University College Dublin
An Coláiste Ollscoile, Baile Átha Cliath

Operating Systems Memory Management

Dr. Vivek Nallur (vivek.nallur@ucd.ie)

Memory Management

Burst Cycles

- Process execution typically consists of a cycle of two states
  - CPU burst
    - A period of CPU execution with no I/O usage
  - I/O burst
    - wait for I/O signal

Memory Units

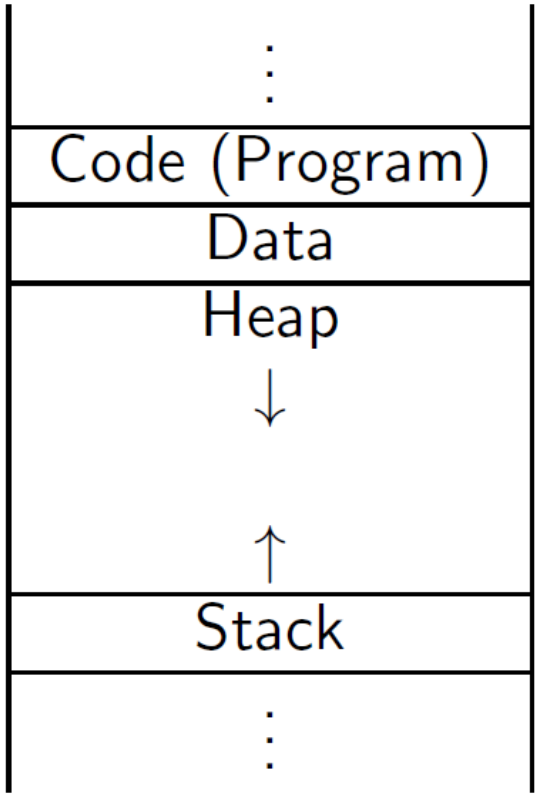
- A memory unit (MU) is any storage device that can be represented as a large array of words or bytes of data, each addressed by its own physical address
  - examples: random access memory (RAM), flash memory, disk
  - important: the MU is not concerned with the generation of the addresses presented to it

Memory Units

- Memory is central to the operation of a multiprogramming OS, since a process must be in memory to execute; typical instruction execution cycle:
- CPU loads a program instruction from memory according to content of the PC (program counter) register
- this instruction is decoded, and its execution may cause additional loading from/storing to memory
- eventually the process terminates, and its space in memory is declared available

Process in Memory

- What does a process in memory look like?
  - in Unix, memory allocated to a process is divided into areas called segments: code, data and stack segments



Address Binding

- A program usually resides on disk as a binary executable file
- to be executed it must be loaded, that is, brought into memory within the code section of the address space of the process
- in systems with virtual memory, a process (or parts of it) may be moved between memory and disk (swapped) during execution

Address Binding

- Most systems allow a user process to reside anywhere free in the memory unit
  - then the first physical address of the process needs not be 0
  - because of this an address binding mechanism is needed, so that the addresses used in the program relate to correct physical addresses

Address Binding Points (older)

1. Compile time (older systems)
   - if the memory location of the process is known beforehand: absolute code can be generated in the executable of a program by the compiler
   - in this case address references within the program are also physical addresses
   - however: need to recompile program (with new addresses) if process location changes

Address Binding Points (Old)

2. Load time (older systems)
   - if memory location not known at compile time: the compiler binds symbolic addresses to relocatable ones (i.e.: offsets)
   - physical addresses are bound at load time
   - no need to recompile program if process location changes, but it must be reloaded if it does

Address Binding Points (Modern)

3. Execution time (modern general-purpose OSs)
   - if a process can be moved to another memory area during execution, binding must done at run time (dynamic binding)
   - special hardware is necessary to translate process addresses into physical addresses: memory management unit (MMU)

