OS Project 2 Report

In Search of an Efficient Linux Process Scheduler

Zenan Li 519021911033

Emiyali@sjtu.edu.cn

Date: June 4, 2021

Abstract

In this project, we first implement a trivial **WRR Scheduler** that will assign different time slice for tasks in foreground and background. Secondly, we introduce **Group Scheduling** into the scheduler to improve its robustness. Thirdly, we implement a WRR Scheduler based on **Random Multilevel Feedback Queue** and conduct comparisons to test the schedulers' efficiency. The experiments results prove that our scheduler achieves **high throughput** and **low latency**.

Keywords: Linux Scheduler, Weighted Round Robin, Random Multilvel Feedback Queue

Contents

1 Introduction

A process is an instance of a computer program that is being executed, and a thread is also the programmed code in execution. The thread is the smallest unit that can be managed independently by a kernel scheduler. In modern computer systems, there may be many threads waiting to be served at the same time. Thus, one of the most important jobs of the kernel is to decide which thread to run for how long. The part of the kernel in charge of this business is called the **scheduler**.

On a single processor system (Our main topic in this paper), the scheduler alternates different threads in a time-division manner, which may lead to the illusion of multiple threads running concurrently. On a multi-processor system, the scheduler assigns a thread at each processor so that the threads can truly run in parallel.

Most of scheduling algorithms are priority-based. A thread is assigned a priority according to its importance. The general idea On Linux is that threads with a higher priority run before those with a lower priority, whereas threads with the same priority are scheduled in a round-robin (RR) fashion.

To deal with different kinds of scheduling situations, Linux has developed corresponding kinds of schedulers. For example, the **CFS Scheduler** for normal processes, the **RT Scheduler** for

real-time processes. However, their different implementations could lead to different scheduling patterns like **Preemption**, hence, result in diverse performance.

In this paper, we implement a basic Weighted Round Robin (**WRR**) Scheduler (which assigns different time slices to processes in foreground and background). And be enlightened by the idea of Multilevel Feedback Queue (MLFQ in short) and Markov Scheduler, we revise it to achieve a scheduler with **high throughput** and **low latency**.

2 Motivation

In this section, we'll talk about the pros and cons of the current Linux RT Scheduler, along with the MLFQ and Markov Scheduler that motivate us to develop a new scheduler.

2.1 Current Linux RT Scheduler

First of all, it's really a leap of progress for Linux to develop a clear structure of Scheduler, that is: The **Scheduling Classes**.

For a typical Linux Scheduler, it would have the following structure:

```
struct sched_class {
  const struct sched_class *next;
  ...
  void (*enqueue_task) (struct rq *rq, struct task_struct *p, int flags);
  void (*dequeue_task) (struct rq *rq, struct task_struct *p, int flags);
  ...
  struct task_struct * (*pick_next_task) (struct rq *rq, struct
     task_struct *prev);
  void (*put_prev_task) (struct rq *rq, struct task_struct *p);
  ...
  void (*task_tick) (struct rq *rq, struct task_struct *p, int queued);
  ...
};
```

Typically, we need and only need to implement the functions above (to enqueue or dequeue a new process, select next process and so on), so that we could develop our own Linux Scheduler.

Currently, the RT Scheduler supports the following two scheduling policies:

1. `SCHED_RR`: Threads of this type run one by one for a pre-defined time interval in their turn (round robin).

2. `SCHED_FIFO`: Threads of this type run until done once selected (first-in/first-out).

Both two scheduling policies will run processes with higher priority in front of those with lower priority (pay attention that the RR policy also wouldn't select a low-priority process unless all the high-priority processes have been completed). However, it could raise problems like **starvation** along with **high latency**. With that in mind, we should consider a way to assign priorities reasonably, or to dynamically update the processes' priorities. Here we choose the latter approach to improve the scheduler's performance.

2.2 Multilevel Feed Back Queue (MLFQ)

The multilevel feedback queue scheduling algorithm allows a process to move between queues. The idea is to separate processes according to the characteristics of their CPU bursts. If a process uses too much CPU time, it will be moved to a lower-priority queue (with larger time slice). This scheme leaves I/O-bound interactive processes (usually with short CPU bursts) in the high-priority queues. In addition, a process waits too long in a lower-priority queue may be moved to a higher-priority queue. This form of aging prevents starvation. Its main structure is shown in Figure 1.

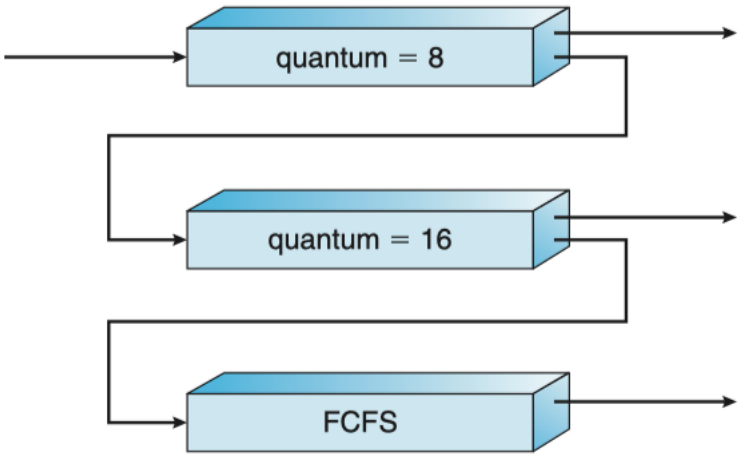

Figure 1: Multilevel feedback queues [1]

Take the idea from MLFQ, we could dynamically change a process's priority whenever it uses up its time slice, which is a reasonable way to achieve less **context switchings**, hence, low latency and high throughputs.

