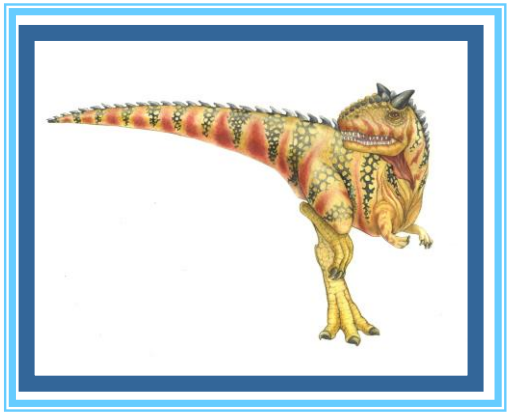
Memory Management
Day5: Sep 2021

Kiran Waghmare



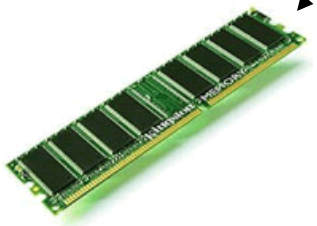
Everything has an address

0x60:0x6f

0x3c0:0x3cf

Processor
i386

0x1f0:0x1f7

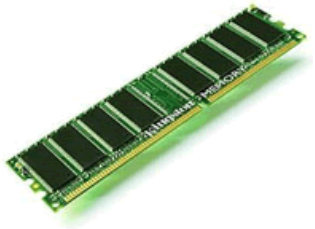


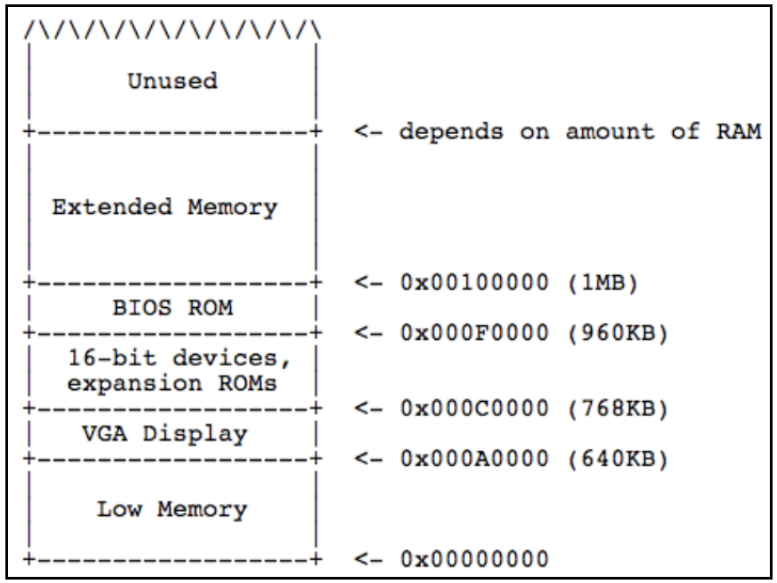
0x0 : 0x200000

0x60:0x6f

Address Types : (Memory Addresses)

- Range : 0 to (RAM size or 2^{32}-1)
- Where main memory is mapped
  - Used to store data for code, heap, stack, OS, etc.
- Accessed by load/store instructions

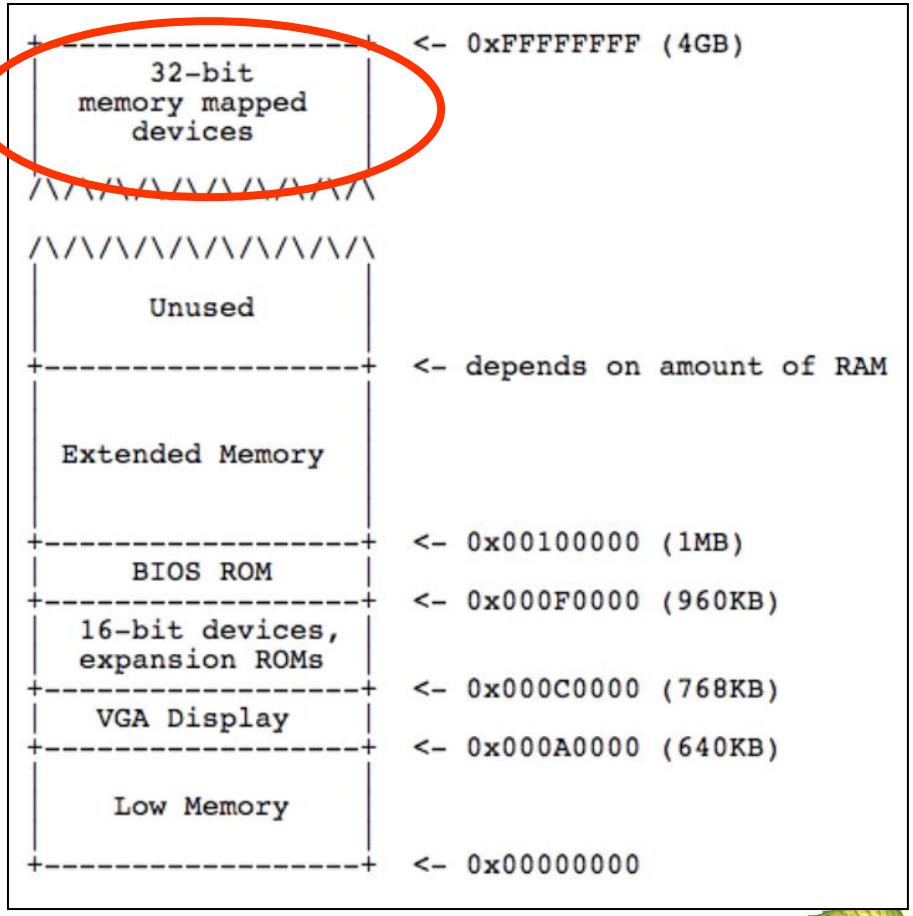




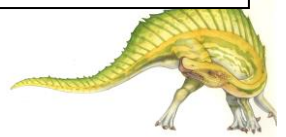
RAM

Memory Mapped I/O

- Why?
  - More space
- Devices and RAM share the same address space
- Instructions used to access RAM can also be used to access devices.
  - Eg load/store



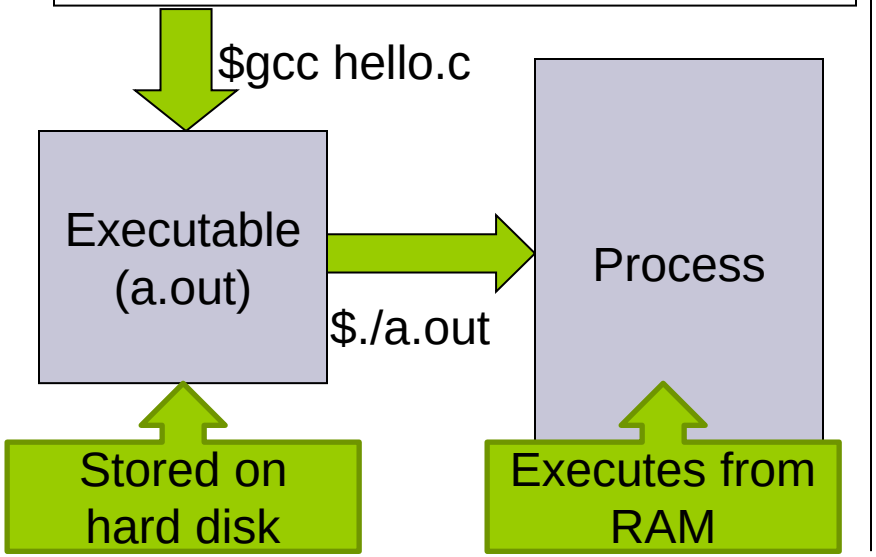
Memory Map



Executing Programs (Process)

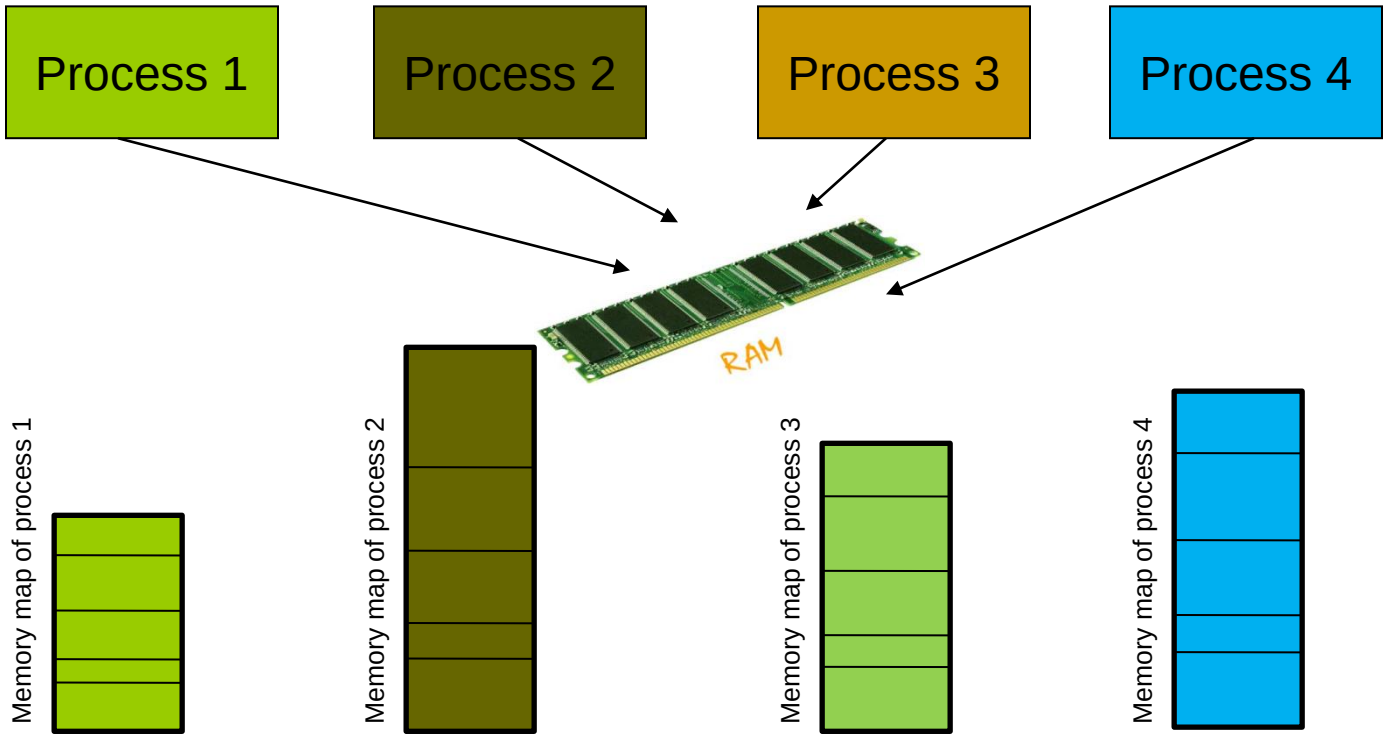
```
#include <stdio.h>

int main(){
   char str[] = "Hello World\n";
   printf("%s", str);
}
```



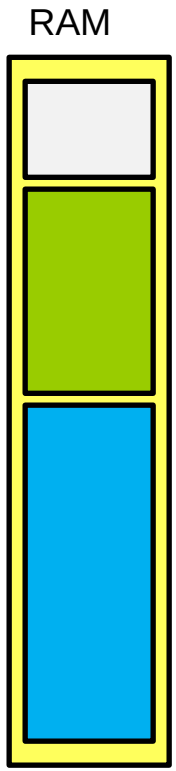
- Process
  - A program in execution
  - Present in the RAM
  - Comprises of
    - Executable instructions
    - Stack
    - Heap
    - State in the OS (in kernel)
  - State contains : registers, list of open files, related processes, etc.

