CS 167 Final Exam Solutions

Spring 2016

Do all questions.

1. *The implementation given of thread_switch in class is as follows:*

```
void thread_switch() {
  thread_t NextThread, OldCurrent;

  NextThread = dequeue(RunQueue);
  OldCurrent = CurrentThread;
  CurrentThread = NextThread;
  swapcontext(&OldCurrent->context, &NextThread->context);

  // We're now in the new thread's context

}
```

   a. *Suppose we don't have a swapcontext routine to call. Instead, we have to use getcontext and setcontext. Recall that getcontext saves all of the calling thread's registers (including the stack pointer, but not the instruction pointer) into its argument, a pointer to a data structure of type ucontext_t and returns nothing. The routine setcontext restores the registers from its first (and only) argument (of type ucontext_t). Show how to implement thread_switch using getcontext and setcontext. If changes to the thread control block (of type thread_t) are required, indicate what they are.*

```
// the context component of thread_t has been replaced
// with "ucontext_t ctx;"
void thread_switch() {
  volatile int first = 1;
  getcontext(&CurrentThread->ctx);
  if (!first) {
    // the thread has just been "switched to"
    return;
  }
  first = 0;
  // have saved context of CurrentThread, now get into context
  // of the next thread
  CurrentThread = dequeue(RunQueue);
  setcontext(&CurrentThread->ctx);
  // unreached!
}
```

   b. *Suppose thread_switch takes an argument and this argument is referred to after setcontext returns. Explain what, if anything, must be done to ensure that what is referred to is what was passed in the most recent call to thread_switch.*

   Since the argument to *thread_switch* is stored on the stack and the stack is switched to that of *NextThread* by *setcontext*, it is necessary to copy the argument someplace else so that it can be accessed after *setcontext*. One place to copy it is to a field within the *thread_t* structure of *NextThread*.

2. *Recall that in Unix, when a file whose set-user-id bit is one is exec'd, the process's effective user ID and the saved user ID become the owner of the file, while the real user ID is not modified. A program may set its effective user ID to be anything if it's currently superuser, otherwise it may set its effective user ID only to its real, effective, or saved user IDs. Also, file access checks are done using a process's effective UID. A set-UID program uses these features to check if its caller has permission to access a given file, perhaps one passed as an argument. (It first sets its effective UID to that of the caller, opens the file, then sets its effective UID back to the saved UID.) A primary use of these features is to allow the owner of a file to keep others from directly accessing the file, but provide a program (whose set-UID bit is set) that performs operations on the file on the behalf of others, but under the control of the file's owner.*

   a. *This approach works well if the set-UID program is owned by superuser — such programs are effectively extensions to the kernel. However, it's not that useful if the file containing the program is owned by a normal (untrusted) user. Explain why not.*

      Suppose we are running a set-UID-to-Mary program, where Mary is the untrusted (for good reason) user. Even though Mary's program is supposed to access just the file whose name was passed via an argument, since the program may change its effective user ID to the caller, it may access everything the caller can. Mary's program might take full advantage of this for nefarious purposes.

   b. *We'd like to fix this: we want to make it possible for some user, say Mary, to create a program that others may run as set-UID-to-Mary, with none of the problems mentioned in the answer to part a. Explain how this might be done. Hint: certain functionality may have to be turned off, and we may need to adopt a convention for passing files as parameters to Mary's program.*

      We first disable the ability to change the effective UID while running the set-UID program. Rather than have the invoker of a set-UID program provide the names of files it would like accessed, it instead opens the files it wants accessed and provides the file descriptors as, for example, file descriptors 3, 4, 5 … . Thus the file descriptors act as capabilities and provide to Mary just the capabilities the invoker wishes.

