Assignment 4 Tutorial

Linux Scheduler

Papadogiannakis Manos
papamano@csd.uoc.gr

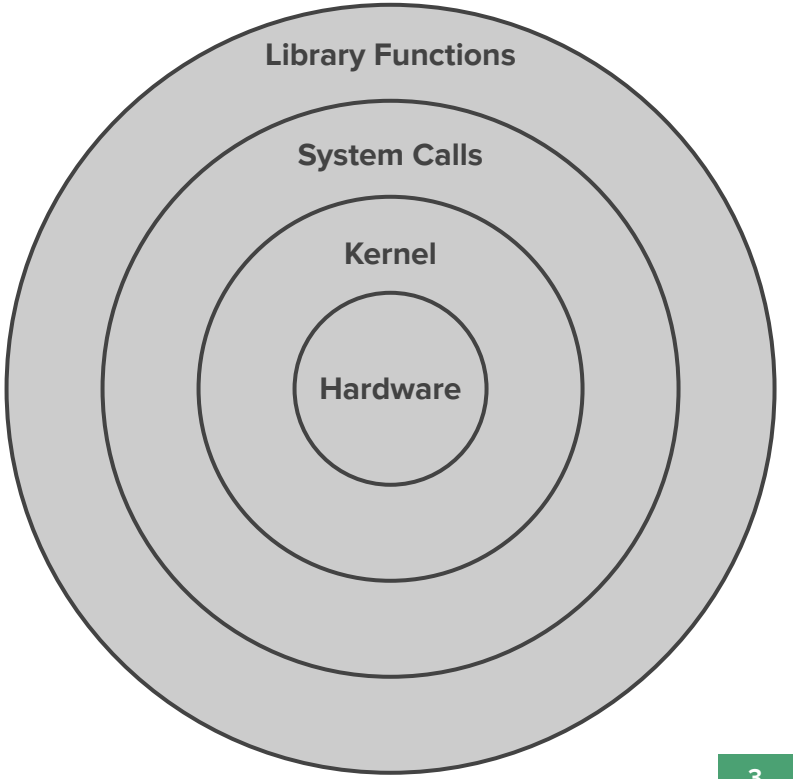
CS-345: Operating Systems
Computer Science Department
University of Crete

Outline

- Linux Scheduler
- Scheduler internals
- History
- Assignment 4

Linux Kernel

- **Heart of the Operating System**
- **Interface between resources and user processes**
- **What the Kernel does**
  - Memory Management
  - **Process Management**
  - Device Drivers
  - System Calls

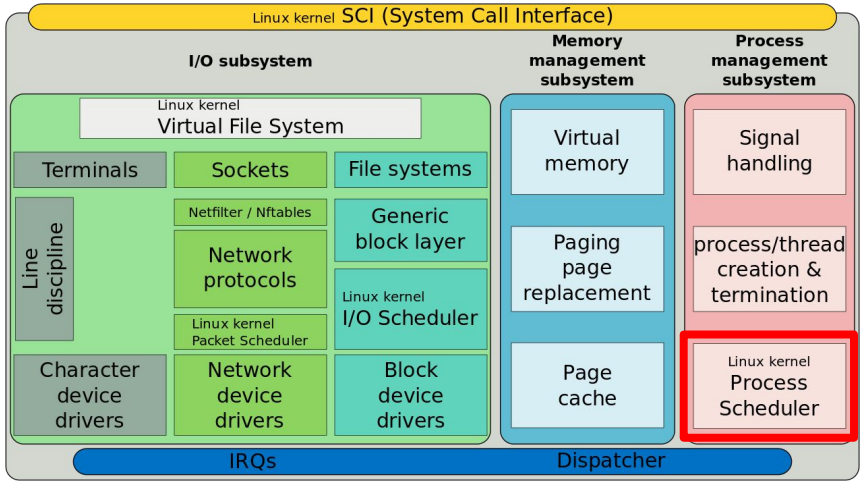


Process Management

- **Multitasking operating systems**
  - Tasks must run in parallel
- **Usually tasks are more than the CPU cores**
- **Need to make it possible to execute tasks at the "same" time**

