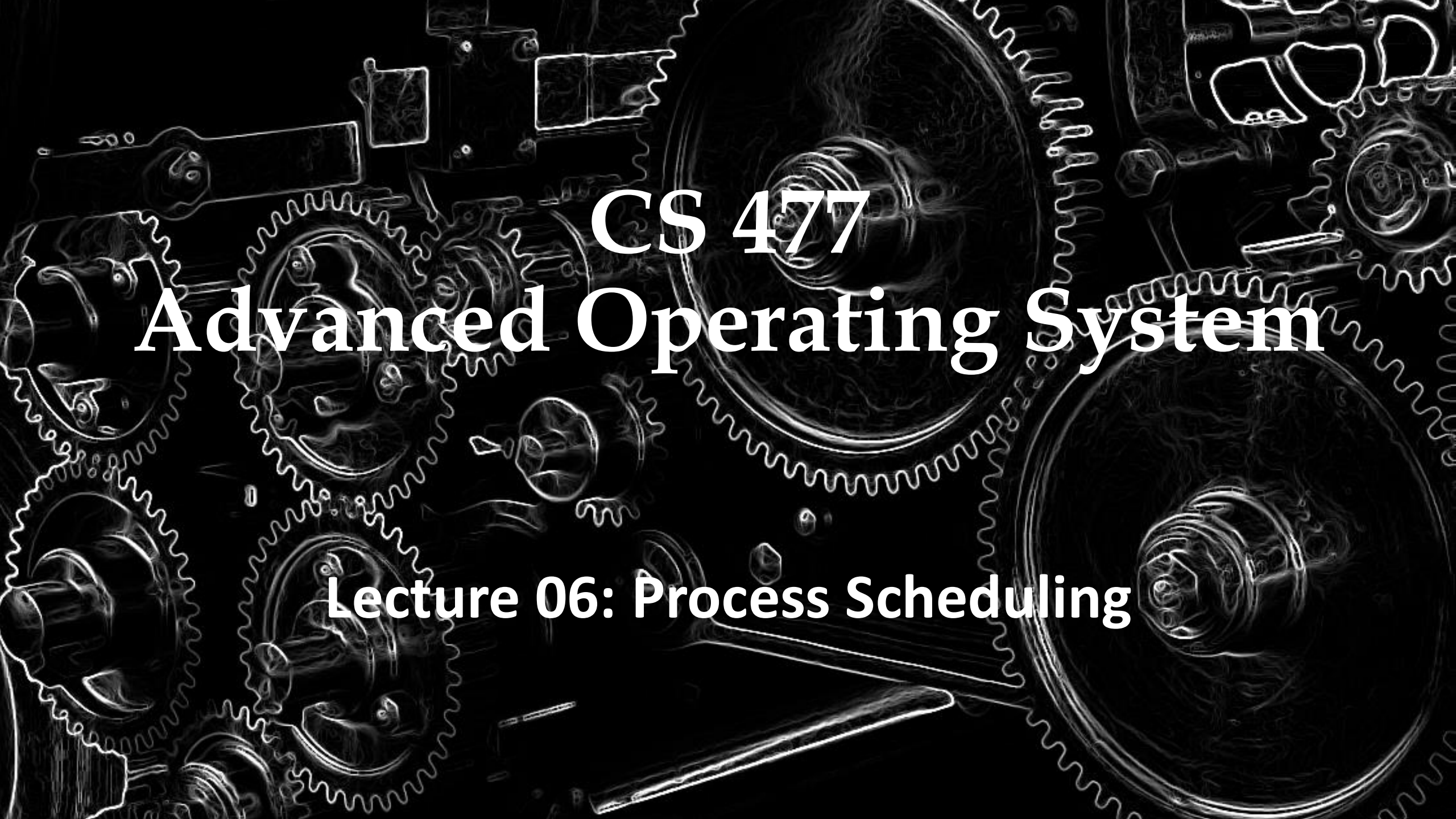


CS 477
Advanced Operating System

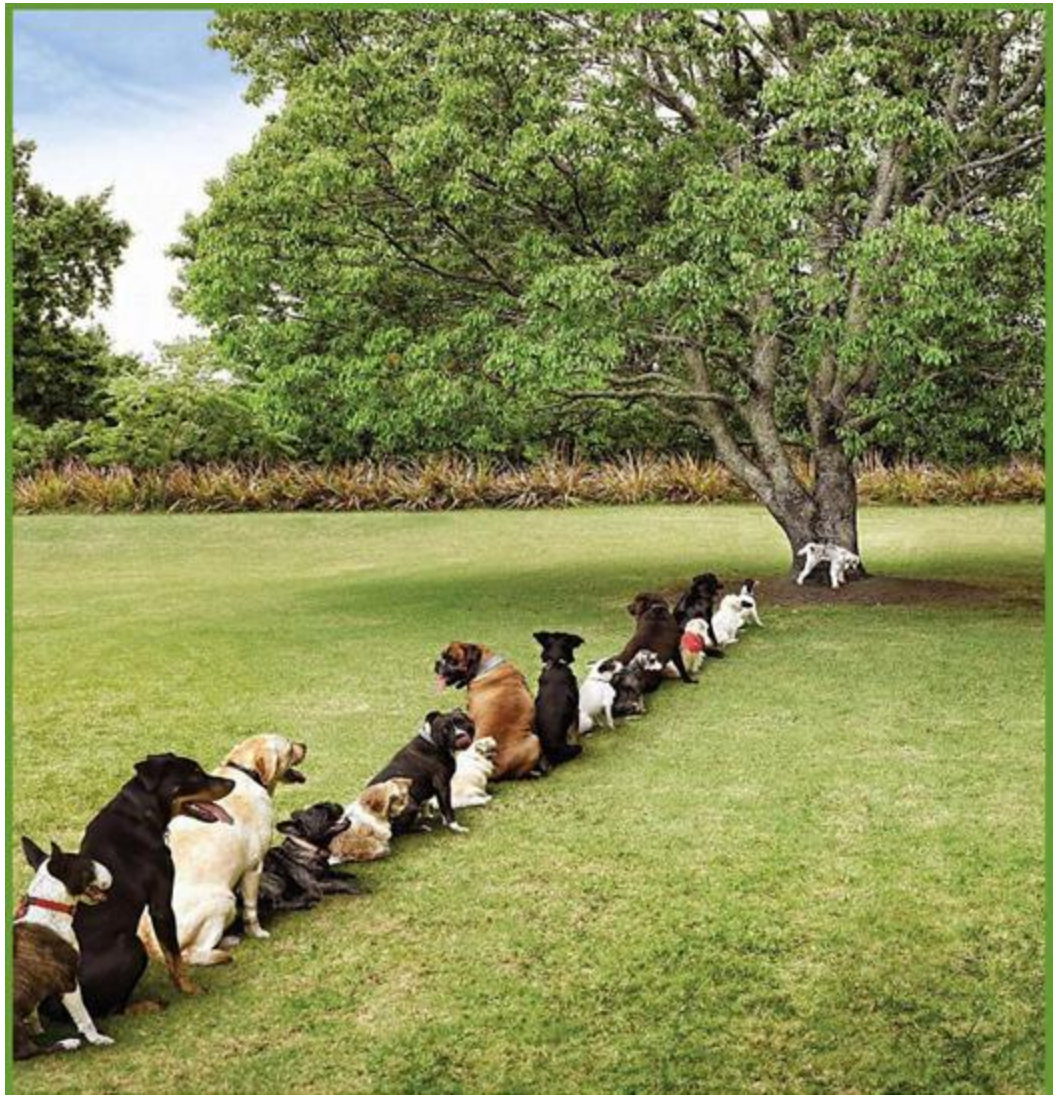
Lecture 06: Process Scheduling

Some logistics

- Lab 2 deadline is soon
- Lab 3 will be soon released
- Project proposal deadline is this Friday

Today's agenda

- Process scheduling
- Scheduling policy
- Linux EEVDF
- Scheduler class in Linux
- Preemption and context switch



Processor scheduler

- Decides which process runs next, and for how long
- Responsible for making the best use of processor (CPU)
  - Do not was CPU cycles for waiting processe
  - Give higher priority to higher-priority processes
  - Do not starve low-priority process

