Student ID: _____

Student Name: _____

Dhirubhai Ambani Institute of Information & Communication Technology
End Semester Examination, Winter Semester 2021-2022

Course Title	IT215 System Software	**Max Marks**	74
Date	6-May-2022	**Time**	120 mins

Instructions:

- **All questions are compulsory.**
- **The answer to each question must be written in the space provided after the question in the question paper only (No need for Answer Booklet).**
- **Write your answers in brief and with clarity. Writing long answers does not fetch a higher score.**

Q1 Concurrent Programming: **[16 Marks]**

Consider the code1, code2, code3 and code4 below. For each of these codes, what are all possible counter variables values that can be printed as the first value on stdout? Briefly explain your answer.

// code1

```
int counter = 2;
void foo() {
counter++;
  printf("%d", counter);
}
int main() {
  pthread_t tid[2];
  int i;
  for (i = 0; i < 2; i++) {
    pthread_create(&tid[i], 0, foo, 0);
  }
  counter++;
  printf("%d", counter);
}
```

// code2

```
int counter = 2;
void foo() {
counter++;
  printf("%d", counter);
}
int main() {
  pthread_t tid[2];
  int i;
  for (i = 0; i < 2; i++) {
    pthread_create(&tid[i], 0, foo, 0);
    pthread_join(tid[i], 0);
  }
  counter++;
  printf("%d", counter);
}
```

// code3

```
int counter = 2;
void foo() {
  pthread_mutex_lock(&m);
  counter++;
  printf("%d", counter);
  pthread_mutex_unlock(&m);
```

// code4

```
int counter = 2;
void foo() {
counter++;
  printf("%d", counter);
}
int main() {
```

```
}
int main() {
  pthread_t tid[2];
  int i;
  pthread_mutex_t m;
  ret=pthread_mutex_init(&m, NULL);
  for (i = 0; i < 2; i++) {
  pthread_create(&tid[i], 0, foo, 0);
  }
  counter++;
  printf("%d", counter);
  ret=pthread_mutex_destroy(&m);
}
```

```
  pthread_t tid[2];
  int i;
  for (i = 0; i < 2; i++) {
    if (fork() == 0) {
    foo();
    }
  }
  counter++;
  printf("%d", counter);
}
```

Answer For Code1: 3 and 4 ,5.

Explanation:

If parent process executes counter++ and printf statement before any child thread, when counter will be 3 and therefore 3 will be printed.

If two of threads increment counter before executing printf, then counter will become 4 and thus 4 will be printed.

If all of three threads increment counter before any thread executes printf statement, then counter is increased by 3 and therefore 5 will be printed.

Answer For Code2:
3
Explanation: Here we are using pthread_join, therefore the first printf will be executed by the first created thread. Until this thread is not finished, nothing will be done.Therefore , the first thread will print 3 after incrementing counter.

Answer For Code3:
3 or 4.

Explanation: If main thread and one of child thread executes counter++, then counter will become 4, and thus 4 is printed.
If only main thread or one of child thread executes counter++ before any printf , then counter will be 3 and thus 3 is printed.

5 is not possible as two child threads can't increment counter before any printf, as there is mutex lock.

Answer For Code4:3

Explanation: As we are doing fork() here, all the processes that will be created will have its own copy of counter , so it is initially 2 in all processes. In all processes the first operation done is counter++ and printf, therefore the first printed value from any of these processes is 3.

Q2 Synchronization with Concurrent Programming

a. Consider the thread_fn function as shown below. Assume that two threads are allowed to call the function, which is controlling the execution of the critical section part of the code using a shared variable. Do you see any problem with this code? Briefly explain the reason. If you see a problem, rewrite the code to fix it. **[2 Marks]**

```
int shared=0;
void *thread_fn(void *arg) {
  if (shared == 0)
    shared = 1;
  else return;
  // critical section code goes here
  shared = 0;
}
```

Answer:

Problem: Race condition may occur when both the threads try to check the if condition at the same time. Therefore, both will update shared to 1 and enters the critical section. This should not be allowed.

