CS 343 Operating Systems, Fall 2020
Paging Lab: Implementing Virtual Address Spaces

1 Introduction

The purpose of this lab is to introduce you to virtual memory by implementing virtual address spaces using paging. Paging requires you to think at a deep level about indirection and its management via joint hardware/software mechanisms. In this lab, you will build an implementation of virtual memory using x64 paging within NK's address space abstraction.

You may work in a group of up to three people in this lab. Clarifications and revisions will be posted to the course discussion group.

2 Setup

You can work on this lab on any modern Linux system, but our class server, Moore, has been specifically set up for it. This lab will work best on our class server, Moore. Although you can set up your own machine with a special build of QEMU by following a guide on Piazza. We will describe the details of how to access the lab repo via Github Classroom on Piazza. You will use this information to clone the assignment repo using a command like this:

```
server> git clone [url]
```

This will give you the entire codebase and history of the Nautilus kernel framework ("NK"), just as in the Getting Started Lab. As before, you may want to use `chmod` to control access to your directory.

Important! You will need to make sure you have a valid display for NK to run. You can get that through FastX or with `ssh -Y`. See the Getting Started Lab for more details.

To start, you will need to configure the kernel:

```
server> cd [assignment-directory]
server> cp configs/cs343-paginglab-config .config
```

Now build it (you may need to run `bash` first if it is not your default terminal):

```
server> make clean
server> make -j 8 isoimage
```

You can now boot your kernel:

```
server> source ENV
server> ./run
```

The `run` command will execute the emulator (QEMU) with a set of options that are appropriate for the lab.

Boot failure! The emulated machine will boot NK. *You will see that the kernel is in a boot loop!* It will try to boot, get to a certain point, then the machine will spontaneously reboot. This is because the shell is trying to create and place itself into a new address space. Unfortunately, paging is mostly unimplemented, so what happens is a switch to a bad page table. Fetching the very next instruction immediately causes a *page fault*, which, to be handled, requires paging, and, with a bad page table, this faults again, this time with a *double fault*. To handle a double fault, the machine once again needs paging, so it faults again. This *triple fault* is handled directly by the hardware, by reseting itself. Hence the boot loop. Three strikes and you are out.

3 Files

While NK is tiny compared to the Linux, Darwin, or Windows kernels, it does have several hundred thousand lines of code spread over over more than a thousand files. Therefore it is important to focus on what is important for your goals. As with any significant codebase, trying to grok the whole thing is either impossible or will take far too long. Your strategy for approaching the code has to be adaptive. In part, we are throwing you into this codebase to help you learn how to do this.

Here are some important files for this lab. Your edits will be in `src/nautilus/shell.c` and `src/aspace/paging/paging.c` (bolded below):

- `include/nautilus/aspace.h`: This is NK's address space abstraction. You will be creating an address space implementation that conforms to it. You will not change this.
- `src/nautilus/aspace.c`: This is the implementation of NK's address space abstraction. You will not change this.
- **`src/nautilus/shell.c`**: This is the shell implementation, which uses the address space abstraction. It's where you can test things out. The boot loop is occurring as a result of the call to `nk_aspace_move_thread()`. The surrounding code shows how the address space abstraction is used. You will want to change this to try different tests.
- `src/asm/thread_lowlevel.S`: The call instruction to `nk_aspace_switch` in this code is what does a possible address space switch when a different thread is scheduled. You do not need to change this file.
- `include/nautilus/thread.h`: The field `aspace` within `struct nk_thread` points to the address space the thread is in. If this is null, it means the thread is in the default (or boot) address space. You do not need to change this file.
- `include/nautilus/smp.h`: The field `cur_aspace` within `struct cpu` points to the currently active address space for the CPU (the hardware thread). If this is null, it means the CPU is in the default (or boot) address space. You do not need to change this file.
- **`src/aspace/paging/paging.c`** This is the stub source code for your paging address space implementation. It is heavily commented. You will add to this.
- `src/aspace/paging/paging_helpers.[ch]` These files contain heavily commented helper code for building 4-level x64 page tables. You can leverage this helper code or write your own. You might find the `paging_helper_walk()` and `paging_helper_drill()` utility functions helpful. You are welcome to add to these files if desired.

