Synchronization

Barriers

Stefano Quer

Dipartimento di Automatica e Informatica

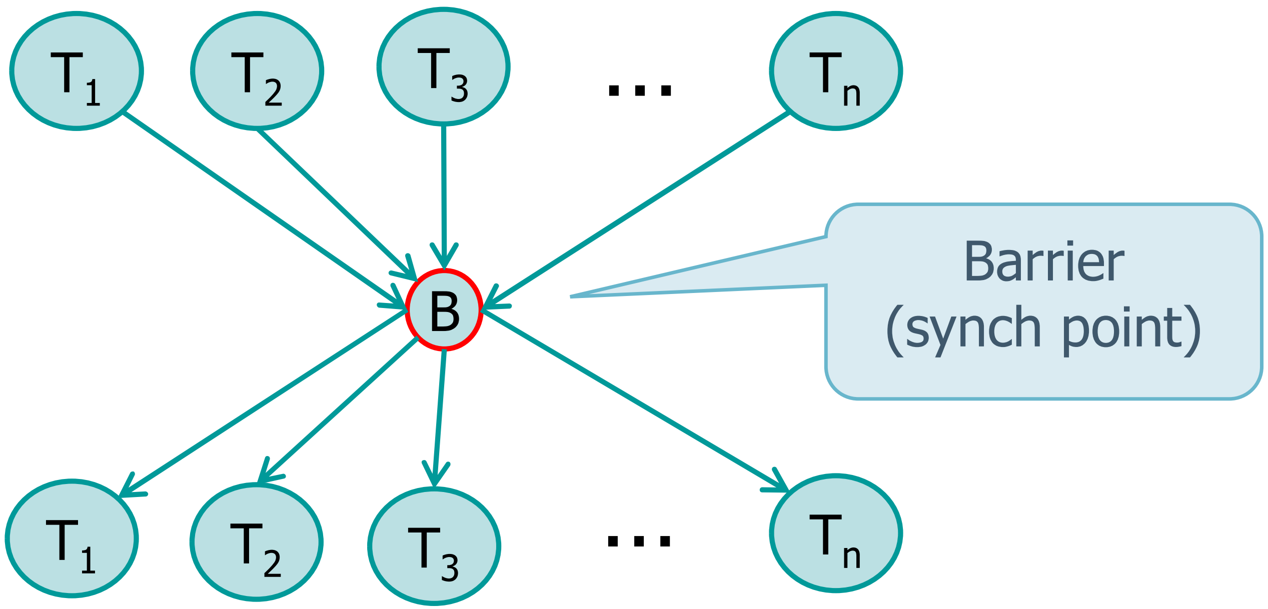
Politecnico di Torino

License Information

Barriers

- Barriers can be used to coordinate multiple threads working in parallel
  - A barrier allows each thread to wait until all cooperating threads have reached the same point, and then continue executing from there

o ensure that all threads have completed a certain task before moving on to the next task. This is useful in scenarios where threads need to synchronize their progress.



Barriers

- Barriers generalize the pthread_join function
  - The function **pthread_join** acts as a barrier to allow **one** thread to wait until another thread completed their processing
  - Barriers allow **an arbitrary number** of threads to wait until all of the threads have completed their processing
  - The threads don't have to exit, as they can continue working after all threads have reached the barrier

Generalization of pthread_join:
pthread_join: This function allows one thread to wait for another thread to finish. Think of it as waiting for one friend to catch up before you can continue.
Barriers: Instead of waiting for just one friend, you wait for all your friends. Barriers allow multiple threads to wait until all of them have reached the barrier point.
Advantages of Barriers: Threads don't have to stop or exit after reaching the barrier. They can continue working on new tasks once all threads have synchronized at the barrier.

Trivial solution

Semaphores: Think of semaphores as signals or flags that threads use to communicate with each other.

