CS345 - Tutorial 4

Implementation of the “Least Slack Time (LST)” scheduling policy in Linux Kernel

Eva Papadogiannaki

papadogian@csd.uoc.gr

Process Scheduling

- Switching from one process to another in a very short time frame
- Scheduler:
  - When to switch processes
  - Which process to choose next
- Major part of the operating system kernel

Scheduler

- Allows the execution of multiple tasks at the “same” time
- Responsible for:
  - Task coordination among processors
  - Avoiding task starvation and preserving fairness
  - Taking into account system-level tasks (e.g., drivers)

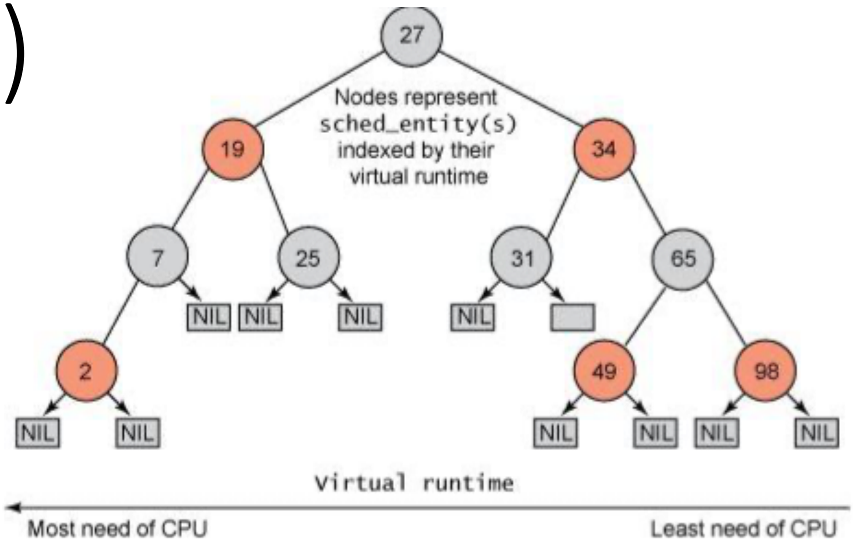
Linux Scheduler

- Executes multiple programs at the “same” time, sharing the CPU with users of varying needs
  - Minimizes response time
  - Maximizes overall CPU utilization
- Ideal Scheduling: *n* tasks share 100/*n* percentage of CPU effort each
- Preemptive: Higher priority processes evict lower-priority running processes
- Quantum Duration
  - Variable
  - Keep it as long as possible, while keeping good response time

History of schedulers in Linux

- Linux v1.2 – Round Robin
- Linux v2.2 – Scheduling Classes & Policies, Categorizing tasks as non/real-time, non-preemptible
- Linux v2.4 – Division in epochs, goodness of function
- Linux v2.6.0 – v2.6.22 –O(1), Runqueues & priority arrays
- Linux v2.6.23 (and after) – Completely Fair Scheduler (CFS)

Completely Fair Scheduler (CFS)

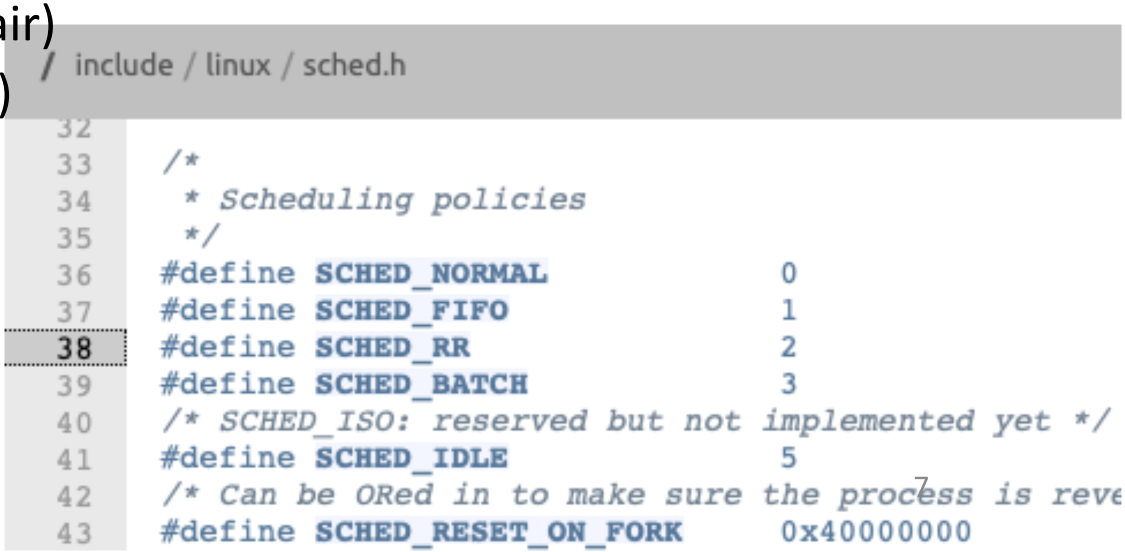


- Virtual time concept
- Running tasks are sorted using “vruntime”
- Time-ordered red-black tree instead of queue
- Maintains balance in providing processor time to tasks
- At each scheduling invocation:
  - The vruntime of the current task is incremented (time spent in processor)
  - The scheduler chooses the leftmost leaf in the tree (lowest vruntime)
- The leftmost node is cached (O(1))
- Reinsertion of a preempted task takes O(logn)

