Q1

1) Address Binding: To execute a program, it must be brought into memory and placed within a process. Depending on the memory management in use, the process may be moved between disk and memory during its execution. The processes on the disk that are waiting to be brought into memory for execution form the input queue. Addresses in the source program are generally symbolic. Each binding is a mapping from one address space to another. The binding of instructions and data to memory addresses can be done at any step along the way: Compile time, Load time, or Execution time.

2) Multithreaded Process: A thread is a basic unit of CPU utilization; it comprises a thread ID, a program counter, a register set, and a stack. It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals. A traditional (or heavyweight) process has a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time.
3) Graphical user interface: A strategy for interfacing with the operating system is through a user friendly graphical user interface or CUI. Rather than having users directly enter commands via a command-line interface, a CUI allows provides a mouse-based window-and-menu system as an interface. A CUI provides a desktop metaphor where the mouse is moved to position its pointer on images, or icons, on the screen (the desktop) that represent programs, files, directories, and system functions. Depending on the mouse pointer's location, clicking a button on the mouse can invoke a program, select a file or directory-known as a folderor or pull down a menu that contains commands.

4) Virtual Machine: The fundamental idea behind a virtual machine is to abstract the hardware of a single computer (the CPU, memory, disk drives, network interface cards, and so forth) into several different execution environments, thereby creating the illusion that each separate execution environment is running its own private computer. By using CPU scheduling and virtual-memory techniques an operating system can create the illusion that a process has its own processor with its own (virtual) memory. Normally, a process has additional features, such as system calls and a file system, that are not provided by the bare hardware. The virtual-machine approach does not provide any such additional functionality but rather provides an interface that is identical to the underlying bare hardware. Each process is provided with a (virtual) copy of the underlying computer. There are several reasons for creating a virtual machine, all of which are fundamentally related to being able to share the same hardware yet run several different execution environments (that is, different operating systems) concurrently.

5) Context Switch: Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process. This task is known as a context switch. When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run. Context-switch time is pure overhead, because the system does no useful work while switching. Its speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied, and the existence of special instructions (such as a single instruction to load or store all registers). Typical speeds are a few milliseconds.

Q2

1-

First Fit Algorithm

200 KB	200 KB	145 KB
380 KB	100 KB	100 KB
400 KB	400 KB	400 KB
150 KB	150 KB	150 KB
75 KB	75 KB	75 KB
Memory management Table	P1	P2

35 KB	35 KB	35 KB
100 KB	100 KB	100 KB
400 KB	30 KB	30 KB
150 KB	150 KB	150 KB
75 KB	75 KB	75 KB
P3	P4	P5 not allocated

Best Fit Algorithm

200 KB	200 KB	200 KB
380 KB	100 KB	100 KB
400 KB	400 KB	400 KB
150 KB	150 KB	150 KB
75 KB	75 KB	20 KB
Memory management Table	P1	P2
200 KB	200 KB	30 KB
100 KB	100 KB	100 KB
400 KB	30 KB	30 KB
40 KB	40 KB	40 KB
20 KB	20 KB	20 KB
P3	P4	P5

Worst Fit Algorithm

200 KB	200 KB	200 KB
380 KB	380 KB	325 KB
400 KB	120KB	120KB
150 KB	150 KB	150 KB
75 KB	75 KB	75 KB
Memory management Table	P1	P2
200 KB	200 KB	200 KB
215 KB	215 KB	45 KB
120KB	120KB	120KB
150 KB	150 KB	150 KB
75 KB	75 KB	75 KB
	P4 not allocated	P5

2-time and efficiency
3- Best fit

Q3//
Semaphore

A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait( ) and signal ( ). wait ( ) means "to test" and signal ( ) means "to increment". The definition of wait () is as follows:

```
wait (S) {
while S <= 0
; // no-op
S--;
```

The definition of signal ( ) is as follows:

```
signal (S)
S++;
```

All the modifications to the integer value of the semaphore in the wait ( ) and signal ( ) operations must be executed indivisibly.
The main disadvantage of the semaphore definition given here is that it requires 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. This continual looping is clearly a problem in a real multiprogramming system, where a single CPU is shared among many processes. Busy waiting wastes CPU cycles that some other process might be able to use productively.
The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. The event in question is the execution of a signal ( ) operation. When such a state is reached, these processes are said to be deadlocked. Another problem related to deadlocks is indefinite blocking, or starvation a situation in which processes wait indefinitely within the semaphore. Indefinite blocking may occur if we add and remove processes from the list associated with a semaphore in LIFO (last-in, first-out) order.
Although semaphores provide a convenient and effective mechanism for process synchronization, using them incorrectly can result in timing errors that are difficult to detect, since these errors happen only if some particular execution sequences take place and these sequences do not always occur.

