Linking & Loading

CS-3013 Operating Systems
Hugh C. Lauer

(Slides include materials from
Slides include materials from *Modern Operating Systems*, 3rd ed., by Andrew Tanenbaum
and from *Operating System Concepts*, 7th ed., by Silbershatz, Galvin, & Gagne)

What happens to your program …

…after it is compiled, but before it can be run?

Executable files

- Every 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 located within your program.
    - List of *names* of things defined elsewhere that are used by your program, and where they are used.

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`**

Two-step operation

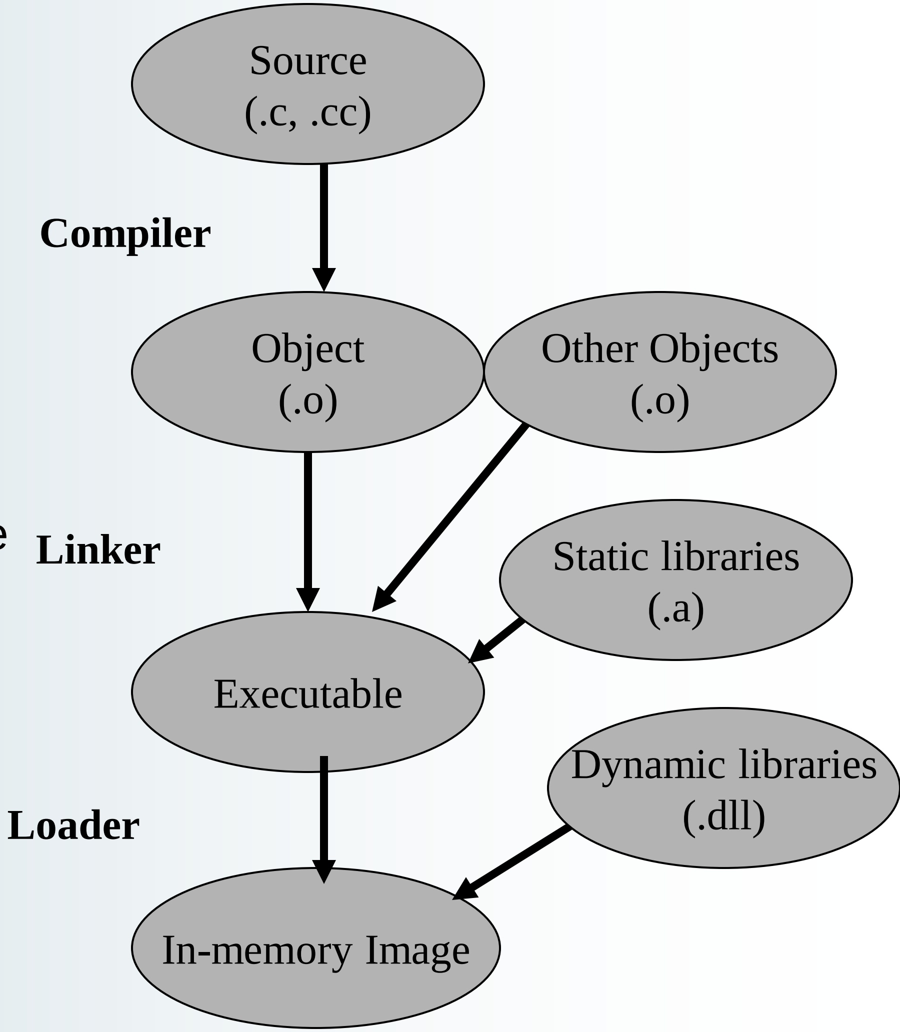
(in most systems)

- *Linking*: Combining a set of programs, including library routines, to create a *loadable image*
  a) Resolving symbols defined within the set
  b) Listing symbols needing to be resolved by loader

