USN | | | | | | | | | | |

Internal Assessment Test 1 – April 2018
Questions and Solutions

Sub:	Operating Systems					Sub Code:	15CS64	Branch:	ISE	
Date:	17/04/2018	Duration:	90 mins	Max Marks:	50	Sem / Sec:	VI/ A, B		OBE	

Answer any FIVE FULL Questions

		MARKS	CO	RBT
1 (a)	Servers can be designed to limit the number of open connections. For example, a server may wish to have only N socket connections at any point in time. As soon as N connections are made, the server will not accept another incoming connection until an existing connection is released. Explain how semaphores can be used by the server to limit the number of concurrent connections.	[05]	CO2	L3

```
do {

//Declare a semaphore S and initialize to N
semaphore S=N;
...
wait(S);
//allow socket connection
...
signal(S);
// remainder section
}while (TRUE);
```

		MARKS	CO	RBT
1 (b)	Explain an N-process solution to critical section problem which uses testAndSet() atomic instruction. Also explain how the algorithm satisfies all the necessary conditions of the critical section.	[05]	CO3	L2

The common data structures used are

```
boolean waiting[n];
boolean lock;
```

These data structures are initialized to false.

```
do {
      waiting [i] = TRUE;
      key = TRUE;
      while (waiting[i] && key)
            key = TestAndSet(&lock);
      waiting[i] = FALSE;

      // critical section

      j = (i + 1) % n;
      while ((j != i) && ! w a i t i n g [ j ] )
            j = (j + 1) % n;

      if (j == i)
            lock = FALSE;
      else
```

```
                    waiting[j] = FALSE;
                // remainder section
        }while (TRUE);
```

Mutual Exclusion:

Process P can enter its critical section only if either waiting[i] is false or key is false. The value of key can become false only if the TestAndSet() is executed. The first process to execute the TestAndSet () will find key == false; all others must wait.

The variable waiting[i] can become false only if another process leaves its critical section; only one waiting [i] is set to false, maintaining the mutual-exclusion requirement.

Progress:

A process exiting the critical section either sets lock to false or sets waiting[j] to false. Both allow a process that is waiting to enter its critical section to proceed.

Bounded Waiting:
When a process leaves its critical section, it scans the array waiting in the cyclic ordering (i+1, *i+2, ... , n*-1, 0, ..., *i*-1). It designates the first process in this ordering that is in the entry section (waiting [j] =- true) as the next one to enter the critical section. Any process waiting to enter its critical section will thus do so within *n* — 1 turns.

2 (a) Explain the Dining Philosophers solution using monitors. [05] CO2 L2

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];

thinking: State when philosopher does not need chopsticks
hungry: State when philosopher needs chopsticks, but didn't obtain them
eating: State when philosopher needs chopsticks, and has obtained them

Philosopher *i* can set the variable state[i] = eating only if her two neighbours are not eating:
(state[(i+4) °/» 5] != eating) and (state[(i+1) % 5] != eating).
We also need to declare condition self [5] where philosopher i can wait when she is hungry but is unable to obtain the chopsticks she needs.
The following is the solution for each philosopher. Each philosopher i must invoke the operations pickup () and putdownO in the following sequence:

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

The monitor implementation is as follows:

```
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;
            }
      }
```

2 (b) What do you mean by race condition? Explain Readers-Writes problem with semaphore in detail. [05] CO2 L2

When several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **race condition.**
To guard against the race condition above, we need to ensure that only one process at a time can be manipulating the shared variable/data by means of synchronization.

Readers-Writers Problem:
The reader processes share the following data structures:

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

The semaphores mutex and wrt are initialized to 1; readcount is initialized to 0.

The semaphore wrt is common to both reader and writer processes.
The mutex semaphore is used to ensure mutual exclusion when the variable readcount is updated. The readcount variable keeps track of how many processes are currently reading the object.
The semaphore wrt functions as a mutual-exclusion semaphore for the writers. It is also used by the first or last reader that enters or exits the critical section. It is not used by readers who
enter or exit while other readers are in their critical sections.

If a writer is in the critical section and n readers are waiting, then one reader is queued on wrt, and n-1 readers are queued on mutex.

When a writer executes signal (wrt), we may resume the execution of either the waiting readers or a single waiting writer.

```
//Writer Process
do {
wait(wrt);
// writing is performed
signal (wrt) ,-
}while (TRUE);

//Reader Process
do {
wait(mutex);
readcount + + ;
if (readcount == 1)
wait(wrt);
signal(mutex);
// reading is performed
wait (mutex) ,-
readcount--;
if (readcount == 0)
signal(wrt);
signal(mutex);
}while (TRUE);
```

