INDIAN INSTITUTE OF INFORMATION TECHNOLOGY, ALLAHABAD

Mid-Semester Examination, September 2016

Date of Examination (Meeting) : 30.09.2016 (1st Meeting)

Program Code & Semester : B.Tech.(IT)/ B.Tech.(ECE)/ Dual Degree - 3rd Semester

Paper Title: Operating System
Paper Code: IOPS332C

Paper Setter: Dr. Bibhas Ghoshal
(Sec. A - BG, Sec. B – JS/AD, Sec. C - BG, RGIIT - SA)

Max Marks: 30 **Duration: 2 hours**

Note: There are **six questions** in this question paper. **Answer All questions**. The subquestions of Question 1 (multiple choice questions) are worth 0.5 mark each with a *penalty of 0.25 marks for incorrect answer*, the marks for each sub question in Question 2 (multiple choice questions) are worth 2 mark each. However, for each sub question in Question 2 you need to **justify your answer**, failing which your answer will not be considered. The marks for rest of the questions have been provided alongside each question.

Q1. Choose the best option from the following (**no justification needed**). [5 marks]

(A). Which of the following is not a function of the operating system?
(i) Generate interrupts
(ii) making the computer system convenient to use
(iii) manage i/o devices
(iv) protecting user programs from one another

Ans : (i)

(B). Choose the best definition of process :
(i) An executable program
(ii) program code +contents of processor's registers+stack
(iii) program code +contents of processor's registers+stack+PCB+ready queue
(iv) program code +contents of processor's registers

Ans : (ii)

(C). What is the difference between multithreading and multiprocessing?
(i) multiple threads share code and data sections
(ii) only processes require context switching
(iii) only threads can support parallelism

Ans : (i)

(D). What is the advantage of multiprogramming:
(i) efficient use of CPU
(ii) fast response
(iii) efficent use of disk

Ans : (i)

(E). Which component ensures that process can execute within its own address space?
(i) I/O device
(ii) memory addressing hardware
(ii) stack pointers

Ans : (ii)

(F). Which of the following instructions should be priviledged?
(i) Read data from disk
(ii) set priority of process
(iii) read the clock

Ans : (ii)

(G). Threads of the same task....I. Share the same address space II. Reduce context switching overhead III. Are protected from each other the same way as heavy weight processes. Which of the following options are correct ?
(i) Only Statement I about threads is true
(ii) Statements I and II about threads are both true
(iii) Statements I, II and III about threads are all true

Ans : (ii)

(H). Assume a single thread kernel OS running multiple user threads. If one user thread requests a read system call then....
(i) other system threads continue to run
(ii) some user threads run and some are blocked
(iii) all user threads are blocked

Ans : (i)

(I). Which of the following statements about the process state transitions are FALSE:

(i) When a running process receives interrupt, it goes to ready state.
(ii) Upon finishing, the running process exits and goes to terminated state.
(iii) A ready state goes to running state when the scheduler schedules it.
(iv) An I/O process on completion of I/O request goes back to running state.

Ans : (iv)

(J) Pipe is used for interprocess communication. Which statements about PIPE are true?

(i) we may read and write from pipe at the same time
(ii) pipe is used for unidirectional flow of data
(iii) The pipe() system call requires an array of two integers as parameter.
(iv) All of these

Ans : (iv)

2. Choose the best option with justification in each case. Without justification you would not be awarded any marks for your answer. [10 marks]

(A). Consider the following code fragment:

```
if (fork() == 0)
        { a = a + 5; printf("%d,%d\n", a, &a); }
else { a = a – 5; printf("%d,%d\n", a, &a); }
```

Let u, v be the values printed by the parent process, and x, y be the values printed by the child process. Which one of the following is TRUE? Why?

(i) u = x + 10 and v = y
(ii) u = x + 10 and v != y
(iii) u + 10 = x and v = y
(iv) u + 10 = x and v != y

Ans: option (iii)

Explanation : Child process will execute the if part and parent process will execute the else part. Assume that the initial value of a = 6. Then the value of a printed by the child process will be 11, and the value of a printed by the parent process in 1. Therefore u+10=x.

Now the second part. The answer is v = y.