Sharing RAM


Process 1
Process 2
Process 3
Process 4
RAM
Memory map of process 1
Memory map of process 2
Memory map of process 3
Memory map of process 4

Partition Model

- As long sufficient contiguous space is available, new processes are allocated memory.



Partition Table

Memory Address	Size	Process	Usage
0x0	120k	4	In use
120k	60k	1	In use
180k	30k		Free

Partition Model (allocating/deallocating processes)

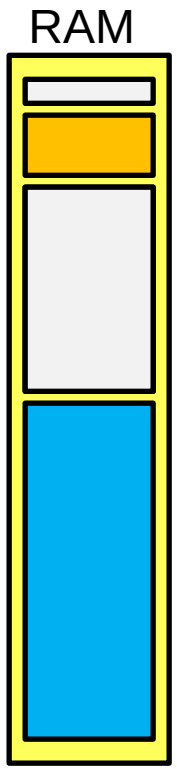
- As long sufficient contiguous space is available, new processes allocated memory.

Process 5 of size 20k started — Process 5 allocated RAM

Process 1 completed — Process 1 deallocated from RAM

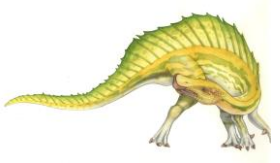
Partition Table

Memory Address	Size	Process	Usage
0x0	120k	4	In use
120k	60k		Free
180k	20k	5	In Use
200k	10k		Free



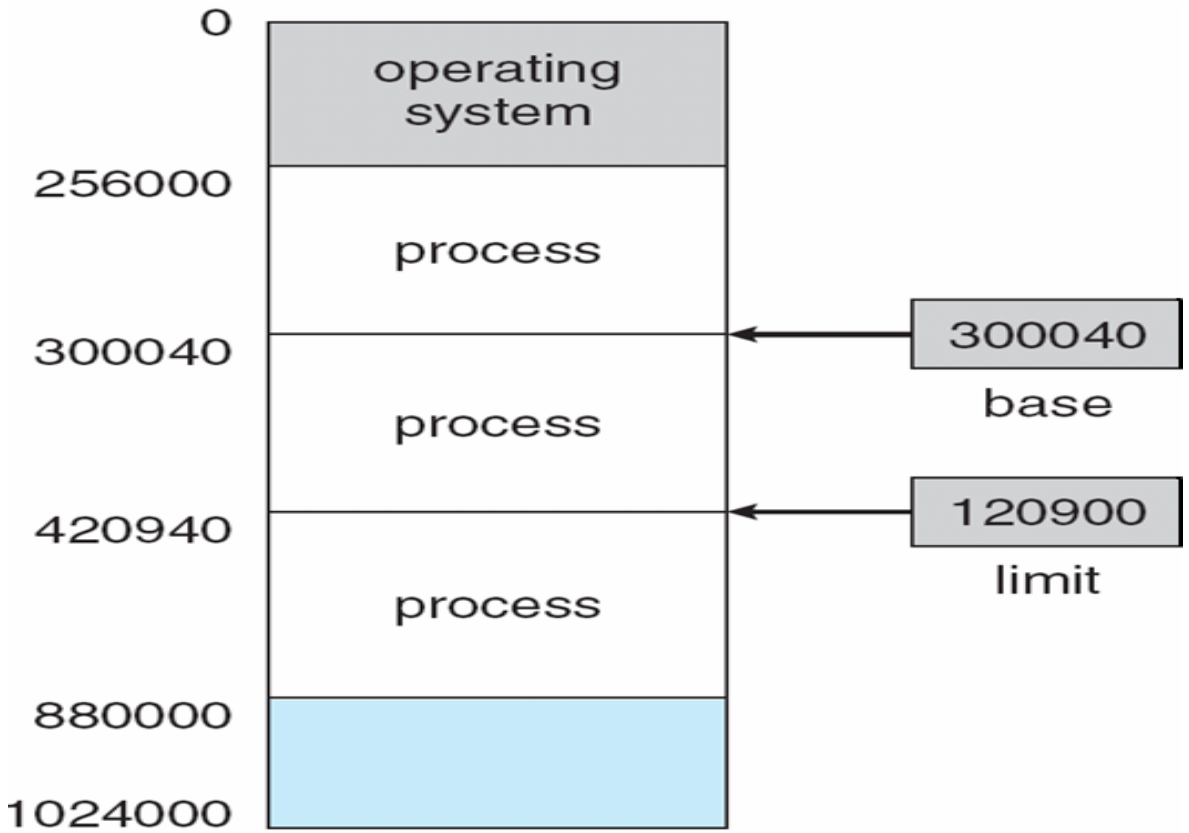
Background

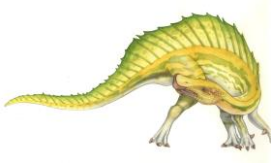
- Program must **be brought (from disk) into memory** and placed within a process for it to be run
- **Main memory and registers are only storage CPU** can access directly
- Register access in **one CPU clock** (or less)
- Main memory **can take many cycles**
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation



