Linking and virtual memory

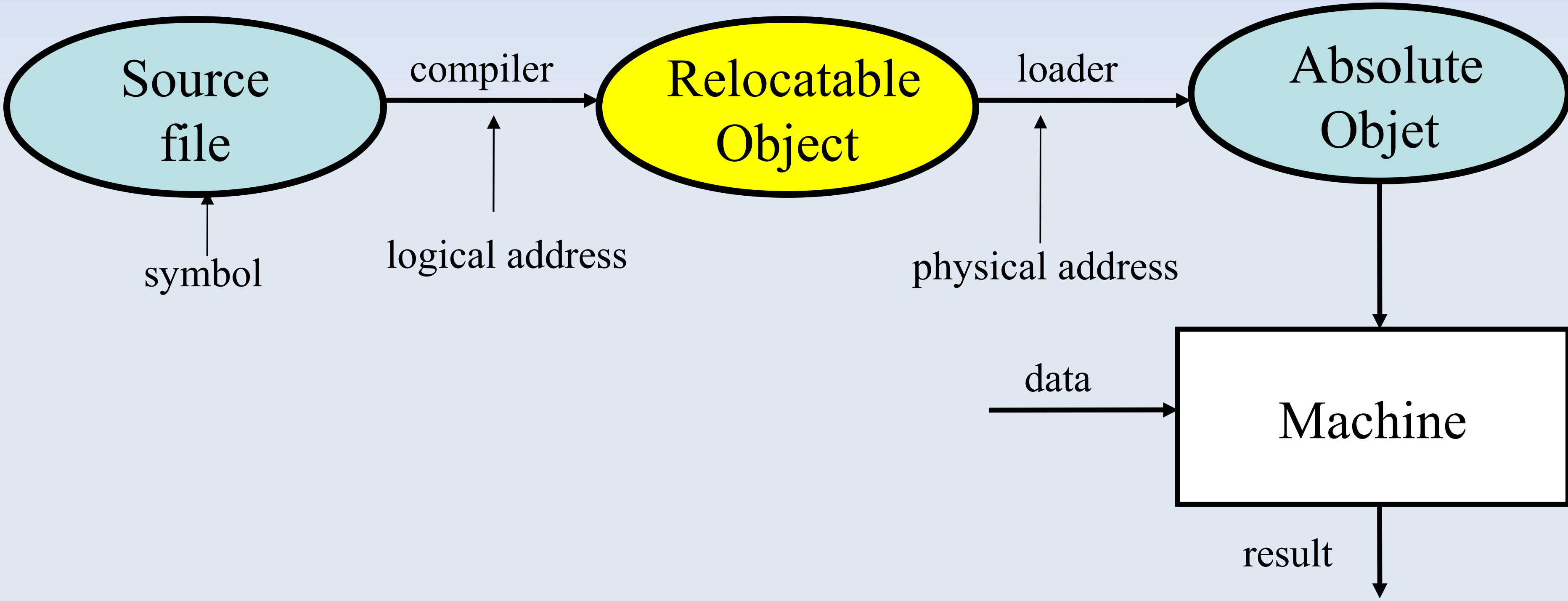
Noël De Palma
UJF

Thanks to Fabienne Boyer and Arnaud Legrand

Introduction

- Memory is a ressource required by all processes
    - Every program need to be **loaded** in memory to be running
- Problem
    - Address translation
        - Symbol → Logical address → physical address
    - Memory allocation and exhaustion
    - Memory sharing
    - Memory protection

