Synchronization

Exercises on semaphores and mutexes

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

License Information

Exercise 01

- Implement a C or C++ program that
  - Runs 1 thread TA and 1 thread TB
  - TA and TB include an infinite cycle in which they display one single character 'A' or 'B', respectively
  - Synchronize threads such that for each set of 3 characters there is 1 character A and 2 characters B in any position
  - Execution example

```
pgrm
    ABB
    BBA
    BAB
    etc.
```

Solution

```
#include <iostream>
#include <semaphore>
#include <thread>
#include <unistd.h>

using std::cout;
using std::endl;

std::counting_semaphore sa{1}, sb{2}, me{1};
int n;

static void TA (int);
static void TB (int);
```

To "sleep" for a random time

Mutexes cannot be used because they must be locked and unlocked by the same thread

fme can also be a mutex cuz its only in charge of updating the count and printing.

Counter

2 Threads
2 semaphores
1 mutex (semaphore)
TA (sa) is the one to start

Solution

```
int main (int argc, char **argv) {
  int n1, n2;

  if (argc != 2) {
    fprintf (stderr, "Syntax: %s num_threads\n", argv[0]);
    return (1);
  }
  n1 = atoi(argv[1]);
  n2 = 2 * n1;
  n = 0;

  std::thread ta (TA, n1);
  std::thread tb (TB, n2);

  ta.join();
  tb.join();

  return (0);
}
```

To avoid running for ever we generate n1 threads TA n2 threads TB

The error message you're seeing, "Syntax: C:\Users\meela\Desktop\System And Device Programming\Practical\cmake-build-debug\LectureNotes\Unit6-synchronization\Exercise.exe num_threads", indicates that the program is expecting a command line argument representing the number of threads, but it's not receiving one. In your main function, you're checking if argc is not equal to 2. argc represents the number of command line arguments passed to the program. By convention, the first argument (at index 0) is the name of the program itself, so if argc is 2, it means one additional argument has been provided. If argc is not 2, your program prints an error message and returns 1, indicating an error. This is why your program is not running as expected. To fix this, you need to provide the required command line argument when you run your program. The argument should be an integer representing the number of threads. For example, if you want to run your program with 5 threads, you would run it like this:

// Here n1 is 1, which means the TA function will be called once.

// here n2 is 2, which means the TB function will be called twice.

Your code is implementing a multithreaded program in C++ where two threads, `TA` and `TB`, are printing characters 'A' and 'B' respectively. The goal is to synchronize these threads such that for every set of 3 characters printed, there is 1 'A' and 2 'B's in any order.

Here's a breakdown of how your code works:

1. You have two semaphores `sa` and `sb` initialized with values 1 and 2 respectively. Semaphores are used for controlling access to a common resource by multiple processes in a concurrent system such as a multitasking operating system.

2. You also have a semaphore `me` initialized with value 1, which is used as a mutex to ensure mutual exclusion when printing and incrementing the counter `n`.

3. The `TA` thread function acquires the `sa` semaphore, prints 'A', increments the counter `n`, and if `n` is greater than or equal to 3, it prints a newline, resets `n` to 0, and releases the semaphores `sa` and `sb` twice. The `me` semaphore is released at the end to allow the other thread to print.

4. The `TB` thread function works similarly, but it acquires the `sb` semaphore and prints 'B'.

5. In the `main` function, you create the `TA` and `TB` threads and join them. The `join` function blocks the main thread until the respective threads have finished execution.

The synchronization is achieved by using the semaphores `sa` and `sb`. The `TA` thread can only run when `sa` is available, and the `TB` thread can only run when `sb` is available. By releasing `sa` and `sb` twice when `n` reaches 3, you ensure that for every set of 3 characters printed, there is 1 'A' and 2 'B's. The order in which 'A' and 'B' are printed depends on the scheduling of the threads by the operating system, which can be non-deterministic.

