Laurea Specialistica in Ingegneria dell'Automazione

Sistemi in Tempo Reale

Giuseppe Lipari

Introduzione alla concorrenza - IV

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 implicitely provides mutual exclusion
  - the source code that is used to implement the mutual exclusion is automatically inserted by the compiler
  - Example: Java language
    - It provides a way to implement monitors through the keyword Synchronized

condition variables

- 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
  - there are 6 ways to implement a monitor construct
    - we will only look at the POSIX approach
      (that is the same used by the MESA language)

CircularArray with monitors

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

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

```
void CircularArray::extract(int &elem) synchronized
{
	while (num== 0) empty.wait();
	elem = array[tail];
	tail = (tail+1)%10;
	num--;
	if (num == 9) full.signal();
}
```

models of concurrency: message passing

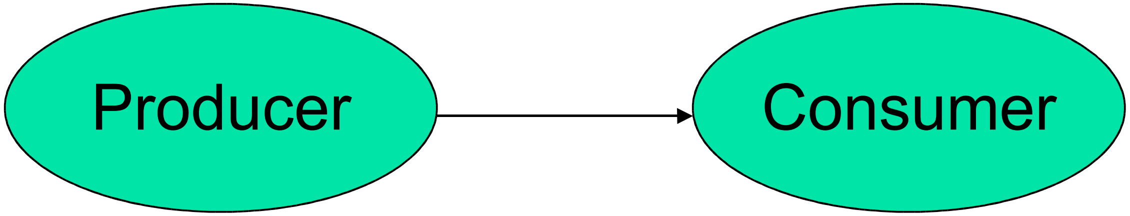
message passing

- 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)



resources and message passing

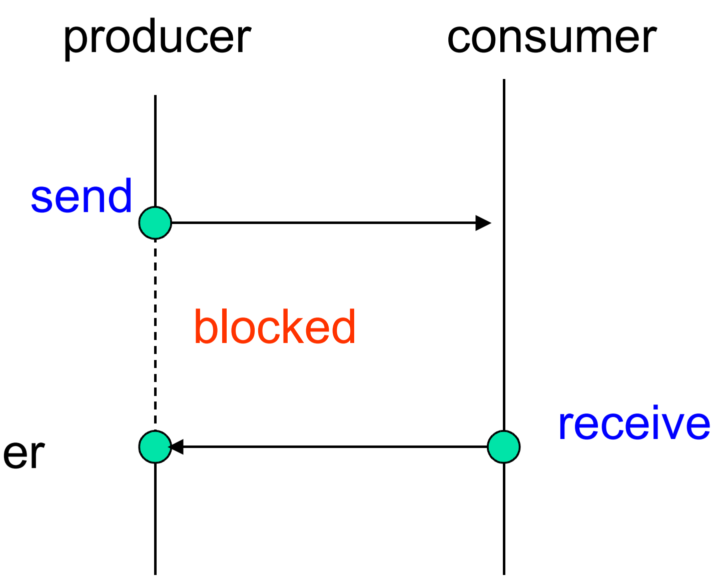
- there are no shared resources in the message passing model
  - all the resources are allocated statically, accessed in a dedicated way
- each resource is handled by a manager process that is the only one that have right to access to a resource
- the consistency of a data structure is guaranteed by the manager process
  - there is no more competition, only cooperation!!!

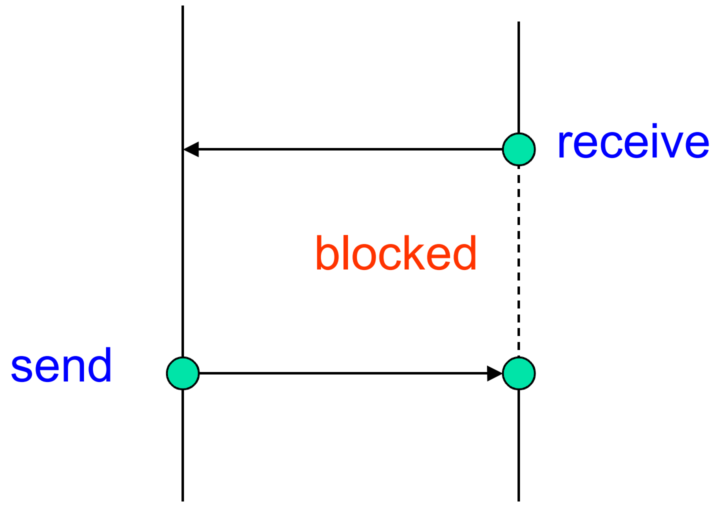
synchronous communication

- synchronous send/receive
  - no buffers!

