

Real-Time Inter-Processor Synchronization Algorithms

– for predictability and scalability –

July 10, 2001

Hiroaki Takada

Dept. of Information and Computer Sciences

Toyohashi Univ. of Technology

hiro@ertl.ics.tut.ac.jp

<http://www.ertl.ics.tut.ac.jp/~hiro/>

Agenda

<http://www.ertl.ics.tut.ac.jp/~hiro/tmp/mpsoc.pdf>

A Brief Introduction to the ITRON Project

RTOS for Shared-Memory Multiprocessors

- ▶ Desired Properties for Scalable RTOS

Problem of Inopportune Preemptions

- ▶ Queueing Spin Locks with Preemption
- ▶ SPEPP Synchronization

Scalability of Nested Spin Locks

Priority Inheritance Spin Locks

RTOS Implementation Issues

- ▶ Solved Problem and Open Problem

! "*RTOS*" and "*real-time kernel*" (or just "*kernel*") are used interchangeably in this presentation.

A Brief Introduction to the ITRON Project

What is the ITRON Project?

<http://www.itron.gr.jp/>

- ▶ a project to standardize RTOS and related specifications for embedded systems
(*esp. small-scale embedded systems*)
- ▶ a joint project of industry and academia
(*not a government project*)
 - core members:*
Fujitsu, Hitachi, Matsushita (Panasonic),
Mitsubishi Electric, NEC, Oki Electric, Toshiba
 - US companies (or its subsidiaries):*
Accelerated Technology Inc. (ATI), Hewlett-Packard,
Metrowerks, Red Hat, U S Software
 - academia:*
Univ. of Tokyo, Toyohashi Univ. of Technology
- ▶ one of the subprojects of the TRON Project

Advantages of the ITRON Real-Time Kernel Specifications

- ▶ compact and low-overhead real-time kernel specifications
 - ▶ *fit in a single chip MCU*
- ▶ easy to understand
- ▶ open specification
 - ▶ *anyone can use the specifications in free*
 - ▶ *complete specification documents on the web site*
- ▶ applicable to wide variety of processors
 - ▶ *from low-cost 8-bit MCU to high-performance 64-bit RISC*
- ▶ widely used for various embedded systems
 - ▶ *used in over 30% of embedded systems in Japan*
- ▶ supported by many companies

Design Concept

- ▶ *loose standardization*

*maximum performance cannot be obtained
with strict standardization*



adaptability & scalability

Design Principles

- ▶ allow for adaptation to hardware, avoiding excessive hardware virtualization
- ▶ allow for adaptation to the application
- ▶ **emphasize software engineer training ease**
- ▶ organize specification series and divide into levels
- ▶ provide a wealth of functions

Functions of μ ITRON4.0 Specification

- ▶ task management
- ▶ task-dependent synchronization
- ▶ task exception management
- ▶ basic synchronization and communication
- ▶ extended synchronization and communication
- ▶ memory pool management
- ▶ time management
- ▶ system state management
- ▶ interrupt management
- ▶ service call management
- ▶ system configuration management
- ▶ *! no I/O handling functions defined*

Number of Service Calls

- ▶ full set
 - service calls: 166
 - static API: 21
- ▶ standard profile
 - service calls: 170
 - static API: 11
- ▶ automotive control profile
 - service calls: 43
 - static API: 8
- ▶ minimum set

Application Status of ITRON-specification OS

- ▶ most widely used OS specification for embedded systems in Japan
- ▶ widely used especially in consumer applications

Typical Applications

Audio/Visual Equipment, Home Appliance

TVs, VCRs, digital cameras, settop boxes, audio components

Personal Information Appliance, Entertainment/Education

PDA's, car navigation systems, electronic musical instruments

PC Peripheral, Office Equipment

printers, scanners, disk drives, CD-ROM drives, copiers, FAX

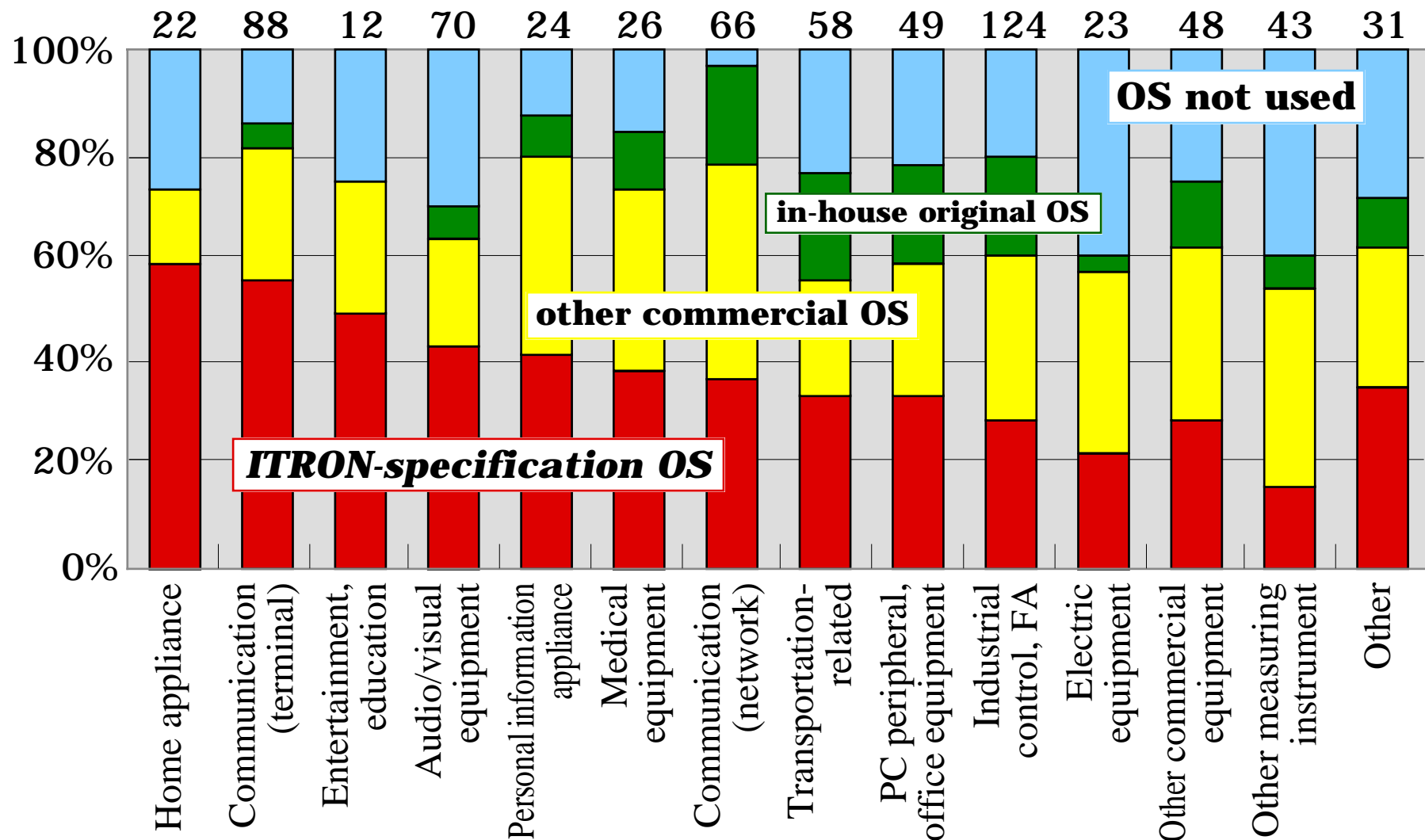
Communication Equipment

ISDN telephones, cellular phones, ATM switches, satellites

Transportation, Industrial Control, and Others

automobiles, plant control, industrial robots, medical equipment

Real-Time Inter-Processor Synchronizations



Embedded OS for each application field

(TRON Association Survey, late 1999 - early 2000, Japan)

RTOS for Shared-Memory Multiprocessors

Target Architecture

- ▶ *function-distributed* shared-memory multiprocessors
more practical for embedded systems

