Introduction to Computer Systems
15-213/18-243, fall 2009
9th Lecture, Sep 29th

Instructors:
Roger B. Dannenberg and Greg Ganger
First Exam

Exam1 Grade as Percent
Today

- Address Space at Run Time
- Linking
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

```c
char big_array[1<<24]; /* 16 MB */
char huge_array[1<<28]; /* 256 MB */

int beyond;
char *p1, *p2, *p3, *p4;

int useless() { return 0; }

int main()
{
    p1 = malloc(1 <<28);  /* 256 MB */
p2 = malloc(1 << 8);  /* 256 B */
p3 = malloc(1 <<28);  /* 256 MB */
p4 = malloc(1 << 8);  /* 256 B */
/* Some print statements ... */
}
```

Where does everything go?
IA32 Example Addresses

address range \( \sim 2^{32} \)

$esp \quad \text{0xffffbcd0}
p3 \quad \text{0x65586008}
p1 \quad \text{0x55585008}
p4 \quad \text{0x1904a110}
p2 \quad \text{0x1904a008}
&p2 \quad \text{0x18049760}
beyond \quad \text{0x08049744}
big_array \quad \text{0x18049780}
huge_array \quad \text{0x08049760}
main() \quad \text{0x080483c6}
useless() \quad \text{0x08049744}
final malloc() \quad \text{0x006be166}

malloc() is dynamically linked
address determined at runtime
x86-64 Example Addresses

address range $\sim 2^{47}$

- $\texttt{rsp}$: 0x7fff,7576,0050
- $\texttt{p1}$: 0x2b7f,e3f7,1010
- $\texttt{p3}$: 0x2b7f,f3f7,2010
- $\texttt{p4}$: 0x0000,1150,1120
- $\texttt{p2}$: 0x0000,1150,1010
- $\texttt{beyond}$: 0x0000,1150,0a28
- $\texttt{big\_array}$: 0x0000,1050,0a20
- $\&\texttt{p2}$: 0x0000,1050,0a00
- $\texttt{huge\_array}$: 0x0000,0050,0a00
- $\texttt{main()}$: 0x0000,0040,0510
- $\texttt{useless()}$: 0x0000,0040,0500
- $\texttt{final\_malloc()}$: 0x0038,6ae6,a170

$\texttt{malloc()}$ is dynamically linked
address determined at runtime

Diagram showing memory allocation with Stack, Heap, Data, and Text sections.
Example C Program

main.c

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

swap.c

```c
extern int buf[];
static int *bufp0 = &buf[0];
static int *bufp1;

void swap()
{
    int temp;

    bufp1 = &buf[1];
    temp = *bufp0;
    *bufp0 = *bufp1;
    *bufp1 = temp;
}
```
Static Linking

- Programs are translated and linked using a compiler driver:
  
  unix> gcc -O2 -g -o p main.c swap.c
  unix> ./p

Source files

Separately compiled relocatable object files

Fully linked executable object file (contains code and data for all functions defined in main.c and swap.c)
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
Why Linkers? 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() { ... } /* define symbol swap */`
    - `swap(); /* reference symbol swap */`
    - `int *xp = &x; /* define xp, 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
  - “Block Started by Symbol”
  - “Better Save Space”
  - Has section header but occupies no space

<table>
<thead>
<tr>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELF header</td>
</tr>
<tr>
<td>Segment header table</td>
</tr>
<tr>
<td>(required for executables)</td>
</tr>
<tr>
<td>. text section</td>
</tr>
<tr>
<td>. rodata section</td>
</tr>
<tr>
<td>. data section</td>
</tr>
<tr>
<td>. bss section</td>
</tr>
<tr>
<td>. symtab section</td>
</tr>
<tr>
<td>. rel.txt section</td>
</tr>
<tr>
<td>. rel.data section</td>
</tr>
<tr>
<td>. debug section</td>
</tr>
<tr>
<td>Section header table</td>
</tr>
</tbody>
</table>
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 (`gcc -g`)