Multitasking

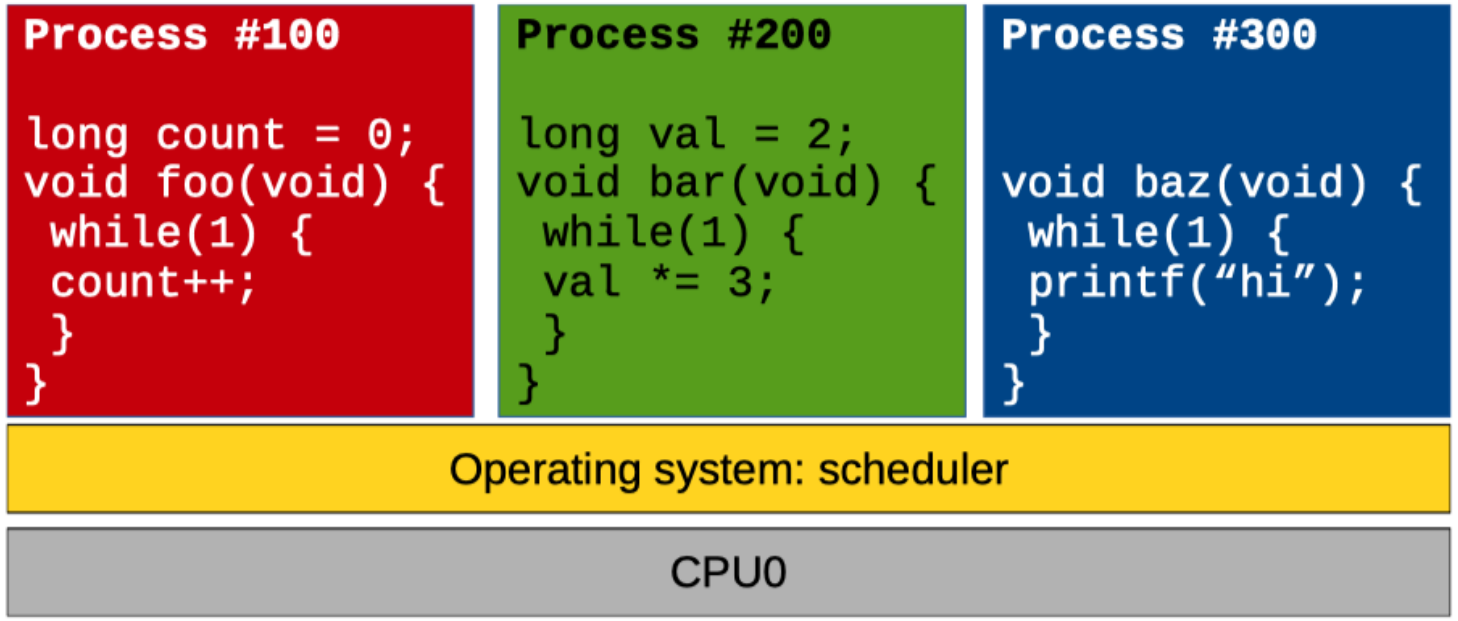
- Simultaneously interleave execution of more than one process
- Single core
  - Illusion → multiple processes running concurrently
- Multicore
  - Scheduler can exploit parallelism

Types of multitasking

- **Cooperative multitasking**: old OS (Windows 3.1) & language runtimes
  - A process does not stop running until it decides to yield CPU
  - OS cannot enforce **fair scheduling**

- **Preemptive multitasking:** almost all modern OS
  - OS can interrupt process execution (preemption)
  - Interrupt only after the time quota is over (timeslice)
    - Priority decides timeslice duration

Question about preemptive scheduling



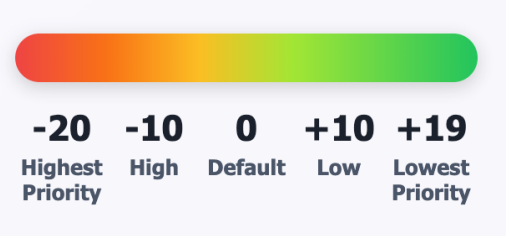
Q. How can the preemptive scheduler take back control if a process executes an infinite loop?

Types of tasks: IO vs. CPU

- Scheduling policy: A set of rules determining **which task runs when**
- **IO-bound process**
  - Spend most of the time waiting for IO: disk, network, keyboard, mouse etc.
  - Short duration runs
  - **Response time** is important
- **CPU-bound process**
  - Heavy use of the CPU: MATLAB, scientific computation, etc.
  - Caches stay hot when they run for a long duration