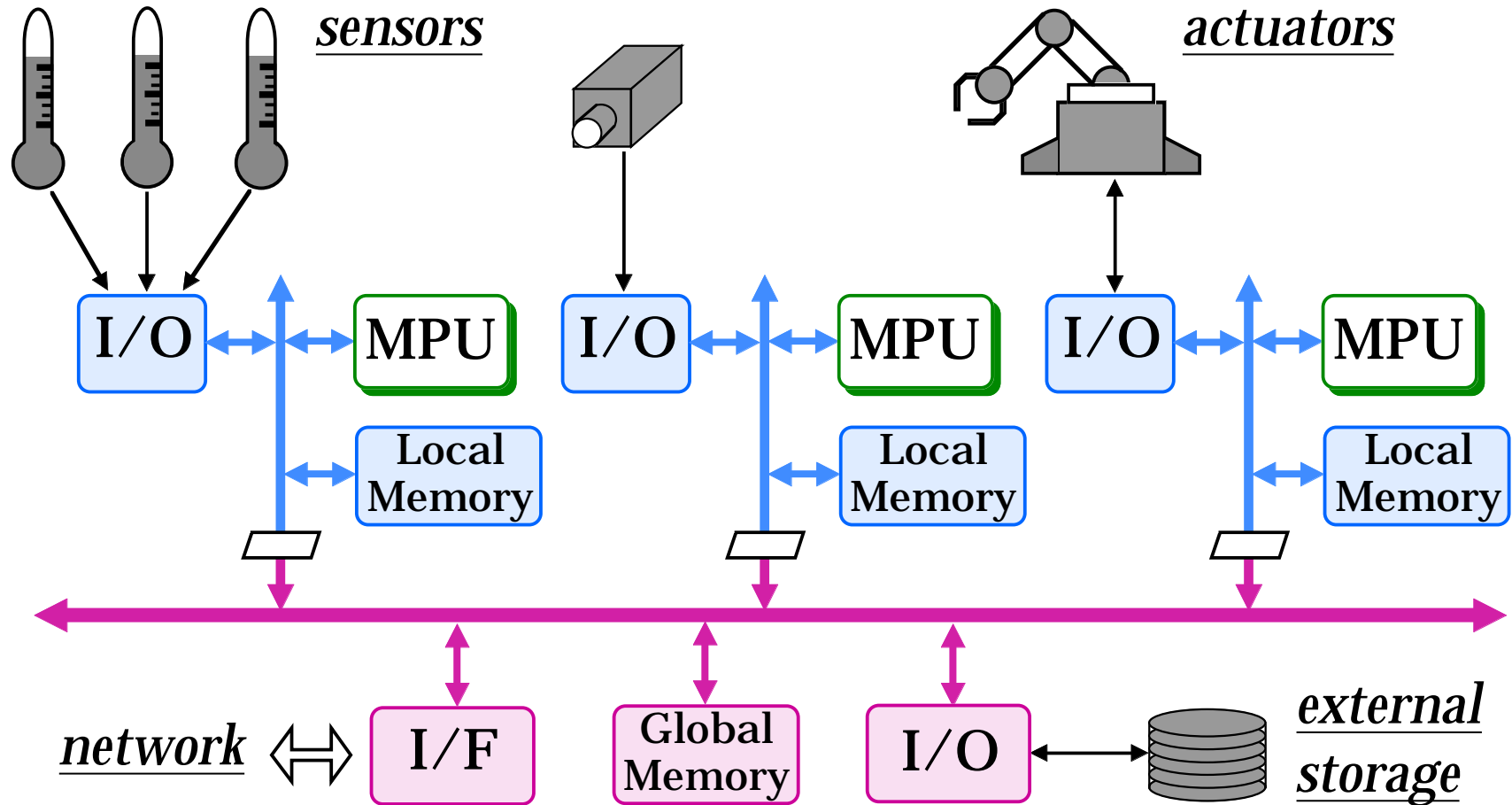
Basic RTOS Model for FDM

- ▶ Each task is bounded to a processor, called a local task of the processor.
 - ▶ Multiple local tasks are executed *preemptively* on each processor.
- *multiprogrammed multiprocessors*

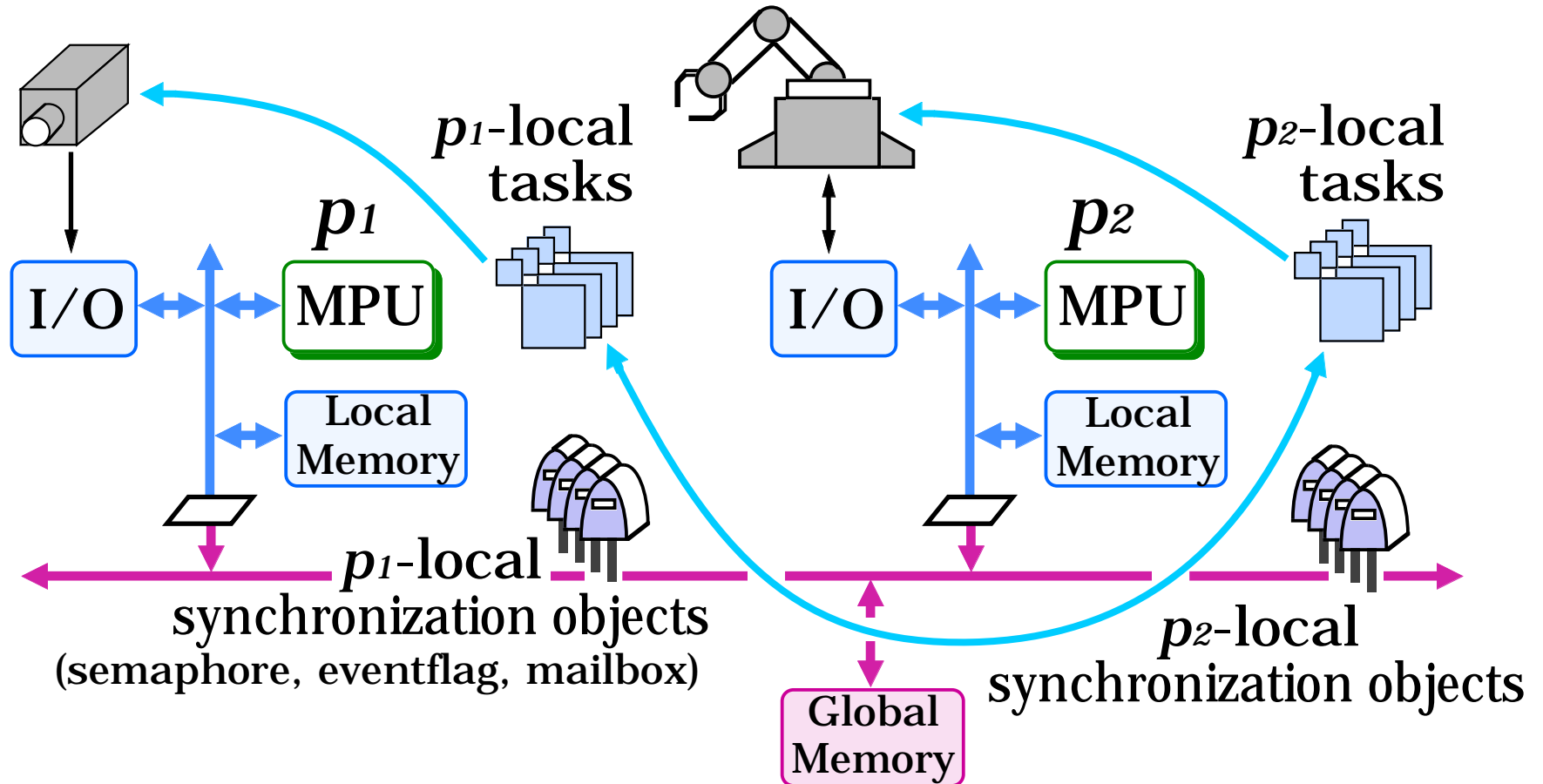
Requirements

- ▶ predictability
- ▶ scalability (of worst-case behavior)

Function-Distributed Shared-Memory Multiprocessor



Basic RTOS Model for FDM



- ▶ Extensions adding private and/or global resources to this model are possible.

