Process Address Spaces and Binary Formats
Don Porter – CSE 506

Housekeeping
- Lab deadline extended to Wed night (9/14)
- Enrollment finalized – if you still want in, email me
- All students should have VMs at this point
  - Email Don if you don't have one
- TA office hours posted
- Private git repositories should be setup soon

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
Problem 1: How to represent?

+ 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 be to represent each page
  + 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

0

Process Address Space

0xffffffff

vma
/bin/ls

vma
anon
(data)

vma
libc.so

next

start

end

mm_struct
(process)
**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: $\log 100$ makes 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
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.
  - Example?
  - 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

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

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

Unix fork()

- Recall: this function creates and starts a copy of the process; identical except for the return value
- Example:
  ```c
  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
- Like demand paging, lazy is better. Why?
- Most processes immediately exec( ) a new binary without using any of these pages

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

Idiosyncrasy 1: Stacks Grow Down

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

```
main() Stack "bottom" – 0x13000
0x12600
0x12500
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
- Remember data segment abstraction?
- Unix solution:
  
  ![Diagram of heap and stack]

  - 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()
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/exec 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
  - If static elf, it loads the text, data, and bss sections, then drops into the program
  - If it is a dynamic elf, it instead loads the dynamic linker and drops into that
  - If something else, it loads the specified linker (dynamic elf is somewhat a special case of this)

Dynamic Linking

- Rather than start at `main()`, start at a setup routine
- As long as the setup routine is self-contained, it can:
  - 1) Walk the headers to identify needed libraries
  - 2) Issue `mmap()` calls to map in said libraries
  - 3) Do other bookkeeping
  - 4) Call `main()`

Position-Independent Code

- Quick definition anyone?
- How implemented?
  - Intuition: All jump targets and calls must be PC-relative
  - Or relative to the start of the section (i.e., dedicate a register to hold a base address that is added to a jump target)
- Libraries (shared objects) must be position-independent

How to call a .so function? (from a program)

- If the linker doesn't know where a function will end up, it creates a relocation
  - Index into the symbol table, location of call in code, type
  - Part of loading: linker marches through each relocation and overwrites the call target
  - But I thought .text was read-only?
  - Linker must modify page permissions, or kernel must set .text copy-on-write
How to call a .so function? (from another .so)

- Compiler creates a jump table for all external calls
- Called the plt; entries point to a global offset table (got) entry
- got stores location where a symbol was loaded in memory
- Lazily resolved (laziness is a virtue, remember?)
- Initially points to a fixup routine in the linker
- First time it is called, it figures out the relocation
  - Overwrites appropriate got entry

Windows PE (portable executable, or .exe)

- Import and Export Table (not just an import table)
- Setup routines called when:
  - The dll is loaded into a process
  - Unloaded
  - When a thread enters and exits
- DLLs are generally not position independent
  - Loading one at the non-preferred address requires code fixup (called rebasing)

Recap

- Goal is to convey intuitions about how programs are set up in Linux and Windows
- OS does preliminary executable parsing, maps in program and maybe dynamic linker
- Linker does needed fixup for the program to work

Advanced Topics

- How to handle other binary formats
- How to run 32-bit executables on a 64-bit OS?
### Non-native formats
- Most binary formats are identified in the first few bytes with a magic string
- Windows .exe files start with ascii characters “MZ”, for its designer [Mark Zbikowski](#)
- Interpreted languages (sh, perl, python) use “#!/” followed by the path to the interpreter
- Assuming the magic text can be found easily, Linux allows an interpreter to be associated with a format
- Like the ELF linker, this gets started upon exec

### Ex: Other Unix Flavors
- The APIs on most Unix programs are quite similar
- POSIX interfaces can just call Linux libc directly
- Others may require a shim, or small bits of code to emulate expected differences on the host platform

### Ex: WINE
- The same strategy is used to emulate Windows on Linux
- WINE includes reimplementations of Windows low-level libraries on Linux system calls
- And a “dynamic linker” that emulates the one in ntdll

### Linux32 on 64-bit Linux
- 64-bit x86 chips can run in 32-bit mode
- ELF can identify target architecture
- What does the OS need to do for 32-bit programs?
  - Set up 32-bit page tables
  - Keep old system call table around
  - Add shims for calling convention and other low-level ops
  - Have 32-bit binaries and libraries on disk
FatELF

- Experimental new feature (not in kernel yet)
- Rather than one .text, .bss, etc, have:
  - .text-x86, .text-x86-64, .text-arm, etc.
- Kernel/linker select appropriate sections for architecture
- Wastes some disk space, but no memory
- Saves human effort
- Same idea as Apple's Universal Binary format

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