50 Questions

1. Define the stack, text section, data section, and heap of a process.

2. Suppose I start multiple copies of the Chrome browser (it shows up multiple times on the list of running processes). Which of those sections mentioned in (1) can be shared among those processes?

3. What is dual mode operation?

4. What is the definition of a process?

5. What is the definition of a thread?

6. What states can a thread be in?

7. What is an interrupt?

8. What is a trap?

9. What is an exception?

10. What is a process control block?

11. What is paravirtualization?

12. What is a microkernel architecture?

13. How would you describe the state of a process on the ready queue? On the wait queue?

14. What is VMWare?

15. What is the difference between virtualization and simulation?

16. What is a bootstrap program?

17. How many threads can be in the running state on an Intel i7 processor with four cores (ignoring hyperthreading)? How many can be in the ready state?

18. Consider the following two threads with a shared variable x initially set to 0. What is the final value of x?

```
THREAD 0            THREAD 1
x = x + 1           x = x + 2
```

19. What is contained in a thread control block?

20. What is multiprocessing?

21. What is multiprogramming?

22. What is the difference between independent and cooperating threads?

23. What is the definition of mutual exclusion?

24. What is a critical section?

25. Suppose I have multiple threads running the same code. How would I add semaphores to the following code to enforce the constraint that only one thread can be in the critical section (include the initial values of any semaphores)?

```
regular code
more regular code
critical section
yet more non-critical code
```

26. What is the difference between the signal operation of a monitor and the signal operation of a semaphore?

27. What is the definition of starvation?

28. What is the definition of deadlock?

29. How would you describe the behavior of the following code?

```
SHARED SEMAPHORES
mutexA = Semaphore(0)
mutexB = Semaphore(0)
```

```
THREAD 1                        THREAD 2

initialization code 1           mutexA.wait()
mutexA.signal()                 initialization code 1
mutexB.wait()                   mutexB.signal()
code section 1                  code section 2
```

30.Consider the following code:

```
SHARED
int xers = 0
mutex = Semaphore(1)
empty = Semaphore(1)
hugeData = a lot of shared data that x and z threads have
access to.
```

```
THREADS OF TYPE Z               THREADS OF TYPE X
empty.wait()                    mutex.wait()
code accessing hugeData         xers += 1
empty.signal()                  if xers == 1:
                                   empty.wait()
                                mutex.signal()
                                code accessing hugeData
                                mutex.wait()
                                xers -= 1
                                if xers == 0:
                                   empty.signal()
                                mutex.signal()
```

What does this code do?

31. What does this code do?

```
SHARED
mutex = Semaphore(1)
bigData = lots of data that is shared among processes

ALL THREADS EXECUTE:

some code
some more code
mutex.signal()
some code accessing bigData
mutex.wait()
additional code
```

32.What is a memory barrier?

33. Consider the following psuedo-code that is executed by n threads

```
prologue code
main body code
```

I would like to create a **barrier** between the prologue code and the main body meaning that all threads need to execute their prologue code before any thread can execute the main body. I have two integer variables:

```
n = total number of threads
count = 0
```

What is a solution to this problem?

34.What are the necessary conditions for deadlock?

35.In discussing deadlock, what is the definition of a safe sequence?

36.Consider the following example

```
5 processes p0-p5; 3 resource types: A (10 instances), B (5
instances), and C (7 instances)
```

at a certain time of execution this is the current state:

	Alloc			Max			Avail		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	3	3	2
P1	2	0	0	3	2	2			
P2	3	0	2	9	0	2			
P3	2	1	1	2	2	2			
P4	0	0	2	4	3	3			

What is the value of need P2?

What is a safe sequence for this example?

Which of the following can be granted in this state and still keep the system safe?

```
P0 -> 0 3 0 ; P1 -> 1 0 2 ; P2 -> 4 0 0 ; P4 -> 4 0 1 ;
P0 -> 1 1 3
```

37. Consider the Java class theSharedData listed at the end of this test. What does the synchronized keyword indicate? What does wait() in the get method do?

