NC STATE UNIVERSITY
Department of Computer Science

CSC501
Operating Systems Principles

Linking & Loading

Previous Lectures

- Midterm

- Today
  - Linking and Loading

Computer Science NC STATE UNIVERSITY

Question:

What happened to your program after it is compiled, but before it can be run?

Background: Executable Files

- The OS expects executable files to have a specific format
  - Header info
    - Code locations
    - Data locations
  - Code & data
  - Symbol Table
    - List of names of things defined in your program and where they are defined
    - List of names of things defined elsewhere that are used by your program, and where they are used.

Example: ELF Files (x86/Linux)

Linkable sections		***Executable segments***
	ELF Header	
Demo! (optional, ignored)	Program Header Table	describes sections
sections		segments
describes sections	Section Header Table	(optional, ignored)

Example

```
#include <stdio.h>

int main () {

   printf ("hello, world\n")

}
```

- Symbol defined in your program and used elsewhere
  - `main`
- Symbol defined elsewhere and used by your program
  - `printf`

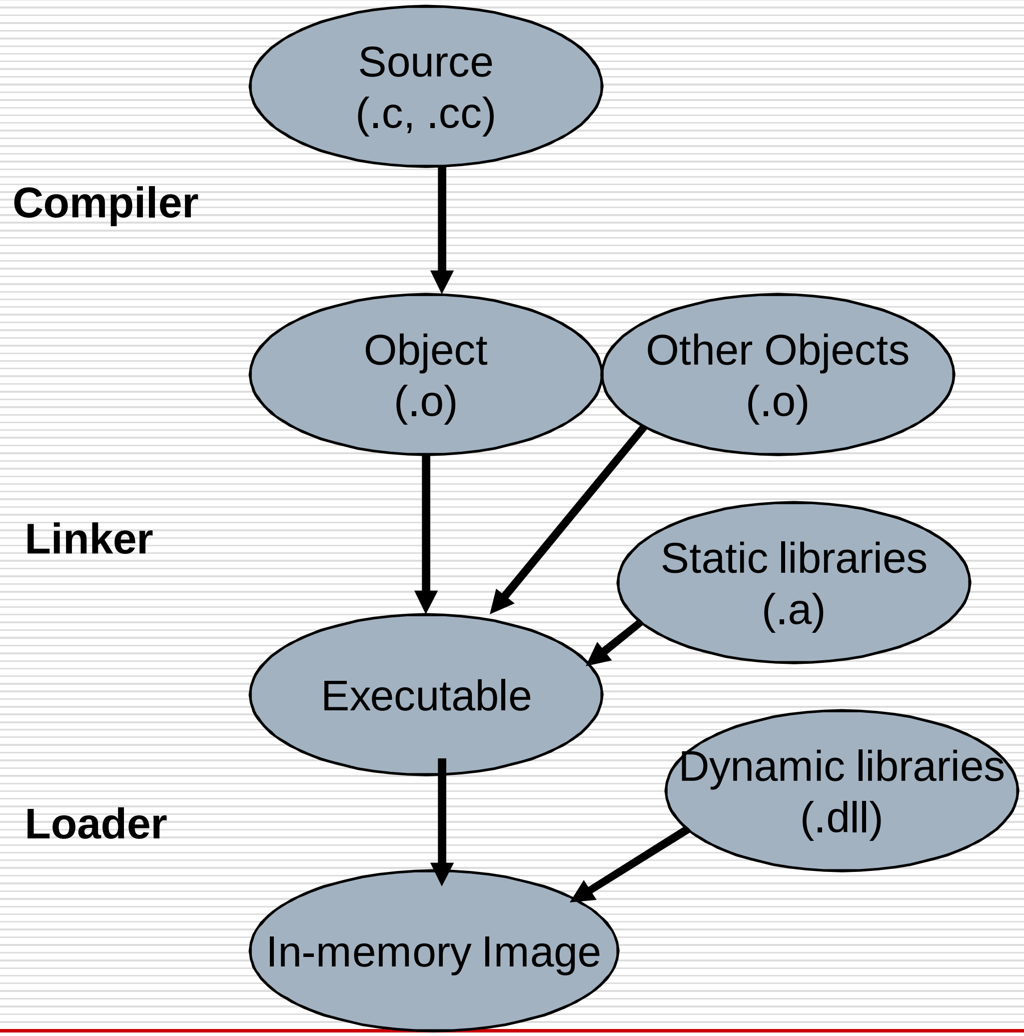
Example

```
#include <stdio.h>
extern int errno;

int main () {

   printf ("hello,
   world\n")

   <check errno for
   errors>
}
```

