Lecture 7: Memory Management
Operating Systems – EDA093/DIT401

Vincenzo Gulisano
vincenzo.gulisano@chalmers.se

UNIVERSITY OF GOTHENBURG

What to read (main textbook)

- Chapter 3.1, 3.2, 3.7 (up to 3.7.1 - excluded)

  (extra facultative reading: 8.1-8.5 from Silberschatz Operating System Concepts)

Objective

- Discuss how different types of memory are used
- Discuss how information is continuously moved between “slow memory” and “fast memory” (accessed directly by the CPU)
- Discuss how and why user/programmer/program view of memory and physical memory are kept separated

Agenda

- Introduction
- Base and limit registers
- Logical/Virtual vs Physical addresses
- Dynamic loading and linking optimizations [self reading]
- Swapping
- Memory allocation
  - Contiguous allocation
  - Segmentation
  - Paging

Agenda

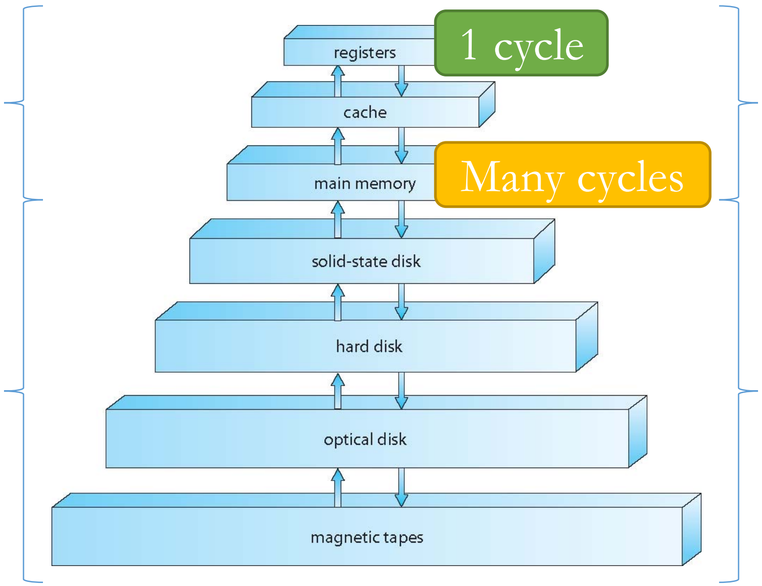
- Introduction
- Base and limit registers
- Logical/Virtual vs Physical addresses
- Dynamic loading and linking optimizations [self reading]
- Swapping
- Memory allocation
  - Contiguous allocation
  - Segmentation
  - Paging

Background and basic hardware (I)

Goal of the OS: maximize CPU utilization → Run multiple processes concurrently / in parallel → Share memory!

We first have to move it up here!

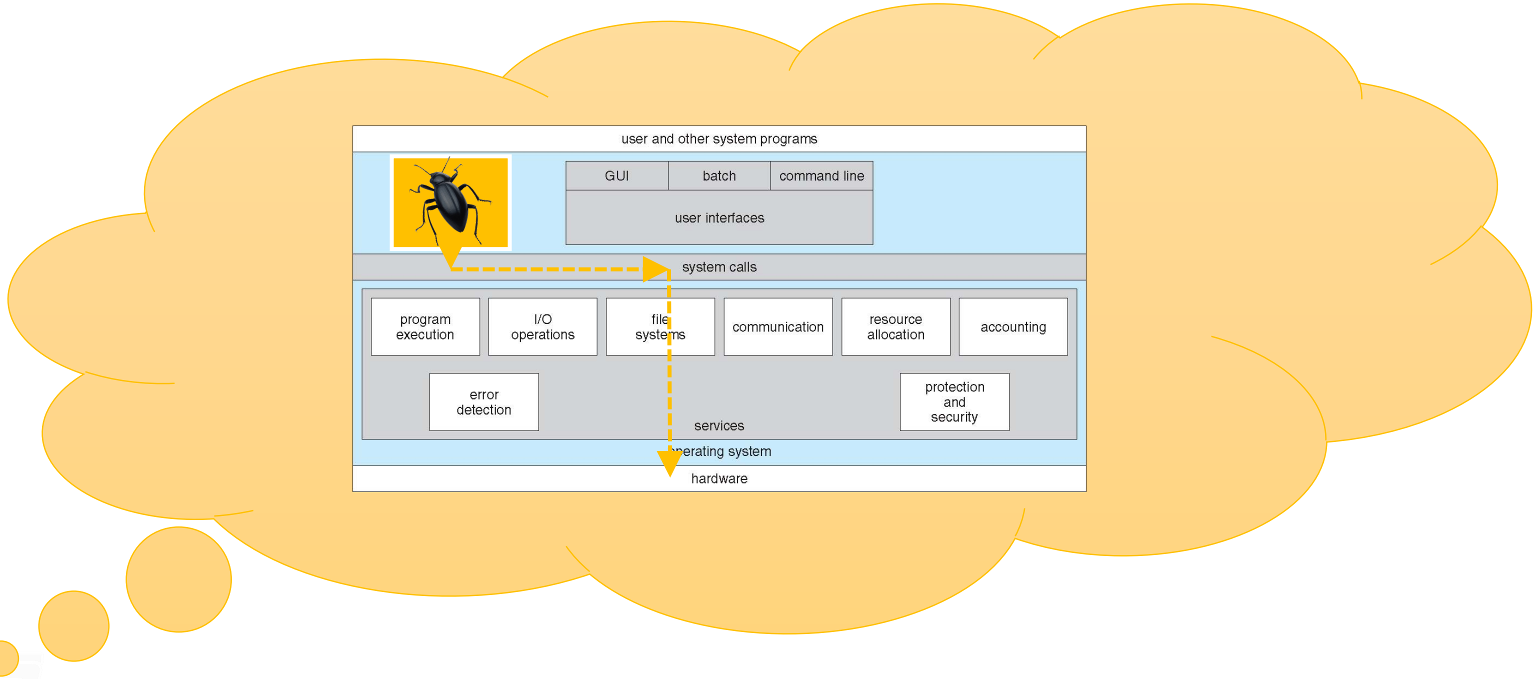
If some data the CPU wants to access is here

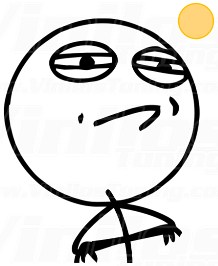


Can be accessed directly by the CPU

Cannot be accessed directly by the CPU

Background and basic hardware (II)



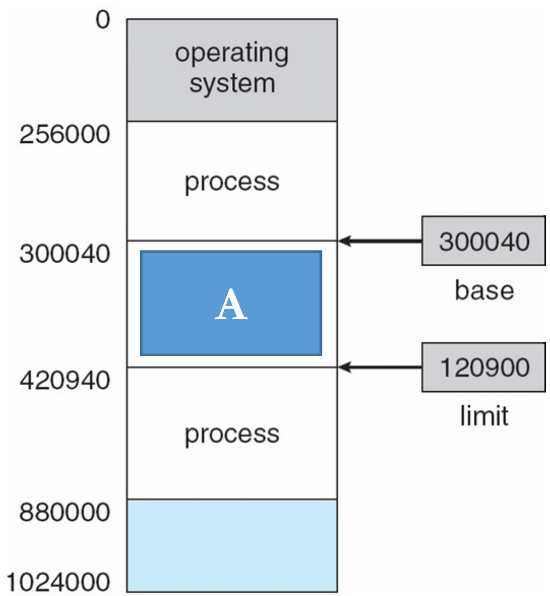


Need for fast CPU access to registers / caches and main memory?
→ hardware support without OS interaction
CHALLENGE: how to ensure correctness?

