

Linking, Loading, Libraries



One-Slide Summary

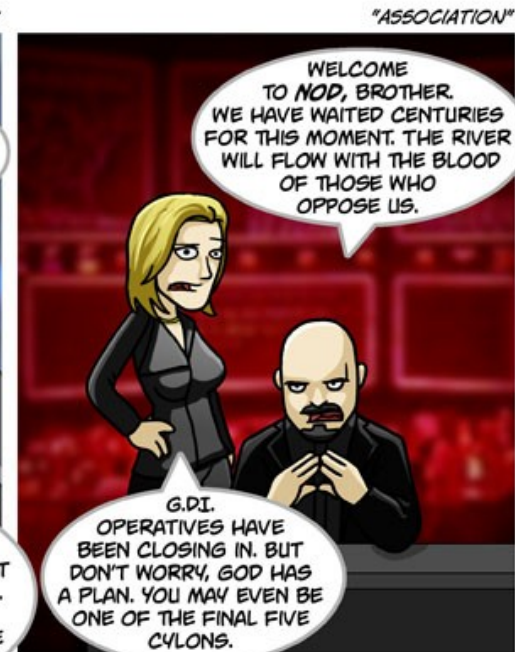
- We want **separate compilation** for program pieces. So we must **link** those compiled pieces together later. We must **resolve** references from one **object** to another.
- We also want to **share** libraries between programs.
- We also want to **typecheck** separately-compiled modules.

Course Goals

- At the end of this course, you will be acquainted with the fundamental **concepts in the design and implementation** of high-level programming languages. In particular, you will understand the **theory and practice of lexing, parsing, semantic analysis, and code interpretation**. You will also have gained practical experience programming in multiple **different languages**.

Lecture Outline

- Object Files
- Linking
- Relocations
- Shared Libraries
- Type Checking



Separate Compilation

- Compile different parts of your program at different times
- And then **link** them together later
- This is a big win
 - Faster compile times on small changes
 - Software Engineering (modularity)
 - Independently develop different parts (libraries)
- All major languages and all big projects use this

Pieces

- A compiled program fragment is called an **object file**
- An object file contains
 - Code (for methods, etc.)
 - Variables (e.g., values for global variables)
 - Debugging information
 - References to code and data that appear elsewhere (e.g., printf)
 - **Tables** for organizing the above
- Object files are implicit for interpreters

Two Big Tasks

- The operating system uses **virtual memory** so every program starts at a standard [virtual] address (e.g., address 0)
- **Linking** involves two tasks
 - **Relocating** the code and data from each object file to a particular fixed virtual address
 - **Resolving references** (e.g., to variable locations or jump-target labels) so that they point to concrete and correct virtual addresses in the New World Order



Relocatable Object Files

- For this to work, a **relocatable object file** comes equipped with three **tables**
 - **Import Table**: points to places in the code where an external **symbol** (variable or method) is referenced
 - List of (external_symbol_name, where_in_code) pairs
 - One external_symbol_name may come up **many times!**
 - **Export Table**: points to symbol definitions in the code that are exported for use by others
 - List of (internal_symbol_name, where_in_code) pairs
 - **Relocation Table**: points to places in the code where local symbols are referenced
 - List of (internal_symbol_name, where_in_code) pairs
 - One internal_symbol may come up **many times!**

FROM THE MAKERS OF THE BLOGOSPHERE,
BLOGOCUBE, AND BLOGODROME COMES

the Blogofractal

So Many Tables

- Tables must contain quite a bit of information
- Tables must also be easy to understand
- Let's see some examples ...



C/Asm Example

- Consider this program:

```
extern double sqrt(double x);
```

```
static double temp = 0.0;
```

```
double quadratic(double a, b, c) {  
    temp = b*b - 4.0*a*c;  
    if (temp >= 0.0) { goto has_roots; }  
    throw Invalid_Argument;  
has_roots:  
    return (-b + sqrt(temp)) / (2.0*a);  
}
```

Imports

```
extern double sqrt(double x);
```

```
static double temp = 0.0;
```

```
double quadratic(double a, b, c) {  
    temp = b*b - 4.0*a*c;  
    if (temp >= 0.0) { goto has_roots; }  
    throw Invalid_Argument;
```

```
has_roots:
```

```
    return (-b + sqrt(temp)) / (2.0*a);  
}
```

0x1000	...
0x1004	push r1
0x1008	call loc _{sqrt}

Import Table:
Replace address used at 0x1008
with final location of sqrt.

