Machine-Level Programming III: Procedures
Sept 18, 2001

Topics
• IA32 stack discipline
• Register saving conventions
• Creating pointers to local variables
IA32 Stack

- Region of memory managed with stack discipline
- Register %esp indicates lowest allocated position in stack
  - i.e., address of top element

Pushing
- pushl Src
  - Fetch operand at Src
  - Decrement %esp by 4
  - Write operand at address given by %esp

Popping
- popl Dest
  - Read operand at address given by %esp
  - Increment %esp by 4
  - Write to Dest
Stack Operation Examples

\[
\begin{array}{c|c|c}
0x110 & 0x10c & 0x108 \\
0x110 & 0x10c & 0x108 \\
0x108 & 0x108 & 0x104 \\
\hline
0x108 & 0x108 & 0x104 \\
0x108 & 0x108 & 0x104 \\
\end{array}
\]

\[
\begin{array}{c|c|c}
\%eax & 213 & 213 \\
\%edx & 555 & 555 \\
\%esp & 0x108 & 0x104 \\
\hline
\%eax & 213 & 213 \\
\%edx & 555 & 555 \\
\%esp & 0x108 & 0x104 \\
\end{array}
\]

pushl %eax
popl %edx
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 value

- Address of instruction beyond `call`
- Example from disassembly

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

- Return address = 0x8048553

Procedure return:

- `ret` Pop address from stack; Jump to address
Procedure Call / Return Example

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

\[
\begin{align*}
\%esp & \quad 0x108 & \quad 0x110 \\
\%eip & \quad 0x804854e & \quad 0x8048553 \\
\end{align*}
\]

\[
\begin{align*}
\text{call} & \quad 8048b90 \\
\text{ret} & \\
\%esp & \quad 0x108 & \quad 0x104 \\
\%eip & \quad 0x804854e & \quad 0x8048553 \\
\end{align*}
\]

\%eip is program counter
Stack-Based Languages

Languages that Support Recursion

• e.g., C, Pascal, Java
• Code must be "Reentrant"
  – Multiple simultaneous instantiations of single procedure
• Need some place to store state of each instantiation
  – Arguments
  – Local variables
  – Return pointer

Stack Discipline

• State for given procedure needed for limited time
  – From when called to when return
• Callee returns before caller does

Stack Allocated in Frames

• state for single procedure instantiation
Call Chain Example

Code Structure

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

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

amI(...) {
  
}
```

- Procedure `amI` recursive

Call Chain

```
→ yoo
   ↓ who
   ↓ amI
   ↓ amI
   ↓ amI
```
IA32 Stack Structure

Stack Growth
- Toward lower addresses

Stack Pointer
- Address of next available location in stack
- Use register %esp

Frame Pointer
- Start of current stack frame
- Use register %ebp
IA32/Linux Stack Frame

Callee Stack Frame ("Top" to Bottom)
- Parameters for called functions
- 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
Revisiting swap

```c
#include <stdio.h>

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

int zip1 = 15213;
int zip2 = 91125;

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

call_swap:

```assembly
    pushl $zip2  # Global Var
    pushl $zip1  # Global Var
    call swap
    ...  
```

Resulting Stack

```assembly
    %esp  
        &zip2  
        &zip1  
          Rtn adr
```

Resulting Stack

```assembly
    %esp  
        &zip2  
        &zip1  
          Rtn adr
```
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
```
**swap Setup #1**

**Entering Stack**
- `%ebp`
- `&zip2`
- `&zipl`
- `Rtn adr`

**Resulting Stack**
- `%ebp`
- `yp`
- `xp`
- `Rtn adr`
- `Old %ebp`

**swap:**
- `pushl %ebp`
- `movl %esp,%ebp`
- `pushl %ebx`
swap Setup #2

Entering Stack

Resulting Stack

\[
\begin{align*}
\text{swap:} & \\
& \text{pushl } \%ebp \\
& \text{movl } \%esp,\%ebp \\
& \text{pushl } \%ebx
\end{align*}
\]
swap Setup #3

**Entering Stack**

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

- `%ebp`
- `&zip2`
- `&zip1`
- `Rtn adr`

**Resulting Stack**

- `%ebp`
- `%esp`
- `yp`
- `xp`
- `Rtn adr`
- `Old %ebp`
- `Old %ebx`
Effect of \texttt{swap} Setup

\begin{itemize}
  \item \texttt{pushl \%ebp}
  \item \texttt{movl \%esp,\%ebp}
  \item \texttt{pushl \%ebx}
\end{itemize}

\texttt{swap:}

\begin{verbatim}
  pushl \%ebp
  movl \%esp,\%ebp
  pushl \%ebx
\end{verbatim}
Observation

- Saved & restored register %ebx

