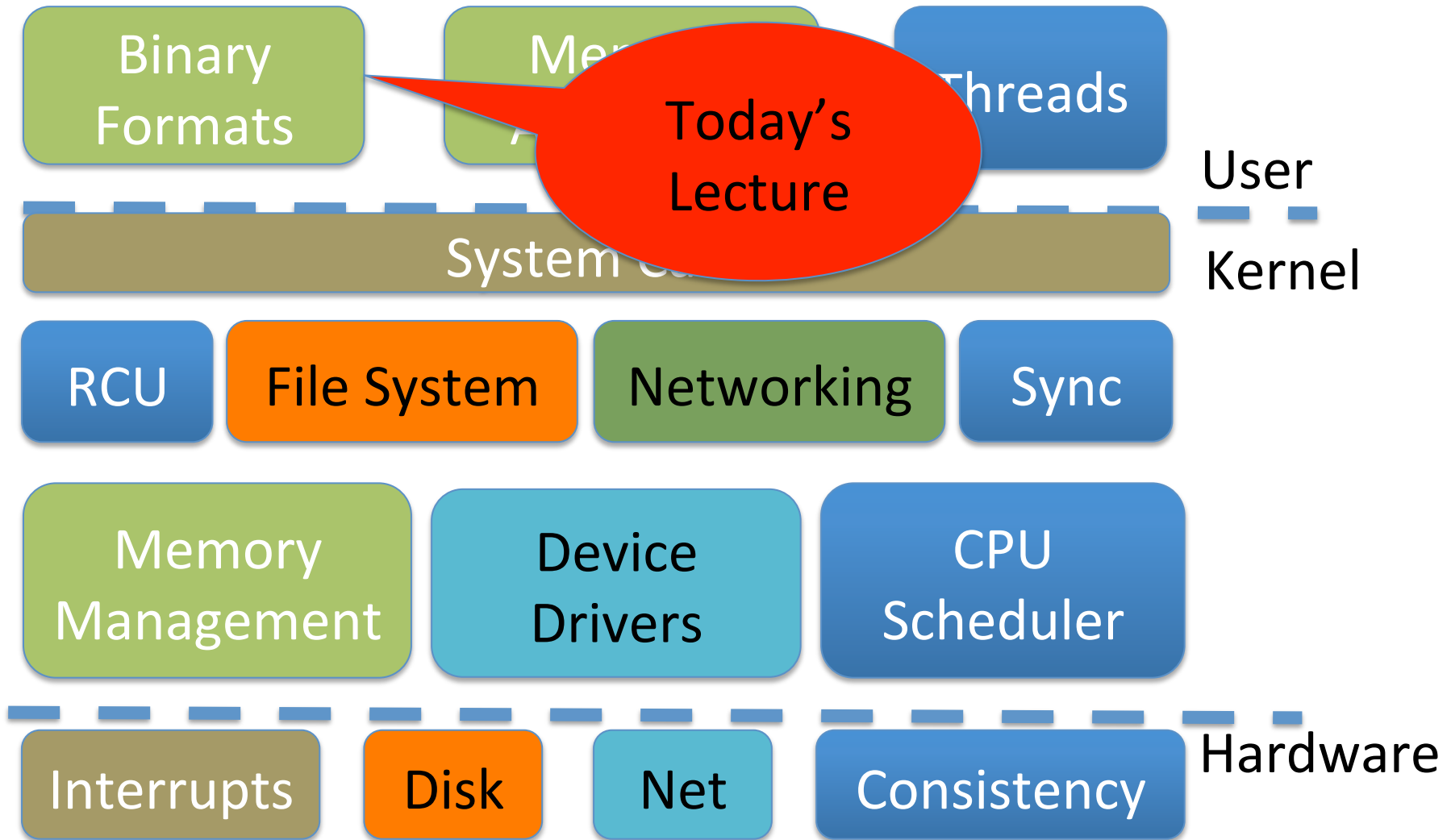


Process Address Spaces and Binary Formats

Don Porter

Logical Diagram



Review

- We've seen how paging and segmentation work on x86
 - Maps logical addresses to physical pages
 - These are the low-level hardware tools
- This lecture: build up to higher-level abstractions
- Namely, the process address space

Definitions (can vary)

