



Scuola Superiore Sant'Anna



Concurrency: mutual exclusion and synchronisation

Giuseppe Lipari



The need for concurrency

- There are many reason for concurrency
- 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

- 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



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 case, 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 analyze both models in the following



Resource Allocation

- Allocation of 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
 - Static: once the resource is granted, it is never revoked
 - Dynamic: resource can be granted and revoked dynamically

| | Dedicated | Shared |
|----------------|------------------|---------------|
| Static | Compile Time | Manager |
| Dynamic | Manager | Manager |



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



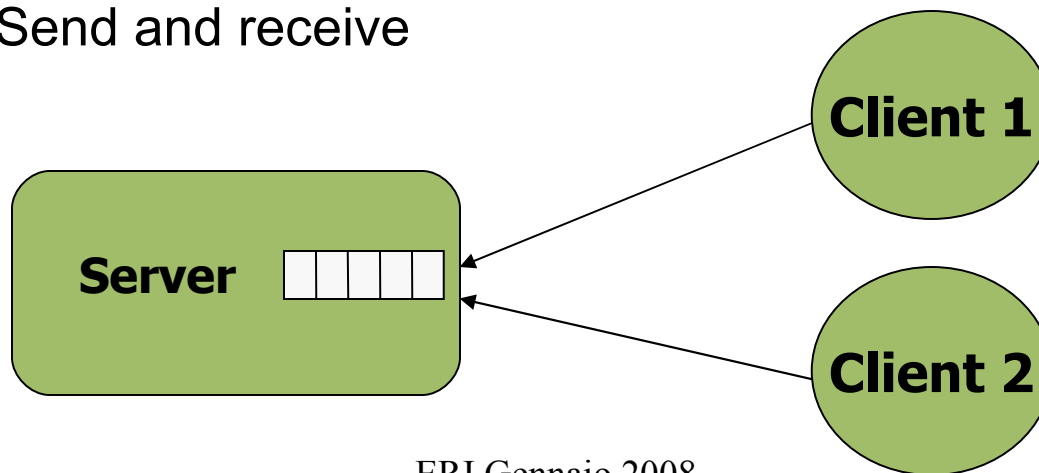
Competing activities

- 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



Message passing

- 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





Cooperative model

- Cooperative activities know about each other
 - They do not need protection
 - ✓ Not using 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 Competition

- Competition 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 a 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



