Anthony Ventresque
anthony.ventresque@ucd.ie

Memory Management

School of Computer Science, UCD

Scoil na Ríomheolaíochta, UCD

Outline

- To understand memory units and swapping
- To understand address binding, translation, and physical and logical address spaces
- To understand basic memory-management techniques
- To understand memory organisation
- To understand the memory techniques and address translation of:
  - Fixed partitions
  - Variable partitions
  - Paging
  - Segmentation

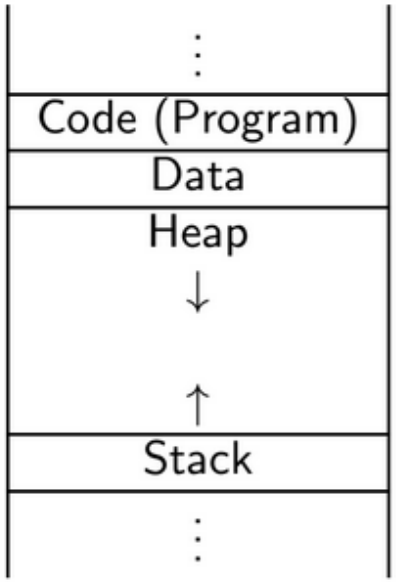
Memory Units

- 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, Hard Disk

- 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

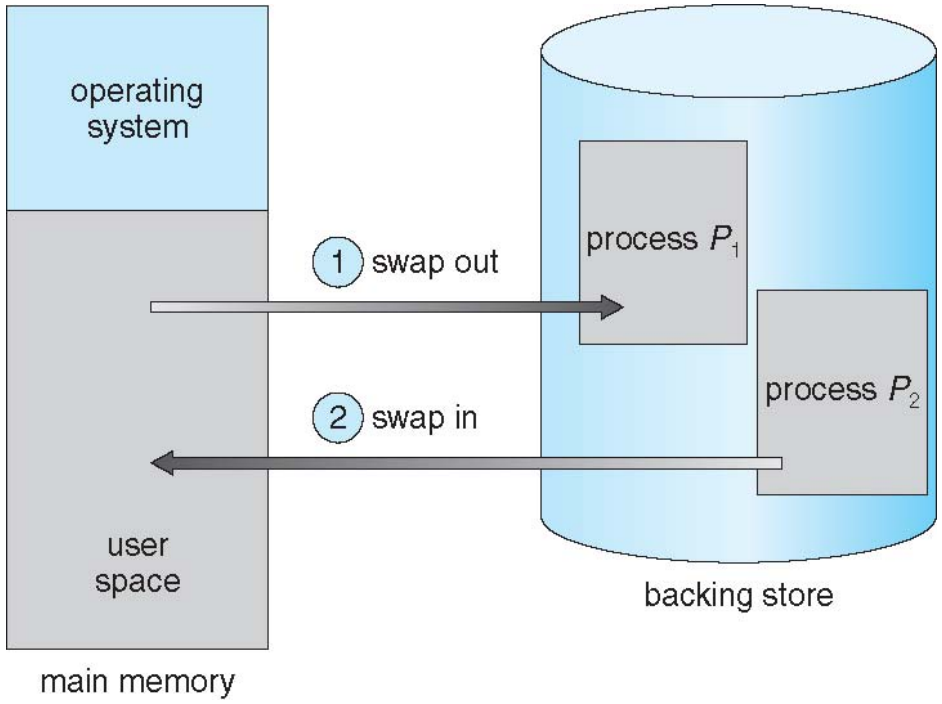
- What does a process in memory look like?
  - In Unix, memory allocated to a process is divided into areas called segments: code, data and stack segments



  - Segments facilitate the isolation of read-only data from read-write data

Loading and Swapping

- 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
  - Thus, the first physical address of the process may not be 0
  - Because of this an ***address binding mechanism*** is needed, so that the addresses used in the program relates to correct physical addresses

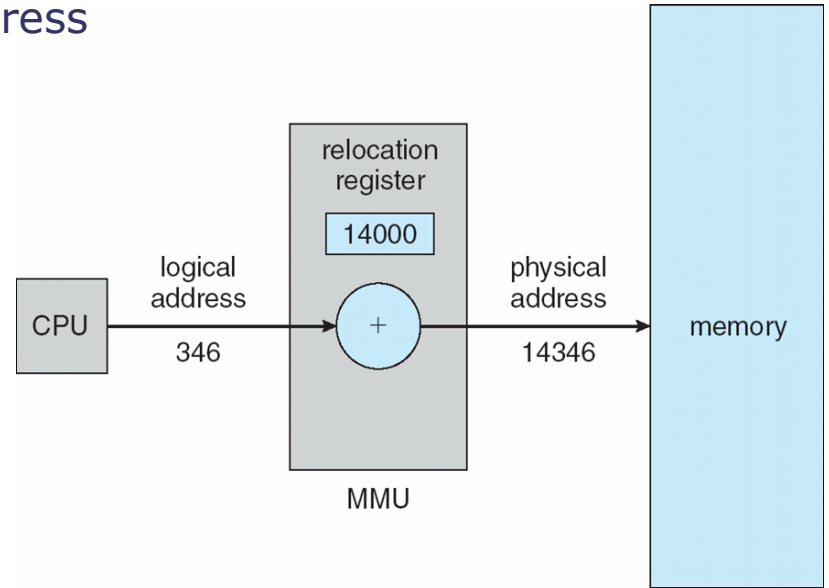
Address Binding Points

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
  - In this case address references within the program are also physical addresses
- ***Load time***
  - Must generate ***relocatable code*** if memory location is 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
- ***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) Memory Management Unit (MMU)

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
- Memory-Management Unit (MMU)
  - Hardware device that at run time maps virtual to physical address



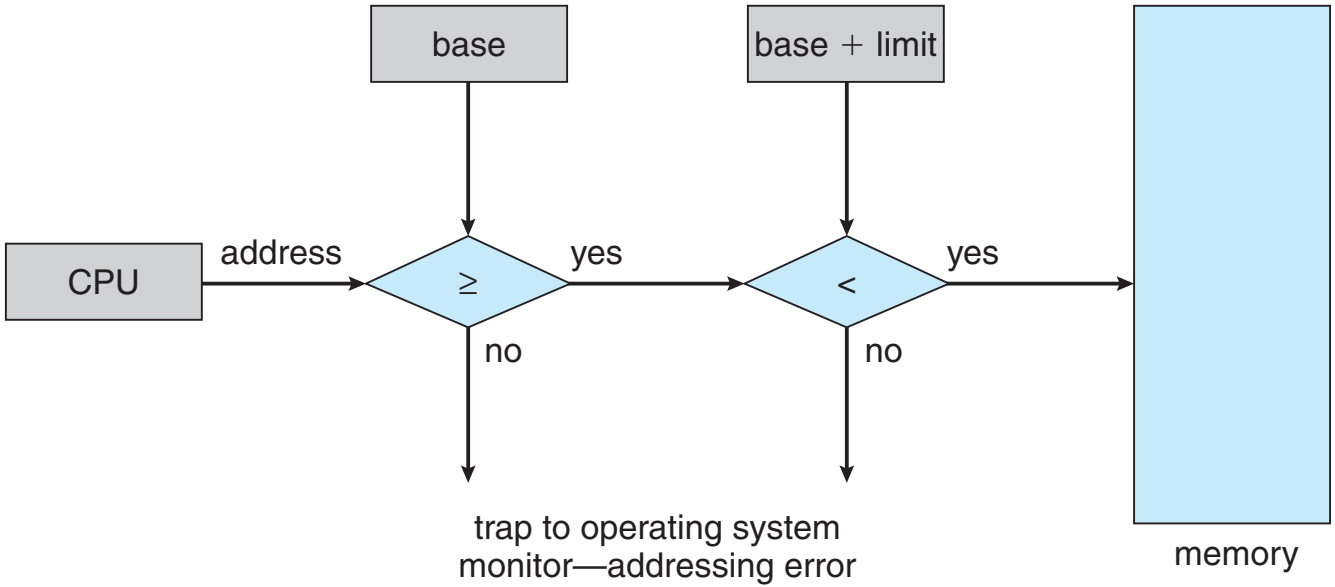
Example: Address Binding at Execution Time

