11001010
0011010
101111
01010
CS410/510
Languages & Low-Level Programming

Capabilities

Spring Term, 2016

This page last updated: May 11, 2016.

The exercises described on this page build on the ideas that you have developed in the earlier labs on context switching and paging, but also incorporate new ideas that were introduced in the Week 7 lecture about capabilities. You should make sure that you have completed the earlier labs, and reviewed the slides from the lecture before proceeding.

Step 1: Getting set up

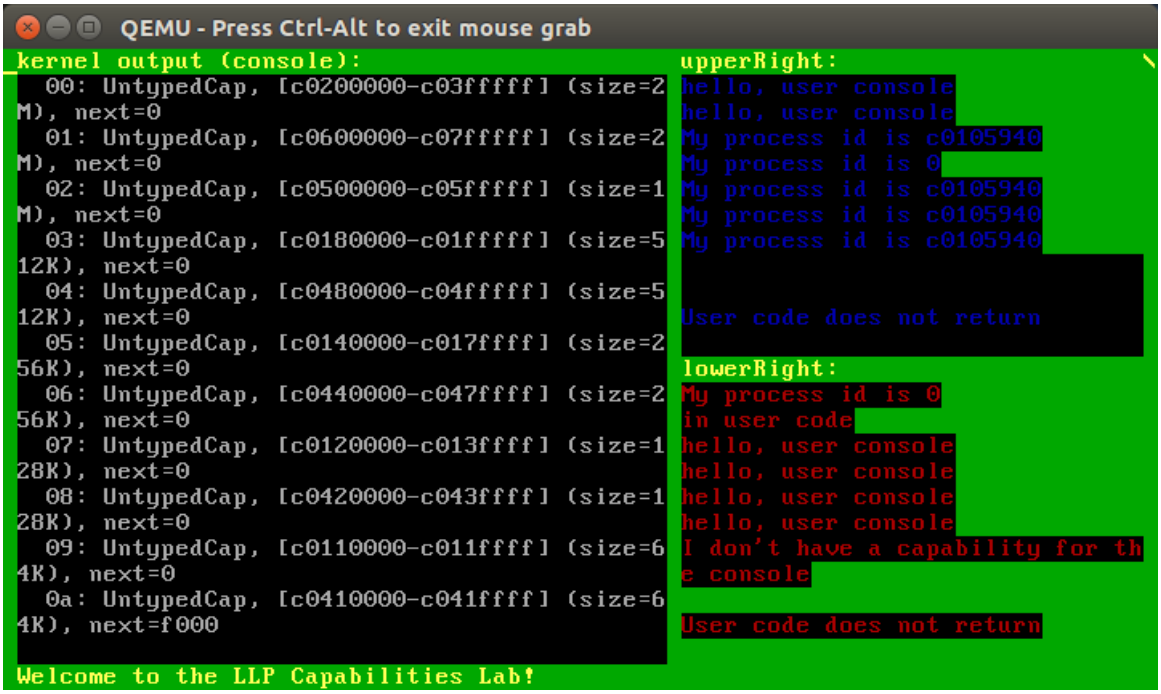
There are three new packages of software that you'll need to download, unpack, and build for this lab:

- A new version of the Window I/O Library, winio.tar.gz (also available via D2L or from the stash): Download a copy of this file into your `llp` directory; unpack it using `tar xvzf winio.tar.gz` (move the old version of `winio` to `winio.bak` first if you want to keep it, although the new version should be backwards compatible); and then run `make` in the new `winio` directory. The new version of this library:
    - Makes sure that the cursor is set to a sensible position when the window is configured without requiring an initial call to the clear screen function.
    - Provides a new version of `printf` that is able to print unsigned integers in a (sometimes) more readable form as a short decimal number followed by a suffix of either `G`, `M`, or `K` if the number is a multiple of 2^{30}, 2^{20}, or 2^{10}, respectively. For example, using this format, `4096` will be displayed as `4K` while `0xc0000000` will be displayed as `3G`. To use this new format, insert a `z` character (chosen as a mnemonic for "siZe") between the `%` and the numeric radix specified (typically a `d` for decimal output). For example, the call `printf("%zd\n", 3*(1<<16));` will display `192K`, which is probably a bit easier to read and understand than `196608`, its full decimal equivalent.

