

Asynchronous Programming

Promises + Futures

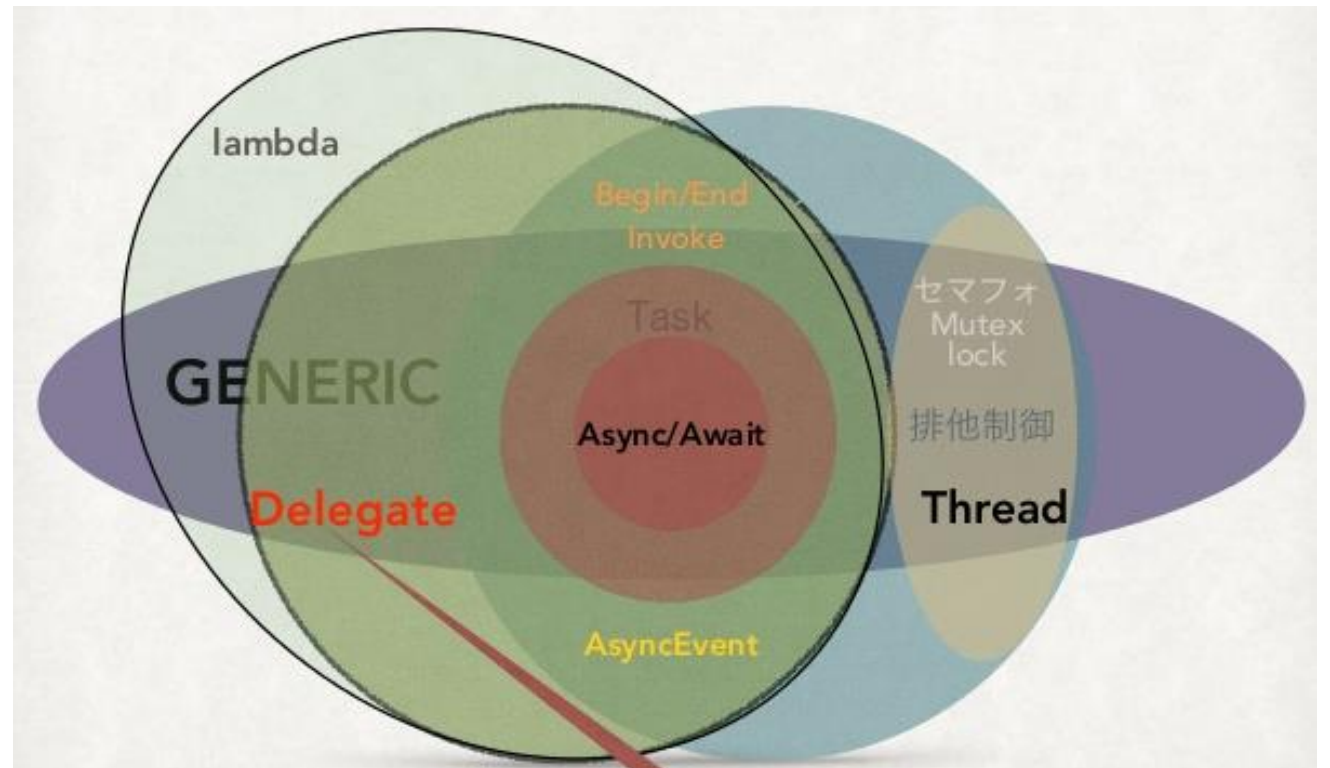
Consistency

Chris Rossbach

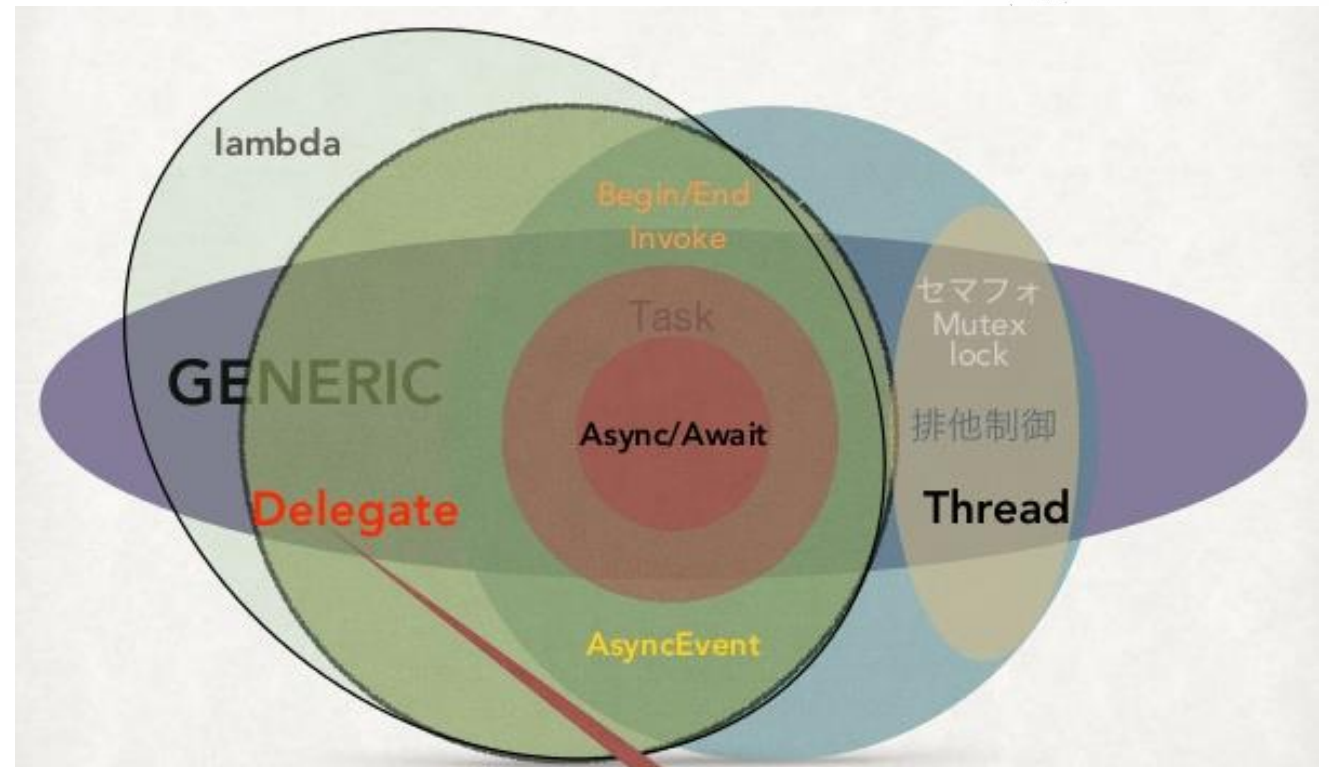
Today

- Questions?
- Administrivia
 - Due dates shifted
- Material for the day
 - Events / Asynchronous programming
 - Promises & Futures
 - Bonus: memory consistency models
- Acknowledgements
 - Consistency slides borrow some materials from Kevin Boos. Thanks!

Asynchronous Programming Events, Promises, and Futures



Asynchronous Programming Events, Promises, and Futures



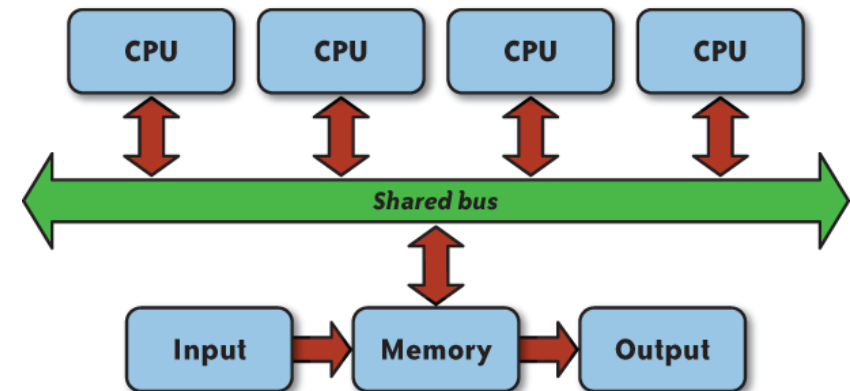
Programming Models for Concurrency

Programming Models for Concurrency

- Hardware execution model:

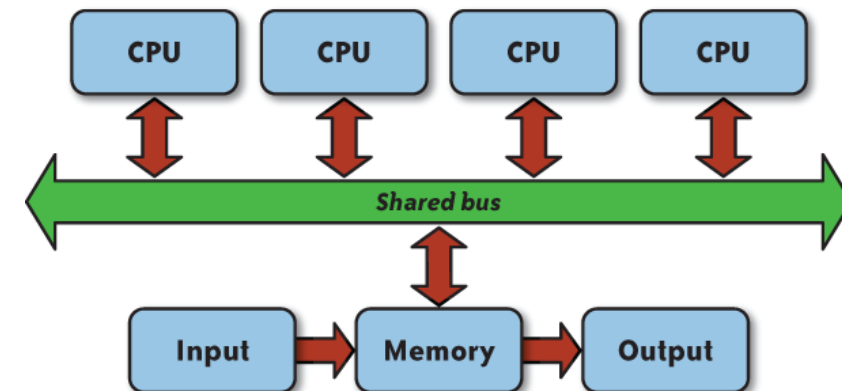
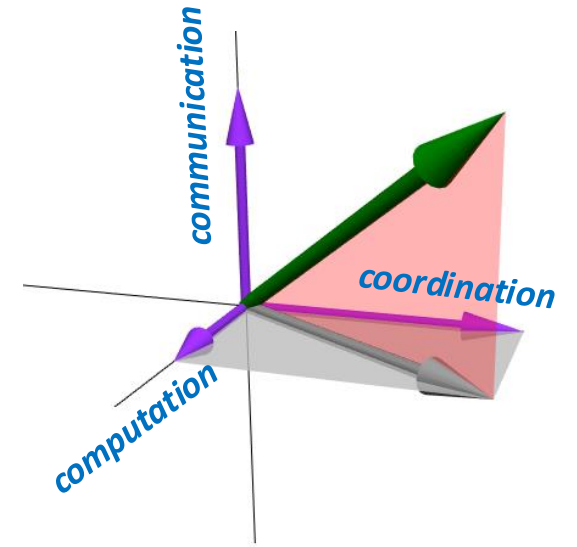
Programming Models for Concurrency

- Hardware execution model:
 - CPU(s) execute instructions sequentially



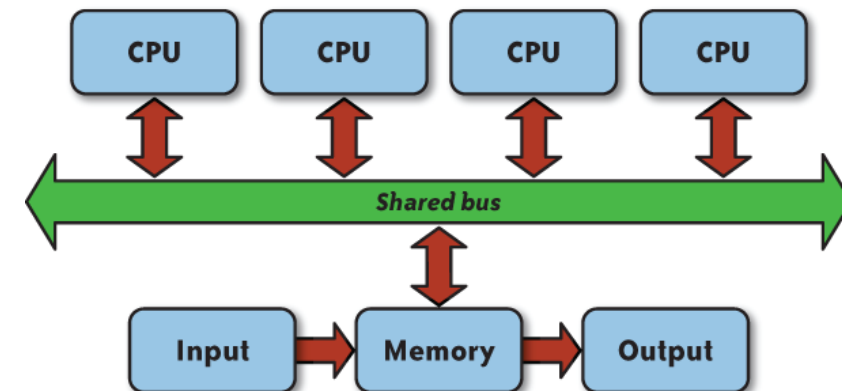
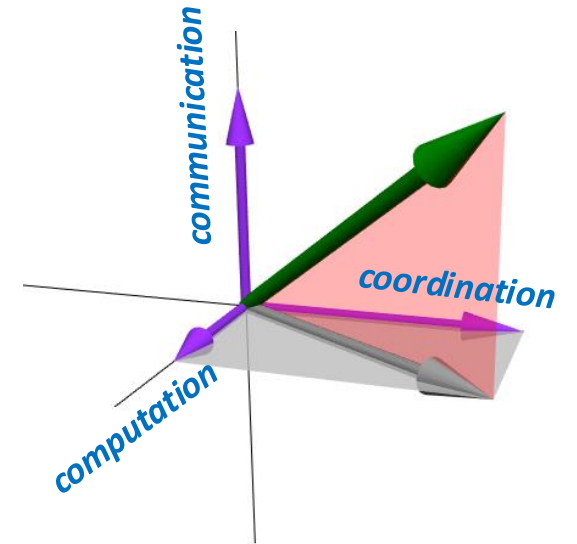
Programming Models for Concurrency

- Hardware execution model:
 - CPU(s) execute instructions sequentially
- Programming model dimensions:
 - How to specify computation
 - How to specify communication
 - How to specify coordination/control transfer



