— Link「ing,

Linking

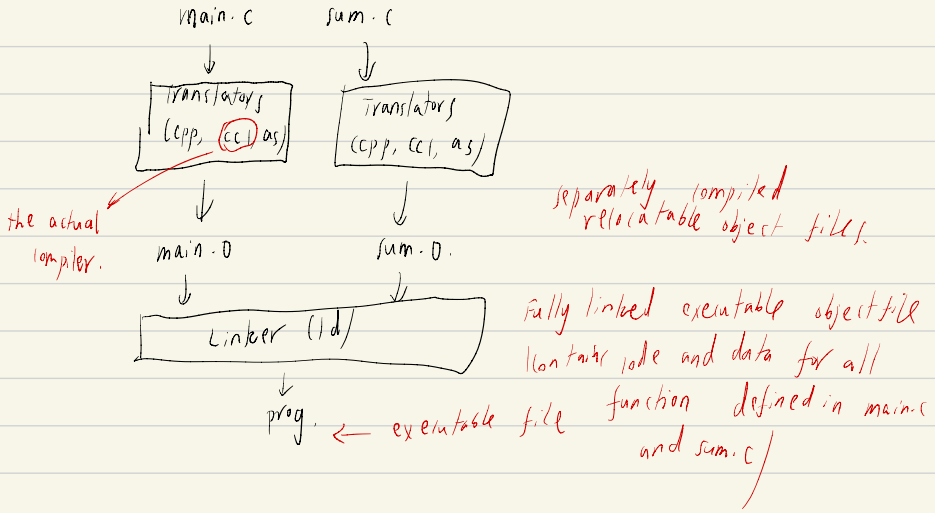
How the system build the program?

Case study: Library interpositioning

(I) Programs are translated and linked using a compiler driver.

linux> gcc -Og -o prog main.c sum.c

linux> ./prog



Why Linkers?

(I) Reason 1: Modularity:

(i) Program can be written as a collection of smaller source files rather than one monolithic mass.

(ii) Can build libraries of common functions (more on this later)

e.g. math lib. std. C. library.

(II) Reason 2: Efficiency:

(i) Time: Separate compilation:

Change one source file, compile, and then relink.

No need to recompile other source files.

(ii) Space: libraries.

(i) Common functions can be aggregated into a single file.

(ii) Yet executable files and running memory images contain only code for the functions they actually use.

What do linkers do?

(I) Step 1: Symbol resolution.

(i) Programs define and reference symbols (global variables and functions):

```
void swap() {...}      /* define symbol swap */
swap();                /* reference symbol swap */
int *xp = &x;          /* define symbol xp, reference x */
```

Symbol definitions are stored in object file (by assembler) in symbol table.

(ii) Symbol table is an array of structs.

(iii) Each entry includes name, size, and location of symbol.

During symbol resolution step, the linker associates each symbol with exactly one symbol definition.

(II) Step 2: Relocation:

(i) Merges separate code and data sections into single sections.

(ii) Relocates symbols from their relative locations in the .o files to their final absolute memory location in the exe file.

(iii) Updates all references to these symbols to reflect their new positions.

Three kinds of object Files (Modules)

(I) Relocatable object file

(i) contains code and data in a form that can be combined with other relocatable object files to form executable object file.

(ii) Each .o file is produced from exactly one source (.c) file.

(II) Executable object file (a.out file).

(i) contains code and data in a form that can be copied directly into memory and then executed.

(III) shared object file (.so file).

(i).

Executable and Linkable Format (ELF)

(I) Std. binary format for object files

(II) One unified format for.
- (i) Relocatable object files (.o),
- (ii) Executable object files (a.out).
- (iii) Shared object files (.so)

(III) Generic name: ELF binaries

ELF Object file Format.

(I) Elf header
- (i) Word size, byte ordering, file type (.o, .exec, .so), machine type, etc.

(II) Segment header table
- (i) Page size, virtual addresses memory segments (sections), segment sizes.

(III) .text section.
- (i) code

(IV) .rodata section
- (i) Read only data: jump tables...

(V) .bss section: (better saved space)
- (i) Uninitialized global variables
- (ii) "Block started by symbol"
- (iii) "Better Save Space"
- (iv) Has section header but occupies no space.

ELF header
Segment header
Segment header table (required for executables)
.text section
.rodata section
.data section
.bss section
.symtab section
.rel.txt section
.rel.data section
.debug section
Section header file.

Symbol table

notes.

