Virtual Memory & Process Address Space

Nima Honarmand

Virtual Memory (VM) Recap

- Primary purpose?
  - Isolate each process to its own address space
- Components
  - Address translation (virtual to physical)
  - Pre-defined permission flags: user/kernel, read/write, present, NX
    → OS to CPU communication
  - Pre-defined Access and Dirty flags
    → CPU to OS communication
  - Software-defined flags
    → OS to OS communication

Things We Can Do with VM

- Hine: VM is a powerful level of indirection
  - Level of indirection: perhaps the most powerful concept in Computer Science

1. Lazy/on-demand physical memory allocation
   - OS can actually allocate physical pages only when a process tries to access it
   - How?
2. Share kernel page tables across processes
   - How?

Things We Can Do with VM

3. Guard page to protect against stack overflow
   - Put a non-mapped page below user stack
   - If stack overflows, application will see page fault
   - Allocate more stack space if that happens
4. Copy-on-write fork
   - Motivation: fork() is often followed by exec(), so no point in copying all the address space on fork()
   - Solution: do copy-on-write fork()
   - How?
5. Use more virtual memory than physical memory
   - How?

Things We Can Do with VM

6. Memory-mapped files
   - Motivation: allow access to files using load/store instructions, rather than having to call read()/write() every time
   - Combine memory-mapped files and demand paging
     - Page-in pages of a file on demand when memory is full, page-out pages of a file that are not frequently used
   - Great for quickly launching programs
     - Load code from the executable file or shared-library on-demand
   - Combine memory-mapped files and virtual-memory sharing
     - Read-only file pages can be shared between multiple processes
     - Again, very useful for shared libraries

Things We Can Do with VM

7. Inter-Process Communication using shared memory
   - How?
8. Distributed Shared Memory
   - Motivation: allow processes on different machines to share virtual memory
   - 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.
   - How?
     - Replicate pages that are only read
     - Invalidate copies on write
     - How to know if a page is only read or also written?

Things We Can Do with VM

- What else can you think of?
  - Use your imagination; VM is a very powerful concept

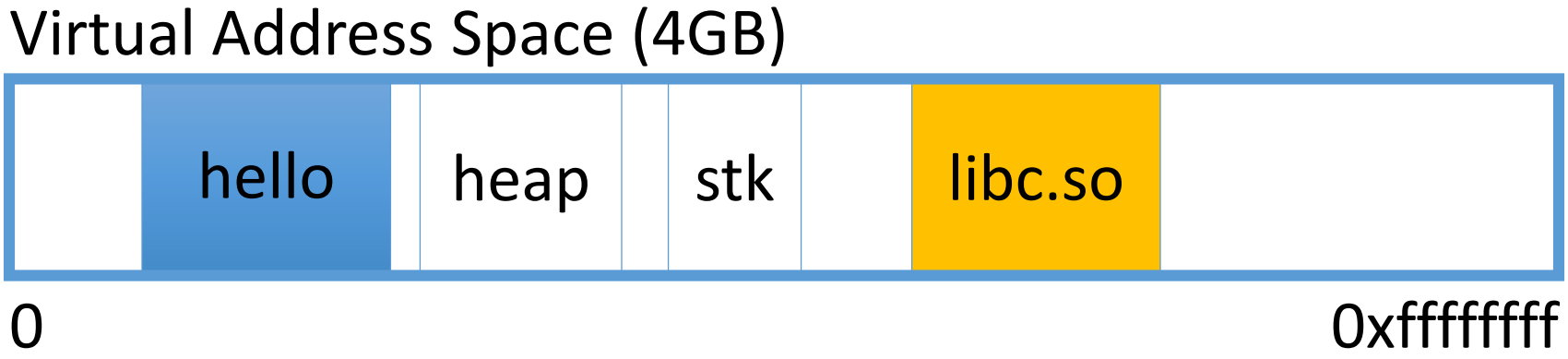
Keeping Track of Virtual Memory Mappings

Process Address Space Layout

- To be able to do many of the above things, we need to keep a lot of information about the 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 information