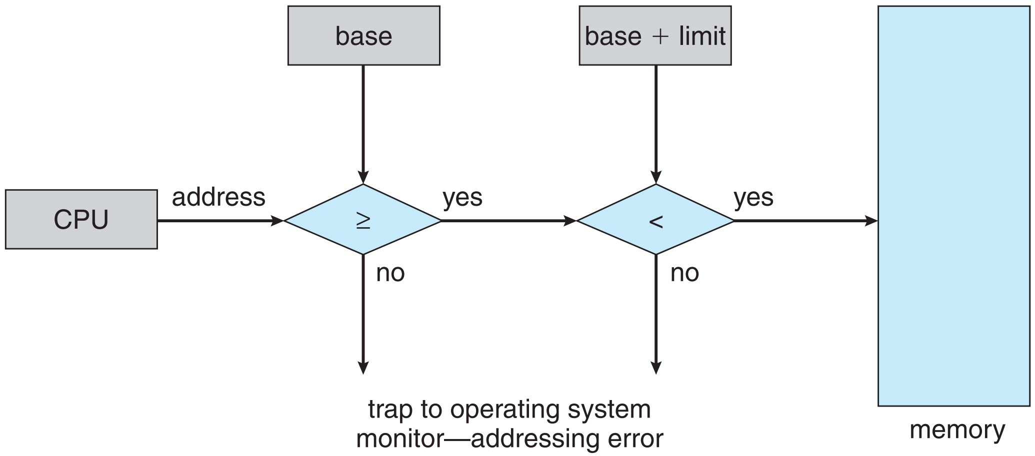
Agenda

- Introduction
- Base and limit registers
- Logical/Virtual vs Physical addresses
- Dynamic loading and linking optimizations [self reading]
- Swapping
- Memory allocation
  - Contiguous allocation
  - Segmentation
  - Paging

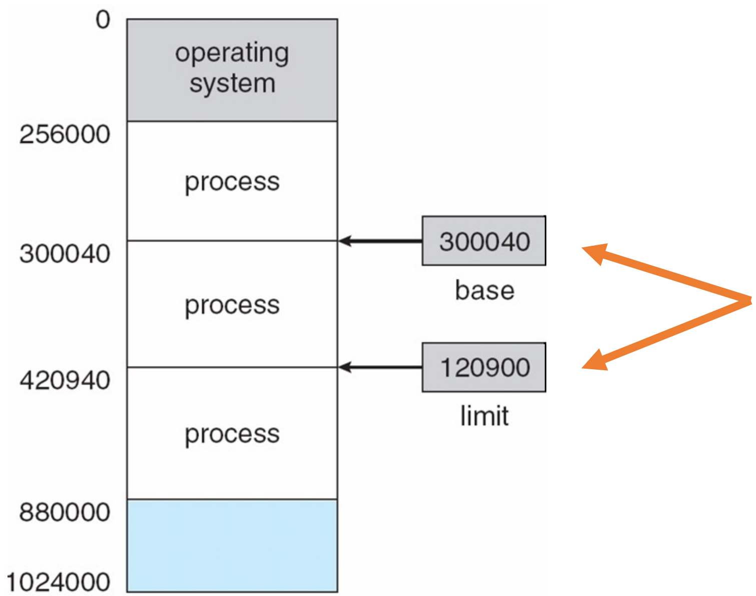
Base and Limit register (I)



- For each process, the CPU maintains two registers:
  - Base register: first address that can be access by the process
  - Limit register: number of addresses (starting from the base register) that can be accessed by the process
- Example: process A can only access addresses from 300040 to 420940



Base and Limit register (II)

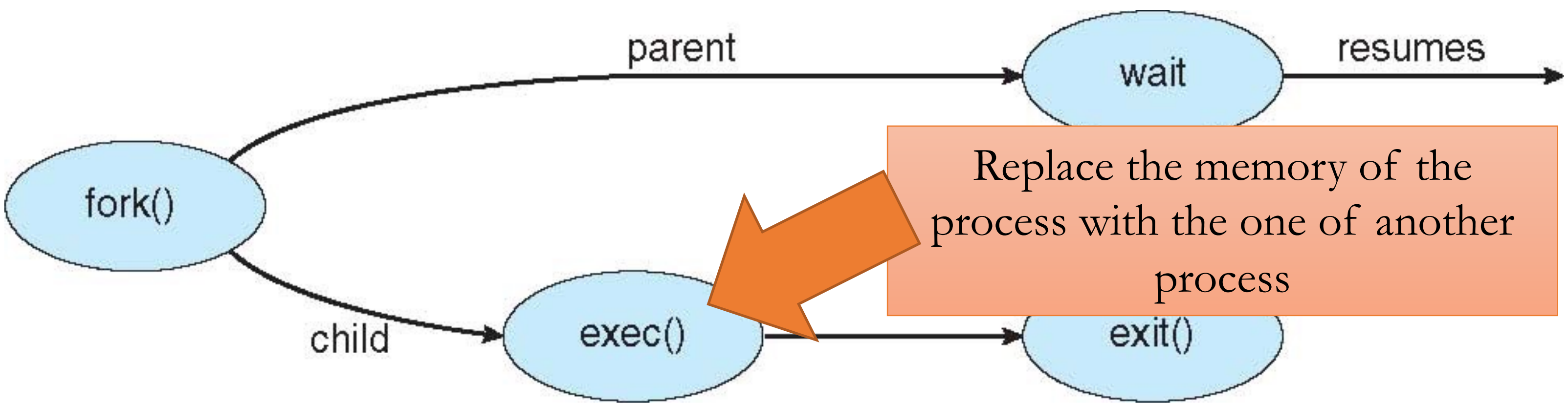


Base and Limit register loaded by the OS using privileged instructions!

Base and Limit register (III)

The operating system can access the memory of the processes without restrictions.

We saw (at least) one example of this in the previous lectures, when?



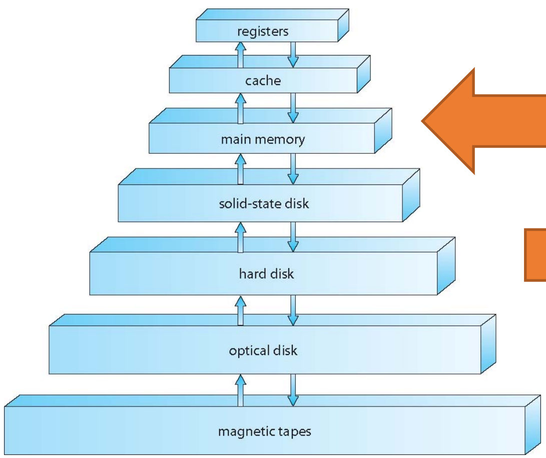
Agenda

- Introduction
- Base and limit registers
- Logical/Virtual vs Physical addresses
- Dynamic loading and linking optimizations [self reading]
- Swapping
- Memory allocation
  - Contiguous allocation
  - Segmentation
  - Paging

Address binding

...
a = b + 2
...

Where do we store a?



Once we load it into memory, we know the address

As long as the program is not loaded in memory, we do not know the address

Logical/Virtual vs Physical address space (i)

Logical/Virtual address

First address of the program 00000

…

a = b + 2

…

Address of variable a 346

Physical address (main memory)

