**FACULTY OF ENGINEERING**

**SCHOOL OF COMPUTING AND IT**

**B. Tech (Computer Science & Engineering) IV Semester**

**I Sessional Examination: 2018-19**

**CS1401 – Operating Systems**

**(CLOSED BOOK)**

**Duration: 1 Hour Max. Marks: --15**

**Instructions:**

* Answer any five full questions
* Missing data, if any, may be assumed suitably.

|  |  |  |
| --- | --- | --- |
|  | Differentiate between System call and Interrupt  Interrupt   * An interrupt (or Hardware interrupt) is a hardware-generated change-of-flow within the system (comes from I/O devices). * At the end of each machine cycle, CPU checks INTR pin, whether there is any interrupt or not. CPU changes its mode from ordinary to privileged in case of any interrupt and control is transferred to operating system. * An interrupt can be used to signal the completion of an I/O to obviate the need for device polling.   System call   * System call is a software-generated interrupt (comes in user program). * CPU recognizes S\W interrupt when it executes software interrupt instruction and changes its mode from ordinary to privileged and control is transferred to operating system. * A trap can be used to call operating system routines or to catch arithmetic errors. | [2] |
|  | How many processes will be spawned after executing the code segment given below? Draw also the corresponding process tree.  *if (fork() || fork() )*  *if (fork() && fork())*  *fork();*  1  3 1 2 2  3 1 2 2 | [2] |
|  | Write a program using pipe in which child process writes a filename into the pipe and parent process open that file, counts the number of characters in the file and prints the count on the monitor.  #include <fcntl.h>  main()  {  int pfd[2], n, fd, count;  char buff[512];  pipe (pfd);  id = fork();  if (id==0){  close (pfd[0]);  n= read (0,buff,512); // reading filename from user  write (pfd[1], buff, n); // writing filename in pipe  }else{  close(pfd[1]);  n=read (pfd[0]. buff, 512); // reading filename from pipe  n--;  buff[n]=0;  fd= open (buff, O\_RDONLY);  count = lseek (fd, 0, 2);  printf (“%d\n”, count);  } | [3] |
|  | 1. Compare user-level threads and kernel-level threads.  |  |  | | --- | --- | | **User level thread** | **Kernel level thread** | | User threads are supported above the kernel and are implemented by a *thread library* at the user level. The library provides support for thread creation, scheduling, and management with no support from the kernel. | Kernel threads are supported directly by the operating system.  The kernel performs thread creation, scheduling, and management in kernel space. | | When threads are managed in user space, each process needs its own private thread table to keep track of the threads in that process | No run-time system is needed in each. Also, there is no thread table in each process. Instead, the kernel has a thread table that keeps track of all the threads in the system. | | User-level  *threads requires non-blocking systems call*, that means a multithreaded kernel. Otherwise, entire process will blocked in the kernel, even if there are runnable threads left in the processes. For example, if one thread causes a page fault, the process blocks. | Kernel threads do not require any new, non-blocking system calls. If one thread in a process causes a page fault, the kernel can easily check to see if the process has any other runnable threads, and if so, run one of them while waiting for the required page to be brought in from the disk. | | User-level threads are generally fast to create and manage | The kernel-level threads are slow and inefficient. For instance, threads operations are hundreds of times slower than that of user-level threads. | | User level thread is generic and can run on any operating system. | Kernel level thread is specific to the operating system. | | *Example:* User-thread libraries include POSIX Pthreads, Mach C-threads, and Solaris 2 UI-threads. | *Example:* Windows NT, Windows 2000, Solaris 2, BeOS, and Tru64 UNIX (formerly Digital UNIX)-support kernel threads. |  1. Write a multithreaded program in ‘C’ (using *PTHREAD*) in which the child thread will display prime numbers less than equal to the number entered by the user through command line.   #include <stdio.h>  #include <pthread.h>  void \*prime(void \*param);  void main(int c,char \*argv[])  {  pthread\_t tid;  pthread\_attr\_t attr;  int x,y;  if (c!=2)  {  fprintf(stderr,"error");  return ;  }  if (atoi(argv[1]) < 0 )  {  fprintf(stderr,"No. must be greater than 0");  return;  }  pthread\_attr\_init(&attr);  pthread\_create(&tid,&attr,prime,argv[1]);  pthread\_join(tid,NULL);  }  void \*prime(void \*param)  {  int i,j,flag,upper=atoi(param);  for ( i=1; i<=upper; i++ )  {  flag=1;  for ( j=2;j<=i / 2; j++ )  {  if (i%j==0)  {  flag=0;  break;  }  }  if (flag==0)  printf("%d ",i);  }  pthread\_exit(0);  } | [3] |
|  | Consider the following set of processes, with the length of the CPU burst time given in milliseconds:  **Process Burst Time Arrival Time**  *P1*  4 0  *P2*  2 1  *P3*  7 2  *P4*  5 3   1. Draw Gantt charts that illustrate the execution of these processes using Round Robin (RR) with time quantum equal to 1ms and Preemptive SJF CPU scheduling algorithms. 2. What is the turnaround time of each process for RR and Preemptive SJF CPU scheduling algorithms? 3. What is the waiting time of each process for RR and Preemptive SJF CPU scheduling algorithms?   Ans.  **Round Robin**   |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | P1 | P2 | P1 | P3 | P2 | P4 | P1 | P3 | P4 | P1 | P3 | P4 | P3 | P4 | P3 | P4 | P3 | P3 |   **SJF**   |  |  |  |  |  | | --- | --- | --- | --- | --- | | P1 | P2 | P1 | P4 | P3 |   **5-b, c**   |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Process** | **Round Robin** | | **Shortest Job First** | | | **Turnaround Time** | **Waiting Time** | **Turnaround Time** | **Waiting Time** | | P1 | 10 | 6 | 6 | 2 | | P2 | 4 | 2 | 2 | 0 | | P3 | 16 | 9 | 16 | 9 | | P4 | 13 | 8 | 8 | 3 | | [5] |