INDIAN INSTITUTE OF INFORMATION TECHNOLOGY, LUCKNOW
End-Semester Examination, September 2017

Date of Examination : 24.11.2017 (1st Session)

Program Code & Semester : B.Tech.(IT) - 3rd Semester

Paper Title: Operating System

Paper Code: IOPS332C

Paper Setter: Dr. Bibhas Ghoshal
(Sec. A - BG, Sec. B – JS)

Max Marks: 75
Duration: 3 hours

Note: There are seven **questions** in this question paper. **Answer All questions**. The marks for each question has been provided alongside.

Q1. State True/False **with justification.** **[15 marks]**

(A) Non-contiguous memory allocation schemes are advantageous, because, unlike the contiguous memory allocation schemes, they do not suffer from Internal Fragmentation or External Fragmentation Problems.

Solution: FALSE.

There are basically two types of non-contiguous memory allocation schemes, viz., Paging and Segmentations. Any fixed-size partition based memory allocation scheme suffers from internal fragmentation. This problem arises when the size of a process is not a whole multiple of the partition size. In such case, for the last portion of the process, we need to allocate an entire partition whose part is only used by the process. Paging is also a fixed-size partition scheme, hence, suffers from internal fragmentation problem. External fragmentation exists when there is enough total memory space available to accommodate a process or part of a process but the available spaces are not contiguous; storage is fragmented into a large number of small blocks of free memory. Segmentation may cause external fragmentation, when all blocks of free memory are too small to accommodate a segment but the sum of the free spaces is larger than the segment size.

Hence, paging suffers from internal fragmentation and segmentation suffers from external fragmentation problem. However, paging does not suffer from external fragmentation and segmentation does not suffer from internal fragmentation problem.

(B) For sequential files, linked allocation is preferred over contiguous allocation.

Solution : FALSE

For sequential access, the address of the kth block of the file which starts at block b can easily be obtained as (b+k). Using contoguous allocation make the access extremely fast since the number of seeks are minimal because of contiguous allocation of file blocks.

(C) In UNIX, the open system call returns pointer to the entry in the open file table.

Solution : TRUE

The open system call returns an integer called the file descriptor. File descriptor is the index from a table called the Process File Descriptor table which is local to every process and contains information like the identifiers of the files opened by the process. Whenever, a process creates a file, it gets an index from this table. Other system calls use this file descriptor for reading, writing, seeking, duplicating, closing the file etc. and the corresponding entry in the user file descriptor table points to a unique entry in the global file table even though a file(/var/file1) is opened more then once.

(D) If the shared resources are numbered 1 through N and a process can only ask for resources that are numbered higher than that of any resource that it currently holds, then deadlock can never happen.

Solution : <u>TRUE</u>
The condition imposed on allocation prevents the circular wait condition for deadlock.

(E) Consider the following program fragment: s1, s2, s3 and s4 are semaphores. If two threads run this fragment of code simultaneously, there can be a deadlock.

```
if(a > 0)
        P(s1);
else
        P(s2);
b++;
P(s3);
if(b < 0 && a <= 0)
        P(s1);
else if(b >= 0 && a > 0)
        P(s2);
else
        P(s4);
a++;
V(s4);
V(s3);
V(s2);
V(s1);
```

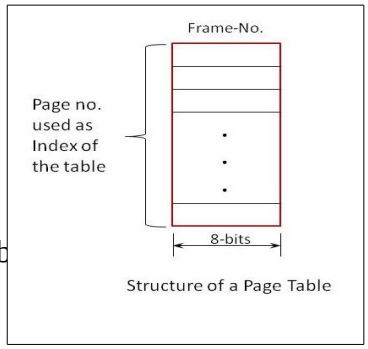
Solution : <u>TRUE</u>
Yes, there can be a deadlock. Consider the scenario where thread 1 starts by locking semaphore s1 then gets switched out while thread 2 runs. Thread 2 may start running with variable a set to 0 (remember, because the variables are automatic, each thread has its own independent set). Thread 2 will acquire semaphore s2, then proceeds to acquire s3. Then, if (b<0 && a <=0), it will try to acquire s1. This forces thread 2 to wait for thread 1 which holds this semaphore. Now, thread 1 may proceed but will block when it tries to acquire semaphore s3. This forms a cyclical wait, and a deadlock occurs.

