Paging, Part 2

Spring Term, 2016

This page last updated: April 27, 2016.

The exercises described on this page build on the code base that you have developed in Paging, Part 1. The starting point for those previous exercises included skeleton code including a number of `TODO` items. But, by the time you were done, almost all of those annotations should have been erased and you won't find any new `TODO`s to help you this time round. In general, I'm assuming that you're getting to be more familiar with finding your way around the code and that you'll be confident in modifying and extending it without quite as much direct guidance. However, the exercises on this page are quite challenging, and will take some time to work through, so please ask if you need a little extra help at any stage.

Step 1: Review

We'll start with a review of what was covered in the previous set of exercises. You should make sure that you have completed all of those steps, with all of the features summarized below, before continuing with this set of exercises. Specifically, as your starting point, you should have built a modified version of the `example-idt` code that has the following features:

- Boot code that creates an initial page directory, enables paging, and relocates to an address in the upper 1GB of the virtual address space.
- A set of macros for finding page boundaries relative to a given address (see Step 2 in the previous set of exercises for details).
- Initialization code that scans the memory map provided in the boot information from `mimgload` to find an unused region of physical memory running from `physStart` to `physEnd`.
- An `allocPage()` function that allocates a single page of memory, together with `allocPdir()` and `allocPtab()` functions that can be used to allocate memory for individual page directory and page table structures, respectively.
- A `mapPage()` function that can be used to create a mapping from a specified virtual address to a corresponding physical address.
- Some simple test code that allocates and initializes a new page directory structure that includes mapping for the kernel space in addresses over 3GB and a mapping for a 4KB page, starting at address `0xb8000` in the virtual address space, to the same address in physical memory.

The 4KB page mapping described in the last step will allow user mode code to access the video RAM. But, up to this point, we don't actually have any user space code that is ready to run! Fixing this will be our first big task in the following exercises. Once we've done that, however, we might start to worry about whether it is appropriate to allow all user mode processes to access video RAM. For example, this might allow different programs to interfere with or monitor one another by reading and writing portions of the video RAM that are supposedly outside their own "window". Fixing this will be our second task in what follows.

Step 2: Finding a user mode program to run

The boot information that is displayed when the `kernel` program starts running should indicate that one (or perhaps more) user programs has been loaded in to physical memory when the system was initialized. But, up to this point, we have not done anything to allow that program to be executed. For this to be possible, we'll need to add some appropriate mappings to our page directory.

The start and end addresses for the (first) user program should be available to you in `hdrs[7]` and `hdrs[8]` from the initial boot information. Maybe you should try printing out those addresses just to confirm that you can read them properly ... Don't forget, however, that those values will produce physical rather than virtual addresses, and that they will be accessed through a `BootData` structure whose address (and contents) were also specified by physical addresses. Now that we're running with the kernel at high virtual addresses, potentially without mappings for low virtual addresses, we should make some adjustments to ensure that we're reading the bootdata through kernel space addresses. For example, the main bootdata structure should be accessible at `fromPhys(struct BootData*, 0x1000)`. You should make corresponding changes to ensure that you are reading the headers and memory map fields using kernel space addresses too ...

Step 3: Adding some user mode mappings

Once you've found the region of memory where the user program is stored, you'll want to add some corresponding entries to the new page directory to ensure that there are appropriate mappings for all of the corresponding pages in the user code. This should be fairly easy to accomplish using your `mapPage()` function, and you can use `showPdir()` to check your results. But do remember that the start and end addresses of the memory where the user code is stored might not always correspond to page boundaries, so again you might need to make some adjustments. Specifically, if you want to make sure that all of the memory between some addresses `lo` and `hi` is accessible, then you will actually need to map everything from `pageStart(lo)` to `pageEnd(hi)`. As a further hint, remember that you can use a macro call `pageNext(p)` to find the address of the first page after `p`.

As a crosscheck (remembering that your output may look slightly different), the output on my system shows that the user program is loaded at addresses `[400000-4025bf]`, and after I've added the user space mappings for this, my new page directory looks something like this:

```
Page directory at c0406000
  [0-3fffff] => page table at c0407000 (physical 407000):
    b8: [b8000-b8fff] => [b8000-b8fff] page
  [400000-7fffff] => page table at c0408000 (physical 408000):
    0: [400000-400fff] => [400000-400fff] page
    1: [401000-401fff] => [401000-401fff] page
    2: [402000-402fff] => [402000-402fff] page
  300: [c0000000-c03fffff] => [0-3fffff], superpage
  301: [c0400000-c07fffff] => [400000-7fffff], superpage
  302: [c0800000-c0bfffff] => [800000-bfffff], superpage
  303: [c0c00000-c0ffffff] => [c00000-ffffff], superpage
  304: [c1000000-c13fffff] => [1000000-13fffff], superpage
  305: [c1400000-c17fffff] => [1400000-17fffff], superpage
  306: [c1800000-c1bfffff] => [1800000-1bfffff], superpage
  307: [c1c00000-c1ffffff] => [1c00000-1ffffff], superpage
```

