System and Device Programming

Synchronization Exercises (part C)

Stefano Quer

Dipartimento di Automatica e Informatica

Politecnico di Torino

Exercise

- Re-implement the following piece of code using only semaphores and mutexes to synchronize threads

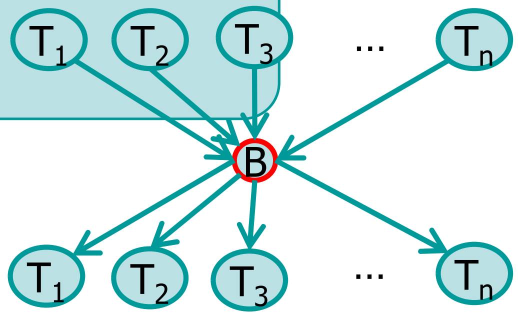
Main

```
pthread_barrier_t b;
...
pthread_barrier_init (&b, NULL, N_THREAD);
for (i=0; i<N_THREAD; i++) {
  err = pthread_create (&tid[i], NULL, thr_fn, NULL);
}
...
```

Threads (acyclic behavior)

```
void *thr_fn () {
  ...
  pthread_barrier_wait (&b);
}
```

Synchronization point among all threads



Solution

```
typedef struct barrier_s {
  sem_t sem;
  pthread_mutex_t mutex;
  int count;
} barrier_t;
```

Barrier structure
sem to enqueue threads
mutex to protect counter
counter to count threads up

Init barrier

```
barrier_d = (barrier_t *) malloc (1 * sizeof(barrier_t));
sem_init (&barrier_d->sem, 0, 0);
pthread_mutex_init (&barrier_d->mutex, NULL);
barrier_d->count = 0;
```

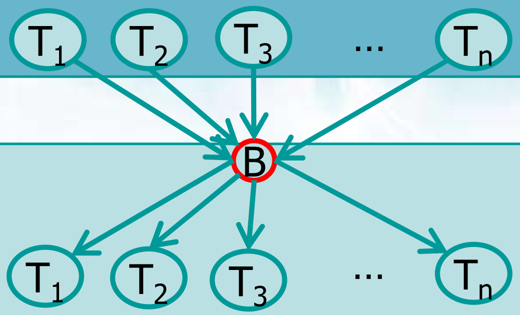
Main

Run threads

```
for (i=0; i<N_THREAD; i++) {
  err = pthread_create (&tid[i], NULL, thr_fn, NULL);
}
```

Solution 1

Threads (acyclic behavior)



```
void *thr_fn () {
  ...
  pthread_mutex_lock (&barrier->mutex);
  barrier->count++;
  if (barrier->count == N_THREAD) {
    for (j=0; j<N_THREAD; j++) {
      sem_post (&barrier_d->sem);
    }
  }
  pthread_mutex_unlock (&barrier->mutex);
  sem_wait (&barrier_d->sem);

  pthread_exit ();
}
```

Protect counter

Last thread awakes all

Waiting point for all threads

Solution 2

Solution with turnstile

```
void *thr_fn () {
  ...
  pthread_mutex_lock (&barrier->mutex);
  barrier->count++;
  if (barrier->count == N_THREAD) {
      sem_post (&barrier_d->sem);
    }
  }
  pthread_mutex_unlock (&barrier->mutex);
  sem_wait (&barrier_d->sem);
  sem_post (&barrier_d->sem);

  pthread_exit ();
}
```

Turnstile

One **extra** sem_post is done (pay attention to cycling threads)

Exercise

- Re-implement the following piece of code using only semaphores and mutexes to synchronize threads

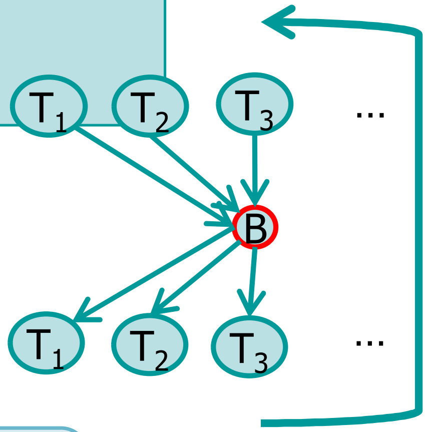
```
pthread_barrier_t b;
...
pthread_barrier_init (&b, NULL, N_THREAD);
for (i=0; i<N_THREAD; i++) {
  err = pthread_create (&tid[i], NULL, thr_fn, NULL);
}
...
```

Main

```
void *thr_fn () {
  while (1)
    ...
    pthread_barrier_wait (&b);
  }
}
```

Threads (**cyclic** behavior)

Synchronization point among all threads



Buggy Solution

```
void *thr_fn () {
  while (1) {
    ..
    pthread_mutex_lock (&barrier->mutex);
    barrier->count++;
    if (barrier->count == N_THREAD) {
      for (j=0; j<N_THREAD; j++) {
       sem_post (&barrier_d->sem);
      }
    }
    pthread_mutex_unlock (&barrier->mutex);
    sem_wait (&barrier_d->sem);
  }
  pthread_exit ();
}
```

Last threads awakes all

Waiting point for all threads

A fast threads can cycle more than once !