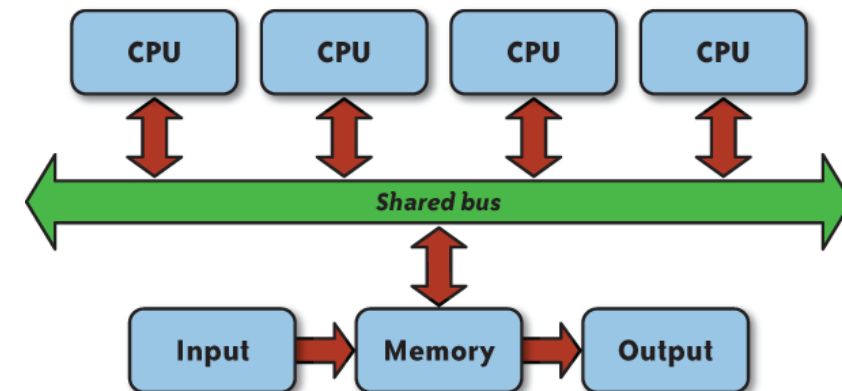
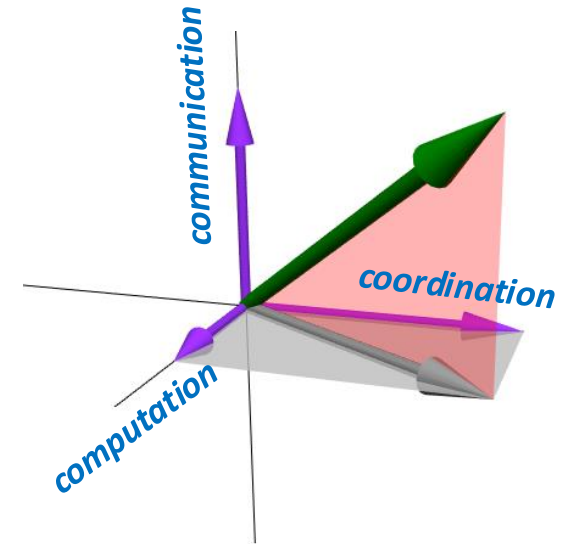
Programming Models for Concurrency

- Hardware execution model:
 - CPU(s) execute instructions sequentially
- Programming model dimensions:
 - How to specify computation
 - How to specify communication
 - How to specify coordination/control transfer
- Techniques/primitives
 - Message passing vs shared memory
 - Preemption vs Non-preemption



Programming Models for Concurrency

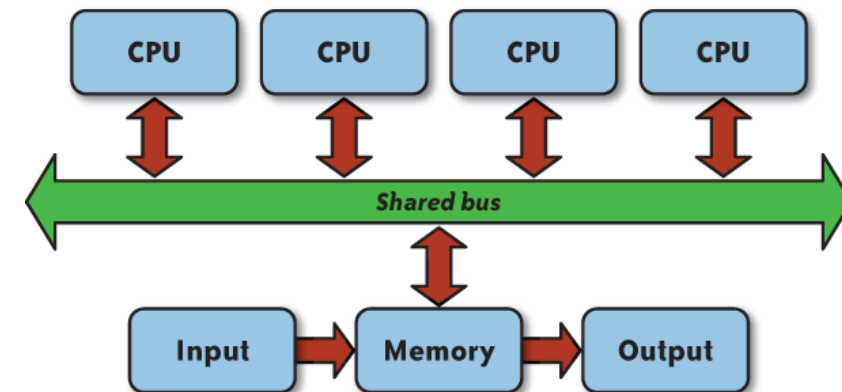
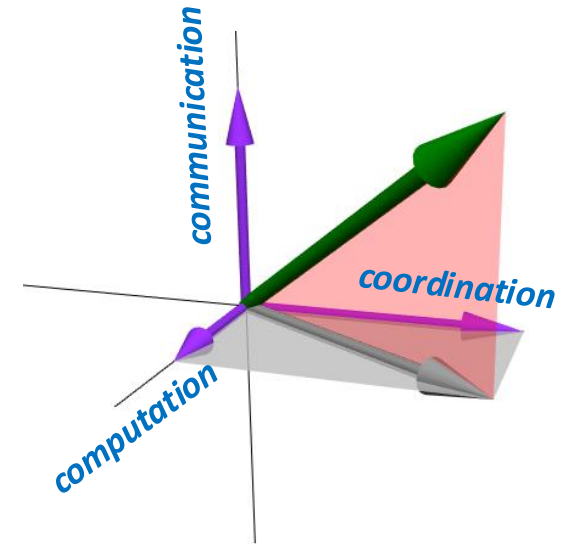
- Hardware execution model:
 - CPU(s) execute instructions sequentially
- Programming model dimensions:
 - How to specify computation
 - How to specify communication
 - How to specify coordination/control transfer
- Techniques/primitives
 - Message passing vs shared memory
 - Preemption vs Non-preemption
- Dimensions/techniques not always orthogonal



Programming Models for Concurrency

- Hardware execution model:
 - CPU(s) execute instructions sequentially
- Programming model dimensions:
 - How to specify computation
 - How to specify communication
 - How to specify coordination/control transfer
- Techniques/primitives
 - Message passing vs shared memory
 - Preemption vs Non-preemption
- Dimensions/techniques not always orthogonal

Futures & Promises touch all three dimension



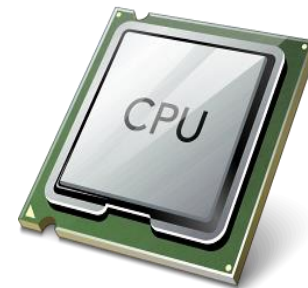
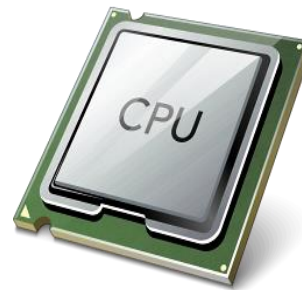
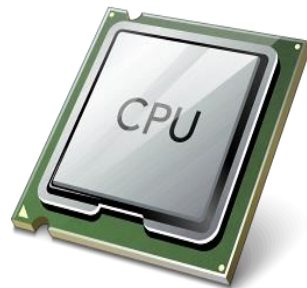
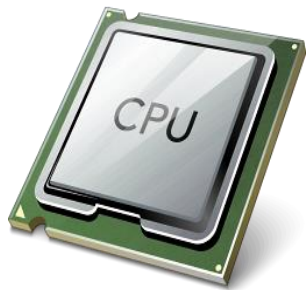
GUI Programming Distilled

```
1  winmain(...) {  
2      while(true) {  
3          message = GetMessage();  
4          switch(message) {  
5              case WM_THIS: DoThis(); break;  
6              case WM_THAT: DoThat(); break;  
7              case WM_OTHERTHING: DoOtherThing(); break;  
8              case WM_DONE: return;  
9          }  
10     }  
11 }
```

GUI Programming Distilled

```
1  winmain(...) {  
2      while(true) {  
3          message = GetMessage();  
4          switch(message) {  
5              case WM_THIS: DoThis(); break;  
6              case WM_THAT: DoThat(); break;  
7              case WM_OTHERTHING: DoOtherThing(); break;  
8              case WM_DONE: return;  
9          }  
10     }  
11 }
```

How can we
parallelize
this?

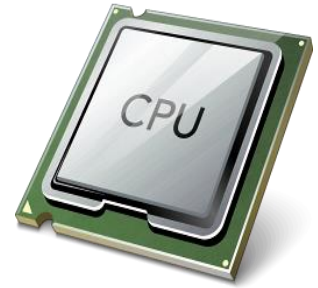


Parallel GUI Implementation 1

```
1 winmain(...) {  
2     while(true) {  
3         message = GetMessage();  
4         switch(message) {  
5             case WM_THIS: DoThis(); break;  
6             case WM_THAT: DoThat(); break;  
7             case WM_OTHERTHING: DoOtherThing(); break;  
8             case WM_DONE: return;  
9         }  
10    }  
11 }
```

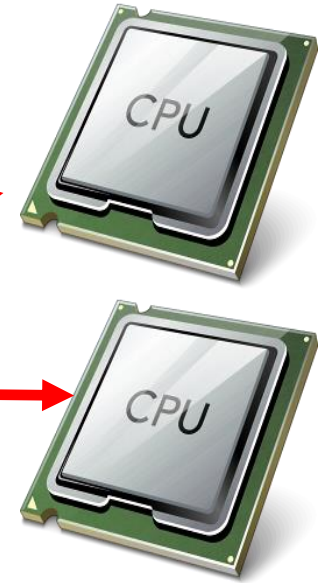
Parallel GUI Implementation 1

```
1 winmain(...) {  
2     while(true) {  
3         message = GetMessage();  
4         switch(message) {  
5             case WM_THIS: DoThis(); break;  
6             case WM_THAT: DoThat(); break;  
7             case WM_OTHERTHING: DoOtherThing(); break;  
8             case WM_DONE: return;  
9         }  
10    }  
11 }
```



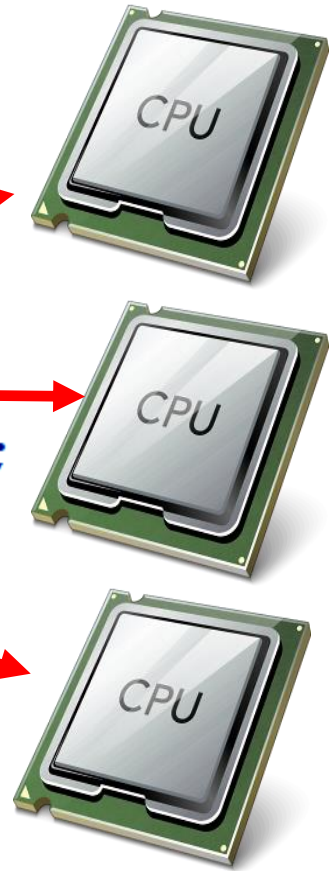
Parallel GUI Implementation 1

```
1 winmain(...) {  
2   while(true) {  
3     message = GetMessage();  
4     switch(message) {  
5       case WM_THIS: DoThis(); break;  
6       case WM_THAT: DoThat(); break;  
7       case WM_OTHERTHING: DoOtherThing(); break;  
8       case WM_DONE: return;  
9     }  
10  }  
11 }
```



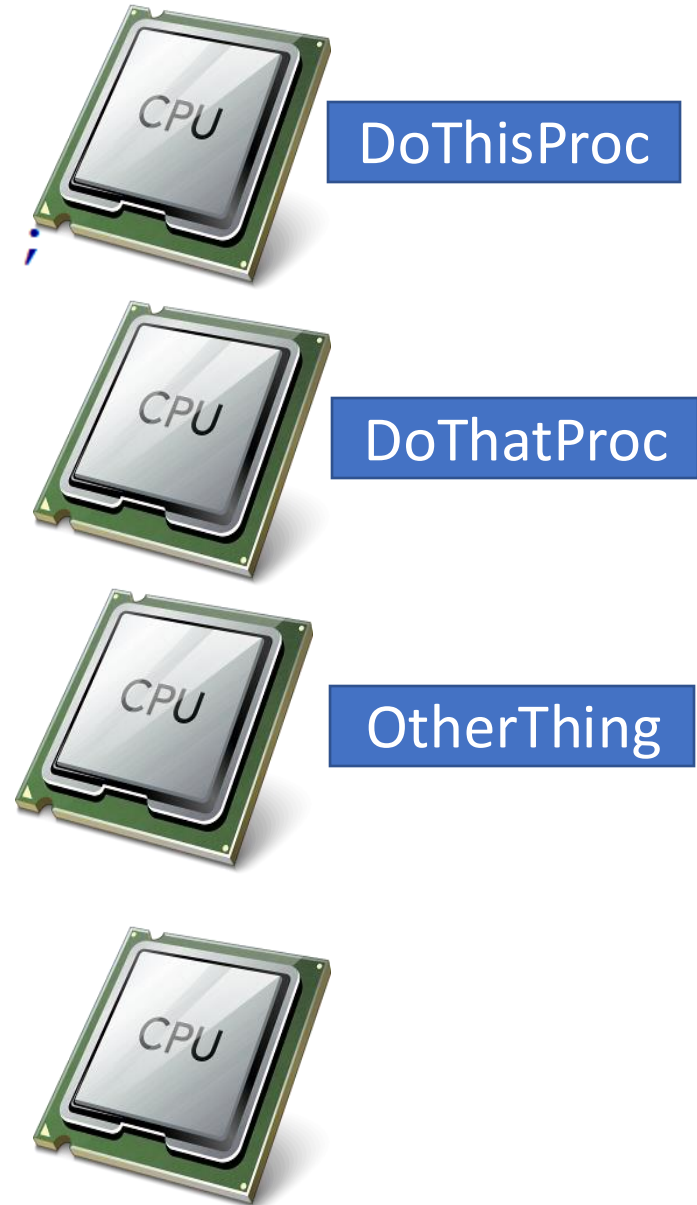
Parallel GUI Implementation 1

```
1 winmain(...) {  
2   while(true) {  
3     message = GetMessage();  
4     switch(message) {  
5       case WM_THIS: DoThis(); break;  
6       case WM_THAT: DoThat(); break;  
7       case WM_OTHERTHING: DoOtherThing(); break;  
8       case WM_DONE: return;  
9     }  
10  }  
11 }
```



Parallel GUI Implementation 1

```
winmain() {  
    pthread_create(&tids[i++], DoThisProc);  
    pthread_create(&tids[i++], DoThatProc);  
    pthread_create(&tids[i++], DoOtherThingProc);  
    for(j=0; j<i; j++)  
        pthread_join(&tids[j]);  
}  
  
DoThisProc() {  
    while(true) {  
        if(ThisHasHappened)  
            DoThis();  
    }  
}
```

