Foundations of Cyber-Physical Systems

Real-Time Synchronization

Rupak Majumdar & Björn Brandenburg

Task Coordination and Synchronization

So far we have assumed that tasks are independent. However, this often not the case in practice.

Two types of coordination:

1. **mutual exclusion constraints** — resource sharing with locks
   → `mutex_lock() / mutex_unlock()`

2. **precedence constraints** — when one job must wait for another
   → producer-consumer or `signal() / wait()` relationships
   → `pipe()` or `socket()` + blocking `read()`
Mutual Exclusion

How to deal with delays due to locking?
Need for Mutual Exclusion

Classic problem: prevent the interleaving of *critical sections* to ensure *atomicity* of multiple read/write accesses.

Examples of **shared resources**:

- accessing control registers of an I/O device
- shared OS services (e.g., timer facility, run queue)
- shared data structures (e.g., history of sensor data)
- message buffers (e.g., `pipe()` implementation)
How to Realize Mutual Exclusion

Three main options:

- **use a static schedule** that prevents interleaving of accesses
  → good solution when possible

- **use locks** *(or binary semaphores)* to block interleaving of accesses
  → need to analyze extra delays due to blocking! (= blocking analysis)

- restrict access to a single sequential task *(a server task)* and invoke resource server via *inter-process communication (IPC)*
  → need to analyze delay due to message backlog! (= blocking analysis)
The Key Locking Problem — Priority Inversion

Priority-based scheduling: at any time, the highest-priority incomplete job should be scheduled.
→ this assumption is the basis of all schedulability analysis!

Problem: what if the highest-priority job requires a lock?

Priority inversion: a job should be scheduled but is not.
→ On a uniprocessor: a lower-priority job is scheduled instead.
→ Only possible if some lower-priority job holds a required lock
• $T_1$ is delayed for at least $e_2$ time units, the entire WCET of $T_2$.
  → An “unbounded” priority inversion: $e_2$ is unbounded in general

• No useful response times with “unbounded” priority inversions.
Priority Inversion — Does it matter?
Mars Pathfinder Mission\textsuperscript{Jo97} (1997)

- Repeated \textit{spurious resets} shortly after “perfect landing” when starting to collect \textit{meteorological data}.

- Central shared-memory buffer (“information bus”) to distribute data within system, \textit{protected by a mutex}.

- \textit{Meteorological data gathering task}: infrequent, \textit{low priority}, holding mutex while writing data

- \textit{Communications task}: \textit{medium priority}

- \textit{Bus management task}: frequent, high priority, monitored by a \textit{watchdog timer} (= check deadline)
  → priority inversion if data gathering task preempted
  → watchdog triggered = system reset

\textsuperscript{Jo97} M. Jones (1997). \textit{What really happened on Mars?}
How was it fixed?

"When created, a VxWorks mutex object accepts a boolean parameter that indicates whether priority inheritance should be performed by the mutex. The mutex in question had been initialized with the parameter off [...] VxWorks contains a C language interpreter intended to allow developers to type in C expressions and functions to be executed on the fly during system debugging. The JPL engineers fortuitously decided to launch the spacecraft with this feature still enabled. By coding convention, the initialization parameter for the mutex in question (and those for two others which could have caused the same problem) were stored in global variables, whose addresses were in symbol tables also included in the launch software, and available to the C interpreter. A short C program was uploaded to the spacecraft, which when interpreted, changed the values of these variables from FALSE to TRUE. No more system resets occurred."

— Mike Jones\(^{97}\) (emphasis added)

\(^{97}\) M. Jones (1997). What really happened on Mars?
Terminology: PI-Blocking

• Priority inversion = increased worst-case response time.

• With locks, priority inversions coincide with **blocking**.

• But not all “blocking” (in the OS sense) is a priority inversion.

To be specific, we refer to *priority inversion blocking* (**pi-blocking**).
Observation: Critical Sections of Higher-Priority Jobs \textbf{Never} Cause PI-Blocking

Why?
Observation: Critical Sections of Higher-Priority Jobs *Never* Cause PI-Blocking

*Priority inversion*: a job should be scheduled but is not.

Recall pi-blocking definition: a job $J_i$ incurs pi-blocking when it is *incomplete* and a *lower-priority* job is scheduled instead.

While a higher-priority job $J_h$ executes, $J_i$ is expected to be delayed anyway—$J_i$ *should not* be scheduled while $J_h$ is incomplete, regardless of whether $J_h$ executes a critical section or not.
How to Ensure **Bounded** PI-Blocking?

$b_i$ ... maximum total pi-blocking incurred by any job of $T_i$.

- How to ensure $b_i$ is bounded (& small) in all possible schedules?

- A **real-time locking protocol** restricts the sets of possible schedules.

- Locking protocol: a combination of
  - regular binary semaphores (= mutexes or just locks) and
  - a **progress mechanism** to expedite completion of critical sections.
The Simple Solution

Non-Preemptive Sections (NPS)
Non-Preemptive Sections (NPS)

Observation: on a uniprocessor, resource contention is always preceded by the preemption of a running, resource-holding job.