→**Solution:** use mutex lock

```
pthread_mutex_t lock;
void *thread_fn(void *arf) {
        pthread_mutex_lock(&lock);
        // critical section code goes here
        pthread_mutex_unlock(&lock);
}
```

b. For each situation, state the one primitive that, when used correctly around the relevant critical section, prevents race conditions and results in the most concurrency. When more than one primitive will work with equal concurrency, give the primitive that is simplest, as defined below. If your answer is a semaphore, you must specify its initial value.

Your response to each question will be exactly one of the following primitives, listed

here in order from simplest to most complex: none needed, mutex, semaphore(n), rwlock.

You may assume that all relevant information has been given to you. For example, if it is not explicitly stated that a thread writes to a variable, then there is no possible race condition involving writes. No additional logic or variables may be added to the programs; you are only wrapping critical sections with concurrency primitives.
Consider the following situations: **[5 Marks]**

Situation:	**Answer:**
A. Two threads read from a global variable.	Not needed
B. Two threads increment a global counter.	Read write lock
C. Two hundred threads search through a regular linked list of integers.	None
D. Two hundred threads search through a regular linked list of integers; one thread occasionally removes and frees nodes from the list.	Mutex
E. At most seven threads may be within the critical section simultaneously.	Semaphore(num)
F. One thread waits, blocked, for events that may occur at any time and are inserted into a queue when they do occur; it is unacceptable for any event to be missed.	Mutex
G. The operating system maintains the process table which can be read by several threads but only the main kernel thread is allowed to create a new process or remove the finished process in the process table.	Mutex
H. Your system has four USB ports that need to be shared by a maximum of four threads.	Semaphore(num)
I. In the producer-consumer problem, a producer generated an item and places it in the circular queue if the queue is not full, otherwise, it waits for the queue to have at least one free space. Assume that queue can have a maximum of 100 items.	Semaphore(num)
J. In the producer-consumer problem, a consumer consumes an item from a circular queue if the queue has at least one item, otherwise, it waits for the queue to have at least one item. Assume that queue can have a maximum of 100 items.	Semaphore(num)

c. Let us consider a 1GB of computer memory is divided into 1024 blocks, each with 1MB size. You are writing an operating system task that will track the occupied and free blocks. The two functions occupy_block is called when a process requests to occupy the available (free) memory block to the kernel and free_up_block is called when a process wants to free up the already occupied block. Both occupy_block and free_up_block must run atomically to ensure no two process requests are handled at the same time. Assume that initially, all the blocks are free. Partial code is provided; please fill in the required code in the space provided. **[10 Marks]**

```
#include <pthread.h>
#include <semaphore.h>
int blocks[1024] = {0};
int read_index=0, write_index=0;
sem_t occupied, free;
pthread_mutex_t mutex;

int main() {
  pthread_t tid;

_____

       sem_init(occupied);
_____
       sem_init(free);
_____

       pthread_mutex_init(mutex);
_____


  pthread_create(&tid, NULL, free_up_block, NULL);
pthread_create(&tid,  NULL,  occupy_block,  NULL);
return;
}

/* occupy_block thread */
void* occupy_block(void* start_block_addr) {
  while(1){

_____

    sem_wait(&occupied);
_____
       sem_wait(&free);
_____

_____
```

```
    block[write_index] = (int *)start_block_addr;
    write_index = (write_index + 1) % 1024;
```

```
sem_post(&occupied);
```

```
        sem_post(&free);
```

```
  sleep(rand()%5);    /* wait for up to 5 sec */
  }
  return NULL;
}

/* free_up_block thread */
void * free_up_block(void* vargp) {
  while(1){
```

```
sem_post(&occupied);
        sem_post(&free);
```

```
    int *start_block_addr = block[read_index];
    read_index = (read_index + 1) % 1024;
```

```
sem_post(&occupied);
```

```
        sem_post(&free);
```

```
  sleep(rand()%5);    /* wait for up to 5 sec */
    }
        return (void *)start_block_addr;
}
```

