# Systems Programming – Part 2
# Concurrent Systems Programming

Dr Lauritz Thamsen
lauritz.thamsen@glasgow.ac.uk
https://lauritzthamsen.org

# Topics of Part 2 of SP(GA)

- Intro to Concurrency (with Processes and Threads)
- Process/Thread Synchronisation
- More on Process Management (from an OS Perspective)
- Concurrency Beyond Threads & Limits of Scalability
- **Virtual Memory & Levels of Storage**

## Lecture Outline

- Memory/Storage Hierarchy
  - **Storage Levels**
  - Caching

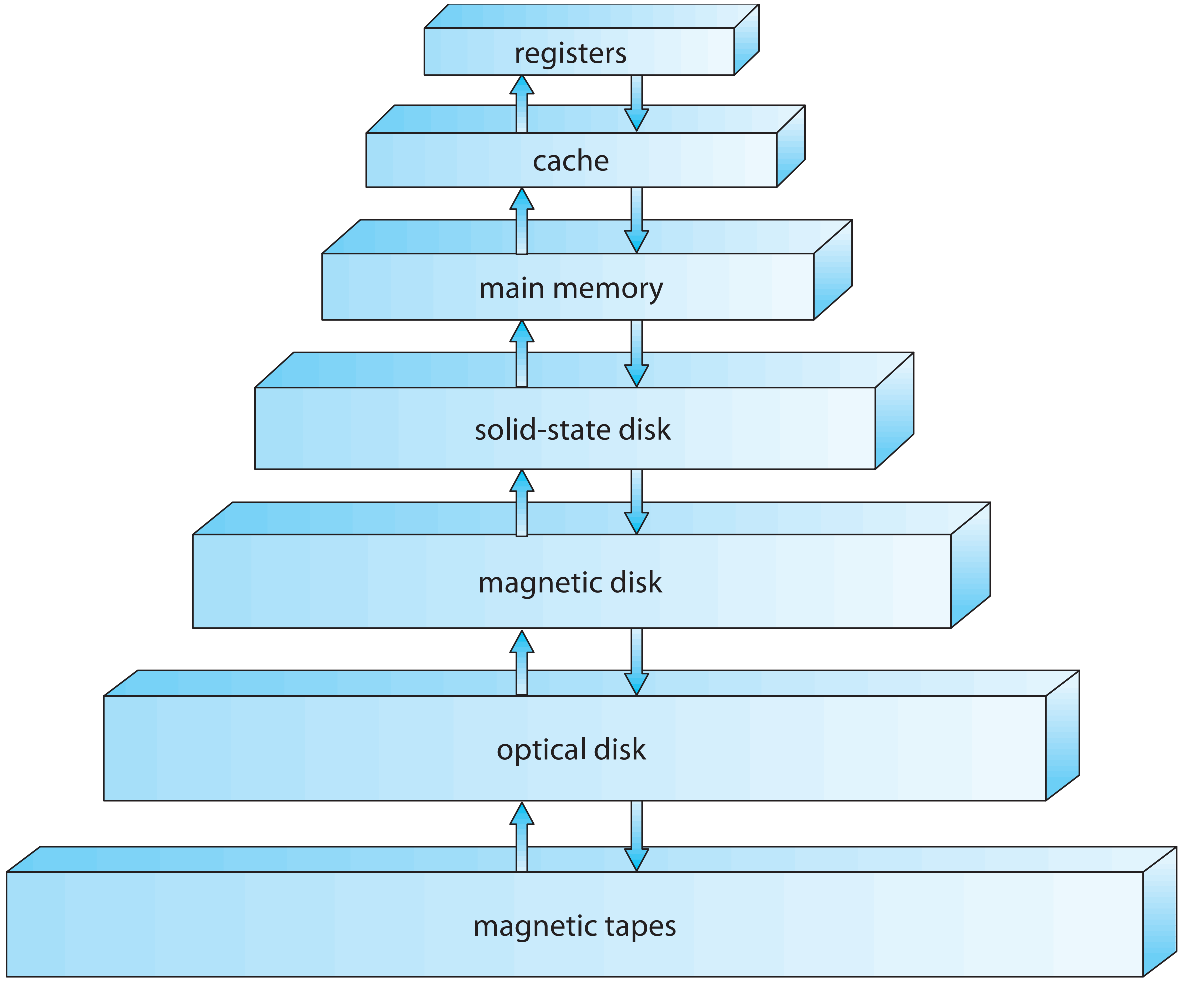
- Virtual Memory
  - Linking, Address Spaces, and Swapping
  - Memory Allocation
  - Paging
  - Implementation of Paging
  - Demand Paging and Page Replacement

## Storage Levels

- Memory/storage systems are organized in a hierarchy
  - Speed – Size – Cost – Availability & Ruggedness

- Caching
  - Holding a copy of information in a faster storage level

## Memory/Storage Hierarchy



University of Glasgow

# Some Metrics for Storage Levels

| Level | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Name | registers | cache | main memory | solid state disk | magnetic disk |
| Typical size | < 1 KB | < 16MB | < 64GB | < 1 TB | < 10 TB |
| Implementation technology | custom memory with multiple ports CMOS | on-chip or off-chip CMOS SRAM | CMOS SRAM | flash memory | magnetic disk |
| Access time (ns) | 0.25 - 0.5 | 0.5 - 25 | 80 - 250 | 25,000 - 50,000 | 5,000,000 |
| Bandwidth (MB/sec) | 20,000 - 100,000 | 5,000 - 10,000 | 1,000 - 5,000 | 500 | 20 - 150 |
| Managed by | compiler | hardware | operating system | operating system | operating system |
| Backed by | cache | main memory | disk | disk | disk or tape |

