Lecture 10: Program Development versus Execution Environment

CSE 30: Computer Organization and Systems Programming
Winter 2010

Rajesh Gupta / Ryan Kastner
Dept. of Computer Science and Engineering
University of California, San Diego
C Memory Management

- C has 3 pools of memory
  - **Static storage**: global variable storage, basically permanent, entire program run
  - **The Stack**: local variable storage, parameters, return address
    ("stack frame" in C)
  - **The Heap** (dynamic storage): data lives until deallocated by programmer

- C requires knowing where objects are in memory, otherwise don't work as expected
Normal C Memory Management

- A program’s *address space* contains 4 regions:
  - **stack**: local variables, grows downward
  - **heap**: space requested for pointers via `malloc()`; resizes dynamically, grows upward
  - **static data**: variables declared outside main, does not grow or shrink
  - **code**: loaded when program starts, does not change

For now, OS somehow prevents accesses between stack and heap (gray hash lines). Wait for virtual memory
Intel 80x86 C Memory Management

- A C program’s 80x86 *address space*:
  - **heap**: space requested for pointers via `malloc()`; resizes dynamically, grows upward
  - **static data**: variables declared outside main, does not grow or shrink
  - **code**: loaded when program starts, does not change
  - **stack**: local variables, grows downward

```
~ 08000000_{hex}
```

UCSD
Memory Management

- How do we manage memory?
- **Code, Static storage are easy:**
  they never grow or shrink
- **Stack space is also easy:**
  stack frames are created and destroyed in last-in, first-out (LIFO) order
- **Managing the heap is tricky:**
  memory can be allocated / deallocated at any time
Where allocated?

```c
int myGlobal;
main() { int temp; }
```

- Structure declaration **does not** allocate memory
- Variable declaration **does** allocate memory
  - If declare **outside** a procedure, allocated in static storage
  - If declare **inside** procedure, allocated on the stack and **freed when procedure returns**
The Stack

- Stack frame includes:
  - Return address
  - Parameters
  - Space for other local variables

- Stack frames contiguous blocks of memory; stack pointer tells where top stack frame is

- When procedure ends, stack frame is tossed off the stack; frees memory for future stack frames
Stack

- Last In, First Out (LIFO) memory usage

```c
main ()
{ a(0); }

void a (int m)
{ b(1); }

void b (int n)
{ c(2); }

void c (int o)
{ d(3); }

void d (int p)
{
}
```

Stack Pointer

Stack Pointer

Stack Pointer

Stack Pointer

UCSD
Who cares about stack management?

- Pointers in C allow access to deallocated memory, leading to hard-to-find bugs!

```c
int * ptr () {
    int y;
    y = 3;
    return &y;
};

main () {
    int *stackAddr, content;
    stackAddr = ptr();
    content = *stackAddr;
    printf("%d", content); /* 3 */
    content = *stackAddr;
    printf("%d", content); /*13451514 */
};
```
The Heap (Dynamic memory)

- Large pool of memory, **not** allocated in contiguous order
  - back-to-back requests for heap memory could result blocks very far apart
  - where C++/Java `new` command allocates memory

- In C, specify number of **bytes** of memory explicitly to allocate item

```c
int *ptr;
ptr = (int *) malloc(4);
/* malloc returns type (void *),
so need to cast to right type */
```

- `malloc()`: Allocates raw, uninitialized memory from heap
Dynamic Memory Allocation

- C has operator `sizeof()` which gives size in bytes (of type or variable)
- Assume size of objects can be misleading & is bad style, so use `sizeof(type)`
  - Many years ago an `int` was 16 bits, and programs assumed it was 2 bytes
Dynamic Memory Allocation

- To allocate room for something new to point to, use `malloc()` (with the help of a typecast and `sizeof`):

  ```c
  ptr = (int *) malloc (sizeof(int));
  ```

  - Now, `ptr` points to a space somewhere in memory of size `(sizeof(int))` in bytes.
  - `(int *)` simply tells the compiler what will go into that space (called a typecast).