Process priority

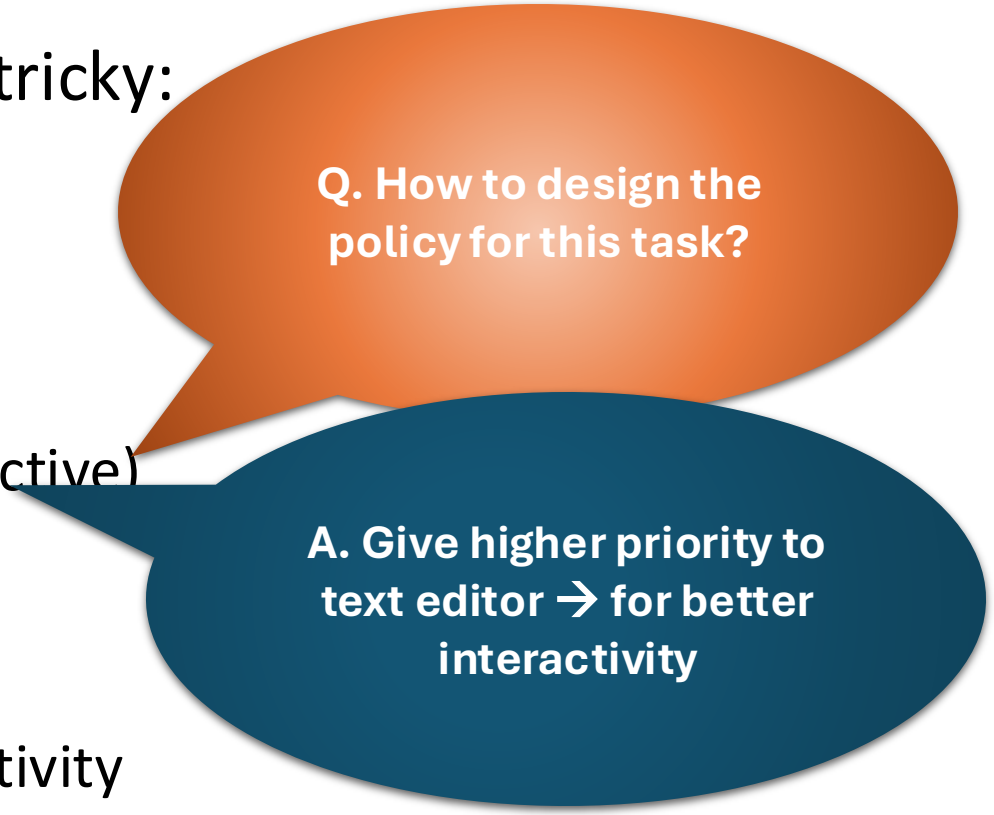
- **Priority-based scheduling**
  - Rank processes based on their worth and need for CPU time
  - Higher priority processes run before lower priority processes



- Linux has two priority ranges:
  - Nice value: -20 ←→ +19
- Real-time priority: 0 ←→ 99
  - Higher value means higher priority
  - Real-time processes have a higher priority than standard processes

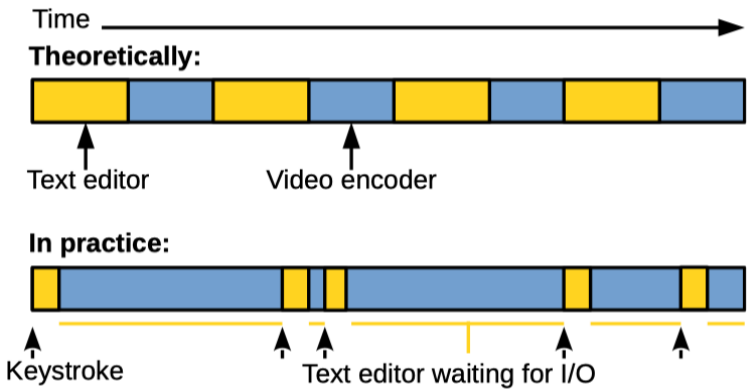
Scheduling policy: timeslice

- How much does should a process execute before being preempted
- Defining the default timeslice (absolute) is tricky:
  - Too long → bad interactive performance
  - Too short → high context switch overhead
- Two tasks:
  - Text editor: IO-bound, latency-sensitive (interactive)
  - video encoder: CPU-bound, background job
- Scheduling goal:
  - Text editor: preempt video encoder for interactivity
  - Video encoder: run as long as possible for better CPU cache utilization



Policy: "proportional" timeslice in the Linux scheduler

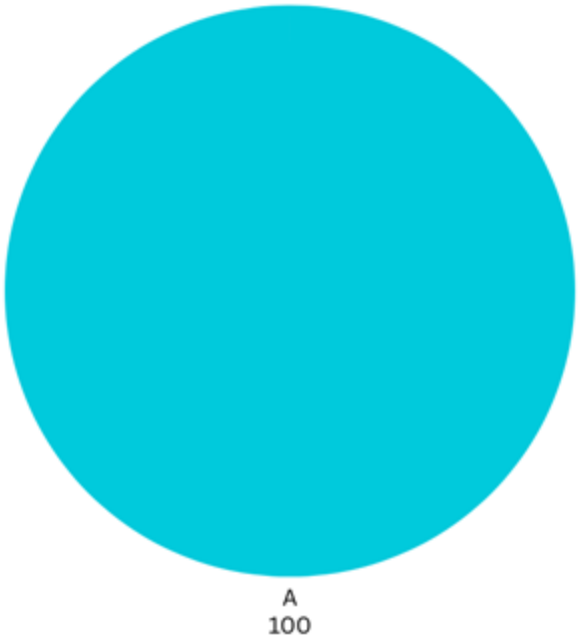
- Linux **does not use a fixed or absolute timeslice**
  - An effective timslice depends on:
    - **System load:** how many runnable processes share the CPU
    - **Process weight (priority)**



- Scheduling behavior
  - When **P** becomes runnable:
    - Scheduler **compares** how much CPU time **P** has already received to other runnable tasks
    - If **P** has used **less** CPU time (i.e., is "behind" its fair share), it can **preempt** the currently running process **C**