It uses too many semaphores

- A possible trivial solution
  - Use one semaphore for each thread T_i
  - It implement one for the extra process B (for the barrier) using one more semaphore

Each thread has its own semaphore.
There is an additional semaphore for the barrier process

Barrier Process:
The barrier process waits for a signal from each thread, indicating that they have reached the barrier.
Once all threads have signaled, the barrier process signals all threads to continue.

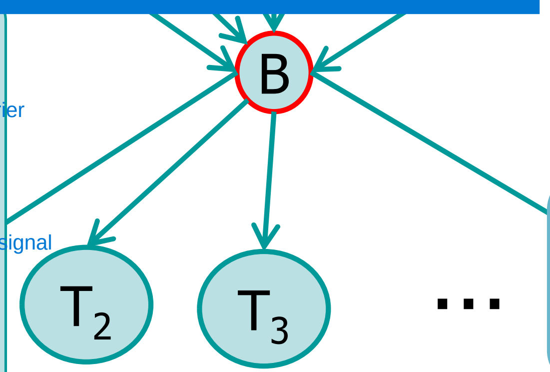
The barrier process waits n times on its semaphores and then wakes-up all threads T_i

B

The barrier process waits for a signal from each thread, indicating that they have reached the barrier. Once all threads have signaled, the barrier process signals all threads to continue.

```
for (i=0; i<n; i++)
  wait (semB); // Wait for each thread to signal
for (i=0; i<n; i++)
  signal (sem[i]);
```

// Signal all threads to continue



T_i

```
signal (semB);
wait (sem[i]);

Waiting ...
```

Signal the barrier process

Wait for the barrier process to signal back

Each T_i wakes-up the barrier process and wait on its own semaphore

POSIX versus C++

- In the POSIX standard barriers are implemented using the primitives

These functions are used to create and use barriers in POSIX-compliant systems (like many Unix-based systems).

  - pthread_barrier_init, pthread_barrier_wait
- In C++ standard barriers are implemented by the class template std::barrier
  - Unlike **std::latch**, barriers are reusable

Difference: Unlike std::latch, which is a one-time use synchronization point, std::barrier can be reused. Once all threads have synchronized, the barrier can be used again

    - Once a group of arriving threads are unblocked, the barrier can be reused
- We will use the POSIX implementation
  - See the documentation for the C++ version

POSIX barries

For more details see the reference documentation

These are implemented in the pthread.h library in POSIX-compliant systems.

- In the POSIX standard barriers are implemented in phtread.h

Type	Meaning
Purpose: Initializes a barrier. Parameters: Takes a barrier object, attributes (usually NULL), and a count of the number of threads that must reach the barrier before any can proceed. Usage: Can be initialized multiple times, but be mindful of the current value of the counter. int pthread_barrier_init (...);	The count argument specifies the number of threads that must reach the barrier before all of the threads will be allowed to continue. The same barrier can be initialized more than once (but pay attention to the current value of the counter).
Once the specified number of threads have called this function, all threads can proceed. int pthread_barrier_wait (...);	Indicate that a thread is done with its work and it is ready to wait for all the other threads to catch up.
int pthread_barrier_destroy (...); De-initializes the barrier and frees any resources allocated for it.	De-initialize a barrier. Any resource allocated for the barrier will be freed.

Example

Main program calling the threas

Define the barrier

Init the barrier with a NULL attribute and a counter equal to N

Destroy the barrier

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#define N 4      Number of threads
#define C 5      some other value c
...
pthread_barrier_t bar;
...
pthread_barrier_init (&bar, NULL, N);   Initializes the barrier bar with N threads. The second parameter is for attributes, which is NULL here.
for (i=0; i<N; i++) {
  v[i] = i;  \      th is an array of thread identifiers. Each element in the array represents a thread that is created and managed by the program.
  pthread_create (&th[i], NULL, f, (void *) &v[i]);
}
for (i=0; i<N; i++) {
  pthread_join (th[i], NULL);      Waits for all threads to finish
}
pthread_barrier_destroy(&bar);
```

pthread_t is a data type used to uniquely identify a thread in POSIX threads (pthreads).
th is an array of pthread_t with N elements, where N is the number of threads defined by #define N 4.

(void *) &v[i] is the argument passed to the thread function f. Here, v[i] is cast to a void pointer.

A loop creates N threads.
Each thread runs the function f.
v[i] is passed as an argument to the thread function f.

Why are we passing void pointer: int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void *), void *arg); The fourth parameter, void *arg, is a pointer to the argument that will be passed to the thread function.Casting to void *:
(void *) &v[i] casts the address of v[i] to a void pointer.
This is necessary because the thread function f is defined to take a void * argument.

Example

Use the barrier to synchronize N threads **once**

f is a function that takes a void pointer and returns a void pointer.

