Computer Science 162

Tom Anderson
Fall 1995, Midterm 1

Problem 1: (6 points)

For each of the following statements, indicate in one sentence whether the statement is true or false, and why.

1. You never need to balance your checkbook, since you can trust the Banker's algorithm to always compute your account balance correctly.

   FALSE. The Banker's algorithm is used to avoid deadlock and not to see if a bank account is balanced or not.

2. Using segmentation allows you to address a larger amount of virtual memory than using paging.

   FALSE. The amount of addressable memory is determined by the number of bits of virtual address, not the translation scheme.

3. Virtual memory address translation is useful even if the total size of virtual memory (summed over all programs) is always smaller than physical memory.

   TRUE. Address translation provides protection.

Problem 2: (8 points)

Provide a short answer for each of the following questions:

1. In Nachos, executing in an interrupt hander is different from executing in a thread. While a thread can block, suspending its exection, yielding to another thread, an interrupt handler is not allowed to block. Why?

   An interrupt handler can't block because the interrupt handler runs in the context of the current thread, so: (i) the interrupted thread won't be able to continue until the interrupt returns, (ii) other interrupts aren't allowed until the interrupt returns, and (iii) an interrupt has doesn't have a thread control block or its own stack, and so it can't be pulled on/off the ready list.

2. Can round robin ever be the optimal CPU scheduling strategy, in terms of minimizing average response

time? If so, when (list all cases)? If not, explain why not?

Shortest job first is optimal. If round robin ever takes a time slice, it can't be optimal, because SJF never time slices between jobs.

However, if there's only one job, or if all jobs are shorter than the time slice, and they are all the same length, then RR = FIFO = SJF.

Problem 3: (8 points)

Four approaches to coping with deadlock are:

___ Prevention by means of resource ordering.
___ Prevention by means of claiming all resources in advance of execution.
___ Prevention by means of eliminating waiting, by starting over from scratch if a resource is busy.
___ Detection and correction, based upon graph reduction.

Asuming there is little contention for shared resources, label the alternatives in this list from 1 to 4, where 1 means "greated potential for executio concurrency" and 4 means "least potential for execution concurrency." That is, in general, 1 allows the most threads to do the most work independent of each other.

3 resource ordering
4 claiming in advance
2 eliminating waiting
1 detect and correction

An explanation:

Deadlock detection lets the program go ahead normally, as if deadlock isn't a consideration, until you get into a deadlock. Maximal concurrency.

Eliminating waiting also lets the program go ahead normally, as if deadlock isn't a consideration, until the program tries to grab a resource that's in use. Since resources are typically unused (contention for resources is low), then almost as good as detecting and breaking deadlock.

Resource ordering is a greater restriction on the behavior of the program -- the program can't grab resources out of order, even if they are unused (as an analogy, this is like making all the cars drive around the Bay clockwise to avoid deadlock).

Claiming all resources in advance is even stricter than resource ordering.

Problem 4: (14 points)

1. In C, define two data structures, one representing the contents of a typical segment table entry, and the other representing the contents of a typical page table entry.

```
struct segment_entry {
    PageTable *pageTable;              // also OK was base & bounds
    int pageTableSize;
}

struct page_entry {
    int physicalPageFrame;             // note: no size field!
                                       // all pages are same length!
}
```

2. Using the above data structures and the C variables listed below, write a C expression for converting a virtual address into a physical address, when the virtual address space is segmented, and each segment is paged. Assume that all segment tables and page tables are in physical memory, there is no translation buffer, and no fault occurs.

   Feel free to define any other quantities you believe might be helpful.

vAddr	the virtual address
segTable	the segment table
segSize	the maximum size of a segment
pageSize	the page size
quot(a, b)	the quotient of a/b
rem(a, b)	the remainder of a/b (a mod b)

   (Assuming segSize and pageSize are in bytes. If in bits, it's a little more complicated, but still do-able.)

```
segTable[quot(vaddr, segSize)].pageTable[
        quot(rem(vaddr,segSize), pageSize)].physicalpageFrame*pageSize
                + rem(vaddr, pageSize)
```

Problem 5 : (24 points)

A local company, Real Time in Real Time, Inc., (their motto is, "not fast software, good software served quickly") ants to use Nachos as the basis for their next product, but they are worried about whether Mesa- style monitors could be used for real-time applications.

1. Briefly explain the problem: with Mesa-style monitors, why a waiting thread might have to go back to sleep after being signalled. Since this could be repeated an arbitrary number of times, there is no guarantee that a waiting thread won't starve!

   With Mesa style monitors, a signalled thread is put on the ready queue with no special priority. As a result, another thread can slip in and grab the monitor lock before the signalled thread. When the signalled thread eventually gets the CPU, the monitor lock could be busy, but even if it isn't, the condition may have changed, so that the signalled thread may have to go back to sleep (in other words, the "if" vs. "while" issue in Mesa-style monitors).

2. To convince them this isn't a real problem, implement strictly FIFO semaphores (each V causes the thread that has been waiting the longest time in P to grab the semaphore) using Mesa-style locks and condition variables for synchronization. Your solution should not have any busy-waiting. You can assume (without providing an implementation) any of the non-synchronized utility routines provided by Nachos.

   *The approach is just to make sure no other thread can grab the semaphore before the signalled thread. So in V, if someone is waiting, signal them, but *don't* increment the semaphore. That way, the signalled thread will go, and no one else can slip in. This assumes condition->Wait() keeps threads in FIFO order (which is true for almost all of your implementations for Nachos assignment 1); you can also build an explicit list of threads, each with their own condition variable, if you don't like that assumption.*

```
class Semaphore {
    Lock lock;
    Condition condition;
    int value;
    int numWaiters;
}

Semaphore::P() {
   lock->Acquire();

   if(count == 0) {
      numWaiters++;
      condition->Wait(lock);
      }
   else
      count--;
   lock->Release();
}

Semaphore::V() {
   lock->Acquire();

   if (numWaiters > 0) {
        numWaiters--;
        condition->Signal(lock);
   } else
```

```
        count++;

    lock->Release();
}
```

Return to CS162 Homework and Exam page

Last modified 4/14/95