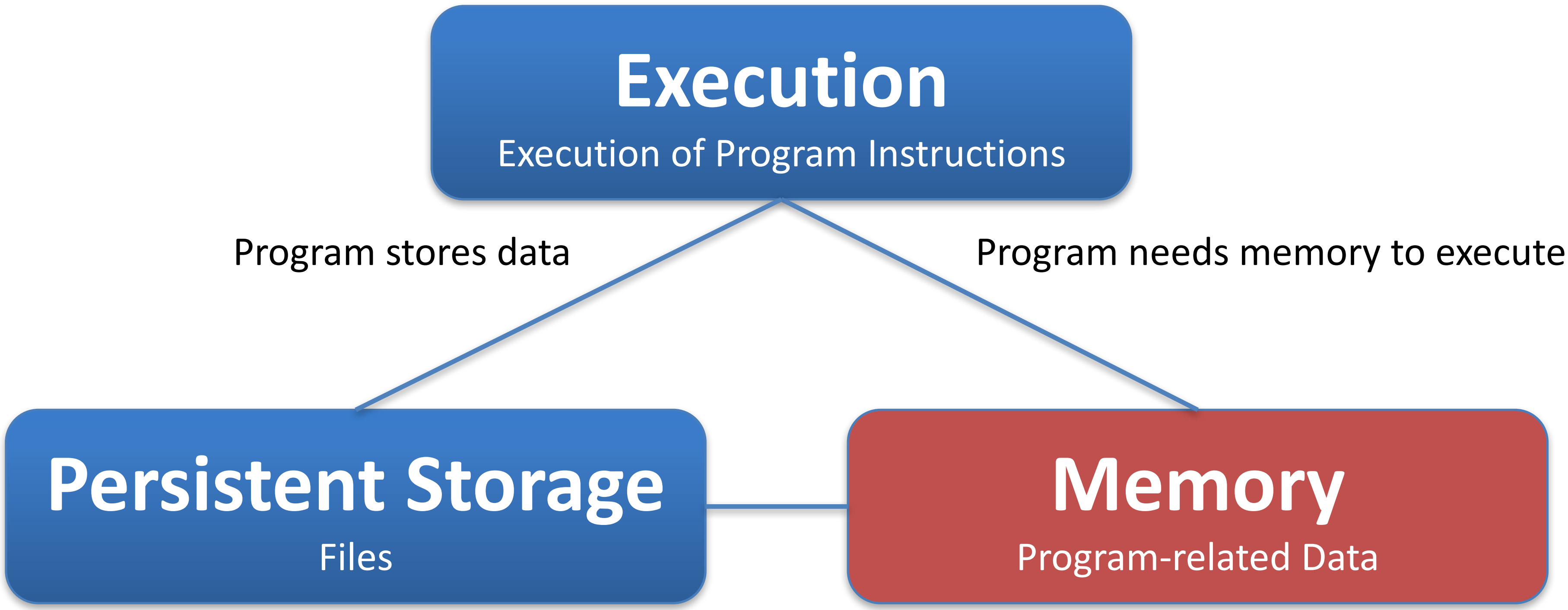
Memory Management

CS3026 Operating Systems

Lecture 07

Computer System Functions



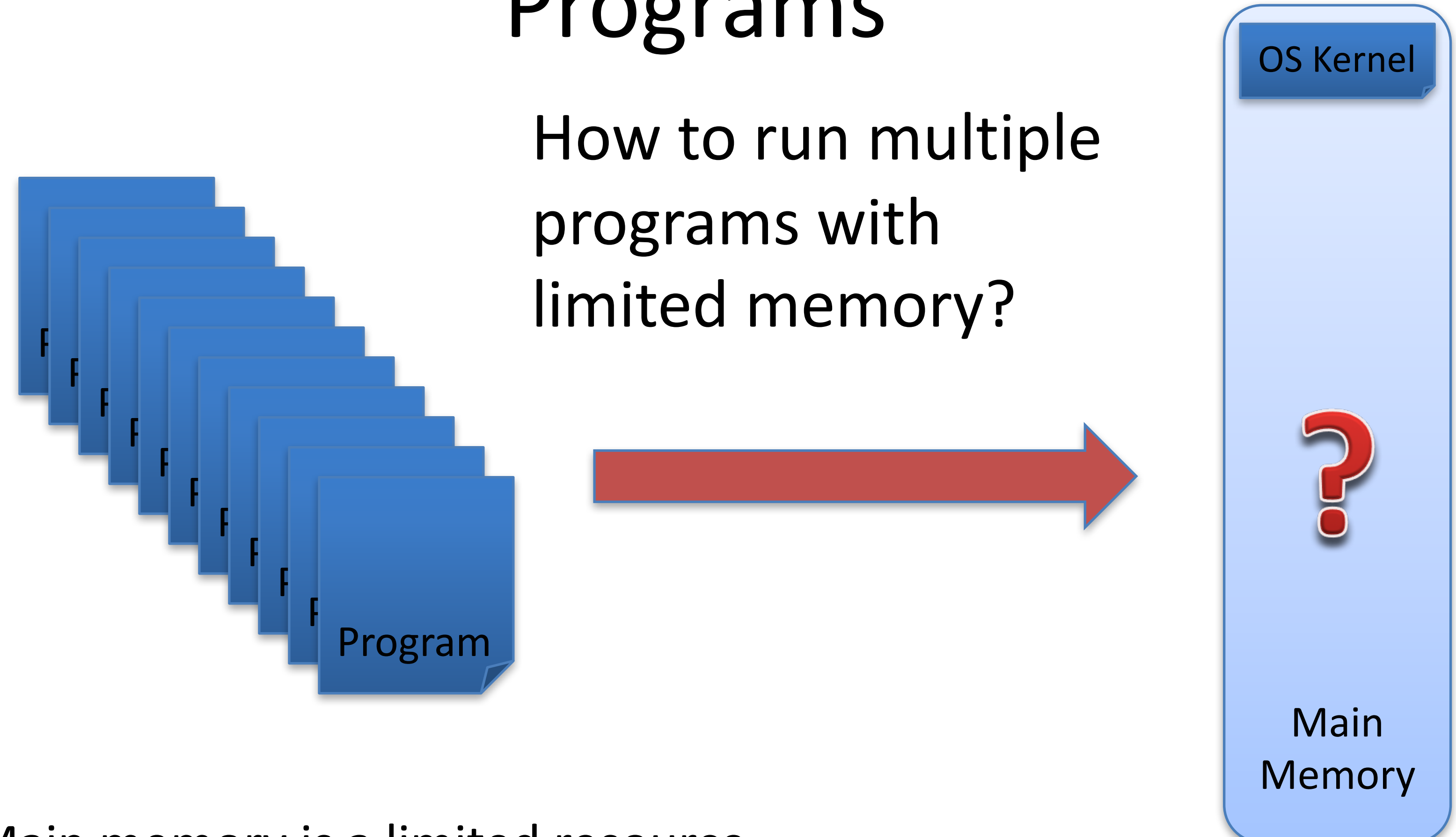
Memory Management

- Memory management is the effective allocation of memory to processes on a computer system
- An operating system has five principal storage management responsibilities
  - Automated allocation
  - Virtualisation
  - Support for modular programming (segmentation)
  - Process isolation, protection and access control
  - Long-term storage

Memory Management

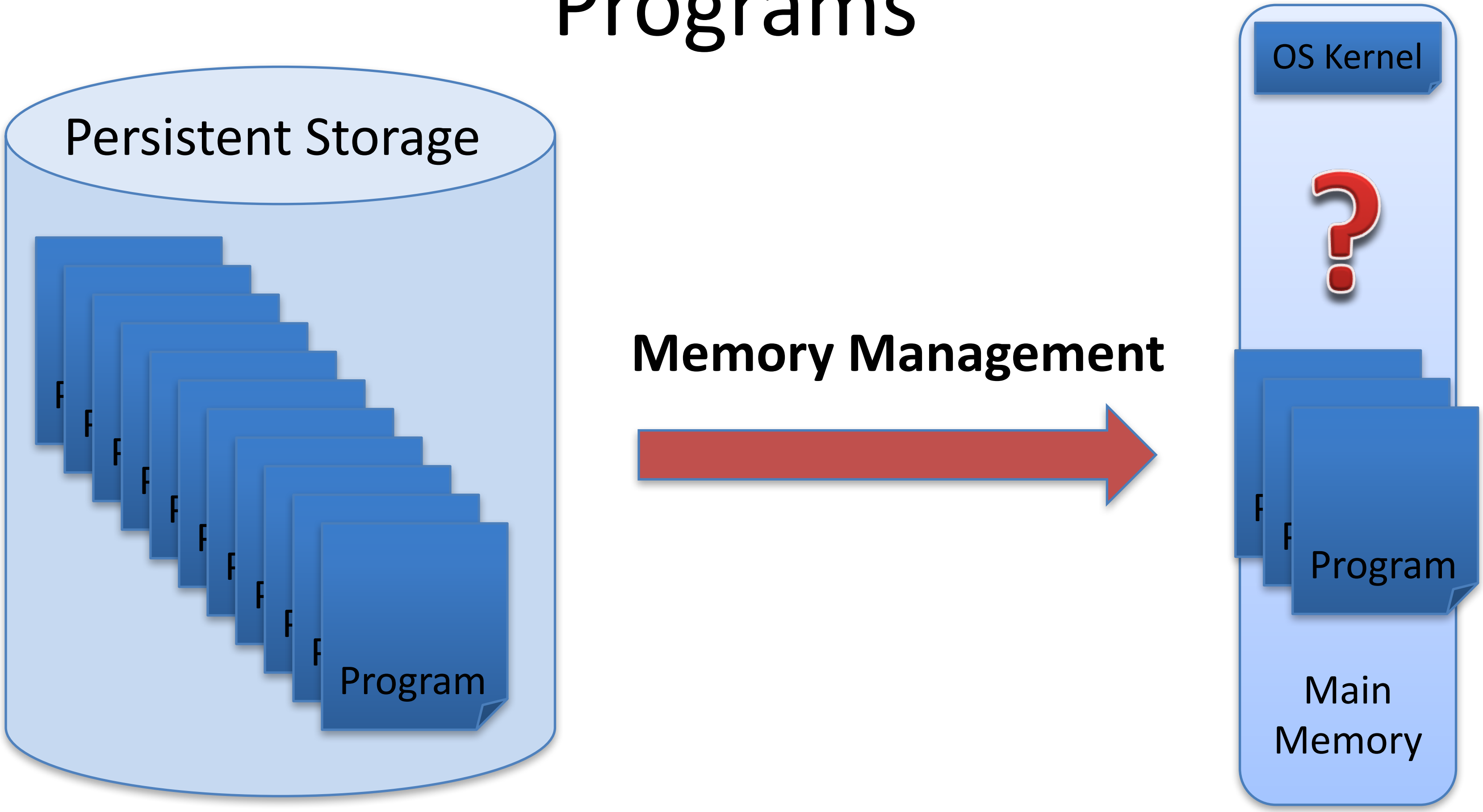
- Main memory
  - Large array of bytes
  - Addressable unit is called a “word”
    - E.g.: a 32-bit processor architecture addresses 32-bit / 4 byte words
  - Volatile
    - Memory looses content in case of system failure

