CSE 451: Operating Systems
Winter 2009

Module 16
Linking, Loading and Process Startup

Mark Zbikowski

Gary Kimura

So you want to run a program..

- How was it created?
  - Someone wrote some C/C#/C++/etc
  - Compile/fix-errors/compile again
  - Get *object files*
- What’s in the .o or .obj files?
  - Code and data and *fixups*
  - Code and data are easy
  - Fixups describe relationships
    - Targets of jumps/calls
    - Data references
  - What do you do about references to other (extern) code/data?
    - Fixups too!

More of what’s in .o / .obj files

- Old style format (reflecting stream view of files)
  - Stream of records <tag><data> where <tag> was
    - DATA: <data section> was constant data
    - BSS: <data section> was just the size of the BSS reserved
    - CODE: just like DATA
    - FIXUP: applies to previous record, may list a name (external) or an offset into some other prior record and describe a width (8, 16, 32, 64) and an operation (ADD, IMM, SELF-REL)
- Modern format (take advantage of memory mapping)
  - Header on file describes *sections* suitable for mmap()

What’s a section?

- Section is a piece of contiguous memory
  - Named
  - Protected: read only, read/write, execute, read/execute, etc.
  - Location in file
  - Location in memory
- Some names are important
  - DATA
  - CODE
  - BSS
  - DEBUG
  - FIXUP

Putting .o/.obj files together

- The “linker”
  - take a collection of object files and produce an executable image
  - Gathers and appends like named/protected sections
  - Evaluates fixups and establishes addressing (linkages) between sections
  - Emits special sections
    - DEBUG
    - IMPORT
    - EXPORT
  - All into a file *with the same general format as .o/.obj files*
    - A few new sections
    - But it’s header says it’s executable
    - Called the *image file*

Executing the image file

- What does exec() or CreateProcess() do?
  - Easy stuff:
    - Allocate PCB
    - Create address space
  - Harder stuff
    - Create first thread
    - Copy handle environment from parent
  - The meat:
    - Opens image file
    - Memory maps header (reading section table)
    - For each section:
      - Memory map the appropriate portion of the file
      - Into the correct address space location
      - With correct memory protection

Is that all?

- Once upon a time, yes
  - All code was in one file
  - Included all special stuff for calling the OS
- Not nearly useful enough
  - What if system call #'s changed?
  - What about sharing common code between apps?
  - What about 3rd party code?
  - What about extensibility?

Dynamic libraries

- Goal: break down single images into multiple pieces
  - Independently distributable
  - Breakdown based on functionality / extensibility
- Implications on image format
  - Need a way to reference between image files
  - Add IMPORT and EXPORT sections
  - IMPORT lists all functions required by the image file (executable or library)
  - EXPORT lists all functions offered by the image file
- Big implications on process creation

Process Creation with libraries

- Easy/Harder stuff still the same
- Hardest stuff:
  - No longer loading just a single file, loading multiple modules
  - Walking each IMPORT table, finding references to modules not yet loaded and loading them
  - Big graph traversal
  - How are linkages established between modules?

Module Linkage

- Naïve approach is to use something similar to fixups
    - Modify the sections to establish linkage
    - Modifies the memory mapped pages
        - Don’t want to modify the original file
        - Copy-on-write
        - Bigger page file
        - More dirty pages in memory
- Work with compiler
    - Observe that inter-module references are always direct (never self-relative). Call or pointer reference
    - Keywords in language (or header files) that change direct calls into indirect calls and direct addressing into indirect addressing.

Efficient Linkage

- Foo(args) turns into (*import_Foo)(args)
- Gather all import_X addresses into a single section
  - Called IAT (import address table)
  - Usually only a single page in size, not inefficient to dirty
  - Still have to do some big work
- Can we do better?

Binding

- Floating modules
  - No known address
  - IAT required to handling differing locations based on other modules’ locations
- Bind modules to specific locations
  - Section table describes location, mapping is trivial
  - IAT can be pre-built with locations already in mind
  - Zero program-startup fixups
- What’s the issue?

Binding

- What address do you assign?
  - 32 bit address space *seems* large enough
  - XP has >1200 modules.
- What if there’s a *collision*?
  - New release of module grows in size (bug fixes, functionality)
  - Modules produced by two independent companies
  - Loader needs to be robust in the face of this
  - Choose another location
  - Fix up IAT (small number of pages)

A few cheats

- Compiler needs to generate self-relative instructions
  - Otherwise relocation of module would require fixups
  - Works well on x86…
  - Most of XP’s DLL’s can be broken into disjoint groups and addresses assigned to each

Vista cool feature

- “dynamic rebasing”
  - At install time, all modules are rebased randomly in memory
  - Just edit the IATs, still have speedy program start
  - What problem would this solve?
- Buffer overflow attacks
  - Operate by overflowing a stack buffer and overwriting a return address
  - Knowing where special code might be would allow attacker to hijack return to code in a module not directly referenced
  - Not if the module moves…

Windows CreateProcess

- Different from fork/exec.
  - Fork/exec are in kernel mode and embody the entire process creation experience
  - Windows Kernel has
    - NtCreateProcess – creates a new process address space. BUT NO THREAD
    - NtCreateThread – creates a new thread in a given process
    - NtSetThreadInformation – sets execution context for thread (notably stack and PC)

Windows CreateProcess

- CreateProcess is user code in kernel32 module
  - Creates process (NtCreateProcess)
  - Maps in kernel call DLL (ntdll)
  - Maps in image (but no libraries)
  - Creates initial thread
  - Sets thread to initialization routine in ntdll (LdrpInitializeProcess)
  - Go!
- LdrpInitializeProcess does all the memory mapping work
  - Executing in the new image’s context
  - Walking module lists is just memory access
  - Makes NtCreateSection calls

Why not do what unix did?

- Extensibility
  - Differing loader policies (OS/2, DOS)
  - New loader implementations
  - Smaller kernel
- Simpler loader code