Overview

Let’s begin by exploring how to virtualize memory with pages.

This section should help us answer the following questions:

- **How can we virtualize memory with pages without segmentation issues?**
- **What are the basic concepts?**
- **How can we make those strategies work well while saving space and time?**

Introduction

When tackling almost any space-management challenge, the operating system is said to use one of two approaches.

1. The first technique is to cut things up into *variable-sized sections*, as we saw with virtual memory **segmentation**. However, this solution has several challenges. When dividing a place into different-sized parts, the area itself might become **fragmented**, making allocation increasingly difficult over time.

As a result, it may be worthwhile to investigate the second approach:

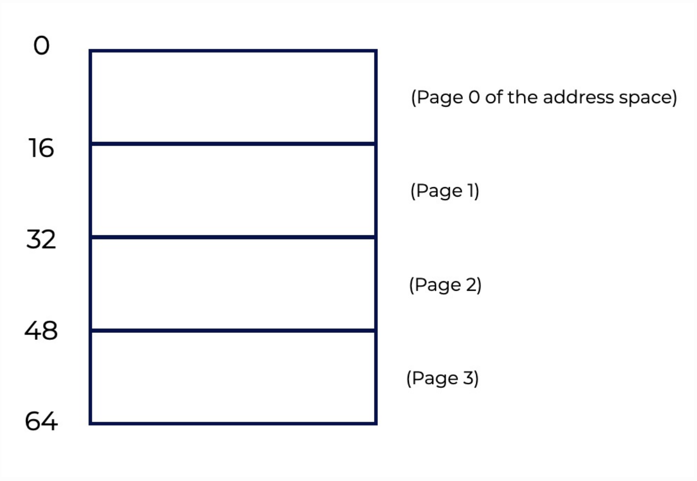
2. Dividing space into *fixed-sized chunks*. This concept is known as **paging** in virtual memory. Instead of dividing a process's address space into variable-sized logical segments (e.g., code, heap, stack), we divide it into fixed-sized units called **pages**. Similarly, we see physical memory as an array of fixed-sized slots called **page frames**, each of which can hold a single virtual-memory page.

Which of the following describes generally how paging works?

Fill in the blanks to complete the statement below.

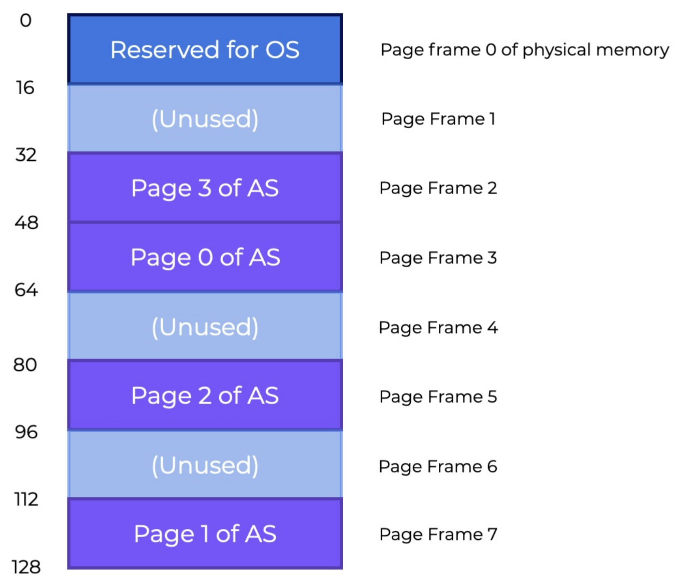
A Simple Example And Overview

Let’s use a basic example to better understand this method. The graphic below shows a 64-byte address space with four 16-byte pages (virtual pages 0, 1, 2, and 3).



Typical address spaces are much larger, however we utilize modest examples to explore these concepts.

Physical memory contains a number of fixed-sized slots. In the graphic below, there are eight page frames that give us 128-bytes of physical memory. The pages of the virtual address space in our graphic have been put at different locations across physical memory. Our graphic also shows that the OS is using some of the physical memory for itself.



.guides/img/paging128

Paging has significant advantages over our earlier methods.
* Paging is more **flexible** than previous methods.
* The system will be able to support the abstraction of an address space effectively, regardless of how a process uses the address space. We won't make assumptions about the way the heap and stack grow and how they are used.

