CS 449 – Executables and Linking

Example C program

main.c

```
void swap();

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

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

swap.c

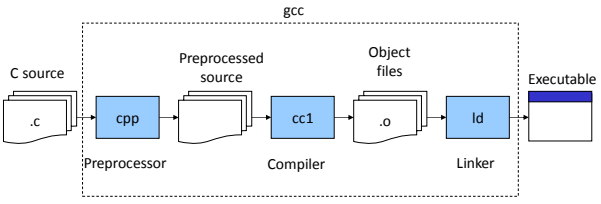
```
extern int buf[];

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

void swap()
{
  int temp;

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


Building an executable from multiple files

- Programs are translated and linked using a compiler driver
- *Compiler driver* coordinates all steps in the translation and linking process.
  - Typically included with each compilation system (e.g., `gcc`)
  - Invokes preprocessor (`cpp`), compiler (`cc1`), assembler (`as`), and linker (`ld`).
  - Passes command line arguments to appropriate phases
- Eg: create executable `p` from `main.c` and `swap.c`:

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


Compiler



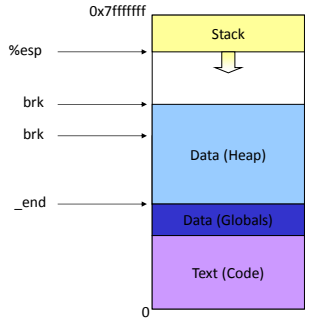
Types of object files

- Generated by compilers and assemblers
  - Relocatable object file
    - Contains code and data in a form that can be combined ***at compile time*** with other relocatable object files to form an executable
    - Each .o file is produced from exactly one source (.c) file
  - Shared object file
    - Special type of relocatable object file that can be loaded into memory and linked dynamically ***at either load or run time***
    - Called Dynamic Link Libraries (DLLs) in Windows

Executables

- Executables include everything needed to run on a system
  - Code
  - Data
  - Dependencies
  - Directions for laying out program in memory
- Self contained files that adhere to a standard format
  - Generating an executable is the job of a linker
  - Combines separate object files into a single executable

Process's Address Space



Why linkers?

- Linkers allow the combination of multiple files
  - object files, libraries, data files
- Modularity
  - Large program can be written as a collection of smaller files, rather than one monolithic mass
  - Can build libraries of common functions
    - e.g., Math library, standard C library
- Efficiency
  - Time:
    - Change one source file, compile, and then re-link
    - No need to recompile other source files
  - Space:
    - Libraries of common functions can be put in a single file…
    - Yet executable files and running memory images contain only code for the functions they actually use

What does a linker 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 compilers) in a *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 does a linker do?

- 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

Older Executable Formats

- a.out (Assembler OUTput)
  - Oldest UNIX format
  - No longer commonly used
- COFF (Common Object File Format)
  - Older UNIX Format
  - No longer commonly used

Modern Executable Formats

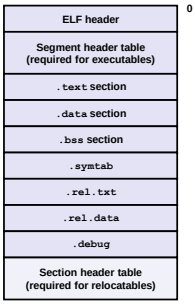
- PE (Portable Executable)
  - Based on COFF
  - Used in 32- and 64-bit Windows
- ELF (Executable and Linkable Format)
  - Linux/UNIX
- Mach-O file
  - Mac

Executable and Linkable Format (ELF)

- Standard binary format for object files
- Derives from AT&T System V Unix (Common Object File Format – COFF)
  - Later adopted by BSD Unix variants and Linux
- One unified format for
  - Executable object files
  - Relocatable object files (`.o`),
  - Shared object files (`.so`)
- Generic name: ELF binaries

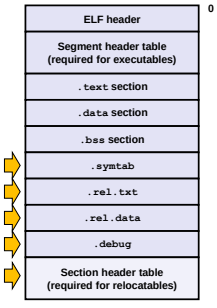
ELF object file format

- ELF header
  - Magic number, type (.o, exec, .so), machine, byte ordering, offset of section header table, etc.
- Segment header table
  - Page size, virtual addresses memory segments (sections), segment sizes.
- `.text` section
  - Code
- `.data` section
  - Initialized (static) data
- `.bss` section
  - Uninitialized (static) data
  - Originally an IBM 704 assembly instruction; "Block Started by Symbol" ("Better Save Space")
  - Has section header but occupies no space



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



Linker symbols

- Every relocatable object module (file) has a symbol table
  - Global symbols
    - Symbols defined by a module 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 a module but defined by some other module
  - Local symbols
    - Symbols that are defined and referenced exclusively by a module
    - E.g. C functions and variables defined with the static attribute
    - *Local linker symbols are not local program variables* (no symbols for local nonstatic program variables that are managed at runtime)

Global, external or local?

main.c

```
void swap();

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

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

swap.c

```
extern int buf[];

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

void swap() {
  int temp;

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

- In main.c
  - buf — Global
  - main — Global
  - swap — External (def. by swap.c)
- In swap.c
  - buf — External (def. by main.c)
  - bufp0 / bufpp1 — Global
  - swap — Global
  - temp — *A local variable; not a local symbol*

Global Symbol: defined by module, use by others (non-static C functions & global vars)

External Symbol: referenced here, defined by others

Local Symbol: defined & ref exclusively here (static vars and C functions)

