Applications of Virtual Memory in OS Design

Nima Honarmand

Introduction

- Virtual memory is a powerful <u>level of indirection</u>
  - Indirection: IMO, the most powerful concept in Computer Science
- Fundamental Theorem of Software Engineering:

  "All problems in computer science can be solved by another level of indirection"

  -- David Wheeler
  - Except (perhaps) the problem of already having too many levels of indirection ☺
- So, what can we do with such a powerful primitive as virtual memory?

Some Ideas

- On-demand memory allocation
- Memory-mapped files
- Copy-on-Write (COW) fork
- Stack guards and automatic stack growth
- Virtual Dynamic Shared Object (VDSO)
- Interprocess communication
- Distributed Shared Memory
- Swapping (to use more virtual memory then physical RAM)
- Mapping kernel to same location in all address spaces
- …

Process Address Space Layout

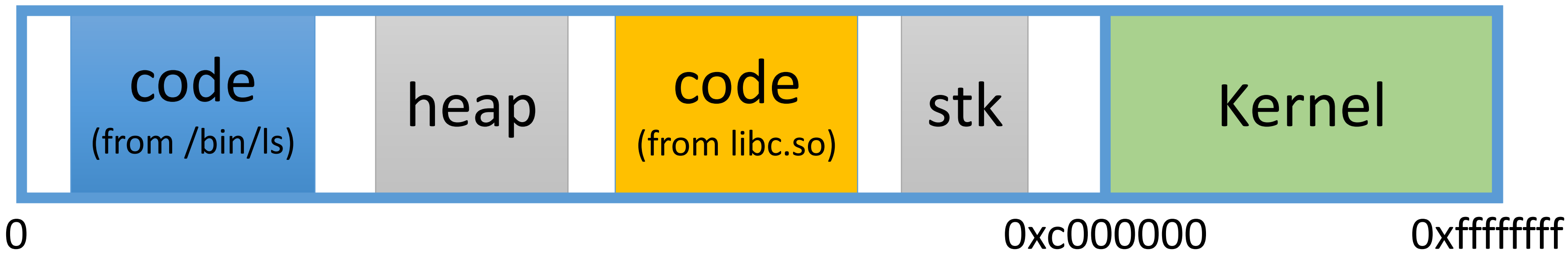
- To the above things, we need to keep some information about Process Address Space Layout
- Kernel always needs to know
  - *What is mapped to virtual address X of a process?*
  - *What are the restrictions of that mapping?*
- Kernel should somehow keep track of this information
  - Question: is a page table versatile enough for this?
  - Answer: Unlikely

  → We need a side data structure to store this info

In-Kernel Tracking of Virtual Memory Mappings

Simple Example

Virtual Address Space (4GB)

