Programming Assignment 3

Due 11:59pm Sunday, February 14

Introduction

This programming assignment is on the topic of synchronization, and consists of 4 parts, A, B, C, and D. Each part consists of a set of exercises which you should do.

You will hand in TWO FILES: **mycode3.c** and **pa3d.c**

1. your modifications to the operating system in **mycode3.c**
2. your solution to Exercise D contained in **pa3d.c**

Each of the programs below can be found in separate files, ***pa3a.c, pa3b.c, pa3c.c,*** and ***pa3d.c.***

To install this assignment using your account on ieng9:

1. Log in to ieng9.ucsd.edu using your class account.

2. Enter in the command, "prep cs120w". This configures your account for this class. You should always do this when doing CSE120 work.

3. Enter in the command, "getprogram3". This will create a directory called"pa3" and will copy the relevant files into that directory. It is important that you do all your work in this directory (i.e., called "pa3" in your home directory). Do NOT create another directory (i.e., with a name different from pa3) to do your work, as this may have an unknown effect on how UMIX runs. You may certainly save your work by copying it to another directory, but your latest work with any changes you make should be in, and all your executions should be run from within, pa3.

4. To compile, enter the command "make" (from within the pa3 directory). This will compile all of the programs. You can compile a particular program by specifying it as a parameter to make, e.g., "make pa3a".

5. To turn in the assignment, make sure that BOTH mycode3.c and pa3d.c files that you need to turn in are in the current directory and enter the command "turninprogram3".

Notes on Grading

---

1. You will be graded primarily on whether your code works and how well it works (is it efficient in time and/or space). While you will not be graded specifically on code structure and commenting/documentation, it is to your benefit to use good software engineering principles in structuring and commenting your code. This will especially benefit you if your code requires visual inspection by a grader. If a problem arises and we can't understand your code, or it is difficult to do so, your grade may suffer.

2. Unless indicated otherwise, you must NOT use any library routines or any code that you did not write yourself other than the routines given to you. For example, if you need a data structure like a linked list or queue, you should create it yourself rather than relying on a C library.

3. You should NOT use dynamic memory allocation in your kernel. For example, you should not use malloc (both because it is a dynamic memory allocator, and it is a C library routine which, as indicated above, you should not use). Since any dynamic memory allocator may fail (if all the memory is used), the kernel cannot depend on it (otherwise it might fail, which would be catastrophic).

4. It is your responsibility to proactively come up with YOUR OWN tests that you think are necessary to convince yourself that your kernel is doing what is asked for in the specification. If your code passes tests provided by anyone on the CSE 120 teaching staff, you should not assume that your code "works" and you are done. What ultimately matters as far as what your code is expected to do will be based on the specification you are given (i.e., these instructions). It is up to YOU to interpret it and devise any test cases you think are applicable. This mimics the experience of a real operating system designer/implementer, who has no idea what kind of applications will be written for their operating system, and how their code will be exercised. The real operating system implementer must test for anything and everything, as best they can. So, you must test robustly, and be creative in coming up with tests. You are free to ask questions about cases that you think may matter, and even post tests you think are worthy of sharing (in fact, we encourage this!).

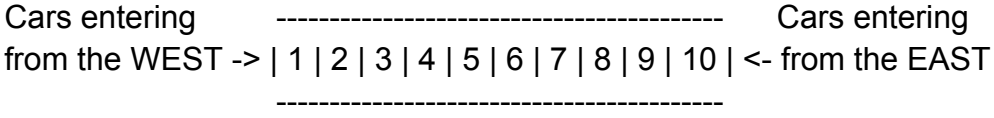
5. All your code must be contained in **mycode3.c and pa3d.c** . ALL OTHER CHANGES YOU MAKE TO ANY OTHER FILES WILL BE COMPLETELY IGNORED! Consequently, do not put declarations, function or variable definitions, defined constants, or anything else in other files that you expect to be part of your solution. We will compile your submitted code with our test programs, and so your entire solution must be contained in what you submit. If your code does not compile with our test programs because you made changes to other files that required inclusion, YOU WILL RECEIVE ZERO CREDIT, so please be careful!

6. There should be no Printf statements in your **mycode3.c**, as if there are, it will disrupt the autograder's ability to check your program and you may lose points. For debugging, you may wish to use DPrintf, which is used and works exactly like Printf, except that it generates no output during the autograder's testing, so it is very safe to use.