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

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



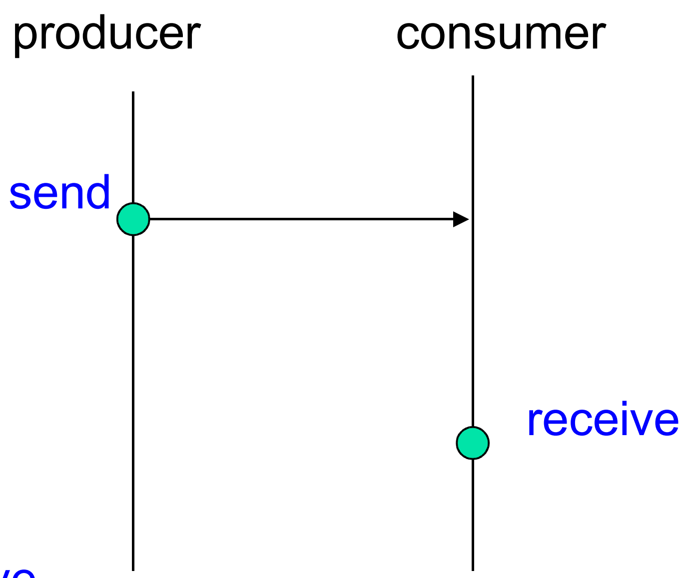


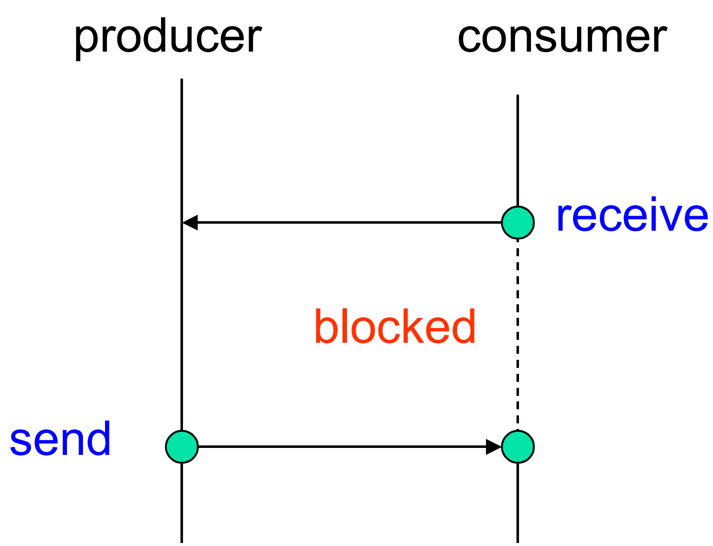
async send/ sync receive

- asynchronous send / synchronous receive
  - there is probably a send buffer somewhere

producer:
  a_send(consumer, d);

consumer:
  s_receive(producer, &d);



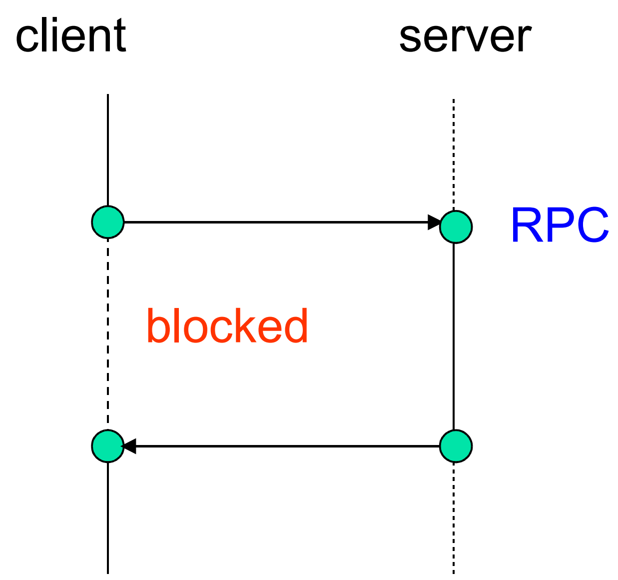


asymmetric receive

- symmetric receive
  - receive(source, &data);
  - the programmer wants a message from a given producer
- asymmetric receive
  - source = receive(&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

remote procedure call

- in a client server system, a client wants to request an action to a server
  - that is typically done using a remote procedure call (RPC)



