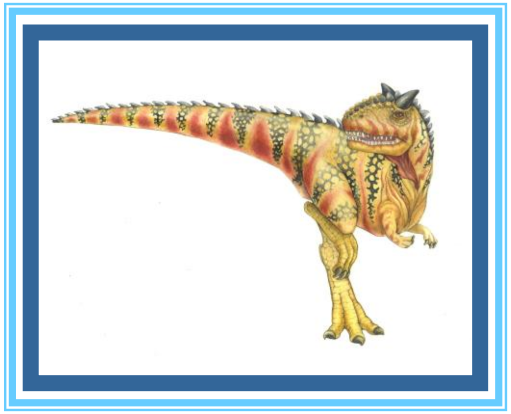
Review

- midterm 2 -

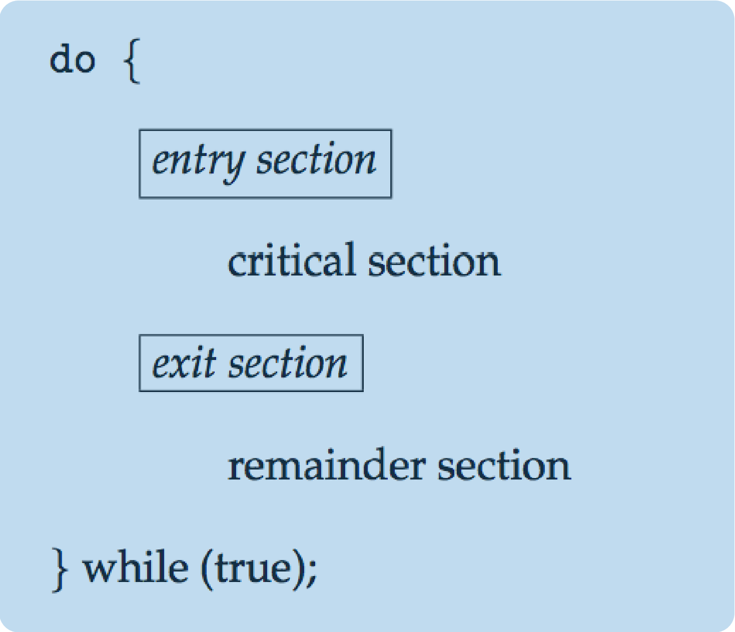


Topics

- Process Synchronization
  - Mutex Locks / Semaphores / Monitors
  - Classical Critical Section Problems
- CPU Scheduling
  - Scheduling Multi- Processes / Threads / Processors
  - Real Time CPU Scheduling
- Deadlocks
  - Deadlock Prevention
  - Deadlock Avoidance
- Main Memory
  - Address Binding
  - Segmentation / Paging

Critical Section

- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design protocol to solve this



```
do {

        entry section

            critical section

        exit section

            remainder section

} while (true);
```

- Two approaches depending on if kernel is preemptive or non-preemptive
  - **Preemptive** – allows preemption of process when running in kernel mode
  - **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU

Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process ***P**$_i$* is executing in its critical section, then no other processes can be executing in their critical sections

2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely

3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted

- Assume that each process executes at a nonzero speed
- No assumption concerning **relative speed** of the ***n*** processes

Solutions

- Peterson’s solution
  - Two process solution
  - Atomic load and store instructions
  - Two variables
    - int turn;
    - Boolean flag[2];

```
do {
        flag[i] = true;
        turn = j;
        while (flag[j] && turn == j);
                critical section
        flag[i] = false;
                remainder section
} while (true);
```

Solutions

- Locking
  - Synchronization hardware with locks
  - Still atomic instructions are needed
  - Approaches:
    - test_and_set()

```
 do {
    while (test_and_set(&lock))
       ; /* do nothing */
    /* critical section */
    lock = false;
    /* remainder section */
 } while (true);
```

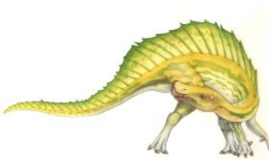
```
boolean test_and_set (boolean *target)
         {
            boolean rv = *target;
            *target = TRUE;
            return rv:
}
```

Solutions

- ❑ Locking
  - ❑ Synchronization hardware with locks
  - ❑ Still atomic instructions are needed
  - ❑ Approaches
    - ❑ test_and_set()
    - ❑ compare_and_swap()

```
do {
   while (compare_and_swap(&lock,0,1)
     !=0)
      ; /* do nothing */
   /* critical section */
   lock = 0;
   /* remainder section */
} while (true);
```

```
int compare and swap(int *value, int
  expected, int new value) {
   int temp = *value;
   if (*value == expected)
      *value = new value;
   return temp;
}
```



Solutions

- ❑ Locking
    - ❑ Synchronization hardware with locks
    - ❑ Still atomic instructions are needed
    - ❑ Approaches
        - ❑ test_and_set()
        - ❑ compare_and_swap()
        - ❑ Bounded-waiting tese_and_set()

```
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&lock);
   waiting[i] = false;
   /* critical section */
   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;
   if (j == i)
      lock = false;
   else
      waiting[j] = false;
   /* remainder section */
} while (true);
```

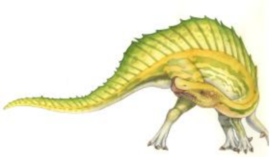
Mutex Locks

- Simpler but power tool
  - Still atomic instructions are needed
    - acquire() and release()
  - Problem - Busy waiting
    - spinlock

```
do {
    acquire lock
         critical section
    release lock
         remainder section
} while (true);
```

```
acquire() {
   while (!available)
      ; /* busy wait */
   available = false;;
}
```

```
release() {
   available = true;
}
```



Semaphores

- Synchronization tool that does not require busy waiting
  - Still atomic instructions are needed
    - wait () and signal ()
  - Semaphore S – integer
    - Counting semaphore
    - Binary semaphore
      - Same as mutex

```
do {
   wait(S)
        critical section
   signal(S)
        remainder section
} while (true);
```

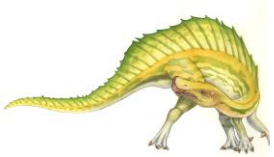
```
wait (S) {
    while (S <= 0)
       ; // busy wait
    S--;
}
```

```
signal (S) {
    S++;
}
```

Semaphore Implementation

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list

- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue

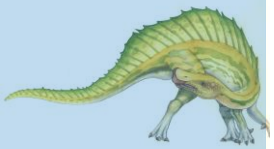


Semaphores without Busy Waiting

```
typedef struct{
   int value;
   struct process *list;
} semaphore;
```

```
wait(semaphore *S) {
   S->value--;
   if (S->value < 0) {
      add this process to S->list;
      block();
   }
}
```

```
signal(semaphore *S) {
   S->value++;
   if (S->value <= 0) {
      remove a process P from S->list;
      wakeup(P);
   }
}
```



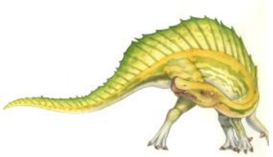
Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let ***S*** and ***Q*** be two semaphores initialized to 1

```
P0
wait (S);
wait (Q);
...
signal (S);
signal (Q);
```

```
P1
wait (Q);
wait (S);
...
signal (Q);
signal (S);
```

- **Starvation** – **indefinite blocking**
  - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via **priority-inheritance protocol**



Synchronization Problem 1:
Producer and Consumer

- ❑ The structure of the producer process

```
do {
      ...
   /* produce an item in next_produced */
      ...
   wait(empty);
               wait(mutex);
      ...
   /* add next produced to the buffer */
      ...
                    signal(mutex);
   signal(full);
} while (true);
```

- ❑ The structure of the consumer process

```
do {
   wait(full);
                    wait(mutex);
      ...
   /* remove an item from buffer to next_consumed */
      ...
                    signal(mutex);
   signal(empty);
      ...
   /* consume the item in next consumed */
      ...
} while (true);
```

- ❑ Shared data
  - ❑ Semaphore **`mutex`** initialized to the value 1
  - ❑ Semaphore **`full`** initialized to the value 0
  - ❑ Semaphore **`empty`** initialized to the value n

Synchronization Problem 2:
Writer and Readers

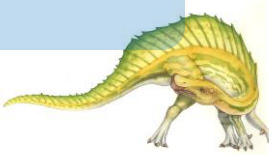
- The structure of a writer process

```
do {
   wait(rw mutex);
      ...
   /* writing is performed */
      ...
   signal(rw mutex);
} while (true);
```

- Shared Data
  - Semaphore **rw_mutex** initialized to 1
  - Semaphore **mutex** initialized to 1
  - Integer **read_count** initialized to 0

- The structure of a reader process

```
do {
          wait(mutex);
          read count++;
          if (read count == 1)
                    wait(rw mutex);
          signal(mutex);
                    ...
          /* reading is performed */
                    ...
          wait(mutex);
          read count--;
          if (read count == 0)
                    signal(rw mutex);
          signal(mutex);
} while (true);
```



Synchronization Problem 3: Dining-Philosophers

- Solution Based on Monitors

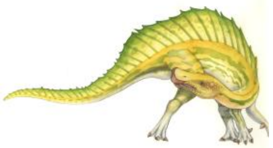
```
monitor DiningPhilosophers {
    enum { THINKING, HUNGRY, EATING } state [5] ;
    condition self [5];

    void pickup (int i) {
            state[i] = HUNGRY;
            test(i);
            if (state[i] != EATING) self [i].wait;
    }

    void putdown (int i) {
            state[i] = THINKING;
            // test left and right neighbors
            test((i + 4) % 5);
            test((i + 1) % 5);
    }
```

