Birla Institute of Technology & Science Pilani, Hyderabad Campus
First Semester 2023-2024
CS F372: Operating Systems
Mid Semester Examination (Regular)

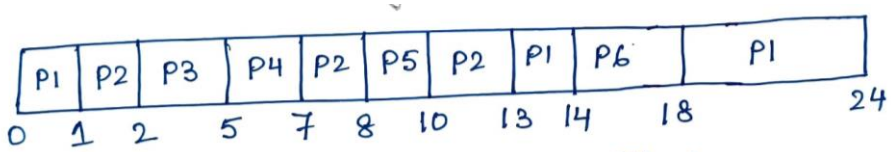
Type: Closed Book **Time: 90 minutes** **Max Marks: 105** **Date: 12/10/2023**

1.a) Consider the processes shown in **Table 1** along with their arrival times and CPU burst cycles given in milliseconds. The processes are scheduled using the **SRTF** algorithm. In case of a tie, process selection is done on FCFS basis. Show a Gantt chart depicting the order of process execution. Calculate the finish time, waiting time and turnaround time of each process, and the average waiting time and the average turnaround time. You are not required to show calculations for waiting time, turnaround time, average waiting time and average turnaround time. Marks will not be awarded if final answers are incorrect. **[12]**

Process	Arrival Time	CPU Burst Cycle (milliseconds)
P1	0	8
P2	1	5
P3	2	3
P4	4	2
P5	8	2
P6	14	4

Table 1

Ans.



Finish Time
P1 → 24
P2 → 13
P3 → 5
P4 → 7
P5 → 10
P6 → 18

Waiting Time
P1 → (0-0) + (13-1) + (18-14)
= 0+12+4 = 16
P2 → (1-1) + (7-2) + (10-8)
= 0+5+2 = 7
P3 → (2-2) = 0
P4 → (5-4) = 1
P5 → (8-8) = 0
P6 → (14-14) = 0

Turnaround Time
P1 → 24-0 = 24, P2 → 13-1 = 12, P3 → 5-2 = 3
P4 → 7-4 = 3, P5 → 10-8 = 2, P6 → 18-14 = 4
Avg. waiting time = 24/6 = 4 ms
Avg. turnaround time = 48/6 = 8 ms

Marking Scheme: 2 marks for Gantt chart, 0.5 marks for each finish time, 0.5 marks for each waiting time, 0.5 marks for each turnaround time, 0.5 marks for avg. waiting time, 0.5 marks for avg. turnaround time. If process order is wrong in Gantt chart, no marks will be awarded for Gantt chart or any times.

1.b) Define data parallelism and task parallelism. **[2]**

Ans. Data parallelism distributes subsets of the same data across multiple cores and the same operation is performed on each subset.

Task parallelism distributes tasks/threads across cores, each thread performing unique operation. The threads may be operating on same or different data.

Marking Scheme: 1 mark for each type of parallelism.

2.a) Consider the following message queue with the initial configuration as shown in **Table 2** and answer the questions that follow. The message queue uses 0-based indexing as shown in the 1st column of the table. The 2nd and the 3rd columns together represent the message queue contents at each position/index.

	Queue Position/Index	mtype	mtext
Head of queue →	0	1	"hello"
	1	2	"all the best"
	2	2	"for the mid-semester"
	3	3	"examination"
Tail of queue →	4	3	"bye"

Table 2

Consider that the message queue structure has two members: `long mtype` (2nd column of Table 2) and `char mtext[200]` (3rd column of Table 2). For each of the following POSIX compliant `msgrcv()` operations executed on an Ubuntu system, performed in sequence, state which of the messages will be read from the queue **and explain your answers properly.** Note that each operation (depending on the behaviour) changes the configuration of the queue and affects the subsequent operations. **[10]**

i) `msgrcv(msqid, buffer, sizeof(buffer.mtext), 0, 0);`
ii) `msgrcv(msqid, buffer, sizeof(buffer.mtext), 1, IPC_NOWAIT);`
iii) `msgrcv(msqid, buffer, sizeof(buffer.mtext), -2, 0);`
iv) `msgrcv(msqid, buffer, sizeof(buffer.mtext), 2, MSG_COPY | IPC_NOWAIT);`
v) `msgrcv(msqid, buffer, sizeof(buffer.mtext), 3, 0);`

