CS 475 Fall 2017

Assignment 2 - PART A Solutions

DUE 11:59 PM Wednesday, October 18

Instructions:

- PART A of Assignment 2 is to be completed **individually** by each student. No collaboration among students is permitted.
- Please upload your solutions to PART A to Blackboard in PDF format.
- **Late submissions will not be graded. There are no slip days for Assignment 2 - PART A.**

Problem 1. (15 points):

This problem tests your understanding of exceptional control flow in C programs. Assume we are running code on a Unix machine. The following problems all concern the value of the variable `counter`.

Part I

```
int counter = 0;

int  main()
{
    int i;

    for (i = 0; i < 3; i ++){
        fork();
        counter ++;
        printf("counter = %d\n", counter);
    }

    printf("counter = %d\n", counter);
    return 0;
}
```

A. How many times would the value of `counter` be printed: _____22_____

B. What is the value of `counter` printed in the first line? _____1_____

C. What is the value of `counter` printed in the last line? _____3_____

Part II

```
pid_t pid;
int counter = 0;

void handler1(int sig)
{
    counter--;
    printf("counter = %d\n", counter);
    fflush(stdout);   /* Flushes the printed string to stdout */
    kill(pid, SIGUSR1);
}

void handler2(int sig)
{
    counter += 2;
    printf("counter = %d\n", counter);
    exit(0);
}

main() {
    signal(SIGUSR1, handler1);
    if ((pid = fork()) == 0) {
        signal(SIGUSR1, handler2);
        kill(getppid(), SIGUSR1);
        while(1) {};
    }
    else {
        pid_t p; int status;
        if ((p = wait(&status)) > 0) {
            counter += 3;
            printf("counter = %d\n", counter);
        }
    }
}
```

What is the output of this program?

```
counter = -1
counter = 2
counter = 2
```

Part III

```
int counter = 0;

void handler(int sig)
{
    counter = counter + 3;
}


int  main()
{
    int i;

    signal(SIGCHLD, handler);

    for (i = 0; i < 5; i ++){
        if (fork() == 0){
            exit(0);
        }
    }

    /* wait for all children to die */
    while (wait(NULL) != -1);

    printf("counter = %d\n", counter);
    return 0;
}
```

A. Does the program output the same value of `counter` every time we run it? `No`

B. If the answer to A is `Yes`, indicate the value of the `counter` variable. Otherwise, list all possible values of the `counter` variable.

Answer: `counter = 3,6,9,12,15`_____

Problem 2. (25 points):

You have been hired by a company to work on a program that does climate modeling. The inner loop of the program matches atoms of different types as they form molecules. In an excessive reliance on threads, each atom is represented by a thread.

A Your task is to write code to form carbon dioxide out of one carbon thread and two oxygen threads (C02). You are to write the two procedures: `CArrives()` and `OArrives()`. A carbon dioxide molecule forms when two O threads are present and one C thread; otherwise the atoms must wait. Once all three are present, one of the threads calls `MakeCO2()`, and only then, all three depart.

B Extending the product line into beer production, your task is to write code to form alcohol (C2H60) out of two carbon atoms, six hydrogens, and one oxygen.

You must use locks and Mesa-style condition variables to implement your solutions. Obviously, an atom that arrives after the molecule is made must wait for a different group of atoms to be present. There should be no busy-waiting. There should also be no useless waiting: atoms should not wait if there is a sufficient number of each type to make a particular molecule.

There are a number of different ways to solve this problem. One solution is given below.

```
/* A. Make CO2 */

integer oxygen = 0;
mutex_lock lock;
cond_variable carbon_cv, oxygen_cv;

CArrives() {
     lock.acquire();
     while(oxygen < 2)
       carbon_cv.wait(lock);
     MakeCO2() //where oxygen -= 2
     for i = 1 to 2 do
       oxygen_cv.signal();
     lock.release();
}

OArrives() {
     lock.acquire();
     oxygen++;
     carbon_cv.signal();
     oxygen_cv.wait(lock);
     lock.release();
}
```

```
/* B. Make C2H6O */

integer carbon = 0, hydrogen = 0;
mutex_lock lock;
cond_variable carbon_cv, oxygen_cv, hydrogen_cv;

OArrives() {
      lock.acquire();
      while(carbon < 2 AND hydrogen < 6)
          oxygen_cv.wait(lock);
      MakeBeer() //where carbon -= 2, hydrogen -= 6
      for i = 1 to 2 do
        carbon_cv.signal();
      for i = 1 to 6 do
        hydrogen_cv.signal();
      lock.release();
}

CArrives() {
      lock.acquire();
      carbon++;
      oxygen_cv.signal();
      carbon_cv.wait(lock);
      lock.release();
}

HArrives() {
      lock.acquire();
      hydrogen++;
      oxygen_cv.signal();
      hydrogen_cv.wait(lock);
      lock.release();
}
```

Problem 3. (10 points):

We explore the so-called barbershop problem. A barbershop consists of N waiting chairs and the barber chair. If there are no customers, the barber waits. If a customer enters, and all the waiting chairs are occupied, then the customer leaves the shop. If the barber is busy, but waiting chairs are available, then the customer sits in one of the free chairs.

Here is the skeleton of the code, without synchronization.

There are a number of solutions with one given below. For this solution, initial values are mutex = 1 (for protecting the variable `num_customers`), customer = 0 (no customers), and barber = 0 (barber is not busy)

```
extern int N;              /* initialized elsewhere to value > 0 */
int num_customers = 0;

void* customer() {
    P(&mutex);
    if (num_customers > N) {
      V(&mutex);
     return NULL;
    }

    num_customers += 1;
     V(&mutex);
     V(&customer);
     P(&barber);
    getHairCut();
    P(&mutex);
    num_customers -= 1;
   V(&mutex);
   return NULL;
}

void* barber() {
    while(1) {
         P(&customer);
         V(&barber);
         cutHair();

   }
}
```

For the solution, we use three binary semaphores:

- `mutex` to control access to the global variable `num_customers`.
- `customer` to signal a customer is in the shop.
- `barber` to signal the barber is busy.

1 Indicate the initial values for the three semaphores.

- `mutex` 1
- `customer` 0
- `barber` 0

2 Complete the code on the previous page filling in as many copies of the following commands as you need, but no other code.

```
P(&mutex);
V(&mutex);
P(&customer);
V(&customer);
P(&barber);
V(&barber);
```

Problem 4. (10 points):

Consider the following three threads and four semaphores:

```
/* Initialize x */
 x = 1;
/* Initialize semaphores */
s1 =  1;
s2 =  3;
s3 = 3;
s4 = 2;
```

```
void thread1()
{
  while (x != 5400)
  {
    P(s2);
    P(s1);
    x = x * 2;
    V(s1);
  }
  exit(0);
}
```

```
void thread2()
{
  while (x != 5400)
  {
     P(s3);
     P(s1);
     x = x * 3;
     V(s1);
  }
  exit(0);
}
```

```
void thread3()
{
  while (x != 5400)
  {
     P(s4);
     P(s1);
     x = x * 5;
     V(s1);
  }
  exit(0);
}
```

Provide initial values for the four semaphores and add P(), V() semaphore operations (using the four semaphores) in the code for thread 1, 2 and 3 such that the process is guaranteed to terminate.