First address of the program 14000

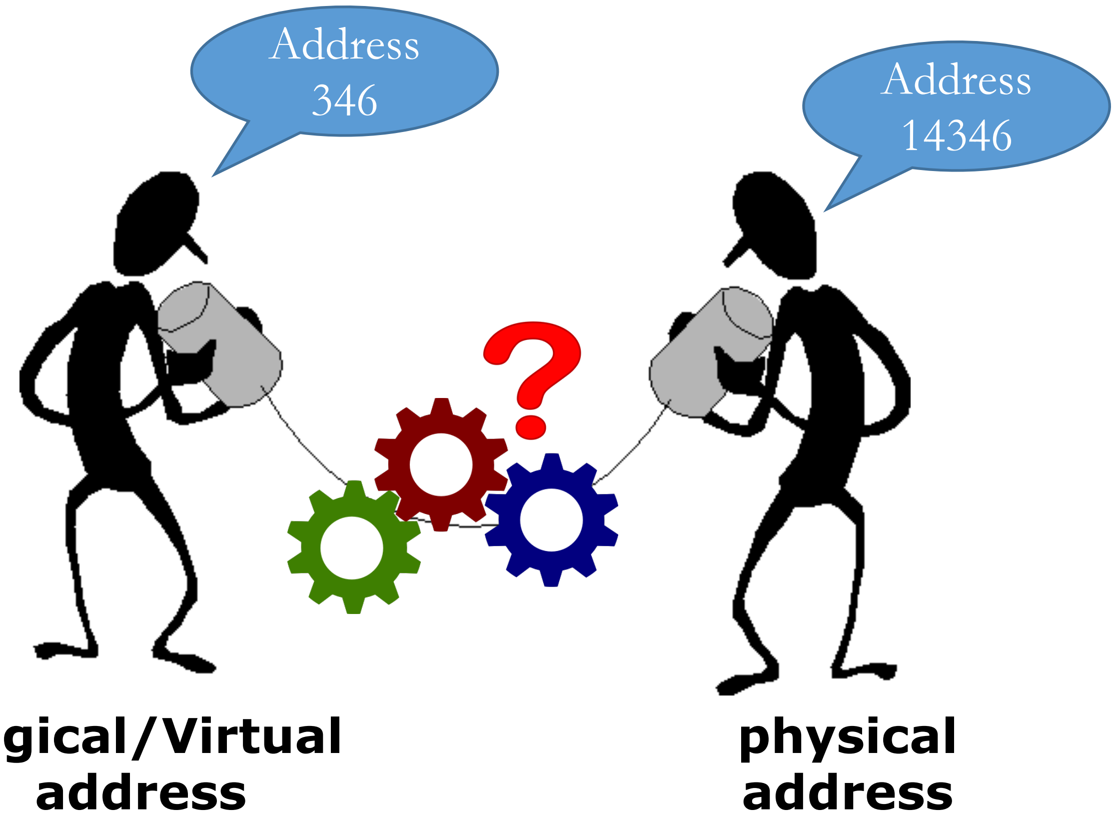
…

a = b + 2

…

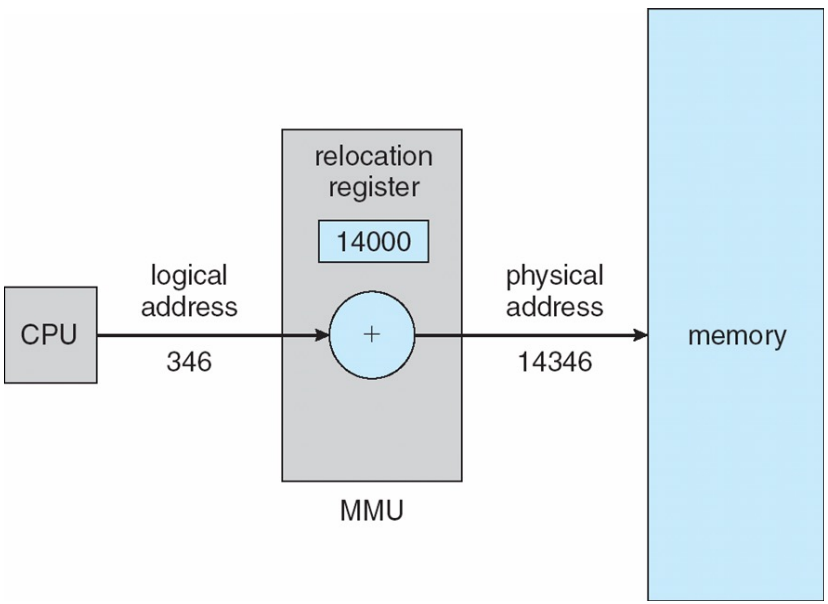
Address of variable a 14346

Logical/Virtual vs Physical address space (ii)



How to convert between logical/virtual addresses to physical addresses?

Memory-Management Unit (MMU) – Relocation Register



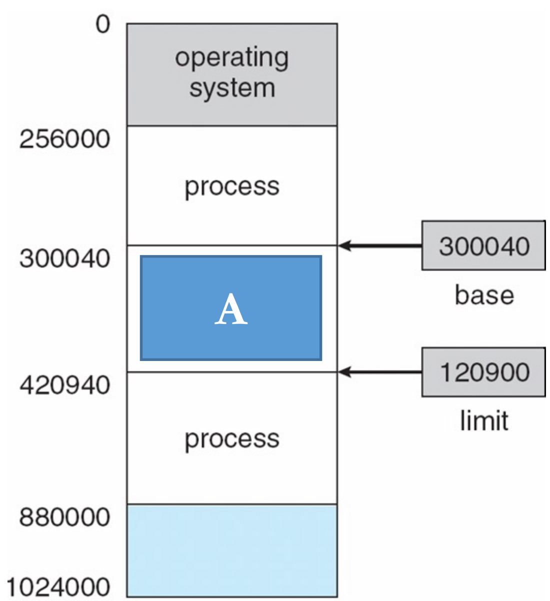
Logical/Virtual addresses [0,max]

Physical addresses [R+0,R+max]

Agenda

- Introduction
- Base and limit registers
- Logical/Virtual vs Physical addresses
- Dynamic loading and linking optimizations [self reading]
- Swapping
- Memory allocation
  - Contiguous allocation
  - Segmentation
  - Paging

Optimizations – Dynamic loading



We assumed so far the entire program A is loaded

If A is large and parts of it are almost never used → slower to load

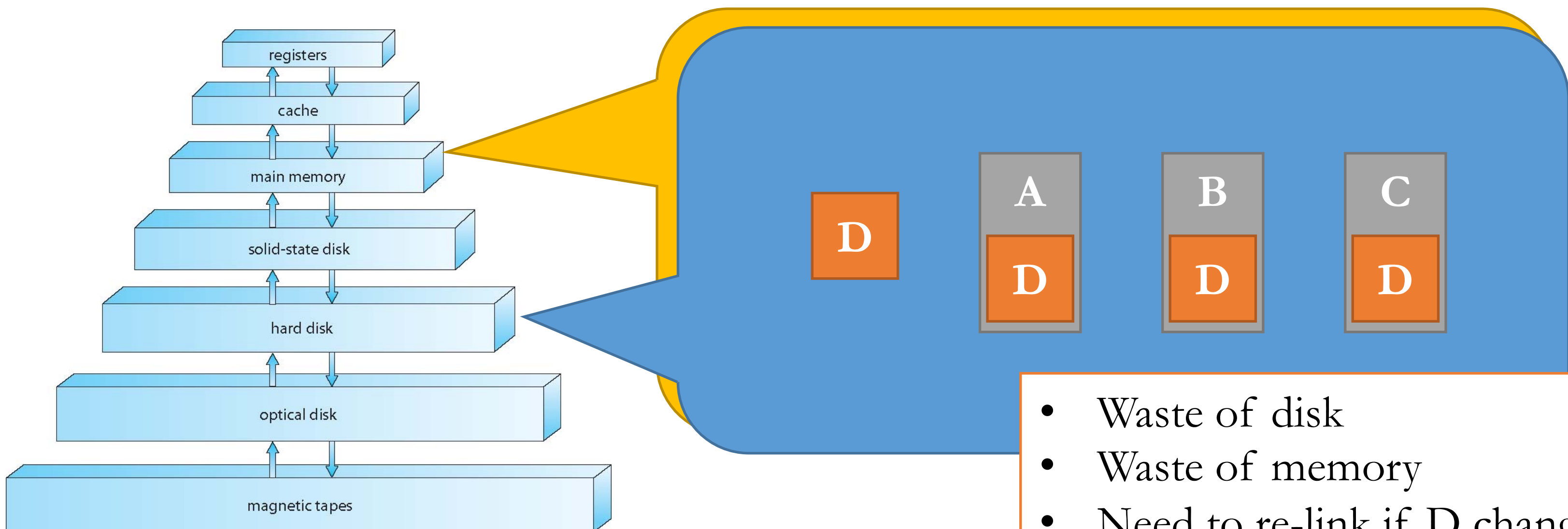
With dynamic loading, routines of a program can be loaded only when needed