3. *An operating system is said to be kernel-preemptible if threads may be preempted, even while running in kernel mode, and their processor switched to another thread. There may, of course, be certain code sequences during which such preemption is disabled. An operating system is said to be SMP-safe (symmetric multiprocessor safe) if it can run on a multiprocessor system in which any thread can run on any processor, whether in user mode or privileged mode, and any interrupt handler can run on any processor.*

   a. *If an operating system is kernel-preemptible, is it necessarily SMP-safe? Explain why not.*

      On a non-MP kernel-preemptible system, one can make certain that two interrupt handlers don't simultaneously access the same data structure by relying on the interrupt mask (or interrupt priority level) to insure that no more than one is accessing the data structure at once. But on an SMP system, even though interrupts may be masked on one processor, they are not necessarily masked on the others. Thus additional machinery is required to protect the data structure.

   b. *If an operating system is SMP-safe, is it necessarily kernel-preemptible? Explain why not.*

      Consider a processor-specific resource, such as a page table, that is manipulated only in the context of a thread running in kernel mode and never in the interrupt context. Thus no

special considerations are required for protecting it on an SMP-safe system. However, the system could assume that threads are not kernel-preemptible, and thus not need any form of synchronization to protect the resource.

4. *Striping is a technique in which files are spread across multiple disks. The striping unit is the block size on each disk; the first striping-unit-sized portion of a file is written to the first disk, the second to the second disk, and so forth. An obvious concern is the size of the striping unit. Assume that data is read from and written to a striped file system in some standard length L. Thus the smallest possible striping unit is the size of a disk sector; the largest useful striping unit is L. You may assume that L is sufficiently large enough that it is worth discussing what the size of the striping unit is.*

   a. *Explain why it makes sense for the striping unit to be small (such as one sector in length) on systems in which just a single thread is generating file-system requests.*

      If the striping unit is small relative to L, transfers to and from the disk array will utilize a number of disks and thus achieve the advantages of parallelism. Successive requests come from the same thread and are likely to be to consecutive locations within a file. Thus seek delays are minimal.

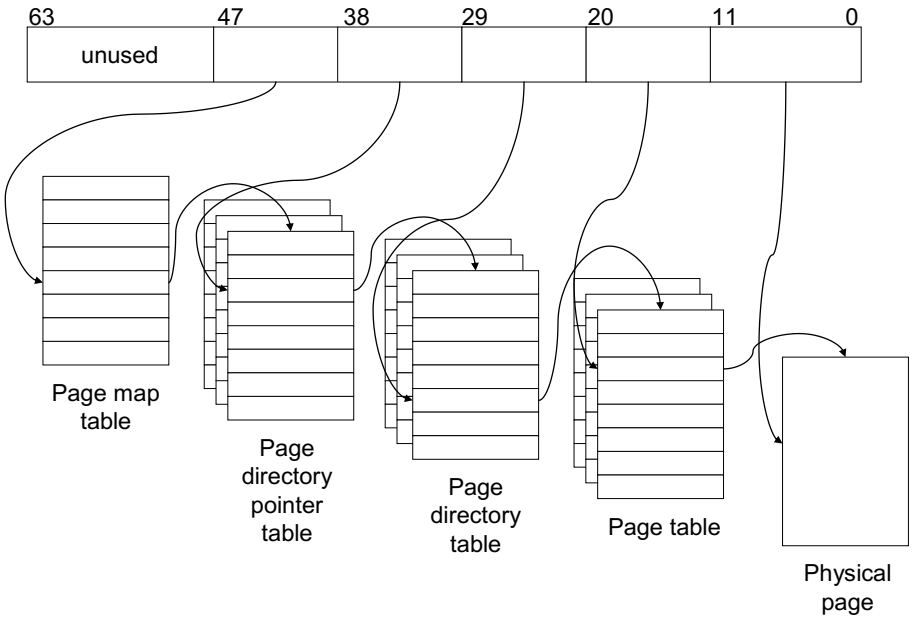
   b. *Explain why it makes sense for the striping unit to be large on busy servers (i.e., on systems in which many threads are generating file-system requests). (Hint: you might answer this by showing why a small striping unit is not good.)*

