Scuola Superiore Sant'Anna

Real-Time System Laboratory

Deadlock and Starvation

Giuseppe Lipari

Deadlock

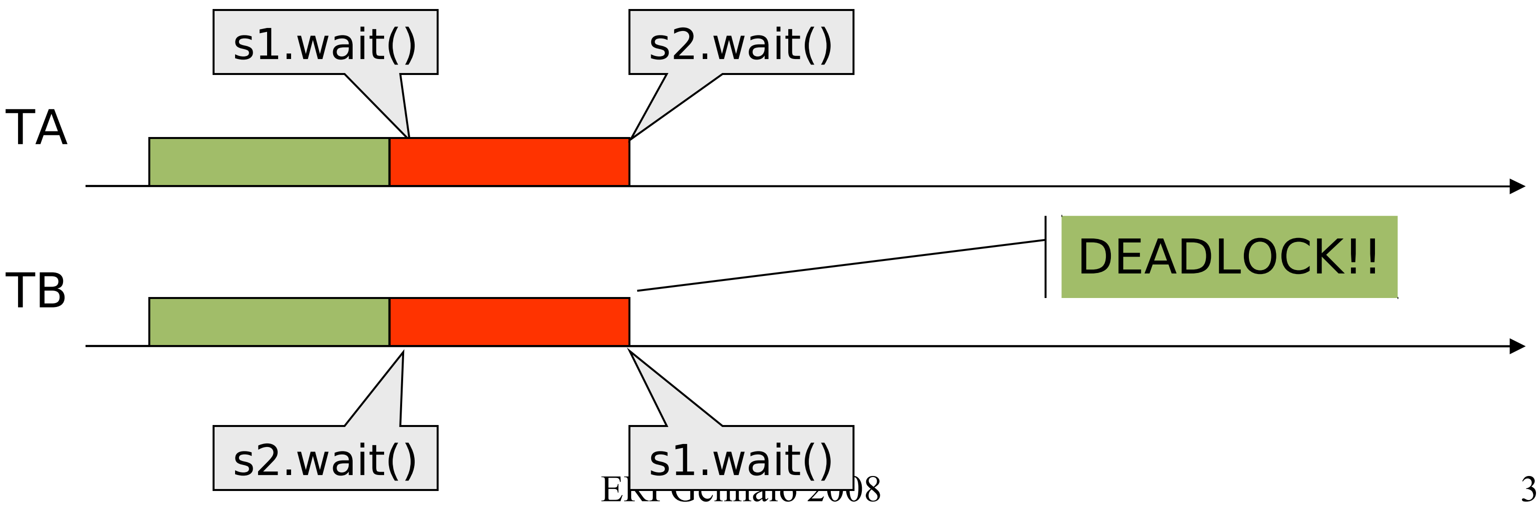
- Deadlock is the situation in which a group of threads are permanently blocked waiting for some resource
- Deadlock can happen in many subtle cases
- Here we will study ways of avoiding deadlock situations

