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

Paging, Part 1

Spring Term, 2016

This page last updated: April 19, 2016.

The code for this set of exercises is a modified version of the `example-idt` codebase that you worked on in the context switching lab; the modifications in this file include some changes to initialize the paging mechanisms (as described in the Week 4 lecture) and the addition of some TODO notices to guide you through these exercsies. You can download a copy of this code by clicking on this link, or by taking a copy from the stash or from D2L. However you obtain your copy, you should unpack it (using `tar xvzf paging1.tar.gz`) in your main llp folder (i.e, the one that is shared between your LinuxLab account and your llpVM virtual machine). Because there are some fairly complicated steps in this work, we'll be splitting these exercises over two weeks:

- Week 1: Implement basic support for paging, including functions to support allocation and initialization of paging structures (page directories and page tables).
- Week 2: Implement context switching between multiple user programs, each running in a distinct virtual address space.

All of the work for this first week will be done on the files in the `kernel` subdirectory, but you may wish to keep one window open in the main `paging1` directory (so that you can `"make run"` for the purposes of testing), and another in `paging/kernel` so that you are in the most convenient place to edit the files.

The provided code should compile and run as supplied, but it is incomplete and has multiple sections with "TODO" comments describing code that you will need to write. This document is designed to be read in conjunction with those source files and annotations. You are strongly suggested to follow the specific sequence of steps described below to complete these tasks (augmented with your own tests along the way, as you see fit), but you may choose to proceed in a different way if you prefer.

As you work on these exercises, you'll probably want to remind yourself about the bit layout patterns for page directory and page table entries. Here is a snapshot of the table in the Intel documentation that might be useful for that purpose (clickable for a full size version):

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Address of page directory[1]																				Ignored							PCD	PWT	Ignored			CR3
Bits 31:22 of address of 4MB page frame										Reserved (must be 0)							Bits 39:32 of address[2]			PAT	Ignored			G	1	D	A	PCD	PWT	U/S	R/W	1	PDE: 4MB page
Address of page table																				Ignored				0	Ign	A	PCD	PWT	U/S	R/W	1	PDE: page table
Ignored																															0	PDE: not present
Address of 4KB page frame																				Ignored				PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored																															0	PTE: not present

Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging

Ok, let's get started!

Step 1: Review

Review the supplied code, particularly the assembly language sequence at the start of `kernel/init.S` that enables the IA32's paging mechanism. Much of this should look familiar from the code that we have discussed in the Week 4 lecture. The code that creates the initial page directory is particularly relevant (i.e., worth your time and careful study) because you'll be doing something very similar in C later on ...

Run the program. How can you tell that paging has been enabled? (Hint: the clue you're looking for is just one character ...)

You'll notice that the program also displays a textual description of the initial page directory that is used by the code in `kernel/init.S` when the memory management/paging support is first turned on. The output looks like this:

```
initial page directory is at 0xc0102000
  Page directory at c0102000
    0: [0-3fffff] => [0-3fffff], superpage
    1: [400000-7fffff] => [400000-7fffff], superpage
    2: [800000-bfffff] => [800000-bfffff], superpage
    3: [c00000-ffffff] => [c00000-ffffff], superpage
    4: [1000000-13fffff] => [1000000-13fffff], superpage
    5: [1400000-17fffff] => [1400000-17fffff], superpage
    6: [1800000-1bfffff] => [1800000-1bfffff], superpage
    7: [1c00000-1ffffff] => [1c00000-1ffffff], superpage
    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
```

