Language Systems

Outline

- The classical sequence

- Variations on the classical sequence

- Binding times

- Debuggers

- Runtime support

The Classical Sequence

- Integrated development environments are wonderful, but…

- Old-fashioned, un-integrated systems make the steps involved in running a program more clear

- We will look the classical sequence of steps involved in running a program

- (The example is generic: details vary from machine to machine)

Creating

- The programmer uses an editor to create a text file containing the program

- A high-level language: machine independent

- This C-like example program calls **fred** 100 times, passing each **i** from 1 to 100:

```
int i;
void main() {
  for (i=1; i<=100; i++)
    fred(i);
}
```

Compiling

- Compiler translates to assembly language
- Machine-specific
- Each line represents either a piece of data, or a single machine-level instruction
- Programs used to be written directly in assembly language, before Fortran (1957)
- Now used directly only when the compiler does not do what you want, which is rare

```
int i;
void main() {
  for (i=1; i<=100; i++)
    fred(i);
}
```
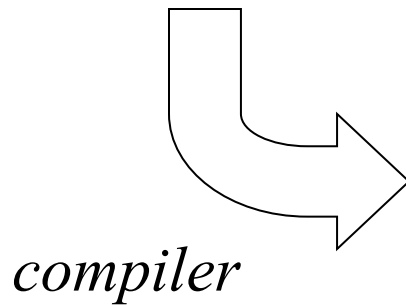
compiler

```
i:      data word 0
main:   move 1 to i
t1:     compare i with 100
        jump to t2 if greater
        push i
        call fred
        add 1 to i
        go to t1
t2:     return
```

Assembling

■ Assembly language is still not directly executable

    – Still text format, readable by people

    – Still has names, not memory addresses

■ Assembler converts each assembly-language instruction into the machine's binary format: its *machine language*

■ Resulting object file not readable by people

```
i:      data word 0
main:   move 1 to i
t1:     compare i with 100
        jump to t2 if greater
        push i
        call fred
        add 1 to i
        go to t1
t2:     return
```
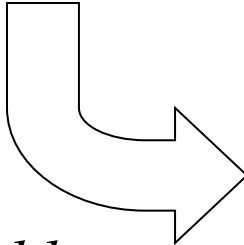
assembler

i: | 0 |

main:
| ~~xxxx~~ i |
| ~~xx~~ i ~~x~~ |
| ~~xxxxxx~~ |
| ~~xxxx~~ i |
| ~~x~~ fred |
| ~~xxxx~~ i |
| ~~xxxxxx~~ |
| ~~xxxxxx~~ |

Linking

■ Object file *still* not directly executable

   – Missing some parts

   – Still has some names

   – Mostly machine language, but not entirely

■ Linker collects and combines all the different parts

■ In our example, **fred** was compiled separately, and may even have been written in a different high-level language

■ Result is the executable file

i: 0

main:
| xxxx i |
| xx i x |
| xxxxxx |
| xxxx i |
| x fred |
| xxxx i |
| xxxxxx |
| xxxxxx |

linker

i: 0

main:
| xxxx i |
| xx i x |
| xxxxxx |
| xxxx i |
| x fred |
| xxxx i |
| xxxxxx |
| xxxxxx |

