CS 167 Final Exam

Three Hours; Closed Book
May 12, 2017

Please write in black ink

Your Name: _____

Your CS Login ID: _____

Do all questions.

1. You are to write a *thread_switch* procedure, i.e., a procedure that causes the processor to switch from one thread to another. Its general form is the following:

```
void thread_switch() {

  NextThread = dequeue(RunQueue);

  swapcontext(…)  // switch from calling thread to NextThread

}
```

Assume that the computer has at least the following special registers: sp (stack pointer), fp (frame pointer), and ip (instruction pointer). Though the computer might employ multiple processors sharing memory, we are concerned strictly with switching one processor from running one thread to running another thread.

a. [8%] Fill in the details of *thread*_switch (in particular, provide some additional C code). Be sure to indicate what the scope of *NextThread* is (e.g., global or local) and describe the contents of what it refers to. If other variables are needed, be sure to explain their purposes and make it clear what their scopes are. What are the arguments to *swapcontext*? What does *swapcontext* do? (Be specific: if it affects certain registers, which registers does it affect and how?) You may assume that dequeue removes and returns the first element of the queue given as its argument and that all necessary synchronization is dealt with — i.e., you need say nothing further about dequeue. You may also assume that there is always another thread to run.

b. [7%] Suppose *thread_switch* takes an argument and this argument is referred to after *swapcontext* returns. Explain what, if anything, must be done to ensure that what is referred to is what was passed in the call to *thread_switch*.

2. Weenix and other operating systems have instances of the following pattern in their kernels, which is executed by kernel threads:

```
while (!sufficient_quantity(resource)) {
        wakeup(resource_daemon);
        wait(daemon_finished);
}
```

The *resource_daemon* is a separate thread that increases the quantity of the resource, then wakes up all threads waiting on *daemon_finished*. If *wakeup* is called but no thread is waiting, nothing happens. Threads who subsequently call *wait* must wait for another *wakeup*. Assume that all threads are of equal priority (your answers should not change this assumption!)

a. [6%] Weenix has a non-preemptible kernel. Explain why this code wouldn't work on an OS with a preemptible kernel.

b. [3%] Suppose we have an implementation of POSIX threads in the kernel. Explain why the following code does not fix the problem of part a.

```
pthread_mutex_lock(&resource_mutex);
while (!sufficient_quantity(resource)) {
        wakeup(resource_daemon);
        pthread_cond_wait(&daemon_finished, &resource_mutex);
}
pthread_mutex_unlock(&resource_mutex);
```

Assume that *resource_daemon* calls *pthread_cond_broadcast(&daemon_finished)* once it has increased the quantity of the resource.

c. [6 %] Fix the above code so that it works. (Hint: are *wait* and *wakeup* the appropriate functions to be using to manage the execution of *resource_daemon*? You might sketch the *resource_daemon's* code.)

3. [15%] Suppose we have a uniprocessor computer that supports multiple security compartments. The intent is that there cannot be any communication between compartments: each compartment provides an isolated environment. We showed in class how the processor load can be used to implement a covert communication channel. In particular, a one-bit message might be sent by using a heavy processor load to indicate a one and a light processor load to indicate a zero. Assuming there are N compartments, describe the design of a scheduler that would eliminate this covert channel.

4. Recall the difference between lazy allocation of memory and eager allocation of memory. In the former, virtual memory is allocated, but the necessary backing store is allocated only when needed. In the latter, the backing store is allocated at the same time the virtual memory is allocated.

