26 Libraries and Foreign Language Interfaces

Outline

- **Announcements**
- **Overview**
- **Tools Revisited**
- **Foreign Language Calling**
- **Parts of the process**
- **Using gcc to drive the process**
- MIPS **Function examples**
  - **Multiply — calling Assembly from C**
  - **putchar() — calling C from Assembly**

26.1 Announcements

- See my Linking and Loading Handout on the web.
- Review Session — June 10, 7:00, 147 Olsson

26.2 Examples: a.s and b.s from strcat.s

26.3 Utilities from Last Time

nm Reads the symbol tables.

N	nil storage class, compiler internal usage
⟶ T	external text
t	local text
D	external initialized data
d	local initialized data
B	external zeroed data
b	local zeroed data
A	external absolute
a	local absolute
⟶ U	external undefined
G	external small initialized data
g	local small initialized data

Table 6: Symbol types reported by `nm`

dis Disassembles object code.

This lets you see what code was actually produced. It also drives home the one-to-one correspondence between source and object.

26.4 Separate Assembly

Now let's look at our little program from last time but do it as two separate pieces, assembled and linked.

26.4.1 The world of a.s

26.4.2 The world of b.s

26.4.3 The Final Version `a.out`

The program can be assembled via gcc...

26.5 Overview

Today we will do for real all that material I talked about quickly at the end of last time.

The fundamental motivation for all this is that people do not write monolithic systems in assembly. They write small parts of systems in assembly and the rest in a high level language. Why? Because writing in assembly is a pain.

But it gives you access to the lowest level things. (Context switch example.)

26.6 Libraries and Foreign Language Calling

The gcc calling protocol. The process of producing a mixed-language program is illustrated in Figure 54.

26.7 Parts of the process

A program is built in several phases. The programs involved are the:

Assembler or Compiler These translators translate source files into object code. Each part is assembled as if it were the whole program starting at address 0. Unknown (*unresolved*) and exported (global) symbols are listed in the object file's symbol table.

Linker (ld) The linker takes the separately generated object files and *links* them together into a runnable program. The linker attempts to resolve any unknown symbols in object files in other object files or in libraries.

Loader The loader takes the load module generated by the linker, copies it into memory at the appropriate locations, and calls the funcion at __Start. This function is supplied by the crt0.o file, and sets up things that `main()` can expect.

26.8 Using gcc to drive the process

It is possible to run each program separately, but for convenience' sake, it is possible to use the gcc compiler to drive the process. It is important to remember that gcc recognizes assembly input based on it's ".s" extension.

26.9 Calling Foriegn Languages

If we follow all the calling conventions, is there anything special about one language call another?

No, they all get compiled down to assembly so as long as we follow all calling conventions (and don't have any symbol conflicts) we'll be fine.

```
# char* strcat(char *dest, char *src)
# returns the value of the first string

            .globl mystrcat

mystrcat:   move $2,$4             # set return value and preserve initial pointer

l1:         lbu $8,0($4)           # get the character a0 points to
            addi $4,$4,1           # increment a0 (in delay slot)
            bgtz $8,l1             # loop until the nul is found
            addi $4,-1             # (a0 went one too far) now points to the end
l2:         lbu $9,0($5)           # get the first character of s2
            addi $5,$5,1           # increment a1 (in the delay slot)
            sb $9,0($4)            # store the byte
            addi $4,$4,1           # increment a0
            bgtz $9,l2             # loop until the nul is found (already stored)
            jr $31
```

```
elysium% nm a.o
0000000004 N l1
0000000018 N l2
0000000000 T mystrcat

elysium% dis a.o
  0x0:  00801021        move    v0,a0
  0x4:  90880000        lbu     t0,0(a0)
  0x8:  20840001        addi    a0,a0,1
  0xc:  1d00fffd        bgtz    t0,0x4
  0x10: 00000000        nop
  0x14: 2084ffff        addi    a0,a0,-1
  0x18: 90a90000        lbu     t1,0(a1)
  0x1c: 20a50001        addi    a1,a1,1
  0x20: a0890000        sb      t1,0(a0)
  0x24: 1d20fffc        bgtz    t1,0x18
  0x28: 20840001        addi    a0,a0,1
  0x2c: 03e00008        jr      ra
  0x30: 00000000        nop
  0x34: 00000000        nop
  0x38: 00000000        nop
  0x3c: 00000000        nop
```