Simple Example

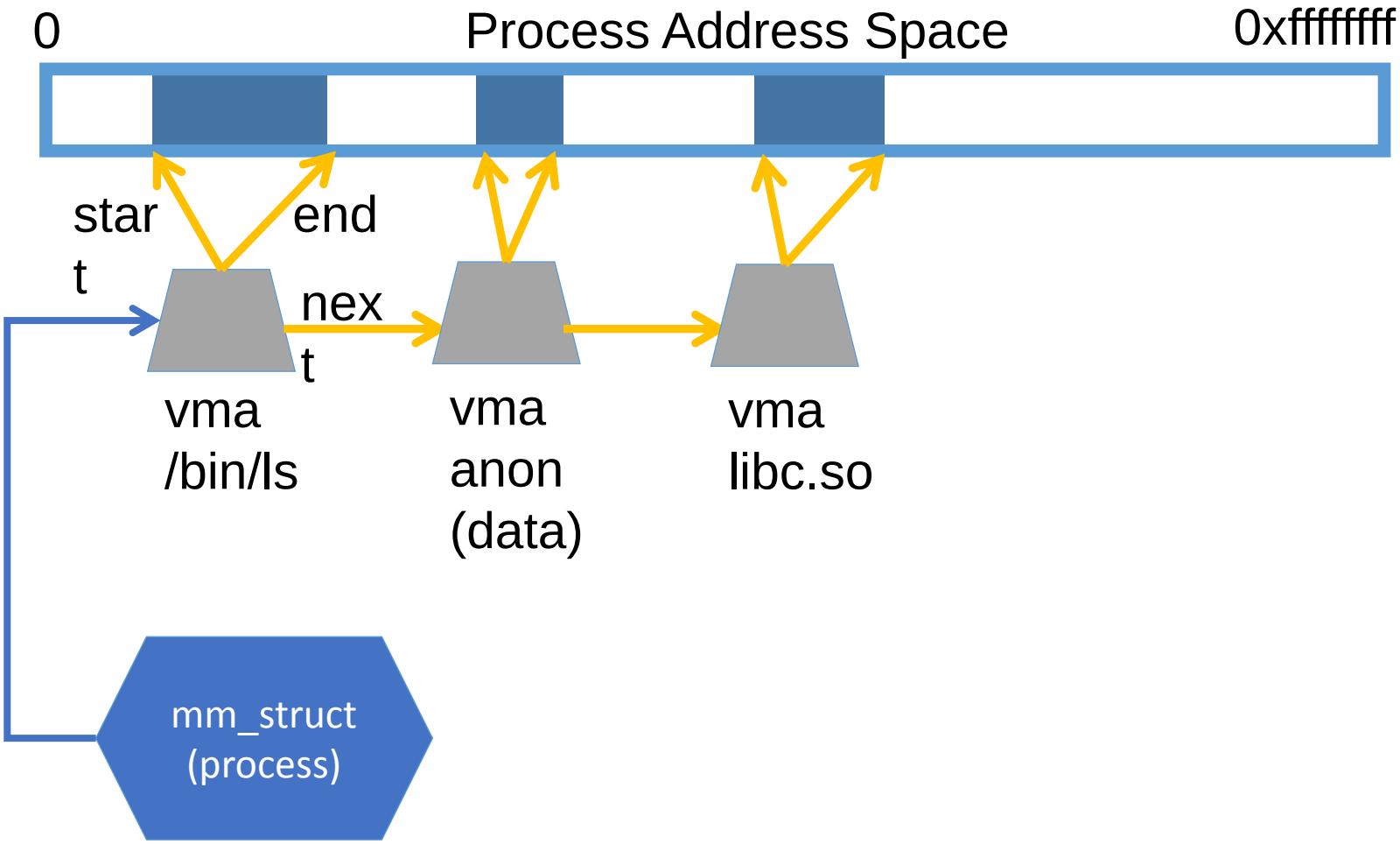


- "Hello world" binary specifies load address
- Optionally, specifies where it wants libc
- Dynamically asks kernel for "anonymous" pages for its heap and stack
  - Anonymous = not from an mmap()ed file

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



Simple list

- Linear traversal – O(n)
  - Shouldn't we use a data structure with the smallest O?
- Practical system building question:
  - What is the common case?
  - Is it past the asymptotic crossover point?
- If tree traversal is O(log n), but adds bookkeeping overhead, which makes sense for:
  - 10 vmas: log 10 =~ 3; 10/2 = 5; Comparable either way
  - 100 vmas: log 100 starts making sense

Common cases

- Many programs are simple
  - Only load a few libraries
  - Small amount of data
- Some programs are large and complicated
  - Databases
- Linux splits the difference and uses both a list and a red-black tree

