COMSW4118-17-1 11/2/2017

- How to write a scheduler. Look in few places: (3.10 kernel)
  - /kernel/sched ← define scheduling class, look at idle/rt/fair.
  - /include/linux/sched.h
    - (1) Define a scheduling class
    - enqueue_task_rt uses sched_rt_entity defined in
    - sched.h also has sched_entity within the task_struct
  - The entity is the thing that is getting scheduled. Rather than schedule tasks directly, you go through entity (which could be used to represent a group of tasks, for example).
    An entity eventually gives you a task, since this is what gets ultimately scheduled.
  - (2) Define an entity for the new scheduler (class specific one) in include/linux/sched.h
    - Add stuff that your scheduler may need there: ex. timeslice
    - You will need to link list entities together since you will need to organize them in some structure later (runqueue or sth. like that).
    - From task_struct, you have ~~pointers~~ member vars to the "encapsulating" entities
  - (3) Add the entity member to task_struct in sched.h and make sure the entity can support being in a queue or list (ex. next pointer).
  - kernel/sched/sched.h ← check out what's there for rt.

rt_rq and cfs_rq ← look we have class-related runqs in here!

these are the class-specific runqueues.

(?) Both are part of struct rq ← the main, per-CPU runqueue (that has a rung lock!)

(4) Add a class-specific runqueue for the new scheduling class to the general runqueue and define that new rung in sched/sched.h ← THIS IS WHERE YOU KEEP THE ENTITIES FOR THE NEW SCHEDULER.

NOTE: the RT scheduler defines an array of subqueues "struct rt_prio_array active" in "struct rt_rq" ← one subqueue for each priority level 0..99.

⑤ Add initialization for the new runque ~~in sched~~ similar to:
init_cfs_rq and init_rt_rq in sched/sched.h.
those are external in sched.h and defined in rt.c, etc.

NOTE: Searching code for "sched_rt" and "_class" (found "sched_class *" in linux/sched.h but said not important)

▶ struct sched_class in sched/sched.h is the general scheduling class. We also have init_sched_rt_class(void)

⑥ Add initialization for the new scheduling class.

▶ Note that in sched/sched.h the ~~classes~~ scheduling classes are listed as "extern const struct sched_class" and there are "sched_class_highest" and "for_each_class" macros

⑦ Add the new scheduling class to the scheduling class list

▶ kernel/sched/core.c is the core area of the scheduler
- Look for class-specific stuff there: "_rt_", etc. see if they matter.
- Look for system calls that set:
  - ⊛ scheduling parameters
  - ⊛ what scheduling class you are part of

! Look for SYSCALL_DEFINE

! EX: __setscheduler(...)

⑧ Hook up your new scheduling class here, so it gets used

__sched_setscheduler(...) ← gets task, policy, parameters and assigns itself to a scheduling class

⑨ Add new scheduling class here, if necessary.

! Basically, find all SYSCALL_DEFINES in sched/core.c and see if they do anything that is scheduling-class-specific and if we need to modify them for the new scheduling class.

- --sched_fork() - sets up scheduler stuff for new tasks

(10) Modify sched_fork for new scheduling class, also see sched_fork() w/o --at start!

NOTE: Here we set the policy constant to SCHED_NORMAL, which is the policy for the CFS class, ~~another~~ we need to change that default? (this is done for non-RT tasks) If we want to start w/ the new class.

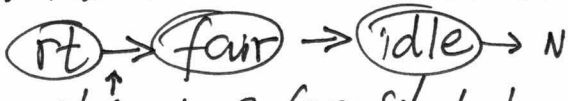
Most of the stuff we want to do is in the class specific file:
/kernel/sched/ idle_task.c, rt.c, fair.c

(11) Define a new class (idle_task.c is simplest example but rt.c is more useful)

Some key elements of the class and related functions

(A) init_rt_rq - initialize the RT runque (the CFS init is much simpler)

(B) .next is the link to the next scheduling class in list:



Add new class here b/w rt and fair

...

(C) enqueue_task and dequeue_task ← check out RT implementations and CFS implementations

NOTE: "for_each_sched_entity" is for group scheduling, if you don't worry about that this loop gets executed only once.

enqueue_entity calls __enqueue_entity and this:

enque_entity_rt calls
① gets the rung array
② figures out where to put in the array

Essentially, we find the entity and put it on a rung - the complexity comes because RT allows for groups where entities form a tree-like structure. If we don't worry about this, it is simpler:

① take the task, find the scheduling entity associated w/ it
② stick it on your runque which is a list or sth. like that

dequeue_task is the opposite of enqueue_task

for RT: dequeue_rt_entity → dequeue_rt_stack → dequeue_rt_entity
should be simpler w/o groups

(D) pick_next_task

for RT: pick_next_task → pick_next_task_rt → _pick_next_task_rt

Go thru the runque and find next entity to go to.

(!) rt_task_of is what is used to find the task of an entity!