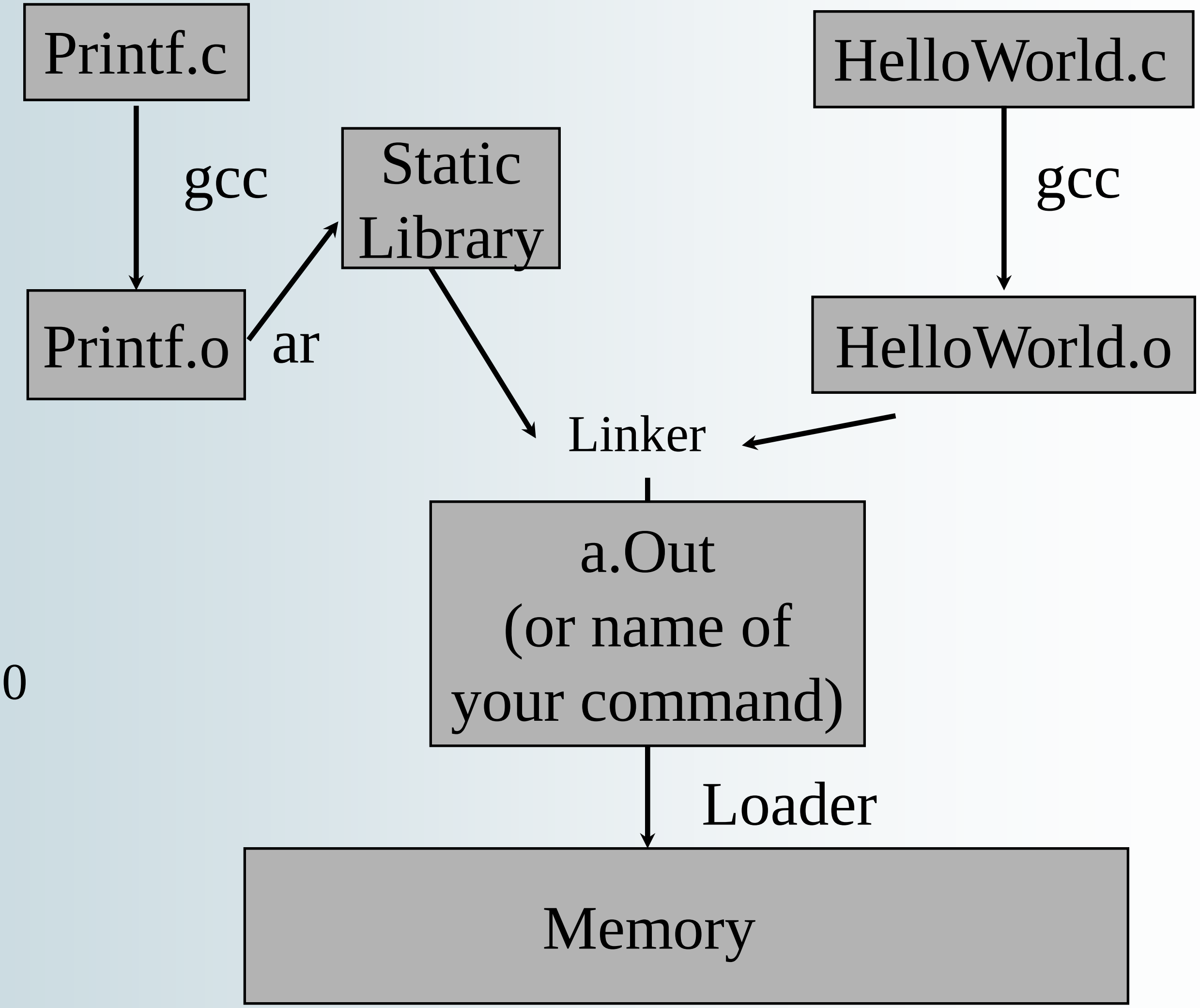
- *Loading:* Copying the loadable image into memory, connecting it with any other programs already loaded, and updating addresses as needed
  - (In Unix) interpreting file to initialize the process address space
  - (in all systems) kernel image is special (own format)

From source code to a process

- *Binding* is the act of connecting *names* to *addresses*
- Most compilers produce *relocatable object code*
  - Addresses relative to *zero*
- The linker combines multiple object files and library modules into a single executable file
  - Addresses also relative to *zero*
- The Loader reads the executable file
  - Allocates memory
  - Maps addresses within file to memory addresses
  - Resolves names of dynamic library items



Static Linking and Loading



See also Fig 1-30
in Tanenbaum

Classic Unix

- Linker lives inside of *cc* or *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 its own copy of common library routines
- Every loaded program contain duplicate copy of library routines

Dynamic Loading

- Routine is not loaded until it is called
- Better memory-space utilization; unused routines are never loaded.
- Useful when large amounts of code needed to handle infrequently occurring cases.

