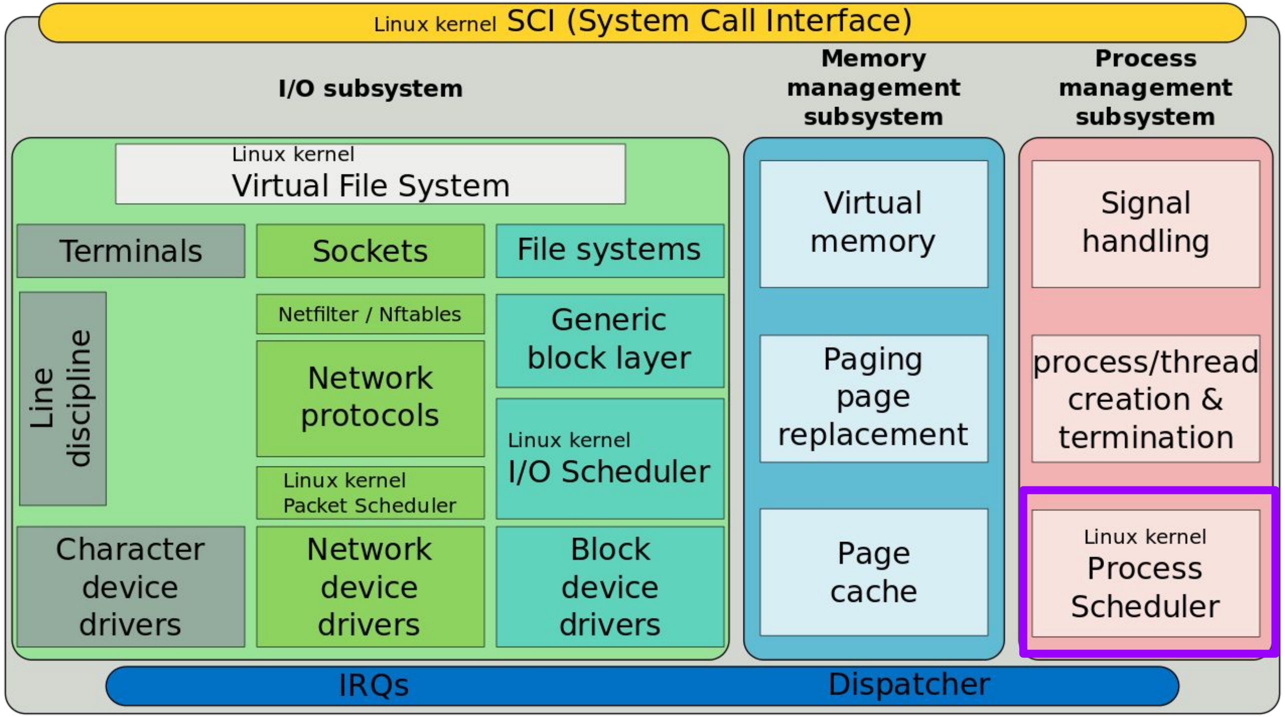
ELEC 424/553
Mobile & Embedded Systems

Lecture 6 - How Does Linux Schedule Processes? (2)

The saga continues…

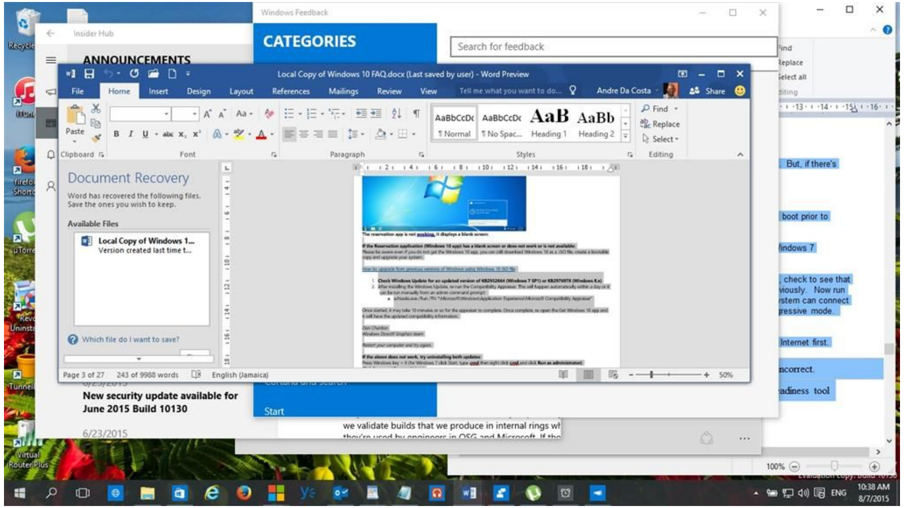
Google Slides Link:
https://docs.google.com/presentation/d/12W4xdbzLT6lrQzBd98b0p9yFqOZMo9iRd4hxi3lBweM/edit?usp=sharing

Where Are We In The Linux Kernel?



Scheduling Processes

- Goal: Run a bunch of applications
- **Multitasking** makes it look like OS is simultaneously executing more than 1 application/process
  - Capable of alternating between processes



Andre Da Costa, “How to: manage running programs and virtual desktops using Task View in Windows 10”. URL: https://answers.microsoft.com/en-us/insider/forum/all/how-to-manage-running-programs-and-virtual/17d068b7-5e4a-4351-a019-afa528a81538

Robert Love, *Linux Kernel Development*, 3rd Edition

Two Modern Scheduling Concerns

1. Priority
2. Timeslice

Robert Love, *Linux Kernel Development*, 3rd Edition

Priority & Nature of Scheduler Classes

Priority & Nature of Scheduler Classes

Highest Priority → Lowest Priority

Figure & info from: Viresh Kumar, “Fixing SCHED_IDLE”. URL: https://lwn.net/Articles/805317/
Denzel Washing photo modified from: Peter Swanson, “Unstoppable”. URL: http://armchairaudience.blogspot.com/2011/02/unstoppable.html

Priority & Nature of Scheduler Classes

Literally halts
everything
Unstoppable

Process
migration

Highest Priority → Lowest Priority

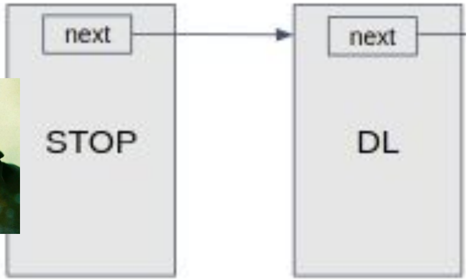
Priority & Nature of Scheduler Classes

Literally halts everything *Unstoppable*

Process migration

Hard deadline processes soonest first

Video



Highest Priority → Lowest Priority

Figure & info from: Viresh Kumar, “Fixing SCHED_IDLE”. URL: https://lwn.net/Articles/805317/
Denzel Washing photo modified from: Peter Swanson, “Unstoppable”. URL: http://armchairaudience.blogspot.com/2011/02/unstoppable.html

Priority & Nature of Scheduler Classes

Literally halts everything
Unstoppable

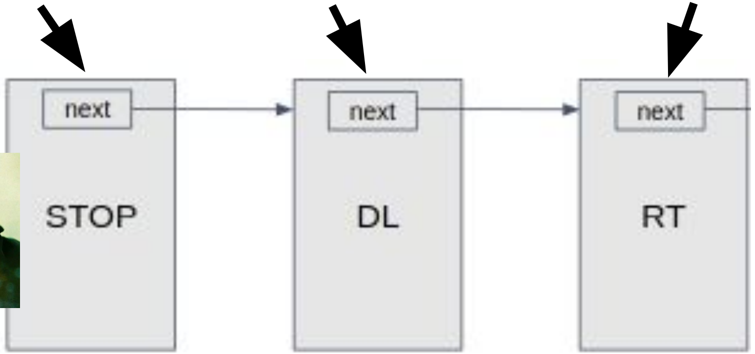
Process migration

Hard deadline processes soonest first

Video

Realtime
Latency-important processes

Interrupt request (IRQ)



Highest Priority → Lowest Priority

Figure & info from: Viresh Kumar, “Fixing SCHED_IDLE”. URL: https://lwn.net/Articles/805317/
Denzel Washing photo modified from: Peter Swanson, “Unstoppable”. URL: http://armchairaudience.blogspot.com/2011/02/unstoppable.html

Priority & Nature of Scheduler Classes

Literally halts everything
Unstoppable

Process migration

Hard deadline processes soonest first

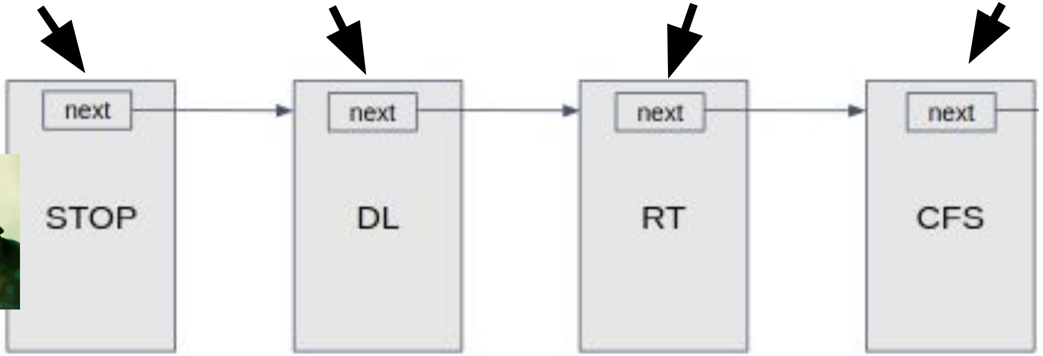
Video

Realtime
Latency-important processes

Interrupt request (IRQ)

Our normal process scheduler

Most processes (e.g. shell)



Highest Priority → Lowest Priority

Figure & info from: Viresh Kumar, “Fixing SCHED_IDLE”. URL: https://lwn.net/Articles/805317/
Denzel Washing photo modified from: Peter Swanson, “Unstoppable”. URL: http://armchairaudience.blogspot.com/2011/02/unstoppable.html

Priority & Nature of Scheduler Classes

Literally halts everything
Unstoppable

Process migration

Hard deadline processes soonest first

Video

Realtime
Latency-important processes

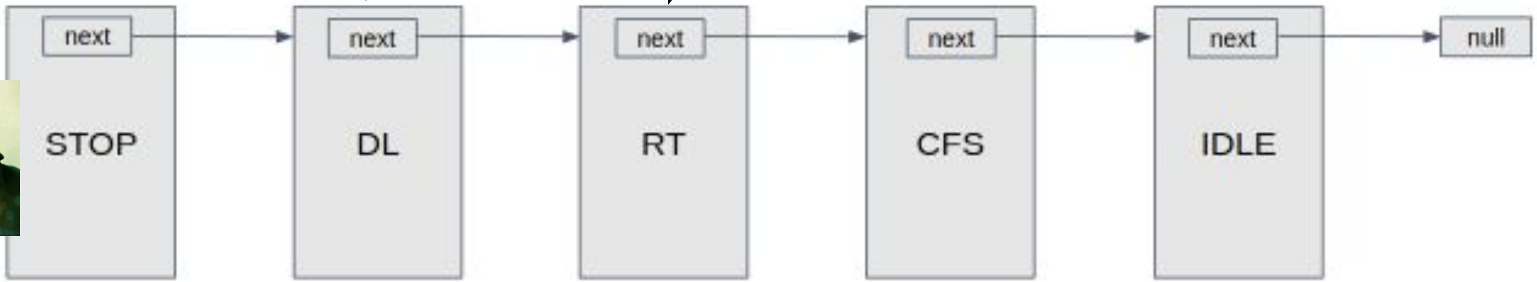
Interrupt request (IRQ)

Our normal process scheduler

Most processes (e.g. shell)

Nothing else runnable
Burn CPU cycles

Power saving



Highest Priority → Lowest Priority

Figure & info from: Viresh Kumar, “Fixing SCHED_IDLE”. URL: https://lwn.net/Articles/805317/
Denzel Washing photo modified from: Peter Swanson, “Unstoppable”. URL: http://armchairaudience.blogspot.com/2011/02/unstoppable.html

Scheduler Classes

/ kernel / sched / core.c

```c
 * WARNING: must be called with preemption disabled!
 */
static void __sched notrace __schedule(bool preempt)
{
	struct task_struct *prev, *next;
	unsigned long *switch_count;
	unsigned long prev_state;
	struct rq_flags rf;
	struct rq *rq;
	int cpu;

	cpu = smp_processor_id();
	rq = cpu_rq(cpu);
	prev = rq->curr;
```

/ kernel / sched / sched.h

```c
/*
 * This is the main, per-CPU runqueue data structure.
 *
 * Locking rule: those places that want to lock multiple runqueues
 * (such as the load balancing or the thread migration code), lock
 * acquire operations must be ordered by ascending &runqueue.
 */
struct rq {
	/* runqueue lock: */
	raw_spinlock_t		__lock;

	/*
	 * nr_running and cpu_load should be in the same cacheline bec
	 * remote CPUs use both these fields when doing load calculat
```

```c
	unsigned int		uclamp_flags;
#define UCLAMP_FLAG_IDLE 0x01
#endif

	struct cfs_rq		cfs;
	struct rt_rq		rt;
	struct dl_rq		dl;
```

Linux 5.14.2 Source via Bootlin Elixir Cross Referencer
Left: https://elixir.bootlin.com/linux/v5.14.2/source/kernel/sched/core.c#L5904
Right: https://elixir.bootlin.com/linux/v5.14.2/source/kernel/sched/sched.h#L912

Kernel Calls Scheduler via `schedule()`

- Independent of scheduler classes
- Asks scheduler of class with greatest priority for process that should be executed
  - (if class has process ready to execute)

/ kernel / sched / core.c

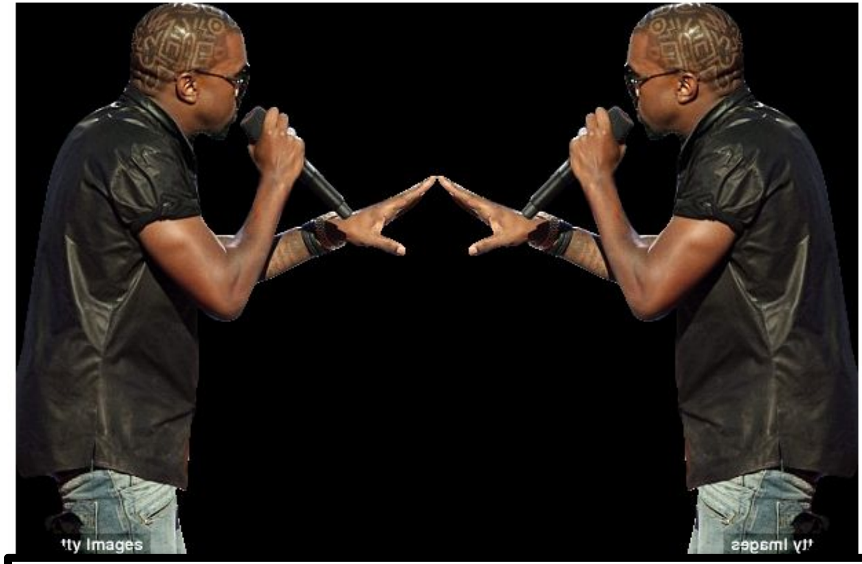
All sy

```c
asmlinkage __visible void __sched schedule(void)
{
        struct task_struct *tsk = current;

        sched_submit_work(tsk);
        do {
                preempt_disable();
                __schedule(false);
                sched_preempt_enable_no_resched();
        } while (need_resched());
        sched_update_worker(tsk);
}
EXPORT_SYMBOL(schedule);
```