2.3 Markov Scheduler

In situations of transaction scheduling, transactions are blocked or allowed to run depending on some scheduling policy. For every specific model, there may exist an optimal concurrency level to achieve the highest throughput. And the Markov-Chain Based Scheduler, is a prediction model to find a system's optimal concurrency level [2]. It models the system behavior through a set of states and states transitions (each with a transition probability), as shown in Figure 2.

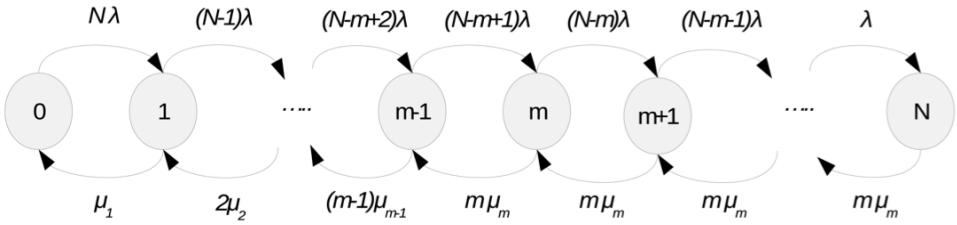

Figure 2: Markov Chain [2]

The paper [2] uses the model above to predict a system's throughput under different levels of concurrency. Here we skip the details of mathematical deduction and simply take the **randomizing** idea, could we set reasonable transition probability between each priority queue so that the process's priority could change in a more rational pattern?

3 Design

In this section, we'll unfold our design of the **WRR Scheduler** in a step-by-step approach.

3.1 Trivial WRR Scheduler

The trivial WRR Scheduler achieves the project's baseline. As we've all known, the Round-robin scheduling treats all tasks equally, but there are times when it is desirable to give some tasks preference over others. On that consideration, the tasks in Android are classified into different groups so that the various task groups (different from "group" that we'll discuss in the next subsection) could be handled appropriately.

Typically, Android tasks are assigned to the foreground and background group. And our trivial WRR Scheduler would assign more milliseconds as a time slice for foreground groups (Specifically, 100ms for fore and 10ms for back).

3.2 Group WRR Scheduler

Group Scheduling is a significant concept in Linux Scheduling. Consider a situation as following: If 50 processes are trying to run at any given time, the CFS Scheduler will carefully ensure that each gets 2% of the CPU. However, it could be that one of those processes is a server belonging to Alice, while the other 49 are part of a massive kernel build launched by Bob to preempt the portion of CPU time. Clearly, we couldn't say that we've achieve balance between Alice and Bob. One reasonable idea is to say that Bob's 49 compiler processes should, as a group, share the processor with Alice's server. In other words, the server should get 50% percent of the CPU while all of Bob's processes share the other 50%.

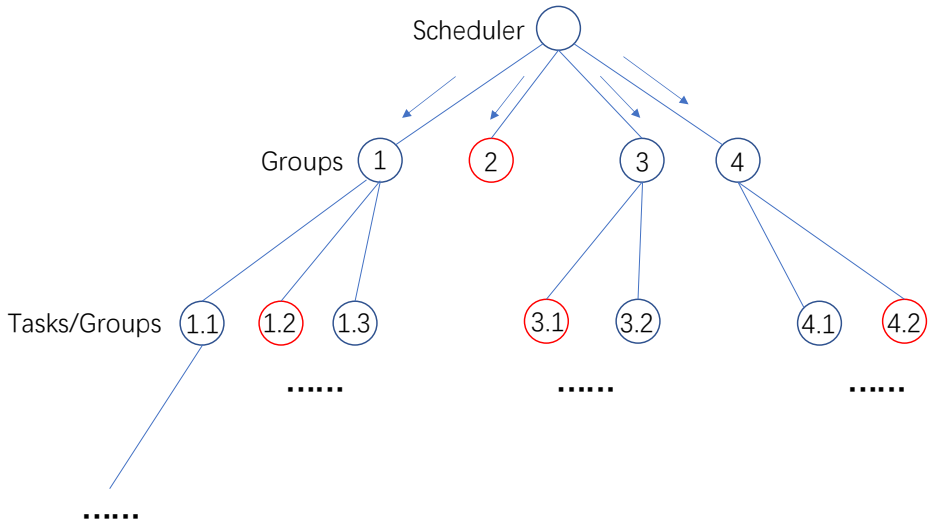

Figure 3: Group Scheduling

To achieve effective group scheduling, we could maintain a process tree structure as shown in Figure 3. For each process in the tree, we'll use a doubly linked list to store its parent and child processes. And each time when the scheduler are looking for the next process to be run, it will use **DFS** to find a **leaf node** and launch it based on the specific scheduling strategy (for example, in Figure 3, we will run processes 1.2, 2, 3.1, 4.2 in a fair way). Besides, whenever a new process is pushed into the waiting queue or an old process is popped out, we will need to chase down or go up the chain to update the whole group's priority as well as time slice.

