CSC 526 - Principles of Compiler Design II

Linking and Loading

Outline

- Object files
- Linking
- Relocations
- Shared libraries
- Type checking

Separate Compilation

- Compile different parts of your program at different times
- Then, link them together later
- This is a good thing
  - Faster compile times on small changes
  - Software engineering (modularity)
  - Independently develop different parts (libraries)
- All major languages and big projects use this concept

Pieces

- A compiled program fragment is called an object file
- An object file contains:
  - Code
  - Variables
  - Debugging information
  - References to code and data that appear elsewhere
  - 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
- Linking involves tow tasks:
  - Relocating the code and data from each object file to a particular fixed virtual address
  - Resolving references so that they point to concrete and correct virtual addresses

Relocatable Object Files

- For this to work, a relocatable object file has three tables:
  - Import table: points to places in the code where an external symbol is referenced
  - Export table: points to symbol definitions in the code that are exported for use by others
  - Relocation table: points to places in the code where local symbols are referenced

Example

```
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);
}
```

Example

- Import table: the call to `sqrt` on line 12 needs to use the address of final location of `sqrt`
- Export table: The `quadratic` function is exported, so client code can patch the symbolic labels with the concrete address once it is known
- Relocation table: references to the location of the temp variable declared on line 3 needs to be replaced with the concrete address. This also needs to be done for the `has_roots` label.

Considerations

- If two programs both use `math.o`, then they will each get a copy of it
- If we run both programs we will load both copies of `math.o` into memory – this is wasteful because both copies are identical
- Can we share `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
  - Inspect globals, etc.

Relocations in Shared Libraries

- Since we are sharing the code to `math.so`, we cannot set its relocations separately for each client
- If `math.so` has a jump to a label, that must be resolved to the same location for all clients
  - We can only patch the instruction once
  - Every thread/program shares that patched code
- Idea: instead of say, "jump to 0x1060", use "jump to PC+0x60"
  - This code can be relocated to any address
  - This is called position-independent code (PIC)

Data Linkage Table

- Store shared-library global variable addresses starting at some virtual address A
- Compile the PIC assuming that some register will hold the current value of A
- The entry point to a shared library (or the caller) sets the register to hold A

Shared Data

- Typically each client of a shared library X wants its own copies of X's global data
- When dynamically linking, the code segment is shared, but the data segment is copied
- 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()`
    - This allows the unshared non-PIC target address to be kept in the data segment, which is private to each program

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
  - Important for scripting languages
- Use the linkage table as before
  - Instead of loading the for for, say `foo()`, point to a special stub procedure that loads `foo()` and all the variables from the library and then updates the linkage table to point to the newly-loaded `foo()`

Type Checking

- Is this a problem?
- File: `main.c`

```
extern string sqrt();
int main() {
    string str = sqrt();
    printf("%s\n", str);
    return 0;
}
```

- File: `math.c`

```
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 functions and their types
  - Listing all exported globals and their types
- When we compile a client of a library, we check the interface file for the types of external symbols
- Can anything go wrong?

Checksums and Name Mangling

- Take all of the exported symbols and all of their types from the interface file and store them in a list, then perform a hash (or checksum)
- Include the hash value in the 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 performed on a per symbol basis, rather than a per interface basis

Summary

- We want separate compilation for program pieces, so we must link those compiled pieces together later.
- We need to resolve references from one object to another
- We also want to share libraries between programs
- We also want to type-check separately compiled modules