The Hardware/Software Interface
CSE351 Winter 2013

Procedures and Stacks I
Roadmap

C:

```c
#include <stdio.h>

int main() {
    Car c = new Car();
    c.setMiles(100);
    c.setGals(17);
    float mpg = c.getMPG();
    free(c);
    return 0;
}
```

Java:

```java
Car c = new Car();
c.setMiles(100);  
c.setGals(17);  
float mpg = c.getMPG();
```

Assembly language:

```
get_mpg:
    pushq    %rbp
    movq     %rsp, %rbp
    ...
    popq     %rbp
    ret
```

Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111110100001111
```

Computer system:

OS:

Windows 8, Mac, and Linux.

Data & addressing
Integers & floats
Machine code & C
x86 assembly programming
Procedures & stacks
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C

Winter 2013

Procedures and Stacks I
Procedures and Call Stacks

- How do I pass arguments to a procedure?
- How do I get a return value from a procedure?
- Where do I put local variables?
- When a function returns, how does it know where to return to?

- To answer these questions, we need a call stack ...
Memory Layout

- **Stack**: local variables; procedure context
- **Dynamic Data (Heap)**: variables allocated with `new` or `malloc`
- **Static Data**: static variables (including global variables (C))
- **Literals**: literals (e.g., “example”)
- **Instructions**: 2^{N-1}

 niveaux de mémoire :

- **Instructions**
- **Literals**
- **Static Data** (y compris les variables globales (C))
- **Dynamic Data (Heap)** : variables allouées avec `new` ou `malloc`
- **Stack** : variables locales ; contexte de la procédure
Memory Layout

- **Stack**
  - Managed “automatically” (by compiler)
  - writable; not executable

- **Dynamic Data (Heap)**
  - Managed by programmer
  - writable; not executable

- **Static Data**
  - Initialized when process starts
  - writable; not executable

- **Literals**
  - Initialized when process starts
  - Read-only; not executable

- **Instructions**
  - Initialized when process starts
  - Read-only; executable
IA32 Call Stack

- Region of memory managed with a stack “discipline”
- Grows toward lower addresses
- Customarily shown “upside-down”

- Register %esp contains lowest stack address
  = address of “top” element

Stack Pointer: %esp

Stack "Bottom"

Stack Grows Down

Increasing Addresses

Stack "Top"
IA32 Call Stack: Push

- pushl $src$

**Stack Pointer:** $esp$

**Stack “Top”**

**Stack “Bottom”**

Increasing Addresses

Stack Grows Down
IA32 Call Stack: Push

- `pushl Src`
  - Fetch value from `Src`
  - Decrement `%esp` by 4 (*why 4?*)
  - Store value at address given by `%esp`

**Stack Pointer:** `%esp`

**Stack “Bottom”**

- Increasing Addresses
- Stack Grows Down

**Stack “Top”**

Winter 2013

Procedures and Stacks I
IA32 Call Stack: Pop

- popl Dest

Stack Pointer: %esp

Stack "Bottom"

Stack Grows Down

Increasing Addresses

Stack "Top"
IA32 Call Stack: Pop

- `popl Dest`
  - Load value from address `%esp`
  - Write value to `Dest`
  - Increment `%esp` by 4
Procedure Call Overview

- **Callee** must know where to find args
- **Callee** must know where to find “return address”
- **Caller** must know where to find return val
- **Caller** and **Callee** run on same CPU → use the same registers
  - **Caller** might need to save registers that **Callee** might use
  - **Callee** might need to save registers that **Caller** has used
The **convention** of where to leave/find things is called the **procedure call linkage**

- Details vary between systems
- We will see the convention for IA32/Linux in detail
- What could happen if our program didn’t follow these conventions?
Procedure Control Flow

- Use stack to support procedure call and return
- **Procedure call:** `call label`
  - Push return address on stack
  - Jump to `label`
Procedure Control Flow

- Use stack to support procedure call and return

- **Procedure call:** `call label`
  - Push return address on stack
  - Jump to `label`

- **Return address:**
  - Address of instruction after `call`
  - Example from disassembly:

    | Address   | Instruction                  |
    |-----------|------------------------------|
    | 804854e   | e8 3d 06 00 00 call 8048b90 <main> |
    | 8048553   | 50 pushl %eax               |

  - Return address = 0x8048553

- **Procedure return:** `ret`
  - Pop return address from stack
  - Jump to address
Procedure Call Example

```
804854e:   e8 3d 06 00 00    call 8048b90 <main>
8048553:   50
            pushl %eax
```

