Fall 2015 COMP 3511

Operating Systems

Lab 06

Outline

- Monitor
- Deadlocks
- Logical vs. Physical Address Space
- Segmentation
- Example of segmentation scheme
- Paging
- Example of paging scheme
- Paging-Segmentation Combination

Monitors

- Motivation

  Use locks for mutual exclusion and condition variables for scheduling constraints

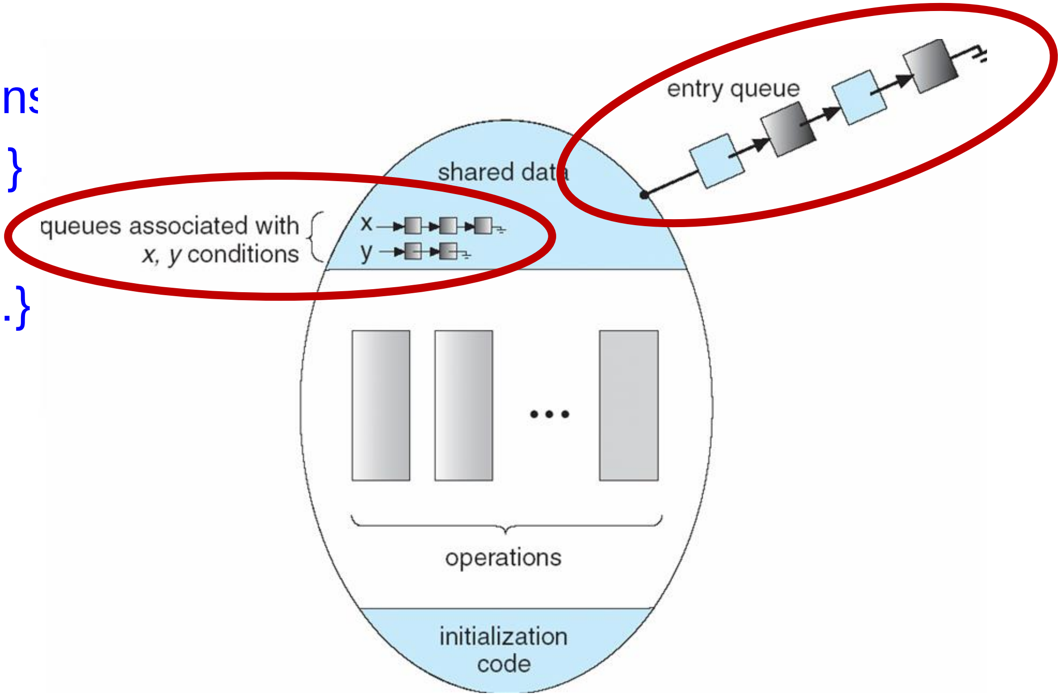
Definition

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- A lock and zero or more condition variables for managing concurrent access to shared data inside a monitor
- Only one process may be active within the monitor at a time

Monitors

```
monitor monitor-name
{
    // shared variable declarations
        procedure P1 (…) { …. }
          …
        procedure Pn (…) {……}

        Initialization code ( ….)
          { … }
          …
}
```



- Some languages like Java provide this natively
- Most commercial OS use locks and condition variables

Monitors

- Lock: the lock provides mutual exclusion to shared data
  - Always acquire before accessing shared data structure
  - Always release after finishing with shared data
  - Lock initially free
- Condition variable: a queue of threads waiting for something inside a critical section
  - Key idea: make it possible to go to sleep inside critical section by atomically releasing lock at time it goes to sleep
  - Contrast to semaphores: Cant wait on a semaphore inside critical section
- Condition variables x, y;
- Two operations on a condition variable:
  - x.wait () – a process that invokes the operation is suspended.
  - x.signal () – resumes one of processes (if any) that invoked x.wait ()

Difference between semaphore and condition

Semaphore	Condition
counting	don't count
wait: may be pass immediately (it may decrement the semaphore value without wait)	wait: alway wait (suspend the process)
signal: increase semaphore value, may wake up or may not wake up another process	signal: if there exists a process waiting, wake up. otherwise, nothing happens.

Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex;  // (initially  = 1)
semaphore next;     // (initially  = 0)
int next-count = 0;
```

- Each procedure ***F*** will be replaced by

```
wait(mutex);
    …
                    body of F;

    …
if (next_count > 0)
        signal(next)
else
        signal(mutex);
```

- Mutual exclusion within a monitor is ensured.

Monitor Implementation

- For each condition variable ***x***, we have:

```
semaphore x_sem; // (initially  = 0)
int x-count = 0;
```

- The operation x.wait can be implemented as:

```
x-count++;
if (next_count > 0)
        signal(next);
else
        signal(mutex);
wait(x_sem);
x-count--;
```

- The operation x.signal can be implemented as:

