Project Report: Android Scheduler

519021911288 韩森宇

1. Project Overview

This project requires us to implement a Weighted Round Robin scheduler in Android kernel under the Linux development environment. In `work` folder, I have successfully implemented a simple runnable WRR scheduler in Android schedulers by modifying kernel files, adding the new source file `wrr.c` and recompiling the kernel. In `test` folder, a test program was written to change a process's scheduler by calling related system calls. Furthermore, in `bench` folder, I also wrote a benchmark program to evaluate the performances of different schedulers by calculating matrix multiplication.

Each of those three folders has a README file, indicating the structure of the directory and other useful information.

2. `work` – Files changed in Linux kernel

About 400 lines of changes have been made in kernel codes. Using Github's pull requests and commit merge, I made it much easier to check where was changed in the original kernel files. You can check this commit to learn the changes in the kernel after June 4th.

2.1. Work in other files

To put `wrr.c` into effect, kernel files related to the scheduler need to be modified.

2.1.1. `Makefile`

```diff
  # Default value for CROSS_COMPILE is not to prefix executables
  # Note: Some architectures assign CROSS_COMPILE in their arch/*/Makefile
  export KBUILD_BUILDHOST := $(SUBARCH)
- ARCH           ?=              $(SUBARCH)
- CROSS_COMPILE ?=
+ ARCH           ?= arm
+ CROSS_COMPILE ?= arm-linux-androideabi-
```

To compile the Android kernel in Linux environment, cross compile is needed.

2.1.2. `arch/arm/configs/goldfish_armv7_defconfig`

```diff
  CONFIG_CGROUP_CPUACCT=y
  CONFIG_RESOURCE_COUNTERS=y
  CONFIG_CGROUP_SCHED=y
+ CONFIG_WRR_GROUP_SCHED=y
  CONFIG_RT_GROUP_SCHED=y
  CONFIG_BLK_DEV_INITRD=y
  CONFIG_CC_OPTIMIZE_FOR_SIZE=y
```

Setting `CONFIG_WRR_GROUP_SCHED=y`, which is consistent with other schedulers.

2.1.3. `include/linux/sched.h`

```diff
 #define SCHED_BATCH            3
 /* SCHED_ISO: reserved but not implemented yet */
 #define SCHED_IDLE             5
+/* Modified: SCHED_WRR */
+#define SCHED_WRR              6
 /* Can be ORed in to make sure the process is reverted back to SCHED_NORMAL on fork */
 #define SCHED_RESET_ON_FORK     0x40000000

```

Define the value 6 to refers to WRR scheduler.

```diff

 struct seq_file;
 struct cfs_rq;
+/* Modified: struct wrr_rq declaration */
+struct wrr_rq;
 struct task_group;
 #ifdef CONFIG_SCHED_DEBUG
 extern void proc_sched_show_task(struct task_struct *p, struct seq_file *m);
```

Declare the run queue as `wrr_rq` in WRR scheduling class.

```diff
+/* Modified: wrr entity */
+struct sched_wrr_entity{
+	struct list_head run_list;
+	unsigned long timeout;
+	unsigned int time_slice;
+	int nr_cpus_allowed;
+
+	struct sched_wrr_entity *back;
+
+	struct sched_wrr_entity *parent;
+	struct wrr_rq *wrr_rq;  // This entity belongs to ...
+	struct wrr_rq *my_q;    // Its child entity belongs to ...
+};
+
```

Define a scheduling entity belongs to WRR scheduler. This part is imitating the `sched_rt_entity` above, but I deleted some unused parts.

```diff
 #define RR_TIMESLICE           (100 * HZ / 1000)

+/* Modified: WRR timeslice */
+#define WRR_FG_TIMESLICE       (100 * HZ / 1000)
+#define WRR_BG_TIMESLICE       (10 * HZ / 1000)
+
```

Define two time slices in WRR scheduling: `WRR_FG_TIMESLICE` for foreground tasks and `WRR_BG_TIMESLICE` for background tasks. I give foreground tasks 100ms per slice and background tasks 10ms per slice.

```diff
@@ -1280,6 +1302,8 @@ struct task_struct {
 	const struct sched_class *sched_class;
 	struct sched_entity se;
 	struct sched_rt_entity rt;
+	/* Modified: varaible added */
+	struct sched_wrr_entity wrr;
```

Define a WRR variable `sched_wrr_entity wrr` in the `task_struct`. `wrr` plays as the role of the linked node in the run queue, thus the scheduler can find the task struct that owns the `wrr`.

