Project 2: Android Memory and Scheduler

Fan Wu

Department of Computer Science and Engineering

Shanghai Jiao Tong University

Spring 2022

Objectives

1. Compile the Android kernel
2. Familiarize Android memory
3. Tracing memory for tasks
4. Implement a Race-Averse Scheduler

Recompile the Kernel

Compile the Linux Kernel

- Make sure that your environment variables are correct.

```
export JAVA_HOME=/usr/lib/jdk1.8.0_73
export JRE_HOME=/usr/lib/jdk1.8.0_73/jre
export CLASSPATH=.:$CLASSPATH:$JAVA_HOME/lib:$JRE_HOME/lib
export PATH=$PATH:$JAVA_HOME/bin:$JRE_HOME/bin
export PATH=~/Kit/android-sdk-linux/platform-tools:$PATH
export PATH=~/Kit/android-sdk-linux/tools:$PATH
export PATH=~/Kit/android-ndk-linux:$PATH
export PATH=~/Kit/android-ndk-linux/toolchains/arm-linux-androideabi-4.9/prebuilt/linux-x86_64/bin:$PATH
```

Compile the Linux Kernel (cont.)

- Modify Makefile in the kernel
  - Change
    - ARCH ?= $(SUBARCH)
    - CROSS_COMPILE ?=
  - To

```
export KBUILD_BUILDHOST := $(SUBARCH)
ARCH            ?= arm
CROSS_COMPILE   ?= arm-linux-androideabi-

# Architecture as present in compile.h
```

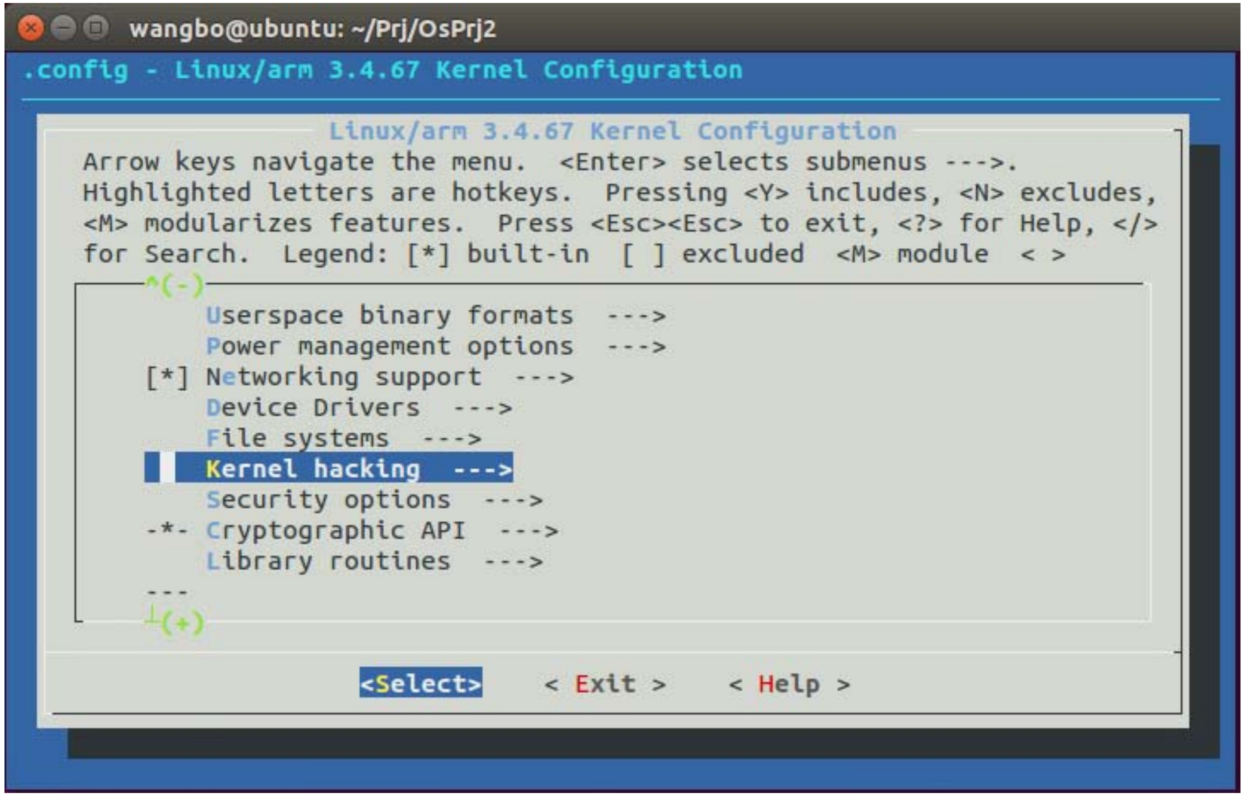
Compile the Linux Kernel (cont.)

