OS Project 2 Report

Linux Race-Averse Scheduler

Ziqian Zhao 520021910171

May 10, 2022

LINUX RACE-AVERSE SCHEDULER

ABSTRACT

In this project, we first implement a Memory Access Tracing (MAT) method to trace the number of times a given range of memory is written by a particular task. Then, based on the Memory Access Tracing, we design a Race-Averse Linux Process Scheduler (RAS), using a Weighted Round Robin style according to race probabilities of each task. Furthermore, we benchmark the performance of Linux native schedulers and our Race-Averse Scheduler. The results prove that RAS achieves higher throughput, shorter turnaround time and lower latency.

Key words: Linux Scheduler, Memory Access Tracing, Race-Averse, Weighted Round Robin

Contents

Chapter 1 Memory Access Tracing

1.1 MAT Introduction

In Linux, tracing memory access for tasks① is useful. Although the hardware uses a R bit and a M bit to track if a page has been referenced or modified, it does not maintain access frequency for each task. Therefore, we try to propose a per task memory access tracing method, which can trace the number of times a given range of memory is written by a particular task. Based on the tracing mechanism, we can use these tracing data to calculate the race probabilities between different tasks.

1.2 MAT Design

To tracing the writes of a given range of memory, we can take full advantage of the Page Fault mechanism. Firstly, invoke linux system call *mprotect* on given range of memory before any writes to set the memory as Read-Only. The system call *mprotect* changes the access protections for the calling task's memory pages containing any part of the given address range. If the calling task tries to access memory in a manner that violates the protections, then the kernel generates a SIGSEGV signal for the task[1]. Now, if we try to write to the protected memory, the kernel will send us a SIGSEGV signal. Secondly, after capturing the signal in user space, we can customize the method to deal with it. We call it the SIGSEGV handler. In the handler, we just simply invoke the *mprotect* again, but this time we set the memory as Read-Write. Then we can write to the memory normally. Thirdly, in the kernel space, on the path that the kernel generates the SIGSEGV signal due to access permission error, we can perceive that in the user space a write was performing. Thus, memory access tracing is achieved.

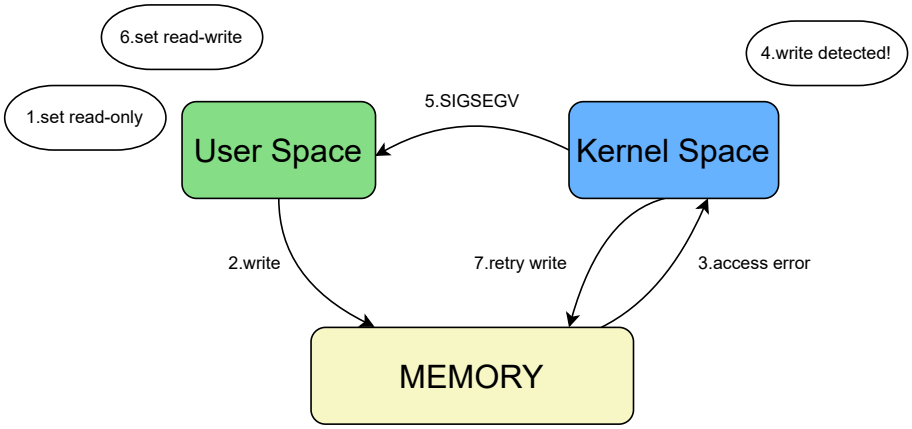

Figure 1–1 Memory Access Tracing Procedure

After the analysis above, we can find that to detect and record memory writes, we need to do some modification on the path that the kernel sends the SIGSEGV signal. If a SIGSEGV is to be generated and we are tracing the task, the kernel would add one to the write counts of the task.

① In Linux Scheduler, processes and threads are often treated equally, so here we don't make clear distinction, and refer to both processes and threads as tasks.

1.3 MAT Implementation

To apply the memory access tracing to linux kernel, we implement three system calls. Load the three modules into the kernel, and then we can use these system calls to start tracing, stop tracing, and getting the tracing results:

- ***sys_start_trace(pid_t pid)***: tell the kernel to start tracing memory writes of the given pid.
- ***sys_stop_trace(pid_t pid)***: tell the kernel to stop tracing memory writes of the given pid.
- ***sys_get_trace(pid_t pid, int *wcounts)***: get the memory writes times of the given pid.

Besides, we add some variables to the *task_struct*① to record the tracing states of the task. Finally, we add some code to the kernel on the path to send SIGSEGV. These code segments will update the write counts if we are tracing the running task.