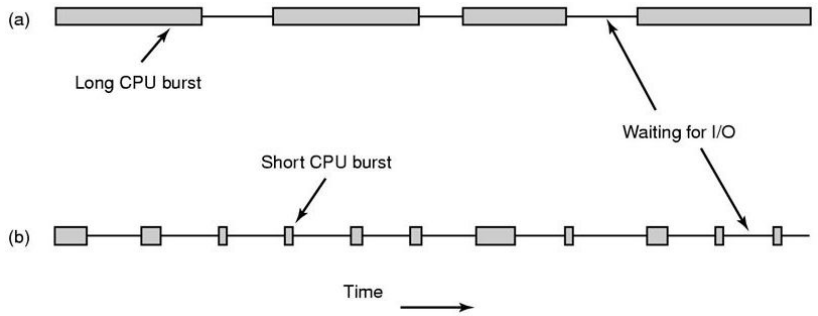
Scheduler

- Coordinates how tasks share the available processor(s)
- Prevents task starvation and preserves fairness
- Take into account system tasks



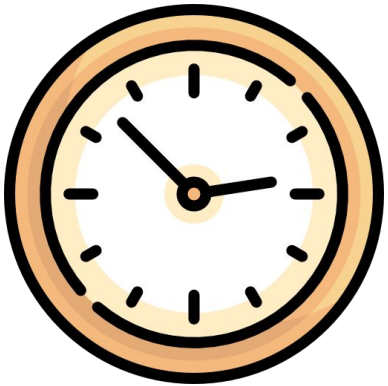
Task Types

- **Balance between two types of processes:**
  a. Batch processes
  b. I/O Bound tasks
- **Not mutually exclusive**
- **Preemption: temporarily evict a running task**
- **Quantum: Variable but keep it as long as possible**



Real-time processes

- **Need guarantee about their execution in time boundaries**
- **Soft real-time processes**
  - A task might run a bit late
- **Hard real-time processes**
  - Strict time limits
  - Not supported by default Linux



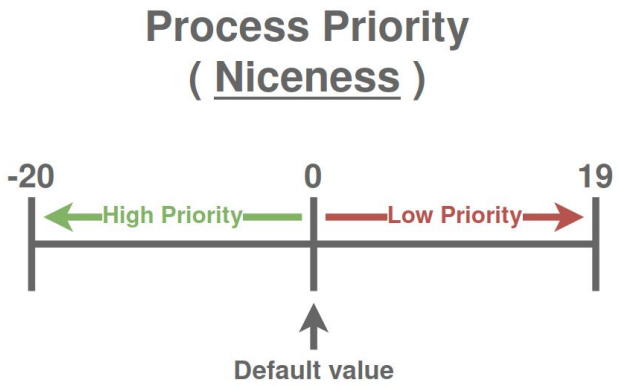
Scheduler Internals

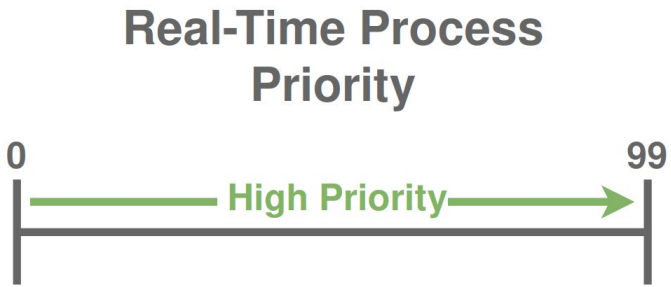
Terms

- **Scheduling Policy: defines what type of scheduling behavior should apply to a process**
  - Expresses rules and priorities
  - SCHED_NORMAL, SCHED_BATCH, SCHED_IDLE, SCHED_RR
- **Scheduling Algorithm: defines how the scheduler actually selects the next task to run, within the rules set by the policy.**
  - Completely Fair Scheduler, First-In-First-Out, Round-Robin, Earliest Deadline First
- **Scheduling Class: kernel structure that defines how tasks belonging to certain scheduling policies are managed**

Priority

- Linux provides **Priority-based** scheduling
- A “number” determines how important a task is

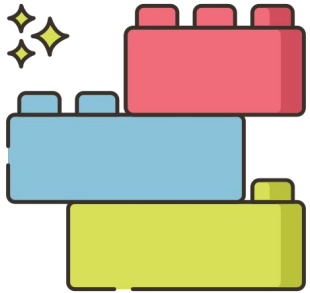




Process Descriptor

- **Scheduler needs information for each process**

- **Useful fields in task_struct:**
  - prio: Process priority
  - sched_class: Scheduling class
  - policy: Scheduling policy



