Question 1. (10 points) (Caches)

Suppose we have a memory and a direct-mapped cache with the following characteristics.

- Memory is byte addressable
- Memory addresses are 16 bits (i.e., the total memory size is 2^{16} = 65536 bytes)
- The cache has 8 rows (i.e., 8 cache lines)
- Each cache row (line) holds 16 bytes of data

(a) In the spaces below, indicate how the 16 address bits are allocated to the offset, index, and tag parts of the address used to reference the cache:

____**4**____ offset bits

____**9**____ tag bits

____**3**____ index bits

(b) Below is a sequence of four binary memory addresses in the order they are used to reference memory. Assume that the cache is initially empty. For each reference, write down the tag and index bits and circle either hit or miss to indicate whether that reference is a hit or a miss.

Memory address	Tag	Index	Hit / Miss (circle)
0010 1101 1011 0011	**0010 1101 1**	**011**	Hit (Miss)
0000 0110 1111 1100	**0000 0110 1**	**111**	Hit (Miss)
0010 1101 1011 1000	**0010 1101 1**	**011**	(Hit) Miss
1010 1010 1010 1011	**1010 1010 1**	**010**	Hit (Miss)

(Goofy placement of circles in the last column courtesy of Word 2007, which insists on moving the underlying words in strange ways and won't allow the circles to be placed arbitrarily.)

Question 2. (6 points) (hardware) Different storage devices have different access times and costs. For each of the following devices, circle the access time that is closest to the typical time needed to read information from that device given current hardware technologies. The important thing is to get the order of magnitude right; the exact number is not the issue. (ms = millisecond, μs = microsecond, ns = nanosecond). If it matters, assume that these numbers are for a computer with a 1GHz clock (i.e., a low-end current machine)

Main memory (RAM):	1sec	100ms	10ms	1ms	100μs	10μs	1μs	**100ns**	10ns	1ns
Hard disk drive:	1sec	100ms	**10ms**	1ms	100μs	10μs	1μs	100ns	10ns	1ns
CPU register:	1sec	100ms	10ms	1ms	100μs	10μs	1μs	100ns	10ns	**1ns**

Question 3. (8 points) (cache organization) Two of the design choices in a cache are the row size (number of bytes per row or line) and whether each row is organized as a single block of data (direct mapped cache) or as more than one block (2-way or 4-way set associative).

The goal of a cache is to reduce overall memory access time. Suppose that we are designing a cache and we have a choice between a direct-mapped cache where each row has a single 64-byte block of data, or a 2-way set associative cache where each row has two 32-byte blocks of data. Which one would you choose and why? Give a brief technical justification for your answer. If the choice would make no difference in performance then explain why not.

Two blocks of 32-bytes each should give better performance. The main reason is that it will reduce conflicts between cache lines that have the same index bits but different tags. It also will probably avoid bringing in extra data in the wider cache line that is less likely to be used because it has lower spatial locality, and that will reduce the memory traffic.

Question 4. (8 points) (threads and processes) In a system with both processes and threads, context switches occur both between threads in the same process and between processes. Below is a list of items that are often stored in Thread Control Blocks (TCBs) or Process Control Blocks (PCBs) or both. For each item, if the item is loaded or saved to/from a processor register during a process or thread context switch, put an X or a check mark in the respective column. If it is not loaded or saved, leave the column blank.

Item	Saved/restored during process context switch	Saved/restored during thread context switch
Program counter register (PC)	**X**	**X**
Stack pointer register (SP)	**X**	**X**
General purpose registers	**X**	**X**
User name of owner		
Scheduling priority		
Processor register containing address of page table	**X**	
Information about open files		

Question 5. (8 points) (threads) A common model for implementing threads is a user-level thread package where a single kernel thread is shared among all the threads in a user address space. Using this model, if the running user thread performs an I/O operation that blocks the kernel thread, no other user thread can run, even if other user threads have work available that they could do. Give two different ways that this model can be changed or extended to allow other user threads in the same address space to continue executing even if one thread performs a blocking I/O operation.

Some possible strategies:

- **Provide more than one kernel thread per process address space. Then if a user thread blocks its kernel thread, other kernel threads can still run other user threads.**
- **Add some sort of communication between the user thread package and the kernel, so the kernel can notify the thread package when the kernel thread blocks and then the thread package can then select another thread to run.**
- **Similar to the above, add a timer interrupt so that the user thread package receives control from the kernel periodically and can reschedule the user threads if desired**
- **Add some sort of mechanism so that a user thread that wants to do a blocking I/O operation can yield the kernel thread to another thread as part of the I/O operation.**

Question 6. (8 points) (synchronization) Both semaphores and monitors provide a method of synchronizing concurrent access to data shared by multiple threads. Why would anyone want to use monitors instead of semaphores? What advantage(s) is (are) there to doing this?

