CMRIT
CMR INSTITUTE OF TECHNOLOGY, BENGALURU.
ACCREDITED WITH A+ GRADE BY NAAC
No.132, AECS Layout
I.T. Park Road, Bangalore 560 037
T: +91 80 28524466/77, F : +91 80 28524630
E: info@cmr.ac.in I www.cmr.ac.in

DEPARTMENT OF COMPUTER SCIENCE

Operating System – 10CS64
IAT – 2 Solution

1. a) Consider the following snapshot of resource allocation at time **t_1**

	Allocation			**Request**			**Available**		
	A	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
P0	0	1	0	0	0	0	0	0	0
P1	2	0	0	2	0	2			
P2	3	0	3	0	0	0			
P3	2	1	1	1	0	0			
P4	0	0	2	0	0	2			

i) Show that the system is not deadlocked by generating one safe sequence.
ii) At instance **t_2**, P2 makes one additional request for instance of type C. Show that the system is deadlocked if the request is granted. Write down the deadlocked processes.

Need Matrix :	A	B	C
P0	0	0	0
P1	2	0	0
P2	0	0	0
P3	1	0	0
P4	0	0	2

Safe sequence<p4,p1>

2. a) What is a deadlock? What are the necessary conditions an OS must satisfy for a deadlock to occur?

A deadlock situation can occur if the following 4 conditions occur simultaneously in a system:-

1. **Mutual Exclusion**: Only one process is holding the resource at a time. If any other process requests for the resource, the requesting process must wait until the resource has been released.

2. **Hold and Wait**: A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by the other process.

3. **No Preemption**: Resources cannot be preempted i.e., only the process holding the resources must release it after the process has completed its task.

4. **Circular Wait**: A set {P0,P1……..Pn} of waiting process must exist such that P0 is waiting for a resource held by P1, P1 is waiting for a resource held by P2. Pn-1 is waiting for resource held by process Pn and Pn is waiting for the resource held by P1.

All the four conditions must hold for a deadlock to occur.

3 . a) Consider the following set of processes with length of the CPU burst time given in milliseconds:

Process	**Arrival Time**	**Burst Time**	**Priority**
P1	0	10	3
P2	0	1	1
P3	3	2	3
P4	5	1	4
P5	10	5	2

i) Draw five Gantt charts illustrating the execution of these processes using FCFS, SJF, SRTF, Priority and RR (**Quantum = 2**) scheduling.
ii) What is the turnaround time of each process for each of the scheduling algorithms?

(iii) What is waiting time of each process in (i) (5 Marks)

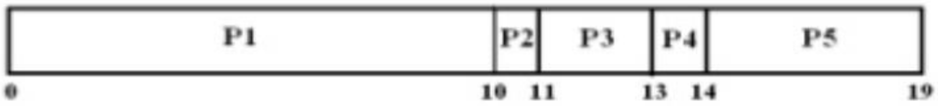
Ans:

Solution:

$$\text{Average turn around time} = \frac{\text{Sum of waiting time of individual process}}{\text{Number of processes}}$$

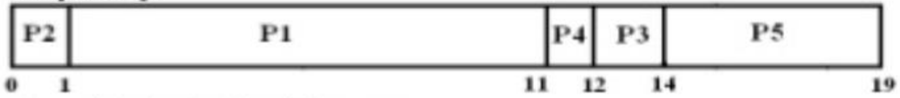
$$\text{Average waiting time} = \frac{\text{Sum of turn around time of individual process}}{\text{Number of processes}}$$

(i) FCFS:



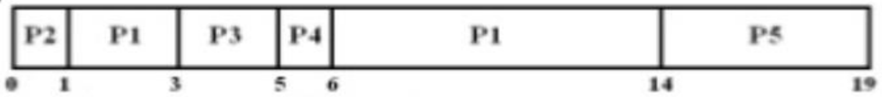
Average waiting time = (0+10+8+8+4)/5 = 6
Average turnaround time = (10+11+13+14+19)/5 = 13.4

(ii) SJF (non-preemptive):



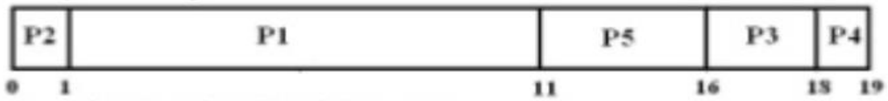
Average waiting time = (1+0+9+6+4)/5 = 4
Average turnaround time = (11+1+14+12+19)/5 = 11.4

SJF (preemptive):



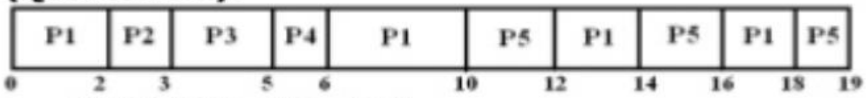
Average waiting time = (4+0+0+0+4)/5 = 1.6
Average turnaround time = (14+1+5+6+19)/5 = 9

(iii) Non-preemptive, Priority:



Average waiting time = (1+0+13+13+1)/5 = 5.6
Average turnaround time = (11+1+18+19+16)/5 = 13

(iv) Round Robin (Quantum=2):



Average waiting time = (8+2+0+0+4)/5 = 2.8
Average turnaround time = (18+3+5+6+19)/5 = 10.2

3. b) Briefly explain the methods for handling deadlock.

METHODS FOR HANDLING DEADLOCKS

Deadlock problem can be solved in one of three ways:

- Use a protocol to prevent or avoid deadlocks, ensuring that the system will *never* enter a deadlock state.
- Allow the system to enter a deadlock state, detect it, and recover.
- Ignore the problem altogether and pretend that deadlocks never occur in the system.

4 .a) Explain different scheduling criteria that must be kept in mind while choosing different scheduling algorithms.

