4.8 (8 points) Which of the following components of program state are shared across threads in a multithreaded process.

(a) Register Values

(b) Heap Memory

(c) Global Variables

(d) Stack Memory

Heap Memory and Global Variables are shared between threads in a multithreaded process. Both the stack and registers are needed for executing the separate paths of code, and thus will not be shared.

4.11 (5 points) Is it possible to have concurrency but not parallelism? Explain.

Yes. Parallelism requires multiple CPU's, so that multiple processes/threads are being run simultaneously in real time. Concurrency only requires that more than one process/thread is in the process of being computed at a time, thus, multiple process/thread may share a single CPU (not parallel) and alternate turns executing code by following some scheduling algorithm.

4.17 (10 points) The program shown below uses the pthreads API. What would be the output from the program at LINE C and LINE P?

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int value = 0;
void * runner(void * param); /* the thread */

int main(int argc, char ** argv) {
        pid_t pid;
```

```
        pthread_t tid;
        pthread_attr_t attr;
        pid = fork();

        if (pid == 0) { /* child process */
                pthread_attr_init(&attr);
                pthread_create(&tid, &attr, runner, NULL);
                pthread_join(tid, NULL);
                printf("CHILD: value = %d\n", value); /* LINE C */

        } else if (pid > 0) { /* parent process */
                wait(NULL);
                printf("PARENT: value = %d\n", value); /* LINE P */
        }

