Memory management

Tran, Van Hoai

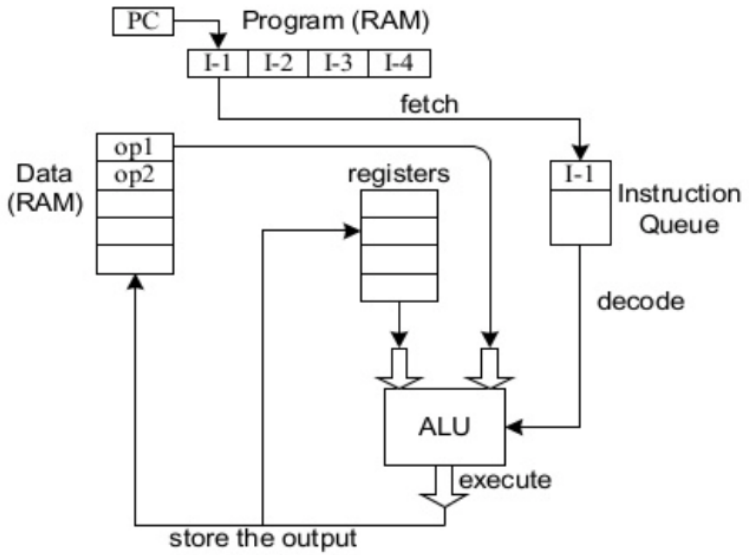
Faculty of Computer Science & Engineering
HCMC University of Technology

E-mail: hoai@hcmut.edu.vn
(partly based on slides of Le Thanh Van)

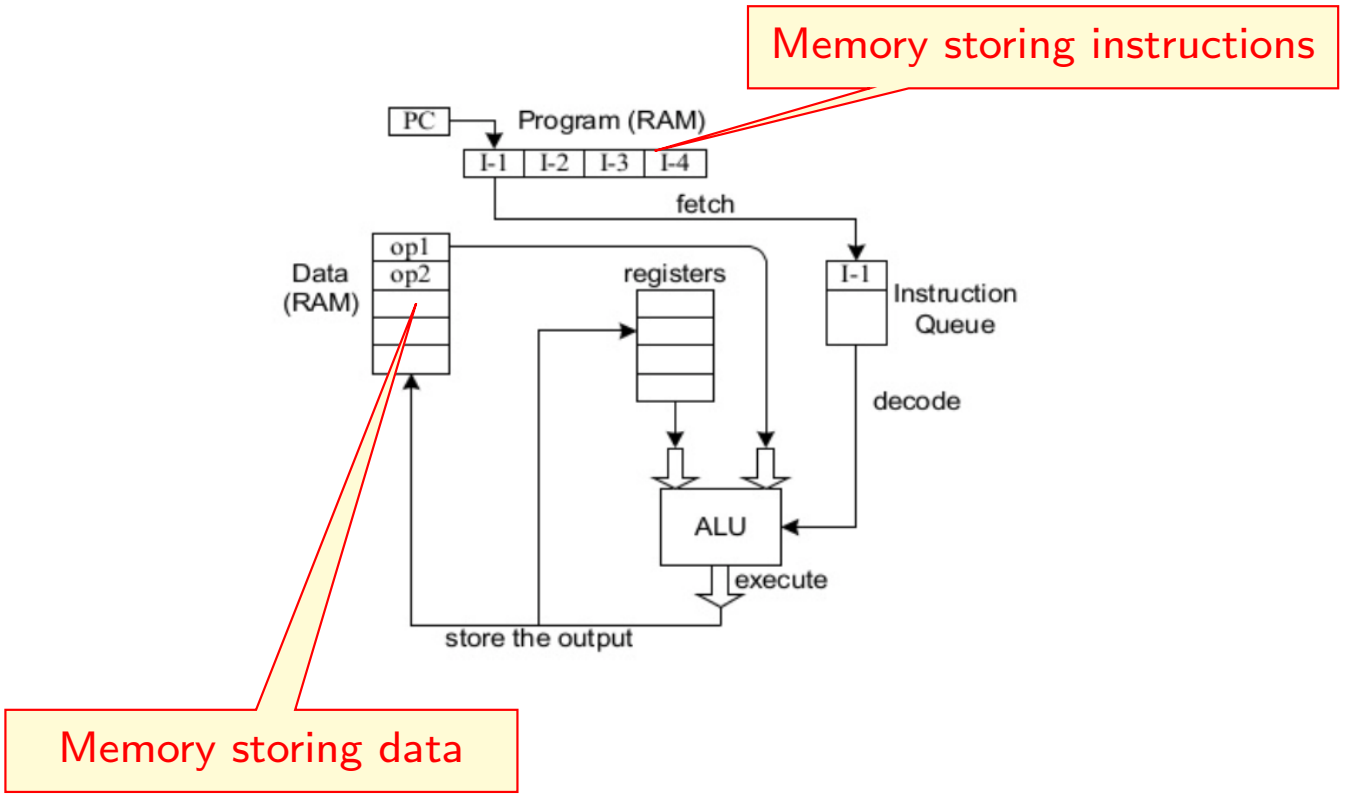
Outline

Outline

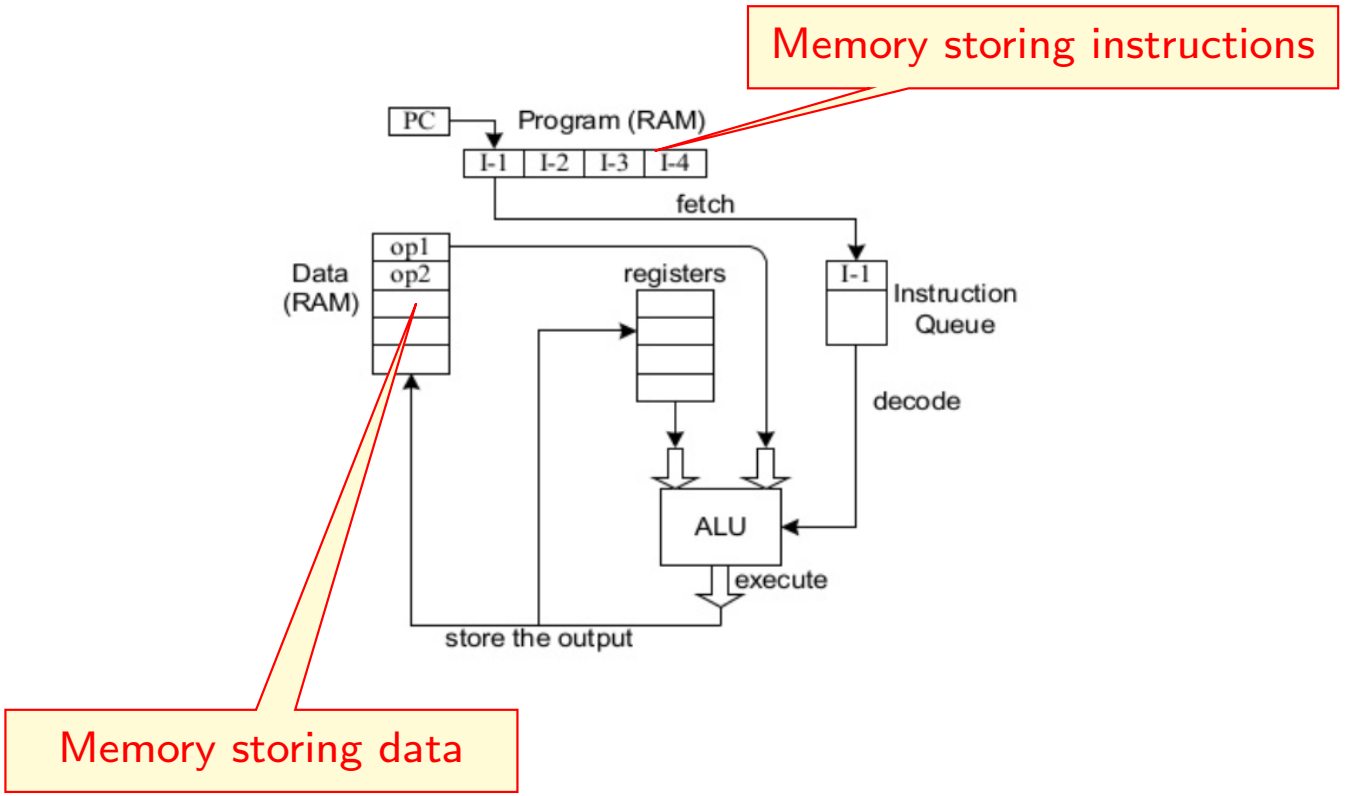
Role of memory



Role of memory



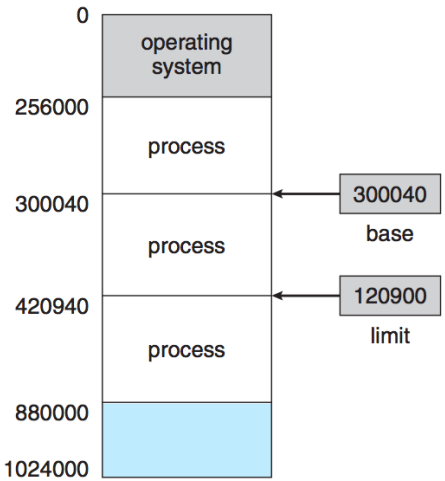
Role of memory

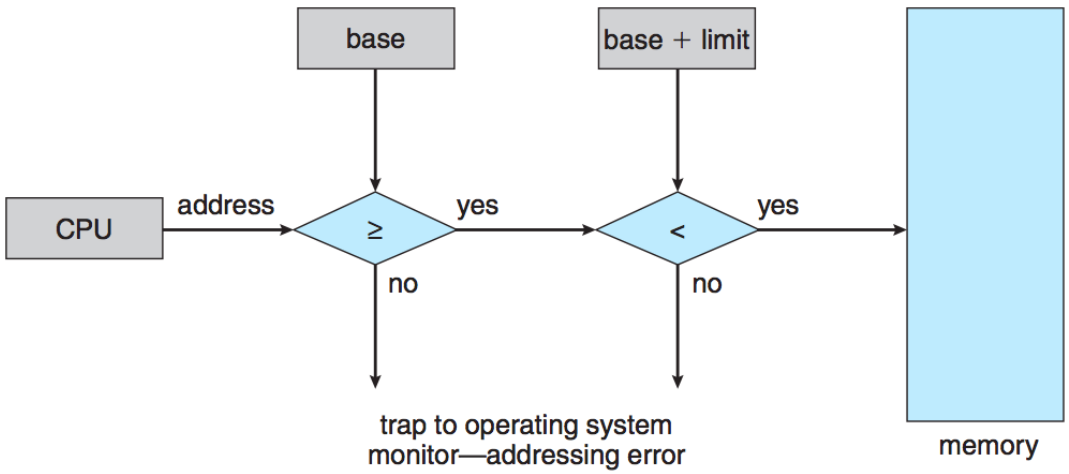


- Instructions of programs are only executed when they are in memory
- Memory management requires some hardware support

Hardware support

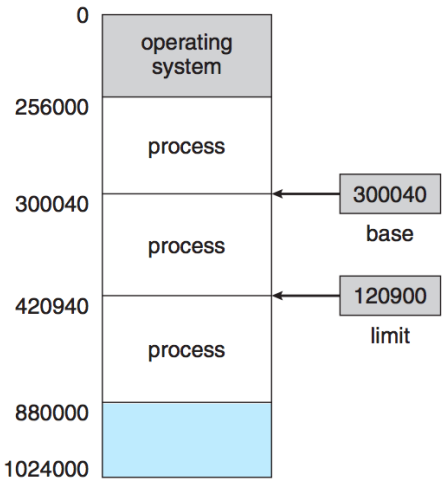
- Each user process has a separate memory space
- Base and limit registers give a range of the memory space