Base and Limit Registers

- A pair of **base** and **limit** registers define the logical address space



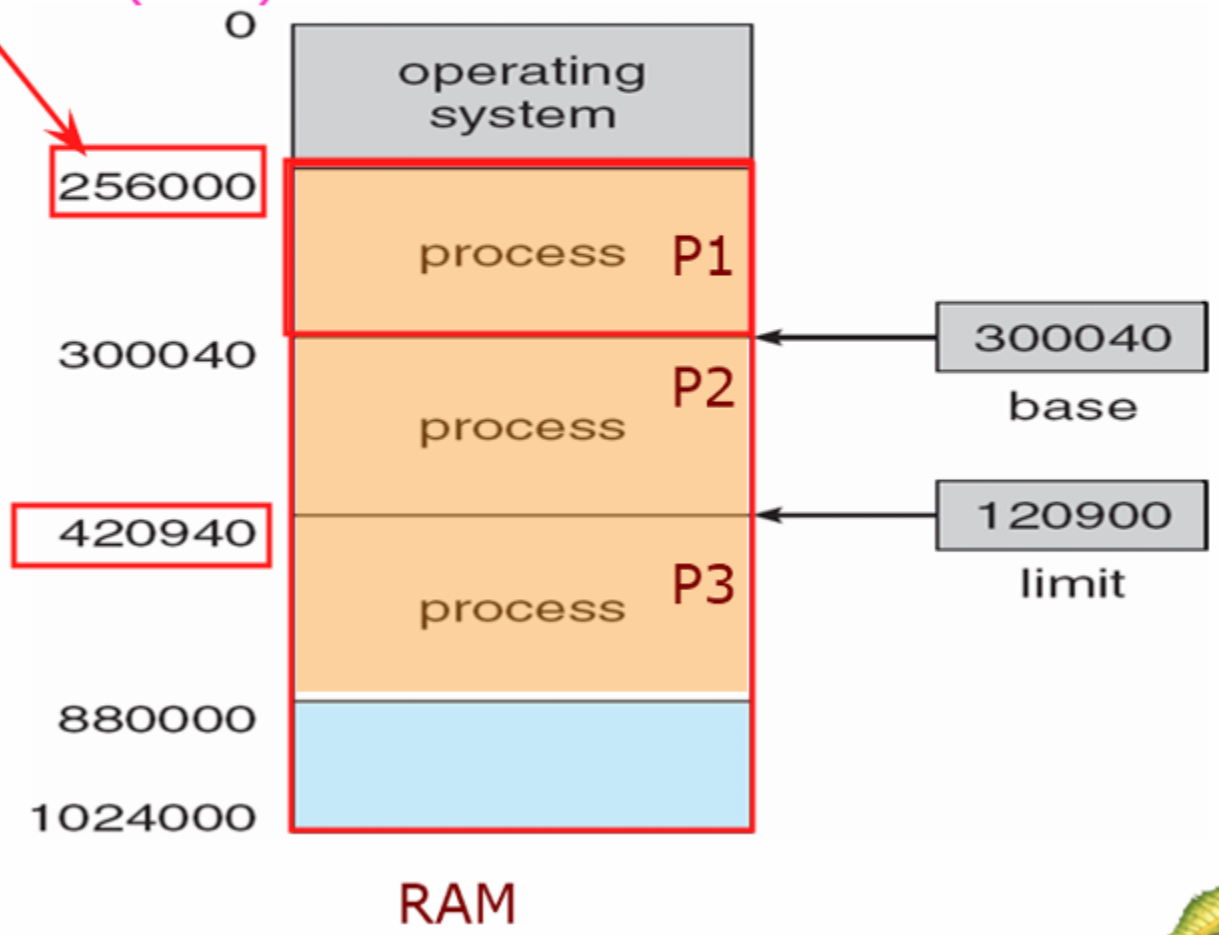


Base and Limit Registers

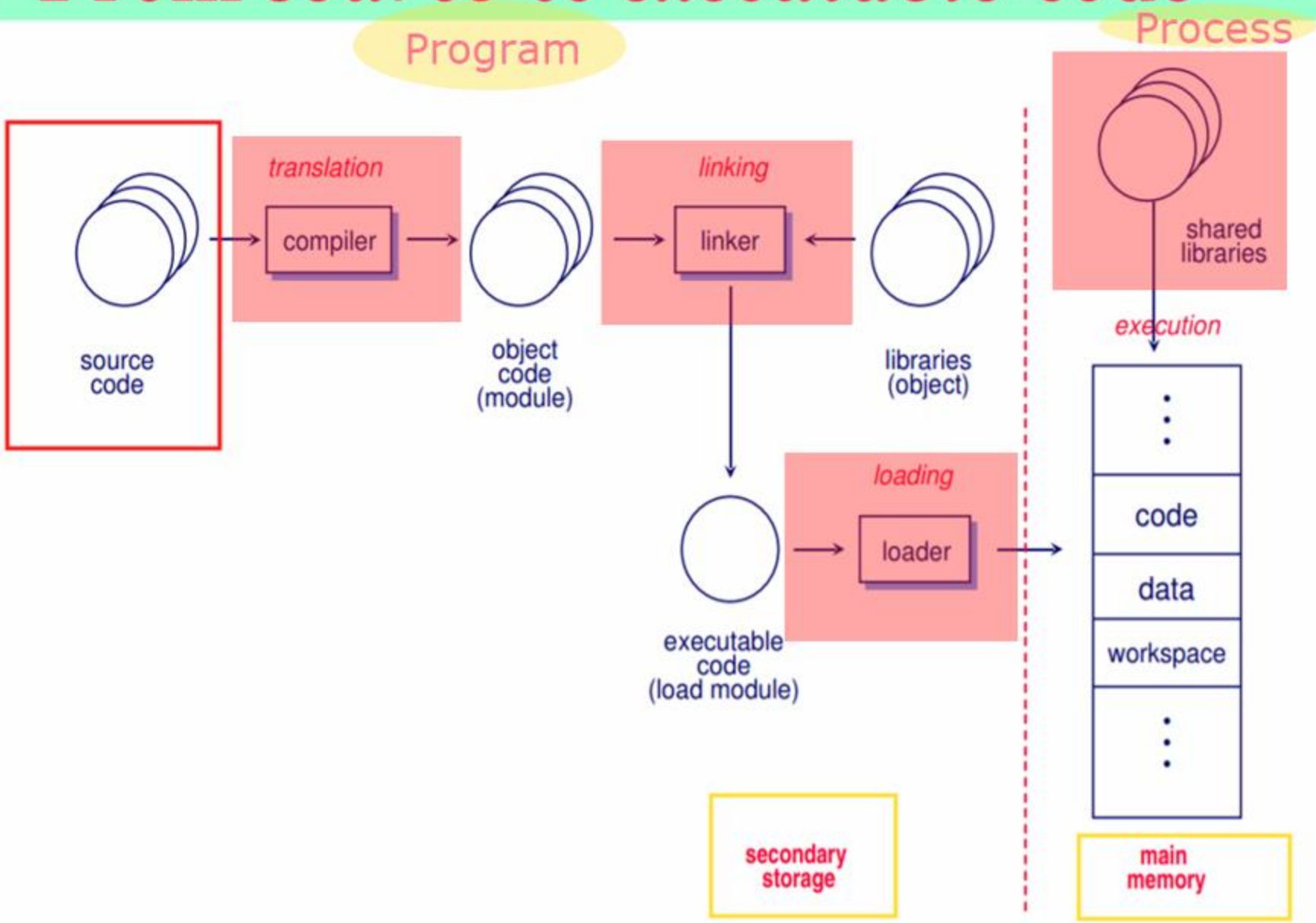
200+ 330=530

- A pair of **base** and **limit** registers define the logical address space

(size)

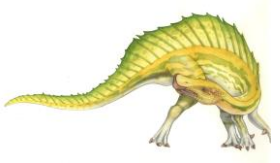


From *source* to *executable* code

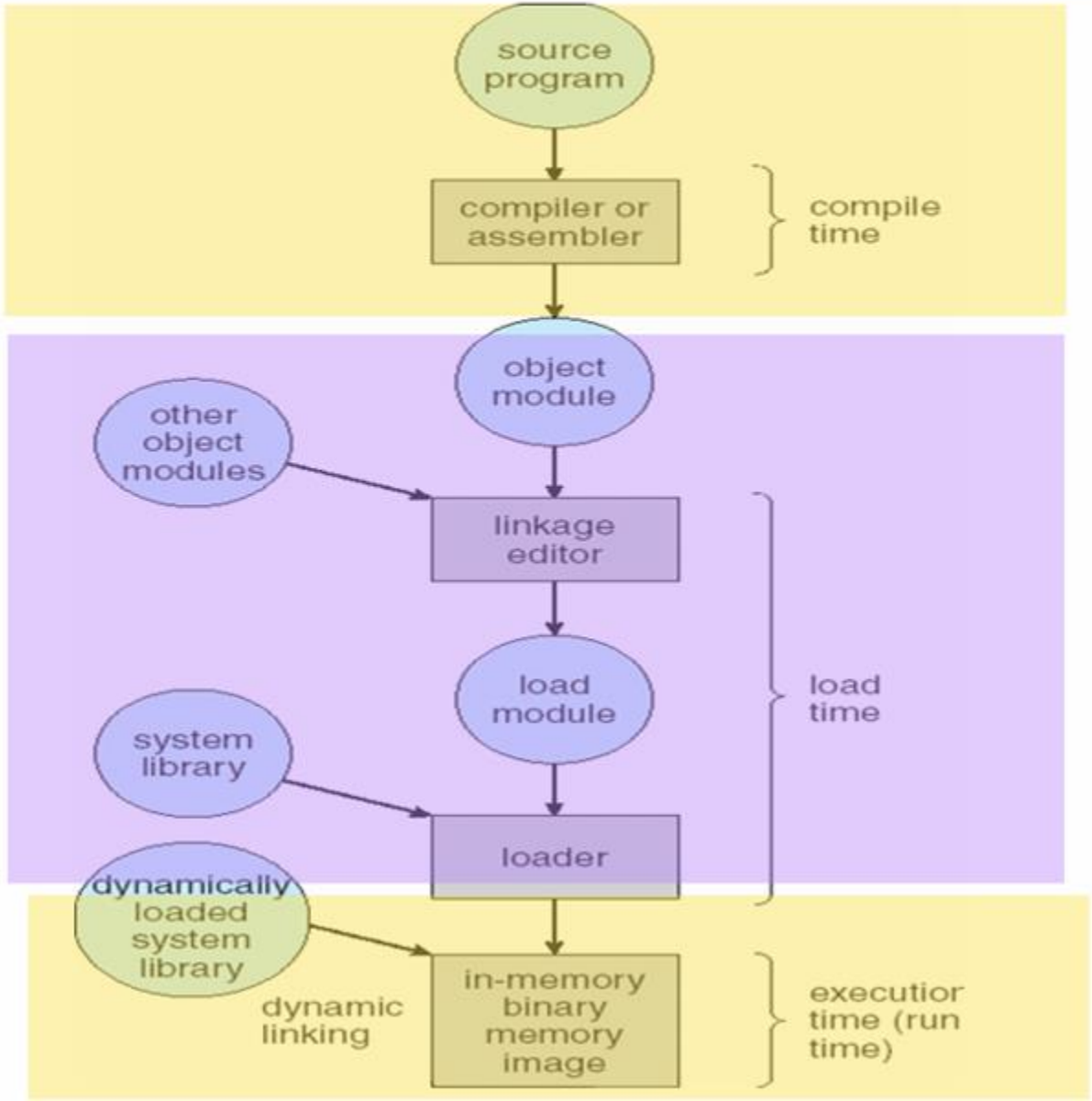


Binding of Instructions and Data to Memory

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



Multistep Processing of a User Program

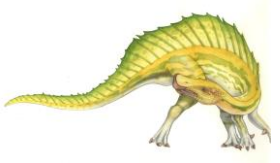


Logical vs. Physical Address Space

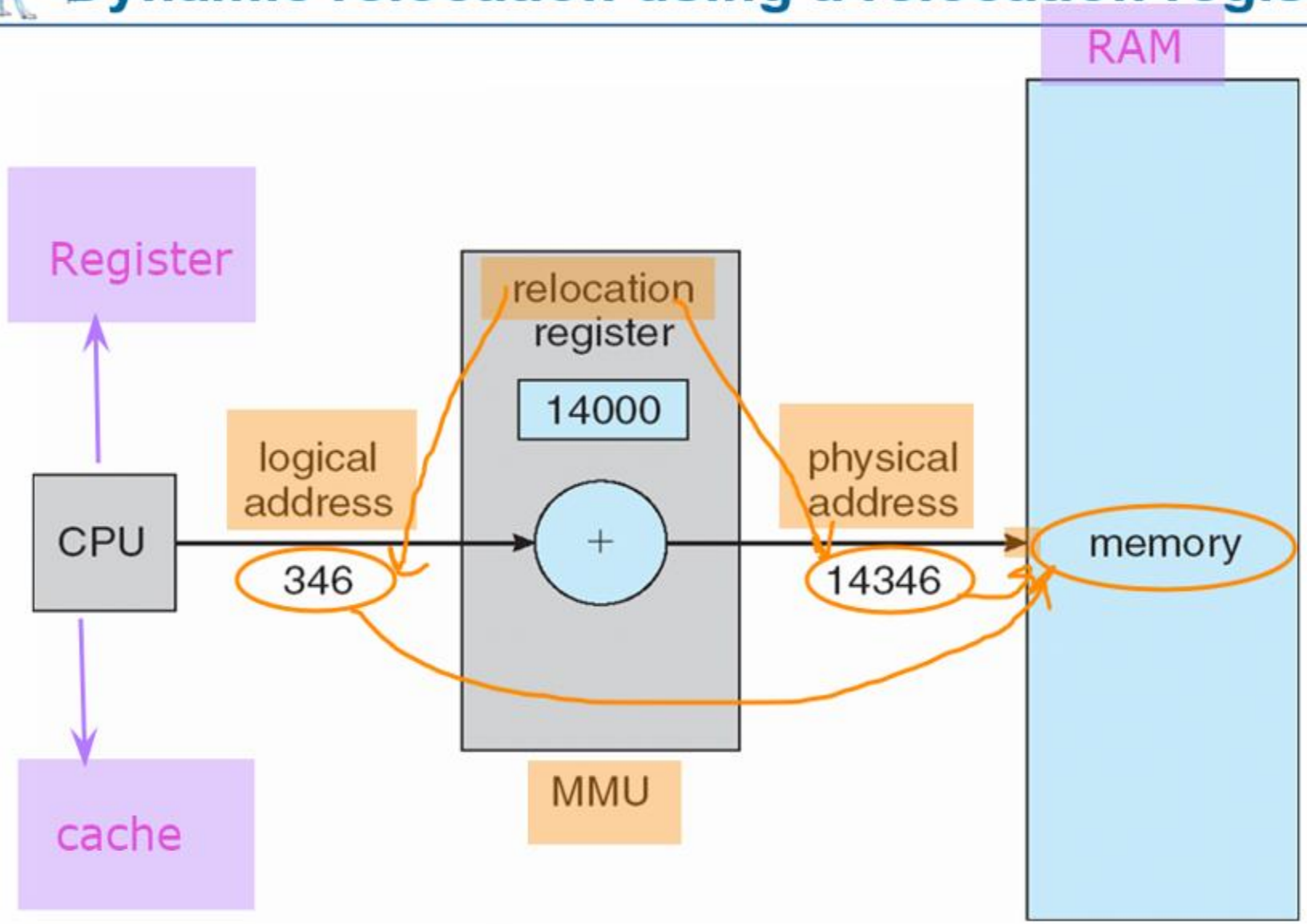
- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
  - **Logical address** – generated by the CPU; also referred to as **virtual address**
  - **Physical address** – address seen by the memory unit
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme

Memory-Management Unit (MMU)

- Hardware device that **maps virtual to physical address**
- In MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
- The user program deals with ***logical* addresses;** it never sees the *real* physical addresses



Dynamic relocation using a relocation register

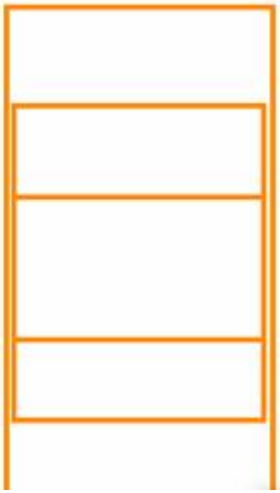


Dynamic Loading

- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded
- Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required implemented through program design

