CS 343 Operating Systems, Fall 2024
Paging Lab: Implementing Virtual Address Spaces and Processes

Contents

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. Furthermore, your paging implementation will add the capability to run userspace processes to NK.

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, and this is also where we will evaluate it. 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 of the Nautilus kernel framework ("NK"), just as in the Getting Started and Driver Labs. 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.

You can now boot your kernel: (this will also build the kernel if needed, so don't worry about running `make` every time)

```
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 Important 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. You should spend most of your time in the files which are bolded below, and most of your edits will be in `shell.c` and `paging.c`.

- `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()` called in the `shell()` function. The surrounding code shows how the address space abstraction is used. This is your first test!

- `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.

- `include/nautilus/paging.h`: This includes helpful definitions that can be used in paging implementations. For example, `PAGE_SIZE_4KB` can be used in your code instead of remembering how many bytes a standard 4 kB page is. 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.

- **`src/aspace/paging/paging_test.c`** This file contains additional test code, which you can run using the shell command `pagingtest`. You are welcome to add additional tests. Also included are several commented-out tests which *should* fail if your implementation is correct. When you believe your implementation is complete, you should bring them back, one at a time, and check that they fail as expected.

- **`src/user/process.c`** This file contains an extremely minimal process abstraction built on top of the aspace abstraction you will work on in this lab. We recommend looking through it a little bit in the latter parts of the lab to help you understand how *file-backed* and *anonymous* regions should work in your aspace implementation. The primary thing this file does is manage the creation and execution of processes, and the handling of system calls that the process makes (in the `process_dispatch_syscall` function). This will be covered in more detail later in the lab (Task 5). Don't feel like you need to understand the inner-workings of the user/kernel separation, as the interaction with the hardware which enables it is fairly complex and outside the scope of this lab.

- `user/bin/*.c` These files are all individual userspace programs that can be run from the userspace shell. If you want to make a new program, simply add a new c file to that folder and the build system will do the rest for you.

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 (Though we have added one specially for this lab).
- 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.

Even if you're not using them, 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 spaces 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. When implementing this functionality, consider how it can be extended to additional region types. Primarially, consider how your implementation might work if physical memory came from elsewhere (see task 5 later)

- *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.

[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. However, the previously mentioned anonymous and file-backed regions we have added for this lab are similar to Linux's model.

[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.

[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. In Task 5, you will do just that.

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.
- *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:

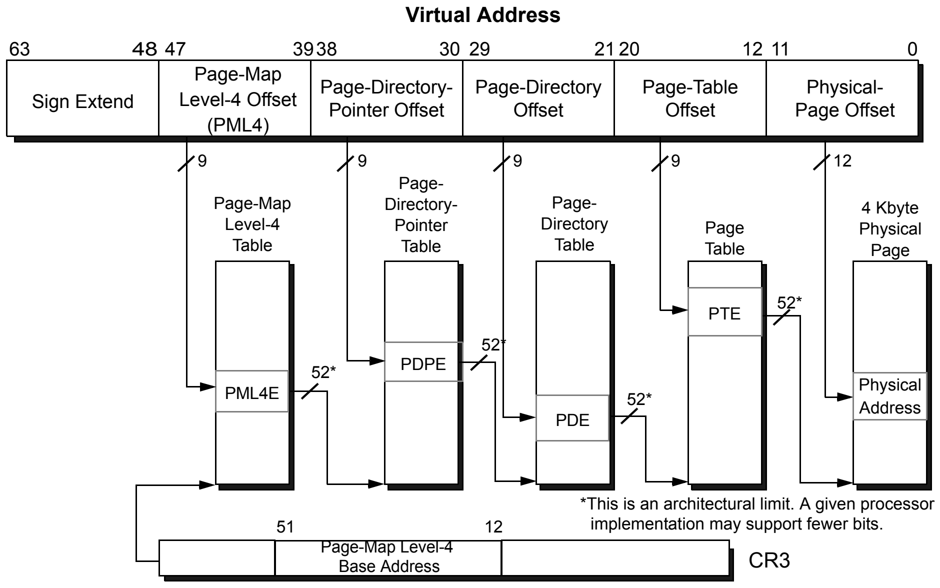


Figure 5-17. 4-Kbyte Page Translation—Long Mode

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.

