Phase 3: User Processes and System Calls

1 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. Therefore, the kernel specification (phase 1 & 2) 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 capability that supports user processes and provides system services to the user processes. These system services are 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 will be expanded to contain system driver processes to handle the pseudo-clock, terminal input and output, 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 your phase3 code with **lib452phase1.a** and **lib452phase2.a** using the Makefile. Note that Phase 3 will be graded using the supplied phase 1 and 2 libraries.

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 **usloss/include**. You should include this header file in all your C files for this phase.

2 System Calls

The following system calls are supported by phase 3. Executing a syscall causes control to be transferred to the kernel **SYSCALL** handler, which in phase 2 terminated the program for every syscall (as implemented in the nullsys() function). 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 synchronization or exclusion when accessing shared data structures. Thus, you will not disable interrupts when implementing this phase!

USLOSS allows only a single argument to a system call, and there is 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 **usloss/include/phase2.h**:

```
typedef struct _sysargs

{

      int number;

      void *arg1;

      void *arg2;

      void *arg3;

      void *arg4;

      void *arg5;

} sysargs;
```

An interface that makes these calls look more like regular procedure calls is provided in **libuser.c**. The interface file should be translated and linked to the runnable test case files (see the provided Makefile). This file will make it easier for you to write test programs. Note that the test cases I will use to grade your submission are provided in the starter package. Please feel free to write your own test cases.

A user process that attempts to invoke an undefined system call should be terminated using the equivalent of the **Terminate** syscall.

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.

`Wait`

Wait for a child process to terminate. If the process is zapped while it is waiting, the process should be terminated using the equivalent of the **Terminate** syscall.

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.

`Terminate`

Terminates the invoking process and all its children and synchronizes with its parent's **Wait** system call. The child 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.

`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 the initial value is negative or no more semaphores are available; 0 otherwise.

`SemP`

Performs a “P” operation on a semaphore.

Input:

arg1: semaphore handle.

Output:

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

`SemV`

Performs a ”V” operation on a semaphore.

Input:

arg1: semaphore handle.

Output:

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

`SemFree`

Frees a semaphore.

Input:

arg1: semaphore handle.

Output:

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

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

`GetTimeofDay`

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

Output:

arg1: the time of day.

`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 process's CPU time.

GetPID

Returns the process ID of the currently running process.

Output:

arg1: the process ID.

3 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. You can make use of **fork1, join, quit, zap, is _zapped, getpid, readtime, and dump processes**. You cannot use **block_me**, **unblock_proc, read_cur_start_time, or time_slice**. You cannot disable interrupts in phase 3. Instead, use mailboxes for mutual exclusion when necessary.

4 Initial Startup

First, your phase 3 should implement the routine **start2** to initializing the necessary data structures. Note that **start2** 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*. The **start2** function should then wait for **start3** to finish; **start2** will then call **quit**.

5 Phase 1 and 2 Libraries

I will grade your phase 3 using the provided phase 1 and phase 2 libraries. The provided phase 1 library will be named: **lib452phase1.a** and the provided phase 2 library will be named: **lib452phase2.a**. These libraries are available to download at the course D2L website. You may use either your earlier libraries or the ones I supply while developing your phase 3. However, it will be graded using the supplied phase 1 and 2 libraries.

6 Submitting Phase 3 for Grading

For the turn-in, you will need to submit all the files that make up your phase 3. Design and implementation will be considered in the grading process, so make sure your code contains

insightful comments, uses variables and functions have reasonable names, contains no hard-coded constants, etc. You need to turn in a **README** file to help me understand your solution. Your **README** file should include information as below:

- Your group members and working system you used to develop your phase 3
- If it requires special handling to run your phase 3 code with test cases, please describe.
- For each test case, provide the output as well as a self-evaluation whether the test case is passed or not. For special case, you may argue that the test case is partially passed.

Your **Makefile** must be arranged so that typing 'make' in your directory will create an archive named **libphase3.a**. Your Makefile must also have a target called 'clean' which will delete any

generated files, e.g. .o files, .a files, core files, etc.