Linux Source via Bootlin Elixir Cross Referencer
https://elixir.bootlin.com/linux/latest/source/kernel/sched/core.c#L6098

Robert Love, *Linux Kernel Development*, 3rd Edition

Kernel Calls Scheduler via `schedule()`



We want to prevent this from happening

https://knowyourmeme.com/photos/292972-kanye-interrupts-imma-let-you-finish

sched / core.c

```
asmlinkage __visible void __sched schedule(void)
{
        struct task_struct *tsk = current;

        sched_submit_work(tsk);
        do {
                preempt_disable();
                __schedule(false);
                sched_preempt_enable_no_resched();
        } while (need_resched());
        sched_update_worker(tsk);
}
EXPORT_SYMBOL(schedule);
```

Linux Source via Bootlin Elixir Cross Referencer
https://elixir.bootlin.com/linux/latest/source/kernel/sched/core.c#L6098

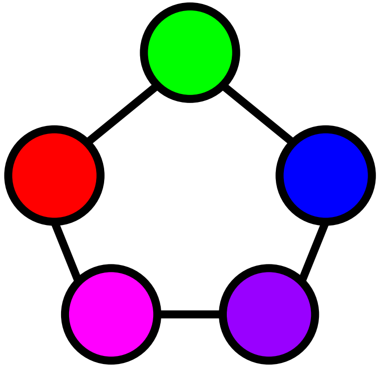
Robert Love, *Linux Kernel Development*, 3rd Edition

A Brief History of Linux Schedulers

M. Jones, “Inside the Linux 2.6 Completely Fair Scheduler”. URL: https://developer.ibm.com/tutorials/l-completely-fair-scheduler/
Robert Love, *Linux Kernel Development*, 3rd Edition

A Brief History of Linux Schedulers

1.2: Circular Queue

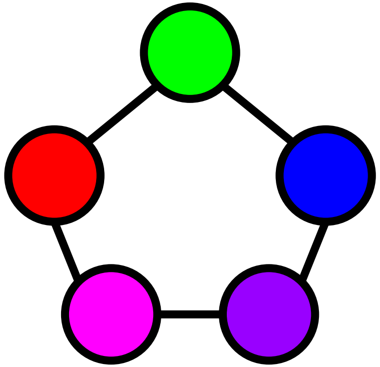


Round-robin
Simple, fast

M. Jones, “Inside the Linux 2.6 Completely Fair Scheduler”. URL: https://developer.ibm.com/tutorials/l-completely-fair-scheduler/
Robert Love, *Linux Kernel Development*, 3rd Edition

A Brief History of Linux Schedulers

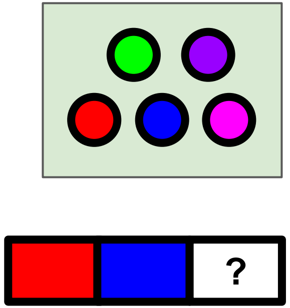
1.2: Circular Queue



Round-robin
Simple, fast

2.4: O(n)

Which next?

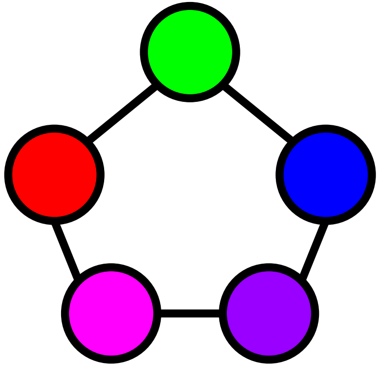


Check all processes
Scalability issues

M. Jones, “Inside the Linux 2.6 Completely Fair Scheduler”. URL: https://developer.ibm.com/tutorials/l-completely-fair-scheduler/
Robert Love, *Linux Kernel Development*, 3rd Edition

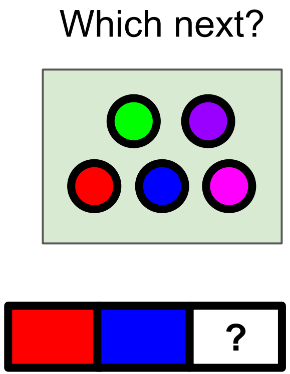
A Brief History of Linux Schedulers

1.2: Circular Queue



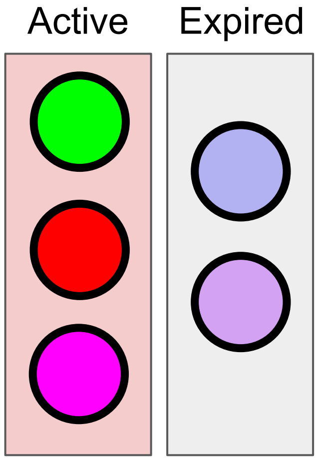
Round-robin
Simple, fast

2.4: O(n)



Check all processes
Scalability issues

2.6: O(1)

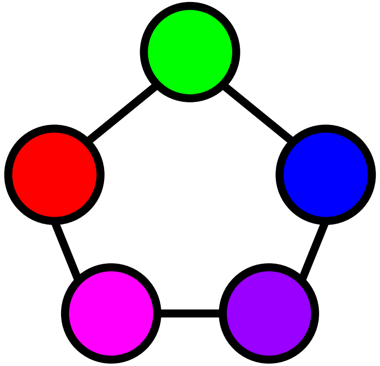


Constant time
Arrays

M. Jones, "Inside the Linux 2.6 Completely Fair Scheduler". URL: https://developer.ibm.com/tutorials/l-completely-fair-scheduler/
Robert Love, *Linux Kernel Development*, 3rd Edition

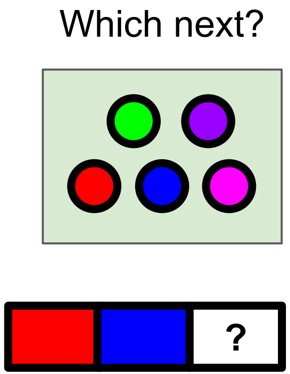
A Brief History of Linux Schedulers

1.2: Circular Queue



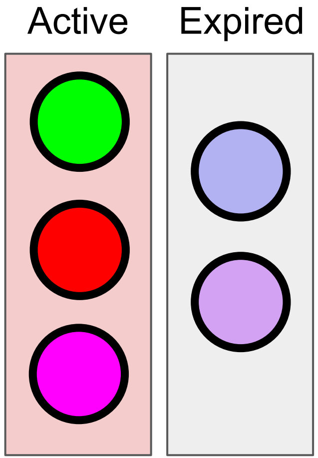
Round-robin
Simple, fast

2.4: O(n)



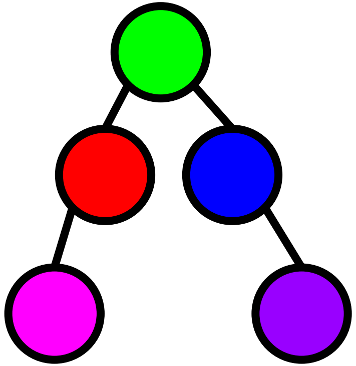
Check all processes
Scalability issues

2.6: O(1)



Constant time
Arrays

2.6.23: CFS



Red-black tree
Virtual runtime

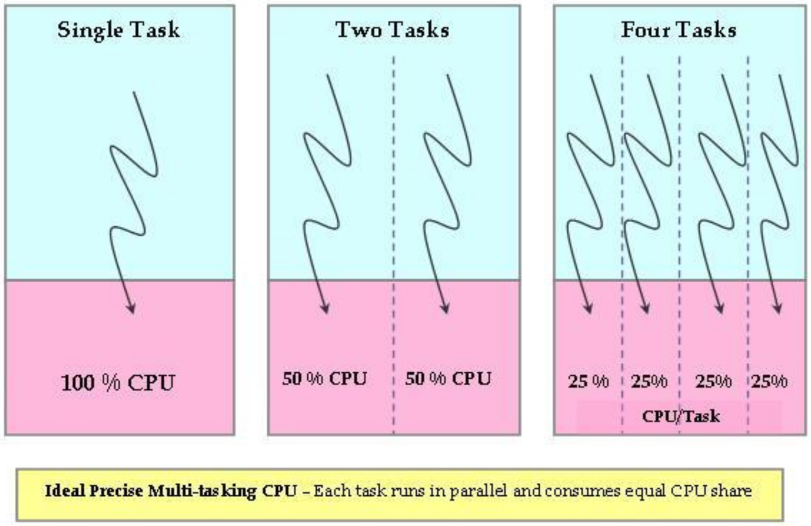
M. Jones, “Inside the Linux 2.6 Completely Fair Scheduler”. URL: https://developer.ibm.com/tutorials/l-completely-fair-scheduler/
Robert Love, *Linux Kernel Development*, 3rd Edition

CFS Has Arrived

https://www.looper.com/189632/the-tragic-real-life-story-behind-the-fast-and-the-furious-films/

The Idea Behind Linux's Completely Fair Scheduler (CFS)

- Just imagine a perfect processor!
- n processes that can execute
- Split processor's schedule into 1/n length blocks
- Drive schedule down to infinitesimal length of time
- Could say that processes were executed using (1/n)*100% of the processor's power
- "Perfect multitasking"

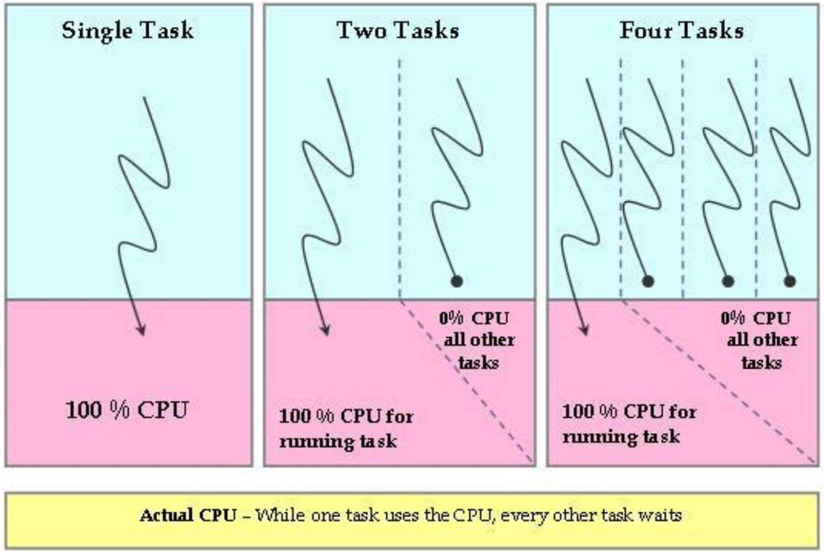


Chandandeep Singh Pabla, "Completely Fair Scheduler". URL: https://www.linuxjournal.com/node/10267

Robert Love, *Linux Kernel Development*, 3rd Edition

Reality Sinks In

- Processor can only run one process at a time!
- Rapidly alternating processes (preempting) would be terrible
- Instead, set a schedule period and give equal “virtual time” to processes
- By *simply* redefining time, we become *fair*



Chandandeep Singh Pabla, “Completely Fair Scheduler”. URL: https://www.linuxjournal.com/node/10267

Robert Love, *Linux Kernel Development*, 3rd Edition

Math Behind CFS: Target Latency & Timeslice

- **Target latency**: Total amount of time established by CFS to allow each task to get a turn
  - `sysctl_sched_latency` (ms)
- Processes given weights based on `nice` (see table)
- $$timeslice = (\,target\ latency) \times \frac{(\,process\ weight)}{(\,sum\ of\ \ all\ process\ weights)}$$

nice	-20	-19	-18	-17	-16	-15	-14	-13	-12	-11
weight	88761	71755	56483	46273	36291	29154	23254	18705	14949	11916
nice	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
weight	9548	7620	6100	4904	3906	3121	2501	1991	1586	1277
nice	0	1	2	3	4	5	6	7	8	9
weight	1024	820	655	526	423	335	272	215	172	137
nice	10	11	12	13	14	15	16	17	18	19
weight	110	87	70	56	45	36	29	23	18	15

Nice-to-weight conversion

Jinkyu Koo, “Linux kernel scheduler”. URL: https://helix979.github.io/jkoo/post/os-scheduler/

Robert Love, *Linux Kernel Development*, 3rd Edition
Jinkyu Koo, “Linux kernel scheduler”. URL: https://helix979.github.io/jkoo/post/os-scheduler/

Math Behind CFS (2): Virtual Runtime

- **Virtual runtime**, stored as `vruntime`, is warped version of physical runtime (wall-time)
- Warping based on weight (priority)
- Higher priority (nice < 0):
  - vruntime < physical runtime
- Lower priority (nice > 0):
  - vruntime > physical runtime

nice	-20	-19	-18	-17	-16	-15	-14	-13	-12	-11
weight	88761	71755	56483	46273	36291	29154	23254	18705	14949	11916
nice	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
weight	9548	7620	6100	4904	3906	3121	2501	1991	1586	1277
nice	0	1	2	3	4	5	6	7	8	9
weight	1024	820	655	526	423	335	272	215	172	137
nice	10	11	12	13	14	15	16	17	18	19
weight	110	87	70	56	45	36	29	23	18	15

Nice-to-weight conversion

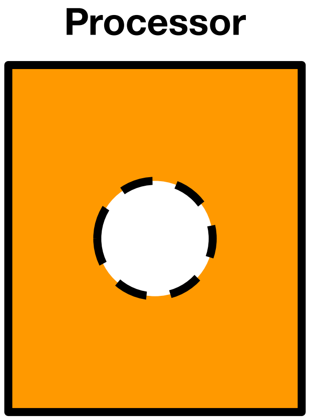
Jinkyu Koo, “Linux kernel scheduler”. URL: https://helix979.github.io/jkoo/post/os-scheduler/

$$vruntime = (\,physical\ runtime\,) \cdot \frac{1024}{weight}$$

Jinkyu Koo, “Linux kernel scheduler”. URL: https://helix979.github.io/jkoo/post/os-scheduler/

