OS Project2 Report

520021910063 苏寄闲

1. Problem 2

1.1 Problem restatement

Implement three system calls to support per-task page access tracing, which can trace the number of times a page is written by a particular task. Then revise kernel code to assist page tracing.

1.2 Problem analysis and implementation

When page fault happens, the kernel invokes function do_page_fault in arch/arm/mm/fault.c which in turns invokes the function ___do_page_fault, which returns the type of fault.

Function __do_page_fault first checks whether the mm_struct has an invalid vma and whether the address is invalid. If not, then the vma for memory access is valid and then the page fault could be caused by forbidden operation. Therefore, access_error is invoked.

Function access_error compares the flags in vma->flags and the flags in register fsr which is defined in arch/arm/mm/fault.h. This also where the kernel **check the write fault** and **revises the wcounts**. If the write permission declared in vma->flags does not agree with the register fsr, the kernel would check the trace_flag and increase the wcounts accordingly. If they are distinct, return true and __do_page_fault returns VM_FAULT_BADACCESS.

```
if (fsr & FSR_WRITE){ // check if the process tries to write
    mask = VM_WRITE;

    // check if the vma allows to write and if the process is being traced
    if(tsk->trace_flag && !(vma->vm_flags & mask))
      tsk->wcounts++;
  }
```

Then back in do_page_fault, if the page fault is VM_FAULT_BADACCESS, it would call function do_user_fault which then sends signal to user thread. The rest could be handled in segv_handler in user program.

2. Problem3

2.1 Problem restatement

Implement a Race-Averse-Scheduling algorithm to the Linux which adopts a weighted round robin style scheduling according to race probabilities of each tasks.

2.2 Race probabilities

According to the requirements, the wcounts of tasks should be mapped into the range $[0, 10)$.

As race probability is an amount related to the overall situation, the wcounts of other tasks should also be taken into consideration. Also, if the number of tasks running on ras_rq is huge, the wcounts of a specific task would only account for a small percentage. Therefore, it is assumed that the race probability is positively correlated with ratio of the wcounts to the average wcounts on ras_rq.

$$prob \propto ratio = \frac{wcounts}{average}$$

It is noticed that the ratio is unbounded, and it is not desired that extreme data have too much impact on the algorithm. Therefore, the probability should not be proportional to the ratio. Instead, I used a function for the calculation.

$$prob = A + \frac{B}{ratio + C}$$

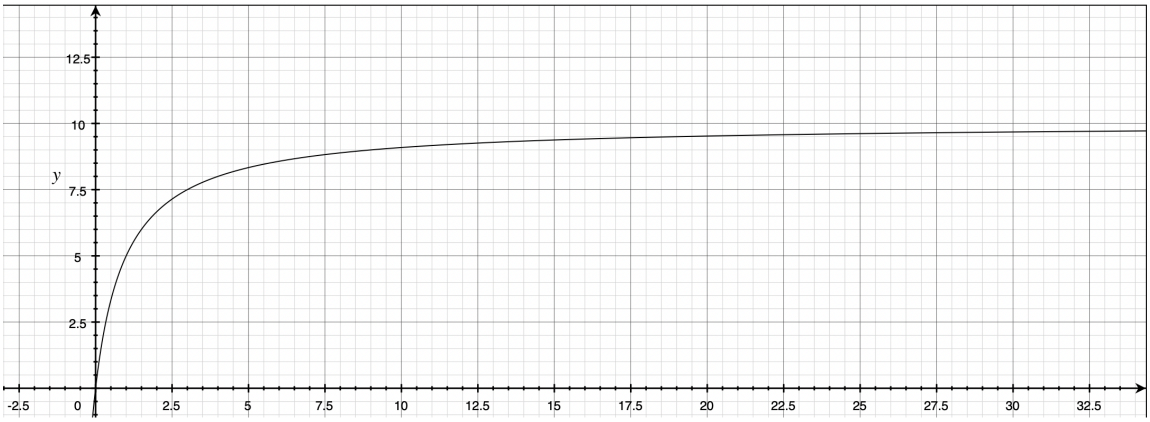
It is assumed that

- When the ratio is 0, which means the wcounts is 0 (or close to 0), the probability is 0.
- When the ratio is 1, which means the wcounts is equal to the mean, the probability is 5.
- When the ratio approaches $+\infty$, the probability is 9.

