Lab #4 Solutions: COMP 3000A (Operating Systems)
November 26, 2007

Please answer all questions below, there are 60 marks total. Part A of this lab is intended to be completed within the lab during Lab hours. Part B can be completed on your own time, either in the lab or on your own computer.

You may find the Linux Cross Reference website, `http://lxr.linux.no/source/`, useful when completing this assignment.

1 Part A

This section is designed to be completed in the lab. You get 10% of the total marks for attempting to do part A during assigned lab hours. Please ensure that one of the Lab instructors takes your attendance.

1. **[6]** Have the instructor mark down that you were present and attempted part A during lab hours.

2. **[3]** In the Linux kernel, what file contains the code for the system call dispatcher that executes on the machines in the lab? What language is it written in? Why?

   linux-2.6.19.1/arch/i386/kernel/entry.S is the file. It is written in x86 assembly because it must directly manipulate CPU registers (it must save the register state of the calling process and setup the kernel's execution environment).

3. **[2]** What is the highest numbered system call currently implemented? What is the name of this system call?

   319, epoll_pwait

4. **[2]** Look in the file linux-2.6.19.1/fs/ext3/file.c. What function implements the read operation for ext3 filesystems? The write function? (Note that one is filesystem specific, while the other is generic.)

   do_sync_read and do_sync_write implement the synchronous versions of read and write. (That's all you had to say to get this question right; however, these functions are actually thin wrappers around the asynchronous functions which are also specified in the same structure: generic_file_aio_read and ext3_file_write.) Note that the hint for this question is wrong!

5. **[2]** In what file are the directory inode operations of ext3 defined? In what structure?

   fs/ext3/namei.c, in ext3_dir_inode_operations

6. **[3]** What function (from what file) calls do_ execve() on the kernels running in the lab? Why can't the system call dispatcher directly call do_execve()?

   sys_execve() in arch/i386/kernel/process.c. The dispatcher cannot call this function directly because it needs to get arguments from the appropriate register, and this cannot be done by an architecture-independent function - hence the arch-dependent wrapper.

7. **[2]** Briefly, what is the purpose of the `bprm` structure as used in the function do_execve()?

   It holds the arguments that are used when loading binaries (loaded by the execve system call).

2 Part B

The kernel source on the lab computers contains a skeleton file which will be used in completing this lab. This file is located at `kernel/comp3000.c`. Currently, the code in the file creates a proc filesystem entry at `/proc/comp3000` that outputs *Hello World.* In this lab, you will be modifying the file comp3000 to do more.

While you do not need to turn in the entire comp3000.c source file, please do give enough context so it is clear where you made your modifications.

1. **[10]** Modify `kernel/comp3000.c` to output the process number and name of the process which has a PID closest to 1000 (without going over).

2. **[20]** Modify `kernel/comp3000.c` to output a complete list of process ID and command lines. Your output should be similar to that seen when running the `ps -e --format "pid args"`.

3. **[10]** Modify `kernel/comp3000.c` so that you can write a number to the proc filesystem and subsequent reads will return all processes with ID's above the number written. Writing an ID number of 0 should result in all processes being shown when reading from the proc file.