Different CPU scheduling algorithms have different properties, and the choice of a particular algorithm may favour one class of processes over another. Many criteria have been suggested for comparing CPU scheduling algorithms. The criteria include the following:

- **CPU utilization** - The CPU must be kept as busy as possible. Conceptually, CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 to 90 percent .
- **Throughput** - If the CPU is busy executing processes, then work is done fast. One measure of work is the number of processes that are completed per time unit, called throughput.
- **Turnaround time -** From the point of view of a particular process, the important criterion is how long it takes to execute that process. The interval from the time of submission of a process to the time of completion is the turnaround time. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.
  Time spent waiting (to get into memory + ready queue + execution + I/O)
- **Waiting time -** The total amount of time the process spends waiting in the ready queue.
- **Response time -** The time taken from the submission of a request until the first response is produced is called the response time. It is the time taken to start responding. In interactive system, response time is given criterion.

It is desirable to **maximize** CPU utilization and throughput and to **minimize** turnaround time, waiting time, and response time.

4 . b) What is wait for graph? Describe resource allocation graph with and without deadlock.

Resource Allocation Graph:

Deadlocks are described by using a directed graph called system resource allocation graph. The graph consists of set of vertices (v) and set of edges (e).
The set of vertices (v) can be described into two different types of nodes

1. P={P1,P2……..Pn} i.e., set consisting of all active processes, represented by circle.
2. R={R1,R2……….Rn}i.e., set consisting of all resource types in the system, represented by rectangle. With dot representing the instances of that resource type.

There are two types of edges :

1. A directed edge from process Pi to resource type Rj denoted by Pi _ Ri indicates that Pi requested an instance of resource Rj and is waiting. This edge is called **Request edge**.
2. A directed edge Ri Pj signifies that resource Rj is held by process Pi. This is called **Assignment edge/ Allocation edge**.

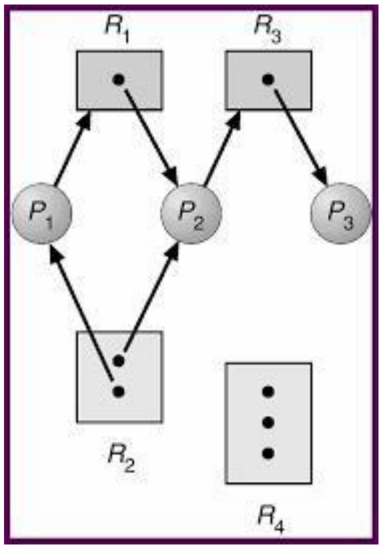
The sets P, *R*, and E:

P={P1,P2,P3}
R= {R1,R2,R3,R4}
E= {P1}

Resource instances:

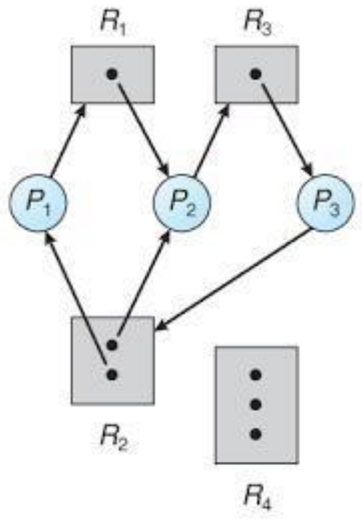
- One instance of resource type R1
- Two instances of resource type R2

- One instance of resource type *R3*
- Three instances of resource type*R4*

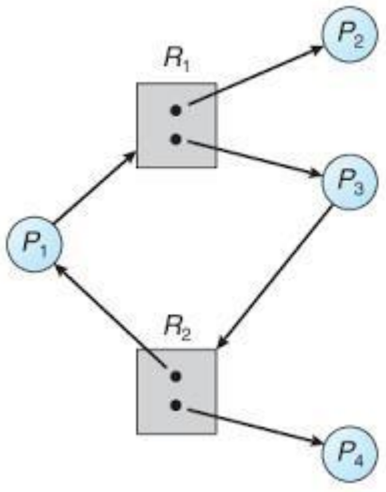


Resource Allocation Graph

- If the graph contains no cycle, then no process in the system is deadlock. If the graph contains a cycle then a deadlock may exist.



A)Resource allocation graph with a deadlock



B)Resource allocation graph with a cycle but no deadlock

- If each resource type has exactly one instance than a cycle implies that a deadlock has occurred. If each resource has several instances then a cycle do not necessarily implies that a deadlock has occurred.
- Note that a **request edge** can be converted into an **assignment edge** by reversing the direction of the arc when the request is granted.

5 . a) Explain critical section problem. What are the requirements that critical section problem must satisfy?

A solution to the problem must satisfy the following 3 requirements:

1. Mutual Exclusion

Only one process can be in its critical-section.

2. Progress

Only processes that are not in their remainder-section can enter their critical-section, and the selection of a process cannot be postponed indefinitely.

3. Bounded Waiting

There must be a bound 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 the request is granted.

5.b) What are Translation look aside buffers (TLB)? Explain TLB in detail with a neat diagram.

- Page lookups must be done for every memory reference, and whenever a process gets swapped in or out of the CPU, its page table must be swapped in and out too, along with the instruction registers, etc. It is therefore appropriate to provide hardware support for this operation, in order to make it as fast as possible and to make process switches as fast as possible also.

- One option is to use **a set of dedicated registers** for the page table. Here each register content is loaded, when the program is loaded into memory. For example, the DEC PDP-11 uses 16-bit addressing and 8 KB pages, resulting in only 8 pages per process. (It takes 13 bits to address 8 KB of offset, leaving only 3 bits to define a page number.)