Example: Address Binding at Execution Time

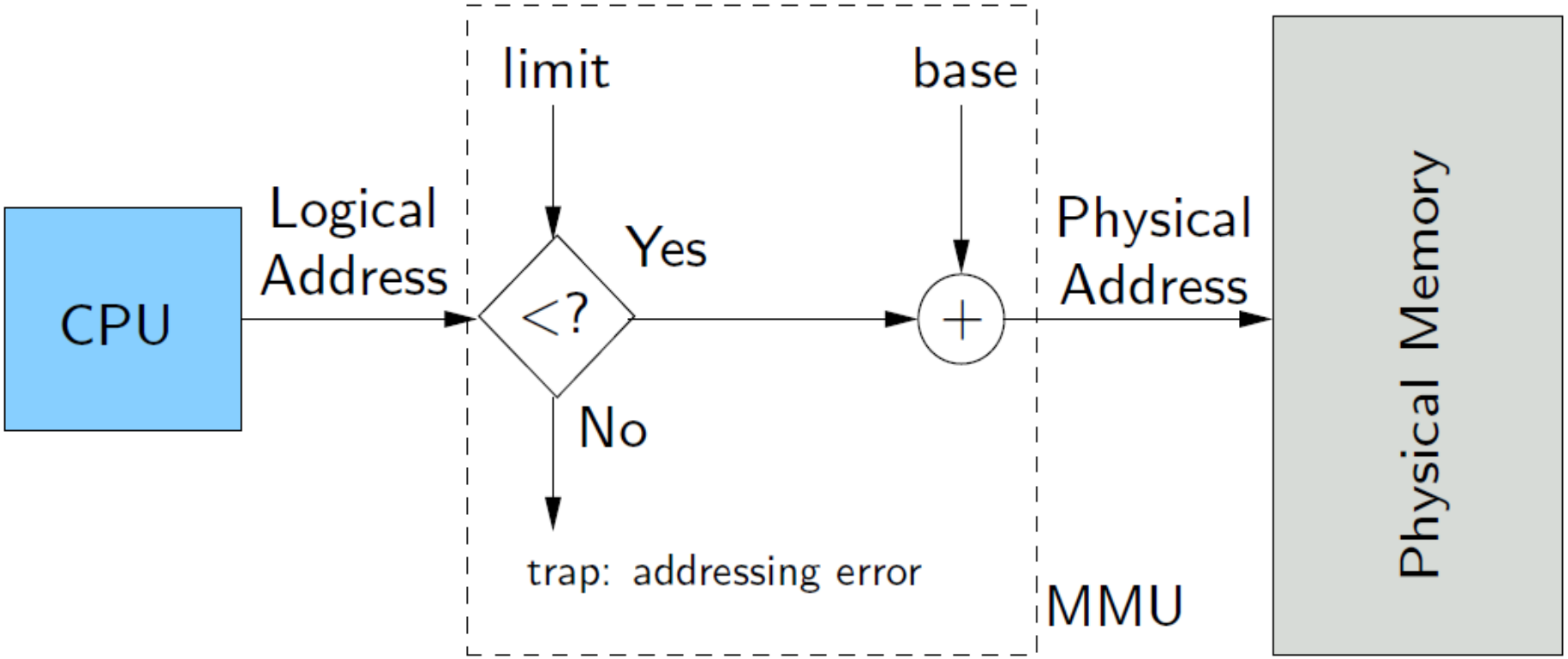
- Base and limit registers
  - base register: smallest physical address accessible
  - limit register: process address space size
- A memory address m generated by a running process is translated into a physical address m + base when accessing the memory unit
    - address is translated on the fly by MMU (hardware operation)
    - if m >= limit an addressing error trap is triggered
    - base and limit registers can only be set by means of privileged instructions (kernel mode)

Address Binding: Logical vs Physical Address Space

- An address handled by the CPU is commonly referred to as a logical address
  - it corresponds to a position in the address space of a process
  - e.g. a 32-bit processor handles 32-bit long addresses, and then the address space of a process ranges between 0 and $2^{32} - 1$
- Logical and physical addresses:
  - are identical in compile-time and load-time address binding
  - are different in run-time address binding (modern systems)

Address Binding: Logical vs Physical Address Space

- Example: in a system using base & limit registers



Memory Management (MM)

- Memory management is clearly vital in computer systems
  - ideally, we would like to have infinitely large and infinitely fast nonvolatile memory
  - in real life computers tend to have a memory hierarchy
    - primary memory: relatively small amount of fast volatile
    - memory (RAM, cache), also called main memory
    - secondary memory: large amount of slower nonvolatile memory (disk, flash memory)
    - (tertiary memory, in some special systems) Arguably this concept of tertiary memory is being restarted by the ideas behind Cloud Computing

Memory Management (MM)

- Memory manager: OS component which

1. keeps track of which parts of memory are in use and which parts are not
2. allocates memory to processes when they need it and deallocate it when they are done
3. manages swapping between main memory and disk when the former cannot hold all the processes: hierarchy management

Memory Management Systems

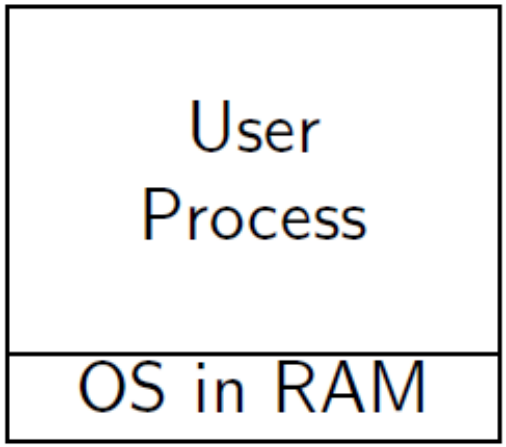
- MM systems can be divided into two classes:
  - Monoprogramming
  - Multiprogramming