The big advantage is that a monitor makes explicit the association between the lock or semaphore and the data it is guarding. The compiler is able to automatically insert the appropriate lock or semaphore operations needed to guarantee atomic access to the monitor data and not have to depend on the programmer to get this right. Further, the compiler can guarantee that no code accesses the monitor data unless it uses appropriate monitor operations, with the associated implicit locks.

Question 7. (8 points) (process scheduling) In most priority schedulers, a thread's priority is increased and decreased dynamically over time.

(a) Why would a typical scheduler increase the priority of an executing thread? What problem(s) would be solved by doing this?

Primarily to avoid resource starvation for a thread that has a low priority and is constantly preempted by higher priority threads. If we increase the priority of a thread over time, it eventually will have a high enough priority to be run.

(b) Why would a typical scheduler decrease the priority of an executing thread? What problems(s) would be solved by doing this?

This will avoid having a long-running thread consume an excessive amount of available processor time. By lowering the priority of the thread, other threads, particularly short or interactive ones, can get processor cycles, and this should increase overall system responsiveness.

Question 8. (10 points) (synchronization) Your friend Ben Bitblitter is trying to implement locks to use with concurrent processes. In Bitblitter's implementation, a lock is a variable whose value is 0 if the lock is free and is the non-zero process id number of the process that holds the lock if it is in use. Bitblitter proposes to implement the lock operations as follows (using the pseudo-notation from the lecture slides, which doesn't necessarily match any specific programming language, but is similar to C or Java).

```
struct lock {          /* lock data structure:          */
  int owner = 0;       /*    lock owner process id or 0 */
}                      /*    if not locked              */

void acquire(lock) {
  pid = get_process_id();       /* get process id */
  while (lock->owner != 0) { }  /* busy wait */
  lock->owner = pid;
}

void release(lock) {
  lock->owner = 0;
}
```

Bitblitter's implementation is intended to run on multiprocessor machines, so there is no problem using a busy wait in the implementation of `acquire`. But even so, his code doesn't always work. Most of the time it behaves properly, but every now and then something goes wrong.

(a) What is the bug in the code? What can go wrong? Your answer does not need to be long, but it does need to be precise.

The trouble happens if there is a context switch to another thread in the middle of executing `acquire` between the end of the while loop and the completion of the assignment that stores the pid in the lock's owner field. If that happens, the first thread will proceed as if the lock were free when it resumes execution, even if another thread acquired the lock in the meantime.

The same problem can happen in a multiprocessor system if another processor accesses the lock between the while loop and the assignment.

(b) What needs to be changed for this code to work properly? How might we do this? (Recall that this code is intended to work on a multiprocessor system, if that makes any difference.)

The while loop and the following assignment need to be executed as an atomic critical section. The most likely way to do this is to use an atomic read-modify-write hardware instruction like compare-and-swap or test-and-set, that can guarantee that no other process or processor can interrupt the critical section. It could also be done by globally disabling interrupts or something similar, but this is not a great idea since it can drastically reduce performance, particularly in a multiprocessor.

Question 9. (8 points) (concurrency) Three painters are working on repainting an old house. They all are working part-time, so often only one or two of them are around.

Because of budget cuts, only two buckets of paint and two paintbrushes are available. When a painter shows up to work, he grabs one of each and starts painting. However, if he can only grab a bucket or a brush but not the other, or if neither is available, he grabs what he can get and then waits around until one of the painters who is working finishes and frees up a brush and a bucket. Painters are stubborn and once they have either a bucket or a brush they will wait indefinitely for the other to become available so they can paint. But once a painter gets both a brush and a bucket, he will start working and continue until he is done, without worrying about what the other painters are doing.

Is it possible for deadlock to occur in this situation, where at least one painter is waiting indefinitely for either a bucket or a brush or both? Describe how this could happen, or explain why it will never occur.

This cannot deadlock.