Figure 51: The strcat file

```
            .data
s1:         .asciiz "Hello, "
            .space 128                # to be sure there's room for the catenation
s2:         .asciiz "World!\n"

            .globl main
            .text
            .align 2
main:       addi $29,$29,-4           # allocate a word on the stack
            sw $31,0($29)             # save since we're going to make calls
            la $4,s1                  # Set up the argument
            la $5,s2                  # Set up the argument
            jal mystrcat
            move $4,$2                # give result to printf
            addi $sp,$sp,-16# printf wants room for its arguments
            jal printf
            addi $sp,$sp,16
            lw $31,0($29)             # restore the return address
            addi $29,$29,4            # restore the stack
            jr $31
```

```
elysium% nm b.o
0000000000 T main
0000000000 U mystrcat
0000000000 U printf
0000000040 N s1
0000000040 d s1
00000000c8 d s2
00000000c8 N s2
```

```
elysium% dis b.o
  0x0:  23bdfffc      addi    sp,sp,-4
  0x4:  afbf0000      sw      ra,0(sp)
  0x8:  3c040000      lui     a0,0
  0xc:  24840040      addiu   a0,a0,64
  0x10: 3c050000      lui     a1,0
  0x14: 0c000000      jal     mystrcat
  0x18: 24a500c8      addiu   a1,a1,200
  0x1c: 00402021      move    a0,v0
  0x20: 0c000000      jal     printf
  0x24: 23bdfff0      addi    sp,sp,-16
  0x28: 23bd0010      addi    sp,sp,16
  0x2c: 8fbf0000      lw      ra,0(sp)
  0x30: 23bd0004      addi    sp,sp,4
  0x34: 03e00008      jr      ra
  0x38: 00000000      nop
  0x3c: 00000000      nop
```


Figure 52: The main program

```
elysium% gcc --verbose a.s b.s
 /pkg/mips-dec-ultrix4.4/bin/as -v -o /usr/tmp/cca281661.o a.s
 /pkg/mips-dec-ultrix4.4/bin/as -v -o /usr/tmp/cca281662.o b.s
 /pkg/mips-dec-ultrix4.4/bin/ld /usr/lib/cmplrs/cc/crt0.o \
-L/pkg/lib/gcc-lib/mips-dec-ultrix4.4/2.7.2.2 \
-L/pkg/mips-dec-ultrix4.4/lib -L/usr/lib/cmplrs/cc \
-L/usr/lib/cmplrs/cc -L/pkg/lib /usr/tmp/cca281661.o \
/usr/tmp/cca281662.o -lgcc -lc -lgcc
```

Figure 53: The command line for strcat()

26.10 MIPS Function examples

Here we develop a few examples of MIPS functions. The general approach:

When Called

- Save any callee-save registers used (s-registers)
- If using the stack pointer
  - Preserve old SP (or be sure pushes and pops match)
- If using the frame pointer for parameters
  - preserve old FP
  - set FP to top of stack (or other appropriate location)
- If making function calls
  - preserve any necessary parameters (a - registers)
  - preserve the return address (ra == \$31)
  - set up parameters for nested call
  - allocates stack space for $a0$–$a3$

Before Return:

- Restore any callee-save registers
- Restore stack to original state.
- Restore frame pointer, if necessary
- Restore return address, if necessary
- Place return value, if any, in $v0$ (\$r2)

Where to store the stuff? On the stack, usually. Sometimes can optimize by doing several operations at once. (*e.g.,* allocate space on the stack for several variables rather than doing independent "pushes")

26.10.1 Multiply — calling Assembly from C

See Figure 55.

26.10.2 putchar() — calling C from Assembly

See Figure 56.

