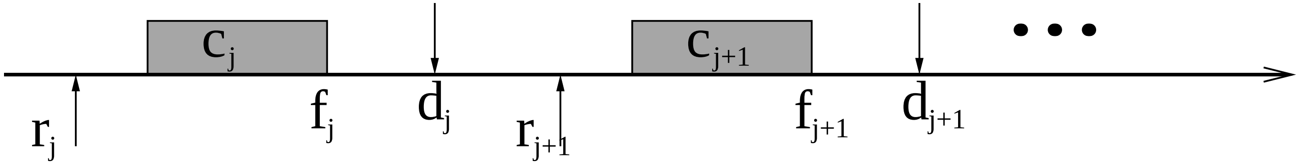
SCHED_DEADLINE: a real-time CPU scheduler for Linux

Luca Abeni
luca.abeni@santannapisa.it

Scheduling Real-Time Tasks



- Consider a set of N real-time tasks $\Gamma = \{\tau_0, ... \tau_{N-1}\}$
- Scheduled on M CPUs
- Real-Time theory → lot of scheduling algorithms...
- ...But which ones are available on a commonly used OS?
- POSIX: fixed priorities
  - ☐ Can be used to do RM, DM, etc...
  - ☐ Multiple processors: DkC, etc...
- Linux also provides `SCHED_DEADLINE`: resource reservations + EDF

Definitions

- Real-time task τ: sequence of jobs $J_i = (r_i, c_i, d_i)$
  - Finishing time f_i
  - Goal: $f_i \leq d_i$
  - $\forall J_i$, or control the amount of missed deadlines
- Schedule on multiple CPUS: partitioned or global
- Schedule in a general-purpose OS
  - Open System (with online admission control)
  - Presence of non real-time tasks (do not starve them!)

Using Fixed Priorities with POSIX

- `SCHED_FIFO` and `SCHED_RR` use fixed priorities
  - They can be used for real-time tasks, to implement RM and DM
  - Real-time tasks have priority over non real-time (`SCHED_OTHER`) tasks
- The difference between the two policies is visible when more tasks have the same priority
  - In real-time applications, try to avoid multiple tasks with the same priority

Setting the Scheduling Policy

```
int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);

int sched_setscheduler(pid_t pid, int policy,
                       const struct sched_param *param);
int sched_setparam(pid_t pid,
                   const struct sched_param *param);
```

- If `pid == 0`, then the parameters of the running task are changed
- The only meaningful field of `struct sched_param` is `sched_priority`

Issues with Real-Time Priorities

- Open Systems → real-time tasks can dynamically arrive (in an unpredictable way)
    - ☐ Need to re-arrange priorities to respect RM / DM / ...
- Interactions with non real-time tasks?
    - ☐ Scheduled in background respect to real-time tasks
- Suboptimal utilization?

Real-Time Priorities vs “Regular Tasks”

- In general, “regular” (`SCHED_OTHER`) tasks are scheduled in background respect to real-time ones
- Real-time tasks can starve other applications
- Example: the following task scheduled at high priority can make a CPU / core unusable

```
void bad_bad_task()
{
        while(1);
}
```

Starvation of Non Real-Time Tasks

- Starvation of non real-time tasks
  - Real-time computation have to be limited (use real-time priorities only when **really needed**!)
- On sane systems, running applications with real-time priorities requires root privileges (or part of them!)
  - Not usable by everyone

Real-Time Throttling

- A “bad” high-priority task can make a CPU / core unusable...
- ...Linux provides the *real-time throttling* mechanism to address this problem
    - How does real-time throttling interfere with real-time guarantees?
    - Given a priority assignment, a taskset is guaranteed all the deadlines if no throttling mechanism is used...
    - ...But, what happens in case of throttling?
- Very useful idea, but something more “theoretically founded” might be needed...

Can We Do Better?

- Avoid starvation issues by using resource reservations
- Use EDF instead of fixed priorities
  - CPU Reservations + EDF = `SCHED_DEADLINE`!!!
- So, how to implement EDF (or something similar) in Linux?
  - Issue: the kernel is (was?) not aware of tasks deadlines...
  - ...But deadlines are needed in order to schedule the tasks!

EDF in the Linux Kernel

- EDF assigns dynamic priorities based on absolute deadlines
- So, a more advanced API for the scheduler is needed...
  - Assign at least a relative deadline D to the task...
  - We will see that we need a *runtime* and a *period* too
- Moreover, $d_j = r_j + D$...
  - ...However, how can the scheduler know r_j?
  - The scheduler is not aware of jobs...