Scheduler Design

- **Extensible** hierarchy of scheduler modules
- Each module encapsulates a **scheduling policy**
- Real-time classes:
  - SCHED_FIFO
  - SCHED_RR

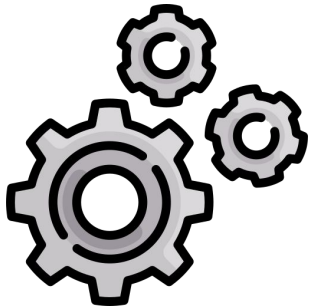
```
static const struct sched_class fair_sched_class = {
  .next                 = &idle_sched_class,
  .enqueue_task         = enqueue_task_fair,
  .dequeue_task         = dequeue_task_fair,
  .yield_task           = yield_task_fair,
  .check_preempt_curr   = check_preempt_wakeup,
  .pick_next_task       = pick_next_task_fair,
  .put_prev_task        = put_prev_task_fair,

...
```

https://elixir.bootlin.com/linux/v2.6.38.1/source/kernel/sched_fair.c

`schedule(void)`

- **Main scheduler function is `schedule( )`**
  - Replace currently executing process with another

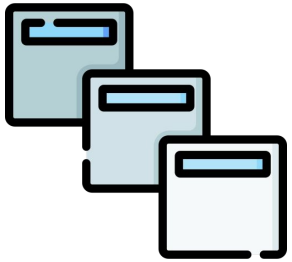
- **Called from different places**
  - Periodic scheduler
  - Current task enters sleep state
  - Sleeping task wakes up



Run queue

- Data structure that manages active processes

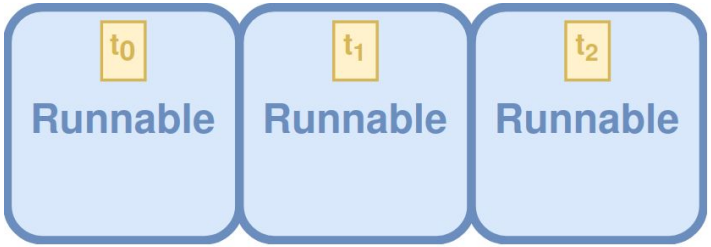
- Holds tasks in the “runnable” state



History

History

- **Genesis**
  - Circular queue
  - Round-robin policy



- **Linux v2.4 - O(n) scheduler**
  - Each task runs a quantum of time in each epoch
  - Epoch advances after all runnable tasks have their quantum
  - At the beginning of each epoch, all tasks get a new quantum

History

- **Linux v2.6 - O(1) Scheduler**
  - Division between real-time and normal tasks
  - Early preemption based on priority
  - One list per priority (as many as 140 lists)

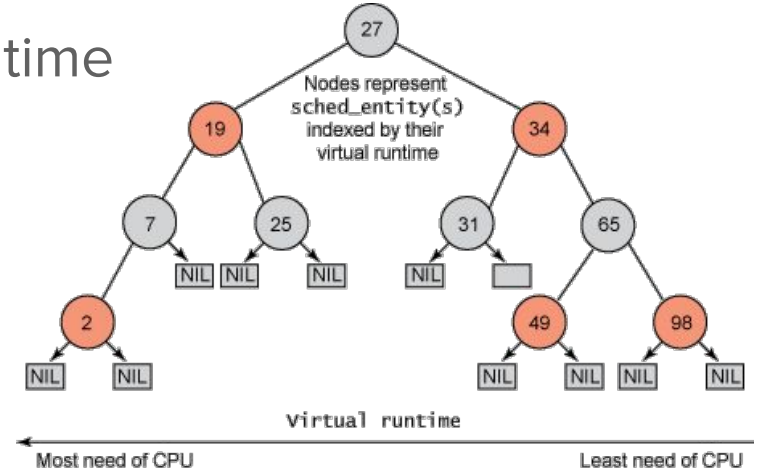
- **Linux v2.6.23 - CFS**
  - Introduced in 2007, Improved in 2016
  - Default scheduler