2.1.4. `kernel/sched/sched.h`

```diff
+/* Modified */
+static inline int wrr_policy(int policy)
+{
+	if (policy == SCHED_WRR)
+		return 1;
+	return 0;
+}
+
```

A function that checks whether the policy is WRR. (However I rarely use this function; I directly use the `if` inside when it needs to check the policy.)

```
+ /* Modified: struct wrr_rq declaration */
+ struct wrr_rq;
```

```
+ /* Modified: wrr runqueue */
+ struct wrr_rq{
+       struct list_head queue;         /* Only one active queue */
+       unsigned long wrr_nr_running;
+
+       int wrr_throttled;
+       u64 wrr_time;
+       u64 wrr_runtime;
+       raw_spinlock_t wrr_runtime_lock;
+
+       unsigned long wrr_nr_boosted;
+       struct rq *rq;
+       struct list_head leaf_wrr_rq_list;
+       struct task_group *tg;
+
+ };
+
```

Declaration & definition of WRR's run queue `wrr_rq`. This part is also imitating the `rt_rq` above, but different from RT scheduler, in `wrr_rq` there's only one active run queue (while `rt_rq` has 100 queues for 100 priorities). `wrr_nr_running` indicates the number of tasks in this queue. `*rq` points to the corresponding run queue (which contains other schedulers' run queues). `*tg` enables us to know whether the task is in foreground or background.

```
@@ -370,6 +401,8 @@ struct rq {

        struct cfs_rq cfs;
        struct rt_rq rt;
+       /* Modified: struct wrr_rq added */
+       struct wrr_rq wrr;
```

As is mentioned above, I add a `wrr_rq wrr` here in `rq`.

```
  #ifdef CONFIG_RT_GROUP_SCHED
        struct list_head leaf_rt_rq_list;
  #endif
+ /* Modified: list_head added */
+       struct list_head leaf_wrr_rq_list;
```

Add a list head. Since we have enabled `CONFIG_WRR_GROUP_SCHED`, the `#ifdef` part is omitted.

```
+ /* Modified: extern declaration (need more info) */
+ extern void free_wrr_sched_group(struct task_group *tg);
+ extern int alloc_wrr_sched_group(struct task_group *tg, struct task_group *parent);
+ extern char *task_group_path(struct task_group *tg);
```

```
  extern const struct sched_class rt_sched_class;
  extern const struct sched_class fair_sched_class;
  extern const struct sched_class idle_sched_class;
-
+ /* Modified: extern declaration */
+ extern const struct sched_class wrr_sched_class;
```

```
  extern void init_sched_rt_class(void);
  extern void init_sched_fair_class(void);
+ /* Modified: extern declaration (need more info) */
+ extern void init_sched_wrr_class(void);
```

```

  extern void init_cfs_rq(struct cfs_rq *cfs_rq);
  extern void init_rt_rq(struct rt_rq *rt_rq, struct rq *rq);
+ /* Modified: extern declaration (need more info) */
+ extern void init_wrr_rq(struct wrr_rq *wrr_rq);
+
```

Extern declarations. Most of them are defined in `wrr.c`, but `*task_group_path` is defined in `debug.c` to get the string of the task group, which indicates whether the task is in foreground or not.

2.1.5. `kernel/sched/core.c`

```
@@ -1723,6 +1724,8 @@ static void __sched_fork(struct task_struct *p)
  #endif

        INIT_LIST_HEAD(&p->rt.run_list);
+       /* Modified: wrr init */
+       INIT_LIST_HEAD(&p->wrr.run_list);
```

Initialize the list in WRR entity.

```diff
 static void
 __setscheduler(struct rq *rq, struct task_struct *p, int policy, int prio)
 {
+	// printk(">>> __setscheduler, 1\n");
+	/* Need revised! */
 	p->policy = policy;
 	p->rt_priority = prio;
 	p->normal_prio = normal_prio(p);
 	/* we are holding p->pi_lock already */
 	p->prio = rt_mutex_getprio(p);
-	if (rt_prio(p->prio))
-		p->sched_class = &rt_sched_class;
-	else
+	if(p->policy == SCHED_WRR){
+		// printk(">>> __setscheduler, wrr\n");
+		p->sched_class = &wrr_sched_class;
+	}
+	else if (rt_prio(p->prio)){	/* prio: 1~99 */
+		// printk(">>> __setscheduler, rt\n");
+		p->sched_class = &rt_sched_class;
+	}
+	else	/* prio: 100~139 */
 		p->sched_class = &fair_sched_class;
 	set_load_weight(p);
 }
```