Tasks, and Jobs...

- EDF → need to know when a job starts / finishes
  - Applications must be modified to signal the beginning / end of a job (some kind of `startjob()` / `endjob()` system call)...
  - ...Or the scheduler can assume that a new job arrives each time a task wakes up!
- Or, some other algorithm can be used to assign dynamic *scheduling deadlines* to tasks

...And Scheduling Deadlines!

- The scheduler does EDF on scheduling deadlines
  - Scheduling deadline d^s: assigned by the kernel to task τ
- But the task cares about its absolute deadlines
  - If the scheduling deadline d^s matches the absolute deadline d_j of a job, then the scheduler can respect d_j!!!

CBS: The Basic Idea

- Constant Bandwidth Server (CBS): algorithm used to assign a dynamic scheduling deadline d^s to a task τ
- Based on the *Resource Reservation* paradigm
  - Task τ is periodically reserved a *maximum runtime* Q every *reservation period* P
- Temporal isolation between tasks
  - The worst case finishing time for a task does not depend on the other tasks running in the system...
  - ...Because the task is guaranteed to receive its reserved time

CBS: Some More Details

- Solves the issue with “bad tasks” trying to consume too much execution time
- Based on CPU reservations (Q, P)
  - If τ tries to execute for more than Q every P, the algorithm decreases its priority, or throttles it
  - τ consumes the same amount of CPU time consumed by a periodic task with WCET Q and period P
- Q/P: fraction of CPU time reserved to τ

CBS: Admission Control

- The CBS is based on EDF
  - ☐ Assigns scheduling deadlines d^s
  - ☐ EDF on $d^s \Rightarrow$ good CPU utilization (optimal on UP!)
- If EDF is used (based on the scheduling deadlines assigned by the CBS), then τ_i is guaranteed to receive Q_i time units every P_i if $\sum_j Q_j/P_j \leq 1$!!!
  - ☐ Only on uni-processor / partitioned systems...
  - ☐ M CPUs / cores with global scheduling: if $\sum_j Q_j/P_j \leq M$ each task is guaranteed to receive Q_i every P_i with a maximum delay

CBS vs Other Reservation Algorithms

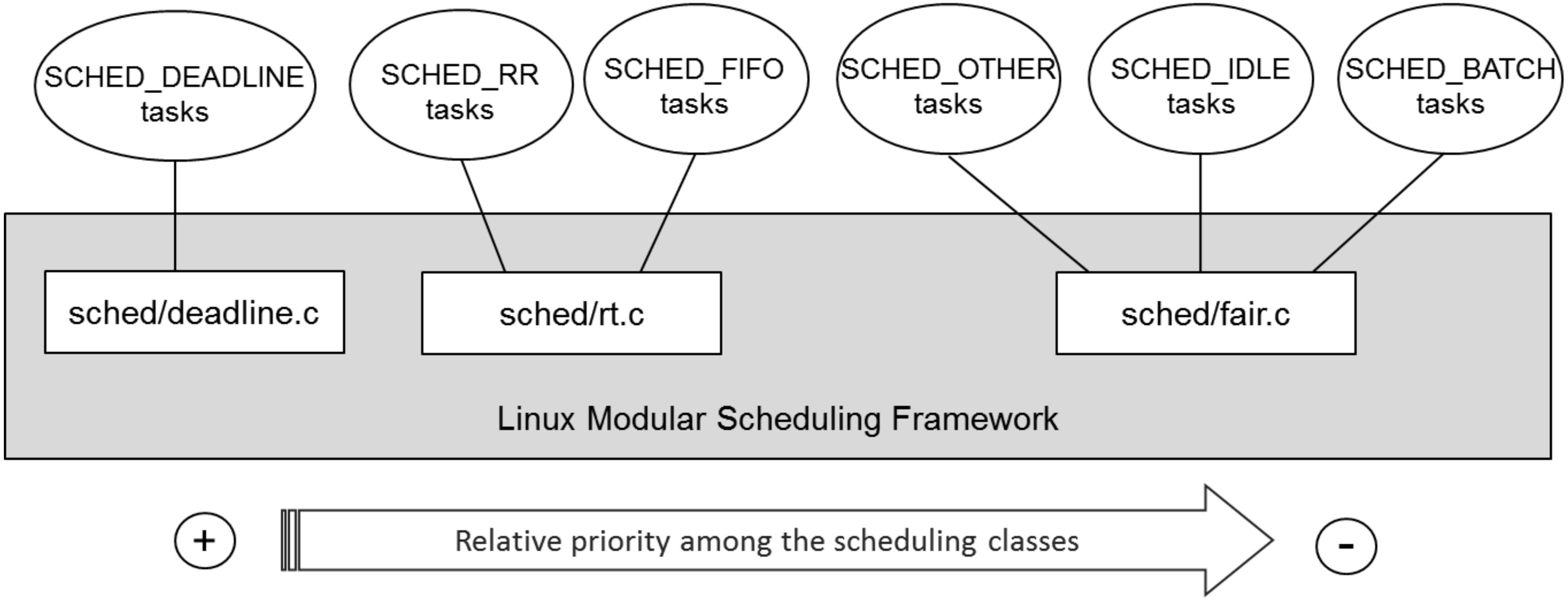
- The CBS allows to serve *non periodic tasks*
  - Some reservation-based schedulers have problems with aperiodic job arrivals - due to the (in)famous “deferrable server problem”
  - The CBS explicitly supports aperiodic arrivals (see the rule for assigning deadlines when a task wakes up)
- Allows to support “self-suspending” tasks
  - No need to strictly respect the Liu&Layland task model
  - No need to explicitly signal job arrivals / terminations