Solution

```
typedef struct barrier_s {
  sem_t sem1, sem2;
  pthread_mutex_t mutex;
  int count;
} barrier_t;
```

Barrier structure
2 sems to enqueue threads
mutex to protect counter
counter to count threads up

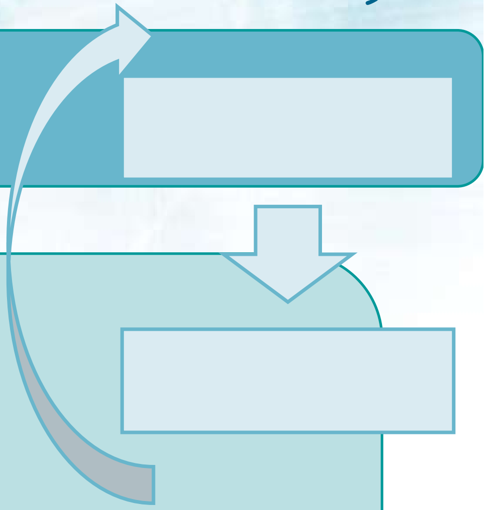
Init barrier

```
barrier_d = (barrier_t *) malloc (1 * sizeof(barrier_t));
sem_init (&barrier_d->sem1, 0, 0);
sem_init (&barrier_d->sem2, 0, 0);
pthread_mutex_init (&barrier_d->mutex, NULL);
barrier_d->count = 0;
```

Main

Run threads

```
for (i=0; i<N_THREAD; i++) {
  err = pthread_create (&tid[i], NULL, thr_fn, NULL);
}
```



Cyclic behavior

Barrier #1

```
...
pthread_mutex_lock (&barrier->mutex);
barrier->count++;
if (barrier->count == N_THREAD) {
  for (j=0; j<N_THREAD; j++) sem_post (&barrier_d->sem1);
}
pthread_mutex_unlock (&barrier->mutex);
sem_wait (&barrier_d->sem1);
```

Barrier #2

```
pthread_mutex_lock (&barrier->mutex);
barrier->count--;
if (barrier->count == 0) {
  for (j=0; j<N_THREAD; j++) sem_post (&barrier_d->sem2);
}
pthread_mutex_unlock (&barrier->mutex);
sem_wait (&barrier_d->sem2);
...
```

Exercise

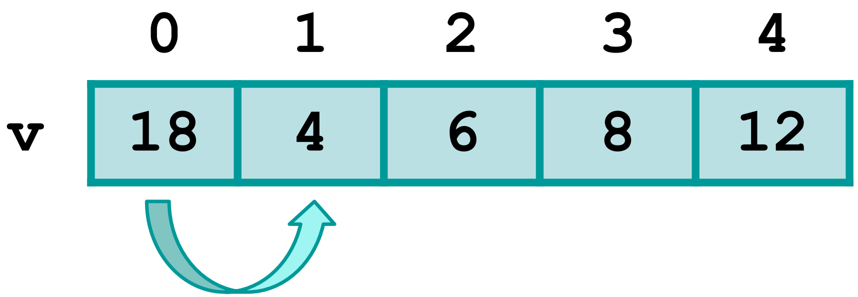
- A concurrent program want to sort an array using the bubble sort algorithm as follow
  - A static vector contains n integer elements
  - The main thread runs n-1 identical threads
  - Each thread manages two adjacent elements
    - Thread 0 manages elements 0 and 1
    - Thread 1 manages elements 1 and 2
    - …
    - Thread n-1 manages elements n-1 and n

Exercise

- Each thread
  - Compare the two elements it deals with, and exchange them if they are not in the correct order
  - Once their work is finished, all the threads wait for each-other, and if
    - All the elements are correctly ordered, the program terminates
    - Otherwise, all threads are run again to make a new series of exchanges

As the order in which all swaps are performed is not defined (inner iteration) the number of necessary outer iterations is upper bounded by n