- Execute the following command:

```
wangbo@ubuntu:~/Prj/OsPrj2$ make goldfish_armv7_defconfig
#
# configuration written to .config
#
wangbo@ubuntu:~/Prj/OsPrj2$ sudo apt-get install ncurses-dev
wangbo@ubuntu:~/Prj/OsPrj2$ make menuconfig
```

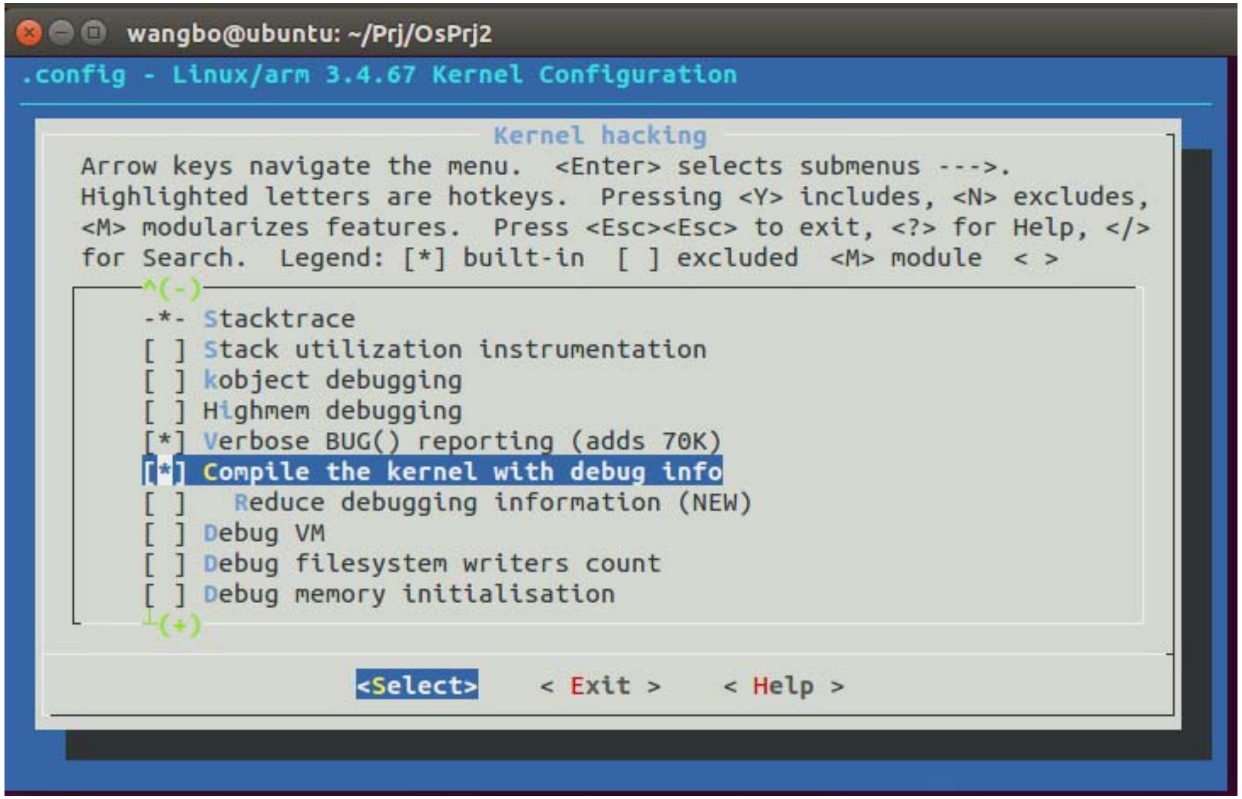
Compile the Linux Kernel (cont.)

