Programming Paradigms

Control Abstraction

Prof. Dr. Michael Pradel

Software Lab, University of Stuttgart

Summer 2023

Control vs. Data Abstraction

- Abstract a well-defined operation
- E.g., a subroutine or an exception handler

- Abstract how to represent information
- E.g., types and classes

Control vs. Data Abstraction

- Abstract a well-defined operation
- E.g., a subroutine or an exception handler

Focus of this lecture

- Abstract how to represent information
- E.g., types and classes

Overview

- Calling Sequences ←
- Coroutines
- Promises, Async, and Await

Terminology

- **Subroutine: Mechanism for control abstraction**
  - □ Function: Subroutine that returns a value
  - □ Procedure: Subroutine that doesn’t return a value
- **Parameters**
  - □ Actual parameters = arguments: Data passed by caller
  - □ Formal parameters: Data received by callee

Calling Sequences

- **Low-level code executed to maintain call stack**
  - Before subroutine call in caller
  - At beginning of subroutine in callee ("prologue")
  - At end of subroutine in callee ("epilogue")
  - After subroutine call in caller

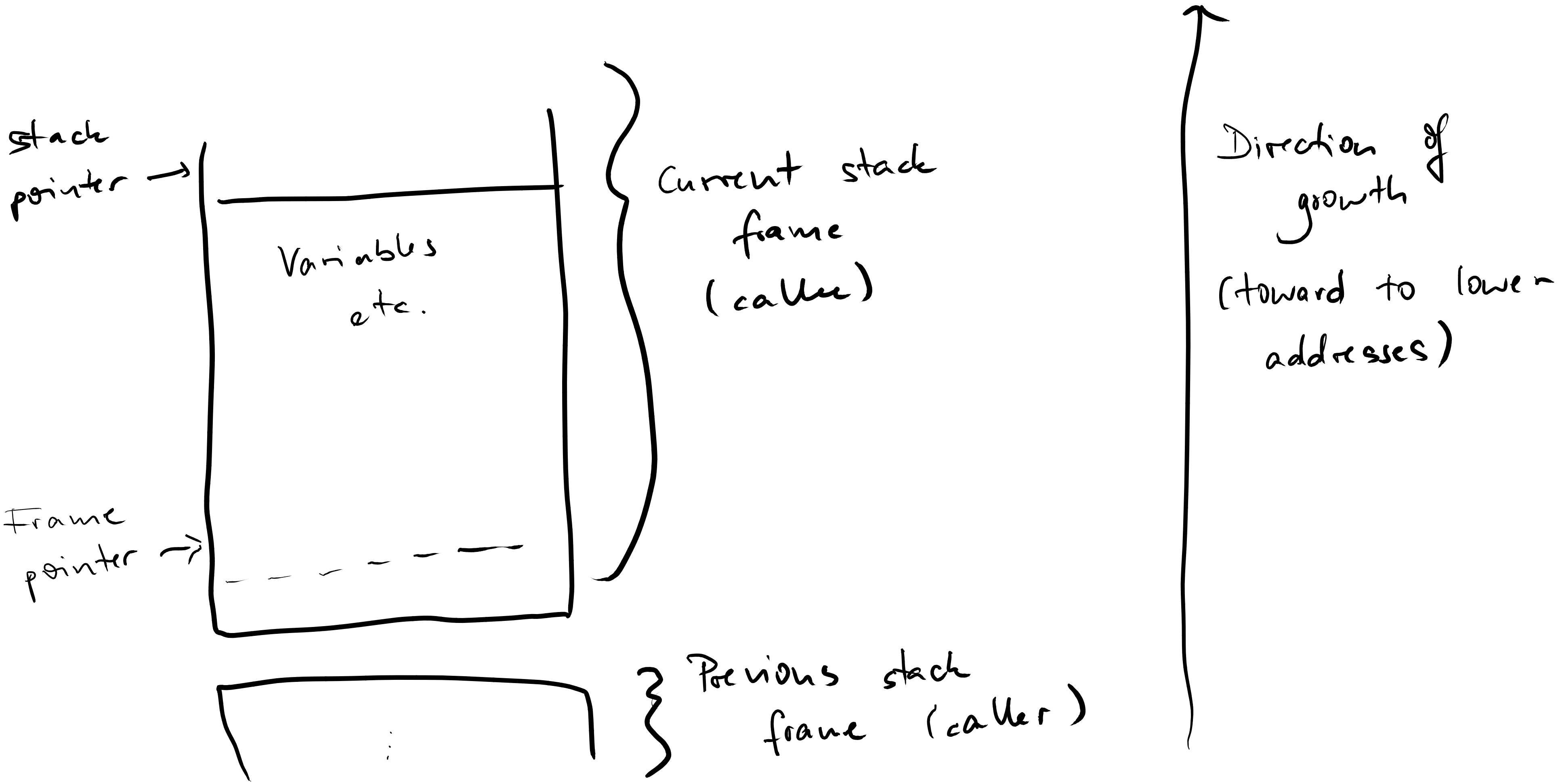
Why Does It Matter?