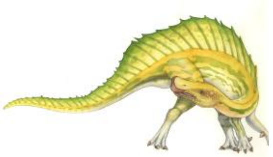
```
    void test (int i) {
            if ( (state[(i + 4) % 5] != EATING) &&
                (state[i] == HUNGRY) &&
                (state[(i + 1) % 5] != EATING) ) {
                        state[i] = EATING ;
                        self[i].signal () ;
            }
    }

    initialization_code() {
            for (int i = 0; i < 5; i++)
                state[i] = THINKING;
            }
}
```



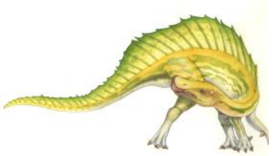
CPU Scheduler

- **Short-term scheduler** selects from among the processes in ready queue, and allocates the CPU to one of them
  - Queue may be ordered in various ways
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
- Scheduling under 1 and 4 is **nonpreemptive**
- All other scheduling is **preemptive**
  - Consider access to shared data
  - Consider preemption while in kernel mode
  - Consider interrupts occurring during crucial OS activities



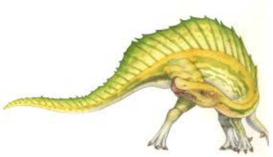
Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running



Scheduling Criteria

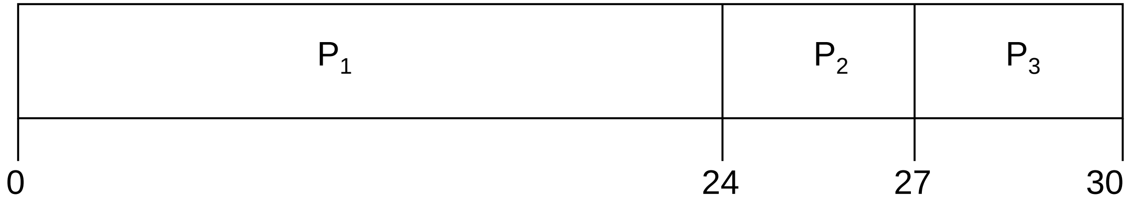
- **CPU utilization** – keep the CPU as busy as possible
  - Optimize algorithms to maximize CPU utilization
- **Throughput** – # of processes that complete their execution per time unit
  - Optimize algorithms to maximize throughput
- **Turnaround time –** amount of time to execute a particular process
  - Optimize algorithms to minimize turnaround time
- **Waiting time –** amount of time a process has been waiting in the ready queue
  - Optimize algorithms to minimize waiting time
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)
  - Optimize algorithms to minimize response time



First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
  The Gantt Chart for the schedule is:

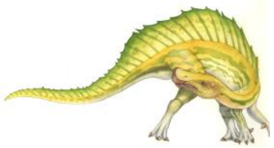


P1 P2 P3

0 24 27 30

  - Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
  - Average waiting time: (0 + 24 + 27)/3 = 17

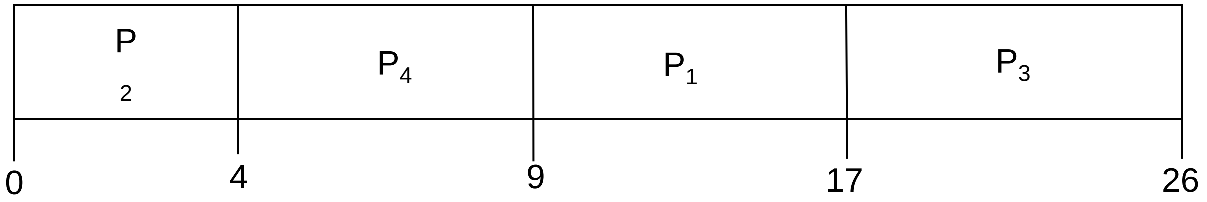
- **Convoy effect** - short process behind long process
  - Consider one CPU-bound and many I/O-bound processes



Shortest Job First (SJF) Scheduling

Process	Burst Time
P_1	8
P_2	4
P_3	9
P_4	5

- SJF scheduling chart



  - Average waiting time = (9 + 0 + 17 + 4) / 4 = 7.5

- Used frequently in long-term scheduling
  - For short-term scheduling, exponential averaging is used (see the lecture)

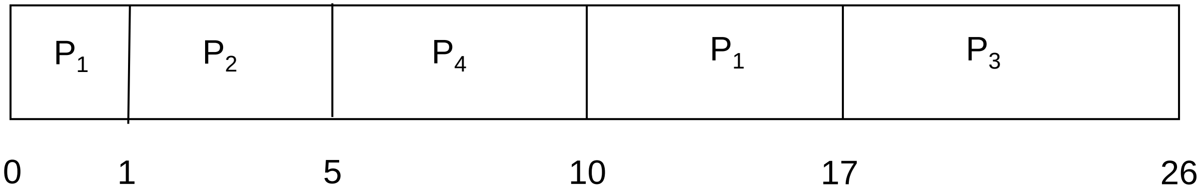
Shortest-Remaining-Time-First (SRTF) Scheduling

- Preemptive version

- Now we add the concepts of varying **arrival times** and preemption to the analysis

Process	*Arrival* Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- *Preemptive* SJF Gantt Chart

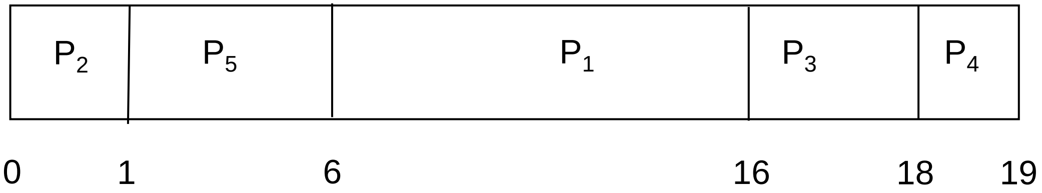


  - Average waiting time = [(10-1)+(1-1)+(17-2)+5-3)]/4 = 26/4 = 6.5 msec
  - Compare with the average waiting time by nonpreemtpvie SJF

Priority Scheduling

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

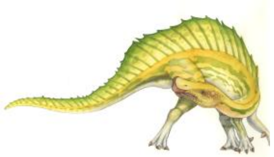
- Priority scheduling Gantt Chart



  - Average waiting time = 8.2 msec

- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time

- **Starvation** – low priority processes may never execute
  - Solution: **Aging** – as time progresses increase the priority of the process



Round Robin (RR) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

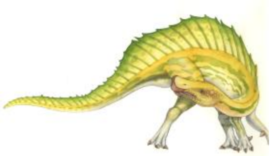
- When Time Quantum is 4, the Gantt chart is:

P_1	P_2	P_3	P_1	P_1	P_1	P_1	P_1

0 4 7 10 14 18 22 26 30

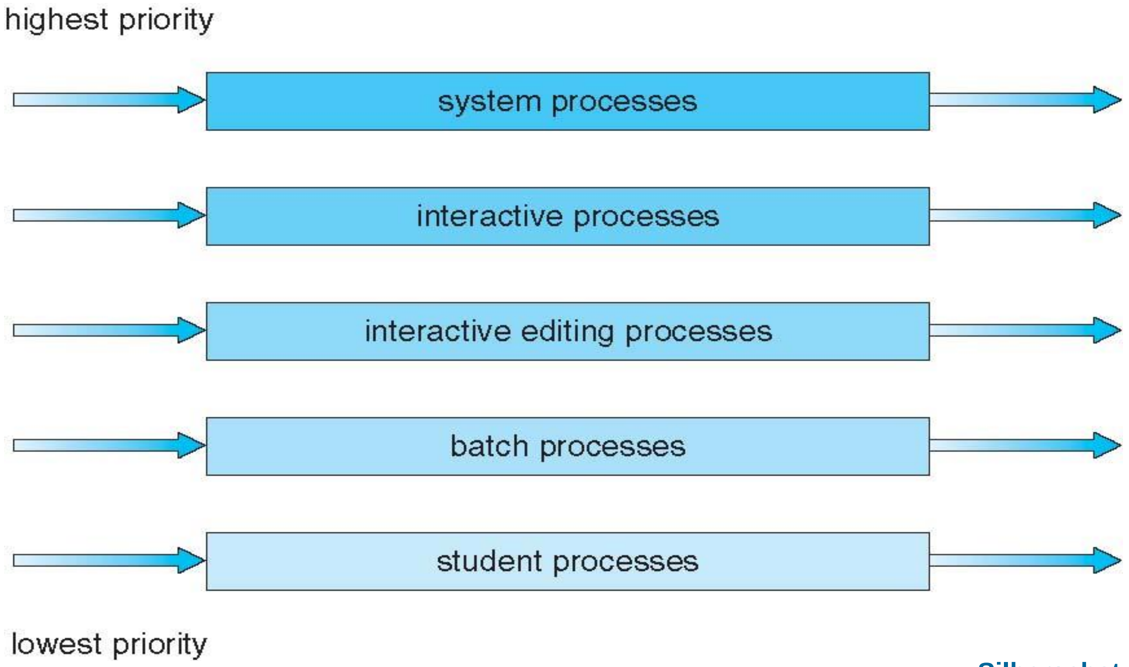
- Average waiting time?
  - Does RR have better performance than the previous algorithms?
  - Typically, higher average turnaround than SJF, but better ***response***

- Size of quantum q
  - If *q* is large enough, it would work like FIFO.
  - q should be large compared to context switch time
  - *q* must be large with respect to context switch, otherwise overhead is too high