1.3.1 Task Struct

We add a integer variable *wcounts* to record the write times and a boolean variable *trace_flag* to determine when to start tracing:

```
// include/linux/sched.h
struct task_struct {
  ...
  int wcounts; // record the write times
  int trace_flag; // 0 not tracing, 1 tracing
  ...
};
```

1.3.2 Update Write Count

In the kernel source code of the page fault process, we add some code to update the write counts if we are tracing the task:

```
// arch/arm/mm/fault.c
static int __kprobes
do_page_fault(unsigned long addr, unsigned int fsr, struct pt_regs *regs)
{
    ...
    if (fault & VM_FAULT_SIGBUS) {
    ...
    } else {
      ...
      if (tsk->trace_flag) // SIGSEGV! If tracing, add one to task wcounts
        tsk->wcounts++;
    }
    ...
}
```

1.3.3 System Calls

In *sys_start_trace*, we set the *trace_flag* of the task corresponding to the given pid to 1. If we have already been tracing the task, do nothing and return *-EINVAL*:

① *task_struct* is a data structure in linux kernel. It records the information of a process.

```c
static int sys_start_trace(pid_t pid)
{
    struct task_struct *task;
    task = get_pid_task(find_get_pid(pid), PIDTYPE_PID); // get task by pid
    if (!task || task->trace_flag == 1)
        return -EINVAL;
    task->trace_flag = 1; // start tracing
    task->wcounts = 0;
    return 0;
}
```

In *sys_stop_trace*, we set the *trace_flag* to 0, and reset the *wcounts* to 0. If we haven't traced the task yet, return *-EINVAL*:

```c
static int sys_stop_trace(pid_t pid)
{
    struct task_struct *task;
    task = get_pid_task(find_get_pid(_pid), PIDTYPE_PID); // get task by pid
    if (!task || task->trace_flag == 0)
        return -EINVAL;
    task->trace_flag = 0; // stop tracing
    task->wcounts = 0;
    return 0;
}
```

In *sys_get_trace*, we get the task's *wcounts*. If we haven't traced the task yet, return *-EINVAL*:

```c
static int sys_get_trace(pid_t _pid, int *wcounts)
{
    struct task_struct *task;
    task = get_pid_task(find_get_pid(_pid), PIDTYPE_PID); // get task by pid
    if (!task || task->trace_flag == 0)
        return -EINVAL;
    *wcounts = task->wcounts; // get the wcounts
    return 0;
}
```

1.4 MAT Test

We have written a test program to test three system calls and memory access tracing we implement. The test results are shown as following:

```
Invoke sys_start_trace
find memory accessed!
set memory read write!
memory[0] = 0
find memory accessed!
set memory read write!
memory[1] = 1
find memory accessed!
set memory read write!
memory[2] = 2
find memory accessed!
set memory read write!
memory[3] = 3
Task pid : 206, wcounts = 4, actual write times = 4
```

Figure 1–2 Start Tracing Test

```
Invoke sys_stop_trace
find memory accessed!
set memory read write!
memory[0] = 0
find memory accessed!
set memory read write!
memory[1] = 1
find memory accessed!
set memory read write!
memory[2] = 2
find memory accessed!
set memory read write!
memory[3] = 3
Task pid : 206, Wcount = 0, actual write times = 4
```

Figure 1–3 Stop Tracing Test

In Figure 1–2 we can see that after we invoke *sys_start_trace*, the memory access tracer can record the write times correctly. In Figure 1–3 we can see that after we invoke *sys_stop_trace*, the

kernel would stop tracing the task's memory accesses. Thus, we successfully implement the memory access tracing.

1.5 MAT Conclusion

In this project, we implemented the Memory Access Tracing and got the correct test results. Using the tracing data, we can further compute the race probabilities between different tasks. This can help us to accomplish the Race-Averse Scheduler.

However, there still exist some imperfections in our implementation. First, the tracing can not be done only by kernel. We have to use *mprotect* to set protection for the memory before any write in user space. Can we set protection automatically in kernel space? If we achieve this, we can trace the memory access only by invoking *sys_start_trace*, and perform writes without additional explicit protection in user space, which is apparently more reasonable and elegant. Second, because we borrow the page fault handler to detect the writes in kernel space, every write associated with the tracing memory will result in a page fault. Is it an acceptable overhead? If it significantly affects the write performance, we have to re-evaluate its value.

Chapter 2 Race-Averse Scheduler

2.1 RAS Introduction

CPU scheduling is the basis of multiprogrammed operating systems. By switching the CPU among tasks, the operating system can make the computer more productive. In a single-processor system, only one task can run at a time. Others must wait until the CPU is free and can be rescheduled[2]. Through frequently scheduling and context switch, tasks can run in a concurrent manner. The role of the Linux scheduler is to select the next task to run and distribute the running time.

Linux implements several kinds of schedulers for different scheduling situation. For example, Linux has RT Scheduler for real-time tasks, CFS Scheduler for normal tasks and IDLE Scheduler for tasks with least importance. When these schedulers pick a task, they take a variety of factors into account. However, the likelihood that a task may race with other currently running tasks is not considered. Therefore, we propose a Race-Averse Scheduling (RAS) algorithm. The scheduling is a Weighted Round Robin style scheduling according to race probabilities of each running task.

2.2 RAS Design

2.2.1 How RAS works

First we will introduce the Weighted Round Robin scheduling. Round Robin scheduling treats all tasks equally. It distribute the same *timeslice*① to every picked task. When a task runs out of its timeslice, it would yield the execution by its self, and be put back on the end of the *run queue*②.

