ASSIGNMENT 3–MONITORS

CONCURRENT PROGRAMMING

TEAM: 4

DIMITRIS VOITSIDIS 03480

IORDANA GAISIDOU 03570

STAVROS STATHOUDAKIS 03512

Project Tasks Summary

3.1. Binary Semaphores Library

3.2. Primary Numbers Tester

3.3. Narrow Bridge

3.4. Train Ride

3.1. Binary Semaphores Library

MonitorsForBinarySemaphores.h

structs

```
struct sem_monitor_var_t {
    pthread_mutex_t mtx
    pthread_cond_t semQ
}
```

```
struct rw_monitor_var_t {
    pthread_mutex_t mtx
    pthread_cond_t readQ, writeQ

    int readers, readers_waiting
    int writers, writers_waiting
}
```

Globals

```
static rw_monitor_var_t rw_monitor
```

MonitorsForBinarySemaphores.c

1st monitor Functions

```
function start_read():
    lock(mutex)
        while writers > 0:
            readers_waiting++
            wait(readQ, mutex)
        readers++
        if readers_waiting > 0:
            signal(readQ)
    unlock(mutex)
```

```
function stop_read():
    lock(mutex)
        readers--
        if readers == 0 and writers_waiting > 0:
            signal(writeQ)
    release(mutex)
```

```
function start_write():
    lock(mutex)
        while readers > 0 or writers > 0:
            writers_waiting++
            wait(writeQ, mutex)
        writers++
    unlock(mutex)
```

```
function stop_write():
    lock(mutex)
        writers--
        if readers_waiting > 0:
            signal(readQ)
        else if writers_waiting > 0:
            signal(writeQ)
    unlock(mutex)
```

3.1. Binary Semaphores Library

MonitorsForBinarySemaphores.c

```
function initialize_semaphore_monitor(semaphore_monitor):
    initialize mutex for semaphore_monitor
    initialize condition variable for semaphore_monitor
```

```
function decrement_semaphore(semaphore_monitor, semaphore_value):
    lock(mutex)
        while semaphore_value == 0:
            wait(semaphore_monitor.condition_variable, mutex)
        semaphore_value = 0
    unlock(mutex)
```

```
function increment_semaphore(semaphore_monitor, semaphore_value):
    lock(mutex)
        if semaphore_value == 1:
            unlock(mutex)
            return 1 // Semaphore already at maximum value
        else:
            semaphore_value = 1
            signal(semaphore_monitor.condition_variable)
            unlock(mutex)
            return 0 // Semaphore incremented successfully
```

3.1. Binary Semaphores Library

binarySemaphores.h

structs

```
struct mysem_t {
    sem_monitor_var_t sem_monitor
    int sem_id, sem_val
}
```

binarySemaphores.c

globals

static mysem_t sem_arr**

-> array of pointers to the created semaphores

static int sem_arr_size

-> size of the previous array

functions

function mysem_destroy(semaphore s):

start_read_lock

for each semaphore in sem_arr:

    if semaphore.id equals s.id:

        exists = true : break

if not exists:

    stop_read_lock

    return -1 // **ERROR** if semaphore not initialized

stop_read_lock

start_write_lock

  remove s from sem_arr

  resize sem_arr

stop_write_lock

return 1 // **SUCCESS**

3.1. Binary Semaphores Library

binarySemaphores.c functions

```
function mysem_init(semaphore s, integer n):
 if n is not 0 or 1: return 0

 start_read_lock

 for each semaphore in sem_arr:
 if semaphore.id equals s.id:
    stop_read_lock
    return -1 // ERROR if semaphore already initialized

stop_read_lock
```

```
s.id = id_assign
id_assign = id_assign + 1
s.value = n

initialize s.monitor

start_write_lock

allocate memory for s in sem_arr

add s to sem_arr
increment sem_arr_size

stop_write_lock

return 1 // SUCCES
```

3.1. Binary Semaphores Library

binarySemaphores.c functions

```
function mysem_down(semaphore s):
    start_read_lock

    for each semaphore in sem_arr:
        if semaphore.id equals s.id:
            exists = true : break

    if not exists:
        stop_read_lock
        return -1  // ERROR if semaphore not initiliazed

    stop_read_lock

    semaphore_down(s)  // Atomically decrement semaphore value

    return 1  // SUCCESS
```

```
function mysem_up(semaphore s):

start_read_lock

for each semaphore in sem_arr:
  if semaphore.id equals s.id:
      exists = true : break

if not exists:
    stop_read_lock
    return -1 // ERROR if semaphore not initiliazed

stop_read_lock

res = semaphore_up(s)

if res equals 1:
    return 0 // ERROR if semaphore already

return 1 // SUCCESS
```

3.2. Primality Numbers Tester

primalityTesterWithMonitors.h

structs

```
struct monitor_var_t {
    pthread_mutex_t mtx
    pthread_cond_t mainQ
    pthread_cond_t workerQ
}
```

```
struct workerArgS {
    int *job
    bool *job_taken
    bool *terminate
    monitor_var_t *monitor
}
```

primalityTesterWithMonitors.c

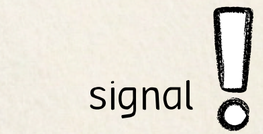
```
main()
  if number of arguments < 2
      print error message
      exit

  initialize monitor and worker arguments
  get num of workers from command line
  create worker threads

  for each num in command line arguments
      lock monitor.mutex
      if (job_taken == false) {
            wait for worker to take job (wait on mainQ)
      }
      assign number to job
      set job_taken to false

      signal worker to take job (signal workerQ)
      unlock monitor.mutex
```

monitor function :
assigning job



3.2. Primality Numbers Tester

primalityTesterWithMonitors.c

rest of main...

