

EXAM 1 Solutions

CS 4210 Advanced Operating Systems

Fall 1999 • Georgia Tech/Computer Science • Hutto

This exam is closed-book. There are 12 questions worth 80 points. You have 1 hour and 20 minutes. The number of points is a rough estimate of the amount of time you should spend on a question. Don't spend too much time on one question! All questions have relatively short answers. Partial credit is possible. Ask if you have any questions. Good luck!

Birrell paper

1. [5 points] Why must the condition variable `wait(cv, mutex)` primitive be implemented atomically? Why can't we just say:

```
lock( mutex );
...
if ( !condition ) {
    unlock( mutex );
    wait( cv );
    lock( mutex );
}
...
unlock( mutex );
```

It creates a synchronization “window” between the unlock and wait operations. Some other process could lock, change the condition and signal, all in the interval between the unlock and wait. Condition variable signal is different from semaphore signal in an important way. If no process is actually waiting when the signal is sent, it is discarded (lost). When the process finally does wait, it may wait forever (deadlock) because it has “missed” its wakeup (signal). This is the “lost wakeup” problem.

Some people said the if should be a while. This is correct but addressed the “spurious wakeup” problem. The lost wakeup (more serious) is still possible as long as the three operations in the inner block are not atomic.

2. [5 points] What problem does the following code transformation avoid on multiprocessors?

Straight out of the Birrell paper. This is only an issue on a true multiprocessor. If the process executing the above code (P1) is delayed for any time between the signal and unlock in the first case, the signaled process (P2) may actually start running on another processor while P1 is still executing and holding the mutex. P2 will try to acquire the mutex and fail causing it to block again. This involves several unnecessary scheduling operations. By moving the signal outside the unlock, we can increase the likelihood that the signaled process (P2) will be able to successfully acquire the mutex when it wakes up, improving efficiency.

Lewis and Berg chapters

3. [5 points] Lewis and Berg list 11 advantages of using threads! Describe 5 advantages.

1] The threads abstraction “exposes” parallelism to the programmer in the same way that the process abstraction does. It allows programmers to actually write programs that can be executed in parallel if multiple processors are available.

2] Threads serve as a code structuring mechanism, allowing otherwise “asynchronous” activities to be expressed as coherent individual activities (threads). This simplifies and clarifies program structure, enhancing reliability and maintainability.

3] Threads can obviously provide “speedup” (increased efficiency) when run on a multiprocessor if threads can be executed in parallel. (This isn’t always the case. Some applications are inherently serial. Also if you program your application badly, overhead may actually INCREASE the execution time.)

4] Threads can even provide speedup on a uniprocessor if threads are executing slow blocking io operations. While one thread blocks, another thread can execute if the current process has available time remaining in its time slice.

5] Threads are an improvement over processes. Traditionally, to get the above benefits you must decompose your program into multiple processes. Processes are big and heavy and require lots of intervention and support from the kernel. Threads are light-weight and allow the above benefits without much overhead.

Some others:

- kernel threads can be used to structure the kernel; in Solaris they are even used to handle interrupts
- threads increase “responsiveness” in programs with user interfaces (like a Web browser)
- good for “multi-threaded” servers in distributed environments

4. [10 points] Lewis and Berg have an interesting way of describing the relationship between counting semaphores and condition variables. In class I said something like “Condition variables can be viewed as generalized semaphores that have been ‘ripped open’”. Explain.

See Figures 6-6 and 6-7 on pages 94-95 and the associated discussion. Basically, manipulation of a semaphore requires mutual exclusion so conceptually there is a mutex associated with the semaphore implementation. The semaphore wait “condition” is $\text{value} > 0$. Threads executing a `sem_wait` continue if the semaphore value is greater than 0. They block otherwise. Condition variables generalize this mechanism, allowing “user-supplied” conditions. I say they are “ripped open” because condition variables expose the mutex which is “implicit” in a semaphore. The programmer must actually define and associate a mutex with the condition variable and explicitly acquire and release the mutex before testing and changing the condition.

Solaris papers

5. [5 points] What is a thread stack “red zone” as described in the Solaris implementation papers?

This is a standard “trick” to avoid invalid memory references. Some fancy “debugging malloc’s” will actually do this as well and map invalid pages around a malloc’ed region. Attempts to touch the page before or after will result in a page fault that will be interpreted as a “segmentation violation”. Solaris maps an invalid page after the end of each thread stack to detect stack overflow. Remember all the thread stacks live in the user-accessible process address space. An attempt to write beyond the end of one stack (a stack overflow) is likely to corrupt the thread stack higher (or lower, depending on how stacks grow) in memory.

Notice that this trick is helpful but doesn’t prevent corruption. If a thread overflows the stack by more than a page (8K under Solaris) you essentially “hop over” the red zone and corrupt the next stack. Dooh!

6. [10 points] How does the Solaris user-level threads library automatically “manage” the number of light-weight processes (LWPs) allocated to a process? Be sure to mention SIGWAITING.

There are two mechanisms: one allocates LWPs, the other deallocates them. Most people vaguely remembered the idea but didn't get the details right. The Solaris user-level threads library attempts to avoid deadlock because of a lack of (virtual) processor resources. In other words, the goal is: never allow a process to deadlock because there are not enough LWPs. How can this happen? If a thread executing on an LWP blocks, that LWP is “unavailable” for scheduling another thread until the thread is unblocked. If a process has 5 threads and 5 LWPs and they all block, there are no LWPs (execution “contexts”) remaining. If all 5 blocked threads are waiting on a 6th thread in the same process to release a resource that is causing the blocking, the process is now deadlocked!

