Asynchronous Programming
Promises + Futures
Consistency

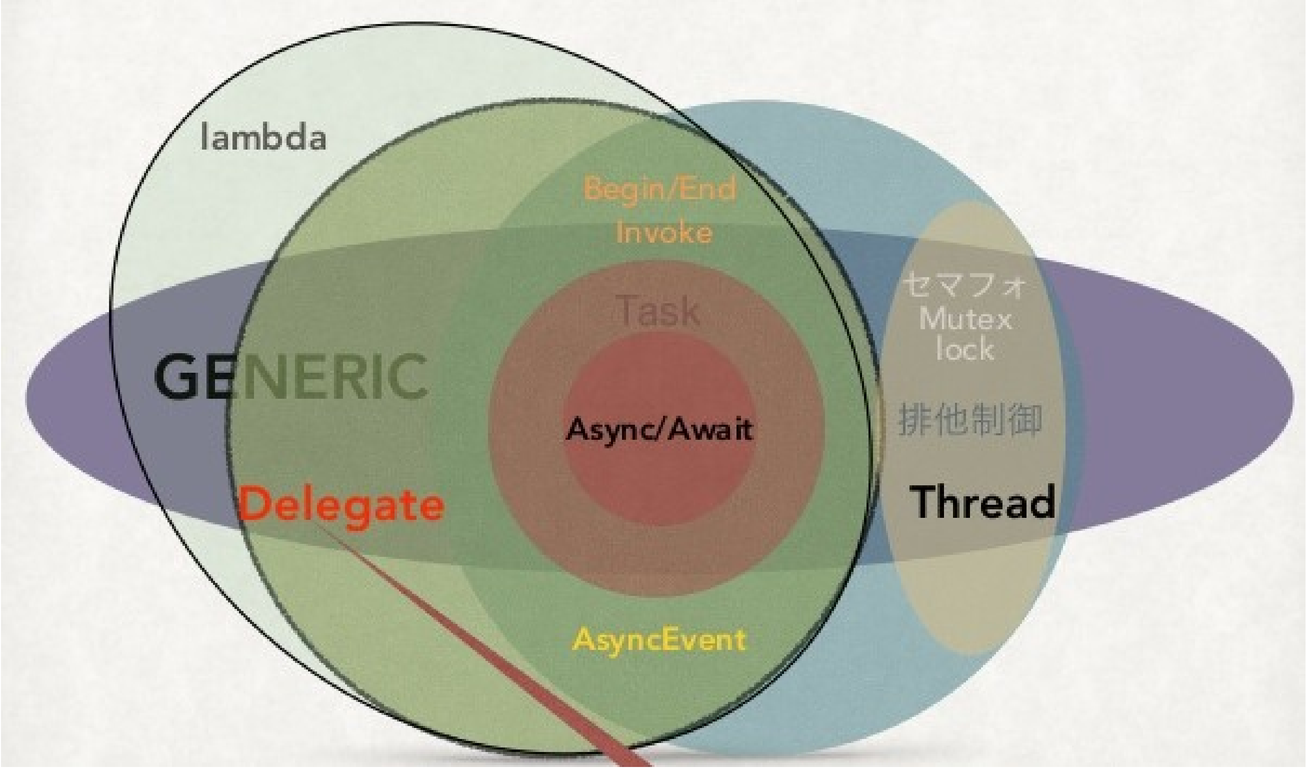
Chris Rossbach

Today

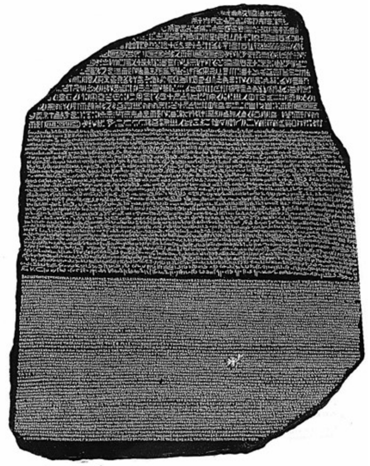
- Questions?
- Administrivia
- Material for the day
  - Events / Asynchronous programming
  - Promises & Futures
  - Bonus: memory consistency models

