

# CSE 410: Systems Programming

## Virtual Memory

Ethan Blanton

Department of Computer Science and Engineering  
University at Buffalo

# Virtual Memory

Virtual memory is a mechanism by which a system divorces the address space in programs from the physical layout of memory.

Virtual addresses are locations in program address space.

Physical addresses are locations in actual hardware RAM.

With virtual memory, the two need not be equal.

# Process Layout

As previously discussed:

- Every process has unmapped memory near NULL
- Processes may have access to the **entire address space**
- Each process is **denied access** to the memory used by other processes

Some of these statements seem **contradictory**.

Virtual memory is the **mechanism** by which this is accomplished.

Every address in a process's address space is a **virtual address**.

# Physical Layout

The **physical layout** of hardware RAM may vary significantly from machine to machine or platform to platform.

- Sometimes certain locations are **restricted**
- **Devices** may appear in the memory address space
- Different amounts of RAM may be present

Historically, programs were **aware of these restrictions**.

Today, **virtual memory** hides these details.

The kernel **must still be aware** of physical layout.

# The Memory Management Unit

The **Memory Management Unit (MMU)** **translates** addresses.

It uses a **per-process mapping structure** to transform **virtual addresses** into **physical addresses**.

The MMU is physical hardware between the CPU and the memory bus.

This translation must be **typically be very fast**, but occasionally has a **large performance penalty**.

Managing the translation mappings requires **tight integration** between the kernel and hardware.

# Address Spaces

Both **virtual** and **physical addresses** are in **address spaces**.

An **address space** is a range of **potentially valid locations**.

These spaces **need not be the same!**

- For example, on x86-64, the **virtual address space** is all locations from 0 to  $2^{64} - 1$ .
- Current x86-64 processors only allow **48 of those bits**.<sup>1</sup>
- A **given piece of hardware** may support **much less** memory.

---

<sup>1</sup>...in a somewhat strange fashion

# Linear Address Spaces

Many modern machines use a **linear address space**.

Linear addresses map to a small number of (sometimes one) **contiguous blocks of memory** in the **same address space** that are **address-disjoint**.

*In other words:*

- A particular address represents a **unique location** in the address space.
- **Every location in the address space** can be named with a single address.

# Segmented Address Spaces

Many older systems, and some modern systems, use **segmented address spaces**.

In a **segmented address space**, an address is divided into two (or more) parts:

- A **segment identifier**
- An **offset within the segment**

Each **segment** is often a **linear address space**.

The **segment identifier** may be **implicit** or **provided separately** from the address within the segment.

We will not consider segmented addresses further.



# Address Locations

The addresses we have used are **byte addresses**.

This is **not necessary**, however!

Some machines use **word addresses**, in particular.<sup>2</sup>

On a **word addressed** machine, **every address** is a word.

*E.g.*, address 0x1 would be the **second word**, or the **fifth byte**, on a 32-bit word machine!

---

<sup>2</sup>Early UNIX was developed on word-addressed machines (such as the PDP-7).

# The MMU

Every time the CPU accesses an address:

- The MMU intercepts that address
- It converts the **virtual address** from the **virtual address space** into a **physical address space**
- The converted address is used to access physical RAM

We call this **address translation**.

These address spaces **may not use the same addressing model**.

# Paging

There are **many possible virtual memory models**.

The x86-64 architecture offers several!

Linux on x86-64 uses **paging**.

In paged virtual memory, the **MMU** breaks memory into fixed-sized **pages**.

