

Sistemi in tempo reale  
Anno accademico 2006 - 2007  
Concorrenza - II

Giuseppe Lipari  
<http://feanor.sssup.it/~lipari>

Scuola Superiore Sant'Anna

# Outline

- 1 Introduction to concurrency
- 2 Models of concurrency: shared memory
  - Critical Sections
  - Synchronization
- 3 Semaphores
- 4 Solutions

# The need for concurrency

There are many reason for concurrency

- functional
- performance
- expressive power

Functional

- many users may be connected to the same system at the same time
  - each user can have its own processes that execute concurrently with the processes of the other users
- perform many operations concurrently
  - for example, listen to music, write with a word processor, burn a CD, etc...
  - they are all different and independent activities
  - they can be done *at the same time*

# The need for concurrency (2)

## Performance

- take advantage of **blocking time**
  - while some thread waits for a blocking condition, another thread performs another operation
- parallelism in **multi-processor machines**
  - if we have a multi-processor machine, independent activities can be carried out on different processors at the same time

## Expressive power

- many control applications are inherently concurrent
- concurrency support helps in expressing concurrency, making application development simpler

# Concurrency model

- a system is a set of **concurrent activities**
  - they can be processes or threads
- they interact in two ways
  - they access the hardware resources (processor, disk, memory, etc.)
  - they exchange data
- these activities **compete** for the resources and/or **cooperate** for some common objective

# Resources

- a resource can be
  - a HW resource like a I/O device
  - a SW resource, i.e. a data structure
  - in both cases, access to a resource must be regulated to avoid interference
- example 1
  - if two processes want to print on the same printer, their access must be sequentialised, otherwise the two printing could be intermingled!
- example 2
  - if two threads access the same data structure, the operation on the data must be sequentialized otherwise the data could be inconsistent!

# Interaction model

Activities can interact according to two fundamental models

- shared memory
  - All activities access the same memory space
- message passing
  - All activities communicate each other by sending messages through OS primitives
- we will analyze both models in the following slides

# Outline

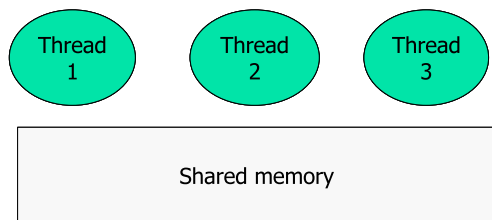
- 1 Introduction to concurrency
- 2 Models of concurrency: shared memory**
  - Critical Sections
  - Synchronization
- 3 Semaphores
- 4 Solutions



# Shared memory

## Shared memory communication

- it was the first one to be supported in old OS
- it is the simplest one and the closest to the machine
- all threads can access the same memory locations



# Mutual Exclusion Problem

- We do not know in advance the relative speed of the processes
  - hence, we do not know the order of execution of the hardware instructions
- Recall the example of incrementing variable  $x$ 
  - incrementing  $x$  is not an atomic operation
  - atomic behavior can be obtained using interrupt disabling or special atomic instructions

# Example 1

```
/* Shared memory */  
int x;
```

```
void *threadA(void *)  
{  
    ...;  
    x = x + 1;  
    ...;  
}
```

```
void *threadB(void *)  
{  
    ...;  
    x = x + 1;  
    ...;  
}
```

## ● Bad Interleaving:

```
...  
LD    R0, x    (TA)  x = 0  
LD    R0, x    (TB)  x = 0  
INC   R0       (TB)  x = 0  
ST    x, R0    (TB)  x = 1  
INC   R0       (TA)  x = 1  
ST    x, R0    (TA)  x = 1  
...
```

## Example 2

```
// Shared object (sw resource)
class A {
    int a;
    int b;
public:
    A() : a(1), b(1) {};
    void inc() {
        a = a + 1; b = b + 1;
    }
    void mult() {
        b = b * 2; a = a * 2;
    }
} obj;
```

```
void * threadA(void *)
{
    ...
    obj.inc();
    ...
}
```

```
void * threadB(void *)
{
    ...
    obj.mult();
    ...
}
```

## Example 2

```
// Shared object (sw resource)
class A {
    int a;
    int b;
public:
    A() : a(1), b(1) {};
    void inc() {
        a = a + 1; b = b + 1;
    }
    void mult() {
        b = b * 2; a = a * 2;
    }
} obj;
```

*Consistency:*  
After each operation,  $a == b$

```
void * threadA(void *)
{
    ...
    obj.inc();
    ...
}
```

```
void * threadB(void *)
{
    ...
    obj.mult();
    ...
}
```

## Example 2

```
// Shared object (sw resource)
class A {
    int a;
    int b;
public:
    A() : a(1), b(1) {};
    void inc() {
        a = a + 1; b = b + 1;
    }
    void mult() {
        b = b * 2; a = a * 2;
    }
} obj;
```