Multitasking – Concurrent Execution of Programs



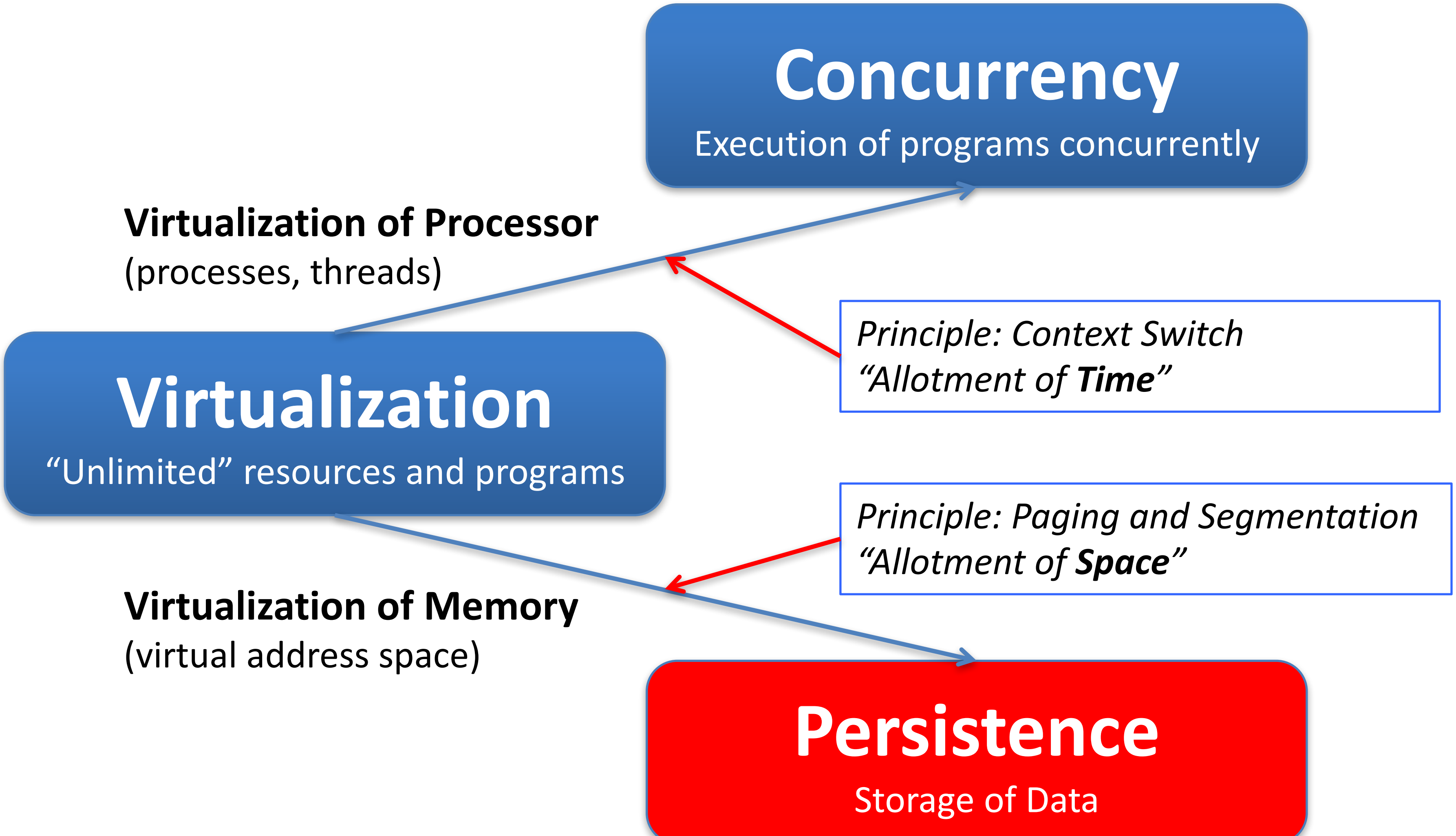
- Main memory is a limited resource
- A program needs memory for execution
- How to run multiple programs concurrently when only a few of them may be loaded into memory at the same time?

Multitasking – Concurrent Execution of Programs



- We need a memory management strategy to allow concurrent execution of programs

Core Concepts



Contiguous vs Non-contiguous Allocation

- Early approaches – contiguous allocation
  - Used partitioning of memory
  - Program (process image) fits into memory in its entirety
    - It occupies a contiguous block of physical memory
- Modern approaches – non-contiguous allocation
  - Use paging and segmentation
  - A process lives in a logical address space that is mapped to a set of non-contiguous physical memory locations (page frames)
    - One process image occupies different blocks of physical memory
    - Needs a mapping between **parts** of the process image and the actual physical **memory blocks** where they are currently stored

Historical Concepts
Partitioning of Memory

Contiguous Allocation

A program fits completely into memory

Partitioning

- An early method of managing memory
  - Memory was divided into partitions in order to allow multiple programs being held in memory at the same time
  - Goal: better CPU utilisation, multitasking
- No modern concepts, such as paging and virtual memory
- However: important precursor to modern memory management
  - Virtual memory management has evolved from partitioning methods into paging mechanisms

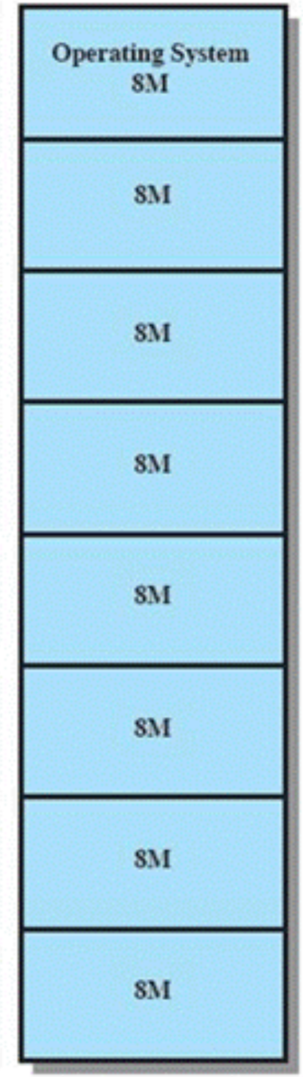
Partitioning

- A program is located in memory in its completeness
- Partitioning methods form the basis for modern memory management
  - Fixed partitioning
  - Dynamic partitioning
- Partitioning creates problems, may need garbage-collection approaches
  - Buddy system