Note that we are pointing out a lot of different files above. This is to show you how deeply embedded the notion of a virtual memory tends to be in a kernel, and to be complete. In this lab, you will mostly be working in `src/aspace/paging/paging.c`, which is heavily commented to help you.

4 Address spaces in NK

NK is somewhat different than the general purpose kernels, such as Linux, Windows, MacOS, BSD, etc, described in the book, as well as from the microkernels your author likes. In particular:

- The use of virtual memory is *optional* in NK. Using physical memory directly (or as close as possible) is the common case.
- There is no kernel/user distinction in NK by default.
- There is no process abstraction in NK by default.
- There is an address space abstraction designed to allow the use of models of virtual memory that do *not* involve paging as well as those that do. The address space abstraction is optional.

On x64 hardware, page tables *must* be installed. When NK boots, it builds a page table hierarchy that does an *identity map*, meaning that every virtual address maps to exactly the same physical address, with full kernel privileges. NK implements this page table hierarchy using the largest possible pages. Within the address space abstraction, this forms the *default address space*. Everything lives within this single address space unless it chooses otherwise.

The *address space abstraction* (`include/nautilus/aspace.h`) allows the creation and management of additional address spaces. A thread can choose to join an address space (`nk_aspace_move_thread()`). When a new thread is spawned, it joins the address space of its parent. Each CPU has a current address space, that of the currently running thread. Interrupt handlers and the rest of the kernel run in the context of the current address space of the CPU. You can find the current list of all address aspaces on the system using the `ases` shell command.

An address space is implemented by a *address space implementation*, and a design goal here is to allow very unusual implementations that manage address spaces at arbitrary granularities, and to allow address aspaces from multiple implementations to coexist at runtime. You are writing an address space implementation based on x64 4-level paging. You can get a list of all the available address space implementations with the `asis` shell command.

The address space abstraction centers around a *region* (`nk_aspace_region_t`), which is a mapping from a virtual address to a physical address for some number of bytes (not pages), combined with a set of protection flags. An address space contains a set of regions, and the set constitutes the address space's *memory map*. The memory map is an implementation-independent representation of the virtual address to physical address mapping.[1] In this lab, you will implement this mapping using paging.

Once an address space is created, it can be manipulated using regions:

- *Add region*: This expands the memory map. If the region is *eager*, then this part of the memory map must be immediately implemented by the underlying mechanism. For example, you would need to build matching page table entries in this lab. On the other hand, the page table entries for a lazy region could be built on demand.

[1] Note that this is different from the concept of memory map in Linux, which instead provides a mapping of virtual memory regions to file sections.

- *Remove region*: This shrinks the memory map. For paging you need to be sure that any page table entries you created for the region are also deleted. Page table entries may be cached in the TLB, so you also need to flush them from the TLB.
- *Move region*: The idea here is that we are changing the virtual to physical mapping of a region in the memory map. The virtual address stays the same, but the physical address changes. Similar to removing a region, you need to assure that old page table entries are edited, and that old entries are flushed from the TLB.
- *Protect region*: Here, we are changing the protections of an existing region. The virtual and physical addresses stay the same, but the protections change. Similar to moving or removing a region, you need to edit page table entries and flush the old ones from the TLB.[2]

Your address space also needs to handle the following requests:

- *Switch from*: This is invoked when your address space is about to stop being the current address space for the CPU. For paging, there is little you probably need to do.
- *Switch to*: This is invoked when your address aspace is about to become the current address space for the CPU. For paging, you need to install your page tables at this point.
- *Exception*: This is invoked in interrupt context whenever a page fault or general protection fault is encountered. For paging, you might build a page table entry for a lazy region at this point.[3]
- *Add thread*: This is invoked when a thread is joining the address space. For paging, you probably don't care.
- *Remove thread*: This is invoked when a thread is leaving the address space. For paging, you probably don't care.
- *Print*: Display the details of the address space. This supports the `ases` shell command.
- *Destroy*: The address space will no longer be used, and you should free all of its state.

