Sistemi in tempo reale
Semaphores

Giuseppe Lipari
http://retis.sssup.it/~lipari

Scuola Superiore Sant’Anna – Pisa

October 5, 2011
Outline

1. Semaphores
   - Mutual exclusion
   - Synchronization
   - Exercise
   - Producer / Consumer

2. Solutions
Outline

1. Semaphores
   - Mutual exclusion
   - Synchronization
   - Exercise
   - Producer / Consumer

2. Solutions
The semaphore mechanism was first proposed by Dijkstra. A semaphore is an abstract data type that consists of:
- a counter
- a blocking queue
- operation wait
- operation signal

The operations on a semaphore must be atomic:
- the OS makes them atomic by appropriate low-level mechanisms.
Semaphore definition

- semaphores are a basic mechanisms for providing synchronization
- it has been shown that every kind of synchronization and mutual exclusion can be implemented by using semaphores
- we will analyze possible implementation of the semaphore mechanism later

```cpp
class Semaphore {
  <blocked queue> blocked;
  int counter;
public:
  Semaphore (int n) : count (n) {...}
  void wait();
  void signal();
};
```
Wait and signal

- a *wait* operation has the following behavior:
Wait and signal

- a `wait` operation has the following behavior:
  - if `counter == 0`, the requiring thread is blocked;
Wait and signal

- a **wait** operation has the following behavior:
  - if `counter == 0`, the requiring thread is blocked;
  - it is removed from the ready queue and inserted in the blocked queue;
a **wait** operation has the following behavior:

- if `counter == 0`, the requiring thread is blocked;
  - it is removed from the ready queue and inserted in the blocked queue;
- if `counter > 0`, then `counter--`;
Wait and signal

- a wait operation has the following behavior:
  - if \( \text{counter} == 0 \), the requiring thread is blocked;
    - it is removed from the ready queue and inserted in the blocked queue;
  - if \( \text{counter} > 0 \), then \( \text{counter}-- \);

- a signal operation has the following behavior:
Wait and signal

- a **wait** operation has the following behavior:
  - if `counter == 0`, the requiring thread is blocked;
    - it is removed from the ready queue and inserted in the blocked queue;
  - if `counter > 0`, then `counter--`

- a **signal** operation has the following behavior:
  - if `counter == 0` and there is some blocked thread, unblock it;
a **wait** operation has the following behavior:

- if `counter == 0`, the requiring thread is blocked;
  - it is removed from the ready queue and inserted in the blocked queue;
- if `counter > 0`, then `counter--`;

a **signal** operation has the following behavior:

- if `counter == 0` and there is some blocked thread, unblock it;
  - the thread is removed from the blocked queue and inserted in the ready queue
a **wait** operation has the following behavior:

- if \( \text{counter} == 0 \), the requiring thread is blocked;
  - it is removed from the ready queue and inserted in the blocked queue;
- if \( \text{counter} > 0 \), then \( \text{counter}-- \);

a **signal** operation has the following behavior:

- if \( \text{counter} == 0 \) and there is some blocked thread, unblock it;
  - the thread is removed from the blocked queue and inserted in the ready queue
- otherwise, increment counter;
Pseudo-code for wait and signal

class Semaphore {
    <blocked queue> blocked;
    int counter;

public:
    Semaphore (int n) : counter (n) {...}
    void wait() {
        if (counter == 0)
            <block the thread>
        else
            counter--;
    }
    void signal() {
        if (<some blocked thread>)
            <unblock the thread>
        else
            counter++;
    }
};
Outline

1 Semaphores
   - Mutual exclusion
   - Synchronization
   - Exercise
   - Producer / Consumer

2 Solutions
Mutual exclusion with semaphores

To use a semaphore for mutual exclusions:
- define a semaphore initialized to 1
- before entering the critical section, perform a wait
- after leaving the critical section, perform a signal

```c
void *threadA(void *) {
    ...
    s.wait();
    <critical section>
    s.signal();
    ...
}

void *threadB(void *) {
    ...
    s.wait();
    <critical section>
    s.signal();
    ...
}
```
Mutual exclusion: example