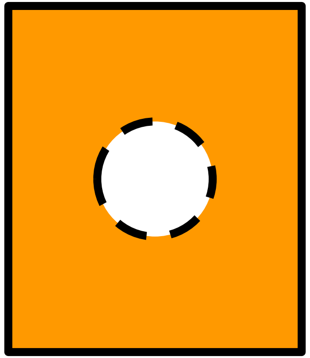
CFS Example

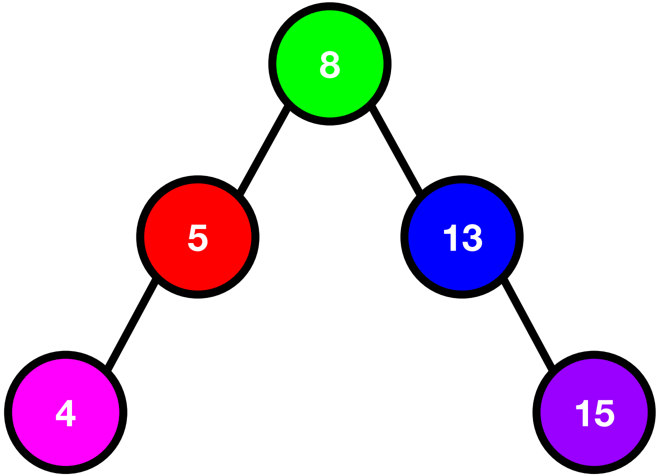
CFS Example



CFS Example

Processor

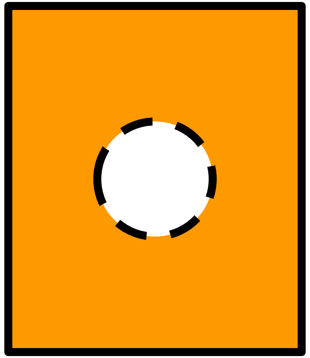




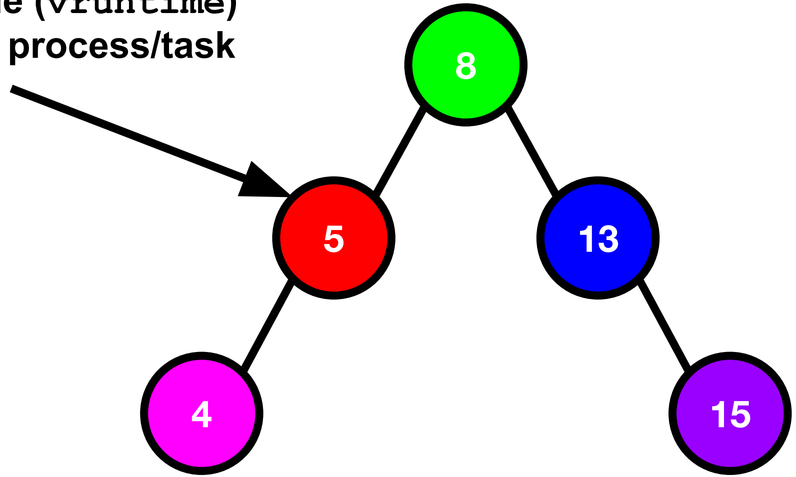
CFS Run Queue
Red-Black Tree

CFS Example

Processor



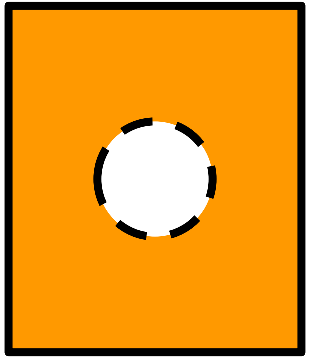
Virtual Runtime (`vruntime`)
for a runnable process/task



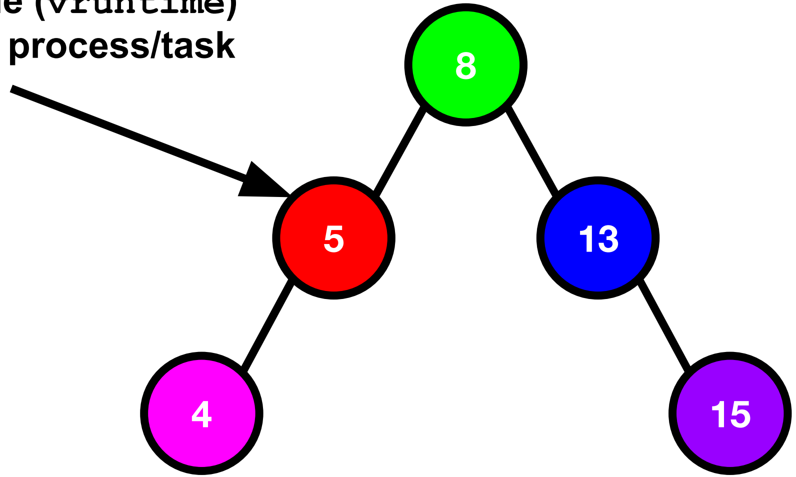
CFS Run Queue
Red-Black Tree

CFS Example

Processor



Virtual Runtime (`vruntime`)
for a runnable process/task

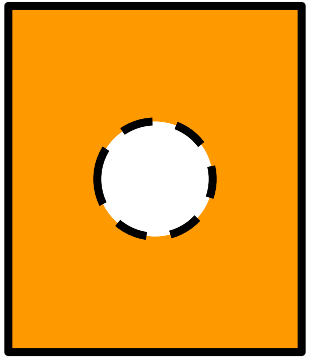


CFS Run Queue
Red-Black Tree

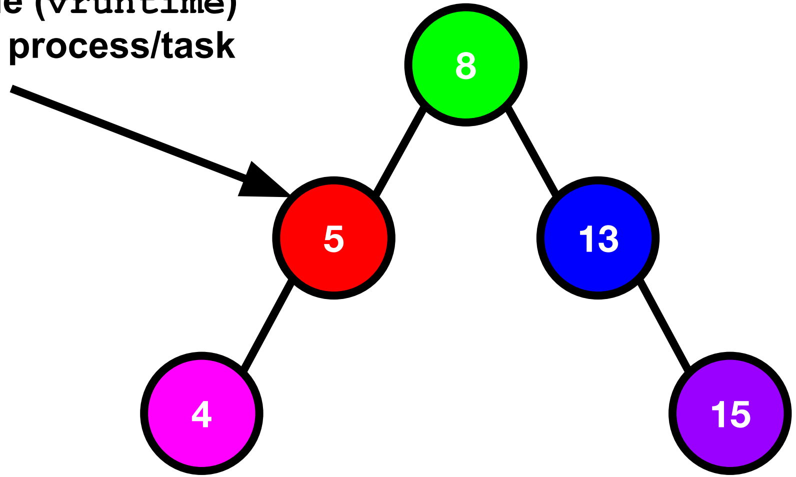
Processor Virtual Runtime Timeline

CFS Example

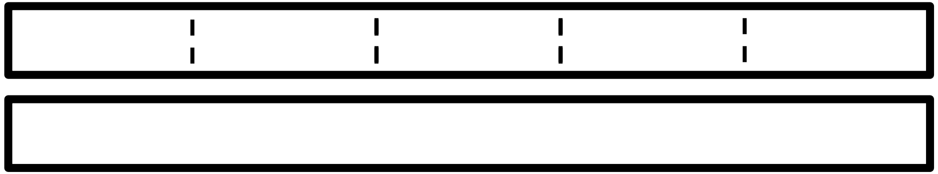
Processor



Virtual Runtime (`vruntime`)
for a runnable process/task



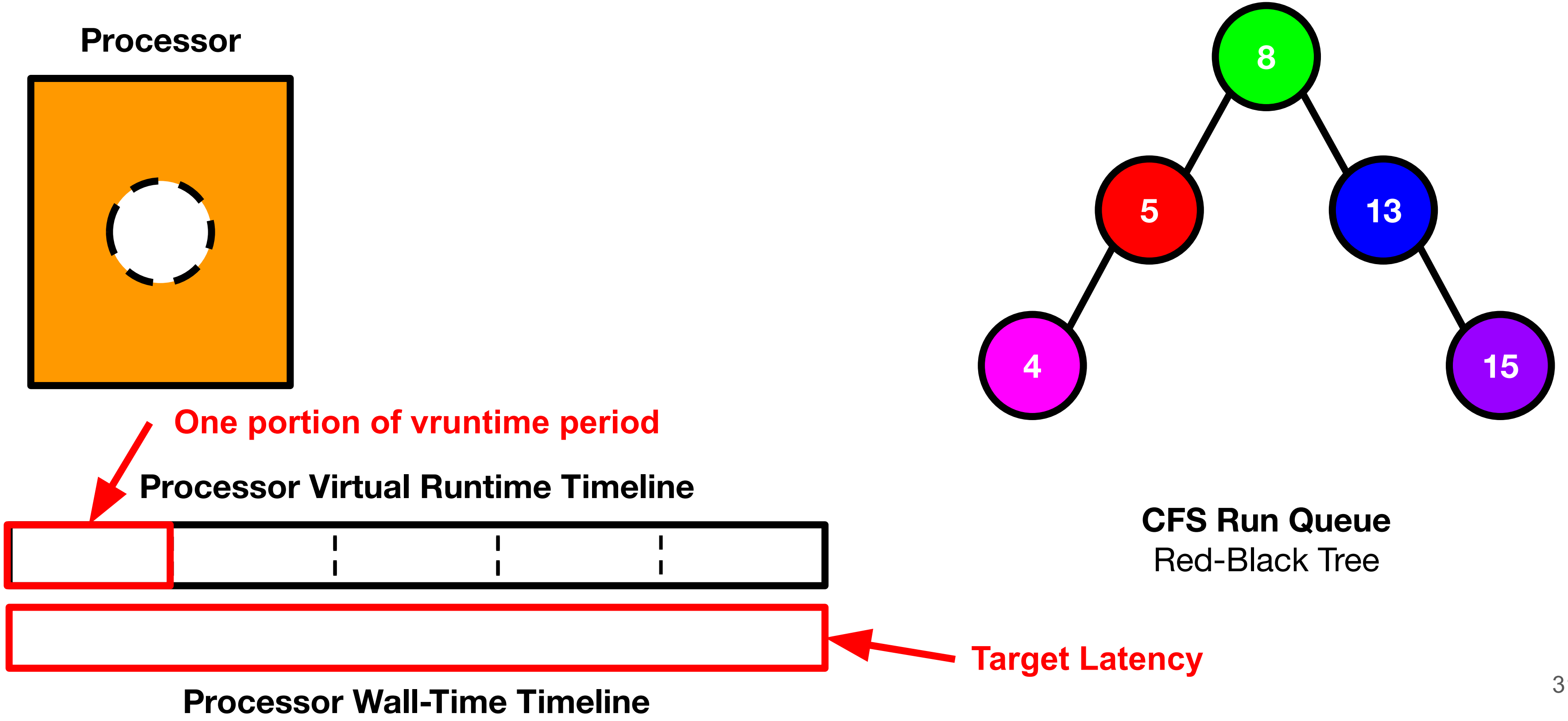
Processor Virtual Runtime Timeline



Processor Wall-Time Timeline

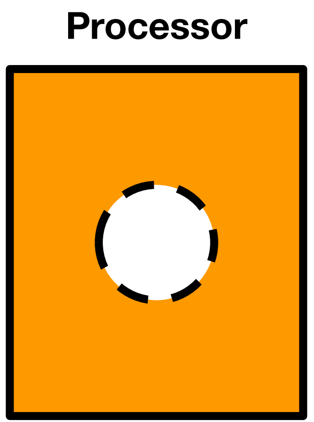
CFS Run Queue
Red-Black Tree

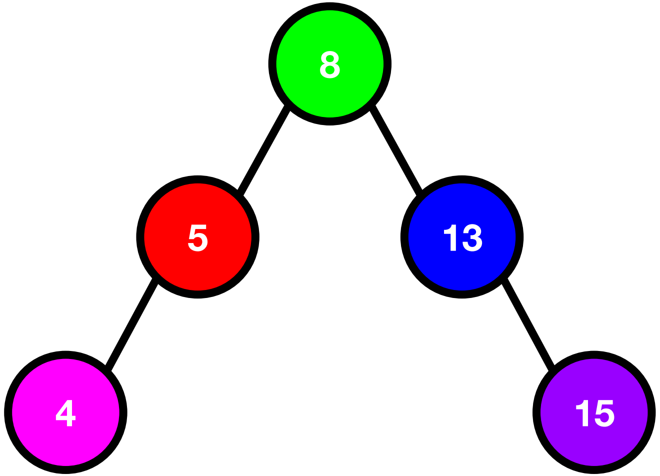
CFS Example



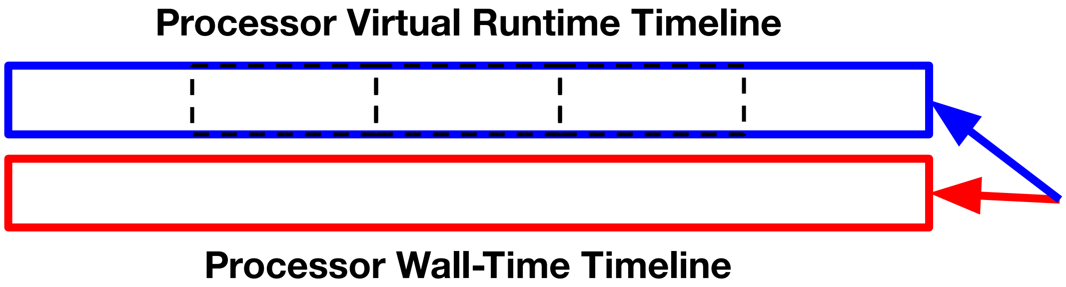
Processor
One portion of vruntime period
Processor Virtual Runtime Timeline
Target Latency
Processor Wall-Time Timeline
8
5
13
4
15
CFS Run Queue
Red-Black Tree

