Synchronization

Exercises on Concurrent Programming

Stefano Quer and Stefano Scanzio
Dipartimento di Automatica e Informatica
Politecnico di Torino
skenz.it/os stefano.scanzio@polito.it

Exam Italian Course: 2016/02/17

Exercise

- Realize a synchronization schema in which
  - There are two sets of Readers (i.e., R1 and R2)
  - A set of Writers (i.e., W)
  - Each member of R1 (R2) can access the critical section together with other members of R1 (R2), however members of R1 and R2, or R1 and W, or R2 and W must access the critical section in mutual exclusion

Solution

- Semaphores and variables

```
n1 = n2 = 0;
init (s1, 1);
init (s2, 1);
```

```
init (busy, 1);
init (meW, 1);
```

- Writer

```
wait (busy);
...
writing
...
signal (busy);
```

```
wait (meW);
wait (busy);
writing
signal (busy);
signal (meW);
```

Solution

- Reader 1

```
wait (s1);
  n1++;
  if (n1==1)
    wait (busy);
signal (s1);
...
reading
...
wait (s1);
  n1--;
  if (n1==0)
    signal (busy);
signal (s1);
```

- Reader 2

```
wait (s2);
  n2++;
  if (n2==1)
    wait (busy);
signal (s2);
...
reading
...
wait (s2);
  n2--;
  if (n2==0)
    signal (busy);
signal (s2);
```

Exam Italian Course: 2016/06/29

Exercise

- You must manage
  - P producers
  - Two sets of consumers (i.e., C1 and C2)
  - Two queues Q1 and Q2, of length N1 and N2
- in that system
  - Each of the P producers produces an element in the queue Q1, and subsequently they produce an element each in the Q2 queue
  - Consumers C1 consume only elements from Q1
  - Consumers C2 consume only elements from Q2

Solution

- Semaphores and variables

```
init (full1, 0);
init (full2, 0);
init (empty1, N1);
init (empty2, N2);
init (MEp1, 1);
init (MEp2, 1);
init (MEc1, 1);
init (MEc2, 1);
```

Solution

- Producer

```
P() {
   item m;
   while (1) {
      m = produce ();
      wait (empty1);
      wait (MEp1);
      enqueue1 (m);
      signal (MEp1);
      signal (full1);
```

```
      m = produce ();
      wait (empty2);
      wait (MEp2);
      enqueue1 (m);
      signal (MEp2);
      signal (full2);
   }
}
```

Solution

- Consumer 1

```
C1() {
   item m;
   while (1) {
      wait (full1);
      wait (MEc1);
      m = dequeue1 ();
      signal (MEc1);
      signal (empty1);
      consume (m);
   }
}
```

- Consumer 2

```
C2() {
   item m;
   while (1) {
      wait (full2);
      wait (MEc2);
      m = dequeue2 ();
      signal (MEc2);
      signal (empty2);
      consume (m);
   }
}
```

Exam Italian Course: 2017/02/27

Exercise

- Given three processes, which use semaphores A-F (all initialize to 1)
- Is there an execution sequence of P1, P2 and P3 such that they can block for deadlock?
- Motivate your answer

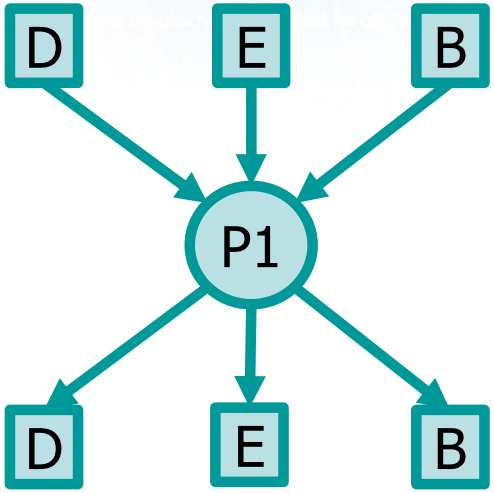
Exercise

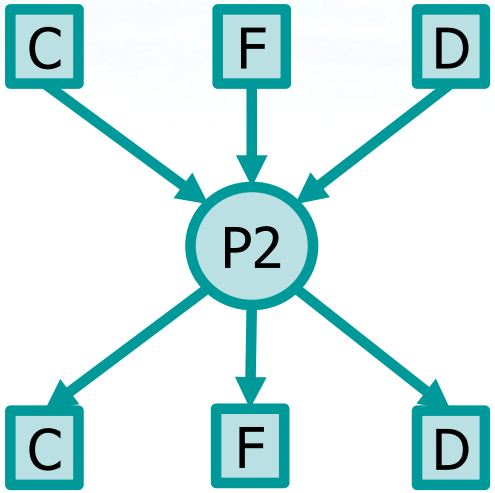
```
P1:
while (1) {
   wait (D);
   wait (E);
   wait (B);
   printf ("P1\n");
   signal (D);
   signal (E);
   signal (B);
}
```