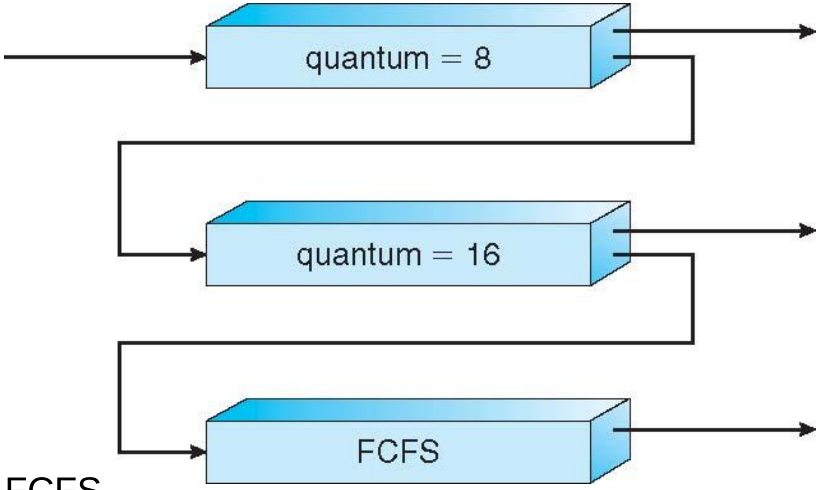
Multilevel Queue Scheduling

- Ready queue is partitioned into separate queues, each of which has its own scheduling algorithm
  - **foreground** (interactive) - RR
  - **background** (batch) - FCFS
- Scheduling must be done between the queues:
  - Fixed priority scheduling → Possibility of starvation.
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR, 20% to background in FCFS



Multilevel Feedback Queue Scheduling

- A process can move between the various queues; aging can be implemented this way



- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS

- Scheduling
  - A new job enters queue Q_0 which is served FCFS
    - When it gains CPU, job receives 8 milliseconds
    - If it does not finish in 8 milliseconds, job is moved to queue Q_1

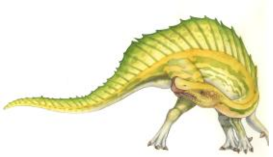
  - At Q_1 job is again served FCFS and receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue Q_2

Thread Scheduling

- Distinction between user-level and kernel-level threads
- When threads supported, threads scheduled, not processes
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
  - Known as **process-contention scope** (**PCS**) since scheduling competition is within the process
  - Typically done via *priority* set by programmer
  - Not needed for one-to-one does model
- Kernel thread scheduled onto available CPU is **system-contention scope** (**SCS**) – competition among all threads in system

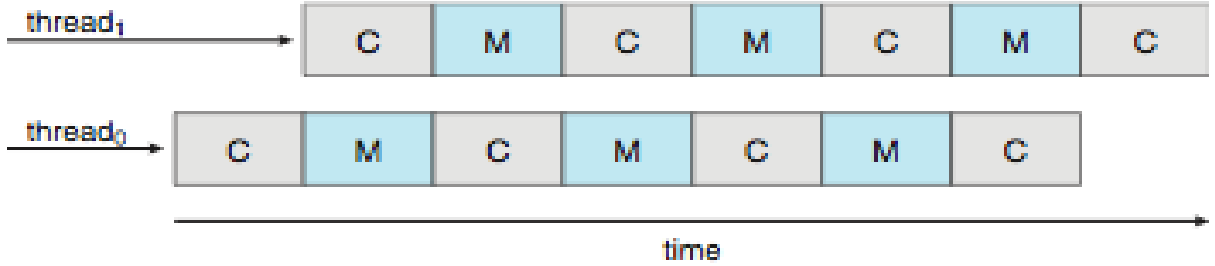
Multiple-Processor Scheduling

- **Homogeneous processors** within a multiprocessor
  - **Asymmetric multiprocessing** – only one processor accesses the system data structures, alleviating the need for data sharing
  - **Symmetric multiprocessing (SMP)** – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes
- **Processor affinity** – process has affinity for processor on which it is currently running
  - **soft affinity**
  - **hard affinity**
  - Variations including **processor sets**
- **Load balancing** attempts to keep workload evenly distributed
  - **Push migration** – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs
  - **Pull migration** – idle processors pulls waiting task from busy processor



Multicore Processors Scheduling

- Recent trend to place multiple processor cores on same physical chip
  - Faster and consumes less power
  - Multiple threads per core also growing
- Takes advantage of memory stall to make progress on another thread while memory retrieve happens

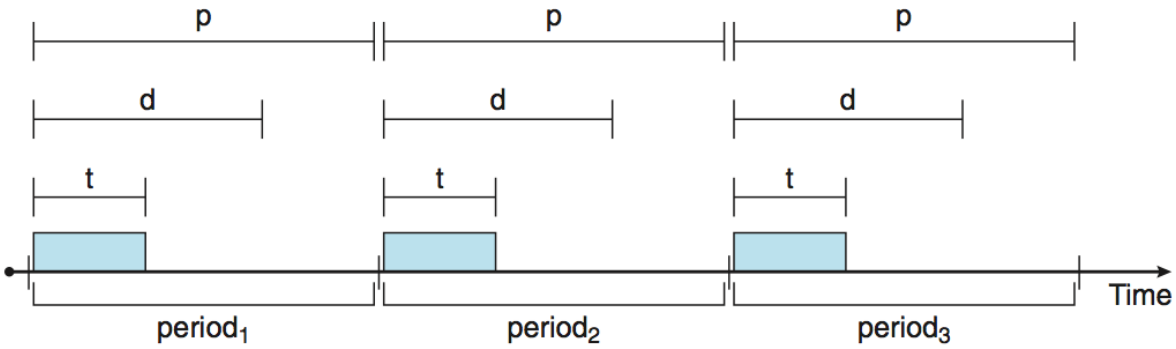


- Coarse-grained / Fine-grained multithreading
  - Coarse-grained: a thread executes on a processor until a long-latency event such as a memory stall occurs
  - Fine-grained: interleaved multithreading switches; much finer level of granularity – by instruction cycle

Priority-based Real-Time Scheduling

- For soft real-time scheduling, scheduler must support preemptive, priority-based scheduling
  - **Soft real-time systems** – no guarantee as to when critical real-time process will be scheduled
  - But only guarantees soft real-time

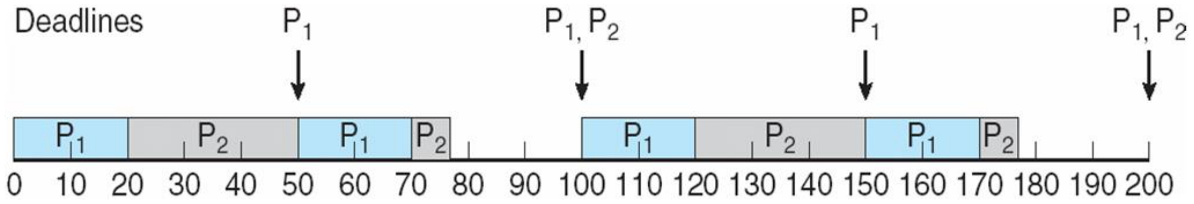
- For hard real-time must also provide ability to meet deadlines
  - **Hard real-time systems** – task must be serviced by its deadline
  - Processes have new characteristics: **periodic** ones require CPU at constant intervals
  - Has processing time *t*, deadline *d,* period *p;* $0 \le t \le d \le p$
  - **Rate** of periodic task



Rate Montonic Scheduling

- A priority is assigned based on the inverse of its period
  - Shorter **periods** = higher priority
  - Longer **periods** = lower priority

- P_1 is assigned a higher priority than P_2.
  - P_1 = 50; P_2 = 100
  - t_1 = 20; t_2 = 35

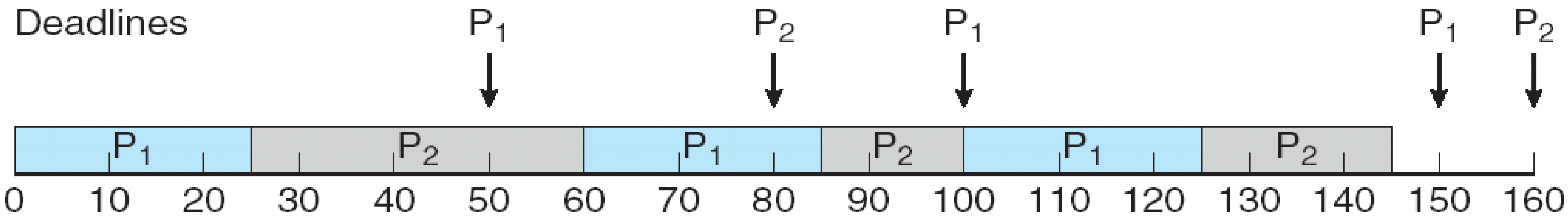


- CPU Utilization
  - The worst-case CPU Utilization = $N\left(2^{1/N} - 1\right)$
    - When N = 2 (the number of processes), 83%
  - In the example, $P_1 + P_2$ = 0.75 (=75% << 83%)
    - What if P_1 = 50, P_2 = 80, t_1 = 25, t_2= 35?