Scheduling Classes and Policies

Modular design to easily support different scheduling policies

- Each task belongs to a scheduling class
- The scheduling class defines the scheduling policy
- Scheduling policy is set by sched_setscheduler()
- Some scheduling policies:
  - SCHED_NORMAL – Default linux task policy (CFS, fair)
  - SCHED_FIFO – Special time-critical tasks (real-time)
  - SCHED_RR – Round-robin scheduling (real-time)



/ include / linux / sched.h

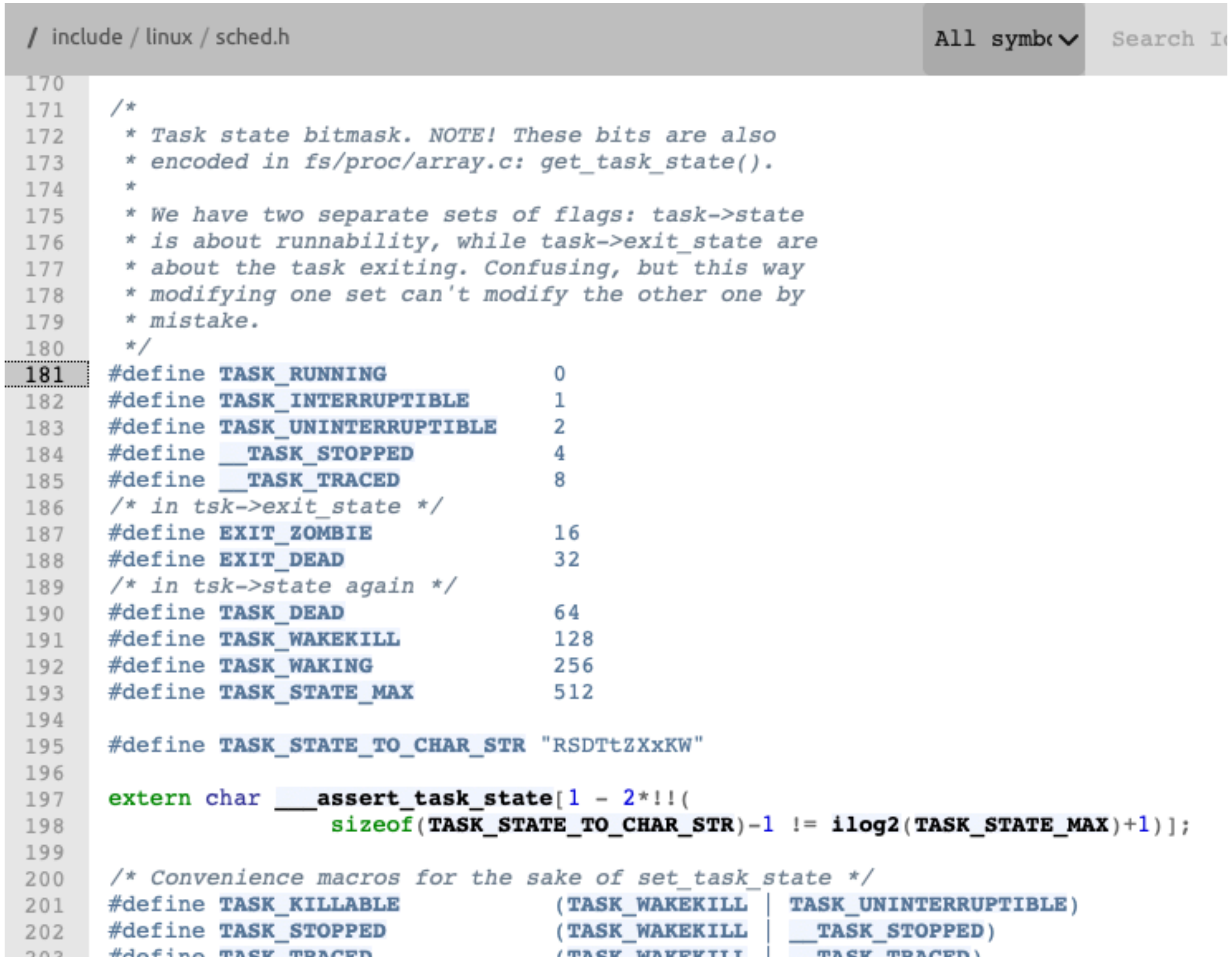
```
/*
 * Scheduling policies
 */
#define SCHED_NORMAL            0
#define SCHED_FIFO              1
#define SCHED_RR                2
#define SCHED_BATCH             3
/* SCHED_ISO: reserved but not implemented yet */
#define SCHED_IDLE              5
/* Can be ORed in to make sure the process is reve
#define SCHED_RESET_ON_FORK     0x40000000
```

