
Jim Ras and Albert Mo Kim Cheng

Outline

- Handling shared resources and mutual exclusion
- Priority inversion
- Resource access protocols
  - Uniprocessor systems (PIP, PCP, SRP)
    - Direct blocking
    - Push-through blocking
  - Multiprocessor systems (MPCP, MSRP)
    - Remote blocking
- Ada-2005, results and conclusion

15th IEEE Real-Time and Embedded Technology and Applications Symposium
San Francisco, CA
April 13 - April 16, 2009
Types of constraints

• Timing constraints
  – activation, completion, jitter.

• Precedence constraints
  – impose an ordering in the execution.

• Resource constraints
  – enforce a synchronization in the access of mutually exclusive resources.
Resource constraints

- To preserve data consistency, shared resources must be accessed in mutual exclusion:
Mutual exclusion

- However, mutual exclusion introduces extra delays:
Handling shared resources

- Problems caused by mutual exclusion

Blocking on a semaphore

$\tau_1$  $\tau_2$

CS  CS

$p_1 > p_2$

$\tau_1$  $\tau_2$

$\Delta$

It seems that the maximum blocking time for $\tau_1$ is equal to the length of the critical section of $\tau_2$, but …
Priority Inversion

A high priority task is blocked by a lower-priority task for an unbounded interval of time.

Solution

Introduce a concurrency control protocol for accessing critical sections.
Priority Inversion:
Conflict on a critical section

A high-priority task is blocked waiting on a resource being used by a lower-priority task, which itself is blocked by an unrelated medium-priority task.
Resource Access Protocols

- **Under single processor systems**
  - Non-Preemptive Protocol (NPP)
  - Highest Locker Priority (HLP)
  - Priority Inheritance Protocol (PIP)
  - Priority Ceiling Protocol (PCP)
  - Stack Resource Policy (SRP)

- **Under multiprocessor systems**
  - Multiprocessor Priority Ceiling Protocol (MPCP)
  - Multiprocessor Stack Resource Policy (MSRP)
Non Preemptive Protocol

- Preemption is forbidden in critical sections
- Implementation: when a task enters a CS, its priority is increased to the maximum value.

Advantages: simplicity
Problems: high-priority tasks that do not use CS may also block
Schedule with NPP

\[ P_{CS} = \max\{P_1, \ldots, P_n\} \]
Remarks on the NPP

Priority

\( \tau_1 \)

\( \tau_2 \)

\( \tau_3 \)

unnecessary Blocking

\( \tau_1 \) cannot preempt even though it does not use the CS
Highest Locker Priority

A task in a CS gets the highest priority among the tasks that use it.

FEATURES:

- Simple implementation.
- A task is blocked when attempting to preempt, not when entering the CS.
Schedule with HLP

\[ P_{CS} = \max \{ P_k | \tau_k \text{ uses CS} \} \]

\( \tau_2 \) is blocked, but \( \tau_1 \) can preempt within a CS
Priority Inheritance Protocol (PIP)

[Sha, Rajkumar, Lehoczky, 90]

- A task in a CS increases its priority only if it blocks other tasks.
- A task in a CS inherits the highest priority among those tasks it blocks.

\[ P_{CS} = \max \{ P_k \mid \tau_k \text{ blocked on CS} \} \]
Types of blocking

- Direct blocking
  A task blocks on a locked semaphore

- Push-through blocking
  A task blocks because a lower priority task inherited a higher priority.

**BLOCKING:**
a delay caused by a lower-priority task
Schedule with PIP

priority

τ₁

τ₂

τ₃

p₁

p₃

direct blocking

push-through blocking
Remarks on the PIP

**ADVANTAGES**
- It is transparent to the programmer.
- It bounds priority inversion.

**PROBLEMS**
- It does not avoid deadlocks and chained blocking.
Chained blocking with PIP

Theorem: $\tau_i$ can be blocked at most once by each lower-priority task.
Priority Ceiling Protocol

- Can be viewed as PIP + access test.
- A task can enter a CS only if it is free and there is no risk of chained blocking.