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

The CR3 register has a special format:

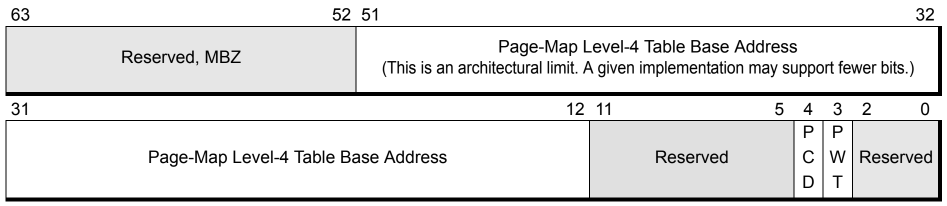

63
52 51
32
Reserved, MBZ
Page-Map Level-4 Table Base Address
(This is an architectural limit. A given implementation may support fewer bits.)
31
12 11
5 4 3 2 0
Page-Map Level-4 Table Base Address
Reserved
PCD
PWT
Reserved

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

as do the page table entries at the different levels:

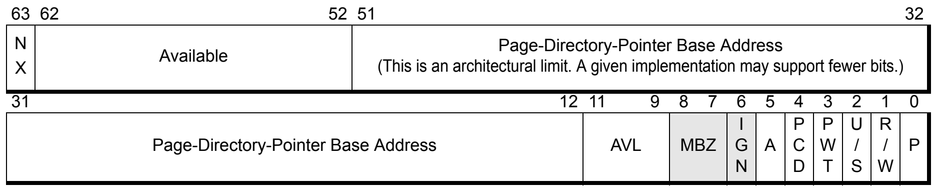

63 62
52 51
32
NX
Available
Page-Directory-Pointer Base Address
(This is an architectural limit. A given implementation may support fewer bits.)
31
12 11 9 8 7 6 5 4 3 2 1 0
Page-Directory-Pointer Base Address
AVL
MBZ
IGN
A
PCD
PWT
U/S
R/W
P

Figure 5-18. 4-Kbyte PML4E—Long Mode

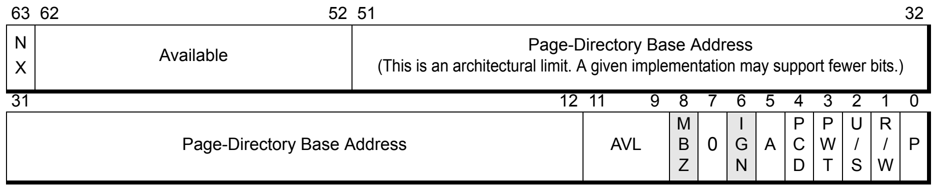

63 62
52 51
32
NX
Available
Page-Directory Base Address
(This is an architectural limit. A given implementation may support fewer bits.)
31
12 11 9 8 7 6 5 4 3 2 1 0
Page-Directory Base Address
AVL
MBZ
0
IGN
A
PCD
PWT
U/S
R/W
P

Figure 5-19. 4-Kbyte PDPE—Long Mode

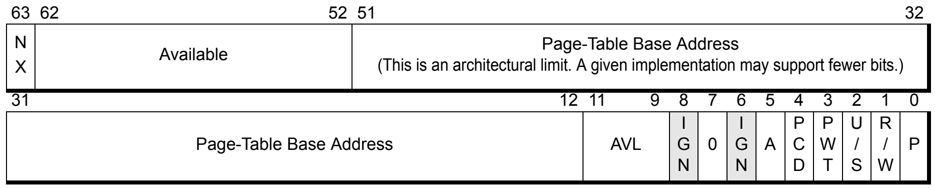


Figure 5-20. 4-Kbyte PDE—Long Mode

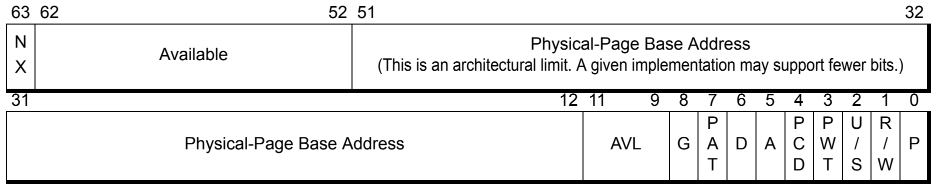

63 62
52 51
32
NX
Available
Physical-Page Base Address
(This is an architectural limit. A given implementation may support fewer bits.)
31
12 11 9 8 7 6 5 4 3 2 1 0
Physical-Page Base Address
AVL
G
PAT
D
A
PCD
PWT
U/S
R/W
P

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 sometimes 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 that file.

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. However, only write to CR3 if the address space you're modifying is also the currently running process. Otherwise, you'll install the wrong page table for the current process!!
- `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 `paging.c` for more.
- `invlpg()` Given a virtual address, this function flushes the corresponding entry out of the TLB, if it's in the TLB. You should probably be using this function to clear entries from the TLB in most cases.