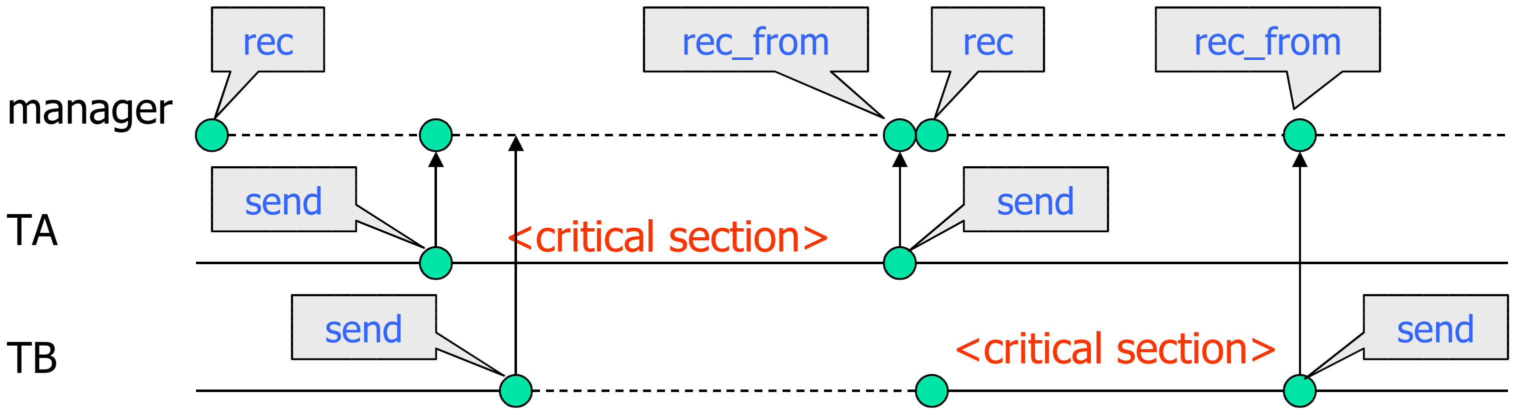
message passing systems

- 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 resourec 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);
      }
}
```

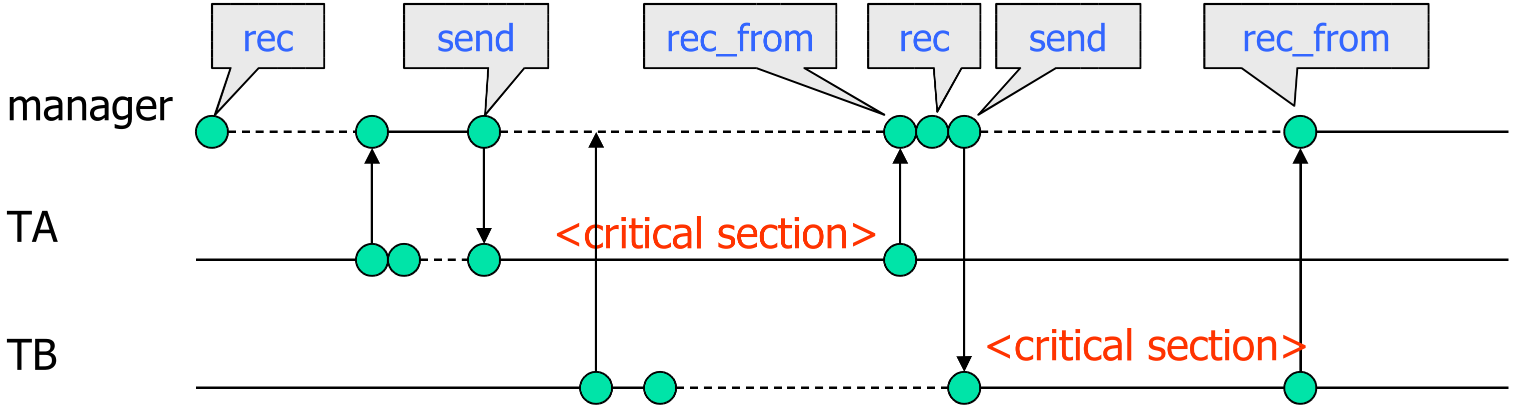
```
void * thread(void *)
{
      int d;
      while (true) {
          s_send(manager, d);
          <critical section>
          s_send(manager, d);
      }
}
```



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);
    }
}
```



deadlock

deadlock and livelock

- deadlock is the situation in which a group of threads are permanently blocked waiting for some resource
- deadlock can happen in many subtle cases
  - example: dining philosophers
- here we will study ways of avoiding deadlock situations