- Symbol defined in your program and used elsewhere
  - `main`

- Symbol defined elsewhere and used by your program
  - `printf`
  - `errno`

From source code to a process



From source code to a process

- Most compilers produce relocatable object code
  - Addresses relative to *zero* or a prefixed location
- The linker combines multiple object files and library modules into a single executable file
  - Addresses also relative to *zero* or a prefixed location
  - Resolving symbols defined within these files
  - Listing symbols needing to be resolved by loader
- The Loader reads the executable file
  - Allocates memory
  - Maps addresses within file to physical memory addresses
  - Resolves names of dynamic library items

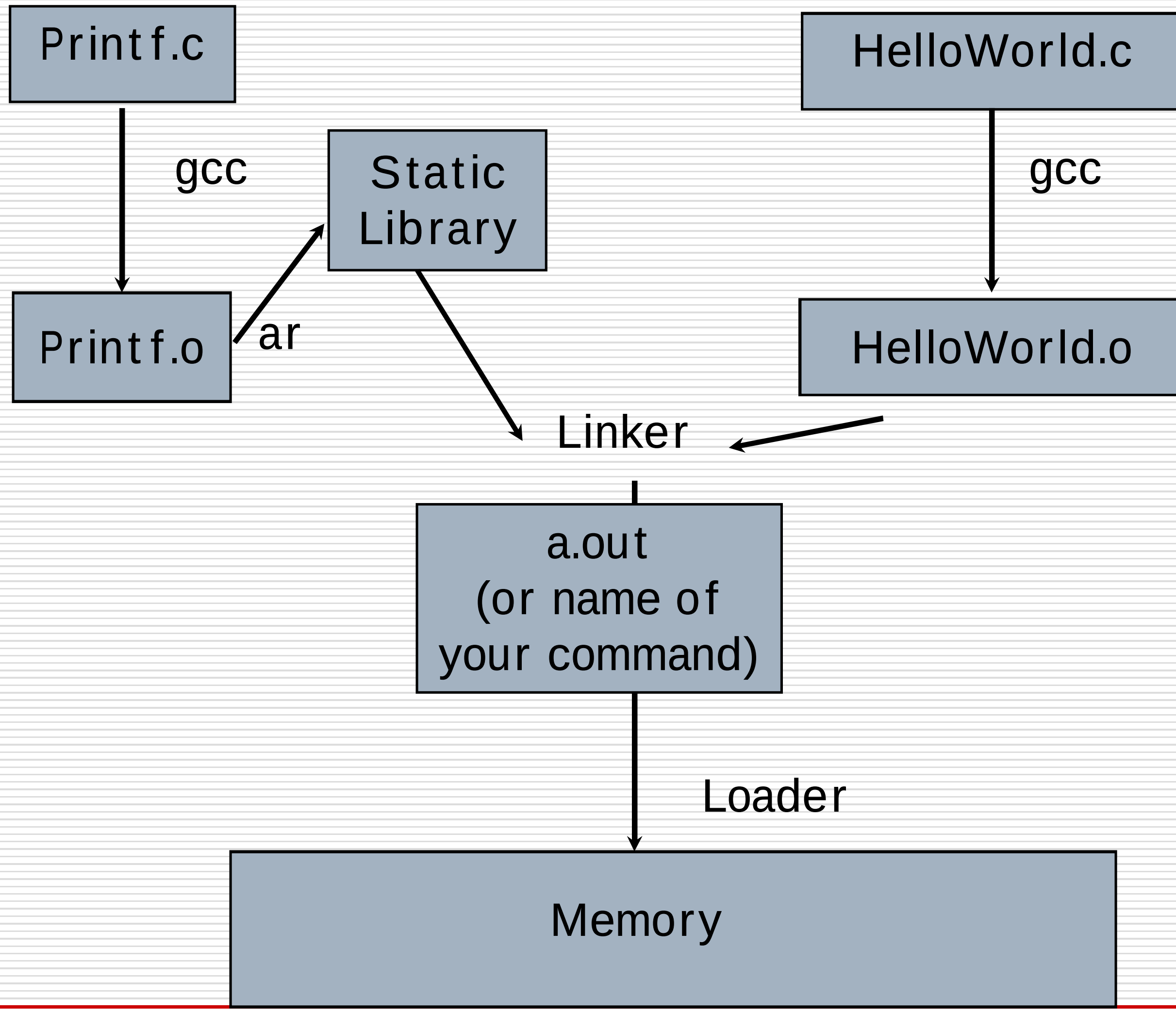
Compiling

- Why isn't everything written and compiled as just one big program, saving the necessity of linking?
  - Efficiency: if just one function is changed in a 100K line program, why recompile the whole program? Just recompile the one function and relink
  - Multiple-language programs
  - Other reasons?

Linking

- Linker collects procedures and links them together object modules into one executable program
- Two approaches
  - Static linking
  - Dynamic dinking

Static Linking



Static Linking -- Classic Unix

- Linker is inside of *gcc* command
- Loader is part of *exec* system call
- Executable image contains *all* object and library modules needed by program
- Entire image is loaded at once

- Every image contains copy of common library routines
- Every loaded program contain duplicate copy of library routines

Dynamic Linking

- Complete linking postponed until execution time.
- Stub used to locate the appropriate memory-resident library routine.
- Stub replaces itself with the address of the routine, and executes the routine.
- Operating system needs to check if routine is in processes' memory address space.

Dynamic Linking

- Dynamic vs. static linking

```
$ gcc -static hello.c -o hello-static
$ gcc hello.c -o hello-dynamic
$ ls -l hello
     80 hello.c
  13724 hello-dynamic
    383 hello.s
1688756 hello-static
```

- If you are the sys admin, which do you prefer?

Advantages of Dynamic Linking

- The executable is smaller (it not include the library information explicitly),