Q2. (a) A small computing device has 512KB RAM. The tiny O/S installed in this device uses paging mechanism with a page-size of 2 KB. What would be the minimum size of the page table for a process P whose size is 200 KB? (Tiny O/S does not use demand paging.) [**2 marks]**

Solution.

Given:

Size of the Main Memory (MM) = 512 KB
Page Size = 2 KB
Process Size = 200 KB



Hence, Frame Size = Page Size = 2 KB.
So, total no. of frames in MM = 512KB / 2KB = 256
Hence, min. Size of an Entry (row) of the page table
= min. Size of the 'Frame-No.' field in the page tab
= $\log_2$ (256)
= 8 bits = 1 byte
And, total no. of entries in the page table
= total no. of pages in process P
(since no demand paging, entries for all pages must be there in the page table)
= Process Size / Page Size
= 200KB / 2KB = 100
So, min. Size of the Page Table
= min. size of each entry in the page table * total no. of entries
= 1 byte * 100 = 100 bytes (Ans.)

(b) Suppose the CPU generates the following sequence of 15 logical addresses while executing a process P.

[1010, 0413, 0691, 0010, 1020, 0575, 1024, 0500, 0780, 0220, 0340, 0381, 0268, 0950, 0321]

i. Calculate the number of page faults using 'Least Frequently Used' page replacement strategy, if page-size = 200 bytes and no. of frames allocated to process P is 4 (Assume that word-size = 1 byte). [**3 marks]**

i. Solution:

Given:

Page Size = 200 bytes
No. of Frames allocated to process P = 4
Word size = 1 byte
CPU generated string of logical addresses:
[1010, 0413, 0691, 0010, 1020, 0575, 1024, 0500, 0780, 0220, 0340, 0381, 0268, 0950, 0321]

Since, each data word is 1 byte long and page-size is 200 bytes, logical address locations 0 to 199 belong to Page Number 0. Similarly, the mapping for all logical addresses to their corresponding page no.s can be shown using the following table:-

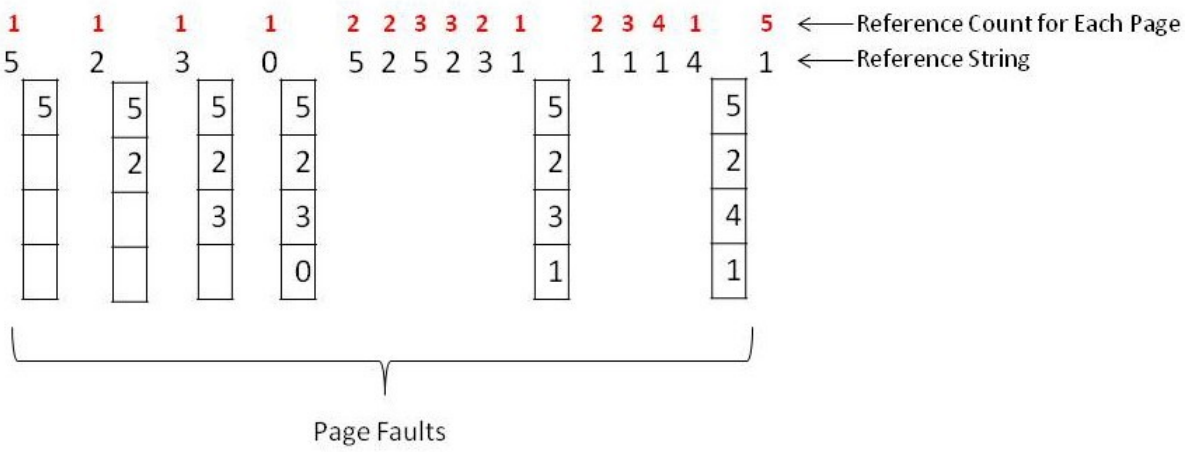
Address Locations	Page No.
0 – 199	0
200 – 399	1
400 – 599	2
600 – 799	3
800 – 999	4
1000 - 1199	5

Hence, the CPU generated string of page no.s (Reference String) obtained from the given string of logical addresses is as shown below:-

[5, 2, 3, 0, 5, 2, 5, 2, 3, 1, 1, 1, 1, 4, 1]

(Note: *In this problem, we should not merge the same consecutive page no.s, because in Least Frequently used page replacement approach, the 'Count' of references to each page is important* !)