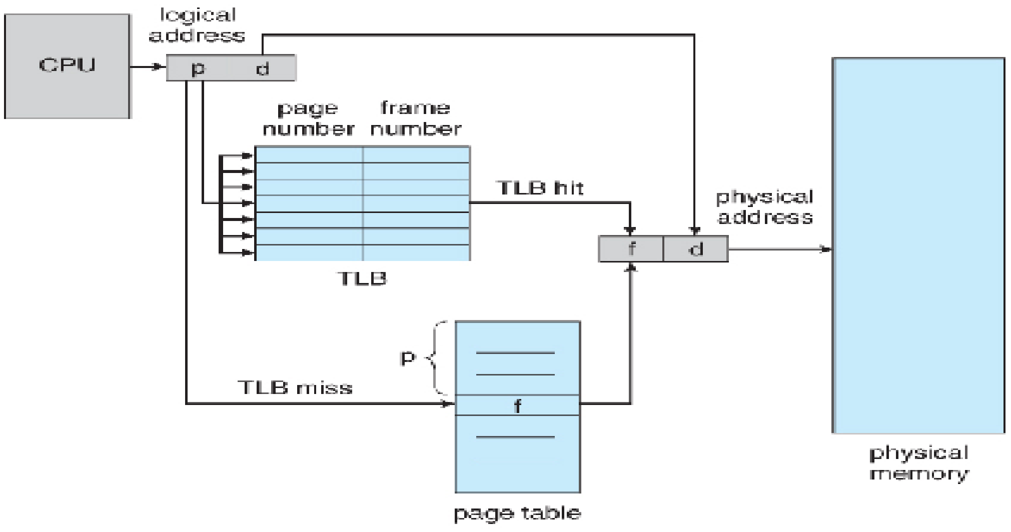


Fig : **Paging Hardware with Translation Lookaside Buffer(TLB)**

in main memory, and to use a single register (called the ***page-table base register, PTBR***) to record the address of the page table is memory.

- Process switching is fast, because only the single register needs to be changed.

- However memory access is slow, because every memory access now requires ***two*** memory accesses - One to fetch the frame number from memory and then another one to access the desired memory location.

- The solution to this problem is to use a very special high-speed memory device called the ***translation look-aside buffer, TLB.***

  - The benefit of the TLB is that it can search an entire table for a key value in parallel, and if it is found anywhere in the table, then the corresponding lookup value is returned.

  - It is used as a cache device.

  - Addresses are first checked against the TLB, and if the page is not there (a TLB miss), then the frame is looked up from main memory and the TLB is updated.

    - If the TLB is full, then replacement strategies range from ***least-recently used, LRU*** to random.
    - Some TLBs allow some entries to be ***wired down***, which means that they cannot be removed from the TLB. Typically these would be kernel frames.

- Some TLBs store ***address-space identifiers, ASIDs***, to keep track of which process "owns" a particular entry in the TLB. This allows entries from multiple processes to be stored simultaneously in the TLB without granting one process access to some other process's memory location. Without this feature the TLB has to be flushed clean with every process switch.

- The percentage of time that the desired information is found in the TLB is termed the ***hit ratio***.

- For example, suppose that it takes 100 nanoseconds to access main memory, and only 20 nanoseconds to search the TLB. So a TLB hit takes 120 nanoseconds total (20 to find the frame number and then another 100 to go get the data), and a TLB miss takes 220 (20 to search the TLB, 100 to go get the frame number, and then another 100 to go get the data.) So with an 80% TLB hit ratio, the average memory access time would be:

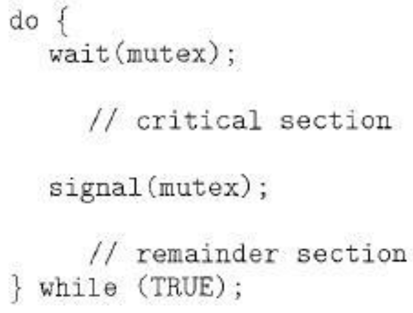
  0.80 * 120 + 0.20 * 220 = 140 nanoseconds

  for a 40% slowdown to get the frame number. A 98% hit rate would yield 122 nanoseconds average access time (you should verify this), for a 22% slowdown.

6. a) What do you mean by binary semaphore and counting semaphore? Explain the implementation of wait() and signal() semaphore operation.

Counting Semaphore
• The value of a semaphore can range over an unrestricted domain
Binary Semaphore
• The value of a semaphore can range only between 0 and 1.
• On some systems, binary semaphores are known as **mutex locks**, as they are locks that provide mutual-exclusion.

1) Solution for Critical-section Problem using Binary Semaphores
• Binary semaphores can be used to solve the critical-section problem for multiple processes.
• The 'n' processes share a semaphore *mutex* initialized to 1 (Figure 3.9).



```
do {
   wait(mutex);

      // critical section

   signal(mutex);

      // remainder section
} while (TRUE);
```

Figure 3.9 Mutual-exclusion implementation with

semaphores **2) Use of counting semaphores**

• Counting semaphores can be used to control access to a given resource consisting of a finite number o£ instances.
• The semaphore is initialized to the number of resources available.
• Each process that wishes to use a resource performs a wait() operation on the semaphore (thereby decrementing the count).
• When a process releases a resource, it performs a signal() operation (incrementing the count).
• When the count for the semaphore goes to 0, all resources are being used.
• After that, processes that wish to use a resource will block until the count becomes greater than 0.

3) Solving synchronization problems
• Semaphores can also be used to solve synchronization problems.
• For example, consider 2 concurrently running-processes:

1. Suppose we require that S2 be executed only after S1 has completed.
2. We can implement this scheme readily
   by letting P1 and P2 share a common semaphore *synch* initialized to 0, and
   by inserting the following statements in process P1

```
S1;
signal(synch);
```

and the following statements in process P2

```
wait(synch);
S2;
```

• Because *synch* is initialized to 0, P2 will execute S2 only after P1 has invoked signal (*synch*), which is after statement S1 has been executed.

Semaphore Implementation
• Main disadvantage of semaphore:
→ Busy waiting.
• **Busy waiting**: While a process is in its critical-section, any other process that tries to enter its critical-section must loop continuously in the entry code.
• Busy waiting wastes CPU cycles that some other process might be able to use productively.
• This type of semaphore is also called a **spinlock** (because the process "spins" while waiting for the lock).
• To overcome busy waiting, we can modify the definition of the wait() and signal() as follows:
→ When a process executes the wait() and finds that the semaphore-value is not positive, it must wait. However, rather than engaging in busy waiting, the process can *block* itself.
→ A process that is blocked (waiting on a semaphore S) should be restarted when some other process executes a signal(). The process is restarted by a wakeup().
• We assume 2 simple operations:
→ **block()** suspends the process that invokes it.
→ **wakeup(P)** resumes the execution of a blocked process P.
• We define a semaphore as follows:

•

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

• **Definition of wait():**

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

Definition of signal():

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

• This (critical-section) problem can be solved in two ways:

1. In a **uni-processor** environment
   Inhibit interrupts when the wait and signal operations execute.
   Only current process executes, until interrupts are re-enabled & the scheduler regains control.
2. In a **multiprocessor** environment
   Inhibiting interrupts doesn't work.
   Use the hardware / software solutions described above.