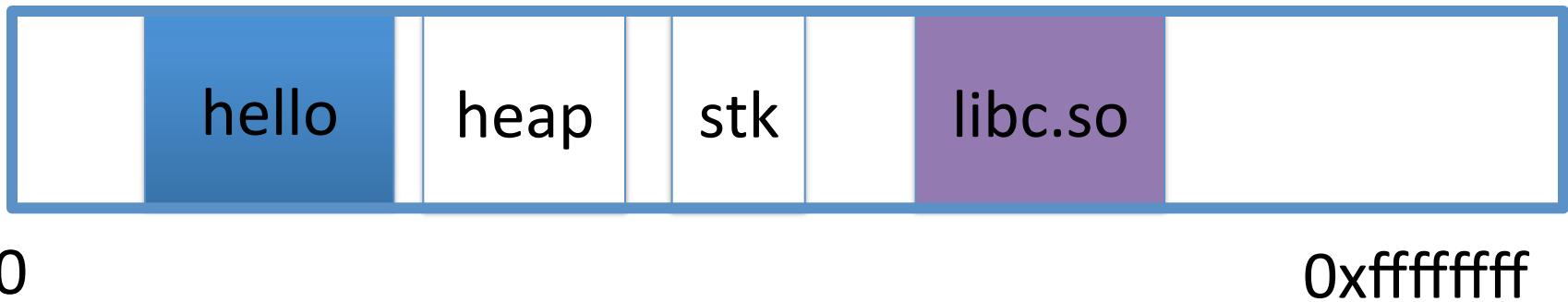
- Process is a virtual address space
 - 1+ threads of execution work within this address space
- A process is composed of:
 - Memory-mapped files
 - Includes program binary
 - Anonymous pages: no file backing
 - When the process exits, their contents go away

Address Space Layout

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

Simple Example

Virtual Address Space



- “Hello world” binary specified load address
- Also specifies where it wants libc
- Dynamically asks kernel for “anonymous” pages for its heap and stack

In practice

- You can see (part of) the requested memory layout of a program using `ldd`:

```
$ ldd /usr/bin/git
linux-vdso.so.1 => (0x00007fff197be000)
libc.so.6 => /lib/libc.so.6
(0x00007f31b97ac000)
/lib64/ld-linux-x86-64.so.2
(0x00007f31b9f86000)
libpthread.so.0 => /lib/libpthread.so.0
(0x00007f31b9b31000)
libz.so.1 => /lib/libz.so.1
(0x00007f31b9d4e000)
```

Problem 1: How to represent in the kernel?

- What is the best way to represent the components of a process?
 - Common question: is mapped at address x ?
 - Page faults, new memory mappings, etc.
- Hint: a 64-bit address space is seriously huge
- Hint: some programs (like databases) map tons of data
 - Others map very little
- No one size fits all