```
movl -4(%ebp), %ebx
movl %ebp, %esp
popl %ebp
ret
```
swap Finish #2

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

swap’s Stack

Offset
12
8
4
0

Old %ebp

%ebp

%esp

swap’s Stack

Offset
12
8
4

Rtn adr

%esp

%ebp

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

- Saved & restored register %ebx
- Didn’t do so for %eax, %ecx, or %edx

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

When procedure \texttt{yoo} calls \texttt{who}: 

- \texttt{yoo} is the \textit{caller}, \texttt{who} is the \textit{callee}

Can Register be Used for Temporary Storage?

\begin{align*}
\texttt{yoo:} & \begin{array}{l}
\text{• • •} \\
\text{movl $15213, %edx} \\
\text{call who} \\
\text{addl %edx, %eax} \\
\text{• • •} \\
\text{ret}
\end{array} \\
\texttt{who:} & \begin{array}{l}
\text{• • •} \\
\text{movl 8(%ebp), %edx} \\
\text{addl $91125, %edx} \\
\text{• • •} \\
\text{ret}
\end{array}
\end{align*}

- Contents of register %edx overwritten by \texttt{who}

Conventions

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

• Surmised by looking at code examples

Integer Registers

• Two have special uses
  \%ebp, \%esp
• Three managed as callee-save
  \%ebx, \%esi, \%edi
  – Old values saved on stack prior to using
• Three managed as caller-save
  \%eax, \%edx, \%ecx
  – Do what you please, but expect any callee to do so, as well
• Register \%eax also stores returned value

Caller-Save Temporaries

Callee-Save Temporaries

Special
Recursive Factorial

```c
int rfact(int x)
{
    int rval;
    if (x <= 1)
        return 1;
    rval = rfact(x-1);
    return rval * x;
}
```

Complete Assembly

- Assembler directives
  - Lines beginning with “.”
  - Not of concern to us
- Labels
  - .Lxx
- Actual instructions

```assembly
.globl rfact
.type
rfact,@function
rfact:
pushl %ebp
movl %esp,%ebp
pushl %ebx
movl 8(%ebp),%ebx
cmpl $1,%ebx
jle .L78
leal -1(%ebx),%eax
pushl %eax
call rfact
imull %ebx,%eax
jmp .L79
.align 4
.L78:
movl $1,%eax
.L79:
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```
Rfact Stack Setup

Calling:

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

Changing:

```
Callee

Rtn adr

x

8

4

0

-4

Old %ebp

Old %ebx

מילה: %esp

 Caller

%ebp

Callee
Rfact Body

Recursion

movl 8(%ebp),%ebx  # ebx = x
cmp $1,%ebx        # Compare x : 1
jle .L78           # If <= goto Term
leal -1(%ebx),%eax # eax = x-1
pushl %eax         # Push x-1
call rfact         # rfact(x-1)
imull %ebx,%eax    # rval * x
jmp .L79           # Goto done

.L78:               # Term:
movl $1,%eax       # return val = 1
.L79:               # Done:

int rfact(int x)  
{                  
    int rval;
    if (x <= 1)    
        return 1;
    rval = rfact(x-1);
    return rval * x;
}

Registers

$ebx  Stored value of x
$eax
    – Temporary value of x-1
    – Returned value from rfact(x-1)
    – Returned value from this call
Rfact Recursion

leal -1(%ebx),%eax

pushl %eax

call rfact

%eax  x-1
%ebx  x
Rfact Result

Return from Call

\[
\begin{array}{|c|c|}
\hline
\text{Rtn adr} & x \\
\hline
\text{Old } \%ebp & \%ebp \\
\hline
\text{Old } \%ebx & x-1 \\
\hline
\%eax & (x-1)! \\
\%ebx & x \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
\text{Rtn adr} & x \\
\hline
\text{Old } \%ebp & \%ebp \\
\hline
\text{Old } \%ebx & x-1 \\
\hline
\%eax & x! \\
\%ebx & x \\
\hline
\end{array}
\]

imull \%ebx,\%eax
Rfact Completion

8  x
4  Rtn adr
0  Old %ebp
-4 Old %ebx
-8 x-1

%esp

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

x
%esp

%eax  x!
%ebx  x

Old %ebx

%eax  x!
%ebx  Old %ebx


**Pointer Code**

### 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
- Uses tail recursion
  - But GCC only partially optimizes it
Creating & Initializing Pointer

Initial part of \texttt{sfact}

\begin{verbatim}
_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
\end{verbatim}

Using Stack for Local Variable

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

\begin{verbatim}
int sfact(int x)
{
    int val = 1;
    s_helper(x, &val);
    return val;
}
\end{verbatim}
Passing Pointer

Calling `s_helper` from `sfact`

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

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:

- `x`
- `val = 1`
- `&val`
- `x`
Using Pointer

```c
void s_helper
  (int x, int *accum)
{
  . . .
  int z = *accum * x;
  *accum = z;
  . . .
}
```

```c
  . . .
  movl %ecx,%eax  # z = x
  imull (%edx),%eax # z *= *accum
  movl %eax,(%edx) # *accum = z
  . . .
```

- Register `%ecx` holds `x`
- Register `%edx` holds `accum`
  - Use access (%edx) to reference memory
Tail Recursion

Tail Recursive Procedure

```c
int t_helper (int x, int val)
{
    if (x <= 1)
        return val;
    return t_helper(x-1, val*x);
}
```

General Form

```c
t_helper(x, val)
{
    ...
    return t_helper(Xexpr, Vexpr)
}
```

Top-Level Call

```c
int tfact(int x)
{
    return t_helper(x, 1);
}
```

Form

• Directly return value returned by recursive call

Consequence

• Can convert into loop
Removing Tail Recursion

Optimized General Form

```c
int t_helper(int x, int val)
{
    start:
    
    if (x <= 1)
        return val;
    val = val * x;
    x = x - 1;
    goto start;
}
```

Resulting Code

```c
int t_helper(int x, int val)
{
    start:
    
    if (x <= 1)
        return val;
    val = val * x;
    x = x - 1;
    goto start;
}
```

Effect of Optimization

- Turn recursive chain into single procedure
- No stack frame needed
- Constant space requirement
  - Vs. linear for recursive version
Generated Code for Tail Recursive Proc.

Optimized Form

```c
int t_helper
  (int x, int val)
{
    int x, val;
    start:
    if (x <= 1)
      return val;
    val = val*x;
    x = x-1;
    goto start;
}
```

Code for Loop

```assembly
L53:    # start:
  cmp $1,%edx  # x : 1
  jle L52  # if <= goto done
  movl %edx,%eax  # eax = x
  imull %ecx,%eax  # eax = val * x
  decl %edx  # x--
  movl %eax,%ecx  # val = val * x
  jmp L53  # goto start
L52:    # done:
```

Registers

- $edx x
- $ecx val
Summary

The Stack Makes Recursion Work

• Private storage for each *instance* of procedure call
  – Instantiations don’t clobber each other
  – Addressing of locals + arguments can be relative to stack positions
• Can be managed by stack discipline
  – Procedures return in inverse order of calls

IA32 Procedures Combination of Instructions + Conventions

• Call / Ret instructions
• Register usage conventions
  – Caller / Callee save
  – %ebp and %esp
• Stack frame organization conventions