Process Address Spaces and Binary Formats

Don Porter

Background

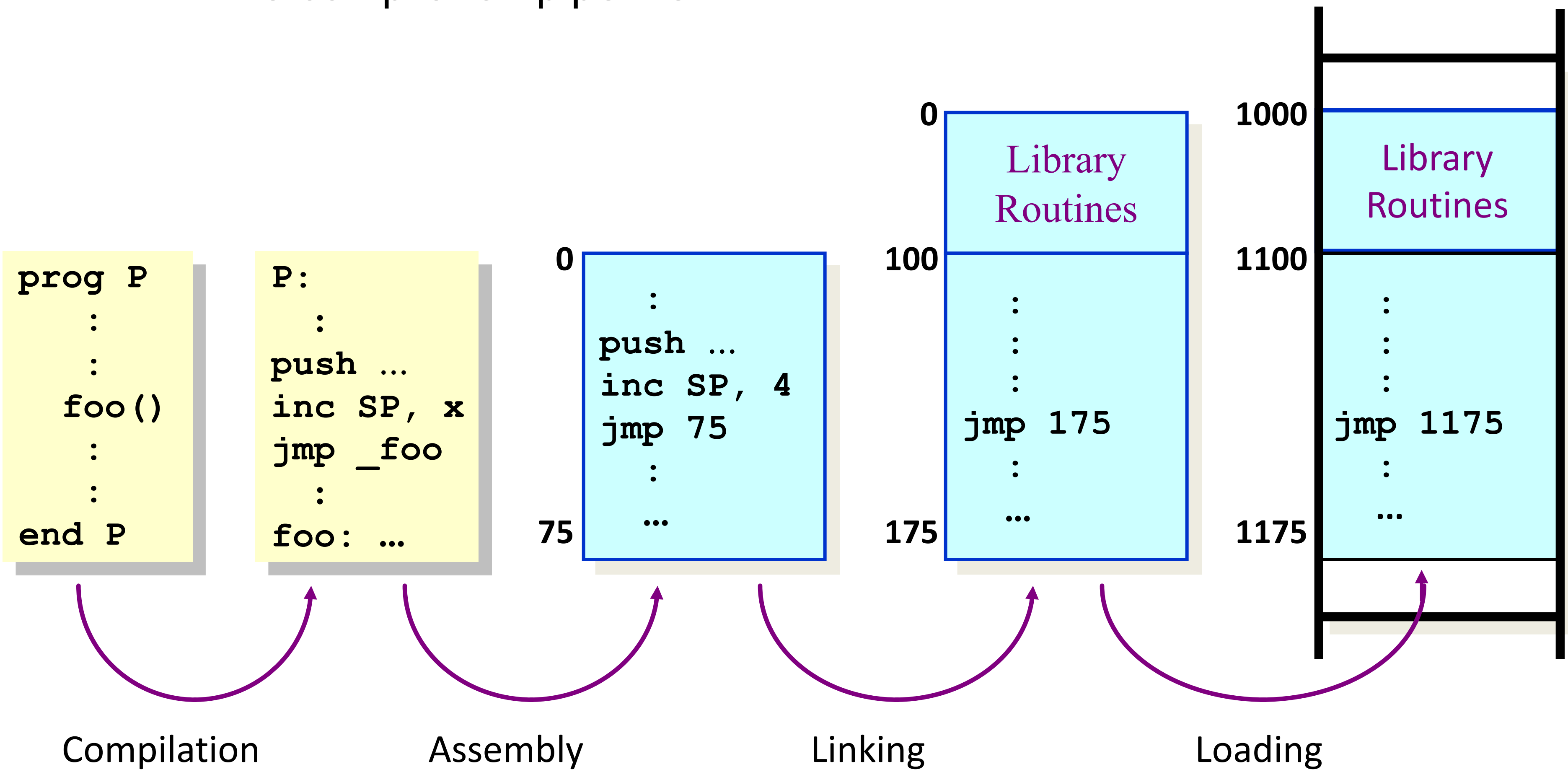
- We’ve talked some about processes
- This lecture: overall virtual memory abstractions
  - Key abstraction: Address space
- We will learn about the mechanics of virtual memory later

Basics

- Process includes a virtual address space
- An address space is composed of:
  - Memory-mapped files
    - Includes program binary
  - Anonymous pages: no file backing
    - When the process exits, their contents go away