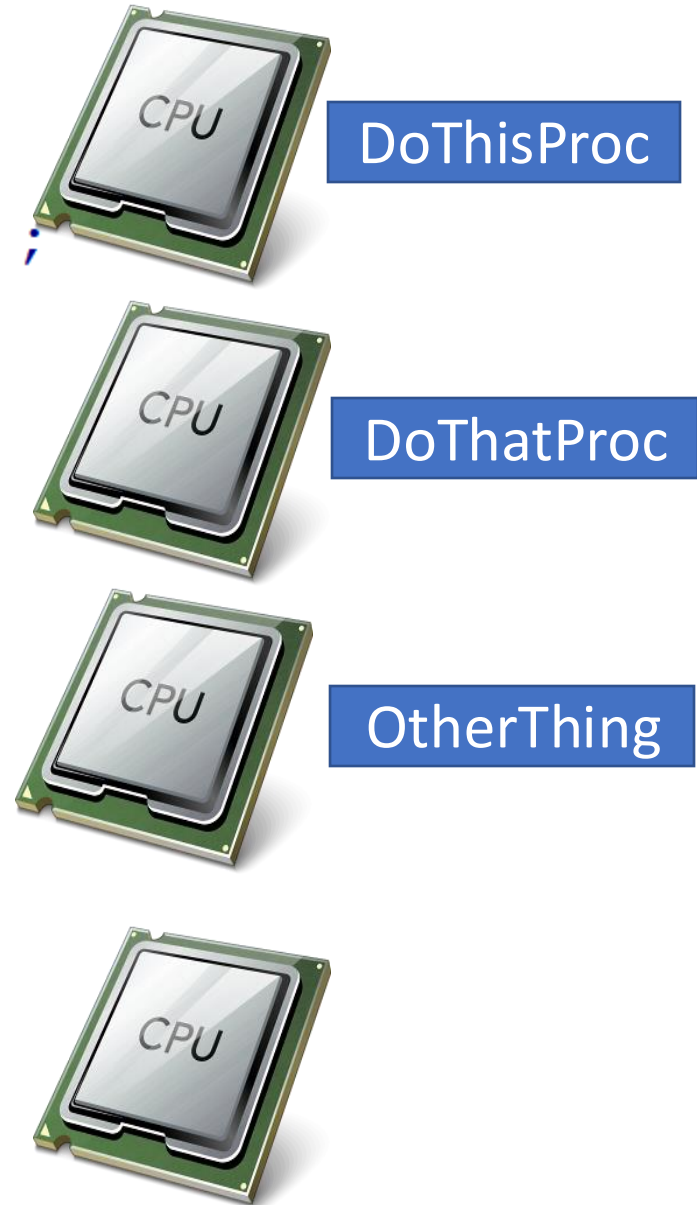


Parallel GUI Implementation 1

```
winmain() {  
    pthread_create(&tids[i++], DoThisProc);  
    pthread_create(&tids[i++], DoThatProc);  
    pthread_create(&tids[i++], DoOtherThingProc);  
    for(j=0; j<i; j++)  
        pthread_join(&tids[j]);  
}
```

Pros/cons?

```
DoThisProc() {  
    while(true) {  
        if(ThisHasHappened)  
            DoThis();  
    }  
}
```



Parallel GUI Implementation 1

```
winmain() {  
    pthread_create(&tids[i++], DoThisProc);  
    pthread_create(&tids[i++], DoThatProc);  
    pthread_create(&tids[i++], DoOtherThingProc);  
    for(j=0; j<i; j++)  
        pthread_join(&tids[j]);  
}  
  
DoThisProc() {  
    while(true) {  
        if(ThisHasHappened) {  
            DoThis();  
        }  
    }  
}
```

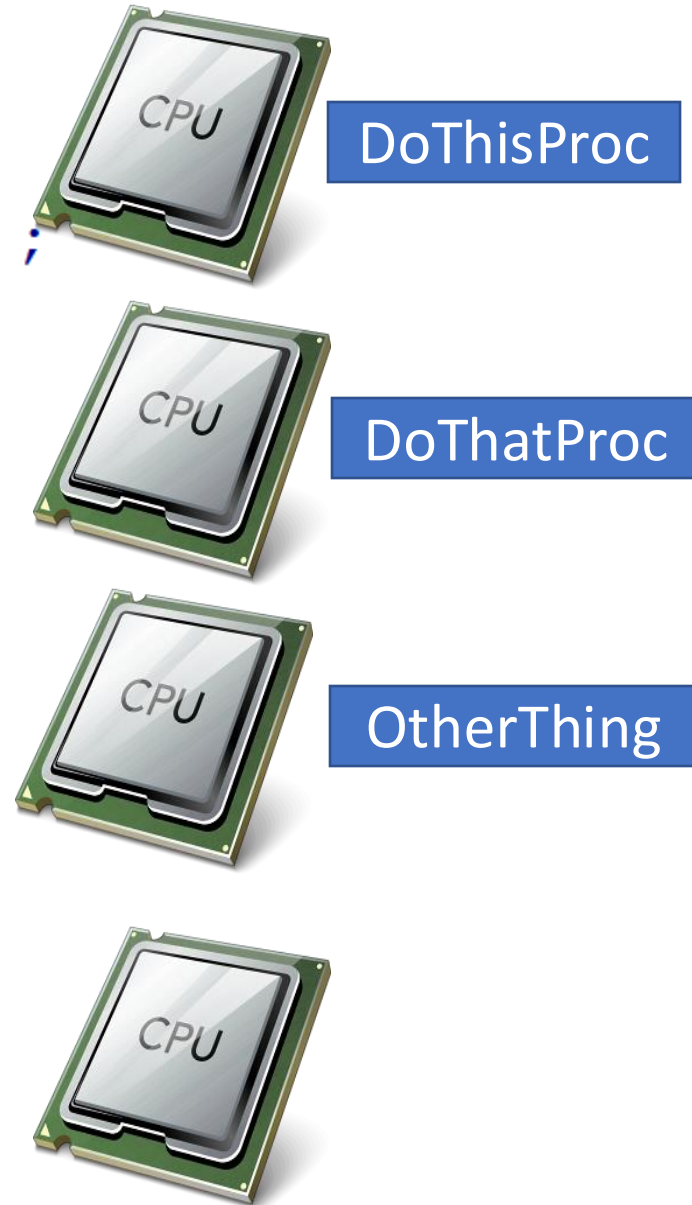
Pros/cons?

Pros:

- Encapsulates parallel work

Cons:

- Obliterates original code structure
- How to assign handlers → CPUs?
- Load balance?!?
- Utilization



Parallel GUI Implementation 2

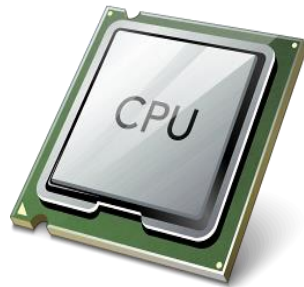
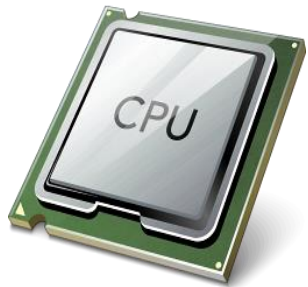
```
winmain() {  
    for(i=0; i<NUMPROCS; i++)  
        pthread_create(&tids[i], HandlerProc);  
    for(i=0; i<NUMPROCS; i++)  
        pthread_join(&tids[i]);  
}
```

```
threadproc(...) {  
    while(true) {  
        message = GetMessage();  
        switch(message) {  
            case WM_THIS: DoThis();  
            case WM_THAT: DoThat();  
        }  
    }  
}
```

```
threadproc(...) {  
    while(true) {  
        message = GetMessage();  
        switch(message) {  
            case WM_THIS: DoThis();  
            case WM_THAT: DoThat();  
        }  
    }  
}
```

```
threadproc(...) {  
    while(true) {  
        message = GetMessage();  
        switch(message) {  
            case WM_THIS: DoThis();  
            case WM_THAT: DoThat();  
        }  
    }  
}
```

```
threadproc(...) {  
    while(true) {  
        message = GetMessage();  
        switch(message) {  
            case WM_THIS: DoThis();  
            case WM_THAT: DoThat();  
        }  
    }  
}
```



Parallel GUI Implementation 2

Pros/cons?

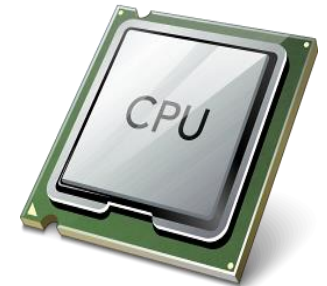
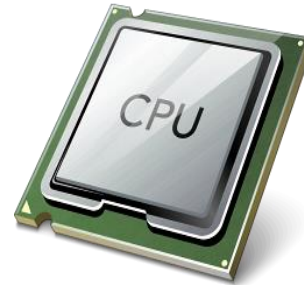
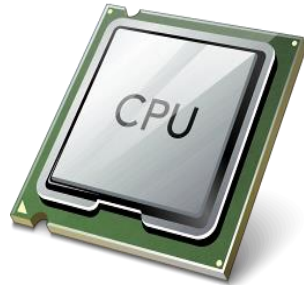
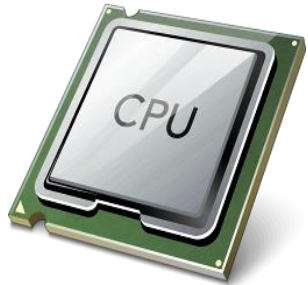
```
winmain() {  
    for(i=0; i<NUMPROCS; i++)  
        pthread_create(&tids[i], HandlerProc);  
    for(i=0; i<NUMPROCS; i++)  
        pthread_join(&tids[i]);  
}
```

```
threadproc(...) {  
    while(true) {  
        message = GetMessage();  
        switch(message) {  
            case WM_THIS: DoThis();  
            case WM_THAT: DoThat();  
        }  
    }  
}
```

```
threadproc(...) {  
    while(true) {  
        message = GetMessage();  
        switch(message) {  
            case WM_THIS: DoThis();  
            case WM_THAT: DoThat();  
        }  
    }  
}
```

```
threadproc(...) {  
    while(true) {  
        message = GetMessage();  
        switch(message) {  
            case WM_THIS: DoThis();  
            case WM_THAT: DoThat();  
        }  
    }  
}
```

```
threadproc(...) {  
    while(true) {  
        message = GetMessage();  
        switch(message) {  
            case WM_THIS: DoThis();  
            case WM_THAT: DoThat();  
        }  
    }  
}
```



Parallel GUI Implementation 2

Pros/cons?

```
winmain() {  
    for(i=0; i<NUMPROCS; i++)  
        pthread_create(&tids[i], H  
    for(i=0; i<NUMPROCS; i++)  
        pthread_join(&tids[i]);  
}
```

Pros:

- Preserves programming model
- Can recover some parallelism

Cons:

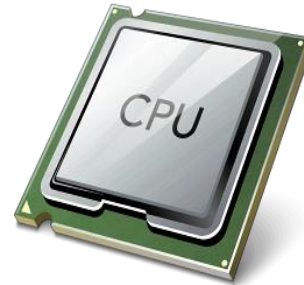
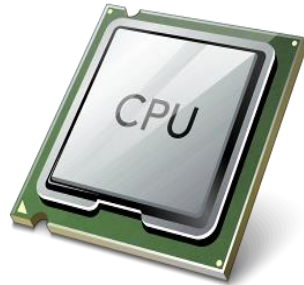
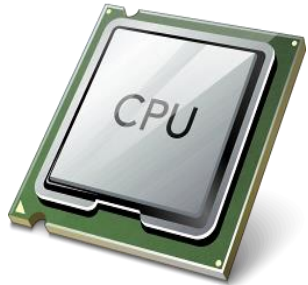
- Workers still have same problem
- How to load balance?
- Shared mutable state a problem

```
threadproc(...) {  
    while(true) {  
        message = GetMessage();  
        switch(message) {  
            case WM_THIS: DoThis();  
            case WM_THAT: DoThat();  
        }  
    }  
}
```

```
threadproc(...) {  
    while(true) {  
        message = GetMessage();  
        switch(message) {  
            case WM_THIS: DoThis();  
            case WM_THAT: DoThat();  
        }  
    }  
}
```

```
threadproc(...) {  
    while(true) {  
        message = GetMessage();  
        switch(message) {  
            case WM_THIS: DoThis();  
            case WM_THAT: DoThat();  
        }  
    }  
}
```

```
threadproc(...) {  
    while(true) {  
        message = GetMessage();  
        switch(message) {  
            case WM_THIS: DoThis();  
            case WM_THAT: DoThat();  
        }  
    }  
}
```



Parallel GUI Implementation 2

Pros/cons?

```
winmain() {  
    for(i=0; i<NUMPROCS; i++)  
        pthread_create(&tids[i], H  
    for(i=0; i<NUMPROCS; i++)  
        pthread_join(&tids[i]);  
}
```

Pros:

- Preserves programming model
- Can recover some parallelism

Cons:

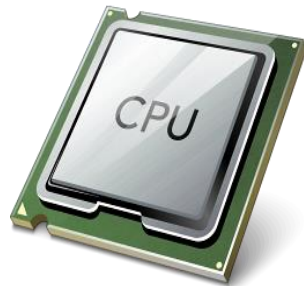
- Workers still have same problem
- How to load balance?
- Shared mutable state a problem

```
threadproc(...) {  
    while(true) {  
        message = GetMessage();  
        switch(message) {  
            case WM_THIS: DoThis();  
            case WM_THAT: DoThat();  
        }  
    }  
}
```

```
threadproc(...) {  
    while(true) {  
        message = GetMessage();  
        switch(message) {  
            case WM_THIS: DoThis();  
            case WM_THAT: DoThat();  
        }  
    }  
}
```

```
threadproc(...) {  
    while(true) {  
        message = GetMessage();  
        switch(message) {  
            case WM_THIS: DoThis();  
            case WM_THAT: DoThat();  
        }  
    }  
}
```

```
threadproc(...) {  
    while(true) {  
        message = GetMessage();  
        switch(message) {  
            case WM_THIS: DoThis();  
            case WM_THAT: DoThat();  
        }  
    }  
}
```



*Extremely difficult to solve
without changing the whole
programming model...so*

change it

Event-based Programming: Motivation

Event-based Programming: Motivation

- Threads have a **lot** of down-sides:
 - Tuning parallelism for different environments
 - Load balancing/assignment brittle
 - Shared state requires locks →
 - Priority inversion
 - Deadlock
 - Incorrect synchronization
 - ...

