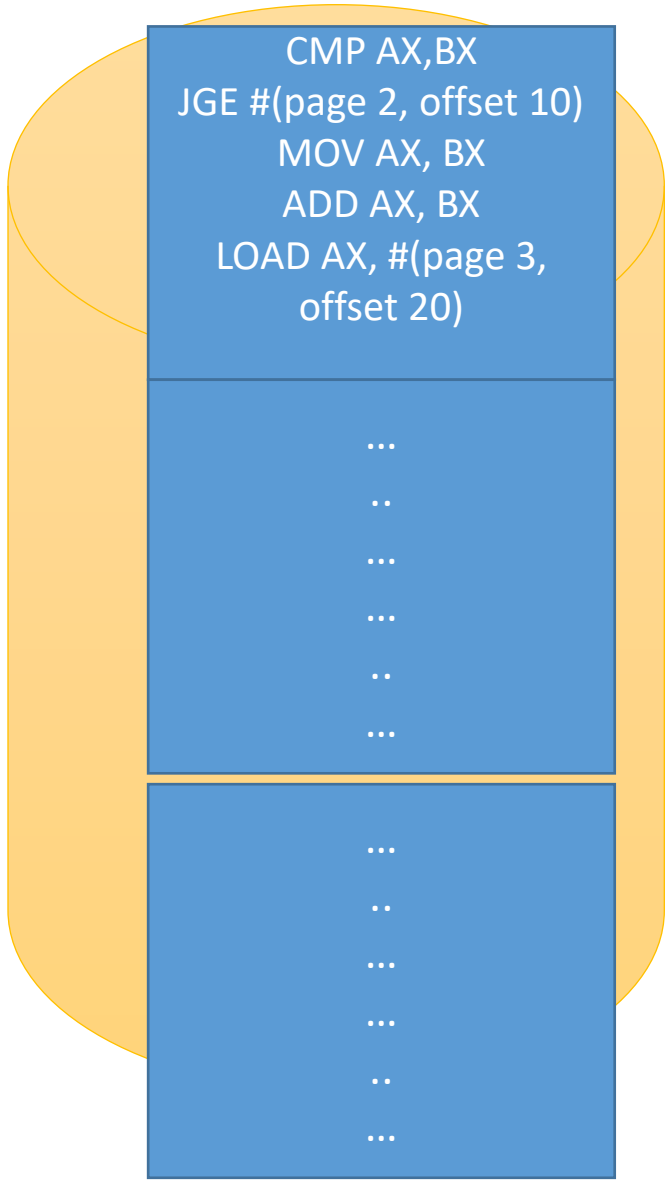
CMP AX,BX
JGE #(page 2, offset 10)
MOV AX, BX
ADD AX, BX
LOAD AX, #(page 3, offset 20)

...
..
...
..
...

...
..
...
..
...

...
..
...
..
...

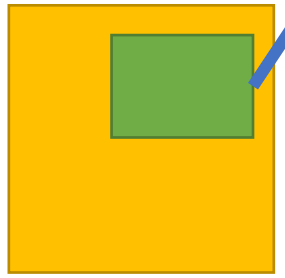
Demand Paging

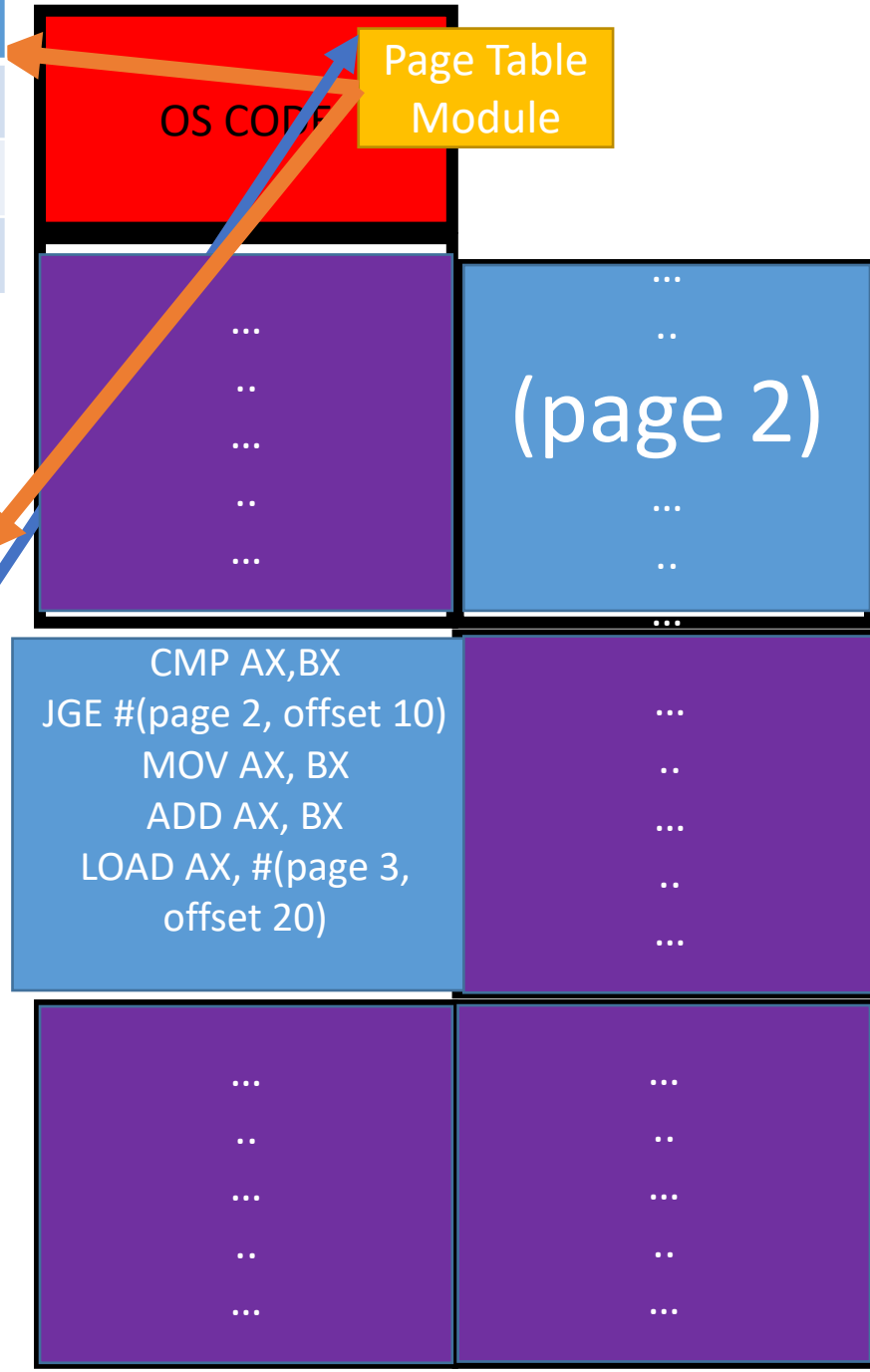


Page	Frame
Page 1	Frame 2
Page 2	Frame 4

Page	Frame
Page 1	Frame 1
Page 2	Frame 3
Page 3	Frame 5
Page 4	Frame 6

The Processor goes to the OS's page Page table module





Demand Paging

CMP AX,BX
JGE #(page 2, offset 10)
MOV AX, BX
ADD AX, BX
LOAD AX, #(page 3, offset 20)

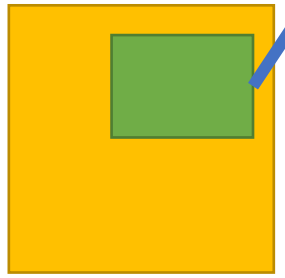
...
..
...
...
..
...

...
..
...
...
..
...

Page	Frame
Page 1	Frame 2
Page 2	Frame 4

Page	Frame
Page 1	Frame 1
Page 2	Frame 3
Page 3	Frame 5
Page 4	Frame 6

The Processor goes to the OS's page Page table module



But no data of page 3 is there:
This situation is called **PAGE FAULT**

OS CODE

Page Table Module

...
..
...
..
...

...
..
(page 2)
...
..

CMP AX,BX
JGE #(page 2, offset 10)
MOV AX, BX
ADD AX, BX
LOAD AX, #(page 3, offset 20)

...
..
...
..
...

...
..
...
..
...

...
..
...
..
...

Demand Paging

CMP AX,BX
JGE #(page 2, offset 10)
MOV AX, BX
ADD AX, BX
LOAD AX, #(page 3, offset 20)

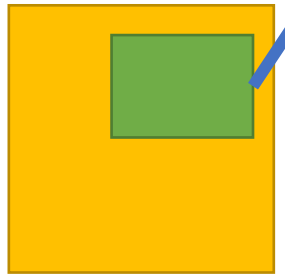
...
..
...
...
..
...

...
..
...
...
..
...

Page	Frame
Page 1	Frame 2
Page 2	Frame 4

Page	Frame
Page 1	Frame 1
Page 2	Frame 3
Page 3	Frame 5
Page 4	Frame 6

The Processor goes to the OS's page Page table module



But no data of page 3 is there

OS CODE

Page Table Module

Page rep algo

...
..
...
..
...

...
..
(page 2)
...
..
...

CMP AX,BX
JGE #(page 2, offset 10)
MOV AX, BX
ADD AX, BX
LOAD AX, #(page 3, offset 20)

...
..
...
..
...

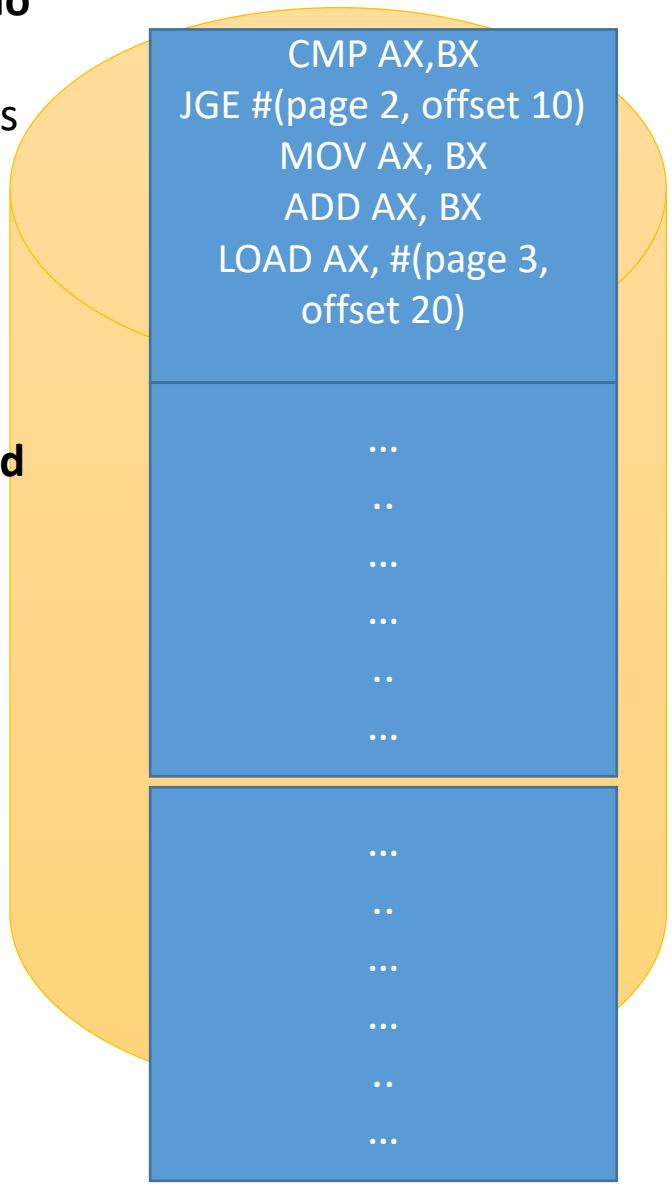
...
..
...
...
..
...

...
..
...
..
...

...
..
...
..
...

Demand Paging

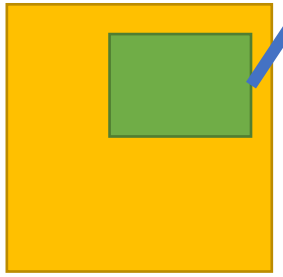
Since, **no empty Frame** is there, one of the Pages will be **replaced**



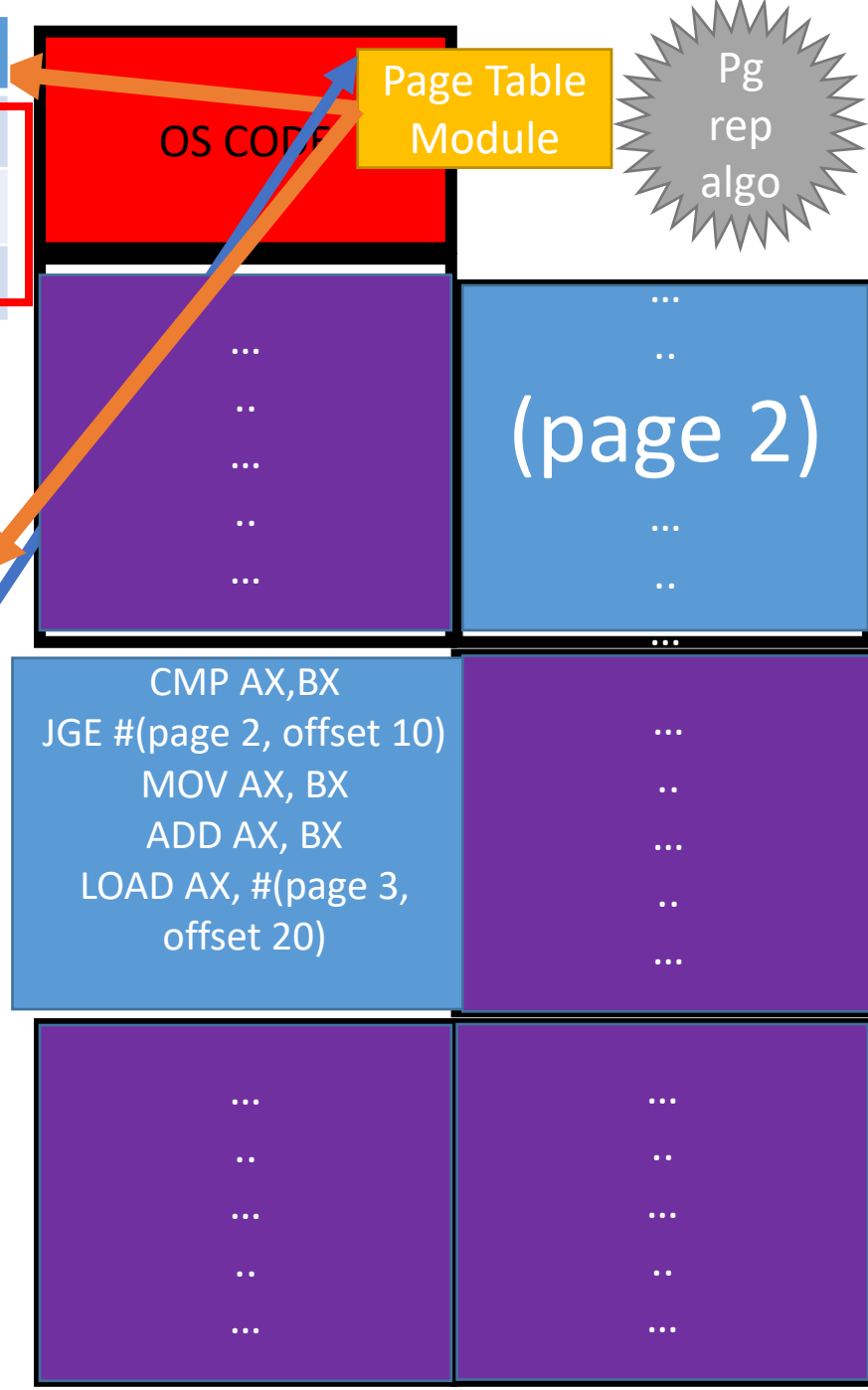
Page	Frame
Page 1	Frame 2
Page 2	Frame 4

Page	Frame
Page 1	Frame 1
Page 2	Frame 3
Page 3	Frame 5
Page 4	Frame 6

The Processor goes to the OS's page Page table module



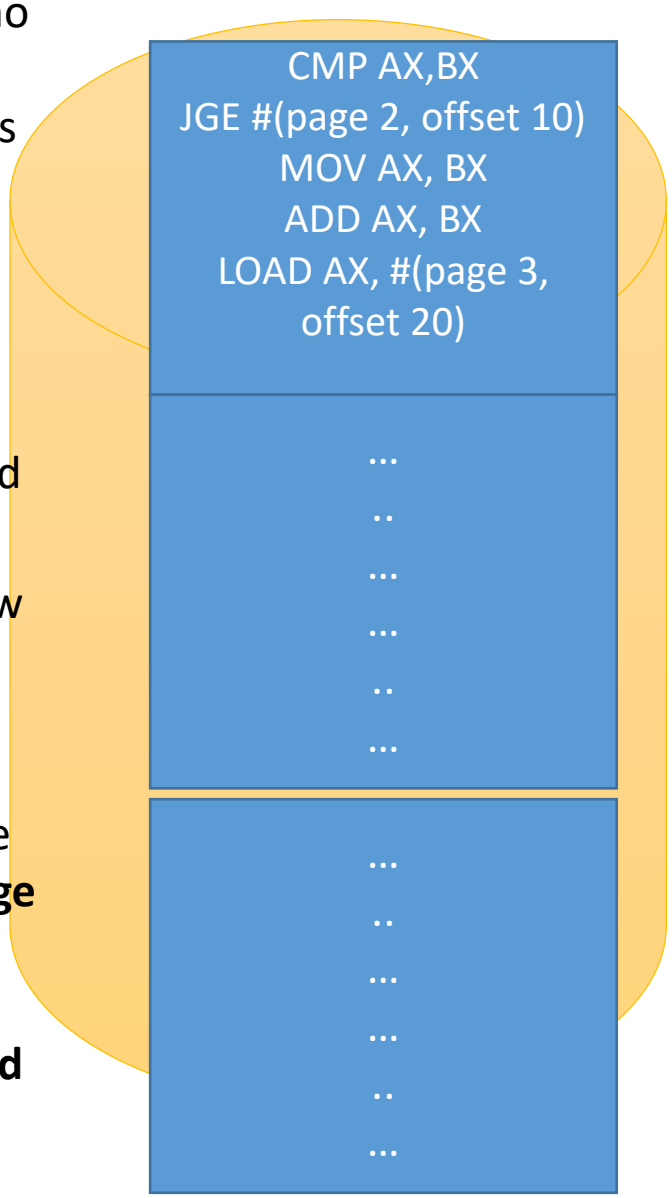
But no data of page 3 is there



Demand Paging

Since, no empty Frame is there, one of the Pages will be replaced

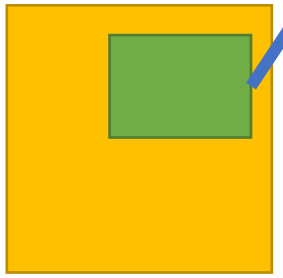
The new Page is Placed inside And the The **page Tables** are **updated**



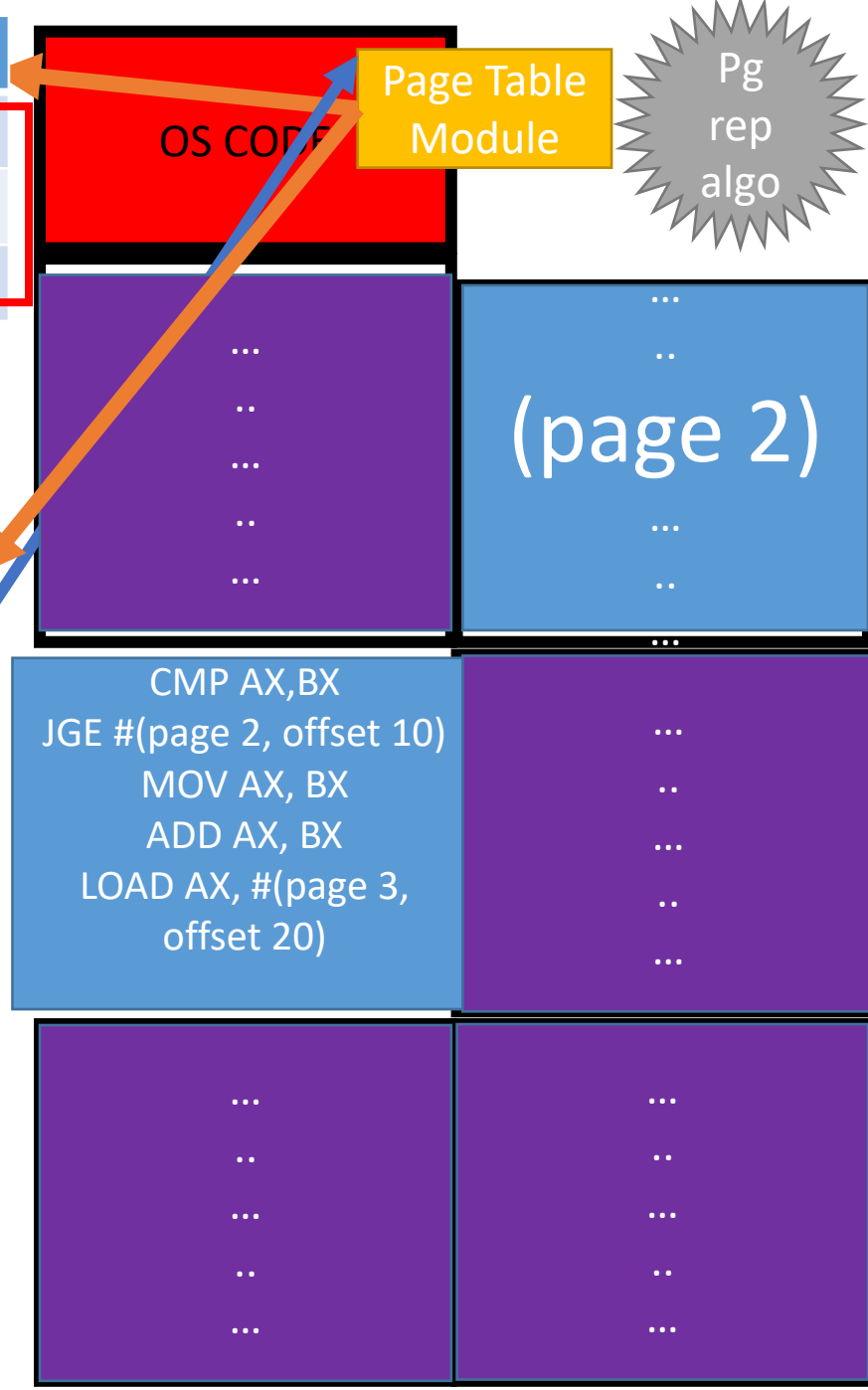
Page	Frame
Page 1	Frame 2
Page 2	Frame 4

Page	Frame
Page 1	Frame 1
Page 2	Frame 3
Page 3	Frame 5
Page 4	Frame 6

The Processor goes to the OS's page Page table module



But no data of page 3 is there



Demand Paging

Since, no empty Frame is there, one of the Pages will be replaced

The new Page is Placed inside And the The page Tables are updated

CMP AX,BX
JGE #(page 2, offset 10)
MOV AX, BX
ADD AX, BX
LOAD AX, #(page 3, offset 20)

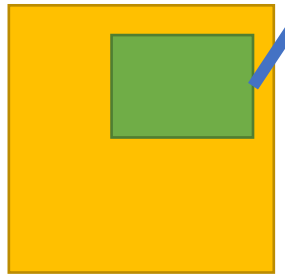
...
..
...
...
..
...