Address Space Generation

- The compilation pipeline



Need addresses at compile time

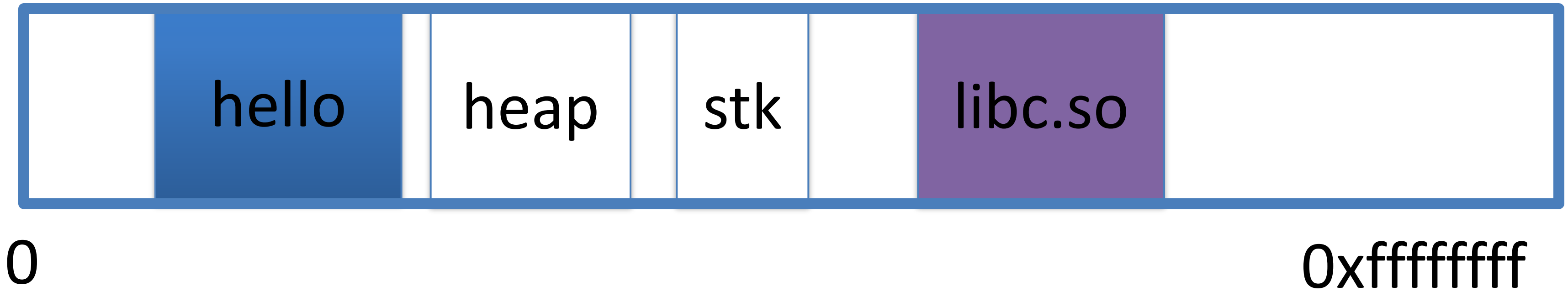
- You write code (even in assembly) using **symbolic** names
- Machine code ultimately needs to use **addresses**
  - Recall from 311/411 the arguments for jump, load, store…
- At compile time:
  - Compiler needs to generate machine code using ***run time addresses***
  - So, compiler must specify where data and code go
    - And/or generate code that can be “fixed up” at runtime

Address Space Layout

- Determined (mostly) by the application + compiler
  - Link directives can influence this
- OS 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

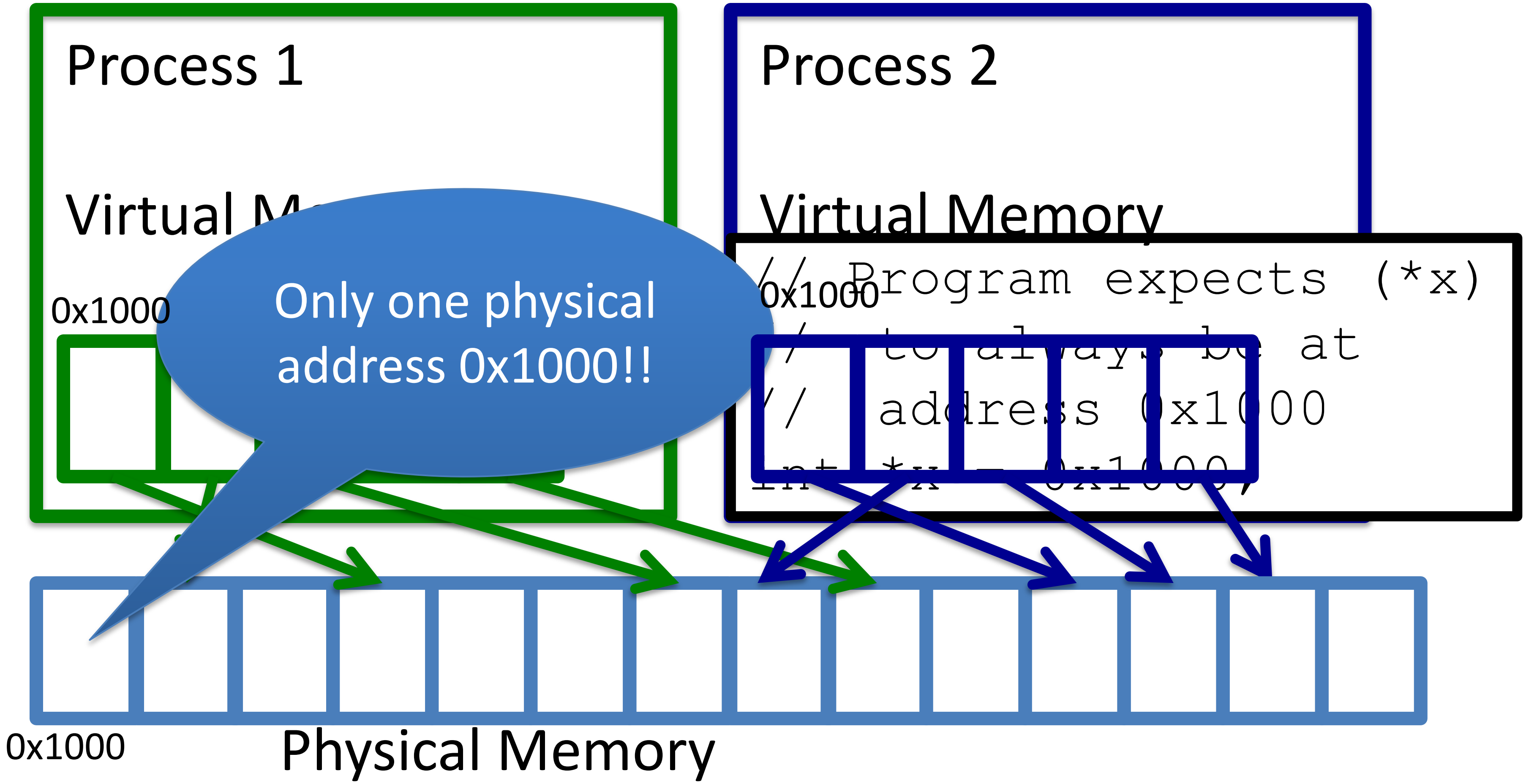
- You can see (part of) the requested memory layout of a program using ldd:

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

Many address spaces

- What if every program wants to map libc at the same address?
- No problem!
  - Every process has the abstraction of its own address space
  - Only one active at a given time (on a given core)
  - But many can exist in DRAM
- How does this work?

Memory Mapping



Two System Goals

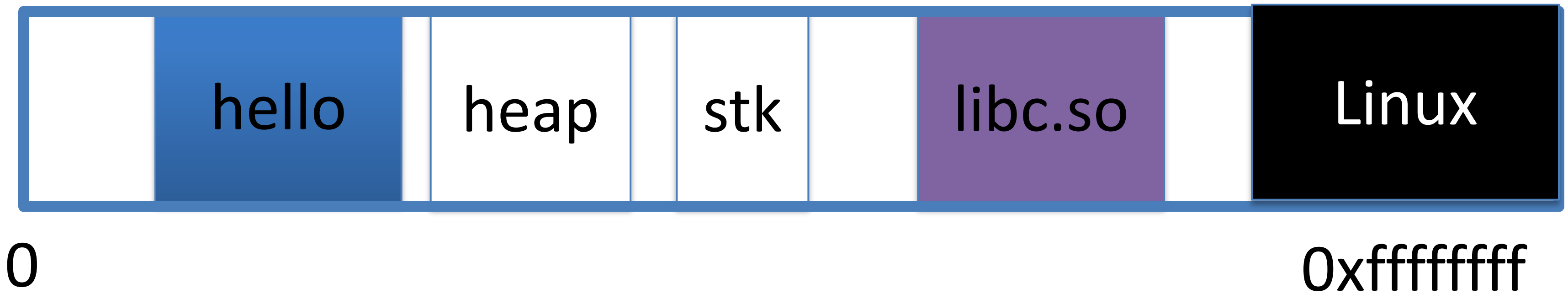
1) Provide an abstraction of contiguous, isolated virtual memory to a program
   - We will study the details of virtual memory later
2) Prevent illegal operations
   - Prevent access to other application
     - No way to address another application's memory
   - Detect failures early (e.g., segfault on address 0)

What about the kernel?

- Most OSes reserve part of the address space in every process by convention
  - Other ways to do this, nothing mandated by hardware

Example Redux

Virtual Address Space



- Kernel always at the “top” of the address space
- “Hello world” binary specifies most of the memory map
- Dynamically asks kernel for “anonymous” pages for its heap and stack

Why a fixed mapping?

- Makes the kernel-internal bookkeeping simpler
- Example: Remember how interrupt handlers are organized in a big table?
  - How does the table refer to these handlers?
    - By (virtual) address
    - Awfully nice when one table works in every process

Kernel protection?

- So, I protect programs from each other by running in different virtual address spaces
- But the kernel is in every virtual address space?