*Consistency:*

After each operation,  $a == b$

```
void * threadA(void *)
{
    ...
    obj.inc();
    ...
}
```

```
void * threadB(void *)
{
    ...
    obj.mult();
    ...
}
```

```
a = a + 1;    TA    a = 2
b = b * 2;    TB    b = 2
b = b + 1;    TA    b = 3
a = a * 2;    TB    a = 4
```

## Example 2

```
// Shared object (sw resource)
class A {
    int a;
    int b;
public:
    A() : a(1), b(1) {};
    void inc() {
        a = a + 1; b = b + 1;
    }
    void mult() {
        b = b * 2; a = a * 2;
    }
} obj;
```

*Consistency:*  
After each operation,  $a == b$

```
void * threadA(void *)
{
    ...
    obj.inc();
    ...
}
```

```
void * threadB(void *)
{
    ...
    obj.mult();
    ...
}
```

Resource in a non-consistent state!!

```
a = a + 1;    TA    a = 2
b = b * 2;    TB    b = 2
b = b + 1;    TA    b = 3
a = a * 2;    TB    a = 4
```

# Consistency

- for any resource, we can state a set of **consistency properties**
  - a consistency property  $C_i$  is a boolean expression on the values of the **internal variables**
  - a consistency property must hold before and after each operation
  - it does not need to hold during an operation
  - if the operations are properly sequentialized, the consistency properties will always hold
- formal verification
  - let  $R$  be a resource, and let  $C(R)$  be a set of consistency properties on the resource
  - $C(R) = \{C_i\}$
  - A concurrent program is correct if, for every possible interleaving of the operations on the resource,  $\forall C_i \in C(R)$ ,  $C_i$  holds.



# Example: Circular Array

Implementation of a FIFO queue.

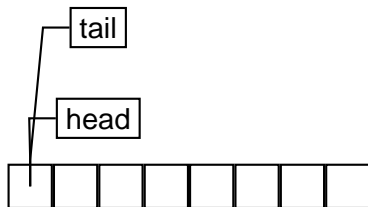
```
struct CA {
    int array[10];
    int head, tail, num;
}

void init(struct CA *ca) {
    ca->head=0; ca->tail=0;
    ca->num=0;
}

boolean insert(struct CA *ca, int elem) {
    if (ca->num == 10) return false;
    ca->array[ca->head] = elem;
    ca->head = (ca->head + 1) % 10;
    ca->num ++;
    return true;
}

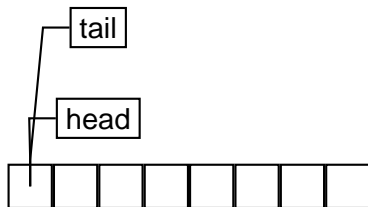
boolean extract(struct CA *ca, int *elem) {
    if (ca->num == 0) return false;
    *elem = ca->array[ca->tail];
    ca->tail = (ca->tail + 1) % 10;
    ca->num--;
    return true;
}
```

## Example: empty queue



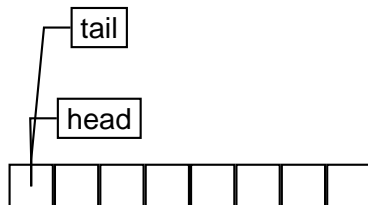
- head: index of the first free element in the queue
  - here will be inserted the next element

## Example: empty queue



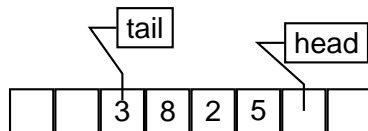
- head: index of the first free element in the queue
  - here will be inserted the next element
- tail: index of the first occupied element in the queue
  - will be the one that will be extracted next time

## Example: empty queue



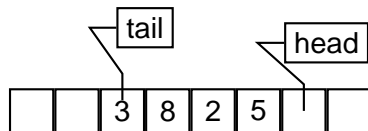
- head: index of the first free element in the queue
  - here will be inserted the next element
- tail: index of the first occupied element in the queue
  - will be the one that will be extracted next time
- the queue is empty, hence  $\text{head} == \text{tail}$

## Example: insert



```
boolean insert(struct CA *ca,
              int elem)
{
    if (ca->num == 10)
        return false;
    ca->array[ca->head] = elem;
    ca->head = (ca->head+1)%10;
    ca->num++;
    return true;
}
```

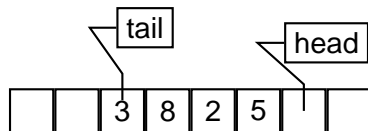
## Example: insert



```
boolean insert(struct CA *ca,
              int elem)
{
    if (ca->num == 10)
        return false;
    ca->array[ca->head] = elem;
    ca->head = (ca->head+1)%10;
    ca->num++;
    return true;
}
```