In this case, we can see that the user code fits in to three pages, starting at address `0x400000`. Experiment

with changing the address at which the user program is loaded [Hint: this is specified in `user/user.ld`] and you should see corresponding changes in the way that the new page directory is populated. For example, what happens if the load address is `0x400eff` (close to the end of the first page after 4MB) or `0x402eff` (a couple of pages further on)?

Step 4: Running the user program in the new page directory

You probably have just one lingering `TODO` item near the bottom of the `kernel` code where some old code from `example-idt` to initialize and run the user program was commented out. Now might be a good time to reinstate that code and check that it works as you would hope and expect.

As a general rule, it would be good to delay switching to your new page directory until after all of the appropriate entries have been added. Reorder the statements in your code so that it doesn't make the call to `setPdir()` until immediately before the `switchToUser()` call. This likely won't make a difference in this situation, but it is a good practice in general: loading a new page directory will flush out old TLB entries and make sure that the memory management unit reads the most recent entries that you have installed in the new page directory and any associated page tables.

Step 5: Preparing to context switch

We're going to want to run multiple user programs at the same time, and that's going to require similar steps to what you did in the context switching lab to enable timer interrupts. If you don't remember what you did there, then you can either look at the original instructions or else take a peek at your code from that lab. As a quick summary, you'll need to make some changes:

- in `kernel/kernel.c`: add an include for `"hardware.h"`; insert a call to `startTimer()`; and define a suitable `timerInterrupt()` function.
- in `kernel/init.S`: add an entry to the IDT for the timer interrupt (hardware interrupt 0, so it goes in at slot 32), calling a new system call handler (I'd suggest that you call the handler something like `timerHandler`); then add the assembly code for your new handler (you can model that on the code for another system call) that ends by branching to the `timerInterrupt()` function that you defined in `kernel/init.S`.

There are quite a few fiddly details to get right here, so proceed carefully, and be prepared to edit the revised code if it doesn't compile first time. If you didn't get as far as adding a clock display in the context switching demo, then you might at least want to arrange for the timer interrupt handler to print an occasional `"."` on the screen so that you can be sure that the timer is running. But make sure you don't actually try to context switch just yet: we'll need to have more than one context first!

Step 6: A simple process abstraction

In our original work with context switching, it was enough just to package up the registers associated with each different user program in a `struct Context` data structure. But to prevent interference between different user space programs, we're going to run each one in a different address space: in other words, we're going to need a different page directory for each user program that we run. With this in

mind, it makes sense to define a new data structure to store all of the information that we need for each user program:

```
struct Process {
    struct Context ctxt;
    struct Pdir*   pdir;
};
```

(You might think that we could just add the `pdir` field to our `Context` structure. That would work right now, but having a separate `Process` structure will probably work out better later on when we start adding more fields for each process.)

As a relatively simple step, you should update your code to work with a `struct Process` object called `proc`, containing both a context and a page directory, instead of the original `user` object (which was a `struct Context`). In particular, this will mean changing every reference to `user` in the previous version of your code to a reference to `proc.ctxt`. It would also be a good idea to initialize the `pdir` field of `proc` to the `newpdir` that you've worked hard to create, and then use `proc.pdir` when you activate the new page directory.

Step 7: Creating multiple processes

In this step, we're going to create two separate processes, each of which can run its own copy of the user-level program. Unfortunately, the `BootData` headers tell us where the code and data for the user program begins and ends, but they don't provide any more details about which parts of that memory correspond to `.text` sections (i.e., program code that could potentially be shared between multiple instances of a user program) and `.data` sections (which would require a separate copy for each program). For this reason, we'll need to create two completely separate copies of the user program so that each one can have its own `.data`.