```
void *f (void *par) {
  int *np, n;

  np = (int *) par;
  n = *np;

  fprintf (stdout, "T%d-In\n", n);
  pthread_barrier_wait(&bar);
  fprintf (stdout, "  T%d-Out\n", n);

  pthread_exit (NULL);
}
```

np is a pointer to an integer.
n is an integer.

The thread function f is defined to take a void * argument and return a void *.
This is required by the pthread_create function, which expects the thread function to have this signature.
Parameter Handling:
void *par is the argument passed to the thread function.
Inside the function, par is cast to the appropriate type:
This cast converts the void * back to an int *, allowing the function to access the integer value.
Dereferencing:
n = *np; dereferences the integer pointer to get the actual integer value.

Casts the void pointer par to an integer pointer and dereferences it to get the value.

Prints a message before and after the barrier wait.
pthread_barrier_wait(&bar) makes the thread wait at the barrier until all threads reach this point.

Example

```
void *f (void *par) {
  int i, *np, n;

  np = (int *) par;
  n = *np;
  for (i=0; i<C; i++) {
    fprintf (stdout, "T%d-In%d\n", n, i);
    pthread_barrier_wait(&bar);
    fprintf (stdout, "  T%d-Out%d\n", n, i);
  }

  pthread_exit (NULL);
}
```

Adds a loop counter i.

Use the barrier to synchronize N threads **C times**

A loop runs C times.
Each iteration prints a message before and after the barrier wait.
pthread_barrier_wait(&bar) makes the thread wait at the barrier in each iteration.

Summary
POSIX Barriers: Use pthread_barrier_init, pthread_barrier_wait, and pthread_barrier_destroy to manage barriers.
Example Code: Demonstrates how to initialize, use, and destroy a barrier.
Function f: Shows how threads can synchronize using barriers, with one example synchronizing once and another synchronizing multiple times.

The barrier does not have to be re-initialized

Barrier Initialization: pthread_barrier_init(&bar, NULL, N) initializes the barrier for N threads.
Barrier Wait: pthread_barrier_wait(&bar) makes threads wait until all N threads reach this point.
Barrier Destruction: pthread_barrier_destroy(&bar) cleans up the barrier.
Thread Function: The function f demonstrates how threads can synchronize using barriers, with one example synchronizing once and another synchronizing multiple times without reinitializing the barrier.

Conclusions

- Barriers are used to coordinate multiple threads working in parallel
  - You want all threads to wait until everyone has arrived at a certain point
  - A simple semaphore would do the exact opposite, i.e., each thread would keep running and the last one will go to sleep

Exercise

- Suppose barrier constructs do not exist
  - Re-implement them using only **one** semaphore **and** one mutex

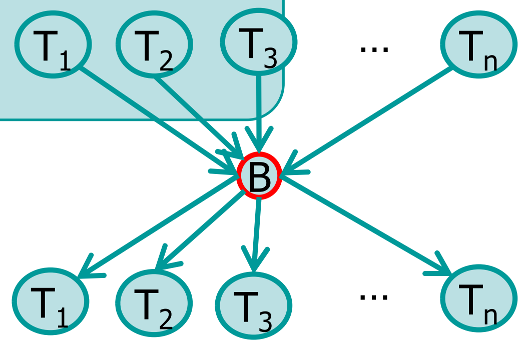
Main