Solution

```
static void TA (int nc) {
  for (int i=0; i<nc; i++) {
    sleep (rand()%2);
    sa.acquire();
    me.acquire();
    cout << "A";
    n++;
    if (n>=3) {
      cout << endl;
      n = 0; sa.release(); sb.release(); sb.release();
    }
    me.release();
  }
  return;
}
```

Wait for a random time

If TA starts

It must not start with TB

The last thread wakes-up one A and two B threads

Solution

```
static void TB (int nc) {
  for (int i=0; i<nc; i++) {
    sleep (rand()%2);
    sb.acquire();
    me.acquire();
    cout << "B";
    n++;
    if (n>=3) {
      cout << endl;
      n = 0;
      sa.release(); sb.release(); sb.release();
    }
    me.release();
  }
  return;
}
```

Wait for a random time

If TB starts

It must not start with TA

The last thread wakes-up one A and two B threads

In your code, you're using semaphores to control the execution of your threads. Semaphores are a synchronization primitive that can be used to protect shared resources and to coordinate the execution of threads.

In your case, you have three semaphores: `sa`, `sb`, and `me`.

- `sa` and `sb` are used to control the execution of the `TA` and `TB` threads respectively. When a thread acquires its semaphore (with `sa.acquire()` or `sb.acquire()`), it can proceed with its execution. If the semaphore is not available (i.e., its value is 0), the thread will block until it becomes available.

- `me` is used as a mutex to ensure that only one thread can print and increment the counter `n` at a time. This is necessary to prevent race conditions, where the two threads might try to print or increment `n` at the same time, leading to unpredictable results.

The `release()` function increases the value of the semaphore, making it available to other threads that might be waiting to acquire it. In your code, when `n` reaches 3, both `sa` and `sb` are released twice. This means that after every three characters printed, both `TA` and `TB` will be able to run again, ensuring that the next set of three characters will also include one 'A' and two 'B's.

The `acquire()` function decreases the value of the semaphore. If the value of the semaphore is already 0, the thread calling `acquire()` will block until another thread releases the semaphore.

In summary, the `acquire()` and `release()` functions are used to control the execution of the threads and to

Exercise 02

Exam of September 08, 2023

- A C program can execute four different threads
  - TP (thread plus), TM (thread minus), TS (thread star), and TNL (thread newline)
- Each thread is organized through an infinite cycle containing synchronization instructions but a single IO instruction
  - Thread TP displays a "+"
  - Thread TM displays a "-"
  - Thread TS displays a "*"
  - Thread TNL displays a "\n" (endln)

Exercise 02

- Synchronize the four threads to print the following sequence of lines

```
++++++++++
----------
**********
++++++++++
----------
**********
   etc.
```

  - Where the number of characters on each row is given as a parameter to the main program (e.g., 10)

Solution

```
#include <iostream>
#include <semaphore>
#include <thread>
#include <unistd.h>

using std::cout;
using std::endl;

std::counting_semaphore sp{1}, sm{0}, ss{0}, snl{0};

static void TP (int);
static void TM (int);
static void TS (int);
static void TNL ();
```

4 Threads
4 Semaphores
SP (+) is the one to start

Solution

```
int main (int argc, char **argv) {
  int n;
  if (argc != 2) {
    ... error ...
  }
  n = atoi(argv[1]);
  std::thread tp (TP, n);
  std::thread tm (TM, n);
  std::thread ts (TS, n);
  std::thread tnl (TNL);
  tp.join();
  tm.join();
  ts.join();
  tnl.join();
  return (0);
}
```

Threads never stop; but if we do not wait, we return and we stop all threads (there is no pthread_exit)

Solution

```
static void TP (int n) {
  int np = 0;
  while (1) {        In C and C++, while(1) is a loop construct that creates an infinite loop.
    sp.acquire();
    cout << "+";
    np++;
    if (np<n) {
      sp.release();      Re-wake up TP
    } else {
      np = 0;
      snl.release();     Reset the number of calls for TP and call TNL
    }
  }
  return;
}
```

Solution

```
static void TM (int n) {
  int nm = 0;
  while (1) {
    sm.acquire();
    cout << "-";
    nm++;
    if (nm<n) {
      sm.release();
    } else {
      nm = 0;
      snl.release();
    }
  }
  return;
}
```

Re-wake up TM

Reset the number of calls for TM and call TNL

Solution

```
static void TS (int n) {
  int ns = 0;
   while (1) {
    ss.acquire();
    cout << "*";
    ns++;
    if (ns<n) {
      ss.release();
    } else {
      ns = 0;
      snl.release();
    }
  }
  return;
}
```

