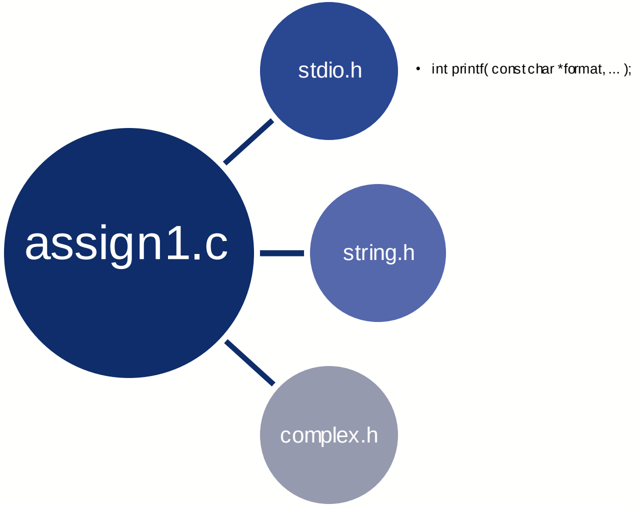
Johns Hopkins Engineering

Module 11: Linkers (Link Editors)

EN605.204: Computer Organization

Modules

- Assembler produces machine code in an "object file" (module)
- Each object file has a set of related functions
- C Standard Library uses header files (*.h) to specify a module's API
- Header files contain function declarations and constants
  - stdio.h, string.h, complex.h (C header files)
- C modules are implemented in *.c files
  - stdio.c, string.c, complex.c
- Client code (assign1.c) imports pre-written modules
  - C header files: #include <stdio.h>
  - Custom header files: #include "my_module.h"

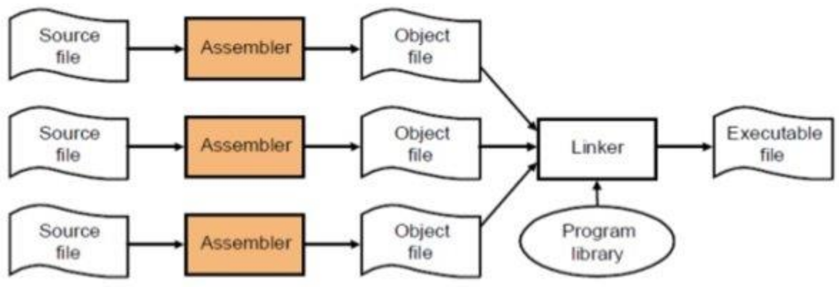


Advantages of Modularity

- Logical separation of source code by function
- Developers do not have to re-implement commonly used code
- Easy to swap or update modules
- Change to one module does not require ENTIRE library to be recompiled
- How do modules get linked into our program?

Linker (Link Editor)

- Links all referenced object files into a single executable
- Linkers are responsible for:
  - Symbol resolution ([recursive] dependencies)
    - Associates each symbol reference with a single symbol definition
  - Search installed libraries to find function code
  - Relocate code and combine pieces into a single executable



Symbols

- Symbols: think variables and functions
- Symbol table maps symbols to their metadata at compile-time
- "Global": defined by a module for use reference by other modules
  - Ex: a function named 'fibonacci()' or a constant named 'PI'
- "External": used by one module but defined by another
  - Ex: assign1.c includes stdio.h and calls printf()
    - printf() is "global" within stdio.c and "external" with respect to assign1.c
- "Local": defined by a module and used only within that module

global

external

main.c	math.c

`int i, j = 0;`
`int sum = add(i, j)`
	`int add(int i, int j)`
	`{`
	`    int sum = i+j;`
	`    return sum;`
	`{`

Symbol Resolution

- Linker ensures no unresolved references exist
- main.c and stdio.c are both compiled and a symbol table is built
- stdio.c contains a symbol named 'printf', main.c contains a reference to 'printf' and 'sqrt'
- Linker examines the symbol table and sees main.c's reference to 'printf' and 'sqrt'
- Linker finds a corresponding label called 'printf' from 'stdio' but does not find a correspond label for 'sqrt'
  - 'printf' is define in stdio.h which was included
  - 'sqrt' is defined in math.h which was not included
- If you've ever gotten a "method not found error", this is why!
  - linker could not find the method you tried to call in the symbol table because you forgot to import it

```
main.c:6:43: warning: implicitly declaring library function 'sqrt' with type
      'double (double)' [-Wimplicit-function-declaration]
printf("The square root of %f is %f.", i, sqrt(i));
                                          ^
main.c:6:43: note: include the header <math.h> or explicitly provide a declaration for 'sqrt'
1 warning generated.
```

main.c

```
#include <stdio.h>

double i = 4;
printf("The square root of %f is %f.", i, sqrt(i));
```

Compiler File Types

- The entire compiling process consists of multiple discrete steps
- Output from one step becomes the input to the next step
- Here are some common (Linux) file types (in order):

.s file: assembly code file generated by the compiler (gcc)

↓

.o file: object file generated by the assembler (as) - unassembled, relocatable machine code

↓

binary executable generated by the linker (ldd) - ELF format on Linux, EXE on Windows