```
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

Initializazion

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

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

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;
```

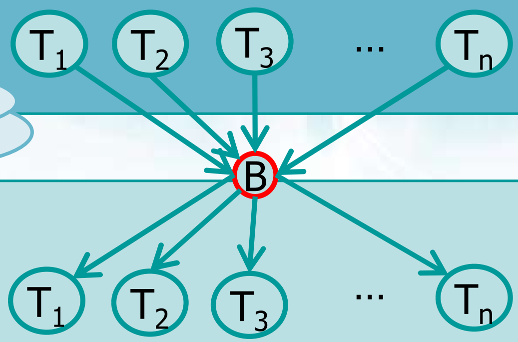
Main

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

Un-protect counter

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 ();
}
```

Protect counter

Un-protect counter

Turnstile

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

Exercise

- Re-implement the following (cyclic) piece of code using only one semaphore and one mutex

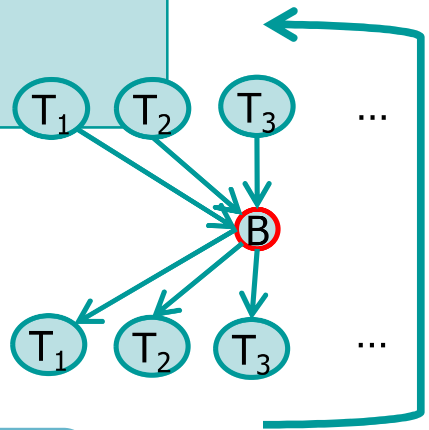
```
pthread_barrier_t b;
...
pthread_barrier_init (&b, NULL, N_THREAD);
for (i=0; i<N_THREAD; i++) {
  err = pthread_create (&tid[i], NULL, thr_fn, NULL);
}
...
```

Main

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

Threads (cyclic behavior)

Synchronization point among all threads



Buggy Solution

Buggy attempt

```
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

Initializazion

```
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);
}
```

Threads (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);

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);
...
```

Barrier #2

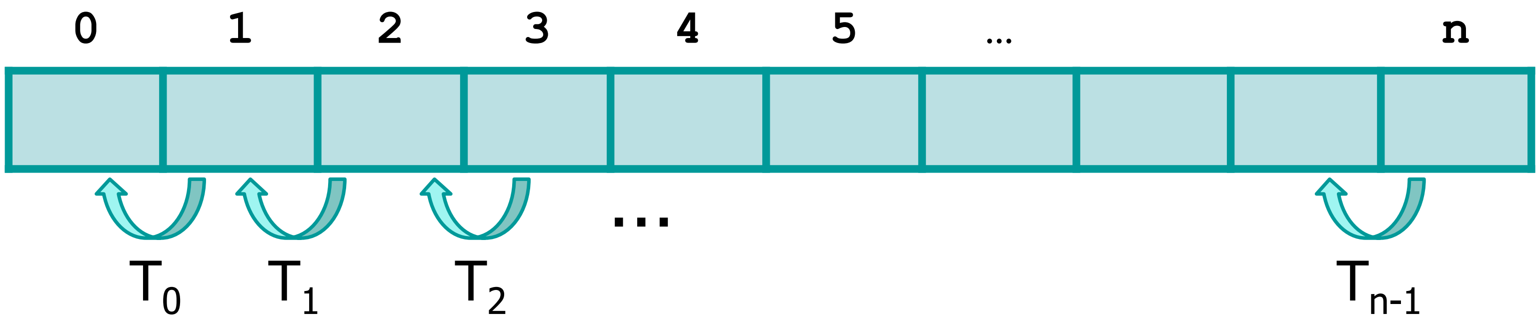
Exercise

Concurrent Bubble-sort

- Write a version of the exchange (bubblesort) sorting algorithm) as follows
  - A static array include n integer values
  - We want to ort it using n identical threads
  - Each thread is in charge of sorting two adjacent elements
    - Thread 0 sort elements 0 and 1
    - Thread 1 sort elements 1 and 2
    - …
    - Thread n-1 sort 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



