Linux Scheduler Internals

Luca Abeni
luca.abeni@santannapisa.it

March 14, 2022

Multiprocessor Scheduling

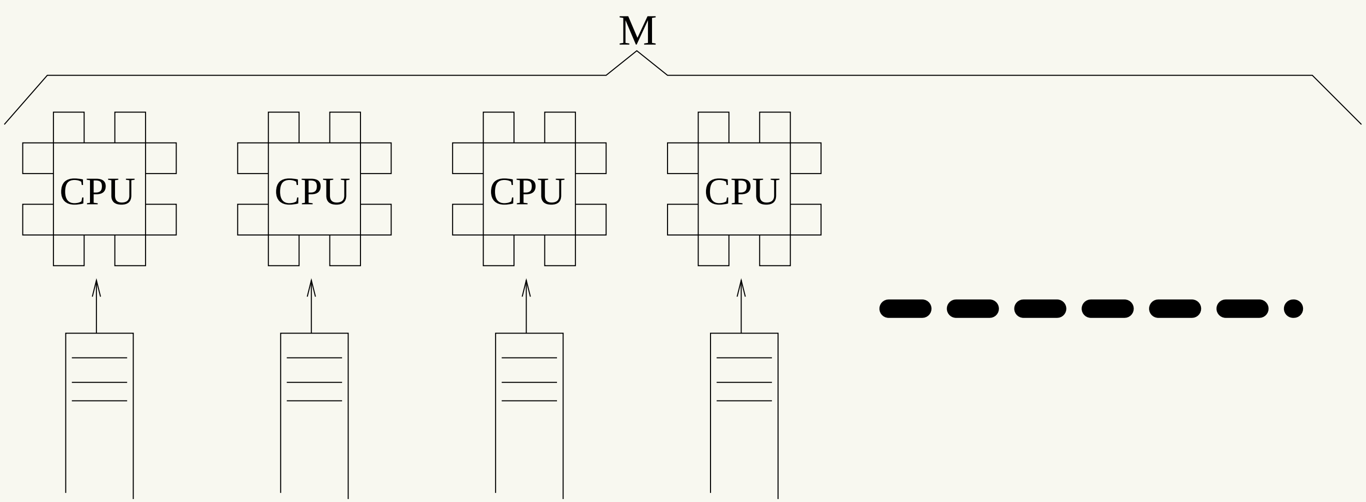
- UniProcessor Systems
  - A schedule $\sigma(t)$ is a function mapping time t into an executing task $\sigma : t \rightarrow \mathcal{T} \cup \{\tau_{idle}\}$ where $\mathcal{T}$ is the set of tasks running in the system
  - τ_{idle} is the *idle task*
- For a multiprocessor system with M CPUs, $\sigma(t)$ is extended to map t in vectors $\tau \in (\mathcal{T} \cup \{\tau_{idle}\})^M$
- Scheduling algorithms for $M > 1$ processors?
  - Partitioned scheduling
  - Global scheduling

The Quest for Optimality

- UP Scheduling:
  - N periodic tasks with $D_i = T_i$: (C_i, T_i, T_i)
  - Optimal scheduler: if $\sum \frac{C_i}{T_i} \leq 1$, then the task set is schedulable
  - EDF is optimal
- Multiprocessor scheduling:
  - Goal: schedule periodic task sets with $\sum \frac{C_i}{T_i} \leq M$
  - Is this possible?
  - Optimal algorithms

Partitioned Scheduling - 1

- Reduce $\sigma : t \rightarrow (\mathcal{T} \cup \{\tau_{idle}\})^M$ to M uniprocessor schedules $\sigma_p : t \rightarrow \mathcal{T} \cup \{\tau_{idle}\}, 0 \leq p < M$
  - Statically assign tasks to CPUs
  - Reduce the problem of scheduling on M CPUs to M instances of uniprocessor scheduling
  - Problem: system underutilisation

