University of Pennsylvania
CSE380 – Operating Systems
1st Midterm Exam – 10/12/2004
©2004 Matt Blaze

Instructions: Write your answers in an exam book. Remember to print your name clearly on the exam book cover, to erase or cross out any material you don't want graded, and to indicate with which question number each answer is associated. For answers that include programs or program segments, you may use C or any reasonable pseudocode whose syntax and semantics are clear.

Long answers are not required. The short answer / essay questions are designed to be answered in about one paragraph each, and some can be answered fully in one or two economical sentences.

Don't cheat. This is a closed-book exam.

1. **(20 points total):** In an attempt to get tenure in the Department of Philosophy and Gastronomy at the University of Southern North Dakota at Hoople[1], Professor Feckless C. Coder, PhD is building his own operating system. So far, the only difference between Prof. Coder's system and Unix is that his system does not have a `wait()` system call. Instead, Prof. Coder implements a non-blocking `child_is_alive()` system call that returns `TRUE` if a child is still alive and `FALSE` otherwise. According to Prof. Coder, this is more flexible, since a function that behaves just like the regular `wait()` can be built using `child_is_alive()`.

   1a. (10 points): Implement `wait()` using `child_is_alive()`. You may assume that neither function takes arguments and that wait should return immediately once `child_is_alive()` returns `FALSE`. You need not worry about error conditions.

   Sample Answer: We can implement a functional equivalent to `wait()` with a simple loop that polls `child_is_alive()`, as shown here in C:

```
wait()
{
   while (child_is_alive())
       ;
   return 0;
}
```

   1b. (10 points): Compare the expected performance of your implementation of `wait()` with that of the standard Unix version. Are there any disadvantages to Prof. Coder's design?

   Sample Answer: In conventional Unix systems, `wait()` blocks; processes blocked in `wait()` are not scheduled and do not consume CPU resources. In Coder's system, `wait()` busy-waits, consuming CPU resources that would otherwise be available to other processes (including the child being waited for). Coder's system is much less efficient.

[1] **Extra Credit A (2 points):** Name another (in)famous member of the faculty of the UofSNDatH. (Hint: Think of the music department).

2. (10 points): Bolstered by his success replacing `wait()` and in an attempt to become known as the next Ken Thompson[2], Prof. Feckless C. Coder, PhD has decided to optimize his operating system especially for scientific computation. The main difference between this system and Unix is that instead of providing standard user-level library functions for the various trigonometric functions (e.g., *sin*, *cos*, *tan*, etc.) as Unix does, Prof. Coder's system implements these functions as system calls.

Is this a good idea? Comment on the likely performance impact of implementing these functions as system calls, assuming a workload consisting mostly of programs that make frequent use of these functions.

Sample Answer: This is a bad idea; there is nothing about trigonometric functions that would benefit from the services of the operating system and therefore no advantage to implementing them as system calls. There is considerable disadvantage to doing so, however, especially for workloads consisting of programs that use these functions frequently. Each call to system call versions of these functions would undergo the added overhead of invoking a trap and switching from user-mode to supervisor-mode (and back).

3. (15 points total): In an attempt to become known as the next Linus Torvalds[3], Prof. Feckless C. Coder, PhD intends to modify the the standard Unix scheduler. Instead of the complicated multi-level priority queue scheme implemented by most versions of Unix, Prof. Coder proposes using non-preemptive Shortest Job First (SJF) scheduling.

3a. (5 points): Assuming a correct and successful implementation, what impact on *average turnaround time* would you expect Prof. Coder's modified scheduler to have, assuming typical interactive computing workloads?

Sample Answer: Average turnaround time in SJF is *optimal;* we'd therefore expect this scheduler to have at least as fast turnaround as Unix.

3b. (5 points): Assuming a correct and successful implementation, what impact on the worst-case *maximum waiting time* would you expect Prof. Coder's modified scheduler to have?

Sample Answer: The worst-case waiting time in SJF is *infinite*; while theoretically, preemptive Unix-style priority-queue-based schedulers can also have starvation, the movement of jobs into different priority queues over time makes this very unlikely.

3c. (5 points): Assuming he is able to get enough expert programming help (such as by hiring someone who finishes this course), do you expect Prof. Coder to be successful in implementing his scheduler? Why or why not?

Sample Answer: It is theoretically impossible to implement SJF on general programs; doing so would require solving the halting problem, which we know to be intractable. (The best we can do is make predictions – guesses – based on recent program behavior, but that's not SJF anymore). We can be confident that Coder and his team, no matter how skilled, will not succeed at implementing this scheduler.

4. (15 points): In an attempt to become known as the next Bill Gates[4], Prof. Feckless C. Coder, PhD has built yet another scheduler for interactive multitasking operating systems. Intended for modest desktop computers running at about 10 million instructions per second (10 MIPS), the scheduler is based on a single-queue preemptive round-robin scheme with a quantum of 1 microsecond (1/1,000,000 of a second).

Discuss the performance of this scheduler, especially with respect to efficiency and response time.

Sample Answer: This scheduler only allows ten instructions to execute before switching to another process. The quantum is much too short. Most of the system's CPU cycles will be spent in the

[2]**Extra Credit B (1 point):** Who is Ken Thompson?
[3]**Extra Credit C (1 point):** Who is Linus Torvalds?
[4]**Extra Credit D (0 points):** Who is Bill Gates?

operating system, scheduling and context switching. This is very inefficient. Although to a point a small quantum can improve response time, one that's so small that it makes the system very inefficient (as is the case here) has the effect of *increasing* response time.