Linux Kernel source files

- Browse easily through the Linux Kernel source files using this link https://elixir.bootlin.com/linux/v2.6.38.1/source
- Actual context switch code, runqueue struct definition, etc.
  - kernel/sched.c https://elixir.bootlin.com/linux/v2.6.38.1/source/kernel/sched.c
- Implementation of Completely Fair Scheduling (CFS)
  - kernel/sched_fair.c https://elixir.bootlin.com/linux/v2.6.38.1/source/kernel/sched_fair.c
- Implementation of Real-Time Scheduling (RT)
  - kernel/sched_rt.c https://elixir.bootlin.com/linux/v2.6.38.1/source/kernel/sched_rt.c
- Tasks are abstracted as struct sched_entity and struct sched_rt_entity (for rt class); Also, check struct sched_class
  - include/linux/sched.h https://elixir.bootlin.com/linux/v2.6.38.1/source/include/linux/sched.h

Some code snippets

A task’s scheduling state (defined in include/linux/sched.h)



```
/*
 * Task state bitmask. NOTE! These bits are also
 * encoded in fs/proc/array.c: get_task_state().
 *
 * We have two separate sets of flags: task->state
 * is about runnability, while task->exit_state are
 * about the task exiting. Confusing, but this way
 * modifying one set can't modify the other one by
 * mistake.
 */
#define TASK_RUNNING            0
#define TASK_INTERRUPTIBLE      1
#define TASK_UNINTERRUPTIBLE    2
#define __TASK_STOPPED          4
#define __TASK_TRACED           8
/* in tsk->exit_state */
#define EXIT_ZOMBIE             16
#define EXIT_DEAD               32
/* in tsk->state again */
#define TASK_DEAD               64
#define TASK_WAKEKILL           128
#define TASK_WAKING             256
#define TASK_STATE_MAX          512

#define TASK_STATE_TO_CHAR_STR "RSDTtZXxKW"

extern char ___assert_task_state[1 - 2*!!(
                sizeof(TASK_STATE_TO_CHAR_STR)-1 != ilog2(TASK_STATE_MAX)+1)];

/* Convenience macros for the sake of set_task_state */
#define TASK_KILLABLE           (TASK_WAKEKILL | TASK_UNINTERRUPTIBLE)
#define TASK_STOPPED            (TASK_WAKEKILL | __TASK_STOPPED)
#define TASK_TRACED             (TASK_WAKEKILL | __TASK_TRACED)
```

Some code snippets

The rq (runqueue) struct (defined in kernel/sched.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 because
         * remote CPUs use both these fields when doing load calculation.
         */
        unsigned long nr_running;
        #define CPU_LOAD_IDX_MAX 5
        unsigned long cpu_load[CPU_LOAD_IDX_MAX];
        unsigned long last_load_update_tick;
#ifdef CONFIG_NO_HZ
        u64 nohz_stamp;
        unsigned char nohz_balance_kick;
#endif
        unsigned int skip_clock_update;

        /* capture load from *all* tasks on this cpu: */
        struct load_weight load;
        unsigned long nr_load_updates;
        u64 nr_switches;

        struct cfs_rq cfs;
        struct rt_rq rt;

#ifdef CONFIG_FAIR_GROUP_SCHED
        /* list of leaf cfs_rq on this cpu: */
        struct list_head leaf_cfs_rq_list;
#endif
#ifdef CONFIG_RT_GROUP_SCHED
        struct list_head leaf_rt_rq_list;
#endif

