Contents

Group 9 - Enhancing and Experimenting with the Linux Scheduler

Modifications to the Linux Scheduler

This is an overview of the work done by Group 9 in order to modify the Linux scheduler.

1. Implementing a New Scheduling Algorithm

We decided to implement a new scheduling algorithm based on the lottery scheduling policy. Its behaviour is simple: when a process is created, it is assigned a number of tickets, which are used to determine the process's chance of being scheduled. The scheduler randomly picks a ticket from the available tickets and runs the process that holds that ticket. Of course, the more tickets a process has, the higher its chance of being scheduled.

To implement a new scheduling algorithm we had to modify the Linux kernel source code. The main steps were:

1. Define a new scheduling class Inside the folder 'linux/kernel/sched' we created a new file called 'lottery.c'. In this file, we defined a new scheduling class called 'lottery' via the DEFINE_SCHED_CLASS macro, like this:

```
DEFINE_SCHED_CLASS(lottery) = {
    .enqueue_task = enqueue_task_lottery,
    .dequeue_task = dequeue_task_lottery,
    // And so on ...
};
```

This macro is used to define a new scheduling class and its associated functions. Basically the macro is useful to register the new scheduling class of type 'struct sched_class' in the kernel.

2. Tweak the linker file We then had to modify the linker file 'linux/include/asm-generic/vmlinux.lds.h' in order to include our new scheduling class. This is necessary because the scheduler will traverse the various scheduling classes not via an ordinary linked list (so via pointer), but simply by traversing the contiguous memory space where the scheduling classes are stored, knowing where the first one and the last one are located. To do this, we added the following line to the linker file:

```
/*
 * The order of the sched class addresses are important, as they are
 * used to determine the order of the priority of each sched class in
 * relation to each other.
 */
#define SCHED_DATA              \
    STRUCT_ALIGN();             \
    __begin_sched_classes = .;      \
    *(__idle_sched_class)           \
    *(__fair_sched_class)           \
    *(__rt_sched_class)         \
    *(__dl_sched_class)         \
    *(__stop_sched_class)           \
    *(__lottery_sched_class)        \
    __end_sched_classes = .;
```

Also, we put the class at the end of the list, so that it has the lowest priority compared to the other scheduling classes.

3. Create the scheduling policy In order to create the a scheduling policy that the system would recognize, we had to modify the file 'linux/include/uapi/linux/sched.h'. In this file, we added a new scheduling policy called 'SCHED_LOTTERY' after the existing scheduling policies, like so:

```
/*
 * Scheduling policies
 */
#define SCHED_NORMAL        0
#define SCHED_FIFO      1
#define SCHED_RR        2
#define SCHED_BATCH     3
/* SCHED_ISO: reserved but not implemented yet */
#define SCHED_IDLE      5
#define SCHED_DEADLINE      6
/* New scheduling policy for lottery scheduling */
#define SCHED_LOTTERY       7
```

4. Initialize the data structures Inside 'linux/kernel/sched/core.c' we modified the function *sched_init* that is called during the kernel initialization to initialize the runqueue for our new scheduling class. We created the function *init_ltr_rq* that we defined in 'lottery.c'.

The lottery runqueue is a data structure that keeps track of the tasks that are currently in the lottery scheduling class. It keeps track of:

- The total number of tickets available in the runqueue.
- The number of tasks in the runqueue.
- A doubly linked list of tasks in the runqueue. This kind of data structure was implemented to allow for efficient insertion and deletion of tasks in the runqueue by the linux kernel developer, also in order to avoid having too many implementations of the same data structure in the kernel.

5. Modify the task_struct The task_struct is the data structure that represents a task in the Linux kernel. We had to modify this kind of data structure to include in it a field of type *sched_ltr_entity*, a section that would hold the number of tickets acquired by a process, and in order to do so we modified the file 'linux/include/linux/sched.h'.

6. Implement the scheduling functions Then it was time to implement the scheduling functions for our new scheduling class.

