COMP 3511
Operating Systems

Lab 06

Outline

- Deadlocks
- Resource-Allocation Graph
- Banker's Algorithm
- Logical vs. Physical Address Space
- Segmentation
- Paging

The Deadlock Problem

- A set of blocked processes each holding a resource while waiting to acquire a resource held by another process in the set.
- Example 1
  - A system has 2 tape drives.
  - P_1 and P_2 each hold one tape drive and each needs another one.
- Example 2
  - semaphores *A* and *B*, initialized to 1

P_0	P_1
wait (A);	*wait(B)*
wait (B);	*wait(A)*

Deadlock Characterization

- Deadlock can arise if four conditions hold simultaneously
- Mutual exclusion
  - only one process at a time can use a resource.
- Hold and wait
  - a process holding at least one resource is waiting to acquire additional resources held by other processes.
- No preemption
  - a resource can be released only voluntarily by the process holding it, after that process has completed its task.
- Circular wait
  - there exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

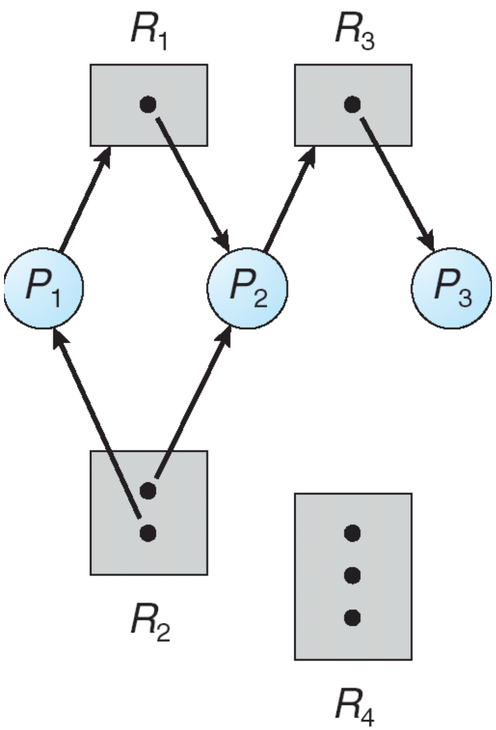
Outline

- Deadlocks
- Resource-Allocation Graph
- Banker's Algorithm
- Logical vs. Physical Address Space
- Segmentation
- Paging

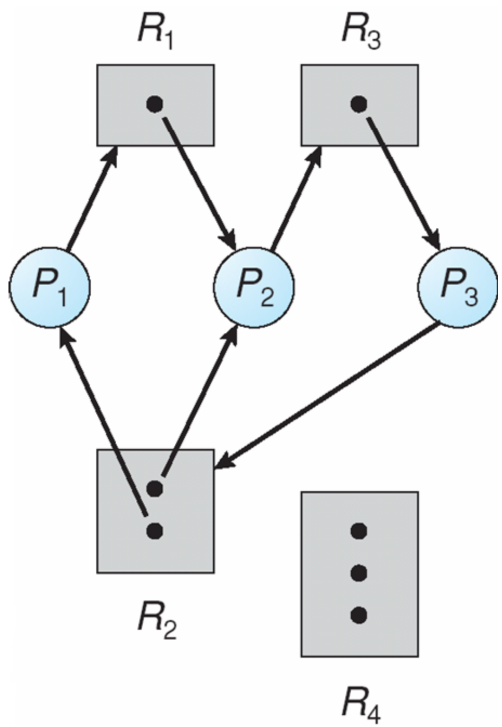
Resource-Allocation Graph

- A set of vertices V and a set of edges E.
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows: request, use, release
- Request edge – directed edge $P_i \rightarrow R_j$
- Assignment edge – directed edge $R_j \rightarrow P_i$

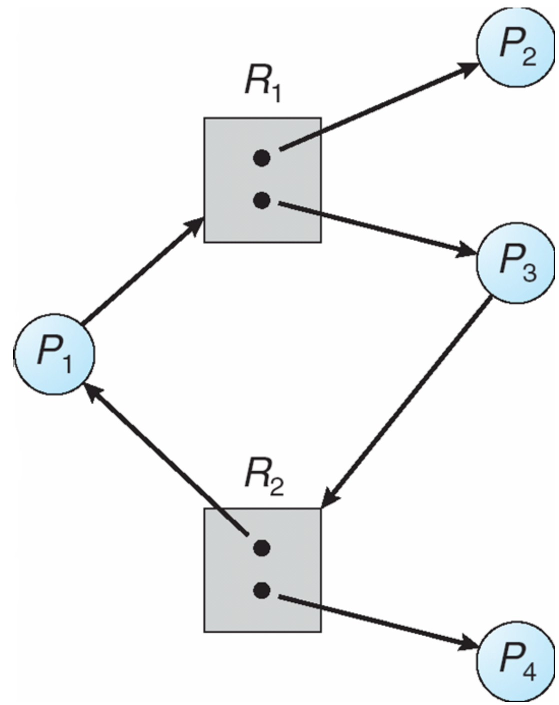
Resource Allocation Graph: Examples



A resource allocation graph with no cycle no deadlock



A resource allocation graph with a deadlock



A resource allocation graph with a cycle but no deadlock

Outline

- Deadlocks
- Resource-Allocation Graph
- Banker's Algorithm
- Logical vs. Physical Address Space
- Segmentation
- Paging

Banker's Algorithm

- Multiple instances
- Each process must in priori claim maximum use
- When a process requests a resource it may have to wait
- When a process gets all its resources it must return them in a finite amount of time

Data Structures for the Banker's Algorithm

Let n = number of processes, m = number of resources types.

- **Available***:* Vector of length m. If available $[j] = k$, there are k instances of resource type R_j available
- **Max***:* n x m matrix. If $Max\,[i,j] = k$, then process P_i may request at most k instances of resource type R_j
- **Allocation***:* n x m matrix. If Allocation$[i,j] = k$ then P_i is currently allocated k instances of R_j
- **Need***:* n x m matrix. If $Need[i,j] = k$, then P_i may need k more instances of R_j to complete its task

  $Need\,[i,j] = Max[i,j] – Allocation\,[i,j]$

Safety Algorithm

1. Let ***Work*** and ***Finish*** be vectors of length *m* and *n*, respectively. Initialize:

    Work* = *Available

    ***Finish* [*i*] = *false* for *i* = 0, 1, …, *n*- 1**

2. Find an ***i*** such that both:

    (a) ***Finish* [*i*] = *false***

    (b) ***Need*$_i$ ≤ *Work***

    If no such ***i*** exists, go to step 4

3. ***Work* = *Work* + *Allocation*$_i$**

    Finish*[*i*] = *true

    go to step 2

4. If ***Finish* [*i*] == *true*** for all ***i***, then the system is in a safe state

Resource-Request Algorithm for Process P_i

Request = request vector for process P_i.