- `malloc` is almost never used for 1 variable

  ```c
  ptr = (int *) malloc (n*sizeof(int));
  ```

  - This allocates an array of `n` integers.
Dynamic Memory Allocation

- Once `malloc()` is called, the memory location might contain anything, so don’t use it until you’ve set its value.
- After dynamically allocating space, we must dynamically free it:
  ```c
  free(ptr);
  ```
- Use this command to clean up.
In general, we interpret a high level language if efficiency is not critical or translated to a lower level language to improve performance.
Interpretation vs. Translation

- How do we run a program written in a source language?
  - **Interpreter**: Directly executes a program in the source language
  - **Translator**: Converts a program from the source language to an equivalent program in another language
Interpretation

MIPS program: foo.m

MARS
Translation

Scheme program: foo.c

C Compiler (gcc)

Executable (mach lang pgm): a.out

Hardware

° C Compiler is a translator from C to machine language
Interpretation

- Any good reason to interpret machine language in software?
- MARS – useful for learning / debugging
- Apple Macintosh conversion
  - Switched from Motorola 680x0 instruction architecture to PowerPC.
  - Could require all programs to be re-translated from high level language
  - Instead, let executables contain old and/or new machine code, interpret old code in software if necessary
Interpretation vs. Translation

- Easier to write interpreter
- Interpreter closer to high-level, so gives better error messages (e.g., MARS)
  - Translator reaction: add extra information to help debugging (line numbers, names)
- Interpreter slower (10x?) but code is smaller (1.5X to 2X?)
- Interpreter provides instruction set independence: run on any machine
  - HP switched to VLIW processor. Instead of retranslating all SW, let executables contain old and/or new machine code, interpret old code in software if necessary
Steps to Running a Program

C program: foo.c

Compiler

Assembly program: foo.s

Assembler

Object (mach lang module): foo.o

Linker

Executable (mach lang pgm): a.out

Loader

Memory

lib.o
Compiler

❑ Input: High-Level Language Code (e.g., C, Java such as foo.c, foo.java)

❑ Output: Assembly Language Code (e.g., foo.s for MIPS)

❑ Note: Output *may* contain pseudoinstructions

❑ **Pseudoinstructions**: instructions that assembler understands but not in machine For example:

❑ `mov $s1, $s2 = or $s1, $s2, $zero`
Where Are We Now?

C program: foo.c

Compiler

Assembly program: foo.s

Assembler

Object (mach lang module): foo.o

Linker

Executable (mach lang pgm): a.out

Lib.o

Loader

Memory
Assembler

- Input: Assembly Language Code (e.g., foo.s for MIPS)
- Output: Object Code, information tables (e.g., foo.o for MIPS)
- Reads and Uses Directives
- Replace Pseudoinstructions
- Produce Machine Language
- Creates Object File
Assembler Directives (p. A–51 to A–53)

- Give directions to assembler, but do not produce machine instructions
  - `.text`: Subsequent items put in user text segment
  - `.data`: Subsequent items put in user data segment
  - `.globl sym`: declares `sym` global and can be referenced from other files
  - `.ascii str`: Store the string `str` in memory and null-terminate it
  - `.word w1...wn`: Store the `n` 32-bit quantities in successive memory words
### Pseudoinstruction Replacement

- Assembler treats convenient variations of machine language instructions as if real instructions.

<table>
<thead>
<tr>
<th>Pseudo:</th>
<th>Real:</th>
</tr>
</thead>
<tbody>
<tr>
<td>subu $sp,$sp,32</td>
<td>addiu $sp,$sp,-32</td>
</tr>
<tr>
<td>sd $a0, 32($sp)</td>
<td>sw $a0, 32($sp)</td>
</tr>
<tr>
<td>mul $t7,$t6,$t5</td>
<td>mul $t6,$t5</td>
</tr>
<tr>
<td>addu $t0,$t6,1</td>
<td>addiu $t0,$t6,1</td>
</tr>
<tr>
<td>ble $t0,100,loop</td>
<td>slti $at,$t0,101</td>
</tr>
<tr>
<td>la $a0, str</td>
<td>lui $at,left(str)</td>
</tr>
<tr>
<td></td>
<td>ori $a0,$at,right(str)</td>
</tr>
</tbody>
</table>
Producing Machine Language

