CS24: Introduction to Computing Systems

Spring 2015

Lecture 6

Last Time: CDECL

- How to implement basic C abstractions in IA32?
  - C subroutines with arguments, local/global variables
- Began discussing the *cdecl* calling convention
  - Widely used on *NIX systems running on x86
- Both the procedure caller and the callee have to coordinate the operation!
  - Shared resources: the stack, the register file
- Calling convention specifies:
  - Who sets up which parts of the call
  - What needs to be saved, and by whom
  - How to return values back to the caller
  - Who cleans up which parts of the call

CDECL Cheat Sheet (1)

- Caller pushes arguments in <u>reverse</u> order
- Caller uses **`call`** to invoke subroutine
- Callee pushes caller's **`%ebp`** onto stack, then sets **`%ebp`** = **`%esp`**

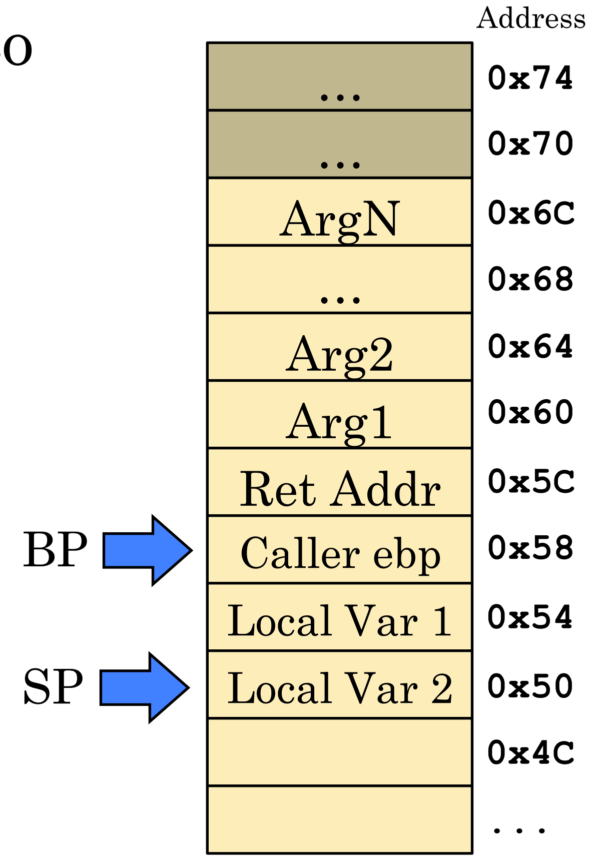
```
pushl    %ebp
movl     %esp, %ebp
```