What are monitors? Explain its usage and implementation.

• **Monitor** is a high-level synchronization construct.
• It provides a convenient and effective mechanism for process synchronization.

Need for Monitors

• When programmers use semaphores incorrectly, following types of errors may occur:

1. Suppose that a process interchanges the order in which the wait() and signal() operations on the semaphore "mutex" are executed, resulting in the following execution:

```
signal(mutex);
     ...
  critical section
     ...
wait(mutex);
```

❿ In this situation, several processes may be executing in their critical-sections simultaneously, violating the mutual-exclusion requirement.

2. Suppose that a process replaces signal(mutex) with wait(mutex). That is, it executes

```
wait(mutex);
     ...
  critical section
     ...
wait(mutex);
```

❿ In this case, a deadlock will occur.

• Suppose that a process omits the wait(mutex), or the signal(mutex), or both. ➢ In this case, either mutual-exclusion is violated or a deadlock will occur.

Monitors Usage

• A ***monitor type*** presents a set of programmer-defined operations that are provided to ensure mutual-exclusion within the monitor.
• It also contains (Figure 3.12):

declaration of variables
bodies of procedures(or functions).

• A procedure defined within a monitor can access only those variables declared locally within the monitor and its formal-parameters.

Similarly, the local-variables of a monitor can be accessed by only the local-procedures.

```
monitor monitor name
{
   /* shared variable declarations */

   function P1 ( . . . ) {
      . . .
   }

   function P2 ( . . . ) {
      . . .
   }

         .
         .
         .
   function Pn ( . . . ) {
      . . .
   }

   initialization_code ( . . . ) {
      . . .
   }
}
```

Figure 3.12 Syntax of a monitor

☐ Only one process at a time is active within the monitor (Figure 3.13).

☐ To allow a process to wait within the monitor, a condition variable must be declared, as

```
condition x, y;
```

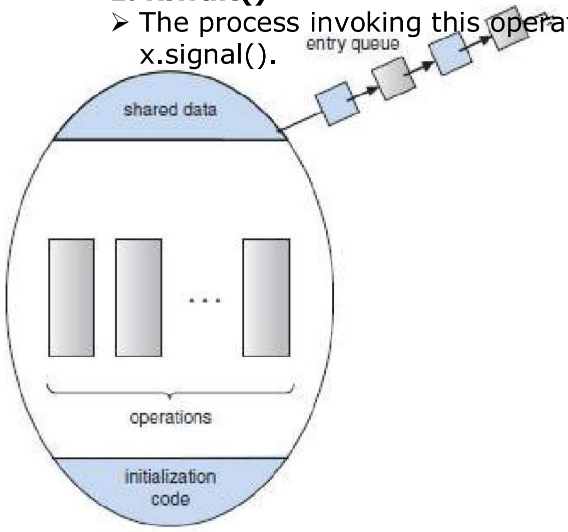
☐ Condition variable can only be used with the following 2 operations (Figure 3.14):

1. **x.signal()**

➢ This operation resumes exactly one suspended process. If no process is suspended, then the signal operation has no effect.

2. **x.wait()**

➢ The process invoking this operation is suspended until another process invokes x.signal().



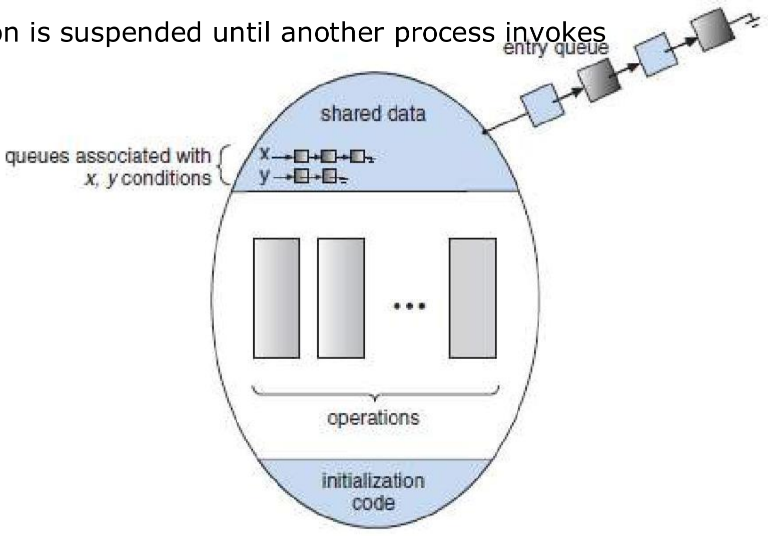


Figure 3.13 Schematic view of a monitor Figure 3.14 Monitor with condition variables

- Suppose when the x.signal() operation is invoked by a process P, there exists a suspended process Q associated with condition x.
- Both processes can conceptually continue with their execution. Two possibilities exist:

1. Signal and wait

➢ P either waits until *Q* leaves the monitor or waits for another condition.