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

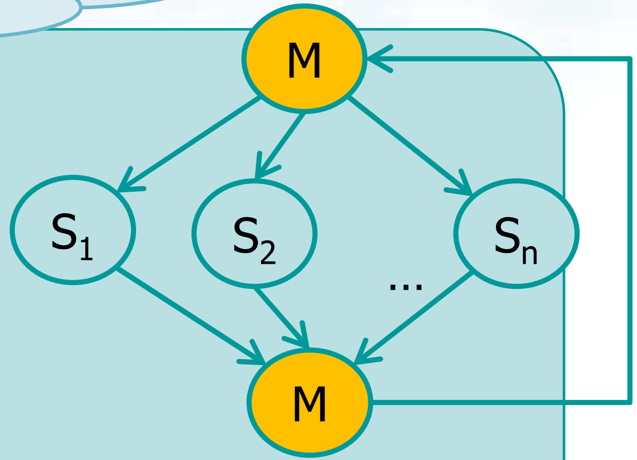
As the order in which all swaps are performed is not defined (inner iteration) the number of necessary outer iterations is upper bounded by n

```
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 in C with semaphores (no barriers)

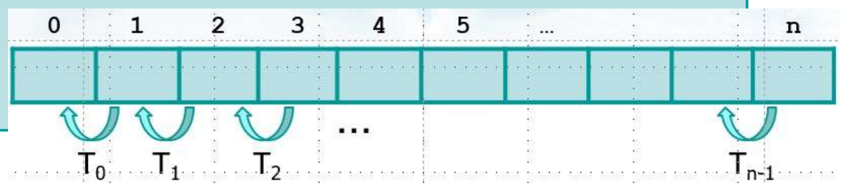



```
#include <stdio.h>

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 (implicit in C++)

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

Prototypes



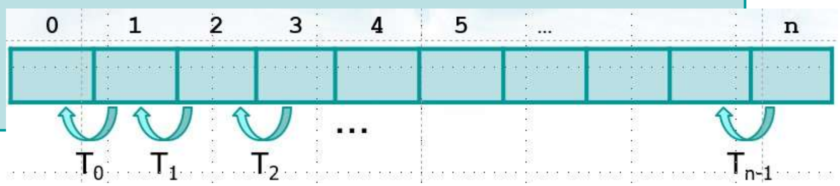
Solution 1

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

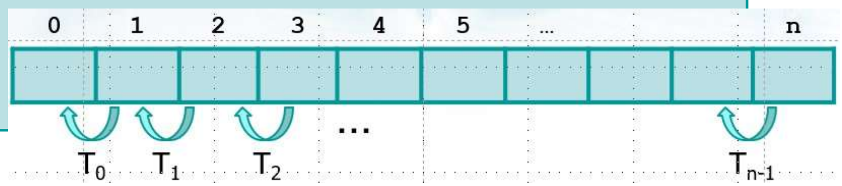
```
sem_init (&semMaster, 0, num_threads);
pthread_create (&thMaster, NULL, master, &num_threads);

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

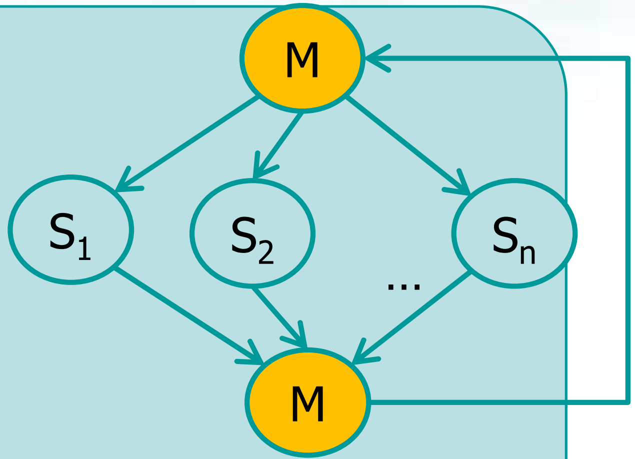
Creates num_threads slave threads

Wait all



Solution 1

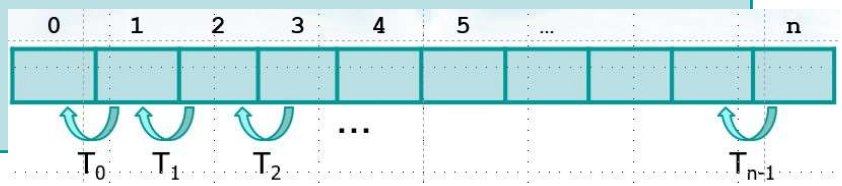
```
void *master (void *arg) {
  int *ntp, nt, i;
  ntp = (int *) arg;
  nt = *ntp;
  while (!sorted) {
    for (i=0; i<nt; i++)
      sem_wait (&semMaster);
    if (all_ok) {
      sorted = true;
    } else {
      all_ok = true;
    }
    for (i=0; i<nt; i++)
      sem_post (&semSlave[i]);
  }
  pthread_exit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