```
call 8048b90
```

```
0x110
0x10c
0x108  123
```

```
%esp 0x108
%eip 0x804854e
```

%eip: program counter
Procedure Call Example

804854e: e8 3d 06 00 00 call 8048b90 <main>
8048553: 50 pushl %eax

0x110 0x10c 0x108 123
%esp 0x108 %esp 0x10c
%eip 0x804854e %eip 0x804854e

call 8048b90

%eip: program counter

Winter 2013
Procedures and Stacks I
16
Procedure Call Example

804854e:   e8 3d 06 00 00    call   8048b90 <main>
8048553:   50                  pushl  %eax

Procedure Call Example

call   8048b90

program counter
%esp 0x108
%esp 0x108
%eip 0x804854e
%eip 0x8048553

Winter 2013
Procedures and Stacks I
Procedure Call Example

```
804854e:       e8 3d 06 00 00    call   8048b90 <main>
8048553:       50          pushl  %eax
```

call   8048b90

```
0x110
0x10c
0x108   123
%esp   0x108
%eip   0x804854e
```

```
0x110
0x10c
0x108   123
0x104   0x8048553
%esp   0x104
%eip   0x8048553
```

*eip: program counter*
Procedure Call Example

```
804854e:   e8 3d 06 00 00   call 8048b90 <main>
8048553:   50   pushl %eax
```

call 8048b90

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x110</td>
<td>123</td>
</tr>
<tr>
<td>0x10c</td>
<td>123</td>
</tr>
<tr>
<td>0x108</td>
<td>0x8048553</td>
</tr>
<tr>
<td>%esp</td>
<td>0x108</td>
</tr>
<tr>
<td>%eip</td>
<td>0x804854e</td>
</tr>
</tbody>
</table>

%eip: program counter
Procedure Return Example

8048591: c3 ret

%esp 0x104
%eip 0x8048591

ret

0x110
0x10c
0x108 123
0x104 0x8048553

%esp 0x104
%eip 0x8048591

%eip: program counter
Procedure Return Example

8048591: c3 ret

%esp %eip
0x104 0x8048591
0x108 123
0x10c 0x8048553
0x110

%esp %eip
0x104 0x8048591
0x108 123
0x10c 0x8048553
Procedure Return Example

8048591: c3 ret

%esp %eip
0x104 0x8048591
0x108 0x10c 123
0x104 0x8048553

%esp 0x104
%eip 0x8048591

ret
0x110 0x10c 123
0x108 0x10c 123
0x104 0x8048553

%esp 0x104
%eip 0x8048553

%eip: program counter
Procedure Return Example

8048591:  c3  ret

%esp  0x104
%eip  0x8048591
0x108  123
0x10c  
0x110  
0x104  0x8048553

ret

%esp  0x104
%eip  0x8048591
0x108  123
0x10c  
0x110  
0x104  0x8048553

ret

%esp  0x108
%eip  0x8048553

%eip: program counter
Stack-Based Languages

- Languages that support recursion
  - e.g., C, Pascal, Java
  - Code must be *re-entrant*
    - Multiple simultaneous instantiations of single procedure
  - Need some place to store state of each instantiation
    - Arguments
    - Local variables
    - Return pointer

- Stack discipline
  - State for a given procedure needed for a limited time
    - Starting from when it is called to when it returns
  - Callee always returns before caller does

- Stack allocated in *frames*
  - State for a single procedure instantiation
Call Chain Example

yoo(...)
{
  ...
  who();
  ...
}

who(...)
{
  ...
  amI();
  ...
}

amI(...)
{
  ...
  amI();
  ...
}

Example Call Chain

Procedure amI is recursive (calls itself)
Stack Frames

Contents
- Local variables
- Function arguments
- Return information
- Temporary space

Management
- Space allocated when procedure is entered
  - “Set-up” code
- Space deallocated upon return
  - “Finish” code
Example

```c
yoo(...) {
    who();
}
```

Stack

```
%ebp
%esp
```

```c
yoo
who
amI
amI
```

```
```
```
```
Example

```c
who(...) {
  ...
  amI();
  ...
  amI();
  ...
}
```

Stack

```
%ebp
%esp
```

```
Stack
```
Example

```
amI(...) {
  ...
  ...
  amI();
  ...
  ...
}
```

Stack

- `yoo`
- `who`
- `amI`
- `%ebp`
- `%esp`

Winter 2013
Example

```c
amI(...) {
    
    amI();
    
}
```

Stack

```c
yoo

who

amI

amI

%ebp

%esp

amI

amI

amI

amI
```
Example

```
ami(...) {
    •
    •
    ami();
    •
}
```

Stack

```
%ebp
%

ami
who
yoo
```
Example

```c
amI(...) {
  ...
  ...
  amI();
  ...
}
```

Stack

```
yoo
who
amI
amI
%ebp
%esp
```
Example

```c
ami(...) {
  
  
  ami();
  
