Procedures Calls

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

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

pushl %eax
popl %edx

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
  804854e: e8 3d 06 00 00 call 8048b90 <main>
  8048553: 50 pushl %eax
- Return address = 0x8048553

Procedure return:
- Pop address from stack; Jump to address

Procedure Call Example

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

Procedure Return Example

8048591: c3 ret
**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**

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

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

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

**Call Chain**

---

**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**

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

---
Stack Operation

Call Chain

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

Stack Operation

Call Chain

amI(...) {
  • • amI(); • •
}

Stack Operation

Call Chain

amI(...) {
  • • amI(); • •
}

Stack Operation

Call Chain

amI(...) {
  • • amI(); • •
}
Stack Operation

Call Chain

\[
\text{amI(\ldots)} \{
\text{\ldots}
\text{amI();}
\text{\ldots}
\}
\]

Stack Operation

Call Chain

\[
\text{amI(\ldots)} \{
\text{\ldots}
\text{amI();}
\text{\ldots}
\}
\]

Stack Operation

Call Chain

\[
\text{who(\ldots)} \{
\text{\ldots}
\text{amI();}
\text{\ldots}
\}
\]

Stack Operation

Call Chain

\[
\text{amI(\ldots)} \{
\text{\ldots}
\text{amI();}
\text{\ldots}
\}
\]
Stack Operation

Call Chain

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

Frame Pointer (%ebp)
Stack Pointer (%esp)

Stack Operation

Call Chain

yoo(…)
{  
  ••who();  
  ••
}

Frame Pointer (%ebp)
Stack Pointer (%esp)

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:
  - Pushed by call instruction
- Arguments for this call

Revisiting swap

Calling swap from call_swap

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

int zipl = 15213;
int zip2 = 91125;

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

int zip1 = 15213;
int zip2 = 91125;

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

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

setup #1

setup #2

setup #3
Effect of swap Setup

Entering Stack

Resulting Stack

Offset (relative to %ebp)

%ebp

yp

xp

Old %ebp

Old %ebx

Old %ebp

%esp

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

Body

swap Finish #1

Observation

• Saved & restored register %ebx

swap Finish #2

swap Finish #3

Body

 swapped stack
Register Saving Conventions

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

Can Register be Used for Temporary Storage?

Conventions
- "Caller Save"
  - Caller saves temporary in its frame before calling
- "Callee Save"
  - Callee saves temporary in its frame before using

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

IA32/Linux Register Usage

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
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

**Entering Stack**

- Caller
  - pre %ebp
  - %ebp
  - x
  - %esp
- Callee
  - %ebp
  - %esp
  - -4

**Rfact Body**

```
    movl 8(%ebp),%ebx  # ebx = x
    cmpl $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:
    movl $1,%eax      # return val = 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
    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
.
.L79:                # Done:
    movl $1,%eax      # return val = 1
```

### Rfact Recursion

**Recursion**

```
    leal -1(%ebx),%eax  # eax = x-1
    pushl %eax
    call rfact
    imull %ebx,%eax    # rval * x
    jmp .L79          # Goto done
.
.L78:                # Term:
    movl $1,%eax      # return val = 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
.
.L79:                # Done:
    movl $1,%eax      # return val = 1
```

Registers
- %eax Stored value of x
- %ebx Temporary value of x-1
- Returned value from rfact(x-1)
- Returned value from this call
### 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
```
### 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

- `x`: Value of `x`
- `val`: Value of `val`

### Using Pointer

#### void `s_helper`

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

#### Register usage

- `%ecx`: Holds `x`
- `%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

### Disclaimer

Parts (all in this lecture) of the slides were developed by the course text authors: Dave O’Hallaron and Randy Bryant. The slides are intended for the sole purpose of instruction computer organization at the University of Rochester. All copyrighted materials belong to their original owner(s).