

CS 24

Introduction to Computer Systems

7: Advanced Control

Questions?

Agenda

- VM24 Ports needed?
- Advanced Control Flow: Exceptions
- VM24 Procedure Calls

Question for you

- Do you do your labs on the CS cluster, or somewhere else?
- If somewhere else:
 - Do you wish you could install VM24 on your own computer?
 - What platform(s) do you use?

Advanced Control Flow

- C has if, loops, functions, switch, “long jump”.
- Other languages have more.
 - Exceptions.
 - Coroutines.
 - Threads (!)
 - Call-with-current-continuation (call/cc).

Advanced Control Flow

- C has if, loops, functions, switch, “long jump”.
- Other languages have more.
 - Exceptions. (Will discuss these today.)
 - Coroutines.
 - Threads (!) (Will discuss these later.)
 - Call-with-current-continuation (call/cc).

Tell Me About Exceptions

- *What are they for?*
- *What do they do?*
- *What are the challenges in implementing them?*

(My Answers)

- *What are they for?*
 - Indicate abnormal termination of a computation.
 - Allow recovery.
- *What do they do?*
 - Allow “return” to someplace other than return address.
 - Jump may not be to caller – may be caller’s caller, *etc.*
 - Carry along data (like return value) indicating what happened.
- *What are the challenges in implementing them?*
 - Where to jump to?
 - What to do with the stack? (Must restore handler’s frame.)
 - How to pass “exception object”?

Exn Syntax in a Fake Language

- Set up exception handler for a block of statements:

```
try {  
    ...  
} handle (e) {  
    ... // Examine e to decide what to do.  
}
```

- Throw value to most recently installed handler:
 throw(...);

- For simplicity, assume:

- No nested try's.
- No throws inside a try.
(Exceptions thrown only from called procedures.)

Handling

```
try {  
    f(3);  
  
} handle (e) {  
    ...  
}
```

- Calling f, must:
 - Communicate location of handler.
 - Communicate location of current frame.
- If try block finishes normally, jump to after handler.
- If f throws exception, will end up here.
 - Receive exception value e.
 - Proceed with handler and following code.

Handling

```
try {  
    f(3);
```

```
# Devote %edi to exception handling.  
    pushl %edi      # save original  
    pushl $handle   # push handler addr.  
    mov %esp,%edi
```

```
} handle (e) {  
    ...  
}
```

```
handle:
```

Handling

```
try {  
    f(3);
```

```
# Devote %edi to exception handling.  
    pushl %edi      # save original  
    pushl $handle   # push handler addr.  
    mov %esp,%edi  
    pushl $3  
    call f
```

```
handle:
```

```
} handle (e) {  
    ...  
}
```

Handling

```
try {  
    f(3);
```

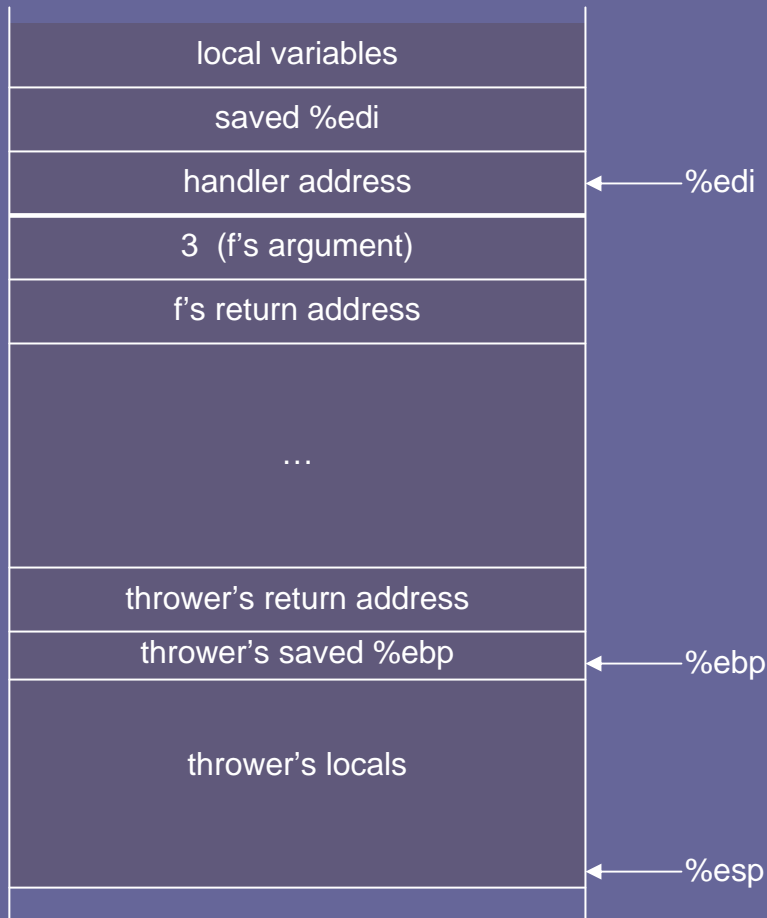
```
} handle (e) {  
    ...  
}
```

```
# Devote %edi to exception handling.  
    pushl %edi      # save original  
    pushl $handle  # push handler addr.  
    mov %esp,%edi  
    pushl $3  
    call f  
    add $8,%esp    # pop arg, handler  
    popl %edi     # restore original  
    jmp end
```

```
handle:
```