Solution: simply disable preemptions before acquiring a shared resource; reenable upon exit of (outermost) critical section.
• $T_2$ and $T_1$ cannot preempt $T_3$ in the middle of its critical section.

• Priority inversion upon release during $[3, 6)$, but $T_1$ preempts at end of critical section → priority inversion bounded by CS length.
A Resource Model for the Sporadic Tasks

To derive **pi-blocking bounds**, we need to model shared resources.

- \( n_r \) ... number of shared resources
- \( \ell_1, \ell_2, \ldots, \ell_{n_r} \) ... shared resources
- \( N_{i,q} \) ... per job of \( T_i \), the maximum number of requests for \( \ell_q \) (or critical sections pertaining to \( \ell_q \))
- \( L_{i,q} \) ... maximum request duration (or **critical section length**) of \( T_i \) w.r.t. \( \ell_q \).

**Assumption:** jobs do not **self-suspend** (= sleep)—they are ready until completed.
NPS Blocking Analysis — Under FP Scheduling

• Pi-blocking occurs when a newly released higher-priority task cannot preempt a resource-holding lower-priority task.

• As soon as the non-preemptive section ends, the delayed preemption takes place.
  $\rightarrow b_i \approx \text{maximum outermost lower-priority critical section length.}$

  $$b_i = \max\{L_{k,q} \mid N_{k,q} \geq 1 \land j > i\}$$

(Assuming FP scheduling, lower priority = larger index.)
NPS Blocking Analysis — Under EDF Scheduling

• Pi-blocking occurs when a newly released higher-priority job $J_{h,x}$ cannot preempt a resource-holding lower-priority job $J_{l,y}$.  

$$\Rightarrow a_{l,y} < a_{h,x}$$

• Under EDF, higher priority = earlier absolute deadline ($d_{h,x} < d_{l,y}$).

$$\Rightarrow a_{h,x} + d_h = d_{h,x} < d_{l,y} = a_{l,y} + d_y \Rightarrow d_h < d_l$$

$$b_i = \max\{ L_{k,q} \mid N_{k,q} \geq 1 \land d_k > d_i \}$$

---

1 Homework: what about jitter?
NPS Benefits & Limitations

• Most simple way to prevent unbounded pi-blocking.
• Can be realized by disabling/reenabling interrupts.
• OS does not need to support suspending / resuming tasks.²

But:

• Turning off interrupts risks large interrupt latency.
• All tasks affected: even independent tasks can be pi-blocked.

² Homework: what happens if tasks self-suspend (e.g., due to I/O) and resume? Effect on blocking analysis?
Problem: what if high-frequency tasks cannot tolerate blocking even due to a single, long non-preemptive section?

The Linux PREEMPT_RT patch targets exactly this problem.
Reducing Latency

The Priority Inheritance Protocol (PIP)
How to avoid **latency** and **unbounded pi-blocking**?

![Diagram showing job release, job completion, job deadline, lock attempt, and critical section with time axes labeled T_1, T_2, T_3, and T_4.](image_url)

© 2014 B. Brandenburg (MPI-SWS)
Reducing Latency — Priority Inheritance Protocol (PIP)

**Observation**: we only need to expedite progress of low-priority, resource-holding jobs if they actually block the highest-priority job.

**Solution**:

- preempt resource-holding jobs as needed, but also
- let blocking jobs *inherit priorities* of blocked jobs to raise the *effective priority* of lock holders.
Priority Inheritance Definition\textsuperscript{SRL90 Raj91}

1. Resource-holding jobs remain \textbf{fully preemptive}.

2. The \textit{effective priority} $\pi_i(t)$ of a task $T_i$ at time $t$ is the maximum of its own base priority and the \textit{effective} priority of any job blocked on a lock held by $T_i$ (if any).

$$\pi_i(t) = \min(\{i\} \cup \{\pi_k(t) \mid T_k \text{ is blocked by } T_i\})$$

$\rightarrow$ Priority inheritance takes effect \textit{transitively}.


PIP Example Schedule 1: Decreased Latency for Independent High-Priority Jobs

- Jobs of the high-priority, latency-sensitive task $T_1$ can preempt lower-priority resource-holding jobs as needed.
PIP Example Schedule 2: Bounded PI-Blocking

- Due to priority inheritance, the “middle-priority” task $T_2$ cannot preempt the nominally lower-priority task $T_3$ while it blocks $T_1$.

- Priority inheritance prevents unbounded pi-blocking.
PIP Blocking Analysis

More complicated than any of the other uniprocessor protocols.

• *Direct* pi-blocking: requested lock unavailable.

• *Transitive* pi-blocking: $T_a$ blocked on $T_b$, which is blocked on $T_c$, which...

• *Preemption* pi-blocking: $T_i$ not scheduled because lower-priority job $T_l$ ($l > i$) currently inherits from higher-priority task $T_h$ ($h < i$).

