15-213

"The course that gives CMU its Zip!"

Machine-Level Programming III: Procedures
Sept 19, 2000

Topics

- **IA32 stack discipline**
- **Register saving conventions**
- **Creating pointers to local variables**
- **Stack buffer overflow exploits**
  - finger
  - AIM (AOL Instant Messenger)

IA32 Stack

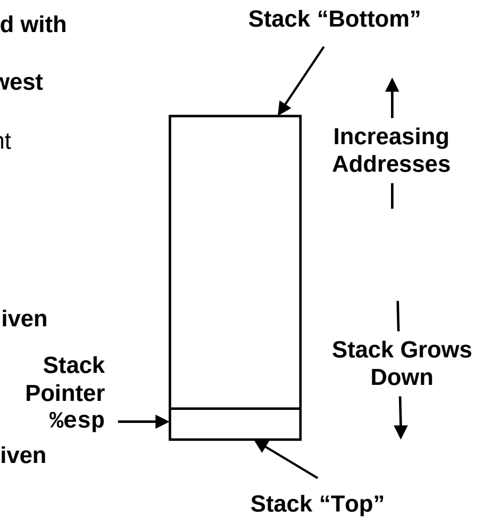
- **Region of memory managed with stack discipline**
- **Register `%esp` indicates lowest allocated position in stack**
  - i.e., address of top element

Pushing

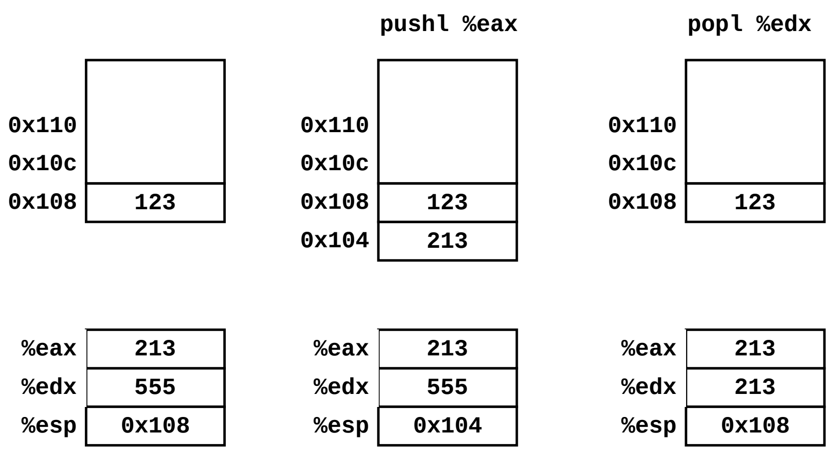
- **`pushl` *Src***
- **Fetch operand at *Src***
- **Decrement `%esp` by 4**
- **Write operand at address given by `%esp`**

Popping

- **`popl` *Dest***
- **Read operand at address given by `%esp`**
- **Increment `%esp` by 4**
- **Write to *Dest***



Stack Operation Examples



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
```

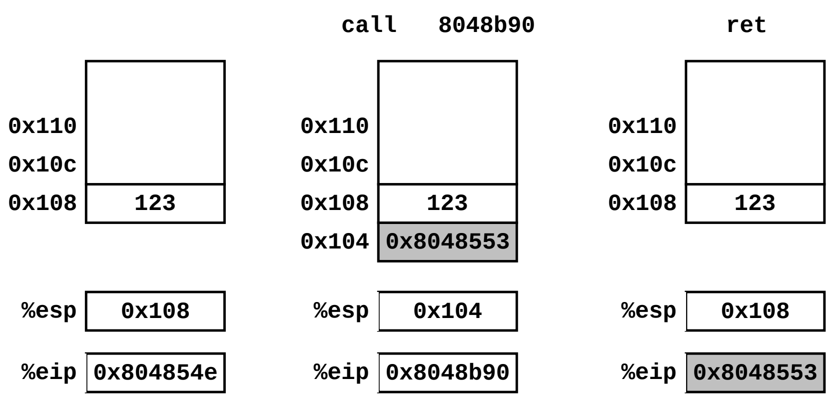
  - Return address = **`0x8048553`**

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
```



