Synchronization

Basic functions of RTOS

- Time management
- Task mangement
- Interrupt handling
- Memory management
- Exception handling
- Task scheduling
- Task synchronization

Synchronization primitives

- Semaphore: counting semaphore and binary semaphore
  - A semaphore is created with initial_count, which is the number of allowed holders of the semaphore lock. (initial_count=1: binary sem)
  - Sem_wait will decrease the count; while sem_signal will increase it.
  - A task can get the semaphore when the count > 0; otherwise, block on it.
- Mutex: similar to a binary semaphore, but mutex has an owner.
  - a semaphore can be "waited for" and "signaled" by any task,
  - while only the task that has taken a mutex is allowed to release it.
- Spinlock: lock mechanism for multi-processor systems,
  - A task wanting to get spinlock has to get a lock shared by all processors.
- Barrier: to synchronize a lot of tasks,
  - they should wait until all of them have reached a certain "barrier."

Potential problems in synchronization

- **Critical section** (data, service, code) protected by lock mechanism e.g. Semaphore etc. In a RTOS, the maximum time a task can be delayed because of locks held by other tasks should be less than its timing constraints.
- **Deadlock, livelock, starvation** Some deadlock avoidance/prevention algorithms are too complicate and indeterministic for real-time execution. Simplicity preferred, e..g.
  - all tasks always take locks in the same order.
- **Priority inversion** using priority-based task scheduling and locking primitives should know the "priority inversion" danger: a medium-priority job runs while a highpriority task is ready to proceed.

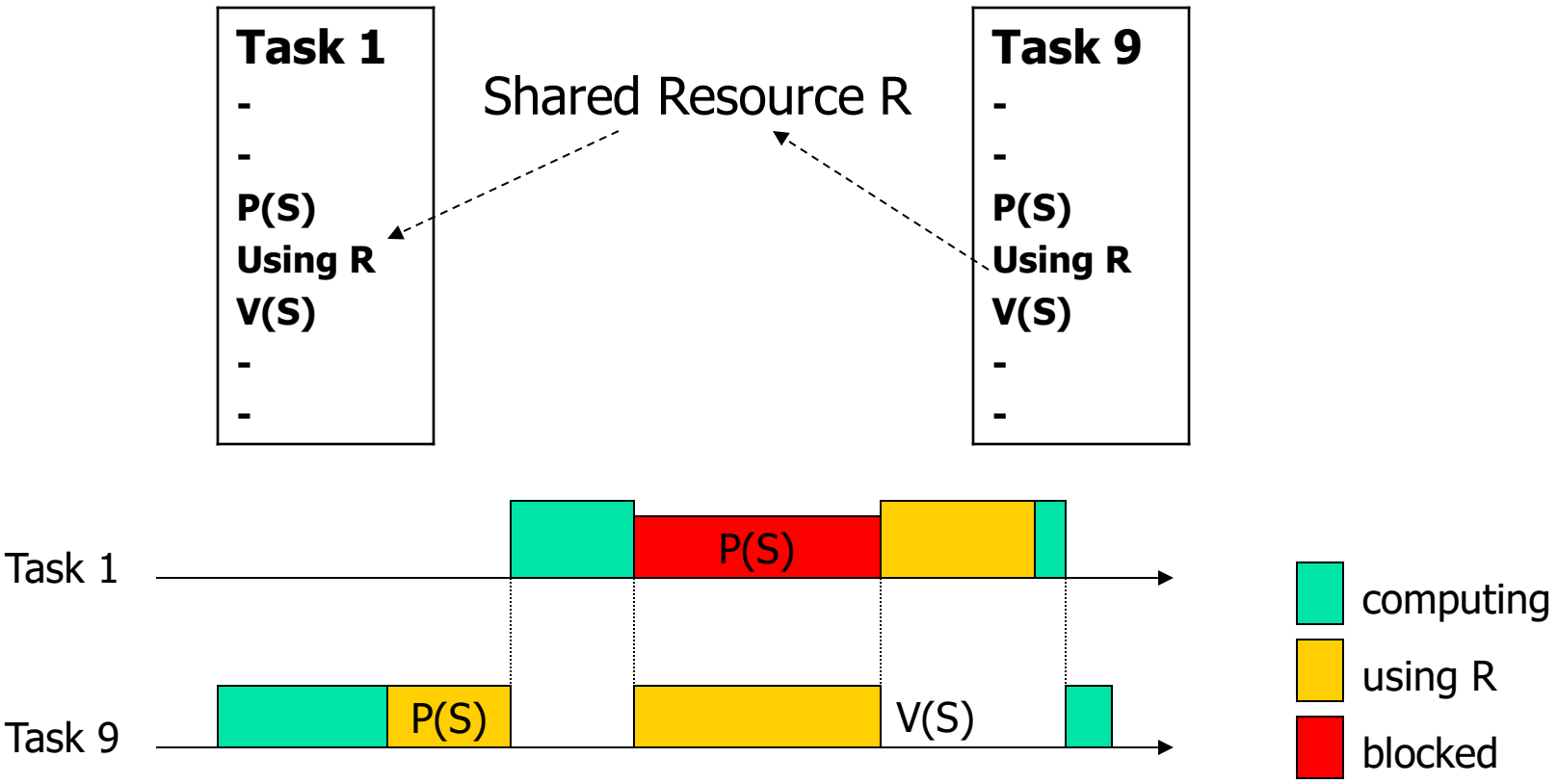
Solutions:

Resource Sharing and
Priority Ceiling Protocols

A classic paper on real-time systems

- L. Sha, R. Rajkumar, and J. P. Lehoczky, Priority Inheritance Protocols: An Approach to Real-Time Synchronization. In *IEEE Transactions on Computers*, vol. 39, pp. 1175-1185, Sep. 1990.

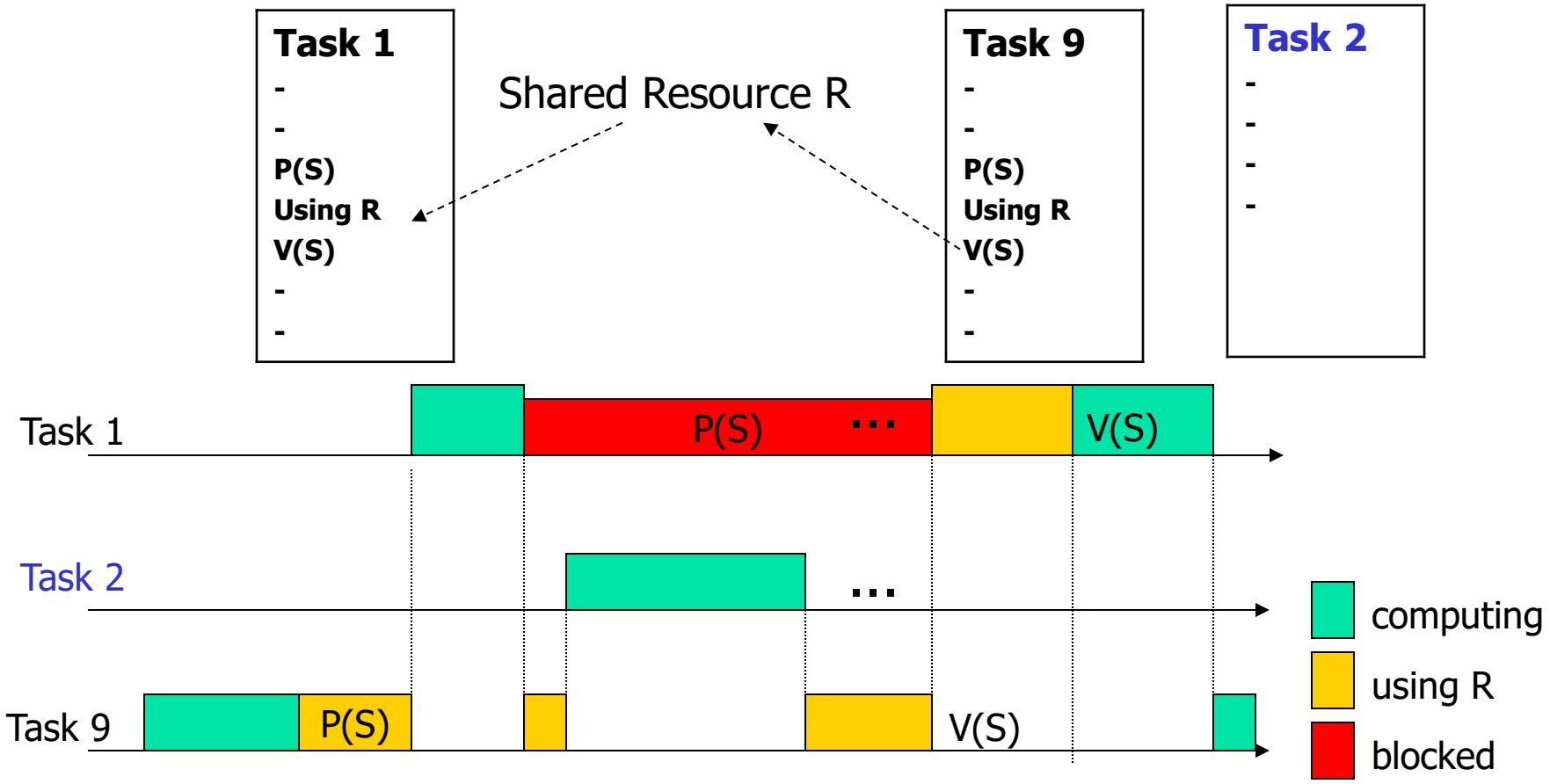
The simpliest form of priority inversion



Priority inversion problem

- Assume 3 tasks: A, B, C with priorities Ap<Bp<Cp
- Assume semaphore: S shared by A and C
- The following may happen:
  - A gets S by P(S)
  - C wants S by P(S) and blocked
  - B is released and preempts A
  - Now B can run for a long long period .....
  - A is blocked by B, and C is blocked by A
  - So C is blocked by B
- The above senario is called 'priority inversion'
- It can be much worse if there are more tasks with priorities in between Bp and Cp, that may block C as B does!

