CS 167 Final Exam Solutions

Spring 2018

Do all questions.

1. *[20%] This question concerns a system employing a single (single-core) processor running a Unix-like operating system, in which interrupts are handled on the current thread's kernel stack. We have a user-level process in which two threads are running. One (the producer) is reading characters from standard input and depositing these characters into a buffer. The other thread (the consumer) is taking characters from the buffer (in the same order as the characters were put into the buffer) and writing them to standard output. Synchronization on the buffer is done with semaphores, using the standard producer/consumer pattern. Note that if a thread is waiting on a P (or wait) operation, it blocks, i.e., sleeps. Assume that standard input comes from the keyboard, and that it is set up so that each time a character is typed in, an interrupt is generated. The interrupt handler puts each incoming character into an input buffer in the OS kernel (different, of course, from the user-space buffer the two threads are using). Something similar may be happening for standard output, but we're not concerned about it for this problem.*

   The OS kernel's input buffer is of finite size. The interrupt handler puts characters into it. Characters are removed when the producer thread issues read system calls. In the current implementation, when the OS kernel's input buffer is full, the interrupt handler must discard all typed-in characters.

   A suggested improvement is to use producer-consumer synchronization on the OS kernel's input buffer: the interrupt handler waits, using a P (wait) operation on a semaphore if the buffer is full. Explain why this might lead to deadlock.

   Recall that semaphores are non-negative integers. The P (wait) operation on a semaphore S is defined as:

```
when (S > 0) [
  S = S-1;
]
```

   The "when(exp) [...]" notation means that at some point when exp evaluates to true, the sequence of statements inside the square brackets is executed atomically. If exp is false, then the thread waits by blocking. The V (post) operation on a semaphore S is defined as:

```
[S = S+1;]
```

   The square brackets mean that the statement inside them is executed atomically. The V operation wakes up blocked threads waiting for the semaphore to become positive.

   Suppose, when the input buffer is full, a character is typed in while the producer thread is running. Thus the interrupt handler executes on this thread's kernel stack. Since the input buffer is full, the interrupt handler waits for it to empty by invoking the P operation on the semaphore. The only thread that can empty the input buffer is the producer thread. However, it cannot run because the interrupt handler is using its stack. The interrupt handler won't release the stack until the input buffer is emptied. Thus we have a deadlock. Note that deadlock could also happen if the consumer thread is interrupted when both the user-space buffer and the input buffer are full, and the producer thread is waiting for the user-space buffer to be emptied. But one potential deadlock is sufficient to make the solution unusable.

2. *An operating system has a simple round-robin scheduler used in conjunction with time slicing: when a thread's time slice is over, it goes to the end of the run queue and the next thread runs. The run queue is implemented as a singly linked list of threads, with pointers to the first and last threads in the queue. Assume for parts a and b that we have a uniprocessor system.*

   a. *[5%] The system has a mix of long-running compute threads that rarely block and interactive threads that spend most of their time blocked, waiting for keyboard input, then have very brief bursts of using the processor. It's known which threads are compute threads and which threads are interactive. Assuming we want the system to have good interactive response, explain what is wrong with the scheduler.*

      The interactive threads and the compute threads are treated equally. Thus when an interactive thread wakes up, it will likely have to wait in the run queue for an appreciable period of time while the compute threads in front of it each run for their complete time quanta. This means that interactive response will not be good.

   b. *[5%] How might the scheduler be improved to provide good interactive response? (Hint: a simple improvement is sufficient.)*

      We could have two separate run queues: one for compute threads and one for interactive threads. The next thread to run is taken from the interactive-thread queue if it's non-empty, otherwise it's taken from the compute-thread queue.