        /*
         * This is part of a global counter where only the total sum
         * over all CPUs matters. A task can increase this counter on
         * one CPU and if it got migrated afterwards it may decrease
```

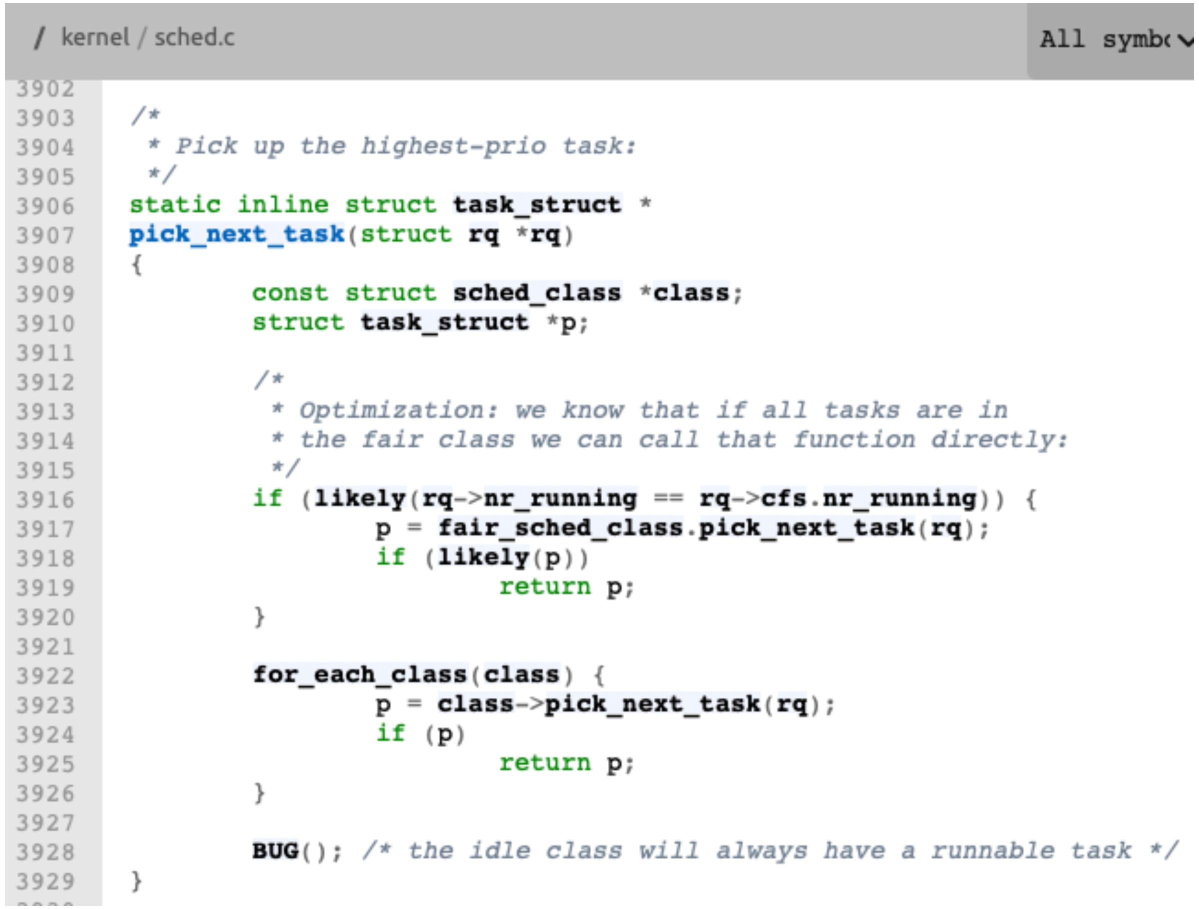
Some code snippets

The schedule function (in kernel/sched.c)

```
/*
 * schedule() is the main scheduler function.
 */
asmlinkage void __sched schedule(void)
{
        struct task_struct *prev, *next;
        unsigned long *switch_count;
        struct rq *rq;
        int cpu;

need_resched:
        preempt_disable();
        cpu = smp_processor_id();
        rq = cpu_rq(cpu);
        rcu_note_context_switch(cpu);
        prev = rq->curr;

        release_kernel_lock(prev);
need_resched_nonpreemptible:

        schedule_debug(prev);

        if (sched_feat(HRTICK))
                hrtick_clear(rq);

        raw_spin_lock_irq(&rq->lock);

        switch_count = &prev->nivcsw;
        if (prev->state && !(preempt_count() & PREEMPT_ACTIVE)) {
                if (unlikely(signal_pending_state(prev->state, prev))) {
                        prev->state = TASK_RUNNING;
                } else {
                        /*
                         * If a worker is going to sleep, notify and
                         * ask workqueue whether it wants to wake up a
                         * task to maintain concurrency.  If so, wake
                         * up the task.
                         */
                        if (prev->flags & PF_WQ_WORKER) {
                                struct task_struct *to_wakeup;

