Laurea Specialistica in Ingegneria dell'Automazione

Sistemi in Tempo Reale

Giuseppe Lipari

Introduzione alla concorrenza - III

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 are the same time
- expressive power
  - many control application are inherently concurrent
  - concurrency support helps in expressing concurrency, making application development simpler

theoretical 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

resource

- 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 intermangled!
- 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 analize both models in the following slides

cooperative vs competitive

the interaction between concurrent activities (threads or processes) can be classified into:

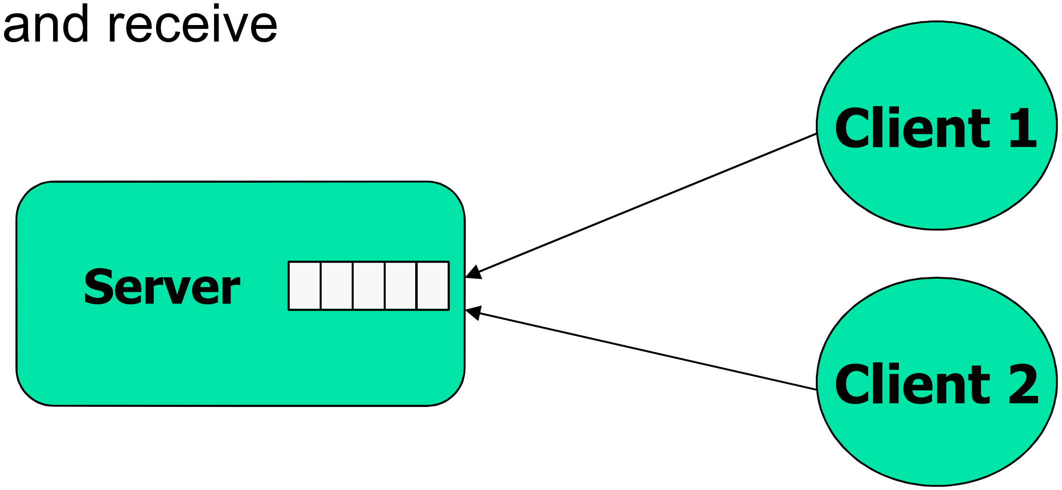
- competitive concurrency
  - different activities compete for the resources
  - one activity does not know anything about the other
  - the OS must manage the resources so to
    - avoid conflicts
    - be fair
- cooperative concurrency
  - many activities cooperate to perform an operation
  - every activity knows about the others
  - they must synchronize on particular events
- interference

competition

- cooperative and competitive activities need different models of execution and synchronization
  - competing activities need to be “protected” from each other
    - separate memory spaces
    - the process model is the best
  - the allocation of the resource and the synchronization must be centralized
    - competitive activities requests for services to a central manager (the OS or some dedicated process) which allocates the resources in a fair way
  - Client/Server model
    - communication is usually done through messages
  - the process model of execution is the best one

competition (2)

- in a client/server system
  - a server manages the resource exclusively
    - for example, the printer
  - if a process needs to access the resource, it sends a request to the server
    - for example, printing a file, or asking for the status
  - the server can send back the responses
  - the server can also be on a remote system
- two basic primitives
  - send and receive



cooperation

- cooperative activities know about each other
  - they do not need memory protection
    - not using memory protection, we have less overhead
  - they need to access the same data structures
  - allocation of the resource is de-centralized
  - shared memory is the best model
  - the thread model of execution is the best one

cooperation and competion

- competion is best resolved by using the message passing model
  - however it can be implemented using a shared memory paradigm too
- cooperation is best implemented by using the shared memory paradigm
  - however, it can be realized by using pure message passing mechanisms
- shared memory or message passing?
  - in the past, there were OS that supported only shared memory or only message passing

cooperation and competion (2)

