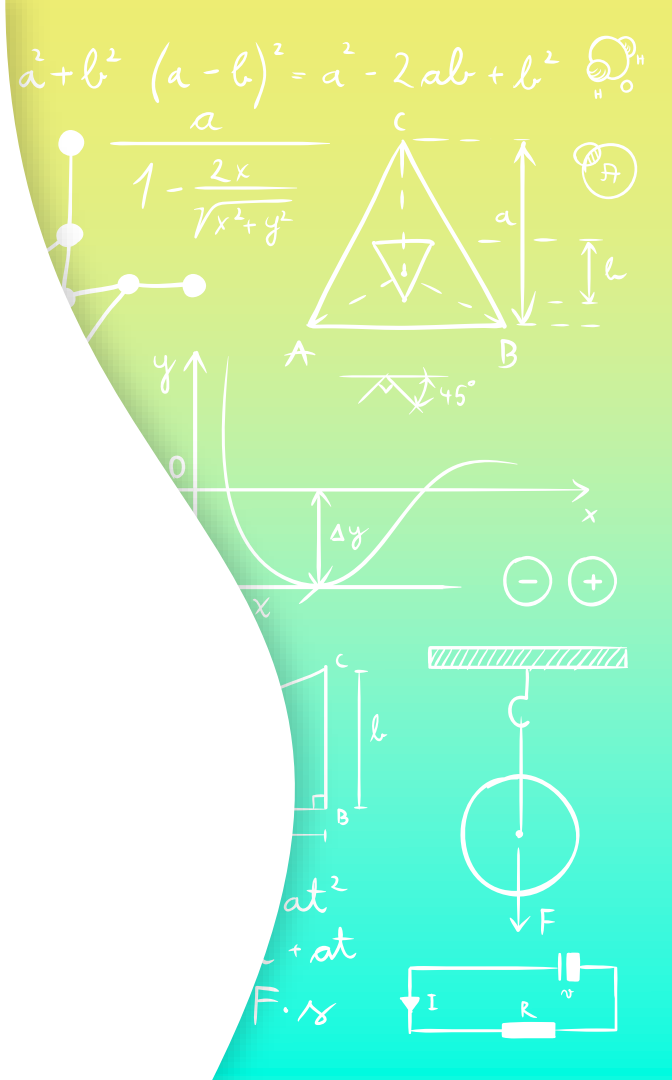
November 2024

University of Thessaly, Department of
Electrical and Computer Engineering



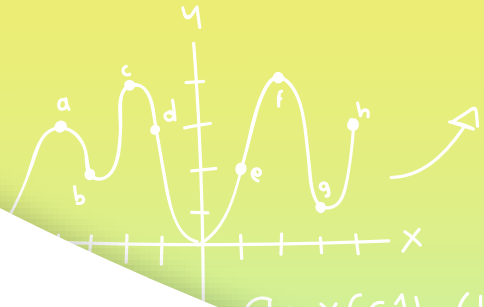
CONCURRENT PROGRAMMING ASSIGNMENT SERIES 2

TEAM 11

2.1

BINARY SEMAPHORES

A binary semaphores' library implementation in C.



ΠΑΝΕΠΙΣΤΗΜΙΟ
ΘΕΣΣΑΛΙΑΣ

MAIN IDEA

Implement a library of binary semaphores' operations using System's V semaphores.

The basic binary semaphores' operations are Initialize, Down, Up and Destroy.

System V semaphores are general. To transform them into binary semaphores we need to keep them atomic and restrict their value to either 0 or 1.

In order to have mutual exclusion in the library's implementation mutexes are used.

THE STUCTURE

```
typedef struct {
    int sem_id;
    int init;
    pthread_mutex_t mutex;
} mysem_t;
```

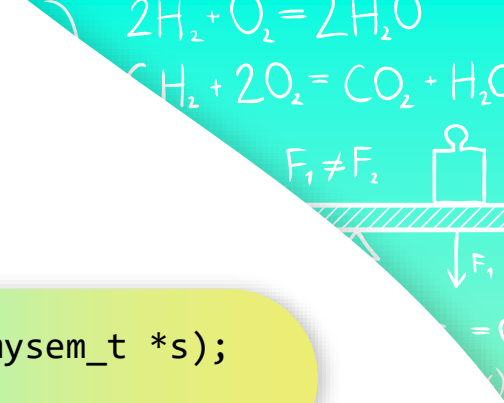
```
//Binary Semaphore

mysem_t
```

```
//Initialization flag

int init;
```

```
//Semaphore's Identifier

int sem_id;
```

```
//Semaphore's mutex

pthread_mutex_t mutex;
```