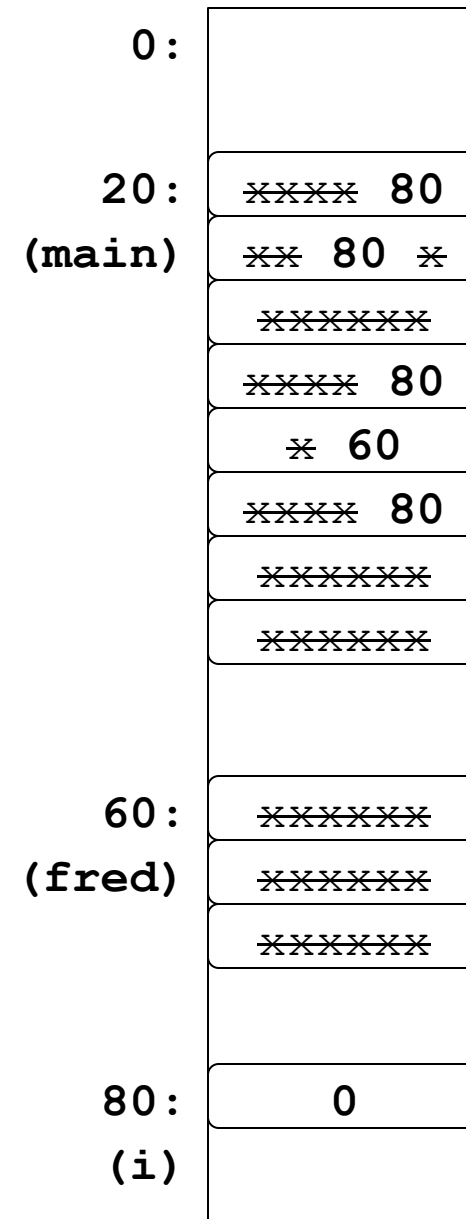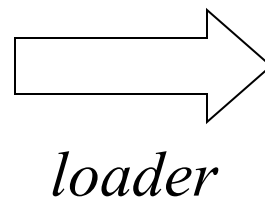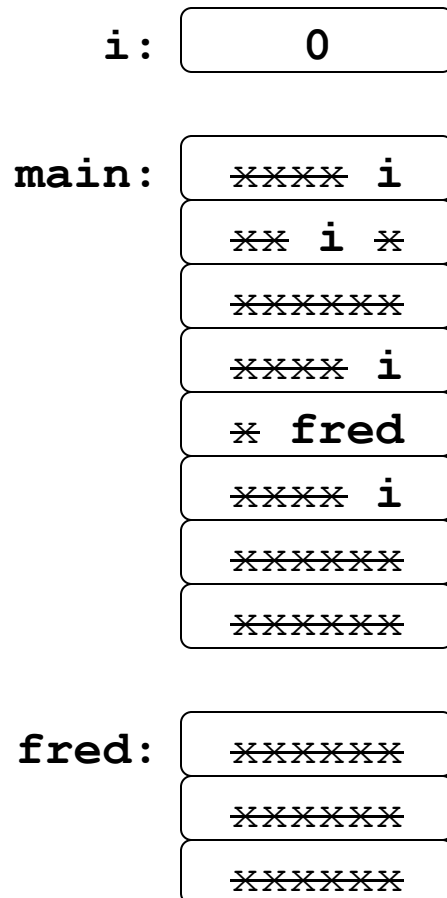fred:
| xxxxxx |
| xxxxxx |
| xxxxxx |
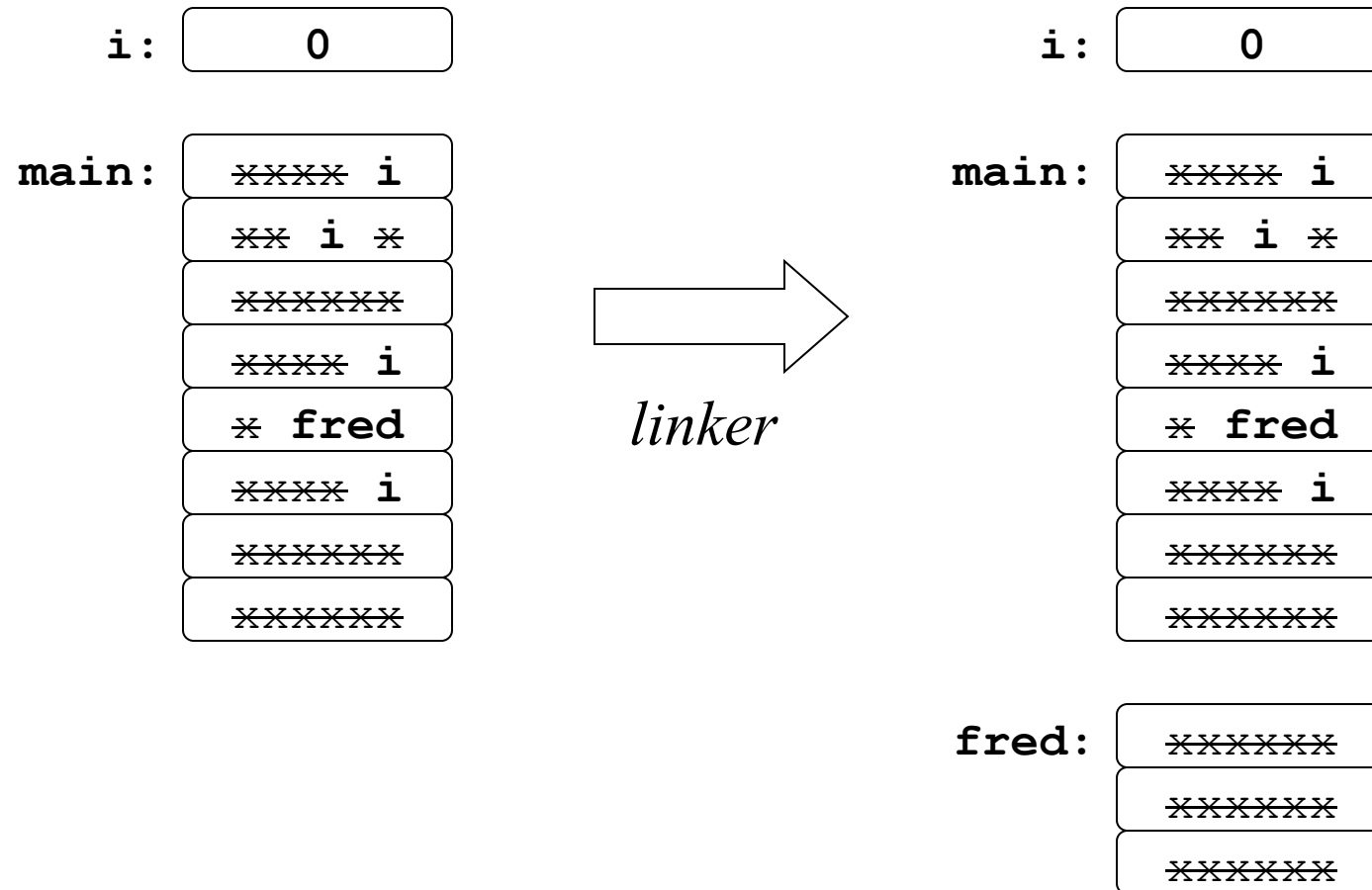
Loading