To prevent chained blocking, a task may stop at the entrance of a free CS (ceiling blocking).
Resource Ceilings with PCP

- Each semaphore $s_k$ is assigned a ceiling:

$$C(s_k) = \max \{ P_j : \tau_j \text{ uses } s_k \}$$

- A task $\tau_i$ can enter a CS only if

$$P_i > \max \{ C(s_k) : s_k \text{ locked by tasks } \neq \tau_i \}$$
Schedule with PCP

\[ t_1: \tau_2 \text{ is blocked by the PCP, since } P_2 < C(s_1) \]
Deadlock avoidance with PCP

\[ \tau_1 \quad \tau_2 \]

\[ P_1 > P_2 \]

- \( C_A = P_1 \)
- \( C_B = P_1 \)

ceiling blocking
Remarks on the PCP

ADVANTAGES

- Blocking is reduced to only one CS
- It prevents deadlocks

PROBLEMS

- It is not transparent to the programmer: semaphores need ceilings
Guarantee with RM ($D = T$)

- $C_i$: Worst-case execution time (WCET) of task $\tau_i$
- $T_i$: Period of task $\tau_i$
- $B_i$: Worst-case blocking time

Liu and Layland Test

$$\forall i \sum_{k=1}^{i-1} \frac{C_k}{T_k} + \frac{C_i + B_i}{T_i} \leq i(2^{1/i} - 1)$$
Stack Resource Policy (SRP)

- Each task is assigned a preemption level. Preemption levels reflect the relative deadlines of the tasks.

- At run-time, resources are given ceiling values based on the maximum preemption level of the tasks that use the resource.

- Under the SRP, a task is blocked at the time it attempts to preempt.

- Under the PCP, a task is blocked at the time it makes a resource request and the resource is not available.
Remarks on the SRP

Advantages
- SRP is a consistent extension of the PCP
- Fewer context switches than the PCP

Problems
SRP's early blocking policy causes more tasks to miss their deadlines, even those tasks that require no shared resources to complete.
EDF Guarantee ($D_i = T_i$)

$\forall i \sum_{k=1}^{i-1} \frac{C_k}{T_k} + \frac{C_i + B_i}{T_i} \leq 1$
Remote Blocking

- J4 wants to lock S, but S is locked by J3.
  - J4 must also wait for J1 and J2 to complete.
- Must prevent J1 and J2 from running until J3 unlocks S.
Multiprocessor Priority Ceiling Protocol (MPCP)

An extension of the PCP
Solves the remote blocking problem

[Sha, Rajkumar, Lehoczky, 90]

- A task in a local CS increases its priority only if it blocks other tasks.
- A task in a local CS inherits the highest priority among those tasks it blocks.
- A task in a global CS increases its priority to the CS priority as it enters the CS to reduce remote blocking.

Local CS (uses the PCP)

\[
P_{CS_{local}} = \max \{ P_k \mid \tau_k \text{ blocked on CS} \}
\]

Global CS

\[
P_{GS_i} = P_H + p(J_{GS_i}) + 1
\]

\[
P_H = \text{highest-priority task in the system}
\]

\[
p(J_{GS_i}) = \text{priority of highest-priority task that will use GS}_i
\]
Multiprocessor Stack Resource Policy (MSRP)

An extension of the SRP

[Gai et al 2003]

- For local resources, MSRP is the same as the SRP.
- When a task is allocated to a processor and locks a global resource, the task becomes non-preemptable. (about the same idea as MPCP)
Ada-2005

- Ada in its Annex on System Programming and Annex on Real-Time Systems, provides the features for implementing MPCP and MSRP.

- Ada Protected Objects provide mutually exclusive access to data.

  - In our implementation, the protected object provides two operations in its public interface: the entry Request and the procedure Release.

