CS 304 Operating Systems Mid-Semester Exam			Marks obtained ↓
Date: 21.03.2023,	Total questions: 5	Total Marks: 30	
Name:	Roll No:		Time: 120 min

Question:	1	2	3	4	5	Total
Points:	5	5	8	6	6	30
Score:						

Instructions:

1. **You are not allowed to carry any electronic gadgets including calculators and mobile phones.**
2. **Please note, merely writing the answer without justification/ reason (wherever explicitly asked to give reason) will not fetch FULL marks.**
3. **Answer to all the parts of a question needs to be written together and the final answer needs to be written with a pen.**

$\boxed{5}$ 1. Assume you have a system with three processes (A, B, and C) and a single CPU. Assume an MLFQ scheduler. Processes can be in one of five states: RUNNING, READY, BLOCKED, not yet created, or terminated. Given the following cumulative timeline of process behavior, indicate the state the specified process is in AFTER that step, and all preceding steps, have taken place.

(a) Step 1: Process A is loaded into memory and begins: it is the only user-level process in the system. Process A is in which state?

(b) Step 2: Process A calls fork() and creates Process B. Process B is scheduled. Process A is in which state? Process B is in which state?

(c) Step 3: The running process issues an I/O request to the disk. Process A is in which state? Process B is in which state?

(d) Step 4: The running process calls fork() and creates process C. Process C is not yet scheduled. Process A is in which state? Process B is in which state? Process C is in which state?

(e) Step 5: The time-slice of the running process expires. Process C is scheduled. Process A is in which state? Process B is in which state? Process C is in which state?

(f) Step 6: The previously issued I/O request completes; the process that issued that I/O request is scheduled.. Process A is in which state? Process B is in which state? Process C is in which state?

$\boxed{5}$ 2. Student X has come up with a new scheduling policy: Least Used First (LUF). When given the choice to schedule two processes (jobs) on the run queue, the scheduler will select the one that has used the fewest CPU cycles thus far. In case of a tie, the queue is FIFO. When a process is de-scheduled it goes to the end of the queue. The CPU should not sit idle when there are processes on the run queue. Recall that a job has an arrival time and a duration (the amount of CPU time it will need, in time units). Assume all the jobs are CPU-bound.

(a) Supposing a running job is only pre-empted when a new job arrives (no interrupts), fill the following table given below:

Job	Arrival Time	Duration	Turnaround Time	Response Time	Waiting Time
A	0	25			
B	15	25			
C	25	5			
D	40	5			

(b) Suppose the clock interrupts the CPU every 2 time units (i.e., at times 0, 2, 4, etc.) and job arrivals do not preempt the current job. If a job arrives at the same time as a clock interrupt, the job can run immediately. Fill the following table below :

Job	Arrival Time	Duration	Turnaround Time	Response Time	Waiting Time
A	0	25			
B	15	25			
C	25	5			
D	40	5			

(c) Can the LUF policy cause starvation? Explain your answer.

S 3. (a) Read the code given below. Assume the program `/bin/true`, when it runs, never prints anything and just returns 0 in all cases. Also, assuming all system calls succeed and printf() prints its outputs immediately, what outputs are possible?

```
int main(int argc, char *argv[]) {
   int rc = fork();
   if (rc == 0) {
      char *my_argv[] = { "/bin/true", NULL };
      execv(my_argv[0], my_argv);
      printf("1");
    } else if (rc > 0) {
       wait(NULL);
       printf("2");
    } else {
       printf("3");
    }
    return 0;
}
```

(b) The following code is given to you:

```
typedef struct __lock_t {
     int flag;
} lock_t;
void init(lock_t *lock) {
     lock->flag = FREE;
}
void acquire(lock_t *lock) {
    while(xchg(&lock->flag, HELD) == HELD)
            ; // spin-wait (do nothing)
}
void release(lock_t *lock) {
    lock->flag = 1;
}
```

What value should FREE be in init() and what should HELD be set to in acquire() for this code to work as a mutual exclusion primitive (i.e., a lock)? Explain your answer.

