Process identifiers
Process scheduling
Linux schedulers – overview

O(N), O(1), RSDS

Table of contents

Process, thread, group, session

A **thread** has its own execution context and its own **thread ID** number but can share most other details, particularly an address space, a set of signal handlers, and a **process ID** number, with other threads in the same process (the same **thread group**). The identifier shared by the threads is the PID of the **thread group leader (TGL)**, that is, the PID of the first thread in the group. A thread can only leave a process by exiting.

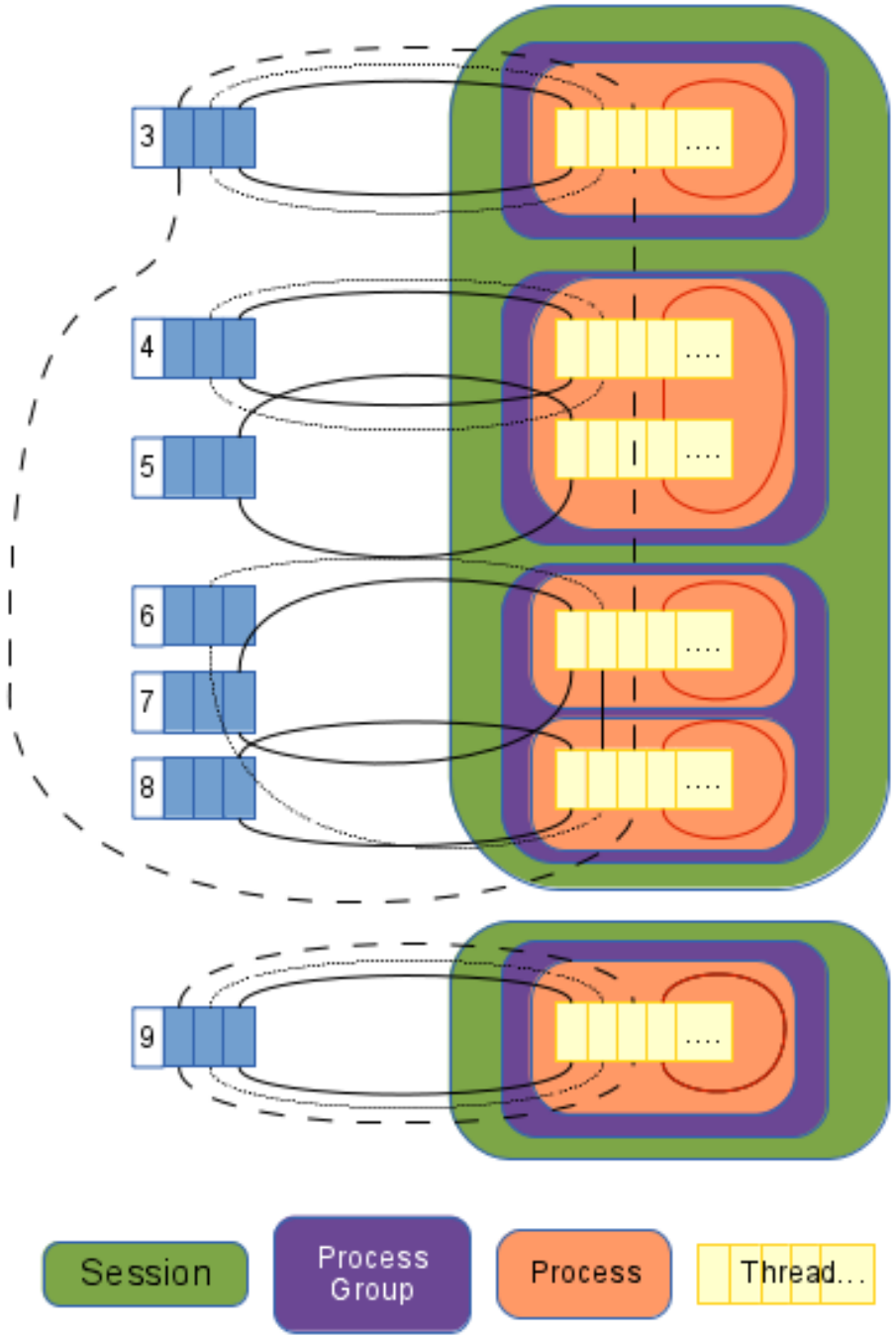
Processes may belong to various **process groups**. Each process descriptor includes a field containing the **process group ID**. Each group of processes may have a **group leader**, which is the process whose PID coincides with the process group ID. A **process group leader** (PGL) must also be a **thread group leader** (TGL). A process can move from one process group to another.

A **login session** contains all processes that are descendants of the process that has started a working session on a specific terminal – usually, the first command shell process created for the user. All processes in a **process group** must be in the same **login session**. A session leader can exit before the end of the session.

See: setsid(), setpgid()

Thread groups, process groups, sessions (**struct pid** is pictured blue)
(source: [Control groups, part 6: A look under the hood](), Neil Brown, August 2014)

PIDTYPE_TGID // added in 2018



Cgroups, namespaces, and beyond: what are containers made from?

- What is **important**:
  - **uniqueness** (unambiguous identification),
  - **safety** (processes appear and disappear),
  - **isolation** (because containerization).
- **Cgroups** = limits how much you can use.
- **Namespaces** = limits what you can see.
- **Multiple namespaces**: **pid**, net, mnt, uts, ipc, user.
- Each **process** is in **one namespace** of each **type**.
- Processes within a **PID namespace** only see processes in the **same PID namespace**.
- Each **PID namespace** has its own numbering **starting at 1**.
- If PID 1 goes away, whole namespace is **killed**.

https://www.youtube.com/watch?v=sK5i-N34im8&t=2028s, Jérôme Petazzoni, Tinkerer Extraordinaire, Docker

Process identifier

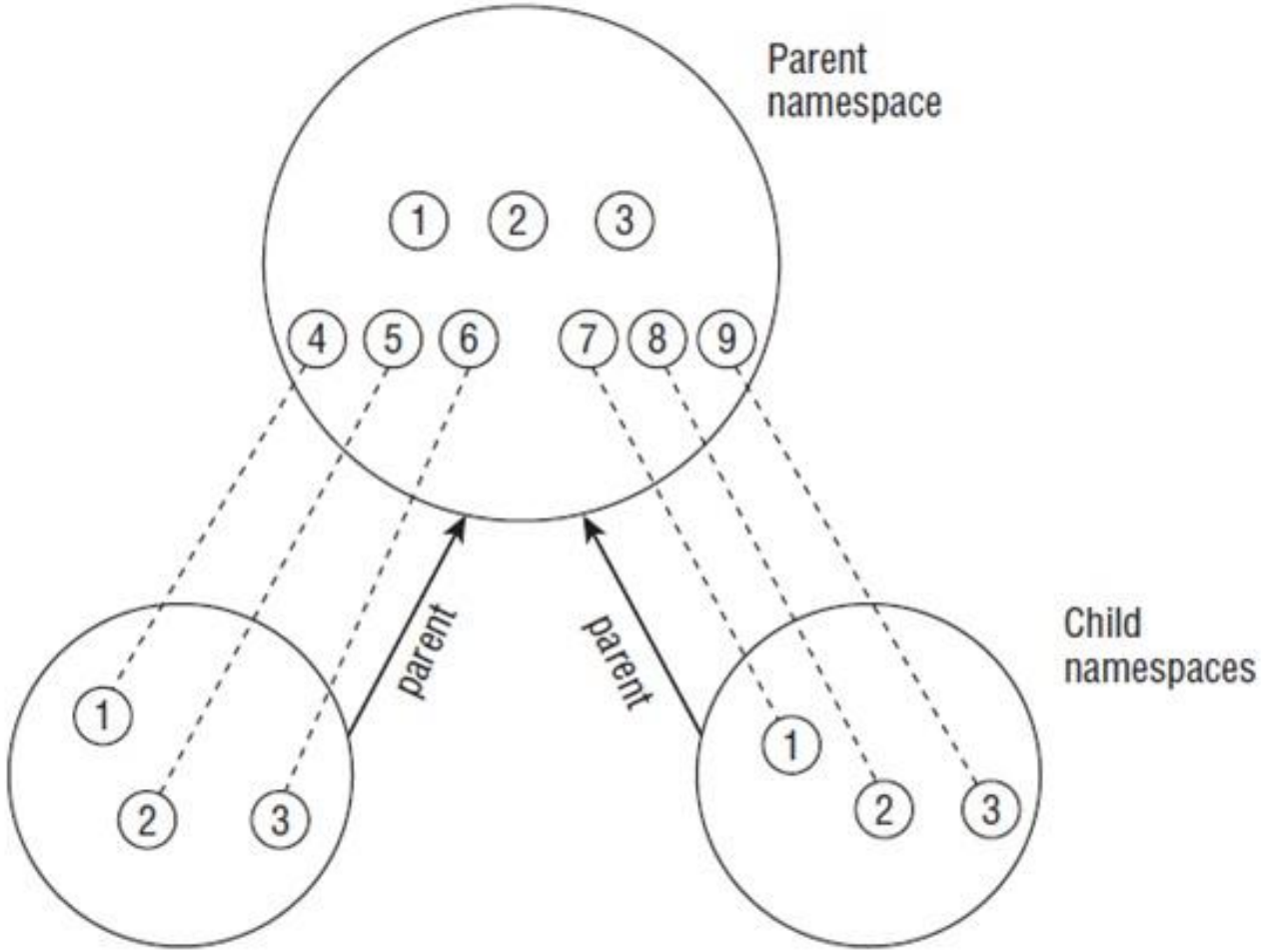
Each **namespace** has its own unique set of **process identifiers** (**PID**s).