Predictability of RTOS

- ▶ Worst-case (maximum) execution time of each RTOS system call should be bounded and known.

Scalability of RTOS (to the number of processors)

- ▶ Changes in *worst-case* timing behavior of the system when *a processor is added* should be minimized.

An Obvious Limitation

- ▶ Worst-case execution times of a job that uses shared resources *exclusively* become at least $O(n)$.
 n : the number of contending processors
- ▶ Various data structures within RTOS are shared and must be accessed *exclusively*. (using *spin locks*)
- ➔ Worst-case execution times of some RTOS systems calls become at least $O(n)$.

An Approach to Ease this Limitation

- ! taking account of the design rule of FDM
- ▶ Many of the tasks can be processed without direct synchronizations with tasks on other processors.
- ➔ It is advantageous that *the worst-case behavior of such tasks is independent of n .*
- ▶ Predictable interrupt response is also important in designing real-time systems.
- ➔ *The worst-case interrupt response time should be independent of n .*



Inter-processor synchronization mechanisms, such as *spin lock algorithms*, are among the most important issues to be addressed.

Desirable Properties for Scalable RTOS

- (A) The maximum execution time of a system call that is to synchronize with tasks on the **same processor** should be $O(1)$.
- (B) The maximum execution time of a system call that is to synchronize tasks on **other processors** should be $O(n)$.
- (C) The maximum **interrupt response** time on each processor should be $O(1)$.
- (D) The **interrupt service overhead** should be $O(1)$.

interrupt service overhead: wasted computation time
by an interrupt service

- ★ These times should be determined ***independently*** of the other processors' activities.

Desirable Properties at a Glance

<i>system call of RTOS</i>		<i>worst-case times</i>	
<i>name</i>	<i>function</i>	<i>local operations</i>	<i>remote operations</i>
:	:	:	:
sus_tsk	suspend task execution (with task switch)	T_{sus_tsk} T_{sus_tsk}'	$n \cdot T_{wait} + T_{sus_tsk}''$ $n \cdot T_{wait} + T_{sus_tsk}'''$
rsm_tsk	resume task execution	T_{rsm_tsk}	$n \cdot T_{wait} + T_{rsm_tsk}''$
	(with task switch)	T_{rsm_tsk}'	$n \cdot T_{wait} + T_{rsm_tsk}'''$
:	:	:	:
sig_sem	signal semaphore	T_{sig_sem}	$n \cdot T_{wait}' + T_{sig_sem}''$
	(with task switch)	T_{sig_sem}'	$n \cdot T_{wait}' + T_{sig_sem}'''$
wai_sem	wait semaphore	T_{wai_sem}	$n \cdot T_{wait}' + T_{wai_sem}''$
	(with task switch)	T_{wai_sem}'	$n \cdot T_{wait}' + T_{wai_sem}'''$
:	:	:	:
<i>interrupt response time</i>		$T_{int_response}$	
<i>interrupt service overhead</i>		$T_{int_overhead}$	

} $O(n)$
} $O(1)$

Assumptions

- ! In this presentation, inter-processor synchronization algorithms are discussed under the following assumptions.

Hardware Support

- ▶ A *universal* atomic operation on a single word of memory is supported.
 - eg) *compare_and_swap* (CAS)
 - load_linked/store_conditional* pair

Used for the Implementation of an RTOS

- ▶ *Task preemptions can be inhibited by disabling interrupt services.*
 - ← Unintentional task preemptions are triggered by interrupt requests.
- ▶ *Interrupt requests can be probed.*

Remark on the Assumption

- ▶ Hardware support for inter-processor synchronization is limited to an atomic operation on a single word of memory.
- ▶ With a SoC, more sophisticated hardware support is possible.

With more sophisticated hardware support,

- ▶ Concrete algorithms cannot be applied and then are not important. So, I will almost omit them in the presentation.
 - ➔ *Refer to the cited papers for concrete algorithms.*
- ▶ But, the problems are common and the synchronization approaches are useful when *predictability* and *scalability* are important.

Reviewing Spin Lock Algorithms

Test&Set Lock

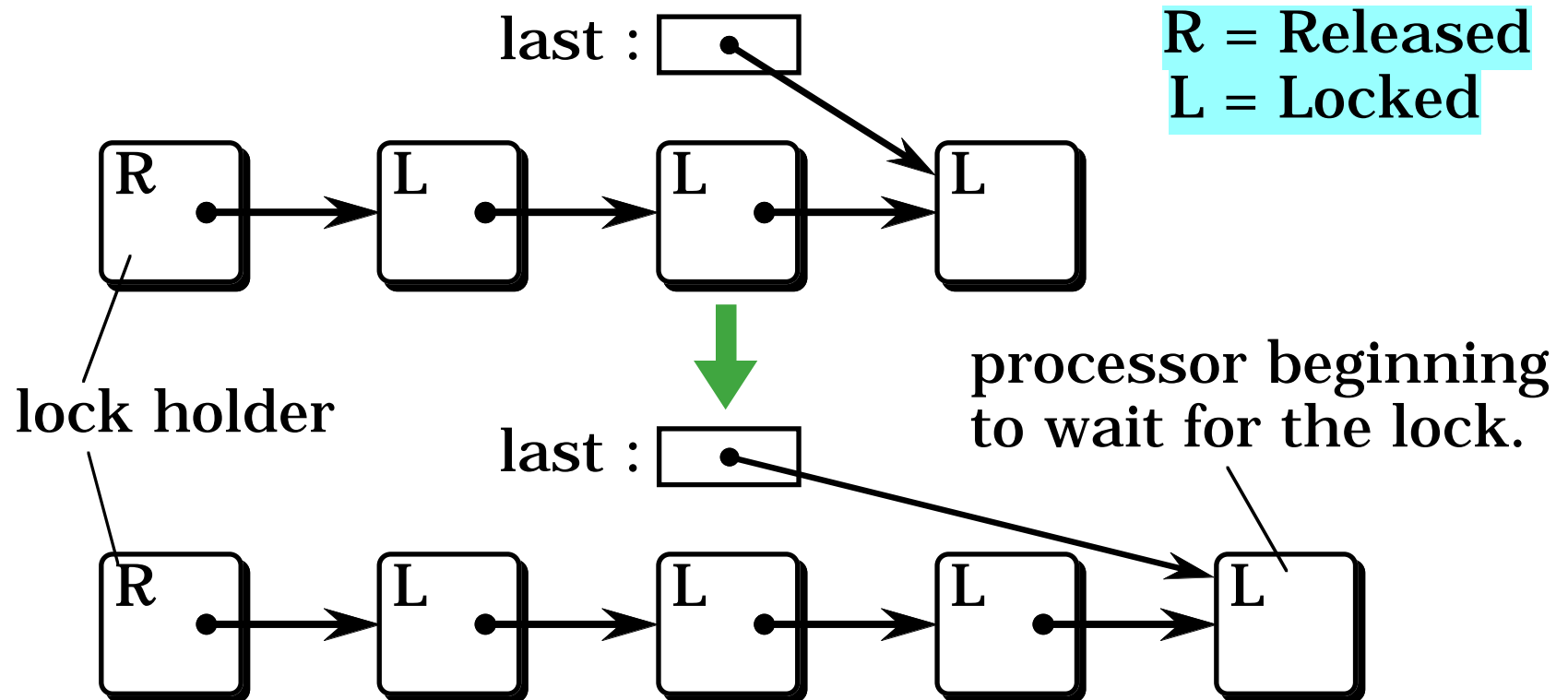
```
while test_and_set(L) = Locked do  
    wait_a_while;  
end;  
// critical section.  
L = Unlocked;
```

- ▶ most simple and popular spin lock algorithm.
- ▶ should not be used for real-time systems, because worst-case execution time is not bounded (i.e. not predictable).
- ▶ also has a bus saturation problem.
 - ▶ Exponential back-off scheme helps, but is even worse for real-time systems.