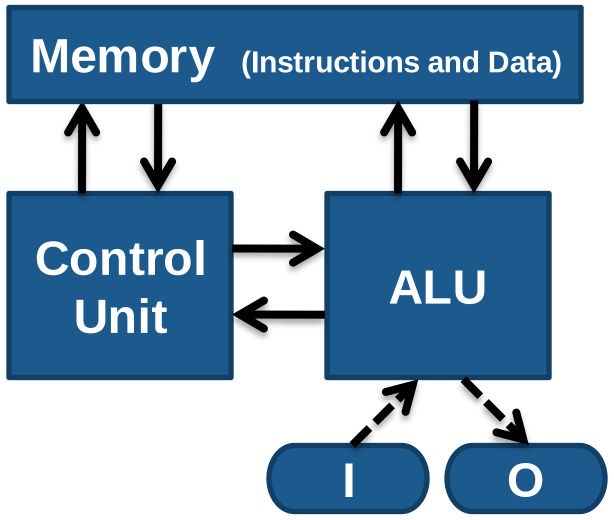
## Lecture Outline

- Memory/Storage Hierarchy
  - Storage Levels
  - **Caching**

- Virtual Memory
  - Linking, Address Spaces, and Swapping
  - Memory Allocation
  - Paging
  - Implementation of Paging
  - Demand Paging and Page Replacement

# Re-Cap: Computer Architecture

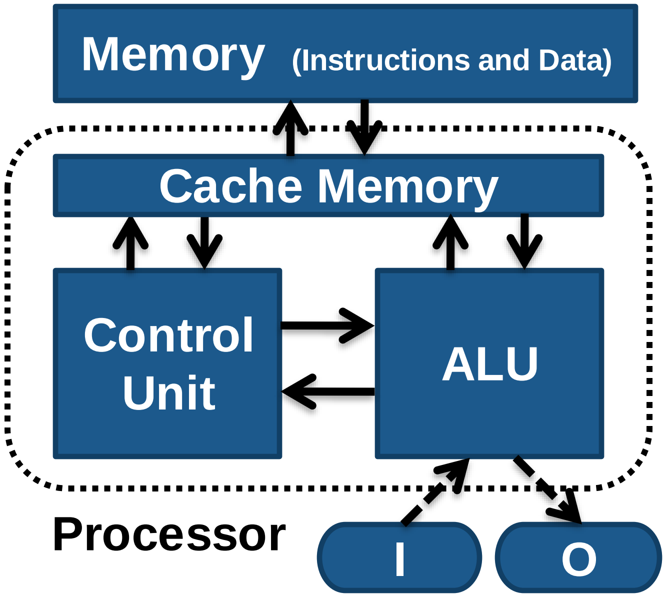
- **Main memory** and **registers** are the only storage the **CPU** can access directly
  - Programs must be loaded (from disk) into main memory for them to be executed
  - Code and data need to be loaded from main memory to CPU registers before they can be decoded/executed/operated on
  - A register access takes one(-ish) CPU clock cycle, while main memory accesses can take many cycles



Von Neumann Computer Architecture

## Caching (1/2)

- The aim: **Make the common case fast**
  - Introduce two memories:
    - The **main memory** is large, but slow
    - The **cache memory** is fast enough to match the processor speed but it is small
  - Keep recently accessed data/instructions in the cache
  - Not all of the main memory can fit in the cache…
  - … but there is a high probability that a memory access will refer to data already in the cache



- How? → Program code and data exhibit **spatial** and **temporal locality**
  - Spatial: loops in code, sequential access to data
  - Temporal: likely that the same data/instructions will be reused before long

## Caching (2/2)

- On every memory access, **hardware** looks for the memory address in the cache memory
  - If the address refers to data in the cache, it is a **cache hit**
    - The data is retrieved quickly, at processor speed
  - Otherwise, it is a **cache miss**
    - Primary memory is accessed, and the result is placed in the cache for future use
    - The processor has to wait for the slow memory access to complete

- Implementation:
  - Amount of data in a cache location (called a *cache line)*
  - Given an address, determine whether the data is in the cache and where
    - If not, determine where to put the data in the cache, after the memory fetch
    - If cache is full, determine which locations to overwrite: ***cache replacement policy***

## Lecture Outline

- Memory/Storage Hierarchy
  - Storage Levels
  - Caching

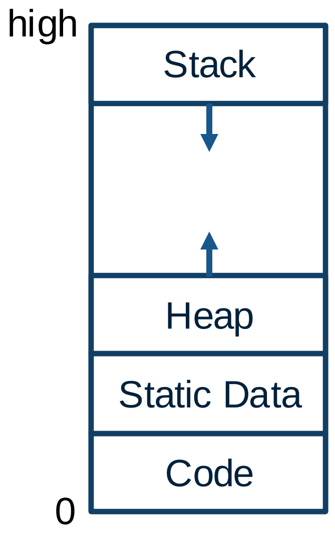
- Virtual Memory
  - **Linking, Address Spaces, and Swapping**
  - Memory Allocation
  - Paging
  - Implementation of Paging
  - Demand Paging and Page Replacement

## Memory Management

- You already know about the stack and heap memory of a program
  - You also know how to allocate such memory in C (using local variables, parameters, and malloc)
- But what if we have **many programs and processes**? How does each know which memory it is supposed to use (and which not)?
- Memory is managed not just by us and our runtimes (e.g. using garbage collectors) but also by **operating systems**