- When the library is changed, the code that references it does not usually need to be recompiled.
- The executable accesses the .DLL at run time; therefore, multiple codes can access the same .DLL at the same time (saves memory)

Disadvantages of Dynamic Linking

- Performance hit ~10%
  - Need to load shared objects (once)
  - Need to resolve addresses (once or every time)
- What if the necessary dynamic library is missing?
- Could have the library, but wrong version

Unix Dynamic Objects (.so)

- Compiler Options (cont)
  - -static link only to static (.a=archive) libraries
  - -shared if possible, prefer shared libraries over static
  - -nostartfiles skip linking of standard start files, like /usr/lib/crt[0,1].o, /usr/lib/crti.o, etc
- Linker Options (gcc gives unknown options to linker)
  - -l lib (default naming convention lib*lib*.a)
  - -L lib path (in addition to default /usr/lib and /lib)
  - -s strip final executable code of symbol and relocation tables

Loading

- It loads a program file for execution

- Two approaches
  - Static loading
  - Dynamic loading

- Advantages of dynamic loading
  - Better memory-space utilization; unused routine is never loaded.
  - Useful when large amounts of code are needed to handle infrequently occurring cases

Dynamic Loading

- Program-controlled dynamic loading
- Linker-assisted dynamic loading

Program-controlled Dynamic Loading

- Requires:
  - A *load* system call to invoke loader
  - ability to leave symbols unresolved and resolve at run time
- E.g.,

```
void myPrintf (**arg) {
   static int loaded = 0;
   if (!loaded ) {
   load (“printf”);
   loaded = 1;
   printf(arg);
   }
}
```

Linker-assisted Dynamic Loading

- Programmer marks modules as “dynamic” to linker
- For function call to a dynamic function
  - Call is indirect through a *link table*
  - Each link table initialized with address small *stub* of code to locate and load module.
  - When loaded, loader replaces link table entry with address of loaded function
  - When unloaded, loader replaces table entry with stub address
  - Static data cannot be made dynamic

Shared Libraries

- Observation – “everyone” links to standard libraries (*libc.a*, etc.)
- Consume space in
  - every executable image
  - every process memory at runtime

- Would it be possible to share the common libraries?
  - Automatically load at runtime?