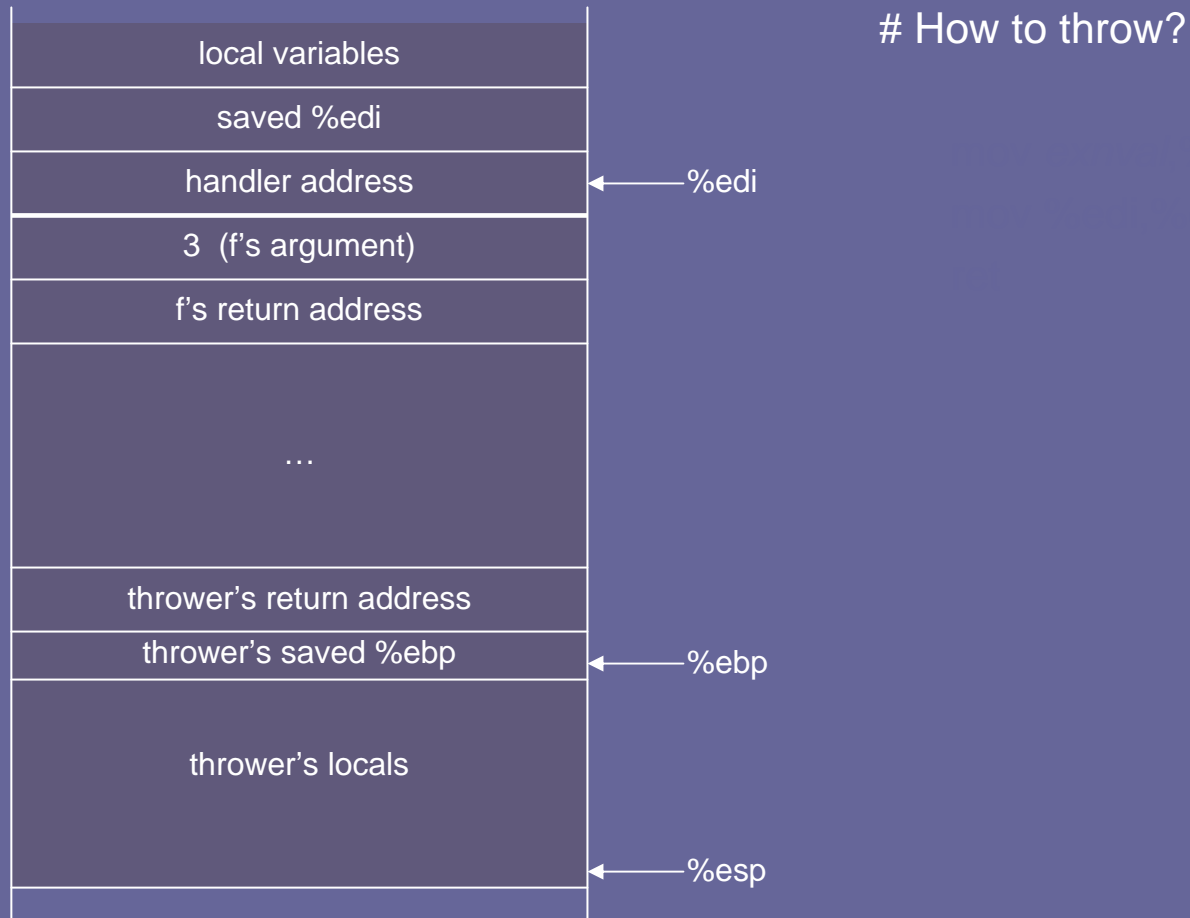
```
end:
```

Stack When Exn is Thrown

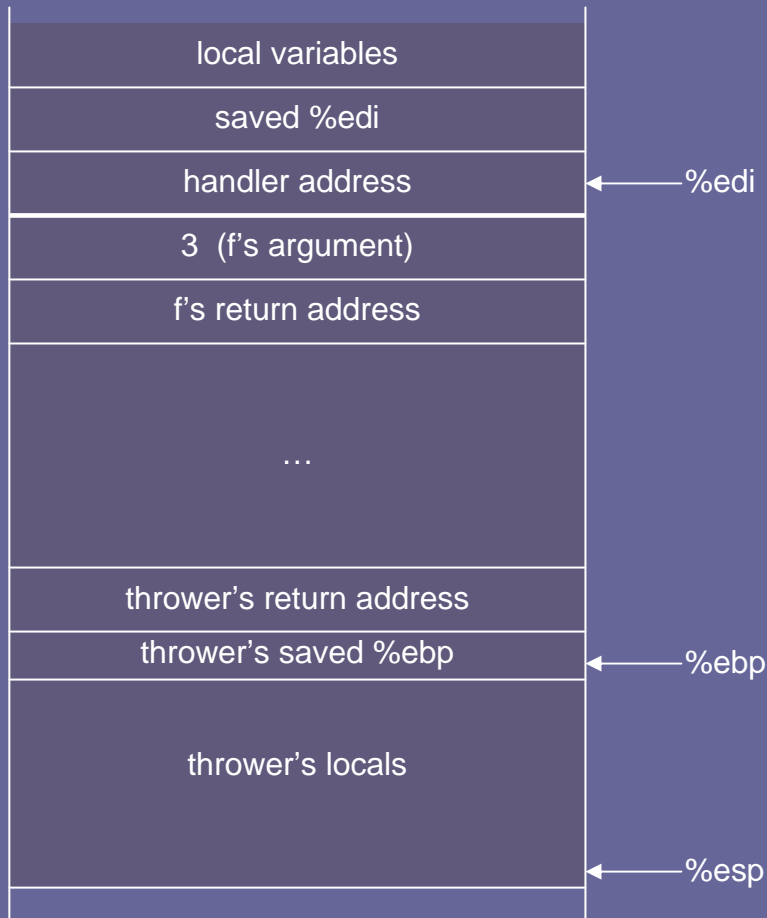


```
# Devote %edi to exception handling.  
    pushl %edi      # save original  
    pushl $handle  # push handler addr.  
    mov %esp,%edi  
    pushl $3  
    call f  
  
    add $8,%esp    # pop arg, handler  
    popl %edi     # restore original  
    jmp end  
  
handle:  
  
    ...  
end:
```

