Emily Soto

Implementing a Linux Scheduler

Introduction

The kernel's scheduling code is complicated.

- Today we'll be focusing on Linux's implementation of scheduling and scheduling classes

- We'll refrain from digging too far into the code itself, but focus on a conceptual understanding of what the scheduler actually *does*

- Certain bullet points in this presentation link to Bootlin, showing you the line number and file where the relevant code is. Click on anything underlined for this.

Review: Giving up the CPU

Yielding and Preemption

When a task yields it voluntarily stops running.

When a thread is preempted, the kernel forces the task to stop running unless it's holding a spinlock.

Q: What might happen if we could preempt tasks holding spinlocks? Why is it okay to preempt semaphores or mutexes?

Run Queues and Wait Queues

Any task in state TASK_RUNNING is either currently running or on the runqueue.

Any task in TASK_INTERRUPTIBLE/UNINTERRUPTIBLE is on a wait queue for some event

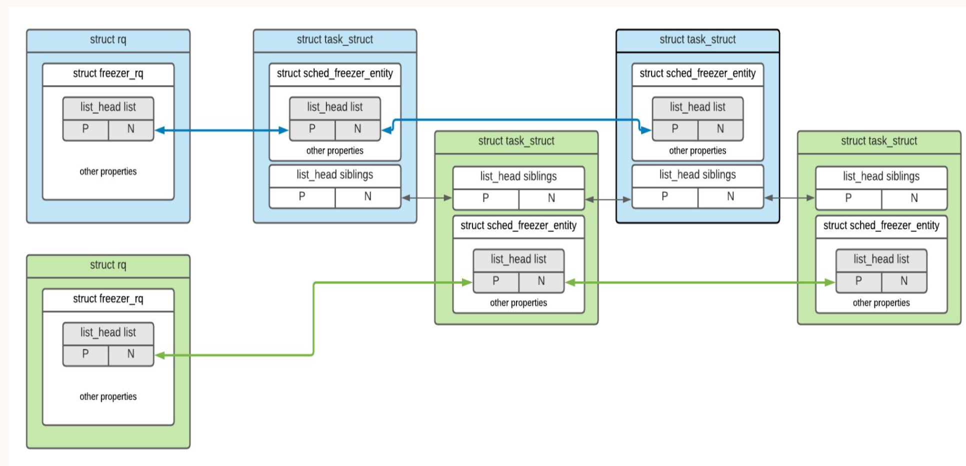
Scheduler Overview

- The scheduler functions at a **per-task** granularity and asks a series of questions
    - Recall: A task is a thread. A multithreaded process can have multiple associated tasks.

1. What task runs next?
2. How long does it have to run?
3. Where should it run?

Sched_Entity

- Embedded in task struct
- Connects tasks in a runqueue structure
- Note: not all runqueues are simple linked lists



cred. Dave Dirnfield

schedule Function

schedule()

- Context switches to another class; sets off the scheduling loop
- Each CPU calls `schedule()` individually
  - If they get too busy they try to redistribute their tasks
- Wrapper function for `schedule_loop` which then calls `__schedule`

Breaking Down `__schedule`: Preliminaries

1. Find the current CPU and the currently running task (`prev`)

2. Check if we can safely context switch (`sched_debug`)

3. Disable local interrupts; we don't want to be interrupted here

4. Lock the runqueue `rq`

5. Memory barriers: ensure all memory operations are finished

Breaking Down `__schedule`: Picking `next`

1. If the task yielded, check for pending signals
   1.1. If the task has any then `next = prev`
   1.2. Else deactivate the task (remove it from the runqueue)

2. `pick_next_task` *asks* the various scheduling classes to nominate the best task to run next. Note: You'll have to implement the logic for finding this task in the assignment

3. If there are no tasks ready to run, the kernel picks the special `idle` task to keep the CPU in a low-power state

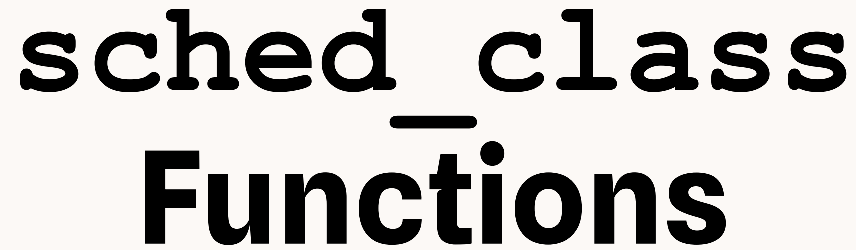
Breaking Down `__schedule`: Context Switch

1. Check if next is the same as prev.
   1.1. If it is, no switch is needed; the function just unlocks everything and continues.

2. The kernel updates the `rq->curr` pointer to the new task.

3. A memory barrier: ensures all memory operations from the old task are done before the new task starts

4. The `context_switch` function
   4.1. Saves the CPU registers of prev
   4.2. Saves the stack pointer of prev
   4.3. Loads the registers and stack pointer of next
   4.4. Changes the memory address space if next is a different process


sched_class
Functions

`struct sched_class`

- An interface every new scheduling class must follow
- To create a new scheduling class we must instantiate a struct `sched_class` and set its function pointer members to point to functions that we define

Daily Routine: Time-Related Functions

- `update_curr`: Called constantly (ticks, before switching, during wakeups) to calculate the ns a task has spent using the CPU

- `task_tick`: Called every hardware timer interrupt to check if a task if a task is out of time. If so, it sets a `need_resched` flag to trigger a switch.