Function `__setscheduler` is the key step to change a process's scheduling policy. It judges the scheduler by process's priority, but here WRR has the same priority range (1~99) with RT. So I let the function judge whether the task has WRR policy first, and set its scheduler to WRR directly if so.

```diff
@@ -4222,7 +4235,7 @@ static int __sched_setscheduler(struct task_struct *p, int policy,

 		if (policy != SCHED_FIFO && policy != SCHED_RR &&
 				policy != SCHED_NORMAL && policy != SCHED_BATCH &&
-				policy != SCHED_IDLE)
+				policy != SCHED_IDLE && policy != SCHED_WRR)
 			return -EINVAL;
 	}

```

Modified for the newly added WRR scheduler.

```diff
@@ -4235,7 +4248,8 @@ static int __sched_setscheduler(struct task_struct *p, int policy,
 	    (p->mm && param->sched_priority > MAX_USER_RT_PRIO-1) ||
 	    (!p->mm && param->sched_priority > MAX_RT_PRIO-1))
 		return -EINVAL;
-	if (rt_policy(policy) != (param->sched_priority != 0))
+	/* Modified: for prio 1-99 */
+	if ((rt_policy(policy) | wrr_policy(policy)) != (param->sched_priority != 0))
 		return -EINVAL;

```

Do not return `-EINVAL` if the policy is RT or WRR.

```diff
@@ -4895,6 +4920,8 @@ SYSCALL_DEFINE1(sched_get_priority_max, int, policy)
 	switch (policy) {
 	case SCHED_FIFO:
 	case SCHED_RR:
+	/* Modified here. We set wrr's prio range the same as rt's (1-99) */
+	case SCHED_WRR:
 		ret = MAX_USER_RT_PRIO-1;
 		break;
 	case SCHED_NORMAL:
@@ -4920,9 +4947,11 @@ SYSCALL_DEFINE1(sched_get_priority_min, int, policy)
 	switch (policy) {
 	case SCHED_FIFO:
 	case SCHED_RR:
+	/* Modified here */
+	case SCHED_WRR:
 		ret = 1;
 		break;
-	case SCHED_NORMAL:
+	        case SCHED_NORMAL:
 	case SCHED_BATCH:
 	case SCHED_IDLE:
 		ret = 0;
```

Here, I set WRR processes have the same priority range with RT (1~99).

```diff
@@ -7143,6 +7172,8 @@ void __init sched_init(void)
 		rq->calc_load_update = jiffies + LOAD_FREQ;
 		init_cfs_rq(&rq->cfs);
 		init_rt_rq(&rq->rt, rq);
+		/* Modified: function added */
+		init_wrr_rq(&rq->wrr);
```

Initialize the run queue of WRR.

```diff
@@ -7412,6 +7443,8 @@ static void free_sched_group(struct task_group *tg)
 {
 	free_fair_sched_group(tg);
 	free_rt_sched_group(tg);
+	/* Modified */
+	free_wrr_sched_group(tg);
 	autogroup_free(tg);
 	kfree(tg);
 }
```

```diff
@@ -7432,6 +7465,10 @@ struct task_group *sched_create_group(struct task_group *parent)
 	if (!alloc_rt_sched_group(tg, parent))
 		goto err;
 
+	/* Modified */
+	if (!alloc_wrr_sched_group(tg, parent))
+		goto err;
+
```

Be consistent with other scheduling classes.

2.1.6. `kernel/sched/rt.c`

```diff
 const struct sched_class rt_sched_class = {
-	.next			= &fair_sched_class,
+	/* Don't forget to link the node (this pains me a lot) */
+	.next			= &wrr_sched_class,
 	.enqueue_task		= enqueue_task_rt,
 	.dequeue_task		= dequeue_task_rt,
 	.yield_task		= yield_task_rt,
```

Scheduling classes are linked by their member `.next`. I add a `wrr_sched_class` between `rt_sched_class` and `fair_sched_class`, and to let classes be linked together again, `rt_sched_class.next` should point to `wrr_sched_class`. This is the same for `wrr_sched_class.next`.

2.1.7. `kernel/sched/Makefile`

```diff
- obj-y += core.o clock.o idle_task.o fair.o rt.o stop_task.o
+ obj-y += core.o clock.o idle_task.o fair.o rt.o stop_task.o wrr.o
```

Compile the `wrr.c` and make a corresponding object file.

2.2. Work in `wrr.c`

`wrr.c` is the major part of this project. Several scheduling functions required to be implemented to make the scheduler work normal.

