Project 2: Android Memory and Scheduler

1 Recompile the Kernel

To recompile the linux kernel, just do the following instruction step by step:

- Make sure the environment variables in `bashrc` are corret.
- make `goldfish_armv7_defconfig` in `android-kernel/goldfish`
- in the same file folder execute command `sudo apt-get install ncurses-dev` and `make menuconfig`
- modify some settings in the **GUI confguration dialog** referring to *CS2303-Project2.pdf*
- compile the kernel by executing `make -j4`

2 Page Access Tracing

2.1 Problem restatement

Tracing memory access for tasks (or say processes) is useful in Android systems. In this problems, I 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.

2.2 Files to Modification

- `/include/linux/sched.h` -In this file, we 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 tracing.

```
//sched.h struct task_struct
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;
        int wcounts;    /* record the write times */
        bool trace_flag;            /* 1 start trace,0 stop trace */
        //...
 };
```

- `/arch/arm/mm/fault.c` This file have some functions dedicated to handling page faults， like `do_page_fault`, `__do_page_fault` and `access_error`.

- do_page_fault is the core function for handling page missing exceptions. If it is in the interrupt context or preemption forbidden state, it means that the system is running in the atomic context, and the bar will be No_ In the context tag, call __do_kernel_fault function. If this is a kernel thread, also jump to __do_kernel_fault function. More cases are state in the fault.c file. After some exceptional cases, __do_page_fault will be executed to deal with page fault.
- In __do_page_fault, find_vma will find *vma* through the invalid address *addr*. If *vma* can not find , the fuction will return *VM_FAULT_BADMAP*. If find *vma*, call the function access_error to determine whether it has writable or executable permitions. If a page missing exception with a write error occurs, first judge whether the *vma* attribute is writable. If not, return to *VM_FAULT_BADACCESS*.
- So when *fault* = *VM_FAULT_BADACCESS* , it means a write error occurs, and wcounts shuold increase.

```
__do_page_fault(struct mm_struct *mm,  unsigned  long addr,  unsigned  int fsr,
unsigned  int flags,  struct task_struct *tsk)  {
      //...
good_area:
       if (access_error(fsr, vma)) {
               fault = VM_FAULT_BADACCESS;
               if(current->trace_flag) {
                       printk("<0>""do not have a write permit\n");
                       current->wcounts++;
                       printk("<0>""now the wcounts is %d\n",current->wcounts);
               }
               goto out;
       }
       if(current->trace_flag) printk("<0>""have a good vm_area\n");

       return handle_mm_fault(mm, vma, addr & PAGE_MASK, flags);

     //...
```

2.3 Three system calls

- System call 361 void* sys_start_trace(pid_t pid) This module tell the kernel to start tracing page writes for the process pid.
  We use find_get_pid() and pid_task to find the task with corresponding pid. Then we initialize the task's **wcounts = 0** and set **trace_flag = 1** which means start tracing this task.

```
void* sys_start_trace(pid_t pid){
    struct pid *pidk = find_get_pid(pid);       /* get the information of kpid */
    struct task_struct *task = pid_task(pidk,PIDTYPE_PID);    /* get the
information of task */
    task->wcounts = 0;        /* initialize wconuts */
    task->trace_flag = 1;            /* change the trace state to 'start trace' */

    printk(KERN_INFO"START TRACING : %d\n",task->pid);
}
```

- System call 362 `void* sys_stop_trace(pid_t pid)` This module tell the kernel to stop tracing page writes of process pid. Just set **trace_flag = 0** which means stop tracing.

```
void* sys_stop_trace(pid_t pid){
    struct pid *pidk = find_get_pid(pid);       /* get the information of kpid */
    struct task_struct *task = pid_task(pidk,PIDTYPE_PID);    /* get the
information of task */
    task->trace_flag = 0;            /* change the trace state to 'stop trace' */
    task->wcounts = 0;
    printk(KERN_INFO"Stop trace\n");
}
```

-System call 363 `void* sys_get_trace(pid_t pid, int* wcounts)` This module return the page writes frequency **wcounts** of the process pid and print it. We use the same method to find the task of process pid and assign the value of **task->wcounts** to wcounts. We use the function `put_user()` to copy data from kernel space to user space.

```
void* sys_get_trace(pid_t pid, int *wcounts){
        struct pid *pidk = find_get_pid(pid);       /* get the information of kpid
*/
        struct task_struct *task = pid_task(pidk,PIDTYPE_PID);    /* get the
information of task */
        int kwcount = task->wcounts;
        printk(KERN_INFO "GET TRACE !\n");
        //copy data from kernel space to user space
        if(put_user(kwcount, wcounts) == -EFAULT){
            printk(KERN_INFO "Error copying to user!\n");
            return -EFAULT;
        }
        else printk(KERN_INFO "the value of wcounts: %d\n", *wcounts);
}
```