For the curious, these functions consist of assembly code that directly performs these actions using special instructions that can only be run in kernel mode.

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 will find the `paging_helper_drill()` function useful for this.

Drilling a page builds the page table hierarchy that represents the translation for that page. Multiple pages will end up sharing parts of their hierarchy if they are adjacent. Drilling requires a virtual and physical address for the page as well as a `ph_pf_access_t` union that specifies permissions for the page. The union is defined in `src/aspace/paging/paging_helpers.c` and is identical to `ph_pf_error_t`. In an `access_type` the bits are used to represent the permissions granted to the page. Be careful here, as the bits used to represent permissions in the `protect` field of a region are not identical to the bits used in `access_type`.

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 get a 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. Most students create their own data structure, which is totally fine. You could even add new `.c` and `.h` files inside the paging directory and add them to the `Makefile`. For now, keep the implementation simple, and you can add features as needed when working on later tasks.

Initialize your data structure in the `create()` function and clean up any memory allocated for it in the `destroy()` function.

Note that your data structure should hold region structs that are copies of the passed in region data, rather than storing the region pointer itself. The region pointer passed into the `add_region()` function may not persist past the return of the function or may be modified in the future.

To help yourself out, you might consider writing a debug function which iterates through the regions in your data structure and prints out some information about each such as starting virtual address, length, and permissions. We won't require this functionality, but we do strongly recommend it!

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 implemented in `exception()` 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 single page table entry for the faulting address, with the physical address corresponding to the region. Do not drill the entire region, only the faulting page.

To determine if the permissions of the region are valid, you should look at the `ph_pf_error_t` created from the value passed into `exception()`. The union is defined in `src/aspace/paging/paging_helpers.c` and each field provides characteristics of the access that faulted. You can compare those with the permissions of the region to determine if they may be problematic.

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.

It's pretty easy for things to go wrong here. Perhaps the implementation drilled the wrong virtual page, or maybe it gave it the wrong permissions, or it may have mapped to an incorrect physical address. To help you debug, `src/aspace/paging/paging_helpers.c` defines the `paging_helper_print_translation()` function. Given the base address for a page table and a virtual address, it will perform a lookup of the page table entry for that virtual address and print a debug message with the results. Feel free to modify this function to add additional information that you think would be useful.

10 Task 4: Fleshing out your implementation

Next, you will expand your implementation to support deleting and moving regions, as well as changing their protections. Note that applying these changes may be different depending on whether the region has actually affected the page tables due to being an eager region or at least one page fault having already occurred.

Regions that are removed, but have already been drilled in the page table hierarchy, should be marked as not present. A full implementation would scan the page table hierarchy and free branches that are entirely not present, but you do not need to implement that logic for this lab.

Regions are only moved from one physical address to another. This could represent memory being placed in a new location in RAM after a swap to disk and back. Region virtual addresses will never change.

Regions can be created with or modified to have arbitrary permissions. Any combination of writeable and/or executable is possible and should be enforced, but regions will always be readable. Regions can also be "pinned". Pinned regions cannot be moved or removed unless they are first unpinned. This prevents important memory regions from being swapped to disk.

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 testing code once, sometime before calling `nk_aspace_protect_region()`.

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

We already do this for you as an example in the `pagingtest` code (`src/aspace/paging/paging_test.c`).

Generally, the rest of the paging 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.

11 Task 5: Processes

Now that the kernel successfully boots and the paging tests pass, it's time to get your paging implementation working under an environment which more closely approximates a real-world usage of paging. The entire process abstraction has been built for you, and you will be adding two features to your aspace implementation to get them working. Surprisingly, adapting your paging implementation to allow a process to use it shouldn't take much work! For example, our solution only took 10 extra lines of code in `add_region`, 10 lines in `exception`, and 3 lines in `remove_region`.

NK's Process Abstraction

For this lab, we have added an extremely minimal userspace environment to the Nautilus kernel located in `src/user/process.c`. This environment is designed around the aspace abstraction you have just created.[5]

An important thing to note is that this userspace environment is *not* "POSIX compliant." That is, it is not designed to look or act like the Linux kernel in any way. NK processes, at their core, consist of a single thread (the main thread), an address space, and a set of open files. You can see the definition of the `nk_process_t` structure in `include/nautilus/user.h`.

