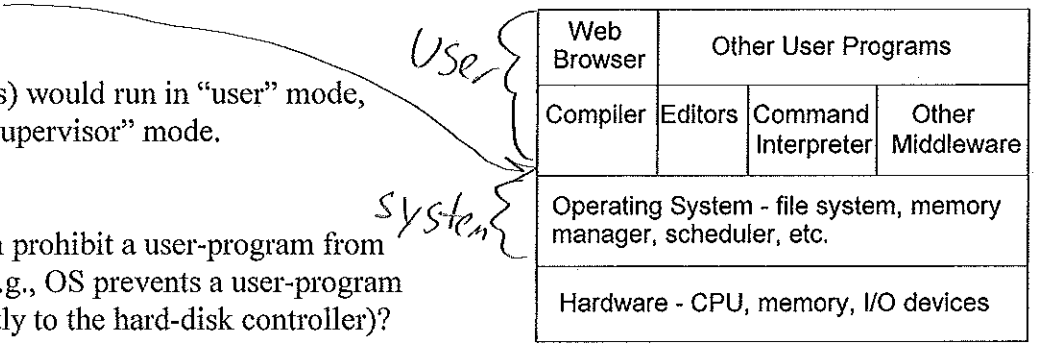
Spring 2012 OS Test 1 Name: Mark F.

Question 1. (10 points) Consider the various layers within a computer system.

a) Identify where the OS's system call API would be located in the diagram.

b) Indicate which software layer(s) would run in "user" mode, and which would run in "system/supervisor" mode.

c) Why does the operating system prohibit a user-program from directly accessing the hardware (e.g., OS prevents a user-program from issuing I/O commands directly to the hard-disk controller)?



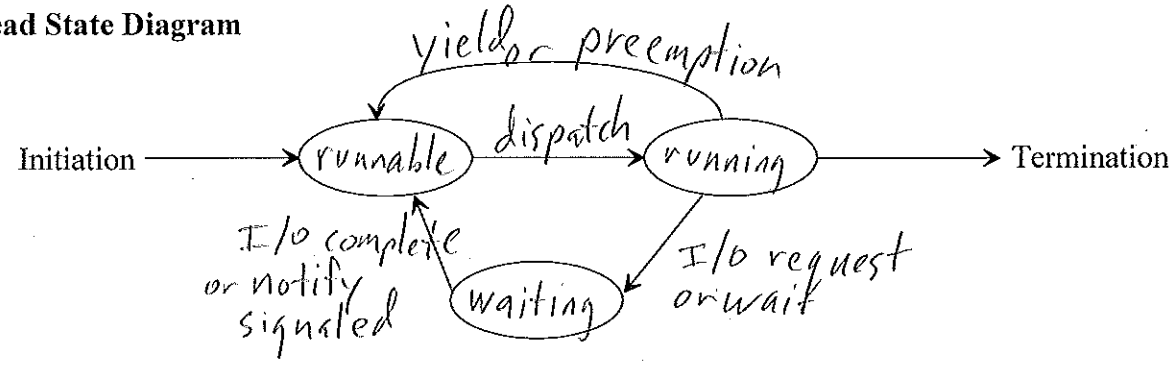
If a user pgm can access hardware, then it could by-pass to OS protection checks, e.g., user A's pgm might read user B's private files.

Question 2. (14 points) Complete the thread state diagram below by:

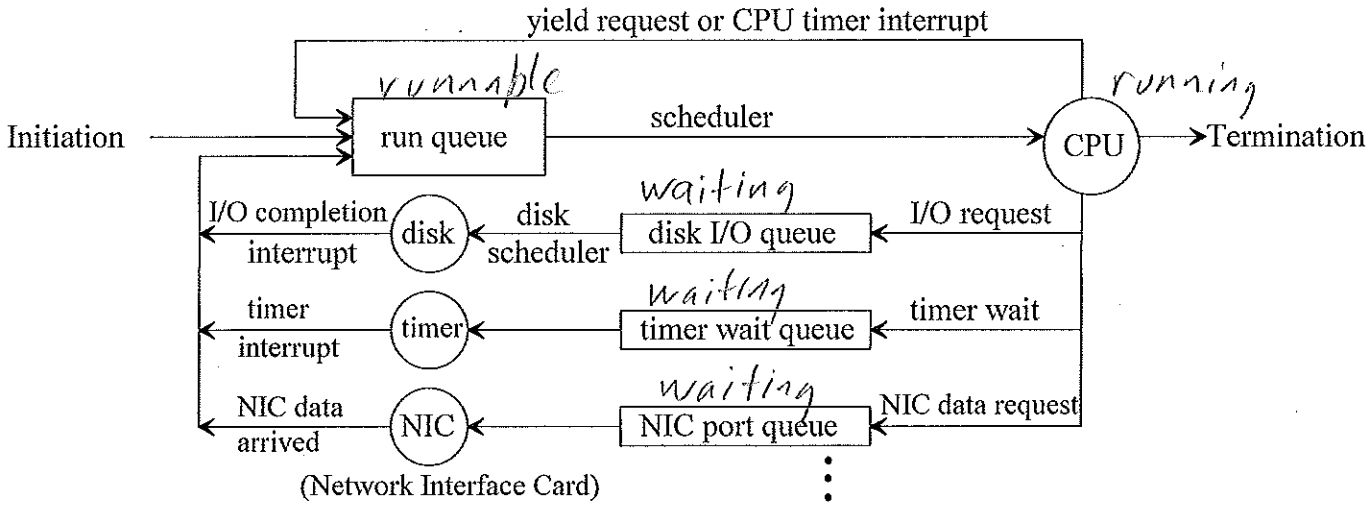
a) Labeling the states (i.e., ovals): "waiting", "runnable", "running"

b) Labeling the transistion between states: "I/O or wait request", "dispatch", "yield or preemption", "I/O complete or notify signaled"

Thread State Diagram



c) Think of the TCB (thread control block) for a thread as moving around from queue to queue depending on its state. Put the state labels **("waiting", "runnable", "running")** on the components contains thread(s) in that state.



Question 3. (6 points) Threads within a multi-threaded program share some components, but have unique copies of others. From the list below, **circle all shared components**.

- program instructions
- global data
- stack pointer
- run-time stack
- set of registers
- priority

6

Question 4. (5 points) Consider an example where we might want to dynamically adjust priorities: have the OS adjust disk-bound threads (e.g., virus scan) toward higher priorities, and CPU/processor-bound threads (e.g., graphics rendering) toward lower priorities.

a) Which performance criteria does this improve: throughput or response time?

b) Explain why performance is improved.

5 If disk-bound threads run first due to higher priority, then I/O for it can overlap execution with CPU-bound threads.

Question 5. (5 points) Consider an example where we might want to dynamically adjust priorities: have the OS adjust interactive threads (e.g., word processor) toward higher priorities and CPU/processor-bound threads (e.g., graphics rendering) toward lower priorities.

a) Which performance criteria does this improve: throughput or response time?

b) Explain why performance is improved. Clearly, having interactive threads have

5 higher priorities will reduce response time since they run sooner after entering the run queue.

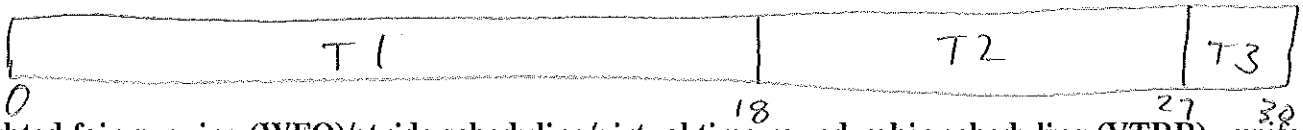
Question 6. (15 points) *Proportional-Share scheduling* - allow the user (or system) to specify the proportion of CPU time each thread should be allocated. Answer the following questions on the main types of proportional-share scheduling.

weighted round-robin scheduling (WRR) - round-robin, but threads with higher weights get longer time-slices

a) Consider three threads we want to run with the following proportions. If we want to run all three in 30 ms, how big of time slices would each be allocated?