- Must be implemented through program design
  - Needs OS support to for loading on demand

Program-controlled Dynamic Loading

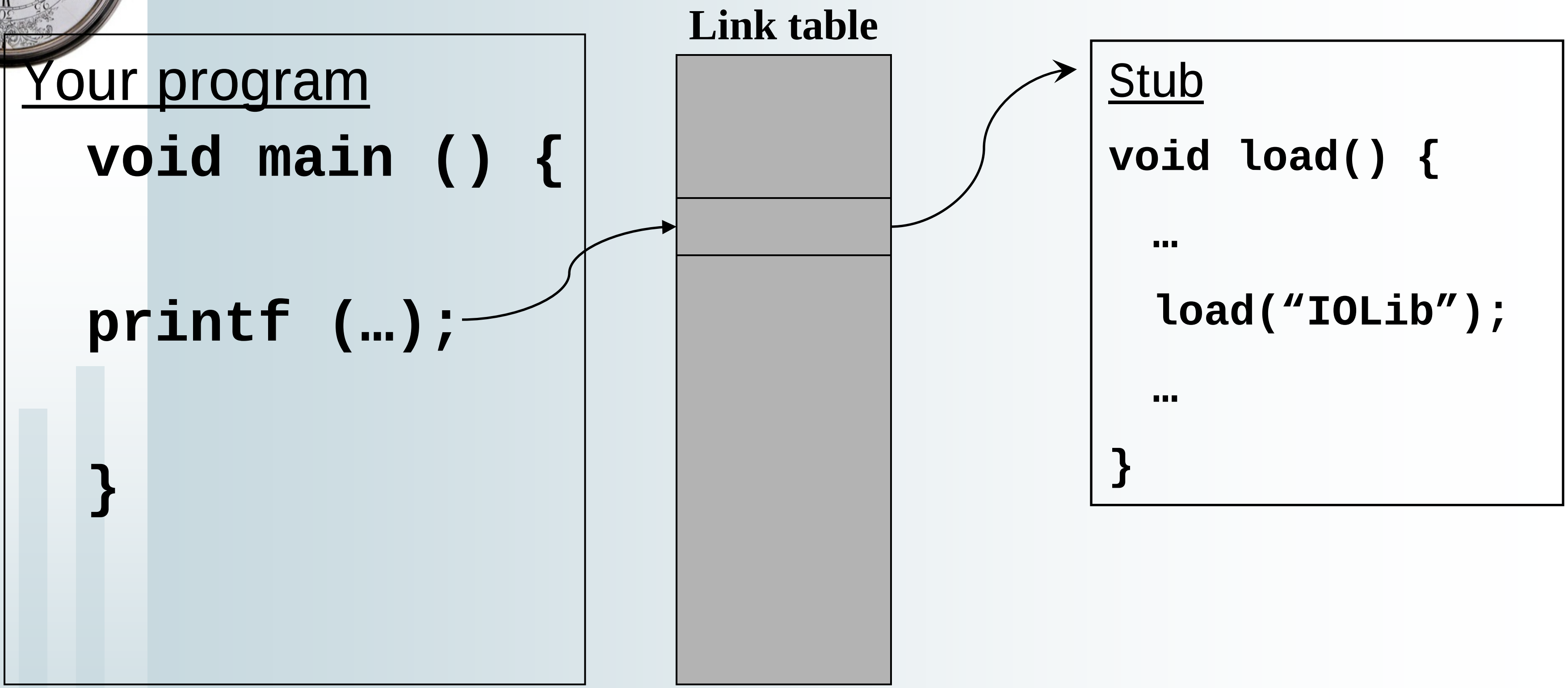
- Requires:
  - A *load* system call to invoke loader (not in classical Unix)
  - ability to leave symbols unresolved and resolve at run time (not in classical Unix)
- E.g.,

```
