Priority Inheritance Protocols: An Approach to Real-Time Synchronization

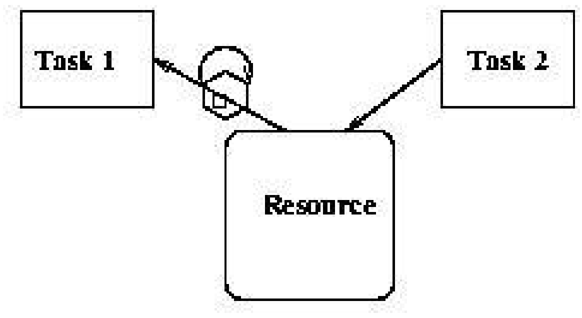
Lui Sha, Raghunathan Rajkumar and John Lehoczky
IEEE Trans. on Computers, September 1990, vol. 39 no. 9, pp. 1175-1184

Key Result: *Use of synchronization mechanisms like semaphores, monitors, critical regions, can lead to uncontrolled priority inversion. This paper reports two protocols that resolve this issue and further the authors derive a modified schedulability analysis to accommodate the effects of these protocols.*

Introduction:

- Why do we need synchronization?
- In the liu and layland paper we assumed that tasks are *independent*, this assumption is clearly simplistic as interaction between tasks will be needed in almost all meaningful applications. Such interaction may happen in one of several ways:
  - IPC using pipes, message queues
  - Shared memory regions
  - Shared resources
  - Socket communication
  - Remote Method Invocation

  Interactions that need synchronization are typically one's where a mutually exclusive access to a shared resource (variable or memory or physical entity) are involved. Such accesses are performed using semaphores, locking mechanisms, monitors or critical regions (please review your understanding of theses OS concepts). The basic idea is as simple as locking something that a process is currently using so that no other process can obtain it:



- *Critical Section (using semaphores):*

```
P(S)

      Critical
      Section
      (Use
      Resource
      )

V(S)
```

P and V operations are wait and signal functions and their code is as follows:

```
P(S)
{
    while (S <= 0) do no-op;
    S := S - 1;
}
```
	```
	V(Sem)
	{
	   S := S + 1;
	}
	```

<u>The Priority Inversion Problem</u>

- *Priority Inversion is the phenomenon where a higher priority task is blocked by lower priority tasks.*

For example, a situation could arise in which a lower priority task uses a resource that is guarded by a *semaphore*, then gets preempted by a tasks with a higher priority. If that higher priority task tries to lock the semaphore, it will be delayed until the semaphore is released, and so it could be delayed past its deadline. In the example below, *task 1* is blocked while attempting to lock resource *S1*, which is already locked by *task 3*. This priority inversion is necessary to retain the integrity of the data being shared. Now, *task 1* can be further delayed while the medium priority *task 2* that does not need the shared resource continues to preempt the low-priority task.

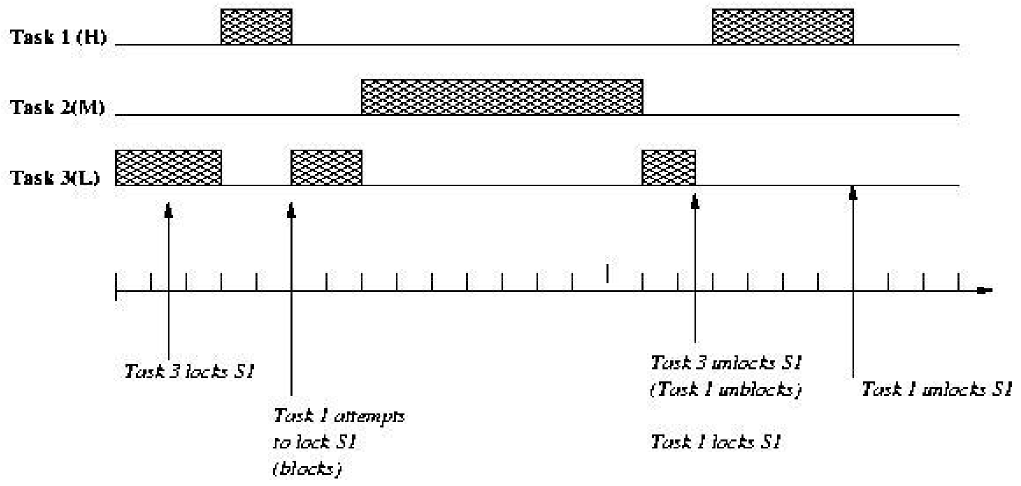


Figure: *Unbounded Priority Inversion*

The delay experienced by a task due to priority inversion is also referred to as *blocking.*

<u>Assumptions</u>

- A job(task) is a sequence of instructions
- Jobs do not suspend themselves, say for doing I/O operations
- The *critical sections* of a job are *properly* nested and a job will release all of its locks, if it holds any, before or at the end of its execution.
- A fixed priority scheduling mechanism is used
- Jobs are listed as *J1, J2, ... , Jn* in descending order of priority with *J1* having the highest priority.
- When a job *J* is forced to wait for the execution of lower priority jobs, job *J* is said to be "blocked".