...
..
...
...
..
...

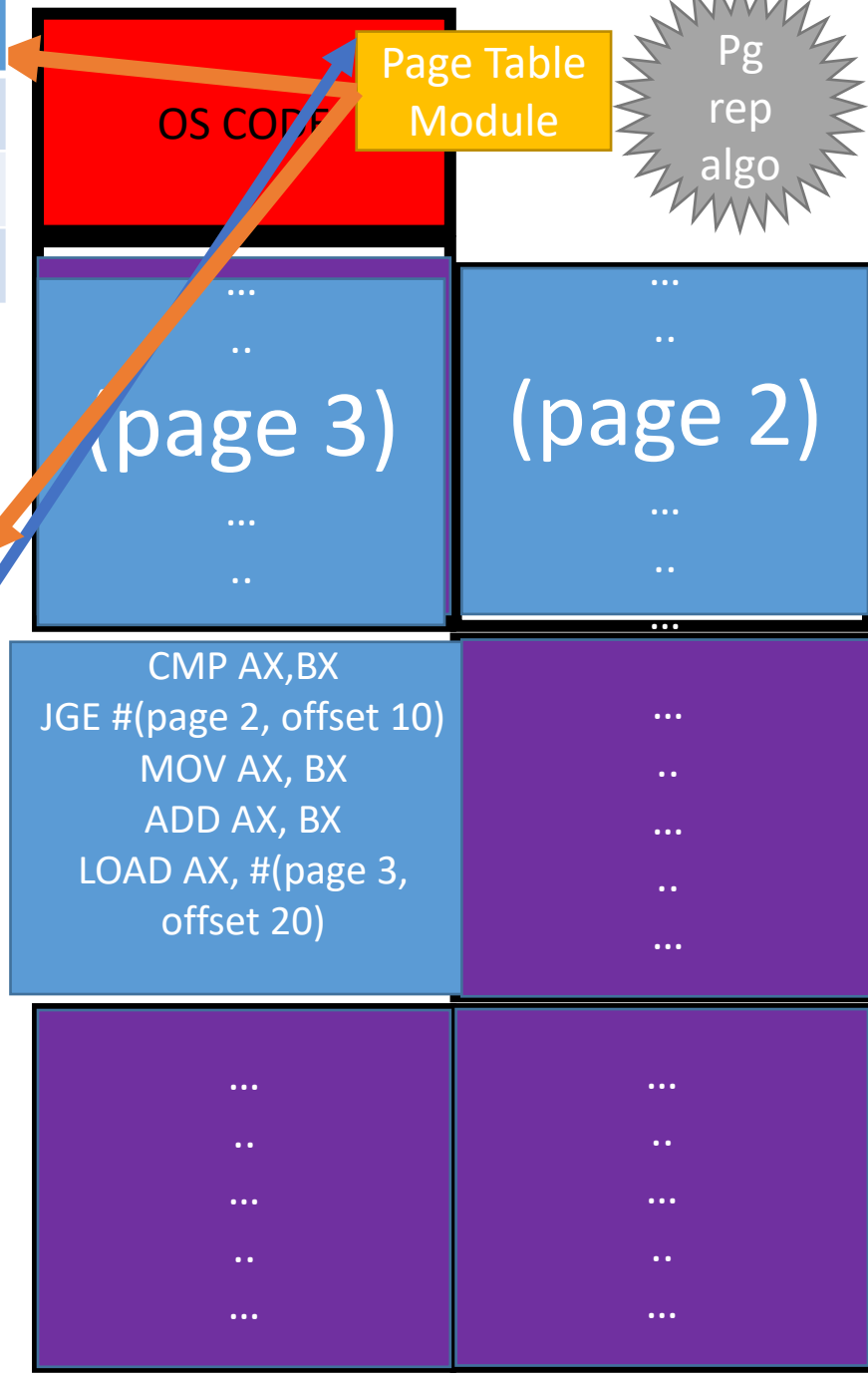
Page	Frame
Page 1	Frame 2
Page 2	Frame 4
Page 3	Frame 1

Page	Frame
Page 2	Frame 3
Page 3	Frame 5
Page 4	Frame 6

The Processor goes to the OS's page Page table module



But no data of page 3 is there



Demand Paging

Using The new Entry The process **Execution Is continued** By updating The **program counter**

CMP AX,BX
JGE #(page 2, offset 10)
MOV AX, BX
ADD AX, BX
LOAD AX, #(page 3, offset 20)

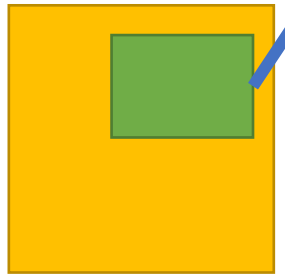
...
..
...
...
..
...

...
..
...
...
..
...

Page	Frame
Page 1	Frame 2
Page 2	Frame 4
Page 3	Frame 1

Page	Frame
Page 2	Frame 3
Page 3	Frame 5
Page 4	Frame 6

The Processor goes to the OS's page Page table module



But no data of page 3 is there

OS CODE

Page Table Module

Pg rep algo

...
..
(page 3)
...
..

...
..
(page 2)
...
..

CMP AX,BX
JGE #(page 2, offset 10)
MOV AX, BX
ADD AX, BX
LOAD AX, #(page 3, offset 20)

...
..
...
..
...

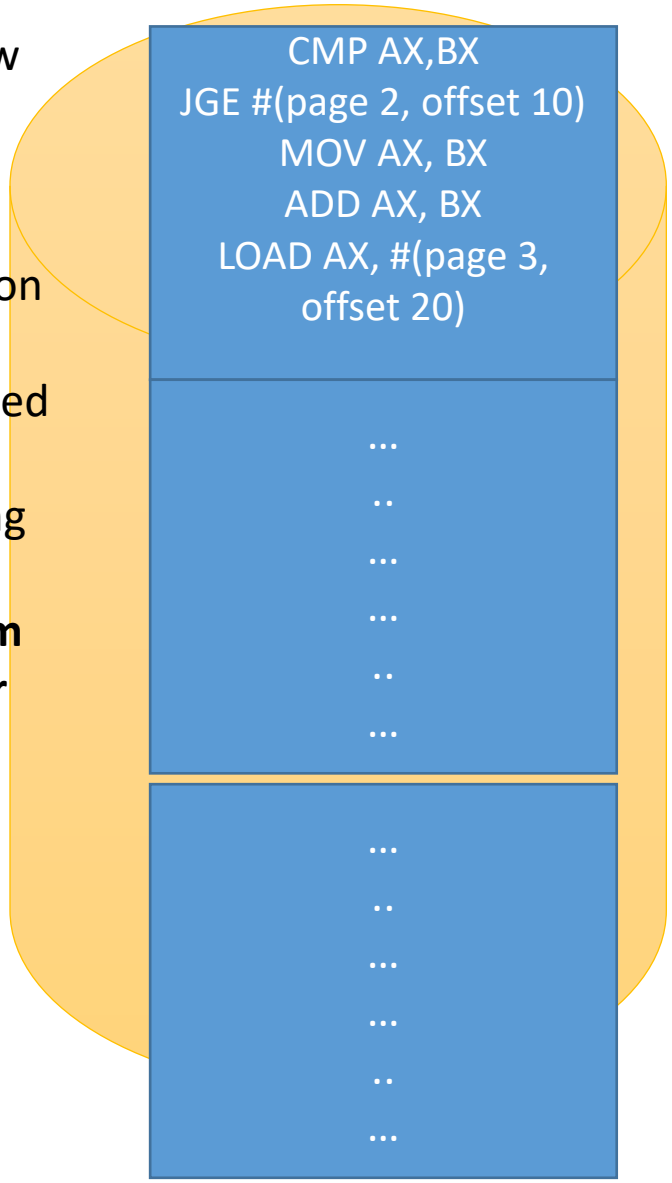
...
..
...
..
...

...
..
...
..
...

...
..
...
..
...

Demand Paging

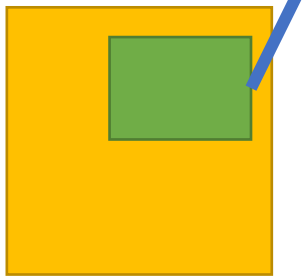
Using The new Entry The process Execution Is continued By updating The **program counter**



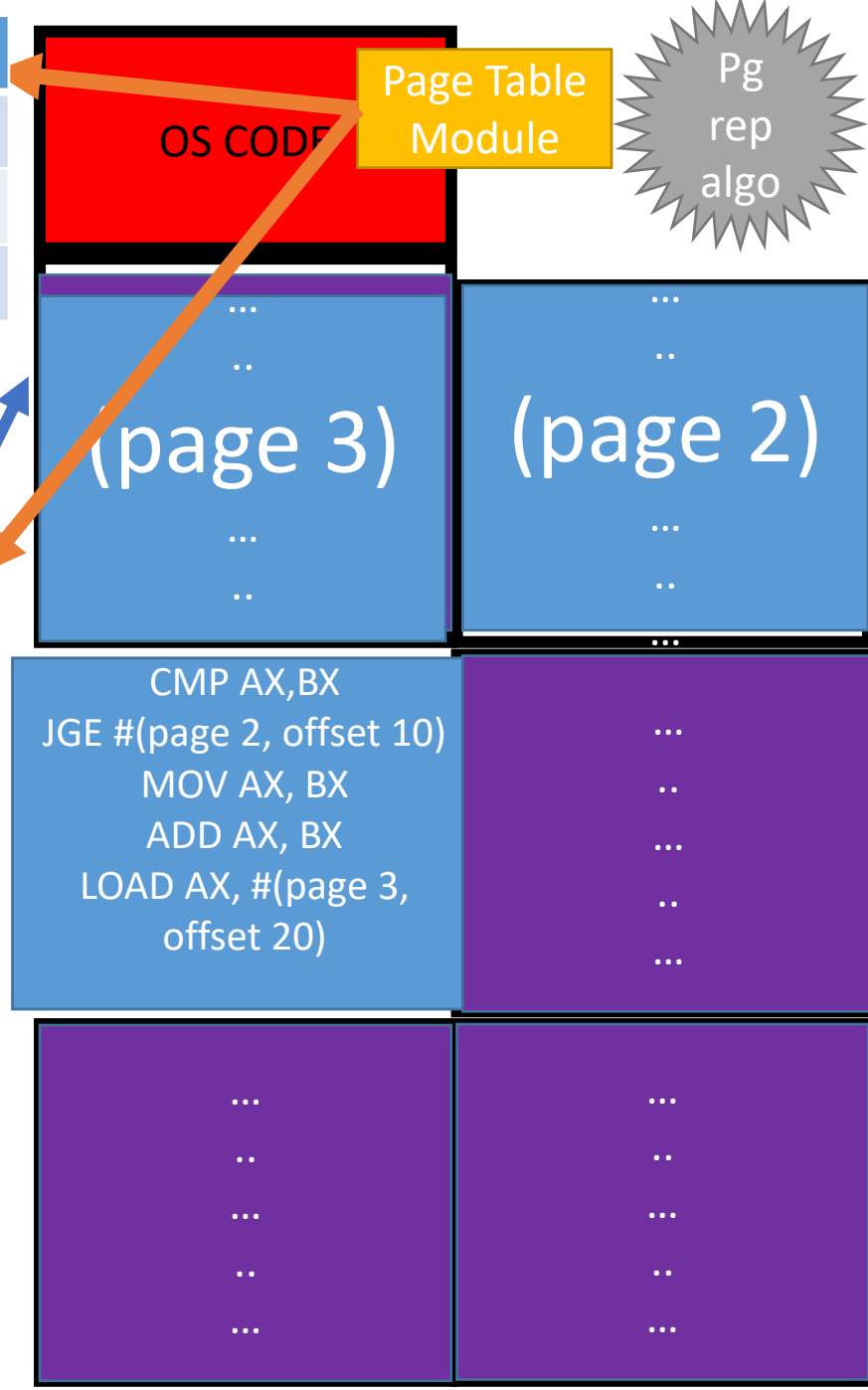
Page	Frame
Page 1	Frame 2
Page 2	Frame 4
Page 3	Frame 1

Page	Frame
Page 2	Frame 3
Page 3	Frame 5
Page 4	Frame 6

The Processor goes to the OS's page Page table module

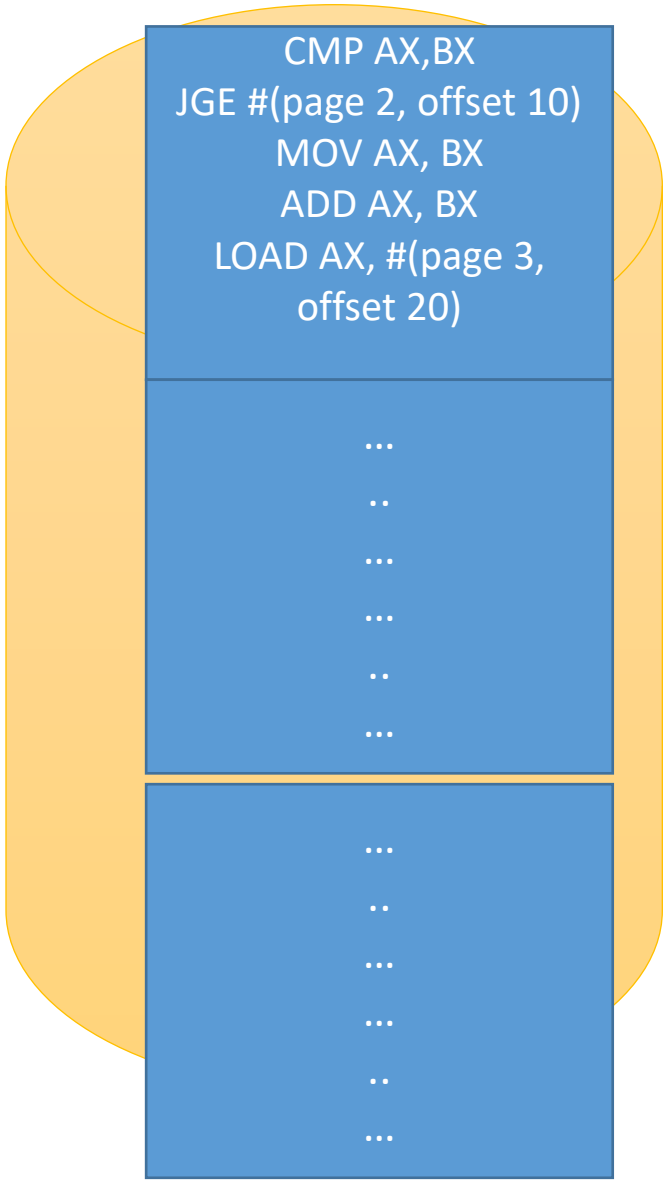


But no data of page 3 is there



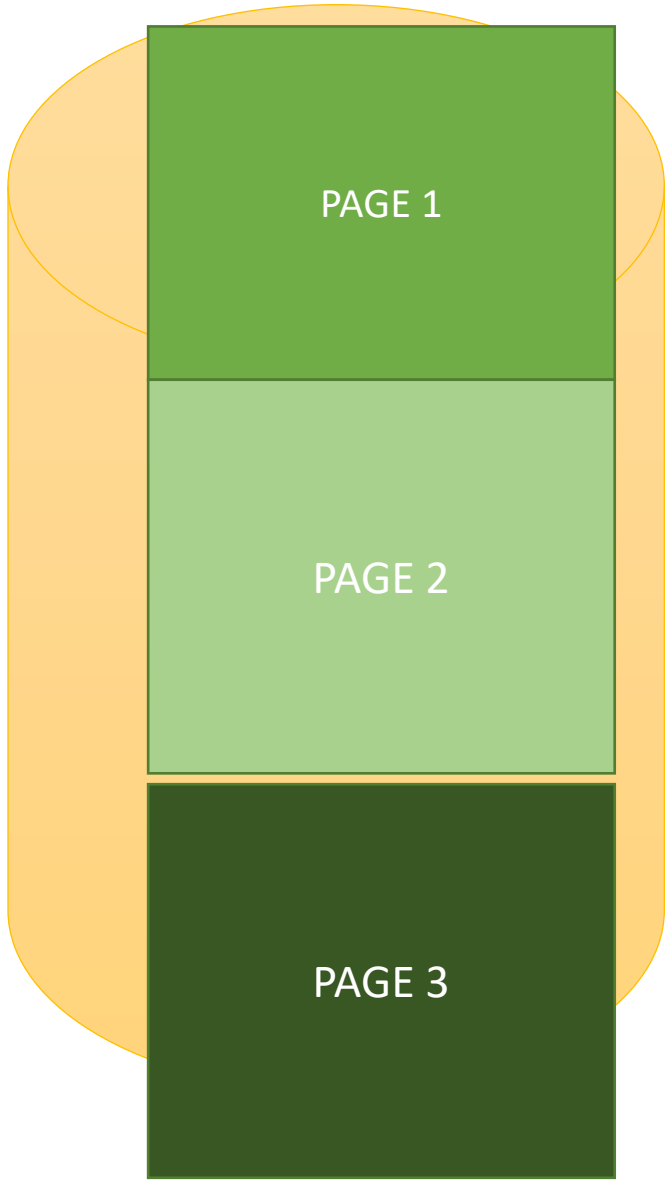
Demand Paging : Thrashing

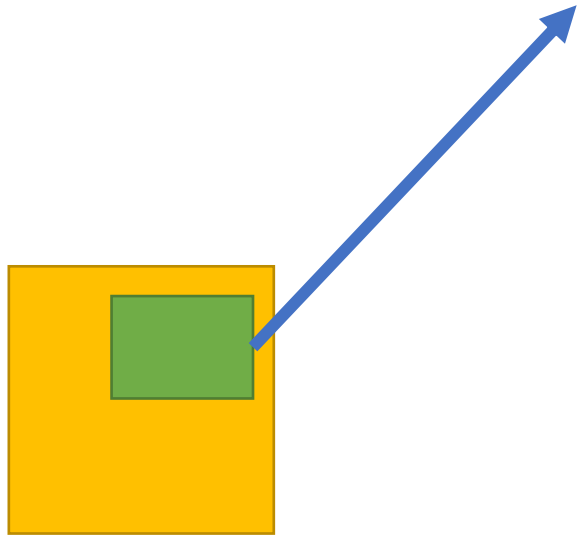
- This happens due to **inefficient Page replacement algorithms**