3 (a) Explain how monitors can be used to solve bounded buffer problem. [05] CO2 L3

```
monitor PC{
        //Shared variables
        type buffer[BUFFER_SIZE];
        int count;
        int p_index, c_index;
        condition full, empty; //to track how many full/empty
slots are currently present


        //procedure
        produce_item(type *data){
        if (count == BUFFER_SIZE)
                empty.wait();     // if no empty space then wait
        put_item(data);          // Place the produced item in
buffer
        count = count + 1;       // increment count of full slots
        full.signal();           // signal as we have at least 1
full slot
        }

        //procedure
        consume_item(type *data){
        if (count == 0)
                full.wait();    // wait for full signal
        remove_item(data);     // remove item from buffer
        count = count - 1;     // decrement count of full slots
        empty.signal();  //signal producer as we have at least 1
empty slot
        }
```

```
        //procedure
        put_item(type *data){
                buffer[p_index]=*data;
                index=(p_index+1)%BUFFER_SIZE;
        }

        //procedure
        remove_item(type *data){
                *data = buffer[c_index];
                c_index=(c_index+1)%BUFFER_SIZE;
        }

        //initialization code
        count = 0;
        p_index=0, c_index=0;
  }

  Producer();
  {
    while (TRUE)
   {
      PC.produce_item(&item);             // make a new item

   }
  }

  Consumer();
  {
    while (TRUE)
    {
      PC.consume_item(&item);          // call remove function in
monitor
    }
  }
```

3 (b) Differentiate the following with examples: [05] CO2 L2

(i) Paging and Segmentation
(ii) Logical and Physical addresses
(iii) Internal and External Fragmentation
(iv) First-fit, worst-fit and best-fit algorithms.

Paging	**Segmentation**
A page is of fixed block size.	A segment is of variable size.
Paging may lead to internal fragmentation.	Segmentation may lead to external fragmentation.
The user specified address is divided by CPU into a page number and offset.	The user specifies each address by two quantities a segment number and the offset (Segment limit).
The hardware decides the page size.	The segment size is specified by the user.
Paging involves a page table that contains base address of each	Segmentation involves the segment table that contains segment number

page.	and offset (segment length).

Logical addresses	**Physical addresses**
It is the virtual address generated by CPU	The physical address is a location in a memory unit.
Set of all logical addresses generated by CPU in reference to a program is referred as Logical Address Space.	Set of all physical addresses mapped to the corresponding logical addresses is referred as Physical Address.
The user can view the logical address of a program.	The user can never view physical address of program
The user uses the logical address to access the physical address.	The user can not directly access physical address.
The Logical Address is generated by the CPU	Physical Address is Computed by MMU

Logical addresses	**Physical addresses**
It is the virtual address generated by CPU	The physical address is a location in a memory unit.
Set of all logical addresses generated by CPU in reference to a program is referred as Logical Address Space.	Set of all physical addresses mapped to the corresponding logical addresses is referred as Physical Address.
The user can view the logical address of a program.	The user can never view physical address of program
The user uses the logical address to access the physical address.	The user can not directly access physical address.
The Logical Address is generated by the CPU	Physical Address is Computed by MMU

Internal Fragmentation	**External Fragmentation**
Internal fragmentation is the wasted space within each allocated block because of rounding up from the actual requested allocation to the allocation granularity.	External fragmentation is the various free spaced holes that are generated in either your memory or disk space. External fragmented blocks are available for allocation, but may be too small to be of any use.
It occurs when fixed sized	It occurs when variable size memory

memory blocks are allocated to the processes.	space are allocated to the processes dynamically.
When the memory assigned to the process is slightly larger than the memory requested by the process this creates free space in the allocated block causing internal fragmentation.	When the process is removed from the memory, it creates the free space in the memory causing external fragmentation.
Solution: The memory must be partitioned into variable sized blocks and assign the best fit block to the process.	Solution: Compaction, paging and segmentation.
Example: Consider a multiple-partition allocation scheme with a hole of 18,464 bytes. Suppose that the next process requests 18,462 bytes. If we allocate exactly the requested block, we are left with a hole of 2 bytes.	Example: First-fit and Best-fit strategies. We could have a block of free (or wasted) memory between every two processes. If all these small pieces of memory were in one big free block instead, we might be able to run several more processes.

First Fit	**Best fit**	**Worst fit**
Allocates memory from the first hole it encounters large enough to satisfy the request.	The allocator places a process in the smallest block of unallocated memory in which it will fit.	The memory manager places a process in the largest block of unallocated memory available.
Example: Unallocated blocks of 6KB, 14KB, 19KB, 11KB, and 13KB blocks, suppose a process requests 12KB of memory.		
First fit will allocate 12KB of the 14KB block to the process	Best-fit strategy will allocate 12KB of the 13KB block to the process.	Worst fit will allocate 12KB of the 19KB block to the process, leaving a 7KB block for future use.

4 (a) Answer the following: [05] CO2 L1

(i) What are deadlocks? (ii) What are its characteristics? (iii) What are the necessary conditions for deadlock to occur? (iv) How many of these should occur for a deadlock to happen. (v) What are the different methods to handle deadlocks?

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; and if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested

are held by other waiting processes. This situation is called a deadlock.

Characteristics (or Necessary conditions):

A deadlock situation can arise if the following four conditions hold simultaneously in a system:

1. Mutual exclusion. At least one resource must be held in a non-sharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed 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 other processes.