CFS Example





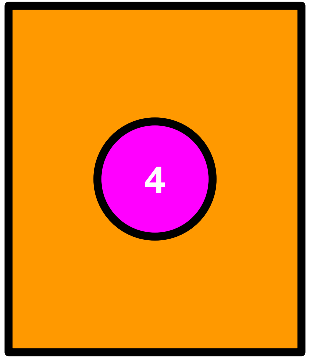
CFS Run Queue
Red-Black Tree



Can be different lengths of time

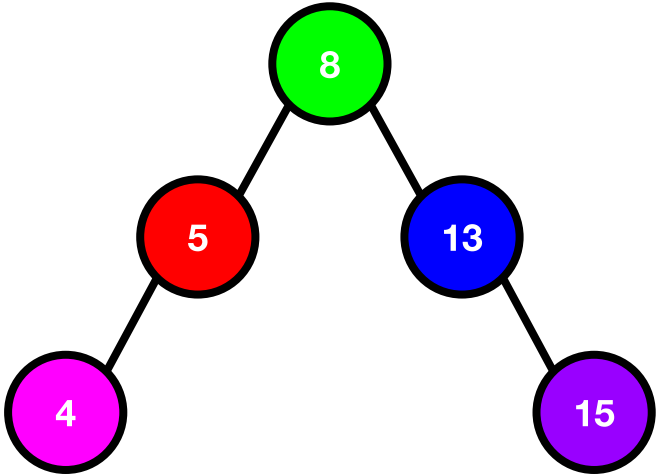
CFS Example

Processor



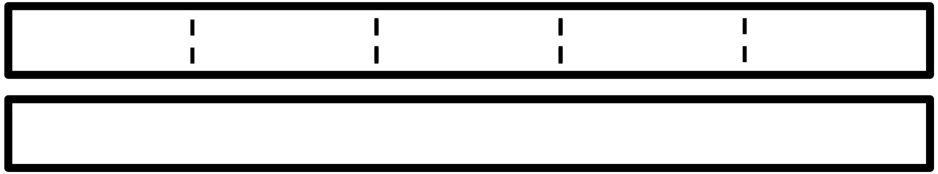
We chose this process because it has the lowest cumulative `vruntime`

An advantage of a RB tree is that the process with the lowest vruntime will always be to the bottom left



CFS Run Queue
Red-Black Tree

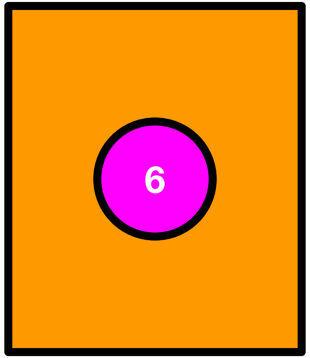
Processor Virtual Runtime Timeline



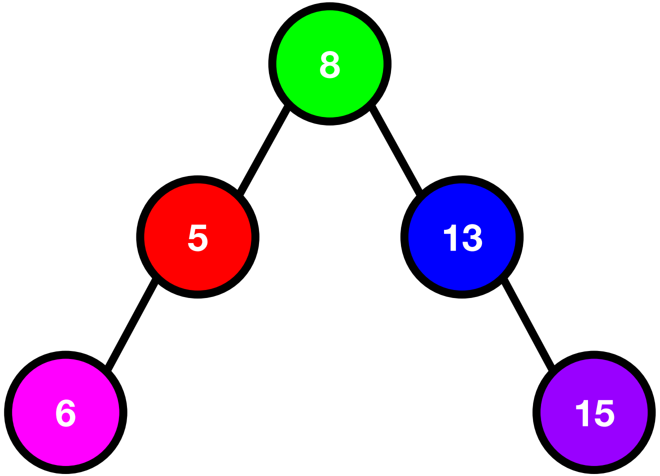
Processor Wall-Time Timeline

CFS Example

Processor

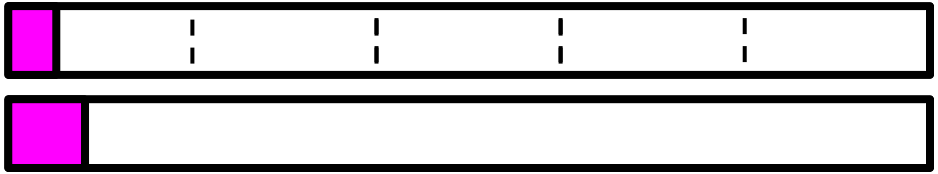


Less nice (nice<0)
Higher priority



CFS Run Queue
Red-Black Tree

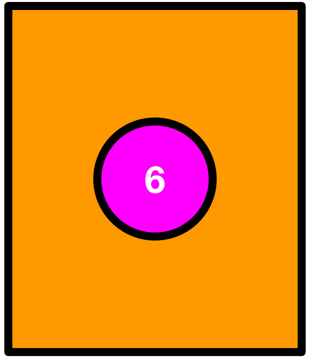
Processor Virtual Runtime Timeline



Processor Wall-Time Timeline

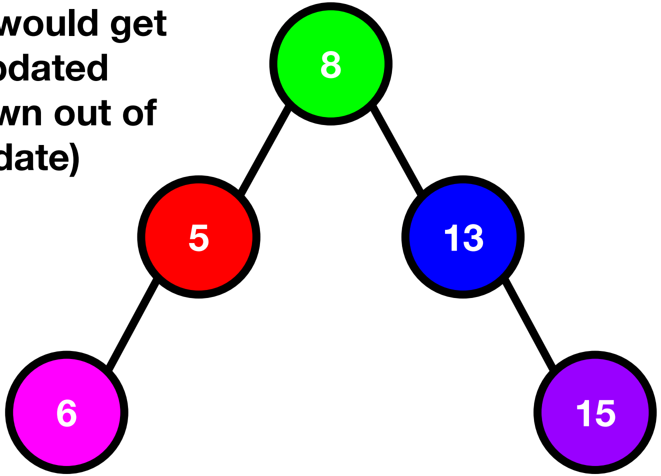
CFS Example

Processor



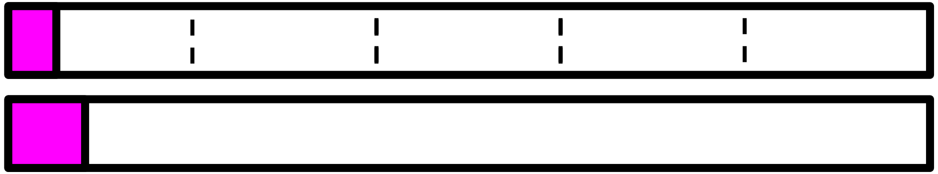
Less nice (nice<0)
Higher priority

Tree would get updated (shown out of date)



CFS Run Queue
Red-Black Tree

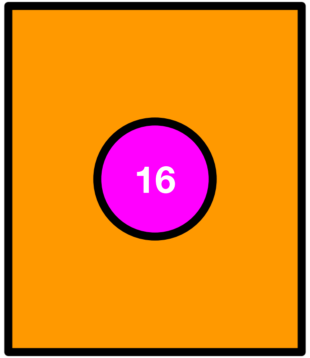
Processor Virtual Runtime Timeline



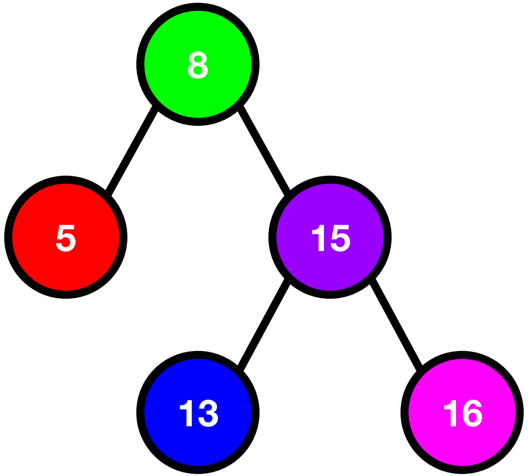
Processor Wall-Time Timeline

CFS Example

Processor

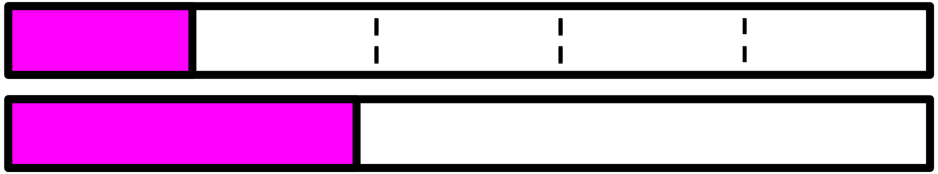


Less nice (nice<0)
Higher priority



CFS Run Queue
Red-Black Tree

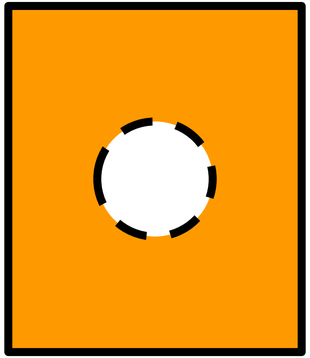
Processor Virtual Runtime Timeline



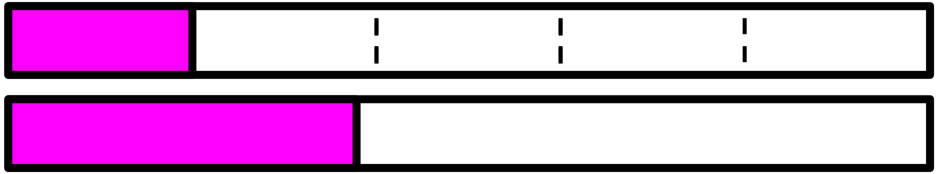
Processor Wall-Time Timeline

CFS Example

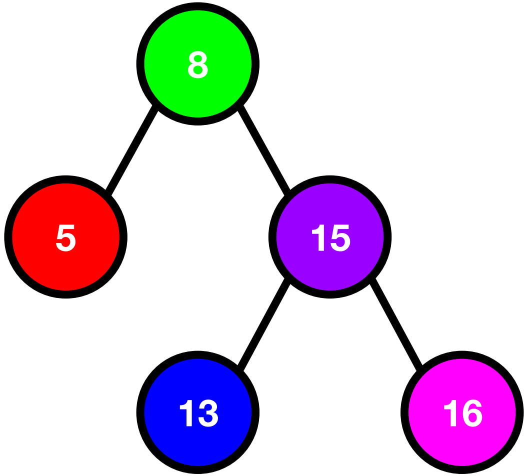
Processor



Processor Virtual Runtime Timeline

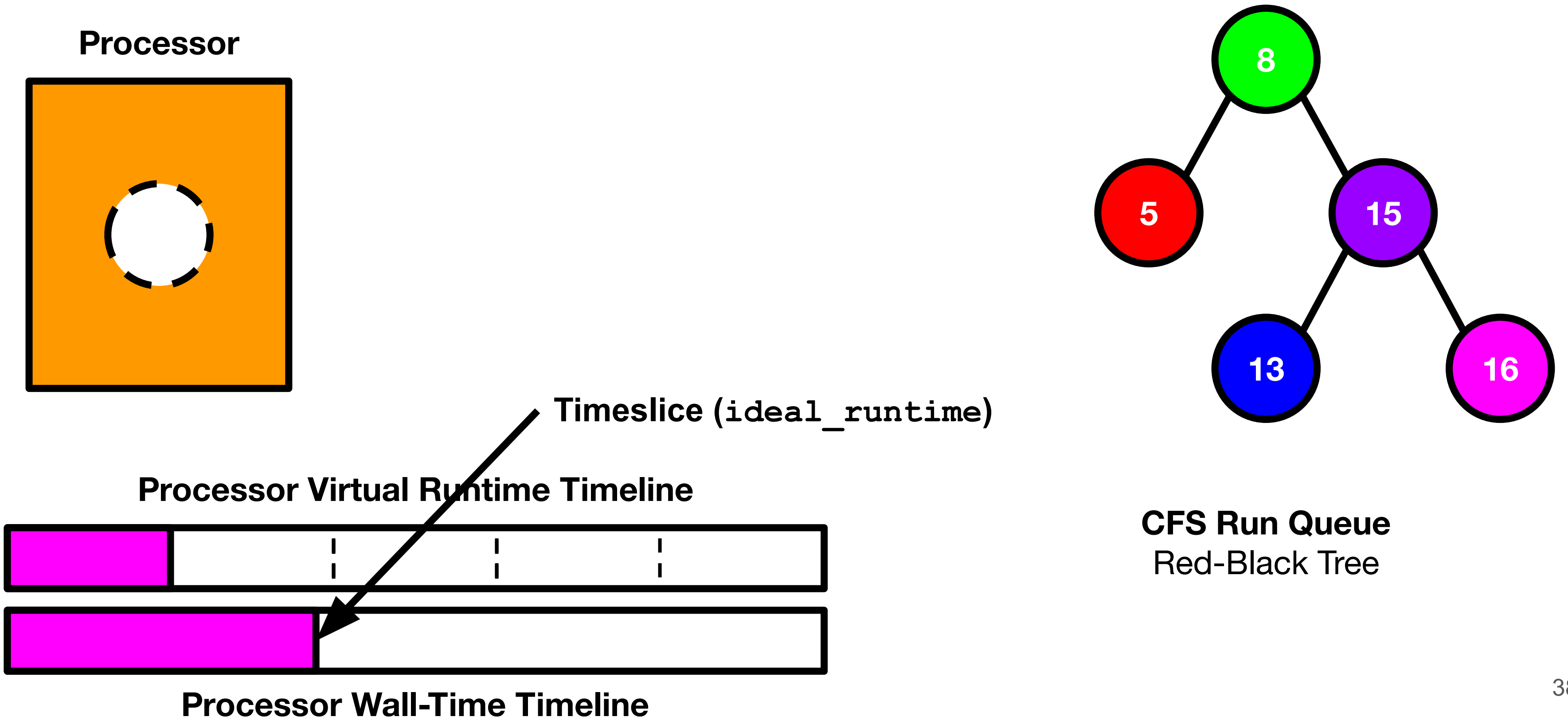


Processor Wall-Time Timeline



CFS Run Queue
Red-Black Tree

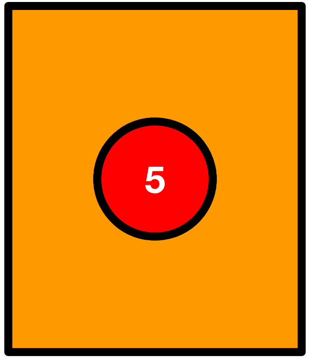
CFS Example



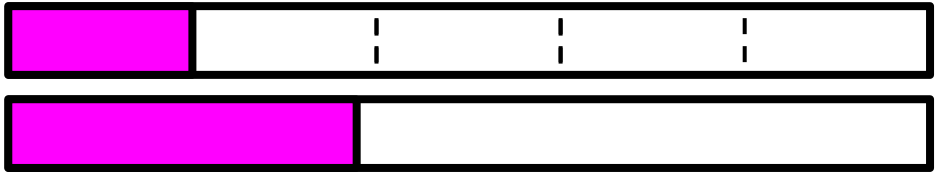
Processor
Timeslice (ideal_runtime)
Processor Virtual Runtime Timeline
Processor Wall-Time Timeline
8
5
15
13
16
CFS Run Queue
Red-Black Tree