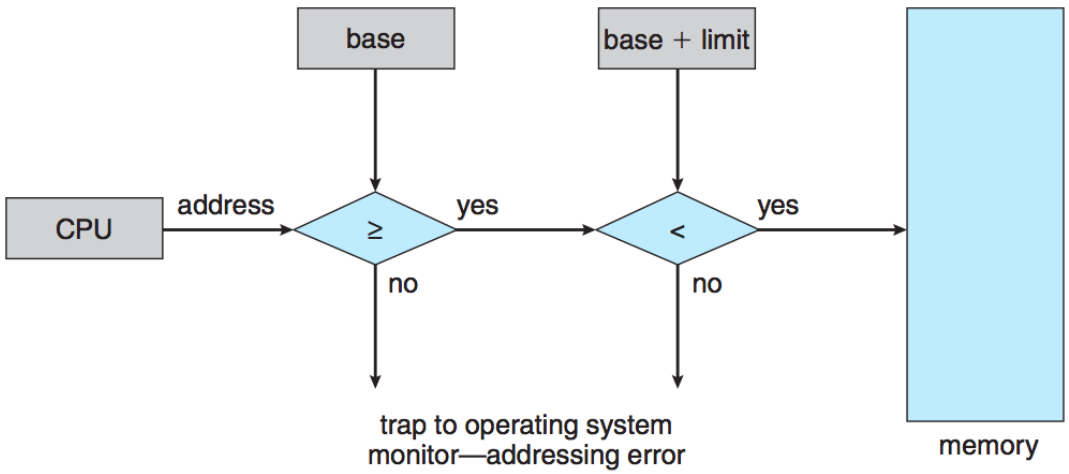




Hardware support

- Each user process has a separate memory space
- Base and limit registers give a range of the memory space

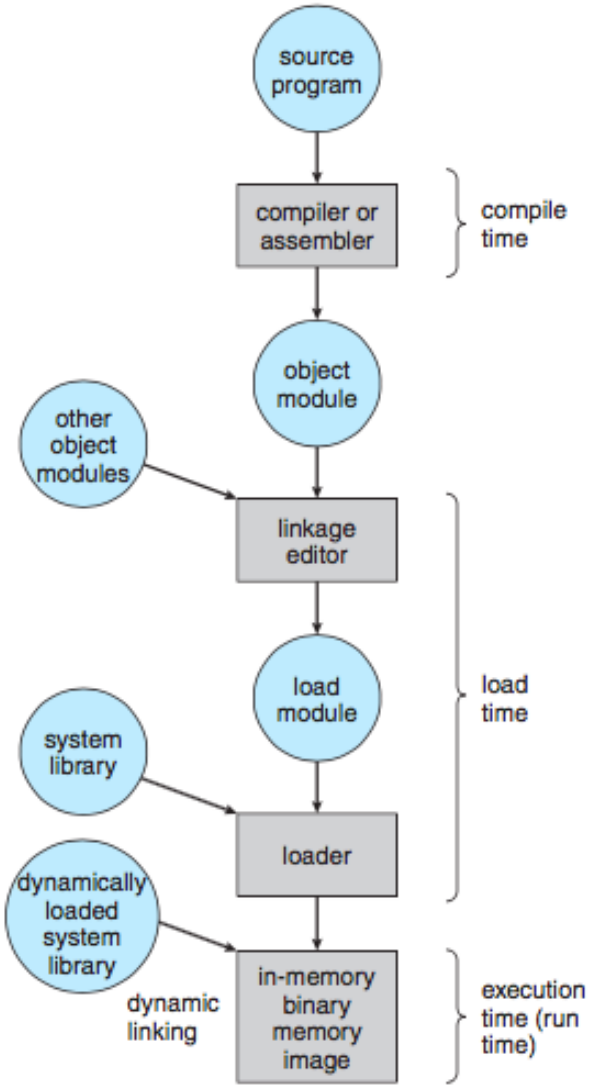




Memory protection mechanism is not applied for operating system executing in kernel mode

Address binding

User program to execution



Address binding

A process to convert one kind of high abstract memory address to low abstract memory address

- a symbolic address to a relocatable address
- a relocatable address to a absolute address

Example:

- a variable "`count`" → a relocatable address "14 bytes from the beginning of this module"
- "14 bytes from the beginning of this module" → an absolute address "74014"

Address binding

When it happens?

Binding is performed in any of following 3 steps

- Compile time: if starting location changes, absolute address must be regenerated (by compile)
- Load time: with relocatable code, only reloading is needed when starting address changes
- Execution time: if a process moved in memory, binding is delayed until its run time. Most general-purpose OSs use this method

Logical address vs. physical address

Logical address

Address generated by CPU

Physical address

Address seen by memory management unit (MMU)

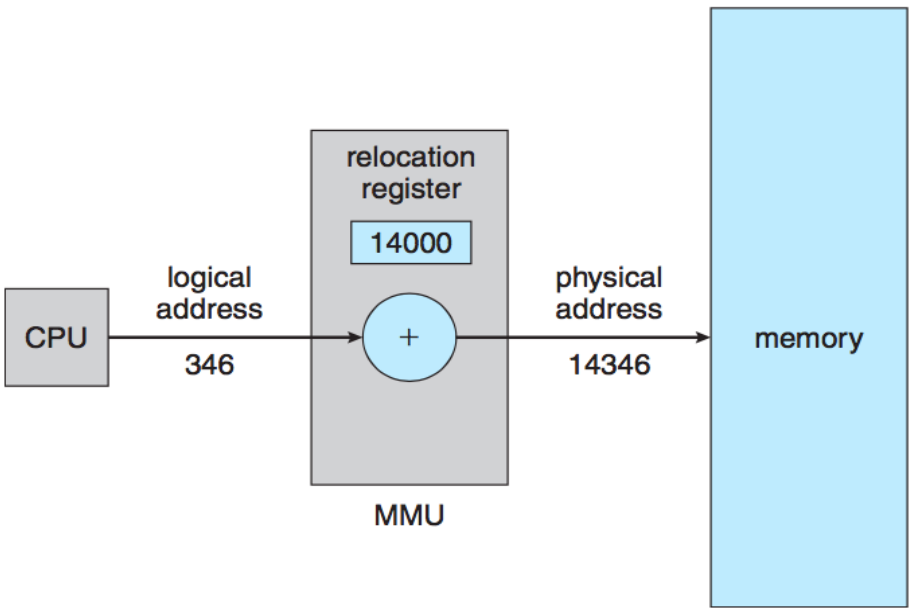
Logical address vs. physical address

Logical address

Address generated by CPU

Physical address

Address seen by memory management unit (MMU)



- Compile/load-time binding: logical address = physical address
- Execution-time binding: logical address ≠ physical address
  User program thinks logical (virtual) address range is $[0, max]$, but physical address range is $[R + 0, R + max]$.

Dynamic loading

Dynamic loading

A routine is not loaded until it is called

- Routine must be in relocatable load format
- Relocatable linking loader is responsible to load dynamic loading routing when a caller has not loaded yet
- Better memory=space utilization, useful to handle infrequently occuring cases
- Dynamic loading does not require support from OS

Dynamic linking and shared library

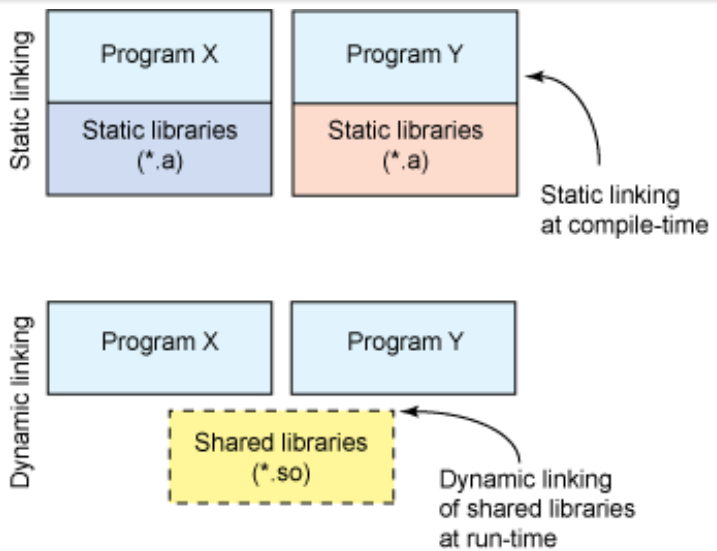
If multiple programs use a same routine, dynamic loading requires duplication of the routine in memory

Dynamic linking and shared library

If multiple programs use a same routine, dynamic loading requires duplication of the routine in memory

Dynamic linking

A routine is linked to user programs when the programs are run



(source:ibm)

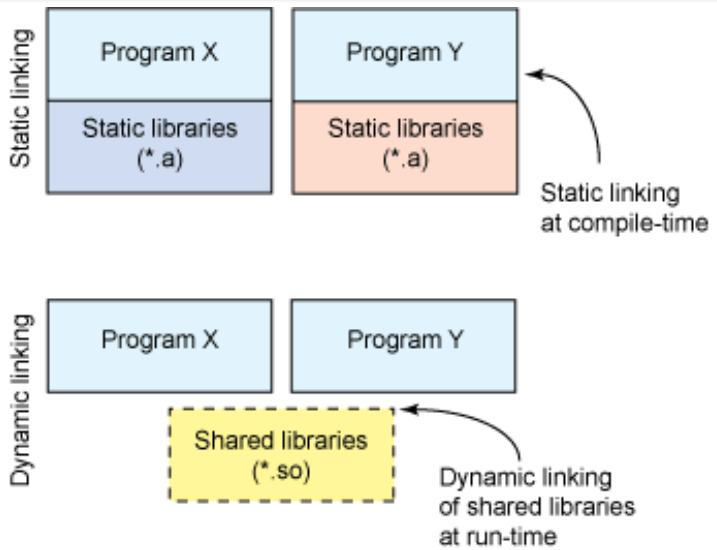
Dynamic linking and shared library

If multiple programs use a same routine, dynamic loading requires duplication of the routine in memory

Dynamic linking

A routine is linked to user programs when the programs are run

- A stub is put in the image of each reference
- The stub is replaced with the address of the needed routine, and then can be referred to by other references without loading



(source:ibm)

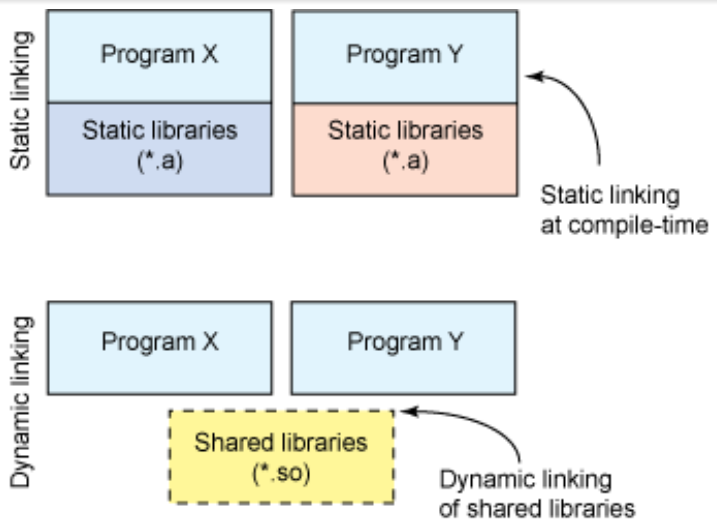
Dynamic linking and shared library

If multiple programs use a same routine, dynamic loading requires duplication of the routine in memory

Dynamic linking

A routine is linked to user programs when the programs are run

- A stub is put in the image of each reference
- The stub is replaced with the address of the needed routine, and then can be referred to by other