CBS: the Algorithm

- Each task τ is associated a scheduling deadline d^s and a current runtime q
  - ☐ Both initialized to 0 when the task is created
- When a job arrives:
  - ☐ If the previous job is not finished yet, queue the activation
  - ☐ Otherwise, check if the current scheduling deadline can be used ($d^s > t$ and $q/(d^s - t) < Q/P$)
    - If not, $d^s = t + P$, $q = Q$
- When τ executes for a time δ, $q = q - \delta$
- When $q = 0$, τ cannot be scheduled (until time d^s)

SCHED_DEADLINE

- New `SCHED_DEADLINE` scheduling policy
  - ☐ Foreground respect to all of the other policies



SCHED_DEADLINE and CBS

- Uses the CBS to assign scheduling deadline to `SCHED_DEADLINE` tasks
  - ☐ Assign a (maximum) runtime Q and a (reservation) period P to `SCHED_DEADLINE` tasks
  - ☐ Additional parameter: relative deadline D
  - ☐ The “check if the current scheduling deadline can be used” rule is used at task wake-up
- Then uses EDF to schedule them
  - ☐ Both global EDF and partitioned EDF are possible
  - ☐ Configurable through the `cpuset` mechanism

SCHED_DEADLINE Design: Flexibility

- Supports both global and partitioned scheduling
  - For partitioned scheduling, use `cpusets`
- Flexible utilization-based admission control
  - $\sum_j \frac{Q_j}{P_j} \leq U^L$
  - U^L configurable, ranging from 0 to M
    - `/proc/sys/kernel/sched_rt_{runtime, period}_us`
  - Can leave CPU time for non-deadline tasks
  - Bounded tardiness; hard respect of deadlines for partitioned scheduling
- Even supports arbitrary affinities!
  - But admission control must be disabled...

Setting the Scheduling Policy

- No `sched_setsched()` ← new syscalls (and data structures added to be extensible)
  - ☐ Maybe even too extensible!

```
int sched_setattr(pid_t pid, const struct sched_attr *attr,
                  unsigned int flags);
int sched_getattr(pid_t pid, struct sched_attr *attr,
                  unsigned int size, unsigned int flags);

struct sched_attr {
        __u32 size;

        __u32 sched_policy;
        __u64 sched_flags;
...
        __u64 sched_runtime;
        __u64 sched_deadline;
        __u64 sched_period;
};
```

Using `sched_setattr()`

- `pid`: as for `sched_setscheduler()`
- `flags`: currently unused (for future extensions!)
- `attr`: scheduling parameters for the task
  - ☐ `size`: must be set to `sizeof(struct sched_attr)`
  - ☐ `sched_policy`: set to `SCHED_DEADLINE`!
  - ☐ `sched_runtime`: Q
  - ☐ `sched_deadline`: D
  - ☐ `sched_period`: P
  - ☐ `sched_flags`: will see later (set to 0 for now)

libdl

- So, can we use `SCHED_DEADLINE` in our user programs?
- `sched_setattr()` & friends are in the kernel since 3.14...
- But the user-space side of things is still missing in many Linux distributions
  - No support in glibc, no definition of `struct sched_attr`, etc...
- Solution: small user-space library providing the `sched_*attr()` system calls and related data structures
- `libdl`, released by Juri Lelli under GPL