Life cycle of a single program



Source file

compiler

Relocatable Object

loader

Absolute Objet

symbol

logical address

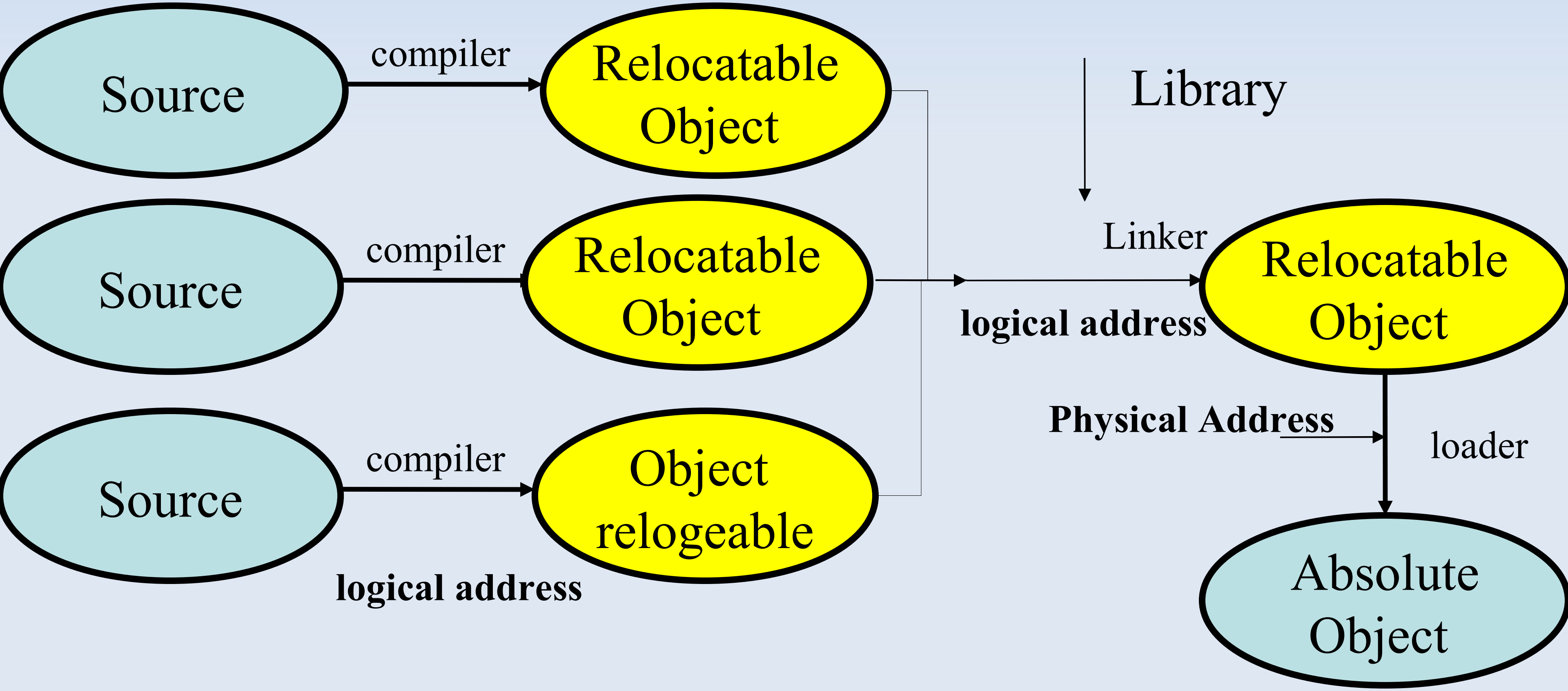
physical address

data

Machine

result

Lifecycle of a program assembled from multiple part



Source

compiler

Relocatable Object

Source

compiler

Relocatable Object

Source

compiler

Object relogeable

logical address

Library

Linker

logical address

Relocatable Object

Physical Address

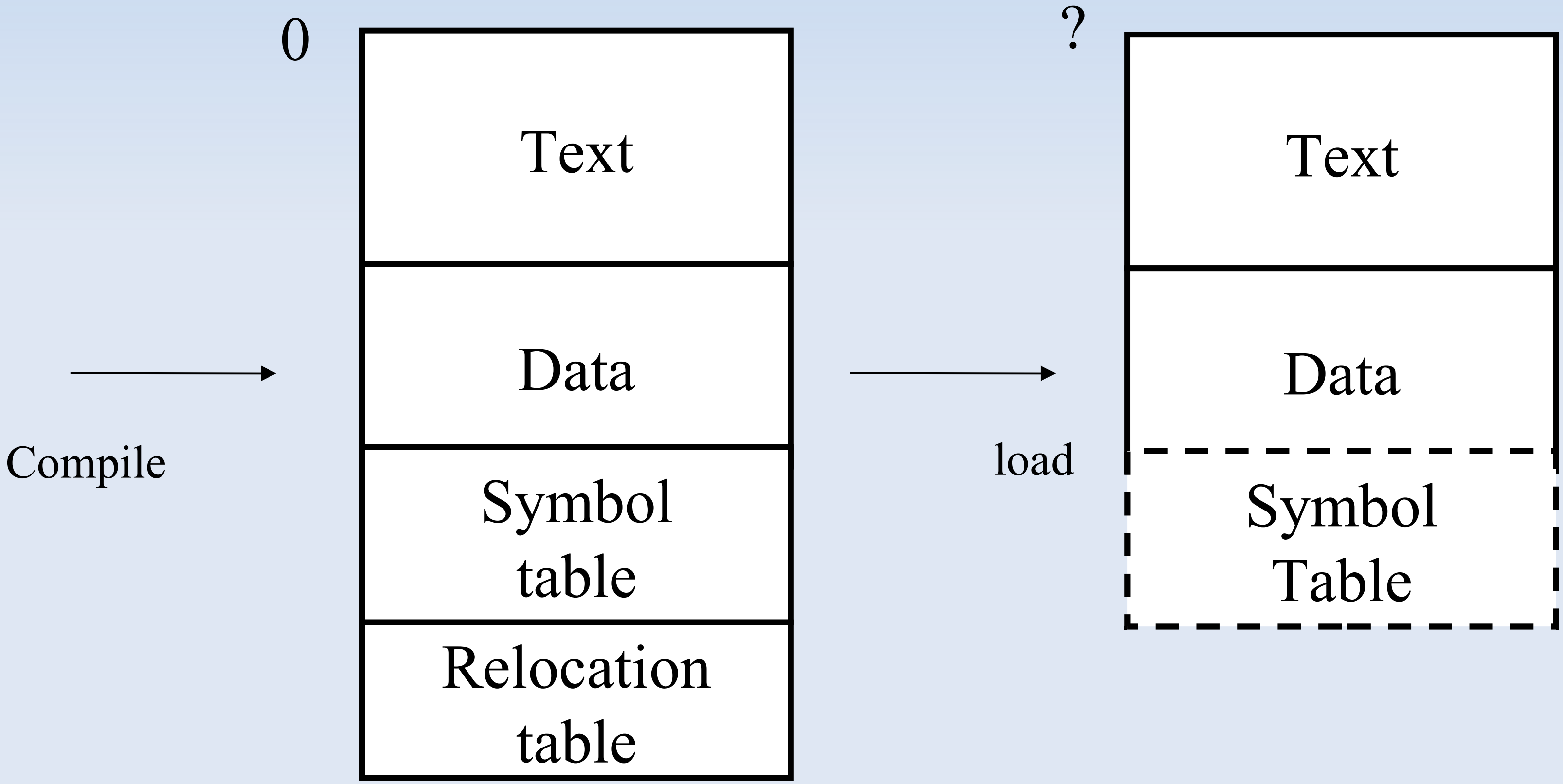
loader

Absolute Object

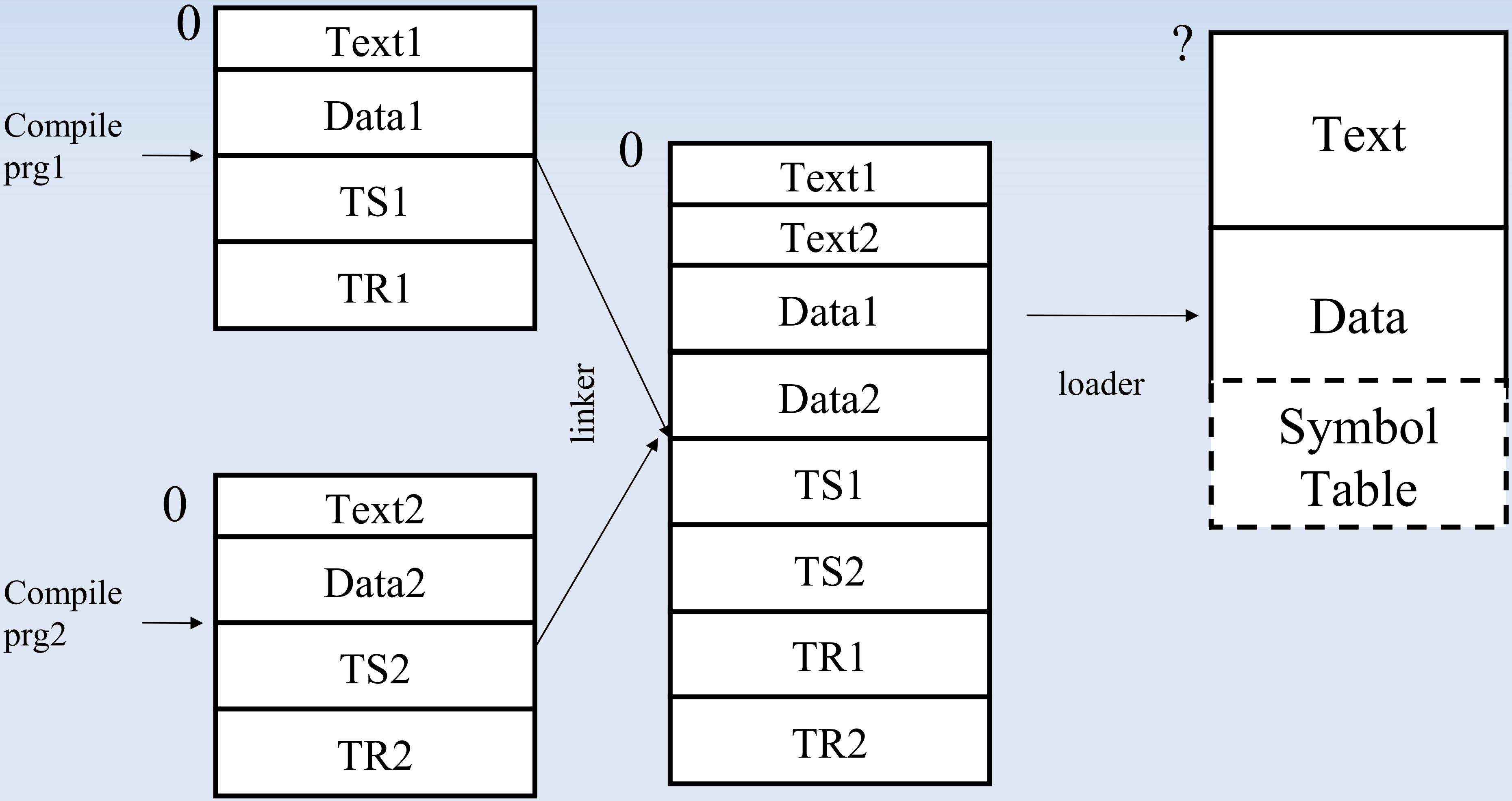
Load-time translation

- Translation between logical and physical adresses
    - Determine where process will reside in memory
    - Translate all references within program
    - Established once for all
- Monoprogramming
    - One program in memory
    - Easy
- Multiprogramming
    - N program in memory
    - Compiler and linker do not know the implantation of processes in memory
    - Need to track op-code that must be updated

Simple program binary structure



Complex program binary structure



Data structure

Symbol table

Section	Relative @	Symbol name
	undef	

Relocation table (track the addresses that must be updated in the code)

