Phase 3: User Processes and System Calls
Complete program due: Friday October 23rd, 9:00 pm

1.0 Introduction

The kernel constructed in Phases 1 & 2 provides valuable system services to the levels above it, but is inappropriate for direct use by user processes. For example, if a user process calls **`waitDevice`** directly, there is no guarantee that the process will get the interrupt associated with its I/O operation, rather than another process's. This is why the kernel specification required that a process be in kernel mode before calling its procedures.

For the next level of your operating system, you are to design and implement a support level that creates user processes and then provides system services to the user processes. All of these services will be requested by user programs through the **`syscall`** interface. Many of the services that the support level provides are almost completely implemented by the kernel primitives. In addition, in phase 4, the support level be expanded to contain system driver processes to handle the pseudo-clock, terminal input, and disk I/O. These system processes run in kernel mode and with interrupts enabled.

You may use the kernel that you constructed in Phases 1 and 2. I will also make libraries available that I think are correct. To use this, link with *libpatrickphase1.a* and *libpatrickphase2.a* in */home/cs452/fall15/lib*. Note that Phase 3 will be graded using the phase 1 and 2 libraries that I am supplying, not yours.

Recall that your phase 2 started running a process named **`start2`** after it had completed initialization. Thus, the entry point for your phase 3 code will be **`start2`**. Use **`start2`** to initialize your phase 3 data structures.

The header file *phase3.h* can be found in */home/cs452/fall15/include*. You should include this header file in all of your C files for this phase.

2.0 System Calls

The following system calls are supported by phase 3. Executing a **`USLOSS_Syscall`** causes control to be transferred to the kernel **`SYSCALL`** handler, which in phase 2 terminated the program for every syscall. You must now fill in the system call vector for the 10 system calls you will implement in this phase. Do this as part of the initialization of phase 3 in **`start2`**.

System calls execute with interrupts enabled. You will need to use the messaging system from phase 2 to provide appropriate exclusion for shared data structures. Thus, you will not disable interrupts when implementing this phase!

USLOSS allows only a single argument to a system call, and no return value. Therefore, all communication between a user program and the support layer will go through a single structure; the address of this structure is the argument to the syscall. The structure is defined as follows in */home/cs452/include/fall15/phase2.h*:

```
typedef struct systemArgs
{
        int  number;
        void *arg1;
        void *arg2;
        void *arg3;
        void *arg4;
        void *arg5;
} systemArgs;
```

An interface that makes these calls look more like regular procedure calls is provided in */home/cs452/fall15/phase3/libuser.c*. The code for Spawn and Wait is provided as an example. You will need to complete similar code for the remaining 8 system calls.

A user process that attempts to invoke an undefined system call should be terminated using the equivalent of the Terminate system call. In phase 2, you had a **`nullsys`** function that halted USLOSS for system calls that were not implemented. In phase 3, you will need to write a replacement for the **`nullsys`** function, called **`nullsys3`**, that will terminate the process rather than halting USLOSS.

You should first populate the **`systemCallVec`** array by assigning **`nullsys3`** to every element. Then, fill in the 10 elements of the **`systemCallVec`** array for the 10 functions you will write. As an example, you would use:

```
systemCallVec[SYS_SPAWN] = spawn;
```

for your **`spawn`** function. Write similar lines for each of the other 9 function calls. Use the constants specified below (**`SYS_SPAWN`**, **`SYS_WAIT`**, etc.) rather than putting in numbers. These constants are defined in **`usyscall.h`**, which you should include in your code. The provided **`skeleton.c`** file has an include for **`usyscall.h`**.

2.1 Spawn (syscall SYS_SPAWN)

Create a user-level process. Use **`fork1`** to create the process, then change it to user-mode. If the spawned function returns, it should have the same effect as calling terminate.

Input

arg1: address of the function to spawn.
arg2: parameter passed to spawned function.
arg3: stack size (in bytes).
arg4: priority.
arg5: character string containing process's name.

Output

arg1: the PID of the newly created process; -1 if a process could not be created.
arg4: -1 if illegal values are given as input; 0 otherwise.

2.2 Wait (syscall SYS_WAIT)

Wait for a child process to terminate.

Output

arg1: process id of the terminating child.
arg2: the termination code of the child.
arg4: -1 if the process has no children, 0 otherwise.

2.3 Terminate (syscall SYS_TERMINATE)

Terminates the invoking process and all of its children, and synchronizes with its parent's **`Wait`** system call. Processes are terminated by **`zap`**'ing them.

When all user processes have terminated, your operating system should shut down. Thus, after **`start3`** terminates (or returns) all user processes should have terminated. Since there should then be no runnable or blocked processes, the kernel will halt.

Input

arg1: termination code for the process.

2.4 SemCreate (syscall SYS_SEMCREATE)

Creates a user-level semaphore.

Input

arg1: initial semaphore value.

Output

arg1: semaphore handle to be used in subsequent semaphore system calls.
arg4: -1 if initial value is negative or no more semaphores are available; 0 otherwise.

2.5 SemP (syscall SYS_SEMP)

Performs a "P" operation on a semaphore.

Input

arg1: semaphore handle.

Output

arg4: -1 if semaphore handle is invalid, 0 otherwise.