Un-bounded priority inversion



Solutions

- Tasks are ’forced’ to follow pre-defined rules when requesting and releasing resources (locking and unlocking semaphores)
- The rules are called ’Resource access protocols’

Semaphore, Dijkstra 60s

- A semaphore is a simple data structure with
  - a counter
    - the number of ”copies of a resource”
    - binary semaphore
  - a queue
    - Tasks waiting

and two operations:

- P(S): get or wait for semaphore
- V(S): release semaphore

Shared resources may be protected using semaphores

Implementation of Semaphores: SCB

- SCB: Semaphores Control Block

Counter
Queue of TCBs (tasks waiting)
Pointer to next SCB

The queue should be sorted by priorities (Why not FIFO?)

Implementation of semaphores: P-operation

- P(scb):

```
Disable-interrupt;
If scb.counter>0 then
   scb.counter - -1;
end then
else
     save-context();
     current-tcb.state := blocked;
     insert(current-tcb, scb.queue);
     dispatch();
     load-context();
end else
Enable-interrupt
```

Implementation of Semaphores: V-operation

- V(scb):

```
Disable-interrupt;
If not-empty(scb.queue)  then
        tcb := get-first(scb.queue);
        tcb.state := ready;
        insert(tcb, ready-queue);
        save-context();
        schedule(); /* dispatch invoked*/
        load-context();
 end then
 else scb.counter ++1;
 end else
Enable-interrupt
```

Resource Access Protocols

- Highest Priority Inheritance
  - Non preemption protocol (NPP)
- Basic Priority Inheritance Protocol (BIP)
  - POSIX (RT OS standard) mutexes
- Priority Ceiling Protocols (PCP)
- Immedate Priority Inheritance
  - Highest Locker's priority Protocol (HLP)
    - Ada95 (protected object) and POSIX mutexes

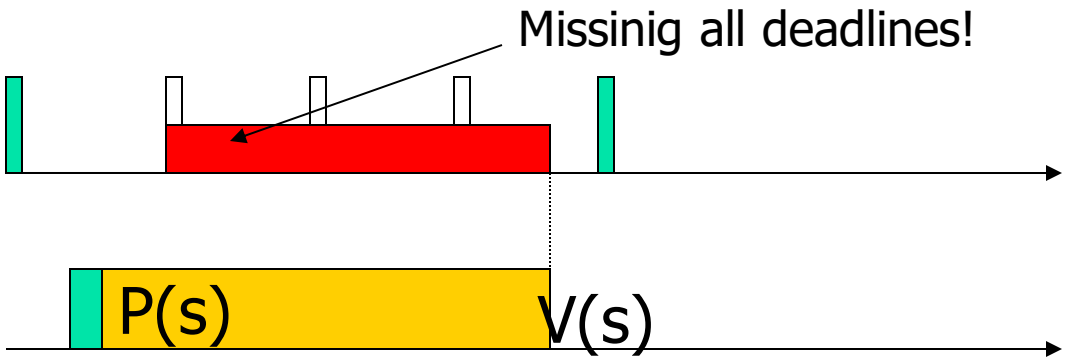
Non Preemption Protocol (NPP)

- Modify P(S) so that the ”caller” is assigned the highest priority if it succeeds in locking S
  - Highest priority=non preemtion!
- Modify V(S) so that the ”caller” is assigned its own priority back when it releases S

This is the simplest method to avoid Priority Inversion!

NPP: + and –

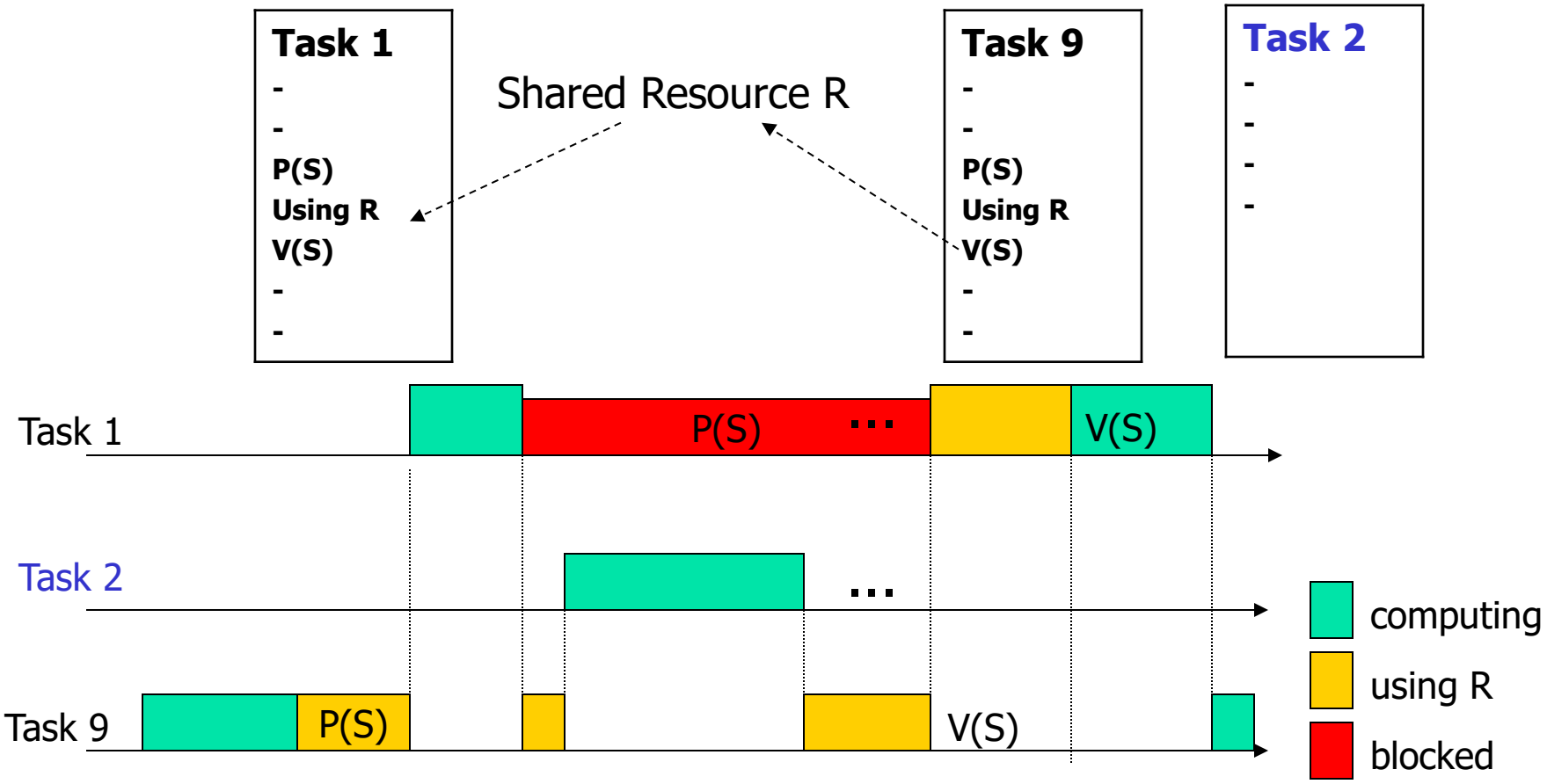
- Simple and easy to implement (+), how?
- Deadlock free (++)
- Number of blockings = 1 (+)
- Allow low-priority tasks to block high-priority tasks including those that have no sharing resources (-)