Hole Addresse	Symbol Section	Symbol name

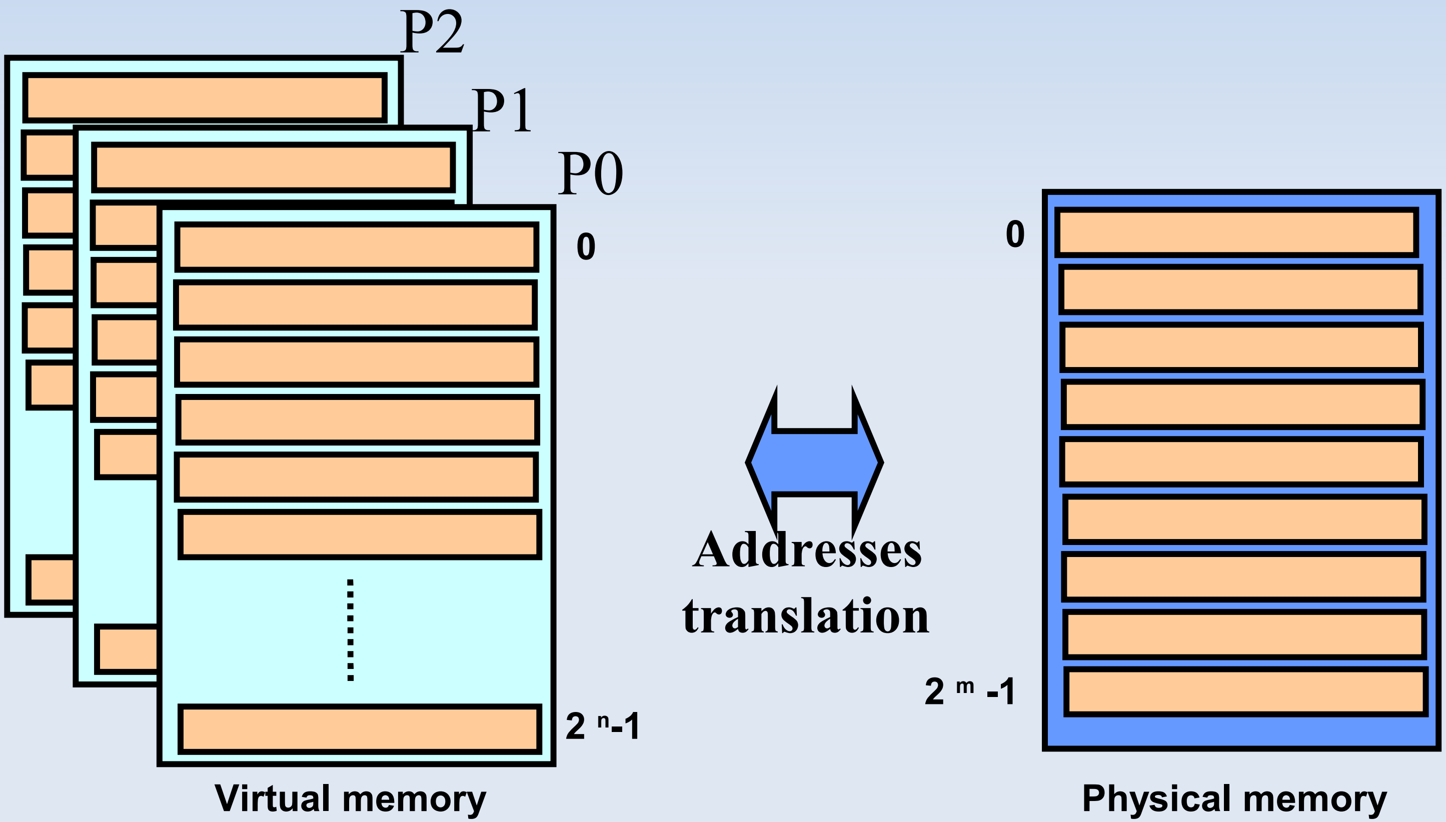
Load-time translation summary

- Remaining problems
    - How to to enforce protection ?
    - How to move program once in memory ?
    - What if no contiguous free region fits programs
    - Can we separate linking from memory management problems ?

Virtual memory

- Separate linking problem from memory management
- Give each program its own virtual address space
  - Linker works on virtual addresses
  - Virtual address translation done at runtime
    - Relocate each load/store to its physical address
    - Require specific hardware (MMU)

Virtual memory



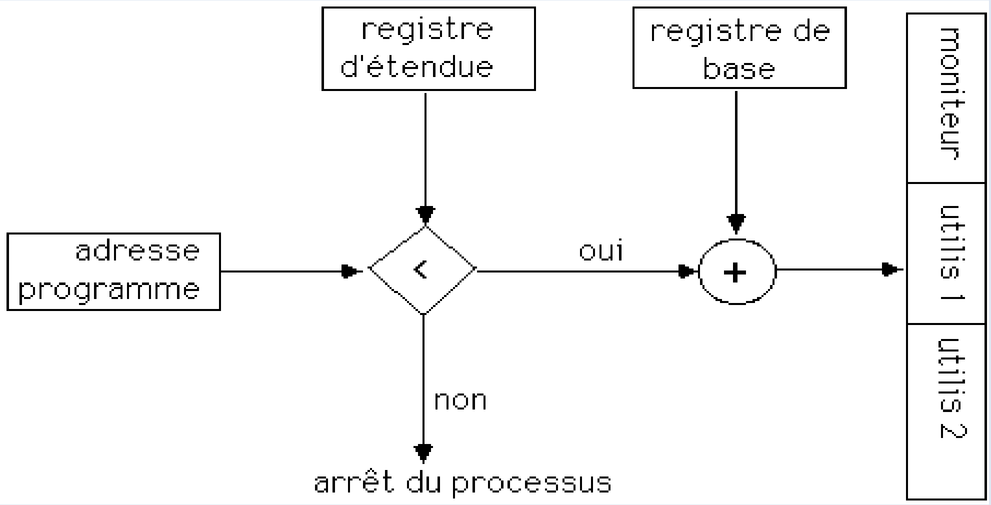
Ideally we want to enable n > m and non contiguous allocation

Virtual memory expected benefits

- Programs can be relocated while running
  - Ease swap in/swap out
- Enforce protection
  - Prevent one app from messing with another's memory
- Programs can see more memory than exist
  - Most of a process's memory will be idle
  - Write idle part to disk until needed

1st idea : Base + bound registers

- Contiguous allocation of variable size
- Two special privileged registers: base and bound
- On each load/store:
- Check 0 <= virtual address < bound, else trap to kernel
- Physical address = virtual address (plus) base



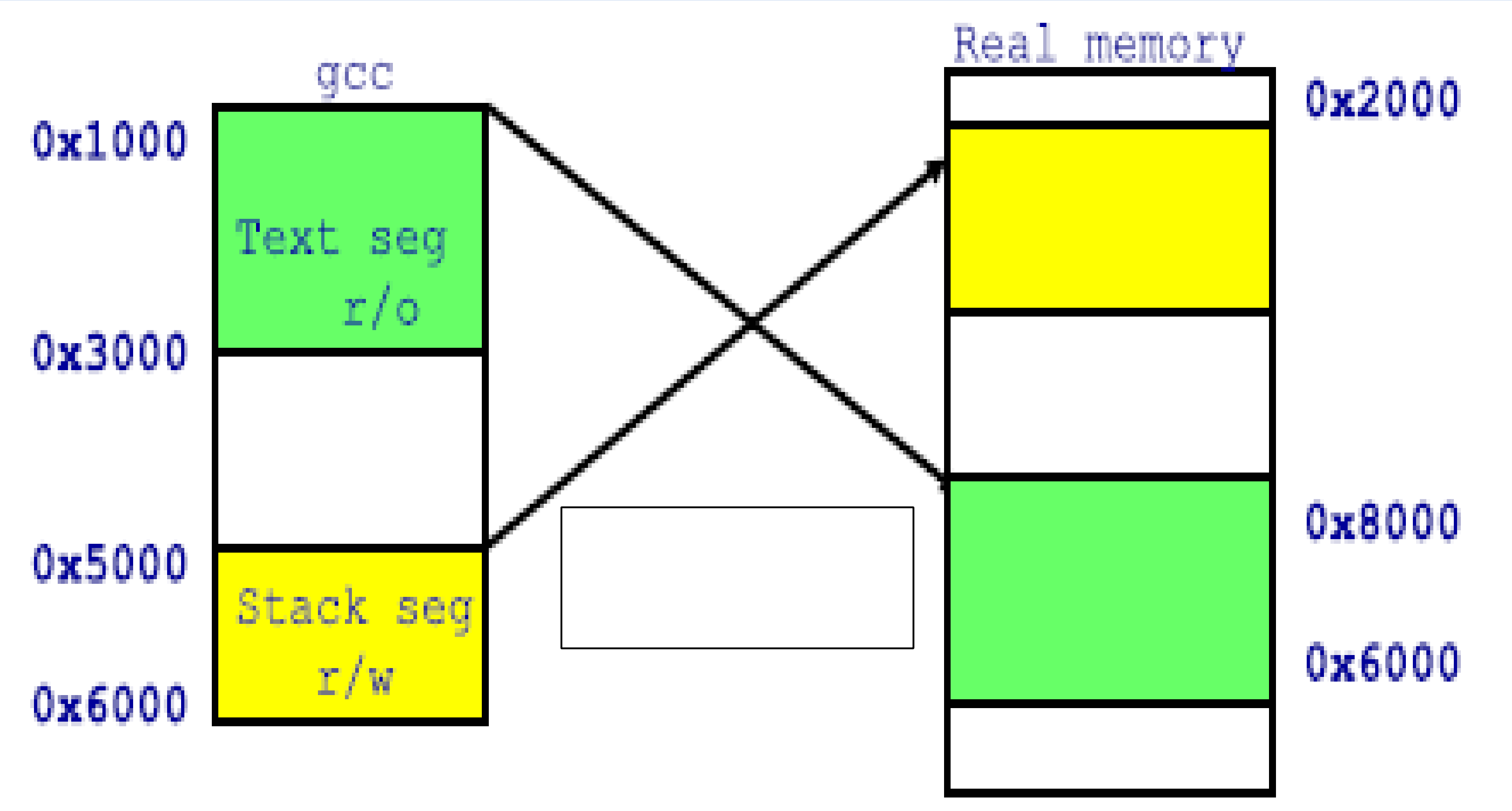
Base + bounds register

- Moving a process in memory
  - Change base register
- Context switch
  - OS must re-load base and bound register
- Advantages
  - Cheap in terms of hardware: only two registers
  - Cheap in terms of cycles: do add and compare in parallel
- Disadvantages
  - Still contiguous allocation
  - Growing a process is expensive or impossible
  - Hard to share code or data