## Re-Cap: Linking

- Lifecycle of a program:
  - **High level language** source code $\rightarrow$ (compiler) **Assembly** code $\rightarrow$ (assembler) **Object** code $\rightarrow$ (linker) **Executable** code $\rightarrow$ (loader) **Runnable** code in memory

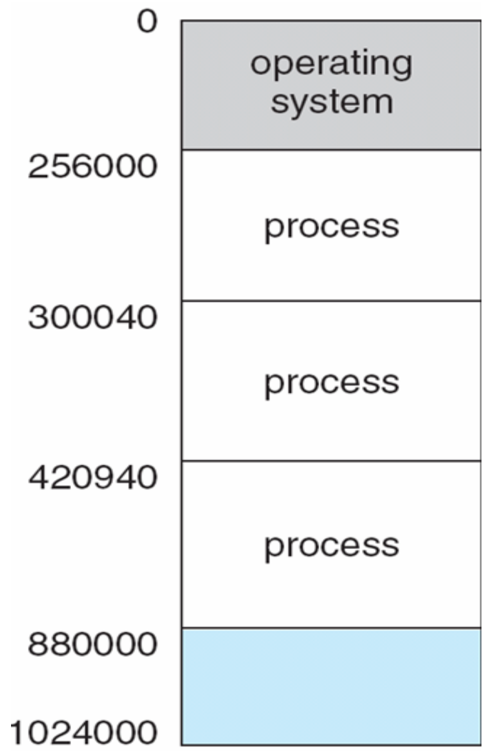


- Basic linking example:
  - Code assumed to be loaded at memory position 0
  - What if there were multiple modules in different source code files?
    1. Each would be compiled as if to be loaded at memory position 0
    2. All files would then be concatenated into one, with the main module first
    3. Each module, apart from the main module, would then be relocated (→ Calculate its starting address in the concatenated file, increment all internal addresses by that)
    4. Then all external names (variables, functions, etc.) are resolved and replaced with their actual address
    5. Then the resulting file is saved, ready to be executed
- Also known as: **static linking**
  - **Dynamic linking**: same, only without Step 5, and Steps 2-4 only happen at runtime

## Memory Address Space

- Each process is assumed to have its own *address space*
  - A region of memory locations
  - Usually split into two main parts:
    - **Program segment**: contains code, is read-only, may be shared across processes
    - **Data segment**: contains variables/data, usually is read-write, should only be accessible by its owner process
  - The lowest address in an address space is called **origin**

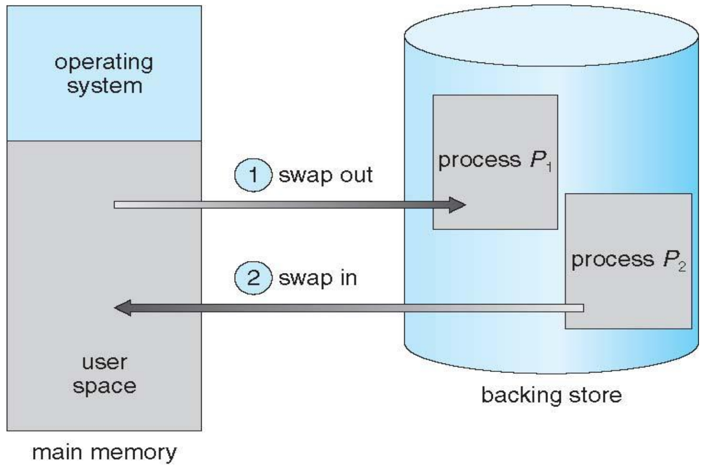
- When multiple processes are running, their code is loaded at an arbitrary location in memory; what then?
  - What happens to the addresses in the code?



Source: A. Silberschatz, "Operating System Concepts", 9th Ed., 2012.

# Memory Address Space

- Could have loader re-relocate all of the addresses after program loaded into RAM
  - Would take more time to load the program
  - Would need to be repeated if process is swapped out/in
  - Does not provide memory protection: What if a **program accesses memory locations beyond its own**?



Source: A. Silberschatz, "Operating System Concepts", 9th Ed., 2012.

- Okay, so what are other ideas?