- A new library for user-level I/O via window capabilities, capio.tar.gz (also available via D2L or from the stash): Download a copy of this file into your `llp` directory; unpack it using `tar xvzf capio.tar.gz`; and then run `make` in the new `capio` directory. A brief description of this library was included in the capabilities lectures.

- The source code for the simple kernel and user-level demo with capability-based system calls that was the main focus of the capabilities lecture: This code is packaged as caps.tar.gz (again, also available from D2L or the stash) and should be installed in the same way as the previous two files: place the file in your `llp` directory; unpack using `tar xvzf caps.tar.gz`; and then run `make` in the new `caps` folder.

In fact, as you've likely guessed based on past exercises, you can use `make run` from within the `caps` folder to build and run the demo program in a single step. If you've installed everything correctly, you should now be looking at a window that looks something like this:



Note that there is a "spinner" in the top right hand corner that is updated by the timer interrupt handler. This is a simplified version of the clock exercise that was suggested as a stretch goal in the context switching lab. Its primary purpose here is to provide an indication that the kernel is still running, even when no output appears on screen. But those who did not get a working clock for the switching demo may find some useful hints about how to accomplish that by studying the implementation of this simple spinner.

Step 2: A guided tour of the `caps` folder

This lab draws on the techniques and ideas that have been introduced in the previous switching and paging labs. But with all the extensions and modifications that we've made in the course of those exercises, you're probably finding that the code base is getting a little tangled and hard to navigate. So we're going to take advantage of this new lab to do a little spring cleaning, reorganizing the code and breaking it in to smaller pieces with a more clearly defined purpose or focus. In this section, we'll provide a brief tour of the important files.

But before you go further, it's probably a good idea to do a `make clean` in the `caps` folder to clear out the compiled files so that we can focus on the source files; you can always regenerate the other files later by repeating the `make` or `make run` commands. Done that? Then let's proceed.

We'll start our tour in the `kernel` folder; I encourage you to follow along by reviewing each of the files in an editor (or using a tool like `less`, or similar), as you see fit:

- We'll start with the easiest ones: `Makefile`, `kernel.ld`, `memory.h`, `context.h`, and `hardware.h` are all pretty much the same as the versions of these files that you've seen before. There are a few extra lines in the `Makefile` to handle the new `.c` files described below, but otherwise there's not a lot of new material to see here.

- `init.S` is a (mostly unchanged) version of the startup code, written in assembly, from the previous labs. One small change that might be worth pointing out here, however, is a new macro called

`syscall` that simplifies the task of adding a new system call handler. Specifically, you want to add a new system call for a function called `XXXX`, then you should add a couple of lines of the following form after the definition of the `syscall` macro:

```
handleXXXX:
        syscall syscallXXXX
```

and also add an entry for the new system call in the IDT initialization code that looks something like this:

```
idtcalc handler=handleXXXX, slot=N, dpl=3
```

where `N` is the IDT slot/interrupt number that you'll be using for the new system call. Finally, of course, you'll need to add an implementation of the new system call; the `syscalls.c` file described below is likely a good place to put that code.

- `util.h` just provides definitions for functions that can be used to `halt()` the CPU (assuming you're in kernel mode); report a `fatal()` error; or test for a specific condition using `assert()`. You'll see some uses of `assert()`, in particular, at the start of the `initMemory()` function in `alloc.c` where it is used to perform some basic sanity checks and report an error to the kernel developer if something does not look right. Feel free to use this in your own code as seems useful!

- `alloc.h` and `alloc.c` implement the algorithms for calculating the initial list of untyped memory capabilities that cover all of the unused memory based on the memory map and header information that are passed on at boot time. Once created and sorted with the largest untyped region(s) first, the kernel allocates individual 4K pages from the regions at the end of the list and provides functions for donating entries in the list by moving them to an appropriate capability space slot. You will not be expected to understand or modify the code in these files, but you are welcome to take a peek and try to figure out how they work.