Event-based Programming: Motivation

- Threads have a **lot** of down-sides:
 - Tuning parallelism for different environments
 - Load balancing/assignment brittle
 - Shared state requires locks →
 - Priority inversion
 - Deadlock
 - Incorrect synchronization
 - ...
- Events: *restructure programming model so threads are not exposed!*

Event Programming Model Basics

Event Programming Model Basics

- Programmer *only writes events*

Event Programming Model Basics

- Programmer *only writes events*
- Event: an object queued for a module (think future/promise)

Event Programming Model Basics

- Programmer *only writes events*
- Event: an object queued for a module (think future/promise)
- Basic primitives
 - `create_event_queue(handler) → event_q`
 - `enqueue_event(event_q, event-object)`
 - Invokes handler (eventually)

Event Programming Model Basics

- Programmer *only writes events*
- Event: an object queued for a module (think future/promise)
- Basic primitives
 - `create_event_queue(handler) → event_q`
 - `enqueue_event(event_q, event-object)`
 - Invokes handler (eventually)
- Scheduler decides which event to execute next
 - E.g. based on priority, CPU usage, etc.

Event-based programming

Event-based programming

```
switch (message)
{
    //case WM_COMMAND:
    // handle menu selections etc.
    //break;
    //case WM_PAINT:
    // draw our window - note: you must paint something here or not trap it!
    //break;
    case WM_DESTROY:
        PostQuitMessage(0);
    break;
    default:
        // We do not want to handle this message so pass back to Windows
        // to handle it in a default way
        return DefWindowProc(hWnd, message, wParam, lParam);
}
```

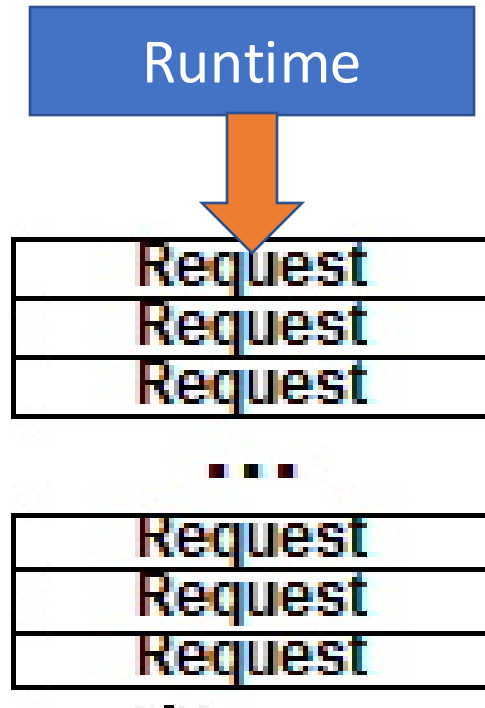
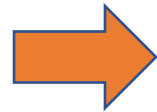
Event-based programming

Event-based programming

```
PROGRAM MyProgram {  
    OnSize () {}  
    OnMove () {}  
    OnClick () {}  
    OnPaint () {}  
}
```

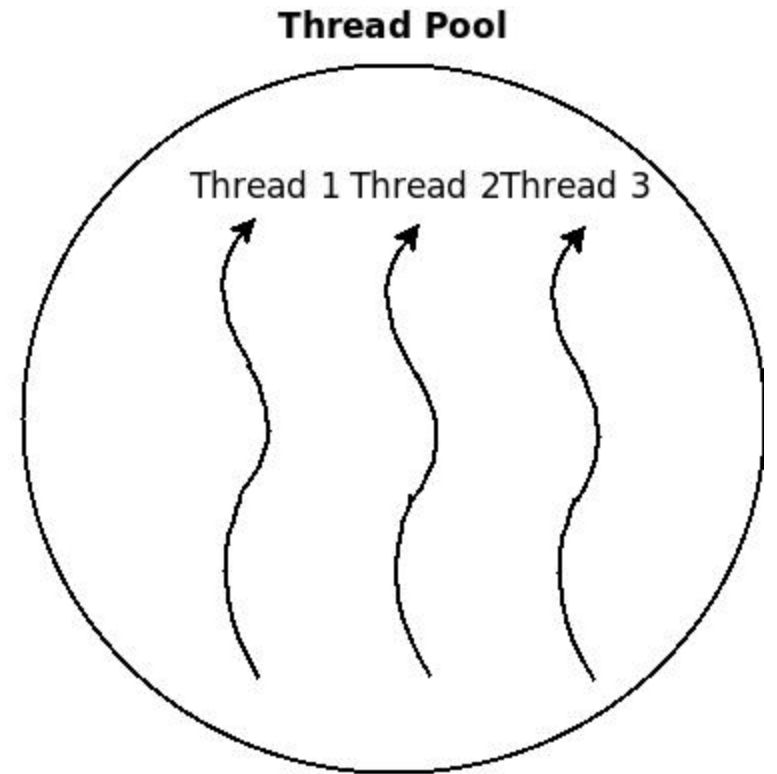
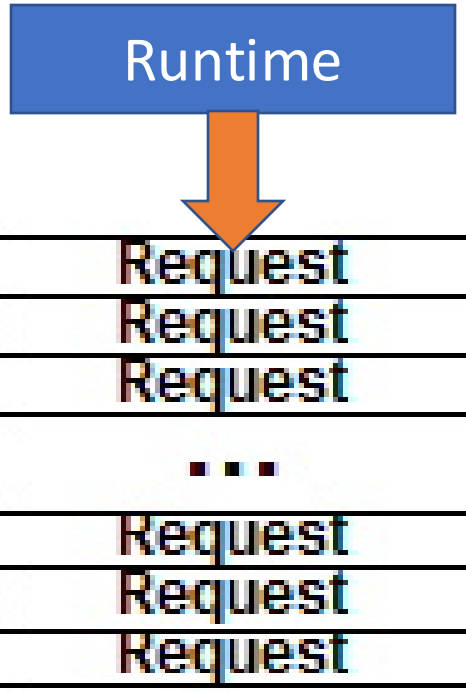
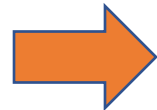
Event-based programming

```
PROGRAM MyProgram {  
    OnSize () {}  
    OnMove () {}  
    OnClick () {}  
    OnPaint () {}  
}
```

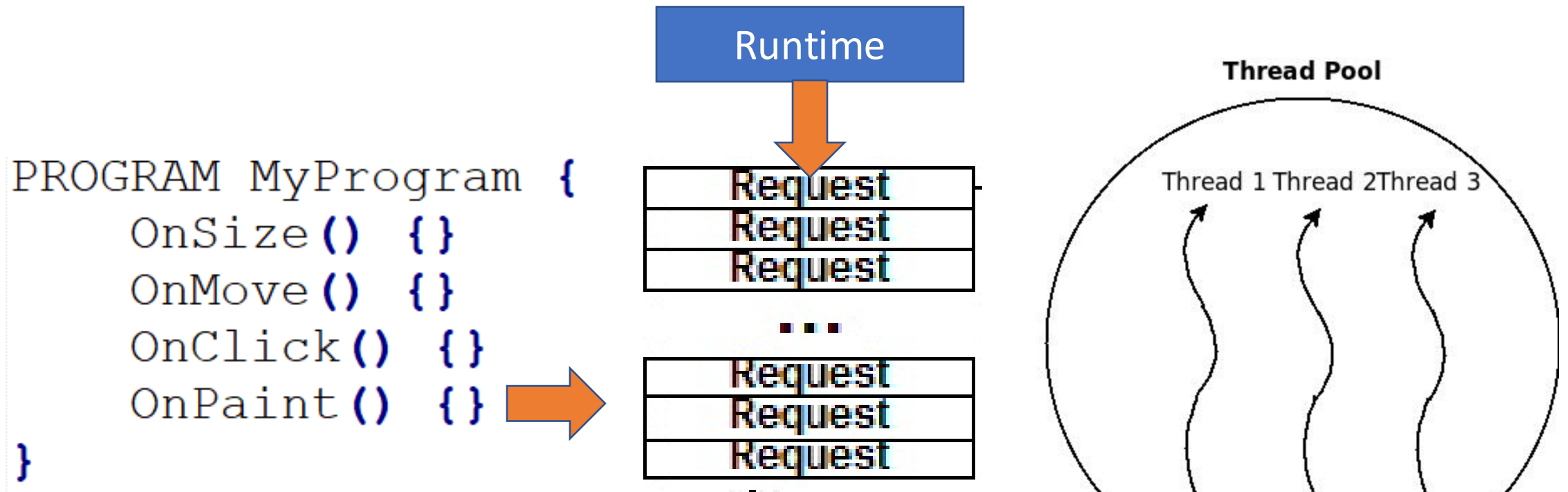


Event-based programming

```
PROGRAM MyProgram {  
    OnSize () {}  
    OnMove () {}  
    OnClick () {}  
    OnPaint () {}  
}
```



Event-based programming



Is the problem solved?

Another Event-based Program

Another Event-based Program

```
1 PROGRAM MyProgram {
2     OnOpenFile () {
3         char szFileName [BUFSIZE]
4         InitFileName (szFileName) ;
5         FILE file = ReadFileEx (szFileName) ;
6         LoadFile (file) ;
7         RedrawScreen () ;
8     }
9     OnPaint () ;
10 }
```

Another Event-based Program

```
1 PROGRAM MyProgram {
2     OnOpenFile () {
3         char szFileName [BUFSIZE]
4         InitFileName (szFileName);
5         FILE file = ReadFileEx (szFileName);
6         LoadFile (file);
7         RedrawScreen ();
8     }
9     OnPaint ();
10 }
```



Blocks!

Another Event-based Program

```
1 PROGRAM MyProgram {
2   OnOpenFile () {
3     char szFileName [BUFSIZE]
4     InitFileName (szFileName);
5     FILE file = ReadFileEx (szFileName);
6     LoadFile (file);
7     RedrawScreen ();
8   }
9   OnPaint ();
10 }
```

Burns CPU!

Blocks!

Another Event-based Program

```
1 PROGRAM MyProgram {
2   OnOpenFile () {
3     char szFileName [BUFSIZE]
4     InitFileName (szFileName);
5     FILE file = ReadFileEx (szFileName);
6     LoadFile (file);
7     RedrawScreen ();
8   }
9   OnPaint ();
10 }
```

Uses Other Handlers!
(call OnPaint?)

Burns CPU!

Blocks!

No problem!