Processes can be created from the shell by running the `urun` command. If you run that command now, before doing this task, it will immediately crash (try it!). Most likely, it will crash due to an invalid opcode exception and the instruction pointer will be somewhere around `0xffff8000000...`, but the crash could occur for many different reasons. This is because the program that you are executing isn't being loaded off the disk, and the userspace stack is not being mapped correctly. **To enable this, you'll have to expand your paging implementation to enable a couple of additional kinds of memory mappings (in addition to eager and lazy regions).**

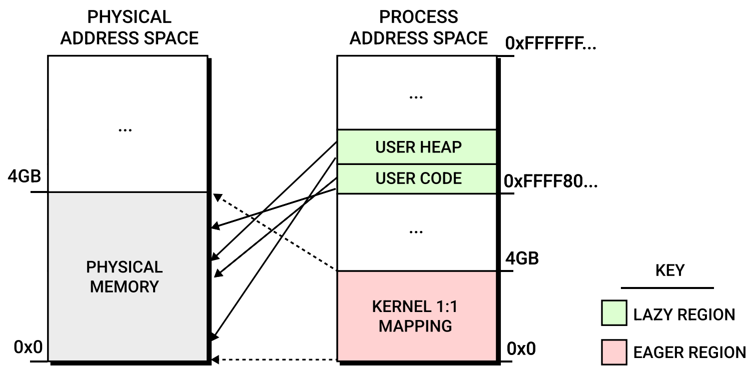
Interlude: The NK User Address Space

Before we get to the tasks, it might be helpful to quickly go over what the address space of a process looks like in NK. Below is a diagram which shows that the "user code" and "user heap" both live above the `0xffff8000000...` address mark (hence the crash from before). This is what is known as a "high and low half split", where the address space is split exactly down the middle, and both the kernel and userspace gets a half. In NK, the userspace lives in the "top half" and the kernel lives in the "bottom half". Keen readers might notice that that is the opposite of what other operating systems might do.[6] The reasoning behind this inversion in NK is due to the need to maintain the kernel's understanding about it's address space. The processes in NK were tacked on late into it's development lifestyle, and as such, most of the kernel relies on the fact that it's virtual memory is mapped 1:1 with physical memory. To that end, the kernel is mapped low (1:1) in red (kernel only) and the userspace is mapped high in green (the U bit is set on the PTEs[7]).

[5]This section of the lab is still relatively new, so there may be issues we are currently unaware of. Please ask us on Piazza or in office hours if you see something happen that doesn't make sense to you.

[6]Linux placed the userspace in the bottom half to allow it to access lower addresses (lower addresses are easier to encode in instructions, and it enables legacy software to run on modern systems). Although that was before Meltdown and Spectre led us to remove the kernel mapping altogether.

[7]This protection should be managed for you.



You may notice that there isn't a stack in the user address space. The reason is because the user stack is actually allocated on the same regions as the heap to simplify the implementation a bit. On a more complex system, the stack would be allocated in a different area (typically at the high addresses of the userspace).

The lower half will be mapped eagerly, and the regions in the upper half will be mapped as lazy regions. Your aspace implementation should already support the lower, eager region. To support the high half user regions, you will need to make a few changes to support two new region types (don't worry, it shouldn't be too many additional lines of code).

Task 5.1: Adding the New Region Types

One of the primary benefits of paging and virtual memory as a whole is that it doesn't really matter where the physical memory comes from. On modern operating systems memory can come from many different places, and it is often moved around at runtime. The first kind of memory mapping you will have to add to your paging implementation is a *file-backed mapping*. On Linux and most operating systems, a user can map an open file into memory, and the kernel will make sure that when the user reads from that memory, the file's contents will be there. If the file is mapped starting at virtual page p in the address space, then the contents of page k of the file will appear in virtual page $p + k$ in the process. Thus, file mappings are a special kind of mapping, and the aspace implementation needs to fully manage the physical memory assigned to a region in the aspace. If a region is file-backed, it is marked with the `NK_ASPACE_FILE` flag, and `pa_start` is meaningless. Instead, you must call `nk_fs_read`, passing `region->file` as an argument, to read each page of the mapping into the memory mapping. The signature of the read function is as follows: (see `include/nautilus/fs.h` for more info)

```
ssize_t nk_fs_read(nk_fs_fd_t fd, void *buf, size_t len);
```

Note, this API will read from the file continuing at the last byte-offset you read from. [8] To ensure you read from the right page, it is good practice to *seek* to the correct location first. If you need to read a particular page, *p*, you must first call `nk_fs_seek` to seek to the right page in the following way:

```
nk_fs_seek(fd, k * 4096, 0); // Seek to byte offset 'k * 4096'
nk_fs_read(fd, pa, 4096);    // read 4096 bytes into pa
```

Your aspace implementation will need to manage allocating and freeing each *page of physical memory* that is mapped into the address space with `malloc(4096)` and `free`. In this lab, a file-backed mapping

[8] This is a common operating mode in most file abstractions. See the POSIX "FILE*" abstraction for more information.

