Embedded Real-Time Systems—Lecture 13

Prof. Dr. Reinhard von Hanxleden

Christian-Albrechts Universität Kiel
Department of Computer Science
Real-Time Systems and Embedded Systems Group

14 January 2009

Last compiled: January 19, 2009, 22:34 hrs

Priority Inversion
Scheduling in POSIX and RT Java

Overview

Priority Inversion
- Process Blocking and Priority Inversion
- Priority Inheritance
- Calculating the blocking time
- Priority ceiling protocols
- Offline vs. Online scheduling

POSIX scheduling support
- Sporadic servers
- Inheritance of scheduling characteristics
- Keeping RT applications in check
- Priority inheritance

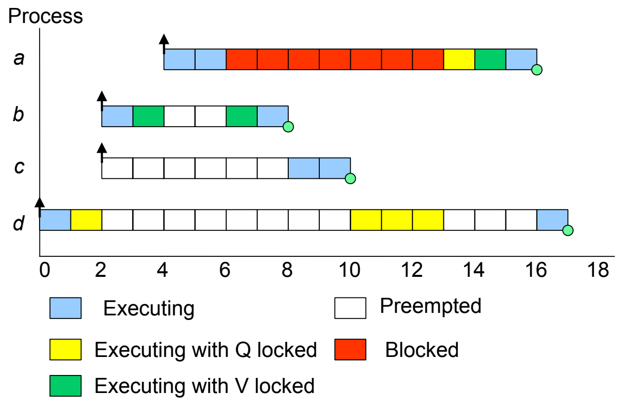
RT Java scheduling support
- The scheduler
- Avoiding priority inversion

Process Interactions and Blocking

- A process may be blocked:
  Suspended waiting for a lower-priority process to complete some required computation or to release a resource
- Example: Processes *a*, *b*, *c*, and *d*, accessing resources Q and V as follows

Process	Priority (P)	Release Time	Execution Sequence E: Execute Q: Access Resource Q V: Access Resource V
a	4	4	EEQVE
b	3	2	EVVE
c	2	2	EE
d	1	0	EQQQQE

Example of Resource Sharing



Priority Inversion

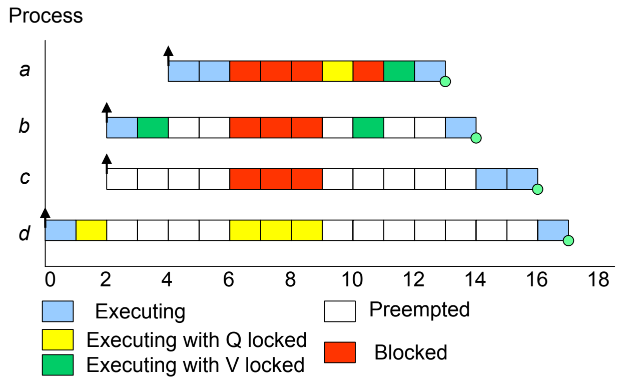
- Priority inversion: A process (a) is blocked waiting for a lower-priority process (d) to complete some required computation or to release a resource
- Some of this is inevitable if
  - Resources are shared among processes of different priority
  - A lower-priority process that uses a resource cannot be preempted from using this resource by a higher-priority process
- **However**, the (indirect) blocking of the high-priority process (a) by medium-priority processes (b and c) that prevent the low-priority process (d) from freeing the resource can (and should) be avoided

One Solution: Nonpreemptive Critical Sections

- Consider the interval from allocating a resource until releasing the resource a critical section
- Nonpreemptive Critical Sections Protocol: Schedule all critical sections nonpreemptively
  - When job requests resource, always allocate the resource
  - When job holds resource, executes at priority higher than priority of all other jobs
- Disadvantage: High-priority jobs may get delayed unnecessarily

Another Solution: Priority Inheritance

If process p is blocking process q, it runs with q's priority



Note: We consider a process "preempted" if it is preempted by a process with a *statically* higher priority; otherwise, we consider it "blocked"