THE FUNCTIONS




```
int mysem_init(mysem_t *s, int n);
```

Initializes binary sem to n value.

```
int mysem_destroy(mysem_t *s);
```

Destroys a binary semaphore.

```
mysem_down(mysem_t *s);
```

Decreases bsem’s value by one.

```
int mysem_up(mysem_t *s);
```

Increases bsem’s value by one.

MYSEM_INIT(MYSEM_T *S, INT N);

```
{IF n not 0 or 1:
      RETURN 0

 INIT mutex
 LOCK mutex
 IF semaphore initialized:
      RETURN -1

 GENERATE a unique key for semaphore
 CREATE semaphore
 INITIALIZE semaphore to value n

 SET semaphore's init flag to 1
 UNLOCK mutex
 RETURN 1
}
```

- Check argument n
- Create a mutex
- Lock the mutex
- Check initialization
- Create a unique key for sem
- Create Semaphore
- Initialize semaphore
- Mutex unlock

MYSEM_DOWN(MYSEM_T *S);

```
{ LOCK mutex
  IF semaphore not initialized:
        RETURN -1

  SET semaphore operation (down by 1)
  UNLOCK mutex

  EXECUTE semaphore operation
  IF it fails:
      RETURN -1

  RETURN 1
}
```

- Lock the mutex
- Check initialization
- Initialize semaphore operation
- Mutex unlock
- Execute semaphore operation (down by 1)
- Error handling

MYSEM_UP(MYSEM_T *S);

```
{    LOCK mutex
     IF semaphore not initialized:
          RETURN -1

     CHECK semaphore's value:
          IF value is 1:
               RETURN 0

     SET semaphore operation (up by 1)
     EXECUTE semaphore operation
     IF it fails:
          RETURN -1

     UNLOCK mutex
     RETURN 1
}
```

- Lock the mutex
- Check initialization
- Check semaphore's value
- If it is equal to 1 return 0
- Else initialize semaphore operation
- Execute semaphore operation (up by 1)
- Mutex unlock

MYSEM_DESTROY(MYSEM_T *S);

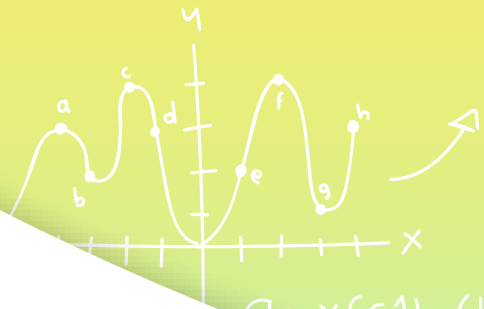
```
{ LOCK mutex

IF semaphore  not initialized:
      RETURN -1

 REMOVE semaphore
 IF that fails:
       RETURN -1

SET semaphore's init flag to 0

UNLOCK mutex
DETROY mutex

 RETURN 1
}
```

