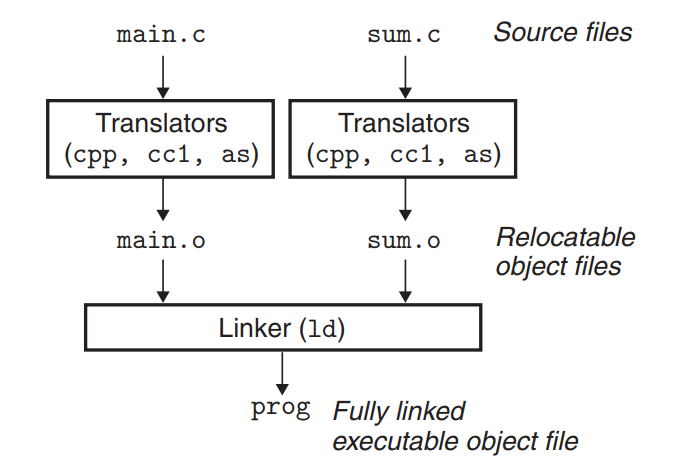
CS 361 Exam1

Linking: Collect and combine code and data into single file

* Then it gets loaded into memory and executed
* Linking can be performed at compile time, when the source code is translated into machine code; at load time, when the program is loaded into memory and executed by the loader; and even at run time, by application programs.
* In the modern system linking is performed automatically by a program called linkers.
* When we change one of these modules, we simply recompile it and relink the application, without having to recompile the other files.

Compiler Drivers

* Compilations systems provide compiler driver
* Invokes the language preprocessor, compiler, assembler and linker
* We can invoke gcc driver by - linux> gcc -Og -o prog main.c sum.c



1. The driver first runs the C preprocessor (cpp),1 which translates the C source file main.c into an ASCII intermediate file main.i:
2. Next, the driver runs the C compiler (cc1), which translates main.i into an ASCII assembly-language file main.s:
3. Then, the driver runs the assembler (as), which translates main.s into a binary relocatable object file main.o:
4. The driver goes through the same process to generate sum.o. Finally, it runs the linker program ld, which combines main.o and sum.o, along with the necessary system object files, to create the binary executable object file prog

Static Linking

* Take as input a collection of relocatable object files and command line arguments then generate a fully linked executable object file
* Input consists of various code and data section
  + Instructions in one section
  + Initialized global variable in another section
  + Uninitialized variables in another section

To build executable the linker performs:

1. Symbol resolution: Symbols correspond to a function, a global variable, or a static variable. Purpose is to associate each symbol reference with exactly one symbol definition.
2. Relocation: Compilers and assemblers generate code and data sections that start at address 0. Liners relocate these sections: associate memory location with symbol definition then modifies references to those symbols so they point to that memory location. Liners perform that using detailed instructions generated by assemblers, called relocation entries.

Object files are merely collections of blocks of bytes. Some contain code, data, data structures, that guide linker and loader

Linker: decides on runtime locations or concatenated blocks.

Object Files

1. Relocatable Object File: Contains binary code and that in a form that can be combined with other relocatable object files at compile time to create executable object files.
2. Executable Object File: Contains binary code and data in a form that can be copied directly into memory and executed.
3. Shared Object File: A special type of relocatable object file can be loaded into memory and linked dynamically at either load time or run time

Compiler and assembler generate relocatable object files (including shared object files) linkers generate executable object files

Object Module: sequence of bytes

Object File: object module stored on disk in a file

Relocatable Object Files

ELF header: begins with a 16 byte sequence that describes word size and byte ordering. Size of ELF header, object file type (Relocatable, executable, shared machine type (x86-64), file offset of the section header table and the size and number of entries in the section header table.

Sections:

* .text: The machine code of the compiled program.
* .rodata: Read-only data such as the format strings in printf statements, and

jump tables for switch statements.

* .data: Initialized global and static C variables. Local C variables are maintained

at run time on the stack and do not appear in either the .data or .bss

sections.

* .bss: Uninitialized global and static C variables, along with any global or static

variables that are initialized to zero. This section occupies no actual space in the object file; it is merely a placeholder. At run time, these variables are allocated in memory with an initial value of zero.

* .symtab: A symbol table with information about functions and global variables

Every relocatable object file has a symbol table in .symtab (unless the programmer has specifically removed it with the strip command).The .symtab symbol table does not contain entries for local variables.

* .rel.text: A list of locations in the .text section that will need to be modified

when the linker combines this object file with others. Any instruction that calls an external function or references a global variable will need to be modified. Instructions that call local functions do not need to be modified.