- $num = (head - tail) \% 8 \rightarrow num = 4;$

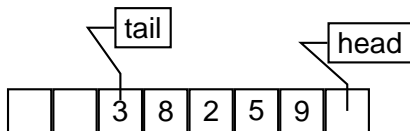
## Example: insert



```
boolean insert(struct CA *ca,
              int elem)
{
    if (ca->num == 10)
        return false;
    ca->array[ca->head] = elem;
    ca->head = (ca->head+1)%10;
    ca->num++;
    return true;
}
```

- $num = (head - tail) \% 8 \rightarrow num = 4;$
- `insert(ca, 9);`

## Example: insert

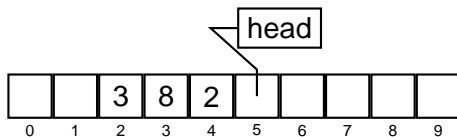


```
boolean insert(struct CA *ca,
              int elem)
{
    if (ca->num == 10)
        return false;
    ca->array[ca->head] = elem;
    ca->head = (ca->head+1)%10;
    ca->num++;
    return true;
}
```

- $num = (head - tail) \% 8 \rightarrow num = 4;$
- `insert(ca, 9);`
- head and num have been increased

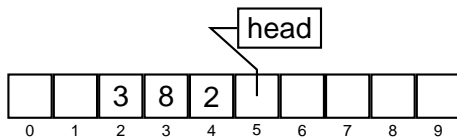


## Example: concurrent insert



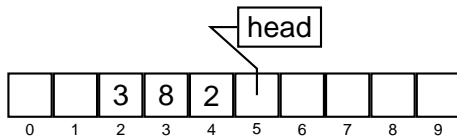
- Two threads, the first calls `insert(9)`, the second calls `insert(4)`;

## Example: concurrent insert



- Two threads, the first calls `insert(9)`, the second calls `insert(4)`;
- thread 1 calls `insert(ca, 9)`;

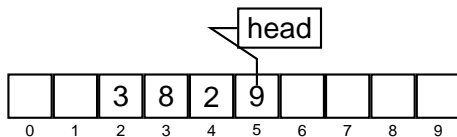
## Example: concurrent insert



```
boolean insert(struct CA *ca,  
              int elem)  
{  
    if (ca->num == 10)  
        return false;  
    ca->array[ca->head] = elem;
```

- Two threads, the first calls insert(9), the second calls insert(4);
- thread 1 calls insert(ca, 9);

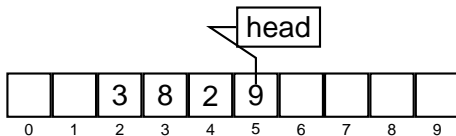
## Example: concurrent insert



```
boolean insert(struct CA *ca,  
              int elem)  
{  
    if (ca->num == 10)  
        return false;  
    ca->array[ca->head] = elem;
```

- Two threads, the first calls insert(9), the second calls insert(4);
- thread 1 calls insert(ca, 9);

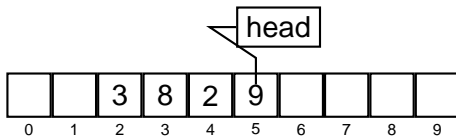
## Example: concurrent insert



```
boolean insert(struct CA *ca,  
              int elem)  
{  
    if (ca->num == 10)  
        return false;  
    ca->array[ca->head] = elem;
```

- Two threads, the first calls insert(9), the second calls insert(4);
- thread 1 calls insert(ca, 9);
- preemption by second thread

## Example: concurrent insert

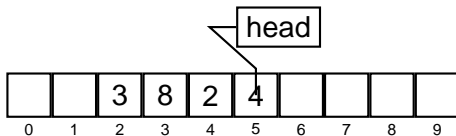


- Two threads, the first calls insert(9), the second calls insert(4);
- thread 1 calls insert(ca, 9);
- preemption by second thread

```
boolean insert(struct CA *ca,
               int elem)
{
    if (ca->num == 10)
        return false;
    ca->array[ca->head] = elem;
```

```
...
boolean insert(struct CA *ca,
               int elem)
{
    if (ca->num == 10)
        return false;
    ca->array[ca->head] = elem;
    ca->head = (ca->head+1)%10;
    ca->num++;
    return true;
}
...
```