Monoprogramming

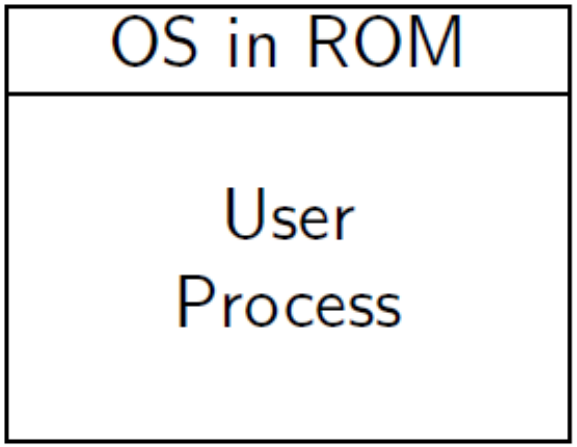
- very simple: only one user process in memory at a time (two processes, when including the OS process)
- MM still needs to decide where the process's data will reside
- no protection, compile-time or load-time binding, no swapping

Monoprogramming

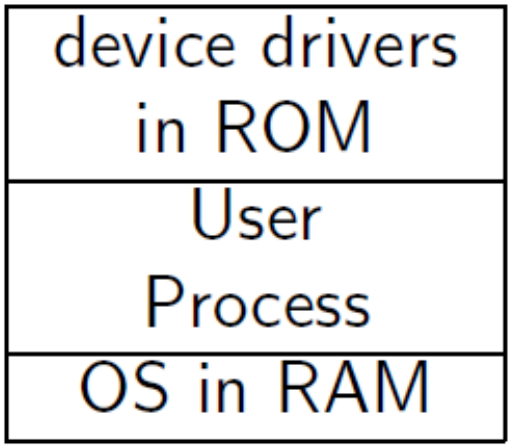
- examples of memory organisation in monoprogramming:



Old mainframes



Some old
embedded systems



Early PCs
(BIOS in ROM)

Multiprogramming

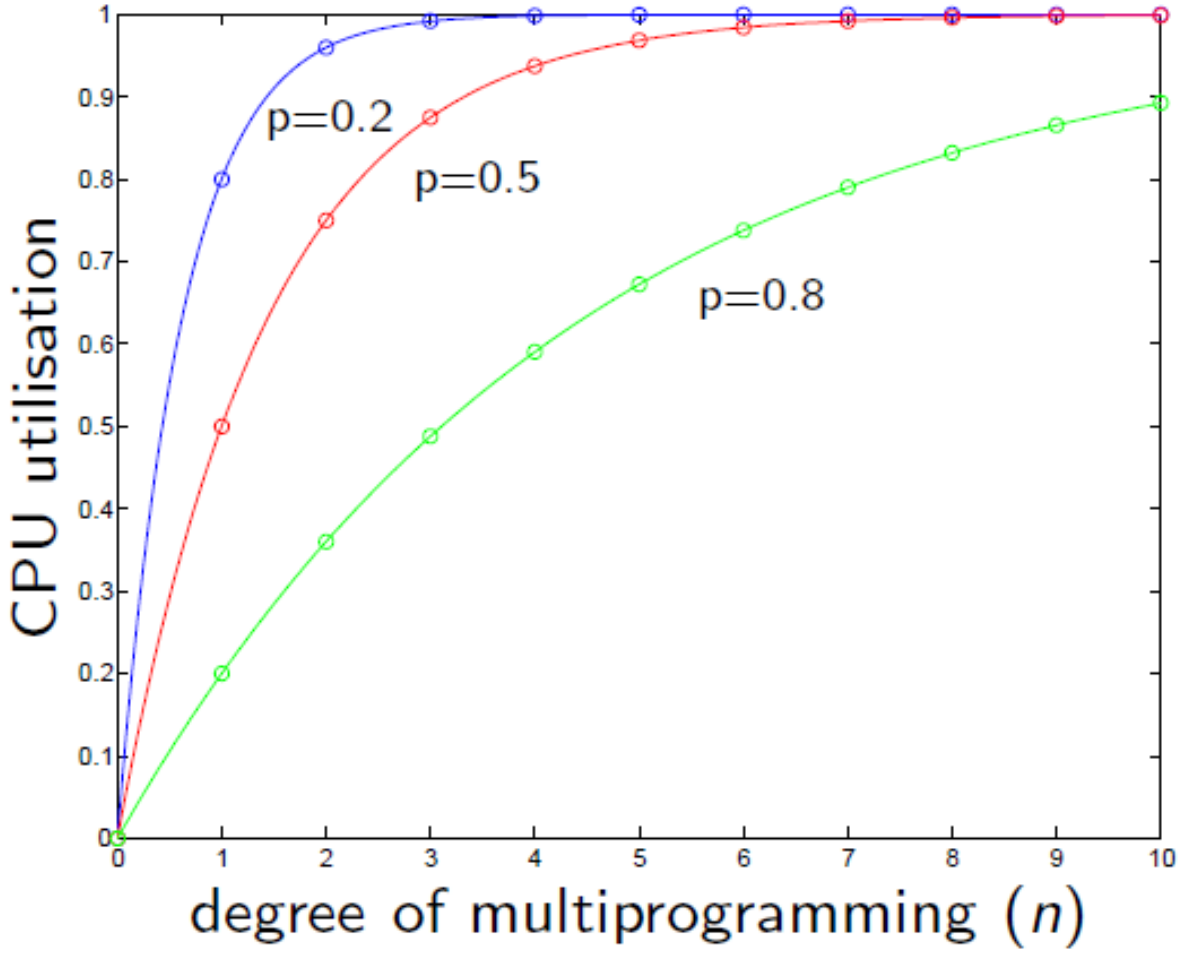
- Aims to provide interactive services to several users simultaneously
  - able to have more than one process in memory at once
  - protection, run-time binding, and swapping (using paging/segmentation, more on this later)
- goal: try to keep the processor(s) busy all the time
  - block processes when they are executing I/O operations
  - run processes which are not executing I/O operations
- multiprogramming increases CPU utilisation