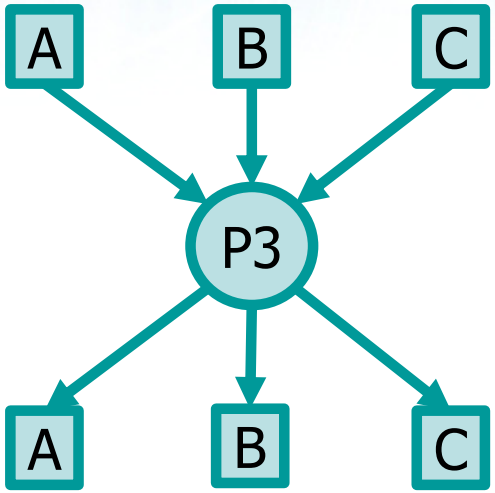
```
P2:
while (1) {
   wait (C);
   wait (F);
   wait (D);
   printf ("P2\n");
   signal (C);
   signal (F);
   signal (D);
}
```

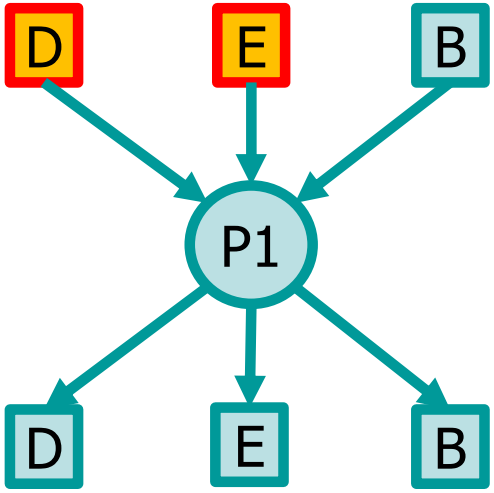
```
P3:
while (1) {
   wait (A);
   wait (B);
   wait (C);
   printf ("P3\n");
   signal (A);
   signal (B);
   signal (C);
}
```

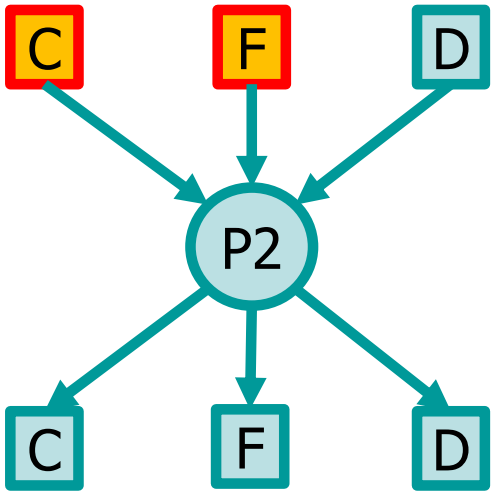
Soluzione

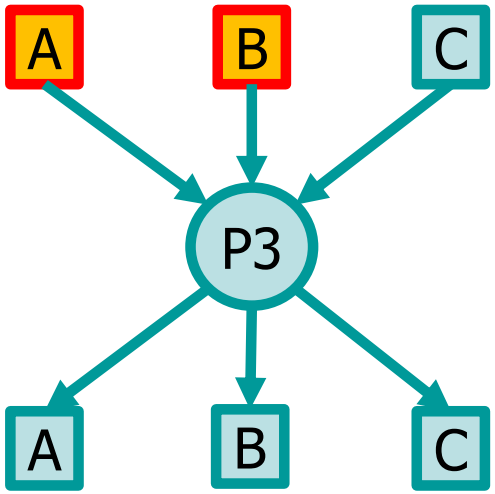












Exercise

- A file, of undefined length and in ASCII format, contains a list of integer numbers
- Write a program that, after receiving a value k (integer) and a string from command line, generates k threads and wait them
- Each thread
  - Reads concurrently the file, and performs the sum of the read integer numbers
  - When the end of file is reached, it must print the number of integer numbers it has read and the computed sum
  - Terminates

Exercise

- After all threads terminate, the main thread has to print the total number of integer numbers and the total sum
- Example

File format: file.txt

```
7
9
2
-4
15
0
3
```

Example of execution

```
> pgrm 2 file.txt
Thread 1: Sum=18 #Line=4
Thread 2: Sum=14 #Line=3
Total   : Sum=32 #Line=7
```

Solution

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <unistd.h>
#include <sys/types.h>
#include <semaphore.h>
#include <pthread.h>

#define L 100
struct threadData {
  pthread_t threadId;
  int id;
  FILE *fp;
  int line;
  int sum;
};
static void *readFile (void *);
sem_t sem;
```

Includes, variables and prototypes

It must be unique (but, or it is global, or it is passed as parameter to threads through this structure)

Solution

Main
Part 1