}
```

Stack

- yoo
- who
- ami
- %ebp
- %esp
Example

who(...) {
    • • •
    amI();
    • • •
    amI();
    • • •
}

Stack

who

yoo

%ebp

%esp

amI

amI

amI

amI
Example

```
amI(...) {
  ...
  ...
  ...
  ...
}
```

Stack

```
%ebp

%esp

amI

who

yoo
```

Procedures and Stacks I
Example

```c
who(...) {
    • • •
    amI();
    • • •
    amI();
}
```

Stack

```
Stack
  | yoo
  | %ebp
  | %esp
  | who
```
Example

```c
yoo(...) {
  •
  •
  who();
  •
  •
}
```

Stack

```
%ebp

yoo

%esp
```

Winter 2013
IA32/Linux Stack Frame

- **Current Stack Frame ("Top" to Bottom)**
  - "Argument build" area (parameters for function about to be called)
  - Local variables (if can’t be kept in registers)
  - Saved register context (when reusing registers)
  - Old frame pointer (for caller)

- **Caller’s Stack Frame**
  - Return address
    - Pushed by `call` instruction
  - Arguments for this call
Revisiting swap

```c
int zip1 = 15213;
int zip2 = 98195;

void call_swap()
{
    swap(&zip1, &zip2);
}

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

```c
int zip1 = 15213;
int zip2 = 98195;

void call_swap()
{
    swap(&zip1, &zip2);
}
```

Calling swap from call_swap

```c
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
call_swap:
    . . .
    pushl $zip2    # Global Var
    pushl $zip1    # Global Var
    call swap
        . . .
```
Revisiting swap

```c
int zip1 = 15213;
int zip2 = 98195;

void call_swap()
{
    swap(&zip1, &zip2);
}

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

Calling swap from call_swap

```c
call_swap:
    • • •
    pushl $zip2  # Global Var
    pushl $zip1  # Global Var
    call swap
    • • •
```

Resulting Stack

```
•
•
•
&zip2
&zip1
Rtn adr
%esp
```
Revisiting swap

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

```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx

    movl 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax,(%edx)
    movl %ebx,(%ecx)

    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```

Set Up

Body

Finish
swap Setup #1

Entering Stack

\[
\begin{array}{c}
\text{\%ebp} \\
\text{\%esp} \\
\text{\%ebp} \\
\text{\%ebp} \\
\text{\&zip2} \\
\text{\&zipl} \\
\text{Rtn adr}
\end{array}
\]

Resulting Stack?

```
swap:
pushl %ebp
movl %esp,%ebp
pushl %ebx
```
swap Setup #1

Entering Stack

Resulting Stack

swap:

```assembly
pushl %ebp
movl %esp,%ebp
pushl %ebx
```
swap Setup #2

Entering Stack

Resulting Stack

\[
\text{swap:} \\
\text{pushl } \%ebp \\
\text{movl} \%esp, \%ebp \\
\text{pushl } \%ebx
\]
swap Setup #3

Entering Stack


diagram showing %ebp and %esp

Resulting Stack


diagram showing YP, xp, Rtn adr, Old %ebp, Old %ebx

swap:
pushl %ebp
movl %esp,%ebp
pushl %ebx
swap Body

Entering Stack

Resulting Stack

Offset relative to new %ebp

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

...
swap Finish #1

swap’s Stack

Resulting Stack?

movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
**swap Finish #1**

---

**swap’ s Stack**

- •
- •
- •
- 
- YP
- xp
- Rtn adr
- Old %ebp
- Old %ebx

**Resulting Stack**

- •
- •
- •
- 
- YP
- xp
- Rtn adr
- Old %ebp
- Old %ebx

```assembly
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

**Observation:** Saved and restored register %ebx
swap Finish #2

swap’s Stack

Resulting Stack

movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
swap Finish #3

swap′ s Stack

Resulting Stack

movl \(-4 (%ebp), %ebx\)
movl %ebp, %esp
popl %ebp
ret
swap Finish #4

swap’s Stack

Resulting Stack

movl $-4(%ebp),%ebx  
movl %ebp,%esp  
popl %ebp  
ret
Disassembled swap

080483a4 <swap>:

```
080483a4:   55          push   %ebp
080483a5:   89 e5       mov    %esp,%ebp
080483a7:   53          push   %ebx
080483a8:   8b 55 08    mov    0x8(%ebp),%edx
080483ab:   8b 4d 0c    mov    0xc(%ebp),%ecx
080483ae:   8b 1a       mov    (%edx),%ebx
080483b0:   8b 01       mov    (%ecx),%eax
080483b2:   89 02       mov    %eax,(%edx)
080483b4:   89 19       mov    %ebx,(%ecx)
080483b6:   5b          pop    %ebx
080483b7:   c9          leave
080483b8:   c3          ret
```

Calling Code