We need a virtual runtime (vruntime)

- Problem: Real CPU time isn't fair by itself
  - If one task runs alone for 100 ms, it uses the whole CPU
  - When a new task joins, we need to "rebalance"

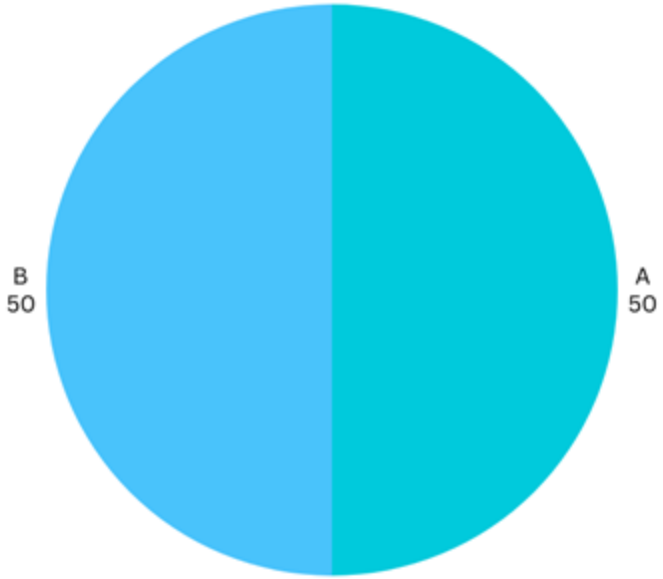


Task	Priority
A	1

Time Quanta:100 ms

We need a virtual runtime (vruntime)

- Problem: Real CPU time isn't fair by itself
  - If one task runs alone for 100 ms, it uses the whole CPU
  - When a new task joins, we need to "rebalance"

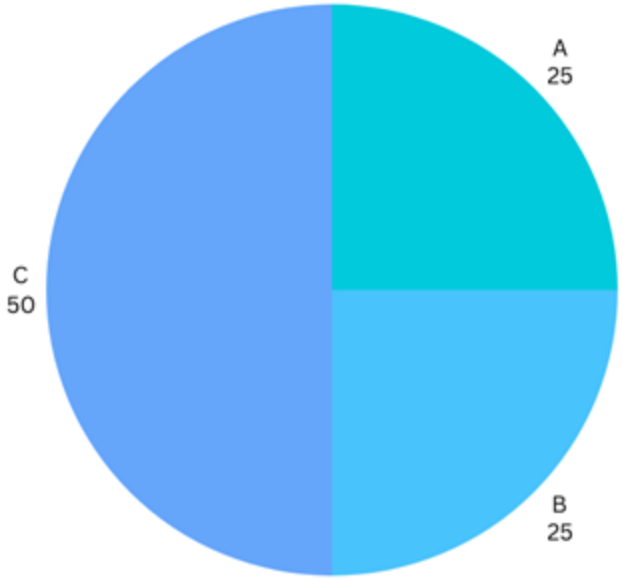


Task	Priority
A	1
B	1

Time Quanta:100 ms

We need a virtual runtime (vruntime)

- Problem: Real CPU time isn't fair by itself
  - If one task runs alone for 100 ms, it uses the whole CPU
  - When a new task joins, we need to "rebalance"



Task	Priority
A	1
B	1
C	2

Time Quanta:100 ms

We need a virtual runtime (vruntime)

- Problem: Real CPU time isn't fair by itself
  - If one task runs alone for 100 ms, it uses the whole CPU
  - When a new task joins, we need to "rebalance"

- vRuntime is that virtual runtime
  - Grows faster when a task runs a lot, slower when others need to be scheduled

We need a virtual runtime (vruntime)

- Instead of slicing the capacity, let's slice the time
- Problem of not accounting dynamic nature

- **vRuntime = time x ∑(weight of all tasks)** - virtual entity to handle dynamic task addition and deletion

How vRuntime evolves?

- Each task maintains its own **virtual clock**
- Whenever it runs, we update it as:
  - curr->vruntime += calc_delta_fair(delta_exec, curr);
    - delta: how long the task actually ran
    - calc_delta_fair: scales that time by the task's weight

$$v_i(t + \Delta t) = v_i(t) + \frac{\Delta t}{\text{weight}i / \text{weightnice 0}}$$

  - High-priority (larger weight) → adds *less* to vRuntime (so stays "behind" longer → gets more CPU)
  - Low-priority (smaller weight) → adds *more* to vRuntime → runs less often