$$
\begin{gathered}
ratio = 0 \Rightarrow prob = 0 \\
ratio = 1 \Rightarrow prob = 5 \\
ratio \to +\infty \Rightarrow prob = 9
\end{gathered}
$$

A possible solution is that $A = 10, B = -10, C = 1$

$$prob = 10 - \frac{10}{1 + ratio}$$



Finally, the time slice allocated to a task is defined as

$$time\ slice = 10 - prob = \frac{10}{1 + ratio}$$

making the time slice allocated inversely proportional to the wcounts.

As the kernel does not support float operation, I discretized the function.

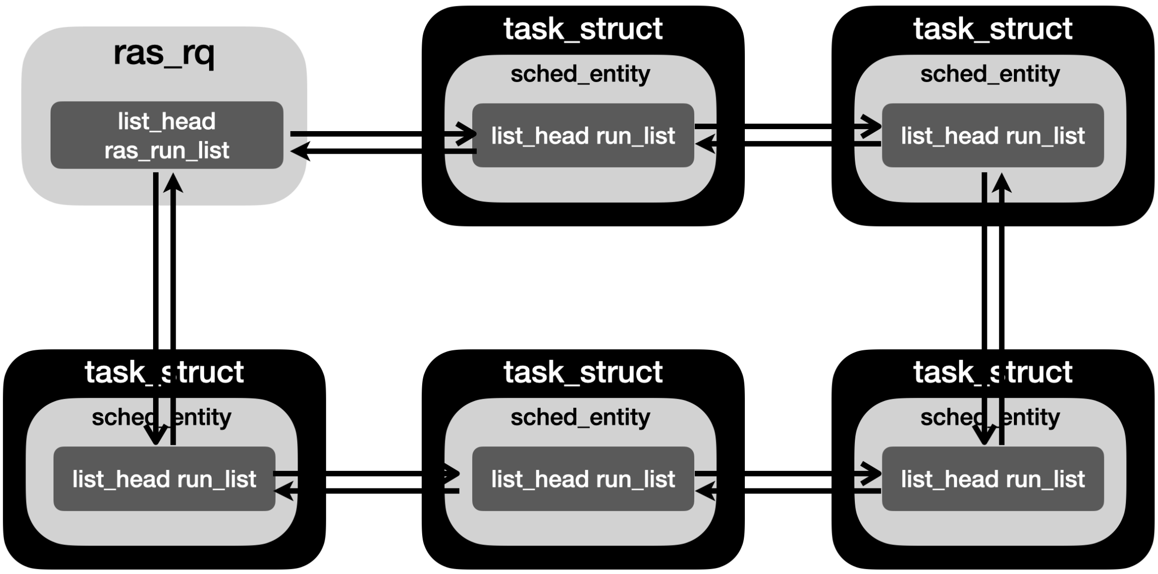
$$ratio = \lfloor \frac{wcounts}{averate} \rfloor$$

$ratio$	0	$(0, \frac{1}{9})$	$[\frac{1}{9}, \frac{1}{4})$	$[\frac{1}{4}, \frac{1}{2})$	$[\frac{1}{2}, 2)$	$[2, 4)$	$[4, 9)$	$[9, \infty)$
$prob$	0	1	2	3	5	7	8	9

2.3 Analysis and implementation

2.3.1 Data structure for RAS

Referring to SCHED_RR, I declared a double circular linked list which links all the sched_entity_ras on ras_rq together, scheduling with sched_entity as unit.



2.3.2 Functions in ras.c

2.3.2.1 State switching of a task

When the state of a task is switched to or from TASK_RUNNING, the kernel will perform enqueueing and dequeueing operations for the task and the ready queue, which is implemented by `enqueue_task_ras` and `dequeue_task_ras`.

