INDIAN INSTITUTE OF INFORMATION TECHNOLOGY, ALLAHABAD
Mid-Semester Examination, September 2017

Date of Examination : 22.09.2017 (2nd Session)

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, Section C - S.Ma)

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

Note: There are five **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). Consider three CPU-intensive processes, which require 10, 20 and 30 time units and arrive at times 0, 2 and 6, respectively. How many context switches are needed if the operating system implements a shortest remaining time first scheduling algorithm? Do not count the context switches at time zero and at the end.
(i) 1 (ii) 2
(iii) 3 (iv) 4

Ans : *process execute in this way ...*

0 ---- p1---- 10 (switching)--------p2-----30(switching)-----p3-----60

Only two context switching possible (since we did not consider the starting and ending switching or even the OS scheduler switching)

(B). What are the main goals today in the design of operating systems?

(i) Convenience for user (ii) efficient utilization of the computer resources (CPU, memory, I/O devices),
(iii) expandibility (iv) all the above

Ans : iv

(C) If a thread invokes the *exec* system call,
(i) only the *exec* executes as a separate process. (ii) the program specified in the parameter to *exec* will replace the entire process.

(iii) the *exec* is ignored as it is invoked by a thread. (iv) None of these

Ans : ii

(D). A CPU has two modes-privileged and non-privileged. In order to change the mode from privileged to non-privileged
(i) a hardware interrupt is needed (ii) a software interrupt is needed
(iii) a privileged instruction (which does not generate an interrupt) is needed
(iv) a non-privileged instruction (which does not generate an interrupt) is needed

Ans : iv

Explanation : Kernel mode to user mode is done via an instruction so neither interrupts, h/w or s/w is needed as these are not instructions. When we move from kernel mode to user mode , we move towards non priviledged mode. Thus, it is not necessary to be privileged to invoke the task . So non priviledged instruction can take from kernel to user mode. To change the mode from kernel mode to user mode ...only a bit(mode bit) is changed in register (by setting mode bit to 1).

(E) How many times does the following C program print *IIITA* :

```
main(){
fork();fork();printf("IIITA");
}
```

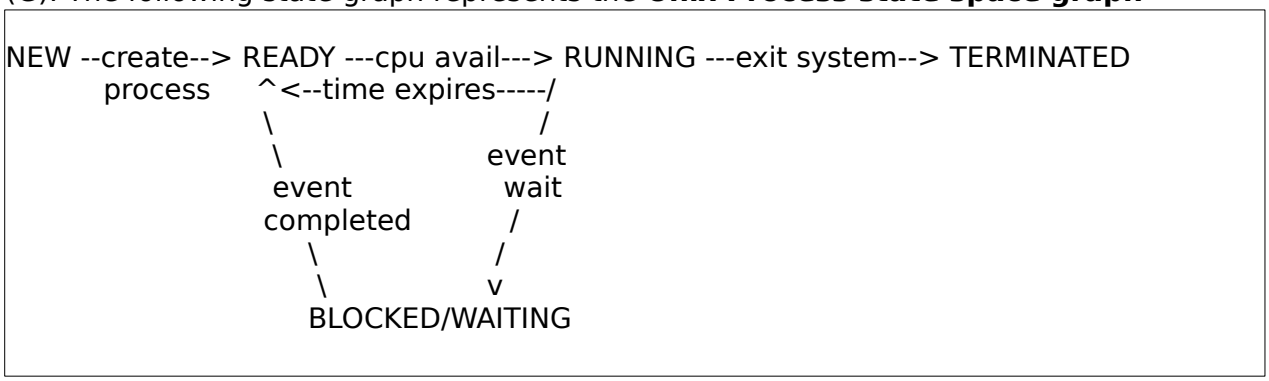
(i) only once (ii) twice (iii) four times (iv) eight times

Ans : (iii) . The first fork() creates a parent child pair. In the second fork() both parent and child create one more process each. Thus, in total there are four processes after the second fork() and each one of them print IIITA.

(F) System calls are usually invoked by using
(i) a software interrupt (ii) polling (iii) an indirect jump (iv) a privileged instruction

Ans : (i)

(G). The following state graph represents the **Unix Process state space graph**



a: create process; b: time expires; c: event wait; d: event completed; e: exit

I: *I/O read* II: *fork() or pthread_create()* III: *Segmentation fault*

IV: *DMA controller interrupts CPU signalling the completion of an I/O read*

V : *Timer interrupt*

a,b,c,d,e represent the five state transition events of the state space graph given above, and **I,II,III,IV and V** are the five possible situations for the events. Choose the combinations that match the situations to their corresponding events.

