Semaphore Primitives

STORK	Java	Java 1.5 java.util.concurrent Class Semaphore	Regler.jar Class Semaphore	Other common names
wait	NA	acquire	take	P
signal	NA	release	give	V

Monitor Primitives

STORK	Java	Java 1.5 java.util.concurrent Class Lock
enter	Provided implicitly by synchronized methods and blocks	lock
leave	Provided implicitly by synchronized methods and blocks	unlock

Condition Variable Primitives

STORK	Java	Java 1.5 java.util.concurrent Class Condition	Regler.jar Class ConditionVariable
await	wait	await	cvWait
cause	notifyAll	signalAll	cvNotifyAll
NA	notify	signal	cvNotify

Lecture 4: Synchronization & Communication - Part 2

[RTCS Ch. 4]

- Deadlock
- Priority Inversion & Inheritance
- Mailbox Communication
- Communication with Objects

Deadlock

Improper allocation of common resources may cause deadlock.

Example:

- A and B both need access to two common resources protected by the semaphores R1 and R2 (initialized to 1).

```
  Process A         Process B
...               ...
Wait(R1);         Wait(R2);
Wait(R2);         Wait(R1);
...               ...
Signal(R2);       Signal(R1);
Signal(R1);       Signal(R2);
...               ...
```

May cause deadlock.

Same situation can occur in Java with synchronized methods.

Deadlock handling

- Deadlock prevention (during design)
  - e.g. hierachical resource allocation
- Deadlock avoidance (at run-time)
  - e.g. priority ceiling protocol
- Deadlock detection and recovery (at run-time)

Necessary conditions

Must hold for a deadlock due to resource handling to occur.

1. *Mutual exclusion*: only a bounded number of processes can use a resource at a time.
2. *Hold and wait:* processes must exist which are holding resources while waiting for other resources.
3. *No preemption:* resources can only be released voluntarily by a process
4. *Circular wait:* a circular chain of processes must exist such that each process holds a resource that is requested by the next process in the chain.

Deadlock prevention

Remove one of the four conditions:

- Mutual exclusion – usually unrealistic
- Hold and wait – require that the processes preallocate all resources before execution or at points when they have no other resources allocated
- No preemption – forced resource deallocation
- Circular wait condition – ensure that resources always are allocated in a fixed order

Hierarchical resource allocation

Pyramidical resource allocation

A resource belongs to one of the classes R_i where $i = 1 \ldots n$.

A process must reserve resources in this order.

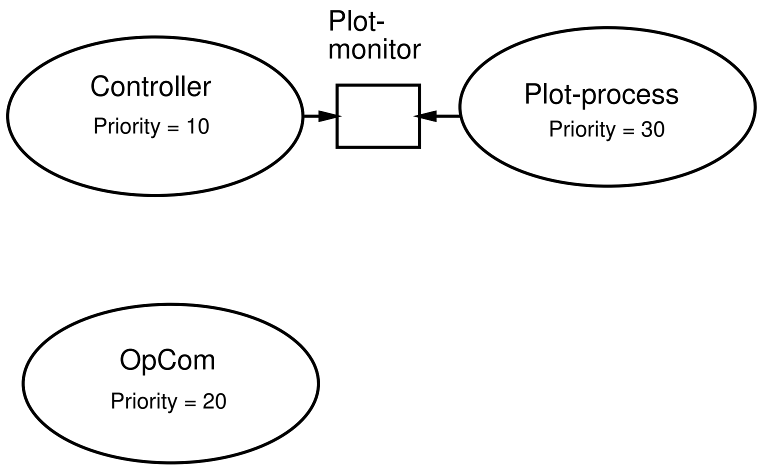
If it has a resource in one class it may not reserve a resource in a lower class.

```
  Process A        Process B


...              ...
Wait(R1);        Wait(R1);
Wait(R2);        Wait(R2);
...              ...
Signal(R2);      Signal(R2);
Signal(R1);      Signal(R1);
...              ...
```

Priority Inversion

A situation where a high-priority process becomes blocked by a lower priority process and there is no common resource involved between the two processes.



1. Plot-process enters PlotMonitor.
2. An interrupts causes OpCom to execute.
3. An interrupt causes Controller to execute.
4. Controller tries to enter PlotMonitor

Controller is indirectly blocked by OpCom.

Solutions:

- Priority Inheritance
- Priority Ceiling Protocol
- Immediate Inheritance

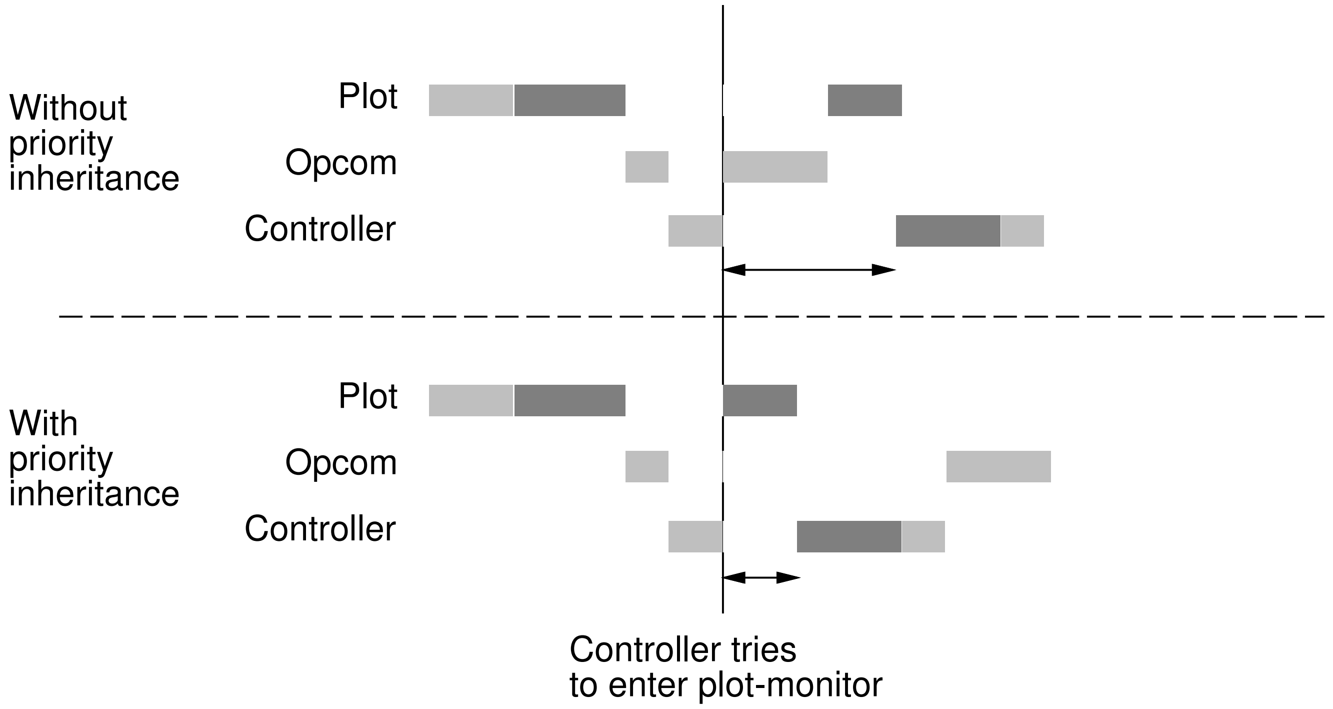
Priority Inheritance

If, during execution of `Enter`, the monitor is occupied then the priority of the process holding the monitor is raised to the priority of the calling process.

The priority is reset in `Leave`.

(The monitor primitives in the STORK kernel behave in this way)

Priority Inheritance


Without priority inheritance
Plot
Opcom
Controller
With priority inheritance
Plot
Opcom
Controller
Controller tries to enter plot-monitor

Plot:	low priority
Opcom:	medium priority
Controller:	high priority

▨	executing
■	executing inside plot-monitor
	suspended by other task

Example: Mars Pathfinder 1997

After a while the spacecraft experienced total system resets, resulting in losses of meteorological data. Reason:

- A mutex-protected shared memory area for passing information
- A high priority bus management task, frequently passing data in and out
- An infrequent data gathering task at low priority, entering data into the memory
- A third communication task at medium priority, not accessing the shared memory
- Occasionally, the situation arised where the mutex was held by the low priority task, the high priority task was blocked on the mutex, and the medium priority task was executing, preventing the low priority task from leaving the mutex
- The classical priority inversion situation

Solution:

- VxWorks from Wind River Systems
- binary mutex semaphores with an optional initialization argument that decides if priority inheritance should be used or not
- upload of code that modified the symbol tables of the Pathfinder so that priority inheritance was used

The Priority Ceiling Protocol

L. Sha, R. Rajkumar, J. Lehoczky, *Priority Inheritance Protocols: An Approach to Real-Time Synchronization,* IEEE Transactions on Computers, Vol. 39, No. 9, 1990

Restrictions on how we can lock (Wait, EnterMonitor) and unlock (Signal, LeaveMonitor) resources:

- a task must release all resources between invocations
- the computation time that a task i needs while holding semaphore s is bounded. $cs_{i,s}$ = the time length of the critical section for task i holding semaphore s
- a task may only lock semaphores from a fixed set of semaphores known a priory. $uses(i) \ = $ the set of semaphores that may be used by task i

The protocol:

- the *ceiling* of a semaphore, $ceil(s)$, is the priority of the highest priority task that uses the semaphore
- notation: $pri(i)$ is the priority of task i
- At run-time:
  - if a task i wants to lock a semaphore s, it can only do so if $pri(i)$ is **strictly higher** than the ceilings of all semaphores currently locked by **other** tasks
  - if not, task i will be blocked (task i is said to be blocked on the semaphore, S^*, with the highest priority ceiling of all semaphores currently locked by other jobs and task i is said to be blocked by the task that holds S^*)
  - when task i is blocked on S^*, the task currently holding S^* inherits the priority of task i

Properties:

- deadlock free
- a given task i is delayed at most once by a lower priority task
- the delay is a function of the time taken to execute the critical section