- Arguments are at positive offsets from **`%ebp`**

  `8(%ebp)` = Arg1
  `12(%ebp)` = Arg2

- Local variables at negative offsets from **`%ebp`**

  `-4(%ebp)` = Local Var 1
  `-8(%ebp)` = Local Var 2

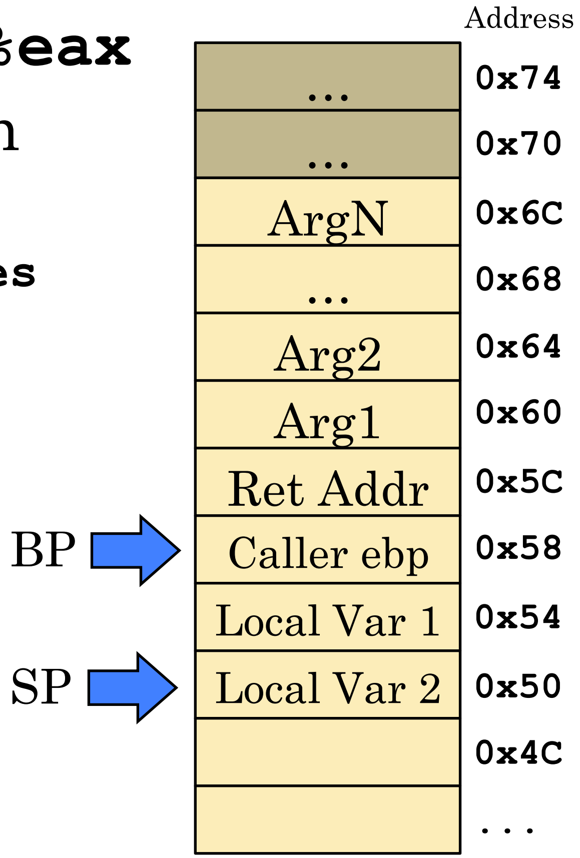


CDECL Cheat Sheet (2)

- Caller-save registers: **`%eax`**, **`%ecx`**, **`%edx`**
- Callee-save registers: **`%ebp`**, **`%ebx`**, **`%esi`**, **`%edi`**
- Callee saves return-value into **`%eax`**
- Callee restores stack state, then uses **`ret`** to return

```
# Also discards local variables
movl %ebp, %esp
popl %ebp
ret
```

- Caller removes arguments from the stack
  - Either using **`pop`** instructions, or by adding a constant to **`%esp`**



Back to Our Example C Program

- A simple accumulator:
- Uses a global variable to store current value
- Functions to update accumulator, or reset it
- Main function to exercise the accumulator
- *How is this program implemented in IA32?*

```
int value;

int accum(int n) {
  value += n;
  return value;
}

int reset() {
  int old = value;
  value = 0;
  return old;
}

int main() {
  int i, n;

  reset();
  for (i = 0; i < 10; i++) {
      n = rand() % 1000;
      printf("n = %d\taccum = %d\n",
             n, accum(n));
  }
  return 0;
}
```

Our Example Program

- Can look at **`gcc`** assembly language output for our accumulator example
  - **`gcc -O2 -S amain.c`**
  - **`-S`** generates assembly output, not a binary file
    - Result is in **`amain.s`**
  - **`-O2`** applies some optimizations to generated code
    - Otherwise, assembly output includes some pretty silly code

- Results vary *widely* based on target platform!
  - We will look at Linux **`gcc`** output
  - (MacOS X output is *very* different… Ask if you want an explanation of what's going on. It's very cool!)

Generated Assembly Code

- Some of the output:
- Lines starting with . are assembler directives
  - e.g. **`.text`** tells assembler to generate machine code for instructions that follow
- Lines with a colon are labels
  - e.g. **`accum`**, **`reset`** are labels specifying start of our functions
- **`.size`** directive specifies the size of various symbols, in bytes
  - **`accum`** = address of function's start
  - **`.`** = current address
  - **`.-accum`** is size of fn. body

```
.file   "amain.c"
        .text
        .p2align 4,,15
.globl accum
        .type   accum, @function
accum:
        pushl   %ebp
        movl    %esp, %ebp
        movl    8(%ebp), %eax
        addl    value, %eax
        popl    %ebp
        movl    %eax, value
        ret
        .size   accum, .-accum
        .p2align 4,,15
.globl reset
        .type   reset, @function
reset:
        pushl   %ebp
        movl    value, %eax
...
```

