CS 452 (Spring 25): Operating Systems

Phase 3 - User Processes and System Calls

due at 7pm, Fri 4 Apr 2025

1 Phase 3 Overview

In this Phase, you will implement mechanisms to support user-mode processes - specifically, system calls.

System calls allow a user-mode process to access basic system services. In some cases, such as the `Spawn()` and `Wait()` system calls, these syscalls will be (mostly) just wrappers around kernel-mode functions. In other cases, such as `SemP()/SemV()`, these will be more complex mechanisms. In real operating systems, some of the syscalls represent **very** complex operations - such as file system access, network sockets, etc.

Why do we have syscalls, instead of allowing the user to access kernel resources directly? We do this in part to protect permissions (for instance, a user-mode process should not be able to access mailboxes not associated with another process); in other cases, it's because we want to do more error checking.

1.1 New Error Checking Philosophy

Here's an important rule for syscalls: **a user-mode process should never be able to crash the entire OS,** no matter what it does wrong. In our previous Phases, our testcases represented kernel code, and so if they did something truly terrible, it was reasonable to crash the entire simulation. However, user mode programs should **never** be able to crash the OS.

Therefore, your system call code must implement careful error checking (sadly, this might duplicate some of the error checking that your kernel mode code already does); you must make sure that the parameters passed to the kernel mode code will not crash the system.

Of course, some sorts of minor errors don't crash the system; the kernel mode code simply returns an error value. For those, you don't have to duplicate the check - you simply pass the error status back to the user.

1.2 New Limitations

Starting in Phase 3, you **must not disable interrupts.** Now that you have a working Mailbox system (from Phase 2), you will use mailboxes (instead of interrupts) to provide yourself with mutexes.

Thus, in Phase 3, while you are free to read the PSR, you should **never set** it, with one exception: in your trampoline for user-mode processes, you will **disable kernel mode,** before you call the user-main function.

Similarly, you **must not call** `blockMe()` or `unblockProc()`; if you want to block a process, use a mailbox.

1.3 Interaction with Phases 1,2

Your code in this Phase may utilize any of the functions that were defined for Phases 1 and 2 - as wells as the `systemCallVec[]` array. You will **not** provide your own implementation; instead, you will use the libraries which I have provided for you.

You will not be able to access any of the private data or helper functions of the previous Phases - the only thing that you can do is to call the functions that I defined in the previous specs.

2 How Does a System Call Work?

From our discussions in lecture, you already know how a system call works, and why: a system call allows user mode code to jump into the kernel, but at a well-controlled location (not as simple as a function call). But in this Phase, it's important to understand the details of how this works.

The problem is that system calls are **ugly** interfaces - they are actually a form of interrupt, not a function call, and that means that it's hard to pass parameters safely. USLOSS provides the following function (which can be called from user mode), which will trigger a system call:

```
void USLOSS_Syscall(void*)
```

When the user calls this function, USLOSS will jump into kernel mode, and drive a `USLOSS_SYSCALL_INT` interrupt. The second parameter to the interrupt - the `void*` - will be whatever value that the user passed as the parameter to `USLOSS_Syscall()`.

In our system, we are going to use this parameter to point at a **real data structure,** sitting on the stack of the user mode process. The struct will be of type `USLOSS_Sysargs`, which is defined in `usyscall.h` (one of the headers provided by USLOSS). This struct will be used both for (a) passing arguments from the user mode, into the kernel; and (b) returning values back from the kernel into user mode.

But ordinary programmers **hate** this sort of interface - they much prefer an ordinary function. And so, in addition to the system call **handler** code (running in kernel mode) system calls usually include **library** code, which runs in user mode, and which provide an attractive, function-call based interface to the low-level system call mechanisms. In this Phase, **I have provided the user library for you** - you are not expected to write or to change it. Instead, you will focus on your kernel-mode implementation.

(spec continues on the next page)

2.1 Sequence of Events

Here is the sequence of events in a complete system call:

- **User Mode Code** - provided by testcase

  The user mode code decides that it wants to make a system call, so it calls a library function, such as `Spawn()`, that performs that action.