The implement of `wrr.c` is referring to RT scheduler `rt.c`. I deleted some codes about multi-CPU, bandwidth and other unused parts, thus `wrr.c` implements a simplified scheduler here.

2.2.1. `wrr_sched_class`

```
const struct sched_class wrr_sched_class
```

This is the implementation of `struct sched_class` in WRR scheduler. It's member `.next` points to another scheduling class `fair_sched_class`, and it's other function members are implemented in `wrr.c`.

2.2.2. `enqueue_task_wrr`

```
static void
enqueue_task_wrr(struct rq *rq, struct task_struct *p, int flags)
```

Enqueue a task to the head or tail of the WRR run queue, depending on the flags. Data structure queue is implemented by list in Linux kernel. After enqueue, increase the running task number of WRR run queue by 1.

2.2.3. `dequeue_task_wrr`

```
static void
dequeue_task_wrr(struct rq *rq, struct task_struct *p, int flags)
```

Dequeue the task at the head of the WRR run queue. Before dequeue, update the task's runtime information. After dequeue, decrease the running task number of WRR run queue by 1.

2.2.4. `yield_task_wrr`

```
static void
yield_task_wrr(struct rq *rq)
```

Current WRR task voluntarily gives up its running and requeue to the tail of the run queue.

2.2.5. `check_preempt_curr_wrr`

```
static void
check_preempt_curr_wrr(struct rq *rq, struct task_struct *p, int flags)
```

Preempt the current task by a new task according to their priority, if necessary. If the new task has a higher priority (low `prio` means high priority), the function informs the scheduler to reschedule the current task.

2.2.6. `pick_next_task_wrr`

```
static struct task_struct *pick_next_task_wrr(struct rq *rq)
```

Scheduler will call this function when it is finding next runnable task. If the WRR run queue is not empty, get next WRR entity in the queue and find its owner `task_struct` by using `container_of`. Start the time counting of this task.

2.2.7. `put_prev_task_wrr`

```
static void
put_prev_task_wrr(struct rq *rq, struct task_struct *p)
```

This function will be called when a task is leaving CPU due to a new task. Similar to dequeue, update the task's time information and clear its time counting.

2.2.8. `set_curr_task_wrr`

```
static void
set_curr_task_wrr(struct rq *rq)
```

Scheduler calls this function when a task has changed its scheduler. Reset the task's time counting.

2.2.9. `task_tick_wrr`

```
static void
task_tick_wrr(struct rq *rq, struct task_struct *p, int queued)
```

Key function in time assigning and counting for WRR scheduling. Decrease the task's time slice, and if the time slice is zero (indicating that it has used up its time slice), re-assign it's time slice according to foreground or background, and move it to the tail of the run queue if it is not the only task in WRR run queue.

2.2.10. `get_rr_interval_wrr`

```
static unsigned int
get_rr_interval_wrr(struct rq *rq, struct task_struct *task)
```

This function is used for the system call `sched_rr_get_interval`. Return the task's time slice according to foreground or background.

2.2.11. `switched_to_wrr`

```
static void
switched_to_wrr(struct rq *rq, struct task_struct *p)
```

Scheduler calls this function when task's scheduler is switching to WRR.

Reschedule the current WRR task if the new task is in WRR run queue, is not the current task itself and has a higher priority than the current task.

2.2.12. update_curr_wrr

```
static void update_curr_wrr(struct rq *rq)
```

Update the time information of WRR run queue. Calculate the current task's total run time and delta run time, then update those time information.

2.2.13. Other functions

```
void init_wrr_rq(struct wrr_rq *wrr_rq)
```

Initialize the WRR run queue. Set running task number to zero.

```
static int wrr_is_foreground(struct task_struct *p)
```

Check whether the task is in foreground or not. The function calls `task_group_path` in `debug.c` and judges the string it returns: "`/`" for foreground and "`/bg_non_interactive`" for background.

```
void free_wrr_sched_group(struct task_group *tg) {}

int alloc_wrr_sched_group(struct task_group *tg, struct task_group *parent)
{
    return 1;
}
```

Consistent with other schedulers.

```
#ifdef CONFIG_SMP
...
#endif
```

Since `CONFIG_SMP` is not defined in configuration, functions inside can just be left dummy.

```
static void switched_from_wrr(struct rq *rq, struct task_struct *p) {}
static void
prio_changed_wrr(struct rq *rq, struct task_struct *p, int oldprio) {}
```

Functions not required to implement. Leave them dummy.