3.3 WRR Based on Random MLFQ

Combining the characteristics of MLFQ and Markov Scheduler, we come up with a new approach to improve the WRR Scheduler's performance, that is: the Random MLFQ (**RMLFQ** in short). The main structure of our design is shown in Figure 4.

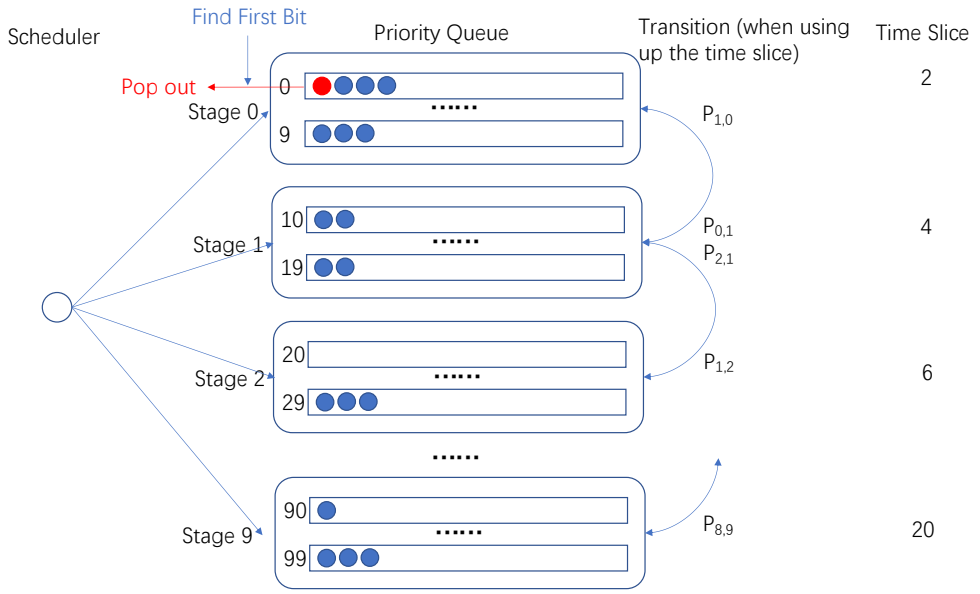

Figure 4: Random Multilevel Feedback Queues

Similar to the RT Scheduler, we set the priority range of WRR scheduling to be 0~99 (we say the priority to be higher when the number is smaller in this paper). One significant shortcoming of the WRR scheduler is that it assign all foreground processes with a same time slice. To compensate for this, we divide the 100 priorities into 10 stages (0 to 9, 10 to 19 ... 90 to 99). A lower stage, which has a higher priority, is assigned a smaller time slice. Specifically, the ten stages' time slices are a arithmetic progression.

Take the idea from the RT Scheduler, we will maintain a bitmap with 100 bits during scheduling. And for each bit in the bit map, a running queue of processes is assigned to it. When a new process is inserted into an empty queue or the last process is removed from a running queue, we will correspondingly set the bitmap to be 1 or 0. In this pattern, each time when we are trying to pick up the next process, we could find the first valid bit of the bitmap and pop out the first process in its corresponding running queue, which could be done in $O(1)$ time.

A step further, what should we do when a process running out of its time slice? In the traditional RT Scheduler, it simply inserts the process back to the tail of the previous running queue. However, the MLFQ tells us that when a process runs up its time slice, it should has more

probability to be a CPU-bound task, and we could assign a larger time slice to it. In this way, we could reduce the times of context switching and improve the system's throughput. Similarly, if an important interactive process (which needs low latency) is originally put at a high stage, we could also raise its priority to achieve lower latency when it uses up its time slice.

Under most situations, the MLFQ could perform well since the processes usually have distinctive characteristics of I/O bound or CPU bound. However, the throughput and latency could be conflicted with each other since a lager time slice means a lower priority, and that is, a higher latency. Thanks to the Markov Scheduler, we could assign probabilities for going up or down the stages ($p_{low,high}$, $p_{high,low}$ and p_{stay}). But a simple rough application of probability could only result in a higher overload of condition judgments. So we should consider how to set those probabilities reasonably. First, because of probability normalization, we have:

$$p_{low,high} + p_{high,low} + p_{stay} = 1$$

Second, the symmetry tells us that, when a priority is already in a relative low stage, it should have a higher probability of jumping to a higher stage than a lower stage. Besides, if a process keeps running out of its time slice in the current stage, it should has a bigger probability to jump to another more fittable stage. For these two reasons, we additionally store the stage (different from "stages" in Figure 4, here we only set 5 stages from 0 to 4) and times of staying in the current stage for each process.

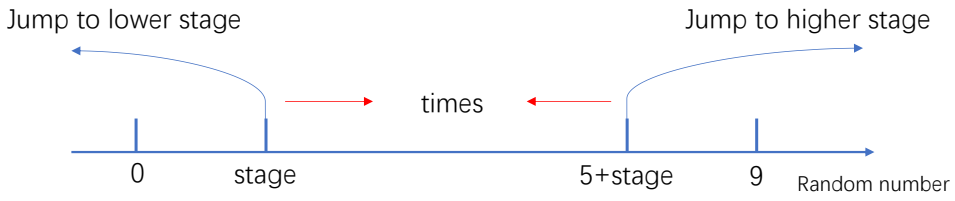

Figure 5: Jump Strategy

So as shown in Figure 5, we initialize $times = 1$, and each time a process has run out of its time slice, we will generate a random number $0 \leq r \leq 9$. Then we'll make the transition decision based on the following equation:

$$r \begin{cases} < stage + times, & \text{Jump to lower stage}, times = 1 \\ > 5 + stage - times, & \text{Jump to higher stage}, times = 1 \\ otherwise, & \text{Stay in current stage}, times++ \end{cases}$$

It could be clearly seen from Figure 5 that a process in a higher stage will have more probability to jump to a lower stage, while a lager $times$ means more probability of transition. We'll implement

this improved WRR Scheduler in the next section and show its excellent performance in the **Benchmark** section.