Optimizations – Dynamic Linking (i)

Use library from

Program A
Program B → **Program D**
Program C

Static linking



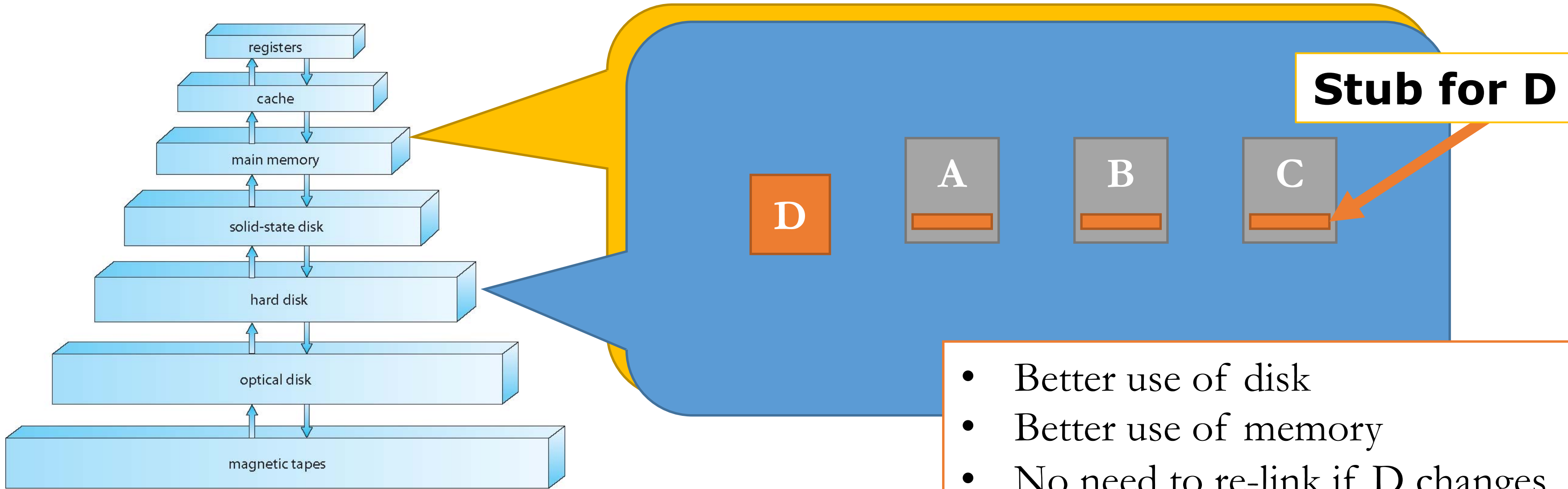
- Waste of disk
- Waste of memory
- Need to re-link if D changes

Optimizations – Dynamic Linking (ii)

Use library from

Program A → **Program D**
Program B → **Program D**
Program C → **Program D**

Dynamic linking

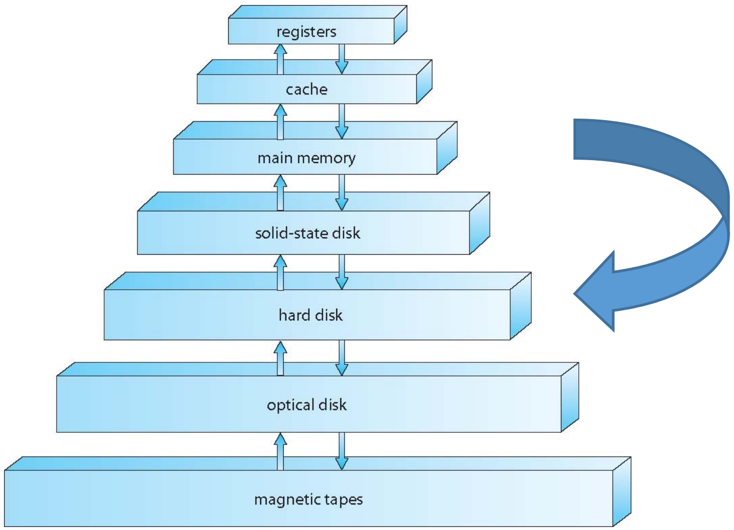


- Better use of disk
- Better use of memory
- No need to re-link if D changes

Agenda

- Introduction
- Base and limit registers
- Logical/Virtual vs Physical addresses
- Dynamic loading and linking optimizations [self reading]
- Swapping
- Memory allocation
  - Contiguous allocation
  - Segmentation
  - Paging

Swapping (i)



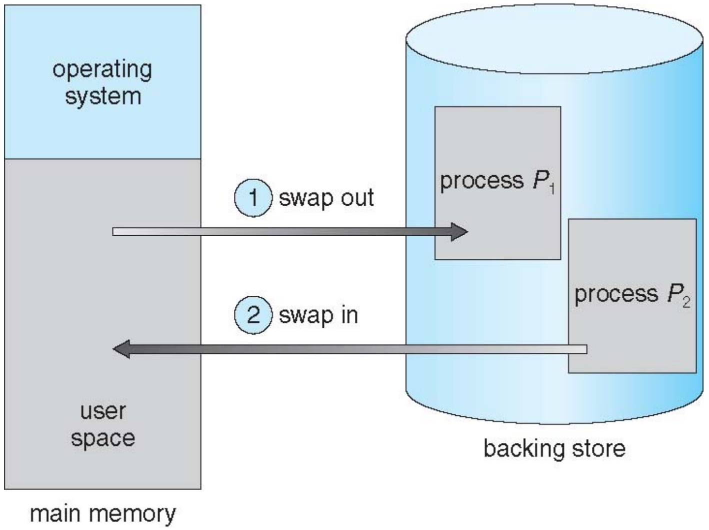
Moving a process from memory to backing store

Why?

size(all logical addresses) > size(real physical memory)

Swapping (ii)

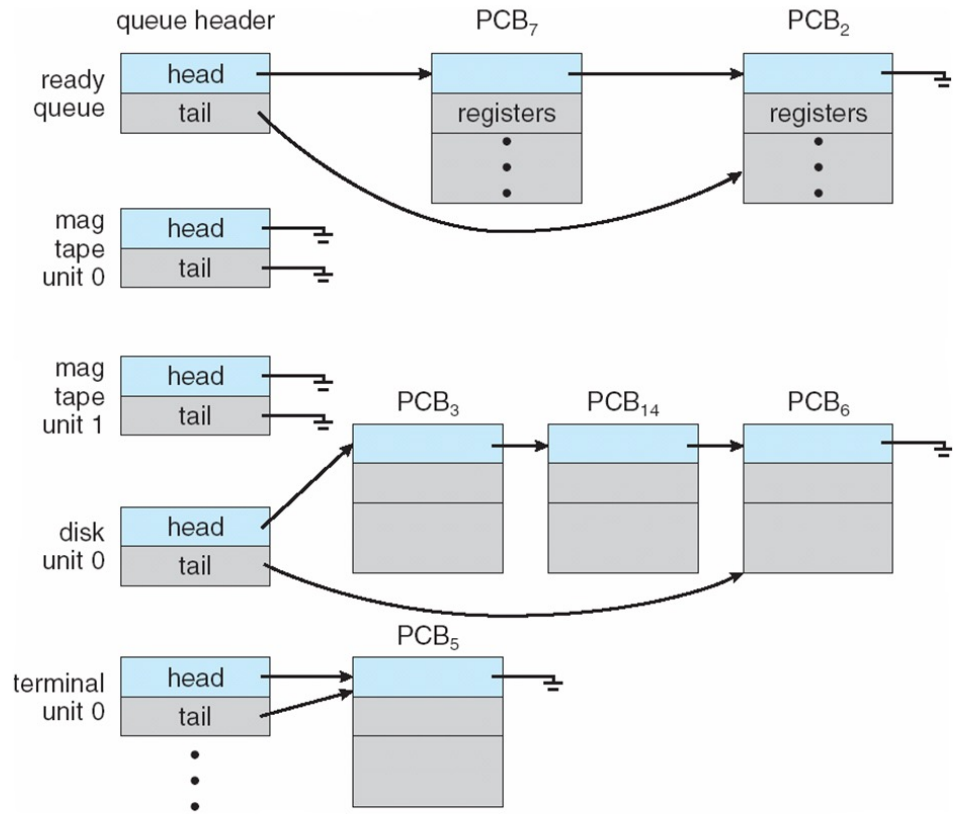
- Basic idea:



Questions:

- Which processes can we swap?
- How much time does it take?
- When does it make sense to use swapping?