Signature of `msgrcv()`:

```
ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);
```

Ans.

i) `msgrcv(msqid, buffer, sizeof(buffer.mtext), 0, 0);`
Since the mtype in msgrcv() is 0, the first message in the queue will be read. So, message in Queue Position 0 ("hello") will be read.

ii) `msgrcv(msqid, buffer, sizeof(buffer.mtext), 1, IPC_NOWAIT);`
The mtype in msgrcv() is 1, however, currently no message with mtype 1 exists in the queue. Also, flag IPC_NOWAIT is passed, so, msgrcv() will return immediately without blocking.

iii) `msgrcv(msqid, buffer, sizeof(buffer.mtext), -2, 0);`
Since the mtype in msgrcv() is -2, the first message in the queue with the mtype less than or equal to 2 will be read. So, message in the original Queue Position 1 ("all the best") will be read.

iv) `msgrcv(msqid, buffer, sizeof(buffer.mtext), 2, MSG_COPY | IPC_NOWAIT);`
Since the mtype in msgrcv() is 2, and MSG_COPY is specified in the flags, the message will be non-destructively read. That is, the message will not be removed from the queue after reading. Also, the message in the queue position specified by msgtype is fetched when MSG_COPY is used with the messages numbered starting from 0 (this is different from the behaviour when MSG_COPY is not specified). So, the message in the original Queue Position 4 ("bye") will be read.

v) `msgrcv(msqid, buffer, sizeof(buffer.mtext), 3, 0);`
Since the mtype in msgrcv() is 3, the first message of type 3 in the queue will be read. So, message in the original Queue Position 3 ("examination") will be read.

Marking Scheme: 2 marks for each part. Marks will be deducted for insufficient or unclear explanations.

2.b) What does POST stand for and who performs POST? **[1]**

Ans. POST stands for Power On Self Test.
BIOS or the Basic Input Output System performs POST.

Marking Scheme: 0.5 marks for writing the full form and 0.5 marks for writing BIOS performs POST. No part marking for either part.

3.a) Consider the following POSIX compliant C program which spawns two child processes. One of the child processes uses `execlp()` with '`ls`' as an argument and the other child process uses `execlp()` and '`wc -l`' to count the number of lines. The first child process redirects its standard output to the write end of a pipe and the second child process redirects its standard input to the read end of the same pipe. Essentially, the program prints the number of files in the current directory. Fill in the blanks in the program to align with the behaviour described. Assume that the code should run on an Ubuntu system. Write your answers as per the blank numbers mentioned in the code. Ambiguous answers without the blank numbers will not be considered. Assume that the correct header files have been already included. Assume that Lines A and B represent actual executable statements in the parent process. **[9]**

```
int main(void) {
      pid_t pid;
      int pfds[2], status1, status2;
      pipe(pfds);
      pid = fork();
      if (pid == ___1___) { // First Child Process
            dup2(___2___, ___3___);
            close(___4___);
            close(___5___);
            if (execlp(_____6_____) == -1) {
                  perror("Error in execlp()\n");
            }
      }
      else { // Parent process
            if (fork() == ___7___) { // Second Child Process
                  dup2(___8___, ___9___);
                  close(___10___);
                  close(___11___);
                  if (execlp(_____12_____) == -1) {
                        perror("Error in execlp()\n");
                  }
            }
            else {
                  close(___13___);
                  close(___14___);
                  wait(___15___);
                  wait(___16___);
                  // Line A: Parent prints exit status of First Child Process
                  // Line B: Parent prints exit status of Second Child Process
            }
      }
      return 0;
}
```

Signatures of the following functions might be helpful:

- `pid_t fork(void);`
- `int dup2(int oldfd, int newfd);`
- `int execlp(const char *file, const char *arg1, ..., const char *argN, NULL);`
- `pid_t wait(int *wstatus);`
- `int close(int fd);`
- `int pipe(int pipefd[2]);`

Ans.