- Base and limit registers
  - ***base register*****:** smallest physical address accessible
  - ***limit register*****:** process address space size

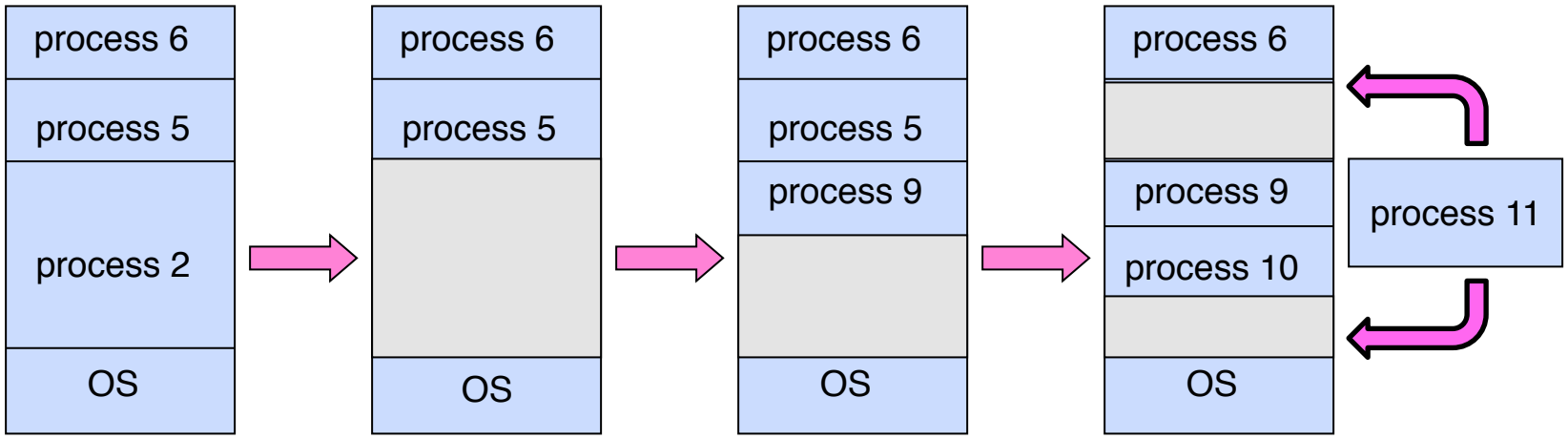
- A memory address *m* generated by a running process is translated into a physical address *m* + *base* when accessing the memory unit
  - Address is translated on the fly by MMU (***hardware operation***)
  - If *m* >= *limit* an addressing error trap is triggered
  - Base and limit registers can only 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)
- **Example: in a system using base & limit registers**



Issues with Basic Mapping Method

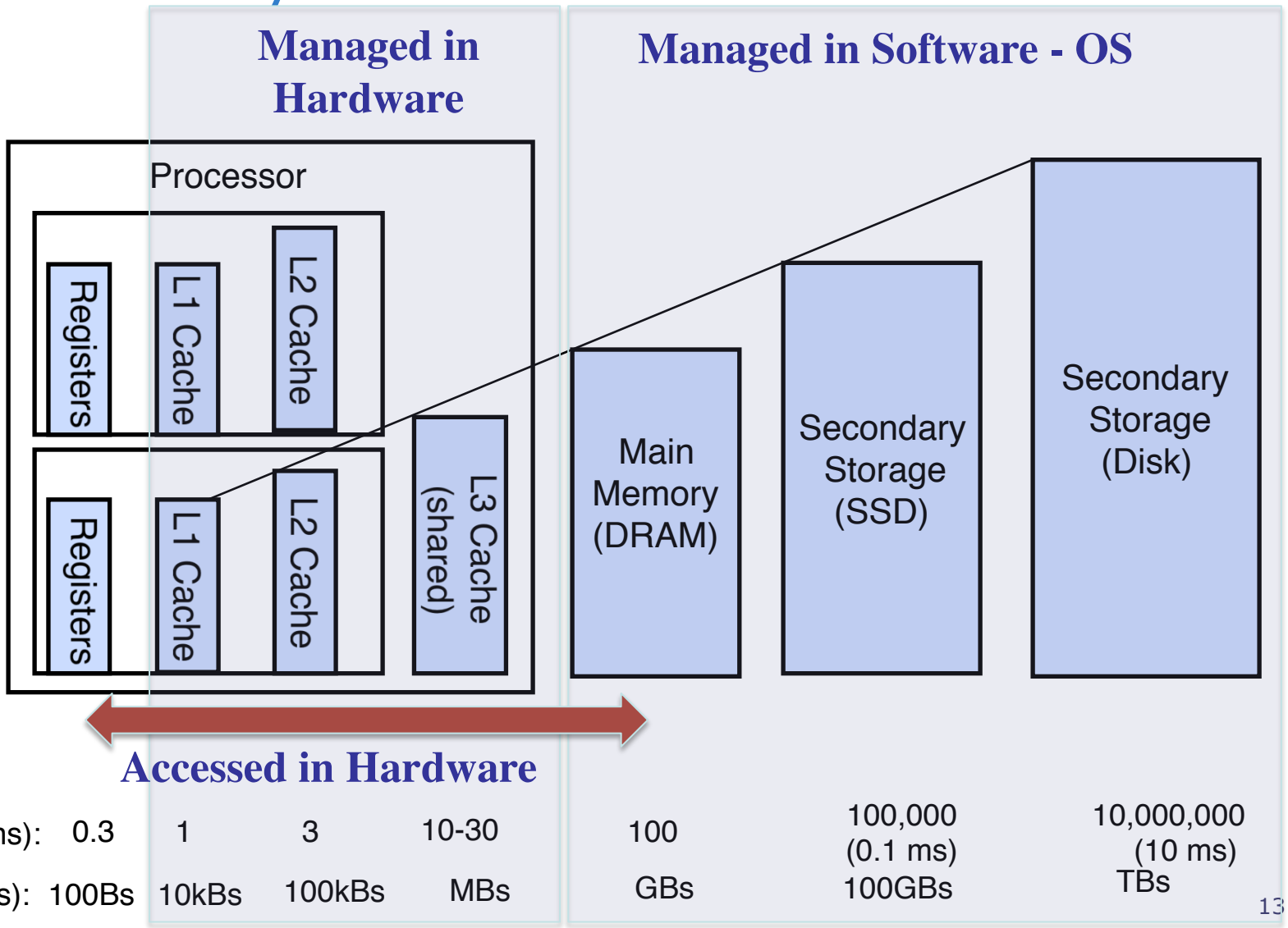


- Fragmentation problem
  - Not every process is the same size
  - Over time, memory space becomes fragmented
- Missing support for sparse address space
  - Would like to have multiple chunks/program
  - E.g.: Code, Data, Stack
- Hard to do inter-process sharing
  - Want to share code segments when possible
  - Want to share memory between processes
  - Helped by providing multiple segments per process

Memory Management (MM)

- Memory management is clearly vital in computer systems
  - Ideally, we would like to have infinitely large and infinitely fast non-volatile 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 non-volatile memory (disk, flash memory)
    - (tertiary memory, in some special systems)

Management & Access to the Memory hierarchy



MM Systems

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

MM systems can be divided into two classes:

1. *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

MM Systems

2. Multiprogramming

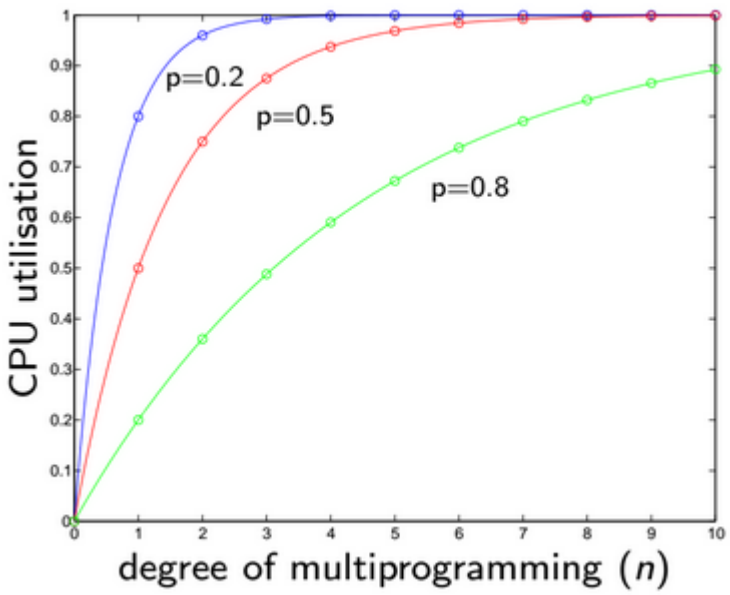
- Provide interactive service to several users simultaneously
- Able to have more than one process in memory at once
  - protection, run-time binding, and swapping (using paging/segmentation, more on this later)
- **Goal:** try to keep the processor(s) busy all the time
  - Block processes when they are executing I/O operations
  - Run processes which are not executing I/O operations
- **Multiprogramming increases CPU utilisation**

Important Aspects of Memory Multiplexing

- ***Controlled overlap***
  - Separate state of threads should not collide in physical memory.
  - Conversely, would like the ability to overlap when desired (for communication)
- ***Translation:***
  - Ability to translate accesses from one address space (virtual) to a different one (physical)
  - When translation exists, processor uses virtual addresses, physical memory uses physical addresses
  - Side effects:
    - Can be used to avoid overlap
    - Can be used to give uniform view of memory to programs
- ***Protection*:**
  - Prevent access to private memory of other processes
    - Different pages of memory can be given special behavior (Read Only, Invisible to user programs, etc).
    - Kernel data protected from User programs
    - Programs protected from themselves

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 (i.e., probability that the CPU is idle)
  - therefore, $\boxed{\text{CPU utilisation} = 1 - p^n}$



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)?

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
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
   - notes:
     - Memory addresses are still contiguous within a block
     - with ***virtual memory***, blocks can be swapped out to disk

Fixed Partitions

- **Multiple input queues**
  - new process placed in queue for smallest partition that can accommodate it
- **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
  - M=Multiple input queues can lead to dramatic loss of performance
    - Consider the case when large partitions are empty and the queue for small processes is full
  - Does not account for dynamic behaviour:
    - Not suitable for time-sharing
    - The possibly of processes changing size is not considered
    - It may not be easy to know how big the partition needed by a process is before execution

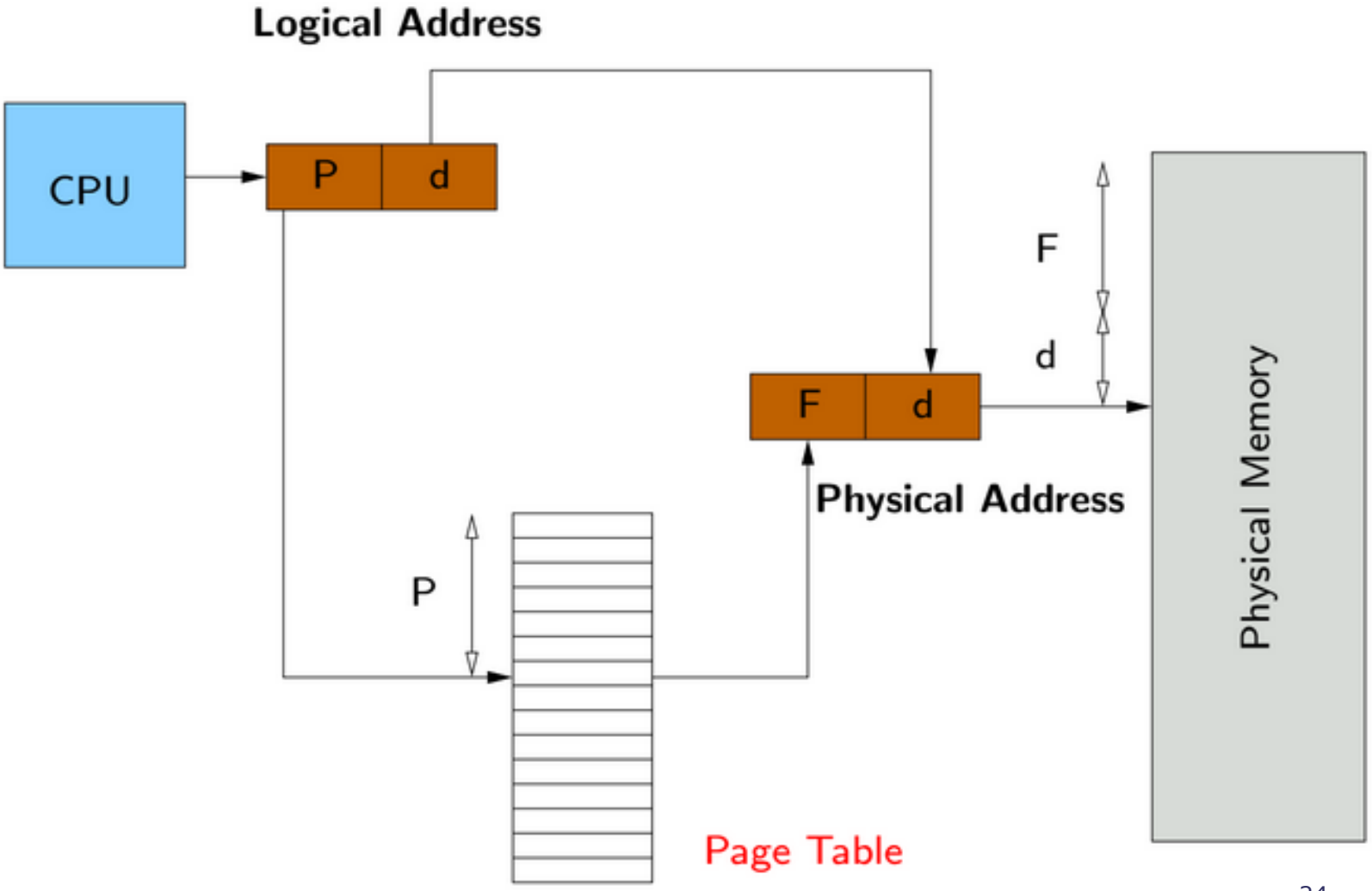
Variable Partitions

- 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
- An algorithm to allocate free partitions is required:
  - **First fit:** allocate the first big enough gap found
  - **Best fit:** allocate the smallest big enough gap
  - **Worst fit:** allocate the largest gap
- ***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
- **Advantages:**
  - A program can be loaded if there are enough free frames, and there is no need for fitting algorithms
  - Moreover, we could just load the few pages relevant for execution (virtual memory: more about it in next lecture)

