Program Translation

Program translation uses system translation in order to translate it to machine code.

- There are four general areas of memory in a process
- The text area contains the instructions for the application and is fixed in size
- The static data area is also fixed in size and contains
  - Global variables
  - Static local variables
  - String and sometimes floating-point constants
- The run-time stack
  - Contains activation records, each with an associated functions invocation
  - Saved levels of callee-saved registers
  - Local variables and arguments not allocated to registers
  - Space for the max words of arguments passed on stack to other functions
- The heap contains dynamically allocated data

1. Loads onto memory (in a format -> code space + operation space)
2. OS gives program a control, "launches process"

Name	Process
sp	Stack↓ ↑Dynamic Data/ Heap
gp	Static Data
pc	Text
Address 0	Resereved

- The stack starts are the high-end of memory growing downwards, while the head grows upwards into the same space
- The lower end of memory is reserved
- The text segment follows, housing MIPS machine code

If the stack and the Dynamic Data/Heap meet than we unfortunately have a segmentation fault

The Translation Process

- Preprocessing - handled by the compiler
- Compiling - translates to assembly file
- Assembling - translates to machine language
- Linking - creates an executable

- Loading - the executable is ran through this

The compiler depends on the programming language, the assembler depends on the chipset. It must be translated to meet the machine code of the chips type.

The linker finds functions, and other libraries. It "links" the library code into the program.

This eventually produces an executable. The loader must be invoked after to run the executable.

Compiling

- We use the term "compiling" as a general term to refer to the entire translation process from source file to executable
- Compilers are responsible for...
  - checking syntax
  - making semantic checks
  - performing optimizations to improve performance, code size, and energy use

Assembling

- Assemblers take an assembly file as input and produce an object file as output
- Assembling is accomplished in two passes (typically)
- First pass: Stores all identifiers representing address or values in a table as there can be forward reference
- Second pass: translates the instructions and data into bits for the object file

A symbol table is create. It is essentially a dictionary for the names and addresses. Any identifier created by the user and not the language is stored in here

The Object File

- An object file header describing the size and position of the other portions of the object file
- Text segment containing the machine instructions
- The Data Segment containing data values
- Relocation information identifying the list of instructions and data words that depend on the absolute address
- A symbol table containing global labels and associated addresses in object file and the list of unresolved references
- Debugging information to allow a symbolic debugger to associate machine language with the addresses of the source line and the variable name

Linking

- Linkers take the object file and object libraries as input and output executables

- Resolve any external references by finding symbols in another object or library
- Aborts if any external reference cannot be resolved
- Determines the address of absolute references using relocation information in the object files
- The executable has similar format to object files with no unresolved references or relocation information

Loading

The loader copies the executables file from disk

- reads the file header to determine its size of text and data segments
- allocates the address space for processing
- copies the instructions into the text segment and data into static data segment
- copies arguments passes to the program onto the stack
- initializes the machine registers the stack pointer
- jumps to a start-up routine that will call the main function