is usually lazy, and so data is loaded from the file only as needed – as reading from the disk is very expensive, and you want to avoid it at all costs. The term for this is *demand paging*. Thus, you will have to handle file-backed mappings in `add_region`, `exception`, and `remove_region`. Be cautious in your `exception` implementation, as the faulting address does not necessarily align with the start of the virtual page containing it.

The other kind of mapping you will need to manage is the *anonymous mapping*. Up until now, your paging implementation has mapped contiguous regions of physical memory using the `pa_start` in the region structure. One of the benefits of paging is that it doesn't really matter where the physical memory comes from. As such, the process abstraction uses *anonymous regions* to tell the aspace implementation that the physical memory can come from anywhere. When a region is anonymous, `pa_start` is meaningless, and you will need to allocate each physical page you map using `malloc` much like the file-based mapping. **Effectively, anonymous mappings are just file-based mappings, but instead of loading from a file, they are initialized to zeroes.** Anonymous regions are marked with the `NK_ASPACE_ANON` flag. Much like file-backed mappings, anonymous mappings need support in `add_region`, `exception`, and `remove_region`.

Task 5.2: Signal Delivery

Currently, your aspace handles invalid pagefaults in `exception()` by returning -1. This leads the kernel to panic and stop running. However on a real system, if a userspace process causes an invalid pagefault, you don't want the kernel to panic. Thus, the kernel sends a *signal* to the process that it should either handle the fault, or exit. The NK process abstraction in this lab allows you to do this, and you should deliver a signal by returning `return set_pending_signal();` instead of -1. If you do this correctly, the `hack` program should print that it got a signal in the next section. NOTE: Signals are delivered by just changing the instruction pointer, as such signals in NK processes are not able to return to the faulting instruction.

Running programs and testing your process support

Congratulations! Now that you have both of these mappings working, the `urun` command should work, and you should see a userspace shell that looks like the following:

```
root-shell> urun
Hello from userspace!
[init] starting shell (/sh). Run the 'help' program for help
user#
```

You can type `exit` at the userspace shell to exit the process abstraction and return to the NK shell.[9] From the NK shell, you can run `ls` to see a list of programs you can run. An important program to try is the `hack` program, found in `user/bin/hack.c`. This program will attempt to hack into the kernel in a few ways. First, it will map some virtual memory and check if it is all zeroes. If you forgot to zero out your anonymous memory allocations, this program will catch it. Second, it will attempt to zero out kernel memory, and your page fault function, `exception()` should protect it from doing so and send a signal.

[9]You might be wondering what the difference is between the NK shell and the userspace shell. The NK shell is part of the kernel and runs in kernel mode with full access to everything. The userspace shell runs in a user process, in user mode, with limited access. Additionally, the kernel is protected from the userspace shell, and the other userspace processes are protected from each other.

If you want to add your own userspace programs to the system (which we recommend as a learning exercise) you can add a C file to `user/bin/`. This C file will automatically be compiled for you, and an associated program will be runnable from the user shell. After you are done with the process abstraction, make sure your aspace still passes the 'pagingtest' test from earlier.

12 Task 6: 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 answering 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 your implementation handles the following situations/questions:
   (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 physical 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`? What will happen after the paging library code completes?
3. Given your understanding of file based mappings, go read the man page for mmap. (`man mmap` on moore). What would you need to change in your implementation to support the semantics of the `MAP_SHARED` flag when `fd != -1`, and the user wrote to the memory region?

13 Grading

Your group should regularly push commits to Github. You also should create a file named `STATUS` (no file extension, please) 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.

You will also need to submit your files to Gradescope. To do so, in the root of the repository run:

```
$ make submit
```

That will upload your `STATUS`, `src/aspace/paging/*`, and `src/nautilus/shell.c`files to Gradescope. After you submit, you'll need to mark your group members on your submission. Unfortunately, you will need to do this each time you submit.

The breakdown in score will be as follows:

- 25% 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.
- 10% Task 2—Sensible implementation of a memory map data structure. It needs to provide add, remove, change, and lookup functionality. The lookup needs to work for arbitrary addresses.
- 20% 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.
- 10% Task 4—Fleshed out implementation. There should be support for removing, moving, and changing the protection of regions. You must pass the `pagingtest` test.
- 25% Task 5—Processes. Get virtual memory working for the process abstraction. You should be able to run the urun command at the NK shell, and the hack program at the user shell.
- 10% Task 6–Reflection on your implementation. Put the answers in your `STATUS` file please.