`enqueue_task_ras` enqueues a `sched_entity` to the `ras_rq`, increases `nr_running` as well as add the wcounts of the task to `total_wcounts`.

`dequeue_task_ras` dequeues a `sched_entity` from the `ras_rq`, decreases `nr_running` as well as sub the `total_wcounts` by the `wcounts` of the task.

2.3.2.2 Invoking scheduling operations

There are two cases where the current task will be marked that it needs rescheduling. One is when a sleeping task is woken up by `try_to_wake_up`. In this case, the task is enqueued and the current task is marked as TIF_NEED_RESCHED if the woken task can preemt it. Another case is that the kernel will periodically call the function `scheduler_tick` which in turns calls the function `task_tick` of the sched_class of the current task.

`task_tick_ras` is periodically called if the current task has scheduling policy SCHED_RAS. It decreases the time slice of the current task by one each time called. If the current task has no remaining time slice, it would be requeued to the end of the `ras_rq`, reallocated proper time slice and would be set as TIF_NEED_RESCHED.

2.3.2.3 Performing scheduling operations

There are two cases where `__scheduler` will be invoked. One is that when a system call or interrupt returns, the kernel would call it according to the flag TIF_NEED_RESCHED of the current task. The other is when the current task voluntarily yields.

When `__schedule` is invoked, it calls `put_prev_task` and `pick_next_task` which in turns calls the corresponding function of the sched class.

`put_prev_task_ras` updates the clock for ras_rq. As I didn't implement load balance, no other work should be done.

`pick_next_task_ras` returns the task at the head of the queue.

`yield_task_ras` requeues the task to the end of the queue to pause it.

2.3.2.4 Other functions

`get_rr_interval_ras` calls `get_timeslice` and returns the time slice for a task. This function is invoked by the system call `sys_sched_rr_get_interval`.

`switched_to_ras`, `prio_changed_ras`and `check_preempt_curr_ras` that are related to priority and preemption would be discussed in extra work.

2.3.3 Revision in other files

- `/include/linux/sched.h`
    - ***line 40*** Defined `SCHED_RAS` as 6
    - ***line 1301*** Defined `sched_ras_entity` referring to `sched_rt_entity`
    - ***line 1256*** Defined `RAS_TIMESLICE` referring to `RT_TIMESLICE`
    - ***line 1252*** Defined `sched_ras_entity`
    - ***line 153*** Declared struct `ras_rq`
- `include/linux/init_task.h`
    - ***line 173*** Initiate `ras_rq`

- *__line 177__* Initiate `wcounts`
- *__line 178__* Initiate `prev_wcounts`

- `/kernel/sched/sched.h`
    - *__line 82__* *Declare* struct ras_rq
    - *__line 313__* Define struct ras_rq
    - *__line 389__* Declare struct ras_rq ras
    - externs some variables accordingly
- /kernel/sched/core.c
    - *__line 1726__* init `run_list` of `sched_ras_entity` in `__sched_fork`
    - *__line 4178__* set the `sched_class` for a task with policy `SCHED_RAS` in `__setscheduler`
    - *__line 4227__* set `SCHED_RAS` as valid in `sched_setscheduler`
    - *__line 7150__* init_ras_rq
- /kernel/sched/Makefile
    - add ras.o as object file

3. Test result

3.1 Page tracing

In the test file, I forked three processes, each visiting different ranges of memory to test page tracing for the process. The father process will wait until its two child process exit.

```
Start memory trace testing program!
find memory accessed!
set memory read write!
Task pid: 1012,memory[0] = 4
find memory accessed!
set memory read write!
Task pid: 1012,memory[1] = 5
Task pid : 1012, Wcount = 2, times = 2
find memory accessed!
set memory read write!
Task pid: 1011,memory[0] = 2
find memory accessed!
set memory read write!
Task pid: 1011,memory[1] = 3
Task pid : 1011, Wcount = 2, times = 2
find memory accessed!
set memory read write!
Task pid: 1010,memory[0] = 0
find memory accessed!
set memory read write!
Task pid: 1010,memory[1] = 1
Task pid : 1010, Wcount = 2, times = 2
```