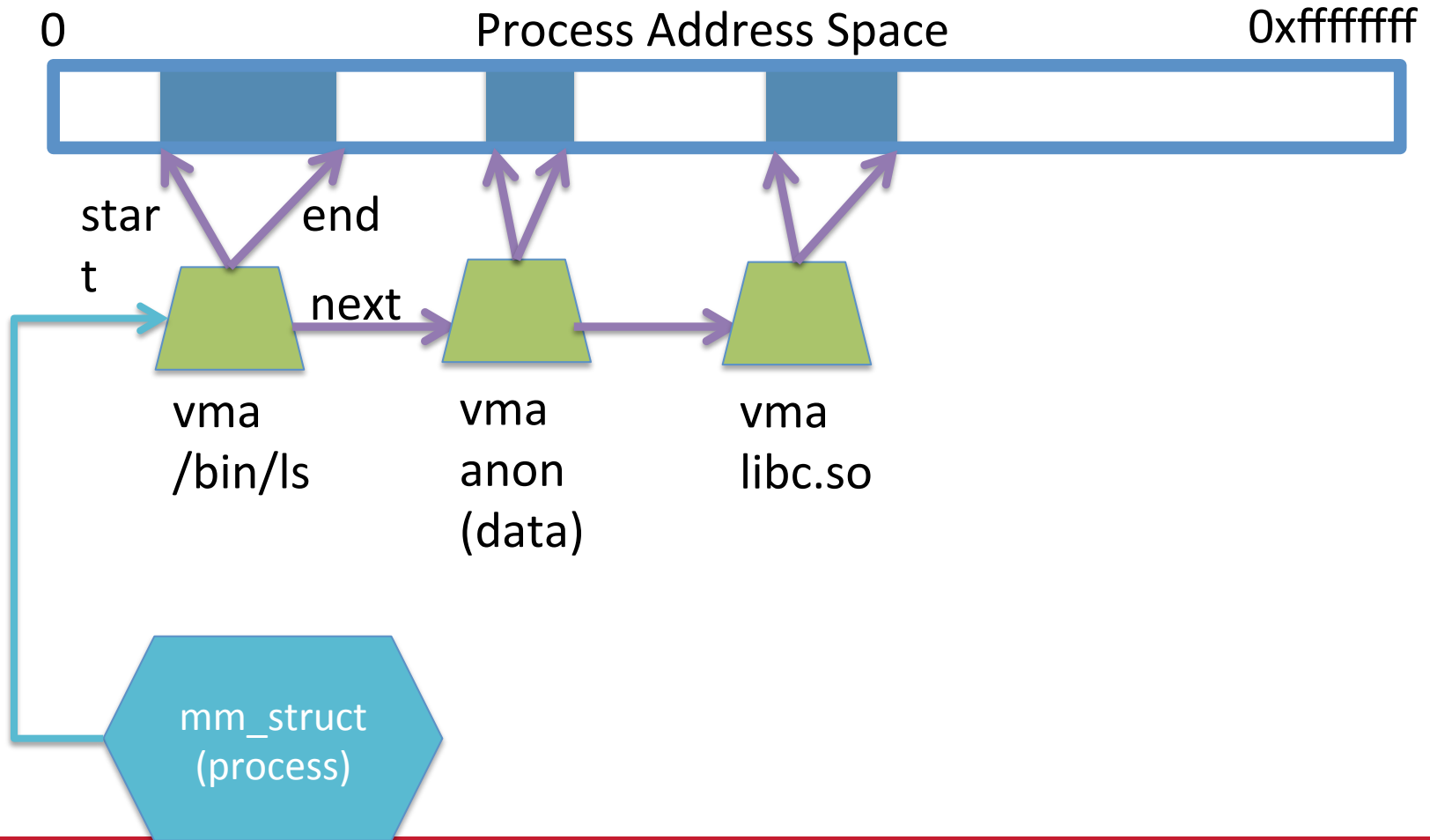
Sparse representation

- Naïve approach might make a big array of pages
 - Mark empty space as unused
 - But this wastes OS memory
- Better idea: only allocate nodes in a data structure for memory that is mapped to something
 - Kernel data structure memory use proportional to complexity of address space!

Linux: `vm_area_struct`

- 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 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: $\log 10 \approx 3$; $10/2 = 5$; 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
- Read the wikipedia article if you aren't familiar with them
- Popular in real systems
 - Asymptotic == worst case behavior
 - Insertion, deletion, search: $\log n$
 - Traversal: n

Optimizations

- Using an RB-tree gets us logarithmic search time
- Other suggestions?
- Locality: If I just accessed region x , there is a reasonably good chance I'll access it again
 - Linux caches a pointer in each process to the last vma looked up
 - Source code (mm/mmap.c) claims 35% hit rate

Memory mapping recap

- VM Area structure tracks regions that are mapped
 - Efficiently represent a sparse address space
 - On both a list and an RB-tree
 - Fast linear traversal
 - Efficient lookup in a large address space
 - Cache last lookup to exploit temporal locality