Exact bound can be computed via dynamic programming.\textsuperscript{Liu00}

PIP Discussion

• **Non-optimal pi-blocking bound**: much larger than under NPS.
• Widely used in practice: POSIX’s PTHREAD_PRI0_INHERIT.
• **No latency penalty** and prevents unbounded pi-blocking.
• In practice, the kernel PIP implementation is often *complicated*.
  → adding a new waiter is typically easy (or easier)
  → but waiter removal (e.g., signal delivery, timeout, ...) is tricky
  → must enable preemptions while “walking” inheritance chain

*Can we do better?*
The PIP allows deadlock if lock acquisitions are not well-ordered.

Here, $T_3$ acquires $l_3$ first and then blocks on $l_2$, while $T_2$ first acquires $l_2$ and then blocks on $l_3$: a $l_2 \rightarrow l_3 \rightarrow l_2$ cycle.
Avoiding Deadlock

The Priority Ceiling Protocol (PCP)
Avoiding Deadlock — Priority Ceiling Protocol (PCP)

• The PIP is a **reactive** locking protocol—it only kicks in when resource contention already exists.

• **Key PCP insight:** it’s better to *prevent* problematic scenarios rather than *resolve* them.

• The PCP is an **anticipatory** locking protocol that exploits knowledge of resource needs at *design time* to avoid excessive blocking at *runtime*.
  → *This knowledge is anyway required to derive static blocking bounds.*
Key Concept 1: Priority Ceilings

Definition **priority ceiling** $\Pi(\ell_q)$ of a shared resource $\ell_q$:

$$\Pi(\ell_q) = \text{priority of the highest-priority task that ever accesses } \ell_q.$$ 

→ assuming FP scheduling with **lower index = higher priority**

$$\Pi(\ell_q) = \min\{i \mid T_i \in \tau \land N_{i,q} > 0\}$$
Key Concept 2: Current System Ceiling

Definition **system ceiling** $\hat{\Pi}(t)$: the highest priority ceiling of all resources locked at time $t$, or $\Omega = n + 1$ if none are locked.

→ assuming FP scheduling with *lower index = higher priority*

$$\hat{\Pi}(t) = \min\left(\{\Omega\} \cup \{\Pi(\ell_q) \mid \ell_q \text{ is locked at time } t\}\right)$$
Priority Ceiling Protocol Definition\textsuperscript{SRL90 Raj91}

1. A task $T_i$ may lock a resource $\ell_q$ at time $t$ \textbf{if and only if}

   \( \rightarrow \pi_i(t) < \hat{\Pi}(t) \), i.e., $T_i$’s effective priority exceeds the system ceiling $\hat{\Pi}(t)$, \textbf{or}
   
   \( \rightarrow T_i \) last raised $\hat{\Pi}(t)$, i.e., $T_i$ “owns” the system ceiling $\hat{\Pi}(t)$.

2. A task blocked due to Rule 1 suspends until $\hat{\Pi}(t)$ is lowered again.

3. The task defining (“owning”) the system ceiling benefits from \textbf{priority inheritance}.

   \( \rightarrow \) Inherits the highest priority of any task blocked due to Rule 1.


At $t = 3$, $T_1$ exceeds $\hat{\Pi}(t)$. At $t = 6$, $T_2$ is blocked by Rule 1.
The PCP Avoids Deadlock (1/3)

Why?
The PCP Avoids Deadlock (2/3)

Suppose two locks \( l_1, l_2 \) are involved in deadlock.

Suppose \( T_a \) holds \( l_1 \) and \( T_b \) holds \( l_2 \).
\[
\Pi(l_1) \leq \min(a, b) \quad \text{and} \quad \Pi(l_2) \leq \min(a, b)
\]
\[
\hat{\Pi}(t) \leq \min(a, b) \quad \text{while either} \ l_1 \ \text{or} \ l_2 \ \text{are being held.}
\]

The task that locked its resource second violated Rule 1!
The PCP Avoids Deadlock (3/3)
The PCP ensures that a job is blocked by at most one *outermost* critical section.

Why?
PCP Blocking Analysis (2/3)

Suppose that a job $J_i$ incurs pi-blocking due to two outermost critical sections related to locks $l_1$ and $l_2$.

$\rightarrow \Pi(l_1) \leq i$ and $\Pi(l_2) \leq i$ \(^3\)

$\rightarrow \hat{\Pi}(t) \leq i$ while either $l_1$ or $l_2$ are being held.

Since tasks are sequential, and since both blocking critical sections are outermost, $l_1$ and $l_2$ are held by two different tasks $T_a, T_b$ when $J_i$ starts to execute. \(^4\)

The task that locked its resource second violated Rule 1!

\(^3\) Why? Does $T_i$ necessarily attempt to lock $l_1$ or $l_2$?

\(^4\) What happens if jobs may self-suspend (i.e., due to I/O)?
PCP Blocking Analysis (3/3)

The PCP ensures that a job is blocked by at most one outermost critical section.

Only critical sections of lower-priority tasks pertaining to locks with equal-or-higher priority ceilings can pi-block $T_i$.

$$b_i = \max\{L_{k,q} \mid N_{k,q} \geq 1 \land \Pi(l_q) \leq i \land i < k\}$$
PCP Discussion

If \( \max \{ L_{i,q} \} \) is considered to be (bounded by) a constant, then the PCP ensures \( O(1) \) maximum pi-blocking—this is obviously optimal! → NPS also provide \( O(1) \) maximum pi-blocking.

