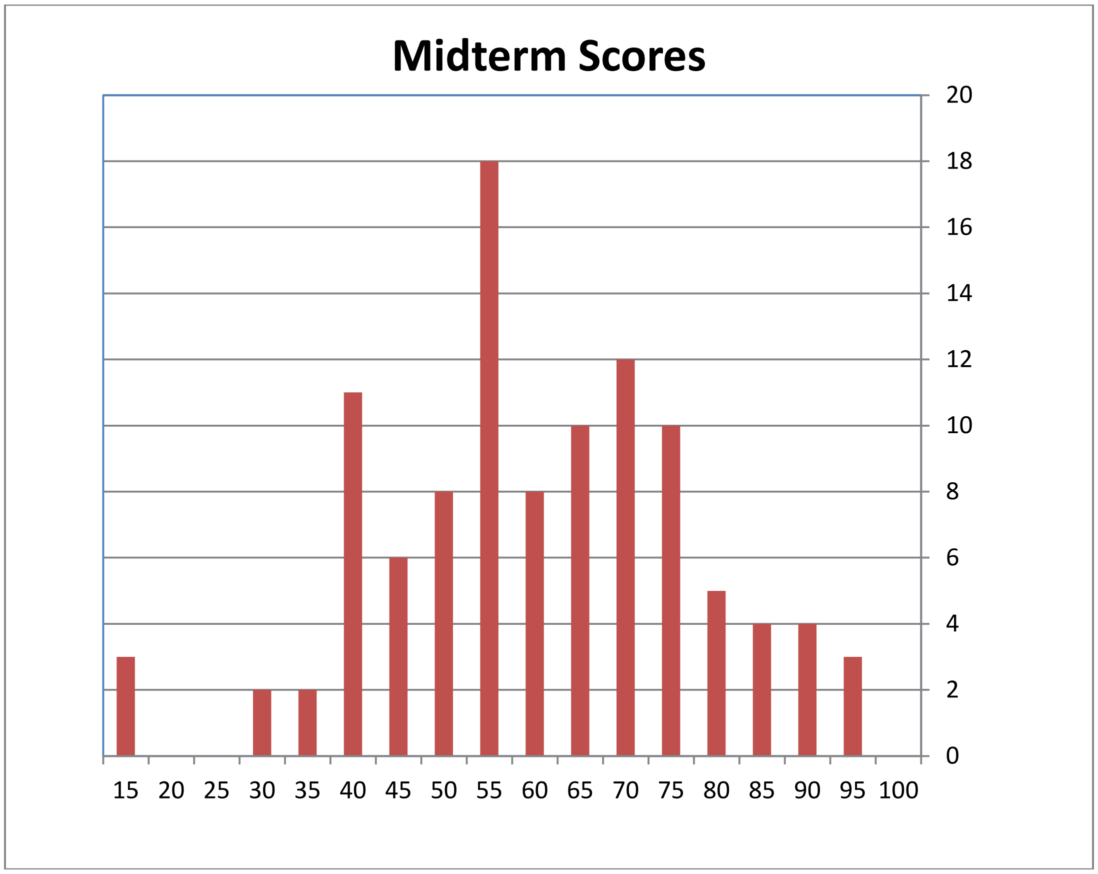
CS 3214 Midterm

Here is the distribution of midterm scores for both sections (combined).



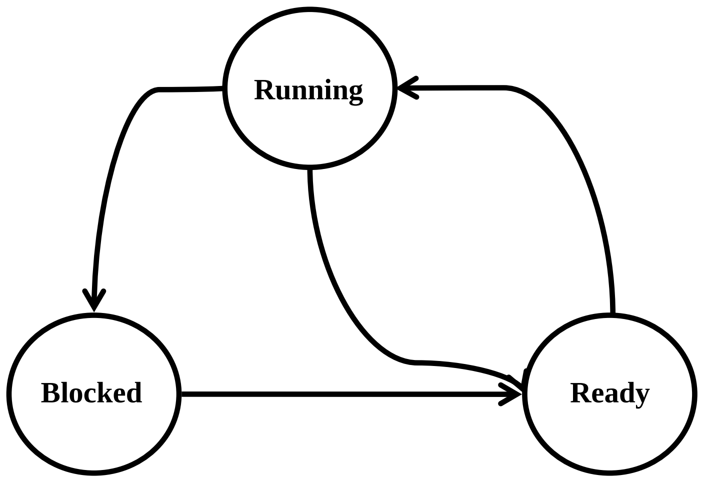
The overall average was 58 points.