We know that, the fork operation creates a separate address space for the child. But the child process has an exact copy of all the memory segments of the parent process. Hence the virtual addresses and the mapping (initially) will be the same for both parent process as well as child process. Note that, the virtual address is same but virtual addresses exist in different processes' virtual address spaces. And when we print &a, its actually printing the virtual address. Hence the answer is ***v = y****. The virtual address of parent & child may or may not be pointing to different physical address as explained below.*

Additional reading :

{ When a fork() system call is issued, a copy of all the pages corresponding to the parent process is created, loaded into a separate memory location by the OS for the child process. But this is not needed in certain cases. When the child is needed just to execute a command for the parent process, there is no need for copying the parent process' pages, since exec replaces the address space of the process which invoked it with the command to be executed.

In such cases, a technique called copy-on-write (COW) is used. With this technique, when a fork occurs, the parent process's pages are not copied for the child process. Instead, the pages are shared between the child and the parent process. Whenever a process (parent or child) modifies a page, a separate copy of that particular page alone is made for that process (parent or child) which performed the modification. This process will then use the newly copied page rather than the shared one in all future references. }

(B). Consider the methods used by processes P1 and P2 for accessing their critical sections whenever needed, as given below. The initial values of shared boolean variables S1 and S2 are randomly assigned.

Method used by P1	Method by P2
While (S1==S2); critical section S1=S2;	While (S1!=S2); critical section S2 = not(S1);

Which one of the following statements describes the properties achieved?
(i) Mutual exclusion but not progress
(ii) Progress but not mutual exclusion
(iii) Neither mutual exclusion nor progress
(iv) Both mutual exclusion and progress

Ans : option (i)

Explanation:
Principle of Mutual Exclusion: No two processes may be simultaneously present in the critical section at the same time. That is, if one process is present in the critical section other should not be allowed. P1 can enter critical section only if S1 is not equal to S2, and P2 can enter critical section only if S1 is equal to S2. Therefore Mutual Exclusion is satisfied.

Progress: no process running outside the critical section should block the other interested process from entering critical section whenever critical section is free.
Suppose P1 after executing critical section again want to execute the critical section and P2 dont want to enter the critical section, then in that case P1 has to unnecesarily wait for P2. Hence progress is not satisfied.

(C). The enter_CS() and leave_CS() functions to implement critical section of a process are realized using test-and-set instruction as follows:

```
void enter_CS(X)
{
        while test-and-set(X) ;
}

void leave_CS(X)
{
        X = 0;
}
```

In the above solution, X is a memory location associated with the CS and is initialized to 0.
Now consider the following statements:
I. The above solution to CS problem is deadlock-free
II. The solution is starvation free.
III. The processes enter CS in FIFO order.
IV. More than one process can enter CS at the same time.

Which of the above statements is TRUE?
(i) I only
(ii) I and II
(iii) II and III
(iv) IV only

Ans: option (i)

Explanation:

The test-and-set instruction is an instruction used to write to a memory location and return its old value as a single atomic (i.e., non-interruptible) operation. Since it is an atomic instruction it guarantees mutual exclusion.

(D). A uni-processor computer system only has two processes, both of which alternate 10 ms CPU bursts with 90 ms I/O bursts. Both the processes were created at nearly the same time. The I/O of both processes can proceed in parallel. Which ofthe following scheduling strategies will result in the least CPU utilization (over a long period of time) for this system?
(i) First come first served scheduling
(ii) Shortest remaining time first scheduling
(iii) Static priority scheduling with different priorities for the two processes
(iv) Round robin scheduling with a time quantum of 5 ms

Ans: option (iv)

Explanation :

When Round Robin scheduling is used

We are given that the time slice is 5ms. Consider process P and Q. Say P utilizes 5ms of CPU and then Q utilizes 5ms of CPU. Hence after 15ms P starts with I/O And after 20ms Q also starts with I/O. Since I/O can be done in parallel, P finishes I\O at 105th ms (15 + 90) and Q finishes its I\O at 110th ms (20 + 90). Therefore we can see that CPU remains idle from 20th to 105th ms.