To let the assembler to know where the symbol is going to be stored in the memory.

section for the symbol table:

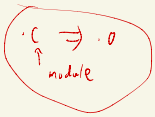
(I) symtab section:
  (i) Symbol table
  (ii) Procedure and static variable names.
  (iii) Section names and locations.

(II) .rel.text section.
  (i) Relocation info for .text section.
  (ii) Addresses of instructions that will need to be modified in the executable.
  (iii) Instructions for modifying

(III) .rel.data section.
  (i) Relocation info for .data section.
  (ii) Addresses of pointer data that will need to be modified in the merged executable.

(IV) .debug section:
  info for symbolic debugging (gcc -g)

(V) section header table
  Offsets and sizes of each section.



Linkers

three different kinds of symbols.

(I) Global symbols.

(i) Symbols defined by module m that can be referenced by other modules.

(ii) E.g.: non-static C functions and non-static global variables.

(II) External symbols

(i) Global symbols that are referenced by module m but defined by some other module.

(III) Local symbols

(i) Symbols that are defined and referenced exclusively by module m.

E.g. C functions and global variables defined with the static attribute.

Local linker symbols are not local program variables

Local symbols.

(2) Local non-static C variables vs. Local static C variables.

(i) Local non-static C variables: stored on the stack

(ii) Local static C variables: stored in either .bss or .data.

Stored in the wrong way as a global rather than on the stack

lec maint. and sym. C.

How Linker Resolves Duplicate Symbol definitions?

(a) Program symbols are either strong or weak.

(I) strong: procedures and initialized globals.

(II) Weak: Uninitialized globals.

See p1.c & p2.c

Rule 1: Multiple strong symbols are not allowed.

(i) Each item can be defined only once.

(ii) Otherwise: Linker error

Rule 2: Given a strong symbol and multiple weak symbols, choose the strong symbol

(i) References to the weak symbol resolve to the strong symbol.

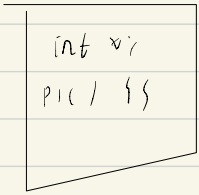
Rule 3: If there are multiple weak symbol, pick an arbitrary one.

Can override this with.

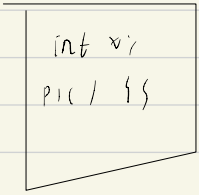
gcc –fno –common

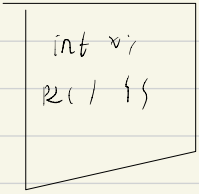
It will throw an multiple weak symbol error.

See Linker_Puzzles_1.C.

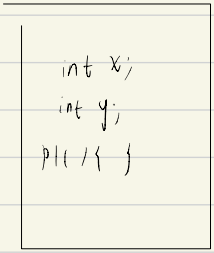


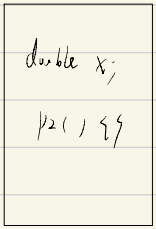
Link time error: two strong symbols (p1)





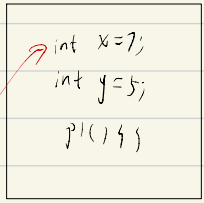
References to x will refer to the same uninitialized int. Is this what you really want?

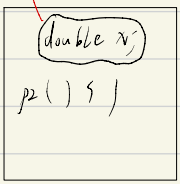




Writes to x in p2 might overwrite y! Evil!

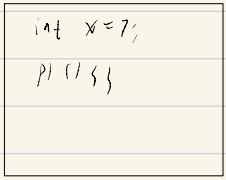
size is as twice as int.

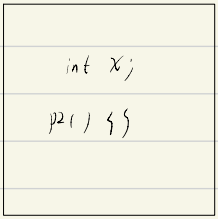




Writes to x in p2 will overwrite y! Nasty!

Linker will always associate all references to x to this integer sized this integer sized symbol.





Reference to x will refer to the same int variable.

