Student ID: ______________________________

Student Name: __________________________________

Dhirubhai Ambani Institute of Information & Communication Technology
Second In-Semester Examination, Autumn Semester 2022-2023

Course Title: IT227 Computer Systems Programming **Max Marks: 55**

Date: 9th Dec 2022 **Time: 2 Hours**

Q1a(2)	Q1b(2)	Q1c(2)	Q1d(2)	Q1e(2)	Q1f(2)	Q1g(2)	Q1h(2)	Q1i(2)	Q1j(2)	Q2a(5)	Q2b(5)
Q2c(5)	**Q2d(5)**	**Q3(10)**	**Q4(5)**								

Instructions:

1. All questions are compulsory.

2. Write the answers for all questions in the space provided in the question paper only.

3. Write your answers in brief; writing long answers does not fetch higher marks.

Q1. Short Answer 1-3 lines **[2 Marks each]**

a. **We are using MKDEV(250, 0) to statically create device major number as 250. How will you check that this major number is not utilized by any other device?**

Answer:

Either check in /proc/devices file or in /dev directory

b. **While performing getaddrinfo() on server side, why do we use AI_PASSIVE and AI_NUMERICSERV?**

```
hints.ai_socktype = SOCK_STREAM;
 hints.ai_flags = AI_PASSIVE | AI_NUMERICSERV;
getaddrinfo(NULL, port, &hints, &listp);
```

Answer:

AI_PASSIVE → By default socket connections are active used on client side to initiate the connection request but for server side we need socket in passive mode to listen to client request, hence AI_PASSIVE allow us to change socket from active to passive.

AI_NUMERICSERV → Allow us to provide port number instead or service name, e.g. port 80 instead of "http" service.

c. **Is there a problem you see with the below code? If yes, suggest changes to fix the code.**

```
void *thread_fn(void *) {
   // important task this thread performs that takes atleast 10 mins and must be completed
}
main () {
   int tid;
   pthread_create(&tid, NULL, thread_fn, NULL);
   exit(0);
}
```
Answer:

Option1:
```
main () {
   int tid;
   pthread_create(&tid, NULL, thread_fn, NULL);
   pthread_join(tid, NULL)
   exit(0);
}
```
Option2:
```
main () {
   int tid;
   pthread_create(&tid, NULL, thread_fn, NULL);
   pthread_exit(0);
}
```

d. **Match the systems calls on the left to its functionality on the right.**

A. exit()	**1. terminates current thread but lets the child threads continue to run**
B. return	**2. parent thread must wait for child thread to terminate**
C. pthread_exit()	**3. terminates the process terminating all threads**
D. pthread_join()	**4. terminates the thread function and all child threads**

Answer:

A – 3, B-4, C-1, D-2

e. **Explain the significance of inode, cdev and file_operations stuctures in relation to creating character device driver.**

Answer:

file_operation structure contains function pointers for the support operations on the character device.
cdev structure contains reference to the file operation, driver code's kernel object, major/minor number and owner of the character device.
inode structure contains the cdev structure reference as a member indicating a particular character device.

f. **Assume parent process has pid=10, pgid=10 and child has pid=11, pgid=11. If the following kill system call is issued by the parent process, which of the process(es) will be terminated. Explain why?**
kill(-10, SIGKILL);

Answer:

Only parent process will be killed because the child process is in a different process group

g. **If a user runs a following command on a file that has 444 permission, what will happen?**
rm file1.txt
Answer:
User is prompted for confirmation. if user provides yes to remove the file then it will be deleted.

h. **Why do we need to use copy_to_user() and copy_from_user() functions in the driver code while copying data from kernel buffer to user buffer and vice a versa?**
Answer:
Because driver code need to switch to kernel model while copying data to/from kernel buffer.

i. **Why most thread implementation is kernel level threads (KLT) and not user level threads?**
Answer:
Because in KTL kernel is aware of threads allowing it to have fine grain scheduling. That means different threads can be allocated to different cores/threads.

j. **What is the return value ($?) of the following shell script? Why?**

```
s = System
[$sProgramming = SystemProgramming] && exit 10
[${s}Programming = SystemProgramming] && exit 20
```

Answer:
20 – because ${s} will be replaced with System

Q2. There is NO PARTIAL MARKING for this question.
For each of the questions below write the output generated by the code and explain briefly the reason for this output. Just writing output will not fetch any marks. **[5 Marks Each]**

a. **Assume that file.txt contains 6 characters shown below.**
A1B2C3

```
int main() {
   int fd1, fd2; char c;
   fd1 = open("file.txt", O_RDONLY);
   fd2 = open("file.txt", O_RDONLY);
   read(fd1, &c, 1);
   dup2(fd2, fd1);
   read(fd1, &c, 1);
   printf("c = %c", c);
}
```

Answer:
c=A
It will print the first character of the file. After the first read fd1 will be forwarded by one byte but dup2 function will make fd1 same as fd2 which means it will again move back to the beginning of the file. Hence the 2nd read will again read the first byte itself.

b. **Considering the code below what will be the output generated?**

```
#define NUM_THREADS 4
char array[NUM_THREADS+2];
int pos = 0;
char outs[9] ={'8','7','6','5','4','3','2','1','0'};
void* work(void* id){
        int index = (*((int*)id))*2;
        char writeMe = outs[index];
        array[pos++] = writeMe;
}
```

```
int main(){
        char three = '3';
        int i;
        pthread_t tid[NUM_THREADS];
        for(i = 0; i<NUM_THREADS; i++){
                pthread_create(&(tid[i]), NULL,
work, (void*)(&i));
                pthread_join(tid[i], NULL);
        }
        array[pos++] = three;
        array[pos] = '\0';
        printf("%s", array);
        exit(0);
}
```

