CS 1550 – Introduction to Operating Systems

Process Synchronization – Readers and Writers

Due Date – Sunday, November 24th 2013 (11:59pm)

Project Description

In order to ensure orderly execution of cooperating processes, mechanisms must be in place to ensure that concurrent access to shared memory preserves data consistency. In this project, the focus is on two types of processes, namely readers and writers. During their execution, writers may seek to write data to a shared memory area, while readers may seek to read data from the shared area. The readers and writers must be synchronized so that the writer does not overwrite existing data with new data until the reader has processed it. Similarly, the reader should not start to read until data has actually been written to the area. The critical-section concept has been introduced to ensure the consistency of shared data, and various synchronization techniques have been proposed to solve the critical-section problem. This project builds on the previous project and aims to augment the Linux Kernel with two system calls, **signal**() and **wait**(), to achieve synchronization of concurrent execution of readers and writers processes, using semaphores. More specifically, you are required to:

- Create a semaphore data type, for mutual exclusion,
- Implement the semaphore atomic operations, wait() and signal(), and add these operations to the Linux Kern, and
- Use the wait() and signal() synchronization operations to implement a solution to the readers and writers cooperation, for the following cases:
    - **Readers-Preference** – this policy requires that no reader be kept waiting unless a writer has already gained access to the shared memory
    - **Writers-Preference** – this policy requires that no writer, once added to the queue, shall be kept waiting longer than absolutely necessary. Once a writer is ready, the writer should perform its write as soon as possible.

Project Design

The project design involves two main components: two user-level applications, to implement the readers-preference and writers-preference policies, respectively; and two new system calls to synchronize access to shared data.

User-level application

The user-level application implements the readers-writers problem using processes. The input to this application consists of the number of writers and the number of readers. The application will use **fork**() to create the number of readers() and writers(), as specified in the command line. When it becomes ready to write, a writer generates a random integer, writes into the shared memory and outputs it to the screen. A reader reads the current value of the shared memory and prints it out to the screen. All readers and writers share the same buffer.

Invoking the ReadersWriters() executable code with two readers and two writers, may result in the following output:

```
➢ ReadersWriters  2  2

  Writer 1    – Wrote: 1
  Reader 2  – Read: 1
  Reader 1  – Read: 1
  Writer 2    – Wrote: 20
  Reader 1  – Read: 20
  Writer 1    – Wrote: 24
  Reader 2  – Read: 24
```

Of course, the final output depends on how the execution of the processes is interleaved and on the type of the readers-writers policy used. In all scenarios, the program should be execute as a deadlock-free, infinite loop.

Readers-Writers Synchronization

Semaphores will be used to achieve synchronization among readers and writers. To this end, you are required to create a semaphore data type and two atomic operations, signal() and wait().

Semaphore Structure

The following structure is used to implement semaphores:

```
struct RW_Sem
 {
   int value;
   struct ProcQ *waitingQ;
 }
```

Each semaphore has an integer value and a process queue. When a process must wait on a semaphore, it is added to the waiting queue. A signal operation removes one process from the queue of waiting processes, if the queue is not empty, and awakens that process.

Semaphore Operations

Two new system calls, with the following signatures, will be added to operate atomically on semphaores:

- asmlinkage long sys_RW_wait(struct RW_Sem *sem)
- asmlinkage long sys_RW_signal(struct RW_Sem *sem)

When the current process invokes a wait() operation, it may be put to sleep in the waiting queue. Furthermore, when a current process invokes a signal() operation, a process must be awaken if the waiting queue is not empty. To achieve this, we will use the following three commands and macro:

- **set_current_state**(TASK_INTERRUPTBLE) – this function changes the state of the currently executing process from TASK_RUNNING to TASK_INTERRUPTBLE. The TASK_INTERRUPTIBLE state means the task is sleeping but can be woken when a specific event occurs.
- **schedule**() – this function invokes the scheduler to select a new ready process to be executed.
- **wake_up_process**(sleeping_task) – this function attempts to wake up the nominated process, set its state to TASK_RUNNING, and move it to the set of runnable processes. The function returns 1 if the process was woken up and 0 if the process is already running.
- **current** –The Linux kernel uses a circular doubly-linked list of struct task_structs to store process descriptors. This structure is declared in linux/sched.h. The **current** macro is a link to the process descriptor (a pointer to a task_struct) of the currently executing process.