HINT: Maybe use a list of entities and don't search the runque but just pick the entity at the head and then get its task and return it. (list_head and list_entry/container_of)

(E) You probably want to implement everything that is implemented in idle_task.C - don't need to actually do work, but set the function pointer. EX: prio_changed and switched_to do nothing for idle, but the functions are defined!

NOTE: prio_changed has code for fair.c but if you don't have priorities, then you probably don't have to worry about it (except define & set pointer) switched_to has code for fair.c - maybe could be left empty or the resched flag could be set

get_rr_interval - nothing for idle.

(?) → Do we need to do nothing/return 0 here or something else?

check_preempt_curr - for idle calls resched, since idle tasks are unconditionally rescheduled

(!) → Do we do nothing?

put_prev_task - called right before pick_next_task in core.c.

(!) Do we need to do sth. to the task that was already running before picking the next task? PROBABLY!

set_curr_task - nothing for idle, for fair.c it "accounts for a task changing its policy or group." Gets called when a task migrates b/n classes. If a task is currently running & its scheduling parameters change (ex. scheduling

COMSW4118-17-5 11/2/2017

Class changes, do you need to do anything special for it?
PROBABLY!

dequeue, enqueue, and pick_next_task are the functions that must be defined for sure!

F) task_tick is the other thing you probably need.
gets called on a scheduler tick. task_tick_rt finds the scheduling entity, then (for RR tasks only, FIFO have no slice) decrements the time slice, checks to see if it has expired and, if so, update the timeslice back to starting count, and re-enqueue to end of runque and set_tsk_need_resched() since task_tick gets called in the context of a timer interrupt, so we can't call schedule() directly.

G) yield_task - gets called when a task yields the CPU: for RT what we do is requeue the task

H) SMP Stuff (ifdef CONFIG_SMP)

- select_task_rq
- pre_schedule
- post_schedule
- ? Do we need this?

nothing for idle (idle tasks not moved)
for RT: gets called from fork, wakeup, i.e. when a task becomes runnable and we need to pick a CPU to put the task on its rung.
LOCKING is required & think about deadlock!

Called before pick_next_task, toward the beginning of schedule. Do we need to do sth. before pick_next_task that is SMP related?

HINT: More things between runqueues, for example, before picking the next task to run!

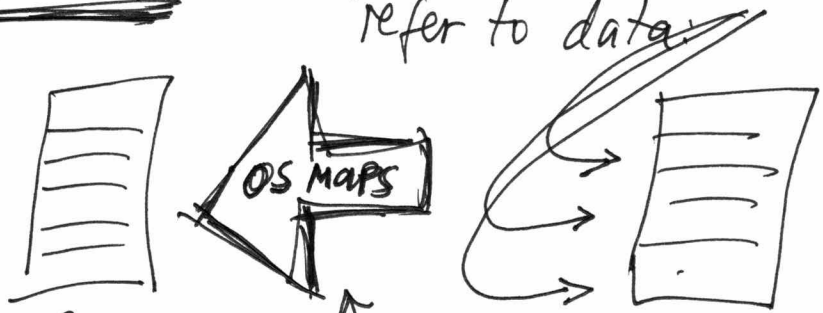
SUMMARY:
① linux/sched.h - add entity stuff (definition & task_struct member)
② sched/sched.h - define custom rung & insert in general rung
③ core.c - check system calls and the underlying set_scheduler & set_sched_scheduler that set scheduling parameters & set scheduling policies
④ class-specific.c - implementation of scheduling class:
everything in idle_task.c + enqueue + yield_task + task_tick + select_task_rq + pre_schedule + post_schedule (?)

COMSW4118-17-6 11/2/2017

- Group scheduling - the "for_each_..." macro in scheduler code has to do w/ group scheduling
- An entity may have several tasks - i.e. a group of tasks are scheduled as a unit. Ex: while(1) fork(); will kill system if scheduled as separate entities w/ separate time quanta. With group scheduling, we have the original sched_entity and all subsequent tasks will be part of it (as a tree). When we schedule we only look at that one group.
- The "for_each..." gets to the leaves in the entity group tree and gets their tasks.
- sched_entity has a parent if we use groups, if parent is NULL, then the "for_each..." loop is executed only once.

MEMORY:

▶ processes have virtual addresses that refer to data



Benefits: conserve space: say you do malloc(20GB) and exit() - the OS would not really reserve 20GB of physical memory for you.

▶ When and How does mapping occur?

- compile-time ← you would map all virtual addresses to physical ones, but that would not work well w/ multiple processes, also, this is not feasible for dynamic allocation.
- load-time ← see what is free at time of loading and then map to physical addresses that are free. Still not feasible for dynamic allocation & not efficient.
- Execution-time ← as the program is running, the mapping b/n virtual & physical memory is adjusted, if necessary. Most flexible but overhead of translation for each memory access is not feasible unless hardware support is available.

Needs hardware support! (ex. paging)