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

Global Variable

(2) Avoid if you can

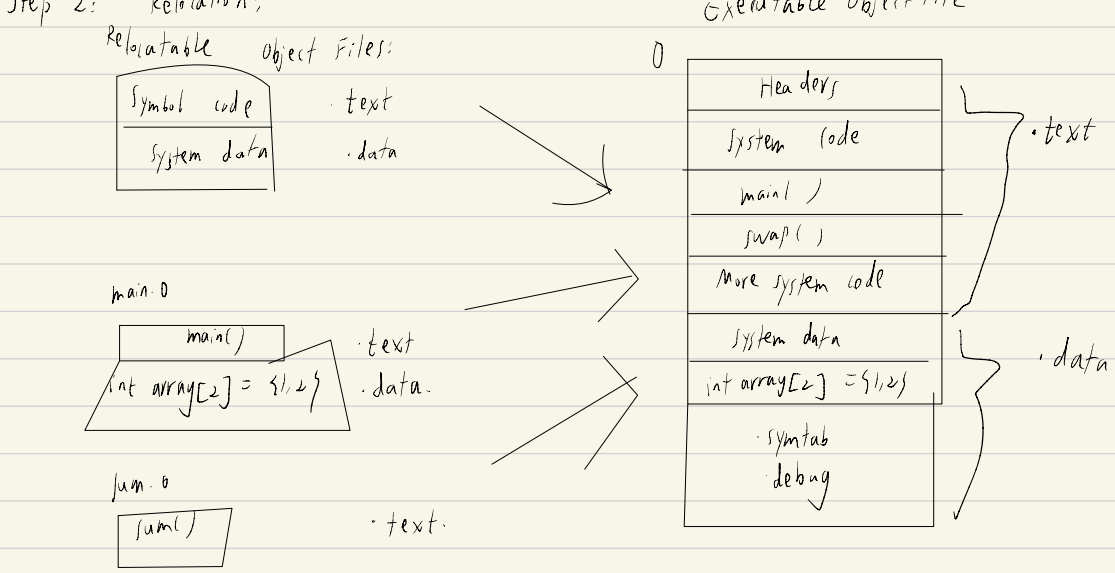
Otherwise

(i) Use static if you can

(ii) initialize if you define a global variable

(iii) Use extern if you reference an external global variable.

Step 2: Relocation;



Linker should always have enough info to actually fill in the right address

Packaging Commonly Used Functions?

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

(II) Awkward, given the linker framework so far:
(i) Option 1: Put all functions into a single source file:
1. programmers link big object file into their programs
3. Space and time inefficient,

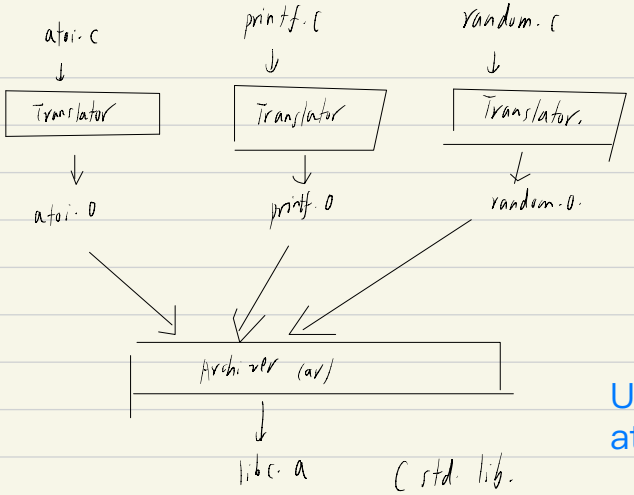
(ii) Option 2: put each function in a separate source file,
Programmers explicitly link appropriate binaries into their programs
Most efficient, but burdensome on the programmer

static libraries (.a archive files)

(i) concatenate related relocatable object files into a single file with an index (called an archive).

(ii) Enhance linker so that it tries to resolve unresolved external references by looking for the symbols in one or more archives.

(iii) If an archive member resolves reference, link it into the executable.



Unix> ar ram libc.a
atoi.o prints.o ... random.o.

Id Archiver allows incremental updates.

Recompile function that changes and replace .o file in archive.

Using Static Libraries:

(4 Linker's algorithm for resolving external references:

(i) Scan .o files and .a files in the command line order.

(ii) During the scan, keep a list of the current unresolved references.

(iii) 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.

(iiii) If any entries in the unresolved list at the end of scan, then error.

problem: Command line order matters!

Moral: put libs at the end of the command line

Modern solution: Shared libraries

(2) Static libraries have the following disadvantages:

(i) Duplication in the stored executables (every function needs libc)

(ii) Duplication in the running executables

(iii) Minor bug fixes of system libs. require each application to explicitly relink.

Modern solution: Shared libraries (dll) (dynamic lib.)

- 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

(I) Dynamic linking can occur when executable is first loaded and run (load-time linking)

(II) Dynamic linking can also occur after program has begun (run-time linking)

(III) Shared libraries routines can be shared by multiple processes.