Earliest Deadline First Scheduling (EDF) Scheduling

- Priorities are assigned according to deadlines:
  - the earlier the **deadline**, the higher the priority;
  - the later the **deadline**, the lower the priority

- In the previous scenario,
  - $P_1 = 50$; $P_2 = 80$
  - $t_1 = 25$; $t_2 = 35$



Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion:** only one process at a time can use a resource

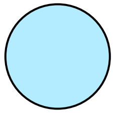
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes

- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task

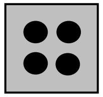
- **Circular wait:** there exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

Resource-Allocation Graph Instances

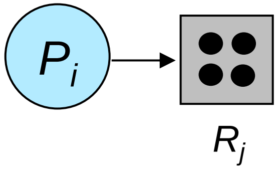
- Process



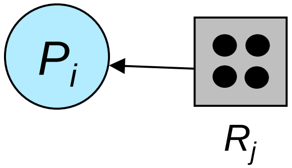
- Resource Type with 4 instances

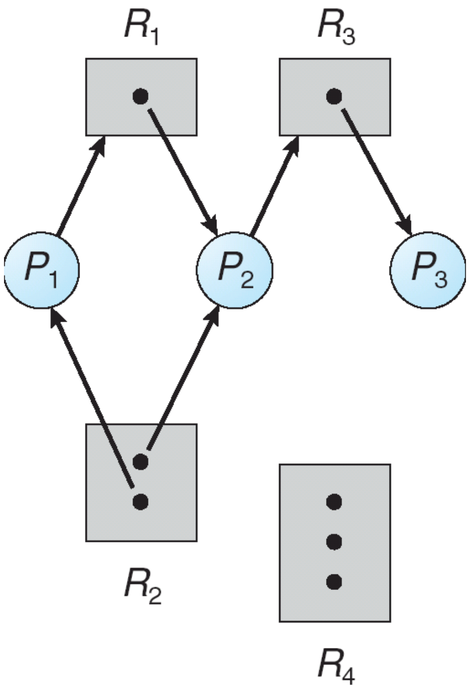


- P_i requests instance of R_j (Request edge)



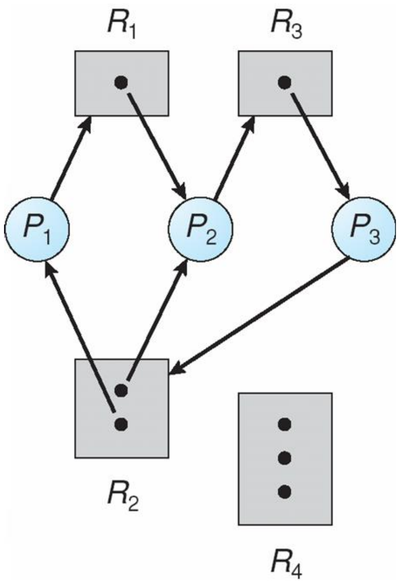
- P_i is holding an instance of R_j (assignment edge)

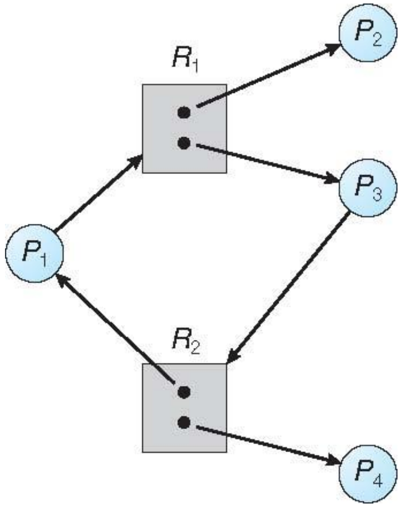




Cycles and Deadlocks

- If graph contains no cycles, then no deadlock
- If graph contains a cycle, then …
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, possibility of deadlock

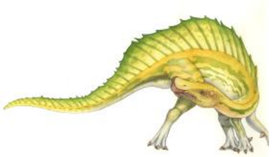




Deadlock Prevention

Restrain the ways request can be made

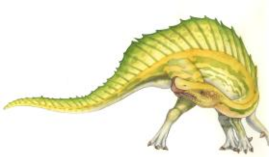
- **Mutual Exclusion** – not required for sharable resources; must hold for nonsharable resources
- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
- **No Preemption** – If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
- **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

Requires that the system has some additional ***a priori*** information available

- Simplest and most useful model requires that each process declare the ***maximum number*** of resources of each type that it may need
- 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

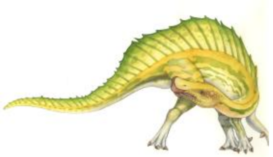


Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state

- System is in **safe state** if there exists a sequence <P_1, P_2, …, P_n> of ALL the processes in the systems such that 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 $j < i$

- Deadlock in safe or unsafe state
  - If a system is in safe state → no deadlocks
  - If a system is in unsafe state → possibility of deadlock
  - Avoidance → ensure that a system will never enter an unsafe state.

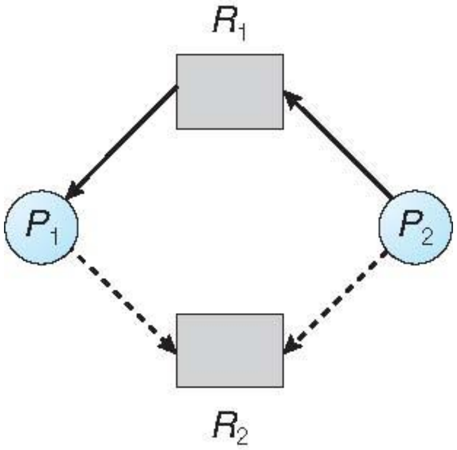


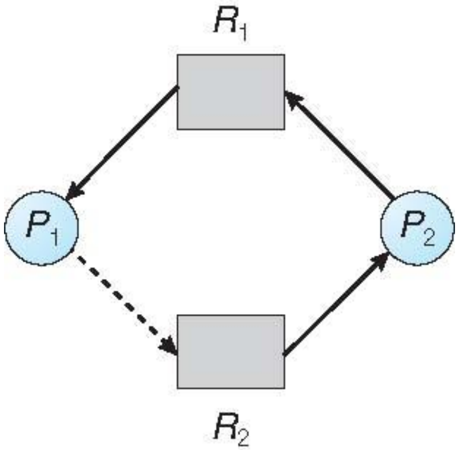
Resource-Allocation Graph Algorithm

- Avoidance algorithm for single instance scenarios

- **Claim edge** (P_i --> R_j) indicated that process P_j *may* request resource R_j
  - Claim edge converts to request edge when a process requests a resource
  - Request edge converted to an assignment edge when the resource is allocated to the process
  - When a resource is released by a process, assignment edge reconverts to a claim edge

- The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph





Banker’s Algorithm

- Avoidance algorithm for multiple instances
- Each process must a *priori* claim maximum use
  - Safety Algorithm
- When a process requests a resource it may have to wait
  - Resource-request Algorithm
- When a process gets all its resources it must return them in a finite amount of time
- Data structures
  - **Available**: If $available[j] = k$, there are *k* instances of resource type R_j available
  - **Max**: If $Max\ [i,j] = k$, then process P_i may request at most *k* instances of resource type R_j
  - **Allocation**: If $Allocation[i,j] = k$ then P_i is currently allocated *k* instances of R_j
  - **Need**: 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:
   $\boldsymbol{Work} = \boldsymbol{Available}$
   $\boldsymbol{Finish}\,[\boldsymbol{i}] = \textbf{false for } \boldsymbol{i} = \mathbf{0, 1, \dots,} \boldsymbol{n} - \mathbf{1}$

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

   (a) $\boldsymbol{Finish}\,[\boldsymbol{i}] = \textbf{false}$

   (b) $\boldsymbol{Need_i} \leq \boldsymbol{Work}$

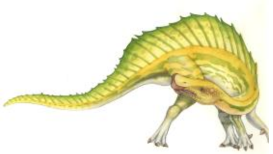
   If no such i exists, go to step 4

3. Update
   $\boldsymbol{Work} = \boldsymbol{Work} + \boldsymbol{Allocation_i}$
   $\boldsymbol{Finish}[\boldsymbol{i}] = \textbf{true}$

   And then go to step 2

4. If $\boldsymbol{Finish}\,[\boldsymbol{i}] == \textbf{true}$ 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.

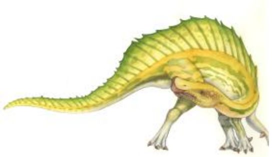
If $Request_i[j] = k$ then process P_i wants k instances of resource type R_j

1. If $Request_i \leq Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If $Request_i \leq Available$, 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:

   $Available = Available – Request;$

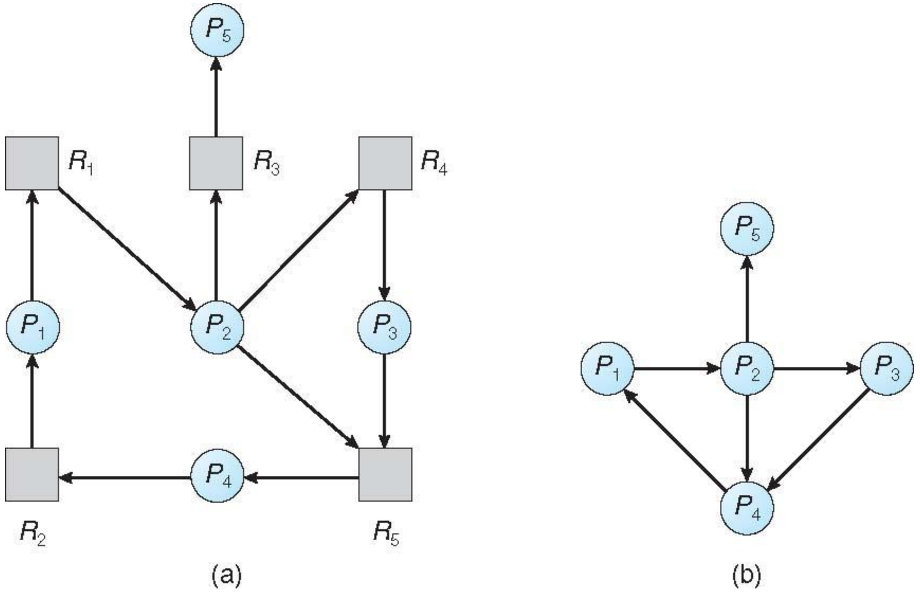
   $Allocation_i = Allocation_i + Request_i;$
   $Need_i = Need_i – Request_i;$

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



Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
  - Single instance: use wait-for graph
    - Periodically invoke an algorithm that searches for a cycle in the graph.