Demand Paging : Thrashing

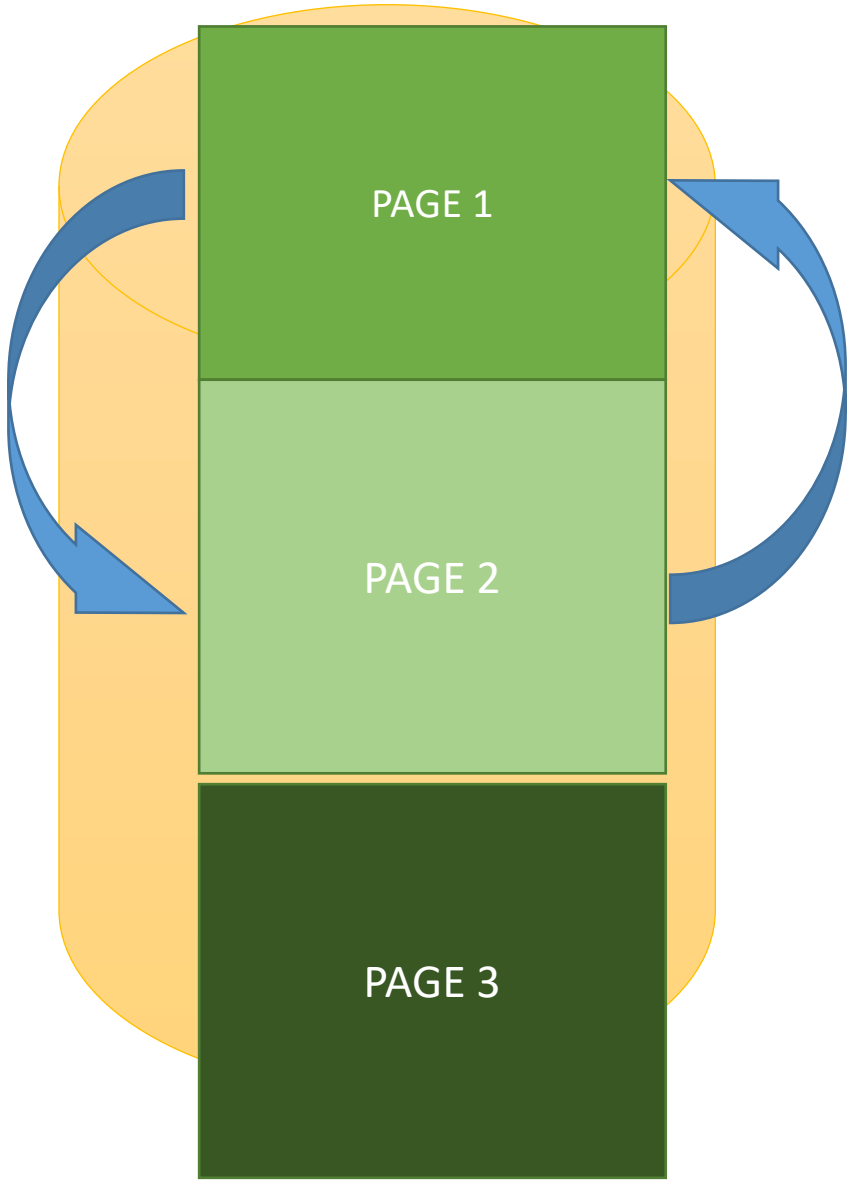
- This happens due to inefficient Page replacement algorithms

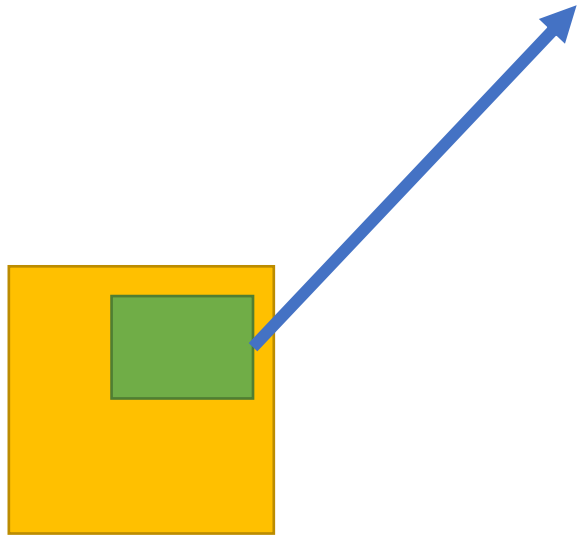




Demand Paging : Thrashing

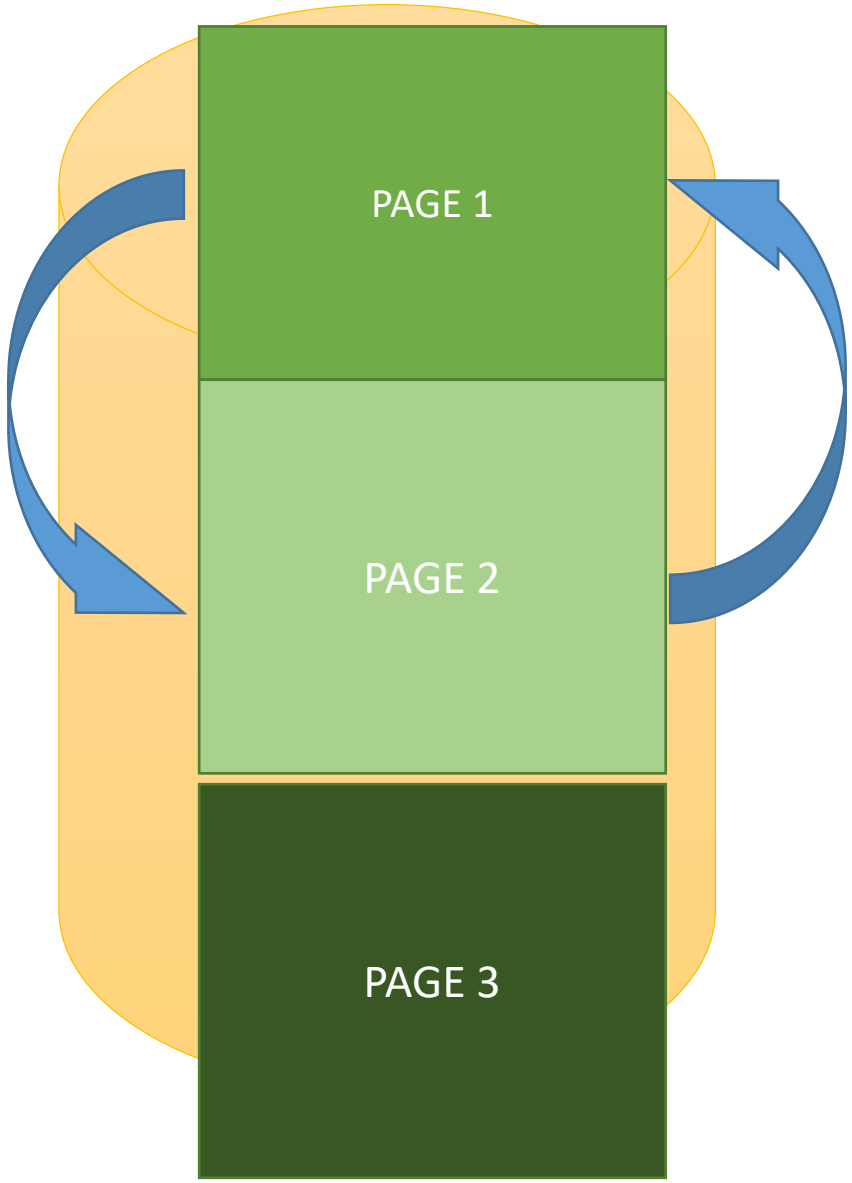
- This happens due to inefficient Page replacement algorithms

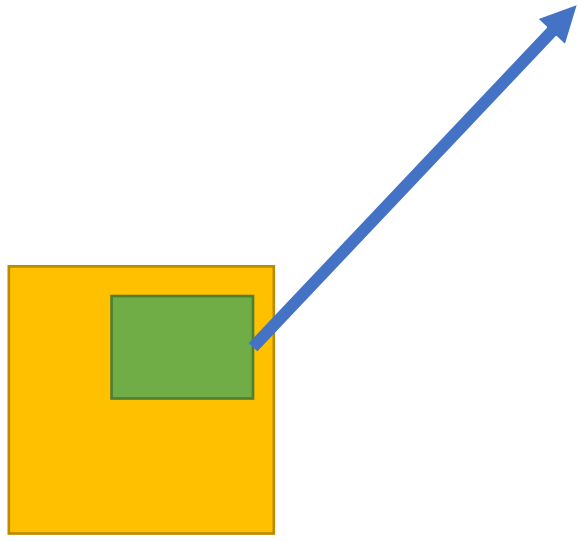




Demand Paging : Thrashing

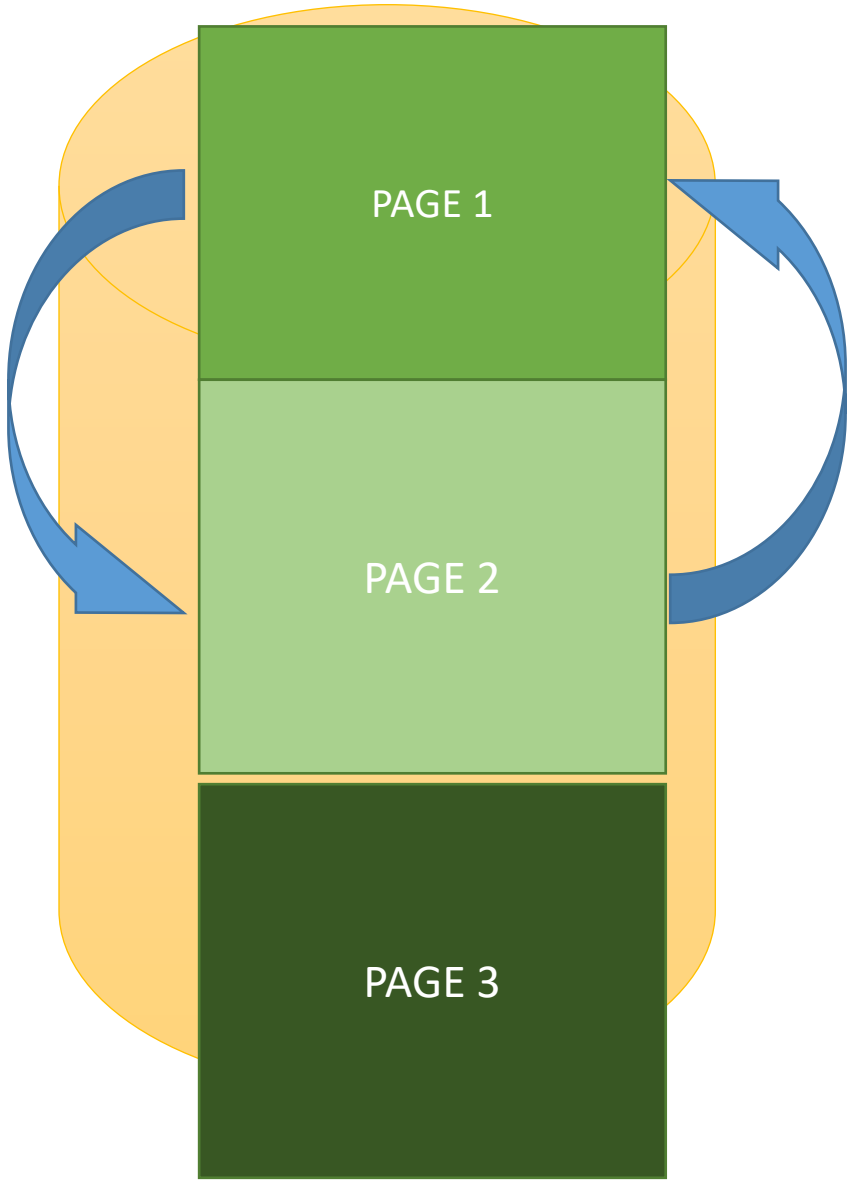
- This happens due to inefficient Page replacement algorithms

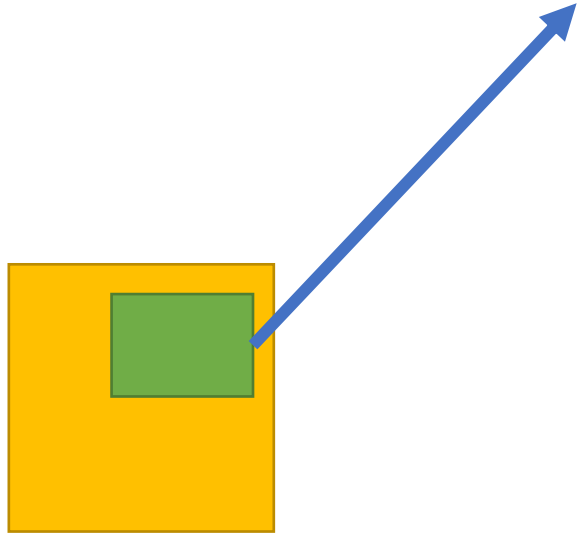


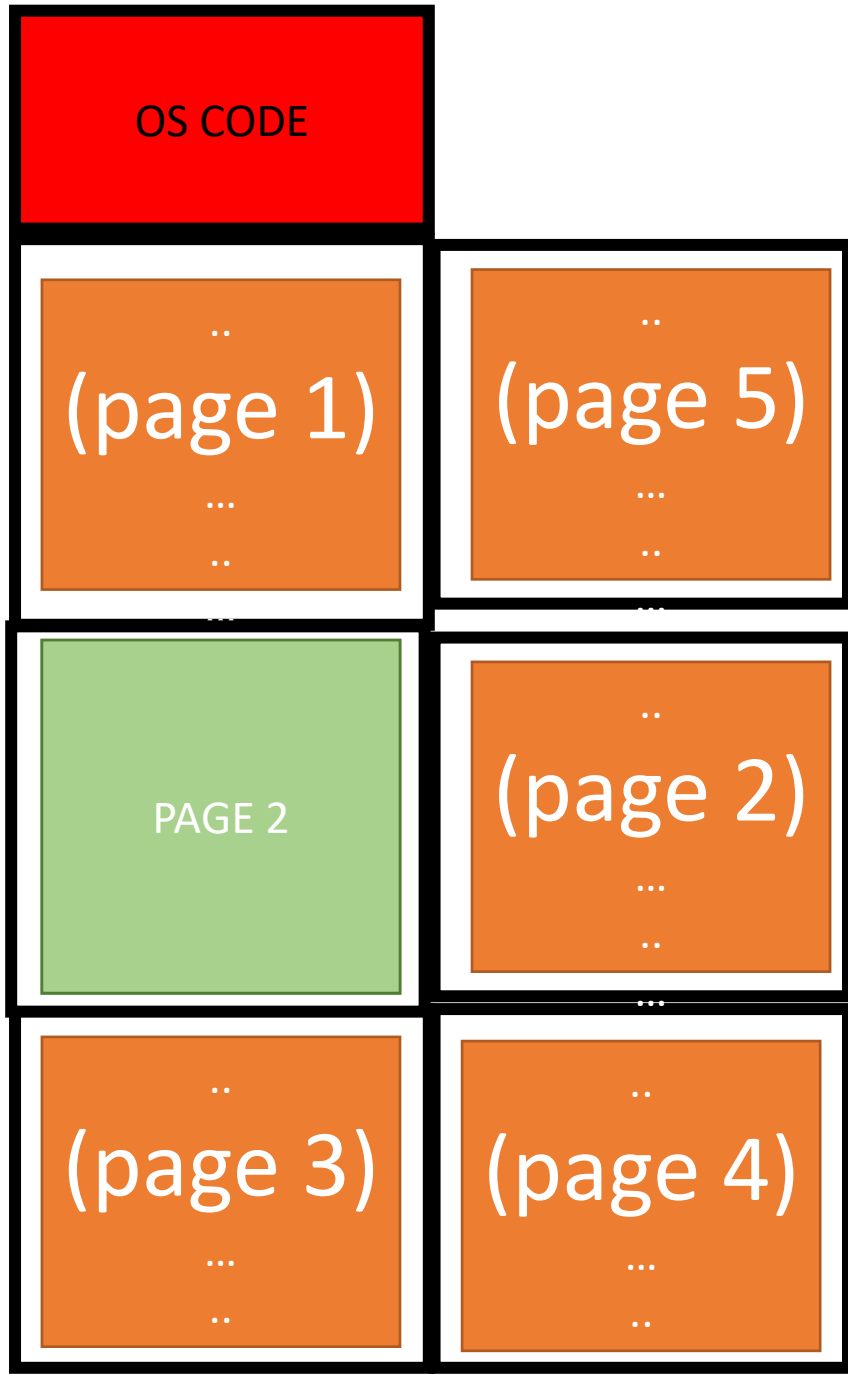


Demand Paging : Thrashing

- This happens due to inefficient Page replacement algorithms

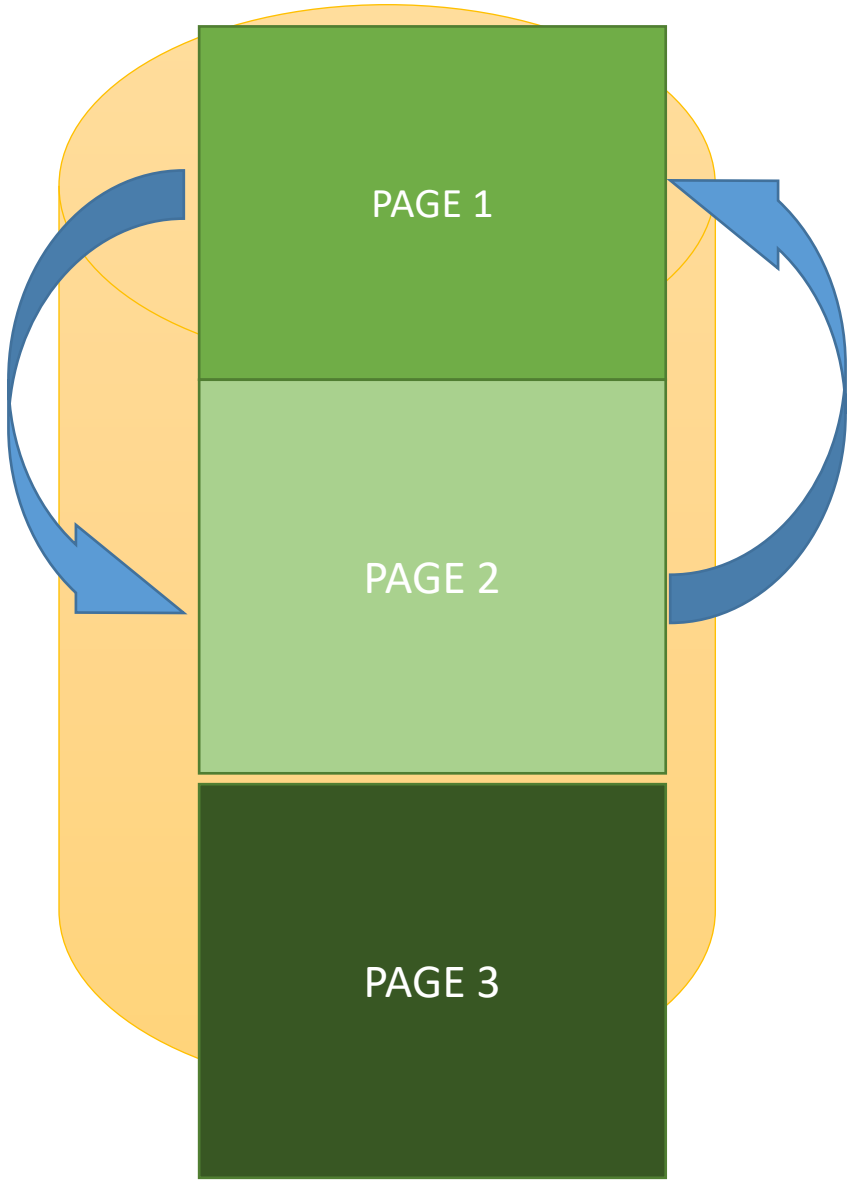


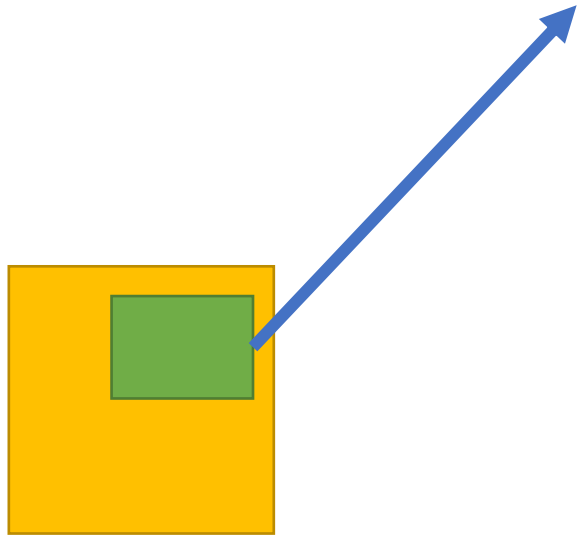




Demand Paging : Thrashing

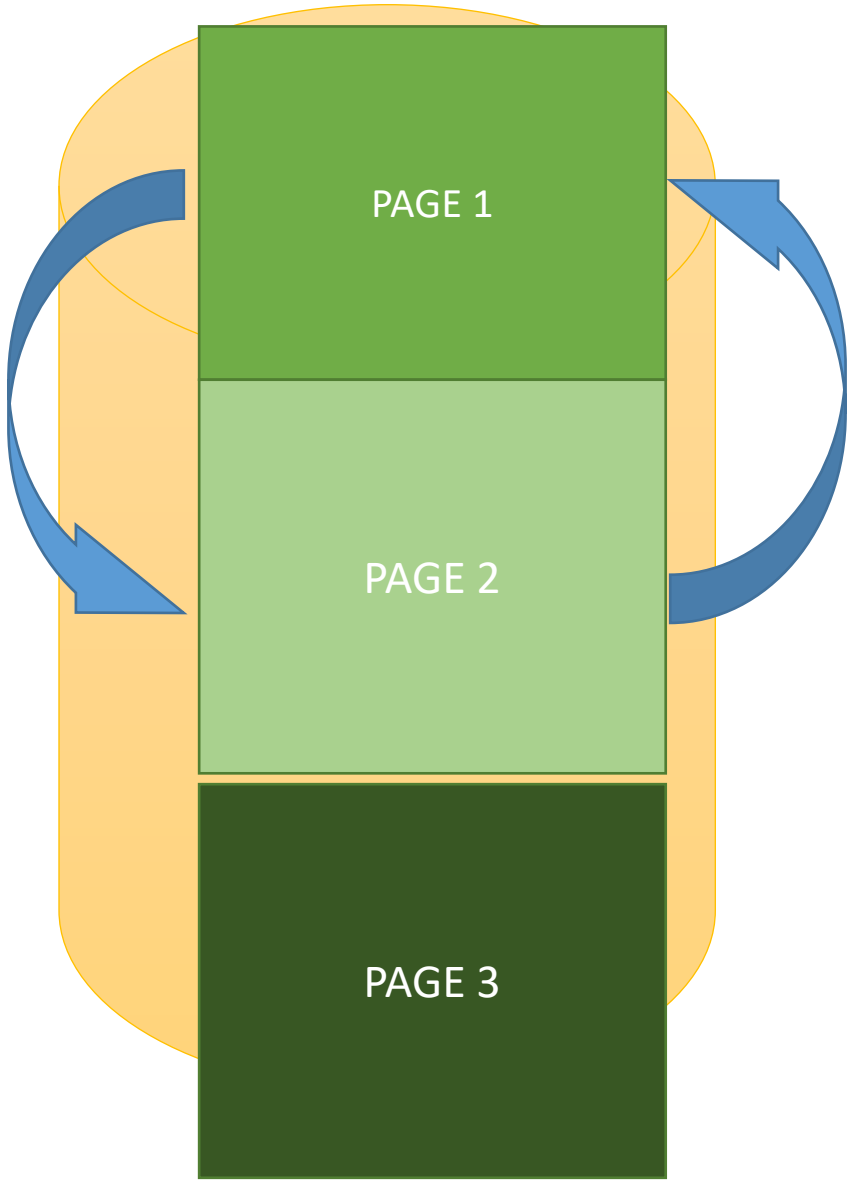
- This happens due to inefficient Page replacement algorithms