```
if (x-count > 0) {
        next_count++;
        signal(x_sem);
        wait(next);
        next_count--;
}
```

The Deadlock Problem

- A set of blocked processes each holding resource(s) while waiting to acquire more resource(s) 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)*

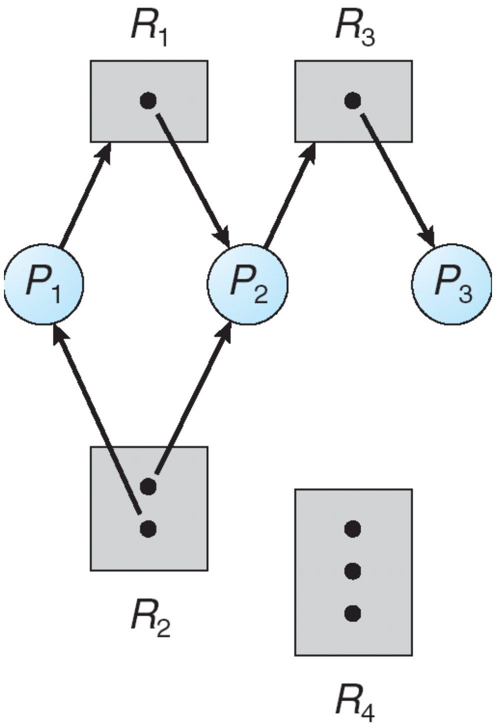
Deadlock Characterization

- If Deadlock occurs, **four** conditions must 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.

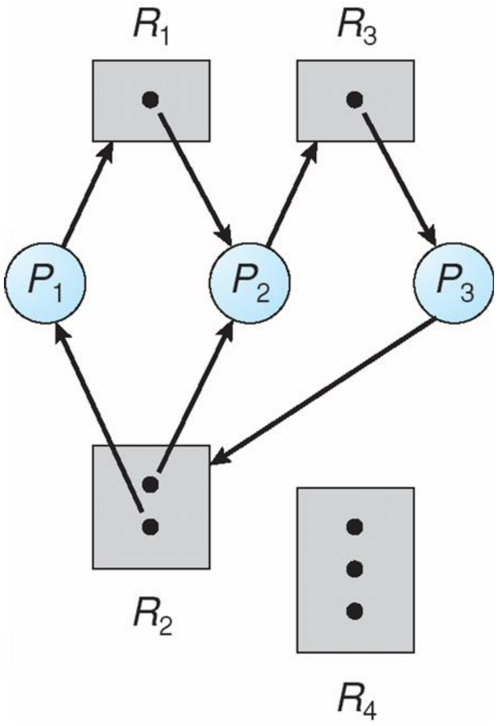
Resource-Allocation Graph

- 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 $R_j \rightarrow P_i$

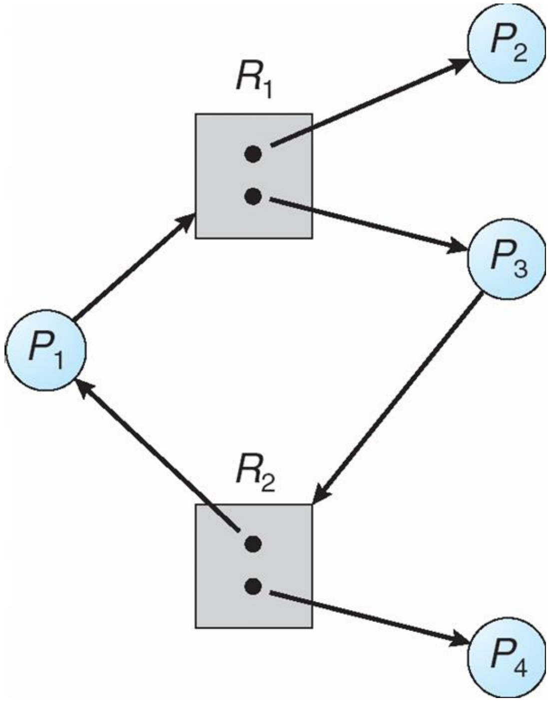
Resource Allocation Graph: Examples



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

Facts & Methods

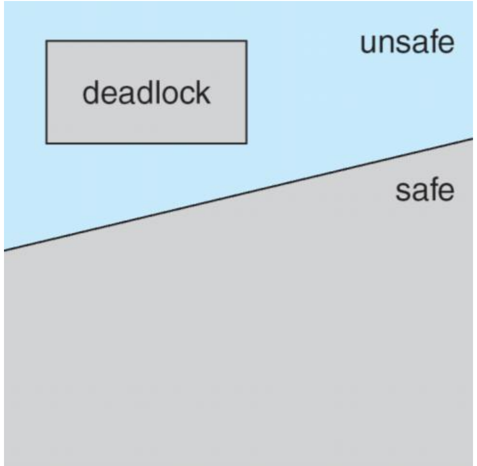
- If graph contains no cycles ⇒ no deadlock.
- If graph contains a cycle ⇒
  - if only one instance per resource type, then deadlock.
  - if several instances per resource type, possibility of deadlock.
- Deadlock Prevention: ensure that the system will *never* enter a deadlock state – expensive operations
  - Need to monitor all lock acquisitions
  - Selectively deny those that *might* lead to deadlock
- Deadlock Detection: allow the system to enter a deadlock state and then recover.
  - Requires deadlock detection algorithm
  - Technique for forcibly preempting resources and/or terminating tasks

Deadlock Prevention

- **Mutual Exclusion** – not required for sharable resources; must hold for nonsharable resources.
- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources.
  - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none.
  - Low resource utilization; starvation possible
- **No Preemption** –
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released.
