Linking and Loading

When your program is more than a single .asm file...

**Linker**: links separate object codes into one.

1. Assign all text and data sections symbolic spots in address space.
2. Resolve symbolic labels to addresses.
3. Patch up absolute address references.

**Loader**: loads executable into memory.

1. Determine size of text and data segments.
2. Create address space large enough for both.
3. Copy instructions and data into memory.
4. Copy parameters to main program onto stack.
5. Initialize registers (including $sp).
6. Jump to main procedure.
Static vs. dynamic linking

Update flexibility:

Static: Getting library updates requires relinking.
Dynamic: Library updates require no changes to executable.*

Code size:

Static: all code and static data that might ever be called or used
  • must be included in the executable.
  • must be loaded at run-time.
Dynamic: external code is linked and loaded only if used.
  • may save static and dynamic work
  • moves some static work to run-time.

*assuming the API does not change.

First call to dynamically linked code

First call is to a dummy entry via an indirect jump

Identifying the desired routine

The loaded word points to code that loads a register with a number identifying the desired library routine

Dynamic linker/loader

The program jumps to the DLL that finds the correct routine, remaps it...

...and changes the address in the indirect jump to point to that routine
Dynamically linked libraries

Thereafter, the call to the library routine jumps indirectly to the routine without the extra hops.

Virtual Machine Model

Java bytecode

like assembly code for JVM, but works on all JVMs: hardware-independent

typed (unlike ASM)

strong JVM protections

Java bytecode

Holds pointer 'this'

Other arguments to method

Other local variables

variable table

operand stack

constant pool

* memory, structured as objects, not an array of bytes.