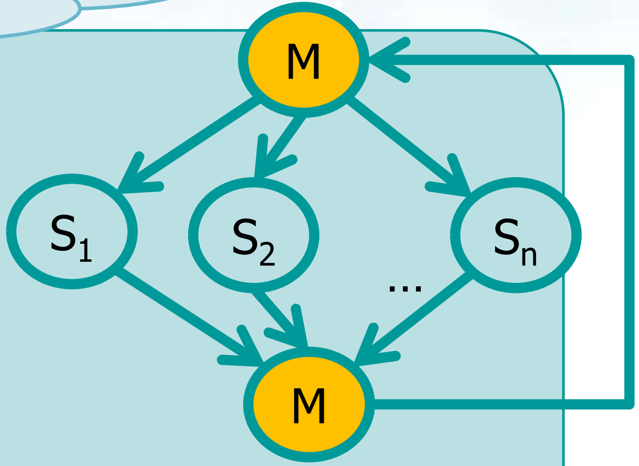
```
for (i=0; i<n-1; i++)
  for (j=0; j<n-i-1; j++)
    if (v[j] > v[j+1])
      swap (v, i, j+1);
```



	0	1	2	3	4
v	18	4	6	8	12

Solution 1

Solution with semaphores
(no barriers)



```
#include <stdio.h>

typedef enum {false, true} boolean;

int num_threads;
int vet_size;
int *vet;
boolean sorted = false;
boolean all_ok = false;
sem_t semMaster;
sem_t *semSlave;
pthread_mutex_t *me;

static int max_random (int);
void *master (void *);
void *slave (void *);
```

Boolean type

Global variables:
1 semaphore for the master thread
1 semaphore for each slave thread
1 mutex for each element of the vector

Prototypes

Solution 1

Main (Estract)
Part 1

```
int main (int argc, char **argv) {

  ... Definitions ...

  vet_size = atoi (argv[1]);
  num_threads = vet_size - 1;

  ... Allocations ...

  for (i=0; i<vet_size; i++) {
    vet[i] = max_random (1000);
  }
  for (i=0; i<vet_size; i++) {
    pthread_mutex_init (&me[i], NULL);
  }
```

Fill the vector with random numbers

Create a mutex for each element of the vector

Solution 1

Main (Estract)
Part 2

MT starts

```
sem_init (&semMaster, 0, num_threads);
pthread_create (&thMaster, NULL, master, &num_threads);

for (i=0; i<num_threads; i++) {
  id[i] = i;
  sem_init (&semSlave[i], 0, 0);
  pthread_create (&thSlave[i], NULL, slave, &id[i]);
}

for (i=0; i<num_threads; i++) {
  pthread_join (thSlave[i], NULL);
}
pthread_join (thMaster, NULL);

... Free memory and semaphores ...
```

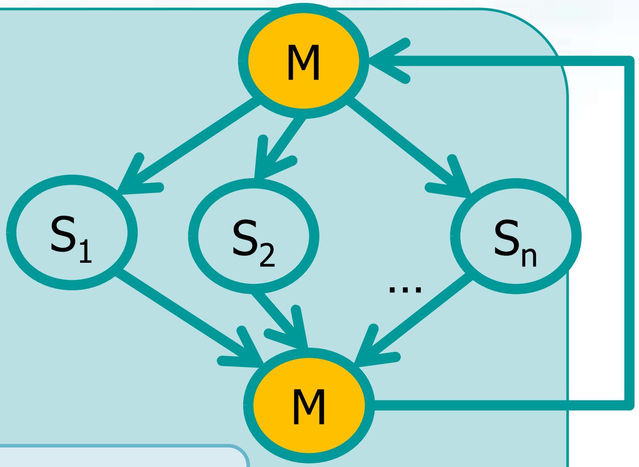
Creates 1 master thread

STs wait

Creates num_threads
slave threads

Solution 1

```
void *master (void *arg) {
  int *ntp, nt, i;
  ntp = (int *) arg;
  nt = *ntp;
  while (!sorted) {
    for (i=0; i<nt; i++)
      sem_wait (&semMaster);
    if (all_ok) {
      sorted = true;
    } else {
      all_ok = true;
    }
    for (i=0; i<nt; i++)
      sem_post (&semSlave[i]);
  }
  pthread_exit (0);
}
```



Wait for slave threads

Initially false

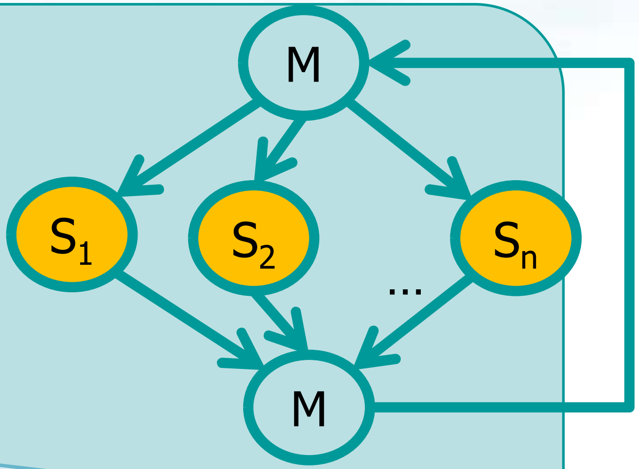
If it remains false, at the next iteration we set sorted=true and we stop

