Cycle Accurate Per Process Timer

Witawas Srisa-an

Due Date: See course's website

In this project, you are asked to implement a new system call for Linux OS. This system call will work exclusively with Intel Performance Counter available in x686 or higher (PII and up or K6 and up). It is specified as follows:

get_pinfo — used to obtain the number of cycles the current process has been executing. This execution time must not include the time that the process is in block state.

This system call must first be created into the kernel without the code. Basically what we want to do here is to create a stub for the new system call. The stub will not do any logic and can be as simple as:

```
unsigned long long int sys_get_pinfo( pid_t pid, struct pinfo *pu)
{
     return 0;
}
```

The *pinfo* should have the following structure:

```
struct pinfo {
     unsigned long long int cycle_count; /* store the cycle count */
};
```

Please refer to the accompanying instruction (Appendix A) on how to create system call stub inside the linux kernel. You will also need to compile the kernel once the kernel code is modified for it to take effect. Again, the instruction on how to compile the kernel is also available in the appendix.

Once the stub is created, we will use loadable module to intercept the system call. With this approach, you can load and unload your module without recompiling or rebooting the kernel. This can save a lot of time during the implementation. If your code should crash the system, you simply need to reboot. I have provided a sample module to read the time-stamp-counter (in the appendix). You can use it as the reference to your implementation. Once the system call is intercepted, any call makes to it (e.g. *get_pinfo*) will not go to the stub. Instead, it will go to your code in the loadable module. When you unload your module, the system call will go back to the stub. I will provide a test program that you can use to test your implementation.

You need to submit a README file that include the project report. Again, the project report should discuss the difficulties encountered, workload distribution, your approach to the problem. The number of hours spend on the project. You should state clearly the steps necessary to complete the task.

Submission Procedure:

We will gread the project in class on due date (date to be set in class). You also need to submit the source files and project report. The steps are as follows:

1. create a project2 directory.
2. create two additional directory under project2, kernel and source.
3. put the system call modules (with proper comments), README file, all the files that you modify into source directory under project1.
4. put the kernel image under kernel directory.
5. go up one level above project2 directory and run the following command

```
tar -czvf project2.tar.gz project2
```

Submit your *project1.tar.gz* file using handin by the due date. There will not be an automatic extension for this project.

Appendix A: Instruction on How to Recompile Linux Kernel

Creating System Call Stubs

We are assuming that we will create a system call named *get_pinfo*. The header file for the system call is *performance.h*.

1. Obtain stable source release of kernel 2.4.
2. go to /usr/src/linux/arch/i386/kernel/entry.S and append the system calls to the end of the definition. e.g. .long SYMBOL_NAME(sys_get_pinfo) /* 243 get_pinfo */
3. go to /usr/src/linux/include/asm-i386/unistd.h and define your system call. For example
   - #define __NR_get_pinfo 243
4. go to /usr/include/bits/syscall.h and add the line
   - #define SYS_get_pinfo __NR_get_pinfo
5. go to /usr/src/linux/kernel/sys.c and add simple function as stub for our syscall module.
6. go to /usr/src/linux/include/linux/ and add the header file (performance.h) also leave a copy in /usr/include

Compiling the kernel

1. go to /usr/src/linux and type make mrproper (probably needed for the first compilation only)
2. from the same directory type make menuconfig (you can choose to include your network card if you like).
3. from the same directory type make dep; make clean
4. type make bzImage

Note just repeat the last two steps for subsequent compilation.

At this point, there should be a kernel image (bzImage) in /usr/src/linux/arch/i386/boot directory. You need to move this file to /boot directory. Now, you can rename this file to anything you want so I suggest

```
mv /usr/src/linux/arch/i386/boot/bzImage /boot/cs351Image
```

You then need to modify lilo.conf or grub.conf file (e.g. /etc/lilo.conf). For example, for my machine at home, this is what I have:

```
FILE LILO.CONF

boot=/dev/hda
map=/boot/map
install=/boot/boot.b
prompt
timeout=50
message=/boot/message
default=winnt

image=/boot/vmlinuz-2.4.2-2
     label=linux
     read-only
     root=/dev/hda8
     append="hdc=ide-scsi hdd=ide-scsi"

image=/boot/bzImage
     label=linux_mod
     read-only
     root=/dev/hda8
     append="hdc=ide-scsi hdd=ide-scsi"

image=/boot/hires_img
     label=hires_krnl
     read-only
     root=/dev/hda8
     append="hdc=ide-scsi hdd=ide-scsi"

image=/boot/cs351Image
     label=cs351
     read-only
     root=/dev/hda8
     append="hdc=ide-scsi hdd=ide-scsi"
```

```
other=/dev/hda2
     optional
     label=winnt

other=/dev/hdb2
     optional
     label=win2000
```

the variable ”default” specified the default boot. To create a boot menu to your kernel (i.e. cs351Image), just copy your default linux boot section and then modified the label image. If you want this kernel to be the default boot, you need to change ”default” value up top.

After you finish modifying you lilo.conf file, just run /sbin/lilo You will see the boot option changes. REBOOT YOUR SYSTEM and make sure that YOU BOOT WITH the NEW KERNEL. If you use GRUB, you won't need to run any command

At this point, you should be in a new kernel that would have the function get_pinfo as a new system call. You will then need to compile the module that will intercept the system call. You can do this by:

```
gcc -c test_module.c
insmod test_module.o
```

Then you can test your program (e.g. the given test_module.c).

Without the loadable module, your get_pinfo.c program would return the value from the stub. With the module loaded, it should return the value of read-time-stamp-counter.

```
FILE PERFORMANCE.H
#include<linux/unistd.h>
#define get_pinfo() syscall(243)
/* add all the specified system calls here */

FILE TEST_MODULE.C

#ifndef __KERNEL__
#define __KERNEL__
#endif
#ifndef MODULE
```

```
#define MODULE
#endif

#include <linux/module.h>
#include <linux/modsetver.h>
#include <linux/kernel.h>
#include <performance.h>

extern void *sys_call_table[];
void *original_get_pinfo;
unsigned long long int my_rdtscll(void);

extern long long int our_new_get_pinfo()
{
     return my_rdtscll();
}

int init_module()
{
     original_get_pinfo = sys_call_table[243];
     sys_call_table[243] = our_new_get_pinfo;
     printk("System call has been intercepted\n");
     return 0;
}
void cleanup_module()
{
     sys_call_table[243]=original_get_pinfo;
     printk("System call has been returned to normal\n");
}

unsigned long long int my_rdtscll()
{
     unsigned long long int val;
     __asm__ __volatile__("rdtsc" : "=A" (val));
     return val;
}
```