CFS Example

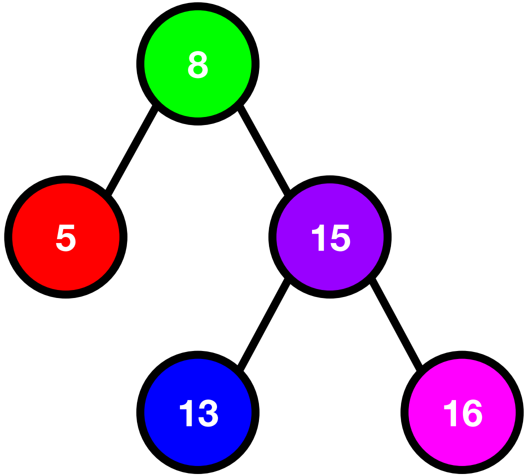
Processor



Processor Virtual Runtime Timeline



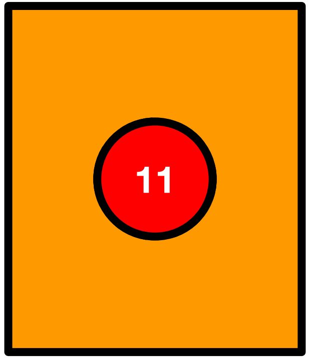
Processor Wall-Time Timeline



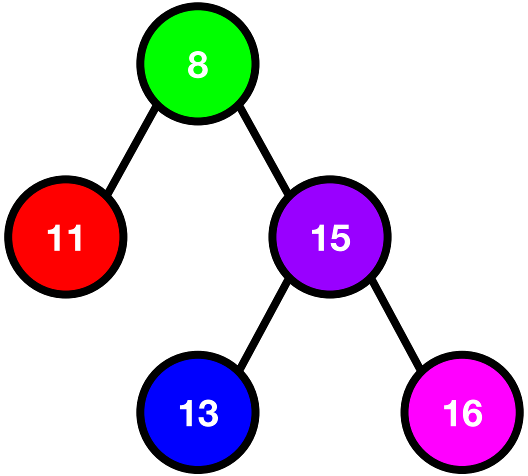
CFS Run Queue
Red-Black Tree

CFS Example

Processor

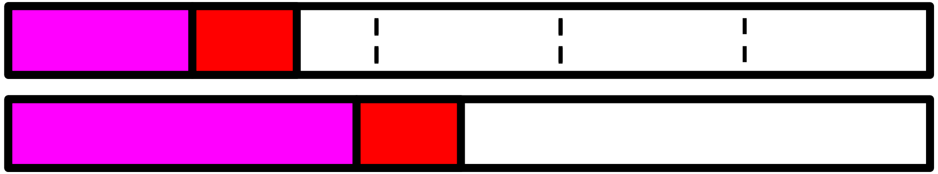


Default nice (nice=0)
Normal priority



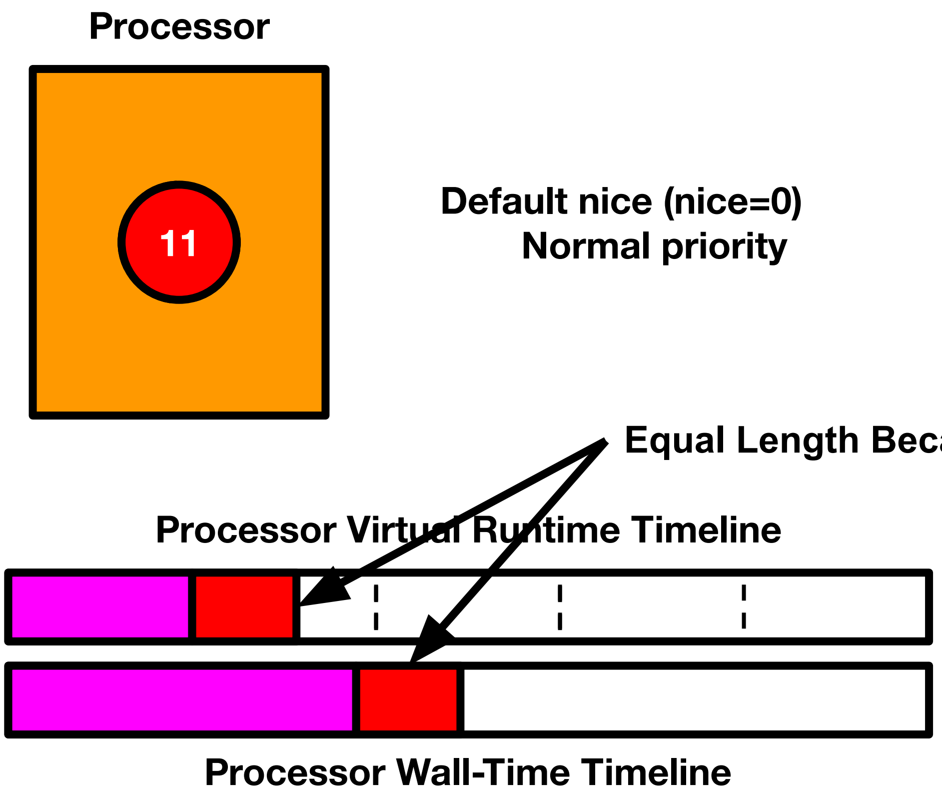
CFS Run Queue
Red-Black Tree

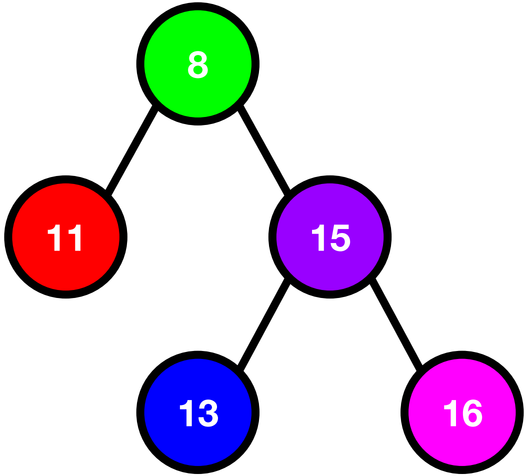
Processor Virtual Runtime Timeline



Processor Wall-Time Timeline

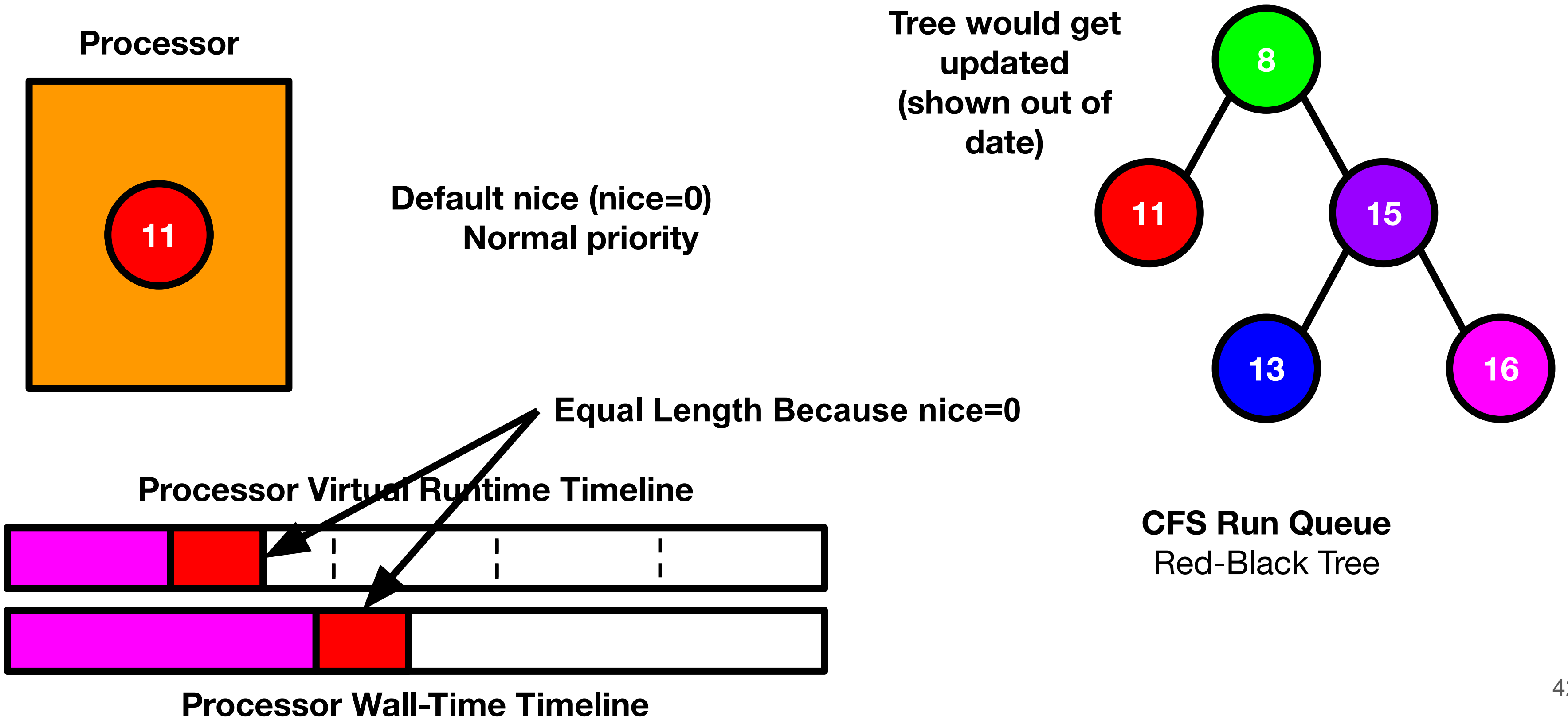
CFS Example



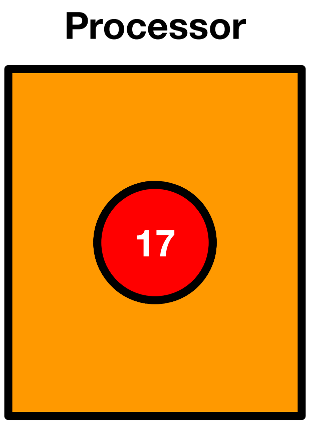


CFS Run Queue
Red-Black Tree

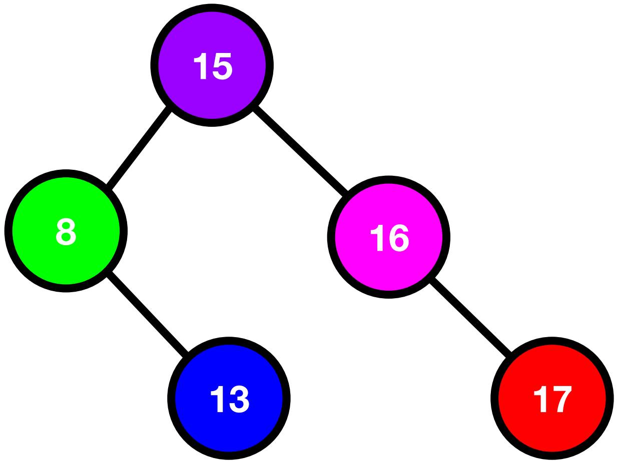
CFS Example


Processor
11
Default nice (nice=0)
Normal priority
Tree would get updated (shown out of date)
8
11
15
13
16
Equal Length Because nice=0
Processor Virtual Runtime Timeline
Processor Wall-Time Timeline
CFS Run Queue
Red-Black Tree