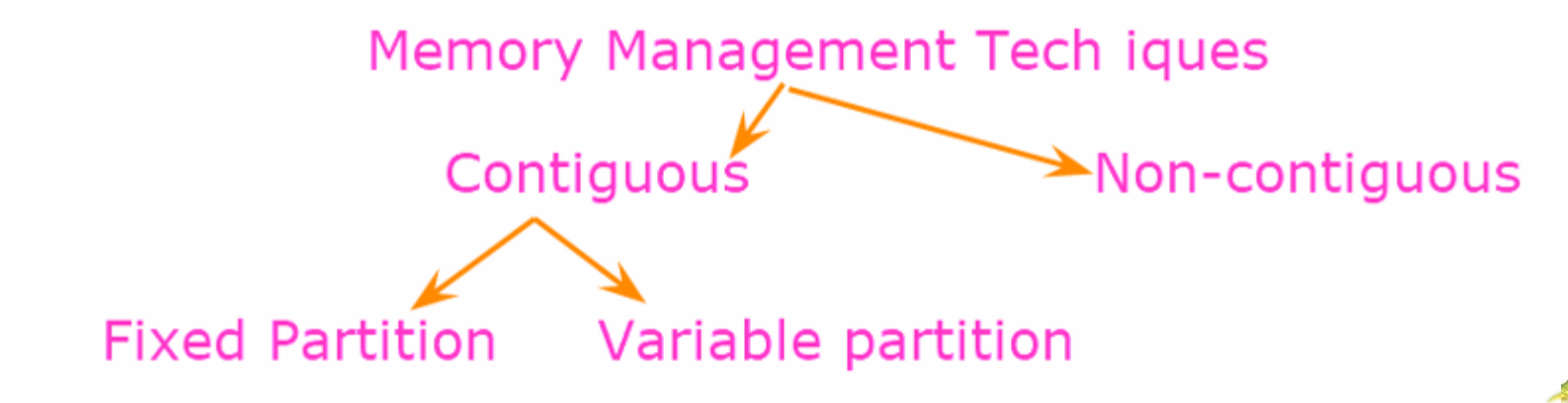
Static Loading →

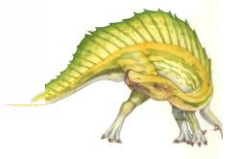
Dynamic Loading →



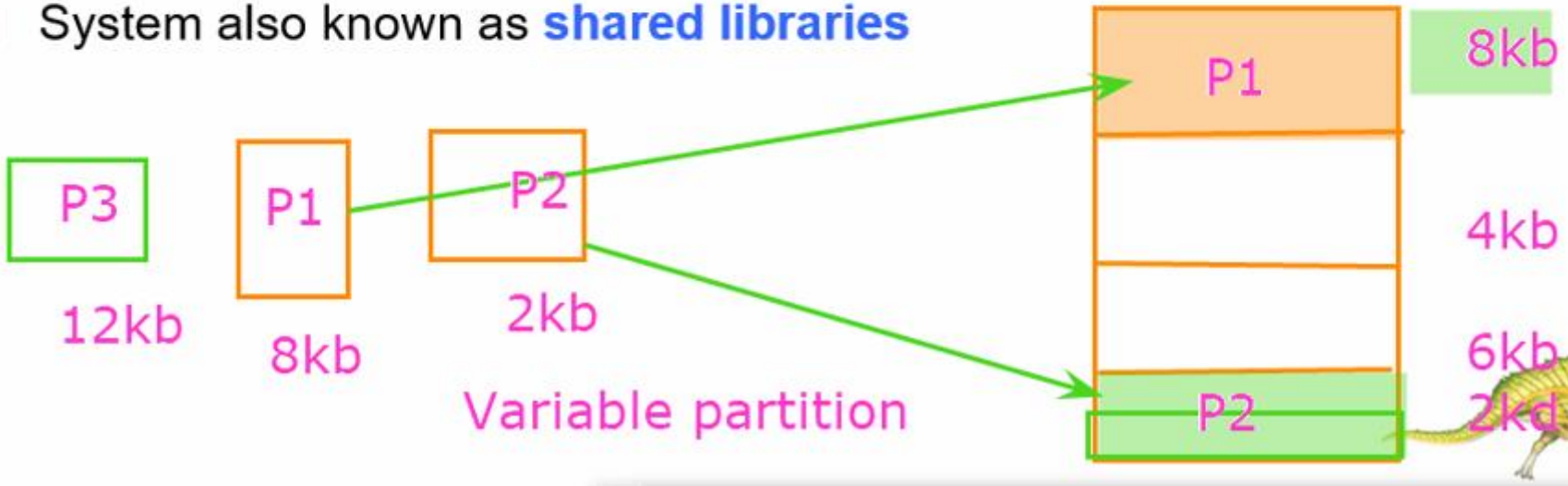
Dynamic Linking

- Linking postponed until execution time
- Small piece of code, *stub*, used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes the routine
- Operating system needed to check if routine is in processes' memory address
- Dynamic linking is particularly useful for libraries
- System also known as **shared libraries**





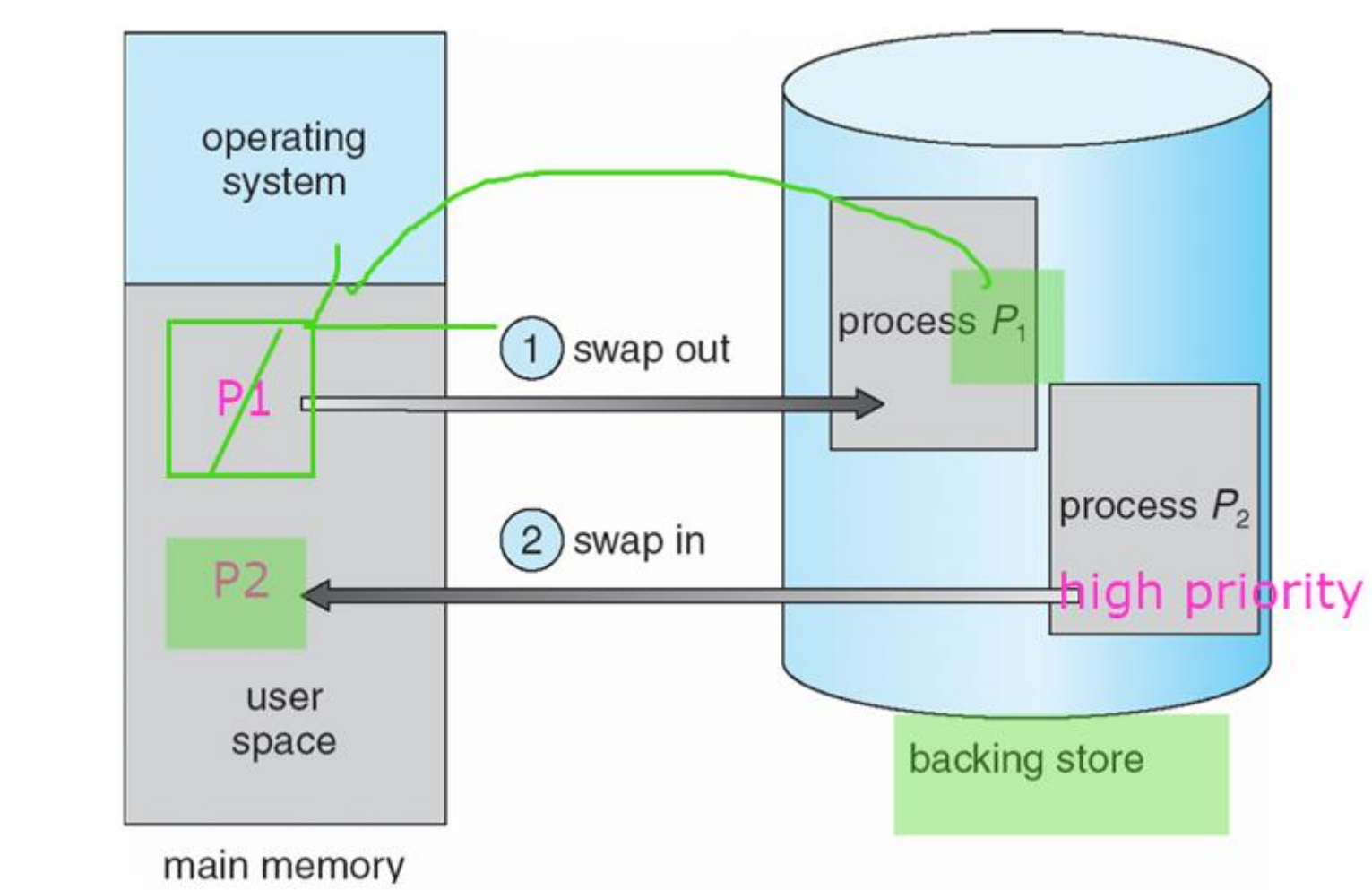
- Linking postponed until execution time
- Small piece of code, *stub*, used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes the routine
- Operating system needed to check if routine is in processes' memory address
- Dynamic linking is particularly useful for libraries
- System also known as **shared libraries**



Swapping

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

Schematic View of Swapping

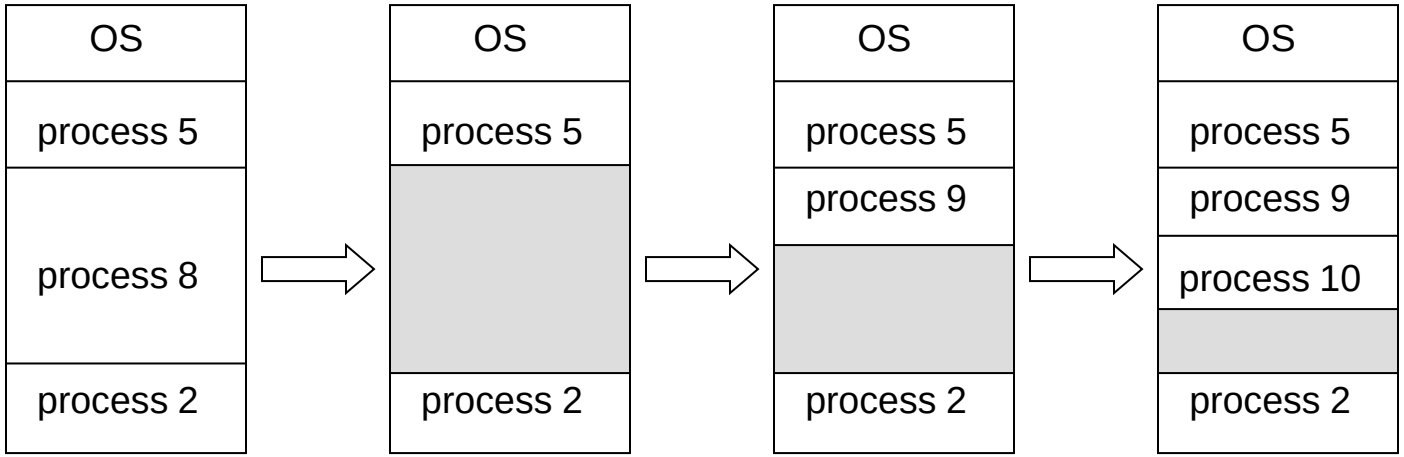


Contiguous Allocation

- Main memory usually into two partitions:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory
- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - Base register contains value of smallest physical address
  - Limit register contains range of logical addresses – each logical address must be less than the limit register
  - MMU maps logical address *dynamically*

Contiguous Allocation (Cont)

- Multiple-partition allocation
  - Hole – block of available memory; holes of various size are scattered throughout memory
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it
  - Operating system maintains information about:
    a) allocated partitions b) free partitions (hole)

