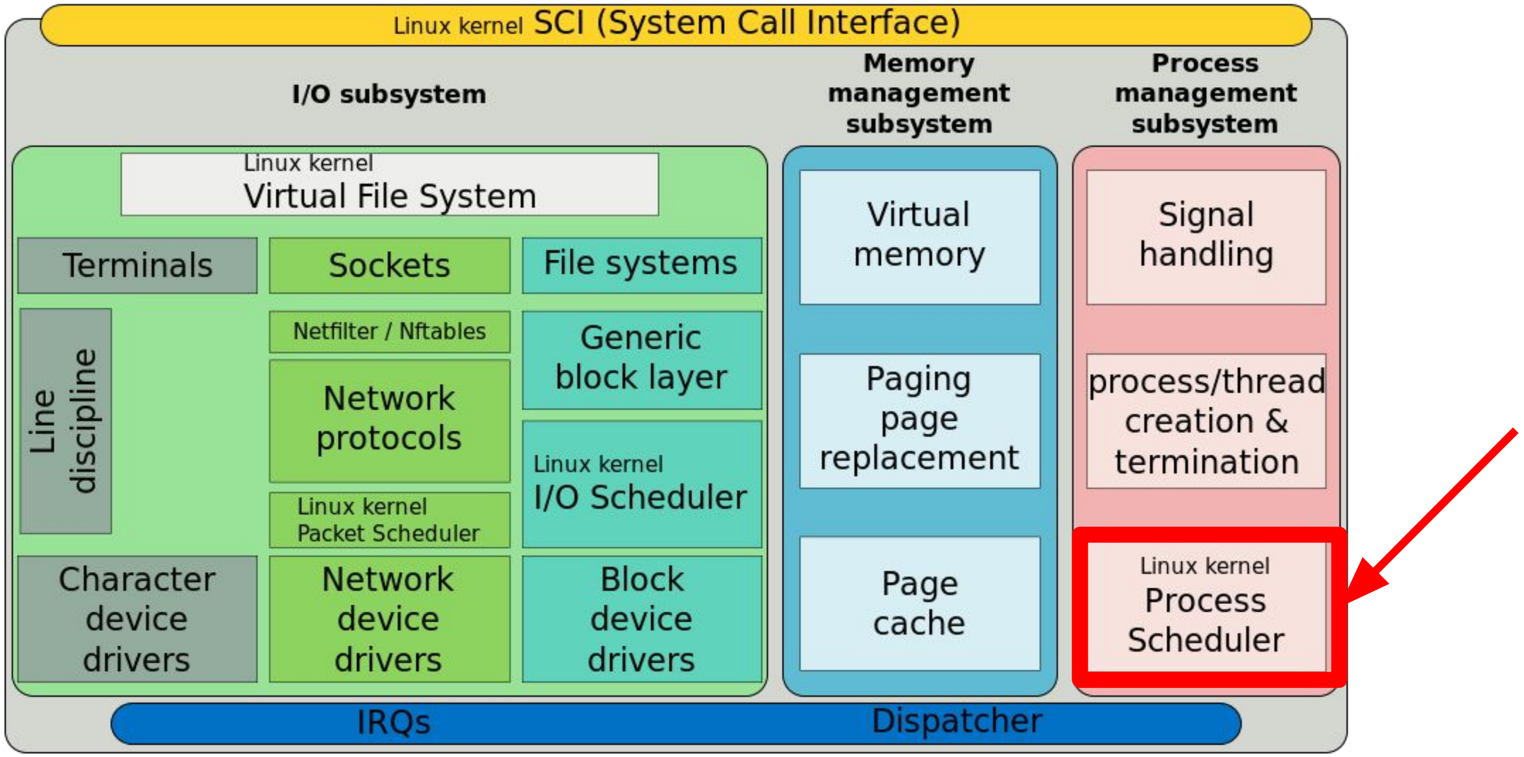
Linux Scheduler
(Φροντιστήριο για την 4η σειρά)

christou@csd.uoc.gr

What is a scheduler



Why is it usefull

- Many tasks have to run in parallel
- Almost all times tasks are more than the CPU cores (i.e. playing music while talking on skype and playing a game…)

The Scheduler is responsible:

- To coordinate how tasks, share the available processors (how much time each (**Quantum**))
- To avoid task starvation and preserve fairness (i.e. music will continue while gaming)
- To also take into account system tasks (e.g. drivers...)

Linux Scheduler - definition

- The scheduler makes it possible to execute multiple programs at the "same" time, thus sharing the CPU with users of varying needs.
    - minimizing response time
    - maximizing 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