Thus, when Round Robin scheduling is used,
Idle time of CPU = 85ms
CPU Utilization = 20/105 = 19.05%

When First Come First Served scheduling scheduling or Shortest Remaining Time First is used

Say P utilizes 10ms of CPU and then starts its I/O. At 11th ms Q starts processing. Q utilizes 10ms of CPU.
P completes its I/O at 100ms (10 + 90)
Q completes its I/O at 110ms (20 + 90)
At 101th ms P again utilizes CPU. Hence, Idle time of CPU = 80ms
CPU Utilization = 20/100 = 20%

Since only two processes are involved and I\O time is much more than CPU time, "Static priority scheduling with different priorities" for the two processes reduces to FCFS or Shortest remaining time first.

Therefore, Round robin will result in least CPU utilization.

(E). The following program:

```
main(){
    if(fork()>0)
    sleep(100);
}
```

results in the creation of:
(i) an orphan process
(ii) a zombie process
(iii) a process that executes forever
(iv) None of these

Ans: option (ii)

Explanation :

The fork() call creates a child process and checks its return value. In case the return value of fork() comes out to be grater than zero, it executes the parent process and goes to sleep. Otherwise, the child is executed and it finishes before the parent. Since the child process finishes before the parent process and the parent does not call wait, the child process becomes a zombie.

3. (A). You write a UNIX shell, but instead of calling fork() then exec() to launch a new job, you instead insert a subtle difference: the code first calls exec() and then calls fork() like the following:

[2 marks]

```
shell (..) {
.. ..
exec (cmd, args);
fork();
.. ..
}
```

Does it work? What is the impact of this change to the shell, if any? Explain.

Ans : Doesn't work. Shell's address space is entirely replaced with the new command (cmd), therefore the shell will terminate once cmd is terminated.

3. (B). What is busy waiting with respect to a critical section problem? Can busy waiting be avoided? [2 marks]

Ans: Waiting is the act of suspending the current thread of execution until some future event which might be the availability of a contested resource, the passage of time, or the release of a lock.

Busy waiting means a process is waiting for an event to occur and it does so by executing instructions. Systems implement busy waiting by simply spinning in a tight loop, constantly checking if the event in question has occurred. Busy waiting wastes CPU cycles since nothing useful is done during looping.

Alternative to busy waiting is sleeping. There needs to be built a list of threads who wish to wait, called a wait queue. The kernel is asked to wake up a process from the list whenever the event in question happens. For example, the kernel might be asked to to wake up a thread when a specific mutex becomes available. You then yield to the kernel, allowing it to schedule something else instead of you.

The benefits of sleeping over busy looping is that the kernel can run something useful instead for the duration of the wait. The downside is the overhead: Managing the list, putting the thread to sleep, and context switching into a new process .

4. Consider the workload in the following table :

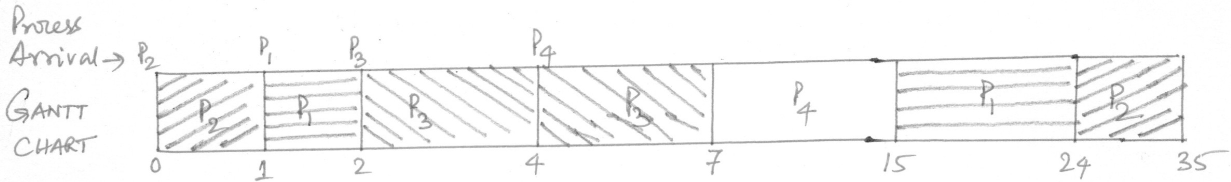
Process	Burst Time	Priority	Arrival Time
P1	10	4	1
P2	12	3	0
P3	5	2	2
P4	8	1	4

Draw the Gantt chart for preemptive shortest job first and preemptive priority scheduling .
What is the average waiting time and response time in each case? [4 marks]

Ans :

4. (a) Pre-emptive Shortest Job First Scheduling

Process	Arrival Time	Burst Time	Priority	Completion Time	Turn Around Time	Waiting Time	Response Time
P_1	1	10	4	24	23	13	0
P_2	0	12	3	35	35	23	0
P_3	2	5	2	7	5	0	0
P_4	4	8	1	15	11	3	3