- **Important to**
  - Understand performance implications
  - Understand security implications, e.g., stack smashing attacks
  - Choose/design/implement compilers

Reminder: Stack Layout

- Each procedure call:
  One stack frame (or activation record)
- Frame pointer: Base address used to access data in current stack frame
- Stack pointer: First unused (or, sometimes, last used) location in current stack frame

Stack Layout


stack pointer
Variables etc.
Frame pointer
Current stack frame (callee)
Previous stack frame (caller)
Direction of growth
(toward to lower addresses)

Tasks to Perform

- Pass parameters and return value(s)
- Update program counter
- Save return address
- Save and restore registers
- Update stack and frame pointers

Tasks to Perform

- Pass **parameters** and **return value(s)**
- Update **program counter**
- Save **return address**
- Save and restore **registers**
- Update **stack and frame pointers**

Program counter: Address of code to execute next

Tasks to Perform

- Pass parameters and return value(s)
- Update program counter
- Save return address
- Save and restore registers
- Update stack and frame pointers

Otherwise, don’t know what code location to return back to

Tasks to Perform

- Pass parameters and return value(s)
- Update program counter
- Save return address
- Save and restore registers
- Update stack and frame pointers

Registers: Very fast but limited intermediate memory

Tasks to Perform

- Pass **parameters** and **return value(s)**
- Update **program counter**
- Save **return address**
- Save and restore **registers**
- Update **stack and frame pointers**

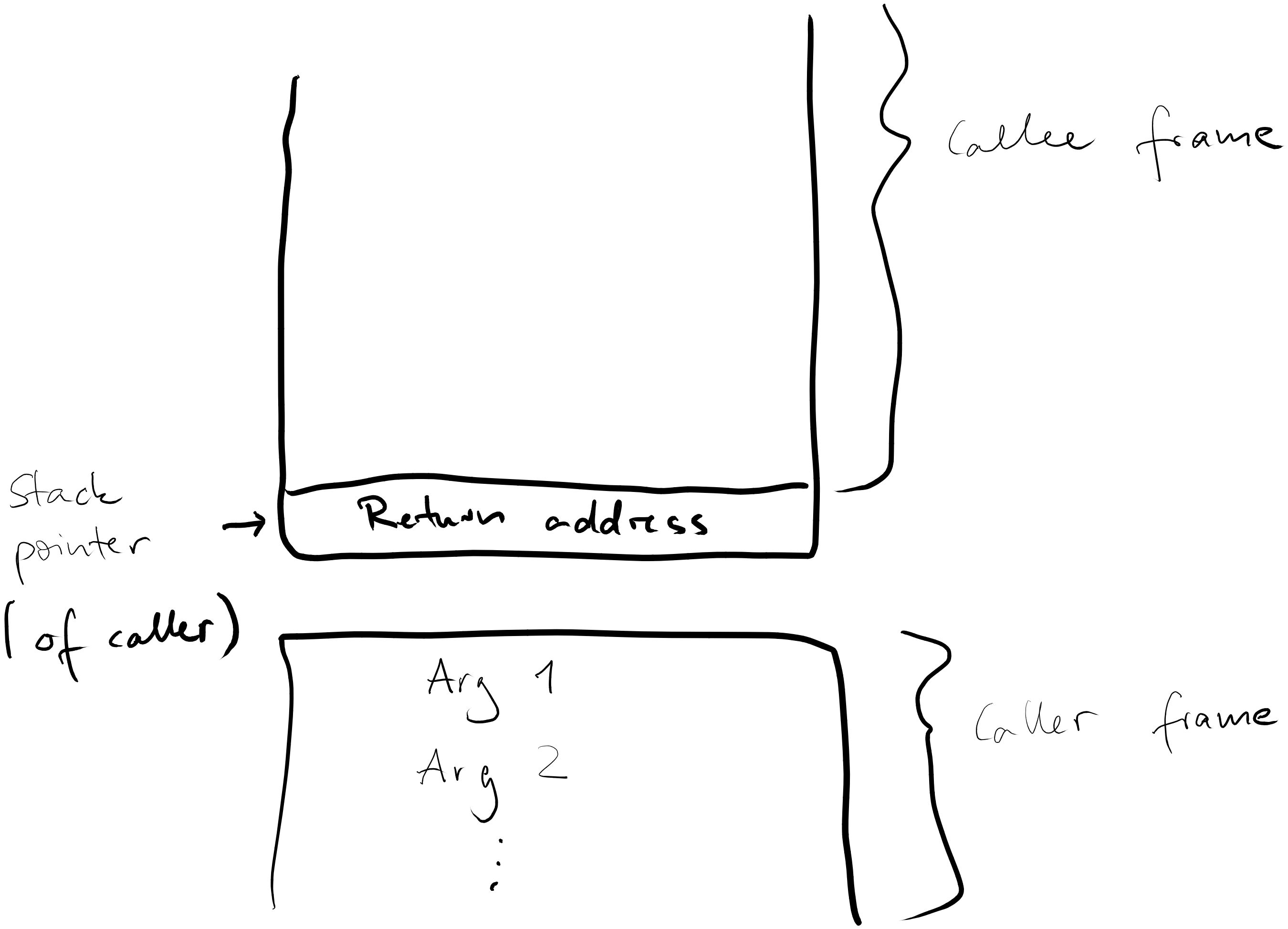
Where to perform those?