Stack When Exn is Thrown



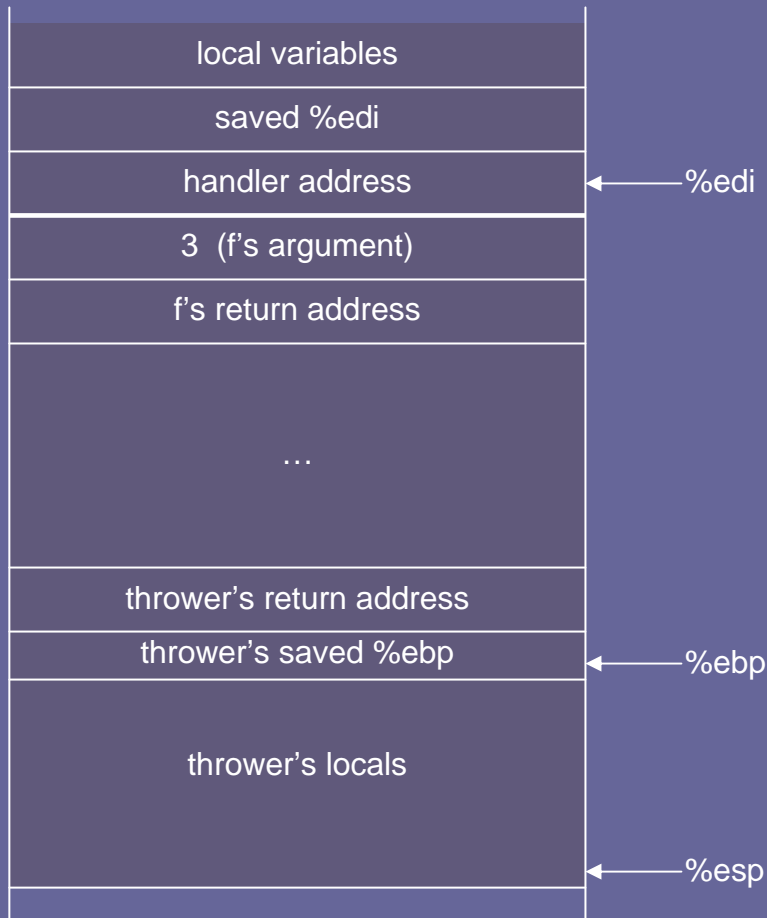
Stack When Exn is Thrown



How to throw?

```
mov exnval,%eax  
mov %edi,%esp  
ret
```

Stack When Exn is Thrown



How to throw?

```
mov exnval,%eax  
mov %edi,%esp  
ret
```

handle:

```
popl %edi  
lea LOCALSIZE(%esp),%ebp
```

end:

Handling

```
try {  
    f(3);  
  
} handle (e) {  
    ...  
}
```

```
# Devote %edi to exception handling.  
    pushl %edi      # save original  
    pushl $handle   # push handler addr.  
    mov %esp,%edi  
    pushl $3  
    call f  
    add $8,%esp     # pop arg, handler  
    popl %edi      # restore original  
    jmp end  
  
handle:  
    popl %edi  
    lea LOCALSIZE(%esp),%ebp  
    mov %eax,edisp(%ebp)  
    ...  
end:
```

One More Detail

- Callee-saves registers?
 - Survive function call if it returns normally, but not if it throws an exception.
 - Any function between handler and thrower might have saved/reused.
- Two choices:
 - Arrange to “unwind” register saving before discarding stack.
 - Let the code installing a handler deal with it.

```
# Devote %edi to exception handling.
    pushl %edi      # save original
    pushl $handle  # push handler addr.
    mov %esp,%edi
    pushl $3
    call f
    add $8,%esp    # pop arg, handler
    popl %edi      # restore original
    jmp end

handle:
    popl %edi
    lea LOCALSIZE(%esp),%ebp
    mov %eax,edisp(%ebp)
    ...
end:
```

Wrapup

- This was just one way to do exception handling.
- I don't claim this is how any real compiler for any real language does it.

VM24 Function Calls

VM24 Project Status

- After this week, you'll have implemented:
 - All but one of VM24's data movement instructions.
 - All of VM24's arithmetic instructions.
 - All of VM24's **intra**procedural control flow.
- What is left:
 - Function call and return.
 - Memory management and array instructions.