Calculating Maximal Blocking Time

- If a process has K critical sections that can lead to it being blocked, then the maximum number of times it can be blocked is K
- Let $usage(k, i)$ be 1 if resource used by critical section k is used by at least one process with priority less than P_i, and by at least one process with priority greater or equal to P_i; otherwise 0
- Let $C(k)$ be the WCET of critical section k
- The the maximum blocking time B_i of process i is then given by:

$$\boxed{B_i = \sum_{k=1}^{K} usage(k, i) C(k)}$$

- Note: In the definition of $usage(k, i)$, "one process with priority greater or equal to P_i" may be i itself as well.
- Note also that [Burns and Wellings] here treats "critical sections" and "resources" as synonymous. We here separate these concepts, to allow one resource to be used multiple times by the same process, in different critical sections.

Response Time and Blocking

- If we have obtained a blocking time, we can extend formula (5) as follows:

$$\boxed{R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j}$$

- This again can be expressed as a recurrence:

$$\boxed{w_i^{n+1} = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j}$$

Priority Ceiling Protocols

- The standard priority inheritance protocol gives an upper bound on the number of blocks a high-priority process can encounter—however, this may lead to an unacceptably pessimistic worst case
- Furthermore, transitive blocking may lead to chains of blocks—process c is blocked by b which is blocked by a etc.
- In addition, we want to eliminate deadlock when accessing the resource
- All these issues are addressed by the Priority Ceiling protocols (PCPs), of which we will discuss two forms:
  - Original priority ceiling protocol, OPCP
  - Immediate priority ceiling protocol, IPCP

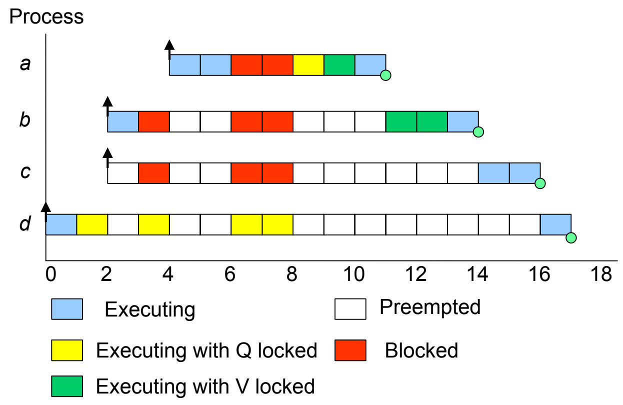
PCP Properties on a Single Processor

- A high-priority process can be blocked at most once during its execution by any lower-priority processes
- Deadlocks are prevented
- Transitive blocking is prevented
- Mutual exclusive access to resources is ensured, by the protocol itself (no semaphores etc. required)

The Original Priority Ceiling Protocol (OPCP)

1. Each process has a static default priority assigned, perhaps by the deadline monotonic scheme
2. Each resource has a static ceiling value defined, this is the maximum priority of the processes that will use it
3. A process has a dynamic priority that is the maximum of its own static priority and any it inherits due to it blocking higher-priority processes
4. A process can only lock a resource if:
   - Its dynamic priority is higher than the ceiling of any currently locked resource (excluding any that it has already locked itself)

OPCP Inheritance

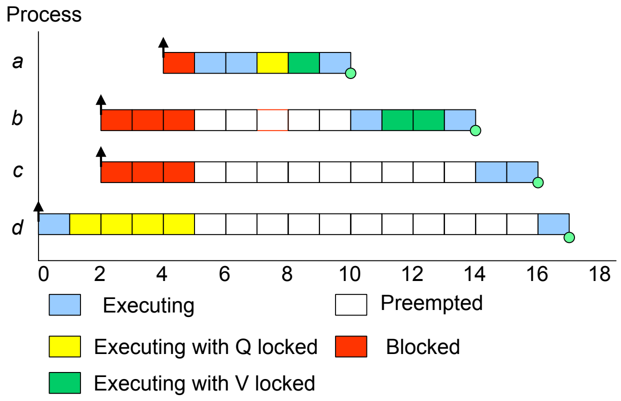


The Immediate Priority Ceiling Protocol (IPCP)