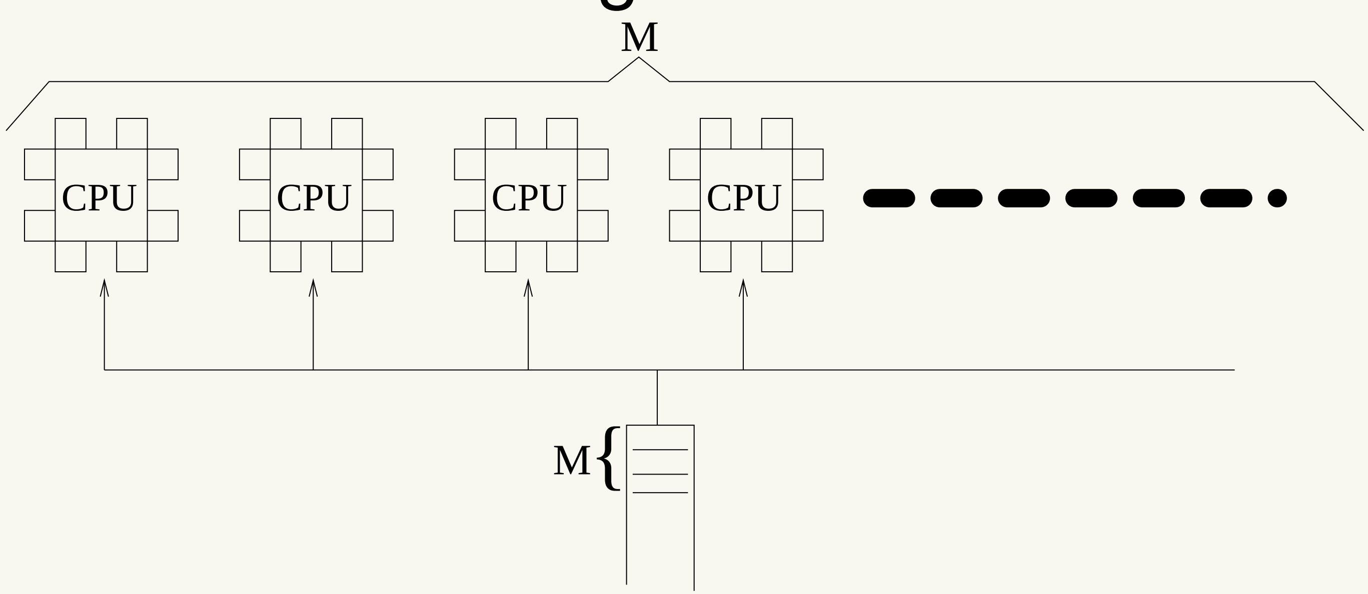


Partitioned Scheduling - 2

- Reduce an M CPUs scheduling problem to M single CPU scheduling problems and a bin-packing problem
- CPU schedulers: uni-processor, EDF can be used
- Bin-packing: assign tasks to CPUs so that every CPU has load ≤ 1
  - Is this possible?
- Think about 2 CPUs with $\{(6, 10, 10), (6, 10, 10), (6, 10, 10)\}$

Global Scheduling

- One single task queue, shared by M CPUs
  - The first M ready tasks are selected
  - What happens using fixed priorities (or EDF)?
  - Tasks are not bound to specific CPUs
  - Tasks can often migrate between different CPUs
- Problem: schedulers designed for UP...