38. What is the difference between user-level threads and kernel-supported threads?

39. On an Intel processor what are the %EIP, %ESP and %EBP registers? What are their functions?

40. In the C what does the structure `struct ucontext` contain and what is its function?

41. Describe a thread's stack on the x86 architecture.

42. What is a condition variable?

43. What is the difference between Mesa/Hansen and Hoare monitors?

44. What is a spinlock?

45. In Java, is it better to use a synchronized method or a synchronized block within a method? (see last page of this handout)

46.Is it better to use monitors or semaphores?

47. Making water. You need two hydrogen atoms (H) and one oxygen (O) to make water. Write a semaphore solution that generates water as soon as the atoms are available.

48. What is priority inversion? How can it be prevented?

49. For the project, when you create a new thread with ULT_CreateThread() (a function you write) when does the thread start executing?

50.For the project, what (if anything) do you need to change in the saved ucontext_t when you do a ULT_Yield()?

```
class theSharedData extends Object {
   int                     dataPresent;
   int                     sharedData;

   public theSharedData() {
      dataPresent=0;
      sharedData=0;
   }
   public synchronized void get() {
      while (dataPresent == 0) {
         try {
            System.out.print("Consumer " +
                             Thread.currentThread().getName() +
                             ": Wait for the data to be produced\n");
            wait();
         }
         catch (InterruptedException e) {
            System.out.print("Consumer " +
                             Thread.currentThread().getName() +
                             ": wait interrupted\n");
         }
      }
      System.out.print("Consumer " +
                       Thread.currentThread().getName() +
                       ": Found data or notified, CONSUME IT " +
                       "while holding inside the monitor\n");
      --sharedData;
      if (sharedData == 0) {dataPresent=0;}
   /* in a real world application, the actual data would be returned */
   /* here                                                           */
   }
   public synchronized void put() {
      System.out.print("Producer: Make data shared and notify consumer\n");
      ++sharedData;
      dataPresent=1;
      notify();
      System.out.print("Producer: Unlock shared data and flag\n");
      /* unlock occurs when leaving the synchronized method */
   }
}
```

Two Pitfalls in Java (from Essential Operating Systems Principles by Tom Anderson, Peter Chen, Mike Dahlin, and Steve Gribble.

Java is a modern type-safe language that included support for threads from its inception. This built-in support makes multi-threaded programming in Java convenient. However, some aspects of the language are *too* flexible and can encourage bad practices. We highlight two pitfalls here.

1. Avoid defining a synchronized block in the middle of a method
Java provides built in language support for shared objects ("monitors.") The base Object class, from which all classes inherit, includes a lock and a condition variable as members. Then, any method declaration can include the keyword synchronized to indicate that the object's lock is to be automatically acquired on entry to the method and automatically released on any return from the method. E.g.,

```
public synchronized foo(){
// Do something; lock is automatically acquired/released.

}
```

This syntax is useful—it follows rule #2 above, and it frees the programmer from having to worry about details like making sure the lock is released before every possible return point including exceptions. The pitfall is that Java also allows a *synchronized block* in the middle of a method. E.g.,

```
public bar (){
// Do something without holding the lock

synchronized{

// Do something while holding the lock

}

// Do something without holding the lock

}
```

This construct violates rule #2 above and suffers from the disadvantages listed there. The solution is the same as discussed above: when you find yourself tempted to write a synchronized block in the middle of a Java method, treat that as a strong hint that you should define a separate method to more clearly encap- sulate the logical chunk you have identified.

2. Keep *shared state classes* separate from *thread classes*
Java defines a class called Thread that implements an interface called Runnable that

other classes can implement in order to be treated as threads by the runtime system. The pitfall is that the Thread class inherits from the Object class, so the line between shared objects and threads can be muddled; programmers may even be tempted to define a single class that includes both a shared object and the code for the threads that act on it. As Figure 2.9 illustrates, it is better to think of threads and shared objects as separate. *Shared objects* should include all *shared state*. *Thread objects*—objects that inherit from Thread or implement Runnable—should include only *per-thread state*.