`%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

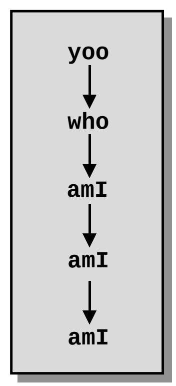
```
yoo(…)
{
   •
   •
   who();
   •
   •
}
```

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

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

- Procedure `amI` recursive

Call Chain



IA32 Stack Structure

Stack Growth

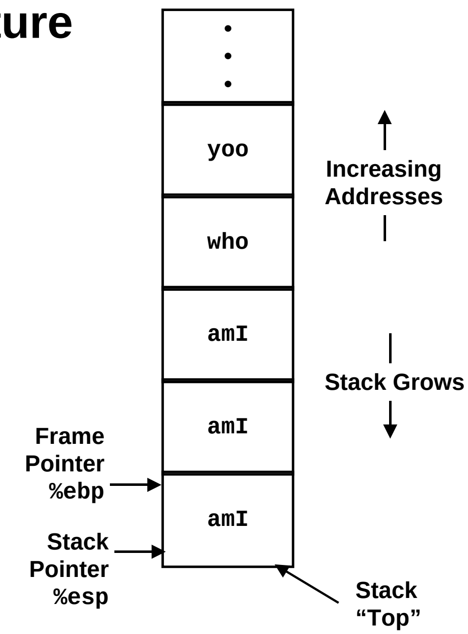
- Toward lower addresses

Stack Pointer

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

Frame Pointer

- Start of current stack frame
- Use register `%ebp`



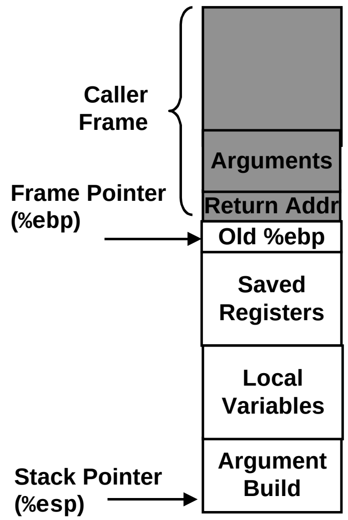
IA32/Linux Stack Frame

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

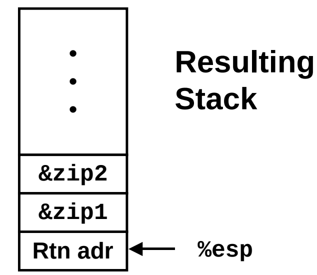


Revisiting swap

```
int zip1 = 15213;
int zip2 = 91125;

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

```
call_swap:
    • • •
    pushl $zip2
    pushl $zip1
    call swap
    • • •
```

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

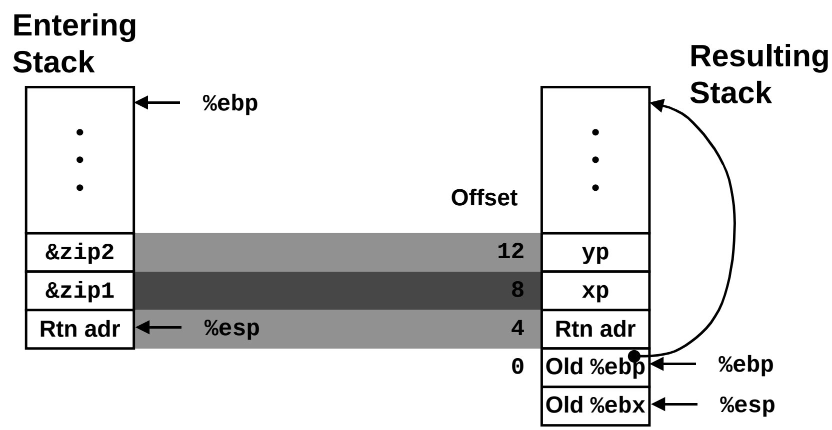


Revisiting swap

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

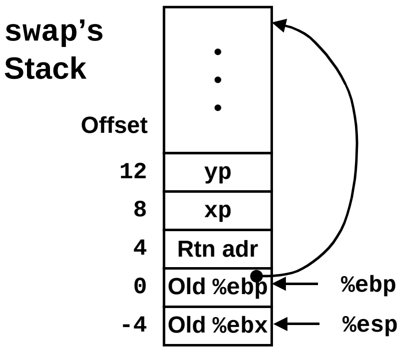
```
swap:
    pushl %ebp              } Set
    movl %esp,%ebp          } Up
    pushl %ebx