- Possibly either in caller or in callee
- Preferably in callee: Requires space only once per subroutine, not at each call site

Typical Calling Sequence (1/4)

- **Steps performed by caller before the call**
  - Save registers whose values may be needed after the call
  - Compute values of arguments and move them into stack or registers
  - Pass return address and jump to subroutine

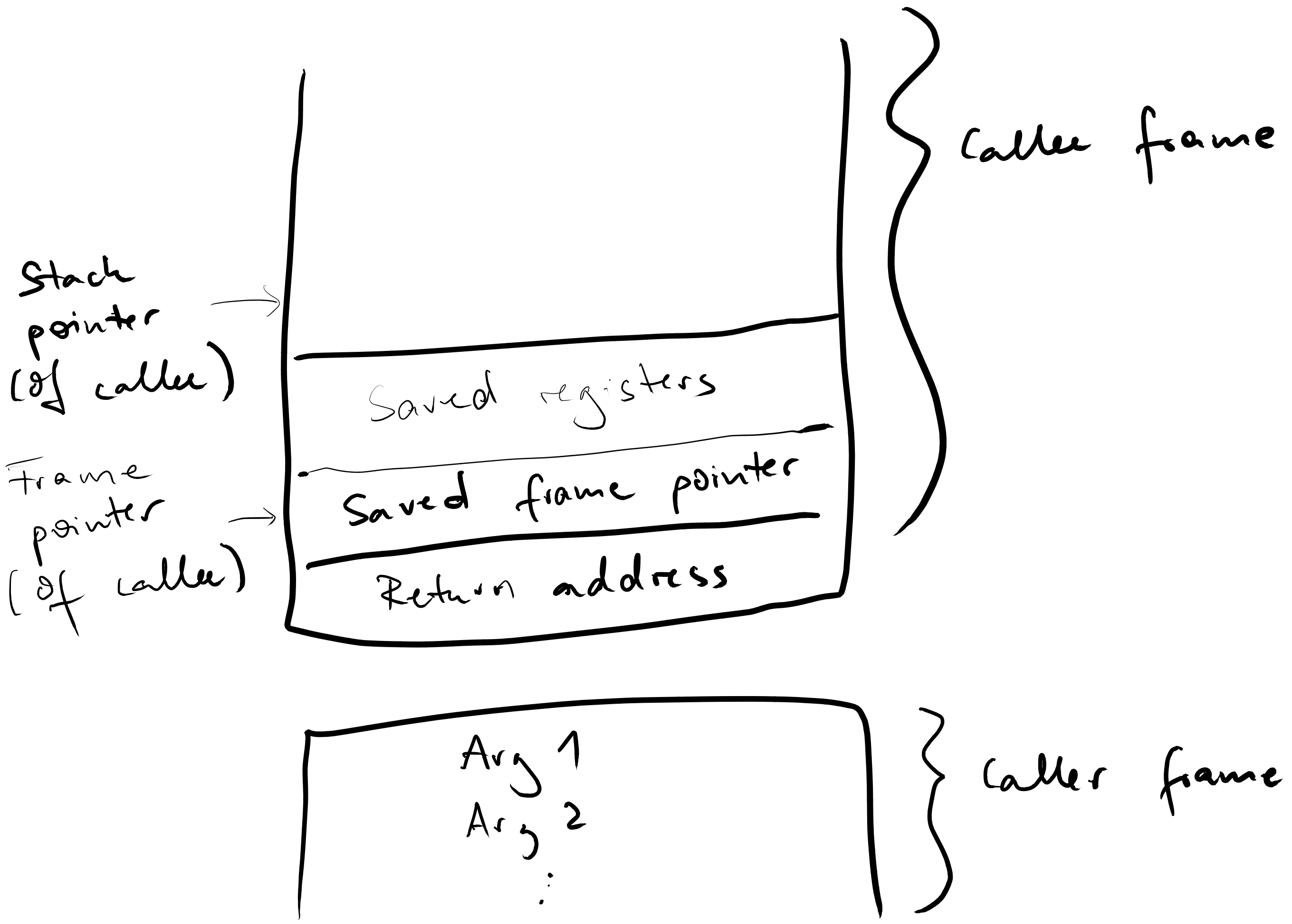
Stack before call



Typical Calling Sequence (2/4)

- **Steps performed by callee in prologue**
  - Allocate a frame: Subtract an appropriate constant from the stack pointer
  - Save old frame pointer on stack and update it to point to newly allocated frame
  - Save registers that may be overwritten by current subroutine