Exports

```
extern double sqrt(double x);
```

```
static double temp = 0.0;
```

```
double quadratic(double a, b, c) {  
    temp = b*b - 4.0*a*c;  
    if (temp >= 0.0) { goto has_roots; }  
    throw Invalid_Argument;  
has_roots:  
    return (-b + sqrt(temp)) / (2.0*a);  
}
```

0x0200	r1 = b
0x0204	r1 = r1 * r1
0x0208	r2 = 4.0
0x020c	r2 = r2 * a

Export Table:

We provide `quadratic`. If anyone else wants it, they can figure out where `0x0200` is finally relocated to. Call that new location `R`. They then replace all of their references to `locquadratic` with `R`.

(Internal) Relocations

```
extern double sqrt(double x);
```

```
static double temp = 0.0;
```

```
double quadratic(double a, b, c) {  
    temp = b*b - 4.0*a*c;  
    if (temp >= 0.0) { goto has_roots; }  
    throw Invalid_Argument;
```

```
has_roots:
```

```
    return (-b + sqrt(temp)) / (2.0*  
}
```

0x0600	r1 = ld loc_{temp}
0x0604	jgz r1 loc_{has{roots}}

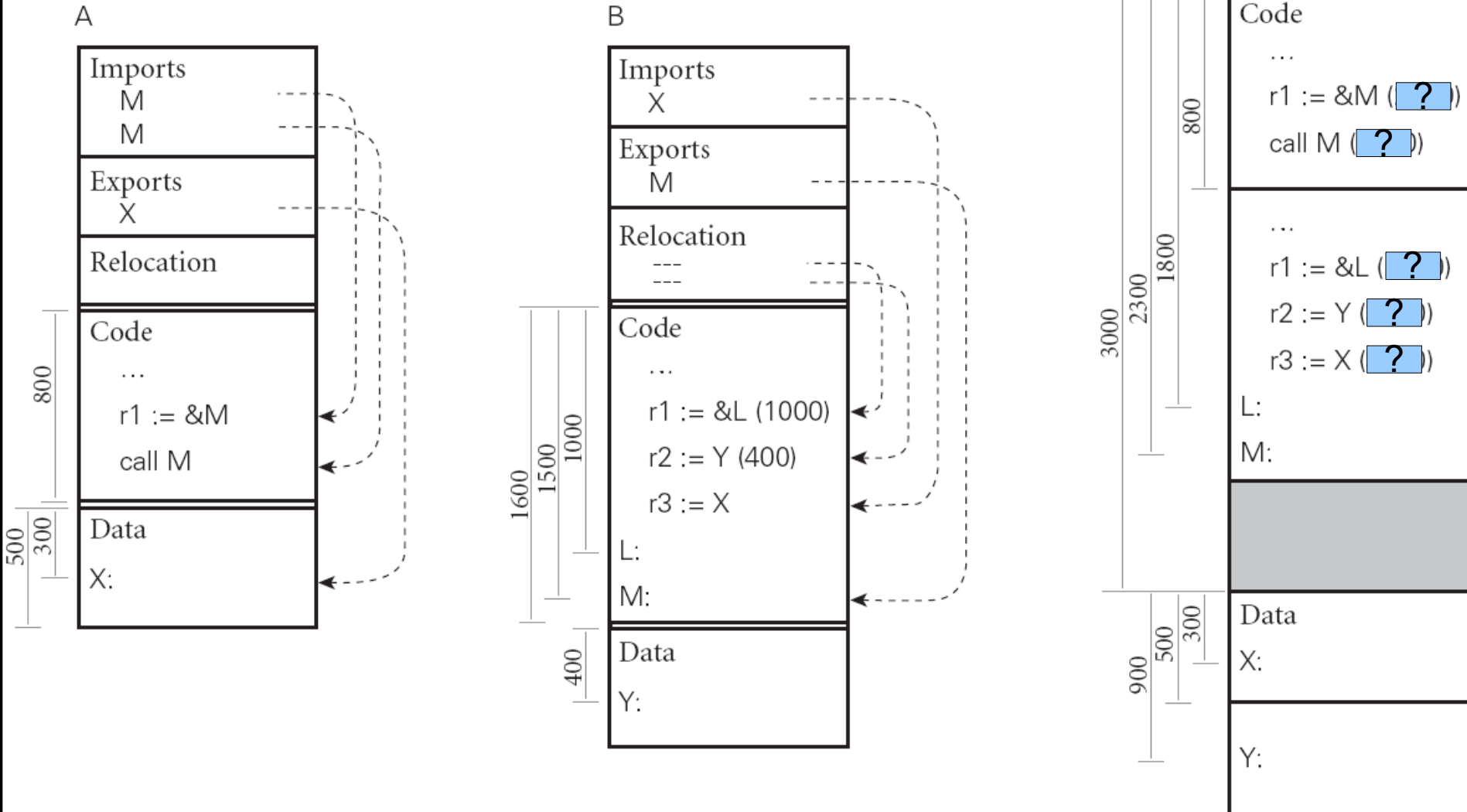
Relocation Table:
Find final relocated address of temp. Call that R_{temp} . Find final relocated address of 0x0600. Call that R_{0x0600} . Replace address referenced at R_{0x0600} with R_{temp} .