- There may be **several page sizes** in a system
- Page sizes are **typically powers of two**
- x86-64 small pages are 4 kB

# Page Mapping

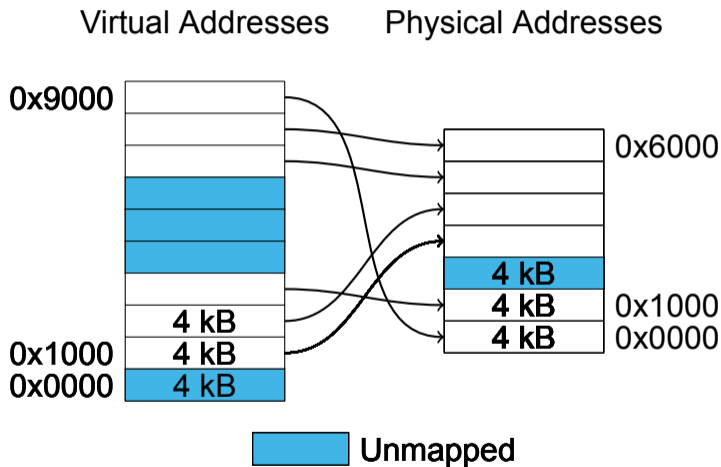
All addresses on a single page **share a translation**.

For example:

- Suppose that pages are 4 kB (0x1000 bytes hex).
- Suppose that a **page** has virtual address 0x8000.
- Suppose that that page is at **physical address** 0x1000.
- **All virtual addresses** between 0x8000 and 0x8fff will be mapped to physical addresses between 0x1000 and 0x1fff.

Different pages may be mapped **entirely differently**.

# Paging Example



# Page Tables

The MMU uses some **data structure** to perform address mapping.

On x86-64 (and many machines), it uses **page tables**.

Page tables are a **tree** of **arrays** containing **pointers and metadata**.

The **pointers** are to **physical addresses**.

The **metadata** describes what the pointers **point to**.

# Page Translation Example

Consider a system with a 14 bit pointer and 256 byte pages.

Each pointer on this system consists of:

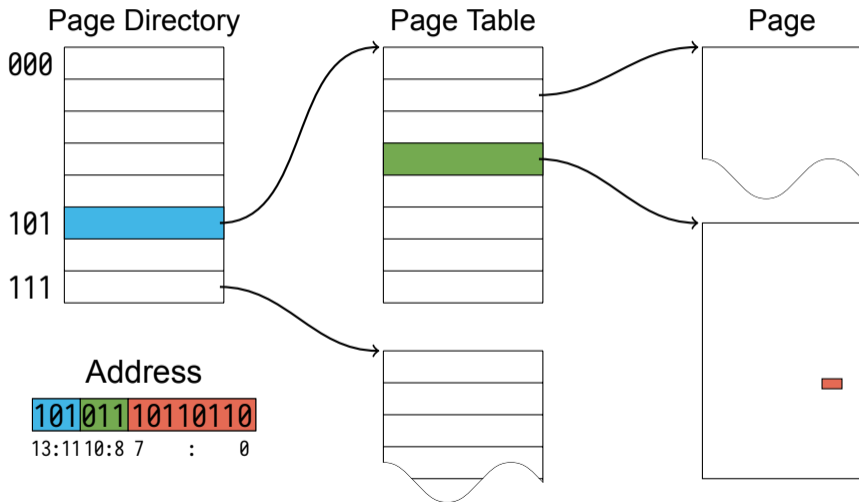
- a 6 bit **page identifier**
- an 8 bit **page offset**

101011	10110110				
13	:	8	7	:	0

The MMU **translates the page identifier to a physical page.**

This translation **may be performed in multiple steps.**

# Page Tables Example (14 b pointer, 256 B page)





# Paging Advantages

Paging allows **very high memory usage efficiency**.

By mapping **only the needed pages** from a process, its **occupied memory** can be **much smaller** than its **virtual address space**.

Processes can be interleaved in physical memory.

Even **page table structures** can be left out if unneeded.

(*I.e.*, if a range of pages is empty, a **page directory entry** can be empty.)

# Page Metadata

The **metadata** stored in page tables defines features like:

- Whether a virtual page is readable/writable
- If executing code from a virtual page is allowable
- Whether a virtual page is **currently present in memory**
- ...

