

Review

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

CSE 306: Operating Systems

Address Space Layout

Determined (mostly) by the application

Determined at compile time

Link directives can influence this

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

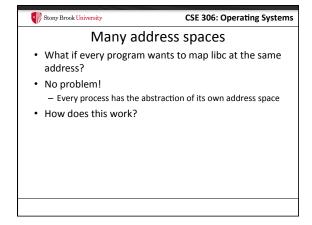
hello heap stk libc.so

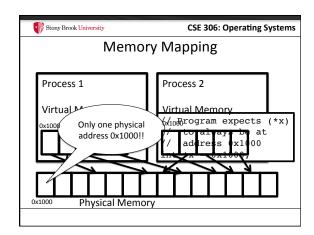
O Oxffffffff

"Hello world" binary specified load address

Also specifies where it wants libc

Dynamically asks kernel for "anonymous" pages for its heap and stack





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

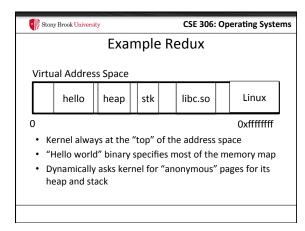
• No way to address another application's memory

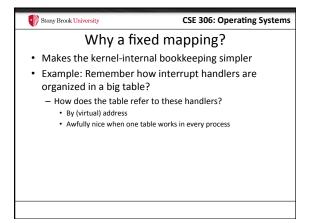
- Detect failures early (e.g., segfault on address 0)

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







CSE 306: Operating Systems

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?



CSE 306: Operating Systems

Protection rings

- Intel's hardware-level permission model
 - Ring 0 (supervisor mode) can issue any instruction
 - Ring 3 (user mode) no privileged instructions
 - Rings 1&2 mostly unused, some subset of privilege
- Note: this is not the same thing as superuser or administrator in the OS
 - Similar idea
- Key intuition: Memory mappings include a ring level and read only/read-write permission
 - Ring 3 mapping user + kernel, ring 0 only kernel



CSE 306: Operating Systems

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

Stony Brook University

CSE 306: Operating Systems

Outline

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



CSE 306: Operating Systems

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)?

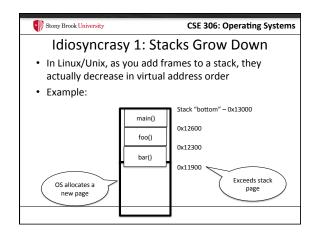
Stony Brook University

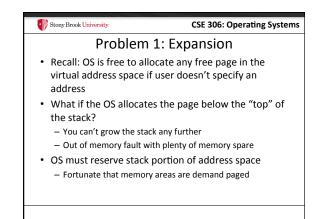
CSE 306: Operating Systems

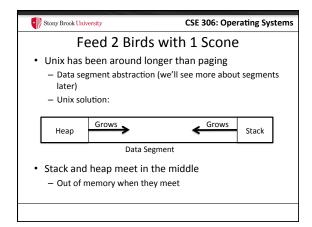
Example:

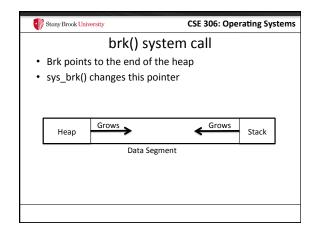
- Let's map a 1 page (4k) anonymous region for data, read-write at address 0x40000
- mmap(0x40000, 4096, PROT_READ|PROT_WRITE, MAP_ANONYMOUS, -1, 0);
 - Why wouldn't we want exec permission?

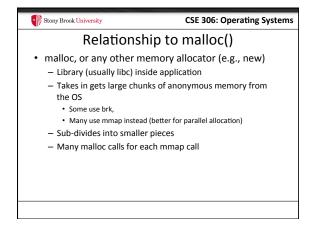
18

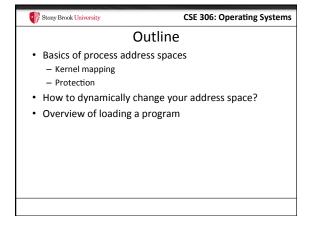












CSE 306: Operating Systems

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)

CSE 306: Operating Systems

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

Stony Brook University CSE 306: Operating Systems

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

Stony Brook University

How F

CSE 306: Operating Systems

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

27

CSE 306: Operating Systems

Static vs. Dynamic Linking

· Static Linking:

Stony Brook University

- 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

Stony Brook University

CSE 306: Operating Systems

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:

 - call *rax
- · Linker initializes the jump tables at runtime



CSE 306: Operating Systems

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()



CSE 306: Operating Systems

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, Id.so does the rest
 - Minimizes risk of a bug in complicated ELF parsing corrupting the
 leaned.



CSE 306: Operating Systems

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



CSE 306: Operating Systems

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