1. **0**
2. **pfds[1]**
3. **1**
4. **pfds[0] //4 and 5 may be interchanged**
5. **pfds[1]**
6. **"ls", "ls", NULL //1st "ls" can be "/bin/ls"**
7. **0**
8. **pfds[0]**
9. **0**
10. **pfds[0] //10 and 11 may be interchanged**
11. **pfds[1]**
12. **"wc", "wc", "-l", NULL //1st "wc" can be "/bin/wc"**
13. **pfds[0] //13 and 14 may be interchanged**
14. **pfds[1]**
15. **&status1**
16. **&status2**

Marking Scheme: 0.5 marks for each blank except for blanks 6 and 12. Blanks 6 and 12 are of 1 mark each.

3.b) Describe the steps by which interrupts are handled in a computer system. No diagram is required. **[4]**

Ans. The steps are as follows:

1. When an interrupt is received, the execution of the current process is halted, the context (contents of CPU registers and program counter) of the current process is saved on the system stack.
2. The Interrupt Vector Table (IVT) is examined using the interrupt number to find the address of the required Interrupt Service Routine (ISR).
3. Once the address is found, the control is transferred to the ISR and the ISR is executed.
4. When ISR execution completes, control is transferred back to the interrupted process, the context of the process is retrieved from the stack and the process execution resumes.

Marking Scheme: 1 mark for each point.

4.a) Write a POSIX compliant C program that can run on an Ubuntu system and uses the Pthreads API. The main process creates a child process. The child process will be multithreaded. The child process first prints its own PID. Then, the child process prompts the user to enter the number of threads to create. Say this number is **n**. **n** should not be hardcoded or entered as a command line argument. It is guaranteed that **n** is an even number. The child process creates **n** threads. Out of these threads, **n/2** threads will perform one task and the remaining **n/2** threads will perform a different task. Each of the first set of **n/2** threads receives a positive integer from the child process and prints whether the integer is odd or even along with its own thread ID. This positive integer should not be hardcoded. Each of the remaining set of **n/2** threads receives a positive integer, say **x**, from the child process and prints the sum from 1 to **x** along with its own thread ID. **x** should not be hardcoded. The threads should not use `gettid()` to print their IDs. The child process waits for all the **n** threads to terminate. Note that each thread should have its own unique thread attribute object that should be initialized properly. The main process prints its own PID, waits for the child process to terminate and then displays the contents of two existing files, `input.txt` and `test.txt`, on the console. The file contents should be displayed using a single line of user written code. You are not allowed to open `input.txt` and `test.txt` in your program. Assume that the files are present in the same directory as your program. You cannot use `sleep()` to make any of the processes or threads wait. You are not allowed to use any

global variable. Your code should include all requisite header files. You are not required to perform any error handling. **[10.5]**

Ans.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include<pthread.h>

void *runner1(void *param1){
      int *p = (int *)param1;
      if((*p % 2) == 0)
            printf("\nPrinted by Thread %lu: The number %d is even\n",
pthread_self(), *p);
      else
            printf("\nPrinted by Thread %lu: The number %d is odd\n",
pthread_self(), *p);
      pthread_exit(0);
}

void *runner2(void *param2){
      int *a = (int *)param2;
      int sum = 0;
      for(int i = 1; i <= *a; i++)
            sum += i;
      printf("\nPrinted by Thread %lu: The sum upto %d is %d\n",
pthread_self(), *a, sum);
      pthread_exit(0);
}