- **User Library** - provided by me, in Phase 3

  The user-mode library code creates a `USLOSS_Sysargs` struct, on the stack. It sets the `number` field, which will tell the kernel which syscall is requested. It will also set some number of the `arg*` fields.

  Once the struct is initialized, it calls `USLOSS_Syscall()`, passing it the pointer to the struct. USLOSS immediately drives an interrupt, which throws the process into kernel mode.

- **Interrupt Handler** - provided by Phase 2

  The System Call Interrupt handler, which you wrote in Phase 2, will receive the interrupt. It will cast the `void*` parameter to the proper struct, and read the `number` field; it will then use that number to index into the `systemCallVec[]` array.

  It will call the system call handler, for the system call requested, passing it the pointer to the struct full of args.

- **System Call Handler** - **New in Phase 3!**

  Your code will read the parameters from the `arg*` fields of the struct. You will perform whatever logic is required (almost certainly, you will be calling functions from Phases 1,2). You will then write to the various `arg*` fields to return values to the user.

  You will then return, which will return back, through the various handlers, until USLOSS returns back into the user mode code.

- **User Library** - provided by me, in Phase 3

  When `USLOSS_Syscall()` returns, the user library will "unpack" the returned value(s) that it received from the kernel, and deliver them to the user, in whatever way they expected them.

3 Overview of New System Services

In Phase 3, you will be providing, to the user mode code, two basic types of services.

First, you will provide some process control services - but they will be better designed for the way that real users would likely use them. `Spawn()` works pretty much like `spork()`, and `Wait()` works pretty much like `join()`. The big change is in `Terminate()`. While `quit()` would report an error if any child

processes remained, `Terminate()` will instead wait for each one. That is, it will call `join()`, over and over, until `join()` returns -2; only then will it call `quit()`.

Second, you will provide a **semaphore** system, using mailboxes for synchronization and wakeup mechanisms. You will provide the functions `SemCreate()`, `SemP()`, and `SemV()`. (You will not have to implement the `SemFree()` syscall; maybe I'll add it in a future semester.)

4 Tips and Tricks

Before we get into the details of the Required Functions, let's review a couple tools that you will likely find handy!

Casting to a function pointer
The syntax for function pointers in C is weird, there's no doubt about it! Remember that the syntax for declaring a function pointer variable looks like this, where `asdf` is the name of the variable:

```
int (*asdf)(int,char,void*) = ...
```

But how in the world might you **cast** a pointer, to a function pointer? The trick is similar to how casting to any other pointer works: look at the syntax for declaring the variable, and remove the variable name! So in C, you can cast a `void*` to a function pointer like this:

```
void *raw_ptr = ...
int (*asdf)(int,char,void) = (int(*)(int,char,void))raw_ptr;
```

Mailboxes as Mutex
We've talked about this in lecture, but I wanted to remind you: if you want to build a mutex out of a mailbox, the trick is to use a (1,0) mailbox (1 slot, length 0). You can then use `Send()/Recieve()` operations to lock/unlock the mechanism.

Of course, you might find it confusing to remember which operation is lock, and which is unlock - not to mention ugly! So I **strongly recommend** that you wrap up each operation in a helper function:

```
int feature_xyz_mutex_mailbox_num;
void feature_xyz_lock(void);
void feature_xyz_unlock(void);
```

(spec continues on next page)

5 Required Functions

In this Phase, while you are required to implement several system calls (which we'll detail later), there are only two functions that have specific names; they are the two bootstrap-related functions.

You must implement (see the previous Phases for documentation about what these processes do):

- `void phase3_init(void)`
- `void phase3_start_service_processes(void)`

6 Required Syscalls

Each system call, listed below, has two parts: the user-mode library, and the syscall handler in the kernel. Remember that the user-mode library has **already been defined,** in `phase3_usermode.c`; you will not need to write or change it. But I will show how the library works in the documentation below so that you can understand the functions that the testcases will be calling.

You can find the syscall constants, `SYS_*`, in a lesser-used include file provided by USLOSS, `usyscall.h` .

(spec continues on next page)

6.1 `int Spawn(char *name, int (*func)(void *), void *arg, int stack_size, int priority, int *pid)`

Creates a new process, which will run entirely in user mode. If the process returns, then the trampoline will automatically call `Terminate()` (still from user mode) on behalf of this process, passing as the argument the value returned from the user-main function.