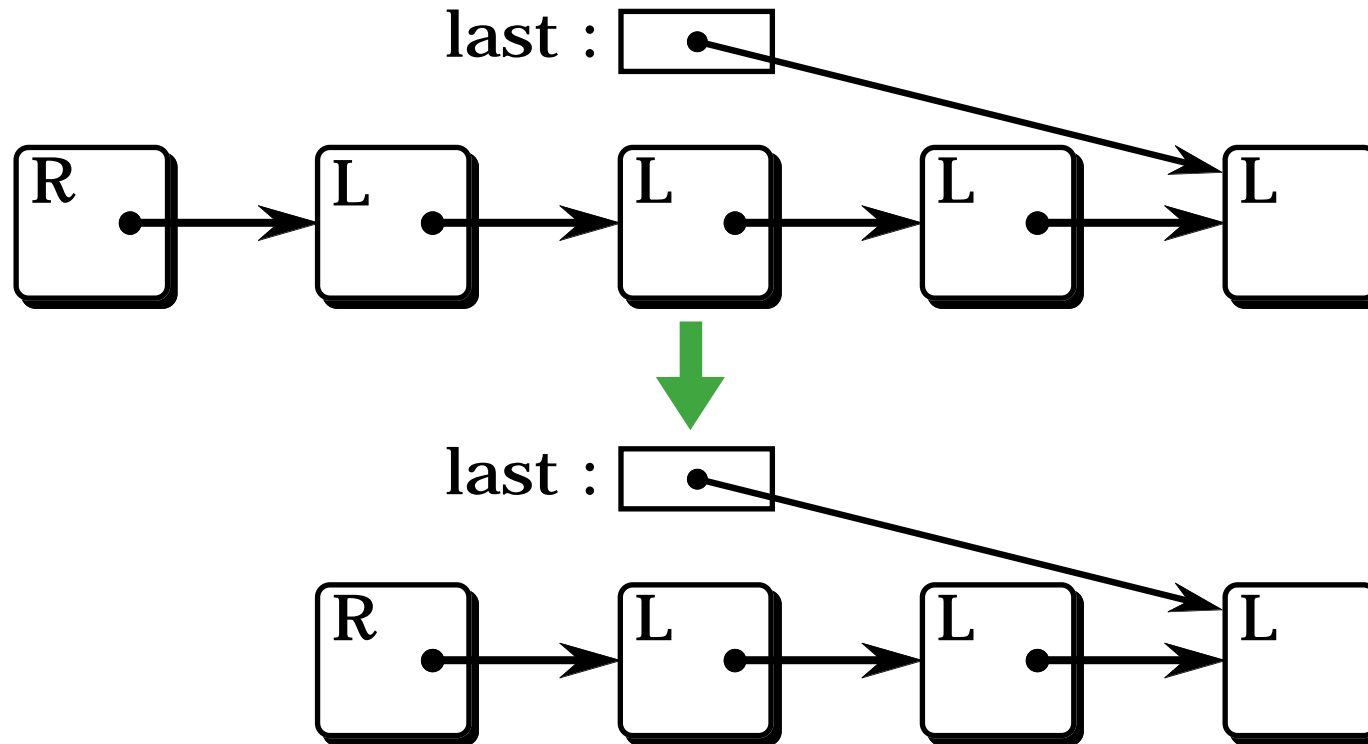
MCS Lock – a queueing spin lock algorithm

[1] J. M. Mellor-Crummey and M. L. Scott, Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors, ACM Trans. Computer Systems, vol. 9, no. 1, pp. 21–65, Feb. 1991.

- ▶ Waiting processors for a lock form a FIFO queue.



- ▶ When the lock holder releases the lock, it passes the lock to the top processor in the queue.



- ▶ *An FIFO order is guaranteed.*
- ▶ *Only local spin occurs.*

Priority-ordered Spin Locks

- ▶ There are several priority-ordered spin lock algorithms proposed.
- ▶ **Markatos' Lock**
 - [2] E. P. Markatos, Multiprocessor Synchronization Primitives with Priorities, Proc. 8th IEEE Workshop on Real-Time Operating Systems and Software, May 1991.
- ▶ **Craig's Lock**
 - [3] T. S. Craig, Queuing Spin Lock Algorithms to Support Timing Predictability, Proc. Real-Time Systems Symposium, pp. 148–157, Dec. 1993.
- ▶ **PR-Lock**
 - [4] T. Johnson and K. Harathi, A Prioritized Multiprocessor Spin Lock, Technical Report TR-93-005, Dept. of Computer Science, Univ. of Florida, 1993.

Problem of Inopportune Preemption

What is the Problem of Inopportune Preemption?

- ▶ occurs when spin locks are used for *multiprogrammed multiprocessors*.
- ▶ two problematic cases:

(1) A task is preempted while it is holding a lock.

(2) A task is handed a lock while it is preempted.

in implementing an RTOS....

- ▶ Case (1) is prevented by inhibiting task preemptions (by disabling interrupt services) while a task is holding a lock.
- ▶ Case (2) is serious.
 - ← With *queueing spin locks*, the turn that a task acquires a lock is *reserved*.

Lock Acquisition and Disabling Interrupt Services

! Lock acquisition and disabling interrupt services
must be atomic,



- ▶ Interrupt requests should be serviced while waiting for a lock.

for making the maximum interrupt latency independent of the number of processors.

- ▶ Interrupt requests should be suspended once a processor acquires a lock.

because maximum interrupt service time is quite long in general

and the other processors contending for the lock must wait for the time wastefully.

Test&Set Lock with Preemption

- ▶ Test&Set Lock can be easily modified to be *preemption-safe*.

```
disable_interrupt;
while test_and_set(L) = Locked do
    if interrupt_requested then
        enable_interrupt;
        // service interrupt.
        disable_interrupt;
    end
end;
// critical section.
```

- ! The same scheme cannot be applied to queueing spin locks straightforwardly because of the case (2) problem.

Queueing Spin Locks with Preemption

- ▶ Two schemes to add preemption to MCS lock have been proposed.