If $Request_i[j] = k$ then process P_i wants k instances of resource type R_j

1. If $Request_i \le Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If $Request_i \le Available$, go to step 3. Otherwise P_i must wait, since resources are not available
3. Pretend to allocate requested resources to P_i by modifying the state as follows:

$$Available = Available - Request;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

- If safe → the resources are allocated to ***Pi***
- If unsafe → ***Pi*** must wait, and the old resource-allocation state is restored

Example of Banker's Algorithm

- 5 processes P_0 through P_4;
- 3 resource types: A (10 instances), B (5 instances), and C (7 instances)
- Snapshot at time T_0:

	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Example (Cont.)

- The content of the matrix ***Need*** is defined to be ***Max – Allocation***

	Need
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

- The system is in a safe state since the sequence < P_1, P_3, P_4, P_2, P_0> satisfies safety criteria

Example: P_1 Request (1,0,2)

- Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2))

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Executing safety algorithm shows that sequence < P_1, P_3, P_4, P_0, P_2> satisfies safety requirement
- Can request for (3,3,0) by P_4 be granted? Can request for (0,2,0) by P_0 be granted?

Outline

- Deadlocks
- Resource-Allocation Graph
- Banker’s Algorithm
- Logical vs. Physical Address Space
- Segmentation
- Paging

Motivation of virtual memory

- Should an entire process be in memory before it can execute?
  - In fact, real programs show us that, in many cases, the entire program is not needed
    - *e.g., figure in the next slide*
  - Even in those cases where the entire program is needed, it may not all be needed at the same time

Arrays, lists, and tables are often allocated more memory than actual need, e.g., maybe only 10×10 elements are actually used.

Since these errors seldom, if ever, occur in practice, this code is almost never executed.

A Program

Initialization

...

Array M[100][100]

...

...

Code to handle unusual error conditions

...

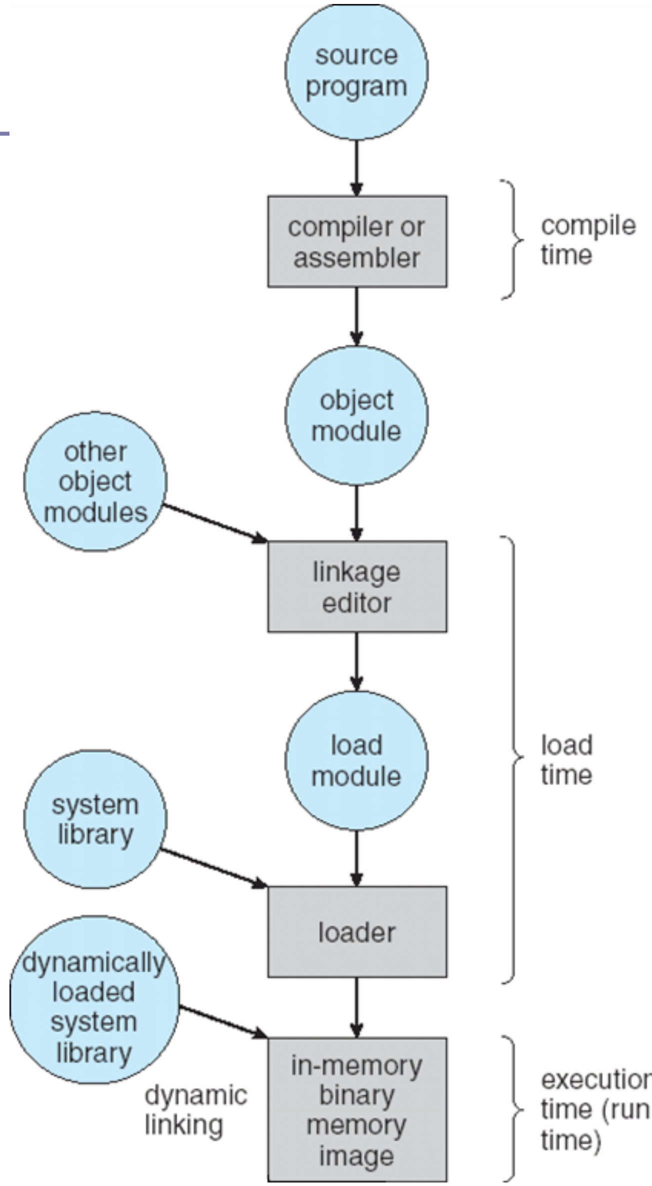
Motivation of virtual memory

- Virtual memory benefits both the system and the user
  - *Logical address space can be much larger than physical address space*
  - *A program would no longer be constrained by the amount of available physical memory*

Logical vs. Physical Address Space

- **Logical address** (also referred to as *virtual address*)
  - address seen by the CPU (application on the CPU)
- **Physical address**
  - actual address seen by the memory unit
- The user program deals with *logical* addresses; it never sees the *real* physical addresses
  - They are the same for compile-time and load-time address binding
  - They are different for execution-time address-binding

Address binding can happen at three different stages



Compile time:

- Address in the source program are generally symbolic, e.g., *count*.
- A compiler typically bind these symbolic addresses to relocatable addresses, e.g., "14B from the beginning of this module".

Load time:

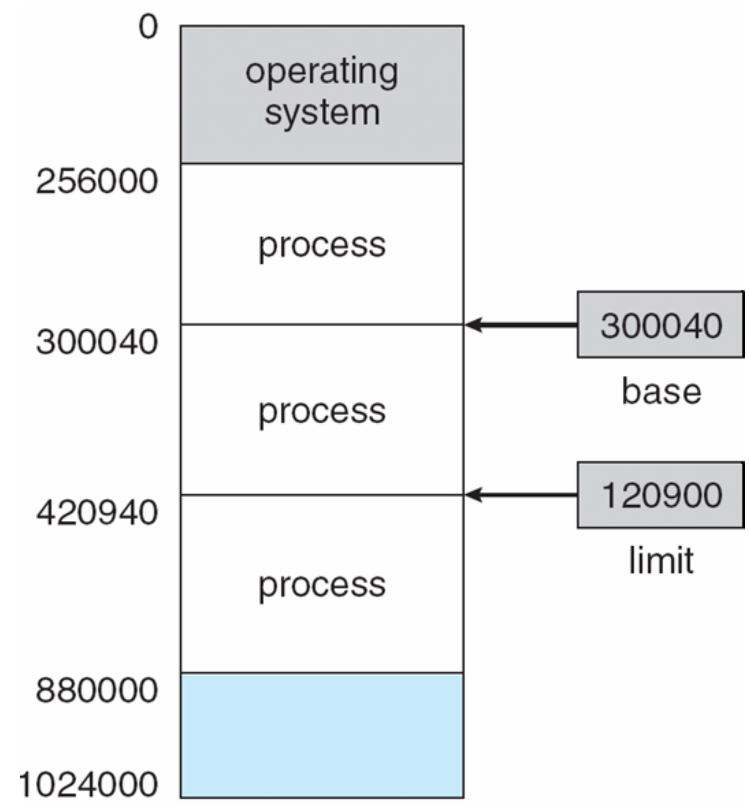
Bind the relocatable addresses to absolute address, e.g., 74014.

Execution time:

Binding delayed until run time if the process can be moved during execution from one memory segment to another.