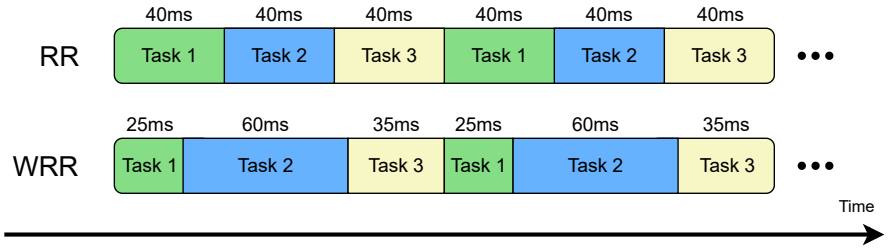

Figure 2–1 Round Robin & Weighted Round Robin

However, there are some situations where it is desirable to give some tasks preference over the others. That is, distributing longer timeslice to some tasks. Weighted Round Robin actually does this. The only difference between Round Robin and Weighted Round Robin is that Weighted Round Robin will assign a weight to each task, and tasks with higher weights will be given longer timeslice to run.

Obviously, in Race-Averse Scheduler, the weights are determined by the race probabilities. In our design, the race probability of a task is represented as an integer in the range of [0, 10). The higher the integer is, the more likely that the task may race with others. For CPU, it is rational to give

① A concept in Linux Scheduling. Refers to a period of time for the task to run. Usually 10ms~100ms.

② A concept in Linux Scheduling. Every task which is ready but not running will be waiting in the run queue for being picked to run.

preference to the tasks with lower race probabilities. That is exactly what the Race-Averse means. Therefore, tasks with lower race probabilities will be assigned with higher weights, which is, with longer timeslice to run.

With the assistance of the Memory Access Tracing we implemented before, we can quantify the race probabilities between different running tasks. To simplify the situation, we assume that a large enough number of tasks are in scheduling, and they may compete at a given range of virtual addresses. Thus, the race probability of a certain task is proportional to its frequency of accessing the given range of virtual addresses. We can use the following equation to calculate the race probability:

$$race\_prob_i = 10 \times \frac{wcounts_i}{\sum_i wcounts_i} \tag{2–1}$$

We set the maximum timeslice of the Race-Averse Scheduler to 100ms and the minimum timeslice to 10ms. And we can inversely map the race probabilities from 0 to 9 into the timeslice from 100ms to 10ms:

$$\begin{aligned} timeslice &= \frac{max\_timeslice - min\_timeslice}{0-9} \times race\_prob + max\_timeslice \\ &= -10ms \times race\_prob + max\_timeslice \end{aligned} \tag{2–2}$$

When a task runs out of its timeslice, the RAS will compute the sum of *wcounts* of the tasks in RAS run queue, and further calculate its race probability. Then assign a new timeslice according to its race probability, and put the task back to the end of the run queue. We should notice that the weight of a task can be changed dynamically, due to the newly coming tasks, completed tasks and some other reasons. Therefore, we have to re-compute the race probability every time the task needs to be assigned with new timeslice.

2.2.2 Into Linux Kernal

Now we are familiar with how the Race-Averse Scheduler works. Next, we will look into the Linux kernel and introduce how to embed the Race-Averse Scheduler into the kernel.

There are many ready tasks waiting for running in kernel. To manage these tasks, the kernel maintains a *Run Queue* of these tasks. Besides, we call the task that the CPU is running *Current Task*. To switch different tasks to Current Task, first the kernel should set the *Need Resched* flag to Current Task, and second the kernel should check the Need Resched flag at a proper time. If Current Task needs to be rescheduled, the kernel calls *Schedule* function to perform a context switch. The proper time actually refers to the time when CPU is interrupted by the ticker timer and etc.

The Schedule function picking the next task needs the help of *Scheduling Class*. Linux uses Scheduling Class to implement different scheduling algorithms. Here we take *rt_sched_class*①, *cfs_sched_class*②, *idle_sched_class*③ and our *ras_sched_class* as example. These classes are linked together through a linked list. When Schedule function wants to pick the next task, it will iterate through the schedule classes linked list and try to get a task from their own run queues. Once it gets

① *rt_sched_class* implements FIFO and RR scheduling algorithms.
② *cfs_sched_class* implements Completely Fair Scheduling algorithm.
③ *idle_sched_class* implements IDLE scheduling algorithm.

a valid task, it would return directly and set the task as Current Task. So it is easy to see that every schedule class has different priorities. The head class of the linked list has higher priority than the tail class. Only when the former class doesn't have any task in run queue, the current class can pick its task in its run queue to return. Therefore, if we make different queue arrangement rules for different schedule classes, the tasks can be scheduled with different scheduling algorithms. For example, the run queue of *rt_sched_class* contains multi priorities queues, and the run queue of *cfs_sched_class* is a Red-Black Tree. From our previous discussion, we can learn that the run queue of RAS is a simple single queue.