- `caps.h` and `caps.c` provide definitions and implementations for the different capability types that we are working with here. This includes: a unique integer code for each different type of capability; a structure describing the layout for that capability type; and associated test and set functions (all of this is in `caps.h`). The code in `caps.c` mostly provides functionality for printing out descriptions of individual capabilities or complete capability spaces (for debugging purposes), but also provides the main `alloc()` function for allocating a flex page from an untyped memory object.

- `paging.h` and `paging.c` provide definitions and implementations of functions for working with page directory and page table structures. You've seen most of this code before, but there is a completed version of `mapPage()` from the end of the first paging lab that you can compare with your own implementation. If you didn't quite get around to writing your own `mapPage()` just yet, fear not: you'll still have an opportunity to demonstrate your understanding of how these kinds of operations are written when it comes to implementing the `mapPage` and `mapPageTable` system calls later in this set of lab exercises.

- `proc.h` and `proc.c` are short files that provide the implementation for the data structure that we use to represent individual processes (`struct Proc`) as well as a `reschedule()` function for switching from one process to another. There isn't a lot of code here that you haven't seen before, but you might find it useful to look at these files if you need to make changes to the way that processes are created or organized.

- `kernel.c` contains the code for initializing the kernel and the initial user-level processes. You

might want to modify this code if you are trying to test specific kernel functions that, for example, are used to configure a capability space.

[**Quick quiz**: (not related to capabilities!) The `upperRight` and `lowerRight` variables that are defined in this file as global variables are only actually referenced within the `kernel()` function in this file. Why then would it be a mistake to declare them as local variables of the `kernel()` function?]

- `syscalls.c` all of the system call implementations have been collected in this one file so that we can compare them side by side. You might notice that there are some `TODO` items here, so perhaps you'll be coming back to look at this file in more detail quite soon ...

Phew, quite a lot to take in there! Fortunately, the changes to our `user` folder are a little easier to follow (it helps that we didn't have much code there to begin with):

- `Makefile` and `user.ld` are unchanged.

- `syscalls.h` is a simple header file that contains C function prototypes for the system calls that we want to be able to use in our user-level program. In other words, the definitions in this file describe how individual system calls are made available to user-level programs as if they were just regular function calls.

- `userlib.s` contains the assembly language initialization code for the kernel, as well as stubs for each of the system calls that are declared in `syscalls.h`. There are plenty of examples for you to work from here if you are trying to add a new system call. Just make sure that you are loading the system call parameters into the appropriate registers, and that you are using the right IDT slot/interrupt number for the `int` instruction; these must match the slot numbers you've used in `kernel/init.S`, or else bad things might happen ...

- `user.c`contains the C code for a simple user-level program. This is the place to add uses of system calls that you're testing. For the time being, it's important that we don't allow this user-level program to terminate because the kernel doesn't know how to handle that. The kernel also doesn't know how to respond to page faults just yet, so let's hope that our user-level programs are well-behaved. Of course, we might also want to address these problems in other ways to make a more robust system ...

That's a lot of information to digest. Feel free to take a break to let it all sink in, but also remember that you can always come back to check these details again later as you need them.

Step 3: Fixing those TODOs

As mentioned previously, there are a few `TODO` items in `kernel/syscalls.c` ... and now is the time to start filling them in! Essentially, what you are being asked to do here is to complete the implementations of some of the system calls that were described in the lecture. The supplied code already includes the code for system calls whose implementation was shown in the lecture, so you can use those as starting points/inspiration for the missing system calls. Each of the `TODO` items includes comments to describe what that particular system call is expected to do, so we will not waste space repeating that information here. All of the code that is needed to support these system calls in `kernel/init.S`, `user/syscalls.h`, and `user/userlib.s` has already been provided so that you can focus your attention on `kernel/syscalls.c`. But of course you're welcome to look at the other files if you want to check any

details there or to make sure you understand all of the steps that are needed to get from user-level code to the implementation of a system call.