Segmentation

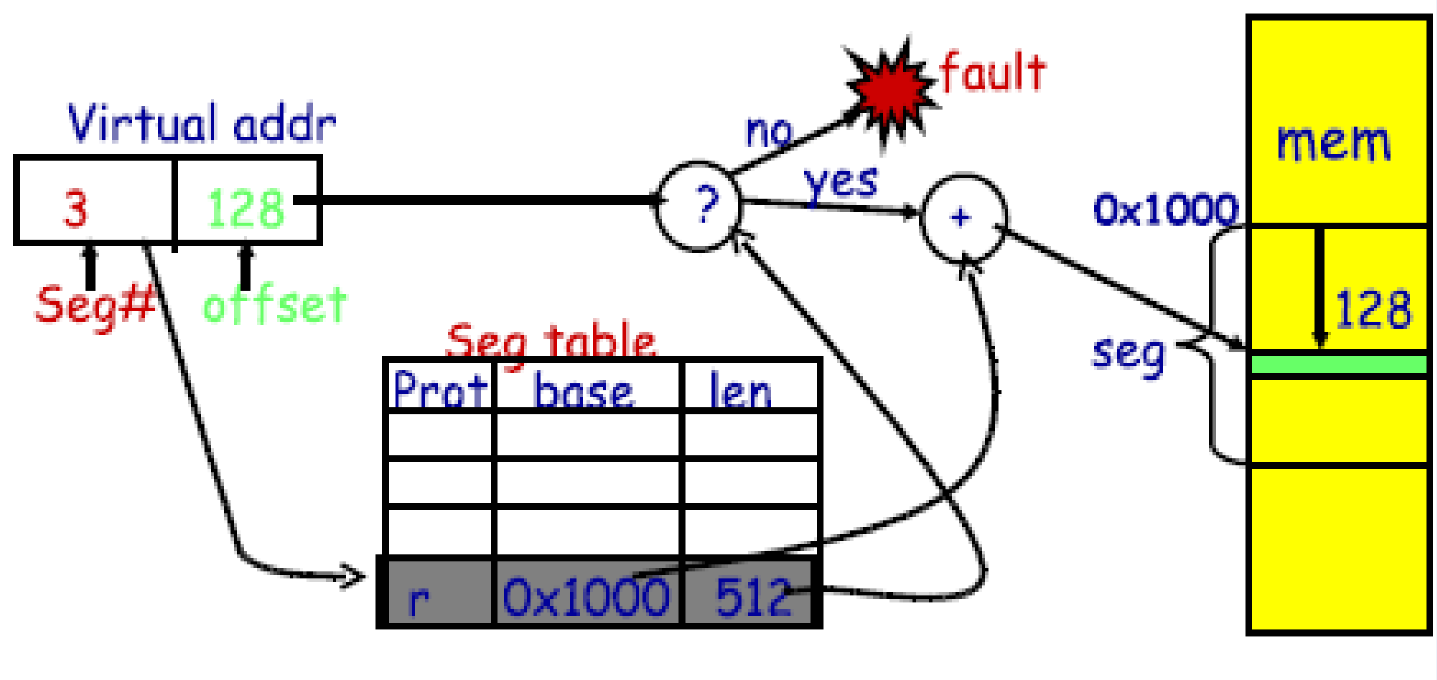
- Non contiguous allocation
  - Split a program in differents non contiguous segments of variable size
- Let processes have many base/bound regs
  - Address space built from many segments
  - Can share/protect memory on segment granularity
- Must specify segment as part of virtual address

Segmentation



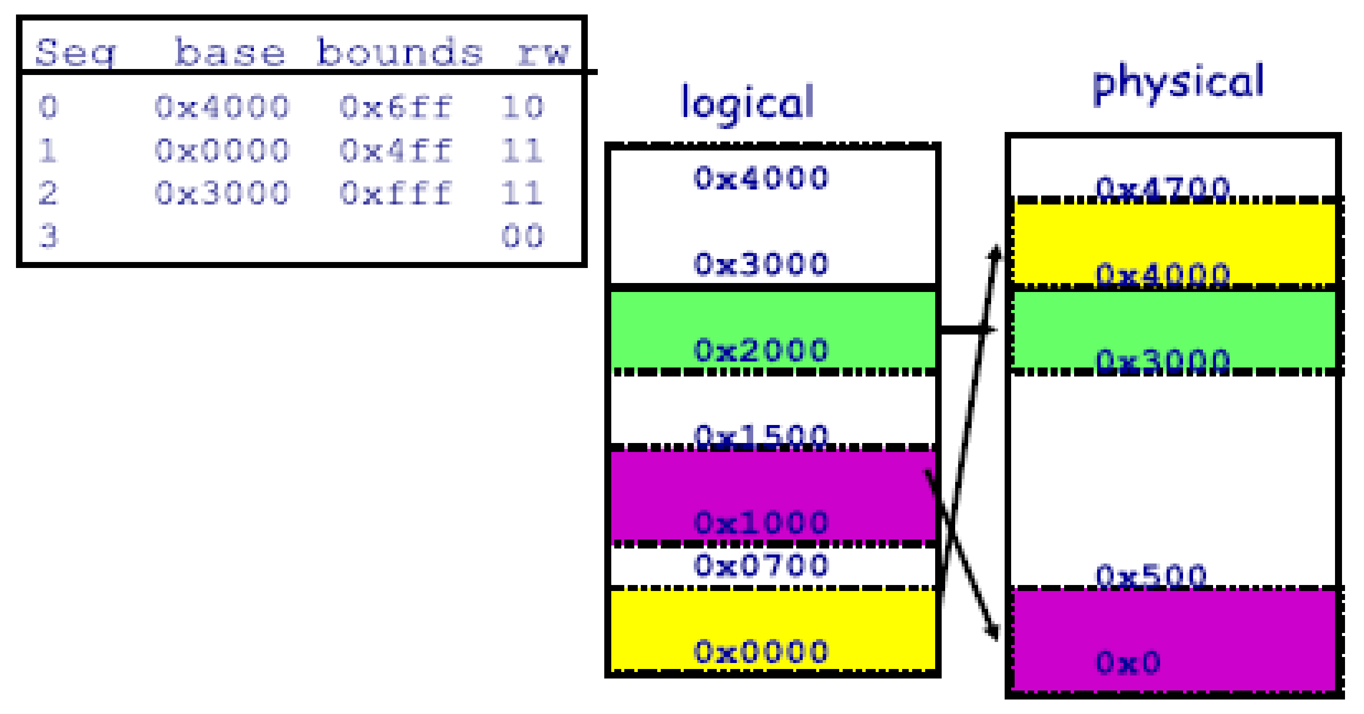
Segmentation mechanisms

- Each process has a segment table
  - Each VA indicates a segment and offset:
    - Top bits of addr select segment, low bits select offset



Segmentation example

- 4-bit segment number (1st digit), 12 bit offset (last 3)
  - Where is 0x0240? 0x1108? 0x265c? 0x3002? 0x1600?



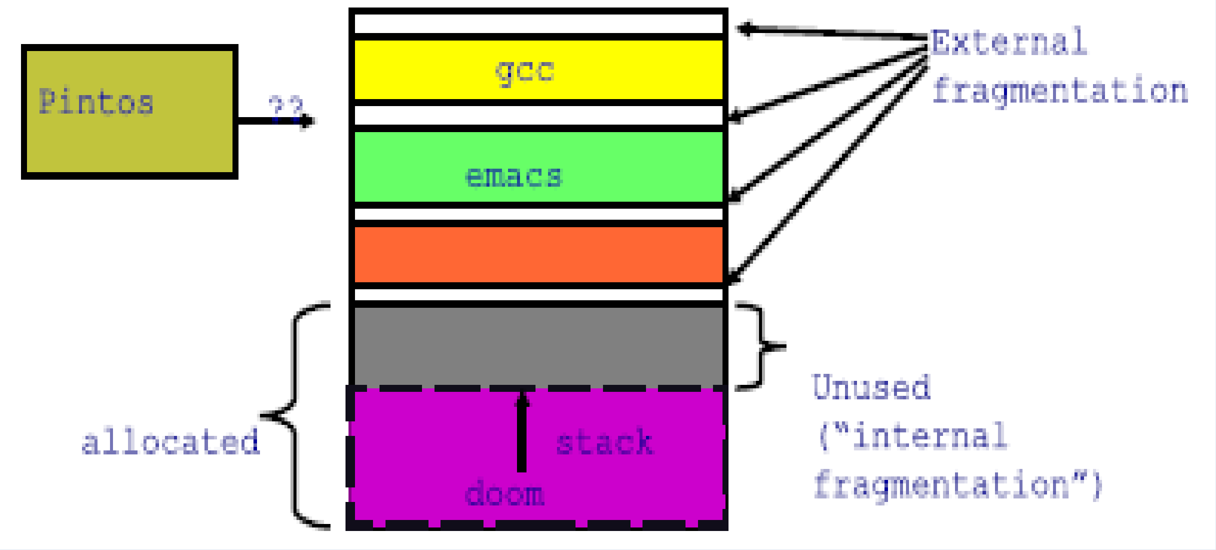
Seq	base	bounds	rw
0	0x4000	0x6ff	10
1	0x0000	0x4ff	11
2	0x3000	0xfff	11
3			00