Multiprogramming and Performance

- n processes in memory -> degree of multiprogramming = n
- Suppose that any process spends a fraction p of its time waiting for I/O to complete ($0 < p < 1$)
- The probability that all n are blocked waiting for I/O is p^n (i.e., probability that the CPU is idle therefore is

$$\text{CPU utilisation} = 1 - p^n$$

CPU Utilisation example



Example: Multiprogramming and Performance

- Although the previous model is simplistic (it assumes processes independent, same probability of I/O block, . . .), it allows to make specific performance predictions
- Assume a computer with 1 GB of memory of which the OS takes up 200 MB, leaving room for four 200 MB processes
  - assuming I/O wait time of 80% then we will achieve just under 60% CPU utilisation (ignoring OS execution overhead)
  - if we add another gigabyte of memory we can run five more processes; we can now achieve about 86% CPU utilisation (performance increase: 26%)
  - however, if we add another gigabyte of memory (14 processes & 1 OS) the CPU utilisation will only increase to about 96% (performance increase: just 10%)

Memory Organisation

- How should memory be organised in order to have more than one process in it (i.e. for multiprogramming)?
  - Contiguous allocation
  - Noncontiguous allocation

Memory Organisation

1. Contiguous allocation: section of consecutive physical memory addresses (partition) is allocated to a process
   - fixed partitions: divide the memory statically into a number of partitions, not necessarily equal
   - variable partitions: divide the memory dynamically

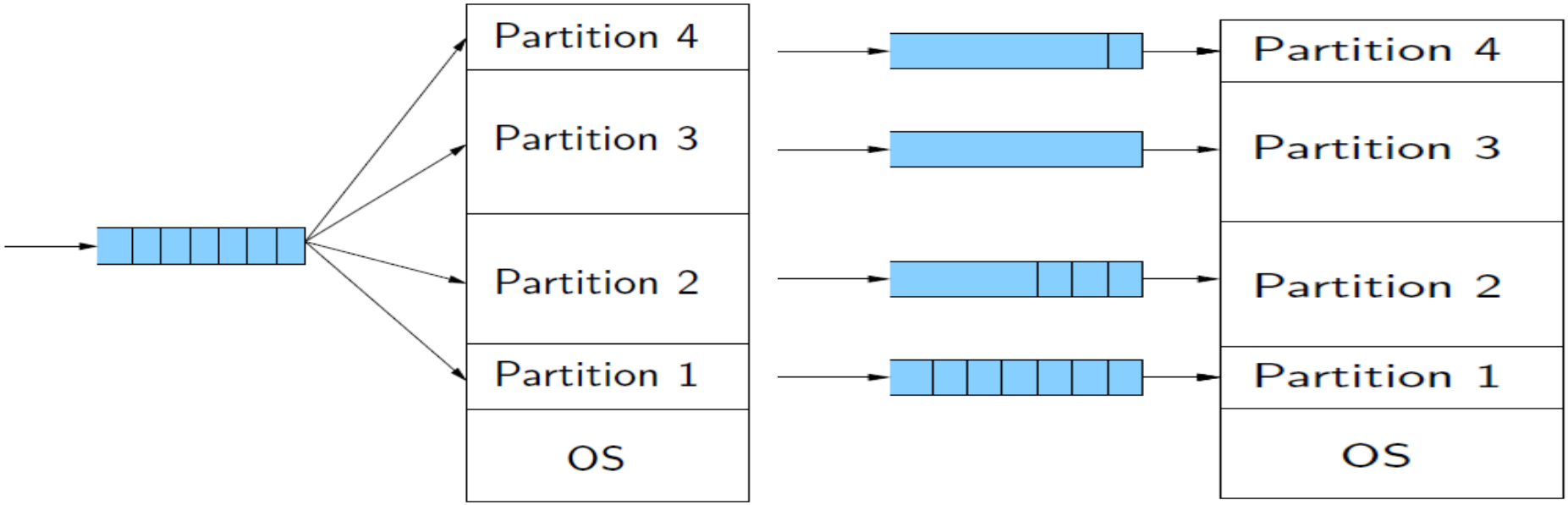
Memory Organisation

2. Noncontiguous allocation: memory allocated to a process is divided into more than one partition (block), which can be scattered across memory

- fixed blocks (paging): all blocks are of same fixed size
- variable blocks (segmentation): blocks are of variable size
    - memory addresses are still contiguous within a block
    - with virtual memory, blocks can be swapped out to disk