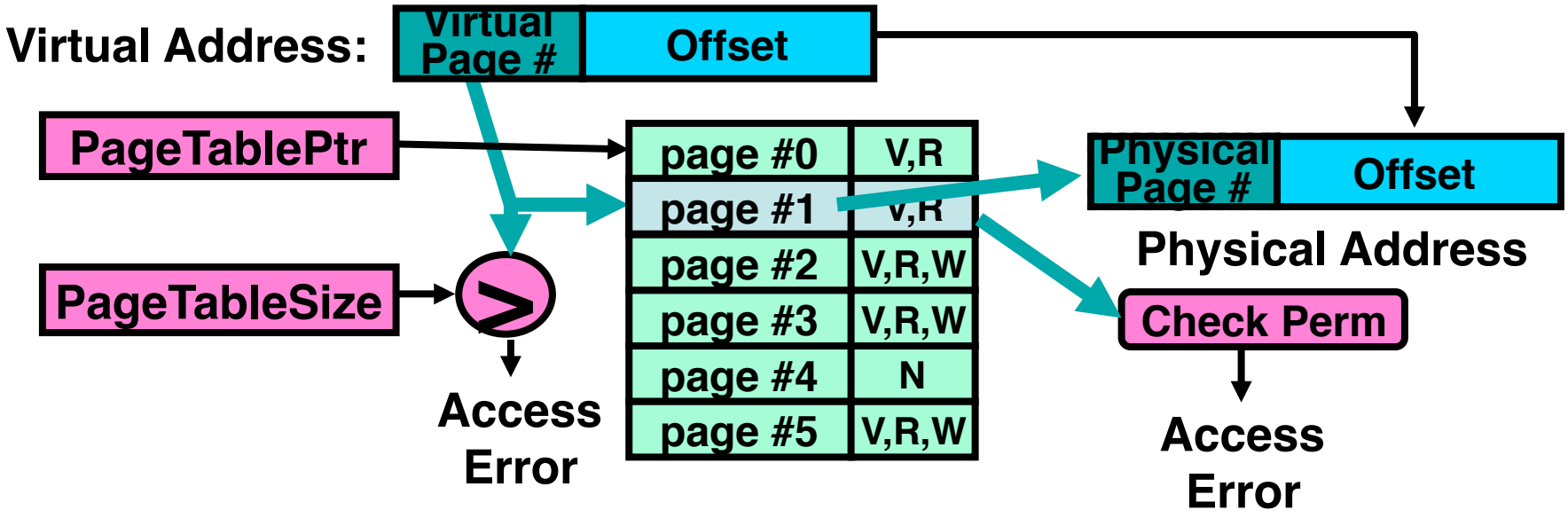
Paging Hardware



Paging Technique Features

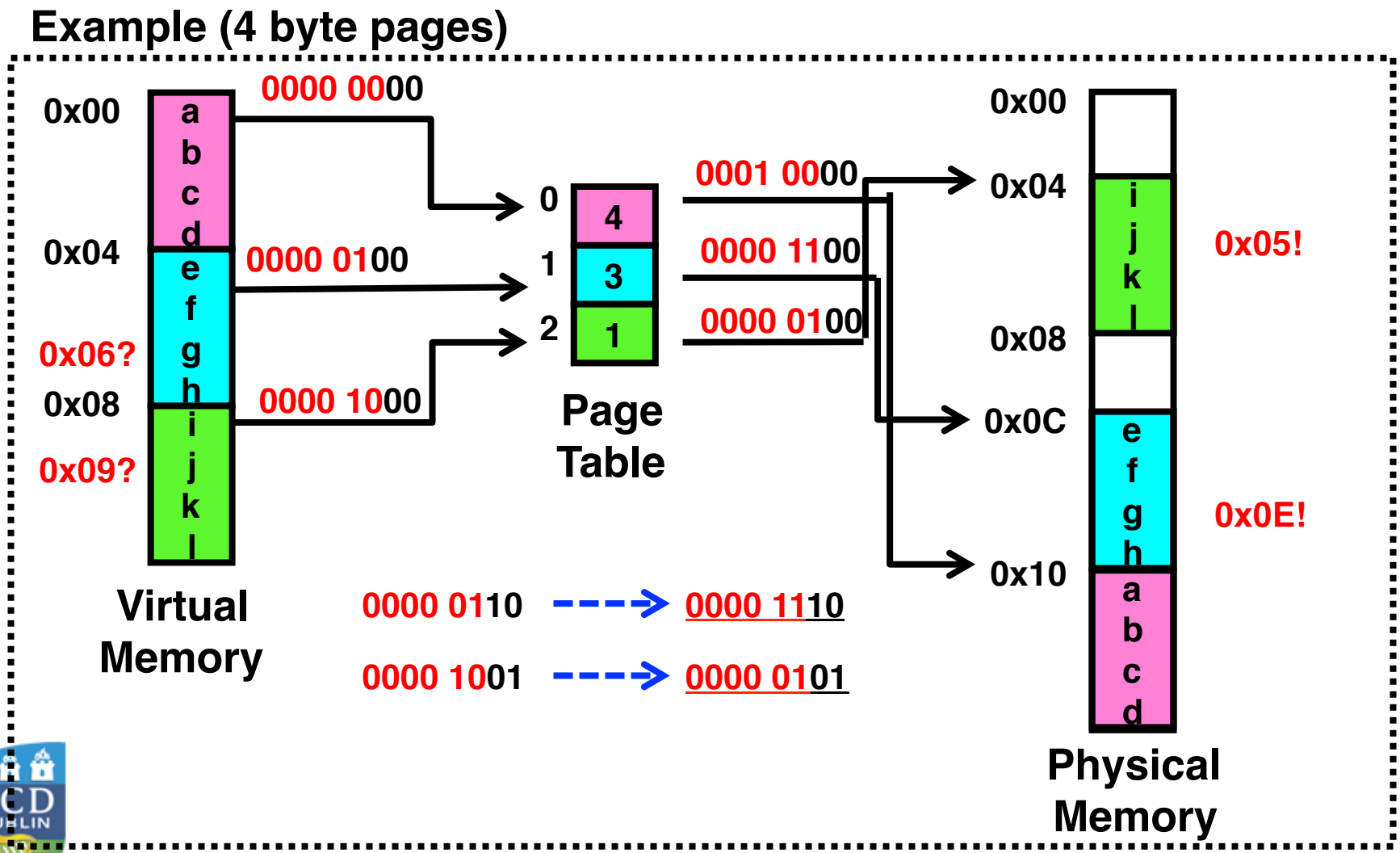
- Fragmentation
  - **Internal:** only a fraction of the last page of a process
  - **External:** none (no need for compaction)
- Every logical address (i.e. CPU or process addresses) is divided into two parts
  - A ***page number*** and an ***offset*** within the page
- The page size is typically 2^n (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

How to Implement Paging?