Linux EEVDF

- Earliest eligible virtual deadline first
- New scheduler that replaced the completely fair scheduler (CFS)
- Builds upon CFS but introduces **explicit fairness timing (lag + deadline)**
- Before running a task, scheduler computes **virtual deadline**
  - Add the time remaining in the timeslice to the time it became eligible
    - Longer time slice: later virtual deadline (will run later)
    - Shorter time slice: Run first (denotes latency sensitive tasks)

EEVDF scheduler: example

- 3 CPU-bound tasks start at the same time
- Over those 30 ms, each task should run 10 ms
- Scheduler picks A and runs with a 30 ms timeslice

	A	B	C
Lag:	**0**	**0**	**0**

EEVDF scheduler: example

- 3 CPU-bound tasks start at the same time
- Over 30 ms, each task should run 10 ms
- Scheduler picks A and runs with a 30 ms timeslice
- A is no longer eligible, B is picked up and runs 30 ms

	A	B	C
Lag:	**-20**	**10**	**10**

EEVDF scheduler: example

- 3 CPU-bound tasks start at the same time
- Over 30 ms, each task should run 10 ms
- Scheduler picks A and runs with a 30 ms timeslice
- A is no longer eligible, B is picked up and runs 30 ms
- Now, only C is eligible, which will run next

	A	B	C
Lag:	**-10**	**-10**	**20**

Linux EEVDF design

- Everyone gets a fair slice, but we also prioritize *when* each slice happens (who needs CPU **now**)
- Divides the available CPU time fairly among contending tasks
- **Lag**: expected virtual run time - actual running time of a task

  T(Runnable) – T (Running) → how far a task's actual CPU time deviates from the ideal
  - Positive lag: A task is owed CPU time
  - Negative lag: A task has received more than its share
- **Eligible** task: lag >= 0
- Maintains an invariant: **sum of all the lag value in the system is zero**
- Scheduler chooses a task from a set of eligible tasks

EEVDF scheduler: Time-slice control

- Preemption based on virtual deadlines
  - Enables running short time-slice tasks sooner
  - Short time-slice task can now preempt an already running task
- Tasks can specify desired time slice (100us–100ms)
  - Use **sched_setattr()** system call

EEVDF implementation

- Four main components
  - Time accounting
  - Process selection
  - Scheduler entry poitnt: schedule(), scheduler_tick() (pseudocode)
  - Sleeping and waking up (not covered)

Time accounting in EEVDF

- Calculates the virtual runtime

```
/* linux/include/linux/sched.h */
struct task_struct {
    /* ... */
    const struct sched_class *sched_class; /* sched_class of this task */
    struct sched_entity      se; /* for time-sharing scheduling */
    struct sched_rt_entity   rt; /* for real-time scheduling */
    /* ... */
};
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; /* how much time a process
```

Process selection in EEVDF

- EEVDF maintain an rbtree of tasks indexed by vruntime (i.e., runqueue)
- Always pick a task with smallest vruntime, the left most node

```c
/* linux/kernel/sched/fair.c */
struct sched_entity *__pick_first_entity(struct cfs_rq *cfs_rq) /* CODE */
{
    struct rb_node *left = cfs_rq->rb_leftmost;

    if (!left)
        return NULL;

    return rb_entry(left, struct sched_entity, run_node);
}
```

Add a task to a runqueue

- When a task is woken up or migrated, it is added to a runqueue

```
/* linux/kernel/sched/fair.c */
void enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int flags)
{
    bool renorm = !(flags & ENQUEUE_WAKEUP) || (flags & ENQUEUE_MIGRATED);
    bool curr = cfs_rq->curr == se;

    /* Update run-time statistics */
    update_curr(cfs_rq);

    update_load_avg(se, UPDATE_TG);
    enqueue_entity_load_avg(cfs_rq, se);
    update_cfs_shares(se);
    account_entity_enqueue(cfs_rq, se);
    /* ... */

    /* Add this to the rbtree */
    if (!curr)
        __enqueue_entity(cfs_rq, se);
```

Add a task to a runqueue

- When a task is woken up or migrated, it is added to a runqueue

```
/* linux/kernel/sched/fair.c */
static void __enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se)
{
    struct rb_node **link = &cfs_rq->tasks_timeline.rb_node;
    struct rb_node *parent = NULL;
    struct sched_entity *entry;
    int leftmost = 1;
    /* Find the right place in the rbtree: */
    while (*link) {
        parent = *link;
        entry = rb_entry(parent, struct sched_entity, run_node);
        if (entity_before(se, entry)) {
            link = &parent->rb_left;
        } else {
            link = &parent->rb_right;
            leftmost = 0;
        }
    }
    /* Maintain a cache of leftmost tree entries (it is frequently used): */
    if (leftmost)
        cfs_rq->rb_leftmost = &se->run_node;
    rb_link_node(&se->run_node, parent, link);
```