                                to_wakeup = wq_worker_sleeping(prev, cpu);
                                if (to_wakeup)
                                        try_to_wake_up_local(to_wakeup);
                        }
```

Some code snippets

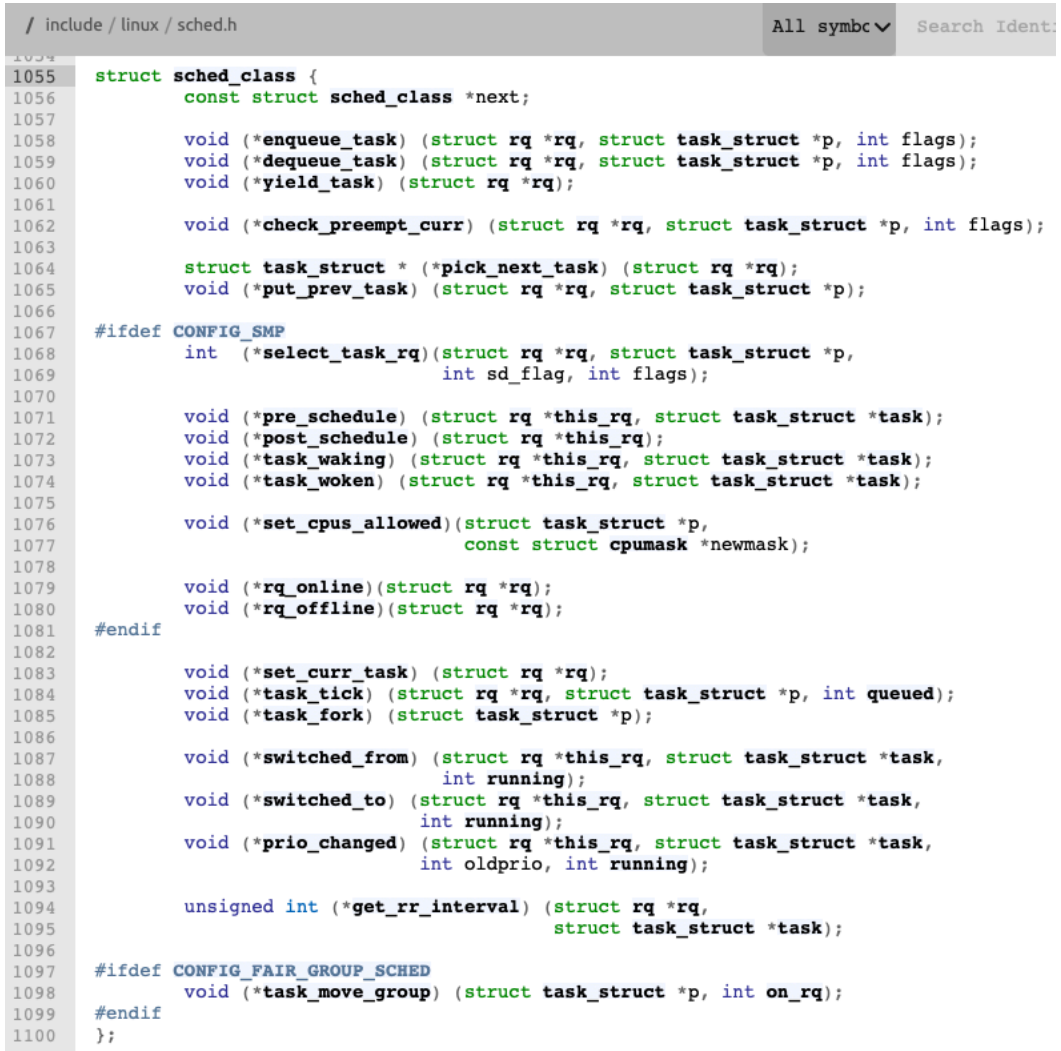
The pick_next_task function (in kernel/sched.c)



```c
/*
 * Pick up the highest-prio task:
 */
static inline struct task_struct *
pick_next_task(struct rq *rq)
{
        const struct sched_class *class;
        struct task_struct *p;

        /*
         * Optimization: we know that if all tasks are in
         * the fair class we can call that function directly:
         */
        if (likely(rq->nr_running == rq->cfs.nr_running)) {
                p = fair_sched_class.pick_next_task(rq);
                if (likely(p))
                        return p;
        }

        for_each_class(class) {
                p = class->pick_next_task(rq);
                if (p)
                        return p;
        }