Segmentation trade offs

- Advantages
  - Multiple segments per process
  - Allows sharing
- Disadvantages
  - N byte segment needs n contiguous bytes of physical memory
  - Fragmentation

Remember fragmentation problem

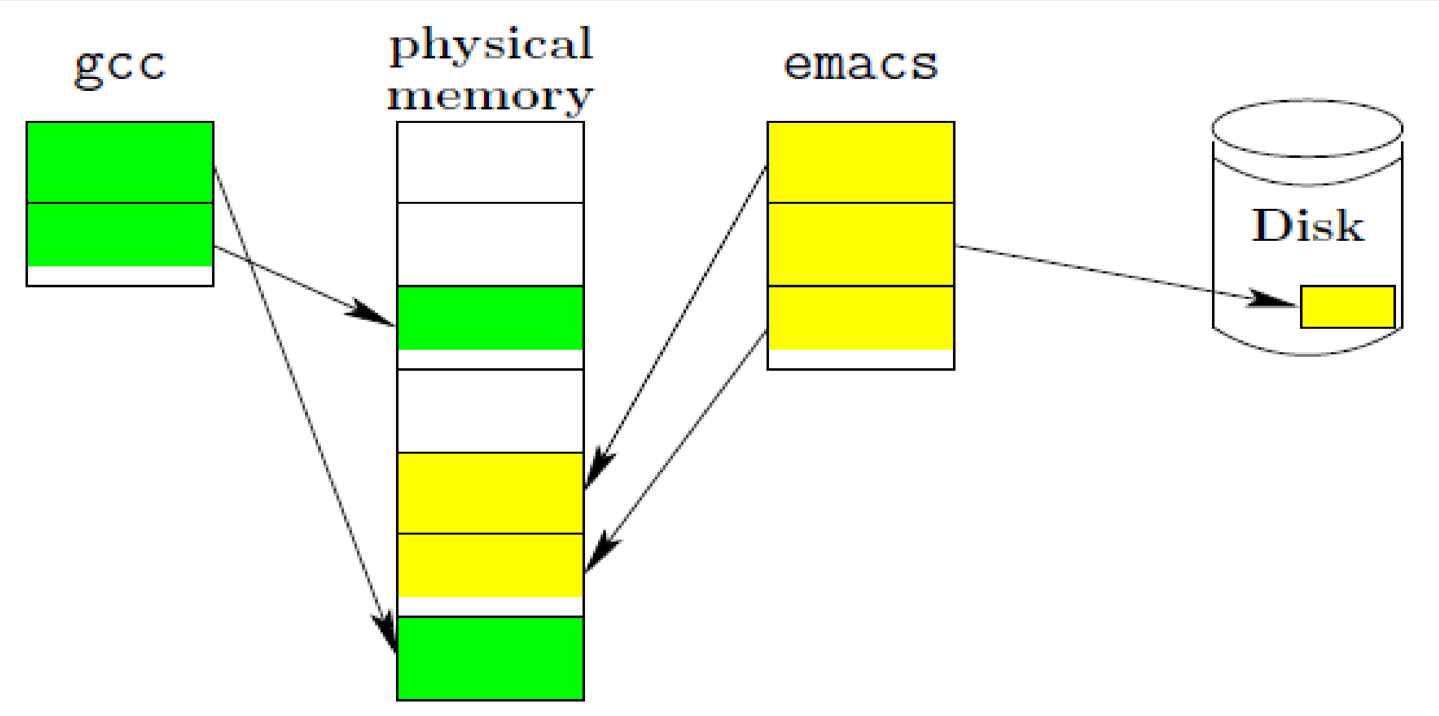
- Fragmentation => inability to use free memory
- Overtime:
  - Variable-sized pieces = many small holes (external fragmentation)



Paging

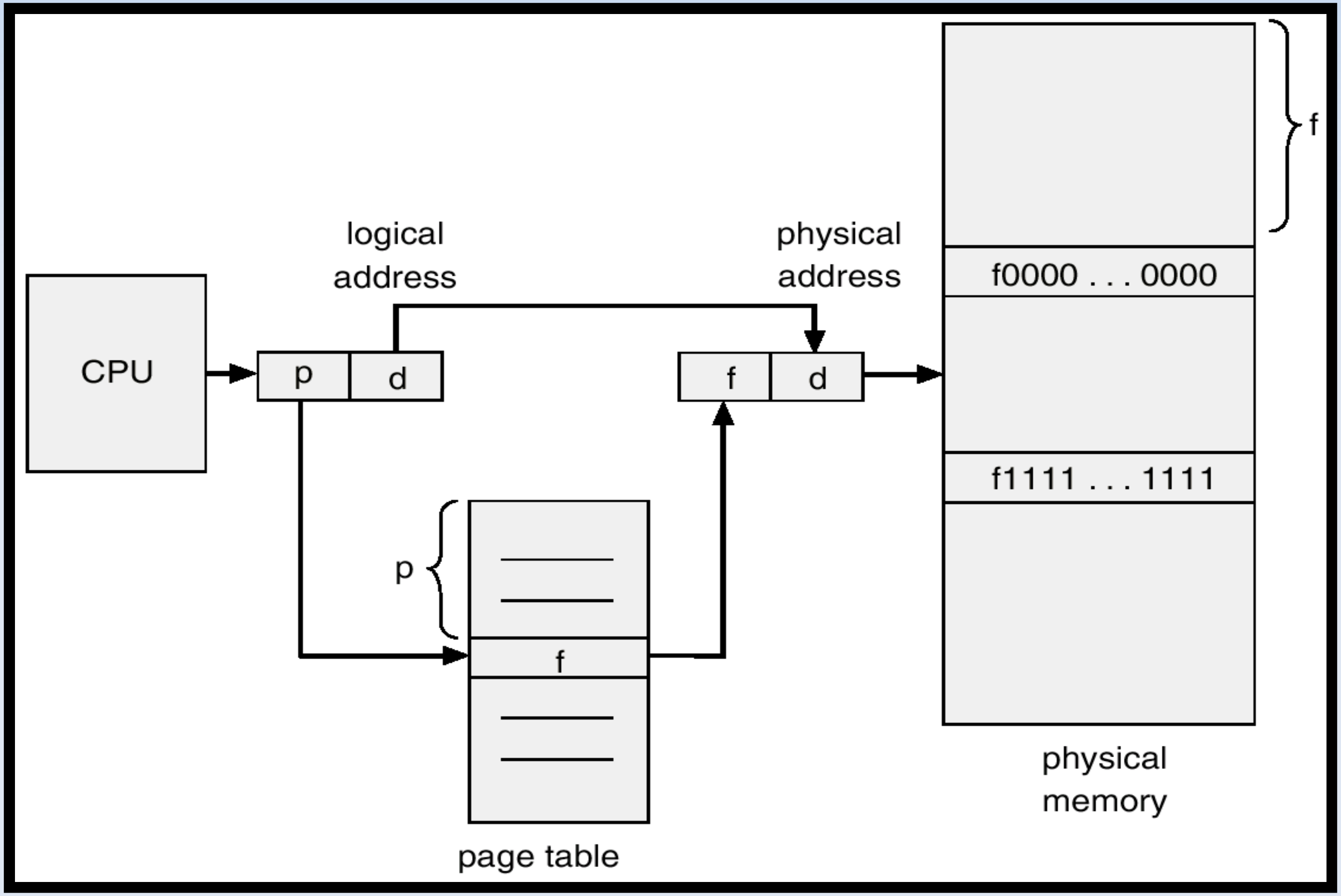
- Virtual memory is divided into small pages
  - Pages are fixed size
  - Page is contiguous
- Map virtual pages to physical block
  - Non contiguous allocation
  - Each process has a separate mapping
  - MMU
- OS gains control on certain operations
  - Read only pages trap to OS on write
  - OS can change the mapping

