Project: Round-Robin Scheduler in Linux

PURPOSE

- Getting familiar with the Linux kernel source code.
- Understanding process scheduling and how different parameters can affect scheduling
- Develop and evaluate a custom scheduler for the Linux kernel.
- Add a system call to modify a parameter for your custom scheduler.

OVERVIEW

In this project, you will examine process scheduling in Linux. You will modify the Linux kernel source code to implement a round robin scheduling algorithm. You will also need to provide a system call to allow you to select and set parameters for your scheduler. Finally, you will place a multi-threaded process under the control of your scheduler and analyze its behavior with different time slice values. You will build and test your custom kernel image on the kernel virtual machine (KVM) setup described in the lab sessions.

DESCRIPTION

Linux scheduling framework

The Linux scheduler can be activated in two different ways: (1) when a task goes to sleep (voluntary yield), or (2) when the scheduler tick timer is expired. In either case, the kernel scheduler is invoked to schedule the next task. Linux provides a generic scheduling framework that supports multiple different schedulers, called scheduling classes, and a new scheduler can be seperately developed and integrated within the framework.

Every task belongs to exactly one of the scheduling classes, and each scheduling class is responsible for managing the tasks associated with the scheduling class. The generic scheduler is not involved in managing tasks at all; this is completely delegated to the scheduler classes.

Round-robin scheduler

For this project, you will implement a round robin scheduler in your custom Linux kernel. The round-robin scheduling algorithm arranges processes in a queue in the order they are received (first-come, first-serve order). Each process has a time slice that is reduced as the process runs. Once the process has used all of its allotted slice, this value is reset to the default quantum value, and the process is placed at the end of the queue. Additionally, the default quantum value will be configurable via a custom system call. If the user sets the

default quantum to 0, your scheduler should run each process in the queue (without preemption) until its completion (that is, FCFS scheduling).

To guide your implementation, we have declared the data structures and scheduling class you should use to implement the round robin scheduler. The implementation will be confined to `kernel/sched.c` and `kernel/sched_other_rr.c`. The most important data structures for your implementation are discussed below. In this list, items described in sub-bullets are defined as members of the structure described in the parent bullet.

- `struct rq` – the generalized per-CPU run queue structure. This is not the queue of tasks that you will manipulate for round robin scheduling. Rather, this structure contains a more specific run queue type for each scheduler class (as well as some other fields relevant to all scheduling classes). For this assignment, you will focus on the `other_rr_rq` structure.
    - `struct other_rr_rq` – contains fields related to the run queue for the round robin scheduler. You can access the appropriate other rr rq object from within your scheduling class implementation by doing `rq->other_rr`.
        - `struct list_head queue` – the actual run queue for the round robin scheduler. You can use the kernel list API (defined in `include/linux/list.h`) to manipulate the queue. In particular, you should use the following methods for your round robin implementation (you might not need to use all of these):
            - `list add(new, head)`
            - `list add tail(new, head)`
            - `list del(entry)`
            - `list entry(ptr, type, member)`
        - `unsigned long nr running` – a count to keep track of how many processes are on the run queue. You will have to maintain this count when you insert and remove tasks from the queue.
        - You can ignore the other members of the `other_rr_rq` structure. These are used to balance the thread load on systems with multiple compute cores.
    - `struct task_struct *curr` – the task struct pointer to the current (running) process on the run queue.
- `struct task_struct` – the process control block for Linux processes and tasks. This is defined in `include/linux/sched.h`. You do not need to modify this structure.
    - `struct list head other_rr_run_list` – list item to insert tasks on the other_rr run queue.
    - `unsigned int policy` – current scheduling policy of this task (e.g. `SCHED NORMAL, SCHED IDLE, SCHED OTHER RR`, etc.).
    - `unsigned int task_time_slice` – current time slice for this task (for use with the other_rr scheduler).

- `unsigned int other_rr_time_slice` – the default time slice for each task with the other_rr scheduler. This is a global value defined in `kernel/sched.c`.

The scheduling class for the round robin implementation is in kernel/sched other rr.c. This file includes several stub functions. Some of these are already complete or partially complete. You will complete the implementation of the round robin scheduler by filling out these functions.