  - Preempted resources are added to the list of resources for which the process is waiting.
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.
- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.

Deadlock Avoidance

- Avoidance ⇒ ensure that a system never enters an unsafe state.
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
- Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes.



- System is in safe state if there exists a safe sequence of all processes:

  Sequence $<P_1, P_2, \ldots, P_n>$ is safe if for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j<i$.
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished.
  - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate.
  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on.

Banker's Algorithm

- Each resource can have multiple instances.
- Each process must a 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.

Let n = number of processes, and 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:* $n \times m$ matrix. If *Max* $[i,j] = k$, then process P_i may request at most k instances of resource type R_j.
- *Allocation:* $n \times m$ matrix. If Allocation$[i,j] = k$ then P_i is currently allocated k instances of R_j.
- *Need:* $n \times m$ matrix. If *Need*$[i,j] = k$, then P_i may need k more instances of R_j to complete its task

$$Need\ [i,j] = Max[i,j] - Allocation\ [i,j].$$

Banker's Algorithm - Example

- Consider the following snapshot of a system

	Allocation					Max					Available			
	A	**B**	**C**	**D**		**A**	**B**	**C**	**D**		**A**	**B**	**C**	**D**
P0	0	0	1	2		0	0	3	2		2	1	2	0
P1	2	0	0	0		2	7	5	0					
P2	0	0	3	4		6	6	5	6					
P3	2	3	5	4		4	3	5	6					
P4	0	3	3	2		0	6	5	2					

Banker's Algorithm - Example

What is the content of the matrix *Need* denoting the number of resources needed by each process?