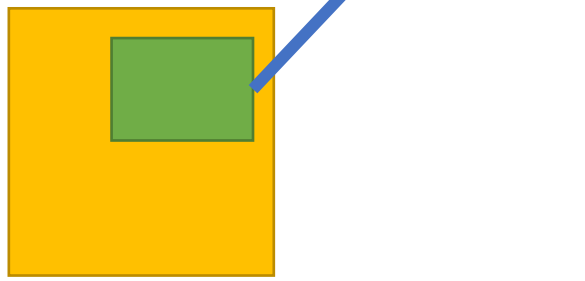


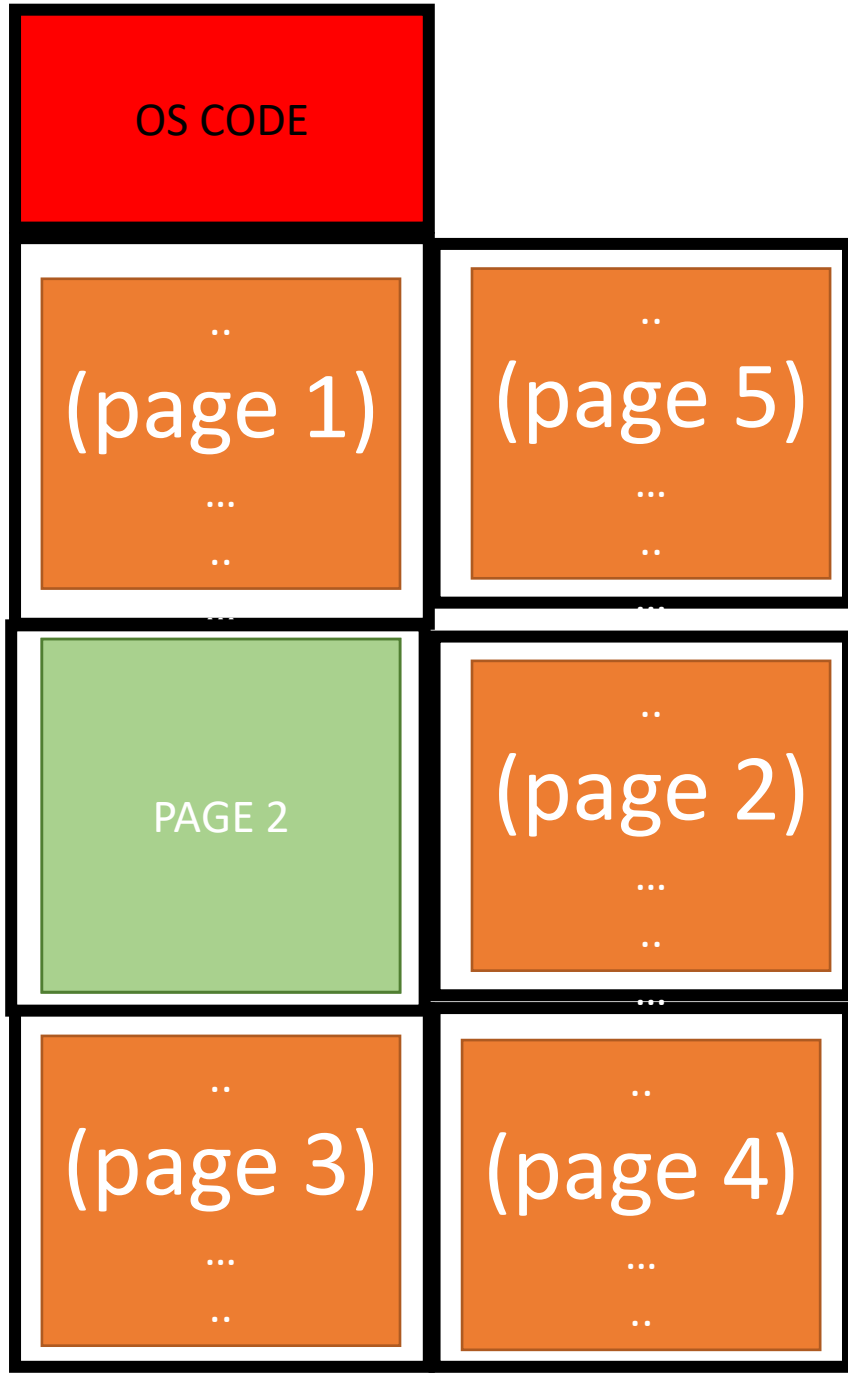
Demand Paging : Thrashing

- This happens due to inefficient Page replacement algorithms



Thrashing:
Results in
More **memory**
Read write operations
And hence **slows down**
The system





4.2 Improving Paging

Inverted Page Table

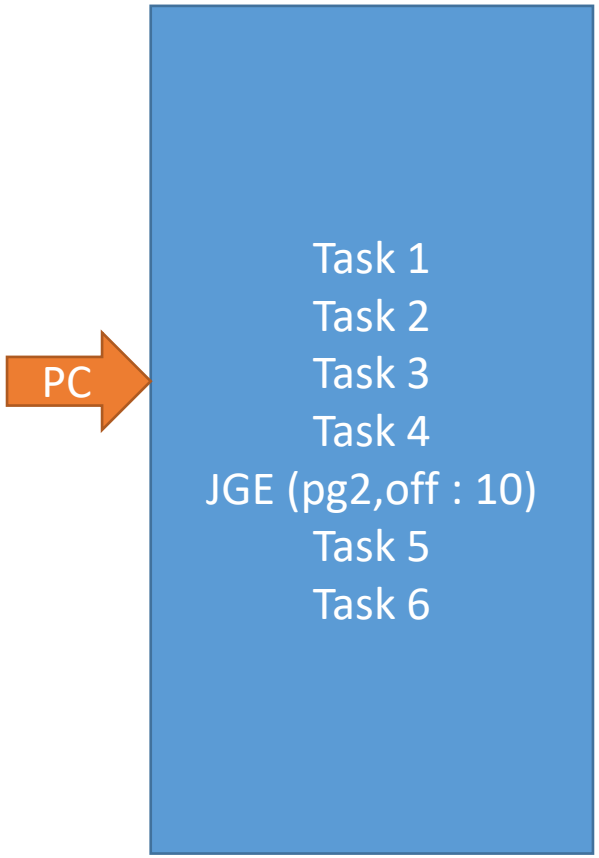
Translation Lookaside Buffer

Combining TLB with Cache

Inverted Page Table

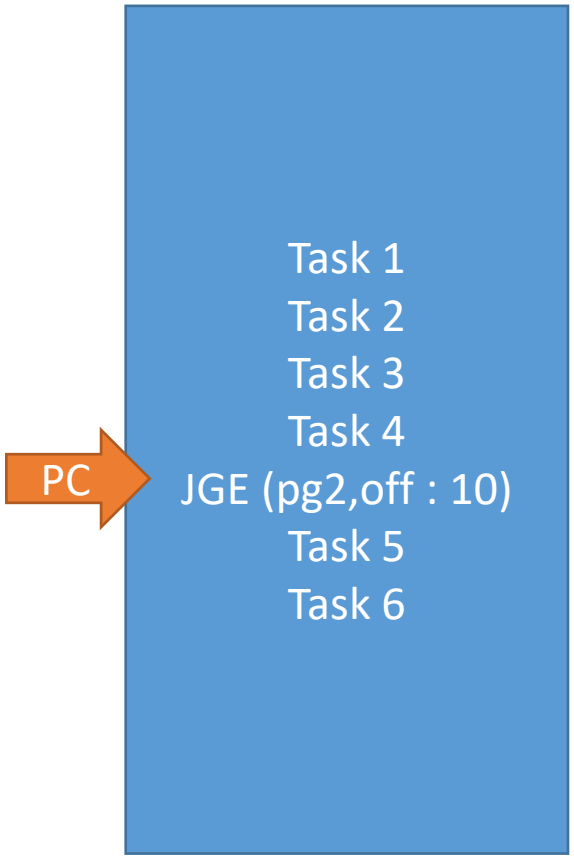
- So far we have seen that:
  - If there are **n processes**, then there will be **n corresponding page tables**
  - This takes up a **lot** of space in the **memory** of OS.
- Solution for this:
  - A new kind of table
  - Known as : **Inverted page table**
- Characteristics of Inverted Page Table
  - **Page** to **Frame** mapping
  - **Process** ID tracking (this feature allows the use of **only 1 table**)

Inverted Page Table



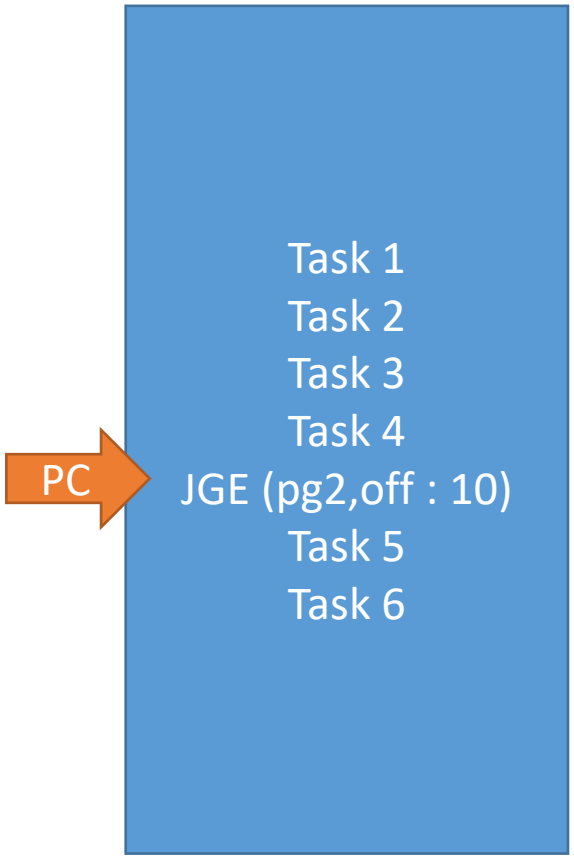
This is the page 1 of a process with process ID 1

Inverted Page Table



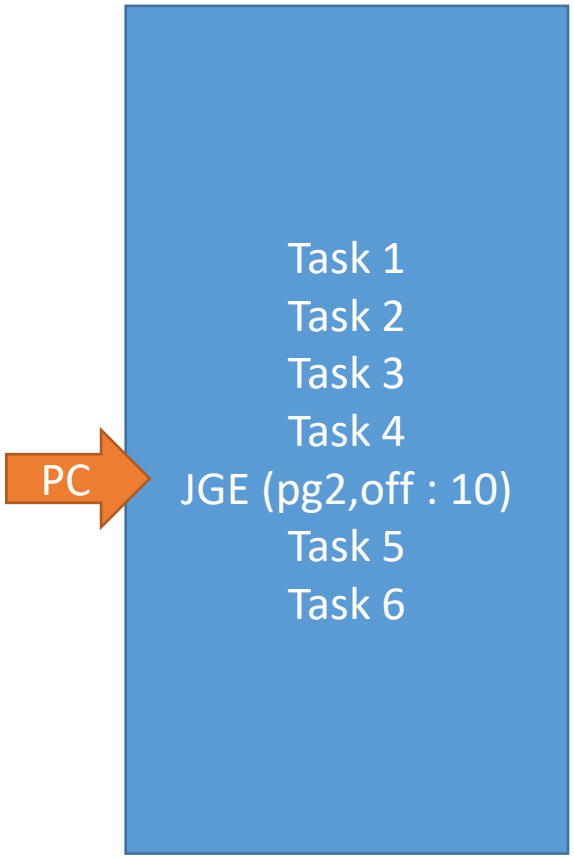
- The processor will shift control to OS.
- OS code will contain a **hash function**

Inverted Page Table

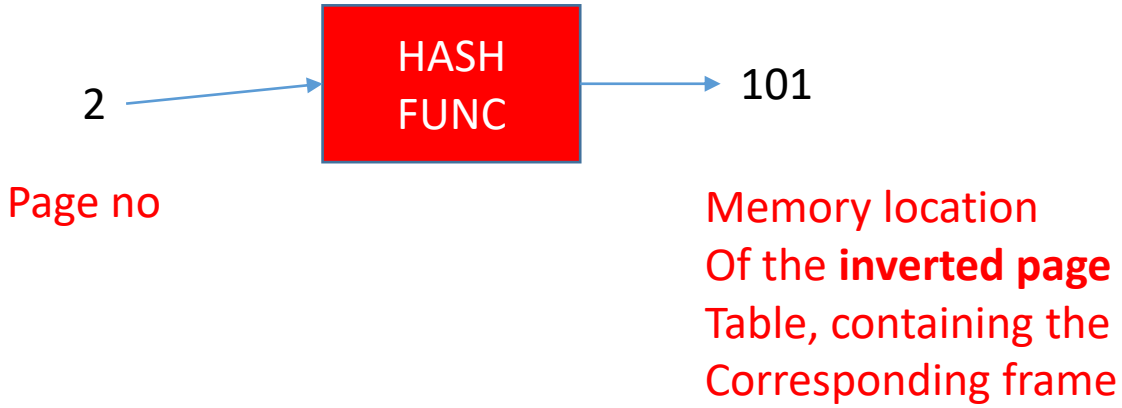


- The processor will shift control to OS.
- OS code will contain a hash function

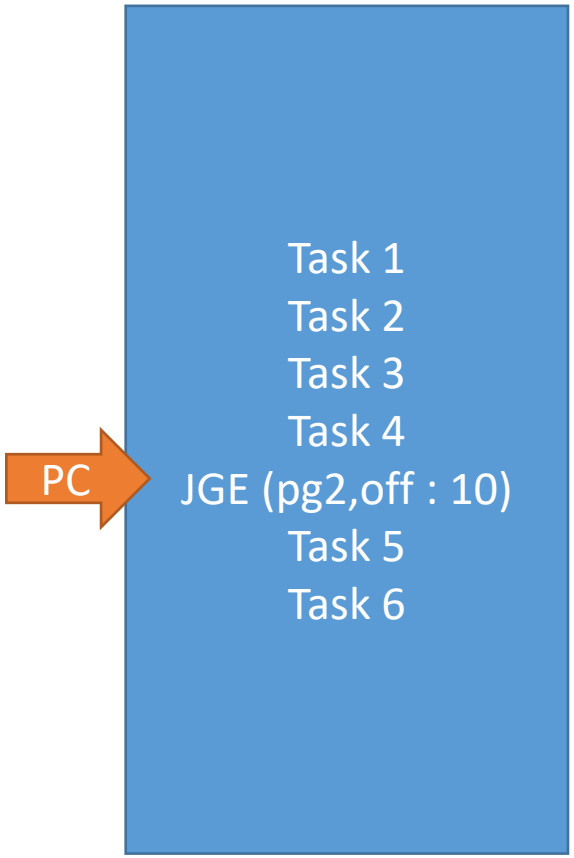
Inverted Page Table



- The processor will shift control to OS.
- OS code will contain a hash function

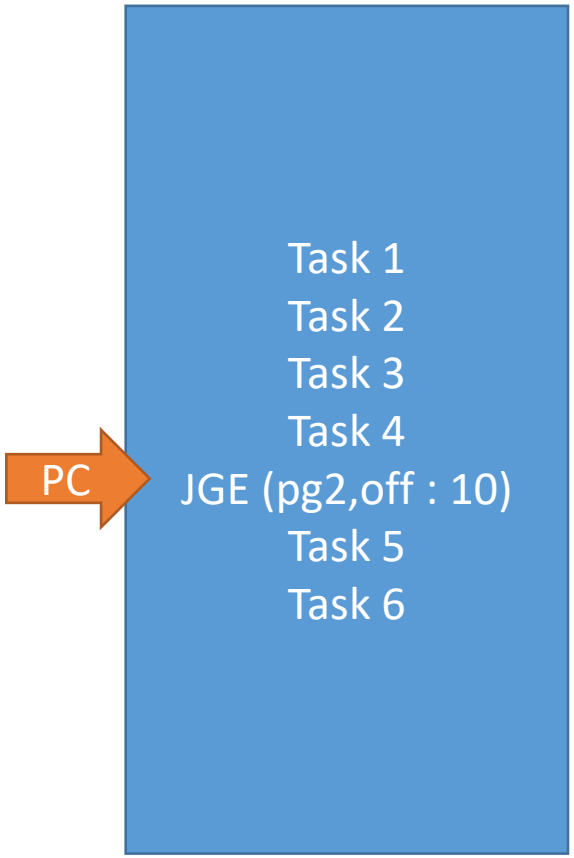


Inverted Page Table



OS will
Use this
Value of **101**
And **search**
The **inverted**
Page table

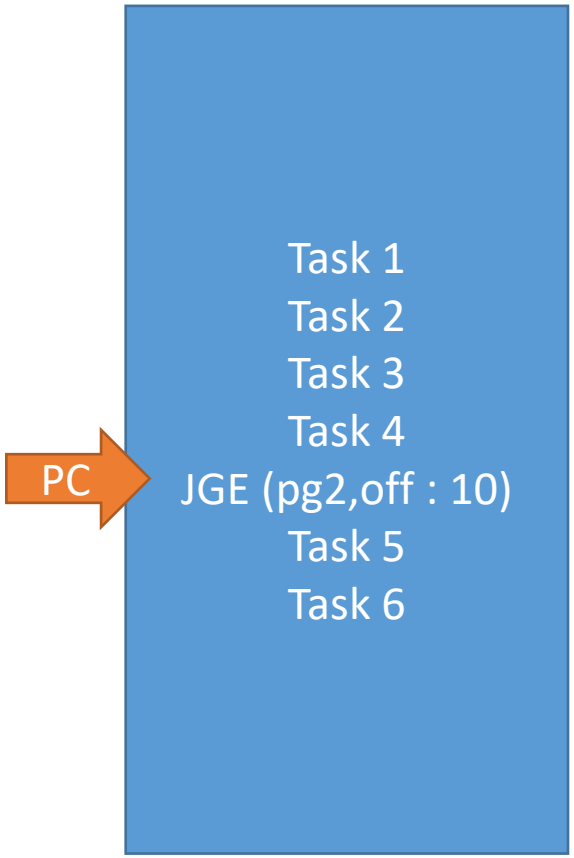
Inverted Page Table



OS will
Use this
Value of 101
And search
The inverted
Page table

Memory Locn	Process ID	Frame	Chain
101	1	20	-
102			
103			
104			
105			
106			
107			

Inverted Page Table



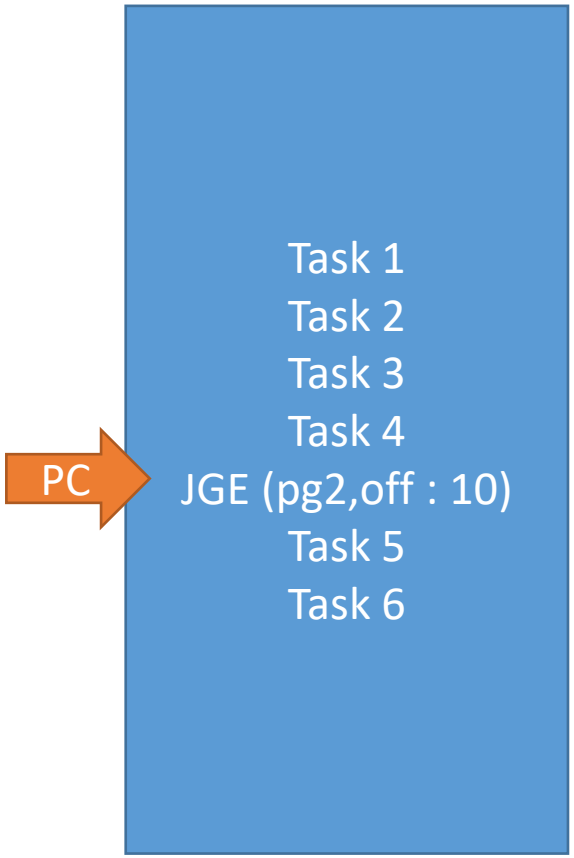
OS will
Use this
Value of 101
And search
The inverted
Page table

So for this process with ID 1, **the page Is located in the 20th frame**

Memory Locn	Process ID	Frame	Chain
101	1	20	-
102			
103			
104			
105			
106			
107			

Inverted Page Table

BUT WHAT IF **THERE WAS NO ENTRY**?