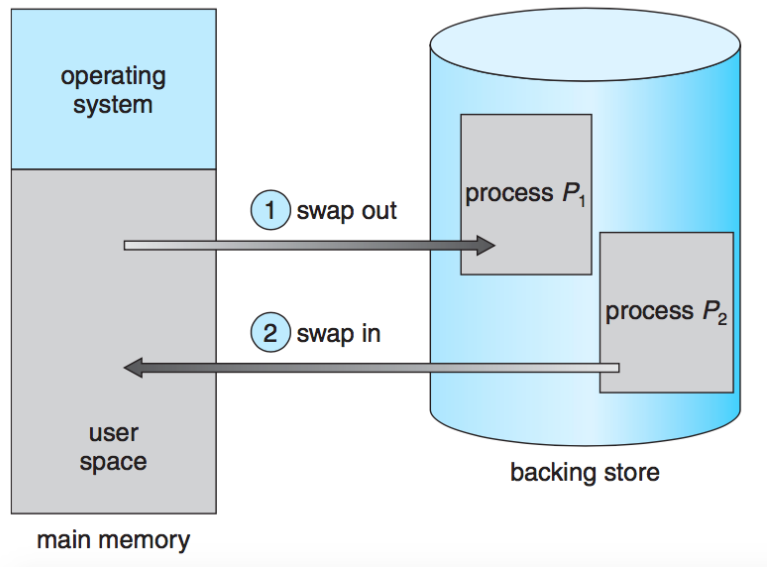


Dynamic linking requires support from OS

Outline

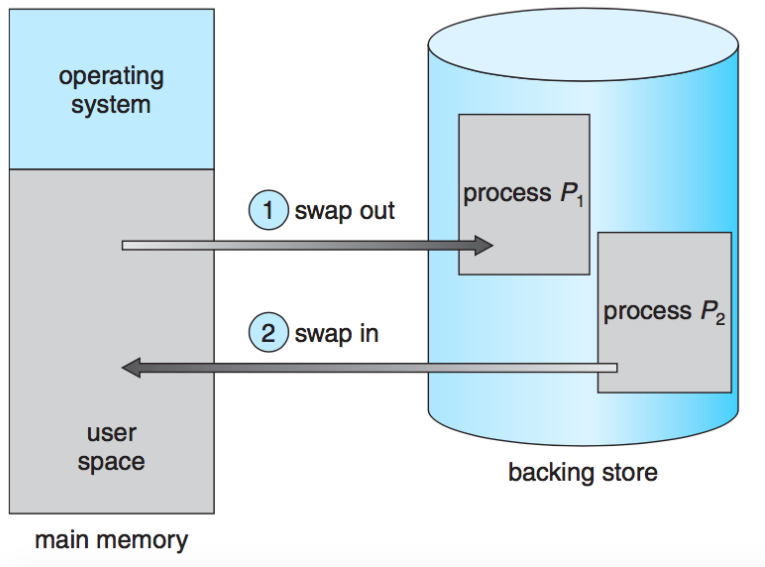
Standard swapping

- Backing store must be large enough to accommodate copies of all memory images of all users
- Images of processes in ready queue are in backing store



Standard swapping

- Backing store must be large enough to accommodate copies of all memory images of all users
- Images of processes in ready queue are in backing store

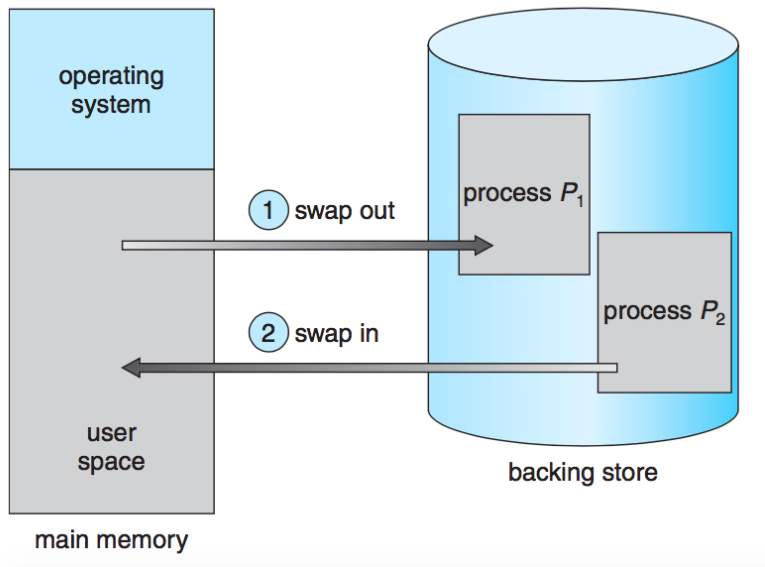


Swapping is influenced by factors

- Transfer time between fast disk and memory
- Informing mechanism to activate swapping
- I/O waiting processes should be swapped out carefully

Standard swapping

- Backing store must be large enough to accommodate copies of all memory images of all users
- Images of processes in ready queue are in backing store



Swapping is influenced by factors

- Transfer time between fast disk and memory
- Informing mechanism to activate swapping
- I/O waiting processes should be swapped out carefully

Standard swapping is not used in modern OSs

Swapping on mobile systems

Flash drives have finite number of write/erase cycles

- Mobile OSs do not support swapping
- OSs may terminate processes if memory is not sufficient
- Developers for mobile systems must carefully allocate and release memory

Outline

Contiguous memory allocation

- Memory divided into 2 partitions: resident OS & user processes
- Interrupt vector is often in low memory $\Rightarrow$ resident OS is low memory partition

Contiguous memory allocation

- Memory divided into 2 partitions: resident OS & user processes
- Interrupt vector is often in low memory $\Rightarrow$ resident OS is low memory partition

Contiguous memory allocation

Each process is contained in a single section of memory that is contiguous to the section of the next process

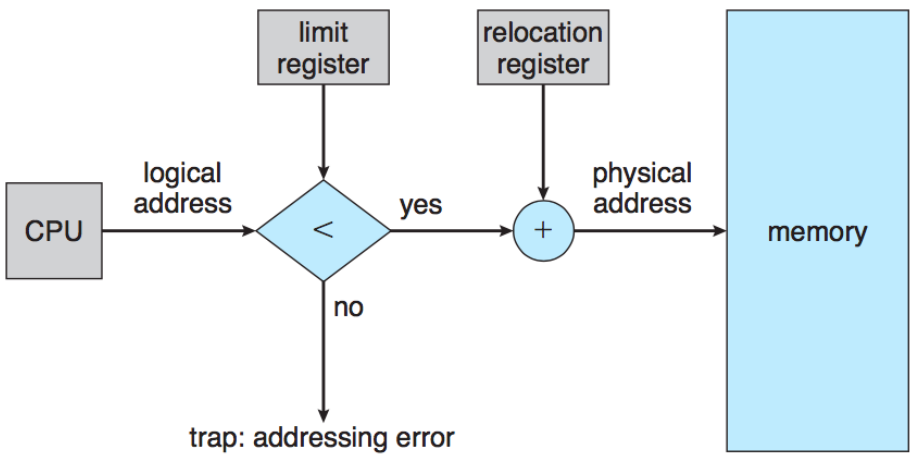
Contiguous memory allocation

- Memory divided into 2 partitions: resident OS & user processes
- Interrupt vector is often in low memory $\Rightarrow$ resident OS is low memory partition

Contiguous memory allocation

Each process is contained in a single section of memory that is contiguous to the section of the next process

OSs use relocation register for memory protection



Memory allocation

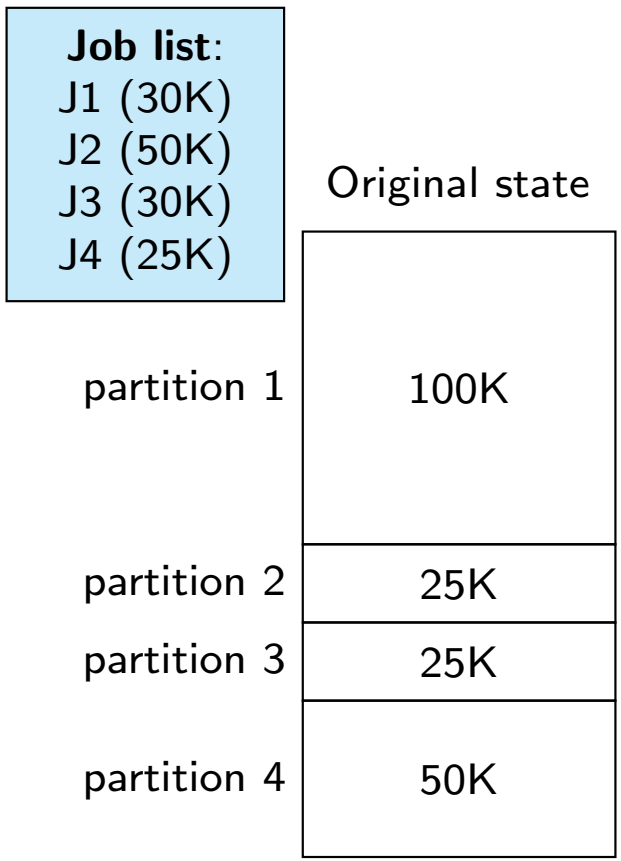
There are 2 ways

- Fixed-partition: memory is divided into several fixed-size partitions. A process is in a single partition
- Variable-partition: only assign enough memory for processes and OS keeps a table storing status of memory

Memory allocation

There are 2 ways

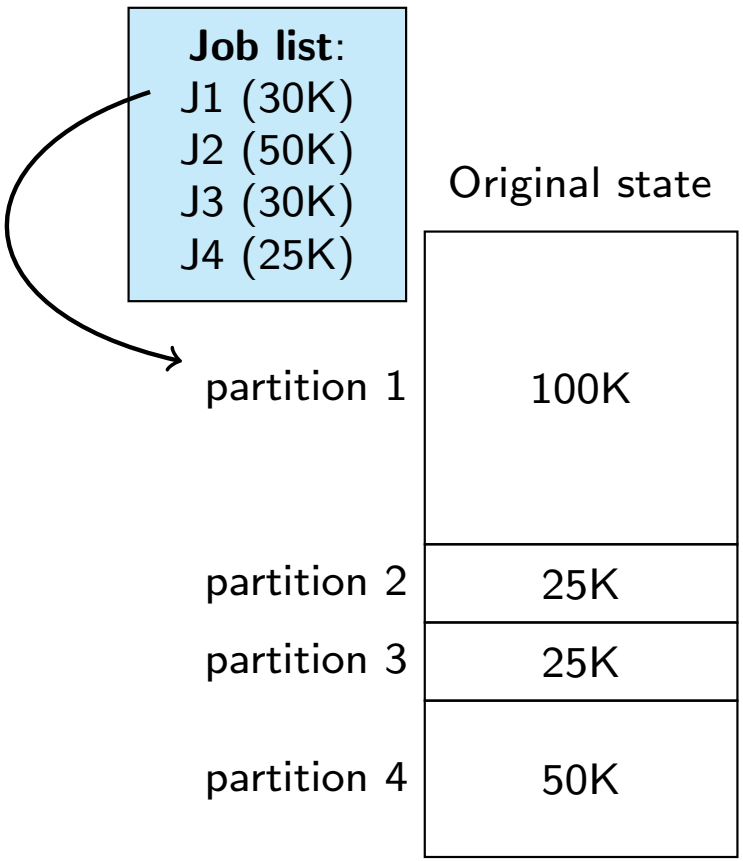
- Fixed-partition: memory is divided into several fixed-size partitions. A process is in a single partition
- Variable-partition: only assign enough memory for processes and OS keeps a table storing status of memory