1. Again, each process has a static default priority assigned, perhaps by the deadline monotonic scheme
2. Also, each resource has a static ceiling value defined, this is the maximum priority of the processes that will use it.
3. A process has a dynamic priority that is the maximum of its own static priority and the ceiling values of any resources it has locked

- As a consequence, a process will only suffer a block at the very beginning of its execution
  Note: We assume that a process is only pre-empted by other processes of *higher* priority, not of the same priority
- Once the process starts actually executing, all the resources it needs must be free
  If they were not, then some process would have an equal or higher priority and the process's execution would be postponed

IPCP Inheritance



OPCP versus IPCP

- Although the worst-case behaviour of the two ceiling schemes is identical (from a scheduling view point), there are some points of difference:
  - IPCP is easier to implement than the original (OPCP) as blocking relationships need not be monitored
  - IPCP leads to less context switches as blocking is prior to first execution
  - IPCP requires more priority movements as this happens with all resource usage
  - OPCP changes priority only if an actual block has occurred
- Note: IPCP is also called
  - Priority Protect Protocol (in POSIX)
  - Priority Ceiling Emulation (in Real-Time Java)

Extensions

- There are numerous extensions to the simple process model discussed so far—for example:
  - Cooperative Scheduling
  - Release Jitter
  - Arbitrary Deadlines
  - Fault Tolerance
  - Offsets
  - Optimal Priority Assignment
- These extensions address real problems—however, today's RT languages and OSs very rarely support any of these extensions
- See [Burns and Wellings 2001] or [Liu 2000] for a detailed treatment

Offline vs. Online Scheduling Analysis

- For hard RT systems, an offline scheduling analysis is desirable—and often mandatory
- However, this analysis requires
  - bounded and known arrival patterns of incoming work
  - bounded and known (!) computation times
  - a predictable scheduling scheme
- However, there are dynamic soft real-time applications in which arrival patterns and computation times are not known a priori
- Although some level of offline analysis may still be applicable, this can no longer be complete and hence some form of online analysis is required

Dynamic Systems and Online Analysis

- The main task of an on-line scheduling scheme is to manage any overload that is likely to occur due to the dynamics of the system's environment
- EDF is a dynamic scheduling scheme that is optimal
- However, during transient overloads EDF performs very badly. It is possible to get a domino effect in which each process misses its deadline but uses sufficient resources to result in the next process also missing its deadline.

Admission Schemes

- To counter this detrimental domino effect, many on-line schemes have two mechanisms:
  - an admissions control module that limits the number of processes that are allowed to compete for the processors, and
  - an EDF dispatching routine for the admitted processes
- An ideal admissions algorithm prevents the processors getting overloaded so that the EDF routine works effectively
- If some processes are to be admitted, whilst others rejected, the relative importance of each process must be known
- This is usually achieved by assigning a value

Values

- Values can be classified
  - Static: the process always has the same value whenever it is released.
  - Dynamic: the process's value can only be computed at the time the process is released (because it is dependent on either environmental factors or the current state of the system)
  - Adaptive: here the dynamic nature of the system is such that the value of the process will change during its execution
- To assign static values requires the domain specialists to articulate their understanding of the desirable behaviour of the system—this is not always a trivial task . . .

Programming Priority-Based Systems

- Traditionally, priority-based scheduling has been more an issue with OSs rather than with programming languages
- In the introduction to scheduling, we discussed the scheduling-related facilities provided by standard UNIX
- Will further discuss the scheduling interfaces provided by
  - POSIX
  - Real-Time Java

Summary Priority Inversion

- Processes may be blocked by lower-priority processes—this is referred to as priority inversion
- A solution for this is priority inheritance—however, this may still lead to more blocking than necessary
- Priority ceiling protocols are an improvement over priority inheritance protocols—we discussed the Original PCP and the Immediate PCP

Overview

Priority Inversion

Process Blocking and Priority Inversion
Priority Inheritance
Calculating the blocking time
Priority ceiling protocols
Offline vs. Online scheduling

POSIX scheduling support

Sporadic servers
Inheritance of scheduling characteristics
Keeping RT applications in check
Priority inheritance