Decoupling CPU mode and Addr. Space

- CPU operates in 2 modes – user and supervisor
  - Applications execute in user mode
  - Kernel executes in supervisor mode
- Idea: restrict some addresses to supervisor mode
  - Although mapped, will fault if touched in user mode

Putting protection together

- Permissions on the memory map protect against programs:
  - Randomly reading secret data (like cached file contents)
  - Writing into kernel data structures
- The only way to access protected data is to trap into the kernel. How?
  - Interrupt (or syscall instruction)
- Interrupt table entries protect against jumping into unexpected code

Outline

- Basics of process address spaces
  - Kernel mapping
  - Protection
- How to dynamically change your address space?
- Overview of loading a program

Reminder: Two types of mappings

- Memory-mapped files
  - Includes program binary
- Anonymous pages: no file backing
  - When the process exits, their contents go away

Packing flags into a single integer

- Common Linux/C idiom
- Example: Access modes:

  PROT_READ == 2^0

  PROT_WRITE == 2^1

  PROT_EXEC == 2^2

- How to request read and write permission?
  - int flags = PROT_READ|PROT_WRITE; // == 1 + 2 == 3
  - Sets bits 0 and 1, but leaves other blank

Make sure you understand why flags are OR-ed

Linux APIs

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

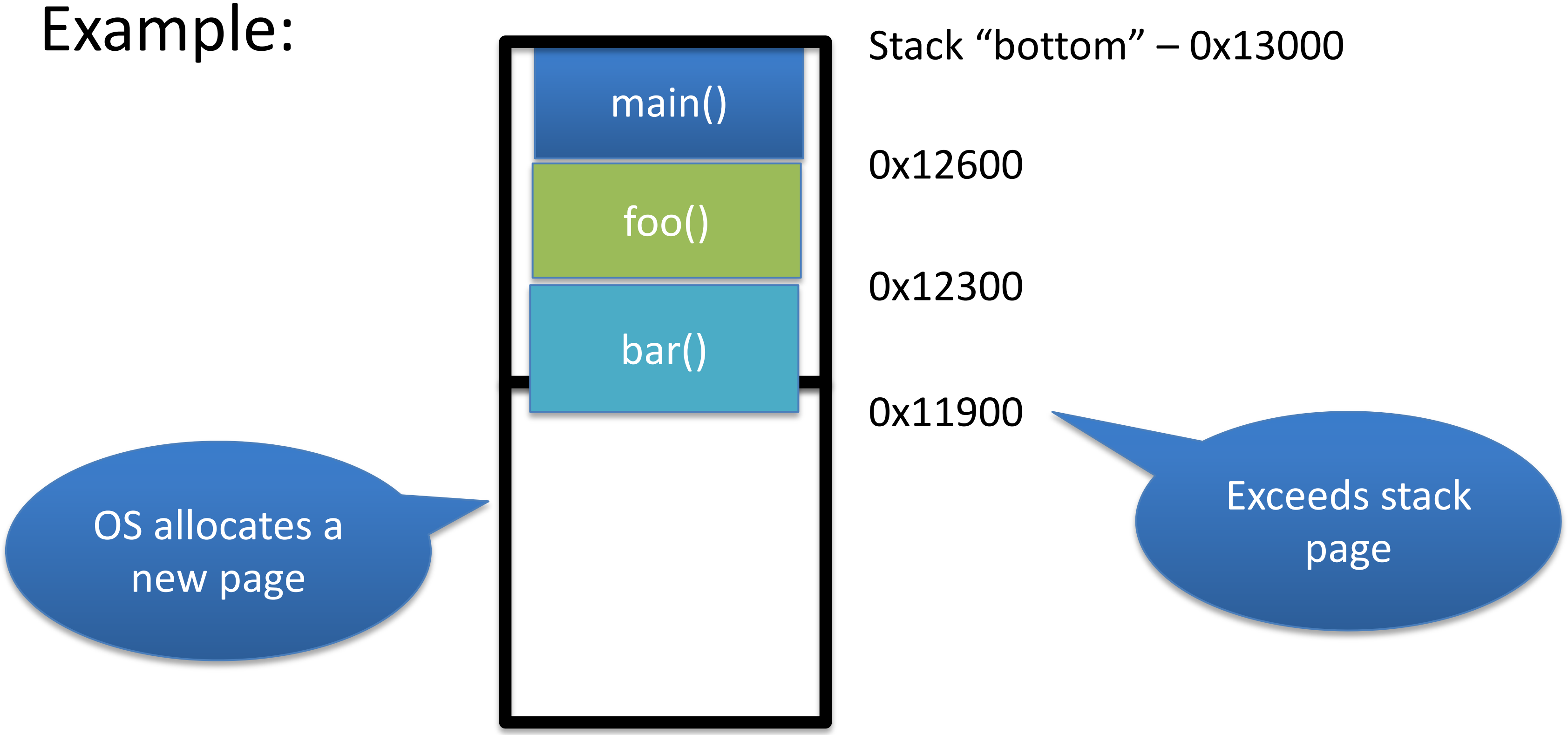
- How to create an anonymous mapping?
- What if you don't care where a memory region goes (as long as it doesn't clobber something else)?