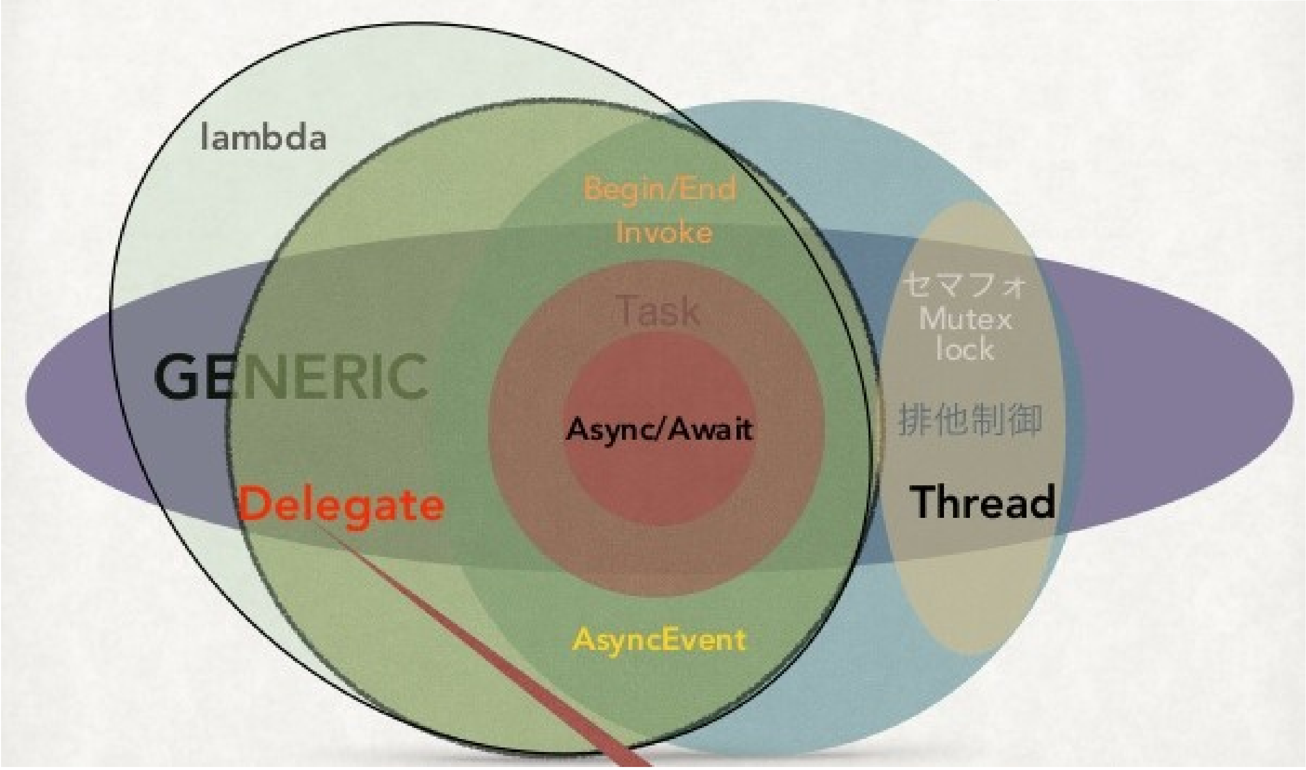
- Acknowledgements
  - Consistency slides borrow some materials from Kevin Boos. Thanks!

Asynchronous Programming
Events, Promises, and Futures



Asynchronous Programming
Events, Promises, and Futures





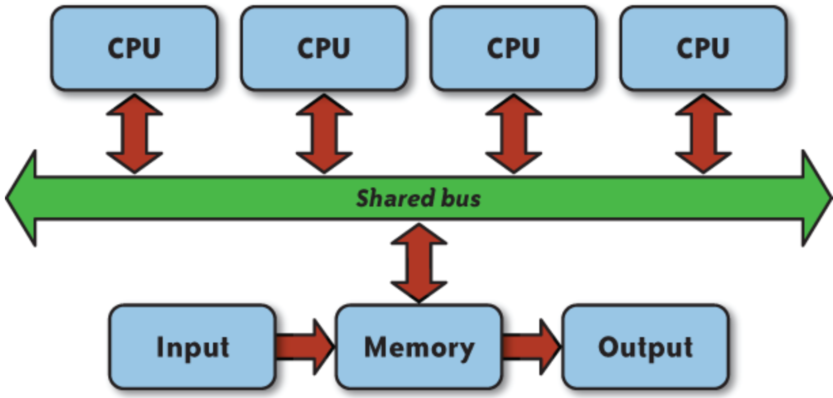
Programming Models for Concurrency

Programming Models for Concurrency

- Hardware execution model:

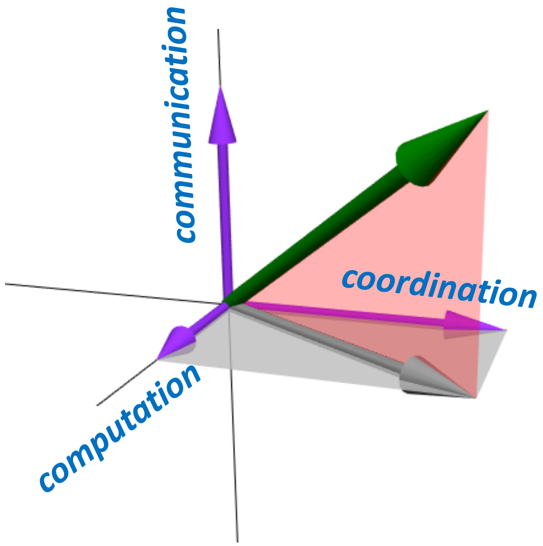
Programming Models for Concurrency

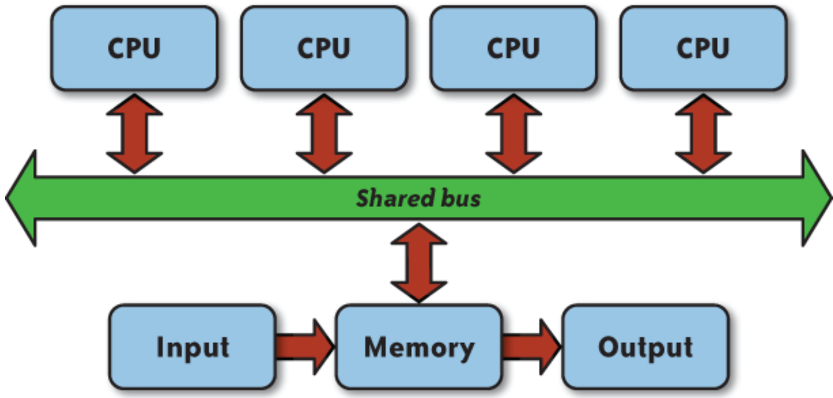
- Hardware execution model:
  - CPU(s) execute instructions sequentially



Programming Models for Concurrency

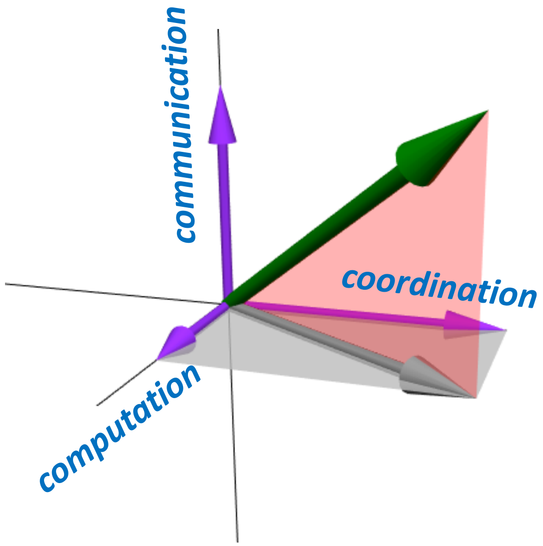
- Hardware execution model:
  - CPU(s) execute instructions sequentially
- Programming model dimensions:
  - How to specify computation
  - How to specify communication
  - How to specify coordination/control transfer