The round robin scheduler maintains statistics for its running processes. There is also functionality for balancing compute load on machines with multiple compute cores. You will not need to concern yourself with these aspects of the scheduler. For this project, you will only need to focus on implementing the round robin semantics described above. Read the following list carefully, as it describes exactly which functions in the other rr scheduling class that you will need to modify:

- `update_curr_other_rr` – updates the current tasks runtime statistics. You do not need to modify this function.
- `enqueue_task_other_rr` – adds a new process to the run queue. This is called when a process changes from a sleeping state to a runnable state. **You need to complete the implementation of this function.**
- `dequeue_task_other_rr` – takes a process off the run queue. This happens when a process switches from a runnable to an un-runnable state, or when the kernel decides to take it off the run queue for other reasons. **You need to complete the implementation of this function.**
- `requeue_task_other_rr` – moves a task from the head of the run queue to the end of the run queue. You do not need to modify this function.
- `yield_task_other_rr` – happens when the current process is relinquishing control of the CPU voluntarily. This is invoked from the sched yield system call. This should move the current task to the end of the run queue. **You need to complete the implementation of this function.**
- `check_preempt_curr_other_rr` – happens when we need to preempt the current task with a newly woken task. This is called, for instance, when a new task is woken up with wake up new task. For the round robin scheduler, you do not need to take any action here. You do not need to modify this function.
- `pick_next_task_other_rr` – selects the next task which is supposed to run. This should return a pointer to the task at the head of the run queue (or NULL if the run queue is empty). **You need to complete the implementation of this function.**
- `put_prev_task_other_rr` – called before the currently executing task is replaced with another one. Note that `pick_next_task` and `put_prev_task` are not equivalent to putting tasks on and off the run queue like enqueue task and dequeue task. Instead, they are responsible for giving the CPU to a task and taking it away, respectively. Switching between tasks, however, still requires performing a low-level context switch. You do not need to modify this function.

- `load_balance ...` – several functions are used to balance system load on machines with multiple cores. These are `load_balance_start_other_rr, load_balance_next_other_rr, load_balance_other_rr, and move_one_task_other_rr`. You do not need to modify any of these functions.
- `task_tick_other_rr` – the scheduler's 'timer tick'. This is called by the periodic scheduler each time it is activated. If the default quantum value has been set to 0, the current task should hold the CPU until it completes (FCFS semantics). In that case, there is no action required and this function should return immediately. Otherwise, each task should execute in intervals equal to the quantum value. To accomplish this, this function should decrement the current task's time slice to indicate that it has run for 1 time unit. When a task has run for its entire allotted time slice (i.e. the task's time slice reaches 0), this function should reset its time slice value, move it to the end of the run queue, and set its `TIF_NEED_RESCHED` flag (using the `set_tsk_need_resched(task)` function) to indicate to the generic scheduler that it should be preempted. **You need to complete the implementation of this function.**
- `set_curr_task_other_rr` – updates statistics for the current task when the scheduling policy changes. You do not need to modify this function.
- `switched_to_other_rr` – the current process switched to the other rr scheduling policy. You do not need to modify this function.
- `select_task_rq_other_rr` – called when a task that was not previously on a run queue (e.g. because it was blocked or newly created) needs to be placed on a run queue. You do not need to modify this function.

Modifying `sched_setscheduler` to Select your Scheduler

Applications do not directly interact with the functions defined in the scheduler class. Applications use these functions by selecting from the provided scheduling policies using the sched setscheduler system call. This system call uses several helper functions to set the scheduling class for the calling task. You need to modify (`sched_setscheduler`) to recognize `SCHED_OTHER_RR` as a valid scheduling policy instead of returning an error. You also need to modify setscheduler to set the task's sched class appropriately when the `SCHED_OTHER_RR` policy is specified. You should print out a message at this point (using printk) to indicate that the `SCHED_OTHER_RR` policy has been selected.

Adding a system call to set the quantum

You will also implement a system call that enables you to change the time slice value (or quantum) that is allocated to each process when it is inserted into this scheduler's run queue:

```
sched_other_rr_setquantum( unsigned int quantum )
```

The system call should take a time slice value (in jiffies) and should set the global variable other rr time slice that you should use to reset each process' time slice value when it is

inserted into the run queue. A larger time slice value should enable longer execution intervals for each process. If the user sets the time slice to 0, the scheduler should not preempt any process that holds the CPU until it has completed (first-come, first-serve scheduling).