3. No preemption. Resources cannot be preempted. That is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.

4. Circular wait. A set {P0, P1, ..., Pn} of waiting processes 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 a resource held by Pn, and Pn is waiting for a resource held by P0.

Methods to handle deadlocks: Prevention, Avoidance, Detect and recover

4 (b) Consider the following snapshot of the system: [05] CO2 L3

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

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

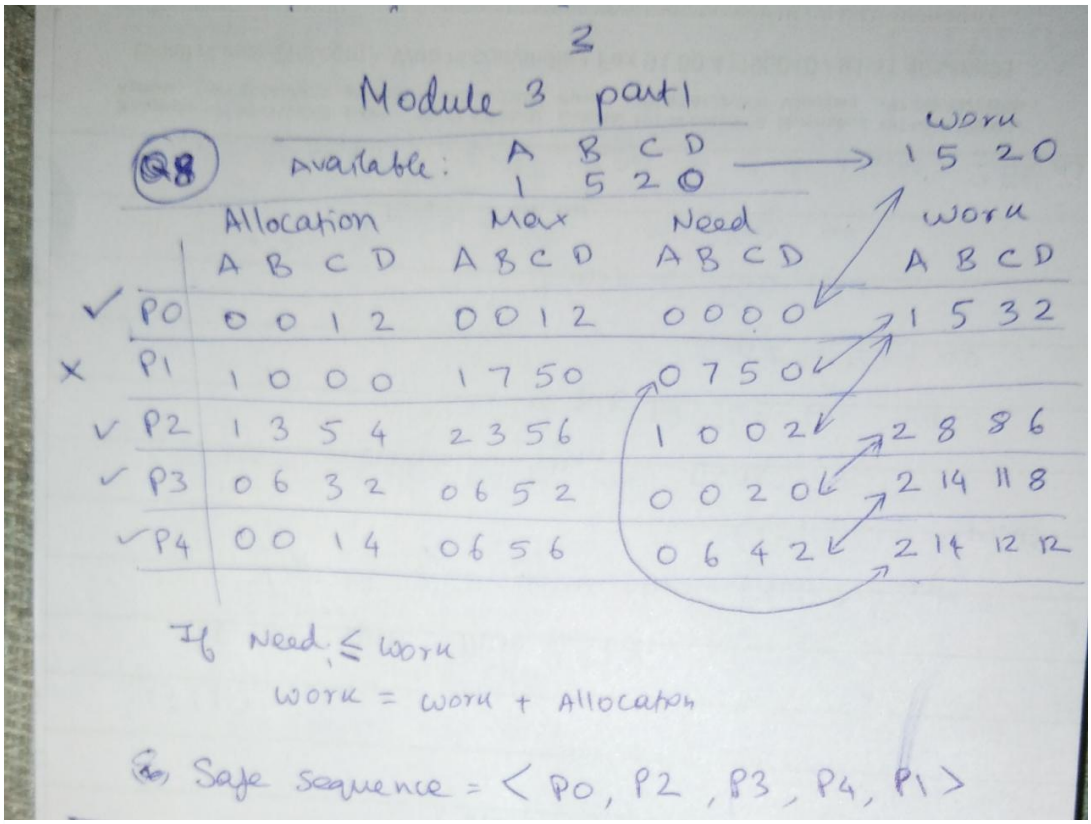
Available			
A	B	C	D
1	5	2	0

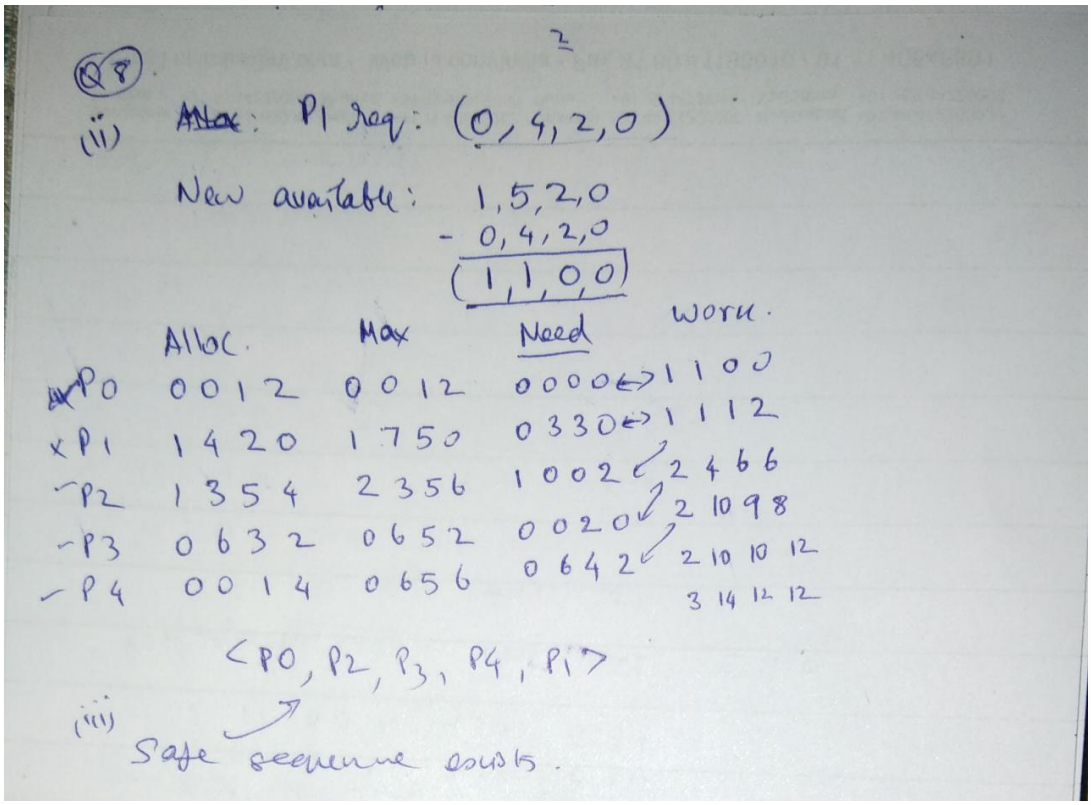
(i) Find out need matrix.