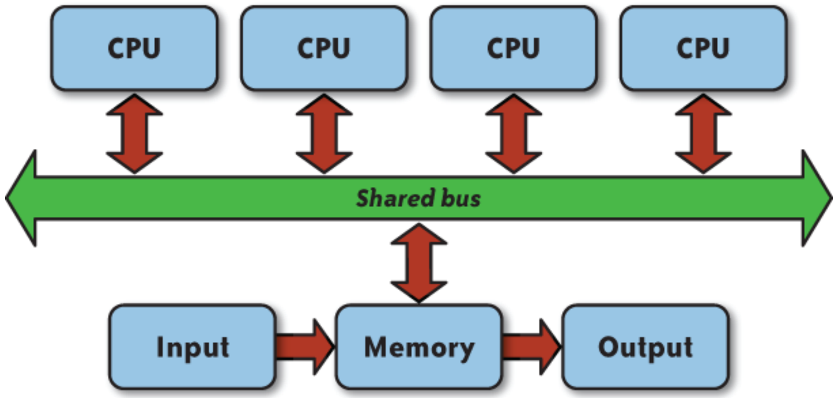




Programming Models for Concurrency

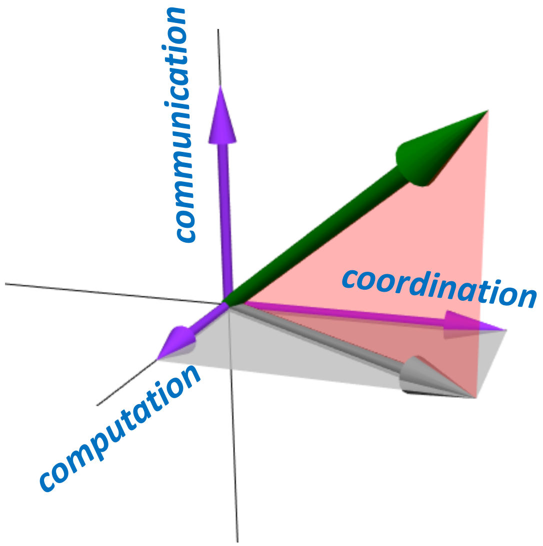
- Hardware execution model:
  - CPU(s) execute instructions sequentially
- Programming model dimensions:
  - How to specify computation
  - How to specify communication
  - How to specify coordination/control transfer
- Techniques/primitives
  - Message passing vs shared memory
  - Preemption vs Non-preemption

