

Design and Implementation of POSIX Compliant Sporadic Server for the Linux Kernel ¹

D. Faggioli, A. Mancina, F. Checconi, G. Lipari
ReTiS Lab
Scuola Superiore Sant'Anna, CEIIC
via G. Moruzzi 1, 56124 Pisa (Italy)
{d.faggioli, a.mancina, f.checconi, g.lipari}@sssup.it

October 30, 2008

¹This work has been partially supported by the European projects
FRESCOR FP6/2005/IST/5-034026 and IRMOS FP7/2008/ICT/214777.

I am presenting a work that is still in progress:

- code is in highly development stage;
- deep testing is being and has to continue being performed;
- thorough overhead evaluation has to be undertaken.

... So why I am presenting it?

- We want to know what both the **research** and the **Linux kernel** community thinks about it.
- “Release early. Release often”.

I am presenting a work that is still in progress:

- code is in highly development stage;
- deep testing is being and has to continue being performed;
- through overhead evaluation has to be undertaken.

... So why I am presenting it?

- We want to know what both the **research** and the **Linux kernel** community thinks about it.
- “Release early. Release often”.

I am presenting a work that is still in progress:

- code is in highly development stage;
- deep testing is being and has to continue being performed;
- through overhead evaluation has to be undertaken.

... So why I am presenting it?

- We want to know what both the **research** and the **Linux kernel** community thinks about it.
- “Release early. Release often”.

I am presenting a work that is still in progress:

- code is in highly development stage;
- deep testing is being and has to continue being performed;
- thorough overhead evaluation has to be undertaken.

... So why I am presenting it?

- We want to know what both the **research** and the **Linux kernel** community thinks about it.
- “Release early. Release often”.

It is becoming common to integrate real-time support in the so-called general purpose operating systems:

- better exploitation of available resources;
- drop in the cost;
- existing software useable for non real-time activities (e.g., user interface).

Linux is the most suitable choice.

It is becoming common to integrate real-time support in the so-called general purpose operating systems:

- better exploitation of available resources;
- drop in the cost;
- existing software useable for non real-time activities (e.g., user interface).

Linux is the most suitable choice.

It is becoming common to integrate real-time support in the so-called general purpose operating systems:

- better exploitation of available resources;
- drop in the cost;
- existing software useable for non real-time activities (e.g., user interface).

Linux is the most suitable choice.

Why Linux?

Main motivations:

- it is Free Software;
- it has a huge and active development community;
- it has drivers for many of the exiting devices as well as wide availability for user-space libraries;
- it has billions of applications running on it, for each and every purpose;
- it provides standard programming interfaces (e.g., POSIX).

Furthermore, Linux has been, and is still being, enriched with more and more real-time related capabilities.

Why Linux?

Main motivations:

- it is Free Software;
- it has a huge and active development community;
- it has drivers for many of the exiting devices as well as wide availability for user-space libraries;
- it has billions of applications running on it, for each and every purpose;
- it provides standard programming interfaces (e.g., POSIX).

Furthermore, Linux has been, and is still being, enriched with more and more real-time related capabilities.

Linux “real-time variants”:

- *RTLinux* small real-time kernel which also runs an unmodified version of the Linux kernel at lowest priority;
- *RTAI/Xenomai* similar approach (e.g., by means of Adeos nanokernel) but cleaner and extensible interface;
- *LITMUS^{RT}* research framework for real-time algorithm evaluation and comparison.

rt – preempt patch:

- greatly reduces the non preemptable code sections;
- priority inheritance for in-kernel locking primitives;
- interrupt handling code is moved to schedulable threads;
- much more ...

Linux “real-time variants”:

- *RTLinux* small real-time kernel which also runs an unmodified version of the Linux kernel at lowest priority;
- *RTAI/Xenomai* similar approach (e.g., by means of Adeos nanokernel) but cleaner and extensible interface;
- *LITMUS^{RT}* research framework for real-time algorithm evaluation and comparison.

rt – preempt patch:

- greatly reduces the non preemptable code sections;
- priority inheritance for in-kernel locking primitives;
- interrupt handling code is moved to schedulable threads;
- much more ...

Anyway, also mainline Linux kernel offers notable real-time features:

- full preemption, i.e., low amount of latency introduced by the kernel;
- efficient locking policies implementations (e.g., preemptable RCU);
- high resolution timers for kernel and user space;
- Priority Inheritance and Priority Ceiling Emulation (via glibc) provided to user-space;
- support for POSIX real-time interface (POSIX.1b) **close to be complete.**

Linux and real-time (cont.)

Anyway, also mainline Linux kernel offers notable real-time features:

- full preemption, i.e., low amount of latency introduced by the kernel;
- efficient locking policies implementations (e.g., preemptable RCU);
- high resolution timers for kernel and user space;
- Priority Inheritance and Priority Ceiling Emulation (via glibc) provided to user-space;
- support for POSIX real-time interface (POSIX.1b) **close to be** complete.

Some needed features:

- temporal/bandwidth isolation for real-time activities;
- hierarchical scheduling.