I suggest that you approach this in multiple steps:

- First, write a `copyRegion()` function that will make a fresh copy of the pages containing the memory between two given physical addresses, `lo` and `hi`. The basic definition for this function might look something like the following:

```
unsigned copyRegion(unsigned lo, unsigned hi) {
    // Check that lo<hi, and take appropriate steps to
    // ensure that lo and hi correspond to suitable page
    // boundaries.

    // Figure out if there is enough memory left in the
    // pool between physStart and physEnd to make a copy
    // of the data between lo and hi.

    // Make a copy of the data.  You'll need to convert
    // between physical and virtual addresses in some way
    // here, but it would be nice to avoid having to do
    // that on every loop iteration ...

    // Update physStart as necessary.

    // Return the physical address of the start of the
    // region where the new copy was placed.
```

```
}
```

Your implementation of this function will have some strong similarities to the code that you wrote for `allocPage()` except that `copyRegion()` will often need to allocate more than one page of physical memory at a time. By all means add some `printf()` calls in your code so that you'll be able to check that your `copyRegion()` function works correctly when it is executed.

- Next, you can write a function that will allocate a new page directory that includes appropriate mappings for the user program. Something like the following would likely be a good place to start:

```
struct Pdir* newUserPdir(unsigned lo, unsigned hi) {
  struct Pdir* pdir = allocPdir();
  unsigned phys = copyRegion(lo, hi);

  // Add a mapping for video RAM

  // Add mappings to ensure that the region between lo and
  // hi in the new address space is mapped to the appropriate
  // portions of the pages starting at address phys.

  return pdir;
}
```

You should be able to adapt some of the code that you wrote previously in Step 3 for this; the difference now is that you'll be using different virtual and physical addresses for the extra mappings: you'll be mapping the region from `lo` to `hi` to pages starting at the physical address that is returned by your `copyRegion()` function.

- Now you can write a function that will initialize a given `struct Process` using appropriate `lo` and `hi` values to initialize the page directory, and a suitable `entry` point address to initialize the context. Try the following skeleton as a starting point:

```
void initProcess(struct Process* proc,
                 unsigned lo,
                 unsigned hi,
                 unsigned entry) {
    // TODO: fill this in ...
}
```

After all this, you you should be able to initialize two distinct `struct Process` objects and use either one to start a user program. A good way to approach this would be to define an array of two `Process` structures, as well as a pointer to the `current` process that looks something like this:

```
struct Process  proc[2];
struct Process* current;
```

Be sure to initialize `current` to `proc+0` or `proc+1`, and then modify the rest of your code to use `current` rather than `proc`. Note that, in places where you previously wrote something like `proc.blah`, you will now need to write `current->blah`, reflecting the fact that `current` is a pointer to `struct`.

(It might be difficult to tell exactly which of the two programs is running from the output on screen, but you'll know by looking at the code ... or perhaps by inserting some extra `printf()` calls ...

Step 8: Context switching, at last

Now you have multiple `Process` objects and a timer interrupt signal, you should be able to get the two user programs running together at the same time, context switching from one to the other on a regular basis. Don't forget that switching between processes doesn't just require a change of context any more ... it also requires a change of page directory: you should use a call that looks something like the following to accomplish this:

```
setPdir(toPhys(current->pdir));
```

Bad things will happen if you forget to translate the virtual address in `current->pdir` in to a physical address that is suitable for loading in to the `CR3` register. Just to reinforce that point, see what happens when you leave out the `toPhys()` call. Wouldn't it be nice if the compiler printed an error message and prevented us from running the program when we make an error like this?

Step 9: Protecting access to video RAM

At this point, you should have two user processes running, but it might be hard to tell because both of them are writing to the same region of video RAM. This highlights the fact that our two programs are not quite as *protected* from one another as we might hope. Although neither process has access to the main code or data sections of the other, they can both read and write the same locations in video RAM.

We could avoid the problem of having both user programs using the same window on screen by arranging for some additional information to be passed to the user programs when they are first run. (For example, we could provide some parameters to each process, such as the coordinates for its video window, by writing appropriate values in its initial `Context`.) This, however, would not do anything to prevent the two processes from communicating or interfering with one another via the video RAM.

One way to ensure protection is to move the functionality for writing to video RAM inside the kernel, and to provide system calls that user programs can invoke to produce output on the screen. It will take a couple of steps to make this work:

- First, we'll need a version of the original `simpleio` library that supports multiple output windows within a single program. The key idea is to package up all of the details about a single output window—including the position, size, attribute, and current coordinates—in a new `struct Window` object. You can download a copy of a suitable library from this link. Unpack this file in your main `llp` folder; run `make` inside the `winio` directory; and peruse the `winio.h` header file for details of the functions that it provides. (It's not very long!)