Fixed Partitions

- Single input queue
  - if a partition becomes free, allocate it to next process in queue that will fit in



Fixed Partitions

- Advantages
  - simple implementation and effective on batch systems
- Disadvantages
  - internal fragmentation problem: as the partition sizes are fixed, any space not used by a particular process is lost
  - multiple input queues can lead to dramatic loss of performance as large partition could be empty but small partitions could have full queues.
  - does not account for dynamic behaviour

Variable Partitions

- Advantages
  - Variable partitioning allows for more adaptability:
    - the system keeps a table indicating free parts of the memory
    - processes' sizes are taken into account: a process is granted as much contiguous space as it needs (if available)
    - number, location and size of the partitions vary dynamically depending on the processes running in the system
    - Needs an Algorithm to allocate free partitions
      - first fit: allocate the first big enough gap found
      - best fit: allocate the smallest big enough gap
      - worst fit: allocate the largest gap

Variable Partitions

- Disadvantages
  - External fragmentation problem
    - completing processes may leave many memory gaps
    - there may not be enough free contiguous memory for a new process, despite the total free memory in fragments being large
    - this problem can be solved by compaction

Paging Technique

- Basic strategy in the paging technique:
  - the physical memory is partitioned into small equal fixed-size chunks (called frames)
  - the logical memory (i.e. the address space of a process) is also divided into chunks of same size as the frames (called pages)
- To execute a process, its pages are loaded from disk into available memory frames relying on a page table
  - frames associated to a process can be noncontiguous

Paging Technique

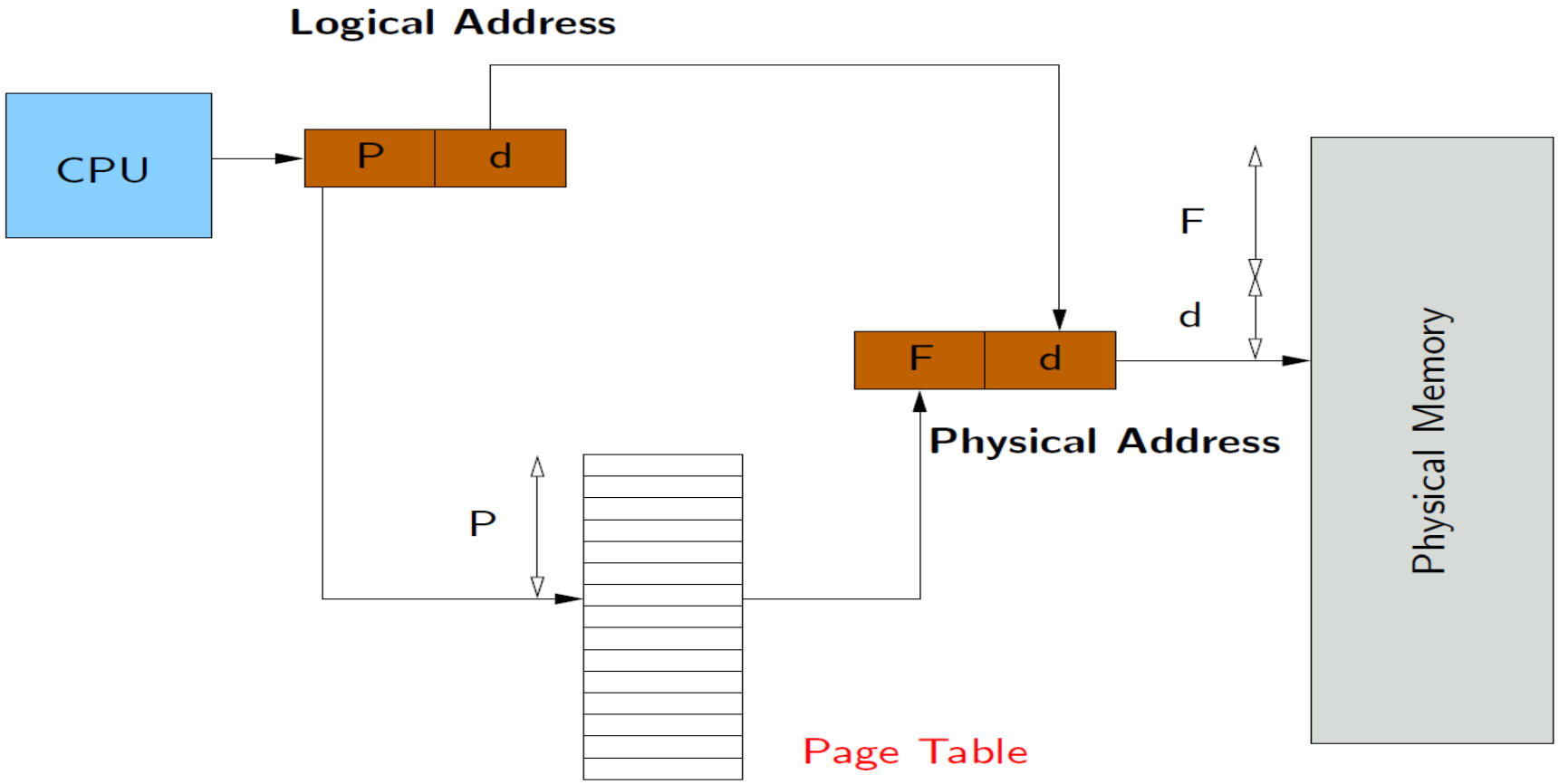
- A program can be loaded if there are enough free frames, and there is no need for fitting algorithms
- Also we can just load the few pages relevant for execution (virtual memory: more about it in next lecture)