Competition and cooperation

- A general purpose 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



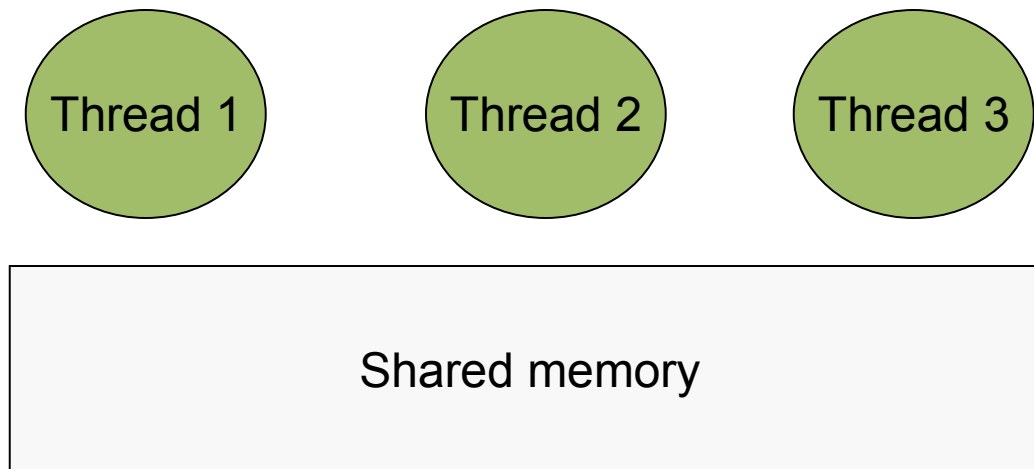
SHARED MEMORY

mutual exclusion and synchronization



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

Shared memory

```
int x ;
```

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

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




Example

- 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 - II

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();  
    ...  
}
```

■ 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 |

Consistency:
after each
operation,
a == b

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
 - ✓ $C = \{ C_i \}$
 - A concurrent program is *correct* if, for every possible interleaving of the operations on the resource, the consistency properties hold after each operation



Example

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

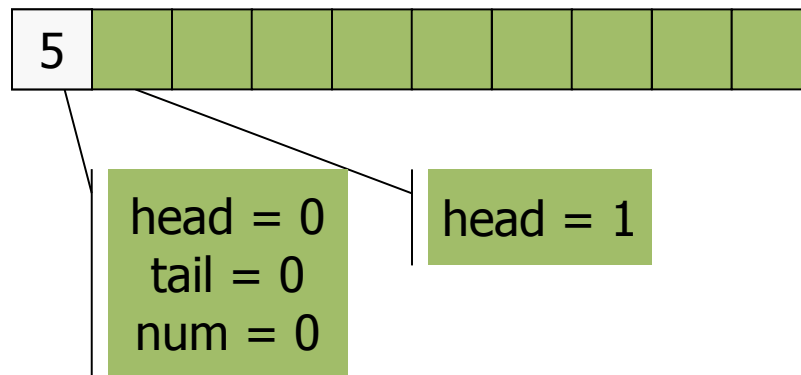
    bool insert(int elem) {
        if (num == 10) return false;
        array[head] = elem;
        head = (head + 1) % 10;
        num++;
        return true;
    }

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

ERI Gennaio 2008



Example - Insert



Initial state:

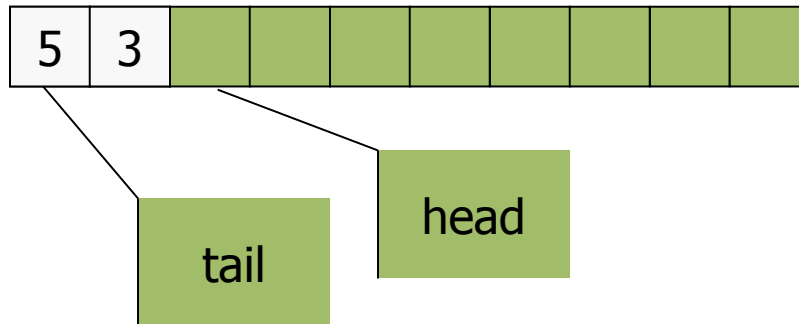
head = 0; tail = 0; num = 0;

queue.insert (5) ;

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



Example - Insert II



Initial state:

$\text{head} = 0; \text{tail} = 0; \text{num} = 0;$

`queue.insert (5) ;`

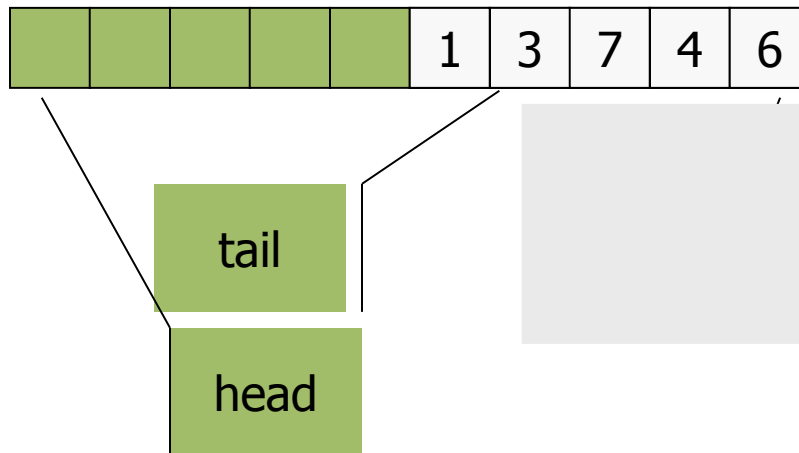
$\text{head} = 1; \text{tail} = 0; \text{num} = 1;$

`queue.insert (3) ;`

$\text{head} = 2; \text{tail} = 0; \text{num} = 2;$



Example - Inserting III



Initial state:

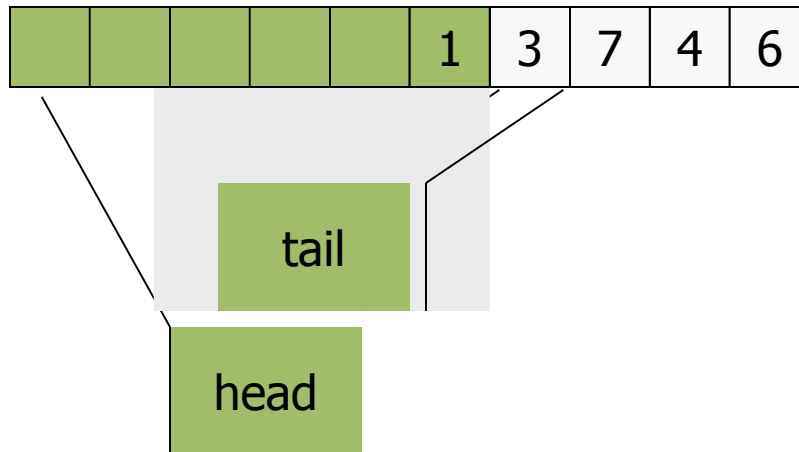
$\text{head} = 9; \text{tail} = 5; \text{num} = 4;$

`queue.insert(6);`

$\text{head} = 0; \text{tail} = 5; \text{num} = 5$



Example - Extract



Initial state:

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

```
queue.extract(int &elem) ;
```

```
head = 0; tail = 6; num = 4
```




Consistency properties

- Let $INS(k)$ be the data inserted in the queue after the k -th successful call of the insert function
- Let $EXT(k)$ be the data extracted from the queue after the k -th successful call of the extract function

Consistency Properties

C1: for all k , $EXT(k) == INS(k)$

C2: $(num == 0 \text{ OR } num == 10) \Leftrightarrow (head == tail)$

C3: $(0 < num < 9) \Rightarrow (num == (head - tail) \% 10)$

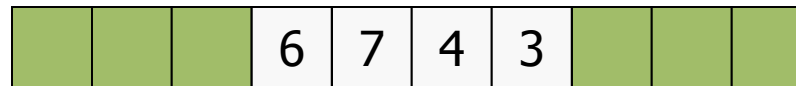
These properties must hold *before* or *after* the insert and extract operation, not *during*



Problems

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

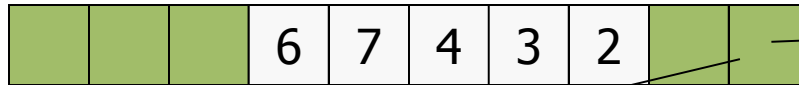
CircularArray queue;



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

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

Interference



head (2)

head (1)

C₁ is violated!

```
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;



Correctness

- The previous program is not correct
 - There exists a possible interleaving of two 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



Assumption: one producer / one consumer

- Assume that
 - only one thread calls insert (producer)
 - only one thread calls extract (consumer)
- Is the program correct?
- Answer:
 - NO! operations $\text{num}++$ and $\text{num}--$ are not atomic!
 - See previous examples on variable x



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**



A SW solution to mutual exclusion

- Mutual exclusion
 - We need one general mechanism to protect critical sections and synchronization
 - First algorithm by Dekker
 - later improved by Peterson
 - another by Lamport



Dekker's algorithm

```
f0 = false;  
f1 = false;  
turn = 0; // or 1
```

```
p0:  
  f0 = true;  
  while (f1) {  
    if (turn ≠ 0) {  
      f0 = false;  
      while (turn ≠ 0);  
      f0 = true;  
    }  
  }
```

// critical section

```
...  
turn := 1  
f0 := false
```

```
p1:  
  f1 = true;  
  while (f0) {  
    if (turn ≠ 1) {  
      f1 = false;  
      while (turn ≠ 1);  
      f1 = true;  
    }  
  }
```

// critical section

```
...  
turn := 0  
f1 := false
```




Peterson's algorithm

```
flag[0] = 0  
flag[1] = 0  
turn = 0
```

```
P0: flag[0] = 1;  
    turn = 1;  
    while(flag[1] && turn == 1);
```