- **Section header table**
  - Offsets and sizes of each section

```
+-------------------+-------------------+
| ELF header        | Section header table |
| Segment header table (required for executables) | |
| .text section     | .symtab section   |
| .rodata section   | .rel.txt section  |
| .data section     | .rel.data section |
| .bss section      | .debug section    |
| Section header table |                 |
```
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**
Resolving Symbols

```c
int buf[2] = {1, 2};

int main()
{
  swap();
  return 0;
}

void swap()
{
  int temp;
  bufp1 = &buf[1];
  temp = *bufp0;
  *bufp0 = *bufp1;
  *bufp1 = temp;
}
```

```c
extern int buf[];

static int *bufp0 = &buf[0];
static int *bufp1;

void swap()
{
  int temp;
  bufp1 = &buf[1];
  temp = *bufp0;
  *bufp0 = *bufp1;
  *bufp1 = temp;
}
```
Relocating Code and Data

Relocatable Object Files

<table>
<thead>
<tr>
<th>System code</th>
<th>.text</th>
</tr>
</thead>
<tbody>
<tr>
<td>System data</td>
<td>.data</td>
</tr>
</tbody>
</table>

main.o

main() .text
int buf[2]={1,2} .data

swap.o

swap() .text
int *bufp0=&buf[0] .data
int *bufp1 .bss

Executable Object File

Headers

System code

main() .text

swap() .text

More system code

System data
int buf[2]={1,2} .data
int *bufp0=&buf[0] .data
Uninitialized data

Symtab
Debug
Relocation Info (main)

int buf[2] = {1,2};

int main()
{
    swap();
    return 0;
}

Source: objdump
Relocation Info (swap, .text)

swap.c

extern int buf[];
static int *bufp0 = &buf[0];
static int *bufp1;

void swap()
{
    int temp;
    bufp1 = &buf[1];
    temp = *bufp0;
    *bufp0 = *bufp1;
    *bufp1 = temp;
}

swap.o

Disassembly of section .text:

00000000 <swap>:
  0: 55  push   %ebp
  1: 8b 15 00 00 00 00  mov  0x0,%edx
  7: a1 0 00 00 00 00  mov   0x4,%eax
  c: 89 e5  mov   %esp,%ebp
  e: c7 05 00 00 00 00 04  movl $0x4,0x0
  15: 00 00 00 10: R_386_32 bufp0
  14: R_386_32 buf
  18: 89 ec  mov   %ebp,%esp
  1a: 8b 0a  mov  (%edx),%ecx
  1c: 89 02  mov   %eax,(%edx)
  1e: a1 00 00 00 00  mov  0x0,%eax
  1f: R_386_32 bufp1
  23: 89 08  mov   %ecx,(%eax)
  25: 5d  pop   %ebp
  26: c3  ret
Relocation Info (swap, .data)

```c
extern int buf[];
static int *bufp0 = &buf[0];
static int *bufp1;

void swap()
{
    int temp;

    bufp1 = &buf[1];
    temp = *bufp0;
    *bufp0 = *bufp1;
    *bufp1 = temp;
}
```

Disassembly of section .data:

00000000 <bufp0>:

0: 00 00 00 00

0: R_386_32 buf
## Executable After Relocation (.text)