RT Java scheduling support

The scheduler
Avoiding priority inversion

POSIX

- POSIX supports priority-based scheduling, and has options for priority inheritance and ceiling protocols
- Priorities may be set dynamically
- Within the priority-based facilities, there are four policies:
  1. `SCHED_FIFO`: a process/thread runs until it completes or it is blocked
  2. `SCHED_RR`: a process/thread runs until it completes or it is blocked or its time quantum has expired (Round-Robin)
     This is equivalent to `priocntl`'s `RT_TQDEF`, but here it cannot be changed
  3. `SCHED_SPORADIC`: a process/thread runs as a sporadic server (still rarely implemented)
  4. `SCHED_OTHER`: implementation-defined

Sporadic Server

A sporadic server process is assigned a certain budget

- Runs at high priority when it has some budget left
- Consumed execution time is subtracted from budget
- Runs at low priority when budget is exhausted
- The amount of budget consumed is replenished at the time the server was activated plus the replenishment period

A sporadic server is useful to integrate background, soft RT processes in a hard RT environment

- The sporadic server is assured a high priority for a limited time
- Beyond that, the sporadic server process cannot interfere with hard RT tasks (that presumably have a higher priority than the sporadic server's low priority)

POSIX

- For each policy, must support a minimum range of priorities
- 32 priorities for FIFO and round-robin

```
#include <unistd.h>
#ifdef _POSIX_PRIORITY_SCHEDULING
#include <sched.h>

typedef ... pid_t;
struct sched_param {
  ...
  int      sched_priority;          // SCHED_FIFO and SCHED_RR
  int      sched_ss_low_priority;   // SCHED_SS
  timespec sched_ss_repl_period;
  timespec sched_ss_init_budget;
  int      sched_ss_max_repl;
  ...
}
```

POSIX Process-Based Scheduling

```
// Set and get the scheduling parameters of process pid
int sched_setparam (pid_t                        pid,
                    const struct sched_param *param);
int sched_getparam (pid_t                    pid,
                    struct sched_param     *param);

// Set and get the scheduling policy/params of pid
int sched_setscheduler (pid_t                        pid,
                        int                        policy,
                        const struct sched_param *param);
int sched_getscheduler (pid_t                    pid);

// Get info on scheduling policies
int sched_get_priority_max (int policy);
int sched_get_priority_min (int policy);
int sched_rr_get_interval(pid_t                pid,
                          struct timespec *t);

// Places current process/thread at the back of the queue
int sched_yield (void);
```

POSIX

- The scheduling policy can be set on a per process and a per thread basis
- Threads compete with other threads
  - across the whole system (PTHREAD_SCOPE_SYSTEM), or
  - per process (PTHREAD_SCOPE_PROCESS)
    It is unspecified how such threads are scheduled relative to threads in other processes or to threads with global contention
- A specific implementation must decide which to support

POSIX Thread-Based Scheduling

```
// Set and get the contention scope for a thread attribute
int pthread_attr_setscope(pthread_attr_t *attr,
                          int            contentionscope);
int pthread_attr_getscope(pthread_attr_t *attr,
                          int            contentionscope);

// Set and get the scheduling policy for a thread attribute
int pthread_attr_setschedpolicy(pthread_attr_t *attr,
                                int            policy);
int pthread_attr_getschedpolicy(pthread_attr_t      *attr,
                                int              policy);

// Set and get the scheduling params for a thread attribute
int pthread_attr_setschedparam(pthread_attr_t *attr,
                               const struct sched_param *p);
int pthread_attr_getschedparam(pthread_attr_t *attr,
                               const struct sched_param *p);
```

Like Father, Like Son

- By default, each process inherits the scheduling policy and priority from the parent process
- Scheduling attributes are inherited across a fork or one of the `exec` family of calls
- Example:
  `% get_data | process_data > /dev/out < /dev/in`
  - All processes in the pipe run with the scheduling attributes of the shell
- POSIX.1b exactly defined when a child process runs after a fork
  Assuming a single processor, the parent occupies the CPU until it blocks

Forking From the Shell

- Consider a program `atprio` that sets the priority of a process from the shell, using the FIFO scheduler (⇒ Homework)
- Example:
  `% atprio 127 get_data | process_data > /dev/out < /dev/in`
  - This only sets the priority of `get_data` to 127—the other processes inherit their scheduling params from the shell!
- Example:
  `% atprio 127 sh -c ğet_data | process_data > /dev/out < /dev/in¨`
  - All elements in the pipe will be set to priority 127

Keeping RT Applications in Check

Assume

- you are developing an RT application and let it run on your development platform
- you happen to accidentally create a process as part of that application that runs at the highest priority and performs an infinite loop

What happens?