  - The Request operation is implemented as an entry since it contains a requeue statement.

- The requeue statement can be used to put the calling task in the queue of the requeue until the requested resource becomes available.
Beyond Ada 2005:
Allocating Tasks to Processors

- Ada allows a program’s implementation to be on a multiprocessor system. However, it provides no direct support that allows programmers to partition their tasks onto the processor in the given system [Wellings and Burns - University of York].

- Since Kernel version 2.5.8, Linux has provided support for SMP systems via the notion of CPU affinity. Each process in the system can have its CPU affinity set according to a CPU affinity mask.

- Thus, we took an approach using the Ada **pragma** feature to interface Ada to C functions that worked with the underlying system libraries. Accordingly, Linux processes are used instead of tasks.
Allocating Tasks to Processors

package SMP is -- not all features shown
    -- determine how many processors are available
    function available_cpus return integer;
    pragma Import (C, available_cpus);

    -- set the affinity for a processor
    function set_affinity(cpu_n: natural; pid: natural) return integer;
    pragma Import (C, set_affinity);

    -- get the affinity of a process
    function get_affinity return integer; -- returns the task's affinity
    pragma Import (C, get_affinity);
end SMP;
The set_affinity function

```c
int set_affinity (int cpu_n, pid_t pid) {  // cpu_n : which cpu  // pid: the process id

    int num_procs = sysconf (_SC_NPROCESSORS_CONF);
    if((cpu_n < 0) || (cpu_n > (num_procs-1))) return Affinity_Error;

    cpu_set_t mask;
    // set all the bits in the mask to zero
    CPU_ZERO (&mask);

    // set only the bit corresponding to cpu_n
    CPU_SET(cpu_n, &mask);

    // set the CPU affinity mask of the task denoted by pid
    return sched_setaffinity (pid, sizeof (mask), &mask);
}
```
Experimental results

- General purpose computer system
- SMP architecture hosting a version of Linux 2.6.18
- The x86_64 system has 8 processors

The graphs show the percentage of task sets that can be guaranteed to be schedulable.

- The first set of tests were performed on task sets with randomly chosen task periods.
- The MPCP policy performed better than MSRP on all tests
- The advantage of MPCP is its closer adherence to priority scheduling
Task sets with randomly selected periods and lighter resource usage

The graphs show the percentage of task sets that can be guaranteed to be Schedulable.

With fewer and shorter critical sections, MSRP performed better than before.

MSRP’s performance improved when the number of critical sections was reduced by a half and their lengths shortened.
Percentage of schedulable solutions, harmonic periods, and variable percentage of resource utilizations

The graph shows that EDF scheduling still managed to deliver better performance.

The MPCP and the MSRP performed equally well.

MSRP’s early blocking policy caused more tasks to miss their deadlines.
A comparison of MPCP and MSRP with respect to priority changes and context switches

With MPCP the priority of a task is only raised when it is necessary to reduce priority inversion. However, MSRP’s overhead is higher.

With MSRP, a positive consequence of the early blocking policy of MSRP is a reduction in unnecessary context switches.

Priority change is much simpler than context switching, but only if implemented over kernel-based threads.
Conclusion

- MSRP is a consistent extension of MPCP.

- With MSRP, a task that locks a critical resource will always have its priority raised to that of the ceiling priority of the critical resource, throughout the protected operation.

- One of the strengths of MPCP is that it does not raise priority until it is necessary to prevent priority inversion, so in many cases, the priority need not change. For this reason, MPCP adheres to priority scheduling better. However, there is still some overhead to determine whether a priority change is needed or not.

- With MSRP, whenever we raise the priority of the lock-holder that is causing a temporary priority inversion, we gain over ‘no protocol’ by the fact that this inversion is strictly bounded in duration.
Future work

- Currently it is not possible to map tasks to processors with Ada or Real-time Java. However, in the next full revisions it will be possible. We plan to test both protocols and compare their performance and the performance of Ada and Java with respect to each other.