Stack after prologue

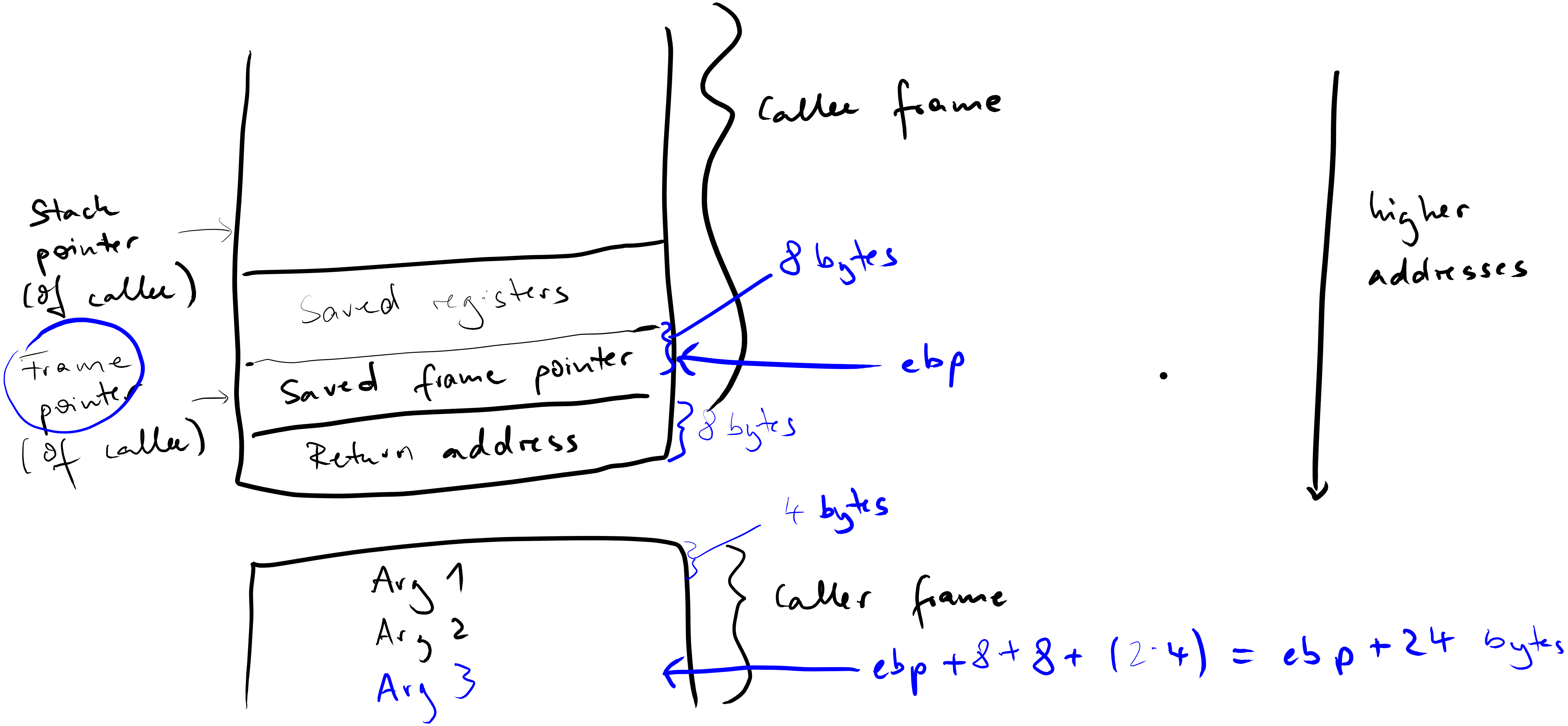


Quiz: Stack Frames

Assume the frame pointer is stored in register `ebp`, addresses are 8 bytes long, and all arguments are 32-bit integers.

What is the address the callee uses to access the third argument?

Quiz


Callee frame
Stack pointer (of callee)
Saved registers
Frame pointer (of callee)
Saved frame pointer
Return address
8 bytes
ebp
8 bytes
4 bytes
higher addresses
Arg 1
Arg 2
Arg 3
Caller frame
ebp + 8 + 8 + (2·4) = ebp + 24 bytes

Quiz: Stack Frames

Assume the frame pointer is stored in register `ebp`, addresses are 8 bytes long, and all arguments are 32-bit integers.

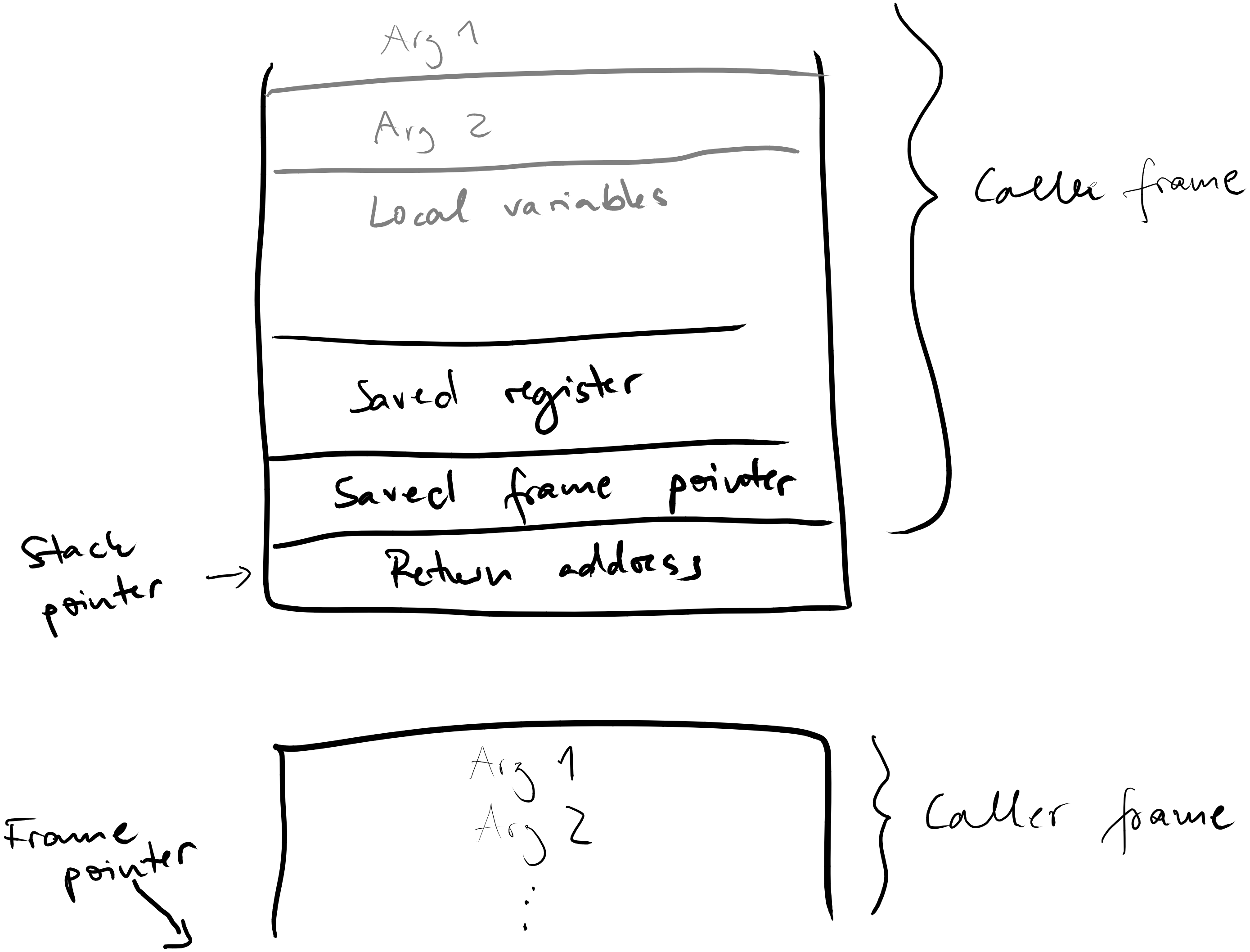
What is the address the callee uses to access the third argument?