3.2 RAS

3.2.1 set_scheuler

Test file set_scheduler.c calls the system call `sched_setscheduler` and `sched_getscheduler` to get the scheduler of the current task as well as changing the policy of the given task.

```
root@generic:/data/misc # ./set_sched
Please input the Choice of Scheduling algorithms (0 - NORMAL, 1 - FIFO, 2 - RR, 6 - RAS) :
6
Current sched algorithm is SCHED_RAS
Please input the id (PID) of the testprocess :
1189
Wcount for this process is : 10
set process's priority(1-99) :
9
Pre scheduler : SCHED_NORMAL
cur scheduler : SCHED_RAS
root@generic:/data/misc #
```

3.2.2 multi_process test

Test file prob2_test.c creates the input number of processes, each making different numbers of memory writing related to its process number.

The ith process makes i^3 memory writing to make the variance of their wcounts bigger. The last process makes $(i-1)^3$ writing.

E.g. With input 12, the total wcounts on the rq would be $1^3+2^3+\ldots+11^3+11^3=5687$ and the average would be $5687/12=473$.

```
avg: 473, wcounts: 512, prob : 5, pid: 1254, total: 5687, nr: 12, ratio: 1,
time_slice : 5, pid: 1254
new time_slice: 5 pid : 1254
remaining time_slice: 4 pid : 1253
remaining time_slice: 3 pid : 1253
remaining time_slice: 2 pid : 1253
remaining time_slice: 1 pid : 1253
avg: 473, wcounts: 343, prob : 5, pid: 1253, total: 5687, nr: 12, ratio: -1,
time_slice : 5, pid: 1253
new time_slice: 5 pid : 1253
remaining time_slice: 4 pid : 1252
remaining time_slice: 3 pid : 1252
remaining time_slice: 2 pid : 1252
remaining time_slice: 1 pid : 1252
avg: 473, wcounts: 216, prob : 5, pid: 1252, total: 5687, nr: 12, ratio: -2,
time_slice : 5, pid: 1252
new time_slice: 5 pid : 1252
remaining time_slice: 2 pid : 1246
remaining time_slice: 1 pid : 1246
avg: 473, wcounts: 1331, prob : 7, pid: 1246, total: 5687, nr: 12, ratio: 2,
time_slice : 3, pid: 1246
new time_slice: 3 pid : 1246
remaining time_slice: 8 pid : 1249
remaining time_slice: 7 pid : 1249
remaining time_slice: 6 pid : 1249
remaining time_slice: 5 pid : 1249
remaining time_slice: 4 pid : 1249
remaining time_slice: 3 pid : 1249
remaining time_slice: 2 pid : 1249
remaining time_slice: 1 pid : 1249
avg: 473, wcounts: 27, prob : 1, pid: 1249, total: 5687, nr: 12, ratio: -17,
time_slice : 9, pid: 1249
new time_slice: 9 pid : 1249
remaining time_slice: 8 pid : 1248
remaining time_slice: 7 pid : 1248