Completely Fair Scheduler

- **Models an "*ideal, precise multitasking CPU*"**
- **Ideal scheduling: *n* tasks share 100/*n* percentage of CPU effort each**
- **Calculates how long a task should run based on the number of runnable processes**
- **Fairness:**
  - Tasks get their share of the CPU relative to others
  - A task should run for a period proportional to its priority

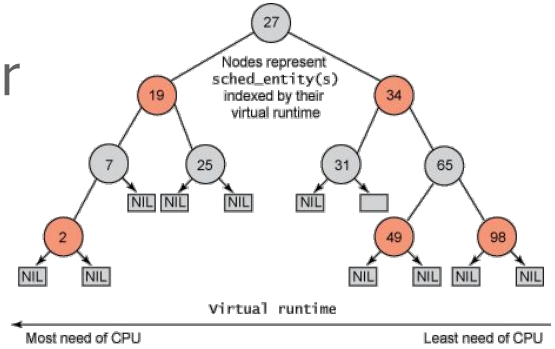
Completely Fair Scheduler

- **Time-ordered red-black tree**
  - Runnable tasks are sorted by vruntime
- **When a task is executing its vruntime increases**
  - Moves to the right of the tree
- **Scheduler always selects leftmost leaf**
  - Task with smallest vruntime



Completely Fair Scheduler - Improvements

- **Virtual clock ticks slowly for important tasks**
  - Move slower to the right of the tree
  - Chance to be scheduled again sooner
- **Leftmost node is cached**
  - O(1) access



- **Reinsertion of preempted tasks takes `O(logn)`**

Assignment 4

Assignment 4 - Least Tolerance First

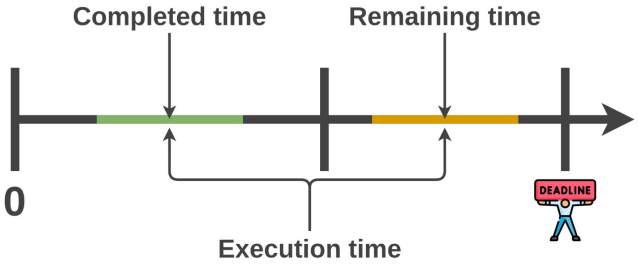
- **Assume computation-heavy processes:**

```
while(true) Simulate_Universe(...);
```

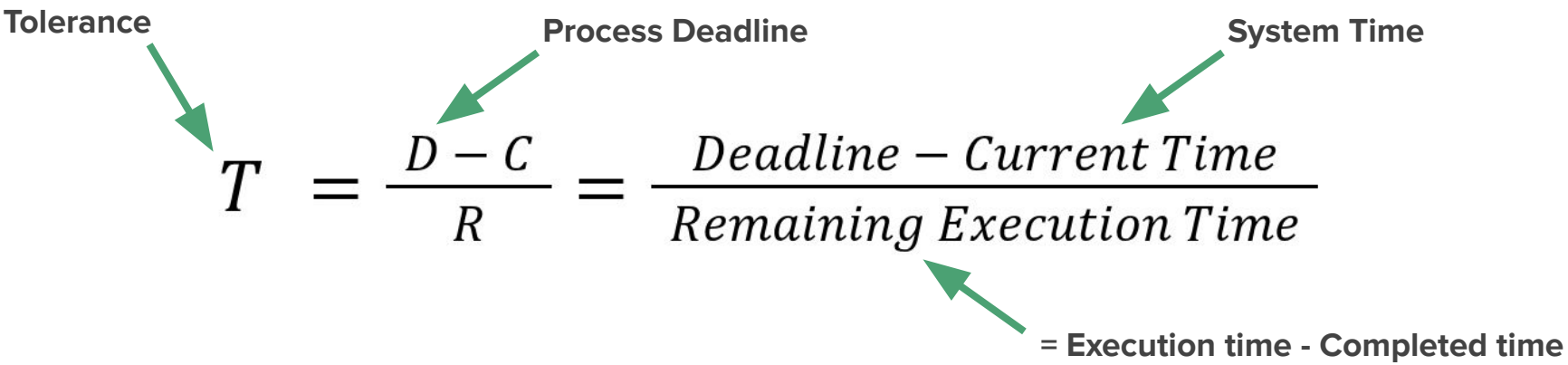
- **Each process is defined by:**
  - Deadlines (**D**)
  - (Estimated) Execution Time (**E**)

- **Our algorithm prioritizes tasks that are most in danger of missing their deadlines**
  - Relative to their remaining execution time

How do we measure that?