CFS Example



Default nice (nice=0)
Normal priority



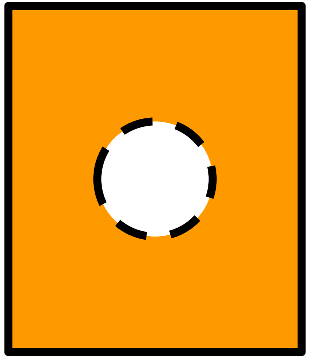
CFS Run Queue
Red-Black Tree

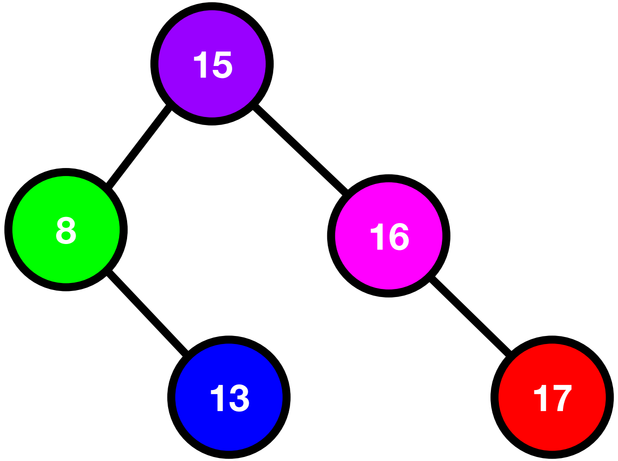
Processor Virtual Runtime Timeline

Processor Wall-Time Timeline

CFS Example

Processor



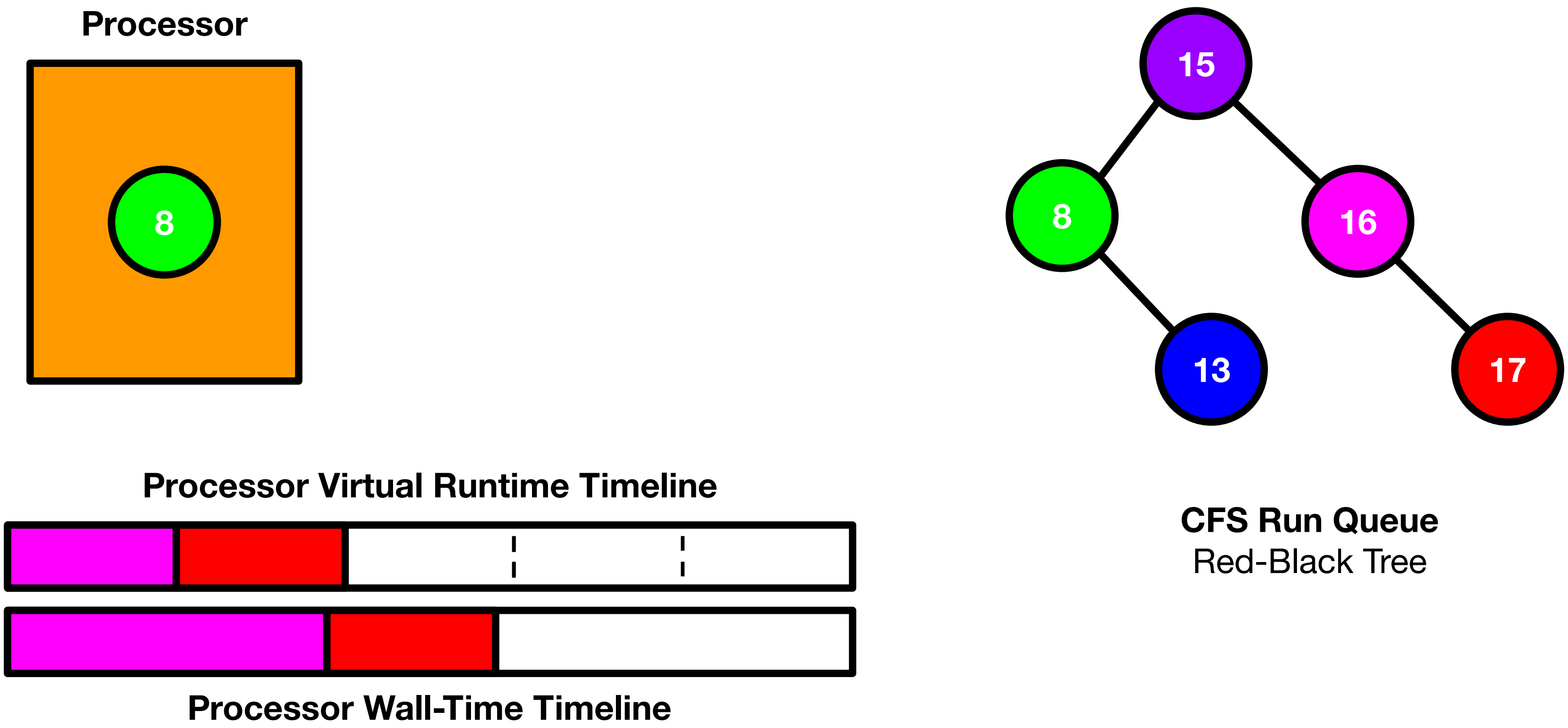


CFS Run Queue
Red-Black Tree

Processor Virtual Runtime Timeline

Processor Wall-Time Timeline

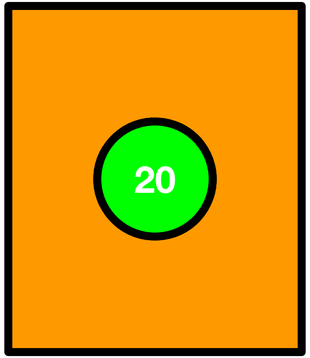
CFS Example



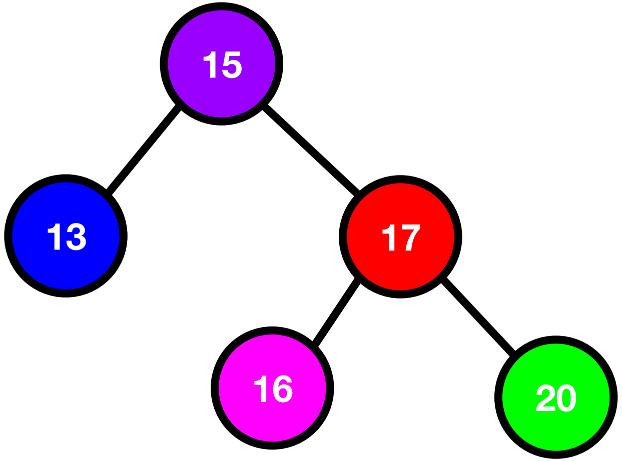
CFS Run Queue
Red-Black Tree

CFS Example

Processor



More nice (nice>0)
Lower priority

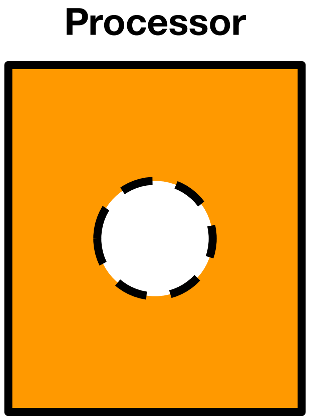


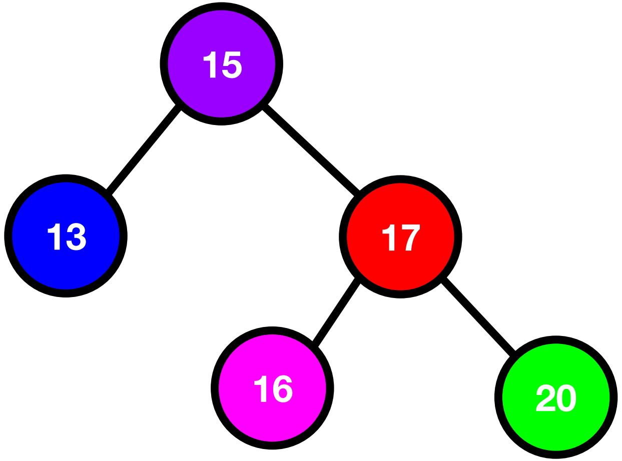
CFS Run Queue
Red-Black Tree

Processor Virtual Runtime Timeline

Processor Wall-Time Timeline

CFS Example



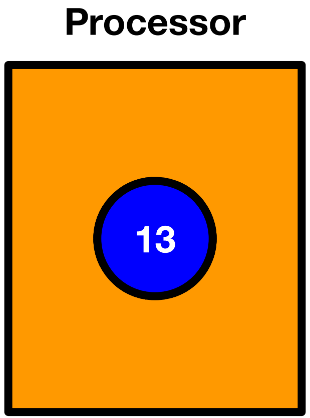


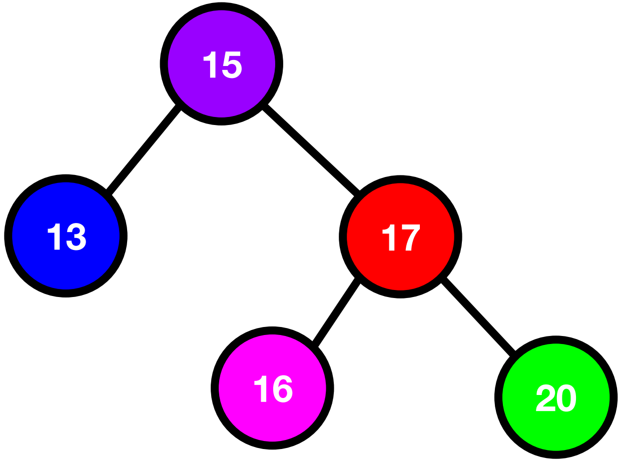
CFS Run Queue
Red-Black Tree

Processor Virtual Runtime Timeline

Processor Wall-Time Timeline

CFS Example





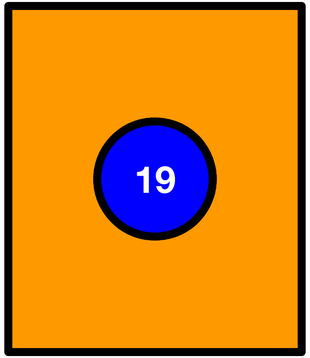
CFS Run Queue
Red-Black Tree

Processor Virtual Runtime Timeline

Processor Wall-Time Timeline

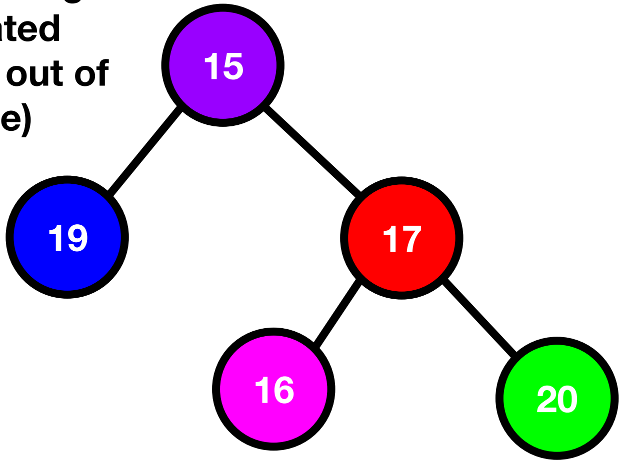
CFS Example

Processor



More more nice (nice>>0)
Very Low priority

Tree would get updated (shown out of date)



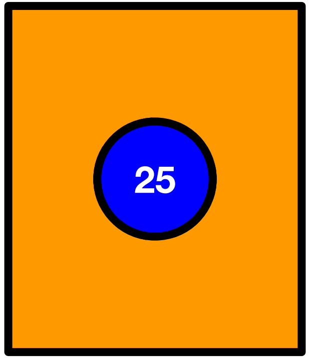
CFS Run Queue
Red-Black Tree

Processor Virtual Runtime Timeline

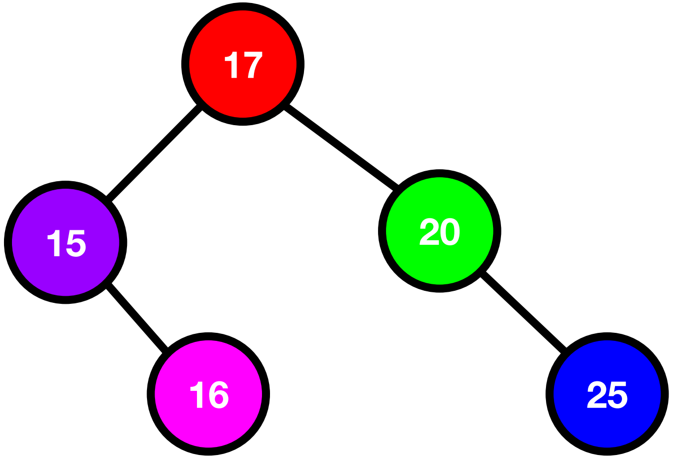
Processor Wall-Time Timeline

CFS Example

Processor



More more nice (nice>>0)
Very Low priority



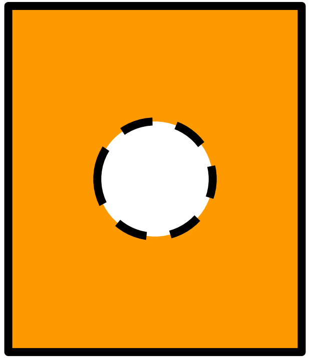
CFS Run Queue
Red-Black Tree

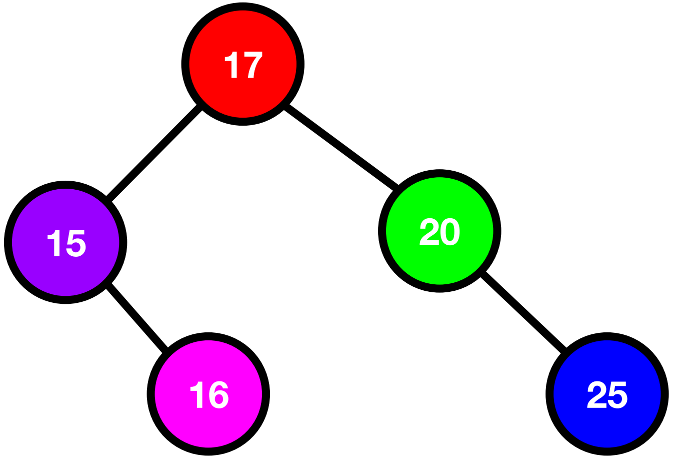
Processor Virtual Runtime Timeline

Processor Wall-Time Timeline

CFS Example

Processor



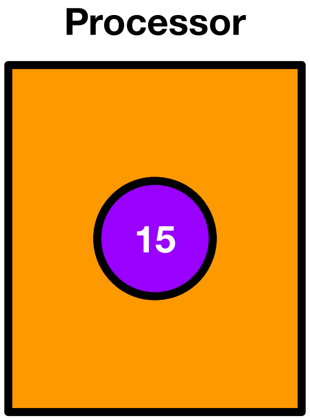


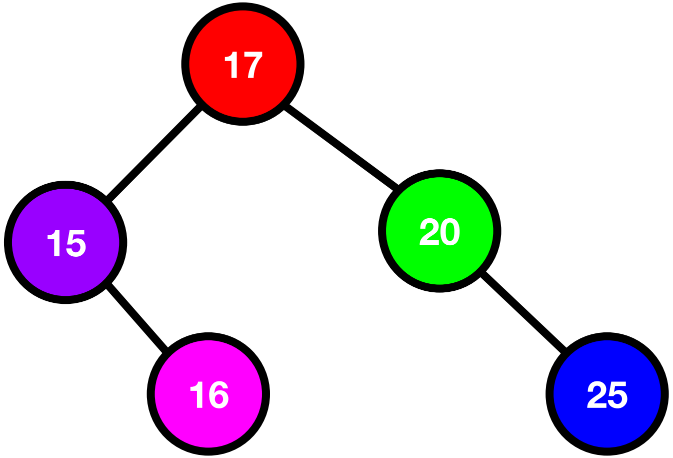
Processor Virtual Runtime Timeline

Processor Wall-Time Timeline

CFS Run Queue
Red-Black Tree

CFS Example





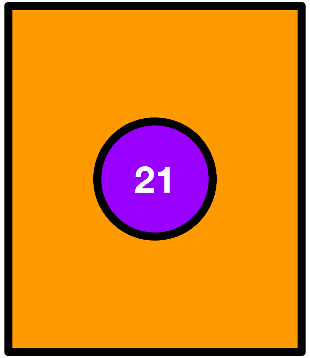
CFS Run Queue
Red-Black Tree

Processor Virtual Runtime Timeline

Processor Wall-Time Timeline

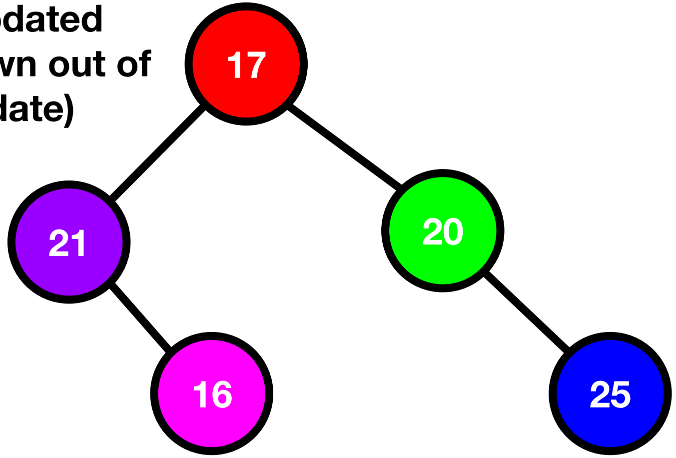
CFS Example

Processor



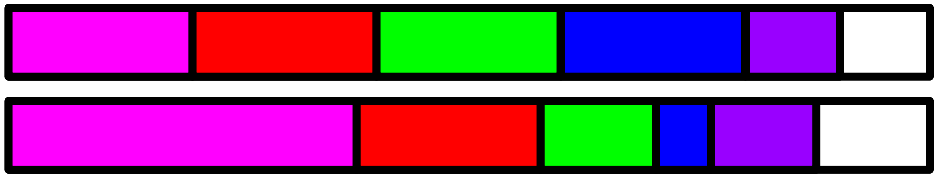
Slightly not nice (nice<0)
Slightly high priority

Tree would get updated (shown out of date)



