Lecture M:

Running MiniJava & Bootstrapping

CSE401/501m:

Introduction to Compiler Construction

Instructor: Gilbert Bernstein

Administrivia

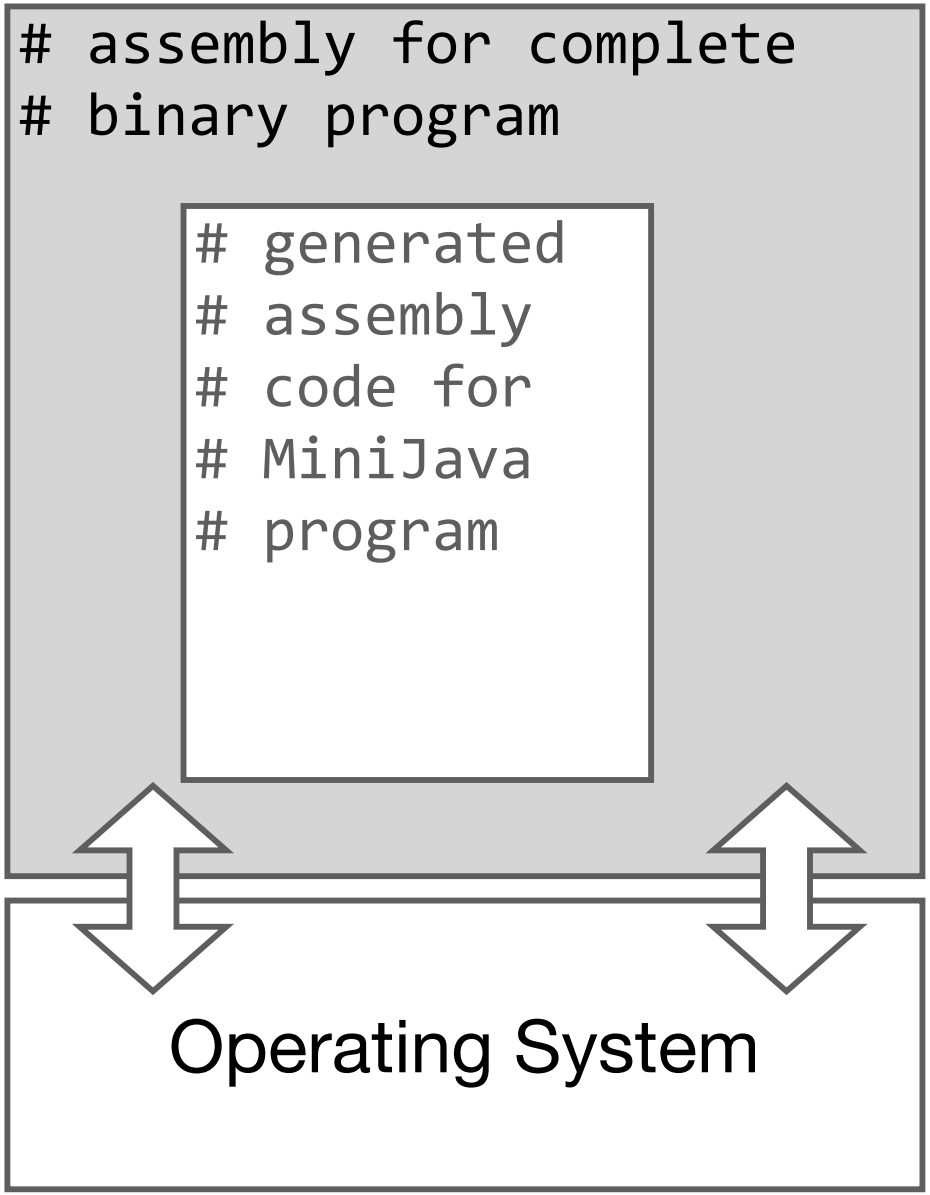
- Final part of the project — codegen — is out
  - ✦ Due Thur. May 29 (less time than checking)
  - ✦ Biggest hurdle to get started
    - Goal get `System.out.println(17)` in main method working ASAP (e.g. before the weekend)
  - ✦ Once that's done, look at the project page for a suggestion of one reasonable order to add feature support in; work incrementally and check at each stage before you proceed
    - write good tests for each stage, before starting it!

