CSE 120: Homework #2 Solutions

Fall 2015

1. Consider the following C program:

```
#include <stdlib.h>

int main (int argc, char *arg[])
{
    fork ();
    if (fork ()) {
        fork ();
    } else {
        char *argv[2] = {"/bin/ls", NULL};
        execv (argv[0], argv);
        fork ();
    }
}
```

How many total processes are created (including the first process running the program)? (Note that `execv` is just one of multiple ways of invoking `exec`.)

Hint: You can always add debugging code, compile it, and run the program to experiment with what happens.

a. 6 processes are created, including the first process. The process tree looks like:

```
30938
30938 -> 30939
         30939 -> 30940
                  30940 -> 30942
30938 -> 30941
         30941 -> 30943
```

Note that exec does not create a new process, it overwrites the currently running process with a new program. As a result, the branch in the program that calls exec will not call the fork in the else-clause.

b. The `/bin/ls` program executes twice.

2. Assume we are using multiple user-level threads in a program (multiplexed on just one kernel-level thread). If one thread in the program calls the system call fork() to create a child process, does the child process duplicate all the threads that were in the parent, or is the child process single-threaded? If instead a thread invokes exec(), will the program specified in the

parameter to exec() replace the entire process, including all of the threads?

Since we are using user-level threads, all thread management (code and data) is in a library in the address space of the process and invisible to the OS. As a result, `fork()` will duplicate all user-level threads in the child (just like all other code and data in the parent process).

Since `exec()` replaces an entire process with a new progeam, including the text, data, and stack segments, all user-level threads will indeed be replaced.

3. The Intel x86 instruction set architecture provides an atomic instruction called XCHG for implementing synchronization primitives. Semantically, XCHG works as follows (although keep in mind it is executed atomically by the CPU):

```
void XCHG (bool *X, bool *Y) {
  bool tmp = *X;
  *X = *Y;
  *Y = tmp;
}
```

Show how XCHG can be used instead of test-and-set to implement the acquire() and release() functions of the spinlock data structure described in the "Synchronization" lecture.

```
struct lock {
    bool isAvailable = TRUE;
}

void acquire (struct lock* p_lock) {
    bool isAvailable = FALSE;
    while (!isAvailable) {
        XCHG (&(p_lock->isAvailable), &isAvailable);
    }
}

void release (struct lock* p_lock) {
    p_lock->isAvailable = TRUE;
}
```

4. A common pattern in parallel scientific programs is to have a set of threads do a computation in a sequence of phases. In each phase i, all threads must finish phase i before any thread starts computing phase i+1. One way to accomplish this is with barrier synchronization. At the end of each phase, each thread executes Barrier::Done(n), where n is the number of threads in the computation. A call to Barrier::Done blocks until all of the n threads have called Barrier::Done. Then, all threads proceed. You may assume that the process allocates a new Barrier for each iteration, and that all threads of the program will call Done with the same value.

a. Write a monitor that implements Barrier using Mesa semantics.

```
monitor Barrier {
  int called = 0;
  Condition barrier;

  void Done (int needed) {
    called++;
    if (called == needed) {
      called = 0;
      barrier.Broadcast();
    } else {
      barrier.Wait();
    }
  }
}
```

b. Implement Barrier using an explicit lock and condition variable. The lock and condition variable have the semantics described at the end of the "Semaphore and Monitor" lecture in the ping_pong example, and as implemented by you in Project 1.

```
class Barrier {
  int called = 0;
  Lock lock;
  Condition barrier;

  void Done (int needed) {
    lock.Acquire();

    called++;
    if (called == needed) {
      called = 0;
      barrier.Broadcast(&lock);
    } else {
      barrier.Wait(&lock);
    }

    lock.Release();
  }
}
```

5. Microsoft .NET provides a synchronization primitive called a CountdownEvent. Programs use CountdownEvent to synchronize on the completion of many threads (similar to CountDownLatch in Java). A CountdownEvent is initialized with a *count*, and a CountdownEvent can be in two states, *nonsignalled* and *signalled*. Threads use a CountdownEvent in the nonsignalled state to *Wait* (block) until the internal count reaches zero. When the internal count of a CountdownEvent reaches zero, the CountdownEvent transitions to the signalled state and wakes up (unblocks) all waiting threads. Once a CountdownEvent has transitioned from nonsignalled to signalled, the CountdownEvent remains in the signalled state. In the nonsignalled state, at any time a thread may call the *Decrement* operation to decrease the count

and *Increment* to increase the count. In the signalled state, *Wait*, *Decrement*, and *Increment* have no effect and return immediately.