Memory allocation

There are 2 ways

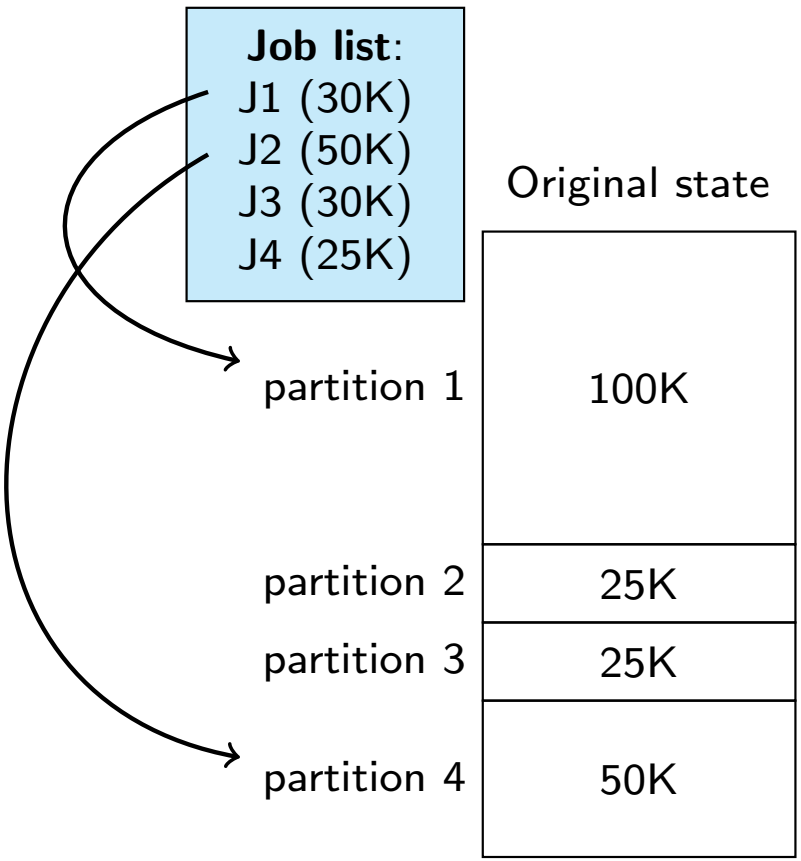
- Fixed-partition: memory is divided into several fixed-size partitions. A process is in a single partition
- Variable-partition: only assign enough memory for processes and OS keeps a table storing status of memory



Memory allocation

There are 2 ways

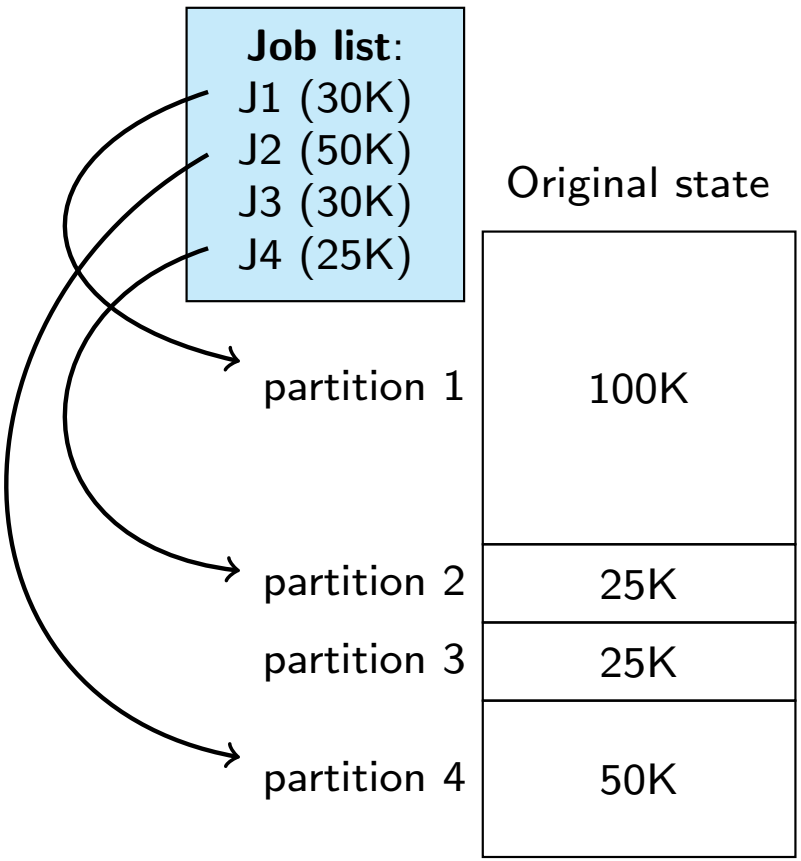
- Fixed-partition: memory is divided into several fixed-size partitions. A process is in a single partition
- Variable-partition: only assign enough memory for processes and OS keeps a table storing status of memory



Memory allocation

There are 2 ways

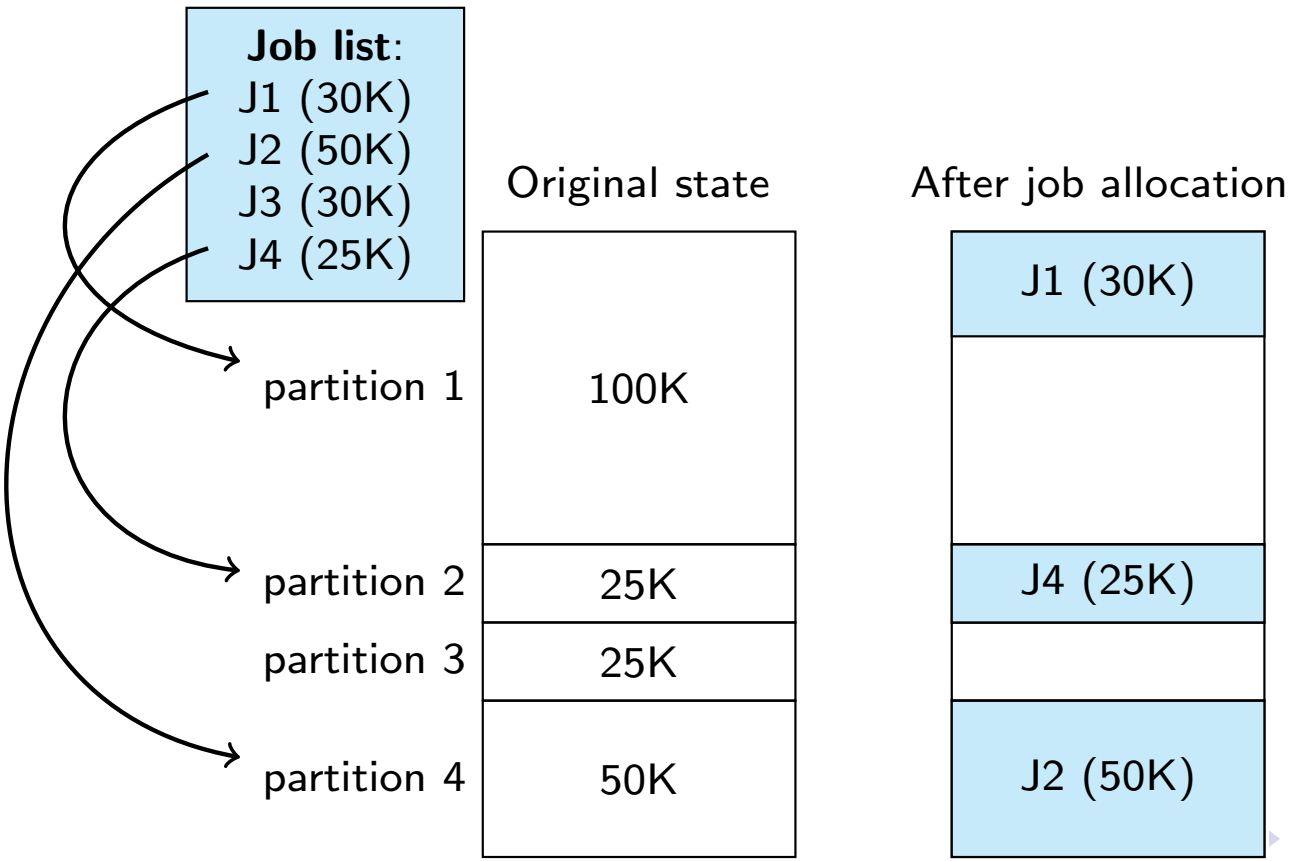
- Fixed-partition: memory is divided into several fixed-size partitions. A process is in a single partition
- Variable-partition: only assign enough memory for processes and OS keeps a table storing status of memory



Memory allocation

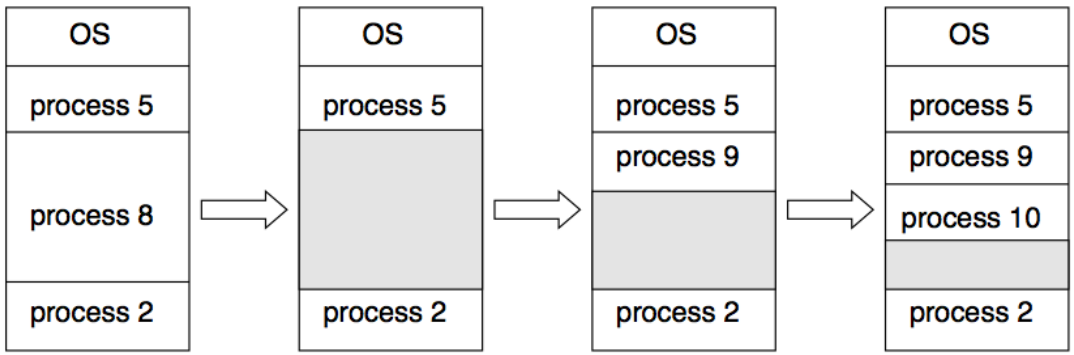
There are 2 ways

- Fixed-partition: memory is divided into several fixed-size partitions. A process is in a single partition
- Variable-partition: only assign enough memory for processes and OS keeps a table storing status of memory



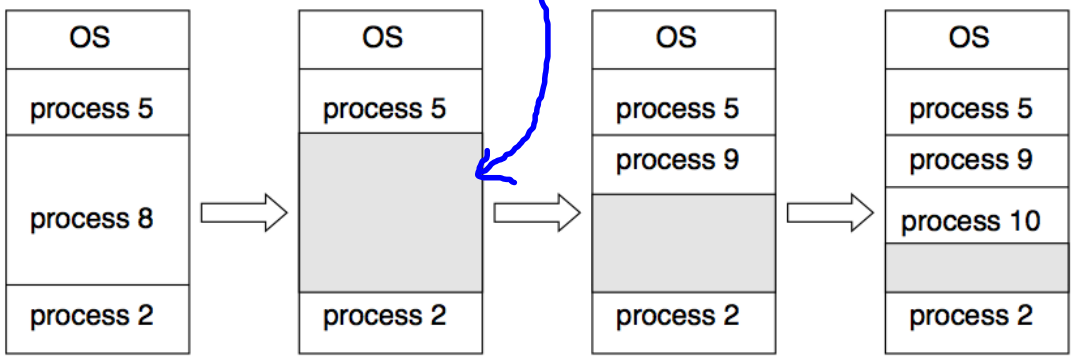
Variable partition

- Fixed-partition scheme limits the level of multiprogramming
- Variable-partition scheme needs a mechanism to deal efficiently with set of holes