Outline

System Interaction & Bootstrapping

(rest in section)

MiniJava Programs



- On load
  - ✦ we need to allocate space for a heap and stack
  - ✦ We need to intialize `%rsp` and other reg.
- We need some way to communicate with the OS & outside world (e.g. to allocate more memory)

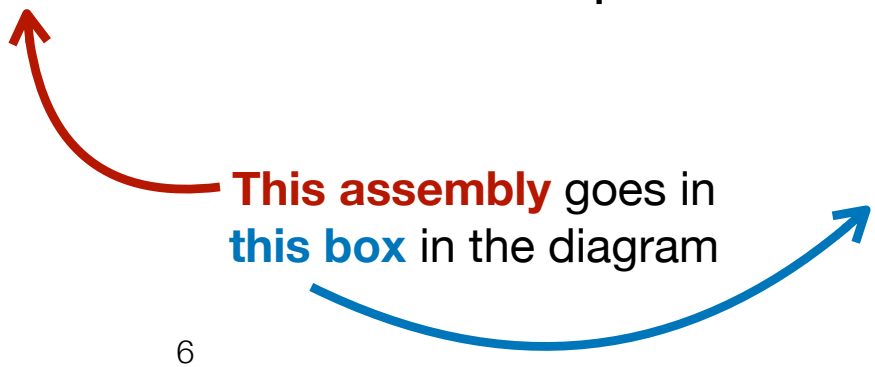
Strategy — Bootstrap from C

- Idea — let's take advantage of the existing C runtime
- Implementation — use a small C main program that sets up the process and then calls into the generated MiniJava main method (as if it is a C function)
- Wrap the C standard library in custom functions to expose I/O, malloc/calloc, etc. to the generated assembly program.

Our Assembly File Format

- Our MiniJava compiler should generate an assembly program (an ascii .s file) by writing the result to `stdout`. We can redirect this into a file using ant or a shell.
- What assembly should we generate? see `src/runtime/demo.s` in the starter project for example assembly.

```
    .data                       # static data in asm program
    <generated method tables>
    .text                       # code segment
    .globl asm_main             # expose label to linked code
asm_main:                       # start of compiled main
    <generated code>
class_1$method_1:
    <generated code>
    …
```

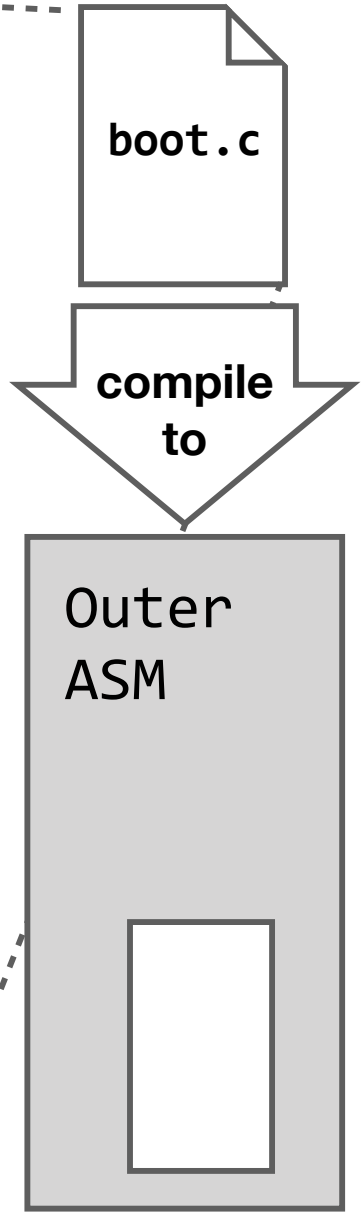


Bootstrap Program

- The bootstrap is a tiny C program that calls your compiled code as if it were an ordinary C function
- Also contains functions that your assembly can call
  - ✦ MiniJava's "runtime library"
  - ✦ You can add C functions if you want to
    - It can be easier to write some C code than generate a moderate/large amount of inline assembly
    - e.g. do for "exit if subscript out of bounds" error
    - don't go overboard and turn *everything* into a call
- File — `src/runtime/boot.c` in project starter code