Wait for slave threads

If a worker performs a swap, it sets all_ok to false and here we set if back to true.
If no worker performs a swap, all_ok remais true, we set sorted to true, and slaves will stop at the next iteration

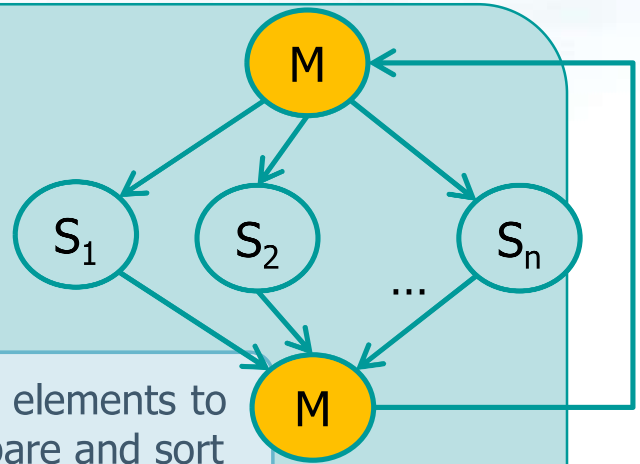
Wake up slave threads



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 the master thread

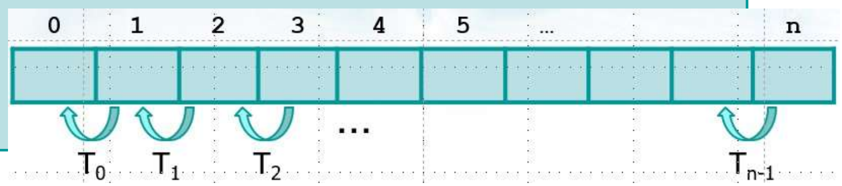


Get 2 elements to compare and sort

Sort them: If we do, set all_ok to false (cycle again)