Semaphore
Counter
1
Blocked queue

Ready queue
TB TA
Mutual exclusion: example

Semaphore

Counter
0

Blocked queue

Ready queue

example1.c

s.wait(); (TA)
Mutual exclusion: example

```
example1.c
s.wait(); (TA)
<critical section (1)> (TA)
```
Mutual exclusion: example

element

Semaphore
  Counter
    0
  Blocked queue

Ready queue
  TA  TB

element

example1.c
s.wait();  
<critical section (1)>  
s.wait();
Mutual exclusion: example

Semaphore
  Counter
    0
  Blocked queue
    TB

Ready queue
  TA

example1.c

s.wait(); (TA)
<critical section (1)> (TA)
s.wait(); (TB)
<critical section (2)> (TA)
Mutual exclusion: example

Semaphore
  Counter
    0
  Blocked queue
    TB

Ready queue
  TA

example1.c

s.wait();               (TA)
<critical section (1)> (TA)
s.wait();               (TB)
<critical section (2)> (TA)
s.signal();             (TA)
Mutual exclusion: example

Semaphore

Counter

Blocked queue

Ready queue

example1.c

```c
s.wait(); (TA)
<critical section (1)> (TA)
s.wait(); (TB)
<critical section (2)> (TA)
s.signal(); (TA)
<critical section> (TB)
```
Mutual exclusion: example

Semaphore
Counter
1
Blocked queue

Ready queue

example1.c

s.wait(); (TA)
<critical section (1)> (TA)
s.wait(); (TB)
<critical section (2)> (TA)
s.signal(); (TA)
<critical section> (TB)
s.signal(); (TB)
Outline

1 Semaphores
   - Mutual exclusion
   - Synchronization
   - Exercise
   - Producer / Consumer

2 Solutions
Synchronization with semaphores

- How to use a semaphore for synchronizing two or more threads
  - define a semaphore initialized to 0
  - at the synchronization point, the task to be blocked performs a wait
  - at the synchronization point, the other task performs a signal
- Example: thread A must block if it arrives at the synchronization point before thread B

```c
Semaphore s(0);

void *threadA(void *) {
    ...
    s.wait();
    ...
}

void *threadB(void *) {
    ...
    s.signal();
    ...
}
```
Problem 1

- How to make each thread wait for the other one?
  - The first one that arrives at the synchronization point waits for the other one.
Problem 1

- How to make each thread wait for the other one?
  - The first one that arrives at the synchronization point waits for the other one.
- Solution: use two semaphores!

```c
Semaphore sa(0), sb(0);

void *threadA(void *) {
    ...
    sa.signal();
    sb.wait();
    ...
}

void *threadB(void *) {
    ...
    sb.signal();
    sa.wait();
    ...
}
```
Outline

1. Semaphores
   - Mutual exclusion
   - Synchronization
   - Exercise
   - Producer / Consumer

2. Solutions
Problem 2

- Generalize the previous synchronization problem to N threads
  - The first N-1 threads must block waiting for the last one
Problem 2

- Generalize the previous synchronization problem to N threads
  - The first N-1 threads must block waiting for the last one
- First solution (more elegant)
Problem 2

- Generalize the previous synchronization problem to N threads
  - The first N-1 threads must block waiting for the last one
- First solution (more elegant)
- Second solution (more practical)
Outline

1. **Semaphores**
   - Mutual exclusion
   - Synchronization
   - Exercise
   - Producer / Consumer

2. **Solutions**
We now want to implement a mailbox with a circular array avoiding busy wait

- The producer must be blocked when the mailbox is full
- The consumer must be blocked when the mailbox is empty
- We use appropriate semaphores to block these threads