Deadlock free

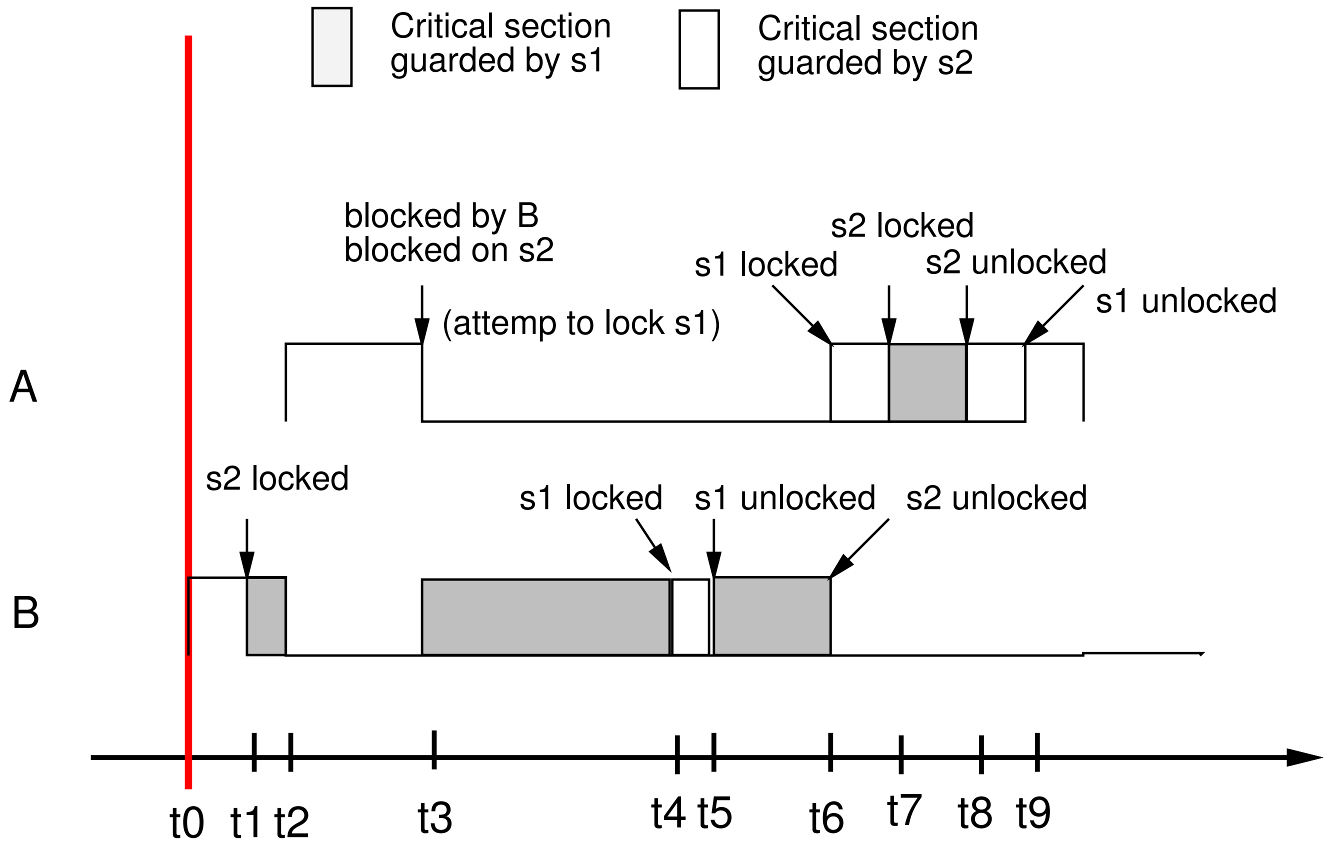
Example:

Task name	T	Priority
A	50	10
B	500	9

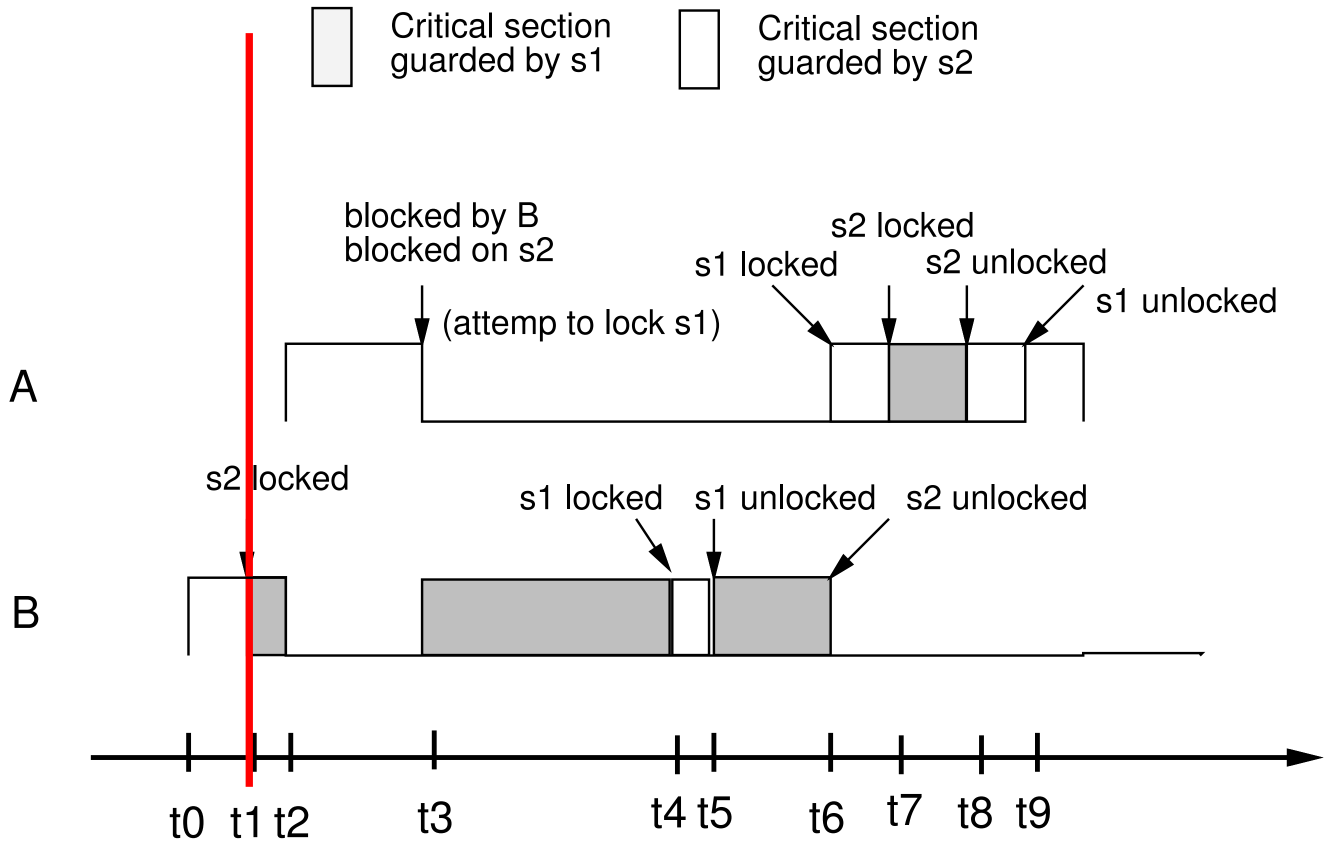
```
Task A       Task B

lock(s1)     lock(s2)
lock(s2)     lock(s1)
...          ...
unlock(s1)   unlock(s1)
unlock(s2)   unlock(s2)
```

