Process Address Spaces and Binary Formats

Don Porter

Background

• We’ve talked some about processes
• This lecture: discuss overall virtual memory organization
  – Key abstraction: Address space
• We will learn about the mechanics of virtual memory later

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

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

<table>
<thead>
<tr>
<th>hello</th>
<th>heap</th>
<th>stk</th>
<th>libc.so</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>0xffffffff</td>
</tr>
</tbody>
</table>

• “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
• 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’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

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?

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

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

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:

```plaintext
main()
foo()
bar()
```

Stack "bottom" = 0x13000
0x12600
0x12300
0x11900

OS allocates a new page

Exceeds stack page

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
  – Data segment abstraction (we’ll see more about segments later)
  – Unix solution:

```plaintext
Heap Grows Stack
Data Segment Grows
```

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

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
  – Takes in 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 runtime 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