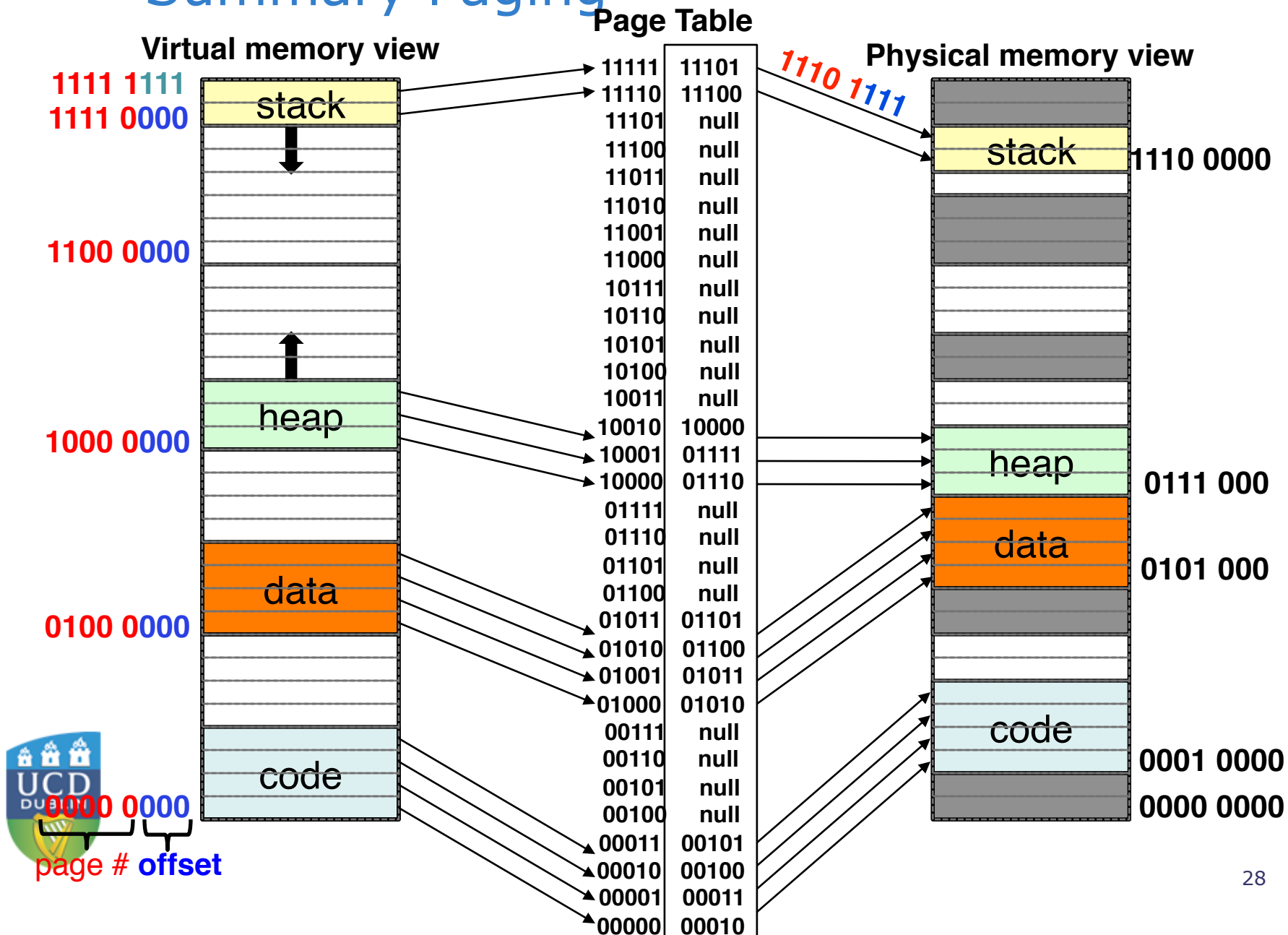


- Page Table (One per process)
  - Resides in physical memory
  - Contains physical page and permission for each virtual page
    - Permissions include: Valid bits, Read, Write, etc
- Virtual address mapping
  - Offset from Virtual address copied to Physical Address
    - Example: 10 bit offset ⇒ 1024-byte pages
  - Virtual page # is all remaining bits
    - Example for 32-bits: 32-10 = 22 bits, i.e. 4 million entries
    - Physical page # copied from table into physical address
  - Check Page Table bounds and permissions

