HOMEWORK 2
SOLUTIONS
CSE 120 OPERATING SYSTEMS
FALL 2006

NB: Questions are in boldface and the solutions follow them.

1) What is the difference between Multitasking, Multithreading and Multiprogramming?

Multiprogramming is the technique of running several programs at a time using timesharing. The OS keeps several jobs in memory simultaneously and selects a job from the job pool and starts executing the job. When that job needs to wait for any I/O operations, the CPU is switched to another job. The goal is to ensure the CPU is never idle when work can be done. Multitasking is the logical extension of multiprogramming. The difference is that the switching between jobs occurs so frequently that users can interact with each program apparently simultaneously. In multithreading, parallelism is extended into the application itself, allowing each application to have multiple simultaneous threads of control within the same address space.

2) What would be a possible problem if you executed the following program? How can you solve it?

```
#include <signal.h>
#include <sys/wait.h>

main()
{
        for(;;) {
                if( !fork() )
            {
                exit(0);
            }
            sleep(1);
    }
}
```

The children are created, and then terminated. They remain as zombies because their exit status must be reported to the main process. The parent process is in an infinite

loop, however, so they pile up without bound. Eventually the OS would run out of memory, or, if poorly implemented, crash. The easiest way to fix this problem would be to have the main process wait on its children to avoid creating zombies.

3) Does exec() return? What happens if the command passed to exec() doesn't exist? Does exec() create a new process?

Exec() does not return upon success; it may, however, fail and return. If the command passed to exec() doesn't exist, it fails. When Exec() succeeds, it loads a new program into the calling process' address space. It does *not* create a new process, but instead overwrites the current program with a new one, which is called the new process image.

4) "The separation of fork() and exec() is probably the most brilliant innovation in the design of Unix." Justify this statement. Why do we need fork() and exec() as separate system calls? Why not a single one that does both?

This was intended to be in jest; you need not agree that it is the 'most brilliant innovation,' but the key benefit of the separation between fork() and exec() is the ability for the parent to easily pass large amounts of context to its child, which is likely to perform related tasks. Because they are separate system calls, processes may use one without the other. For example, the Web server example in class forked a child process to deal with each client separately, but had no need to load a new program so did not call exec()—it would be difficult to do this with a single, unified system call.

5) If one thread in a program calls fork(), does the new process duplicate all threads, or is the new process single-threaded? If a thread invokes exec(), will the program specified in the parameter to exec() replace the entire process – including ALL the threads?

Because fork() copies the entire contents of the address space, user-level threads will all be duplicated. If kernel-level threads are in use, it is possible to imagine an alternative implementation that did not duplicate all the threads, but that would introduce significant complications (e.g., what would the OS do with the thread stacks? Copy them? Zero them out? Recall they are stored in the address space that was copied.) In general, all threads are duplicated. Similarly, because exec() overwrites the entire contents of the address space, a call to exec() will destroy all threads in the address space.

6) [Silberschatz Galvin Gagne – 7th Edition – Chapter 6 – Problem 6.16] How does the signal() operation associated with monitors differ from the corresponding operation defined for semaphores?

Assume x is a condition variable within a monitor. The x.signal() operation resumes exactly one suspended process. If no process is suspended, then the signal() operation has no effect; that is the state of x is the same as if the operation had never been executed. In contrast, the signal() operation associated with semaphores always affects the state of the semaphore.

7) Illustrate with an example how the implementation of a semaphore with a waiting queue may result in a deadlock. Another problem related to deadlocks is indefinite blocking or starvation. What is indefinite blocking or starvation and how might this be caused?

The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. When such a state is reached, these processes are said to be deadlocked.
Consider a system consisting of 2 processes, P0 and P1, each accessing two semaphores, S and Q, set to the value 1:

P0
wait (S);
wait (Q);
.
.
.
signal (S);
signal (Q);

P1
wait (Q);
wait (S);
.
.
.
signal (Q);
signal (S);

Suppose P0 executes wait(S) and then P1 executes wait(Q). When P0 executes wait(Q), it must wait until P1 executes signal(Q). Similarly, when P1 executes wait(S), it must wait until P0 executes signal(S). Since these signal() operations cannot be executed, P0 and P1 are deadlocked.

Indefinite blocking or starvation: A situation in which processes wait indefinitely within the semaphore. Indefinite blocking may occur if we add and remove processes from the list associated with a semaphore in LIFO (Last In, First Out) order.

8) Operating systems often distinguish between two kinds of semaphores – 'binary' and 'counting'. What scenarios do you think each of them is most suitable for?

Ans: Binary semaphores: Deal with critical section problem for multiple processes. Counting semaphores: Used to control access to a given resource consisting of a finite number of instances.

9) Consider the swap() procedure given below:

```
void Swap(boolean *a, boolean *b)
{
        boolean temp=*a;
        *a=*b;
        *b=temp;
}
```

Assuming that this swap() procedure is executed atomically as a single instruction, write a piece of code that demonstrates how this can be used to provide mutual exclusion that satisfies the bounded-waiting requirement. Use the following skeleton code:

```
while(TRUE)
{

        /* your code */

                //critical section

        /* your code */
```

//remainder section

}

(Hint: Use common data structures initialized to FALSE, like:

boolean waiting[n];
boolean lock;

)

Ans:

```
while(TRUE)
{
        waiting[i]=TRUE;
        key=TRUE;
        while (waiting[i] && key)
                Swap(&lock,&var);
        waiting[i]=FALSE;

        //critical section

        j=(i+1)%n;
        while((j!=i) && !waiting[j])
                j=(j+1)%n;

        if (j==i)
                lock=FALSE;
        else
                waiting[j]=FALSE;

}
```

10) [Silberschatz Galvin Gagne – 7th Edition – Chapter 6 – Problem 6.10] Show that, if the wait() and signal() semaphore operations are not executed atomically, then mutual exclusion may be violated.

Ans: Consider the wait() and signal() operations for the semaphore S as given below:

```
wait(S)
{
```

```
        while (S<=0)
                ; //no-op
        S--;
}

signal(S)
{
        S++;
}
```

If these operations are not executed atomically, then increments and decrements to the semaphore S may happen simultaneously thus violating mutual exclusion. So, when one process modifies the semaphore value, no other process should be able to simultaneously modify that same semaphore value. In addition, in the case of the wait() operation, the testing of the integer value of S (S<=0), and its possible modification (S--), must also be implemented without interruption.