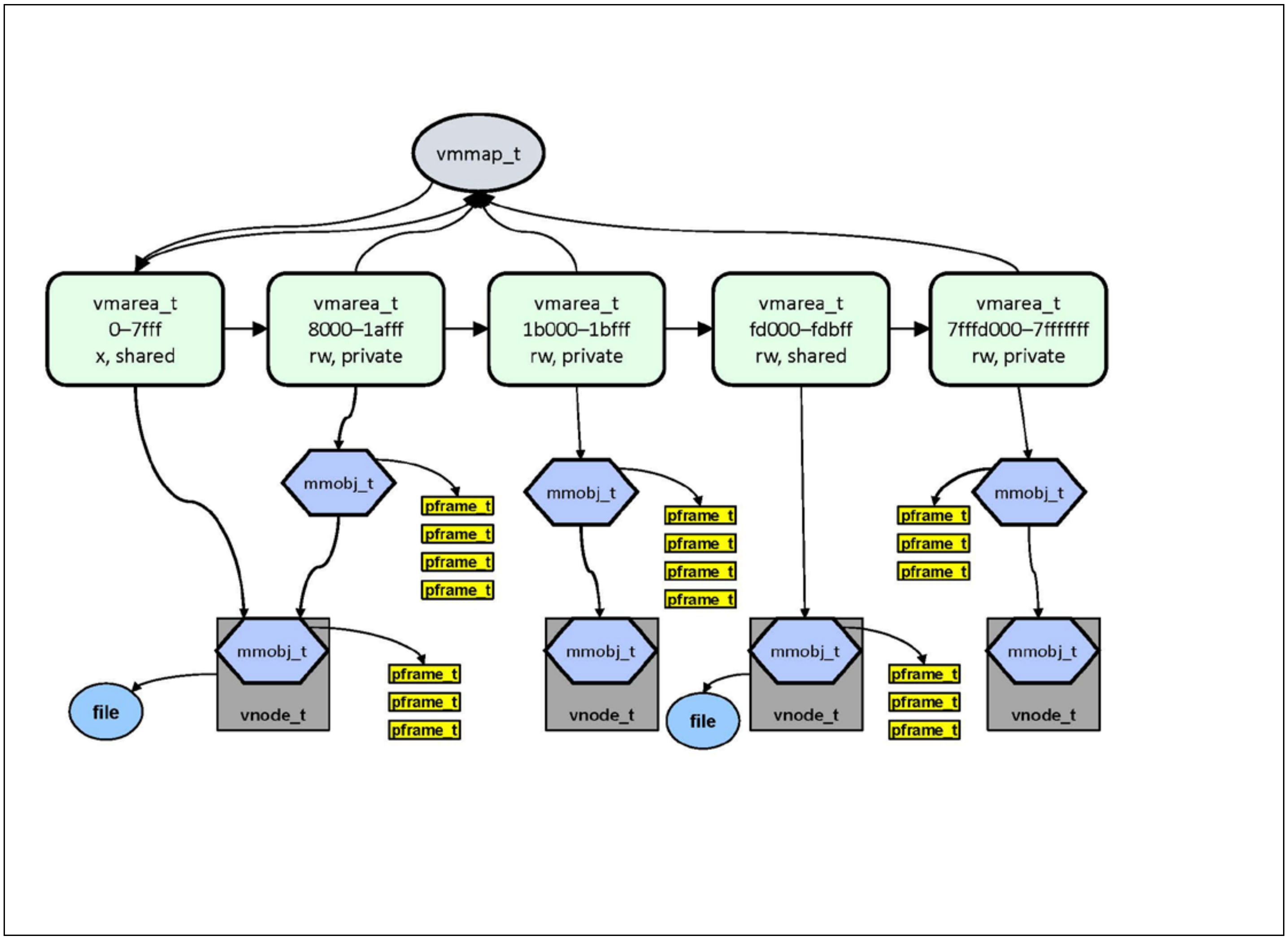
   c. *[5%] We add three more processors to our system and add the appropriate synchronization (spin locks) to our scheduler data structures. But this may result in a performance issue due to the multiple processors. Describe this performance issue.*

      There will be contention for the spin locks if processors simultaneously attempt to remove the next threads from the run queues.

   d. *[5%] Describe what might be done to alleviate these performance problems, yet still have reasonable parallelism (by which we mean that, in general, threads will not be in run queues waiting for processors while at the same time there are idle processors).*

      We provide a separate pair of queues for each processor. Only when a processor's queues are empty does it attempt to remove threads from the queues of other processors. In addition, to make certain that the load is well balanced, each processor periodically compares its load with the others and moves some of its threads to the queues of other processors if necessary to achieve reasonable load balancing.

3. *In this problem we consider how virtual memory is represented in Weenix. The representation of a process's address space is given below, as it appeared in a lecture slide. Each vmarea_t structure represents a portion of the address space into which a file or some other object has been mapped. Each mmobj_t structure represents some sort of memory object. Those at the bottom represent files or other sources of pages. Those between the vmarea_t structures and the bottom mmobj_t structures are shadow objects that represent private changes made to pages originally from the bottom objects. Associated with each mmobj_t is a possibly empty list of pframe_t structures that refer to page frames holding pages associated with the object, along with an indication of which pages they represent (i.e., a mapping from page to page frame).*


vmmap_t
vmarea_t 0–7fff x, shared
vmarea_t 8000–1afff rw, private
vmarea_t 1b000–1bfff rw, private
vmarea_t fd000–fdbff rw, shared
vmarea_t 7fffd000–7fffffff rw, private
mmobj_t
pframe_t
vnode_t
file

a. *[4%] The Weenix data structures depicted above define a mapping from virtual to real memory. The IA32 hardware doesn't use the Weenix data structures, but uses page tables, which can represent the same mapping that the Weenix data structures do. Is it necessary that all of the mapping represented by the Weenix data structures appears in the page tables? Explain. (I.e., if a virtual-address-to-real-address mapping appears in the Weenix data structures, must it also exist in the page tables?)*