Example of deadlock

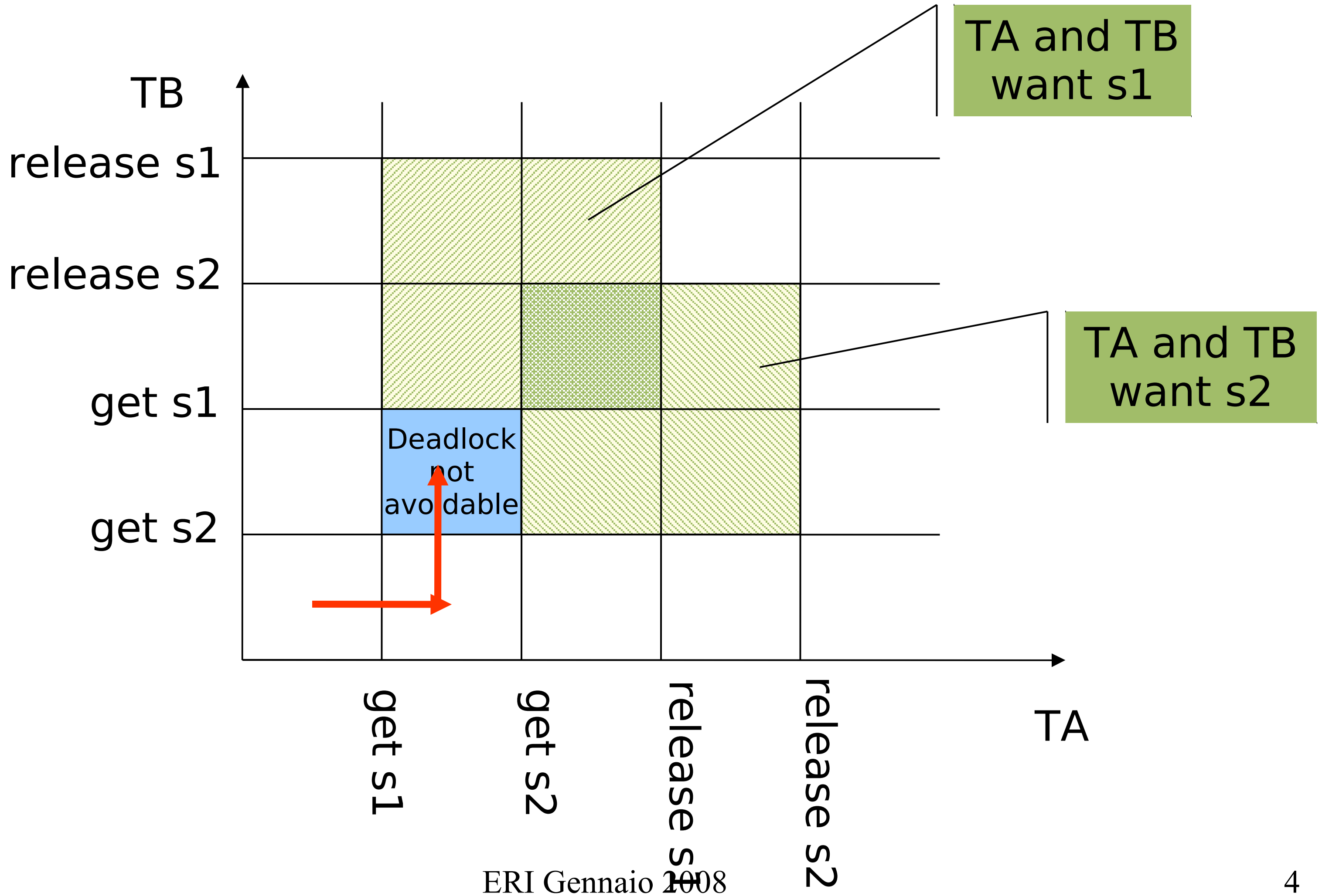
```
Semaphore s1(1);
Semaphore s2(1);
```

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

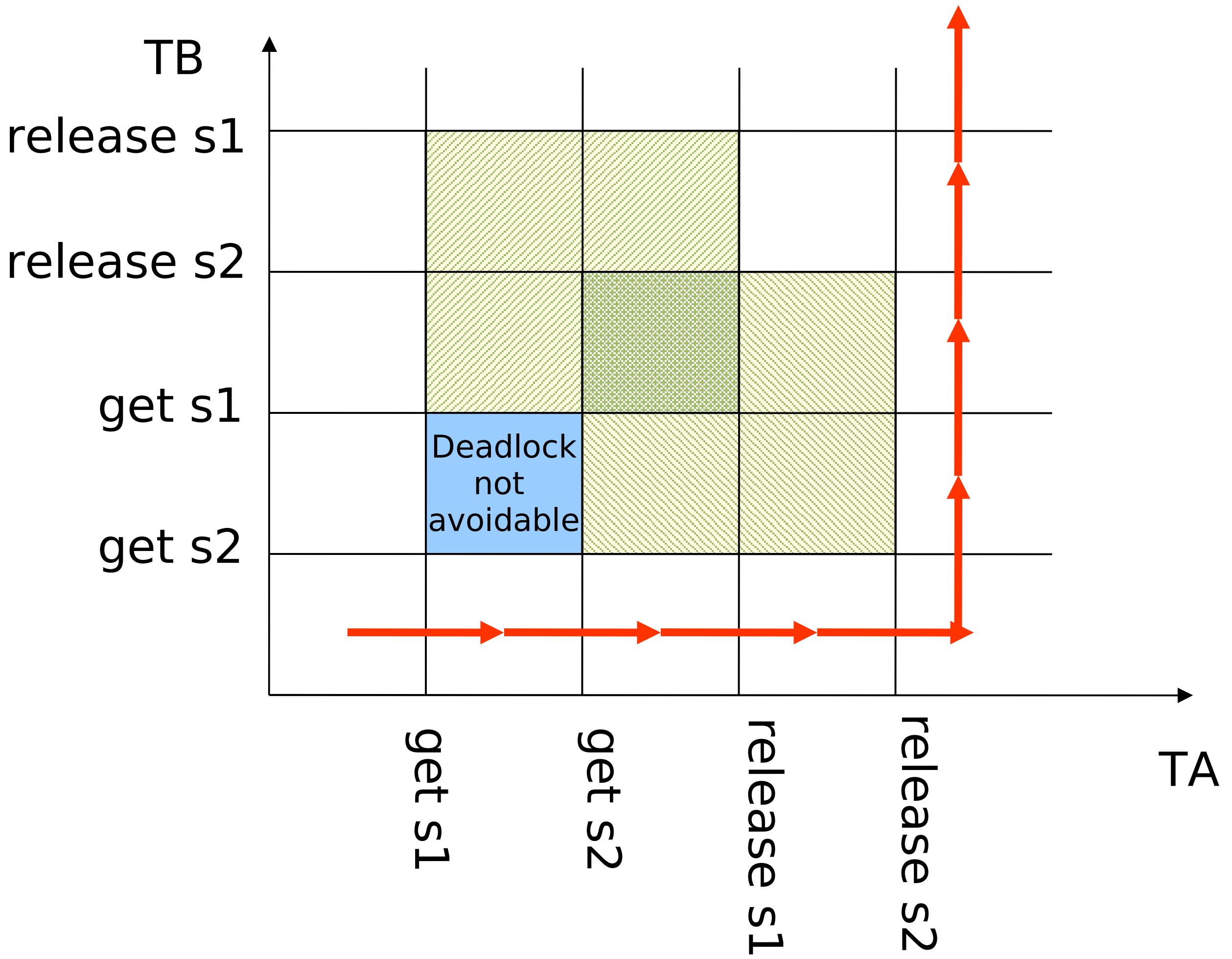
```
void *threadB(void *)
{
        ...
        s2.wait();
        s1.wait();
        ...
        s2.signal();
        s1.signal();
        ...
}
```