Example:

- 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?

Idiosyncrasy 1: Stacks Grow Down

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



2 issues: How to expand, and why down (not up?)

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 "enough" virtual address space after "top" of stack

But how much is "enough"?

Feed 2 Birds with 1 Scone

- Unix has been around longer than paging
  - Data segment abstraction (we'll see more about segments later)
  - Unix solution:

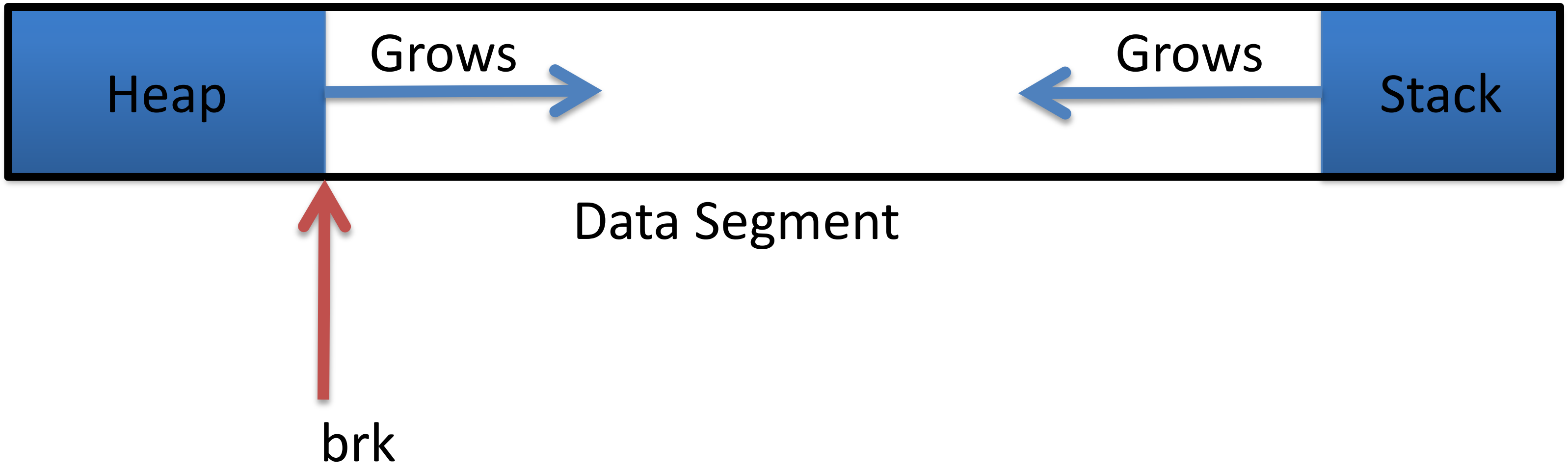
Data Segment

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

Just have to decide how much total data space

brk() system call

- Brk points to the end of the heap
- sys_brk() changes this pointer



Relationship to malloc()

- malloc, or any other memory allocator (e.g., new)
  - Library (usually libc) inside application
  - Gets large chunks of anonymous memory from the OS
    - Some use brk,
    - Many use mmap instead (better for parallel allocation)
  - Sub-divides into smaller pieces
  - Many malloc calls for each mmap call

Outline

- Basics of process address spaces
  - Kernel mapping
  - Protection
- How to dynamically change your address space?
- Overview of loading a program

Linux: ELF

- Executable and Linkable Format
- Standard on most Unix systems
- 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 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

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 linker

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

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()

Key point

- Most program loading work is done *by the loader in user space*
  - If you 'strace' any substantial program, there will be beaucoup **mmap** calls early on
  - Nice design point: the kernel only does very basic loading, ld.so does the rest
    - Minimizes risk of a bug in complicated ELF parsing corrupting the kernel

Other formats?

- The first two bytes of a file are a “magic number”
  - Kernel reads these and decides what loader to invoke
  - ‘#!’ says “I’m a script”, followed by the “loader” for that script
    - The loader itself may be an ELF binary
- Linux allows you to register new binary types (as long as you have a supported binary format that can load them

Recap

- Understand the idea of an address space
- Understand how a process sets up its address space, how it is dynamically changed
- Understand the basics of program loading