- Paging also **simplifies** free-space management.
  - For example, to fit our little 64-byte address space into our eight-page physical memory, the OS simply seeks four free pages. Maybe the OS keeps a **free list** of all free pages and grabs the first four free pages from this list. In our example, the OS puts:
    - Virtual **page** 0 of the address space (AS) in physical **frame** 3
    - **Page** 2 in **frame** 5, and
    - **Page** 3 in **frame** 2
    - **Page Frames** 1, 4, and 6 are **free**

A **page table** is a per-process data structure that records where each virtual page of the address space is located in physical memory. The page table **stores address translations for each virtual page of the address space, indicating where in physical memory each page is located**. In our example, the **page table** would have *four entries*:

- (VP 0 → PF 3)
- (VP 1 → PF 7)
- (VP 2 → PF 5)
- (VP 3 → PF 2)

This page table is a **per-process** data structure (most page table structures we discuss are per-process structures. The **inverted page table** is an exception). In our example, if another process ran, the OS would have to manage a different page table for it because its virtual pages corresponded to different physical pages (aside from any sharing going on).

Which of the following describes a page table?

Address Translation Example

With this knowledge, we can do an address-translation example. Assume the process with the small address space (64 bytes) is accessing memory:

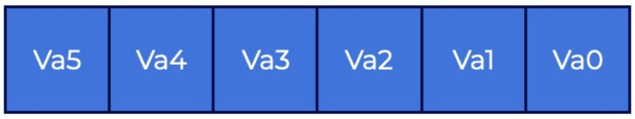
```
movl <virtual address>, %eax
```

Notice the explicit load of data from address `<virtual address>` into register `eax` (and ignore the instruction fetch that must have happened prior).

To **translate** the virtual address the process generates:

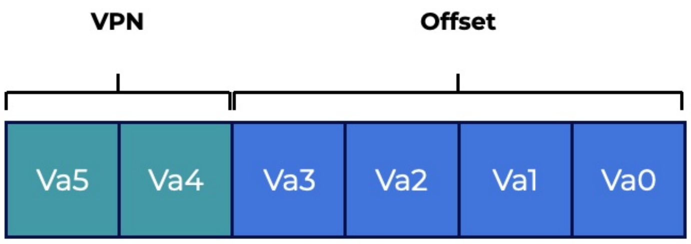
1. We have to break the resulting virtual address into two parts:
   - The **virtual page number (VPN)** and
   - The **offset** within the page.

Because the process's virtual address space is 64 bytes, we need 6 bits total for our virtual address ($2^6 = 64$). So, we can think about our virtual address as follows:



.guides/img/paging2

In this diagram, $Va5$ is the highest-order bit while $Va0$ is the lowest. We can further divide the virtual address as follows, knowing that the page size is 16 bytes:



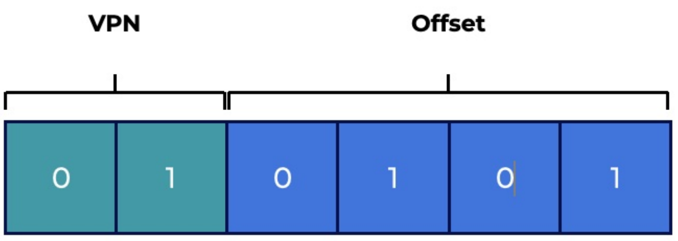
.guides/img/paging3

The page size is 16 bytes in a 64-byte address space, so we need to be able to choose 4 pages, which the top 2 bits do. Now we have a 2-bit **virtual page number** (VPN). The remaining bits tell us which byte of the page we want to look at, in this case 4 bits. This is called the **offset**.

When a process generates a virtual address, the OS and hardware have to work together to translate it into a meaningful physical address. Let's say the load we issued earlier, `movl <virtual address>, %eax`, was to virtual address 21:

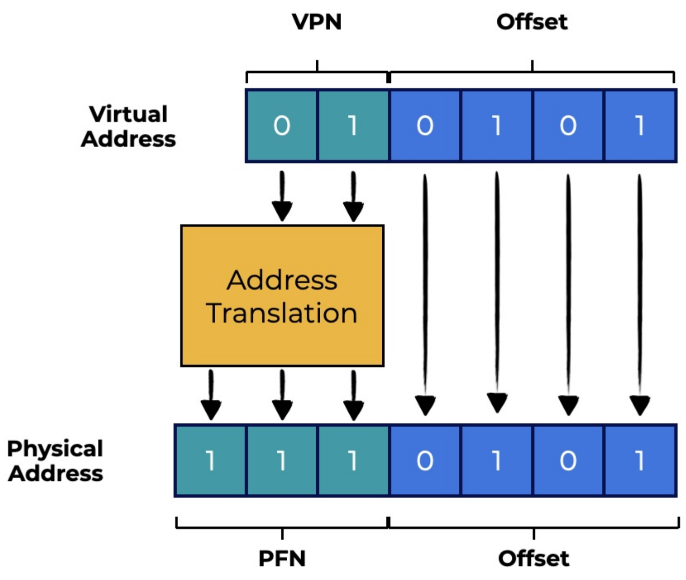
```
movl 21, %eax
```

If we convert "21" into its binary form, we'll get "010101". With this, we can explore this virtual address and see how it breaks down into a **virtual page number** and **offset**



.guides/img/paging4

So, virtual address "21" is on the $5th$ ("0101"th) byte of virtual page "01" (or 1). Using our **virtual page number**, we can now index our **page table** and find out which physical frame virtual page 1 lives in. The **physical frame number (PFN)** is 7 in the page table above (binary 111). So, we can translate this virtual address by replacing the **VPN** with the **PFN**, then issue the load to physical memory, as in our graphic below.



.guides/img/paging6

Because the offset just tells us the byte inside the page we want, it remains constant (it is not translated). Our final physical address is 1110101 (117 in decimal), and this is the location from which we want our load to retrieve data.

With this basic understanding in mind, we can now ask (and hopefully answer) a few fundamental paging questions.

- Where, for example, are these page tables stored?
- What are the typical page table contents, and how large are the tables?
- What are the typical page table contents, and how large are the tables?
- Is paging causing the system to (become) too slow?

These and other enticing questions are addressed, at least in part, in the following material.

Page Table Storage

Page tables can grow to be much larger than the segment tables or base/bound pairs we previously discussed.

Say we have a standard 32-bit address space with $4KB$ pages. This virtual address is composed of:
* A 20-bit VPN, and
* A 12-bit offset
* Recall that 10 bits would be needed for a $1KB$ page size, and just add two more to get to $4KB$).

A 20-bit VPN implies that the OS has to maintain 2^{20} translations for each process (about a million). Assuming 4 bytes per **page table entry (PTE)** to carry the physical translation plus any other helpful information, we get an enormous $4MB$ of RAM required for each page table!

That's a lot of space.

Now, think about 100 processes running. The OS would require $400MB$ of memory just for address translations! Even though computers have gigabytes of memory, its still pretty wild to consume a huge portion of it just for translations. We won't even get into how large such a page table would be for a 64-bit address space. That's just plain scary!

We don't have any specific on-chip hardware in the MMU to hold the page table of the presently-running process because page tables are so large. Instead, **we keep the page table for each process in *memory* somewhere**.

For now, we'll **assume that page tables are stored in physical memory that the OS manages**. Later, we'll see that much of the OS memory itself can be virtualized, so page tables can be stored in OS virtual memory (and even swapped to disk).

The graphic to the left shows us a page table in OS memory. See the little set of translations there?

What’s in the Page Table?

Let’s look at table page organization.

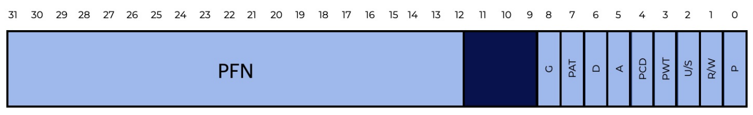
The page table is a **data structure** that maps virtual addresses (or virtual page numbers) into physical addresses (physical frame numbers). So, any data structure could work for this.

A **linear page table** is just an array.
* A **virtual page number (VPN)** is used to index the array, and
* A **page-table entry (PTE)** determines the **physical frame number (PFN)**.

We’ll use a simple linear structure for now. We’ll use more complex data structures in later sections to help ease certain paging difficulties.

As for the contents of each PTE, there are several different **bits** to understand.
* A **valid bit** indicates if a translation is valid.
* For example, when a program starts, the code and heap are at one end of the address space and the stack at the other. The unused space in between will be marked **invalid** If the process tries to access this memory, it will trigger an OS trap, which will likely terminate the process.
* By marking all unused pages in the address space illegal, we eliminate the need to allocate actual frames for those pages, saving huge amounts of memory.