Paging



- Page table
  - Global or per process

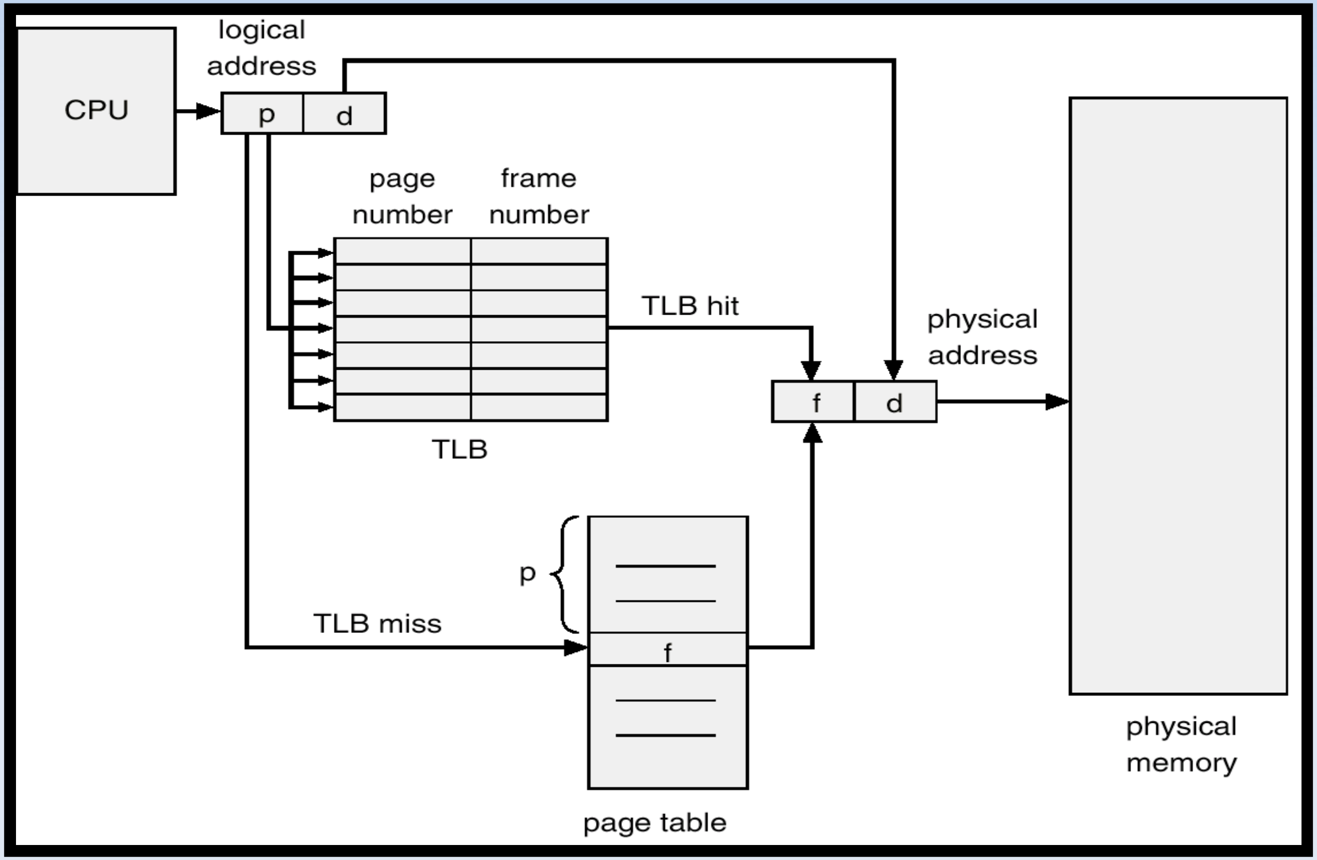
Virtual address translation



Problem : translation speed

- Require extra memory references on each load/store
  - Cache recently used translations
  - Locality principle
    - High probability that the next required address is close
- Translation Lookahead Buffer (TLB)
  - Fast (small) associative memory which can perform a parallel search
  - Typical TLB
    - Hit time : 1 clock cycle
    - Miss rate 1%
  - TLB management : hardware or software

TLB



- What to do when switch address space ?
    - Flush the TLB
    - Tag each entry with the process's id
- In general, OS must manually keep TLB valid
- Invalidates a page translation in TLB

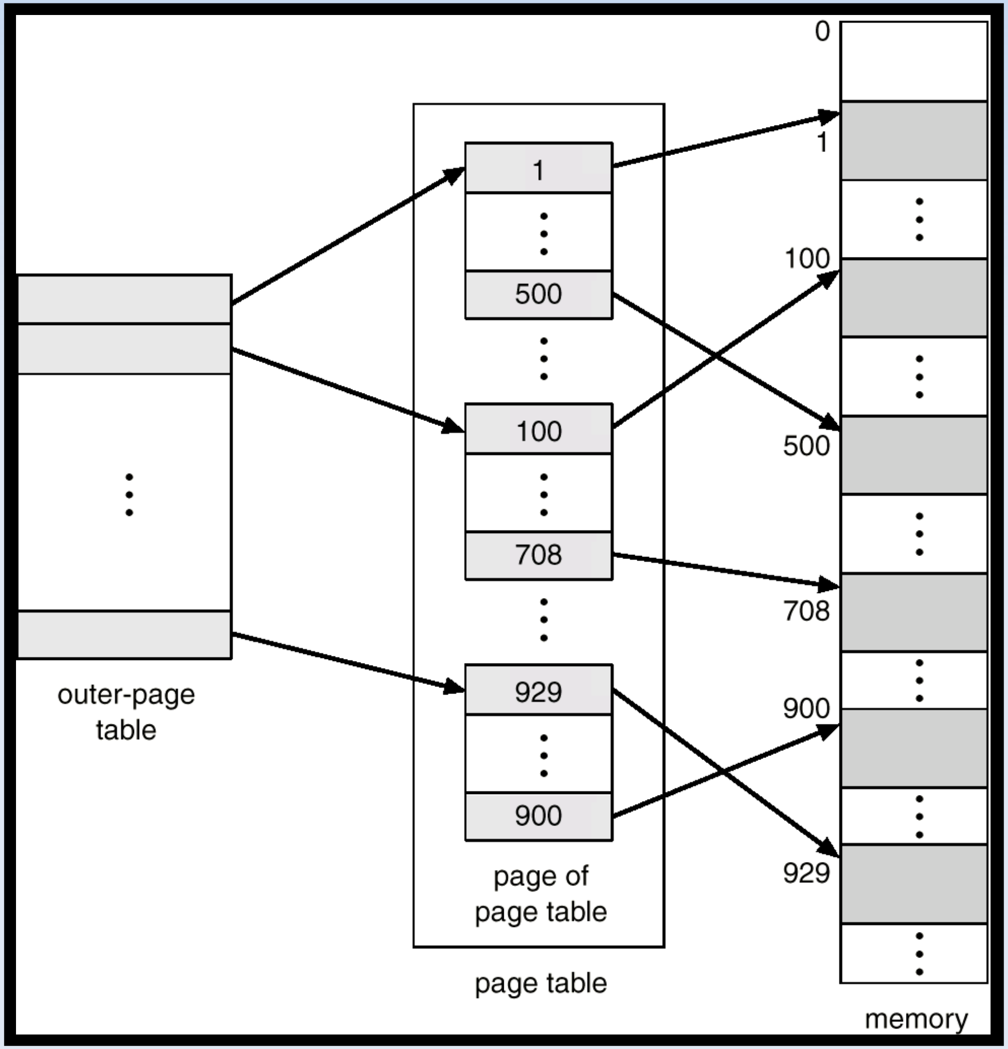
Problem : page table size

- Flat page tables are huge
- Example
  - 4GB of virtual memory (32 bits address)
  - 4KB pages
  - 20bits page number, 12 bits offset
  - 1MB page size :<

