Introduction to Computer Systems
15-213/18-243, fall 2009
9th Lecture, Sep 29th
Instructors:
Roger B. Dannenberg and Greg Ganger

Today
- Address Space at Run Time
- Linking

First Exam

IA32 Linux Memory Layout

- Stack
  - Runtime stack (8MB limit)
- Heap
  - Dynamically allocated storage
  - When call malloc(), calloc(), new() 
- Data
  - Statically allocated data
  - E.g., arrays & strings declared in code
- Text
  - Executable machine instructions
  - Read-only

Memory Allocation Example

IA32 Example Addresses

address range ~2^32

Where does everything go?
x86-64 Example Addresses  
address range “2^37”

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>$rsp</td>
<td>0x7fff,7576,0050</td>
</tr>
<tr>
<td>p1</td>
<td>0x2b7f,e3f7,1010</td>
</tr>
<tr>
<td>p3</td>
<td>0x0000,1150,1120</td>
</tr>
<tr>
<td>p4</td>
<td>0x0000,1150,1010</td>
</tr>
<tr>
<td>beyond</td>
<td>0x0000,1150,0a28</td>
</tr>
<tr>
<td>big_array</td>
<td>0x0000,1050,0a20</td>
</tr>
<tr>
<td>4p2</td>
<td>0x0000,1050,0a00</td>
</tr>
<tr>
<td>huge_array</td>
<td>0x0000,0050,0a00</td>
</tr>
<tr>
<td>main()</td>
<td>0x0000,0040,0510</td>
</tr>
<tr>
<td>useless()</td>
<td>0x0000,0040,0500</td>
</tr>
<tr>
<td>malloc()</td>
<td>0x0038,6ae6,a170</td>
</tr>
</tbody>
</table>

malloc() is dynamically linked
address determined at runtime

Example C Program

```c
#include <stdio.h>

int buf[] = {1, 2};
int main()
{
    swap();
    return 0;
}

void swap()
{
    int temp;
    if (buf[1] != buf[0])
    {
        temp = buf[0];
        buf[0] = buf[1];
        buf[1] = temp;
    }
}
```

Static Linking

Programs are translated and linked using a compiler driver:

```
unix> gcc -O2 -g -o p main.c swap.c
unix> ./p
```

Why Linkers?

- Modularity!
  
  Program can be written as a collection of smaller source files,
  rather than one monolithic mass.

  - Can build libraries of common functions (more on this later)
    
      e.g., Math library, standard C library

- Efficiency!
  
  - Time: Separate Compilation
    
      - Change one source file, compile, and then relink.
      
        No need to recompile other source files.

  - Space: Libraries
    
      Common functions can be aggregated into a single file...
      Yet executable files and running memory images contain only code
      for the functions they actually use.

What Do Linkers Do?

- Step 1: Symbol resolution
  
    Programs define and reference symbols (variables and functions):
    
    - void swap() i.i. /* define symbol swap */
    - swap(); /* reference symbol swap */
    - int *x = 1; /* define x, reference x */

    Symbol definitions are stored (by compiler) in symbol table.
    
    - Symbol table is an array of structs
      
      - Each entry includes name, type, size, and location of symbol.

    Linker associates each symbol reference with exactly one symbol
    definition.
What Do Linkers Do? (cont.)

- Step 2: Relocation
  - Merges separate code and data sections into single sections
  - Relocates symbols from their relative locations in the .o files to their final absolute memory locations in the executable.
  - Updates all references to these symbols to reflect their new positions.

Three Kinds of Object Files (Modules)

- Relocatable object file (.o file)
  - Contains code and data in a form that can be combined with other relocatable object files to form executable object file.
  - Each .o file is produced from exactly one source (.c) file

- Executable object file
  - Contains code and data in a form that can be copied directly into memory and then executed.