5 x64 Paging

You will implement 4-level x64 paging. This is the most basic form of paging used on x86 (Intel/AMD/etc) processors when running in 64-bit mode ("long mode"). Please note that many teaching OS examples you might find (i.e. xv6, Pebbles, GeekOS, etc) use 32-bit mode, where paging is substantially different.

Three references for 64 bit paging that you should be aware of are the following:

- *CS 213 Textbook:* R. Bryant, D. O'Hallaron, Computer Systems: A Programmer's Perspective, 3rd edition, Section 9.7, shows the big picture of this kind of machine.

[2] Technically, whenever you flush entries from your CPU's TLB, you also need to make sure they are flushed from the other CPUs' TLBs. Typically, this is done using an inter-processor interrupt (IPI) called a TLB shootdown. This is outside of the scope of this lab, although you can try it for extra credit.

[3] If the reason for the page fault is unfixable by the kernel, then you should panic. In a kernel with a user space, if the page fault originated in user space, you would instead inject a signal (SIGSEGV (i.e., segfault) on Unix-like systems) into the user space program. If it originated in the kernel, you would be panic.

- *AMD Documentation:* AMD64 Architecture Programmer's Manual Volume 2: System Programming, Chapter 5, and in particular 5.3, describe paging on this form of architecture.

- *Intel Documentation:* Intel 64 and IA-32 Architectures Software Developer's Manual: Volume 3, Chapter 4, and, in particular 4.5 ("IA32e" is Intel's name for 64 bit mode)

The material, particularly the Intel documentation, can be quite daunting. In part this is because it is explaining all of the various modes and aspects of the machine tied to paging all at once. We are looking for you to build one thin slice through this. Also keep in mind that we have implemented a lot of support for you.

The following figure[4] shows how a virtual address is translated into a physical address by the hardware:

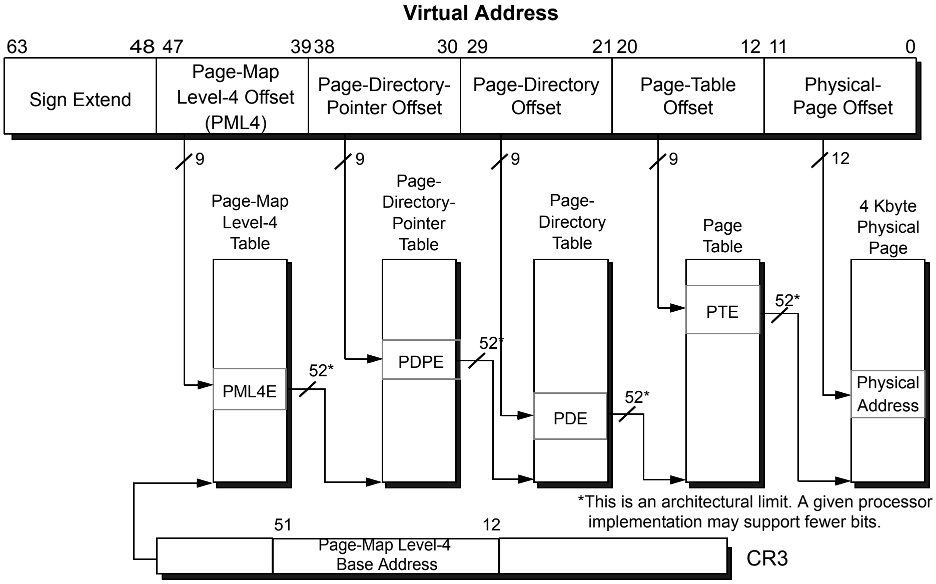

Figure 5-17. 4-Kbyte Page Translation—Long Mode

Here, the page table hierarchy we will use is selected by a pointer stored in the CR3 register—this pointer points to the root page table. The first group of 9 bits in the address are used to select one of the 512 entries on this table. The entry contains a pointer to the next level page table. The next group of 9 bits in the address are used to select an entry within it. And so on, all the way down to the last level page table, where the pointer indicates the physical page that corresponds to the virtual address.