```
int main (int argc,char *argv[]) {
  int i, nT, total, line;
  struct threadData *td;
  void *retval;
  FILE *fp;

  nT = atoi (argv[1]);
  td = (struct threadData *) malloc
       (nT * sizeof (struct threadData));
  fp = fopen (argv[2], "r");
  if (td==NULL || fp==NULL) {
    fprintf (stderr, "Error ...\n");
    exit (1);
  }
  sem_init (&sem, 0, 1);
```

Not shared

Init to 1

Solution

Main Part 2

File pointer common to all the threads

```
  for (i=0; i<nT; i++) {
    td[i].id = i;
    td[i].fp = fp;
    td[i].line = td[i].sum = 0;
    pthread_create (&(td[i].threadId),
      NULL, readFile, (void *) &td[i]);
  }
  total = line = 0;
  for (i=0; i<nT; i++) {
    pthread_join (td[i].threadId, &retval);
    total += td[i].sum;
    line += td[i].line;
  }
  fprintf (stdout, "Total: Sum=%d #Line=%d\n",
    total, line);
  sem_destroy (&sem);
  fclose (fp);
  return (1);
}
```

Solution

Thread function

```
static void *readFile (void *arg){
  int n, retVal;
  struct threadData *td;
  td = (struct threadData *) arg;
  do {
    sem_wait (&sem);
    retVal = fscanf (td->fp, "%d", &n);
    sem_post (&sem);
    if (retVal!=EOF) {
      td->line++;
      td->sum += n;
    }
    sleep (1);  // Delay Threads
  } while (retVal!=EOF);
  fprintf (stdout, "Thread: %d Sum=%d #Line=%d\n",
    td->id, td->sum, td->line);
  pthread_exit ((void *) 1);
}
```

Mutual exclusion for file reading

Exercise

- A concurrent program must use the bubble sort algorithm as follow
  - A static vector contains n integer elements
  - Order it by running 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 synchronize
  - If
    - All the elements are correctly ordered, the program terminates
    - Otherwise all threads are run again to make a new series of exchanges

Solution 1

```
#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
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
Part 2

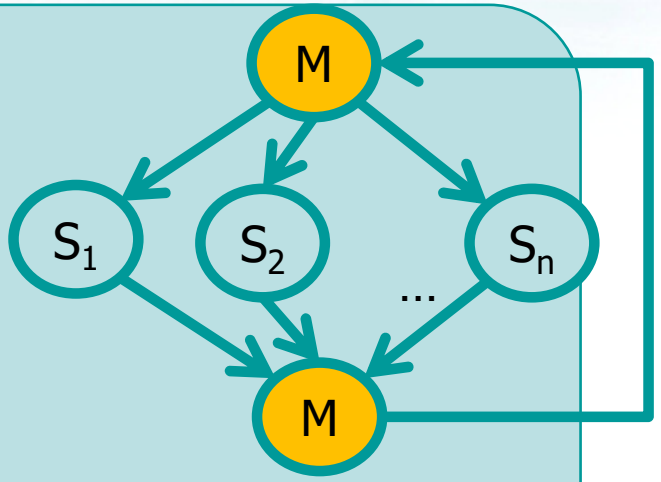
```
sem_init (&semMaster, 0, num_threads);
pthread_create (&thMaster, NULL, master, &num_threads);

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

Creation of 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);
}
```



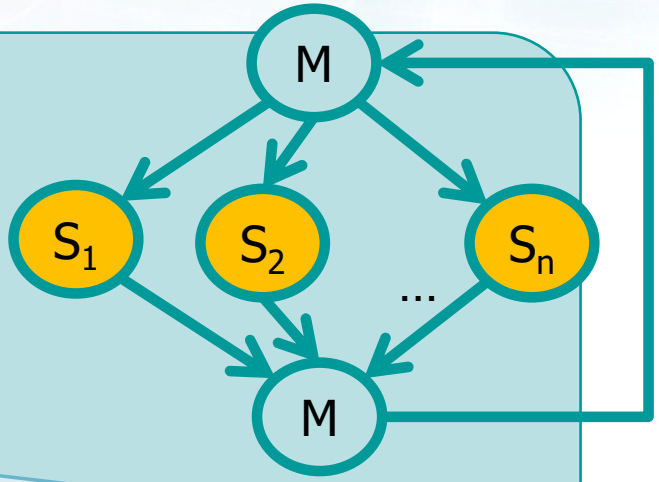
Wait slave threads

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

```
#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;
```

Avoid the use of a semaphore for each slave by using a barrier

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 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

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 to 1 if no thread makes an exchange

Release the access of the 2 elements of the vector

Solution 2

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

sem_wait (&barrier1);
```

Barrier #1

Before the iteration, you need to synchronize all the threads

The last thread to arrive unblock all

All the other threads wait on a barrier

Mutex to protect count

Solution 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;
}
```

Barrier #2

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