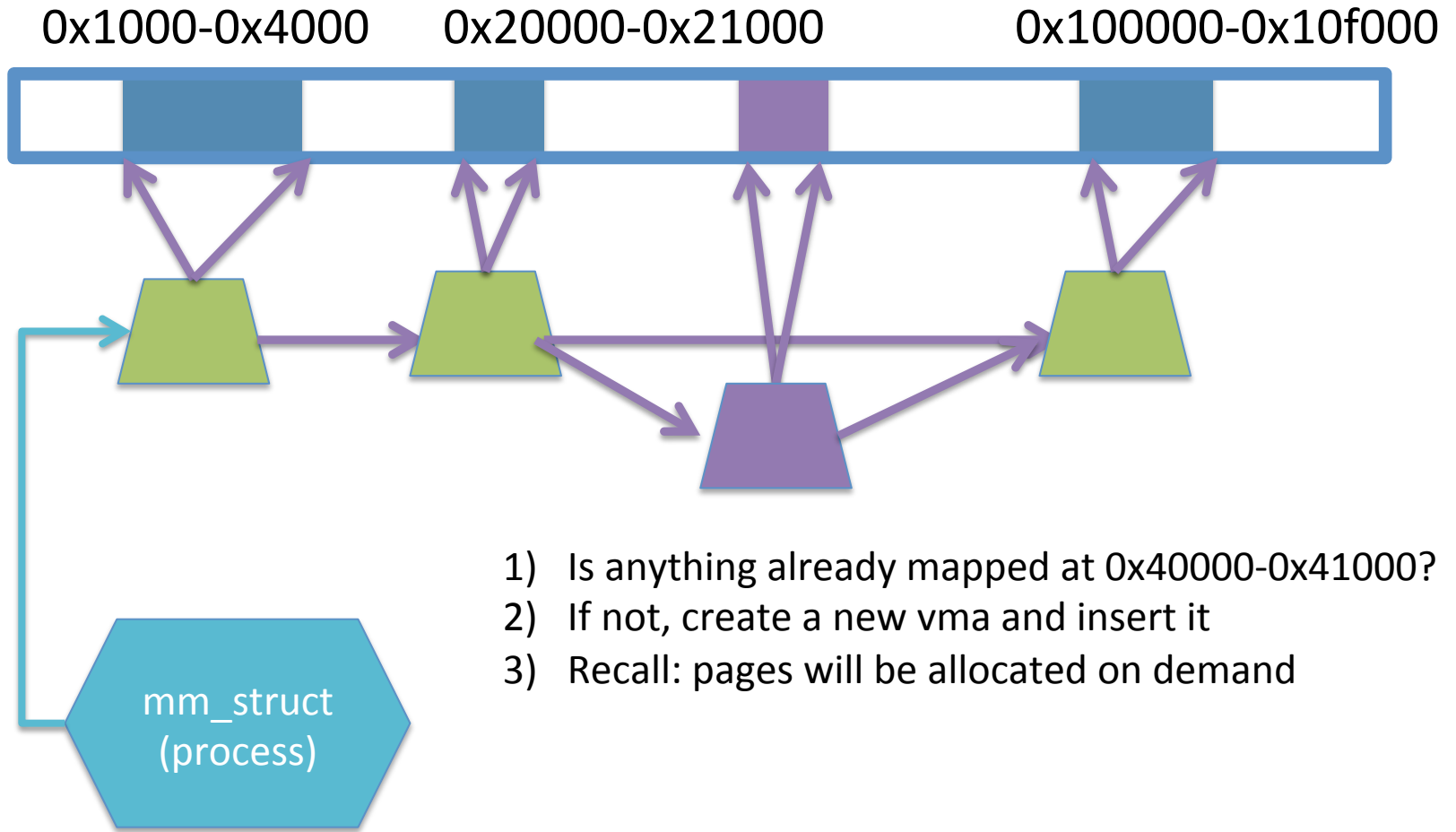
Linux APIs

- `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)?

Example 1:

- Let's map a 1 page (4k) anonymous region for data, read-write at address 0x40000
- `mmap(0x40000, 4096, PROT_READ|PROT_WRITE, MAP_ANONYMOUS, -1, 0);`
 - Why wouldn't we want exec permission?