The main function that make up a scheduling class are:

- enqueue_task_lottery: This function is called when a task is added to the run queue. It adds the task to the lottery queue and updates the total number of tickets.
- dequeue_task_lottery: This function is called when a task is removed from the run queue. It removes the task from the lottery queue and updates the total number of tickets.
- yield_task_lottery: This function is called when a task voluntarily yields the CPU. It updates the task's state and may trigger a reschedule based on the lottery tickets.
- check_preempt_curr_lottery: This function is called to check if the current task should be preempted by a higher priority task. It compares the lottery tickets of the current task and the new task and decides whether to preempt or not.
- pick_next_task_lottery: This function is called to select the next task to run. It randomly selects a ticket from the available tickets and returns the task associated with that ticket.
- set_next_task_lottery: This function is called to set the next task to run. It updates the current task pointer to the selected task.
- task_tick_lottery: This function is called when a task is running and its time slice expires. It checks if the task should be rescheduled based on the lottery tickets and updates the task's state accordingly.

7. Modify the Makefile Finally, we had to modify the Makefile in the 'linux/kernel/sched' directory to include our new scheduling class. We added the following line:

```
obj-y += lottery.o
```

2. Scripting

When dealing with such big projects, it is important to have a good scripting system to automate the process of building and testing the kernel. We created two scripts to help us with this task:

1. 'compile.sh': to compile the kernel and the header files. Obtaining the headers was essential for the next step: compiling the user-space programs used to test our new scheduling class, as in the minimal filesystem provided by busybox there is no compiler available.
2. 'qemu_launch.sh': This shell script initializes a QEMU ARM virtual machine by locating required files (zImage, versatile-pb.dtb, and rootfs.cpio.gz) either from a previously saved configuration or through a filesystem search. If the –refresh flag is provided, it forces a re-search of the files. When multiple matching files are found, the user is prompted to select one. Once identified, the file paths are cached in ~/.qemu_config.sh for future runs. After confirming, the script launches QEMU using the selected files with predefined parameters.

Benchmarking the Scheduler

The benchmarking phase of the project aimed to monitor and collect the waiting times of selected processes, for which both the start and finish times were known. This phase was structured into four main steps:

1. **Program Generation for Scheduling Analysis**
   We developed a simple yet computationally intensive C program, `dummy_program.c`, which calculates the value of π using the Leibniz series. This ensured enough CPU load to effectively test the scheduler's behavior.

2. **Concurrent Execution Under Different Scheduling Policies**
   To evaluate how the scheduler manages multiple processes, we executed several instances of

`dummy_program.c` concurrently using another C program, `dummy_launcher.c`. This launcher modified the scheduling policy before execution, switching from the default to our custom `SCHED_LOTTERY` policy.

3. **Logging Scheduling Decisions**
   Each time a process was selected by the scheduler, the kernel logged a message containing the PID, the timestamp of selection, and the number of tickets assigned to that process.

4. **Data Extraction and Visualization**
   Those data were extracted and stored in a file. The results were then plotted using the Python script `monitor.py`, allowing us to analyze scheduling patterns and waiting times visually.

Performance Metrics

For the analysis of the results we used the Gantt charts previously obtained during the benchmarking phase. The results were obtained in two separate conditions regarding the tickets assigned for each process inside the file `linux/kernel/fork.c`. - With a fixed number of tickets per process (5):

```
p->ltr.tickets = 5;
```

- With a variable number of tickets per process, assigned randomically (1 to 15):

```
unsigned int random_n = 0;
get_random_bytes(&random_n, sizeof(random_n));
random_n = random_n % 14 + 1; /* 1 to 15 */
p->ltr.tickets = random_n;
```

The results of the benchmarking in the case of 4 concurrent processes can be found in figures 1 and 2.

1) In the first case, the waiting times of the processes are totally random and based only on the ticket extracted by the scheduler. So we can expect, at every execution, a different behavior and it's impossible to estimate which process will finish before the others. As it can be seen, there is not a huge difference in the end times of the programs and their executions are almost uniformly distributed.

2) Instead in the second case, where each process is assigned a random amount of tickets, the waiting times reflects this first allocation: as expected initially the CPU is assigned almost only to processes with PIDs 42 and 43, that have way more tickets with respect to the others, that are executed only sporadically until the the end of first two; then they alternate. Here we would have expected the process with PID 44 to finish its execution way before the other one due to the fact that it has twice the tickets, but because of the randomic behavior of the scheduler in this case we can't see a big difference.