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

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

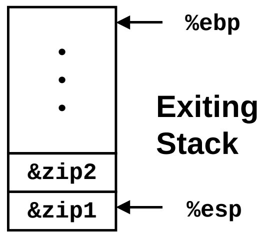

swap Setup




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

swap Finish






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

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 Register be Used for Temporary Storage?

```
yoo:
   • • •
   movl $15213, %edx
   call who
   addl %edx, %eax
   • • •
   ret
```

```
who:
   • • •
   movl 8(%ebp), %edx
   addl $91125, %edx
   • • •
   ret
```

- Contents of register `%edx` overwritten by who

Conventions

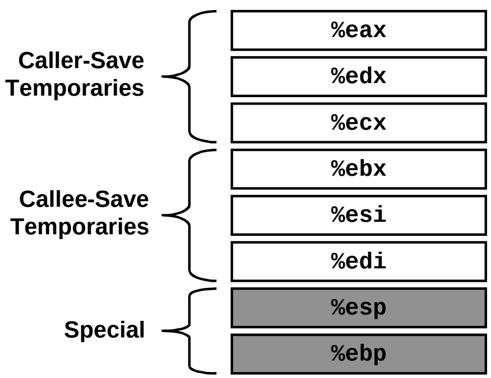
- "Caller Save"
  - Caller saves temporary in its frame before calling
- "Callee Save"
  - Callee saves temporary in its frame before using

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



Recursive Factorial

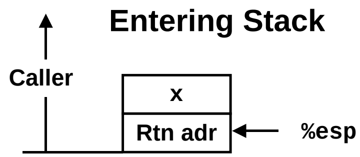
```
int rfact(int x)
{
  int rval;
  if (x <= 1)
    return 1;
  rval = rfact(x-1);
  return rval * x;
}
```

Complete Assembly

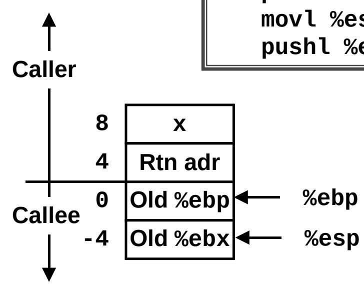
- Assembler directives
  - Lines beginning with "."
  - Not of concern to us
- Labels
  - `.Lxx`
- Actual instructions

```
.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




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



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:
```

