# CPE454 Laboratory Assignment #2
## *Adding a System Call*

Michael Haungs, Spring 2009

## 1 Objective

In this assignment, you will add a new system call, *swipe()*, to the Linux kernel that transfers the remaining timeslice of each process in a specified set to a target process. You will also demonstrate various uses of the system call (both advantageous and detrimental).

## 2 Resources

You should read the following Linux manual pages:

- nice: change process priority.
- kill: sends a signal to a process (or group of processes).
- time: time a simple command or give resource usage. This will help you in doing performance comparisons.
- gettimeofday: get current time with microsecond accuracy. This will help you in doing performance comparisons.

You should look at the following functions/macros found in the Linux kernel in the file *kernel/sched.c*:

- sys_kill(): backing system call of the *kill* command.
- for_each_process(): kernel macro used to iterate through all processes.
- task_struct: Linux's data structure that represents a Process Control Block (PCB).
- schedule(): the scheduler function for the linux kernel.
- task_timeslice(): Converts priority to a timeslice value.
- scheduler_tick(): Notice how the variable *timeslice* is used and updated.
- sys_nice(): the implementation of the *nice()* system call.

The resource links found at the bottom of our course web page will be **very helpful for this lab assignment**.

# 3 Assignment

There are multiple stages to this assignment. The first goal is to teach you how to add a system call to the linux kernel. Next, you will implement a new system call, *swipe()*, and add it to linux. Last, you will investigate different uses of this new system call.

## 3.1 Adding a simple system call

### 3.1.1 Simple Example

Say, we wanted to add our own version of the system call *getpid()*. Let's call our version *mygetpid()*. The implementation of *mygetpid()* is:

```
asmlinkage long sys_getpid(void)

{

     return current->tgid;

}

NOTE: asmlinkage must appear before every system call.

It tells the compiler to only look on the stack for the functions

arguments (aka compiler magic).
```

Here are concise steps you need to follow to add this system call:

1. Implement the function call and put it in the appropriate file. Since *getpid()* is defined in kernel/timer.c we'll put the above implementation of *mygetpid()* in the same file.
2. Add an entry to the end of the system call table.
   (a) vi arch/i386/kernel/syscall_table.S
      i. For 2.6.24, change to "arch/x86/kernel/syscall_table.S
   (b) Add the line ".long sys_mygetpid" after the line ".long sys_inotify_rm_watch" (the last one in the list of system calls).
      i. For 2.6.17, add after ".long sys_vmsplice".
      ii. For 2.6.24, just go to then end of the list.
3. Define the system call number in include/asm/unistd.h
   (a) vi include/asm-i386/unistd.h
      i. For 2.6.24, change to "include/asm-x86/unistd_32.h"
   (b) Add the line "#define __NR_mygetpid 294" after the line "#define __NR_inotify_rm_watch 293"
      i. For 2.6.17, add after "#define __NR_vmsplice 316".
      ii. For 2.6.24, change "294" to "325".
   (c) Change the line "#define NR_syscalls 294" to be "#define NR_syscalls 295"
      i. For 2.6.17, change to "#define __NR_syscalls 317"
      ii. For 2.6.24, change to "#define __NR_syscalls 326"
4. Recompile your kernel and boot to it.

### 3.1.2 Accessing the System Call from User-Space

To test this new system call use the following code:

```
Compile Command:

gcc -I/The/Location/of/your/linux/include testSysCall.c

Note:

Change the above to reflect the location of your kernel source code.

Code:

#include<stdio.h>

#include<linux/unistd.h> /* Defines _syscall0 and has mygetpid syscall number */

#include<errno.h> /* need this for syscall0 macro too */

_syscall0(long, mygetpid) /*Can call "mygetpid" now */
/* If you are using 2.6.24, delete the above line */

int main()

{

  printf("Process ID: %d\n", mygetpid());
  /* For 2.6.24, change to "printf("Process ID: %d\n", syscall(325)); */

  return 0;

}
```

### 3.1.3 Tutorial for adding a System Call to the 2.4 kernel

The tutorial below provides a good explanation of adding a system call to the linux 2.4 kernel. While the steps in the tutorial are slightly different from the ones given above, I still highly recommend that you read it.

"How To Add a System Call to Linux on an i386" `http://www.superfrink.net/docs/sys_call_howto.html`

## 3.2 Adding the *swipe()* System Call

You are going to add a system call, *swipe(),* that steals the collective timeslices for a specified set of processes and adds it to the target process' timeslice. The *swipe()* system call takes a target process id, an integer that provides the type of the set of processes, and an integer that provides the process set. The system call returns the total amount of timeslice swiped or a negative number if an error occured. Therefore, the prototype of the function is:

```
int swipe(pid_t target, int which, int who)
```

The parameter *target* provides a process id for the process the will receive the extra timeslice. If this process id is invalid, then *swipe()* should return -EINVAL. The *which* parameter can take the value 0, 1, or 2 to indicate that the parameter *who* specifies a process id, a process group id, or a user id, respectively. If *who* is a process id, then the process set is that process plus all of its decendants. If *who* is a process group id, then the process set is all processes in that group. If *who* is a user id, then the process set is all processes owned by that user.

Look at the use of the functions *sys_kill(), sys_setpriority(),* and *scheduler_tick()* for clues on how to implement *swipe().* The best way to examine these functions is to use the "Cross-Referencing Linux" link found at the bottom of our course web page.

A few more tips:

1. Read this online chapter: The Linux Process Scheduler `http://www.samspublishing.com/articles/article.asp?p=101760`
2. Read the following article on Linux's 2.6 scheduler: Inside the Linux 2.6 Scheduler `http://arstechnica.com/etc/linux/2003/linux.ars-12242003.html`

# 4 Using *swipe()*

Now, you're going to explore two uses of swipe(). Use swipe() to:

1. Create a process that monopolizes the CPU.
2. Create a wrapper program that takes a simple command as an argument and arguments to specify a set of processes and executes the simple command starting with the collective timeslices of all processes in the set. Here is an example of the wrapper (called swipe_it):

```
> swipe_it 0 3500 ls -l
```

In the above command, the wrapper (*swipe_it*) will steal all the current timeslices from the process 3500 and all of its children and give it to the simple command "ls -l".

# Deliverables

You need to handin a kernel patch, called *swipe_patch*, that implements *swipe()*, the code for *swipe_it*, and a README file (must be spelled as shown...no lowercase) that describes your implementation of swipe(), any problems/bugs with your implementation, and any other information you feel would be useful during grading.

In lab the day it is due, April 23rd, you will also be required to demonstrate a program that uses swipe() to monopolize the CPU as well as *swipe_it*.

***You submit swipe_patch, swipe_it.c, and README files using the handin command to the 2lab directory on or before 11:59pm on April 23.***