Recursion & Pointers
Lecture 8

Instructor:
Alvin R. Lebeck

Some Slides based on those from Randy Bryant and Dave O’Hallaron

Admin
- Homework #2 Due Sunday Sept 25 11:59 pm
- Homework #3 will be up by Friday, Due Sunday Oct 2 11:59pm
- Reading: Chapter 3
- Midterm Wednesday Oct 5

Assembly Programming (x86)
- Calling Conventions
- Procedure Calls
- Recursion
- Examples
Procedure Call GAP

ISA Level
- call and return instructions

C Level
- Local Name Scope
  - change tsame to same
- Recursion
- Arguments/parameters and Return Value (functions)

Assembly Level
- Must bridge gap between HLL and ISA
- Supporting Local Names
- Passing Arguments/Parameters (arbitrary number?)
- What data structure?

IA32/Linux Stack Frame

- Current Stack Frame (“Top” to Bottom)
  - “Argument build:”
    - Parameters for function about to call
  - Local variables
    - If can’t keep in registers
  - Saved register context
  - Old frame pointer

- Caller Stack Frame
  - Return address
    - Pushed by call instruction
  - Arguments for this call

<table>
<thead>
<tr>
<th>Caller Frame</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame pointer %ebp</td>
<td>Return Addr</td>
</tr>
<tr>
<td>Old %ebp</td>
<td>Saved Registers</td>
</tr>
<tr>
<td>+ Local Variables</td>
<td>Argument Build</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stack pointer %esp</th>
<th></th>
</tr>
</thead>
</table>

3
Register Saving Conventions

■ When procedure \texttt{yoo} calls \texttt{who}:
  ▪ \texttt{yoo} is the \texttt{caller}
  ▪ \texttt{who} is the \texttt{callee}

■ Can Register be used for temporary storage?

\begin{verbatim}
\texttt{yoo:} \\
  ... \\
  movl $15213, %edx \\
  call who \\
  addl %edx, %eax \\
  ... \\
  ret
\end{verbatim}

\begin{verbatim}
\texttt{who:} \\
  ... \\
  movl 8(%ebp), %edx \\
  addl $18243, %edx \\
  ... \\
  ret
\end{verbatim}

■ This could be trouble → something should be done!
  ▪ Need some coordination

■ Conventions
  ▪ “\texttt{Caller Save}”
    ▪ Caller saves temporary values in its frame before the call
  ▪ “\texttt{Callee Save}”
    ▪ Callee saves temporary values in its frame before using
IA32/Linux+Windows Register Usage

- `%eax`, `%edx`, `%ecx`  
  - Caller saves prior to call if values are used later

- `%eax`  
  - also used to return integer or pointer value

- `%ebx`, `%esi`, `%edi`  
  - Callee saves if wants to use them

- `%esp`, `%ebp`  
  - special form of callee save  
  - Restored to original values upon exit from procedure

---

IA32/GCC Procedure Calling Conventions

**Calling Procedure**

- **Step-1:** Save *caller-saved* registers  
  - Save registers `%eax`, `%ecx`, `%edx` if they contain live values at the call site.

- **Step-2:** Setup the arguments:
  - Push arguments onto the stack in reverse order

- **Step-3:** Execute a *call* instruction.
IA32/GCC Calling Conventions (cont.)

**Called Routine**
- **Step-1:** Update the frame pointer
  
  ```
  pushl %ebp  
  movl %esp, %ebp  
  ```
- **Step-2:** (optional) Allocate space for frame
  - Subtract the frame size from the stack pointer
    
    ```
    subl $<frame-size>, %esp  
    ```
  - Space is for local variables and saved registers
  - May often allocate more space than necessary.
- **Step-3:** Save *callee-saved* registers in the frame.
  - Registers %ebx, %esi, %edi are saved if they are used.
  - Can use either movl if Step-2 used, or just use pushl

On return from a call
- **Step-1:** Put return value in register %eax.
  