The actual system call implementation should be kept in kernel/sched.c. You can create an entry point for a system call using the `DEFINE SYSCALL`*`N`* macro. The N should match the number of arguments passed to your system call. In this case, the system call takes only one argument, so you will use the `DEFINE SYSCALL1` macro. The arguments for this macro should be the name of the system call followed by the types and names for each argument (see other examples of how to create system calls with this macro in `kernel/sched.c`). You can provide the implementation of your system call in the body of this function.

Next, you will need to modify several files to ensure that the system call is installed and linked against correctly. The following list names each file you need to change and describes the necessary modifications. For your implementation, replace *system_call_name* with the name of your system call.

- `arch/x86/kernel/syscall_table_32.S` – This file contains system call names. Add the following line to the end of this file:
  `.long sys_`*`system_call_name`*
- `arch/x86/include/asm/unistd_32.h` – This file contains the system call number that is passed through the %eax register when a system call is invoked. Add the following line to this file:

  `#define __NR_`*`system_call_name last_system_call_num`* `+ 1`

  Where last system call num is the number of last system call listed in this file. You should also increment the NR syscalls macro by 1.
- `include/linux/syscalls.h` – This file contains declarations for system calls. Add a declaration for your system call:

  `asmlinkage long sys_`*`systemc_call_name`*`(unsigned int)`

Finally, you will need to test your system call implementation. Print out a message in your system call implementation (using printk) to indicate that your system call has been reached. After this is in place, your kernel modifications are complete and you can build and install your kernel with a custom system call. Unfortunately, you cannot call your system call directly from user applications. Your system call name is not visible to user programs as this would require modifications to system header files and / or library software. Thus, you should use the syscall system call to invoke your system call with the system call number you assigned it in `unistd_32.h`:

```
#include <unistd.h>
```

```
#include <sys/syscall.h>
int syscall (int number, ...)
```

The '...' indicates that you can specify the arguments to your system call after the number argument. You can use the thread runner program (described in the next section) with `--quantum=`*`arg`* to invoke your new system call with the argument you provide.1 On running this program, you should see the printk message you placed in your system call implementation printed out to the kernel logs. You can view the kernel logs by examining the serial log file from KVM. This file is named serial.out if you invoked the KVM with `-serial file:./serial.out`. Alternatively, you can view these logs by typing `dmesg` from within your KVM terminal:

```
mjantz@kvm: $ dmesg | tail
```

Observing scheduling behavior

We provide the thread runner program to test the Linux scheduler:

```
mjantz@kvm: $ ./thread runner
Usage: thread runner [OPTIONS] num threads buffer size
Try 'thread runner --help' for more information.
```

This program creates the specified number of threads and allocates a global buffer of the specified size. A global pointer is initialized to point to the start of the global buffer, and each thread is assigned a unique character. Each thread continuously writes its character to the current position in the global buffer and increments the global position pointer until it has written its share of characters into the buffer. The number of characters each thread writes depends on the size of the global buffer, but each thread always writes the same number of characters as each other thread. Thus, at the end of the run, the characters written in the buffer record the execution pattern for each thread.

The program prints this global buffer at the end of each run. However, to reduce the size of the output, it first applies a simple post-processing pass to aggregate multiple consecutive characters of the same type to one character that represents a longer execution interval. This program uses the completely fair scheduler by default. It includes options for setting thread priorities with this scheduling policy. You should try different thread priorities to see how priority affects scheduling behavior.

You should use this same program to test your implementation of the round robin scheduler and custom system call. Try a few different quantum values (including 0 for FCFS) with different thread combinations to test your scheduler implementation.

SUBMISSION

You will demonstrate the use of your round robin scheduler (using the thread runner program).

You should also submit a patch file so we can examine all of the kernel modifications you have made. Follow the instructions on the TA's website to generate your patch file. The patch file you submit should match the built code you demonstrate for the TA. If we notice any discrepancies (for instance, the demonstration went fine, but there are obvious errors in your patch file), we will test your patch file on our own KVM and use that test to grade your project.

The report should (briefly) describe each of the stub functions you modified and their role in implementing the round robin scheduler. Also, please discuss how you tested the scheduler and any difficulties you had in implementing the project.

GRADING POLICY

The point breakdown for grading is shown below:

- Modified sched setscheduler to select round robin scheduler (10)
- System call to set round robin quantum (20)
- Round robin scheduler implementation (40)
- FCFS semantics with quantum=0 (10)
- Report (20)