* .rel.data: Relocation information for any global variables that are referenced

or defined by the module. Any initialized global variable whose initial value is the address of a global variable or externally defined function will need to be modified.

* .debug: A debugging symbol table with entries for local variables and typedefs

defined in the program, global variables defined and referenced in the program, and the original C source file. It is only present if the compiler driver is invoked with the -g option.

* .line: A mapping between line numbers in the original C source program and

machine code instructions in the .text section. It is only present if the compiler driver is invoked with the -g option.

* .strtab: A string table for the symbol tables in the .symtab and .debug sections

and for the section names in the section headers. A string table is a sequence of null-terminated character strings.

Symbols and Symbol Tables

* Each relocatable object module, m has a symtable information about symbols defined and referenced by m.

1. Global symbols: that are defined by module m and that can be referenced by

other modules. Global linker symbols correspond to nonstatic C functions and global variables.

1. Global symbols that are referenced by module m but defined by some other

module. Such symbols are called externals and correspond to nonstatic C

functions and global variables that are defined in other modules.

1. Local symbols: that are defined and referenced exclusively by modulem. These

correspond to static C functions and global variables that are defined with the static attribute. These symbols are visible anywhere within module m, but cannot be referenced by other modules.

Local linker symbols != local program variable

* Symtab doesn’t contain any symbols that corresponds to local non static program variables
* These are managed at runtime on stack, no interest to linker
* Binding field indicates whether the symbol is local or global

Pseudosections(no entries in section header file):

* ABS: symbols that should not be relocated
* UNDEF: undefined symbols (symbols referenced in this object module but define elsewhere
* COMMON: uninitialized objects not yet allocated

Pseudosections only exist in relocatable object files not in executable object files

* COMMON: uninitialized global variables
* .bss: uninitialized static variables, and global or static variables initialized to zero

Symbol Resolution

* The linker resolves symbol references by associating each reference with exactly one symbol definition from the symbol tables of its input relocatable object files.
* The compiler allows only one definition of each local symbol per module.
* The compiler also ensures that static local variables, which get local linker symbols, have unique names.
* Resolving references to global symbols, however, is trickier.
* When the compiler encounters a symbol (either a variable or function name) that is not defined in the current module, it assumes that it is defined in some other module, generates a linker symbol table entry, and leaves it for the linker to handle.
* If the linker is unable to find a definition for the referenced symbol in any of its input modules, it prints an (often cryptic) error message and terminates.

How Linkers Resolve Duplicate Symbol Names

Rule 1. Multiple strong symbols with the same name are not allowed.

Rule 2. Given a strong symbol and multiple weak symbols with the same name, choose the strong symbol.

Rule 3. Given multiple weak symbols with the same name, choose any of the weak symbols.

Linking with Static Libraries

* When building the output executable, the linker copies only the object modules in the library that are references by the application program
* On Linux systems, static libraries are stored on disk in a particular file format known as an archive.

How Linkers Use Static Libraries to Resolve References

* For each input file f on the command line, the linker determines if f is an

object file or an archive. If f is an object file, the linker adds f to E, updates U and D to reflect the symbol definitions and references in f , and proceeds to the next input file.

* If f is an archive, the linker attempts to match the unresolved symbols in U

against the symbols defined by the members of the archive. If some archive member m defines a symbol that resolves a reference in U, then m is added to E, and the linker updates U and D to reflect the symbol definitions and references in m.

* If U is nonempty when the linker finishes scanning the input files on the

command line, it prints an error and terminates. Otherwise, it merges and relocates the object files in E to build the output executable file.

Relocation

Linker merges input modules and assigns run time addresses to each symbol

1. Relocating sections and symbol definitions: In this step, the linker merges all

sections of the same type into a new aggregate section of the same type. Then assigns a run-time memory address to it. Each instruction and global variable will have a unique run-time memory address.

1. Relocating symbol references within sections: In this step, the linker modifies

every symbol reference in the bodies of the code and data sections so that they point to the correct run-time addresses. To perform this step, the linker relies on data structures in the relocatable object modules

Relocation Entries

* When an assembler generates an object module, it does not know where the code

and data will ultimately be stored in memory. Nor does it know the locations of any externally defined functions or global variables that are referenced by the module. So whenever the assembler encounters a reference to an object whose ultimate location is unknown, it generates a relocation entry that tells the linker how to modify the reference when it merges the object file into an executable.