Global Variables

- End of our output:
- **`.size main, .-main`** is end of **`main()`** function

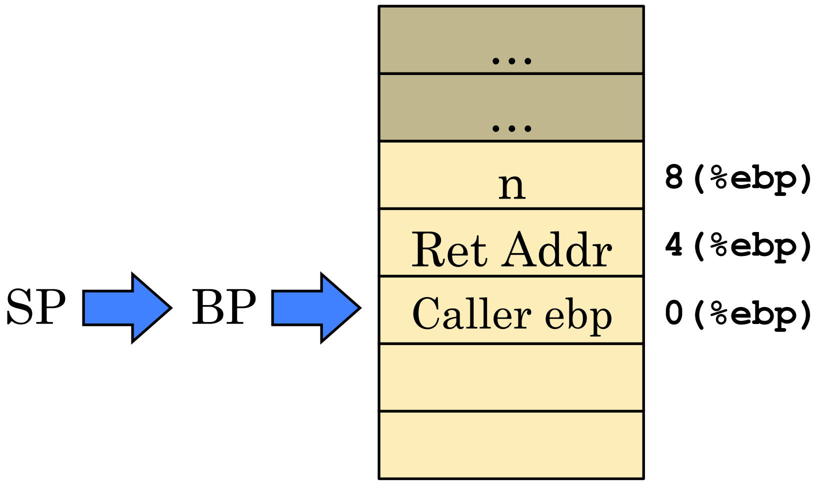
```
...
    popl    %esi
    popl    %ebp
    leal    -4(%ecx), %esp
    ret
    .size   main, .-main
    .comm   value,4,4
    .ident  "GCC: (GNU) ...
    .section         ...
```

- Global variable **`value`** specified with **`.comm`** directive
  - A “common symbol,” possibly shared across multiple files
  - Specifies name, size, optional alignment of variable
  - Address is assigned when assembling the code
  - The actual memory is uninitialized

Accumulator Code

- Accumulator function:

```
int accum(int n) {
  value += n;
  return value;
}
```



- Translated into:

```
accum:
  pushl    %ebp             # Set up stack frame
  movl     %esp, %ebp
  movl     8(%ebp), %eax    # Move n into eax
  addl     value, %eax      # eax += value
  popl     %ebp             # Restore caller ebp
  movl     %eax, value      # Store updated value
  ret
```

Reset Code

- Reset function:

```
int reset() {
    int old = value;
    value = 0;
    return old;
}
```

- Translated into:

```
reset:
  pushl   %ebp
  movl    value, %eax    # Result goes into eax
  movl    %esp, %ebp
  popl    %ebp
  movl    $0, value
  ret
```

- Clearly involves some unnecessary steps…
  - **ebp** isn't used at all! Could reduce down to 3 instructions.

Main Function

- Main function code:

```
int main() {
  int i, n;

  reset();

  for (i = 0; i < 10; i++) {
    n = rand() % 1000;
    printf("n = %d\taccum = %d\n",
           n, accum(n));
  }

  return 0;
}
```

Main Function (2)

- Main function code:

```
int main() {
  int i, n;

  reset();
  ...
```

- Assembly code:

```
main:
   leal      4(%esp), %ecx   # Stack init:  aligns stack with
   andl      $-16, %esp      #   16-byte boundary, then
   pushl     -4(%ecx)        #   copies return-addr to TOS.
   pushl     %ebp            # Set up stack frame
   movl      %esp, %ebp
   pushl     %esi            # Callee-save registers
   xorl      %esi, %esi      # %esi is i; sets i = 0.
   pushl     %ebx
   pushl     %ecx
   subl      $12, %esp       # Alloc space for fn. args
   call      reset           # Clear accumulator value
 ...
```

Main Function (3)

- Main function code, cont.

```
for (i = 0; i < 10; i++) {
  n = rand() % 1000;
  printf("n = %d\taccum = %d\n",
         n, accum(n));
}
```

- Assembly code:
  - **`esi`** is loop variable **`i`**
  - **`ebx`** is **`n`**
  - **`.L6`** is start of loop
- **`rand() % 1000`** implemented in a *very* unintuitive way…
  - Integer division/modulus with a constant can be implemented as multiplication
  - (See the book Hacker's Delight)

```
.L6:
        call    rand
        movl    $274877907, %edx
        addl    $1, %esi
        movl    %eax, %ecx
        imull   %edx
        movl    %ecx, %eax
        sarl    $31, %eax
        movl    %ecx, %ebx
        sarl    $6, %edx
        subl    %eax, %edx
        imull   $1000, %edx, %edx
        subl    %edx, %ebx
        movl    %ebx, (%esp)
        call    accum
        movl    %ebx, 4(%esp)
        movl    $.LC0, (%esp)
        movl    %eax, 8(%esp)
        call    printf
        cmpl    $10, %esi
        jne     .L6
        ...
```