Max – Allocation = Need (maxtrix)

	Need			
	A	B	C	D
P0	0	0	2	0
P1	0	7	5	0
P2	6	6	2	2
P3	2	0	0	2
P4	0	3	2	0

Banker's Algorithm - Example

- Is the system in a safe state? Why?
  - The allocation should be safe right now, with a sequence of process execution.
  - Yes, with <P0, P3, P4, P1, P2>

	Resources available after each process finished			
	A	**B**	**C**	**D**
P0	2	1	3	2
P3	4	4	8	6
P4	4	7	11	8
P1	6	7	11	8
P2	6	7	14	12

Banker's Algorithm - Example

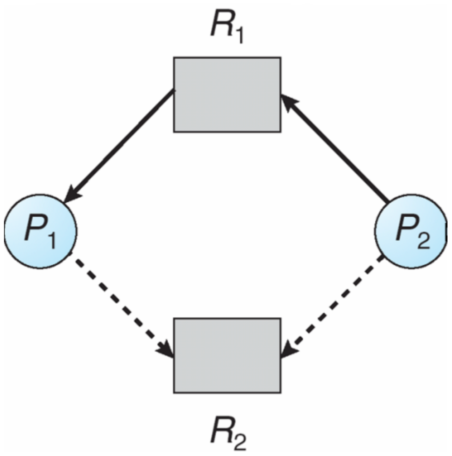
If a request from process P2 arrives for (0, 1, 2, 0), can the requested be granted immediately? Why?

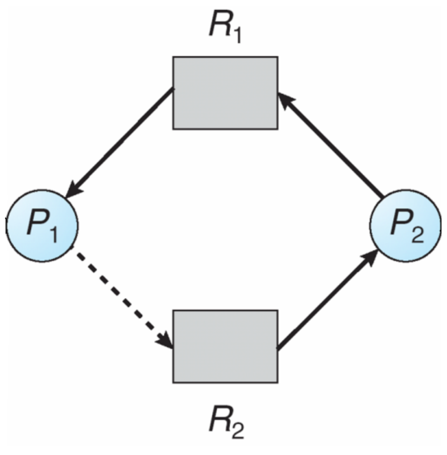
No, this can not be allocated.

If this is allocated, the resulting Available() is (2, 0, 0, 0), there is no sequence of the process execution order that lead to the completion of all processes. This is an unsafe state.

Resource-Allocation Graph Algorithm

- *Claim edge Pi* → *Rj* indicated that process *Pi* may request resource *Rj*; represented by a dashed line.
- Claim edge converts to request edge when a process requests a resource.
- When a resource is released by a process, assignment edge reconverts to a claim edge.
- Resources must be claimed *a priori* in the system.





Example

- A system is composed of four processes

  {P1, P2, P3, P4}

- And three types of resources

  {R1, R2, R3}

- The number of units of the resources are

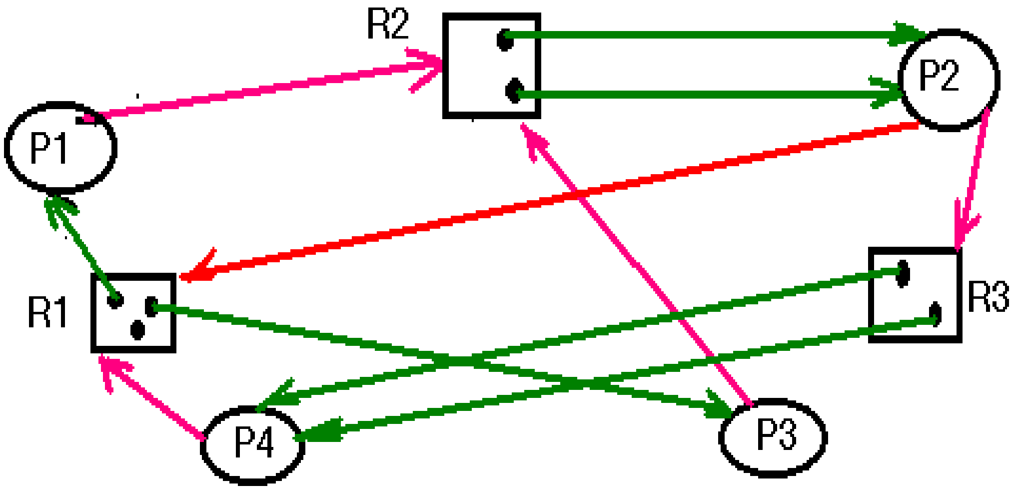
  C= <3, 2, 2>

- System state
  - *process P1 holds 1 unit of R1 and requests 1 unit of R2.*
  - *P2 holds 2 units of R2 and requests 1 unit each of R1 and R3.*
  - *P3 holds 1 unit of R1 and requests 1 unit of R2.*
  - *P4 holds 2 units of R3 and requests 1 unit of R1.*

- Show the resource graph to represent the system state.