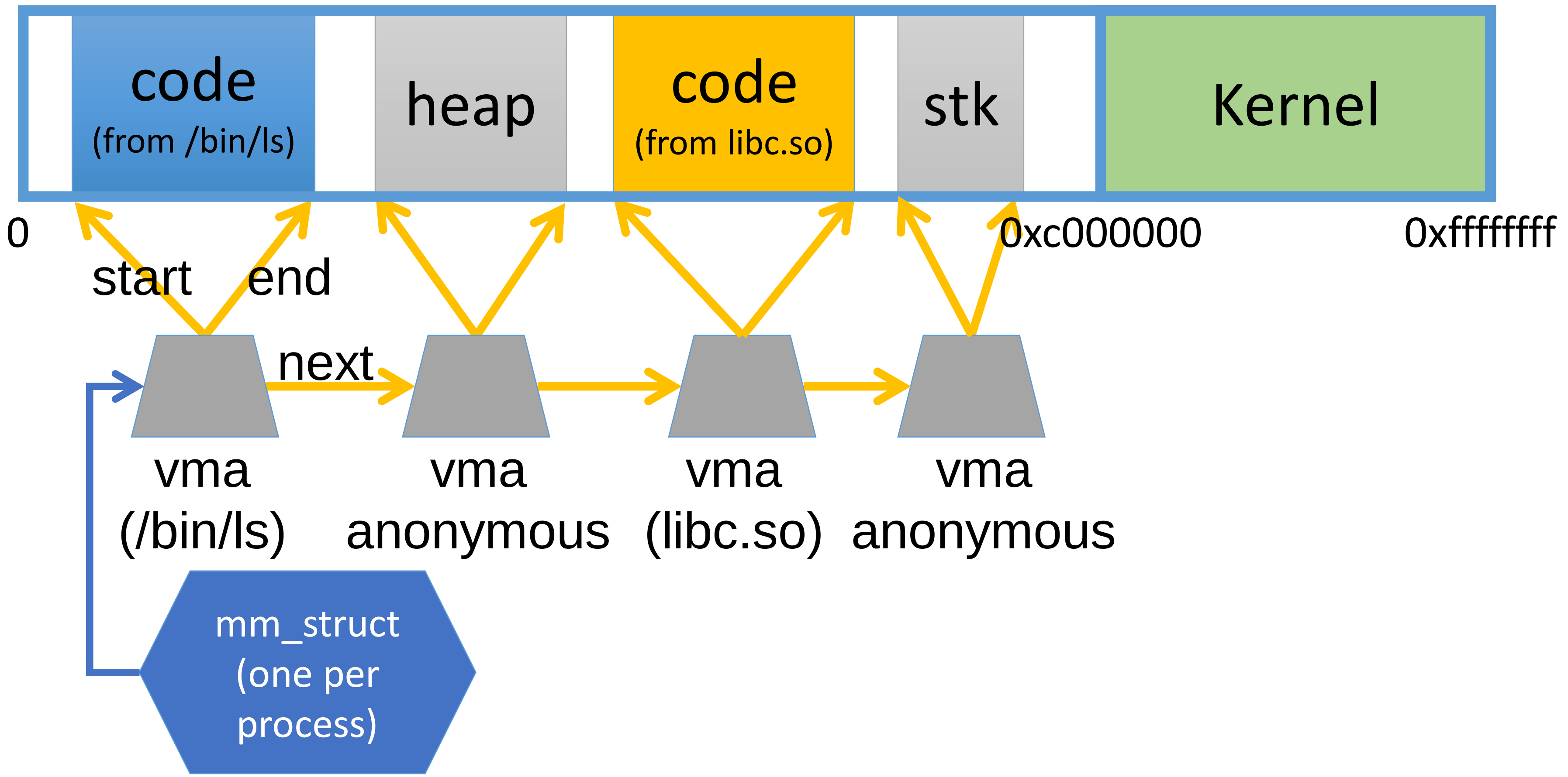


- “/bin/ls” binary specifies load address
- Optionally, specifies where it wants libc
  - And other libraries it uses
- Dynamically asks kernel for “anonymous” pages for its heap and stack
  - Anonymous = not from a file

How to Represent in the Kernel?

- Linux represents portions of a process with a `vm_area_struct`, or VMA
- Includes:
  - Start address (virtual)
  - End address (first address after VMA) – why?
    - Memory regions are page aligned
  - Protection (read, write, execute, etc.) – implication?
    - Different page protections means new VMA
  - Pointer to file (if one)
  - Other bookkeeping

Simple VMA List Representation



Process Address Space Layout

- Determined (mostly) by the application
  - Partly determined at compile time
    - Link directives can influence this
  - Application dynamically requests new mappings from the OS, or deletes existing mappings, or changes protection flags, etc.
- OS usually reserves part of the address space to map itself
  - E.g., upper 1GB on 32-bit x86 Linux

Key Unix/Linux API: `mmap()`

- `void * **mmap**(void *addr, size_t length, int prot, int flags, int fd, off_t offset)`
- Arguments:
  - `addr`: virtual address where program wants the region (0 means anywhere is okay)
  - `length`: size of the mapped region
  - `prot`: protection flags
  - `flags`: field or'ing a bunch of flag (future slides)
  - `fd`: file descriptor for memory-mapped file regions
  - `offset`: offset of the region within the file
- Return value:
  - Beginning address of the mapped region (>= 0)
  - Error (< 0)

Anonymous Mappings with `mmap()`

- To ask for memory that is not backed by any file — hence “anonymous”
- `flags` should contain `MAP_ANONYMOUS`
- OS will create a new VMA of the right size at the appropriate location
  - Based on `addr` if not zero, otherwise wherever there is a big-enough hole in the address space
- Information related to `flags` and `prot` are stored in VMA for the OS to know how to treat this VMA