Variable partition

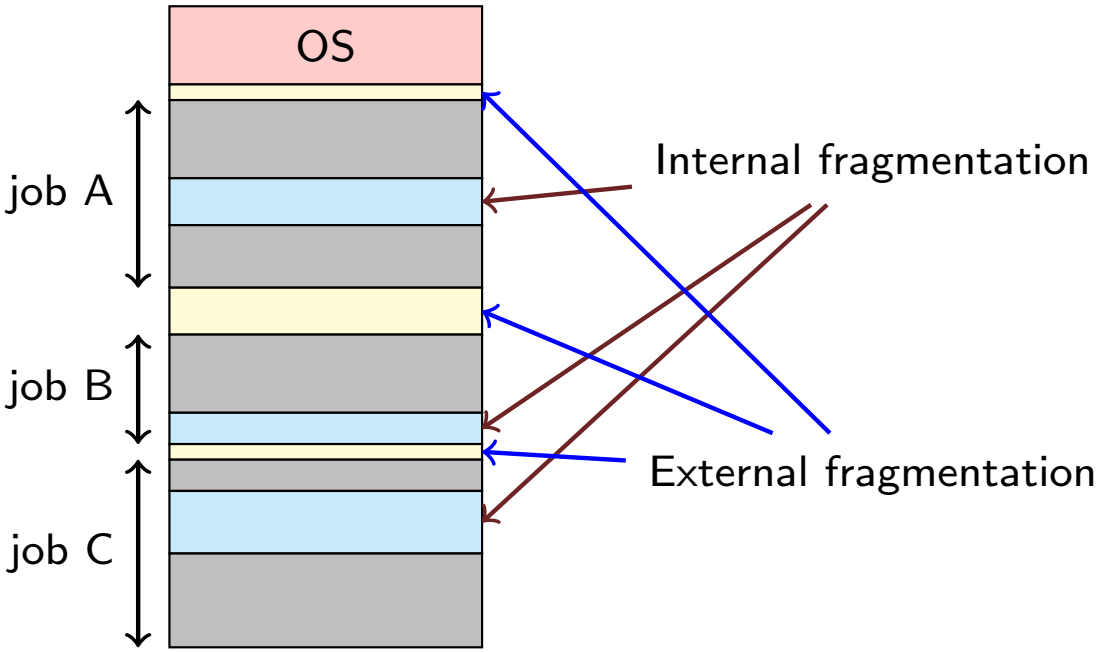
- Fixed-partition scheme limits the level of multiprogramming
- Variable-partition scheme needs a mechanism to deal efficiently with set of holes



  - State: list of available (free) block sizes, input queue
  - Dynamic storage-allocation problem: how to satisfy a request of size n from a list of free holes
  - Strategies: first-fit, best-fit, worst-fit

Fragmentation

Internal vs. external

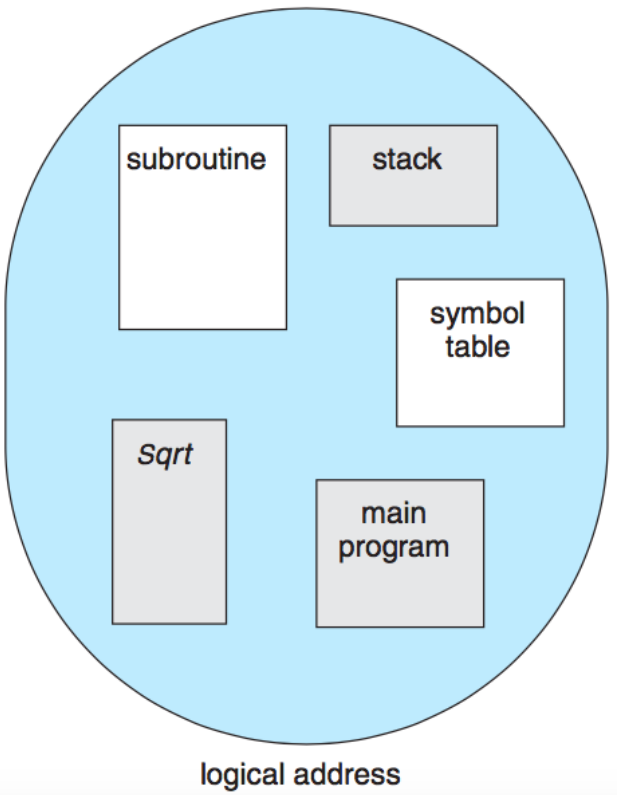


- External fragmentation: small holes which cannot satisfy large request
- Internal fragmentation: unused memory which has been allocated for process

Outline

What is segmentation ?

Memory management should be convenient to both OS and programmers

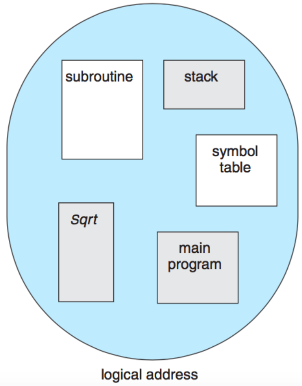


Programmers think of programs as set of data structures and methods. With memory, they need

- Collection of variable-sized memory blocks (segments)
- No neccessary ordering among segments

What is segmentation ?

Memory management should be convenient to both OS and programmers



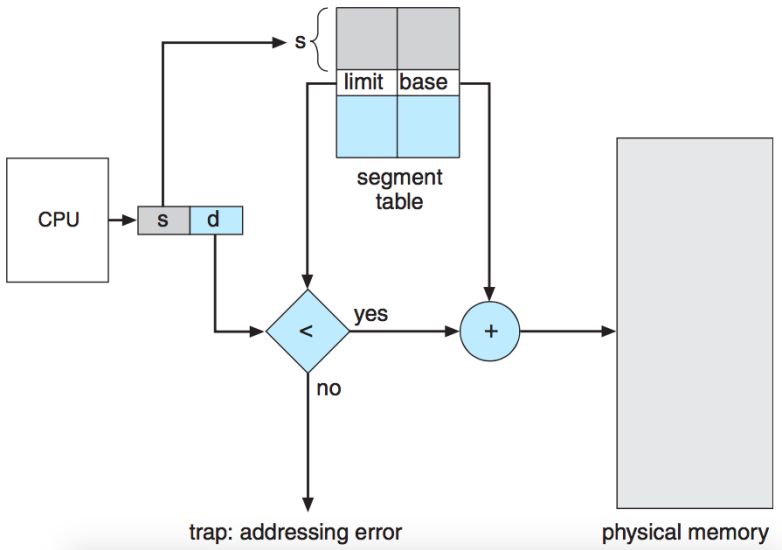
Programmers think of programs as set of data structures and methods. With memory, they need

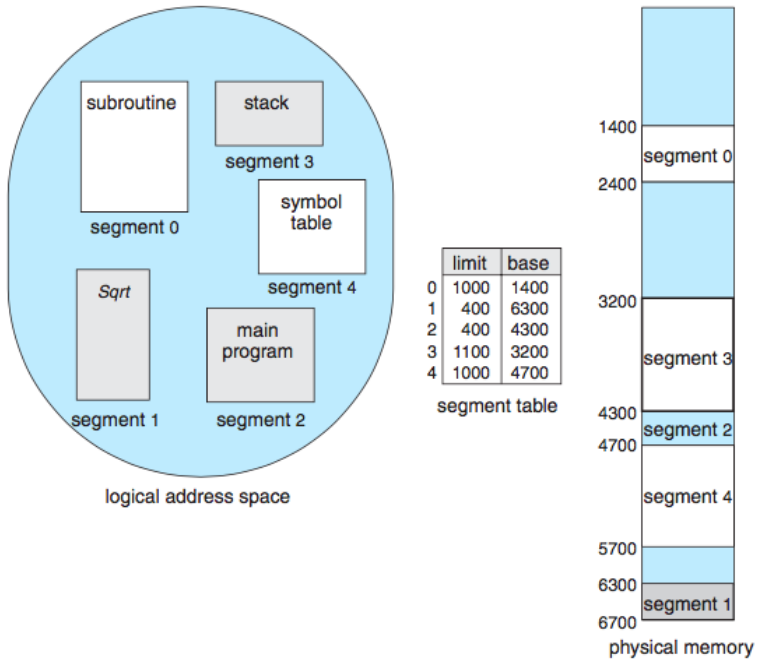
- Collection of variable-sized memory blocks (segments)
- No neccessary ordering among segments

Segment = <segment-number, offset>

Segmentation hardware

Segment address is 2-dimensional, but physical memory address is 1-dimensional





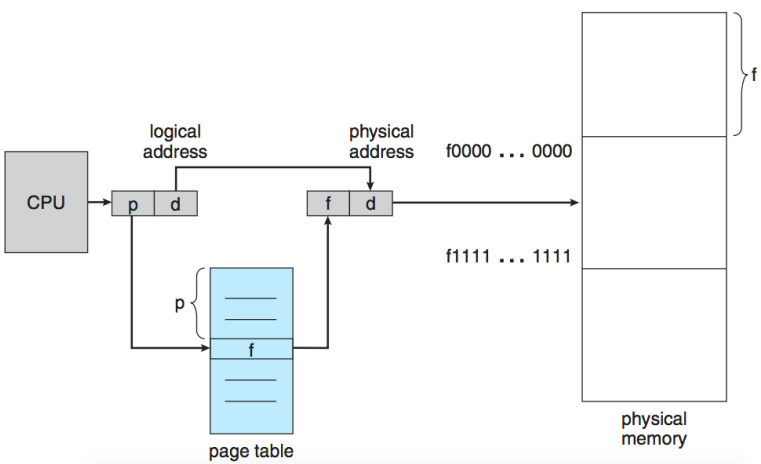
- Segment table is an array of base–limit register pairs

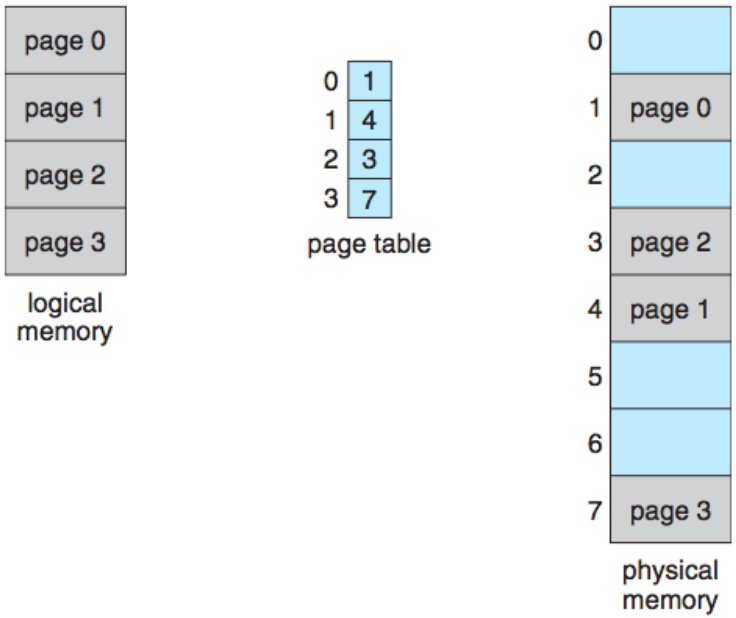
Outline

What is paging ?