4 Implementation

In this section, we will briefly summarize our work in the Linux Kernel.

4.1 Define Structures

The basic structure of our WRR Scheduler is shown as following:

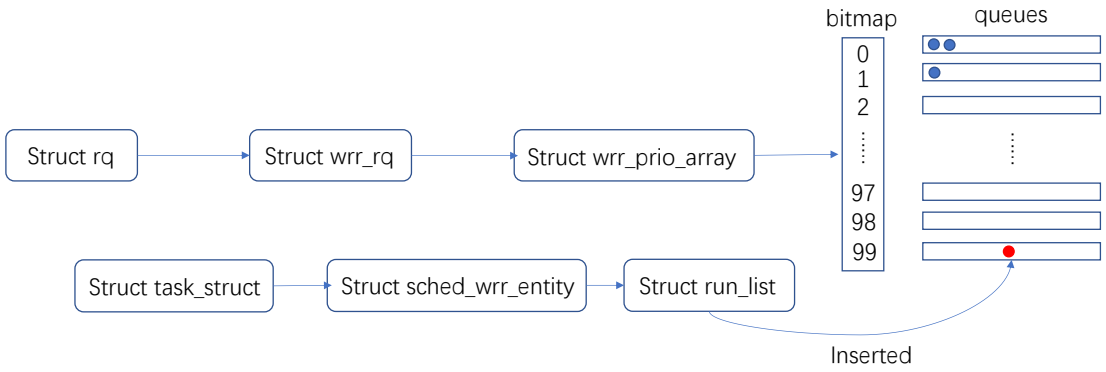

Figure 6: WRR Scheduler Structure

In a nutshell, the running queue (`rq`) is a per CPU struct used to manage the processes waiting to run on that CPU. A step further, the `rq` has child structs like `cfs_rq`, `rt_rq`, and here, we add the `wrr_rq` to the `rq` struct. The `wrr_rq` is the main struct we use to manage the processes running under WRR Scheduler, as shown in Figure 6, we define it as following:

```
struct wrr_rq {
  struct wrr_prio_array active; // The priority queue for scheduling
  unsigned long wrr_nr_running; // The running wrr processes
  int curr; //highest queued wrr task prio
  // For group scheduling
  struct rq * rq;
  struct list_head leaf_wrr_rq_list;
  struct task_group *tg;
};
```

Here the child struct `wrr_prio_array` is used for quicker scheduling, imitating the RT Scheduler, we define it as following:

```
struct wrr_prio_array {
  DECLARE_BITMAP(bitmap, MAX_WRR_PRIO+1);
  struct list_head queue[MAX_WRR_PRIO]; // Define priority queues
};
```

We declare a 100-bits bitmap, each corresponding to a process queue. When the i-th bit of the bitmap is 0, it means there doesn't exit a process of priority i and vice versa.

The `task_struct` is a per process struct used to store the information of the process. Here we add the scheduling entity `sched_wrr_entity` to it for further scheduling, which is defined as following:

```
struct sched_wrr_entity {
  struct list_head run_list; // insert into the specific priority queue
  unsigned int time_slice; // the time slice left for this process
  unsigned long timeout;
  // For group scheduling
  struct sched_wrr_entity *back; // points to child process
  struct sched-wrr_entity *parent; // points to parent process
  struct wrr_rq *wrr_rq;
  struct wrr_rq *my_rq;
}
```

We insert the `run_list` into its corresponding priority queue (defined in `wrr_prio_array`) to note that it is waiting for running.

4.2 Implement Basic Wrr_Sched_Class

We should implement the necessary functions in `sched_class` to make our WRR Scheduler work. However, since the basic WRR Scheduler is almost the same as the RT Scheduler, here we only take some key implementation details into discussion.

1. **enqueue_task.** When we want to enqueue a new process into the running queue, we should first find its corresponding priority queue, then insert its `run_list` into it and set the corresponding bit to be 1.

   ```
   static void enqueue_wrr_entity(struct sched_wrr_entity *wrr_se, bool
       head)
   {
   ```

```
  struct wrr_rq *wrr_rq = wrr_rq_of_se(wrr_se);
  struct wrr_prio_array *array = &wrr_rq->active;
  struct list_head *queue = array->queue + wrr_se_prio(wrr_se);

  if (!wrr_rq->wrr_nr_running)
    list_add_leaf_wrr_rq(wrr_rq);
  // Add to the tail of the corresponding priority queue
  if (head)
    list_add(&wrr_se->run_list, queue);
  else
    list_add_tail(&wrr_se->run_list, queue);

  __set_bit(wrr_se_prio(wrr_se), array->bitmap);
  wrr_rq->wrr_nr_running++;
}
```

We use some predefined functions in this implementation, their effects are intuitive as their function names show.