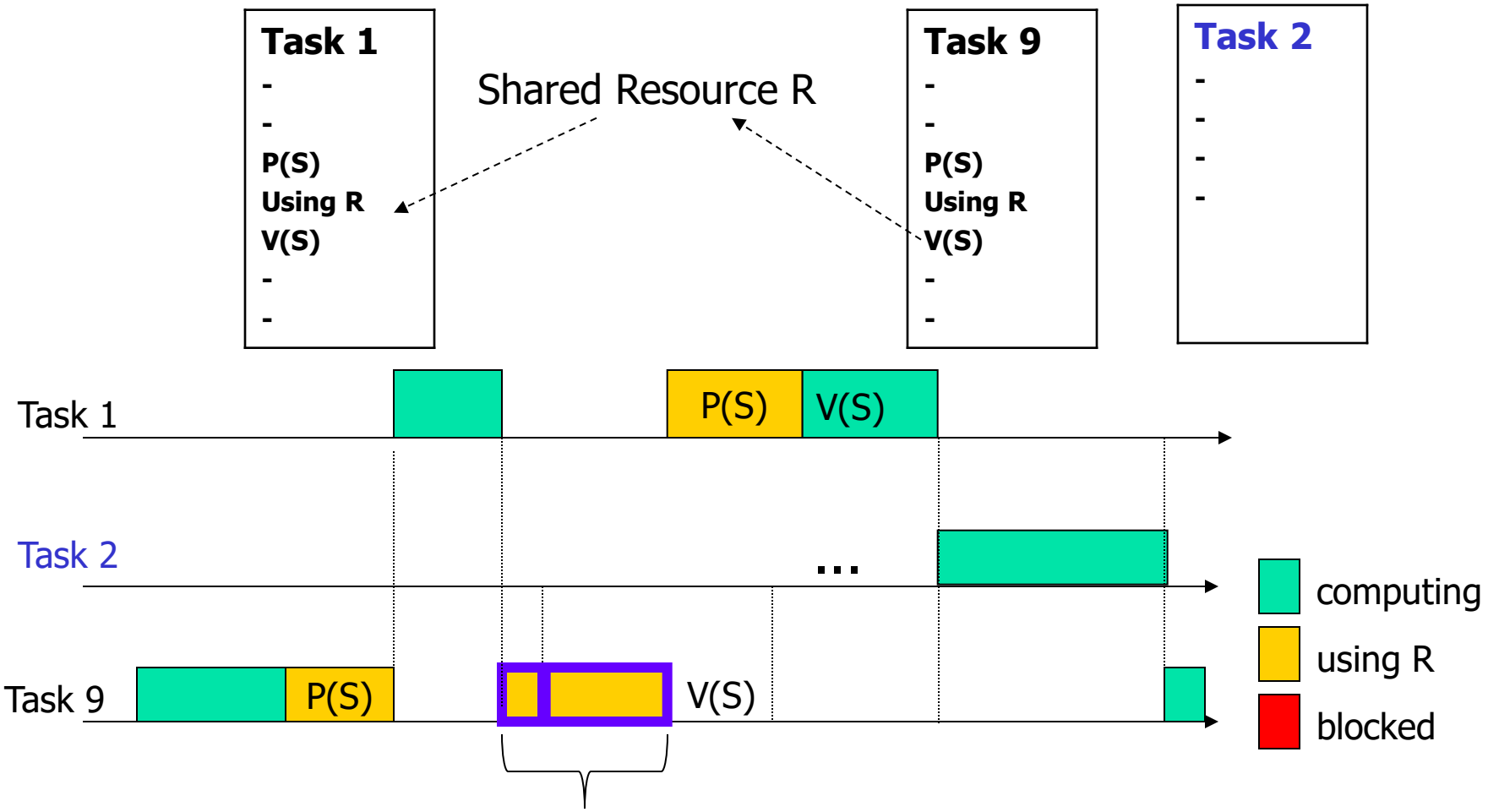
Basic Priority Inheritance Protocol (BIP)

- supported in RT POSIX
- Idea:
  - A gets semaphore S
  - B with higher priority tries to lock S, and blocked by S
  - B transfers its priority to A (so A is resumed and run with B's priority)
- Run time behaviour: whenever a lower-priotity task blocks a higher priority task, it inherits the priority of the blocked task

Un-bounded priority inversion



BIP protocol: Example

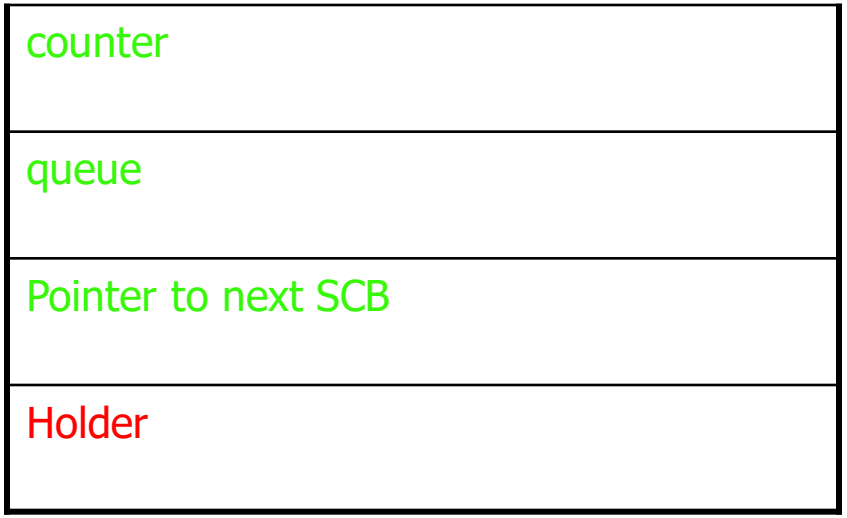


Run with the priority of Task 1

Implementation of Ceiling Protocols

- Main ideas:
  - Priority-based scheduling
  - Implement P/V operations on Semaphores to assign task priorities dynamically

Semaphore Control Block for BIP



Standard P-operation (without BIP)

- P(scb):

```
Disable-interrupt;
If scb.counter>0 then  {scb.counter - -1;
else
   {save-context( );
    current-task.state := blocked;
    insert(current-task, scb.queue);
    dispatch();
    load-context() }
 Enable-interrupt
```

P-operation with BIP

- P(scb):

```
Disable-interrupt;
If scb.counter>0 then  {scb.counter - -1;
                        scb.holder:= current-task
                        add(current-task.sem-list,scb)}
else
   {save-context( );
    current-task.state := blocked;
    insert(current-task, scb.queue);
    save(scb.holder.priotiry);
    scb.holder.priority := current-task.priority;
    dispatch();
    load-context() }
Enable-interrupt
```

Standard V-operation (without BIP)

- V(scb):

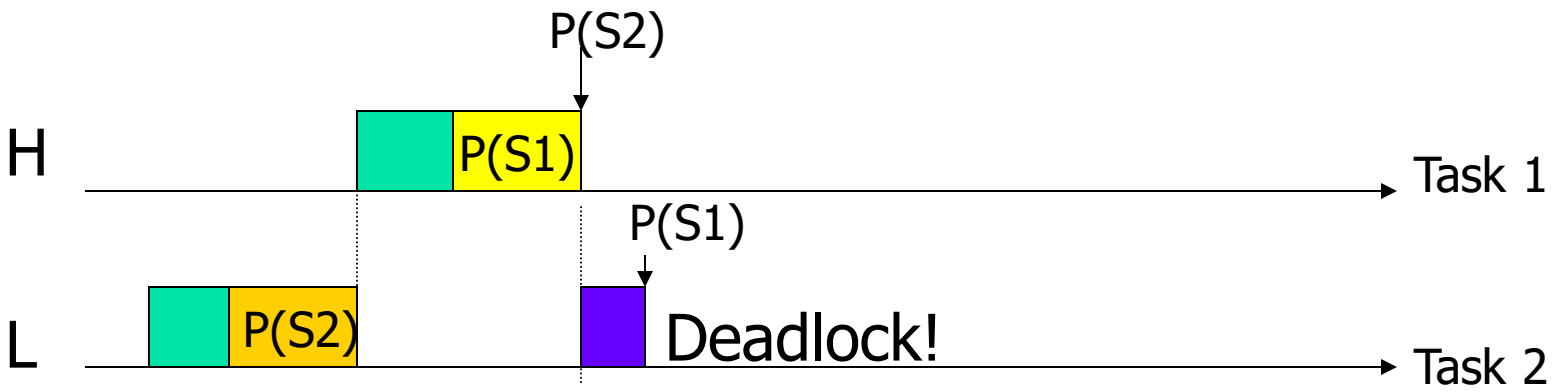
```
Disable-interrupt;
If not-empty(scb.queue)  then
        { next-to-run := get-first(scb.queue);
          next-to-run.state := ready;
          insert(next-to-run, ready-queue);
          save-context();
          schedule(); /* dispatch invoked*/
          load-context() }
    else scb.counter ++1;
Enable-interrupt
```

V-operation with BIP

- V(scb):