3. test – Scheduler-changing testfile

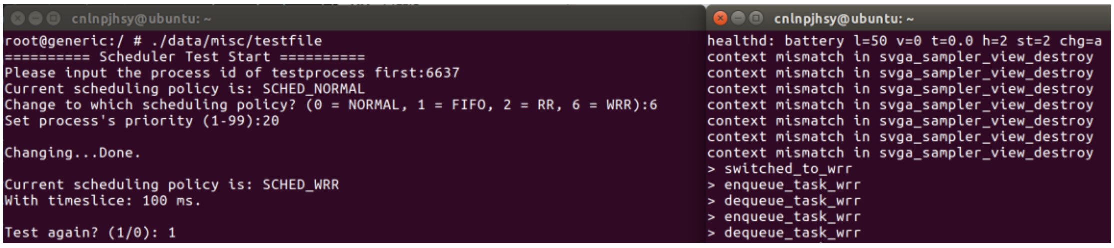
3.1. Work in testfile.c

Using system calls below, the program can get some task's information: current scheduler and it's time slice. The program can also change a task's scheduler and priority.

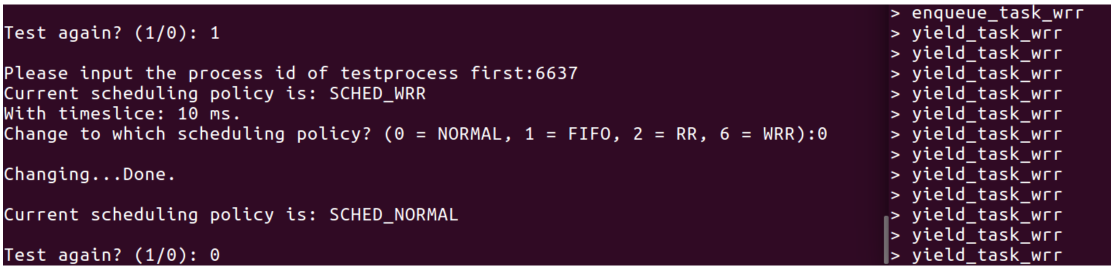
```
#156: sched_setscheduler(pid_t pid, int policy, struct sched_param* param)
#157: sched_getscheduler(pid_t pid)
#159: sched_get_priority_max(int policy)
#160: sched_get_priority_min(int policy)
#161: sched_rr_get_interval(pid_t pid, struct timespec* interval)
```

3.2. Test on WRR scheduling

Push the executable file to AVD and execute it in Android shell. Here, `processtest.apk` has a PID 6637. The screenshot below shows that the scheduler was successfully changed to WRR, and `printk` information began spamming the shell. `processtest` was assigned 100ms time slice because it was running in foreground.



Let AVD return to the main screen and `processtest` became a background process. `yield_task_wrr` was called when the home button was pressed. Now `processtest` was assigned 10ms time slice.



For more detailed test information, you can check `testscript.txt` under this directory.

4. bench – Scheduler performance benchmark

At the beginning, I intended to use the in-built performance analysis tool *perf*. Sad it was, troubles encountered when I tried to make perf and I had to give up. But fortunately, in one lab of CS359 Coumpter System Architecture, a test file was given to test the performance of CPU. I modified that file and it can give us a benchmark for different schedulers now.

4.1. Work in bench.c

`bench.c` tests a scheduler's performance by executing multi-process matrix multiplication. Type shell command

```
./benchmark #MATRIX_SIZE #CHILD_PROCESS_NUMBER
```

will start the performance test.

Let `#MATRIX_SIZE` $= n$ and `#CHILD_PROCESS_NUMER` $= m$, and the program will generate three $n \times n$ random matrixes A, B, C. The program tests `SCHED_NORMAL`, `SCHED_FIFO`, `SCHED_RR`, `SCHED_WRR` in turn. In each scheduler, program forks m child processes, and each of them will be set to the corresponding scheduler and calculate matrix multiplication $C = A \times B$ independently. Each scheduler has a timer, and the timer starts timing when the program begins forking, ends timing when all child processes have finished their calculation. Code was designed not to take in the time cost of I/O and conditional events, so the time here is just CPU computation time (and some forking time).

Naturally, timing result t (seconds) varies due to different values of n and m.