If a memory access **violates this metadata**

(e.g., a write to a page that is not writeable):

- the MMU **notifies the processor**
- the processor **jumps to a particular kernel routine**
- the kernel:
  - **fixes** the problem, or
  - **notifies** the offending process

# Page Eviction

Virtual pages can be **backed** by files, physical pages, or both.

A **backed page** is **based on the contents** of its backing. If it is:

- **clean**: it is **identical to its backing**
- **dirty**: it is **different from its backing**

If the system is **low on memory**, it can **evict** a page.

A page is **evicted** by:

- **removing it from the map** if it is **clean**
- **writing it to its backing** and removing it from the map if it is **dirty**

# Demand Paging

In some cases, a virtual page may be **backed** but **not present**.  
(*E.g.*, if it has been **evicted**, or was simply never **paged in**.)

Such a page will be **marked as not present** in the page tables.

Attempts to access this page will **notify the kernel**.

This is called a **page fault**.

The kernel will **page in** the page by:

- finding an unused **physical page**
- locating the virtual page's **backing**
- reading the **backing data** into the physical page

# Demand Paging Benefits

Demand paging allows physical memory to be **allocated quickly** by simply updating page tables.

It also speeds **loading of executable files** as programs:

- pages are marked as **not present** but **backed by the file**
- access to pages causes the file to be **read into memory**
- **unused pages** are **never loaded**

It makes **fork()** **very efficient**:

- a new page table is created for the new process
- pages in both process's page tables are marked **read only**
- write to **shared pages** causes a **copy to be made**

# The Program Break

Calling `brk()` or `sbrk()` modifies a process memory map.

Additional pages **adjacent to the old break** will be marked as:

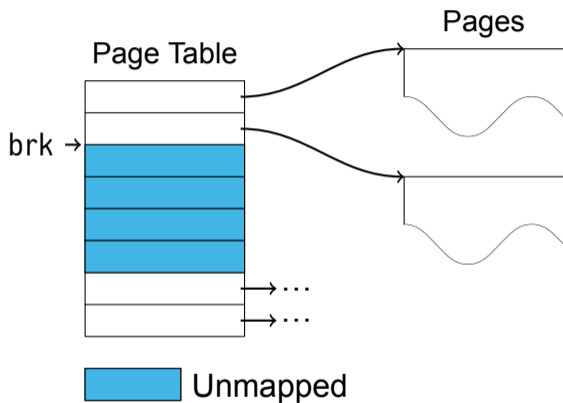
- **Not** present
- Readable and writable

However, **this affects only the page table metadata**, the pages are **not actually allocated!**

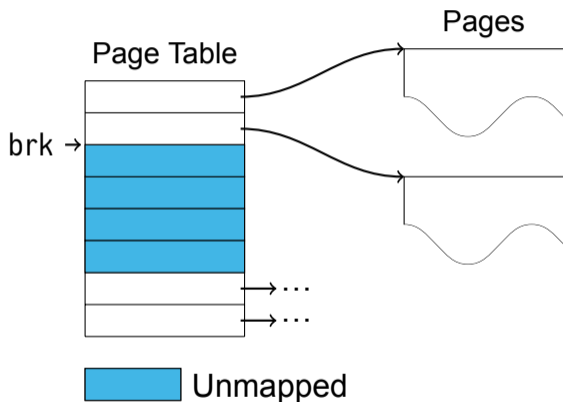
When the process tries to use a new page:

- The MMU will notify the processor
- The kernel will find an unused page (see Page Eviction)
- The kernel will **clear the unused page**
- The kernel will insert the page into the process's page tables