Big Linking Example

get out some paper!

Relocatable object files

Executable object file

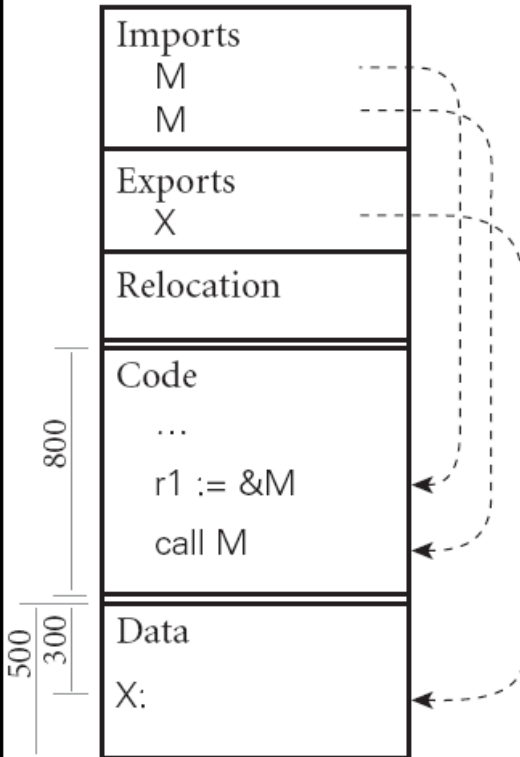


Big Linking Example Answers

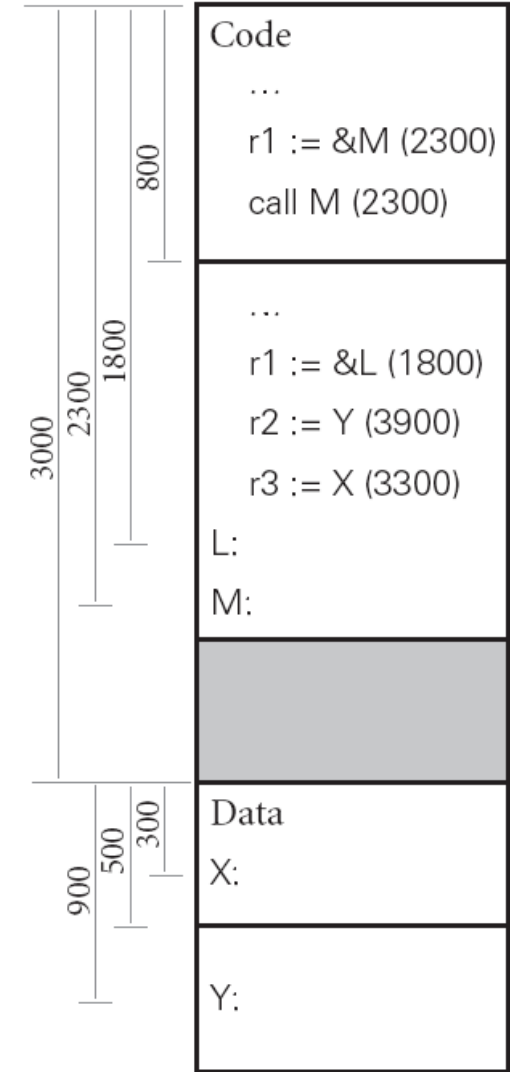
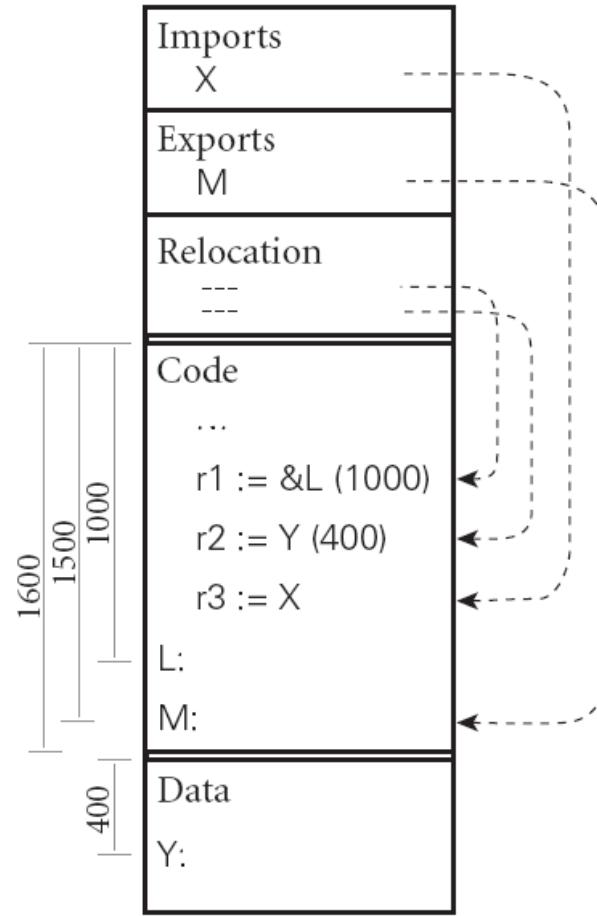
Relocatable object files

Executable object file

A



B



Summary

- Your relocatable object file: main.o
 - Exports main(), imports sqrt(), relocations ...
- Your math library: math.o
 - Exports sqrt(), relocations
 - Libraries **can have imports**: give an example!
 - In Unix, math.o lives in libmath.a and -lmath on the command line will find it
- The linker reads them in, picks a fixed final relocation address for all code and data (1st pass) and then goes through and modifies every instruction with a symbol reference (2nd pass)

Q: Theatre (002 / 842)

- What is Jean Valjean's prisoner number in **Les Miserables**?

Q: TV (051 / 842)

- This 1993 television series typically begins with the phrase "*The Truth Is Out There*".

Q: TV (089 / 842)

- Name any two of the five "Satellite Of Love" characters mentioned by name in the first-season opening theme song to **Mystery Science Theatre: 3000.**

Q: Movies (393 / 842)

- This phrase is Swahilian for "there are no concerns here" and was popularized by a 1994 Hamlet-like, mammal-centric Disney film.

Something Missing?

- That was fine, but if two programs both use math.o they will each get a copy of it



Are We Done?

- That was fine, but if two programs both use `math.o` they will each get a copy of it
 - You can optimize this a bit by only linking and copying in the parts of a library that you really need (transitive closure of dependencies), but that's just a band-aid
- If we run both programs we will load both copies of `math.o` into memory - wasting memory (recall: they're identical)!
- How could we go about sharing `math.o`?