PID namespaces are **hierarchical**; once a new PID namespace is created, all the tasks in the current PID namespace will see the tasks in this new namespace.

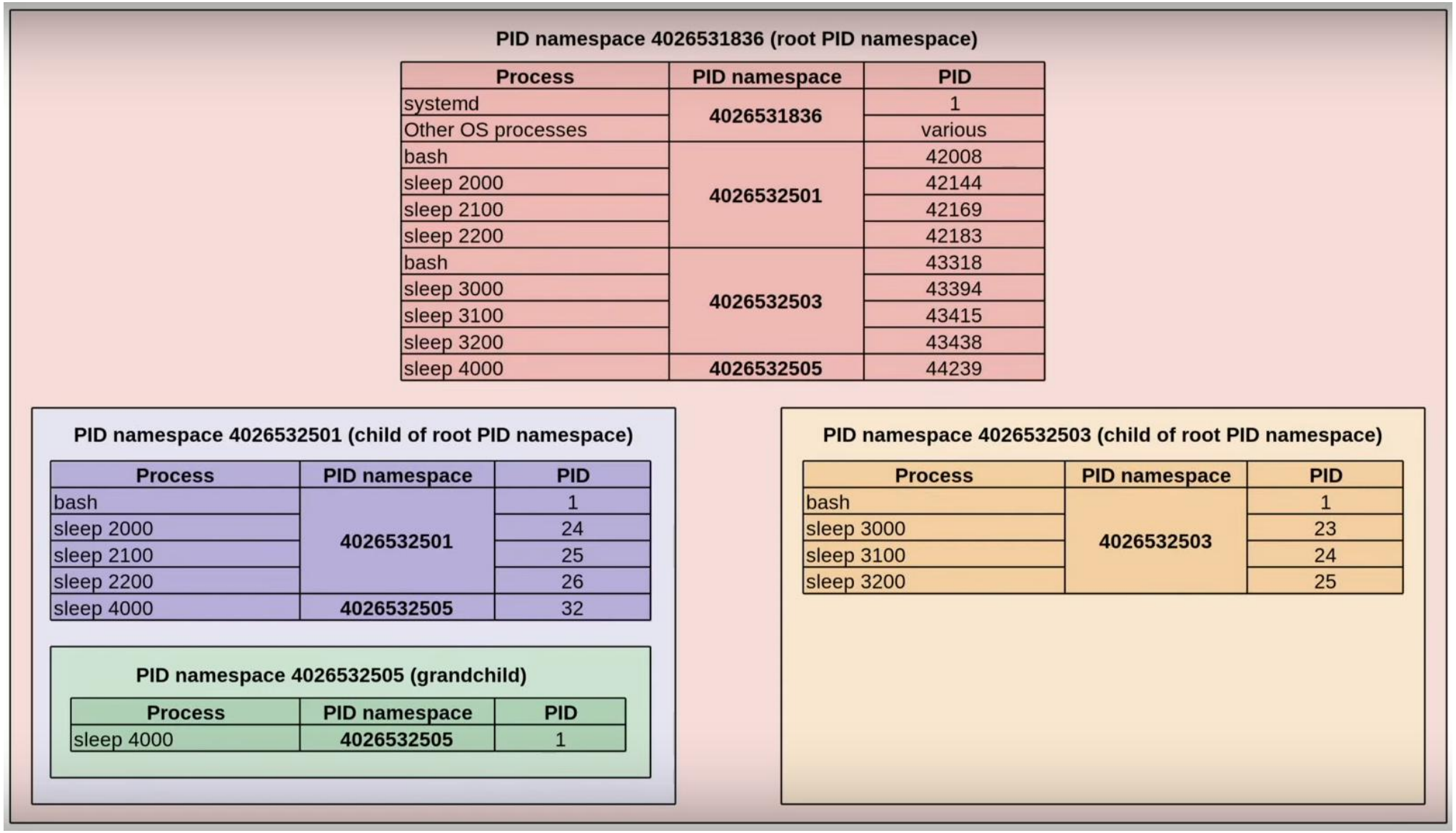
Tasks from the new namespace will not see the ones from the current.

Each task has more than one PID – one for each namespace.

A **new PID namespace** is created by calling **clone()** with the **CLONE_NEWPID** flag.



PID Namespace Hierarchy
(source: From the presentation by Fu-Hau Hsu)



How containers use PID namespaces to provide process isolation, Brian Smith , RedHat

```
enum pid_type                                          Simplified version
{
        PIDTYPE_PID,
        PIDTYPE_TGID,          // added in 2018
        PIDTYPE_PGID,
        PIDTYPE_SID,
        PIDTYPE_MAX,
};
struct signal_struct {
        int             nr_threads;
        struct list_head thread_head;
        /* PID/PID hash table linkage. */
        struct pid *pids[PIDTYPE_MAX];
        ...
};
struct task_struct
{
   struct task_struct      *group_leader;
                           // thread group leader
        ...
   pid_t           pid;  /* global PID */
   pid_t           tgid; /* global TGID */

   struct signal_struct   *signal;
   struct pid             *thread_pid;
   struct hlist_node      pid_links[PIDTYPE_MAX];
   struct list_head       thread_group;
   struct list_head       thread_node;
}
```

Processes in the same thread group are chained together through the **thread_group** field of their **tast_struct** structures (is being replaced by **thread_node**).

signal_struct contains a number of fields that are specific to a **process** (as opposed to a **thread**).

```
struct upid {                                          Simplified version
        int nr;
        struct pid_namespace *ns;
};
struct pid
{
        refcount_t          count;
        unsigned int        level;
        /* lists of tasks that use this pid */
        struct hlist_head   tasks[PIDTYPE_MAX];
        struct rcu_head     rcu;
        struct upid         numbers[1];
};
```

Process identifier

Global IDs are identification numbers that are **valid within the kernel** itself and in the **initial global namespace**. For each ID type, a given global identifier is guaranteed to be **unique** in the whole system. The global PID and TGID are directly stored in the **task_struct**, namely, in the elements **pid** and **tgid**.

Local IDs belong to a **specific namespace** and are **not globally valid**. For each ID type, they are valid within the namespace to which they belong, but identifiers of identical type may appear with the same ID number in a different namespace.

A **struct pid** is the kernel's **internal notion** of a process identifier. It refers to individual **tasks**, **process groups**, and **sessions**. While there are processes attached to it, the **struct pid** lives in a **hash table**, so it and then the processes that it refers to can be found quickly from the numeric **pid** value. The attached processes may be quickly accessed by following pointers from **struct pid**.

Process identifier

A process will have **one PID** in each of the layers of the PID namespace hierarchy.