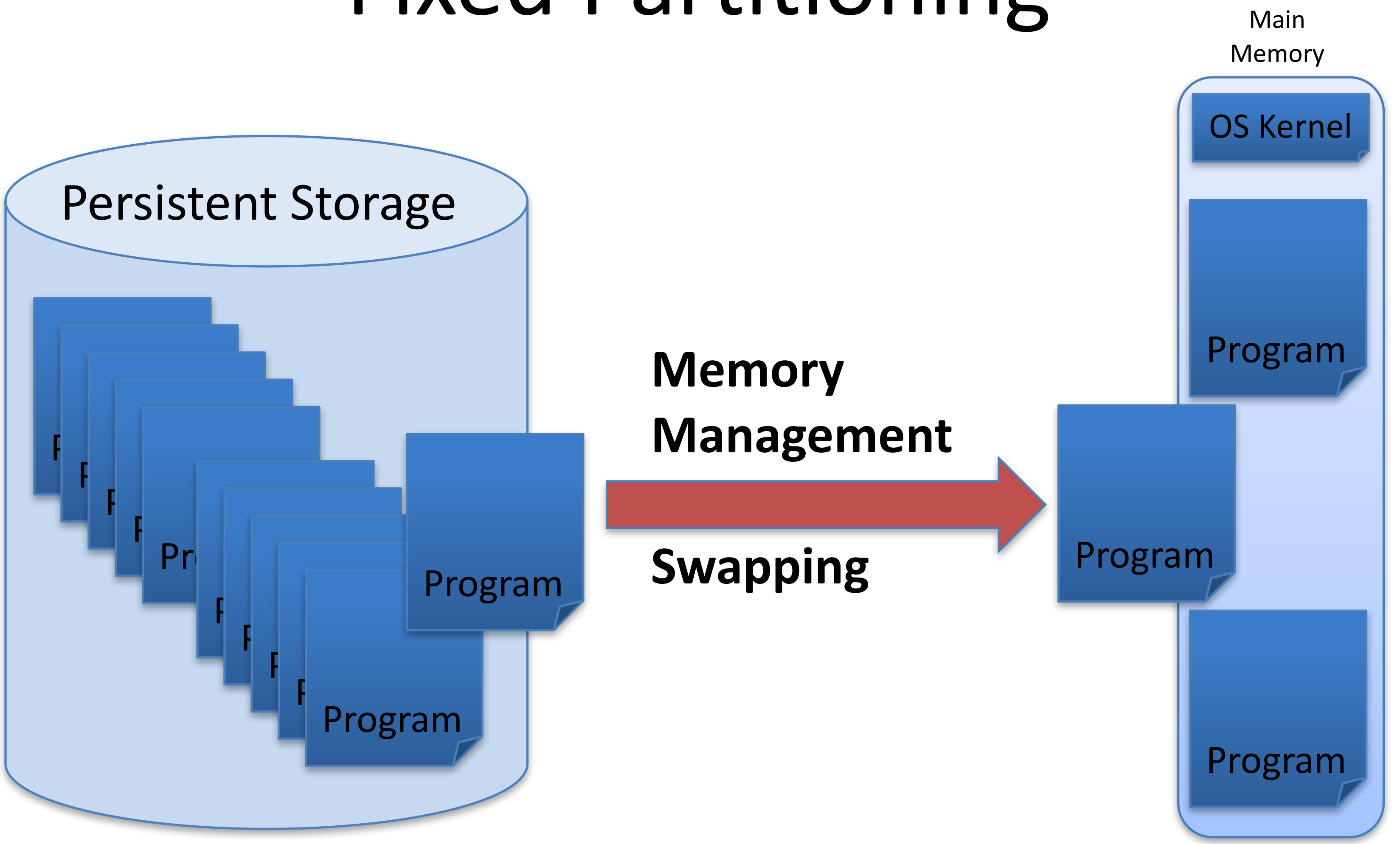
Fixed Partitioning

Equal Partition Size

- Static partitioning of physical memory
  - Memory organised as a set of fixed equal-sized partitions
  - No overlap of partitions
  - A program may be loaded into a partition of equal or greater size
- Number of partitions determines, how many programs may be loaded into memory at the same time
  - When all partitions are occupied, operating system may swap programs in and out of partitions
- Strength:
  - Simple to implement



Fixed Partitioning



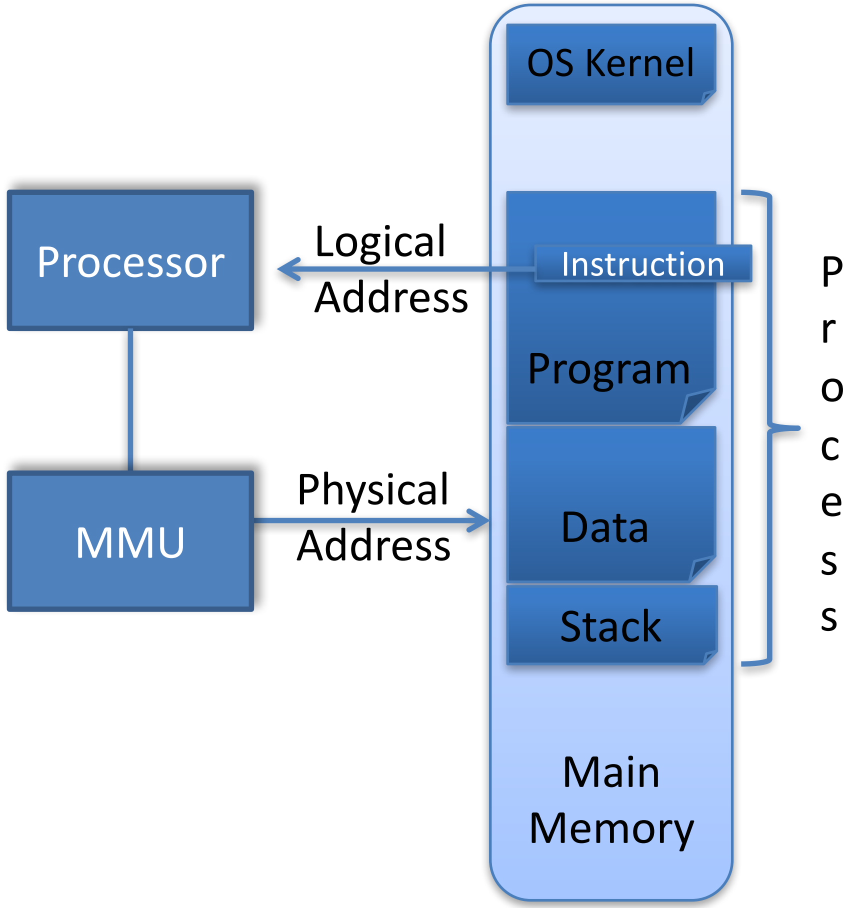
- Programs (process images) are swapped in and out of memory
- If memory is full – which program to swap?

Process Replacement, Swapping

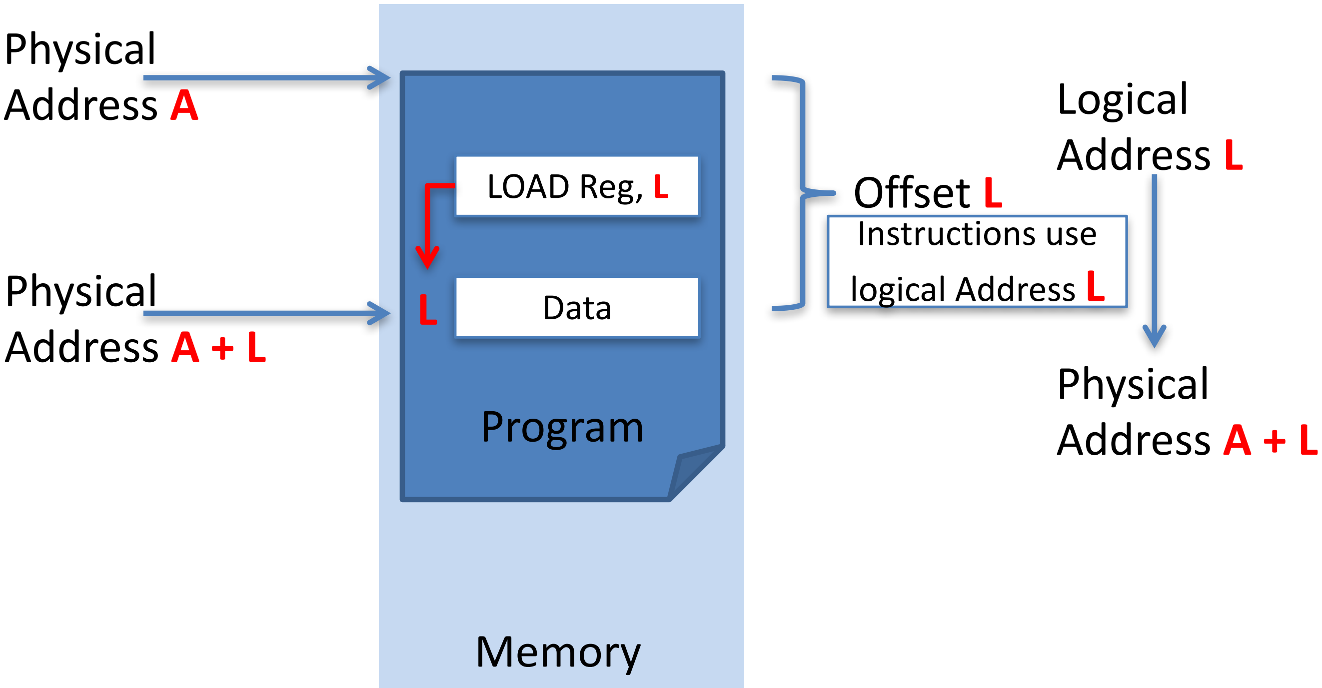
- Swapping and process replacement
  - If all partitions are allocated, a new process may have to wait for a partition to become free
  - ***Replacement decision***: Programs may be swapped out to make partitions available for new programs
    - Which program / process is the best one to replace?
  - ***Relocation decisions***: if process is swapped in
    - swapped into same partition as before? What about addresses used in program?
    - May need relocation mechanisms
- Arrival of new process
  - Will be added to a waiting queue for memory allocation
  - Either one queue for all partitions or a separate queue for each partition

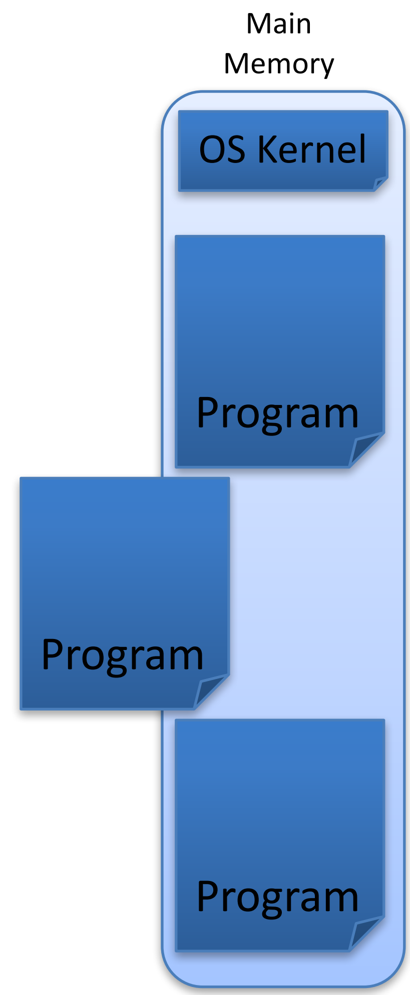
Relocation with Relative Addressing Logical vs Physical Addressing