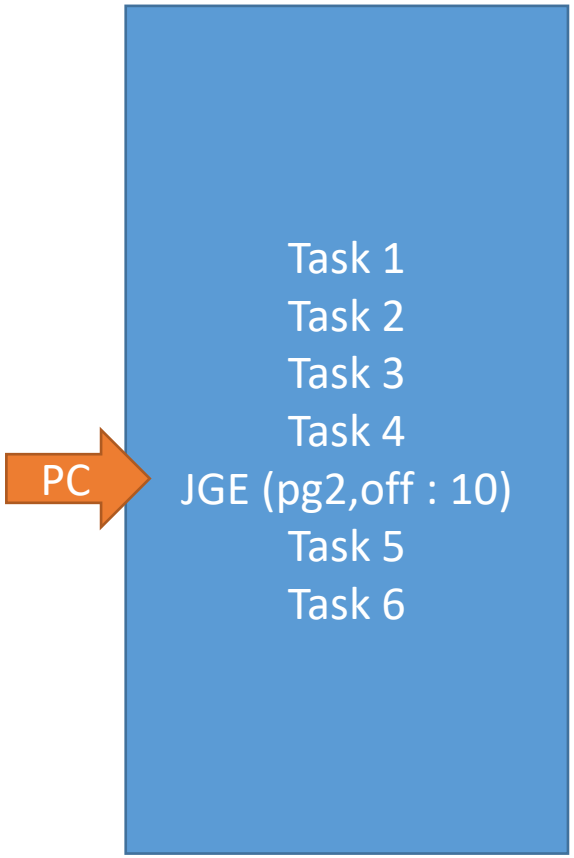


OS will
Use this
Value of 101
And search
The inverted
Page table

Memory Locn	Process ID	Frame	Chain
101			-
102			
103			
104			
105			
106			
107			

Inverted Page Table

BUT WHAT IF THERE WAS NO ENTRY?



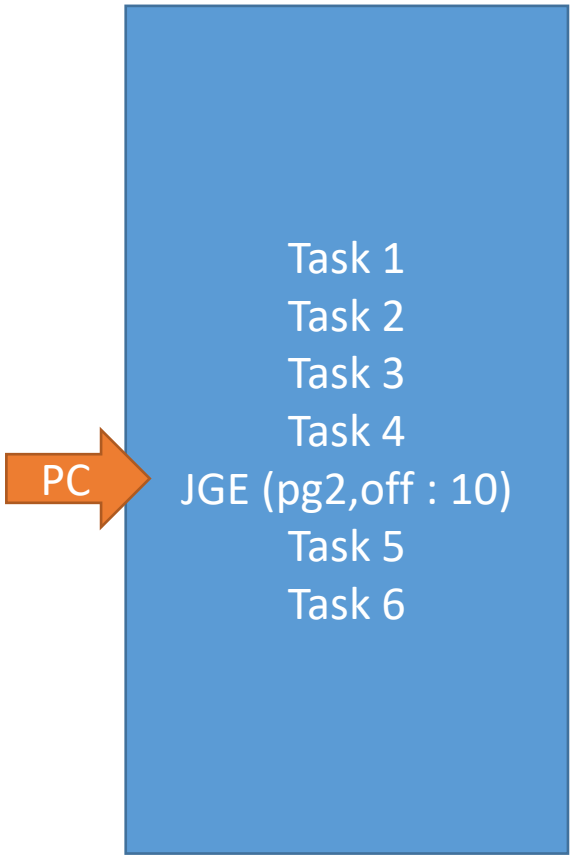
OS will
Use this
Value of 101
And search
The inverted
Page table

Memory Locn	Process ID	Frame	Chain
101			-
102			
103			
104			
105			
106			
107			

It would mean that, the **page 2 is**
Not in the memory yet

Inverted Page Table

BUT WHAT IF THERE WAS NO ENTRY?



OS will
Use this
Value of 101
And search
The inverted
Page table

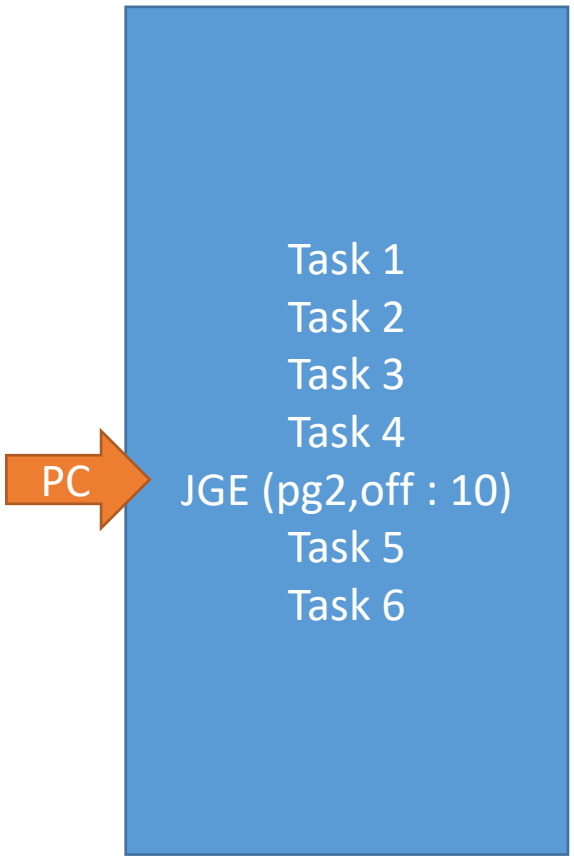
Memory Locn	Process ID	Frame	Chain
101			-
102			
103			
104			
105			
106			
107			

It would mean that, the page 2 is
Not in the memory yet

That page will have to brought in

- **Empty frame** –place it there
- No empty frame- use **page replacement algo**

Inverted Page Table



OS will
Use this
Value of 101
And search
The inverted
Page table

BUT WHAT IF THERE WAS NO ENTRY?

Then we would **update the table** with the
Info about the frame it has been placed

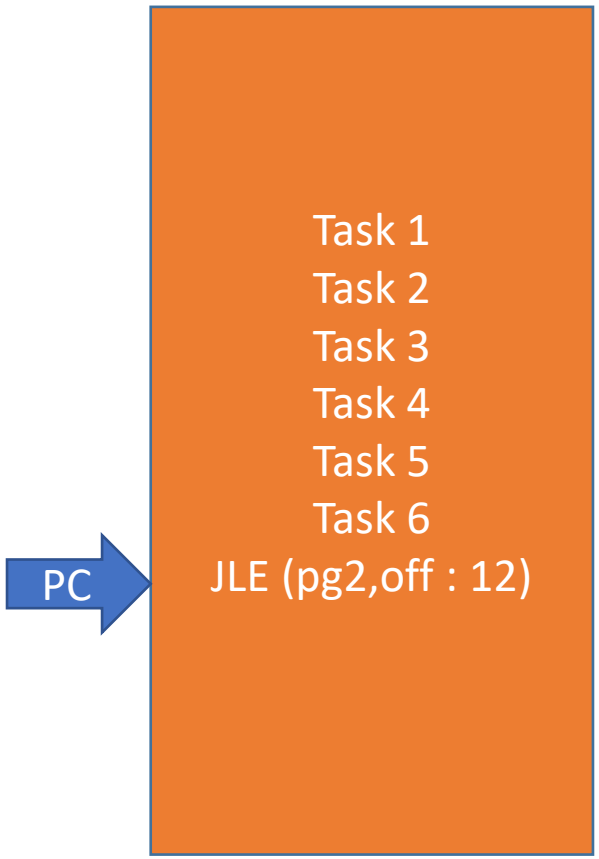
Memory Locn	Process ID	Frame	Chain
101	1	30	--
102			
103			
104			
105			
106			
107			

It would mean that, the page 2 is
Not in the memory yet

That page will have to brought in

- Empty frame –place it there
- No empty frame- use page replacement algo

Inverted Page Table (The Chain field)

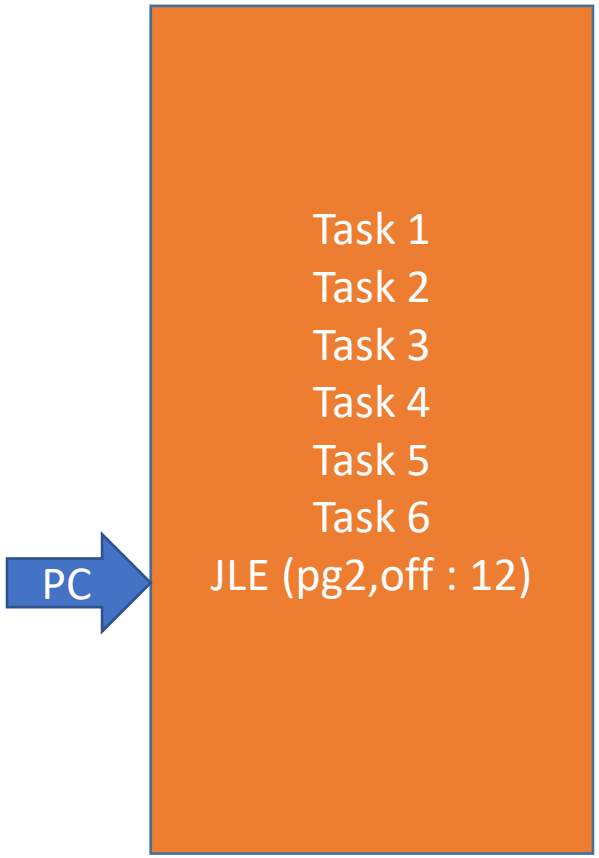


PROCESS WITH ID 2

Memory Locn	Process ID	Frame	Chain
101	1	30	--
102			
103			
104			
105			
106			
107			

Let us assume that its **page 2** is not in the table

Inverted Page Table (The Chain field)



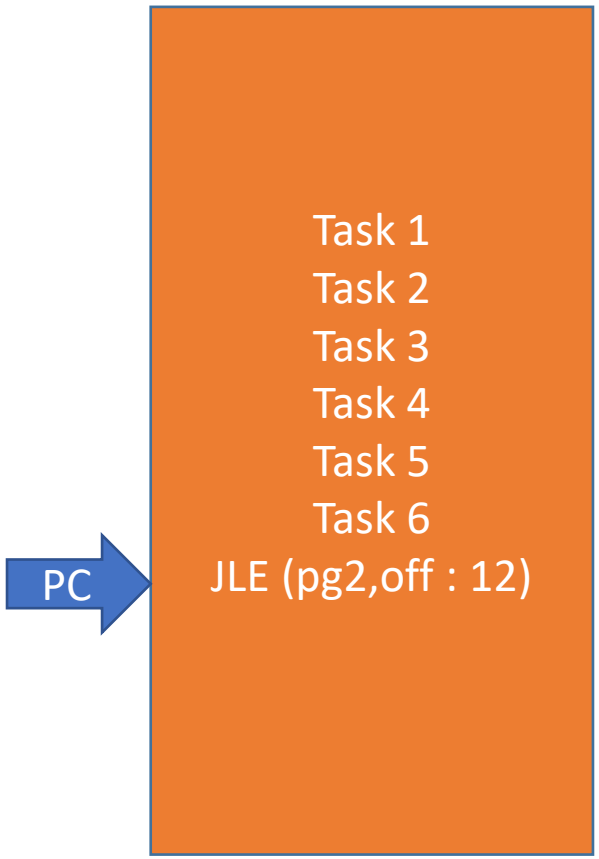
Memory Locn	Process ID	Frame	Chain
101	1	30	--
102			
103			
104			
105			
106			
107			

Page no

The **same hash** function

Memory location
Of the inverted page
Table, containing the
Corresponding frame

Inverted Page Table (The Chain field)



Already occupied
Can't create entry here

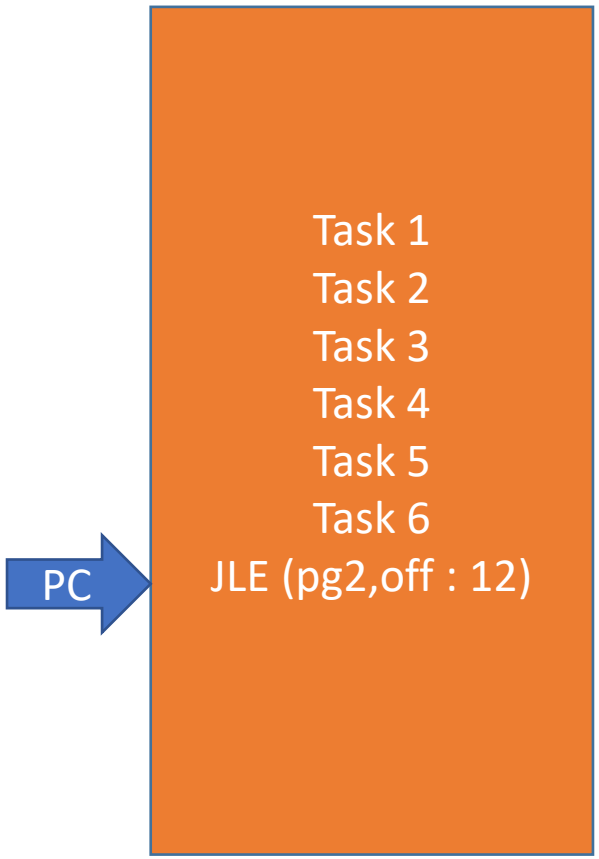
Memory Locn	Process ID	Frame	Chain
101	1	30	--
102			
103			
104			
105			
106			
107			

Page no

The same hash function

Memory location
Of the inverted page
Table, containing the
Corresponding frame

Inverted Page Table (The Chain field)



Search for the **nearest location**, that is **not occupied**
With page frame mapping data

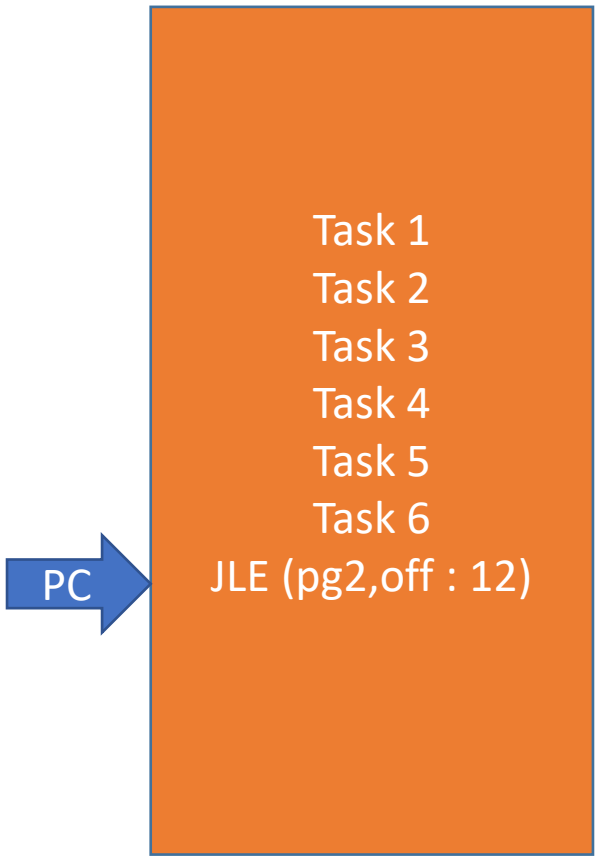
Memory Locn	Process ID	Frame	Chain
101	1	30	--
102			
103			
104			
105			
106			
107			

Page no

The same hash function

Memory location
Of the inverted page
Table, containing the
Corresponding frame

Inverted Page Table (The Chain field)



Search for the nearest location, that is not occupied
With page frame mapping data

Memory Locn	Process ID	Frame	Chain
101	1	30	--
102	occupied		
103	occupied		
104	Occupied		
105	occupied		
106			
107	occupied		

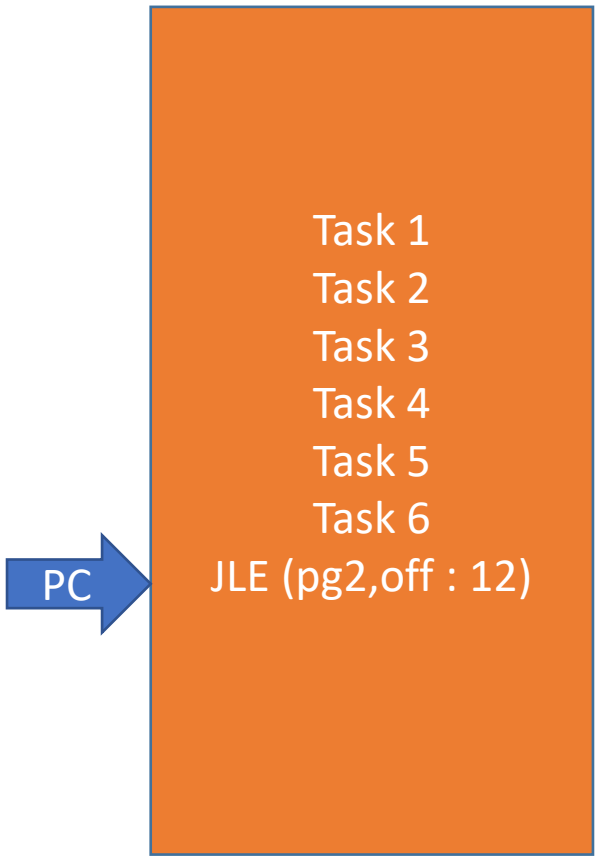
Let us **assume** that **these** rows
Were already occupied

Page no

The same hash function

Memory location
Of the inverted page
Table, containing the
Corresponding frame

Inverted Page Table (The Chain field)



Let us assume that these rows
Were already occupied

Search for the nearest location, that is not occupied
With page frame mapping data

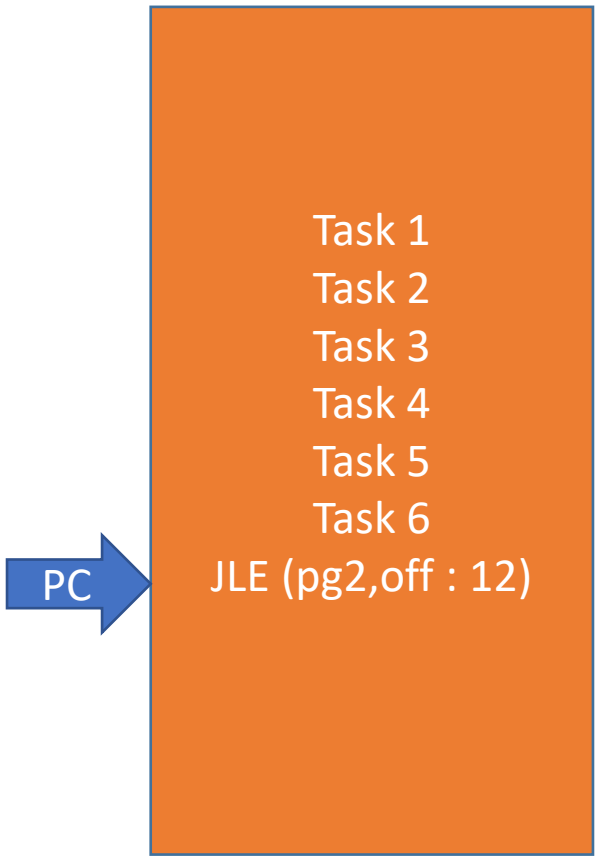
Memory Locn	Process ID	Frame	Chain
101	1	30	--
102	occupied		
103	occupied		
104	Occupied		
105	occupied		
106			
107	occupied		