- Then you can see a GUI configuration dialog:



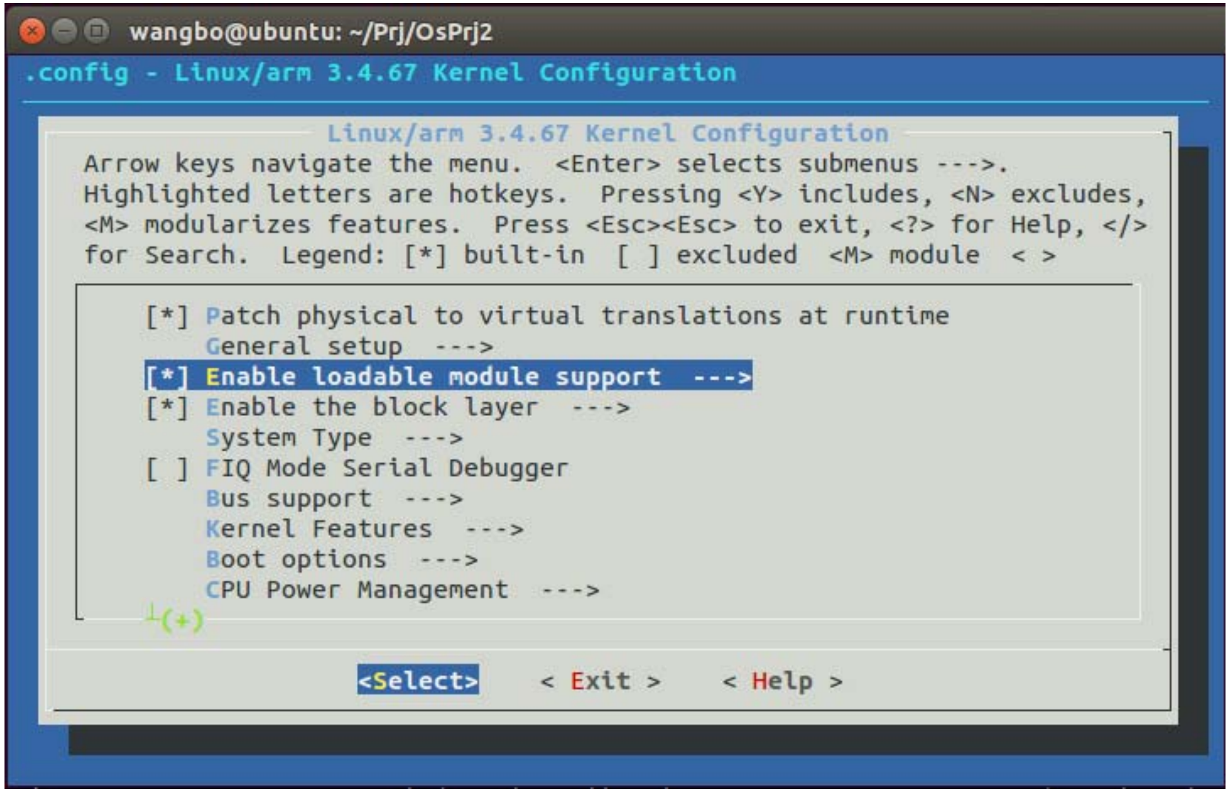
Compile the Linux Kernel (cont.)

- Open the *Compile the kernel with debug info* in *Kernel hacking*:



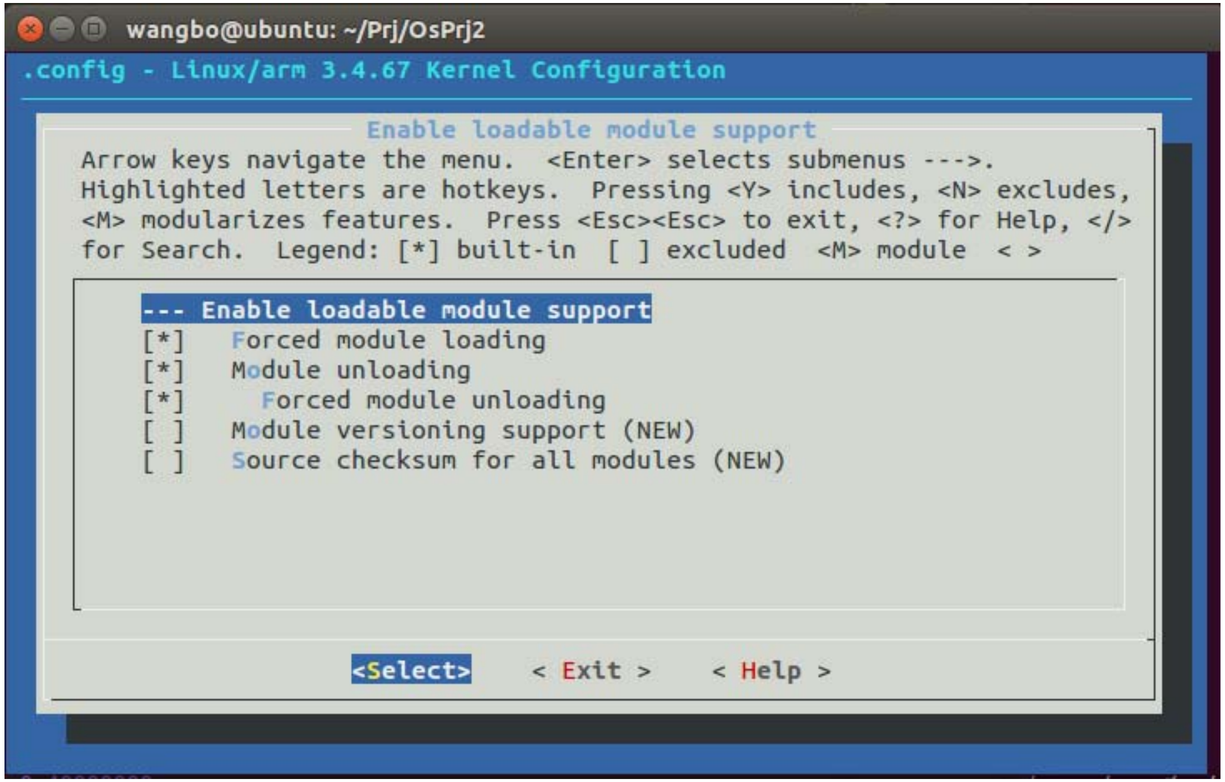
Compile the Linux Kernel (cont.)

- *Enable loadable module support* with Forced module loading, Module unloading and Forced module unloading in it:



Compile the Linux Kernel (cont.)

- *Enable loadable module support* with Forced module loading, Module unloading and Forced module unloading in it:



Compile the Linux Kernel (cont.)

- Compile it
  - The number of -j* depends on the number of cores of your system.

```
wangbo@ubuntu:~/Prj/OsPrj2$ make -j4
  SYSMAP  System.map
  SYSMAP  .tmp_System.map
  OBJCOPY arch/arm/boot/Image
  Kernel: arch/arm/boot/Image is ready
  GZIP    arch/arm/boot/compressed/piggy.gzip
  AS      arch/arm/boot/compressed/piggy.gzip.o
  LD      arch/arm/boot/compressed/vmlinux
  OBJCOPY arch/arm/boot/zImage
  Kernel: arch/arm/boot/zImage is ready
wangbo@ubuntu:~/Prj/OsPrj2$
```

Page Access Tracing

√ 20分.

Problem

- Tracing memory access for tasks (or say processes) is useful in Android systems.
- In Linux, 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.
- In this problem, you are required to 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.

Problem

- **System call 361** should tell the kernel to start tracing page writes for the process pid.
- void* sys_start_trace(pid_t pid);

- **System call 362** should tell the kernel to stop tracing page writes of process pid.
- void* sys_stop_trace(pid_t pid);

- **System call 363** should return the page writes frequency wcounts of the process pid and print it.
- void* sys_get_trace(pid_t pid, int *wcounts);

Hints

- Hints on implementation:
  - To trace the write operation of a task, the recommended way is to set the page tables as if memory writes fails, such that you can increase the task's write counter in the page fault handler.
  - You may find more information in the system call mprotect in /mm/mprotect.c and page fault function in /arch/arm/mm/fault.c

Hints

- Hints on implementation:
  - The page fault handler will send a fault signal to the process. If the process want to write into the protected page, you can temporally deactivate the fault signal for the process.
  - You need to implement segv_handler function and use the struct sigaction to handle the page fault signal.
  - In segv_handler function, you can cancel the protection for pages. And after finishing the access, you need to set protection again.

```c
void segv_handler(int signal_number)
{
    ... // cancel mprotect
}

int main()
{
    ...
    struct sigaction sa;
    memset(&sa, 0, sizeof(sa));
    sa.sa_handler = &segv_handler;
    sigaction(SIGSEGV, &sa, NULL);
    ...

    //mprotect before each memory access
}
```