Dynamic Linking

- Idea: **shared libraries** (.so) or **dynamically linked libraries** (.dll) use virtual memory so that multiple programs can share the same libraries in main memory
 - Load the library into physical memory *once*
 - Each program using it has a virtual address V that points to it
 - During **dynamic linking**, resolve references to library symbols using that virtual address V
- **What could go wrong? Code? Security?**

Relocations In The DLL

- Since we are sharing the code to math.dll, we **cannot** set its relocations separately for each client
- So if math.dll has a jump to $\text{loc}_{\text{math_label}}$, that must be resolved to the *same location* (e.g., 0x1234) for *all clients*
 - Because we can only patch the instruction once!
 - And every thread/program shares that patched code!
- So either:
 - Every program using math.dll agrees to put it at virtual address location 0x1000 (*problems? Unix SVR3 ...*)
 - math.dll uses *no* relocations in its *code* segment (*how?*)

Position-Independent Code

- Rather than “0x1000: jump to 0x1060”, use “jump to PC+0x60”
 - This code can be relocated to any address
 - This is called **position-independent code** (PIC)
- OK, that works for branches.
- But what about **global variables**?
 - **You tell me:**
 - Where should they live?
 - Should they be shared?

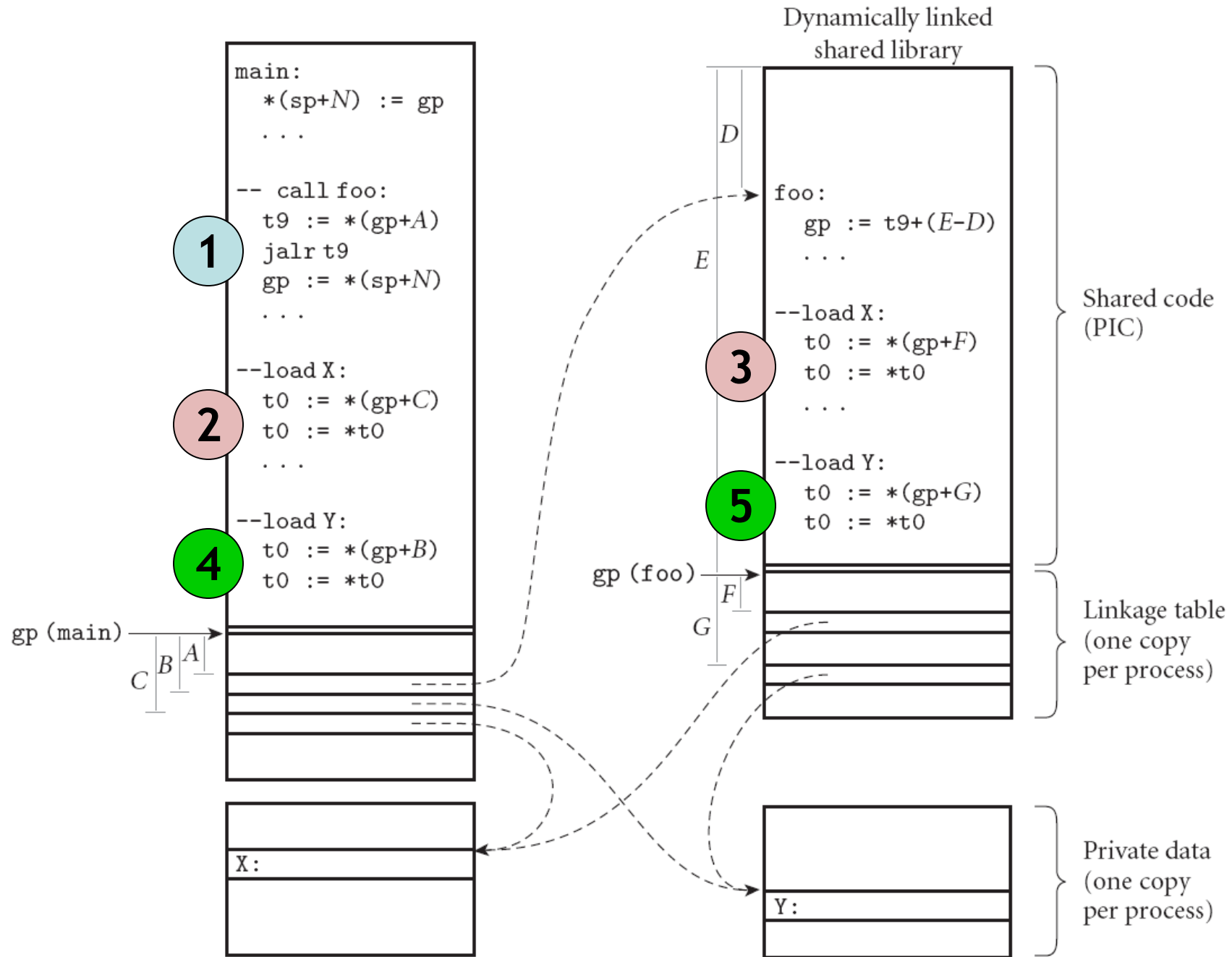
Data Linkage Table

- Store shared-library global variable addresses starting at some virtual address B
 - This table of addresses is the **linkage table**
- Compile the PIC assuming that register 5 (or GP or ...) will hold the current value of B
 - **Problems?**
- The entry point to a shared library (or the caller) sets register GP to hold B
 - Optimization: if the code and data live at fixed offsets, can do e.g. $GP = ((PC \ \& \ 0xFF00) + 0x0100)$

Shared Library = Shared Data?

- Typically each client of a shared library X wants its **own copies** of X's globals
 - Example: **errno** variable in libc (cf. Exceptions lecture)
- When dynamically linking, you share the code segment but get your own copy of the data segment
 - And thus your own base address B to put in GP
 - Optimization: use copy-on-write virtual memory
- Detail: use an **extra level of indirection** when the PIC shared library code does **callbacks** to unshared `main()` or references global variables from unshared `main()`
 - Allows the unshared non-PIC target address to be kept in the data segment, which is private to each program