- Consider a sequence of processes executions without deadlock.

Example

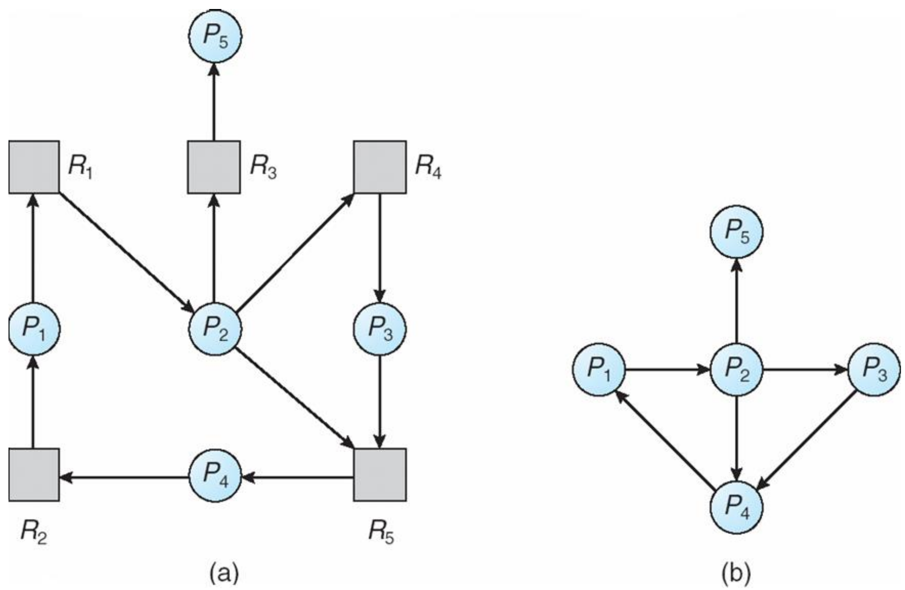


- Sequence of processes executions is
  - P4 gets the unit of R1 and finishes,
  - P2 gets 1 unit of R1 and 1 unit of R3 and finishes,
  - then P1 and P3 can finish.
- There is no deadlock in this situation.

Deadlock Detection

Maintain wait-for graph if each resource has a single instance

Periodically invoke an algorithm that searches for a cycle in the graph. If there is a *cycle* => a deadlock



Resource-Allocation Graph Corresponding wait-for graph

Detection Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively Initialize:
   (a) *Work* = *Available*
   (b) For *i* = 1,2, …, *n*, if $Allocation_i \neq 0$, then *Finish*[i] = false;otherwise, *Finish*[i] = *true*.
2. Find an index *i* such that both:
   (a) *Finish*[*i*] == *false*
   (b) $Request_i \leq Work$

   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*] == false, for some *i*, $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if *Finish*[*i*] == *false*, then P_i is deadlocked.

Algorithm requires an order of $O(m \times n^2)$ operations to detect whether the system is in deadlocked state.

Example of Detection Algorithm

- Five processes P_0 through P_4; three resource types A (7 instances), *B* (2 instances), and *C* (6 instances).
- Snapshot at time T_0:

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- Sequence <P_0, P_2, P_3, P_1, P_4> will result in *Finish*[*i*] = true for all *i*.

Example (Cont.)

- P_2 requests an additional instance of type C.

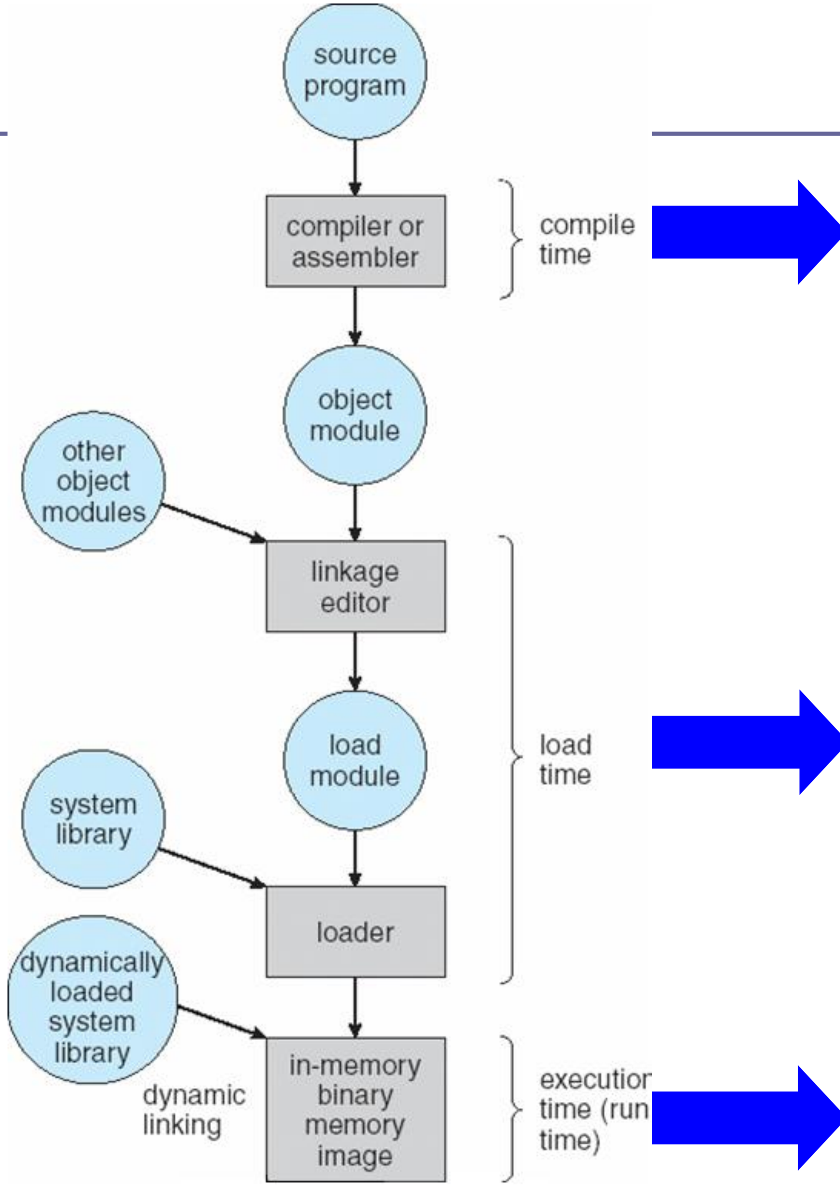
	Request
	A B C
P_0	0 0 0
P_1	2 0 1
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

- State of system?
  - Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes; requests.
  - Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4.

Logical vs. Physical Address Space

- **Logical address** (also referred to as virtual address)
  - address seen by 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:

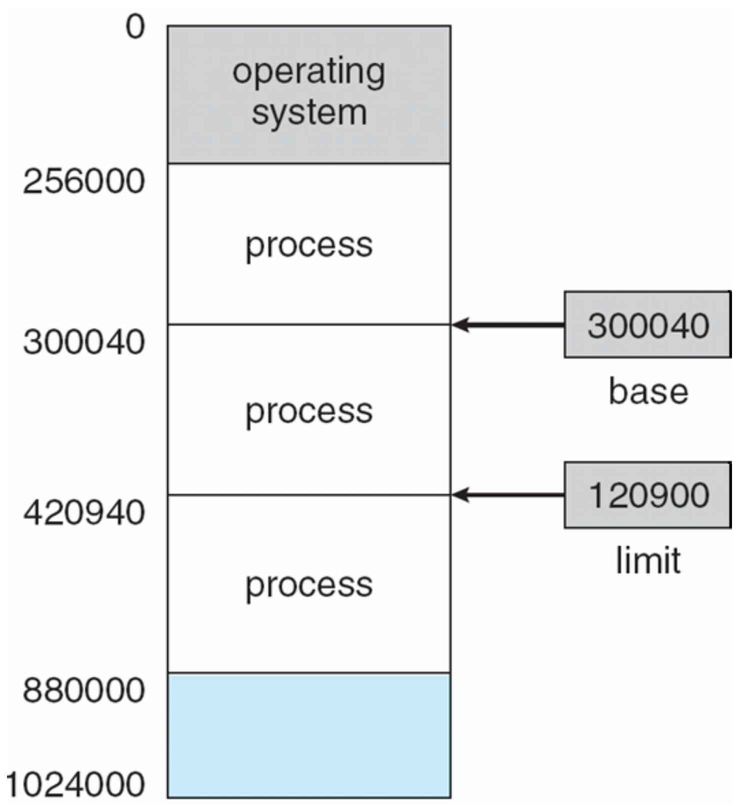
Binding delayed until run time if the process can be moved during execution from one memory segment to another.

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)

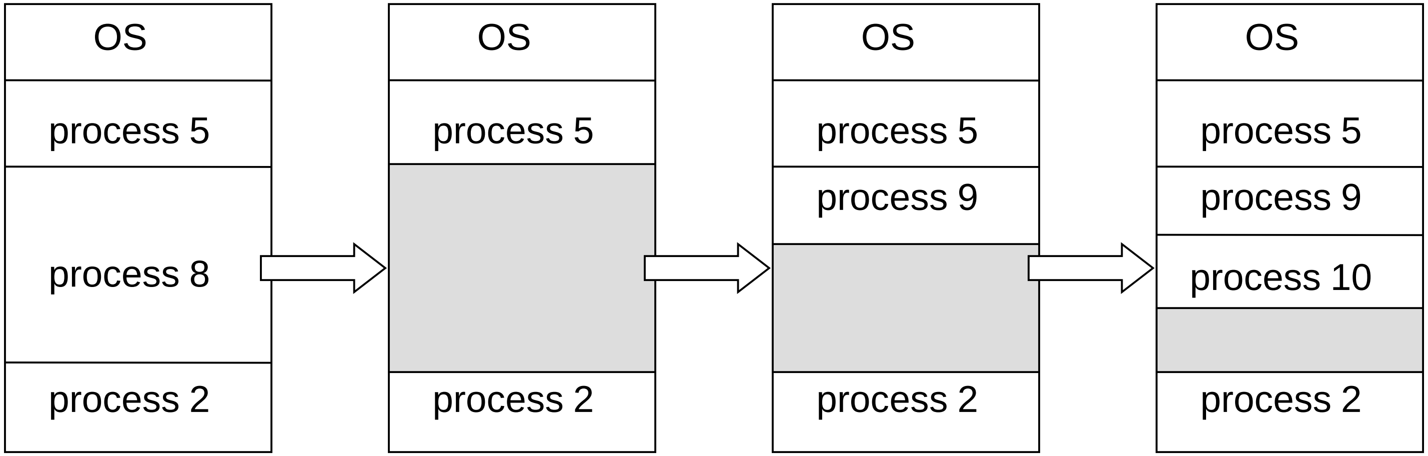
Base and Limit Registers

- 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

- 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

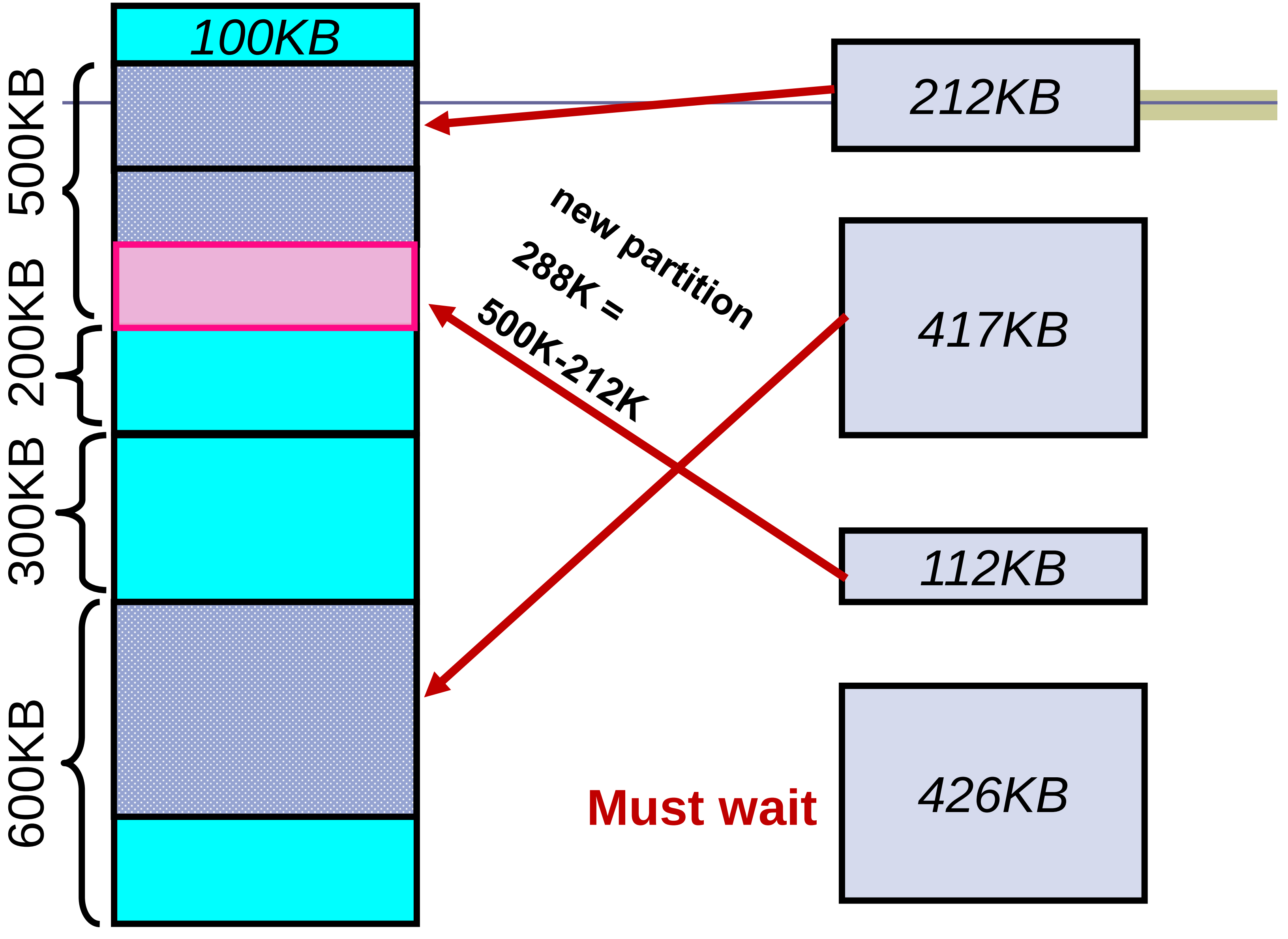
- **Best-fit**
  - allocate the *smallest* hole that is big enough
  - must search entire list, unless ordered by size
  - produces the smallest leftover hole

- **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

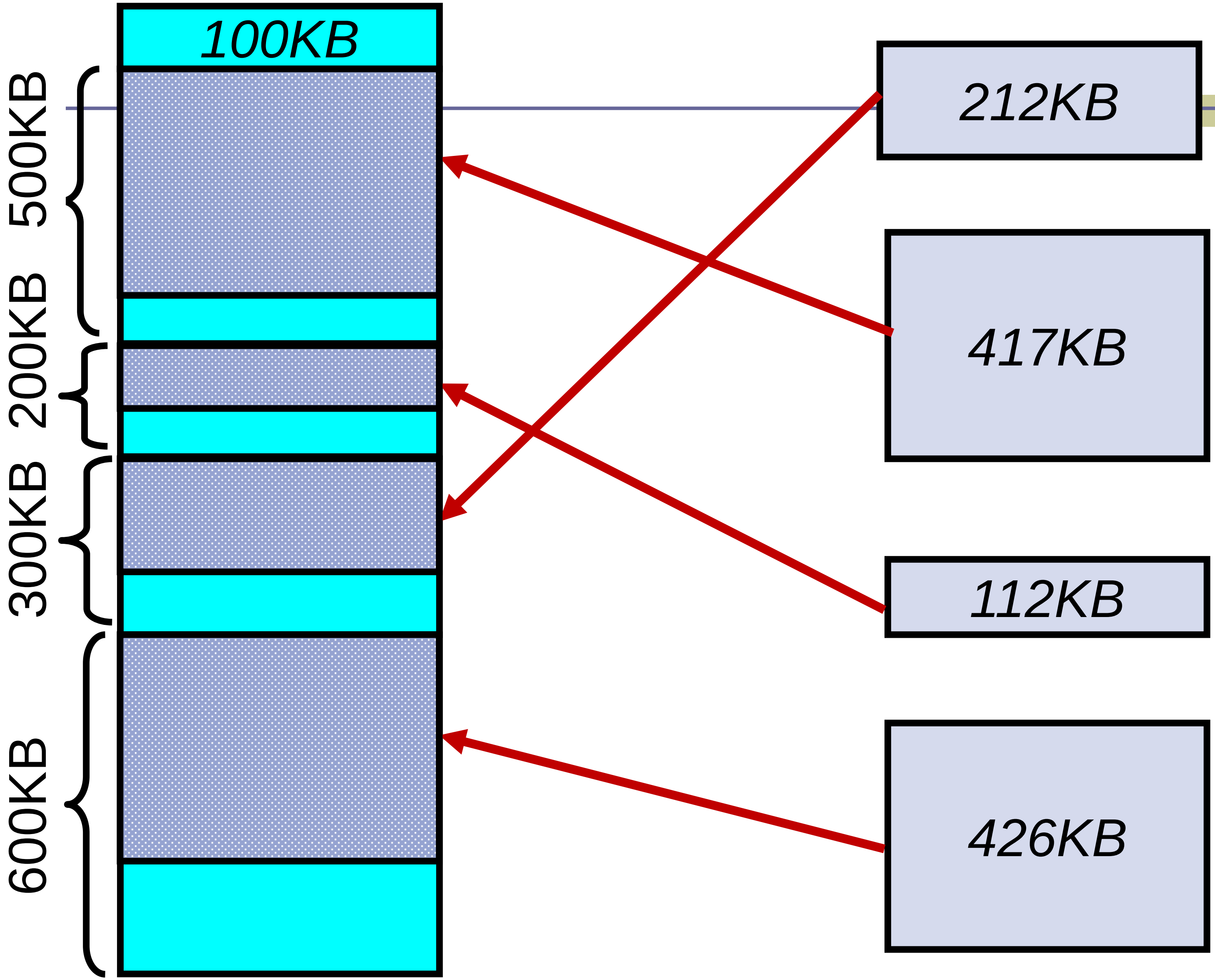
- Given five memory partitions of *100 KB, 500 KB, 200 KB, 300 KB, and 600 KB (in order)*
- How would each of the first-fit, best-fit, and worst-fit algorithms place processes of *212 KB, 417 KB, 112 KB, and 426 KB (in order)*?
- Which algorithm makes the most efficient use of memory?

First-fit

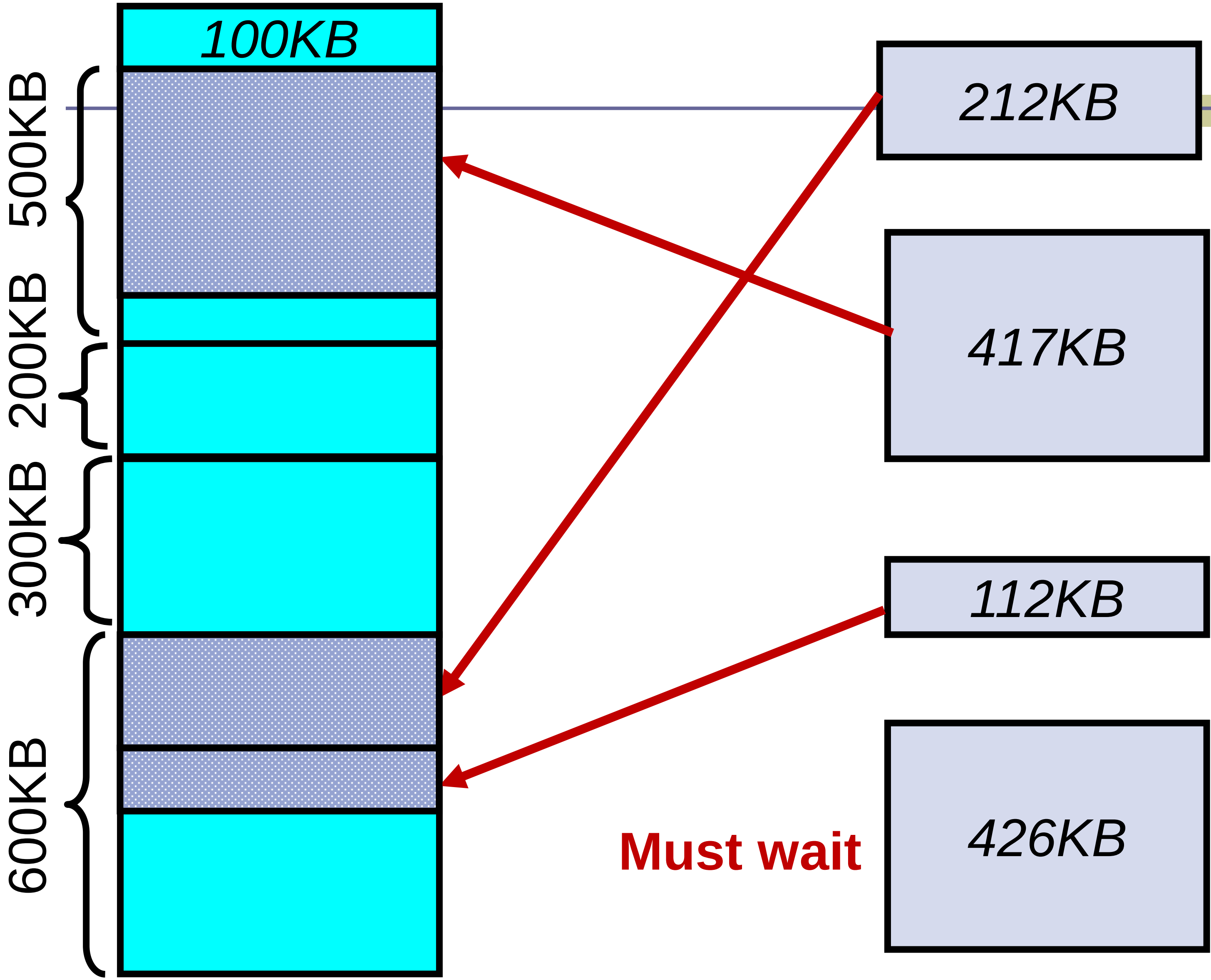


100KB
500KB
200KB
300KB
600KB
212KB
417KB
112KB
426KB
new partition
288K =
500K-212K
Must wait

Best-fit



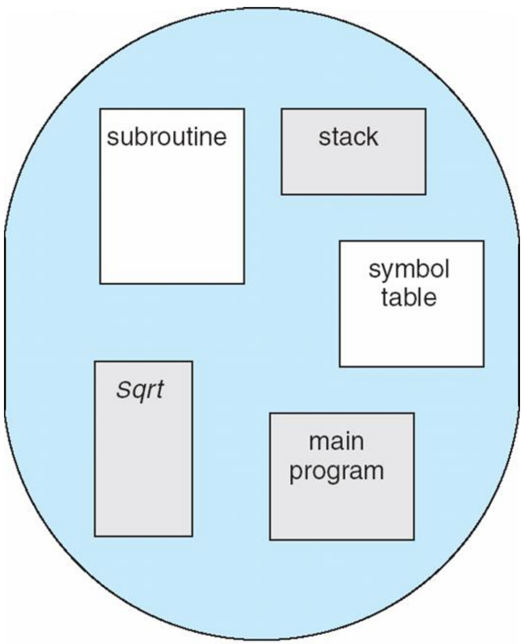
Worst-fit



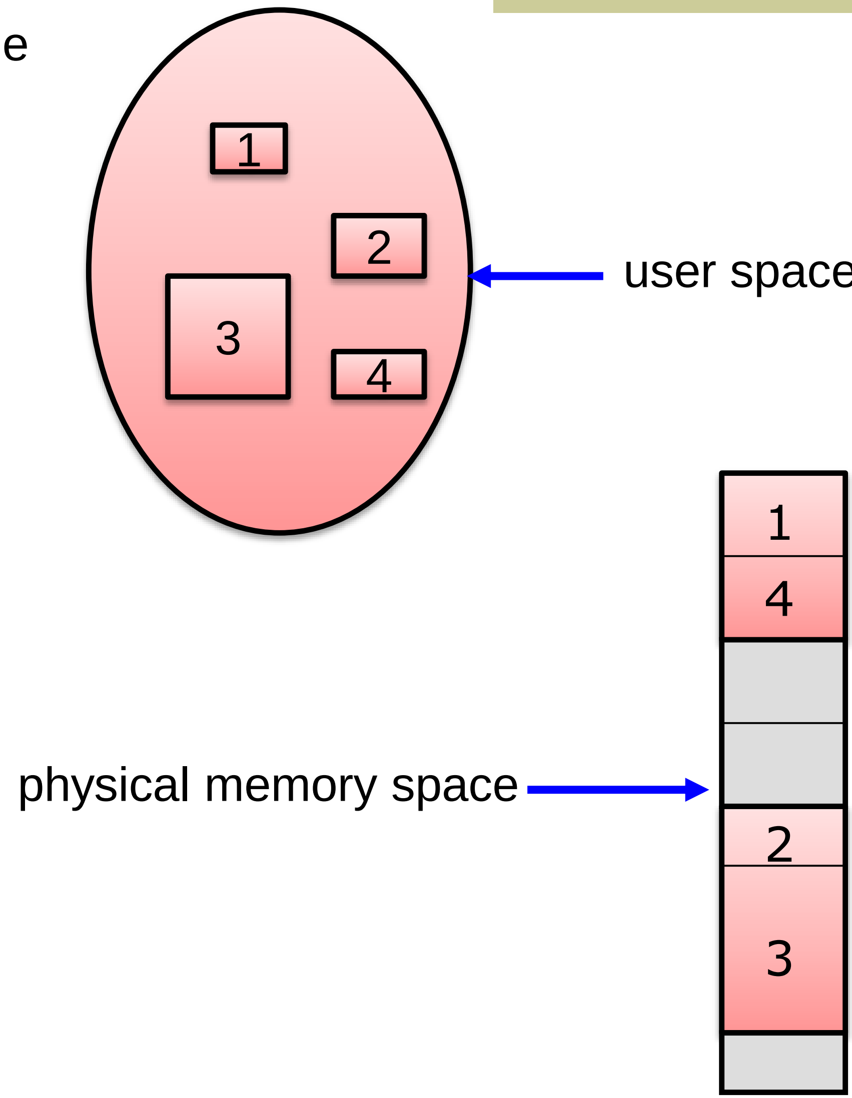
100KB
500KB
200KB
300KB
600KB
212KB
417KB
112KB
426KB
Must wait

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

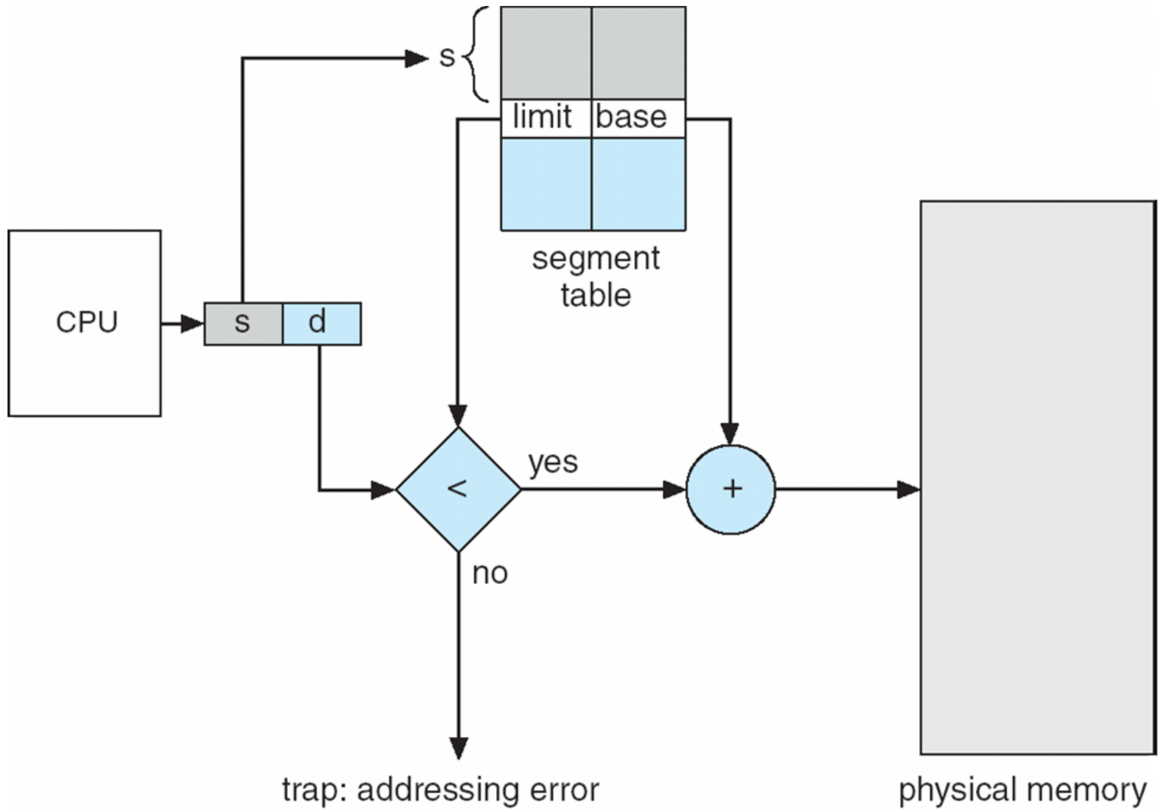
- Logical address consists of a two tuple:

  $\langle \text{segment-number, offset} \rangle$

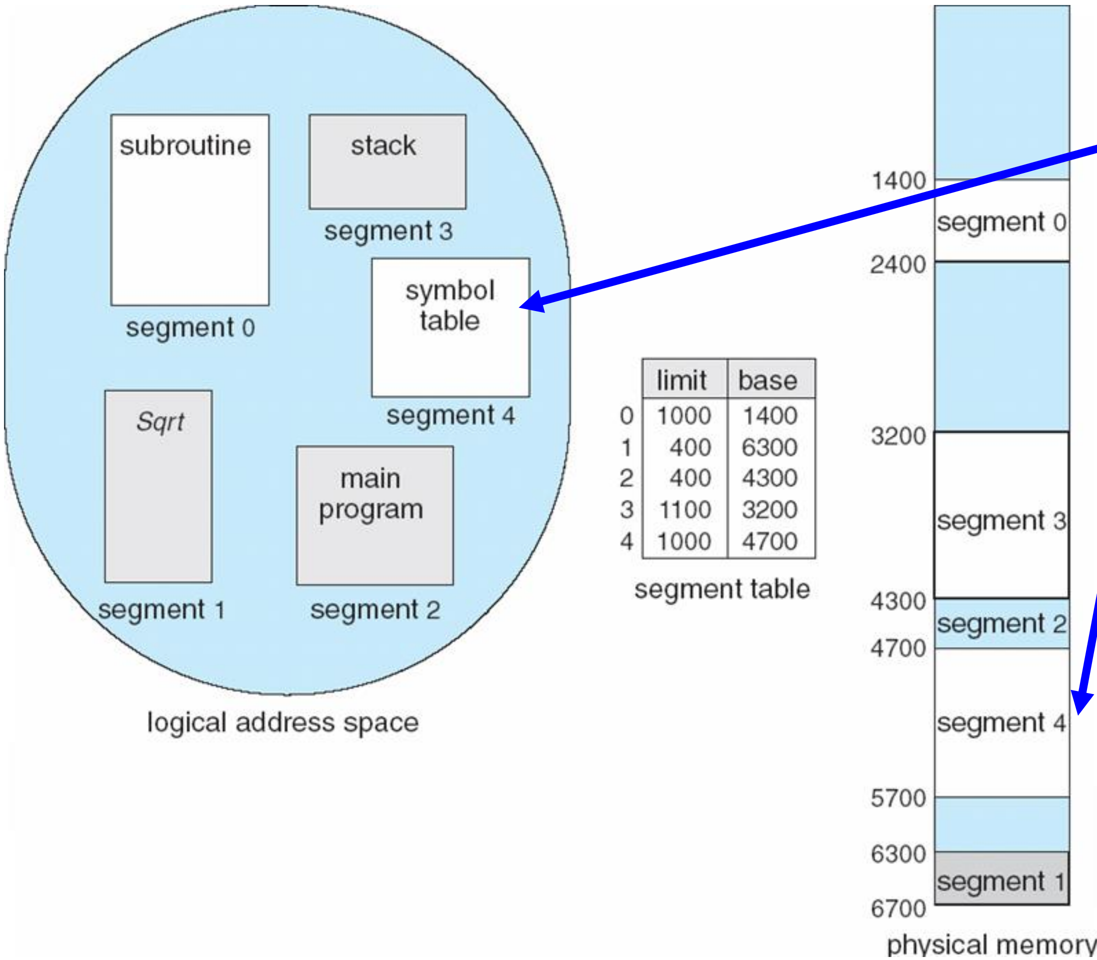