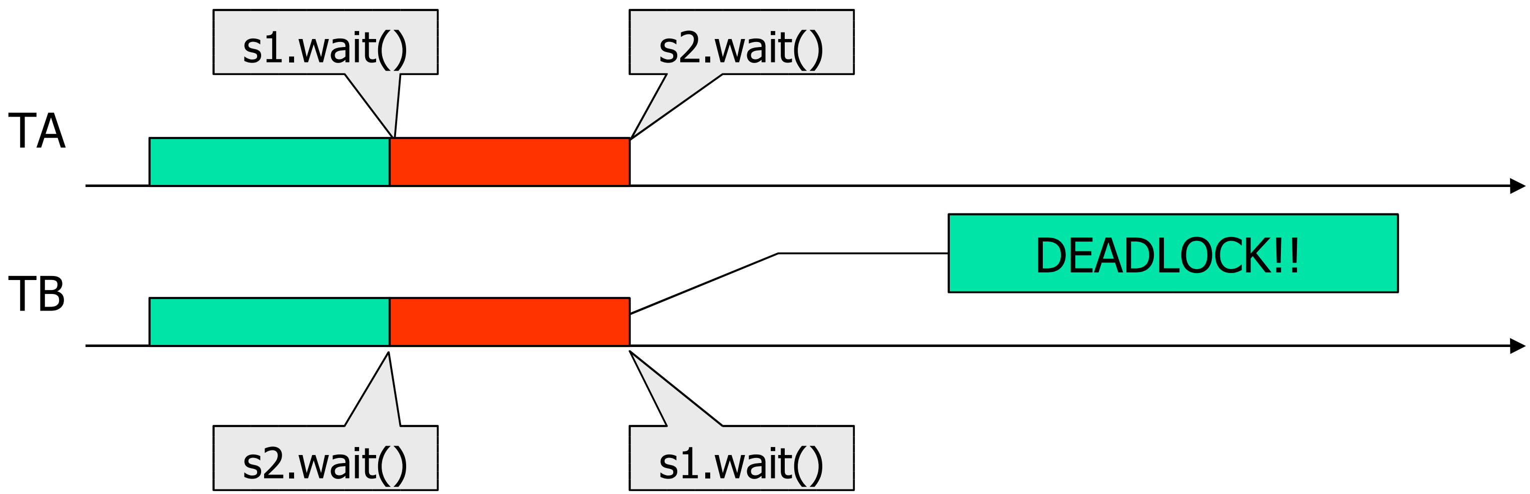
- livelock is the situation where a group of threads tries to get some resource, but they never succeed
  - the idea is that they have a non-blocking wait
  - example: dining philosophers with non-blocking wait
-
- deadlocks and livelocks can be total or partial

example of deadlock

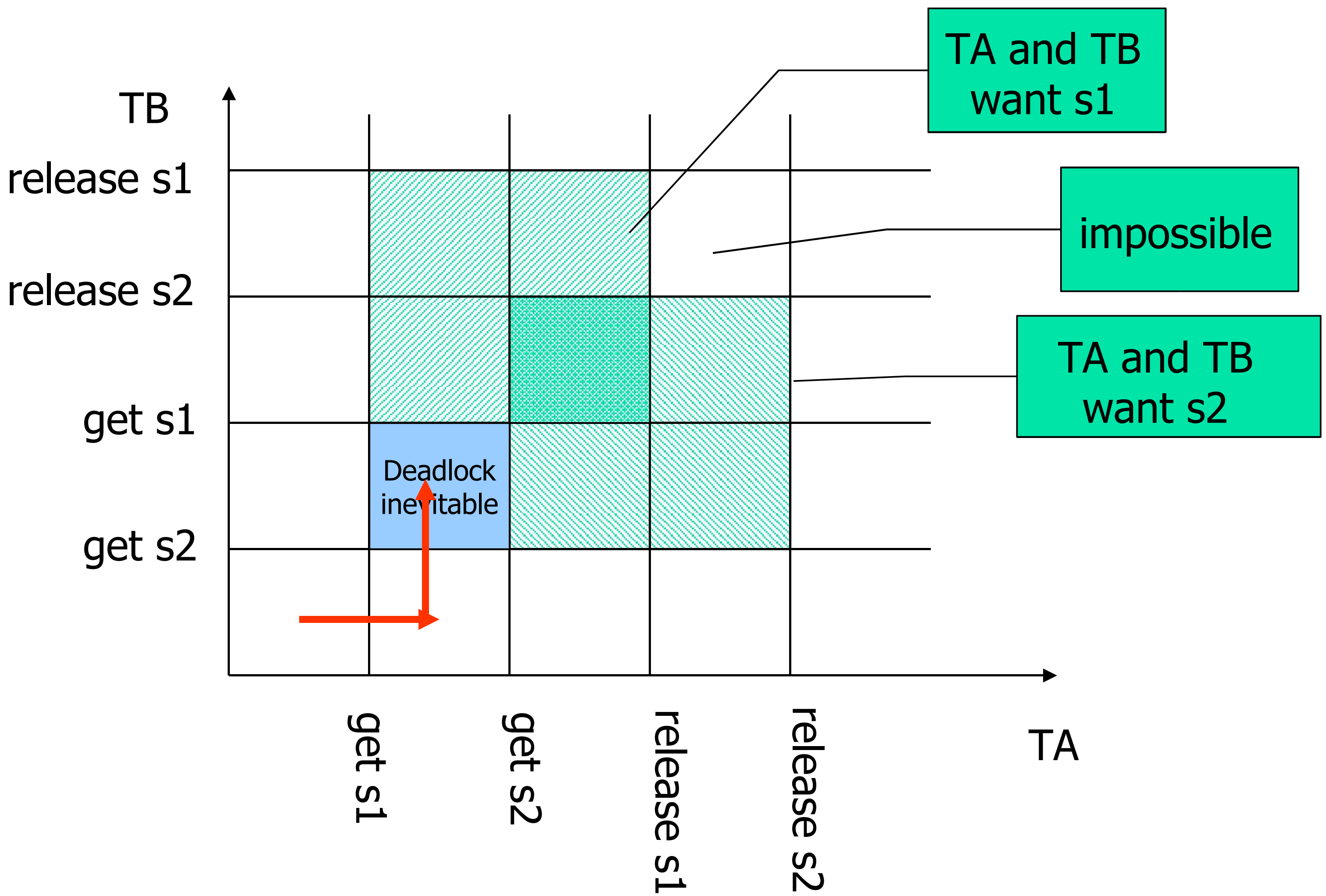
```
Semaphore s1(1);
Semaphore s2(1);
```

```
void *threadA(void *)
{
    ...
    s1.wait();
    s2.wait();
    ...
    s1.signal();
    s2.signal();
    ...
}
```