2.6 SemV (syscall SYS_SEMV)

Performs a "V" operation on a semaphore.

Input

arg1: semaphore handle.

Output

arg4: -1 if semaphore handle is invalid, 0 otherwise.

2.7 SemFree (syscall SYS_SEMFREE)

Frees a semaphore.

Input

arg1: semaphore handle.

Output

arg4: -1 if semaphore handle is invalid, 1 if there were processes blocked on the semaphore, 0 otherwise.

Any process waiting on a semaphore when it is freed should be terminated using the equivalent of the Terminate system call.

2.8 GetTimeofDay (syscall SYS_GETTIMEOFDAY)

Returns the value of USLOSS time-of-day clock.

Output

arg1: the time of day.

2.9 CPUTime (syscall CPUTIME)

Returns the CPU time of the process (this is the actual CPU time used, not just the time since the current time slice started).

Output

arg1: the CPU time used by the currently running process.

2.10 GetPID (syscall GETPID)

Returns the process ID of the currently running process.

Output

arg1: the process ID.

3.0 Phase 1 and 2 Functions

In general, in a layered design such as this one, you should not have to refer to global variables or data structures from a previous phase. For phase 3, you can make use of any of the phase 2 functions: **`MboxCreate`**, **`MboxRelease`**, **`MboxSend`**, **`MboxReceive`**, **`MboxCondSend`**, **`MboxCondReceive`**, and **`waitdevice`**.

For phase 3, you can use some (not all) of the phase 1 functions. In particular, you can make use of: **`fork1`**, **`join`**, **`quit`**, **`zap`**, **`isZapped`**, **`getpid`**, **`readtime`**, and **`dumpProcesses`**. You can not use: **`blockMe`**, **`unblockProc`**, **`readCurStartTime`**, and **`timeSlice`**.

You can not disable interrupts in phase 3. Instead, use mailboxes for mutual exclusion when necessary.

4.0 Initial Startup

After your phase 3 has completed initializing the necessary data structures, it should spawn a user-mode process running **`start3`**. This initial user process should be allocated **`4 * USLOSS_MIN_STACK`** of stack space, and should run at priority three. **`start2`** should then wait for **`start3`** to finish; **`start2`** will then call **`quit`**.

5.0 Phase 1 and 2 Libraries

We will grade your phase 3 using the phase 1 and phase 2 libraries that we supply. The provided phase 1 library will be named: *libpatrickphase1.a* and the provided phase 2 library will be named: *libpatrickphase2.a*. These libraries will be available in */home/cs452/fall15/lib*.

You may use either your earlier libraries or the ones the we supply while developing your phase 3. However, it will be graded using the phase 1 and 2 libraries that we will supply.

6.0 Submitting Phase 3 for Grading

Note: Programs are submitted via D2L only.

Before submitting to D2L, you will need to create a **`phase3.tgz`** file. There is a target in the provided phase1 **`Makefile`** (*/home/cs452/fall15/phase3/Makefile*) named **`submit`**. It is at the very end of the **`Makefile`**.

Doing make submit will create a **`phase3.tgz`** file that contains:

- The **`.c`** files for your phase3. This will include **`phase3.c`**, **`p1.c`**, and any other **`.c`** files that you have created. If you have created additional **`.c`** files, you should have already modified **`COBJS`** to include these additional **`.o`** files.
- The **`.h`** files for your phase1. This will include **`sems.h`**. If you have created other **`.h`** files, you should have already modified **`HDRS`** to include the additional **`.h`** file(s).
- The **`Makefile`**.

You will *not* submit the usloss library or directory. You will *not* submit the phase 1 library or the phase 2 library. We will put a link for usloss and the phase1 and 2 libraries into the grading directory after we extract your files. The same is true for the **`testcases`** directory — do not submit the testcases. We will add links for the **`testcases`** and **`testResults`** directories.

You are encouraged to create the **`phase3.tgz`** file, then copy it to a different area of your home directory, extract the file using **`tar xvfz phase3.tgz`** and do a compile to see if everything has been included.

Any file(s) that are missing from your submission that we have to request from you will result in a reduction of your grade!

To submit the **`phase3.tgz`** file to D2L

- Log on to D2L, and select CSc 452.
- Select "Dropbox".
- You will find *Phase3*, and *Phase3-late*. Only one of these will be active. Click on the active one — will be *Phase3* if you are turning in the work on time.
- You should now be at the page: "Submit Files - Phase3". Click on the "Add a File" button. A pop-up window will appear. Click on the "Choose File" button. Use the file browser that will appear to select your file(s). Click on the "Upload" button in the lower-right corner of the pop-up window.
- You are now back at "Submit Files - Phase3". Click on "Upload" in the lower-right of this window. You should now be at the "File Upload Results" page, and should see the message **`File Submission Successful`**.

D2L will send you a confirmation email after each item is placed in the Dropbox. This email is sent to your UA email (which is <yourUANetId>@email.arizona.edu). Retain these emails at least until you have received a grade for the assignment; it is your confirmation that you did submit the program(s).

You may submit phase3 more than once. We will grade the last one that you put in the Dropbox.

7.0 Groups

You can change groups at this time if you wish. If you choose to do so, send mail to "452fall15@cs.arizona.edu" giving us details of how groups are being changed.