CSE120
Homework_2
ZE LI
A11628864

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 ();
    }
}
```

a. There has six processes been created

b. program executes twice

2. code:

```
struct lock {
    int check;
    //default 0
}

void acquire (struct lock *) {
    int check = 1;
    while(check = 0){
        XCHG(&lock->check, &check);
    }
}

void release (struct lock *) {
    int check = 0;

}
```

3.

a. context switches: main(), A() , B() , A(), main()

b. Output: fee, foe, foo, far, fie, fum, fun
c. Thread queues when selfTest returns:
currentThread: main()
readyQueue: none
joinwait queue: none

4.

a.
```
Monitor Barrier{
        int called = 0;
        Condition barrier;
}

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

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

void Finished (int n){
        lock.acquire();

        called++;
        if(called == n){
                called = 0;
                barrier.broadcast(lock);
        }else{
                barrier.wait(lock)
        }
        lock.release();
}
}
```

5.
(Base on homework 2 solution)

```
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.

a. using round-robin scheduler, 11 context switches with 0.1ms, so 1+(10)+(11*0.1) = 12.1 ms, using slice divide by 12.1 => 11/12.1 * 100 = 91%

b. quantum is 10 milliseconds

Cpu bound task once for 10ms, and 11context switches with 0.1 ms, 10+(10)+(11*0.1) = 21.1ms and 20 / 21.1 * 100 = 95%.

7.

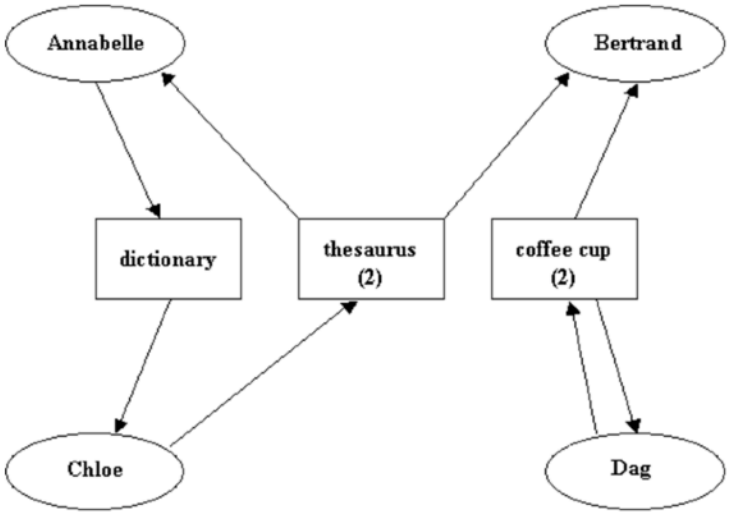
a. FCFS - First come first serve, with non-preemptive FCFS, the first job to complete is the first job to arrive.

b. RR - round-robin scheduler with reasonable time quanta, short processes are more likely to run to completion first, compared to FCFS, jobs that arrive first complete first.

c. Multilevel feedback queues - 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.

a.



b.

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