- Simple Case
  - Arithmetic, Logical, Shifts, and so on
  - All necessary info is within the instruction already

- What about Branches?
  - PC-Relative
  - So once pseudoinstructions are replaced by real ones, we know by how many instructions to branch

- So these can be handled easily
What about jumps (j and jal)?
- Jumps require absolute address

What about references to data?
- la gets broken up into lui and ori
- These will require the full 32-bit address of the data
- These can’t be determined yet, so we create two tables...
Symbol Table

- List of “items” in this file that may be used by other files
- What are they?
  - Labels: function calling
  - Data: anything in the `.data` section; variables which may be accessed across files
- First Pass: record label-address pairs
- Second Pass: produce machine code
  - Result: can jump to a later label without first declaring it
Relocation Table

- List of “items” for which this file needs the address
  - What are they?
    - Any label jumped to: j or jal
    - internal
    - external (including lib files)
  - Any piece of data
    - such as the la instruction
Object File Format

- **object file header**: size and position of the other pieces of the object file
- **text segment**: the machine code
- **data segment**: binary representation of the data in the source file
- **relocation information**: identifies lines of code that need to be “handled”
- **symbol table**: list of this file’s labels and data that can be referenced
- **debugging information**
Where Are We Now?

- C program: foo.c
- Assembly program: foo.s
- Assembly program: foo.s
- Object (mach lang module): foo.o
- Linker
- Executable (mach lang pgm): a.out
- Loader
- Memory
Link Editor/Linker

- **Input:** Object Code, information tables (e.g., `foo.o` for MIPS)
- **Output:** Executable Code (e.g., `a.out` for MIPS)
- Combines several object (.o) files into a single executable ("linking")
- Enable Separate Compilation of files
  - Changes to one file do not require recompilation of whole program
    - Red Hat Linux 7.1 source is ~30 M lines of code!
  - Link Editor name from editing the "links" in jump and link instructions
Link Editor/Linker

- `.o` file 1:
  - text 1
  - data 1
  - info 1

- `.o` file 2:
  - text 2
  - data 2
  - info 2

- Linker
  - Relocated text 1
  - Relocated text 2
  - Relocated data 1
  - Relocated data 2

- a.out
Link Editor/Linker

- Step 1: Take text segment from each .o file and put them together.
- Step 2: Take data segment from each .o file, put them together, and concatenate this onto end of text segments.
- Step 3: Resolve References
  - Go through Relocation Table and handle each entry
  - That is, fill in all absolute addresses
Four Types of Addresses

- PC-Relative Addressing (beq, bne): never relocate
- Absolute Address (j, jal): always relocate
- External Reference (usually jal): always relocate
- Data Reference (often lui and ori): always (??) relocate
Resolving References

- Linker *assumes* first word of first text segment is at address 0x00000000

- Linker knows:
  - length of each text and data segment
  - ordering of text and data segments

- Linker calculates:
  - absolute address of each label to be jumped to (internal or external) and each piece of data being referenced
Resolving References

- To resolve references:
  - search for reference (data or label) in all symbol tables
  - if not found, search library files (for example, for `printf`)
  - once absolute address is determined, fill in the machine code appropriately

- Output of linker: executable file containing text and data (plus header)
Where Are We Now?

C program: foo.c

Compiler

Assembly program: foo.s

Assembler

Object (mach lang module): foo.o

Linker

Executable (mach lang pgm): a.out

Loader

Memory
Loader

- Input: Executable Code (e.g., $a.out$ for MIPS)
- Output: (program is run)
- Executable files are stored on disk
- When one is run, loader’s job is to load it into memory and start it running
- In reality, loader is the operating system (OS)
  - loading is one of the OS tasks
So what does a loader do?