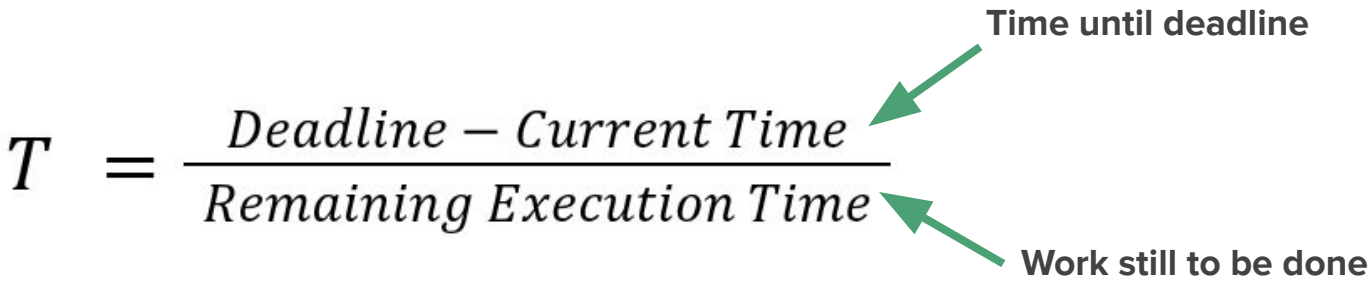


Tolerance Definition



LTF schedules the process with the smallest Tolerance, prioritizing tasks that have the least wiggle room before their deadlines

Intuition

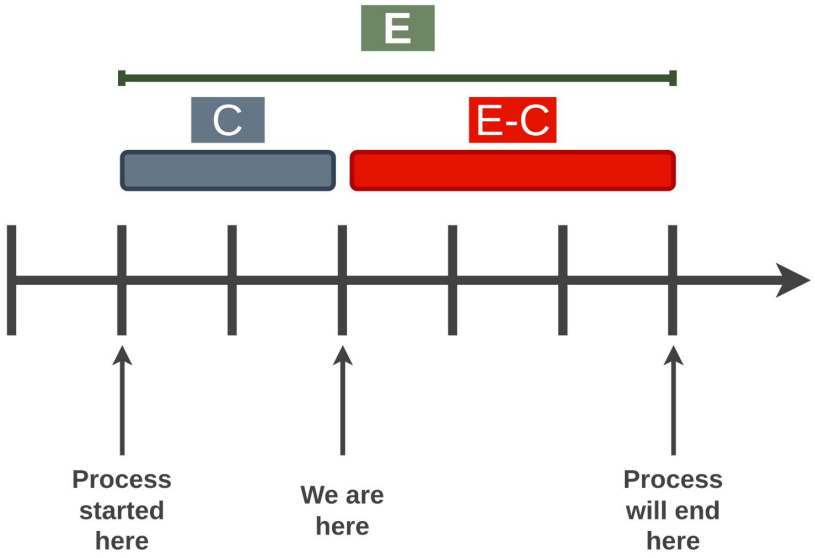


- **Large Tolerance**
  - Plenty of time to work
  - Process can wait
- **Small Tolerance**
  - Little time left until deadline
  - Process should be scheduled first

➢ **Smaller Tolerance = Less room to maneuver = Higher Priority**

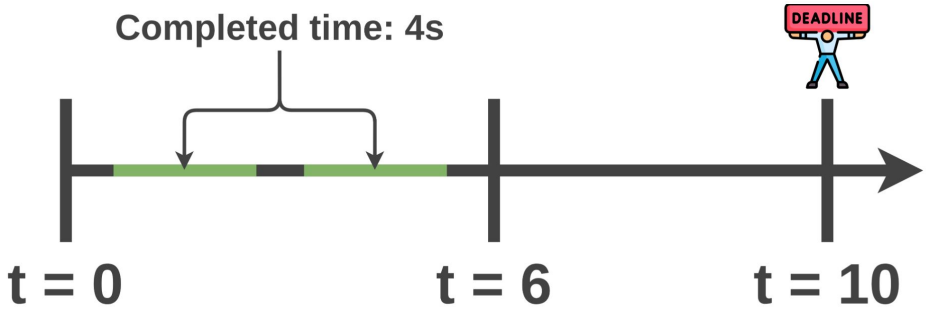
Remaining Time

- Process has already run for C ms
- Its execution time is E ms
- Its remaining computation time is (E - C) ms
- Might not be consecutive



