CSE306 - Homework2 and Solutions

Note: your solutions must contain enough information to enable the instructor to judge whether you understand the material or not. Simple yes/no answers do not count. A simple core dump of everything you remember will not do it either. Take a look at the solutions to see what level of detail is required.

1. Consider the following snapshot of the system, where P1–4 are processes and R1-4 are resource types:

processes	allocation				max-need			
P0:	0	0	1	2	0	0	1	2
P1:	1	0	0	0	1	7	5	0
P2:	1	3	5	4	2	3	5	6
P3:	0	0	1	4	0	6	5	6
	R1	R2	R3	R4	R1	R2	R3	R4
free:	1	5	2	0				

   Suppose process P1 issues a request for 4 instances of R2 and 2 instances of R3. Will the situation be safe if OS grants this request? Explain.

2. Write a program using semaphores, where each process represents one dining philosopher and which coordinates the philosophers in a deadlock-free manner. The solution must use only as many semaphores as there are philosophers. (**Note**: this means that the additional semaphore "`room`", as in the textbook, is not allowed.)

3. Write a monitor that implements an *alarm clock*. Alarm clock is a system call built around the hardware timer interrupt, which works like this. A process issues a system call that sets alarm clock to some specified number of time units. Once the time is up, the timer interrupt handler calls your monitor routine *wakeup*, at which point the process waiting in the monitor wakes up.

4. Suppose it takes 5 ms to service a read/write request, and it takes $10 + 5(N - 1)$ ms to move the disk read/write head across N cylinders. Suppose that at time 0 the head was at cylinder 0, and that new requests for cylinders #5, #20, #30, #15, #25, #50, #40 had arrived at times 5 ms, 50 ms, 70 ms, 110 ms, 300 ms, 355 ms, and 500 ms, respectively.

   Assuming the LOOK strategy (disk head reverses its course when there are no outstanding requests in the current direction), what will be the order in which these requests will be serviced? Show how you arrived at your conclusions.

5. A DMA module transfers data into the main memory at the rate of 9600 bits/sec. The CPU can fetch instructions from the main memory at the rate of 1,000,000 instructions/sec. As you know, DMA and CPU can access main memory simultaneously using a technique called cycle-stealing (which allows both the CPU and DMA to access main memory without relying on interrupts).

   Explain by how much cycle-stealing due to DMA might slow down the processor. Assume that DMA has to access main memory to transfer each byte it gets from the external device.

6. Executable programs are stored in binary files. Someone suggested that in order to save on the backing store needed to support virtual memory, the OS can try to page programs right out of the binary files where they are stored. What's wrong with this idea?

7. Implement a general semaphore using monitors.

8. Suggest how one can implement a monitor using semaphores.

9. Consider the following resource allocation:

	Allocated				Requests				Available			
	R1	R2	R3	R4	R1	R2	R3	R4	R1	R2	R3	R4
P1	0	0	1	0	2	0	0	1	2	1	0	0
P2	2	0	0	1	1	0	1	0				
P3	0	1	2	0	2	1	0	0				

   Use the deadlock detection algorithm to determine if there is a deadlock.

10. Consider RAID disks with and without disk striping (*i.e.*, when individual bytes of the same file are striped across different disks). Which access patterns favor disk striping and which do not?

11. Consider a Unix file system. Suppose the file Inode is in the main memory, but nothing else (pertaining this file) is. How many disk accesses will be required to get to byte 15,423,000 of this file?

    Assume these file system characteristics: 12 direct block pointers, 1 singly indirect, one doubly-indirect, and one triply-indirect pointer per inode. System block size (which is transfered in one I/O op) is 8K. A disk block pointer takes 8 bytes, so you can assume that each block of indirection has 8K/8 = 1024 pointers.

12. Consider a file system that uses the chained space allocation method. What are the disadvantages of this method in terms of reliability and efficiency?

13. What is easier to update (add or delete records): an indexed file or an index-sequential file? An index-sequential or a hashed file?

14. Which files are best for random access, hashed, indexed, or index-sequential?

15. As you know, a timer is a special register that gets decremented automatically and when its value becomes zero, an interrupt occurs. So, when only one process in the system requests the timer service, the mechanism by which the process gets notification is clear. However, in reality, there might be many overlaping requests for timer interrups, which come from different processes. How does the OS manage all these requests with just one timer register?

16. Consider a real time system with process characteristics depicted in the following table:

Process	Arrival Time	Execution Time	Starting Deadline
A	10	20	100
B	20	20	25
C	40	20	60
D	50	20	80
E	60	20	70

    Show how these processes will be scheduled using the *non-preemptive* Earliest Deadline First strategy that uses starting deadlines. Next, do the same for the case of *unforced* idle times under Earliest Deadline strategy.