Answer: `ebp` + 24 bytes

Typical Calling Sequence (3/4)

- **Steps performed by callee in epilogue**
  - Move return value into register or reserved location in stack
  - Restore registers (to state before call)
  - Restore frame pointer and stack pointer
  - Jump back to return address

Stack after epilogue



Typical Calling Sequence (4/4)

- **Steps performed by caller after the call**
  - Move return value to where it is needed
  - Restore registers (to state before call)

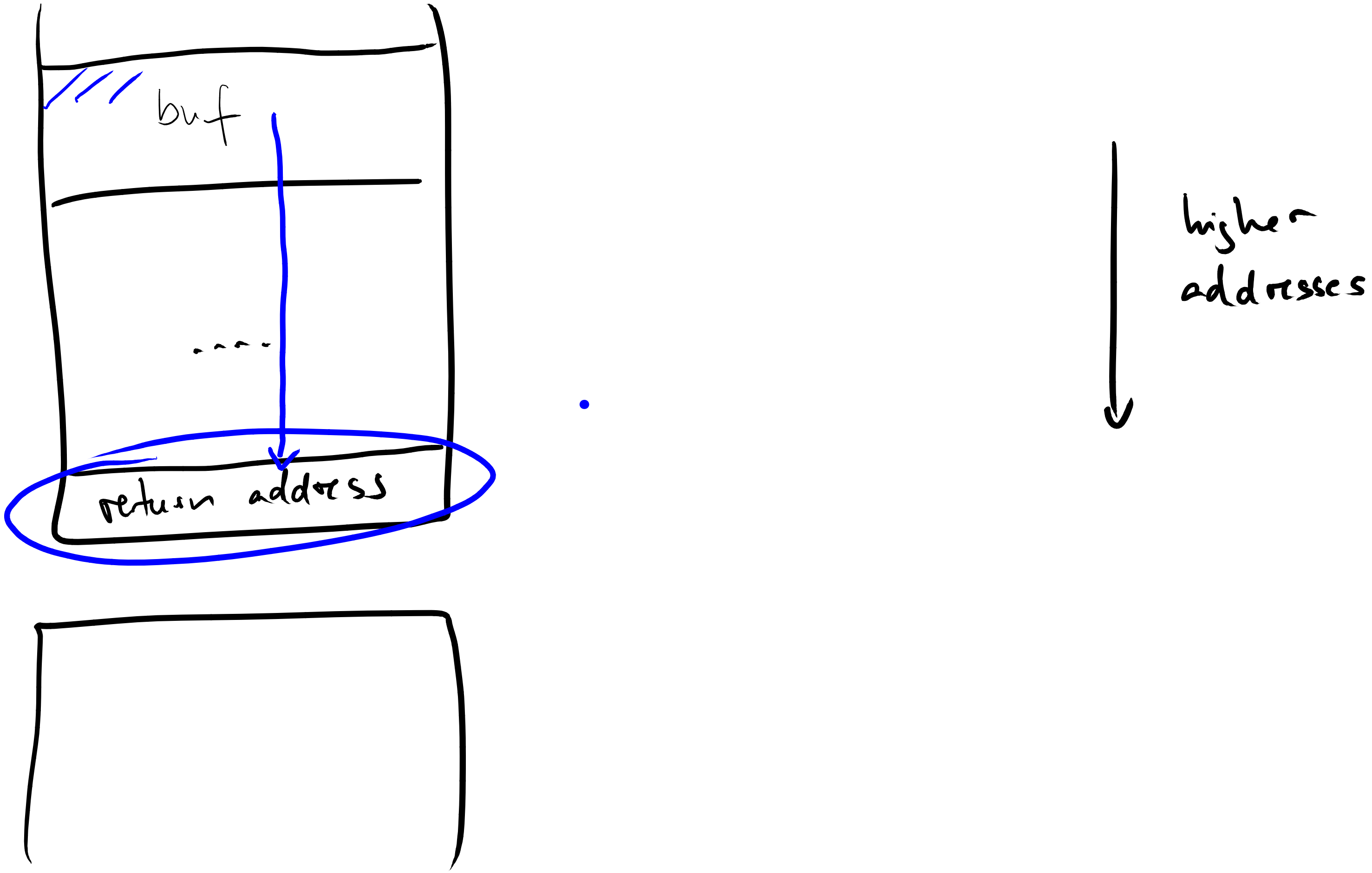
Application: Stack Smashing

- **Special kind of buffer overflow vulnerability**
  - Lack of bounds checking: May write beyond space allocated for a local variable
  - Malicious input can overwrite return address
  - Program can jump into malicious code