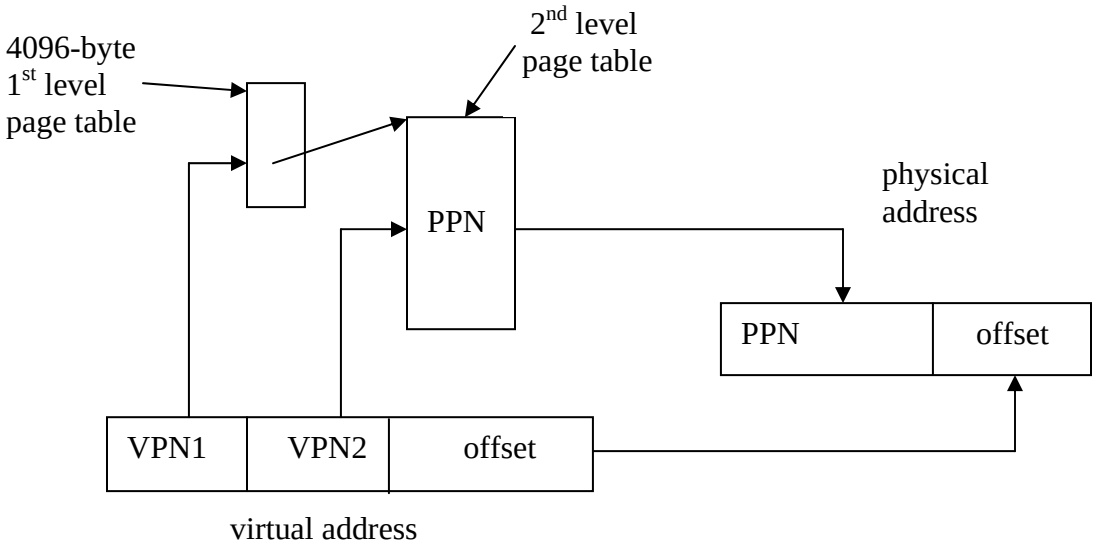
The easiest way to see it is that there are 3 painters and 4 things (buckets and brushes). In the worst case, each painter grabs something, but one of them will get two things, and that will be a brush and a bucket. The other two painters are sitting around waiting for either a brush or a bucket, but the system is not deadlocked. Eventually the painter who does have a brush and a bucket will finish and the other two will be able to get to work.

(If we had 4 or more painters the system could deadlock, since then it would be possible for 4 painters to have only a brush or bucket and none of them could paint since they all would be waiting for the other implement.)

Question 10. (10 points) (VM) To avoid excessively large page tables, many virtual memory systems have multi-level page tables. For this problem, we would like to figure out the details of a 2-level paging system with the following characteristics:

- Both virtual and real (physical) memory addresses are 32 bits each
- Pages are 4096 bytes each
- Page table entries are 4 bytes each, containing the physical page frame number, plus additional control bits including the valid bit.
- The top-level page table consists of a single 4096-byte page and has as many page table entries as will fit.

The page table structure and address translation can be diagramed like this (similar to the multi-level page table diagram in the slides).



Fill in the blanks with the correct numbers to match the given constraints:

Number of entries in the first level page table _____**1024**_____

Number of bits in the VPN1 part of the virtual address _____**10**_____

Number of bits in the offset field of each address _____**12**_____

Number of bits in the VPN2 part of the virtual address _____**10**_____

Number of page table entries in each 2nd level page table ___**1024**_____

Question 11. (8 points) (VM and caches) Most memory systems contain both a translation lookaside buffer (TLB) used by the virtual memory system, and a cache memory. Both the cache and the TLB have similar functions: they store main memory data in a faster "cache-like" memory that can be accessed at nearly processor speeds.

The question is why do these two separate mechanisms exist when they have such a similar purpose? Why have both a cache and a TLB? Why not just a single "super-cache" that would hold both regular memory data as well as frequently used page table entries?

Hint: What is different about these two fast memories, if anything? How are they used in the memory system?

Although the TLB and the cache both hold copies of main memory data, they are used in different parts of the memory subsystem, and accesses to them have different characteristics.

Each entry in the TLB contains a pair of addresses: a virtual page number and a page frame address. Because individual TLB entries translate a large number of virtual addresses (a full page each), relatively few of them are active at any time. So a TLB tends to be fairly small, and the hardware to look up a virtual page number is fully associative.

The regular cache, on the other hand, has much more data per row to exploit locality of instruction and data memory references. It also tends to be much larger than a TLB to get a reasonably large hit rate, so it is not practical to use fully associative addressing. Instead, most caches tend to be 2-way or 4-way set associative.

A TLB miss and a cache miss are also handled in quite different ways. A TLB miss requires walking the page table; a cache miss does not use a secondary data structure.

A final difference is that the TLB and the cache are used in different parts of a memory access. A TLB uses the virtual address to locate data, while a cache normally uses the physical (translated) address as the index and tag.

Because of these differences it makes sense to use specialized hardware optimized for each function rather than having a single "super cache".

(Note: Answers did not need to be nearly this detailed to get credit as long as it was clear that they did understand the main issues. Scores were based on overall quality of the answers and absence of misinformation.)

Question 12. (8 points) (file systems) The classic Unix file system stores information about files in various places: the file system superblock, file i-nodes, and data blocks that contain file and directory data.

For this question, place an X in the correct column showing where each of the given pieces of information are stored in a classic Unix file system.

What	Superblock	i-node	directory data block	file data block
File name			**X**	
File owner name		**X**		
File permissions (owner, group, others)		**X**		
Disk location of first data block	*****	**X**		
Disk location of start of i-node section on the disk	**X**			
File creation date		**X**		
File data (text files, databases, etc.)				**X**
File link count (number of directory links to a file)		**X**		

***It turns out that this part of the question was slightly ambiguous. It is true that the superblock points to the start of the data area on the disk, but the i-nodes point to the first data blocks for individual files. We gave credit for either although the later was intended, and probably what most people interpreted it to be.**