- Next, we'll need to modify the kernel to use `winio` instead of `simpleio`. This will require changes to the `Makefile` (replace references to the `simpleio` folder with references to the `winio` folder, and change library specification from `-lio` to `-lwinio`. You will also need to change the include lines at the top of `kernel/kernel.c` and `kernel/paging.c` to use `"winio.h"` rather than `"simpleio.h"`. Finally, you'll need to edit the code for `nohandler` in `kernel/init.S`. Previously, this code used a call to a function called `printf()`, but with `winio`, `printf()` is implemented using a macro that calls the function `wprintf()` with an extra argument called `console`. In other words, you'll need to replace the line:

  ```
      call    printf
  ```

with the two lines:

```
pushl   $console      # add the extra argument
call    wprintf       # call the new function
```

After these changes, your code should compile and run as before.

- Now we'll need to allow for different user programs to have different windows on the screen. You can handle this by adding a `struct Window` field called `win` to the definition of `struct Process` and initializing the window parameters in an appropriate way when the rest of the fields in each `struct Process` are initialized. For consistency with past examples, I would suggest using:

```
wsetWindow(&proc[0].win, 1, 11, 47, 32); // process 0 upper right
wsetAttr(&proc[0].win, 1);               // blue output

wsetWindow(&proc[1].win, 13, 11, 47, 32);// process 1 lower right
wsetAttr(&proc[1].win, 4);               // red output
```

  Output from the kernel should still use `printf()` and related functions, but output that is produced on behalf of user programs (e.g., in a system call) should use `wprintf()` with the address of the `Window` structure in the corresponding `Process` object. (In fact, `printf()` is implemented by a C macro that expands to a `wprintf()` call with the previously mentioned`console` as an extra argument.

- Now you should make sure that your kernel provides three system calls: one for outputing a single character within the user program's window; one for clearing the user program's portion of the screen; and one for setting the appropriate video attribute. I suggest that you use interrupts 129, 130, and 131 for these respectively, keeping the existing entry 128 for `kputc` in place. In each case, this will require: an entry in the IDT; an assembly language handler to save the context; and the main implementation. Following the pattern we have seen previously, the first two of those items can be handled by code in `kernel/init.S`, while the last can be handled by adding code in `kernel/kernel.c`.

- Now that you have provided system calls, you can delete the code that includes a mapping for the video RAM in every page directory structure you create. This will force user programs to use the kernel system calls for video output and prevent them from interfering with one another. At the same time, removing that mapping from each page directory will also prevent the kernel from accessing the video RAM at address `0xb8000`. Fortunately, the kernel can still access this region of memory through the high address mapping to physical memory (i.e., at address `0xc00b8000`, which is just `KERNEL_SPACE+0xb8000`. However, we do need to add one extra line at the start of the `kernel` main code to configure the `winio` library to work at this address:

```
setVideo(KERNEL_SPACE+0xb8000);
```

- Finally, you'll need to change the library that user programs use for video output: the old `simpleio` library won't work now that we've taken away the mapping for video RAM. To simplify this step, I've packaged up another variant of `simpleio` called `userio` that you can download, unpack (in your main llp directory), and run `make` in the new `userio` folder. Once again, you should peruse the code to get a brief idea about how it works and then make the appropriate changes to the code in the `user` directory.

Although there is nothing very deep or new in the above steps, there is quite a lot of work to do, and hence quite a lot of potential for making mistakes. It might take a little while to get this working properly,

but keep going, ask for help if necessary, and you'll get it up and running properly in the end!

When everything is finally working as you'd expect, try experimenting with the code in `user/user.c`; for example, you might want to add more loop iterations, and perhaps insert a (nested) loop inside the main loop just so you can slow things down enough to see the effect of having two processes run at the same time. And you can also experiment with the `setAttr()` function from the `userio` library to change the color of the output text. One final tweak: perhaps you should arrange for output that appears in the kernel window but originates from a user mode program (via the `kputc` system call) to be displayed in a different color so that there is no confusion about which parts of the output in the kernel window are not part of the kernel's own output.

Step 10: Pause for reflection

Congratulations: this was a long and arduous journey ... but you made it!

It's appropriate to pause at this point and enjoy the satisfaction of reflection on what you have accomplished: Specifically, you now have a very simple operating system that allows you to run two distinct copies of the same user level program. The two programs are protected from one another because neither one has access to the memory (either code or data) that the other is using.

As you continue to reflect on this, however, you might start to wonder whether it's a good idea to include all of the functionality for video RAM output as part of a kernel ... continuing in this way will ultimately lead us to a so-called *monolithic kernel* that has an unnecessarily large trusted computing base (TCB) because it includes a lot of code that doesn't strictly need to be executed in kernel mode. We don't have a mechanism for IPC between processes here, but perhaps you'll start to ponder how such a feature might be used to move the functionality for output on screen to a new, user-level process that has exclusive access to the video RAM at physical address `0xb8000` thanks to a special kernel mapping ... but all this sounds like an exercise for another time, don't you think?