- **Protection bits** may also signal whether a page can be read, written to or executed. Trying to read a page in a way that these bits don’t allow will result in an OS trap.
- A **present bit** shows whether a page is in RAM or disk (i.e., it has been **swapped out**).
    - We’ll learn more about this mechanism when we look at how to **swap** parts of the address space to disk to support address spaces larger than physical memory.
- A **dirty bit** indicates if the page has been modified since it was brought into memory.
- A **reference bit** (also known as an **accessed bit**) is occasionally used to track whether a page has been accessed, and is so preserved in memory. This information is crucial during **page replacement**, which we will explore in depth in later sections.



.guides/img/paging8

The graphic above shows an x86 architecture page table entry. It contains:
* A present bit - **P**
* A read/write bit - **R/W**
* This determines if writes are allowed on this page
* A user/ supervisor bit - **U/S**
* This determines if user-mode processes can access the page
* **PWT**, **PCD**, **PAT**, and **G** determine how hardware caching will work for these pages
* An accessed bit (**A**), and
* A dirty bit (**D**)
* This dictates how hardware caching works for these pages.
* Finally, the **page frame number (PFN)**.

Paging is Slow!

We already know that page tables in memory may be too big. **They can potentially slow things down**.

Consider this instruction as an example:

```
movl 21, %eax
```

Let's look at the explicit reference to address 21 and not so much at the instruction fetch. We'll assume the hardware does the translation for us. To retrieve the required data, the system has to first translate the virtual address (21) into the proper physical address (117). As a result, before requesting the data at address 117, the system has to:
1. Get the correct page table entry from the process's page table
1. Translate it, and
1. Load the data from physical memory

The hardware has to know the process's page table location to do this.

Let's say, for now, that a single **page-table base register** has the page table's physical address. The hardware will conduct the following functions to locate the necessary PTE:

```
VPN     = (VirtualAddress & VPN_MASK) >> SHIFT
PTEAddr = PageTableBaseRegister + (VPN * sizeof(PTE))
```

In our example:
* `VPN_MASK` is set to `0x30` (hex `30`, or binary `110000`), and
* `SHIFT` is set to `4` (the number of bits in the offset), such that the VPN bits generate the right integer virtual page number.
* For example, masking turns virtual address 21 (010101) into 010000. Shifting puts it into 01, or virtual page 1. The page table base register's index into the array of PTEs.

In order to get the physical address, the hardware has to:
* Retrieve the PTE from memory,
* Extract the PFN, and
* Combine it with the virtual address offset.

The PFN is left-shifted by `SHIFT`, then bitwise OR'd with the offset to generate the final address:

```
offset   = VirtualAddress & OFFSET_MASK
PhysAddr = (PFN << SHIFT) | offset
```

Finally, the device can read data from memory into register `eax`. The application now has a value loaded from memory!

In short, we can describe the initial memory reference mechanism. The code example to the left shows us this approach. To use paging, we have to first get the translation from the page table for every memory reference (instruction fetch, explicit load, or store). That takes effort! This will slow down the procedure by a factor of two or more.

Now you can see that we have two serious issues to address. Page tables **slow down the system** and **consume too much memory** if not designed carefully. In order to use this for memory virtualization, we have to first solve these two problems.

Summary

We introduced the concept of **paging** to solve the challenges of virtualizing memory. There are many advantages to **paging** over prior methods (like segmentation).
* The first benefit is that **paging does not cause external fragmentation** since by design, memory is split into fixed-sized segments.

- In addition, it is **adaptable**, allowing for minimal use of virtual address spaces.

- Implementing paging functionality carelessly will result in a **slower computer** (due to the additional memory visits required to reach the page table) and memory waste (with memory filled with page tables instead of useful application data).

Lab Intro

In this lab, we’ll use a program, `paging-linear-translate.py` to explore virtual-to-physical address translation with linear page tables.

Let’s run this program with the `-h` flag to see the options we have with this program.

```
./paging-linear-translate.py -h
```

Run the program without any arguments to see the natural output:

```
./paging-linear-translate.py
```

For each **virtual address**, your job is to either:

- Write down the physical address it translates to, or
- Write down that it is an out-of-bounds address (segmentation fault)

This program gives you a page table for a particular process. Recall that, in a real system with linear page tables, there is **one page table per process**. Here we focus on:

- One process
- Its address space, and so
- A single page table

For each virtual page number of the address space, the page table tells you that the virtual page is mapped to a specific physical frame number and is either valid or invalid.