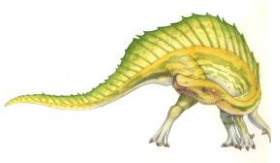


Dynamic Storage-Allocation Problem

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

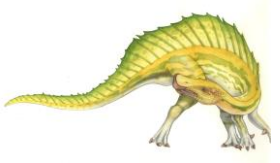
- **First-fit**: Allocate the *first* hole that is big enough
- **Best-fit**: Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size
  - Produces the smallest leftover hole
- **Worst-fit**: Allocate the *largest* hole; must also search entire list
  - Produces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

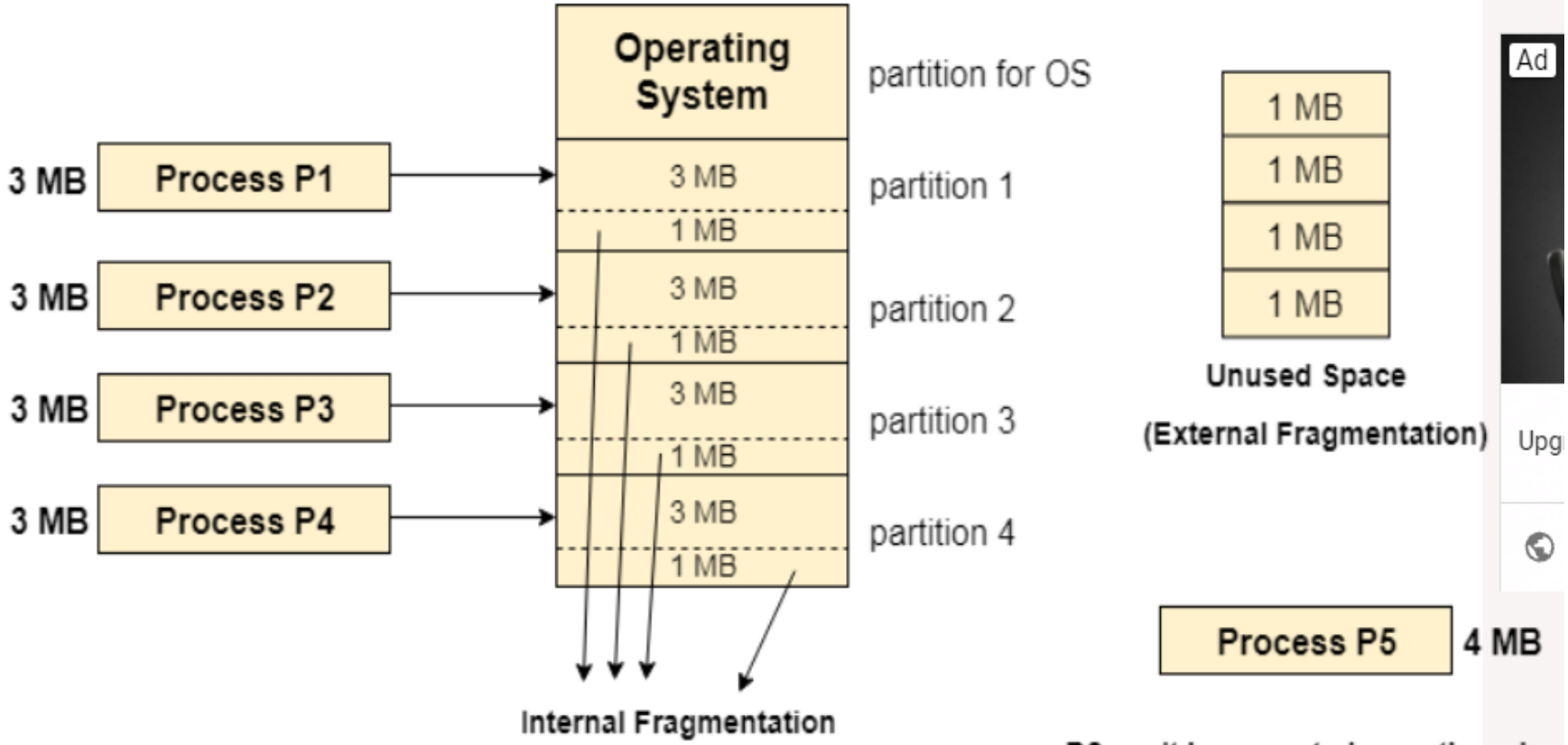
First-fit and best-fit better than worst-fit in terms of speed and storage utilization



Contiguous Memory Allocation :

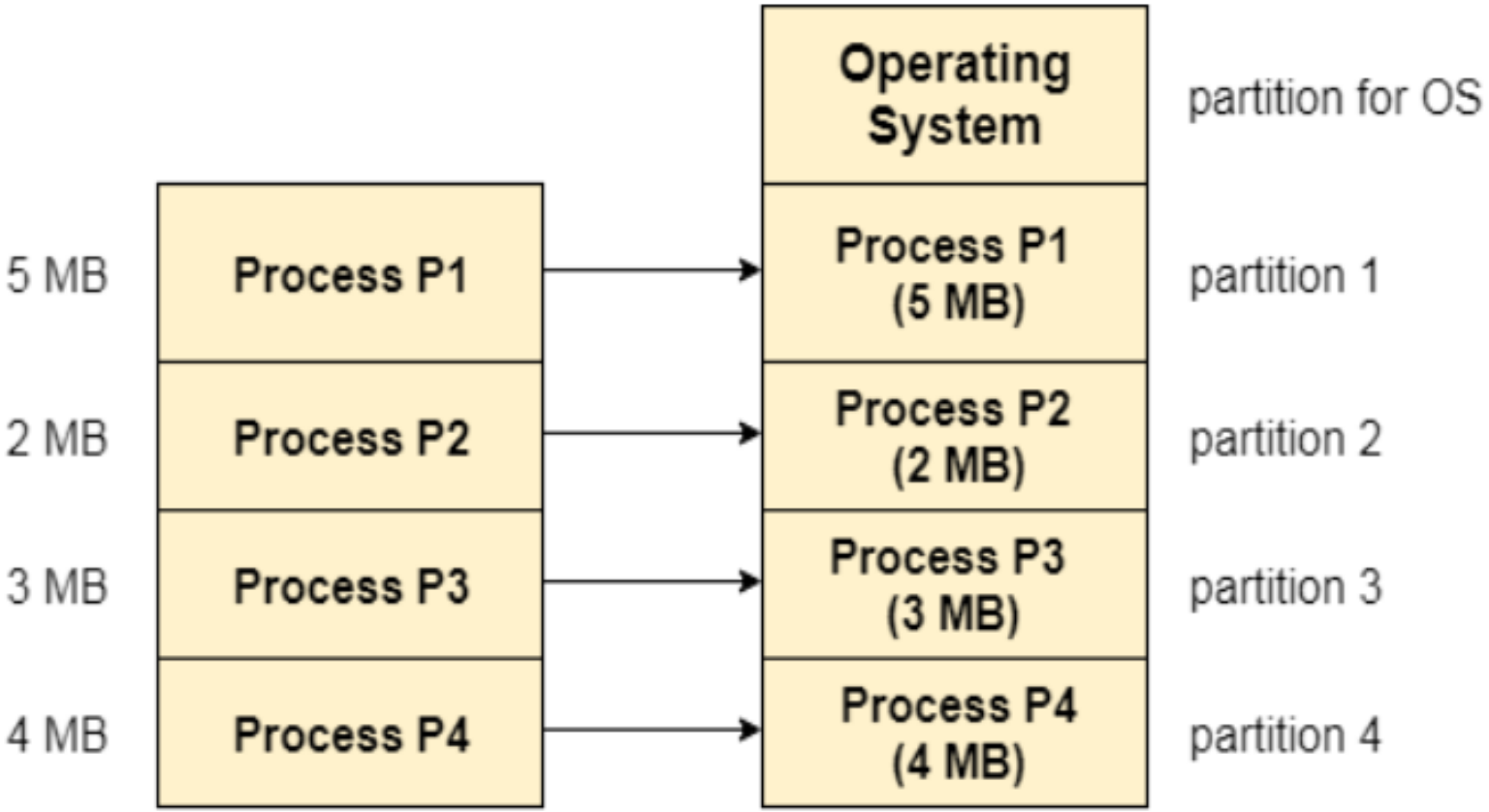
- The main memory should **oblige** both the operating system and the different client processes.
- Therefore, the **allocation of memory** becomes an important task in the operating system.
- The memory is **usually divided into two partitions**: one for the resident operating system and one for the user processes.
- We **normally need several user processes** to reside in memory simultaneously.
- Therefore, we need to consider how to allocate available memory to the processes





Fixed Partitioning

(Contiguous memory allocation)