(ii) Is the system in a safe in its current state?

(iii) If a request from P1 arrived for (0,4,2,0), can it be granted immediately?

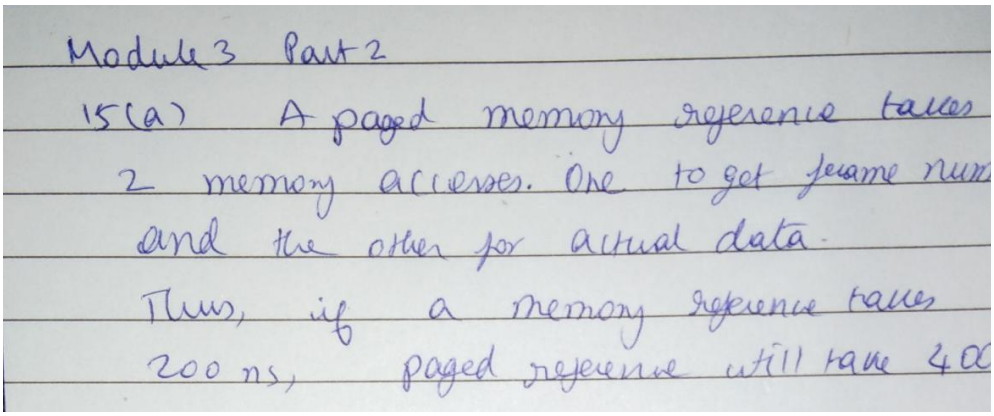
(iv) Is the system in a safe state after the new request?





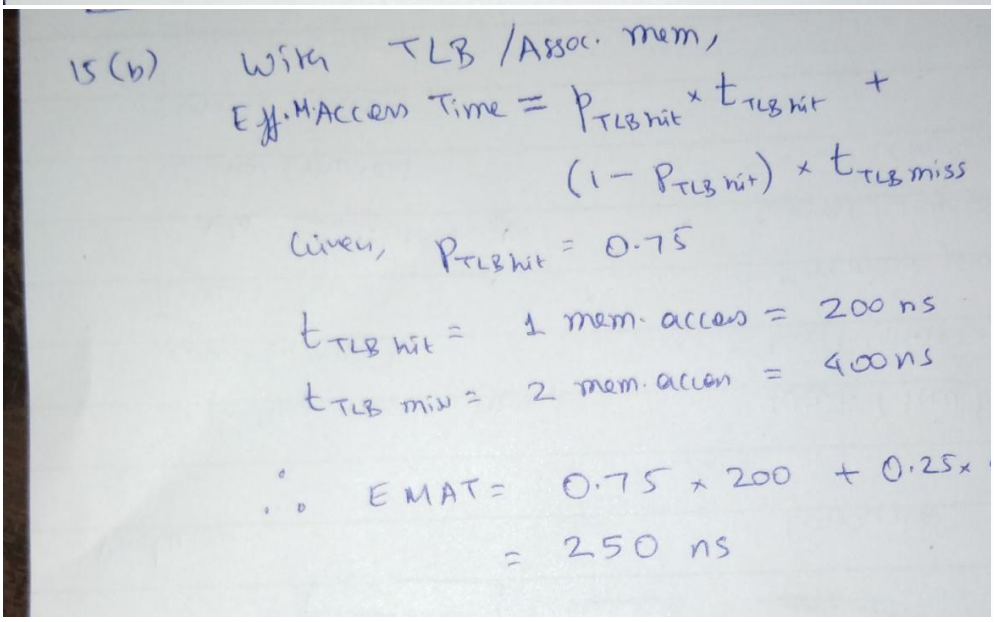
5 (a)	(i) Consider a paging system with page table stored in memory. If memory reference takes 200 ns, how long does a paged memory reference take? (ii) If we add associative register and 75% of all page table references are found in the associative registers, what is the effective memory access time? (Assume that finding a page table entry in the associative memory/register takes zero time, if the	[05]	CO3	L3

entry is found).


