CONCURRENT PROGRAMMING

ASSIGNMENT 2 - SEMAPHORES

Dimitris Voitsidis 03480
Iordana Gaisidou 03570
Stavros Stathoudakis 03512

Project Tasks Summary

2.1. BINARY SEMAPHORES LIBRARY

2.2. PRIMARY NUMBERS TESTER

2.3. NARROW BRIDGE

2.4. TRAIN RIDE

2.1. Binary Semaphores Library

a) binarySemaphores.c

a) **mysem_init** (mysem_t, int n)

b) **mysem_down** (mysem_t *s)

c) **mysem_up** (mysem_t *s)

d) **mysem_destroy** (mysem_t *s)

b) mysem_t

```
struct mysem_t {
    int semid;}
```

c) global variables

static mysem_t **sem_arr -> Array of pointers to semaphores that have been created.

static int sem_arr_size -> size of the array of the previous array.

static int rds -> keeping track of readers

d) global mutexes

static pthread_mutex_t rd_mutex

static pthread_mutex_t wd_mutex

Algorithm 1 mysem_init

Require: semaphore s, integer n

Ensure: 1 if successful, 0 if invalid n, -1 if already initialized

```
if n ≠ 0 and n ≠ 1 then
    return 0
end if

lock rd_mutex
if reader_count = 0 then
  lock wd_mutex
end if
increment reader_count
unlock rd_mutex

for each semaphore in semaphore_array do
      if semaphore = s then
    lock rd_mutex
    decrement reader_count
    if reader_count = 0 then
            unlock wd_mutex
    end if
    unlock rd_mutex
    return -1
  end if
end for

lock rd_mutex
decrement reader_count
if reader_count = 0 then
    unlock wd_mutex
end if
unlock rd_mutex

create semaphore s with initial value n

lock wd_mutex
append s to semaphore_array
unlock wd_mutex
return 1
```

Checking if n is not equal to 0 or 1

Checking if semaphore is already initialized

semget with IPC_PRIVATE flag for a unique id
semctl with SETVAL flag for initializing to n

Put the semaphore in the array

2.1. Binary Semaphores Library

Algorithm 2 mysem_down

Require: semaphore s
Ensure: 1 if successful, -1 if semaphore is not initialized

```
lock rd_mutex
if reader_count = 0 then
  lock wd_mutex
end if
increment reader_count
unlock rd_mutex

for each semaphore in semaphore_array do
  if semaphore = s then
    set found to true
    break
  end if
end for

if not found then
  lock rd_mutex
  decrement reader_count
  if reader_count = 0 then
    unlock wd_mutex
  end if
  unlock rd_mutex
  return -1
end if

lock rd_mutex
decrement reader_count
if reader_count = 0 then
  unlock wd_mutex
end if
unlock rd_mutex

decrement semaphore s
return 1
```

Checking if semaphore is already initialized

Return -1 if semaphore not initialized

decrease (-1) using **semop**

2.1. Binary Semaphores Library

Algorithm 3 mysem_up

Require: semaphore s

Ensure: 1 if successful, 0 if the semaphore is already at its maximum value, -1 if the semaphore is not initialized

```
lock rd_mutex
if reader_count = 0 then
  lock wd_mutex
end if
increment reader_count
unlock rd_mutex

for each semaphore in semaphore_array do
  if semaphore = s then
    set found to true
    break
  end if
end for

if not found then
  lock rd_mutex
  decrement reader_count
  if reader_count = 0 then
    unlock wd_mutex
  end if
  unlock rd_mutex
  return -1
end if

lock rd_mutex
decrement reader_count
if reader_count = 0 then
  unlock wd_mutex
end if
unlock rd_mutex

if semaphore value = 1 then
  return 0
end if

increment semaphore s

return 1
```

Checking if semaphore is already initialized

Return -1 if semaphore not initialized

GETVAL using **semctl**
If == 1 then return 0 because we have two up's in a row

increase using **semop**

2.1. Binary Semaphores Library

Algorithm 4 mysem_destroy

Require: semaphore s
Ensure: 1 if successful, -1 if the semaphore is not initialized

```
lock rd_mutex
if reader_count = 0 then
  lock wd_mutex
end if
increment reader_count
unlock rd_mutex

for each semaphore in semaphore_array do
  if semaphore = s then
    set found to true
    break
  end if
end for

if not found then
  lock rd_mutex
  decrement reader_count
  if reader_count = 0 then
    unlock wd_mutex
  end if
  unlock rd_mutex
  return -1
end if

lock rd_mutex
decrement reader_count
if reader_count = 0 then
  unlock wd_mutex
end if
unlock rd_mutex

lock wd_mutex
remove s from semaphore_array
unlock wd_mutex

destroy semaphore s

return 1
```

Checking if semaphore is already initialized

Return -1 if semaphore not initialized

Remove the semaphore from the array and realloc it

semctl using **IPC_RMID flag**

2.2. Primary Numbers Tester

```
struct workerArg {
        int *job
        bool *terminate
        mysem_t *semWorker
        mysem_t *semMain }
```

void *worker(void *Wargs)

```
while (1):
      mysem_down(semWorker)

      if terminate
         mysem_up(semWorker)
         print terminate message and terminate

      get the job (number to check)

      mysem_up(semMain)

      if num <= 1
         print number is not equal
      else
         use square root algorithm to check if the  number is prime
         print result (prime or not prime)
```

Worker x took the job offered by Main

Wake the rest workers to terminate

Inform main that I took the job to let her give the rest. (wake Main)

Wake workers to let them terminate

Main Function:

Get number of workers from command line as workersNo

Initialize semaphores semMain to 1 and semWorker to 0

Assign the following to worker_args:
- job = address of num
- semWorker = address of semWorker
- semMain = address of semMain
- terminate = address of terminate