- **Segment table:** maps two-dimensional physical addresses
  - base – contains the starting physical address
  - limit – specifies the length of the segment

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

Address Translation



Example of Segmentation



Logical view: multiple separate segments

Each segment is allocated with a contiguous memory

External fragmentation

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

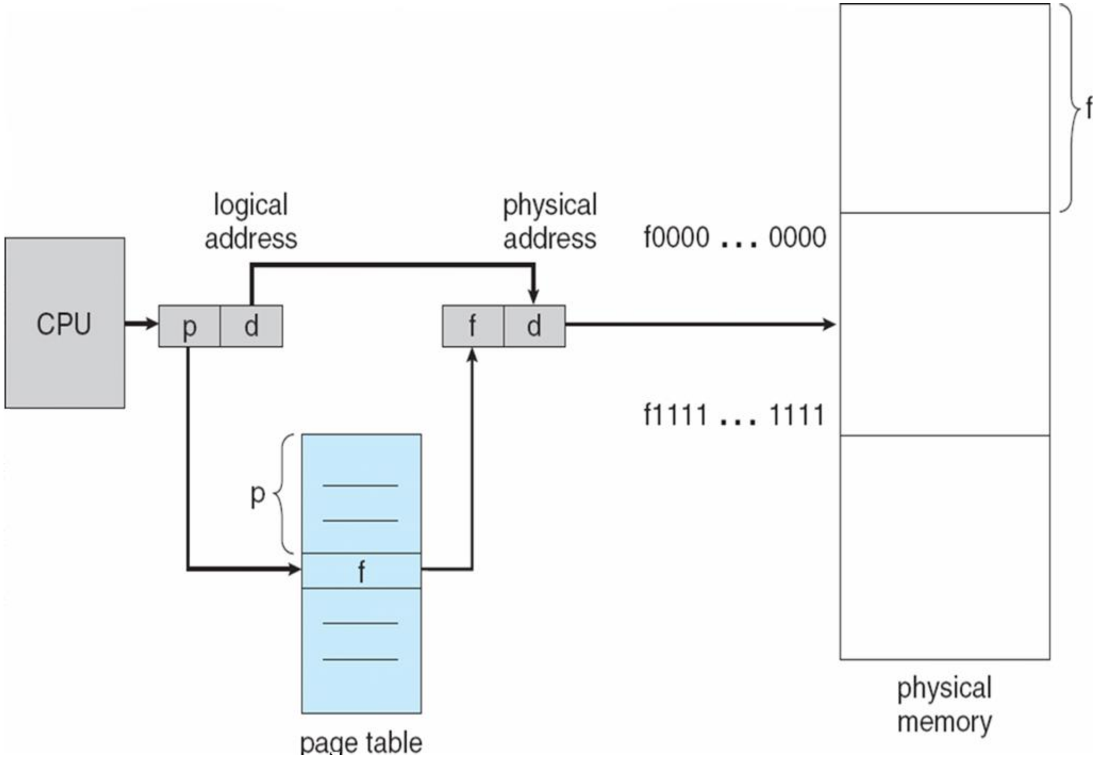
Paging

- Physical address space of a process can be *Non-contiguous*
  - Divide *physical memory* into fixed-sized blocks called **frames,**
  - Divide *logical memory* into blocks of same size called **pages**.
  - Keep track of all free frames

- Set up a page table to translate logical to physical addresses

Address Translation

- 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



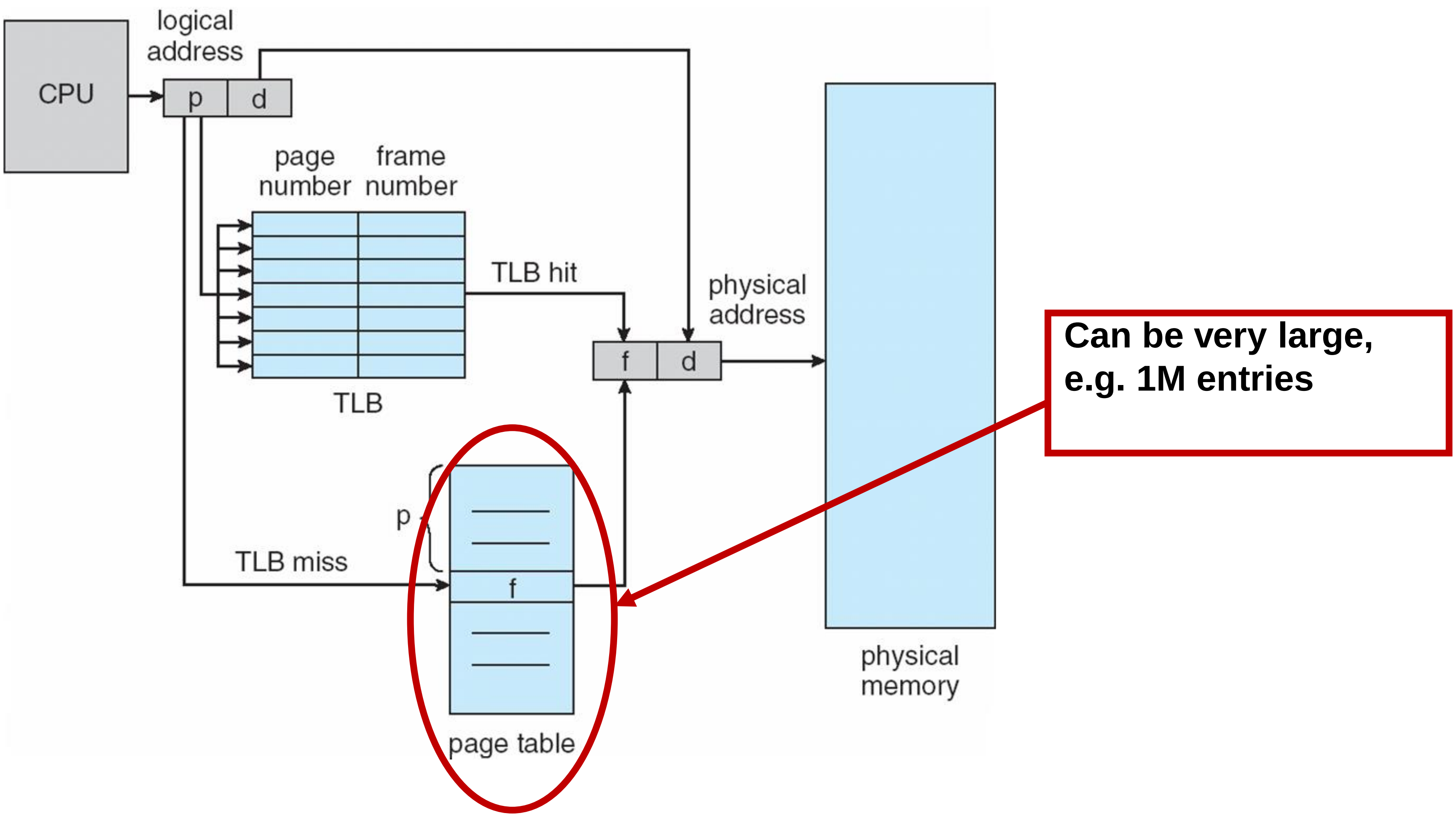
Page Table Implementation

- Implementation of Page Table
  - Page table is kept in main memory
  - *Page-table base register* (PTBR) points to the page table
  - *Page-table length register* (PRLR) indicates size of the page table
  - In this scheme every data/instruction access requires two memory accesses.
    - One for the page table and one for the data/instruction

TLB

- The two memory access problem can be solved by using TLB (translation look-aside buffer)
  - a special, small, fast-lookup hardware cache
  - each entry in the TLB consists of a key (or tag) and a value
  - page number is presented to the TLB, if found, its frame number is immediately available to access memory
  - fast but expensive

Paging Hardware With TLB



TLB miss and Hit ratio

- **TLB miss**:
  - If the page number is not in the TLB, a memory reference to the page table must be made
- **Hit ratio**:
  - percentage of times that a page number is found in the TLB.
- For example:
  - Assume TLB search takes ε = 20ns; memory access takes 100ns

Effective Access Time (EAT)

- TLB hit → 1 memory access = (1 + ε)
- TLB miss → 2 memory accesses = (2 + ε)

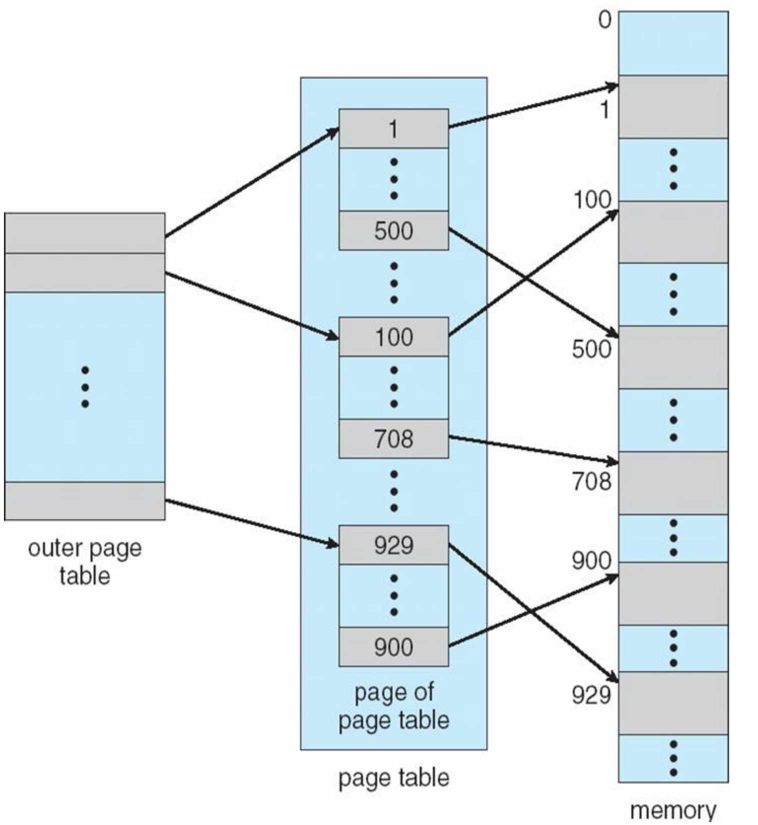
- If Hit ratio = 80%
  - EAT = (20 + 100) * 0.8 + (20 + 200) * 0.2 = 140ns
- If Hit ratio = 98%
  - EAT = (20 + 100) * 0.98 + (20 + 200) * 0.02 = 122ns

Hierarchical Page Tables

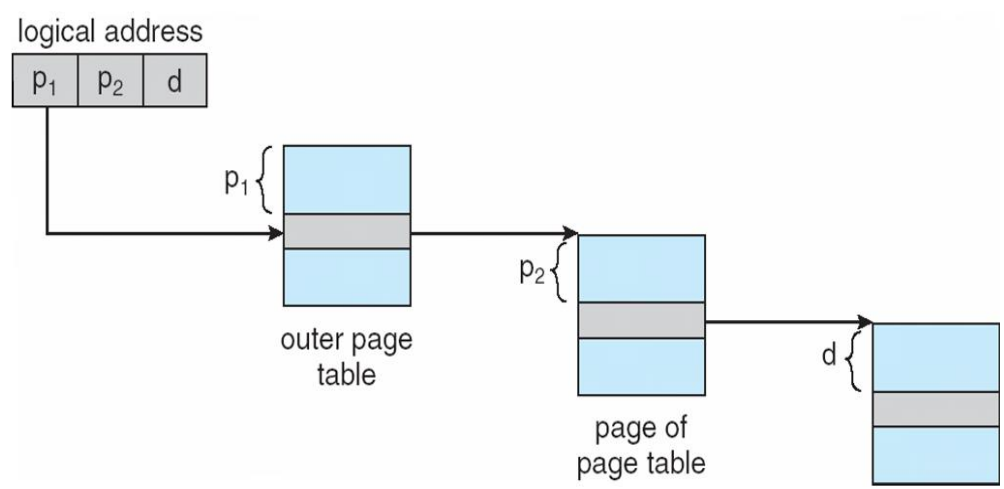
- Modern computer supports a large logical address space
  - computer system: 32 bit address
  - page: 4KB
  - page table: 1 million entries ($2^{32}/2^{12}$)
  - page table entry: 4 bytes
  - page table of each process: 4MB physical address
  - Too big!!

- Solution: To break up the logical address space into multiple page tables
  - A simple technique is a two-level page table

Two-Level Page-Table Scheme

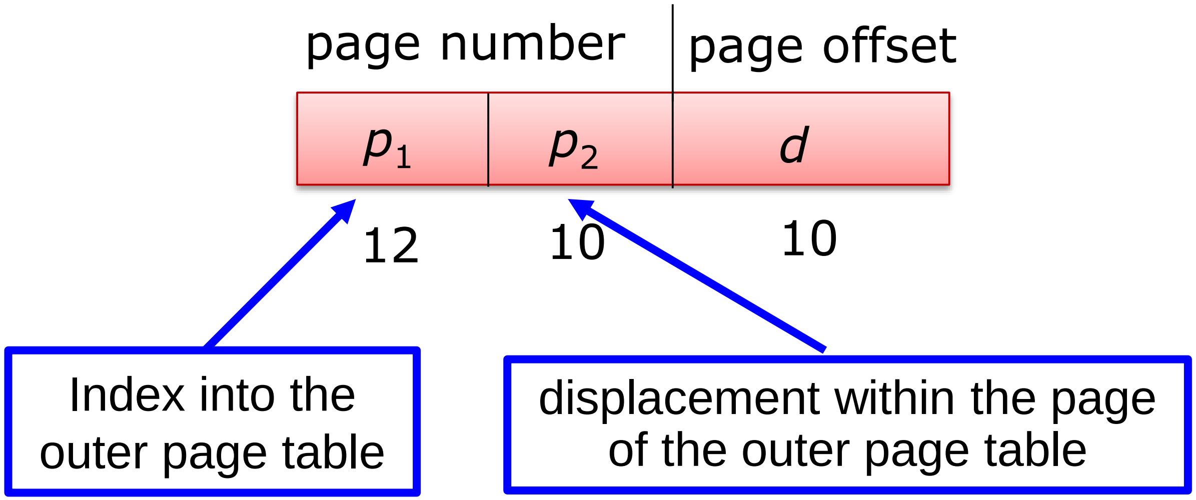


- Each page can be placed in any physical frame inside main memory!
- Address-translation scheme for a two-level 32-bit paging architecture



Two-Level Paging Example

- A logical address (on 32-bit machine with 1K page size) is divided into:
  - a page number consisting of 22 bits
  - a page offset consisting of 10 bits
- Since the page table is paged, the page number is further divided into:
  - a 12-bit page number
  - a 10-bit page offset
- logical address



Paging-Segmentation Combination

- Segmentation and Paging are often combined in order to improve upon each other

- Segmented paging is helpful when the page table becomes very large

  - *e.g., a large contiguous section of the page table that is unused can be collapsed into a single segment table entry with a page table address of zero*

Paging-Segmentation Combination