CFS Run Queue
Red-Black Tree

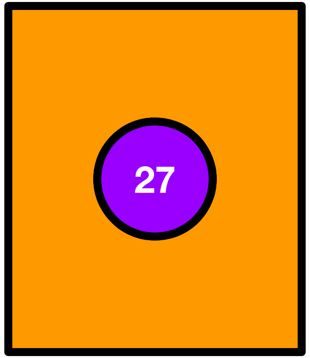
Processor Virtual Runtime Timeline



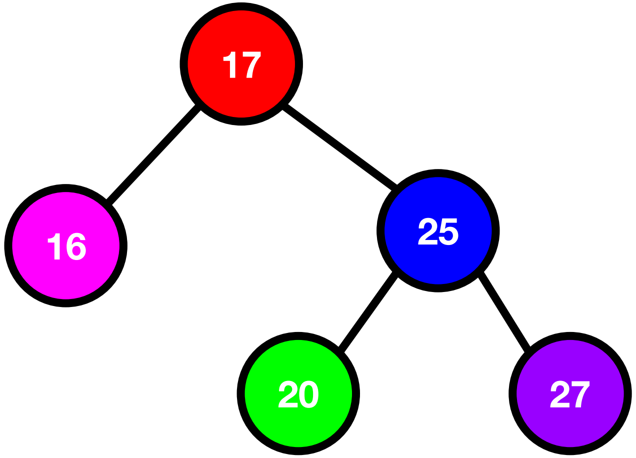
Processor Wall-Time Timeline

CFS Example

Processor



Slightly not nice (nice<0)
Slightly high priority



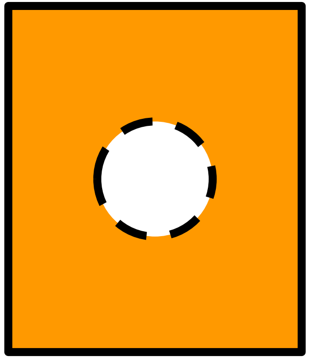
CFS Run Queue
Red-Black Tree

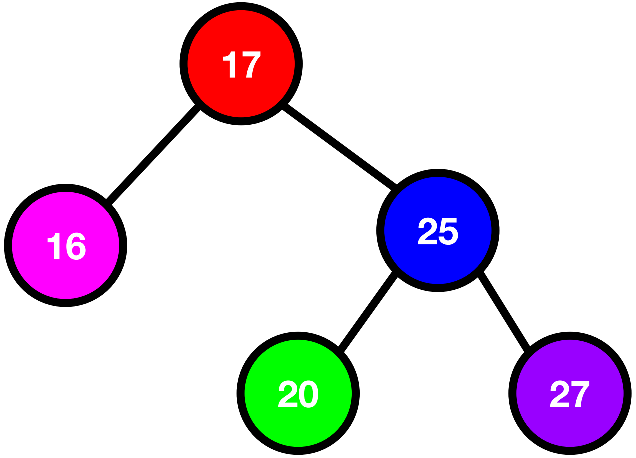
Processor Virtual Runtime Timeline

Processor Wall-Time Timeline

CFS Example

Processor





CFS Run Queue
Red-Black Tree

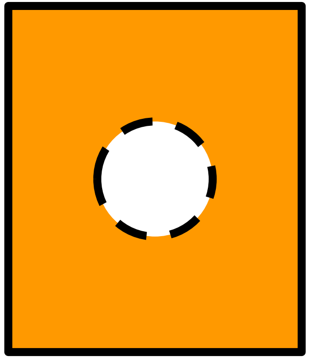
Processor Virtual Runtime Timeline

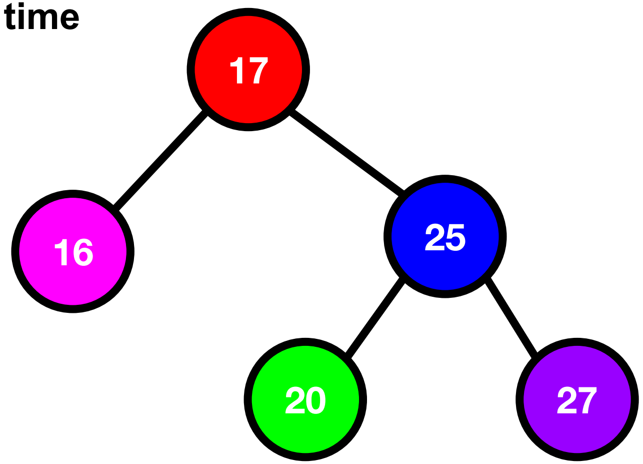
Processor Wall-Time Timeline

CFS Example

Normalization makes it so all processes' `vruntime` values ultimately have a maximum difference of minimum granularity in virtual time

Processor





CFS Run Queue
Red-Black Tree

Processor Virtual Runtime Timeline

Processor Wall-Time Timeline

CFS Example

Normalization makes it so all processes' `vruntime` values ultimately have a maximum difference of minimum granularity in virtual time

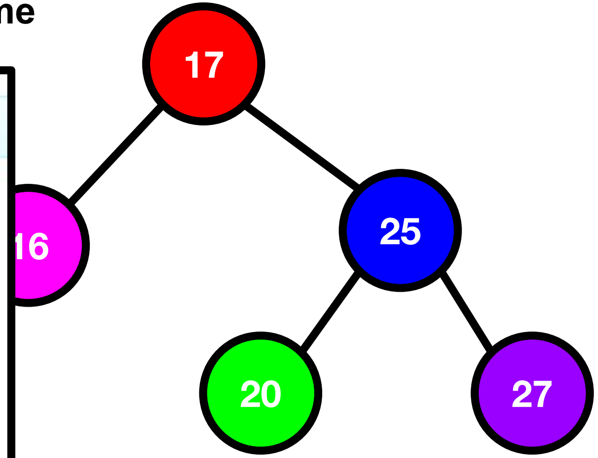
Processor

Mike Galbraith

> *Post by XingChao Wang*
> Hi Ingo,peter,
> When check_preempt_tick() selects next leftmost sched_entity,it calculates delta vruntime of curr and leftmost entity, then compares it with ideal_runtime. But ideal_runtime is real-time type, need convert it to virtual-time ,right?

Why? The scheduler converges vruntimes to within min_granularity, that's it's mission. What this test is trying to say is that if the

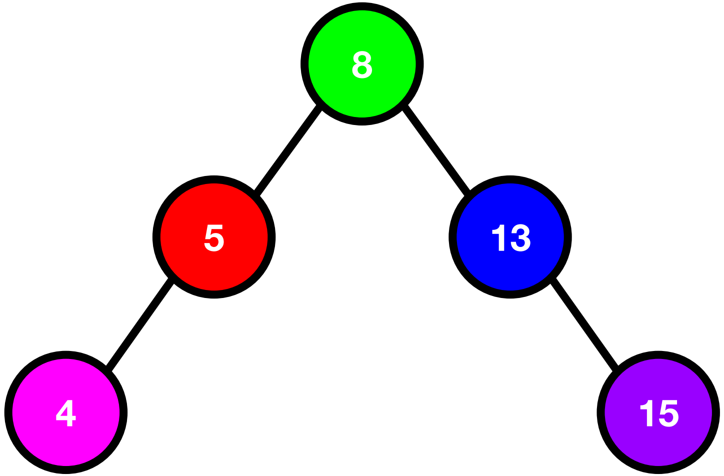
https://linux.kernel.narkive.com/FKiX036A/check-preempt-tick-check-vruntime-mistake



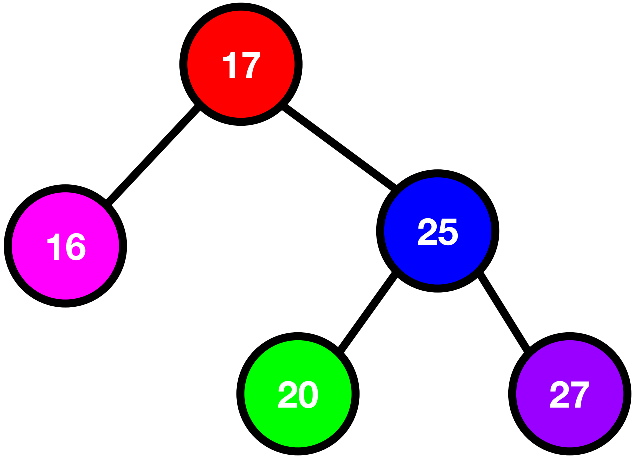
CFS Run Queue
Red-Black Tree

Processor Wall-Time Timeline

Before & After: CFS Run Queue (Red-Black Tree)



CFS Run Queue
Red-Black Tree

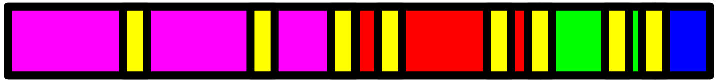


CFS Run Queue
Red-Black Tree

How Does The Scheduler Update?

Follow links for info on HR-timer
https://github.com/torvalds/linux/commit/8f4d37ec073c
https://lwn.net/Articles/549754/

- Periodic timer interrupt
  - 100-1000 Hz (every 1-10ms)
  - In practice, high resolution timers (HR-timer) implemented
- Interrupt calls `scheduler_tick()`
- Statistics about current process computed
- Check if preemption needed
  - Call `schedule()` for deciding next process



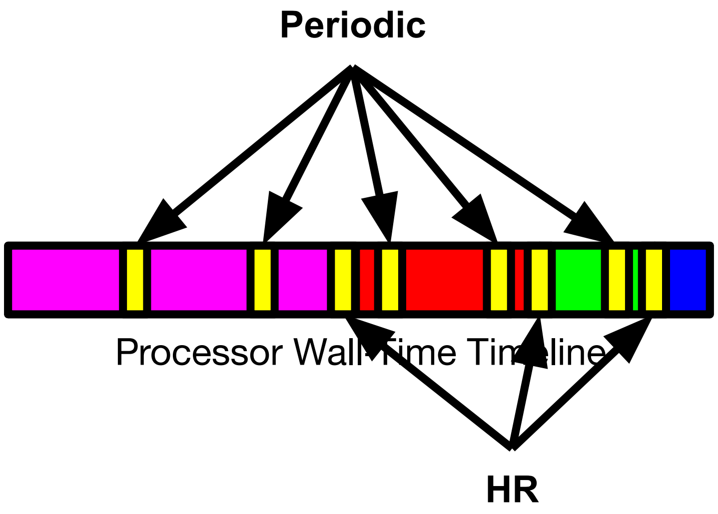
Processor Wall-Time Timeline

"Learning operating system development using Linux kernel and Raspberry Pi". URL: https://s-matyukevich.github.io/raspberry-pi-os/docs/lesson04/linux/scheduler.html

How Does The Scheduler Update?

Follow links for info on HR-timer
https://github.com/torvalds/linux/commit/8f4d37ec073c
https://lwn.net/Articles/549754/

- Periodic timer interrupt
  - 100-1000 Hz (every 1-10ms)
  - In practice, high resolution timers (HR-timer) implemented
- Interrupt calls `scheduler_tick()`
- Statistics about current process computed
- Check if preemption needed
  - Call `schedule()` for deciding next process



"Learning operating system development using Linux kernel and Raspberry Pi". URL: https://s-matyukevich.github.io/raspberry-pi-os/docs/lesson04/linux/scheduler.html

Math Behind CFS (3): Increasing Target Latency

- Definitions:
  - **`sched_nr_latency`**: number of runnable processes that should be scheduled in a period
  - **`sysctl_sched_latency`**: scheduling period (ms)
  - **`sysctl_min_granularity`**: supposed wall-time a process would execute
- Consider when the number of runnable processes exceeds `sched_nr_latency`, what happens?
  - period = (# proc)x(min_gran)

$$sysctl\ min\ granularity = \frac{sysctl\ sched\ latency}{sched\ nr\ latency}$$

"Can we get that again in English?"

$$supposed\ time\ slice = \frac{scheduling\ period}{number\ of\ processes}$$

$$timeslice = (scheduling\ period) \times \frac{(process\ weight)}{(sum\ of\ all\ process\ weights)}$$

Jinkyu Koo, "Linux kernel scheduler". URL: https://helix979.github.io/jkoo/post/os-scheduler/

“Pop quiz hotshot”

Author: pgrizzaffi, “Pop Quiz, Hotshot. Is It Automated?”. From the movie *Speed* (1994). URL: https://responsibleautomation.wordpress.com/2018/11/01/pop-quiz-hotshot-is-it-automated/

“Pop quiz hotshot”
Exercise 5
Access Code:

You only need one of these equations - think it over :)

Author: pgrizzaffi, “Pop Quiz, H
https://responsibleautomation.v

Math time. We have a scheduling period of 20 ms. We have 4 processes: 3 with nice = 0, 1 with nice = 5. What is the wall-time of the nice 5 process? In other words, how long does it *actually* run? How long does each of the individual nice = 0 processes run for?

$$sysctl\ min\ granularity = \frac{sysctl\ sched\ latency}{sched\ nr\ latency}$$

$$timeslice = (\,scheduling\ period) \times \frac{(\,process\ weight)}{(\,sum\ of\ \ all\ process\ weights)}$$

nice	-20	-19	-18	-17	-16	-15	-14	-13	-12	-11
weight	88761	71755	56483	46273	36291	29154	23254	18705	14949	11916
nice	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
weight	9548	7620	6100	4904	3906	3121	2501	1991	1586	1277
nice	0	1	2	3	4	5	6	7	8	9
weight	1024	820	655	526	423	335	272	215	172	137
nice	10	11	12	13	14	15	16	17	18	19
weight	110	87	70	56	45	36	29	23	18	15

Nice-to-weight conversion

Jinkyu Koo, “Linux kernel scheduler”. URL: https://helix979.github.io/jkoo/post/os-scheduler/

CFS Implementation - `sched_entity`

- [/kernel/sched/fair.c]()
  - Only ~10k lines of code!
- [/include/linux/sched.h]()
  - `struct sched_entity`
    - `u64     vruntime;`
- `vruntime` - virtual runtime
  - Weighted runtime

/ include / linux / sched.h   All sym   Sear

```
struct sched_entity {
        /* For load-balancing: */
        struct load_weight              load;
        struct rb_node                  run_node;
        struct list_head                group_node;
        unsigned int                    on_rq;

        u64                             exec_start;
        u64                             sum_exec_runtime;
        u64                             vruntime;
        u64                             prev_sum_exec_runtime;

        u64                             nr_migrations;
```

Linux 5.14.2 Source via Bootlin Elixir Cross Referencer
https://elixir.bootlin.com/linux/v5.14.2/source/include/linux/sched.h#L465

Robert Love, *Linux Kernel Development*, 3rd Edition

CFS Implementation - `vruntime` & `update_curr()`

- `update_curr()` keeps `vruntime` updated
- It also calls `update_min_vruntime()` to keep `min_vruntime` up to date

/ kernel / sched / fair.c

```c
/*
 * delta /= w
 */
static inline u64 calc_delta_fair(u64 delta, struct sched_entity *se)
{
	if (unlikely(se->load.weight != NICE_0_LOAD))
		delta = __calc_delta(delta, NICE_0_LOAD, &se->load);

	return delta;
}
```

