CS 3013 Operating Systems

WPI, C Term 2007
Project 3 -- Diving Into LINUX

Important: This project is to be done by each student group. This is not an individual student project.

Introduction

The getrusage() system call returns resource information about a process. (This is what you used in project 1.) The rusage structure it uses has a number of fields, not all of which are filled in by a given operating system. In this project you will examine the Linux kernel code to see what fields it fills in for the rusage structure and extend the kernel implementation so that the getrusage() system call returns information about context switches. Context switches occur in the process scheduler, which is at the core of the operating system.

Description

For this project, you should initially look at the getrusage() system call. It is in /usr/src/linux/kernel/sys.c. The system call itself is sys getrusage() which calls the routine getrusage(). You should see that the filled-in fields of the rusage structure come from the structure task_struct. This latter structure contains information about each process (task) in the system. Its definition is in /usr/include/linux/sched.h.

Once you have identified the task_struct fields that are used to fill in the rusage structure, you should find where these task_struct fields are initialized and modified in the source code. You will need to look in kernel/fork.c and kernel/exit.c for initialization and updates of these fields. You can look in kernel/sys.c and files in the directory /usr/src/linux/mm to see where the task_struct fields are updated. You can use grep to determine if there are other references to this structure in the code.

After familiarizing yourself with what getrusage() already does, you need to extend its functionality to return meaningful values for **voluntary and involuntary context switches**. You will first need to add fields to the task_struct to keep track of context switches for each process. You will need fields both for the process itself and its children. You can model your changes on how minor and major page faults are handled, although as described later you will be keeping track of voluntary and total context switches. When you add to struct task_struct, you also need to change the INIT_TASK macro (also in sched.h) to be sure the initial values are in place. Also, note that sched.h has a lot of files depending upon it, meaning there will be a lot that need recompilation every time you modify it. Change sched.h as little as possible.

Once you have the structure changes in place, you need to modify appropriate code to return the

value of these fields for a call to getrusage(). Initially, just initialize the values to a fixed, non-zero value, such as one, so you can verify your code is working. When you call getrusage() at this point it will just return this fixed value. Your code should accumulate values for child processes when these processes exit (as done for other fields in kernel/exit.c). If you test your code with the mini-shell from project 1, you should see the number of context switches increase for each forked child process.

Recording Context Switches

When you have verified that your changes work, you are ready to modify the appropriate kernel code to actually update the task_struct fields when a context switch does occur. Context switches occur in the scheduler.

The scheduler is a kernel function called *schedule*(), that gets called from other system call functions (usually when a process goes to sleep waiting for I/O), after every system call and after some interrupts. When invoked, the scheduler:

1. Performs some basic periodic tasks, like handling interrupt service routines (not a concern of this project)

2. Chooses one process to execute according to the scheduling policy

3. Dispatches the chosen process to run

The Linux scheduler contains different built-in scheduling strategies with SCHED_OTHER as the default. You will not need to be concerned about specific policies for this project.

Linux maintains a counter *kstat.context_swtch*, which is a global counter that is incremented whenever a context switch occurs. This increment occurs in the *schedule*() function, when the process identified by the task_struct pointer variable prev switches to the task_struct pointer variable next (where prev and next are different). At this point you can insert a statement to increment the total number of context switches (both voluntary and involuntary) for the process pointed to by prev (since you will keep track of the number of context switches **from** a process rather than **to** a process so you should use prev rather than next). You will then have the total number of context switches, both voluntary and involuntary. At this point, a voluntary context switch means the process is in a state other than TASK RUNNING and is most likely waiting for I/O. TASK RUNNING is used as the state for both running and ready processes. Therefore, if prev->state is not TASK RUNNING then the voluntary context switch count can be incremented as well.

Hints

When writing kernel code, you will want to print messages to stdout, as you do in printf(). Since many parts of the kernel may not have access to the stdio library, kernel developers wrote their own version of printf() called printk(). printk() basically behaves the same as printf(), in terms of formatting. Furthermore, printk() also writes messages to the log file /var/log/messages, so you can view output there in case your modified OS crashes. You might add prefixes to your printk() messages, such as ”Our-Msg: ” or ”Fossil: ” so you can more easily pick out your messages from the log file. Be careful! If you have printk() messages in a part of the kernel that is accessed frequently it can fill up your log file

quickly. When this happens, your system can become unstable. Check the size of your log file (using ls -l) and the disk space that is free (using du) frequently.

You are advised to take a conservative, incremental strategy for developing your new scheduling policy:

- **Plan, Plan, Plan**: This project has many steps. These steps are listed below, but I recommended that you write out your plan of attack in greater detail than is given in this outline.