Base and Limit Registers

- Two special registers, base and limit are used to prevent user from straying outside the designated area
- During context switch, OS loads new base and limit register from TCB
- User is NOT allowed to change the base and limit registers (privileged instructions)



Contiguous memory allocation

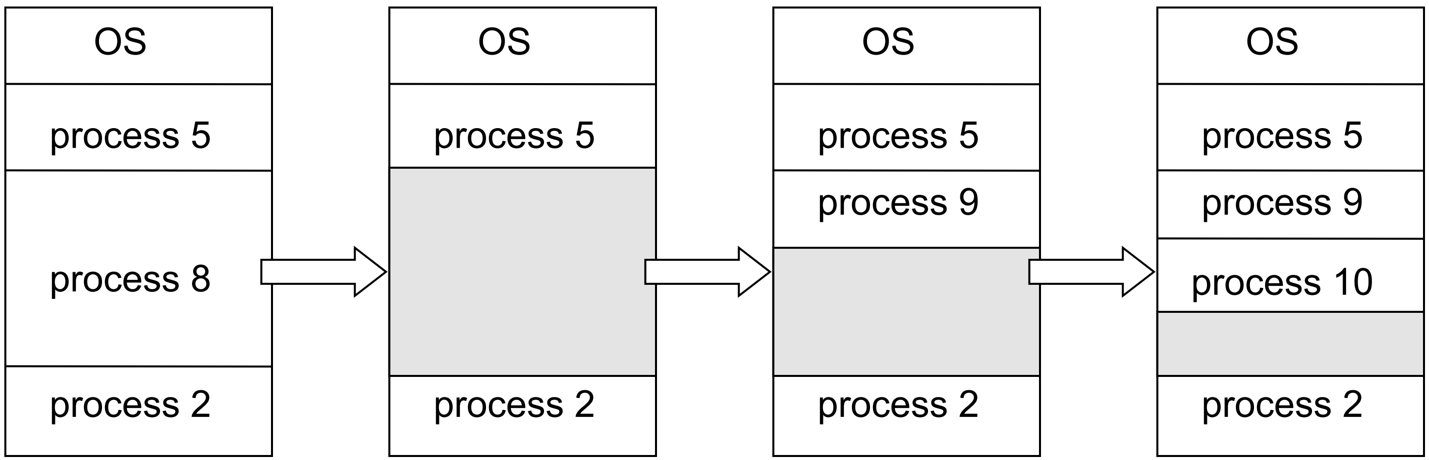
- Each process is contained in a single contiguous section of memory

  - **Hole:** block of available memory
    - holes of various size are scattered throughout memory

  - Operating system maintains information about
    - a) allocated partitions
    - b) free partitions (hole)

Contiguous memory allocation

- When a process arrives, it is allocated memory from a hole large enough to accommodate it



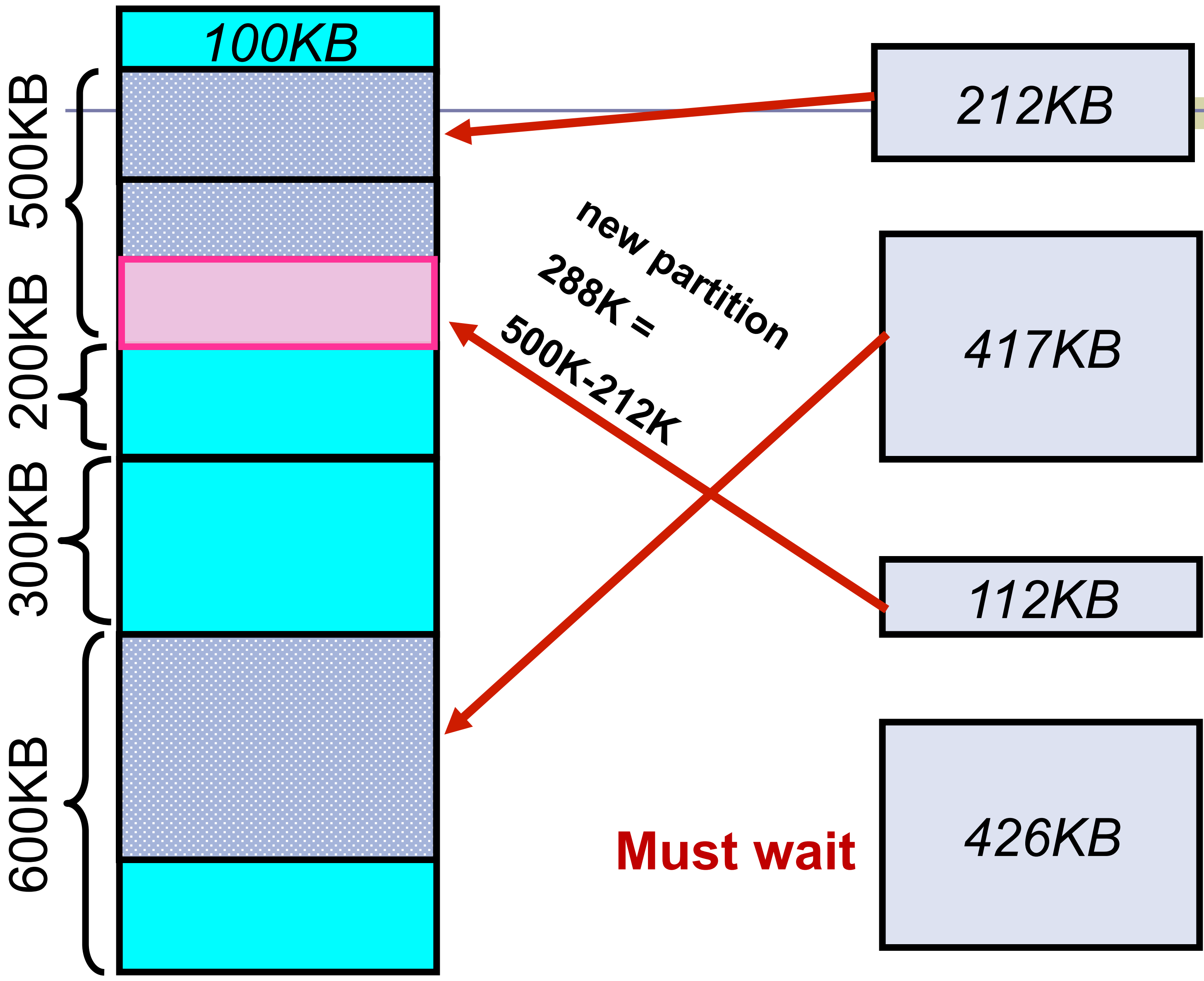
An example of First-fit, Best-fit, and Worst-fit

- **First-fit**
  - Allocate the *first* hole that is big enough
- **Best-fit**
  - allocate the *smallest* hole that is big enough
  - must search entire list, unless ordered by size
  - produces the smallest leftover hole
- **Worst-fit**
  - allocate the *largest* hole; must also search entire list
  - produces the largest leftover hole