<u>Notation</u>

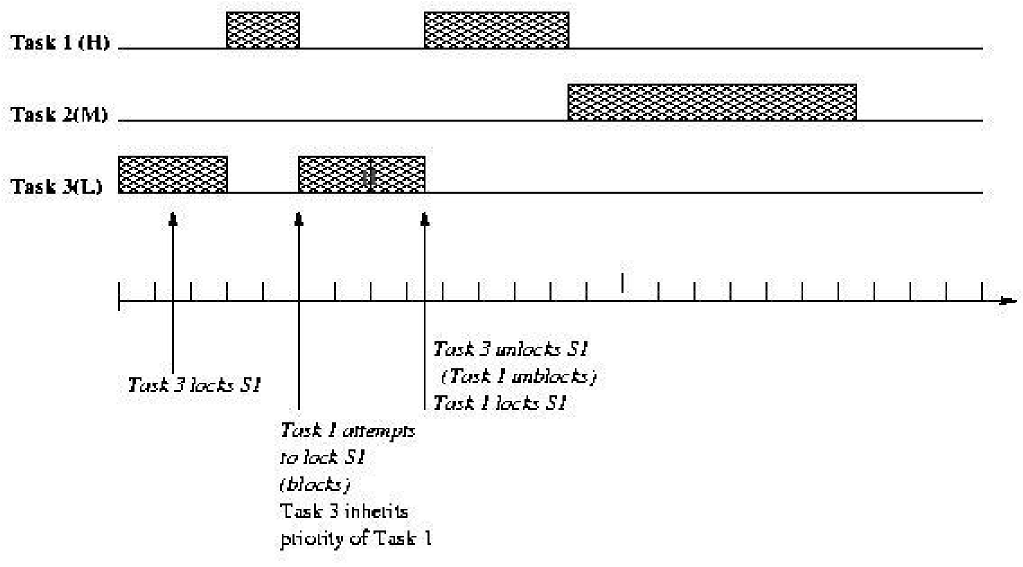
- *Ci, Pi* and *Ti* denote the execution time requirement, priority and periodicity respectively of job *Ji.*
- A binary semaphore guarding shared data and/or resource is denoted by *Si*.
- The critical section in job *Ji* that is guarded by semaphore S_j is denoted by *Zi,j* and corresponds to the *P* operation and its corresponding *V* operation on the

semaphore.

- We write *Zi,j* << *Zi,k* if the critical section *Zi.j* is entirely contained in *Zi,k.*
- The duration of the critical section *Zi,j* denoted by *Di,j* is the time to execute *Zi,j* when *Ji* executes on the processor alone.
- The set $ß^*_{i,j}$ denotes the longest critical sections of job *Jj* that can block *Ji*
- $ß^*_i$ denotes the set of all longest critical sections that can block *Ji* ($ß^*_i = \bigcup_{j>i} ß^*_{i,j}$)

The Basic Priority Inheritance Protocol (PIP)

When job J blocks one or more higher priority jobs, it ignores its original priority assignment and executes its critical section at the highest priority level of all the jobs it blocks.


Task 1 (H)
Task 2(M)
Task 3(L)
Task 3 locks S1
Task 1 attempts to lock S1 (blocks) Task 3 inherits priority of Task 1
Task 3 unlocks S1 (Task 1 unblocks) Task 1 locks S1

The Protocol

1. Before a job *J* enters a CS, it must first obtain a lock on the semaphore *S* guarding the CS. *J* will be blocked if the semaphore is already locked. Otherwise *J* will obtain the lock and enter its CS. When *J* exits the CS, the lock is released and the highest priority job (if any) blocked by *J* is awakened.
2. A job *J* uses its assigned priority, unless it is in its CS and blocks higher priority jobs. In which case, *J* inherits P_H, the highest

priority of the jobs blocked by *J*. When *J* exits the CS, it resumes the priority it had at the point of entry into the CS.
3. Priority inheritance is *transitive*.

In summary, under PIP, a high priority job can be blocked by a lower priority job in one of two situations:

- *Direct* blocking
- *Push-through* blocking: A medium priority job can be blocked by a low priority job which inherits the priority of a high priority job.