- Processes (programs in execution) should be able to reside at any location in memory
- This is achieved by a **virtual** or “relative” addressing scheme, that calculates a “physical” memory address from the program-specific “logical” address
- Logical address (also called “virtual” or “relative” address)
  - A value that specifies a generic location relative to the start of the program
- Physical address
  - The actual address of a location in the physical memory of a computer system



Relative Addressing

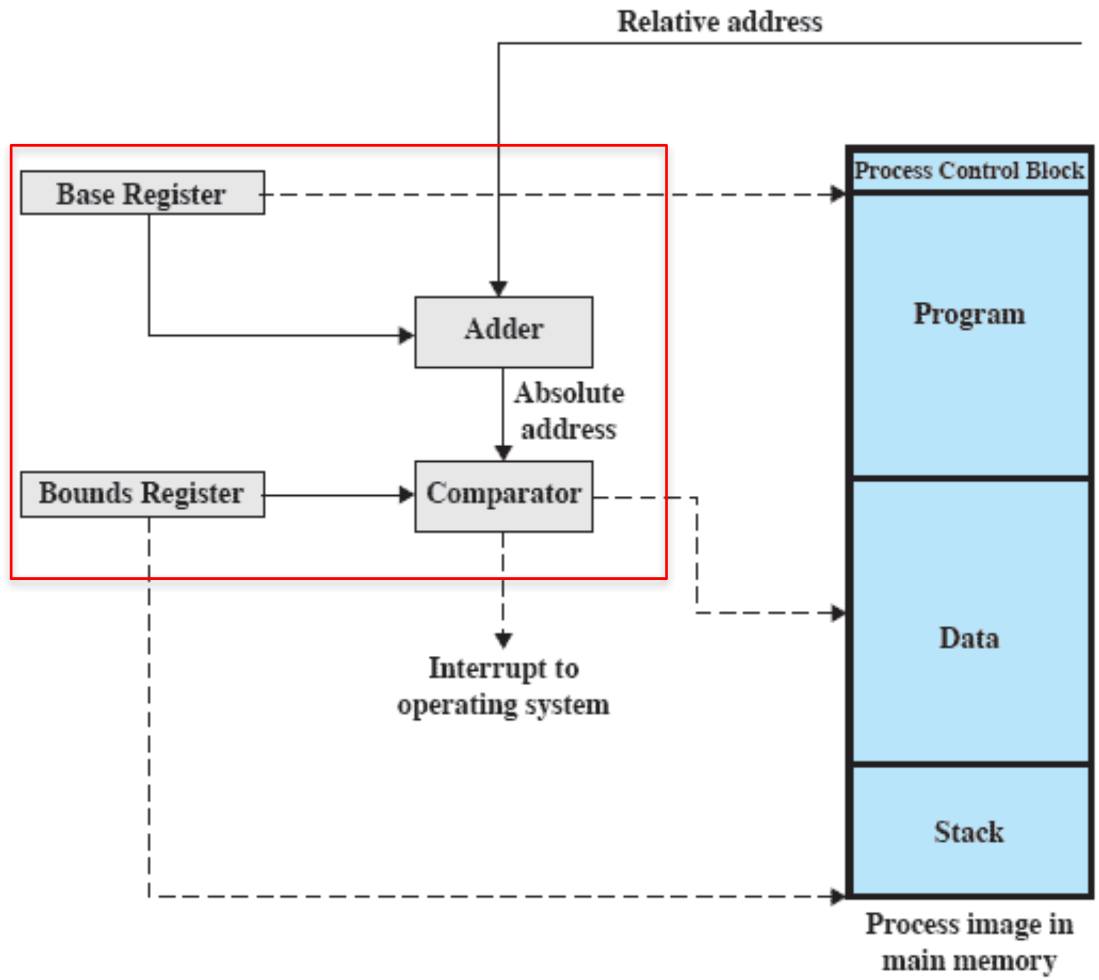




- Solution:
  - Programs operate as if they all start at address 0
- Programs operate with relative addressing mechanism:
  - Reference to memory in program independent of actual physical location
  - Addresses in program are offsets from its base address (the starting address of the process image)
  - add the base address whenever there is an access to memory

Relocation with Relative Addressing

- Address translation with hardware
- Two registers
  - Base register: physical start address of the process image in memory
  - Bound register: physical end address of the process image in memory



Relocation with Relative Addressing

- The address space of a process is mapped onto physical memory
- Address translation with hardware
- Two registers
  - Base register: physical start address of the process image in memory
  - Bound register: physical end address of the process image in memory
- Memory access
  - Process uses relative address for memory location
  - This access is translated into a physical address by adding the content of the base register to the relative address

Problem: Memory Waste

- “Not all programs are equal”
- Programs may be of different size and, therefore, different memory requirements
- We can load a program only into partitions that provide enough space
- If program is smaller than partition – memory waste, cannot be allocated to other program
- If a program is larger, we cannot run it

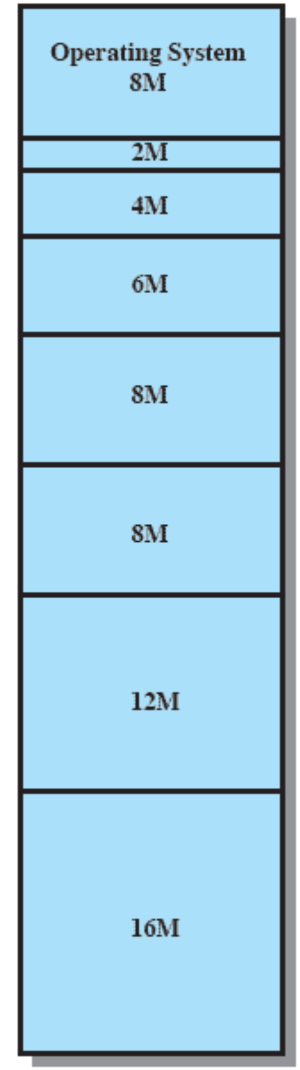
Problem: Internal Fragmentation

- Inefficient use of main memory
  - Process image may not use the complete partition, some waste of memory inside the partition
  - Waste of memory: a partition is *internally* fragmented, fragments of a partition not used cannot be given to other processes
- This phenomenon is called ***internal fragmentation***
  - Partition is not completely used
  - the block of data loaded (swapped in) is smaller than the partition
- Whenever there is a fixed-size partitioning, internal fragmentation may occur

Fixed Partitioning, Unequal Size

Minimize Internal Fragmentation

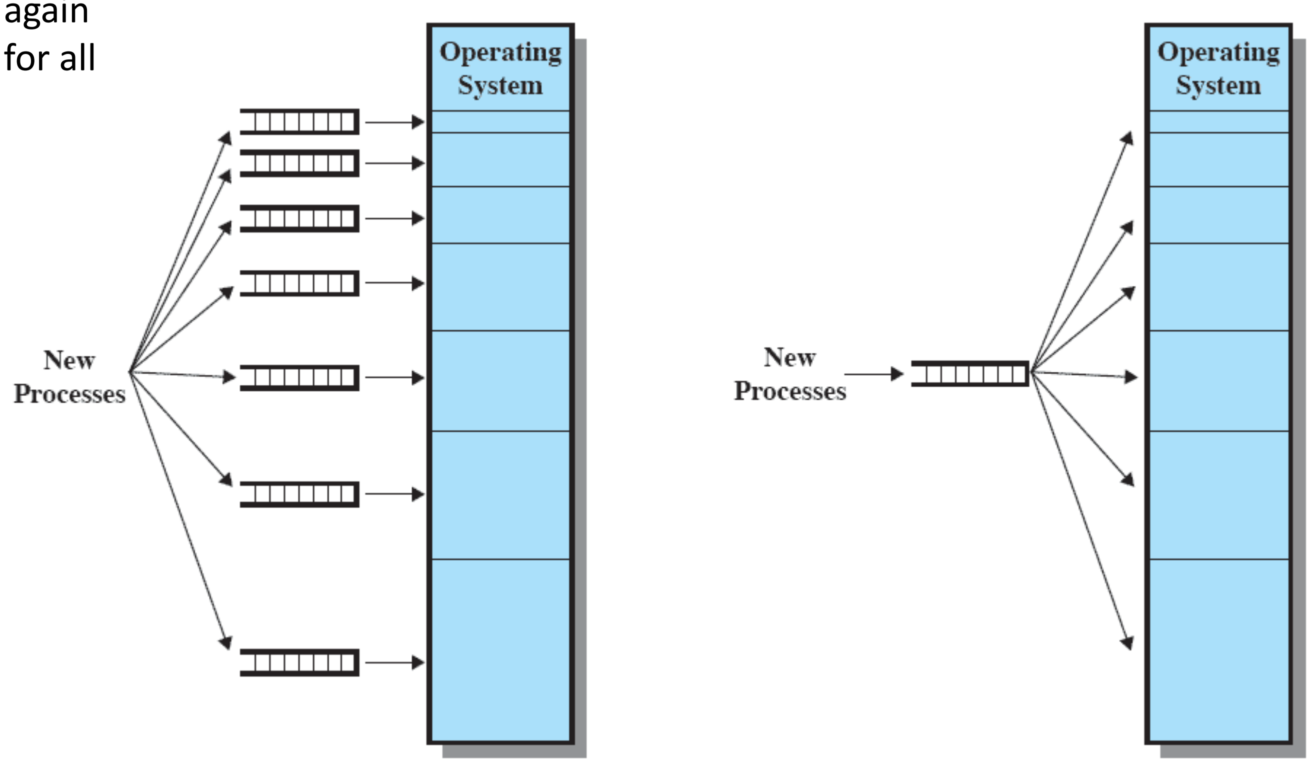
- Using unequal partitions helps to lessen the problem of internal fragmentation
  - Partitions of different sizes pre-allocated in memory
  - Processes are placed into partitions where they fit best to minimize internal fragmentation
- Tries to minimise internal fragmentation
- Doesn't solve the problem completely