Here is a list of the various `TODO` items in a suggested order of working through them (although any order is fine, and you could also skip to some of the later exercises if you want a break part way through this list and then come back to finish off these items later on):

- `syscallCls()` and `syscallSetAttr()` are used to perform operations on a `WindowCap`, assuming that the caller has the necessary permissions ...
- `syscallCapclear()` is used to clear entries in a capability space. Beware that, if you remove the last capability to a given resource, then there might not be any way to get it back!
- `syscallAllocPage()` and `syscallAllocPageTable()` are used to allocate objects from an untyped memory object. These operations follow a very similar pattern; we might want to integrate these into a single function later on to avoid a profusion of system calls.
- `syscallMapPage()` and `syscallMapPageTable()` allow a process to extend its own address space by adding new mappings using previously allocated page table and page objects.

Step 4: Adding a new system call to an existing capability type

Add a new system call, invoked as `remaining(ucap)` from user-level code, that can be used to determine how much unallocated memory is left in the untyped capability specified by `ucap`. That description makes it sound very simple, but it might take a little bit more work to get this up and running!

Step 5: Adding a new type of capability

Add a new capability type, `TimerCap`, and an associated system call, `getTicks(tcap)`, that a user-level process can invoke to access the current timer value via a reference, `tcap`, to a `TimerCap`. In the distributed code, the timer is stored in the `ticks` variable inside the code for the timer interrupt handler; you'll likely need to make this a global variable so that your system call implementation can access it.

Some operating systems block access to accurate timer information from user-level processes to prevent timing attacks. One way to address this is to limit the resolution of the timing information. Extend your implementation of `TimerCap` to include a number of bits that will be zero in every timer reading that is returned through that capability. For example, if the number of bits is set to 3, then actual timer values of 0, 1, 2, 3, 4, 5, 6, and 7 will all be reported back to the holder as 0. The holder will still be able to see that the timer is advancing, but will not have access to the highest available resolution.

Note: Remember that this is just an exercise in using capabilities: I'm not really proposing it as a serious practical solution to avoid timing attacks, nor am I claiming that timing attacks are a concern for us in this lab, especially because the tick count in this implementation does not provide a particularly high resolution to begin with!

Step 6: Improving an existing operation

Using the `WindowCap` capability type, we can print a string of characters on an output window using a sequence of system calls, one for each character. While this is enough to get the job done, it is not very efficient or practical to use so many system calls for such a simple task! Wouldn't it be better if we could use a single system call to pass and display a whole string of characters at once?

To implement this we need a region of memory that both the user-level program and the kernel can use to communicate with one another. From the kernel's perspective, we can handle this by adding a new field, `unsigned* buffer`, to each `struct Proc` object that is initialized to `null`. For the user-level side, we can provide a system call `allocBuffer(ucap, addr)` that can be invoked to allocate a page of memory (from the untyped memory specified by `ucap`) that is mapped into the address space of the user-level process at the specified address, `addr`. Of course, this operation will fail if there is a problem with the capability (wrong type, not enough space, etc.) or the address (in `KERNEL_SPACE`, already mapped, etc.), or if a `buffer` has already been allocated for this process. But if it succeeds, then a pointer to the newly allocated page will be saved in the `buffer` for the process.

The next step is to add a new system call, `printBuffer(wcap)` that will print the string in the process buffer in the window specified by the given `WindowCap`, `wcap`. A user-level process will typically invoke this system call after it has filled the buffer, via the address that it specified in the `allocBuffer()` call. On the kernel side, we have to check not only that `wcap` is valid (and, presumably that it includes the permission to printing a single character), but also that the invoking process has previously allocated a `buffer`. We'll assume that the user-level process marks the end of the string that it wants to be displayed by adding a null byte after all the characters, thereby following the time-honored (and highly error-prone) tradition of C. This creates another obligation for the kernel, because if the user-level program `forgets' to add that null byte, whether by mistake or design, then an over-eager implementation of `printBuffer()` might be tricked into reading past the end of the buffer page, and trigger a kernel page fault. That would be pretty embarrassing, so make sure your implementation stops before it goes past the end of the page! If you like, you could even add another field to the `WindowCap` capability to set a "maximum message length" value as an extra badge. Presumably the kernel should also be ready to ignore that value if it is more than the length of the buffer page...