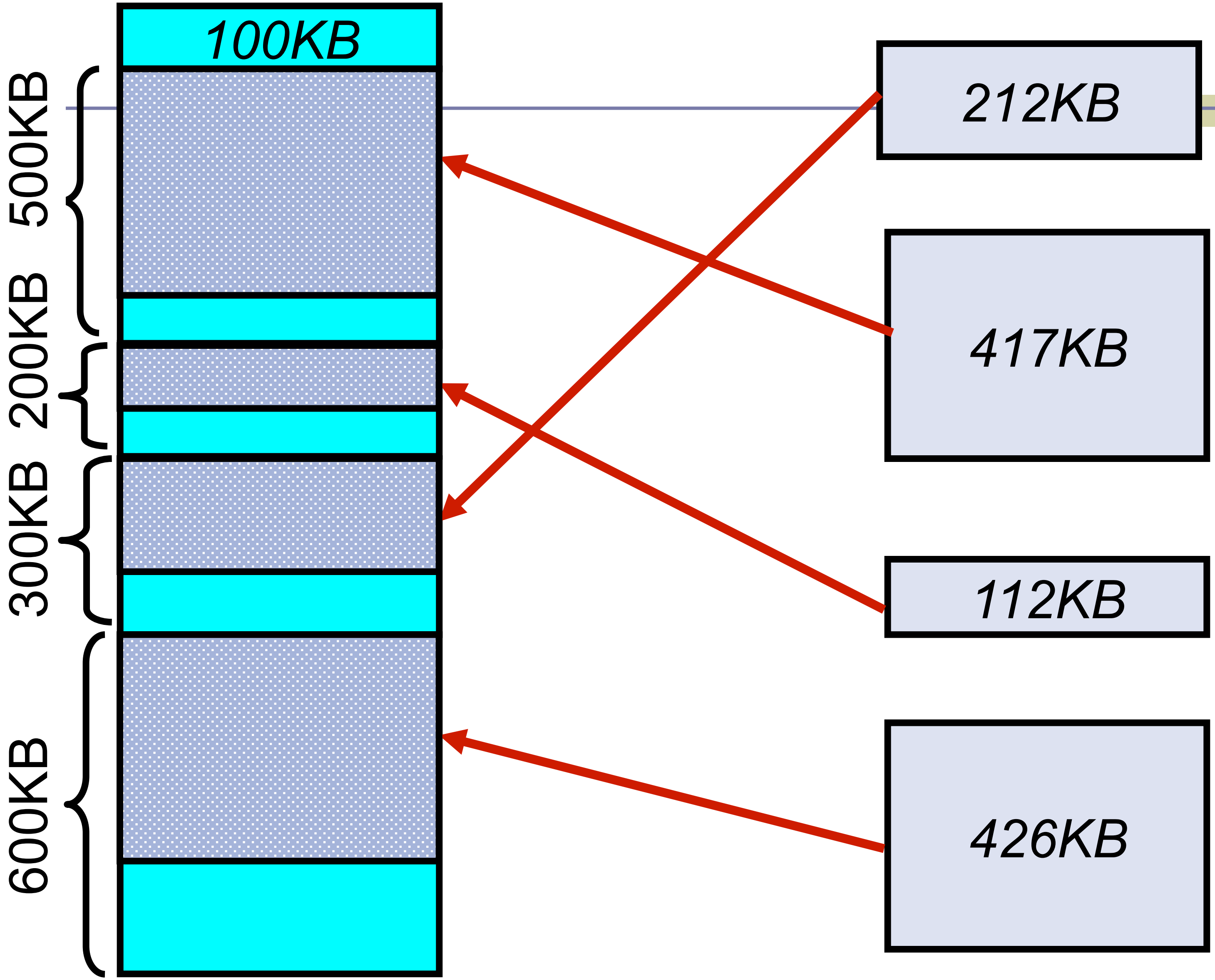
An example of First-fit, Best-fit, and Worst-fit

- Given five memory partitions of *100 KB, 500 KB, 200 KB, 300 KB, and 600 KB (in order)*
- How would each of the first-fit, best-fit, and worst-fit algorithms place processes of *212 KB, 417 KB, 112 KB, and 426 KB (in order)*?
- Which algorithm makes the most efficient use of memory?

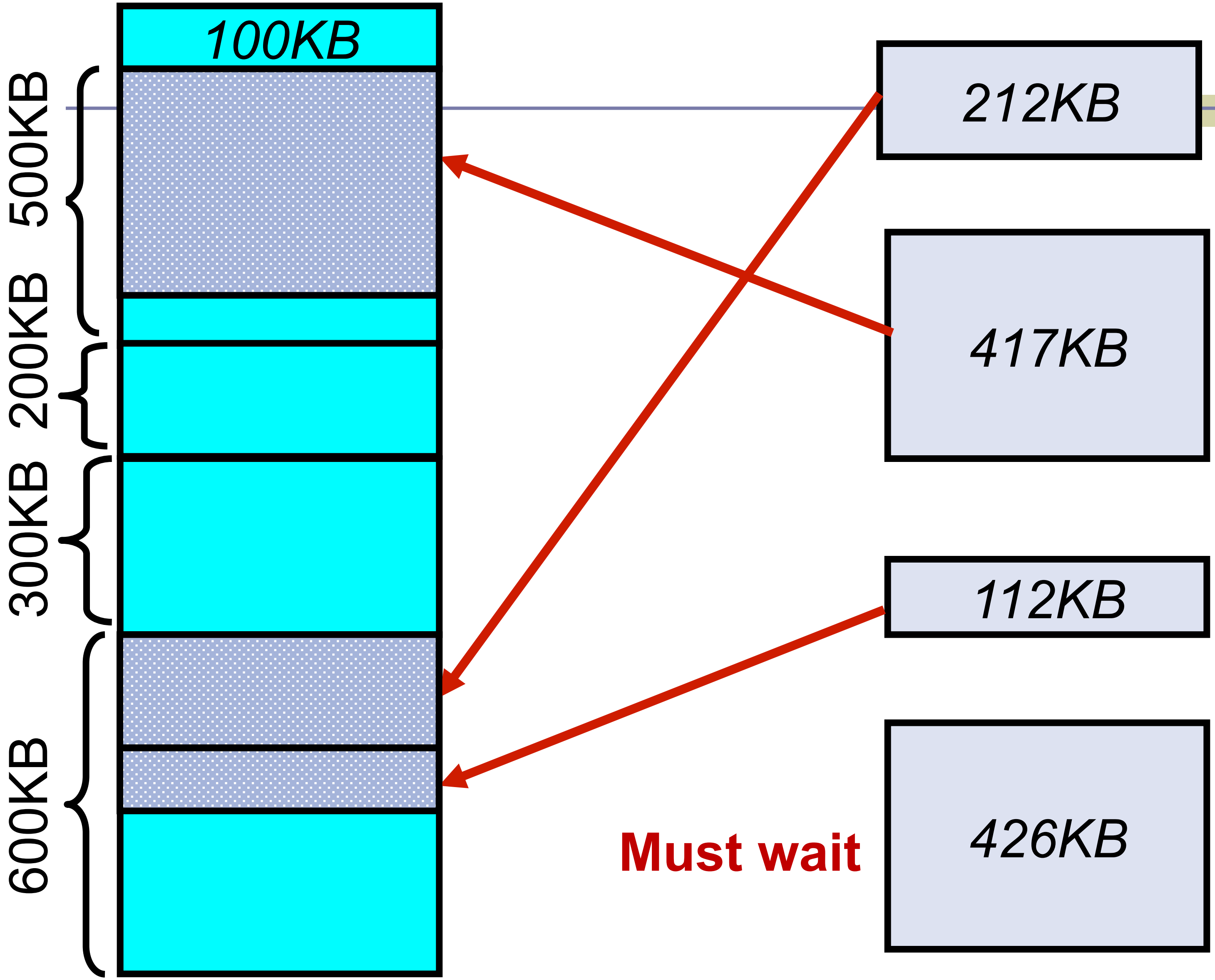
First-fit


100KB
500KB
200KB
300KB
600KB
212KB
417KB
112KB
426KB
new partition
288K =
500K-212K
Must wait