## Example: concurrent insert

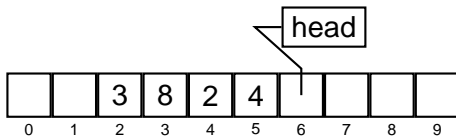


- Two threads, the first calls insert(9), the second calls insert(4);
- thread 1 calls insert(ca, 9);
- preemption by second thread

```
boolean insert(struct CA *ca,
               int elem)
{
    if (ca->num == 10)
        return false;
    ca->array[ca->head] = elem;
```

```
...
boolean insert(struct CA *ca,
               int elem)
{
    if (ca->num == 10)
        return false;
    ca->array[ca->head] = elem;
    ca->head = (ca->head+1)%10;
    ca->num++;
    return true;
}
...
```

## Example: concurrent insert



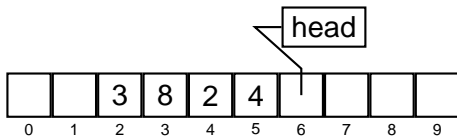
- Two threads, the first calls insert(9), the second calls insert(4);
- thread 1 calls insert(ca, 9);
- preemption by second thread

```
boolean insert(struct CA *ca,
               int elem)
{
    if (ca->num == 10)
        return false;
    ca->array[ca->head] = elem;
```

```
...
boolean insert(struct CA *ca,
               int elem)
{
    if (ca->num == 10)
        return false;
    ca->array[ca->head] = elem;
    ca->head = (ca->head+1)%10;
    ca->num++;
    return true;
}
...
```



## Example: concurrent insert

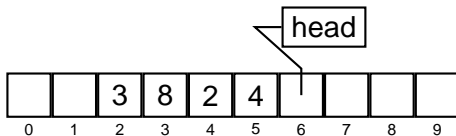


- Two threads, the first calls insert(9), the second calls insert(4);
- thread 1 calls insert(ca, 9);
- preemption by second thread
- second thread completes

```
boolean insert(struct CA *ca,
               int elem)
{
    if (ca->num == 10)
        return false;
    ca->array[ca->head] = elem;
```

```
...
boolean insert(struct CA *ca,
               int elem)
{
    if (ca->num == 10)
        return false;
    ca->array[ca->head] = elem;
    ca->head = (ca->head+1)%10;
    ca->num++;
    return true;
}
...
```

# Example: concurrent insert



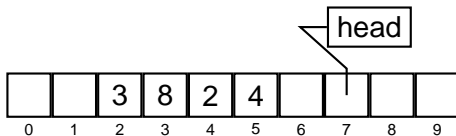
- Two threads, the first calls insert(9), the second calls insert(4);
- thread 1 calls insert(ca, 9);
- preemption by second thread
- second thread completes

```
boolean insert(struct CA *ca,
               int elem)
{
    if (ca->num == 10)
        return false;
    ca->array[ca->head] = elem;
```

```
...
boolean insert(struct CA *ca,
               int elem)
{
    if (ca->num == 10)
        return false;
    ca->array[ca->head] = elem;
    ca->head = (ca->head+1)%10;
    ca->num++;
    return true;
}
...
```

```
ca->head = (ca->head+1)%10;
ca->num++;
return true;
}
```

## Example: concurrent insert



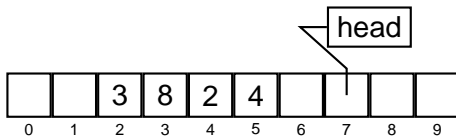
- Two threads, the first calls insert(9), the second calls insert(4);
- thread 1 calls insert(ca, 9);
- preemption by second thread
- second thread completes

```
boolean insert(struct CA *ca,
               int elem)
{
    if (ca->num == 10)
        return false;
    ca->array[ca->head] = elem;
```

```
...
boolean insert(struct CA *ca,
               int elem)
{
    if (ca->num == 10)
        return false;
    ca->array[ca->head] = elem;
    ca->head = (ca->head+1)%10;
    ca->num++;
    return true;
}
...
```

```
ca->head = (ca->head+1)%10;
ca->num++;
return true;
}
```

## Example: concurrent insert



- Two threads, the first calls `insert(9)`, the second calls `insert(4)`;
- thread 1 calls `insert(ca, 9)`;
- preemption by second thread
- second thread completes
- there is a **hole**! At some point, the extract will read a 4 and a random value, instead of a 9 and a 4.

```
boolean insert(struct CA *ca,
               int elem)
{
    if (ca->num == 10)
        return false;
    ca->array[ca->head] = elem;
```

```
...
boolean insert(struct CA *ca,
               int elem)
{
    if (ca->num == 10)
        return false;
    ca->array[ca->head] = elem;
    ca->head = (ca->head+1)%10;
    ca->num++;
    return true;
}
...
```

```
ca->head = (ca->head+1)%10;
ca->num++;
return true;
}
```

# Consistency properties for struct CA

- 1 when the queue is empty, or when the queue is full,  
`head == tail`
- 2 `num` is equal to the number of times `insert` has been called minus the number of times that `extract` has been called
- 3 ...
- 4 if element  $x$  has been inserted, eventually it must be extracted with an appropriate number of extracts
- 5 Every element that is extracted, has been inserted sometime in the past.