(i) a-I, b-II, c-III, d-IV, e-V (ii) a-II, b-III, c-I, d-V, e-IV

(iii) a-II, b-V, c-I, d-IV, e-III (iv) None of the above

Ans : (iii)

(H). Consider a user level threading library that multiplexes *N>1* user level threads over *M >= 1* kernel threads. The library manages the concurrent scheduling of the multiple user threads that map to the same kernel thread internally, and the programmer using the library has no visibility or control on this scheduling or on the mapping between user threads and kernel threads. The N user level threads all access and update a shared data structure. When (or, under what conditions) should the user level threads use mutexes to guarantee the consistency of the shared data structure?
(i) Only if M>1 (ii)Only if N > M
(iii) Only if the M kernel threads can run in parallel on a multi-core machine.
(iv) User level threads should always use mutexes to protect shared data.

Ans : (iv)

(I). If the waiting time for a process is p and there are n processes in the memory then the CPU utilization is given by,
(i) p/n (ii) p^n (p raised to n)
(iii) 1-p^n (iv) n-(p^n)

Ans : (iii) A process waits when it is doing some I/O operation. Given, the waiting time of 1 process is p. Therefore, the probability that the process is doing some I/O = p.
Probability that two processes are doing I/O = p*p (using bayes theorem).
Probability that n processes are doing I/O = p*p*....(n terms) (using bayes theorem).
Therefore, Probability that atleast one of the n processes is not doing I/O
= 1- (p*p*.....(n terms)) = 1-p^n. This is the probability that the CPU remains busy and is the its utilization

(J). Consider three processes (process id 0, 1, 2 respectively) with compute time bursts 2, 4 and 8 time units. All processes arrive at time zero. Consider the longest remaining time first (LRTF) scheduling algorithm. In LRTF ties are broken by giving priority to the process with the lowest process id. The average turn around time is:

(i) 13 units (ii) 14 units
(iii) 15 units (iv) 16 units

Ans : (i). Let the processes be p0, p1 and p2. These processes will be executed in following order.

p2	p1	p2	p1	p2	p0	p1	p2	p0	p1	p2	
0	4	5	6	7	8	9	10	11	12	13	14

Turn around time of a process is total time between submission of the process and its completion.
Turn around time of p0 = 12 (12-0)
Turn around time of p1 = 13 (13-0)
Turn around time of p2 = 14 (14-0)

Average turn around time is (12+13+14)/3 = 13.

2. **Answer the following with justification.** **[10 marks]**

(A). What scheduling policy will you use for each of the following cases? Explain your reasons for choosing them.

(i). The processes arrive at large time intervals:

Ans : FCFS.
SJF is diificult to implement practically. There is no need to preempt the running job since the next job comes after a long time thus RR is not a good option.

(ii). All the processes take almost equal amounts of time to complete.

Ans : FCFS or RR. Easy to implement and no chance of starvation or convoy effect

(B.) Consider two threads that concurrently execute a line of code

count = count + 1

The variable *count* starts out with a value 1, and ends up with a value of 2 after both threads have incremented it once each. Explain the sequence of events that could lead to such a situation.

Ans : First thread reads *count* into CPU register. CPU switches to second thread. Second thread reads value of *count* and increments. CPU again switches back to the first thread which increments based on the old value of *count* read previously. Both threads then write a value of 2 one after the other.

(C) Two processes, P1 and P2, need to access a critical section of code. Consider the following synchronization construct used by the processes:

/* P1 */	/* P2 */
while (true) { *wants1 = true;* *while (wants2 == true);* */* Critical* *Section */* *wants1=false;* *}* /* Remainder section */	*while (true) {* *wants2 = true;* *while (wants1==true);* */* Critical* *Section */* *wants2 = false;* *}* /* Remainder section */

Here, wants1 and wants2 are shared variables, which are initialized to false. Which one of the following statements is TRUE about the above construct? Justify your answer.
(i) It does not ensure mutual exclusion.
(ii) It does not ensure bounded waiting.
(iii) It requires that processes enter the critical section in strict alternation.
(iv) It does not prevent deadlocks, but ensures mutual exclusion.

Ans : (iv). The above synchronization constructs don't prevent deadlock. When both wants1 and wants2 become true, both P1 and P2 stuck forever in their while loops waiting for each other to finish.

(D) A forked process may share a Run-Time Stack with its parent . Answer true/false with justification.