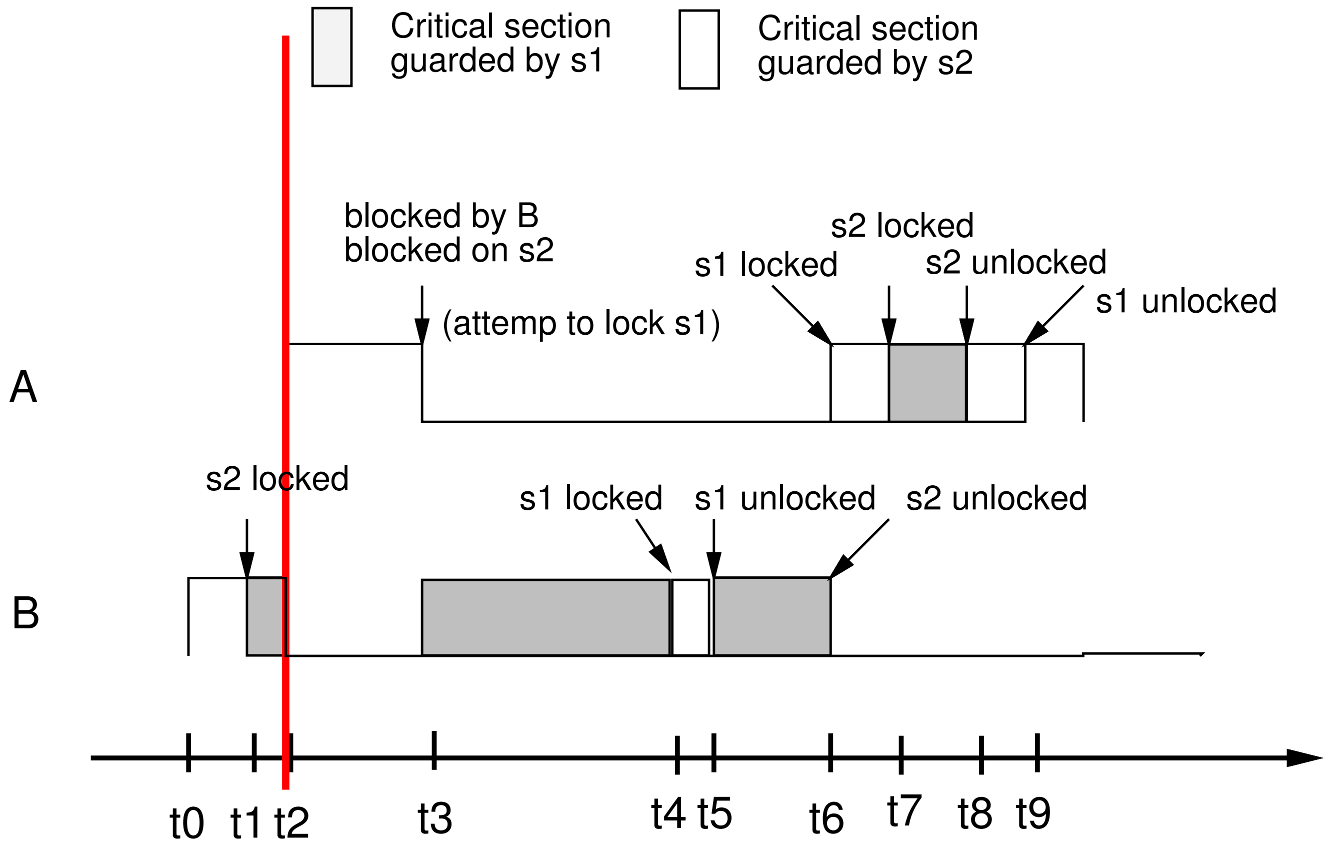
$$ceil(s_1) = 10, ceil(s_2) = 10$$



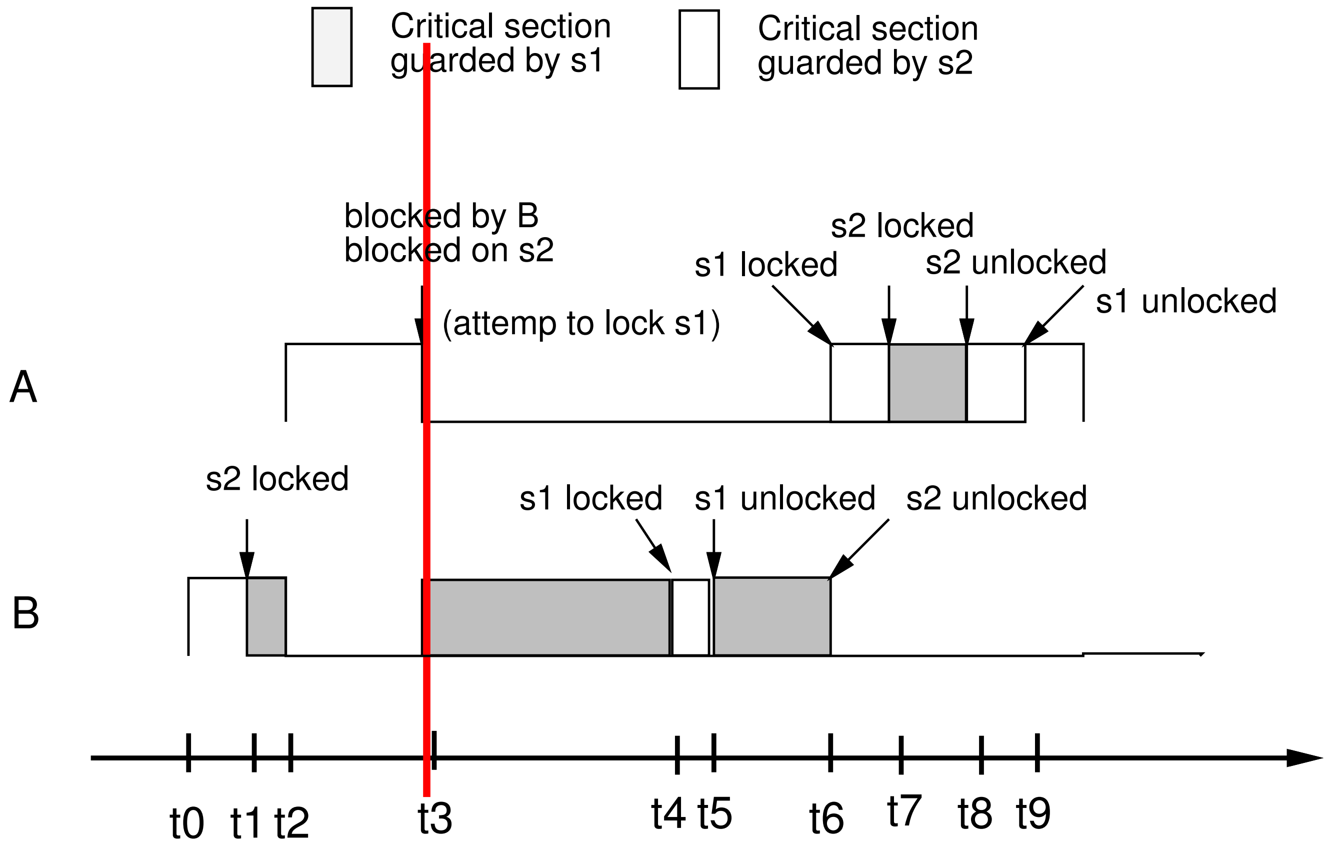
- t_0: B starts executing



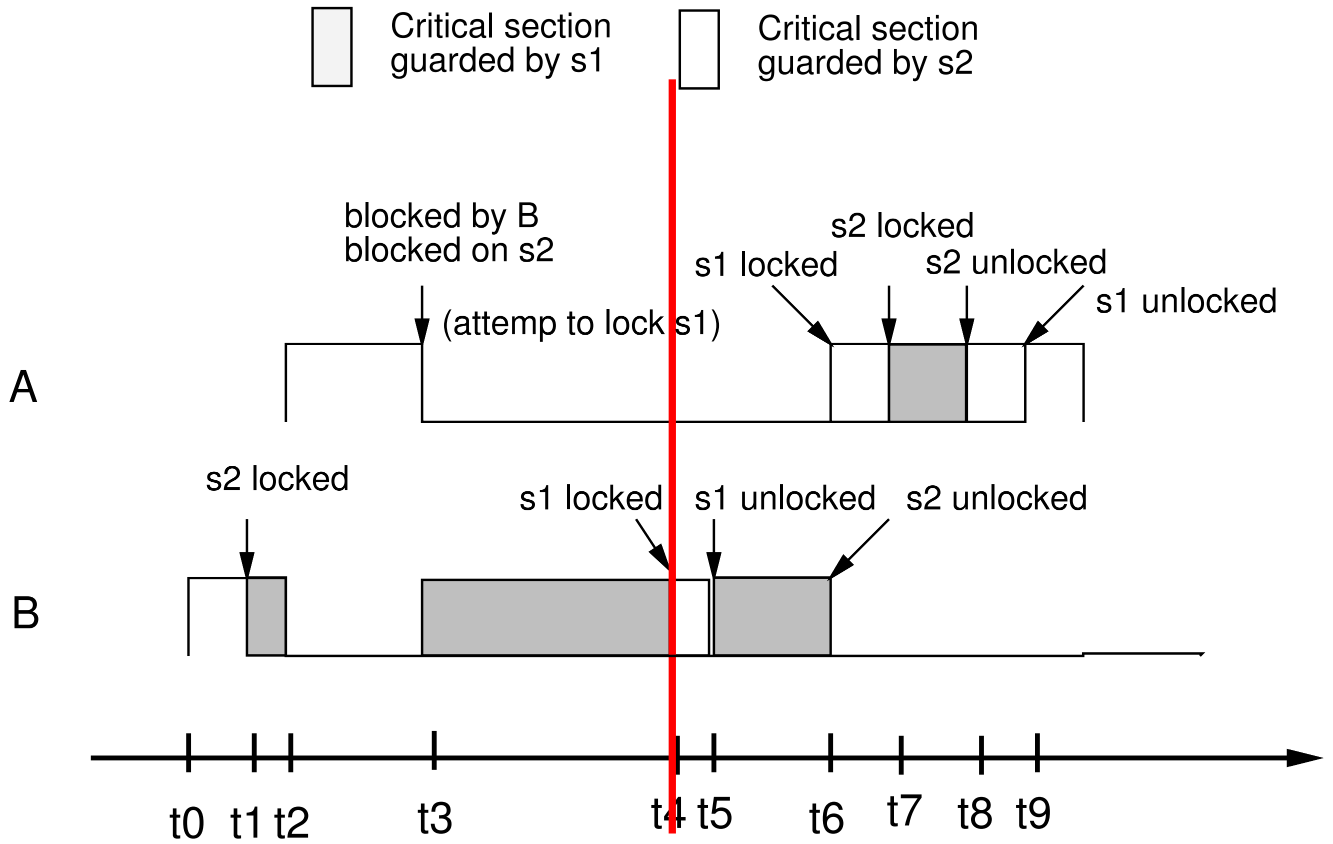
- t_1: B attempts to lock s_2. It succeeds since no lock is held by another task.



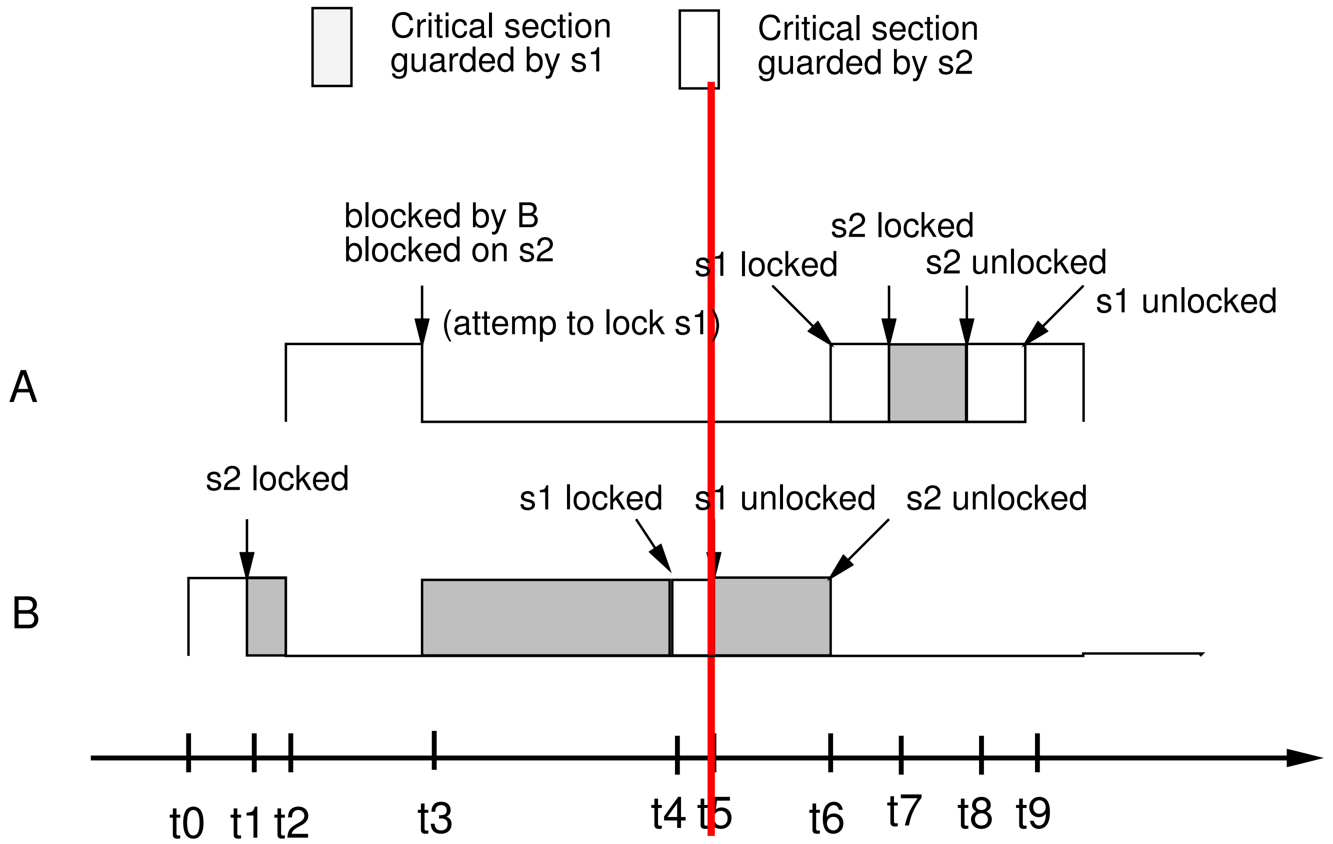
- t_2: A preempts B



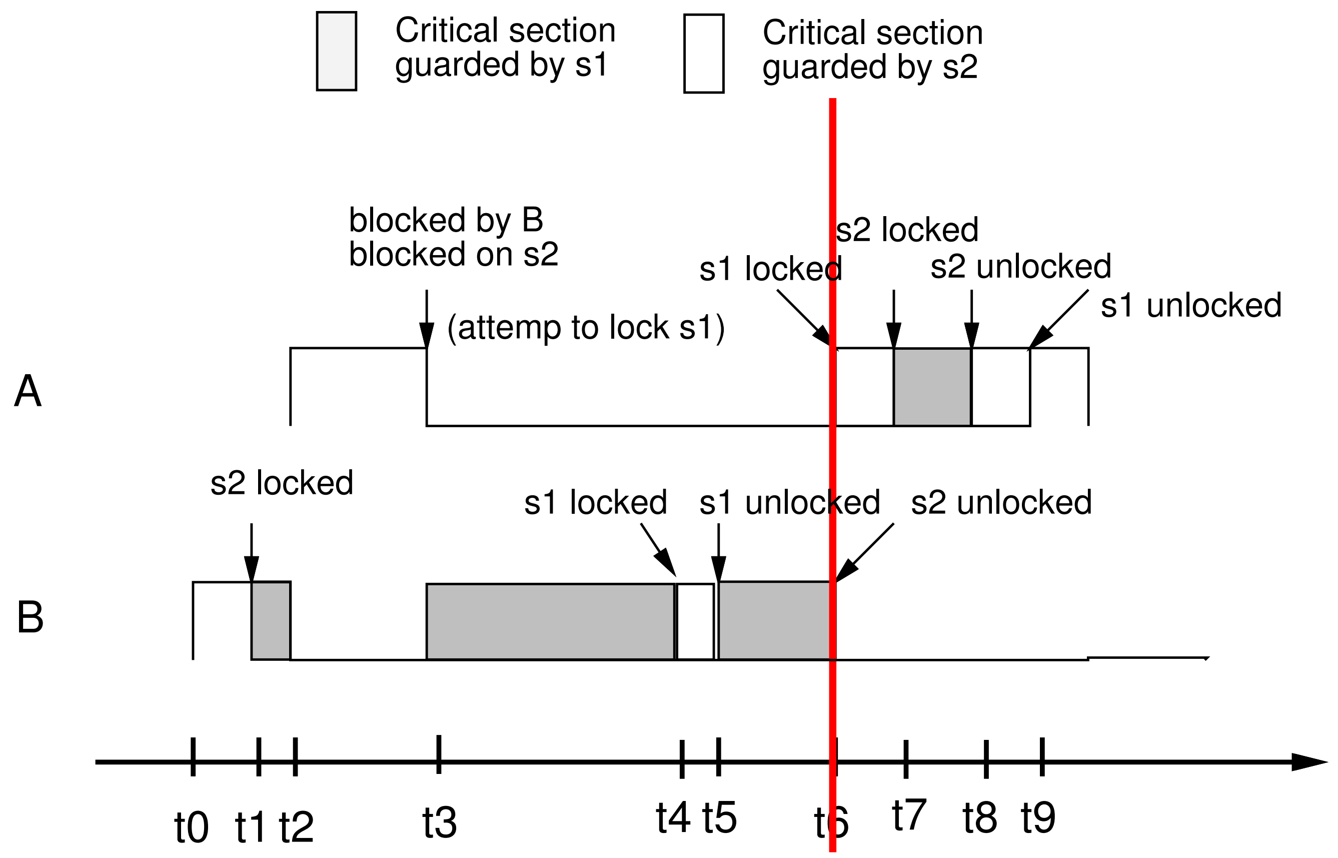
- t_3: A tries to lock s_1. A fails since A's priority (10) is not strictly higher than the ceiling of s_2 (10) that is held by B
- A is blocked by B
- A is blocked on s_2
- The priority of B is raised to 10.