The CR3 register has a special format:

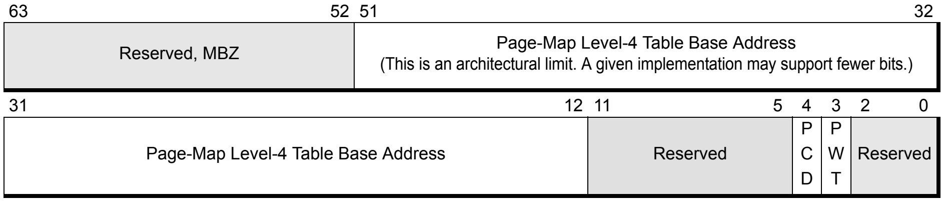
63
52
51
32
Reserved, MBZ
Page-Map Level-4 Table Base Address
(This is an architectural limit. A given implementation may support fewer bits.)
31
12
11
5
4
3
2
0
Page-Map Level-4 Table Base Address
Reserved
PCD
PWT
Reserved

Figure 5-16. Control Register 3 (CR3)—Long Mode

as do the page table entries at the different levels:

[4] Figures are from the AMD documentation unless otherwise noted.

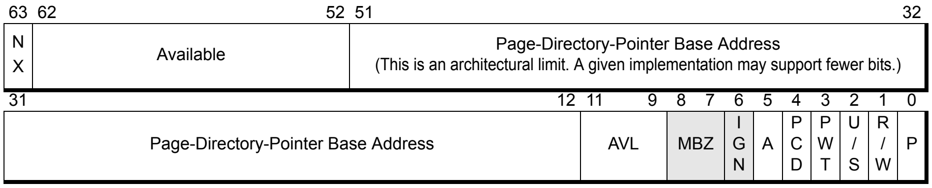
63 62 52 51 32
N X
Available
Page-Directory-Pointer Base Address
(This is an architectural limit. A given implementation may support fewer bits.)
31 12 11 9 8 7 6 5 4 3 2 1 0
Page-Directory-Pointer Base Address
AVL
MBZ
IGN
A
PCD
PWT
U/S
R/W
P

Figure 5-18. 4-Kbyte PML4E—Long Mode

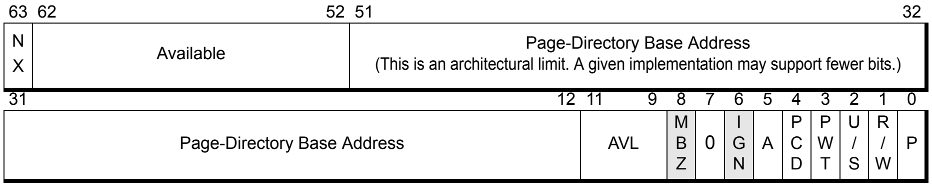
63 62 52 51 32
N X
Available
Page-Directory Base Address
(This is an architectural limit. A given implementation may support fewer bits.)
31 12 11 9 8 7 6 5 4 3 2 1 0
Page-Directory Base Address
AVL
MBZ
0
IGN
A
PCD
PWT
U/S
R/W
P

Figure 5-19. 4-Kbyte PDPE—Long Mode

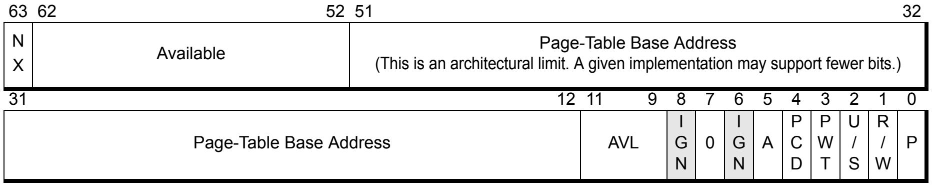

Figure 5-20. 4-Kbyte PDE—Long Mode

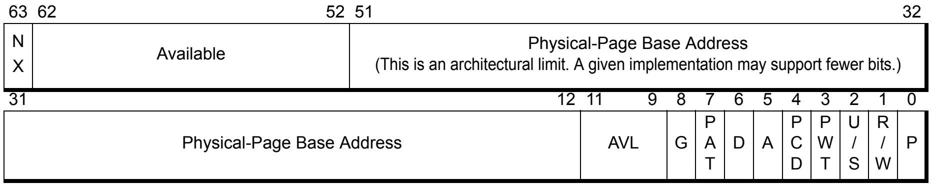
63 62 52 51 32
N X
Available
Physical-Page Base Address
(This is an architectural limit. A given implementation may support fewer bits.)
31 12 11 9 8 7 6 5 4 3 2 1 0
Physical-Page Base Address
AVL
G
PAT
D
A
PCD
PWT
U/S
R/W
P