Solutions start on next page.

```
#include <linux/module.h>
#include <linux/types.h>
#include <linux/kernel.h>
#include <linux/errno.h>

#include <linux/proc_fs.h>

/* How did I know what files to include... I tried compiling and then
 * fixed the 'undefined function' errors by doing a search on the
 * linux cross reference to determine what include file it was
 * declared in.  A fairly simple process. */
#include <linux/mm.h> /* For access_process_vm */
#include <asm/uaccess.h> /* For copy_from_user */

static struct proc_dir_entry * entry = NULL;
static struct proc_dir_entry * entry2 = NULL;

static int minpid;

/* Proc filesystem data structure.  We use this and the associated
 * proc_out function so that we don't have to keep track of every
 * write explicitly in the proc_read function.  It lets the proc_read
 * function be simplified by a great deal. You could have done this
 * without the helper function, but your proc_read function would have
 * been significantly longer. */
struct proc_out_data {
    char * buffer;
    off_t offset;
    int buffer_len;
    int outlen;
};

/* This function prints the string into a temporary buffer and then
 * depending on the offset of the read and the size of the data buffer
 * for the read can copy all or part of the resulting temporary string
 * into the output buffer.  It updates the structure accordingly for
 * the next call to this function. */
static int proc_out( struct proc_out_data * file, const char * format, ... ) {
    char buffer[256];
    int len, i;
    va_list ap;

    va_start( ap, format );
    len = vsnprintf( buffer, sizeof(buffer), format, ap );
    for( i = 0; i < len; i++ ) {
        if( file->offset > 0 ) { file->offset--; }
        else if( file->buffer_len == 0 ) { return 0; }
        else { file->buffer[0] = buffer[i]; file->buffer++; file->outlen++; }
    }
    return len;
}

/* This function is copied straight from fs/proc/base.c */
static int proc_pid_cmdline(struct task_struct *task, char * buffer) {
    int res = 0;
    unsigned int len;
    struct mm_struct *mm = get_task_mm(task);
    if (!mm)
        goto out;
    if (!mm->arg_end)
        goto out_mm;    /* Shh! No looking before we're done */

    len = mm->arg_end - mm->arg_start;

    if (len > PAGE_SIZE)
        len = PAGE_SIZE;

    res = access_process_vm(task, mm->arg_start, buffer, len, 0);

    // If the nul at the end of args has been overwritten, then
    // assume application is using setproctitle(3).
    if (res > 0 && buffer[res-1] != '\0' && len < PAGE_SIZE) {
        len = strnlen(buffer, res);
        if (len < res) {
            res = len;
        } else {
            len = mm->env_end - mm->env_start;
            if (len > PAGE_SIZE - res)
                len = PAGE_SIZE - res;
            res += access_process_vm(task, mm->env_start, buffer+res, len, 0);
            res = strnlen(buffer, res);
        }
    }
 out_mm:
    mmput(mm);
 out:
    return res;
}


/*
 * Used by proc to find the first pid that is greater then or equal to nr.
 *
 * If there is a pid at nr this function is exactly the same as find_pid.
 *
 * Defined in 2.6.19, backported to 2.6.18 here.
 */
struct pid *find_ge_pid(int nr)
{
    struct pid *pid;
    do {
        pid = find_pid(nr);
        if (pid)
            break;
        nr++;
    } while (nr > 0 && nr < PID_MAX_DEFAULT);
    return pid;
}


static int proc_read(char *buffer, char **buffer_loc, off_t offset, int buffer_len, int *eof, void *data) {
    struct proc_out_data proc_data = { .buffer = buffer,
                                        .offset = offset,
                                        .buffer_len = buffer_len,
                                        .outlen = 0 };

int curpidid = minpid;
char * tmp_buffer;

tmp_buffer = kmalloc(PAGE_SIZE + 1, GFP_KERNEL);

proc_out( &proc_data, "  PID COMMAND\n" );

/* Ok, we have a PID, it's stored in minpid.    We need to find the
 * next PID greater than or equal to the current PID */

/* How did we get this information?
   - Recognize that the kill systemcall sends a signal to a PID.
   - It has to get the structure for a given PID...
   - The System call for kill calls sys_kill
   - Using the LXR, sys_kill is defined in kernel/signal.c
   - It just calls kill_something_info
   - kill_something_info is defined in kernel/signal.c
   - It calls kill_proc_info if PID is valid
   - kill_proc_info is defined in kernel/signal.c
   - It references a function called find_pid...
   - Slightly lower in the same file is a function called pid_task
     which returns a task_struct!
   - In seeing how to use it, we stumble across kernel/pid.c which
     has a whole lot of usefull functions for dealing with PIDS!
*/

/* We need to hold the lock to the pid structures when we itterate
 * through them so that a process exiting does not cause us to
 * oops. */
rcu_read_lock();

struct pid * curPid;
do {
    struct task_struct * curTask;

    curPid = find_ge_pid( curpidid );
    if( !curPid )
        break;
    curTask = pid_task( curPid, PIDTYPE_PID );

    /* Ok, we have the PID, but we still need the command line.
     * It just so happens that the proc entry cmdline for each
     * file gives us that, so let's see how that function is
     * implemented! So, we find the function proc_pid_cmdline in
     * proc/base.c (by doing a grep for cmdline inside the fs/proc