The PCP is not greedy—it can deny access to (seemingly) available resources. This is necessary to avoid deadlock.

NPS & PIP can be applied without any preparation; the PCP fundamentally requires ceilings to be set correctly.
Detour: **PCP in Linux** or other UNIX-like kernels?

The Linux kernel supports the PIP, not the PCP, for internal locks. Why?
Detour: **PCP in Linux** or other UNIX-like kernels?

The Linux kernel supports the PIP, not the PCP, for internal locks. Why?

1. There are thousands of locks in the kernel, many dynamically created. Who sets the appropriate priority ceilings?

2. The priority ceiling for most locks depend on the priority of the tasks that invoke a given system call—this is not known until runtime.

3. Virtually any path in the kernel *could* be executed by the highest-priority task, so **static ceilings would all collapse** to the maximum priority.
The PCP in Practice

The PCP is widely used in practice—in virtually any modern car!

• The AUTOSAR standard, and prior to it the OSEK/VDX standard, mandates a variant of the PCP for (core-local) resource management. Why are priority ceilings not a problem here?

• API: GetResource() and ReleaseResource()
Why a variant? How can we improve the PCP?

The PCP provides an **optimal maximum pi-blocking bound** ...

$\rightarrow$ What is the main source of *runtime* overhead?
Avoiding Context Switches

The Stack Resource Policy (SRP)
What’s Wrong with Context Switches?

• Each contended critical section causes *two additional context switches*.  
  → Regular preemption: LO-HI-LO  
  → With critical section: LO-HI-LO-HI-LO

• Context switches can be a main source of overhead: OS context switching code, possibly TLB flushes, *loss of cache affinity*.

• Switching *back* to a preempted job implies the need for *separate function call stacks* for each task → *memory* overhead. *Why?*

→ Avoiding LO-HI-LO-HI-LO context switches avoids runtime overheads and enables single-stack designs.
The Stack Resource Policy (SRP)

Observation: if a preemtping job requires a locked resource, then a LO-HI-LO-HI-LO context switch sequence becomes inevitable only if the preempting job is allowed to start executing.

Solution: do not allow jobs to commence execution until all (possibly) required resources are available.
→ No more LO-HI-LO-HI-LO context switch sequences...
SRP Definition\textsuperscript{Ba91}

1. Define priority ceilings and system ceilings as under the PCP.

2. When a job is released, it may not commence execution until its (base) priority exceeds the system ceiling (or \textit{preemption threshold}).

3. Whenever a job requires a resource, it gains access immediately.

SRP Example Schedule

- Preemption of resource-holding jobs is still possible ($T_1$ at $t = 2$).

© 2014 B. Brandenburg (MPI-SWS)
SRP Blocking Analysis

The bound on worst-case pi-blocking under the SRP is identical to the PCP’s bound.

\[ b_i = \max\{L_{k,q} \mid N_{k,q} \geq 1 \land \Pi(l_q) \leq i \land i < k\} \]

• The actual pi-blocking differs under the SRP and the PCP.
  → The SRP moves pi-blocking to an earlier point in time.
  → On average, the PCP may yield slightly less pi-blocking. (Why?)
An “Implicit” SRP Implementation

**OSEK-PCP**: immediately raise the priority of lock-holding tasks.

- "If a task requires a resource, and its current priority is lower than the ceiling priority of the resource, the priority of the task is raised to the ceiling priority of the resource."

- "If the task releases the resource, the priority of this task is reset to the priority which was dynamically assigned before requiring that resource."

- Newly arrived tasks with priority not exceeding the priority ceiling cannot preempt the lock holder—this is the SRP!
SRP Discussion

• With implicit ceiling tracking via “current priority”, the SRP is easy to implement.

• It can be implemented on top of commodity interrupt controllers.

• Low runtime overheads.

• Enables stack sharing.

• The SRP (and the PCP) can be extended to multi-unit resources and can thus also be used as reader-writer locks.
The “implicit” SRP/PCP is known under many names...

- MESA-style semaphores
- OSEK-PCP
- Immediate PCP (IPCP)
- PTHREAD_PRI0_PROTECT (POSIX)
- Ceiling Locking (ADA)
- Priority Ceiling Emulation (Real-Time Java)
- “Highest Locker” Protocol
Immediate Priority Ceilings in MESA

“Unless care is taken, the ordering implied by the assignment of priorities can be subverted by monitors. [...] A simple way to avoid this situation is to associate with each monitor the priority of the highest priority process which ever enters that monitor. Then whenever a process enters a monitor, its priority is temporarily increased to the monitor’s priority.”

— Butler W. Lampson & David D. Redell, 1980

LR80 B. Lampson and D. Redell (1980). Experience with Processes and Monitors in Mesa. CACM 23(2).
Accounting for PI-Blocking

RTA in the Presence of Locking Delays
Effects of PI-Blocking

• Pi-blocking causes “gaps” in the level-$i$ busy window. → during times of pi-blocking, a lower-priority task is scheduled.