Swapping - Which processes can we swap?



Depends on the pending I/O…

Can the I/O results be buffered at the OS?

Swapping – How much time does it take?

- Process in memory 100 MB
- Disk transfer rate 50 MB/s

→ 100 MP / 50 MB/s = 2 seconds

→ Swap out + Swap in = 4 seconds

(Notice: being optimistic! Disk writes/reads might be requested by other processes, interrupts will occur in the meantime, etc…)

It is slow and expensive (because write/read to disks are slower and more expensive than memory ones)!

Swapping – When does it make sense to use it?

- Not used "as it is"…

- Modified versions are used in UNIX, Linux and Windows
  - Swapping is activated when free memory goes below a certain threshold
  - Swapping is activated when free memory goes above a certain threshold

- Modified swapping (swap out / in only parts of the process) used in conjuction with Virtual Memory (next lecture…)

Agenda

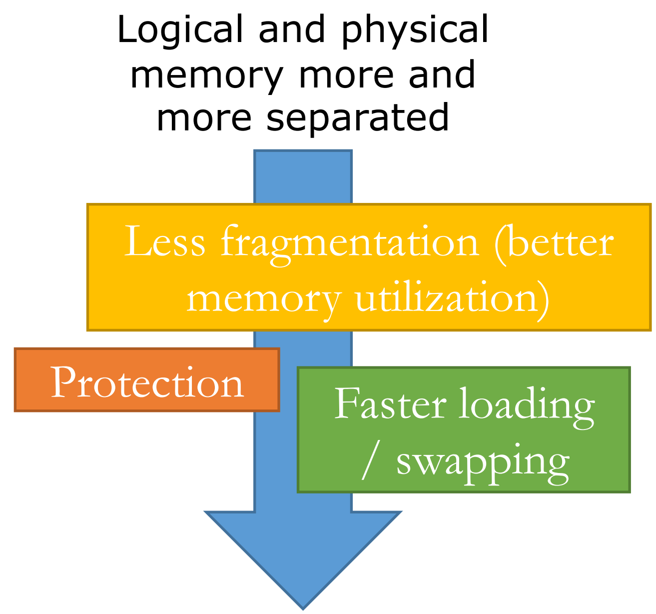
- Introduction
- Base and limit registers
- Logical/Virtual vs Physical addresses
- Dynamic loading and linking optimizations [self reading]
- Swapping
- Memory allocation
  - Contiguous allocation
  - Segmentation
  - Paging

Memory allocation

- Memory contains
  - The operating system itself
  - The users' processes

- Challenge: how to allocate memory efficiently?

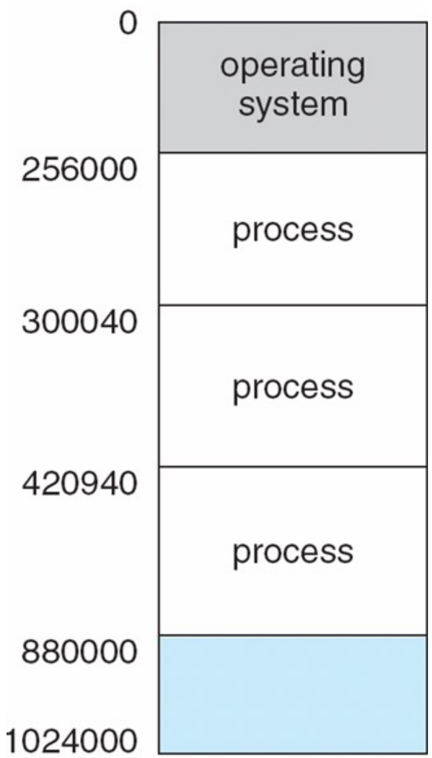
- We will see 3 cases
  - Contiguous allocation
  - Segmentation
  - Paging



Agenda

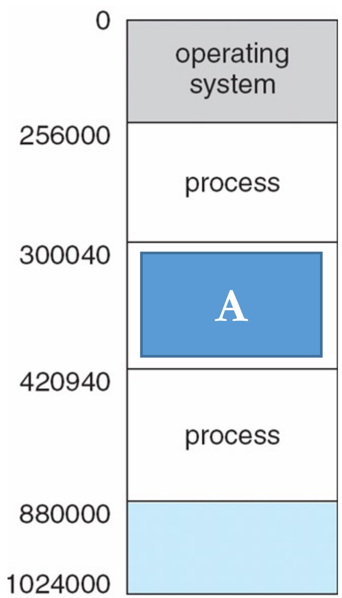
- Introduction
- Base and limit registers
- Logical/Virtual vs Physical addresses
- Dynamic loading and linking optimizations [self reading]
- Swapping
- Memory allocation
  - Contiguous allocation
  - Segmentation
  - Paging

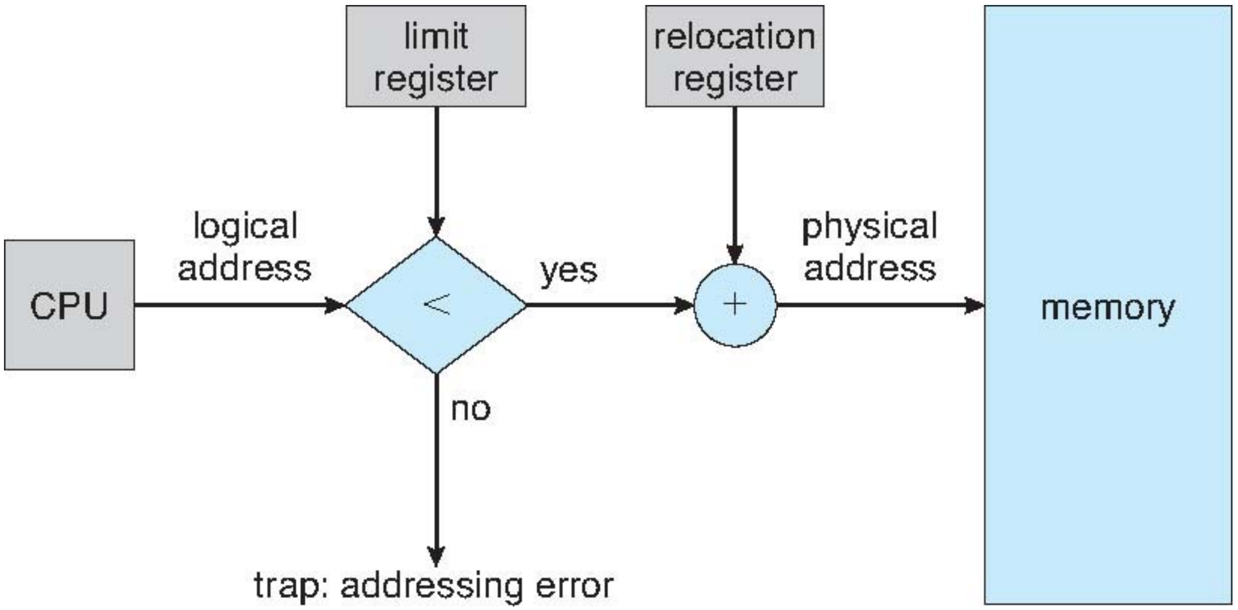
Contiguous allocation



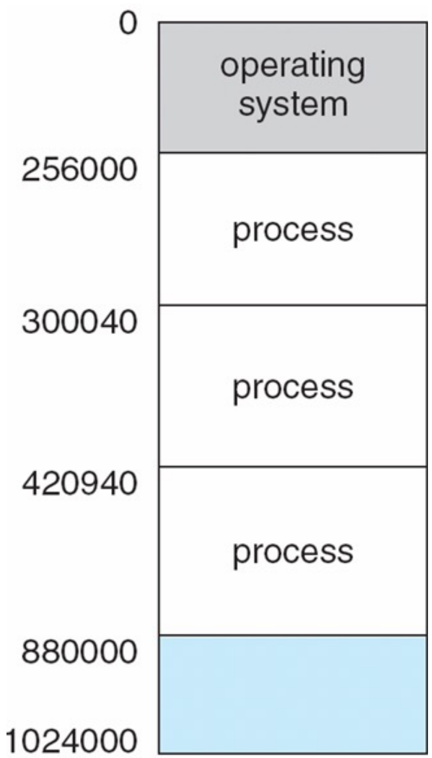
- Operating system at the bottom
- Processes one after the other
  (each process contiguous to the previous)

