90 min, closed book, closed notes exam.

1. **Short answer questions. (20pts)**

   Write between one sentence and a paragraph in response to each of the following.

   (a) The OS uses a process control block (PCB) to maintain the state of a process and a thread control block (TCB) to maintain the state of a thread. What differentiates a PCB from a TCB, and what impact does this difference have on context switching?

   A Thread Control Block does not have many of the items in a Process Control Block, since each TCB is also connected to a process. For example, a TCB does not need to have a process ID, memory management information, a list of open files, etc. The TCB and PCB both contain a PC, Registers, and SP. The OS can switch between threads within the same process more quickly than between different processes because there is less information to store and restore.

   (b) To which type of jobs do scheduling policies for interactive systems give priority and why?

   Jobs with long I/O bursts and short CPU bursts get priority since I/O is interactive, and the CPU processing is typically shorter than a time slice.

   (c) What is the purpose of synchronization?

   To coordinate access to shared state between different processes or threads.

   (d) What advantages does the atomic instruction "test&set" have over atomic loads and stores?

   Since test&set instructions read and modify a value atomically, they can usually enforce synchronization constraints with less busy waiting than atomic loads and stores.

2. **Processes & Threads. (20pts)**

   Using the *fork(), waitpid(), exit()* and *kill()* system calls, write a program in which a parent creates a child and the child creates a grandchild. The grandchild prints "I am a grandchild" and sleeps for 60 seconds. The child then kills the grandchild and exits. The parent waits for the child to finish and prints "Child finished" and exits. You may use pseudo-code to write your program.

```
int main() {
  int  child,grand_child,status;
  char c[5];

    child = fork();
    if (child == 0) { /* this is the child */
          grand_child = fork();
           if(grand_child == 0) { /* this is the grand child */
                printf("I am the grandchild with pid %d .\n",getpid());
```

```
            sleep(60);
            }
          else { /* child */
               printf("I am the child with pid %i.\n",getpid());
               printf("hit any key to kill the grandchild\n");
               gets(c);
               kill(grand_child,SIGKILL);
               exit(0);
          }

      }
        else {/* this is the parent */
          pid = getpid();  /*get pid for parent*/
          printf("I am the parent with pid %i.\n",pid);
           wait(&status); /* wait for child to finish;
                        can also use wait pid(child) here */
           printf("Child is finished.\n");
        }
        return 0;
}
```

3. **CPU Scheduling. (20pts)**

   Consider a variant of Round Robin scheduling in which the usual ready queue (in which PCBs are linked together) is replaced by a queue of pointers to PCBs.

   (a) What is the effect of putting two pointers to a PCB on the queue? (5pts)

   The OS will give the process twice as many time slices as the other processes that only get one pointer, in effect, it increases the job's priority.

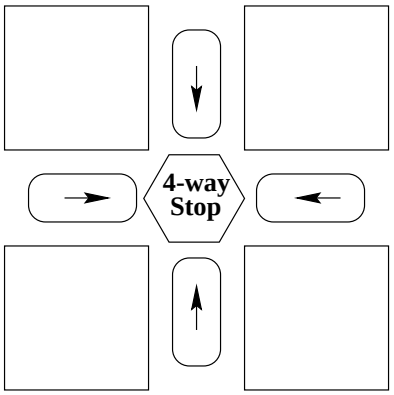
   (b) What is one advantage and one disadvantage of this approach? (5pts)

   Higher priority jobs can get more time slices without changing the basic Round Robin scheduling algorithm. The disadvantage is that short jobs will suffer even more than they already do in Round Robin scheduling.

   (c) How could you modify the original Round Robin scheduling algorithm to achieve the same effect without using the queue of pointers? Is your approach better or worse than queues with multiple entries? (10pts)

   Assign priorities to jobs and then give higher priority jobs longer time slices. This approach is better than multiple queue entries because it will perform fewer context switches.

4. **Synchronization & Deadlock. (40pts)**



Consider this 4-way stop sign, with traffic from all four directions.

(a) Does the rule "yield to the car on the right" prevent deadlock? Why or why not? (10pts)

No this rule does not prevent deadlock, since all four are waiting for the car on the right to go and the wait is circular. All four conditions for deadlock occur.

(b) What is the difference between deadlock and starvation? (5pts)

Deadlock means that none of the processes in the set can make progress, while starvation means that some processes are making progress, it is just that some unlucky process(es) are not. For example, if a process has a low priority, higher priority processes may keep getting the resources.

(c) Instead of the traditional stop sign, you must design an electronic system based on Monitors that allows only one car to enter the intersection at once. Your solution should prevent starvation and deadlock. (25pts)

This solution is fair, but it does not minimize response time. If a car arrives from direction 3, but in the mean time a car arrives from direction 2, the car from direction 2 will always get to go first. However, no one will starve or deadlock, since on each execution of Go all directions are signaled.

```
class IntersectionMonitor {
  public:
    void Arrive&Stop (d: 0..3);
    void Go (d: 0..3);
  private:
    lock = FREE: Monitor Lock;
    atLight[0..3] = False: Condition Variable;
}
Intersection::Arrive&Stop(d: 0..3){
  lock->Acquire();
  atLight[d]->wait();
  ... go thru intersection ...
  lock->Release();
}
Intersection::Go(d: 0..3){{
  lock->Acquire();
```

```
  for (i = 0 to 3)
    atLight[i]->signal;
  lock->Release();
}

  repeat forever
    Go();
```