- Familiarize yourself with the kernel code where fields used by *getrusage()* are updated. Use printk() statements as needed to build up confidence where to add your modifications.

- Add fields to the task_struct to keep track of context switches for each process. You will need fields both for the process itself and its children. Once you have the structure changes in place, you need to modify appropriate code to return the value of this fields for a call to getrusage(). Initially, just initialize the values to a fixed, non-zero value, such as one, so you can verify your code is working. When you call *getrusage()* at this point it will just return this fixed value.

- Modify kernel/sched.c to properly record context switches. You may use printk() statements here to build up confidence, but this code is a core part of the operating system and will result in numerous log messages so pay attention to the size of the logfile.

- Test your changes using a copy of your shell from project 1 compiled to use your modified *getrusage()* system call.

- The file sched.h has a lot of files depending upon it, meaning there will be a lot that need recompilation every time you modify it. Try to minimize the number of times you modify this file.

Remember to save your work frequently in case you crash your machine or need to "roll-back" to a previous working source code version! Refer to http://fossil.wpi.edu/ for more information on how to do this and general use of the Fossil lab and other useful Linux links.

Additional Work

Completion of all portions up to this point defines the basic objective for this project. These portions are worth a total of 85 out of the 100 points for the project. For the final 15 points of the project, you need to implement an additional system call *getrusagepid()* along with a sample test program. See the Fossil Web page for an overview of adding a system call to the Linux operating system.

The *getrusagepid()* system call is similar to getrusage(), but the first argument is now a process id,

rather than RUSAGE_SELF or RUSAGE_CHILDREN. This system call should get a pointer to the current process using the routine *find_task_by_pid(),* which is defined in sched.h. This routine returns the task_struct pointer for the given pid or NULL if the given pid does not exist. Once your routine has the task_struct pointer just call getrusage() in kernel/sys.c to return RUSAGE_SELF information for the given process id. If the given process id does not exist then return a value of -1 just as getrusage() does.

Write a simple test program, getrinfo, which takes a single optional argument. The argument is the numeric process id for which to obtain resource information. You can use the function atoi() to convert the command line string to an integer. If the argument is not given, then default to the process id of the current process.

The output of getrinfo should be similar to that used for project 1 except you will not print wall-clock time. You need to only print fields that you know have a meaningful value returned by the system call.

```
$getrinfo 341
< print rusage stats for pid 341 >
$getrinfo
< print rusage stats for this process >
```

Submission of Assignment

You must hand in the following Components:

1. All modified source code files for your solution (such as sys.c, sched.c, sched.h and other files you modified).

2. A compiled version of your kernel.

3. A brief description of your design.

4. Instructions on how to incorporate your code into the kernel tree and compile it. Note to the wise. You would like to make the TAs happy. You can do this by making their life easy. If you give them a script to run, their life is easy!

5. A copy of project 1 code that can be compiled to demonstrate the use of the new version of getrusage().

6. A copy of the source code for getrinfo if you did the additional work.

Please use the following submission procedure:

Use “proj3” as the project name for turnin (/cs/bin/turnin). When doing turnin, also include a file named ”group.txt” which contains the following:

```
group_name
```

```
login_name1              last_name1, first_name1
login_name2              last_name2, first_name2
```

Also, before you do the turnin, tar up (with gzip) your files. For example:

```
mkdir proj3
cp * proj3    /* copy all your files to submit to proj3 directory */
tar -czf proj3.tgz proj3
```

then:

```
scp proj3.tgz login_name@ccc:~/
ssh  login_name@ccc               /*  will  ask  your  ccc  passwd  */
/cs/bin/turnin submit cs3013 proj3 proj3.tgz
```

Evaluation Guidelines
Project 3

Submission Mechanics	**35 total**
You used turnin and followed the rules given above, thus saving the TA's considerable time and effort.	5 points
Your kernel could be built based on the source code, and instructions that you provided. (#1 & #4 above)	10 points
The description of your design makes sense. (#3 above)	5 points
The kernel that you provided actually boots (#2 above)	10 points
Your copy of Project 1 code compiles and runs. (#5 above)	5 points
Context Switches	**50 points**
Your kernel and Project 1 rewrite code delivers a number for voluntary context switches.	15 points
Your kernel and Project 1 rewrite code delivers a number for INvoluntary context switches	15 points
The context switch numbers provided are correct. This means that you've tested your code under various circumstances and have thought about what the result SHOULD be in each of these instances.	1 - 20 points
Additional Work	**15 total**
The source code for getrinfo is included and compiles.	5 points
The new system call behaves as specified.	1 - 10 points