17. Describe a situation where strategies with unforced idle time are practical.

1. If the request is granted, then the state will become:

processes	allocation				max-need			
P0:	0	0	1	2	0	0	1	2
P1:	1	4	2	0	1	7	5	0
P2:	1	3	5	4	2	3	5	6
P3:	0	0	1	4	0	6	5	6
	R1	R2	R3	R4	R1	R2	R3	R4
free:	1	1	0	0				

In this situation, all processes can run to completion. Indeed, P0 has all the resources it needs and it can run to completion. The state then will become:

processes	allocation				max-need			
P1:	1	4	2	0	1	7	5	0
P2:	1	3	5	4	2	3	5	6
P3:	0	0	1	4	0	6	5	6
	R1	R2	R3	R4	R1	R2	R3	R4
free:	1	1	1	2				

Next, P2 can run to completion, since it needs one instance of R1 and two of R4, and these resources are available. This leads to the following state:

processes	allocation				max-need			
P1:	1	4	2	0	1	7	5	0
P3:	0	0	1	4	0	6	5	6
	R1	R2	R3	R4	R1	R2	R3	R4
free:	2	4	6	6				

Next, P1 can complete:

```
processes       allocation       max-need
-----------------------------------------
P3:             0  0  1  4      0  6  5  6
-----------------------------------------
                R1 R2 R3 R4     R1 R2 R3 R4
-----------------------------------------
free:           3  8  8  6
-----------------------------------------
```

Finally, P3 can complete.

2. Here is a solution:

```
#define NumberOfPhilosophers 4;
main () {
    binary semaphore fork[NumberOfPhilosophers];

    /* initialize semaphore */
    for (i=0; i < NumberOfPhilosophers; i++) fork[i].value = 1;

    parbegin
        philosopher(1);
        philosopher(2);
        philosopher(3);
        philosopher(4);
    parend
}

philosopher (int i) {
  while (1) {
    if odd(i) { /* odd philosopher picks up left fork first */
        wait(fork[i]);
        wait(fork[(i+1) mod NumberOfPhilosophers]);
    } else { /* even philosophers picks up right fork first*/
        wait(fork[(i+1) mod NumberOfPhilosophers]);
        wait(fork[i]);
    }
    <eat>
    signal(fork[i]);
    signal(fork[(i+1) mod NumberOfPhilosophers]);
    <think>
  }
}
```

This program is deadlock-free. Indeed, suppose p(i) needs and cannot get a fork. Assume i is odd. Then p(i) must already hold fork i (the left fork of p(i)) and needs fork i+1 (mod NumberOfPhilosophers). If fork i+1 is unavailable, p(i+1) must be holding it. But i+1 is even, so p(i+1) can get fork i+1 only after it already has fork i+2. Thus, p(i+1) has all the forks it needs. When p(i+1) finishes, p(i) can get fork i+1 and proceed. Thus, p(i) cannot be involved in a deadlock.

A similar argument shows that p(i) cannot be deadlocked if i is even.

3. The monitor can be specified as follows:

```
monitor Alarm
{
    condition timer; /* condition variable */
    extern current_time();

    /* monitor entry points */
    public sleep(int interval)  { // interval  to sleep
        /* Note: the argument to wait is priority assigned to the calling
        ** process in the wait queue. Processes are waken up according to their
        ** priority. */
        timer.wait(current_time() + interval);
    }

    public wakeup() {
        timer.signal;
    }
}
```

Each process then looks like this:

```
...
Alarm.sleep(how_long);
...
```

4. The following table shows how I/O requests will be scheduled:

Track#	0	5	15	20	25	30	40	50
I/O request time	-	5	110	50	300	70	500	355
Head arrived at	0	35		130		190		
I/O request done	-	40		135		195		
Head leaves	5	50		135		195		
Head arrived at			275		355			490
I/O request done			280		360			495
Head leaves			300		360			500
Head arrived at							555	
I/O request done							560	
Head leaves							-	

   Thus, the order of the served requests will be: 5, 20, 30, 15 25, 50, 40.

5. The DMA module transfers 1200 bytes/sec, so it needs a memory cycle every 1/1200 sec. The CPU needs a memory cycle every 1/1000000 sec. Thus, CPU must loose a memory cycle every (1/1200)/(1/1000000) = 833 cycle. Thus, the slow-down is 100% x (1/833) = 0.12%.

6. This idea is feasible, in principle, provided that the programs are *reentrant*, *i.e.*, their execution does not modify their code. For instance, static variables in C would not be allowed. Besides, storage for the program code is not the whole story. When program executes, it has a run-time stack that contains the current values of local (non-static) variables. This stack needs to be saved somewhere in virtual memory, and this place must be different from where the program binary is located. (Why?) So, even with reentrant programs, it is not possible to completely avoid additional backing storage for virtual memory.