Initially we consider only one producer and one consumer
Implementation

circulararray1.c

```c
#define N 10

class CA {
    int array[N];
    int head(0);
    int tail(0);

    Semaphore empty(0);
    Semaphore full(N);

public:
    void insert(int elem);
    void extract(int &elem);
};

circulararray1.c

void CA::insert(int elem)
{
    full.wait();
    array[head++] = elem;
    head = head % N;
    empty.signal();
}

void CA::extract(int &elem)
{
    empty.wait();
    elem = array[tail++];
    tail = tail % N;
    full.signal();
}
```
Proof of correctness

- when the number of elements in the queue is between 1 and 9, there is no problem;
Proof of correctness

- when the number of elements in the queue is between 1 and 9, there is no problem;
  - insert and extract work on different variables (head and tail respectively) and different elements of the array;
Proof of correctness

- when the number of elements in the queue is between 1 and 9, there is no problem;
  - insert and extract work on different variables (head and tail respectively) and different elements of the array;
  - The value of full and empty is always greater than 0, so neither the producer nor the consumer can block;
Proof of correctness

- when the number of elements in the queue is between 1 and 9, there is no problem;
  - insert and extract work on different variables (head and tail respectively) and different elements of the array;
  - The value of full and empty is always greater than 0, so neither the producer nor the consumer can block;
- when there is no element in the queue, head = tail, counter of empty = 0, counter of full = N;
Proof of correctness

- when the number of elements in the queue is between 1 and 9, there is no problem;
  - insert and extract work on different variables (head and tail respectively) and different elements of the array;
  - The value of full and empty is always greater than 0, so neither the producer nor the consumer can block;
- when there is no element in the queue, head = tail, counter of empty = 0, counter of full = N;
  - If the extract begins before the end of an insert, it will be blocked
Proof of correctness

- when the number of elements in the queue is between 1 and 9, there is no problem;
  - insert and extract work on different variables (head and tail respectively) and different elements of the array;
  - The value of full and empty is always greater than 0, so neither the producer nor the consumer can block;

- when there is no element in the queue, head = tail, counter of empty = 0, counter of full = N;
  - If the extract begins before the end of an insert, it will be blocked
  - After an insert, there is an element in the queue, so we are in the previous case
Proof of correctness

- when the number of elements in the queue is between 1 and 9, there is no problem;
  - insert and extract work on different variables (head and tail respectively) and different elements of the array;
  - The value of full and empty is always greater than 0, so neither the producer nor the consumer can block;
- when there is no element in the queue, head = tail, counter of empty = 0, counter of full = N;
  - If the extract begins before the end of an insert, it will be blocked
  - After an insert, there is an element in the queue, so we are in the previous case
- For symmetry, the same holds for the case of N elements in the queue. Again, head = tail, counter of empty = N, counter of full = 0;
Proof of correctness

- when the number of elements in the queue is between 1 and 9, there is no problem;
  - insert and extract work on different variables (head and tail respectively) and different elements of the array;
  - The value of full and empty is always greater than 0, so neither the producer nor the consumer can block;
- when there is no element in the queue, head = tail, counter of empty = 0, counter of full = N;
  - If the extract begins before the end of an insert, it will be blocked
  - After an insert, there is an element in the queue, so we are in the previous case
- For symmetry, the same holds for the case of N elements in the queue. Again, head = tail, counter of empty = N, counter of full = 0;
  - If the insert begins before the end of an extract, it will be blocked
Proof of correctness

- when the number of elements in the queue is between 1 and 9, there is no problem;
  - insert and extract work on different variables (head and tail respectively) and different elements of the array;
  - The value of full and empty is always greater than 0, so neither the producer nor the consumer can block;
- when there is no element in the queue, head = tail, counter of empty = 0, counter of full = N;
  - If the extract begins before the end of an insert, it will be blocked
  - After an insert, there is an element in the queue, so we are in the previous case
- For symmetry, the same holds for the case of N elements in the queue. Again, head = tail, counter of empty = N, counter of full = 0;
  - If the insert begins before the end of an extract, it will be blocked
  - After an extract, we fall back in the previous case
Multiple producers/consumers