Linux present support for these features:

- provides real-time bandwidth reservation but in **no** predictable and analyzable way;
- provides group scheduling, which is **different** from real-time hierarchical scheduling yet.

More on these issues later in the presentation ...

Some needed features:

- temporal/bandwidth isolation for real-time activities;
- hierarchical scheduling.

Linux present support for these features:

- provides real-time bandwidth reservation but in **no** predictable and analyzable way;
- provides group scheduling, which is **different** from real-time hierarchical scheduling yet.

More on these issues later in the presentation ...

Some needed features:

- temporal/bandwidth isolation for real-time activities;
- hierarchical scheduling.

Linux present support for these features:

- provides real-time bandwidth reservation but in **no** predictable and analyzable way;
- provides group scheduling, which is **different** from real-time hierarchical scheduling yet.

More on these issues later in the presentation ...

It was in 1998 that IEEE Std 1003.13-1998, the first POSIX real-time profile was published. More recently, a compliant X/Open System Interface implementation has to support at least:

- file synchronization,
- memory mapped files,
- memory protection,
- threads,
- thread synchronization,
- thread stack instrumentation;

POSIX and real-time (cont.)

and may also support:

Real-time sync., async and prioritized I/O, shared memory objects, memory locking, semaphores, timers, real-time signals, message passing, **process scheduling**;

Advanced Real-time clock selection, processes clocks, monotonic clock, timeouts;

Real-time Threads thread priority inheritance and protection, thread scheduling;

Advanced Real-time Threads thread clocks, thread **sporadic server**, spin locks and barriers.

SCHED_SPORADIC is optional part of *Real-time*, for processes, and *Advanced Real-time Threads*, for threads.

A recent Linux kernel already supports **almost all** the features required in a POSIX compliant real-time system, with the `SCHED_SPORADIC` scheduling policy being the biggest gap.

Real-time systems basics

A real-time system is a a set of activities with timing requirements. This *time consciousness* implies predictability is more important, for a real-time system, than throughput or any other performance metric.

There exist *hard* real-time systems and *soft* real-time systems:

Hard real-time systems completely fail if they not respect their constraints even once.

Soft real-time systems can tolerate some guarantee miss to happen, considering these events only temporary failures.

Hard real-time systems are used in critical activities (e.g., nuclear power plants, flight control systems, etc.), soft real-time systems typically are multimedia, entertainment or communication systems.

Real-time scheduling

In real-time scheduling each activity is usually called a *task* τ_i .
Each task:

- consists of a stream of jobs, $J_{i,j}$ ($j = 1, 2, \dots, n$);
- each job is characterized by:
 - an arrival time $a_{i,j}$,
 - an execution time $c_{i,j}$,
 - a finishing time $f_{i,j}$,
 - an absolute deadline $d_{i,j}$,
 - a relative deadline $D_{i,j} = d_{i,j} - a_{i,j}$.

A hard real-time task is also characterized by:

- a worst case execution time (WCET) $C_i = \max\{c_{i,j}\}$,
- a minimum inter-arrival time (MIT) of consecutive jobs,
 $T_i = \min\{a_{i,j+1} - a_{i,j}\}$.

In real-time scheduling each activity is usually called a *task* τ_i .
Each task:

- consists of a stream of jobs, $J_{i,j}$ ($j = 1, 2, \dots, n$);
- each job is characterized by:
 - an arrival time $a_{i,j}$,
 - an execution time $c_{i,j}$,
 - a finishing time $f_{i,j}$,
 - an absolute deadline $d_{i,j}$,
 - a relative deadline $D_{i,j} = d_{i,j} - a_{i,j}$.

A hard real-time task is also characterized by:

- a worst case execution time (*WCET*) $C_i = \max\{c_{i,j}\}$,
- a minimum inter-arrival time (*MIT*) of consecutive jobs,
 $T_i = \min\{a_{i,j+1} - a_{i,j}\}$.

Real-time scheduling (cont.)

A task is periodic if:

$$a_{i,j+1} = a_{i,j} + T_i$$

for any job $J_{i,j}$, otherwise it is called sporadic.

Real-time scheduling (cont.)

A task is periodic if:

$$a_{i,j+1} = a_{i,j} + T_i$$

for any job $J_{i,j}$, otherwise it is called sporadic.

Usually hard real-time tasks are periodic activities, while soft real-time ones issue requests of job activation in an aperiodic or sporadic fashion.

Real-time scheduling (cont.)

A task is periodic if:

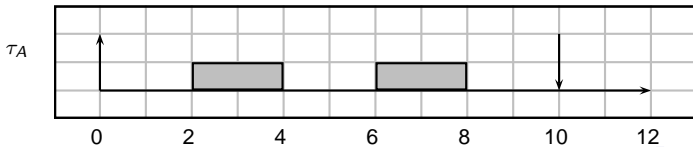
$$a_{i,j+1} = a_{i,j} + T_i$$

for any job $J_{i,j}$, otherwise it is called sporadic.

Usually hard real-time tasks are periodic activities, while soft real-time ones issue requests of job activation in an aperiodic or sporadic fashion.