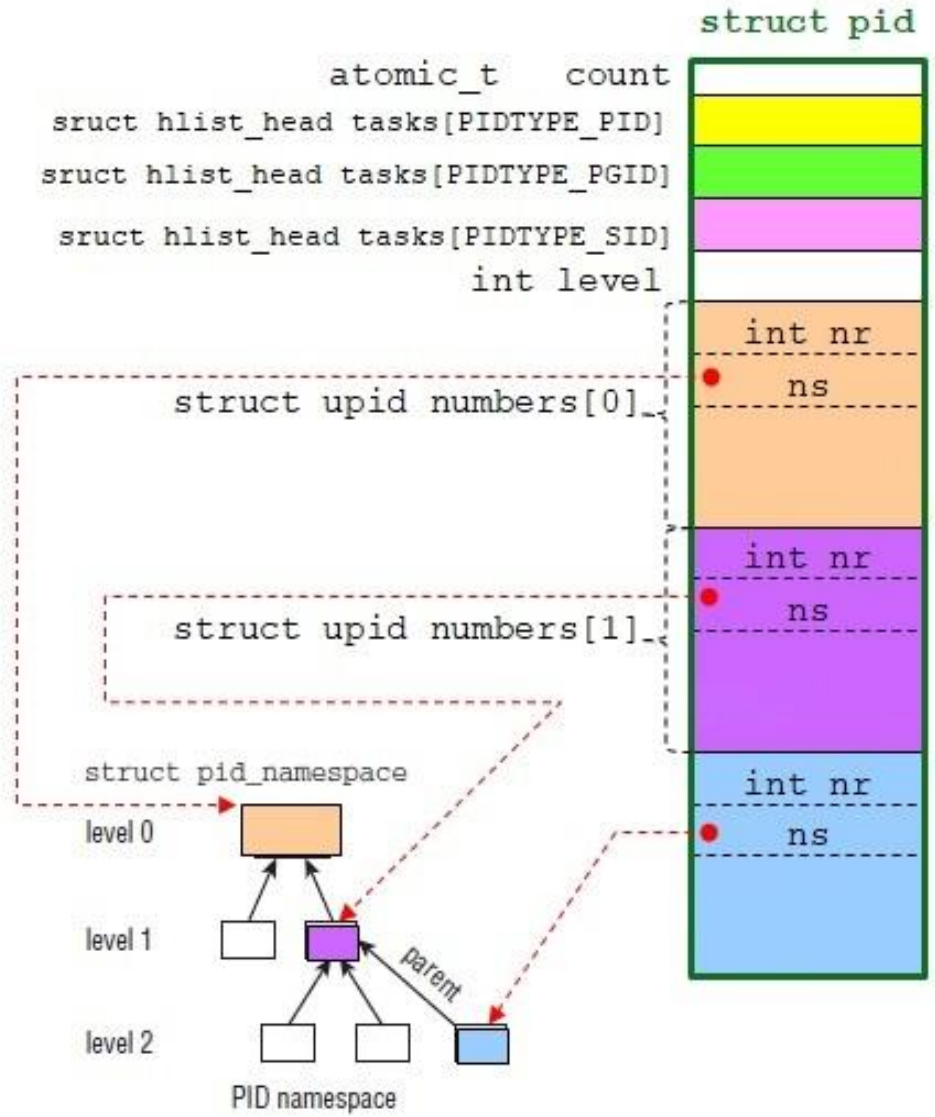
struct upid is used to get the **id** of the **struct pid**, as it is seen in particular **namespace**.

Field **level** denotes in how many namespaces the process is visible (this is the depth of the containing namespace in the namespace hierarchy).

Field **numbers** contains a **upid** instance for each level. The array consists formally of one element, and this is true if a process is contained only in the global namespace. Since the element is at the end of the structure, additional entries can be added to the array by simply allocating more space.

struct upid numbers[]

```
struct upid {
    int nr;
    struct pid_namespace *ns;
};
struct pid
{
    refcount_t          count;
    unsigned int        level;
   struct hlist_head
                tasks[PIDTYPE_MAX];
    struct rcu_head     rcu;
    struct upid         numbers[1];
};
```



PIDTYPE_TGID // added in 2018

struct upid numbers[] (source: From the presentation by Fu-Hau Hsu)

PID space

There is a list of **bitmap pages**, which represent the **PID space**. Allocating and freeing PIDs is completely **lockless**.

The worst-case allocation scenario when all but one out of 1 million PIDs possible are allocated already: the scanning of 32 list entries and at most PAGE_SIZE bytes. The typical fastpath is a single successful setbit. Freeing is O(1).

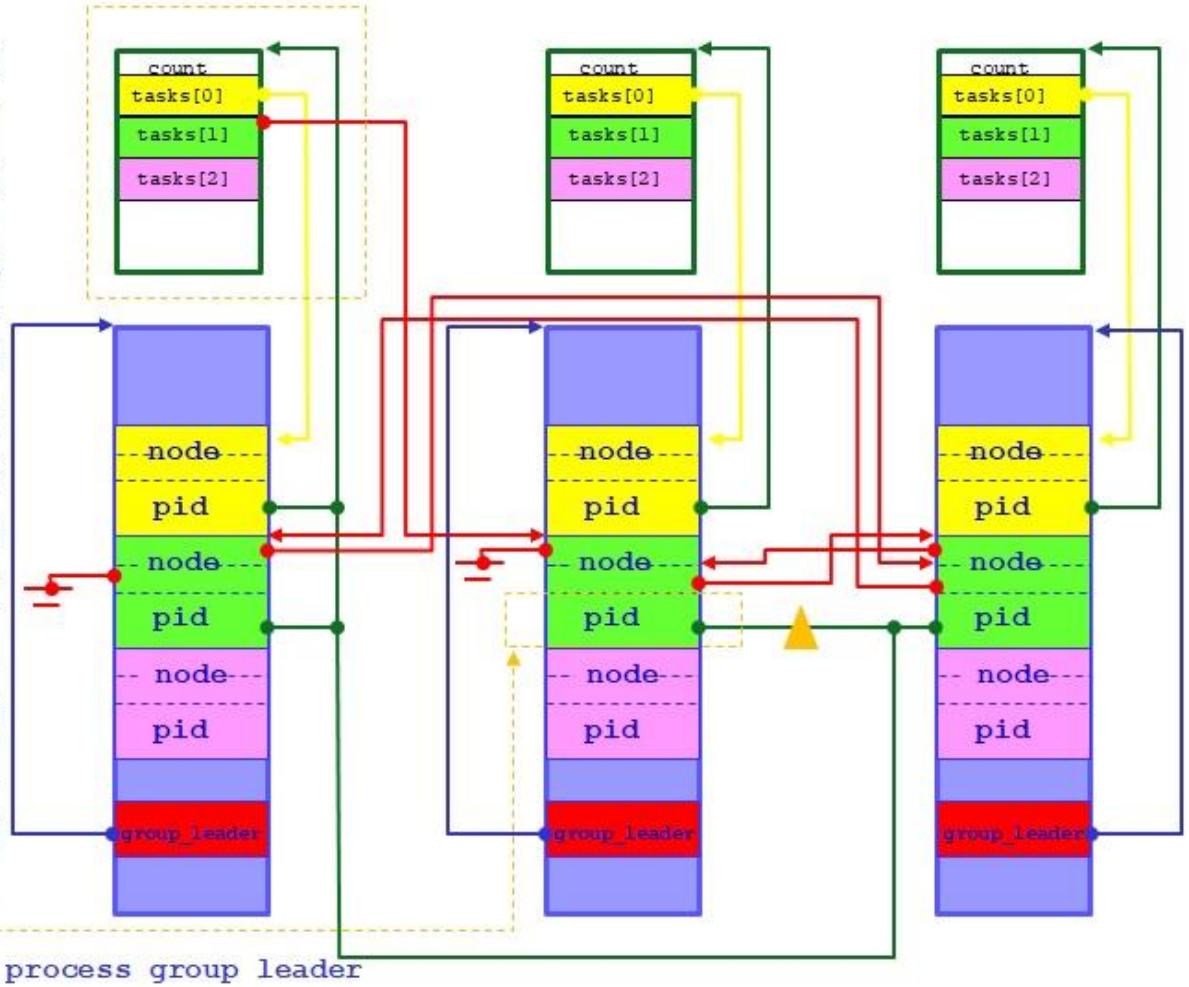
The **IDR facility** is used for that purpose (**integer ID management** – idr object can be thought of as a **sparse array** mapping **integer IDs** onto **arbitrary pointers**).

```
struct pid *find_pid_ns(int nr, struct pid_namespace *ns)
        { return idr_find(&ns->idr, nr); }
```

The **maximum value of PID** can be listed by **cat /proc/sys/kernel/pid_max** (on 'students' server the value is **4 194 304**).