Reads executable file’s header to determine size of text and data segments

Creates new address space for program large enough to hold text and data segments, along with a stack segment

Copies instructions and data from executable file into the new address space (this may be anywhere in memory)
Loader

- Copies arguments passed to the program onto the stack
- Initializes machine registers
  - Most registers cleared, but stack pointer assigned address of 1st free stack location
- Jumps to start-up routine that copies program’s arguments from stack to registers and sets the PC
  - If main routine returns, start-up routine terminates program with the exit system call
Example: C ⇒ Asm ⇒ Obj ⇒ Exe ⇒ Run

```c
#include <stdio.h>
int main (int argc, char *argv[]) {
    int i;
    int sum = 0;

    for (i = 0; i <= 100; i = i + 1)
        sum = sum + i * i;
    printf ("The sum of squares from 0 .. 100 is %d\n",    sum);
}
```
Example: C ⇒ Asm ⇒ Obj ⇒ Exe ⇒ Run

```assembly
.text
.align 2
.globl main
main:
    subu $sp,$sp,32
    sw $ra, 20($sp)
    sd $a0, 32($sp)
    sw $0, 24($sp)
    sw $0, 28($sp)
loop:
    lw $t6, 28($sp)
    mul $t7, $t6, $t6
    lw $t8, 24($sp)
    addu $t9,$t8,$t7
    sw $t9, 24($sp)
    addu $t0, $t6, 1
    sw $t0, 28($sp)
    ble $t0, 100, loop
    la $a0, str
    lw $a1, 24($sp)
    jal printf
    move $v0, $0
    lw $ra, 20($sp)
    addiu $sp,$sp,32
    jr $ra
.data
.align 0
.str:
    .asciiz "The sum from 0 .. 100 is %d\n"
```

Where are 7 pseudo-instructions?
Example: C ⇒ Asm ⇒ Obj ⇒ Exe ⇒ Run

.text
.align 2
.globl main
main:
    subu $sp, $sp, 32
    sw $ra, 20($sp)
    sd $a0, 32($sp)
    sw $0, 24($sp)
    sw $0, 28($sp)
    loop:
    lw $t6, 28($sp)
    mul $t7, $t6, $t6
    lw $t8, 24($sp)
    addu $t9, $t8, $t7
    sw $t9, 24($sp)
    addu $t0, $t6, 1
    sw $t0, 28($sp)
    ble $t0, 100, loop
    la $a0, str
    lw $a1, 24($sp)
    jal printf
    move $v0, $0
    lw $ra, 20($sp)
    addiu $sp, $sp, 32
    jr $ra
.data
    .align 0
    str:
    .asciiz "The sum from 0 .. 100 is %d\n"
7 pseudo-instructions underlined
## Symbol Table Entries

- **Symbol Table**
  - **Label**       **Address**
  - main:            ?
  - loop:            
  - str:             
  - printf:          

- **Relocation Table**
  - **Address**    **Instr. Type**  **Dependency**
Example: C ⇒ Asm ⇒ Obj ⇒ Exe ⇒ Run

- Remove pseudoinstructions, assign addresses

<table>
<thead>
<tr>
<th>C</th>
<th>Asm</th>
<th>Obj</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>addiu $29, $29, -32</td>
<td>30 addiu $8, $14, 1</td>
</tr>
<tr>
<td>04</td>
<td>sw $31, 20 ($29)</td>
<td>34 sw $8, 28 ($29)</td>
</tr>
<tr>
<td>08</td>
<td>sw $4, 32 ($29)</td>
<td>38 slti $1, $8, 101</td>
</tr>
<tr>
<td>0c</td>
<td>sw $5, 36 ($29)</td>
<td>3c bne $1, $0, loop</td>
</tr>
<tr>
<td>10</td>
<td>sw $0, 24 ($29)</td>
<td>40 lui $4, l.str</td>
</tr>
<tr>
<td>14</td>
<td>sw $0, 28 ($29)</td>
<td>44 ori $4, $4, r.str</td>
</tr>
<tr>
<td>18</td>
<td>lw $14, 28 ($29)</td>
<td>48 lw $5, 24 ($29)</td>
</tr>
<tr>
<td>1c</td>
<td>multu $14, $14</td>
<td>4c jal printf</td>
</tr>
<tr>
<td>20</td>
<td>mflo $15</td>
<td>50 add $2, $0, $0</td>
</tr>
<tr>
<td>24</td>
<td>lw $24, 24 ($29)</td>
<td>54 lw $31, 20 ($29)</td>
</tr>
<tr>
<td>28</td>
<td>addu $25, $24, $15</td>
<td>58 addiu $29, $29, 32</td>
</tr>
<tr>
<td>2c</td>
<td>sw $25, 24 ($29)</td>
<td>5c jr $31</td>
</tr>
</tbody>
</table>
Symbol Table Entries