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

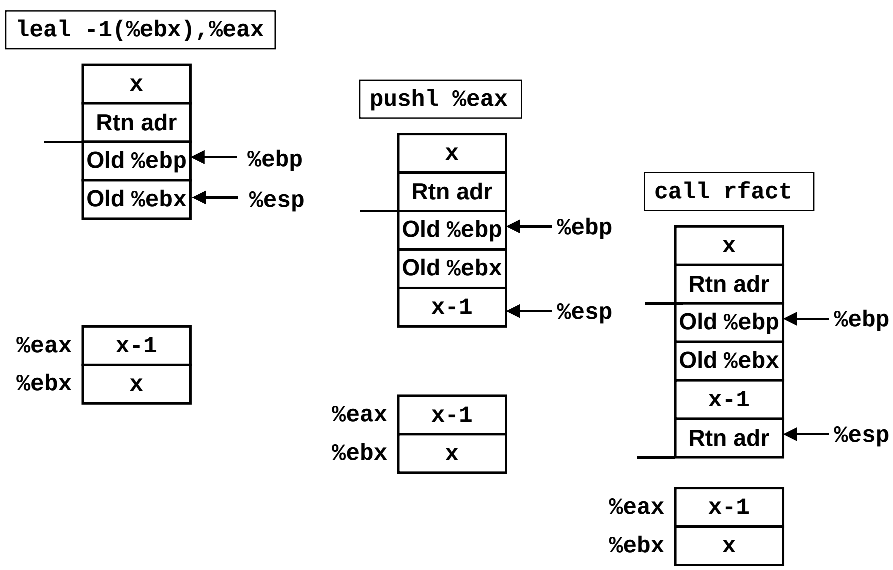
Registers

$ebx **Stored value of x**

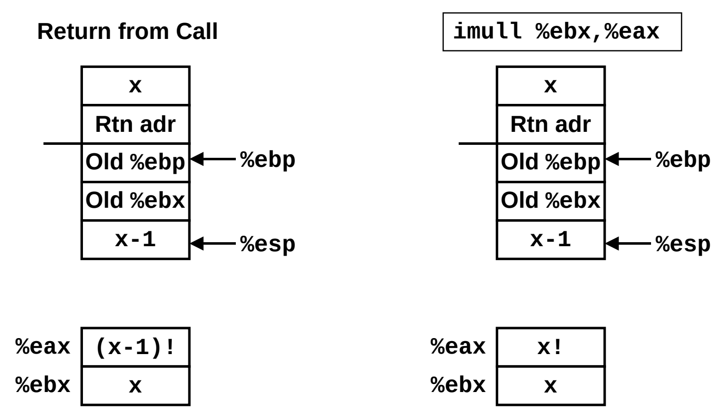
$eax

- Temporary value of **x-1**
- Returned value from **rfact(x-1)**
- Returned value from this call

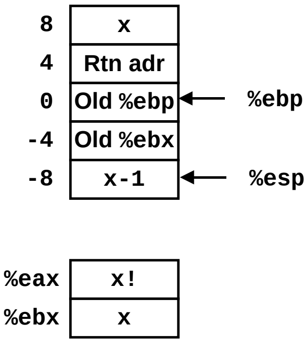
Rfact Recursion



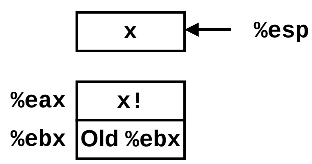
Rfact Result



Rfact Completion




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



Pointer Code

Recursive Procedure

```
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

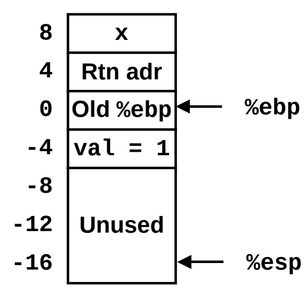
```
int sfact(int x)
{
  int val = 1;
  s_helper(x, &val);
  return val;
}
```

- **Pass pointer to update location**
- **Uses tail recursion**
  - But GCC only partially optimizes it

Creating & Initializing Pointer

Initial part of `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
```



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**

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

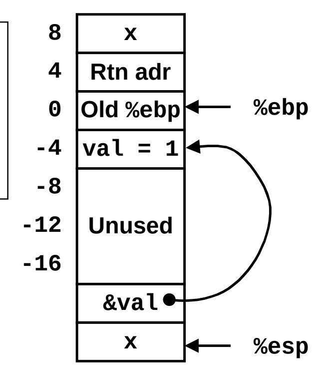

Passing Pointer

Calling `s_helper` from `sfact`

```
  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
```