- a general purpouse OS needs to support both models
  - we need at least protection for competing activities
  - we need to support client/server models. So we need message passing primitives
  - we need to support shared memory for reducing the overhead
- some special OS supports only one of the two
  - for example, many RTOS support only shared memory

interference

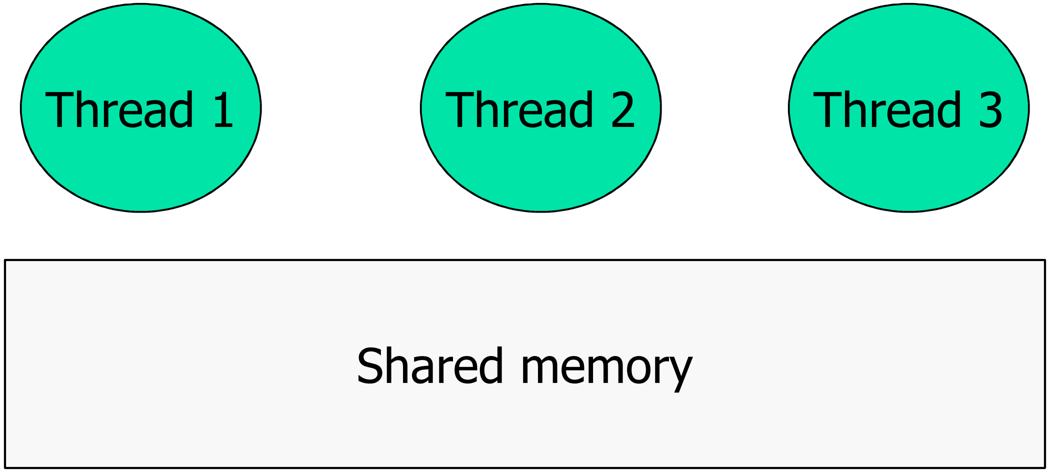
- there is a third kind of interaction, that is interference, due to one of the following programming errors:
  - interactions between processes that are not required by the semantic of the problem
  - erroneous solution to the problems of interaction

- interference problems are usually time-dependent problems

models of concurrency: shared memory

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



resource allocation

- allocation of resource can be
  - static: once the resource is granted, it is never revoked
  - dynamic: resource can be granted and revoked dynamically
    - manager
- access to a resource can be
  - dedicated: one activity at time only is granted access to the resource
  - shared: many activity can access the resource at the same time
    - mutual exclusion

	Dedicated	**Shared**
Static	Compile Time	Manager
Dynamic	Manager	Manager

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 double() { b = b * 2; a = a * 2;}
} obj;
```

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

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

consistency: after each operation, a == b

- bad interleaving

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

resource in a non-consistent state!

consistency

- for each resource, we can state some consistency property
  - 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 hold during an operation
  - if the operations are properly sequentialized, the consistency properties must hold
- formal verification
  - let *R* be a resource, and let *C(R)* be a set of consistency properties on the resource
    - $C(R) = \{ C_i \}$

Definition: a concurrent program is *correct* if, for every possible interleaving of the operations on the resource, the consistency properties hold after each operation

example 3: circular array

```
class CircularArray {
      int array[10];
      int head, tail, num;
public:
      CircularArray() : head(0),
            tail(0), num(0) {}