Programming Assignment 3: Exercise A

General Overview: In this assignment, you will implement semaphores, and use them in a program that simulates cars sharing a single-lane road and that require synchronization.

In this first exercise, we begin by studying a simulation program that creates multiple processes, where each process simulates a car entering, proceeding over, and exiting, a single-lane road. The road is 10 miles long, and consists of **10 positions** (each of length 1 mile).



Cars may enter the road from the West, at position 1, or they may enter from the East, at position 10. Cars enter and proceed at a certain speed (expressed in MPH, miles per hour) and maintain that speed until they exit.

The primary rule of the road is that each position can be occupied by at most 1 car at any time. If a car tries to proceed to a position that is already occupied, a **CRASH** occurs. When a crash occurs, both cars get automatically removed from the road (allowing other cars to proceed).

In the program below, two cars drive over the road in the same direction, both entering from the West (in pos 1). Car 1 drives at 40 MPH, and Car 2 at 60 MPH. The procedure ***driveRoad*** simulates this activity: it takes as parameters *the entrance point (EAST or WEST)*, and *speed (in MPH)*. To make sure that both cars do not collide by entering the road at the same time, Car 2 waits some time before entering. This is implemented by the procedure ***Delay(t)*** which takes a time (in seconds) as a parameter. In the program, Car 2 delays by 900 seconds. Since Car 1 drives at 40 MPH, it will travel the 10-mile road in 15 minutes, or 900 seconds. Hence, Car 2 will enter at just about the time that Car 1 exits.

Exercises

1. Run the program and note the output. A line is printed each time a car makes a move (either to enter, proceed, or exit, the road) or if a collision occurs.

2. Since Car 2 is delayed by 900 seconds, it will have waited until Car 1 exits (note that times are NOT exact in the simulation, so Car 1 may or may not exit right at the time Car 2 enters). Modify the program so that Car 2 delays by only 450 seconds. What happens, and why?

3. Now modify the program so Car 2 delays by only 300 seconds. What happens, and why? Run the program a few times to see if the same events occur in the same way each time.

4. Modify the program so Car 2 delays by 0 seconds, just to see where the crash occurs.

5. Now change the direction of Car 2 so that it enters from the East (thus, starting at position 10). Leave the delay at 0. Where does the crash occur, and why?

6. Still having both cars go in opposite directions, modify the delay so that a crash does not occur. How large does the delay have to be?

7. Study the implementation of ***driveRoad***. In addition to the ***Delay*** procedure, it makes use of two other procedures, ***EnterRoad***, and ***ProceedRoad***. ***EnterRoad*** causes a car to enter the road, and takes a parameter indicating the car's point of entrance. ***ProceedRoad*** simply causes the car to move one position (one mile), in the direction it entered. Notice how movement is implemented by having the cars delay by 3600/mph, the number of seconds in an hour divided by the speed in miles per hour.

```
#include <stdio.h>
#include "aux.h"
#include "sys.h"
#include "umix.h"

int driveRoad(int from, int mph);

void Main()
{
     if (Fork() == 0) {
                              // car 2
          Delay(900);              // wait 900 simulated secs
          driveRoad(WEST, 60);          // enter from West at 60 mph
          Exit();
     }

     driveRoad(WEST, 40);              // car 1

     Exit();
}

#define IPOS(FROM)  (((FROM) == WEST) ? 1 : NUMPOS)

int driveRoad(int from, int mph)
     // from: coming from which direction
     // mph: speed of car
{
     int p;
```

```
        EnterRoad(from);

        for (p = 1; p <= NUMPOS; p++) {

            Delay(3600/mph);          // MUST NOT MODIFY!

            ProceedRoad();
        }
}
```

Programming Assignment 3: Exercise B

One way to solve the "race condition" causing the cars to crash is to add synchronization directives that cause cars to wait for others. In this assignment, we will use semaphores. The kernel supports a large number of semaphores (defined by MAXSEMS in sys.h, currently set to 100), and each semaphore is identified by an integer 0 - 99 (MAXSEMS-1).

Semaphore operations are:

```
int s = Seminit(int v)
```

Allocates a semaphore and initializes its value to v.
Returns a unique identifier s of the semaphore, which is then used to refer to the semaphore in Wait and Signal operations. Returns -1 if unsuccessful (e.g., if there are no available semaphores).

```
int Wait(int s)
```

Semaphore wait operation.
Returns -1 if unsuccessful, else 0 (success).

```
int Signal(int s)
```

Semaphore signal operation.
Returns -1 if unsuccessful, else 0 (success).