Last two can also be expressed as:

- Let  $(x_1, x_2, \dots, x_k)$  be the sequence of inserted elements, and let  $(y_1, y_2, \dots, y_k)$  be the sequence of extracted elements;
- then  $\forall i = 1, \dots, k \quad y_i = x_i$

# Correctness of Circular Array implementation

- The previous program is not correct, as the last property is not verified
  - the sequence of elements extracted does not correspond to the sequence of elements inserted
  - The problem is that the first thread was preempted while updating the data structure in a critical point.
  - we must prevent thread 2 from accessing the data structure while another thread is completing an operation on it
- Proving non-correctness is *easy*, in the sense that we must find a counterexample
- Proving correctness is a very complex task!
  - it is necessary to prove the correctness for every possible interleaving of every operation, for every possible input data, and for every possible internal state

# Insert and Extract

Let's assume that increments and decrements are atomic operations

- Producer: thread that inserts elements
- Consumer: thread that extracts elements
- It can be proved that interleaving exactly one producer and one consumer does not bring any problem
  - proof: if  $0 < num < 10$ , `insert()` and `extract()` are independent
  - if `num==0`
    - if `extract()` begins before `insert`, it immediately returns false,
    - if `insert` begins before, `extract` will still return false, so it cannot interfere with `insert`
  - same thing when `num==10`
- correctness is guaranteed for one consumer and one producer.

## Insert and Extract - II

- What happens if we exchange the sequence of instructions in insert?

```
boolean insert(struct CA *ca,
              int elem)
{
    if (ca->num == 10)
        return false;
    ca->num++;
    ca->array[ca->head] = elem;
    ca->head = (ca->head+1)%10;
    return true;
}
```



## Insert and Extract - II

- What happens if we exchange the sequence of instructions in insert?

```
boolean insert(struct CA *ca,
               int elem)
{
    if (ca->num == 10)
        return false;
    ca->num++;
    ca->array[ca->head] = elem;
    ca->head = (ca->head+1)%10;
    return true;
}
```

- It is easy to prove that in this case `insert()` cannot be interleaved with `extract`

# Outline

- 1 Introduction to concurrency
- 2 Models of concurrency: shared memory
  - Critical Sections
  - Synchronization
- 3 Semaphores
- 4 Solutions

# Critical sections

- the shared object where the conflict may happen is a **resource**

# Critical sections

- the shared object where the conflict may happen is a **resource**
- the parts of the code where the problem may happen are called **critical sections**

# Critical sections

- the shared object where the conflict may happen is a **resource**
- the parts of the code where the problem may happen are called **critical sections**
- a critical section is a sequence of operations that cannot be interleaved with other operations on the same resource

# Critical sections

- the shared object where the conflict may happen is a **resource**
- the parts of the code where the problem may happen are called **critical sections**
- a critical section is a sequence of operations that cannot be interleaved with other operations on the same resource
- two critical sections on the same resource must be properly sequentialized

# Critical sections

- the shared object where the conflict may happen is a **resource**
- the parts of the code where the problem may happen are called **critical sections**
- a critical section is a sequence of operations that cannot be interleaved with other operations on the same resource
- two critical sections on the same resource must be properly sequentialized
- we say that two critical sections on the same resource must execute in **MUTUAL EXCLUSION**

# Critical sections

- the shared object where the conflict may happen is a **resource**
- the parts of the code where the problem may happen are called **critical sections**
- a critical section is a sequence of operations that cannot be interleaved with other operations on the same resource
- two critical sections on the same resource must be properly sequentialized
- we say that two critical sections on the same resource must execute in **MUTUAL EXCLUSION**
- there are three ways to obtain mutual exclusion



# Critical sections

- the shared object where the conflict may happen is a **resource**
- the parts of the code where the problem may happen are called **critical sections**
- a critical section is a sequence of operations that cannot be interleaved with other operations on the same resource
- two critical sections on the same resource must be properly sequentialized
- we say that two critical sections on the same resource must execute in **MUTUAL EXCLUSION**
- there are three ways to obtain mutual exclusion
  - implementing the critical section as an atomic operation

# Critical sections

- the shared object where the conflict may happen is a **resource**
- the parts of the code where the problem may happen are called **critical sections**
- a critical section is a sequence of operations that cannot be interleaved with other operations on the same resource
- two critical sections on the same resource must be properly sequentialized
- we say that two critical sections on the same resource must execute in **MUTUAL EXCLUSION**
- there are three ways to obtain mutual exclusion
  - implementing the critical section as an atomic operation
  - disabling the preemption (system-wide)