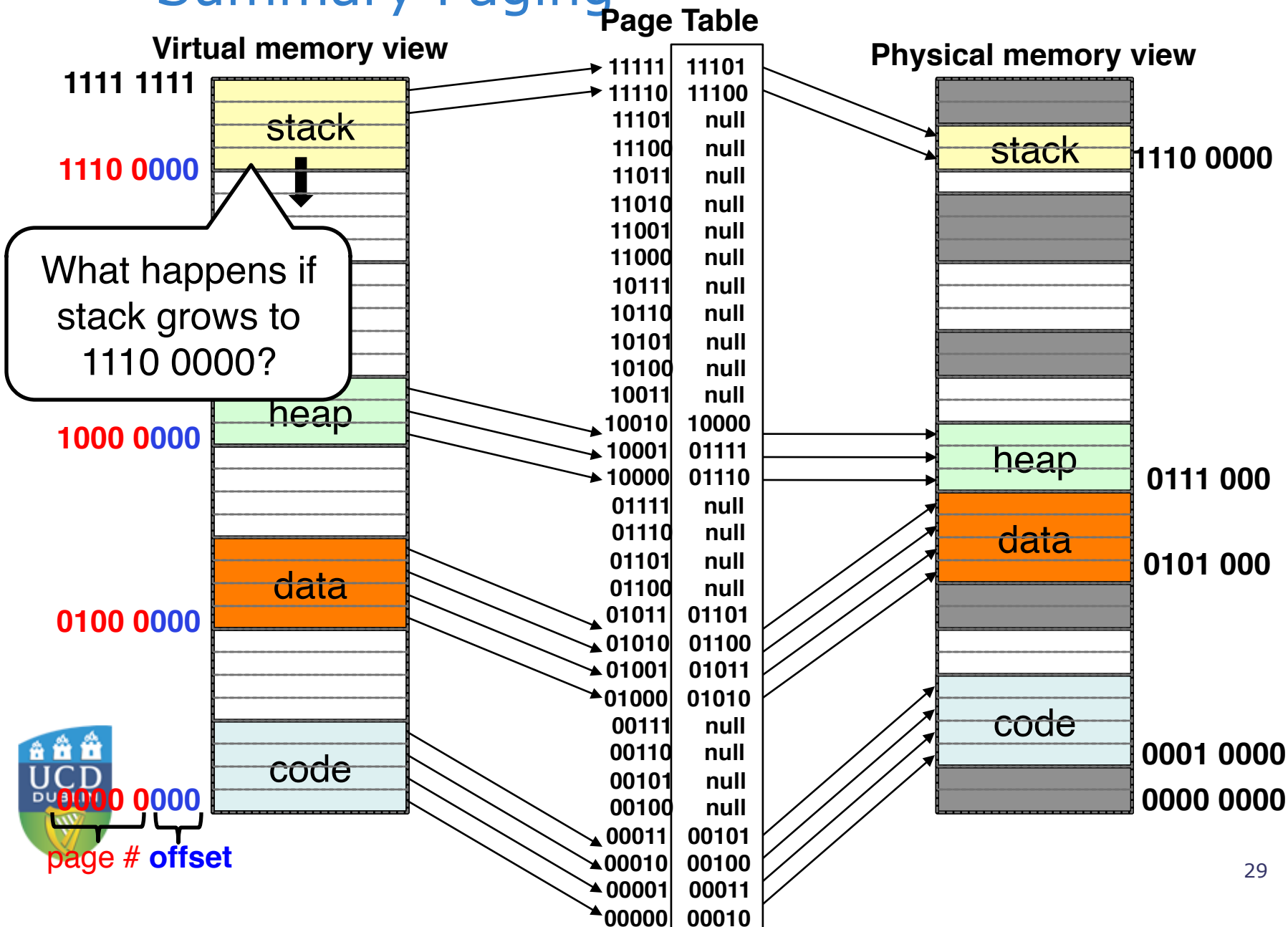
Simple Page Table Example



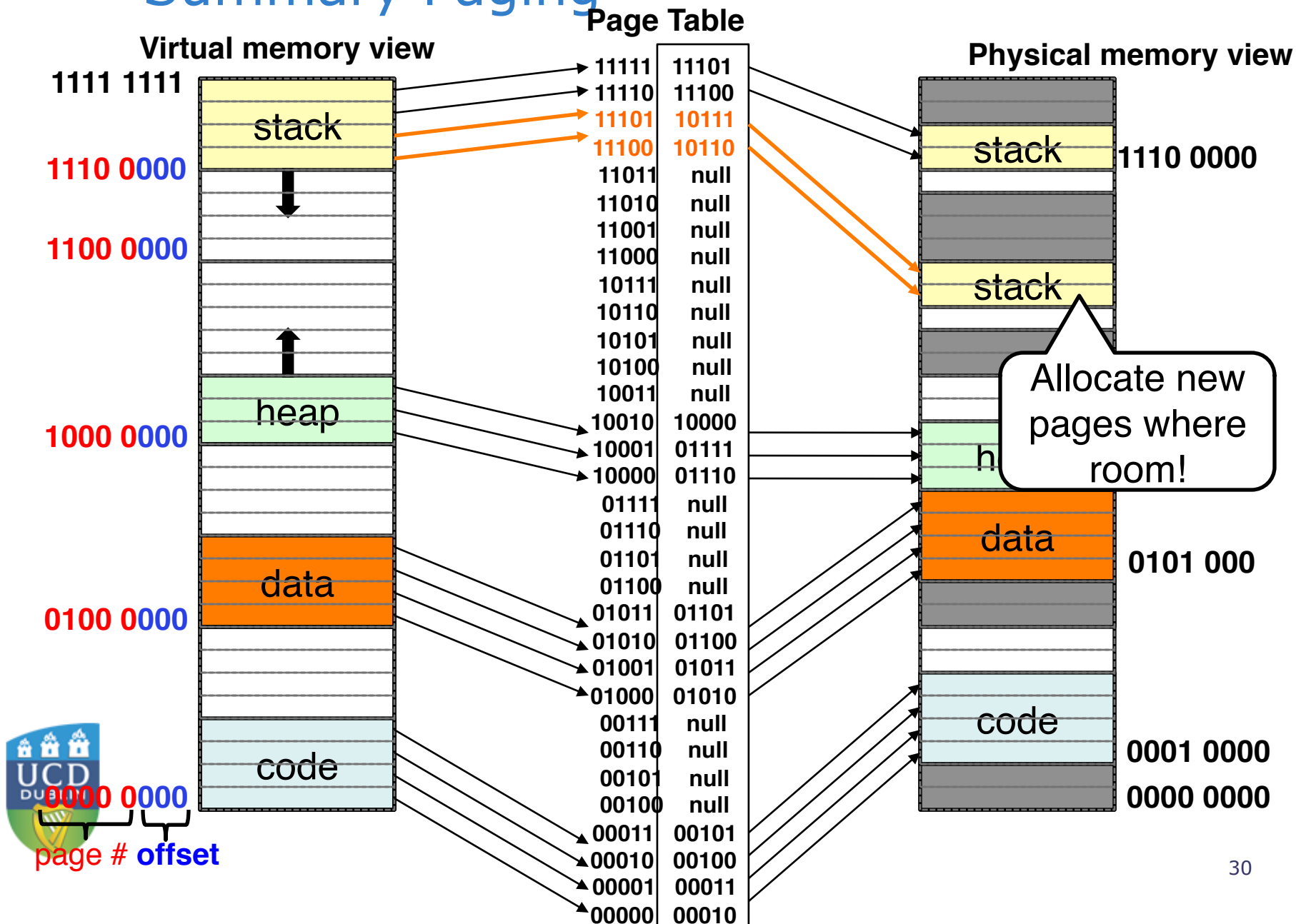
Summary Paging



Summary Paging



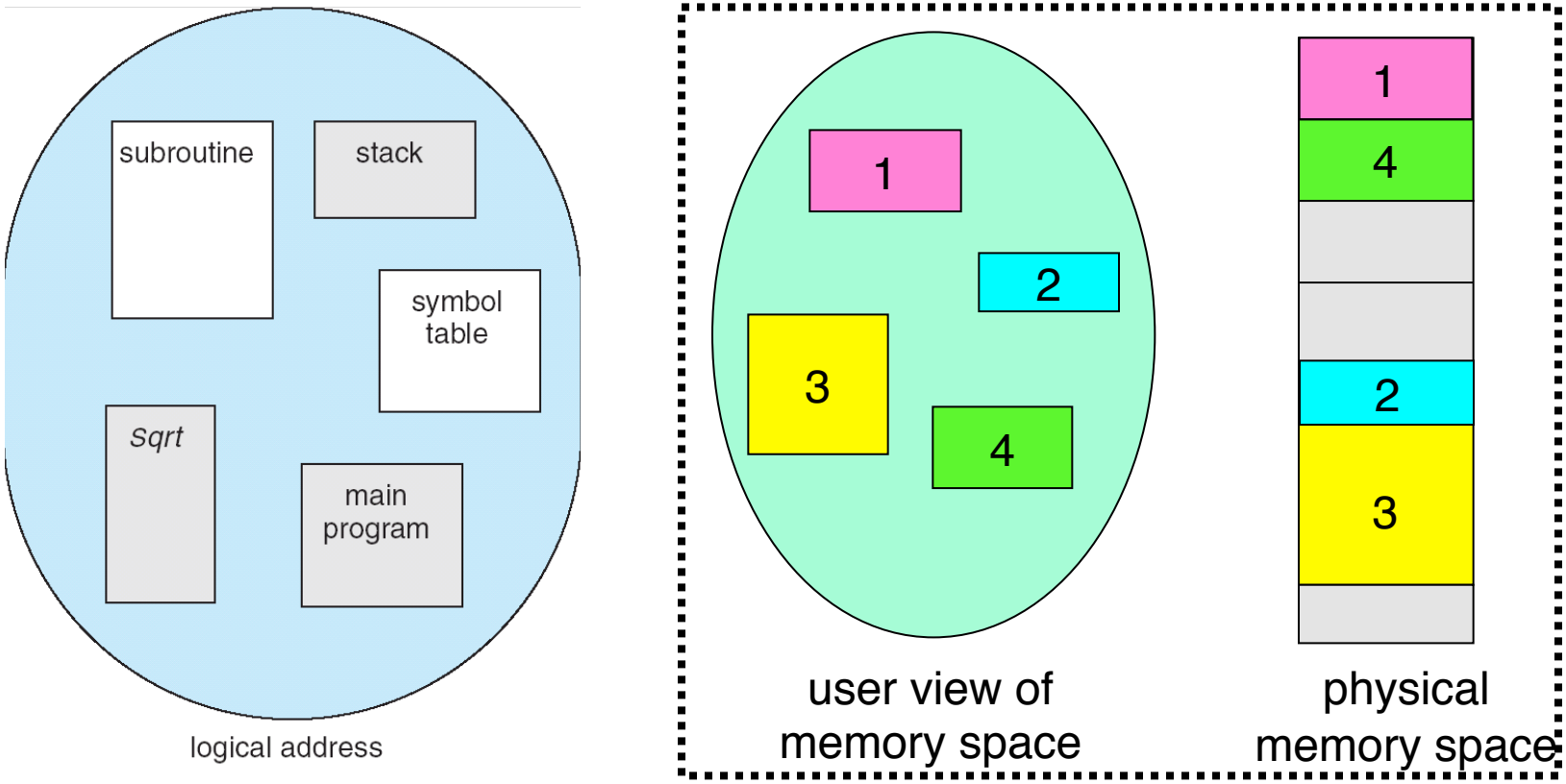
Summary Paging



Paging Hardware Optimisation

- Page table usually big (i.e. 10^6 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