The format of the page-table entry is:
* The left-most bit is the valid bit
* The remaining bit, provided that the valid bit is 1, is the PFN

In our example above, the page table maps:
* VPN 0 to PFN $0xc$ (decimal 12),
* VPN 3 to PFN 0x6 (decimal 6), and
* leaves the other two virtual pages, 1 and 2, as invalid.

Because the page table is a linear array, what is printed above is a representation of what you would see in memory if you checked the bits yourself.

Running this program with the `-v` flag also prints the VPN (index) into the page table. Run the example above with the `-v` flag:

```
./paging-linear-translate.py -v
```

Your task is to use this page table to convert virtual addresses to physical addresses.

Let's start with `VA 0x3229`. To convert this virtual address into a physical address, we have to:

- Break it down it into a virtual page number and an offset
  - We can do this by writing down the size of the **address space** and the **page size**.
  - In this case, the address space is $16KB$
  - The page size is $4KB$

We know that there are 14 bits in the virtual address and that the offset is 12 bits. This leaves 2 bits for the VPN. So, with our address $0x3229$, which is binary 11001000101001, we know the top two bits specify the VPN.

So, $0x3229$ is on virtual page 3 with an offset of $0x229$.

Next, we check the page table to see if VPN 3 is valid and mapped to a physical frame, and we find that it is (the high bit is 1 and it is mapped to physical page 6). As a result, we can build our final physical address by adding the physical page 6 to the offset, as follows:

- $0x6000 is the physical page shifted into the correct spot, or
- $0x0229$, the offset, gives us
- the final physical address, $0x6229

So we see that the virtual address, $0x3229$ translates to physical address $0x6229$

After you've computed the solutions yourself, run the program with the `-c` flag to see the answers.

```
./paging-linear-translate.py -c
```

You can change many of these parameters to make more interesting problems to solve. Remember to run the program with the `-h` flag to see what options there are.

- The `-s` flag changes the random seed and thus generates different page table values as well as different virtual addresses to translate.
- The `-a` flag changes the size of the address space.
- The `-p` flag changes the size of physical memory.

- The `-P` flag changes the size of a page.
- The `-n` flag can be used to generate more addresses to translate (instead of the default 5).
- The `-u` flag changes the fraction of mappings that are valid, from 0% (-u 0) up to 100% (-u 100). The default is 50, which means that roughly 1/2 of the pages in the virtual address space will be valid.
- The `-v` flag prints out the VPN numbers to make your life easier.

Lab 1

1. Before we start translating, let’s use the simulator to see how different parameters affect the size of linear page tables. Calculate the size of linear page tables as different parameters are changed. Below are some suggested inputs. You can see how many page-table entries are filled by using the `-v` flag. To start, consider how the size of a linear page table changes as the address space grows:

```
./paging-linear-translate.py -P 1k -a 1m -p 512m -v -n 0
./paging-linear-translate.py -P 1k -a 2m -p 512m -v -n 0
./paging-linear-translate.py -P 1k -a 4m -p 512m -v -n 0
```

Then, to see how the size of a linear page table changes as the page size grows, try the following:

```
./paging-linear-translate.py -P 1k -a 1m -p 512m -v -n 0
./paging-linear-translate.py -P 2k -a 1m -p 512m -v -n 0
./paging-linear-translate.py -P 4k -a 1m -p 512m -v -n 0
```

Try to consider the expected trends before running any of these.

- How should the size of page tables change as the address space grows?
- What happens as the page size grows?
- Why don’t we just use really large pages all the time?

2. Let’s get started with some translations. Begin with a few simple examples, then use the `-u` flag to alter the number of pages assigned to the address space. Consider the following:

```
./paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 0
./paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 25
./paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 50
./paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 75
./paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 100
```

- What happens as you increase the percentage of pages that are allocated in each address space?

3. Now, for some spice, let’s test a few other random seeds and some different (and sometimes pretty strange) address-space parameters:

```
./paging-linear-translate.py -P 8  -a 32   -p 1024 -v -s 1
./paging-linear-translate.py -P 8k -a 32k  -p 1m   -v -s 2
./paging-linear-translate.py -P 1m -a 256m -p 512m -v -s 3
```

- Which of these parameter combinations are unrealistic? Why?