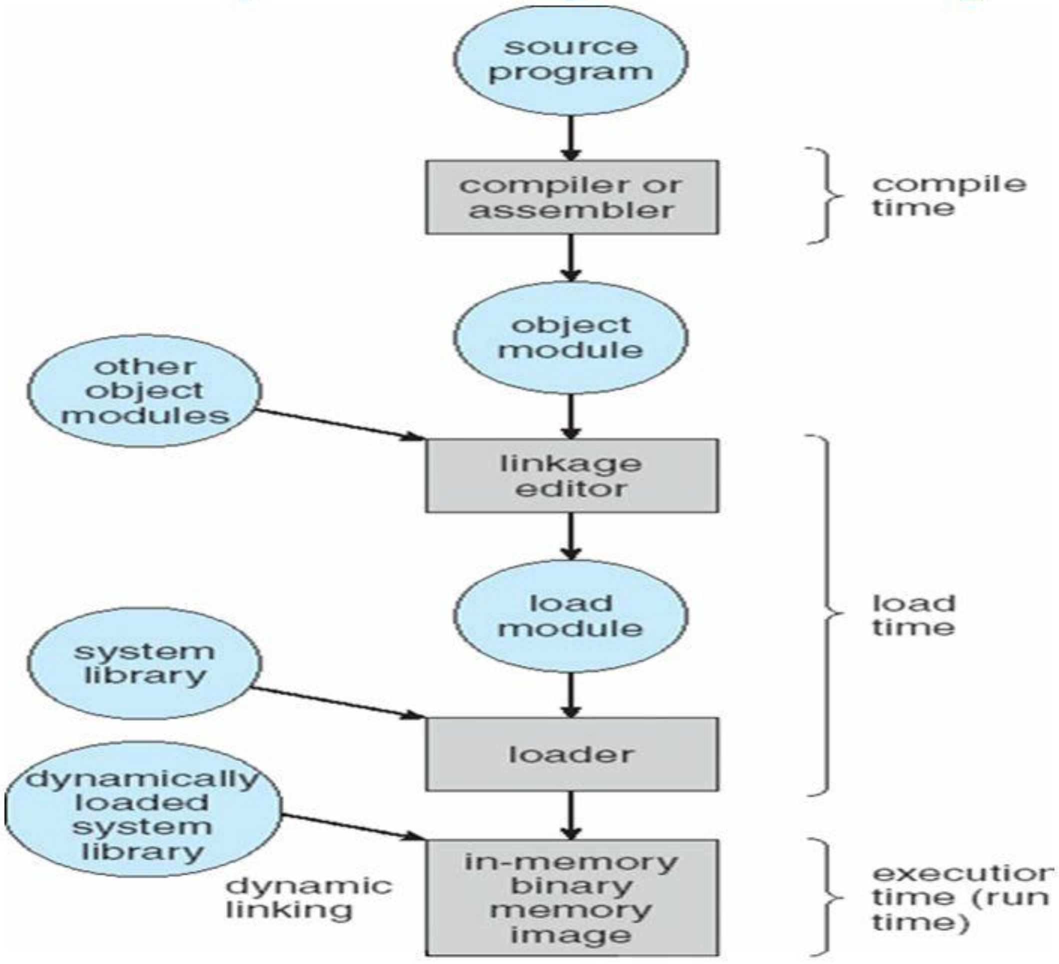
2. Signal and continue

➢ Q either waits until P leaves the monitor or waits for another condition.

7. a) Explain the multistep processing of a user program.

- User programs typically refer to memory addresses with symbolic names such as "i", "count", and "average Temperature". These symbolic names must be mapped or ***bound*** to physical memory addresses, which typically occurs in several stages:
    - **Compile Time**- If it is known at compile time where a program will reside in physical memory,then ***absolute code*** can be generated by the compiler, containing actual physical addresses. However if the load address changes at some later time, then the program will have to be recompiled.
    - **Load Time-** If the location at which a program will be loaded is not known at compile time, thenthe compiler must generate ***relocatable code***, which references addresses relative to the start of the program. If that starting address changes, then the program must be reloaded but not recompiled.
    - **Execution Time-** If a program can be moved around in memory during the course of its execution,then binding must be delayed until execution time.
- Figure 8.3 shows the various stages of the binding processes and the units involved in each stage:

Multistep Processing of a User Program



7. b) Explain internal fragmentation and external fragmentation

- Note that paging is like having a table of relocation registers, one for each page of the logical memory.
- There is **no external fragmentation** with paging. All blocks of physical memory are used, and there are no gaps in between and no problems with finding the right sized hole for a particular chunk of memory.
- There is, however, internal fragmentation. Memory is allocated in chunks the size of a page, and on the average, the last page will only be half full, wasting on the average half a page of memory per process.
- Larger page sizes waste more memory, but are more efficient in terms of overhead. Modern trends have been to increase page sizes, and some systems even have multiple size pages to try and make the best of both worlds.

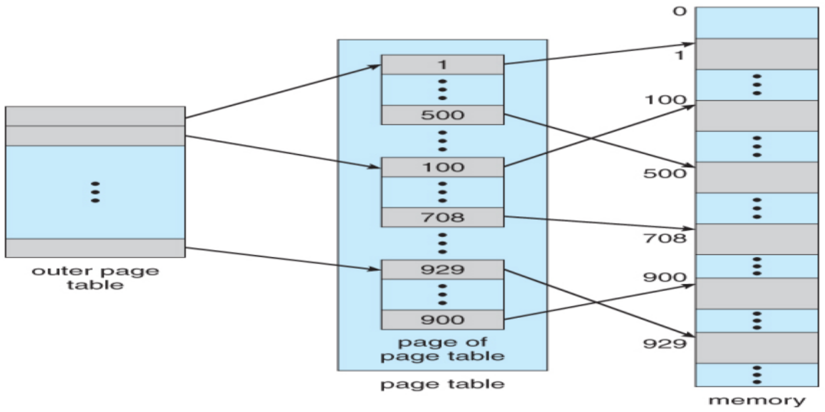
- Consider the following example, in which a process has 16 bytes of logical memory, mapped in 4 byte (Presumably some other processes would be consuming the remaining 16 bytes of physical memory.)pages into 32 bytes of physical memory.

8. a) Briefly explain the structure of page table.

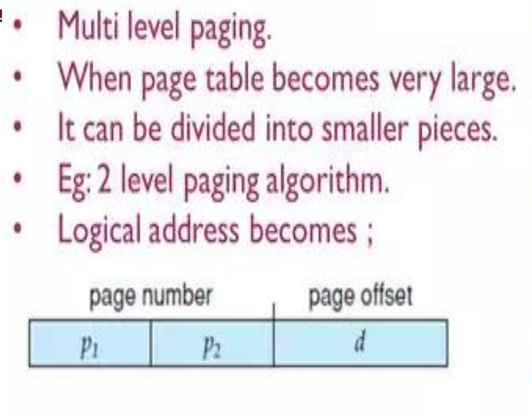
8.5.1 Hierarchical Paging

- This structure supports two or more page tables at different levels (tiers).
- Most modern computer systems support logical address spaces of 2^32 to 2^64.