Moreover, each task τ_i is assigned a processor utilization factor:

$$U_i = \frac{C_i}{T_i}$$



Fixed priority periodic real-time scheduling

In fixed priority algorithms the priority assigned to each task does not (automatically) vary along the task lifetime.

An example of an on-line fixed priority scheduling algorithm is the Rate Monotonic (*RM*) algorithm, where each task is given a priority p_i inversely proportional to its period:

$$p_i = \frac{1}{T_i}$$

Some property of RM algorithm:

- RM priority assignment is optimum;
- it is guaranteed that a feasible RM schedule exists if:

$$U = \sum_{i=0}^n U_i = \sum_{i=0}^n \frac{C_i}{T_i} \leq n \cdot (2^{\frac{1}{n}} - 1)$$

0.693 ($\ln(2)$) as n approaches infinity), but this is only sufficient and quite pessimistic condition.

Fixed priority periodic real-time scheduling

In fixed priority algorithms the priority assigned to each task does not (automatically) vary along the task lifetime.

An example of an on-line fixed priority scheduling algorithm is the Rate Monotonic (*RM*) algorithm, where each task is given a priority p_i inversely proportional to its period:

$$p_i = \frac{1}{T_i}$$

Some property of RM algorithm:

- RM priority assignment is optimum;
- it is guaranteed that a feasible RM schedule exists if:

$$U = \sum_{i=0}^n U_i = \sum_{i=0}^n \frac{C_i}{T_i} \leq n \cdot (2^{\frac{1}{n}} - 1)$$

0.693 ($\ln(2)$) as n approaches infinity), but this is only sufficient and quite pessimistic condition.

Fixed priority aperiodic real-time scheduling

We are interested in fixed priority scheduling of mixed periodic and aperiodic tasks so that:

- 1 aperiodic/sporadic activities are not jeopardizing the guarantees provided to periodic tasks;
- 2 execute aperiodic/sporadic activities with fast response time, either for each instance and on average.

Idea is to create an *aperiodic server*, that gives the aperiodic requests a chance to run:

- it is one more task with period T_s and a *budget* Q_s ;
- a request to run from a sporadic task is satisfied only when the server is scheduled;
- as its execution proceeds, the budget is diminished by the same amount, until it reaches zero;
- an aperiodic server with zero budget is no longer considered a ready to run.

Fixed priority aperiodic real-time scheduling

We are interested in fixed priority scheduling of mixed periodic and aperiodic tasks so that:

- 1 aperiodic/sporadic activities are not jeopardizing the guarantees provided to periodic tasks;
- 2 execute aperiodic/sporadic activities with fast response time, either for each instance and on average.

Idea is to create an *aperiodic server*, that gives the aperiodic requests a chance to run:

- it is one more task with period T_s and a *budget* Q_s ;
- a request to run from a sporadic task is satisfied only when the server is scheduled;
- as its execution proceeds, the budget is diminished by the same amount, until it reaches zero;
- an aperiodic server with zero budget is no longer considered a ready to run.

Aperiodic server strategies

The strategy of budget replenishing is what characterizes the different aperiodic server algorithms:

Background Server only execute aperiodic tasks *in background*. It is very simple but may provide poor aperiodic task response time.

Polling Server it starts at regular intervals and services pending aperiodic requests if any, otherwise it suspends. Main drawback is aperiodic response times can still be quite long.

Deferrable Server similar to Polling Server, but it preserves its budget even if no requests are pending. It provides better aperiodic responsiveness, at the price of a schedulability penalty.

Sporadic Server variation of Deferrable Server so that it replenishes its budget after some of its execution time is consumed. Offers **comparable performance with DS without schedulability loss.**

Aperiodic server strategies

The strategy of budget replenishing is what characterizes the different aperiodic server algorithms:

Background Server only execute aperiodic tasks *in background*. It is very simple but may provide poor aperiodic task response time.

Polling Server it starts at regular intervals and services pending aperiodic requests if any, otherwise it suspends. Main drawback is aperiodic response times can still be quite long.

Deferrable Server similar to Polling Server, but it preserves its budget even if no requests are pending. It provides better aperiodic responsiveness, at the price of a schedulability penalty.

Sporadic Server variation of Deferrable Server so that it replenishes its budget after some of its execution time is consumed. Offers **comparable performance** with DS **without schedulability loss**.

Aperiodic server strategies

The strategy of budget replenishing is what characterizes the different aperiodic server algorithms:

Background Server only execute aperiodic tasks *in background*. It is very simple but may provide poor aperiodic task response time.

Polling Server it starts at regular intervals and services pending aperiodic requests if any, otherwise it suspends. Main drawback is aperiodic response times can still be quite long.

Deferrable Server similar to Polling Server, but it preserves its budget even if no requests are pending. It provides better aperiodic responsiveness, at the price of a schedulability penalty.

Sporadic Server variation of Deferrable Server so that it replenishes its budget after some of its execution time is consumed. Offers **comparable performance** with DS **without schedulability loss**.

Aperiodic server strategies

The strategy of budget replenishing is what characterizes the different aperiodic server algorithms:

Background Server only execute aperiodic tasks *in background*. It is very simple but may provide poor aperiodic task response time.