Thread	Proportion of CPU		CPU Time-Slice (ms)
T1	6	$\frac{6}{10}$ $\frac{3}{5}$	18sec
T2	3	$\frac{3}{10}$	9sec
T3	1	$\frac{1}{10}$	3sec

b) Draw the Gantt chart for this schedule.

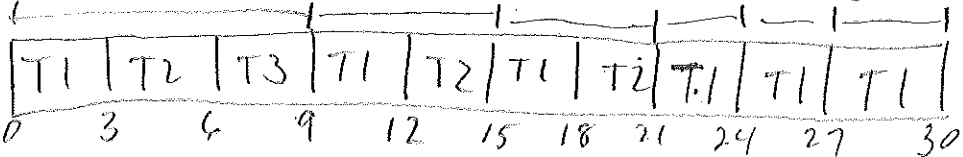


weighted fair queuing (WFQ)/stride scheduling/virtual time round-robin scheduling (VTRR) - uniform time-slice, but smaller allocation threads sit out some round-robin rotations through the runnable-thread list.

c) Consider three threads we want to run with the following proportions, complete the table with the percentage of round-robin rotations each thread participates?

Thread	Proportion of CPU		Percentage participation in round-robin rotations
T1	6	$\frac{6}{6}$	100%
T2	3	$\frac{3}{6}$	50%
T3	1		1/6 =

d) Draw the Gantt chart for the first 6 round-robin rotations assuming a uniform time-slice of 3 ms.



6

lottery scheduling - uniform time slice, but large allocation threads win lottery more often

e) Why is lottery scheduling rarely used on real systems?

5 Due to the randomness of the lottery, a high proportion thread might not run for relatively long periods.

Question 7. (10 points) What advantages does a higher-level concurrent programming tool such as monitors have over using mutexes?

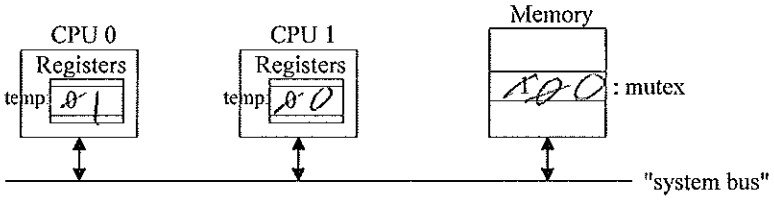
10 A high level concurrent tools are less likely to have simple programming errors, e.g., mismatched lock & unlock of mutexes, etc.

Question 8. (10 points) Consider the three mutex mechanisms we discussed: spinlock, cache-conscious spinlock, and queuing mutex.

a) Modern architectures have an "*exchange operation*" which atomically (not interruptable) swaps contents of a register and a memory location. On the diagram, show the results after executing the **sequence** of operations:

CPU 0: exchange temp, mutex
CPU 1: exchange temp, mutex

2



Spinlock Algorithm: (mutex values: 0 ≡ locked, 1 ≡ unlocked)

Code to Lock mutex:	**Code to Unlock mutex:**
let register temp = 0 repeat atomic exchange temp, mutex until temp == 1	mutex = 1

b) What disadvantage does this simple spinlock implementation have?

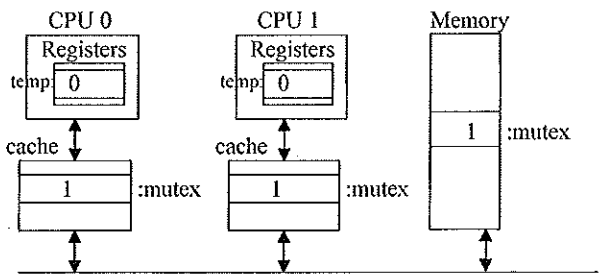
4 The CPU busy-waits when a thread waiting on the lock is scheduled and continues to write to memory.

