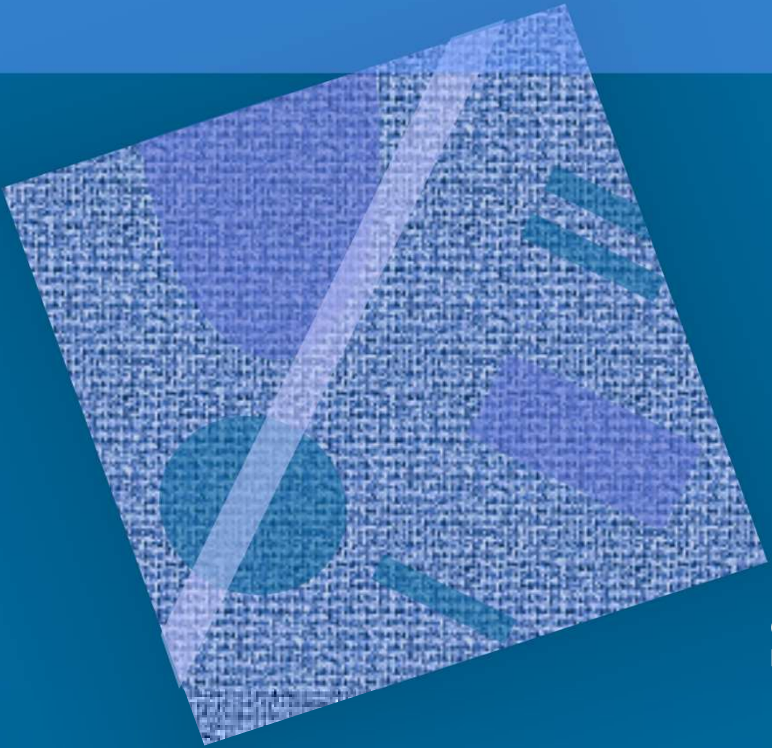
Lecture 10

Compilation, Linking and Loading



From program to process

Compilation unit

Compilation phases

Macros, literals

Static and dynamic linking

From High Level Language (HLL) Program to Executable Process

Compile from HLL

Compilation unit
HLL program module
addresses: symbolic

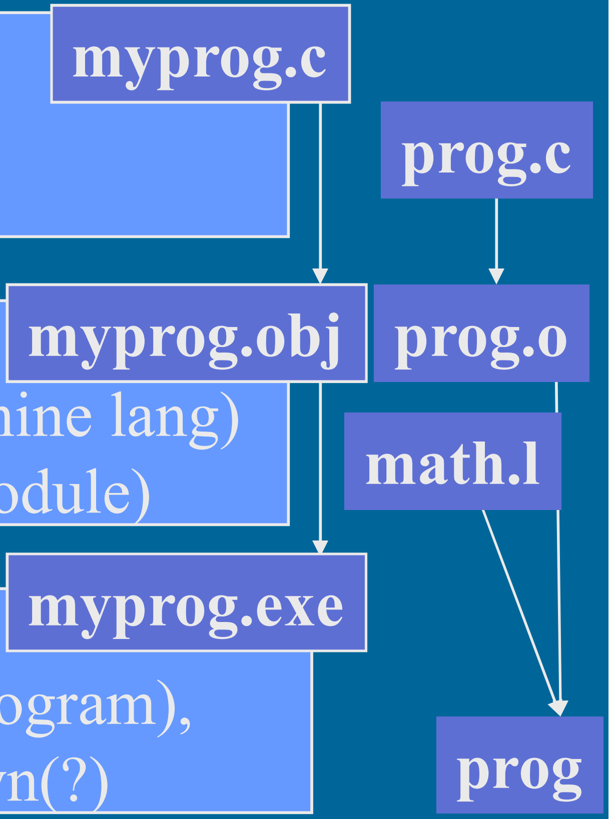
Link with other modules and library modules

Object module
Compiled module (machine lang)
addresses: linear (per module)

Load to memory as executable process

Run module
addresses: linear (per program),
partly unknown(?)

Process
Executable program
addresses: linear (virt. addr. space?)



