**CS 153**

**Design of Operating Systems**

**Ch. 2 Binding**

**2.1 Resolution**

*token* – something that represents something else: a *value*

*binding* – attaching a value to a token

*resolving* – looking up a token’s value

*unbound* – a token with no binding

Caching *tightens* the binding.

Six binding times for caching bindings:

* compile
* link
* load
* pre-run access
* access
* never

The first four are anticipatory caching.

Most costly to modify: constants as immediate operands in the operand fields of instruction

**2.2 Compiling, Linking, and Loading**

**Programs and modules**

A program can consist of hundreds of modules (possibly shared with other programs).

A module defines a set of variables and functions

May be viewed as defining a class of servers whose attributes are the variables the module defines and whose service routines are the functions that the module defines.

**2.2.1 Compiling**

Input: module source file

Output: module object file

Contains:

* initialized-data sesgment containing the initial values of the module’s initialized internal variables
* number indicating the size of the data segment
* code-segment containing the machine code for the functions
* external-reference table; lists the variables and functions that the module imports
* table of contents; lists the module-relative address of each of the internal functions and global variables

**2.2.2 Static Linking**

*linker* – combines object files and resolves and caches their external references by their combined table of contents.

Incorporating an object file:

* Initialized variables: append to the end of the combined initialized-data segment
* Uninitialized variables: virtually append it to the end of the bss
* Functions: append the machine code to the end of the new code segment; each of the external references is added to the combined external-reference table

**2.2.3 Loading**

*loader* – handler for the system call for a program to run another. Creates an execution image and runs it.

In most operating systems, all executable images produced share a single memory-resident copy of the executable’s code segment.

The loader creates a data segment private to each execution image.

**2.2.4 Dynamic Loading/Linking of Unshared Modules**

We can trap attempts to use unresolved references and invoke the appropriate loader to dynamically link a module.

A process can also dynamically unload a module.

**Static vs. dynamic linking**

static linking – avoids overhead of resolving and caching external references each time an object file is loaded; however, statically linked copies can become obsolete when the module is updated

dynamic linking – will always get the updated version; saves disk space

**2.2.5 Dynamic Loading/Linking of Shared Modules**

There is a problem if two different programs share the same module. This is because the module’s code segment may be loaded into one program’s code segment at an address not available in the other

A single copy of a module’s code segment is projected into multiple code segments of different executable files and different locations in each one via *page remapping*.

Each process running one of the executables can store its copy of the module’s attribute record wherever it wants in its own data segment.

The module is compiled to position-independent code