Q3 GCC Compilation and Makefiles

a. We have the following content in the makefile where myapp.c uses functions defined in abc.c and def.c from the dynamic library. You need to find all the errors in the makefile. Write the new makefile with all corrections. **[5 Marks]**

```
CC=gcc
%.o: %.c
        $(CC) -o $@ $^
libmylib.so: abc.o def.o
        ar rs $@ $<
myapp.out: myapp.o
        $(CC) -o $@ $^
```

Answer:

```
CC=gcc
%.o: %.c
        $(CC) –fPIC -c -o $@ $^
libmylib.so: abc.o def.o
        $(CC) –shared -o $@ $^
myapp.out: myapp.o libmylib.so
        $(CC) -o $@ $^
.phony:all
all:
    myapp.out libmylib.so
```

b. Let us say you are building libraries for an application used in a time-sensitive (real-time) environment. Which type of libraries, static or dynamic, will you provide? Why? **[2 Marks]**

Answer:

Static library has less run time than dynamic library. Because in dynamic library actual linking happens when the program is run so it took more time than static library.

c. Match tool (1,2,3,4) on the left side to its functionality (A,B,C,D) on the right from the table below. **[2 Marks]**

1. ldd	A. Links multiple object and library files to generate the final executable
2. ld	B. Converts source code to assembly code
3. cc	C. Converts assembly code to object code
4. as	D. Used to check unreachable libraries

Answer:

1->D
2->A
3->B
4->C

Q4 Socket Programming

a. Let's assume that we have a server computer with 2 ethernet connections on 2 different network interface cards, therefore each connection having different IPv4 addresses (191.168.1.1, 191.168.1.2). We want to have 4 instances of server code running that will use two IP addresses each with 15000, 15001 port numbers. Each of the 4 server instances should support a maximum of 500 client connections. In other words, server 1 will run on 191.16.1.1:15000, server 2 on 191.16.1.1:15001, server 3 on 191.16.1.2:15000 and server 4 on 191.16.1.2:15001 each accepting maximum of 500 clients' connection. Write the server code using the template code provided below for this implementation. **[5 Marks]**

```
main()
{
   getaddrinfo(_____,_____,
&hints, &listp);
   _____= socket(listp->ai_family, listp->ai_socktype, listp-
>ai_protocol);
   bind(_____, listp->ai_addr, listp->ai_addrlen);
   listen(_____,_____);
}
```

Answer: (Additional space is available on the next page)

```
int open_listenfd(char *host,char *port)
{
   struct sockaddr_in *listp,*hints,*p;
   getaddrinfo(host,char,&listp,&hints);
   for(p=listp;p!=NULL;p=p->next)
   {
          int listen_fd=socket(/*given in question*/);
    if(listen_fd<0)
          continue;
     if(bind(listen_fd,/*given in qeustion*/)<0)
          continue;
}
 if(p==NULL)
          reutrn -1;
 if(listen(listen_fd,100)<0)
          return -1;
else return listen_fd;
}
```

```
int listen_fd1=open_listenfd('191.16.1.1',15000)
int listen_fd2=open_listenfd('191.16.1.1',15001)
int listen_fd3=open_listenfd('191.16.1.2',15000)
int listen_fd4=open_listenfd('191.16.1.2',15001)
```