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>OpCode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>80483b4</td>
<td>&lt;main&gt;:</td>
<td>55</td>
<td>push %ebp</td>
</tr>
<tr>
<td>80483b5</td>
<td></td>
<td>89 e5</td>
<td>mov %esp,%ebp</td>
</tr>
<tr>
<td>80483b7</td>
<td></td>
<td>83 ec 08</td>
<td>sub $0x8,%esp</td>
</tr>
<tr>
<td>80483ba</td>
<td></td>
<td>e8 09 00 00 00</td>
<td>call 80483c8 &lt;swap&gt;</td>
</tr>
<tr>
<td>80483bf</td>
<td></td>
<td>31 c0</td>
<td>xor %eax,%eax</td>
</tr>
<tr>
<td>80483c1</td>
<td></td>
<td>89 ec</td>
<td>mov %ebp,%esp</td>
</tr>
<tr>
<td>80483c3</td>
<td></td>
<td>5d</td>
<td>pop %ebp</td>
</tr>
<tr>
<td>80483c4</td>
<td></td>
<td>c3</td>
<td>ret</td>
</tr>
<tr>
<td>80483c8</td>
<td>&lt;swap&gt;:</td>
<td>55</td>
<td>push %ebp</td>
</tr>
<tr>
<td>80483c9</td>
<td></td>
<td>8b 15 5c 94 04 08</td>
<td>mov 0x804945c,%edx</td>
</tr>
<tr>
<td>80483cf</td>
<td></td>
<td>a1 58 94 04 08</td>
<td>mov 0x8049458,%eax</td>
</tr>
<tr>
<td>80483d4</td>
<td></td>
<td>89 e5</td>
<td>mov %esp,%ebp</td>
</tr>
<tr>
<td>80483d6</td>
<td></td>
<td>c7 05 48 95 04 08 58</td>
<td>movl $0x8049458,0x8049548</td>
</tr>
<tr>
<td>80483dd</td>
<td></td>
<td>94 04 08</td>
<td></td>
</tr>
<tr>
<td>80483e0</td>
<td></td>
<td>89 ec</td>
<td>mov %ebp,%esp</td>
</tr>
<tr>
<td>80483e2</td>
<td></td>
<td>8b 0a</td>
<td>mov (%edx),%ecx</td>
</tr>
<tr>
<td>80483e4</td>
<td></td>
<td>89 02</td>
<td>mov %eax, (%edx)</td>
</tr>
<tr>
<td>80483e6</td>
<td></td>
<td>a1 48 95 04 08</td>
<td>mov 0x8049548,%eax</td>
</tr>
<tr>
<td>80483eb</td>
<td></td>
<td>89 08</td>
<td>mov %ecx, (%eax)</td>
</tr>
<tr>
<td>80483ed</td>
<td></td>
<td>5d</td>
<td>pop %ebp</td>
</tr>
<tr>
<td>80483ee</td>
<td></td>
<td>c3</td>
<td>ret</td>
</tr>
</tbody>
</table>
Executable After Relocation (.data)

Disassembly of section .data:

08049454 <buf>:
  8049454: 01 00 00 00 02 00 00 00

0804945c <bufp0>:
  804945c: 54 94 04 08
Strong and Weak Symbols

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

```
int foo=5;
p1()
}
```

```
int foo;
p2()
}
```
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 -fno-common`
Linker Puzzles

```
int x;
p1() {}  
```

```
int x;
p1() {}  
```

```
int x=7;
int y=5;
p1() {}  
```

```
int x=7;
p1() {}  
```

```
double x;
p2() {}  
```

```
double x;
p2() {}  
```

```
int x;
p2() {}  
```

```
int x;
p2() {}  
```

Link time error: two strong symbols (p1)

References to x will refer to the same uninitialized int. Is this what you really want?

Writes to x in p2 might overwrite y!

Evil!

Writes to x in p2 will overwrite y!

Nasty!

References to x will refer to the same initialized variable.

Nightmare scenario: two identical weak structs, compiled by different compilers with different alignment rules.
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

- 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, ...)

```bash
% ar -t /usr/lib/libc.a | sort
... 
fork.o
...
fprintf.o
fpu_control.o
fputc.o
freopen.o
fscanf.o
fseek.o
fstab.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
e_asinl.o
...
```
Linking with Static Libraries

Translators (cpp, cc1, as)

main2.c, vector.h

Archiver (ar)

libvector.a, libc.a

Addvec.o, multvec.o

Linker (ld)

main2.o, printf.o

p2

Fully linked executable object file

Libraries

Relocatable object files

Printf.o and any other modules called by printf.o
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.