Segmentation still has external fragmentation, requiring compaction

- Physical memory is divided into fixed-sized blocks (frame)
- Logical memory is divided into same-sized blocks (page)

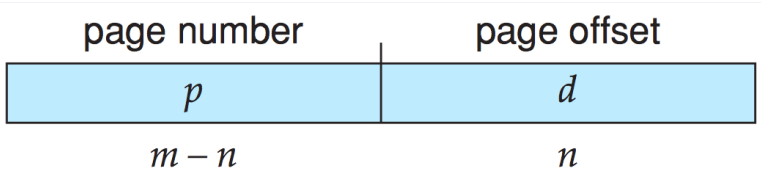




Page address = <page number, page offset>

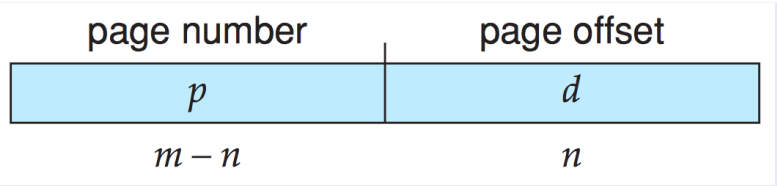
Paging in practice

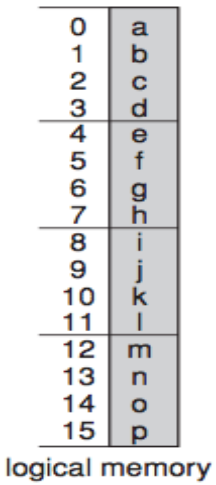
Logical address space $= 2^m$
Page size $= 2^n$

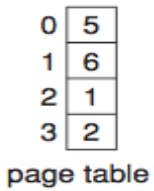


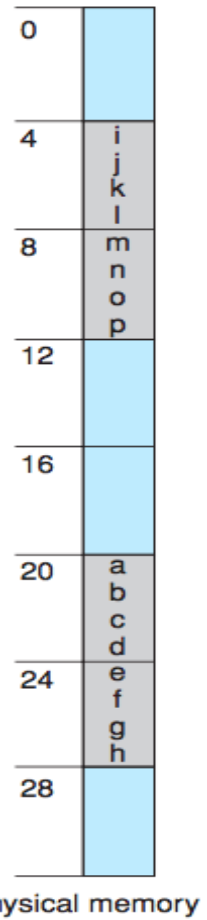
Paging in practice

Logical address space $= 2^m$
Page size $= 2^n$







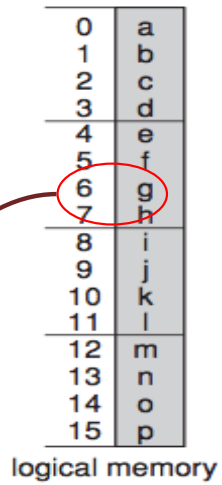


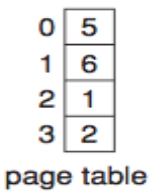
$n = 2, m = 4$

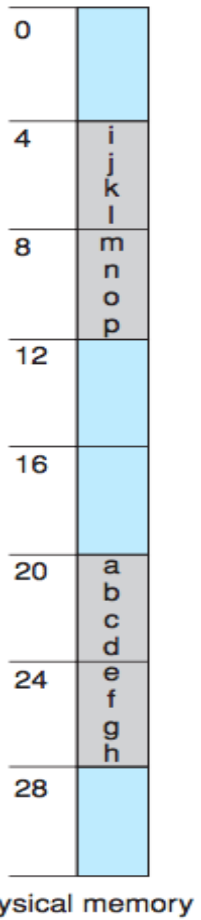
Paging in practice

Logical address space $= 2^m$
Page size $= 2^n$

page number	page offset
p	d
$m-n$	n







$6 = 0110_2$

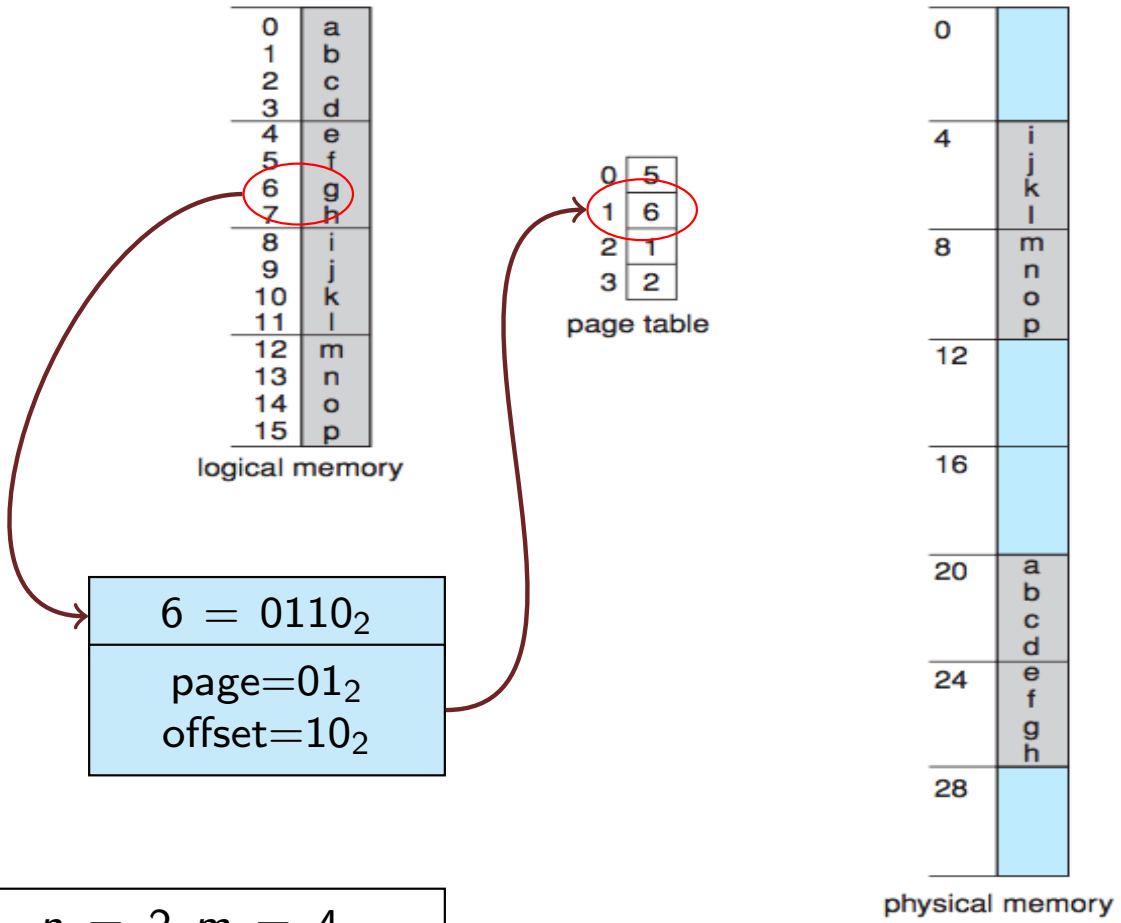
page=01_2
offset=10_2

$n = 2, m = 4$

Paging in practice

Logical address space $= 2^m$
Page size $= 2^n$

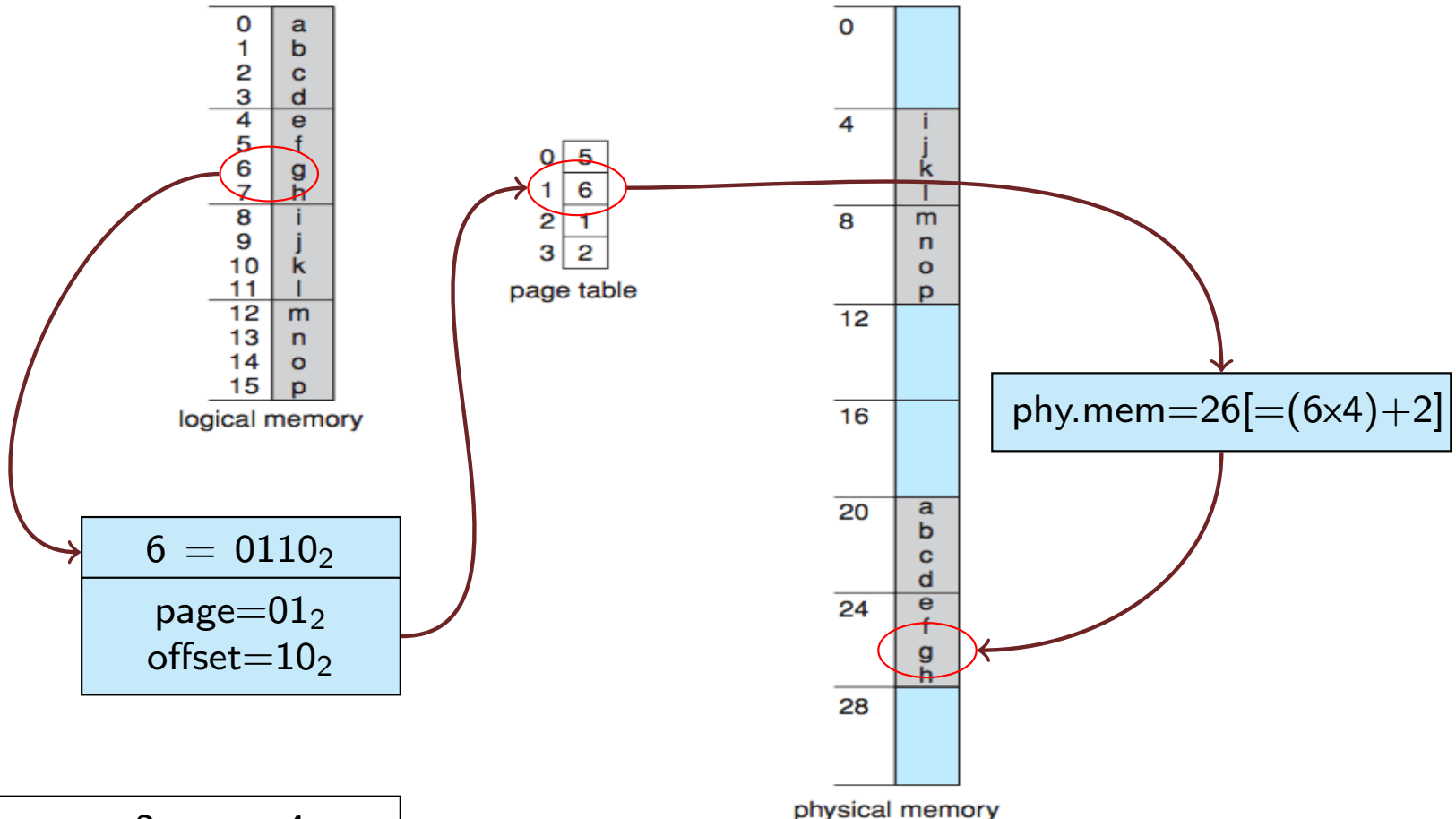
page number	page offset
p	d
$m-n$	n



Paging in practice

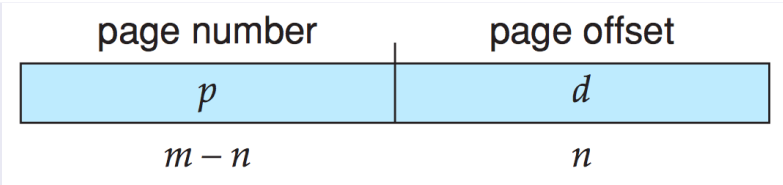
Logical address space $= 2^m$
Page size $= 2^n$