      bool insert(int elem) {
            if (num == 10) return false;
            array[head] = elem;
            head = (head + 1) % 10;
            num ++;
            return true;
      }
      bool extract(int &elem) {
            if (num == 0) return false;
            elem = array[tail];
            tail = (tail + 1) % 10;
            num - -;
            return true;
      }
} queue;
```

consistency properties

(suppose num++ and num-- atomic)

C_1: if (num == 0 || num == 10)
    head == tail;

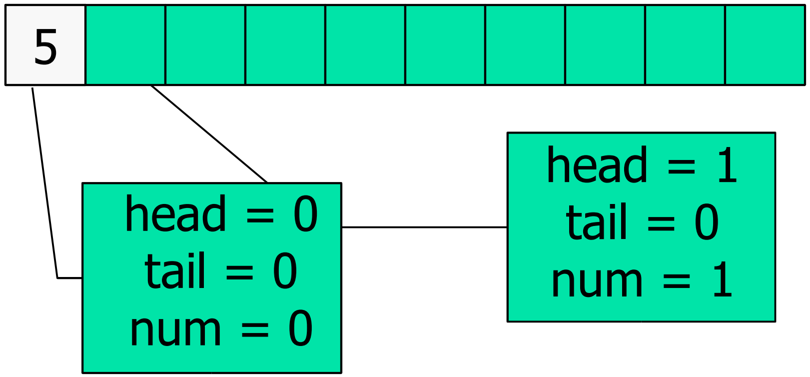
C_2: if (0 < num < 10)
    num == (head – tail) % 10

C_3: num == NI - NE

C_4: (insert x)
  pre: if (num < 10)
  post: num == num + 1 &&
    array[(head-1)%10] = x;

C_5: (extract &x)
  pre: if (num > 0)
  post: num == num –1 &&
    x = array[(tail-1)%10];

example 3: circular array - insert



Initial state:

head = 0; tail = 0; num = 0;

queue.insert (5) ;

head = 1; tail = 0; num = 1;

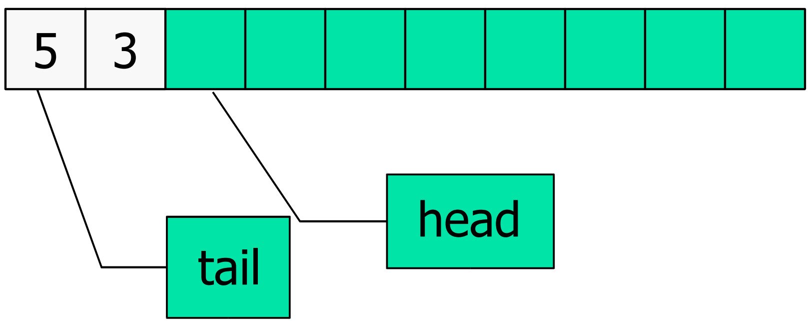
C_2, C_3, C_4 hold

C_2: if ($0 < num < 10$)
num == (head – tail) % 10

C_3: num == NI – NE

C_4: (insert x)
pre: if (num $<$ 10)
post: num == num + 1 &&
array[(head-1)%10] = x;

example 3: circular array – insert (2)



Initial state:

head = 0; tail = 0; num = 0;

queue.insert (5) ;

head = 1; tail = 0; num = 1;

queue. insert (3) ;

head = 2; tail = 0; num = 2;

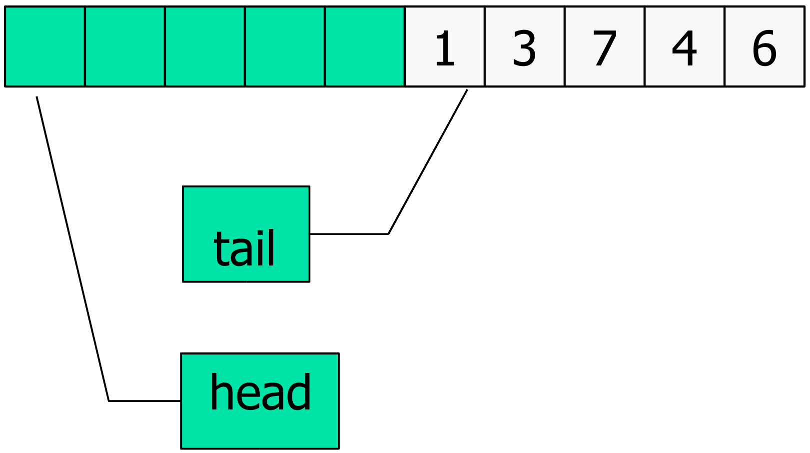
C_2, C_3, C_4 hold

C_2: if (0 < num < 10)
num == (head – tail) % 10

C_3: num == NI – NE

C_4: (insert x)
pre: if (num < 10)
post: num == num + 1 &&
array[(head-1)%10] = x;

example 3: circular array – insert (3)



Initial state:

head = 9; tail = 5; num = 4;

queue. insert (6) ;

head = 0; tail = 5; num = 5

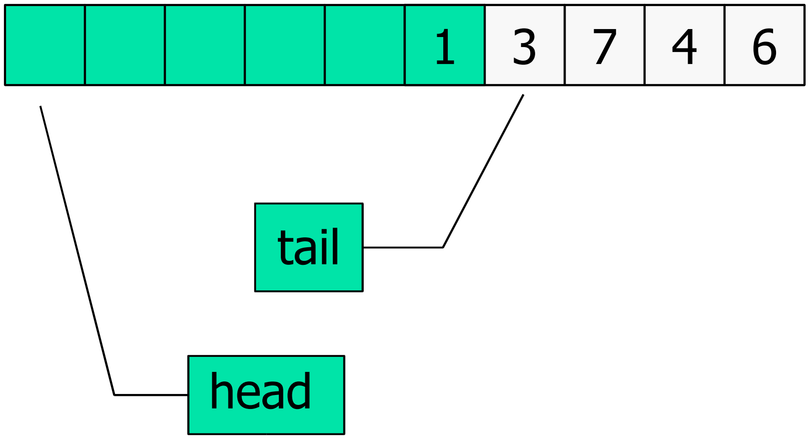
C_2, C_3, C_4 hold

C_2: if (0 < num < 10)
num == (head – tail) % 10

C_3: num == NI – NE

C_4: (insert x)
pre: if (num < 10)
post: num == num + 1 &&
array[(head-1)%10] = x;

example 3: circular array – extract



Initial state:

head = 0; tail = 5; num = 0;

queue. extract (int &elem) ;

head = 0; tail = 6; num = 4

C_2, C_3, C_5 hold

C_2: if (0 < num < 10)
num == (head – tail) % 10

C_3: num == NI – NE

C_5: (extract &x)
pre: if (num > 0)
post: num == num –1
x = array[(tail-1)%10];

example 3: the problem

- if the insert operation is performed by two processes, some consistency property may be violated!

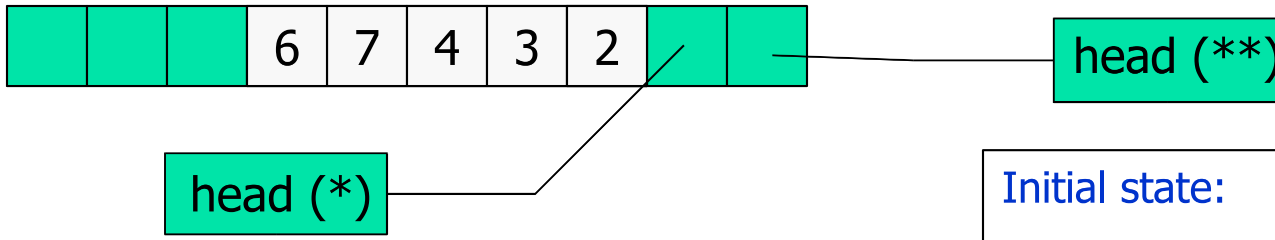
```
CircularArray queue;
```

			6	7	4	3			