Just use more events/handlers, right?

```
1 PROGRAM MyProgram {
2     TASK ReadFileAsync(name, callback) {
3         ReadFileSync(name);
4         Call(callback);
5     }
6     CALLBACK FinishOpeningFile() {
7         LoadFile(file);
8         RedrawScreen();
9     }
10    OnOpenFile() {
11        FILE file;
12        char szName[BUFSIZE]
13        InitFileName(szName);
14        EnqueueTask(ReadFileAsync(szName, FinishOpeningFile));
15    }
16    OnPaint();
17 }
```

Continuations, BTW

```
1 PROGRAM MyProgram {  
2     OnOpenFile () {  
3         ReadFile (file, FinishOpeningFile);  
4     }  
5     OnFinishOpeningFile () {  
6         LoadFile (file, OnFinishLoadingFile);  
7     }  
8     OnFinishLoadingFile () {  
9         RedrawScreen ();  
10    }  
11    OnPaint ();  
12 }
```

Stack-Ripping

```
1 PROGRAM MyProgram {
2     TASK ReadFileAsync(name, callback) {
3         ReadFileSync(name);
4         Call(callback);
5     }
6     CALLBACK FinishOpeningFile() {
7         LoadFile(file);
8         RedrawScreen();
9     }
10    OnOpenFile() {
11        FILE file;
12        char szName[BUFSIZE]
13        InitFileName(szName);
14        EnqueueTask(ReadFileAsync(szName, FinishOpeningFile));
15    }
16    OnPaint();
17 }
```

Stack-Ripping

```
1 PROGRAM MyProgram {
2     TASK ReadFileAsync(name, callback) {
3         ReadFileSync(name);
4         Call(callback);
5     }
6     CALLBACK FinishOpeningFile() {
7         LoadFile(file);
8         RedrawScreen();
9     }
10    OnOpenFile() {
11        FILE file;
12        char szName[BUFSIZE];
13        InitFileName(szName);
14        EnqueueTask(ReadFileAsync(szName, FinishOpeningFile));
15    }
16    OnPaint();
17 }
```


Stack-Ripping

```
1 PROGRAM MyProgram {
2     TASK ReadFileAsync(name, callback) {
3         ReadFileSync(name);
4         Call(callback);
5     }
6     CALLBACK FinishOpeningFile() {
7         LoadFile(file);
8         RedrawScreen();
9     }
10    OnOpenFile() {
11        FILE file;
12        char szName[BUFSIZE];
13        InitFileName(szName);
14        EnqueueTask(ReadFileAsync(szName, FinishOpeningFile));
15    }
16    OnPaint();
17 }
```

Stack-Ripping

```
1 PROGRAM MyProgram {
2     TASK ReadFileAsync(name, callback) {
3         ReadFileSync(name);
4         Call(callback);
5     }
6     CALLBACK FinishOpeningFile() {
7         LoadFile(file);
8         RedrawScreen();
9     }
10    OnOpenFile() {
11        FILE file;
12        char szName[BUFSIZE];
13        InitFileName(szName);
14        EnqueueTask(ReadFileAsync(szName, FinishOpeningFile));
15    }
16    OnPaint();
17 }
```

Stack-based state out-of-scope!
Requests must carry state

Threads vs Events

- Thread Pros

- Event Pros

- Thread Cons

- Event Cons

Threads vs Events

- Thread Pros

- Overlap I/O and computation
 - While looking sequential
- Intermediate state on stack
- Control flow naturally expressed

- Thread Cons

- Synchronization required
- Overflowable stack
- Stack memory pressure

- Event Pros

- Easier to create well-conditioned system
- Easier to express dynamic change in level of parallelism

- Event Cons

- Difficult to program
- Control flow between callbacks obscure
- When to deallocate memory
- Incomplete language/tool/debugger support
- Difficult to exploit concurrent hardware

Threads vs Events

- Thread Pros

- Overlap I/O and computation
 - While looking sequential
- Intermediate state on stack
- Control flow naturally

- Thread Cons

- Synchronization required
- Overflowable stack
- Stack memory pressure

- Event Pros

- Easier to create well-conditioned system
- Easier to express dynamic change in level of parallelism

- Event Cons

- Difficult to program
- Control flow between callbacks obscure
- When to deallocate memory
- Incomplete language/tool/debugger support
- Difficult to exploit concurrent hardware

Language-level
Futures: the
sweet spot?

Threads vs Events



- Thread Pros

- Overlap I/O and computation
 - While looking sequential
- Intermediate state on stack
- Control flow naturally

- Thread Cons

- Synchronization required
- Overflowable stack
- Stack memory pressure

- Event Pros

- Easier to create well-conditioned system
- Easier to express dynamic change in level of parallelism

- Event Cons

- Difficult to program
- Control flow between callbacks obscure
- When to deallocate memory
- Incomplete language/tool/debugger support
- Difficult to exploit concurrent hardware

Language-level
Futures: the
sweet spot?

Futures & Promises

Futures & Promises

- *Values that will eventually become available*

Futures & Promises

- Values *that will eventually become available*
- Time-dependent states:
 - **Completed/determined**
 - Computation complete, value concrete
 - **Incomplete/undetermined**
 - Computation not complete yet

Futures & Promises

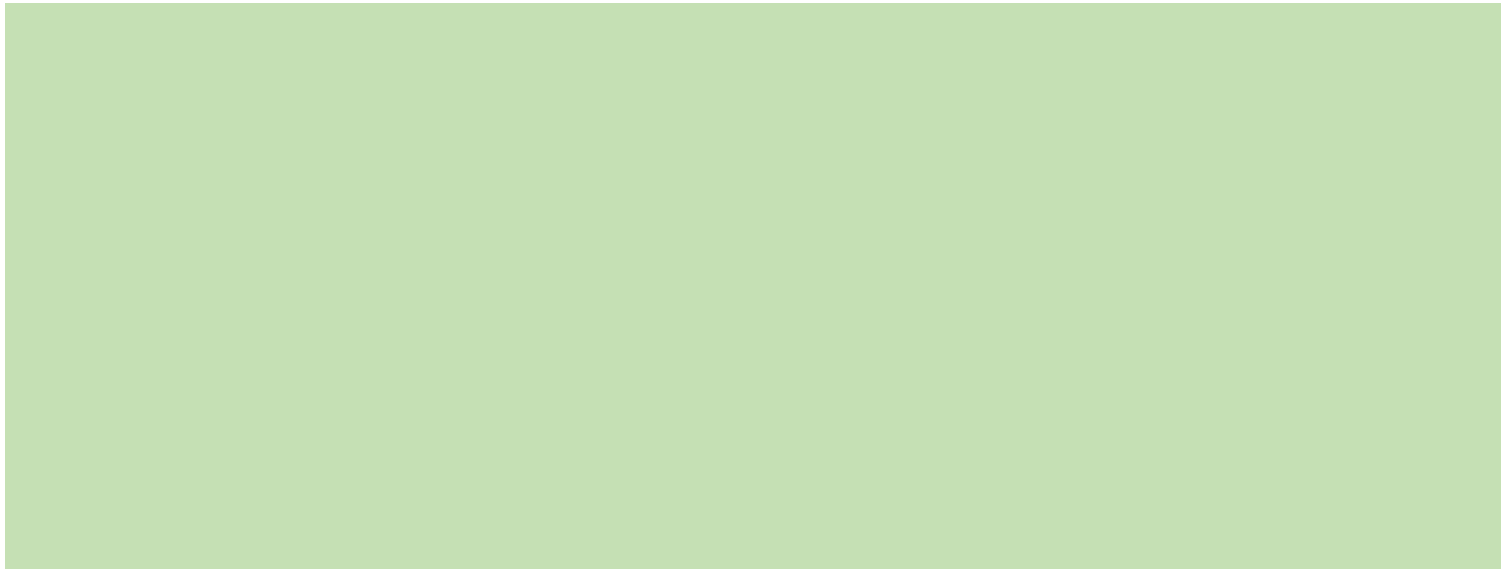
- *Values that will eventually become available*
- Time-dependent states:
 - **Completed/determined**
 - Computation complete, value concrete
 - **Incomplete/undetermined**
 - Computation not complete yet
- Construct (future X)
 - immediately returns value
 - concurrently executes X

Java Example

```
1 static void runAsyncExample() {
2     CompletableFuture cf = CompletableFuture.runAsync(() -> {
3         assertTrue(Thread.currentThread().isDaemon());
4         randomSleep();
5     });
6     assertFalse(cf.isDone());
7     sleepEnough();
8     assertTrue(cf.isDone());
9 }
```

Java Example

```
1 static void runAsyncExample() {  
2     CompletableFuture cf = CompletableFuture.runAsync(() -> {  
3         assertTrue(Thread.currentThread().isDaemon());  
4         randomSleep();  
5     });  
6     assertFalse(cf.isDone());  
7     sleepEnough();  
8     assertTrue(cf.isDone());  
9 }
```



Java Example

```
1 static void runAsyncExample() {
2     CompletableFuture cf = CompletableFuture.runAsync(() -> {
3         assertTrue(Thread.currentThread().isDaemon());
4         randomSleep();
5     });
6     assertFalse(cf.isDone());
7     sleepEnough();
8     assertTrue(cf.isDone());
9 }
```

- CompletableFuture is a container for Future object type

Java Example

```
1 static void runAsyncExample() {
2     CompletableFuture cf = CompletableFuture.runAsync(() -> {
3         assertTrue(Thread.currentThread().isDaemon());
4         randomSleep();
5     });
6     assertFalse(cf.isDone());
7     sleepEnough();
8     assertTrue(cf.isDone());
9 }
```

- CompletableFuture is a container for Future object type
- cf is an instance

Java Example

```
1 static void runAsyncExample() {
2     CompletableFuture cf = CompletableFuture.runAsync(() -> {
3         assertTrue(Thread.currentThread().isDaemon());
4         randomSleep();
5     });
6     assertFalse(cf.isDone());
7     sleepEnough();
8     assertTrue(cf.isDone());
9 }
```

- CompletableFuture is a container for Future object type
- cf is an instance
- runAsync() accepts
 - Lambda expression
 - Anonymous function
 - Functor

Java Example

```
1 static void runAsyncExample() {
2     CompletableFuture cf = CompletableFuture.runAsync(() -> {
3         assertTrue(Thread.currentThread().isDaemon());
4         randomSleep();
5     });
6     assertFalse(cf.isDone());
7     sleepEnough();
8     assertTrue(cf.isDone());
9 }
```

- CompletableFuture is a container for Future object type
- cf is an instance
- runAsync() accepts
 - Lambda expression
 - Anonymous function
 - Functor
- runAsync() immediately returns a waitable object (cf)

Java Example

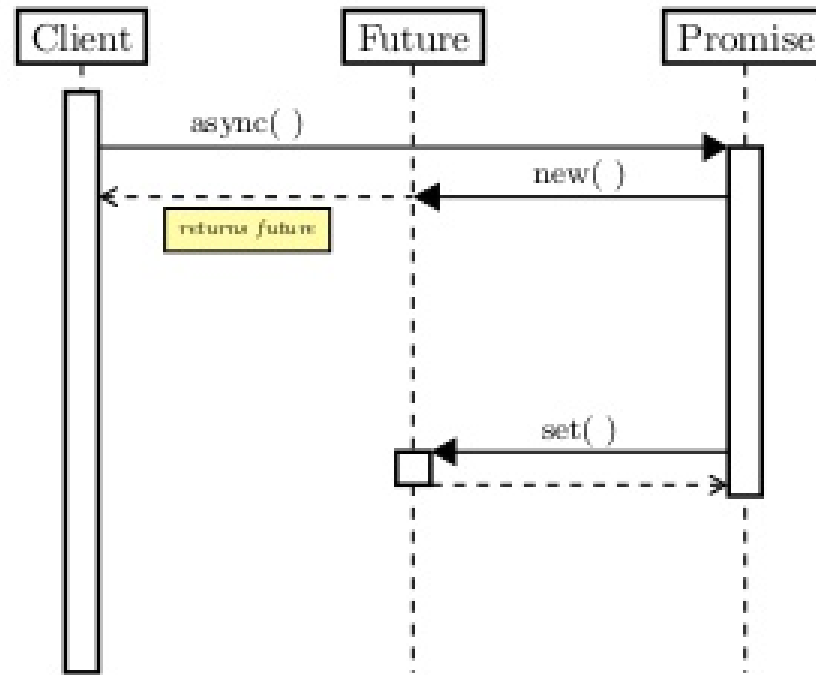
```
1 static void runAsyncExample() {
2     CompletableFuture cf = CompletableFuture.runAsync(() -> {
3         assertTrue(Thread.currentThread().isDaemon());
4         randomSleep();
5     });
6     assertFalse(cf.isDone());
7     sleepEnough();
8     assertTrue(cf.isDone());
9 }
```

- CompletableFuture is a container for Future object type
- cf is an instance
- runAsync() accepts
 - Lambda expression
 - Anonymous function
 - Functor
- runAsync() immediately returns a waitable object (cf)
- Where (on what thread) does the lambda expression run?

Futures and Promises:

Why two kinds of objects?