- "Executable" file *still* not directly executable
  - Still has some names
  - Mostly machine language, but not entirely
- Final step: when the program is run, the loader loads it into memory and replaces names with addresses
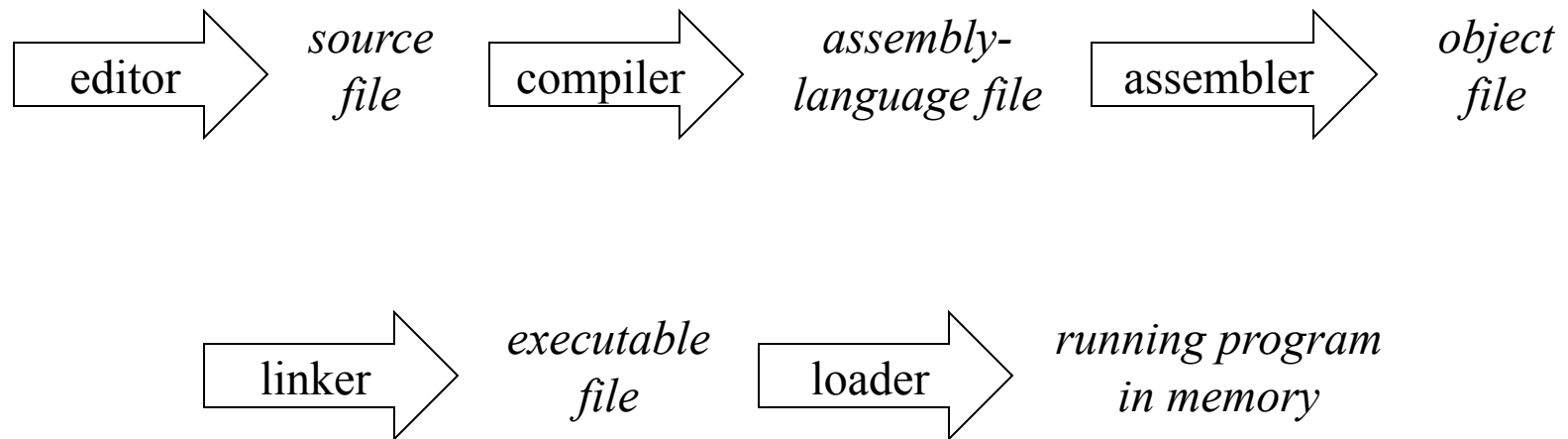
A Word About Memory

