1. Processes and threads

a) What is a process? What is a thread? How are they related? [5]

A process is a program in execution (1). A thread is a "thread of control" in the execution of a program (1). Each process has one or more threads (1). A thread shares the process address space, open files, and other resources with other threads (1). Each thread has its own program counter, CPU registers, and stack (1).

b) Explain the circumstances under which the line of code marked printf("LINE J") in the following code will be reached. (write your answer in the space below ↓) [5]

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

int main( {
   pid_t pid;

   pid = fork();
   if (pid < 0) {
      fprintf(stderr, "Fork Failed\n"); return 1;
   } else if (pid == 0) {
      execlp("/bin/ls", "ls", NULL);
      printf("LINE J");
   } else {
      wait(NULL); printf("Child Complete");
   }
   return 0;
}
```

The call to execlp() replaces the address space of the process with the program specified in the first parameter ("/bin/ls"). If the call to execlp() succeeds, the new program is now running and control from the call to execlp() never returns – thus, if the execlp() call is successful, the call to printf() never occurs. However, if an error occurs in the call to execlp() (e.g., "/bin/ls" does not exist, is not executable, …), execlp() returns an error indication, and the call to printf() would be performed.

c) Explain the difference between "at most once" and "exactly once" semantics for remote procedure call interprocess communication. [5]

At most once: each remote procedure call request is guaranteed to not be delivered to the server more than once; it is possible for a particular remote procedure call request to not be delivered at all. (2)
Exactly once: each remote procedure call is guaranteed to be delivered to the server exactly once. This requires a more complex implementation to make this guarantee (Acknowledgements). (2)
Exactly once is often constructed over an at most once implementation. (1)

Your initials _____

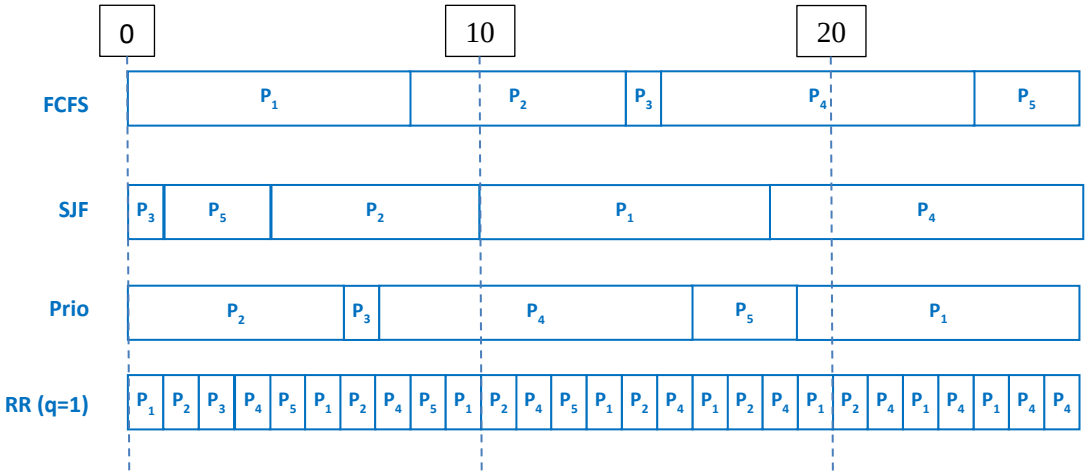
2. CPU Scheduling

a) Consider the following set of processes, with the CPU bursts given in milliseconds:

Process	Burst Time	Priority
P1	8	4
P2	6	1
P3	1	2
P4	9	2
P5	3	3

It is assumed below that all processes have arrived at time 0, and the initial ordering of the ready queue is P1, P2, P3, P4, P5.

(i) Draw four Gantt charts that illustrate the execution of these processes using the following scheduling algorithms: First Come First Served, Shortest Job First, Non-Preemptive Priority (a smaller priority number implies a higher priority) and Round Robin (quantum = 1). [16]



Your initials _____

(ii) For each schedule above, compute the average wait time and average turnaround time, completing the table below. Which scheduling algorithm yields the minimum average wait time? Minimum average turnaround time? [4]

Algorithm	Average Wait Time	Average Turnaround Time
FCFS	12.2	17.6
SJF	6.6	12.0
NonP Priority	9.6	15.0
RR	12.4	17.8

FCFS:

- ave wait time = (0+8+14+15+24)/5 = 61/5 = 12.2
- ave turnaround time = (8+14+15+24+27)/5 = 88/5 = 17.6

SJF:

- wait time = (0+1+4+10+18)/5 = 33/5 = 6.6
- turnaround time = (1+4+10+18+27)/5 = 60/5 = 12.0

NonP Priority:

- wait time = (0+6+7+16+19)/5 = 48/5 = 9.6
- turnaround = (6+7+16+19+27)/5 = 75/5 = 15.0.0

RR:

- wait time:
  - P1 = 0+4+3+3+2+2+2+1 = 17
  - P2 = 1+4+3+3+2+2 = 15
  - P3 = 2
  - P4 = 3+3+3+3+2+2+1+1 = 18
  - P5 = 4+3+3 = 10
  - ave wait time = (17+15+2+18+10)/5 = 62/5 = 12.4
- turnaround time = (25+21+3+27+13)/5 = 89/5 = 17.8

Min average wait time: SJF

Min average turnaround time: SJF

Your initials _____

3. Pthreads and Synchronization

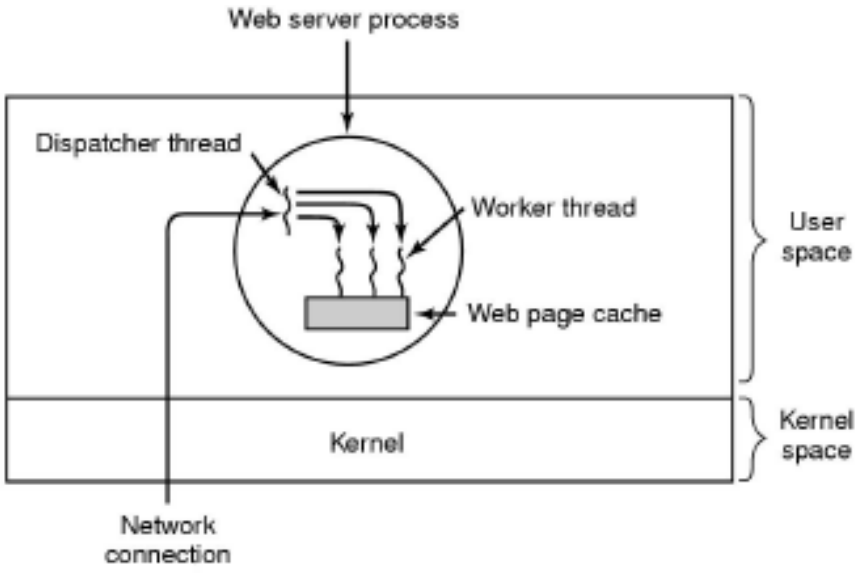
a) How does PThreads support the ability for threads to create critical regions around shared data? How does it support the creation of conditional critical regions around shared data? [6]

PThreads provides the abstraction of a mutex for creating critical regions; a mutex is used to achieve mutually exclusive access to shared data; each thread that wants to access data protected by a mutex must perform a lock(mutex)/unlock(mutex) sandwich around code that accesses the data.

To support conditional critical regions, PThreads provides the additional abstraction of a condition variable; a condition variable, when used together with an associated mutex, enables one thread to signal that an event has occurred that might be of interest to other threads; enables one to implement conditional critical regions based upon values of the shared state. after locking the associated mutex, code can test the shared state, and wait on the condition variable if it is not in the correct state to proceed.

For each type of critical region, 1 mark for name of mechanism, 2 marks for adequate description of abstraction and use.

b) One of the examples we discussed in lecture was a multi-threaded web server. In such an application, shown below, a dispatcher thread receives HTTP requests from the network, placing those requests in a queue; each worker thread obtains a request from the queue, reads the page from the disk, and returns the page to the requesting browser.



A multithreaded web server

Pseudo-code for the dispatcher function and the worker function are shown below. The queue behind the add_to_queue() and remove_from_queue() is unbounded; it is also ***not*** thread-safe; finally, remove_from_queue() returns NULL if there is nothing in the queue.

Since the queue is not thread-safe, the code as written is unsafe, since the dispatcher and worker threads may simultaneously attempt to modify the queue; additionally, the worker thread busy waits for a request.

Your initials _____

Sketch the changes needed to the code below to eliminate the unsafe race condition over the shared queue and to eliminate the busy wait loop in the worker function; do this using the PThreads support you have described in your answer to 1(a) above. [14]

Since it is unlikely that you remember the exact signatures for the PThread functions that provide the abstractions you described in 1(a) above, indicate immediately below the function signatures that you will be using in your modifications.