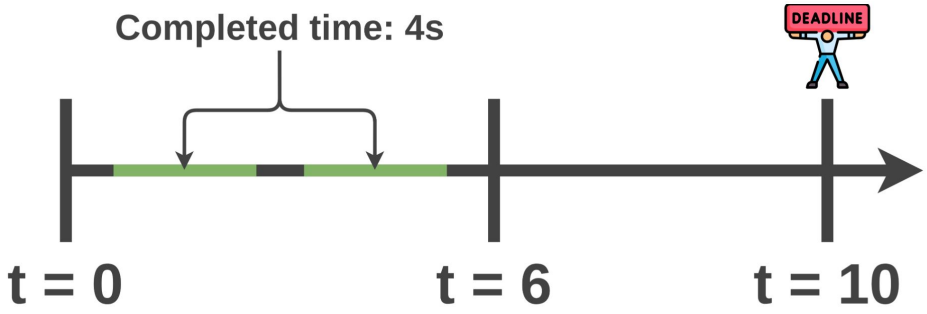
Example 1

- **Parameters**
  - Deadline: 10s
  - Execution Time: 6s
- **Has run for 4s**
- **Current time: 6s**
- $$\text{Tolerance} = \frac{D - \text{Current Time}}{\text{Remaining Execution Time}}$$

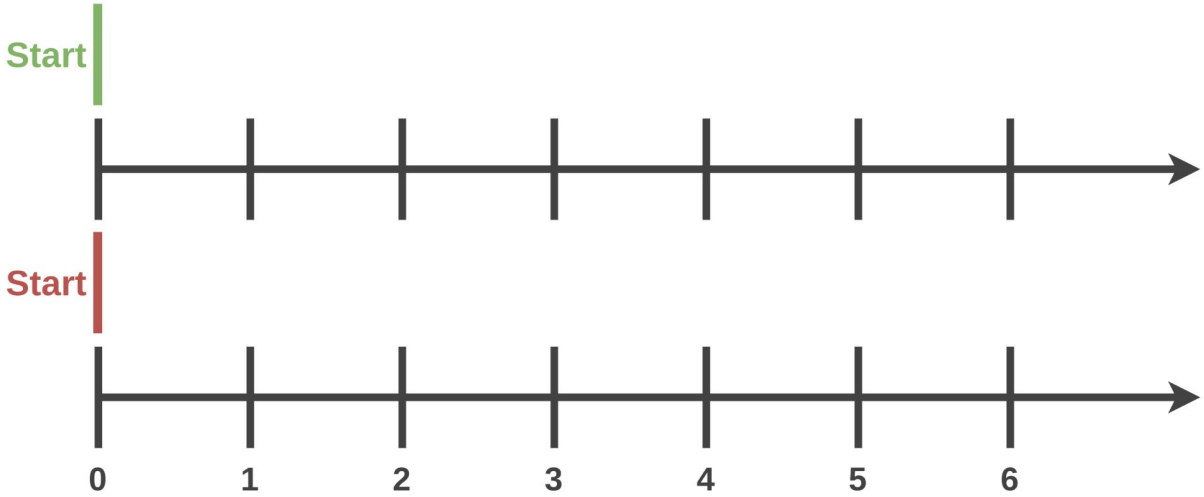


Example 1

- **Parameters**
  - Deadline: 10s
  - Execution Time: 6s
- **Has run for 4s**
- **Current time: 6s**
- $$\text{Tolerance} = \frac{D - \text{Current Time}}{\text{Remaining Execution Time}} = \frac{10 - 6}{6 - 4} = \frac{4}{2} = 2$$