Paging Technique Features

- Fragmentation
  - internal: only a fraction of the last page of a process
  - external: none (no need for compaction)
- Every logical address (ie CPU or process addresses) is divided into two parts
  - a page number and an offset within the page
- The page size is typically 2^n (e.g. 512 bytes – 16 MB); if the size of the logical address space is 2^m then
  - the m − n high-order bits give the page number
  - the n low-order bits give the page offset
- A logical address is translated to a physical address using the processor hardware

Paging Hardware

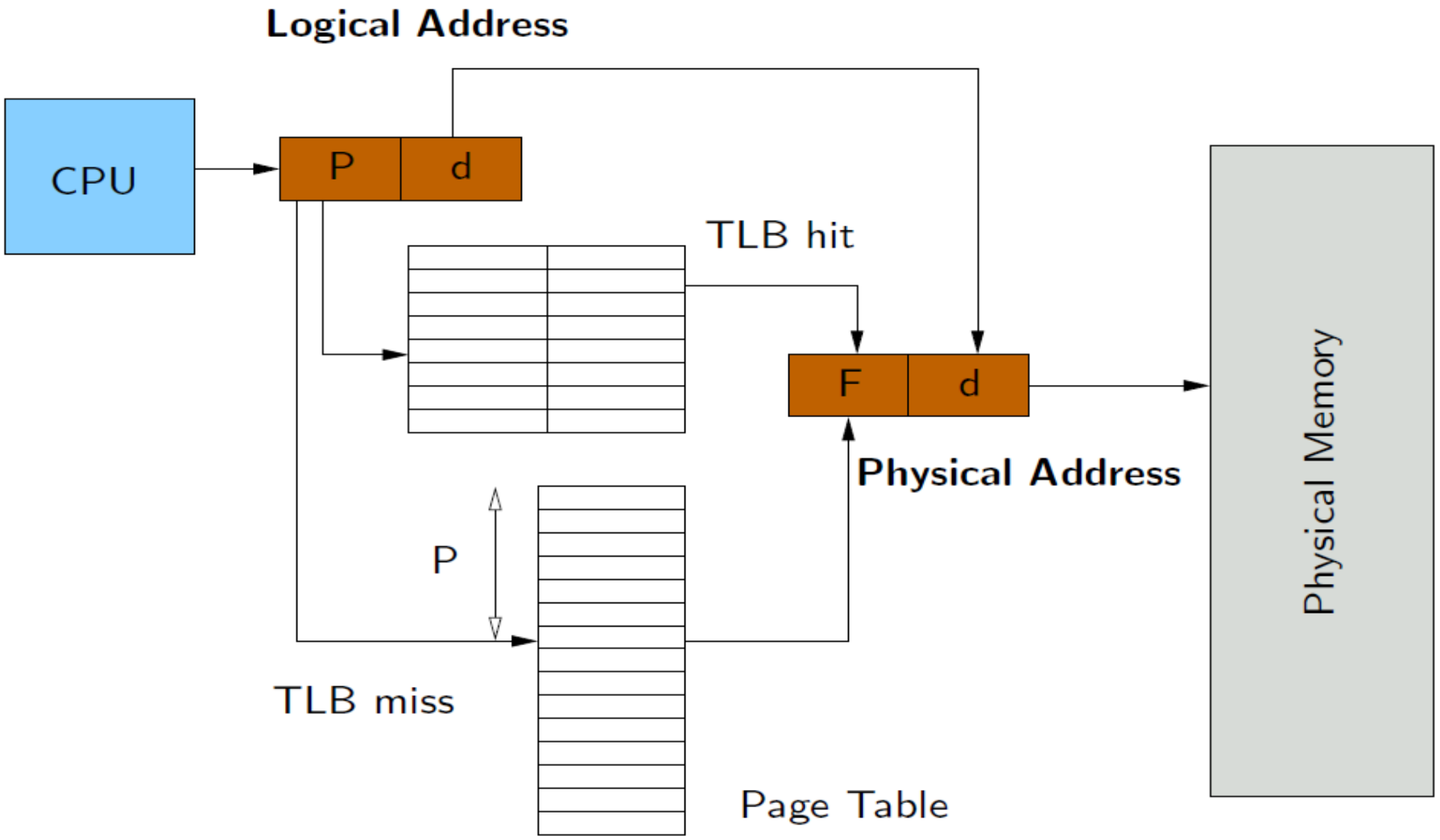
- If page table small (256 entries), it can be implemented with dedicated registers