The above are system calls that can be called by user processes. These procedures cause a trap into the kernel, and each calls a corresponding procedure located in mycode3.c:

```
int s = MySeminit(int v)
```

```
int MyWait(int s)

int MySignal(int s)
```

Given these interfaces, you are to implement these routines.

One additional note about semaphores in UMIX: Once a semaphore is created by a process, that semaphore is available for use by all processes. So, even a process that did not create the semaphore may use it by calling Wait(s) and Signal(s), where s is the semaphore identifier. This is because semaphores are implemented in the kernel, and thus are available to (shared by) all processes. A separate question is: How do all the processes that are to use a semaphore learn what its integer identifier is (after all, only one process created the semaphore, and so the identifier is initially known only to that process). The solution is to place the variable that stores the identifier into shared memory (to be discussed in Part C).

Exercises

1. Study the program below. Process 1 (Car 1) allocates a semaphore, storing its ID in sem, and initializes its value to 0. It then creates process 2 (Car 2) which immediately executes Wait(sem). What should happen to process 2 given that sem is initialized to 0? After driving over the road, process 1 executes Signal(sem). What should happen to process 2?

2. Study the file **mycode3.c**. It contains a skeletal data structure and code for the semaphore operations. Notice how ***MySeminit*** finds a free entry in the semaphore table, allocates it, initializes it, and uses the index as the semaphore ID that is returned. The other routines, ***MyWait*** and ***MySignal*** have minimal bodies that decrement and increment the semaphore value, but have no effect on synchronization. To implement synchronization, you need two utility kernel functions that are provided to you:

Block() causes the current process to block. It basically removes the current process from being eligible for scheduling, and causes a context switch to another process that is eligible.

Unblock(int p) causes process p to be eligible for scheduling. This does not mean p will execute immediately, but only that when a scheduling decision is made, p may be selected.

Given these utility routines, implement the semaphore routines.

NOTE: The kernel already enforces atomicity of ***MySignal*** and ***MyWait***, so you do NOT need to implement any additional mechanisms for atomicity.

3. Run the program below. It should now cause Car 2 to wait for Car 1 before driving over the road, thus avoiding a crash.

```
#include <stdio.h>
#include "aux.h"
#include "sys.h"
```

```
#include "umix.h"

void driveRoad(int from, int mph);

void Main()
{
     int sem;

     sem = Seminit(0);

     if (Fork() == 0) {
          Wait(sem);               // car 2
          Delay(0);
          driveRoad(EAST, 60);
          Exit();
     }

     driveRoad(WEST, 40);              // car 1
     Signal(sem);

     Exit();
}

#define IPOS(FROM)  (((FROM) == WEST) ? 1 : NUMPOS)

void driveRoad(int from, int mph)
     // from: coming from which direction
     // mph: speed of car
{
     int p;

     EnterRoad(from);

     for (p = 1; p <= NUMPOS; p++) {

          Delay(3600/mph);         // MUST NOT MODIFY!

          ProceedRoad();
     }
}
```

Programming Assignment 3: Exercise C

How to Use Shared Memory

Processes in UMIX are able to share memory. An explicit mechanism is needed because the default for processes is that, as in Unix, their memories are independent of each other, with no portion being shared. When a Fork occurs, a new process is created with its own memory that is initialized with the state of the memory of the creating process. But from that point on, when one of the processes modifies its private memory, the private memory of the other process is unaffected. Hence, the need for a shared memory area.

Two processes can share memory by identifying and registering variables to be shared. The variables in each process should be the same size and type. To share a set of variables, one can simply define a struct containing the variables as members and then share a variable declared as the struct. The following example will clarify this. For two processes to share, say, three integers, a struct should be defined in both programs as follows:

```
struct {
    int x, y, z;
} shm;
```

This structure defines three integers x, y, and z, as its members, and the variable shm is declared of this structure's type.

Within the program, shm must be registered with the operating system as a shared variable using the ***Regshm system call***:

```
Regshm((char ) &shm, sizeof(shm))
```

The first parameter is the address of the shared (struct) variable, and the second is its size. The latter is needed because there is a maximum size (MAXSHM) that the operating system supports for shared memory.