- **Problem:**
  - Command line order matters!
  - Moral: put libraries at the end of the command line.

```bash
unix> gcc -L. libtest.o -lmine
unix> gcc -L. -lmine libtest.o
libtest.o: In function `main':
libtest.o(.text+0x4): undefined reference to `libfun'
```
# Loading Executable Object Files

<table>
<thead>
<tr>
<th>Executable Object File</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELF header</td>
<td>0</td>
</tr>
<tr>
<td>Program header table (required for executables)</td>
<td>0xc0000000</td>
</tr>
<tr>
<td>.init section</td>
<td>0x40000000</td>
</tr>
<tr>
<td>.text section</td>
<td>0x80480000</td>
</tr>
<tr>
<td>.rodata section</td>
<td></td>
</tr>
<tr>
<td>.data section</td>
<td></td>
</tr>
<tr>
<td>.bss section</td>
<td></td>
</tr>
<tr>
<td>.symtab</td>
<td></td>
</tr>
<tr>
<td>.debug</td>
<td></td>
</tr>
<tr>
<td>.line</td>
<td></td>
</tr>
<tr>
<td>.strtab</td>
<td></td>
</tr>
<tr>
<td>Section header table (required for relocatables)</td>
<td></td>
</tr>
</tbody>
</table>

## Kernel virtual memory
- User stack (created at runtime)
- Memory-mapped region for shared libraries
- Run-time heap (created by malloc)
- Read/write segment (.data, .bss)
- Read-only segment (.init, .text, .rodata)
- Unused

- Memory invisible to user code
- `%esp` (stack pointer)
- `brk`

- Loaded from the executable file
Shared Libraries

- **Static libraries have the following disadvantages:**
  - Duplication in the stored executables (every function need std libc)
  - 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
Shared Libraries (cont.)

- Dynamic linking can occur when executable is first loaded and run (load-time linking).
  - Common case for Linux, handled automatically by the dynamic linker (`ld-linux.so`).
  - Standard C library (`libc.so`) usually dynamically linked.

- Dynamic linking can also occur after program has begun (run-time linking).
  - In Unix, this is done by calls to the `dlopen()` interface.
    - High-performance web servers.
    - Runtime library interpositioning

- Shared library routines can be shared by multiple processes.
  - More on this when we learn about virtual memory
Dynamic Linking at Load-time

main2.c  vector.h

Translators
(cpp, ccl, as)

main2.o

Linker (ld)

p2

Loader
(execve)

Dynamic linker (ld-linux.so)

unix> gcc -shared -o libvector.so \ addvec.c multvec.c

libc.so

libvector.so

Relocation and symbol table info

Relocatable object file

Partially linked executable object file

Fully linked executable in memory

Code and data
#include <stdio.h>
#include <dlfcn.h>

int x[2] = {1, 2};
int y[2] = {3, 4};
int z[2];

int main()
{
  void *handle;
  void (*addvec)(int *, int *, int *, int);
  char *error;

  /* dynamically load the shared lib that contains addvec() */
  handle = dlopen("./libvector.so", RTLD_LAZY);
  if (!handle) {
    fprintf(stderr, "%s\n", dlerror());
    exit(1);
  }
}
/* get a pointer to the addvec() function we just loaded */
addvec = dlsym(handle, "addvec");
if ((error = dlerror()) != NULL) {
    fprintf(stderr, "%s\n", error);
    exit(1);
}

/* Now we can call addvec() it just like any other function */
addvec(x, y, z, 2);
printf("z = [%d %d]\n", z[0], z[1]);

/* unload the shared library */
if (dlclose(handle) < 0) {
    fprintf(stderr, "%s\n", dlerror());
    exit(1);
}
return 0;
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
  - When the source code is compiled
- link time
  - 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
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\n", 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?