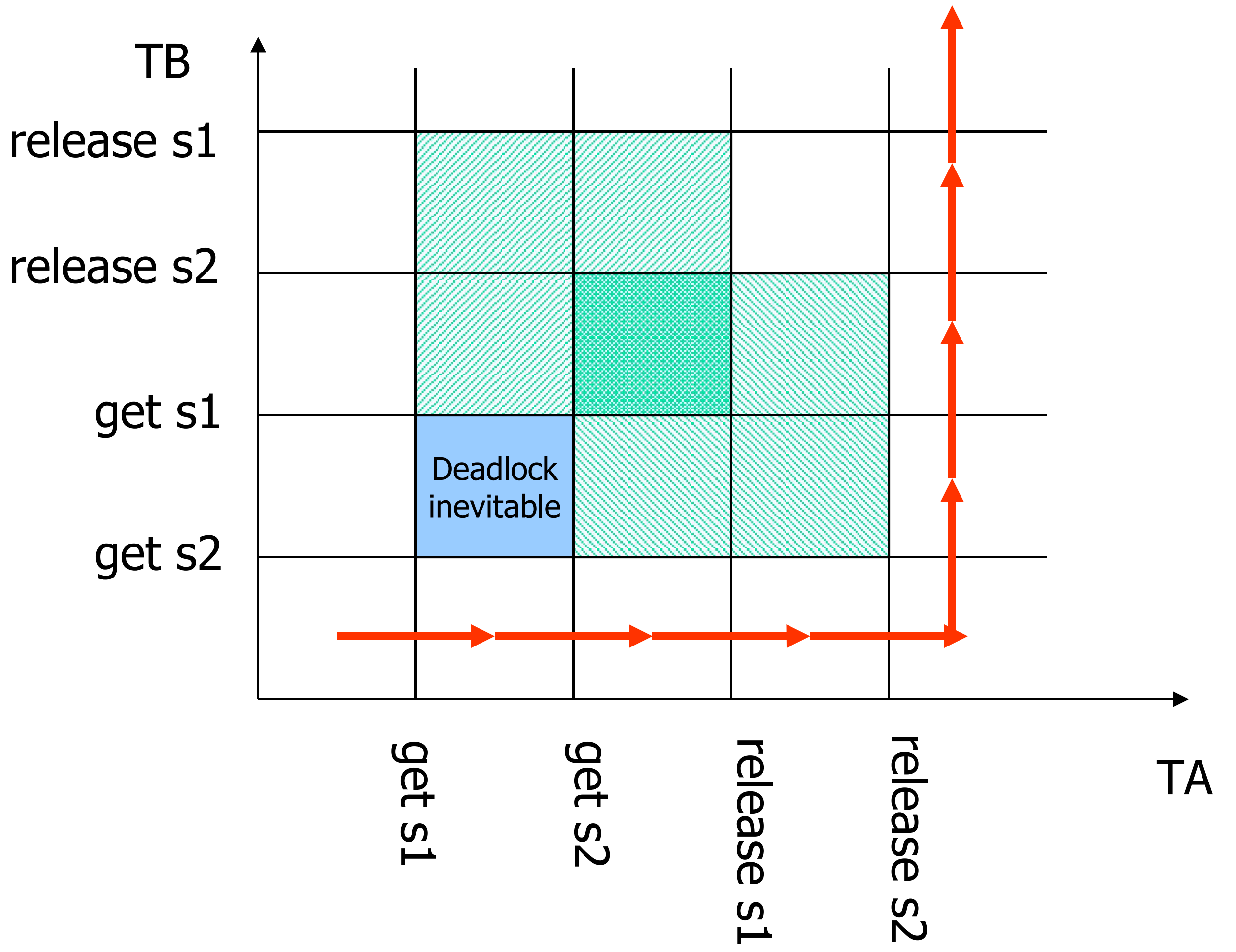
```
void *threadB(void *)
{
    ...
    s2.wait();
    s1.wait();
    ...
    s2.signal();
    s1.signal();
    ...
}
```