Answer:

18243

Because each thread share global variable value of pos will be incremented from previous value each time a new thread is created and work() is called. For thread1, i=0, index=i*2=0 and pos=0, writeMe=outs[index]='8' and array[pos]=writeMe='8', then for thread 2 i=1, index=i*2=2 and pos=1, writeMe=outs[index]='6' and array[pos]=writeMe='6', so we can see that array will values {'8', '6', '4','2'} for the 4 threads that are created and finally main thread will add last '3'. Final array will be {'8', '6', '4','2', '3'} and so will print string "86423".

c. **When the below shell script is executed, what will be the content of the sorted.txt when the input file contains following data?**

#! /bin/bash	**content of the input file:**
prev=	
cat $1 | sort |	**1**
while read line	**1 2**
do	**1 2 3**
if [[$prev != $line]]	**1 2 3 4**
then	**1 2 3**
echo $line >> sorted.txt	**1 2**
fi	**1**
prev=$line	
done	

Answer:

Removes duplicate lines

1

1 2

1 2 3

1 2 3 4

d. You may assume that printf is unbuffered and executes atomically. The program /bin/echo prints its command line argument to stdout

```
pid_t pid;
int counter = 0;
void handler1(int sig)
{
        counter++;
        printf("counter = %d\n", counter);
        fflush(stdout);
        kill(pid, SIGUSR1);
}
void handler2(int sig)
{
        counter += 3;
        printf("counter = %d\n", counter);
        exit(0);
}
```

```
int main()
{
        pid_t p;
        int status;
        signal(SIGUSR1, handler1);
        if ((pid = fork()) == 0)
        {
                signal(SIGUSR1, handler2);
                kill(getppid(), SIGUSR1);
                while(1) ;
        }
        if ((p = wait(&status)) > 0)
        {
                counter += 4;
                printf("counter = %d\n",
counter);
        }
}
```

Answer:
counter = 1 (parent process)
counter = 3 (child process)
counter = 5 (main process)

Q3. A web server is running on port 20000 that is supporting maximum of 500 client connections at a time as shown in the server code below. The requirement has changed to support 2000 client connections that connect to this server. Obviously we do not want to run multiple instances of the same server because it would take more resources instead we want to modify this server code so that the server runs on 4 different ports each supporting 500 client connections with the total of 2000 client connections simultaneously. Please note that server should be running on all 4 ports at the same time using the same server code to support 2000 client connections. Rewrite the server code from below to support this new requirement. [10 Marks]

```
main()
{

 /* Run server on port 20000 */
 getaddrinfo(NULL, 20000, &hints, &listp);
 listenfd = socket(listp->ai_family, listp->ai_socktype, listp->ai_protocol);
```

```
  bind(listenfd, listp->ai_addr, listp->ai_addrlen);
  /* Maximum 500 connections allowed on this port*/
  listen(listenfd, 500);
  while(1)
  {
    connfd = accept(listenfd, (struct sockaddr *)&clientaddr, &clientlen);
    /* Each client connection is handled by a new thread using handle_client function */
    pthread_create(&tid, null, handle_client, connfd);
  }
}
```

Answer:

```
handle_server(listenfd)
{
  while(1)
  {
    connfd = accept(listenfd, (struct sockaddr *)&clientaddr, &clientlen);
    /* Each client connection is handled by a new thread using handle_client function */
    pthread_create(&tid, null, handle_client, connfd);
  }
}
main()
{

  for (port=15000; port<=15003; port++)
  {
              /* Run server on port */
              getaddrinfo(NULL, port, &hints, &listp);
              listenfd = socket(listp->ai_family, listp->ai_socktype, listp->ai_protocol);
              bind(listenfd, listp->ai_addr, listp->ai_addrlen);
              /* Maximum 500 connections allowed on this port*/
              listen(listenfd, 500);
              pthread_create(&tid, null, handle_server, listenfd);
  }
}
```

Q4. Consider a process file descriptor table as studied in file IO that is maintained for each process by operating system. A process can have multiple threads which can create a new file descriptor using open or dup system calls that will require adding an entry in process file descriptor table. There may be multiple threads that reads the file descriptors from process file descriptor table for reading and writing from files. Below are the two partial implementation of functions add_process_file_descriptor() and read_process_file_descriptor(). Implement the code in such a way that only thread is able to update the

process file descriptor table and multiple threads are able to read process file descriptor table and reading and writing must be mutually exclusive. [5 Marks]

```
#define MAX_PROCESSES 1024
#define MAX_FDS 20
int process_fds[MAX_PROCESSES][MAX_FDS];
int index=0;
int main() {

_______pthread_rwlock_t rw_lock;_______________________________________________________
ret = pthread_rwlock_init(&rw_lock, NULL);

_______________________________________________________________________________________

_______________________________________________________________________________________

_______________________________________________________________________________________

ret = pthread_rwlock_destroy(&rw_lock);
}

void add_process_file_descriptor(int pid, int fd) {
pthread_rwlock_wrlock(&rw_lock);
_______________________________________________________________________________________

_______________________________________________________________________________________

_______________________________________________________________________________________
process_fds[pid][index++] = fd;
_______________________________________________________________________________________

_______________________________________________________________________________________

_______________________________________________________________________________________
pthread_rwlock_unlock(&rw_lock);
}

int read_process_file_descriptor(int pid, int ind) {
pthread_rwlock_rdlock(&rw_lock);
_______________________________________________________________________________________

_______________________________________________________________________________________
int fd = process_fds[pid][ind];

pthread_rwlock_unlock(&rw_lock);

return fd;
}
```