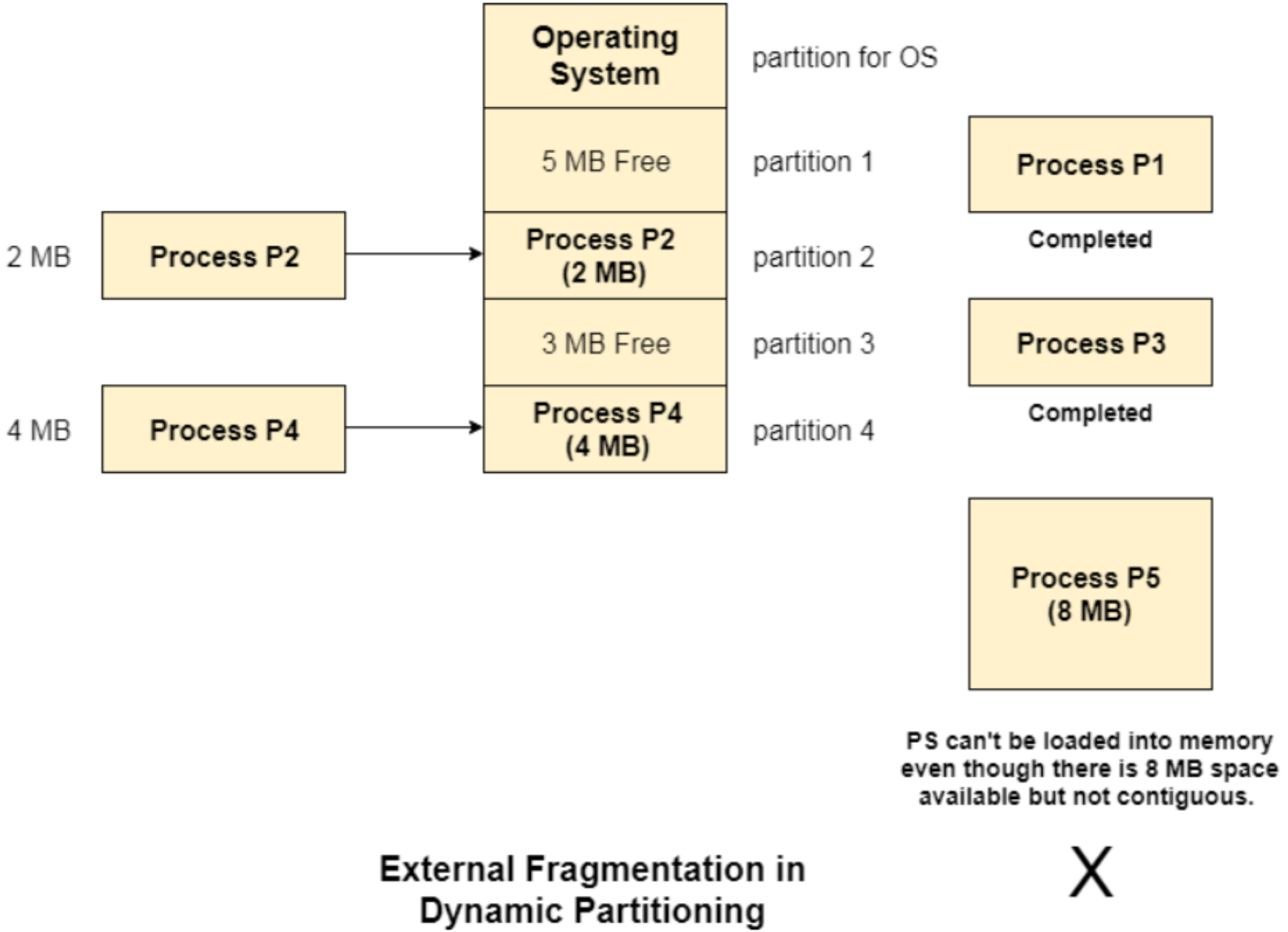


Dynamic Partitioning

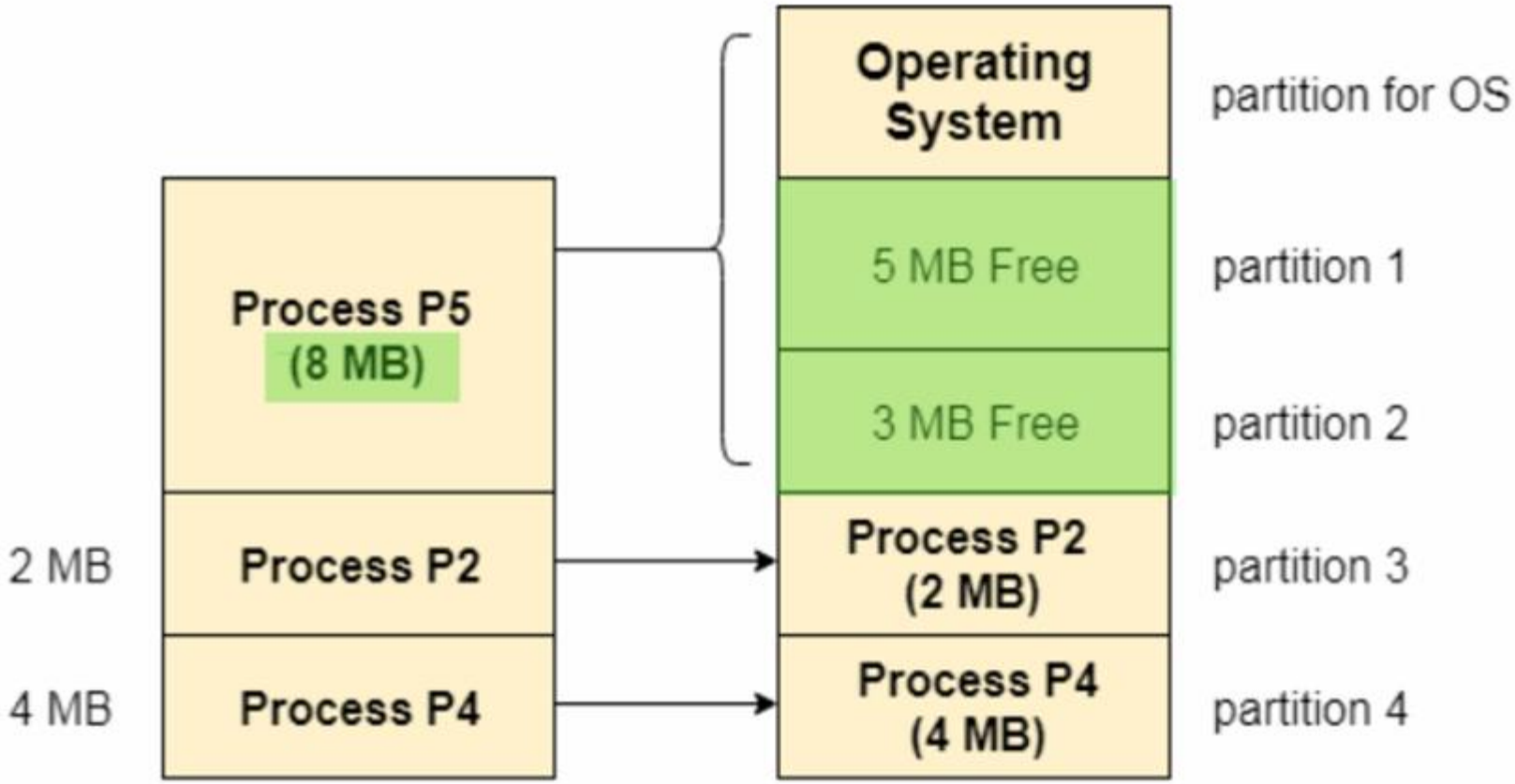
(Process Size = Partition Size)

Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time
  - I/O problem
    - Latch job in memory while it is involved in I/O
    - Do I/O only into OS buffers

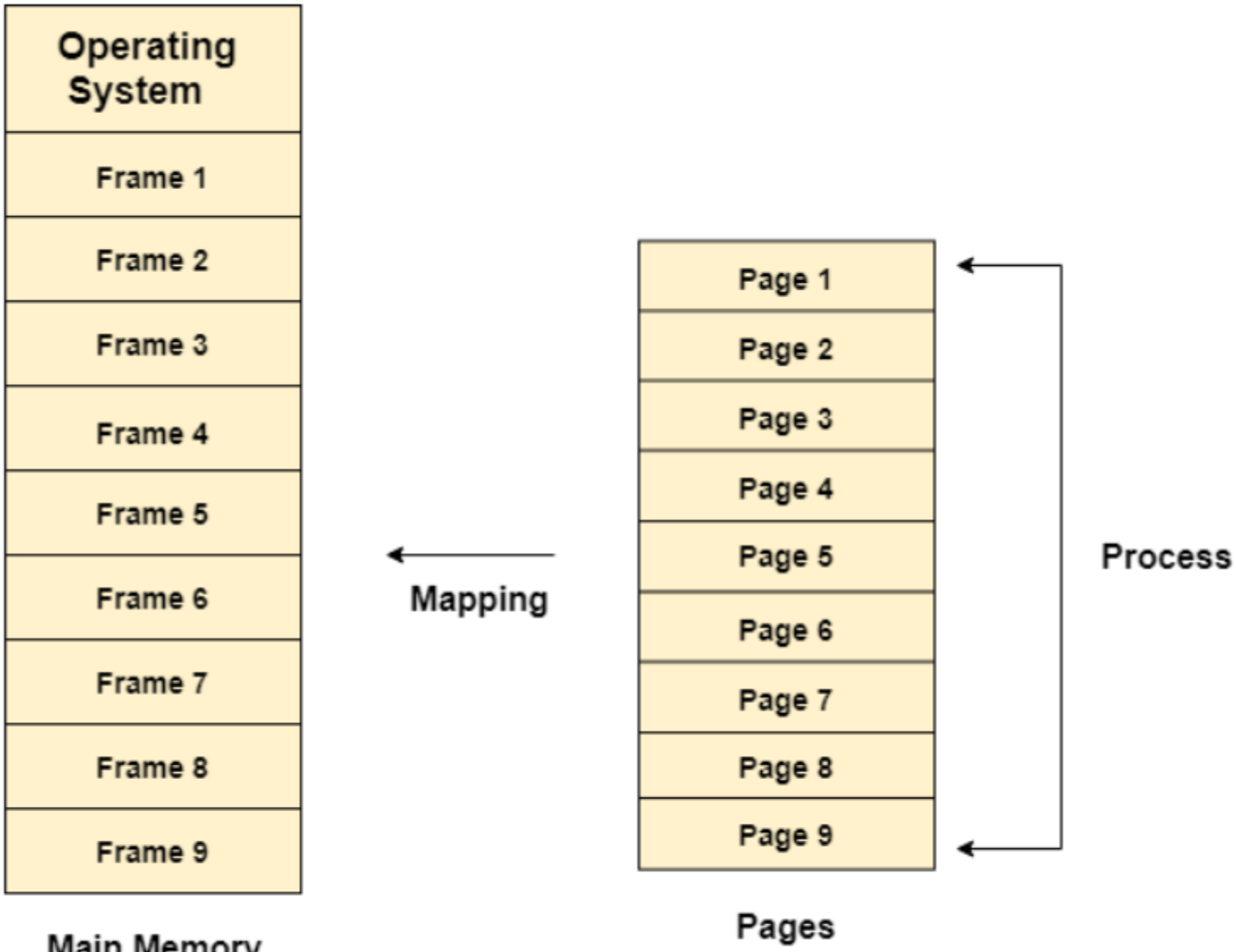


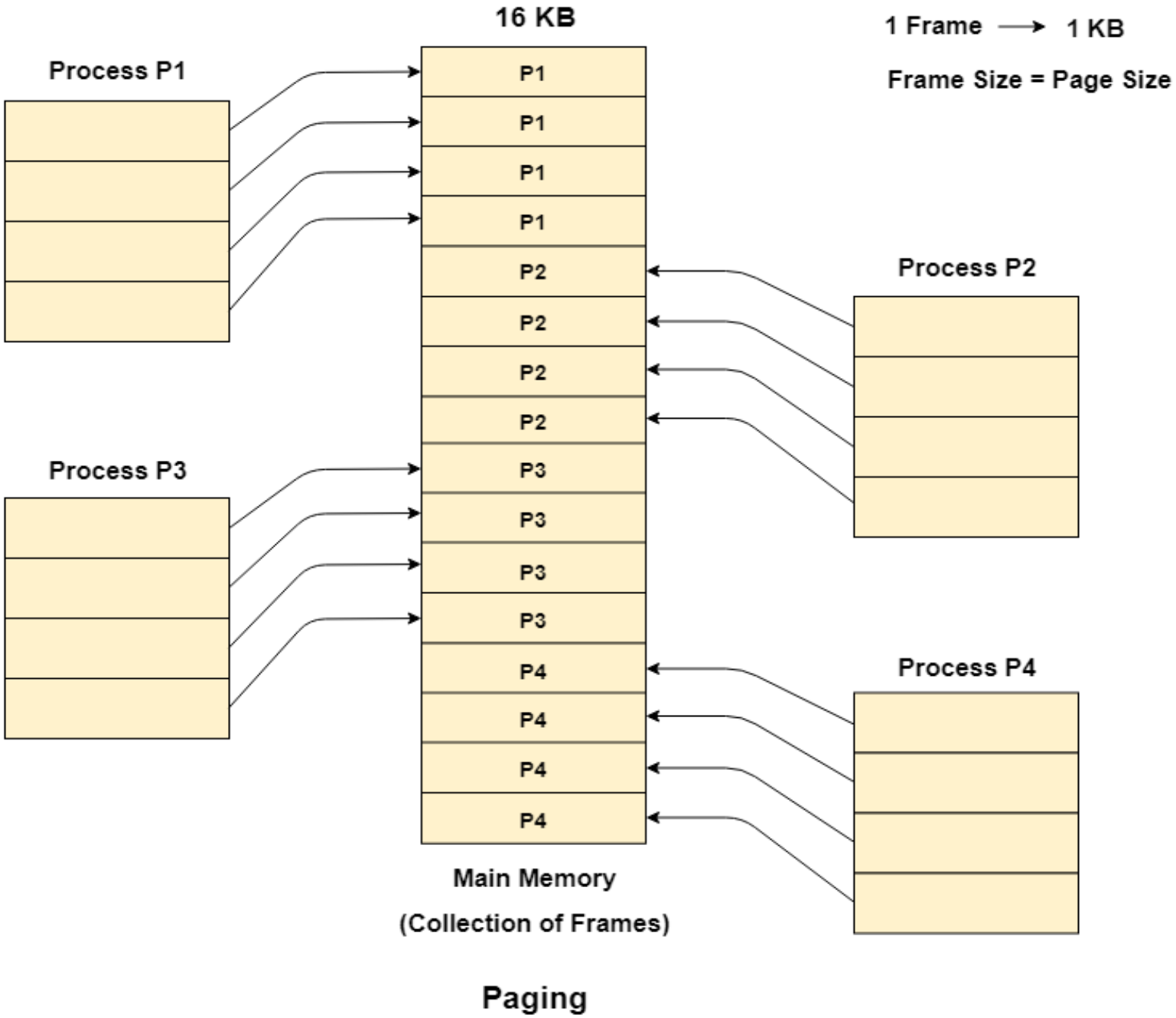
External Fragmentation in Dynamic Partitioning