Linked List That Links All Processes in a Process Group

If a process is a process group leader (P.S.: it must also be a thread group leader), the `tasks[1]` field of the `pid` instance pointed by the `pids[1].pid` of its `task_struct` is the head of the linked list that links the field `pids[1].pid` of the `task_struct` of all *thread group leaders* in the same *process group*.



Linked list that links all processes in a process group, (source: From the presentation by Fu-Hau Hsu)

Process identifier

Code from

/include/linux/sched/signal.h

showing how to get various identifiers

Simplified version

```
static inline struct pid *task_pid(struct task_struct *task)
{
        return task->thread_pid;
}
static inline struct pid *task_tgid(struct task_struct *task)
{
        return task->signal->pids[PIDTYPE_TGID];
}
static inline struct pid *task_pgrp(struct task_struct *task)
{
        return task->signal->pids[PIDTYPE_PGID];
}
static inline struct pid *task_session(struct task_struct *task)
{
        return task->signal->pids[PIDTYPE_SID];
}
static inline int get_nr_threads(struct task_struct *task)
{
        return task->signal->nr_threads;
}
```

Pidfd – file descriptor that refers to a process

Adding the pidfd abstraction to the kernel (Jonathan Corbet, October 2019)

When a process exits, its **PID** will eventually be **recycled** and assigned to an entirely **unrelated process**. This leads to a number of **security issues**.

A **pidfd** is a **file descriptor** that refers to a process's **thread-group leader**.

Pidfds are **stable** (they always refer to the same process) and **private** to the owner of the file descriptor. Internally to the kernel, a pidfd refers to the **pid structure** for the target process.

Why needed:

- to avoid vulnerabilities resulting from PID reuse,
- to support shared libraries that need to create invisible helper processes (persistency),
- for process-management applications that want to delegate the handling of specific processes to a non-parent process, e.g. for waiting, signalling (examples are the Android low-memory killer daemon and systemd).

Implemented by **Christian Brauner**

Additional reading

- Process file descriptors on Linux (Christian Brauner, 2019)
- Toward race-free process signaling (Marta Rybczyńska, December 2018).
- PID Allocation in Linux Kernel (Gargi Sharma, March 2017).
- Control groups, part 6: A look under the hood (Neil Brown, August 2014).
- Namespaces in operation, part 3: PID namespaces (Michael Kerrisk, January 2013).
- PID namespaces in the 2.6.24 kernel (Pavel Emelyanov and Kir Kolyshkin, November 2007).
- idr – integer ID management (Jonathan Corbet, September 2004).

Pointer to the current process – macro **current**

The kernel most often refer to processes through a **pointer** to the **process descriptor**.

It uses a special **macro** (architecture dependent), because these references must be very fast.

In some architectures, the pointer to the current process is stored in the **registry**, where there are not too many registers the **stack pointer** is used (kept in the registry **esp**).

When a certain process is performed in kernel mode, the **esp** register points to the top of the **kernel-mode stack** of that particular process.

In /arch/x86/include/asm/current.h

```
DECLARE_PER_CPU(struct task_struct *, current_task);

static __always_inline struct task_struct *get_current(void)
{
        return this_cpu_read_stable(current_task);
}
#define current get_current()
```

Simplified version

The macros **current** and **current_thread_info** usually appear in the kernel code as the prefix of the **task_struct** fields of the process.

Context switch

When the **schedule()** function is called (as a result of handling an interrupt or a system call), there may be a need to **switch the context** (change the currently executed process).

The set of data that must be loaded into the registers before the process resumes its execution on the CPU is called the **hardware context**. The hardware context is a subset of the **process execution context**, which includes all information needed for the process execution.

In Linux, a part of the hardware context of a process is stored in the **process descriptor**, while the remaining part is saved in the **kernel mode stack of the process**.

Linux uses **software** to perform a **process switch**. Process switching occurs only in **kernel mode**. The contents of all registers used by a process in user mode have already been saved on the **kernel mode stack** before performing process switching. This includes the contents of the **ss** and **esp** pair that specifies the **user mode stack pointer address**.

Context switch

The 80x86 architecture includes a specific segment type called the **Task State Segment** (**TSS**) to store hardware contexts. The **tss_struct** structure describes the format of the TSS. The **init_tss** array stores one TSS for each CPU on the system. At each process switch, the kernel updates some fields of the TSS so that the corresponding CPU's control unit may safely retrieve the information it needs. Thus, the TSS reflects the privilege of the current process on the CPU, but there is **no need to maintain TSSs for processes when they're not running**.

At every process switch, the hardware context of the process being replaced must be saved somewhere. It cannot be saved on the TSS, as in the original Intel design, because Linux uses a **single TSS for each processor**, instead of **one for every process**.

Thus, each process descriptor includes a field called **thread** of type **thread_struct**, in which the kernel saves the hardware context whenever the process is being switched out. This data structure includes fields for most of the CPU registers, except the general-purpose registers such as eax, ebx, etc., which are stored in the **kernel mode stack**.

Context switch

Simplified version

```
struct task_struct {
    ...
    struct  thread_struct  thread;
    ...
};

struct  thread_struct {
   /* Cached TLS descriptors: */
    struct  desc_struct  tls_array[GDT_ENTRY_TLS_ENTRIES];
    unsigned long   sp;
    unsigned short  es;
    unsigned short  ds;
    struct io_bitmap *io_bitmap;
     ...
};
```

Simplified version

```
struct  tss_struct {
    /*
     * The fixed hardware portion...
     */
    struct x86_hw_tss   x86_tss;
    unsigned long       io_bitmap[IO_BITMAP_LONGS + 1];
     ...
};
```

Context switch

Every process switch consists of two steps:

- Switching the **Page Global Directory** to install a new address space.
- Switching the **kernel mode stack** and the **hardware context**, which provides all the information needed by the kernel to execute the new process, including the CPU registers.

The second step of the process switch is performed by the **switch_to()** macro.
It is one of the most hardware-dependent routines of the kernel.

A process switch may occur at one well-defined point – the **schedule()** function.

```
static __always_inline struct rq *
context_switch  (struct rq *rq, struct task_struct *prev,
                 struct task_struct *next, struct rq_flags *rf)
{
        prepare_task_switch(rq, prev, next);
        arch_start_context_switch(prev);

        ....

        switch_to(prev, next, prev);
        barrier();

        return finish_task_switch(prev);
}
```

Simplified version

The switch_to macro

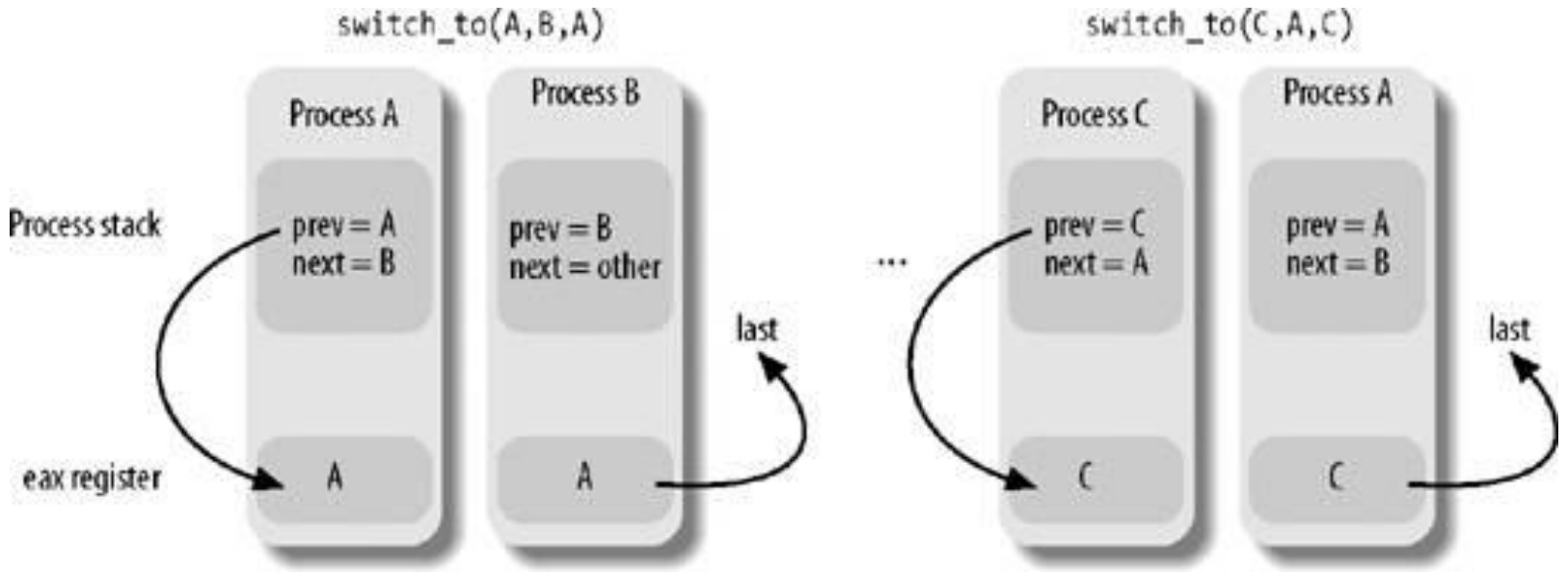
The function **context_switch()** calls **switch_to(prev, next, last)**.