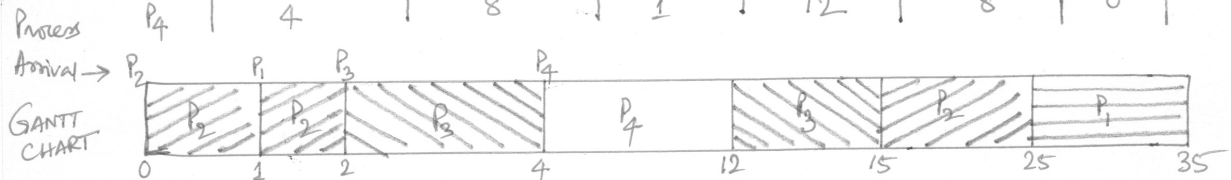
READY QUEUE		P_2	P_1, P_2	P_1, P_2, P_4	P_1, P_2	P_2	

$$\text{Average Waiting Time } (AVG_{WT}) = \frac{13+23+0+3}{4} = \frac{39}{4} = 9.75$$

$$\text{Average Response Time } (AVG_{RT}) = \frac{0+0+0+3}{4} = \frac{3}{4} = 0.75$$

(b) Pre-emptive Priority Scheduling (Lower Priority no. ⇒ Higher priority)

Process	Arrival Time	Burst Time	Priority	Completion Time	Turn Around Time	Waiting Time	Response Time
P_1	1	10	4	35	34	24	25
P_2	0	12	3	25	25	13	
P_3	2	5	2	15	13	8	
P_4	4	8	1	12	8	0	



READY QUEUE		P_1	P_1, P_2	P_1, P_2, P_3	P_1, P_2	P_1	

$$\text{Average Waiting Time } (AVG_{WT}) = \frac{24+13+8+0}{4} = \frac{45}{4} = 11.25$$

$$\text{Average Response Time } (AVG_{RT}) = \frac{25+0+0+0}{4} = \frac{25}{4} = 6.25$$

5 (A). What are user level and kernel level threads? [1mark]

Ans :

User Level Threads

User level threads are managed by a user level library however, they still require a kernel system call to operate. The kernel knows nothing about thread management and only takes care of the execution part. User level threads are typically fast. Creating threads, switching between threads and synchronizing threads only needs a procedure call. They are a good choice for non blocking tasks otherwise the entire process will block if any of the threads blocks.

Kernel Level Threads

Kernel level threads are managed by the OS, therefore, thread operations (ex. Scheduling) are implemented in the kernel code. This means kernel level threads may favor thread heavy processes. Moreover, they can also utilize multiprocessor systems by splitting threads on different processors or cores. They are a good choice for processes that block frequently. If one thread blocks it does not cause the entire process to block. Kernel level threads have disadvantages as well. They are slower than user level threads due to the management overhead. Kernel level context switch involves more steps than just saving some registers. Finally, they are not portable because the implementation is operating system dependent.

(B). Assume you want to implement a web-server for YouTube by using multithreading, where each thread serves one incoming request by loading a video file from the disk. Assume the OS only provides the normal blocking read system call for disk reads. Which threads should be used, user-level threads or kernel-level threads? Why? [2 marks]

Ans : Kernel-level threads. Because each thread will make blocking I/O calls. With kernel-level thread, one thread won't block others. But if user-level thread is used, then one thread will block all other thread

OR

(B). You want to implement a web-server for Facebook, to serve each user's "Home" page (the first page you see after you log in). Your web-server needs to perform many tasks: load the news feeds from each of your friends, load the advertisement, check for new messages, etc. Now you want to implement your web-server by using multithreading, and have one thread to perform each of the tasks, and later these threads will cooperate with each other to collectively construct the "Home" page. For performance reasons, Facebook makes sure that all the data these threads need is already cached in the memory (so they don't have to perform any disk I/O). What do you use, user-level threads or kernel-level threads? Why? [2 marks]

Ans : User-level thread. Here since the concern of user-level thread, namely "one thread can block all other threads within the same process", no longer exists (as threads won't make blocking I/O calls), so we can use user-level thread for its efficiency. This is in particular beneficial since these threads needs to communicate frequently with each other. If kernel-thread is used, everytime such communication needs to go through the kernel, which is more expensive.

6 (A). List the different inter process communication (IPC) mechanisms (with examples).

[2 marks]

Ans :

1. Shared Memory : Shared memory allows one or more processes to communicate via memory that appears in all of their virtual address spaces. Each process that wishes to share the memory must attach to that virtual memory via a system call. The process can choose where in its virtual address space the shared memory goes or it can let Linux choose a free area large enough. When processes no longer wish to share the virtual memory, they detach from it. So long as other processes are still using the memory the detach only affects the current process. Its data structure is removed from the shared memory data structure and deallocated. The current process's page tables are updated to invalidate the area of virtual memory that it used to share. When the last process sharing the memory detaches from it, the pages of the shared memory current in physical memory are freed.