lock monitor.mutex
    if (job_taken == false) {
        wait for the last job to be taken(**wait on mainQ**)
    }
    terminate = true;
    signal workers terminate (**signal workerQ**)
unlock monitor.mutex

wait for workers' threads to terminate
free memory
destroy monitor
return

monitor function : terminate

worker()
    get worker arguments
    lock monitor.mutex
    while (job_taken) == true && (terminate) == false) {
        worker wait till assigned a job or terminated (**wait workerQ**)
    }

    if((terminate) == true){
        signal the rest of workers (**signal workerQ**)
        unlock monitor.mutex
        return NULL;
    }

    num = job
    job_taken = true;
    signal the main that you've taken the job (**signal mainQ**)
    unlock monitor.mutex

    if number <= 1 : print "not prime"
    else : check for divisibility
            print "prime" or "not prime"

monitor function : get the job/terminate

3.3. Narrow Bridge

bridge.h structs

```
struct monitor_var_t {
 pthread_mutex_t mtx;
 pthread_cond_t redQ;
 pthread_cond_t blueQ;
}
```

```
struct carArgS{
   int maxCarsCrossing;
   int maxCarsCrossed;
   int *carsCrossing;
   int *carsCrossed;
   int *waitingUs;
   int *waitingOthers;
   carColorE color;
   carColorE *color_bridge;
   monitor_var_t *monitor;
}
```

bridge.c

car(Cargs):

lock monitor.mtx

while true:

If bridge is empty **and** car's color is different from current bridge color)

color_bridge = car.color

break

if ((bridge is not the same color as car **or** bridge is full **or**

max cars crossed) **and** other color cars are waiting):

wait on your color's queue (**wait monitor.(yourcolorQ)**)

if cars from your color are waiting > 0:

signal a car in your color's queue (**wait monitor.(yourcolorQ)**)

unlock monitor.mtx

CAR STARTS CROSSING THE BRIDGE

monitor function : check if you can cross the bridge

3.3. Train Ride

bridge.h

CAR ENDS CROSSING THE BRIDGE

monitor function :
let the next “right” car
to cross the bridge

lock monitor.mtx

If bridge is empty **and** there are waiting cars of the other color:
signal a car in other color’s queue (**signal monitor.(othercolorQ)**)

If more cars of the same color can cross **or** there are no other color cars waiting:
signal a car in your color’s queue (**signal monitor.(yourcolorQ)**)

unlock monitor.mtx

3.4. Train Ride

train.h structs

```
struct monitor_val_t{
    pthread_mutex_t mtx;
    pthread_cond_t trainQ;
    pthread_cond_t disembarkQ;
    pthread_cond_t boardingQ;
}
```

```
struct TargsS {
    int train_capacity;
    bool* destroy;
   // Flag to indicate if the train is traveling
    travelE* travel;
    int* boarded_passengers;
    int* waiting_passengers;
    monitor_var_t* monitor;
}
```

train.c

monitor function : train traveling

```
train(Targs)
    lock monitor.mtx
    while not destroy {
        if ( (train not full or travel = TRAVELED) and not destroy)
            wait monitor.trainQ : wait till a passenger wakes you
        if destroy and boarded_passengers = 0 : break

        travel ← TRAVELING
        TRAVELING
        travel ← TRAVELED

        signal monitor.disembarkQ -> signals the boarded passengers to disembark
    }
    if waiting_passengers != 0
        signal monitor.boardingQ ->signals waiting passengers not board and terminate

    while boarded_passengers != 0
        wait monitor.trainQ -> wait till all boarded passengers disembark
    unlock monitor.mtx
```

3.4. Train Ride

passenger(Pargs)

lock monitor.mtx

while (travel != WAITING **or** boarded_passengers = train_capacity)
and not destroy –> Wait to board if necessary
    wait monitor.boardingQ

if destroy
    signal monitor.boardingQ –> passenger didnt even board
    unlock monitor.mtx
    return

if boarded_passengers = train_capacity
    signal monitor.trainQ –> signal the train up to do the ride
else
    signal monitor.boardingQ –> signal other passenger to board

monitor function :
passenger traveling

if travel != TRAVELED
    wait monitor.disembarkQ –> wait to disembark

if destroy **and** boarded_passengers = 0
    signal monitor.trainQ –> the last one signals the train to terminate

if **not** destroy **and** boarded_passengers = 0
    travel ← WAITING
    signal monitor.boardingQ –> board for the next ride
else
    signal monitor.disembarkQ –> disembark others

unlock monitor.mtx

3.4. Train Ride

```
procedure main(argc, argv)
  create threads array for passengers, for the train
  initializes the arguments for both

  while true
    if user presses ENTER
      create a new passenger thread
    else
      lock monitor(mtx)
      set destroy to true
      signal train to terminate (using monitor.trainQ)
      unlock monitor(mtx)
      break

  wait for all threads to terminate
  destroy monitor
  free passengers' threads memory
  return 0
```


Thank you
Merry Christmas!