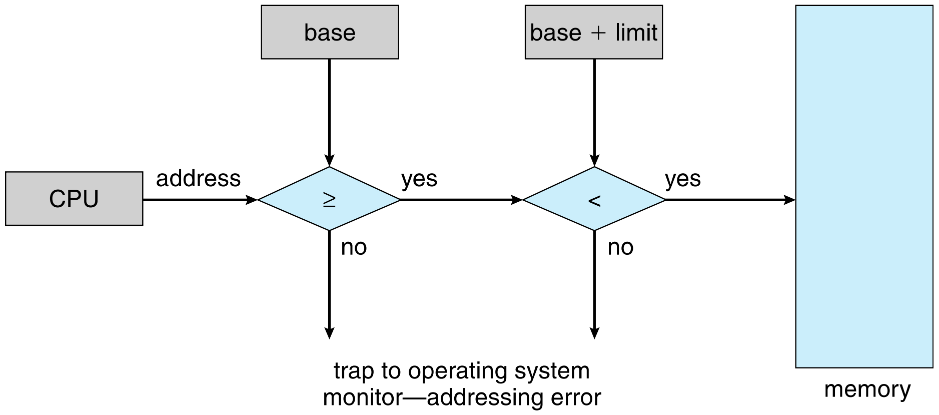
## Lecture Outline

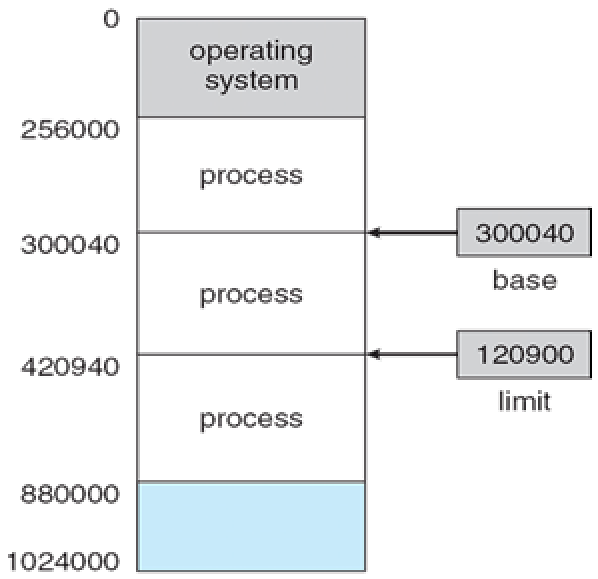
- Memory/Storage Hierarchy
  - Storage Levels
  - Caching

- Virtual Memory
  - Linking, Address Spaces, and Swapping
  - **Memory Allocation**
  - Paging
  - Implementation of Paging
  - Demand Paging and Page Replacement

# Base Register and Limit Register

- Idea #1: Let’s use special registers to store the first and last address in a process’s address space
  - Base Register (BR) and Limit Register (LR)
  - Add BR’s value to the memory addresses of all memory-accessing instructions
  - Check that the resulting memory address is not beyond LR
  - Only the OS can set/update the values in BR and LR
- A **virtual address space**!





Source: A. Silberschatz, “Operating System Concepts”, 9th Ed., 2012.

## Memory Management

- Idea #2: Partition RAM into fixed-size partitions, allocate one to each running process
  - Would work, but is inflexible and clunky
  - Creates **internal fragmentation**: space within partitions goes unused

- Idea #3: Variable-sized partitions
  - The OS keeps track of lists of allocated ranges and “holes” in RAM
  - When a new process arrives, it blocks until a hole large enough to fit it is found
  - Several strategies are available: e.g. first or best fit
  - If a hole is too large, it is split in two parts: one allocated to the new process, one added to the list of holes
  - Can create **external fragmentation**: space in between partitions too small to be used

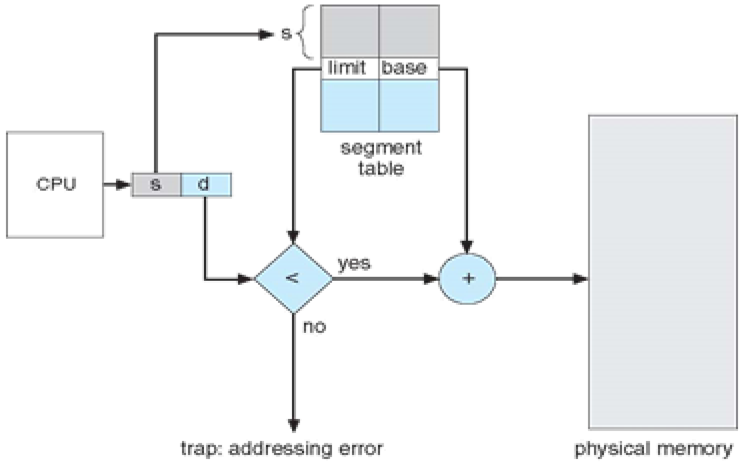
## Memory Management

- So far, we have assumed that a process’s memory address space is allocated as a big contiguous space in RAM
  - Is this realistic?
  - What could we do to allow non-contiguous allocation of memory locations?

# Memory Management

- Idea #4: Segments
  - Maintain several segments (~= mini address spaces) per process
  - Maintain a table with info for each segment
    - Base: Starting address for that segment
    - Limit: Size of the segment

➢Extension of BR/LR but to multiple segments



Source: A. Silberschatz, "Operating System Concepts", 9th Ed., 2012.

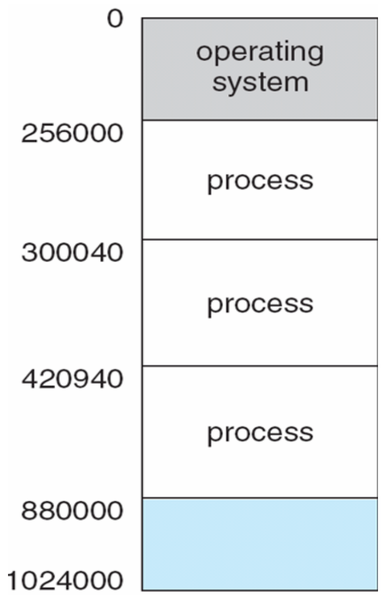
## Lecture Outline

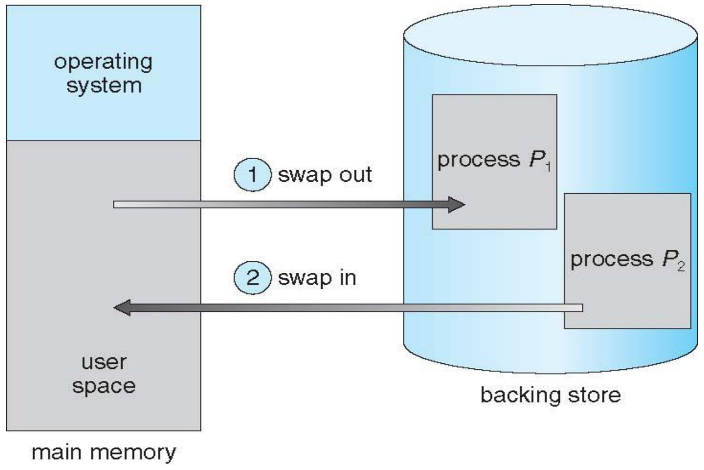
- Memory/Storage Hierarchy
  - Storage Levels
  - Caching