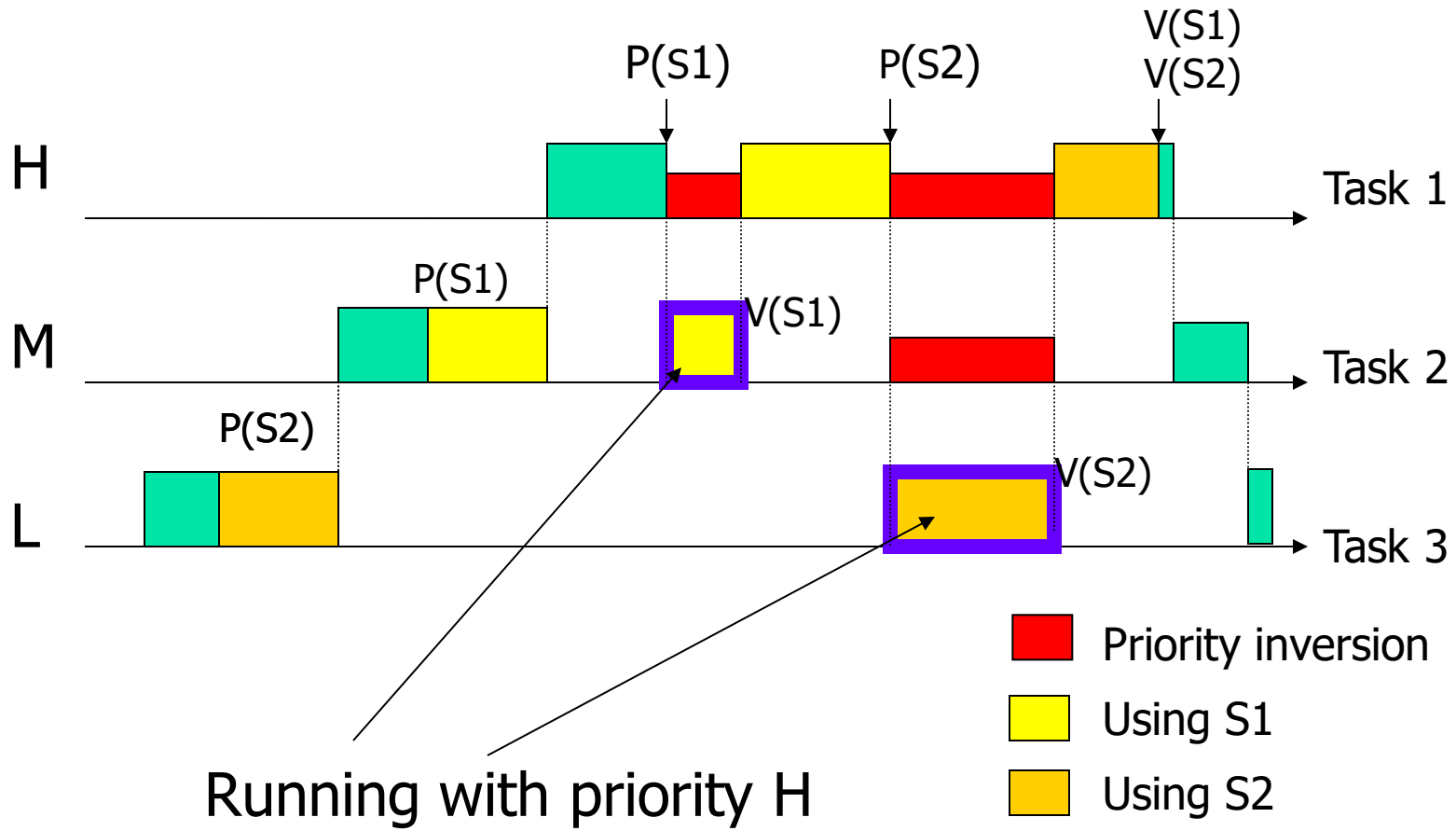
```
Disable-interrupt;
current-task.priority := ”original/previous priority”
      /* restore the previous priority of the ”caller” */
      If not-empty(scb.queue)  then
        { next-to-run := get-first(scb.queue);
          next-to-run.state := ready;
          scb.holder := next-to-run;
          add(next-to-run.sem-list, scb);
          insert(next-to-run, ready-queue);
          save-context();
          schedule(); /* dispatch invoked*/
          load-context() }
    else scb.counter ++1;
Enable-interrupt
```

Problem 1: potential deadlock



Task 2: ... P(S2) ... P(S1)...
Task 1: ... P(S1) ... P(S2)...

Problem 2: chained blocking – many preemptions



Task H needs M resources may be blocked M times:

→ many preemptions/run-time overheads

→ maximal blocking= at least, the sum of all CS sections for lower-priority tasks

Properties of BIP: + and -

- Bounded Priority inversion (+)
- Reasonable Run-time performance (+)
- Potential deadlocks (-)
- Chain-blocking – many preemptions (-)

Immediate Priority Inheritance:

=Highest Locker's Priority Protocol (HLP)

- Adopted in Ada95 (protected object), POSIX mutexes
- Idea: define the ceiling C(S) of a semaphore S to be the highest priority of all tasks that use S during execution. Note that C(S) can be calculated statically (off-line).

Run-time behaviour of HLP

- Whenever a task succeeds in holding a semaphor S, its priority is changed dynamically to the maximum of its current priority and C(S).
- When it finishes with S, it sets its priority back to what it was before

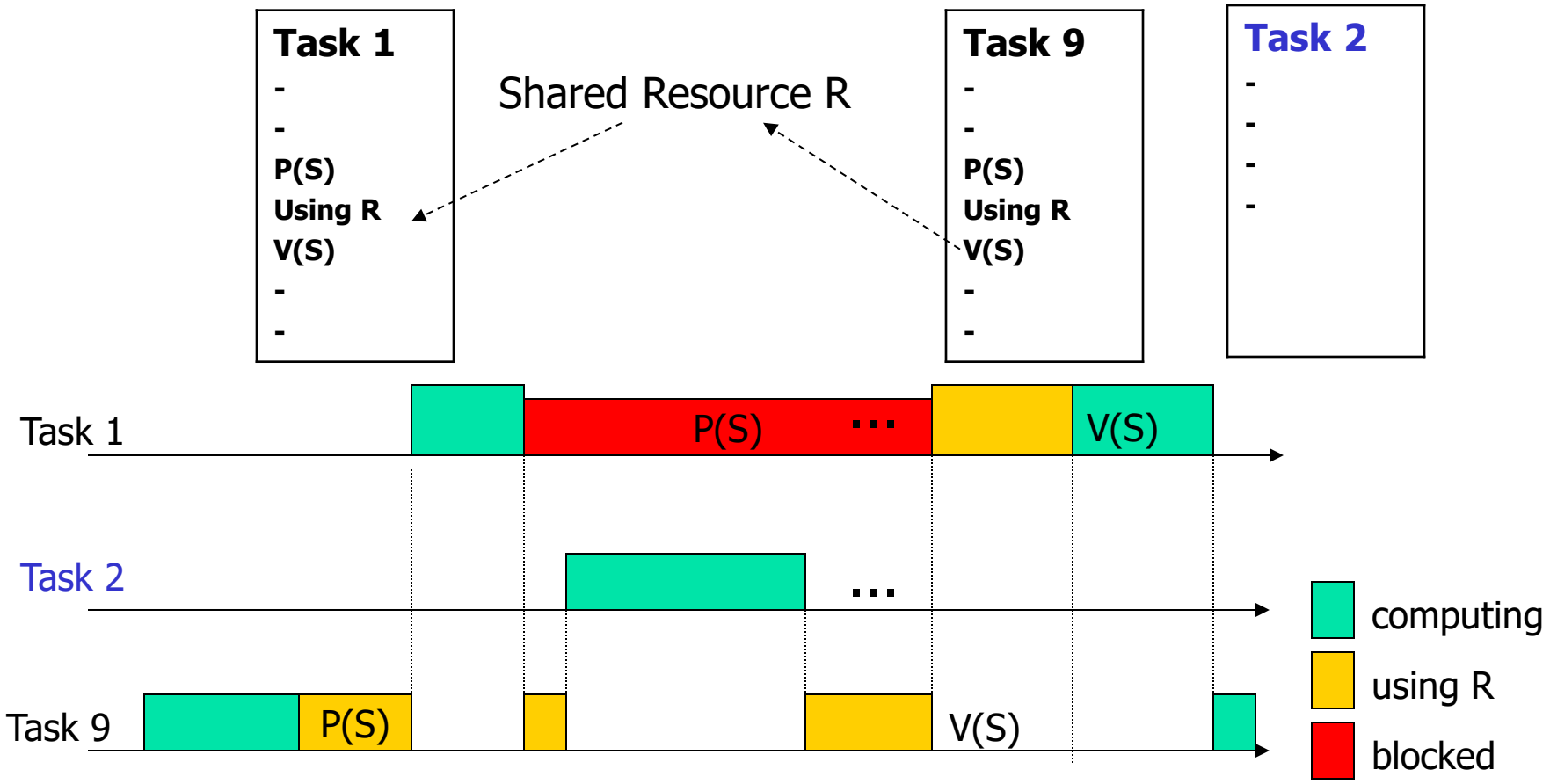
Priority Ceiling of Semaphore: Examples

	Priority	Share Semaphors
Task 1	H	S3
Task 2	M	S1, S
Task 3	L	S1, S2
Task 4	Lower	S2, S