Memory-Mapped Files and `mmap()`

- `mmap()` can be used to map part of a file in the address space
  - If `flags` contain `MAP_FILE`, then kernel looks at `fd` and `offset` fields
- Another way of accessing files in addition to `read()`/`write()`
- After mem-mapping a part of a file, you can use regular load/store instructions to access that part

Memory-Mapped Files and `mmap()`

- OS allocates a big-enough region of the address space, <u>copies that part of the file to the region</u>, and returns region's beginning address
  - Again, returned addr. depends on whether `addr` is 0 or not
- This is the main mechanism used for loading .text and .data from binary files and shared libraries
- Also, data bases use it to access their data
- Windows API: `CreateFileMapping()`

Other important mmap() Flags

- MAP_FIXED
  - Do not treat addr as a hint; mmap() should fail if it cannot allocate at addr
- MAP_HUGETLB/MAP_HUGE_2MB/MAP_HUGE_1GB
  - Use huge pages for this part of the address space
- MAP_SHARED/MAP_PRIVATE (for mem-mapped files)
  - Should my writes to this file be visible to other processes mem-mapping it?
  - Should my writes carry through to the file itself?

Memory-Mapped File Example

```
char *p; int fd; struct stat sb;

fd = open("/my_file.txt", O_RDONLY);
fstat(fd, &sb);
p = mmap(0, sb.st_size, PROT_READ, MAP_SHARED, fd, 0);
close (fd);

for (len = 0; len < sb.st_size; len++)
    putchar(p[len]);

munmap(p, sb.st_size);
```

Open the file (in the right mode)

Get its size (and other stats)

mmap the part that you need (doesn't have to be all of it)

Can close the file descriptor now.

Access the mmaped memory.

Unmap the file from address space

- **DISCLAIMER:** This code is over-simplified. In reality, there should be quite a bit of error checking after each system call.

Other Related Syscalls

- **`munmap`**`(void *addr, size_t length)`
  - Removes the given region from VMAs, potentially truncating/splitting existing VMAs
- **`mremap`**`(void *old_address, size_t old_size, size_t new_size, int flags, ... /* void *new_address */)`
  - Expands (or shrinks) an existing region, potentially moving it to a new address (depends on flags)
- **`mprotect`**`(void *addr, size_t len, int prot)`
  - Changes the access protection for the given range of virtual memory addresses

OS is Lazy

- On `mmap()`, most OSes just find a big enough hole in address space, create a VMA and keep `mmap()` arguments in it
  - No physical memory is allocated initially
  - No file access is performed (if mem-mapped file)
  - Even the page table is not modified at that point
- A page fault happens upon the first access to an address in that range
  - Because there is no mapping in the page table
- Based on VMA info, OS determines if the access should be allowed

OS is Lazy (cont'd)

- If VMA is anonymous, OS allocates a physical page and adds it to page table at the accessed address
  - The page is typically zeroed out for security reasons

- If VMA is file-backed, OS allocates a page and copies corresponding file data to it

→ OS only allocates physical memory on-demand, when a valid virtual address is accessed for the first time

Laziness Rules in OS

- As a general rule, OSes try to be as lazy as possible
  - Postpone doing something until it is absolutely necessary (OS; not you!)
- On-demand physical page allocation is one example
- Second example: file writes are not sent immediately to disk
  - Kept in memory and written back in the background when system load is low
- Third example: copy-on-write `fork()`

Laziness Rules in OS (cont'd)

- Laziness generally makes OS more responsive
  - System calls can return more quickly than otherwise
    - Acknowledge program's request, and do the work later when it is really necessary
- Laziness often eliminates unnecessary work, e.g.
  - Will not allocate memory if user never touches it
  - Program might link with many libraries but not use most of them (or parts of them) in a given run
  - Why write to disk immediately if the same file data will be written again soon
  - Why write to a disk file at all if it is going to be deleted?
    - Happens a lot with temp files
  - And numerous other examples…

Applications of Virtual Memory

1) On-Demand Paging

- Discussed previously in slides on “OS Laziness”

2) Memory-Mapped Files

- Makes file content accessible using simple load/store instructions
  - No need to pay the cost of `read()/write()` system calls
- Combined with demand paging, allows mapping large portions of file in the address space with little cost
  - Read a file page from disk and allocate physical page for it upon first access (on-demand)
- Allows OS to share same file pages between multiple processes
  - If mappings are read-only or `MAP_SHARED`
  - <u>OS only keeps one copy in physical memory and map it to multiple address spaces</u>
  - We'll discuss this more in the context of page caches
- Very useful in dealing with shared libraries