Now, the status of the 4 Frames after Every Page Reference is shown below (note that, whenever a page-replacement is required, we are replacing the page with the least reference-count value):-



Hence, number of page faults = 6

ii. Does the above algorithm suffer from Belady's Anomaly? Explain. **[2 marks]**

Solution: Any 'Stack Algorithm' does not suffer from Belady's Anomaly. A stack algorithm is one, in which, set of pages of a process in main memory with *n* allocated frames is always a subset of pages in memory with (*n*+1) allocated frames to the process. Least Frequently Used page replacement algorithm is also a stack algo., because with *n* allocated frames, there will be *n* most frequently used frames of a process, which is certainly a subset of (*n*+1) most frequently used frames that would be in memory with (*n*+1) frames allocated to it.

(c) State 3 main advantages of Virtual Memory. [**3 marks]**

3 main advantages of Virtual Memory:-

- A program needs not be constrained by the amount of physical memory that is available. So, programmers would be able to write programs for an extremely large virtual address space, simplifying the programming task.
- Because each user program could take less physical memory, more programs could be run at the same time, with a corresponding increase in CPU utilization and throughput but with no increase in response time or turnaround time.
- Less I/O would be needed to load or swap user programs into memory, so each user program would run faster.

Q3. (a) If an O/S uses Global Page replacement approach, then a Compiler or Loader running on the system must use relocatable address binding. Why? [**2 marks]**

Solution: Global replacement allows a process to select a replacement frame from the set of all frames, even if that frame is currently allocated to some other process; that is, one process can take a frame from the physical address space of another. In other words, physical address space of a process changes during its execution.

If a compiler or loader needs to use absolute address binding, then absolute physical addresses must be decided by the O/S during compile time or load time of a program. These physical addresses cannot be changed during the execution time of the corresponding process. This means the physical address space of the process should remain the same during it's execution. However, global page replacement cannot ensure this condition. Hence, relocatable address binding must be used by compiler/loader in a system which supports global page replacement because; reloadable addresses are mapped to absolute physical addresses only at the run time.

(b) LRU Approximation Page Replacement algorithm can be considered as a "Less" recently used page replacement strategy rather than "Least" recently used page replacement strategy. Explain. [**2 marks]**

Solution: LRU page replacement is costly to implement without special hardware support because, in order to get the 'least' recently used page number, the O/S needs to keep track of the last references to all pages of a process which, in turn, requires an extra access to main memory for every instruction cycle.

LRU approximation algorithms reduce this cost by eliminating the need for a total ordering of all the page no.s (which is needed for determining the 'Least' recently used page). Rather, these algorithms only maintain a partial ordering of the page no.s according to their most recent time of access/use. In reference bit algo., the number of classes of pages is 2, i.e., pages with reference bit = 0 and pages with reference bit =1. Same is for second chance algo. For additional reference bit algo., the number of classes is 256 (comparison value is 8-bits long, so 2^8), and for enhanced second chance algo., it is 4 (comparison value is 2-bits long, so 2^2). Pages belonging to a class with a smaller value is 'Less' recently used than the pages belonging to other classes with larger values, and is thus chosen for replacement. Hence, the algorithms can be considered as Less Recently used page replacement approaches, not Least recently used.

(c) In which of the following two cases the TLB search time would be minimum? [**1 mark]**

Case A: The entries in the TLB are kept in ascending order of their values.
Case B: The entries in the TLB are kept in descending order of their values.

Solution: In both cases, the search time would be the same. TLB is a device which implements parallel search in hardware. The submitted page no. is compared simultaneously with all the page no. values in the TLB and corresponding frame no. is returned if a match is found.