• Logically, however, the level-$i$ busy window continues: the level-$i$ processor demand has not yet been met.

• During busy window, lower-priority tasks cannot acquire resources.

• The time during which a job $J_i$ can be delayed by newly arriving higher-priority work is increased by up to $b_i$ time units. → We need to add $b_i$ to the response-time recurrence.
RTA for tasks with arbitrary deadlines

The response time of jobs of task $T_i$ is bounded by

$$R_i = j_i + \max\{w_i(q) \mid q = 1, 2, \ldots\},$$

where $w_i(q)$ is the smallest positive solution to the recurrence

$$w_i(q) = q \cdot e_i + b_i + \left( \sum_{T_h \in hp_i} \eta^+ (w_i(q) + \delta^-_i(q)) \cdot e_h \right) - \delta^-_i(q).$$
Optimistic Synchronization

Alternative to Mutual Exclusion

---

6 Based in part on slides by Jim Anderson (UNC Chapel Hill)
Nonblocking Algorithms

fully preemptive, no priority inversion, no blocking

- **Wait-free data structures** provide **guaranteed progress**: each operation succeeds after (at most) a fixed number of instructions.
  → Real-time: Conceptually, can be treated as “straight-line code.”

- **Lock-free data structures** provide **eventual progress**: if two or more operations conflict, at least one progresses.
  → Need to (individually?) retry while operation is interfered with.
  → Real-time: Need to derive **retry bounds**!
Lock-Free Example: Linked List

/* compare-and-swap on two locations */
extern bool CAS2(void *ptr1, void *ptr2,
    void* expected1, void* expected2,
    void* new1, void* new2);

struct List { int val; struct List *next; };

/* global list tail pointer */
struct List* tail; /* non-NULL */

void enqueue_at_tail(int val) {
    struct List* l = malloc(sizeof(*elem));
    struct List* observed_tail;
    l->val = val; l->next = NULL;
    while (1) { /* retry loop */
        observed_tail = tail;
        if (CAS2(&tail, &tail->next,
            observed_tail, NULL,
            l, l))
            break;
    }
}
Observation: a lock-free operation fails only if it is preempted.

Approach: reason about *preempting* task, not *preempted* task

How many times is a given retry loop preempted?
→ **Difficult to say.**

How many retry-loop failures per preempting job?
→ **At most one.**

**Observation:** A constant delay per higher-priority job.
→ We can **charge** this delay as part of the higher-priority WCET.
Illustration: Retry-Loop Delay $\approx$ Higher-Priority Demand

Failed retry-loop  Successful retry-loop
Uniprocessor Analysis with Retry Loops (1/2)

Notation: $e_i^{retry}$ ... the maximum WCET of any retry loop in $T_i$.

Without retry loops: if $T_i$ is preempted by a single job of $T_h$, then $T_i$ is delayed by at most $e_h$ time units.

With retry loops: if $T_i$ is preempted by a single job of $T_h$, then $T_i$ is delayed by at most $e_h + \max\{e_x^{retry} \mid i \geq x > h\}$ time units.

Assuming FP scheduling, with lower index = higher priority. What about EDF?
Uniprocessor Analysis with Retry Loops (2/2)

The response time of job in task $T_i$ is bounded by $R_i = j_i + \max\{w_i(q) \mid q = 1, 2, \ldots\}$, where $w_i(q)$ is the smallest positive solution to the recurrence

$$w_i(q) = q \cdot e_i + \left( \sum_{T_h \in hp_i} \eta_h^+ (w_i(q) + \delta_i^-(q)) \cdot e'_h \right) - \delta_i^-(q)$$

and $e'_h = e_h + \max\{e_x^{\text{retry}} \mid i \geq x > h\}$. 
Nonblocking Algorithms: Major Limitations

• Optimistic synchronization only works for data structures, not shared resources in general
  → what about synchronizing access to I/O ports?

• memory overhead (extra copies), need dynamic memory management
  → many lock-free algorithms work easiest with garbage collection
  → fine UNIX-like RTOSs (such as QNX, Linux, etc.), but for OSEK class

• for general data structures (universal constructions), copying overhead can be prohibitive
  → best used with specialized algorithms (e.g., linked lists, etc.)
Nonblocking Synchronization on Multiprocessors

1. **Lock-free** algorithms: with state of the art, very pessimistic retry bounds. (Why do you think this is the case?)

2. **Wait-free** algorithms: all of a sudden a lot more attractive.

*Wait-free doesn't get any cheaper on multiprocessors than it is on uniprocessors, but all other options get much worse...*
Locking in Multiprocessor Systems

A Brief Introduction
Two Fundamental Ways to Implement "Blocking on a Lock" on a Multiprocessor

1. As in uniprocessor systems, with semaphores.
   → allow other tasks to use the processor while waiting
   → suspension-based locking protocols

2. By busy-waiting, with spin locks.
   → do not yield the processor, instead poll the lock
   → spin-based locking protocols
Why is real-time locking fundamentally more difficult on multiprocessors?

