CS2303 Operating System Projects Spring 2022

Project 2: Android Memory and Scheduler

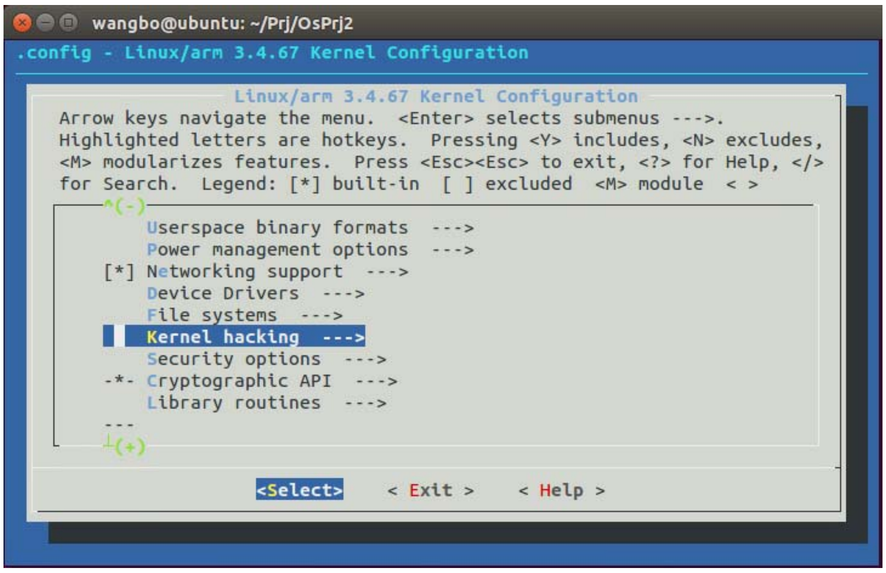
Objectives:

- Compile the Android kernel
- Familiarize Android memory
- Tracing memory for tasks
- Implement Race-Averse Scheduler

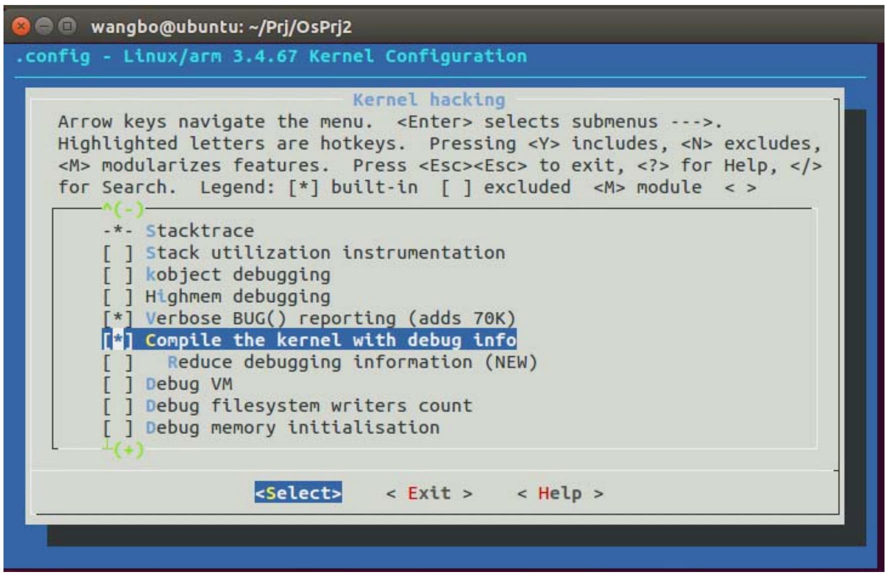
Make sure your system is 64-bits system.

Problem Statement:

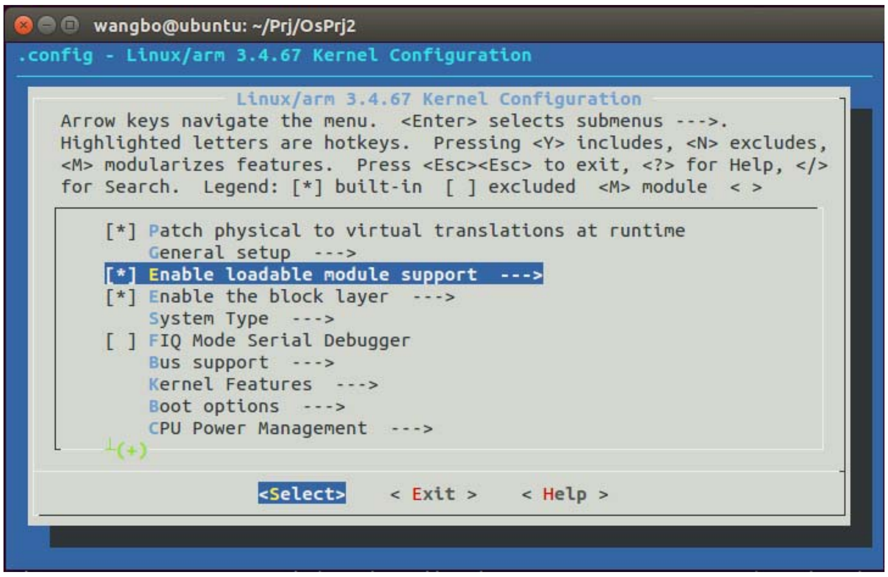
1. **Compile the Linux kernel.**
   a. Make sure that you have added the following path into your environment variable.
   ANDROID_NDK_HOME/toolchains/arm-linux-androideabi-4.6/prebuilt/linux-x86_64/bin
   b. Open Makefile in KERNEL_SOURCE/goldfish/, and find these:
   ```
   ARCH            ?=  $(SUBARCH)
   CROSS_COMPILE   ?=
   ```
   Change it to:
   ```
   ARCH            ?=  arm
   CROSS_COMPILE   ?=  arm-linux-androideabi-
   ```
   Save it and exit.
   c. Execute the following command in terminal to set compiling configration:
   ```
   make goldfish_armv7_defconfig
   ```
   d. Modify compiling configration:
   ```
   sudo apt-get install ncurses-dev
   make menuconfig
   ```
   Then you can see a GUI config.



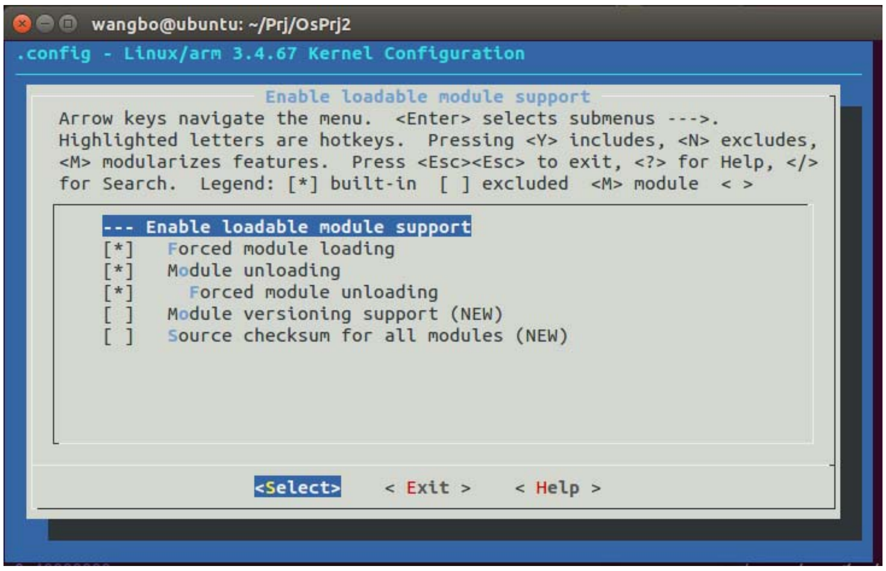
```
wangbo@ubuntu: ~/Prj/OsPrj2
.config - Linux/arm 3.4.67 Kernel Configuration
                 Linux/arm 3.4.67 Kernel Configuration
  Arrow keys navigate the menu.  <Enter> selects submenus --->.
  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes,
  <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </>
  for Search.  Legend: [*] built-in  [ ] excluded  <M> module  < >
      ^(-)
          Userspace binary formats  --->
          Power management options  --->
      [*] Networking support  --->
          Device Drivers  --->
          File systems  --->
          Kernel hacking  --->
          Security options  --->
      -*- Cryptographic API  --->
          Library routines  --->
      ---
      v(+)

              <Select>    < Exit >    < Help >
```



```
wangbo@ubuntu: ~/Prj/OsPrj2
.config - Linux/arm 3.4.67 Kernel Configuration
                         Kernel hacking
  Arrow keys navigate the menu.  <Enter> selects submenus --->.
  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes,
  <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </>
  for Search.  Legend: [*] built-in  [ ] excluded  <M> module  < >
      ^(-)
      -*- Stacktrace
      [ ] Stack utilization instrumentation
      [ ] kobject debugging
      [ ] Highmem debugging
      [*] Verbose BUG() reporting (adds 70K)
      [*] Compile the kernel with debug info
      [ ]   Reduce debugging information (NEW)
      [ ] Debug VM
      [ ] Debug filesystem writers count
      [ ] Debug memory initialisation
      └(+)

                <Select>    < Exit >    < Help >
```



```
wangbo@ubuntu: ~/Prj/OsPrj2
.config - Linux/arm 3.4.67 Kernel Configuration
                Linux/arm 3.4.67 Kernel Configuration
  Arrow keys navigate the menu.  <Enter> selects submenus --->.
  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes,
  <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </>
  for Search.  Legend: [*] built-in  [ ] excluded  <M> module  < >

      [*] Patch physical to virtual translations at runtime
          General setup  --->
      [*] Enable loadable module support  --->
      [*] Enable the block layer  --->
          System Type  --->
      [ ] FIQ Mode Serial Debugger
          Bus support  --->
          Kernel Features  --->
          Boot options  --->
          CPU Power Management  --->
      └(+)

                <Select>    < Exit >    < Help >
```



```
wangbo@ubuntu: ~/Prj/OsPrj2
.config - Linux/arm 3.4.67 Kernel Configuration
                  Enable loadable module support
  Arrow keys navigate the menu.  <Enter> selects submenus --->.
  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes,
  <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </>
  for Search.  Legend: [*] built-in  [ ] excluded  <M> module  < >

      --- Enable loadable module support
      [*]   Forced module loading
      [*]   Module unloading
      [*]     Forced module unloading
      [ ]   Module versioning support (NEW)
      [ ]   Source checksum for all modules (NEW)

                <Select>    < Exit >    < Help >
```