Re-wake up TS

Reset the number of calls for TS and call TNL

Solution

```
static void TNL () {
  int nnl = 0;
  while (1) {
    snl.acquire(); nnl++; cout << endl;
    sleep (rand()%2);
    if (nnl==1) {
      sm.release();
    } else {
      if (nnl==2) {
        ss.release();
      } else {
        sp.release(); nnl = 0;
      }
    }
  }
  return;
}
```

POSIX
(we can use C++ to sleep)

Wake up TM

Wake up TS

Wake up TP
and restart

Exercise 03

- Fairness consideration on synchronization primitives
  - C++ synchronization primitives are unfair
    - Some threads can lock a mutex more often than others
      - A simple experiment on Linux shows that if threads repeatedly try to lock the same mutex, some threads lock the mutex 1.13x more often than others
    - Some threads can lock a semaphore or a spinlock 3.91x more often than others

Exercise 03

In a typical semaphore, threads waiting on the semaphore are woken up in the order they arrived (FIFO). However, in a priority semaphore, each thread has an associated priority, and the semaphore wakes up the highest priority thread first.

- Implement a **priority semaphore**, i.e., a semaphore in which
  - Each thread has an intrinsic priority
    - The priority is an integer value
    - The **higher** priority corresponds to the **lower** value
  - Unlocking is done in order following the threads priority

Solution

- Core idea
  - The semaphore must have a **priority queue** associated with it, where threads await to be signalled
  - When a call to the signal function wakes-up a thread, threads must be woken-up following their priority
    - We have to awake the threads with the higher priority among the ones waiting on that semaphore

"The semaphore must have a priority queue associated with it, where threads await to be signalled": This means that the semaphore should maintain a priority queue of threads that are waiting to acquire the semaphore. The priority queue ensures that threads with higher priority are served before those with lower priority.
"When a call to the signal function wakes-up a thread, threads must be woken-up following their priority": This means that when the semaphore is released (signalled), it should wake up the highest priority thread that is waiting on the semaphore. In other words, among all the threads that are waiting on the semaphore, the one with the highest priority should be allowed to proceed first.
"We have to awake the threads with the higher priority among the ones waiting on that semaphore": This is reiterating the point above. The semaphore should always wake up the highest priority thread first.
"In C++ lock and unlock must be called by the same thread": This is a rule in C++ for mutexes, where the same thread that locked a mutex must be the one to unlock it. However, this rule does not apply to semaphores, which can be released by any thread.
"We should use C++ semaphores but semaphores are not copyable": This means that you should use the semaphore implementation provided by C++, but you should be aware that semaphore objects cannot be copied. This is because copying a semaphore could lead to unexpected behavior, as it would create two semaphores with the same state but which are updated independently.

In C++ lock and unlock must be called by the same thread.
We should use C++ semaphores but semaphores are not copyble

Solution

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <map>
#include <thread>
#include <semaphore>

using std::cout;
using std::endl;
...

const int TIME = 3;

map<int,std::unique_ptr<std::binary_semaphore>> my_sem;
std::mutex m;
```

A queue is a linear data structure that follows a particular order in which the operations are performed. The order is First In First Out (FIFO). A good example of a queue is any queue of consumers for a resource where the consumer that came first is served first. A priority queue is a special type of queue in which each element is associated with a priority and is served according to its priority. If elements with the same priority occur, they are served according to their ordering in the queue. The element with the highest priority is served first. Here are the main differences:
Order of elements: In a standard queue, the first element that was enqueued is the first one to be dequeued. In a priority queue, the element with the highest priority is the first one to be dequeued.
Use cases: A standard queue is used when things don't have a priority and follow a sequential order. A priority queue is used when elements are supposed to be served on the basis of priority.
Implementation: A priority queue is typically implemented using Heap data structures (Binary Heap, Fibonacci Heap, etc), but it can be implemented with arrays, linked-lists, or binary trees. A queue is implemented using arrays or linked-lists.
Complexity: For a priority queue, insertion takes O(log N) time complexity and removal takes O(1) time complexity. For a standard queue, both insertion and removal operations take O(1) time complexity.

C++20 semaphore are neither copyble nor movable.
We need to carefully use dynamic memory allocation

Solution

Worker running threads