Contiguous allocation – memory protection





Contiguous allocation - limitations



- When we terminate (or swap) a process
  - We create a **hole**
- The OS needs to keep track of holes to reuse them later on for other processes
- External fragmentation

sum(all holes size) > size(a new process)
BUT
size(of any hole) < size(a new process)

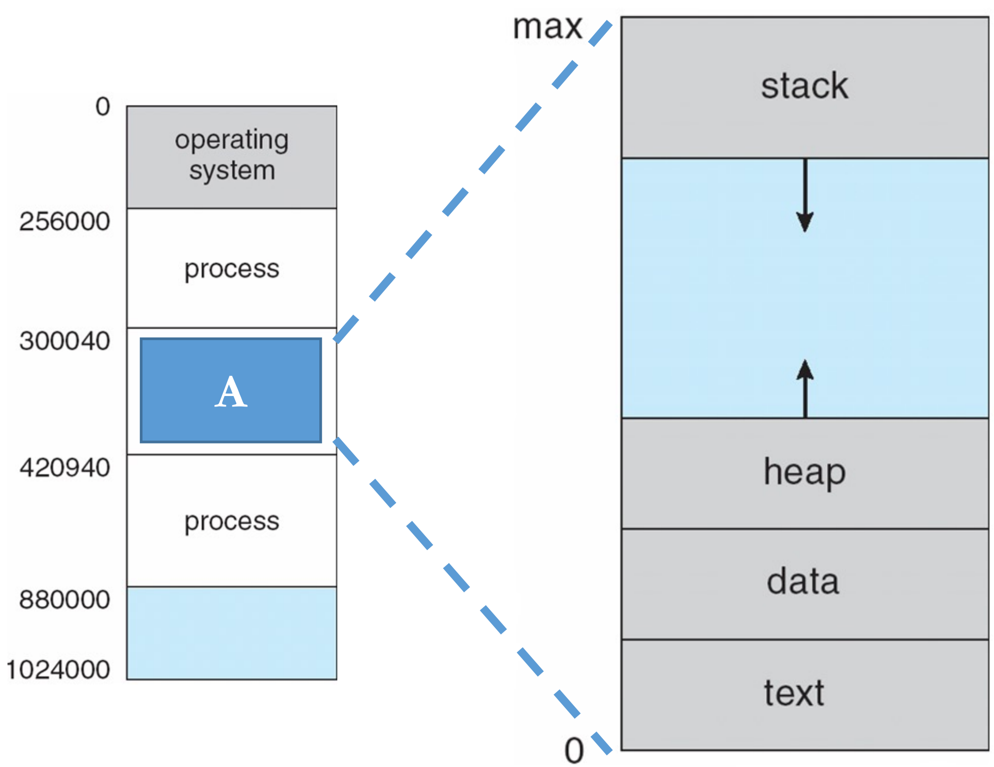
Contiguous allocation – which hole to assign to a new process?

- First fit → allocate the first hole that is big enough
- Best fit → allocate the smallest hole that is big enough
  - PROS: produces the smallest leftover hole
  - CONS: Search entire list (or keep list of holes sorted)
- Worst fit → allocate the largest hole
  - PROS: produces the largest leftover hole
  - CONS: Search entire list (or keep list of holes sorted)

Agenda

- Introduction
- Base and limit registers
- Logical/Virtual vs Physical addresses
- Dynamic loading and linking optimizations [self reading]
- Swapping
- Memory allocation
  - Contiguous allocation
  - Segmentation
  - Paging

Segmentation (i)

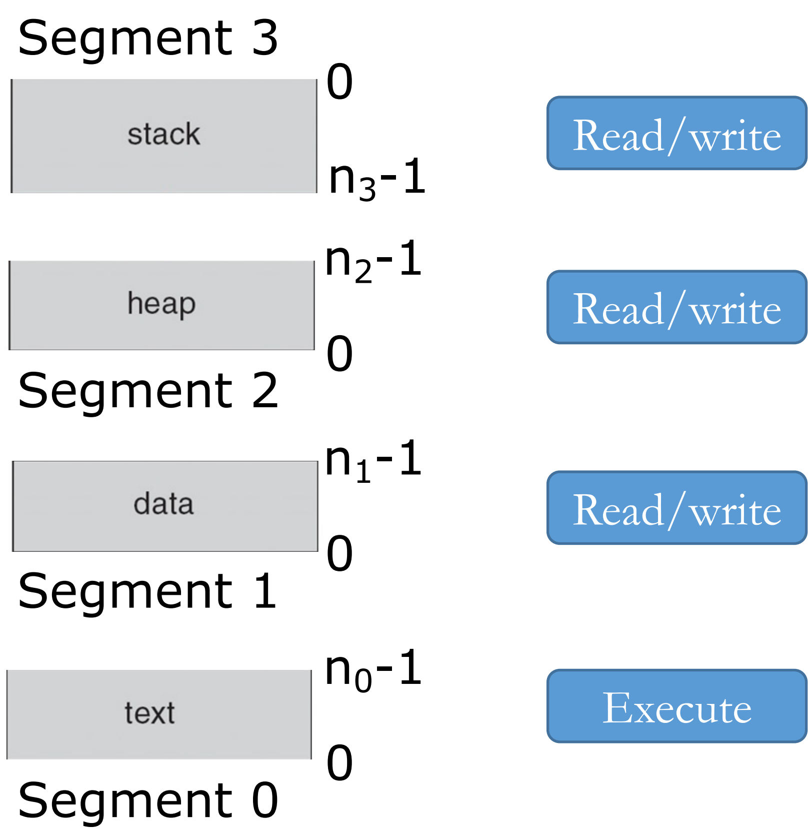


- So far we referred to the memory of a process as a single unidimensional space [0,max]

- Nevertheless, from the programmer/program perspective:
    - Different parts have different meaning, and each should be "protected from others"
    - Different parts have different and variable size