Use pseudo-code to implement a thread-safe CountdownEvent using locks and condition variables by implementing the following methods:

```
class CountdownEvent {
  ...private variables...
  CountdownEvent (int count) { ... }
  void Increment () { ... }
  void Decrement () { ... }
  void Wait () { ... }
}
```

Notes:

- The *CountdownEvent* constructor takes an integer *count* as input and initializes the CountdownEvent counter with *count*. Positive values of *count* cause the CountdownEvent to be constructed in the nonsignalled state. Other values of *count* will construct it in the signalled state.
- *Increment* increments the internal counter.
- *Decrement* decrements the internal counter. If the counter reaches zero, the CountdownEvent transitions to the signalled state and unblocks any waiting threads.
- *Wait* blocks the calling thread if the CountdownEvent is in the nonsignalled state, and otherwise returns.
- Each of these methods is relatively short.

```
class CountdownEvent {
  int counter;
  bool signalled;
  Lock lock;
  Condition cond;

  CountdownEvent (int count) {
    counter = count;
    if (counter > 0) {
      signalled = false;
    } else {
      signalled = true;
    }
    lock = new Lock ();
    cond = new Condition ();
  }

  void Increment () {
    lock.Acquire ();
    if (signalled == false) {
      counter++;
    } // otherwise do nothing if already signalled
    lock.Release ();
  }

  void Decrement () {
    lock.Acquire ();
    if (signalled == false) {
      counter--;
      if (counter == 0) {
        signalled = true;
        cond.Broadcast ();
      }
    } // otherwise do nothing if already signalled
    lock.Release ();
  }

  void Wait () {
    lock.Acquire ();
    if (signalled == false) {
      cond.Wait (&lock);
    } // otherwise do nothing if already signalled
    lock.Release ();
  }
}
```

6. [Silberschatz] Consider a system running ten I/O-bound tasks and one CPU-bound task. Assume that the I/O-bound tasks issue an I/O operation once for every millisecond of CPU computing and that each I/O operation takes 10 milliseconds to complete. Also assume that the context-switching overhead is 0.1 millisecond and that all processes are long-running tasks. What is the CPU utilization for a round-robin scheduler when:

   a. The time quantum is 1 millisecond

      In one iteration of the steady state behavior, a round-robin scheduler will run the CPU-bound task once for 1 ms, and each I/O-bound task once for 1ms each. There will be 11 context switches with 0.1 ms overhead each, so the total time taken for one iteration is 1 + (10 * 1) + (11 * 0.1) = 12.1 ms. The CPU was actually doing useful work for 11 ms, so the CPU utilization is 11 / 12.1 = 91%.

   b. The time quantum is 10 milliseconds

      In one iteration of the steady state behavior, a round-robin scheduler will run the CPU-bound task once for 10 ms, and each I/O-bound task once for 1ms each. There will be 11 context switches with 0.1 ms overhead each, so the total time taken for one iteration is 10 + (10 * 1) + (11 * 0.1) = 21.1 ms. The CPU was actually doing useful work for 20 ms, so the CPU utilization is 20 / 21.1 = 95%.

7. [Silberschatz] Explain the differences in the degree to which the following scheduling

algorithms discriminate in favor of short processes:

a. FCFS

First-come-first-serve does not discriminate at all in favor of short processes. FCFS just runs jobs as they arrive; it does not consider job length. With non-preemptive FCFS, the first job to complete is the first job to arrive.

b. RR

For a round-robin scheduler with reasonable time quanta, short processes usually complete before long processes. In this way, a round-robin scheduler favors short processes - short processes are more likely to run to completion first, compared to FCFS, where jobs that arrive first complete first.

c. Multilevel feedback queues

A multilevel feedback scheduler actively discriminates in favor of short processes by maintaining separate queues for short ("interactive") processes and long-running ("CPU-bound") processes. Jobs in the interactive queue always run first, so jobs in the CPU-bound queue get to run only when no interactive jobs are ready to run.