Wake up slave threads

Solution 1

```
void *slave (void *arg) {
  int i = *((int *) arg);
  while (1) {
    sem_wait (&semSlave[i]);
    if (sorted) break;
    pthread_mutex_lock(&me[i]);
    pthread_mutex_lock(&me[i+1]);
    if (vet[i] > vet[i + 1]) {
      swap (vet[i], vet[i + 1]);
      all_ok = false;
    }
    pthread_mutex_unlock(&me[i+1]);
    pthread_mutex_unlock(&me[i]);
    sem_post (&semMaster);
  }
  pthread_exit (0);
}
```

Wait master thread



Acquires the 2 elements it has to manage

It orders them

Wake up master thread

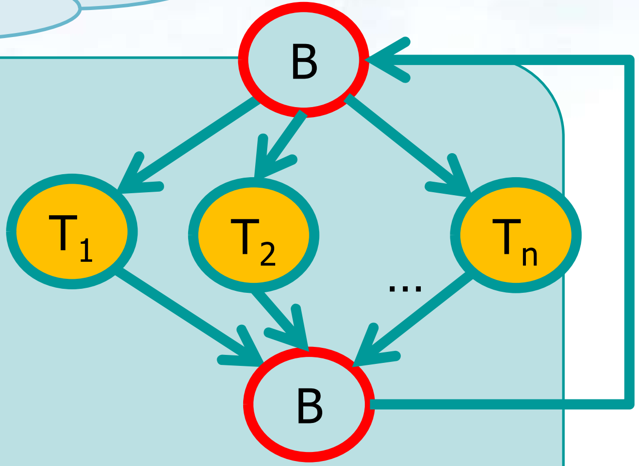
Solution 2

Solution with barriers

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

#define N 10

int count, vet[N];
int sorted = 0;
int all_ok = 1;
sem_t me[N];
sem_t mutex, barrier1, barrier2;
```



Instead of using one semaphore for each slave, why do not use barriers?

Solution 2

Read or generate the array

```
int main (int argc, char * argv[]) {
  ...
  count = 0;
  sem_init (&mutex, 0, 1);
  sem_init (&barrier1, 0, 0);
  sem_init (&barrier2, 0, 0);

  for (i=0; i<N; i++)
    sem_init (&me[i], 0, 1);

  for (i=0; i<N-1; i++) {
    id[i] = i;
    pthread_create (&th[i], NULL, sorter, &id[i]);
  }

  pthread_exit (0);
}
```

Create a mutex to protect the counter, and 2 barriers based on semaphores

Create a semaphore for each element of the vector

Create N threads

No joins
(threads are detached)

Solution 2

```
static void *sorter (void *arg) {
  int *a = (int *) arg;
  int i, j, tmp;

  i = *a;

  pthread_detach (pthread_self ());

  while (!sorted) {
    sem_wait (&me[i]);
    sem_wait (&me[i+1]);
    if (vet[i] > vet[i+1]) {
      swap (vet[i], vet[i + 1]);
      all_ok = 0;
    }
    sem_post (&me[i + 1]);
    sem_post (&me[i]);
```

Acquires the 2 elements it has to manage

It orders them

all_ok remains 1 if no thread makes an exchange

Release the access of the 2 elements of the vector

Solution 2

Barrier #1

```
sem_wait (&mutex);
count++;
if (count == N-1) {
  for (j=0; j<N-1; j++)
    sem_post (&barrier1);
}
sem_post (&mutex);

sem_wait (&barrier1);
```

Before the iteration, you need to synchronize all the threads

The last thread to arrive unblock all threads

All the other threads wait on a barrier

Mutex to protect count

Solution 2

Barrier #2

```
    sem_wait (&mutex);
    count--;
    if (count == 0) {
      printf ("all_ok %d\n", all_ok);
      for (j=0; j<N; j++)
        printf ("%d ", vet[j]);
      printf ("\n");
      if (all_ok)
        sorted = 1;
      all_ok = 1;
      for (j=0; j<N-1; j++)
        sem_post (&barrier2);
    }
    sem_post (&mutex);
    sem_wait (&barrier2);
  }
  return 0;
}
```

Only one barrier is not enough, because the last thread wake up all the threads, and a fast thread can iterate more times

For this reason a second barrier is used

Restart (if necessary)

Block everything

The last thread to arrive unblock all

All the other threads wait on a barrier

Solution 3

- How can we use pthread_barrier_wait?

```
... Hug ???
```