- Suppose now there are many producers and many consumers;
- all producers will act on the same variable head, and all consumers on tail;
- If one producer preempts another producer, an inconsistency can arise
  - Exercise: prove the above sentence
- Therefore, we need to combine synchronization and mutual exclusion
First solution

circulararray-wrong.c

```c
#define N 10

class CA {
    int array[N];
    int head(0);
    int tail(0);

    Semaphore empty(0);
    Semaphore full(N);
    Semaphore mutex(1);

public:
    void insert(int elem);
    void extract(int &elem);
};

circulararray-wrong.c

```
Wrong solution

- The previous solution is wrong!
- Counter example:
  - A consumer thread executes first, locks the mutex and blocks on the empty semaphore
  - All other threads (producers or consumers) will block on the mutex
- Lesson learned: never block inside a mutex!
Deadlock

Deadlock situation

- A thread executes `mutex.wait()` and then blocks on a synchronisation semaphore
- To be unblocked another thread must enter a critical section guarded by the same mutex semaphore
- So, the first thread cannot be unblocked and free the mutex
- The situation cannot be solved
Correct solution

circulararray-correct.c

```c
#define N 10

class CA {
    int array[N];
    int head(0);
    int tail(0);

    Semaphore empty(0);
    Semaphore full(N);
    Semaphore mutex(1);

public:
    void insert(int elem);
    void extract(int &elem);
};

void CA::insert(int elem) {
    full.wait();
    mutex.wait();
    array[head++] = elem;
    head = head % N;
    mutex.signal();
    empty.signal();
}

void CA::extract(int &elem) {
    empty.wait();
    mutex.wait();
    elem = array[tail++];
    tail = tail % N;
    mutex.signal();
    full.signal();
}
```

Exercises

- Solve the previous exercise with two mutex (one for the consumers and one for the producers)
  - Prove the solution is correct
- Suppose there are one producer and N consumer. Every message has to be received by each consumer.
  - Write the data structure, the insert and extract functions
  - Suppose that extract() takes an additional arguments that specifies the consumer ID (between 0 and N-1).
Outline

1. Semaphores
   - Mutual exclusion
   - Synchronization
   - Exercise
   - Producer / Consumer

2. Solutions
First solution to problem 2

Elegant solution. Uses many semaphores! (with the pthread interface)

prob2-solution1.c

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

#define N 8

sem_t s[N][N];

void init()
{
    int i, j;
    for (i=0; i<N; i++)
        for (j=0; j<N; j++)
            sem_init(&s[i][j], 0, 0);
}

void *thread(void *arg)
{
    int k = *((int *) arg); int j;
    printf("TH%d: before synch\n", k);
    for (j=0; j<N; j++)
        if (j!=k) sem_post(&s[k][j]);
    for (j=0; j<N; j++)
        if (j!=k) sem_wait(&s[j][k]);
    printf("TH%d: after synch\n", k);
    return 0;
}

int main()
{
    pthread_t tid[N];
    int i;
    int args[N];

    init();

    for (i=0; i<N; i++)
        args[i] = i;
        pthread_create(&tid[i], 0, thread, (void *)&args[i]);
}
```
Second solution to problem 2

Practical solution. We need a mutex semaphore, a counter, and a semaphore to block threads. (with the pthread interface)

solution2.c

```c
struct synch {
    int count;
    sem_t m; // mutex
    sem_t b; // blocked
    int N;   // number of threads
};

void initsynch(struct synch *s, int n) {
    int i;
    s->count = 0;
    sem_init(&s->m, 0, 1);
    sem_init(&s->b, 0, 0);
    s->N = n;
}

void my_synch(struct synch *s) {
    int i;
    sem_wait(&s->m);
    if (++s->count < s->N) {
        sem_post(&s->m);
        sem_wait(&s->b);
    } else {
        for (i=0; i < s->N - 1; i++)
            sem_post(&s->b);
        sem_post(&s->m);
    }

    struct synch sp;

    void *thread(void *arg) {
        ...
        mysynch(&sp);
        ...
    }
```