Polling Server it starts at regular intervals and services pending aperiodic requests if any, otherwise it suspends. Main drawback is aperiodic response times can still be quite long.

Deferrable Server similar to Polling Server, but it preserves its budget even if no requests are pending. It provides better aperiodic responsiveness, at the price of a schedulability penalty.

Sporadic Server variation of Deferrable Server so that it replenishes its budget after some of its execution time is consumed. Offers **comparable performance** with DS **without schedulability loss**.

With **Deferrable Server** aperiodic tasks or requests can be served anytime as long as the budget itself has not been exhausted:

- provides fast response time
- easy to implement
- if a DS U_s the periodic load for RM has to stay below:

$$\ln\left(\frac{U_s + 2}{2 \cdot U_s + 1}\right)$$

which is **worse** than the original.

With **Deferrable Server** aperiodic tasks or requests can be served anytime as long as the budget itself has not been exhausted:

- provides fast response time
- easy to implement
- if a DS U_s the periodic load for RM has to stay below:

$$\ln\left(\frac{U_s + 2}{2 \cdot U_s + 1}\right)$$

which is **worse** than the original.

When using **Sporadic Server**, its particular method of scheduling replenishments sets it apart from other algorithms.

For a Sporadic Server task $U_{SS} = \frac{Q_{SS}}{T_{SS}}$ with priority p_{SS} :

- 1 if the server has budget and the running task has is greater priority than p_{SS} , replenishment time RT_{SS} is set;
- 2 when the priority of the running task becomes lower than p_{SS} , or the budget is exhausted, amount of the next replenishment is decided as the time consumed from the last instant when priority of the running task was higher than p_{SS} .

When using **Sporadic Server**, its particular method of scheduling replenishments sets it apart from other algorithms.

For a Sporadic Server task $U_{SS} = \frac{Q_{SS}}{T_{SS}}$ with priority p_{SS} :

- 1 if the server has budget and the running task has is greater priority than p_{SS} , replenishment time RT_{SS} is set;
- 2 when the priority of the running task becomes lower than p_{SS} , or the budget is exhausted, amount of the next replenishment is decided as the time consumed from the last instant when priority of the running task was higher than p_{SS} .

When using **Sporadic Server**, its particular method of scheduling replenishments sets it apart from other algorithms.

For a Sporadic Server task $U_{SS} = \frac{Q_{SS}}{T_{SS}}$ with priority p_{SS} :

- 1 if the server has budget and the running task has is greater priority than p_{SS} , replenishment time RT_{SS} is set;
- 2 when the priority of the running task becomes lower than p_{SS} , or the budget is exhausted, amount of the next replenishment is decided as the time consumed from the last instant when priority of the running task was higher than p_{SS} .

Sporadic Server details (cont.)

Main advantages are:

- fast response time, as DS;
- no schedulability loss, U_{lub} is the same than in original RM.

Furthermore:

- can be used to implement hierarchical scheduling: a Sporadic Server may contain more than one task and also other Sporadic Servers;
- enforces temporal isolation among tasks/servers:
 - misbehaviors of a task inside a Sporadic Server do not affect other tasks outside of it;
 - misbehaviors of a task outside a Sporadic Server, do not affect tasks inside it.

Sporadic Server details (cont.)

Main advantages are:

- fast response time, as DS;
- no schedulability loss, U_{lub} is the same than in original RM.

Furthermore:

- can be used to implement hierarchical scheduling: a Sporadic Server may contain more than one task and also other Sporadic Servers;
- enforces temporal isolation among tasks/servers:
 - misbehaviors of a task inside a Sporadic Server do not affect other tasks outside of it;
 - misbehaviors of a task outside a Sporadic Server, do not affect tasks inside it.

Sporadic Server details (cont.)

Main advantages are:

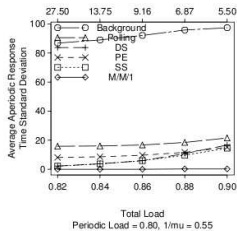
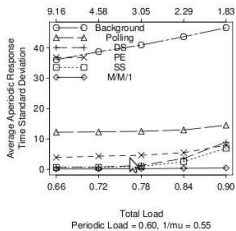
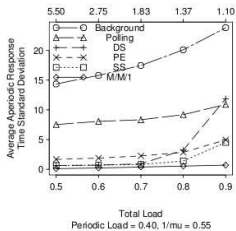
- fast response time, as DS;
- no schedulability loss, U_{lub} is the same than in original RM.

Furthermore:

- can be used to implement hierarchical scheduling: a Sporadic Server may contain more than one task and also other Sporadic Servers;
- enforces temporal isolation among tasks/servers:
 - misbehaviors of a task inside a Sporadic Server do not affect other tasks outside of it;
 - misbehaviors of a task outside a Sporadic Server, do not affect tasks inside it.