- On **uniprocessors**, lower-priority tasks *cannot* acquire any locks while a job of $T_i$ is incomplete. ➔ *No new contention!*

- On **multiprocessors**, remote tasks can contest shared resources *at any time and any number of times*. ➔ *"Never-ending" contention!*

- On **multiprocessors**, waiting for a *remotely held lock* can allow *local lower-priority tasks* to create new contention! ➔ *Even more...*
What about the uniprocessor approaches?

- **Priority ceilings** across all processors: not useful.
  → *Restricted parallelism* essentially equivalent to a **single lock**.

- **Priority ceiling** on each processor: not sufficient.
  → Doesn't help with controlling **remote blocking**.

- **Priority inheritance** under **partitioned** scheduling: ineffective.
  → Also ineffective across cluster boundaries under clustered scheduling.

- **Priority inheritance** under **global** scheduling: **works**.
  → But much more difficult to analyze due to remote blocking.
Example: Priority Inheritance is Ineffective Across Partitions

- $T_3$ is delayed for entire WCET of task $T_1$, which is unbounded in general (and typically much longer than a critical section).
- $T_3$ is pi-blocked: no local higher-priority task to explain delay.
Comparison: Uniprocessor Unbounded PI vs. PIP Across Partitions

© 2014 B. Brandenburg (MPI-SWS)
Classic Progress Mechanism for Partitioned Scheduling: **Priority Boosting**

Example shows that progress mechanism to expedite lock release is essential, but *priority inheritance is ineffective*.

Root cause: priorities are **incomparable** across partitions.

→ *lock holder is not scheduled* while pi-blocked job waits.

**Priority Boosting**: simply raise the priority of lock-holding jobs.

→ lock-holder priorities higher than that of any non-lock-holding job

→ newly released job doesn't hold any resources yet → **CS never preempted**!
Priority Boosting Example

- Newly arrived $T_1$ (not priority-boosted) cannot preempt the priority-boosted $T_2$ in its critical section.
- The lock is released quickly and hence $T_3$ is delayed only briefly.
PI-Blocking due to Priority Boosting
≈ Latency of Non-Preemptive Sections!

Priority boosted job effectively non-preemptive—just like NPS!
Partitioning + Locks + **Low Latency**!

Entire uniprocessor real-time locking literature: *how to avoid NPS!*
→ Goal: minimize *scheduling latency!*

Why should we just accept **NPS-like behavior** on multiprocessors?
→ Few alternatives... but there exists one if migration is *feasible, but discouraged.*

**Migratory Priority Inheritance:**[^13] migrate lock holder to core of pi-blocked job
→ inherit priority as before ("right to execute at a certain priority")
→ but also *inherit processor affinity* ("right to execute on a specific core")

Migratory Priority Inheritance Example

Zero pi-blocking (= latency) for $J_1$, bounded pi-blocking for $J_3$. 
Observation: Migrating the critical section is the **only** solution in this scenario

Just **regular priority inheritance** leads to deadline miss...
Discussion: Semaphores in Multiprocessor Real-Time Systems

• Analysis more **difficult** than on uniprocessors
  → remote blocking, repeated lower-priority blocking, deadlock...

• **Progress mechanisms**: both essential to have and a problematic source of pi-blocking

• Many **open problems**... (come talk to me if you are interested!)
  → questions of optimality, questions of practicality, "tuning knobs,...
  → **nesting** of critical sections not fully supported (not all protocols)
Spin Locks in Multiprocessor Real-Time Systems
Spinning vs. Suspending

Spinning **wastes cycles.** ➔ *Aren't suspension-based protocols more efficient? No!*

1. While a job spins (= delays lower-priority tasks), lower-priority tasks cannot try to acquire locks. ➔ less **local blocking.**

2. Suspensions incur non-negligible **overheads.**

3. **Shorter queues** ➔ worst case assumption is less pessimistic.

---


Semaphores in Theory

Theory assumption: **zero overheads.**
In the diagram, the critical section is shown to be locked during the execution of a task. The diagram illustrates the sequence of events:

1. **syscall entry**: The syscall entry point is reached.
2. **lock() backend**: A lock is requested.
3. **schedule()**: The task is suspended.
4. **lock()**: The lock is acquired.
5. **IPI latency**: An inter-processor interrupt (IPI) latency is shown.
6. **schedule()**: The context switch occurs.
7. **lock()**: Another lock is acquired.
8. **context switch**: The lock holder is resumed.
9. **unlock()**: The lock is released.
10. **syscall exit**: The syscall exit is reached.
11. **resume next lock holder**: The next lock holder is resumed.
12. **send IPI**: An IPI is sent.
13. **syscall exit**: The syscall exit is reached.

The diagram also shows the reestablishment of cache affinity after the critical section.