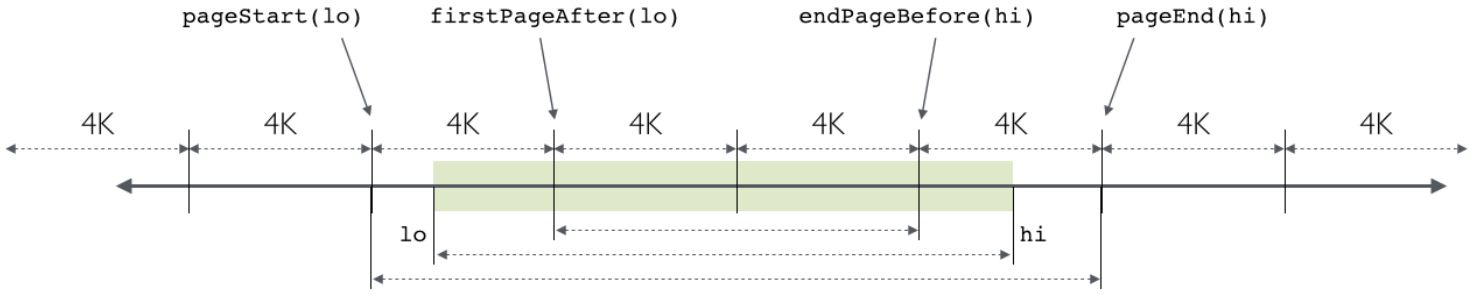
This output (produced by the `showPdir()` function that is defined in `kernel/paging.c`) shows that the initial page directory includes mappings for a total of sixteen superpages: This includes the first 8 slots of the page directory (covering addresses `0-0x1f_ffff`, which corresponds to 32MB of physical memory, or 8 superpages, each 4MB in size), and then another eight entries starting at `0x300`, which maps the same 32MB range of physical memory addresses (shown by the ranges on the right of the `=>` symbol) in to memory with the virtual addresses starting at `0xc000_0000`, which is just the `KERNEL_SPACE` start address that was describe in the Lecture. Indeed, you should recognize the above as a textual description of the initial page directory described in the lecture that maps the same region of physical memory in to virtual memory at two different addresses: a 1:1 mapping allows us to access these memory locations using low virtual addresses that correspond exactly to the underlying physical addresses, while the second group of mappings provide access to the same memory using addresses in the kernel space region. Take time to make sure you understand what the text above is describing: we're going to be creating and manipulating new page directory structures as we proceed, so it's important to have a clear idea of what they describe!

Ok, it's time to move on. But a quick heads up before we go further: you're going to be doing quite a lot of work to get this code up an running properly. It won't be easy, and you'll deserve some credit. So open up `kernel/kernel.c` in your text editor and add your name at the top of the file! This will almost certainly be this easiest of all the changes that you make to the code during this lab, but it's always nice to start with something simple :-)

Step 2: Bit twiddling

For some practice in basic bit twiddling, implement the macros that are defined/commented out at the end of `memory.h`. The descriptions for each macro/function given there should provide all the detail that you need, and none of them require anything very complicated. (Although a clear head will be useful!) Test your implementations to make sure they work as you expect because you'll be relying on them soon. (Hint: you can always `#include "memory.h"` in a regular C program that you can run directly from Linux to help with your tests; you might even find a suitable program lurking in the `kernel` folder ...)

The following diagram may help to provide some graphical intuition for these bit-twiddling macros:



Specifically, the green region shown here represents a block of memory running from the address `lo` to the address `hi`, inclusive. The vertical ticks indicate page boundaries, spaced at 4K intervals. The smallest set of (full) pages that includes all of the memory in the green region runs from `pageStart(lo)` to `pageEnd(hi)`. Conversely, the largest set of full pages that is completely contained within the green region runs from `firstPageAfter(lo)` to `endPageBefore(hi)`. In this particular example, `firstPageAfter(lo)` is also the same as `pageNext(lo)` and `pageNext(pageStart(lo))`. Note, however, that `pageStart(a)` and `firstPageAfter(a)` are not always a whole page apart: if `a` is a multiple of 4K, then `pageStart(a) = a = firstPageAfter(a)`.

Step 3: Scanning the memory map

We're going to need to allocate some memory for page directory and page table structures. Study the code that prints out a summary of the memory map at the start of the `kernel()` function in `kernel/kernel.c` and then start on the first few `TODO` items in the body of that function. Our high level goal here is to identify a block of pages that we can use as a pool to allocate new page table or page directory structures (or other blocks of memory, in units of 4KB that start on a 4KB boundary). Be careful to note that a memory map entry spanning from `lo` to `hi` must be trimmed to the potentially smaller region between `firstPageAfter(lo)` and `endPageBefore(hi)` before it can be used for allocating whole pages of memory. (See the diagram in Step 2.)

Once this portion of your code is complete you will have set the `physStart` and `physEnd` parameters to span a contiguous block of pages that (a) fits within the available physical memory, and (b) does not conflict with any region of memory in which other code has been loaded at boot time.