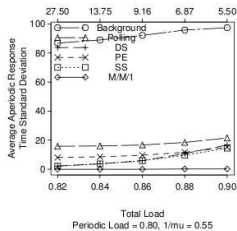
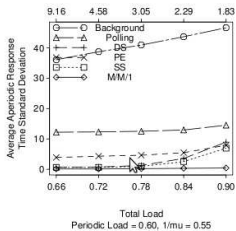
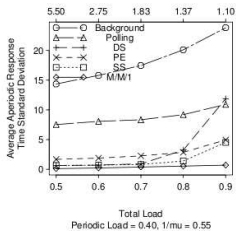
Aperiodic servers comparisons

Response time:

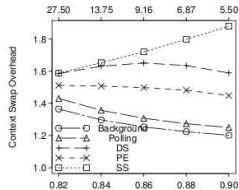
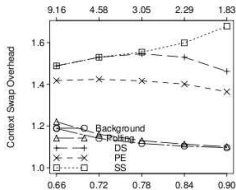
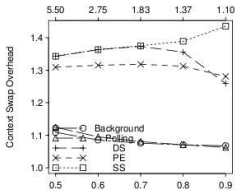


Aperiodic servers comparisons

Response time:



Context switch overhead:



POSIX real-time scheduling

POSIX standard specifies *Fixed Priority Preemptive Scheduling*, by means of three possible policies:

- SCHED_FIFO
- SCHED_RR
- SCHED_SPORADIC

In some more detail:

SCHED_FIFO the task with highest priority is able to run undisturbed unless it blocks or voluntarily relinquish the CPU. Tasks with the same priority are ordered by their arrival time.

SCHED_RR similar to SCHED_FIFO, but tasks have a time quantum.

When reached, they are re-queued, guaranteeing that other tasks at the same priority have a chance to run.

POSIX real-time scheduling

POSIX standard specifies *Fixed Priority Preemptive Scheduling*, by means of three possible policies:

- `SCHED_FIFO`
- `SCHED_RR`
- `SCHED_SPORADIC`

In some more detail:

SCHED_FIFO the task with highest priority is able to run undisturbed unless it blocks or voluntarily relinquish the CPU. Tasks with the same priority are ordered by their arrival time.

SCHED_RR similar to `SCHED_FIFO`, but tasks have a time quantum.
When reached, they are re-queued, guaranteeing that other tasks at the same priority have a chance to run.

`SCHED_SPORADIC` it needs more scheduling parameters than a single priority value:

- `sched_priority` (also used in `SCHED_{FIFO,RR}`),
- `sched_ss_low_priority`,
- `sched_ss_init_budget`,
- `sched_ss_repl_period`,
- `sched_ss_low_priority`.

It is very close to `SCHED_FIFO`, provided the actual priority of a task is:

`sched_priority` if the current budget is greater than zero and the number of pending replenishment is less than `sched_ss_max_repl`;
`sched_ss_low_priority` otherwise.

`SCHED_SPORADIC` it needs more scheduling parameters than a single priority value:

- `sched_priority` (also used in `SCHED_{FIFO,RR}`),
- `sched_ss_low_priority`,
- `sched_ss_init_budget`,
- `sched_ss_repl_period`,
- `sched_ss_low_priority`.

It is very close to `SCHED_FIFO`, provided the actual priority of a task is:

`sched_priority` if the current budget is greater than zero and the number of pending replenishment is less than `sched_ss_max_repl`;

`sched_ss_low_priority` otherwise.

POSIX SCHED_SPORADIC scheduling policy (cont.)

SCHED_SPORADIC Modification of the current budget of a task is done as follows, for a task having `sched_priority`:

- 1 time at which it becomes ready at is posted as `activation_time`;
- 2 while it runs, its budget is diminished by the same amount;
- 3 when it blocks a replenishment is scheduled;
- 4 when it exhausts the budget, replenishment is scheduled, and the task become `sched_ss_low_priority`;
- 5 each replenishment is scheduled to happen at `activation_time` plus `sched_ss_repl_period`, and to replenish the amount of budget consumed since `activation_time`;
- 6 replenishment consists of adding the replenish amount to current budget. If it was running at `sched_ss_low_priority`, then it suddenly becomes `sched_priority`.

POSIX SCHED_SPORADIC scheduling policy (cont.)

SCHED_SPORADIC Modification of the current budget of a task is done as follows, for a task having `sched_priority`:

- 1 time at which it becomes ready at is posted as `activation_time`;
- 2 while it runs, its budget is diminished by the same amount;
- 3 when it blocks a replenishment is scheduled;
- 4 when it exhausts the budget, replenishment is scheduled, and the task become `sched_ss_low_priority`;
- 5 each replenishment is scheduled to happen at `activation_time` plus `sched_ss_repl_period`, and to replenish the amount of budget consumed since `activation_time`;
- 6 replenishment consists of adding the replenish amount to current budget. If it was running at `sched_ss_low_priority`, then it suddenly becomes `sched_priority`.

POSIX SCHED_SPORADIC scheduling policy (cont.)