17

ELF symbol table example

ELF symbol

```
int name;        /* String table offset */
int value;       /* Section offset, or VM address */
int size;        /* Object size in bytes */
char type:4,     /* Data, func, section or src file name (4 bits) */
     binding:4;  /* Local or global (4 bits) */
char reserved;   /* Unused */
char section;    /* Section header index, ABS, UNDEF or COMMON */
```

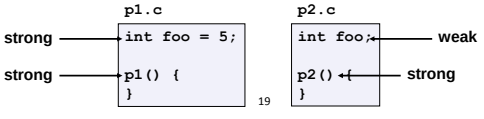
```
unix% readelf -s main.o

Symbol table '.symtab' contains 10 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 FILE    LOCAL  DEFAULT  ABS main.c
     2: 00000000     0 SECTION LOCAL  DEFAULT    1
     3: 00000000     0 SECTION LOCAL  DEFAULT    3
     4: 00000000     0 SECTION LOCAL  DEFAULT    4
     5: 00000000     0 SECTION LOCAL  DEFAULT    6
     6: 00000000     0 SECTION LOCAL  DEFAULT    5
     7: 00000000    20 FUNC    GLOBAL DEFAULT    1 main
     8: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND swap
     9: 00000000     8 OBJECT  GLOBAL DEFAULT    3 buf
```

Local symbols for use by the linker

18

Symbol resolution

- Linker associates each reference with exactly one symbol definition from symbol tables of its input files
  - Easy for references to local symbols
  - Trickier with global symbols – compiler assumes it is defined only once, *somewhere*, and that the linker will take care of it
    - If not anywhere, linker will complain
    - If more than once, maybe it will complain
- Program symbols are either strong or weak
  - *strong*: procedures and initialized globals
  - *weak*: uninitialized globals



19

Libraries

- Not all code in a program is what you wrote
- Use code that others have written in your own program
- How to include this code in your address space?

Linking

- Static Linking
  - Copy code into executable at compile time
  - Done by linker
- Dynamic Linking
  - Copy code into Address Space at load time or later
  - Done by link loader

Commonly used libraries

- `libc.a` (the C standard library)
  - 5 MB archive of 1500 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 400 object files.
  - floating point math (sin, cos, tan, log, exp, sqrt, …)

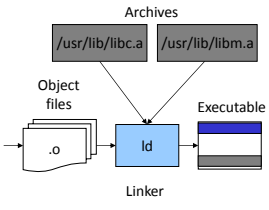
```
% ar -t /usr/lib/libc.a | sort
…
fork.o
…
fprintf.o
fpu_control.o
fputc.o
freopen.o
fscanf.o
fseek.o
fstab.o
…
```

```
% 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
…
```

22

Static Linking

```
#include <stdio.h>
#include <math.h>

int main() {
   printf("The sqrt of 9 is %f\n", sqrt(9));
   return 0;
}
```



Creating static libraries

- To create the library

```
unix% gcc -c addvec.c mulvec.c
unix% ar rcs libvector.a addvec.o mulvec.o
```

libvector.a

```
void addvec(int *x, int *y,
            int *z, int n)
{
  int i;

  for (i = 0; i < n; i++)
    z[i] = x[i] + y[i];
}
```

```
void multvec(int *x, int *y,
             int *z, int n)
{
  int i;

  for (i = 0; i < n; i++)
    z[i] = x[i] * y[i];
}
```

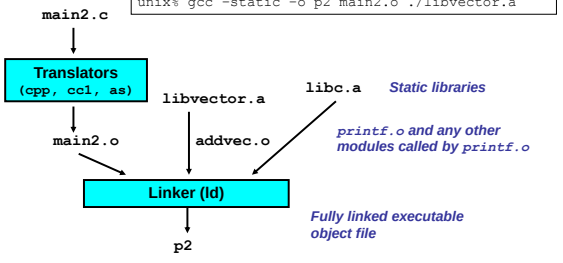
main2.c

```
#include <stdio.h>
#include "vector.h"

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

int main()
{
  addvec(x, y, z, 2);
  printf("z = [%d %d]\n", z[0], z[1]);
  return 0;
}
```

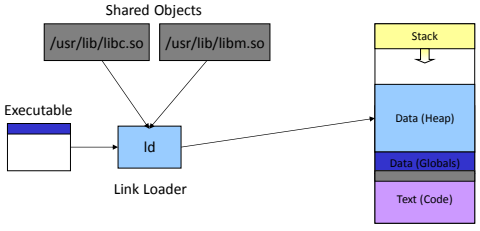
24

Linking with static libraries

```
unix% gcc -O2 -c main2.c
unix% gcc -static -o p2 main2.o ./libvector.a
```



Dynamic Linking



Dynamic linking from applications

- Why?
  - Distributing software – new versions of shared libraries used as they become available (MS Windows)
  - Building high-performance web servers – functions that generate dynamic content available as dll

```
#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 shared lib */
   handle = dlopen("./libvector.so",
    RTLD_LAZY);
   if (!handle) {
     fprintf(stderr, "%s\n", dlerror());
     exit(1);
   }
```

```
   /* get pointer to addvec() func loaded */
   addvec = dlsym(handle, "addvec");
   if ((error = dlerror()) != NULL) {
      fprintf(stderr, "%s\n", error);
      exit(1);
    }

   /* Now call addvec() as usual */
   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;
}
```

Dynamic Loading