- **Symbol Table**
  - **Label** | **Address**
    - main: 0x00000000
    - loop: 0x00000018
    - str: 0x10000430
    - printf: 0x000003b0

- **Relocation Information**
  - **Address** | **Instr.** | **Type** | **Dependency**
    - 0x00000040 | lui  | l.str
    - 0x00000044 | ori  | r.str
    - 0x0000004c | jal  | printf
### Addresses start at 0x00000000

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>addiu $29, $29, -32</td>
<td>30</td>
<td>addiu $8, $14, 1</td>
</tr>
<tr>
<td>04</td>
<td>sw $31, 20($29)</td>
<td>34</td>
<td>sw $8, 28($29)</td>
</tr>
<tr>
<td>08</td>
<td>sw $4, 32($29)</td>
<td>38</td>
<td>slti $1, $8, 101</td>
</tr>
<tr>
<td>0c</td>
<td>sw $5, 36($29)</td>
<td>3c</td>
<td>bne $1, $0, -10</td>
</tr>
<tr>
<td>10</td>
<td>sw $0, 24($29)</td>
<td>40</td>
<td>lui $4, 4096</td>
</tr>
<tr>
<td>14</td>
<td>sw $0, 28($29)</td>
<td>44</td>
<td>ori $4, $4, 1072</td>
</tr>
<tr>
<td>18</td>
<td>lw $14, 28($29)</td>
<td>48</td>
<td>lw $5, 24($29)</td>
</tr>
<tr>
<td>1c</td>
<td>multu $14, $14</td>
<td>4c</td>
<td>jal 236</td>
</tr>
<tr>
<td>20</td>
<td>mflo $15</td>
<td>50</td>
<td>add $2, $0, $0</td>
</tr>
<tr>
<td>24</td>
<td>lw $24, 24($29)</td>
<td>54</td>
<td>lw $31, 20($29)</td>
</tr>
<tr>
<td>28</td>
<td>addu $25, $24, $15</td>
<td>58</td>
<td>addiu $29, $29, 32</td>
</tr>
<tr>
<td>2c</td>
<td>sw $25, 24($29)</td>
<td>5c</td>
<td>jr $31</td>
</tr>
</tbody>
</table>
Example: C ⇒ Asm ⇒ Obj ⇒ Exe ⇒ Run
Things to Remember

C program: foo.c

Compiler

Assembly program: foo.s

Assembler

Object (mach lang module): foo.o

Linker

Executable (mach lang pgm): a.out

Loader

Memory

lib.o
Things to Remember

- Compiler converts a single HLL file into a single assembly language file.
- Assembler removes pseudoinstructions, converts what it can to machine language, and creates a checklist for the linker (relocation/symbol table). This changes each .s file into a .o file.
- Linker combines several .o files and resolves absolute addresses.
- Loader loads executable into memory and begins execution.
Things to Remember

- Stored Program concept means instructions just like data, so can take data from storage, and keep transforming it until load registers and jump to routine to begin execution
  - Compiler $\Rightarrow$ Assembler $\Rightarrow$ Linker ($\Rightarrow$ Loader)
- Assembler does 2 passes to resolve addresses, handling internal forward references
- Linker enables separate compilation, libraries that need not be compiled, and resolves remaining addresses
Conclusion

- C has 3 pools of memory
  - **Static storage**: global variable storage, basically permanent, entire program run
  - **The Stack**: local variable storage, parameters, return address
  - **The Heap** (dynamic storage): `malloc()` grabs space from here, `free()` returns it.

- `malloc()` handles free space with freelist. Three different ways to find free space when given a request:
  - **First fit** (find first one that’s free)
  - **Next fit** (same as first, but remembers where left off)
  - **Best fit** (finds most “snug” free space)