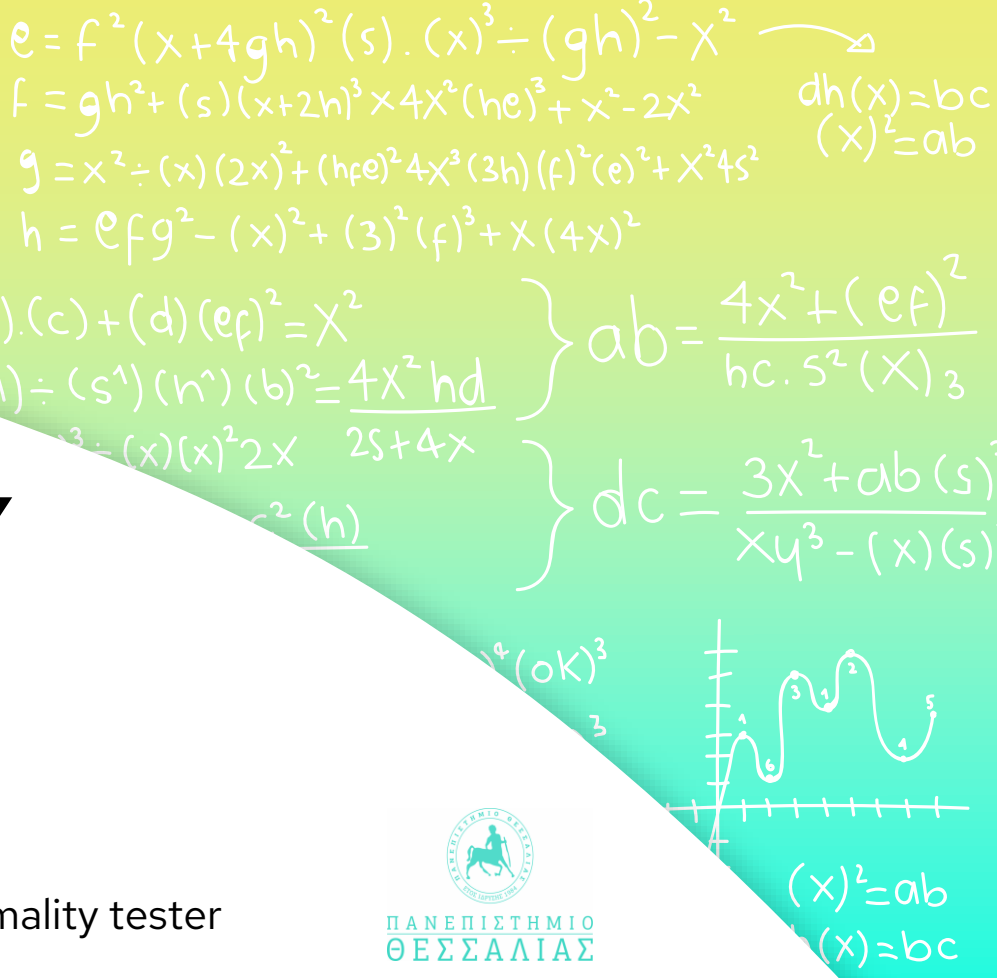
- Lock the mutex
- Check initialization
- Remove semaphore
- In case of failure return -1
- Set semaphores flag to 0
- Mutex unlock
- Mutex destroy

2.2

PRIMALITY TESTER

Implementation of a multi-thread primality tester





ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

MAIN IDEA

This program takes positive integers as input and checks if they are prime or not

Main assigns jobs to threads the so-called workers. Workers wait for a job to be assigned to them.

All the waiting and synchronization of the program is made using the binary semaphore library made in ex. 2.1

THE STUCTURE

```
typedef struct {
    pthread_t thread_id;
    mysem_t *s1, *s2;
    int number_to_check;
    int terminate;
} thread_infoT;
```

```
//Thread identifier
thread_id
```

```
//Binary Semaphores
*s1, *s2;
```

```
//Number to check primality
int number_to_check;
```

```
//Terminating flag
int terminate;
```

IS_PRIME

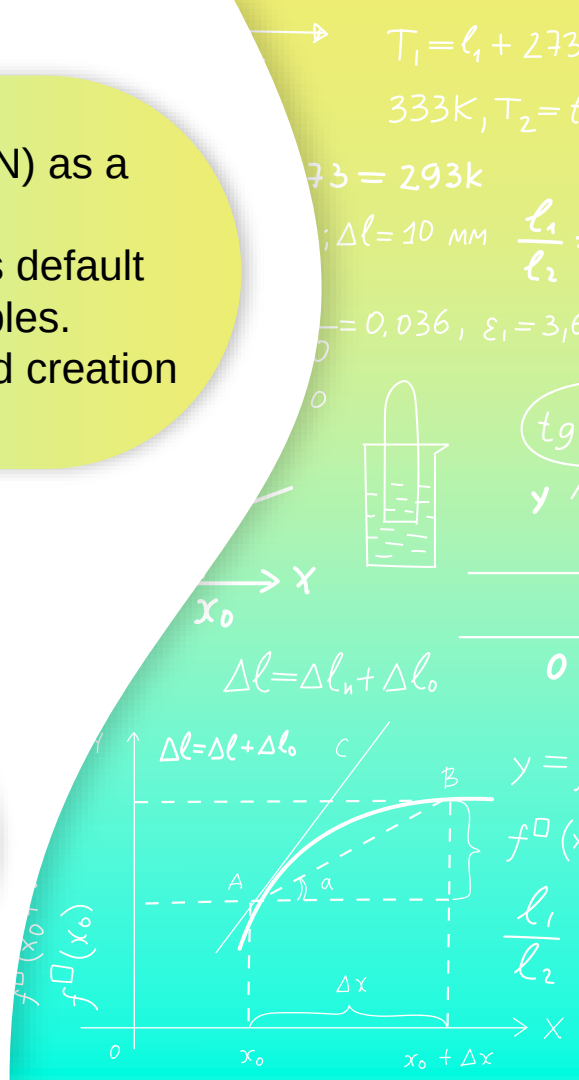
- Calculates if a number is a prime or not