Resource-Allocation Graph Corresponding wait-for graph

  - Multiple instances: modified from the Banker's algorithm

Detection Algorithm

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

   (a) $\boldsymbol{Work} = \boldsymbol{Available}$

   (b) For $\boldsymbol{i} = \mathbf{1}, \mathbf{2}, \ldots, \boldsymbol{n}$,

   If $\boldsymbol{Allocationi} \neq \mathbf{0}$, then $\boldsymbol{Finish}[\boldsymbol{i}] = \mathbf{false}$;
   Otherwise, $\boldsymbol{Finish}[\boldsymbol{i}] = \mathbf{true}$

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

   (a) $\boldsymbol{Finish}[\boldsymbol{i}] == \mathbf{false}$

   (b) $\boldsymbol{Request}_{\boldsymbol{i}} \leq \boldsymbol{Work}$

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

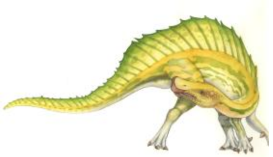
3. $\boldsymbol{Work} = \boldsymbol{Work} + \boldsymbol{Allocation}_{\boldsymbol{i}}$

   $\boldsymbol{Finish}[\boldsymbol{i}] = \mathbf{true}$

   Go to step 2

4. If $\boldsymbol{Finish}[\boldsymbol{i}] == \mathbf{false}$, for some i, $1 \leq \boldsymbol{i} \leq \boldsymbol{n}$;

   Then the system is in deadlock state, and ***P**$_i$* is deadlocked

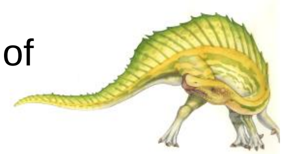


Recovery from Deadlock

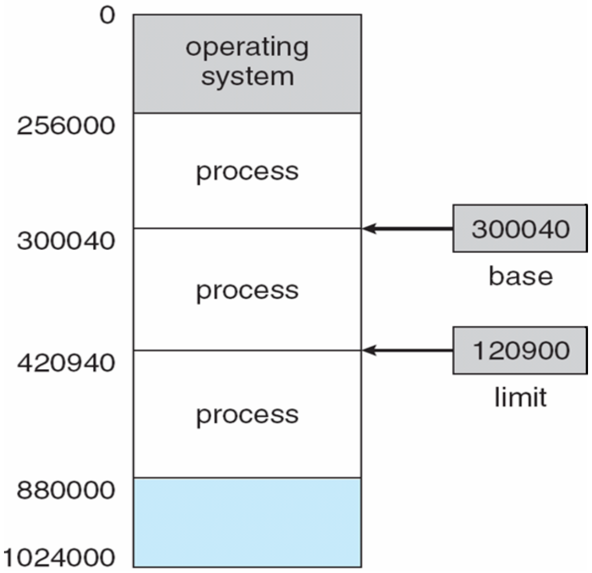
- Process Termination
  - Abort all deadlocked processes
  - Abort one process at a time until the deadlock cycle is eliminated

- In which order should we choose to abort?
  1. Priority of the process
  2. How long process has computed, and how much longer to completion
  3. Resources the process has used
  4. Resources process needs to complete
  5. How many processes will need to be terminated
  6. Is process interactive or batch?

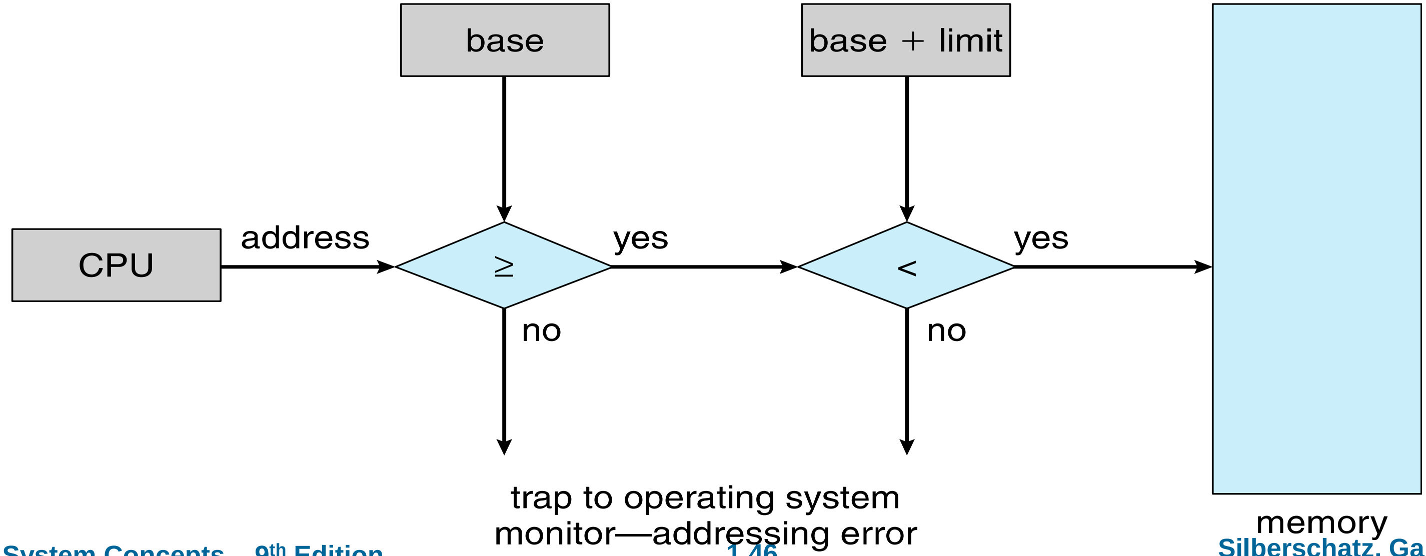
- More issues:
  - **Selecting a victim** – minimize cost
  - **Rollback** – return to some safe state, restart process for that state
  - **Starvation** – same process may always be picked as victim, include number of rollback in cost factor



Base and Limit Registers



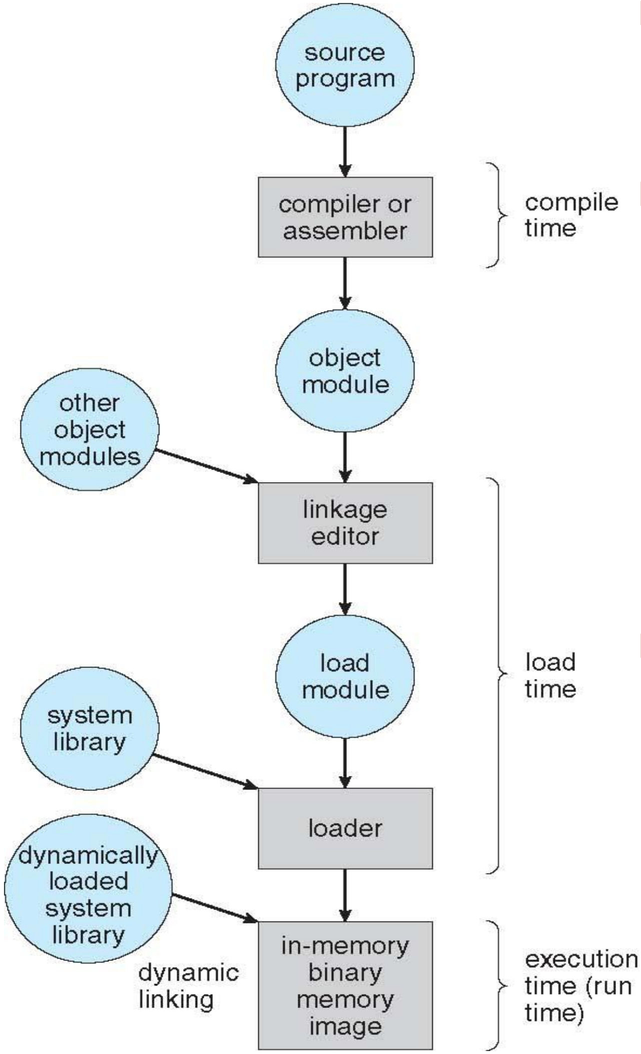
- A pair of **base** and **limit registers** define the logical address space
- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user
  - Used for hardware address protection



Address Binding

- Programs on disk, ready to be brought into memory to execute form an **input queue**
  - Without support, must be loaded into address 0000
- Further, addresses represented in different ways at different stages of a program’s life
  - Source code addresses usually symbolic
  - Compiled code addresses **bind** to relocatable addresses
    - i.e. “14 bytes from beginning of this module”
  - Linker or loader will bind *relocatable* addresses to absolute addresses
    - i.e. 74014
  - Each binding maps one address space to another

Address Binding



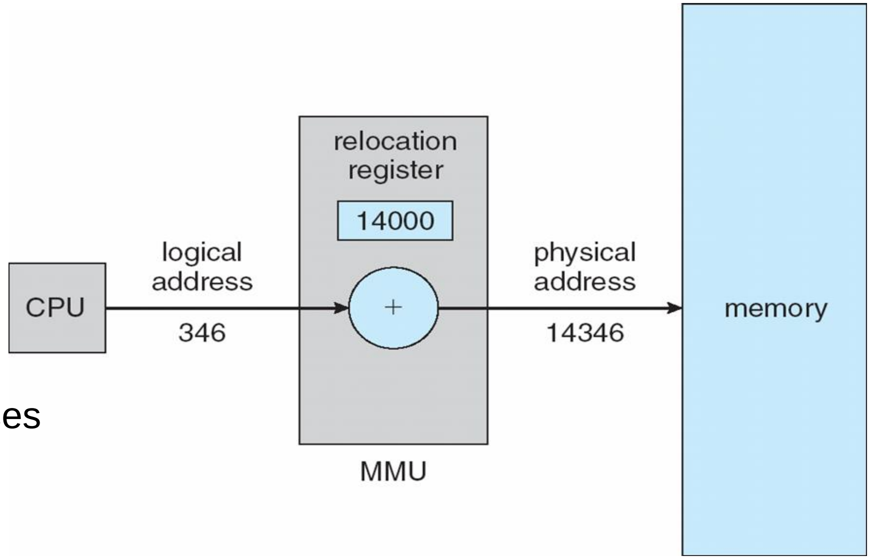
- Programs on disk, ready to be brought into memory to execute form an **input queue**
  - Without support, must be loaded into address 0000
- Further, addresses represented in different ways at different stages of a program's life
  - Source code addresses usually symbolic
  - Compiled code addresses **bind** to relocatable addresses (e.g., 14 bytes from beginning of this module)
  - Linker or loader will bind *relocatable* addresses to absolute addresses (e.g., 74014)
  - Each binding maps one address space to another
- Address binding of instructions and data to memory addresses can happen at three different stages
  - **Compile time**: **absolute code**; must recompile code if starting location changes
  - **Load time**: **relocatable code**
  - **Execution time**: delay binding until run time; the process can be moved during its execution from one memory segment to another
    - Need hardware support for address maps (e.g., base and limit registers)