Process Replacement

Memory Assignment Problem

- Unequal size creates another problem: how to best assign a memory partition?
- With unequal-size partitions, there are two possible ways to assign processes to partitions
  - One queue per partition, holding swapped-out processes that wait for this partition to become available again
  - One single queue for all



(a) One process queue per partition

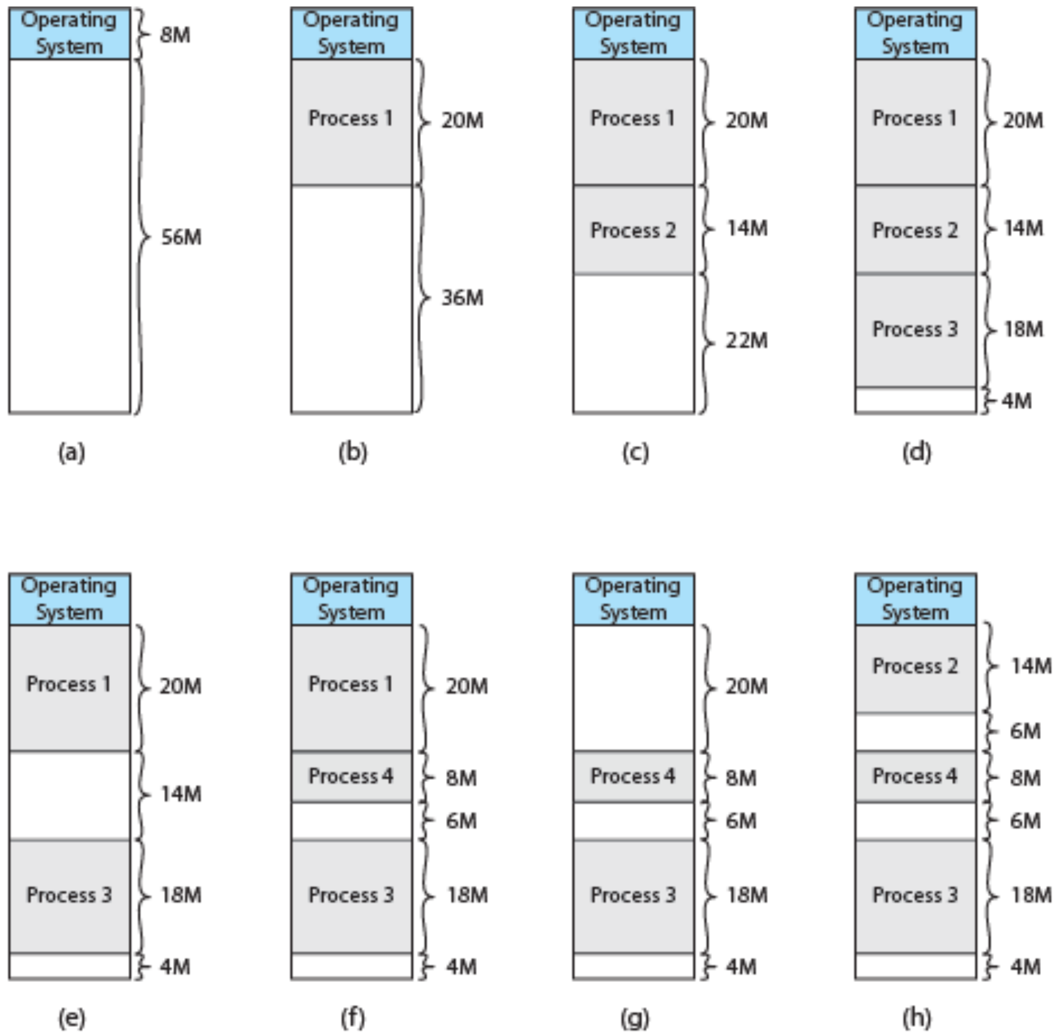
(b) Single queue

Dynamic Partitioning

On-demand allocation of a Slice of Memory

- No pre-partition at system start, memory is one contiguous partition
- Processes that are swapped in, are allocated the exact amount of memory needed
- Partitions are of variable length and number
  - Have the exact size needed
- Memory is allocated “on-demand” to processes
  - When a new process arrives or is swapped in, the exact amount of memory needed by the process is allocated as a partition
- When a process terminates or is swapped out, the partition is released

Effect of Dynamic Partitioning



Problem: External Fragmentation

- Dynamic Partitioning
  - Over time, fragmentation of memory into smaller and smaller pieces occurs (“garbage”)
  - If a free *contiguous* space is large enough, it can be re-allocated for new processes
  - If free space is too small for new processes, it remains unused
    - Processes have to wait for memory large enough to become available
- External memory fragmentation
  - Waste of memory outside partitions, free memory areas too small for re-allocation
- Whenever there is a variable-size partitioning, external fragmentation may occur

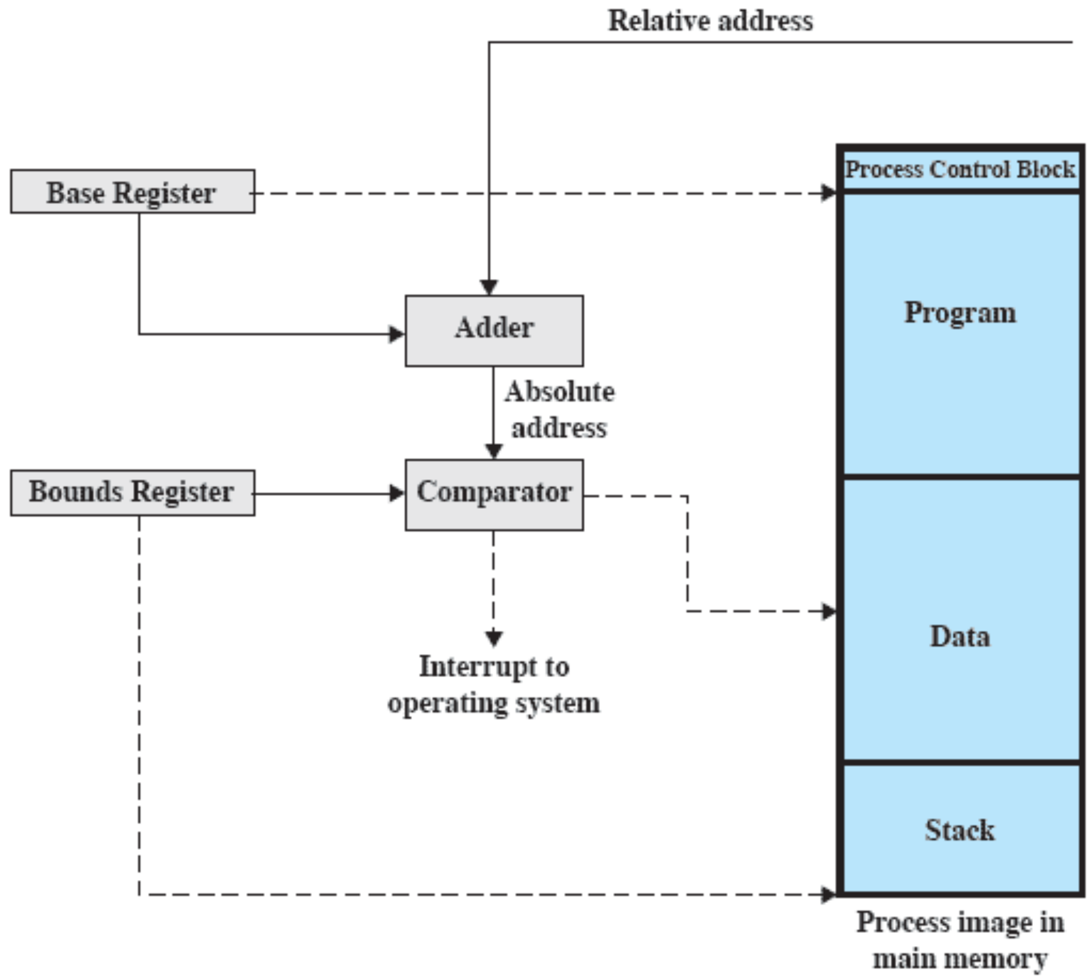
Problem: External Fragmentation

- The left-over memory not assigned to partitions may be too small for any new process to fit in – waste of memory
- This phenomenon is called ***external fragmentation***
  - Memory external to partitions becomes increasingly fragmented
  - Memory utilisation declines, number of processes held in memory declines
- **We need some form of "garbage collection"**

Strategy: Compaction

- In order to overcome external fragmentation, a technique called *Compaction* was proposed:
  - This is a kind of “garbage collection”
  - partitions with assigned processes are shifted to collect free memory in one contiguous block
- However: Compaction requires dynamic relocation capabilities in an operating system
  - It must be possible to move programs in memory and re-adjust all of its memory references
  - Is time consuming and a waste of CPU time