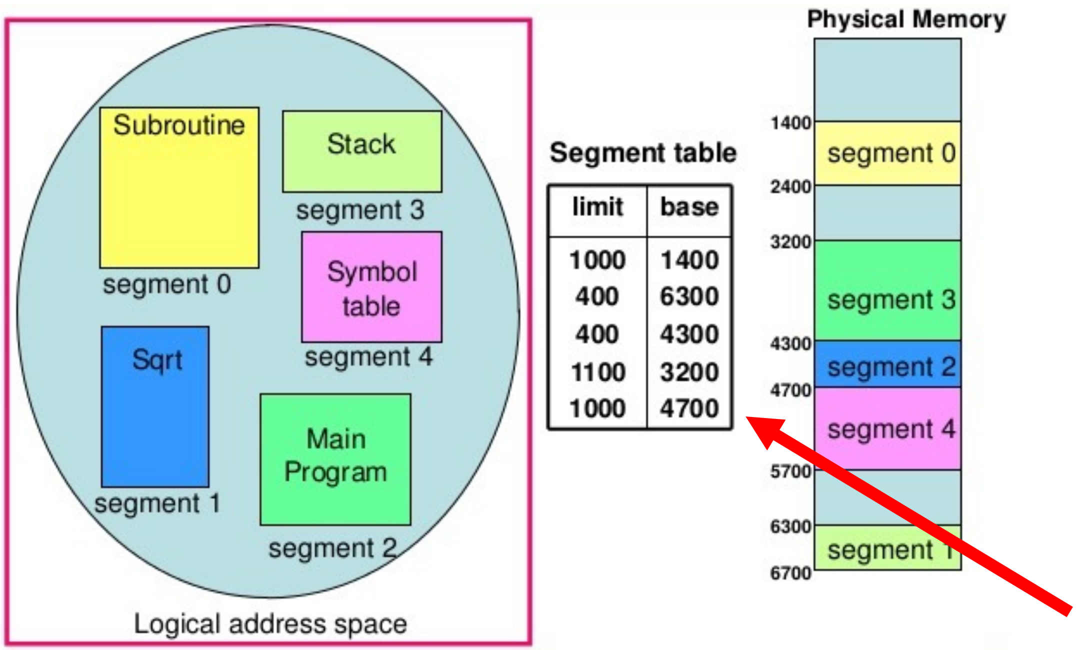
Segmentation (ii)

- With segmentation a memory position can be accessed not referring to its global position in [0,max] but rather as a pair <segment, position in the segment>



- Easier to check if an access to a certain section is actually reaching a different section
- Allows to have different permissions for the different segments!
- Different parts of the process can be at different locations in the physical memory!

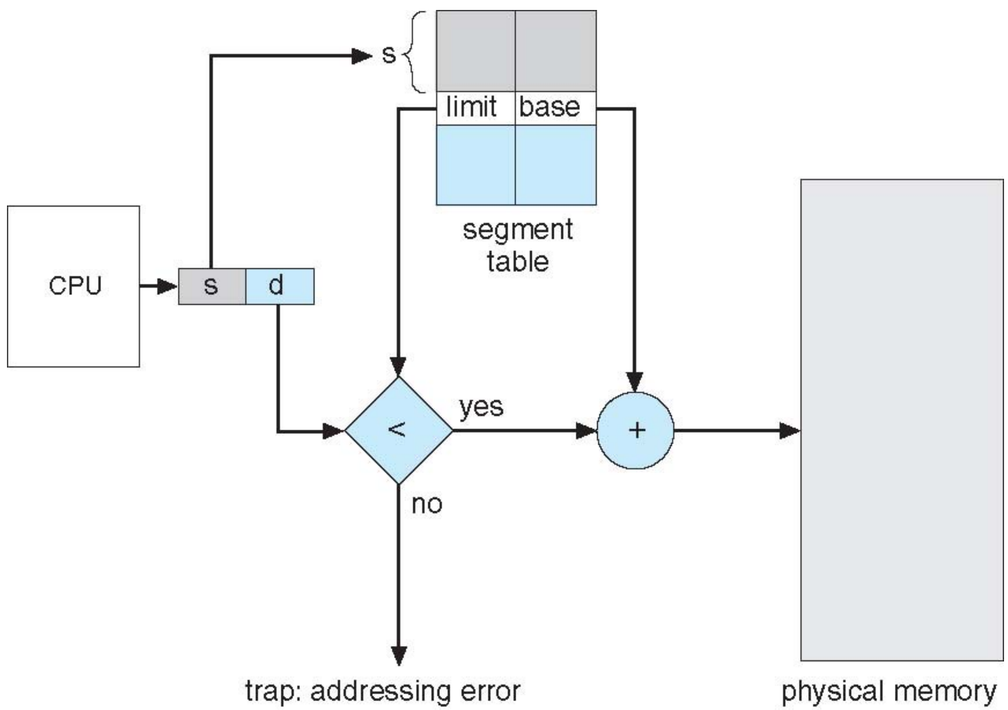
Segmentation – segment table



Need to maintain extra information to link logical address space / segments to the physical memory

Segmentation – hardware support

Hardware is able to do this conversion automatically (and check whether the process is trying to access memory outside the segment)



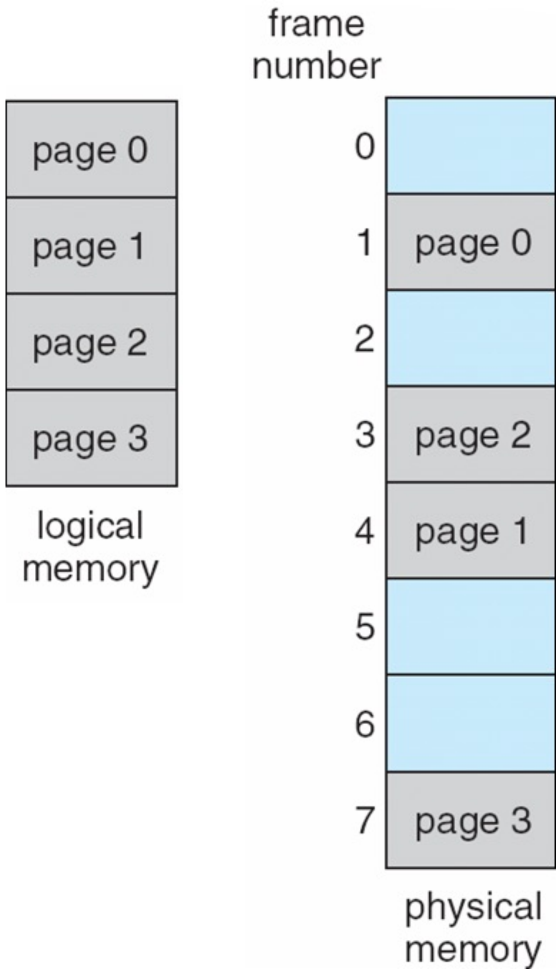
Agenda

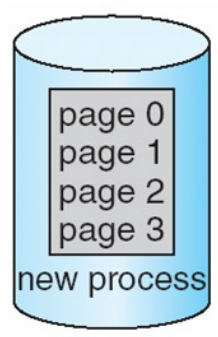
- Introduction
- Base and limit registers
- Logical/Virtual vs Physical addresses
- Dynamic loading and linking optimizations [self reading]
- Swapping
- Memory allocation
  - Contiguous allocation
  - Segmentation
  - Paging

Paging

- Mechanism that allows for the physical address space of a process to be non-contiguous
- Used in most modern OSes (from mainframes to smartphones)
- Requires cooperation between OS and hardware (as common)