MAIN FUNCTION (4)

- Main function code, cont.

```
for (i = 0; i < 10; i++) {
  n = rand() % 1000;
  printf("n = %d\taccum = %d\n",
         n, accum(n));
}
```

- Calls to `accum(n)`, `printf(...)`
- Note that **gcc** doesn't explicitly push arguments onto stack!
  - Also doesn't pop off stack when done
- This is a compiler optimization
  - Why do the pushes and pops, when it can be faked? ☺
  - **gcc** allocates extra space on the stack to speed up these calls

```
.L6:
     call     rand
     movl     $274877907, %edx
     addl     $1, %esi
     movl     %eax, %ecx
     imull    %edx
     movl     %ecx, %eax
     sarl     $31, %eax
     movl     %ecx, %ebx
     sarl     $6, %edx
     subl     %eax, %edx
     imull    $1000, %edx, %edx
     subl     %edx, %ebx
     movl     %ebx, (%esp)
     call     accum
     movl     %ebx, 4(%esp)
     movl     $.LC0, (%esp)
     movl     %eax, 8(%esp)
     call     printf
     cmpl     $10, %esi
     jne      .L6
     ...
```

MAIN FUNCTION (5)

- Main function code, cont.

```
for (i = 0; i < 10; i++) {
  n = rand() % 1000;
  printf("n = %d\taccum = %d\n",
         n, accum(n));
}
```

```
.L6:
        call    rand
        movl    $274877907, %edx
        addl    $1, %esi
        movl    %eax, %ecx
        imull   %edx
        movl    %ecx, %eax
        sarl    $31, %eax
        movl    %ecx, %ebx
        sarl    $6, %edx
        subl    %eax, %edx
        imull   $1000, %edx, %edx
        subl    %edx, %ebx
        movl    %ebx, (%esp)
        call    accum
        movl    %ebx, 4(%esp)
        movl    $.LC0, (%esp)
        movl    %eax, 8(%esp)
        call    printf
        cmpl    $10, %esi
        jne     .L6
        ...
```

- Also need a string constant to pass to **`printf()`**
- Before **`main()`** code:

```
.LC0:
    .string "n = %d\taccum = %d\n"
```

  - **`as`** copies this data to the output binary file
  - Address of data is **`.LC0`**

Calling Convention and Recursion

- The cdecl calling convention:
  - Each function call has its own region of the stack
    - Caller pushes arguments onto stack, then calls the callee
    - Callee saves caller's frame pointer, then sets up its own frame pointer
    - Callee stores its local variables after the frame pointer
    - When callee returns to caller, stack is restored to prev state
- This calling convention easily supports recursion
- A procedure can call itself:
  - Each recursive invocation of the procedure will have its own stack space, as long as the conventions are followed!
- You get to explore this more on Assignment 2! ☺

C Flow-Control Statements

- C provides a variety of flow-control statements
  - **`if`** statements

    `if` (*test-expr*)
        then-statement
    `else`
        else-statement
  - **`while`** loops, **`for`** loops, **`do`** loops

    `while` (*test-expr*)
        body-statement
- Conceptually straightforward to use in your C programs
- How are these normally translated to IA32 assembly language?
  - Helps us better understand what the compiler generates
  - Also helps us know how to write them in IA32 ourselves!

C Flow-Control Statements (2)

- C flow-control statements implemented in IA32 using a combination of conditional and unconditional jumps
- Example: **`if`** statements

```
if (test-expr)
  then-statement ;
else
  else-statement ;
```

- A common translation:

```
  t = test-expr ;
  if (t)
    goto true_branch;
  else-statement ;
  goto done;
true_branch:
  then-statement ;
done:
```

- Compiler frequently optimizes/rearranges this flow

Do-While Loops

- **`do`**-loops not used as frequently in programs, but very easy to implement in assembly language
  - Requires minimum number of branching operations

```
do
   body-statement
while (test-expr);
```