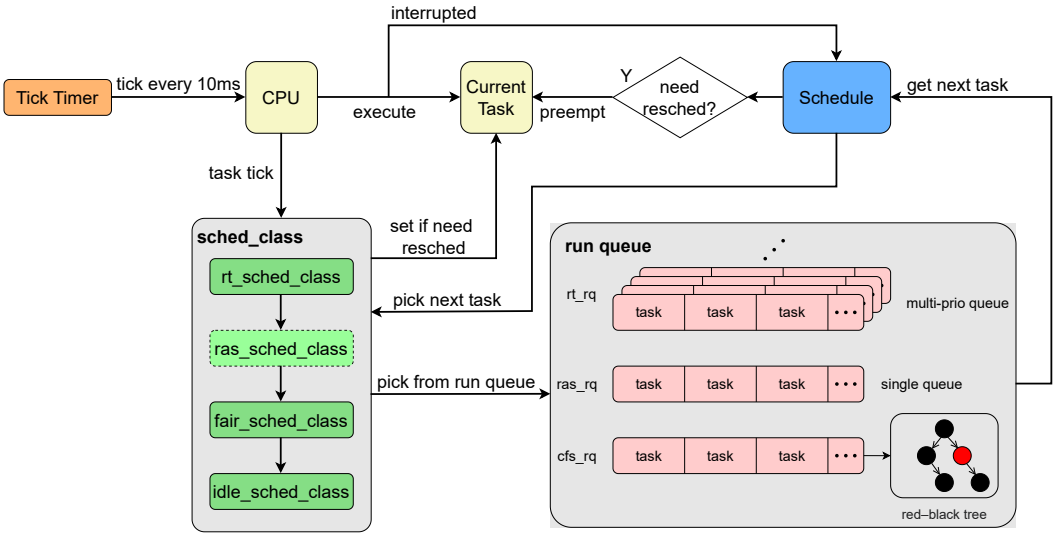

Figure 2–2 Linux Scheduling Procedure

Therefore, what we really need to implement is a new schedule class, *ras_sched_class*.

2.3 RAS Implementation

The Scheduler Class has the following structure:

```
// include/linux/sched.h
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);
  void (*task_tick) (struct rq *rq, struct task_struct *p, int queued);
  ...
};
```

All the scheduler classes should implement these functions. Their roles are closely related to the scheduling procedure of Linux Scheduler introduced above.

- ***next***: points to the next scheduler class in the linked list. In our design, the next scheduler of RAS is Fair Scheduler, and the former scheduler of RAS is RT Scheduler.
- ***enqueue_task***: enqueue a task into the run queue.
- ***dequeue_task***: dequeue a task from the run queue.
- ***pick_next_task***: pick a task from the run queue to be the next Current Task.

- ***task_tick***: invoked when the tick timer ticks. Update some variables.

These four functions are the most important functions that we should implement for our Race-Averse Scheduler. Before implement these functions, we will define some structures associated with RAS.

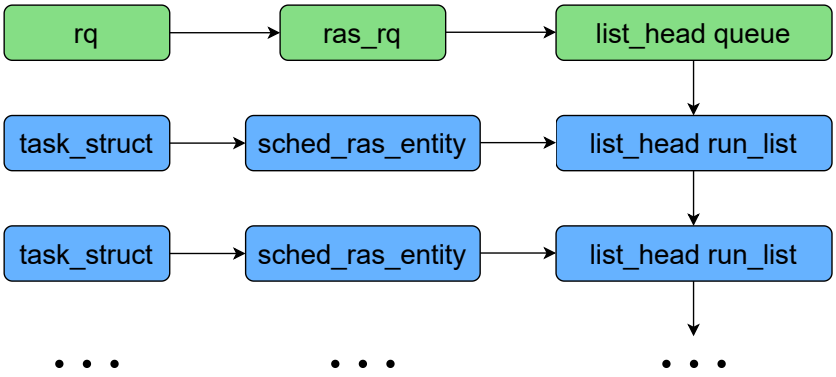

Figure 2–3 RAS Structures

2.3.1 Structure ras_rq

Structure *ras_rq* is the implementation of the run queue of RAS.

```
// kernel/sched/sched.h
struct ras_rq
{
  struct list_head queue;
  unsigned long ras_nr_running;
};
```

- ***queue***: the head of the RAS run queue linked list.
- ***ras_nr_running***: the number of tasks in RAS run queue.

2.3.2 Structure sched_ras_entity

Structure *sched_ras_entity* represents an entity which can be scheduled by RAS, in other words, a task with the RAS scheduling strategy. Therefore, we should add a *sched_ras_entity* variable in the *task_struct*.

```
// include/linux/sched.h
struct sched_ras_entity
{
  struct list_head run_list;
  unsigned int time_slice;
  unsigned int total_timeslice;
};
```

- ***run_list***: the node in the RAS run queue linked list.
- ***time_slice***: the timeslice left to run.
- ***total_timeslice***: the timeslice allocated in this round.

With these structures defined, we can implement our RAS functions.

2.3.3 Function enqueue_task_ras

When a new task with RAS scheduling strategy comes into the kernel, or a task was switched from other scheduling strategy to RAS, the kernel will invoke *enqueue_task_ras* to put the task into the RAS run queue.

```
// kernel/sched/ras.c
static void
enqueue_task_ras(struct rq *rq, struct task_struct *p, int flags)
{
  struct sched_ras_entity *ras_se = &p->ras;
  struct ras_rq *ras_rq = &rq->ras;
  struct list_head *queue = &ras_rq->queue;
  list_add_tail(&ras_se->run_list, queue);
  ras_rq->ras_nr_running++;
  inc_nr_running(rq);
}
```

We add the *sched_ras_entity* to the tail of RAS run queue, and increase the count of running tasks.