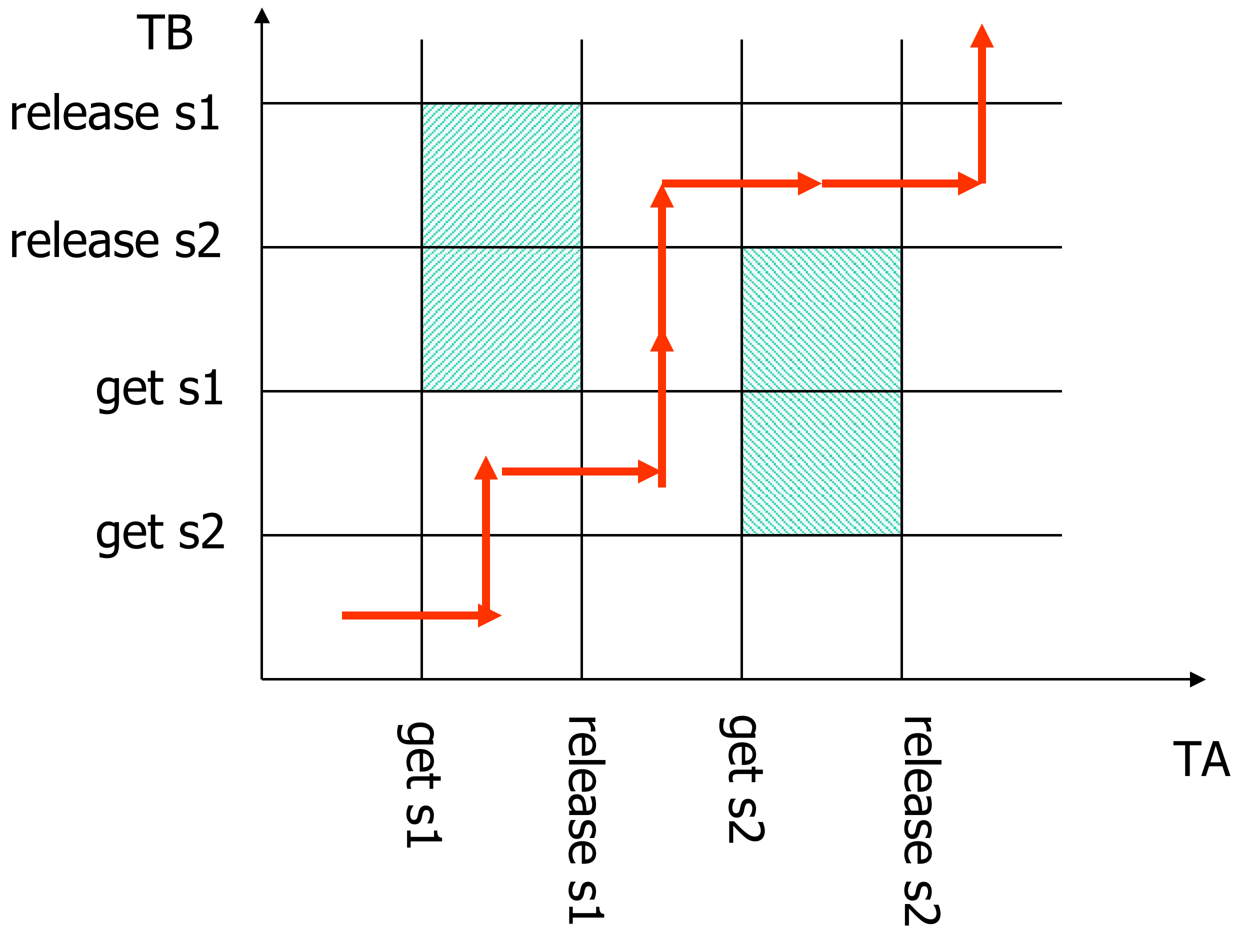
graphical situation



graphical situation



example with no deadlock



other examples of deadlock

- bad situations can happen even when the resource is not "on-off"
- consider a memory allocator
  - suppose that the maximum memory allocable is 200 Kb

```
void * threadA(void *)
{
    request(80kb);
    ...
    request(60kb);
    ...
    release(140kb);
}
```

```
void * threadB(void *)
{
    request(70kb);
    ...
    request(80kb);
    ...
    release(150kb);
}
```