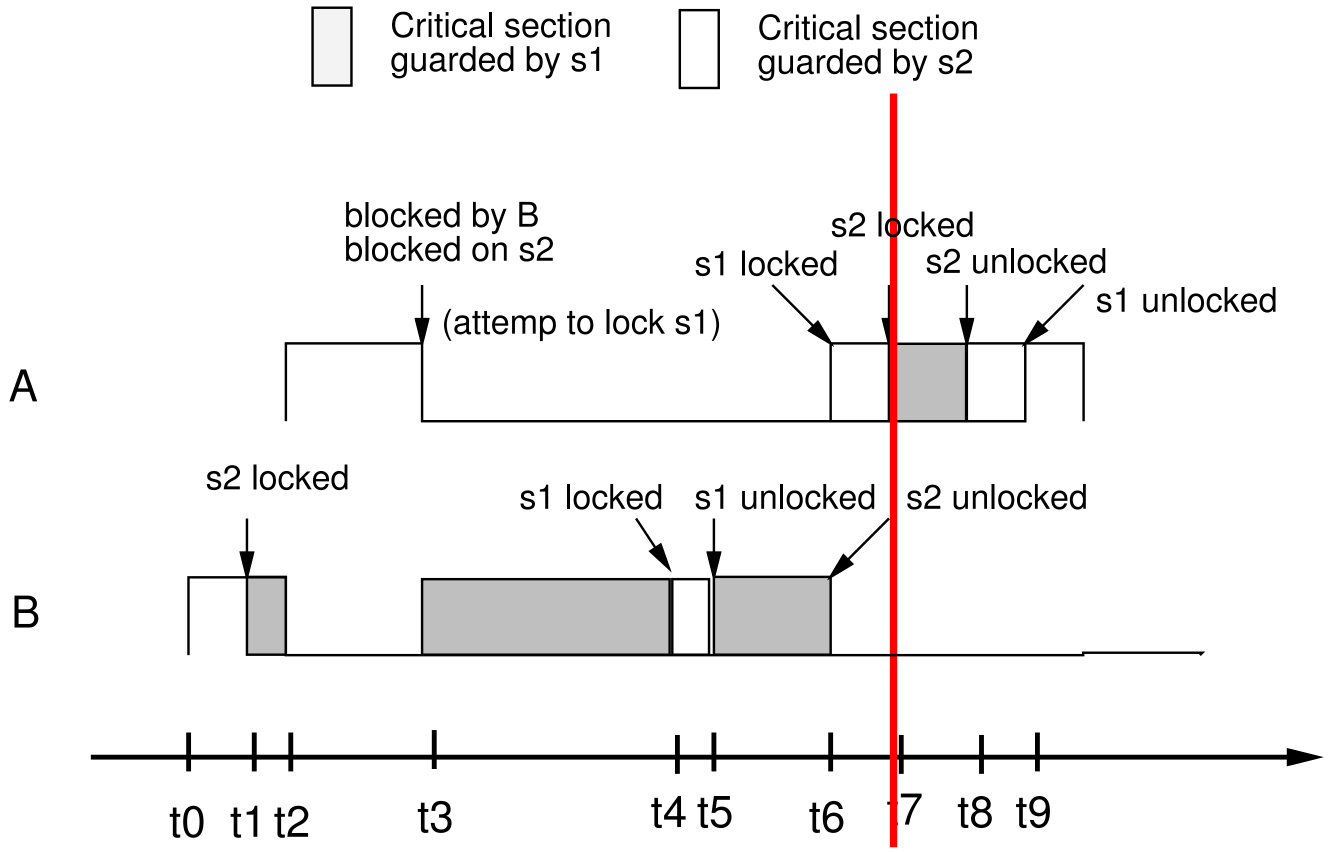
- t_4: B attempts to lock s_1. B succeeds since there are no locks held by any other tasks.



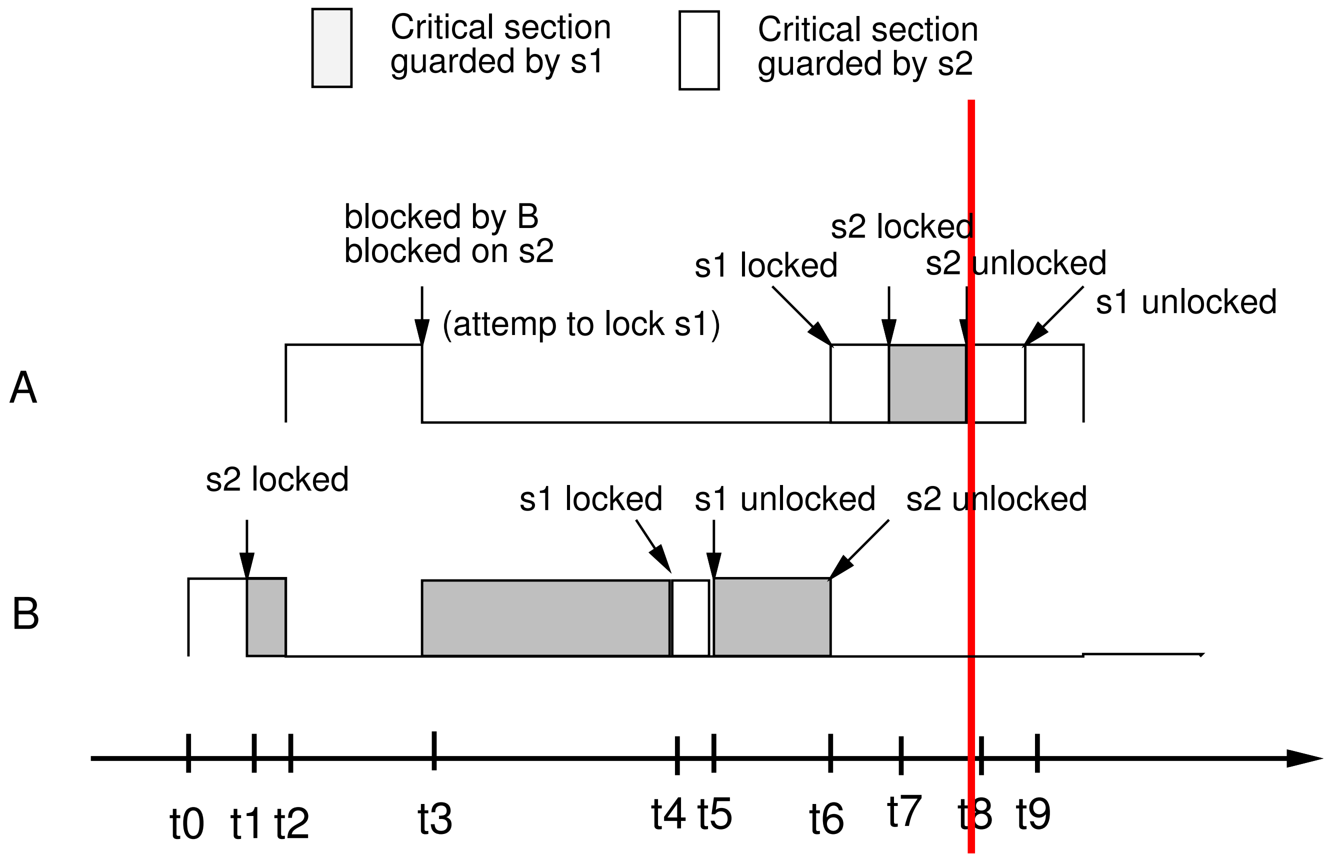
- t_5: B unlocks s_1



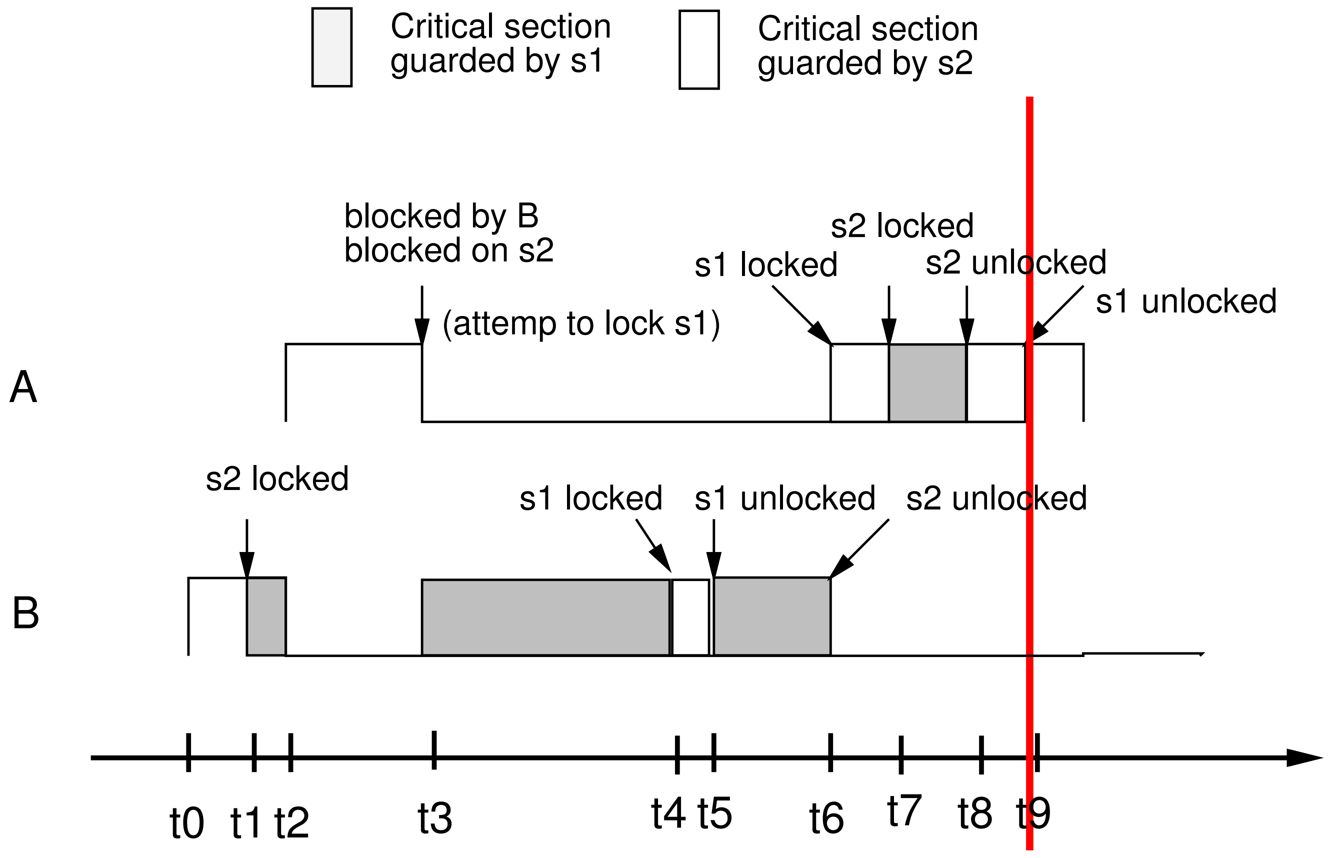
- t_6: B unlocks s_2
- The priority of B is lowered to its assigned priority (9)
- A preempts B, attempts to lock s_1 and succeeds