b. Consider the server-side code of network communication using socket between server and client. Assume that there are multiple clients requesting for connection at a time. Assuming all system calls succeed and therefore the error handling code is never executed. **[5 Marks]**

```
#define BUFF_SIZE 512
#define SERVER_PORT 15213
char buffer[BUFF_SIZE];
int main(){
    int server_sock, recvSize;
    struct sockaddr_in serverAddr, clientAddr;
    /*ignore the SIGPIPE signal*/
    signal(SIGPIPE,SIG_IGN);
    /*open server_socket */
    if((server_sock = socket(AF_INET,SOCK_STREAM,IPPROTO_TCP))<0){
            exit(-1);
    }
    serverAddr.sin_addr.s_addr = htonl(INADDR_ANY);
    serverAddr.sin_port = htons(SERVER_PORT);
    serverAddr.sin_family = AF_INET;
    if(bind(server_sock,(struct sockaddr *)&serverAddr,sizeof(struct sockaddr)<0)){
        /*handle bind failing*/
                exit(-1);
    }
    if(listen(server_sock,15)<0){
        /*handle listen failing*/
                exit(-1);
    }
    while(1){
```

```
        int client_socket;
        size_t clientLen = sizeof(struct sockaddr);
        if((client_socket = accept(server_sock,(struct sockaddr
*)&clientAddr,&clientLen))<0){
            /*handle failing of accept*/
```

```
                        continue;
    }
    do{
        if((recvSize = recv(socket,buffer,BUFF_SIZE,0))<0){
            break;
              }
        if(send(socket,buffer,recvSize,0)<0){
            break;
              }
    }while(recvSize >0);
    /*once the code reaches this point, we have received 0 bytes from the recv* call*/
    close(socket);
}
```

There are two bugs in this code. Please locate the 2 logic bugs in this code and describe them. A logic bug is one where the programmer misunderstood how their program will execute and will produce unwanted behavior under certain input conditions. Show the code changes required to fix these bugs.

Answer:

Bug 1: After accepting a request from client, the communication with client is happening in the main thread itself. Due to this, server can't accept any other client requests until this client finishes.

Solution: To solve, this after one client request is accepted, a new thread should be created and any communication with this client should be done within the thread. This way server is free to accept other incoming requests.

Q5 File IO

Assume that the code below is executed as a program with both files having some data. For each row of the code section, provide details of changes in Process Open File

Tables and System-Wide Open File Table. Changes to both tables are shown as an example when line 3 is executed to open file1.txt and return fd1. Please make sure to copy the required information from the previous row to the next row before you update. The last row of tables must have complete information. **[10 Marks]**

Line	Code	Process Open File Tables	System-Wide Open File Table

1
 	
2
3
 	
4
	char wrt_msg[10]="abcdefghij"
	char read_msg[10];
	int fd1 = open("file1.txt", "O_RDWR");
	int fd2 = open("file2.txt", "O_RDWR");
		Index | **SysFD Ptr**
		1 | fd1=10
		2 | Fd2=11
		3 |
		4 |
			SysFD | **Offset** | **RefCnt** | **inode Ptr**
			10 | 0 | 1 | 1000
			11 | 0 | 1 | 1004

5
6
7
8
9
	write(fd1, wrt_msg,
	strlen(wrt_msg));
	lseek(fd1, -2, SEEK_END);
	read(fd2, wrt_msg, 2);
	write(fd2, read_msg, 3);
		Index | **SysFD Ptr**
		1 | Fd1=10
		2 | Fd2=11
		3 |
		4 |
			SysFD | **Offset** | **RefCnt** | **inode Ptr**
			10 | 8 | 1 | 1000
			11 | 5 | 1 | 1004

10
11
12
13
14
15
	if (fork() > 0)
	{
	int fd3 = dup(fd1);
	read(fd3, read_msg, 5);
	int fd4 = dup(fd2);
	}
		Parent Process
		Index | **SysFD Ptr**
		1 | Fd1=10
		2 | Fd2=11
		3 | Fd3=10
		4 | Fd4=11
			SysFD | **Offset** | **RefCnt** | **inode Ptr**
			10 | 13 | 2 | 1000
			11 | 5 | 2 | 1004