        return 0;
}

void * runner(void * param) {
        value = 5;
        pthread_exit(0);
}
```

The parent process creates a new child process, this child process then creates a new thread. The parent and the child have entirely separate memory spaces, where the child and it's created thread share some memory as detailed in problem 4.8, this happens to include 'value'. The 'value' is only changed by the process created by the child, thus, as the child waits for this process to finish via pthread˙join, LINE C will output a value of 5 (set by the thread). However, this all happens outside of the parents memory, thus it will be unchanged. Therefore LINE P will output a value of 0.

6.14 (15 points) Consider the exponential average formula used to predict the length of the next CPU burst. What are the implications of assigning the values to the parameters used by the algorithm?

(a) $\alpha = 0$ and $\tau_0 = 100$ ms

(b) $\alpha = 0.99$ and $\tau_0 = 10$ ms

(a) $(\alpha = 0, \tau_0 = 100)$
$\tau_{n+1} = \alpha \times t_n + (1 - \alpha)\tau_n$
$\tau_{n+1} = 0 \times t_n + \tau_n$
$\tau_{n+1} = \tau_n$
Note that we have killed off t_i the actual length of the i^{th} CPU burst. Thus, the prediction will always be equivalent to the last prediction (τ_n). Therefore, this algorithm will always estimate a constant CPU burst time of 100ms regardless of process.

(b) $(\alpha = 0.99, \tau_0 = 10)$
$\tau_{n+1} = \alpha \times t_n + (1 - \alpha)\tau_n$
$\tau_{n+1} = 0.99 \times t_n + 0.01\tau_n$
Here we see the algorithm very heavily weighted in favor of t_n, the actual length of the last CPU burst. Thus, this algorithm will fluctuate easily. For example, if a process has many short CPU bursts, but then one long one, for the following burst estimate, the algorithm will mostly ignore the last estimate (which would have been short) and go with a long estimate, very close to the last CPU burst time (t_n), even though we would assume given the long term behavior of the process, a short burst would be more likely.

6.16 (30 points) Consider the following set of processes, with the length of the CPU burst given in milliseconds:

Process	P1	P2	P3	P4	P3
Burst Time	2	1	8	4	5
Priority	2	1	4	2	3

The processes are assumed to have arrived in the order P1, P2, P3, P4, P5, all at time 0.

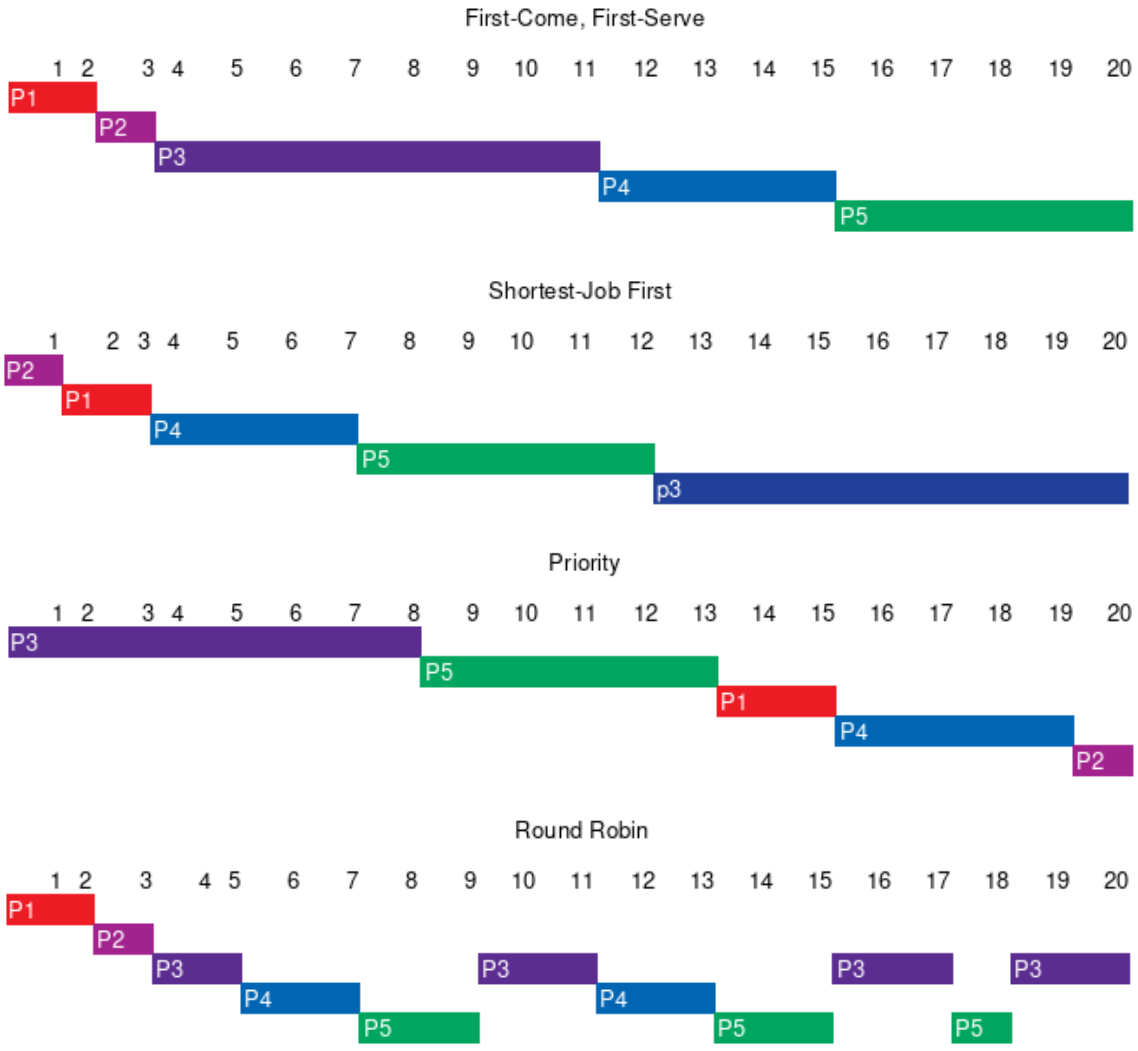
(a) Draw four Gantt charts that illustrate the execution of these processes using the following scheduling algorithms: FCFS, SJF, non-preemptive priority (a larger priority number implies a higher priority), and RR (quantum = 2).

(b) What is the turnaround time of each process for each of the scheduling algorithms in part a?

(c) What is the waiting time of each process for each of these scheduling algorithms.

(d) Which of the algorithms results in the minimum average waiting time (over all processes)?

(a)



(b)

	P1	P2	P3	P4	P5
First-Come, First Serve	2	1	8	4	5
Shortest-Job First	2	1	8	4	5
Priority	2	1	8	4	5
Round Robin	2	1	17	8	11

From the graphs, we can clearly see that Round Robin is the only method which interrupts execution of a process, thus for FCFS, SJF, and Priority, the turn around times are equivalent to the CPU burst time. Round Robin however has much longer turn around times as long process are evenly distributed over the entire execution time.

(c)

	P1	P2	P3	P4	P5
First-Come, First-Serve	0	2	3	11	15
Shortest-Job First	1	0	12	3	7
Priority	13	19	0	15	8
Round Robin	0	2	11.25	8	12.33

(d)

FCFS	Shortest-Job First	Priority	Round Robin
6.2	4.6	11	6.716

In the table of average wait times above (the sum of each row in part (c) divided by 5), we see that Shortest Job First has the smallest wait time, which is as expected. Leaving long jobs for the end minimizes the wait time for all processes and is the primary strength of the SJF algorithm.

6.19 (5 points) Which of the following scheduling algorithms could result in starvation?

(a) First-Come, First-Served

(b) Shortest Job First

(c) Round Robin

(d) Priority

Priority can, if you have a process with low priority, and new process constantly filter in with high priority as other process complete, these new processes will always take control of the CPU over the low priority process. Thus starvation occurs and the low priority process is never executed.

In a similar manner, Shortest Job First could also create starvation, where instead of a low priority process, we have a long job, then as short jobs are completed, more process are introduced with short CPU requirements. These short processes will continue to grab the CPU, and the long job process will be starved.

FCFS guarantees that all process will eventually receive the CPU (by the order they come in), Round Robin is treats all processes equally and constantly switches between them, thus all processes will eventually be processed.

6.23 (7 points) Consider a preemptive priority scheduling algorithm based on dynamically changing priorities. Larger priority numbers imply higher priority. When a process is waiting for the CPU (in the ready queue, but not running), its priority change at a rate α. When it is running, its priority changes at a rate β. All processes are given a priority of 0 when they enter the ready queue. The parameters α and β can be set to give many different scheduling algorithms.

(a) What is the algorithm that results from $\beta > \alpha > 0$?

(b) What is the algorithm that results from $\alpha < \beta < 0$?

(a) First-Come, First-Serve
The instant a process starts running, it will have the highest priority, as $\beta > \alpha > 0$ the priority will continue to increase at a rate greater than any process, thus it will have the highest priority and will continue to execute until completion.
For processes in the queue, processes who have been waiting longer (first come) will have higher priority (first serve) than newer processes in the queue.

(b) Last-In, First-Out
As $\alpha < \beta < 0$, processes in the queue will always have lower priority than any running priority. Further, as running process lose priority slower, running processes will never be interrupted by processes currently in queue. However, any new process with priority 0 will automatically have a higher priority than the running process and will preempt that process pushing it back to the queue (where it will maintain highest priority in queue). Thus, newest processes (Last-In), will be the first to be executed (First-Out), and will only complete if no new process is added during their execution.

7 (20 points) Convert the following program to use threads. Under the following restrictions:

(a) One thread will print "hello", one thread will print "world", and the main function will print the trailing "\n", using just pthread˙create(), pthread˙exit(), pthread˙yield(), and pthread˙join().

(b) You must use a synchronization method to ensure the "world" thread runs after the "hello" thread.

(c) You must use a synchronization method to ensure that the main thread does not execute until after the "world" thread.