  (if value is returned)
- **Step-2:** Restore callee-saved registers.
  - Restore %ebx, %esi, %edi if needed
- **Step-3:** “Pop” the stack
  
  ```
  leave  
  ```
  - Equivalent to
    
    ```
    movl %ebp, %esp  
    popl %ebp  
    ```
- **Step-4:** Return
  - ```
    ret  
    # %eip = M[esp]; %esp = %esp - 4  
    ```
Outline

- Calling Conventions
- Procedure Calls
- Recursion
- Examples

Revisiting `swap`

```c
int course1 = 15213;
int course2 = 18243;

void call_swap() {
    swap(&course1, &course2);
}

void swap(int *xp, int *yp) {
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

Calling `swap` from `call_swap`:

```assembly
subl $8, %esp
movl $course2, 4(%esp)
movl $course1, (%esp)
call swap
```

Resulting Stack:

```
%esp  %esp  %esp  %esp
%esp  %esp  %esp  %esp

$course2
%course1

subl call
```
Revisiting swap

```c
void swap(int *xp, int *yp) {
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

### swap Setup #1

**Entering Stack**

- `%ebp`
- `%esp`
- `%course2`
- `%course1`
- `Rtn adr`

**Resulting Stack**

- `%ebp`
- `%esp`
- `yp`
- `xp`
- `Rtn adr`
- `Old %ebp`

### swap

```assembly
swap:  
pushl %ebp  
movl %esp, %ebp  
pushl %ebx  
                
movl 8(%ebp), %edx  
movl 12(%ebp), %ecx  
movl (%edx), %ebx  
movl (%ecx), %eax  
movl %eax, (%edx)  
movl %ebx, (%ecx)  

popl %ebx  
popl %ebp  
ret
```
**swap Setup #2**

Entering Stack

- %ebp
- %esp
- %course2
- %course1
- Rtn adr

Resulting Stack

- %ebp
- %esp
- yp
- xp
- Rtn adr
- Old %ebp
- Old %ebx

**swap**

- pushl %ebp
- movl %esp, %ebp
- pushl %ebx

**swap Setup #3**

Entering Stack

- %ebp
- %esp
- %course2
- %course1
- Rtn adr

Resulting Stack

- %ebp
- %esp
- yp
- xp
- Rtn adr
- Old %ebp
- Old %ebx
### swap Body

**Entering Stack**
- %ebp
- Offset relative to %ebp
- &course2
- &course1
- Rtn adr

**Resulting Stack**
- %ebp
- Offset relative to %ebp
- &course2
- &course1
- Rtn adr

```
movl 8(%ebp),%edx  # get xp
movl 12(%ebp),%ecx  # get yp
...```

### swap Finish

**Stack Before Finish**
- %ebp
- Offset relative to %ebp
- &course2
- &course1
- Rtn adr
- Old %ebp
- Old %ebx

**Resulting Stack**
- %ebp
- Offset relative to %ebp
- &course2
- &course1
- Rtn adr
- Old %ebp
- Old %ebx

- **Observation**
  - Saved and restored register %ebx
  - Not so for %eax, %ecx, %edx
Simple Assembly Procedure Call Examples

- Return sum of two arguments
- Return pointer to a string (string is declared in .s file)

Sum array: revisited

Task: sum together the integers stored in memory

.text
.globl sum

sum:
# Fill in what goes here

.data
num_array: .long 35, 16, 42, 19, 55, 91, 24, 61, 53
C Function call with one parameter

#include <stdio.h>
#include <stdlib.h>

// declare the function as externally defined
// computes the sum of elements 0 to i of an array defined in sum_array
extern int sum_array(int i);

int main(void) {
    int result;
    result = sum_array(7);
    printf("Array sum = %d\n",result);
    return EXIT_SUCCESS;
}