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

Stack at time of call



Using Pointer

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

```
   • • •
   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

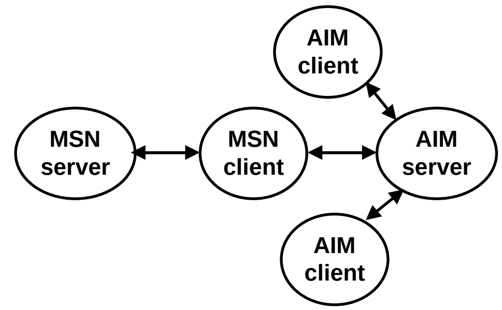
Internet worm and IM War

November, 1988

- **Internet Worm attacks thousands of Internet hosts.**
- **How did it happen?**

July, 1999

- **Microsoft launches MSN Messenger (instant messaging system).**
- **Messenger clients can access popular AOL Instant Messaging Service (AIM) servers**



Internet Worm and IM War (cont)

August 1999

- **Mysteriously, Messenger clients can no longer access AIM servers.**
- **Even though the AIM protocol is an open, published standard.**
- **Microsoft and AOL begin the IM war:**
  - AOL changes server to disallow Messenger clients
  - Microsoft makes changes to clients to defeat AOL changes.
  - At least 13 such skirmishes.
- **How did it happen?**

The Internet Worm and AOL/Microsoft War were both based on *stack buffer overflow* exploits!

- many Unix functions, such as gets() and strcpy(), do not check argument sizes.
- allows target buffers to overflow.

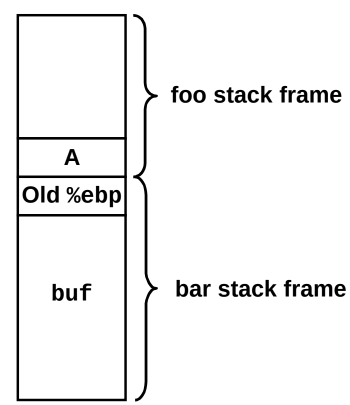
Stack buffer overflows

Stack before call to `gets()`

return address A

```
void foo(){
  bar();
  ...
}
```

```
void bar() {
  char buf[64];
  gets(buf);
  ...
}
```

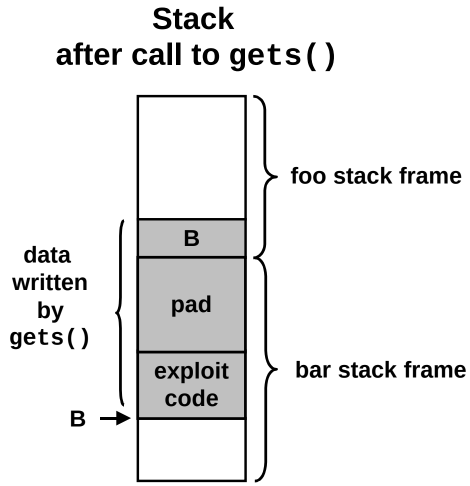


Stack buffer overflows (cont)

return address A

```
void foo(){
  bar();
  ...
}
```

```
void bar() {
  char buf[64];
  gets(buf);
  ...
}
```



When bar() returns, control passes silently to B instead of A!!

Exploits based on buffer overflows

Buffer overflow bugs allow remote machines to execute arbitrary code on victim machines.

Internet worm

- **Early versions of the finger server (fingerd) used `gets()` to read the argument sent by the client:**
  - *`finger droh@cs.cmu.edu`*
- **Worm attacked fingerd client by sending phony argument:**
  - *`finger "exploit code  padding  new return address"`*
  - **exploit code: executed a root shell on the victim machine with a direct TCP connection to the attacker.**

IM War

- **AOL exploited existing buffer overflow bug in AIM clients**
- **exploit code: returned 4-byte signature (the bytes at some location in the AIM client) to server.**
- **When Microsoft changed code to match signature, AOL changed signature location.**

Main Ideas

Stack Provides Storage for Procedure Instantiation

- **Save state**
- **Local variables**
- **Any variable for which must create pointer**

Assembly Code Must Manage Stack

- **Allocate / deallocate by decrementing / incrementing stack pointer**
- **Saving / restoring register state**

Stack Adequate for All Forms of Recursion

- **Including multi-way and mutual recursion examples in the bonus slides.**

Good programmers know the stack discipline and are aware of the dangers of stack buffer overflows.

Free Bonus Slides!

(not covered in lecture)

Topics

- **how the stack supports multi-way recursion.**
- **how the stack supports mutual recursion.**

Multi-Way Recursion