Implementation Details

- Files MAY NEED Modification
  - /include/linux/sched.h
  - /arch/arm/mm/fault.c
- To implement three new system calls, you need to write three *.c files and make them.
  - start_trace.c
  - stop_trace.c
  - get_trace.c
- You need to create test files (your processes) according to the framework described in the previous slide.

Implementation Details

/include/linux/sched.h

- In this file, you need to:
  - You should modify task_struct : add a integer variable wcounts to record the write times and a boolean variable trace_flag to determine when to start tracin.

```
struct task_struct {
    volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
    void *stack;
    atomic_t usage;
    unsigned int flags; /* per process flags, defined below */
    unsigned int ptrace;

    //my race_proc
```

Implementation Details

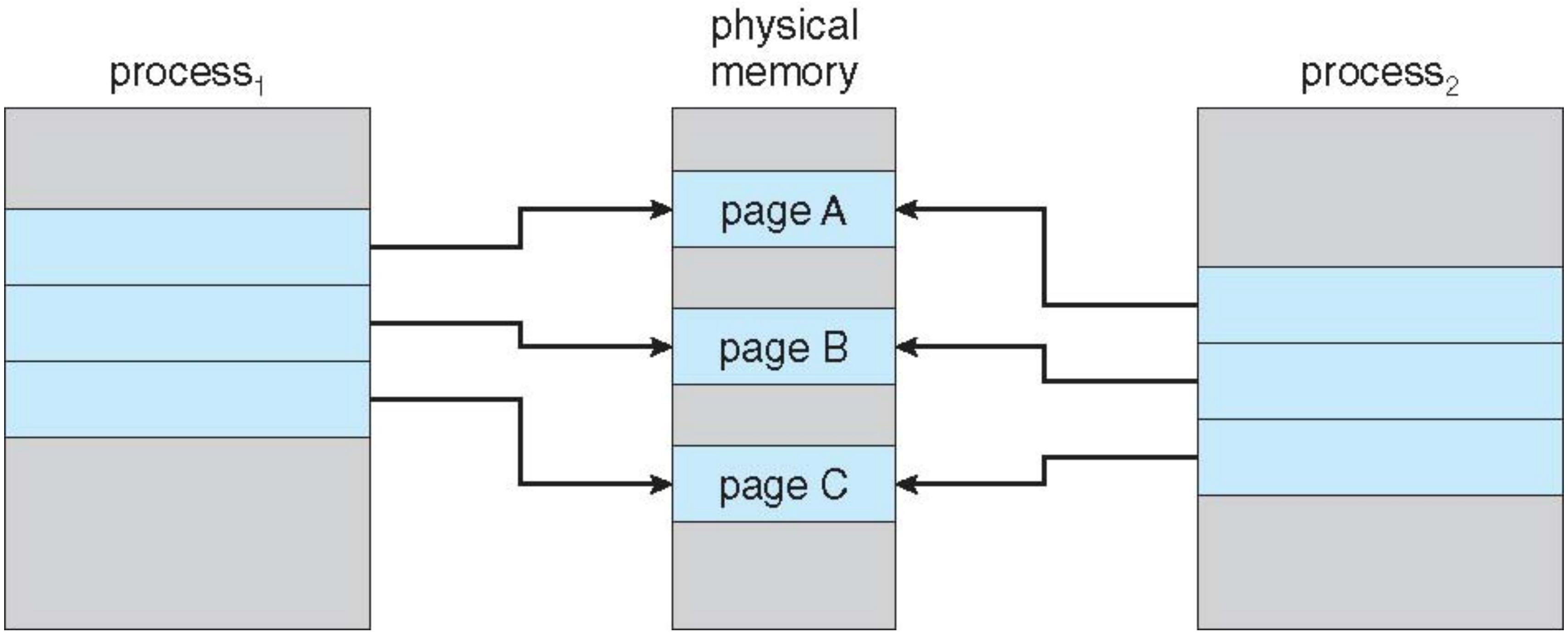
/arch/arm/mm/fault.c

- In this file, you need to:
  - You should modify do_page_fault and access_error : increase wcount when the kernel catches a page write fault.

```c
do_page_fault(unsigned long addr, unsigned int fsr, struct pt_regs *regs)
{
    struct task_struct *tsk;
    struct mm_struct *mm;
    int fault, sig, code;
    int write = fsr & FSR_WRITE;
    unsigned int flags = FAULT_FLAG_ALLOW_RETRY | FAULT_FLAG_KILLABLE |
                (write ? FAULT_FLAG_WRITE : 0);
```