Red-black trees

- (Roughly) balanced tree
  - Popular in real systems
- Asymptotic == worst case behavior
  - Insertion, deletion, search: log n
  - Traversal: n
- Read the Wikipedia article if not familiar with them

Back to Address Space Layout

- Determined (mostly) by the application
- Partly determined at compile time
  - Link directives can influence this
    - See kern/kernel.ld in JOS; specifies kernel starting address
- Application can dynamically request new mappings from the OS, or delete mappings
- OS usually reserves part of the address space to map itself
  - E.g., upper GB on 32-bit x86 Linux

Linux APIs

- **mmap**(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
- **munmap**(void *addr, size_t length);

- How to create an anonymous mapping?
- What if you don't care where a memory region goes (as long as it doesn't clobber something else)?

Demand paging

- Creating a memory mapping (vma) doesn't necessarily allocate physical memory or setup page table entries
  - What mechanism do you use to tell when a page is needed?
- It pays to be lazy!
  - A program may never touch the memory it maps.
    - Examples?
      - Program may not use all code in a library
  - Save work compared to traversing up front
  - Hidden costs? Optimizations?
    - Page faults are expensive; heuristics could help performance

Unix `fork()`

- Recall: this function 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)

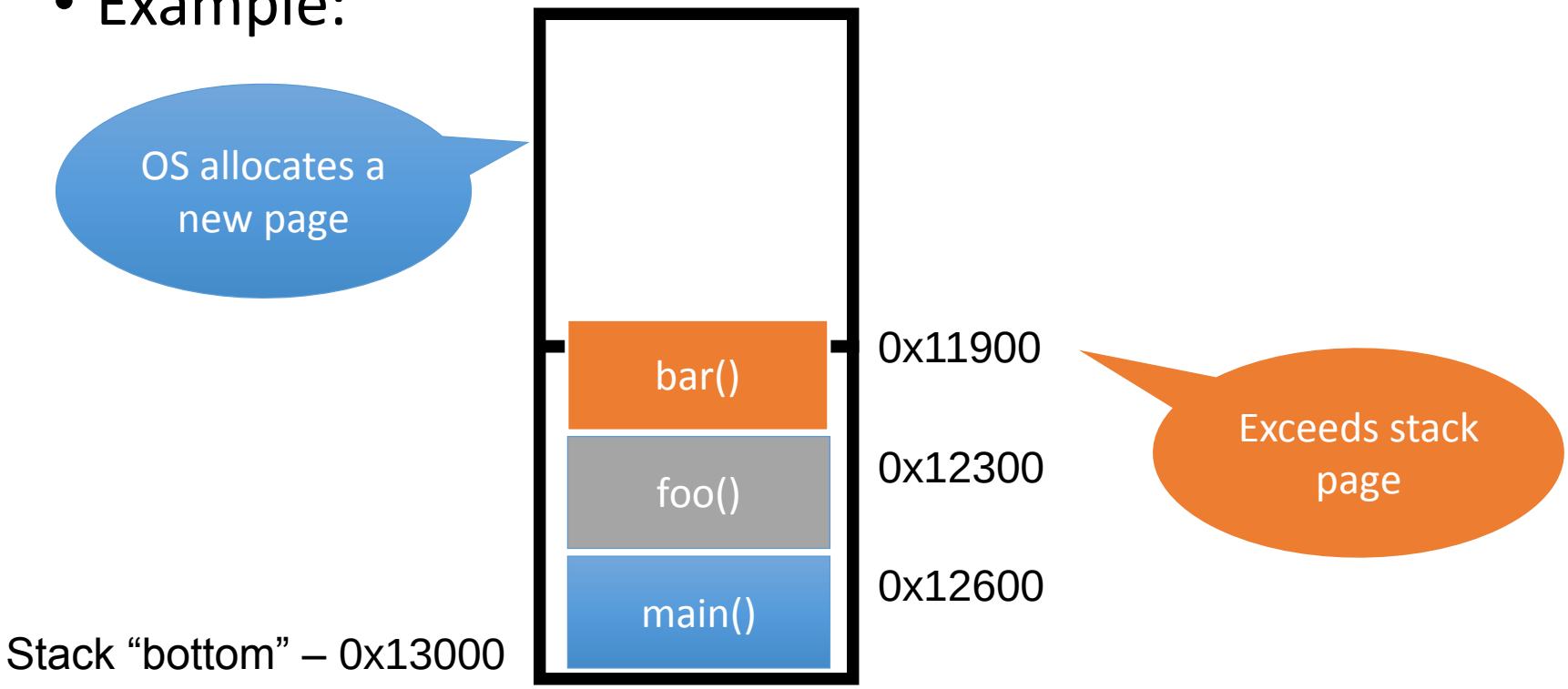
- Naïve approach would march through address space and copy each page
  - Most processes immediately `exec()` a new binary without using any of these pages
  - Again, lazy is better!