```
int r_prod
  (int from, int to)
{
  int middle;
  int prodA, prodB;
  if (from >= to)
    return from;
  middle = (from + to) >> 1;
  prodA = r_prod(from, middle);
  prodB = r_prod(middle+1, to);
  return prodA * prodB;
}
```

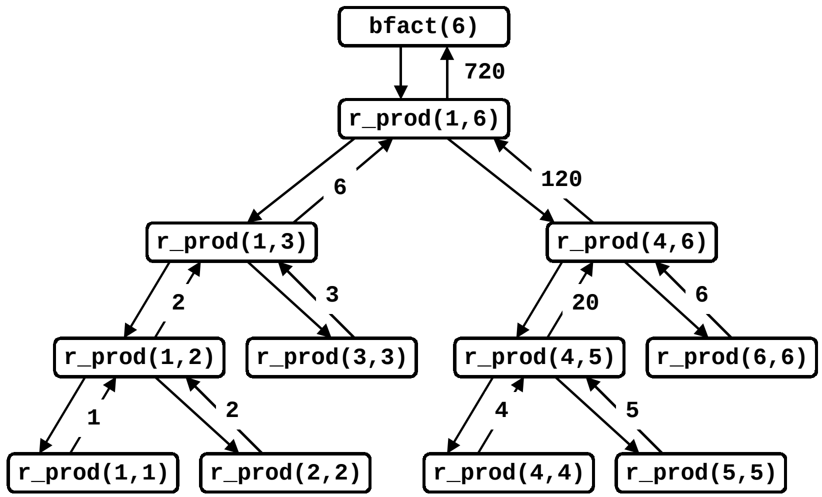
Top-Level Call

```
int bfact(int x)
{
  return r_prod(1,x);
}
```

- **Compute product** *x* * (*x*+1) * … * (*y*–1) * *y*
- **Split into two ranges:**
  - **Left:** *x* * (*x*+1) * … * (*m*–1) * *m*
  - **Right:** (*m*+1) * … * (*y*–1) * *y*

    $m = \lfloor (x+y)/2 \rfloor$
- **No real advantage algorithmically**

Binary Splitting Example



Multi-Way Recursive Code

Stack Frame

12	from
8	to
4	Rtn Adr
0	Old $ebp
-4	Old $edi
-8	Old $esi
-12	Old $ebx

$eax
- **from**
- **return values**

Callee Save Regs.

$ebx	middle
$edi	to
$esi	prodA

```
_r_prod:
   • • •                 # Setup
  movl 8(%ebp),%eax      # eax = from
  movl 12(%ebp),%edi     # edi = to
  cmpl %edi,%eax         # from : to
  jge L8                 # if >= goto done
  leal (%edi,%eax),%ebx  # from + to
  sarl $1,%ebx           # middle
  pushl %ebx             # 2nd arg: middle
  pushl %eax             # 1st arg: from
  call _r_prod           # 1st call
  pushl %edi             # 2nd arg: to
  movl %eax,%esi         # esi = ProdA
  incl %ebx              # middle + 1
  pushl %ebx             # ... 1st arg
  call _r_prod           # 2nd call
  imull %eax,%esi        # ProdA * ProdB
  movl %esi,%eax         # Return value
L8:                    # done:
   • • •                 # Finish
```


Multi-Way Recursive Code Finish

12	from
8	to
4	Rtn Adr
0	Old $ebp
-4	Old $edi
-8	Old $esi
-12	Old $ebx
-16	Arg 2
-20	Arg 1