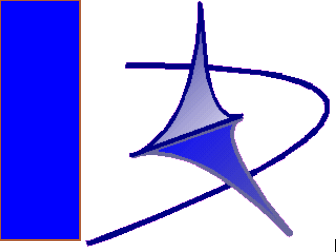
// critical section

```
...  
// end of critical section  
flag[0] = 0;
```

```
P0: flag[1] = 1;  
    turn = 0;  
    while( flag[0] && turn == 0 );
```

// critical section

```
...  
// end of critical section  
flag[1] = 0;
```



Lamport's bakery algorithm

```
bool Entering[N]; // init to false
int  Number[N]; // init to 0

void lock(int i) {
    Entering[i] = true;
    Number[i] = 1 + max(Number[1], ..., Number[N]);
    Entering[i] = false;
    for (j = 1; j <= N; j++) {
        while (Entering[j]);
        while ((Number[j] != 0) && ((Number[j], j) < (Number[i], i)));
    }
}

unlock(integer i) { Number[i] = 0; }

Thread(integer i) {
    ...
    lock(i);
    // The critical section goes here...
    unlock(i);
    ...
}
```



Problems with the previous solutions

- All these solutions are based on *busy-waiting*
 - the algorithms perform one or more loops for checking flags: in single processor systems, this is a waste of time
 - all algorithms require special variables (flags and numbers) for each thread; cannot be easily generalized to an arbitrary (unknown) number of threads
- We need a more efficient approach



Q&D solutions to mutual exclusion

- 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!
 - Concurrency is disabled during the critical section!
 - ✓ We must avoid conflicts on the resource, not disabling interrupts!



Q&D solutions to mutual exclusion

- 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



Q&D solutions to mutual exclusion

- 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);  
...
```

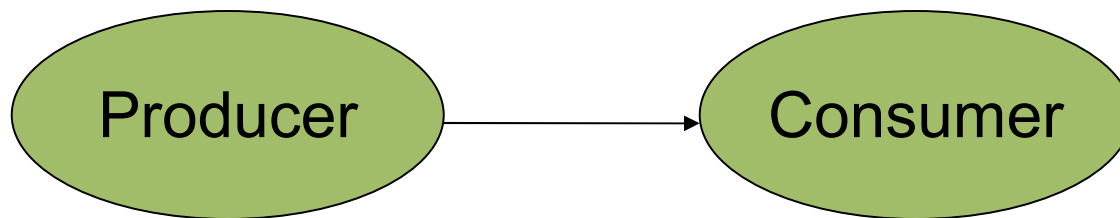


SYNCHRONIZATION



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 (the 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

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 synchronisation
- Requirements
 - Synchronise two threads on one condition
 - ✓ For example, block the producer when the queue is full
 - Provide mutual exclusion between critical sections
 - ✓ Avoid two insertions operation to interleave



SEMAPHORES



General mechanism

- Dijkstra proposed the semaphore mechanism
 - A semaphore is an abstract entity that consists
 - ✓ a counter
 - ✓ a blocking queue
 - ✓ operation wait
 - ✓ operation signal
 - The operations on a semaphore are considered atomic



Semaphores

- Semaphores are 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

- A wait operation has the following behaviour
 - If $\text{counter} == 0$, the requiring thread is blocked
 - ✓ It is removed from the ready queue
 - ✓ It is inserted in the blocked queue
 - If $\text{counter} > 0$, then $\text{counter}--$;
- A signal operation has the following behaviour
 - If $\text{counter} == 0$ 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++;  
    }  
};
```




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

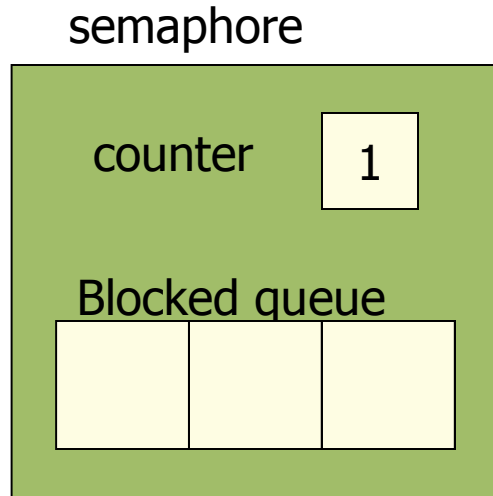
Semaphore s(1);

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

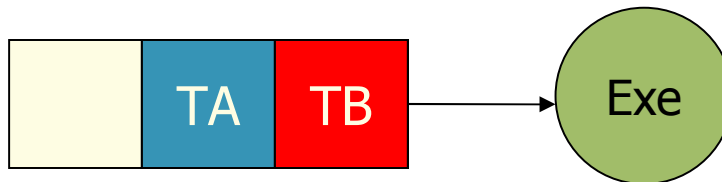
```
void *threadB(void *)  
{  
    ...  
    s.wait();  
    <critical section>  
    s.signal();  
    ...  
}
```