Example

```
#include "libdl/dl_syscalls.h"
...
struct sched_attr attr;
attr.size = sizeof(struct attr);
attr.sched_policy = SCHED_DEADLINE;
attr.sched_runtime = 30000000;
attr.sched_period = 100000000;
attr.sched_deadline = 100000000;
...
res = sched_setattr(0, &attr, 0);
if (res < 0)
  perror("sched_setattr()");
...
```

Admission Control

- `sched_setattr()` might fail if admission control fails
  - ☐ Sum of reserved utilizations exceed the limit U^L
  - ☐ Affinity of the task is different from its root domain
- Why the check on the affinity?
  - ☐ $\sum_j \frac{Q_j}{P_j} \leq M$ guarantees bounded tardiness for global scheduling!
  - ☐ Arbitrary affinities need a different analysis...
- So, how to use arbitrary affinities?
  - ☐ Disable admission control!
  - ☐ `echo -1 > /proc/sys/kernel/sched_rt_runtime_us`

Partitioned Scheduling

- `cpuset`: mechanism for assigning a set of CPUs to a set of tasks
  - Exclusive `cpuset`: CPUs not shared
- Tasks migrate inside *scheduling domains* ⇐ `cpusets` can bee used to create isolated domains
- Only one CPU ⇒ partitioned scheduling

```
mount -t tmpfs cgroup_root        /sys/fs/cgroup
mkdir                             /sys/fs/cgroup/cpuset
mount -t cgroup -o cpuset cpuset /sys/fs/cgroup/cpuset

mkdir        /sys/fs/cgroup/cpuset/Set1
echo 3     > /sys/fs/cgroup/cpuset/Set1/cpuset.cpus
echo 0     > /sys/fs/cgroup/cpuset/Set1/cpuset.mems
echo 0     > cpuset.sched_load_balance
echo 1     > /sys/fs/cgroup/cpuset/Set1/cpuset.cpu_exclusive
echo $PID > /sys/fs/cgroup/cpuset/Set1/tasks
```

Warning!

- `sched_setaffinity()` on `SCHED_DEADLINE` tasks can fail
  - Again, disable admission control to use something different from global scheduling
- `SCHED_DEADLINE` tasks cannot fork
  - Which scheduling parameters would be inherited?
- Remember: runtimes and periods are in nanoseconds (not microseconds)

Using SCHED_DEADLINE

- ...How to dimension the scheduling parameters?
  - (Maximum) runtime Q: `rt_runtime` (in $nsec$)
  - (Reservation) period P: `rt_period` (in $nsec$)
  - `SCHED_DEADLINE` also provides a (relative) deadline D
- Obviously, it must be

$$\sum_j \frac{Q_j}{P_j} \leq M$$

  - The kernel can do this admission control
  - Better to use a limit U^L smaller than M (so that other tasks are not starved!)

Assigning Runtime and Period

- Temporal isolation
  - Each task can be guaranteed independently from the others
- Hard Schedulability property
  - If $Q \geq WCET$ and $P \leq MIT$ (maximum runtime larger than WCET, and server period smaller than task period)...
  - ...Then the scheduling deadlines are equal to the jobs' deadlines!!!
  - All deadlines are guaranteed to be respected (on UP / partitioned systems), or an upper bound for the tardiness is provided (if global scheduling is used)!!!

What About Soft Real-Time?

- What happens if $Q < WCET$, or $P > MIT$?
  - ☐ $\frac{Q}{P}$ must be larger than the ratio between average execution time $\overline{c_i}$ and average inter-arrival time $\overline{t_i}$...
  - ☐ ...Otherwise, $d_i^s \rightarrow \infty$ and there will be no control on the task's response times
- Possible to perform some stochastic analysis (Markov chains, etc...)

Changing Parameters...

- Tasks' parameters (execution and inter-arrival times) can change during the tasks lifetime... So, how to dimension Q and P?
- Short-term variations: CPU reclaiming mechanisms (GRUB, ...)
  - If a job does not consume all of the runtime Q, try to reuse the residual
- Long-term variations: adaptive reservations
  - Generally "slower", can be implemented by a user-space daemon
  - Monitor the difference between d^s and d_j
    - If $d^s - d_j$ increases, Q needs to be increased

CPU Reclaiming!

- As mentioned, CPU reclaiming can be used to better tolerate short-term variations in the execution times...
- ...And a CPU reclaiming mechanism has just been added to `SCHED_DEADLINE`!
    - ☐ Available since Linux 4.13
    - ☐ M-GRUB: multi-processor GRUB: per-runqueue reclaiming of unused CPU time
- Ah... This is what the `sched_flags` field is for! Set `SCHED_FLAG_RECLAIM` (2)