```
static void worker (int i, int priority) {
  m.lock();
  cout << "Locking thread " << i <<
          " with priority " << priority << endl;
  m.unlock();
  my_sem.insert
    ({priority,std::make_unique<std::binary_semaphore>(0)});
  (*my_sem[priority]).acquire();
  m.lock();
  cout << "           Unlocked thread " << i <<
          " with priority " << priority << endl;
  m.unlock();
  return;
}
```

The worker function is the function that each thread will run. It first locks a mutex to ensure that the following operations are atomic (i.e., not interrupted by other threads). It then prints a message indicating that the thread has been locked and its priority.

The worker function then creates a new binary semaphore and stores it in the my_sems map, with the thread's priority as the key. The binary semaphore is initially set to 0, meaning that any thread that tries to acquire it will block.

The worker function then tries to acquire the binary semaphore. Because the semaphore's count is 0, this causes the thread to block.

The worker function then unlocks the mutex and prints a message indicating that the thread has been unlocked and its priority.

Solution

Main: Part 1

```
int main (int argc, char *argv[]) {
  int i, priority;
  if (argc != 2) {
    cout << "Syntax: " << argv[0] << " num_threads\n";
    return (1);
  }
  int n = atoi (argv[1]);
  vector<thread> pool;
  for (i=0; i<n; i++) {
    priority = (i+1) * 10;
    pool.emplace_back([i, priority] { worker (i, priority); });
  }
  std::this_thread::sleep_for
    (std::chrono::seconds(rand()%TIME));
```

The main function creates a number of threads and stores them in a vector. Each thread is assigned a priority, which is simply (i + 1) * 10.

Running workers

he main function then sleeps for a certain amount of time. This is to ensure that all the threads have had a chance to start and block on their respective semaphores.

The main function then iterates over the my_sems map, which is ordered by the keys (i.e., the thread priorities). For each semaphore in the map, it releases the semaphore, which unblocks the corresponding thread.

From POSIX sleep to C++

Put the thread in a sleep status for rand()%TIME seconds

Solution

Main: Part 2

```
  i = 0;
  for (const auto &t : my_sem) {
    m.lock();
    cout << "      Unlocking thread " << i++ <<
                 " with priority " << t.first << endl;
    m.unlock();
    (*(t.second)).release();
  }
  for (i=0; i<n; i++) {
    pool[i].join();
  }
  cout << "Main exits." << endl;
  return (1);
}
```

n C++, std::map automatically sorts its keys in ascending order. So, when you iterate over my_sems, you're effectively iterating over the semaphores in order of ascending priority. Since a lower priority value indicates a higher priority level, this means you're releasing the threads in order of highest to lowest priority.

In this loop, t.first is the priority of the thread, and t.second is the semaphore associated with that thread. By calling (*t.second).release(), you're releasing the semaphore, which unblocks the thread that is waiting on it.

Wait workers

Finally, the main function waits for all the threads to finish by calling join on each thread in the vector.

Solution

Output

```
Locking thread 0 with priority 10
Locking thread 6 with priority 70
Locking thread 2 with priority 30
Locking thread 1 with priority 20
Locking thread 9 with priority 100
...
     Unlocking thread 0 with priority 10
     Unlocking thread 1 with priority 20
     ...
          Unlocked thread 0 with priority 10
          Unlocked thread 1 with priority 20
     Unlocking thread 5 with priority 60
     ...
          Unlocked thread 5 with priority 60
          Unlocked thread 9 with priority 100
          Unlocked thread 4 with priority 50
          Unlocked thread 3 with priority 40
          ...
Main exits.
```

Locking the threads

Unlocking them …

… which then start

Exercise 04

- Write a program to implement an election algorithm that elects a leader thread
  - The system has N threads
  - Each thread has its
    - Thread identifier
    - Rank, i.e., and integer value randomly generated
  - To elect the leader each thread must
    - Compare its own rank value with the current value in **best_rank** to decide if it is the leader or not
      - To do that, it synchronizes with all the other threads
      - It re-start when the election process is completed (i.e., all other threads have updated the value of **best_rank**)

Exercise 04

- When all threads have done their job, each thread displays
  - Its identifier and its rank value
  - The leader thread identifier and its rank value
- Restriction
  - Threads cannot access the rank value of other threads, only the current best thread rank value is available in a global variable **best_rank** together with the corresponding thread identifier
  - Hint: Referring to a voting algorithm, use a global variable to count the number of threads that completed their voting process

Solution

C Code
Write the corresponding C++