How does COW work?

- Memory regions:
  - New copies of each vma 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 fault

Stacks

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



Problem 1: Expansion

- Recall: OS is free to allocate any free page in the virtual address space if user doesn't specify an address
- What if the OS allocates the page above the "top" of the stack?
  - You can't grow the stack any further
  - Out of memory fault with plenty of memory spare
- OS must reserve stack portion of address space
  - Fortunate that memory areas are demand paged

Feed 2 Birds with 1 Scone

- Unix has been around longer than paging
  - Remember data segment abstraction?
  - Unix solution:

- Stack and heap meet in the middle
  - Out of memory when they meet

But now we have paging

- Unix and Linux still have a data segment abstraction
  - Even though they use flat data segmentation!
- `brk()` system call adjusts the endpoint of the heap
  - Still used by many memory allocators today
- Today, most modern libraries use `mmap()` instead of `brk()`
  - But we still need to support `brk()` for legacy code

Program Binaries and Address Space Layout

Program Binaries

- How are address spaces represented in a binary file?

- How are processes loaded?

Linux: ELF

- *Executable and Linkable Format*
- Standard on most Unix systems
  - And used in JOS
  - You will implement part of the loader in lab 3
- 2 types of headers:
  - Program header: 0+ segments (memory layout)
  - Section header: 0+ sections (linking information)

Helpful tools

- `readelf` - Linux tool that prints part of the elf headers
- `objdump` – Linux tool that dumps portions of a binary
  - Includes a disassembler; reads debugging symbols if present

Key ELF Sections

- `.text` – Where read/execute code goes
  - Can be mapped without write permission
- `.data` – Programmer initialized read/write data
  - Ex: a global int that starts at 3 goes here
- `.bss` – Uninitialized data (initially zero by convention)
- Many other sections

Sections

- Also describe text, data, and bss segments
- Plus:
  - Procedure Linkage Table (`.plt`) – trampoline table for libraries
  - Global Offset Table (`.got`) – data/code addresses for libraries
  - `.rel.text` – Relocation table for external targets
  - `.symtab` – Program symbols

ELF Segments vs. Sections

- ELF sections represent a compiler/linker's view of the program binary
- ELF segments represent the in-memory view of the program (corresponding to VMAs)

- For example, you might have multiple *code sections* generated by the compiler
  - .text, .ctor (constructors), .dtor(destructors), .init (run only once), …
  - You can put all of these in one *code segment* of the ELF to be loaded at one (instead of separately)

How ELF Loading Works

- `execve("foo", …)`
- Kernel parses the file enough to identify whether it is a supported format
  - Kernel loads the text, data, and bss segments
- ELF header also gives first instruction to execute
  - Kernel transfers control to this application instruction

Static vs. Dynamic Linking

- Static Linking:
  - Application binary is self-contained
- Dynamic Linking:
  - Application needs code and/or variables from an external library
- How does dynamic linking work?
  - Each binary includes a “jump table” for external references
  - Jump table is filled in at run time by the dynamic linker

Simplified jump table example

- Suppose I want to call foo() in another library
- Compiler allocates an entry in the jump table for `foo`
  - Say it is index 3, and an entry is 8 bytes
- Compiler generates local code like this:

```
mov rax, 24(rbx)   // rbx points to the
                   // jump table
call *rax
```

- Linker initializes the jump tables at runtime
- **Note:** Actual PLT/GOT mechanism used today is more complicated. The general idea is similar though. (See the lecture readings for details)

Dynamic Linking (Overview)

- Rather than loading the application, load the linker (ld.so), give the linker the actual program as an argument
- Kernel transfers control to linker (in user space)
- Linker:
  - 1) Walks the program's ELF headers to identify needed libraries
  - 2) Issue mmap() calls to map in said libraries
  - 3) Fix the jump tables in each binary
  - 4) Call main()