Therefore, I use **Mflop/s** (Million float operations per second) to evaluate scheduler's performance more generally. $n \times n$ matrix multiplication needs n^3 times of additions and n^3 times of multiplications. Take multiplications into count, and since I use double precision float numbers here, the formula to calculate Mflop/s is

$$\text{Mflop/s} = \frac{2n^3 \times m}{t}$$

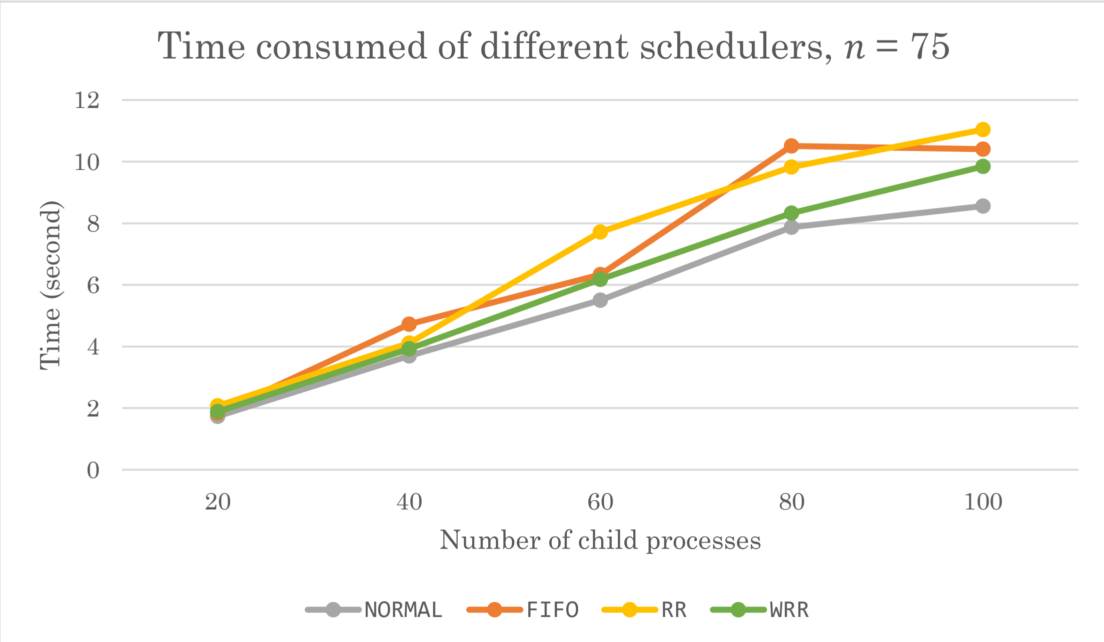
The program will give both time and Mflop/s for each scheduler as the result. By comparing different scheduler's Mflop/s, it will be more visually to evaluate how efficient the scheduler utilizes CPU.

4.2. Performance test result & analysis

Let $n = 75$, 100, 150 and $m = 20$, 40, 60, 80, 100. Total 15 tests were done. The original result is in `testscript.txt` under this directory.

4.2.1. $n = 75$

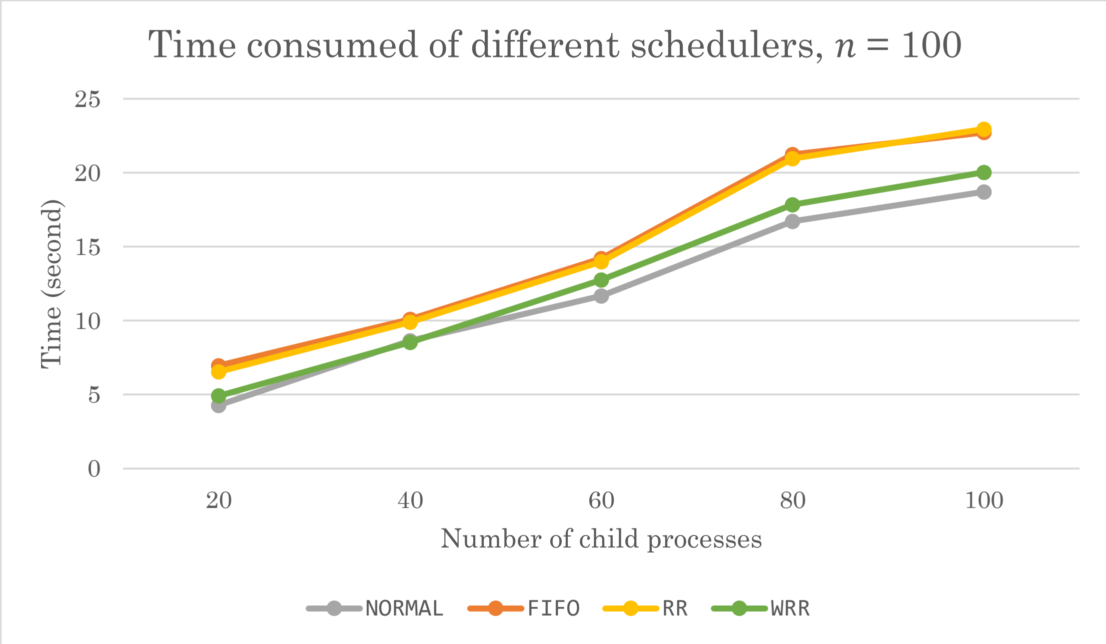
$n = 75$	$m = 20$		$m = 40$		$m = 60$		$m = 80$		$m = 100$	
	t (s)	Mflop/s	t (s)	Mflop/s	t (s)	Mflop/s	t (s)	Mflop/s	t (s)	Mflop/s
NORMAL	1.737	9.715	3.695	9.135	5.502	9.201	7.868	8.579	8.556	9.861
FIFO	1.838	9.182	4.721	7.149	6.337	7.989	10.506	6.425	10.400	8.113
RR	2.073	8.141	4.108	8.216	7.713	6.564	9.824	6.871	11.038	7.644
WRR	1.885	8.954	3.923	8.604	6.178	8.194	8.327	8.106	9.840	8.575