# Critical sections

- the shared object where the conflict may happen is a **resource**
- the parts of the code where the problem may happen are called **critical sections**
- a critical section is a sequence of operations that cannot be interleaved with other operations on the same resource
- two critical sections on the same resource must be properly sequentialized
- we say that two critical sections on the same resource must execute in **MUTUAL EXCLUSION**
- there are three ways to obtain mutual exclusion
  - implementing the critical section as an atomic operation
  - disabling the preemption (system-wide)
  - selectively disabling the preemption (using semaphores and mutex)

# Implementing atomic operations

- In single processor systems
  - disable interrupts during a critical section
  - non-voluntary context switch is disabled!

```
CLI;  
<critical section>  
STI;
```

# Implementing atomic operations

- In single processor systems
  - disable interrupts during a critical section
  - non-voluntary context switch is disabled!
- Limitations:
  - if the critical section is long, no interrupt can arrive during the critical section
    - consider a timer interrupt that arrives every 1 msec.
    - if a critical section lasts for more than 1 msec, a timer interrupt could be lost
    - It must be done only for very short critical section;
  - Non voluntary context switch is disabled during the critical section
    - Disabling interrupts is a very low level solution: it is not possible in user space.

```
CLI;  
<critical section>  
STI;
```

# Atomic operations on multiprocessors

- Disabling interrupts is not sufficient
  - disabling interrupts on one processor lets a thread on another processor free to access the resource
- Solution: use `lock()` and `unlock()` operations
  - define a flag `s` for each resource, and then surround a critical section with `lock(s)` and `unlock(s)`;

```
int s;  
...  
lock(s);  
<critical section>  
unlock(s);  
...
```

# Disabling preemption

- On single processor systems
  - in some scheduler, it is possible to disable preemption for a limited interval of time
- problems:
  - if a high priority critical thread needs to execute, it cannot make preemption and it is delayed
  - even if the high priority task does not access the resource!

```
disable_preemption();  
<critical section>  
enable_preemption();
```

no context switch may happen during the critical section, but interrupts are enabled

# Critical sections: a general approach

- General techniques exist to protect critical sections
  - Semaphores
  - Mutex
- Properties:
  - Interrupts always enabled
  - Preemption always enabled
- Basic idea:
  - if a thread is inside a critical section on a given resource
  - all other threads are **blocked** upon entrance on a critical section on the same resource
- We will study such techniques in the following

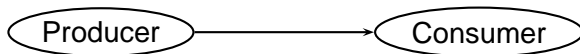


# Outline

- 1 Introduction to concurrency
- 2 Models of concurrency: shared memory**
  - Critical Sections
  - Synchronization**
- 3 Semaphores
- 4 Solutions

# Producer / Consumer model

- mutual exclusion is not the only problem
  - we need a way of synchronise two or more threads
- example: producer/consumer
  - suppose we have two threads,
  - one produces some integers and sends them to another thread (PRODUCER)
  - another one takes the integer and elaborates it (CONSUMER)



# Implementation with the circular array

- Suppose that the two threads have different speeds
  - for example, the producer is much faster than the consumer
  - we need to store the temporary results of the producer in some memory buffer
  - for our example, we will use the circular array structure

# Producer/Consumer implementation

```
struct CA qu;
```

```
void *producer(void *)
{
    bool res;
    int data;
    while(1) {
        <obtain data>
        while (!insert(&qu, data));
    }
}
```

```
void *consumer(void *)
{
    bool res;
    int data;
    while(1) {
        while (!extract(&qu, &data));
        <use data>
    }
}
```

- Problem with this approach:
  - if the queue is full, the producer waits *actively*
  - if the queue is empty, the consumer waits *actively*

# A more general approach

- we need to provide a general mechanism for synchronisation and mutual exclusion
- requirements
  - provide mutual exclusion between critical sections
    - avoid two interleaved insert operations
    - (semaphores, mutexes)
  - synchronise two threads on one condition
    - for example, block the producer when the queue is full
    - (semaphores, condition variables)

# Outline

- 1 Introduction to concurrency
- 2 Models of concurrency: shared memory
  - Critical Sections
  - Synchronization
- 3 Semaphores**
- 4 Solutions

# A general mechanism for blocking tasks

- 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

```
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;

# 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--`;

# 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:

# 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;

# 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;
    - the thread is removed from the blocked queue and inserted in the ready queue

# 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;
    - 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++;
    }
};
```