Object Module

- Code in machine language
  - Memory references within module complete (in modules own linear address space)
  - References external to module are marked
- For linking:

uudelleensijoitustaulu

  - **Relocation table**. Info on those addresses, that must be updated when this module address space is linked to another
  - Info on references outside this module
  - Info on locations in this module that can be referenced to from outside
  - Symbol table

IMPORT

EXPORT

SYMBOL TABLE

Symbol table

- What is the value of each symbol?
  - Static value can (also) be memory address in modules current address space
- Compiler will generate, linker may update
- Sometimes kept up also after loading for smarter run time error messages
  - Software development environments keep up symbol table all the time
- Usually left out of finished product (program)
  - Takes space, not needed in normal execution

Macro

- Often repeated code sequence, helps programming
  - No environment, just code
- May include call-by-name parameters
- Processed before compilation
  - Part of symbolic assembly language or HLL
  - Not part of machine language
  - Used macro is replaced by its body
- Example usage
  - Subroutine prolog and epilog
  - Compiler macros, programmer's own macros
- Differences to subroutines
  - Use time (before compilation vs. execution time)
  - Call/return, amount of code, cost of use

Literals

- In HLL all large constants are literals
  - Compiler should prevent changing literal values
  - One should not pass literals as call-by-reference parameters
    - Subroutine could change its value?
- Some symbolic assembly languages have implicit (automatic) literal definitions
  - Easily writable/readable code
  - E.g., automatic space allocation to 234567

```
N := 35000;
```

```
tmp1 dc  35000
       load r1, tmp1
       store r1, N
```

```
var myStr = ”literal"
```

```
FortranX:  5 = 6;
```

```
LOAD  R1, six
STORE R1, five
```

???

Java string?

```
Load  R14, =F’234567’
```

Definition of symbol F’234567’:
F’234567’ ≡ “address of memory location with value 234567”

Assembly Language Compilation

- 0th pass – process macros – generate code from them
- 1st pass (of all code)

  koodin läpikäynti
  - Calculate space requirements for all code
  - Start to generate relocation tables (symbol table, etc)
- 2nd pass
  - Generate object module
  - Complete relocation tables
  - Give error messages
  - May be combined to 1st pass, but usually not
- 3rd pass
  - Code generation, code optimization
  - May be combined to 2nd pass
  - Print listing of program in symbolic assembly language

HLL Compilation (Translation)

BEGIN 123.45 IF (

- More phases
  - Search for syntactic elements
    - Generate and parse syntax tree
  - Recognize statements with syntax tree
  - Generate intermediate language (not always) IL
  - Code optimization
  - Code generation
    - Not (always) for Java-programs

(front end)

P-code, bytecode, …

Program in IL, symbol tables

(back end)

More info? → **Courses on Compilers and Programming Languages**

Linking

uudelleensijoitusvakio

- *Relocation constant (RC)*
  - A: 0, B: L, C: L+M
- Add RC to all local reference addresses
- Set references between modules correct
  - Consider the RC of referenced module