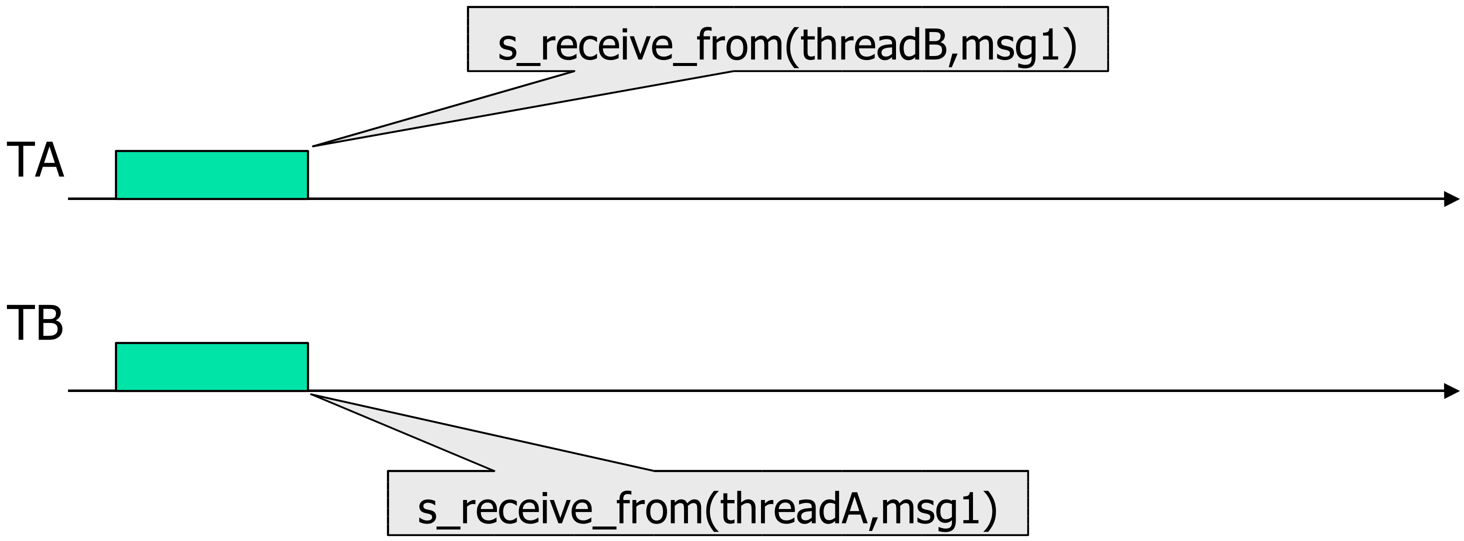
consumable and reusable resources

- reusable resources
  - it can be safely used by only one thread at time and is not depleted by the use
  - threads must request the resource and later release it, so it can be *reused* by other threads
  - examples are processor, memory, semaphores, etc.
- consumable resources
  - it is created and destroyed dynamically
  - once the resource is acquired by a thread, it is immediately “destroyed” and cannot be reused
  - examples are messages in a FIFO queue, interrupts, I/O data, etc.

deadlock with consumable resources

```
void *threadA(void *)
{
    s_receive_from(threadB, msg1);
    ...
    s_send(threadB, msg2);
    ...
}
```

```
void *threadB(void *)
{
    s_receive_from(threadA, msg1);
    ...
    s_send(threadA, msg2);
    ...
}
```



conditions for deadlock

- three conditions
  - dynamic allocation of dedicated resources (in mutual exclusion)
    - only one process may use the resource at the same time
  - hold and wait
    - a process may hold allocated resources when it blocks
  - no preemption
    - the resource cannot be revoked
      (note: the CPU is a revokable resource)
- if the three above conditions hold and
  - circular wait
    - a closed chain of threads exists such that each thread holds at least one resources needed by the next thread in the chain
- then a deadlock can occur!
- these are necessary and sufficient conditions for a deadlock

how to solve the problem of deadlock

- the basic idea is to avoid that one of the previous conditions hold
- to prevent deadlock from happening we can distinguish two class of techniques
  - static: we impose strict rules in the way resources may be requested so that a deadlock cannot occur
  - dynamic: dynamically, we avoid the system to enter in dangerous situations
- three strategies
  - deadlock prevention (static)
  - deadlock avoidance (dynamic)
  - deadlock detection (dynamic)

deadlock: something that cannot be changed

- there is something that cannot be disallowed, because some behavior is forced by the interaction between the different concurrent activities
  - mutual exclusion
  - communication
- there is nothing we can do!