- Virtual Memory
  - Linking, Address Spaces, and Swapping
  - **(Still) Memory Allocation**
  - Paging
  - Implementation of Paging
  - Demand Paging and Page Replacement

# Re-Cap from Yesterday

- When multiple processes are running, they are loaded at different locations in the main memory
  - What happens to the addresses of each? How does each process know which memory locations to use? How is it ensured that a process only uses its own?
  - … when the set of processes changes dynamically (e.g. due to swapping)





## Summary of Ideas # 1 - 4

- Idea #1→ Let's use special registers to store the first and last address in a process's address space (Base Register & Limit Register)

- Idea #2 → Partition RAM into fixed size partitions, allocate one to each running process
  - Would work, but is inflexible and clunky
  - Creates **internal fragmentation**: space within partitions of some processes goes unused

- Idea #3 → Variable-sized partitions:
  - The OS keeps track of lists of allocated and unallocated ranges in RAM
  - When a new process arrives, it blocks until an unallocated range large enough to fit it is found
  - If an unallocated range is too large, it is split into two parts: one allocated to the new process, one added to the list of unallocated ranges
  - Can create **external fragmentation**: space in between partitions too small to be used

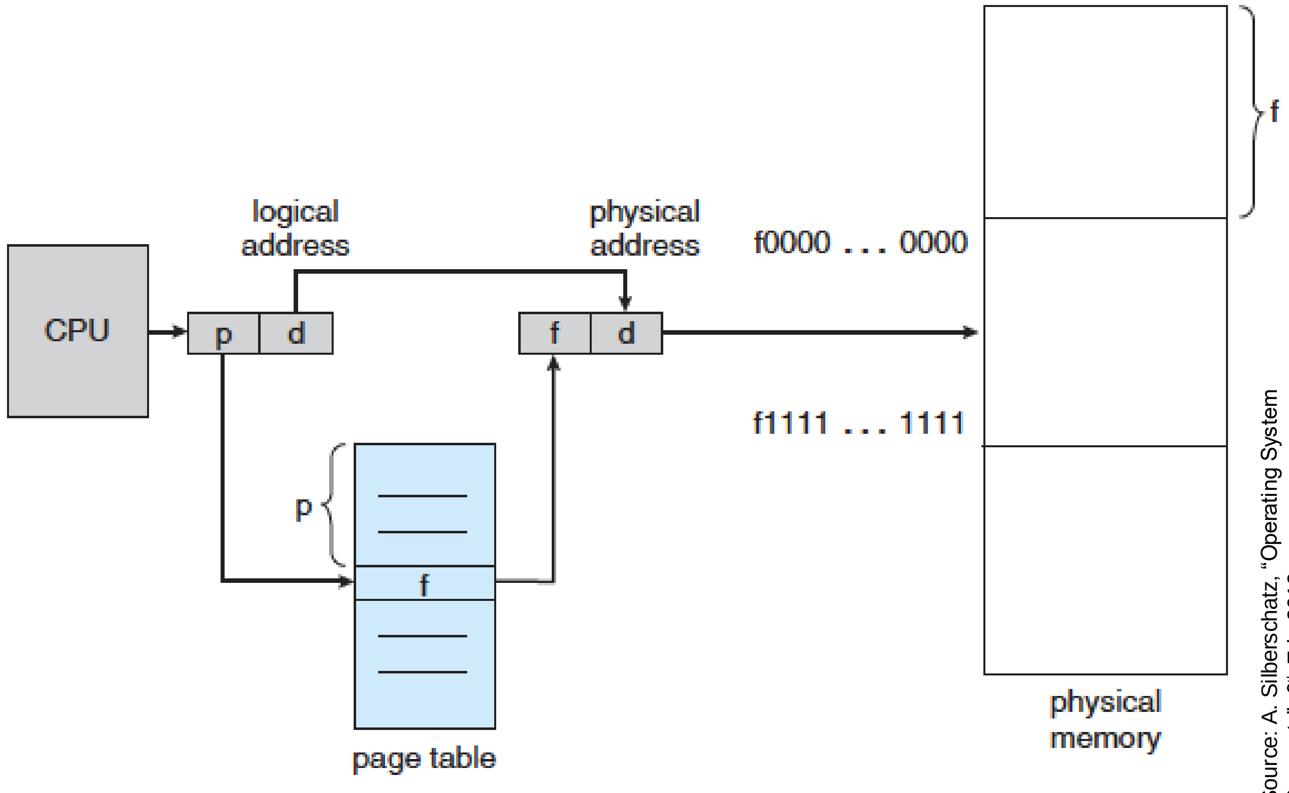
- Idea #4:→ Segments: Maintain several segments per process and a table with information for each segment (extension of BR/LR to "mini" address spaces)
  - … but how to map segments to unallocated ranges in RAM?

## Lecture Outline

- Memory/Storage Hierarchy
  - Storage Levels
  - Caching
- Virtual Memory
  - Linking, Address Spaces, and Swapping
  - Memory Allocation
  - **Paging**
  - Implementation of Paging
  - Demand Paging and Page Replacement

## Paging

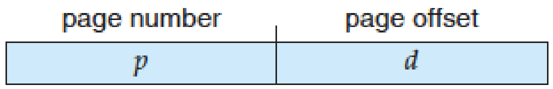
- Idea #5



Source: A. Silberschatz, "Operating System Concepts", 9th Ed., 2012.