Module 3 Part 2

15(a) A paged memory reference takes 2 memory accesses. One to get frame num and the other for actual data.

Thus, if a memory reference takes 200 ns, paged reference will take 400


15(b) With TLB / Assoc. mem,

Eff. MAccess Time = $p_{TLB\,hit} \times t_{TLB\,hit}$ + $(1 - p_{TLB\,hit}) \times t_{TLB\,miss}$

Given, $p_{TLB\,hit} = 0.75$

$t_{TLB\,hit}$ = 1 mem. access = 200 ns

$t_{TLB\,miss}$ = 2 mem. access = 400 ns

∴ EMAT = 0.75 × 200 + 0.25 ×

= 250 ns

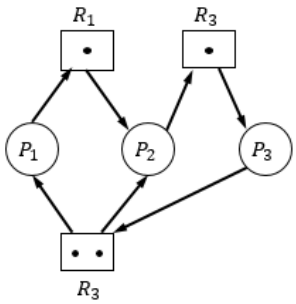
5 (b) Answer the following: [05] CO2 L2

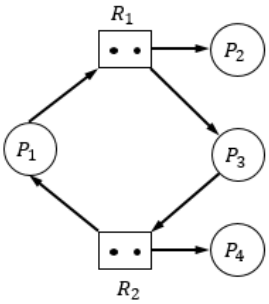
(i) What is Resource Allocation Graph (RAG)?

(ii) Explain resource allocation graph with (a) deadlock (b) cycle but no deadlock.

(iii) Explain how RAG is useful describing deadly embrace by considering your own example

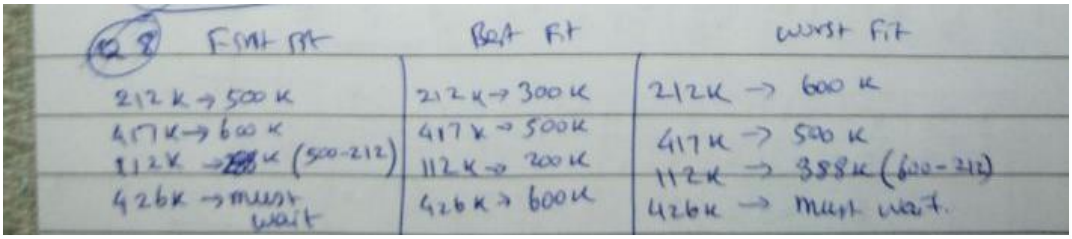
Deadlocks can be described more precisely in terms of a directed graph called a **system resource-allocation** graph. This graph consists of a set of vertices *V* and a set of edges *E*. The set of vertices *V* is partitioned into two different types of nodes: P = {P1, *P2,…, Pn*}, the set consisting of all the active processes in the system, and *R = {R1, R2, … Rm}*, the set consisting of all resource types in the system.





Take example on the left. Here all the resources are part of a cycle. From this, we learn that the system is in a deadlocked state. Take example on the right. Here, even though all the resources are occupied by all the processes, not all resources are part of a cycle. Hence, no deadlock.

6 (a)	Given the memory partitions of 100K, 500K, 200K, 300K, and 600K, apply first fit, worst fit, and best fit algorithms to place 212K, 417K, 112K, 426K.	[05]	CO3	L3



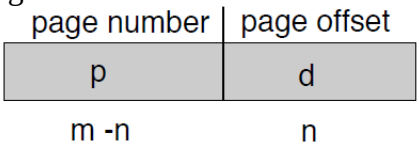
6 (b)	What is the principle behind paging. Explain its operation, clearly indicating how the logical addresses are converted to physical addresses.	[05]	CO3	L2

Paging is a memory-management scheme that permits the physical address space of a process to be non-contiguous. Paging avoids the considerable problem of fitting memory chunks of varying sizes onto the backing store.

The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called **frames** and breaking logical memory into blocks of the same size called **pages.** When a process is to be executed, its pages are loaded into any available memory frames from the backing store. The backing store is divided into fixed-sized blocks that are of the same size as the memory frames.

Every address generated by the CPU is divided into two parts: a **page number (p)** and a **page offset (d).** The page number is used as an index into a **page table.** The page table contains the base address of each page in physical memory. This base address is combined with the page offset to define the physical memory address that is sent to the memory unit.