- t_7: A attempts to lock s_2. Succeeds



- t_8: A unlocks s_2



- t_9: A unlocks s_1

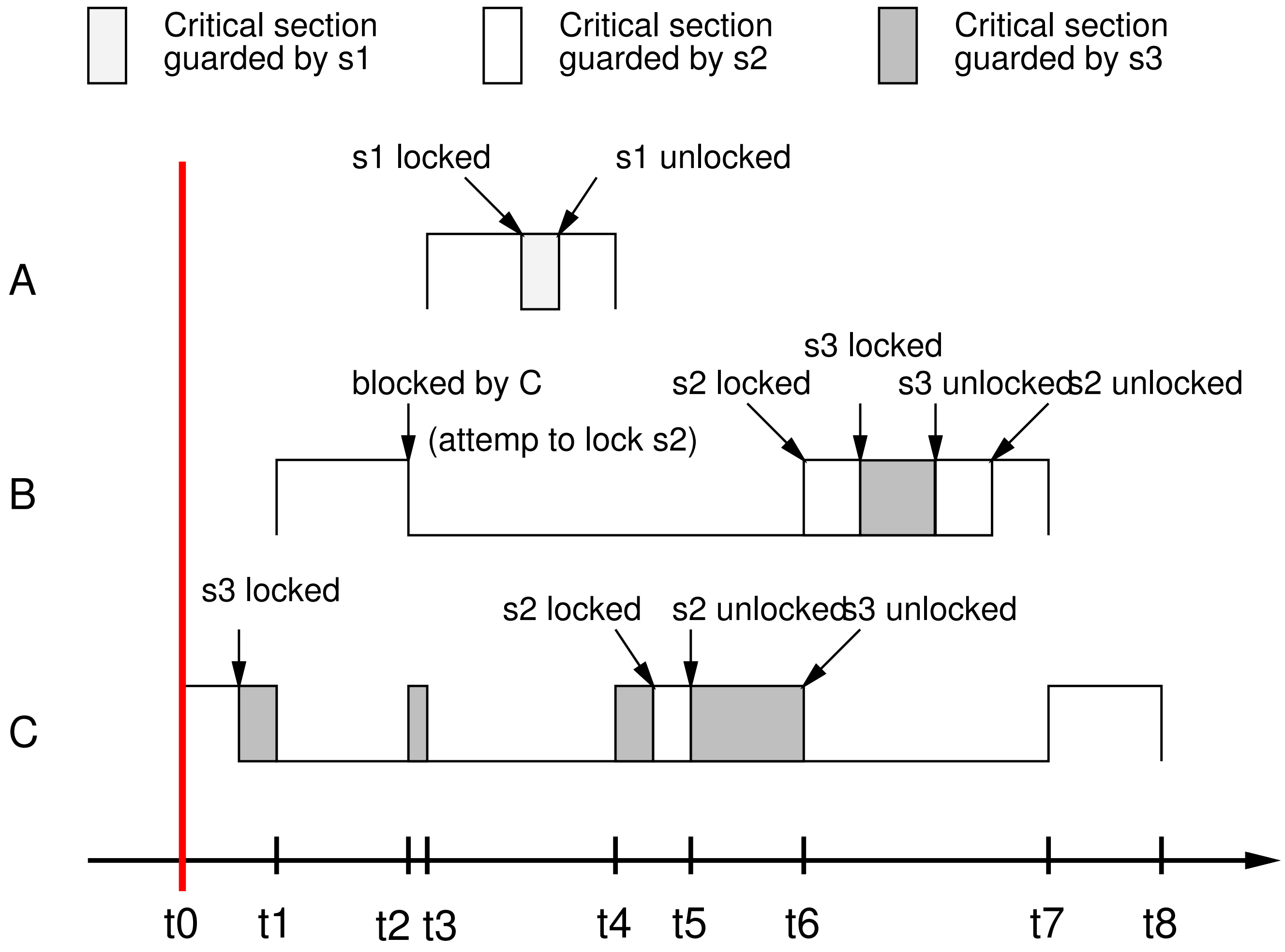
Example:

Task name	T	Priority
A	50	10
B	500	9
C	3000	8

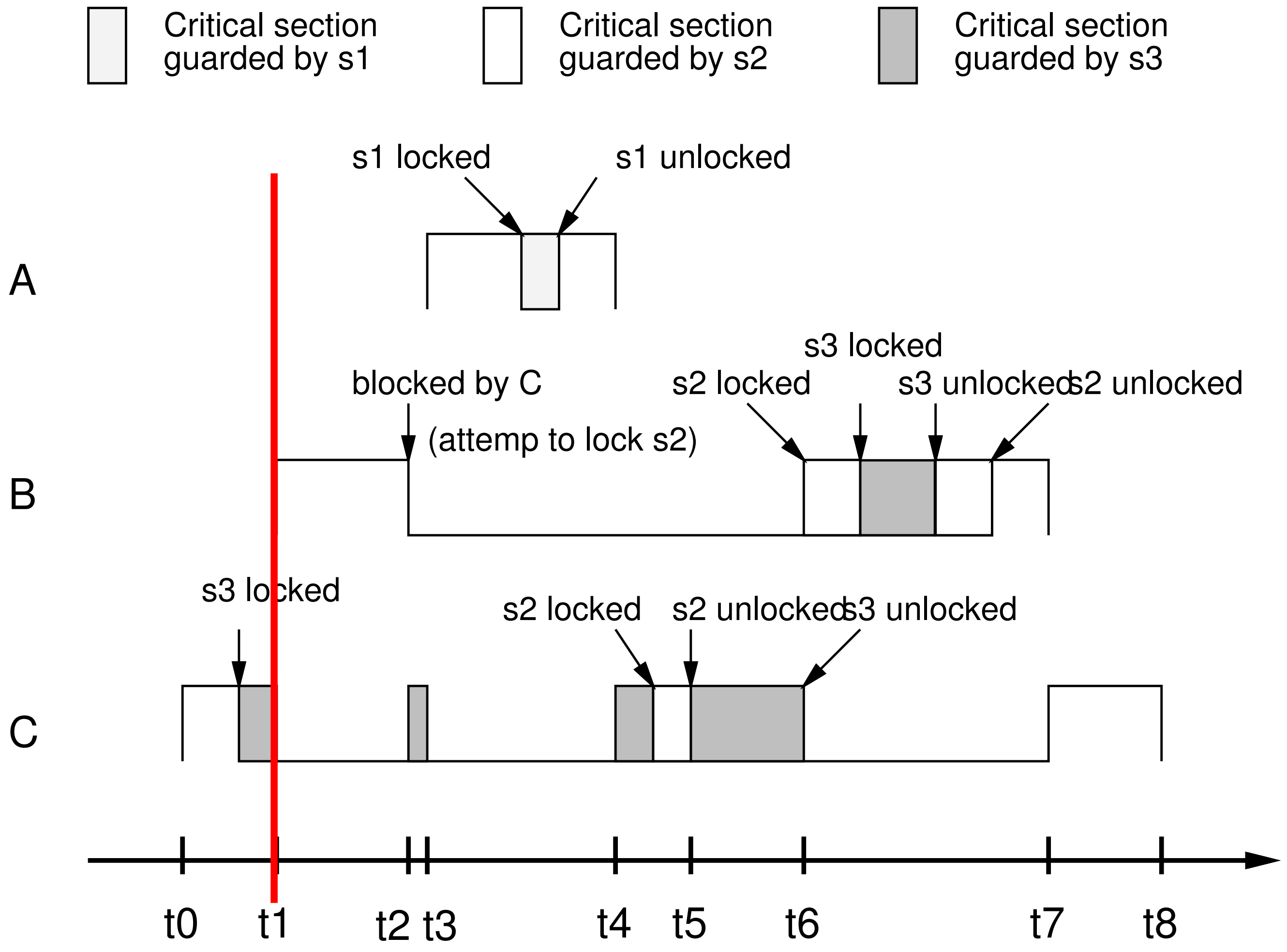
```
Task A       Task B       Task C

lock(s1)     lock(s2)     lock(s3)
..           ..           ..
unlock(s1)   lock(s3)     lock(s2)
..           ..           ..
             unlock(s3)   unlock(s2)
             ..           ..
             unlock(s2)   unlock(s3)
```

