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

IA32 Stack
- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register %esp indicates lowest stack address
  - address of top element

IA32 Stack Pushing
- pushl \textit{Src}
- Fetch operand at \textit{Src}
- Decrement \%esp by 4
- Write operand at address given by \%esp

IA32 Stack Popping
- popl \textit{Dest}
- Read operand at address given by \%esp
- Increment \%esp by 4
- Write to \textit{Dest}
Stack Operation Examples

- Use stack to support procedure call and return
- Procedure call: Push return address on stack; Jump to label
- Return address value: Address of instruction beyond call
- Example from disassembly:
  - Address of instruction beyond call: 0x804854e
  - Example from disassembly:
  - Procedure call: Push return address on stack; Jump to label
  - Procedure return: Pop address from stack; Jump to address

Procedure Control Flow

- Use stack to support procedure call and return
- Procedure call: Push return address on stack; Jump to label
- Return address value: Address of instruction beyond call
- Example from disassembly:
  - Address of instruction beyond call: 0x804854e
  - Example from disassembly:
  - Procedure call: Push return address on stack; Jump to label
  - Procedure return: Pop address from stack; Jump to address

Procedure Call Example

- Use stack to support procedure call and return
- Procedure call: Push return address on stack; Jump to label
- Return address value: Address of instruction beyond call
- Example from disassembly:
  - Address of instruction beyond call: 0x804854e
  - Example from disassembly:
  - Procedure call: Push return address on stack; Jump to label
  - Procedure return: Pop address from stack; Jump to address

Procedure Return Example

- Use stack to support procedure call and return
- Procedure call: Push return address on stack; Jump to label
- Return address value: Address of instruction beyond call
- Example from disassembly:
  - Address of instruction beyond call: 0x804854e
  - Example from disassembly:
  - Procedure call: Push return address on stack; Jump to label
  - Procedure return: Pop address from stack; Jump to address
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

Stack Frames

Contents
- Local variables
- Return information
- Temporary space

Management
- Space allocated when enter procedure
  - "Set-up" code
- Deallocated when return
  - "Finish" code

Pointers
- Stack pointer `%esp` indicates stack top
- Frame pointer `%ebp` indicates start of current frame
Stack Operation

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

Stack Operation

Call Chain

Stack Operation

Call Chain

amI(…)
{ • • •
  amI(); • • •
  amI(); • • •
}

Stack Operation

Call Chain
amI(…) {
  •
  •
  •
} amI();

Call Chain

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

Frame Pointer %ebp
Stack Pointer %esp

Stack Operation

Stack Operation

amI(…) {
  •
  •
} amI();

Call Chain

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

Frame Pointer %ebp
Stack Pointer %esp

Stack Operation

Stack Operation
Stack Operation

Call Chain

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

Frame Pointer %ebp
Stack Pointer %esp

Pushed by call instruction
Arguments for this call

IA32/Linux Stack Frame

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

Caller Stack Frame
- Return address
- Arguments for this call

IA32/Linux Stack Frame

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

Caller Stack Frame
- Return address
- Arguments for this call

Revisiting swap

Calling swap from call_swap

int zip1 = 15213;
int zip2 = 91125;
void call_swap() {
      swap(&zip1, &zip2);
    }

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

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

\[ \begin{array}{l}
  %ebp \\
  %esp \\
  &zip1 \\
  &zip2 \\
  Rtn adr
\end{array} \]

Resulting Stack

\[ \begin{array}{l}
  %ebp \\
  %esp \\
  yp \\
  xp \\
  Rtn adr
\end{array} \]

swap Setup #2

Entering Stack

\[ \begin{array}{l}
  %ebp \\
  %esp \\
  &zip1 \\
  &zip2 \\
  Rtn adr
\end{array} \]

Resulting Stack

\[ \begin{array}{l}
  %ebp \\
  %esp \\
  %esp \\
  yp \\
  xp
\end{array} \]

swap Setup #3

Entering Stack

\[ \begin{array}{l}
  %ebp \\
  %esp \\
  &zip1 \\
  &zip2 \\
  Rtn adr
\end{array} \]

Resulting Stack

\[ \begin{array}{l}
  %ebp \\
  %esp \\
  &zip1 \\
  &zip2 \\
  %esp \\
  %esp
\end{array} \]
Effect of swap Setup

Entering Stack

<table>
<thead>
<tr>
<th>Offset (relative to %ebp)</th>
<th>%ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>yp</td>
</tr>
<tr>
<td>8</td>
<td>xp</td>
</tr>
<tr>
<td>4</td>
<td>Rtn adr</td>
</tr>
<tr>
<td>0</td>
<td>%esp</td>
</tr>
<tr>
<td>-4</td>
<td>Old %ebp Old %esp</td>
</tr>
</tbody>
</table>

Resulting Stack

Body

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

Observation

- Saved & restored register %ebx

swap Finish #1

swap’s Stack

Offset

12 yp
8 xp
4 Rtn adr
0 Old %ebp
-4 Old %esp

Observation

- Saved & restored register %ebx

swap Finish #2

swap’s Stack

Offset

12 yp
8 xp
4 Rtn adr
0 Old %ebp
-4 Old %esp

Body

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

swap Finish #3

swap’s Stack

Offset

12 yp
8 xp
4 Rtn adr
0 Old %ebp
-4 Old %esp

Body

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

When procedure \texttt{yoo} calls \texttt{who}:
\begin{itemize}
  \item \texttt{yoo} is the \textit{caller}, who is the \textit{callee}
\end{itemize}

Can Register be Used for Temporary Storage?
\begin{itemize}
  \item \texttt{Caller Save}:
    \begin{itemize}
      \item Caller saves temporary in its frame before calling
    \end{itemize}
  \item \texttt{Callee Save}:
    \begin{itemize}
      \item Callee saves temporary in its frame before using
    \end{itemize}
\end{itemize}

Observation
\begin{itemize}
  \item Saved & restored register \%ebx
  \item Didn't do so for \%eax, \%ecx, or \%edx
\end{itemize}

Register Saving Conventions

IA32/Linux Register Usage

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

\begin{itemize}
  \item \%eax \%edx \%ebx \%esi \%edi \%esp \%ebp
  \item Caller-Save Temporaries
  \item Callee-Save Temporaries
  \item Special
\end{itemize}
Recursive Factorial

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

Registers
- %eax used without first saving
- %ebx used, but save at beginning & restore at end

Rfact Stack Setup
- Pre %ebp
- Pre %ebx
- %ebp
- %ebx

Rfact Body
- Recursion
- %ebx

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
Rfact Result

Return from Call

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

Assume that \( \text{rfact}(x-1) \) returns \((x-1)!\) in register %eax

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

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);
    }
}
```

Pass pointer to update location

Top-Level Call

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

Creating & Initializing Pointer

Initial part of sfact

```c
_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
```

Using Stack for Local Variable

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

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

**Calling s_helper from sfact**

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

**Stack at time of call**

- `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

**Stack at time of call**

- `-4` value
- `-8` return value
- `-12` unused
- `-16` stack at time of call

### Using Pointer

**void s_helper**

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

- Register `%ecx` holds `x`
- Register `%edx` holds pointer to `accum`  
- Use access `%edx` to reference memory

### 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