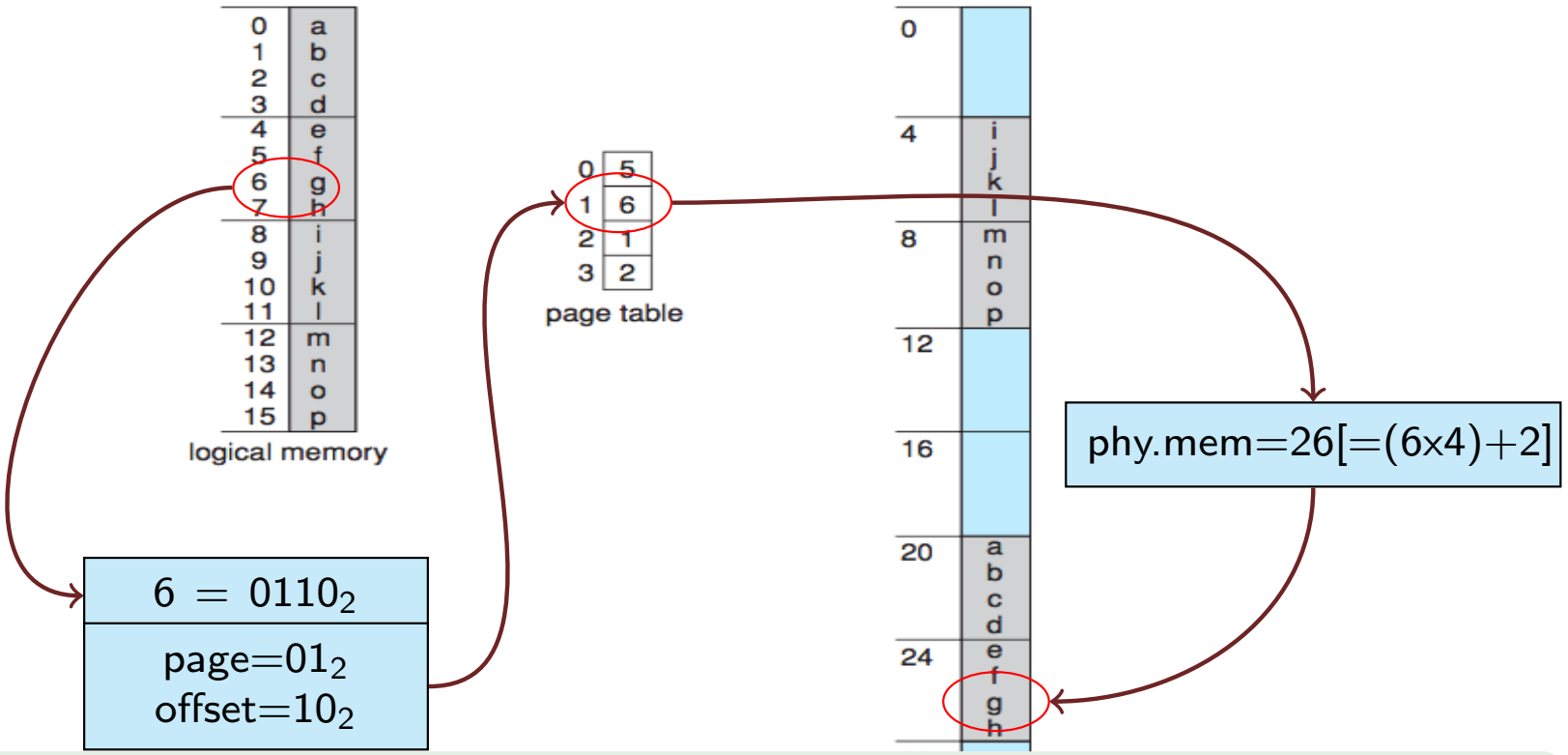
page number	page offset
p	d
$m-n$	n



Paging in practice

Logical address space $= 2^m$
Page size $= 2^n$



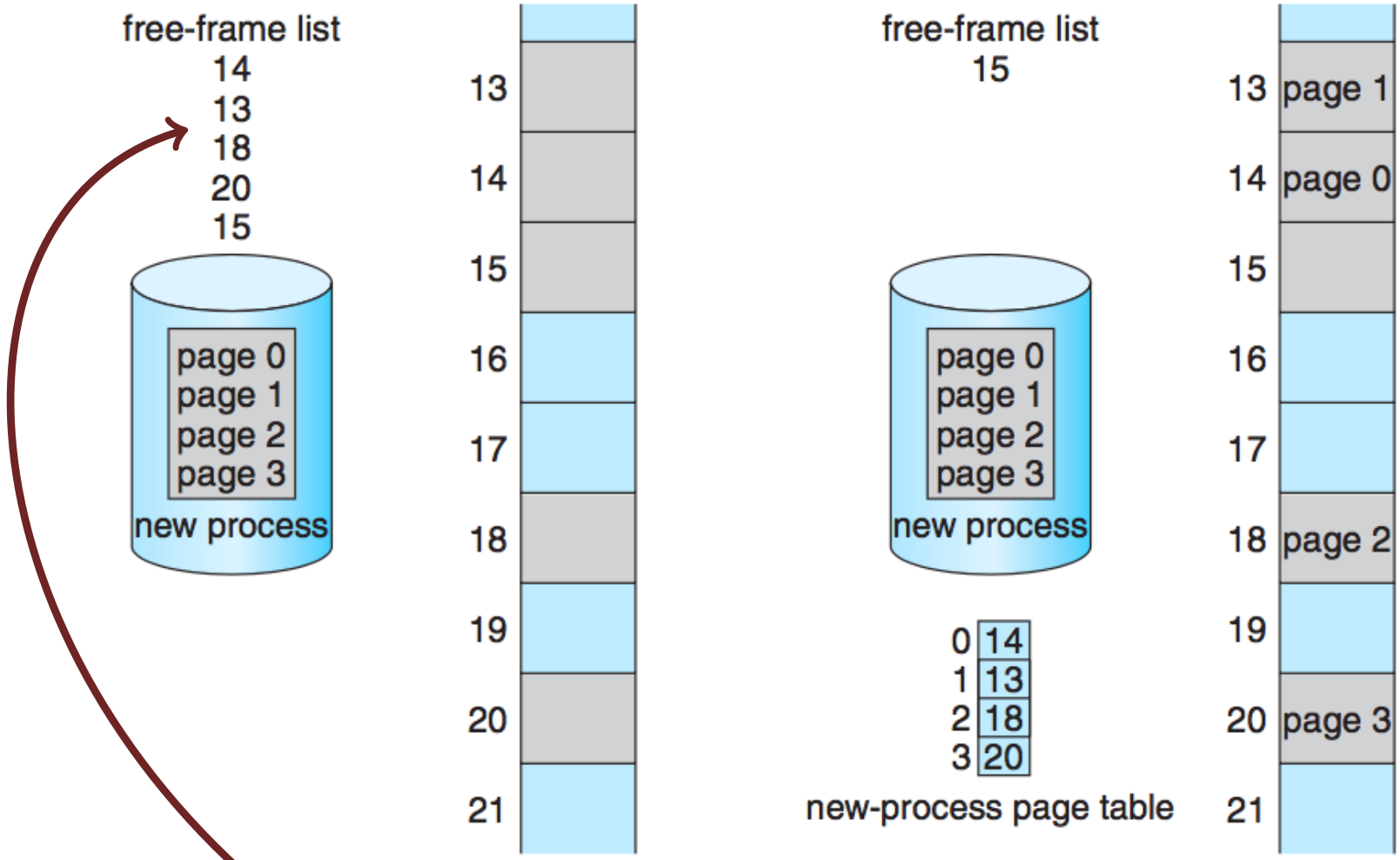


Paging has no external fragmentation, but has internal fragmentation.

Frame allocation

A process can be given a non-contiguous set of frames

Frame allocation



A process can be given a non contiguous set

A data structure (frame-table) is needed to manage frames.

Hardware support

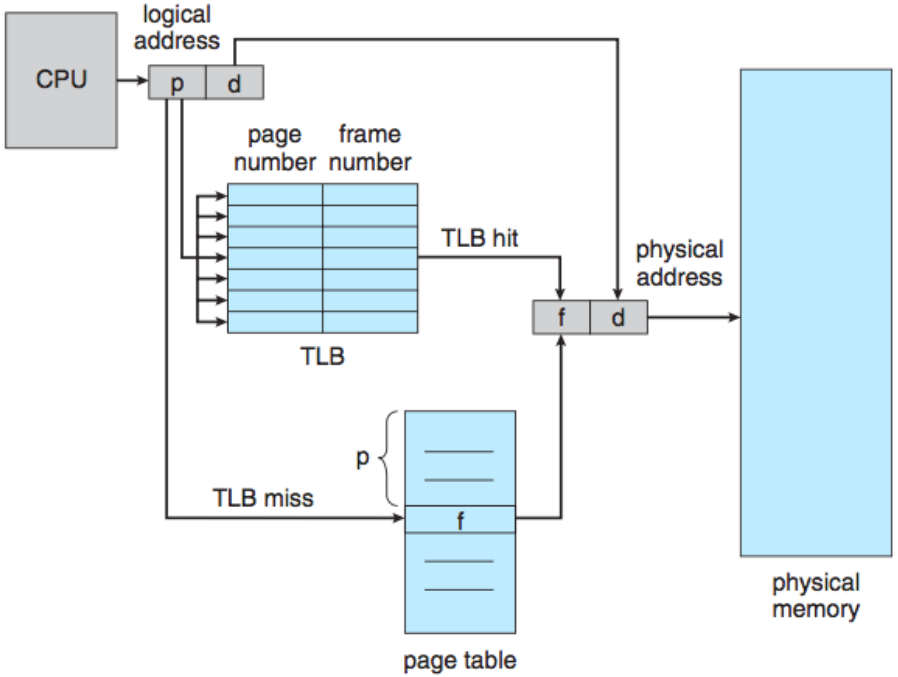
- Page table stored in main memory,
  - A page table for each process, pointer to it in PCB
  - Few page tables for all processes

Hardware support

- Page table stored in main memory,
  - A page table for each process, pointer to it in PCB
  - Few page tables for all processes
- Page-address translation overhead $\Rightarrow$ hardware support
  - Page-table base register (PTBR): point to page table
  - Translation look-aside buffer (TLB): associative, high-speed memory (<key,value>)

Hardware support

- Page table stored in main memory,
  - A page table for each process, pointer to it in PCB
  - Few page tables for all processes
- Page-address translation overhead $\Rightarrow$ hardware support
  - Page-table base register (PTBR): point to page table
  - Translation look-aside buffer (TLB): associative, high-speed memory (<key,value>)