Example: Stack Smashing

```
int read_nb_from_file(FILE *s) {
  char buf[100];
  char *p = buf;
  do {
    /* read from stream s */
   *p = getc(s);
  } while (*p++ != '\n');
  *p = '\0';
  return atoi(buf);
}
```

Example: Stack Smashing



Overview

- Calling Sequences
- Coroutines ←
- Promises, Async, and Await

Coroutines

- **Control abstraction that allows for**
  - suspending execution
  - resuming where it was suspended
- **For implementing non-preemptive multi-tasking**

Example: Fibers in Ruby

```ruby
fiber1 = Fiber.new do
    puts "Fiber 1"
    Fiber.yield
    puts "Fiber 1 again"
end

fiber2 = Fiber.new do
    puts "Fiber 2"
    Fiber.yield
    puts "Fiber 2 again"
end

fiber1.resume
fiber2.resume
fiber2.resume
fiber1.resume
```

Example: Fibers in Ruby

```
fiber1 = Fiber.new do
    puts "Fiber 1"
    Fiber.yield
    puts "Fiber 1 again"
end

fiber2 = Fiber.new do
    puts "Fiber 2"
    Fiber.yield
    puts "Fiber 2 again"
end

fiber1.resume
fiber2.resume
fiber2.resume
fiber1.resume
```

Creates a coroutine (“fiber”)

Example: Fibers in Ruby

```
fiber1 = Fiber.new do
    puts "Fiber 1"
    Fiber.yield
    puts "Fiber 1 again"
end

fiber2 = Fiber.new do
    puts "Fiber 2"
    Fiber.yield
    puts "Fiber 2 again"
end

fiber1.resume
fiber2.resume
fiber2.resume
fiber1.resume
```

Continues to run a coroutine from where it last stopped

Example: Fibers in Ruby

```ruby
fiber1 = Fiber.new do
    puts "Fiber 1"
    Fiber.yield
    puts "Fiber 1 again"
end

fiber2 = Fiber.new do
    puts "Fiber 2"
    Fiber.yield
    puts "Fiber 2 again"
end

fiber1.resume
fiber2.resume
fiber2.resume
fiber1.resume
```

Passes control back to where the coroutine was resumed

Example: Fibers in Ruby

```
fiber1 = Fiber.new do
    puts "Fiber 1"
    Fiber.yield
    puts "Fiber 1 again"
end

fiber2 = Fiber.new do
    puts "Fiber 2"
    Fiber.yield
    puts "Fiber 2 again"
end

fiber1.resume
fiber2.resume
fiber2.resume
fiber1.resume
```

Prints:

Fiber 1

Fiber 2

Fiber 2 again

Fiber 1 again

Coroutines vs. Threads

- **Explicit transfer of control (non-preemptive)**
- **Only one coroutines runs at a time**

- **Control flow transfered implicitly and preemptively**
- **Multiple threads may run concurrently**

Coroutines vs. Continuations

- Changes every time it runs
- Old program counter saved when transfering to another coroutines
- When transfering back, continue where we left off

- Once created, doesn’t change
- When invoking, old program counter is lost
- Multiple jumps to same continuation always start at same position

Coroutines vs. Continuations

- Changes every time it runs
- Old program counter saved when transfering to another coroutines
- When transfering back, continue where we left off