Properties

- Given a job *J0* for which there are *n* lower priority jobs *{J1,.... Jn}*, job *J0* can be blocked for at most the duration of one CS in each of $ß^*_{0,i}$, $1 <= i <= n$;
- If there are *m* semaphores which can block job *J*, then *J* can be blocked by at most *m* times.

Shortcomings

- Does not prevent deadlocks
  - E.g., job *J2* locks *S2* at time t1 and enters its CS. At time t2, job *J2* attempts to make a nested access to lock *S1*. However, job *J1* a higher priority job is ready and preempts *J2* and locks *S1*. Next, *J1* tries to lock *S2* and deadlocks.
- Blocking duration is bounded but substantial
  - Due to chain blocking

The Priority Ceiling Protocol

- Idea:
  - A *priority ceiling* is assigned to each semaphore, which is equal to the highest priority task that *may* use the semaphore.
  - Job *J* is allowed to start a new CS only if *J*'s priority is higher than all priority ceilings of all the semaphores locked by jobs other than *J*.
- Prevents Deadlocks

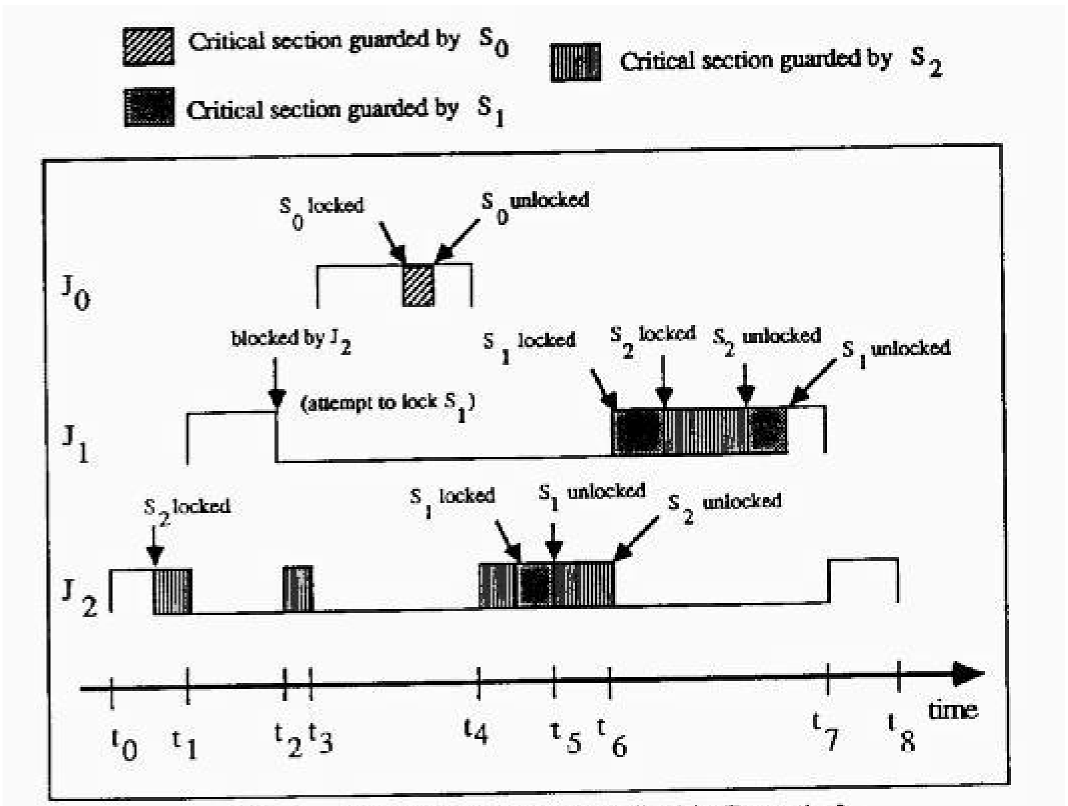


Fig. 1. Sequence of events described in Example 2.

- Chained Blocking avoided

The Protocol

1. Let S^* be the semaphore with the highest priority ceiling of all semaphores currently locked by the jobs other than job *J*. If job *J* attempts to enter its CS guarded by semaphore *S*, it will be blocked *if*
   1. the priority of job *J* is not higher than the priority ceiling of semaphore S^*

   otherwise

   2. job *J* will obtain a lock on *S* and enter its CS.

   When *J* exits its CS the semaphore will be released and the highest priority job (if any) blocked by *J* will be awakened.