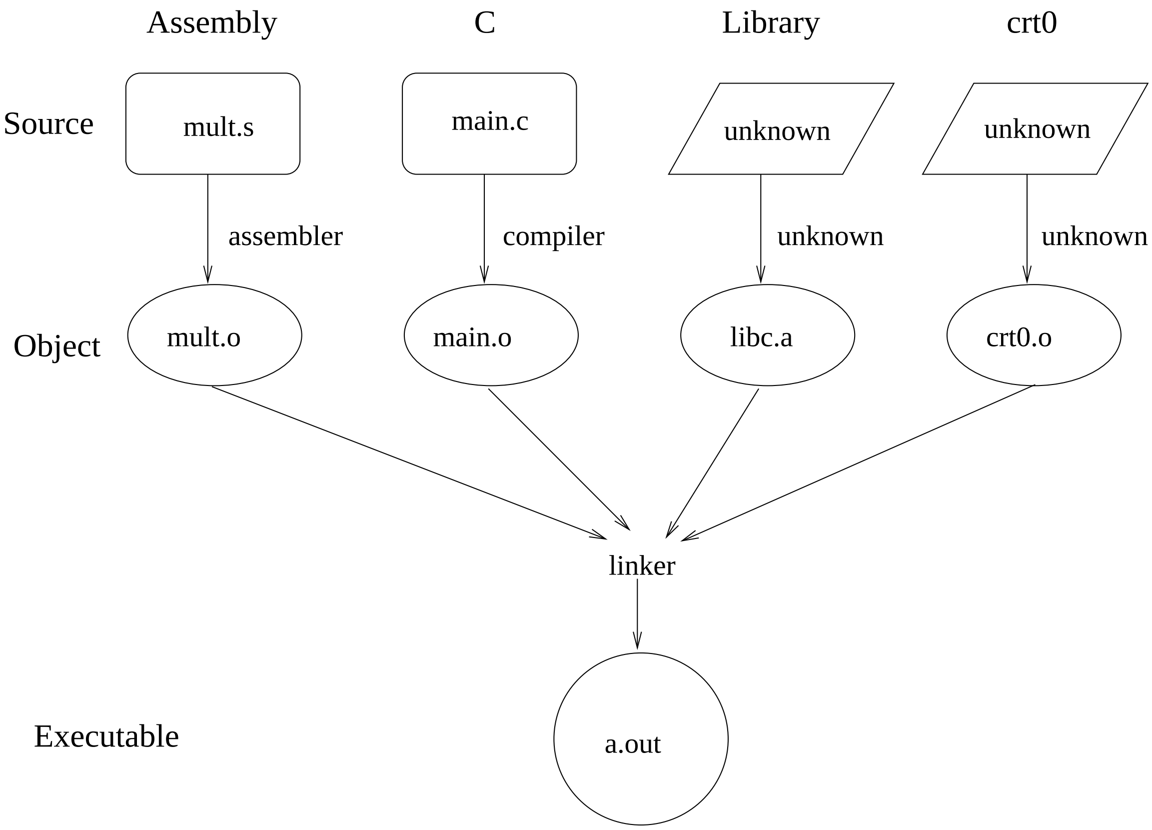

Figure 54: Linking a mixed-language program.

mult.s

```
          .text
          .globl multiply
multiply: subu $sp,$sp,8   # save return address and fp
          sw $31,0($sp)
          sw $fp,4($sp)

          mult $4,$5       # multiply the two parameters
          mflo $2          # put the result in $v0

          lw $31,0($sp)    # restore my return address
          lw $fp,4($sp)
          addi $sp,$sp,8
          jr $31
```

main.c

```
/* prototype for multiply() */
int multiply(int a, int b);

int main(int argc, char *argv[]){
   int res;
   res = multiply(6,7);
   printf("the answer is %d\n",res);
   return 0;
}
```

```
elysium% gcc -c mult.s
elysium% gcc -c main.c
elysium% gcc main.o mult.o
elysium% a.out
the answer is 42
```

Figure 55: Example: Calling assembly from a C main

```
        .text
        .globl main
main:   subu $sp,$sp,8        # save my return address and fp
        sw $31,0($sp)
        sw $fp,4($sp)

        li $4,'Q'             # print a 'Q' to test
        addi $sp,$sp,-16      # allocate those critical 4 words on the stack
        jal putchar
        addi $sp,$sp,16       # clean them up.

        lw $31,0($sp)         # restore my return address and fp
        lw $fp,4($sp)
        addi $sp,$sp,8
        jr $31
```

```
elysium% gcc foo.s
elysium% a.out
Qelysium%
```

Figure 56: Example: Calling `putchar()` from an assembly program