Logical vs. Physical Address Space

- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
  - **Logical address** – generated by the CPU; also referred to as **virtual address**
  - **Physical address** – address seen by the memory unit
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme
  - **Logical address space** is the set of all logical addresses generated by a program
  - **Physical address space** is the set of all physical addresses generated by a program

Memory-Management Unit (MMU)

- Hardware device that at run time maps virtual to physical address
- To start, consider simple scheme where the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
  - Base register called **relocation register**
- The user program deals with *logical* addresses; it never sees the *real* physical addresses
  - Execution-time binding occurs when reference is made to location in memory
  - Logical address bound to physical addresses

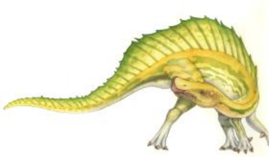


- Dynamic Loading
  - Routine is not loaded until it is called
  - Better memory-space utilization; unused routine is never loaded
  - All routines kept on disk in relocatable load format
  - Useful when large amounts of code are needed to handle infrequently occurring cases
  - OS can help by providing libraries to implement dynamic loading

Static Linking vs Dynamic Linking

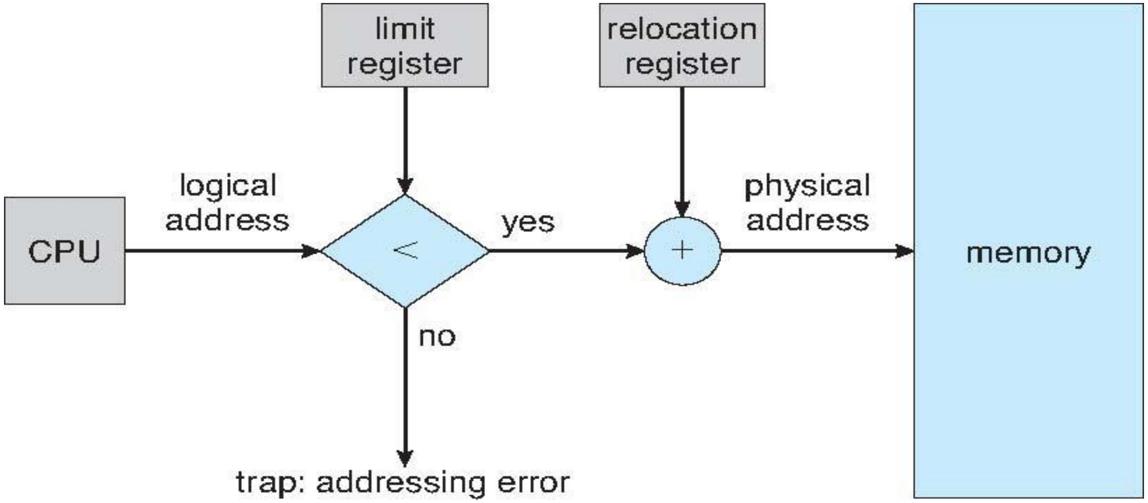
- Linking: loading system libraries
  - **Static linking** – system libraries and program code combined by the loader into the binary program image
  - **Dynamic linking** – linking postponed until execution time

- Dynamic Linking
  - Small piece of code, stub, used to locate the appropriate memory-resident library routine
  - Stub replaces itself with the address of the routine, and executes the routine
  - Operating system checks if routine is in processes’ memory address
  - If not in address space, add to address space
  - Dynamic linking is particularly useful for libraries
  - System also known as shared libraries
  - Consider applicability to patching system libraries
  - Versioning may be needed



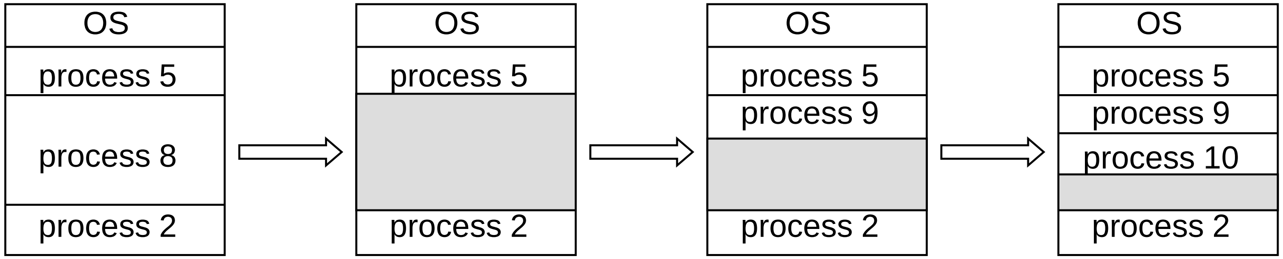
Contiguous Allocation

- Main memory usually into two **partitions**:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory; contained in single contiguous section of memory
- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - Base register contains value of smallest physical address
  - Limit register contains range of logical addresses (logical address < the limit register)
  - MMU maps logical address *dynamically*
  - Can then allow actions such as kernel code being **transient** and kernel changing size



Multiple-Partition Allocation

- Degree of multiprogramming limited by number of partitions
  - **Variable-partition** sizes for efficiency (sized to a given process' needs)
  - **Hole** – block of available memory; holes of various size are scattered throughout memory
- OS maintains information about both allocated partitions and free partitions (hole)
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it
  - Process exiting frees its partition, adjacent free partitions combined



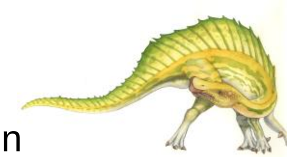
OS
process 5
process 8
process 2

OS
process 5
process 2

OS
process 5
process 9
process 2

OS
process 5
process 9
process 10
process 2

- Dynamic storage allocation
  - **First-fit**: Allocate the ***first*** hole that is big enough
  - **Best-fit**: Allocate the ***smallest*** hole that is big enough; ordered by size or search entire list
    - Produces the smallest leftover hole
  - **Worst-fit**: Allocate the ***largest*** hole; must also search entire list; worse than
    - Produces the largest leftover hole
  - First-fit and best-fit better than worst-fit in terms of speed and storage utilization



Fragmentation

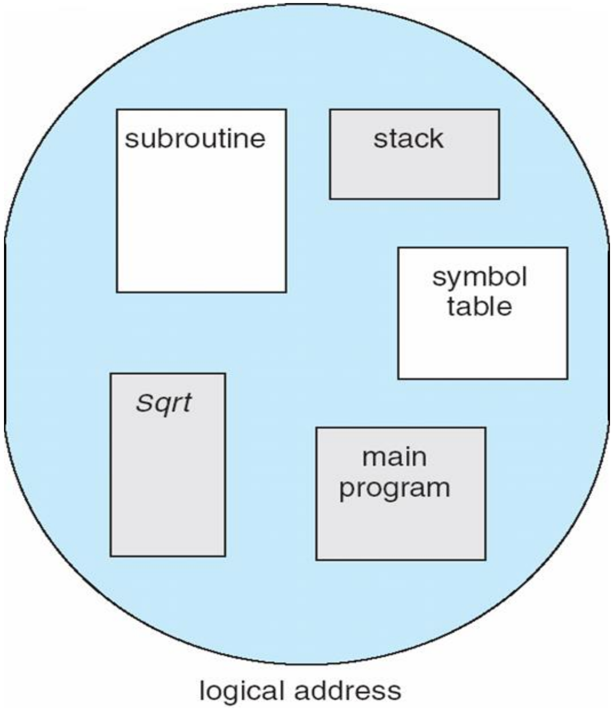
- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
  - 50-percent rule
  - Reduce external fragmentation by **compaction**
    - Shuffle memory contents to place all free memory together in one large block
    - Compaction is possible *only* if relocation is dynamic, and is done at execution time
  - I/O problem
    - Latch job in memory while it is involved in I/O
    - Do I/O only into OS buffers

- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory
  - This size difference is memory internal to a partition, but not being used