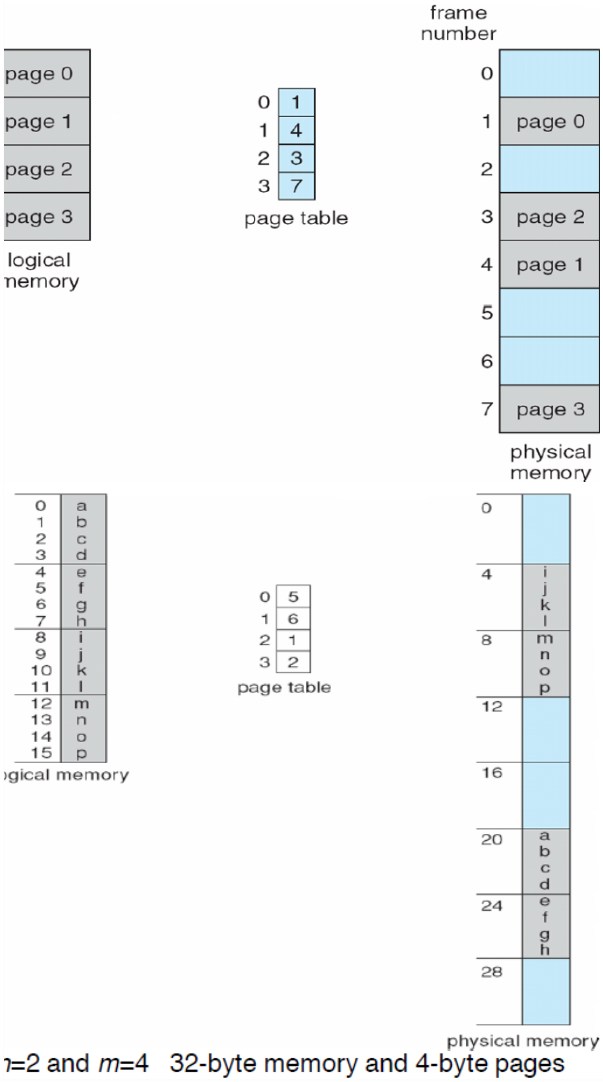
If the size of logical address space is 2^m and a page size is 2^n addressing units (bytes or words), then the high-order m – n bits of a logical address designate the page number, and the n low-order bits designate the page offset. Thus, the logical address is as follows:



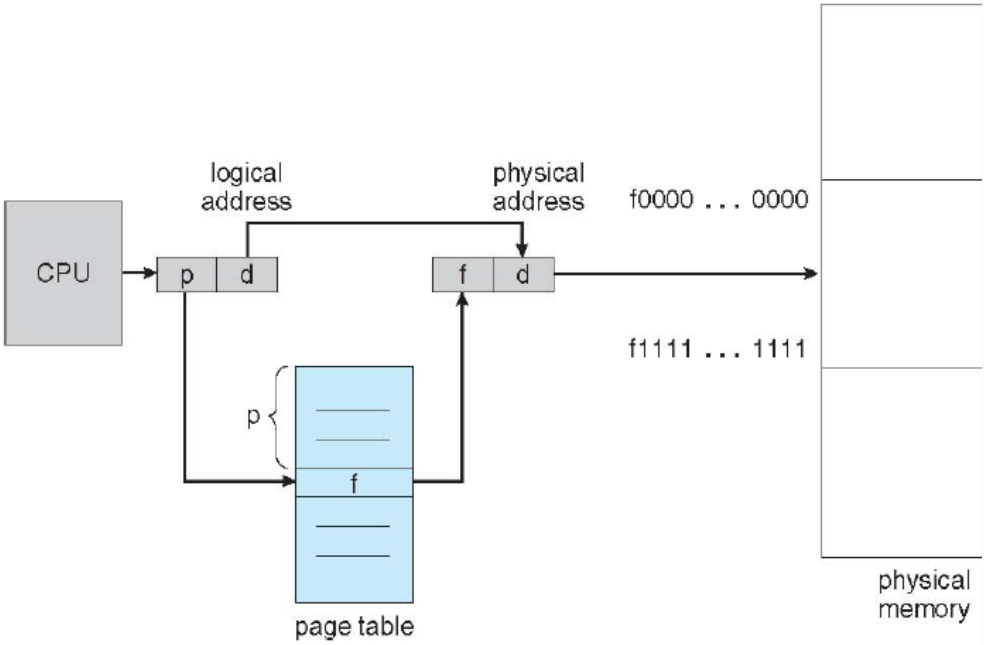
where *p* is an index into the page table and *d* is the displacement within the page.

Logical address to physical address:
As a concrete (although minuscule) example, consider the memory in the Figure below. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages), we show how the user's view of memory can be mapped into physical memory. Logical address 0 is page 0, offset 0. Indexing into the page table, we find that page 0 is in frame 5. Thus, logical address 0

maps to physical address
20 (= (5 x 4) + 0). Logical address 3 (page 0, offset 3) maps to physical address 23 (= (5x4) + 3). Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6. Thus, logical address 4 maps to physical address 24 (= (6x4) + 0). Logical address 13 maps to physical address 9.