Insert at 0x40000

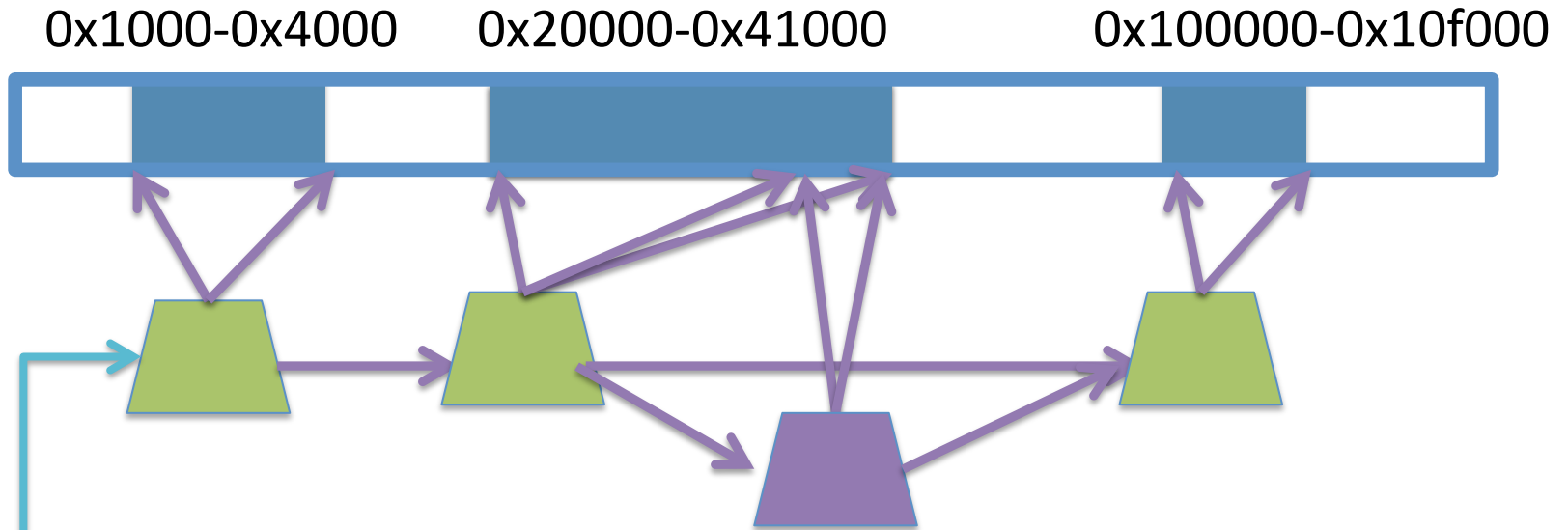


- 1) Is anything already mapped at 0x40000-0x41000?
- 2) If not, create a new vma and insert it
- 3) Recall: pages will be allocated on demand

Scenario 2

- What if there is something already mapped there with read-only permission?
 - Case 1: Last page overlaps
 - Case 2: First page overlaps
 - Case 3: Our target is in the middle

