COMP 264 Fall 2015 Lecture Notes

R. I. Greenberg

October 19, 2015

Machine-Level Rep. of Programs continued

1 Procedures

Three key requirements:

- Transfers of control to and from subroutines.
- Transfer of data (arguments and return values).
- Allocate space for local variables on procedure entry; deallocate on exit.

Transfer of control relatively straightforward. For data needs, use the stack. (Recall *rtimage.ppt*.)

1.1 Stack Frame Structure and Caller/Calleee Conventions

- One *stack frame* per procedure execution.
- Frame for most recently called procedure at top of stack. Occupies locations from *frame pointer* (in `%rbp`) to *stack pointer* (in `%rsp`). See *frame.ppt*.
- Caller obligations:
  1. Save needed values in any registers for which *caller save* applies. In IA32: `%eax`, `%ecx`, `%edx`.
  2. Push arguments onto stack (still in caller's frame).
  3. Push return address onto stack (in caller's frame).
  4. Transfer control to subroutine.

5. (After control returns:) As necessary, clean up argument build area, and restore values in caller save registers.

- Callee obligations:

1. Push `%rbp` onto stack and set `%rbp` to point to stack top. (Marks beginning of new frame; first thing stored is old `%rbp`.)
2. Save values in any needed registers for which *callee save* applies. In IA32: `%ebx`, `%esi`, `%edi`.
3. Allocate space on stack for local variables for which registers don't suffice.
4. Do the actual work of the subroutine body.
5. Restore values for any callee save registers that were needed.
6. Restore `%rbp` and `%rsp` as before the call. (Throws away current frame.)
7. Pop return address and transfer control.

- Two options for caller to save data for after the subroutine:
  - Put it into a callee save register.
  - Push it into caller's stack frame.
- If callee wants to use a callee save register, must push content into callee's stack frame.

1.2 GAS Instructions for Procedure Entry/Exit

- `call` is just like `jmp` except that it causes PC to be pushed onto stack before transfer of control. Has direct (without `*`) and indirect (with `*`) versions like `jmp` does.
- `ret` Returns control from callee to caller, assuming that the return address is on the top of the stack.

Still need some additional standard manipulations involving `%rbp` and `%rsp` to maintain stack frame discipline. (Can review how a stack works in *06-machine-procedures slides* 21–23.) In particular, when procedure ends, following code sequence, will discard current frame and bring return address to top of stack:

```
movl %rbp, %rsp
popl %rbp
```

- GAS abbreviation for above sequence of two instrs. is `leave`.

Need to take complementary steps after entering procedure, i.e.,:

```
pushl %rbp
movl %rsp, %rbp
```

For some reason, this sequence of two instrs. doesn't seem to have an abbreviation.

1.3 Example

For the following code: caller and callee stack frames are as in *swap.ppt*.

Caller puts its variables arg1 and arg2 on the stack since needs to generate addresses for them. Then pushes `&arg1` and `&arg2` onto the stack, since these are the actual arguments that must be made available to the callee. Finally, call to swap_add, pushes return address on stack before control is transfered.

Code generated on our Linux cluster (with `-O2` and `-m32` flags) for the swap_add procedure (with annotations added): is just a little different than the code in our text.

First two lines: save old frame pointer; begin new frame by having frame pointer point to current top of stack (as discussed just above)

Next swap_add does actual work of the subroutine, and since will store into `%ebx`, which is a callee save register, `%ebx` is pushed onto the stack before it is used. The procedure arguments, denoted by the formal parameters `xp` and `yp` are accessed at specific offsets from the callees frame pointer.

Fourth from last line: sets up return value using convention that return values are always placed in `%rax`.

Next: restore old value of the callee save register `%rbx` that we needed to use.

Finally, strip away callee's stack frame and return control to caller. In this case, compiler didn't need complete `leave` sequence, because it figured out that `%rbp` and `%rsp` are already equal at this point.

(Can see similar example in *06-machine-procedures slides* 42–49.)

1.4 Recursive Procedures

Stack frame mechanisms we've seen so far completely suffice to permit recursive procedures. Though each instantiation of procedure has formal parameters and local variables of the same name, they all get separate storage in different stack frames.

Can look at a simpler recursive procedure, e.g., to compute $n!$ (similar to ex. in text):

```
int fact(int n)
{
 if (n<=0) return 1;
 return n*fact(n-1);
}
```

(Can see management of stack frames for a recursive procedure in *06-machine-procedures slides* 55–61.)

2 Array Allocation and Access

- Array declarations in C allocate a region of memory and create a pointer to the beginning.
- Ex.: Assume `short` is 2-byte data. Then after the declaration

  ```
  short D[11];
  ```

  the value of the identifier `D` will be some memory address x_D, and bytes x_D, $x_D + 1$, $x_D + 2$, $\ldots x_D + 21$ ($22 = 11 \cdot 2$ bytes in total) are reserved to hold the contents of the array of shorts. First two bytes (x_D and $x_D + 1$) hold element `D[0]`; next two hold `D[1]`, etc. In general, ith elt. begins at address $x_D + L \cdot i$, where L is the no. of bytes in each elt. (2 in this ex.).
- Operand forms (addressing modes) of Figure 3.3 supported by IA32 allow for convenient array access. e.g., suppose x_D of above ex. is in `%rdx` and i is in `%rsi`. Then GAS instr.

  ```
  movw (%rdx,%rsi,2), %rax
  ```

  puts $D[i]$ into `%rax`.
- C incorporates special treatment of arithmetic on pointers consistent with above.