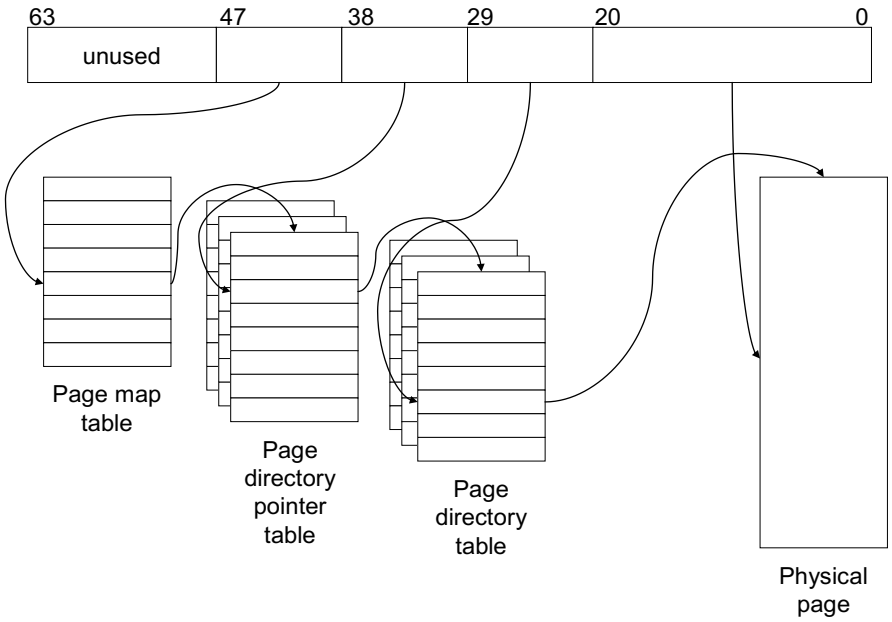
      On a busy server, successive file-system requests come from independent threads and are targeted at essentially randomly scattered disk locations. Thus each request generally requires a substantial seek delay. The number of such seek delays can be minimized by issuing large transfer requests.

   c. *Suppose we replace the disks with flash memory, i.e., we are striping files over a number of flash devices. Would your answers to parts a and b be different? Explain.*

      Yes. Since there are no seek delays with flash memory, the only concern is the amount of parallelism. Thus small striping units would be best.

5. *The x64 architecture supports both 4KB and 2MB pages as shown below.*


63
47
38
29
20
11
0
unused
Page map table
Page directory pointer table
Page directory table
Page table
Physical page


63
47
38
29
20
0
unused
Page map table
Page directory pointer table
Page directory table
Physical page

a. *A little-known Linux feature allows one (if the kernel is properly configured) to map a file into the address space using 2MB pages rather than 4KB pages. What would be the advantage of doing this? Why isn't it always done? (Be sure to consider, among other things, the implementation of fork on Unix and typical use of fork.)*

   The primary advantage of doing this is that many fewer page-table and TLB entries would be required. This would mean less primary storage is required since there would be no need for the lowest-level page table, and, with a limited number of TLB entries, more of the address-space translation could be handled directly by the TLB, thus reducing the number of TLB misses. The disadvantage of doing this is that there would be a greater wastage of memory due to internal fragmentation, since the average wastage would be 1 MB per mapped file with 2MB pages, but only 2KB per mapped file with 4KB pages.

   Unix systems implement fork using copy-on-write techniques, in which the pages of the parent and child are shared until either process attempts to modify a page. At that point, a copy of the page is made for the modifying process. In its typical use, a fork is followed soon after by an exec, and thus not much of a child process's original address space is modified before it does an exec. In such cases, the use of 4KB pages would involve much less copying than the use of 2MB pages.

b. *Suppose the x64 architecture supported 4KB and 4MB pages. Could the two different page sizes coexist in the same address space as the 4KB and 2MB page sizes do in the actual architecture? Explain.*

   No (at least not without radical changes to the architecture). 4KB and 2 MB pages can coexist in the same address space because they share the same format page-map, page-directory-pointer, and page-directory tables. With 2MB pages, the page-directory-table entries point directly to pages, while with 4KB pages, the page-directory-table entries point to page tables, whose entries point to pages. This is made possible because all the tables have 2^9 8-byte entries (occupying 4KB bytes each); thus it takes 9 bits of address to index within a table and 12 bits to index within a 4KB page. If we eliminate the last table, we have 21 bits to index within a 2^{21}-byte page.