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 entry is initialized with address of small *stub* of code to locate and load module.
  - When loaded, loader replaces link table entry with address of loaded function
  - When unloaded, loader restores table entry with stub address
  - Works only for *function calls*, not *static data*

Example – Linker-assisted loading (before)



Link table

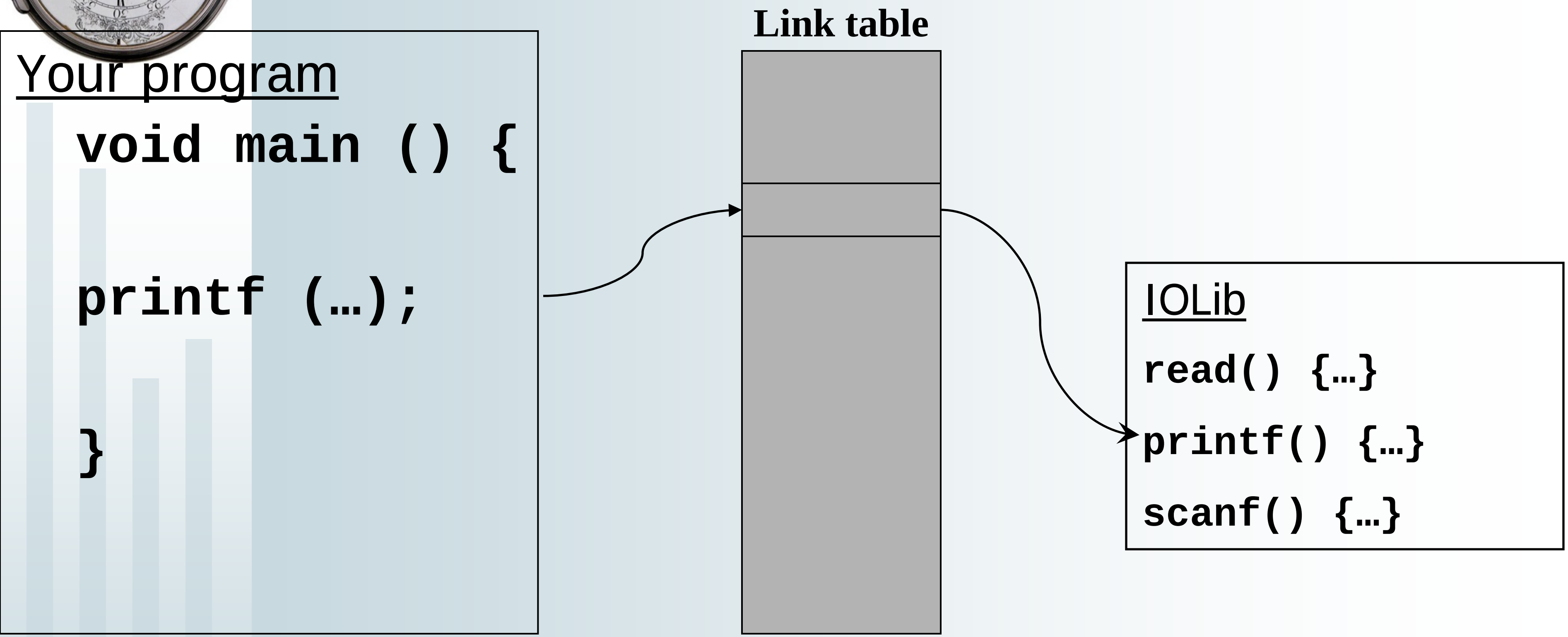
Your program