Based on the above, sending the current process to **sleep** as part of wait() operation can be accomplished as follows:

1. **sleeping_task = current;**
2. **set_current_state(TASK_INTERRUPTIBLE);**
3. **schedule();**

To **wake up** a sleeping process as part of the signal() operation, the function wake_up_process () is invoked

1. **wake_up_process(sleeping_task)**

The sleeping_task is a struct task_struct that represents a process put to sleep when the wait() operation was invoked. Note that the macro **current** can be used to save the pointer to the process structure as shown above.

Semaphore Atomicity

A semaphore can be viewed as a “shared data” that is accessed by concurrent processes. It is, therefore, critical that the semaphore operations be executed atomically, in order to guarantee that no two processes can execute wait() and signal() operations on the **same semaphore** at the **same time**. As discussed in class, disabling interrupts on every processor can be difficult to achieve in a multicore or multi-processor computing environment, with potential to significantly reduce system performance. In this project, since the critical sections of the signal() and wiat() operations are short, we use **spinlocks** to guarantee atomicity of the semaphore operations.

A spinlock is a basic mutual exclusion primitive to achieve locking in order to protect a shared resource from being modified by two or more processes simultaneously. The required include file for the spinlock primitives is <linux/spinlock.h>. An actual lock has the type spinlock_t.

A spinlock can be created with the following macro:

- DEFINE_SPINLOCK(sem_lock);

Like anyother data structure, a spinlock must be initialized. The initialization may be achieved as follows:

- At compile time: spinlock_t sem_lock = SPIN_LOCK_UNLOCKED;
- At run time: spin_lock_init(spinlock_t *sem_lock);

To acquire a lock, the process invokes the following primitive:

- spin_lock(&sem_lock)

To release a lock, the process invokes the following primitive:

- spin_lock(&sem_lock)

The above two primitives can be used to guarantee atomic operation on semaphores. Note, however, that processes should not be put to sleep while holding a lock, as this may result in a deadlock.

Shared Memory Creation

To achieve synchronization and concurrent execution, the readers and writers processes must be able to share the same memory region. The mmap() function can be used to request N bytes of RAM directly from the OS. In this project, we will invoke the mmap() function and request N bytes of RAM as follows:

- void ***ptr** = mmap(NULL, N, PRTO_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, 0, 0)

mmap() creates a new mapping in the virtual address space of the calling process. The return value is an address to the start of the page in RAM. The contents are initialized to zero.

- The first argument of mmap() specifies the starting address for the new mapping. In our case, we set the argument to NULL, and let the kernel choose the address at which to create the mapping.
- The second argument, N, specifies the length of the mapping.
- The third argument of mmap describes the desired memory protection of the mapping. It is either PROT_NONE or the bitwise OR of one or more of the following flags: PROT_EXEC (pages may be), PROT_READ (pages may be read), PROT_WRITE (pages may be written). In our case, both read access and write access are permitted to the page being mapped.
- The forth argument is a set of flags that provides information about the handling of the mapped data. The value of the flags is the bitwise inclusive OR of multiple options. In our case, we select the MAP_SHARED and MAP_ANONYMOUS flags, (i) to allow modifications to be shared between all processes mapping the same range and (ii) to allow updates to the mapping to be visible to other processes mapping the same region. Note that anonymous mapping maps an area of the process's virtual memory **not** backed by any file. In this respect an anonymous mapping is similar to malloc().
- The last two arguments specify the file descriptor of the file that is to be mapped, and the offset from where the file should be mapped. Since we are mapping anonymous memory, these parameters are set to 0.