The first two parameters correspond to the local variables: **prev** (process to be suspended) and **next** (process to be executed on the CPU).

What about the third parameter?

The last parameter identifies the **prev** local variable of A, so **prev** is overwritten with the address of C. The figure shows the contents of the kernel mode stacks of processes A, B, and C, together with the values of the **eax** register. The figure shows the value of the **prev** local variable **before** its value is overwritten with the content of the **eax** register.

Preserving the reference to process C across a process switch (source: Bovet, Cesati)



The switch_to macro

Since control flow comes back to the middle of the function, this cannot be done with regular **function return values**, and that is why a three-parameter macro is used. However, the *conceptional effect* is the same as if **switch_to** were a function of two arguments that would return a pointer to the previously executing process. What **switch_to** essentially does is

```
prev = switch_to(prev,next)
```

where the **prev** value returned is not the **prev** value used as the argument, but the process that executed last in time.

In the above example, process A would feed **switch_to** with A and B, but would obtain **prev**=C as result.

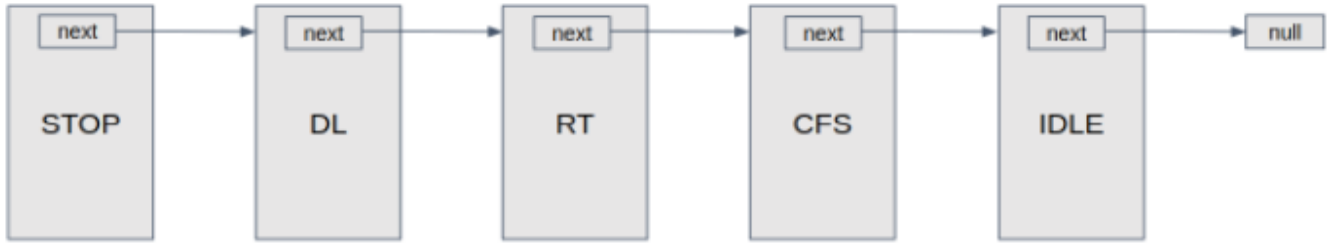
Additional reading:

- Context switch in Linux, Gabriel Kliot, Technion
- Linux Scheduling and Kernel Synchronization, 7.1.2 Context Switch (chapter from the textbook Linux Kernel Primer, A Top-Down Approach for x86 and PowerPC Architectures)

Process classes and fields of task_struct used by process schedulers

Scheduling classes and policies (2019, kernel 5.4) in **descending** order of **priorities**:

- Stop: no policy (available for SMP, used by the kernel for task migration, CPU hotplug, ftrace).
- Deadline: **SCHED_DEADLINE** (used for periodic real time user tasks, eg. video encode/decode).
- Real time: **SCHED_RR** (100ms timeslice by default), **SCHED_FIFO**.
- Fair: **SCHED_NORMAL**, **SCHED_BATCH** (treated as permanently computationally oriented), **SCHED_IDLE** (run only if there is nothing else to run).
- Idle: no policy (keeps a single per-CPU kthread, which may take the CPU to low power state).



Fields of task_struct (*some new details are omitted*):

- const struct sched_class ***sched_class**.
- unsigned int **policy** – the scheduling mode for the process.

Process classes and fields of task_struct used by process schedulers

Fields of task_struct (continued):

- int **static_prio** – **static priority**.
- int **prio** – **dynamic priority**.
- unsigned int **rt_priority** – priority for **real-time** processes.
- unsigned int **time_slice** – time slice.
- struct sched_entity **se**, sched_rt_entity **rt**.
- struct task_group ***sched_task_group**.
- TIF_NEED_RESCHED – a bit indicating that the system should re-schedule processes as soon as possible. Its value is provided by the function **need_resched** presented in a simplified form):

```
static inline int need_resched(void)
{
        return test_thread_flag(TIF_NEED_RESCHED);
}
```

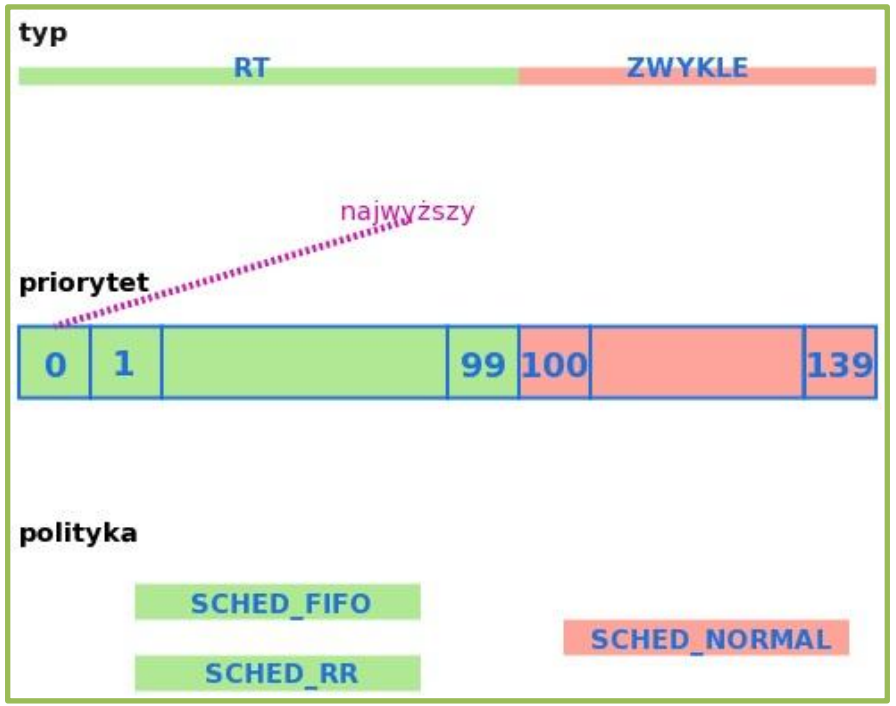
Variable **jiffies** – system time counter.

Priorities

Static priority is the basis for calculating the **time slice** for the process . When creating a child process, the static priority of the parent process is divided equally between the parent and child processes. It is a number between **0 and 139**, with priorities ranging from **0 to 99** for **real-time processes**, and from **100 to 139** for **normal processes**.

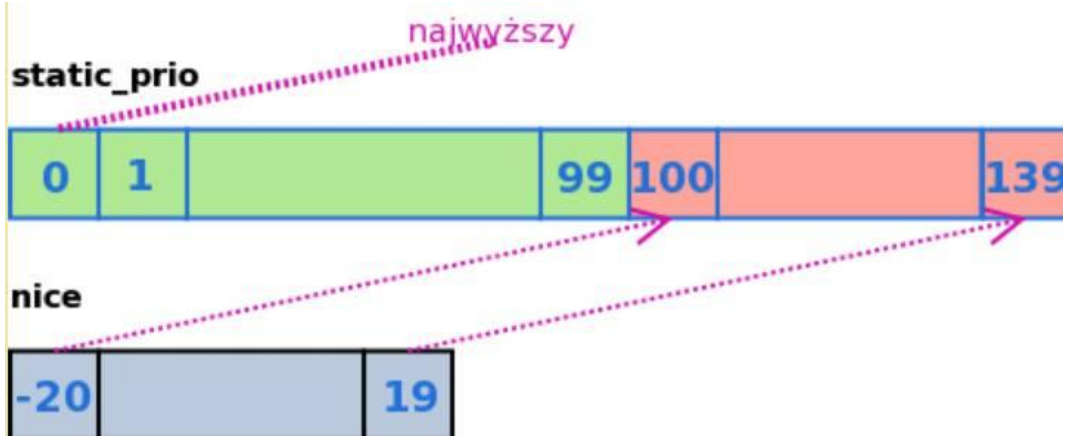
Dynamic priority decides about the choice of the process for execution. For normal processes, it is calculated by the system during system operation based on the static priority and some additional bonus. It accepts values from 100 to 139. The bonus takes values from 0 to 10.

dynamic priority = max(100, min(static priority - bonus + 5, 139))



The lower the number, the higher the priority

Priorities



The static priority can be changed by calling the function **nice()** or **setpriority()**.