Open the *Compile the kernel with debug info* in *Kernel hacking* and *Enable loadable module support* with *Forced module loading, Module unloading and Forced module unloading* in it.

Save it and exit.

e. Compile

```
make -j4
```

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

2. Page Access Tracing Mechanism

Tracing memory access for tasks (or say processes) is useful in Android systems. We can use these traces to compute the race probabilities between different tasks. For example, if two tasks access disjoint set of pages all the time, the race probability between the two tasks should be 0. Further more, we can use the race probabilities for better scheduling. For simplicity, we first trace approximately how often a page is accessed by a particular task. 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. Thus, we'll have to track the access frequency by ourselves in software.

In this part, you should implement three system calls to support per-task page access tracing, which can trace the number of times a page is accessed by a particular task. You only have to track page writes and maintain a per-thread write counter for each page. You don't have to worry about page reads.

The first system call 361 should tell the kernel to start tracing page writes to virtual address range [start, start+size) for the current process pid.

```
void* sys_start_trace(pid_t pid, unsigned long start, size_t size);
```

You can implement this syscall in two steps. Step 1, set the flag for tracing. Stpe 2, call mprotect to protect memory for the current process. (See why page protection is needed in Hints.)

The second system call 362 should tell the kernel to stop tracing page writes. Each process can have only one active tracing session at a time. If sys_start_trace is called twice without sys_stop_trace being called in between, the second call to sys_start_trace should return -EINVAL.

```
void* sys_stop_trace(pid_t pid);
```

The third system call 363 should return the frequency the task pid accessing a page and print it.

```
void* sys_get_trace(pid_t pid, int *wcounts);
```

For this part, you should write a program to test the vm tracing mechanism you implement. as well. Your test program should contain testcases that create a number of processes, each making a number of memory references. You should use your tracing mechanism to trace these references.

Details:

- 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 traceing.
- Please pay attention to the system call mprotect in /mm/mprotect.c and page fault function in /arch/arm/mm/fault.c

Hints:

- The directory mm/ contains majority of code that deals with paging and virtual memory. You will find the struct mm_struct, struct vm_area_struct in include/linux/mm_types.h and pgd_t, pmd_t, pud_t, pte_t in asm/pgtable.h helpful.
- To trace page P for task T, we first modify T's page table to protect page P so that the next access to P by T will cause a page fault. Then, in the page fault handler, we increase thread T's access counter for page P, and let T continue with the access.
- To learn how to set up protection for a virtual address range, check out syscall sys_mprotect.