   For such a scheme to work with 4KB and 4MB pages, we would have to have tables containing 2^{10} entries each. If we have four classes of tables, as in the current x64 scheme, each with 2^{10} entries, and the smaller page size is still 4KB, the maximum address space would be 2^{52} rather than 2^{48} as it is now.

c. *Consider two possible implementations of fork using copy on write. One, which we call shallow copy on write, involves sharing just the actual page frames, but both processes have their own private copies of the all page tables (including page directory tables, page directory pointer tables, etc.). The other, which we call deep copy on write, involves sharing both page frames and all page tables. Thus with deep copy on write, after fork, both parent and child share the page map table and nothing is immediately copied. When page frames are modified, all page tables (and page directory tables, etc.) that must be modified are also copied. But with shallow copy on write, when a process forks, all page tables, including the page map table, are copied at the time of the fork. It should be clear that, with deep copy on write, forks are cheaper than with shallow copy on write. Why might it be advantageous, nevertheless, to use shallow copy on write? While we're looking for a qualitative and not a quantitative explanation, you should mention any important data structures that might be required in one but not the other.*

The cost of handling a copy-on-write fault will clearly be cheaper with shallow copy on write than with deep copy on write. But an important additional concern is that deep copy on write requires the maintenance of reference counts on the various sorts of page tables, while no such reference counts are required with shallow copy on write.

d. *How might one optimize shallow copy on write to make it much cheaper for the usual case of fork, in which it's followed quite soon by an exec? (Hint: this is yet another application of "lazy evaluation").*

One might avoid creating any page tables at the time of fork, except for a copy of the page map table, creating page tables only as necessary to satisfy references to page frames. Thus the child will create, in response to a page fault, a chain of page tables leading to the desired page frame. Initially, there's just a page map table, all of whose entries are marked invalid. The process's first memory reference results in a page fault, and a chain of three page tables will be created to map the referenced page frame.

6. *NFS v2 and v3 systems allow clients to "hard-mount" remote file systems, so that in the event of a server crash, clients repeatedly retry RPC calls until the server comes back up. If clients held locks on any of the server's files, the NLM protocol recovers these locks for them.*

a. *Assume that all RPC calls are to idempotent procedures and that the only possible failure is a server crash (i.e., clients don't crash and the network is well behaved). Other than timing issues, will server crashes have any adverse effects on clients (assuming the server comes back up)? Explain.*

No. When the server comes back up, clients will reclaim all locks they had prior to the crash. Applications running on the clients will be effectively suspended if they perform operations that require access to the server, but will resume once the server comes back up. No system call will fail due to the server's being unresponsive.

b. *Suppose now that the network is not well behaved — it contains routers and it is possible that the network might partition itself for short periods so that some clients can communicate with the server and others cannot. Are clients immune from any ill effects of server crashes combined with network partitions? Explain.*

In this case, it is possible that a client might lock a file, but due to a network partition and server crashes, while the client is waiting for the server to respond, another client is able to lock the file, modify the file, unlock the file, without the original client being aware. This could happen in the following scenario:

1. Client A locks byte 0 of file X.
2. There is a network partition separating client A from the server.
3. The server crashes.
4. The server reboots.
5. Client B locks byte 0 of file X.
6. Client B modifies byte 0 of file X.
7. Client B unlocks the file.
8. The server crashes.
9. The network partition is repaired.
10. The server reboots.

11. Client A reclaims its lock.

c. *In DFS, servers give clients tokens that allow clients to use cached data from files. These tokens have time limits associated with them, so that if the token times out, the client has to assume that any cached data it has from the server is invalid and must obtain a new token (and new data) from the server. Explain how tokens might be used to help with the situation of part b. (Hint: it won't necessarily solve the problem, but it might help client applications become aware of problems.)*

   If the time-out period of a token is sufficiently short, then client A of the example given in the solution to part b would discover that its tokens had timed out by the time Client B could have locked the file. Thus the client-A application could be delivered an error (from the return of a file-related system call), indicating that its lock is no longer good and has been effectively revoked.

d. *Explain why DFS does not support hard-mounted file systems.*

   If a server is down long enough, clients' tokens will time-out. The time-out requirement cannot be relaxed, because doing so would make it impossible for clients to detect potential problems due to network partitions.