preemption-safe versions of MCS lock

Basic Preemption Scheme

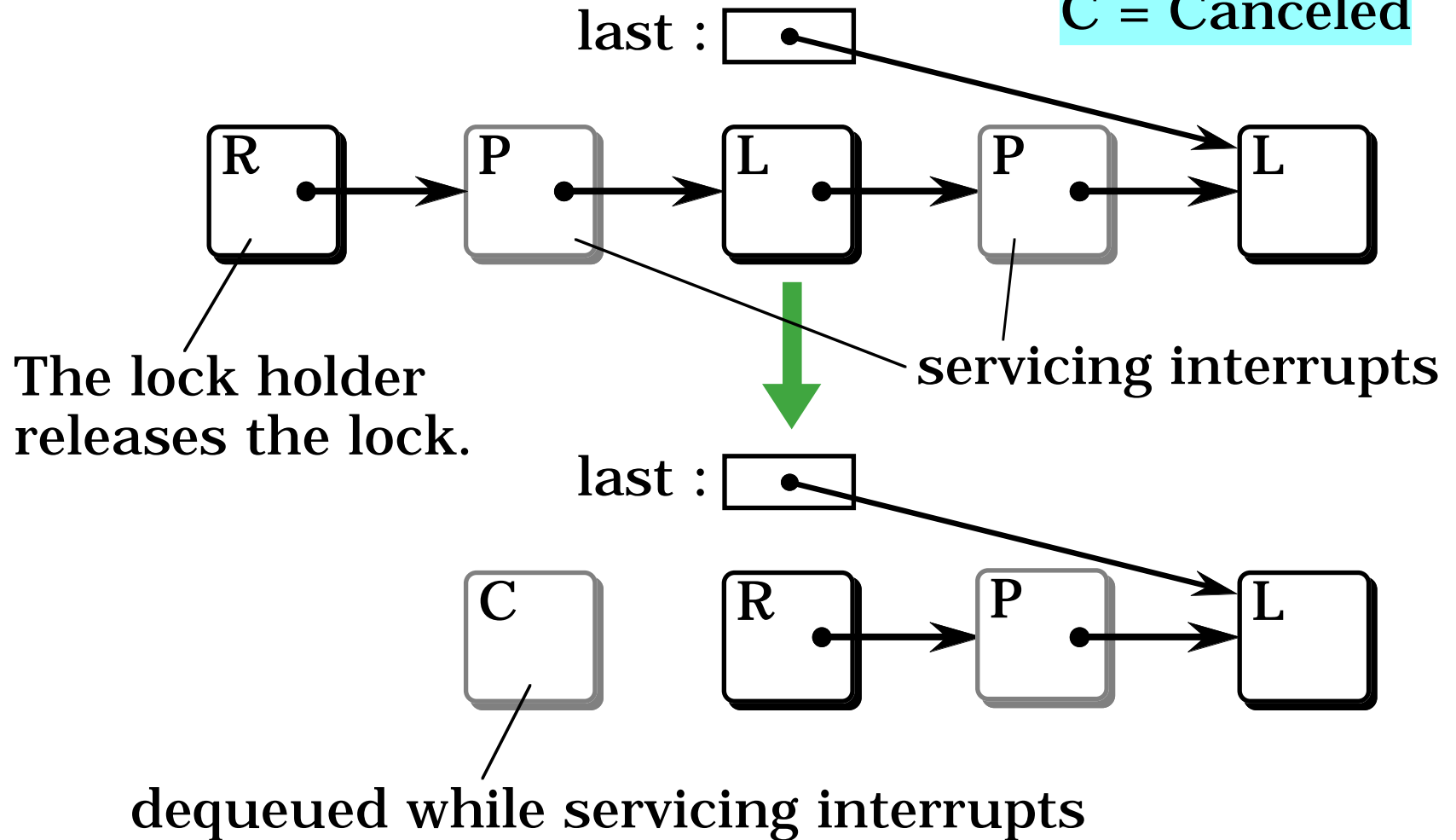
[5] R. W. Wisniewski, L. Kontothanassis, and M. L. Scott, Scalable Spin Locks for Multiprogrammed Systems, Proc. 8th Int'l Parallel Processing Symposium, Apr. 1994.

[6] H. Takada and K. Sakamura, A Bounded Spin Lock Algorithm with Preemption, Technical Report 93-2, Dept. of Information Science, Univ. of Tokyo, Jul. 1993.

- ▶ A task *informs other processors* that *it is preempted*, when it is preempted while waiting for a lock.
- ▶ The releasing task removes the preempted task from the queue, in other words, *Cancels the reservation.*

Illustrating Basic Preemption Scheme

P = Preempted
C = Canceled



Drawback of Basic Preemption Scheme

- ▶ A dequeued processor (while servicing interrupts) must re-execute the lock acquisition routine from the *beginning* after it finishes the interrupt service.
- ▶ This re-execution overhead should be added to the interrupt service time in schedulability analysis.
*then called as the **interrupt service overhead***



- ▶ The interrupt service overhead depends on the number of processors.
This violates property (D).

Improved Preemption Scheme

[7] H. Takada and K. Sakamura, Predictable Spin Lock Algorithms with Preemption, Proc. Real-Time Operating Systems and Software, pp. 2-6, May 1994.

- ▶ The task releasing the lock remains the preempted processors in the queue, in other words, *postpones the reservation*.



- ▶ The interrupt service overhead can be bounded with a *constant time length*.

SPEPP Synchronization

- [8] H. Takada and K. Sakamura, A novel approach to multiprogrammed multiprocessor synchronization for real-time kernels, Proc. 18th IEEE Real-Time Systems Symposium, Dec. 1997.

Approach

- ▶ If a task's turn to acquire the lock comes while the task is preempted, its operation is *executed by another processor* that is spinning on the lock.



Basic Idea

- ▶ making *one of* the idling (*spinning*) processors work for busy (*preempted*) processors!

Naming

- ▶ SPEPP = ***S**pinning **P**rocessor **E**xecutes for **P**reempted **P**rocessors*

Framework of SPEPP Synchronization Algorithms

- ▶ The kind of operation and its parameters are stored in an operation block.
 - ▶ The area to write the return values is included in the operation block.
- 
- ▶ The operation block is *posted to a **FIFO-ordered** operation queue*
- 
- ▶ *One of the spinning processors **is selected** and executes the operation at the head of the operation queue.* → **Operation blocks in the queue are processed in a *strict* FIFO order.**
 - ▶ *Any spin lock algorithm can be used for the selection.*

Basic SPEPP Synchronization based on Test&Set Lock

Rough Description of the Algorithm

- ▶ The preemption-safe test&set lock is adopted to the previous framework.
- ▶ A task tries to acquire the lock, only when the lock is idle and its operation has not been executed.
- ▶ two optimizations
 - (1) A task can execute more than one operations in the queue without releasing the lock.
 - (2) The data structures for the spin lock and the operation queue are merged.

Timing Behavior

T : the maximum execution time of an operation

N : the number of contending tasks

- ▶ The *maximum interrupt response* is T (+ *const*).
- ▶ The *maximum execution time* until a task finishes its own operation is $N \cdot T$ (+ *const*).
- ▶ The *interrupt service overhead* is **zero** except for some lock handling overhead.
 - ▶ The execution of the operations continues to make progress, unless all the tasks are preempted.
 - ▶ If all the tasks are preempted, the execution is suspended, but its is shorter than any one of the preemptions.

Problem

- ▶ $N \cdot T$ becomes very large when the number of tasks is large.

Extended SPEPP Synchronization

Rough Description of the Algorithm

- ▶ An operation block is prepared *for each processor*.
- ▶ The task trying to acquire the lock uses the queue node, *even if a lower priority task is using it*.
- ▶ The lower priority task must *retry the operation*, if the queue node is stolen by a higher priority one.

Timing Behavior

n : the number of contending *processors*

- ▶ The *maximum execution time* until a task finishes its own operation may be considered as $(n+1) \cdot T (+ \text{const})$.

An operation by a lower priority task may have already started by another processor when stealing the queue node.

Other SPEPP Synchronization Algorithms

SPEPP Synchronization based on MCS Lock

- ▶ A *preemption-safe MCS lock* is adopted instead of test&set lock.

Priority-ordered Execution

- ▶ can be realized with a *priority-ordered operation queue* (or a *preemption-safe priority-order spin lock*)

Limitation of SPEPP Synchronization

- ▶ Operations on a shared data structure must be executable on any processor.

in other words

Private data of a processor must be passed in the operation block and restored from it.

→ *performance penalty*

Other Approaches to Inopportune Preemption

Wait-free / Lock-free Synchronization

- ▶ *synchronization without locking*
- ▶ *wait-free* → *inefficient* to implement a complex data structures within a real-time kernel
- ▶ *lock-free* → *difficult to predict* the maximum (= *non-blocking*) execution time with *multiprocessors*

Solutions with Task Scheduling

- ▶ Locks and/or preemptions are avoided with task scheduling.

Scalability of Nested Spin Locks

Nested Spin Locks

- ▶ System resources that must be accessed exclusively by a processor are usually divided into some *lock units* to exploit parallelism.
- ▶ When a processor accesses some resources included in different lock units, the processor must acquire *multiple locks one by one*.

Example

```
acquire_lock(L2);  
acquire_lock(L1);  
// critical section  
release_lock(L1);  
release_lock(L2);
```

Scalability Problem of Nested Spin Locks

- ▶ **Scalability** of the **maximum execution times** of critical sections guarded by **nested spin locks** is discussed below.
- ▶ With the simple methods, the maximum execution times of nested spin locks is $O(n^m)$.
 - n : the number of contending processors
 - m : the maximum nesting level of locks
 - *unacceptable from scalability point of view*
- ▶ With the ***totally FIFO approach***, this can be reduced to $O(n \cdot e^m)$.

