Operating Systems Extra Credit Problem Set

Carl Cortright

November 19, 2016

1 CAS mutex implementation

The goal of this exercise was to use the CAS instruction in the IA32 instruction set to implement mutual exclusion.

```
/*
* Mutex implemented using the CAS instrution
*/
typedef int mutex;

/*
* Wait before entering the critical zone
*/
int wait(mutex* m){
  while(!CAS(m, 0, 1)){ /* Wait for the mutex to be free */ }
}

/*
* Signal that another process can enter the critical zone
*/
int signal(mutex* m){
  // If the mutex is is use, set it to be not in use
  CAS(m, 1, 0);
}
```

2 Mutually Exclusive Linked List Operations

The trick here was to realize that although searchers only examine the linked list, they must be mutually exclusive with delete, otherwise the search method can return positive even though an entry in the list has been deleted. This means that all three methods must be mutually exclusive with the delete semaphore, or in my case del, and only the insert method must be mutually exclusive with itself.

```
/*
* Search, Insert and Delete operations
* on a linked list without race conditions
*/
semaphore ins;
semaphore del;

/*
* Mutually Exclusive search method
*/
int Search(int i){
  wait(&del);
  int result = search(i);
  signal(&del);
}

/*
* Mutually Exclusive insert method
*/
void Insert(int i){
  wait(&ins);
  wait(&del);
  insert(i);
  signal(&del);
  signal(&ins);
}

/*
* Mutually Exclusive delete method
*/
void Delete(int i){
  wait(&del);
  delete(i);
  signal(&del);
}
```

3 River Crossing

This problem was tricky because it was important to cover all of the possible combinations of how hackers and employees could arrive, and figure out which arrivals would trigger a boarding and rowing of the boat. There can be 3 possible combinations, 4 employees, 4 hackers, or 2 hackers and 2 employees.

The only time to board 4 employees or 4 hackers is when there are either 3 hackers or 3 employees waiting and the 4th arrives. There were two situations to board 2 hackers and 2 employees, when there are 2 hackers and 1 employee

waiting and another employee arrives or when there are 2 employees waiting and 1 hacker and another hacker arrives. These 3 scenarios are covered the if statements in EmployeeArrives() and HackerArrives(). In all other situations, the person who is arriving should wait until enough people arrive to safely board the rowboat.

In each case, I signaled the relevant hackers and employees that they should board the boat, called RowBoat(), and signaled the remaining people waiting to return. These threads that returned without having the boat row will need to call EmployeeArrives() or HackerArrives() again to restart the waiting process.

```
/*
* Monitor based solution to the rowboat problem
*/
monitor rowboat{
  int hackersWaiting = 0;
  int employeesWaiting = 0;
  int onboard = 0;

  // There can be at most 3 hackers or 3 employees waiting
  condition hackers[3];
  condition employees[3];
  condition rowing;

  /*
  * Method called when an employee arrives
  */
  void EmployeeArrives(){
    if(employeesWaiting == 3){
      /*
      * Have all 4 employees board the boat
      * This will happen in order, as the threads will
      * re-enter the entry queue.
      * https://www.cs.mtu.edu/~shene/NSF-3/e-Book/MONITOR/CV.html
      */
      employees[0].signal();
      employees[1].signal();
      employees[2].signal();
      onboard = 3;
      BoardBoat();
      RowBoat();
      // Release all of the threads that were waiting for Rowboat()
      rowing.signal();
      // Signal other threads to resume execution and reset boat
      hackers[0].signal();
      hackers[1].signal();
      hackers[2].signal();
```

```
    onboard = 0;
  } else if(hackersWaiting == 2 && employeesWaiting == 1) {
    // Board 2 hackers and 2 employees
    hackers[0].signal();
    hackers[1].signal();
    employees[0].signal();
    onboard = 3;
    BoardBoat();
    // Release all threads that were waiting for RowBoat()
    RowBoat();
    rowing.signal();
    // Realease all other waiting threads and reset boat
    hackers[2].signal();
    employees[1].signal();
    employees[2].signal()
    onboard = 0;
  } else {
    employeesWaiting++;
    employees[employeesWaiting - 1].wait();
    if(onboard != 3){
      BoardBoat();
      // Wait until RowBoat() has returned
      rowing.wait();
    }
    employeesWaiting--;
  }
}

/*
* Method called when a hacker arrives.
* This is just a mirror of EmployeeArrives() except for hackers
*/
void HackerArrives(){
  if(hackersWaiting == 3){
    hackers[0].signal();
    hackers[1].signal();
    hackers[2].signal();
    onboard = 3;
    BoardBoat();
    RowBoat();
    // Release all of the threads that were waiting for Rowboat()
    rowing.signal();
    // Signal other threads to resume execution and reset boat
    employees[0].signal();
    employees[1].signal();
    employees[2].signal();
```

```
        onboard = 0;
      } else if(hackersWaiting == 1 && employeesWaiting == 2) {
        // Board 2 hackers and 2 employees
        employees[0].signal();
        employees[1].signal();
        hackers[0].signal();
        onboard = 3;
        BoardBoat();
        // Release all threads that were waiting for RowBoat()
        RowBoat();
        rowing.signal();
        // Realease all other waiting threads and reset boat
        employees[2].signal();
        hackers[1].signal();
        hackers[2].signal()
        onboard = 0;
      } else {
        hackersWaiting++;
        hackers[employeesWaiting - 1].wait();
        if(onboard != 3){
          BoardBoat();
          // Wait until RowBoat() has returned
          rowing.wait();
        }
        hackersWaiting--;
      }
    }
  }
```

4 Process Synchronization

The trick here was to remember that when you signal a semaphore it increments the value regardless if the value is negative or not. By using this property I was able to signal the semaphore multiple times after a process crossed its synchronization point. Then, when one of the processes that needs to wait on that process calls wait(x) it will be able to proceed without sleeping.

```
/*
* Solution to the x1...5 synchronization problem
*/
include <stdio>

// Global semaphores and counters
semaphore x1, x2, x3, x3, x4, x5;
```

```
int main(){
  // Create 5 processes to play with
  int x[5];
  int pid;

  for(int i = 0; i < 5; i++){
    pid = fork();
    if(pid != 0){
      x[i] = pid;
      break;
    }
  }

  // Create artificial sync point for each process
  switch(pid){
    case x[0]:
      // Sync point for process 1
      // Let x4, x2, and x5 go
      signal(x1);
      signal(x1);
      signal(x1);
      break;
    case x[1]:
      wait(x1);
      // Sync point for process 2
      break;
    case x[2]:
      wait(x4);
      // Sync point for process 3
      signal(x3);
      signal(x3);
      break;
    case x[3]:
      wait(x1);
      wait(x3);
      // Sync point for process 4
      signal(x4);
      break;
    case x[4]:
      wait(x1);
      wait(x3);
      wait(x4);
      // Sync point for process 5
      break;
    default:
      // Parent process, wait for all other processes to finish
```

```
        wait(NULL);
        printf("Parent exiting...");
    }
}
```