The **task_nice** function calculates the **nice** value for the given **static_prio**.

NICE_TO_PRIO macro computes the **static_prio** value for the given **nice**.

The **task_prio** function returns the task priority value as seen in /proc.

```
#define MAX_NICE          19
#define MIN_NICE          -20
#define NICE_WIDTH        (MAX_NICE – MIN_NICE + 1)
#define MAX_RT_PRIO           100
#define MAX_PRIO          (MAX_RT_PRIO + NICE_WIDTH)
#define DEFAULT_PRIO      (MAX_RT_PRIO + NICE_WIDTH / 2)

#define NICE_TO_PRIO(nice)   ((nice) + DEFAULT_PRIO)
#define PRIO_TO_NICE(prio)   ((prio) - DEFAULT_PRIO)

static inline int task_nice (const struct task_struct *p)
        { return PRIO_TO_NICE((p)->static_prio); }

int task_prio (const struct task_struct *p)
        { return p->prio - MAX_RT_PRIO; }
```

Functions of the scheduler

The main procedures of the scheduling process are **schedule()** and **scheduler_tick()**.

The main task of the procedure **schedule()** is selecting the next process to be executed and handing him the processor. The procedure is invoked by the kernel when it wants to fall asleep or when the process needs to be preempted.

If the newly selected process (**next**) is different from the one previously executed (**prev**), then the context switches.

Simplified version

```
....
if (likely (prev != next)) {
        rq->nr_switches++;
         RCU_INIT_POINTER(rq->curr , next);
         ++*switch_count;
        ...
        rq = context_switch (rq, prev, next, &rf);
```

The scheduler_tick function

The procedure **scheduler_tick()** recalculates priorities and time quanta. It is invoked once every tick to perform some operations related to scheduling.

```
void scheduler_tick (void)
{
        ...
         curr->sched_class->task_tick(rq, curr, 0);
        ...
}
DEFINE_SCHED_CLASS (rt) = {
        ...
        .task_tick = task_tick_fair,   // differ for various sched_class

        ...
}
```

Linux schedulers – overview (to be continued)

O(N), O(1), RSDS

Scheduler developers

Ingo Molnar – official developer of schedulers in the Linux kernel, full-time programmer, employee of RedHat, from Hungary.

Peter Zijlstra – co-maintainer of various Linux kernel subsystems including: the scheduler, locking primitives and performance monitoring, currently employed by Intel where he assists in enabling new hardware features on Linux. (2017)

Con Kolivas – anesthesiologist from Australia, hobbyist, digging in the kernel learned to program, mainly interested in responsiveness and good performance of the scheduler in "home" applications.

Shortcuts ...

- **Scheduler O(n)**
- **Scheduler O(1)**
- PS – Processor Sharing
- RSDL (RSDS) – Rotating Staircase Deadline Scheduler
- SD – Staircase Deadline Scheduler
- **CFS** – Completely Fair Scheduler
- BFS – Brain Fuck Scheduler
- **Deadline** – Deadline (real-time) Scheduler
- MuQSS – The Multiple Queue Skiplist Scheduler
- **EAS** – Energy Aware Scheduling

Chronology of events

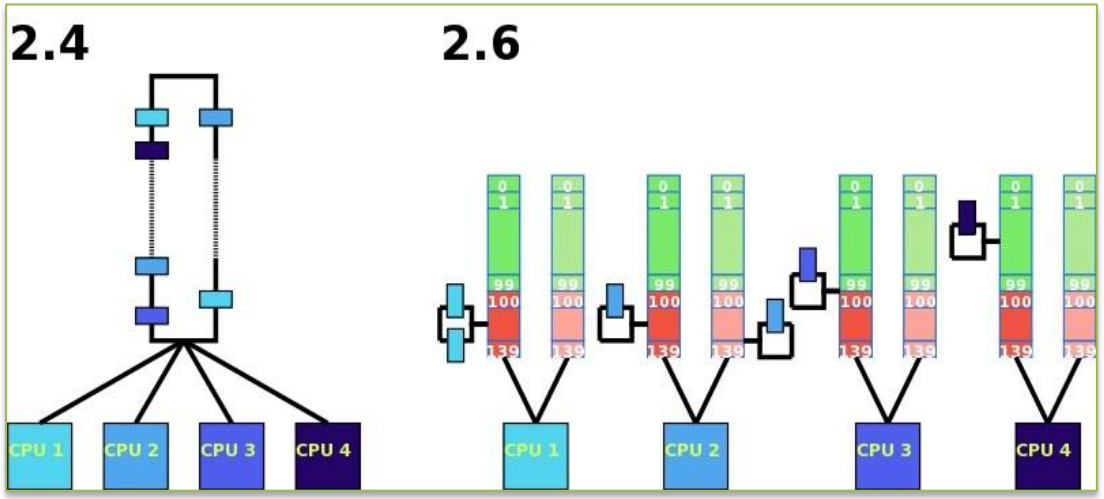
- **Scheduler O(n)** – in the 2.4 kernel versions.
- **Scheduler O(1)** – in the 2.6 kernel versions (up to 2.6.22).
- **Staircase** – designed by Con in 2004 (he became interested in the Linux kernel in 1999) (https://lwn.net/Articles/87729/, Corbet June 2004)
- **RSDL** – developed by Con w 2007, then renamed to **SD**, Linus seemed to agree to include RSDL into the main Linux distribution.
- **CFS** – developed by Ingo in March 2007, almost immediately entered the main distribution of the kernel (2.6.23), scheduler O(log(n)).
- **BFS** – developed by Con in 2009, although he promised he would never look into the Linux code again.
- **Deadline** (real-time) scheduler – 3.14 kernel version, March 2014.
- **MuQSS** – Con comes back in 2016 to address scalability concerns that BFS had with an increasing number of CPUs.
- **EAS** – Energy Aware Scheduling (2019)

What's important when comparing schedulers

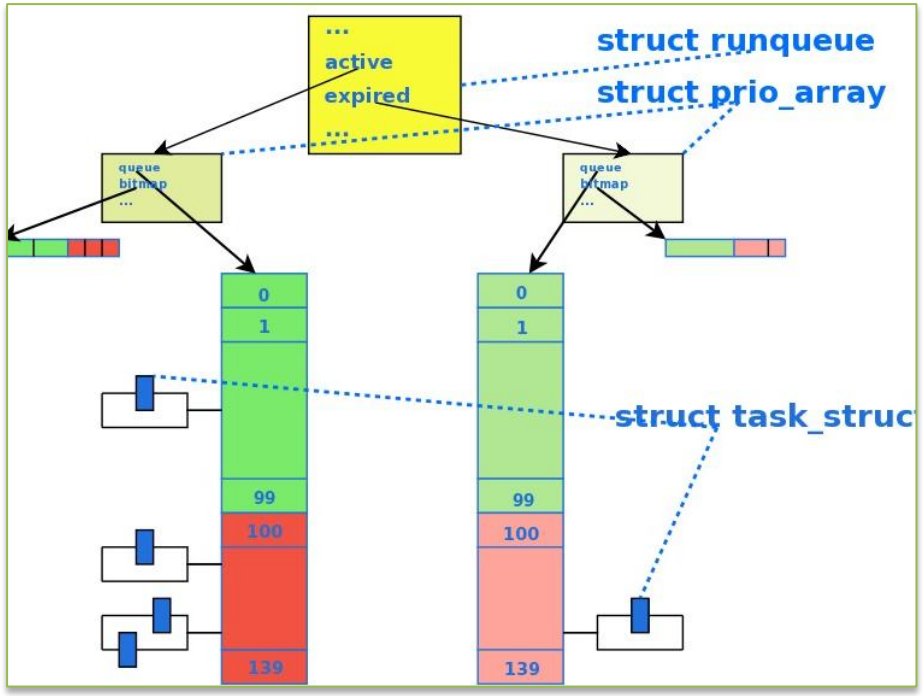
- Should be **maximized**:
  - **CPU utilization** – How saturated the CPU is from 0 to 100%.
  - **Throughput** – The number of processes completed per unit of time.
- Should be **minimized**:
  - **Turnaround time** – The total time taken for a process to complete once started.
  - **Waiting time** – The total time a process spends in the ready queue.
  - **Scheduler latency** – The time between a wakeup signal being sent to a thread on the wait queue and the scheduler executing that thread. Including the dispatch latency – the time taken to handle a context switch, change in user-mode, and jump to starting location.