```
void *threadA(void *)
{
    ...
    queue.insert(5);
    ...
}
```

```
void *threadB(void *)
{
    ...
    queue.insert(2);
    ...
}
```

example 3: interference



C_4 is violated!

5 != array[head – 1]

```
if (num == 10) return false;
array[head] = 5;
head = (head + 1) % 10; (**)
num ++;
return true;
```

Initial state:

head = 7; tail = 3; num = 4;

queue. insert (5) ; (TA)

queue. insert (2) ; (TB)

```
if (num == 10) return false; (TA)
array[head] = 5;             (TA)
if (num == 10) return false; (TB)
array[head] = 2;             (TB)
head = (head + 1) % 10;      (TB) (*)
num ++;                      (TB)
return true;                 (TB)
head = (head + 1) % 10;      (TA) (**)
num ++;                      (TA)
return true;                 (TA)
```

```
if (num == 10) return false;
array[head] = 2;
head = (head + 1) % 10; (*)
num ++;
return true;
```

Final State:

head = 9; tail = 3; num = 6;

example 3: correctness

- the previous program is not correct
  - it exist a possible interleaving of two insert operations that leaves the resource in a inconsistent state
- proving the non-correctness is easy
  - it suffices to find a counter example
- proving the correctness is not easy
  - it is necessary to prove the correctness for every possible interleaving of every operation

example 3: problem

- what if an insert and an extract are interleaved?
  - nothing bad can happen!!
  - proof
    - if 0<num<10, insert() and extract() are independent
    - if num==0
      - if extract begins before insert, it immediately returns false, so nothing bad can happen
      - if insert begins before, extract will still return false, so it cannot interfere with insert
    - same thing when num==10
- question: what happens if we exchange the sequence of instructions in insert or extract?

example 3: CircularArray properties

- a) if more than one thread executes queue.insert()
  - inconsistency!!
- b) if we have only two threads
  - one threads calls queue.insert() and the other thread calls queue.extract()
  - no inconsistency!
- the order of the operations is important!
  - a wrong order can make the object inconsistency even under the assumption b)
    - the case is when num is incremented but the data has not yet been inserted
    - in any case, the final result depends on the timings of the different requests (e.g, an insertion when the buffer is full)

example 3: questions

- problem:
  - in the previous example, we supposed that num++ and num-- are atomic operations
  - what happens if they are not atomic?
- question:
  - assuming that operation -- and ++ are not atomic, can we make the circularArray safe under the assumption b) ?
    - hint: try to substitute variable num with two boolean variables, bool empty and bool full;

critical sections

- definitions
  - 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 mutual exclusion)

critical sections: atomic operations

- in single processor systems
  - disable interrupts during a critical section
- problems:
  - 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!
  - Non voluntary context switch is disabled during the critical section!
    - we must avoid conflicts on the resource, not disabling interrupts!

critical sections: atomic operations (2)

- multi-processor
  - define a flag s for each resource
  - use lock(s)/unlock(s) around the critical section
- problems:
  - busy waiting: if the critical section is long, we waste a lot of time
  - cannot be used in single processors!

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

critical sections: disabling preemption

- 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>

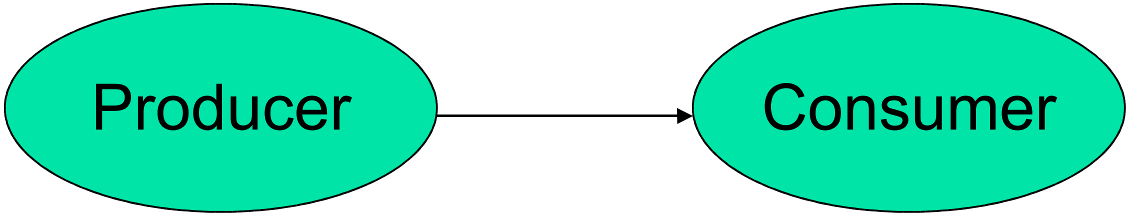
no context switch may happen during the critical section but interrupts are enabled

critical sections: selectively dis. preemption

- there exist some general mechanisms to implement mutual exclusion only between the tasks that uses the resource.
  - semaphores
  - mutexes

synchronisation

- 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)



producer/consumer

- the two threads have different speeds
  - for example the producer is much faster than the consumer
  - we need to store the integers in a queue, so that no data is lost
  - let’s use the CircularArray class

producer/consumer (2)

```
CircularArray queue;
```

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

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

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

a more general approach

- we need to provide a general mechanism for synchonisation 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)

general mechanism: semaphores

- Djikstra proposed the semaphore mechanism
  - a semaphore is an abstract entity that consists of
    - a counter
    - a blocking queue
    - operation wait
    - operation signal
  - the operations on a semaphore must be atomic

semaphores

- 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 sempahores
  - 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
  - if counter == 0, the requiring thread is blocked
    - it is removed from the ready queue
    - it is 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
    - it is inserted in the ready queue
  - otherwise, increment counter

semaphores

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

signal semantics

- what happens when a thread blocks on a semaphore?
  - in general, it is inserted in a BLOCKED queue
- extraction from the blocking queue can follow different semantics:
  - strong semaphore
    - the threads are removed in well-specified order
    - for example, the FIFO order is the fairest policy, priority based ordering, ...
  - signal and suspend
    - after the new thread has been unblocked, a thread switch happens
  - signal and continue
    - after the new thread has been unblocked, the thread that executed the signal continues to execute
- concurrent programs should not rely too much on the semaphore semantic

mutual exclusion with semaphores

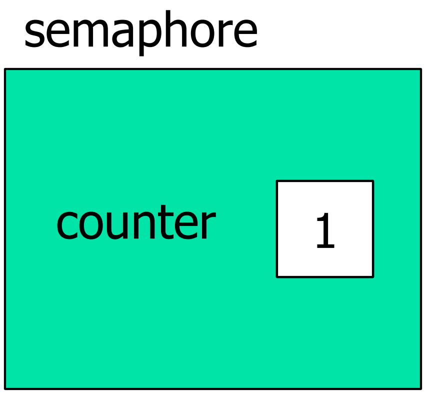
- how to use a semaphore for critical sections
  - define a semaphore initialized to 1
  - before entering the critical section, perform a wait
  - after leaving the critical section, perform a signal