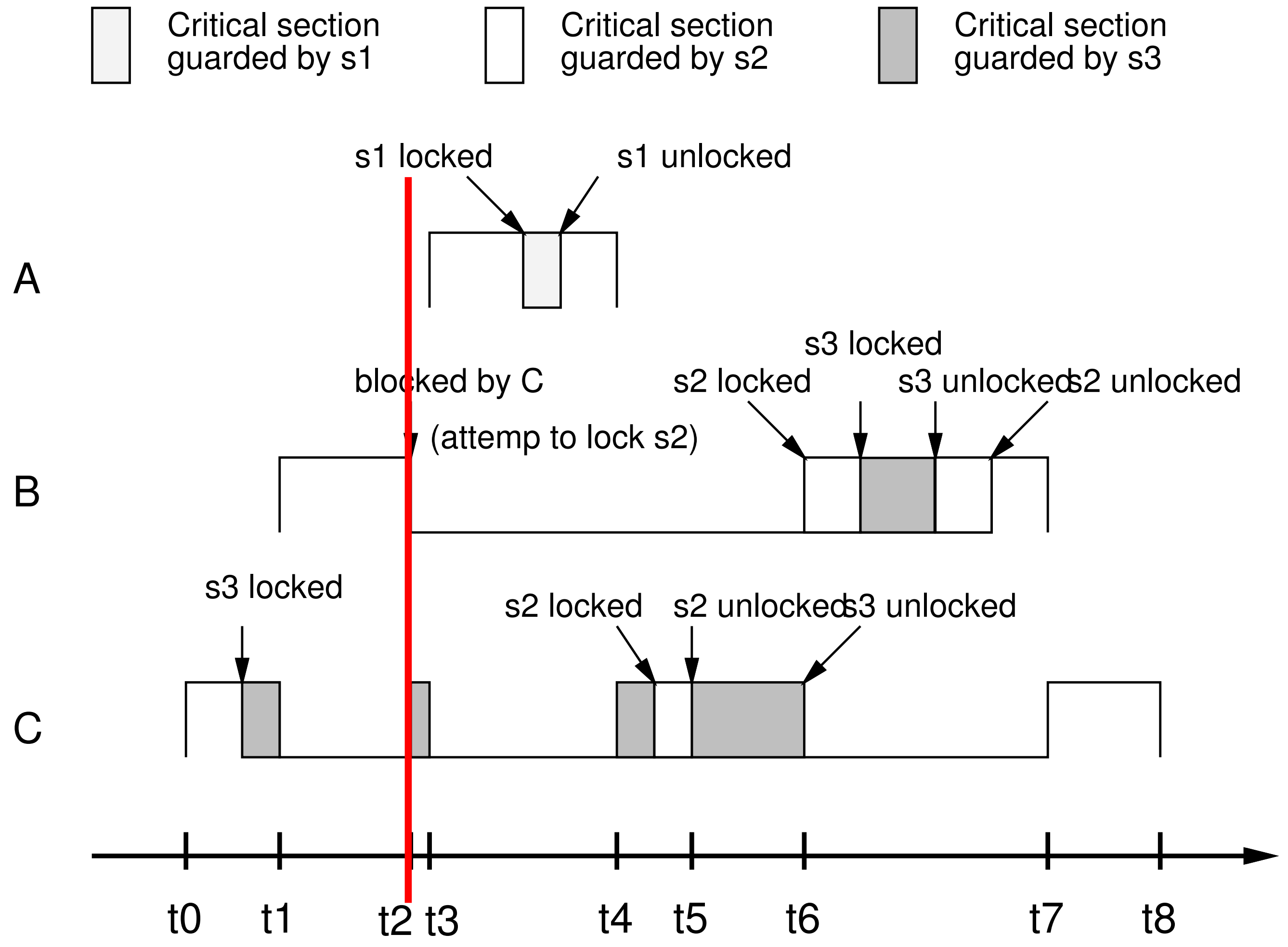
$$ceil(s_1) = 10, ceil(s_2) = ceil(s_3) = 9$$



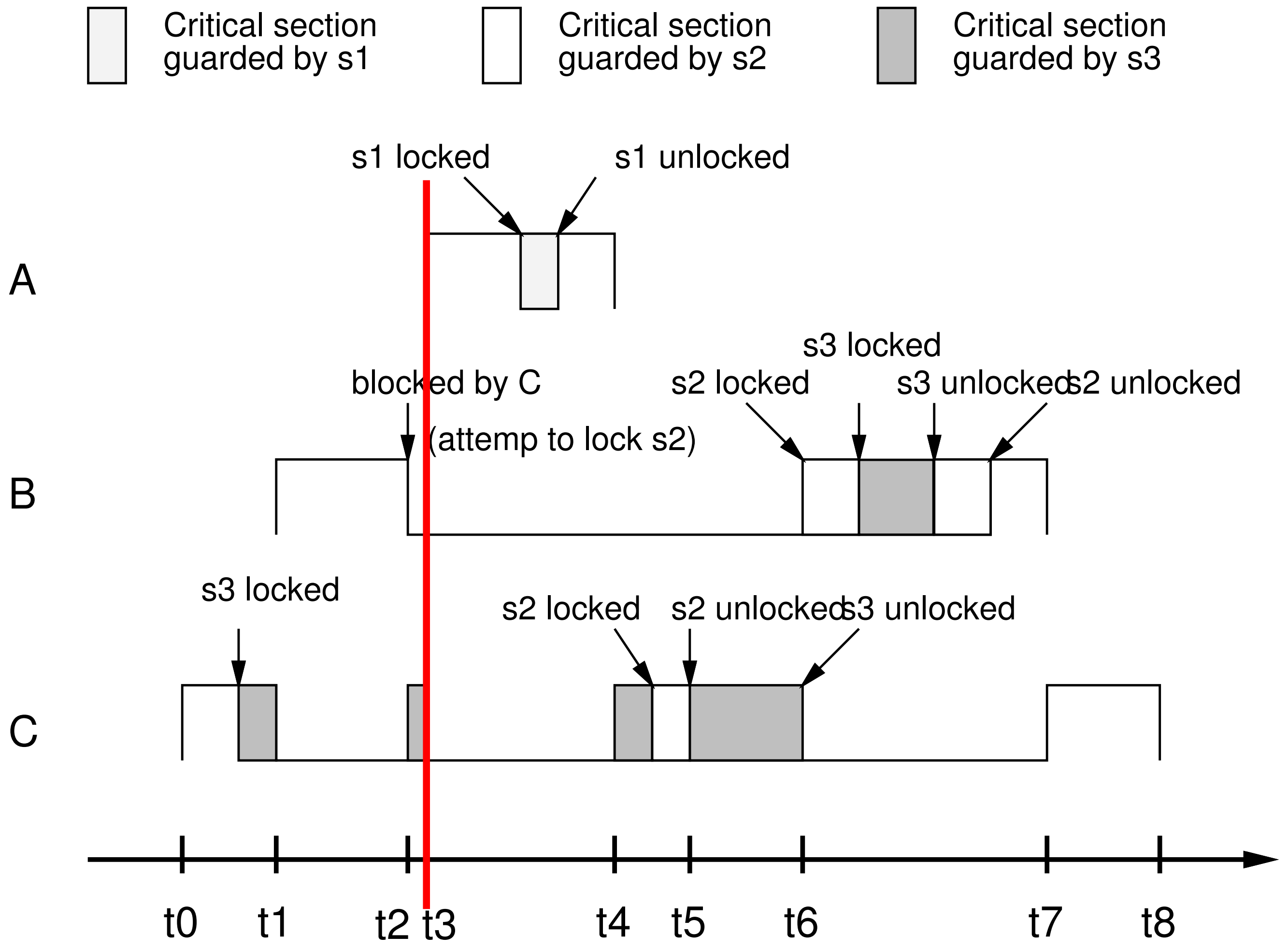
- t_0: C starts execution and then locks s_3