Paging – Basic idea



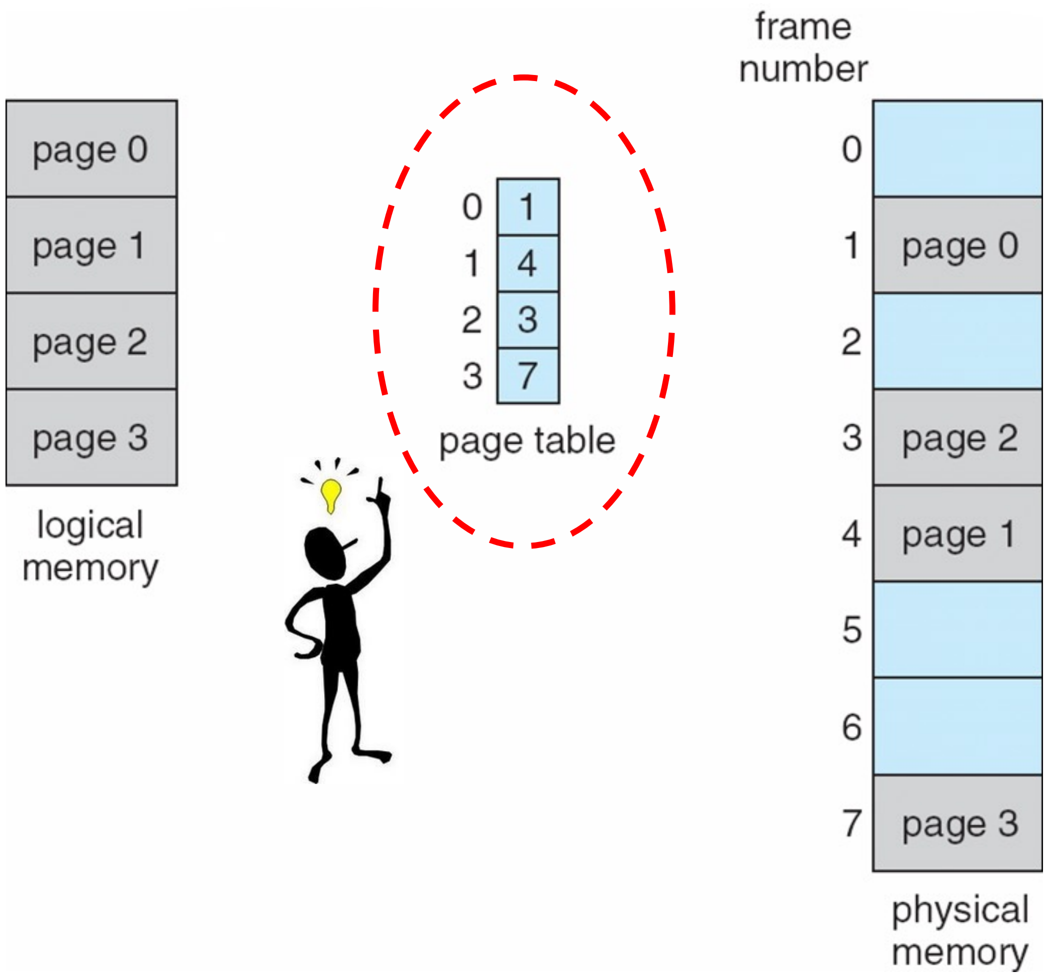


- Split logical memory in pages
- Split physical memory in frames
- Keep a program (disk) organized in pages

<u>Page size = Frame size!</u>

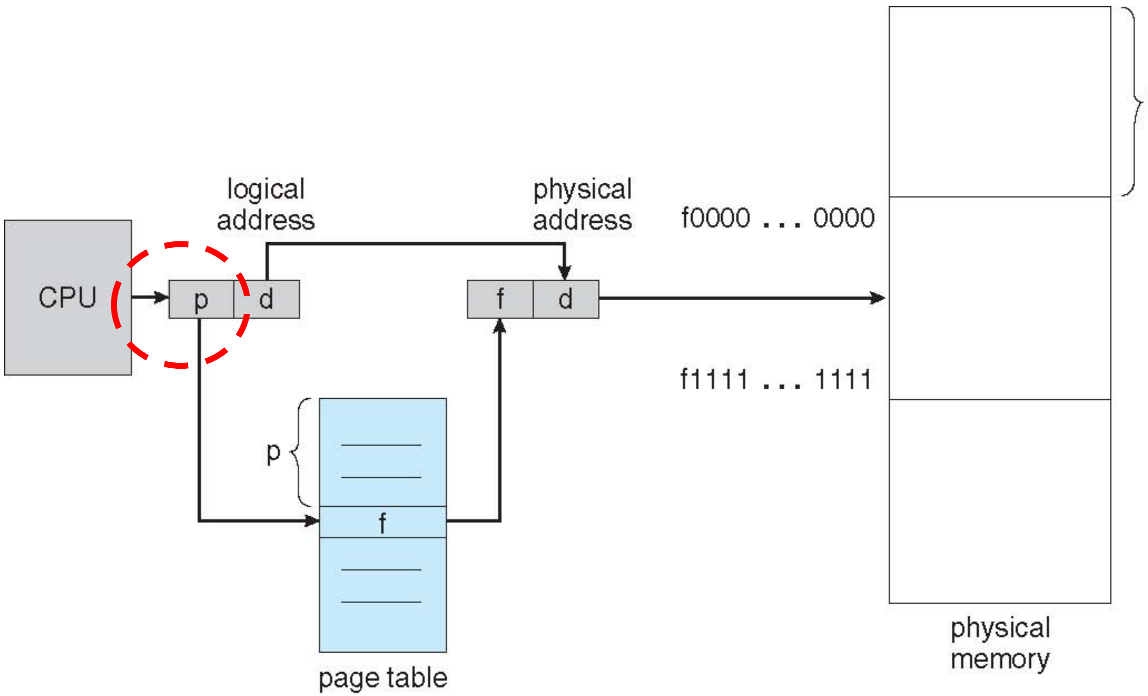
How do we keep track of where each page is?

Paging – the page table



- Page 0? Frame 1
- Page 1? Frame 4
- Page 2? Frame 3
- Page 3? Frame 7

Paging – hardware support



How does the CPU know which page it should access?

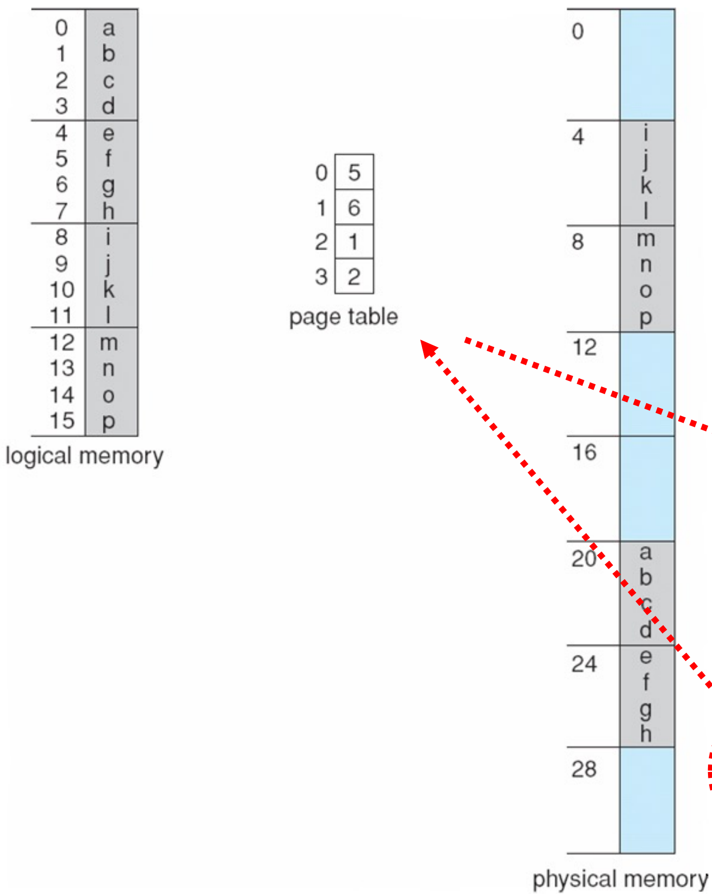
Similarly to Segmentation, we can access physical memory specifying page + offset

Paging – from logical address to physical address (i)

- If page/frame size is power of 2 → 2^n
- If size of logical space is power of 2 → 2^m

Logical address (binary) $\underbrace{10110100}_{\substack{m-n \\ \text{page}}}\underbrace{1110110}_{\substack{n \\ \text{offset}}}$

Paging – from logical address to physical address (ii)



Page size = 2^2 bytes (n=2)

Size of logical space = 2^4 bytes (m=4)

→ logical space = 4 pages

Physical address of logical address 7?

$\underbrace{0\ 1}_{\text{page } 1}\ \underbrace{1\ 1}_{\text{offset } 3}$

Frame 6 offset 3

→ 6x4+3=27

Paging - Fragmentation

- No external fragmentation
  - Process needs a page? Can take any free frame
- Internal fragmentation
  - Last frame assigned to a process' page might be not full
  - On average, half frame is wasted for each process

- Small pages → less fragmentation / more overhead (e.g., page table size)
- Large pages → more fragmentation / less overhead