Mutual exclusion with semaphores



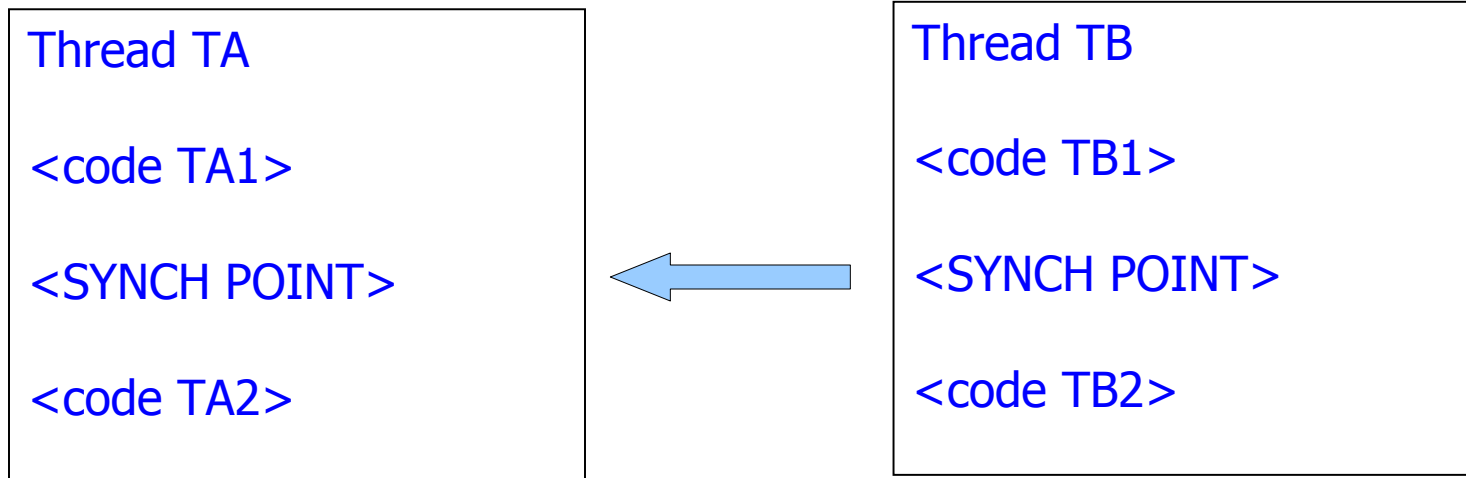
| | |
|------------------------|------|
| s.wait(); | (TA) |
| <critical section (1)> | (TA) |
| s.wait() | (TB) |
| <critical section (2)> | (TA) |
| s.signal() | (TA) |
| <critical section> | (TB) |
| s.signal() | (TB) |





Synchronization with semaphores

- Suppose a thread TA wants to synchronize with thread TB
 - thread TA cannot execute <code TA2> before thread TB complete <code TB1>





Solution with semaphores

Semaphore `s(0);`

Thread TA

<code TA 1>

`s.wait();`

<code TA 2>

Thread TB

<code TB 1>

`s.signal();`

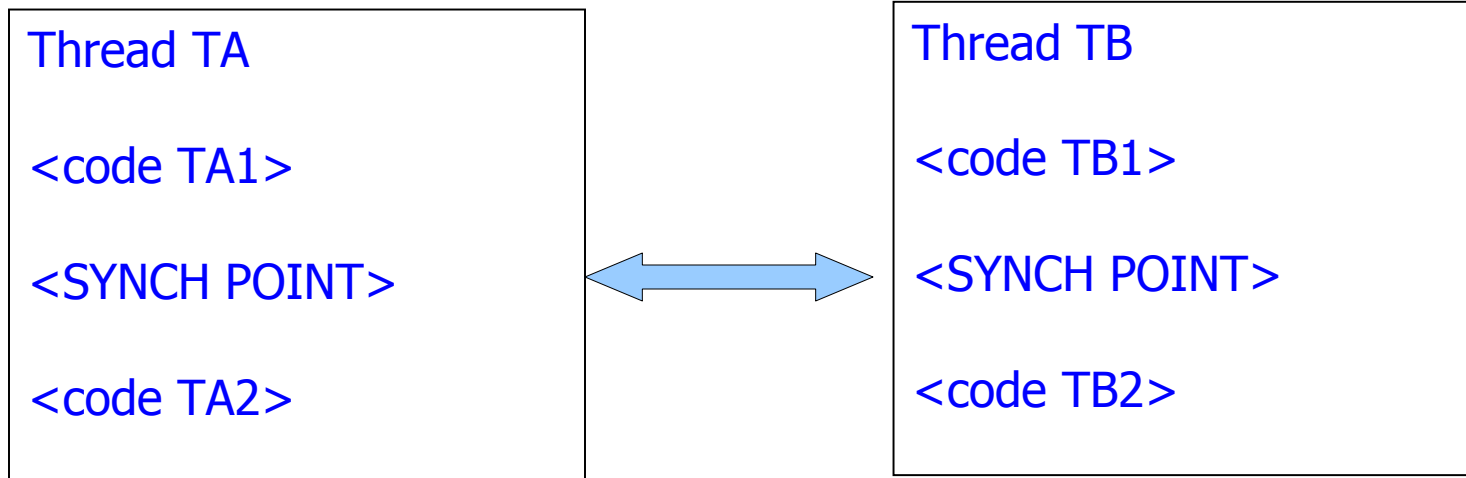
<code TB 2>

- Try it out with pen and paper!



Problem: Symmetric synchronization

- Suppose a thread TA wants to synchronize with thread TB
 - thread TA cannot execute <code TA2> before thread TB complete <code TB1>, and viceversa





Solution

- When we have 2 threads
 - ???
- Generalization to N threads
 - ???



Producer/Consumer with semaphores

- 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 with semaphores

```
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;
```




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 and many consumers
 - We want to implement synchronization
 - We want to protect the data structure



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();  
}
```



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();  
}
```



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!



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



Implementation

```
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();
}
```



POSIX THREADS AND SEMAPHORES



See slides on pthreads



Problems

- See exercises



A DESIGN METHODOLOGY FOR SHARED MEMORY PROGRAMS



Methodology

- We present here a structured methodology for programming in shared memory systems
- The methodology leads to “safe” programs
- Not necessarily optimized!
- Programmers can always use their intelligence and come up with “elegant” solutions



Shared memory programming

- A program is a set of
 - threads that interact by accessing ...
 - data structures (*shared resources*)
- A data structure can be seen as
 - a set of variables
 - a set of functions operating on the variables
- Threads
 - access the data structures only through functions



Object Oriented programming

- In C++

```
class SharedData {  
    int array[10];  
    int first, last;  
    ...  
public:  
    SharedData();  
    int insert(int a);  
    int extract();  
}
```

- In C

```
struct SharedData {  
    int array[10];  
    int first, last;  
};    ...  
  
SharedData_init(struct SharedData *d);  
int SharedData_insert(struct SharedData *d, int a);  
int SharedData_extract(struct SharedData *d);
```



Shared Data structure

- Encapsulating the semaphores
 - The data structure
 - should already include the mechanisms for mutual exclusion and synchronization
 - for example, the CircularArray data structure
 - Functions on the data structure
 - should use the semaphores inside the function
 - Threads
 - can only access the data structure through functions



How to program the data structure?