A semaphore solution for Dining Philosopher Problem is to represent each chopstick with a semaphore. A philosopher tries to grab a chopstick by executing a wait ( ) operation on that semaphore; she releases her chopsticks by executing the signal ( ) operation
on the appropriate semaphores. Thus, the shared data are

```
semaphore chopstick [5] ;
```

where all the elements of chopstick are initialized to 1.
Although this solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because it could create a deadlock. Suppose that all five philosophers become hungry simultaneously and each grabs her left chopstick. All the elements of chopstick will now be equal to O. When each philosopher tries to grab her right chopstick, she will be delayed forever.

```
do {
wait (chopstick[i] );
wait(chopstick[(i+l) % 5]);
// eat
signal (chopstick[i] );
signal (chopstick[(i+l) % 5]);
// think
}while (TRUE);
```

The structure of philosopher *i.*

A Monitor is a developed high-level synchronization construct. A type, or abstract data type, encapsulates private data with public methods to operate on that data. A monitor type presents a set of programmer-defined operations that are provided mutual exclusion within the monitor. The monitor type also contains the declaration of variables whose values define the state of an instance of that type, along with the bodies of procedures or functions that operate on those variables. The representation of a monitor type cannot be used directly by the various processes. Thus, 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
procedure Pl ( . . . ) {
. . .
}
procedure P2 ( . . . ) {
. . .
}
procedure Pn ( . . . ) {
. . .
}

initialization code ( . . . ) {
. . .
}
}
```

Syntax of a monitor.

The monitor construct ensures that only one process at a time can be active within the monitor. Consequently, the programmer does not need to code this synchronization constraint explicitly. However, the monitor construct, as defined so far, is not sufficiently powerful for modeling some synchronization schemes. For this purpose, we need to define additional synchronization mechanisms. These mechanisms are provided by the condition construct. A programmer who needs to write a tailor-made synchronization scheme can define one or more variables of type *condition:*

```
condition x, y;
```

The only operations that can be invoked on a condition variable are wait( ) and signal ( ). The operation

```
x.wait ( );
```

means that the process invoking this operation is suspended until another process invokes

```
x . signal ( ) ;
```

The x. signal ( ) operation resumes exactly one suspended process. If no process is suspended, then the signal( ) operation has no effect; that is, the state of x is the same as if the operation had never been executed. Contrast this operation with the signal ( ) operation associated with semaphores, which always affects the state of the semaphore.

We now illustrate monitor concepts by presenting a deadlock-free solution to the dining-philosophers problem. This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available. To code this solution, we need to distinguish among three states in which we may find a philosopher. For this purpose, we introduce the following data structure:

```
enum{thinking, hungry, eating} state [5] ;
```

Philosopher *i* can set the variable state [i] = eating only if her two

neighborsarenoteating:(state[(i+4) %5] != eating)and(state[(i+1)
%5] ! = eating).

We also need to declare

condition self [5] ;

where philosopher *i* can delay herself when she is hungry but is unable to obtain the chopsticks she needs.
The distribution of the chopsticks is controlled by the monitor dp. Each philosopher, before starting to eat, must invoke the operation pickup ( ). This 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[i] = THINKING;
test ( (i + 4) % 5);
test((i + 1) % 5);

void test (int i) {
if (( s tate [ (i + 4) % 5] ! = EATI NG ) &&
(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;
```

A monitor solution to the dining-philosopher problem.

this solution ensures that no two neighbors are eating simultaneously and that no deadlocks will occur. We note, however, that it is possible for a philosopher to starve to death.

Q4//

1.

a. 250 ⟹ 11111100 (111= 7 page no.) and (11100=28 offset)

250 ⟹ (10 * 32 + 28)=348

b. 78 ⟹ 01001110 (010= 2 page no.) and (01110=14 offset)