Remove a task from a runqueue

- When a task goes to sleep or gets migrated, it is removed from the runqueue

```c
/* linux/kernel/sched/fair.c */
void dequeue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int flags)
{
    /* Update run-time statistics of the 'current'. */
    update_curr(cfs_rq);
    update_load_avg(se, UPDATE_TG);
    dequeue_entity_load_avg(cfs_rq, se);
    update_stats_dequeue(cfs_rq, se, flags);
    clear_buddies(cfs_rq, se);

    /* Remove this to the rbtree */
    if (se != cfs_rq->curr)
        __dequeue_entity(cfs_rq, se);
    se->on_rq = 0;
    account_entity_dequeue(cfs_rq, se);
```

```c
static void __dequeue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se)
{
    if (cfs_rq->rb_leftmost == &se->run_node) {
        struct rb_node *next_node;

        next_node = rb_next(&se->run_node);
        cfs_rq->rb_leftmost = next_node;
    }

    rb_erase(&se->run_node, &cfs_rq->tasks_timeline);
}
```

Scheduler class design

- The Linux scheduler is modular and provides a pluggable interface for scheduling algorithms
  - Enables different scheduling algorithms co-exist, scheduling their own types of processes
- **Scheduler class** is a scheduling algorithm
  - Each scheduler class has a priority.
  - E.g., SCHED_FIFO , SCHED_RR , SCHED_BATCH
- The base scheduler code iterates over each scheduler in priority order
  - linux/kernel/sched/core.c: scheduler_tick() , schedule()

Scheduler class design

- Time-sharing scheduling: SCHED_BATCH
  - SCHED_NORMAL in kernel code
  - EEVDF
  - linux/kernel/sched/fair.c

- Real-time scheduling
  - SCHED_FIFO : First in-first out scheduling
  - SCHED_RR : Round-robin scheduling
  - SCHED_DEADLINE : Sporadic task model deadline scheduling

Scheduler class implementation

- sched_class: an abstract base class for all scheduler classes

```
/* linux/kernel/sched/sched.h */
struct sched_class {
    /* Called when a task enters a runnable state */
    void (*enqueue_task) (struct rq *rq, struct task_struct *p, int flags);
    /* Called when a task becomes unrunnable */
    void (*dequeue_task) (struct rq *rq, struct task_struct *p, int flags);
    /* Yield the processor (dequeue then enqueue back immediately) */
    void (*yield_task) (struct rq *rq);
    /* Preempt the current task with a newly woken task if needed */
    void (*check_preempt_curr) (struct rq *rq, struct task_struct *p, int flags);
    /* Choose a next task to run */
    struct task_struct * (*pick_next_task) (struct rq *rq,
                                            struct task_struct *prev,
                                            struct rq_flags *rf);
    /* Called periodically (e.g., 10 msec) by a system timer tick handler */
    void (*task_tick) (struct rq *rq, struct task_struct *p, int queued);
    /* Update the current task's runtime statistics */
    void (*update_curr) (struct rq *rq);
};
```

Scheduler class implementation

- Each scheduler implements its own functions

```
/* linux/kernel/sched/fair.c */
DEFINE_SCHED_CLASS(fair) = {
    /* const struct sched_class fair_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,
    .task_tick          = task_tick_fair,
    .update_curr        = update_curr_fair, /* ... */
};
/* scheduler tick hitting a task of our scheduling class: */
static void task_tick_fair(struct rq *rq, struct task_struct *curr, int queued)
{
    struct cfs_rq *cfs_rq;
    struct sched_entity *se = &curr->se;
    for_each_sched_entity(se) {
        cfs_rq = cfs_rq_of(se);
        entity_tick(cfs_rq, se, queued);
```

Scheduler class implementation

- task_struct maintains scheduler-related fields

```
/* linux/include/linux/sched.h */
struct task_struct {
    /* ... */
    const struct sched_class *sched_class; /* sched_class of this task */
    struct sched_entity      se; /* for time-sharing scheduling */
    struct sched_rt_entity   rt; /* for real-time scheduling */
    /* ... */
};
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; /* how much time a process
```

Scheduler class implementation

- The base scheduler code triggers scheduling operations in two cases

→When processing a timer interrupt (scheduler_tick())

→Updates running time statistics of the task

→Also calculates the global load on the system

→When the kernel calls schedule()

→Picks up the next task that has the highest priority

→While picking the task, scheduler iterates over all each class based on the priority

```
again:
    for_each_class(class) {
        /* In CFS, pick_next_task_fair() will be called */
        p = class->pick_next_task(rq, prev, rf);
        if (p) {
            if (unlikely(p == RETRY_TASK))
                goto again;
            return p;
        }
    }
```