/ kernel / sched / fair.c

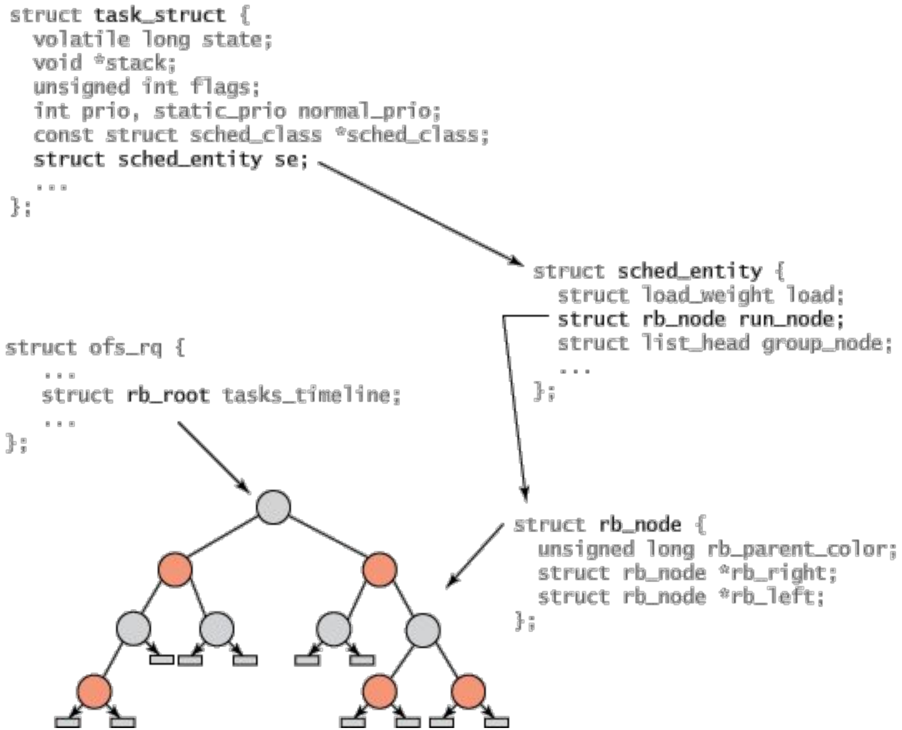
```c
static void update_curr(struct cfs_rq *cfs_rq)
{
	struct sched_entity *curr = cfs_rq->curr;
	u64 now = rq_clock_task(rq_of(cfs_rq));
	u64 delta_exec;

	if (unlikely(!curr))
		return;

	delta_exec = now - curr->exec_start;
	if (unlikely((s64)delta_exec <= 0))
		return;

	curr->exec_start = now;

	schedstat_set(curr->statistics.exec_max,
		      max(delta_exec, curr->statistics.exec_max));

	curr->sum_exec_runtime += delta_exec;
	schedstat_add(cfs_rq->exec_clock, delta_exec);

	curr->vruntime += calc_delta_fair(delta_exec, curr);
	update_min_vruntime(cfs_rq);

	if (entity_is_task(curr)) {
		struct task_struct *curtask = task_of(curr);

		trace_sched_stat_runtime(curtask, delta_exec, curr->vruntime);
		cgroup_account_cputime(curtask, delta_exec);
		account_group_exec_runtime(curtask, delta_exec);
	}

	account_cfs_rq_runtime(cfs_rq, delta_exec);
}
```

Robert Love, *Linux Kernel Development*, 3rd Edition

Linux 5.14.2 Source via Bootlin Elixir Cross Referencer
https://elixir.bootlin.com/linux/latest/source/kernel/sched/fair.c#L793

CFS Implementation - Process Selection

- What do we execute next?
- Process having shortest `vruntime`
- This is why we say “fair”
- Red-black tree (rbtree) data structure implemented for processes to sort them by `vruntime`
- rbtree is self-balancing binary search tree

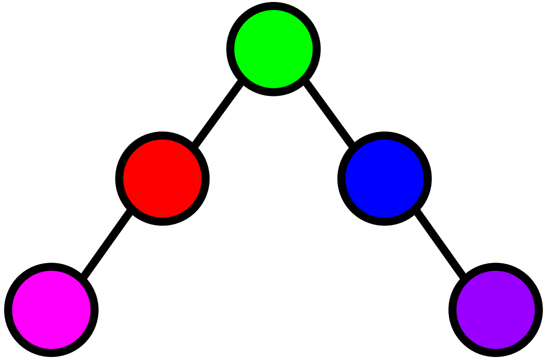


M. Jones, “Inside the Linux 2.6 Completely Fair Scheduler”. URL: https://developer.ibm.com/tutorials/l-completely-fair-scheduler/

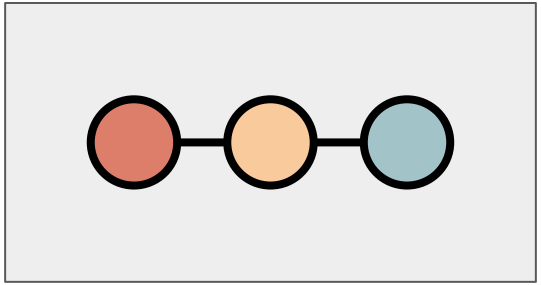
Robert Love, *Linux Kernel Development*, 3rd Edition

CFS Implementation - Sleeping & Waking Up

- Sleeping (blocked) processes cannot be run
  - Awaiting event
- Events include:
  - Time length
  - File I/O
  - Keyboard input
- Process voluntarily sleeps
  - Moves from run queue to wait queue
  - Invokes `schedule()`
- Reverse this to wake up
  - What will happen when process added to run queue? `vruntime`?



Run Queue ☕
Red-black tree



Wait Queue 💤
Linked list

Shh! They're **sleeping**!

Robert Love, *Linux Kernel Development*, 3rd Edition

Context Switching

(Switching Processes)

- `context_switch()`
- Switches mm (memory map)
- Switches register states

/ kernel / sched / core.c   All sym   Search Identifier

```c
/*
 * context_switch - switch to the new MM and the new thread's register state.
 */
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);

	/*
	 * For paravirt, this is coupled with an exit in switch_to to
	 * combine the page table reload and the switch backend into
	 * one hypercall.
	 */
	arch_start_context_switch(prev);

	/*
	 * kernel -> kernel   lazy + transfer active
	 *   user -> kernel   lazy + mmgrab() active
	 *
	 * kernel ->   user   switch + mmdrop() active
	 *   user ->   user   switch
	 */
	if (!next->mm) {                                // to kernel
		enter_lazy_tlb(prev->active_mm, next);
```

Linux Source via Bootlin Elixir Cross Referencer
https://elixir.bootlin.com/linux/latest/source/kernel/sched/core.c#L4644

Robert Love, *Linux Kernel Development*, 3rd Edition

We Just Talked About CFS (Normal Processes)

Literally halts everything
Unstoppable

Process migration

Hard deadline processes
Soonest first

Video encoding

Realtime
Latency-important processes

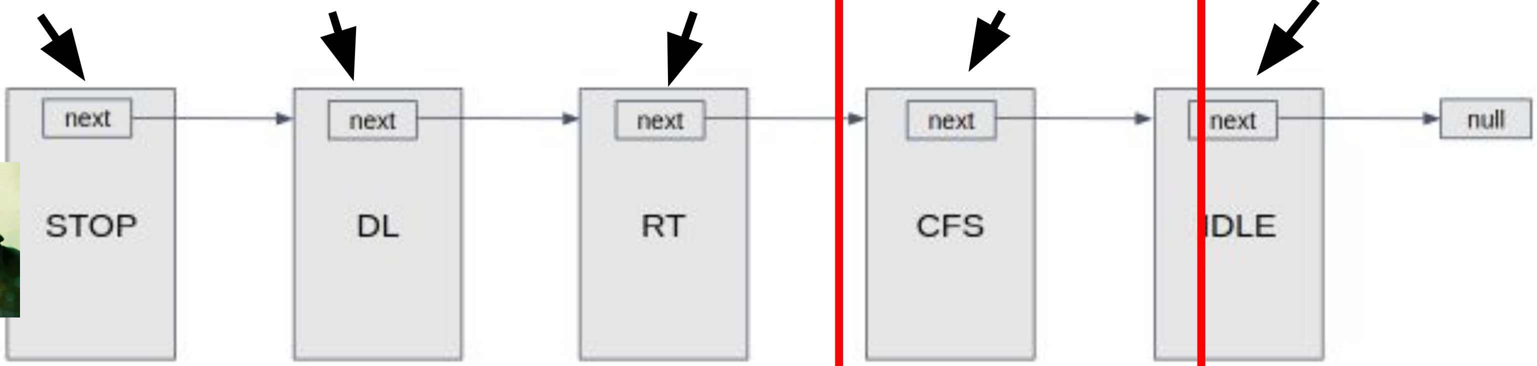
Interrupt request (IRQ)

Our normal process scheduler

Most processes (e.g. shell)

Nothing else runnable
Burn CPU cycles

Power saving



Highest Priority → Lowest Priority

Figure & info from: Viresh Kumar, “Fixing SCHED_IDLE”. URL: https://lwn.net/Articles/805317/
Denzel Washing photo modified from: Peter Swanson, “Unstoppable”. URL: http://armchairaudience.blogspot.com/2011/02/unstoppable.html

Let’s Talk About Realtime

Literally halts everything *Unstoppable*

Process migration

Hard deadline processes Soonest first

Video encoding

Realtime Latency-important processes

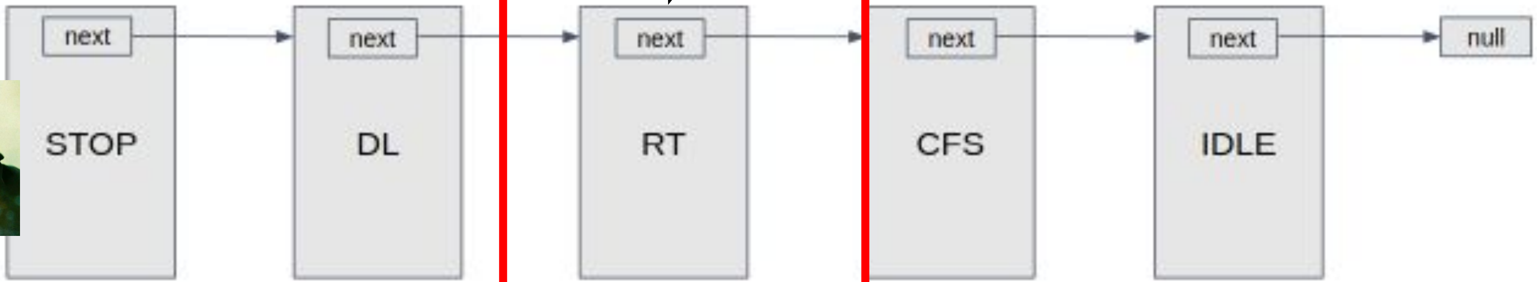
Interrupt request (IRQ)

Our normal process scheduler

Most processes (e.g. shell)

Nothing else runnable Burn CPU cycles

Power saving



Highest Priority → Lowest Priority

Figure & info from: Viresh Kumar, “Fixing SCHED_IDLE”. URL: https://lwn.net/Articles/805317/
Denzel Washing photo modified from: Peter Swanson, “Unstoppable”. URL: http://armchairaudience.blogspot.com/2011/02/unstoppable.html

Real Time - A Tale of Two Policies

- `SCHED_FIFO`
    - FIFO: First in, first out
    - Process executes until yielding
        - No timeslices
        - Very different from CFS
    - Could block or be done executing
- `SCHED_RR`
    - RR: Round-robin
    - Round-robin with timeslices and priorities

Viresh Kumar, “Fixing SCHED_IDLE”. URL: https://lwn.net/Articles/805317/
Robert Love, *Linux Kernel Development*, 3rd Edition

Real Time - A Tale of Two Policies

- `SCHED_FIFO`
  - FIFO: First in, first out
  - Process executes until yielding
    - No timeslices
    - Very different from CFS
  - Could block or be done executing
- `SCHED_RR`
  - RR: Round-robin
  - Round-robin with timeslices and priorities

So we have a beautiful real-time system if we need it?

Viresh Kumar, “Fixing SCHED_IDLE”. URL: https://lwn.net/Articles/805317/
Robert Love, *Linux Kernel Development*, 3rd Edition

Real Time - A Tale of Two Policies

- `SCHED_FIFO`
  - FIFO: First in, first out
  - Process executes until yielding
    - No timeslices
    - Very different from CFS
  - Could block or be done executing
- `SCHED_RR`
  - RR: Round-robin
  - Round-robin with timeslices and priorities

So we have a beautiful real-time system if we need it?

Not at all

Viresh Kumar, "Fixing SCHED_IDLE". URL: https://lwn.net/Articles/805317/
Robert Love, *Linux Kernel Development*, 3rd Edition

Real Time - A Tale of Two Policies

- `SCHED_FIFO`
  - FIFO: First in, first out
  - Process executes until yielding
    - No timeslices
    - Very different from CFS
  - Could block or be done executing
- `SCHED_RR`
  - RR: Round-robin
  - Round-robin with timeslices and priorities

The Reality

https://twitter.com/Marjan_Lion/status/727692489264484352/photo/1

Viresh Kumar, “Fixing SCHED_IDLE”. URL: https://lwn.net/Articles/805317/
Robert Love, *Linux Kernel Development*, 3rd Edition

Real Time - A Tale of Two Policies

- `SCHED_FIFO`
  - FIFO: First in, first out
  - Process executes until yielding
    - No timeslices
    - Very different from CFS
  - Could block or be done executing
- `SCHED_RR`
  - RR: Round-robin
  - Round-robin with timeslices and priorities

The Reality

https://twitter.com/Marjan_Lion/status/727692489264484352/photo/1

Soft Real-Time
No guarantees on timing

Viresh Kumar, "Fixing SCHED_IDLE". URL: https://lwn.net/Articles/805317/
Robert Love, *Linux Kernel Development*, 3rd Edition

System Calls

Table 4.2. Scheduler-Related System Calls

System Call	Description
`nice()`	Sets a process's nice value
`sched_setscheduler()`	Sets a process's scheduling policy
`sched_getscheduler()`	Gets a process's scheduling policy
`sched_setparam()`	Sets a process's real-time priority
`sched_getparam()`	Gets a process's real-time priority
`sched_get_priority_max()`	Gets the maximum real-time priority
`sched_get_priority_min()`	Gets the minimum real-time priority
`sched_rr_get_interval()`	Gets a process's timeslice value
`sched_setaffinity()`	Sets a process's processor affinity
`sched_getaffinity()`	Gets a process's processor affinity
`sched_yield()`	Temporarily yields the processor

Robert Love, *Linux Kernel Development*, 3rd Edition.
Page 64 in Kindle edition.

Next Lecture:

System
Calls