```
#include <sys/time.h>
#include <time.h>
#include <stdlib.h>

#define N 10

typedef struct best_s {
  int rank;
  long int id;
  int num_votes;
  pthread_mutex_t mutex;
} best_t;

best_t *best;
sem_t *sem;
int max_random (int max);
```

Thread structure

The code starts by defining a structure best_s that contains the best rank, the id of the thread with the best rank, the number of votes (threads that have compared their rank with the best rank), and a mutex for synchronizing access to the structure.

mutex for synchronizing access to the structure.

Semaphore to make threads wait

Solution

```
int main (int argc, char **argv){
  pthread_t th;
  int i, j, k, pi;
  best = (best_t *) malloc (sizeof (best_t));
  best->rank = best->num_votes = 0;
  pthread_mutex_init (&best->mutex, NULL);
  sem = (sem_t *) malloc (sizeof (sem_t));
  sem_init (sem, 0, 0);
  for (i = 0; i < N; i++) {
    // Assign a rank to pi
    ...
    pthread_create (&th, NULL, process, (void *) pi);
  }
  pthread_exit (0);
}
```

The main function then creates N threads. Each thread is assigned a rank and then started with the process function as its entry point.

n the main function, memory is allocated for an instance of best_s and a semaphore

The rank and number of votes in the best_s instance are initialized to 0, and the mutex is initialized.

The main function then creates N threads. Each thread is assigned a rank and then started with the process function as its entry point.

Must assign different rank values

The process function is where the election algorithm is implemented. Each thread locks the mutex to safely access the best_s instance. It then compares its rank with the best rank. If its rank is higher, it updates the best rank and the id of the thread with the best rank.

Solution

```
static void *process (void *arg){
  int rank = (int) arg;
  int i;
  long int id;
  id = pthread_self ();
  pthread_detach (pthread_self ());
```

Solution

```
pthread_mutex_lock (&best->mutex);
if (rank > best->rank) {
  best->rank = rank;
  best->id = id;
}
best->num_votes++;
if (best->num_votes < N){
  pthread_mutex_unlock (&best->mutex);
  sem_wait (sem);                /* wait for all to vote */
} else {
  pthread_mutex_unlock (&best->mutex);
  for (i = 0; i < N - 1; i++)
    sem_post (sem);              /* release all waiting */
}
printf ("my_id=%ld my_rank=%d leader_id=%ld leade_rank=%d\n",
 id, rank, best->id, best->rank);
}
```

Check personal rank with best global rank

Update best rank

If not the last one, wait the others

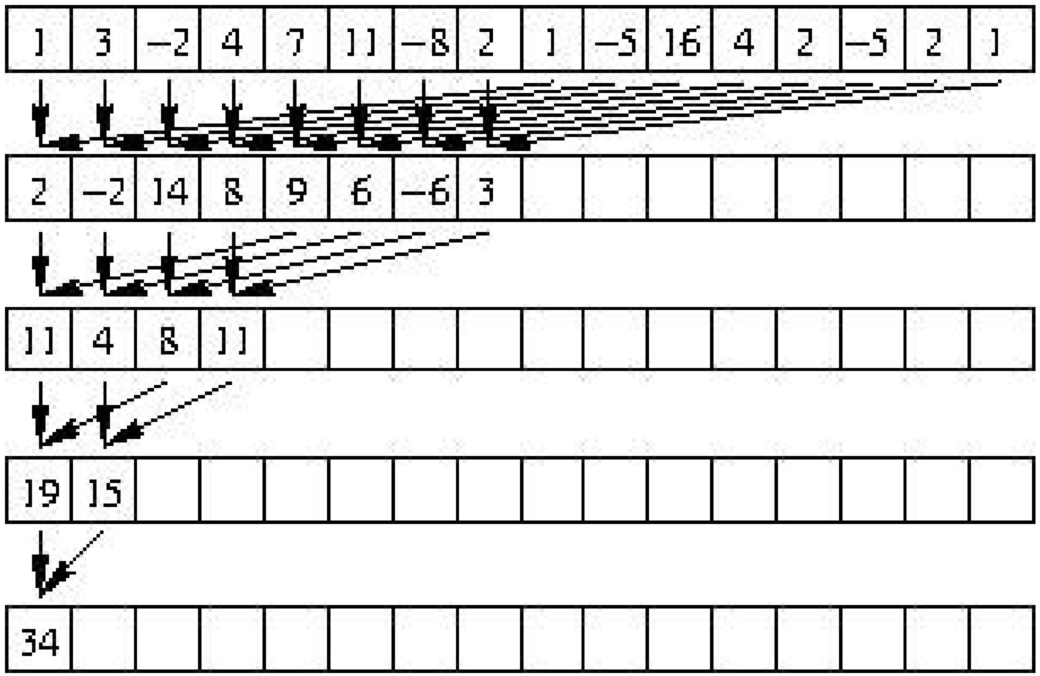
If the last one, release all

Exercise 05

- We are viven an array vet of size n
  - We supposed **n** is a **power of 2** (e.g., 16)
- Write the function