7.
```
monitor genSemaphore {
     int value;
     condition empty;

     public set(int value) {
         value = v;
     }
     public wait () {
         value--;
         if (value < 0) empty.wait;
     }
     public signal () {
         value++;
         if (value =< 0) empty.signal;
     }
}
```

8. Use one binary semaphore, monMutex, to control mutual exclusion inside the monitor. For every conditional variable (say, condvar) in the monitor, use a binary semaphore condvarMutex. Then, each procedure in the monitor should look like this:

```
public whatever (...) {
    monMutex.wait;
    ............
    monMutex.signal;
}
```

The operation `condvar.wait` should be implemented as "`condvarWait();`". the operation `condvar.signal` should be implemented as "`return condvarSignal();`", where:

```
condvarWait() {
    monMutex.signal
    condvarMutex.wait;
    monMutex.wait
}
condvarSignal() {
    condvarMutex.signal;
    monMutex.signal;
}
```

9. P3's requests can be satisfied. When this process is finished, the row of available resources will become: (2,2,2,0). This means that P2's requests can be satisfied. Thus, there is no deadlock.

10. Disk striping helps when concurrency is low and each process issues I/O requests for large chunks of data. In this case, striping can help parallelize each individual I/O request. On the other hand, when concurrency level is high, striping might be a bad idea, since it prevents simultaneous scheduling of separate I/O requests coming from different processes.

11. Since each block pointer takes 8 bytes, we have 8K/8 = 1K pointers per block. The 12 direct pointers cover 12*8K = 96K of the file. The first indirect block covers 1K pointers * 8K = 8MB. The second indirect block covers 1K*1K*8K = 8G. Since we are requesting access within 15M range, this clearly falls within the range of the doubly-indirect block. Thus, we need two disk accesses to get through double indirection plus one access to get to the actual data. Three disk I/Os in all.

12. Reliability: if one link is corrupted, the rest of the file is lost.
    Efficiency: no easy way to efficiently access a given byte within a file directly (without the sequential scan of the file). No simple way to scan backwards (unless the chain of blocks is doubly linked).

13. Updating an indexed file always involves changing the index as well (if the search key of the record is changed). With index-sequential files, the index does not need to be changed each time (only sometimes). With hashed files, no auxiliary structures need to be changed at all. Thus, hashed files are the easiest to update, then comes index-sequential, then indexed. If the update does not involve changing the search key of the record then there is no difference among these methods efficiency-wise.

14. Hashed files take the least number of I/O. Index-sequential files might take fewer disk accesses than indexed files, because they require smaller indices than indexed files.

15. The OS maintains a queue of timer request blocks. Each block should contain an id of the requesting process and the time interval (since the previously scheduled timer interrupt) when the interrupt must occur. When a new timer request comes in, the corresponding request block is inserted into the queue in the right place (possibly resetting the timer [if the newly requested interrupt is to occur prior to the current value of the timer] and adjusting the time interval in the next request block).

    When timer interrupt occurs, the next request block is pulled out of the timer queue and the timer is reset to the value specified in that request block.

```
timer          RB1: pid=453        RB2: pid=59848      RB3: pid=980
 curr val=100       interval=25          interval=40         interval=10
current RB: pid=98
            interval=600
```

In the above situation, the timer is processing the current request coming from process 98. The request was for 600 ms, and there are 100 ms left before the interrupt. When this interrupt occurs, RB1 is snatched and the timer is reset to 25ms. Thus, the second interrupt will occur in 125ms. Likewise, the third interrupt will occur in 165ms, etc.

Suppose that RB1 arrived 100 ms ago and RB2 arrived 200 ms ago. What were the wakeup intervals specified in those requests (relative to the time when these requests were issued)?

Since RB1 arrived 100ms ago and its requested interrupt is due 125ms from now, the requested wakeup interval must have been 225ms. For RB2, it must have been 365ms.

16. Each letter corresponds to 10 ticks of execution of the corresponding process. "?" means that CPU is idle during the corresponding 10 tick interval:

    Earliest deadline: AA?CCEEDD

    Note that B misses its starting deadline of 30, because A is executing between the ticks 10 and 30 and the scheduling strategy is non-preemptive.

    Unforced Earliest Deadlines: ??BBCCEEDDAA

    Here, A is not scheduled immediately because its deadline is far away.

17. In real time systems with periodic tasks the system can predict future times of job arrival. In such a case, the scheduler can make global analysis and create a schedule with deliberately arranged idle times and better service characteristics (as in the previous problem).