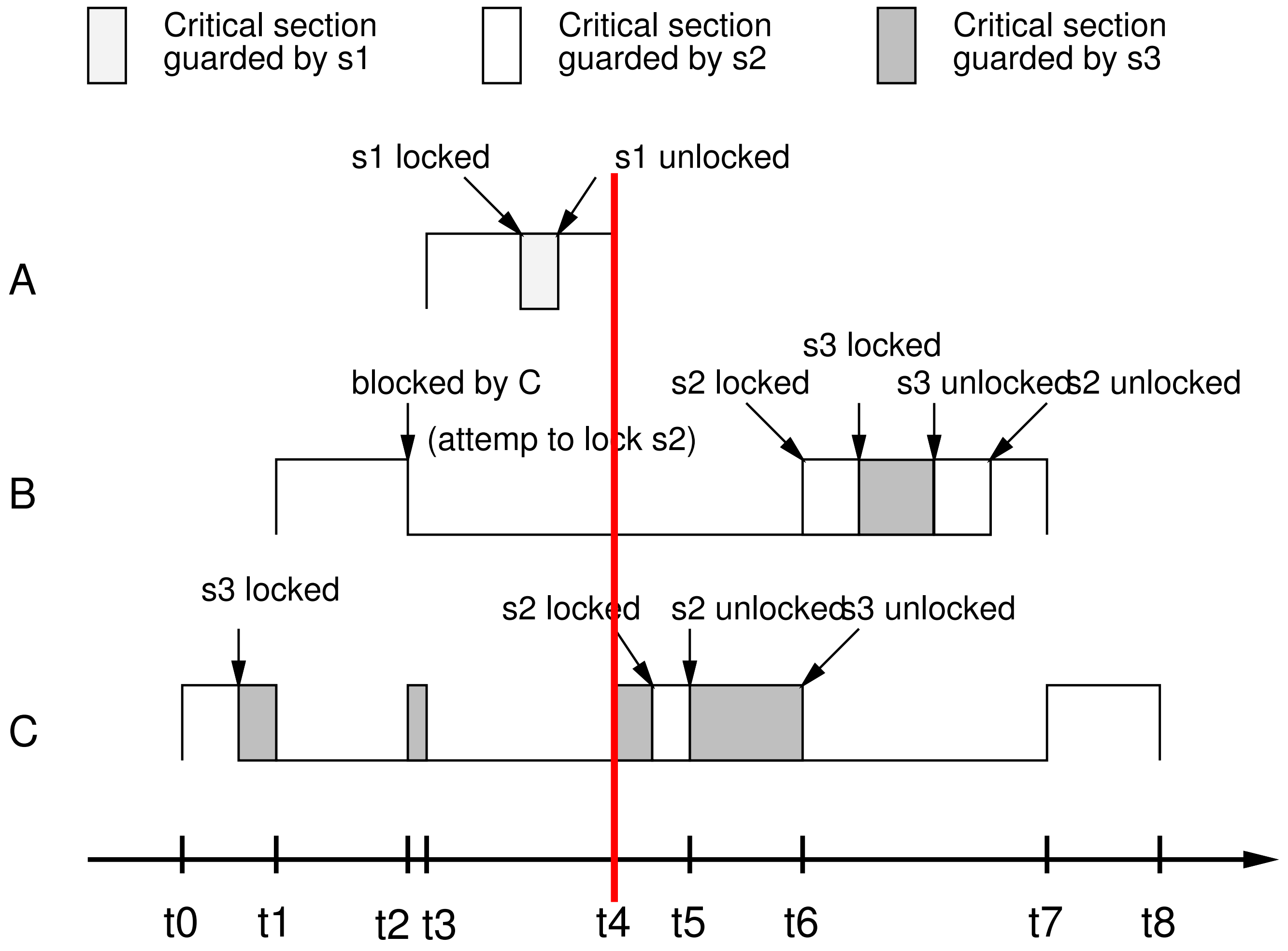
- t_1: B preempts C



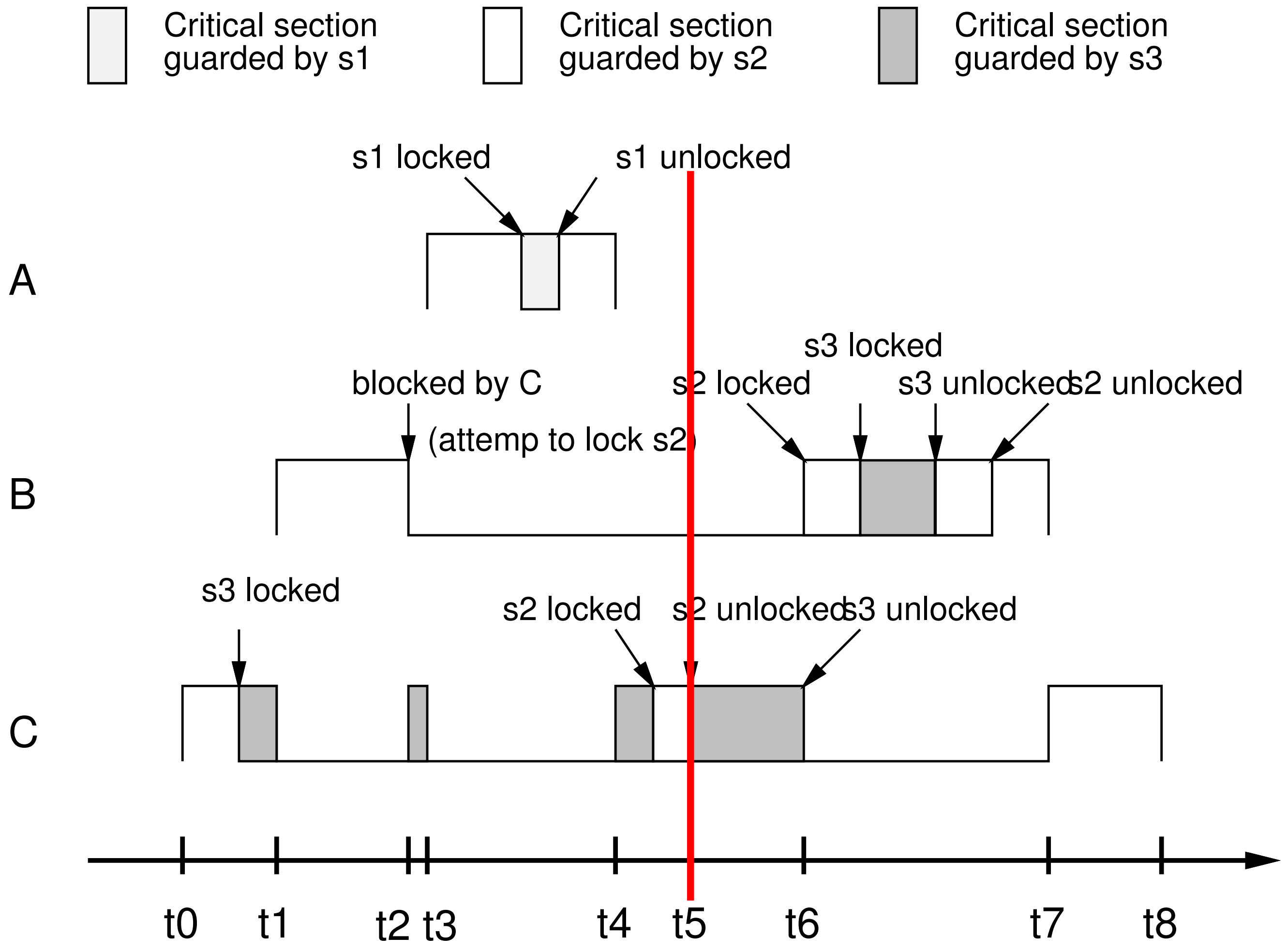
- t_2: B tries to lock s_2. B fails (the priority of B is not strictly higher than the ceiling of s_3 that is held by C) and blocks on s_3 (B is blocked by C). C inherits the priority of B.



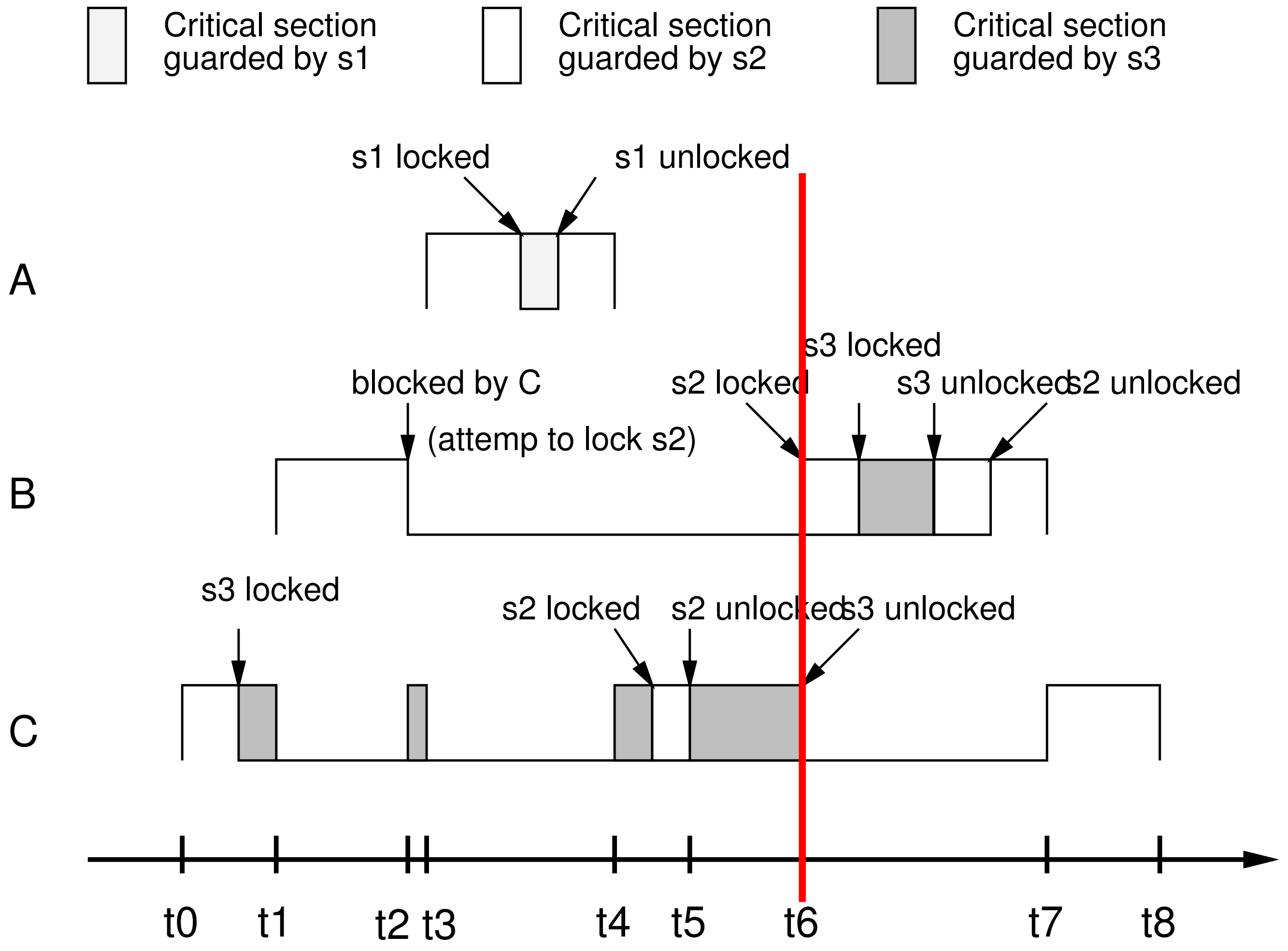
- t_3: A preempts C. Later is tries to lock s_1 and succeeds (the priority of A is higher than the ceiling of s_3).



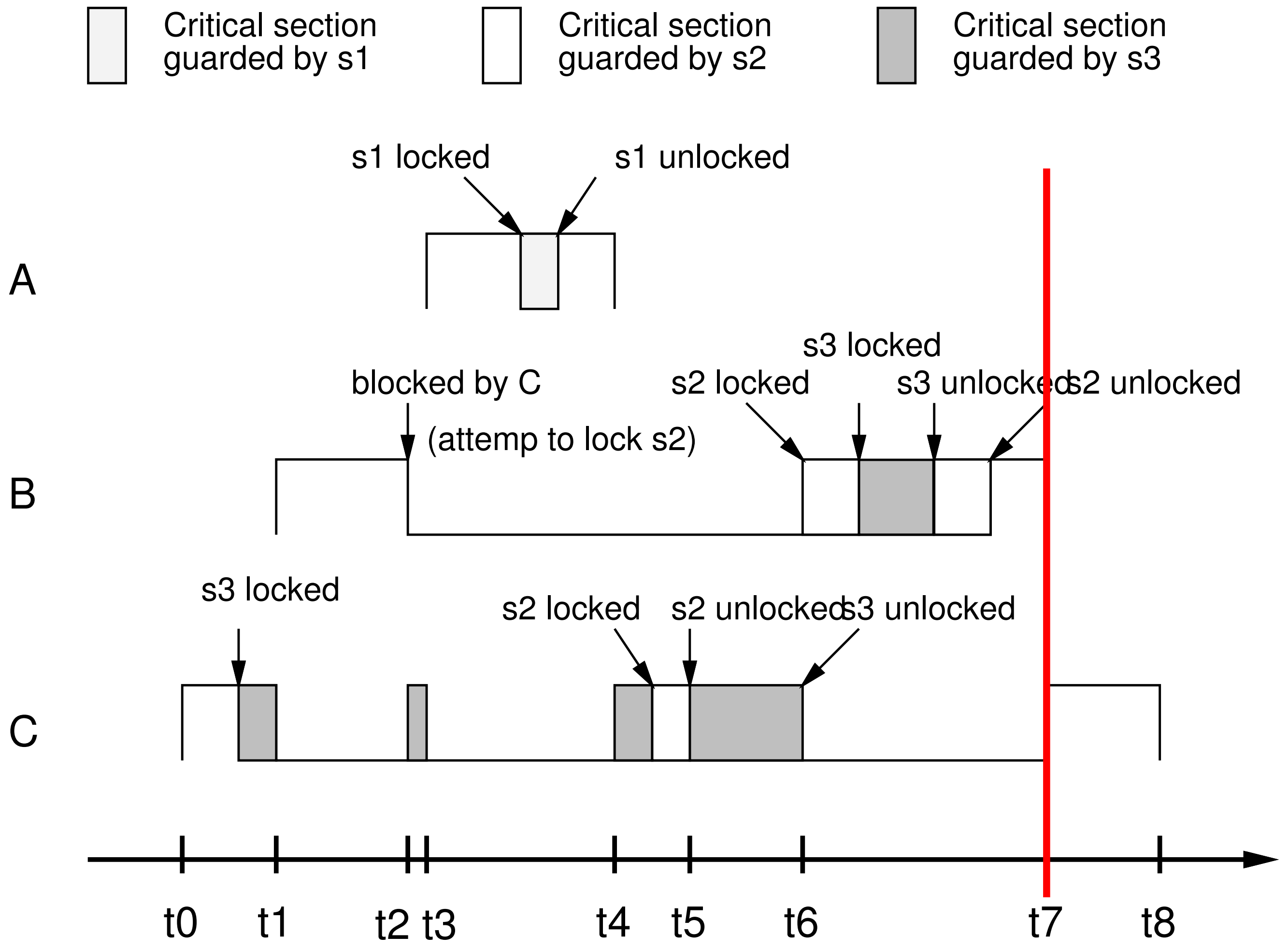
- t_4: A completes. C resumes and later tries to lock s_2 and succeeds (it is C itself that holds s_3).