The technique described here still has some potential performance problems because it forces the user-level process to copy data into the buffer area before making the call to `printBuffer()`. You could imagine an alternative approach that provides user-level processes with a system call `printString(wcap, addr)` where `addr` can be an arbitrary address that points to a string in the caller's address space. The kernel, of course, will be able to read the characters stored at that address. But again, our poor kernel will have to careful and protect itself against misbehaving user-level programs that supply an invalid `addr` that would cause the kernel to page fault if it tried to access that memory. Happily, the kernel also has access to the page directory and page table for the process, so it can check that the address is valid before attempting the operation. But this will take a little bit of extra work ...

Note that the `buffer` technique described above is inspired, in large part, by the approach that is used in seL4 to allocate an `IPC buffer' to threads that need to send or receive messages that are more than a few words in length, and hence cannot be stored completely in CPU registers.

Step 7: Let's retype that ... (a stretch goal)

You may have noticed that the various functions for allocating objects from an untyped memory area (e.g., `allocCspace`, `allocUntyped()`, `allocPage()`, etc.) all have a pretty similar structure. In addition, there are often situations where we need to make multiple allocator calls in a row to allocate a sequence

of objects. One way to tackle these problems is to collapse all of the allocators into a single system call of the form:

```
retype(ucap, type, bits, slot, num)
```

where:

- `ucap` is a capability to untyped memory.
- `type` is an integer code that represents the type of the objects and associated capabilities that we want to allocate. For example, we could use 1 for a `CspaceCap`, 2 for an `UntypedCap`, etc.. The numbers are arbitrary so long as distinct capability types use distinct numbers and both kernel and user-level code use the same mapping from numbers to capability types.
- `bits` is an extra field that can be used to specify the size of the objects to be allocated; this field is needed when we are allocating new `UntypedCap`s, but can be ignored for all other capability types in our current implementation.
- `slot` is a reference to an empty slot in a capability space where the capability to the first new object will be placed.
- `num` is the total number of objects that we want to be allocated by this system call. For the call to be valid, we must ensure that the untyped memory has enough free space for the requested number of objects, and we must also ensure that there are enough empty capability slots—starting at `slot` and continuing with `slot+1`, `slot+2`, and so on, as necessary—to hold all of the new capabilities.

Your task now, of course, is to implement this all-powerful allocator system call. It will take a fair bit of work, but then you will also be able to throw away all those close-to-duplicated `allocXXX()` functions, and it will also be much easier after this to add support for allocating new types of object. In addition, you won't need to rewrite all your existing user level code to use the new `retype()` call: instead, you can just write simple wrappers in `user/syscalls.h` that implement each of the original, more specialized allocators as special cases of `retype()`.

Once again, we should also acknowledge that the `retype()` system call is inspired by a system call of the same name in the seL4 microkernel.

Step 8: What's next?

Really? That was quite a lot of work, and you still want more? Of course, I'm glad to hear that, but I'd still suggest this might be a good time to take a break. In your idle moments, however, you might still want to be considering questions like the following:

- Wouldn't it be nice if there were a way to create new processes on the fly? We could use untyped memory to allocate space for the underlying process structures, but the `proc` array that we've been relying on so far has a fixed size. Maybe it's time to rethink that and allow an arbitrary collection of `struct Proc` objects to be linked together in a circular, doubly-linked list so that it is easy to add and remove individual elements? This would add `struct Proc* prev;` and `struct Proc* next` pointers to every process, but then we could implement round robin scheduling just by switching from `current` to `current->next`.

- Now, before we worry about creating new processes, we could provide a new system call that allows existing processes to terminate themselves, removing them from the list of current processes. This could be handled gracefully by providing a suitable system call. But it is probably also time for us to replace the old `nohandler` with some code that will terminate a process if it triggers an exception. We don't really want a user-level process to bring down the whole system just because it tries to divide by zero, or causes a page fault.

- This raises another issue: we could find ourselves with no runnable processes; that is, with nothing for `current` to point to! For now, we might just decide that its time for the kernel to terminate too. But, if we add even more features to our kernel, then it might just be that we need to wait for some new event—a keyboard or timer interrupt, for example—that will restart a previously suspended task or process. With that in mind, perhaps we'll want to have a special `idle process' that can be used when no other `current` process is ready to run ...

- Finally, we're ready to think about what it might take to allow the creation of a new process ...

Maybe some of this will be a task for another day ... :-)