Ans : True. For child process text(code),data,stack is same as calling process.

(E) Consider a parent process P that has forked a child process in the C program below.

```
int i = 5;
int fd = open(...) //opening a file
int pid = fork();
if(pid >0) {
close(fd);
a = 6;...
}
else if(pid ==0) {
printf("i=%d\n", i);
read(fd, something);
}
```

Assume that the parent process is scheduled first, and the OS implements copy-on-write during fork. Once the parent resumes after fork, it closes the file descriptor and changes the value of a variable as shown above. Assume that the child process is scheduled for the first time only after the parent completes these two changes.

(a) What is the value of the variable `i in parent and child process` as printed in the child process, when it is scheduled next?

(b) Will the attempt to read from the file descriptor succeed in the child? Explain.

Ans : (a) 5. The value is only changed in the parent.
(b) Yes, the file is only closed in the parent

3. The code below is written to provide one possible solution to the READERS/WRITERS problem. Consider the **P = Wait operation and V = Signal operation**

```
int readCount=0, writeCount=0;
semaphore mutex1=1, mutex2=1;
semaphore readBlock=1, writeBlock=1,

reader() {
  while(TRUE) {
    <other computing>;

     P(readBlock);
       P(mutex1);
         readCount++;
         if(readCount == 1)
           P(writeBlock);
       V(mutex1);
     V(readBlock);

     access(resource);
    P(mutex1);
     readCount--;
     if(readCount == 0)
       V(writeBlock);
    V(mutex1);
  }
}

writer() {
  while(TRUE) {
    <other computing>;
    P(mutex2);
     writeCount++;
     if(writeCount == 1)
       P(readBlock);
    V(mutex2);
    P(writeBlock);
     access(resource);
```

```
   V(writeBlock);
   P(mutex2)
     writeCount--;
     if(writeCount == 0)
       V(readBlock);
   V(mutex2);
  }
}
```

Answer the following questions based on the code given above. **[5 marks]**
(i) Explain the role of each semaphore used in this code.
(ii) Does this code provide a correct and fair solution for the readers/writers problem?
(iii) Could you suggest a solution which is both correct and fair?

Ans :

(i) mutex1: to provide mutually exclusive access to the shared variable *"readcount"* by multiple readers.

mutex2: to provide mutually exclusive access to the shared variable *"writecount"* by multiple writers.

readBlock: if a reader arrives while a writer is inside the critical section(CS) and other writers are waiting, this semaphore will cause that reader to get blocked until the last writer leaves the CS.

writeBlock: This semaphore causes any writer to get blocked if there is one writer or reader(s) currently accessing the CS.

(ii) Even though this code is correct, it does not provide a fair solution. With the way it is written, there is a possibility that writers may starve readers if they arrive one after another. In other words, readers may never get a chance to enter the critical section (i.e. access the shared resource) if the writers don't take a break.

(iii) Yes.

Introduce a new semaphore called "writePending" which is shared by both readers and writers. It will force the blocked readers and writers to enter a waiting queue in the order that they have arrived. The associated P and V calls are added to the original code as follows:

```
int readCount=0, writeCount=0;
semaphore mutex1=1, mutex2=1;
semaphore readBlock=1, writeBlock=1;
semaphore writePending=1;

reader() {
  while(TRUE) {
    <other computing>;
    P(writePending);
      P(readBlock);
        P(mutex1);
          .......
        V(mutex1);
      V(readBlock);
    V(writePending);
      access(resource);
    .........
```

```
}
}

writer() {
  while(TRUE) {
    <other computing>;
   P(writePending);
    P(mutex2);
      ........
    V(mutex2);
    P(writeBlock);
      access(resource);
    V(writeBlock);
   V(writePending);

    ...........
  }
}
```

---

4 (A). Gryffindor group (gryff) lead by Harry Potter is competing against Slytherin group (slyth) led by Voldemort for a potion kept at the Chamber of Secret at the Hogwarts school of wizardry. The Principal, Professor Dumbeldore wants to have a fair competition. He wants an expert (you) to propose a solution based on the following rules:

- there should never be more than three people in the Chamber of Secrets at once.
- there should never be both a Gryffindor and Slytherin group member in the Chamber of Secrets at once
- there should never be a deadlock

Here is one possible solution, involving several shared variables and concurrent processes: Consider the **P = Wait operation and V = Signal operation**