int main()
{
      pid_t child;
      child = fork();
      if(child == 0){ //code for child
            printf("\nChild PID is %d\n",getpid());
            printf("\nEnter the number of threads to create");
            int num_thread; //will be an even no.
            scanf("%d",&num_thread);
            int arr[num_thread];
            pthread_t tid[num_thread];
            pthread_attr_t attr[num_thread];
            for(int i = 0; i < num_thread; i++)
                  pthread_attr_init(&attr[i]);
            for(int i = 0; i < num_thread/2; i++){
                  arr[i] = i+5;
                  pthread_create(&tid[i],&attr[i],runner1, &arr[i]);
            }
            for(int i = num_thread/2; i < num_thread; i++){
                  arr[i] = i+10;
                  pthread_create(&tid[i],&attr[i],runner2, &arr[i]);
            }
            for(int i = 0; i < num_thread; i++)
                  pthread_join(tid[i], NULL);
      }
      else{ //code for parent
            printf("\nParent PID is %d\n",getpid());
            wait(NULL);
```

```
            execlp("/bin/cat", "cat", "input.txt", "test.txt", NULL); //can be
just "cat"
      }
      return 0;
}
```

Marking Scheme:

- **Header files – 1 mark**
- **runner1() – 1.5 marks, 0.5 marks will be deducted if gettid() is used**
- **runner2() – 1.5 marks, 0.5 marks will be deducted if gettid() is used**
- **fork() call – 0.5 marks**
- **child process – 4 marks**
- **parent process – 2 marks**

4.b) What will be the output of the POSIX compliant C program given below on an Ubuntu system? Give explanation for your answer. The comments represent the corresponding tasks performed by the code. You should write your answer considering the comments. Assume all relevant header files are included. Assume all function calls execute successfully without any error. **[3.5]**