#	Problem	Points	Score
I	Processes and Threads	35	
II	Input/Output	35	
III	Synchronization	30	
	Total	100	

I. Processes (34 points)

A simple state model of processes has the three states of Running, Ready, and Blocked.

(a) (5 points) Draw a diagram showing the transitions that can occur between these states. Be sure to account for all transitions.



(b) (12 points) In each scenario given in the table indicate which one(s) of the three state names (Running, Ready, or Blocked) in each column. If there is no state that applies answer "None". If more than one state is possible list all that are possible.

Scenario	Before	After
A process performs a mode switch. What is the state of the process immediately before and immediately after the mode switch?	**running**	**running**
A context switch is performed because process P reached the end of the time period (time slice) allocated to it by the scheduler. What is the state of the process immediately before and immediately after the context switch?	**running**	**ready**
A process performs a fork() operation. What is the state of the parent process immediately before the fork() call and what is the state of the child process immediately after the fork() call?	**running**	**ready or running**
A process performs an exit() system call. What is the state of the process immediately before and immediately after the exit() call?	**running**	**none**

Scenario	Before	After
A process performs a read() on a pipe that contains no data. What is the state of the process immediately before and immediately after the read() call?	**running**	**blocked**
A process has done a wait() system call and later a SIGCHLD signal arrives for which the process has a defined signal handler. What is the state of the process immediately before the delivery of the signal and immediately after the signal handler is entered?	**blocked**	**running**
A process performs an exec() system call. What is the state of the process immediately before and immediately after the exec() call?	**running**	**running or ready**
A process previously received a SIGSTP signal. That process now receives a SIGCONT signal. What is the state of the process immediately before and immediately after the SIGCONT signal is received?	**blocked**	**ready or running**

(c) (18 points) If a process has created two threads and then performs a fork() system call does the child process also have two threads? Answer this question by writing the C code whose output unambiguously answers this question.

```
pthread_t tid;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

int result;

void* test_thread(void* arg) {
  pthread_mutex_lock(&lock);
  printf("thread in process %d \n", getpid());
  pthread_mutex_unlock(&lock);
}

int main() {
  pthread_mutex_lock(&lock);
  pthread_create(&tid, NULL, test_thread, (void*)0);
  if (fork() == 0) {
     pthread_mutex_unlock(&lock);
     pthread_join(tid, (void*)&result);
     exit(0);
  }
  wait(&result);
  pthread_mutex_unlock(&lock);
  pthread_join(tid, (void*)&result);
}
```

II. I/O (35 points)

(a) A file descriptor is an abstract identifier for a source/destination for I/O.

1. (6 points) Identify three different types of sources/destinations which can be referred to by a file descriptor.

   files
   pipes
   terminal (or devices in general)
   kernel information
   network connections

2. (6 points) Name two concrete advantages for an application developer because a file descriptor is an abstract identifier.

   - **write programs whose I/O is independent of the actual source/destinations**
   - **the file descriptors can be rearranged allowing child processes to be "rewired" for a different I/O configuration than their parents.**

(b) (8 points) Shown in the left column is code that creates and manipulates file descriptors. For each read/write operation shown in the middle column indicate in the right column the data stream affected by the operation. If the read/write operation is erroneous, write ERROR. Assume that all standard streams are open, that file descriptors are assigned in sequential order, and that no other file descriptors are used except those shown.

code	**read/write operation**	**affected stream**
`int f1, f2, fd[2];`	`read(0,…)`	**pipe**
`pipe[fd];` `f1 =open("file1",O_RDWR);` `f2 =creat("file2",S_IRWXU);`	`write(0,…)`	**error**
	`write(1,…)`	**stdout**
`dup2(3,0);` `dup2(5,3);` `close(5);`	`write(2,…)`	**stderr**
	`read(3,…)`	**file1**