- For our example, we are assuming a very simple kind of memory architecture

- Memory organized as an array of bytes

- Index of each byte in this array is its *address*

- Before loading, language system does not know where in this array the program will be placed

- Loader finds an address for every piece and replaces names with addresses

i: [0]

main: [~~xxxx~~ i]
[~~xx~~ i ~~x~~]
[~~xxxxxx~~]
[~~xxxx~~ i]
[~~x~~ **fred**]
[~~xxxx~~ i]
[~~xxxxxx~~]
[~~xxxxxx~~]

fred: [~~xxxxxx~~]
[~~xxxxxx~~]
[~~xxxxxx~~]

loader

0: [ ]

20: [~~xxxx~~ 80]
(main) [~~xx~~ 80 ~~x~~]
[~~xxxxxx~~]
[~~xxxx~~ 80]
[~~x~~ 60]
[~~xxxx~~ 80]
[~~xxxxxx~~]
[~~xxxxxx~~]

60: [~~xxxxxx~~]
(fred) [~~xxxxxx~~]
[~~xxxxxx~~]

80: [0]
(i)

Running

■ After loading, the program is entirely machine language

  – All names have been replaced with memory addresses

■ Processor begins executing its instructions, and the program runs

The Classical Sequence

editor → *source file* → compiler → *assembly-language file* → assembler → *object file*

linker → *executable file* → loader → *running program in memory*

About Optimization

- Code generated by a compiler is usually *optimized* to make it faster, smaller, or both

- Other optimizations may be done by the assembler, linker, and/or loader

- A misnomer: the resulting code is better, but not guaranteed to be optimal

Example

■ Original code:
```
int i = 0;
while (i < 100) {
  a[i++] = x*x*x;
}
```

■ Improved code, with loop invariant moved:
```
int i = 0;
int temp = x*x*x;
while (i < 100) {
  a[i++] = temp;
}
```

Example

- Loop invariant removal is handled by most compilers

- That is, most compilers generate the same efficient code from both of the previous examples

- So it is a waste of the programmer's time to make the transformation manually

Other Optimizations

- Some, like LIR, add variables

- Others remove variables, remove code, add code, move code around, etc.