Release elements

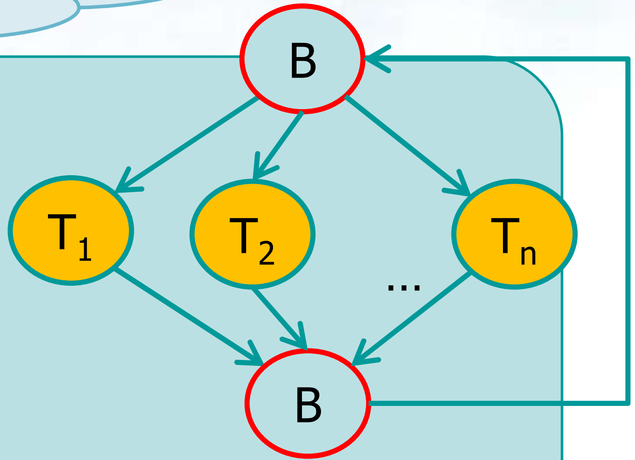
Wake up master thread



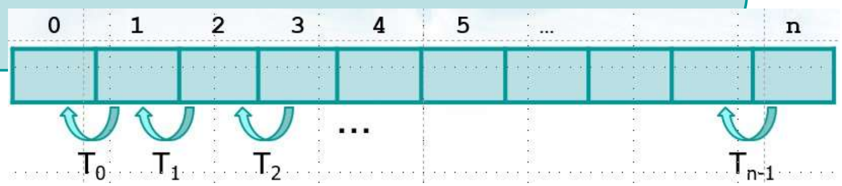
Solution 2

Solution in C 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

```
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);
}
```

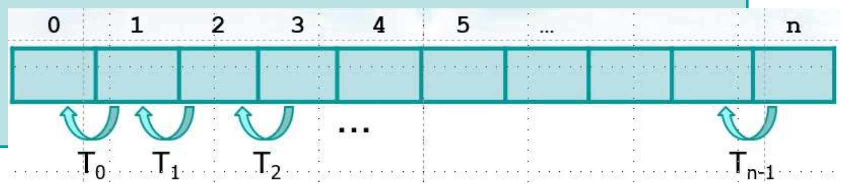
read_array read or generate the array

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]);
```

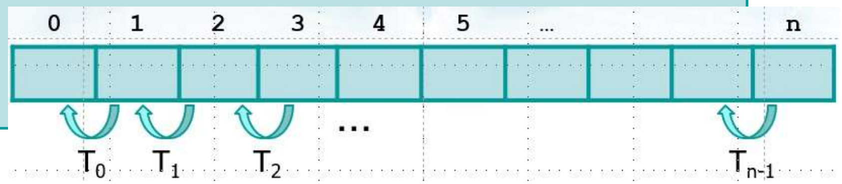
We do not need the master any more !!!

Acquires the 2 elements it has to manage

Sort them

all_ok remains 1 if no thread makes an exchange

Release the array elements



Solution 2

Barrier #1

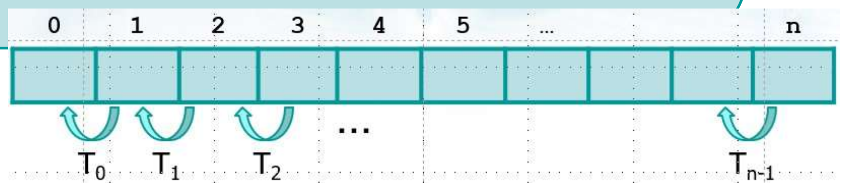
```
sem_wait (&mutex);
count++;
if (count == N-1) {
  for (j=0; j<N-1; j++)
    sem_post (&barrier1);
}
sem_post (&mutex);

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

The mutex protect the counter



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

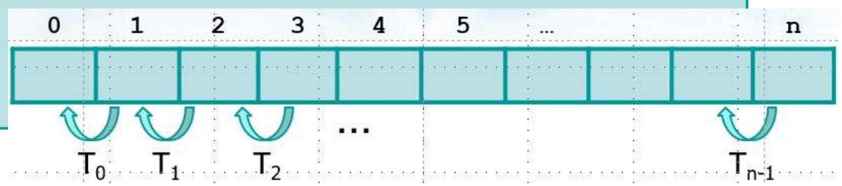
For this reason a second barrier is used

Restart (if necessary)

Block everything

The last thread to arrive unblock all

All the other threads wait on a barrier



Solution 3

- Can we use pthread_barrier_wait?
  - Yes, and one barrier should suffice for synchronization purposes
    - Unfortunately, we must also check if the array is sorted after the barrier
    - Thus, we need a second barrier anyway
  - In the second barrier, the last thread arriving checks the sorting and displays the array
    - It is convenient to implement it with a counter, a mutex, and a semaphore

Solution 3

Solution in C with one library barriers

```
pthread_barrier_wait (&barrier1);

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

Barrier #1

Barrier #2

As Solution 2, but …

… we must reset the counter back to zero