# Moving the Program Break (Page Fault)



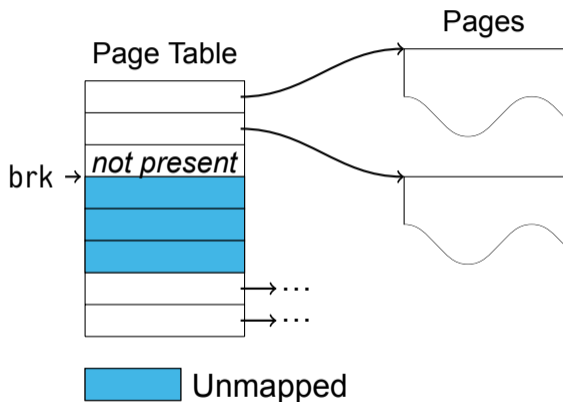
# Moving the Program Break (Page Fault)



`sbrk(PAGE_SIZE)` is called by the program.

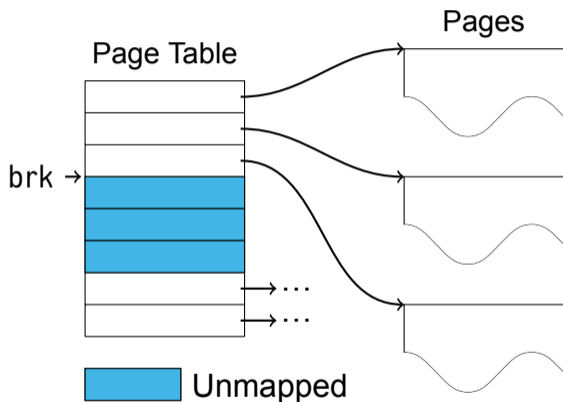


# Moving the Program Break (Page Fault)



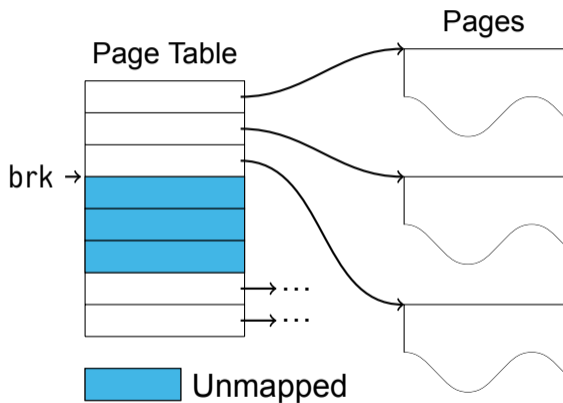
The break is moved, the new page is marked **not present**.

# Moving the Program Break (Page Fault)



Some time later, the process **attempts to access** the page.  
The MMU notifies the kernel, which allocates a page.

# Moving the Program Break (Page Fault)



The process's access to the page continues as normal.

# The C Stack

We previously said that **the kernel manages the program stack**:

- It grows as necessary (to some point)
- The program need not explicitly size it (cf. the break)

More correctly, the **kernel configures the MMU** to manage the program stack.

Similar to newly-allocated memory at the page break, at process creation the kernel will:

- Determine how large the program's stack should be
- Mark stack pages as **not present** but **readable and writeable**

As the **program stack grows**, **page faults** will allocate new pages.

# Summary

- **Virtual memory:**
  - uses a **memory management unit**
  - allows the CPU to operate in a **virtual address space** that may be different from the **physical address space**
  - the MMU **translates** virtual addresses to physical addresses
- **Paging** is a common model for virtual memory.
- Paged systems break **both address spaces** into **pages**.
- Pages can be **mapped individually** between virtual and physical addresses.
- **Page tables** allow the MMU to translate addresses.
- **Page faults** bring mapped but unallocated pages into memory.

# Next Time ...



# References I

## Required Readings

- [1] Randal E. Bryant and David R. O'Hallaron. *Computer Science: A Programmer's Perspective*. Third Edition. Chapter 9: Intro, 9.1-9.4. Pearson, 2016.

# License

Copyright 2018 Ethan Blanton, All Rights Reserved.

Reproduction of this material without written consent of the author is prohibited.

To retrieve a copy of this material, or related materials, see <https://www.cse.buffalo.edu/~eblanton/>.