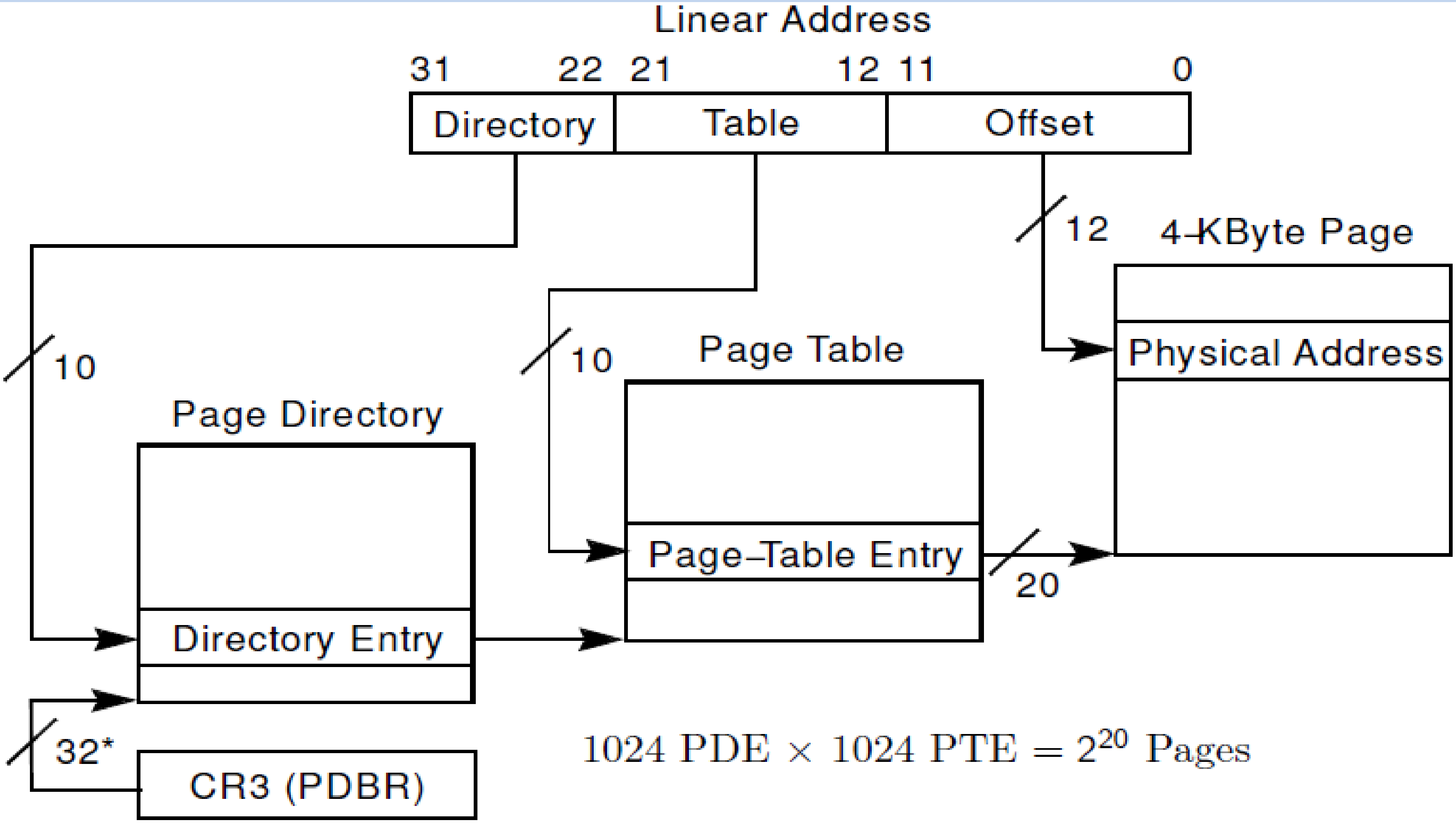
Multilevel Page Tables

- Reduce the size of page table in memory
- Structured page tables in 2 or more levels
  - All the page tables are not present in memory all the time
  - Some page tables are stored on disk and fetched if necessary
- Based on a demand paging mechanism