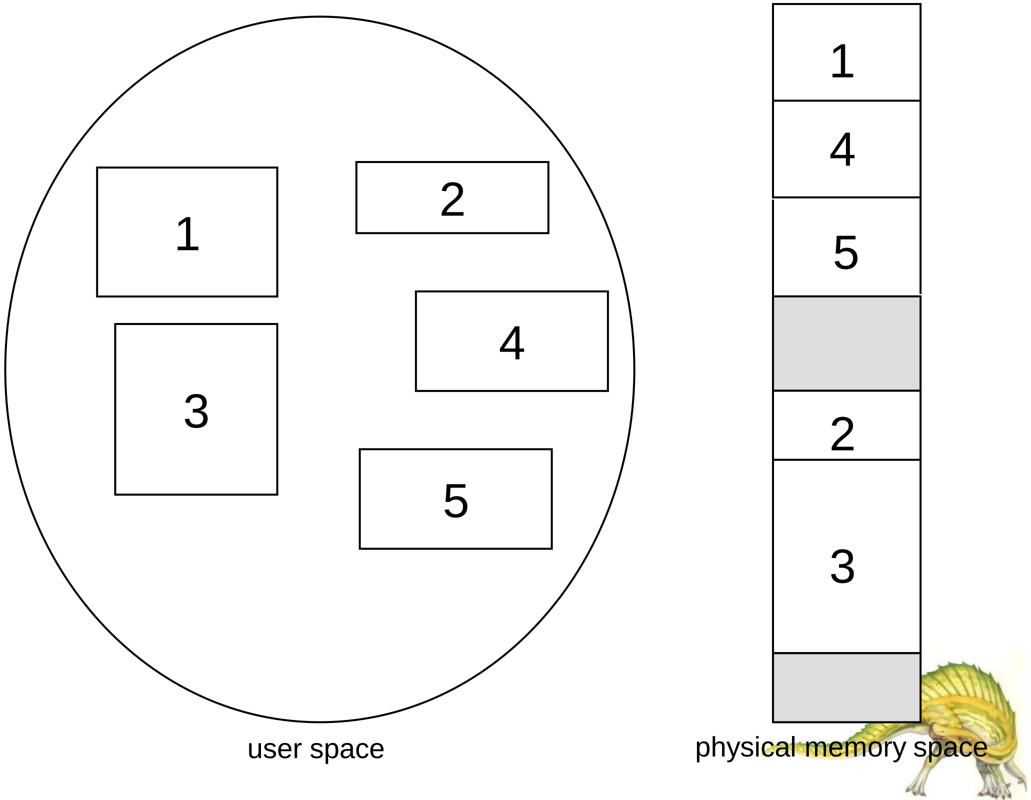
Segmentation

- ❑ Memory-management scheme that supports user view of memory
  - ❑ A program is a collection of segments, which is a logical unit
  - ❑ Such as, main program, procedure, method, common block, stack, etc.

User's View

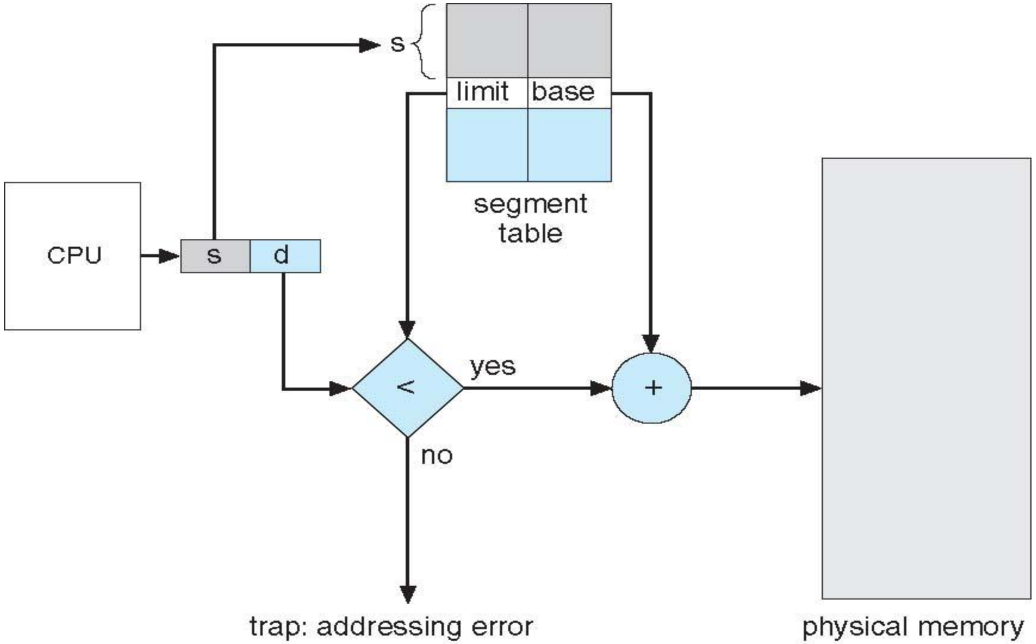


Logical View



Segmentation Architecture

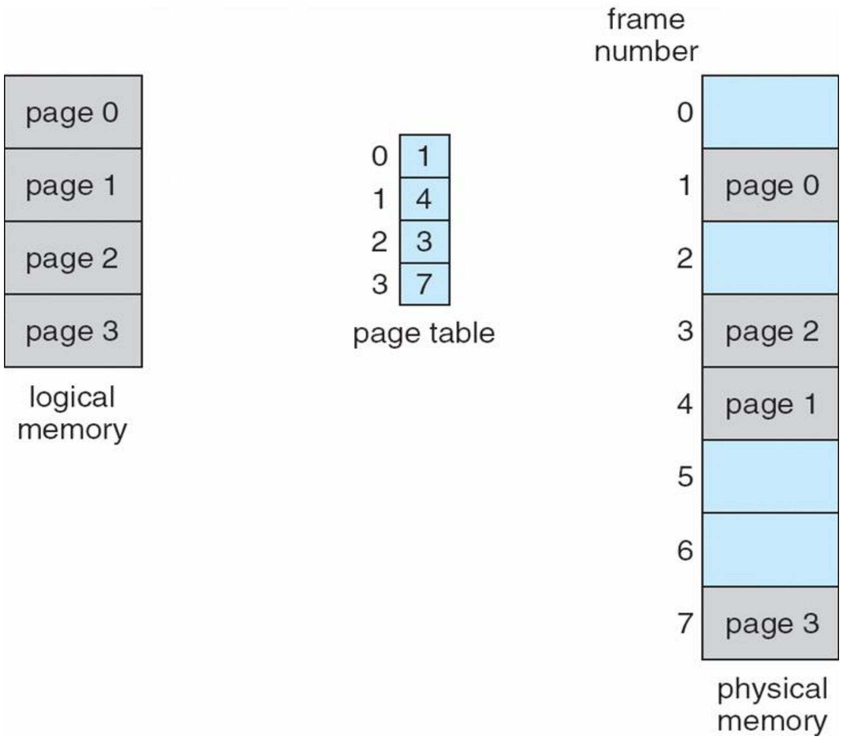
- Logical address consists of a two tuple: <segment-number, offset>,
- **Segment table** – maps two-dimensional physical addresses; each table entry has:
  - **base** – contains the starting physical address where the segments reside in memory
  - **limit** – specifies the length of the segment



- Supportive Registers
  - **Segment-table base register (STBR)** points to the segment table's location in memory
  - **Segment-table length register (STLR)** indicates number of segments used by a program;
    - segment number ***s*** is legal if ***s*** < **STLR**

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



- Method
  - **frames**: fixed-sized blocks in physical memory
    - Size is power of 2, between 512 bytes and 16 Mbytes
  - **pages**: blocks of same size in logical memory
  - Keep track of all free frames

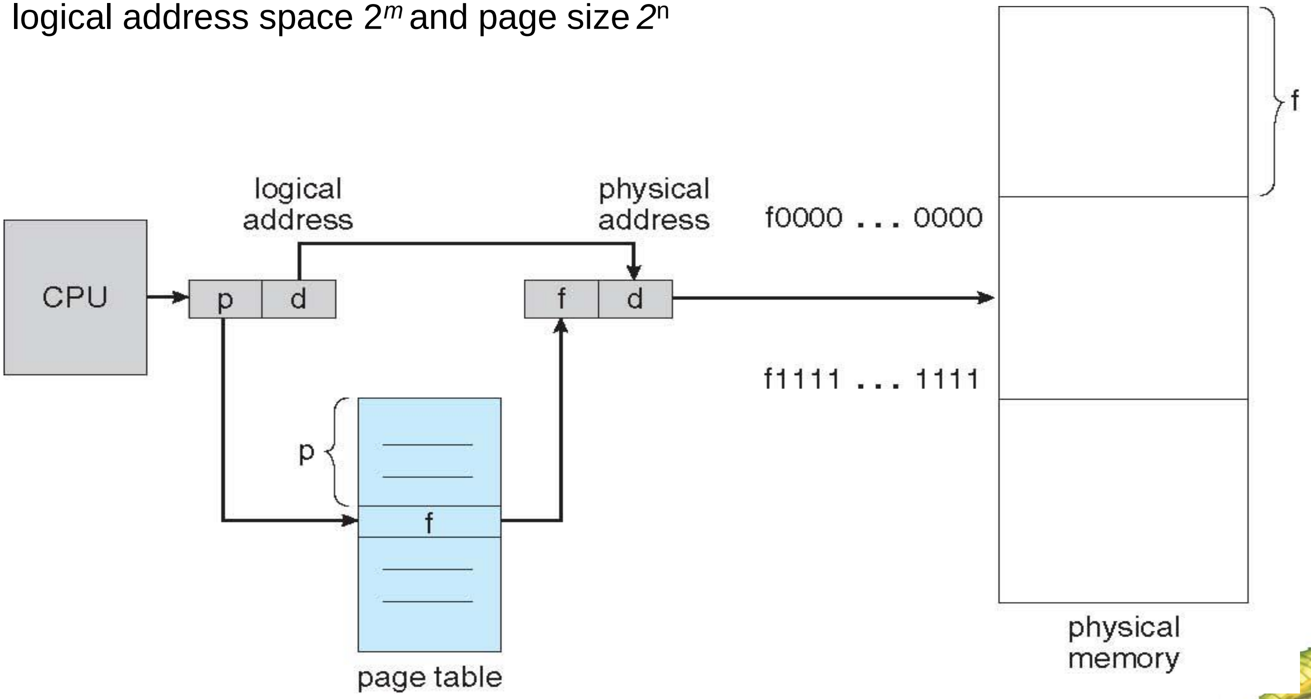
- To run a program of size ***N*** pages, need to find ***N*** free frames and load program
  - **page table** to translate logical to physical addresses