Solaris “detects” this situation when the user-level thread library blocks the “last” thread. Before blocking, it sends SIGWAITING to itself. On receipt of SIGWAITING, the user-level thread library will allocate another LWP, potentially allowing the critical thread to run. Thus, a process is serially allocated additional LWPs up to the number of threads in the application. (This seems to “unrestrained” to me but its what the paper says. It would be interesting to write a program that tried to “acquire” as many LWPs as it can and see what happens when the number gets big...)

To deallocate LWPs, the library keeps a timer for “unused” LWPs. If an LWP is unused for some amount of time, it is deallocated.

Once again, all this is transparent to the programmer, though calls exist to allow explicit control over LWP allocation and deallocation.

Psyche paper

7. [5 points] Traditional user-level threads packages intercept blocking system calls and replace them with asynchronous or non-blocking calls. Why? What clever technique does the Psyche system use to avoid this transformation?

This is the standard user-level thread package “trick”. If the kernel doesn’t know anything about threads or doesn’t have a “virtual processor” abstraction like LWPs, a blocked thread blocks the entire process. If the kernel provide asynchronous (non-blocking) system calls, the user-level thread library can provide a “wrapper” around each apparently synchronous system call and replace it with an asynchronous call and then schedules another thread. When the call completes, the process will receive a signal, the thread-library will notice the completion of the operation and make the “blocked” thread runnable again.

Psyche builds into the kernel a generic signaling mechanism that notifies a process whenever a system call blocks. Individual processes can turn this notification off if they are not interested. This is a nice unification of blocking and non-blocking system calls and does not require the OS to provide separate blocking and non-blocking versions of every system call.

8. [5 points] What do the Psyche designers mean when they say that user-level threads in their system are “first class”?

They suggest several “improvements” that all can be viewed as improving or elevating the “status” of user-level threads. In a many-to-one architecture the kernel doesn’t know anything about threads (“What’s a thread?”). It simply manipulates (schedules, creates, etc.) processes. A process is the “unit of scheduling” that the kernel knows about. Threads are therefore “second class”. They are not as “good as” processes.

The terminology comes from the programming language world. In many programming languages, functions and data types are special entities and what you can do with them is very limited. For example, you can’t assign a data type or a function to a variable and pass it around. In languages that allow this capability (like LISP or JAVA) we say functions and data types are “first class”. No longer “second class citizens”. They have all the rights and capabilities of other entities in the system.

Bloom thesis

9. [10 points] Serializers provide the following primitives:

```
wait( queue ) until ( condition )
join( crowd ) { .../* inside crowd */ }
empty?( queue );
empty?( crowd );
```

Using these primitives **construct a solution to the FCFS readers/writers problem**. (If there is a writer waiting, readers that arrive subsequently may not read until the writer is finished, even though there are currently readers reading.) (**Hint:** This isn't really that hard. You just need to queue arriving readers and writers and let them "in" at the appropriate times.)

The trick here is to have a **SINGLE** waiting queue. If you put readers and writers in separate queues you can't determine whether the head of the writer queue or the head of the reader queue should "go next". The single queue encodes and serializes the "arrival time" of the readers and writers. The condition for admitting readers and writers is slightly different. A writer can go if there are no readers or writers. A reader can go if there is no writer. We need separate readers and writer crowds so we can distinguish who is currently in the critical section.

```
queue waiters;
crowd readers, writer;

read() {
    wait( waiters ) until ( empty?(writer) AND empty?(readers) );
    join( readers ) {
        // do your business
    }
}

write() {
    wait( waiters ) until ( empty?(writer) );
    join ( writer ) {
        // do your business
    }
}
```

10. [5 points] Give a **path expression** that specifies a solution to the readers/writers problem without the FCFS property. In other words, arriving readers are allowed to read even if a writer is waiting. (**Hint:** This is easy.)

+ means OR.

{ } means more than one.

path { read } + write end

Order doesn't matter so you could also say:

path write + { read } end

Anderson paper

11. [10 points] Why does the “test-and-test-and-set” (read from cache and then try test-and-set) spinlock solution perform poorly under high contention?

Say you have a bunch of processes trying to get the lock. One process holds it, all others are currently reading the “test” variable from their cache so there is little contention to begin with. Now the process “releases” the lock by setting the test variable to 0. This invalidates everyone’s cache causing a round of updates over the bus. A few processors will probably be “first” and get the new value of 0. These guys will then try to acquire the lock by doing a test-and-set. Only one will succeed. Each test-and-set will in turn cause another round of invalidations, even if it FAILS! In the horribly worse case, all N waiting processes will see the 0 value each time the lock is released and all N processes will try to do a test-and-set. This results in N^2 invalidations to allow N processes to acquire and release the critical section. Note that all the bus traffic will even slow down the process that HOLDS the lock. Yucko!

12. [5 points] Complete the following code for Anderson's queuing solution from the paper. (P is the maximum number of processes).

```
init:
    flags[0] = HAS_LOCK;
    flags[1..P-1] = MUST_WAIT;
    myPlace = 0;

lock:
    myPlace = atomic_read_increment( myPlace );
    while ( (myPlace mod P) == MUST_WAIT )
        ;
    flags[ myPlace mod P ] = MUST_WAIT;

unlock:
    flags[ (myPlace + 1) mod P ] = HAS_LOCK;
```

It is critically important that read_increment operation be performed atomically. It's something like a test_and_set primitive.