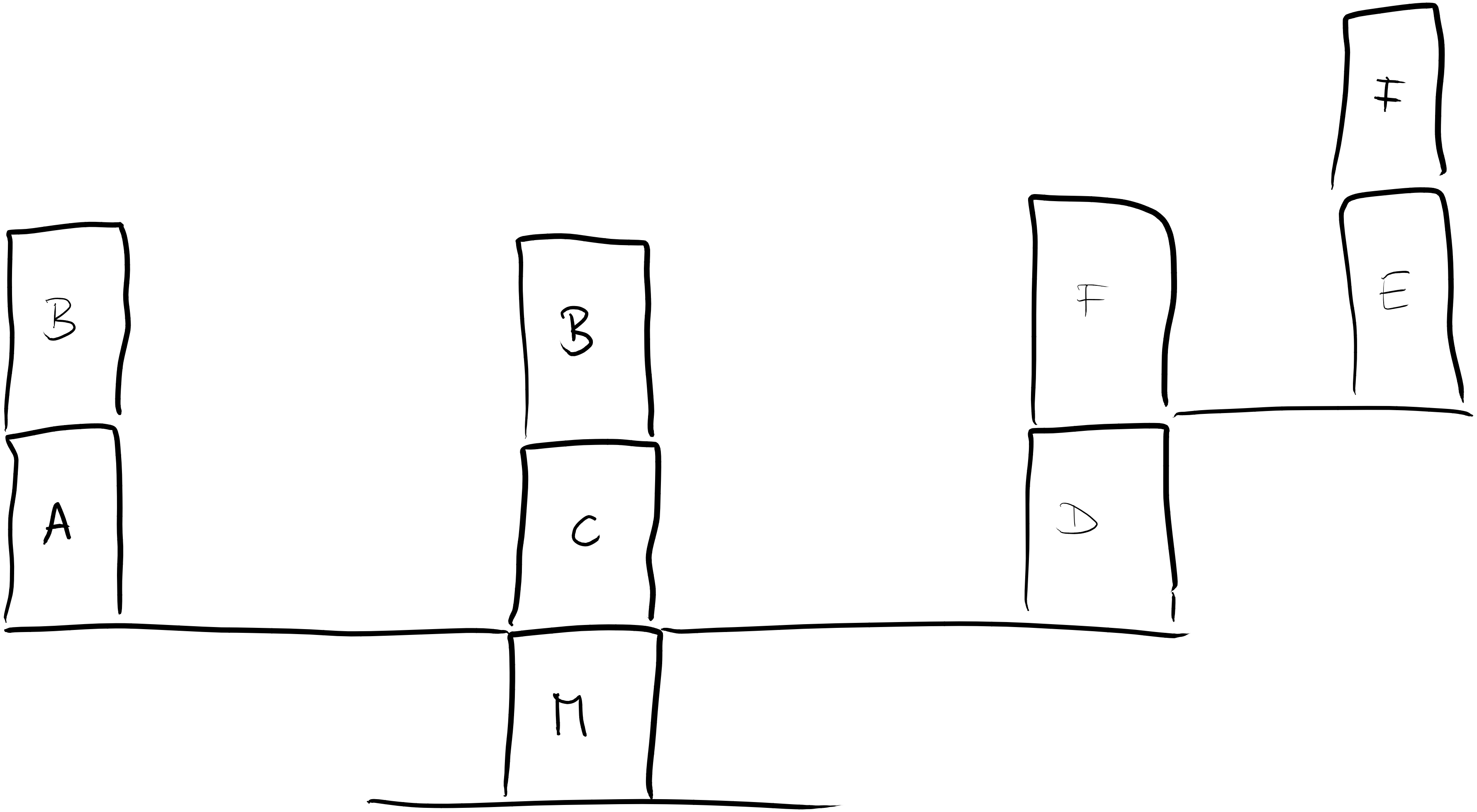
- Once created, doesn’t change
- When invoking, old program counter is lost
- Multiple jumps to same continuation always start at same position

Both: Represented by a closure
(= code address + referencing environment)

Stack Allocation

- **Coroutines may call subroutines and create other coroutines**
- **Each coroutine has its own function stack**
  - Second stack created when a routine creates a coroutine
- **Repeated creation of coroutines: "Cactus stack"**

Cactus Stack



Coroutines in Popular PLs

- **Natively** supported, e.g., in Ruby and Go
- Available as **libraries**, e.g., for Java, C#, JavaScript, Kotlin
- **Specialized variants**, e.g., in Python (generators)

Overview

- Calling Sequences
- Coroutines
- Promises, Async, and Await ←

Following slides partially adapted from Frank Tip

Motivation for Asynchrony

- **Parts of a program may take very long**
  - File I/O
  - Network I/O
  - Waiting for user input
- **Continue with rest of program until long-running parts are finished**

Expressing Asynchrony

- **Event-driven programming**
  - Register callbacks to invoke once finished
- **Promises (aka futures)**
  - Object to represent a not yet computed value
- **Async and await**
  - Syntactic sugar to ease programming with promises

Event-Driven Programming

Minimal example (JavaScript):

```
longRunning(someArg, (err, result) => {
  if (err == null)
    // handle error
  else
    // use result
})
```

Example: Sum of File Sizes

```
function computeSum(path) {
  let sum = 0; let count = 0;
  fs.readdir(path, (err, fileNames) => {










  });
}
```

Goal: Compute total size of all files in given directory

Example: Sum of File Sizes

```
function computeSum(path) {
  let sum = 0; let count = 0;
  fs.readdir(path, (err, fileNames) => {




  });
}
```

Goal: Compute total size of all files in given directory

Read files in directory and invoke callback once done

Example: Sum of File Sizes

```
function computeSum(path) {
  let sum = 0; let count = 0;
  fs.readdir(path, (err, fileNames) => {
    if (err === null){
```

Handle possible errors during file I/O

```
    } else
      console.log("I/O error: " + err);
  });
}
```

Example: Sum of File Sizes

```
function computeSum(path) {
  let sum = 0; let count = 0;
  fs.readdir(path, (err, fileNames) => {
    if (err === null) {
      for (let fileName of fileNames) {
        fs.stat(fileName, (err, fileInfo) => {



        });
      }
    } else
      console.log("I/O error: " + err);
  });
}
```

Get file information (incl. size) for each file in the directory

Example: Sum of File Sizes

```
function computeSum(path) {
  let sum = 0; let count = 0;
  fs.readdir(path, (err, fileNames) => {
    if (err === null){
      for (let fileName of fileNames) {
        fs.stat(fileName, (err, fileInfo) => {
          if (err === null) {




          } else
            console.log("I/O error: " + err);
        });
      }
    } else
      console.log("I/O error: " + err);
  });
}
```