Sample Function

.text
.globl sum_array

sum_array:
    pushl %ebp
    movl %esp, %ebp
    movl 8(%ebp), %eax
    leal num_array, %edx
    leal (%edx,%eax,4), %ecx
    movl $0, %eax
    jmp loop

loop:
    addl (%edx), %eax
    addl $4, %edx
    cmpl %ecx, %edx
    jbe loop
    leave

.data
.num_array: .long 35, 16, 42, 19, 55, 91, 24, 61, 53
Outline

- Calling Conventions
- Procedure Calls
- Recursion
- Examples

### Recursive Function

```c
/* Recursive popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else return (x & 1) + pcount_r(x >> 1);
}
```

**Registers**

- `%eax`, `%edx` used without first saving
- `%ebx` used, but saved at beginning & restored at end

```asm
pcount_r:
pushl %ebp
movl %esp, %ebp
pushl %ebx
subl $4, %esp
movl 8(%ebp), %ebx
movl $0, %eax
testl %ebx, %ebx
je .L3
movl %ebx, %eax
shr %eax
movl %eax, (%esp)
call pcount_r
movl %ebx, %edx
andl $1, %edx
leal (%edx,%eax), %eax
.L3:
addl $4, %esp
popl %ebx
popl %ebp
ret
```
Recursive Call #1

/* Recursive popcount */
int pcount_r(unsigned x) {
  if (x == 0)
    return 0;
  else return
    (x & 1) + pcount_r(x >> 1);
}

- Actions
  - Save old value of %ebx on stack
  - Allocate space for argument to recursive call
  - Store x in %ebx

Recursive Call #2

/* Recursive popcount */
int pcount_r(unsigned x) {
  if (x == 0)
    return 0;
  else return
    (x & 1) + pcount_r(x >> 1);
}

- Actions
  - If x == 0, return
    - with %eax set to 0
/* Recursive popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else return (x & 1) + pcount_r(x >> 1);
}

■ Actions
  ▪ Store x >> 1 on stack
  ▪ Make recursive call

■ Effect
  ▪ %eax set to function result
  ▪ %ebx still has value of x

• • •
  movl %ebx, %eax
  shrl %eax
  movl %eax, (%esp)
call pcount_r
  • • •

Recursi ve Call #4
/* Recursive popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else return (x & 1) + pcount_r(x >> 1);
}

■ Assume
  ▪ %eax holds value from recursive call
  ▪ %ebx holds x

■ Actions
  ▪ Compute (x & 1) + computed value

■ Effect
  ▪ %eax set to function result

• • •
  movl %ebx, %edx
  andl $1, %edx
  leal (%edx,%eax), %eax
  • • •
Recursive Call #5

```c
/* Recursive popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + pcount_r(x >> 1);
}
```

**Actions**
- Restore values of %ebx and %ebp
- Restore %esp

**Observations About Recursion**

**Handled Without Special Consideration**
- Stack frames mean that each function call has private storage
  - Saved registers & local variables
  - Saved return pointer
- Register saving conventions prevent one function call from corrupting another’s data
- Stack discipline follows call / return pattern
  - If P calls Q, then Q returns before P
  - Last-In, First-Out

**Also works for mutual recursion**
- P calls Q; Q calls P
Outline

- Calling Conventions
- Procedure Calls
- Recursion
- Examples

The C code

```c
#include <stdio.h>
#include <stdlib.h>
extern int sum_i_sqr();
int main(void) {
    int result;
    result = sum_i_sqr(100);
    printf("Sum = %d\n", result);
    return EXIT_SUCCESS;
}

int sum_i_sqr(int max) {
    int i;
    int sum = 0;
    for(i=0; i <= 100; i++)
        sum = sum + i*i;
    return(sum);    // put sum into %eax
}
```

Let's write the assembly ... :)

Write a recursive version of this function
Fibonacci Number

F (n) = 0 if n = 0
      = 1 if n = 1
      = F(n-1) + F(n-2) if n > 1;

Find a node in a linked list

struct node {
  int me;
  struct node *next;
};
extern int asm_find_it(int key, struct node *ptr);

- Write recursive asm_find_it