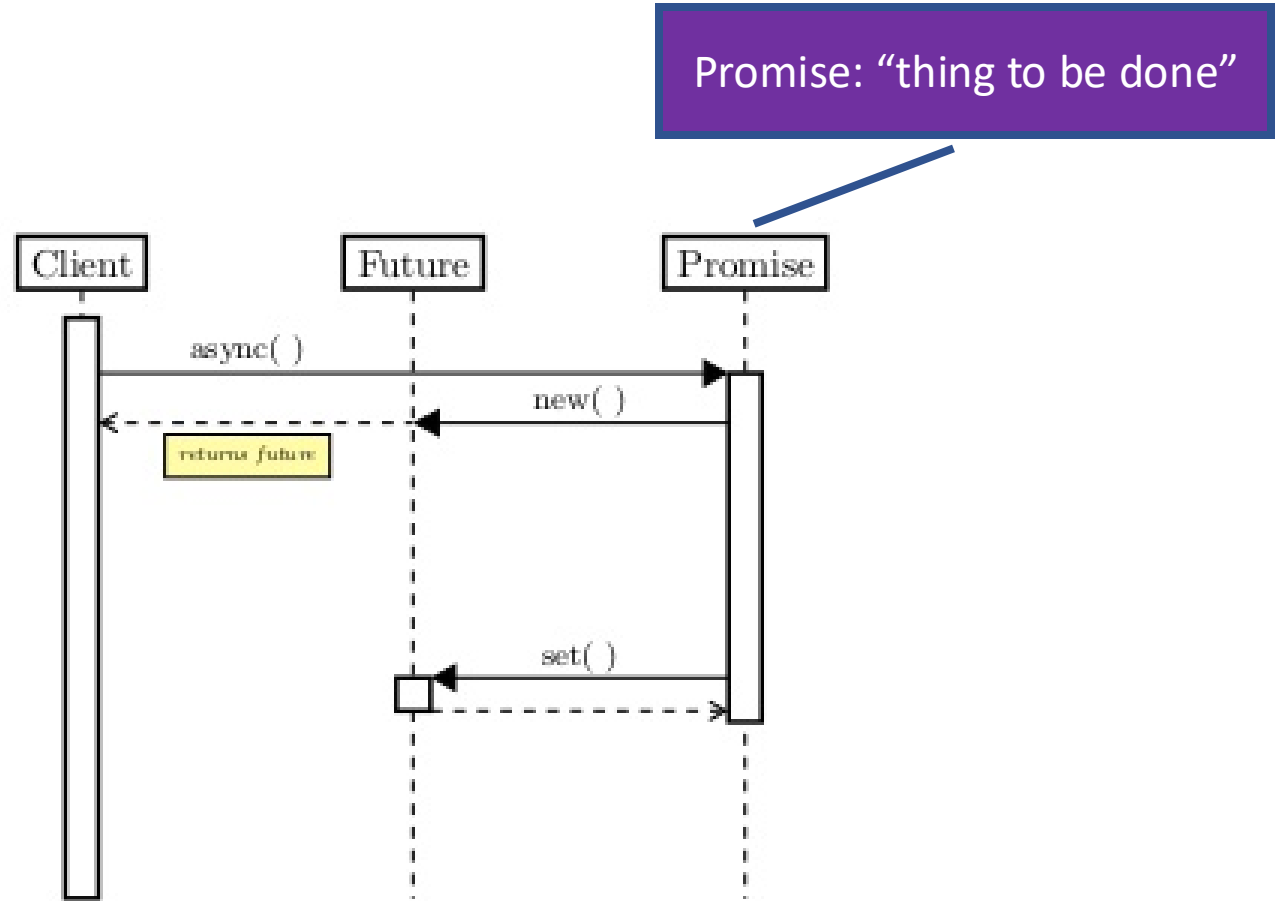
```
future<int> f1 = async(foo1);  
...  
int result = f1.get();
```



Futures and Promises:

Why two kinds of objects?

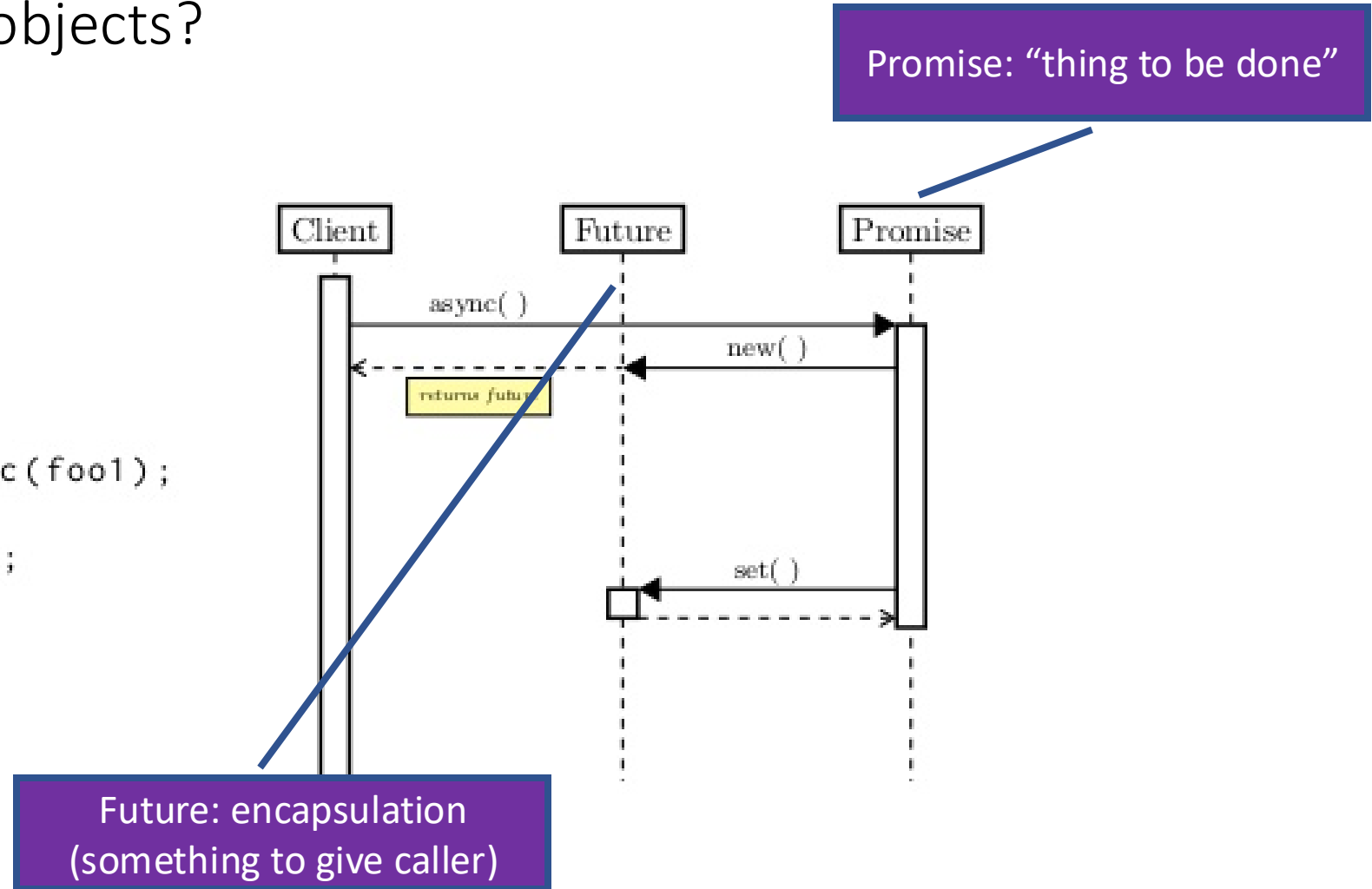
```
future<int> f1 = async(foo1);  
...  
int result = f1.get();
```



Futures and Promises:

Why two kinds of objects?

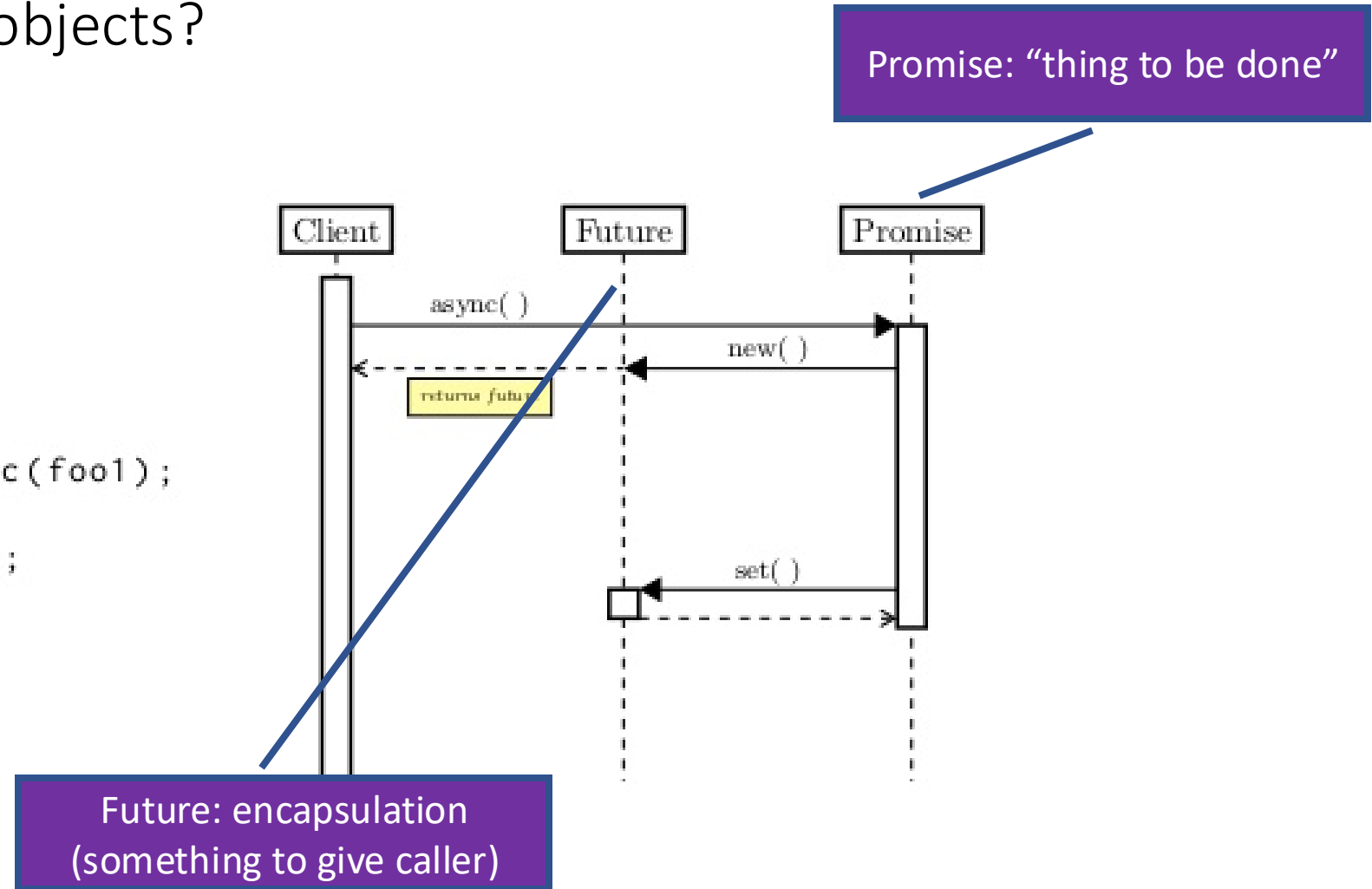
```
future<int> f1 = async(foo1);  
...  
int result = f1.get();
```



Futures and Promises:

Why two kinds of objects?

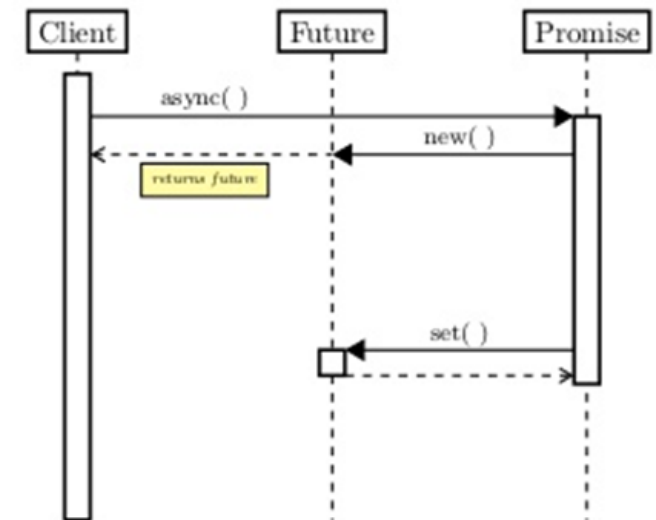
```
future<int> f1 = async(foo1);  
...  
int result = f1.get();
```



Promise to do something in the future

Futures vs Promises

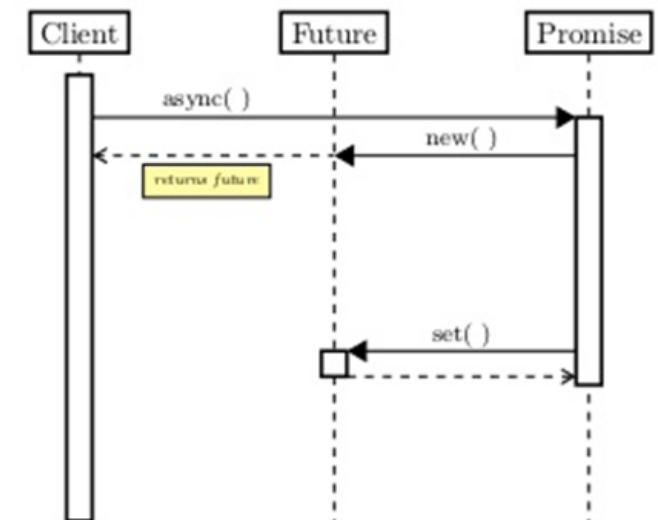
- **Future:** read-only reference to uncompleted value
- **Promise:** single-assignment variable that the future refers to
- Promises *complete* the future with:
 - Result with success/failure
 - Exception



Futures vs Promises

- **Future:** read-only reference to uncompleted value
- **Promise:** single-assignment variable that the future refers to
- Promises *complete* the future with:
 - Result with success/failure
 - Exception

