Homework 6

ComS 352 - Gavin Monroe - gmonroe

1)Exercise 6.12

The value of the semaphore may be > 0when a process callsgetValue(); however, before the process can call wait(), another process maycall wait(), and the value of the semaphore may change to 0. It defeats thepurpose.

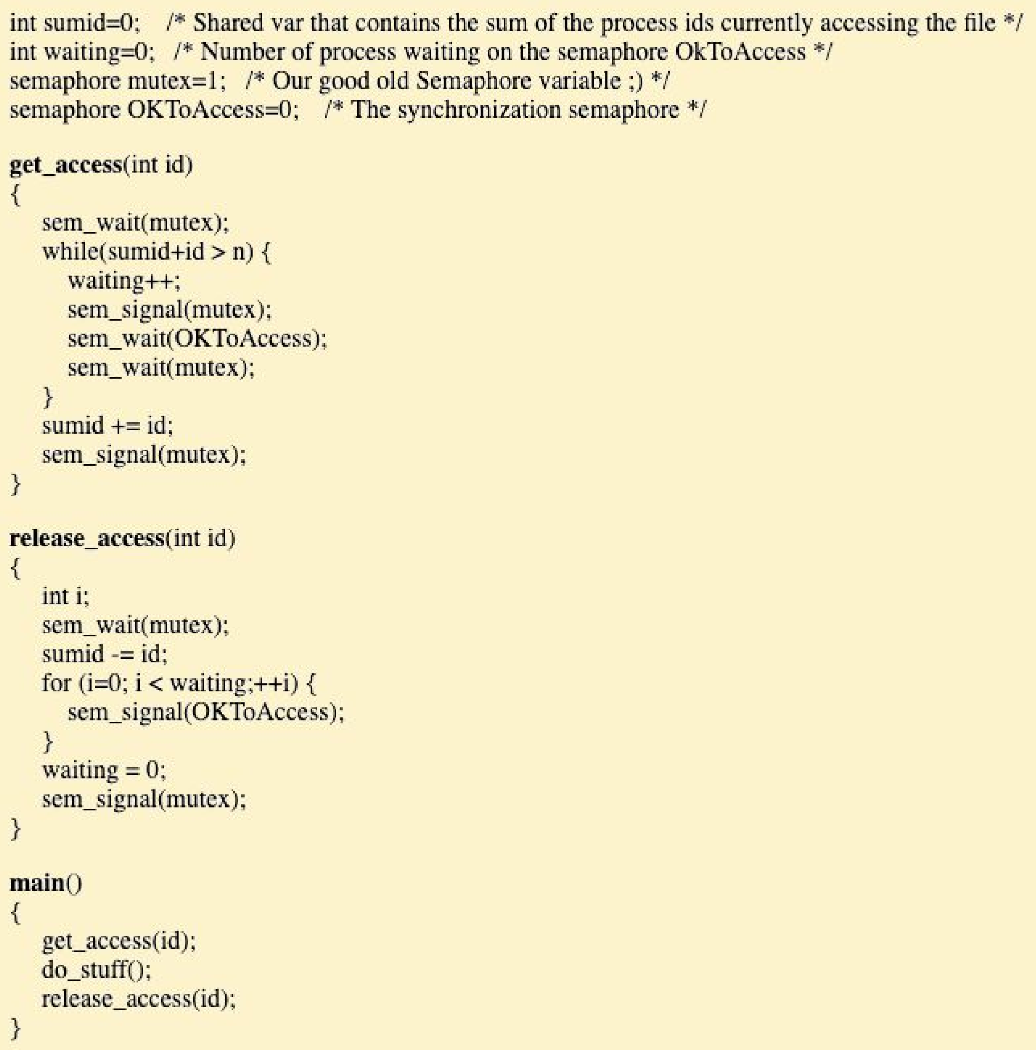
2)Exercise 6.26

The signal( ) operations related to monitors are no longer persistent. If a signal is performed and if there are no waiting threads, then the sign is really not noted and the device does no longer take note of the fact that the signal took place. If a subsequent wait operation is performed, then the corresponding thread surely blocks.