[9] H. Takada and K. Sakamura, Real-Time Scalability of Nested Spin Locks, Proc. 2nd Real-Time Computing Systems and Applications, pp. 160–167, Oct. 1995.

Example (when $m = 2$)

- ▶ Suppose the case that each processor repeatedly executes one of the three routines below.

```
acquire_lock(L1);  
// critical section  
release_lock(L1);
```

routine (a)

```
acquire_lock(L2);  
// critical section  
release_lock(L2);
```

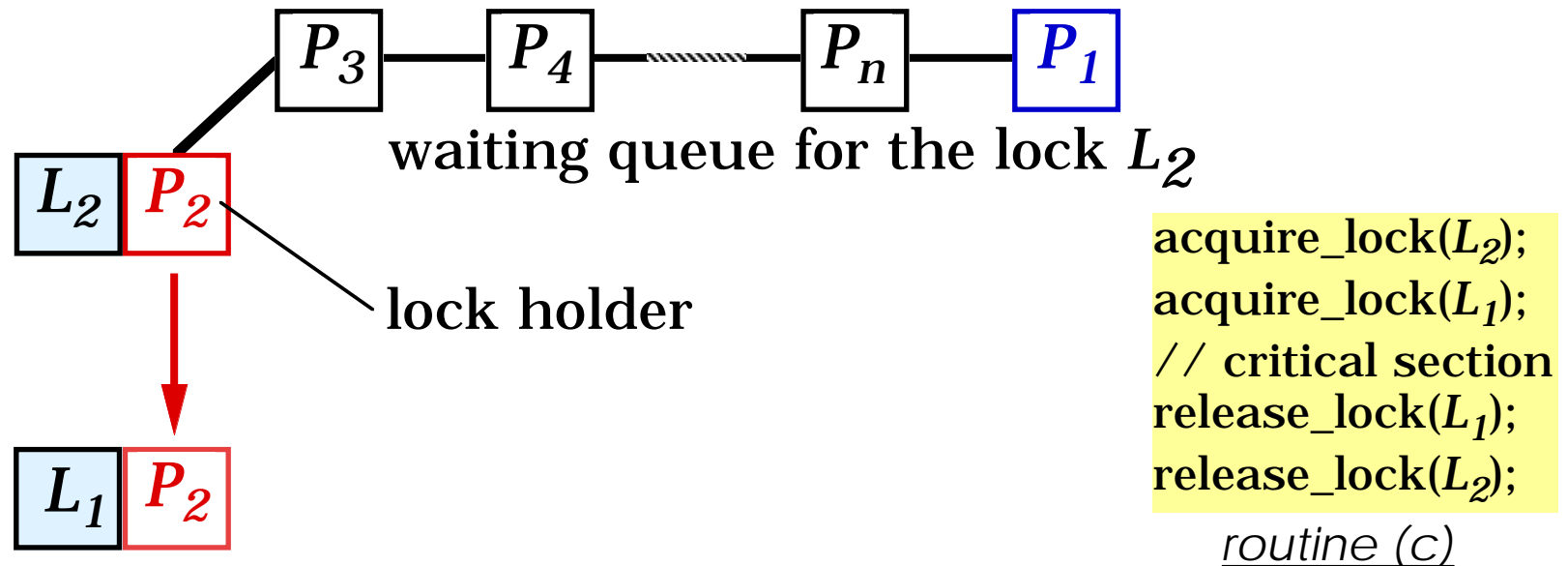
routine (b)

```
acquire_lock(L2);  
acquire_lock(L1);  
// critical section  
release_lock(L1);  
release_lock(L2);
```

routine (c)

Example (cont.)

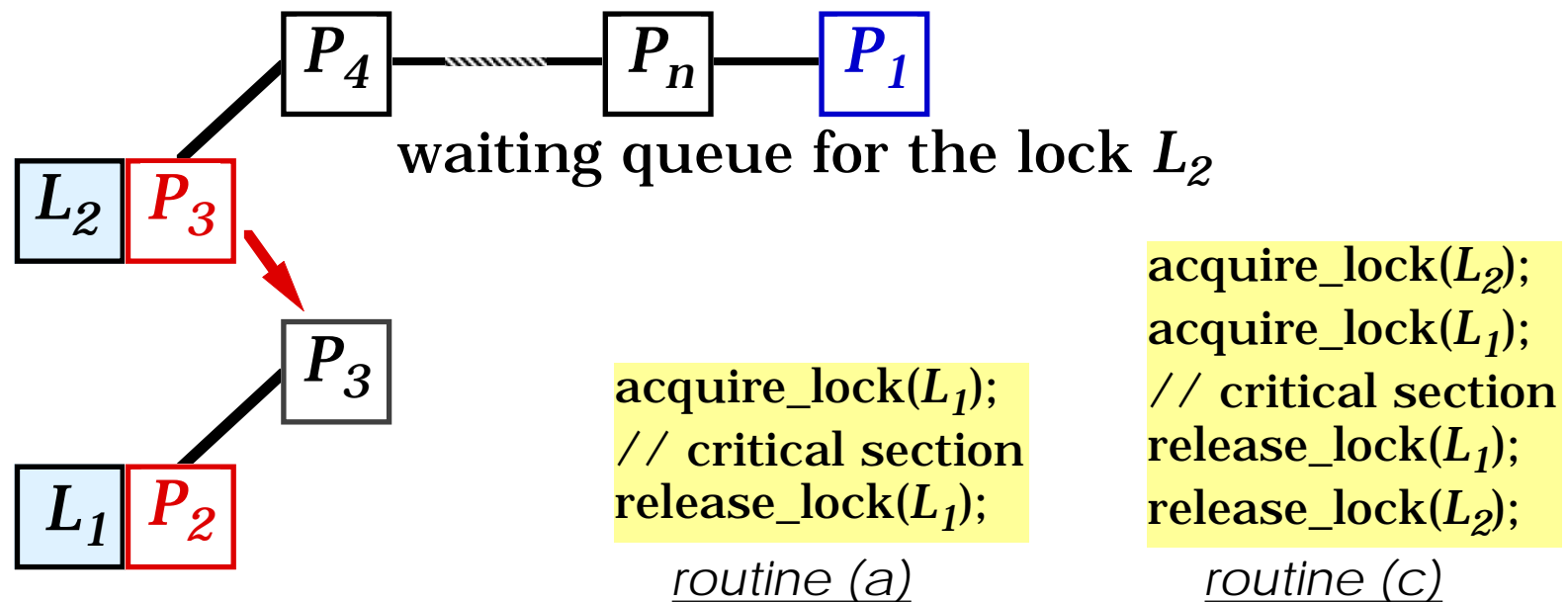
- ▶ If the locks are simply implemented with a **FIFO spin lock** algorithm, the maximum execution times of routine (b) and (c) become $O(n^2)$.
- ▶ The worst-case scenario that P_1 executes routine (c) is as follows.



- ▶ P_1 waits for the critical section executed by P_2 .

Example (cont.)