Paging Hardware Optimisation

- Page table usually big (i.e. 106 entries): kept in main memory
- But this is slow with respect to using registers: translation look-aside buffer (TLB) used to improve performance
  - a TLB is an associative high-speed memory, which associates a key (tag) with a value
  - when presented with a key, it compares it with all keys simultaneously
  - it needs a replacement policy (what happens when it is full?)
  - TLB size: 64-1024 values

Paging Hardware Optimisation



Paging Hardware Optimisation

- Performance analysis of a system using paging with a TLB can be done using the hit ratio: fraction of time that a page number is found in the TLB
  - effective memory access time (T_e)

$$T_e = p \times (T_t + T_a) + (1 - p) \times (T_t + 2T_a)$$

p: probability of a hit (e.g. p = 0.8 if hit ratio is 80%)

T_t : TLB access time

T_a : memory access time

  - Time savings can be up to 90%, but a TLB is expensive

Segmentation Technique

- Basic strategy: the logical memory is divided into a number of segments, each of possibly different length
  - logical address consists now of a segment number and an offset within the segment
  - more complex relationship between logical and physical address

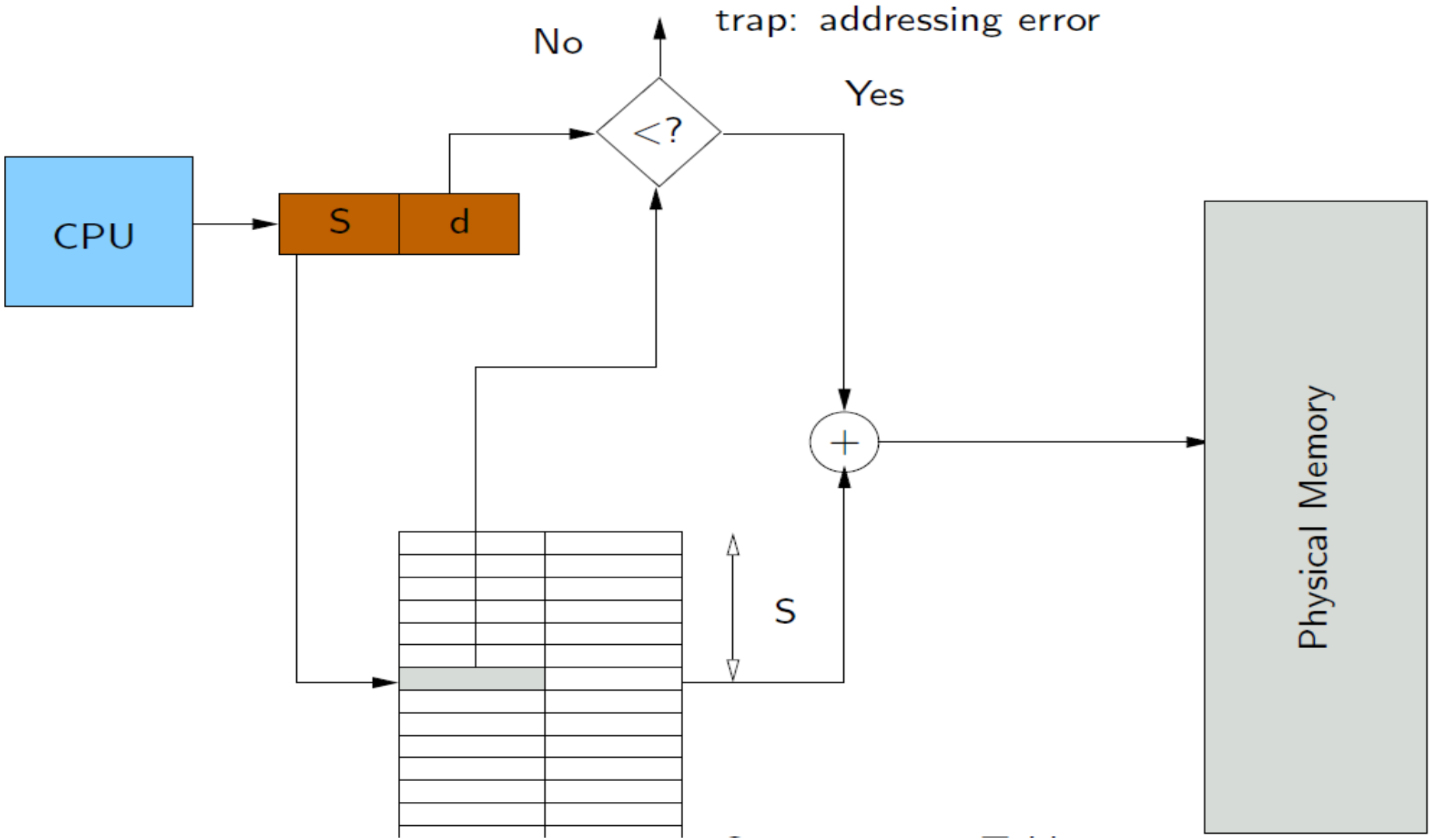
Segmentation Technique

- Unlike paging, segmentation is visible to the programmer
  - typically, the programmer assigns programs and data to different segments: main program, stack, subroutines, data. . .
  - the programmer must be aware of maximum segment limit
- Entries in the segmentation table include the base and limit registers for a segment
  - association of protection with the segments