        BUG(); /* the idle class will always have a runnable task */
}
```

Some code snippets

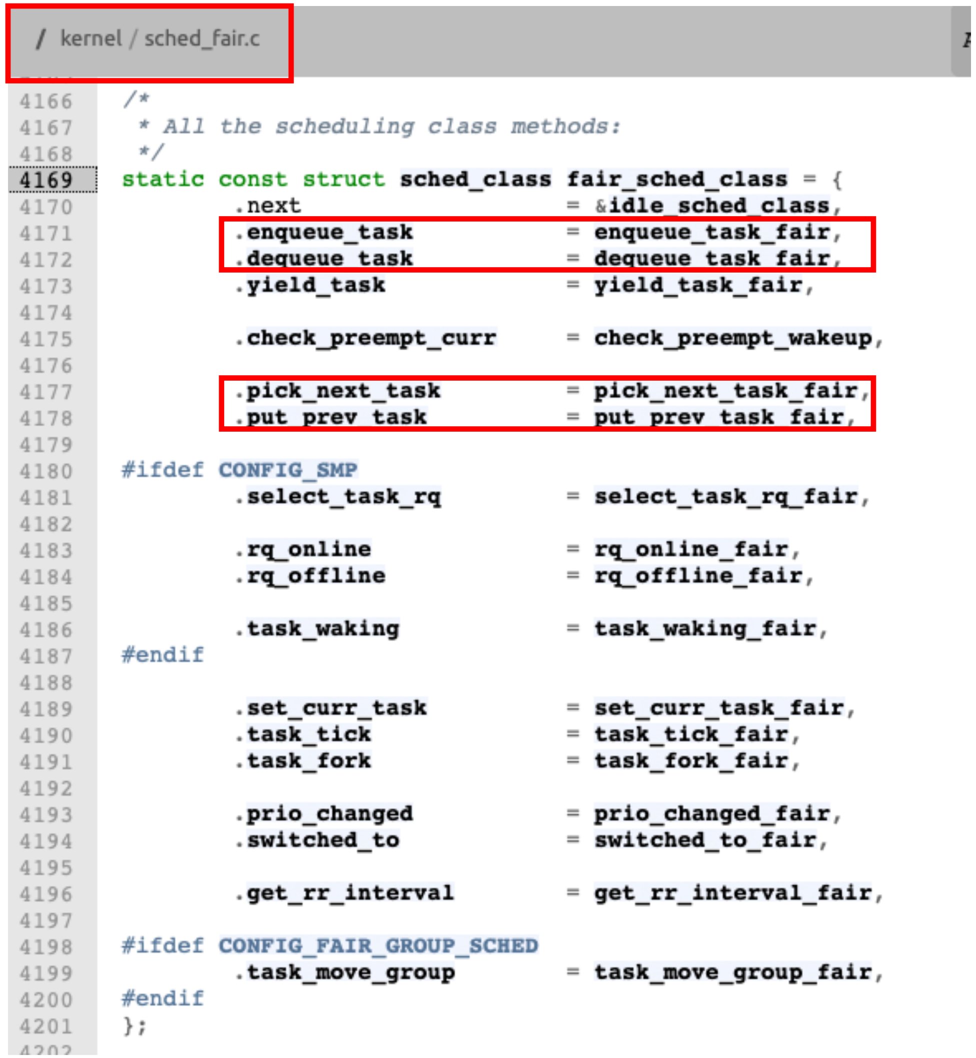
The sched_class struct (defined in include/linux/sched.h)



```
struct sched_class {
        const struct sched_class *next;

        void (*enqueue_task) (struct rq *rq, struct task_struct *p, int flags);
        void (*dequeue_task) (struct rq *rq, struct task_struct *p, int flags);
        void (*yield_task) (struct rq *rq);

        void (*check_preempt_curr) (struct rq *rq, struct task_struct *p, int flags);

        struct task_struct * (*pick_next_task) (struct rq *rq);
        void (*put_prev_task) (struct rq *rq, struct task_struct *p);

#ifdef CONFIG_SMP
        int  (*select_task_rq)(struct rq *rq, struct task_struct *p,
                               int sd_flag, int flags);

        void (*pre_schedule) (struct rq *this_rq, struct task_struct *task);
        void (*post_schedule) (struct rq *this_rq);
        void (*task_waking) (struct rq *this_rq, struct task_struct *task);
        void (*task_woken) (struct rq *this_rq, struct task_struct *task);

        void (*set_cpus_allowed)(struct task_struct *p,
                                 const struct cpumask *newmask);

        void (*rq_online)(struct rq *rq);
        void (*rq_offline)(struct rq *rq);
#endif

        void (*set_curr_task) (struct rq *rq);
        void (*task_tick) (struct rq *rq, struct task_struct *p, int queued);
        void (*task_fork) (struct task_struct *p);

        void (*switched_from) (struct rq *this_rq, struct task_struct *task,
                               int running);
        void (*switched_to) (struct rq *this_rq, struct task_struct *task,
                             int running);
        void (*prio_changed) (struct rq *this_rq, struct task_struct *task,
                             int oldprio, int running);

        unsigned int (*get_rr_interval) (struct rq *rq,
                                         struct task_struct *task);