SCHED_SPORADIC Modification of the current budget of a task is done as follows, for a task having `sched_priority`:

- 1 time at which it becomes ready at is posted as `activation_time`;
- 2 while it runs, its budget is diminished by the same amount;
- 3 when it blocks a replenishment is scheduled;
- 4 when it exhausts the budget, replenishment is scheduled, and the task become `sched_ss_low_priority`;
- 5 each replenishment is scheduled to happen at `activation_time` plus `sched_ss_repl_period`, and to replenish the amount of budget consumed since `activation_time`;
- 6 replenishment consists of adding the replenish amount to current budget. If it was running at `sched_ss_low_priority`, then it suddenly becomes `sched_priority`.

POSIX SCHED_SPORADIC scheduling policy (cont.)

SCHED_SPORADIC Modification of the current budget of a task is done as follows, for a task having `sched_priority`:

- 1 time at which it becomes ready at is posted as `activation_time`;
- 2 while it runs, its budget is diminished by the same amount;
- 3 when it blocks a replenishment is scheduled;
- 4 when it exhausts the budget, replenishment is scheduled, and the task become `sched_ss_low_priority`;
- 5 each replenishment is scheduled to happen at `activation_time` plus `sched_ss_repl_period`, and to replenish the amount of budget consumed since `activation_time`;
- 6 replenishment consists of adding the replenish amount to current budget. If it was running at `sched_ss_low_priority`, then it suddenly becomes `sched_priority`.

POSIX SCHED_SPORADIC scheduling policy (cont.)

SCHED_SPORADIC Modification of the current budget of a task is done as follows, for a task having `sched_priority`:

- 1 time at which it becomes ready at is posted as `activation_time`;
- 2 while it runs, its budget is diminished by the same amount;
- 3 when it blocks a replenishment is scheduled;
- 4 when it exhausts the budget, replenishment is scheduled, and the task become `sched_ss_low_priority`;
- 5 each replenishment is scheduled to happen at `activation_time` plus `sched_ss_repl_period`, and to replenish the amount of budget consumed since `activation_time`;
- 6 replenishment consists of adding the replenish amount to current budget. If it was running at `sched_ss_low_priority`, then it suddenly becomes `sched_priority`.

POSIX SCHED_SPORADIC scheduling policy (cont.)

SCHED_SPORADIC Modification of the current budget of a task is done as follows, for a task having `sched_priority`:

- 1 time at which it becomes ready at is posted as `activation_time`;
- 2 while it runs, its budget is diminished by the same amount;
- 3 when it blocks a replenishment is scheduled;
- 4 when it exhausts the budget, replenishment is scheduled, and the task become `sched_ss_low_priority`;
- 5 each replenishment is scheduled to happen at `activation_time` plus `sched_ss_repl_period`, and to replenish the amount of budget consumed since `activation_time`;
- 6 replenishment consists of adding the replenish amount to current budget. If it was running at `sched_ss_low_priority`, then it suddenly becomes `sched_priority`.

POSIX SCHED_SPORADIC scheduling policy (cont.)

SCHED_SPORADIC Modification of the current budget of a task is done as follows, for a task having `sched_priority`:

- 1 time at which it becomes ready at is posted as `activation_time`;
- 2 while it runs, its budget is diminished by the same amount;
- 3 when it blocks a replenishment is scheduled;
- 4 when it exhausts the budget, replenishment is scheduled, and the task become `sched_ss_low_priority`;
- 5 each replenishment is scheduled to happen at `activation_time` plus `sched_ss_repl_period`, and to replenish the amount of budget consumed since `activation_time`;
- 6 replenishment consists of adding the replenish amount to current budget. If it was running at `sched_ss_low_priority`, then it suddenly becomes `sched_priority`.

POSIX SCHED_SPORADIC policy for group soft tasks

Linux currently support real-time group scheduling.

Applying SCHED_SPORADIC scheduling policy also to groups means:

- 1 define how budget is accounted;
- 2 define when a group is “ready”;
- 3 define when a group “blocks”.

Our proposal:

- 1 when a task executes the budget of all the groups it is inserted in is diminished;
- 2 a task group is ready (unblocked) when it has some task (or groups) to execute in its runqueue;
- 3 a task group is blocked when it has no more tasks (or groups) to execute in its runqueue;

POSIX SCHED_SPORADIC policy for group soft tasks

Linux currently support real-time group scheduling.

Applying SCHED_SPORADIC scheduling policy also to groups means:

- 1 define how budget is accounted;
- 2 define when a group is “ready”;
- 3 define when a group “blocks”.

Our proposal:

- 1 when a task executes the budget of all the groups it is inserted in is diminished;
- 2 a task group is ready (unblocked) when it has some task (or groups) to execute in its runqueue;
- 3 a task group is blocked when it has no more tasks (or groups) to execute in its runqueue;

POSIX SCHED_SPORADIC policy for group soft tasks

Linux currently support real-time group scheduling.

Applying SCHED_SPORADIC scheduling policy also to groups means:

- 1 define how budget is accounted;
- 2 define when a group is “ready”;
- 3 define when a group “blocks”.

Our proposal:

- 1 when a task executes the budget of all the groups it is inserted in is diminished;
- 2 a task group is ready (unblocked) when it has some task (or groups) to execute in its runqueue;
- 3 a task group is blocked when it has no more tasks (or groups) to execute in its runqueue;

POSIX SCHED_SPORADIC policy for group soft tasks

Linux currently support real-time group scheduling.

