COSC 4330-Operating System Fundamentals
Assignment #3: The Bridge
(Due on Monday, May 2, 2011 at 11:59 pm)

OBJECTIVES

This rather short assignment should teach you how to use (a) POSIX semaphores to provide mutual exclusion and (b) UNIX shared memory segments to store shared variables.

THE PROBLEM

A bridge is load-zoned and can only carry a total weight of **`maxweight`** tons. Access to the bridge will be controlled by two functions:

- **`enter_bridge(weight)`** and
- **`leave_bridge(weight)`**,

where **`weight`** is the total loaded weight of each vehicle rounded up to an integer number of tons.

You are to implement these two functions using **`P()`** and **`V()`**system calls to UNIX semaphores and a shared memory segment representing the total weight of the vehicles that are already on the bridge.

Your two functions should:

1. ensure that the total weight of all vehicles on the bridge will never exceed **`maxweight`** tons,
2. ensure that all vehicle crossing the bridge will cross it in strict FCFS order, and
3. avoid deadlocks.

YOUR PROGRAM

Your program should consist of one main program, the two functions **`enter_bridge(weight)`** and **`leave_bridge(weight)`**, and one child process per vehicle arriving to the bridge.

The **`maxweight`** constant should be read from the command line as in:

```
./a.out 10
```

All other parameters will be read from the standard input. Each input line will describe one vehicle arriving at the bridge and will contain four parameters:

1. An alphanumeric string representing the car license plate,
2. A positive integer representing the number of seconds elapsed since the arrival of the previous vehicle (it will be equal to zero for the first vehicle arriving to the bridge);
3. A positive integer representing the total loaded weight of the vehicle rounded up to an integer number of tons,
4. A positive integer representing the number of seconds the vehicle will take to cross the bridge.

One possible set of input could be:

```
HIOFCR   0   1  10
STOL3N   3  10  20
10URED   8   1  30
2DIE4    7   1   5
BYOFCR   2   1  15
```

Your main program should read the input line per line, wait for the appropriate amount of time and fork a different child process for each incoming vehicle. It should print out a descriptive message including the vehicle license number ***and the current bridge load*** every time a vehicle:

1. Arrives at the bridge,
2. Starts crossing the bridge, and
3. Leaves the bridge.

Vehicles whose total loaded weight exceeds the maximum load of the bridge should be rejected and a descriptive message containing the serial number of the vehicle printed out.

The current bridge load when a vehicle starts crossing the bridge includes the weight of that vehicle but the current bridge load after a vehicle leaves the bridge does not.

POSIX SEMAPHORES

1. Don't forget the following includes:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <stdio.h>
#include <semaphore.h>
#include <sys/stat.h>
#include <fcntl.h>
```

2. To create a POSIX semaphore, or to connect to an existing one, use:

```
sem_t *mysem;
char name[] = "/Sem Name";
unsigned int initial_value;
mysem = sem_open(name, O_CREAT,
0600, initial_value);
```

   Note that **sem_open** does not reset the initial value of a semaphore that already exists. To check whether a semaphore exists, you can do:

```
$ ls /dev/shmem
```

   You can delete your semaphores by "removing" the pseudo-files named after them:

```
$ rm –i /dev/shm/sem.Sem\ Name
```

3. To do a **P()** operation on a semaphore, use:

```
sem_t *mysem;
sem_wait(mysem);
```

4. To do a **V()** operation on a semaphore, use:

```
sem_t *mysem;
sem_post(mysem);
```

5. To end the connection to an open semaphore and cause it to be removed when the last process closes it, use:

```
sem_t *mysem;
char name[] = "/Sem Name";
sem_close(mysem);
sem_unlink(name);
```

6. To check the value of a semaphore

```
sem_t *mysem;
sem_getvalue(mysem, &value);
```

7. ***Programs using POSIX semaphores on*** **bayou.cs.uh.edu** ***must be compiled with the library flag*** **–lrt** ***after the list of source code modules as in***

```
gcc bridge.c –lrt
```

UNIX SHARED MEMORY

1. A shared memory segment can be addressed by
   a) Its ***key***: an integer of type **key_t** (use a phone number),
   b) Its ***shared memory id***: an integer assigned by system.
2. Your program should have the above includes plus:

```
#include <sys/shm.h>
```

3. There are four primitive operations on shared memory segments:
   a) **int shmget (key_t key, int nbytes, int flags)**

      Gets **nbytes** bytes of shared memory and returns a shared memory id:

```
int shmid, nbytes;
shmid = shmget(key, nbytes,
0666 | IPC_CREAT);
```

      The flag **IPC_CREAT** requests the creation of the segment if it did not exist already.

   b) **char *shmat (int shmid, int address, int flags)**

      Attaches the shared memory segment to an address space:

```
char *pmem;
pmem = shmat(shmid, 0, 0);
```

      To test for error, use

```
if (pmem == (char *)(-1))…
```

      The shared memory segment looks now like an array of characters created using **malloc();**

   c) **int shmdt(char *pmem)**

      Detaches the shared memory segment, which must be done ***before destroying it***.

   b) **int shmctl (int shmid, int cmd, int arg)**

      To destroy a shared memory segment use:

```
shmctl(shmid, 0, IPC_RMID);
```

HINTS

1. Create your semaphores and your shared memory segment in your main program before you fork any child process.
2. Be sure to terminate your vehicle processes as soon as they exit from the bridge. You might otherwise run out of processes.
3. ***Never*** terminate a child process with **exit(0)** as it resets **stdin**. Use instead **_exit(0)**. Do not forget to delete your semaphores and shared memory segments when you are done with them.

This document was updated last on Sunday, April 17, 2011