Best-fit



Worst-fit



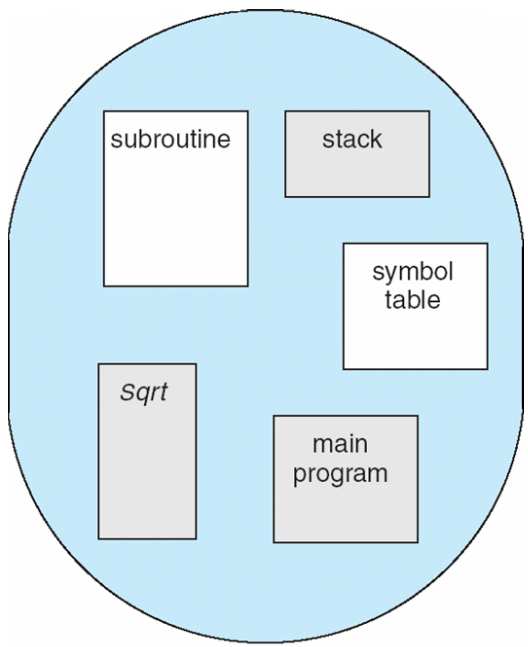
100KB
500KB
200KB
300KB
600KB
212KB
417KB
112KB
Must wait
426KB

Outline

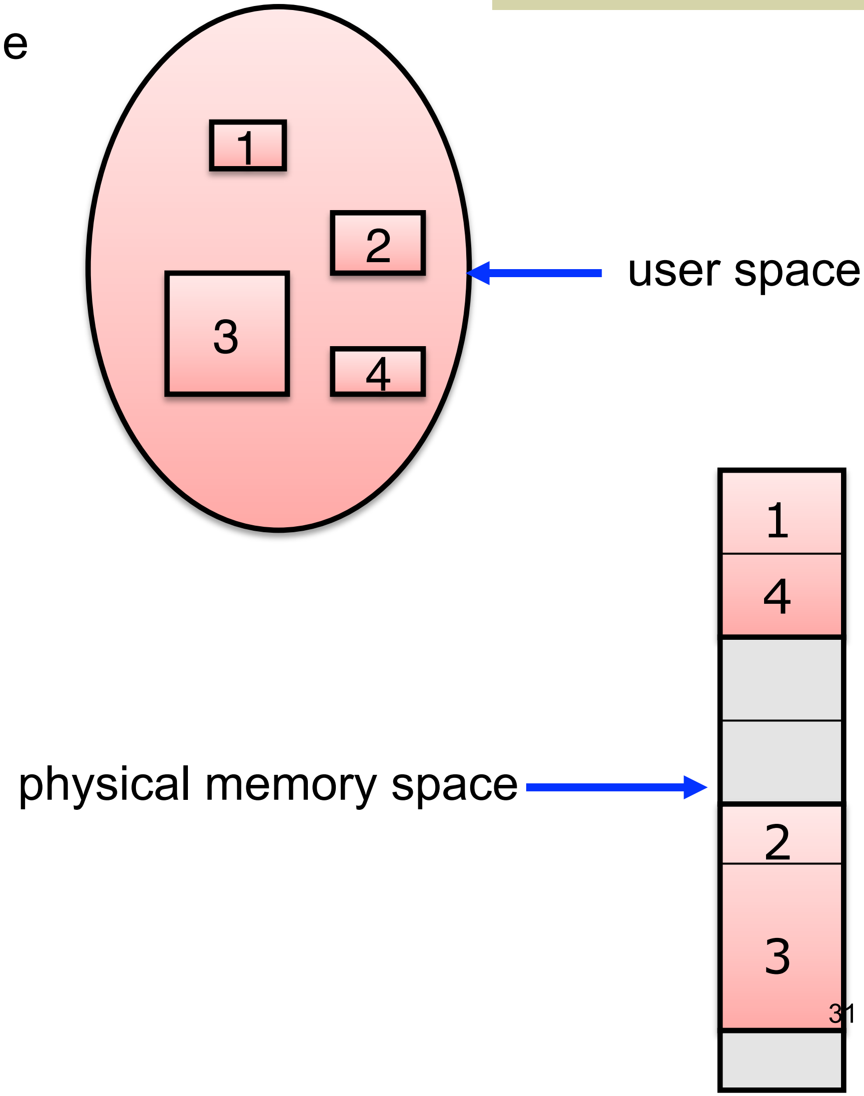
- Deadlocks
- Resource-Allocation Graph
- Banker's Algorithm
- Logical vs. Physical Address Space
- Segmentation
- Paging

Segmentation

- Memory-management scheme that supports user view of memory
- A program is a collection of segments of different sizes
- A segment is a logical unit