Example: two level pages



Example: Two level pages

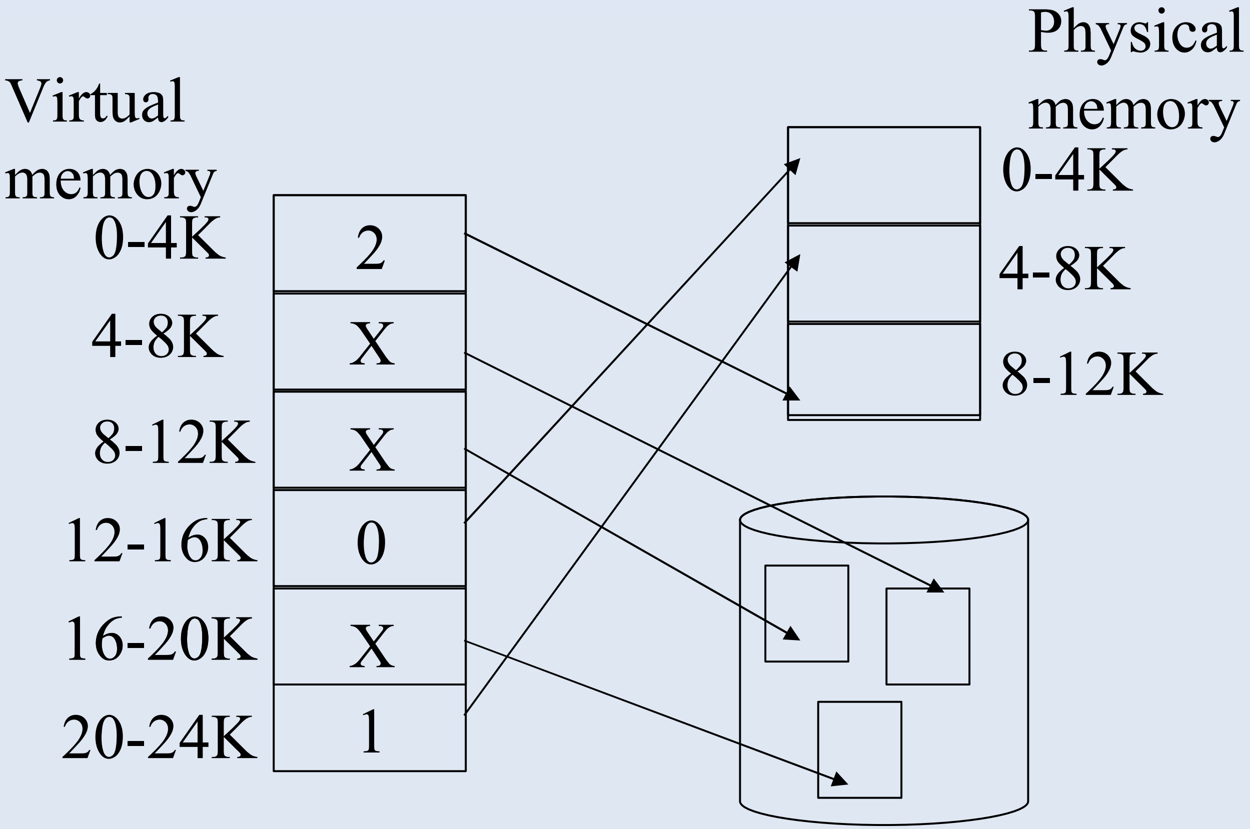


$1024 \text{ PDE} \times 1024 \text{ PTE} = 2^{20} \text{ Pages}$

*32 bits aligned onto a 4–KByte boundary

On Demand Paging

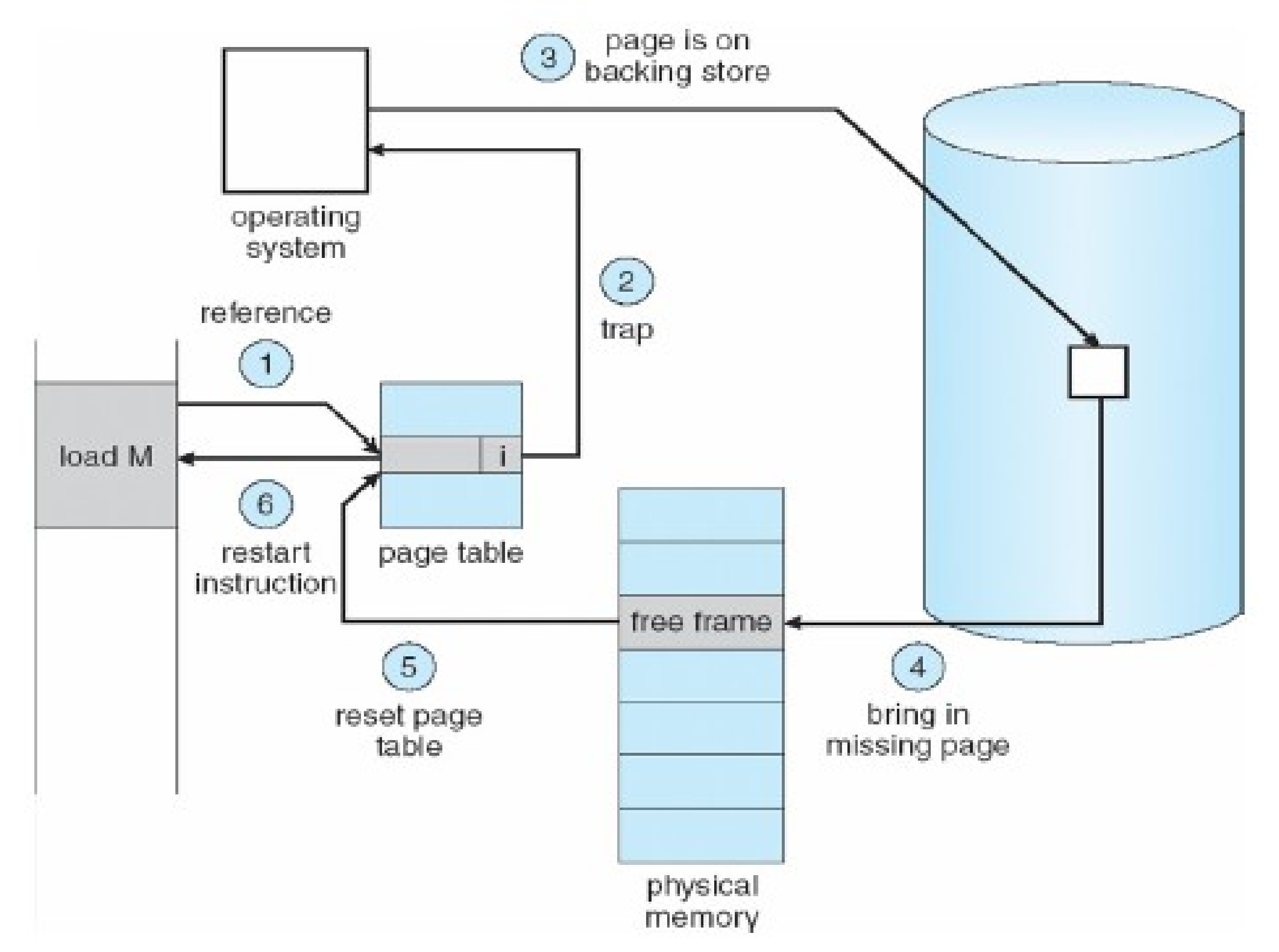
- Virtual memory > physical memory
  - Some pages are not present in memory (X)
  - Stored on disk



Page fault

- Access to an absent page
    - Presence bit
    - Page fault (Trap to OS)
- Page fault management
    - Find a free frame
        - If there is a free frame; use it
        - Select a page to replace
        - Save the replaced page on disk if necessary (dirty page)
    - Load the page from disk in the physical block
    - Update page table
    - Restart instruction
- Require a presence bit, a dirty bit, a disk @ in the page table
- Different page replacement algorithms

On Demand Paging

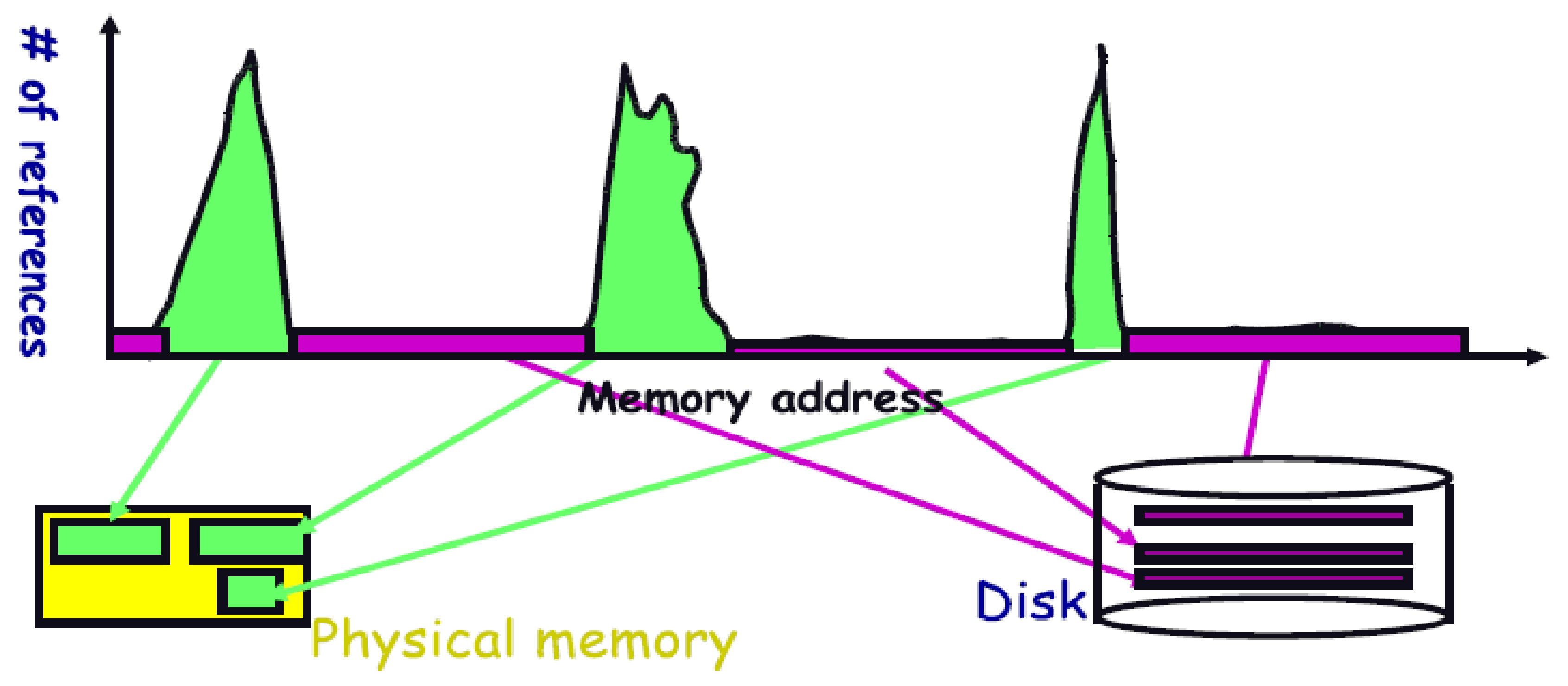


Page replacement algorithms

- Working set model
- Algorithms
  - Optimal
  - FIFO
  - Second chance
  - LRU

Working set model

- Disk much, much slower than memory
    - Goal: Run at memory, not disk speeds
- 90/10 rule: 10% of memory gets 90% of memory refs
    - So, keep that 10% in real memory, the other 90% on disk



Optimal page replacement

- What is optimal (if you knew the future)?
  - Replace page that will not be used for longest period of time
- Example
  - Reference string : 1,2,3,4,1,2,5,1,2,3,4,5,2,3
  - 4 physicals pages:

1	1	4
2	2	2
3	3	3
4	5	5

6 pages faults

FiFo

- Evict oldest page in system
- Example
  - Reference string : 1,2,3,4,1,2,5,1,2,3,4,5,2,3
  - 4 physicals frames:

10 page faults

1	5	5	5	5	4	4
2	2	1	1	1	1	5
3	3	3	2	2	2	2
4	4	4	4	3	3	3

LRU page replacement

- Approximate optimal with least recently used
  - Because past often predicts the future
- Example
  - Reference string : 1,2,3,4,1,2,5,1,2,3,4,5,2,3
  - 4 physicals frames:

1	1	1	1	5
2	2	2	2	2
3	5	5	4	4
4	4	3	3	3

8 page faults

LRU implementation

- Expensive
  - Need specific hardware
- Approximate LRU in software
  - The aging algorithm
    - Add a counter for each page (the date)
    - On a page access, all page counters are shifted left, inject 1 for the accessed page, else 0
    - On a page fault, remove the page with the lowest counter

Aging : example

Accès	Date Page0	Date Page1	Date Page2	Ordre pages /date
	000	000	000	
Page 0	100	000	000	P0,P1=P2
Page 1	010	100	000	P1,P0,P2
Page 2	001	010	100	P2,P1,P0
Page 1	000	101	010	P1,P2,P0

P0 is the oldest

Second chance

- Simple FIFO modification
  - Use an access bit R for each page
    - R = 0 : page not referenced
    - Periodically reset by hardware
  - Inspect the R bit of the oldest page
    - If 0 : replace the page
    - If 1 : clear the bit, put the page at the end of the list

Page buffering

- Naïve paging
  - Page replacement : 2 disk IO per page fault
- Reduce the IO on the critical path
  - Keep a pool of free frames
    - Fetch the page in the already free page

Paging

- Separate linking from memory concern
- Simplifies allocation, free and swap
- Eliminate external fragmentation
- May leverage internal fragmentation