```
8048409:   e8 96 ff ff ff ff   call 80483a4 <swap>
804840e:   8b 45 f8       mov    0xfffffffff8(%ebp),%eax
```

`0x0804840e + 0xffffffff96 = 0x080483a4`
swap Finish #4

Observation
- Saved & restored register `%ebx`
- Didn’t do so for `%eax`, `%ecx`, or `%edx`
Register Saving Conventions

- When procedure `yoo` calls `who`:  
  - `yoo` is the *caller*  
  - `who` is the *callee*

- Can a register be used for temporary storage?

  **yoo:**
  ```
  • • •
  movl $12345, %edx
  call who
  addl %edx, %eax
  • • •
  ret
  ```

  **who:**
  ```
  • • •
  movl 8(%ebp), %edx
  addl $98195, %edx
  • • •
  ret
  ```

- Contents of register `%edx` overwritten by `who`
Register Saving Conventions

- When procedure **\texttt{you}** calls **\texttt{who}**:  
  - **\texttt{you}** is the \textit{caller}  
  - **\texttt{who}** is the \textit{callee}

- Can a register be used for temporary storage?

- Conventions
  - \textit{“Caller Save”}  
    - Caller saves temporary values in its frame before calling  
  - \textit{“Callee Save”}  
    - Callee saves temporary values in its frame before using
IA32/Linux Register Usage

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

- **%eax**
  - also used to return integer 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
Example: Pointers to Local Variables

Recursive Procedure

```c
void s_helper (int x, int *accum)
{
    if (x <= 1)
        return;
    else {
        int z = *accum * x;
        *accum = z;
        s_helper (x-1,accum);
    }
}
```

Top-Level Call

```c
int sfact(int x)
{
    int val = 1;
    s_helper(x, &val);
    return val;
}
```

- Pass pointer to update location
Creating & Initializing Pointer

```c
int sfact(int x)
{
    int val = 1;
    s_helper(x, &val);
    return val;
}
```

- Variable `val` must be stored on stack
  - Because: Need to create pointer to it
- Compute pointer as `-4 (%ebp)`
- Push on stack as second argument

**Initial part of `sfact`**

```
_sfact:
    pushl %ebp        # Save %ebp
    movl %esp,%ebp    # Set %ebp
    subl $16,%esp     # Add 16 bytes
    movl 8(%ebp),%edx # edx = x
    movl $1,-4(%ebp)  # val = 1
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>-16</td>
<td>Unused</td>
</tr>
<tr>
<td>-12</td>
<td>Unused</td>
</tr>
<tr>
<td>-8</td>
<td>val = 1</td>
</tr>
<tr>
<td>0</td>
<td>Old %ebp</td>
</tr>
<tr>
<td>4</td>
<td>Rtn adr</td>
</tr>
<tr>
<td>8</td>
<td>x</td>
</tr>
</tbody>
</table>

- %esp
- %ebp
- %esp
- %esp
Passing Pointer

```c
int sfact(int x) {
    int val = 1;
    s_helper(x, &val);
    return val;
}
```

- **Variable val must be stored on stack**
  - Because: Need to create pointer to it
- **Compute pointer as \(-4 (\%ebp)\)**
- **Push on stack as second argument**

**Calling s_helper from sfact**

```assembly
leal -4(\%ebp),\%eax  # Compute &val
pushl %eax             # Push on stack
pushl %edx             # Push x
    call s_helper      # call
movl -4(\%ebp),\%eax  # Return val
    ...             # Finish
```

**Stack at time of call:**

```
<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>Rtn adr</th>
<th>Old %ebp</th>
<th>val=x!</th>
<th>Unused</th>
<th>%ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td></td>
<td></td>
<td>0</td>
<td>-4</td>
<td>-12</td>
<td>-16</td>
</tr>
</tbody>
</table>
```

Winter 2013
IA 32 Procedure Summary

Important points:
- IA32 procedures are a combination of instructions and conventions
  - Conventions prevent functions from disrupting each other
  - Stack is the right data structure for procedure call / return
    - If P calls Q, then Q returns before P
- Recursion handled by normal calling conventions
  - Can safely store values in local stack frame and in callee-saved registers
  - Put function arguments at top of stack
  - Result returned in %eax

Recursion handled by normal calling conventions

- Can safely store values in local stack frame and in callee-saved registers
- Put function arguments at top of stack
- Result returned in %eax

Call Stack Frame

- Arguments
- Return Address
- Saved Registers + Local Variables
- Argument Build

%ebp -> %esp

Winter 2013