## Paging

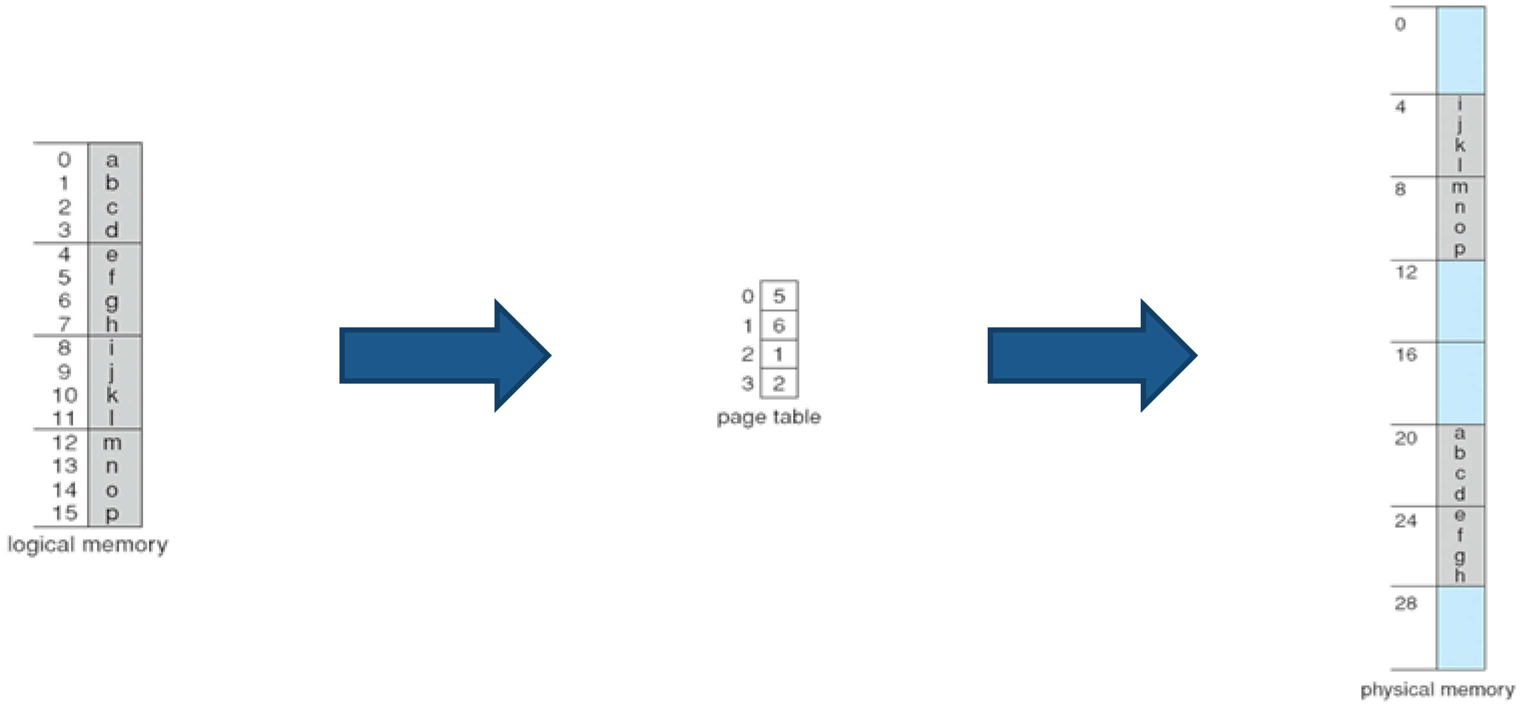
- Partition address space in **equally sized, fixed-size** partitions (== **pages**)
  - Size always a **power of 2** -- i.e., $2^d$
  - Typical page sizes: 1KB – 4KB (could get even bigger in modern OS)
  - Each page is kept on disk, so there can be **a lot** of them
- A location in the address space can be given as either an ***address***, or **a *page number*** plus an ***offset*** within set page
- Given an address with *n* bits:
  - The least significant *d* bits are the offset
  - The most significant *p* = *n* - *d* bits define the page number



- Partition a large portion of RAM into **page frames**, each of size equal to a page
- Maintain a **page table** for each address space; each entry contains:
  - A **resident/valid bit**: 1 if page is loaded into a page frame
  - A **frame address**: Contains the physical/real address of the frame (if resident bit is 1)
  - Can be per process or contain additional data (e.g., process ID) for protection
- **Paging** means moving a page/frame of data from disk to memory or vice-versa

## Paging Example

- Assume we have a main memory of 32 bytes and 4-byte frames → we need 8 frames and an offset with 2 bits (d = 2)
- Assume our process's address space (i.e. logical memory) is 16 bytes large and 4-byte pages → we need at least 4 distinct page numbers



## Allocation of Frames

- Each process needs a minimum number of frames
  - Maximum is the total frames in the system, minus frames allocated to the OS
- Three major allocation schemes:
  - **Fixed allocation**
    - Divide available frames equally among processes
  - **Proportional allocation**
    - Give each process a percentage of frames equal to its size divided by the sum of sizes of all processes
  - **Priority allocation**
    - Like proportional allocation, but taking into account the priority of a process (possibly in combination with its size)