- Shared object file (.so file)
  - Special type of relocatable object file that can be loaded into memory and linked dynamically, at either load time or run-time.
  - Called Dynamic Link Libraries (DLLs) by Windows

Executable and Linkable Format (ELF)

- Standard binary format for object files
- Originally proposed by AT&T System V Unix
  - Later adopted by BSD Unix variants and Linux
- One unified format for
  - Relocatable object files (.o),
  - Executable object files
  - Shared object files (.so)
- Generic name: ELF binaries

ELF Object File Format

- Elf header
  - Word size, byte ordering, file type (.o, exec, .so), machine type, etc.
- Segment header table
  - For executables: virtual address, segment size, alignments
- .text section
  - Code
- .rodata section
  - Read only data: jump tables, ...
- .data section
  - Initialized global variables
- .bss section
  - Uninitialized global variables
  - "Better Save Space"
- .symtab section
  - Symbol table
- .strtab section
  - Procedure and static variable names
- .string section
  - Section names and locations
- .rel.text section
  - Relocation info for .text section
  - Addresses of instructions that will need to be modified in the executable
  - Instructions for modifying.
- .rel.data section
  - Relocation info for .data section
  - Addresses of pointer data that will need to be modified in the merged executable
- .debug section
  - Info for symbolic debugging (see -g)
- Section header table
  - Offsets and sizes of each section

ELF Object File Format (cont.)

- Symtab section
  - Symbol table
  - Procedure and static variable names
  - Section names and locations
- .rel.text section
  - Relocation info for .text section
  - Addresses of instructions that will need to be modified in the executable
  - Instructions for modifying.
- .rel.data section
  - Relocation info for .data section
  - Addresses of pointer data that will need to be modified in the merged executable
- .debug section
  - Info for symbolic debugging (see -g)
- Section header table
  - Offsets and sizes of each section

Linker Symbols

- Global symbols
  - Symbols defined by module m that can be referenced by other modules.
  - E.g.: non-static C functions and non-static global variables.
- External symbols
  - Global symbols that are referenced by module m but defined by some other module.
- Local symbols
  - Symbols that are defined and referenced exclusively by module m.
  - E.g.: C functions and variables defined with the static attribute.
  - Local linker symbols are not local program variables
Executable After Relocation (.data)

Disassembly of section .data:
08049454 <buf>:
0804945c <buf+0>:

Strong and Weak Symbols

- Program symbols are either strong or weak
  - Strong: procedures and initialized globals
  - Weak: uninitialized globals

Strong
Weak

\[
\begin{array}{ll}
\text{p1.c} & \text{p2.c} \\
\text{int foo=5;} & \text{int foo;} \\
p1() & p2() \\
\end{array}
\]

Linker’s Symbol Rules

- Rule 1: Multiple strong symbols are not allowed
  - Each item can be defined only once
  - Otherwise: Linker error

- Rule 2: Given a strong symbol and multiple weak symbol, choose the strong symbol
  - References to the weak symbol resolve to the strong symbol

- Rule 3: If there are multiple weak symbols, pick an arbitrary one
  - Can override this with gcc \texttt{-fno-common}

Linker Puzzles

- \texttt{Link time error: two strong symbols (p1)}
- \texttt{References to x will refer to the same uninitialized int. Is this what you really want?}
- \texttt{Writes to x in p2 might overwrite y! Evil!}
- \texttt{Writes to x in p2 will overwrite y! Nasty!}
- \texttt{References to x will refer to the same initialized variable.}

Global Variables

- Avoid if you can

- Otherwise
  - Use static if you can
  - Initialize if you define a global variable
  - Use extern if you use external global variable

Packaging Commonly Used Functions

- How to package functions commonly used by programmers?
  - Math, I/O, memory management, string manipulation, etc.

- Awkward, given the linker framework so far:
  - Option 1: Put all functions into a single source file
    - Programmers link big object file into their programs
    - Space and time inefficient
  - Option 2: Put each function in a separate source file
    - Programmers explicitly link appropriate binaries into their programs
    - More efficient, but burdensome on the programmer
Solution: Static Libraries

- **Static libraries (.a archive files)***
  - Concatenate related relocatable object files into a single file with an index (called an archive).
  - Enhance linker so that it tries to resolve unresolved external references by looking for the symbols in one or more archives.
  - If an archive member file resolves reference, link into executable.

Creating Static Libraries

```
unx> ar rs libc.a \n  atoi.o printf.o random.o
```

- Archiver allows incremental updates
- Recompile function that changes and replace .o file in archive.