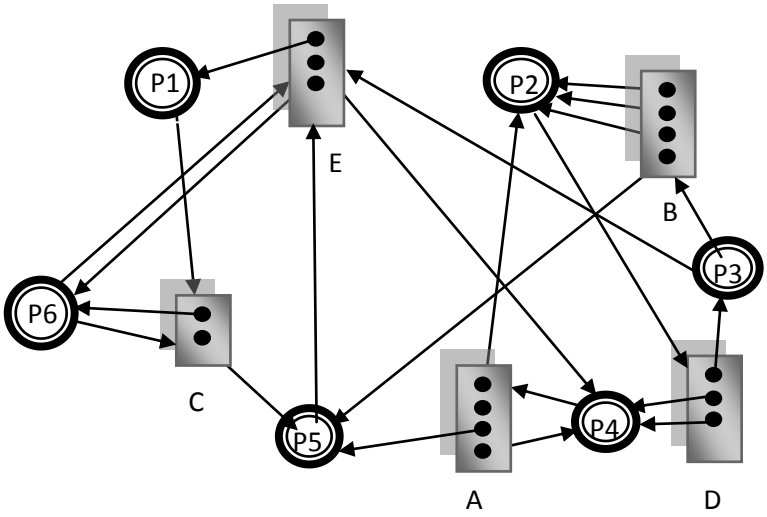
250 ⟹ (6 * 32 + 14)=206

2.

a. <4, 270 > ⟹ 1950+ 270=2220

b. <1, 1100> ⟹ out of range

Q5//



Processes	Allocation					need					Max need					Available				
	A	B	C	D	E	A	B	C	D	E	A	B	C	D	E	A	B	C	D	E
P1	0	0	0	0	1	0	0	1	0	0	0	0	1	0	1	1	0	0	0	0
P2	1	3	0	0	0	0	0	0	1	0	1	3	0	1	0					
P3	0	0	0	1	0	0	1	0	0	1	0	1	0	1	1					
P4	1	0	0	2	1	1	0	0	0	0	2	0	0	2	1					
P5	1	1	1	0	0	0	0	0	0	1	1	1	1	1	0					
P6	0	0	1	0	1	0	0	1	0	1	0	0	2	0	2					

Work=1 0 0 0 0

Finish of P1=false
Need of p1 <= work is false

Finish of P2=false
Need of p2 <= work is false

Finish of P3=false
Need of p3 <= work is false

Finish of P4=false
Need of p4 <= work is true

Work=work + allocation of p4
Work=1 0 0 0 0 + 1 0 0 2 1= 2 0 0 2 1
Finish of P4=true

Finish of P5=false
Need of p5 <= work is true
Work=work + allocation of p5
Work=2 0 0 2 1+ 1 1 1 0 0= 3 1 1 2 1
Finish of P5=true

Finish of P6=false
Need of p6 <= work is true
Work=work + allocation of p6
Work=3 1 1 2 1+ 0 0 1 0 1= 3 1 2 2 2
Finish of P6=true

Finish of P1=false
Need of p1 <= work is true
Work=work + allocation of p1
Work=3 1 2 2 2+ 0 0 0 0 1= 3 1 2 2 3
Finish of P1=true

Finish of P2=false
Need of p2 <= work is true
Work=work + allocation of p2
Work=3 1 2 2 3+ 1 3 0 0 0= 4 4 2 2 3
Finish of P2=true

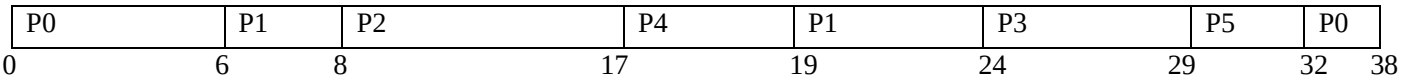
Finish of P3=false
Need of p3 <= work is true
Work=work + allocation of p3
Work=3 1 2 2 3+ 0 0 0 1 0= 4 4 2 3 3
Finish of P3=true

Q6

a)

processes	A.T.	CPU Burst Time	priority
P0	0	12 6	7
P1	6	7 5 0	5
P2	8	9 0	4
P3	14	5 0	6

P4	17	2 0	3
P5	22	3	6



Waiting time (p0)=(o-o)+(32-6)=26
Waiting time (p1)=(6-6)+ (19-8)=11
Waiting time (p2)=(8-8)=0
Waiting time (p3)=(24-14)=10
Waiting time (p4)=17-17=0
Waiting time (p5)=29-22=7

Average waiting time=(26+11+0+10+0+7)/6

b)

1 , 2, 3, 4, 1, 5, 3, 4, 6, 5, 4, 1, 3, 2, 2, 4, 5, 1, 6

1	2	3	4	1	5	3	4	6	5	4	1	3	2	2	4	5	1	6
1	1	1	1		1			1				1	1					6
	2	2	2	ok	5	ok	ok	5	ok	ok	ok	5	5	ok	ok	ok	ok	5
		3	3		3			6				3	2					2
			4		4			4				4	4					4