Page no

The same hash function

Memory location
Of the inverted page
Table, containing the
Corresponding frame

Inverted Page Table (The Chain field)



Let us assume that these rows
Were already occupied

Search for the nearest location, that is not occupied
With page frame mapping data

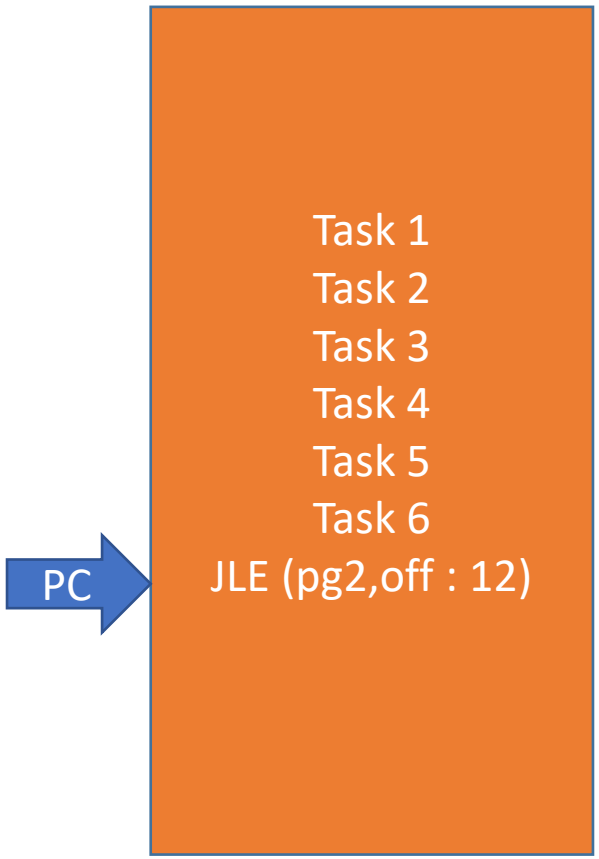
Memory Locn	Process ID	Frame	Chain
101	1	30	106
102	occupied		
103	occupied		
104	Occupied		
105	occupied		
106	2	45	-
107	occupied		

Page no

The same hash function

Memory location
Of the inverted page
Table, containing the
Corresponding frame

Inverted Page Table (The Chain field)



Let us assume that these rows
Were already occupied

Search for the nearest location, that is not occupied
With page frame mapping data

Memory Locn	Process ID	Frame	Chain
101	1	30	106
102	occupied		
103	occupied		
104	Occupied		
105	occupied		
106	2	45	-
107	occupied		

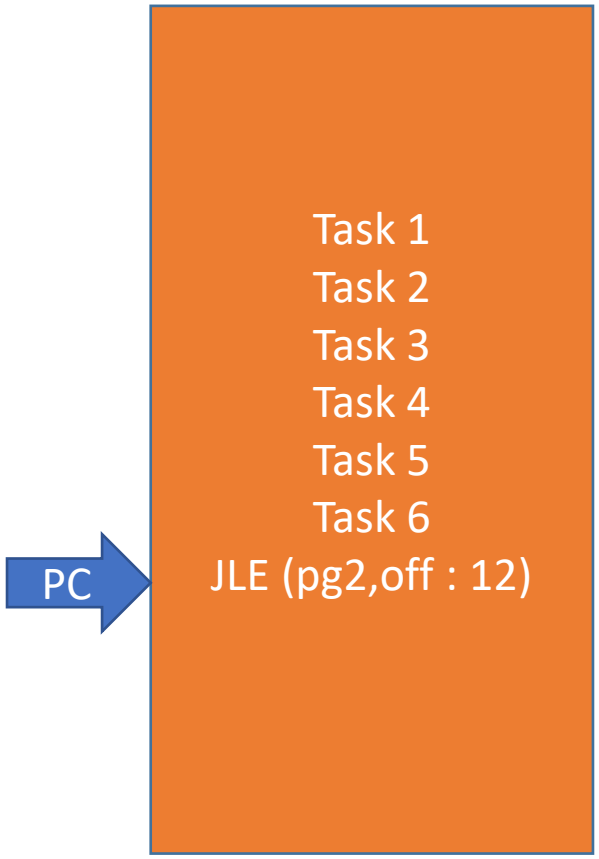
Page no

The same hash function

Memory location
Of the inverted page
Table, containing the
Corresponding frame

The page
Rep algo placed
2nd page of **process 2**
In the 45th frame

Inverted Page Table (The Chain field)

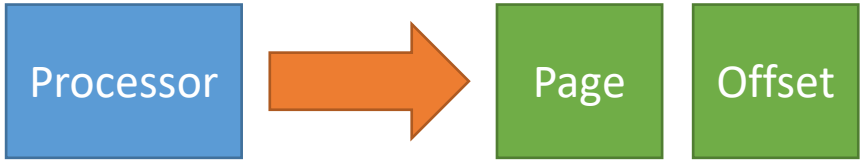


Memory Locn	Process ID	Frame	Chain
101	1	30	106
102	occupied		
103	occupied		
104	Occupied		
105	occupied		
106	2	45	-
107	occupied		

Next time when **page 2** for **process 2** will be searched. The **entry will be available** with the frame info

Translation Lookaside Buffer

Translation Lookaside Buffer

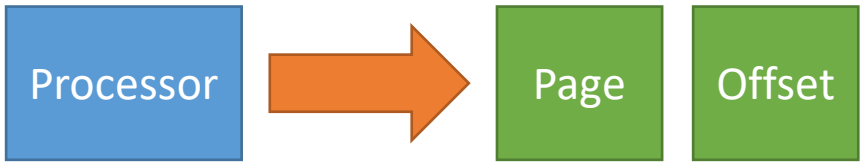


The processor **Requires Instruction/ Data** from this location

ID	Page	Frame

ID	Frame	Chain

Translation Lookaside Buffer



TLB: A h/w with the **recently used** Pages and corresponding mapping

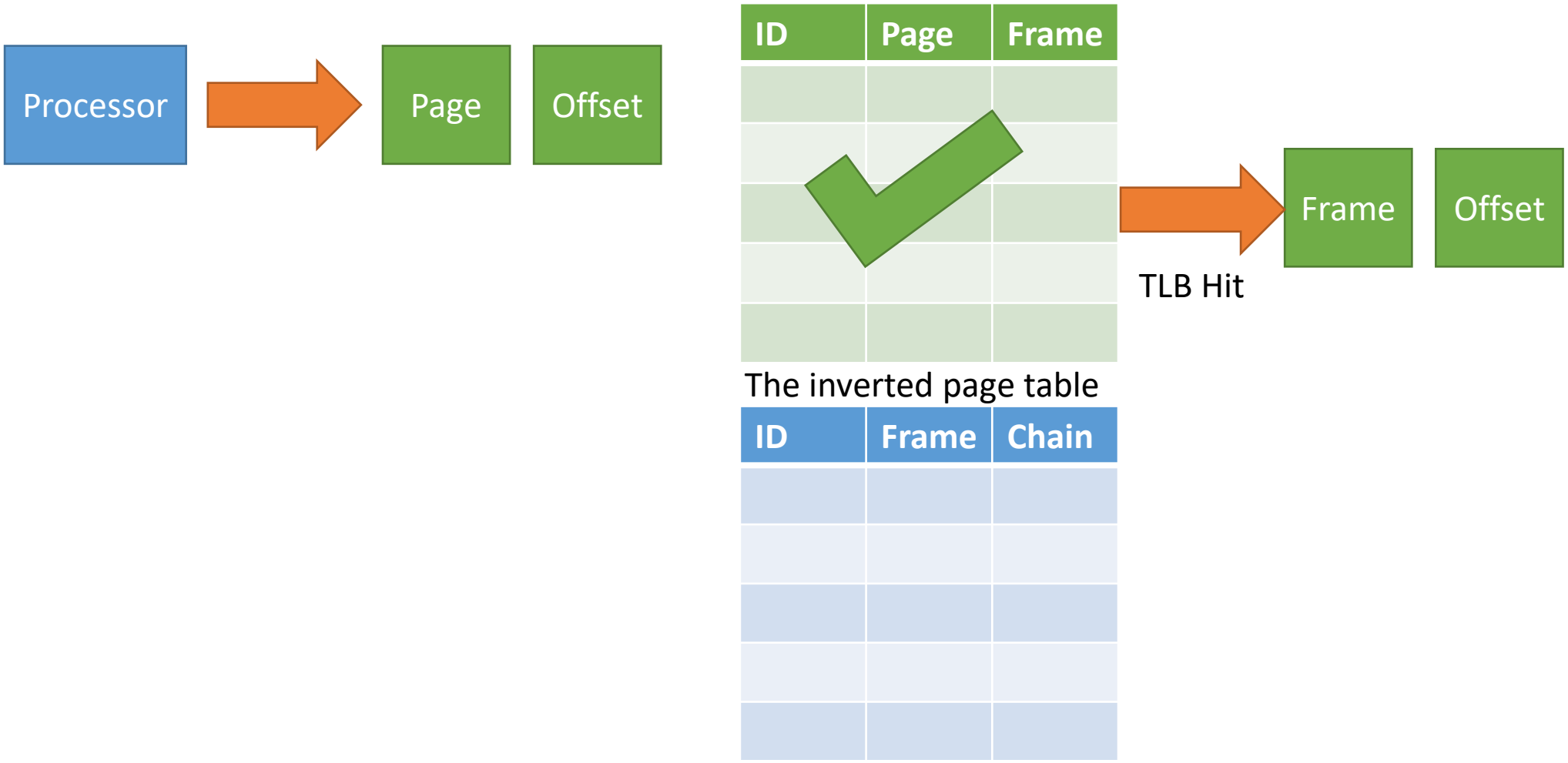
ID	Page	Frame

The **inverted page table**

ID	Frame	Chain

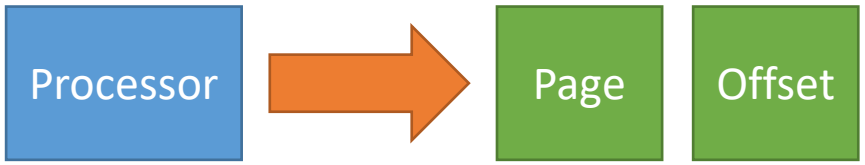
Translation Lookaside Buffer

TLB: A h/w with the recently used
Pages and corresponding mapping



Translation Lookaside Buffer

TLB: A h/w with the recently used
Pages and corresponding mapping



ID	Page	Frame

The inverted page table

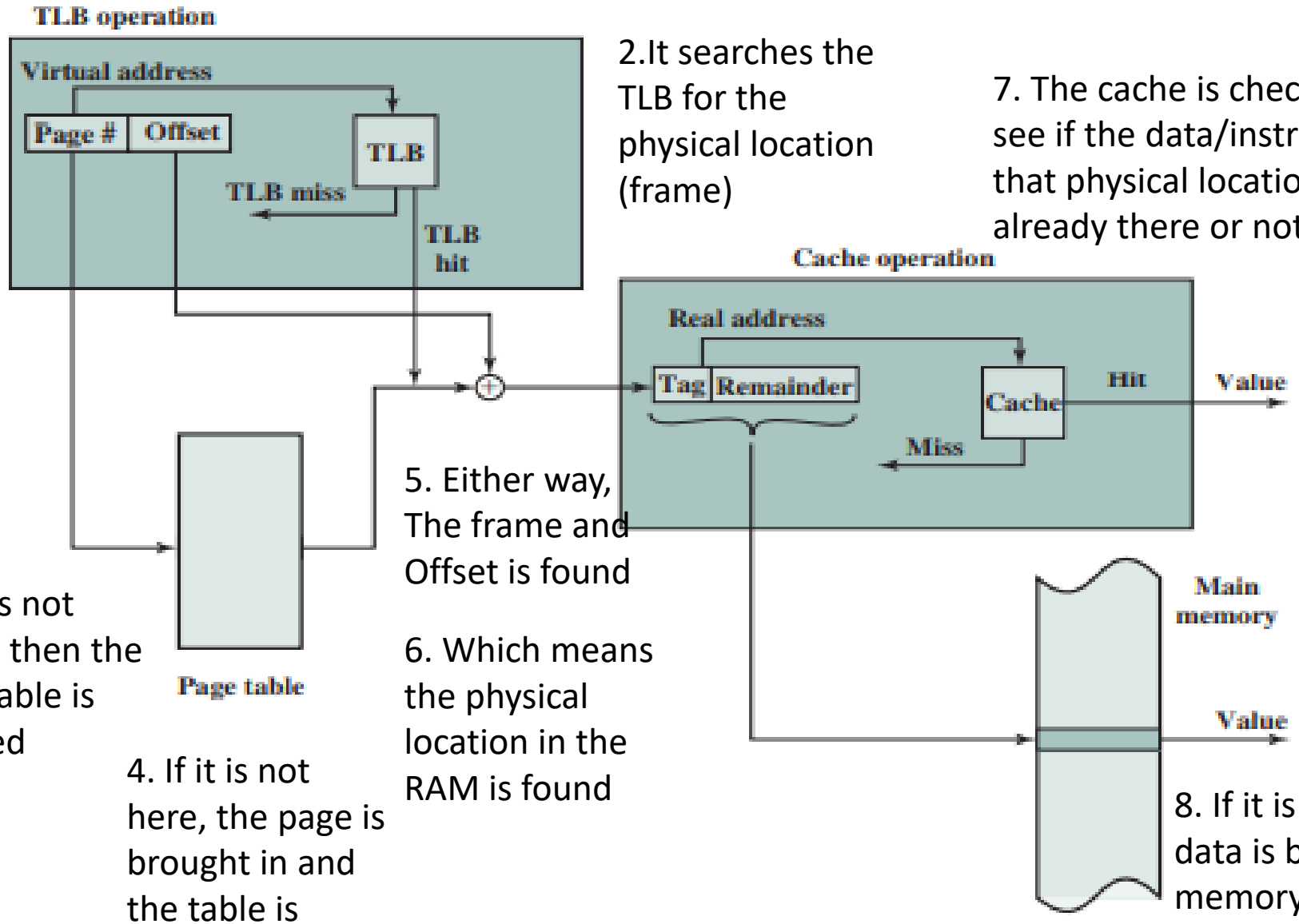
ID	Frame	Chain

TLB Miss

BASICALLY
TLB is a like a **cache**
For page frame mapping

Combining TLB with Cache

1.The processor requires a data from a particular page and offset



2.It searches the TLB for the physical location (frame)

3.If it is not found, then the page table is checked

4. If it is not here, the page is brought in and the table is updated

5. Either way, The frame and Offset is found

6. Which means the physical location in the RAM is found

7. The cache is checked to see if the data/instruction of that physical location is already there or not

8. If it is not there, then the data is brought in from the memory

School Teacher :- You will learn this in college
Professor :- You have learnt this in your school
Me :-

Comp Archi

OS Teacher

CSE 21

~~School~~ Teacher :- You will learn this in college

~~Professo~~or :- You have learnt this in your school

Me :-

Comp Archi

OS Teacher

CSE 21

~~School~~ Teacher :- You will learn this in college

~~Professo~~or :- You have learnt this in your sch~~o~~l

Me :-

THE END

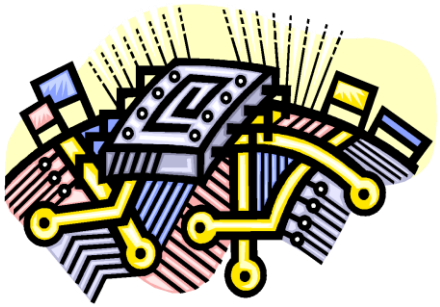
CSE 213
Computer Architecture

Lecture 9: Computer Arithmetic

Military Institute of Science
and Technology

Arithmetic & Logic Unit (ALU)

- Part of the computer that actually performs arithmetic and logical operations on data
- All of the other elements of the computer system are there mainly to bring data into the ALU for it to process and then to take the results back out
- Based on the use of simple digital logic devices that can store binary digits and perform simple Boolean logic operations



ALU Inputs and Outputs

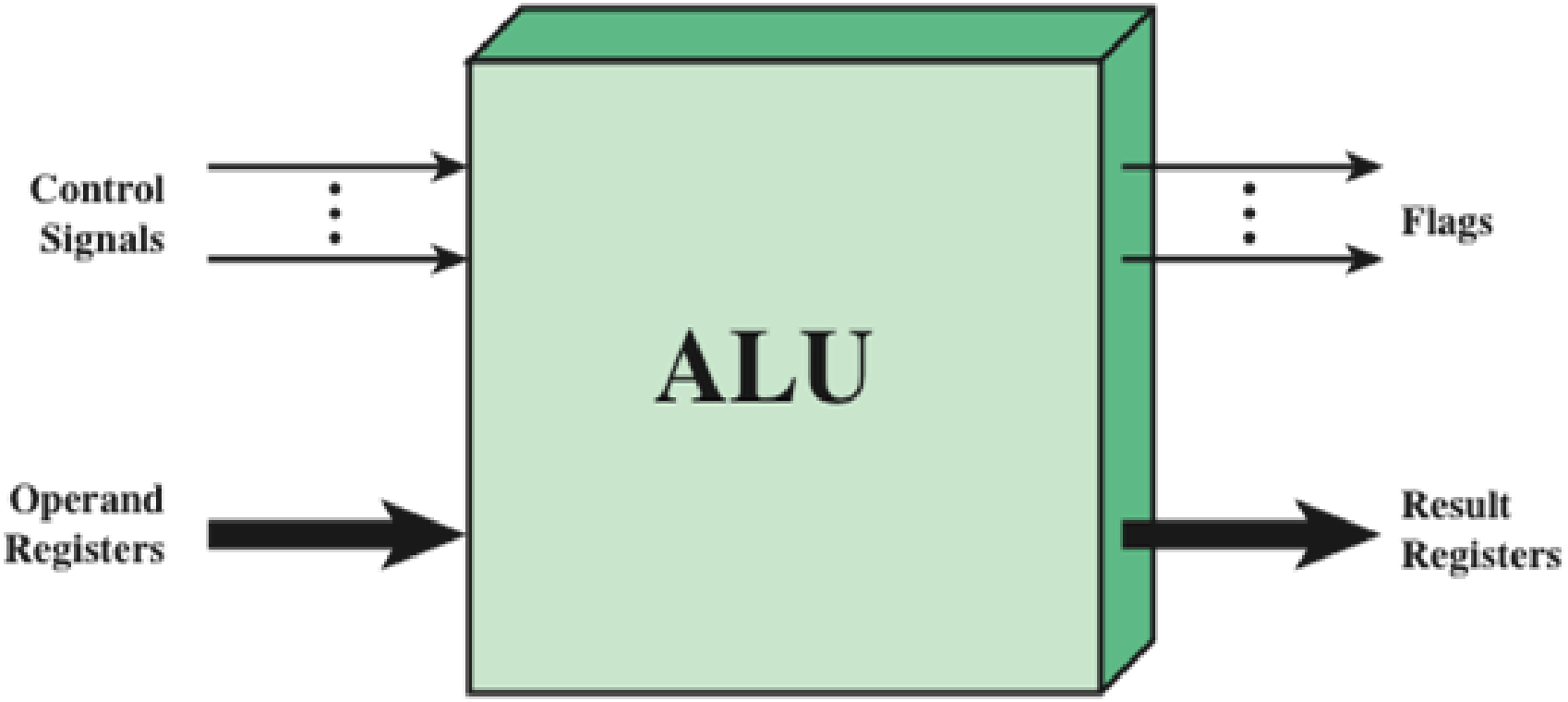
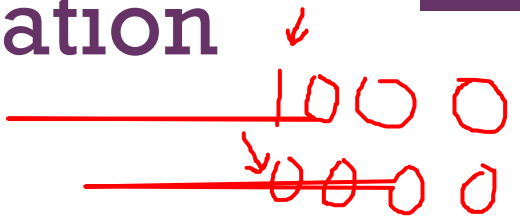


Figure 10.1 ALU Inputs and Outputs

Integer Representation

- In the binary number system arbitrary numbers can be represented with:
  - The digits zero and one
  - The minus sign (for negative numbers)
  - The period, or ***radix point*** (for numbers with a fractional component)
- For purposes of computer storage and processing we do not have the benefit of special symbols for the minus sign and radix point
- Only binary digits (0,1) may be used to represent numbers

Sign-Magnitude Representation



There are several alternative conventions used to represent negative as well as positive integers

- All of these alternatives involve treating the most significant (leftmost) bit in the word as a sign bit
- If the sign bit is 0 the number is positive
- If the sign bit is 1 the number is negative

Sign-magnitude representation is the simplest form that employs a sign bit

Drawbacks:

- Addition and subtraction require a consideration of both the signs of the numbers and their relative magnitudes to carry out the required operation
- There are two representations of 0

Because of these drawbacks, sign-magnitude representation is rarely used in implementing the integer portion of the ALU

Twos Complement Representation

- Uses the most significant bit as a sign bit
- Differs from sign-magnitude representation in the way that the other bits are interpreted

Range	-2_{n-1} through $2_{n-1} - 1$
Number of Representations of Zero	One
Negation	Take the Boolean complement of each bit of the corresponding positive number, then add 1 to the resulting bit pattern viewed as an unsigned integer.
Expansion of Bit Length	Add additional bit positions to the left and fill in with the value of the original sign bit.
Overflow Rule	If two numbers with the same sign (both positive or both negative) are added, then overflow occurs if and only if the result has the opposite sign.
Subtraction Rule	To subtract B from A, take the twos complement of B and add it to A.