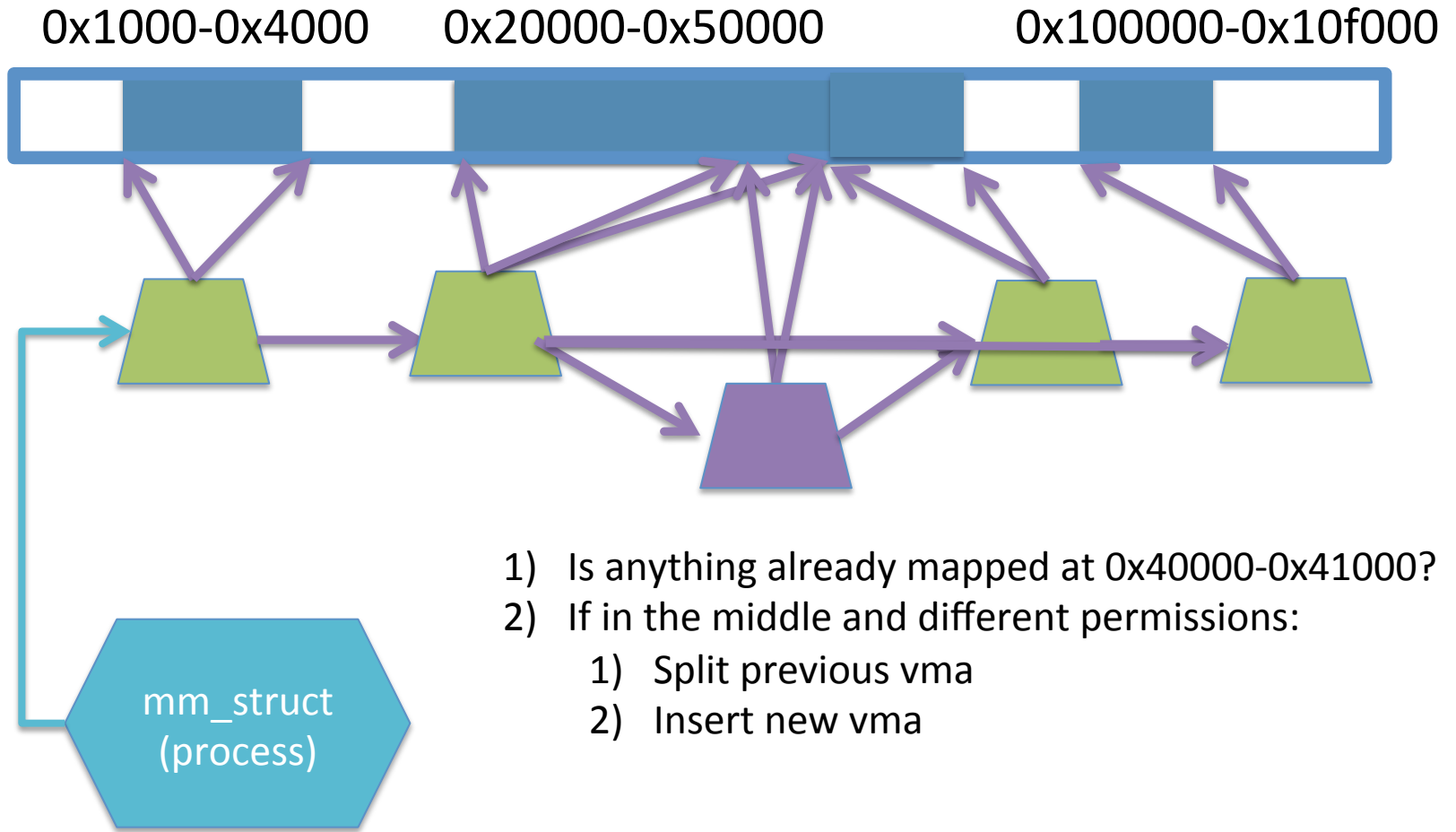
Case 1: Insert at 0x40000



mm_struct
(process)

- 1) Is anything already mapped at 0x40000-0x41000?
- 2) If at the end and different permissions:
 - 1) Truncate previous vma
 - 2) Insert new vma
- 3) If permissions are the same, one can replace pages and/or extend previous vma

Case 3: Insert at 0x40000



- 1) Is anything already mapped at 0x40000-0x41000?
- 2) If in the middle and different permissions:
 - 1) Split previous vma
 - 2) Insert new vma

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!

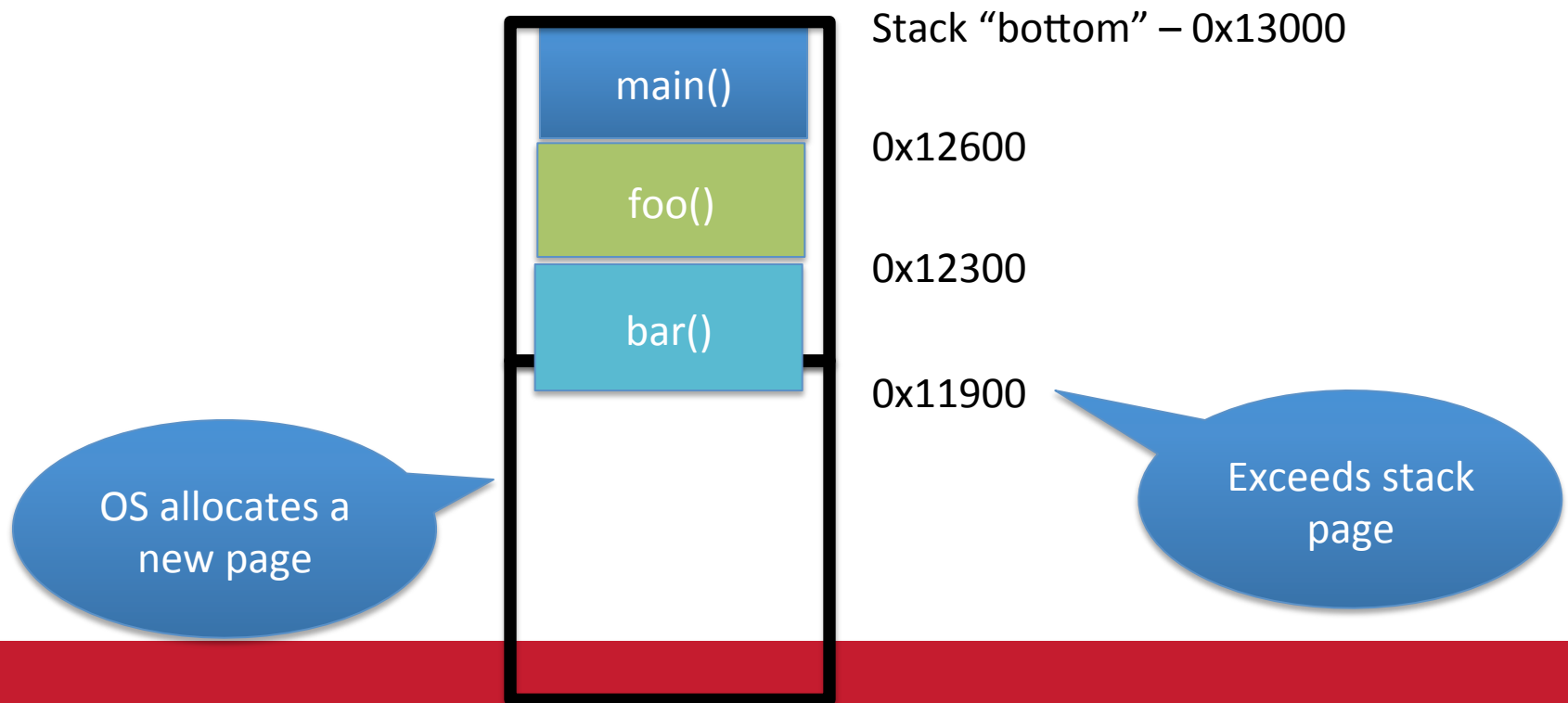
How does COW work?