```
int array_sum (int *vet, int n);
```

- Which computes the sum of the elements of the array as represented in the picture



Exercise 05

- In particular
  - All sums must be executed in parallel by n/2 (at most) separate threads
  - Each thread is associated with one of the first n/2 cells of the array
- Note that the number of sums each thread will have to execute depends on the position of the cell
- Manage synchronization between threads with semaphores, so that all sums are made respecting precedence

Solution

C Code
Write the corresponding C++

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>

typedef struct {
  int *vet;
  sem_t *sem;
  int n;
  int id;
} args_t;

... main ...
```

Array of n elements

Array of n/2 semaphores

User and thread identifier

Initialize variables and calls function array_sum

Solution

Call the thread functions

```
int array_sum (int *vet, int n) {
  int k=n/2; pthread_t *tids; args_t *args; sem_t *sem;
  tids = (pthread_t *) malloc (k*sizeof(pthread_t));
  sem = (sem_t *) malloc (k*sizeof(sem_t));
  for (int i=0; i<k; ++i) sem_init(&sem[i], 0, 0);
  args = (args_t *) malloc (k*sizeof(args_t));
  for (int i=0; i<k; ++i) {
    args[i].id = i; args[i].vet = vet;
    args[i].n = n; args[i].sem = sem;
  }
  for (int i=0; i<k; ++i)
    pthread_create (&tids[i], NULL, adder, &args[i]);
  pthread_join (tids[0], NULL);
  for (int i=0; i<k; ++i) sem_destroy(&sem[i]);
  free (tids);
  free (sem);
  free (args);
  return vet[0];
}
```

n/2 Ts and Sems

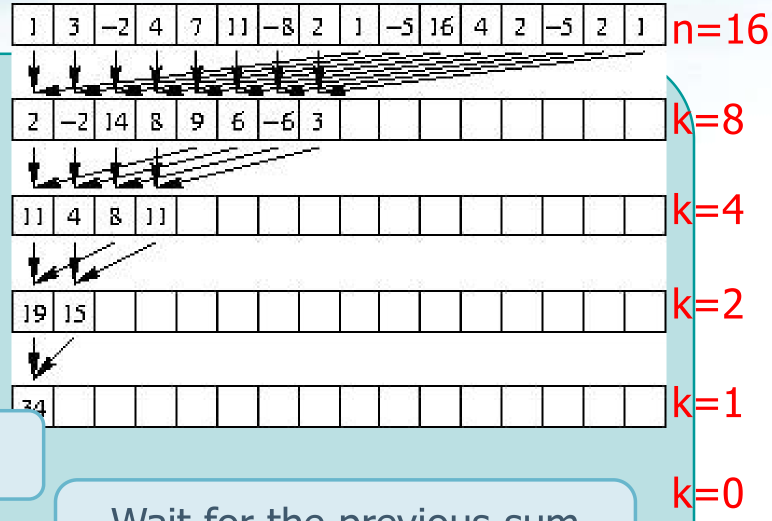
Initialize

Run threads

Wait for threads and free memory

Solution

Thread function




```
void *adder (void * arg) {
  sem_t *sem = ((args_t *) arg)->sem;
  int *vet = ((args_t *) arg)->vet;
  int id = ((args_t *) arg)->id;
  int n = ((args_t *) arg)->n;
  int k = n/2, i = 0;
  while (k != 0) {
    if (i!=0 && k<n/2)
      sem_wait (&sem[id + k]);
    else
      i++;
    vet[id] += vet[id + k];
    k = k/2;
    if (id >= k) {
      sem_post (&sem[id]);
      break;
    }
  }
  pthread_exit(0);
}
```

k = # iterations

Wait for the previous sum to be done id ∊ [0, n/2[

... but not during the fist cycle

Make the sum

My sum has been done

This thread must stop