Virtual Memory

CSE 220: Systems Programming

Ethan Blanton, Carl Alphonce, & Eric Mikida

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 typically be very fast, but occasionally has a large performance penalty.

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

Lecture Question

Ask a review question!

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.[1]
- A given piece of hardware may support much less memory.

[1] …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.[2]

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!

We will not consider word addressing further.

[2] Early Unix was developed on a word-addressed machine (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 Metadata

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, this is a page fault (*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 Backing

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

A backed page is based on the contents of its backing.

Backed pages may not need to be stored in memory at all times.

If it is:

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

Clean pages can be recreated from the backing at any time.

Demand Paging

In some cases, a virtual page may be backed but not present.

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

Attempts to access this page will notify the kernel.

(This is a type of 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

Lecture Question

Ask a VM question!

The Program Break

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

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

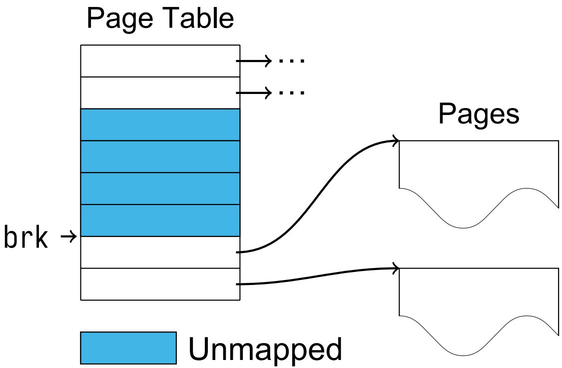
- Not present
- Readable and writable

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

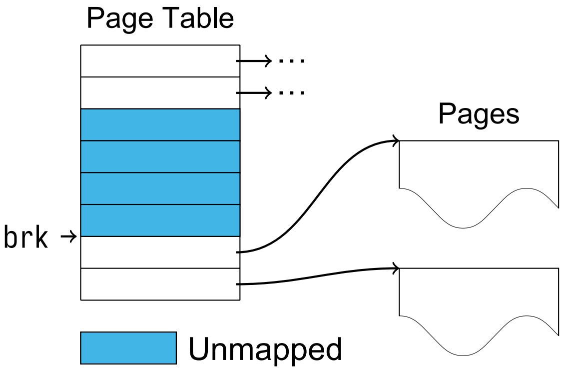
When the process tries to use a new page:

- The MMU will notify the processor
- The kernel will find an unused page
- The kernel will clear the unused page
- The kernel will insert the page into the process's page

Moving the Program Break (Page Fault)

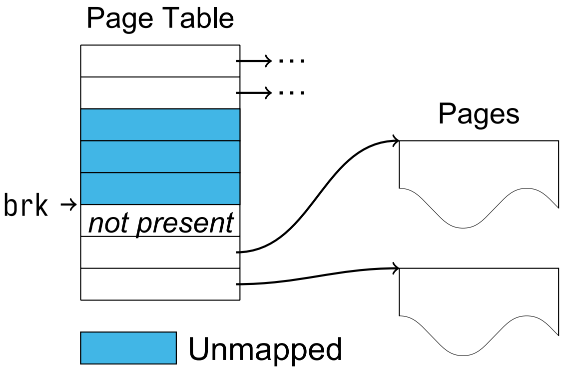


Moving the Program Break (Page Fault)



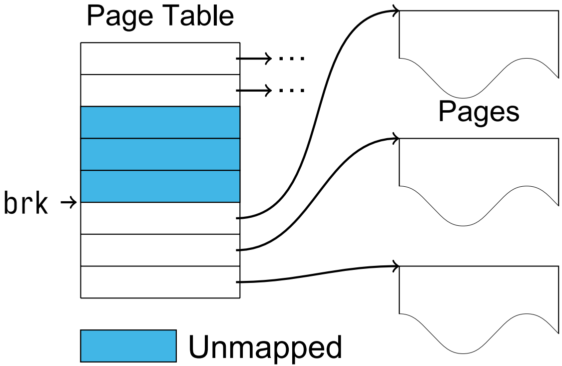
`sbrk(PAGE_SIZE)` is called by the program.

Moving the Program Break (Page Fault)



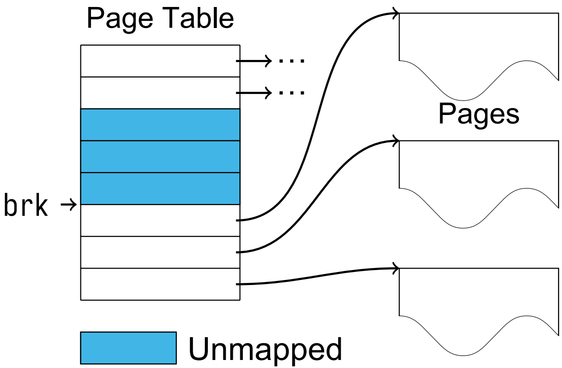
The break is moved, the new page is marked not present.

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.

Page Eviction

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

A page is evicted by:

- clean: simply remove it from the map
- dirty: write it to its backing and remove it

A special backing, swap, can back un-backed dirty pages.

Coupled with demand paging, page eviction can simulate extra memory.

1. A page is needed
2. No page is free
3. A page is evicted (maybe written to swap)
4. The evicted page is remapped

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.

References I

Required Readings

[2] Ian Weinand. *Computer Science from the Bottom Up*. Chapter 6: parts 1–4; part 7; part 8, 8.1 and 8.2. URL: `https://www.bottomupcs.com/index.html`.

Optional Readings

[1] Randal E. Bryant and David R. O'Hallaron. *Computer Science: A Programmer's Perspective*. Third Edition. Chapter 1: 1.7.3; Chapter 9: Intro, 9.1-9.4. Pearson, 2016.

License