remaining time_slice: 6 pid : 1248
```

```
remaining time_slice: 1 pid : 1253
avg: 473, wcounts: 343, prob : 5, pid: 1253, total: 5687, nr: 12, ratio: -1,
time_slice : 5, pid: 1253
new time_slice: 5 pid : 1253
remaining time_slice: 4 pid : 1252
remaining time_slice: 3 pid : 1252
remaining time_slice: 2 pid : 1252
remaining time_slice: 1 pid : 1252
avg: 473, wcounts: 216, prob : 5, pid: 1252, total: 5687, nr: 12, ratio: -2,
time_slice : 5, pid: 1252
new time_slice: 5 pid : 1252
remaining time_slice: 2 pid : 1246
remaining time_slice: 1 pid : 1246
avg: 473, wcounts: 1331, prob : 7, pid: 1246, total: 5687, nr: 12, ratio: 2,
time_slice : 3, pid: 1246
new time_slice: 3 pid : 1246
remaining time_slice: 8 pid : 1249
remaining time_slice: 7 pid : 1249
remaining time_slice: 6 pid : 1249
remaining time_slice: 5 pid : 1249
remaining time_slice: 4 pid : 1249
remaining time_slice: 3 pid : 1249
remaining time_slice: 2 pid : 1249
remaining time_slice: 1 pid : 1249
avg: 473, wcounts: 27, prob : 1, pid: 1249, total: 5687, nr: 12, ratio: -17,
time_slice : 9, pid: 1249
new time_slice: 9 pid : 1249
remaining time_slice: 8 pid : 1248
remaining time_slice: 7 pid : 1248
remaining time_slice: 6 pid : 1248
remaining time_slice: 5 pid : 1248
remaining time_slice: 4 pid : 1248
remaining time_slice: 3 pid : 1248
remaining time_slice: 2 pid : 1248
remaining time_slice: 1 pid : 1248
```

```
avg: 473, wcounts: 1331, prob : 7, pid: 1257, total: 5687, nr: 12, ratio: 2,
time_slice : 3, pid: 1257
new time_slice: 3 pid : 1257
remaining time_slice: 2 pid : 1256
remaining time_slice: 1 pid : 1256
avg: 473, wcounts: 1000, prob : 7, pid: 1256, total: 5687, nr: 12, ratio: 2,
time_slice : 3, pid: 1256
new time_slice: 3 pid : 1256
remaining time_slice: 4 pid : 1255
remaining time_slice: 3 pid : 1255
remaining time_slice: 2 pid : 1255
remaining time_slice: 1 pid : 1255
avg: 473, wcounts: 729, prob : 5, pid: 1255, total: 5687, nr: 12, ratio: 1,
time_slice : 5, pid: 1255
new time_slice: 5 pid : 1255
remaining time_slice: 4 pid : 1254
remaining time_slice: 3 pid : 1254
remaining time_slice: 2 pid : 1254
remaining time_slice: 1 pid : 1254
avg: 473, wcounts: 512, prob : 5, pid: 1254, total: 5687, nr: 12, ratio: 1,
time_slice : 5, pid: 1254
new time_slice: 5 pid : 1254
remaining time_slice: 4 pid : 1253
remaining time_slice: 3 pid : 1253
remaining time_slice: 2 pid : 1253
```