```
void *routine(void *param){
   printf("\nGREETINGS");
   pthread_exit(0);
}

int main(){
   pthread_t t1, t2, t3;
   pthread_attr_t a1, a2, a3;
   //initialize thread attribute objects a1, a2 and a3
   //create thread with t1 and a1, thread will execute routine()
   //create thread with t2 and a2, thread will execute routine()
   //create thread with t3 and a3, thread will execute routine();
   printf("%d",pthread_equal(t1, t2) + pthread_equal(t2, t3) +
   pthread_equal(t3, t1));
   //main thread waits for all 3 threads
   return 0;
}
```

Ans.
Output: 3 times GREETINGS will be printed and 0 will be printed once.
Explanation: 3 threads are created, each of which will execute routine(). Inside routine(), GREETINGS will get printed and the threads will exit. pthread_equal() returns if the 2 TIDs sent as arguments are not equal. Hence, each pthread_equal() invocation inside main() returns zero and hence the sum is also zero. So, corresponding to the printf() statement, zero is printed.

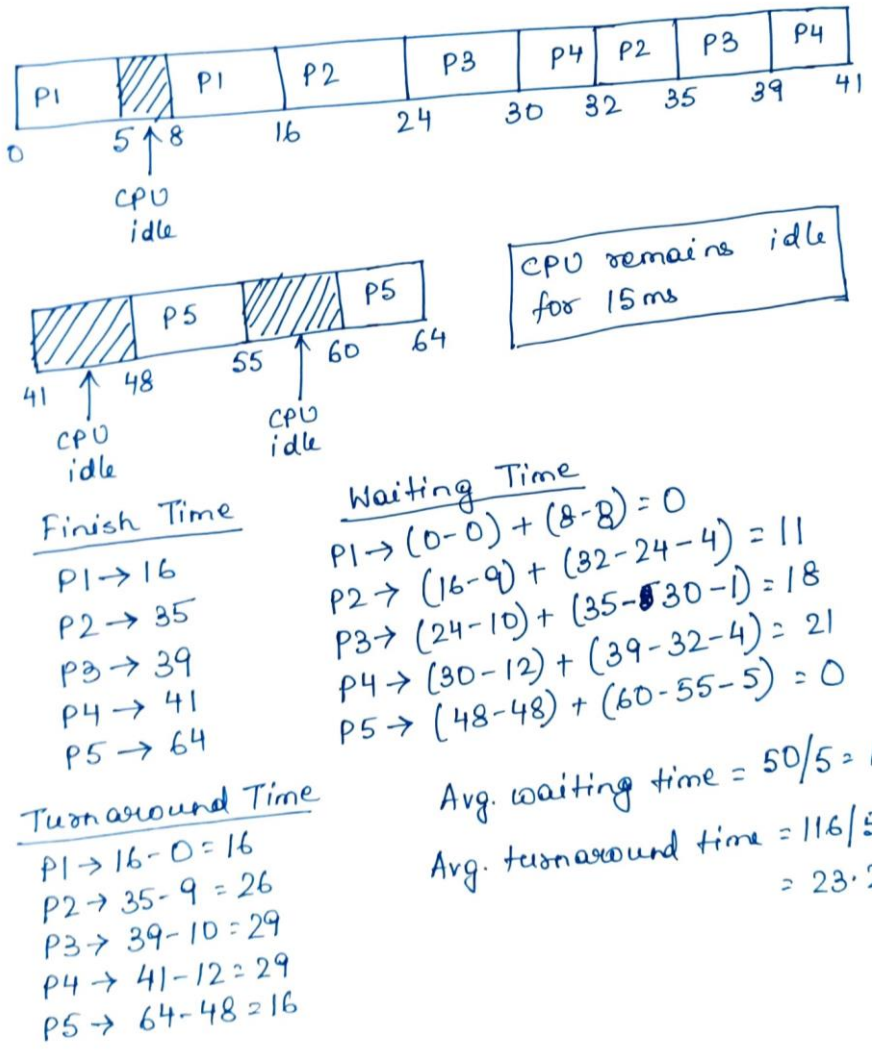
Marking Scheme: 2 marks for output. 1.5 marks for the explanation. Marks will be deducted for incomplete or unclear explanation.

5) Consider the processes shown in **Table 3** along with their arrival times and CPU burst cycles-I/O burst cycles-CPU burst cycles (3^{rd} column of Table 3, the CPU burst cycles are underlined) in milliseconds. The processes are scheduled using the **FCFS** algorithm. In case of a tie, the process with the lower PID is selected. Assume that a process does not have to wait for I/O device access after its CPU burst cycle is over. Show a Gantt chart depicting the order of process execution. Mention explicitly the duration for which the CPU remains idle. Calculate the finish time, waiting time and turnaround time of each process, and the average waiting time and the average turnaround time. You are not required to show calculations for waiting time, turnaround time, average waiting time and average turnaround time. Marks will not be awarded if final answers are incorrect. **[13]**

Process	Arrival Time	CPU Burst Cycle-I/O Burst Cycle-CPU Burst Cycle (milliseconds)
P1	0	5 – 3 – 8
P2	9	8 – 4 – 3
P3	10	6 – 1 – 4
P4	12	2 – 4 – 2
P5	48	7 – 5 – 4

Table 3

Ans.



Marking Scheme: 3.5 marks for Gantt chart, 1 mark for CPU idle time, 0.5 marks for each finish time, 0.5 marks for each waiting time, 0.5 marks for each turnaround time, 0.5 marks for avg. waiting time, 0.5 marks for avg. turnaround time. If process order is wrong in Gantt chart, no marks will be awarded for the Gantt chart and the times.

6.a) What will be the output of the POSIX compliant C program given below on an Ubuntu system? Give explanation for your answer. Assume all relevant header files are included and all function calls execute successfully without any error. **[6]**