Whereas in semaphores, each sign consequences in a corresponding increment of the semaphore price even if there are no ready threads. A future wait operation would without delay prevail due to the fact of the until now increment.

Continue to page 2

3)Exercise 6.29 (10 points) Your solution should use condition variables. Assume a broadcast() operation can be invoked on a condition variable c to resume all processes suspended on c. Hint: Your monitor should contain two functions: one function is called before a process accesses a file and the other function is called after a process accesses a file.

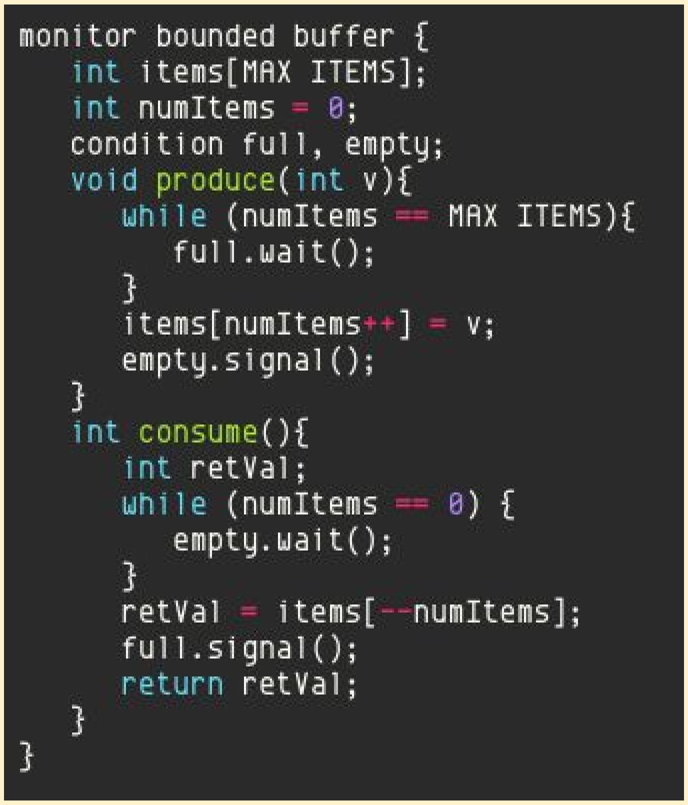


```
int sumid=0;    /* Shared var that contains the sum of the process ids currently accessing the file */
int waiting=0;   /* Number of process waiting on the semaphore OkToAccess */
semaphore mutex=1;   /* Our good old Semaphore variable ;) */
semaphore OKToAccess=0;    /* The synchronization semaphore */

get_access(int id)
{
    sem_wait(mutex);
    while(sumid+id > n) {
        waiting++;
        sem_signal(mutex);
        sem_wait(OKToAccess);
        sem_wait(mutex);
    }
    sumid += id;
    sem_signal(mutex);
}

release_access(int id)
{
    int i;
    sem_wait(mutex);
    sumid -= id;
    for (i=0; i < waiting;++i) {
        sem_signal(OKToAccess);
    }
    waiting = 0;
    sem_signal(mutex);
}

main()
{
    get_access(id);
    do_stuff();
    release_access(id);
}
```

- Release_access wakes up all ready processes. It is NOT adequate to wake up the first ready process in this trouble --- that system might also have too giant a value of id. Also, greater than one system can also be eligible to go in (if these techniques have small ids) when one manner releases access.
- Woken up techniques attempt to see if they can get in. If not, they go again to sleep.
- Waiting is set to zero in release_access after the for loop. That avoids useless signals from subsequent release_accesses. Those indicators would not make the solution wrong, simply less environment friendly as processes will be woken up unnecessarily.