**Definitions**:
- **IPI** = inter-processor interrupt
- **lock** = in-kernel spinlock acquisition

© 2014 B. Brandenburg (MPI-SWS)
Semaphores: Theory vs. Practice

- Increased CS length
- Increased execution cost
- Increased CS latency

© 2014 B. Brandenburg (MPI-SWS)
Simple **Ticket Locks** (= FIFO Spin Locks)

```c
struct ticket_lock {
    volatile unsigned int arrival_counter = 0;
    volatile unsigned int now_serving = 0;
}

void lock(struct ticket_lock* l) {
    disable_preemptions(); /* IRQ off or set a flag in memory */
    var ticket := atomic_fetch_and_inc(&l->arrival_counter);
    while (ticket != l->now_serving)
        /* do nothing */;
}

void unlock(struct ticket_lock* l) {
    l->now_serving++;
    reenable_preemptions(); /* IRQ on or clear flag in memory */
}
```
Spin Lock Advantages

- no system calls or ready queue updates

- just a couple of instructions: **very low overheads**
  → efficient if critical sections are "short"
  → but long critical sections are problematic in general

- cache state is not disturbed while waiting
  → better cache affinity

But: need to analyze **WCET increase** due to delay loops!
The Multiprocessor Stack Resource Policy\textsuperscript{GLD01} (MSRP) = \textit{Partitioned Scheduling + SRP + Non-Preemptive FIFO Locks}

- For \textbf{local resources}, simply use the SRP.
  \(\rightarrow\) local resource: shared only among tasks assigned to same core

- For \textbf{global resources}, use non-preemptive FIFO locks.
  \(\rightarrow\) global resource: shared among tasks on different cores

- Two types of blocking: \textit{spinning} and \textit{pi-blocking}...

Types of Blocking: Spin- vs PI-Blocking

- **pi-blocking**: as before, when a job is prevented from being scheduled by a lower-priority job on its core.

- **s-blocking**: when a job spins due to a job on a remote core.
Classic MSRP analysis — Per-Request Bound

\( \tau_k \) ... the set of tasks assigned to processor \( k \)

\( P_i \) ... the processor to which \( T_i \) is assigned

**Non-preemptive FIFO progress property:** w.r.t. each request, a task is blocked by at most one critical section on each other core.

\( s_{i,q} \) ... max. s-blocking when accessing resource \( \ell_q \) from processor \( P_i \):

\[
s_{i,q} \leq \sum_{\substack{k=1 \\
k \neq P_i}}^m \max \{L_{x,q} \mid T_x \in \tau_k \land N_{x,q} \geq 1\}
\]
Classic MSRP analysis — Overall Spin Bound$^{\text{GLD01}}$

Trivial upper bound: sum per-request bounds across all requests.

\[ s_i \ldots \text{max. s-blocking incurred by any job of } T_i \]

\[ s_i \leq \sum_{q=1}^{n_r} N_{i,q} \cdot s_{i,q} \]

Still Need to Account for PI-Blocking

\[ b_i^{\text{loc}} \] ... max pi-blocking due to \textbf{local} resources (controlled by SRP)

\[ b_i^{\text{np}} \] ... max pi-blocking due to \textbf{non-preemptive spinning}
Classic MSRP Analysis — PI-Blocking Bound

\[ b^{loc}_i = \text{regular SRP analysis w.r.t. local resources (omitted here)} \]
\[ b^{np}_i = \text{maximum non-preemptive section length (→ NPS analysis)} \]

\[ b^{bp}_i \leq \max\{L_{l,q} + s_{l,q} \mid T_l \in \tau_{P_i} \land l > i \land N_{l,q} \geq 1\} \]

where \( \tau_{P_i} \) is the set of tasks local to processor \( P_i \).

---

Classic MSRP analysis$^{GLD01} - RTA$

The response time of jobs of task $T_i$ is bounded by
\[ R_i = j_i + \max\{w_i(q) \mid q = 1, 2, \ldots\}, \]
where $w_i(q)$ is the smallest positive solution to the recurrence

\[ w_i(q) = q \cdot e_i' + \max(b_i^{loc}, b_i^{np}) + \left( \sum_{T_h \in h_{p_i}} \eta_h^+ (w_i(q) + \delta_i^-(q)) \cdot e_h' \right) - \delta_i^-(q). \]

where $e_i' \leq e_i + s_i$ is a bound on the effective (= spin-inflated) WCET.

---

Sources of Pessimism

1. few remote vs. many local critical sections
2. long, but rare remote critical sections
3. disregarding response times when bounding contention
   • And several other sources of over-counting...
Source of Pessimism in MSRP Analysis #1: Many vs. Few Critical Sections

A pathological task set with exaggerated parameters…

<table>
<thead>
<tr>
<th>Core</th>
<th>$e_i$</th>
<th>$p_i$</th>
<th>$d_i$</th>
<th>$N_{i,q}$</th>
<th>$L_{i,q}$</th>
<th>Classic MSRP</th>
<th>Actual total Blocking</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>1</td>
<td>200</td>
<td>1000</td>
<td>250</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$T_2$</td>
<td>2</td>
<td>200</td>
<td>1000</td>
<td>250</td>
<td>100</td>
<td>100</td>
<td>1</td>
</tr>
</tbody>
</table>

FIFO spin locks: can block at most once, but could block any of the CS in $T_2$. 
**Over-counting contention**
A single blocking critical section is counted $N_{i,q}$ times.