```
void main () {

printf (…);

}
```

Stub

```
void load() {
  …
  load("IOLib");
  …
}
```

Example – Linker-assisted loading (after)



Your program

```
void main () {

printf (…);

}
```

Link table

IOLib

```
read() {…}
printf() {…}
scanf() {…}
```

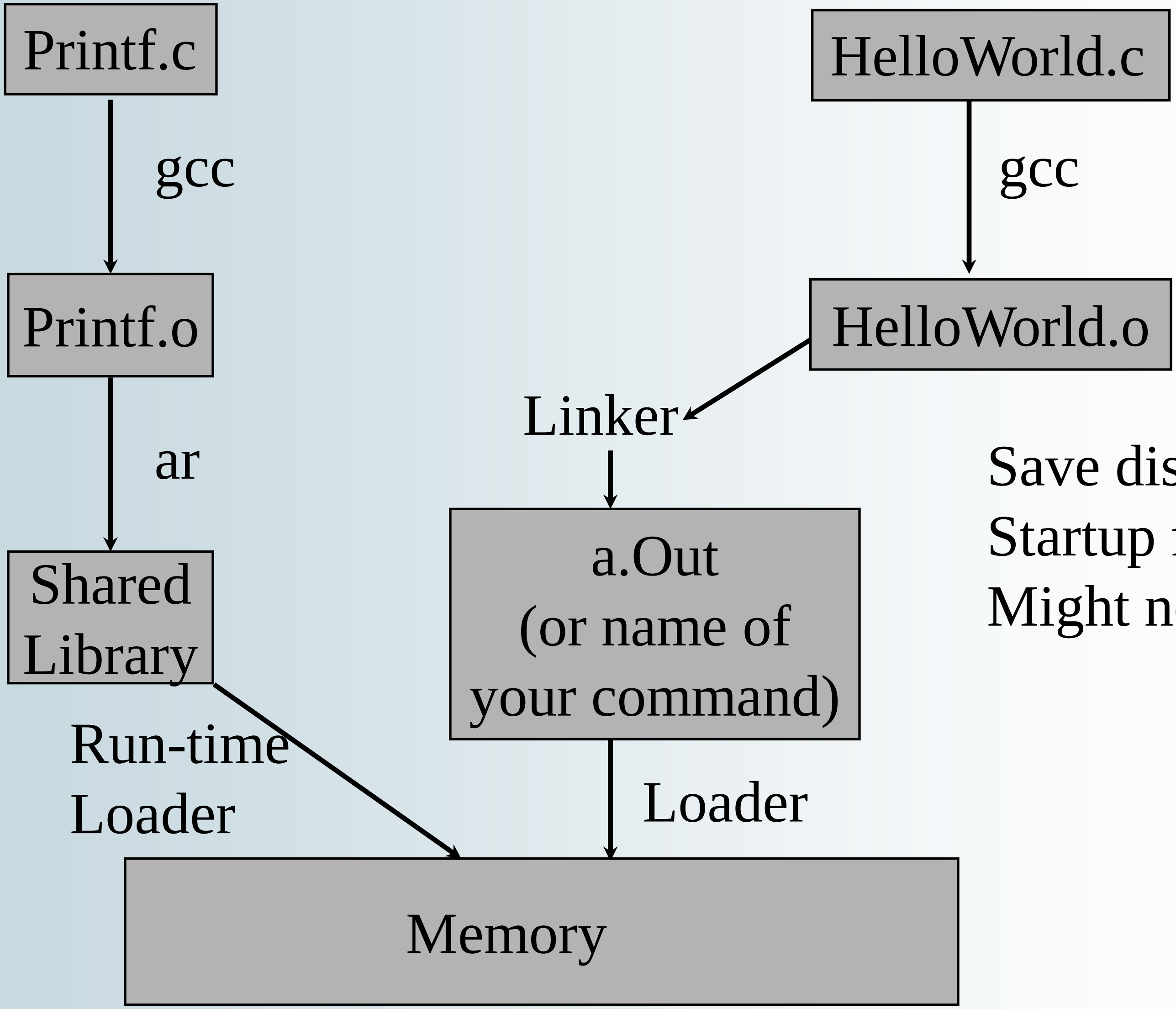
Shared Libraries

- Observation – “everyone” links to standard libraries (*libc.a*, etc.)
- These 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 yes, *map* into new process space
    - “map,” an operation to be defined later in course
  - If not, load and then *map*

Run-time Linking/Loading



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

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 address space of process
- Dynamic linking is particularly useful for libraries.

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
- See Tanenbaum, §3.5.6

Linking – Summary

- Linker – key part of OS – not in kernel
  - Combines object files and libraries into a “standard” format that the OS loader can interpret
  - Resolves references and does static relocation of addresses
  - Creates information for loader to complete binding process
  - Supports dynamic shared libraries

Loader

- An integral part of the OS
- Resolves addresses and symbols that could not be resolved at link-time
- May be small or large
  - Small: Classic Unix
  - Large: Linux, Windows XP, etc.
- May be invoke explicitly or implicitly
  - Explicitly by stub or by program itself
  - Implicitly as part of *exec*

Questions?