Preemption



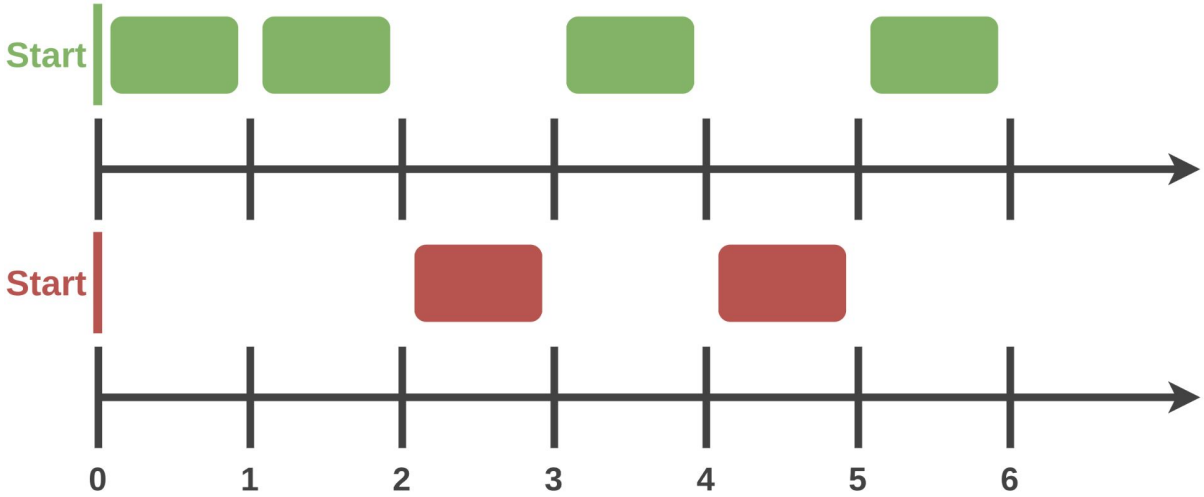
Time	P_1	P2
0		
1		
2		
3		
4		
5		

P1:
Execution Time = 5s, Deadline = 10s

P2:
Execution Time = 4s, Deadline = 11s

$$T = \frac{Deadline - Current\ Time}{Remaining\ Execution\ Time}$$

Preemption



Time	P_1	P2
0	$\frac{10-0}{5} = 2$	$\frac{11-0}{4} = 2.75$
1	$\frac{10-1}{4} = 2.25$	$\frac{11-1}{4} = 2.5$
2	$\frac{10-2}{3} = 2.7$	$\frac{11-2}{4} = 2.25$
3	$\frac{10-3}{3} = 2.3$	$\frac{11-3}{3} = 2.6$
4	$\frac{10-4}{2} = 3$	$\frac{11-4}{3} = 2.3$
5	$\frac{10-5}{2} = 2.5$	$\frac{11-5}{2} = 3$

P1:
Execution Time = 5s, Deadline = 10s

P2:
Execution Time = 4s, Deadline = 11s

$$T = \frac{Deadline - Current\ Time}{Remaining\ Execution\ Time}$$

Termination

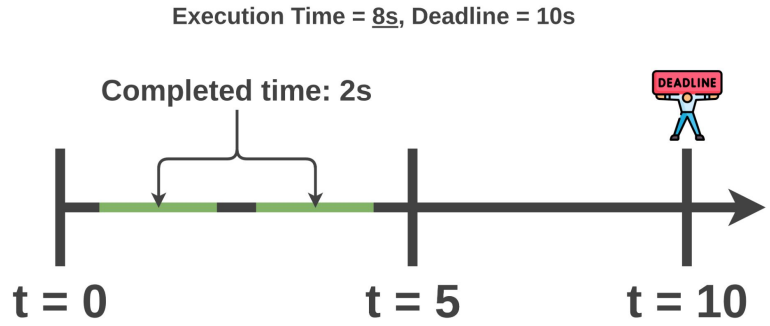
- **When is a process done?**
- **Once it has completed its execution time**
  - Your responsibility to kill it
- **If it misses its deadline (Tolerance < 0)**
  - Your responsibility to kill it
  - Simulate hard deadlines

Special Case

$$T = \frac{Deadline - Current\ Time}{Remaining\ Execution\ Time}$$

- **There is a special case when Tolerance < 1**
  - Time until deadline < Work still to be done
- **The process is doomed to miss its deadline**
  - Infeasible to complete
- **How to handle?**
  1. Kill the process immediately, treating it as infeasible
  2. Skip it temporarily until it misses its deadline
  3. Schedule it anyway (might delay other process)