C Compilation Process

- GCC and CLANG are the most widely-used C compilers
- There are three main components:
  - gcc: compiler
  - as: assembler
  - ld: linker
- Given a file named `demo.c` file, running `gcc demo.c` will complete the entire compile, assemble, and linking process and generate a file called `a.out` which is an executable inary (machine code)
- The resulting file can be viewed using: `objdump -D a.out` (shown at right)

```c
#include <stdio.h>
int main()
{
    printf("Hello, world!");
    return 0;
}
```

```
Disassembly of section .interp:

0000000000000238 <.interp>:
 238:   2f                      (bad)
 239:   6c                      insb   (%dx),%es:(%rdi)
 23a:   69 62 36 34 2f 6c 64    imul   $0x646c2f34,0x36(%rdx),%esp
 241:   2d 6c 69 6e 75          sub    $0x756e696c,%eax
 246:   78 2d                   js     275 <_init-0x27b>
 248:   78 38                   js     282 <_init-0x26e>
 24a:   36 2d 36 34 2e 73       ss sub $0x732e3436,%eax
 250:   6f                      outsl  %ds:(%rsi),(%dx)
 251:   2e 32 00                xor    %cs:(%rax),%al
  Ubuntu Software
Disassembly of section .note.ABI-tag:

0000000000000254 <.note.ABI-tag>:
 254:   04 00                   add    $0x0,%al
 256:   00 00                   add    %al,(%rax)
 258:   10 00                   adc    %al,(%rax)
 25a:   00 00                   add    %al,(%rax)
 25c:   01 00                   add    %eax,(%rax)
 25e:   00 00                   add    %al,(%rax)
 260:   47                      rex.RXB
 261:   4e 55                   rex.WRX push %rbp
 263:   00 00                   add    %al,(%rax)
 265:   00 00                   add    %al,(%rax)
 267:   00 03                   add    %al,(%rbx)
 269:   00 00                   add    %al,(%rax)
 26b:   00 02                   add    %al,(%rdx)
 26d:   00 00                   add    %al,(%rax)
 26f:   00 00                   add    %al,(%rax)
 271:   00 00                   add    %al,(%rax)
        ...
```

C Assembling Process

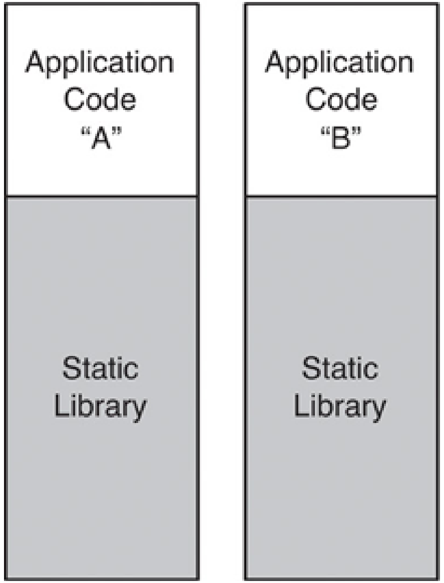
- We can use the –s flag to make GCC stop after the assembly step:
  - `gcc –s demo.c`
  - The result is a file called `demo.s`
- We can then inspect the resulting demo.s file to view the assembly code directly (the example shown is x86 assembly, not MIPS)

```
#include <stdio.h>
int main()
{
    printf("Hello, world!");
    return 0;
}
```

```
        .file   "demo.c"
        .text
        .section        .rodata
.LC0:
        .string "Hello, world!"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        leaq    .LC0(%rip), %rdi
        movl    $0, %eax
        call    printf@PLT
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0"
        .section        .note.GNU-stack,"",@progbits
```

Static Linking

- The assembler produces .o files containing machine code
- .o files can be grouped into an archive file with an extension of .a
  - *.a files contain a group of compiled but unlinked modules
- Static linking pulls your imported library code from a *.a file and embeds it inside of your executable (increases program size and resources needed)
- Each application that references a static library pulls in its own copy
  - storing duplicate (or more) copies of the same code is wasteful
- Execution is fast because library functions are loaded into memory along with your program



Code Relocation

```
my_library.o

myfunc:
  ...
  # do something
  jr ra
```

```
main.o

main:
    ...
    jal myfunc
    ...
    Jal print
```

```
stdio.o

printf:
    # print string
    jr ra
```

Linker

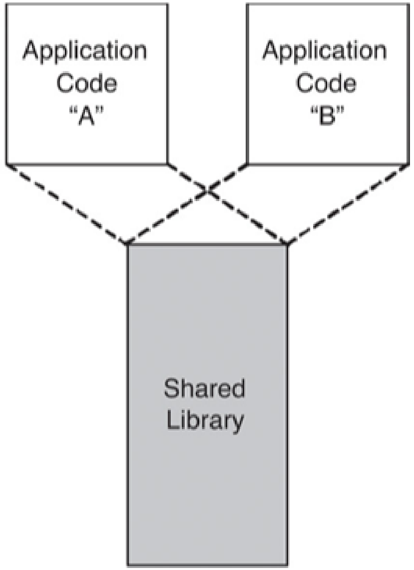
```
main:

    jal myfunc
    ...
    jal printf
    ...
    myfunc:
        ...
    printf:
        ...
```

Fully-linked executable

Linker patches together the reloctable object code

Dynamic Linking

- Assembly code can also be grouped into shared object files (*.so)
- *.so files are linked at run-timeby the OS rather than compile time by the linker
  - called a DLL on Windows (dynamically-linked library)
- *.so files are not embedded into the application like in static linking
  - a reference to the library is inserted into the executable instead
- When a program is executed, shared library is pulled into memory and linked
- If another program that uses the same shared library is executed it will access the shared library code already in memory (see image)
- This reduce the number of times the library needs to be loaded
- It is slower because the application must access an intermediate symbol table for each call to a function in the shared library
  - Why symbol table? What happens if App A and App B finishing running, the shared library is removed from memory, then App C (which also access the same shared library) begins to execute?




```
student@student-VirtualBox:/var/lib$ ls /usr/lib/python3.6/config-3.6m-x86_64-linux-gnu/
config.c  config.c.in  install-sh  libpython3.6m.a  libpython3.6m-pic.a  libpython3.6m.so  libpython3.6.so
```