Compaction needs Reallocation Capabilities



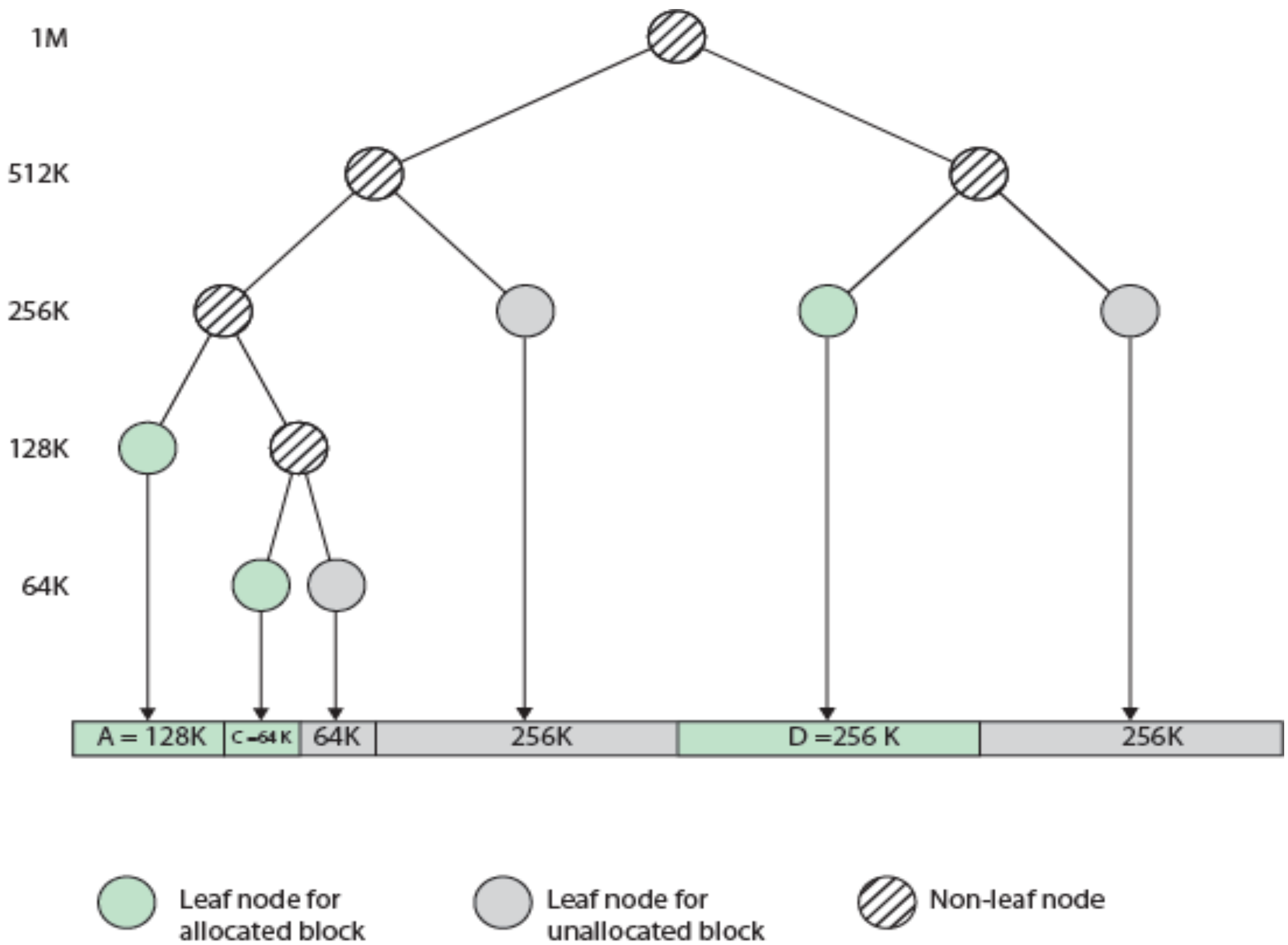
Dynamic Placement Strategies

- Memory compaction is costly, therefore try to allocate free memory for processes as clever as possible
- Three methods
  - Best-fit
    - Find the smallest partition needed
  - First-fit
    - Start scanning memory from its start address and take the first free block that is large enough
  - Next-fit
    - Continue scanning memory for free space from last allocation position

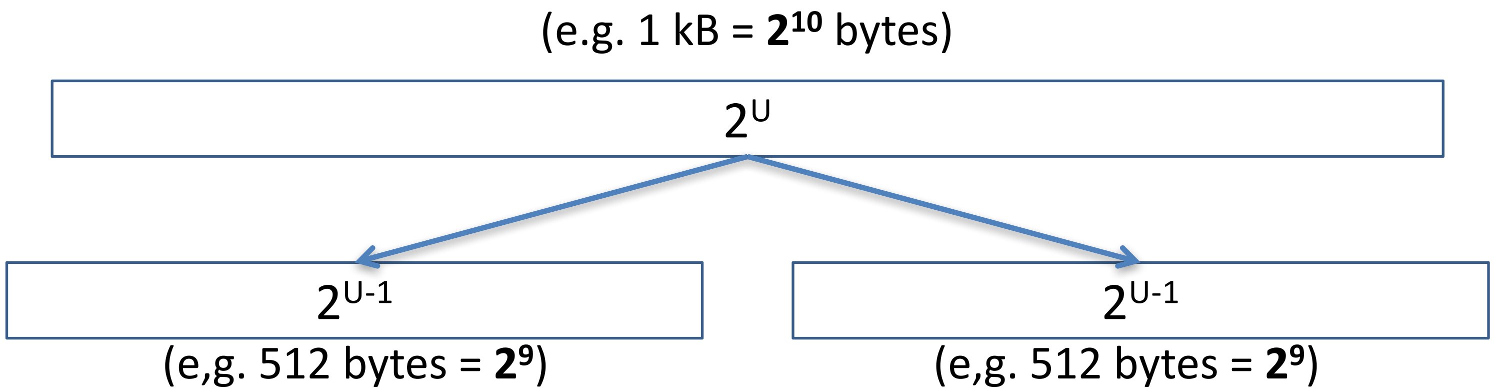
Buddy System

- Try to fix the problems of both static and dynamic partitioning
- Combines fixed and dynamic partitioning scheme
  - Is either allocating a complete block that is small enough for a process or splitting a block into two equal “buddies” to get a better fit
- Minimize internal fragmentation
- Allows the creation of variable sized partitions, however the size of a partition is always 2^N
  - A relationship between these partitions is maintained

Tree Representation of Buddy System



Buddy System



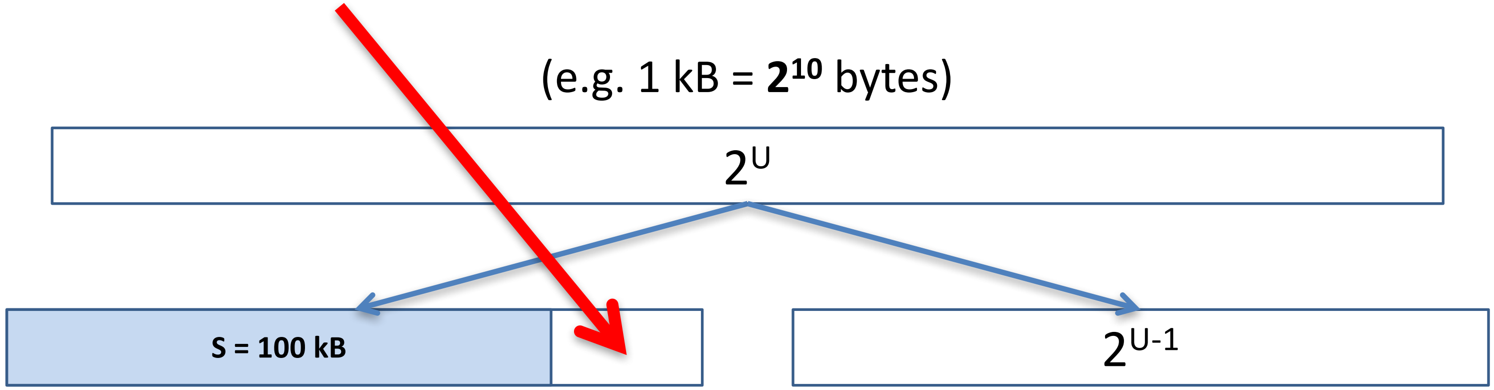
- Maintains relationship between partitions
  - If a partition itself is partitioned into two new and smaller partitions, then they are related
  - Related free partitions can be recombined into a larger contiguous space, as OS memorizes their relationship

Please note: $2^{10} = 2^9 + 2^9$

Buddy System

- Start: entire available space for allocation is treated as a single block of size 2^U
- Process arrives, a memory block of size S is needed:
  - If for the request $2^{U-1} < S <= 2^U$ holds, then a block with size 2^U is allocated
  - Otherwise, the block of size 2^U is split into two “buddies” of size 2^{U-1}
    - Please note: $2^U = 2^{U-1} + 2^{U-1}$
  - This continues, until the smallest block greater or equal to the requested size S is allocated
- If blocks become free, they can be recombined into the original contiguous blocks
  - This avoids memory fragmentation and the need for compaction

Buddy System



- When memory is allocated, it is always allocated in blocks of size 2^K, so that
  - 2^K is the best fit for a requested memory of size $S \le 2^K$
  - If we have a block of size 2^U that is too large, it is split into two blocks of size 2^{U-1} and again it is tested again whether 2^{U-1} is a best fit for the requested memory of size S