```
int main()
{
   if(fork())
      if(fork())
```

```
        if(fork())
           if(fork() * fork())
              printf("\nGOODBYE\n");
   printf("\nNEVER MIND\n");
   return 0;
}
```

Ans. GOODBYE is printed once and NEVER MIND is printed 7 times.
Explanation: The 1st, 2nd and 3rd if statements evaluate to true for the parent process only. The execution of the 3 fork() calls results in the creation of 3 child processes. For the 4th if statement, the 1st fork() before the * operator is executed by the parent process and the parent creates a child process. The 2nd fork() is executed by both the parent and the child processes, resulting in the creation of 2 more child processes. Thus, a total of 6 child processes are created. The entire 4th if statement evaluates to true only for the parent process due to the presence of the * operator. Thus, GOODBYE is printed only by the parent process. The printf() corresponding to NEVER MIND is executed by the parent as well as the 6 child processes.

Marking Scheme: 2 marks for stating the output and 4 marks for the explanation. Marks will be deducted for stating the output incorrectly.

6.b) Write a short note on multiprogramming/batch system. Write the answer in text. No diagram is required. **[4]**

Ans. Multiprogramming/batch system is not an interactive system. Users submit the jobs to a job pool. The job pool is part of the secondary storage. The long-term scheduler / job scheduler selects a subset of the jobs from job pool/queue and brings these processes into the ready queue. The ready queue is part of the main memory. The CPU is allocated to the jobs in the ready queue until they terminate. While executing, the jobs may perform I/O operations for which they will interact with the secondary storage.

Marking Scheme: If only diagram is given, only 2 marks will be awarded.

6.c) What role does the medium-term scheduler play? No diagram is required. **[2]**

Ans. The medium-term scheduler is used to increase the degree of multiprogramming. It is used as an intermediate level of scheduling. Suppose, there are currently 5 processes, P1, P2, P3, P4 and P5 in the ready queue. If now P6 needs to be brought into the main memory, it is required to swap out one of the processes in the ready queue. One process out of P1, P2, P3, P4 and P5 is selected and is swapped out. The medium-term scheduler swaps out a partially executed process and swaps in a new process. The medium-term scheduler can swap out a partially executed process and can swap in another partially executed process which could have been swapped out earlier. Thus, medium-term scheduler is used for freeing up main memory. It can also be used to improve the process mix, i.e., if the no. of CPU-bound processes is more than the no. of I/O-bound processes or the other way round.

7) Consider the following pseudocode for 5 processes. The processes are executing on a uniprocessor system. The processes share a counting semaphore S initialized to 4. Assume semaphore implementation without busy waiting and the S->list is implemented as a FIFO queue. The value of S can become negative also. Assume that whenever a process gets blocked because of `wait()` execution, another process is scheduled.

P1	**P2**	**P3**	**P4**	**P5**

`wait(S);`
`print "A";`
`signal(S);`
	`wait(S);`
	`print "B";`
	`wait(S);`
		`signal(S);`
		`print "A";`
		`wait(S);`
			`wait(S);`
			`print "B";`
			`print "A";`
				`signal(S);`
				`signal(S);`

The processes are scheduled in the following order: P2, P3, P1, P2, P3, P4, P1, P3, P4, P5, P1, P3, P5, P1, P3, P4. Explain clearly the operations carried out by each process, the contents printed and the value of S in each step. **[16]**

Ans.

(1) P2 is scheduled, executes wait(S), S=3, prints B, executes wait(S), S=2.

(2) P3 is scheduled, executes signal(S), S=3, prints A, executes wait(S), S=2.

(3) P1 is scheduled, executes wait(S), S=1, prints A, executes signal(S), S=2.

(4) P2 is scheduled, executes wait(S), S=1, prints B, executes wait(S), S=0.

(5) P3 is scheduled, executes signal(S), S=1, prints A executes wait(S), S=0.

(6) P4 is scheduled, executes wait(S), S=-1, P4 is added to S→List and P4 blocks.

(7) P1 is scheduled, executes wait(S), S=-2, P1 is added to S→List and P1 blocks.

(8) P3 is scheduled, executes signal(S), S=-1, P3 removes P4 from S→List, P3 wakes up P4, P3 prints A, P3 executes wait(S), S=-2, P3 is added to S→List, and P3 blocks

(9) P4 is scheduled, executes/restarts after being wooken up prints B, prints A.

(10) P5 is scheduled, executes signal(S), S=-1, P5 removes P1 from S→List and wakes up P1, P5 executes signal(S), S=0, P5 removes P3 from S→List and wakes up P3.

(11) P1 is scheduled, restarts after being woken up, prints A, executes signal(S), S=1.

(12) P3 is scheduled, restarts after being woken up, nothing to execute.

(13) P5 is scheduled, executes signal(S), S=2, executes signal(S), S=3.

(14) P1 is scheduled, executes wait(S), S=2, prints A, executes signal(S), S=3.

(15) P3 is scheduled, executes signal(S), S=4, prints A, executes wait(S), S=3.

(16) P4 is scheduled, executes wait(S), S=2, prints A, prints B.

Marking Scheme: 1 mark for each point. If semaphore values are not mentioned, only 8 marks will be awarded. If the content that gets printed is not mentioned, only 8 marks will be awarded.

8.a) Consider the two POSIX compliant C programs, **CODE I** and **CODE II**, given below. Assume all relevant header files are included and all function calls execute successfully. Write your observations regarding the execution and the output of the programs on an Ubuntu system. You should also mention the type of process/processes created by each code. Your answer should contain proper explanations. Assume that a child process always exits normally. Assume a multiprocessor environment. Your answer should correspond to the appropriate code. Ambiguous answers will not be considered. **[8]**