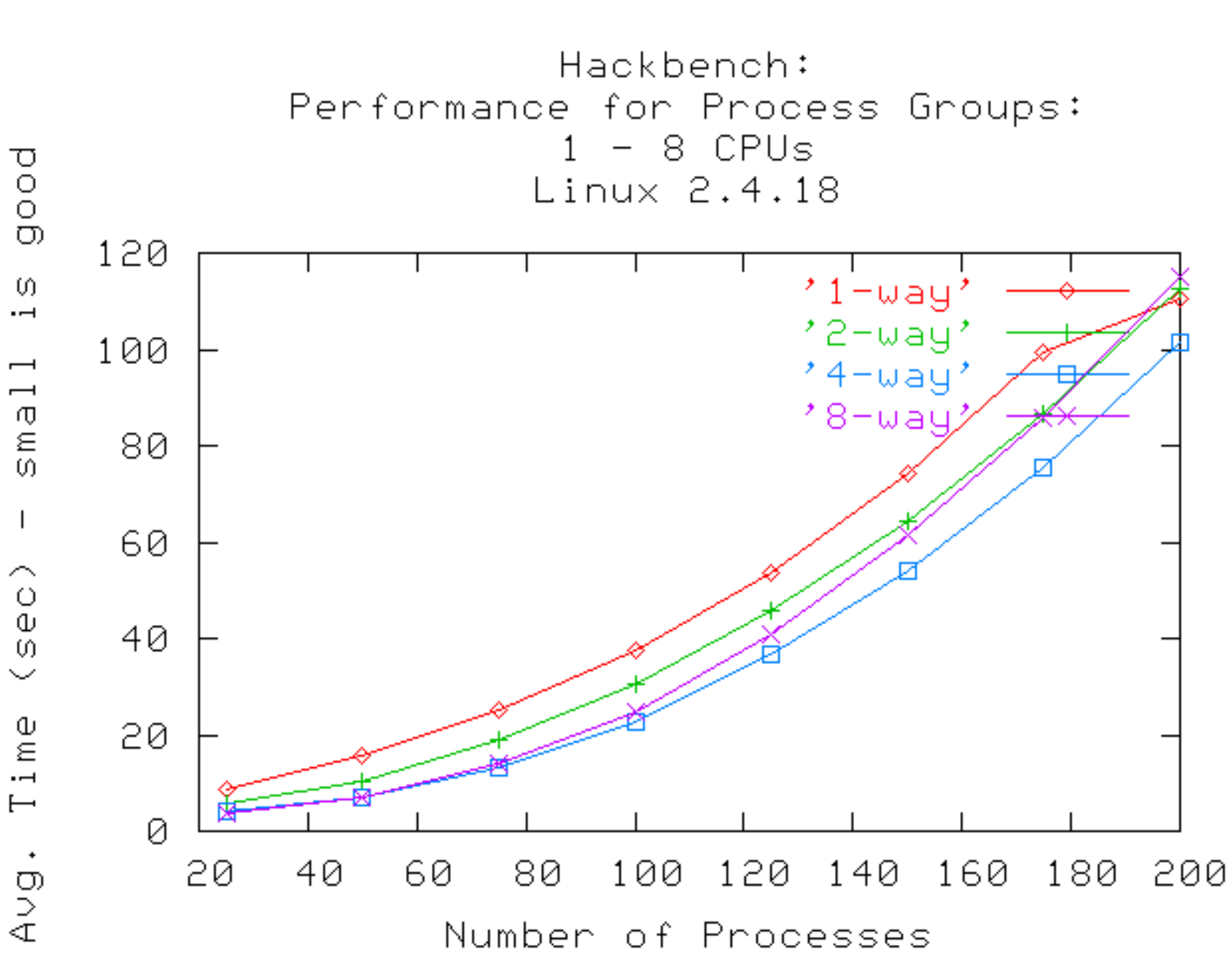
http://cs.unm.edu/~eschulte/classes/cs587/data/bfs-v-cfs_groves-knockel-schulte.pdf

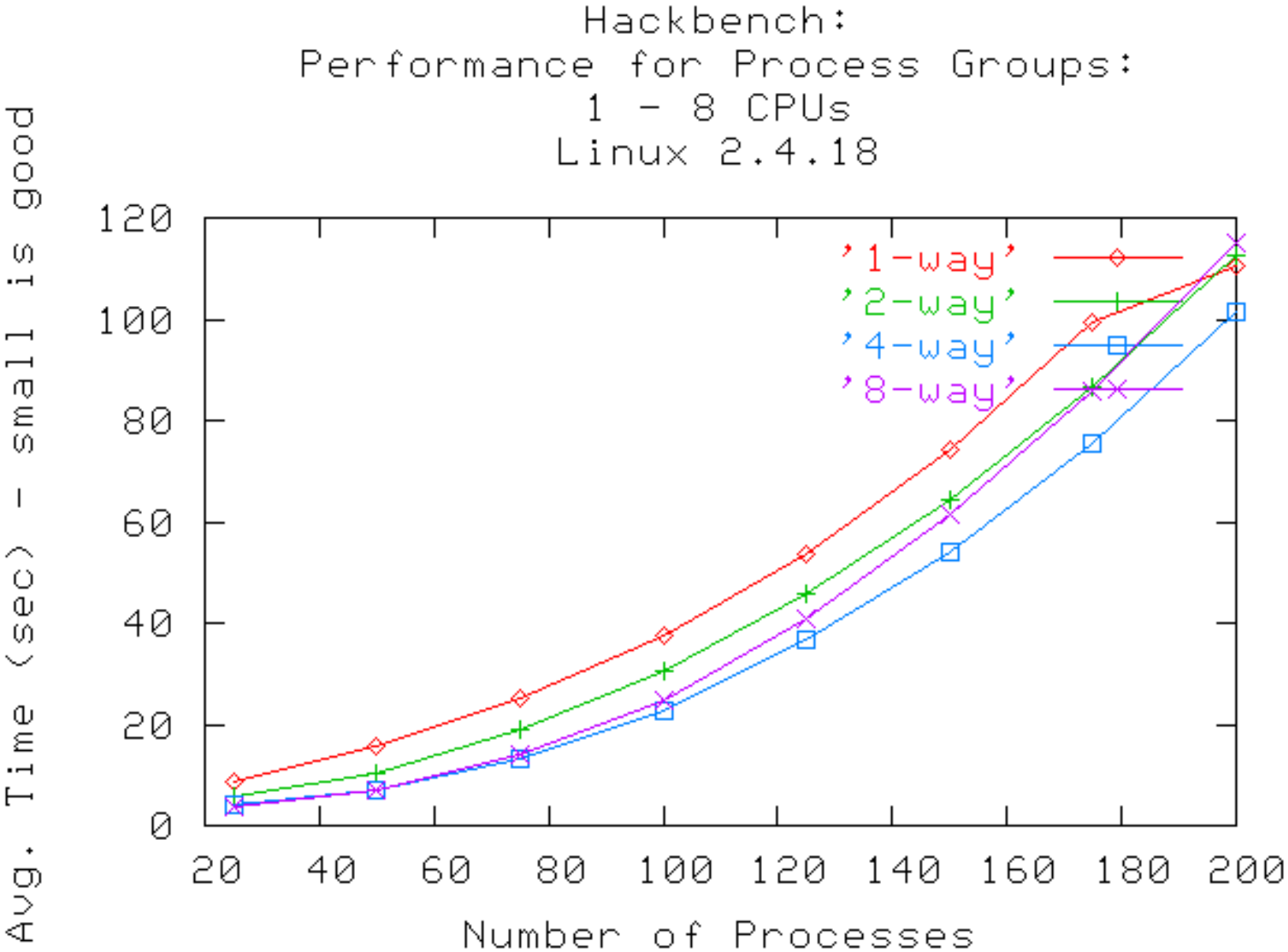
Data structures for schedulers O(n) and O(1)