- All make the connection between source code and object code more complicated

- A simple question, such as "What assembly language code was generated for this statement?" may have a complicated answer

Outline

■ The classical sequence

■ Variations on the classical sequence

■ Binding times

■ Debuggers

■ Runtime support

Variation: Hiding The Steps

- Many language systems make it possible to do the compile-assemble-link part with one command
- Example: **gcc** command on a Unix system:

```
gcc main.c                    gcc main.c -S
                              as main.s -o main.o
                              ld …
```

Compile-assemble-link   *Compile, then assemble,*
                                 then link

Compiling to Object Code

■ Many modern compilers incorporate all the functionality of an assembler

■ They generate object code directly

Variation: Integrated Development Environments

■ A single interface for editing, running and debugging programs

■ Integration can add power at every step:

- Editor knows language syntax

- System may keep a database of source code (not individual text files) and object code

- System may maintain versions, coordinate collaboration

- Rebuilding after incremental changes can be coordinated, like Unix `make` but language-specific

- Debuggers can benefit (more on this in a minute…)

Variation: Interpreters

■ To *interpret* a program is to carry out the steps it specifies, without first translating into a lower-level language

■ Interpreters are usually much slower

  – Compiling takes more time up front, but program runs at hardware speed

  – Interpreting starts right away, but each step must be processed in software

■ Sounds like a simple distinction…

Virtual Machines

■ A language system can produce code in a machine language for which there is no hardware: an *intermediate code*

■ Virtual machine must be simulated in software – interpreted, in fact

■ Language system may do the whole classical sequence, but then interpret the resulting intermediate-code program

■ Why?

Why Virtual Machines

■ **Cross-platform execution**

– Virtual machine can be implemented in software on many different platforms

– Simulating physical machines is harder

■ **Heightened security**

– Running program is never directly in charge

– Interpreter can intervene if the program tries to do something it shouldn't

The Java Virtual Machine

■ Java languages systems usually compile to code for a virtual machine: the JVM

■ JVM language is sometimes called *bytecode*

■ Bytecode interpreter is part of almost every Web browser

■ When you browse a page that contains a Java applet, the browser runs the applet by interpreting its bytecode

Intermediate Language Spectrum

- **Pure interpreter**
  - Intermediate language = high-level language

- **Tokenizing interpreter**
  - Intermediate language = token stream

- **Intermediate-code compiler**
  - Intermediate language = virtual machine language

- **Native-code compiler**
  - Intermediate language = physical machine language

Delayed Linking

■ Delay linking step

■ Code for library functions is not included in the executable file of the calling program

Delayed Linking: Windows

- Libraries of functions for delayed linking are stored in `.dll` files: dynamic-link library
- Many language systems share this format
- Two flavors
  - Load-time dynamic linking
    - Loader finds `.dll` files (which may already be in memory) and links the program to functions it needs, just before running
  - Run-time dynamic linking
    - Running program makes explicit system calls to find `.dll` files and load specific functions

Delayed Linking: Unix

■ Libraries of functions for delayed linking are stored in `.so` files: shared object

■ Suffix `.so` followed by version number

■ Many language systems share this format

■ Two flavors

  – Shared libraries

    ■ Loader links the program to functions it needs before running

  – Dynamically loaded libraries

    ■ Running program makes explicit system calls to find library files and load specific functions

Delayed Linking: Java

■ JVM automatically loads and links classes when a program uses them

■ Class loader does a lot of work:

– May load across Internet

– Thoroughly checks loaded code to make sure it complies with JVM requirements

Delayed Linking Advantages

■ Multiple programs can share a copy of library functions: one copy on disk and in memory

■ Library functions can be updated independently of programs: all programs use repaired library code next time they run

■ Can avoid loading code that is never used

Profiling

■ The classical sequence runs twice

■ First run of the program collects statistics: parts most frequently executed, for example

■ Second compilation uses this information to help generate better code

Dynamic Compilation

■ Some compiling takes place after the program starts running

■ Many variations:

  – Compile each function only when called

  – Start by interpreting, compile only those pieces that are called frequently

  – Compile roughly at first (for instance, to intermediate code); spend more time on frequently executed pieces (for instance, compile to native code and optimize)

■ Just-in-time (JIT) compilation

Outline

■ The classical sequence

■ Variations on the classical sequence

■ **Binding times**

■ Debuggers

■ Runtime support

Binding

■ Binding means associating two things—especially, associating some property with an identifier from the program

■ In our example program:

- What set of values is associated with **int**?