Applying SCHED_SPORADIC scheduling policy also to groups means:

- 1 define how budget is accounted;
- 2 define when a group is “ready”;
- 3 define when a group “blocks”.

Our proposal:

- 1 when a task executes the budget of all the groups it is inserted in is diminished;
- 2 a task group is ready (unblocked) when it has some task (or groups) to execute in its runqueue;
- 3 a task group is blocked when it has no more tasks (or groups) to execute in its runqueue;

POSIX SCHED_SPORADIC policy for group soft tasks

Linux currently support real-time group scheduling.

Applying SCHED_SPORADIC scheduling policy also to groups means:

- 1 define how budget is accounted;
- 2 define when a group is “ready”;
- 3 define when a group “blocks”.

Our proposal:

- 1 when a task executes the budget of all the groups it is inserted in is diminished;
- 2 a task group is ready (unblocked) when it has some task (or groups) to execute in its runqueue;
- 3 a task group is blocked when it has no more tasks (or groups) to execute in its runqueue;

Only these two:

QNX Neutrino Microkernel POSIX-compliant RTOS aimed primarily at the embedded system market, especially automotive.

Real-Time Executive for Multiprocessor Systems (RTEMS) hard real-time executive for embedded critical applications.

Other attempts to implement SCHED_SPORADIC

Related work in Linux and its variants also exists:

- some work on implementing the policy in RTLinux (research paper and master thesis), which, anyway, is **different** than Linux;
- an implementation (master thesis) of the policy in Linux 2.6.25.

Main differences with our work:

- we are the only one targeting not only task but also group scheduling;
- we are the only one tracking the current development of the kernel and not basing our patches on some specific (old) version;
- we are the only one that have submitted preliminary version of the code to the community on the Linux Kernel Mailing List;
- we are writing code that could be applied both to *mainline* and *rt – preempt* Linux.

Other attempts to implement SCHED_SPORADIC

Related work in Linux and its variants also exists:

- some work on implementing the policy in RTLinux (research paper and master thesis), which, anyway, is **different** than Linux;
- an implementation (master thesis) of the policy in Linux 2.6.25.

Main differences with our work:

- we are the only one targeting not only task but also group scheduling;
- we are the only one tracking the current development of the kernel and not basing our patches on some specific (old) version;
- we are the only one that have submitted preliminary version of the code to the community on the Linux Kernel Mailing List;
- we are writing code that could be applied both to *mainline* and *rt – preempt* Linux.

Benefits of having SCHED_SPORADIC task scheduling in mainline Linux kernel

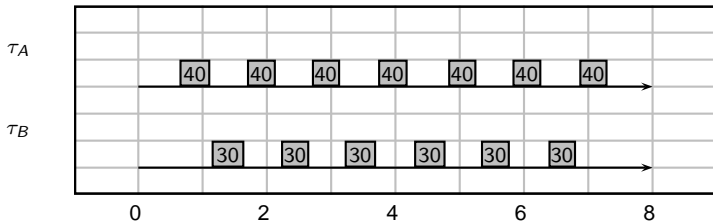
- improved conformance with the POSIX standard and real-time extensions;

Benefits of having SCHED_SPORADIC task scheduling in mainline Linux kernel

- improved conformance with the POSIX standard and real-time extensions;
- match with OSs that already has it implemented, as well as plus feature against the ones that has not;

Benefits of having SCHED_SPORADIC task scheduling in mainline Linux kernel

- improved conformance with the POSIX standard and real-time extensions;
- match with OSs that already has it implemented, as well as plus feature against the ones that has not;
- better control of scheduling of real-time tasks than with only SCHED_FIFO and SCHED_RR.



Benefits of having SCHED_SPORADIC group scheduling in mainline Linux kernel

Linux already provides group scheduling with bandwidth reservation (called throttling), but:

- the algorithm used is not a real-time theoretically funded one;
- the algorithm used, although being somehow different, resembles the Deferrable Server, so:
 - it suffers of schedulability loss;
 - it suffers of poor composability (due to schedulability loss).

SCHED_SPORADIC group scheduling, instead:

- guarantee full schedulability;
- could really help in turning Linux into a more predictable and fully analyzable system

Benefits of having SCHED_SPORADIC group scheduling in mainline Linux kernel

Linux already provides group scheduling with bandwidth reservation (called throttling), but:

- the algorithm used is not a real-time theoretically funded one;
- the algorithm used, although being somehow different, resembles the Deferrable Server, so:
 - it suffers of schedulability loss;
 - it suffers of poor composability (due to schedulability loss).

SCHED_SPORADIC group scheduling, instead:

- guarantee full schedulability;
- could really help in turning Linux into a more predictable and fully analyzable system

Benefits of having SCHED_SPORADIC group scheduling in mainline Linux kernel

Linux already provides group scheduling with bandwidth reservation (called throttling), but:

- the algorithm used is not a real-time theoretically funded one;
- the algorithm used, although being somehow different, resembles the Deferrable Server, so:
 - it suffers of schedulability loss;
 - it suffers of poor composability (due to schedulability loss).