A two-level page-table scheme

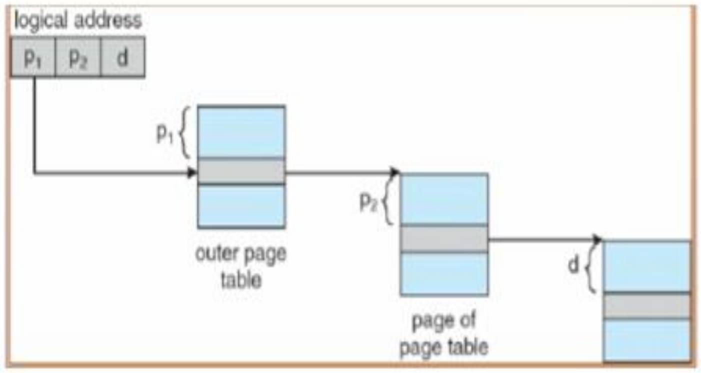


- VAX Architecture divides 32-bit addresses into 4 equal sized sections, and each page is 512 bytes, yielding an address form

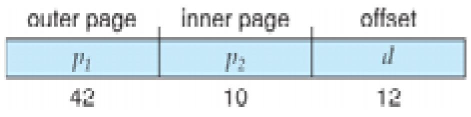


- Multi level paging.
- When page table becomes very large.
- It can be divided into smaller pieces.
- Eg: 2 level paging algorithm.
- Logical address becomes ;

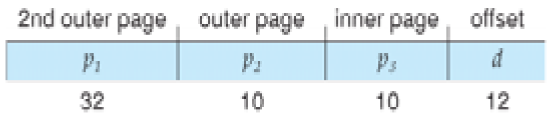
page number		page offset
p_1	p_2	d



Two-level Paging Scheme:



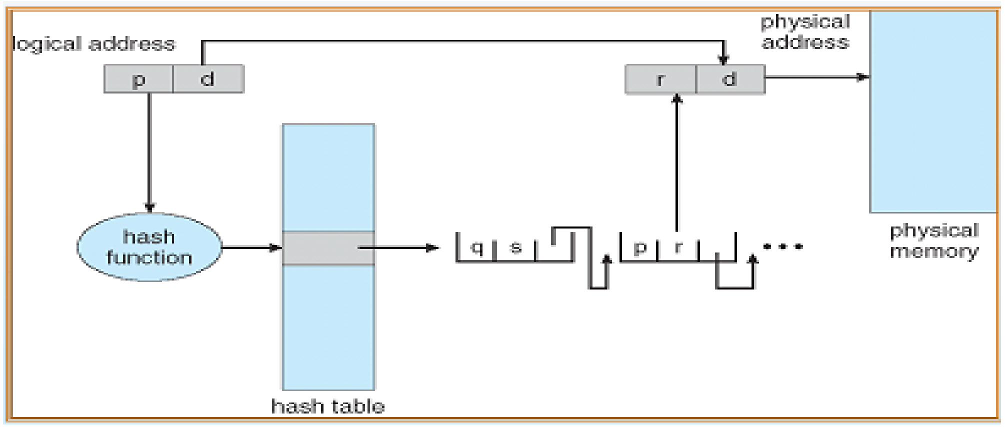
Three-level Paging Scheme:



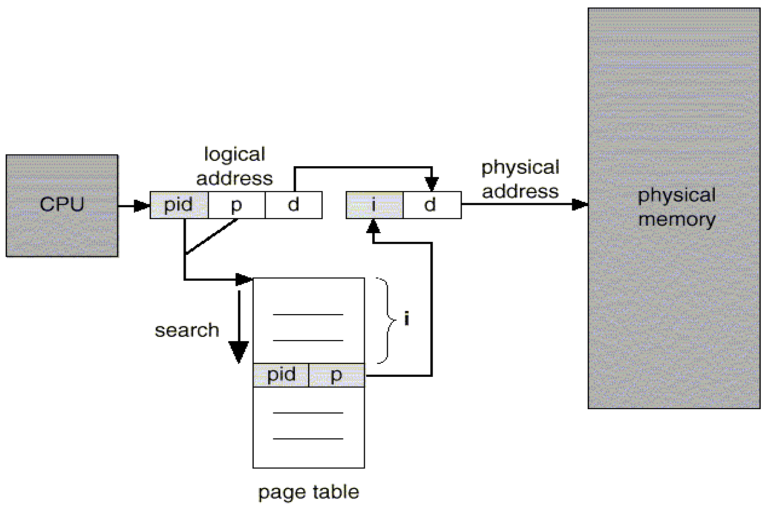
- With a 64-bit logical address space and 4K pages, there are 52 bits worth of page numbers, which is still too many even for two-level paging. One could increase the paging level, but with 10-bit page tables it would take 7 levels of indirection, which would be prohibitively **slow memory access**. So some other approach must be used.

8.5.2 Hashed Page Tables

- One common data structure for accessing data that is sparsely distributed over a broad range of possible values is with ***hash tables***. Figure 8.16 below illustrates a ***hashed page table*** using chain-and-bucket hashing:



8.5.3 Inverted Page Tables:



- Another approach is to use an ***inverted page table***. Instead of a table listing all of the pages for a particular process, an inverted page table lists all of the pages currently loaded in memory, for all processes. (i.e. there is one entry per ***frame*** instead of one entry per ***page***.)

- Access to an inverted page table can be slow, as it may be necessary to search the entire table in order to find the desired page .

- The 'id' of process running in each frame and its corresponding page number is stored in the page table.

8.b) Explain the producer-consumer problem of synchronization.

• Suppose that we wanted to provide a solution to **producer-consumer problem** that fills all buffers. We can do so by having an variable *counter* that keeps track of the no. of full buffers.