More Flexible: Segmentation

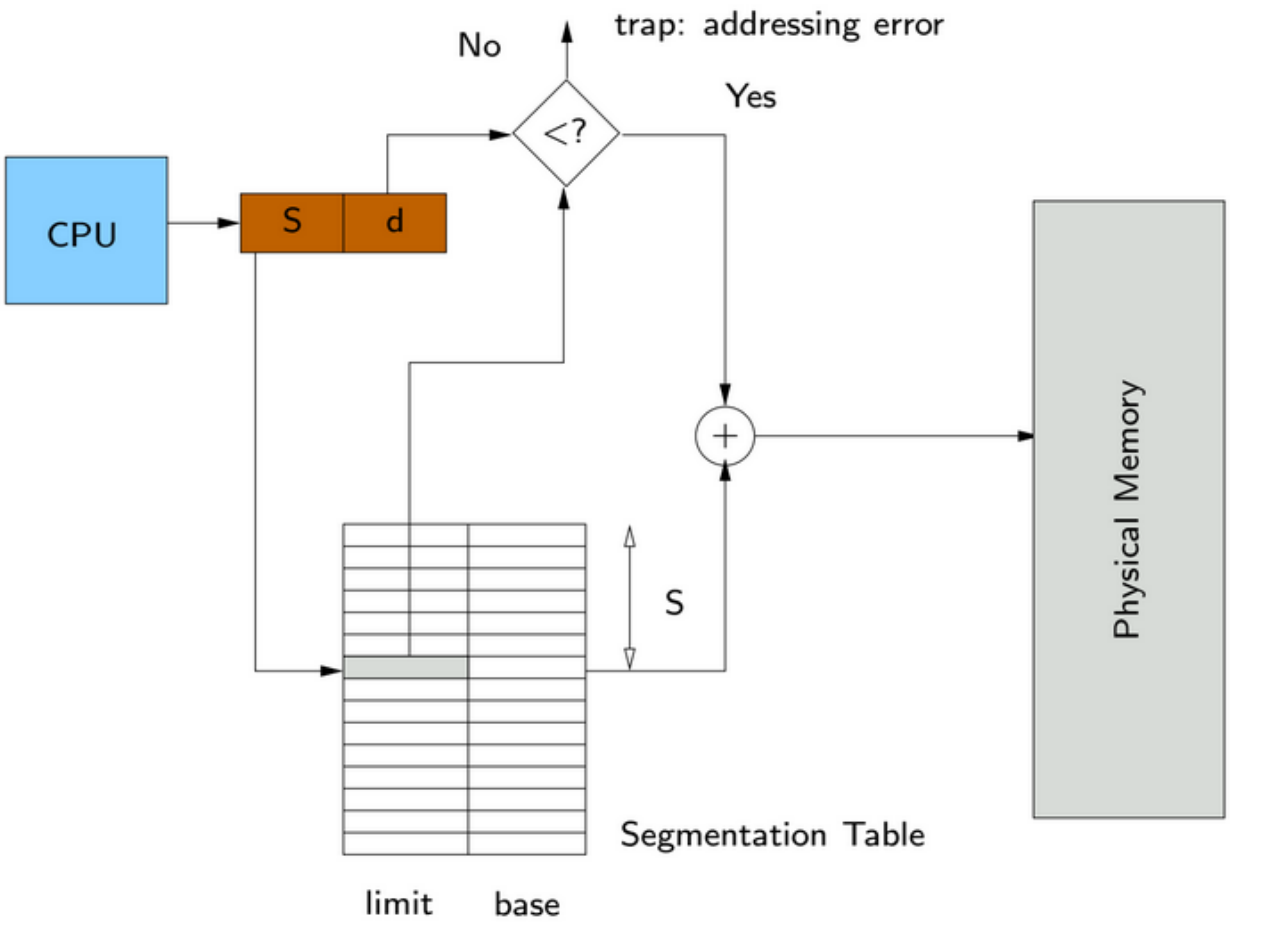


- Logical View: multiple separate segments
  - Typical: Code, Data, Stack
  - Others: memory sharing, etc
- Each segment is given region of contiguous memory
  - Has a base and limit
  - Can reside anywhere in physical memory

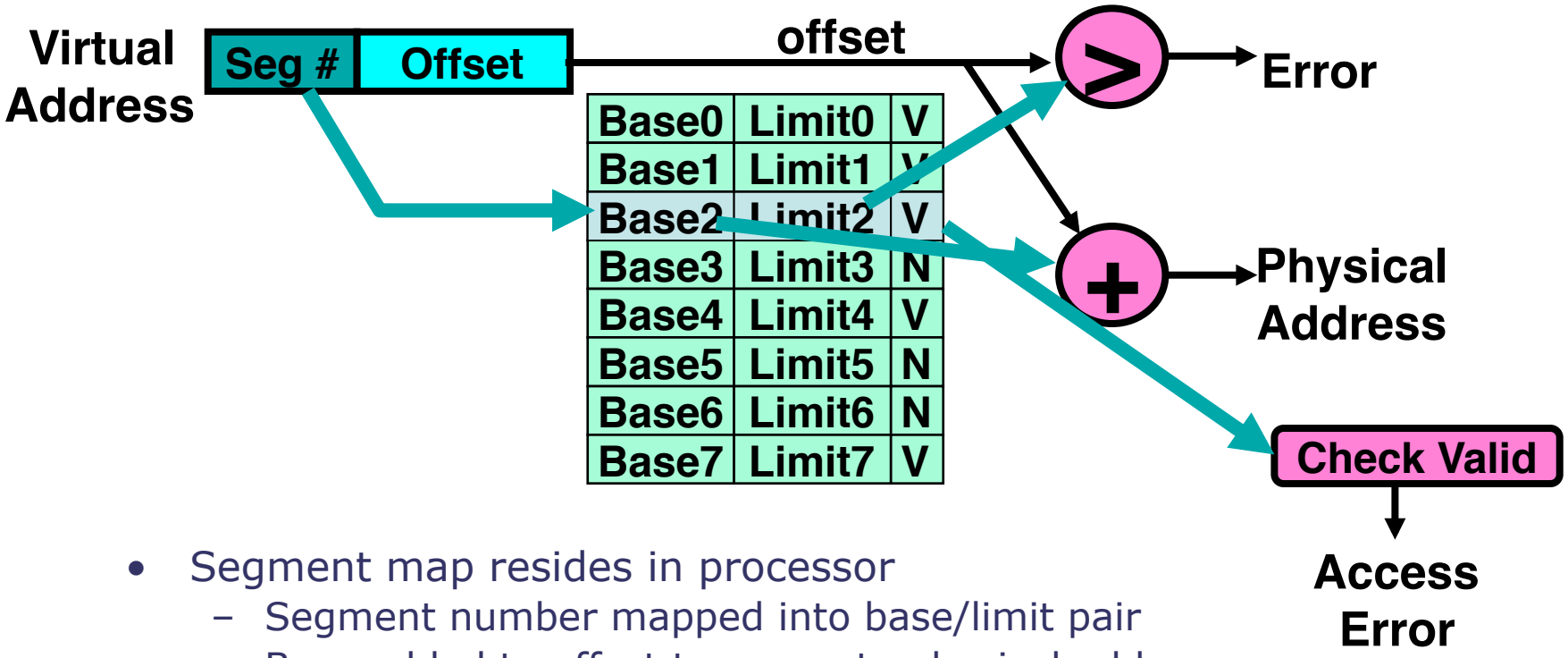
Segmentation Technique

- **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
- Entries in the segmentation table include the base and limit registers for a segment
  - Association of protection with the segments
- Fragmentation:
  - **Internal**: none
  - **External:** not solved, but less severe than variable partitioning because of the smaller pieces a process is divided into