Commonly Used Libraries

**libc.a** (the C standard library)
- 8 MB archive of 900 object files.
- I/O, memory allocation, signal handling, string handling, data and time, random numbers, integer math

**libm.a** (the C math library)
- 1 MB archive of 226 object files.
- Floating point math (sin, cos, tan, log, exp, sqrt, ...)

```
% ar -t /usr/lib/libc.a | sort ... 
fork.o ...
fprintf.o fpu_control.o fputc.o freopen.o fseek.o ...
% ar -t /usr/lib/libm.a | sort ... 
e_acos.o e_acosf.o e_acosh.o e_acoshf.o e_acoshl.o e_acosl.o e_asin.o e_asinf.o e_asinl.o ...
```

Linking with Static Libraries

```
addvec.o multvec.o
```

- Translators
  - **cpp**, **cc1**, **as**

```
main2.c main2.o
```

```
libc.a
```

- Linker (**ld**)
  - printf.o and any other modules called by printf.o

```
libvector.a addvec.o
```

using **ar**

Using Static Libraries

- **Linker’s algorithm for resolving external references:**
  - Scan .o files and .a files in the command line order.
  - During the scan, keep a list of the current unresolved references.
  - As each new .o or .a file, obj, is encountered, try to resolve each unresolved reference in the list against the symbols defined in obj.
  - If any entries in the unresolved list at end of scan, then error.

```
unix> gcc -L . libtest.o -lm
unix> gcc -L . -lm libtest.o libtest.o: In function ‘main’:
libtest.o: undefined reference to ‘libfun’
```

Loading Executable Object Files

```
Kernel virtual memory
User stack
(allocated at runtime)
Memory-mapped region for shared libraries
```

```
run-time heap
(allocated by malloc)
Read/write segment
/data, /bss
Read-only segment
/init, /text, /rodata
```

```
Memory invisible to user code
Heap (stack pointer)
brk
```

```
Loaded from the executable file
```

- Executable Object File
  - ELF header
  - Program header table (required for relocatables)
    - init section
    - .text section
    - .rodata section
    - .data section
    - .bss section
    - .symtab
    - .strtab
    - .debug
    - .note
    - .rela
    - .shstrtab
    - .shnum
    - .shoff
    - .shsize
    - .shnum
    - .shstrsize
  - Section header (required for relocatables)
Shared Libraries

- Static libraries have the following disadvantages:
  - Duplication in the stored executables (every function needs stdlibc)
  - Duplication in the running executables
  - Minor bug fixes of system libraries require each application to explicitly relink

- Modern Solution: Shared Libraries
  - Object files that contain code and data that are loaded and linked into an application dynamically, at either load-time or run-time
  - Also called: dynamic link libraries, DLLs, .so files

Dynamic Linking at Load-time

Dynamic Linking at Runtime

Case Study: Library Interpositioning

Library interpositioning is a powerful linking technique that allows programmers to intercept calls to arbitrary functions. Interpositioning can occur at:

- Compile time
- Link time
- When the source code is compiled
- When the relocatable object files are linked to form an executable object file
- Load/Run time
- When an executable object file is loaded into memory, dynamically linked, and then executed.
Some Interpositioning Applications

Security
- Confinement (sandboxing)
- Interpose calls to libc functions.
- Behind the scenes encryption
  - Automatically encrypt otherwise unencrypted network connections.

Monitoring and Profiling
- Count number of calls to functions
- Characterize call sites and arguments to functions
- Malloc tracing
  - Detecting memory leaks
  - Generating malloc traces

Example: malloc() Statistics

Count how much memory is allocated by a function

```c
#include <stdio.h>

void *malloc(size_t size) {  
  static void *(*fp)(size_t) = 0;
  void *mp;
  char *errorstr;
  /* Get a pointer to the real malloc() */
  if (!fp) {  
    fp = dlsym(RTLD_NEXT, "malloc");
    if ((errorstr = dlerror()) != NULL) {
      fprintf(stderr, "%s(): %s
", fname, errorstr);
      exit(1);
    }
  }
  /* Call the real malloc function */
  mp = fp(size);
  mem_used += size;
  return mp;
}
```

Summary

- ELF files contain
  - Object files
  - Libraries
  - Executables
- Linking
- Loading
- Dynamic Linking
- Details:
  - How are globals, externals, static symbols handled?
  - How are names searched and resolved by linkers?
  - How can you interpose your own library implementation?