The PID of the child is returned using an out parameter `pid`; the return value is 0 if the child was successfully created, and -1 if not.

System Call: `SYS_SPAWN`

System Call Arguments:

- arg1: address of the user-main function `func`
- arg2: parameter `arg` to pass to the user-main function
- arg3: stack size
- arg4: priority
- arg5: pointer to character string with the new process's name

System Call Outputs:

- arg1: PID of the newly created process, or -1 if it could not be created
- arg4: -1 if illegal values were given as input; 0 otherwise

(spec continues on next page)

6.2 `int Wait(int *pid, int *status)`

Calls `join()`, and returns the PID and status that `join()` provided. If `join()` returns -2 (meaning that the current process has no children), then this call will also return -2 (and the PID and status are undefined.)

System Call: `SYS_WAIT`

System Call Arguments:

- None

System Call Outputs:

- arg1: PID of the cleaned up process
- arg2: status of the cleaned up process
- arg4: -2 if no children; 0 otherwise

6.3 `void Terminate(int status)`

Terminates the current process, with the status specified. This function never returns. (Note that if the user-main function returns, then `Terminate()` must be called automatically, still from user mode.)

Unlike `quit()`, it is perfectly valid to call `Terminate()` while the process still has children; `Terminate()` will call `join()`, over and over, until it returns -2 - and then call `quit()`.

System Call: `SYS_TERMINATE`

System Call Arguments:

- arg1: status

System Call Outputs: **n/a** (function never returns

(spec continues on next page)

6.4 `int SemCreate(int value, int *semaphore)`

Creates a new semaphore object; returns (through an out parameter) an integer which is the "semaphore ID," for use in later function calls. The other parameter (an in parameter) is the initial value to store in the semaphore.

Returns -1 if the initial value is negative (or if no semaphores are available). Otherwise returns 0.

(You must never allocate more than `MAXSEMS`[1] semaphores. I recommend a fixed-length array, since `malloc()` is not allowed!)

System Call: `SYS_SEMCREATE`

System Call Arguments:

- arg1: initial semaphore value

System Call Outputs:

- arg1: ID of the newly created semaphore
- arg4: -1 if illegal values were given as input, or if no semaphores were available; 0 otherwise

6.5 `int SemP(int semaphore) / int SemV(int semaphore)`

Performs the "P" and "V" operations on a semaphore.

System Call: `SYS_SEMP, SYS_SEMV`

System Call Arguments:

- arg1: ID of the semaphore

System Call Outputs:

- arg4: -1 if the semaphore ID is invalid; 0 otherwise

6.6 Sem Free?

The `SYS_SEMFREE` syscall has been disabled for this project; you do not have to implement it.

(spec continues on next page)

[1] phase3.h

6.7 `int kernSemCreate(int value, int *semaphore) , int kernSemP(int semaphore) , int kernSemV(int semaphore)`

See: `phase3_kernelInterfaces.h`

The semaphore functions are useful tools, and you may want to make use of them in Phase 4. For that reason, you **must** implement kernel-side versions of each of the three semaphore functions (Create, P, V). Give them the names that I've defined.

These functions work the same way as the user-space versions, except that they must be called from kernel mode. I recommend that you use these as helper functions to implement the syscall handlers - so that you won't have to duplicate any code.

6.8 `void GetTimeofDay(int *tod) / void GetPID(int *pid)`

Three simple query syscalls, mapping to `currentTime(), getpid()` from Phase 1. The values are returned through out parameters, and each function returns `void`.

System Call: `SYS_GETTIMEOFDAY, SYS_GETPID`

System Call Arguments:

- None

System Call Outputs:

- arg1: the value returned

6.9 `void DumpProcesses()`

Calls the kernel function `dumpProcesses()`.

System Call: `SYS_DUMPPROCESSSES`

System Call Arguments:

- None

System Call Outputs:

- None

7 Turning in Your Solution

You must turn in your code using GradeScope. While the autograder cannot actually assign you a grade (because we don't expect you to match the "correct" output with perfect accuracy), it will allow you to compare your output, to the expected output. Use this report to see if you are missing features, crashing when you should be running, etc.