Figure 5-21. 4-Kbyte PTE—Long Mode

CR4 also partially controls paging. The detailed layout of these registers and the page table entries is given in `src/aspace/paging/paging_helpers.h`, along with detailed comments. Typically, in our own code we want to "drill" the page table hierarchy, which means to create a path of page table entries that fully describe a particular virtual address. Also, we often need to "walk" the page table hierarchy ourselves, similar to how the hardware does it, for example to find out why a particular page fault occurred. We supply you with helper functions for these actions in `src/aspace/paging/paging_helpers.c` You can see how both the definitions and the functions are used in the initial code we provide in `src/aspace/paging/paging.c`.

You will also need to make use of the following functions/data.

- `read_cr3()` and `write_cr3()`: These functions read and write the CR3 register, the pointer to the root page able. A write to CR3 also has the side effect of flushing the TLB of all entries.
- `read_cr4()` and `write_cr4()`: These functions read and write the CR4 register, which has several bits that must be set correctly for paging to operate as we want. (The code we provide already does much of this).
- `read_cr2()`: This function reads the CR2 register, which contains the faulting address when a page fault occurs.

- excp_entry_t, field error_code: On a page fault, this value is the reason for the page fault (why it occurred). See the stub code in for more.
- invlpg() Given a virtual address, this function flushes the corresponding entry out of the TLB, if it's in the TLB.

6 Task 0: Understand the boot loop

Start by walking through the code, starting with the aspace-related calls that occur in src/nautilus/shell.c. Your goal is to understand for yourselves why the boot loop is occurring. You might want to try commenting out the nk_aspace_move_thread() call to see what happens. Then bring it back in, but add incremental printouts so that you can see how far it gets. Possibly also attach gdb and use it to single-step through this processing.

Note that the abstraction creates an indirection that can make it hard to follow. In this case, we can assure you that a call to a function like nk_aspace_add_region() will result in call to the add_region() function within the paging implementation. The other high-level calls will also route to the similarly named functions in the paging implementation.

7 Task 1: Eager page table construction

Your next step is to stop the boot loop by building a page table hierarchy that has the necessary parts of the virtual address space mapped. Note that the test code in the shell adds two regions. The first of these is an "eager" region. This means you should build page tables for it immediately, right in your add_region() function. You may find the paging_helper_drill() function useful for this.

Once you correctly construct the page table entries corresponding to this eager region, your boot loop should go away. However, the second region we ask you to add in the shell code is not an eager region. You should not build it eagerly. As a consequence, you will now likely repeated page fault. This is due to the memcmp() test in the shell code. If you comment this out, you should boot all the way to the shell prompt. At this point, you should be able to run the ases command or threads command and see that your shell is running in a new address space.

8 Task 2: Memory map data structure

To continue the lab, you will need a way to manage the regions that comprise your memory map. Recall that the memory map is a set of regions. You need to design and implement a data structure that contains an set of regions. You need to be able to add and remove regions from the data. Regions may not overlap by virtual addresses. That is, a virtual address must map to at most one region. On the other hand, a physical address can map to multiple regions. The data structure also needs to be searchable by virtual address. That is, given any virtual address (not just the start of a region), you need to find the region that contains it, if any exists.

This does not have to be fancy since we will not grade you on performance. A simple linked list can work fine. We provide a Linux-style intrusive linked list implementation (include/nautilus/list.h) if you'd like to leverage that.

9 Task 3: Lazy page table construction

Now that you can keep track of your memory map, you can start handling page faults, and possibly construct new page table entries based on them.

Enable the second region in the shell test code (the lazy one), and enable the `memcmp()` test. The region should now be included in your memory map, but, at least initially, you won't have any page table entries to support it. As a consequence, the `memcmp()` will cause a page fault when it reads from the high address.