```c
#define _GNU_SOURCE
#include <pthread.h>
#include <stdio.h>
#include <sys/types.h>

// every thread will have access to these thread id's
// that way, I can have threads wait for (join with)
// any thread that I want
pthread_t pt_hello;
pthread_t pt_world;

void * world(void * param);
void * hello(void * param);

int main(int argc, char ** argv)  {
        // create the two threads
        // here, I will create pt_hello first so it's handle
        // is available for pthread_join in the world thread
        // if you wanted to create world first, a pthread_yield
        // could be used to halt the world thread until the
        // main thread creates the hello thread, however
        // I found this solution to be a bit more elegant
        // and less bug prone if you were for example to introduce
        // additional threads to further complicate things
        pthread_create(&pt_hello, NULL, hello, NULL);
```

```
        pthread_create(&pt_world, NULL, world, NULL);

        // wait for world to finish (which in turn waits for hello)
        // before going on and printing the new line character
        pthread_join(pt_world, NULL);

        printf("\n");
        return 0;
}

void * world(void * param) {
        // wait for hello to finish before printing world
        pthread_join(pt_hello, NULL);
        printf("world");
        pthread_exit(0);
}

void * hello(void * param) {
        // hello doesn't have to wait for anything
        // it can print hello whenever it sees fit
        printf("hello ");
        pthread_exit(0);
}
```