Global Scheduling - Problems

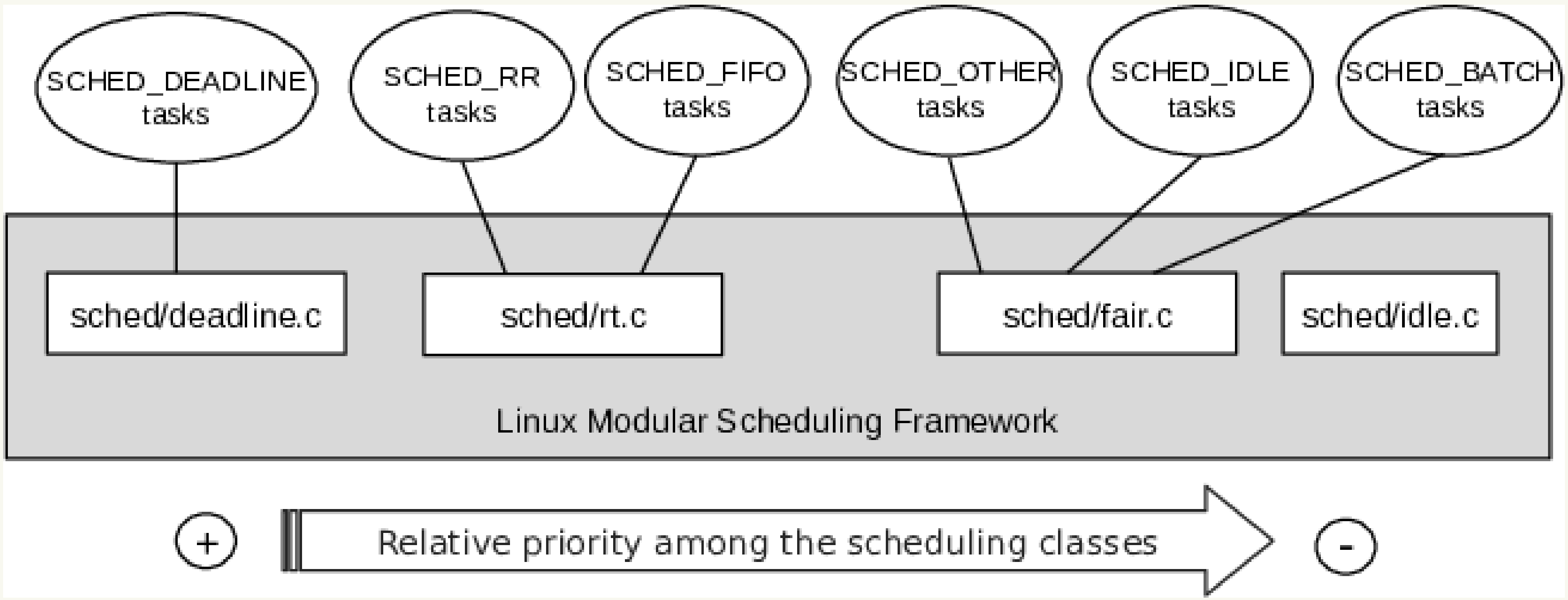
- Dhall's effect: U^{lub} for global multiprocessor scheduling can be 1 (for RM or EDF)
  - Pathological case: M CPUs, $M+1$ tasks. M tasks $(\epsilon, T-1, T-1)$, a task (T, T, T).
  - $U = M\frac{\epsilon}{T-1} + 1$. $\epsilon \to 0 \Rightarrow U \to 1$
- Global scheduling can cause a lot of useless migrations
  - Migrations are overhead!
  - Decrease in the throughput
  - Migrations are not accounted for...

Global Scheduling for Soft Tasks

- Dhall's Effect $\rightarrow$ global EDF and global RM have $U^{lub} = 1$
  - With $U > 1$, deadlines can be missed
  - Global EDF / RM are not useful for hard tasks
- However, global EDF can be useful for scheduling soft tasks...
- When $U \leq M$, global EDF guarantees an upper bound for the *tardiness*!
  - Deadlines can be missed, but by a limited amount of time