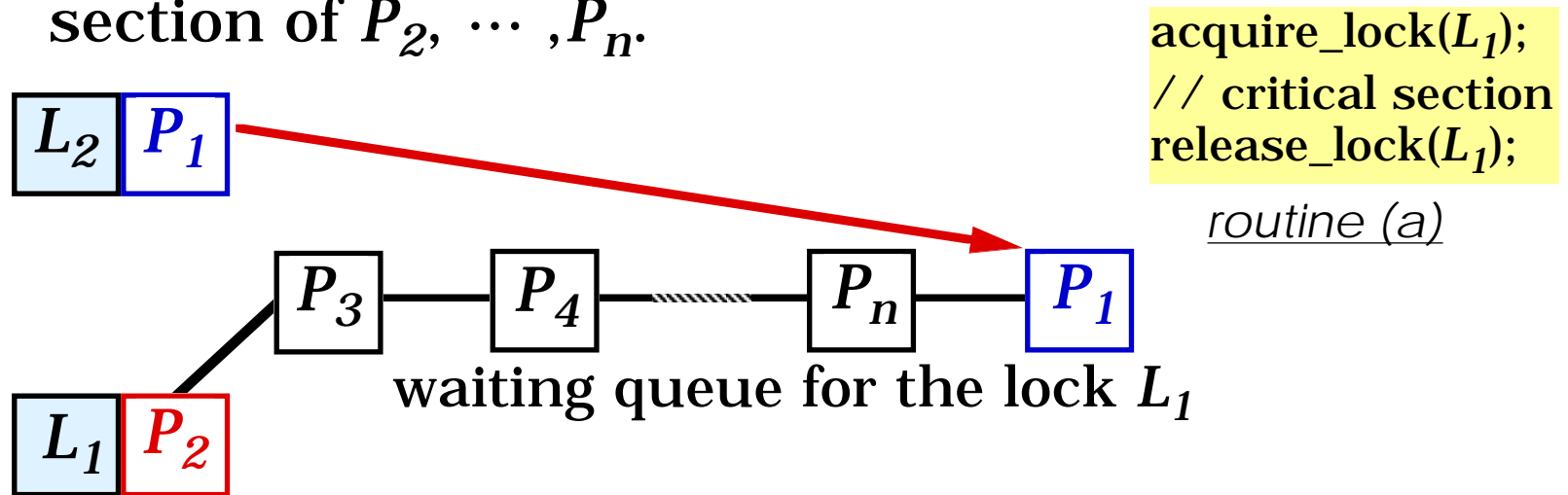
- ▶ After P_2 releases L_2 , P_3 can acquire L_2 .
But, before P_3 acquires L_1 , P_2 can acquire L_1 .
Then, P_3 must wait for the critical section of P_2 .



- ▶ P_1 waits for the critical sections of P_2 and P_3 .

Example (cont.)

- ▶ After P_n releases L_2 , P_1 can acquire L_2 .
But, before P_1 acquires L_1 , P_2, \dots, P_n can be waiting for L_1 . Then, P_1 must wait for the critical section of P_2, \dots, P_n .



- ▶ P_1 waits for the critical sections of P_2, \dots, P_n .
- ▶ As the result, P_1 must wait for **at most $O(n^2)$** critical sections until it finishes an execution of routine (c).

Totally FIFO Approach

- ▶ obtaining a *time stamp* when a processor begins waiting for the outermost lock
- ▶ using a *priority-order spin lock* algorithm with the *time stamps as the priorities*



- ▶ The maximum execution times of critical sections is reduced to $o(n)$ when m is constant.

Optimizations

- ▶ A *FIFO spin lock* can be used for the outermost lock (more exactly, the lock with the maximum nesting level).
- ▶ A *sequence number* that a processor begins waiting for the outermost lock can be used instead of a time stamp.

When Nesting in Three or More Levels

- ▶ With simple application of totally FIFO approach, the maximum execution times **cannot** be improved to $O(n)$ due to uncontrolled priority inversions.



Priority inheritance scheme can be adopted to solve this problem.

priority inheritance spin lock

- ▶ With totally FIFO approach with basic priority inheritance spin lock, the maximum execution times can be improved to $O(n \cdot e^m)$.

Priority Inheritance Spin Locks

Priority Inversion Problem in Nested Spin Locks

- ▶ The problem of *uncontrolled priority inversion* in task scheduling is well-known and well-studied.
- ▶ *Uncontrolled priority inversion* also occurs in *nested spin locks*, but the situation is different.

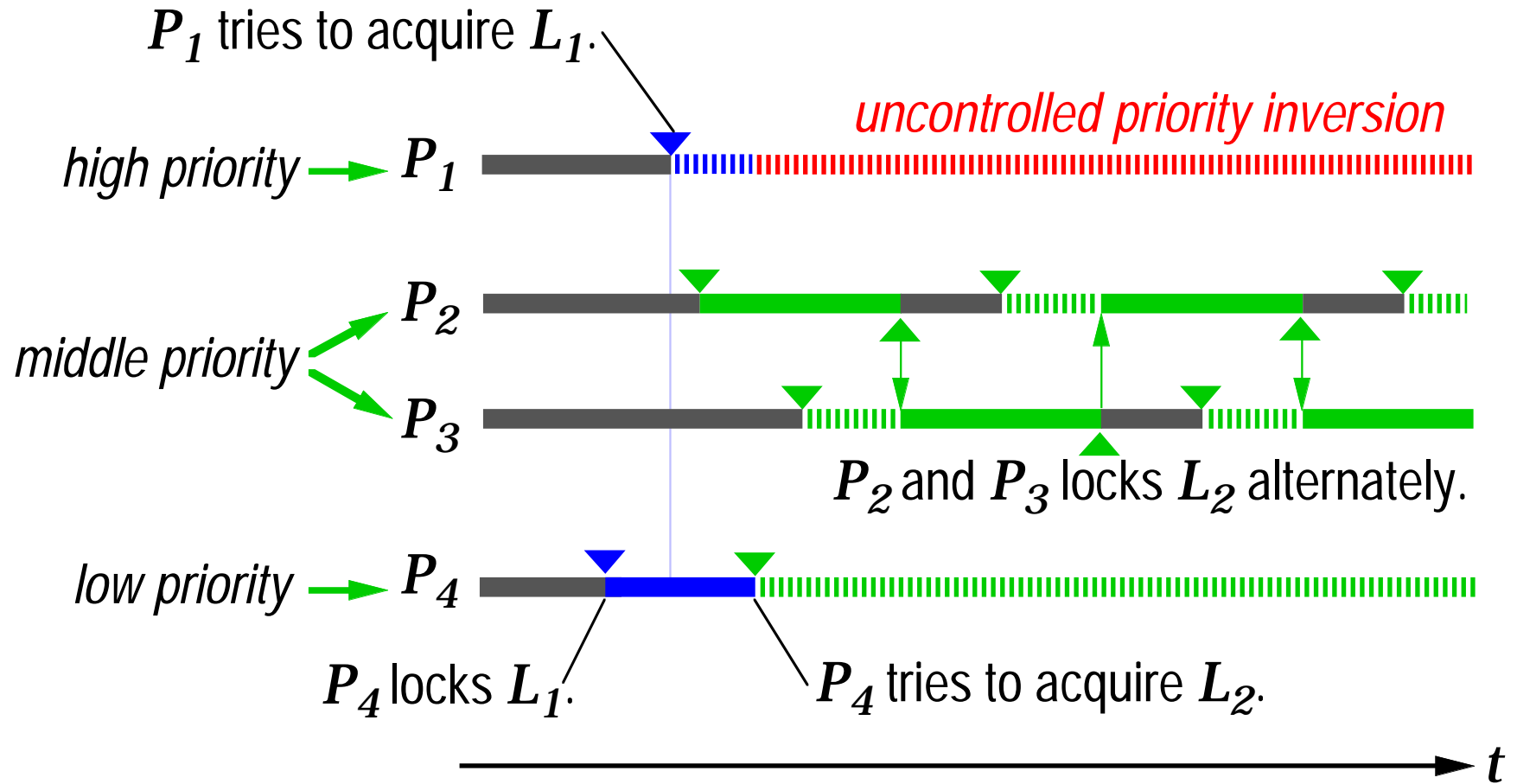
Example (uncontrolled priority inversion)

- ▶ Assume that four processors repeatedly execute one of the two routines below in random order.

```
acquire_lock(L2);  
// critical section.  
release_lock(L2);
```

```
acquire_lock(L1);  
acquire_lock(L2);  
// critical section.  
release_lock(L2);  
release_lock(L1);
```

Example (cont.)