```
int main()
{
   if(fork()){
      sleep(4);
      printf("HELLO");
      int status = 5;
      wait(&status);
      printf("\n%d", status);
   }
   return 0;
}
```

CODE I

```
int main()
{
   if(fork() == 0){
      sleep(4);
      printf("HELLO");
      int status = 5;
      wait(&status);
      printf("\n%d", status);
   }
   return 0;
}
```

CODE II

Ans.

CODE I: The if statement evaluates to true for the parent process. The parent then sleeps for 4 secs and after that prints HELLO. The wait() call is meant for the parent to wait for the child process. But the child has terminated by then. Hence, the wait() call returns immediately and assuming that the child process exited normally, status = 0. So, 0 is printed on the console. The child terminates before the parent and till the time the parent calls wait(), the child becomes a zombie process. When the parent calls wait(), the information of the child is removed from the process table.

Marking Scheme: 4 marks for CODE I. 2 marks for the output/behavior and 2 marks for the explanation. If answer does not contain the point that the child becomes a zombie process, 1 mark will be deducted.

CODE II: The if statement evaluates to true for the child process. The child then sleeps for 4 secs and after that prints HELLO. The wait() call is meant for the parent to wait for the child process. But the child does not need to wait for any process. Hence, the wait() call returns immediately and status retains its initial value, status = 5. So, 5 is printed on the console. The parent terminates before the child and the child becomes an orphan process.

Marking Scheme: 4 marks for CODE I. 2 marks for the output/behavior and 2 marks for the explanation. If answer does not contain the point that the child becomes an orphan process, 1 mark will be deducted.

8.b) Explain how Peterson's solution satisfies the bounded waiting requirement. **[4]**

Ans. Suppose Pj is executing in CS and Pi wants to enter CS. Thus, flag[j] = true, flag[i] = true and turn = j (set by Pi in its entry section). Pi gets stuck in the while loop. When Pj exits CS, Pj sets flag[j] = false and Pi can immediately enter because Pi does not change turn in the while loop. So, even if Pj immediately wants to enter CS again and sets flag[j] = true and turn = i before Pi can perform the next while loop iteration, still Pi will break out of the while loop as turn = i and Pj will be stuck in the while loop as flag[i] = true and turn = i. So, only after one entry by Pj, Pi can enter CS. Hence, bounded waiting is satisfied.

Marking Scheme: If the highlighted part is not written, only 2 marks will be awarded.

****