From the chart, we know that NORMAL has the shortest running time, while WRR has a better performance than FIFO and RR.

4.2.2. $n = 100$

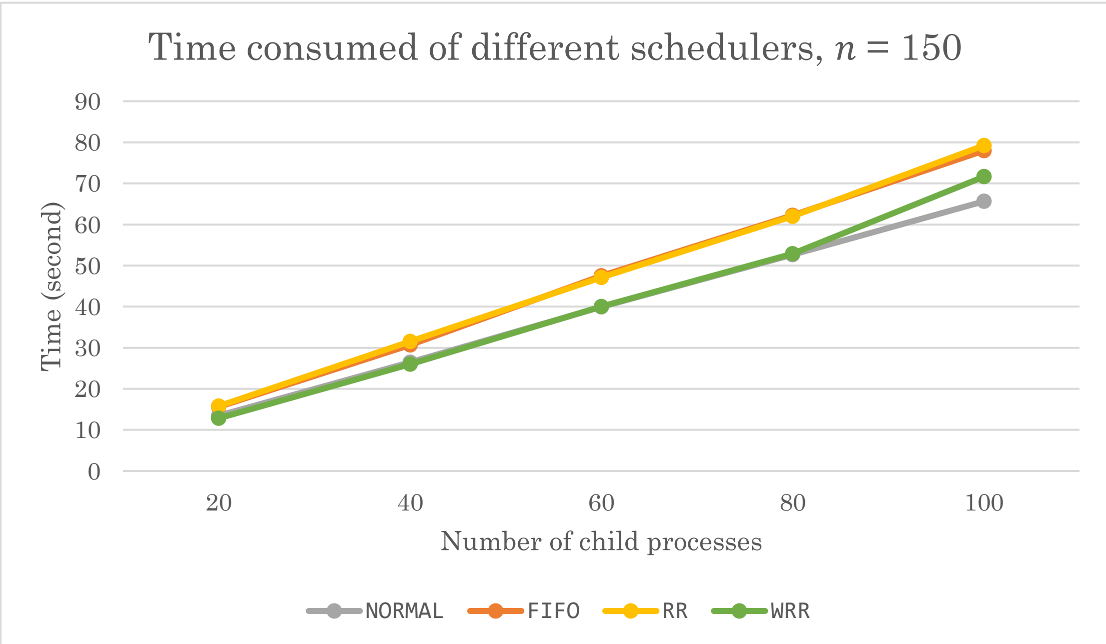
$n = 100$	$m = 20$		$m = 40$		$m = 60$		$m = 80$		$m = 100$	
	t (s)	Mflop/s	t (s)	Mflop/s	t (s)	Mflop/s	t (s)	Mflop/s	t (s)	Mflop/s
NORMAL	4.283	9.340	8.634	9.265	11.680	10.274	16.708	9.576	18.712	10.688
FIFO	6.959	5.748	10.084	7.933	14.212	8.443	21.229	7.537	22.720	8.803
RR	6.541	6.115	9.909	8.074	13.978	8.585	20.964	7.632	22.955	8.713
WRR	4.912	8.144	8.537	9.371	12.765	9.401	17.844	8.967	20.030	9.985



This chart shows that WRR has a better performance than other two RT schedulers more clearly. Still, NORMAL has the best performance, while FIFO and RR have very closed run time.