MAIN

- Takes the number of threads (N) as a command-line argument.
- Initializes each thread and sets default values for thread control variables.
- Error handling in case of thread creation failure

WORKER

- Unblocks when main assigns a number to check.
- Checks if the assigned number is prime.
- Blocks again until the number to check gets updated.
- Terminates when requested by the main program.



Main function

```
SET j to 0
WHILE reading an integer into 'input' from
standard input is successful:
        FOR each thread 'j' from current value to N:

        DOWN semaphore for thread's 's1'
        ASSIGN 'input' to 'number_to_check' of
        thread 'j'
        UP semaphore for thread's 's2'

EXIT loop IF j >= N:
RESET j to 0
```

- Read numbers from input
- Down the thread’s semaphore
- Assign number for check
- Up thread’s second semaphore to complete the work
- Exit when Ctrl+D

Worker Function

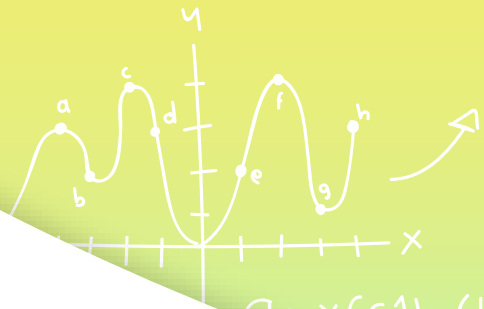
```
WHILE TRUE:
        DOWN semaphore for thread's 's2'
        IF 'terminate' flag of thread is TRUE:
                BREAK loop

        SET 'result' to result of checking if 'number_to_check'
        of thread is prime
        IF 'result' is TRUE:
                PRINT "<number_to_check> is Prime: <result>"
        ELSE:
                PRINT "<number_to_check> is Not Prime:
        <result>"


        UP semaphore for thread's 's1' RETURN NULL
```

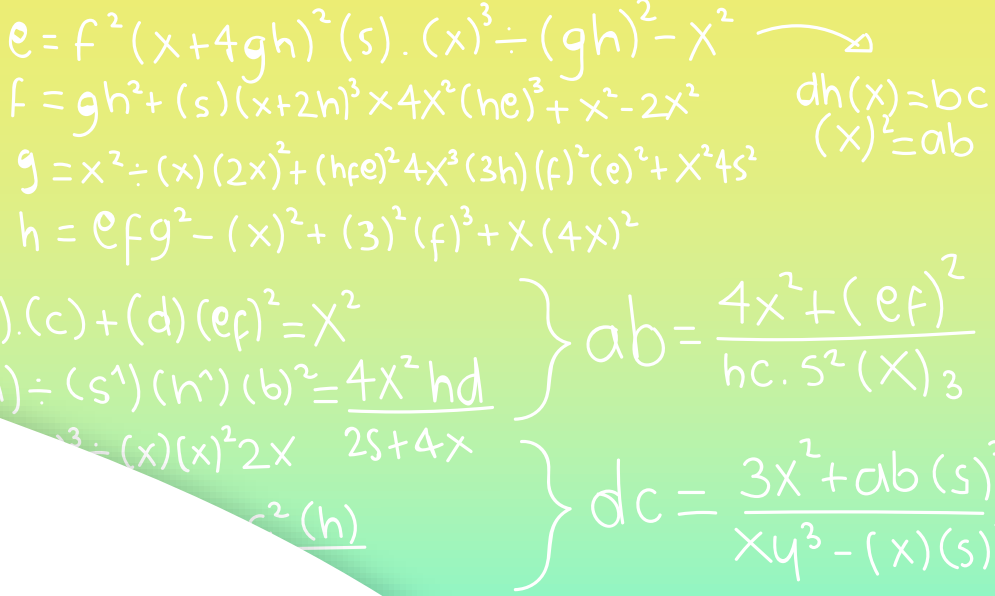
- Down semaphore to start the checking
- Check the number if is prime
- Print the result
- Up the other semaphore to indicate the end of the checking