2. **dequeue_task.** When we try to dequeue an old process, we should also find its corresponding priority queue, remove it, and if the queue becomes empty after that, we should set bit to be 0. We skip the code segment here.
3. **pick_next_task.** When looking for the next process to be launched, we should find the first valid bit in the bitmap, and run the first process in its corresponding priority queue.

```
static struct task_struct *pick_next_task_wrr(struct rq *rq)
{
  // Skip variables definitions similar to enqueue_task
  ...
  idx = sched_find_first_bit(array->bitmap);
  queue = array->queue + idx;
  next = list_entry(queue->next,struct sched_wrr_entity,run_list);
  struct task_struct *p;
  p = wrr_task_of(next);
  ...
  return p;
}
```

4. **get_interval.** It's the function to assign time slice for each process. Here we only distinguish the foreground and background tasks use the function **task_group_path()**.

```
static unsigned int get_rr_interval_wrr(struct rq *rq, struct
    task_struct *task)
{
  if (task == NULL)
    return -EINVAL;
  // Use task_group_path to distinguish the fore and back processes
  if (task_group_path(task->sched_task_group)[1] != 'b')
    return WRR_FORE_TIMESLICE; // 100ms
  else
    return WRR_BACK_TIMESLICE; // 10ms
}
```

5. **task_tick.** This function is periodically (every 10 ms) called by the main scheduler to reduce current process's time slice and trigger rescheduling if needed.

```
static void task_tick_wrr(struct rq *rq, struct task_struct *p, int
    queued)
{
  ...
  if (--p->wrr.time_slice) // Reduce the time slice
    return; // Return if there are time slice left
  // Reassign the time slice
  if (task_group_path(p->sched_task_group)[1] != 'b')
    p->wrr.time_slice = WRR_FORE_TIMESLICE;
  else
    p->wrr.time_slice = WRR_BACK_TIMESLICE;
  // Requeue to the end of queue if we (and all of our ancestors) are
      not the only element on the queue
  if (wrr_se->run_list.prev != wrr_se->run_list.next)
  {
    requeue_task_wrr(rq, p, 0);
    set_tsk_need_resched(p);
    return;
  }
}
```

4.3 Implement Group Wrr_Sched_Class

The key part of Group Scheduling is the holistic thinking. That is, the update of each process should rise to the point of a group. Here we use **enqueue_task** as an example functions to give an

overview of our implementation. First, we define an auxiliary function as following. It's used to traverse all of a certain process's ancestors.

```
#define for_each_sched_wrr_entity(wrr_se) \
  for (; wrr_se; wrr_se = wrr_se->parent)
```

Second, we should notice that the priority of an upper entry depends on its child processes, so whenever we are trying to insert a new process, all its parent processes should be requeued. So before the insertion, we should find all the ancestors of the process and dequeue them in a **top-down** manner (to avoid NULL pointer), update them, then enqueue them again. The code segment is shown as following:

```
static void enqueue_task_wrr(struct rq *rq, struct task_struct *p, int
   flags){
  struct sched_wrr_entity *wrr_se = &p->wrr;
  // Dequeue all its ancestors in a top-down manner
  struct sched_wrr_entity *back = NULL;
  for_each_sched_wrr_entity(wrr_se){
    wrr_se->back = back;
    back = wrr_se;
  }

  for (wrr_se = back; ; wrr_se = wrr_se->back){
    if (on_wrr_rq(wrr_se))
      __dequeue_wrr_entity(wrr_se); // skip definition here
  }

  // Enqueue all the processes again
  for_each_sched_wrr_entity(wrr_se){ // enqueue same as the former section
    ...
  }
  inc_nr_running(rq);
}
```

Based on the similar idea, we could implement the other Group Scheduling functions in the `sched_class`. Please refer to the source code for details.

4.4 Implement Random MLFQ

Without loss of generality, we implement it based on the basic WRR Scheduler. To implement the random MLFQ, we should first assign different time slice to processes with different priorities. As we've discussed in the **Design** Section, we divide the priorities into 10 stages and those processes with a higher priority is assigned a smaller time slice. So we should refine the **task_tick** and **get_interval** as following:

```
...
int stage = (wrr_task_prio(task)) / 10 + 1; // To get time slice

if (task_group_path(task->sched_task_group)[1] != 'b')
  // Assign time slice in arithmetic progression
    return WRR_FORE_TIMESLICE * stage;
else
    return WRR_BACK_TIMESLICE;
...
```

Next, let's look back on Figure 7. There are three things to be considered:

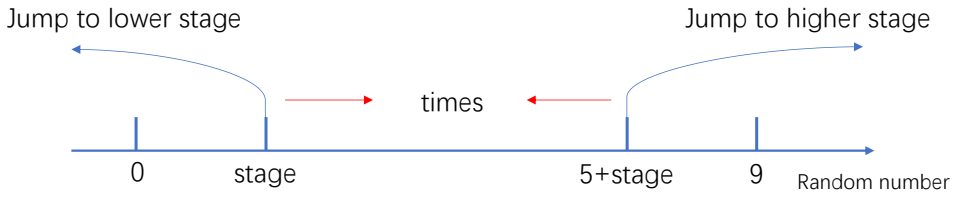

Figure 7: Jump Strategy

1. How to get $stage$? The answer is that the variable $stage$ could be computed based on priority in the function.
2. How to get $times$? We store additional variable $times$ in `task_struct`.
3. How to generate a random number in kernel mode? Fortunately, Linux provide a function **get_random_bytes()** for us to generate a random byte. And we could generate an unsigned integer as following:

```
static unsigned int getRand() {
  unsigned int randNum;
  get_random_bytes(&randNum,sizeof(unsigned int));
  return randNum;
}
```