From then on, you can reference any of the shared variables via the expression shm.x, where shm is the name of the addressing variable and x is a member of the shared variable structure.

Variables that become shared memory should NOT be used prior to the call to Regshm (or if they are used, their values will be wiped away after the Regshm returns). Consequently, all initialization of shared memory variables should happen AFTER the call the Regshm, and not before.

If a process has a registered shared memory area, then any processes it creates will inherit the shared memory, i.e., the child processes do not have to call Regshm. Otherwise, processes need to each call Regshm to share memory.

If Regshm is called more than once, the latest registration supercedes the previous ones (i.e., only the last one matters, the others are ignored). Typically, all the variables that should be shared are placed in a struct as shown above, and then registered once, with no more registrations.

A sample program is given below. Note that the output of the program is actually indeterminate because without any synchronization (e.g., semaphores), Process 2 may print before Process 1 sets the variables.

```
#include <stdio.h>
#include "aux.h"
#include "sys.h"
#include "umix.h"

struct {       // structure of variables to be shared
     int x;          // example of an integer variable
     char y[10];     // example of an array of character variables
} shm;

void Main()
{
     Regshm((char ) &shm, sizeof(shm));      // register as shared

     if (Fork() == 0) {

          /Proc 2 inherits the shared memory registered by Proc 1

          Printf("P2: x = %d, y[3] = %c\n", shm.x, shm.y[3]);
          Exit();
     }

     shm.x = 1062;
     shm.y[3] = 'a';
     Printf("P1: x = %d, y[3] = %c\n", shm.x, shm.y[3]);
}
```

Programming Assignment 3: Exercise D

Now that you have a working implementation of semaphores, you can implement a more sophisticated synchronization scheme for the car simulation.

Study the program below. Car 1 begins driving over the road, entering from the East at 40 mph. After 900 seconds, both Car 3 and Car 4 try to enter the road. Car 1 may or may not have exited by this time (it should exit after 900 seconds, but recall that the times in the simulation are approximate). If Car 1 has not exited and Car 4 enters, they will crash (why?). If Car 1 has exited, Car 3 and Car 4 will be able to enter the road, but since they enter from opposite directions, they will eventually crash. Finally, after 1200 seconds, Car 2 enters the road from the West and traveling twice as fast as Car 4. If Car 3 was not coming from the opposite direction, Car 2 would eventually reach Car 4 from the back and crash. You may wish to experiment with reducing the initial delay of Car 2, or increase the initial delay of Car 3, to cause Car 2 to crash with Car 4 before Car 3 crashes with Car 4.

Exercises

1) Modify the procedure ***driveRoad*** such that the following rules are obeyed:

A) Avoid all collisions.

B) Multiple cars should be allowed on the road, as long as they are traveling in the same direction.

C) If a car arrives and there are already other cars traveling in the SAME DIRECTION, the arriving car should be allowed to enter as soon as it can. Two situations might prevent this car from entering immediately:

(1) there is a car immediately in front of it (going in the same direction), and if it enters it will crash (which would break rule A);
(2) one or more cars have arrived at the other end to travel in the opposite direction and are waiting for the current cars on the road to exit, which is covered by the next rule.

D) If a car arrives and there are already other cars traveling in the OPPOSITE DIRECTION, the arriving car must wait until all these other cars complete their course over the road and exit. It should only wait for the cars already on the road to exit; no new cars traveling in the same direction as the existing ones should be allowed to enter.

E) The previous rule implies that if there are multiple cars at each end waiting to enter the road, each side will take turns in allowing one car to enter and exit. However, if there are no cars waiting at one end, then as cars arrive at the other end, they should be allowed to enter the road immediately.

F) If the road is free (no cars), then any car attempting to enter should not be prevented from doing so.

G) All starvation must be avoided. For example, any car that is waiting must eventually be allowed to proceed.

This must be achieved ONLY by adding synchronization and making use of shared memory (as described in Exercise C). You should NOT modify the delays or speeds to solve the problem. In other words, the delays and speeds are givens, and your goal is to enforce the above rules by making use of only semaphores and shared memory.

2) Devise different tests (using different numbers of cars, speeds directions) to see whether your improved implementation of driveRoad obeys the rules above.