Please note: $2^{10} = 2^9 + 2^9$

Buddy System

Initially:

1 MB

Request: allocate **100 kB**

$2^6 = 64\text{kB} < \mathbf{100}\ \text{kB} <= 2^7 = 128\text{kB}$

Split: create a 128kB partition

128 kB	**128 kB**	**512 kB**	**512 kB**

Allocate: load 100 kB into partition

100 kB		**128 kB**	**512 kB**	**512 kB**

Internal fragmentation

Memory Management

- Functional Requirements:
  - Relocation
    - Process images may be positioned at arbitrary locations in memory and may be relocated
    - Important for Virtual memory management
  - Partitioning
    - Create memory partitions and allocate them to processes
  - Security and Isolation
    - Protect segments of memory, isolate memory areas of processes
    - Needs hardware support
  - Sharing
    - Allow shared access to memory by different processes

Memory Management

- Non-Functional Requirements
  - Performance
    - Minimal overhead of memory management
    - Fast allocation
    - Avoid thrashing
  - Fairness
    - Avoid starvation of processes
    - Tune Working set according to process needs

Meeting the Requirements

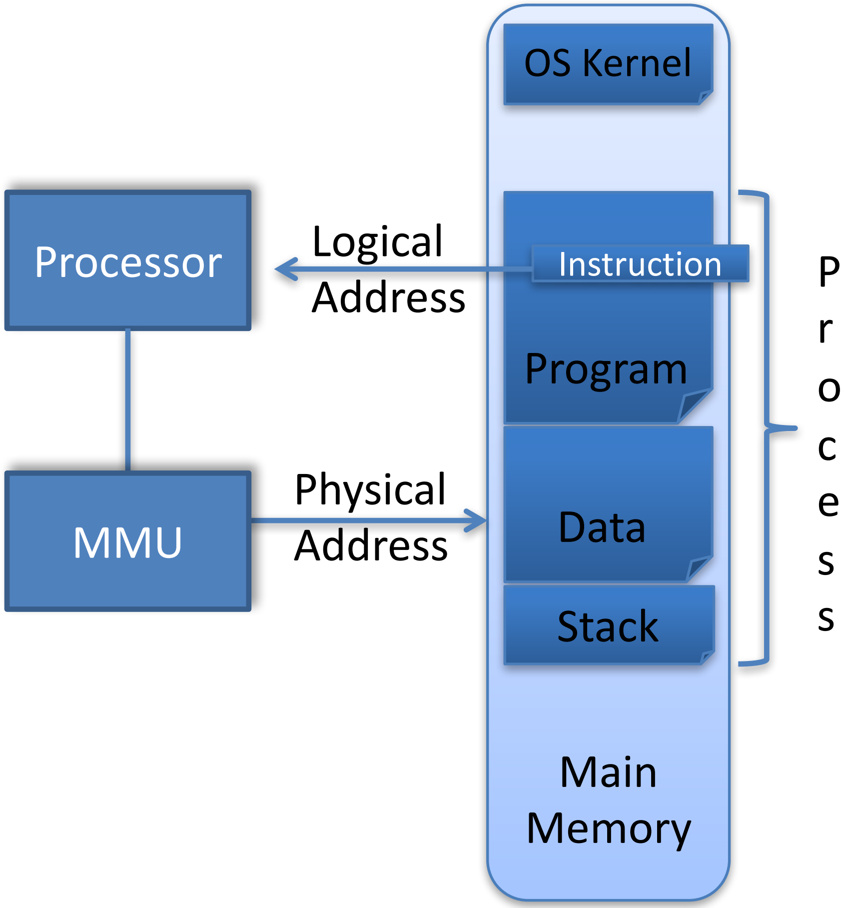
- Modern systems use virtual memory
  - Supported by hardware and software
  - Programs are not restricted in size by actual physical memory
  - Number of processes executing on system is not limited by physical memory
  - Based on paging and segmentation
- We look at historical concepts to understand the development of virtual memory concepts
  - Partitioning (fixed, dynamic)
  - Segmentation (evolved from dynamic partitioning)
  - Paging (evolved from fixed partitioning)

Meeting the Requirements

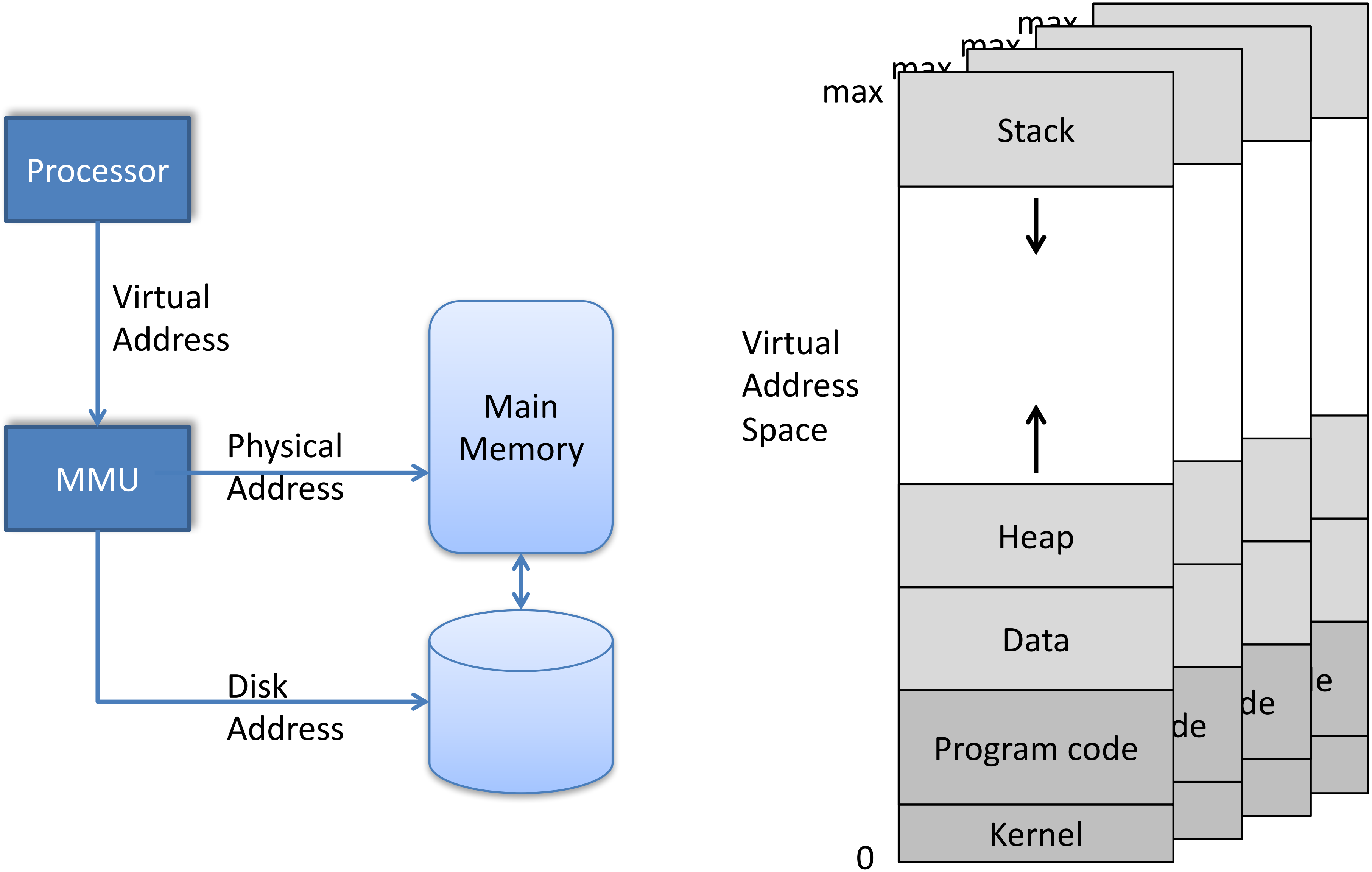
- Logical Organisation
  - User programs operate within a virtual address space
  - User programs have a modular structure, each process is divided into segments (code, data, stack), mechanisms for protecting segments (read-only, execute-only) and sharing among processes
- Physical Organisation
  - Virtual memory based on a two-level hierarchy between physical memory and disk space
  - CPU can only access data in registers and physical memory
- Address Translation
  - MMU Memory Management Unit translates logical address into physical address

Logical vs Physical Addressing

- Processes (programs in execution) should be able to reside at any location in memory
- This is achieved by a **virtual** or "relative" addressing scheme, that calculates a "physical" memory address from the program-specific "logical" address
- Logical address (also called "virtual" or "relative" address)
  - A value that specifies a generic location relative to the start of the program
- Physical address
  - The actual address of a location in the physical memory of a computer system



Address Translation



Address Space vs Physical Memory

- Processes reside within the address space of the processor
  - All (theoretically) addressable memory locations
  - Depends on size of address register
    - 32-bit architecture: 4GB of addressable memory locations
    - 64-bit architecture: 16ExaBytes of addressable memory locations

Bytes	Exponent			How to calculate
1,024	2^{10}	1kb	1024bytes	
1,048,576	2^{20}	1MB	1024kb	1024 x 1024
1,073,741,824	2^{30}	1GB	1024MB	1024 x 1024 x 1024
4,294,967,296	2^{32}	4GB	4 x 1024MB	4 x 1024 x 1024 x 1024
1,099,511,627,776	2^{40}	1TB	1024GB	1024 x 1024 x 1024 x 1024
1,125,899,906,842,620	2^{50}	1PB	1024TB	1024 x 1024 x 1024 x 1024 x 1024
1,152,921,504,606,850,000	2^{60}	1EB	1024PB	1024 x 1024 x 1024 x 1024 x 1024 x 1024
18,446,744,073,709,600,000	2^{64}	16EB		16 x 1024 x 1024 x 1024 x 1024 x 1024 x 1024