2. Pipe : A pipe allows for data flow in one direction; when bidirectional communication is needed, two pipes need to be created.

Only related processes (those in the same branch of the process tree) can communicate through a pipe.

The pipe is created by system call pipe():

int pipe(int filedes[2]);
- creates a pair of file descriptors, pointing to a pipe inode, and places them in the array pointed to by filedes. filedes[0] is for reading, filedes[1] is for writing.

3. Message Queues : Differs from pipes in that the caller need not (but can) read the messages in FIFO manner; it can select the message it wants to acquire instead.

The message has the predetermined structure:

```
struct msgbuf {
        long mtype;     /* message type, must be > 0 */
        char mtext[1];  /* message data  - variable length*/
};
```

4. Signals : They are used to signal asynchronous events to one or more processes. A signal could be generated by a keyboard interrupt or an error condition such as the process attempting to access a non-existent location in its virtual memory. Signals are also used by the shells to signal job control commands to their child processes. Ex. signal(), kill(), alarm()

(B). The following code in C uses an IPC mechanism to communicate between two processes. Identify the IPC mechanism used and document the code (fill the comments sections in the code) to highlight the usage of the different POSIX system calls used for IPC. Additionally, you need to comment on the output. [2 marks]

```
main(){
        int shmid,status;
        int *a, *b;
        int i;

      shmid = shmget(IPC_PRIVATE, 2*sizeof(int), 0777|IPC_CREAT);

           /*The operating system keeps track of the set of shared memory segments. In order to
acquire shared memory, we must first request the shared memory from the OS using the
shmget() system call. The second parameter specifies the number of bytes of memory
requested. shmget() returns a shared memory dentifier (SHMID) which is an integer. */

     if (fork() == 0) {                              /* Child process */

          b = (int *) shmat(shmid, 0, 0);            /* the parent and child must "attach" the
                                                        shared  memory to its local data segment.
                                                       This is done by the shmat() system call.
                                                       shmat() takes the SHMID of the shared
                                                       memory segment as input parameter and
                                                       returns the address at which the segment
                                                       has been attached. Thus shmat() returns a
                                                       char  pointer.  */

           for( i=0; i< 10; i++) {

              sleep(5);

              printf("\t\t\t Child reads: %d,%d\n",b[0],b[1]);    /* Child reads data written  in
                                                                        the shared memory
                                                                  segment by the parent process  */

                }

        shmdt(b);                                    /*each process should "detach" itself from
                                                        the  shared memory after it is used   */

       }

 else {                                              /*  Parent process */

          a = (int *) shmat(shmid, 0, 0);            /* shmat() returns a char pointer which is
                                                         typecast here to int and the address is
                                                         stored in the int pointer a. Thus the
                                                         memory locations a[0] and a[1] of the
                                                         parent are the same as the memory
                                                         locations b[0] and b[1] of the parent, since
                                                         the memory is shared. */

          a[0] = 0; a[1] = 1;

                for( i=0; i< 10; i++) {
```

```
            sleep(1);
            a[0] = a[0] + a[1];
            a[1] = a[0] + a[1];
            printf("Parent writes: %d,%d\n",a[0],a[1]);  /**/
        }

        wait(&status);                          /* The parent waits until child exits */

        shmdt(a);                               /*each process should "detach" itself from
                                                   the  shared memory after it is used   */

        shmctl(shmid, IPC_RMID, 0);             /*Child has exited, so parent process should
                                                   delete  the cretaed shared memory. Unlike
                                                   attach and detach, which is to be done for
                                                   each process separately, deleting the
                                                   shared memory has to be done by only
                                                   one process after making sure that noone
                                                   else   will be using it  */

}}
```

Comment on Output :

1. Child reads all the values written by the parent. However, the child does not print the same values again.

2. On modifying sleep() in parent/child process it is seen that either the the writer is faster than the reader or the reader is faster than the writer. If the writer is faster, the reader may miss some of the values written into the shared memory. Similarly, when the reader is faster than the writer, the reader may read the same values more than once.