	`write(3,…)`	**file1**
	`write(4,…)`	**pipe**
	`write(5,…)`	**error**
	`read(6,…)`	**file2**
	`write(6,…)`	**file2**

(c) (15 points) For a certain shell the command A || B means that the standard output of A should be connected to the standard input of B and the standard input of A should be connected to the standard output of B. Write the C code of a parent process that would create the above configuration.

```
pipe1[2];
pipe2[2];
pipe(pipe1);       // create two pipes
pipe(pipe2);

if(fork() == 0) { // launch child A
   dup2(pipe1[1], STDOUT);
   dup2(pipe2[0], STDIN);
   close(pipe1[0]);
   close(pipe2[1]);
   exec(A);
}
if(fork() == 0) { // launch child B
   dup2(pipe1[0], STDIN);
   dup2(pipe2[1], STDOUT);
   close(pipe1[1]);
   close(pipe2[0]);
   exec(B);
}

close(pipe1[0]);   // parent closes all pipe ends
close(pipe1[1]);
close(pipe2[0]);
close(pipe2[1]);
```

III. Synchronization (30 points)

A multi-threaded application defines a global accessible structure as follows:

```
struct {
  int size;
  app_data_t A[100];
} app_data;
```

This data is shared by several threads each of which adds data to the array. The threads use the functions `size(app_data ad)` to find the current number of elements in the array and `add(app_data ad, app_data_t data)` to add data into the array. The size and add functions internally insure mutual exclusive access to the shared data. The thread code looks like this:

```
    /* body of function executed by each thread */
       app_data_t data;
       while (size(app_data) < 100) {
           produce(&data);
           add(app_data, data)
      }
      pthread_exit();
```

(a) (5) The thread code has an atomicity violation. Explain how this violation can occur.

A scenario illustrating the atomicity violation is as follows:

1. **the thread tests the condition in the while loop and find it true**
2. **before the thread can read the add() function some other thread(s) fill up the array**
3. **the original thread then calls add() overflowing the array**

(b) (10 points) Show all code, including initializations, that you would need to solve the atomicity violation.

```
global declaration:
pthread_mutex_t mutex = PTHREAD_MUTEX_INTIALIZER;
bool done = false;

while(!done) {
     produce(&data);
     pthread_mutex_lock(mutex);
         if (size(app_data) < 100)
            add(app_data, data);
         else done = true;
     pthread_mutex_unlock(mutex);
}
pthread_exit();

Note: an answer with semaphores is also acceptable
```

(c) (15 points) Multiple clients synchronize their use of a shared resource using three synchronizations functions according to this pattern:

```
   ...
int t = get_ticket();
   ...
use_ticket(t);
  // use resource
done_ticket();
```

where the three synchronizing functions are implemented as follows:

```
int get_ticket() {
  pthread_mutex_lock(&mutex);
  int my_ticket = ticket;
  ticket++;
  pthread_mutex_unlock(&mutex)
  return my_ticket;
}

void use_ticket(int my_ticket) {
  pthread_mutex_lock(&mutex);
  while(currrent != my_ticket)
     pthread_cond_wait(&turn, &mutex);
  pthread_mutex_unlock(&mutex)
}
```

```
void done_ticket() {
  pthread_mutex_lock(&mutex);
  current++;
  pthread_cond_broadcast(&turn);
  pthread_mutex_unlock(&mutex);
}
```

1. (5 points) Is the use of the resource mutual exclusive?

Yes. Because the `use_ticket` function delays all threads from proceeding except the single thread whose ticket number matches the current ticket.

2. (5 points) Is there an order by which the clients use the resource? If so, what is the order?

Yes. The clients use the resource in the order of the ticket numbers that they received in `get_ticket`.

3. (5 points) Why does the `done_ticket` function use a broadcast operation?

Because there is no guarantee that a signal will awaken the thread with the lowest ticket number. Therefore, all threads have to be awakened so that they can (re)test whether their ticket number corresponds to the current ticket.