   No. The translations must be in the page tables only when needed to satisfy memory references. If a needed translation is not in the page table, a page fault will occur. The OS page-fault handler can then look up the translation in the OS data structures and load it into the page table.

b. *[4%] Given the above figure, suppose the process references an instruction at location 0x4000, and the page containing this instruction has not been previously referenced by the process. Will it necessarily be the case that the appropriate page must be read in from the underlying file before it can be mapped into the process's address space? Explain.*

   No. It's possible that some other process is using the same file and the necessary page is already in memory. If so, a pframe_t structure for the page will be in the list associated with the file's mmobj_t, which can then be used to map the page frame into the process's address space.

c. *[4%] Suppose the process modifies a word of memory at location 0x10000, and the page containing the word has not been referenced before by the process. Explain, in terms of the diagram where relevant, what the initial contents of the page frame will be and where its associated pframe_t structure will be linked.*

The page frame allocated to hold the page will be initialized with the current copy of the page from the file (which may already be in memory if in use by another process). The pframe_t structure associated with the page frame will be linked into the list of pframe_t's of the mmobj_t that the vmarea_t is directly linked to.

d. *[4%] Suppose the page in question for part c had been referenced by the process (but not modified) before the modification of part c. What would the access-permission bits (some combination of r, w, and x) in the page-table entry have been after this reference, but before the reference of part c? What would they have been after the reference of part c? (Hint: the object is privately mapped.)*

Beforehand they would have been read-only. Afterwards they would be read-write.

e. *[4%] Now suppose the process references memory at location 0x1000000. Explain what happens, including how this affects the process.*

This would generate a page fault because the page is not mapped into the process's address space. The page-fault handler would determine, from the vmarea_t structures, that this is not a legitimate reference. It would arrange for the process to be sent a SIGSEGV signal (i.e., a seg fault).

4. *[15%] Tenex was an operating system for DEC PDP-10 computers used in the late '60s and early '70s. It had a number of features, including one that allowed user code to be invoked in response to each page fault. It stored passwords in plain text (i.e., unencrypted) in a file that was adequately protected. A user could supply his or her password not only when logging in, but also from a program so as to switch from one protection domain to another. The system code that checked for a correct password would do so in place, one character at a time, moving from left to right, stopping when it encountered an incorrect character. It was soon discovered that it was relatively easy to figure out any user's password. Explain how. (Hint: the time required was linear in the length of the password.)*

The technique allows a password to be determined using a number of guesses that are linear in the length of the password, rather than the straightforward approach, which is exponential in the length of the password. One started by putting one's first guess of the password in a buffer in memory such that there was a page boundary between the first and second character. One then arranged so that page containing the first character was in memory, but the page containing the second and subsequent characters was not. One then supplied passwords containing all possible choices for the first character. If the user-provided page-fault-response code was called, then one knew that the first character had been guessed correctly and the system had moved on to checking the second character. One then would shift the buffer left one byte and try again with the second character. Thus, using a relatively small number of guesses, one could determine a password.

5. *You've been hired by Microsoft to fix CIFS so that it's tolerant of failures. Your boss tells you that for this special version of CIFS, the plan is to turn off opportunistic locks. Thus there will be no caching on the clients and all state will be held on the server. "State" in this case means both the open state, i.e., which clients have which files open, and the lock state, i.e., which clients have which files locked and what sort of locks (read or write) they have. Assume that files are locked in* *their entirety — locks don't apply to byte ranges, but to the entire file. Also assume that computers (both clients and servers) are either up and running bug-free code, or are down and not responding at all. Computers may, however, occasionally run very slowly. The network itself could fail, leaving some clients disconnected from the server.*

You've decided to implement crash recovery in much the same way as it's done in NFSv2's network lock manager protocol. After a client crash, when the client comes back up, it notifies the server that it has just rebooted, causing the server to nullify the client's state information. After a server crash, when the server comes back up, it notifies its clients that it has rebooted and gives the clients a grace period during which they inform the server of whatever state they had at the moment the server crashed. Assume that it's not possible for clients to upgrade or downgrade locks — once they have, say, a write lock, they keep the write lock until they release it. They may not downgrade it to a read lock. Also assume that clients will respond to a newly rebooted server before the grace period is over, unless the client has crashed.

