**Problem Set 2**

**Matt Kaiser**

1. **There is a one-lane east-west bridge in Hawaii such that when a car is on the bridge going eastbound, no westbound cars are allowed until the eastbound car has left the bridge. Similarly when a westbound car is on the bridge, no eastbound cars are allowed until the westbound car has left the bridge. To make matters more complicated, if an eastbound car arrives and sees another eastbound car already on the bridge, then that eastbound car will also proceed onto the bridge. This is true even if there is a westbound car already waiting to enter the bridge. Similarly, a westbound car can tailgate behind another westbound car already on the bridge even if an eastbound car was waiting. Deign a synchronization solution using only locks, semaphores and integer variables that achieves the following: allows all cars bound in a certain direction to continue crossing as long as there is at least one car still on the bridge that is bound in that direction, then toggles to allow all cars in the opposite direction to proceed in a similar manner. The solution need not be starvation-free.**

**//The following is pseudo ‘c’ code**

**lock Mutexlock;**

**semaphore Eastsemaphore, Westsemaphore;**

**int Ecar = 0;**

**int Wcar = 0;**

**semaphore bridge = 1;**

**Towardswest() {**

**p(Westsemaphore);**

**p(Mutexlock);**

**Wcar++;**

**if(Wcar == 1) {**

**p(bridge); // lock the east side**

**}**

**v(Mutexlock);**

**v(Westsemaphore); // code to cross the bridge follows**

**p(Mutexlock);**

**Wcar--;**

**if(!Wcar) {**

**v(bridge); // release east side**

**}**

**v(Mutexlock);**

**}**

**Towardseast() {**

**p(Eastsemaphore);**

**p(Mutexlock);**

**Ecar++;**

**if(Ecar == 1) {**

**p(bridge); // lock the west side**

**}**

**v(Mutexlock);**

**v(Eastsemaphore); // code to cross the bridge follows**

**v(Mutexlock);**

**Ecar--;**

**if(!Ecar) {**

**v(bridge); // release west side**

**}**

**v(Mutexlock);**

**}**

1. Suppose task T1 has code C1 that must execute before task T2's code C2.  Design a solution that enforces this ordering using only condition variables, locks, and integer variables.  Semaphores and monitors are not allowed.  
   **//The following like problem 1 is ‘C’ pseudo code**

**Lock Mutexlock;**

**condition y;**

**int T1end = 0;**

**//T1 code**

**Acquire(Mutexlock);**

**//time to execute code for y1**

**T1end = 1;**

**Release(Mutexlock);**

**y.signal();**

**//T1 is over**

**//T2 code**

**Acquire(Mutexlock);**

**While(!T1end){**

**Release(Mutexlock);**

**y.wait();**

**Acquire(Mutexlock);**

**}**

**Release(Mutexlock);**

**//End T2**

1. Explain why the solution to the 3rd Readers/Writers problem is starvation-free.

**The 3rd solution to the Readers/Writers problem states that no thread shall be allowed to starve. This is due to the readblock put in place which makes reading threads wait until the previous writing thread has finished. This prevents writer threads from being starved, as well no reader threads are being starved because due its multiple readers being able to access the critical section while there are no active writer threads.**

1. a. Is the swap() function below thread-safe or not?  Explain your reasoning.

int temp;  
   
void swap(int \*y, int \*z)  
{  
    int local;  
   
    local = temp;  
    temp = \*y;  
    \*y = \*z;  
    \*z = temp;  
    temp = local;

}

**This is unsafe because of its use global variables in a shared space. If a context switch was to occur, it could halt the code resulting in only half of the swap occurring. It could cause an incorrect return by temp if the halt occurred when running the line ‘\*y = \*z’.**

b.  Suppose the swap() function above is called by the interrupt service routine (ISR) below.  Assume that if swap() is interrupted during normal execution, the ISR below is called and calls swap() again, reentering swap() and executing in the context of the interrupted thread.  Is swap() reentrant?  Explain your reasoning.

void interrupt\_service\_routine()  
{  
    int a = 1, b = 2;  
    swap(&a, &b);  
}

**Swap() is reentrant, during execution there aren’t multiple programs sharing global data. Therefore this method being called will pose no risk of deadlock or improper execution, even after interrupting its’ called thread**

1. Suppose you are asked to design a server application consisting of two processes P1 and P2, such that (1) P2 is to sleep until woken up by P1, whereupon (2) P2 would take a 10 MB file from P1 and compress it. What forms of IPC would be best suited to implement these types of information sharing? Describe your solution.

**Memory sharing interprocess communication(IPC) would be the most optimal. Due to the condition of the large file that you wish to share with another Process, we can use P1 to read the data into main memory while we wake Process 2, passing the datas’ pointer to its memory’s location, and wait asynchronously for Process 2 to finish. Once Process 2 has woke they will synchronously access memory and compress data such that the Process requests. Once the compression has ended Process 2 will release its memory lock, let Process 1 know with an interrupt that the compression is over, and resume sleeping as it has completed Process 1&2(P1&P2).**

1. Suppose processes P0 and P1 share variable V1, and processes P1 and P2 share variable V2, while processes P0, P1 and P2 share V3.  Operations on V1 are limited to increment() and decrement().  Operations on V2 are limited to square() and squareroot().  Operations on V3 are limited to sin() and cos().  Design a monitor-based solution that synchronizes access to and manipulation of these variables between the three processes so that race conditions are eliminated.

**#include <math.h>**

**monitor MonitorOPS {**

**private int v1, v2, v3;**

**condition c1, c2, c3;**

**public function increment() { //uses c1, v1**

**c1.wait();**

**v1++;**

**c1.signal();**

**}**

**public function decrement() { //Uses c1, v1**

**c1.wait();**

**v1--;**

**c1.signal();**

**}**

**public function square() { //Uses c2, v2**

**c2.wait();**

**v2 \*= v2;**

**c2.signal();**

**}**

**public function squareroot() { //Use c2, v2**

**c2.wait();**

**v2 = sqrt(v2);**

**c2.signal();**

**}**

**public function sin() { //Use c3, v3**

**c3.wait();**

**v3 = sin(v3);**

**c3.signal();**

**}**

**public function cos() { //Use c3, v3**

**c3.wait();**

**v3 = cos(v3);**

**c3.signal();**

**}**

**}**