## Lecture Outline

- Memory/Storage Hierarchy
  - Storage Levels
  - Caching

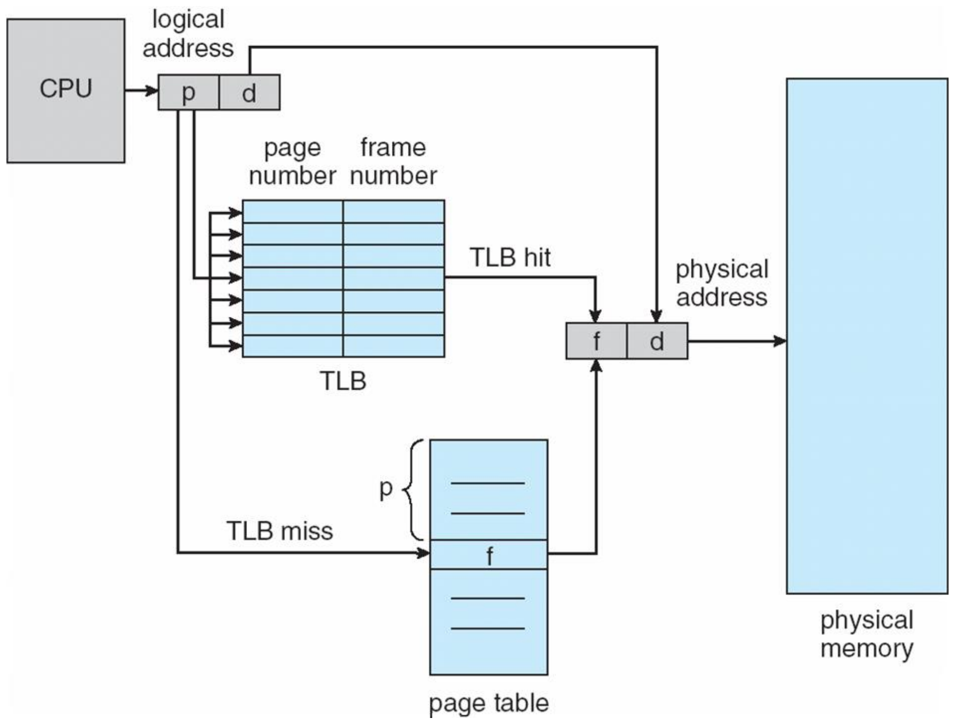
- Virtual Memory
  - Linking, Address Spaces, and Swapping
  - Memory Allocation
  - Paging
  - **Implementation of Paging**
  - Demand Paging and Page Replacement

## Performance Considerations

- The page table is an Operating System construct but with hardware assistance…
- The page table is kept in main memory
  - Page-table base register (PTBR) points to the beginning of page table location
  - Page-table length register (PTLR) indicates the size of the page table
- But wait! In this addressing scheme, **every** data/instruction access requires **two memory accesses**!
  - One for the page table entry and one for the actual data/instruction
- **How is this acceptable?**

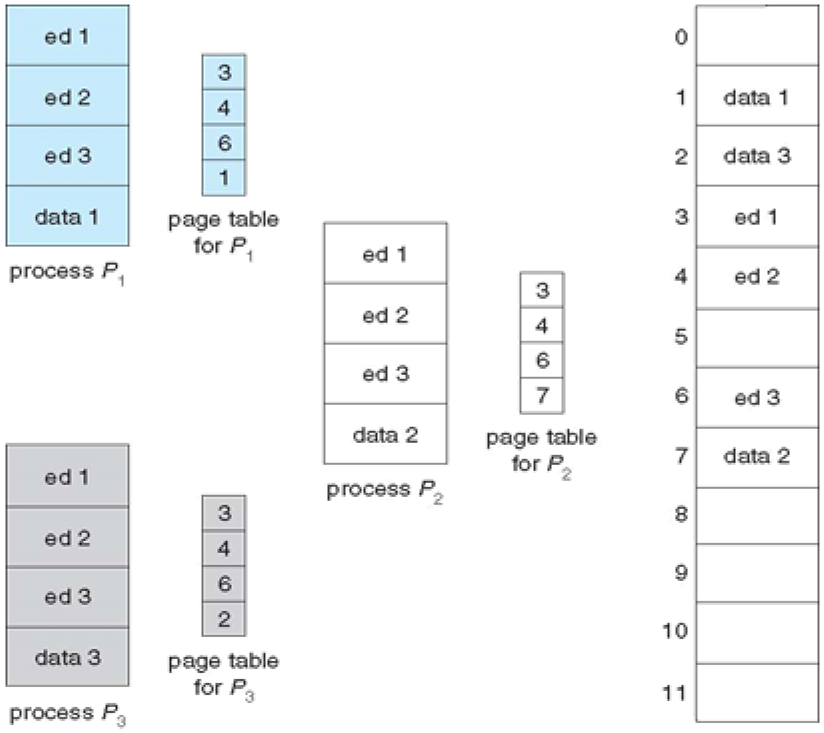
## Translation Lookaside Buffer (TLB)

- **Caching** to the rescue: there is the **Translation Lookaside Buffer (TLB)**, an on-CPU hardware cache for page table entries



## Shared Pages

- Pages may be shared across processes
  - Often done for pages containing code
- Remember *fork*?
  - Can be expensive to create a full copy of parent process's address space
- Can we make it any faster?
- Yes, with **Copy-on-Write**
  - Both processes share the same pages in memory
  - Only when a process tries to modify a page, a private copy is created