```
pthread_mutex_lock(&mutex);
pthread_mutex_unlock(&mutex);
pthread_cond_wait(&cond, &mutex);
pthread_cond_signal(&cond);
```

(sketch the changes on the code below)

```
in global data:
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

void *dispatcher(void *args) {
   struct request *buf;
   while(1) {
      get_next_request(&buf); /* thread blocks waiting for network request */
      pthread_mutex_lock(&mutex);
      add_to_queue(&buf);       /* append to queue */
      pthread_cond_signal(&cond);
      pthread_mutex_unlock(&mutex);
   }
   return NULL;
}

void *worker(void *args) {
   struct request *buf;
   struct page *pg;

   while(1) {
      pthread_mutex_lock(&mutex);
      while ((buf = remove_from_queue()) == NULL) /* remove request from queue */
         pthread_cond_wait(&cond, &mutex);
      pthread_mutex_unlock(&mutex);
      if (! obtain_page_from_cache(&buf, &pg)) {
         read_page_from_disk(&buf, &pg);
         add_page_to_cache(&buf, &pg);
      }
      send_page(&buf, &pg) ; /* return page to requester */
   }
}
```


Your initials _____

2 marks for declaration and initialization of mutex, 2 marks for declaration and initialization of cond variable, 2 marks for lock/unlock sandwich around add_to_queue(), 2 marks for signal() after add_to_queue(), 2 marks for lock/unlock sandwich around remove_from_queue(), 4 marks for while() cond_wait() to eliminate busy wait.

Your initials _____