- Some design considerations
 - First, design the interface: that is, which functions the threads need to call
 - Mutual exclusion
 - for simplicity, all functions on the same data structure should be in mutual exclusion
 - maybe, this is not optimized, but it is SAFE!



Mutual exclusion

- For each data structure,
 - define a mutual exclusion semaphore, initialized to 1
- For each function
 - just after starting the function, take the semaphore, and leave it before returning



Mutual exclusion

- Example

```
class MyData {  
    ...;  
    sem_t m;  
    ...;  
public:  
    MyData() {  
        ...  
        sem_init(&m, 0, 1);  
    }  
  
    int myfun() {  
        sem_wait(&m);  
        ....  
        sem_post(&m);  
    }  
};
```



Synchronization

- Design consideration
 - Specify the behavior of the functions
 - under which conditions a calling thread should be blocked?
 - Identify all different blocking conditions



Synchronization

- Other design considerations
 - identify unblocking conditions
 - when a blocked thread should be unblocked
 - mark the functions that should unblock those threads
- Putting all together
 - draw a state diagram for the resource
 - STATES: the various states of the resource
 - EVENTS: threads that call the functions



Example: CircularArray with Manager

- Problem explanation
- Design the data structure interface
- Identify blocking and unblocking conditions
- Draw the state diagram



Coding Rules

- For each blocking condition
 - a semaphore initialized to 0 (*blocking semaphore*)
 - an integer that counts the number of blocked threads on the condition (*blocking counter*) (init to 0)
- One (or more) integer variable(s) to code the state
- Write the initialization function (constructor in C++)



The functions

- Finally, code the functions
 - take the mutex at the beginning
 - check the blocking conditions (if any)
 - if the thread has to be blocked,
 - increment the blocking counter, signal on the mutex, wait on the blocking semaphore, wait on the mutex
 - perform the code, change state if necessary
 - check unblocking conditions
 - if a thread has to be unblocked
 - decrement the blocked counter, signal the semaphore



The code



A subtle error

- The “man in the middle” problem
- Solution:
 - check blocking conditions in a while() loop



Passing “le baton”

- A (more elegant) solution



MONITORS

PTHREADS – MUTEXES AND CONDITION VARIABLES



Monitors

- Monitors are a language structure equivalent to semaphores, but cleaner
 - A monitor is similar to an object in a OO language
 - It contains variables and provides procedures to other software modules
 - Only one thread can execute a procedure at a certain time
 - ✓ Any other thread that has invoked the procedure is blocked and waits for the first threads to exit
 - ✓ Therefore, a monitor implicitly provides mutual exclusion



Monitors

- Monitors support synchronization with *Condition Variables*
 - A condition variable is a blocking queue
 - Two operations are defined on a condition variable
 - ✓ wait() -> suspends the calling thread on the queue
 - ✓ signal() -> resume execution of one thread blocked on the queue
- Important note:
 - wait() and signal() operation on a condition variable are different from wait and signal on a semaphore!
 - There is not any counter in a condition variable!
 - If we do a signal on a condition variable with an empty queue, the signal is lost



Monitors in Java

- Java provides something that vaguely resembles monitors
 - the “synchronized” keyword allows to define classes with protected functions
 - every “synchronized” class has one implicit condition variable
 - you can “signal” one thread with `notify()`; and all threads with `notifyAll()`;



Monitors in POSIX

- It is not possible to provide monitors in C
 - C and C++ do not provide any concurrency control mechanism; they are “purely sequential languages”
- POSIX allows something similar to Monitors through library calls



Slides on POSIX monitors



Exercises on POSIX monitors



COMPLEX SYNCHRONIZATION PROBLEMS

READERS - WRITERS



Readers/writers

- One shared buffer
- Readers:
 - They read the content of the buffer
 - Many readers can read at the same time
- Writers
 - They write in the buffer
 - While one writer is writing no other reader or writer can access the buffer
- Use semaphores to implement the resource



Simple implementation

```
class Buffer {  
    Semaphore wsem;  
    Semaphore x;  
    int nr;  
  
public:  
    Buffer() : wsem(1), x(1), nr(0) {}  
    void read();  
    void write();  
} buffer;
```

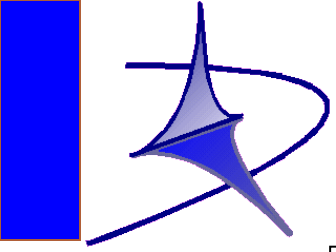
```
void Buffer::read() {  
    x.wait();  
    nr++;  
    if (nr==1) wsem.wait();  
    x.signal();  
    <read the buffer>  
    x.wait();  
    nr--;  
    if (nr==0) wsem.signal();  
    x.signal();  
}
```

```
void Buffer::write() {  
    wsem.wait();  
    <write the buffer>  
    wsem.signal();  
}
```



Problem: starvation

- Suppose we have 2 readers (R1 and R2) and 1 writer (W1)
 - Suppose that R1 starts to read
 - While R1 is reading, W1 blocks because it wants to write
 - Now R2 starts to read
 - Now R1 finishes, but, since R2 is reading, W1 cannot be unblocked
 - Before R2 finishes to read, R1 starts to read again
 - When R2 finishes, W1 cannot be unblocked because R1 is reading



Priority to writers!

```
class Buffer {  
    Semaphore x, y, z, wsem, rsem;  
    int nr, nw;  
  
public:  
    Buffer() : x(1), y(1), z(1), wsem(1), rsem(1), nr(0), nw(0) {}  
}
```

```
void Buffer::write() {  
    y.wait();  
    nw++;  
    if (nw==1) rsem.wait();  
    y.signal();  
    wsem.wait();  
    <write the buffer>  
    wsem.signal();  
    y.wait();  
    nw--;  
    if (nw == 0) rsem.signal();  
    y.signal();  
}
```

```
void Buffer::read() {  
    z.wait();  
    rsem.wait();  
    x.wait();  
    nr++;  
    if (nr==1) wsem.wait();  
    x.signal();  
    rsem.signal();  
    z.signal();  
    <read the buffer>  
    x.wait();  
    nr--;  
    if (nr==0) wsem.signal();  
    x.signal();  
}
```



Problem

- Can you solve the readers/writers problem in the general case?
 - No starvation for readers
 - No starvation for writers
- Solution
 - Maintain a FIFO ordering with requests
 - ✓ If at least one writer is blocked, every next reader blocks
 - ✓ If at least one reader is blocked, every next writer blocks
 - ✓ One single semaphore!



