CS 167 Midterm Exam Solutions

Spring 2017

Do all questions.

1. *A certain operating system has a synchronization construct known as events. Assume there are two operations on variables declared as events: event_wait and event_pulse. When a thread calls event_wait, it blocks until some other thread calls event_pulse. When a thread calls event_pulse, all threads currently waiting on the event are woken up; threads subsequently calling event_wait will block until the next call to event_pulse. We'd like to use events and mutexes to implement POSIX condition variables. Suppose pthread_cond_wait and pthread_cond_broadcast are implemented as follows:*

```
void pthread_cond_wait(pthread_cond_t *cv, pthread_mutex_t *m) {
   pthread_mutex_unlock(m);
   event_wait(cv->event);
   pthread_mutex_lock(m);
}

void pthread_cond_broadcast(pthread_cont_t *cv) {
   event_pulse(cv->event);
}
```

Does this implementation of pthread_cond_wait and pthread_cond_broadcast work correctly? Explain. (Don't worry about pthread_cond_signal or the effects of cancellation.) Hint: consider the following two-threaded program, where x and y are initially 1 and the mutex is unlocked. Under a correct POSIX threads implementation, both threads will always run to completion if both are started in their respective procedures.

```
Thread1() {
   pthread_mutex_lock(m);
   while (x)
      pthread_cond_wait(c, m);
   y = 0;
   pthread_cond_broadcast(c);
   pthread_mutex_unlock();
}
```

```
Thread2() {
   pthread_mutex_lock(m);
   x = 0;
   pthread_cond_broadcast(c);
   while (y)
      pthread_cond_wait(c, m);
   pthread_mutex_unlock(m);
}
```

No. The problem has to do with its being possible for a pulse operation to take place between a thread's unlocking the mutex and calling *event_wait* in *pthread_cond_wait*. In the sample program, suppose thread 1 runs first and gets as far as unlocking the mutex within *pthread_cond_wait*. Thread 2 starts running. Since the mutex is now unlocked, it calls *pthread_cond_broadcast*, causing *event_pulse* to be called, and then gets as far as the call to *event_wait* within *pthread_cond_wait*. Now thread 1 continues execution. It calls *event_wait* and blocks. Thus it will not return from *pthread_cond_wait* and then set y to 0 until some thread calls *event_pulse*. Thread 2 has already called *event_pulse*, yet will not return from *pthread_cond_wait* (or find y to be zero) until thread 1 continues. Thus there is a deadlock, which does not happen in the correct POSIX threads implementation.

2. *In this question we explore some of the details of how signals are implemented in Unix. We saw in class what is done to force a thread to call its signal handler after it notices it has been signaled. What we're discussing here is how the thread notices. Assume that when a thread is signaled, a bit is set in its thread control block (in the kernel) indicating it's been signaled. You may also assume the existence of a routine InvokeSignal, called by a thread running in kernel mode, that determines what the thread's response to a signal should be and sets up the thread's user stack, if necessary, to make that response happen. Once InvokeSignal returns, the thread then immediately returns from kernel mode to user mode — its user stack has been set up so the right thing happens. The questions here are: when is this routine called, and how does the thread know to call it? [Don't answer them yet; we'll guide you through them.] Though it's important, in this problem you need not describe how the return values from system calls are set up.*

   a. *Since it's important that a signaled thread call InvokeSignal itself, and that it call it while in kernel mode, a signaled thread should check whether it's been signaled at some point when it is running in kernel mode. Assuming the thread is either currently running or is runnable (i.e., on the run queue), describe at what points in the thread's execution it should check if it's been signaled and then call InvokeSignal. You may assume there are periodic clock interrupts. (Hint: there may be a number of places where this could happen, but we want to make things as simple as possible, but still viable.)*

      The simplest approach is to check for pending signals when the thread is about to return to user mode, after completing either a system call or interrupt processing.

   b. *Suppose now the signaled thread is sleeping, i.e., waiting on some wait queue in the kernel, and we'd like to wake it up and force it to deal with the signal. In particular, assume the thread has called wait. Normally, when the thread returns from wait, it may assume what it's been waiting for has happened (this is not pthread_cond_wait!). However, whoever signals the thread, upon discovering the thread is sleeping, will call unwait to force the thread to wake up and return from the wait call. Thus wait returns some special value indicating that it's returning because of an unwait, rather than returning because what it's waiting for has happened. The thread should, at this point, check to see if it's been signaled. However, this is not a good place to call InvokeSignal, since we need to make sure the thread immediately returns to user mode. Thus we need to get the thread to a point where it can immediately return to user mode (in particular, its kernel stack must be unwound). Explain how this might be done. (Hint: consider the use of setjmp and longjmp. Recall that setjmp saves the stack state and instruction pointer of a thread in a jmpbuf and returns a 0; longjmp restores that state and causes the thread to return from the original call to setjmp, returning the value of the second argument to longjmp).*

      To unwind the stack, the thread should call *longjmp*. To establish a point to which the stack should be unwound, it should call *setjmp*. So, when the thread enters a system call, it calls *setjmp*, saving its state in a jmpbuf in its thread control block:

```
if (setjmp(tcb->jmpbuf) == 1) {
   return;
}
/* start processing system call */
```

      On return from wait, if it was "unwaited", it longjmps:

```
if (wait(...) == UNWAITED) {
   longjmp(tcb->jmpbuf, 1);
```

```
}
```

3. *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.

4. *Explain how renaming a file can be done using the consistency-preserving approach so that a crash will not create situation in which the file is lost. Be sure to take into account the link count stored in the file's inode. Note that there may have to be a temporary inconsistency; if so, explain why it will not result in lost data.*

   As explained in the text, to rename a file you should first create a link to the file under its new name, then remove the link corresponding to its old name. Creating the new link cannot be done atomically, since both the directory must be updated to contain the link and the file's inode must be updated to increase its link count from 1 to 2. If the directory entry is created before the link count is updated, there could be a problem if the system crashes between the two operations. After the crash, though there are two links to the file, its link count is one. If one of the links is removed, the link count is reduced to zero and the file might be deleted since it appears to be no longer referenced. On the other hand, if the link count is updated before the directory entry is created, and a crash occurs between the two operations, there will be just one entry referring to the file, even though it has a link count of two. The worst that could happen in this case is that someone intends to delete the file by removing that link, but because it would still have a positive link count, the file continues to exist, but is not referenced by any directory. This, as explained in the text, is an innocuous inconsistency.

   After the new link is created, we have a similar problem in removing the old link. Should the link be removed first, then the file's link count decremented by one, or vice versa? Following reasoning similar to the above, we should first remove the old link, then decrement the file's link count, so that the link count is always at least as great as the actual number of links.

If you do all of the following correctly, you'll get an A regardless of how well you do on the first four problems. If you miss any of the following, your grade will be based solely on how well you do on the first four problems.

Each of the following abbreviations would have been familiar to you if you were a tech-savvy person at the appropriate time and place. What does each of them stand for?

5. *EBCDIC*

   Extended Binary Coded Decimal Interchange Code (1963)

6. *PCMCIA*

   Personal Computer Memory Card International Association (1990)

7. *TWAIN*

   Technology Without An Interesting Name (what it stands for was produced after the name was in use, it's originally from Kipling's "The Ballad of East and West": "... and never the twain shall meet ...") (1991)

8. *SCSI*

   Small Computer System Interface (1978)

9. *HIPPI*

   High Performance Parallel Interface (1987)

10. *BRUNET*

    Brown University Network (1980)