Buddy System

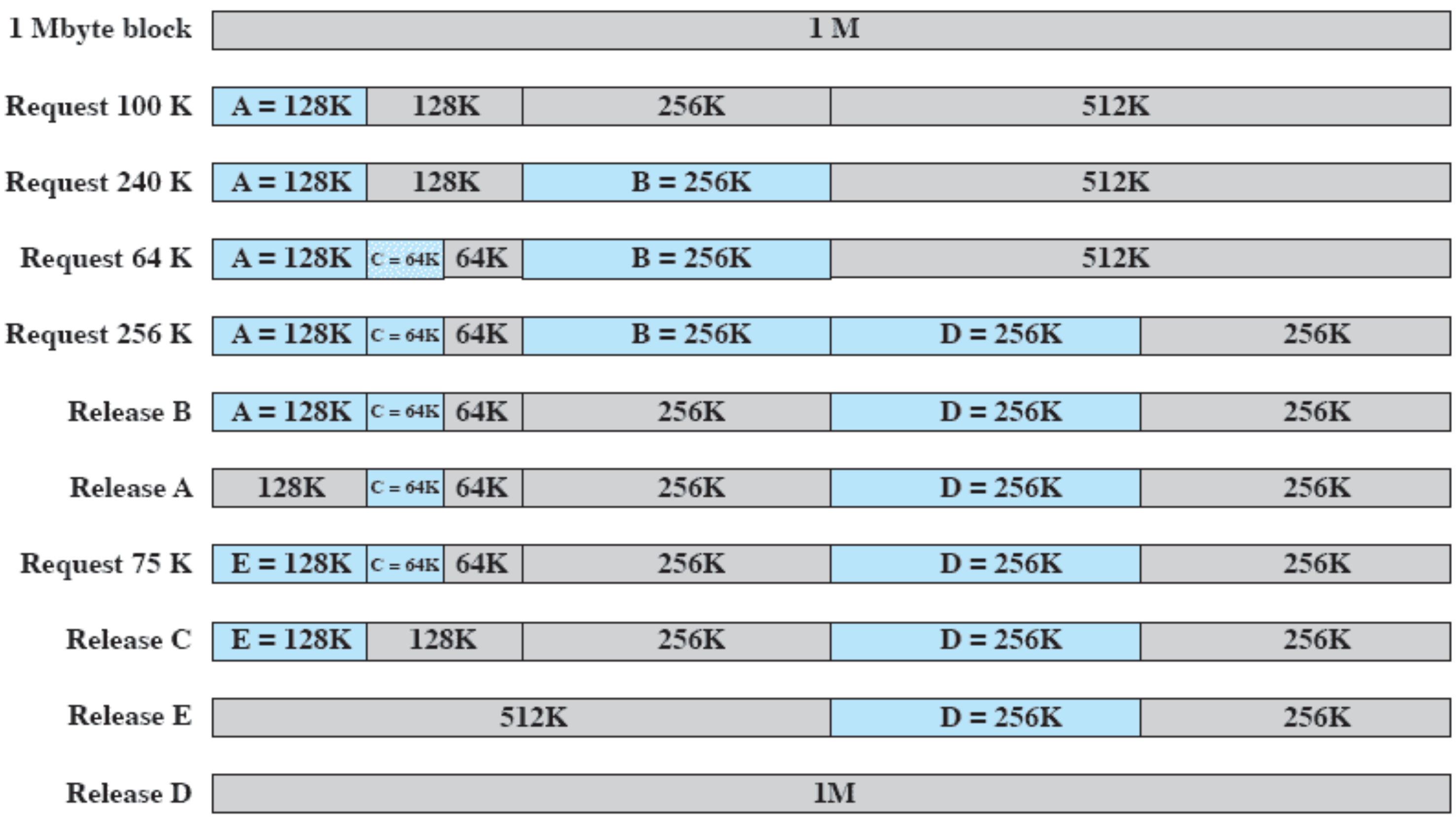


Figure 7.6 Example of Buddy System

Relocation

Addressing Memory

- A programmer cannot know in advance where in memory a program will be loaded
- Compiler produces code that refers to memory locations
  - These addresses cannot be absolute (physical addresses), but have to be "relative" to the start address of the program
- Linker combines pieces of a program (with libraries) into a loadable image
  - Assumes the program to be loaded at address 0
- Addresses generated by the compiler and linker have to be bound to the actual memory locations

Relocation

- A compiled program should be independent of physical memory locations
- Programs have to be executable at any location in memory
  - Program instructions refer to memory addresses
    - Load data into register, or write register back to memory
- With dynamic schemes, programs can be relocated in memory
  - When processes are swapped in they may not be placed in the same memory area
- Addresses used in program cannot be actual physical addresses
  - makes program relocation impossible
  - Use relative addressing: addresses are offsets

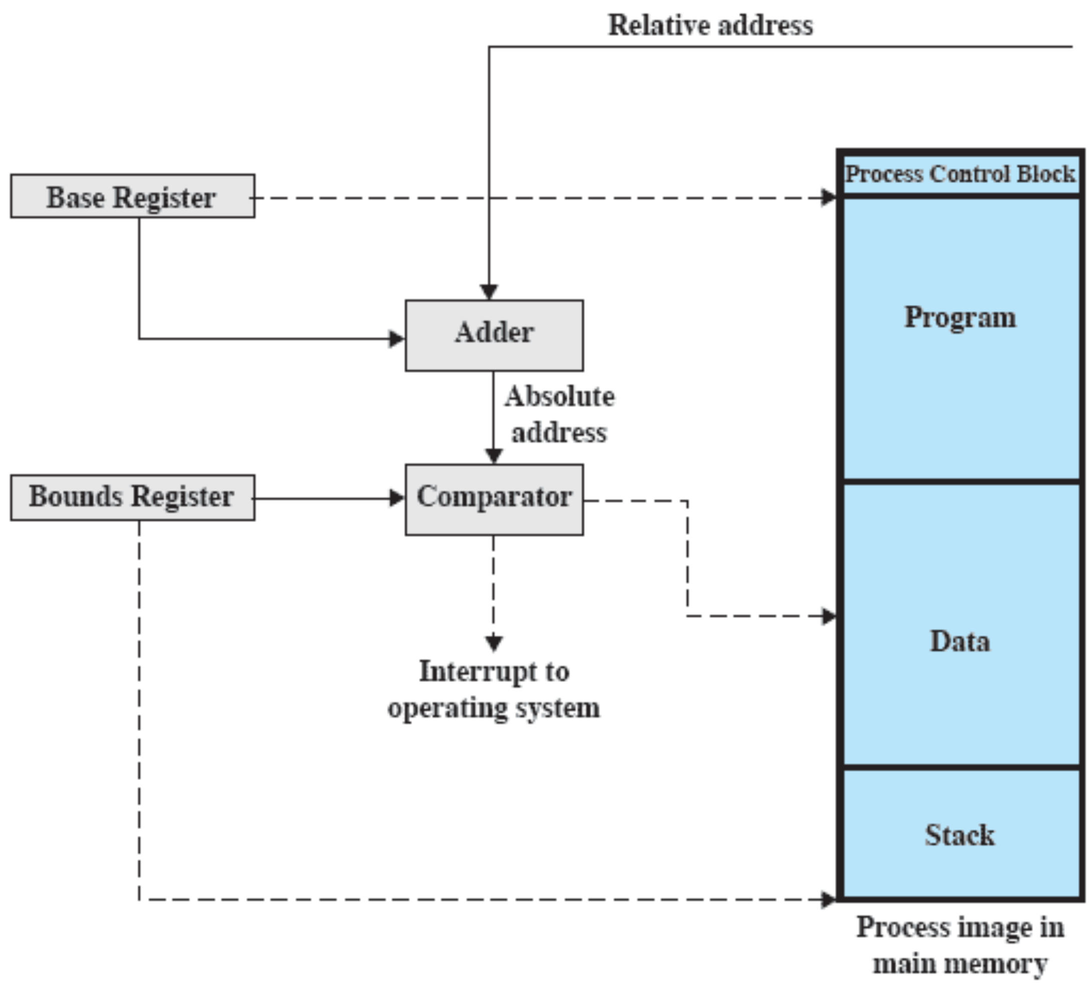
Relocation

- Solution:
  - Programs operate as if they all start at address 0
- Programs operate with relative addressing mechanism:
  - Reference to memory in program independent of actual physical location
  - Addresses in program are offsets from its base address (the starting address of the process image)
  - add the base address whenever there is an access to memory

Relocation with Relative Addressing

- Address translation with hardware
- Relative Addressing mechanism:
  - Logical addresses: addresses in program are offsets from its base address (the starting address of the process image)
  - Logical addresses are translated into actual physical memory addresses, whenever processes access memory locations via logical addresses
  - Address translation performed by hardware, e.g. the MMU
- Two registers
  - Base register: physical start address of the process image in memory
  - Bound register: physical end address of the process image in memory

Relocation with Relative Addressing



Addressing Memory

- Physical Addresses
  - Physical memory locations
- Logical Addresses:
  - Reference to memory in program, independent of actual physical location
  - Relative addressing schemes, addressing mechanisms in hardware
    - MMU translates logical addresses into physical addresses