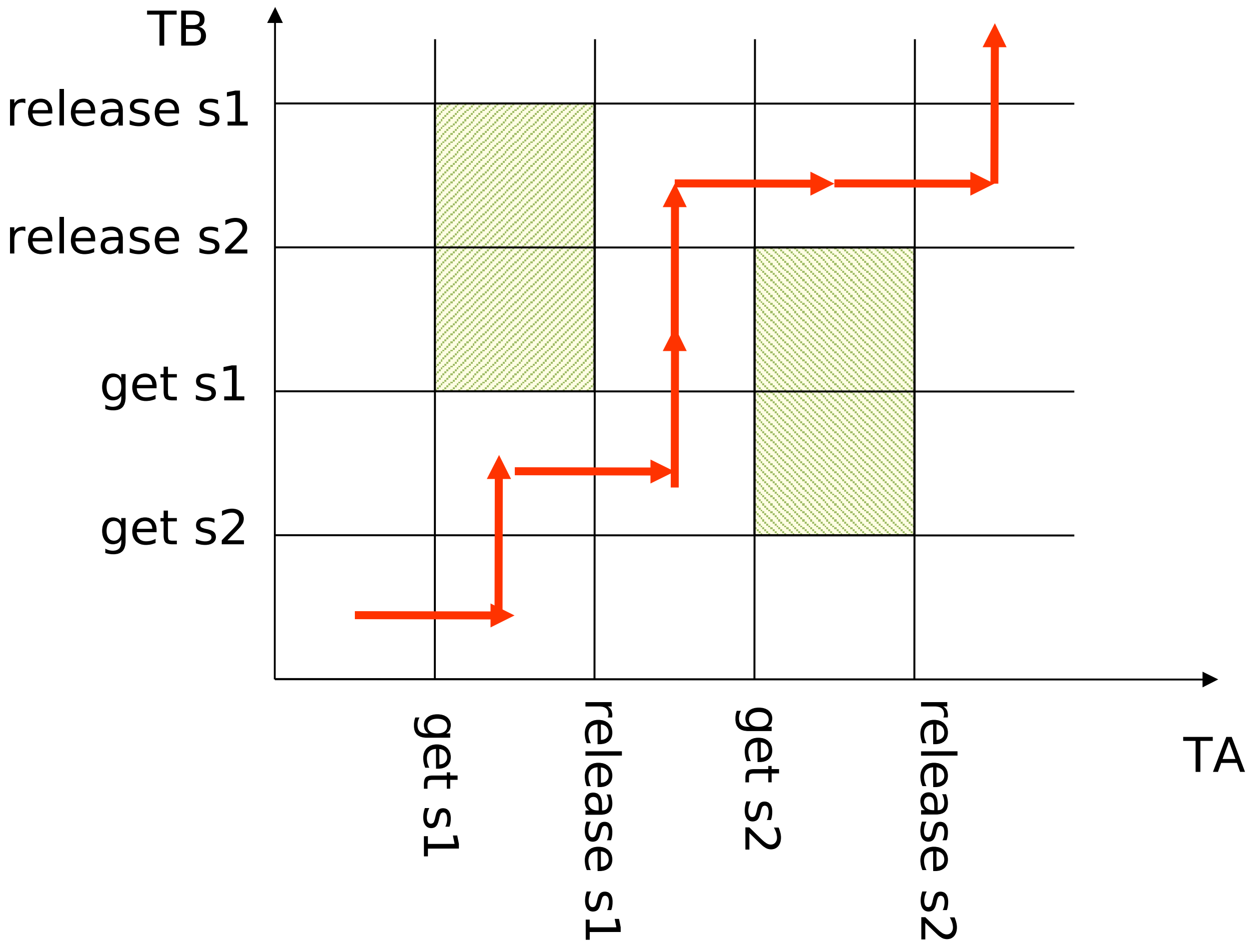
Graphical situation



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);
}
```

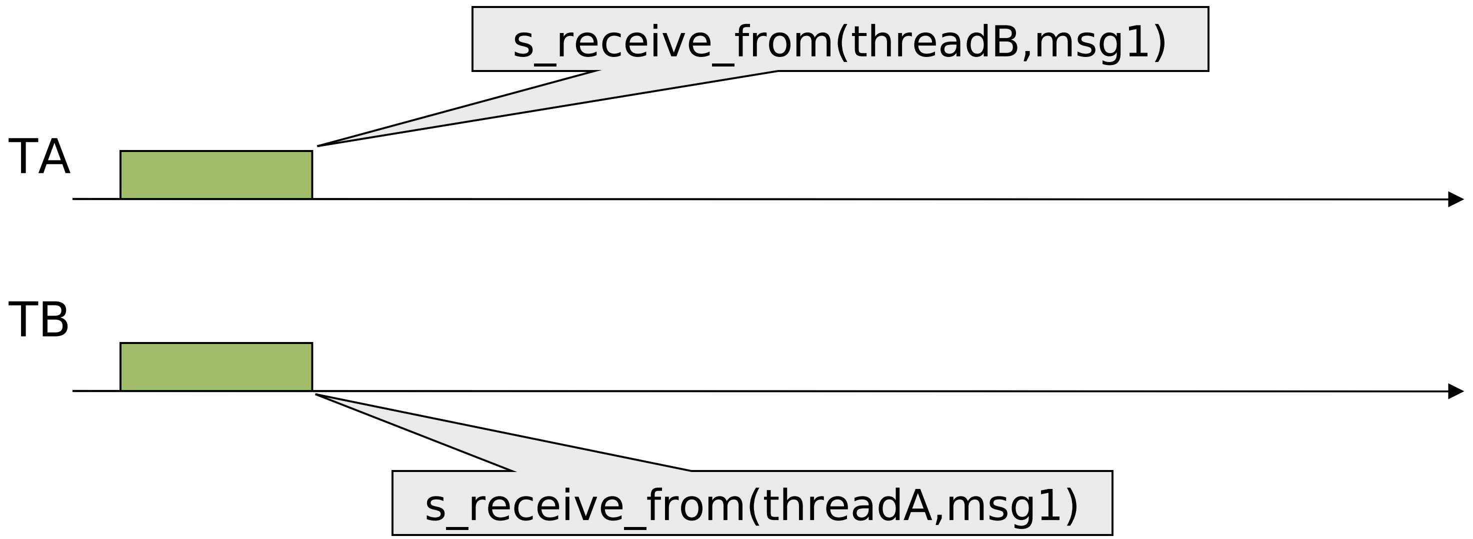
Consumable and reusable resources

- Reusable resources
  - It can be safely used by only one thread at time and is nod 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
  - 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
- 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

- 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
- The basic idea is to avoid that one of the previous conditions hold
- Three strategies
  - Deadlock prevention (static)
  - Deadlock avoidance (dynamic)
  - Deadlock detection (dynamic)

Conditions

- Mutual exclusion
  - This cannot be disallowed. If a resource must be accessed in mutual exclusion, there is nothing else we can do!
- 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

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!

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();
     ...
}
```

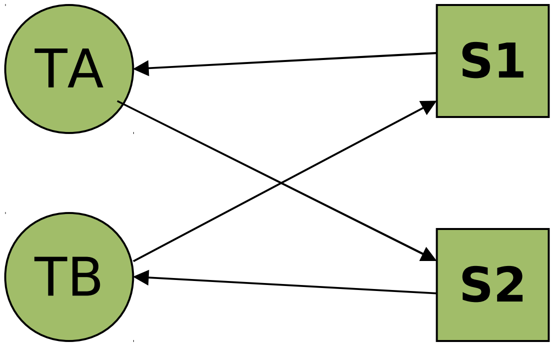
Why this strategy works?

- 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

Graph

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

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



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!

Hierarchies of locks

- To use the previous technique efficiently
  - divide the software into layers (usually the SW is already in layers!)
  - Each layer defines its own locks and semaphores
  - A layer can only call functions from the same layer or from lower layers
  - In this way the resource ordering is automatically guaranteed

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

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
  - Once we discover deadlock, we must *recover*
  - The idea is to kill some blocked thread

Recovery

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. The 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