Table 10.1 Characteristics of Twos Complement Representation and Arithmetic

Table 10.2
Alternative Representations for 4-Bit Integers

Decimal Representation	Sign-Magnitude Representation	Twos Complement Representation	Biased Representation
+8	—	—	1111
+7	0111	0111	1110
+6	0110	0110	1101
+5	0101	0101	1100
+4	0100	0100	1011
+3	0011	0011	1010
+2	0010	0010	1001
+1	0001	0001	1000
+0	0000	0000	0111
–0	1000	—	—
–1	1001	1111	0110
–2	1010	1110	0101
–3	1011	1101	0100
–4	1100	1100	0011
–5	1101	1011	0010
–6	1110	1010	0001
–7	1111	1001	0000
–8	—	1000	—

Geometric Depiction of Twos Complement Integers

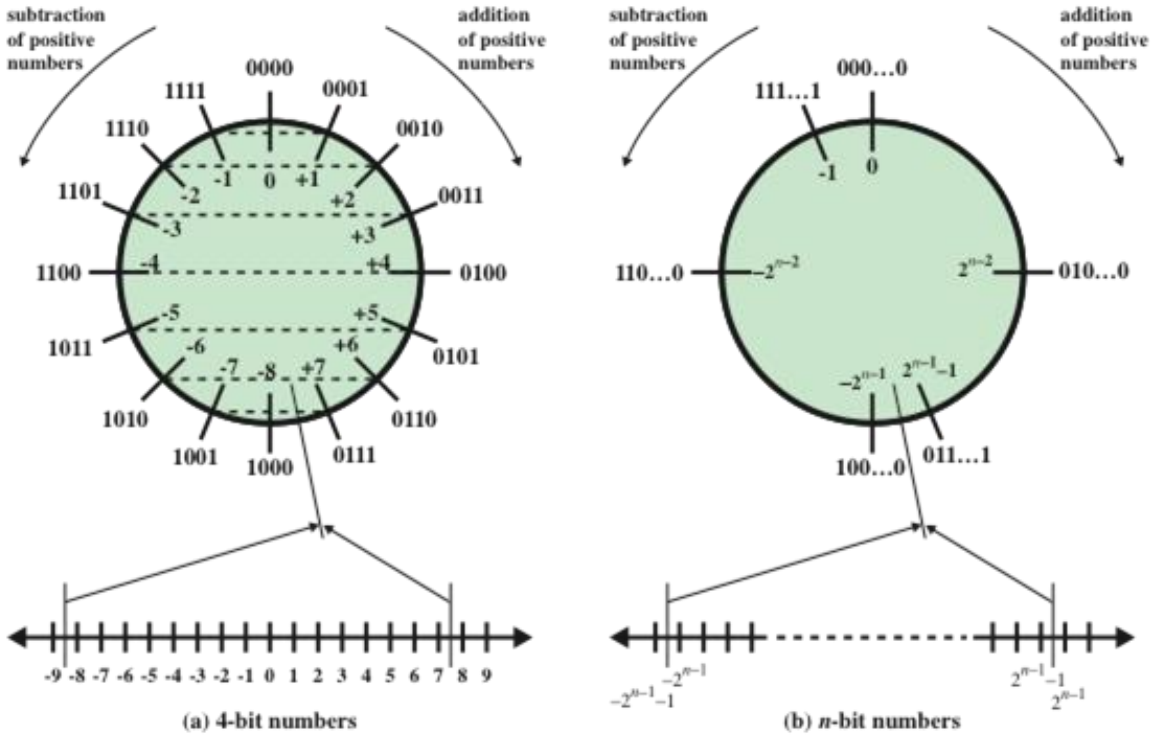
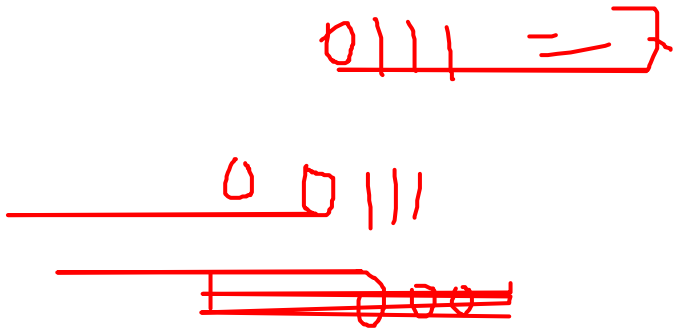


Figure 10.5 Geometric Depiction of Twos Complement Integers

Range Extension



- Range of numbers that can be expressed is extended by increasing the bit length
- In sign-magnitude notation this is accomplished by moving the sign bit to the new leftmost position and fill in with zeros
- This procedure will not work for twos complement negative integers
  - Rule is to move the sign bit to the new leftmost position and fill in with copies of the sign bit
  - For positive numbers, fill in with zeros, and for negative numbers, fill in with ones
  - This is called *sign extension*

Negation

- Twos complement operation
  - Take the Boolean complement of each bit of the integer (including the sign bit)
  - Treating the result as an unsigned binary integer, add 1

```
          +18 = 00010010 (twos complement)
bitwise complement = 11101101
                   +        1
                    11101110 = -18
```

- The negative of the negative of that number is itself:

```
          -18 =  11101110 (twos complement)
bitwise complement =  00010001
                   +        1
                     00010010 = +18
```

Negation Special Case 1

0	=	00000000	(twos complement)
Bitwise complement	=	11111111	
Add 1 to LSB		+ 1	
Result		100000000	

Overflow is ignored, so:

- 0 = 0

Negation Special Case 2

-128	=	10000000	(twos complement)
Bitwise complement	=	01111111	
Add 1 to LSB		+ 1	
Result		10000000	

So:

-(-128) = -128 X

Monitor MSB (sign bit)

It should change during negation

Addition

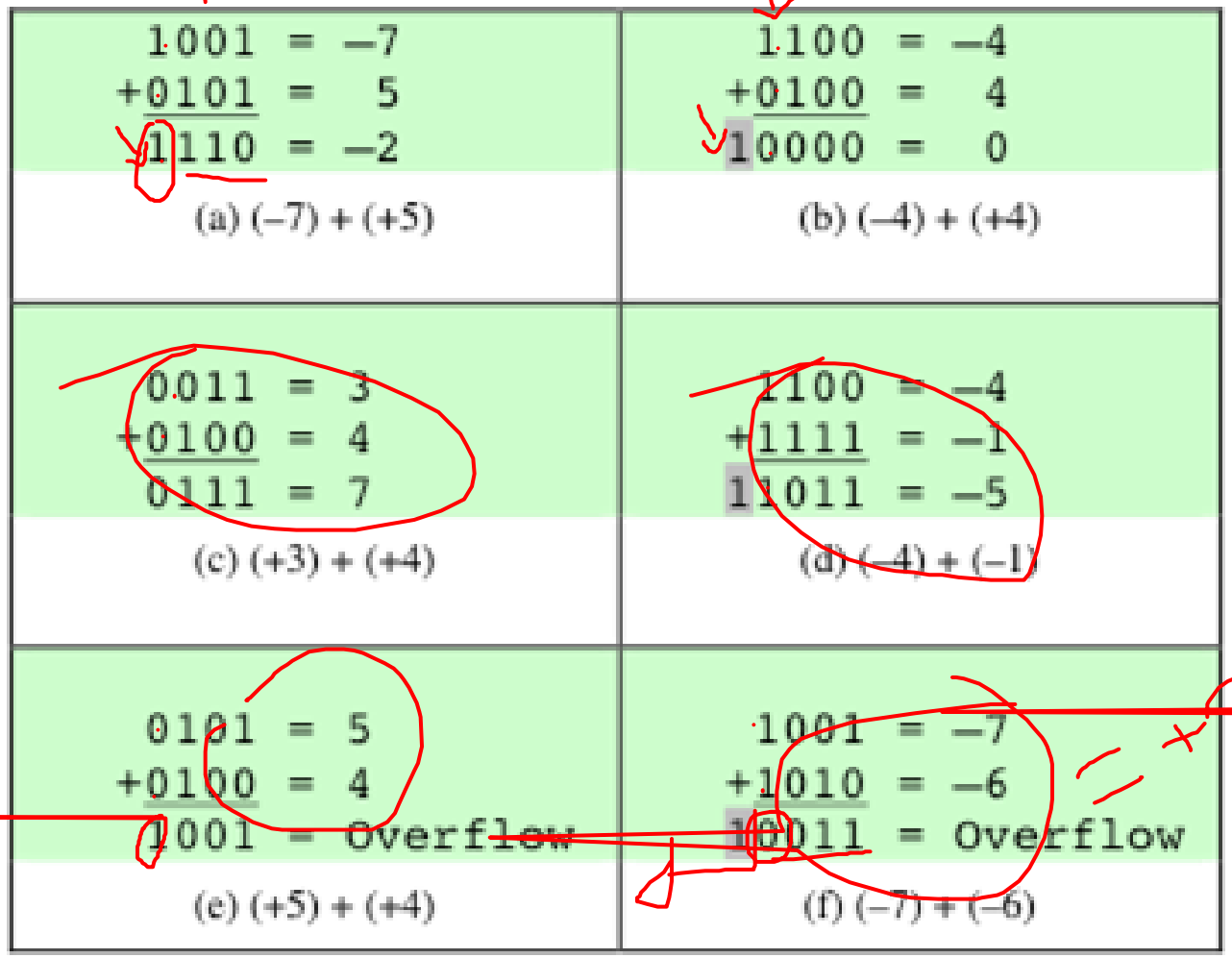
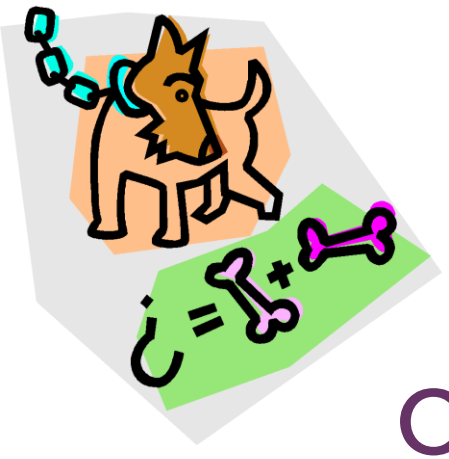


Figure 10.3 Addition of Numbers in Twos Complement Representation



Overflow

Rule

OVERFLOW RULE:

If two numbers are added, and they are both positive or both negative, then overflow occurs if and only if the result has the opposite sign.

Subtraction

Rule

SUBTRACTION RULE:

To subtract one number (subtrahend) from another (minuend), take the twos complement (negation) of the subtrahend and add it to the minuend.

Subtraction

0010 = 2 +1001 = −7 1011 = −5	0101 = 5 +1110 = −2 10011 = 3
(a) M = 2 = 0010 S = 7 = 0111 −S = 1001	(b) M = 5 = 0101 S = 2 = 0010 −S = 1110
1011 = −5 +1110 = −2 11001 = −7	0101 = 5 +0010 = 2 0111 = 7
(c) M = −5 = 1011 S = 2 = 0010 −S = 1110	(d) M = 5 = 0101 S = −2 = 1110 −S = 0010
0111 = 7 +0111 = 7 1110 = Overflow	1010 = −6 +1100 = −4 10110 = Overflow
(e) M = 7 = 0111 S = −7 = 1001 −S = 0111	(f) M = −6 = 1010 S = 4 = 0100 −S = 1100

Figure 10.4 Subtraction of Numbers in Twos Complement Representation (M – S)

Hardware for Addition and Subtraction

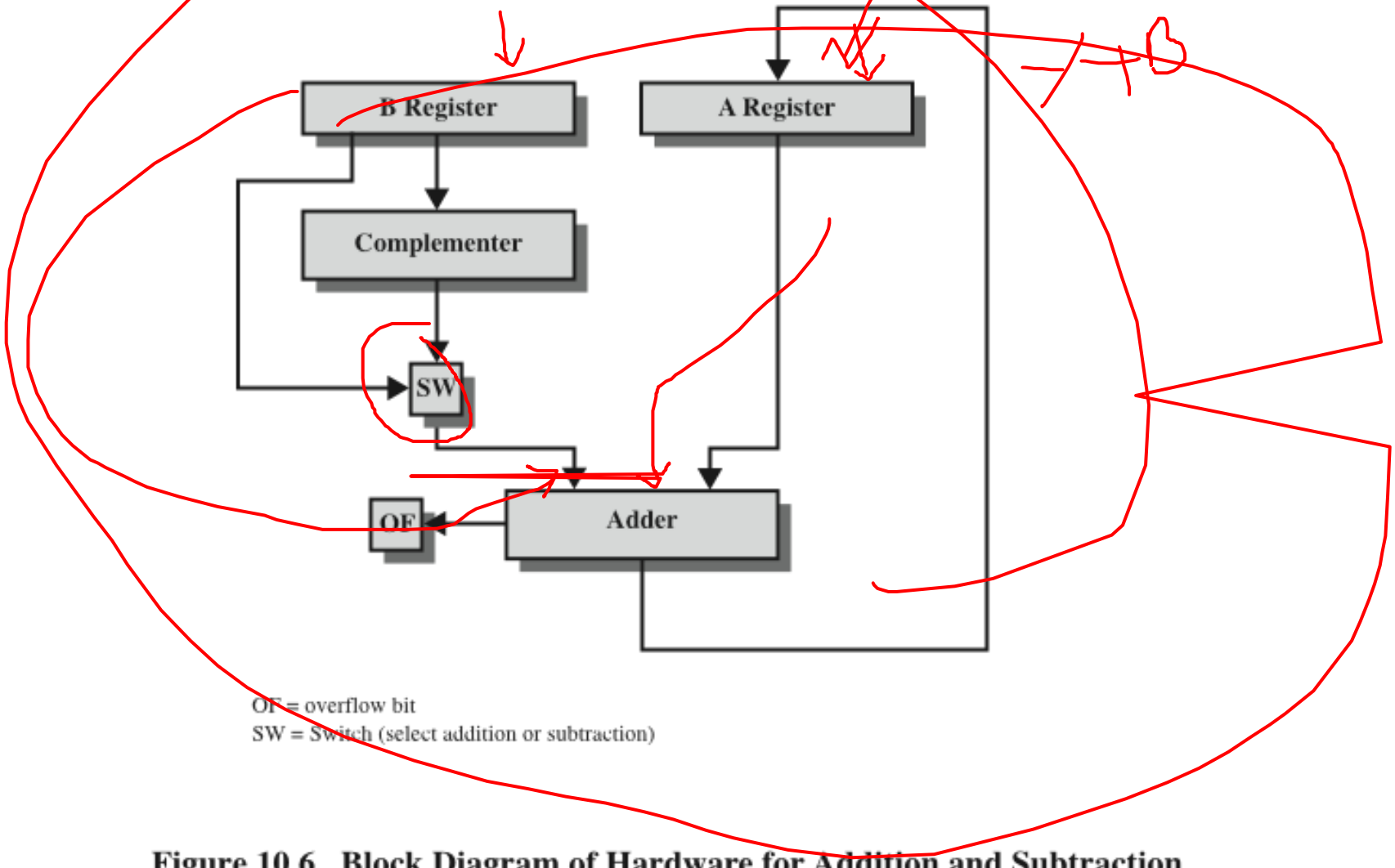
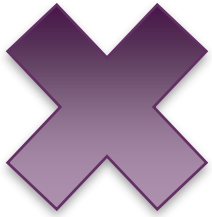


Figure 10.6 Block Diagram of Hardware for Addition and Subtraction



Multiplication

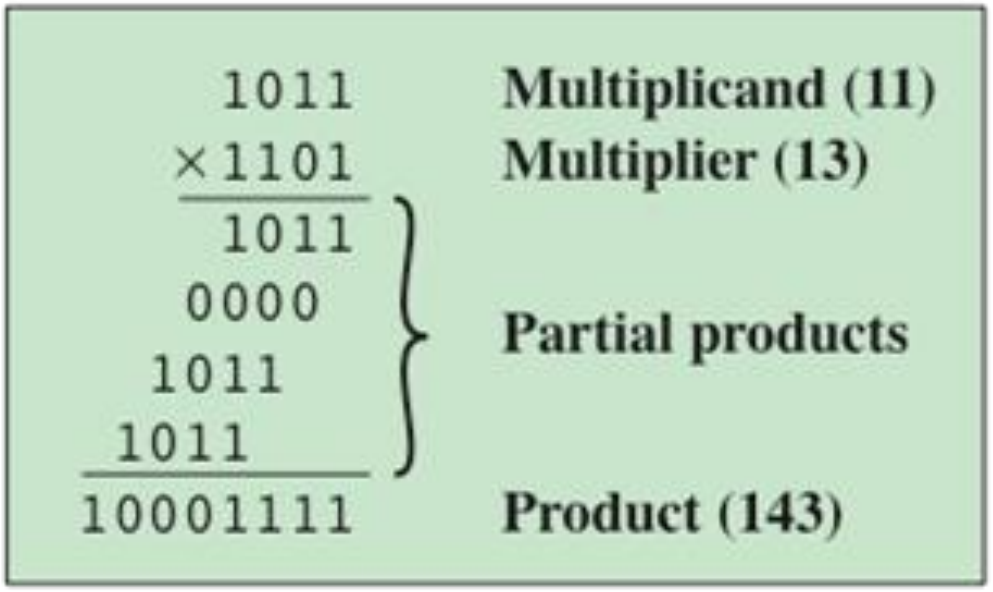
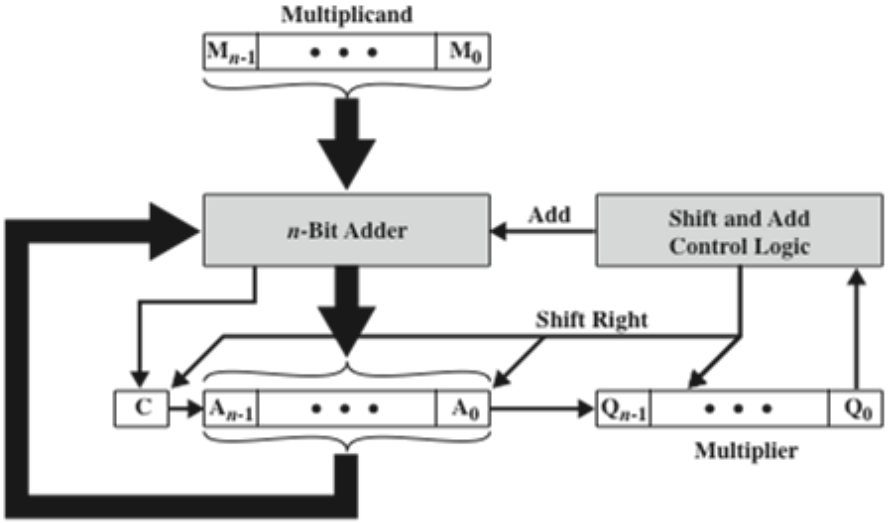


Figure 10.7 Multiplication of Unsigned Binary Integers

Hardware Implementation of Unsigned Binary Multiplication



(a) Block Diagram

C	A	Q	M		
0	0000	1101	1011	Initial Values	
0	1011	1101	1011	Add	First Cycle
0	0101	1110	1011	Shift	
0	0010	1111	1011	Shift	Second Cycle
0	1101	1111	1011	Add	Third Cycle
0	0110	1111	1011	Shift	
1	0001	1111	1011	Add	Fourth Cycle
0	1000	1111	1011	Shift	

(b) Example from Figure 9.7 (product in A, Q)