   We discussed in class the use of shadow objects as part of the representation of address spaces. For example, a shadow object is created when a file is privately mapped into a process. This shadow object becomes the top-most memory object for the file's mapping into the process. Pages from the file that are modified by the process are associated with this top-most shadow object. If these pages must be paged out, they are paged out to backing store associated with the shadow object. When a process creates a new process via fork, new shadow objects are created for each privately mapped file, in both the parent and the child. Each of them becomes the top-most shadow object for the file's mapping into the process, and is linked to what was the top-most shadow object of the parent. We say that shadow object A dominates shadow object B if A is either linked to B, or A is linked to a shadow object that dominates B. A process that is linked to a shadow object that dominates shadow object C is said to be indirectly linked to C.

   a. [5%] Suppose a 1024-page file is privately mapped into a process. If lazy allocation is used, how much virtual memory and how much backing store are each allocated at the time the file is mapped? How much if eager allocation is used?

b. [5%] Suppose the process of part a has modified 10 pages of the file, and then forks, creating a child. Thus two shadow objects are created, one for the parent and one for the child, and each is linked to the original shadow object that is connected to the file object. What is the maximum amount of backing store the two processes will need collectively for the regions of their addresses space into which the file is mapped?

c. [5%] After the fork of part b, and assuming lazy allocation, will any more backing store need to be allocated for the original shadow object created when the file was first mapped?

d. [5%] When can all the backing store associated with a shadow object be freed?

5. We have a guest operating system that supports virtual memory, running in a virtual machine on a VMM that, of course, also supports virtual memory. (You may assume for this problem that we have only a single virtual machine, though there may be other processes supported by the VMM that are competing for memory.) Assume the real processor is an Intel x86-64 without extended page table (EPT) support. Thus it has no special support for implementing page translation on a virtual machine. Thus the guest OS constructs page tables mapping virtual-virtual memory (of its applications) to virtual-real memory. The VMM might have page tables mapping virtual-real memory to real memory. However, if the system is running an application of the guest OS (in virtual-virtual memory), the page table used by the hardware must map virtual-virtual memory to real memory.

    a. [5%] Explain how the VMM constructs this page table (the one mapping virtual-virtual memory to real memory). You do not need to go into the details of how the multi-level page table is set up, but describe how it is determined, for each page of virtual-virtual memory, what its translation into page frames of real memory is, if the translation exists. Otherwise indicate which translations are marked invalid.

b. [15%] A possible concern with such double implementation of virtual memory (on both the guest OS and the VMM) is that both are managing system resources, possibly in conflict with each other. One approach to avoiding such conflict is, essentially, to put the guest OS in charge of managing memory resources. The VMM puts aside a fixed number of page frames, say N, for use by the guest OS. The mapping from virtual-real to real memory would map the first N pages of virtual-real memory to these N page frames and would not change for the life of the virtual machine running the guest OS. The guest OS would then be responsible for mapping virtual-virtual memory to this fixed amount of virtual-real memory. N, of course, would be less than the total amount of page frames on the real machine. The guest OS would thus be solely responsible for determining which virtual-virtual pages are currently mapped to real memory.

   Alternatively, management of memory resources could be handled by the VMM. The size of virtual-real memory could be made large enough so that there are enough virtual-real pages so that all active virtual-virtual pages could be mapped one-one to virtual-real pages. However, the VMM would now have to support mapping this large virtual-real address space onto a much smaller real address space. Thus it's the VMM that determines which virtual-virtual pages are currently mapped to real memory.

   Let's assume the real computer has 2^{32} bytes of real memory. For the first approach, assume it assigns 2^{30} bytes to the virtual machine running the guest OS. For the second approach, assume the virtual-real size is 2^{48} bytes. Assume the page size is 2^{12} bytes.

   What are the advantages of the first approach? What are the advantages of the second approach? Explain. (Hint: consider system calls, such as *madvise*, that give advise to the OS about future referencing behavior. Also consider the data structures required to represent real memory (and virtual real memory)).

6. Consider the following sequence of events in NFSv4:

   i. Client A requests an exclusive lock on all of file X and the client receives the server's response of "success."

   ii. Client A requests that the lock be "downgraded" to a shared lock. The server responds "success," but the response is lost.

   iii. Client B requests a shared lock on all of the same file. The client receives the server's response of "success."

   iv. The server crashes.

   v. The server restarts and starts its grace period.

   vi. Client B quickly recovers its shared lock.

      a. [7%] The server is still in its grace period as client A attempts to reestablish its lock state. What does it do and what are the responses? (Hint: we may have a problem here.)

b. [8%] What might be done to fix this problem with the protocol?