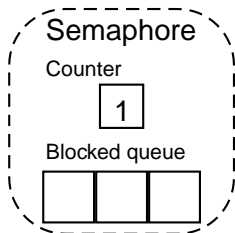
# Mutual exclusion with semaphores

- 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

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

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

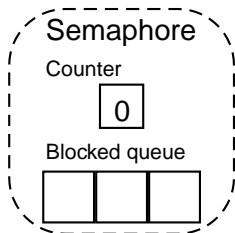
# Mutual exclusion: example



Ready queue



# Mutual exclusion: example



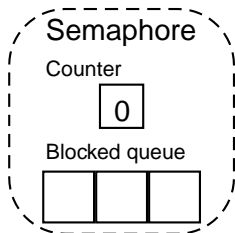
```
s.wait();
```

(TA)

Ready queue



# Mutual exclusion: example

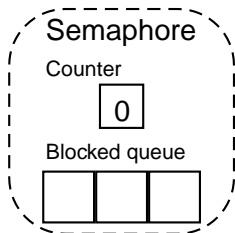


```
s.wait();           (TA)
<critical section (1)> (TA)
```

Ready queue



# Mutual exclusion: example

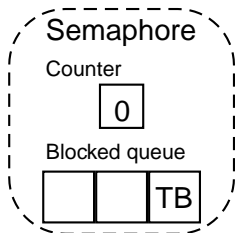


```
s.wait();           (TA)
<critical section (1)> (TA)
s.wait();           (TB)
```

Ready queue



# Mutual exclusion: example

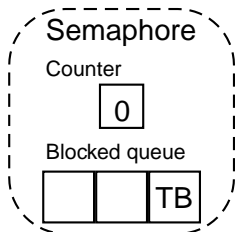


```
s.wait();           (TA)
<critical section (1)> (TA)
s.wait();           (TB)
<critical section (2)> (TA)
```

Ready queue



# Mutual exclusion: example



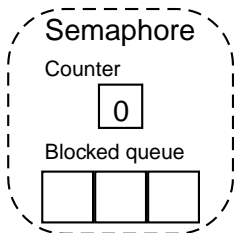
```
s.wait();           (TA)
<critical section (1)> (TA)
s.wait();           (TB)
<critical section (2)> (TA)
s.signal();         (TA)
```

Ready queue





# Mutual exclusion: example

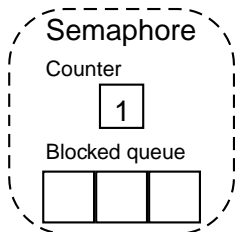


```
s.wait();           (TA)
<critical section (1)> (TA)
s.wait();           (TB)
<critical section (2)> (TA)
s.signal();         (TA)
<critical section>  (TB)
```

Ready queue



# Mutual exclusion: example



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

Ready queue



# 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 synch point before thread B

```
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!

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

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

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

# Semaphores in POSIX

```
sem_t sema;  
  
int sem_init(sem_t *s, int flag, int count);  
int sem_wait(sem_t *s);  
int sem_trywait(sem_t *s);  
int sem_post(sem_t *s);
```

# Semaphores in POSIX

```
sem_t sema;  
  
int sem_init(sem_t *s, int flag, int count);  
int sem_wait(sem_t *s);  
int sem_trywait(sem_t *s);  
int sem_post(sem_t *s);
```

`sem_t` is the semaphore type; it is an “opaque” C structure

# Semaphores in POSIX

```
sem_t sema;  
  
int sem_init(sem_t *s, int flag, int count);  
int sem_wait(sem_t *s);  
int sem_trywait(sem_t *s);  
int sem_post(sem_t *s);
```

`sem_t` is the semaphore type; it is an “opaque” C structure

`sem_init` initializes the semaphore; if `flag = 0`, the semaphore is local to the process; if `flag = 1`, the semaphore is shared with other processes; `count` is the initial value of the counter



# Semaphores in POSIX

```
sem_t sema;  
  
int sem_init(sem_t *s, int flag, int count);  
int sem_wait(sem_t *s);  
int sem_trywait(sem_t *s);  
int sem_post(sem_t *s);
```

`sem_t` is the semaphore type; it is an “opaque” C structure

`sem_init` initializes the semaphore; if `flag = 0`, the semaphore is local to the process; if `flag = 1`, the semaphore is shared with other processes; `count` is the initial value of the counter

`sem_wait` is the normal wait operation;

# Semaphores in POSIX

```
sem_t sema;  
  
int sem_init(sem_t *s, int flag, int count);  
int sem_wait(sem_t *s);  
int sem_trywait(sem_t *s);  
int sem_post(sem_t *s);
```

`sem_t` is the semaphore type; it is an “opaque” C structure

`sem_init` initializes the semaphore; if `flag = 0`, the semaphore is local to the process; if `flag = 1`, the semaphore is shared with other processes; `count` is the initial value of the counter

`sem_wait` is the normal wait operation;

`sem_trywait` does not block the task, but returns with error ( $< 0$ ) if the semaphore counter is 0.

# Semaphores in POSIX