2.4 Test Files

- We use `mprotect` to set page properties. The page fault handler will send a fault signal to the process. If the process want to write into the protected page, we can temporally deactivate the fault signal for the process.
- In `segv_handler` we change the page properties from **read only** to **read write**, and increase the value of times to record **write error**.
- The test files is as follows:

```
void segv_handler(int signal_number)
{
    printf("find memory accessed!\n");
    mprotect(memory, alloc_size, PROT_READ | PROT_WRITE);
```

```
    times++;
    printf("set memory read write!\n");
}
```

```
int main()
{
    int fd;
    struct sigaction sa;
    unsigned long wcount = 0;
    int res = 0;

   printf("Start memory trace testing program!\n");

   syscall(361, getpid());

   /* Init segv_handler to handle SIGSEGV */
   memset(&sa, 0, sizeof(sa));
   sa.sa_handler = &segv_handler;
   sigaction(SIGSEGV, &sa, NULL);

   times = 0;

   /* allocate memory for process, set the memory can only be read */
   alloc_size = 10 * getpagesize();
   fd = open("/dev/zero", O_RDONLY);
   memory = mmap(NULL, alloc_size, PROT_READ, MAP_PRIVATE, fd, 0);
   close(fd);
   /* try to write, will receive a SIGSEGV */
   memory[0] = 0;
   printf("memory[0] = %d\n", memory[0]);

   /* set protection */
   mprotect(memory, alloc_size, PROT_READ);
   /* try to write, will receive a SIGSEGV */
   memory[0] = 1;
   printf("memory[0] = %d\n", memory[0]);

   /*set protection */
   mprotect(memory, alloc_size, PROT_READ);
   /* try to write again, will receive a SIGSEGV */
   memory[1] = 2;
   printf("memory[1] = %d\n", memory[1]);

   mprotect(memory, alloc_size, PROT_READ);
   /* try to write, will receive a SIGSEGV */
   memory[2] = 3;
   printf("memory[2] = %d\n", memory[2]);

   /* Get wcount */
   printf("The initial value of wcounts : %lu\n",wcount);
   res = syscall(363, getpid(), &wcount);
```

```c
    printf("Task pid : %d, Wcount = %lu, times = %d\n", getpid(), wcount, times);
    /* stop trace */
    syscall(362, getpid());
    /* freee */
    munmap(memory, alloc_size);
    return 0;
}
```

2.5 Run Results

The results is :

```
root@generic:/data/misc # ./mem_test
Start memory trace testing program!
find memory accessed!
set memory read write!
memory[0] = 0
find memory accessed!
set memory read write!
memory[0] = 1
The initial value of wcounts : 0
Task pid : 517, Wcount = 2, times = 2
```

```
root@generic:/data/misc # dmesg -c
START TRACING : 517
have a good vm_area
do not have a write permit
now the wcounts is 1
have a good vm_area
have a good vm_area
do not have a write permit
now the wcounts is 2
have a good vm_area
GET TRACE !
the value of wcounts: 2
```

```
root@generic:/data/misc # insmod start_trace.ko
root@generic:/data/misc # insmod get_trace.ko
root@generic:/data/misc # insmod stop_trace.ko
root@generic:/data/misc # ./test1
Start memory trace testing program!
find memory accessed!
set memory read write!
memory[0] = 0
find memory accessed!
set memory read write!
memory[0] = 1
find memory accessed!
set memory read write!
memory[1] = 2
find memory accessed!
set memory read write!
memory[2] = 3
The initial value of wcounts : 0
Task pid : 186, Wcount = 4, times = 4
root@generic:/data/misc #
```

```
module start load!
module get load!
module stop load!
healthd: battery l=50 v=0 t=0.0 h=2 st=2 chg=a
START TRACING : 186
have a good vm_area
do not have a write permit
now the wcounts is 1
have a good vm_area
have a good vm_area
do not have a write permit
now the wcounts is 2
have a good vm_area
do not have a write permit
now the wcounts is 3
have a good vm_area
do not have a write permit
now the wcounts is 4
have a good vm_area
GET TRACE !
the value of wcounts: 4
```

3 Race-Averse Scheduler

3.1 Problem Restatement

- When the linux kernel scheduler choose a task, it considers a variety of factors. There two scheduling policy for real time process : `SCHED_FIFO`, `SCHED_RR`.
- In this problem, we will try to implement a **Race-Averse-Scheduling** algorithm to the Linux.
- The scheduling is a weight round robin style scheduling according to **race probabilities** of each task.
- Race Probabilities - The race probabilities of a task is represented as an integer in the range of [0, 10). The higher the integer is, the more likely that the tasks may race with others. - The race probability of a certain task is proportional to its frequency of accessing the given range of virtual addresses.

3.2 Files to Modification

The follows files are modified to implement RAS:

- `/arch/arm/configs/goldfish_armv7_defconfig`
- `/include/linux/sched.h`
- `/kernel/sched/core.c`
- `/kernel/sched/sched.h`