Race-Averse Scheduler

Page Racing between Processes



Problem

- When the Linux kernel scheduler chooses a task, it considers a variety of factors.
- The likelihood that a task may race with one of the currently running tasks is not considered.
- In this problem, you are required to implement a **Race-Averse Scheduling** (RAS) algorithm to the Linux.
- The scheduling should be a **weighted round robin** style scheduling according to **race probabilities** of each tasks.

Problem

- Weighted Round Robin (WRR)
  - Round-robin scheduling treats all tasks equally, but there are times when it is desirable to give some tasks preference over the others.
  - WRR assigned more milliseconds as a time slice for tasks with lower race probabilities. (In our problem, it is inversely proportional to the race probabilities from 10ms to 100ms)
  - The blog relevant to linux kernel scheduler could be helpful:
    - https://helix979.github.io/jkoo/post/os-scheduler/

Problem

- Race Probabilities
  - The race probability of a task is represented as an integer in the rage of [0, 10). The higher the integer is, the more likely that the tasks may race with others.
  - To simplify the assignment, it is assumed 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.

Implementation Details

- Files MAY NEED Modification
  - /arch/arm/configs/goldfish_armv7_defconfig
  - /include/linux/sched.h
  - /kernel/sched/core.c
  - /kernel/sched/sched.h

- To implement RAS scheduler, you need to create a new class in the directory /kernel/sched/, that is
  - /kernel/sched/ras.c

IMPORTANT: If you feel confused on what to do in ras.c, read /kernel/sched/rt.c carefully. In rt.c, RR and FIFO are well implemented.

Implementation Details

/arch/arm/configs/goldfish_armv7_defconfig

- In this file, you need to add a new line as follows:

```
CONFIG_RT_GROUP_SCHED=y
CONFIG_RAS_GROUP_SCHED=y
CONFIG_BLK_DEV_INITRD=y
```

Implementation Details

/include/linux/sched.h

In this file, you need to:

- Define SCHED_RAS (Refer to SCHED_RR, about Line 40). The value of SCHED_RAS should be 6.
- Define sched_ras_entity (Refer to sched_rt_entity, about Line 1250)
- Define RAS_TIMESLICE. (Refer to RR_TIMESLICE, about Line 1270)
- Add a sched_was_entity varaible to task_struct (About Line 1310).
- Declare a ras_rq struct. (Refer to struct cfs_rq, about Line 150)
- Maybe a little more to be revised. It depends on your implementation.

Implementation Details

/kernel/sched/sched.h （path is different from last page）

In this file, you need to:

- Declare a ras_rq struct. (Refer to struct rt_rq, about Line 80)

```
81  struct rt_rq;
82  struct ras_rq;
```

- Define a new struct ras_rq (Refer to rt_rq, about Line 290).
- Add a ras_rq variable to struct rq (About line 400) and similarly add a list_head variable as the figure shows.

```
428 #ifdef CONFIG_RT_GROUP_SCHED
429     struct list_head leaf_rt_rq_list;
430 #endif
431
432 #ifdef CONFIG_RAS_GROUP_SCHED
433     struct list_head leaf_ras_rq_list;
434 #endif
```

- Declare some extern variables and functions（*）. (You can refer to extern var/func of rt in the same file, About Line 190-210, Line 880, Line 900, Line 1170-1180). E.g.

```
898 extern const struct sched_class rt_sched_class;
899 extern const struct sched_class ras_sched_class;
900 extern const struct sched_class fair_sched_class;
901 extern const struct sched_class idle_sched_class;
```

- Maybe a little more to be revised. It depends on your implementation.