```
L8:                     # done:
  leal -12(%ebp),%esp # Set Stack Ptr
  popl %ebx           # Restore %ebx
  popl %esi           # Restore %esi
  popl %edi           # Restore %edi
  movl %ebp,%esp      # Restore %esp
  popl %ebp           # Restore %ebp
  ret                 # Return
```

Stack

- **After making recursive calls, still has two arguments on top**

Finishing Code

- **Moves stack pointer to start of saved register area**
- **Pops registers**

Mutual Recursion

Top-Level Call

```
int lrfact(int x)
{
  int left = 1;
  return
   left_prod(&left, &x);
}
```

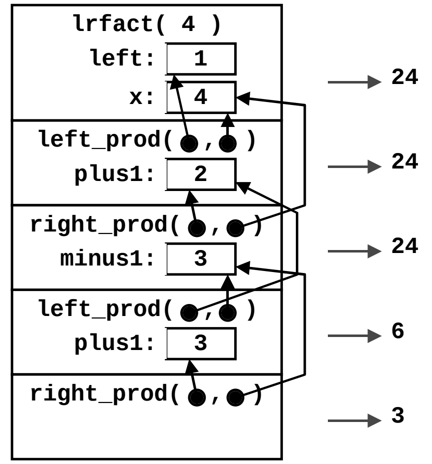
```
int left_prod
(int *leftp, int *rightp)
{
  int left = *leftp;
  if (left >= *rightp)
    return left;
  else {
    int plus1 = left+1;
    return left *
      right_prod(&plus1, rightp);
  }
}
```

```
int right_prod
(int *leftp, int *rightp)
{
  int right = *rightp;
  if (*leftp == right)
    return right;
  else {
    int minus1 = right-1;
    return right *
      left_prod(leftp, &minus1);
  }
}
```


Mutually Recursive Execution Example

Calling

- **Recursive routines pass two arguments**
  - Pointer to own local variable
  - Pointer to caller's local variable



Implementation of `lrfact`

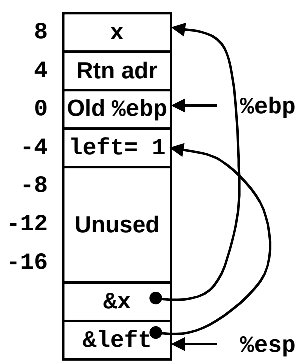
Call to Recursive Routine

```
int left = 1;
return left_prod(&left, &x);
```

Code for Call

```
  leal 8(%ebp),%edx # edx = &x
  pushl %edx        # push &x
  leal -4(%ebp),%eax# eax = &left
  pushl %eax        # push &left
  call _left_prod   # Call
```

Stack at time of call



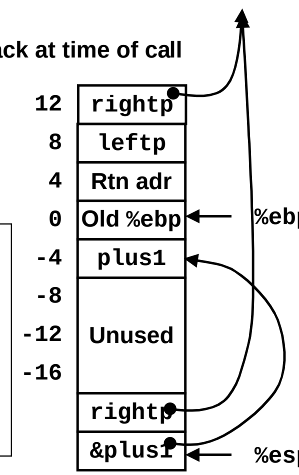
Implementation of `left_prod`

Call to Recursive Routine

```
int plus1 = left+1;
return left *
  right_prod(&plus1, rightp);
```

```
  # %ebx holds left
  # %edx holds rightp
  leal 1(%ebx),%ecx   # left+1
  movl %ecx,-4(%ebp)  # Store in plus1
  pushl %edx          # Push rightp
  leal -4(%ebp),%eax  # &plus1
  pushl %eax          # Push &plus1
  call _right_prod    # Call
```

Stack at time of call



Tail Recursion

Tail Recursive Procedure

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

General Form

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

Top-Level Call

```
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

```
t_helper(x, val)
{
 start:
   • • •
  val = Vexpr;
  x = Xexpr;
  goto start;
}
```

Resulting Code

```
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

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

Code for Loop

```
  # %edx = x
  # %ecx = val
L53:                 # start:
  cmpl $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
```