Solution

```
class Buffer {  
    int nbr, nbw;  
    int nr, nw;  
    Semaphore rsem, wsem;  
    Semaphore m;  
public:  
    Buffer():  
        nbw(0),nbr(0), nr(0), nw(0),  
        rsem(0), wsem(0) {}  
    void read();  
    void write();  
};
```




Solution

```
void Buffer::read()
{
    m.wait();
    if (nw || nbw) {
        nbr++;
        m.signal(); rsem.wait(); m.wait();
        while (nbr > 0)
            {nbr--; rsem.signal();}
    }
    nr++;
    m.signal();
    <read buffer>;
    m.wait();
    nr--;
    if (nbw && nr == 0) wsem.signal();
    m.signal();
}
```

```
void Buffer::write()
{
    m.wait();
    if (nw || nbw || nr || nbr) {
        nbw++;
        m.signal(); wsem.wait(); m.wait();
        nbw--;
    }
    nw++;
    m.signal();
    <write buffer>;
    m.wait();
    nw--;
    if (nbr) {nbr--; rsem.signal();}
    else if (nbw) wsem.signal();
    m.signal();
}
```



MESSAGE PASSING



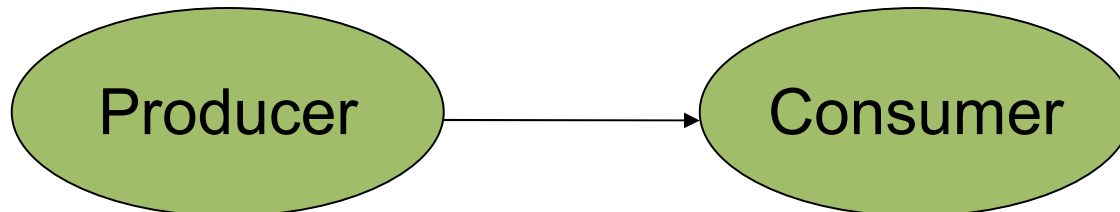
Message passing

- Message passing systems are based on the basic concept of message
- Two basic operations
 - `send(destination, message);`
 - `receive(source, &message);`
- Two variants
 - ✓ Both operations can be synchronous or asynchronous
 - ✓ receive can be symmetric or asymmetric



Producer/Consumer with MP

- The producer executes `send(consumer, data)`
- The consumer executes `receive(producer, data);`
- No need for a special communication structure (already contained in the send/receive semantic)