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 `cpuset`s
- 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* $\Leftarrow$ `cpusets` can be used to create isolated domains
- Only one CPU $\Rightarrow$ partitioned scheduling

```
# The next 3 lines are not needed in many Linux distributions
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    > /sys/fs/cgroup/cpuset/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)

Task Affinities in Linux

- Linux scheduler: more generic than “simple” partitioned or global schedulers
  - Every task has an *affinity mask*
  - Bitmask describing all the CPU cores on which the task can be scheduled
    - Mask == all cores $\rightarrow$ global scheduling
    - Mask == 1 core $\rightarrow$ partitioned scheduling
- Also, `cpuset` mechanism to impose constraints on the tasks affinity masks
  - Remember the previous example with `SCHED_DEADLINE`
- When migrating a task, the scheduler **has** to look at its affinity mask

Affinity Masks in the Task Structure

- The `task_struct` structure has a `cpus_mask` field, of type `cpumask_t`
  - Bitmask containing CPU cores, accessible through the `cpumask_...` functions and macros
  - Example: `cpumask_weight(...)` returns the number of bits set to 1
  - `cpumask_weight(t->cpus_mask)` returns the number of cores on which task `t` can be scheduled
    - Cached in `t->nr_cpus_allowed`
  - The `cpus_ptr` field caches the `cpus_mask` address
- Can be set with `sched_setaffinity()`

Affinity Masks and SCHED_DEADLINE

- The SCHED_DEADLINE policy is subject to admission control
  - Remember? `sched_setattr()` can fail even if you are administrator!!!
  - See `__sched_setscheduler()` returning `-EPERM`...
- The admission control assumes global scheduling
  - So, the affinity mask must contain all the CPU cores!
  - See the check "`!cpumask_subset(span, p->cpus_ptr)`"
  - Here, "`span`" is a bitmask containing all the cores available to the scheduler

Affinity Masks, Again

- If admission control is disabled, then generic affinities can be used
- How are affinities used?
  - Example based on `SCHED_DEADLINE` (as usual)
  - `rt.c` (implementing `SCHED_FIFO` and `SCHED_RR`) is similar
- The "push" and "pull" functions look at "*pushable* dl tasks" (stored in an RB tree)
  - Tasks are stored in such an RB tree only if `nr_cpus_allowed` > 1
- If the affinity mask contains all cores, then push and pull implement global scheduling
- With generic affinities, things are more complex

A Partitioned SCHED_DEADLINE

- `!cpumask_subset(span, p->cpus_ptr)` implies global scheduling...
- ...How to modify it to have partitioned scheduling?
  - Hint: each task should be affine to only 1 CPU...
- Then, other related changes are needed...
  - Cope with `SCHED_DEADLINE` tasks trying to change their affinity...
  - Cope with changes in the `cpuset` configuration...
- The admission test (see `__dl_overflow()`) also needs to be modified
- After that, push and pull functions become useless/unused!

Coping with Changes in Affinity Masks

- Current `SCHED_DEADLINE`: the task's affinity mask must contain all the CPU cores that can be used by the scheduler
  - See the check in `__sched_setscheduler()`
  - What happens if `cpus_allowed` changes *after* the task has become `SCHED_DEADLINE`?
- The kernel must prevent changes in the tasks' affinity masks that break this property
  - See the check in `sched_setaffinity()`
- Special case of affinity change: moving between different `cpuset`s
  - See `deadline.c::set_cpus_allowed_dl()`

Coping with Changes in cpusets

- Current `SCHED_DEADLINE`: the task's affinity mask must contain all the CPU cores that can be used by the scheduler
  - Remember "`span`"? (from `rq->rd->span`)
- The kernel must prevent changes in cpusets that break this property (or break admission control)
  - Look at `kernel/cgroup/cpuset.c::validate_change`
- This must be modified if `SCHED_DEADLINE` does not enforce global scheduling

Admission Control

- Not present in `SCHED_{FIFO,RR}`
- Currently based on global scheduling
  - Considers the `cpuset`'s (root domain's) utilization
  - Remember: utilization $U = \texttt{runtime}/\texttt{period}$
- See `struct dl_bw *dl_b` in `__dl_overflow()`
  - Member of the "root domain" structure
  - Contains a maximum bw field and a current bw field
- Must be changed to a per-rq admission control
  - The rq utilization is already tracked by `this_bw`

The Root Domain Utilization

- Root domain (isolated `cpuset`): contains all the information about the CPU cores usable by the scheduler
  - `rq->rd->dl_bw`: utilization of the dl tasks in the root domain
  - See `kernel/sched/deadline.c::dl_bw_of()` and related stuff
- The root domain utilization is updated when a task switch to/from `SCHED_DEADLINE` and when a dl task ends
  - Search for `TASK_DEAD` in `kernel/sched/deadline.c`