- A simple translation:

```
loop:
   body-statement ;
   t = test-expr ;
   if (t)
     goto loop;
```

WHILE LOOPS

- **`while`**-loops are much more common, but more involved at the assembly-language level

  `while` (*test-expr*)
     body-statement ;

- One translation:

```
loop:
   t = test-expr ;
   if (!t)
      goto done;
   body-statement ;
   goto loop;
done:
```

- Problem: This code is slow to execute.
- Branching has a *big* performance impact
  - Affects instruction caching and pipelining

WHILE LOOPS (2)

- A faster implementation "peels off" the first test

```
while (test-expr)
  body-statement ;
```

  - Equivalent to:

```
if (!test-expr)
  goto done;
do
  body-statement ;
while (test-expr)
done:
```

  - Translating this to assembly code yields a loop body containing only one branching operation

While Loops (3)

- Our original while-loop:

  `while` (*test-expr*)
     body-statement ;

- Completing the translation:

```
    t = test-expr ;
    if (!t)
      goto done;
  loop:
    body-statement ;
    t = test-expr ;
    if (t)
      goto loop;
  done:
```

For Loops

- **`for`**-loops are more sophisticated **`while`** loops:

  `for` (*init-expr* ; *test-expr* ; *update-expr*)
     body-statement ;

  - Equivalent to:

    init-expr ;
    `while` (*test-expr*) **{**
       body-statement ;
       update-expr ;
    }

- We know how to translate these components into assembly language
  - Transform **`while`** loop into **`do-while`** loop
  - Insert additional **`for`**-loop operations into appropriate places

For Loops (2)

- Implementing **`for`** loops:

    `for` (*init-expr* ; *test-expr* ; *update-expr*)
    body-statement ;

- Translate into:

```
    init-expr ;
    t = test-expr ;
    if (!t)
      goto done;
  loop:
    body-statement ;
    update-expr ;
    t = test-expr ;
    if (t)
      goto loop;
  done:
```

More Advanced Programs…

- We can now map basic C programs into IA32 assembly language
  - …including programs that use recursion
- What about this problem?
  - Write a function that takes an argument *n*
  - Return a collection of all prime numbers ≤ *n*
  - e.g. **`int * find_primes(int n)`**
- Don't yet have the tools to implement this!
  - Requires a variable amount of memory
  - Memory lifetime must extend beyond a single function call

Dynamic Allocation and Heap

- When programs need a variable amount of memory, they can allocate it from the heap
  - A large, resizable pool of memory for programs to use
  - Programs can request blocks of memory from the heap
  - When finished, programs release blocks back to the heap
  - This is a *run-time facility* for programs to utilize
- For C programs, standard functions to support heap:
  - Allocate a block of memory using **`malloc()`**
    - **`void * malloc(size_t size)`**
    - Returns pointer to block of memory with specified size, or **`NULL`** if **`size`** bytes are not available
    - **`size_t`** is an unsigned integer data type
  - Release a block of memory using **`free()`**
    - **`void free(void *ptr)`**
    - Specified memory block is returned to heap

HEAP AND STACK

- Programs and stack usage:
  - Stack space automatically reclaimed when function returns
    - Stack values can last for up to the lifetime of a procedure call
  - Procedures are specifically encoded to use the stack
    - Explicit accesses relative to base pointer **`ebp`**
    - Adjustments to stack pointer to allocate/release space
  - Stack space used by a procedure doesn't vary substantially during its execution
    - Set of local variables within each code block is fixed
    - Set of arguments passed to a procedure is also fixed
- Programs and heap usage:
  - Memory required often depends on the input values
    - Can vary quite dramatically!
  - Programs must *explicitly* allocate and release blocks of memory on the heap

Simple Example: Vector Addition

- Variation on an example from lecture 3:

```
int * vector_add(int a[], int b[], int length) {
    int *result;
    int i;

    result = malloc(length * sizeof(int));
    for (i = 0; i < length; i++)
        result[i] = a[i] + b[i];

    return result;
}
```