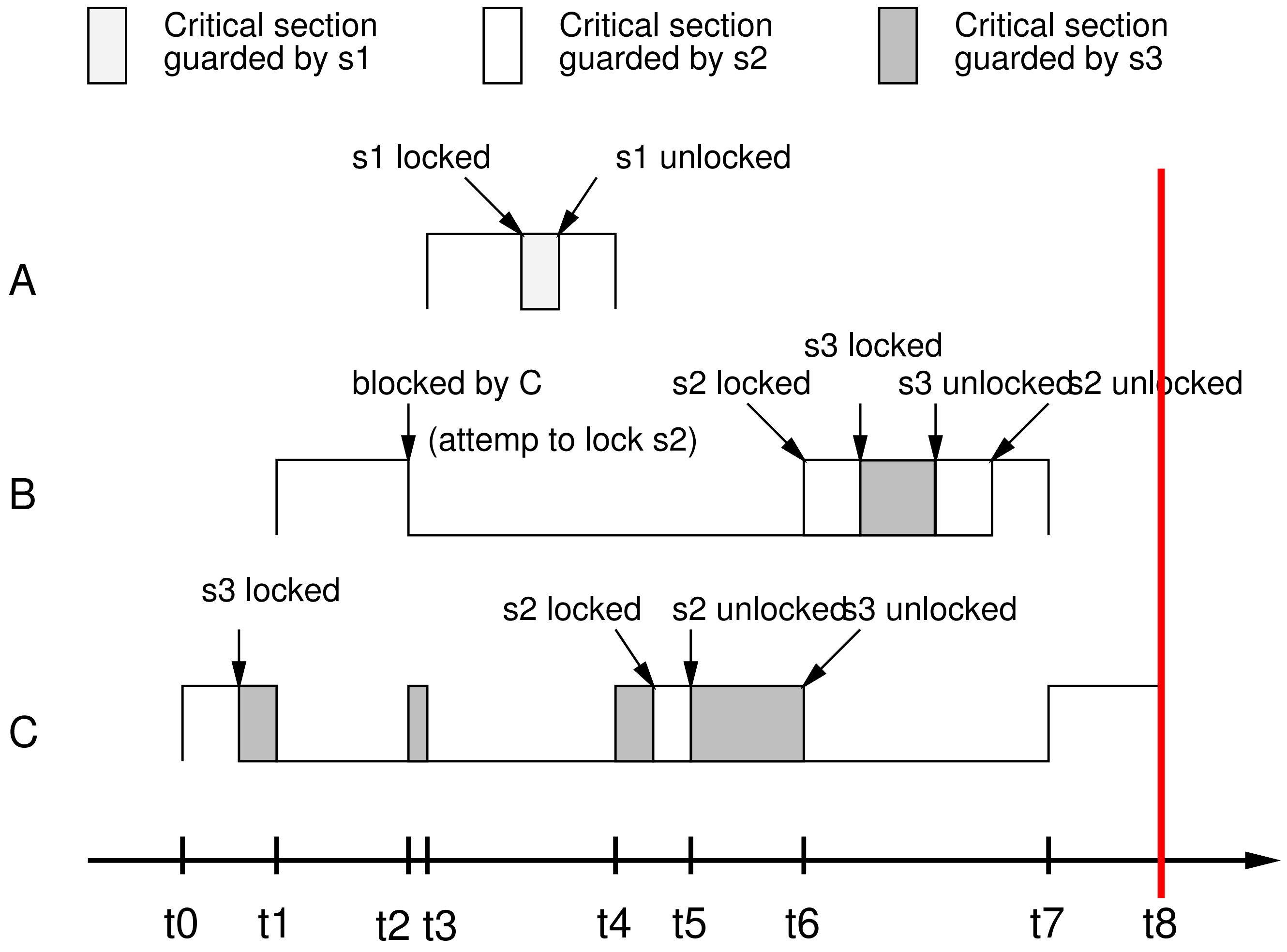
- t_5: C unlocks s_2



- t_6: C unlocks s_3, and gets back its basic priority. B preempts C, tries to lock s_2 and succeeds. Then B locks s_3, unlocks s_3 and unlocks s_2



- t_7: B completes and C is resumed.



- t_8: C completes

- A is never blocked
- B is blocked by C during the intervals $[t_2, t_3]$ and $[t_4, t_6]$. However, B is blocked for no more than the duration of one time critical section of the lower priority task C even though the actual blocking occurs over disjoint time intervals

General properties:

- with ordinary priority inheritance, a task i can be blocked for at most the duration of $\min(n, m)$ critical sections, where n is the number of lower priority tasks that could block i and m is the number of semaphores that can be used to block i
- with the priority ceiling inheritance, a task i can be blocked for at most the duration of one longest critical section
- sometimes priority ceiling introduces unnecessary blocking but the worst-case blocking delay is much less than for ordinary priority inheritance

The Immediate Inheritance Protocol

- when a task obtains a lock the priority of the task is immediately raised to the ceiling of the lock
- the same worst-case timing behavior as the priority ceiling protocol
- easy to implement
- on a single-processor system it is not necessary to have any queues of blocked tasks for the locks (semaphores, monitors) – tasks waiting to acquire the locks will have lower priority than the task holding the lock and can, therefore be queued in `ReadyQueue`.
- also known as the Priority Ceiling Emulation Protocol or the Priority Protect Protocol

Priority Inheritance

Priority inheritance is a common, but not mandatory, feature of most Java implementations.

The Real-Time Java Specification requires that the priority inheritance protocol is implemented by default. The priority ceiling protocol is optional.

Mailbox Communication

A process/thread communicates with another process/thread by sending a message to it.

Synchronization models:

- **Asynchronous:** the sender process proceeds immediately after having sent a message. Requires buffer space for sent but unread messages. Used in the course.
- **Synchronous:** the sender proceeds only when the message has been received. Rendez-vous.
- **Remote Invocation:** the sender proceeds only when a reply has been received from the receiver process. Extended rendez-vous. Remote Procedure/Method Call (RPC/RMC).

Naming schemes

- **Direct naming:**

  ```
  send "message" to "process"
  ```

- **Indirect naming:** uses a mailbox (channel, pipe)

  ```
  send "message" to "mailbox"
  ```

  With indirect naming different structures are possible:

  - many-to-one
  - many-to-many
  - one-to-one
  - one-to-many

Message Types

- system- or user-defined data structures
- the same representation at the sender and at the receiver
- shared address space
  - pointer
  - copy data

Message Buffering

Asynchronous message passing requires buffering.

The buffer size is always bounded.

A process is blocked if it tries to send to a full mailbox.

Problematic for high-priority processes

The message passing system must provide a primitive that only sends a message if the mailbox has enough space

Similarly, the message passing system must provide a primitive that makes it possible for a receiver process to test if there is a message in the mailbox before it reads

Message Passing

The `se.lth.cs.realtime.event` package provides support for mailboxes:

- asynchronous message passing
- both direct naming and indirect naming can be implemented

However, in most examples one assumes that each thread (e.g., Consumer threads) contains a mailbox for incoming messages.

Messages

Messages are implemented as instances of objects that are subclasses to `RTEvent`

Messages are always time-stamped.

Constructors:

- `RTEvent()`: Creates an RTEvent object with the current thread as source and a time-stamp from the current system time.
- `RTEvent(long ts)`: Creates an RTEvent object with the current thread as source and with the specified time stamp.
- `RTEvent(java.lang.Object source)`: Creates an RTEvent object with the specified source object and a time-stamp from the current system time.
- `RTEvent(java.lang.Object source, long ts)` : Creates an RTEvent object with the specified source object and time stamp.

A time-stamp supplied to the constructor may denote the time when input was sampled, rather than when e.g. an output event was created from a control block or digital filter.

The source is by default the current thread, but a supplied source may denote some passive object like a control block run by an external thread (scan group etc.).

Methods:

- `getSource()`: Returns the source object of the RTEvent.
- `getTicks()`: Returns the event's time stamp in number of (system dependent) ticks.
- `getSeconds()`: Returns the time-stamp expressed in seconds.
- `getMillis()`: Returns the time-stamp expressed in milli-seconds.
- and some others

Mailboxes

Mailboxes (message buffers) implemented by the class `RTEventBuffer`

Synchronized bounded buffer with both blocking and non-blocking methods for sending (posting) and reading (fetching) messages.

Constructor:

- `RTEventBuffer(int maxSize)`

Methods:

- `doPost(RTEvent e)`: Adds an RTEvent to the queue, blocks caller if the queue is full.
- `tryPost(RTEvent e)`: Adds an RTEvent to the queue, without blocking if the queue is full. Returns null if the buffer is non-full, the event e otherwise.

- `doFetch()`: Returns the next RTEvent in the queue, blocks if none available.
- `tryFetch()`: Returns the next available RTEvent in the queue, or null if the queue is empty.
- `awaitEmpty()`: Waits for buffer to become empty.
- `awaitFull()`: Waits for buffer to become full.
- `isEmpty()`: Checks if buffer is empty.
- `is Full()`: Checks if buffer is full.
- plus some others

The class attributes are declared `protected` in order to make it possible to create subclasses with different behavior.

Producer-Consumer Example

```
class Producer extends Thread {
  Consumer receiver;
  MyMessage msg;

  public Producer(Consumer theReceiver) {
    receiver = theReceiver;
  }

  public void run() {
    while (true) {
      char c = getChar();
      msg = new MyMessage(c);
      receiver.putEvent(msg);
    }
  }
}
```

```
class Consumer extends Thread {
  private RTEventBuffer inbox;

  public Consumer(int size) {
    inbox = new RTEventBuffer(size);
  }
  public void putEvent(MyMessage msg) {
    inbox.doPost(msg);
  }
  public void run() {
    RTEvent m;
    while (true) {
      m = inbox.doFetch();
      if (m instanceof MyMessage) {
        MyMessage msg = (MyMessage) m ;
        useChar(msg.ch);
      } else ...
        // Handle other messages
      };
    }
  }
}
```

Message Passing add-ons

- **Selective waiting:** a process is only willing to accept messages of a certain category from a mailbox or directly from a set of processes. (Ada)
- **Time out:** time out on receiver processes.
- **Priority-sorted mailboxes:** urgent messages have priority over non-urgent messages.

Mailboxes in Linux

Mailbox communication is supported in a number of ways in Linux

One possibility is to use pipes, named pipes (FIFOs), or sockets, directly

Another possibility is POSIX Message Passing

- Very similar in functionality to the Mailbox system already presented

Several other alternatives, e.g., D-Bus
`http://www.freedesktop.org/wiki/Software/dbus`

Message Passing: Summary

Can be used both for communication and synchronization.

Using empty messages a mailbox corresponds to a semaphore.

Well suited for distributed systems.

Passing objects through a buffer

Using a buffer to pass objects from a sender thread to a receiver thread.

```
public class Buffer {
  private Object data;
  private boolean full = false;
  private boolean empty = true;

  public synchronized void put(Object inData) {
    while (full) {
      try {
        wait();
      } catch (InterruptedException e) {}
    }
    data = inData;
    full = true;
    empty = false;
    notifyAll();
  }
  public synchronized Object get() {
    while (empty) {
      try {
        wait();
      } catch (InterruptedException e) {}
    }
    full = false;
    empty = true;
    notifyAll();
    return data;
    }
}
```

Sender thread:

```
public void run() {
  Object data = new Object();
  while (true) {
    // Generate data
    b.put(data);
  }
}
```

Receiver thread:

```
public void run() {
  Object data;
  while (true) {
    data = b.get();
    // Use data
  }
}
```

Very dangerous. The object reference in the receiver thread points at the same object as the object reference in the sender thread. All modifications will be done without protection.

Approach 1: New objects

- the sender can create new objects before sending

```
public void run() {
  Object data = new Object();
  while (true) {
    // Generate data
    b.put(data);
    data = new Object();
  }
}
```

Approach 2: Copying in the buffer

```
public synchronized void put(Object inData) {
  while (full) {
    try {
      wait();
    } catch (InterruptedException e) {}
  }
  data = inData.clone();
  full = true;
  empty = false;
  notifyAll();
}
```

- the `clone` only performs a "shallow copy" - all references within the object are only copied and not cloned
- write an application-specific `clone` method

Approach 3: *Immutable objects*

- An immutable object is an object that cannot be modified once it has been created.
- An object is immutable if all data attributes are declared private and no methods are declared that may set new values to the data attributes
- The sender sends immutable objects. It is not possible for the user to modify them in any dangerous way.