Implementation Details

/kernel/sched/core.c

In this file, you need to:

- Revise function: static void __sched_fork(struct task_struct *p)
- Revise function: static void __setscheduler(struct rq *rq, struct task_struct *p, int policy, int prio)
- Revise function: static void __sched_setscheduler(struct task_struct *p, int policy, const struct sched_param *param, bool user)
- Add init_ras_rq(&rq->ras). (Refer to init_rt_rq(), about Line 7230)
- Revise function: static void free_sched_group(struct task_group *tg) and struct task__group *sched_create_group(struct task_group *parent) (about Line 7500-7600)
- Maybe a little more to be revised. It depends on your implementation.

Implementation Details

/kernel/sched/core.c

- To know what to revise, you have to read the code carefully and know what are they implemented for.

- For example, when revising *__sched_setscheduler*, we meet the following code segment:

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

Since we have one more policy RAS now, we should change it to:

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

Implementation Details

/kernel/sched/ras.c

- This is the *major file* in which you write codes. You can refer to rt.c in the same directory to learn how to write ras.c
- Here, we give a framework of ras_sched_class

```c
const struct sched_class ras_sched_class = {
	.next			= &idle_sched_class,		/*Required*/
	.enqueue_task		= enqueue_task_ras,		/*Required*/
	.dequeue_task		= dequeue_task_ras,		/*Required*/
	.yield_task		= yield_task_ras,		/*Required*/

	.check_preempt_curr = check_preempt_curr_ras,	/*Required*/

	.pick_next_task		= pick_next_task_ras,		/*Required*/
	.put_prev_task		= put_prev_task_ras,		/*Required*/

#ifdef CONFIG_SMP
	.select_task_rq		= select_task_rq_ras,		/*Never need impl*/

	.set_cpus_allowed		= set_cpus_allowed_ras,	/*Never need impl*/
	.rq_online			= rq_online_ras,		/*Never need impl*/
	.rq_offline			= rq_offline_ras,		/*Never need impl*/
	.pre_schedule		= pre_schedule_ras,		/*Never need impl*/
	.post_schedule		= post_schedule_ras,		/*Never need impl*/
	.task_woken		= task_woken_ras,		/*Never need impl*/
	.switched_from		= switched_from_ras,		/*Never need impl*/
#endif

	.set_curr_task			= set_curr_task_ras,	/*Required*/
	.task_tick		= task_tick_ras,		/*Required*/

	.get_rr_interval	= get_rr_interval_ras,		/*Required*/

	.prio_changed		= prio_changed_ras,		/*Never need impl*/
	.switched_to		= switched_to_ras,		/*Required*/
};
```

Implementation Details

/kernel/sched/ras.c

For functions labeled “Required”, you need to implement it in ras.c

```
static void switched_to_ras(struct rq *rq, struct task_struct *p)⋯
```

For functions labeled “Never need impl”, you can just put them dummy

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

Implementation Details

/kernel/sched/ras.c

- Remember that in ras.c, when you want to allocate time slice to a task with RAS as its policy, you should judge whether it is a foreground task or background task, and allocate corresponding time slice to it.
- The blog relevant to *linux kernel scheduler* could be helpful to you：
  - https://helix979.github.io/jkoo/post/os-scheduler/

Implementation Details

To put your */kernel/sched/ras.c* into effect:

- You need to revise the Makefile in */kernel/sched* like this:

```
ifdef CONFIG_FUNCTION_TRACER
CFLAGS_REMOVE_clock.o = -pg
endif

ifneq ($(CONFIG_SCHED_OMIT_FRAME_POINTER),y)
# According to Alan Modra <alan@linuxcare.com.au>, the -fno-omit-frame-pointer is
# needed for x86 only.  Why this used to be enabled for all architectures is beyond
# me.  I suspect most platforms don't need this, but until we know that for sure
# I turn this off for IA-64 only.  Andreas Schwab says it's also needed on m68k
# to get a correct value for the wait-channel (WCHAN in ps). --davidm
CFLAGS_core.o := $(PROFILING) -fno-omit-frame-pointer
endif

obj-y += core.o clock.o idle_task.o fair.o rt.o stop_task.o ras.o
obj-$(CONFIG_SMP) += cpupri.o
obj-$(CONFIG_SCHED_AUTOGROUP) += auto_group.o
obj-$(CONFIG_SCHEDSTATS) += stats.o
obj-$(CONFIG_SCHED_DEBUG) += debug.o
```

What’s next

What to show

Basic

- Add some printk(“”) in fault.c, mprotect.c, system call files or some other places to proves page access tracing works normally.
- Write a test file, which can test three system calls you implemented.
  - Trace page access of a task and give out the result.
  - You should repeat the test for different virtual address range and different time intervals.