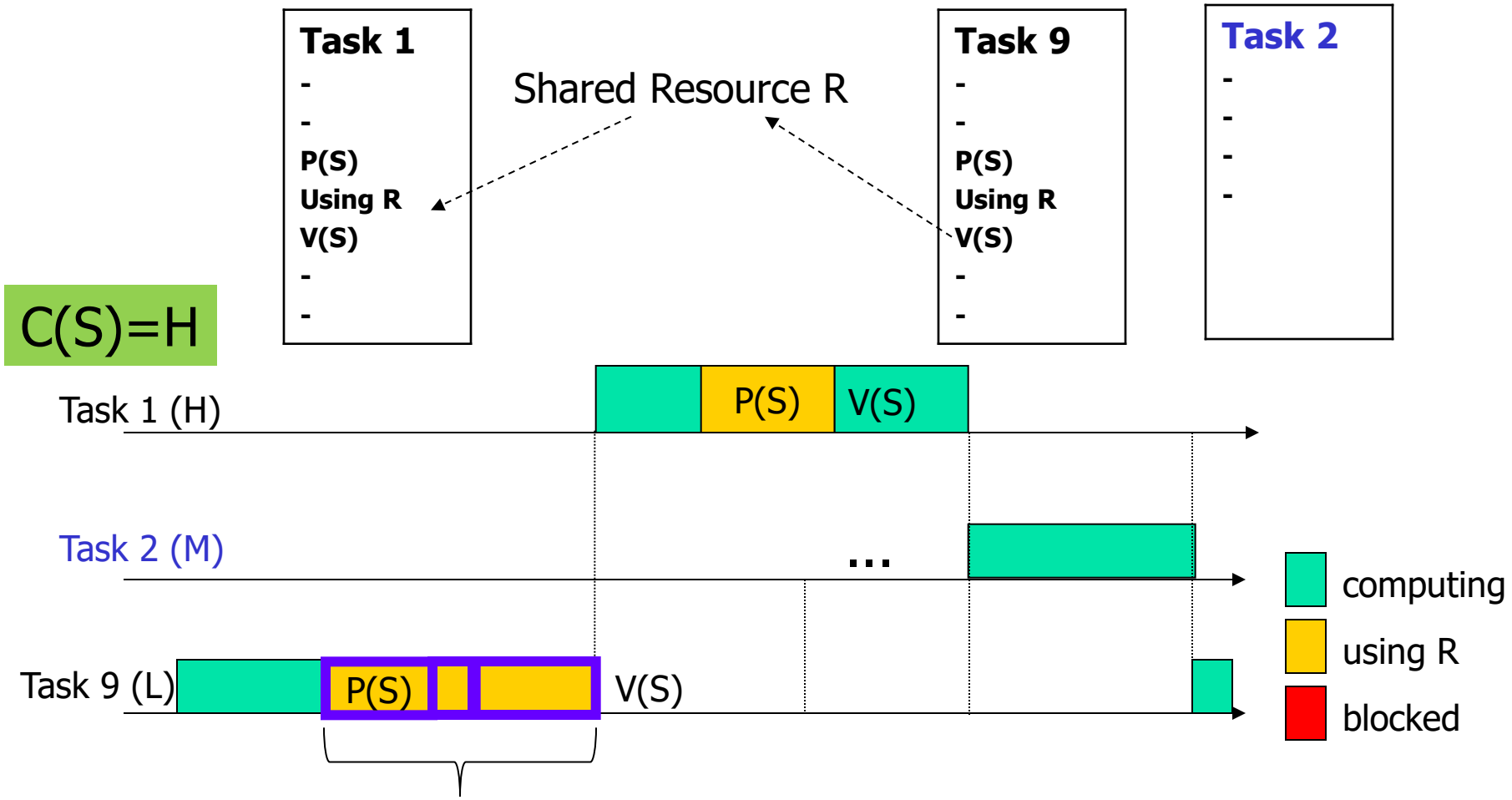
Ceiling of semaphors

C(S1)=M
C(S2)=L
C(S3)=H
C(S)=M

Un-bounded priority inversion

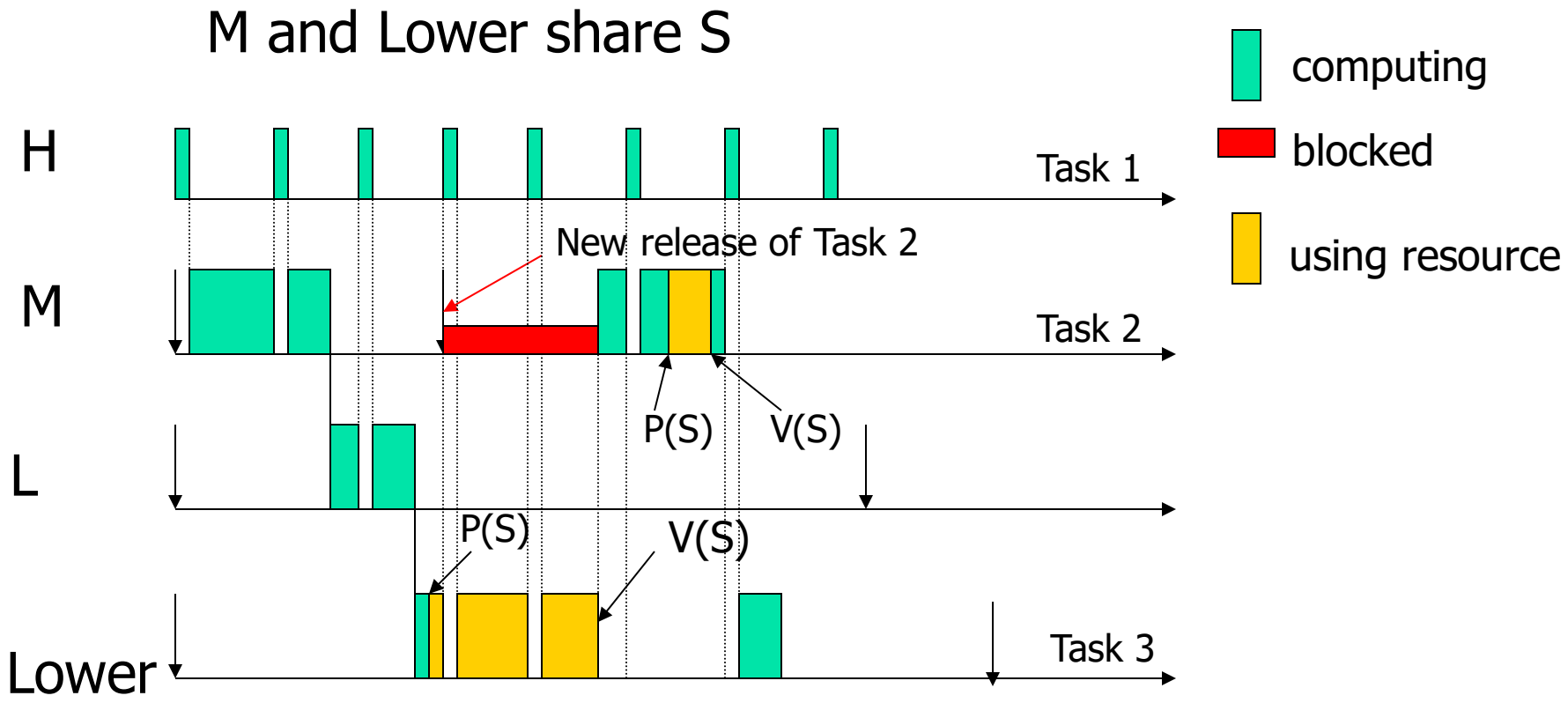


HLP protocol: Example

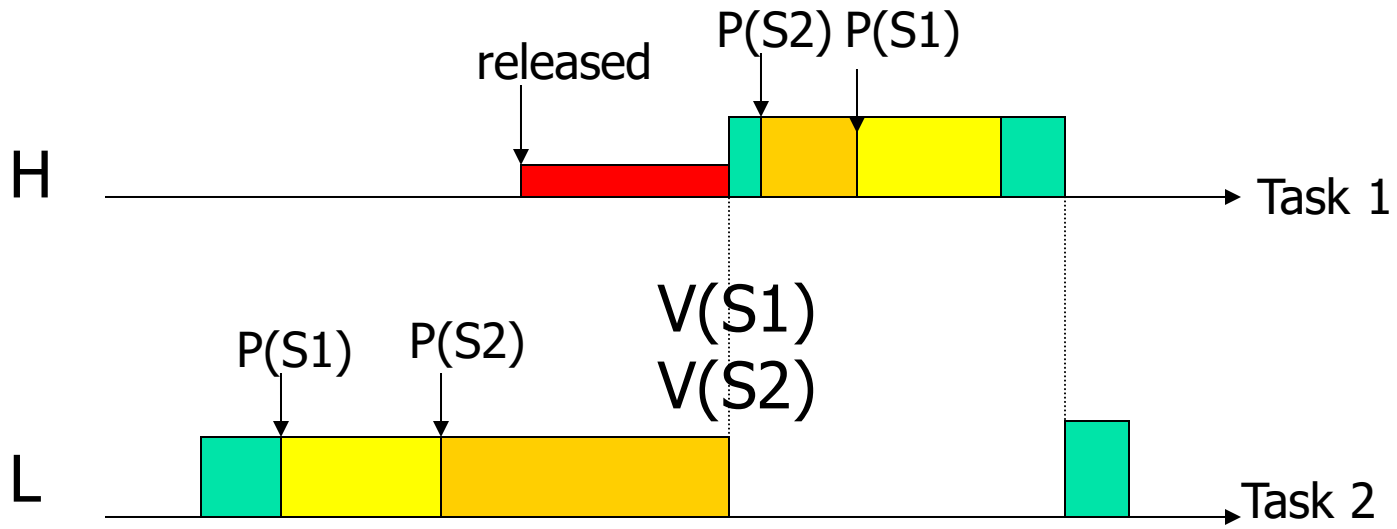


Run with the priority of C(S): H

HLP Protocol: Example

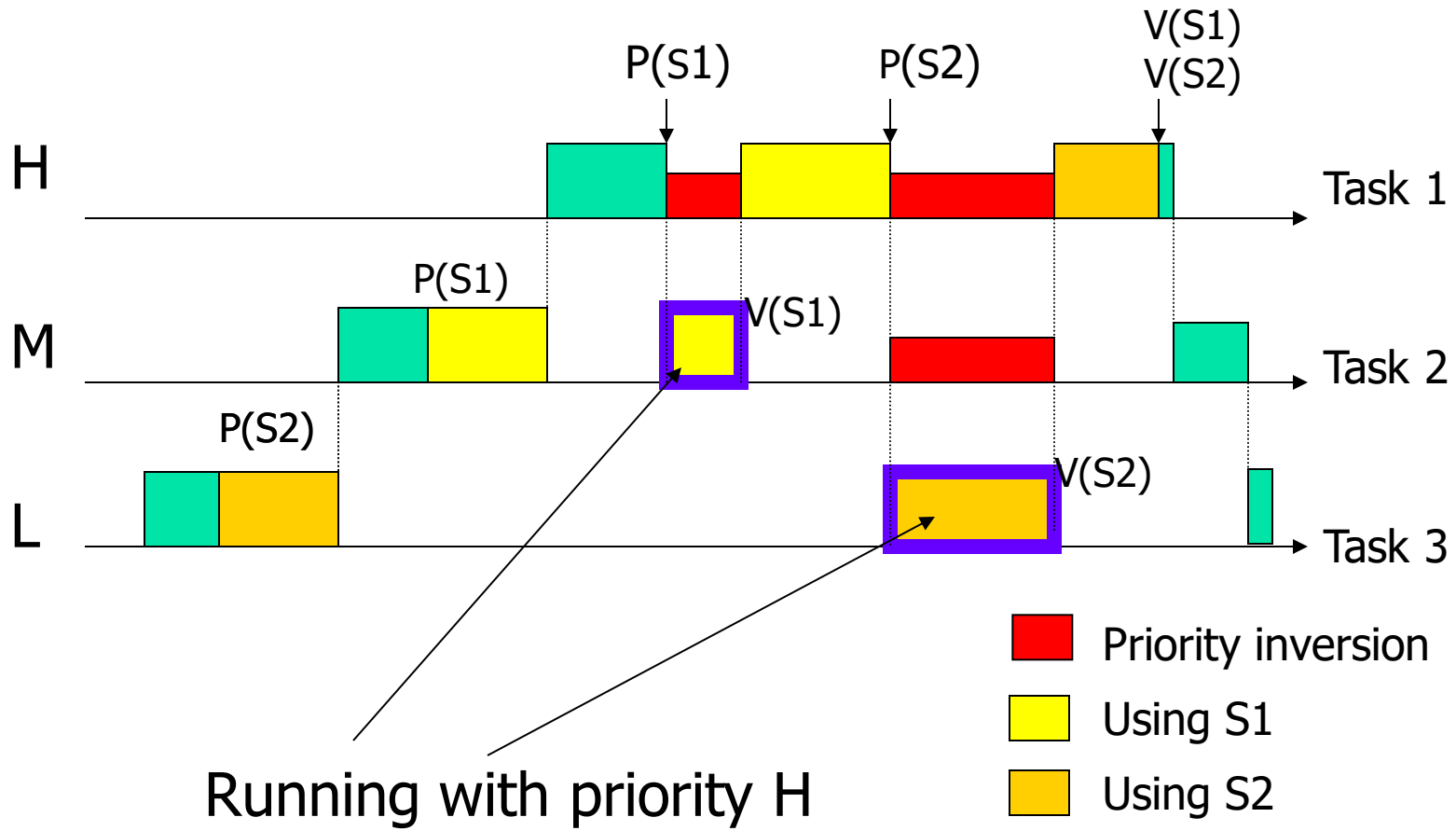


Property 1: Deadlock free (HLP)