Segmentation Hardware



Implementation of Multi-segment Model



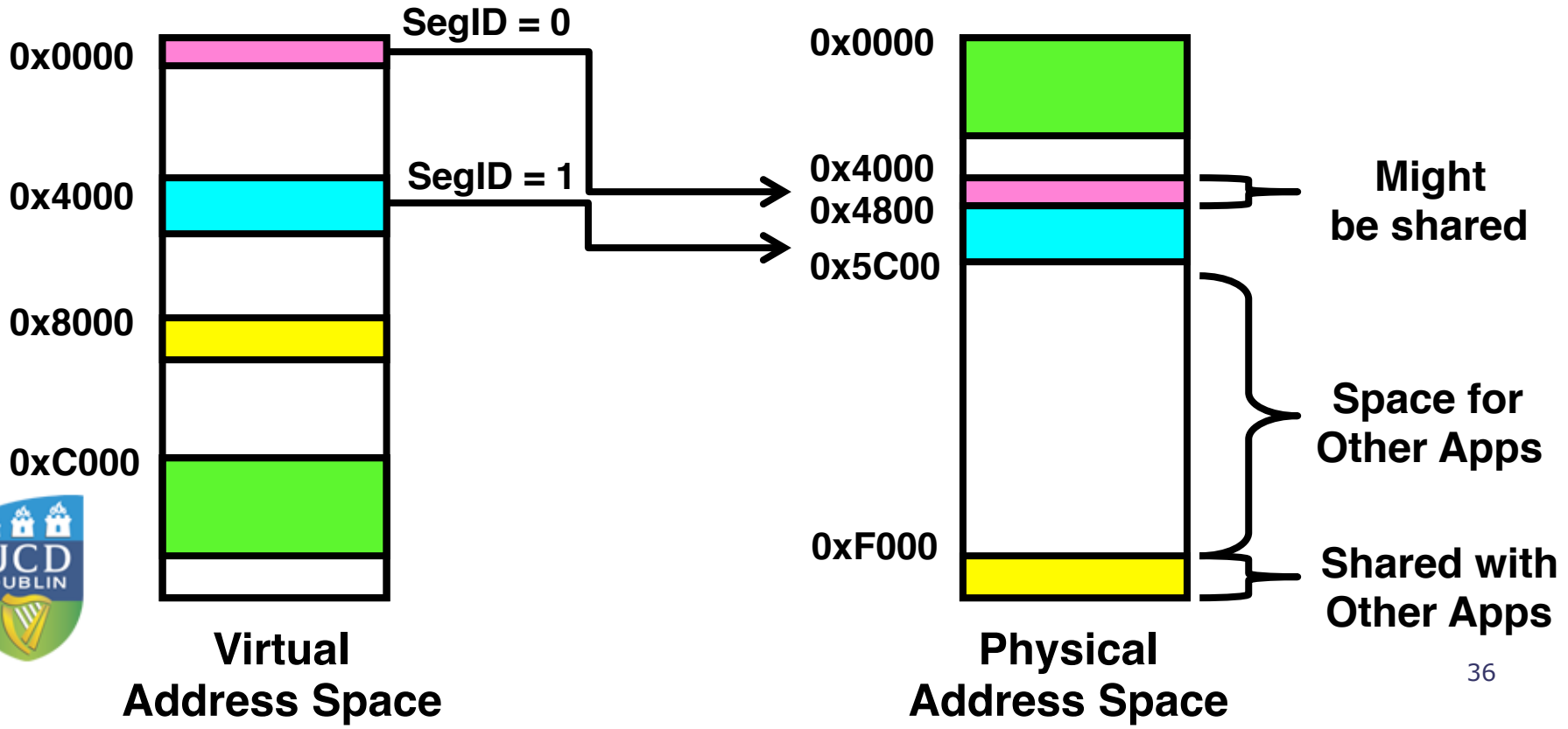
- Segment map resides in processor
  - Segment number mapped into base/limit pair
  - Base added to offset to generate physical address
  - Error check catches offset out of range
- As many chunks of physical memory as entries
  - Segment addressed by portion of virtual address

- What is "V/N" (valid / not valid)?
  - Can mark segments as invalid; requires check as well

Example: 4 Segments (16 bit addresses)

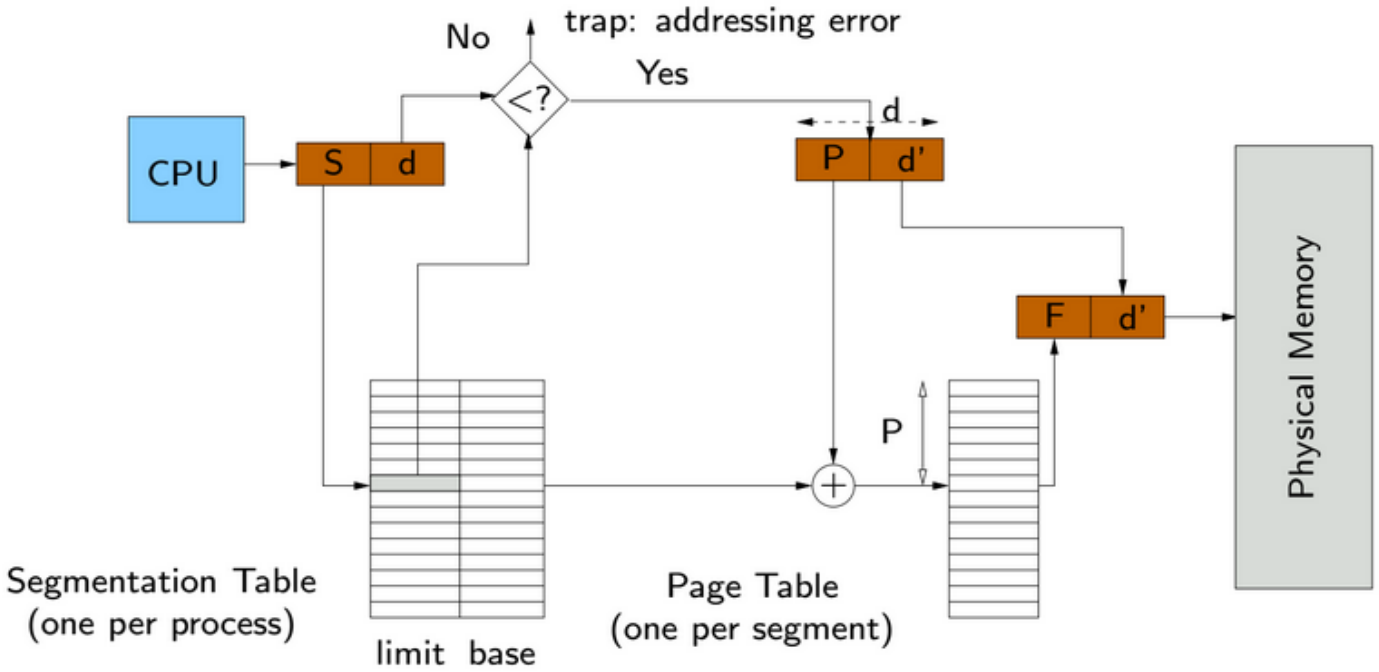
Virtual Address Format

Seg ID #	Base	Limit
0 (code)	0x4000	0x0800
1 (data)	0x4800	0x1400
2 (shared)	0xF000	0x1000
3 (stack)	0x0000	0x3000



Paged Segmentation

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



- base: lowest address of the page table for the segment

Paged Segmentation

- **Advantages:**
  - 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

Conclusion

- 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

- An ***address binding mechanism*** is needed so the addresses used in the program relate to correct physical addresses
  - ***Logical address*** – generated by the CPU; also referred to as ***virtual address***
  - ***Physical address*** – address seen by the memory unit

- Fragmentation is a key problem with basic memory mapping techniques

- Memory is a resource that must be multiplexed
  - ***Controlled Overlap*:** only shared when appropriate
  - ***Translation***: Change virtual addresses into physical addresses
  - ***Protection*:** Prevent unauthorized sharing of resources

Conclusion

- **Contiguous allocation:** section of consecutive physical memory addresses (partition) is allocated to a process
- **Noncontiguous allocation:** memory allocated to a process is divided into more than one partition (block), which can be scattered across memory

- **Paging**
  - Physical memory partitioned in small equal fixed-size chunks (***frames***)
  - Logical memory divided in chunks of same size as the frames (***pages***)
  - Logical addresshas two parts a ***page number*** and an ***offset*** within page

- **Segmentation**
  - Logical memory divided into a number of segments, each of possibly different length
  - Logical address consists now of a ***segment number*** and an ***q***within segment
  - Segmentation table include the base and limit registers for a segment