It is generally a good idea to

- Keep a shell around that runs under the FIFO scheduler, at the highest priority
  - Can use for example the `atprio` program
- Refrain from using the highest priority in the program under development

Keeping RT Applications in Check

- `kill`, `atprio`, and a high-priority shell are usually sufficient to regain control of your machine
- However, they may not always be sufficient
- There are typically layers of processes between a busted program and you (madly typing CTRL-Cs . . .)
  - Operating System
  - X Server
  - Window Manager
  - Xterm
  - shell
- So you better know what you are doing . . . (as always)

POSIX Priority Inheritance

- POSIX also allows priority inheritance to be associated with mutexes (rarely implemented yet)
- This priority protected protocol corresponds to the Immediate Priority Ceiling Protocol discussed earlier

```
// Set and get the priority inheritance protocol
int pthread_mutexattr_setprotocol(pthread_mutexattr_t *ma,
                                  int                 protocol);
int pthread_mutexattr_getprotocol(pthread_mutexattr_t *ma,
                                  int                 protocol);

// Set and get the priority ceiling
int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *ma,
                                     int            prioceiling);
int pthread_mutexattr_getprioceiling(pthread_mutexattr_t *ma,
                                     int            prioceiling);
```

Other POSIX Facilities: POSIX also allows

- message queues to be priority ordered
- functions for dynamically getting and setting a thread's priority
- threads to indicate whether their attributes should be inherited by any child thread they create

Overview

Priority Inversion
- Process Blocking and Priority Inversion
- Priority Inheritance
- Calculating the blocking time
- Priority ceiling protocols
- Offline vs. Online scheduling

POSIX scheduling support
- Sporadic servers
- Inheritance of scheduling characteristics
- Keeping RT applications in check
- Priority inheritance

RT Java scheduling support
- The scheduler
- Avoiding priority inversion

RT Java Threads and Scheduling

There are two entities in Real-Time Java which can be scheduled:

- RealtimeThreads (and NoHeapRealtimeThread)
- AsyncEventHandler (and BoundAyncEventHandler)

Objects which are to be scheduled must

- implement the Schedulable interface
- specify their
  - SchedulingParameters
  - ReleaseParameters
  - MemoryParameters

Real-Time Java

- Supports at least 28 real-time priority levels
- As with Ada and POSIX, the larger the integer value, the higher the priority
- Non real-time threads are given priority levels below the minimum real-time priority
- Scheduling parameters are bound to threads at thread creation time; if the parameter objects are changed, they have an immediate impact on the associated thread
- Like Ada and RT POSIX, RT Java supports a pre-emptive priority-based dispatching policy
- Unlike Ada and RT POSIX, RT Java does not require a preempted thread to be placed at the head of the run queue associated with its priority level

The Schedulable Interface

```
public interface Schedulable extends java.lang.Runnable
{
  public void addToFeasibility();
  public void removeFromFeasibility();

  public MemoryParameters getMemoryParameters();
  public void setMemoryParameters(MemoryParameters memory);

  public ReleaseParameters getReleaseParameters();
  public void setReleaseParameters(ReleaseParameters release);

  public SchedulingParameters getSchedulingParameters();
  public void setSchedulingParameters(
          SchedulingParameters scheduling);

  public Scheduler getScheduler();
  public void setScheduler(Scheduler scheduler);
}
```

Scheduling Parameters

```
{
  public SchedulingParameters();
}

public class PriorityParameters extends SchedulingParameters
{
  public PriorityParameters(int priority);

  public int getPriority(); // At least 28 priority levels
  public void setPriority(int priority) throws IllegalArgumentException;
  ...
}

public class ImportanceParameters extends PriorityParameters
{
  public ImportanceParameters(int priority, int importance);
  public int getImportance();
  public void setImportance(int importance);
  ...
}
```