  If `p` is a pointer variable with value decimal 100, does

  p+1 == 101 ????

  Maybe not! e.g., from ex. above, if `D` evaluates to 100 (the address of `D[0]`), then `D+1` evaluates to 102 (the address of `D[1]`).

- We've seen inverse operators in C of `&` (address of) and `*` (dereference); can apply to expressions as well as simple variables. Some examples based on example from above:

  `&D[3]` evaluates to 106 (i.e., $x_D + 3 \cdot L$ with $L = 2$).
  `&(x+y)` illegal.
  `D[3]` evaluates to `M`[106].
  `*(D+3)` evaluates to `M`[106].

2.1 Arrays and Loops

Looping through elts. of an array to do some operation on each is a common programming idiom. Compilers often introduce optimizations that replace the loop variable in the source code with a more directly useful loop variable, e.g., the address of the next array elt. For example, effectively transform to "pointer code": before translating to assembly.

2.2 Multidimensional or Nested Arrays

Can think of a two-dimensional array as a one-dimensional array of one-dimensional arrays. Similarly for arrays with more dimensions. More detail in text; we won't get bogged down into details of assembly language coding to handle multidimensional arrays. But good to at least understand basic C syntax and storage model for two-dimensional arrays. Can think of

```
float A[5][2];
```

as declaring an array of 5 rows and 2 columns, each crosspoint holding a float. Layed out in memory in *row major* order: `A[0][0]`, then `A[0][1]`, then `A[1][0]`, then `A[1][1]`, then `A[2][0]`, then `A[2][1]`, etc.

2.3 Dynamically Allocated Arrays

In C, you can't do e.g.:

```
int n;
scanf("%d",&n);
int A[n];
```

Instead can explicitly allocate memory during program execution:

```
int *A, n;
scanf("%d",&n);
A=(int *)calloc(sizeof(int),n);
```

Memory allocated this way is on the *heap*; recall *rtimage.ppt*.

2.4 More on Fixed Size Arrays vs Dynamically Allocated

If you're interested, you can see the text for an example of how machine code can be optimized in connection with working with fixed-size or dynamically-allocated two-dimensional arrays.

(Additional overview of arrays also available in *07-machine-data slides*.)

3 Heterogeneous Data Structures

Two C constructs for combining objects of different type:

- Structures with keyword `struct` for multiple non-overlapping objects.
- Unions with keyword `union` to allow a single object to be handled under differing type rules.

3.1 Structures

- e.g., might have a header file `date.h` containing following:
- Use "dot" notation to reference slots (members). e.g., a declaration of a variable of type `struct date` and setting of its slots:
  ```
  struct date today;
  today.year=2015;
  today.month=3;
  today.day=19;
  ```
- Can nest structures, too, e.g.,:

  ```
  struct student {
     int pidno;
     char *name;
     struct date birthdate;
     char *major;
  }
  ```

- Functions operating on structure content will often take a pointer to a structure as an argument so that it's not necessary to copy all the contents. e.g., The idiom of dereferencing a pointer to a structure and then selecting a slot is sufficiently common that there is a C shorthand notation:

  `(*structptr).slotname` is equivalent to `structptr->slotname`

  So we could rewrite function above:

- Memory layout involves allocating space for slot values one after another in memory. Easy for compiler to generate references to slot values, since each slot is at a fixed offset from beginning of structure; examples in text.

  Further examples in text. Also a recursive example in *07-machine-data slides* 38–41. In this example, it is important to recognize the precedence of operators, e.g., that `[]` and `->` are performed (left to right) before `&`. The code for the structure and the subroutines (last written more compactly) with parentheses to clarify the precedence is here: We'll explain in class what these procedures are doing.

3.2 Unions

Usage less common. Could be useful for writing a function that can take an argument that is sometimes of one type and sometimes of another, but this is still not a convenient thing in C; one of the reasons C++ became popular. Look at the text if you're interested.

4 Alignment

Many systems restrict the addresses that can be used for certain types of objects.

Examples:

- IA32 will work without any alignment requirements. But imposing some alignment improves memory system performance. Linux policy: 2 byte objects (e.g., `short int`) must have an address that is a multiple of 2; larger objects (e.g., `int`, `float`, `double`) must have an address that is a multiple of 4.
- Windows requires that any (primitive) object occupying k bytes must have an address that is a multiple of k.

Compiler can place directives in assembly code to achieve specified alignments, e.g., `.align 2` to make sure that the next thing will be at an even address (by leaving a byte empty if necessary). Examples in text for structure/array allocations.

Could go to *08-machine-advanced slides* 3–12 for examples of alignment with structures and arrays.

5 Exam 2

End of material for Exam 2. Exam 2 is 11/11 on Chapter 3. Open book, notes; calculators permitted. Modeled after practice problems 3.1, 6, 7, 35, 36, and 41.

* That only works if a scale factor of 16 is allowed, which actually isn't. So would need something like:

```
leaq -3(,%rcx,8),%rbx
movw (%rdx,%rbx,2),%ax
```