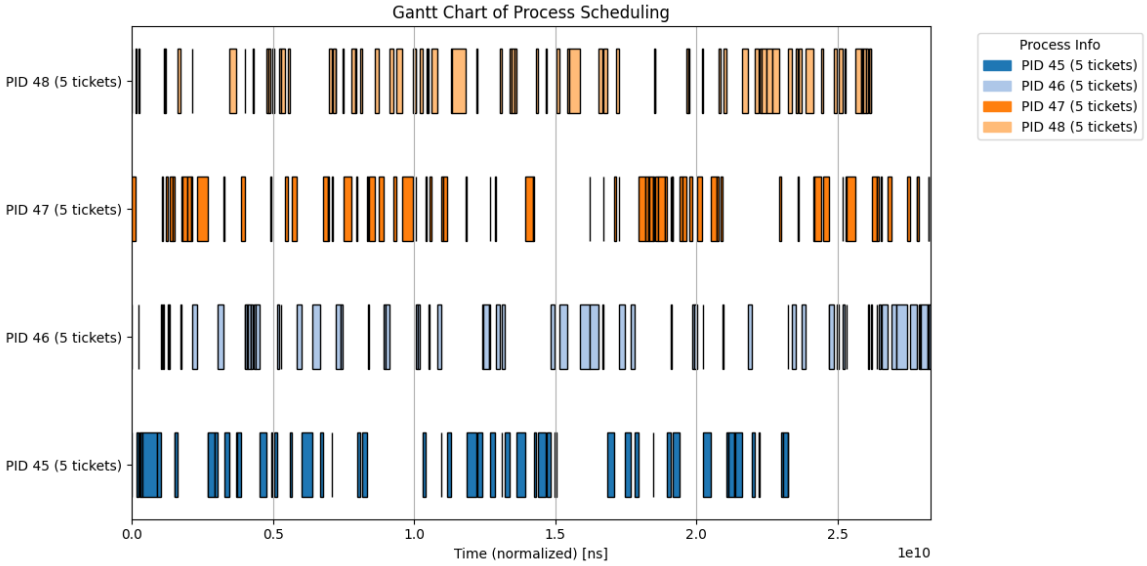

Figure 1: Gantt Chart fixed amount of tickets

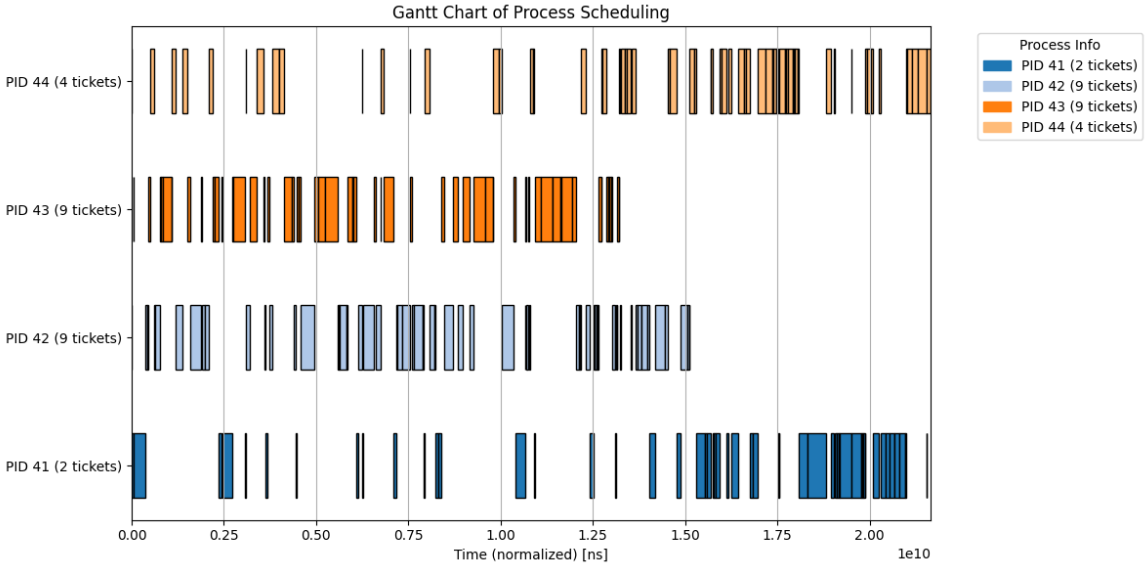

Figure 2: Gantt Chart random number of tickets