It can be easily seen that tasks with different `wcounts` has different `prob`, `ratio` and are allocated different time slices accordingly. It can also be seen that the task would be put back when the time slice is used up.

After all the process have finished, the father process will print a message and gracefully exit.

```
Process 2 returns
78 / 100, pid: 1391
79 / 100, pid: 1391
80 / 100, pid: 1391
81 / 100, pid: 1391
82 / 100, pid: 1391
83 / 100, pid: 1391
84 / 100, pid: 1391
85 / 100, pid: 1391
86 / 100, pid: 1391
87 / 100, pid: 1391
88 / 100, pid: 1391
89 / 100, pid: 1391
90 / 100, pid: 1391
91 / 100, pid: 1391
92 / 100, pid: 1391
93 / 100, pid: 1391
94 / 100, pid: 1391
95 / 100, pid: 1391
96 / 100, pid: 1391
97 / 100, pid: 1391
98 / 100, pid: 1391
99 / 100, pid: 1391
100 / 100, pid: 1391
father process exits
root@generic:/data/misc #
```

⚠️ negative ratio means that it is a fraction. E.g. ratio is -2 means that the ratio is $\frac{1}{2}$.

4. Extra work : priority and preemtion

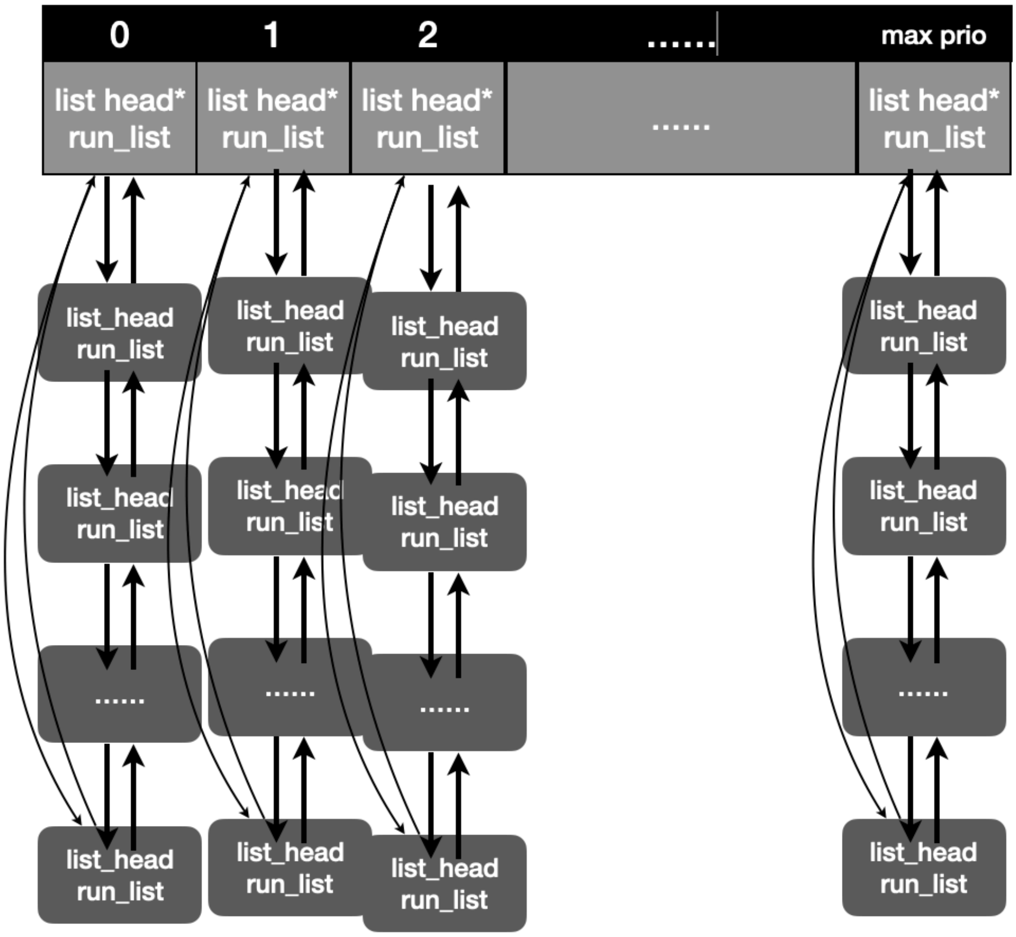
4.1 Multi-level weight round robin

The basic RAS treats each task equally. I introduced priority for SCHED_RAS. The priority is presented as an integer in the range of $[0, 9]$, with 0 being the highest priority. The multi-level weight round robin follows the rules below.

- The scheduler performs weighted round robin for the tasks with the highest priority currently.
- Tasks with lower priorities can only run after the tasks with higher priorities finish.
- Tasks enqueued to the queue can preempt the currently running task with priority.

4.2 Analysis and Implementation

4.2.1. Data struction



Referring to `rt.c`, I declared an array of `list_head*` to store tasks with different priorities as double linked lists, which means that there are in total $max\_prio - 1$ lists. Also, a bitmap is declared to record whether there are tasks with certain priority.

RR can be performed on the double linked list with the highest priority.

4.2.2 Functions revised

`enqueue_task_ras` enqueues the task to the queue of its priority and set the bitmap for its priority.

`dequeue_task_ras` dequeues the task from the queue of its priority and delete the bitmap for its priority if it is the last task on the queue.

`pick_next_task_ras` finds the first bit of the bitmap, which the represents the highest priority on the `ras_rq` and pick the task at the head.

`check_preempt_curr` checks if the given task has higher priority than the running task and set TIF_NEED_RESCHED for the current task if so.

`prio_changed_ras` is invoked when the given task's priority is changed. It checks if the given task has higher priority than the running task and set TIF_NEED_RESCHED for the current task if so.

`switched_to_ras` is invoked when the given task is switched to policy SCHED_RAS. It checks if the given task has higher priority than the running task and set TIF_NEED_RESCHED for the current task if so.