- Scheduler in 2.6 kernels, up to 2.6.22
- Significant improvements over the 2.4 kernel
  - Separate queues per processor and load balancing
  - Task selection in O(1)
  - Queues switched in O(1)
- Behavior based on numerous heuristics




Hackbench:
Performance for Process Groups:
1 - 8 CPUs
Linux 2.4.18
Avg. Time (sec) - small is good
Number of Processes
'1-way'
'2-way'
'4-way'
'8-way'
0
20
40
60
80
100
120
20
40
60
80
100
120
140
160
180
200


Hackbench:
Performance for Process Groups:
1 - 8 CPUs
Linux 2.4.18
Avg. Time (sec) - small is good
Number of Processes
'1-way'
'2-way'
'4-way'
'8-way'
0
20
40
60
80
100
120
20
40
60
80
100
120
140
160
180
200

The important features of the scheduling algorithm O(1)

The scheduling takes place at time **0 (1)**. Each algorithm ends in a constant time, regardless of the number of processes executed or other parameters.

Scalability in SMP systems is provided. Each processor has its own queue of ready processes and sets up own locks.

It provides a good relation of the process with the processor (so called ***affinity***). The process that was executed on a processor, will continue – if possible – to be executed on the same processor, unless it is necessary to balance the load (because of unequal lengths of the run queues).

It provides good support for **interactive processes**, even in a situation of a high load.

It is **fair** – the processes are not starved, they do not get too unnaturally large time quanta.

It works well at both **low** and **high loads**.

BUT!!!

Priorities and time quanta

- Uneven distribution
  - Task with a priority **100** gets only **2.5% more CPU time** than task with a priority **101**.
  - Task with a priority **138** gets **2 times more CPU time** than task with a priority **139**
  - Task with a priority **119** gets **4 times more CPU time** than task with a priority **120**.
- Conclusion: jobs with the same **priority difference** can have very **different CPU time proportions**.

p→static_prio	100	101	105	110	115	119
task_time_slice(p) (msec)	800	780	700	600	500	420
p→static_prio	120	125	130	135	138	139
task_time_slice(p) (msec)	100	75	59	25	10	5

Priorities and time quanta

- **Batch jobs**

  Batch jobs usually run with a **lower priority**, not to reduce the responsiveness of the user's interactive tasks. However, at **low load**, when batch tasks could be performed efficiently, **low priority** means that they have **short quanta** and often (every 5 ms at nice +10) switch the processor, while it would be more efficient if they were performed in longer quanta.

- **Paradox**

  Interactive processes that need a processor for **shorter time** periods (but more often) receive **long quanta**, and processes that could use the processor effectively receive **short quanta**.

Bonuses and process interactivity

Processes that have obtained a large interactivity bonus can be considered **interactive** and be put back into the active process queue. The **higher** the process **priority**, the **smaller** is the **bonus** which makes the process to be considered as **interactive**.

TASK_INTERACTIVE(p)

bonus to prio	-5	-4	-3	-2	-1	0	1	2	3	4	5
nice -20 (100)	T	T	T	T	T	T	T	T	T	N	N
nice -10 (110)	T	T	T	T	T	T	T	N	N	N	N
nice 0 (120)	T	T	T	T	N	N	N	N	N	N	N
nice 10 (130)	T	T	N	N	N	N	N	N	N	N	N
nice 19 (139)	N	N	N	N	N	N	N	N	N	N	N

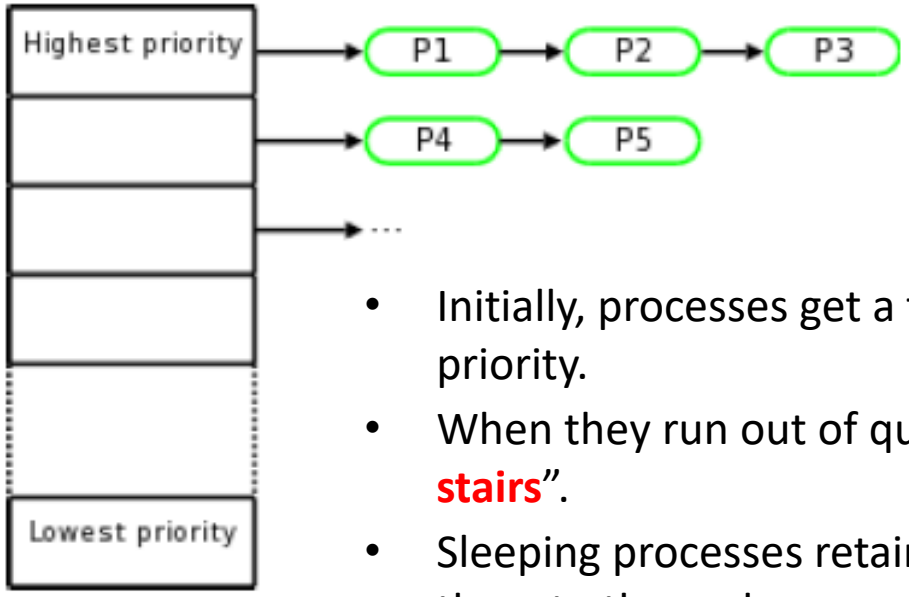
When several processes get a big bonus, **they dominate the processor**, because they are considered interactive and after their quanta expire they are put back into the active queue. In addition, other processes unable to reach the processor are considered to be sleeping too much and do not get a priority bonus.

(S. Takeuchi, 2007) In an extreme test configuration (200 processes on one i386 processor or 400 on a dual-core ia64, nice 0, 2.6.20 kernel) **4 processes** (on 2 processors: **8 processes**) took up **98%** of the processor time.

Summary

- The scheduler performs all operations in O(1). There are many different calculations, heuristics, patches hidden behind this constant.
- Heuristics work fine in most cases, but work very badly in some cases.
- To fix this, additional heuristics are added for special cases.
- There are many more hooks and such special cases in the code.
- Heuristics complicate both the concept and the code, and are not very universal and treat processes very unevenly.
- Is this possible to develop a scheduler that performs better and is **fair to processes**?

Rotating Staircase Deadline Scheduler Con Kolivas, 2007



- It is based on the old scheduler **with two process queues**.
- Process queues form the "**stairs**" of **priorities**.

- Initially, processes get a time quantum (**RR_INTERVAL**, default 6 ms) on their static priority.
- When they run out of quantum, they get a new one on a lower priority – „**going down the stairs**”.
- Sleeping processes retain their quanta and if they wake up in the same era, they can use them to the end.

- Each **priority** also has its own **execution time limit**. When it runs out, all processes from this "step" go down a step, no matter how many quanta they have left – **minor rotation**.
- Processes that have used all their priorities are queued in an **inactive** array on their static priority. When the active array is empty**, major rotation** occurs.
- Time limits on each priority guarantee that the era will end in a **limited time** and processes will **not starve**.
- Processes are treated **fairly**, there are **no heuristics** that favor interactive processes, except that when they return from sleep they will most likely have more time quantum.

Linus Torvalds, March 2007

In 2007, Con Kolivas's Staircase Deadline scheduler was about to become an official scheduler in the Linux kernel.

Linus Torvalds said in March 2007:

> I agree, partly because it's obviously been getting rave reviews so far, but mainly because it looks like you can think about behaviour a lot better, something that was always very hard with the interactivity boosters with process state history.
>
> I'm not at all opposed to this, but we do need:
>
> - To not do it at this stage in the stable kernel
> - to let it sit in –mm for at least a short while
> - and generally more people testing more loads.
>
> So as long as the generic concerns above are under control, I'll happily try something like this if it can be merged early in a merge window.

Linux schedulers – overview
continued at the next lecture

CFS, BFS, Deadline, MuQSS, etc.

What’s new in process scheduling?