Segmentation Technique

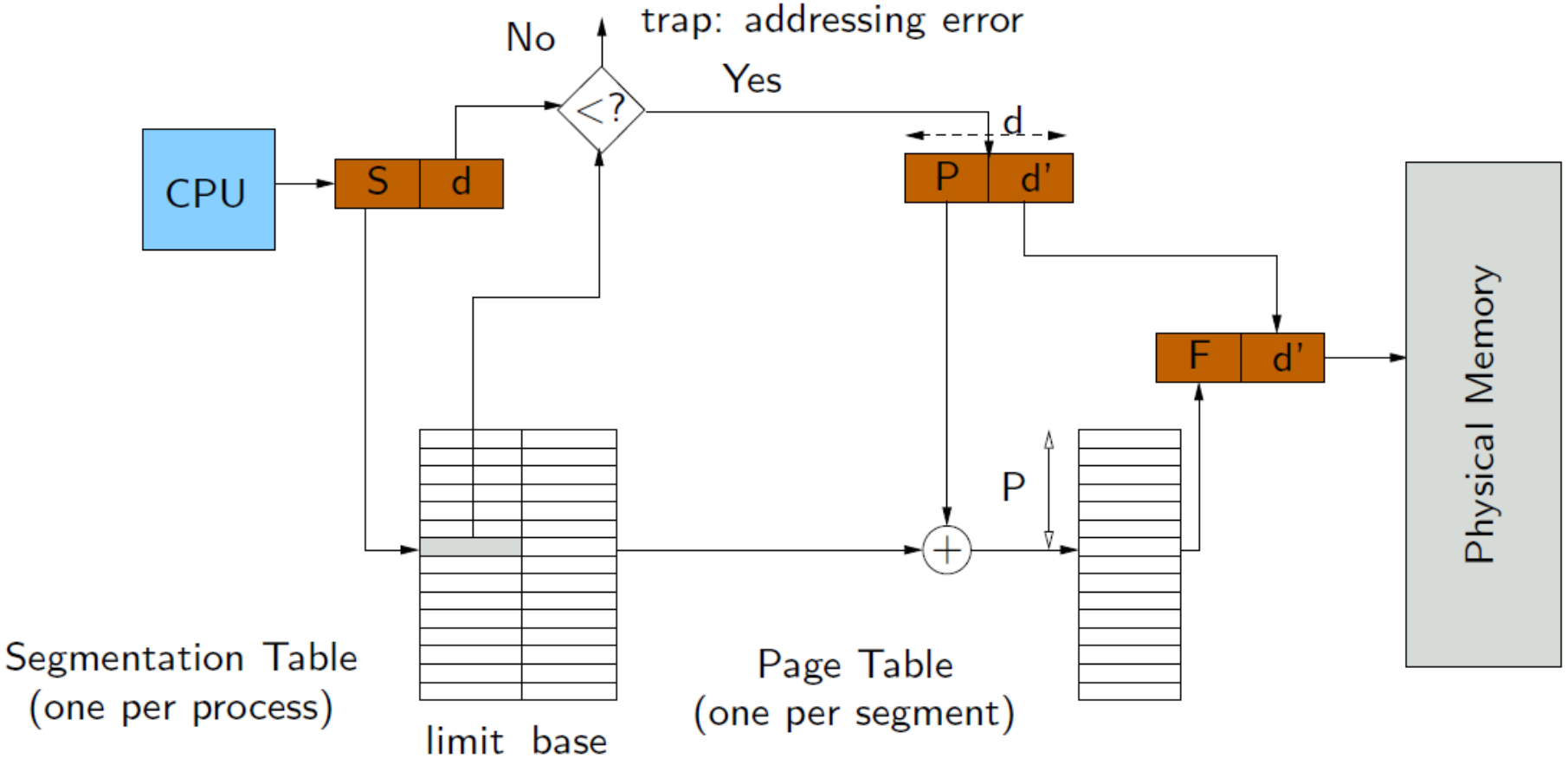
- Fragmentation:
  - internal: none
  - external: not solved, but less severe than variable partitioning because of the smaller pieces a process is divided into

Segmentation Hardware



Paged Segmentation

- A solution to external fragmentation in segmentation is to combine segmentation with paging (e.g. Intel 80x86 CPUs)



Paged Segmentation

- Advantages:
  - it reduces external fragmentation
  - multiple address spaces available, instructions can have smaller
  - address fields (different segments can have different address
  - space sizes)
  - with virtual memory (part of the process swappable to disk):
    - distinction between access violations and page faults (page not in memory)
    - swapping can occur incrementally
- Disadvantages:
  - more complex
  - a page table is needed per segment

Next Week

- Lecture
  - Topic: Virtual Memory
- Study Time
  - Review Chapter 8