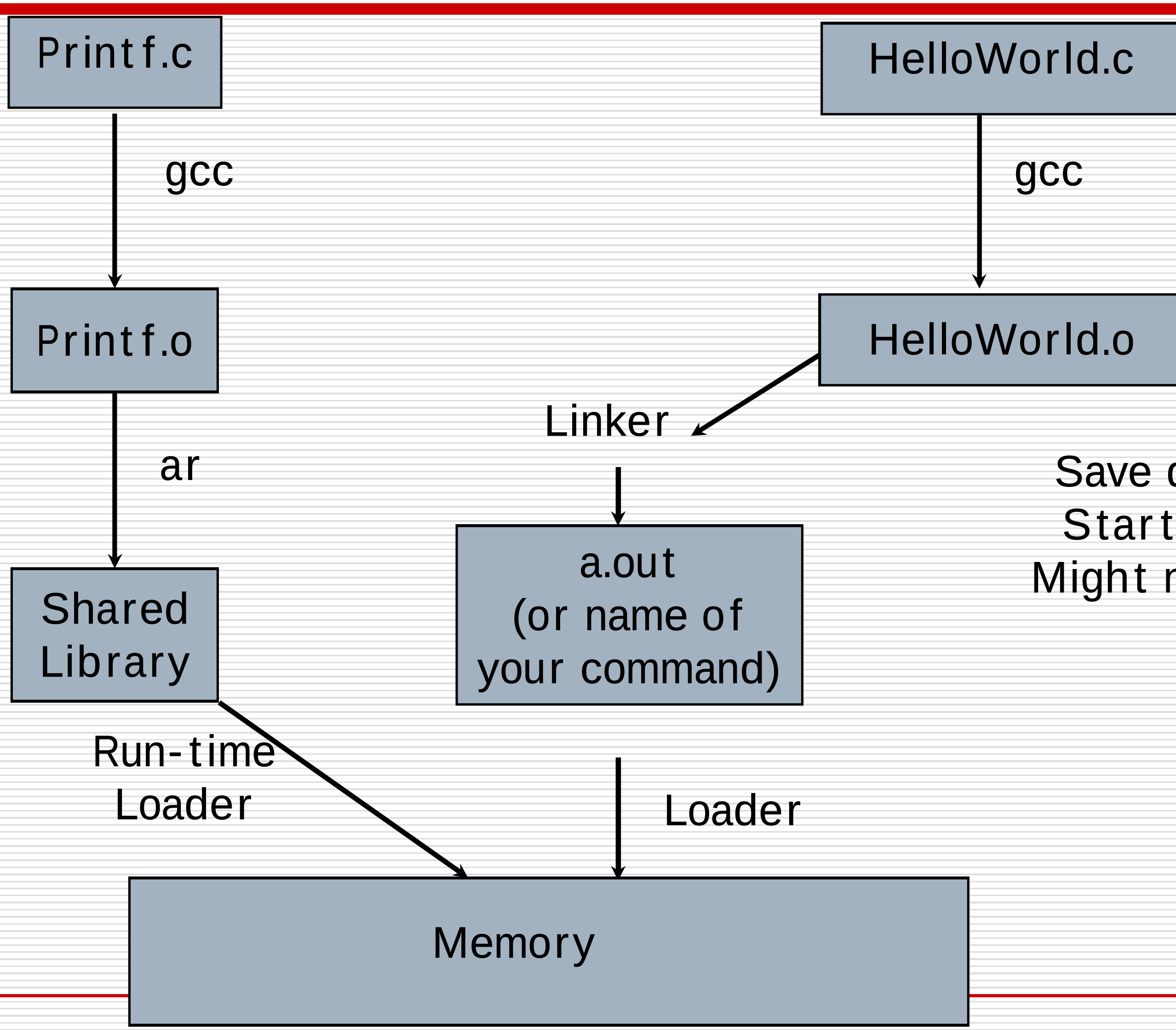
Shared libraries (continued)

- Libraries designated as "shared"
    - .so, .dll, etc.
    - Supported by corresponding ".a" libraries containing symbol information
- Linker sets up symbols to be resolved at runtime
- Loader: Is library already in memory?
    - If so, *map* into new process space
        - "map," an operation to be defined
    - If not, load and then *map*

Dynamic Shared Libraries

- Static shared libraries requires address space pre-allocation
- Dynamic shared libraries – address binding at runtime
  - Code must be position independent
  - At runtime, references are resolved as
    - Library_relative_address + library_base_address

Run-time Linking/Loading



Save disk space.
Startup faster.
Might not need all.

Next Lecture

- Enjoy the Spring Break
- Memory Management

Lab3 Out !