2.3.4 Function dequeue_task_ras

When a task with RAS scheduling strategy finishes its job, or a task was switched from RAS to other scheduling strategy, the kernel will invoke *dequeue_task_ras* to remove the task from the RAS run queue.

```
// kernel/sched/ras.c
static void
dequeue_task_ras(struct rq *rq, struct task_struct *p, int flags)
{
  struct sched_ras_entity *ras_se = &p->ras;
  struct ras_rq *ras_rq = &rq->ras;
  list_del_init(&ras_se->run_list);
  ras_rq->ras_nr_running--;
  dec_nr_running(rq);
}
```

We delete the *sched_ras_entity* from RAS run queue, and decrease the count of running tasks.

2.3.5 Function pick_next_task_ras

When the Scheduler function in the kernel wants to pick a task from the RAS run queue, *pick_next_task_ras* function will be called. It returns a task in the RAS run queue if there was.

```
// kernel/sched/ras.c
static struct task_struct *
pick_next_task_ras(struct rq *rq)
{
  struct sched_ras_entity *ras_se;
  struct task_struct *p;
  struct ras_rq *ras_rq = &rq->ras;
  if (!ras_rq->ras_nr_running){
    // No task in run queue, return null.
    return NULL;
  }
  struct list_head *queue = &ras_rq->queue;
```

```
  ras_se = list_entry(queue->next, struct sched_ras_entity, run_list);
  p = ras_task_of(ras_se);
  return p;
}
```

The *pick_next_task_ras* function will select the head node of the run queue linked list to return. And if there is no task in run queue, it will return null.

2.3.6 Function task_tick_ras

The *task_tick_ras* function is the most vital function for implementing Race-Averse Scheduler. In this function, we will compute the race probability of a task and dynamically allocate the task with some timeslice according to its race probability with other running tasks.

```
// kernel/sched/ras.c
static void
task_tick_ras(struct rq *rq, struct task_struct *task, int queued)
{
  struct sched_ras_entity *ras_se = &task->ras;
  struct ras_rq *ras_rq = &rq->ras;
  if (task->policy != SCHED_RAS)
    return;
  if (--task->ras.time_slice)
    return;
  if (ras_rq->ras_nr_running == 1){
    // No race. Set MAX timeslice to avoid frequently schedule.
    task->ras.time_slice = RAS_MAX_TIMESLICE;
    task->ras.total_timeslice = RAS_MAX_TIMESLICE;
  } else {
    // Calculate race probability.
    int wcounts = task->wcounts;
    struct list_head *p;
    struct sched_ras_entity *ras_se_tmp;
    struct task_struct *t;
    struct list_head *queue;
    queue = &ras_rq->queue;
    int sum = 0;
    int task_cnt = 0;
    list_for_each(p, queue)
    {
      ras_se_tmp = list_entry(p, struct sched_ras_entity, run_list);
      t = ras_task_of(ras_se_tmp);
      sum += t->wcounts;
      task_cnt++;
    }
    if (sum == 0 || sum == wcounts) {
      // not tracing or no memory write
      task->ras.time_slice = RAS_MAX_TIMESLICE;
      task->ras.total_timeslice = RAS_MAX_TIMESLICE;
    } else {
      int race_prob = wcounts * 10 / sum;
      unsigned int timeslice = -1*race_prob + RAS_MAX_TIMESLICE;
      task->ras.time_slice = timeslice;
      task->ras.total_timeslice = timeslice;
    }
  }
```

```
    // Requeue to the end of queue if we are NOT the only element on the queue.
    if (ras_se->run_list.prev != ras_se->run_list.next){
          requeue_task_ras(rq, task, 0);
          set_tsk_need_resched(task);
      }
}
```

The *task_tick_ras* function will be invoked every time the tick timer ticks. The tick timer ticks every 10ms. If the task has timeslice left, the function reduces its timeslice by one. If the task has no timeslice left, it indicates that the task should yield the CPU and be re-queued back to the tail of the run queue. Before the task is re-queued, it will be allocated with a new timeslice. If the task isn't being traced by Memory Access Tracer, or there is no memory write or task race, it will be allocated with the maximum timeslice. Otherwise, we iterate the tasks in the run queue and compute the race probability of the current task. Then allocate the timeslice according to its race probability.

2.3.7 Some other functions

With the four functions mentioned before, we can archive the basic operations of Race-Averse Scheduler. But we still need some other functions to implement the whole operations of RAS.