5. **(15 points):** In an attempt to become known as the next John von Neumann[5], Prof. Feckless C. Coder, PhD has decided to try his hand at computer architecture design. His first project attempted to implement the Test and Set Lock (TSL) instruction for mutual exclusion. Unfortunately, Prof. Coder couldn't remember exactly what the Test and Set instruction was supposed to do or how it could be used to implement mutual exclusion. Recalling that atomically exchanging the values in two memory locations can be used for the same purpose, he implemented an atomic "swap" instruction instead. Thoughtfully, he also implemented a C library function, `swap(a,b)`, that uses his instruction to exchange the contents of integer pointers `a` and `b` atomically. (E.g., if `x` and `y` are integers, `swap(&x,&y)` exchanges their contents).

   Using Prof. Coder's atomic `swap` function, show code sequences for critical section mutual exclusion entry and exit. Use a single global variable `mutex` for controlling access to the critical section; you may assume that any other variables you use are private for each individual thread.

   Sample Answer: An atomic swap operation can be used almost exactly as a TSL instruction for mutual exclusion spin locking. We assume a local integer variable `i`. Simply use the atomic swap to exchange a local variable (initialized to 1) with the mutex until its value changes to 0:

   ```
   i = 1;
   while (i == 1)
        swap(&i,&mutex);
   ```

   To exit the critical section, we set `mutex` back to 0:

   ```
   mutex = 0;
   ```

6. **(25 points total):** Prof. Feckless C. Coder, PhD has been hired by a bank to design and implement an account management database system. The system is to keep track of each account's balance and allow transfers between them. For reasons known only to himself, Prof. Coder has decided to implement the system entirely in the memory of a single process, with all the authorized users running simultaneous threads with access to the shared memory account database.

   The account database consists of an array of individual account structures. These structures contain a flag indicating whether the account is valid, its account balance (an integer - this bank deals only in whole dollars), and a binary semaphore that can be used to lock the account during transactions:

   ```
   struct {
       int valid;  /* account validity flag - 0=invalid, 1=valid */
       int balance; /* balance in dollars */
       SEMAPHORE s;  /* account lock */
   } accounts[NACCOUNTS];
   ```

   Account numbers are simply integer indexes into the accounts array. For example, the `transfer` function moves money between two accounts as follows:

[5] **Extra Credit E (1 point):** Who was John von Neumann?

```
int transfer(int amount, int from, int to)
{
    /* check that account is valid */
    if ((from < 0) || (from >= NACCOUNTS) ||
        (to < 0) || (to >= NACCOUNTS))
            return INVALIDACCOUNT;
    if (amount < 0)
            return INVALIDAMOUNT;
    lock(from,to);  /* lock from and to (prevent race condition) */
    if (!accounts[from].valid || !accounts[to].valid) {
            unlock(from,to);
            return INVALIDACCOUNT;
    }
    accounts[from].balance = account[from].balance - amount;
    accounts[to].balance = acount[to].balance + amount;
    unlock(from,to); /* unlock them */
    return OK;
}
```

Fortunately, the system has standard DOWN (P) and UP (V) semaphore system calls. Unfortunately, about 20 minutes before the system was to go online, Prof. Coder left town, having finished everything except the `lock` and `unlock` functions. The bank manager is naturally very unhappy, and was last heard muttering, "When they find out I hired that Coder idiot, I'll be as dead as Sweetie π"[6].

6a. (15 points): Implement `lock` and `unlock` to provide exclusive access to two accounts in a way that prevents deadlocks. You may assume standard simple DOWN and UP semaphore functions, that the semaphores in the `accounts` array have been initialized properly (although no other semaphores are available), that only your functions will be used to operate the semaphores, and that each thread will call your lock function exactly once for each transaction prior to calling unlock. You may also assume that all appropriate validation and bounds checking has been performed before your functions are called (e.g., the `from` and `to` arguments will always represent valid entries in the `accounts` array).

Sample Answer: We observe that accounts have a natural order, provided by their index in the `accounts[]` array, and perform the (potentially blocking) semaphore DOWN operations in order of the account numbers:

```
lock(int to, int from)
{
     if (to < from)
        DOWN(accounts[to].s);
        DOWN(accounts[from].s);
     } else if (to > from) {
        DOWN(accounts[from].s);
        DOWN(accounts[to].s);
     } else {  /* to == from */
        DOWN(accounts[to].s);
     }
 }
```

(The final clause is needed because if `to` is the same as `from`, a deadlock would occur on the second DOWN. However, it would be acceptable for the purposes of this question to assume that that case is taken care of by the pre-validation of the data performed by the calling functions.)

[6]Extra Credit F (2 points): Who was Sweetie π, and why is this a semi-reasonable extra credit question for an OS exam?

Unlocking can be done in any order:

```
unlock(int to, int from)
{
     UP(accounts[to].s);
     UP(accounts[from].s);
}
```

6b. (10 points): Make a convincing argument for why the use of your functions will prevent deadlock (e.g., explain the technique you are using and the conditions that can or cannot occur with this technique).

Sample Answer: We have implemented *hierarchical allocation*, ensuring that individual accounts can only be locked (and will only be waited for) in the same order by all threads. This prevents *circular waiting*, one of the four required preconditions for deadlock. Since we have eliminated one of the preconditions, deadlock cannot occur under the stated assumptions.