2.3

NARROW BRIDGE PROBLEM

Control vehicle traffic on a narrow bridge.





ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

MAIN IDEA

Control vehicles coming by two opposite directions on a narrow bridge using binary semaphores.

The semaphores operate like traffic lights and they periodically allow either Red or Blue cars pass through the narrow bridge.

There are three rules to make things work better:
1. Vehicles cannot move in opposite directions in the bridge
2.No more than N vehicles can fit in the bridge at the same time.
3.There should not exist any vehicle that waits to get in the bridge forever.

THE STUCTURE

```
//Car's struct
typedef struct {
    int N, numRedCars, numBlueCars;   //Number of total, red and blue cars
    int i, passed, inBridge;          //Number of passed cars or cars in
the bridge
    color_t priority, flow;           //Priority's color and current
flow's color
    mysem_t **s, *semApply;           //Semaphores for synchronization
} carInfo_t;
```

```
//Thread's Struct
typedef struct {
    pthread_t threadId; //Thread_Id
    color_t color;            //Car's color
    int semIndex;      //Semaphore's Index
    int threadIndex;   //Thread's Index
} threadInfoT;
```

```
//Define car colors
typedef enum {
    BLUE = 0,
    RED = 1,
    NONE = 2
} color_t;
```

Enter bridge function

```
WHILE TRUE:
        DOWN semaphore for 'semApply'

IF priority is thread's color AND bridge is not in use:

            SET 'passed' to 0
            CALL 'wannaEnter' with 'thread'
            SET 'flow' to thread's color
            PRINT flow with color
            SET priority to NONE
            BREAK loop

ELSE IF priority is not NONE AND bridge is in use:
            UP semaphore for 'semApply'

ELSE IF passed count > N + 1:
            SET priority to opposite of current flow
            RESET passed count to 0
            UP semaphore for 'semApply'

ELSE IF flow matches thread's color:

            CALL 'wannaEnter' with 'thread'
            PRINT flow with color
            BREAK loop

ELSE:

            IF bridge is not in use:
                  RESET passed count to 0
                  CALL 'wannaEnter' with 'thread'
                  SET 'flow' to thread's color
                  PRINT flow with color
                  BREAK loop
             ELSE:
                  UP semaphore for 'semApply'

UP semaphore for 'semApply'

RETURN 0
```

IF CASES

- If thread’s side has prio
- if the other side has prio
- if too many cars has pass from the same color in the row
- if the flow is in thread’s color
- else (check if bridge is empty first)

Wanna Enter function

```
DOWN semaphore for 's[i]'

SET thread's 'semIndex' to current value of
'i'
INCREMENT 'inBridge' count
INCREMENT 'passed' count
INCREMENT 'i'
IF 'i' equals N:
    RESET 'i' to 0

RETURN 0
```

- Down the passing bridge semaphore
- Increase the “inBridge” count by one
- Increase the “passed_streak” count by one

Exit bridge function

UP semaphore for thread's semaphore index

DOWN semaphore for 'semApply'
DECREMENT 'inBridge' count
UP semaphore for 'semApply'

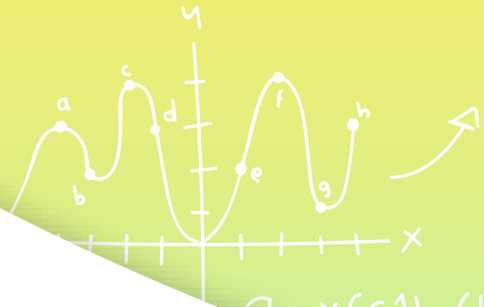
RETURN 0

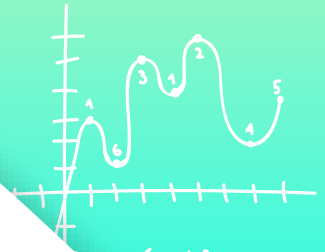
- Up the passing bridge semaphore
- Down the apply semaphore
- Decrease the inbridge count by one
- Up the apply semaphore

2.4

ROLLER COASTER

Synchronize a roller coaster and its passengers.





ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

MAIN IDEA

Synchronize a roller coaster ride in an amusement park using binary semaphores for thread management.

A roller coaster can hold up to N passengers.
The roller coaster only starts its ride when it is full, i.e., all seats are occupied. Once the roller coaster completes a ride, passengers get off the train before next N passengers board.

IMPORTANT SECTIONS:
- Passengers board only when there are empty seats.
- Roller coaster does not start until all seats are filled.
- Passengers don't board while a ride is ongoing or until all previous passengers have disembarked.

THE STUCTURE

```
//Passenger's struct
typedef struct {
    pthread_t *threadId;   //Thread's Id
} passenger_t;
```

```
//Train's struct
typedef struct {
    pthread_t *threadId;   //Thread's Identifier
    mysem_t *trainSem,    //Semaphore to block passengers
      *blockTrain,        //Semaphore to block the train
      *blockExit,                    //Semaphore to block Exit train
      *readyToExit,     //Semaphore to Exit train
      *endedTrip;                    //Semaphore to say that trip Ended
    int emptySeats,     //Number of empty seats on the train
      eof;                           //Flag to indicate End_Of_File
} train_t;
```

Train function

```
DO:
    PRINT "-Train Ready To Go-"
    DOWN semaphore for
'blockTrain'

    IF end-of-file (eof) flag is TRUE:
        UP semaphore for 'trainSem'
        IF 'sem_res' is 0:
            PRINT "lost call of up"
        ELSE IF 'sem_res' is -1:
            PRINT "error in up"

    ELSE:
        FOR each seat 'i' from 0 to
MAX_SEATS:
            UP semaphore for
'trainSem'
            IF 'sem_res' is 0:
                PRINT "lost call of up"
            ELSE IF 'sem_res' is -1:
                PRINT "error in up"

    PAUSE for 3 seconds

    PRINT "-Train Arrived-"

 IF end-of-file (eof) flag is TRUE:
        UP semaphore for
'blockExit'
        IF 'up_res' is 0:
            PRINT "lost call of up"
        ELSE IF 'up_res' is -1:
            PRINT "error in up"

    ELSE:
        FOR each seat 'i' from 0 to
MAX_SEATS:
            UP semaphore for
'blockExit'
            IF 'up_res' is 0:
                PRINT "lost call of up"
            ELSE IF 'up_res' is -1:
                PRINT "error in up"
            DOWN semaphore for
'readyToExit'

    PRINT "-Train Ended Trip-"
    UP semaphore for 'endedTrip'

WHILE end-of-file (eof) flag is
FALSE

RETURN NULL
```

- Down semaphore to start train
- Up semaphore to unblock passengers for entrance
- Pause for 3 seconds (trip time)
- Up semaphore to unblock passenger for exit
- All that while not last trip (ctrl+d)

Enter/Exit funtion

Enter:

DOWN semaphore for 'trainSem'
DECREMENT 'emptySeats' count

Exit:

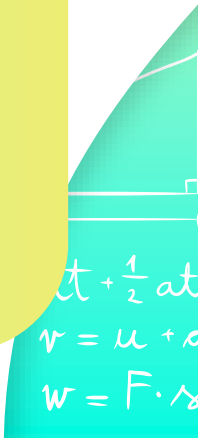
DOWN semaphore for 'blockExit'
INCREMENT 'emptySeats' count

Enter:

- Down semaphore to block from entering the train
- Decrease empty seats

Exit:

- Down semaphore to block exit
- Increase empty seats



Thanks!

Do you have any questions?

Evryviadis Liapis
Evaggelos Plytas
Aikaterini Tsiaousi