1044414: Advanced Operating Systems and Virtualization **Spring 2018**

Lecture 16: May 4

Lecturer: Alessandro Pellegrini *Scribe: Anxhelo Xhebraj*

16.1 Linux Scheduler

One of the fundamentals things in unix are priorities that are related to the niceness of the processes.

In linux there is notion of real time processes. It is soft real time. Hard real time is not really supported in Linux. Soft real time there are boundaries but some times the OS might miss deadlines.

Real time priorities in linux go from 0 to 99. Higher value means higher priority differently from niceness.

In standard priorities higher values mean lower priorities and they directly map to niceness form 100 to 139 ([-20, 19]).

in task struct there are a lot of variables related to the priorities of each thread. Depending on the strategy of scheduling multiple variables can be used.

- static priority is the one given by the user in kernel representation
- normal priority: if there are multiple tasks that have the same static priority but belong to different policies will get different normal priorities
- prio is dynamic, if there is some process with a really high priority and taking a lot of cpu this value is changed to preempt such process
- real time priority

effective prio computes the priority and uses normal prio. Both take as parameter task struct so they have info about the process. normal prio Maps rt priority to kernel representation or the static priority.

The time used by a process is defined by the time slice assigned to it. Linux manages this change through load weights. In task struct there is a sched entity which inside has load weight. Unsigned long has the weight.

sched prio to weight contains one entry for each nice value. These values are used to determine the stretch a process gets with respect to other tasks in the system.

16.2 Scheduling Classes

Sched rr is real time round robin

sched other/sched normal, round robin time-sharing depending on nice values

sched deadline (3.14): constant bandwith server (CBS) algorithm on top of earliest deadline first

sched deadline (4.13): CBS replaced with Greedy Reclamation of Unusued Bandwith

There are some specific tasks that should be carried out by the scheduler during time. The tasks performed by the scheduler depend on the scheduling class and therefore the sched class struct is needed.

enqueue task enques a task dequeue task it re

yield task is called when a process yields spontaneously the CPU for example in FIFO real time

check preempt curr: checks whether the

put prev task and pick next taks permits to perform the context switch.

select task has a variable that specifies a cpu. In SMP we would like to have the scheduler work in isolation on each core and allow tasks to migrate to different cores. This allows to choose which task should be moved from one core to another.

There is one instance of the idle task for each core in the system to allow multiple cores to be idle.

16.3 Run Queues

A run queue is a set of tasks that are logically in the ready state (can be activated). Different tasks are grouped in different run queues depending the policy. A run queue is a per cpu variable.

- nr running: the number of tasks in the run queue
- curr, idle: pointers to the currently running task and the idle process
- rt: for a real time scheduler strategy for the run queue
- cfs: same as rt but for fair scheduling

16.4 Wait Queue

Mentioned when running block devices. For example reading from a disk may be blocking and therefore the thread should be put to sleep. We have to keep track that some task is waiting for some event. Any number of Wait Queues can be defined and are used to put to sleep threads waiting for some event.

Thundering Herd: lots of processes waiting for the same event overload the schedule since many task need to be moved from the wait queue to the run queue.

Wait queue have been completely changed. It is composed of a list linked through the list head. A function pointer wakes up a task.

A task leaving in a waitqueue can be exclusive (?).

add wait queue allows to add a task to some wait queue. WQ flag exclusive is cleared.

add wait queue exclusive does the same with exclusive set.

The list is paritioned into two parts: the first one with not exclusive while the second exclusive. An exclusive wakeup moves just one task of the queue while a not exclusive wakes up all non-exclusive.

wake up interruptible is related to special threads of the linux kernel. sync immediately reschedules some process with higher priority.

16.5 Thread States

- Task running: the task is running on some cpu core
- task zombie: the task has exited but the parent has not asked for the exit value yet (process cannot be scheduled but pcb exists still)
- interruptible/uninterruptible: tasks that are sleeping on some wait queue for some condition. Interruptible: if a signal is triggered for that process then it should be scheduled to handle the signal
- task killable: similar to task uninterruptible but is vulnerable to sigkill.

16.6 PID Management

current allows to know the currently scheduled task on a cpu.

Multiple hashtables allow to efficiently find processes from pid/tgid/pgrp/session.

struct pid: identifies processes in the kernel word and allows to link them to pids in user space world. Useful to know what user space pid belongs to what namespace.

pids within a namespace start always from 0. This pids are then mapped to the actual pid of the system.

Managing tasks has become very difficult. Reference to task structs might not allow to remove PCBs. Storing a pid of a process would create consistency problems because on fork pids are reused therefore a pid might point to a totally different user space process.

struct pid is the only way to access PCBs in a consistent manner.

- count: reference counter tells how many execution traces are referring to this struct
- hlist

Up to 2.6.26 there was find task by pid that allowed to get a PCB from the integer representing the pid. Afterwards the api has been replaced by find task by vpid. Vpid is a virtual pid that takes into account that different namespaces assign the same pid to different processes. These APIs where implemented on top of hash tables.

From 4.14 up radix trees are used. Pids are replaced by idrs. idr is a sparse array. Vpid is split into blocks of 6 bits. Items in the tree are idrs.

Scheduler entry points are multiple. A direct invocation can be issued through `schedule()` and a lazy invocation setting the `need_resched` variable.

`schedule_tick` eventually calls update rq clock which tells how many jiffies have passed since some last event in a specific run queue.

`schedule` is also called when some process enters to sleep for example on read moving a task to the wait queue.

`wake_up*()`.

16.7 Scheduler in 2.4

Had a linear complexity with respect to the number of tasks. Time is divided into epochs and once an epoch finishes, meaning that all processes in a queue have run at least once. If some processes didn't use the whole processing time they had, in the successive epoch half more quantum of the leftover time is given to them.

`schedule(void)`

The idle task is picked and the list of the run queue of the cpu is scanned. If some process could be scheduled (affinity with the current cpu), its goodness was computed. The process with higher goodness is then scheduled.

One single run queue for all the cores.

If `c == 0` then a new epoch needs to be started.

Disadvantages:

- non runnable and runnable tasks mixed in the same run queue
- contention on the list on SMP systems

16.8 Scheduler in 2.6.8

runqueue has two arrays of struct prio array. Active and expired pointers that point to them. Any time a process ends its quantum it is moved to expired. At some point the expired group is moved to the active by swapping just the pointers (new epoch by simply switching to pointers).

Each prio array has a vector of queues of processes. The size of the vector is 140. Bitmap with bit for each entry of the array telling whether there is at least one process in the queue associated with the bit.

Higher priorities have a larger time slice. At position 0 priority 100, position 1 priority 99 etc.

Every 200ms a CPU checks if some other CPU has more processes in its runqueue to rebalance the load. If the CPU is idle then it checks each ms to see if it can schedule some process.

16.9 Staircase Scheduler

Rank based scheme that works better on environments with more or less 10 CPUs.

16.10 Completely Fair Scheduler

Uses a red black tree where each node keeps the total execution time in nanoseconds from last epoch start and maximum execution time for each process. Right side processes that have run longer on the cpu while on the left the contrary. The process to be scheduled is always found on the left branch of the tree.

`__switch_to` updates the TSS of the current cpu and updates control registers. Change stack pointer and therefore execution context. It is also saved for rescheduling again the process.