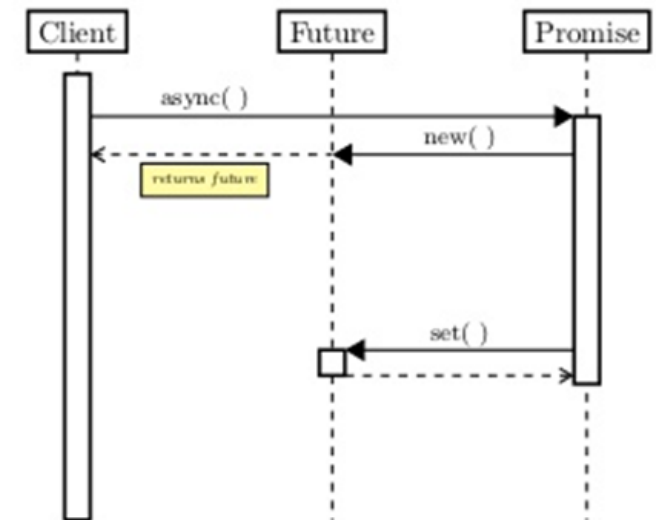
Language	Promise	Future
Algol	Thunk	Address of async result
Java	Future<T>	CompletableFuture<T>
C#/.NET	TaskCompletionSource<T>	Task<T>
JavaScript	Deferred	Promise
C++	std::promise	std::future



Futures vs Promises

- **Future:** read-only reference to uncompleted value
- **Promise:** single-assignment variable that the future refers to
- Promises *complete* the future with:
 - Result with success/failure
 - Exception

Language	Promise	Future
Algol	Thunk	Address of async result
Java	Future<T>	CompletableFuture<T>
C#/.NET	TaskCompletionSource<T>	Task<T>
JavaScript	Deferred	Promise
C++	std::promise	std::future

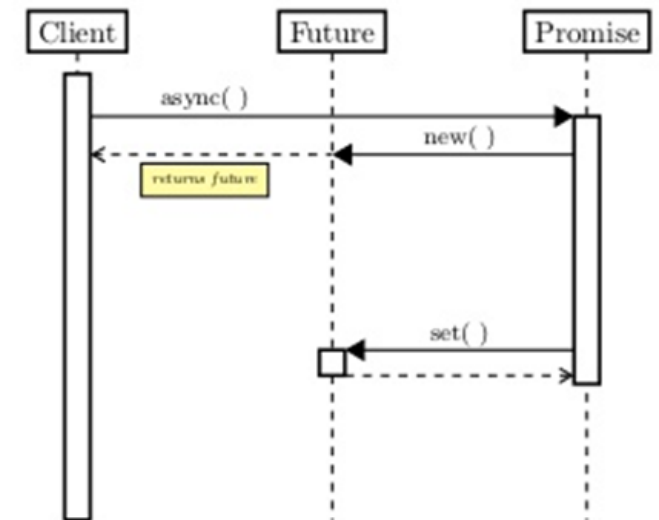


Futures vs Promises

Mnemonic:
Promise to *do* something
Make a promise *for* the future

- **Future:** read-only reference to uncompleted value
- **Promise:** single-assignment variable that the future refers to
- Promises *complete* the future with:
 - Result with success/failure
 - Exception

Language	Promise	Future
Algol	Thunk	Address of async result
Java	Future<T>	CompletableFuture<T>
C#/.NET	TaskCompletionSource<T>	Task<T>
JavaScript	Deferred	Promise
C++	std::promise	std::future



Go to Go!

- We'll come back to consistency after

```

9 static const string success = "success!";
10 struct X {
11     future<future<string>> open(std::string name, std::fstream* fs) {
12         fs->open(name, std::fstream::in);
13         return async(&X::read, this, fs);
14     }
15     future<string> read(std::fstream* fs) {
16         char ch;
17         while(!fs->eof()) {
18             *fs >> ch; cout << ch;
19             // build in-memory data structure
20         }
21         return async(&X::redraw, this);
22     }
23     string redraw() {
24         // redraw
25         return success;
26     }
27     static void handleOpenMenu() {
28         struct X x;
29         std::fstream fs;
30         std::string filename("test.txt");
31         auto openFuture = async(&X::open, &x, filename, &fs);
32         auto result = openFuture.get().get();
33         cout << "Result is: " << result.get() << endl;
34     }
35 };

```

File IO Events Revisited

Async File IO Revisited

```
8  std::fstream& OpenFile(std::string name, std::fstream& fs) { 26  void OnOpenFile() {
9      fs.open(name, std::fstream::in); 27      std::fstream fs;
10     return fs; 28     std::string filename;
11 } 29     std::packaged_task<std::fstream& ()> openTask(std::bind(OpenFile, filename, fs));
12 30     std::packaged_task<std::fstream& ()> readTask(std::bind(ReadFile, fs));
13  std::fstream& ReadFile(std::fstream& fs) { 31     std::packaged_task<void()> redrawTask(std::bind(RedrawScreen));
14     char ch; 32     std::future<std::fstream&> openFuture = openTask.get_future();
15     while(!fs.eof()) { 33     std::future<std::fstream&> readFuture = openTask.get_future();
16         fs >> ch; 34     std::future<std::fstream&> redrawFuture = openTask.get_future();
17         std::cout << ch; 35     std::thread openThread(std::move(openTask));
18     } 36     openFuture.wait();
19     return fs; 37     std::thread readThread(std::move(readTask));
20 } 38     readFuture.wait();
21 39     std::thread redrawThread(std::move(redrawTask));
22 void RedrawScreen() { 40     redrawFuture.wait();
23     // draw the screen 41     openThread.join();
24 } 42     readThread.join();
43     redrawThread.join();
44 }
```

Thread Pool Implementation

```
///-----  
/// <summary> Starts the threads. </summary>  
///  
/// <remarks> crossbac, 8/22/2013. </remarks>  
///  
/// <param name="uiThreads"> The threads. </param>  
/// <param name="bWaitAllThreadsAlive"> The wait all threads alive. </param>  
///-----  
  
void  
ThreadPool::StartThreads(  
    __in UINT uiThreads,  
    __in BOOL bWaitAllThreadsAlive  
)  
{  
    Lock();  
    if(uiThreads != 0 && m_vhThreadDescs.size() < m_uiTargetSize)  
        ResetEvent(m_hAllThreadsAlive);  
    while(m_vhThreadDescs.size() < m_uiTargetSize) {  
        for(UINT i=0; i<uiThreads; i++) {  
            THREADDESC* pDesc = new THREADDESC(this);  
            HANDLE * phThread = &pDesc->hThread;  
            *phThread = CreateThread(NULL, 0, _ThreadProc, pDesc, 0, NULL);  
            m_vhAvailable.push_back(*phThread);  
            m_vhThreadDescs[*phThread] = pDesc;  
        }  
    }  
    m_uiThreads = (UINT)m_vhThreadDescs.size();  
    Unlock();  
    if(bWaitAllThreadsAlive)  
        WaitThreadsAlive();  
}
```

Thread Pool Implementation

```
///-----  
/// <summary> Starts the threads. </summary>  
///  
/// <remarks> crossbac, 8/22/2013. </remarks>  
///  
/// <param name="uiThreads"> The threads. </param>  
/// <param name="bWaitAllThreadsAlive"> The wait all threads alive. </param>  
///-----  
  
void  
ThreadPool::StartThreads(  
    __in UINT uiThreads,  
    __in BOOL bWaitAllThreadsAlive  
)  
{  
    Lock();  
    if(uiThreads != 0 && m_vhThreadDescs.size() < m_uiTargetSize)  
        ResetEvent(m_hAllThreadsAlive);  
    while(m_vhThreadDescs.size() < m_uiTargetSize) {  
        for(UINT i=0; i<uiThreads; i++) {  
            THREADDESC* pDesc = new THREADDESC(this);  
            HANDLE * phThread = &pDesc->hThread;  
            *phThread = CreateThread(NULL, 0, _ThreadProc, pDesc, 0, NULL);  
            m_vhAvailable.push_back(*phThread);  
            m_vhThreadDescs[*phThread] = pDesc;  
        }  
    }  
    m_uiThreads = (UINT)m_vhThreadDescs.size();  
    Unlock();  
    if(bWaitAllThreadsAlive)  
        WaitThreadsAlive();  
}
```

Cool project
idea: build a
thread pool!

Thread Pool Implementation

```
DWORD
ThreadPool::ThreadPoolProc(
    _In_ THREADDESC * pDesc
)
{
    HANDLE hThread = pDesc->hThread;
    HANDLE hStartEvent = pDesc->hStartEvent;
    HANDLE hRuntimeTerminate = PTask::Runtime::GetRuntimeTerminateEvent();
    HANDLE vEvents[] = { hStartEvent, hRuntimeTerminate };

    NotifyThreadAlive(hThread);
    while(!pDesc->bTerminate) {

        DWORD dwWait = WaitForMultipleObjects(dwEvents, vEvents, FALSE, INFINITE);
        pDesc->Lock();
        pDesc->bTerminate |= bTerminate;
        if(pDesc->bRoutineValid && !pDesc->bTerminate) {
            LPTHREAD_START_ROUTINE lpRoutine = pDesc->lpRoutine;
            LPVOID lpParameter = pDesc->lpParameter;
            pDesc->bActive = TRUE;
            pDesc->Unlock();
            dwResult = (*lpRoutine)(lpParameter);
            pDesc->Lock();
            pDesc->bActive = FALSE;
            pDesc->bRoutineValid = FALSE;
        }
        pDesc->Unlock();
        Lock();
        m_vhInFlight.erase(pDesc->hThread);
        if(!pDesc->bTerminate)
            m_vhAvailable.push_back(pDesc->hThread);
        Unlock();
    }
    NotifyThreadExit(hThread);
    return dwResult;
}
```

ThreadPool Implementation

```
///-----  
/// <summary> Starts a thread: if a previous call to RequestThread was made with  
///           the bStartThread parameter set to false, this API signals the thread  
///           to begin. Otherwise, the call has no effect (returns FALSE). </summary>  
///  
/// <remarks> crossbac, 8/29/2013. </remarks>  
///  
/// <param name="hThread"> The thread. </param>  
///  
/// <returns> true if it succeeds, false if it fails. </returns>  
///-----
```

```
BOOL  
ThreadPool::SignalThread(  
    _In HANDLE hThread  
)  
{  
    Lock();  
    BOOL bResult = FALSE;  
    std::set<HANDLE>::iterator si = m_vhWaitingStartSignal.find(hThread);  
    if(si != m_vhWaitingStartSignal.end()) {  
        m_vhWaitingStartSignal.erase(hThread);  
        THREADDESC * pDesc = m_vhThreadDescs[hThread];  
        HANDLE hEvent = pDesc->hStartEvent;  
        SetEvent(hEvent);  
        bResult = TRUE;  
    }  
    Unlock();  
    return bResult;  
}
```


Futures in Context

Futures in Context

Futures:

Futures in Context

Futures:

- *abstraction* for concurrent work supported by
 - Compiler: abstractions are *language-level objects*
 - Runtime: scheduler, task queues, thread-pools are *transparent*

Futures in Context

Futures:

- *abstraction* for concurrent work supported by
 - Compiler: abstractions are *language-level objects*
 - Runtime: scheduler, task queues, thread-pools are *transparent*
- Programming remains **mostly** imperative
 - Threads of control peppered with asynchronous/concurrent tasks

Futures in Context

Futures:

- *abstraction* for concurrent work supported by
 - Compiler: abstractions are *language-level objects*
 - Runtime: scheduler, task queues, thread-pools are *transparent*
- Programming remains **mostly** imperative
 - Threads of control peppered with asynchronous/concurrent tasks

```
1 static void runAsyncExample() {
2     CompletableFuture cf = CompletableFuture.runAsync(() -> {
3         assertTrue(Thread.currentThread().isDaemon());
4         randomSleep();
5     });
6     assertFalse(cf.isDone());
7     sleepEnough();
8     assertTrue(cf.isDone());
9 }
```

Futures in Context

Futures:

- *abstraction* for concurrent work supported by
 - Compiler: abstractions are *language-level objects*
 - Runtime: scheduler, task queues, thread-pools are *transparent*
- Programming remains **mostly** imperative
 - Threads of control peppered with asynchronous/concurrent tasks

Compromise Model:

```
1 static void runAsyncExample() {
2     CompletableFuture cf = CompletableFuture.runAsync(() -> {
3         assertTrue(Thread.currentThread().isDaemon());
4         randomSleep();
5     });
6     assertFalse(cf.isDone());
7     sleepEnough();
8     assertTrue(cf.isDone());
9 }
```

Futures in Context

Futures:

- *abstraction* for concurrent work supported by
 - Compiler: abstractions are *language-level objects*
 - Runtime: scheduler, task queues, thread-pools are *transparent*
- Programming remains **mostly** imperative
 - Threads of control peppered with asynchronous/concurrent tasks

Compromise Model:

- Event-based programming

```
1 static void runAsyncExample() {
2     CompletableFuture cf = CompletableFuture.runAsync(() -> {
3         assertTrue(Thread.currentThread().isDaemon());
4         randomSleep();
5     });
6     assertFalse(cf.isDone());
7     sleepEnough();
8     assertTrue(cf.isDone());
9 }
```

Futures in Context

Futures:

- *abstraction* for concurrent work supported by
 - Compiler: abstractions are *language-level objects*
 - Runtime: scheduler, task queues, thread-pools are *transparent*
- Programming remains **mostly** imperative
 - Threads of control peppered with asynchronous/concurrent tasks

Compromise Model:

- Event-based programming
- Thread-based programming

```
1 static void runAsyncExample() {
2     CompletableFuture cf = CompletableFuture.runAsync(() -> {
3         assertTrue(Thread.currentThread().isDaemon());
4         randomSleep();
5     });
6     assertFalse(cf.isDone());
7     sleepEnough();
8     assertTrue(cf.isDone());
9 }
```


Futures in Context

Futures:

- *abstraction* for concurrent work supported by
 - Compiler: abstractions are *language-level objects*
 - Runtime: scheduler, task queues, thread-pools are *transparent*
- Programming remains **mostly** imperative
 - Threads of control peppered with asynchronous/concurrent tasks

Compromise Model:

- Event-based programming
- Thread-based programming

Currently: 2nd renaissance IMHO

```
1 static void runAsyncExample() {
2     CompletableFuture cf = CompletableFuture.runAsync(() -> {
3         assertTrue(Thread.currentThread().isDaemon());
4         randomSleep();
5     });
6     assertFalse(cf.isDone());
7     sleepEnough();
8     assertTrue(cf.isDone());
9 }
```

Memory Consistency

Memory Consistency

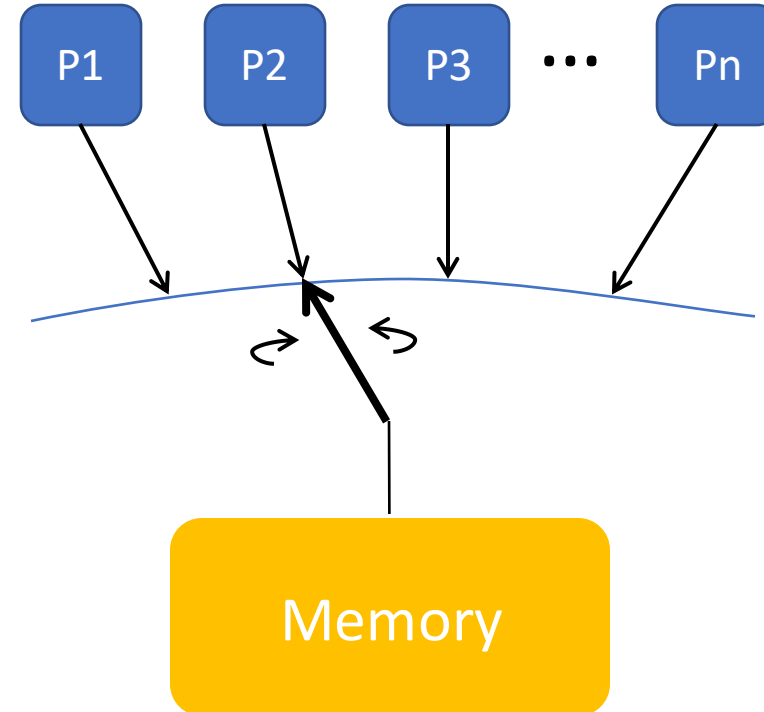
- Formal specification of memory semantics
 - Statement of how shared memory will behave with multiple CPUs
 - Ordering of reads and writes

Memory Consistency

- Formal specification of memory semantics
 - Statement of how shared memory will behave with multiple CPUs
 - Ordering of reads and writes
- Memory Consistency != Cache Coherence
 - Coherence: propagate updates to cached copies
 - Invalidate vs. Update
 - Coherence vs. Consistency?
 - **Coherence:** ordering of ops. at a single location
 - **Consistency:** ordering of ops. at multiple locations

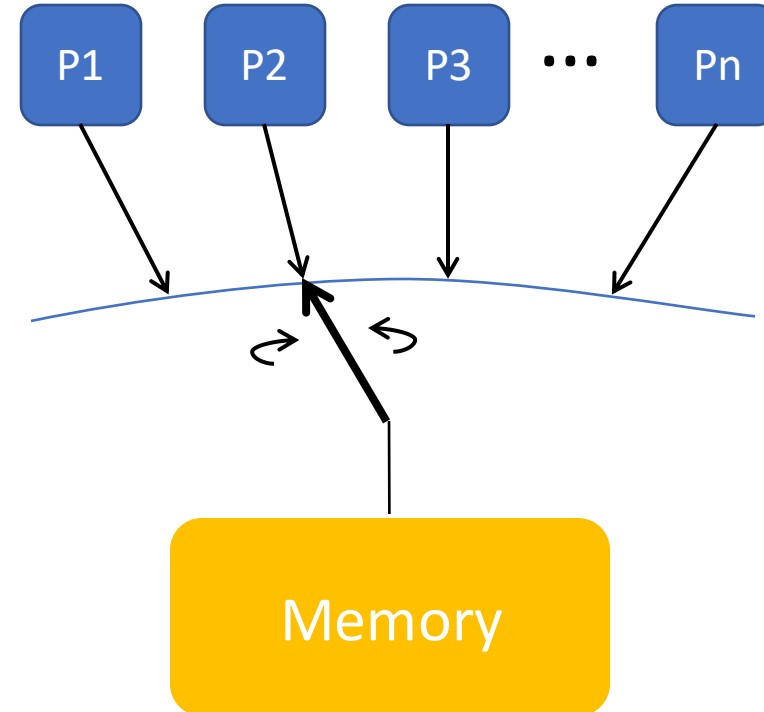
Sequential Consistency

- Result of *any* execution is same as if all operations execute on a uniprocessor
- Operations on each processor are *totally ordered* in the sequence and respect program order for each processor



Sequential Consistency

- Result of *any* execution is same as if all operations execute on a uniprocessor
- Operations on each processor are *totally ordered* in the sequence and respect program order for each processor



Trying to mimic Uniprocessor semantics:

- Memory operations occur:
 - One at a time
 - In program order
- Read returns value of last write

Sequential Consistency: Canonical Example

Initially, Flag1 = Flag2 = 0

P1

```
Flag1 = 1  
if (Flag2 == 0)  
    enter CS
```

P2

```
Flag2 = 1  
if (Flag1 == 0)  
    enter CS
```

Sequential Consistency: Canonical Example

Initially, Flag1 = Flag2 = 0

P1

```
Flag1 = 1  
if (Flag2 == 0)  
    enter CS
```

P2

```
Flag2 = 1  
if (Flag1 == 0)  
    enter CS
```

Can both P1 and P2 wind up in the critical section at the same time?

Do we need Sequential Consistency?

Initially, $A = B = 0$

P1

$A = 1$

P2

if ($A == 1$)
 $B = 1$

P3

if ($B == 1$)
 register1 = A

Do we need Sequential Consistency?

Initially, $A = B = 0$

P1

$A = 1$

P2

if ($A == 1$)
 $B = 1$

P3

if ($B == 1$)
register1 = A

Key issue:

- P2 and P3 may not see writes to A, B in the same order
- Implication: P3 can see $B == 1$, but $A == 0$ which is incorrect
- Wait! Why would this happen?

Do we need Sequential Consistency?

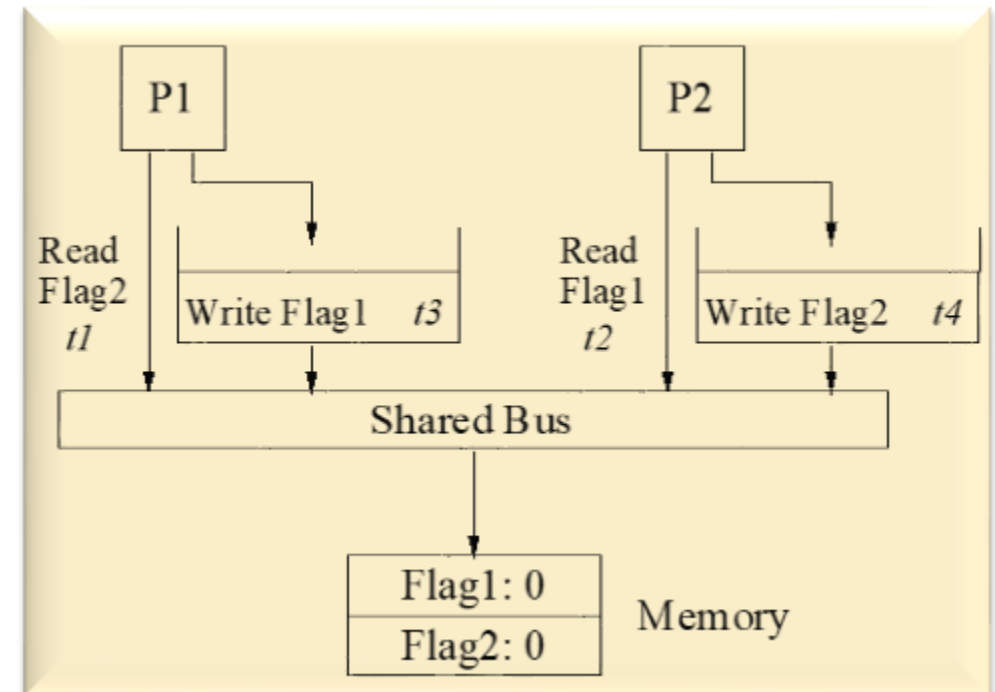
Initially, $A = B = 0$

P1

$A = 1$

P2

if ($A == 1$)
 $B = 1$



Key issue:

- P2 and P3 may not see writes to A, B in the same order
- Implication: P3 can see $B == 1$, but $A == 0$ which is incorrect
- Wait! Why would this happen?

Write Buffers

- P₀ write → queue op in write buffer, proceed
- P₀ read → look in write buffer,
- P_(x != 0) read → old value: write buffer hasn't drained

Requirements for Sequential Consistency

Requirements for Sequential Consistency

- *Program Order*
 - Processor's memory operations must complete in program order

Requirements for Sequential Consistency

- *Program Order*
 - Processor's memory operations must complete in program order
- *Write Atomicity*
 - Writes to the same location seen by all other CPUs
 - Subsequent reads must not return value of a write until propagated to all

Requirements for Sequential Consistency

- *Program Order*
 - Processor's memory operations must complete in program order
- *Write Atomicity*
 - Writes to the same location seen by all other CPUs
 - Subsequent reads must not return value of a write until propagated to all
- Write acknowledgements are necessary
 - Cache coherence provides these properties for a cache-only system

Requirements for Sequential Consistency

- *Program Order*
 - Processor's memory operations must complete in program order
- *Write Atomicity*
 - Writes to the same location seen by all other CPUs
 - Subsequent reads must not return value of a write until propagated to all
- Write acknowledgements are necessary
 - Cache coherence provides these properties for a cache-only system

Disadvantages:

- Difficult to implement!
 - Coherence to (e.g.) write buffers is hard
- Sacrifices many potential optimizations
 - Hardware (cache) and software (compiler)
 - Major performance hit

Relaxed Consistency Models

Relaxed Consistency Models

- **Program Order** relaxations *(different locations)*
 - $W \rightarrow R$; $W \rightarrow W$; $R \rightarrow R/W$

Relaxed Consistency Models

- **Program Order** relaxations *(different locations)*
 - $W \rightarrow R$; $W \rightarrow W$; $R \rightarrow R/W$
- **Write Atomicity** relaxations
 - Read returns another processor's Write early

Relaxed Consistency Models

- **Program Order** relaxations *(different locations)*
 - $W \rightarrow R$; $W \rightarrow W$; $R \rightarrow R/W$
- **Write Atomicity** relaxations
 - Read returns another processor's Write early
- Combined relaxations
 - Read your own Write *(okay for S.C.)*

Relaxed Consistency Models

- **Program Order** relaxations *(different locations)*
 - $W \rightarrow R$; $W \rightarrow W$; $R \rightarrow R/W$
- **Write Atomicity** relaxations
 - Read returns another processor's Write early
- Combined relaxations
 - Read your own Write *(okay for S.C.)*
- *Requirement: synchronization primitives for safety*
 - Fence, barrier instructions etc

Relaxed Consistency Models

- **Program Order** relaxations *(different locations)*
 - $W \rightarrow R$; $W \rightarrow W$; $R \rightarrow R/W$
- **Write Atomicity** relaxations
 - Read returns another processor's V
- Combined relaxations
 - Read your own Write *(okay for S.C)*
- *Requirement: synchronization pri*
 - Fence, barrier instructions etc

Relaxation	W → R Order	W → W Order	R → RW Order	Read Others' Write Early	Read Own Write Early	Safety net
SC [16]					✓	
IBM 370 [14]	✓					serialization instructions
TSO [20]	✓				✓	RMW
PC [13, 12]	✓			✓	✓	RMW
PSO [20]	✓	✓			✓	RMW, STBAR
WO [5]	✓	✓	✓		✓	synchronization
RCsc [13, 12]	✓	✓	✓		✓	release, acquire, nsync, RMW
RCpc [13, 12]	✓	✓	✓	✓	✓	release, acquire, nsync, RMW
Alpha [19]	✓	✓	✓		✓	MB, WMB
RMO [21]	✓	✓	✓		✓	various MEMBAR's
PowerPC [17, 4]	✓	✓	✓	✓	✓	SYNC

Questions?