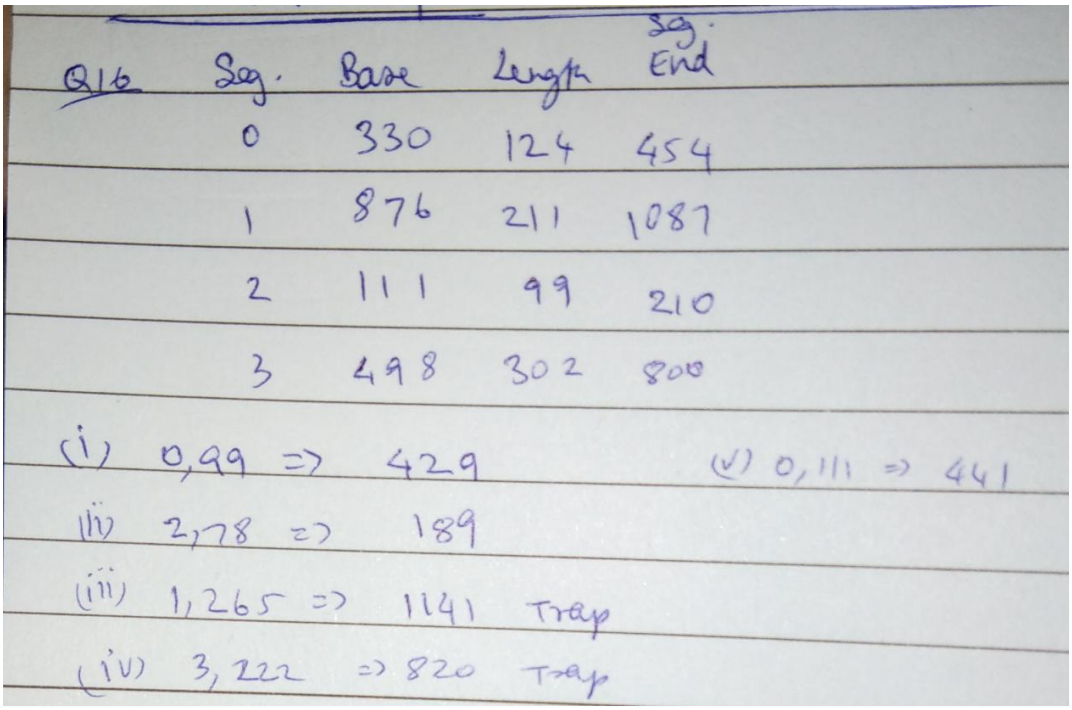
n=2 and *m*=4 32-byte memory and 4-byte pages



7 (a) Consider the following segment table: [05] CO3 L3

Segment	Base	Length
0	330	124
1	876	211
2	111	99
3	498	302

What are the physical addresses for the following logical addresses?
(i) 0,99 (ii) 2,78 (iii) 1,265 (iv) 3,222 (v) 0,111
Indicate which addresses generate segment fault.



7 (b) Answer the following: [05] CO2 L1

(i) What is proportional frame allocation?
(ii) What is Thrashing?
(iii) What causes thrashing?
(iv) How does the system detect thrashing?
(v) How can thrashing be prevented?

(i) In proportional allocation which we allocate available memory to each process according to its size. Let the size of the virtual memory for process p_t be si, and define $S=\Sigma si$
Then, if the total number of available frames is m, we allocate a, frames to process pi, where a, is approximately $a, = Sj/S \times m.$

(ii) If the process does not have the number of frames it needs to support pages in active use, it will quickly page-fault. At this point, it must replace some page. However, since all its pages are in active use, it must replace a page that will be needed again right away.
Consequently, it quickly faults again, and again, and again, replacing pages that it must bring back in immediately.
This high paging activity is called **thrashing.** A process is thrashing if it is spending more time paging than executing.

(iii) Causes of Thrashing:
Consider the following scenario. The operating system sees CPU utilization is too low, it increase the degree of multiprogramming by introducing a new process to the system. A global page-replacement algorithm replaces pages. Now suppose that a process enters a new phase in its execution and needs more frames. It starts faulting and
taking frames away from other processes. These processes need those pages, however, and so they also fault, taking frames from other processes. These faulting processes must use the paging device to swap pages in and out.
As they queue up for the paging device, the ready queue empties. As processes wait for the paging device, CPU utilization decreases.

The CPU scheduler sees the decreasing CPU utilization and *increases* the degree of multiprogramming as a result. The new process introduces in turn causes more page faults and a longer queue for the paging device. As a result, CPU utilization drops even further, and the CPU scheduler tries to increase the degree of multiprogramming even more. Thus Thrashing occurs, and system throughput plunges.

The page fault rate increases tremendously As a result, the effective memory-access time increases. No work is getting done, because the processes are spending all their time paging.

(iv)Detection of Thrashing:
The system can detect thrashing by evaluating the level of CPU utilization as compared to the level of multiprogramming. It can be eliminated by reducing the level of multiprogramming.