2. A job *J* uses its assigned priority, unless it is in its CS and blocks higher priority jobs. In which case, *J* inherits P_H, the highest priority of the jobs blocked by *J*. When *J* exits the CS, it

resumes the priority it had at the point of entry into the CS.

Illustration by example:

- Job *J0* accesses $Z_{0,0}$ and $Z_{0,1}$ by executing the steps {. . . , ***P(S0)***, . . . , ***V(S0)***, . . . , ***P(S1)***, . . . , ***V(S1)***, . . .}
- Job *J1* accesses only $Z_{1,2}$ by executing {. . . , ***P(S2)***, . . . , ***V(S2)***, . . .}
- Job *J2* accesses $Z_{2,2}$ and makes a nested semaphore access to *S1* by executing the steps {. . . , ***P(S2)***, . . . , ***P(S1)***, . . . , ***V(S1)***, . . . , ***V(S2)***, . . .}

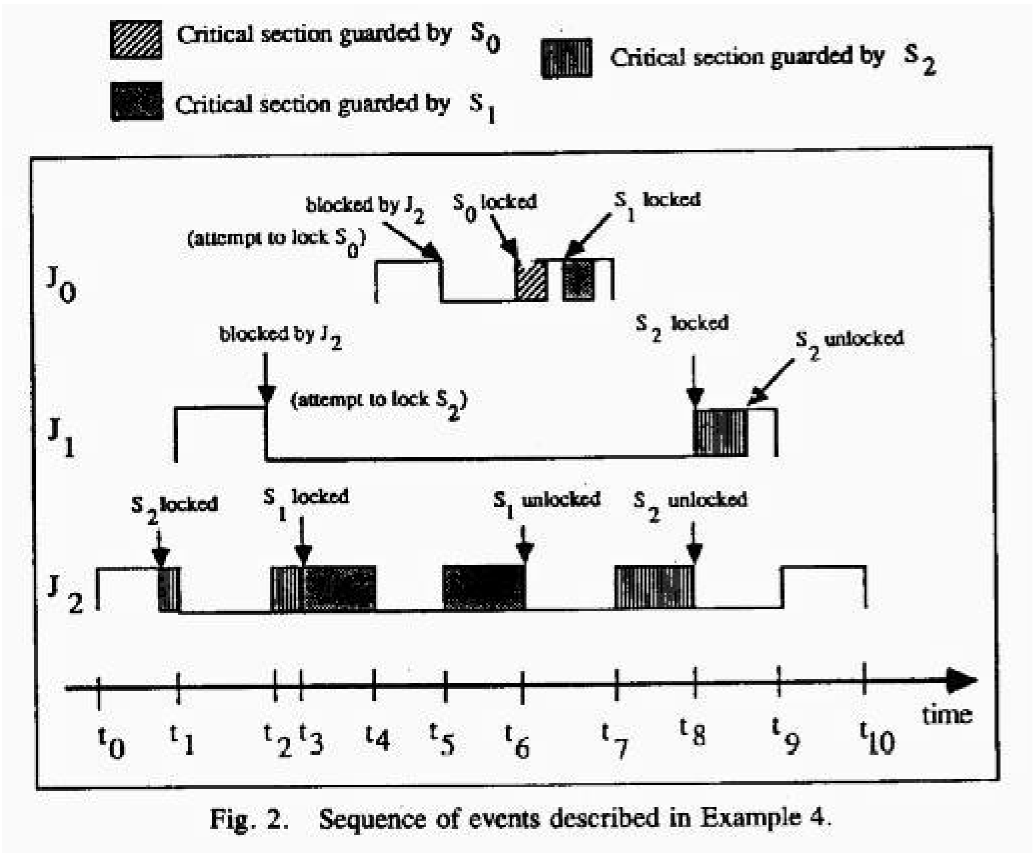

Critical section guarded by S_0
Critical section guarded by S_2
Critical section guarded by S_1
blocked by J_2 (attempt to lock S_0)
S_0 locked
S_1 locked
J_0
blocked by J_2
(attempt to lock S_2)
S_2 locked
S_2 unlocked
J_1
S_2 locked
S_1 locked
S_1 unlocked
S_2 unlocked
J_2
t_0 t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9 t_{10}
time

Fig. 2. Sequence of events described in Example 4.

The priority ceiling protocol introduces a third kind of blocking in addition to *direct* blocking, and *push-through* blocking, called *ceiling* blocking. In the above example, an instance of this occurs at time t5 when *J2* blocks *J0*.

Properties

- PCP prevents deadlocks
- A job *J* can be blocked for at most the duration of at most one element of $ß^*_i$