Allocate memory for worker threads array t

For each worker:
- Create a worker thread and pass worker_args

For each job:
- **mysem_down(semMain)**
- Assign job number from command line to num
- **mysem_up(semWorker)**

mysem_down(semMain)

Set terminate flag to True
mysem_up(semWorker)

For each worker:
- Wait for worker thread to finish using pthread_join

Destroy semaphores semMain and semWorker

Sleep till a worker take the job main offered

Wake workers to take the job main offers

Main should sleep till the last job is taken by a worker

2.3. Narrow Bridge

```
struct carArg {
   int maxCarsCrossing; // Maximum number of cars of the same color that can cross the bridge consecutively
   int maxCarsCrossed;  // Maximum number of cars allowed on the bridge at the same time

   int *carsCrossing;      // Current number of cars crossing the bridge
   int *carsCrossed;       // Current number of cars (same colored) that have passed the cross consecutively

   int *waitingUs;         // Current number of cars of my color waiting to cross the bridge
   int *waitingOthers;     // Current number of cars of the other color waiting to cross the bridge

   carColorE color;        // My color

   carColorE *color_bridge;   // Current color of cars crossing the bridge

   mysem_t *semUs;
   mysem_t *semOthers;
   mysem_t *mtx;
}
```

void *car(void *Cargs):

```
mysem_down(mtx)
   increase waiting cars
mysem_up(mtx)

while (1) {
      mysem_down(mtx)
      if (bridge is empty && bridge color != color) {
         change bridge color
         reset cars_crossed (0)
      }

      if (bridge color != color || bridge is full ||
         (max cars crossed = cars_crossed && other cars waiting > 0)) {
         mysem_up(mtx)
         mysem_down(semUs)
         continue
      }

      decrease waiting cars
      increase crossing cars
      increase crossed cars

      if (same color cars are waiting) {
         mysem_up(semUs)
      }

      mysem_up(mtx)
      break
   }

   simulate passing bridge time using sleep

   mysem_down(mtx)

   Decrease cars crossing
```

Sleep

If a car of our team is waiting to cross the bridge , we wake it

```
if (bridge is empty && other cars waiting > 0) {
      mysem_up(semOthers)
}


if ((cars_crossed < max_cars_crossed || other cars waiting = 0)
    && same color cars waiting > 0) {
      mysem_up(semUs)
   }

   mysem_up(mtx)
}
```

If a car of the other team is waiting and the bridge is empty, we wake it

Wake us

Main Function:

```
initialize_semaphores (semBlue, semRed, mtx) to 1

while (1) {
   read_user_input(carColor)

   if (carColor == 'R' || carColor == 'r') {
      create_red_car_thread
   } else if (carColor == 'B' || carColor == 'b') {
      create_blue_car_thread
   } else {
      break
   }
}

wait for threads to join using pthread_join

destroy_semaphores(semBlue, semRed, mtx)
```

2.4. Train Ride

```
struct Targs {
    int train_capacity; // Train capacity

    bool* destroy;      // Flag to terminate train

    int* boarded_passengers;  // Number of passengers boarded
    int* waiting_passengers;    // Number of passengers waiting

    mysem_t* mtx; // Semaphore to signal waiting passengers
    mysem_t* semBoarding; // Semaphore to signal boarding passengers
    mysem_t* semTrain; // Semaphore to signal train
    mysem_t* semDisembark; // Semaphore to signal disembarking passengers
}
```

2.4. Train Ride

void *train(void *Targs)

```
print_train_start_message()

   while (destroy_flag == false) {
      mysem_down(semtrain)

      if (destroy_flag = true && passengers_boarded = 0) {
         break
      }

      sleep()

      mysem_up(semdisembark)
   }

   mysem_up(mtx)
   if (waiting_passengers != 0) {
      mysem_up(semBoarding)
   }
   mysem_up(mtx)

   while (boarded_passengers != 0) {
      mysem_down(semTrain)
   }

   return
}
```

If train is not full then sleeps

Disembark the boarded passengers

Tell the waiting passengers to leave

Tell the waiting passengers to leave

void *passenger(void *Pargs)

```
passenger(args) {
   mysem_up(mtx)
   increase waiting passengers
   mysem_down(mtx)

   mysem_down(semBoarding)

   mysem_up(mtx)
   decrease waiting passengers
   mysem_down(mtx)

   if (destroy_flag = true) {
      mysem_up(semboarding)
      return
   }

   increase boarded_passengers

   if (train is full) {
      mysem_up(semTrain)
   } else {
      mysem_up(semBoarding)
   }

   mysem_down(semDisembark)

   decrease boarded_passengers
```

Waits till someone tells it to board

Wakes up the next waiting passenger to destroy itself

If the train is full, wake it up to start its journey

else wakes the next passenger to board

Waits the train to complete its journey

2.4. Train Ride

...rest of passenger

```
if (destroy_flag = true && boarded_passengers = 0) {
      mysem_up(semTrain)
}

   if (train is empty && destroy_flag = false) {
      mysem_up(semBoarding)
   } else {
      mysem_down(semDisembark) // Signal other passengers
to disembark
   }
}
```

Wakes up train to terminate

Tell waiting passengers to board the train if its empty
Else
Tell the rest of the boarded passengers to disembark

Main Function:

```
main() {
   mysem_init(semTrain (0) , semBoarding(1) , semDisembark(0) , mtx(1))

   while (1) {
      read_user_input()

      if (user input = enter) {
         destroy_flag = true
         mysem_up(semTrain)
         break
      }

      create_passenger_thread(passengers, i, passenger_args)
   }

   join threads

   mysem_destroy(semTrain, semBoarding, semDisembark, mtx)
}
```

Wake up the train if the user wants to terminate it

Thank you