What to show

Basic

- Add some printk(“”) in ras.c or some other places which proves there is a task using RAS as a policy.
- Write a test file, which can change the scheduler in user space.
  - It is recommend that you should create multiple tasks, and each task requests different sizes of memory to form a race relationship.
  - Change the apk’s scheduler to RAS, and give out some information (pid, name, timeslice, and some others you like).
- To check the pid of *test_file* in Android shell, you may use PS -P

What to show

Basic

Print information when switching scheduler:

```
Please input the Choice of Scheduling algorithms (0-NORMAL, 1-FIFO, 2-RR, 6-RAS) : 6
Current scheduling algorithm is SCHED_RAS
Please input the id (PID) of the testprocess : 248
Wcount for this task : 2
Set process's priority (1-99): 0
current scheduler's priority is : 0
pre scheduler : SCHED_NORMAL
cur scheduler : SCHED_RAS
Switch finish.
```

OR:

```
Please input the Choice of Scheduling algorithms (0-NORMAL, 1-FIFO, 2-RR, 6-RAS) : 6
Current scheduling algorithm is SCHED_RAS
Please input the id (PID) of the testprocess : 131
Start trace for task 131
Set process's priority (1-99): 0
current scheduler's priority is : 0
pre scheduler : SCHED_NORMAL
cur scheduler : SCHED_RAS
Switch finish.
```

What to show

Bonus (10 points in Final Score):

- Any extended ideas can be considered into the bonus!
- Here are some of the ideas we provide, I hope you won't be limited to these:
  - Can you come up with a method to compare the performance of RR, FIFO, and RAS?
  - Can you use different ways to compute race probabilities? For example, identifying the accesses to actually shared pages by checking the tasks' page tables.
  - Can you build RAS in a multi-cpu architecture and implement load balance?

Hints

- To change the scheduler, study several functions with SYSCALL in their names. For example, SYSCALL_DEFINE3 (sched_setscheduler…….). Try to use these system calls in user space.

- You can firstly change scheduler to RR or FIFO to see if your testing file is logically correct.

- You can take full use of printk and the functions defined in /kernel/sched/debug.c for debugging.

- Helpful files:
  - /kernel/sched/core.c and /kernel/sched/sched.h tells you how the Linux scheduler works.
  - /kernel/sched/rt.c tells you how to create a scheduler.
  - /include/linux/sched.h concerns run-state processes.

Be patient enough to read them carefully!

Report

- Explain how you trace page write of a task.
- Explain how you compute race probabilities.
- Explain how your RAS works.
- Show your results of test runs.
- Any further analysis is welcome.

Something to Specify

- To give you an **overview** of this project:
  - You need to write 500 lines (more or less) of codes in:
    - /kernel/sched/ras.c
  - You need to revise the following files to put *ras.c* into effect:
    - /arch/arm/configs/goldfish_armv7_defconfig
    - /include/linux/sched.h
    - /kernel/sched/sched.h
    - /kernel/sched/core.c
    - /arch/arm/mm/fault.c
    - */kernel/sched/Makefile*
  - You need to create test scripts to test page access tracing and RAS scheduling, respectively, to make sure you get at least part of the scores.

Distribution of Credits

- Problem 1: 5 points
- Problem 2: 10 points
- Problem 3: 15 points
- Technical Report: 5 points
- Extra Bonus: up to 10 points
- Optional Presentation: up to 10 points

Deadline

Mid-night, May 27, 2022

Demo & Presentation

- Demo:
  - **May 28-29, 2022**. Demo slots will be posted in the WeChat group. Please sign your name in one of the available slots.
- Presentation:
  - You are encouraged to present your design of the project optionally. The presentation will be in the **afternoon of May 29, 2022**.

For Help?

- Teaching Assistant
  - Da Huo
    - Email: sjtuhuoda@sjtu.edu.cn
  - Hang Zeng
    - Email: nidhogg@sjtu.edu.cn

- Some useful website
  - http://www.csdn.net/
  - http://stackoverflow.com/
  - http://developer.android.com/