The stage transition is triggered when a process has run out of its time slice, so we could implement the transition part in function **requeue_task** in `sched_class`. For simplicity, we just add or reduce the process's priority by 10 to achieve stage transition. The key part of stage transition is shown as following:

```
...
// Get random number
unsigned int r = getRand();

// Set high and low stage
struct list_head *high_queue = (prio < 90) ? array->queue+prio+10 : queue;
struct list_head *low_queue = (prio > 9) ? array->queue+prio-10 :queue;

if(r < ((p->times+prio/20) * UNSIGNED_MAX /10)){ // go to the low stage
    p->times = 1; // reset the times
    list_del_init(&wrr_se->run_list);
    list_add_tail(&wrr_se->run_list, low_queue);
  // if transitioned
    if(prio > 9){
        // update the bitmap
        if(list_empty(queue))
            __clear_bit(prio, array->bitmap);
        __set_bit(prio-10, array->bitmap);
        // reset priority
        p->rt_priority = p->rt_priority + 10;
    }
}
else if(r > ((5-p->times+prio/20) * UNSIGNED_MAX /10)){
  // go to high stage
    p->times = 1; // reset the times
    list_del_init(&wrr_se->run_list);
    list_add_tail(&wrr_se->run_list, high_queue);
  // if transitioned
    if(prio < 90){
        // update the bitmap
        if(list_empty(queue))
            __clear_bit(prio, array->bitmap);
        __set_bit(prio+10, array->bitmap);
        // reset priority
        p->rt_priority = p->rt_priority - 10;
```

```
    }
} else { // stay in the current stage
    p->times += 1; // accumulate the times
    list_move_tail(&wrr_se->run_list, queue);
}
```

4.5 Revise Other Functions

We should do something more in the kernel to assure that our scheduler could execute in a regular way (for example, add variables to `task_struct` or define extern variables in `core.c`). Here we give out some easy-ignored but significant points to be revised:

- We should set the RT Scheduler's next scheduler to be WRR Scheduler in `rt.c` to put our scheduler in use.
- We should add following initialization code to the function **__setscheduler** in `core.c` to make the system call **sched_setscheduler** run correctly:

```
if (p->policy == SCHED_WRR){
  p->sched_class = &wrr_sched_class;
  p->wrr.time_slice = p->sched_class->get_rr_interval(rq,p);
  p->times = 1;
}
```

- We should add following code to the function **sched_fork** in `core.c` to ensure our forked child processes are also scheduled by our WRR Scheduler (otherwise will be the CFS Scheduler):

```
if (p->policy == SCHED_WRR)
  p->sched_class = &wrr_sched_class;
```

There are many other modifications in the kernel and please refer to the source code for details.

5 Basic Results

In this section, we list our test results for the project's basic part.

We write a simple test file to set scheduler for `processtest.apk` and print out its basic information. The results are shown as following:

From Figure 8 and Figure 9 we could see that: After set the scheduler to be our WRR Scheduler, `processtest.apk` has a time slice of 100ms running in the foreground, while 10ms running in the background.

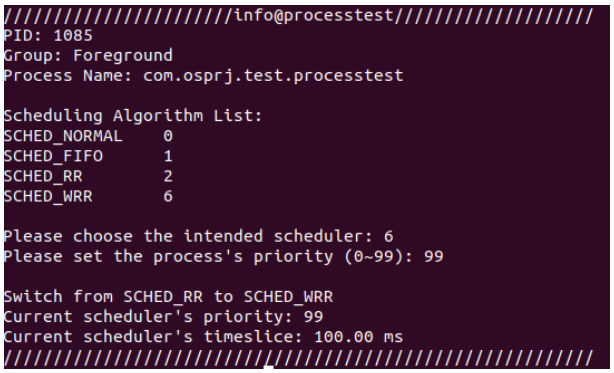

Figure 8: Foreground Test Result

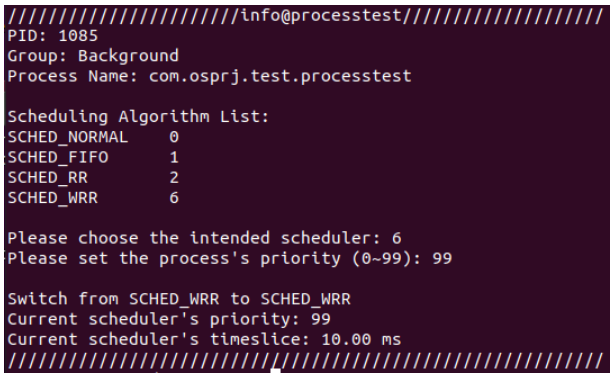

Figure 9: Background Test Result

And if we implement our WRR Scheduler based on the RMLFQ, we could get following output in the kernel:

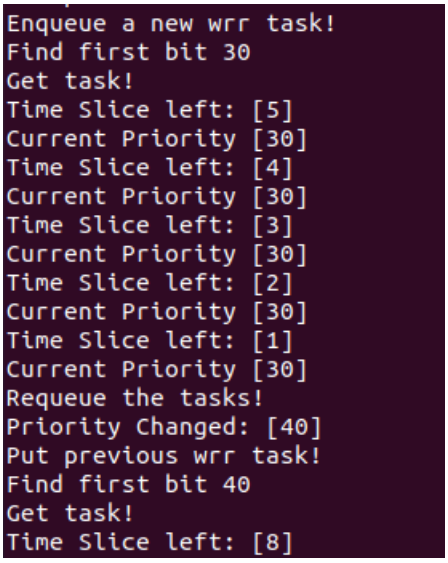

Figure 10: WRR Scheduler Running output

We could clearly see from Figure 10 that a new task has been enqueued. Besides, the timer ticks and the running queue is requeued, the current process transitions to a higher stage with a larger time slice.