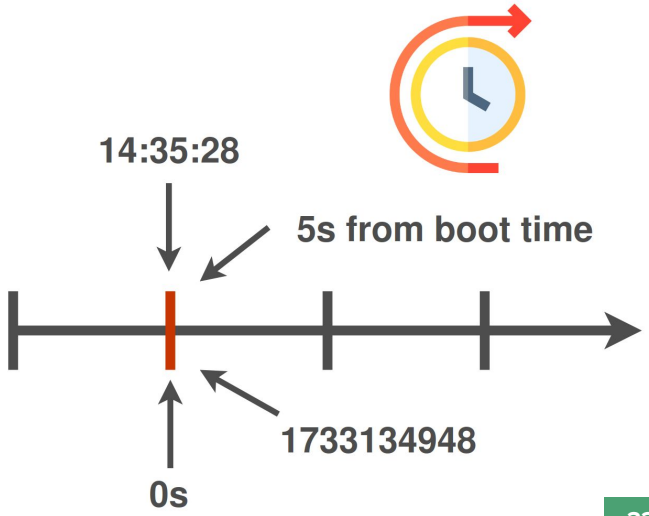
➢ You need to **understand** your decision and **explain** it!!!



Time

- **How do we measure time?**
  - Do we use absolute values (like in examples)?
  - Do we use wall clock time?
  - Do we use a reference point?

- **Free to choose whatever suits your implementation**



Implementation

- **Use your code from assignment 3**
  - System calls set deadlines
- **Linux kernel compilation process**
  - Instructions in assignment 3
- **Might need to make changes to `task_struct`**

Testing

- **Create simple demo processes**
  - Each initially sets its parameters

- **Each process should spin forever**
  - Infinite loop, not sleep
  - Scheduler will kill process once computation time has been fulfilled

- **Scheduler should print:**
  - PID of the task it selected
  - Its parameters

```
[LTF Scheduler][Timestamp: 0] Selected Process 1 [E: 5s, D: 10s] with tolerance 2
[LTF Scheduler][Timestamp: 1] Selected Process 1 [E: 5s, D: 10s] with tolerance 2.25
[LTF Scheduler][Timestamp: 2] Selected Process 2 [E: 4s, D: 11s] with tolerance 2.25
[LTF Scheduler][Timestamp: 3] Selected Process 1 [E: 5s, D: 10s] with tolerance 2.3
[LTF Scheduler][Timestamp: 4] Selected Process 2 [E: 4s, D: 11s] with tolerance 2.3
[LTF Scheduler][Timestamp: 5] Selected Process 1 [E: 5s, D: 10s] with tolerance 2.5
```

- **Don’t forget existing processes**
  - Don’t want to schedule only ours

You can grep this

Notes

Files

- **Actual context switch**
  - kernel/sched.c
- **Completely Fair Scheduler**
  - kernel/sched_fair.c
- **Scheduling structs**
  - include/linux/sched.h
- **Process descriptor**
  - include/linux/sched.h
- **Real-time scheduling**
  - kernel/sched_rt.c

sched.c

```
asmlinkage void __sched schedule(void) {

struct task_struct *prev, *next;
 ...
struct rq * rq;
 ...
preempt_disable();
 ...
prev = rq->curr;
 ...
pur_prev_task(rq, prev);

next = pick_next_task(rq);

 ...
if (likely(prev != next)) {
 ...
    context_switch(rq, prev, next);
```

Previous and next tasks

The processors runqueue (1 in this assignment)

Disable preemption (avoid schedule inside schedule)

Previous is the current task runnin

Put prev task in the runqueue

The appropriate pick function is called depending on the scheduling class

Actual context switch

Notes

- **Use Bootlin to find functions, structs, etc...**
  - https://elixir.bootlin.com/linux/v2.6.38.1/source

- **You can also map source code using ctags**
  - http://www.tutorialspoint.com/unix_commands/ctags.htm

- **Understand how the scheduler works**
  - Use **printk** to observe kernel behavior
  - Follow the call to find out how the next tasked is picked

Notes

- **Reuse existing code snippets within the kernel**
  - E.g. traversing data structures
- **Compile often with small changes**
  - Massively helps debugging
- **Submit anything you can to show your effort!!!**
  - A **README** file goes a long way
  - Even if your implementation does not fully work

Turnin

What to submit:

1. bzImage
2. Modified or created source files
3. Test programs and headers in Guest OS
4. README

Credit

Icons created by Freepik - Flaticon

Thank You!

papamano@csd.uoc.gr

Questions?