 - Symbol resolution.

VM24 Project Status

- After this week, you'll have implemented:
 - All but one of VM24's data movement instructions.
 - All of VM24's arithmetic instructions.
 - All of VM24's **intra**procedural control flow.
- What is left:
 - **Function call and return.** (Lab 4 -- discuss today)
 - Memory management and array instructions.
 - Symbol resolution.

Recall

- VM24 bytecode divided into functions.
- Each function activation gets its own
 - Argument variables
 - Local variables
 - Stack

and must specify the capacities of each.

- Entry point of program is function called “**main**”.
- VM exits when **main** returns.

Function Calls and Returns

- Two ways to call a function (e.g. $f(3, 4)$).

Direct:

Indirect:

Function Calls and Returns

- Two ways to call a function (e.g. $f(3, 4)$).

Direct:

```
ldi    3
ldi    4
call   2,f    # 2 arguments.
# Result of  $f(3, 4)$  now on stack.
```

Indirect:

Function Calls and Returns

- Two ways to call a function (e.g. $f(3, 4)$).

Direct:

```
ldi    3
ldi    4
call   2,f    # 2 arguments.
# Result of  $f(3, 4)$  now on stack.
```

Indirect:

```
ldi    3
ldi    4
ldg    f      # push "global" symbol value
icall  2
```

Function Calls and Returns

- Two ways to call a function (e.g. `f(3,4)`).

Direct:

```
ldi    3
ldi    4
call   2,f    # 2 arguments.
# Result of f(3,4) now on stack.
```

Indirect:

```
ldi    3
ldi    4
ldg    f      # push "global" symbol value
icall  2
```

- Return with `ret` instruction.
Value on top of stack becomes result.

Function Calls and Returns

- Two ways to call a function (e.g. `f(3,4)`).

Direct:

```
ldi    3
ldi    4
call   2,f    # 2 arguments.
# Result of f(3,4) now on stack.
```

Indirect:

```
ldi    3
ldi    4
ldg    f      # push "global" symbol value
icall  2
```

- Note: Saving `ldg` and `call` for Lab 5;
Lab 4 only concerned with `icall` and `ret`.