(c) Given a basic spin lock, assume that spin lock takes A time units (if no one else is holding the lock) unlock also takes A time units. Assume that a context switch takes C time units and a time slice is T time units long. Assume the code sequence given below , executed by two threads on one processor at roughly the same time.

```
    mutex_lock();
    do_something(); // takes no time to execute
    mutex_unlock();
    exit();         // takes no time to execute
```

(i) What is the best case time for the two threads on one CPU to finish this code sequence?

(ii) What is the worst case time for the two threads to finish this code sequence? Assume tha only three context switches can occur at a maximum.

(iii) If the spin lock above is changed to queue based lock, how does this change the worst cas time ?

(d) A concurrent program (with multiple threads) is given below. Assuming pthread create() and pthread join() all work as expected (i.e., they don't return an error), what are the possible outputs that this code will print ? Justify your answer.

```
volatile int counter = 1000;

void *worker(void *arg) {
     counter--;
     return NULL;
}
int main(int argc, char *argv[]) {
    pthread_t p1, p2;
    pthread_create(&p1, NULL, worker, NULL);
    pthread_create(&p2, NULL, worker, NULL);
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("%d\n", counter);
    return 0;
}
```

6 4. (a) Consider a table with five three-handed philosophers and a pile of N chopsticks in the middle of the table. Each philosopher needs 3 chopsticks to eat. What is the smallest N such that deadlock is impossible?

(b) Consider the following snapshot of a system in which four resources A, B, C and D are available. The system contains a total of 6 instances of A, 4 of resource B, 4 of resource C, 2 resource D. Do the following problems using the banker's algorithm:

	Allocation				*Max*				*Need*				*Available*			
	A	*B*	*C*	*D*	*A*	*B*	*C*	*D*	*A*	*B*	*C*	*D*	*A*	*B*	*C*	*D*
P_0	2	0	1	1	3	2	1	1	1	2	0	0	6	4	4	2
P_1	1	1	0	0	1	2	0	2	0	1	0	2				
P_2	1	0	1	0	3	2	1	0	2	2	0	0				
P_3	0	1	0	1	2	1	0	1	2	0	0	0				

(i) Compute what each process might still request and fill this in under the column Need.

(ii) Is the system in a safe state? Why or why not?

(iii) Is the system deadlocked? Why or why not?

(iv) If a request from P3 arrives for (2, 1, 0, 0) can the request be granted immediately?

6 5. (a) The code for handling producer/consumer problem discussed in the class is given below. There is something wrong in the given code (There are no syntax errors). What is it? Explain your answer.

```
// PRODUCER CODE //
 mutex_lock(&m);             //P1
 while (numfull == max) // P2
      cond_wait(&cond, &m); //P3
 do_fill(i);  //P4
 cond_signal(&cond); //P5
 mutex_unlock(&m); //P6
```

```
    // CONSUMER CODE //
    mutex_lock(&m);      //C1
    while (numfull == 0) //C2
          cond_wait(&cond, &m); //C3
    int tmp = do_get();   //C4
    cond_signal(&cond);   //C5
    mutex_unlock(&m);     //C6
```

(b) Consider a process with three threads A, B, and C. The default thread of the process receives multiple requests, and places them in a request queue that is accessible by all the three threads A, B, and C. For each request, we require that the request must first be processed by thread A, then B, then C, then B again, and finally by A before it can be removed and discarded from the queue. Thread A must read the next request from the queue only after it is finished with all the above steps of the previous one. Write down code for the functions run by the threads A, B, and C, to enable this synchronization. You can only worry about the synchronization logic and ignore the application specific processing done by the threads. You should use only **semaphores** to solve this. *Hint: Understand the order of processing of threads and use semaphores to indicate if the processing of that thread in the common request queue has been completed or not and implement synchronization among the threads and fill in code for Thread A() {}; Thread B() {} and Thread C() {}.*