  - Now the procedure dynamically allocates a result vector from heap, then sums inputs into this memory
- Now that we understand IA32 more deeply, let's explore how to implement this function

Pointers and Arrays

- Our vector-add function:

```
int * vector_add(int a[], int b[], int length) {
    int *result;
    int i;

    result = (int *) malloc(length * sizeof(int));
    for (i = 0; i < length; i++)
        result[i] = a[i] + b[i];

    return result;
}
```

- Clear that pointers and arrays are closely related
  - Declare **`result`** as **`int*`**
  - Access it as an array, just like **`a`** and **`b`**

Arrays in C

- C arrays are collections of elements, all of same data type
  - Array elements are contiguous in memory
  - Elements are accessed by indexing into the array
- For an array declaration: ***`T`*** **`A[`*****`N`*****`]`**
  - ***`T`*** is the data type
  - **`A`** is the array's variable name
  - ***`N`*** is the number of elements
- C allocates a contiguous region of memory for the array
  - Allocates ***`N`*** × **`sizeof(`*****`T`*****`)`** bytes for the array
  - **`sizeof(`*****`type`*****`)`** is a standard C operator that returns the size of the specified data type, in bytes
    - e.g. **`sizeof(int)`** = 4 for **`gcc`** on IA32
  - **`sizeof(`*****`type`*****`)`** is resolved to a value at compile-time
- **`A`** holds a <u>pointer</u> to the start of the array
  - Can use **`A`** to access various elements of the array

ARRAYS IN C (2)

- For an array declaration: ***`T`*** **`A[`*****`N`*****`]`**
  - ***`T`*** is the data type
  - **`A`** is the array's variable name
  - ***`N`*** is the number of elements
- **`A`** holds a pointer to the start of the array
  - Can use **`A`** to access various elements of the array
- What address does array-element *i* reside at?
  - **`A`** points to first element in array
  - Each element is **`sizeof(`*****`T`*****`)`** bytes in size
  - Element *i* resides at address **`A + sizeof(`*****`T`*****`) * i`**
- This is what the C array-index operator **`[]`** does
  - **`A[i]`** computes index of element *i*, then reads/writes the element

Pointers and Arrays in C

- C also supports pointer arithmetic
  - Similar idea to array indexing
- For a C pointer variable: **`T *p`**
- Adding 1 to **`p`** advances it one *element*, not one byte
  - e.g. for **`int *p`**, **`p + 1`** actually advances **`p`** by 4 bytes
- Adding/subtracting an offset *N* from a pointer to **`T`** will move forward or backward *N* elements
- Implication:
  - **`A[i]`** is identical to saying **`*(A + i)`**
  - **`A`** is a pointer to first element in array
  - **`A + i`** moves forward **`i`** elements (*not* **`i`** bytes!)
  - **`A + i`** is a *pointer* to element **`i`**, so dereference to get to the actual element **`A[i]`**

Variation on Theme

- Our original function:

```
int * vector_add(int a[], int b[], int length) {
  int *result;
  int i;

  result = (int *) malloc(length * sizeof(int));
  for (i = 0; i < length; i++)
    result[i] = a[i] + b[i];

  return result;
}
```

- Could also write something crazy like this:

```
int * vector_add(int *a, int *b, int length) {
  int *result, *elem;

  result = (int *) malloc(length * sizeof(int));
  for (elem = result; length != 0; length--) {
    *elem = *a + *b;
    elem++; a++; b++;
  }
  return result;
}
```

- Optimizing compilers may generate code like this
- Take advantage of C's equivalence between arrays and pointers

VECTOR-ADD AND IA32

- How do we implement this function in IA32?

```
int * vector_add(int *a, int *b, int length) {
  int *result;
  int i;

  result = (int *) malloc(length * sizeof(int));
  for (i = 0; i < length; i++)
    result[i] = a[i] + b[i];

  return result;
}
```

- Next time, will go through the entire process of implementing this, from scratch. ☺