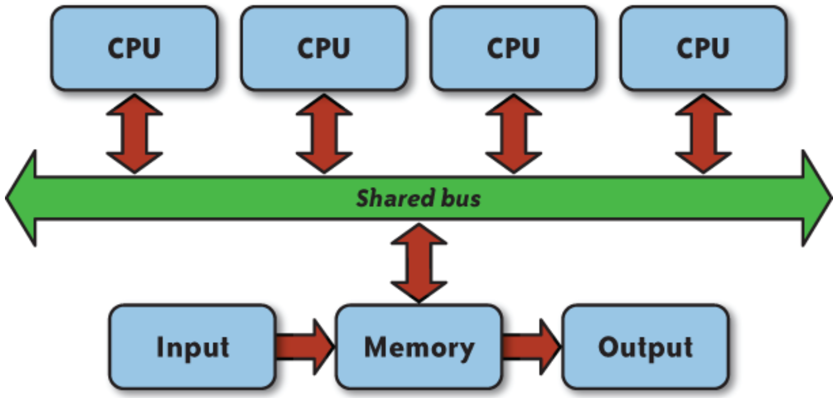




Programming Models for Concurrency

- 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

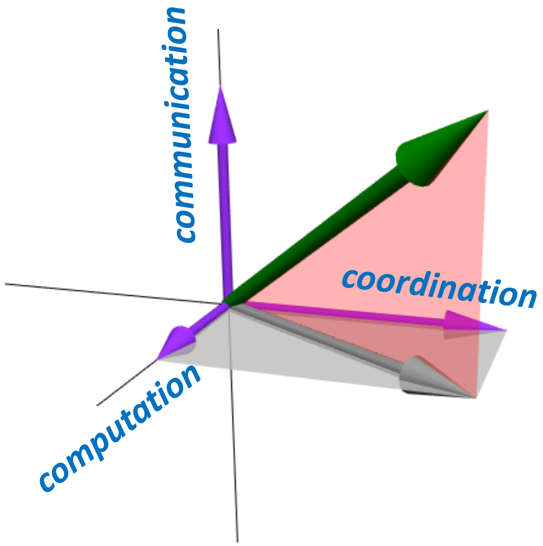


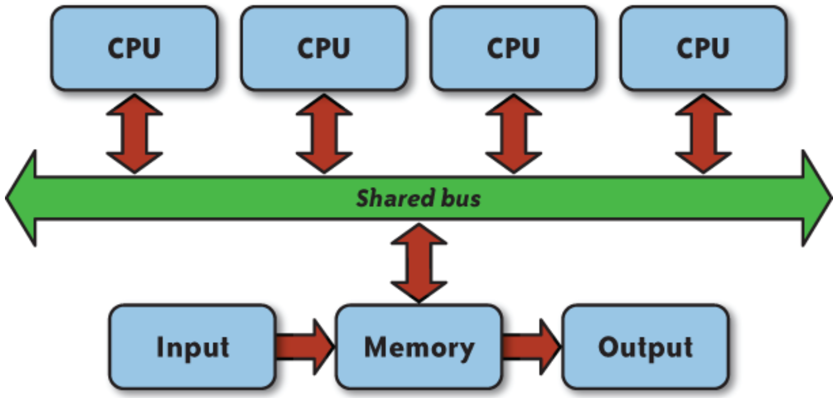


Programming Models for Concurrency

- 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





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

```
static void runAsyncExample() {
    CompletableFuture cf = CompletableFuture.runAsync(() -> {
        assertTrue(Thread.currentThread().isDaemon());
        randomSleep();
    });
    assertFalse(cf.isDone());
    sleepEnough();
    assertTrue(cf.isDone());
}
```

Java Example

```java
static void runAsyncExample() {
    CompletableFuture cf = CompletableFuture.runAsync(() -> {
        assertTrue(Thread.currentThread().isDaemon());
        randomSleep();
    });
    assertFalse(cf.isDone());
    sleepEnough();
    assertTrue(cf.isDone());
}
```

Java Example

```java
static void runAsyncExample() {
    CompletableFuture cf = CompletableFuture.runAsync(() -> {
        assertTrue(Thread.currentThread().isDaemon());
        randomSleep();
    });
    assertFalse(cf.isDone());
    sleepEnough();
    assertTrue(cf.isDone());
}
```

- CompletableFuture is a container for Future object type

Java Example

```
static void runAsyncExample() {
    CompletableFuture cf = CompletableFuture.runAsync(() -> {
        assertTrue(Thread.currentThread().isDaemon());
        randomSleep();
    });
    assertFalse(cf.isDone());
    sleepEnough();
    assertTrue(cf.isDone());
}
```

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

Java Example

```
static void runAsyncExample() {
    CompletableFuture cf = CompletableFuture.runAsync(() -> {
        assertTrue(Thread.currentThread().isDaemon());
        randomSleep();
    });
    assertFalse(cf.isDone());
    sleepEnough();
    assertTrue(cf.isDone());
}
```

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

Java Example

```java
static void runAsyncExample() {
    CompletableFuture cf = CompletableFuture.runAsync(() -> {
        assertTrue(Thread.currentThread().isDaemon());
        randomSleep();
    });
    assertFalse(cf.isDone());
    sleepEnough();
    assertTrue(cf.isDone());
}
```

- 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)

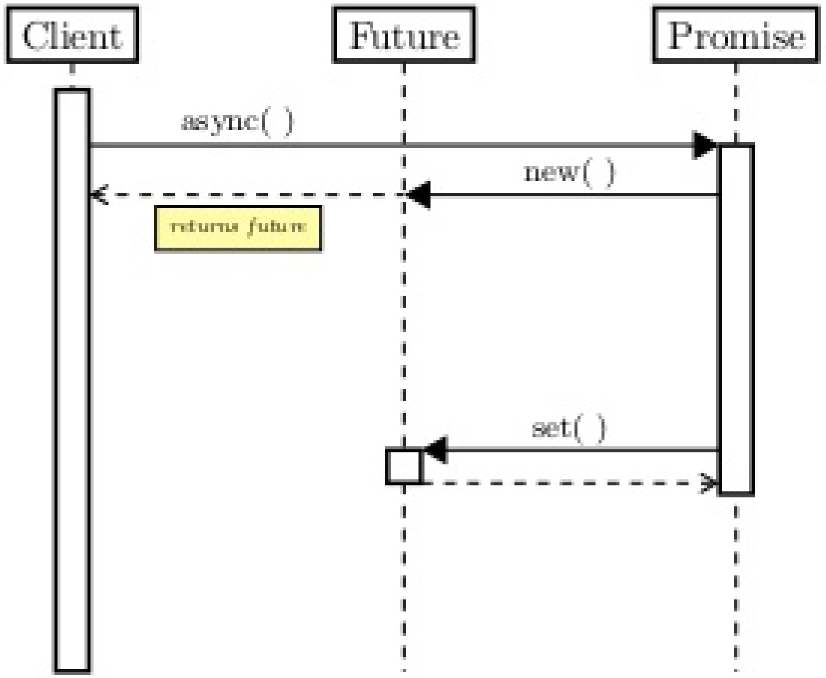
Java Example

```java
static void runAsyncExample() {
    CompletableFuture cf = CompletableFuture.runAsync(() -> {
        assertTrue(Thread.currentThread().isDaemon());
        randomSleep();
    });
    assertFalse(cf.isDone());
    sleepEnough();
    assertTrue(cf.isDone());
}
```