Synchronous communication

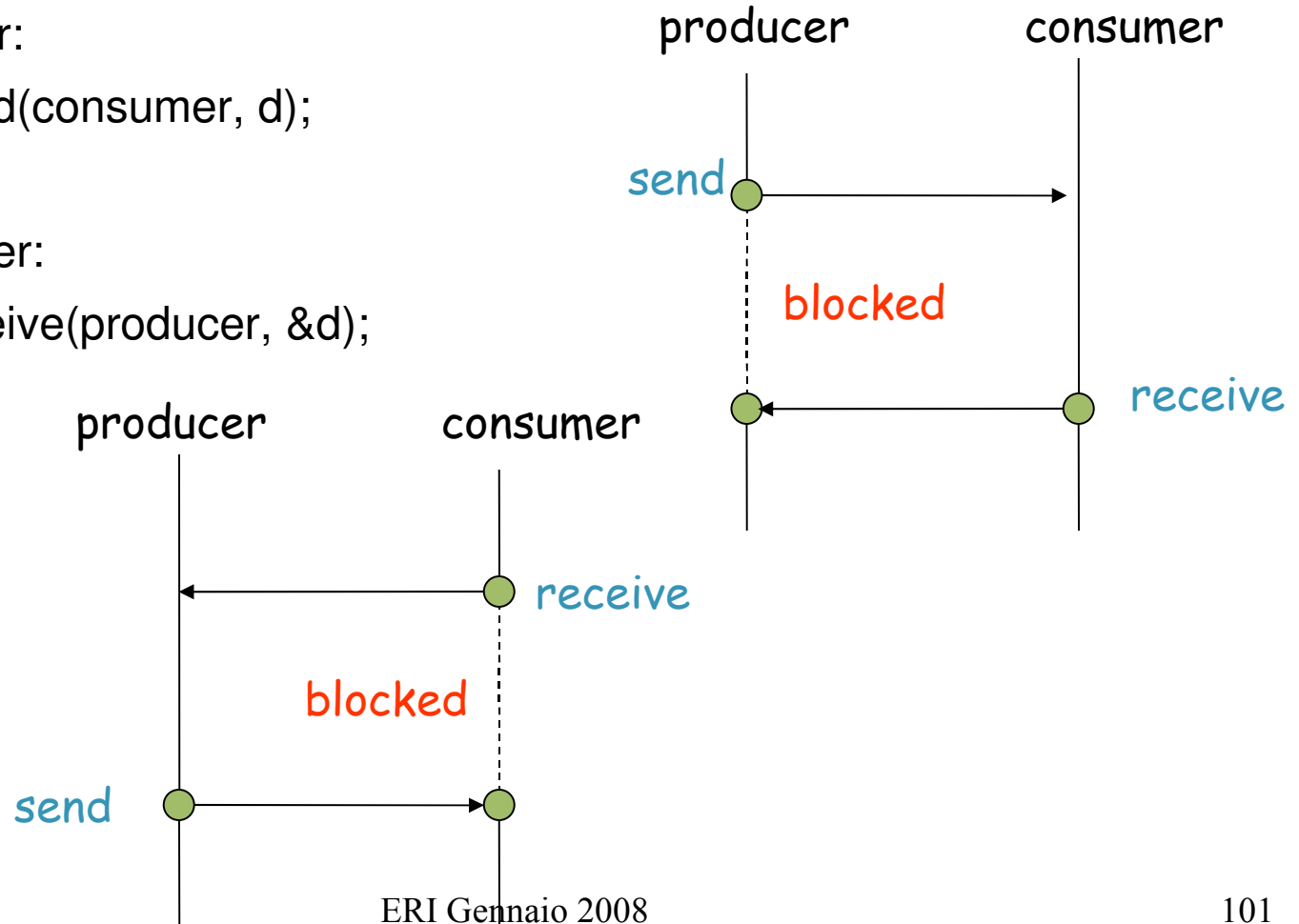
■ Synchronous send/receive

producer:

```
s_send(consumer, d);
```

consumer:

```
s_receive(producer, &d);
```





Async send/ Sync receive

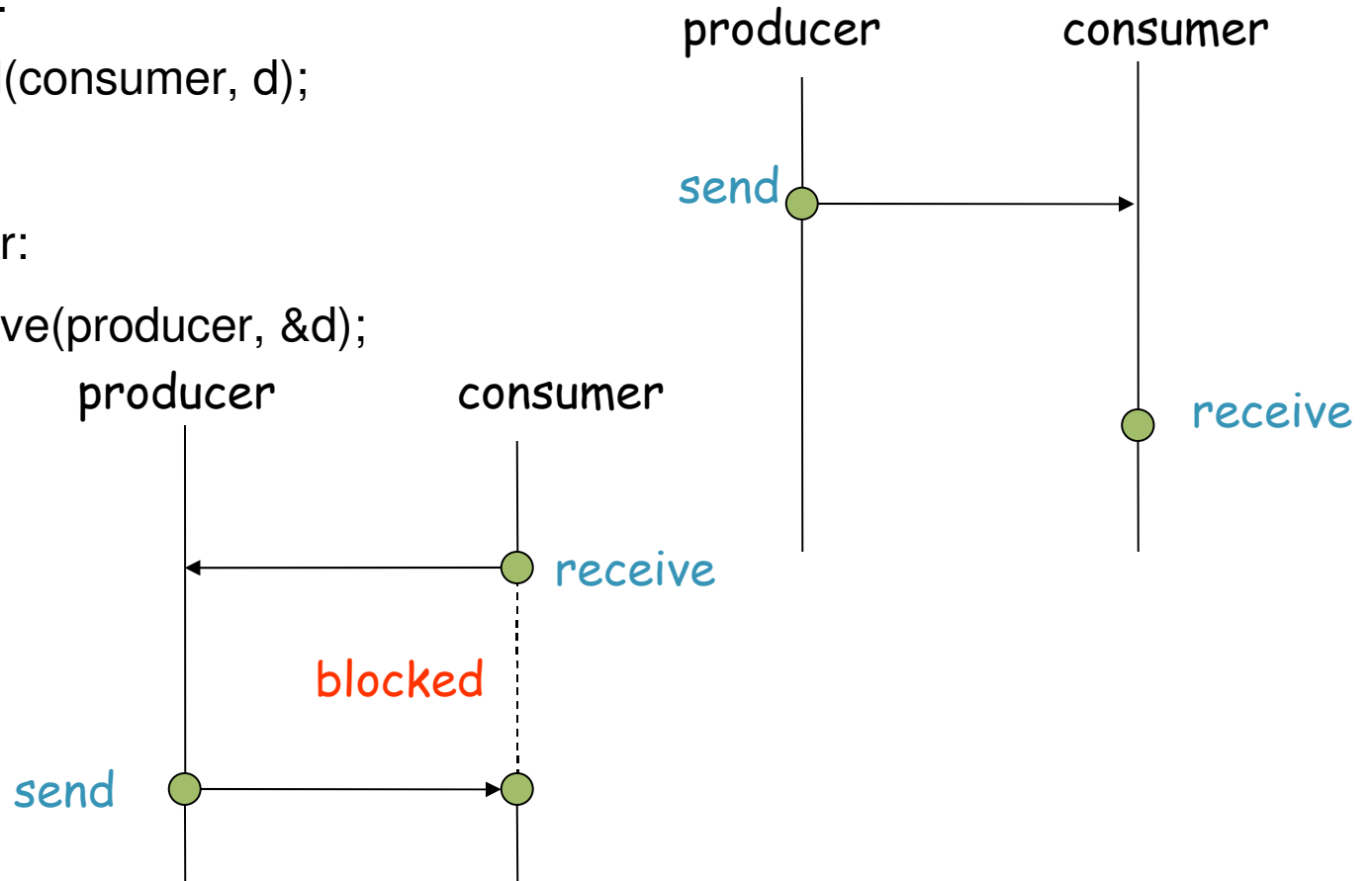
- Asynchronous send / synchronous receive

producer:

```
a_send(consumer, d);
```

consumer:

```
s_receive(producer, &d);
```





Asymmetric receive

- Symmetric receive
 - `receive(source, &data);`
- Often, we do not know who is the sender
 - Imagine a web server;
 - ✓ the programmer cannot know in advance the address of the browser that will request the service
 - ✓ Many browser can ask for the same service
- Asymmetric receive
 - `source = receive(&data);`