deadlock prevention: three methods

- take all the resources at the same time
- preempt a thread and give the resource to someone else
- resource allocation in a given order

deadlock prevention: conditions

- hold and wait
  - we can impose the tasks to take all resources at the same time with a single operation
  - this is very restrictive! even if we use the resource for a small interval of time, we must take it at the beginning!
  - reduces concurrency

deadlock prevention: conditions

- no preemption
  - this technique can be done only if we can actually suspend what we are doing on a resource and give it to another thread
  - for the "processor" resource, this is what we do with a thread switch!
  - for other kinds of resources, we should "undo" what we were doing on the resource
  - this may not be possible in many cases!

deadlock prevention: conditions

- circular wait
  - This condition can be prevented by defining a linear ordering of the resources
  - for example: we impose that each thread must access resources in a certain well-defined order

```
void *threadA(void *)
{
      ...
      s1.wait();
      s2.wait();
      ...
      s1.signal();
      s2.signal();
      ...
}
```

```
void *threadB(void *)
{
      ...
      s2.wait();
      s1.wait();
      ...
      s2.signal();
      s1.signal();
      ...
}
```

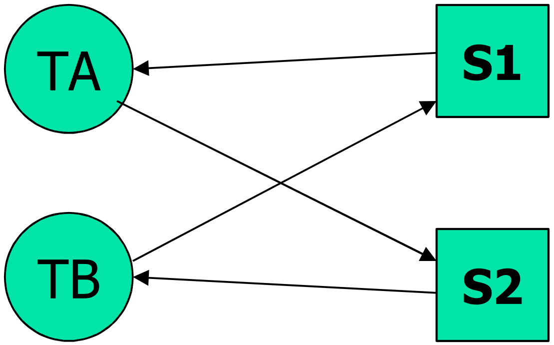
deadlock prevention: why this strategy works?

- let us define a oriented graph
  - a vertex can be
    - a thread (round vertex)
    - a resource (square vertex)
  - an arrow from a thread to a resource denotes that the thread requires the resource
  - an arrow from a resource to a thread denotes that the resource is granted to the thread
- deadlock definition
  - a deadlock happens if at some point in time there is a cycle in the graph

deadlock prevention: graph

```
void *threadA(void *)
{
    ...
    s1.wait();
    s2.wait();
    ...
    s1.signal();
    s2.signal();
    ...
}
```

```
void *threadB(void *)
{
    ...
    s2.wait();
    s1.wait();
    ...
    s2.signal();
    s1.signal();
    ...
}
```



deadlock prevention: theorem

- if all threads access resources in a given order, a deadlock cannot occur
- proof:
  - by contradiction.
  - suppose that a deadlock occurs. then, there is a cycle.
  - by hypothesis all threads access resources by order
  - therefore, each thread is blocked on a resource that has an order number grater than the resources it holds.
  - starting from a thread and following the cycle, the order number of the resource should always increase. however, since there is a cycle, we go back to the first thread. then there must be a thread T that holds a resource Ra and requests a Resource Rb with Ra < Rb
  - this is a contradiction!

deadlock avoidance

- this technique consists in monitoring the system to avoid deadlock
  - we check the behaviour of the system
  - if we see that we are going into a dangerous situation, we block the thread that is doing the request, even if the resource is free
  - that algorithm is called the Banker's algorithm
    - we skip it :-)

deadlock detection

- in this strategy, we monitor the system to check for deadlocks *after* they happen
  - we look for cycles between threads and resources
  - how often should we look?
    - it is a complex thing to do, so it takes precious processing time
    - it can be done not so often
    - a good point to do that is when we lock (but it is computationally expensive)
  - once we discover deadlock, we must *recover*
- the idea is to
  - kill some blocked thread
  - return an error in the wait statement if there is a cycle
    - that is the POSIX approach

recovery strategies

1. abort all threads
   - used in almost all OS. the simplest thing to do.
2. check point
   - all threads define safe *check points*. when the OS discover a deadlock, all involved threads are restarted to a previous check point
     - problem. they can go in the same deadlock again!
3. abort one thread at time
   - threads are aborted one after the other until deadlock disappears
4. successively preempt resources
   - preempt resources one at time until the deadlock disappears