- 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();
```

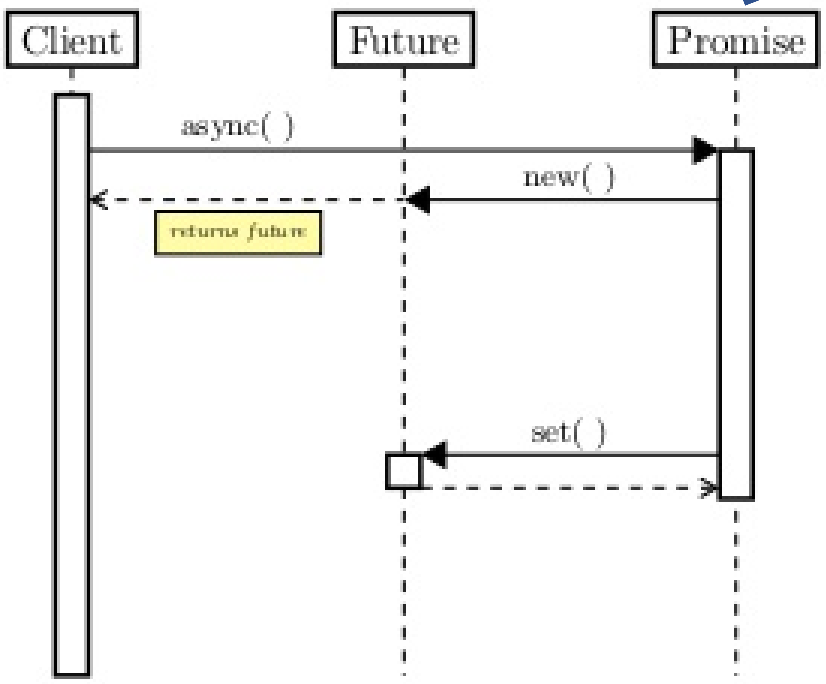


Futures and Promises:

Why two kinds of objects?

Promise: “thing to be done”

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

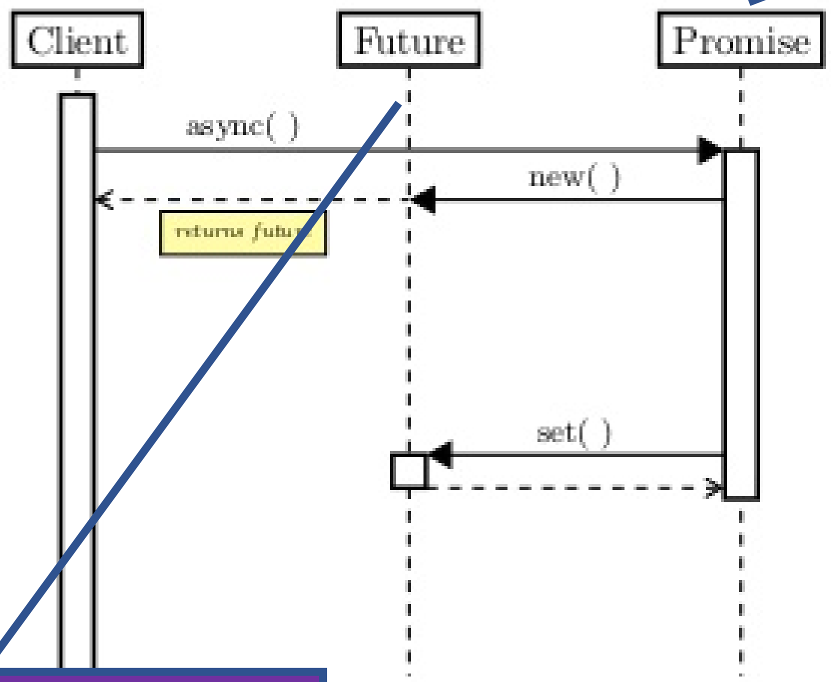


Futures and Promises:

Why two kinds of objects?

Promise: “thing to be done”

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



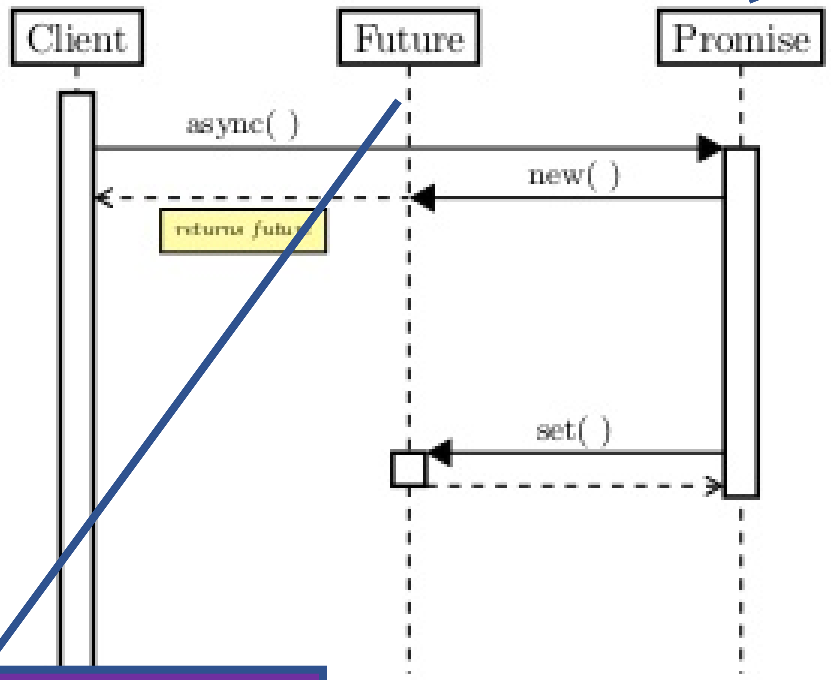
Future: encapsulation
(something to give caller)

Futures and Promises:

Why two kinds of objects?