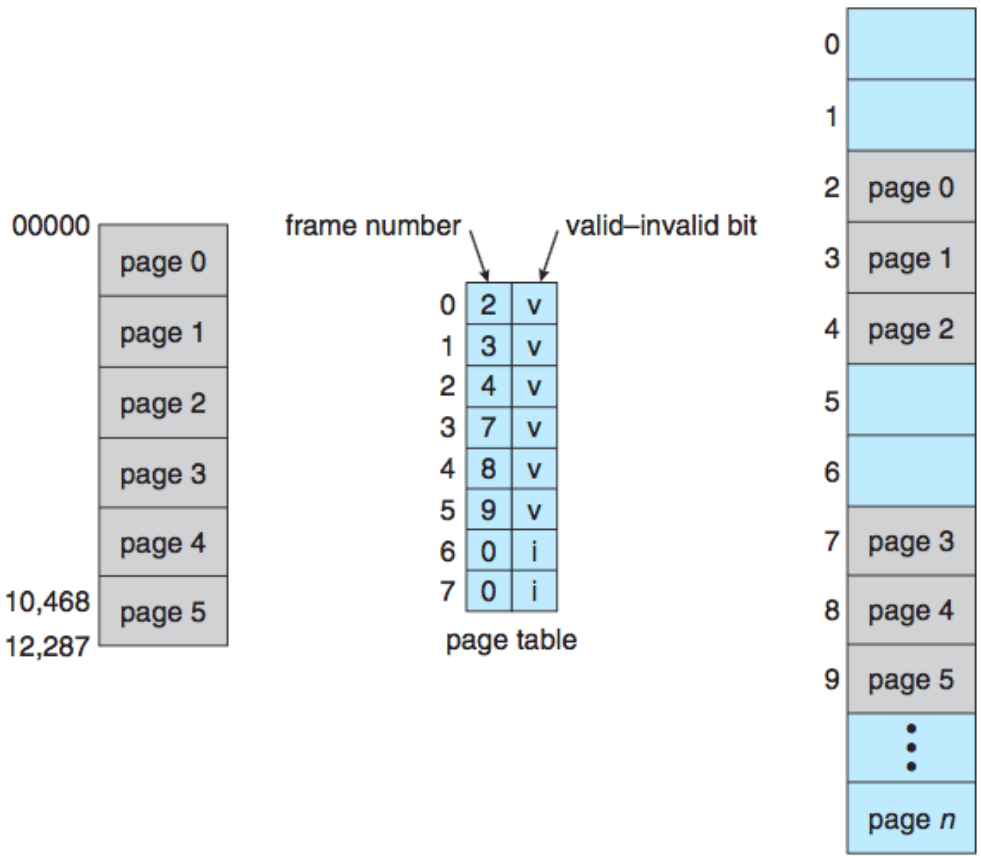


hit ratio

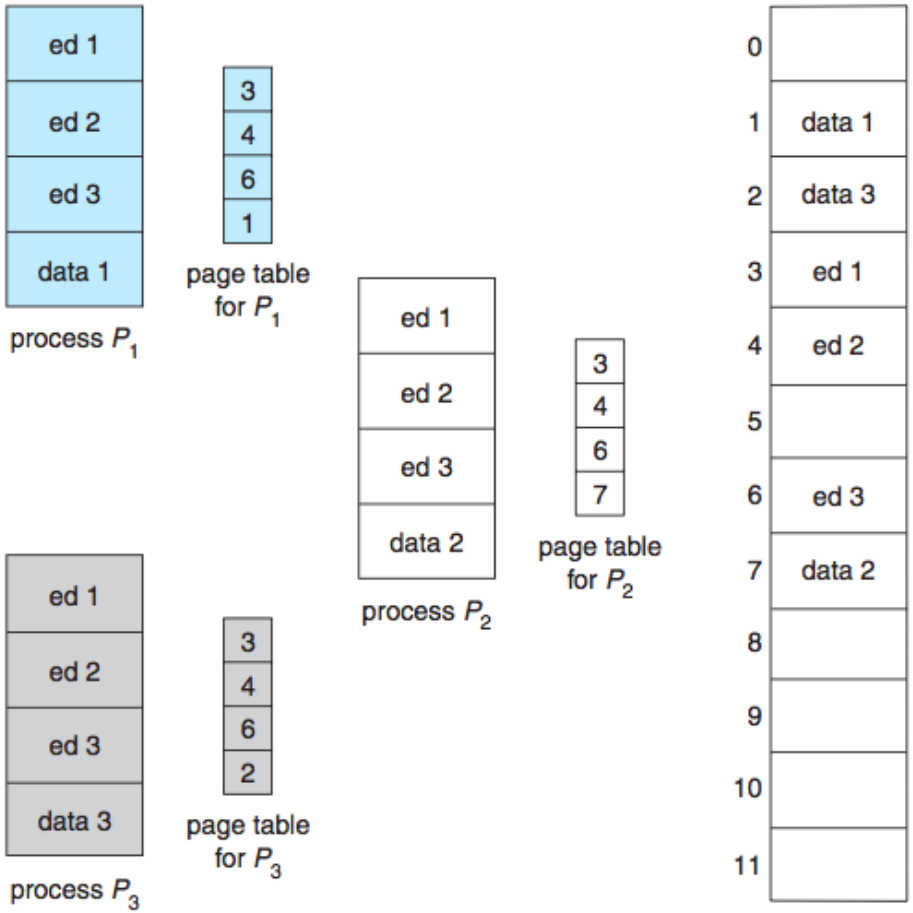
- 80%
  effective access time =
  $80\% \times 100(ns) + 20\% \times 200(ns) = 120(\text{ns})$
- 99%
  effective access time =
  $99\% \times 100(ns) + 1\% \times 200(ns) = 101(\text{ns})$

Protection

- From access permission: read-write/read-only bit
- From out-of-range access: valid-invalid bit
  - Just keep a small range of pages ⇒ page-table length register (PTLR)



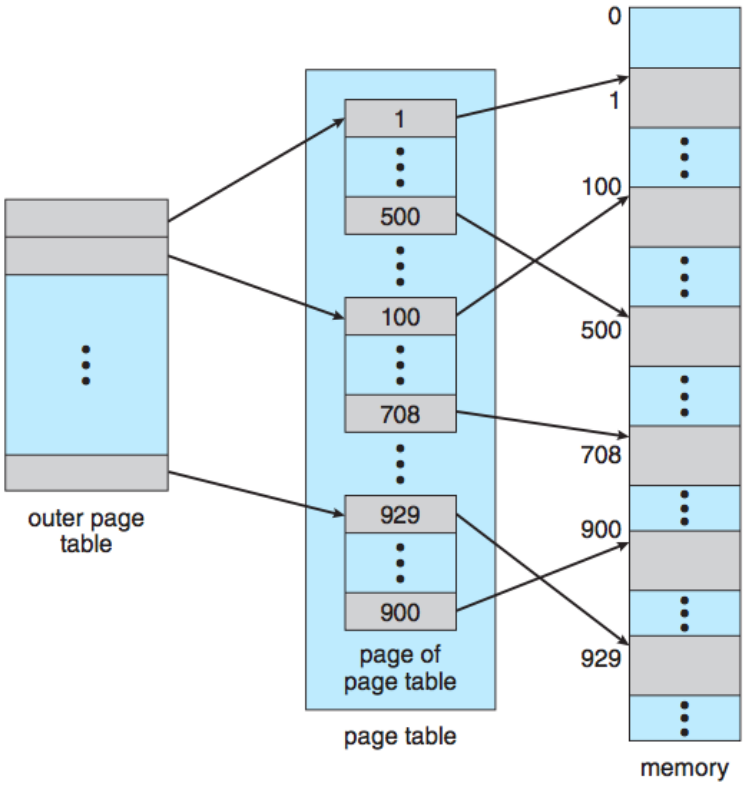
Shared pages

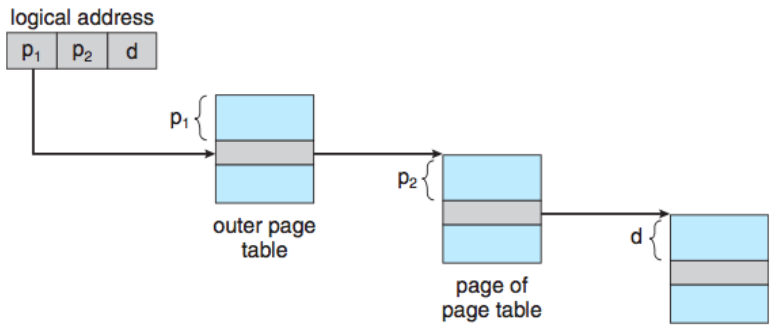


An advantage of paging is possibility of sharing common code

Hierarchical paging

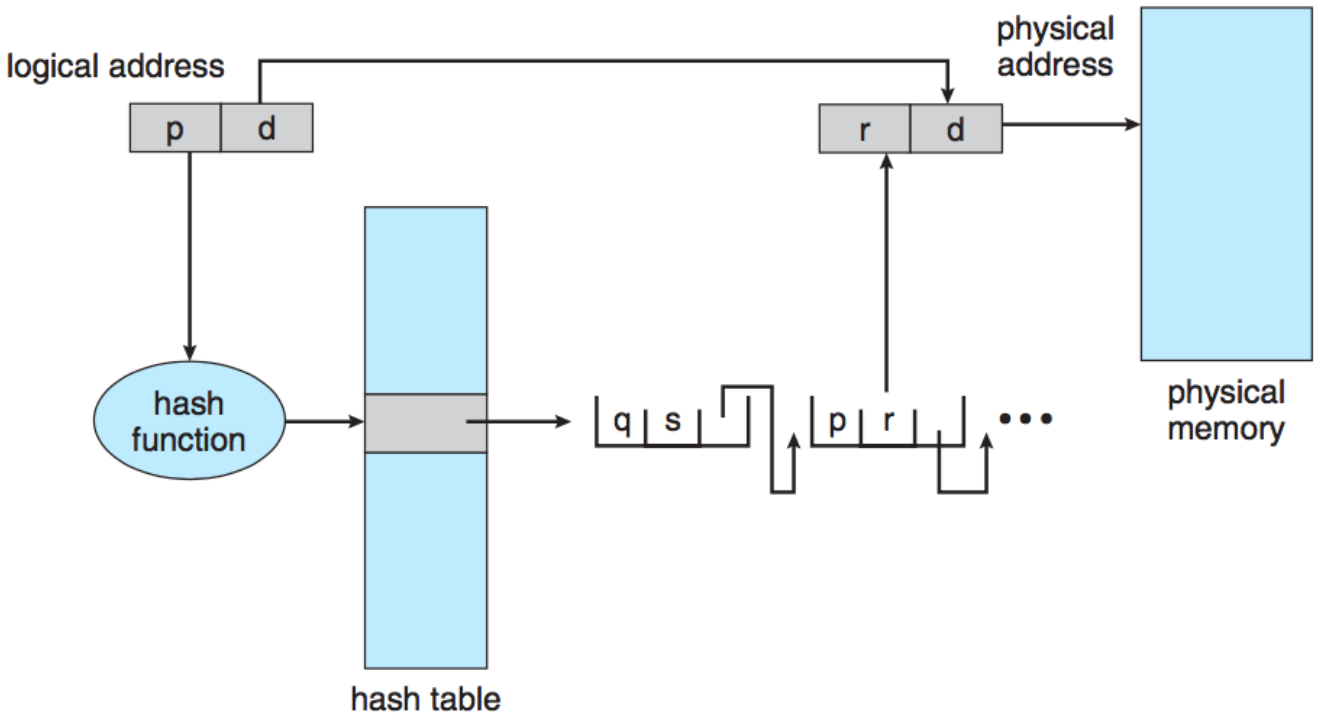
2-level paging scheme





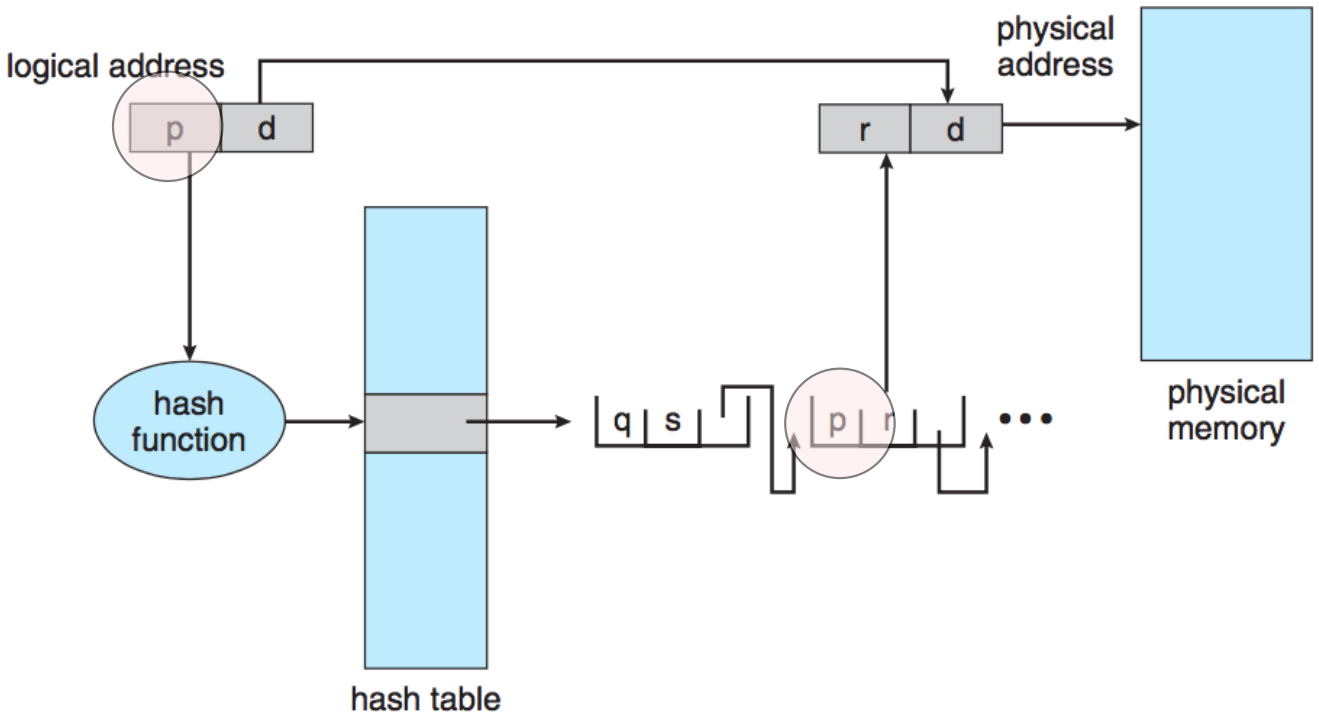
- It is often applied for 32(or less)-bit address space

Hashed paging



- It is often applied for 32(or more)-bit address space

Hashed paging



- It is often applied for 32(or more)-bit address space

Inverted paging

Just keep a page table for all processes. PID is needed to search for an address-space identifier