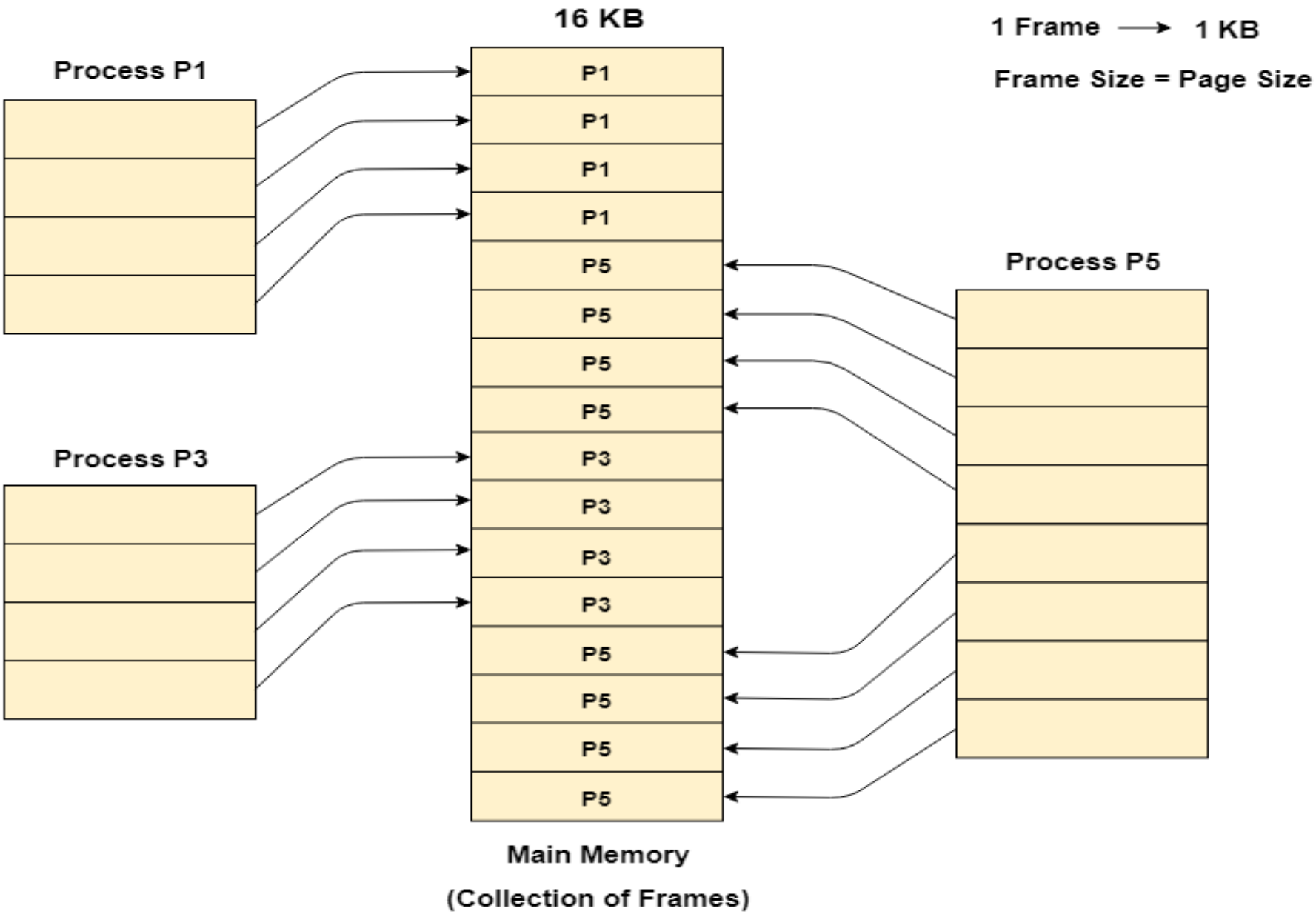
Now PS can be loaded into memory because the free space is now made contiguous by compaction

Compaction





Paging


16 KB
1 Frame → 1 KB
Frame Size = Page Size
Process P1
Process P3
Process P5
P1
P1
P1
P1
P5
P5
P5
P5
P3
P3
P3
P3
P5
P5
P5
P5
Main Memory
(Collection of Frames)

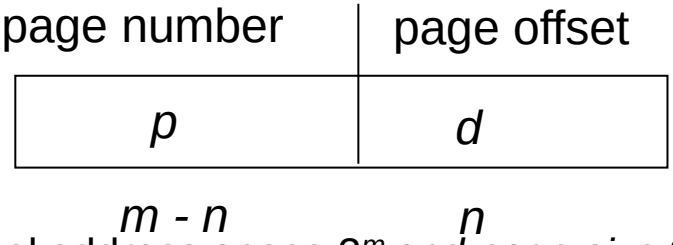
Paging

Paging

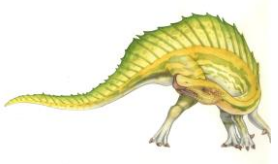
- Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
- Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 8,192 bytes)
- Divide logical memory into blocks of same size called **pages**
- Keep track of all free frames
- To run a program of size ***n*** pages, need to find *n* free frames and load program
- Set up a page table to translate logical to physical addresses
- Internal fragmentation

Address Translation Scheme

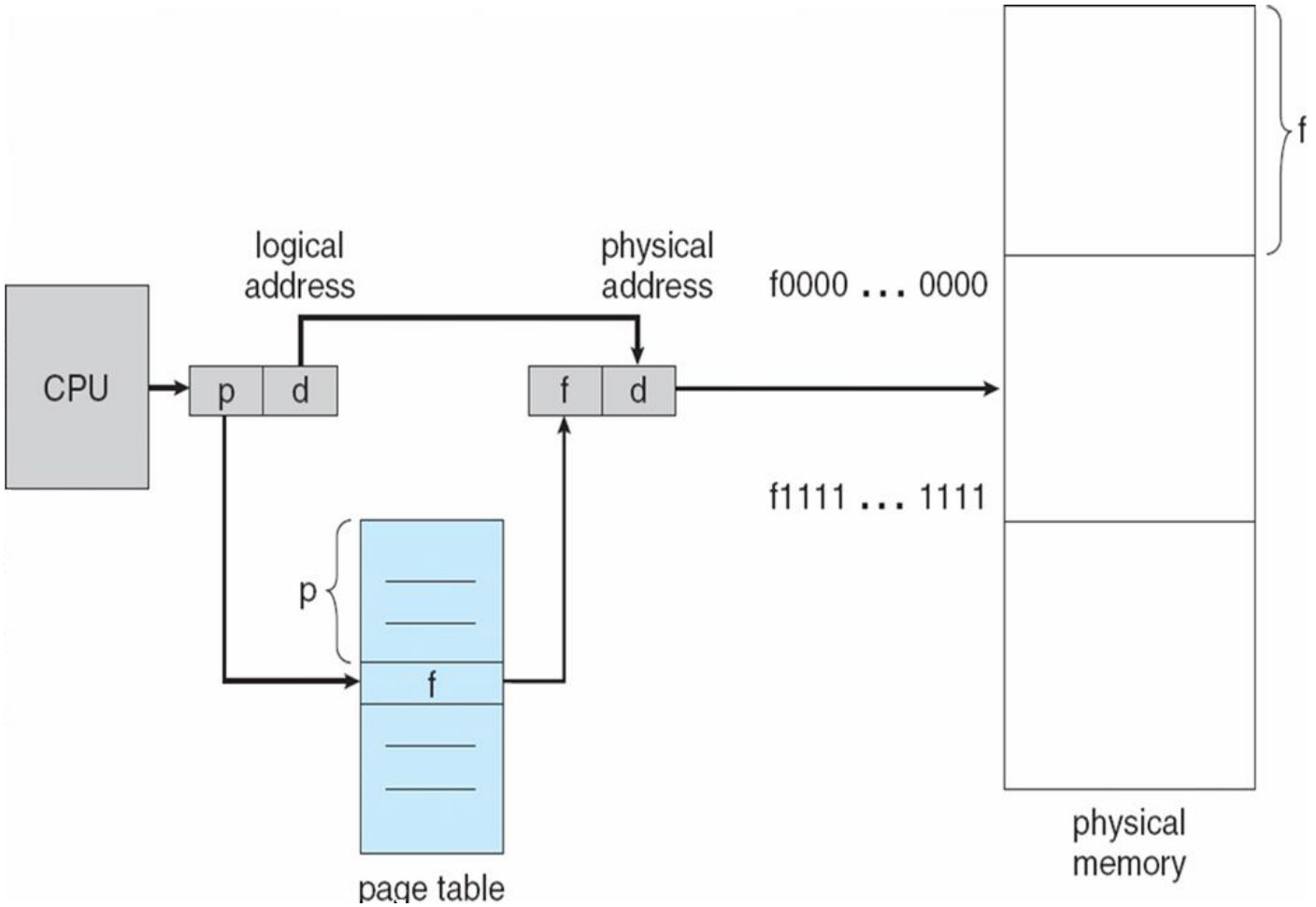
- Address generated by CPU is divided into:
  - **Page number (*p*)** – used as an index into a *page table* which contains base address of each page in physical memory
  - **Page offset (d)** – combined with base address to define the physical memory address that is sent to the memory unit