SCHED_SPORADIC group scheduling, instead:

- guarantee full schedulability;
- could really help in turning Linux into a more predictable and fully analyzable system

Benefits of having SCHED_SPORADIC group scheduling in mainline Linux kernel

Linux already provides group scheduling with bandwidth reservation (called throttling), but:

- the algorithm used is not a real-time theoretically funded one;
- the algorithm used, although being somehow different, resembles the Deferrable Server, so:
 - it suffers of schedulability loss;
 - it suffers of poor composability (due to schedulability loss).

SCHED_SPORADIC group scheduling, instead:

- guarantee full schedulability;
- could really help in turning Linux into a more predictable and fully analyzable system

Benefits of having SCHED_SPORADIC group scheduling in mainline Linux kernel

Linux already provides group scheduling with bandwidth reservation (called throttling), but:

- the algorithm used is not a real-time theoretically funded one;
- the algorithm used, although being somehow different, resembles the Deferrable Server, so:
 - it suffers of schedulability loss;
 - it suffers of poor composability (due to schedulability loss).

SCHED_SPORADIC group scheduling, instead:

- guarantee full schedulability;
- could really help in turning Linux into a more predictable and fully analyzable system

Benefits of having SCHED_SPORADIC group scheduling in mainline Linux kernel (cont.)

Unfortunately, SCHED_SPORADIC has also some drawbacks, with respect to the actual throttling code:

- it increases the memory overhead due to the need of maintaining the queue of pending replenishments;
- it is quite sensitive to imprecise budget accounting, which could be serious issue on some system;
- it is more difficult to implement, to understand, to debug and maintain, and so on.

Benefits of having SCHED_SPORADIC group scheduling in mainline Linux kernel (cont.)

Unfortunately, SCHED_SPORADIC has also some drawbacks, with respect to the actual throttling code:

- it increases the memory overhead due to the need of maintaining the queue of pending replenishments;
- it is quite sensitive to imprecise budget accounting, which could be serious issue on some system;
- it is more difficult to implement, to understand, to debug and maintain, and so on.

Benefits of having SCHED_SPORADIC group scheduling in mainline Linux kernel (cont.)

Unfortunately, SCHED_SPORADIC has also some drawbacks, with respect to the actual throttling code:

- it increases the memory overhead due to the need of maintaining the queue of pending replenishments;
- it is quite sensitive to imprecise budget accounting, which could be serious issue on some system;
- it is more difficult to implement, to understand, to debug and maintain, and so on.

Benefits of having SCHED_SPORADIC group scheduling in mainline Linux kernel (cont.)

Unfortunately, SCHED_SPORADIC has also some drawbacks, with respect to the actual throttling code:

- it increases the memory overhead due to the need of maintaining the queue of pending replenishments;
- it is quite sensitive to imprecise budget accounting, which could be serious issue on some system;
- it is more difficult to implement, to understand, to debug and maintain, and so on.

Some implementation details

Implemented inside the `sched_rt.c` file (real-time scheduling class), together with `SCHED_FIFO` and `SCHED_RR`.

Implemented using only one high resolution timer for each task or task group, exactly as throttling.

Compile time switches are provided in order to make it possible to:

- choose if compile the support for `SCHED_SPORADIC` or not;
- set the maximum dimension of the pending replenishment queue.

Some implementation details

Implemented inside the `sched_rt.c` file (real-time scheduling class), together with `SCHED_FIFO` and `SCHED_RR`.

Implemented using only one high resolution timer for each task or task group, exactly as throttling.

Compile time switches are provided in order to make it possible to:

- choose if compile the support for `SCHED_SPORADIC` or not;
- set the maximum dimension of the pending replenishment queue.

Some implementation details

Implemented inside the `sched_rt.c` file (real-time scheduling class), together with `SCHED_FIFO` and `SCHED_RR`.

Implemented using only one high resolution timer for each task or task group, exactly as throttling.

Compile time switches are provided in order to make it possible to:

- choose if compile the support for `SCHED_SPORADIC` or not;
- set the maximum dimension of the pending replenishment queue.

Some implementation details (cont.)

diffstat of one of the last patches:

arch/x86/kernel/ syscall_table_32 .S | 3

include/asm-x86/unistd_32.h | 4

include/asm-x86/unistd_64.h | 7

include/linux/sched.h | 79

init /Kconfig | 42

kernel/exit.c | 5

kernel/fork.c | 15

kernel/sched.c | 451

kernel/sched_rt.c | 542

9 files changed, 1130 insertions (+), 18 deletions (-)

A lot of things are not finished, or still missing, among them:

- release the code again in LKML and get comments;
- thorough testing of the code both on Linux and *rt – preempt*, especially for SMP systems;
- proving the effectiveness of bandwidth isolation with `SCHED_SPORADIC` not only with synthetic examples, but also with a real application;
- enhancing the Priority Inheritance mechanism to take into account the bandwidth of tasks and task groups (Bandwidth Inheritance);
- enhancing group scheduling code to make it possible to specify real-time priorities for task groups.