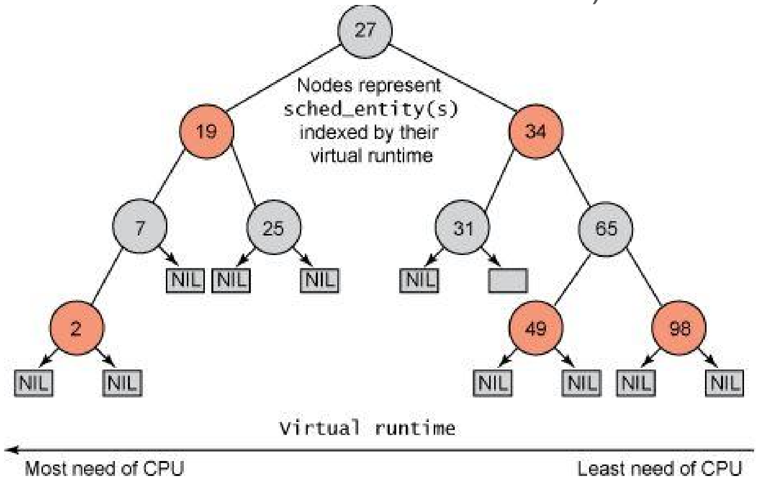
History of schedulers in Linux

- v1.2 : circular queue, round robin (RR) policy
- v2.2 : scheduling classes, categorizing tasks as non/real-time, non-preemptible
- v2.4 : O(n) scheduler,
  - each task could run a quantum of time, each epoch
  - epoch advances after all *runnable* tasks have used their quantum
  - At the beginning of each epoch, all processes get a new quantum
  - **BUT** lacked scalability (O(n)) and was weak for real-time tasks
- v2.6 : Completely Fair Scheduler (CFS)

Assignment version

CFS

- Time-ordered red-black tree “timeline” of future task execution
- Runnable tasks are sorted using “vruntime”
- At each scheduling invocation:
  - the vruntime of the current task is incremented (time it spent using the CPU)
  - the scheduler chooses the leftmost leaf in the tree (i.e the task with the smallest vruntime)
- Leftmost node is cached (O(1)),
  reinsertion of a preempted task takes O(logn)



CFS scheduling classes

Modular design in order to easily support different scheduling policies

- Each task belongs to a scheduling class
- The scheduling class defines the scheduling policy
- fair sched class: the CFS policy
- rt sched class: implements SCHED_FIFO (queue) SCHED_RR policies
  - priority run queues for each RT priority level
  - 100ms time slice for RR tasks

Files in Linux source

- Actual context switch, runqueue struct definition (rq, cfs_rq, rt_rq)
  - kernel/sched.c
- Completely Fair Scheduler, implementation of CFS
  - kernel/sched_fair.c
- Real Time Scheduling, rt implementation
  - kernel/sched_rt.c
- Tasks are abstracted as struct sched_entity and struct sched_rt_entity (for rt class), also sched_class struct
  - include/linux/sched.h

Some code (sched.c)

```
asmlinkage void __sched schedule(void)
{
        struct task_struct *prev, *next;              previous and next (new) tasks
        unsigned long *switch_count;                  statistics
        struct rq *rq;                                the processor's runqueue (1 in this assignment)
…
        preempt_disable();                            disable preemption (avoid schedule inside
        cpu = smp_processor_id();                     schedule)
        rq = cpu_rq(cpu);
        rcu_note_context_switch(cpu);
        prev = rq->curr;                              previous is the current task running
…
        put_prev_task(rq, prev);                      put prev task in the runqueue, in
        next = pick_next_task(rq);                    this functions the appropriate put/pick
…                                                     function is called depending the
        if (likely(prev != next)) {                   scheduling class
…
                context_switch(rq, prev, next);       the actual context switch
```

also in sched.c….

```
static inline struct task_struct *
pick_next_task(struct rq *rq)
{
        const struct sched_class *class;
        struct task_struct *p;
        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;
        }
```

The function that chooses next task

First check CFS rq

Macro to traverse the list of sched classes

Which sched class has our demo program? printk function, can help.

...then in sched_fair.c

```
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,
```

next sched class in the sched class list
the class specific functions
all _fair functions are implemented in
this file.

For this assignment

- Implement Least Time Remaining scheduling algorithm
- At each scheduling interval, decrement the remaining time of the *current* (preempted) task, if the updated remaining time is negative, set infinite flag
- Choose as next, the task with the least remaining time of completion
    - Iterate the processes in the runqueue and find the minimum
- If the next is the same with the preempted, no need for preemption
- If all processes have the infinite flag set, use the default Linux Scheduler behaviour

Continue from assignment 3

- Use your code from assignment 3 to start
  - You will use *set_total_computational_time* system call to set the remaining time for a process
- Use the guidelines from previous assignment in order to compile Linux kernel and run your kernel image

How to test

- Create simple programs that initialy set their total_computation_time
  - total_computational time should be different for each
  - (10 - 20 seconds difference should be good)
- Then, each will *spin* for some time (don’t use sleep, a large while maybe...), the *spin* should be the same for each program
- After spinning, each program should print a unique identifier
- What is the expected behaviour??

Guidelines 1/2

- Familiarize with http://lxr.free-electrons.com/source/?v=2.6.38
    - You can find function implementation, struct definition, etc… within clicks
- Another way to *map* source code is by using ctags
    - http://www.tutorialspoint.com/unix_commands/ctags.htm
- Use printk function, its syntax is quite the same as printf and it's an easy way to observe the kernel behaviour from user level (with dmesg command)
- Kernel data structures implementation is quite different from what you have learned till now
    - https://isis.poly.edu/kulesh/stuff/src/klist/ ,lists examples
    - Search for examples for other data structures also
    - Also check the APIs for each data structure in include/linux folder

Guidelines 2/2

- Understand how the scheduler works
  - start with printing things inside schedule function
- Follow the function call path from schedule in order to find out how the next task is picked
  - Also printing
- Reuse existing code snippets within the kernel source in order to do what you want
  - e.g. reuse code snippets for accessing members in struct nodes, traversing data structures...
- Compile often with small changes in the source from the previous compilation
  - Massively helps with debugging
- Submit anything you can to show your effort!