```
Semaphore s(1);
```

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

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

mutual exclusion with semaphores (2)



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

synchronization

- how to use a semaphore for synchronization
  - define a semaphore initialized to 0
  - at the synchronization point, perform a wait
  - when the synchronization point is reached, perform a signal
  - in the example, threadA blocks until threadB wakes it up

```
Semaphore s(0);
```

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

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

  - how can both A and B synchronize on the same instructions?

producer/consumer

- consider a producer/consumer system
  - one producer executes queue.insert()
    - we want the producer to be blocked when the queue is full
    - the producer will be unblocked when there is some space again
  - one consumer executes queue.extract
    - we want the consumer to be blocked when the queue is empty
    - the consumer will be unblocked when there is some space again
  - first attempt: one producer and one consumer only

producer/consumer (2)

```
class CircularArray {
    int array[10];
    int head, tail;
    Semaphore empty, full;
public:
    CircularArray() : head(0), tail(0),
        empty(0), full(10) {}
    void insert(int elem) {
        wait(full);
          array[head] = elem;
          head = (head + 1) % 10;
        signal(empty);
    }
    void extract(int &elem) {
        wait(empty);
          elem = array[tail];
          tail = (tail + 1) % 10;
        signal(full);
    }
} queue;
```

Note: there is no member called num as in the example 3 (slide 22)!!!

producer/consumer: properties

- notice that
  - the value of the counter of empty is the number of elements in the queue
    - it is the number of times we can call extract without blocking
  - the value of the counter of full is the complement of the elements in the queue
    - it is the number of times we can call insert without blocking

- exercise
  - prove that the implementation is correct
    - insert() never overwrites elements
    - extract() always gets an element of the queue

producers/consumers

- now let’s combine mutual exclusion and synchronization
  - consider a system in which there are
    - many producers
    - many consumers
  - we want to implement synchronization
  - we want to protect the data structure

producers/consumers: does it work?

```
class CircularArray {
      int array[10];
      int head, tail;
      Semaphore full, empty;
      Semaphore mutex;
public:
      CircularArray() : head(0), tail(0),
              empty(0), full(10), mutex(1) {}
      void insert(int elem);
      void extract(int &elem);
} queue;
```

```
void CircularArray::insert(int elem)
{
      mutex.wait();
        full.wait();
          array[head]=elem;
          head = (head+1)%10;
        empty.signal();
      mutex.signal();
}
```

```
void CircularArray::extract(int &elem)
{
      mutex.wait();
        empty.wait();
          elem = array[tail];
          tail = (tail+1)%10;
        full.signal();
      mutex.signal();
}
```

producers/consumers: correct solution

```
class CircularArray {
      int array[10];
      int head, tail;
      Semaphore full, empty;
      Semaphore mutex;
public:
      CircularArray() : head(0), tail(0),
              empty(0), full(10), mutex(1) {}
      void insert(int elem);
      void extract(int &elem);
} queue;
```

```
void CircularArray::insert(int elem)
{
      full.wait();
        mutex.wait();
          array[head]=elem;
          head = (head+1)%10;
        mutex.signal();
      empty.signal();
}
```

```
void CircularArray::extract(int &elem)
{
      empty.wait();
        mutex.wait();
          elem = array[tail];
          tail = (tail+1)%10;
        mutex.signal();
      full.signal();
}
```

producers/consumers: deadlock situation

- 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, and the two threads will never proceeds
- as a rule, never insert a blocking synchronization inside a critical section!!!

semaphore implementation

- system calls
  - 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
- protection:
  - 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 (2)

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

}
```

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

}
```