(v)Preventing Thrashing:
To prevent thrashing, we must provide a process with as many frames as it

	needs. The working-set strategy is a way to look at how many frames a process is actually using.			
8 (a)	For the following reference string calculate the page faults that occur using FIFO, Optimal, and LRU for 3 and 5 page frames, respectively:1,2,3,4,2,1,5,6,2,1,2,3,7,6,3,2,1,2,3,6.	[05]	CO2	L3

FIFO 3 Frames

1	2	3	4	2	1	5	6	2	1	2	3	7	6	3	2	1	2	3	6
1	1	1	4		4	4	6	6	6	3	3	3		2	2		2	6
	2	2	2		1	1	1	2	2	2	7	7		7	1		1	1
		3	3		3	5	5	5	1	1	1	6		6	6		3	3

PF = 16

Optimal

1	2	3	4	2	1	5	6	2	1	3	7	6	3	2	1	2	3	6
1	1	1	1		1	1			3	3				3	3			3
	2	2	2		2	2			2	7				2	2			2
		3	4		5	6			6	6				6	1			6

PF = 11

LRU

1	2	3	4	2	1	5	6	2	1	3	7	6	3	2	1	2	3	6
1	1	1	4		4	5	5	5	1	1	1	6		6	1			6
	2	2	2		2	2	6	6	6	3	3	3		3	3			3
		3	3		1	1	1	2	2	2	7	7		2	2			2

PF = 15

5 Frames

FIFO

1	2	3	4	2	1	5	6	2	1	3	7	6	3	2	1	2	3	6
1	1	1	1			1	6		6		6		6	6				
	2	2	2			2	2		1		1		1	1				
		3	3			3	3		3		7		7	7				
			4			4	4		4		4		3	3				
						5	5		5		5		5	2				

PF = 10

Optimal

1	2	3	4	2	1	5	6	2	1	3	7	6	3	2	1	2	3	6
1	1	1	1			1	1				1							

Optimal.

1	2	3	4	2	1	5	6	2	1	3	7	6	3	2	1	2	3	6
1	1	1	1			1	1				1							
	x2	2	2			2	2				2							
		x3	3			3	3				3							
			x4			4	x6				6							
						x5	5.				x7							

PF = 7

LRU

1	2	3	4	2	1	5	6	2	1	3	7	6	3	2	1	2	3	6
x1	1	1	1			1	1			1	1							
	x2	2	2			2	2			2	2							
		x3	3			3	x6			6	6							
			x4			4	4			x3	3							
						x5	5			5	x7							

PF = 8

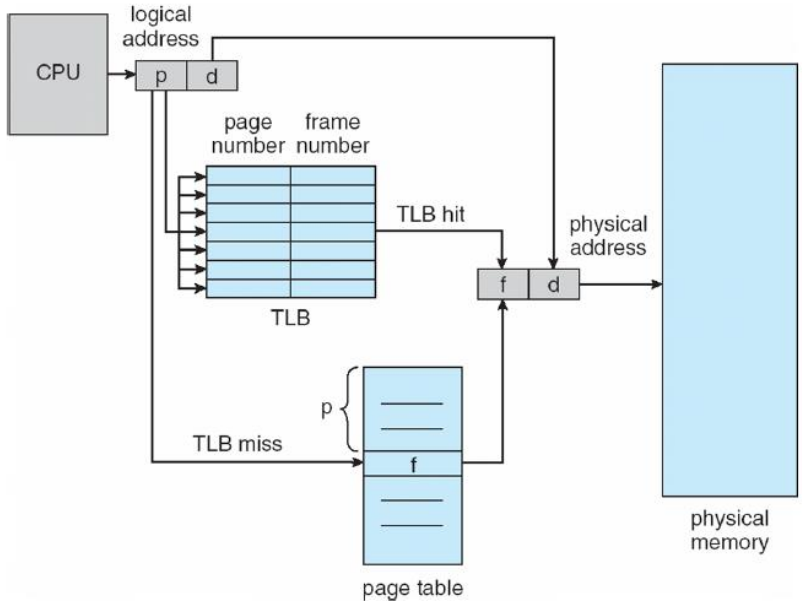
8 (b)	What are Translation Lookaside Buffers (TLBs)? Explain TLB in detail with a simple paging system with a neat diagram.	[05]	CO3	L1

Translation look-aside buffers (TLBs) are a special, small, fast lookup hardware cache. The TLB is associative, high-speed memory. Each entry in the TLB consists of two parts: a key (or tag) and a value.

When the associative memory is presented with an item, the item is compared with all keys simultaneously. If the item is found, the corresponding value field is returned. The search is fast; the hardware, however, is expensive.
Typically, the number of entries in a TLB is small, often numbering between 64 and 1,024.

The TLB is used with page tables in the following way. The TLB contains only a few of the page-table entries. When a logical address is generated by the CPU, its page number is presented to the TLB. If the page number is found, its frame number is immediately available and is used to access memory. The whole task may take less than 10 percent longer than it would if an unmapped memory reference were used.

If the page number is not in the TLB (known as a TLB miss), a memory reference to the page table must be made. When the frame number is obtained, we can use it to access memory. In addition, we add the page number and frame number to the TLB, so that they will be found quickly on the next reference. If the TLB is already full of entries, the operating system must select one for replacement. Replacement policies range from least recently used (LRU) to random. Furthermore, some TLBs allow entries to be wired down, meaning that they cannot be removed from the TLB. Typically, TLB entries for kernel code are wired down.



END