The Big Question

- How do we handle function calls in our interpreter?

Ideas?

The Big Question

- How do we handle function calls in our interpreter?
- Two possible answers:
 - Simulate VM24 call/return with C call/return.
(“Meta-circular” approach)
 - Some other way.
(“Explicit” approach)

Meta-circularity

- Definition:

An interpreter is **meta-circular** if it implements each feature of the language it interprets by using the corresponding feature of the language it is written in.

(Adapted from [Reynolds])

Meta-circularity

- Definition:

An interpreter is meta-circular if it implements each feature of **the language it interprets** by using the corresponding feature of the language it is written in.

(Adapted from [Reynolds])

(i.e., **VM24 bytecode**)

Meta-circularity

- Definition:

An interpreter is meta-circular if it implements each feature of the language it interprets by using the corresponding feature of **the language it is written in.**

(Adapted from [Reynolds])

(i.e., **C**)

Meta-Circularity (disclaimer)

I'm not saying you should strive to write metacircular interpreters, or that you should avoid them.

I'm just introducing the term as a name for a certain way of doing things.

Meta-Circularity

- After Lab 3, VM24 interpreter is meta-circular wrt some features, but not others.

Meta-circular features?

Non-meta-circular features?

Meta-Circularity

- After Lab 3, VM24 interpreter is meta-circular wrt some features, but not others.

Meta-circular features?

- Integer arithmetic, comparison operations

Non-meta-circular features?

- Variables; operand stack; jumps

Meta-Circular icall and ret

```
switch (code[pc]) {  
    case ICALL:
```

```
}
```

Meta-Circular icall and ret

```
switch (code[pc]) {  
    case ICALL:      bytfunction *callee = ... ;  
                    wordval *cargs = ...;  
                    wordval cresult =  
                        call_function(callee,cargs);  
                    sp -= ...;  
                    stack[sp] = cresult;  
                    pc += 2;  
                    break;  
}
```

Function-calling function

```
wordval call_function(bytefunction *bf,wordval *args) {  
  
    localstate *s = ...;  
    ... /* Initialize s with args, etc. */  
  
    wordval result = interp_bytecode(s);  
  
    ... /* Destroy s */  
  
    return result;  
}
```

Meta-Circular icall and ret

```
switch (code[pc]) {  
    case ICALL:      bytefunction *callee = ... ;  
                    wordval *cargs = ...;  
                    wordval cresult =  
                        call_function(callee,cargs);  
                    sp -= ...;  
                    stack[sp] = cresult;  
                    pc += 2;  
                    break;  
  
    case RET:  
  
}
```

Meta-Circular icall and ret

```
switch (code[pc]) {  
    case ICALL:      bytefunction *callee = ... ;  
                    wordval *cargs = ...;  
                    wordval cresult =  
                        call_function(callee,cargs);  
                    sp -= ...;  
                    stack[sp] = cresult;  
                    pc += 2;  
                    break;  
    case RET:        return stack[sp];  
}
```

Non-meta-circular icall and ret

```
switch (code[pc]) {  
  case ICALL:
```

Non-meta-circular icall and ret

```
switch (code[pc]) {                                     /* Warning: Simplified! */
    case ICALL:   callee = ...;
                 cargs = ...;
                 localstate *cs = ...;
                 ... /* Initialize cs */
                 s->pc = pc + 2;
                 s->sp = sp - ...;
                 cs->callerstate = s;
                 s = cs;
                 code = s->code;
                 args = s->args;
                 locals = s->locals;
                 stack = s->stack;
                 pc = s->pc;
                 sp = s->sp;
                 break;
```

Non-meta-circular icall and ret

...

case RET:

}

Non-meta-circular icall and ret

```
...                                     /* Warning: Simplified! */
case RET:    retval = stack[sp];
            s = s->callerstate;
            code=s->code;
            pc = s->pc;
            sp = s->sp;
            stack=s->stack;
            ...etc...
            stack[sp] = retval;
            break;
}
```

Preview

- The preceding was **JUST A SKETCH** of how your Lab 4 implementation will work.
- The lab handout will explain more details.
- It will not be available until Thursday.

Epilogue (time permitting)

- *What if VM24 had exception-handling features?*

Next Time

- Memory Management
 - C: malloc/free
 - Modern languages: automated management
 - Reference counting
 - Garbage collection