16
17
18
 	
19
20
	else
	{
	int fd4=open("file1.txt", "O_RDWR");
	read(fd4, read_msg, 5);
	}
		Child Process
		Index | **SysFD Ptr**
		1 | Fd1=10
		2 | Fd2=11
		3 |
		4 |
			SysFD | **Offset** | **RefCnt** | **inode Ptr**
			10 | 8 | 1 | 1000
			11 | 5 | 1 | 1004

	Show Content of Final Tables	**Child Process** **Index** | **SysFD Ptr** 1 | Fd1=10 2 | Fd2=11 3 | 4 | Fd4=12 **Parent Process** **Index** | **SysFD Ptr** 1 | Fd1=10 2 | Fd2=11 3 | Fd3=10 4 | Fd4=11	**SysFD** | **Offset** | **RefCnt** | **inode Ptr** 10 | 13 | 1 | 1000 11 | 5 | 1 | 1004 12 | 5 | 1 | 1000

Q6 Process Management, Signals

Consider the following two different C code. Assume all functions return without error, no signals are sent from other processes, and printf is atomic. **[5 Marks]**

// Code1:	**// Code2:**

int main() {
int pid = fork();
if(pid > 0) {
kill(pid, SIGKILL);
	int a = 1;
	void handler(int sig) {
	a = 0;
	}

```
    printf("a");
  } else {
    /* getppid() returns the pid of the parent
process */
    kill(getppid(), SIGKILL);
    printf("b");
  }
}
```

```
void emptyhandler(int sig) {
}
int main() {
  signal(SIGINT, handler);
  signal(SIGCONT, emptyhandler);
  int pid = fork();
  if(pid == 0) {
    while(a == 1)
    pause();
    printf("a");
  } else {
    kill(pid, SIGCONT);
    printf("b");
    kill(pid, SIGINT);
    printf("c");
  }
}
```

For each code snippet in the table below write a Y next to an outcome if it could occur, otherwise write N.

Answer:

Code1 Outcome	**Possible (Y/N)?**
Nothing is printed.	N
“a” is printed.	Y
“b” is printed.	N
“ab” is printed.	N
“ba” is printed.	N
A process does not terminate.	N

Code2 Outcome	**Possible (Y/N)?**
Nothing is printed.	N
“ba” is printed.	N
“abc” is printed.	N
“bac” is printed.	Y
“bca” is printed.	Y
A process does not terminate.	N

Q6 Device Driver

a. What is the purpose of using module_init() and module_exit() system calls? Write the shell command to execute module_init(myinit) but not module_exit(myexit). Please note that myinit() and module_init(myinit) are defined in mymodule.c driver code. **[2 Mark]**

Answer: (Additional space on next page is available)

Module_init:
It is used to set the constructor that will be executed when the module is loaded.
Eg: module_init(myinit) will set myinit function to be executed when module is loaded.
Module_exit:
It is used to set the action that should be executed when the module is removed.
Module_exit(myexit) will set the function myexit to be executed when module is removed.

Shell command to execute: insmod
Eg:$sudo insmod mydevice

b. What is the significance of file_operations structure when used as member in cdev structure as shown below? **[2 Mark]**

```
struct cdev {
    struct kobject kobj;
    struct module *owner;
    const struct file_operations *ops;
    struct list_head list;
    dev_t dev;
    unsigned int count;
};
```

Answer:

File_operations structure is used to set the actions performed for operations like read, write,open ,etc. The function that needs to be used for these operations are specified by this structured.
Eg: .read=myfunc
This line inside the file_operations structure will specify that myfunc is used to perform read operation.

c. What are the two ways we can identify whether a device is a character device or a block device? **[2 Mark]**

Answer:
The first letter in the file information of the device will contain 'c' if it is a character device, else it contains 'b' if it is a block device. Using this we can identify the device.
Eg:
ls –ltr /dev
crwxr_xr_x in file information of device means that this device is a character device.
brw-rw---- in file information of device means that this device is a block device.