6 Benchmark

In this section, we will conduct comparisons between different scheduling strategies. Specifically, we will test throughput and latency of `SCHED_FIFO`, `SCHED_RR` and `SCHED_WRR` (both basic and RMLFQ).

6.1 Train of Thought

The most significant thing we should consider in this section is that how could come up with reasonable approaches to measure the schedulers' throughput and latency.

- Firstly, to test our schedulers' robustness under different situations, we write three test programs: CPU-Bound, IO-Bound, CPU/IO Mixed, respectively. Specifically, the CPU-Bound program is a computation task which uses the Monte-Carlo method to compute e in 100,000 iterations; The I/O-Bound program is a repeatedly read and write task that will run 5,000,000 iterations; The CPU/IO mixed program is their simple combination.
- Secondly, to test schedulers' performance, we call **fork()** in the test programs to generate multiple running processes (child processes' priorities are set randomly) to be scheduled. Furthermore, we could compare schedulers' performance under different child process numbers.
- Thirdly, we use the equivalent concept: program running time (when running the same program), to measure the program's throughput. And the Linux Command `time -p` could help us to get the program running time conveniently.
- Finally, we use the accurate function **getdaytime()** to measure one process's latency. Specifically, we record the CPU time before **fork()** and the time that the child process first execute, and their difference is just the latency.

6.2 Test Throughput

We measure the program running time under CPU-Bound, IO-Bound, CPU/IO Mixed tasks and get the result charts as following. Please refer to the **Results** folder for more details of the numeric results.

We could clearly see from the charts that no matter the program is CPU-Bound, IO-Bound or Mixed, our Basic WRR Scheduler and RMLFQ WRR Scheduler often performs better than the FIFO and RR Scheduler. However, as the child process number growing, the Basic WRR Scheduler's throughput gets worse (nearly the same as RR), but the RMLFQ WRR Scheduler always performs well, beating the other schedulers by 10% to 20% optimization. The experiment results strongly validate our implementation's correctness.

It's intuitive that the Basic WRR Scheduler will have a performance nearly the same as the RR Scheduler, for that their implementation are just the same for the foreground tasks (The Basic WRR tends to be better because it doesn't take Group scheduling into consideration so its logic