Assume a cache-coherence protocol using "write-invalidates" (e.g. MESI). Consider the

Cache-Conscious Spinlock Algorithm: (mutex values: 0 ≡ locked, 1 ≡ unlocked)

Code to Lock mutex:	**Code to Unlock mutex:**

let register temp = 0
repeat
  atomic exchange temp, mutex
  if temp == 0 then
  while mutex == 0 do
  do nothing
  end while
  end if
until temp == 1
	mutex = 1



c) How does the cache-conscious spinlock help reduce the memory contention?

4 A mutex waiting on the lock will be looping in the while-loop which only reads the mutex value. By not doing the atomic exchange operation which writes mutex, other CPU's caches values are not invalidated, and forced to be reread.

5

Question 9. (10 points) Consider the textbook Dining Philosopher's code which has a deadlock race condition.

```
public class Philosopher extends Thread{

  private Object leftFork, rightFork;
  private int myNumber;

  public Philosopher(Object left, Object right, int number){
    leftFork = left;
    rightFork = right;
    myNumber = number;
  }

  public void run(){
    int timesDined = 0;
    while(true){
      synchronized(leftFork){
        synchronized(rightFork){
          timesDined++;
        }
      }
      if(timesDined % 100000 == 0)
        System.err.println("Thread "+myNumber+" is running.");
    }
  }

  public static void main(String[] args){
    final int PHILOSOPHERS = 5;
    Object[] forks = new Object[PHILOSOPHERS];
    for(int i = 0; i < PHILOSOPHERS; i++){
      forks[i] = new Object();
    }
    for(int i = 0; i < PHILOSOPHERS; i++){
      int next = (i+1) % PHILOSOPHERS;
      Philosopher p = new Philosopher(forks[i], forks[next], i);
      p.start();
    }
  }
}
```

a) Explain how a deadlock can occur with this code.

Each philosopher picks up their left fork and holds it they circular wait on philosopher on their right.

b) Explain how deadlock can be prevented.

Have one philosopher pick up forks in reverse order breaking the circular wait cycle.

Question 10. (15 points) Consider my Java solution to the Readers/Writers Locks problem.

```
class RWLockManager {
  private int writersOccupied = 0;
  private int readersOccupied = 0;

  public synchronized void readLock()
    throws InterruptedException
  {
    while(writersOccupied > 0)
      wait();
    System.out.println("readLock granted");
    readersOccupied++;
  }

  public synchronized void writeLock()
    throws InterruptedException
  {
    while(writersOccupied > 0 || readersOccupied > 0)
        wait();
    System.out.println("writeLock granted");
    writersOccupied++;
  }

  public synchronized void rwUnlock()
    throws InterruptedException
  {
      if (writersOccupied > 0) {
          writersOccupied--;
          System.out.println("unlocking by a writers!!!");
          notifyAll();
      } else if (readersOccupied > 0) {
          readersOccupied--;
          System.out.println("unlocking by a reader!!!");
          notifyAll();
      } else {
          System.out.println("ERROR: unlocking, but NO"+
                             " readers or writers");
      } // end if
  }// end rwUnlock

} // end class RWLockManager
```

a) The `readLock` method increments the `readersOccupied` variable. How are concurrent readers prevented from "messing" up the `readersOccupied` variable without a mutex?

All the methods are synchronized, so only one thread can be executing a method. Thus, mutual exclusion is guaranteed.

b) Why are the `while` loops around the `wait()` statements necessary for correct functioning of the code?

We are using notifyAll which waits up all readers & writers. The while loops allow repeated checking until conditions are favorable for woke thread to proceed. Java documentation indicates that a thread might also be woke "randomly" without a notify, so check of while loop is needed.