Figure 10.8 Hardware Implementation of Unsigned Binary Multiplication

Flowchart for Unsigned Binary Multiplication

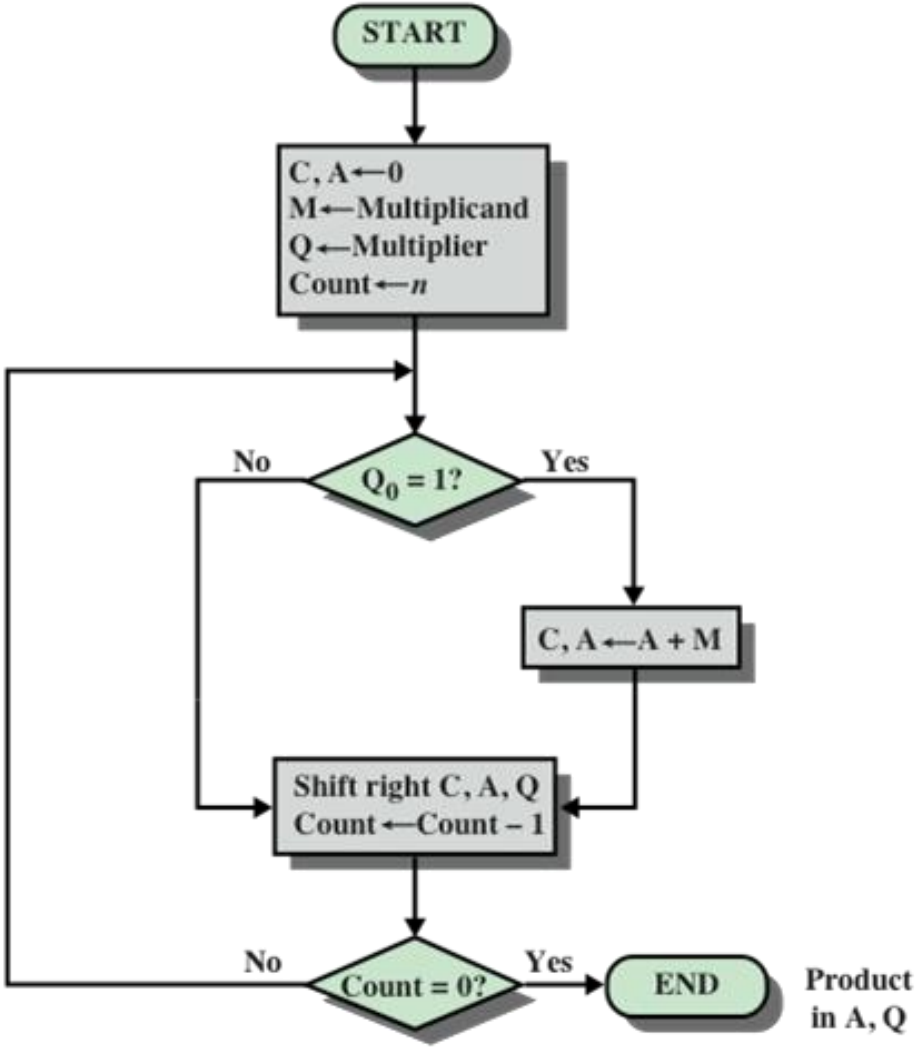


Figure 10.9 Flowchart for Unsigned Binary Multiplication

Twos Complement Multiplication

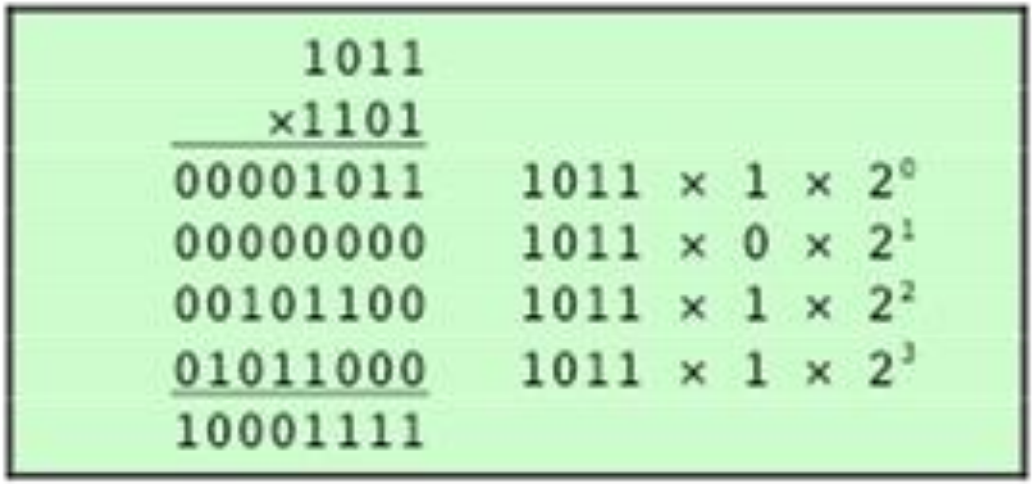
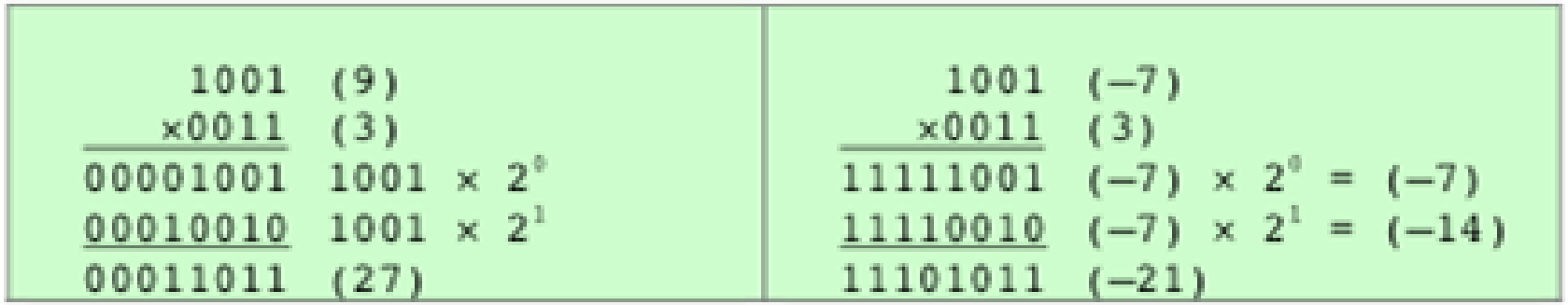


Figure 10.10 Multiplication of Two Unsigned 4-Bit Integers Yielding an 8-Bit Result

Comparison



```
    1001  (9)
   x0011  (3)
--------  ----------
00001001  1001 x 2^0
00010010  1001 x 2^1
--------  ----------
00011011  (27)
```

(a) Unsigned integers

```
    1001  (−7)
   x0011  (3)
--------  ----------------------
11111001  (−7) x 2^0 = (−7)
11110010  (−7) x 2^1 = (−14)
--------  ----------------------
11101011  (−21)
```

(b) Twos complement integers

Figure 10.11 Comparison of Multiplication of Unsigned and Twos Complement Integers

Booth's Algorithm

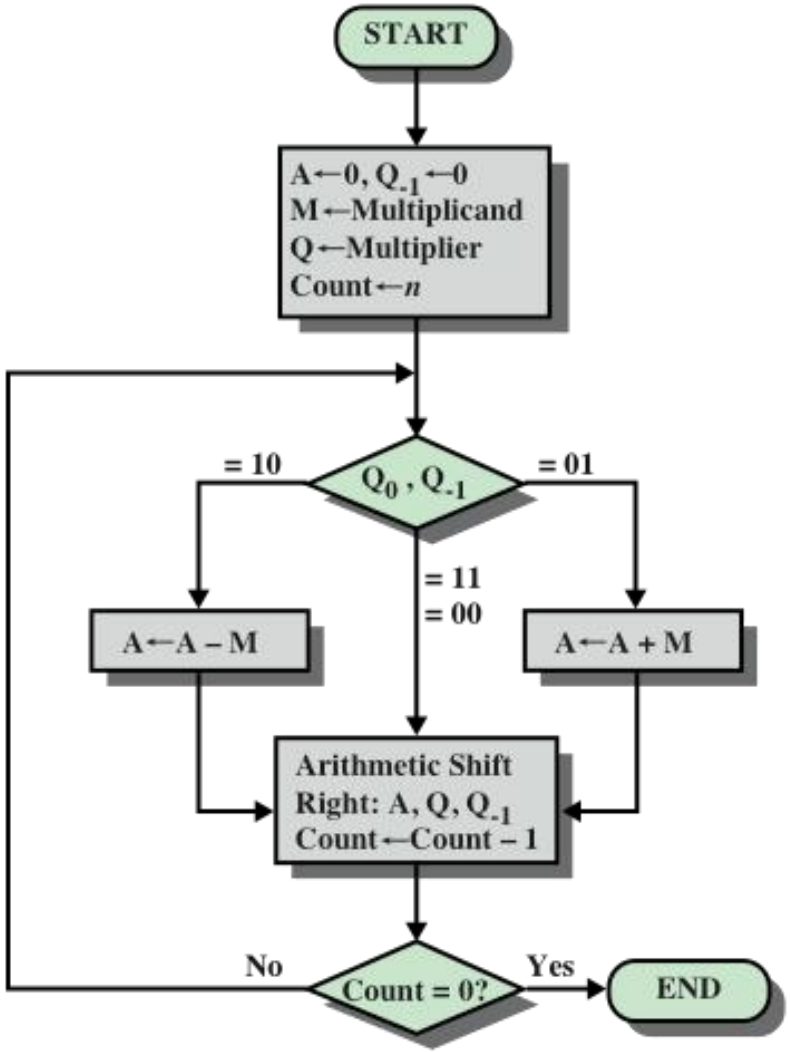


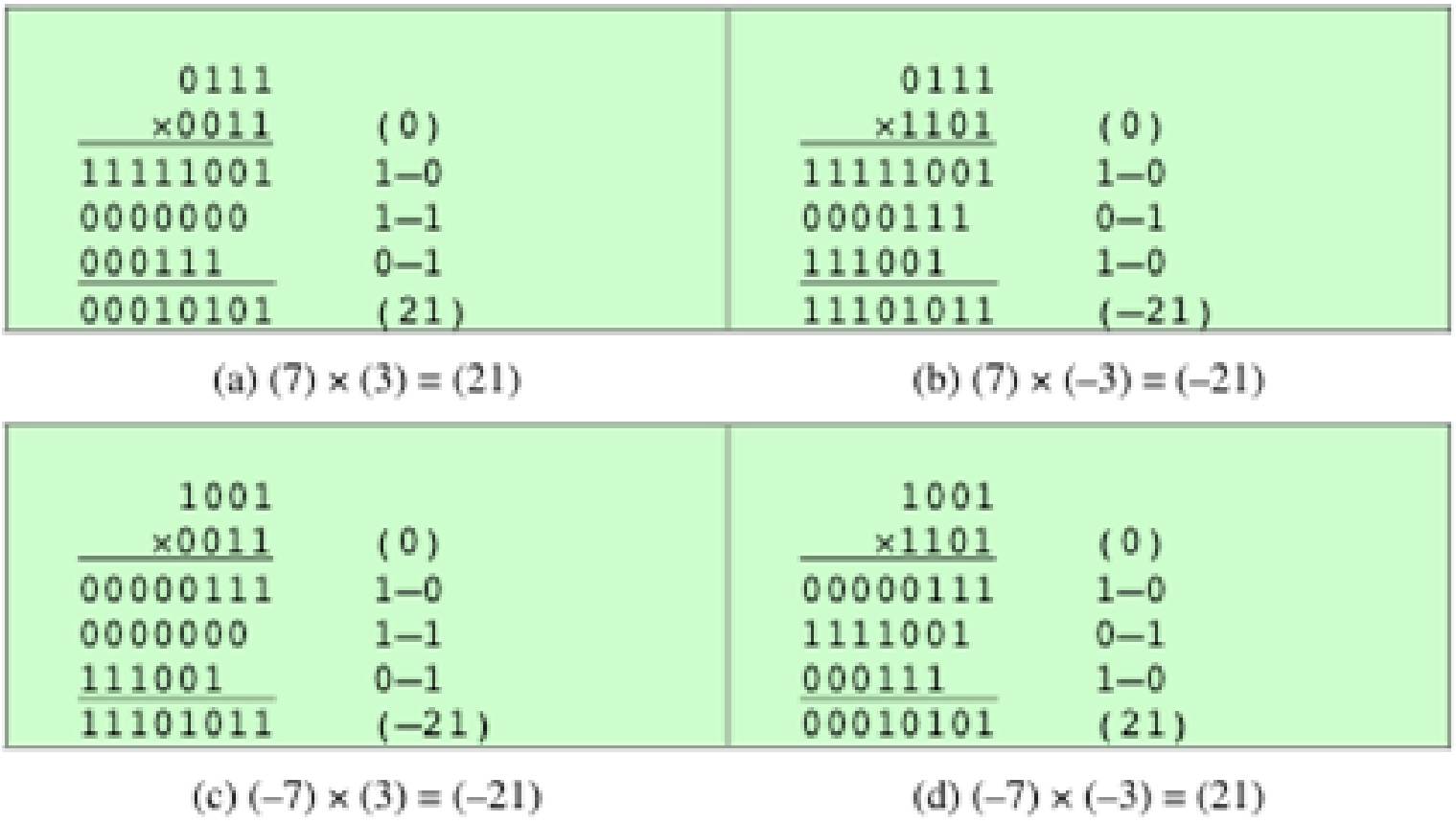
Figure 10.12 Booth's Algorithm for Twos Complement Multiplication

Example of Booth's Algorithm

A	Q	Q_{-1}	M		
0000	0011	0	0111	Initial Values	
1001	0011	0	0111	A ← A - M	First Cycle
1100	1001	1	0111	Shift	
1110	0100	1	0111	Shift	Second Cycle
0101	0100	1	0111	A ← A + M	Third Cycle
0010	1010	0	0111	Shift	
0001	0101	0	0111	Shift	Fourth Cycle

Figure 10.13 Example of Booth's Algorithm (7× 3)

Examples Using Booth's Algorithm



```
    0111
   ×0011     (0)
11111001     1–0
0000000      1–1
000111       0–1
00010101     (21)
```

(a) (7) × (3) = (21)

```
    0111
   ×1101     (0)
11111001     1–0
0000111      0–1
111001       1–0
11101011     (–21)
```

(b) (7) × (–3) = (–21)

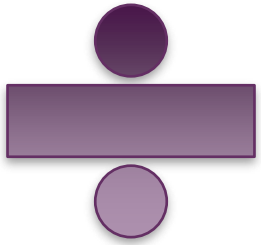
```
    1001
   ×0011     (0)
00000111     1–0
0000000      1–1
111001       0–1
11101011     (–21)
```

(c) (–7) × (3) = (–21)

```
    1001
   ×1101     (0)
00000111     1–0
1111001      0–1
000111       1–0
00010101     (21)
```

(d) (–7) × (–3) = (21)

Figure 10.14 Examples Using Booth's Algorithm



Division

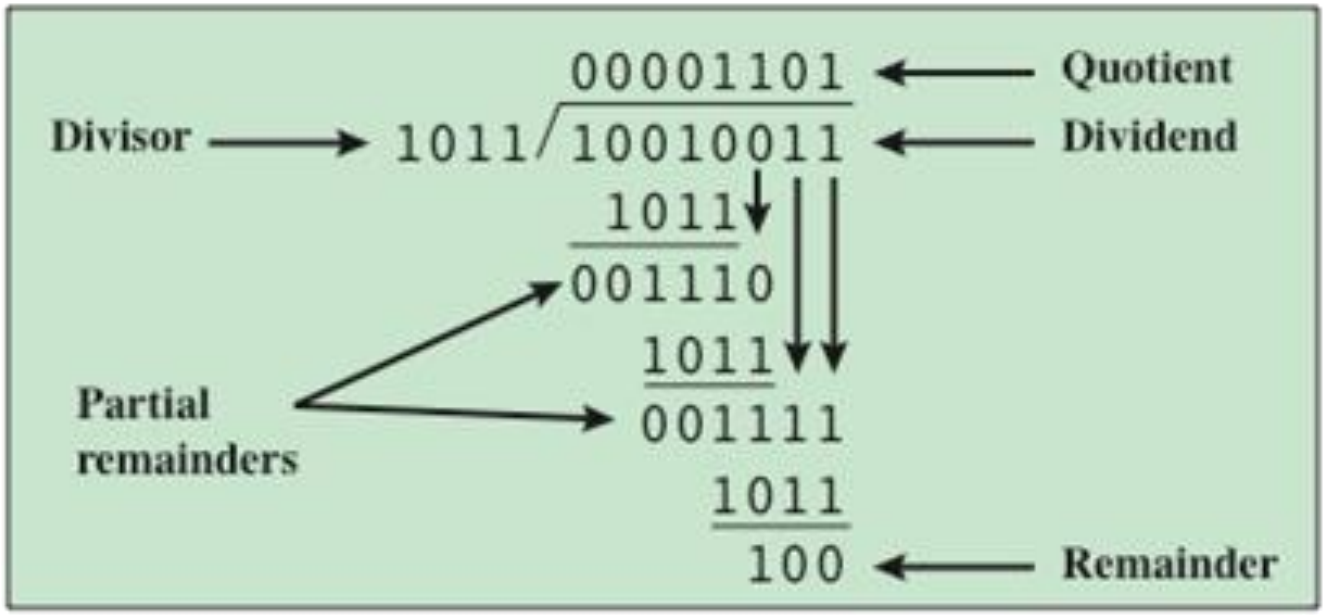


Figure 10.15 Example of Division of Unsigned Binary Integers

Flowchart for Unsigned Binary Division

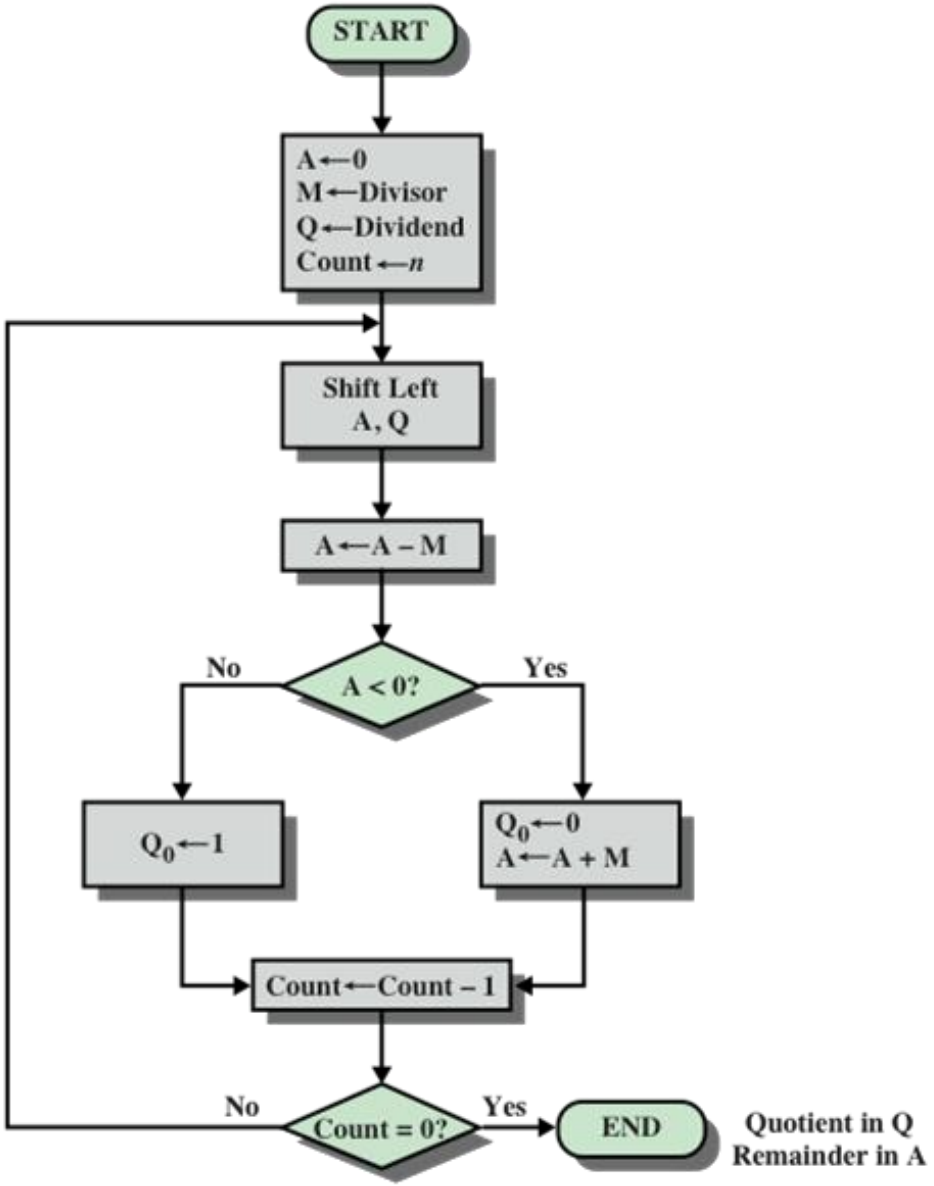


Figure 10.16 Flowchart for Unsigned Binary Division

Flowchart for Unsigned Binary Division

The algorithm can be summarized as follows:

1. Load the twos complement of the divisor into the M register; that is, the M register contains the negative of the divisor. Load the dividend into the A, Q registers. The dividend must be expressed as a 2*n-bit positive number*. *Thus*, for example, the 4-bit 0111 becomes 00000111.

2. Shift A, Q left 1 bit position.

Flowchart for Unsigned Binary Division

3. Perform A =A - M. This operation subtracts the divisor from the contents of A.

4. a. If the result is nonnegative (most significant bit of A = 0), then set Q0 =1.
b. If the result is negative (most significant bit of A = 1), then set Q0 =0 and restore the previous value of A.

5. Repeat steps 2 through 4 as many times as there are bit positions in Q.

6. The remainder is in A and the quotient is in Q.

Example of Restoring Twos Complement Division

A	Q	
0000	0111	Initial value
0000	1110	Shift
1101		Use twos complement of 0011 for subtraction
1101		Subtract
0000	1110	Restore, set $Q_0 = 0$
0001	1100	Shift
1101		
1110		Subtract
0001	1100	Restore, set $Q_0 = 0$
0011	1000	Shift
1101		
0000	1001	Subtract, set $Q_0 = 1$
0001	0010	Shift
1101		
1110		Subtract
0001	0010	Restore, set $Q_0 = 0$

Figure 10.17 Example of Restoring Twos Complement Division (7/3)