(d) Consider the following page replacement algorithms. Rank these algorithms on a five-point scale from "bad" to "perfect" according to their page-fault rates. [**2 marks]**

a. LRU Page Replacement
b. FIFO Page Replacement
c. Optimal Page Replacement
d. Second Chance Page Replacement
e. Enhanced Second Chance Page Replacement

Solution:

a. LRU Page Replacement - 2
b. FIFO Page Replacement - 5
c. Optimal Page Replacement - 1
d. Second Chance Page Replacement - 4
e. Enhanced Second Chance Page Replacement – 3

Explanation: Optimal Page Replacement is best because its page fault count is minimum. Between, LRU and FIFO, LRU is better since, being a stack algorithm, it does not suffer from Belady's anomaly. Hence, the algorithms that try to approximate the characteristic of LRU (d. and e.) are also better than FIFO. And, certainly e. is better than d., since it is an improvement over d. (carries more ordering information).

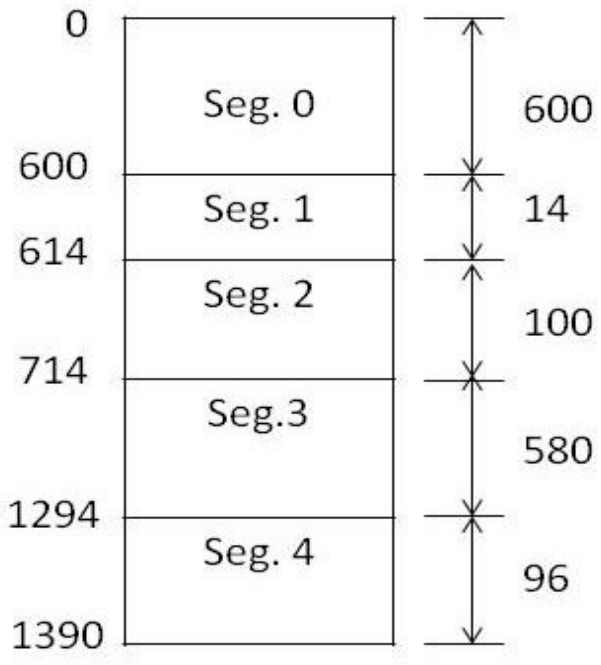
(e) Consider the following Segment Table:

Base	Limit
219	600
2300	14
90	100
1327	580
1952	96

i. What are the physical addresses corresponding to the following two logical addresses: 1000, 2400

ii. What is the logical address for the physical address: 1375 [**3 marks]**

Solution : The logical address space of the process is drawn below according to the segment sizes of the 5 segments given in the segment table.



i.

So, logical address1000 is in Seg. No. 3
Offset of the address in its segment = 1000 – 714 (base logical addr. Of Seg. 3)
= 286
Hence, the corresponding physical address = 1327 (base physical addr. Of Seg. 3) + 286
= 1613

2400 is beyond the logical address space of the process. Hence, it will generate a TRAP: Addressing Error. (Ans.)

ii.

Physical address 1375 belongs to the space allocated to Seg. 3 in main memory. [Because physical address space for Seg. 3 is 1327 to (1327+580) = 1907. // See the Segment table give in the question paper.]

So, offset of this physical address = 1375 – 1327 (base physical addr. Of Seg. 3) = 48
Hence, the corresponding logical address = 714 (base logical addr. Of Seg. 3) + 48
= 762

Q4. (a) What is an atomic operation? **[1 mark]**

Solution : Computer systems provide indivisible instructions called **atomic instructions** to support process synchronization.
Atomic instructions are special hardware instructions that perform an operation on one or more memory locations atomically (from start to finish without being interrupted) . An atomic operation either succeeds or fails in its entirety, regardless of what instructions are being executed by other processors.

Atomic instructions can be used as a form of synchronization since they can be used to change shared data without the need to acquire and release a lock, they can allow greater levels of parallelism. However, because they are low level, and can make only small updates to a data structure, using them to implement parallel data structures is a difficult task.