RT Java: The Scheduler

- RT Java supports a high-level scheduler whose goals are:
  - to decide whether to admit new schedulable objects according to the resources available and a feasibility algorithm, and
  - to set the priority of the schedulable objects according to the priority assignment algorithm associated with the feasibility algorithm
- For comparison:
  - Ada and POSIX assume a static off-line schedulability analysis
  - RT Java addresses more dynamic systems with the potential for on-line analysis

The Scheduler

- Abstract class Scheduler
- The isFeasible method considers only the set of schedulable objects that have been added to its feasibility list (via the addToFeasibility and removeFromFeasibility methods)
- The method changeIfFeasible checks to see if its set of objects is still feasible if the given object has its release and memory parameters changed
  - If it is, the parameters are changed
- Static methods allow the default scheduler to be queried or set
- RT Java does **not** require an implementation to provide an on-line feasibility algorithm

The Scheduler

```
public abstract class Scheduler
{
  public Scheduler();
  protected abstract void addToFeasibility(Schedulable s);
  protected abstract void removeFromFeasibility(Schedulable s);

  // Check the current set of schedulable objects
  public abstract boolean isFeasible();

  public boolean changeIfFeasible(Schedulable    schedulable,
                                  ReleaseParameters release,
                                  MemoryParameters memory);

  public static Scheduler getDefaultScheduler();
  public static void setDefaultScheduler(Scheduler scheduler);

  public abstract java.lang.String getPolicyName();
}
```

Avoidance of Priority Inversion in RT Java

- RT Java provides priority inheritance; to quote from [Bollela+ 2000]:
  Any conforming implementation must provide an implementation of the synchronized primitive with default behavior that ensures that there is *no unbounded priority inversion*. [. . .] The *priority inheritance protocol* must be implemented by default.
- RT Java also specifies IPCP, under a different name:
  A second policy, priority ceiling emulation protocol (or highest locker protocol), is also specified for systems that support it.

The Priority Scheduler

```
class PriorityScheduler extends Scheduler
{
  public PriorityScheduler()

  protected void addToFeasibility(Schedulable s);
  ...

  public void fireSchedulable(Schedulable schedulable);

  public int getMaxPriority();
  public int getMinPriority();
  public int getNormPriority();

  public static PriorityScheduler instance();
  ...
}
```

Standard preemptive priority-based scheduling

A Difficulty with Priority Inversion Avoidance

- What if both a `NoHeapRealTimeThread` and a regular Java thread synchronize on the same object ?
  - A `NoHeapRealTimeThread` object must have a priority (or, to be precise, execution eligibility) higher than the garbage collector
  - A regular Java thread may never have a priority higher than the garbage collector
  - This cannot be resolved by any classic priority inversion schemes
- Therefore, RT Java provides, in addition to the `synchronized` modifier, wait-free queued classes to provide protected, non-blocking, shared access
  - `WaitFreeDequeue()`
  - `WaitFreeReadQueue()`
  - `WaitFreeWriteQueue()`

Summary

- Ada, POSIX and RT Java
  - all support preemptive priority-based scheduling
  - also support the Immediate PCP
- Ada and POSIX focus on static off-line schedulability analysis
- RT Java addresses more dynamic systems with the potential for on-line analysis
- Memory locking avoids the detrimental effects that demand paging and swapping have on timing predictability
- When letting applications run on the development platform under a FIFO scheduler, care must be taken to avoid the process taking over the machine

To go further

- Chapter 13 of [Burns and Wellings 2001]
- [Liu 2000]
- Greg Bollela, Ben Brosgol, Steve Furr, David Hardin, Peter Dbble, James Gosling, Mark Turnbull, The Real-Time Specification for Java, Addison-Wesley, 2000 (`http://www.nist.gov/rt-java`)
- Chapter 5 of Gallmeister, POSIX.4: Programming for the Real World, O'Really, 1995
- What really happened on Mars? (`http://research.microsoft.com/~mbj/Mars_Pathfinder/`)