4.2.3. $n = 150$

$n = 150$	$m = 20$		$m = 40$		$m = 60$		$m = 80$		$m = 100$	
	t (s)	Mflop/s	t (s)	Mflop/s	t (s)	Mflop/s	t (s)	Mflop/s	t (s)	Mflop/s
NORMAL	13.475	10.018	26.477	10.197	39.950	10.138	52.657	10.255	65.627	10.285
FIFO	15.617	8.644	30.708	8.792	47.504	8.526	62.252	8.674	77.964	8.658
RR	15.799	8.545	31.501	8.571	47.145	8.591	61.967	8.714	79.220	8.521
WRR	12.835	10.518	26.009	10.381	40.059	10.110	52.866	10.214	71.623	9.424

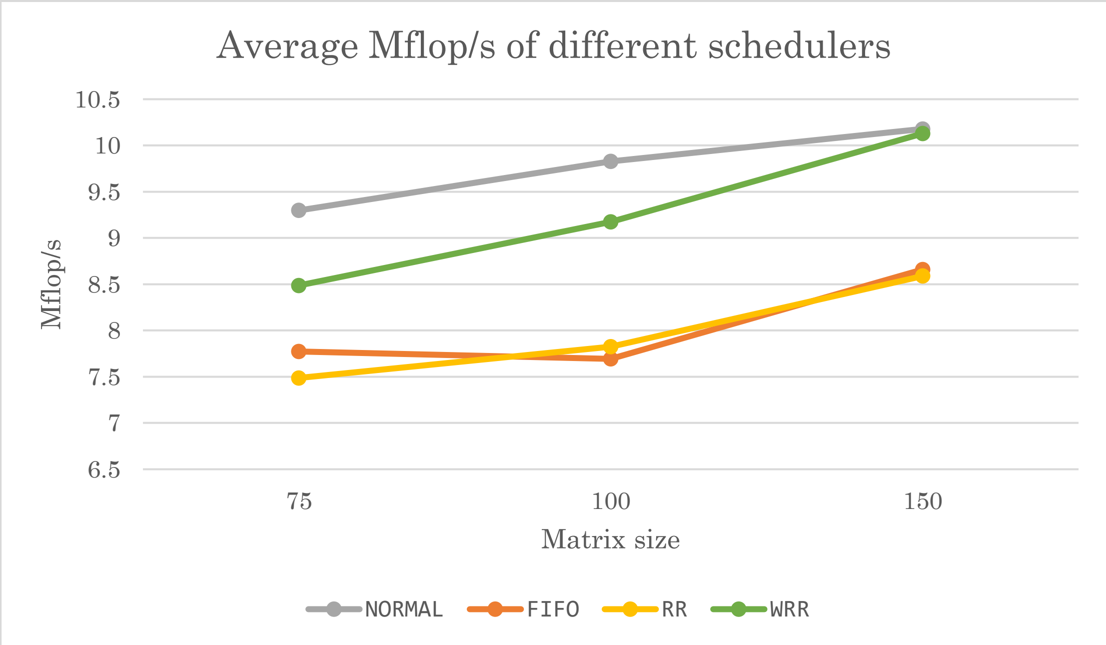


The performance gap becomes more evident when $n = 150$. It's notable that WRR's performance here is very closed to NORMAL scheduler except the last one $m =$

100. It seems that WRR is improving its performance as n goes larger.

4.2.4. Overall performance in Mflop/s

Average Mflop/s				
n	NORMAL	FIFO	RR	WRR
75	9.298	7.772	7.487	8.487
100	9.829	7.693	7.824	9.174
150	10.179	8.659	8.588	10.129



This chart shows that WRR scheduler indeed always has a better performance than two RT scheduler, FIFO and RR. However, its benchmark is still lower than that of NORMAL, but the difference shrinks as matrix size goes larger. The chart also implies that schedulers improve performances as matrix size goes larger. (But not for FIFO at $n = 100$.)

5. References

Linux kernel scheduler – Reference provided in the slides. Briefly introduces the process scheduler in Linux kernel.

Linux 进程管理（二）进程调度 – A very detailed Chinese blog, explaining the process scheduling concepts of different schedulers in Linux.

Linux 进程管理 (9)实时调度类分析，以及 FIFO 和 RR 对比实验 – Another detailed Chinese blog. This blog places more emphasis on the implementation of scheduling functions.