Your page fault handler can now do something about this. It should get the faulting virtual address and error, and then look up the virtual address in the memory map. If a region exists, and the permissions of the region are appropriate given the error, then drill a page table entry for the address, with the physical address corresponding to the region.

Once you have this right, you will survive the boot process all the way to the shell prompt. You will also see numerous page faults during the boot process, all of which are satisfied by your lazy page table construction logic.

10 Task 4: Fleshing out your implementation

Finally, you will expand your implementation to support deleting and moving regions, as well as changing their protections. At some level, this logic is straightforward once you can handle lazy page table construction because it depends similarly on the memory map data structure. However, read carefully the earlier comments about TLB invalidation. When a page table entry can be cached in a TLB, it is important to make sure it is removed from the TLB after you edit the in-memory copy.

Before testing write permissions on an address space, you will need to enable write protections in the kernel. By default, write permissions are normally ignored when in kernel mode. To enable them add the following line to your `src/nautilus/shell.c` once, sometime before calling `nk_aspace_protect_region()`.

```
write_cr0(read_cr0() | (1<<16));
```

11 Task 5: Reflection on your implementation

Answer the following questions about your implementation and how it functions. Put the answers in your `STATUS` file. Don't feel like you need to spend pages answers these. Simple, concise answers are greatly preferred. For many of these questions there is no correct answer, only an answer that is how it would apply to your implementation.

1. Explain what your data structure for the memory map / region set is. What are the costs to insert/remove/change/search for regions?

2. Explain how you handle the following situations/questions in Task 4:

   (a) `add_region` that has a virtual memory overlap with an existing region in the memory map

   (b) `add_region` that has a physical memory overlap with an existing region in the memory map

   (c) `move_region` on the current thread's address space where the move would end up causing a virtual memory overlap. This is for a move involving multiple pages.

(d) protect_region on a non-existent region. How do you find out it is non-existent?

(e) when is it necessary to flush the whole TLB (move to cr3) ?

(f) when is it necessary to flush a single page from the TLB (invlpg) ?

(g) what happens if a valid delete_region request is for a region that contains %rip?

12 Grading

Your group should regularly push commits to Github. You also should create a file named STATUS in which you regularly document (and push) what is going on, todos, what is working, etc. Your commits are visible to us, but not to anyone else outside of your group. The commits that we see up to deadline will constitute your hand-in of the code. The STATUS file should, at that point, clearly document that state of your lab (what works, what doesn't, etc).

In addition to your STATUS file, you should regularly push your work within src/aspace/paging/*, src/nautilus/shell.c, and all other files you are changing.

The breakdown in score will be as follows:

- 20% Task 1—Functional and sensible implementation of eager page table construction. With memcmp() disabled, it should boot to the kernel prompt. Furthermore, creating additional shells should create additional address spaces that work correctly.
- 20% Task 2—Sensible implementation of a memory map data structure. You will have to explain the structure during your interview. It needs to provide add, remove, change, and lookup functionality. The lookup needs to work for arbitrary addresses.
- 35% Task 3—Functional and sensible implementation of lazy page table construction. With memcmp() enabled, it should still boot to the kernel prompt. Creating additional shells should work as before.
- 15% Task 4—Fleshed out implementation. There should be support for removing, moving, and changing the protection of regions. Matching testcases, perhaps implemented in the shell code, are expected. Be prepared to explain your TLB flushing logic.
- 10% Task 5—Reflection on your implementation. Put the answers in your STATUS file please.

13 Extra Credit

We will allow up to 20% extra credit in this lab. If you would like to do extra credit, please complete the main part of the lab first, then reach out to the instructor and TAs with a plan. Some possible extra credit concepts are the following:

- Add TLB shootdowns to provide correct behavior on multicore systems when the same page table is shared among multiple cores.
- Add large, huge, and giant page support. Make use of the multilevel paging structure so that regions that be mapped more efficiently. For example, if a region starts at an address that is a multiple of 2MB boundary, and its length is also a multiple of 2MB, then 2MB pages (3-level hierarchy) can be used instead of 4KB pages (4-level hierarchy).
- Implement a user space. We already have a mechanism for loading position independent executables.