Shared Variables:

```
int num_gryff = 0, num_slyth = 0;
semaphore gryff_mutex = 1;
semaphore slyth_mutex = 1;
semaphore gryff_token = 3;
semaphore slyth_token = 3;
semaphore no_gryff  = 1;
semaphore no_ slyth = 1;
```

gryff() { *while (TRUE) {* *P(gryff _token);* *P(gryff _mutex);* *num_gryff = num_gryff + 1;* *if (num_gryff == 1) {* *P(no_slyth);* *P(no_gryff);* *}* *V(gryff _mutex); // use the chamber* *P(gryff _mutex);* *num_gryff = num_gryff – 1;* *if (num_gryff == 0) {* *V(no_slyth);* *V(no_gryff);* *}* *V(gryff _mutex);* *V(gryff _token);* *}* *}*	*slyth () {* *while (TRUE) {* *P(slyth _token);* *P(slyth _mutex);* *num_ slyth = num_ slyth + 1;* *if (num_slyth == 1) {* *P(no_gryff);* *P(no_slyth);* *}* *V(slyth_mutex); // use the chamber* *P(slyth_mutex);* *num_ slyth = num_ slyth – 1;* *if (num_ slyth == 0) {* *V(no_gryff);* *V(no_slyth);* *}* *V(slyth_mutex);* *V(slyth_token);* *}* *}*

(a) Which semaphore ensures that no more than 3 gryffindors are in the chamber at the same time?
(b) Which semaphore ensures that only one process at a time will have access to the shared variable, *num_gryff*?

(c) *no_slyth* and *no_gryff* are semaphores that indicate that there are no gryffindor (or slytherin) currently in the chamber (if they are set to 1). What is the purpose of this conditional statement:

```
if (num_gryff == 1) {
                P(no_slyth);
               P(no_gryff);
        }
```

(d) Does this solution prevent deadlock? If not, show a specific example (of deadlock). **[5 marks]**

Ans :
(a) gryff_token. This is the one that was initialized to 3.

(b) gryff_mutex. This is the one that was initialized to 1, and its acquired right before each access to num_gryff, then released again soon after the access to num_gryff.

(c) The first gryffindor attempting to enter has to do two things: (1) wait until there are no slytherin, by doing wait(no_gryff). The last slytherin out will signal that same semaphore, which is what would let that first gryyfindor in. (2) prevent any new slytherin from entering simultaneously, which is accomplished by calling wait(no_gryff). The new sltherin would be the first slytherin, and would also call wait(no_gryff), and only one of these would succeed without blocking

(d) No, it does not prevent deadlock. Suppose there are no gryffindor and no slytherin in the chamber. All semaphores will have the same value they started with. Suppose a gryyfindor and a slytherin both try to enter. Both execute the first five lines of their loops (in any order). At this point, the gryffindor has just called wait(no_slyth) and it succeeded. The slytherin has called wait(no_gryff) and it succeeded. Both of those semaphores are now at 0. Now the gryffindor calls wait(no_gryff) and blocks. The slytherin calls wait(no_slyth) and blocks. But also, the gryffindor is still holding gryff_mutex, and the slytherin is holding slyth_mutex. So no other gryffindor or slytherin can make any progress either – they will block at the first line of the loop. All processes are blocked forever. This is deadlock.

5. Answer the following questions. **[1+1+3 = 5 marks]**

(a) Can an application process and a kernel process set up a shared memory region? Justify your answer.

Ans : No. The application processes do not have access to the kernel space and the kernel processes do not have the user space part.

(b) Explain either the message passing or pipe based IPC mechanism.

Ans : Consider an application that requires one process to write a set of values, that are to be read by the other. Pipe is a system construct which facilitates such communication. A pipe is also a shared buffer. When a process tries to read from an empty pipe, it waits until someone has written something into the pipe. When the pipe is full, any process attempting to write into the pipe is made to wait.

A pipe is treated as a file by the system. Unlike in a file, we may want to both read and write from , pipe at the same time. Hence when a pipe is created, two file descriptors are created -- one for reading the pipe and one for writing into the pipe. The pipe() system call requires an array of two integers as parameter. The system returns the file descriptors through this array.

(c) Inter Process Communication using shared memory :

Write two programs writer process (writer.c) and a reader process (reader.c) implementing the following:
i. The writer writes some sequence of data into the shared memory (say phone nos.) and waits for the reader to read it.
ii. The reader reads the data from shared memory segment and prints them.

Ans : Refer to the codes : *shared_memory* and *read-write1.c* codes @ **http://profile.iiita.ac.in/bibhas.ghoshal/teaching_os_lab.html**

 Faculty Signature _____