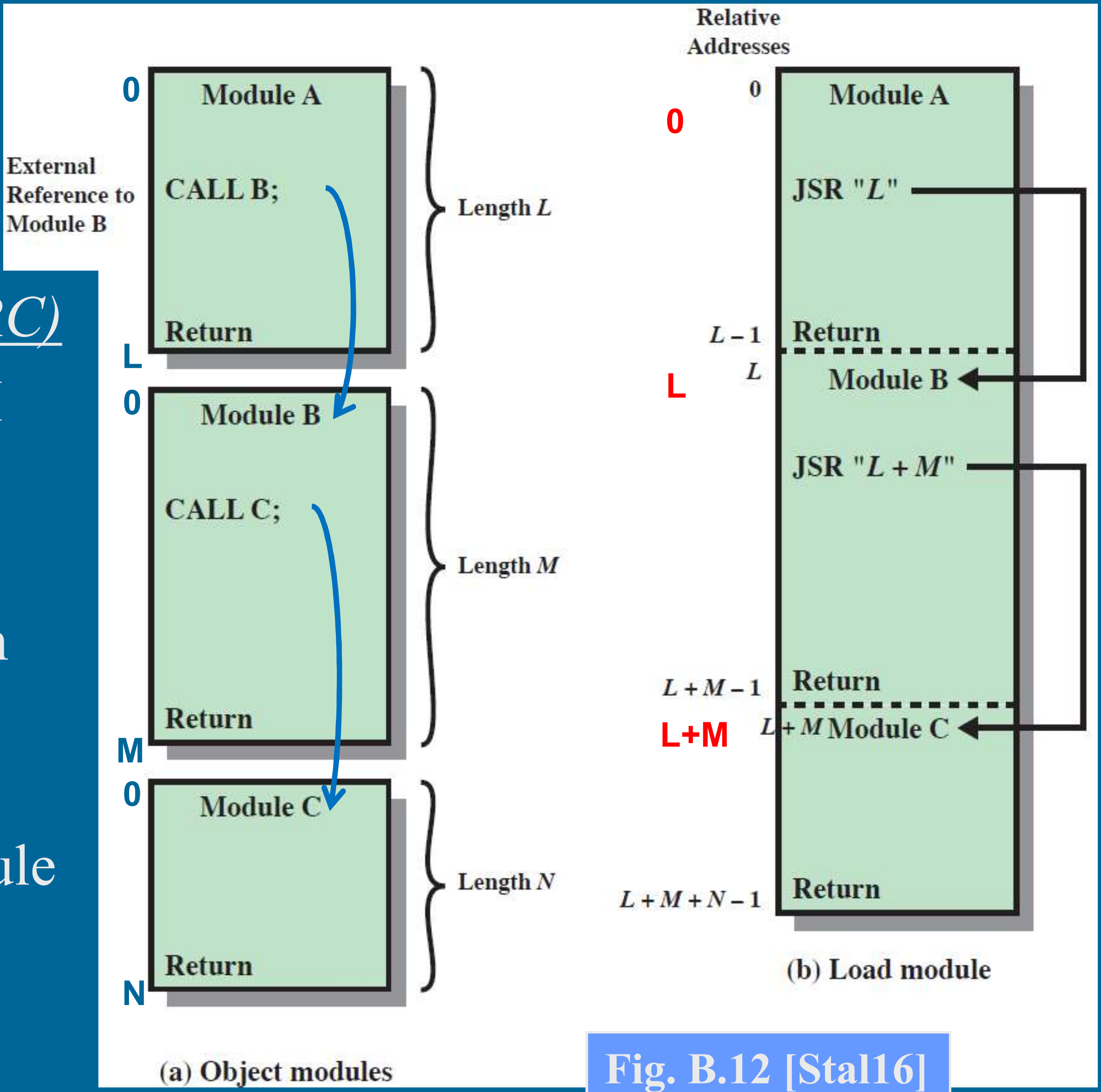


Fig. B.12 [Stal16]

Static and Dynamic Linking

- Static linking
  - All references to other modules and library modules are solved (linked) <u>before</u> loading (and <u>execution</u>)
  - Large load module
    - Includes modules that are never referenced during single execution
- Dynamic linking
  - Calls to dynamically linked modules are left open (not linked, unsolved)
  - Small load module, but possibly slow to run
  - All references to unsolved module is solved at run time
    - Pause execution
    - Link dynamically missing module
    - Continue
  - E.g., many OS libraries, game levels 5-30, …

Loading

- Load module is used to build executable process (Build PCB, allocate memory and other resources)
- Process code and data areas are copied to memory, and process is moved to Ready-to-Run queue
- Different types
  - Absolute – mem location remains static
  - Relocatable – change mem location some times (e.g., after being swapped out to disk)
    - When and how do you change mem ref addresses?
  - Dynamic – mem location changes dynamically at run time
    - When and how do you change mem ref addresses?

-- End --

Nobel 1956

- Transistor
  - J. Bardeen, W.B. Shockley ja W. Brattain, Bell Labs, 1948
  - TX-0, MIT, 1956
  - One of most important 20th century technology inventions in the world?

Nobel 2000

- Integrated circuit (no more wires!)
  - Jack Kilby, Texas Instruments, 1958
  - Robert Noyce, Fairchild Semiconductor, 1959
  - IBM S/360, 1964