Linux Scheduler
(Φροντιστήριο για την 4η σειρά)

tsirbas@csd.uoc.gr
What is a scheduler
Why is it useful?

- 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)
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(log n)
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)

3934 asmlinkage void __sched schedule(void)
3935 {
3936     struct task_struct *prev, *next;
3937     unsigned long *switch_count;
3938     struct rq *rq;
...
3942     preempt_disable();
3943     cpu = smp_processor_id();
3944     rq = cpu_rq(cpu);
3945     rcu_note_context_switch(cpu);
3946     prev = rq->curr;
...
3986     put_prev_task(rq, prev);
3987     next = pick_next_task(rq);
...
3991     if (likely(prev != next)) {
...
3999     context_switch(rq, prev, next);
also in sched.c....

3906 static inline struct task_struct *
3907 pick_next_task(struct rq *rq) \hspace{10em} The function that chooses next task
3908 {
3909     const struct sched_class *class;
3910     struct task_struct *p;
3916     if (likely(rq->nr_running == rq->cfs.nr_running)) { \hspace{1em} First check CFS rq
3917         p = fair_sched_class.pick_next_task(rq);
3918         if (likely(p))
3919             return p;
3920     }
3922     for_each_class(class) { \hspace{1em} Macro to traverse the list of sched classes
3923         p = class->pick_next_task(rq);
3924         if (p)
3925             return p;
3926     }
3927     return NULL; \hspace{1em} Which sched class has our demo program? printk function, can help.
...then in sched_fair.c

4169  static const struct sched_class fair_sched_class = {
4170     .next = &idle_sched_class, // next sched class in the sched class list
4171     .enqueue_task = enqueue_task_fair, // the class specific functions
4172     .dequeue_task = dequeue_task_fair, // all _fair functions are implemented in
4173     .yield_task = yield_task_fair, // this file.
4175     .check_preempt_curr = check_preempt_wakeup,
4177     .pick_next_task = pick_next_task_fair,
4178     .put_prev_task = put_prev_task_fair,
For this assignment

- Implement Shortest Remaining Time scheduling algorithm
- At each scheduling interval, choose the task with the shortest remaining time, which has not run this second as the next task
- Set the scheduled out task as inactive
- After the second has passed, set the inactive tasks as active
- If all processes with a deadline have run or there are no other processes with a deadline, use the default Linux Scheduler behaviour
Continue from assignment 3

- Use your code from assignment 3
  - You will need set_demand system call

- Use the guidelines from the previous assignment in order to compile Linux Kernel and run it.
How to test

- Create simple programs that initially set their deadlines
- Then, each will *spin* for some time
- Your scheduler should print (using printk)
  - The PID of the task it picked next
  - Its deadlines
  - The processes with deadline at every second (i.e. when they are activated again)
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
- 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!