logical address



Segmentation

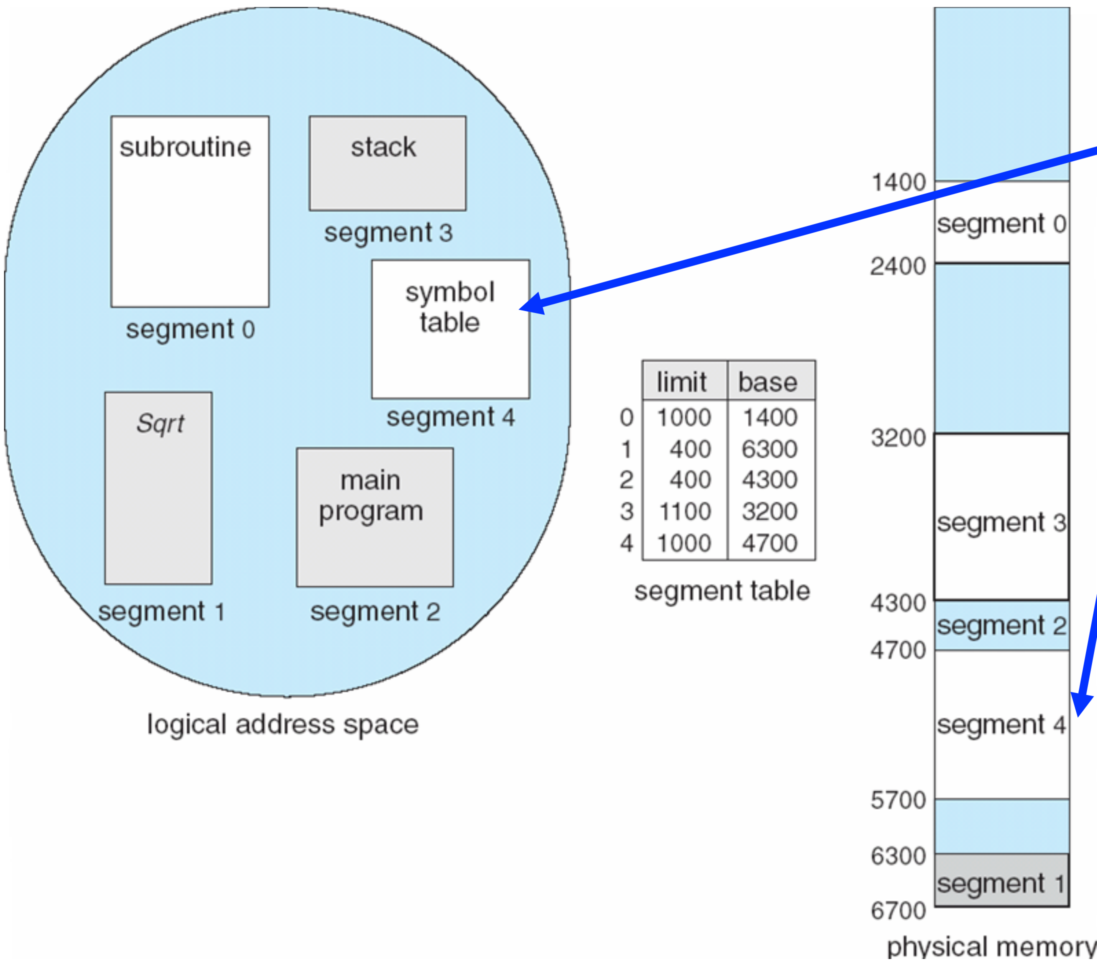
- Logical address consists of a two tuple:

  $\langle$segment-number, offset$\rangle$

- **Segment table:** maps two-dimensional physical addresses
  - base – contains the starting physical address
  - limit – specifies the length of the segment

- Problems with segmentation
  - Must fit variable-sized segments into physical memory
  - Might need to move process multiple times in order to fit everything

Example of Segmentation



	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

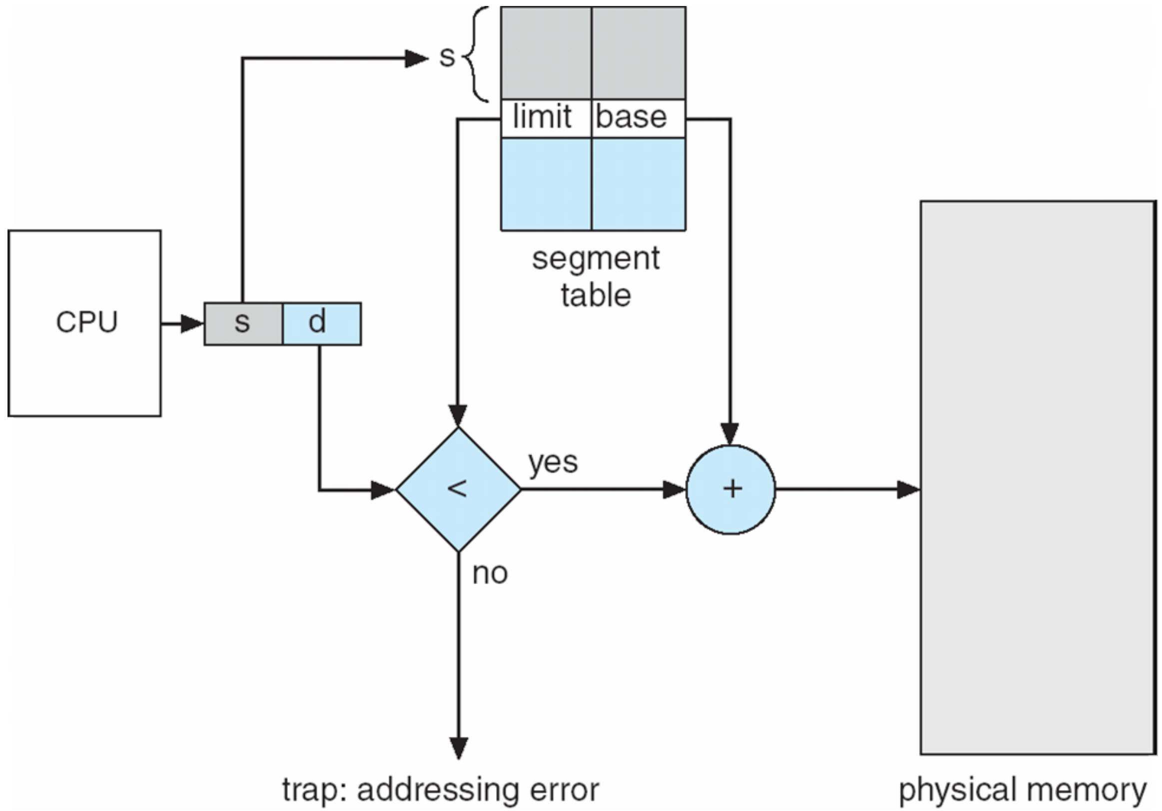
segment table

Logical view: multiple separate segments

Each segment is allocated with a contiguous memory

External fragmentation

Address Translation



Example of Segmentation

- Consider the following segment table

Segment	Base	Length
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96

- What are the physical addresses for the following logical addresses?

a. 0,430 b. 1,10

c. 2,500 d. 3,400 e. 4,112

Example of Segmentation

- Answer
- a. 219 + 430 = 649
- b. 2300 + 10 = 2310
- c. Illegal reference, trap to operating system
- d. 1327 + 400 = 1727
- e. Illegal reference, trap to operating system

Outline

- Deadlocks
- Resource-Allocation Graph
- Banker’s Algorithm
- Logical vs. Physical Address Space
- Segmentation
- Paging