4.2.3 Test result

```
avg: 10, wcounts: 10, prob : 5, pid: 311, total: 10, nr: 1, ratio: 1,
time_slice : 5, pid: 311
new time_slice: 5 pid : 311
remaining time_slice: 4 pid : 311
remaining time_slice: 3 pid : 311
remaining time_slice: 2 pid : 311
set task 347 with prio 0
on_rq :1, curr and p : 0, p->prio : 0, rq->curr->prio : 0
remaining time_slice: 9 pid : 347
remaining time_slice: 8 pid : 347
remaining time_slice: 7 pid : 347
remaining time_slice: 6 pid : 347
remaining time_slice: 5 pid : 347
remaining time_slice: 4 pid : 347
remaining time_slice: 3 pid : 347
remaining time_slice: 2 pid : 347
remaining time_slice: 1 pid : 347
new time_slice: 10 pid : 347
remaining time_slice: 9 pid : 347
remaining time_slice: 8 pid : 347
remaining time_slice: 7 pid : 347
remaining time_slice: 6 pid : 347
remaining time_slice: 5 pid : 347
remaining time_slice: 4 pid : 347
remaining time_slice: 3 pid : 347
remaining time_slice: 2 pid : 347
remaining time_slice: 1 pid : 347
new time_slice: 10 pid : 347
```

It can be seen that task 347 with priority 0 (which is higher in the kernel) preempts task 311 with priority 9. Then task 311 cannot be executed unless task 347 finishes.

```
new time_slice: 5 pid : 1410
remaining time_slice: 4 pid : 1409
remaining time_slice: 3 pid : 1409
remaining time_slice: 2 pid : 1409
remaining time_slice: 1 pid : 1409
avg: 10, wcounts: 10, prob : 5, pid: 1409, total: 30, nr: 3, ratio: 1,
time_slice : 5, pid: 1409
new time_slice: 5 pid : 1409
remaining time_slice: 4 pid : 1411
remaining time_slice: 3 pid : 1411
remaining time_slice: 2 pid : 1411
remaining time_slice: 1 pid : 1411
avg: 10, wcounts: 10, prob : 5, pid: 1411, total: 30, nr: 3, ratio: 1,
time_slice : 5, pid: 1411
new time_slice: 5 pid : 1411
remaining time_slice: 4 pid : 1410
remaining time_slice: 3 pid : 1410
remaining time_slice: 2 pid : 1410
remaining time_slice: 1 pid : 1410
avg: 10, wcounts: 10, prob : 5, pid: 1410, total: 30, nr: 3, ratio: 1,
```

```
new time_slice: 10 pid : 1415
remaining time_slice: 9 pid : 1413
remaining time_slice: 8 pid : 1413
remaining time_slice: 7 pid : 1413
remaining time_slice: 6 pid : 1413
remaining time_slice: 5 pid : 1413
remaining time_slice: 4 pid : 1413
remaining time_slice: 3 pid : 1413
remaining time_slice: 2 pid : 1413
remaining time_slice: 1 pid : 1413
new time_slice: 10 pid : 1413
remaining time_slice: 9 pid : 1414
remaining time_slice: 8 pid : 1414
remaining time_slice: 7 pid : 1414
remaining time_slice: 6 pid : 1414
remaining time_slice: 5 pid : 1414
remaining time_slice: 4 pid : 1414
remaining time_slice: 3 pid : 1414
remaining time_slice: 2 pid : 1414
remaining time_slice: 1 pid : 1414
new time_slice: 10 pid : 1414
remaining time_slice: 9 pid : 1415
remaining time_slice: 8 pid : 1415
remaining time_slice: 7 pid : 1415
remaining time_slice: 6 pid : 1415
remaining time_slice: 5 pid : 1415
remaining time_slice: 4 pid : 1415
remaining time_slice: 3 pid : 1415
```

Another test forks three processes with the lowest priority 9 at first. When the three processes are running, another three processes with the highest priority starts running. The three processes will be preempted and only the newly forked three processes will be running.