<table>
<thead>
<tr>
<th>Core</th>
<th>$e_i$</th>
<th>$p_i$</th>
<th>$d_i$</th>
<th>$N_{i,q}$</th>
<th>$L_{i,q}$</th>
<th>Classic MSRP</th>
<th>Actual total Blocking</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>1</td>
<td>200</td>
<td>1000</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$T_2$</td>
<td>2</td>
<td>200</td>
<td>1000</td>
<td>100</td>
<td>1</td>
<td>100</td>
<td>1</td>
</tr>
</tbody>
</table>

**FIFO spin locks: can block at most once, but could block any of the CS in $T_2$.**
Source of Pessimism in MSRP Analysis #2: Rare Long Critical Sections

**Diagram:**
- **T₁**: NCS → CS 1 → NCS → CS 2 → NCS
- **T₂**: NCS → CS 1 → NCS → ... → NCS → CS 100 → NCS
- **T₃**: NCS → CS 1 [long, but rare] → NCS

**Note:**
- NCS: Non-Critical Section
- CS: Critical Section

**Cannot block both critical sections of T₁**

<table>
<thead>
<tr>
<th></th>
<th>Core</th>
<th>(e_i)</th>
<th>(p_i)</th>
<th>(d_i)</th>
<th>(N_{i,q})</th>
<th>(L_{i,q})</th>
<th>Classic MSRP</th>
<th>Actual total spin blocking</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>T₁</strong></td>
<td>1</td>
<td>200</td>
<td>1000</td>
<td>250</td>
<td>2</td>
<td>1</td>
<td>100</td>
<td>51</td>
</tr>
<tr>
<td><strong>T₂</strong></td>
<td>2</td>
<td>200</td>
<td>1000</td>
<td>250</td>
<td>100</td>
<td>1</td>
<td>100</td>
<td>2</td>
</tr>
<tr>
<td><strong>T₃</strong></td>
<td>2</td>
<td>200</td>
<td>10,000</td>
<td>5000</td>
<td>1</td>
<td>50</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
The long critical section of $T_3$ is executed only rarely, which the classic MSRP analysis cannot exploit.

<table>
<thead>
<tr>
<th>Core</th>
<th>$e_i$</th>
<th>$p_i$</th>
<th>$d_i$</th>
<th>$N_{i,q}$</th>
<th>$L_{i,q}$</th>
<th>Classic MSRP</th>
<th>Actual total spin blocking</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>1</td>
<td>200</td>
<td>1000</td>
<td>250</td>
<td>2</td>
<td>100</td>
<td>51</td>
</tr>
<tr>
<td>$T_2$</td>
<td>2</td>
<td>200</td>
<td>1000</td>
<td>250</td>
<td>100</td>
<td>100</td>
<td>2</td>
</tr>
<tr>
<td>$T_3$</td>
<td>2</td>
<td>200</td>
<td>10000</td>
<td>5000</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Source of Pessimism in MSRP Analysis #3: Disregarding Response Times

<table>
<thead>
<tr>
<th>Core</th>
<th>$e_i$</th>
<th>$p_i$</th>
<th>$d_i$</th>
<th>$N_{i,q}$</th>
<th>$L_{i,q}$</th>
<th>Classic MSRP</th>
<th>Actual total blocking</th>
<th>Resource usage during interval of length 1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>1</td>
<td>15</td>
<td>100</td>
<td>20</td>
<td>1</td>
<td>100</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>$T_2$</td>
<td>2</td>
<td>20</td>
<td>1000</td>
<td>50</td>
<td>10</td>
<td>100</td>
<td>100</td>
<td>10</td>
</tr>
</tbody>
</table>

**minimum separation**: $p_1 - d_1 = 80$

**maximum response time**: $d_2 = 50$

issued when $T_2$ is no longer pending
**Disregarding response times**

If the task set is **schedulable**, any job of \( T_2 \) can encounter at **most one job** of \( T_1 \).

<table>
<thead>
<tr>
<th>Core</th>
<th>( e_i )</th>
<th>( p_i )</th>
<th>( d_i )</th>
<th>( N_{i,q} )</th>
<th>( L_{i,q} )</th>
<th>Classic MSRP</th>
<th>Actual total blocking</th>
<th>Resource usage during interval of length 1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T_1 )</td>
<td>1</td>
<td>15</td>
<td>100</td>
<td>20</td>
<td>1</td>
<td>10</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( T_2 )</td>
<td>2</td>
<td>20</td>
<td>1000</td>
<td>50</td>
<td>10</td>
<td>1</td>
<td>100</td>
<td>10</td>
</tr>
</tbody>
</table>

*minimum separation: \( p_1 - d_1 = 80 \)*

*maximum response time \( \leq d_2 = 50 \)*

issued when \( T_2 \) is no longer pending
LP-Based Analysis of Spin Locks\textsuperscript{WB13}

Basic idea: blocking analysis is an optimization problem. 
→ maximize blocking subject to protocol and workload constraints.

Turns out most (all?) relevant constraints are linear. 
→ express blocking analysis as linear programming problem.

Helpful observation: it's easier to rule out impossible schedules rather than to describe set of possible schedules.
→ $\max ("not\ shown\ to\ be\ impossible") \geq \max ("possible\ schedules")$