Paging

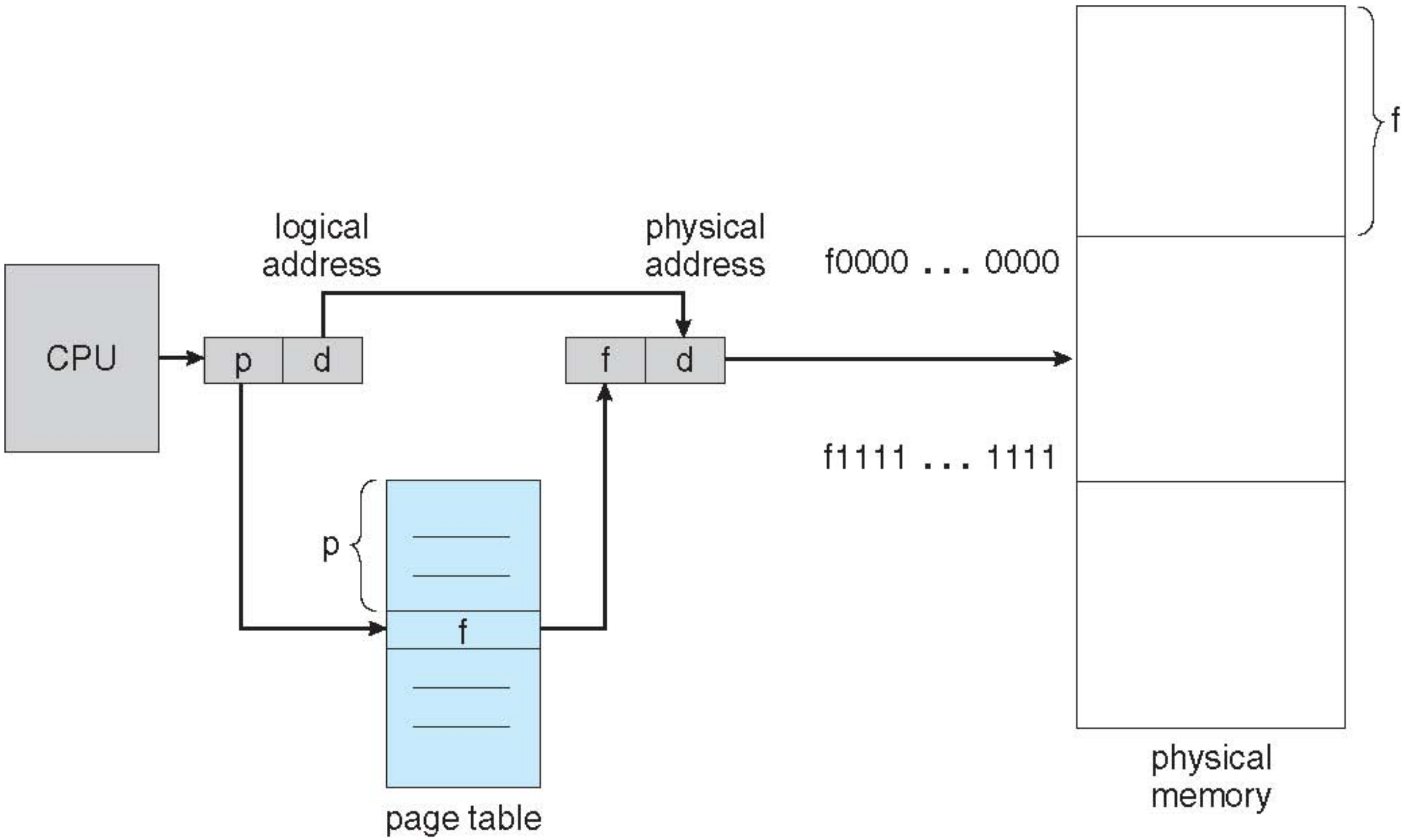
- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
  - Avoids external fragmentation
  - Avoids problem of varying sized memory chunks
- Divide physical memory into fixed-sized blocks called **frames**
  - Size is power of 2, between 512 bytes and 16 Mbytes
- Divide logical memory into blocks of same size called **pages**
- Keep track of all free frames
- To run a program of size ***N*** pages, need to find ***N*** free frames and load program
- Set up a **page table** to translate logical to physical addresses
- Still have Internal fragmentation

Address Translation Scheme

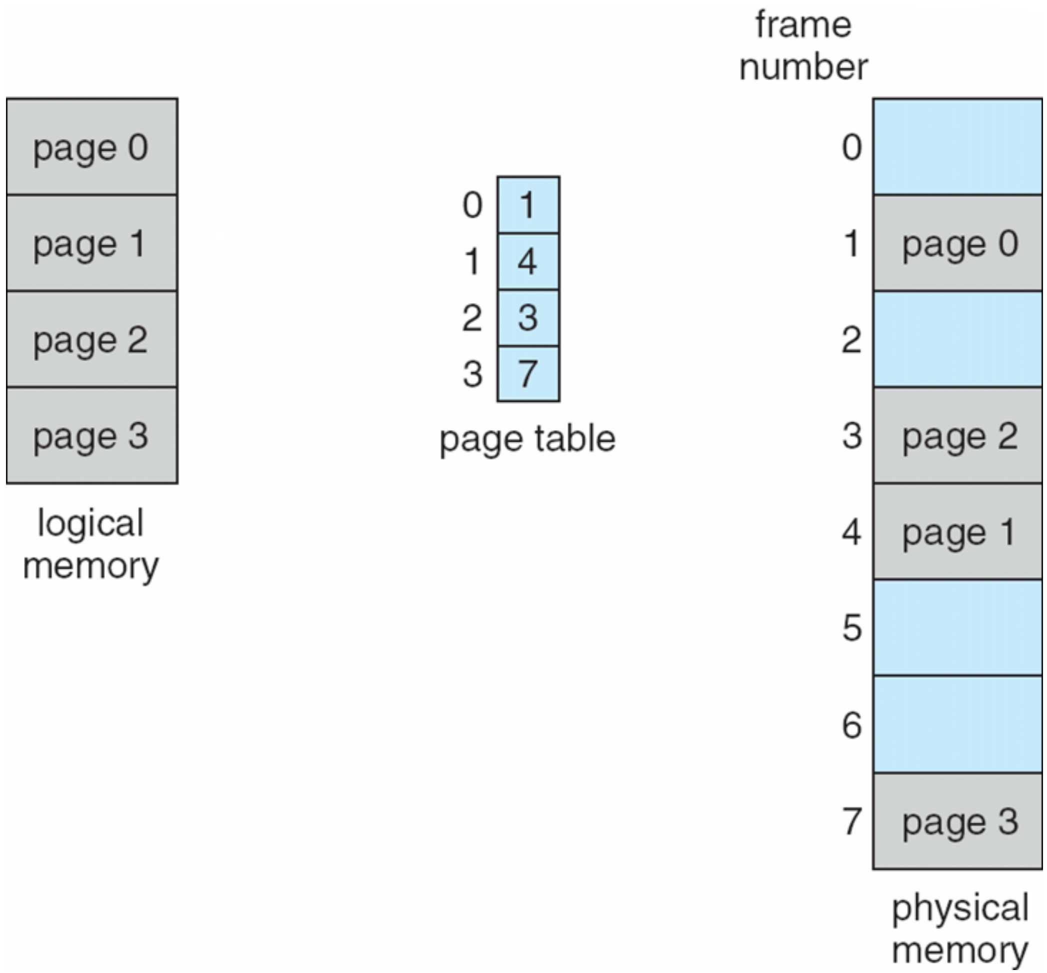
- Address generated by CPU is divided into:
  - **Page number** (***p***) – used as an index into a **page table** which contains base address of each page in physical memory
  - **Page offset** (***d***) – combined with base address to define the physical memory address that is sent to the memory unit