Message passing systems

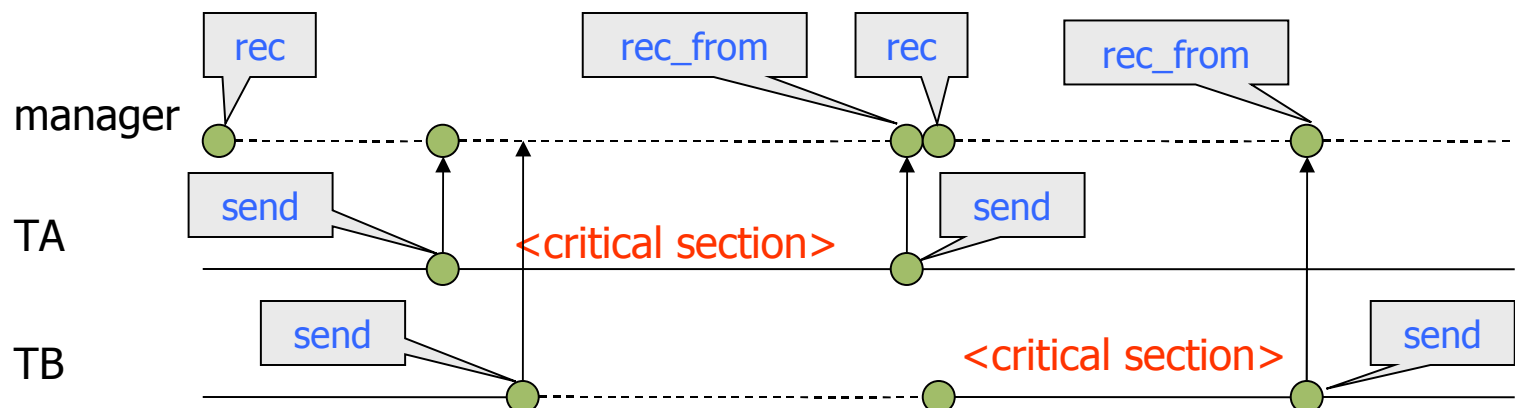
- In message passing
 - Each resource needs one threads manager
 - The threads manager is responsible for giving access to the resource
- Example: let's try to implement mutual exclusion with message passing primitives
 - One thread will ensure mutual exclusion
 - Every thread that wants to access the resource must
 - ✓ send a message to the manager thread
 - ✓ access the critical section
 - ✓ send a message to signal the leaving of the critical section



Sync send / sync receive

```
void * manager(void *)  
{  
    thread_t source;  
    int d;  
    while (true) {  
        source = s_receive(&d);  
        s_receive_from(source, &d);  
    }  
}
```

```
void * thread(void *)  
{  
    int d;  
    while (true) {  
        s_send(manager, d);  
        <critical section>  
        s_send(manager, d);  
    }  
}
```

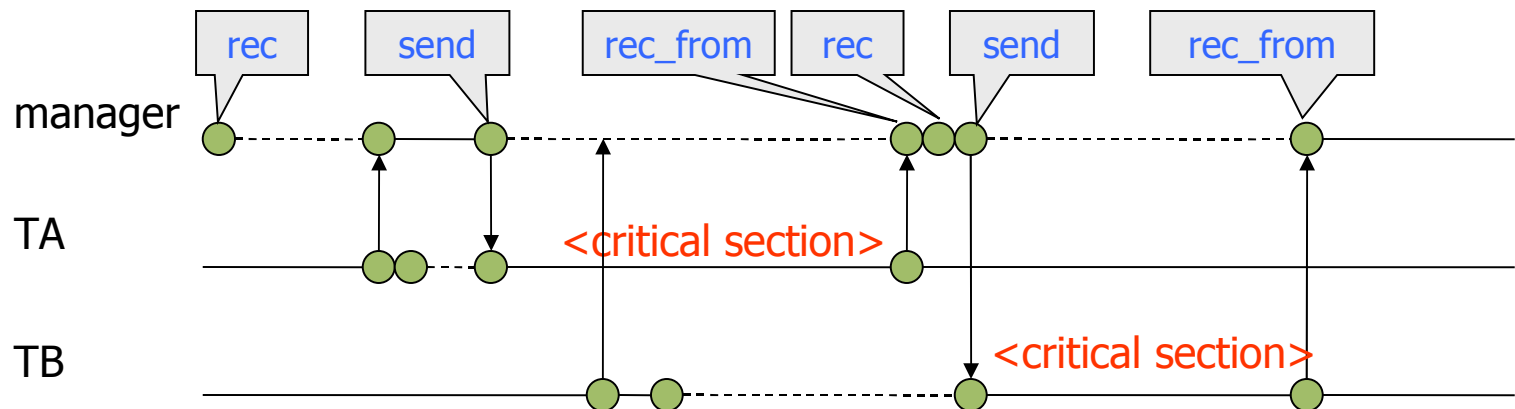




With Async send and sync receive

```
void * manager(void *)  
{  
    thread_t source;  
    int d;  
    while (true) {  
        source = s_receive(&d);  
        a_send(source,d);  
        s_receive_from(source,&d);  
    }  
}
```

```
void * thread(void *)  
{  
    int d;  
    while (true) {  
        a_send(manager, d);  
        s_receive_from(manager, &d);  
        <critical section>  
        a_send(manager, d);  
    }  
}
```





A DESIGN METHODOLOGY WITH MESSAGE PASSING



A different approach

- **Shared memory**

- each resource is a class
- threads ask for services through functions calls
- they synchronize through mutexes and condition variable

- **Message Passing**

- each resource has a manager thread
- threads ask for services through messages and receive response through messages
- they synchronize through blocking receives



Resources and Manager

- For each resource
 - a “manager” thread takes care of executing the services on behalf of the clients/threads
 - general structure of a manager:
 - wait for a message
 - decode the message, execute the service
 - eventually, send back the response
 - The manager sequentializes all services



How to program the data structure?

- Some design considerations
 - Once again, first, design the interface: that is, which functions (services) the threads need to call
 - Mutual exclusion
 - The mutual exclusion is automatically guaranteed by the manager



Synchronization

- As in the shared memory case:
 - Specify the behavior of the functions
 - under which conditions a calling thread should be blocked?
 - Identify all different blocking conditions



Synchronization

- As in the shared memory case:
 - identify unblocking conditions
 - when a blocked thread should be unblocked
 - mark the functions that should unblock those threads
- Putting all together
 - draw a state diagram for the resource
 - STATES: the various states of the resource
 - EVENTS: threads that send the request messages



Example: Bridge

- Problem explanation
- Design the data structure interface
- Identify blocking and unblocking conditions
- Draw the state diagram



Coding Rules

- For each blocking condition
 - a queue of message requests (requests that cannot be completed)
- One (or more) integer variable(s) to code the state
- Write the initialization function, to be called by the manager



The functions

- Finally, code the manager
 - wait for any message, and decode it
 - check the blocking conditions (if any)
 - if the thread has to be blocked,
 - insert the message in the corresponding blocking queue
 - perform the code, change state if necessary
 - if a thread has to be unblocked
 - remove the message from the corresponding blocking queue, and send back a message
- Code the functions
 - build the message, send the message and wait for the response (with a receive)



Exampe of code