#ifdef CONFIG_FAIR_GROUP_SCHED
        void (*task_move_group) (struct task_struct *p, int on_rq);
#endif
};
```

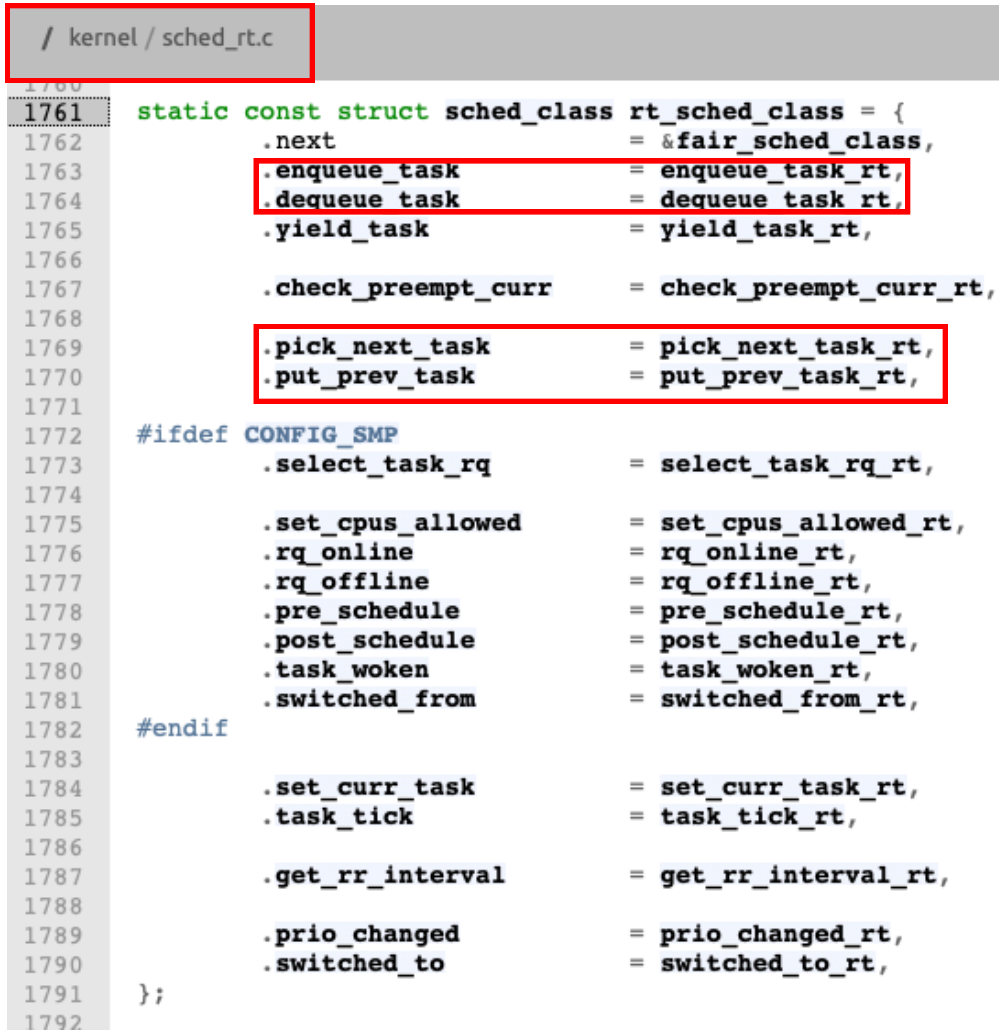
Some code snippets

Handling struct sched_class for fair vs. rt scheduling class



/ kernel / sched_fair.c

```
/*
 * All the scheduling class methods:
 */
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,

#ifdef CONFIG_SMP
        .select_task_rq         = select_task_rq_fair,

        .rq_online              = rq_online_fair,
        .rq_offline             = rq_offline_fair,

        .task_waking            = task_waking_fair,
#endif

        .set_curr_task          = set_curr_task_fair,
        .task_tick              = task_tick_fair,
        .task_fork              = task_fork_fair,

        .prio_changed           = prio_changed_fair,
        .switched_to            = switched_to_fair,

        .get_rr_interval        = get_rr_interval_fair,

#ifdef CONFIG_FAIR_GROUP_SCHED
        .task_move_group        = task_move_group_fair,
#endif
};
```



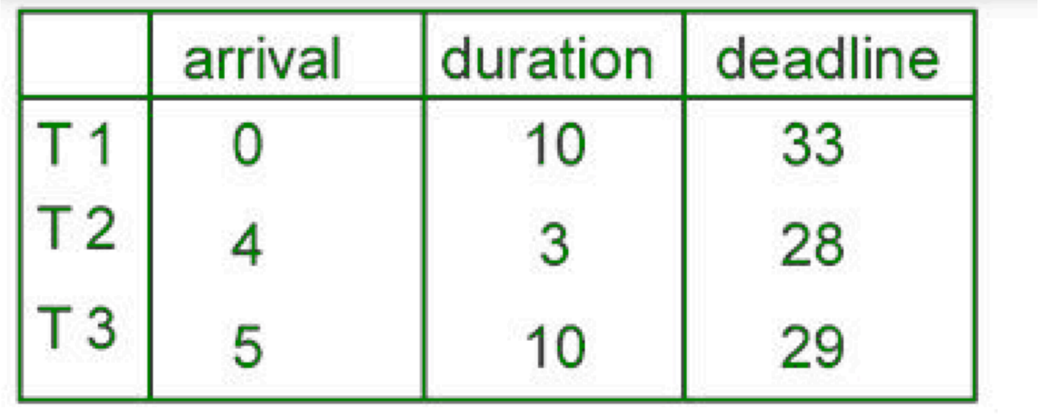
/ kernel / sched_rt.c

```
static const struct sched_class rt_sched_class = {
        .next                   = &fair_sched_class,
        .enqueue_task           = enqueue_task_rt,
        .dequeue_task           = dequeue_task_rt,
        .yield_task             = yield_task_rt,

        .check_preempt_curr     = check_preempt_curr_rt,

        .pick_next_task         = pick_next_task_rt,
        .put_prev_task          = put_prev_task_rt,