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

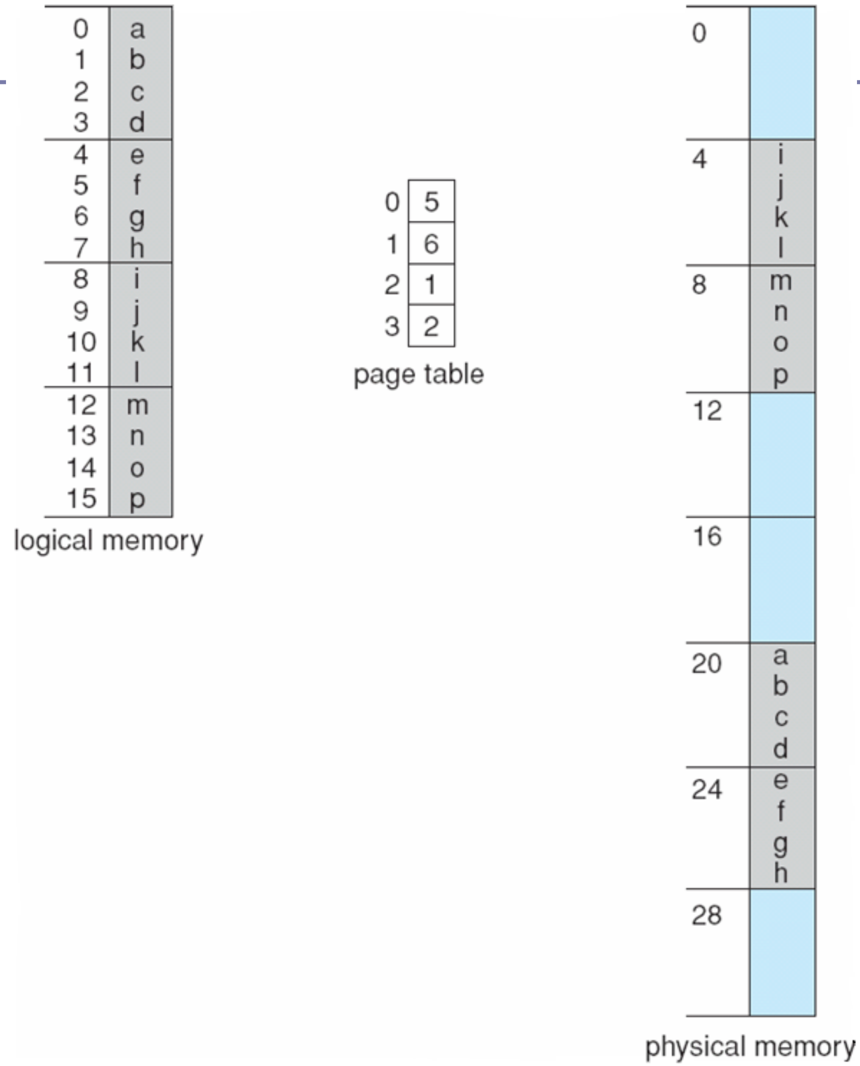
Paging Hardware



Paging Model of Logical and Physical Memory



Paging Example



Paging (Cont.)

- Calculating internal fragmentation
  - Page size = 2,048 bytes
  - Process size = 72,766 bytes
  - 35 pages + 1,086 bytes
  - Internal fragmentation of 2,048 - 1,086 = 962 bytes
  - Worst case fragmentation = 1 frame – 1 byte
  - On average fragmentation = 1 / 2 frame size
  - So small frame sizes desirable?
  - But each page table entry takes memory to track
  - Page sizes growing over time
    - Solaris supports two page sizes – 8 KB and 4 MB

COMP 3511
Operating Systems

Project #2

Objectives and Tasks

- Run Nachos with Pre-implemented Scheduling System Skeleton
- Implement SJF and Non-preemptive Priority Scheduling Algorithms
- Explain the Results

You are strongly recommended to use the servers in this lab for this project.

ssh `username`@csl2wk01(~csl2wk40).cse.ust.hk

Task 1

- **Task 1: Run Nachos with Pre-implemented Scheduling System Skeleton**
  - Step 1: Download Nachos source code of this project
  - Step 2: Extract the source code
  - Step 3: Compile the code
  - Step 4: Run nachos
  - Step 5: Read the code

Task 1

- Three scheduling algorithms
  - First Come First Serve (FCFS)
  - Shortest Job First (SJF)
  - Non-Preemptive Priority (NP_Priority)

Task 1

Executable File	Source File	Corresponding Algorithm	Already Implemented?
test0	test.0.cc	FCFS	Yes
test1	test.1.cc	SJF	No
test2	test.2.cc	NP_Priority	No

Task 1

- *Read the codes*

- *ReadyToRun()*
  - decides the policy of placing a thread into ready queue (or multilevel queues) when the thread gets ready
- *FindNextToRun()*
  - decides the policy of picking one thread to run from the ready queue
- *ShouldISwitch*()
  - decides whether the running thread should preemptively give up to a newly forked thread

Task 2

- **Implement SJF and NP_Priority**

- Only modify *scheduler.cc*
  - Scheduler::ReadyToRun
  - Scheduler::FindNextToRun
  - Scheduler::ShouldISwitch

Task 2

- Shortest Job First
  - the thread with the shortest burst time in the ReadyList should be scheduled for running after the current thread is done with burst.
  - Return first thread when scheduler needs to pick one thread to run
  - *Hint*: insert the thread to ReadyList according to its burst time when a thread gets ready.
    - Make use of the function *SortedInsert()* in *List.cc*
    - Example:

```
list->SortedInsert(thread,thread->getBurstTime());
```

this line of code insert the thread into the list based on its burst time.

Task 2

- Non-Preemptive Priority Scheduling
  - the thread with the highest priority in the ReadyList should be scheduled for running after the current thread is done with burst.
  - Return first thread when scheduler needs to pick one thread to run
  - *Hint*: insert the thread to ReadyList according to its burst time when a thread gets ready.(Mind the order!)
    - Make use of the function *SortedInsert()* in *List.cc*
    - Example:

      list->SortedInsert(thread,thread->MAX_PRIORITY-thread->getPriority());

      this line of code insert the thread into the list based on its priority.

Task 2

- Compile and Run
- Save your outputs to project2_test1.txt and project2_test2.txt, respectively,
- Keep your source code scheduler.cc

Task 3

- Explain the Results

1. Understand the output of test0 (FCFS scheduling) , test1 (SJF scheduling) and test2 (NP_Priority). Then calculate the following performance metrics of each scheduling algorithms:
   a) Average waiting time;
   b) Response time;
   c) Turn-around time.
2. Compare the performance among the first two scheduling algorithms (FCFS and SJF) in the aspects mentioned in question 1, then discuss the pros and cons of each scheduling algorithms. (Note: you are strongly encouraged to change the input threads in *test.0.cc and test .1.cc* in order to make your discussion more convincing. However, when submitting the outputs of test1, please do submit the outputs with the original input threads.)

Outputs

- Please generate a single file using ZIP and submit it through CASS
- Name of the ZIP: "proj2_********.zip" (* as student ID)
- Inside the ZIP file:

File Name	Description
scheduler.cc	Source code you have accomplished by the end of Task2
project2_test1.txt	Output of test1
project2_test2.txt	Output of test2
project2_report.txt	The answer to the questions in Task 3