3) Copy-On-Write (COW) `fork()`

- Recall: `fork()` creates and starts a copy of the process; identical except for the return value

- Example:

```
int pid = fork();
if (pid == 0) {
    // child code
} else if (pid > 0) {
    // parent code
} else {
    // error
}
```

Copy-On-Write (COW) `fork()`

- Naïve `fork()` would march through address space and copy each page
  - As xv6 does
- But most processes immediately `exec()` a new binary without using any of these pages
- Even if not followed by an `exec()`, much of parent's pages may never be touched

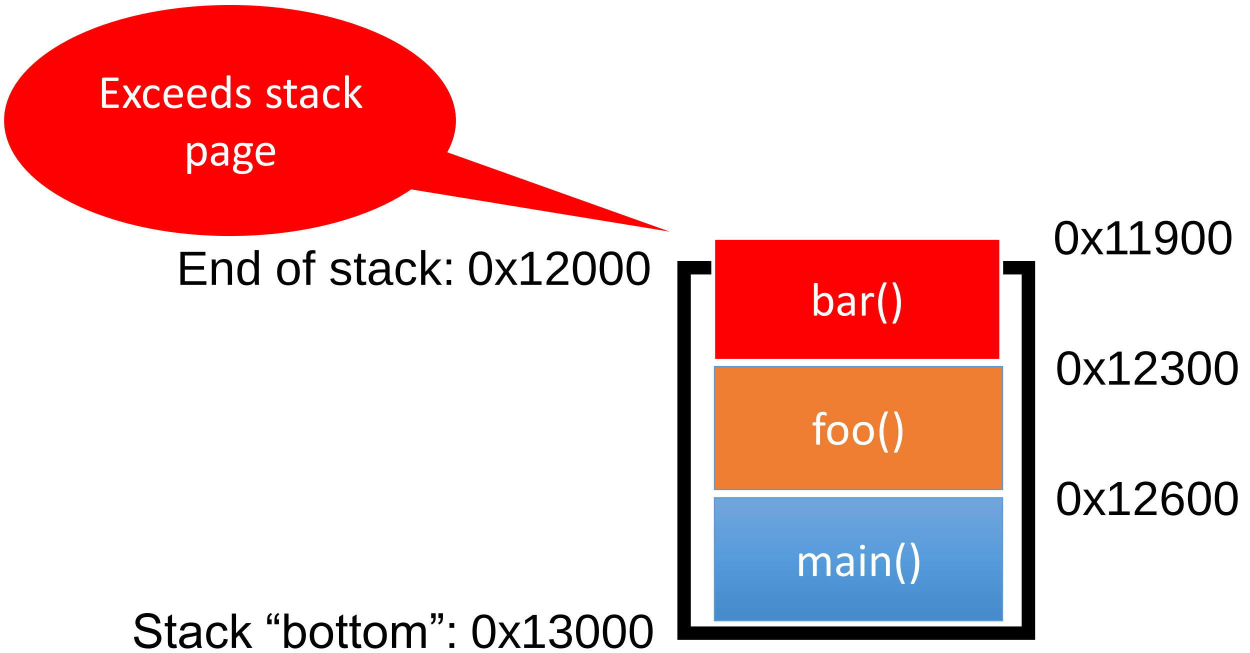
→ Being lazy is better!

How does COW work?

- Memory regions:
  - New copies of all VMAs are allocated for child during fork
  - As are page tables
- Pages in memory:
  - In page table (and in-memory representation), clear write bit, set COW bit
    - Is the COW bit hardware specified?
    - No, OS uses one of the available bits in the PTE
      - But it does not have to; can just keep the info in the VMA like other meta data
  - Make a new, writeable copy on a write page fault
- You will add COW fork to xv6 in Lab2!