Hand-Off: `pick_next_task` and `put_prev_task`

- `pick_next_task`: Called inside __schedule to figure out what task to run immediately
- `put_prev_task`: Called inside pick_next_task before a switch, typically to clean up the task that stops running

State Changes: `enqueue_task` and `dequeue_task`

- `enqueue_task`: Called whenever a task becomes runnable (changes its state to `TASK_RUNNING`)
  - Example: A parent forks a child process

- `dequeue_task`: Called whenever a task is no longer runnable (changes its state from `TASK_RUNNING`)
  - Example: A task reading from a slow hard drive goes to sleep while it waits to read data

Bouncers: `select_task_rq` and `wakeup_preempt`

- `select_task_rq`: Selects the runqueue for a task, called during a wakeup on multi-core systems, when a process is forked, and more
  - Example: Task A wakes up and we must decide to put it on CPU 0, where it was before, or move it to CPU 3
- `wakeup_preempt`: Called immediately after a task is woken up
  - Example: A high-priority Video Player wakes up. We must decide whether to immediately kick the low-priority Background Downloader off the CPU or let the video player wait a few milliseconds

Lifecycle Functions

- `task_fork`: Called when a process creates a child, typically must decide how much of the parent's timeslice the child should inherit

- `task_dead`: Called when after a context switch when the previous to cleanup any scheduling data when the previous function finished execution

Configuration Functions

- `switched_to`/`switched_from`: Called when a user changes a task's policy
  - Example: You're running a database backup task with "Normal" priority (CFS). It's taking too long so you use `chrt` to change it to "Real-Time" (FIFO) priority
- `balance`: Called when a CPU is idle to see if it can steal a task from another CPU
- `migrate_task_rq`: Called to move a task from one CPU's rq to another CPU's rq

Scheduling Story: Database-Query Task

Assume we have a task DB-QUERY that is currently sleeping because it's waiting for a network packet.

1. select_task_rq
2. migrate_task_rq
3. enqueue_task
4. wakeup_preempt
5. pick_next_task
6. set_next_task

Scheduling Story: Swapping CPUs

Suddenly the System Administrator decides to move all database tasks to core 7.

7. set_cpus_allowed
8. put_prev_task
9. balance
10. context_switch

Scheduling Story: Exiting

11. yield_task or dequeue_task
12. task_dead

Tips

- Per-CPU kthreads: respect them. They **have** to run on a given CPU.
- Read the debugging guide and set up the serial console
- Think very carefully about where you lock. Draw out different threads if necessary.
- Start early!

Debugging

Introduction to Linux Kernel Debugging

- Debugging the kernel is difficult because it lacks the safety and isolation layers provided to userspace programs.
- **Kernel OOPS**: Triggered by specific errors like invalid memory access.
- **Kernel Lockup:** Causes the kernel to hang, often due to deadlocking.
- **Leaks, Races, and Undefined Behavior**: Hardest to catch; they affect execution over time rather than causing immediate crashes.

Recommended Config

- **CONFIG_DEBUG_INFO**: Compiles with -g flags to include function names and symbols.
- **CONFIG_KALLSYMS**: Keeps the symbol map in memory for crash reporting.
- **CONFIG_DEBUG_BUGVERBOSE:** Makes BUG() output much more informative.
- **CONFIG_PRINTK_CALLER:** Adds thread/process info to logs, useful for multithreaded debugging.

Memory Debugging Tools

- **KASAN**: A dynamic safety detector for use-after-free and buffer overflows.
- **Kmemleak**: Acts like "Valgrind" for the kernel to find memory leaks.
- **UBSAN**: Checks for "undefined behavior" (e.g., divide by zero, shifts) at compile time.
- **Stack Checking**: **CONFIG_SCHED_STACK_END_CHECK** crashes the kernel on stack overruns to prevent corruption.

Concurrency Debugging

Lockdep (**CONFIG_DEBUG_LOCKDEP**): Checks for circular dependencies to detect potential deadlocks before they happen

CONFIG_PROVE_LOCKING: Enables extensive runtime lock checks

CONFIG_DEBUG_ATOMIC_SLEEP: Alerts you if a function that might sleep is called in an atomic section

Lockup Detectors: **SOFTLOCKUP_DETECTOR** and **HARDLOCKUP_DETECTOR** print alerts when a CPU is stuck

Printk and Serial Console

- **printk()**: The kernel version of printf(), supporting log levels like **KERN_ERR** and **KERN_DEBUG**.
- **dmesg**: Command used to view the kernel log ring buffer.
- <u>**!!Serial Console!!**</u>: *Essential* for when the kernel crashes before logs can be read.
  - Redirects messages to a file on the host machine via a virtual serial port.
  - Requires editing boot parameters (e.g., console=ttyS0).

For more information please read through our debugging guide linked **here**

Acknowledgement

- Previous TAs Ryan Wee, Hans Montero, Tal Zussman, Denzel Farmer, Andy Cheng, and Jeremy Carin

Questions?