1. ***yield_task_ras***: when the task wants to yield the CPU, this function will be called.

```
static void
yield_task_ras(struct rq *rq)
{
    requeue_task_ras(rq, rq->curr, 0);
}
```

2. ***update_curr_ras***: update some statistical information of CPU and tasks.

```
static void update_curr_ras(struct rq *rq)
{
    struct task_struct *curr = rq->curr;
    struct sched_ras_entity *ras_se = &curr->ras;
    struct ras_rq *ras_rq = &rq->ras;
    u64 delta_exec;
    if (curr->sched_class != &ras_sched_class)
        return;
    delta_exec = rq->clock_task - curr->se.exec_start;
    if (unlikely((s64)delta_exec < 0))
        delta_exec = 0;
    schedstat_set(curr->se.statistics.exec_max,
            max(curr->se.statistics.exec_max, delta_exec));
    curr->se.sum_exec_runtime += delta_exec;
    curr->se.exec_start = rq->clock_task;
}
```

3. ***put_prev_task_ras***: called before another task replaces the current task. For some statistical information.

```
static void
put_prev_task_ras(struct rq *rq, struct task_struct *p)
{
    update_curr_ras(rq);
}
```

4. ***set_curr_task_ras***: called when the scheduling policy of the current task changes to RAS. For some statistical information.

```
static void
set_curr_task_ras(struct rq *rq)
{
  struct task_struct *p = rq->curr;
  p->se.exec_start = rq->clock_task;
}
```

5. ***get_rr_interval_ras***: return the total timeslice allocated before. For system calls.

```
static unsigned int
get_rr_interval_ras(struct rq *rq, struct task_struct *task)
{
  if (task->policy == SCHED_RAS)
  {
    struct sched_ras_entity *ras_se = &task->ras;
    return ras_se->total_timeslice;
  }
  return 0;
}
```

Besides these structures and functions, we also have to re-write some code in the kernel source code to apply RAS to the kernel. For example, declare some variables and add some branch conditions of RAS. Please refer to the source code for detailed modification.

2.4 RAS Test

In this section, we write some test programs to check that whether RAS works correctly.

2.4.1 Switch Test

First, we launch a infinite loop program and get its *pid*. Second, we use another program to change its scheduling policy to RAS, and then try to change back. Output information:

```
Start RAS scheduler test ...
Please input the pid of the test task: 278
Current scheduling algorithm is SCHED_NORMAL
Please input the choice of scheduling algorithm (0-NORMAL, 1-FIFO, 2-RR, 6-RAS): 6
Please input the priority: 0
Changing ...
Current scheduling algorithm is SCHED_RAS
Done.
```

Figure 2–4 Change to RAS Policy

```
Start RAS scheduler test ...
Please input the pid of the test task: 278
Current scheduling algorithm is SCHED_RAS
Please input the choice of scheduling algorithm (0-NORMAL, 1-FIFO, 2-RR, 6-RAS): 0
Please input the priority: 0
Changing ...
Current scheduling algorithm is SCHED_NORMAL
Done.
```

Figure 2–5 Change Back to NORMAL Policy

In Figure 2–4 we can see that the task with pid 278 is successfully changed with the RAS policy. In Figure 2–5 we can see that it can also be changed back with the NORMAL policy. This proves that RAS is successfully embedded into Linux kernel.

2.4.2 Multi-Tasks Test

Now we write a test program to fork 10 child tasks and set their scheduling policy as RAS. Each task will write random times [128, 1024) to a given range of memory which is traced. Then we will check that RAS can whether correctly arrange their executions and allocate them with different timeslices according to their race probabilities. Output information is as following:

```
[tick] task count: 2, no write. set task 210 timeslice 100 ms
[tick] task count: 2, wcounts: 335, sum: 512, race_prob: 6. set task 212 timeslice 40 ms
[tick] task count: 7, wcounts: 551, sum: 886, race_prob: 6. set task 210 timeslice 40 ms
[tick] task count: 7, wcounts: 404, sum: 955, race_prob: 4. set task 212 timeslice 60 ms
[tick] task count: 8, wcounts: 250, sum: 1205, race_prob: 2. set task 216 timeslice 80 ms
[tick] task count: 8, wcounts: 314, sum: 1519, race_prob: 2. set task 215 timeslice 80 ms
[tick] task count: 8, wcounts: 441, sum: 1960, race_prob: 2. set task 213 timeslice 80 ms
[tick] task count: 7, wcounts: 757, sum: 2166, race_prob: 3. set task 210 timeslice 70 ms
[tick] task count: 5, wcounts: 282, sum: 2044, race_prob: 1. set task 218 timeslice 90 ms
[tick] task count: 4, wcounts: 496, sum: 1976, race_prob: 2. set task 215 timeslice 80 ms
[tick] task count: 4, wcounts: 717, sum: 2252, race_prob: 3. set task 213 timeslice 70 ms
[tick] task count: 4, wcounts: 958, sum: 2453, race_prob: 3. set task 210 timeslice 70 ms
[tick] task count: 4, wcounts: 496, sum: 2667, race_prob: 1. set task 218 timeslice 90 ms
[tick] task count: 3, wcounts: 810, sum: 2264, race_prob: 3. set task 213 timeslice 70 ms
[tick] task count: 2, wcounts: 706, sum: 1516, race_prob: 4. set task 218 timeslice 60 ms
[tick] task count: 1, no race. set task 218 timeslice 100 ms
```

Figure 2–6 RAS Multi Tasks Test Result

In Figure 2–6 we can see that the RAS correctly scheduled these tasks in a weighted round robin style. When a task run out of its timeslice, it will be allocated with a new timeslice according to its race probability. Then it will be re-queued to the tail of the run queue. And RAS will pick the first task in the run queue as the next task to run.