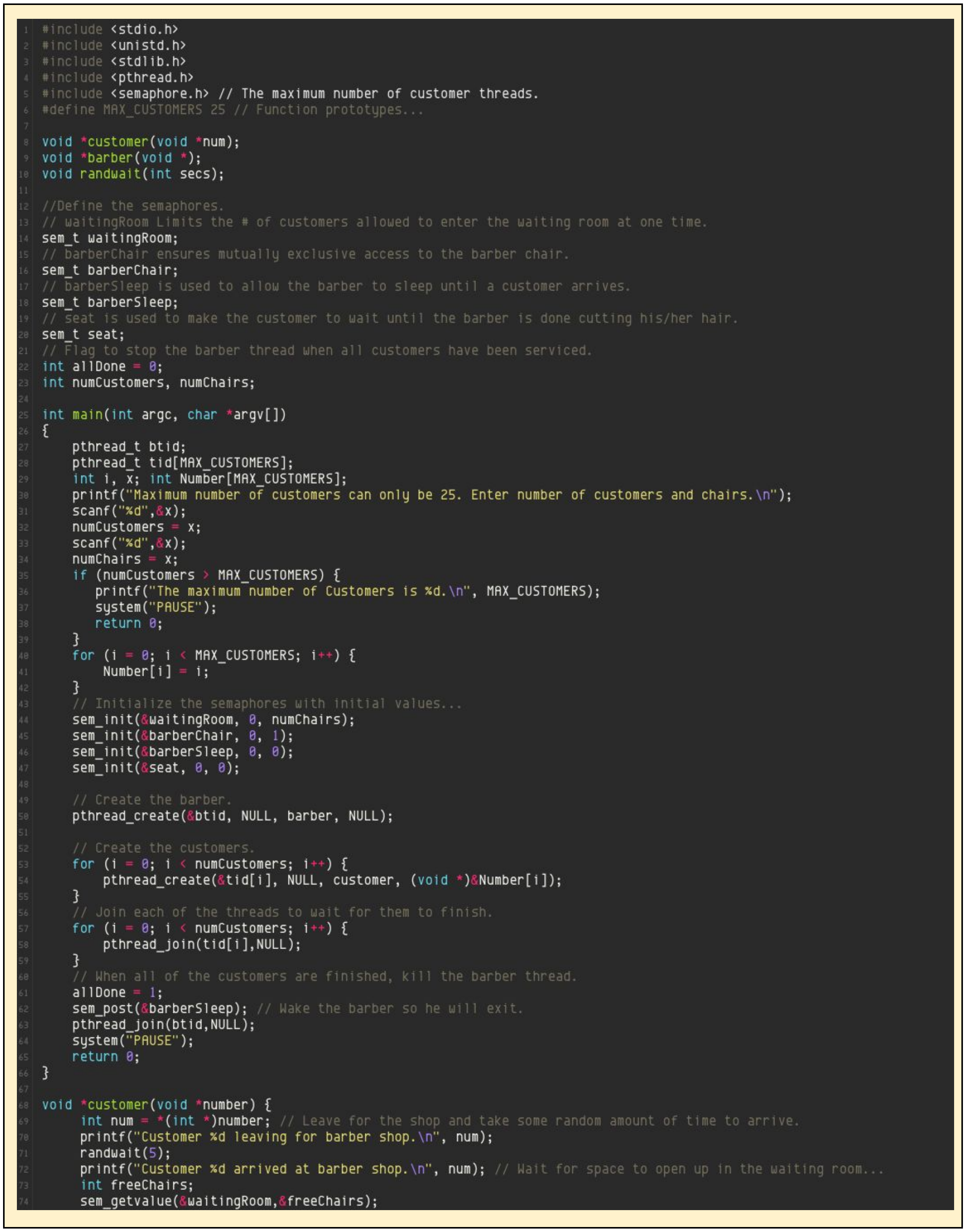
4)Exercise 7.9 (14 points) Assume the buffer can hold MAX_ITEMS items and each item is an integer. You should write two functions:
void produce(int v) - This function is called by the producer to put item v in the buffer
int consume() - This function is called by the consumer to remove an item in the buffer and returns the item