```
sem_t sema;  
  
int sem_init(sem_t *s, int flag, int count);  
int sem_wait(sem_t *s);  
int sem_trywait(sem_t *s);  
int sem_post(sem_t *s);
```

`sem_t` is the semaphore type; it is an “opaque” C structure

`sem_post` is the normal signal operation.

`sem_init` initializes the semaphore; if `flag = 0`, the semaphore is local to the process; if `flag = 1`, the semaphore is shared with other processes; `count` is the initial value of the counter

`sem_wait` is the normal wait operation;

`sem_trywait` does not block the task, but returns with error ( $< 0$ ) if the semaphore counter is 0.

## 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)

# Producer / Consumer

- 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

```
#define N 10

struct CA {
    int array[N];
    int head, tail;
    sem_t empty;
    sem_t full;
} queue;

void init_ca(struct CA *q)
{
    q->head = q->tail = 0;
    sem_init(&q->empty, 0, 0);
    sem_init(&q->full, 0, N);
}
```

```
void insert(struct CA *q,
            int elem)
{
    sem_wait(&q->full);
    q->array[q->head++] = elem;
    q->head = q->head % N;
    sem_post(&q->empty);
}

void extract(struct CA *q,
             int &elem)
{
    sem_wait(&q->empty);
    *elem = q->array[q->tail++];
    q->tail = q->tail % N;
    sem_post(&q->full);
}
```



## 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

```
#define N 10

struct CA {
    int array[N];
    int head, tail;
    sem_t empty;
    sem_t full;
    sem_t m;
} queue;

void init_ca(struct CA *q)
{
    q->head = q->tail = 0;
    sem_init(&q->empty, 0, 0);
    sem_init(&q->full, 0, N);
    sem_init(&q->m, 0, 1);
}
```

```
void insert(struct CA *q,
            int elem)
{
    sem_wait(&q->m);
    sem_wait(&q->full);
    q->array[q->head++] = elem;
    q->head = q->head % N;
    sem_post(&q->empty);
    sem_post(&q->m);
}

void extract(struct CA *q,
             int &elem)
{
    sem_wait(&q->m);
    sem_wait(&q->empty);
    *elem = q->array[q->tail++];
    q->tail = q->tail % N;
    sem_post(&q->full);
    sem_post(&q->m);
}
```

# 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!

## Correct solution

```
#define N 10

struct CA {
    int array[N];
    int head, tail;
    sem_t empty;
    sem_t full;
    sem_t m;
} queue;

void init_ca(struct CA *q)
{
    q->head = q->tail = 0;
    sem_init(&q->empty, 0, 0);
    sem_init(&q->full, 0, N);
    sem_init(&q->m, 0, 1);
}
```

```
void insert(struct CA *q,
            int elem)
{
    sem_wait(&q->full);
    sem_wait(&q->m);
    q->array[q->head++] = elem;
    q->head = q->head % N;
    sem_post(&q->m);
    sem_post(&q->empty);
}

void extract(struct CA *q,
             int &elem)
{
    sem_wait(&q->empty);
    sem_wait(&q->m);
    *elem = q->array[q->tail++];
    q->tail = q->tail % N;
    sem_post(&q->m);
    sem_post(&q->full);
}
```

# 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  $N-1$ ).

# Internal implementation of semaphores

- `wait()` and `signal()` involve a possible thread-switch
- therefore they must be implemented as system calls!
- one blocked thread must be removed from state **RUNNING** and be moved in the semaphore blocking queue
- a semaphore is itself a **shared resource**
- `wait()` and `signal()` are critical sections!
- they must run with interrupt disabled and by using `lock()` and `unlock()` primitives

# Semaphore implementation: pseudo-code

```
void sem_wait()
{
    spin_lock_irqsave();
    if (counter==0) {
        <block the thread>
        schedule();
    } else counter--;
    spin_lock_irqrestore();
}

void sem_post()
{
    spin_lock_irqsave();
    if (counter== 0) {
        <unblock a thread>
        schedule();
    } else counter++;
    spin_lock_irqrestore();
}
```



# Outline

- 1 Introduction to concurrency
- 2 Models of concurrency: shared memory
  - Critical Sections
  - Synchronization
- 3 Semaphores
- 4 Solutions

# First solution to problem 2

Elegant solution. Uses many semaphores!

```
#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);
}
```

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

    init();

    for (i=0; i<N; i++)
        pthread_create(&tid[i], 0, thread,
                      (void *)i);
    for (i=0; i<N; i++)
        pthread_join(tid[i], 0);

    printf("Main: exiting\n");
}
```

## Second solution to problem 2

Practical solution. We need a mutex semaphore, a counter, and a semaphore to block threads.

```
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);
    ...
}
```