- 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
 - Make a new, writeable copy on a write fault

New Topic: Stacks

Idiosyncrasy 1: Stacks Grow Down

- In Linux/Unix, 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 below 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!
- `sys_brk()` adjusts the endpoint of the heap
 - Still used by many memory allocators today

Windows Comparison

- LPVOID VirtualAllocEx(__in HANDLE hProcess, __in_opt LPVOID lpAddress, __in SIZE_T dwSize, __in DWORD flAllocationType, __in DWORD flProtect);
- Library function applications program to
 - Provided by ntdll.dll – the rough equivalent of Unix libc
 - Implemented with an undocumented system call

Windows Comparison

- LPVOID VirtualAllocEx(__in HANDLE hProcess,
 __in_opt LPVOID lpAddress,
 __in SIZE_T dwSize,
 __in DWORD flAllocationType,
 __in DWORD flProtect);
- Programming environment differences:
 - Parameters annotated (__out, __in_opt, etc), compiler checks
 - Name encodes type, by convention
 - dwSize must be page-aligned (just like mmap)

Windows Comparison

- LPVOID VirtualAllocEx(__in HANDLE hProcess,
 __in_opt LPVOID lpAddress,
 __in SIZE_T dwSize,
 __in DWORD flAllocationType,
 __in DWORD flProtect);
- Different capabilities
 - hProcess doesn't have to be you! Pros/Cons?
 - flAllocationType – can be reserved or committed
 - And other flags

Reserved memory

- An explicit abstraction for cases where you want to prevent the OS from mapping anything to an address region
- To use the region, it must be remapped in the committed state
- Why?
 - My speculation: Gives the OS more information for advanced heuristics than demand paging

Part 1 Summary

- Understand what a vma is, how it is manipulated in kernel for calls like mmap
- Demand paging, COW, and other optimizations
- brk and the data segment
- Windows VirtualAllocEx() vs. Unix mmap()

Part 2: 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 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 Segments

- For once, not the same thing as hardware segmentation
 - Similar idea, though
- `.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 segments

Sections

- Also describe text, data, and bss segments
- Plus:
 - Procedure Linkage Table (PLT) – jump table for libraries
 - .rel.text – Relocation table for external targets
 - .symtab – Program symbols

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 sections
- 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 loader

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`
- Loader initializes the jump tables at runtime

Dynamic Linking (Overview)

- Rather than loading the application, load the loader (ld.so), give the loader the actual program as an argument
- Kernel transfers control to loader (in user space)
- Loader:
 - 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()

Recap

- Understand basics of program loading
- OS does preliminary executable parsing, maps in program and maybe dynamic linker
- Linker does needed fixup for the program to work

Summary

- We've seen a lot of details on how programs are represented:
 - In the kernel when running
 - On disk in an executable file
 - And how they are bootstrapped in practice
- Will help with lab 3