Memory in the allocated page can be used to allocate and initialize variables. For example, allocating and initializing two integers in the page can be achieved as follows:

1. int *FirstInt;
2. int *SecInt;
3. FirstInt = **ptr**;
4. SecInt = FirstInt + 1;
5. *FirstInt = 0;
6. *SecInt = 0;

To allow sharing of the memory region, the mmap() should be invoked by the main process to that the memory remains accessible to the readers and writers after they are created using the fork() command. The shared memory can then be used by the processes to create and use the required structures and variables appropriately.

Computing Platform and Resources

Like in the previous project, you will use the CS machine, **thot.cs.pitt.edu** and the QEMU ("Quick EMUlator") virtual machine as the development environment to compile the kernel and execute the new system call.

File Backups

One of the major contributions the university provides for the AFS filesystem is nightly backups. However, the /u/OSLab/ partition is not part of AFS space. Thus, any files you modify under your personal directory in /u/OSLab/ are not backed up. If there is a catastrophic disk failure, all of your work will be irrecoverably lost. As such, it is recommended that you recommendation that you backup all the files you change under /u/OSLab to your ~/private/ directory frequently! Loss of work is not grounds for an extension.

Hints and Notes

- The printk() function is the version of printf(), which can be used for debugging messages from the kernel. Used as you see fit.

Requirements and Submission

For a full credit, the following must be achieved by the due date:

- Make a tar.gz file, named USERNAME-PrjPartII.tar.gz, and copy it to **~znati/submit/1550PrjPartII** by the deadline.
- In the tar file, you need to include **well-commented** files of the source code developed for this project, including the source code for each ReaderWriter() policy.

Appendix— Rebuilding the Kernel and Adding a System Call

To add a new syscall to the Linux kernel, there are three main files that need to be modified:

1. linux-2.6.23.1/kernel/sys.c

This file contains the actual implementation of the system calls.

2. linux-2.6.23.1/arch/i386/kernel/syscall_table.S

This file declares the number that corresponds to the syscalls

3. linux-2.6.23.1/include/asm/unistd.h

This file exposes the syscall number to C programs which wish to use it.

Setting up the Kernel Source

To set up the kernel you need to perform the following steps:

1. Copy the linux-2.6.23.1.tar.bz file to your local space under /u/OSLab/username:
   ```
   cp /u/OSLab/original/linux-2.6.23.1.tar.bz2 .
   ```
2. Extract
   ```
   tar xfj linux-2.6.23.1.tar.bz2
   ```
3. Change into linux-2.6.23.1/ directory
   ```
   cd linux-2.6.23.1
   ```
4. Copy the .config file
   ```
   cp /u/OSLab/original/.config .
   ```
5. Build
   ```
   make ARCH=i386 bzImage
   ```

You should only need to do this once, however redoing step 2 will undo any changes you've made and give you a fresh copy of the kernel should things go horribly awry.

Rebuilding the Kernel

As you work through this assignment, you will need to make changes to the kernel and recompile it. After you make changes, you recompile from the linux-2.6.23.1/ directory by simply issuing the command:

```
make ARCH=i386 bzImage
```

Installing and Setting up QEMU

QEMU is a program that provides you with the ability to execute a virtual machine. For this assignment, an image is provided. Start by downloading and unzipping the following file:

http://people.cs.pitt.edu/~jmisurda/teaching/cs1550/qemu/qemu.zip

If you are running a windows or linux machine, you need to use either qemu-win.bat or linux_boot.bin to launch the qemu image.

If you are running under macOS in addition to downing the image file above, you also need to download this package: http://www.kju-app.org/

Once this is installed you can setup a virtual machine using the image file in the zip file (tty.qcow2). From this point on, there should be no differences between the mac and linux/windows version.

When you boot the virtual machine the first prompt will ask you to select a kernel to boot (red screen). There are two kernel installed in the image (original and development). The original one is really a backup in case your development kernel doesn't boot (ie: bad things happened). Through this process, you will be updating the development kernel to your newly compiled version and rebooting the virtual machine to run your kernel.

The login for this virtual machine is:

user: root
password: root

Copying the Kernel Files to QEMU

At this point you should have QEMU running and a newly compiled kernel ready to be executed. To copy the kernel to your virtual machine, you will need to download two files from the new kernel that you just built. The kernel itself is a file named bzImage that lives in the directory linux-2.6.23.1/arch/i386/boot/. There is also a supporting file called System.map in the linux-2.6.23.1/ directory that tells the system how to find the system calls.