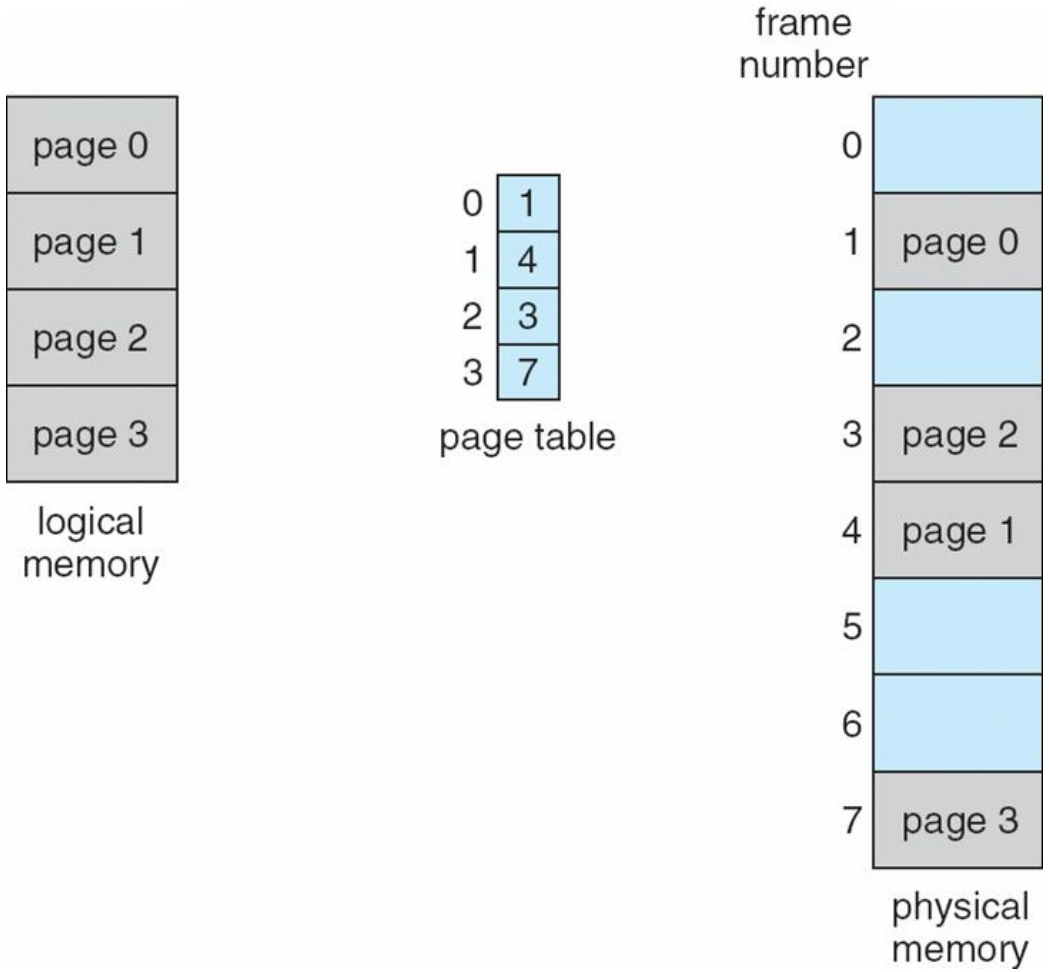
- For given logical address space 2^m *and page size* 2^n



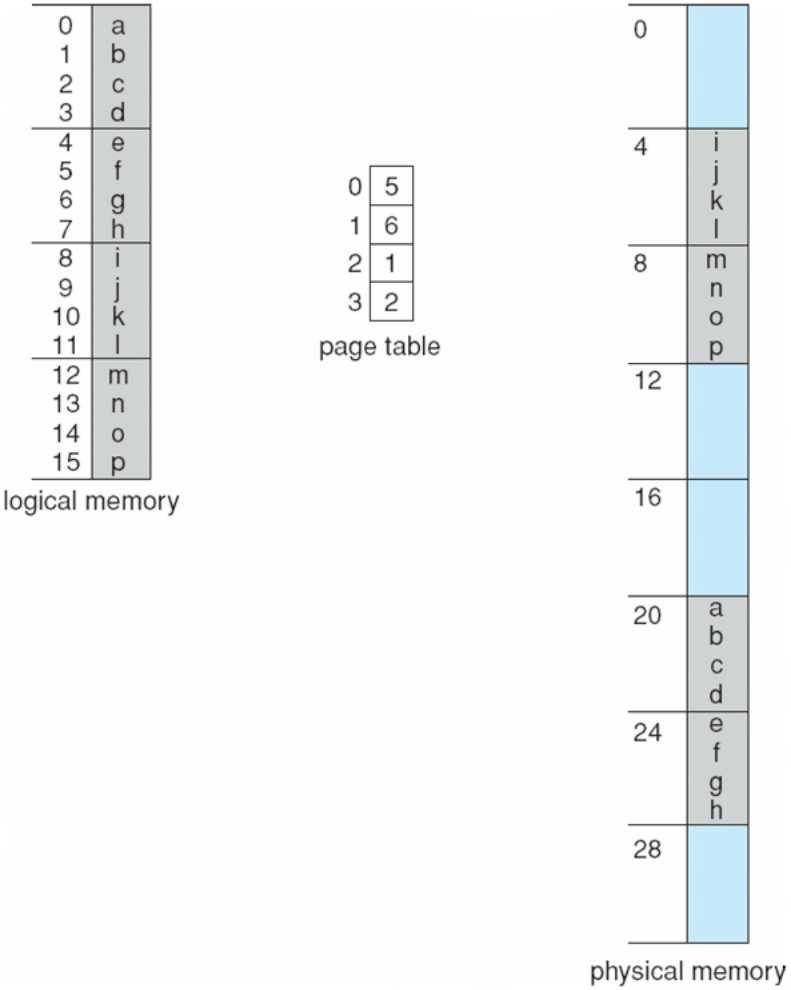
Paging Hardware



Paging Model of Logical and Physical Memory



Paging Example



32-byte memory and 4-byte pages

Segmentation

- Memory-management scheme that supports user view of memory
- A program is a collection of segments
  - A segment is a logical unit such as:

    main program

    procedure

    function

    method

    object

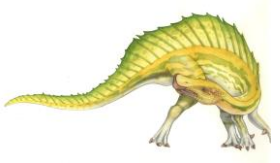
    local variables, global variables

    common block

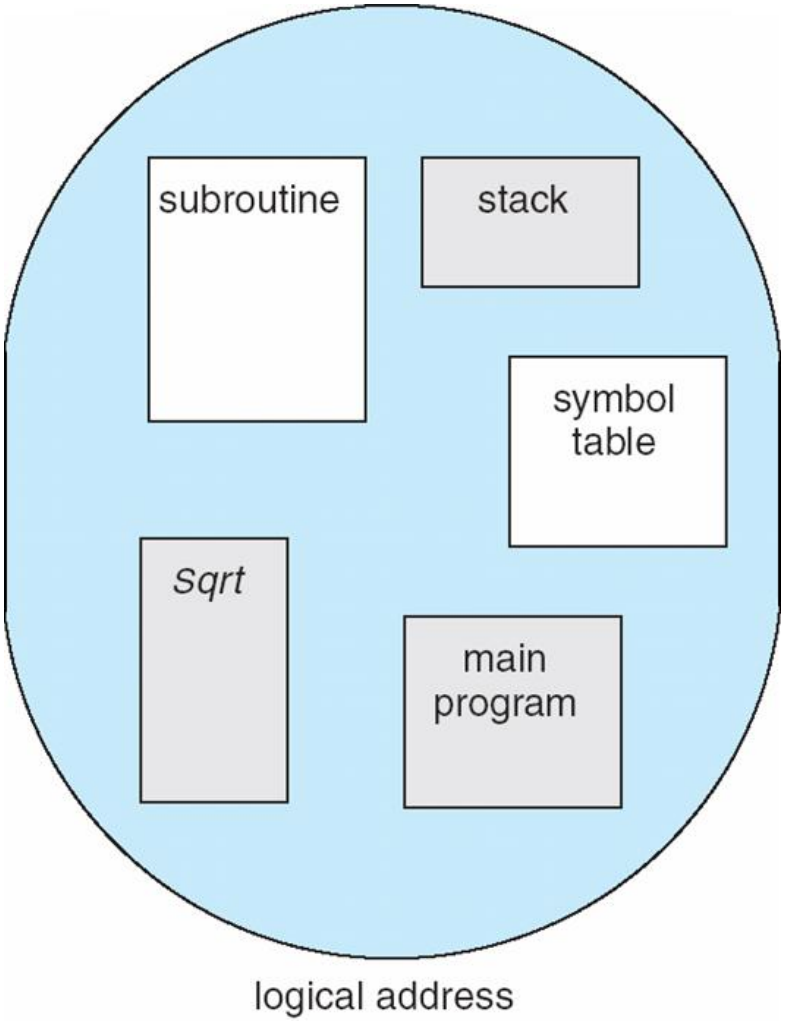
    stack

    symbol table

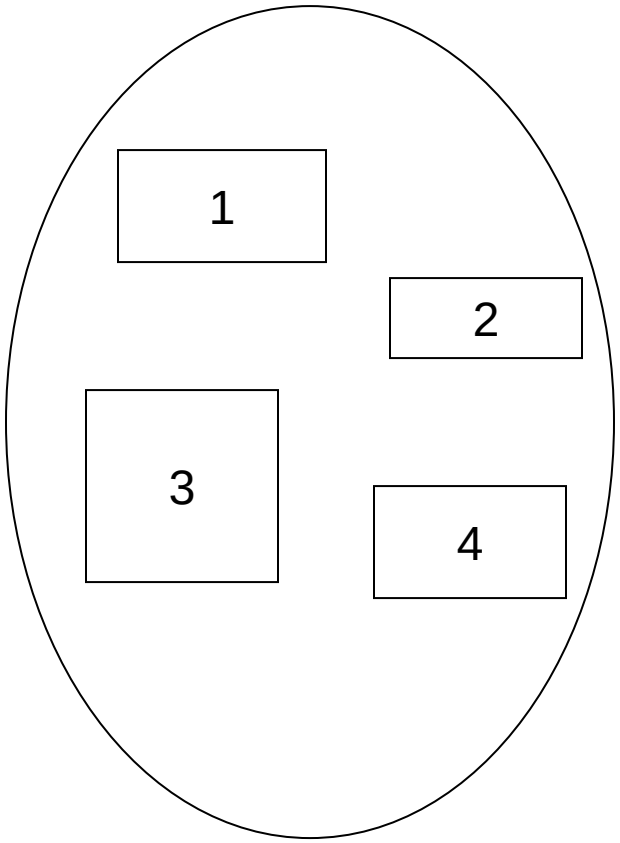
    arrays



User's View of a Program



Logical View of Segmentation



user space

physical memory space

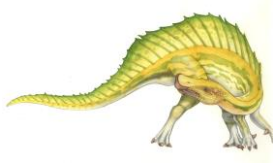
Segmentation Architecture

- Logical address consists of a two tuple:

  <segment-number, offset>,

- **Segment table** – maps two-dimensional physical addresses; each table entry has:
  - **base** – contains the starting physical address where the segments reside in memory
  - **limit** – specifies the length of the segment
- **Segment-table base register (STBR)** points to the segment table's location in memory
- **Segment-table length register (STLR)** indicates number of segments used by a program;

  segment number ***s*** is legal if ***s*** < **STLR**



Segmentation Architecture (Cont.)

- Protection
  - With each entry in segment table associate:
    - validation bit = 0 ⇒ illegal segment
    - read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem
- A segmentation example is shown in the following diagram