The code is as follows

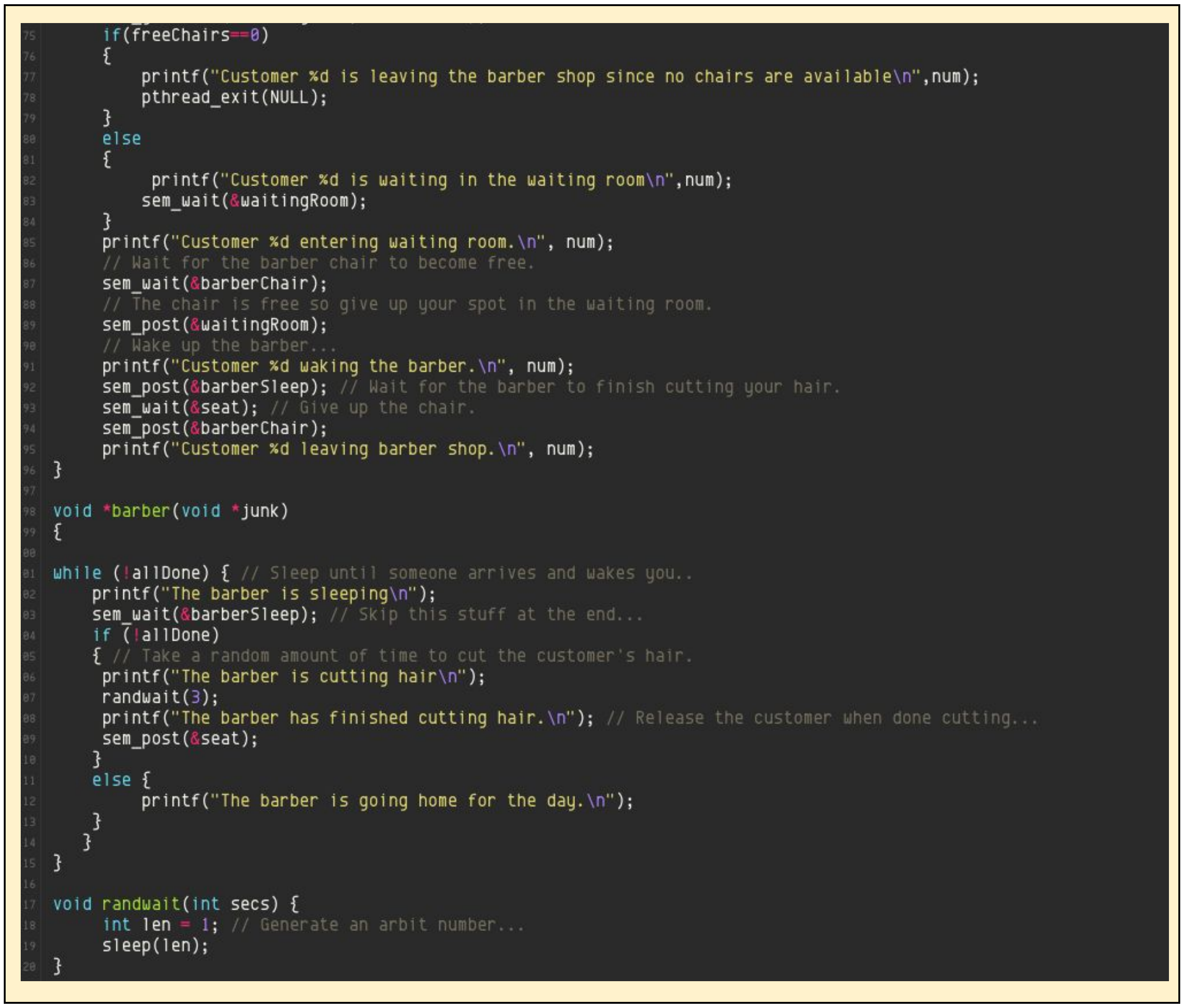


```
monitor bounded buffer {
   int items[MAX ITEMS];
   int numItems = 0;
   condition full, empty;
   void produce(int v){
      while (numItems == MAX ITEMS){
         full.wait();
      }
      items[numItems++] = v;
      empty.signal();
   }
   int consume(){
      int retVal;
      while (numItems == 0) {
         empty.wait();
      }
      retVal = items[--numItems];
      full.signal();
      return retVal;
   }
}
```