To copy these files from the thot machine into your virtual machine use scp. These two commands below should do the trick:

scp USERNAME@thot.cs.pitt.edu:/u/OSLab/<YOUR_USERNAME>/linux-2.6.23.1/arch/i386/boot/bzImage .

scp USERNAME@thot.cs.pitt.edu:/u/OSLab/<YOUR_USERNAME/linux-2.6.23.1/System.map .

Installing the Rebuilt Kernel in QEMU

Once the new kernel is downloaded into the QEMU environment, you **must** install it. To do this, you need to make sure you are either logged into the QEMU virtual machine as root or you **su** (super user) to execute the following commands:

```
cp bzImage /boot/bzImage-devel
cp System.map /boot/System.map-devel
```

You must respond "y" to the prompts to overwrite the existing files. Please note that you are replacing the -devel files, the others are the original unmodified kernel so that if your kernel fails to boot for some reason, you will always have a clean version to boot QEMU.

You need to update the bootloader when the kernel changes. To do this (and you need to do it every time you install a new kernel), you, as root, type:

```
lilo
```

lilo stands for LInux Loader, and is responsible for the menu that allows you to choose which version of the kernel to boot into.

Booting into the Modified Kernel

You are now ready to boot into your newly created kernel. As root, you simply use the reboot command to cause the system to restart. When LILO starts (red menu) make sure to use the arrow keys to select the linux(devel) option and hit enter (the mouse does not work). Hopefully, the new kernel will boot properly and you will be able to login to the machine.

Adding system calls

As stated above, to add a new syscall to the Linux kernel, there are three main files that need to be modified:

1. linux-2.6.23.1/kernel/sys.c

This file contains the actual implementation of the system calls. To add a hellokernel syscall you need to add the following code to the end of the file.

```
asmlinkage int sys_hellokernel( int flag ) {
   printk("Your Kernel Greets you %d Times! \n", flag);
   return 0;
}
```

This is just a simple sys call that prints out the number passed into the call. As mentioned above, printk() is just a special "kernel" version of printf.

2. linux-2.6.23.1/arch/i386/kernel/syscall_table.S

This file declares the number that corresponds to the syscalls. As we have discussed in class each system call has a corresponding number, note in this version of the kernel there are 325 system calls all listed in this file. You will be adding the 326th system call or syscall number 325 (we all know CS folks start counting at zero!). To do add the new system call, just add the following line to the end of this file:

```
        .long sys_hellokernel
```

3. linux-2.6.23.1/include/asm/unistd.h

This file exposes the syscall number to C programs that wish to invoke it. First, you need to tell it the name of your system call and its number. To do this, you need to add the following statement below the block of "defines", near line 333.

```
#define __NR_hellokernel    325
```

Next, next you need to update the number of system calls by updating the following line.

```
#define NR_syscalls 326
```

Implementing a Program to Make the syscall()

After you have rebooted into your newly created kernel with your hellokernel syscall, the next step is to test it out. To do this, you need to make use of the syscall system call. Write the following C code on the **thot** machine.

```
#include <stdio.h>
#include <errno.h>
#include "asm/unistd.h"

int main(int argc, char ** argv) {
   printf("Calling ... \n");
   syscall(__NR_hellokernel, 42);
}
```

The next step is to compile the above program into an executable code. However, creating the code using gcc will result in undefined symbol error. This is because the compiler will include the <linux/unistd.h> file which is part of the kernel version that thot.cs.pitt.edu is running. Recall that the default unistd.h is not the one that we changed. In order to use the proper file, you instead need to direct gcc to import the new include files, using the -I option:

```
gcc -m32 -o testsystemcall -I /u/OSLab/<YOUR_USERNAME>/linux-
2.6.23.1/include/testsystemcall.c
```

This assumes your filename is testsystemcall.c and the output of this compile will be the executable file testsystemcall. Next you need to copy this file into the QEMU instance, in order to be able to execute it from there, and see your newly created system call hard at work.