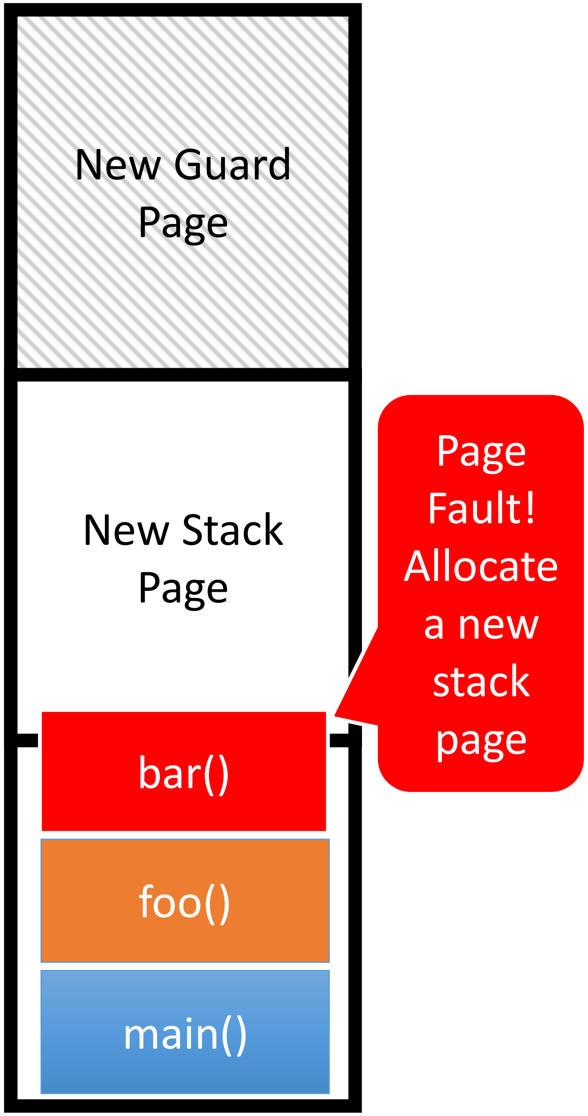
4) Automatic Stack Growth

- Recall: in x86, as you add call frames to a stack, they decrease in virtual address order
- Example:



How to Do This?

- How to support automatic stack growth?
- OS allocates a guard page in the address space below the stack and marks it as not-accessible
  - Just a virtual mapping, no physical page frame
- On a page fault in the guard page, OS knows it needs to grow the stack
- Allocates a physical page and moves the guard page up (i.e., lower address)



5) Interprocess Communication

- OS maps physical pages of memory into multiple process address spaces
  - Enables shared-memory communication between processes
  - Of course, processes should be careful about concurrency issues when accessing such memory
- See `shm...()` family of system calls in Unix/Linux

6) VDSO: Virtual Dynamic Shared Object

- A small shared library mapped automatically (by kernel) into the address space of each process
- Used to reduce the cost of some frequent system calls even further
  - E.g., `getpid()`, `gettimeofday()`
- Goal: turn the system call into a simple function call
  - No need to save/restore context, switch privilege levels, jump to kernel, etc.
- Done by mapping the data needed to serve the system call and (maybe) the code to access that data into the process address space
- You will implement this in Lab 3!

7) Distributed Shared Memory

- Motivation: allow a virtual address space to span multiple physical computers processes
  - Gives your program a lot more memory or processing power than can be had on a single computer
- Gives the illusion of physical shared memory, across a network
- E.g., can be used in scientific computing languages using a Partitioned Global Address Space (PGAS) model
  - UPC (Unified Parallel C), X10, etc.

Distributed Shared Memory (cont'd)

- A virtual address in a process may be "logically" mapped to a physical page that resides on another computer
  - In such a case, on a page fault on that address, you need to get a copy of that page from the machine where it resides
- How?
  - Replicate pages that are read-only
  - Invalidate copies on write
  - How to know if a page is only read or also written?

8) Swapping

- Transparently (i.e., w/o programmer involvement), use more virtual memory than available physical memory
  - One very large process
  - Multiple processes whose combined virtual memory size is more than available physical memory
- Idea: when out of physical memory, swap one physical memory page out to disk and use its space
  - Problem 1: how to select the victim page to be swapped out?
  - Problem 2: how to find which page tables have a mapping for this victim?
- Will cover in detail in conjunction with disks and file systems

Summary

- Virtual memory is a very powerful indirection layer with many application is OS design
- Whenever you see a level of indirection (or, similar concepts such as “virtual”, “abstract”, etc.), ask yourself “what else can I do with it?”