#ifdef CONFIG_SMP
        .select_task_rq         = select_task_rq_rt,

        .set_cpus_allowed       = set_cpus_allowed_rt,
        .rq_online              = rq_online_rt,
        .rq_offline             = rq_offline_rt,
        .pre_schedule           = pre_schedule_rt,
        .post_schedule          = post_schedule_rt,
        .task_woken             = task_woken_rt,
        .switched_from          = switched_from_rt,
#endif

        .set_curr_task          = set_curr_task_rt,
        .task_tick              = task_tick_rt,

        .get_rr_interval        = get_rr_interval_rt,

        .prio_changed           = prio_changed_rt,
        .switched_to            = switched_to_rt,
};
```

Assignment 4 – LST Scheduling Algorithm

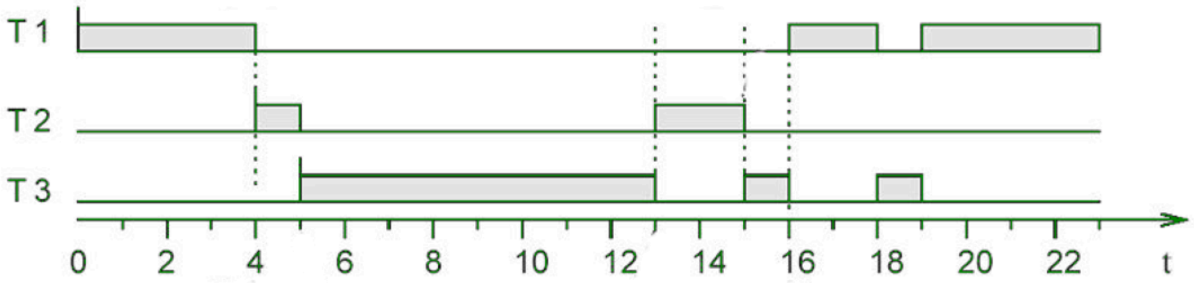
- Implement the Least Slack Time (LST) scheduling algorithm, which assigns priority based on the slack time of a process
- Slack_time = deadline – (computation_time – elapsed_runtime) – time
- Use your code from Assignment 3
    - You will need the set_deadlines system call you implemented
- Use the guidelines from the previous assignment to compile Linux Kernel and run it

Assignment 4 – LST Scheduling Algorithm



	arrival	duration	deadline
T 1	0	10	33
T 2	4	3	28
T 3	5	10	29

- At time t=0: Only task T1, has arrived. T1 is executed till time t=4.
- At time t=4: T2 has arrived.
  Slack time of T1: 33-4-6=23
  Slack time of T2: 28-4-3=21
  Hence T2 starts to execute till time t=5 when T3 arrives.
- At time t=5:
  Slack Time of T1: 33-5-6=22
  Slack Time of T2: 28-5-2=21
  Slack Time of T3: 29-5-10=12
  Hence T3 starts to execute till time t=13
- At time t=13:
  Slack Time of T1: 33-13-6=14
  Slack Time of T2: 28-13-2=13
  Slack Time of T3: 29-13-2=14
  Hence T2 starts to execute till time t=15
- At time t=15:
  Slack Time of T1: 33-15-6=12
  Slack Time of T3: 29-15-2=12
  Hence T3 starts to execute till time t=16
- At time t=16:
  Slack Time of T1: 33-16-6=11
  Slack Time of T3:29-16-=12
  Hence T1 starts to execute till time t=18 and so on..



Example from: https://www.geeksforgeeks.org/least-slack-time-lst-scheduling-algorithm-in-real-time-systems/

Assignment 4 – Pre-processing and Filtering

- Before *schedule()* selects the next process
- Scan all processes in the runqueue list and find if there is any process that has a deadline (deadline != -1)
    - Calculate its slack time
        - If this process has exceeded the given deadline, remove this process from the runqueue list so it'll never be executed
        - If not, iterate the runqueue list rq. For each process p, check if p has less slack value
            - If so execute process p first

Assignment 4 – Demonstrate the new scheduler

Create at least 1 demo.

This demo should do the following:

- Create up to 10 child processes
- For each child process, the parent process will set its remaining computation time and its deadline (set deadline to gettimeofday() + 100 seconds)
- Each child process will spin for the given computation time (e.g., use while, for)

Guidelines for Assignment 4

- Browse kernel code with: https://elixir.bootlin.com/linux/v2.6.38.1/source
- Another way to map source code is by using ctag: http://www.tutorialspoint.com/unix_commands/ctags.htm
- Understand how the scheduler works
  - For example, you can start with printing inside the *schedule()* function
- Follow the function call path from schedule in order to find out how the next task is picked
- Use the printk() function often, its syntax is close to printf and it's an easy way to observe the kernel's behaviour from the user level (with *dmesg*)
- Reuse existing code snippets within the kernel source code (e.g., to traverse data structures or access members in struct nodes)
- Compile after small changes in the source code (good for easy debugging)
- Submit anything you can that helps you show your effort!