Examples of atomic instructions:

atomic increment
atomic exchange register and memory location
compare and swap

(b) How can the code segment *"if(i < y) cout << foobar(i,y);"* be made to behave as if it were atomic? Answer the question by recoding the segment. **[3 marks]**

Solution :

```
pthread_mutex_t m=1;
pthread_mutex_lock( &m );
 if( i < y )
        cout << foobar( i, y );
 pthread_mutex_unlock( &m );
```

(c) You have been hired by Mother Nature to help her out with the chemical reaction to form water, which she does not seem to be able to get right due to synchronization problems. The trick is to get two H atoms and one O atom all together at the same time. The atoms are threads. Each H atom invokes a procedure hReady when it is ready to react, and each O atom invokes a procedure oReady when it is ready. For this problem, you are to write the code for hReady and oReady. The procedures must delay until there are at least two H atoms and one O atom present, and then one of the threads must call the procedure makeWater (which just prints out a debug message that water was made). After the makeWater call, two instances of hReady and one instance of oReady should return. Your solution should avoid starvation and busy-waiting. You may assume that the semaphore implementation enforces FIFO order for wakeups—the thread waiting longest in P() is always the next thread woken up by a call to V().

Provide the pseudo implementation of hReady and oReady using semaphores. **[8 marks]**

Solution :

```
Semaphore mutex = 1;
Semaphore h_wait = 0;
Semaphore o_wait = 0;
int count = 0;
hReady() {
P(mutex);
count++;
if(count %2 == 1) {
V(mutex);
P(h_wait);
} else {
V(o_wait);
P(h_wait);
V(mutex);
}
return;
}

oReady()
{
P(o_wait);
V(h_wait);
V(h_wait);
makeWater();
return;
}
```

Alternative equivalent solution:

```
Semaphore mutex = 1;
Semaphore h_wait = 0;
Semaphore o_wait = 0;
hReady() {
V(o_wait)
P(h_wait)
return;
}
oReady()
{
P(mutex)
P(o_wait);
P(o_wait);
V(h_wait);
V(h_wait);
makeWater();
V(mutex)
return;
}
```

Q5. Given a 'claim matrix', an 'allocation matrix' and a 'resource vector' for a set of processes:
(a) Is there a safe state? If yes, give the order in which the processes should run to completion.

(b) If Process P3 requests 1 unit of R3, should we grant this request?
If yes, give a <sequence> in which all processes can run to completion.

Allocation Matrix

	R1	R2	R3
P1	1	0	0
P2	5	1	1
P3	2	1	1
P4	0	0	2

Claim Matrix

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Resource Vector

R1	R2	R3
9	3	6

[12 marks]

Solution : (a)

Need Matrix

	R1	R2	R3
P1	2	2	2
P2	1	0	2
P3	1	0	3
P4	4	2	0

Available Vector

R1	R2	R3
1	1	2

Yes, this is a safe state. Processes can run to completion in this order:
<P2, P1, P3, P4>

Solution (b) :
No, we should not grant this request; because, if we do, the system goes into an UNSAFE STATE.
If we grant P3's request then the Need Matrix and available vector become:

Need Matrix

	R1	R2	R3
P1	2	2	2
P2	1	0	2
P3	1	0	2
P4	4	2	0

Available Vector

R1	R2	R3
1	1	1

No process is guaranteed to finish. Therefore, the system runs into an
UNSAFE STATE and there is potential for deadlock.
Resource Manager should not grant P3's request!

Q6. (a) We have a disc that has 10 tracks per platter with 8 sectors. The drive supports 10 writable platters on a common spindle. Each sector stores four 512 byte blocks. There is a read write head for every platter. The heads can be switched in 1ms. The track traversal is sequential and is at the rate of 10ms per track.

i. Show how a 7.5 KB file could be stored ideally.

ii. What is the time of retrieval for the file, assuming that (i) the head needs to be switched, (ii) the track needs to be traversed half way to retrieve the first sector of this file.

iii. Suppose we decide to fill up this disk with 10 files of 1.1 KB, 20 files of 2.2KB, 30 files of 3.3 KB, and so on. Finally, how many files will be there in the disk? Also, what will be the internal and external fragmentations when the disk is filled up? **[6 marks]**