The output that I got at this stage of the process was:

```
Will allocate from region [403000-1ffdfff], 29339648 bytes
```

Yours should look similar, but don't worry if it's not exactly the same: differences in the way we've coded our solutions could cause small differences in the output that you see here. Note that we've found a block of 29+MB here, which should keep us going for some time :-)

Debugging hints: Note that, for the purposes of testing, you can add a call to the `halt()` function at any point in your code; this will prevent execution from continuing to parts of the program that you are not ready to run just yet. (And if your program is "blowing up" for some unknown reason, you can experiment with adding calls to `halt()` to figure out how far it is getting before things go wrong!) Note also that you can insert `printf()` calls in your code to print out the values that it is working with; this can often be useful as a way to monitor your code and check that your algorithms are giving the results you expect.

Step 4: Allocation of pages

Now that you've identified a region of available pages, it might be time to turn your attention to the `allocPage()` function that appears near the top of `kernel/kernel.c`. The `TODO` notice should give you enough information to complete this task, but make sure that you are using physical and virtual addresses properly, converting between them using the `toPhys` and `fromPhys` macros as necessary. Note also that, if `physStart` holds the physical address of an unused page of memory on entry to `allocPage()`, then the next available page of memory will begin at `nextPage(physStart)`.

For testing, you could always try inserting a few calls to `allocPage()` at an appropriate point in the `kernel()` function and

then printing out the results. You should see output addresses that are (1) within the selected region of memory, (2) properly aligned to page boundaries (i.e., the least significant twelve bits should be zero), and (3) increasing with each subsequent call to `allocPage()`.

Step 5: Adding kernel space mappings to new page directories

Once you've completed `allocPage()`, you should be able to get past the lines of code in `kernel/kernel.c` that try to allocate and print out the new page directory, `newpdir`. But don't expect to see any entries in that new page directory just yet; we'll need to add these ourselves. This would be a good time to turn your attention to the `TODO` item in the `allocPdir()` function in `kernel/paging.c`. As hinted earlier, the C code that you'll need to write here has some strong similarities to (some parts of) the assembly code at the start of `kernel/init.S` ...

Once this is done, you should get some output that looks something like the following:

```
Page directory at c0406000
  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
```

Of course, this looks a lot like the initial page directory shown previously, albeit without the mappings at low addresses. Don't worry if your output looks ever so slightly different: this might just be the result of minor differences in the way we've written our respective implementations. That said, *it is important that this new page directory does not include any mappings for memory at addresses below `c0000000`*: this is necessary to avoid conflicts with any future user space mappings that we might want to make.

Step 6: Are you feeling lucky?

Head back in to `kernel/kernel.c` (if you dare :-), and enable the code that will switch to your newly created page directory. If things don't work quite as you might hope, try inserting a call to `halt()` immediately after the `setPdir()` call and then use QEMU's monitor to check on the status of the CPU (`"info cpus"`, for example). Then try moving the `halt` after the `printf()` call and try again.

We're obviously running in to a problem here ... but what might that be? Clearly it's something that happens in between those two points where you inserted calls to `halt`. But the only thing there is a straightforward `printf()` call, and we know that worked just fine previously ...

Think carefully about how `printf` works. And if/when you figure out what's going on, be sure that you don't say it out loud: you don't want to spoil the opportunity for the folks around you to figure this out on their own, no matter how strongly they might claim to disagree with that statement! :-)

Step 7: Mapping single pages

Although we've used 4MB super page mappings for the kernel space region of our page directories, we'll restrict ourselves to using 4KB mappings within the user space region. In particular, this means that we'll need to start using some page table structures in addition to the top level page directory.

Head on back to `kernel/paging.c`, and work on the `TODO`s in the `mapPage()` function. You might find it useful to look over the code in the `showPdir()` function at this point for an example of "walking" over page directory and page table structures. (Hint: `showPdir` is just a few lines below `mapPage`.)

You might also benefit from looking at the Intel diagram that shows the bit-level representations that are used for PDEs and

PTEs; remember that there's a snapshot of that at the start of this document that you can reference.

Step 8: The end, for this week

Once you've reached this step, you should have all the tools you need to fix that pesky problem from Step 6 — a single call to `mapPage()` should be enough! When you see a message indicating that `"The kernel will now halt!"`, you can confirm that it is accurate (e.g., using QEMU's `info cpus`), and then declare victory!

... Or you could start working to further configure the new page directory so that you can run a user level program within its own protected address space. (And why stop at just one running user program?) I'm not going to try to stop you ... but that is the topic for next week's lab, so I'm not going to say anything further about it right now, lest I spoil the fun :-)