3. **Implement A New Task Scheduler - Race-Averse Scheduler (RAS)**

When the Linux kernel scheduler chooses a task, it considers a variety of factors. However, it doesn't consider the likelihood that a task may race with one of the currently running tasks. In this assignment, you will add a **race-averse** scheduling algorithm to the Linux kernel, to let the kernel schedule a task that is least likely to race with any of the currently running tasks.

For this scheduling algorithm to work, the kernel scheduler has to know how likely a task races with others. You need to compute race probabilities between tasks, where a probability 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.

Race probabilities should be calculated by tracing and comparing the frequency of accessing the pages by the tasks, based on the knowledge of wcounts in Problem 2. To simplify the assignment, you can calculate the race probability of a certain task based on its frequency of accessing a given range of virtual addresses (for example, half of the memory), assuming that one's race probability against the other tasks is proportional to the sum of wcounts within the range.

The pseudo code for calculating race probabilities:

```
int overall_race_prob(struct task_struct *p) {
        # prob ∝ wcounts;
        return prob;
    }
```

You should implement a round robin style scheduling according to race probabilities of each tasks, which is called Race-Averse Scheduler (RAS). The timeslice (e.g. 10ms) for each is inversely proportional to the race probabilities, ranging from 1 to 10 (you can also try proportional way, if you are be interested it). SCHED_RAS should be defined in /include/linux/sched.h, its value is 6. Your scheduler should operate alongside the existing Linux scheduler. Define RAS_GROUP in /arch/arm/configs/goldfish_armv7_defconfig. You need also change core.c, Makefile, rt.c, sched.h in /kernel/sched, add a new file ras.c for your implementation (Hint: you can refer to rt.c for reference).

You need to write a test file with fork() and memory reference as well. You are required to print some kernel information to show your process is using RAS scheduler properly.
Details:

- You should modify task_struct : add race probability.
- You should change the order of scheduler. (e.g. RT -> FAIR-> RAS->IDLE)
- Add ras.o to Makefile
- Add struct ras_rq, sched_ras_entity and ras_sched_class

Please refer to rt.c for reference. It will be very helpful.

Implementation Details:
In general, the execution of any of the programs above will is carried out by specifying the executable program name followed by the command line arguments.

1. When using system or library calls, you have to make sure that your program will exit gracefully if the requested call cannot be carried out.
2. One of the dangers of forking processes is leaving unwanted processes active and wasting system time. Make sure each process terminates cleanly when processing is completed. Parent process should wait until the child processes complete, print a message and then quit.
3. Your program should be robust. If any of the calls fail, it should print error message and exit with appropriate error code. Always check for failure when invoking a system or library call.
4. Add some printk(“”) in ras.c or some other places to prove there is a task using RAS as a policy.
5. 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 use different ways to compute race probabilities? For example, identifying the accesses to actually shared pages by checking the tasks’ page tables.
   - ✧ Can you come up with a method to compare the performance of RR, FIFIO, NORMAL and RAS?
   - ✧ Can you build RAS in a multi-cpu architecture and implement load balance?

Material to be submitted:

1. Compress the source code of the programs into **Prj2+StudentID.tar** file. It contains all

*.c, *.h files you have changed in Linux kernel. Use meaningful names for the file so that the contents of the file are obvious. Enclose a README file that lists the files you have submitted along with a one sentence explanation. Call it **Prj2README**.

2. Only internal documentation is needed. Please state clearly the purpose of each program at the start of the program. Add comments to explain your program. (-5 points, if insufficient.)
3. Test runs: Screen captures of the scheduler test to show that your scheduler works. Execution time of every tests.
4. A project report of the project2. In this report, you should show the performance of RAS, show how your system-call work in detail, and Explain how you achieve new algorithms.
5. Due date: **May 27, 2022**, submit on-line **before midnight**. (Hard deadline!)
6. Demo slots: May 28-29, 2022. Demo slots will be posted in the Wechat group. Please sign your name in one of the available slots.
7. You are encouraged to present your design of the project optionally. The presentation date is May 29, 2022. Please pay attention to the Wechat group.