8. Annabelle, Bertrand, Chloe and Dag are working on their term papers in CSE 120, which is a 10,000 word essay on *My All-Time Favorite Race Conditions*. To help them work on their papers, they have one dictionary, two copies of Roget's Thesaurus, and two coffee cups.

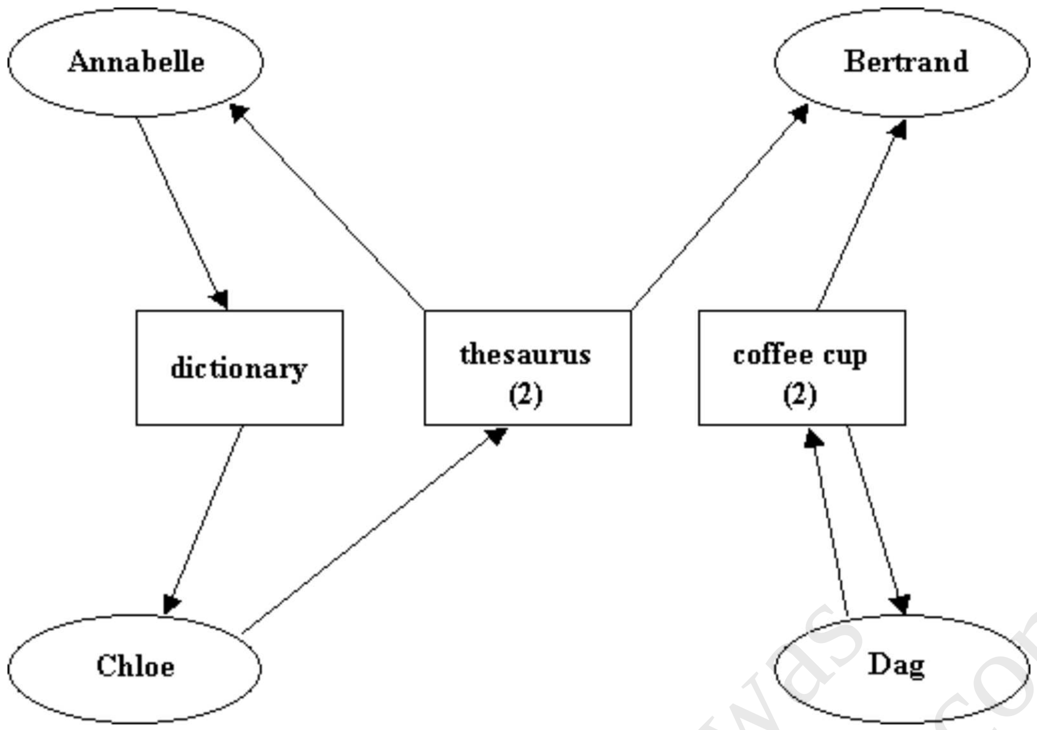
   - Annabelle needs to use the dictionary and a thesaurus to write her paper;
   - Bertrand needs a thesaurus and a coffee cup to write his paper;
   - Chloe needs a dictionary and a thesaurus to write her paper;
   - Dag needs two coffee cups to write his paper (he likes to have a cup of regular and a cup of decaf at the same time to keep himself in balance).

   Consider the following state:

   - Annabelle has a thesaurus and needs the dictionary.
   - Bertrand has a thesaurus and a coffee cup.
   - Chloe has the dictionary and needs a thesaurus.
   - Dag has a coffee cup and needs another coffee cup.

   a. Is the system deadlocked in this state? Explain using a resouce allocation graph.

   The resource allocation graph, shown below, can be fully reduced. Bertrand is not blocked. Erasing the edges incident on Bertrand unblocks Chloe and Dag. Erasing their edges unblocks Annabelle.



b. Is this state reachable if the four people allocated and released their resources using the Banker's algorithm? Explain.

Yes. The resource allocation graph is also the maximum claims graph. Since it can be fully reduced, the state is attainable using the Banker's algorithm.

---

voelker@cs.ucsd.edu