Not As Bad As It Looks



Fully Dynamic Linking

- So far this is all happening at load time when you start the program
- Could we do it at run-time *on demand*?
 - Decrease load times with many libraries
 - Support dynamically-loaded code (e.g., Java)
 - Important for scripting languages
- Use linkage table as before
 - But instead loading the code for foo(), point to a special **stub** procedure that loads foo() and all variables from the library and then updates the linkage table to point to the newly-loaded foo()

Type Checking

- So we have separate compilation
- What's wrong with this picture?

(* Main *)

```
extern string sqrt();  
void main() {  
    string str = sqrt();  
    printf(“%s\n”,str);  
    return;  
}
```

(* math *)

```
export double  
sqrt(double a) {  
    return ...;  
}
```

Header or Interface Files

- When we type-check a piece of code we generate an **interface** file
 - Listing all exported methods *and their types*
 - Listing all exported globals *and their types*
 - The imp map and class map from PA4 suffice perfectly: just throw away the expression information
- When we compile a client of a library we check the interface file for the types of external symbols
 - *Can anything go wrong?*

Bait And Switch

- Write math.cl where sqrt() returns a **string**
- Generate interface file
- Give interface file to user
- Write new math.cl: sqrt() returns a **double**
- Compile source to relocatable object file
- Give object file to user
- ...
- Profit!



Checksums and Name Mangling

- From the interface file, take all of the exported symbols and all of their types and write them down in a list, then **hash** (or **checksum**) it
- Include hash value in relocatable object
- Each library client also computes the hash value based on the interface it was given
- At link time, *check to make sure* the hash values are the same
 - C++ **name mangling** is the same idea, but done on a per symbol basis (rather than a per-interface basis)

I'll Form The Head

- We wanted **separate compilation** for program pieces. So we must **link** those compiled pieces together later. We must **resolve** references from one **object** to another.
- We also wanted to **share** libraries between programs.
- We also wanted to **typecheck** separately-compiled modules.



Homework

- Midterm 2 - Wednesday April 18th
- WA7 Due Monday April 16
- PA5c Due Monday April 16