EEVDF on multi-core machines

- Per-CPU runqueues (rbtrees)
  - To avoid costly access to shared data structures
- Runqueues must be kept balanced
  - E.g., dual-core with one long runqueue of high-priority processes, and a short one with low-priority processes
  - High-priority processes get less CPU time than low-priority ones
- A load balancer runs periodically based on priority and CPU usage

Preemption and context switch

- **Context switch** → swap the process currently running on the CPU to another one
- Performed by context_switch() , which is called by schedule()
  - Switch the address space through switch_mm()
  - Switch the CPU state (registers) through switch_to()

Q. When is the schedule called in the kernel?

Preemption and context switch

- schedule() is called:
  - A task can voluntarily relinquish the CPU by calling schedule()
  - A current task needs to be preempted if
    - It runs long enough (i.e., its vruntime is not smallest anymore)
    - A task with a higher priority is woken up

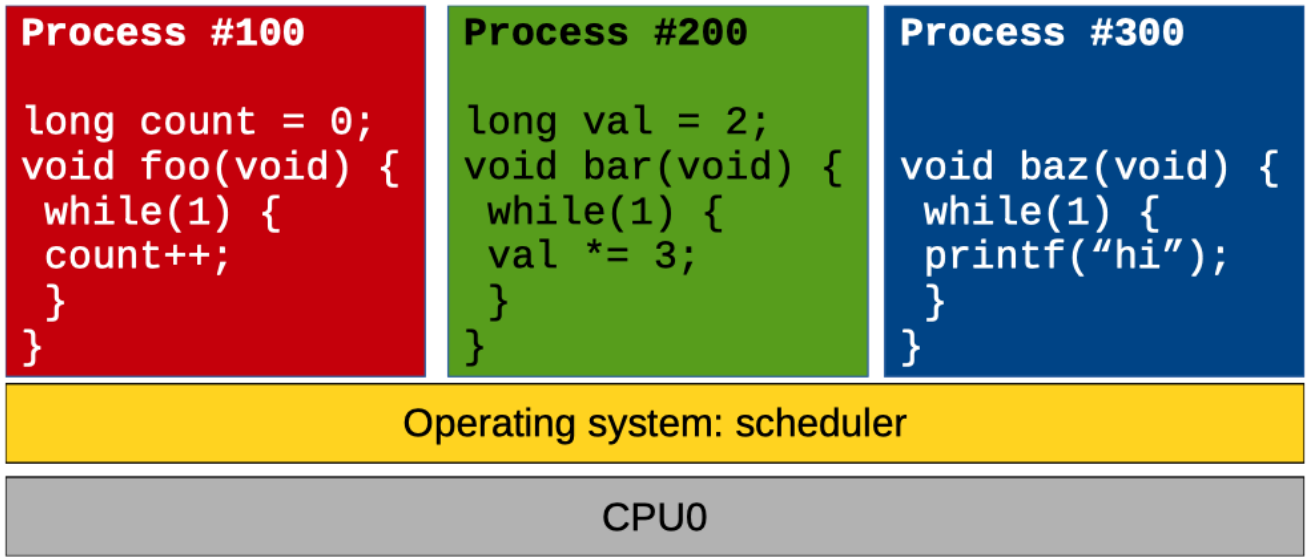
need_resched flag

- A per-task flag that tells the CPU: Need to call the scheduler soon
  - It tells the kernel → please reschedule when safe!

- Events that change the flag:
  - scheduler_tick() → timeslice expired (task used its fair share)
  - try_to_wake_up() → higher-priority (or earlier-deadline) task wakes up
  - Kernel calls:
    - set_tsk_need_resched() → marks current task for rescheduling
    - Later, at safe point: if (need_resched()) → run scheduler
  - After switch → clear_tsk_need_resched() resets flag

need_resched flag

- The need_resched is checked:
  - Upon returning to user space (from syscall or an interrupt)
  - Upon return from an interrupt
- If the flag is set, **schedule** is called



Kernel preemption

- For most UNIX-like operating systems, kernel code is non-preemptive
- In Linux, the kernel code is also preemptive
  - A task can be preempted in the kernel as long as execution is in a safe state without holding any lock
- preempt_count in the thread_info structure indicates the current lock depth
- If (need_resched && !preempt_count) then, it is safe to preempt
  - Checked when returning to the kernel from interrupt
  - Checked when releasing a lock

Kernel preemption

- Kernel preemption can occur:
  - On return from interrupt
  - When kernel code becomes preemptible again
  - If a task in the kernel blocks (e.g., mutex)

```
/* linux/include/linux/preempt.h */
#define preempt_disable() \
do {                      \
    preempt_count_inc();  \
    barrier();            \
} while (0)
#define preempt_enable()  \
do {                      \
    barrier();            \
    if (unlikely(preempt_count_dec_and_test())) \
```