5)5. (16 points) Consider the following sleeping barber problem. There is a barber shop which has one barber, one barber chair, and N chairs for waiting customers. If there is no customer, then the barber sleeps in his own chair. When a customer arrives, he has to wake up the barber. If there are many customers and the barber is cutting a customer's hair, then the remaining customers either wait if there are empty chairs in the waiting room or they leave if no chairs are empty. Write a semaphore solution to the sleeping barber problem. You should write two procedures: barber() and customer().



```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h> // The maximum number of customer threads.
#define MAX_CUSTOMERS 25 // Function prototypes...

void *customer(void *num);
void *barber(void *);
void randwait(int secs);

//Define the semaphores.
// waitingRoom Limits the # of customers allowed to enter the waiting room at one time.
sem_t waitingRoom;
// barberChair ensures mutually exclusive access to the barber chair.
sem_t barberChair;
// barberSleep is used to allow the barber to sleep until a customer arrives.
sem_t barberSleep;
// seat is used to make the customer to wait until the barber is done cutting his/her hair.
sem_t seat;
// Flag to stop the barber thread when all customers have been serviced.
int allDone = 0;
int numCustomers, numChairs;

int main(int argc, char *argv[])
{
    pthread_t btid;
    pthread_t tid[MAX_CUSTOMERS];
    int i, x; int Number[MAX_CUSTOMERS];
    printf("Maximum number of customers can only be 25. Enter number of customers and chairs.\n");
    scanf("%d",&x);
    numCustomers = x;
    scanf("%d",&x);
    numChairs = x;
    if (numCustomers > MAX_CUSTOMERS) {
       printf("The maximum number of Customers is %d.\n", MAX_CUSTOMERS);
       system("PAUSE");
       return 0;
    }
    for (i = 0; i < MAX_CUSTOMERS; i++) {
        Number[i] = i;
    }
    // Initialize the semaphores with initial values...
    sem_init(&waitingRoom, 0, numChairs);
    sem_init(&barberChair, 0, 1);
    sem_init(&barberSleep, 0, 0);
    sem_init(&seat, 0, 0);

    // Create the barber.
    pthread_create(&btid, NULL, barber, NULL);

    // Create the customers.
    for (i = 0; i < numCustomers; i++) {
        pthread_create(&tid[i], NULL, customer, (void *)&Number[i]);
    }
    // Join each of the threads to wait for them to finish.
    for (i = 0; i < numCustomers; i++) {
        pthread_join(tid[i],NULL);
    }
    // When all of the customers are finished, kill the barber thread.
    allDone = 1;
    sem_post(&barberSleep); // Wake the barber so he will exit.
    pthread_join(btid,NULL);
    system("PAUSE");
    return 0;
}

void *customer(void *number) {
     int num = *(int *)number; // Leave for the shop and take some random amount of time to arrive.
     printf("Customer %d leaving for barber shop.\n", num);
     randwait(5);
     printf("Customer %d arrived at barber shop.\n", num); // Wait for space to open up in the waiting room...
     int freeChairs;
     sem_getvalue(&waitingRoom,&freeChairs);
```



```
     if(freeChairs==0)
     {
         printf("Customer %d is leaving the barber shop since no chairs are available\n",num);
         pthread_exit(NULL);
     }
     else
     {
          printf("Customer %d is waiting in the waiting room\n",num);
         sem_wait(&waitingRoom);
     }
     printf("Customer %d entering waiting room.\n", num);
     // Wait for the barber chair to become free.
     sem_wait(&barberChair);
     // The chair is free so give up your spot in the waiting room.
     sem_post(&waitingRoom);
     // Wake up the barber...
     printf("Customer %d waking the barber.\n", num);
     sem_post(&barberSleep); // Wait for the barber to finish cutting your hair.
     sem_wait(&seat); // Give up the chair.
     sem_post(&barberChair);
     printf("Customer %d leaving barber shop.\n", num);
}

void *barber(void *junk)
{

while (!allDone) { // Sleep until someone arrives and wakes you..
    printf("The barber is sleeping\n");
    sem_wait(&barberSleep); // Skip this stuff at the end...
    if (!allDone)
    { // Take a random amount of time to cut the customer's hair.
     printf("The barber is cutting hair\n");
     randwait(3);
     printf("The barber has finished cutting hair.\n"); // Release the customer when done cutting...
     sem_post(&seat);
    }
    else {
         printf("The barber is going home for the day.\n");
    }
   }
}

void randwait(int secs) {
     int len = 1; // Generate an arbit number...
     sleep(len);
}
```