IMPLEMENTATION GUIDELINES

1) Avoid busy waiting. In class one of the reasons given for using semaphores was to avoid busy waiting in user code and limit it to minimal use in certain parts of the kernel. This is because busy waiting uses up CPU time, but a blocked process does not. You have semaphores available to implement the driveRoad function, so you should not use busy waiting anywhere.

2) Prevent race conditions. One reason for using semaphores is to enforce mutual exclusion on critical sections to avoid race conditions. You will be using shared memory in your driveRoad implementation. Identify the places in your code where there may be race conditions (the result of a computation on shared memory depends on the order that processes execute). Prevent these race conditions from occurring by using semaphores.

3) Implement semaphores fully and robustly. It is possible for your driveRoad function to work with an incorrect implementation of semaphores, because controlling cars does not exercise every use of semaphores. You will be penalized if your semaphores are not correctly implemented, even if your driveRoad works.

4) Control cars with semaphores: Semaphores should be the basic mechanism for enforcing the rules on driving cars. You should not force cars to delay in other ways inside driveRoad such as by calling the Delay function or changing the speed of a car. (You can leave in the delay that is already there that represents the car's speed, just don't add any additional delay). Also, you should not be making decisions on what cars do using calculations based on car speed (since there are a number of unpredictable factors that can affect the
actual cars' progress).

GRADING INFORMATION

1) Semaphores: We will run a number of programs that test your semaphores directly (without using cars at all). For example: enforcing mutual exclusion, testing robustness of your list of waiting processes, calling signal and wait many times to make sure the semaphore state is consistent, etc.

2) Cars: We will run some tests of cars arriving in different ways to make sure that you correctly enforce all the rules for cars given in the assignment. We will use a correct semaphore implementation for these tests so that even if your semaphores are not correct you could still get full credit on the driving part of the grade (you may assume that in our implementation, semaphore unblocking is FIFO, i.e., for each semaphore, the order in which processes are unblocked is the same order in which those same processes blocked).

Think about how your driveRoad might handle different situations and write your own tests with cars arriving in different ways to make sure you handle all cases correctly.

WHAT TO TURN IN

You must turn in two files: **mycode3.c** and **pa3d.c**. **mycode3.c** should contain you implementation of semaphores, and pa3d.c should contain your modified version of ***InitRoad*** and ***driveRoad*** (Main will be ignored). Note that you may set up your static shared memory struct and other functions as you wish. They should be accessed via ***InitRoad*** and ***driveRoad*** as those are the functions that we will call to test your code.

Your programs will be tested with various Main programs that will exercise your semaphore implementation, AND different numbers of cars, directions and speeds, to exercise your driveRoad function. Our Main programs will first call ***InitRoad*** before calling ***driveRoad***. Make sure you do as much rigorous testing yourself to be sure your implementations are robust.
Good luck!

```
#include <stdio.h>
#include "aux.h"
#include "sys.h"
#include "umix.h"

void InitRoad(void);
void driveRoad(int from, int mph);

void Main()
{
     InitRoad();

     /The following code is specific to this simulation, e.g.,  number
      of cars,  directions and speeds. You should experiment with
      different numbers of cars,  directions and speeds to test your
      modification of driveRoad.  When your solution is tested, we
      will use different Main procedures,  which will first call
      InitRoad before any calls to driveRoad. So, you should do
      any initializations in InitRoad.


     if (Fork() == 0) {
          Delay(1200);              // car 2
          driveRoad(WEST, 60);
          Exit();
```

```
    }

    if (Fork() == 0) {
        Delay(900);              // car 3
        driveRoad(EAST, 50);
        Exit();
    }

    if (Fork() == 0) {
        Delay(900);              // car 4
        driveRoad(WEST, 30);
        Exit();
    }

    driveRoad(EAST, 40);               // car 1

    Exit();
}
```

Our tests will call your versions of InitRoad and driveRoad, so your solution to the car simulation should be limited to modifying the code below. This is in addition to your implementation of semaphores contained in mycode3.c.

```
void InitRoad()
{
    /do any initializations here
}

#define IPOS(FROM)  (((FROM) == WEST) ? 1 : NUMPOS)

void driveRoad(int from, int mph)
    // from: coming from which direction
    // mph: speed of car
{
    int p;

    EnterRoad(from);

    for (p = 1; p <= NUMPOS; p++) {

        Delay(3600/mph);         // MUST NOT MODIFY!
```

```
            ProceedRoad();
        }
}
```