- What is the type of **fred**?

- What is the address of the object code for **main**?

- What is the value of **i**?

  ```
  int i;
  void main() {
    for (i=1; i<=100; i++)
      fred(i);
  }
  ```

Binding Times

■ Different bindings take place at different times

■ There is a standard way of describing binding times with reference to the classical sequence:

   – Language definition time
   – Language implementation time
   – Compile time
   – Link time
   – Load time
   – Runtime

Language Definition Time

■ Some properties are bound when the
 language is defined:

   – Meanings of keywords: **void**, **for**, etc.

```
int i;
void main() {
  for (i=1; i<=100; i++)
    fred(i);
}
```

Language Implementation Time

■ Some properties are bound when the language system is written:

  – range of values of type **int** in C (but in Java, these are part of the language definition)

  – implementation limitations: max identifier length, max number of array dimensions, etc

```
int i;
void main() {
  for (i=1; i<=100; i++)
    fred(i);
}
```

Compile Time

- Some properties are bound when the program is compiled or prepared for interpretation:
  - Types of variables, in languages like C and ML that use static typing
  - Declaration that goes with a given use of a variable, in languages that use static scoping (most languages)

```
int i;
void main() {
   for (i=1; i<=100; i++)
      fred(i);
}
```

Link Time

■ Some properties are bound when separately-compiled program parts are combined into one executable file by the linker:

  – Object code for external function names

```
int i;
void main() {
  for (i=1; i<=100; i++)
    fred(i);
}
```

Load Time

■ Some properties are bound when the program is loaded into the computer's memory, but before it runs:

- Memory locations for code for functions
- Memory locations for static variables

```
int i;
void main() {
  for (i=1; i<=100; i++)
    fred(i);
}
```

Run Time

■ Some properties are bound only when the code in question is executed:

– Values of variables

– Types of variables, in languages like Lisp that use dynamic typing

– Declaration that goes with a given use of a variable (in languages that use dynamic scoping)

■ Also called *late* or *dynamic* binding (everything before run time is *early* or *static*)

Late Binding, Early Binding

■ The most important question about a binding time: late or early?

  – Late: generally, this is more flexible at runtime (as with types, dynamic loading, etc.)

  – Early: generally, this is faster and more secure at runtime (less to do, less that can go wrong)

■ You can tell a lot about a language by looking at the binding times

Outline

■ The classical sequence

■ Variations on the classical sequence

■ Binding times

■ **Debuggers**

■ Runtime support

Debugging Features

■ Examine a snapshot, such as a core dump

■ Examine a running program on the fly

- – Single stepping, breakpointing, modifying variables

■ Modify currently running program

- – Recompile, relink, reload parts while program runs

■ Advanced debugging features require an integrated development environment

Debugging Information

■ Where is it executing?

■ What is the traceback of calls leading there?

■ What are the values of variables?

■ Source-level information from machine-level code

    – Variables and functions by name

    – Code locations by source position

■ Connection between levels can be hard to maintain, for example because of optimization

Outline

■ The classical sequence

■ Variations on the classical sequence

■ Binding times

■ Debuggers

■ **Runtime support**

Runtime Support

■ Additional code the linker includes even if the program does not refer to it explicitly

– Startup processing: initializing the machine state

– Exception handling: reacting to exceptions

– Memory management: allocating memory, reusing it when the program is finished with it

– Operating system interface: communicating between running program and operating system for I/O, etc.

■ An important hidden player in language systems

Conclusion

■ **Language systems implement languages**

■ **Today: a quick introduction**

■ **More implementation issues later, especially:**

  – Chapter 12: memory locations for variables
  – Chapter 14: memory management
  – Chapter 18: parameters
  – Chapter 21: cost models