Solution:
There are 8 sectors/track and There are 10 tracks/ platter
Tswitch = 1ms; Ttraverse = 10ms/track
1 sectors = 4 blocks of 512 bytes; Total size of disk = 1638.4 KB

i. File of size 7.5 KB. Ideally it can be stored on 4 blocks of 512 bytes. That is on 4 consecutive sectors.

ii. Time to access this file when it is stored on separate platters in 4 sectors will be :
(10/8) *4 + 3 = 8 ms. Time to access this file when it is stored sequentially on a single platter will be :
(1.25)/2 + 3*(1.25) = 4.375 ms

iii. 10 files of 1.1.KB = 11KB; 20 files of 2.2 KB = 44 KB; 30 files of 3.3 KB = 99 KB
40 files of 4.4 KB = 176 KB; 50 files of 5.5 KB = 275 KB ; 60 files of 6.6 KB =396 KB
70 files of 7.7 KB = 539 KB

Total number of files = 280
Total file size is 1540 KB
Total space on disk = 1638.4 KB

External fragmentation = 1638.4 – 1540 = 98.4 KB; I
nternal fragmentation = For 1.1 KB files = (512*3) – 1.1 = 436 bytes,
so total fragmentation for 10 files will be 4360 bytes.

Q6. (b) Assuming the style of i-node storage where the i-node is stored at the first block of a file, how many disk operations are needed to fetch the i-node for the file /a/b/c/d/e (where a/b/c/d is the absolute directory path, and 'e' is the filename)? Assume that the i-node for the root directory is in the memory, but nothing else along the path is in the memory. Also assume that each directory fits in one disk block. **[4 marks]**

Solution :

Let's assume that for any directory, it takes one disk operation to retrieve the i-node, and one more operation for the directory content. Then the following disk reads are needed:
1. dir content for /
2. i- node for a
3. dir content for a
4. i-node for b
5. dir content for b
6. i-node for c
7. dir content for c
8. i-node for d
9. dir content for d
10. i-node for e

Q7. Consider an indexed file allocation using index nodes (inodes). An inode contains among other things, 7 indexes, one indirect index, one double index, and one triple index.

i. What usually is stored in the inode in addition to the indexes?

ii. What is the disadvantage of storing the file name in the inode? Where should the file name be stored?

iii. If the disk sector is 512 bytes, what is the maximum file size in this allocation scheme?

iv. Suppose we would like to enhance the file system by supporting versioning. That is, when a file is updated, the system creates new version leaving the previous one intact. How would you modify the inode allocation scheme to support versioning? You answer should consider how a new version is created and deleted. In a file system supporting versioning, would you put information about the version number in the inode, or in the directory tree? Justify your answer. **[6 marks]**

Solution:

i. An inode usually stores indexes and:
the file size in bytes,
special flags to indicate if the file is of a special kind (directory, symbolic links)
time stamps (creation date, modification date, last read date),
a reference count of how many names refer to that file from the name space,
owner identification (e.g. in UNIX, user id and group id), and
security crendential (who should be able to read the file)

ii. Storing the file name in the inode limits the flexibility of the file system and precludes the use of hard links. Also, since it is desirable to have relatively long file names, it would be cumbersome to store variable size character arrays in the inode structure (or wasteful if a certain maximum is defined).

iii. We must first determine the size of an index. For a 2-byte index, we can have 65536 disk blocks, i.e. 512 * 65536 = 32MB. But a triple index structure can express more disk blocks. Therefore, we go to a 4-byte indexing scheme (3-byte indexing scheme are not attractive and are not sufficient). A 4-byte indexing scheme therefore gives a maximum file size of 7*512 + 128*512 + 128*128*512 + 128*128*128*512 = 108219952 or about 1Gbytes

We augment the inode structure such that different versions of the file share the common blocks. When a new version is created, we copy the information from the older version into the new one. Also, we will need to add a reference count to every disk block, and set that reference count to 1 when a block is allocated to a single file. The reference count is incremented whenever a block is shared among different versions. Then, when a version is modified, we perform a copy-on-write like policy and decrement the reference count of the disk blocks and actually copy the modified disk blocks and make them only accessible to the new version.

It is better put in the directory space in order to make it easier for the user to relate the different versions of the same file.