a. *[5%] Suppose state information on the server is kept in volatile storage (i.e., it disappears after a crash). However, the lists of clients of a server, and servers of a client, are kept in stable storage, which survives crashes. Nothing else about a server's clients or a client's servers is kept in stable storage. Is the crash recovery procedure sufficient to guarantee to clients that if they lose contact with the server (for whatever reason: server crash or network outage), that when contact is restored, they may continue to operate as if nothing had happened? If so, this would mean that no crash recovery of any form is required by application programs running on the client. Explain your answer. (Hint: the correct answer is no.)*

No. Consider a situation in which client A has a write lock on a file. The network connection between client A and the server is lost. Shortly thereafter, the server crashes. The server reboots. Client B obtains a write lock on the file, modifies the file, then crashes (and disappears, never to be seen again). The server crashes again. It reboots and, simultaneously, client A's network connection with the server is reestablished. During the server grace period, client A reclaims its lock on the file. Client A now assumes, wrongly, that no changes have been made to the file since the time it originally obtained a write lock on it.

b. *[5%] Suppose state information on the server is kept in stable storage. Does this change your answer to part a? (I.e., is it the case that clients may be unconcerned if they temporarily lose contact with the server.) Explain. (Keep in mind that a server won't consider a client to have crashed until it is notified by the client of its rebooting.)*

Yes. Since client state is maintained, essentially forever, on the server until the server hears from the client, the server will honor the client state information until it hears from the client that it has rebooted. Thus the server will not grant conflicting locks to other clients, nor allow other clients to perform operations that conflict with the state of the client. Thus clients are assured that they may continue to execute without concern about network and server problems.

c. *[5%] You discover that many of the users of client machines have a nasty habit of powering off their machines before going home for the night, despite the fact that their machines hold a number of locks. Thus it may be hours before the machines reboot; and thus the servers may not give locks conflicting with the down client's to others during this period. You decide to deal with this by having locks timeout after a few minutes: if a server hasn't heard from a client for, say, three minutes, it may unilaterally revoke its locks. This turns out to work fine for dealing with client crashes, but introduces some new issues in situations in which there are server crashes or network outages. In particular, it* *now might be possible that a client has not crashed, but while it is out of touch with the server, its locks were given to other clients. If the client contacts the server claiming to have a lock, the server will tell it that it no longer has a lock. However, the server might itself crash, and the client might contact it during its grace period and attempt to reclaim the locks it had when it lost touch with the server. Keeping in mind that servers keep their state information in stable storage (and you get to define what's state), is it possible for the server to determine that the client's locks had been revoked and thus it should notify the client of this when it attempts to reclaim its locks?*

Yes. The server can keep track of the fact that the client's lock had been revoked as part of its state information.

d. *[5%] Your boss points out that it slows things down too much to keep all server state information in stable storage. She says that you must no longer keep information about locks in stable storage, but that it's ok to keep things there that don't change very often. You recall reading about the approach used in NFSv4 in which the server keeps in stable storage, for each active client, the client ID, the time of the client's first acquisition of a share reservation of lock after a server reboot or client lease expiration, a flag indicating whether the client's most recent state was revoked because of a lease expiration, and the time of the last two server reboots. You decide to do the same in your implementation of CIFS. You decide that this gives pretty good results, but that it doesn't do quite as good of a job as when the server kept everything in stable storage. Give an example of a situation that this NFSv4-based approach cannot handle, but that could be handled if the server keeps everything in stable storage.*

Suppose the client loses contact with the server due to a communication failure and the server subsequently crashes and restarts twice. However, no other client has attempted to take a lock that conflicts with those of our first client. The NFSv4-based approach would conclude that the client's locks have been revoked. But if all lock information were in stable storage, the server could determine that the locks had not been revoked.

e. *[5%] Your boss praises you for your work so far, but tells you that management has decided that opportunistic locks are actually pretty important and that they should be supported. Thus client caching of data should be done when feasible. You, feeling upbeat, tell her "no problem". After thinking about it for a bit, you decide that it really isn't much of a problem. What approach do you use to handle opportunistic locks and client caching? Hint: the server may need to return error messages to the client in certain situations. Recall that an opportunistic lock is given to a client when it is the sole user of a file and thus may safely cache its contents. Such locks are revoked if another client wants to access the file. You don't need to discuss write-backs of client-cached data to the server.*

You treat op locks essentially as normal locks, using the same rules as already discussed. Thus a server will unilaterally revoke them if it gets no response from the client. If the server crashes, clients may reclaim their op locks during the server's grace period. Note that a client may not copy its cached changes to a file to the server if its op lock has been revoked. Thus servers may reject attempts by the client to send changes to it.