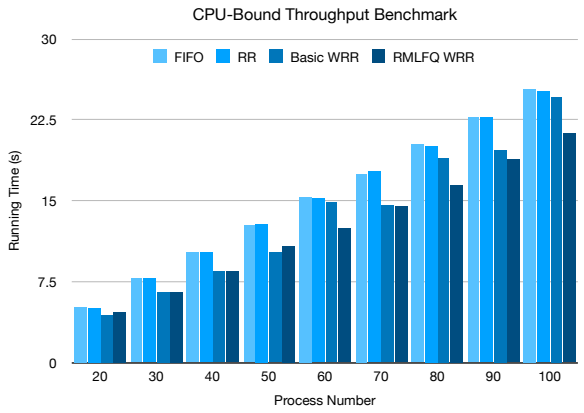

Figure 11: CPU-Bound Throughput Benchmark

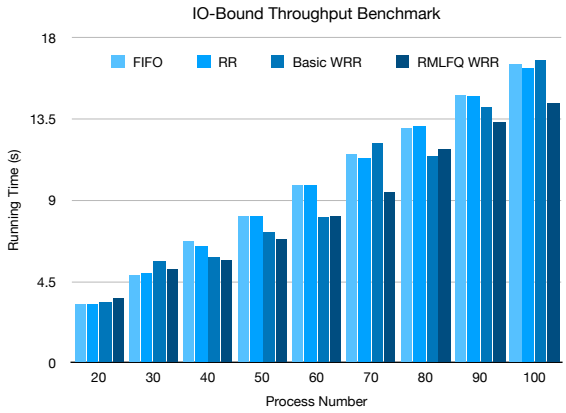

Figure 12: IO-Bound Throughput Benchmark

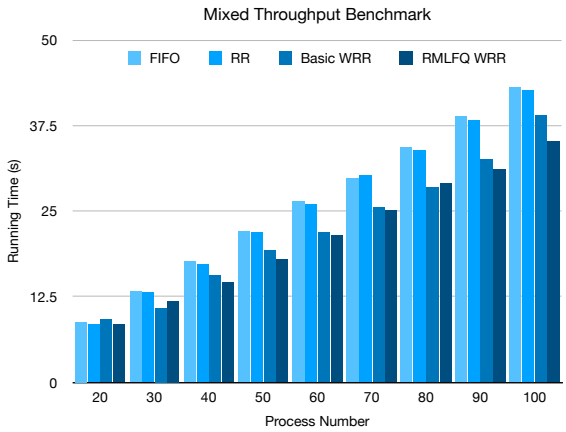

Figure 13: Mixed Throughput Benchmark

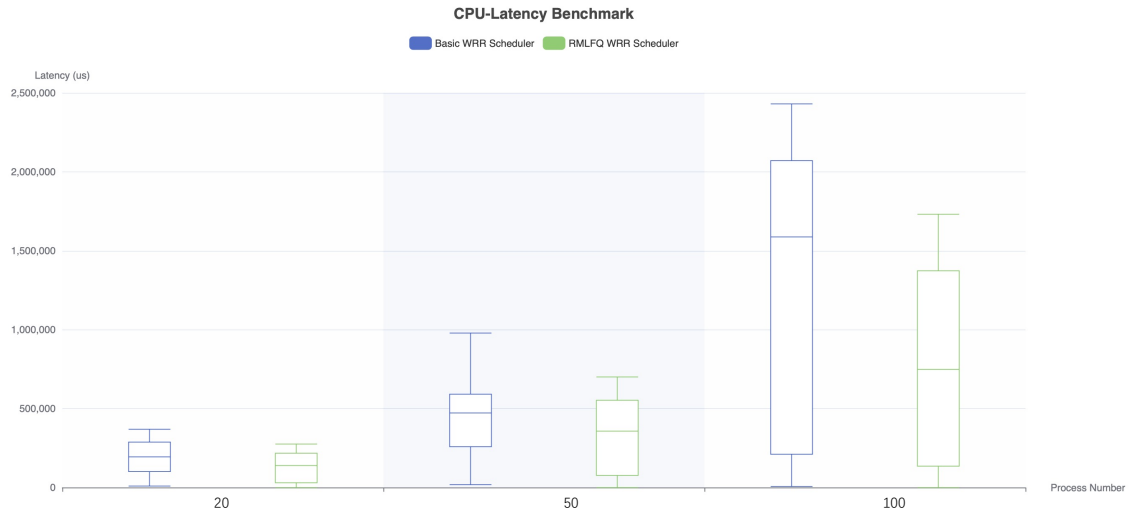

Figure 14: CPU-Bound Latency Benchmark

is simpler). But the RMLFQ WRR Scheduler provides a chance for stage transition and getting a bigger time slice, as we've discussed in the **Design** section, it should have a higher throughput (it could perform worse than the basic WRR when process number is small because of the overload of condition judgments).

6.3 Test Latency

We compute latency for each child process in the CPU-Bound program and draw the box chart as shown in Figure 14 (Please refer to the **Results** folder for more details of the numeric results). Specifically, we compare the latency between the Basic and RMLFQ WRR Scheduler under 20, 50 and 100 child processes. We could clearly see that the RMLFQ WRR Scheduler always have both lower average and highest latency than the basic one, especially when the process number is huge (the average latency is reduced by 40% when the process number is 100), corresponding with our discussion in the **Design** section.

7 Conclusion

In a nutshell, we've achieved following things in this project:

- We develop a trivial WRR Scheduler which assigns different time slice for tasks in the foreground and background.
- We implement **Group Scheduling** in our WRR Scheduler, hence it could be more robust and perform reasonably when there are multiple jobs in a group.
- Combine the idea of MLFQ and Markov Scheduler, we develop a WRR Scheduler based on **Random Multilevel Feedback Queue**, which assigns different time slice to processes in different stages. Moreover, we design a reasonable stage transition pattern for the process when it uses up its time slice, hence enabling our scheduler to get **higher throughput** and **lower latency**.
- We come up with reasonable approaches to test our schedulers' efficiency. As a result, the experiment results show that both basic and RMLFQ WRR Scheduler perform better than FIFO and RR. However, the basic WRR Scheduler's performance may become worse (nearly the same as RR) when process number gets larger while RMLFQ always achieve a 10% to 20% optimization in throughput and 30% optimization in latency.

8 Acknowledge

I spend most of time in this project reading the Linux Kernel Source Code, which is an interesting but tiring work. As Prof. Wu has said in class that, reading classical project's source code is very helpful to cultivate engineering thinking and I really believe so. After this project, I've understood the importance of annotation and learned more tricks of debugging, also, more experience of revising other people's codes. Moreover, I've read many papers of Linux Kernel and finally design my own WRR Scheduler based on RMLFQ. Although it's after all only some trivial work, but the improvement in performance really encourages me.

Thanks for Prof. Wu's excellent teaching in class, the knowledge I learned from the class lays the foundation for this project. Thanks to TAs, your detailed experiment guiding slides help me to build the environment smoothly. Finally, thanks to all my friends, our discussion is really inspiring and help me complete the work successfully.

References

[1] Avi Silberschatz, Peter Baer Galvin, Greg Gagne: Operating System Concepts. 10th edn. John Wiley & Sons, Inc (2018).

[2] Di Sanzo, Pierangelo & Sannicandro, Marco& Ciciani, Bruno & Quaglia, Francesco. (2016). Markov Chain-Based Adaptive Scheduling in Software Transactional Memory. 373-382. 10.1109/IPDPS.2016.104.

[3] Linux kernel scheduler, https://helix979.github.io/jkoo/post/os-scheduler/.

[4] Linux kernel hacking, https://scslab-intern.gitbooks.io/linux-kernel-hacking/content/.

[5] CFS group scheduling, https://lwn.net/Articles/240474/.