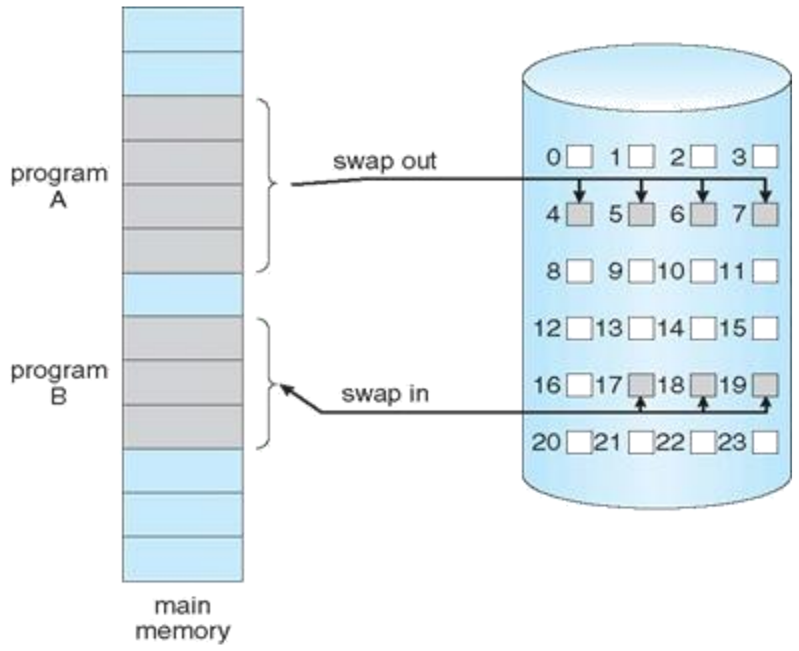
## Lecture Outline

- Memory/Storage Hierarchy
  - Storage Levels
  - Caching

- Virtual Memory
  - Linking, Address Spaces, and Swapping
  - Memory Allocation
  - Paging
  - Implementation of Paging
  - **Demand Paging and Page Replacement**

## Demand Paging

- Remember:
  - Each page is kept on disk, so there can be a lot of them
  - Paging means moving a page/frame of data from disk to memory or vice-versa
  - Initially, all pages are on disk (e.g., before executing a program)
- When should a page be moved from disk to memory?
  - All pages on process startup?
  - Every page only as needed (accessed): **Demand paging** – i.e., using a “lazy” **pager**



## Demand Paging

- What if a process tries to access a page that is not loaded in RAM?
- This leads to a **Page fault** and the pager (a.k.a. page fault handler of the memory management unit) kicks in
    1. Trap to the operating system (original process is blocked)
    2. Pager computes page location on disk and issues a disk read request to load page contents
    3. Another process executes
    4. Interrupt from the disk (I/O completed): Control passed to the pager again, which updates the page table, unblocking the original process
    5. Pager returns to the process scheduler (which selects some process)

## Demand Paging

- Virtual memory can run almost as fast as real memory, **if the proportion of instructions that cause a page fault is low enough**
- In the real world, programs perform almost all their memory accesses at addresses close to where they recently accessed data
  - Nearly all accesses are to a **working set**, which is kept in page frames
- Page faults then occur when a program changes its working set (e.g. a function has finished and another is called)

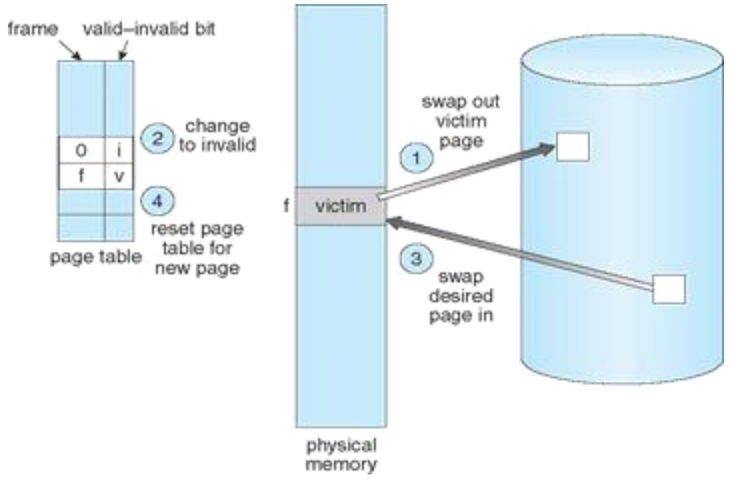
- Occasionally, a program will perform too many page faults
  - The result is that it starts to execute thousands of times slower than it should
  - This is called **thrashing**

University of Glasgow

## Page Replacement

- Normally, most or all of the page frames are in use
- In this case, when a page fault occurs, it is not possible to simply read the page into an empty frame

- The system must:
  1. Select a full frame (**page replacement algorithm**)
  2. Write its contents to disk (**page-out**)
  3. Then read the required page into this frame (**page-in**)



- In practice, it is better to keep empty frames ready
  - On a page fault, the read (to load the data) can start immediately
  - A separate write (to clear another frame) can be done afterwards

## Page Replacement

- If a page is written from a frame back to disk, and data on this page is needed again, it will have to be read back in
  - This can result in too much slow disk I/O…

- Therefore, the pager does not randomly choose a frame to clear
  - It uses a **page replacement policy** (e.g. Least Frequently Used, Least Recently Used, Not Recently Used)
  - Goal: Try to choose a frame containing a page that **probably will not be needed again soon**

## In Summary: The Three Great Advantages of Virtual Memory

- (Virtual) **contiguous address spaces** for processes, without much fragmentation of RAM

- **Memory access control**: Protecting one process’s memory (== heap space) against access by other processes (on lookup)

- **Paging**: More memory can be allocated by processes than is available physically (in RAM)

## Recommended Reading

- Silberschatz, Galvin, Gagne, Operating Systems Concepts, Sections 7.1-5 and 8.1-5