Once task 2 gets S1, it runs with pri H, task 1 will be blocked (no chance to get S2 before task 2)

BIP: Chained blocking – many preemptions



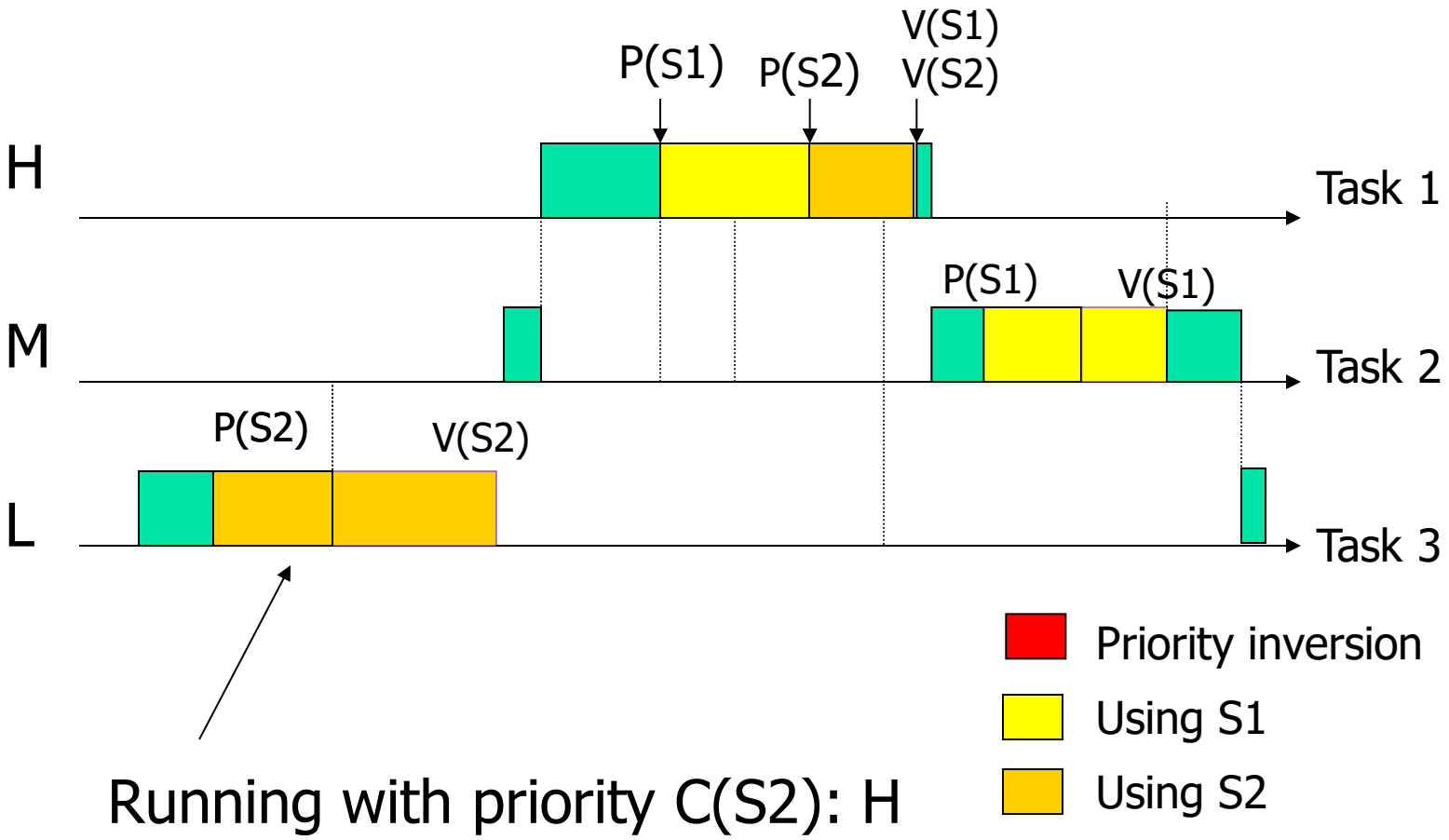
Task H needs M resources may be blocked M times:

→ many preemptions/run-time overheads

→ maximal blocking= at least, the sum of all CS sections for lower-priority tasks