Initially, *counter=0*.

➢ *counter* is incremented by the producer after it produces a new buffer.

➢ *counter* is decremented by the consumer after it consumes a buffer.

• **Shared-data:**

```
#define BUFFER_SIZE 10

typedef struct {
    . . .
}item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

Producer Process:

```
while (true) {
     /* produce an item in next_produced */

     while (counter == BUFFER_SIZE)
        ; /* do nothing */

     buffer[in] = next_produced;
     in = (in + 1) % BUFFER_SIZE;
```

Consumer Process:

```
while (true) {
     while (counter == 0)
        ; /* do nothing */

     next_consumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     counter--;
```

• A situation where several processes access & manipulate same data concurrently and the outcome of the execution depends on particular order in which the access takes place, is called a ***race condition***.

• Example:

counter++ could be implemented as:

$$\begin{aligned} register_1 &= \texttt{counter} \\ register_1 &= register_1 + 1 \\ \texttt{counter} &= register_1 \end{aligned}$$

counter— may be implemented as:

$$\begin{aligned} register_2 &= \texttt{counter} \\ register_2 &= register_2 - 1 \\ \texttt{counter} &= register_2 \end{aligned}$$

• Consider this execution interleaving with *counter = 5* initially:

• The value of *counter* may be either 4 or 6, where the correct result should be 5. This is an example for race condition.

• To prevent race conditions, concurrent-processes must be synchronized.

9. a) Explain dining philosopher's solution using monitors.

Dining-Philosophers Solution Using Monitors

• The restriction is

➢ A philosopher may pick up her chopsticks only if both of them are available.

• Description of the solution:

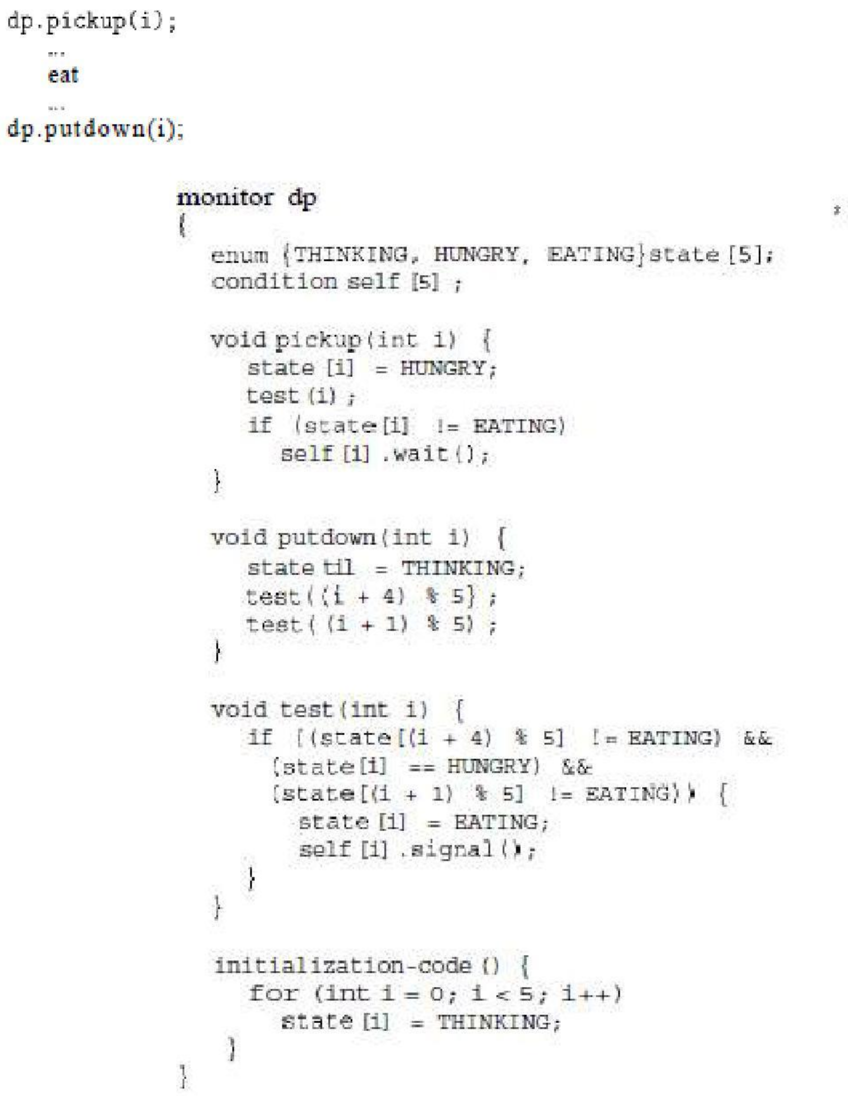
☐ The distribution of the chopsticks is controlled by the monitor *dp* (Figure 3.15).

☐ Each philosopher, before starting to eat, must invoke the operation *pickup()***.** This act may result in the suspension of the philosopher process.

☐ After the successful completion of the operation, the philosopher may eat.

☐ Following this, the philosopher invokes the *putdown()* operation.

☐ Thus, philosopher *i* must invoke the operations *pickup()* and *putdown()* in the following sequence:



```
dp.pickup(i);
    ...
    eat
    ...
dp.putdown(i);
```

```
monitor dp
{
   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 til  = THINKING;
      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;
    }
}
```

Figure 3.15 A monitor solution to the dining-philosopher problem

9. b) Explain the process of recovery from deadlock

Recovery From Deadlock

- There are two basic approaches to recovery from deadlock:
    1. Process Termination

2. Resource Preemption

7.8.1 Process Termination

- Two basic approaches to terminate processes:

  - **Abort all deadlocked processes.** This method breaks the deadlock cycle by terminating all the processes involved in deadlock. But, at a great expense, the deadlocked processes may have computed for a long time, and the results of these partial computations must be discarded and probably will have to be recomputed later.
  - **Abort one process at a time until the deadlock cycle is eliminated.** This method incurs considerable overhead, since, after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.

- In the latter case there are many factors that can go into deciding which processes to terminate next:
  1. Process priorities.
  2. How long the process has been running, and how close it is to finishing.
  3. How many and what type of resources is the process holding. (Are they easy to preempt and restore?)
  4. How many more resources does the process need to complete.
  5. How many processes will need to be terminated
  6. Whether the process is interactive or batch.
  7. (Whether or not the process has made non-restorable changes to any resource.)

7.7.2 Resource Preemption

- When preempting resources to relieve deadlock, there are three important issues to be addressed:
  1. **Selecting a victim -** Deciding which resources to preempt from which processes involves many of the same decision criteria outlined above.
  2. **Rollback -** Ideally one would like to roll back a preempted process to a safe state prior to the point at which that resource was originally allocated to the process. Unfortunately it can be difficult or impossible to determine what such a safe state is, and so the only safe rollback is to roll back all the way back to the beginning. (i.e. abort the process and make it start over.)

3. **Starvation** – There are chances that the same resource is picked from processas a victim, every time the deadlock occurs and this continues. This is starvation. A count can be kept on number of rollback of a process and the process has to be a victim for finite number of times only .

10. a) Briefly explain the classical problems of synchronization.

Classic Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

The Bounded-Buffer Problem

• The bounded-buffer problem is related to the producer consumer problem.
• There is a pool of n buffers, each capable of holding one item.
• **Shared-data**

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0
```

where,

mutex provides mutual-exclusion for accesses to the buffer-pool.
empty counts the number of empty buffers.
full counts the number of full buffers.

3. The symmetry between the producer and the consumer.
   The producer produces full buffers for the consumer.
   The consumer produces empty buffers for the producer.

• **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);
```

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);
```

Readers-Writers Problem

• A data set is shared among a number of concurrent processes.
• **Readers** are processes which want to only read the database (DB).

1. Problem:
   Obviously, if 2 readers can access the shared-DB simultaneously without any problems.
   However, if a writer & other process (either a reader or a writer) access the shared-DB
   simultaneously, problems may arise.

Solution:
   ❼ The writers must have exclusive access to the shared-DB while writing to the DB.

• **Shared-data**

```
semaphore mutex, wrt;
int readcount;
```

where,
   ❿ *mutex* is used to ensure mutual-exclusion when the variable *readcount* is updated.
   ❿ *wrt* is common to both reader and writer processes.
   wrt is used as a mutual-exclusion semaphore for the writers.
   wrt is also used by the first/last reader that enters/exits the critical-section.
   ❿ *readcount* counts no. of processes currently reading the object.

Initialization

mutex = 1, wrt = 1, readcount = 0

Writer Process:

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

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);
```

• The readers-writers problem and its solutions are used to provide **reader-writer locks** on some systems.
• The mode of lock needs to be specified:
   1. **read mode**
   ➢ When a process wishes to read shared-data, it requests the lock in *read mode*.
   2. **write mode**
   ➢ When a process wishes to modify shared-data, it requests the lock in *write mode*.
• Multiple processes are permitted to concurrently acquire a lock in read mode,
   but only one process may acquire the lock for writing.

1. These locks are most useful in the following situations:

In applications where it is easy to identify
which processes only read shared-data and
which threads only write shared-data.
In applications that have more readers than writers.

Dining-Philosophers Problem Problem statement:

There are 5 philosophers with 5 chopsticks (semaphores).
A philosopher is either eating (with two chopsticks) or thinking.
The philosophers share a circular table (Figure 3.10).
The table has a bowl of rice in the center and 5 single chopsticks.
From time to time, a philosopher gets hungry and tries to pick up the 2 chopsticks that are closest to her.
A philosopher may pick up only one chopstick at a time.
Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor.
When hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks.
When she is finished eating, she puts down both of her chopsticks and starts thinking again.

• Problem objective:
To allocate several resources among several processes in a deadlock-free & starvation-free manner.

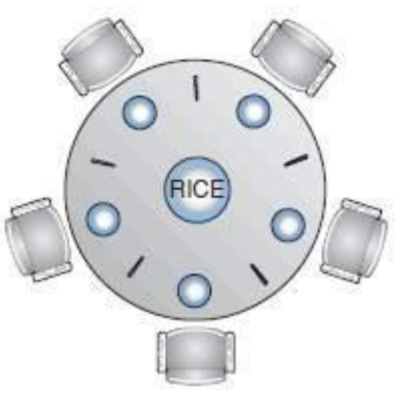
☐ Solution:
Represent each chopstick with a semaphore (Figure 3.11).
A philosopher tries to grab a chopstick by executing a wait() on the semaphore.
The philosopher releases her chopsticks by executing the signal() on the semaphores.
This solution guarantees that no two neighbors are eating simultaneously.

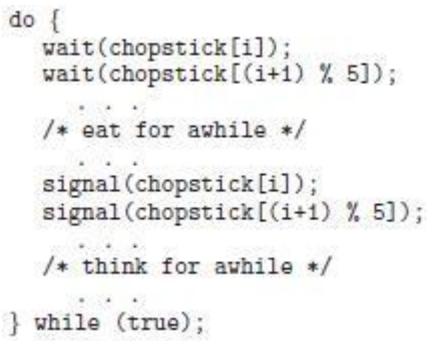
Shared-data

semaphore chopstick[5];

Initialization

chopstick[5]={1,1,1,1,1}.





```
do {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   /* eat for awhile */
      . . .
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      . . .
   /* think for awhile */
      . . .
} while (true);
```

Figure 3.10 Situation of dining philosophers

Figure 3.11 The structure of philosopher

☐ Disadvantage:

Deadlock may occur if all 5 philosophers become hungry simultaneously and grab their left chopstick. When each philosopher tries to grab her right chopstick, she will be delayed forever.

☐ Three possible remedies to the deadlock problem:

1. Allow **at most 4** philosophers to be sitting simultaneously at the table.
2. Allow a philosopher to pick up her chopsticks **only if both chopsticks are available**.
3. Use an **asymmetric solution**; i.e. an odd philosopher picks up first her left chopstick and then her right chopstick, whereas an even philosopher picks up her right chopstick and then her left chopstick.