- Internal fragmentation

Address Translation Scheme

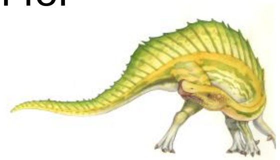
- Address (m bits) generated by CPU is divided into:
  - **Page number** (*p*) – used as an index into a **page table** which contains base address of each page in physical memory (m-n bits)
  - **Page offset** (*d*) – combined with base address to define the physical memory address that is sent to the memory unit (n bits)
  - For given logical address space 2^m and page size 2^n

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



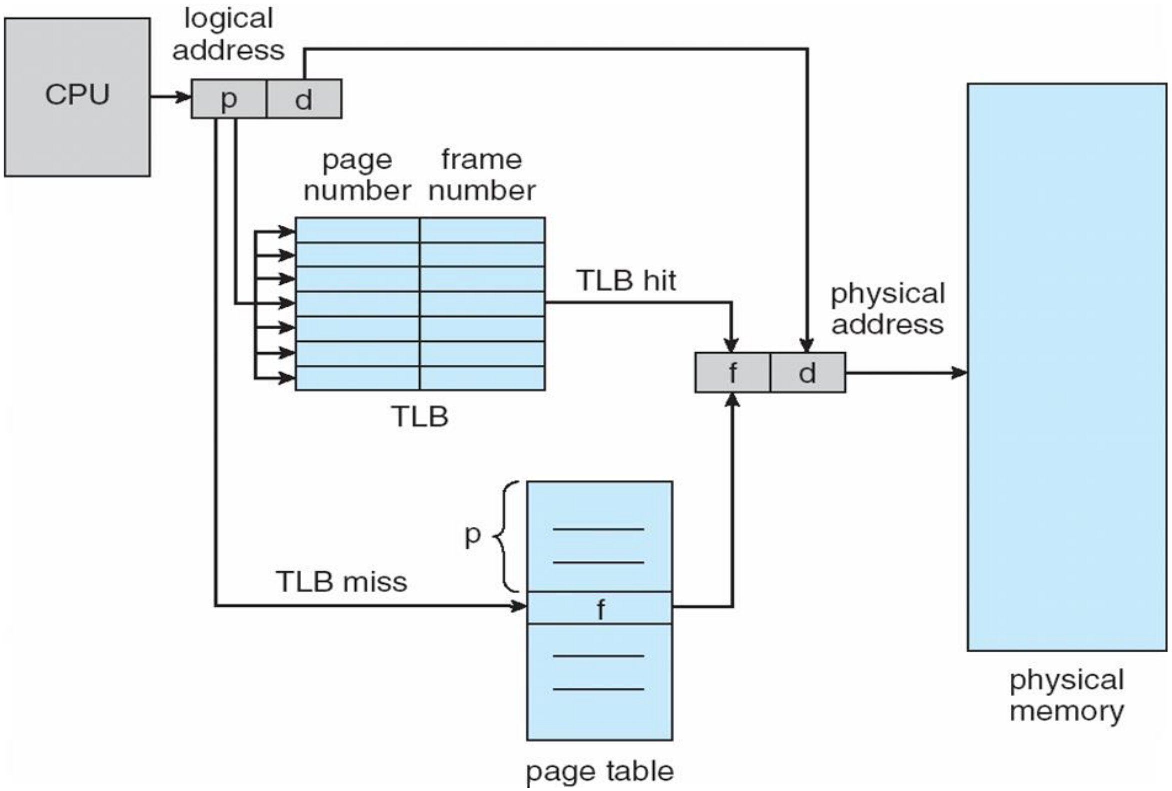
Implementation of Page Table

- Page table is kept in main memory
  - **Page-table base register** (**PTBR**) points to the page table
  - **Page-table length register** (**PTLR**) indicates size of the page table
- **Translation look-aside buffers** (**TLBs**) or **Associative memory**
  - Special fast-lookup hardware cache
  - Solve the two memory accesses problem for every data/instruction access
    - One for the page table and one for the data / instruction
  - TLBs typically small (64 to 1,024 entries)
- On a TLB miss, value is loaded into the TLB for faster access next time
  - Replacement policies must be considered
  - Some entries can be **wired down** for permanent fast access
- **Address-space identifiers** (**ASIDs**)
  - Uniquely identifies each process to provide address-space protection for that process; otherwise need to flush at every context switch



Paging Hardware With TLB

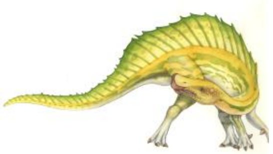
- Associative memory – parallel search
  - Address translation (p, d)
    - If p is in associative register, get frame # out
    - Otherwise get frame # from page table in memory



Sample Questions – Multiple Choices

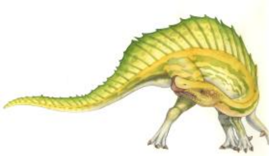
- What is wrong among the following sentences (choose only one)?
  a) Mutex locks suffer from busy waiting
  b) Mutex locks will be operated by acquire() and release()
  c) Mutex locks work for only two processes

- Three processes arrive in the order of P1, P2, and P3, and each of them need 10, 4, and 7 milliseconds. When the system applies FCFS scheduling, what is the average waiting time?
  a) 5
  b) 8
  c) 15
  d) 19.7



- Consider safe state. Find the wrong explanation.
  - a) If a system is in safe state, there are no deadlocks.
  - b) If a system is in unsafe state, there are always deadlocks.
  - c) If a system is in safe state, there should exist a sequence of all the processes, which are all safely complete.

- Three processes arrive in the order of P1, P2, and P3, and each of them need 10, 4, and 7 milliseconds. When the system applies FCFS scheduling, what is the average waiting time?
  - a) 5
  - b) 8
  - c) 15
  - d) 19.7



Sample Questions - Subject

- Suppose that two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes. What is this situation called?

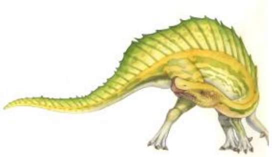
- Three processes of P1, P2, and P3 arrive in the system, and each of them need 20, 9, and 8 milliseconds. Apply FCFS scheduling and show in Gantt Chart. Suppose that they arrive in the order of P1, P2, and P3.

- Consider the following situation of the system. When P_0 request (2,1,0), can we grant the request?

	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	3 2 2	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

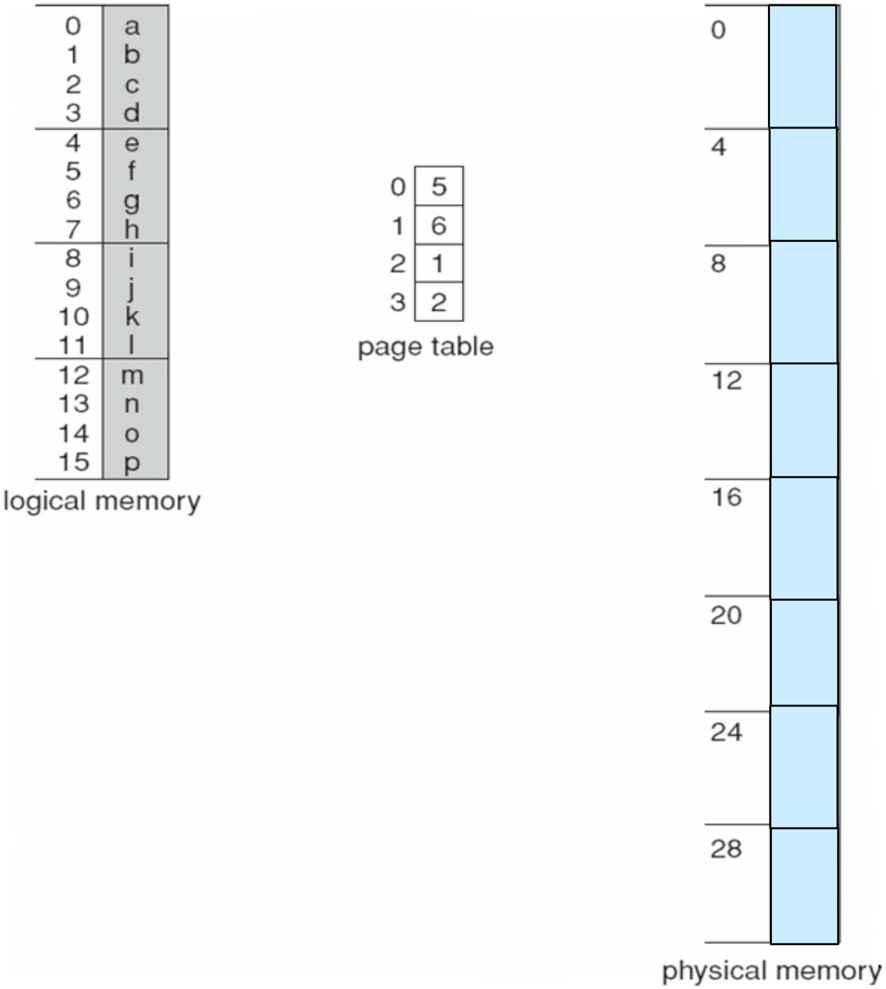
Sample Questions - Essay

- What is the adaptive mutexes used in Solaris?
- In monitors, how do wait and signal operations work differently comparing the operations in semaphores?
- Explain the difference between SJF and SRTF.
- What is the four conditions for deadlocks?
- What is the difference between dynamic linking and static linking?



Sample Questions - Essay

- See the below logical view of your program. When the page table is given, allocate all the alphabets in the physical memory.



0	a
1	b
2	c
3	d
4	e
5	f
6	g
7	h
8	i
9	j
10	k
11	l
12	m
13	n
14	o
15	p

logical memory

0	5
1	6
2	1
3	2

page table

physical memory

More Sample Questions

- In exercises
  - 5.3, 5.6, 5.14, 5.20
  - 6.3, 6.17, 6.19, 6.31
  - 7.3, 7.5, 7.22, 7.23
  - 8.3, 8.12, 8.28