C(S1)=H
C(S2)=H

Property 2: at most one blocking (HLP)



HLP: Blocking time calculation

$B(i)=\max\{CS(k,i,S) \mid pri(k)<pri(i)<=C(S)\}$

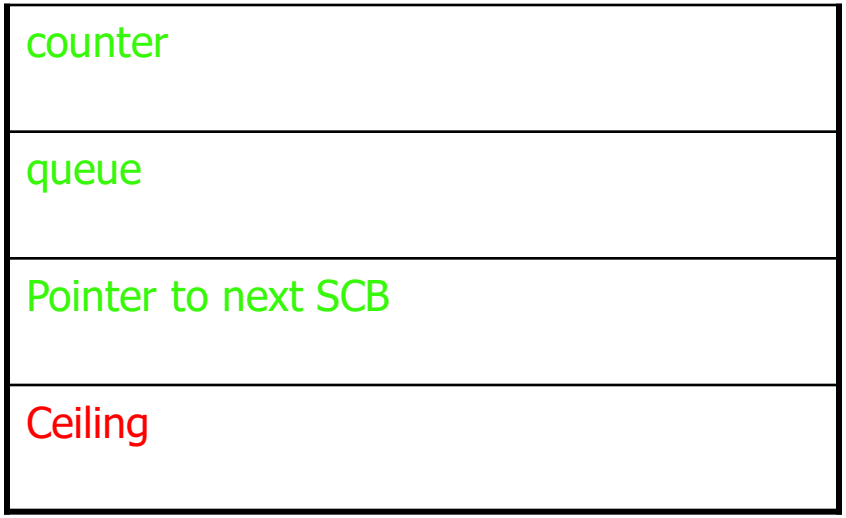
where

- B(i) is the maximal time that task i may be blocked due to HLP
- CS(k, i, S) is the length of a critical section that may be locked by task k and i with S.

Implementation of HLP

- Calculate the ceiling for all semaphores
- Modify SCB
- Modify P and V-operations

Semaphore Control Block for HLP



P-operation with HLP

- P(scb):

```
Disable-interrupt;
If scb.counter>0 then
        { scb.counter - -1;
         save(current-task.priority);
         current-task.priority := Ceiling(scb) }
else
    {save-context();
     current-task.state := blocked
     insert(current-task, scb.queue);
     dispatch();
     load-context() }
Enable-interrupt
```

V-operation with HLP

- V(scb):

```
Disable-interrupt;
"restore previous priority"
If not-empty(scb.queue)  then
        next-to-run := get-first(scb.queue);
        next-to-run.state := ready;
        next-to-run.priority := Ceiling(scb);      /*this is not necessary! */
        insert(next-to-run, ready-queue);
        save-context();
        schedule(); /* dispatch invoked*/
        load-context();
 end then
 else scb.counter ++1;
 end else
Enable-interrupt
```

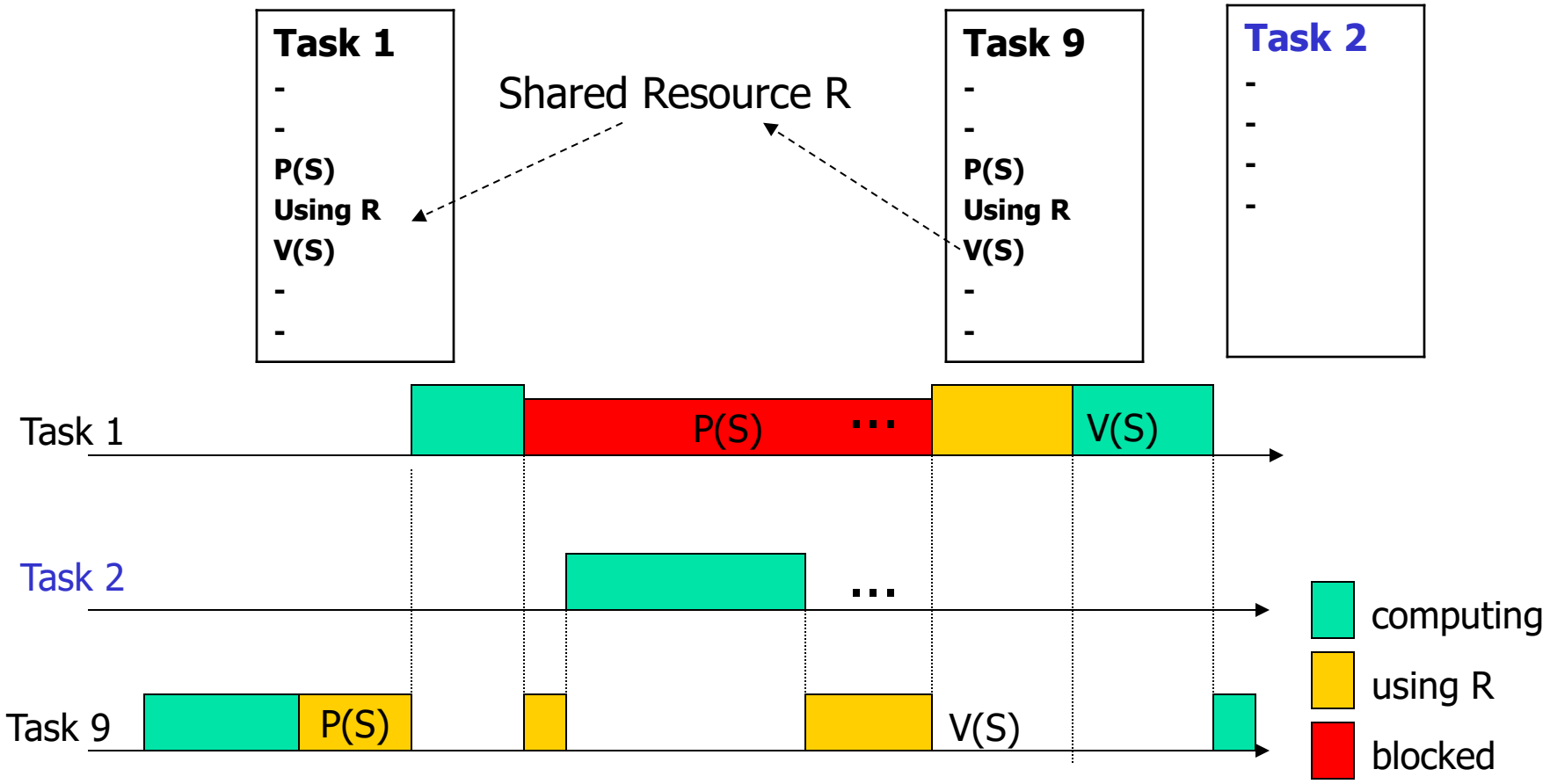
Properties of HLP: + and -

- Bounded priority inversion
- Deadlock free (+), Why?
- Number of blocking = 1 (+), Why?
- The extreme case of HLP=Non-preemtion (-)
  - E.g when the highest priority task uses all semaphors, the lower priority tasks will inherit the highest priority

Summary

	NPP	BIP	HLP
Bounded Priority Inversion	yes	yes	yes
Avoid deadlock	yes	no	yes
Avoid Un-necessary blocking	no	yes	yes/no
Blocking time calculalation	Easy	hard	easy

Un-bounded priority inversion



Priority Ceiling Protocol (combining HLP and BIP)

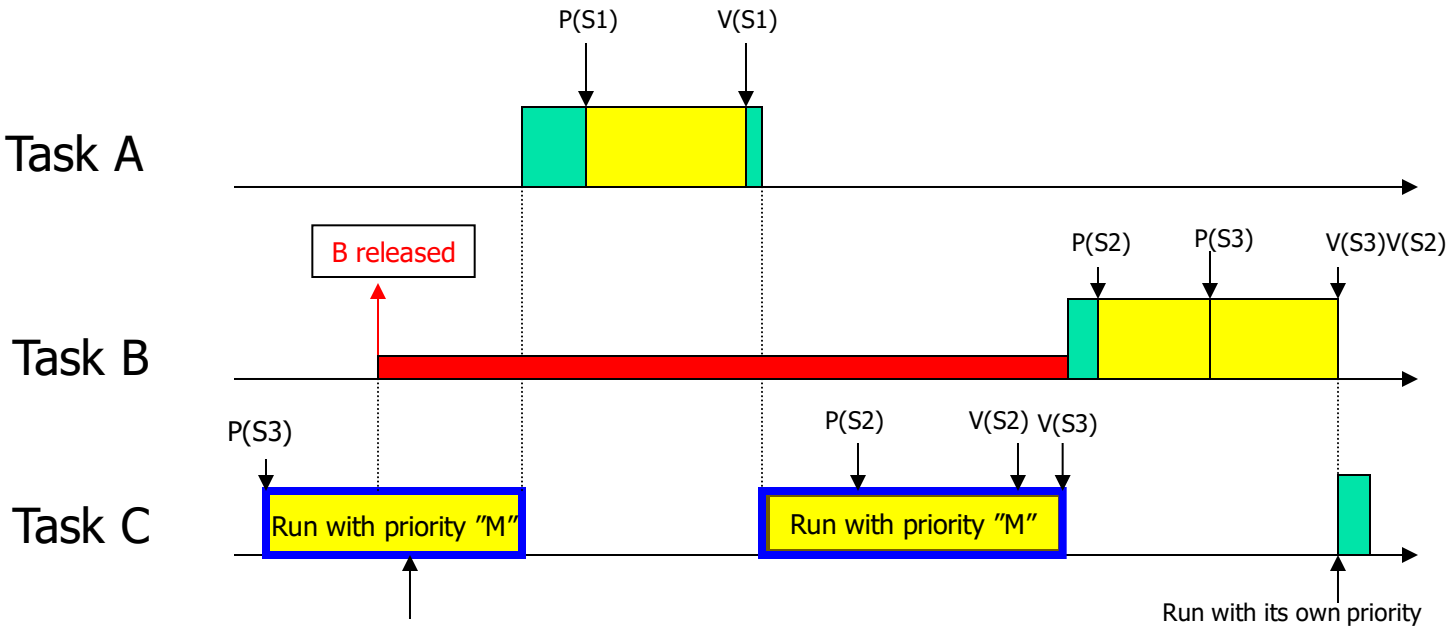
- Each semaphore S has a Ceiling C(S)
- Run-time behaviour:
  - Assume that S is the semaphore with highest ceiling of all semaphores locked by *other* tasks currently:
    - C(S) is ”the current system ceiling”
  - If A wants to lock any semaphore, it must have a strictly higher priority than C(S) i.e. Pri(A) > C(S). Otherwise A is blocked, and it transmitts its priority(+ε) to the task currently holding S

