This is a closed book (and notes) examination. Answer all questions on the exam itself. Take the number of points assigned to each problem and the amount of space provided for your answer as a measure of the length and difficulty of the expected solution. The exam totals 100 points.
1. (8 points) The bounded waiting condition is one of four conditions necessary for a correct solution to the mutual exclusion problem. What is it? What does it prevent?

2. (18 points) As shown below, processes can be in one of three states: running, ready and blocked. There are six possible state transitions (labeled 1-6). For each label, indicate whether the transition is valid or invalid. If valid, indicate when the transition is used for a process. If the transition is not valid then indicate why.

State transitions:
(a) 1: Blocked to Running
(b) 2: Running to Blocked
(c) 3: Ready to Blocked
(d) 4: Blocked to Ready
(e) 5: Ready to Running
(f) 6: Running to Ready
3. (8 points) What data structure in the operating system is used to store information about processes? How is this structure used in context switching between processes?

4. (15 points) In Project 1, you created a child process with *fork()* and had it execute a given command with *execvp()*.

   (a) How did your program know when it was executing code for the child process?

   (b) What would have been the difference in your program if it had used the system call *execve()* rather than the routine *execvp()* to execute a command?

   (c) Your parent process used the *wait()* system call to wait for the child process to complete. In terms of process coordination, is this an example of mutual exclusion, synchronization or a barrier? Why?
5. (8 points) The Test-and-Set-Lock (TSL) instruction used in a spin lock is *atomic*. What does atomic mean in this context and why is it important?

6. (13 points) The mutual exclusion problem can be solved using two routines: *BeginRegion()* and *EndRegion()* to define the critical region for access to a shared resource. Processes call these routines when accessing the shared resource. The following code is a proposed solution to this problem using semaphores where a single semaphore is created and used as shown.

```
Initialization:
    m = semcreate(x);

BeginRegion()
{
    semwait(m);
}

EndRegion()
{
    semsignal(m);
}
```

Consider the following three values for *x* passed to *semcreate()* as the initial count of the created semaphore. In each case briefly explain the outcome if the given value is used.

(a) *x*=0

(b) *x*=1

(c) *x*=2
7. (12 points) The code below shows a portion of code to implement a version of the producer/consumer problem using semaphores and a shared buffer. The code shows the initialization of three semaphores along with the routines executed by producer and consumer processes.

Initialization:
\[
\begin{align*}
p &= \text{semcreate}(50); \\
c &= \text{semcreate}(0); \\
m &= \text{semcreate}(1);
\end{align*}
\]

Producer() // called by producer process(es)
\{
    while (1) {
        semwait(p);
        AddItemToBuffer();
        semsignal(c);
    }
\}

Consumer() // called by consumer process(es)
\{
    while (1) {
        semwait(c);
        semwait(m);
        RemoveItemFromBuffer();
        semsignal(m);
        semsignal(p);
    }
\}

(a) Indicate the number of items that can be stored in the shared buffer or indicate why this value cannot be determined based on the given information.

(b) Indicate the number of producer processes that can use the Producer() routine to add items to the shared buffer or indicate why this value cannot be determined based on the given information.

(c) Indicate the number of consumer processes that can use the Consumer() routine to remove items from the shared buffer or indicate why this value cannot be determined based on the given information.
8. (8 points) The concept of a thread was introduced to allow multiple execution contexts within a process. Name one advantage and one disadvantage of the use of multiple threads relative to the use of multiple processes for an application.

9. (10 points) Assume that an operating system supports kernel-level threads and therefore schedules which thread to run next from a queue of ready threads. All threads are treated equally in scheduling so that if one process has ten threads and another process of the same priority only one then threads of the first process will receive over 90% of the CPU time while the thread of the second process less than 10%.

Describe needed changes for a thread scheduling policy that allocates CPU time equally on a per-process rather than a per-thread basis so that in the above scenario each thread of the first process receives approximately 5% of the CPU time while the single thread of the second process receives 50%. Note that your policy should work on a multiprocessor so multiple threads may be running in parallel.