These results prove that we successfully implemented the Race-Averse Scheduler.

Chapter 3 Benchmark

3.1 Benchmark Introduction

We can benchmark a scheduler by many factors, for example, CPU utilization, throughput, turnaround time, latency, waiting time and etc. In this chapter, we will benchmark throughput, latency and turnaround time of SCHED_NORMAL, SCHED_FIFO, SCHED_RR and SCHED_RAS scheduling algorithms.

3.2 Benchmark Design

First, we will explain the meaning of these test items in detail:

- **throughput**: number of tasks that complete their execution per time unit.
- **latency**: time interval between the arrival of the task and the first execution of the task.
- **turnaround time**: time interval between the arrival of the task and the completion of the task.

We know that using Memory Access Tracing as the basis, RAS is write-sensitive. Thus, our benchmark program is I/O bound. The benchmark program will using *mmap* system call to request a block of memory. We trace this memory's access using the tracer we implemented in Chapter One. Then benchmark program will fork a given number of child tasks to write random times into the memory. The four scheduling policies will be used in rotation under the same conditions.

3.3 Benchmark Results

3.3.1 Throughput

First, we benchmark the total running time of different scheduling policies with different task numbers. Each benchmark performs an average of 16368 writes per 10 tasks. The results are shown as following table:

Table 3–1 Time Consumption of Each Benchmark (s)

SCHED	Task Number				
	20	40	60	80	100
NORMAL	22.259	30.568	35.777	41.616	50.145
FIFO	7.441	15.152	21.938	28.863	37.169
RR	7.166	15.089	21.776	28.040	37.493
RAS	6.140	11.952	17.358	22.330	29.473

It is apparent to see that RAS takes the least amount of time to complete the write tasks. FIFO and RR have nearly the same performance. NORMAL has the worst performance. Compared to FIFO and RR, RAS has about 20% optimazation.

With the time consumption and task number of each scheduling algorithm, we can compute their throughput, number of tasks divided by time consumption. The results are as following chart:

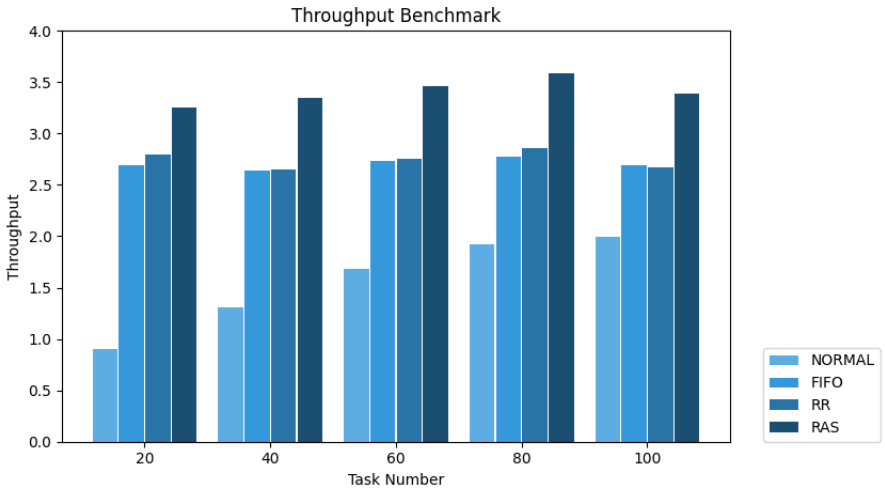

Figure 3–1 Throughput Benchmark Result

In this chart we can see that RAS gets the highest throughput. For different task numbers, the throughput is about 3.4 tasks per second. When the task number is too large, scheduling overhead may affect the throughput performance.

3.3.2 Turnaround Time

We record the fork time of a task and the completion time of it. And the turnaround time is their difference. A task will write minimum 16 times and maximum 8192 times. The results are as following chart:

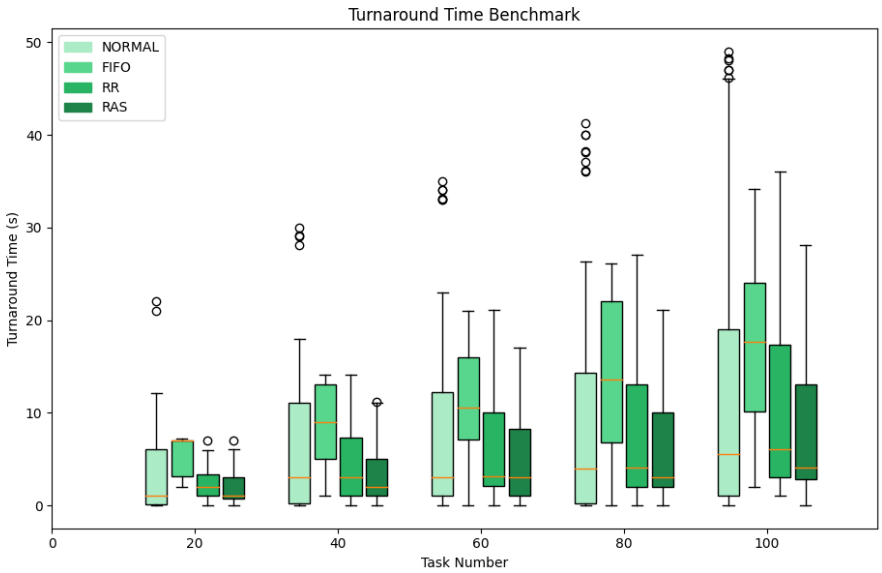

Figure 3–2 Turnaround Time Benchmark Result

It is crystal clear that RAS gains the lowest average, the lowest median and the lowest upper boundary turnaround time. Besides, RAS also has the fewest outliers. It proves that among the four scheduler, RAS has the best and the most stable performance. Furthermore, we can notice that as the number of tasks increases, RAS will get better performance. And when task number is relatively low, RAS will achieve a similar performance to RR. This can be easily explained by the limited race situation when the number of tasks is small. Under these circumstances, the RAS will nearly allocate all the tasks with the maximum timeslice.

3.3.3 Latency

We use the difference between the fork time of a task and the first execution time to represent the latency. Also we choose 20, 40, 60, 80 and 100 task numbers and write minimum 16 times, maximum 8192 times. The results are as following chart:

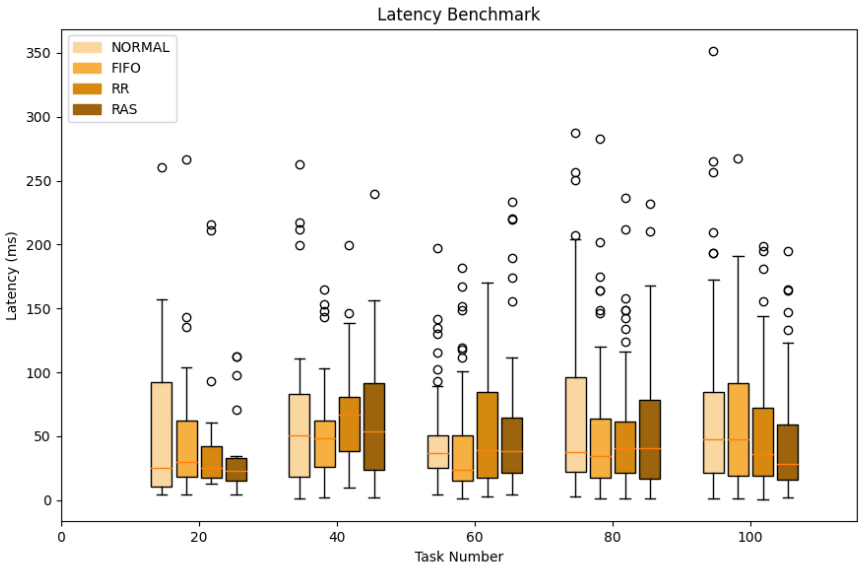

Figure 3–3 Latency Benchmark Result

In this chart we can find that RAS does not achieve great advantage in latency. The performances of RAS and RR are almost the same. The reason is that to reduce the overhead of context switch, we set the maximum timeslice (100ms, the same as RR timeslice) as the initial timeslice to RAS tasks. Therefore, when tasks first enter the kernel, they will be allocated with 100ms timeslice to run. That results in a larger latency for other tasks waiting for CPU. But compared to NORMAL and FIFO policies, RAS still gains optimization.

Summary

In this project, we first implemented the Memory Accessing Tracing method. Then, based on the tracer, we designed and implemented a new Linux scheduling algorithm RAS. Next, we successfully embedded it into Linux kernel, and proved that it can work correctly. Finally, we wrote several benchmark programs to test the performance of RAS and compared RAS with Linux native scheduling algorithms, NORMAL, FIFO and RR. The benchmark results shown that RAS has higher throughput, shorter & more stable turnaround time and slightly lower latency.

Although the results are exciting, we still have to note that RAS has many limitations. For example, Memory Access Tracing can not be done completely in the kernel so far. Therefore, if we want to use RAS, we have to manually set protection for the specified memory again and again. Besides, RAS is only sensitive to write operation. So for CPU-bound tasks, RAS can not gain a better performance. We still need to make further improvements for RAS.

For me, this is my first time to read the Linux source code. That's a really challenge. But I learned a lot from it. This project has laid a foundation for me to explore the field of system in the future.

References

[1] Linux Manual Page[EB / OL]. https://man7.org/linux/man-pages/man2/mprotect.2.html.

[2] SILBERSCHATZ A, GALVIN P B, GAGNE G. Operating System Concepts. 9th edition[M]. America: John Wiley & Sons, Inc, 2013: 261.

Acknowledgements

Thanks Prof. Wu. Your excellent teaching helps me a lot.
Thanks TAs for this project. I appreciate your quick replies.
Thanks my classmates. Our discussion greatly inspired me.
Thanks @sjtug. Your LaTeX templates have saved me a lot of time.