Relocation entries for code -> .rel.text

Relocation entries for data -> .rel.data

Dynamic Linking with Shared Libraries

* Static libraries: making large collections of related function available to application programs
  + Disadvantage: need to be maintained and updated periodically. Programs might need to explicitly relink to updated library

Memory is always scarce resource

* Shared Libraries: are modern innovations that address the disadvantages of

static libraries. A shared library is an object module that, at either run time or load time, can be loaded at an arbitrary memory address and linked with a program in memory. This process is known as dynamic linking and is performed by a program called a dynamic linker. Shared libraries are also referred to as shared objects, and on Linux systems they are indicated by the .so suffix.

1. In any given file system, there is exactly one .so file for a particular library. The

code and data in this .so file are shared by all of the executable object files that reference the library, as opposed to the contents of static libraries, which are copied and embedded in the executables that reference them. Second, a single copy of the .text section of a shared library in memory can be sh

1. Single copy of the .text section of the shared library in memory can be shared by different running processes.

* To build a shared library libvector.so -> invoke compiler
* linux> gcc -shared -fpic -o libvector.so addvec.c multvec.c
  + Directs compiler to generate position independent code. Directs the linker to create shared object files

The we can link it to program: linux> gcc -o prog2l main2.c ./libvector.so

That creates an executable object file: prog2l

1. Some static linking done
2. Dynamic linking when program is loaded
   1. Nothing from data section from libvector.so copied into executable
   2. Linker copies relocation and symbol table info that will allow references to code and data in libvector.so to be resolved at load time

Loader runs and executes prog21:

* It loads the partially linked executable prog2l
* Notices that prog2l contains a .interp section, which contains the path name of the dynamic linker
* Instead of passing control to the application, as it would normally do, the loader loads and runs the dynamic linker.
* The dynamic linker then finishes the linking task
  + Relocating the text and data of libc.so into some memory segment
  + Relocating the text and data of libvector.so into another memory segment
  + Relocating any references in prog2l to symbols defined by libc.so and libvector.so

Finally, the dynamic linker passes control to the application. From this point on,

the locations of the shared libraries are fixed and do not change during execution

of the program.

Loading and Linking Shared Libraries from Applications

void \*dlopen(const char \*filename, int flag);

* The dlopen function loads and links the shared library filename.
* The external symbols in filename are resolved using libraries previously opened with the RTLD\_GLOBAL flag.
* If the current executable was compiled with the -rdynamic flag, then its global symbols are also available for symbol resolution.
* RTLD\_NOW, which tells the linker to resolve references to external symbols immediately
* RTLD\_LAZY flag, which instructs the linker to defer symbol resolution until code from the library is executed.

void \*dlsym(void \*handle, char \*symbol);

The dlsym function takes a handle to a previously opened shared library and a symbol name and returns the address of the symbol, if it exists, or NULL otherwise.

int dlclose (void \*handle);

The dlclose function unloads the shared library if no other shared libraries are still using it.

const char \*dlerror(void);

Returns: error message if previous call to dlopen, dlsym, or dlclose failed; NULL if previous call was OK

Position-Independent Code (PIC)

* Code that can be loaded without needing any relocations is known as position independent code (PIC). Users direct GNU compilation systems to generate PIC code with the -fpic option to gcc. Shared libraries must always be compiled with this option.
* References to symbols in the same executable object module require no special treatment to be PIC. These references can be compiled using PC-relative addressing and relocated by the static linker when it builds the object file. However, references to external procedures and global variables that are defined by shared modules require some special techniques

PIC Data References generated by compilers

* No matter where we load an object module (including shared object modules) in memory, the data segment is always the same distance from the code segment.
* Thus, the distance between any instruction in the code segment and any variable in the data segment is a run-time constant
* Compilers that want to generate PIC references to global variables exploit this fact by creating a table called the global offset table (GOT) at the beginning of the data segment.
* The GOT contains an 8-byte entry for each global data object (procedure or global variable) that is referenced by the object module.
* The compiler also generates a relocation record for each entry in the GOT. At load time, the dynamic linker relocates each GOT entry so that it contains the absolute address of the object. Each object module that references global objects has its own GOT.

PIC Function Calls

* LAZY Binding: defers binding of each procedure address until the first time the procedure is called
  + Dynamic linker, then avoid unnecessary relocations at run-time
* Implemented with a compact yet somewhat complex interaction between two data structures: the GOT and the procedure linkage table (PLT).
* If an object module calls any functions that are defined in shared libraries, then it has its own GOT and PLT. The GOT is part of the data segment. The PLT is part of the code segment.
* PLT is an array of 16-byte code entries
* GOT is an array of 8-byte address entries