Error handling again

Example: Sum of File Sizes

```
function computeSum(path) {
  let sum = 0; let count = 0;
  fs.readdir(path, (err, fileNames) => {
    if (err === null){
      for (let fileName of fileNames) {
        fs.stat(fileName, (err, fileInfo) => {
          if (err === null) {
              sum += fileInfo.size;
              count++;
              if (count == fileNames.length) {
                console.log(sum);
              }
          } else
            console.log("I/O error: " + err);
        });
      }
    } else
      console.log("I/O error: " + err);
  });
}
```

Synchronization: Ensure to write sum once callbacks for all files invoked

Problems

- Deeply nested control flow: "Callback hell"
- Error-handling scattered throughout code
- Need explicit synchronization when depending on multiple asynchronous computations

Promises

- Object that represents **result of asynchronous computation**
- Always in one of **three states**
  - Pending
  - Resolved
  - Rejected

  Settled

- Once **settled**, state doesn’t changed anymore

Minimal Example

```
// 1) Create a promise
let p = new Promise((resolve, reject) => {
  if (...)
    resolve(someValue);
  else
    reject(someError);
});
```

Functions to call for resolving/rejecting the promise

Minimal Example

```
// 1) Create a promise
let p = new Promise((resolve, reject) => {
  if (...)
    resolve(someValue);
  else
    reject(someError);
});

// 2) Use the promise
p.then((x) => {
  // use resulting value
}).catch((e) => {
  // handle error
});
```

Register *reaction* invoked when promise is resolved/rejected

Example: Sum of File Sizes

```
function computeSum(path) {
  fs.readdir(path).then((fileNames) => {



  })



}
```

Now using the promise version of `fs` APIs, which return promises

Example: Sum of File Sizes

```
function computeSum(path) {
  fs.readdir(path).then((fileNames) => {
    const promises = fileNames.map((fn) => fs.stat(fn));
    // wait for all of them to be resolved
    return Promise.all(promises);
  })
}
```

Each call returns a promise

Returns a single promise
once all given promises
are resolved

Example: Sum of File Sizes

```
function computeSum(path) {
  fs.readdir(path).then((fileNames) => {
    const promises = fileNames.map((fn) => fs.stat(fn));
    // wait for all of them to be resolved
    return Promise.all(promises);
  }).then((fileInfos) => {


  })

}
```

Chain multiple promises:
Reactions registered with `then` are executed sequentially

Example: Sum of File Sizes

```
function computeSum(path) {
  fs.readdir(path).then((fileNames) => {
    const promises = fileNames.map((fn) => fs.stat(fn));
    // wait for all of them to be resolved
    return Promise.all(promises);
  }).then((fileInfos) => {
    // compute sum
    const sum = fileInfos.reduce((acc, val) =>
      { return acc + val.size; }, 0);
    console.log(sum);
  })


}
```

Example: Sum of File Sizes

```
function computeSum(path) {
  fs.readdir(path).then((fileNames) => {
    const promises = fileNames.map((fn) => fs.stat(fn));
    // wait for all of them to be resolved
    return Promise.all(promises);
  }).then((fileInfos) => {
    // compute sum
    const sum = fileInfos.reduce((acc, val) =>
      { return acc + val.size; }, 0);
    console.log(sum);
  }).catch((e) => {
    console.log("error: " + e);
  });
}
```

Handles errors in any previous promises in the chain

Pros and Cons

- **Benefits over event-driven code**
  - Control flow now easier to understand
  - Explicit synchronization using `Promise.all`
  - All error handling in one place
- **Still suboptimal:**
  - Somewhat verbose syntax due to higher-order functions

Async and Await

- **Label function as `async` if it performs asynchronous computation**
  - □ Returns a promise
  - □ May `await` other asynchronous computations
  - □ No need for higher-order `then` and `catch` functions
  - □ Error handling using standard `try` and `catch`

Minimal Example

```
async function longRunning() {
  return someValue;
}
```

Returns a promise

```
// in some other async function:
let result = await longRunning();
```

Waits for the promise to resolve

Example: Sum of File Sizes

```
async function computeSum(path) {

    const fileNames = await fs.readdir(path);
    const promises = fileNames.map((fn) => fs.stat(fn));
    const fileInfos = await Promise.all(promises);
    const sum = fileInfos.reduce((acc, val) =>
      { return acc + val.size; }, 0);
    console.log(sum);



}
```

Looks like sequential control flow, but execution isn’t blocked on await expression

Example: Sum of File Sizes

```
async function computeSum(path) {
  try {
    const fileNames = await fs.readdir(path);
    const promises = fileNames.map((fn) => fs.stat(fn));
    const fileInfos = await Promise.all(promises);
    const sum = fileInfos.reduce((acc, val) =>
      { return acc + val.size; }, 0);
    console.log(sum);
  } catch(e) {
    console.log("error: " + e);
  }
}
```

Error handling via standard `try` and `catch`

Quiz: Promises, Async, and Await

Which of the following statements is true?

- The value represented by a promise will exists eventually.
- The semantics of `async` and `await` can be explained in terms of promises.
- All `await` expressions are evaluated in parallel.
- Chained promises are executed concurrently.

Quiz: Promises, Async, and Await

Which of the following statements is true?

- ~~The value represented by a promise will exists eventually.~~
- The semantics of `async` and `await` can be explained in terms of promises.
- ~~All `await` expressions are evaluated in parallel.~~
- ~~Chained promises are executed concurrently.~~

Overview

- Calling Sequences
- Coroutines
- Promises, Async, and Await ✔