Example: HLP

A: ...P(S1)...V(S1)...
B: ...P(S2)...P(S3)...V(S3)...V(S2)...
C: ...P(S3)...P(S2)...V(S2)...V(S3)

Prio(A)=H
Prio(B)=M
Prio(C)=L

C(S1)=H
C(S2)=C(S3)=M

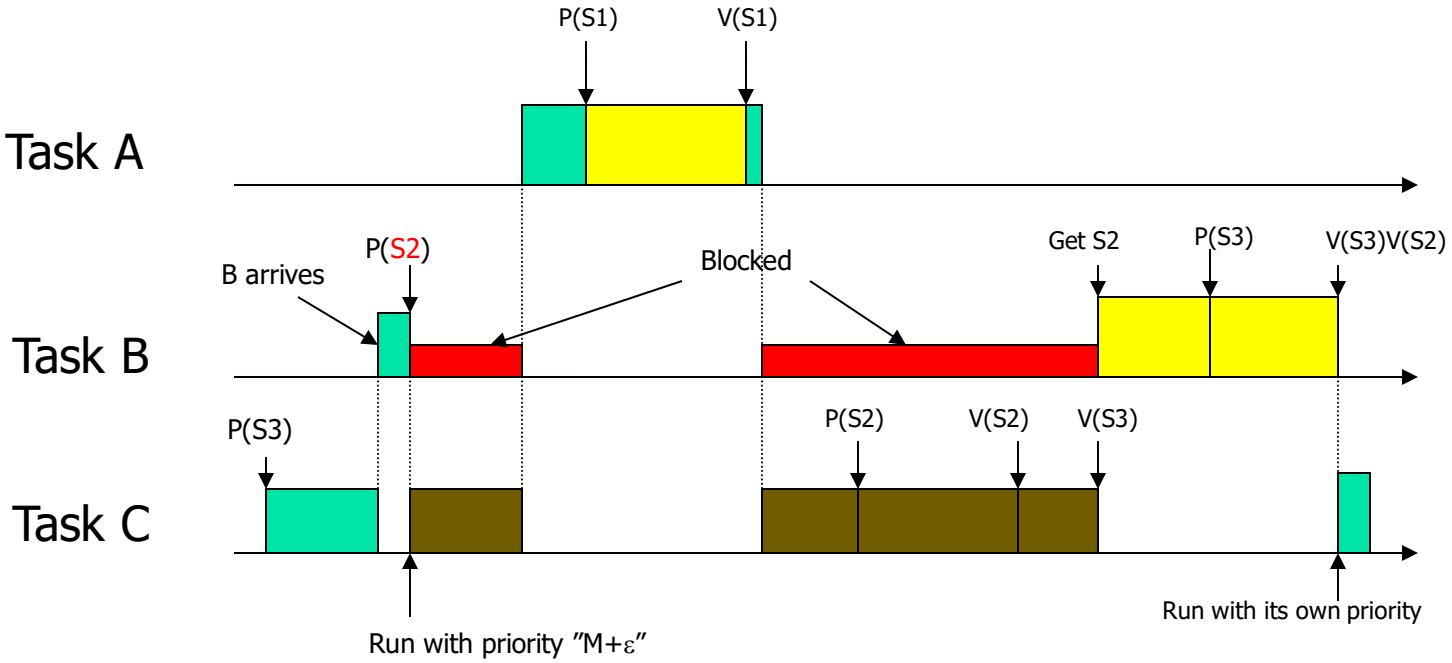


Example: PCP

A: ...P(S1)...V(S1)...
B: ...P(S2)...P(S3)...V(S3)...V(S2)...
C: ...P(S3)...P(S2)...V(S2)...V(S3)

Prio(A)=H
Prio(B)=M
Prio(C)=L

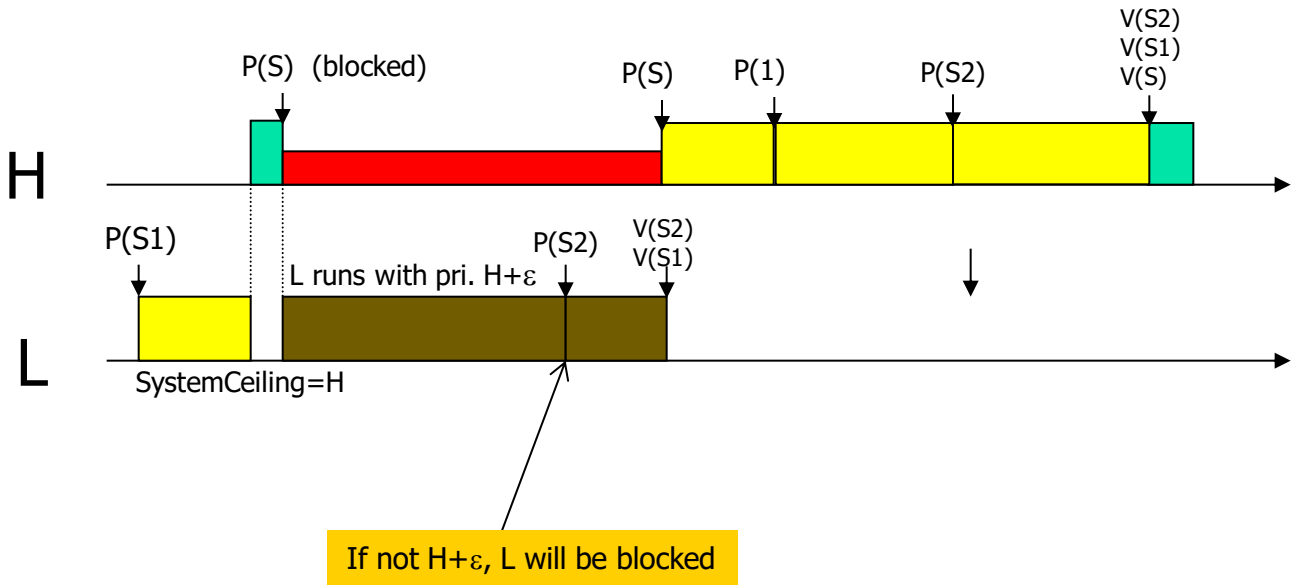
C(S1)=H
C(S2)=C(S3)=M



Why adding ε?

Example: assume

- tasks H and L (H has higher priority than L)
- both share semaphores S1, S2
- H uses S (not shared)
- Thus, ceiling(Si)= H and
- SystemCeiling = H if any semaphore is locked



PCP: Blocking time calculation

(the same as HLP)

$B(i)=\max\{CS(k,i,S) \mid pri(k)<pri(i)<=C(S)\}$

where

- B(i) is the maximal time that task i may be blocked due to HLP
- CS(k, i, S) is the length of a critical section that may be locked by task k and i with S.

Exercise: implementation of PCP

- Implement P,V-operations that follow PCP
- (this is not so easy)

Properties of PCP: + and -

- Bounded priority inversion (+)
- Deadlock free (+)
- Number of blocking = 1 (+)
- Better response times for high priority tasks (+)
  - Avoid un-necessary blocking
- Not easy to implement (-)

Summary

	NPP	BIP	HLP	PCP
Bounded Priority Inversion	yes	yes	yes	yes
Deadlock free	yes	no	yes	yes
Un-necessary blocking	yes	no	yes/no	no
Blocking time calculalation	easy	hard	easy	easy
Number of blocking	1	>1	1	1
Implementation	easy	easy	easy	hard