Bootstrap Program (sketch)

```
# include <stdio.h>

/* extern = someone else will define
   the code for asm_main() */
extern void asm_main();

void main() { asm_main(); }

/* “runtime library” below */

/* print x to stdout */
void put(int64_t x) {…}

/* return a pointer to zero-initialized memory of
   at least nBytes, or return null on failure */
char* mjcalloc(size_t nBytes) {
  return calloc(1,nBytes);
}
```



Linking the Two Files

- In `boot.c`

```
extern void asm_main();
void main() { asm_main(); }
```

- In generated ASM

```
    .globl asm_main
asm_main: <generated code>
```

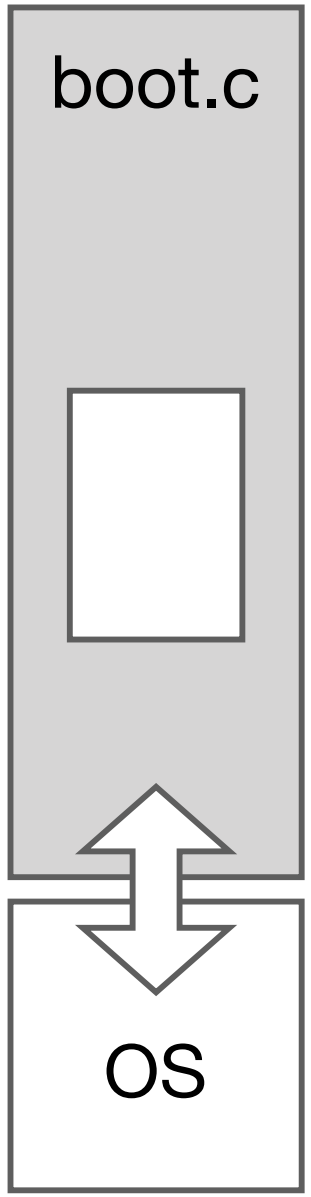
- Why does this work?
- Key point — the name declared **`extern`** and the name marked by **`.globl`** are the same!
  - ✦ You can use any name, but it must be *consistent*
  - ✦ However, why can't you use **`main`**?

Interfacing to “Library” Code

- How do we get access to system functionality?
- We define functions in `boot.c`, e.g.

```
void put(int64_t x) {…}
```

- We can call these in the usual way
  - ✦ i.e. put arguments in the expected registers, get result back in %rax
  - ✦ ***but***, no **`this`** argument (we’re calling C code)
- Generate a call to the right label, e.g. **`call put`**
  - ✦ The linker will resolve labels across files
- *Problem — different OSes mangle names differently…*



System Calls — Names

- What standard governs our function calls? (including system calls)
  - ✦ the ABI!
- On Linux, external symbols for function calls are used as-is (`xyzzy`)
  - ✦ but on Windows & x86-64 MacOS, external symbols are "name mangled" with a leading underscore `_xyzzy`
- Your compiler needs to generate code that runs on attu using Linux conventions, but feel free to support your local machine (e.g. using a compiler switch)

System.out.println(exp)

- MiniJava's "print" statement — can compile as…

```
<compile exp; result in %rax>
movq   %rax,%rdi    # load argument register
call   put          # call external put routine
```

- ***Note!*** If the stack is not properly 16-byte aligned when calls *outside* your code are executed (i.e. to external C or library code) then you can cause a runtime error (It will cause an error on x86-64 MacOS)

Notes for running code on a Mac…

- Pre m-series macs (Intel chips) will run x86-64 code; m-series processors will also run it via translation, but…
  - External labels need to start with _ (e.g. `_put`)
  - `%rsp` ***must*** be 16-byte aligned when call is executed (should be anyway, but Linux may let you get away with 8-byte alignment)
  - The addressing mode `leaq label,%rax` may be rejected, so use the weird `leaq label(%rip),%rax` mode instead
  - Probably use lldb instead of gdb on a mac
  - You may need to include `.align 8` in the assembler code before each vtable to stop linker complaints
- FINALLY, make ***sure*** that you can run your code on attu/cse vm Linux for your final version (don't generate external `_labels`)

Lecture → Section