CPE453 Laboratory Assignment #4
Adding a System Call

Michael Haungs, Spring 2011

1 Objective

In this assignment, you will add a new system call to Linux that acts like a *Fountain of Youth* for processes that call it. Using this system call, processes will be able to shed nanoseconds from their age (measured in execution time). You will then perform experiments to study the system call's impact on compute bound and I/O bound processes and threads.

2 Resources

Of course, you will want to use your *mond* program from Lab2 to monitor resource utilization.

You should read the following Linux manual pages:

- nice: change process priority.
- 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.
- _syscall, syscall: invoking a system call without library support.

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

- schedule(): the scheduler function for the linux kernel.
- tick_periodic(): timer interrupt handler.
- scheduler_tick(): Notice how the variable *timeslice* is used and updated.
- update_curr(): manages updating scheduling accounting data.
- enqueue_entity(): Add process to runqueue (CFS red-black tree)
- struct sched_entity: Keeps track of scheduling data
- struct task_struct: Linux's process control block
- SYSCALL_DEFINE0: A macro to help you define a new system call. There are also versions that end in 1,2,3... The number indicates the number of arguments the system call has.

- sys_nice(): the implementation of the *nice()* system call. Pay special attention to the function *set_user_nice* called inside *sys_nice*.

Finally, you should do some web searches and read as much as you can about Linux's CFS (Completely Fair Scheduler). Here are some links I found (not necessarily the best) :

- http://en.wikipedia.org/wiki/Completely_Fair_Scheduler
- http://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler/
- http://www.scribd.com/doc/24111564/Project-Linux-Scheduler-2-6-32

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, *timemachine()*, and add it to linux. Last, you will perform a series of experiments to analyze the performance and system-wide impact of your newly implemented 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:

```
SYSCALL_DEFINE0(mygetpid)/*expands to: asmlinkage long sys_getpid(void)*/
{
     return task_tgid_vnr(current); /* wrapper for "current->tgid;" */
}
```

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. (Be careful not to nest your function definition inside a #ifdef...#endif block.)
2. Add an entry to the end of the system call table.
   (a) vi arch/x86/kernel/syscall_table_32.S
   (b) Add the line ".long sys_mygetpid" after the line ".long sys_recvmmsg" (the last one in the list of system calls).
3. Define the system call number in the appropriate header file
   (a) vi arch/x86/include/asm/unistd_32.h
   (b) Add the line "#define __NR_mygetpid 338" after the line "#define __NR_recvmmsg 337"
   (c) Change the line "#define NR_syscalls 338" to be "#define NR_syscalls 339"
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:

```
#include<stdio.h>
#define _GNU_SOURCE
#include<unistd.h>
#include<sys/syscall.h>
#include<sys/types.h>

#define __NR_mygetpid 338

int main()
{
  printf("Process ID: %ld\n", syscall(__NR_mygetpid) );
  return 0;
}
```

3.2 Adding the timemachine() System Call

Now that you are linux kernel compiling experts, let's do something more useful. You are going to add a system call, *timemachine(),* that subtracts the provided argument (in nanoseconds) from the calling process' virtual running time[1].

Look at the use of kernel functions I listed in Section 2. This should give you a clue on how to implement *timemachine().* The best way to examine these functions is to use the "Cross-Referencing Linux" link found at the bottom of our course web page. There is one function parameter to *timemachine()* that contains the number of nanoseconds you want to subtract from the virtual running time. Your new system call, *timemachine()*, returns 0 on success or a negative number if the call fails. Therefore, its function prototype is: *int timemachine(unsigned long).*

A few more tips:

1. Read Read Read
2. Read the following text files found in the *Documentation/scheduler* directory of your linux kernel source:
   (a) sched-design-CFS.txt
   (b) sched-nice-design.txt
3. Put your implementation of *timemachine()* in the file kernel/sched_fair.c.

Deliverables

Your lab grade is based on a lab writeup that describes your *timemachine()* implementation in depth. The writeup should be between 10-15 pages long, single column, 1 inch margins, single-spaced, with a font size of 12pt. Your writeup should minimally contain the following sections:

1. Introduction

[1] You'll learn more about the "virtual running time" when you read about CFS.

2. Background: Fully describe how processes are scheduled in Linux 2.6.38. Be sure to mention the effect of process priority.

3. Code Description: You should provide the code for your implementation of *timemachine()* and discuss it thoroughly (justify every line of code). Could you have designed the system call differently? Where there any tradeoffs in the design choices you made?

4. Experiments: Describe the experiments you ran. Your experiments should measure the affect your new system call has on I/O-bound processes, compute-bound processes, and groups of processes for varying input values and frequencies of *timemachine()* use. [2] Why did you choose these experiments? What are you trying to show? Show your results in graph and/or table form. *Give an analysis of the results you got.* How does the use of your system call affect system throughput, response time, resource utilization, fairness, and predictability? What is the associated overhead in using your system call? THIS SECTION IS THE MOST IMPORTANT IN YOUR WRITEUP.

5. Conclusion: Do you think this system call should be added as a standard one? In what other circumstances do you think this system call would be useful? In what ways could it be used to exploit the system?

Submit your implementation of *timemachine()* in these two files: timemachine.c and timemachine.h.[3] Also, submit a README file describing what you got working, any bugs or unusual difficulties you encountered, and any other notes for the grader. You need to submit your writeup as a PDF file. Last, you need to bring a printed copy of your writeup to class on the due date.

You submit the writeup, timemachine.c, timemachine.h, and README file using the digital dropbox on or before 11:59pm on May 19.

[2] Be sure to run each experiment multiple times and report the average result. Why is this important?

[3] You should copy your code for your system call from sched_fair.c and other appropriate files and paste it into timemachine.c and timemachine.h.