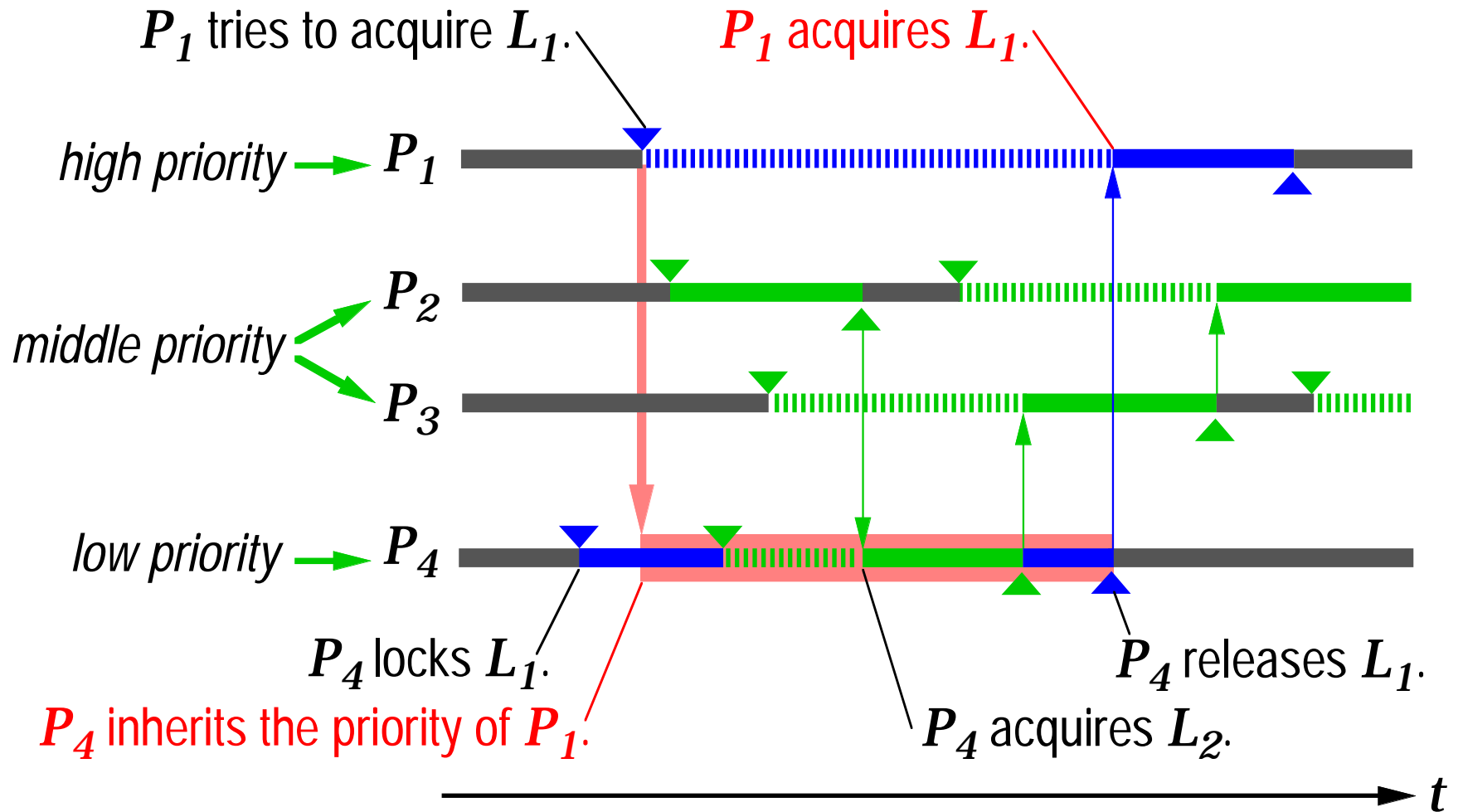


Incorporating Priority Inheritance Scheme

[9] C.-D. Wang, H. Takada and K. Sakamura, Priority Inheritance Spin Locks for Multiprocessor Real-Time Systems, Proc. Int'l Symposium on Parallel Architectures, Algorithms, and Networks, pp. 70–76, Jun. 1996.

- ▶ **Priority inheritance** is a promising approach to solve the uncontrolled priority inversion problem in task scheduling.
 - *apply the scheme to nested spin locks*
- ▶ basic priority inheritance scheme **for spin locks**
 - ▶ When a processor makes some higher priority processors wait, it **inherits** the highest priority among them.
 - ▶ Priority inheritance must be **transitive**.

Example with Priority Inheritance Scheme



RTOS Implementation Issues

? Is a scalable RTOS with the desirable properties implementable with those synchronization techniques?

Desirable Properties for Scalable RTOS (again)

- (A) The maximum execution time of a system call that is to synchronize with tasks on the **same processor** should be $O(1)$.
- (B) The maximum execution time of a system call that is to synchronize tasks on **other processors** should be $O(n)$.
- (C) The maximum **interrupt response** time on each processor should be $O(1)$.
- (D) The **interrupt service overhead** should be $O(1)$.

Granularity of Lock Units

- ▶ Access pattern on kernel data structures in each system call of a real-time kernel is investigated.



- ▶ *two lock units for each processor*

(a) *task lock*: the task control blocks and the ready queue on the processor  *acquisition order*

(b) *object lock*: the control blocks of the task-independent synchronization objects on the processor



- ▶ Usual operations on a task need one lock.
- ▶ Operations on a synchronization object may need at most *two locks at the same time*.

Without Task-Independent Synchronization Objects

- ▶ Each system call requires only one lock at once.
no nested spin locks
- ▶ *in order to satisfy (A),*
 - ▶ **Spin lock with local precedence**, with which a processor can acquire its local lock with precedence over other processors, should be used.
- ▶ *in order to make (B) and (C) compatible,*
 - ▶ **Queueing spin lock with preemption** is necessary.
- ▶ *in order to satisfy (D),*
 - ▶ **Improved preemption scheme** should be used.
- ➔ *All properties can now be satisfied.*
- ! SPEPP synchronization** can also satisfy all properties.

Supporting Task-Independent Synchronization Objects

[10] H. Takada, C.-D. Wang, and K. Sakamura, Issues for realizing a scalable real-time kernel for function-distributed multi-processors, Work in Progress Session of 17th IEEE Real-Time Systems Symposium, pp. 23-26, Dec. 1996.

- ▶ Two locks, *an object lock* then *a task lock*, are necessary to be acquired in some system calls.
- ▶ *in order to satisfy (A)*,
 - ▶ Synchronization objects should be classified into *private objects*, which are accessible only by the tasks on its host processor, and *shared objects*.
- ▶ *in order to satisfy (B)*,
 - ▶ **Totally FIFO approach** must be used.

- ▶ *in order to satisfy (C),*
 - ▶ A preemption scheme is necessary.

```
retry:
  acquire_lock(Object_Lock);
  if [an interrupt request is detected while waiting] then
    [service the interrupt];
    goto retry;
  end;
  [determine which lock to acquire next];
  acquire_lock(Task_Lock);
  if [an interrupt request is detected while waiting] then
    release_lock(Object_Lock);
    [service the interrupt];
    goto retry;
  end;
```

- ▶ *in order to satisfy (D),*
 - ▶ After a processor returns from an interrupt service requested while waiting for the *inner* lock, the processor should wait for the outer lock *at the top* of its waiting queue.



- ▶ *This violates (B), however.*
- *No method with which all properties are satisfied has not been proposed.*

A Possible Approach to the Realization

- ▶ Applying the SPEPP synchronization approach to nested spin locks.
not investigated yet (not easy)

Concluding Remarks

- ▶ Various inter-processor synchronization methods for implementing a scalable RTOS have been discussed, assuming that atomic operations on a single word of memory is supported with hardware.
- ▶ Under this assumption, the overhead of those synchronization algorithms is considerably large.
- ▶ With a SoC, more sophisticated hardware support is possible and should be investigated. What kind of hardware support is desirable is an open question.
 - one extreme: implementing RTOS with hardware*