     * directory) and we copy the function out and paste it
     * above. By reading the function, we know that the maximum
     * length written to the buffer is going to be PAGE_SIZE, so
     * that's how big we make the buffer which takes the data. */
    int cmdlen = proc_pid_cmdline( curTask, tmp_buffer );
    tmp_buffer[cmdlen] = '\0';
    /* Replace NULL's with spaces so we get the full command line. */
    for( cmdlen-- ; cmdlen >= 0; cmdlen-- ) {
            if( tmp_buffer[cmdlen] == '\0' ) tmp_buffer[cmdlen] = ' ';
        }

        /* And we print out the data we found. */
        proc_out( &proc_data, "%5d %s", curPid->nr, tmp_buffer );
        proc_out( &proc_data, "\n" );

        /* And on to the next pid... */
        curpidid = curPid->nr + 1;

    } while ( 1 );

    /* Unlock the lock so we don't freze the kernel. */
    rcu_read_unlock();

    kfree( tmp_buffer );

    /* Well, that was not very painfull... */

    if( proc_data.outlen == 0 )
        *eof = 1;

    return proc_data.outlen;
}

/* Below is the solution to part 1.  It is slightly different in that
 * it has to go through the pids in a decreasing order in order to
 * find the one closest to 1000.  Comments are more sparse as it's
 * just a copy of the code above but tweaked to answer part 1. */
static int proc_read1(char *buffer, char **buffer_loc, off_t offset, int buffer_len, int *eof, void *data) {
    struct proc_out_data proc_data = { .buffer = buffer,
                                       .offset = offset,
                                       .buffer_len = buffer_len,
                                       .outlen = 0 };
    char * tmp_buffer;
    int i;

    tmp_buffer = kmalloc(PAGE_SIZE + 1, GFP_KERNEL);

    rcu_read_lock();

    /* Find the number of the pid... */
    struct pid * curPid = NULL;
    for( i = 1000; i >= 1 && !curPid; i-- )
        curPid = find_pid( i );
    struct task_struct * curTask;
    curTask = pid_task( curPid, PIDTYPE_PID );

    int cmdlen = proc_pid_cmdline( curTask, tmp_buffer );
    tmp_buffer[cmdlen] = '\0';
    /* Replace NULL's with spaces so we get the full command line. */
    for( cmdlen-- ; cmdlen >= 0; cmdlen-- ) {
        if( tmp_buffer[cmdlen] == '\0' ) tmp_buffer[cmdlen] = ' ';
    }

    /* And we print out the data we found. */
    proc_out( &proc_data, "%5d %s", curPid->nr, tmp_buffer );
    proc_out( &proc_data, "\n" );

    /* Unlock the lock so we don't freze the kernel. */
    rcu_read_unlock();

    kfree( tmp_buffer );

    if( proc_data.outlen == 0 )
        *eof = 1;

    return proc_data.outlen;
}

/* This function is responsable for obtaining the integer which
 * represents the lowest numbered PID to display. The function
 * prototype can be found by examining the type of function pointer in
 * the proc_dir_entry structure. As for how to use the buffer, examine
 * how one of the other kernel functions uses write_proc_t by using the
 * Linux cross-reference. In this case, the
 * drivers/acpi/toshiba_acpi.c file was the first hit and so we just
 * modify that code slightly. */
static int proc_write(struct file *file, const char __user *buffer,
                      unsigned long count, void * data) {
    int result;
    char * tmp_buffer;

    if( !buffer )
        return -EINVAL;
    if( !count )
        return -EINVAL;

    tmp_buffer = kmalloc(count + 1, GFP_KERNEL);
    if (!tmp_buffer)
        return -ENOMEM;

    if (copy_from_user(tmp_buffer, buffer, count)) {
        result = -EFAULT;
    } else {
        tmp_buffer[count] = 0;
        sscanf( tmp_buffer, "%i", &minpid );
        result = count;
    }
    kfree(tmp_buffer);
    return result;
}

/* I include both answers in the same file by creating two proc
 * entries. You did not have to do this in your solution, it would
 * have been sufficient to give two different source files. I include
 * both in one file for simplicity (and it allows testing of all
 * solutions at the same time. */
static int proc_setup(void) {
    /* This sets up a proc entry which answers part 1. */
    if( !entry ) {
        entry = create_proc_entry( "comp3000-1", S_IFREG | S_IRUGO, NULL );
        if( !entry ) {
            return -ENOMEM;
        }
        entry->read_proc = proc_read1;
        entry->write_proc = NULL;
        entry->owner = THIS_MODULE;
        entry->uid = 0;
        entry->gid = 0;
        entry->size = 0;
    }
    /* This sets up a proc entry which answers part 2 and 3. */
    if( !entry2 ) {
        entry2 = create_proc_entry( "comp3000-23", S_IFREG | S_IRUGO, NULL );
        if( !entry2 ) {
            return -ENOMEM;
        }
        entry2->read_proc = proc_read;
        entry2->write_proc = proc_write; /* Need to set up the write function */
        entry2->owner = THIS_MODULE;
        entry2->uid = 0;
        entry2->gid = 0;
        entry2->size = 0;
    }
    return 0;
}

static void proc_remove(void) {
    if( entry != NULL ) {
        remove_proc_entry( "comp3000-1", NULL );
        entry = NULL;
    }
    if( entry2 != NULL ) {
        remove_proc_entry( "comp3000-23", NULL );
        entry = NULL;
    }
    return;
}

static int __init comp3000_module_init(void) {
    minpid = 1000;

    return proc_setup();
}

static void __exit comp3000_module_exit(void) {
    proc_remove();
}

module_init(comp3000_module_init)
module_exit(comp3000_module_exit)
MODULE_LICENSE("GPL");
```