Promise: “thing to be done”

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

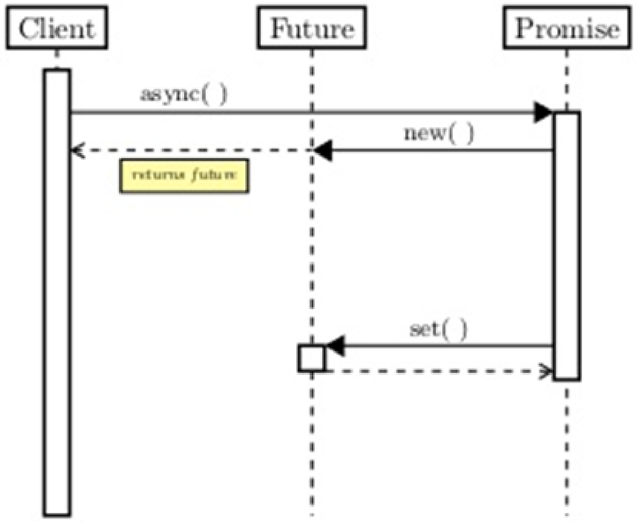


Future: encapsulation
(something to give caller)

Promise to do something in the future

Futures vs Promises

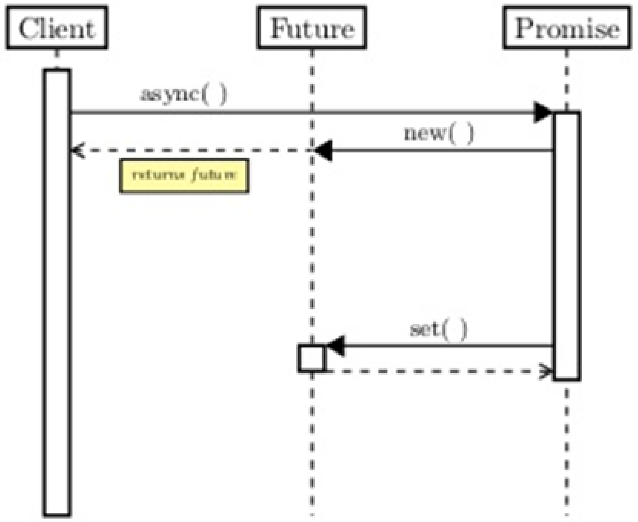
- **Future:** read-only reference to uncompleted value
- **Promise:** single-assignment variable that the future refers to
- Promises *complete* the future with:
  - Result with success/failure
  - Exception



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

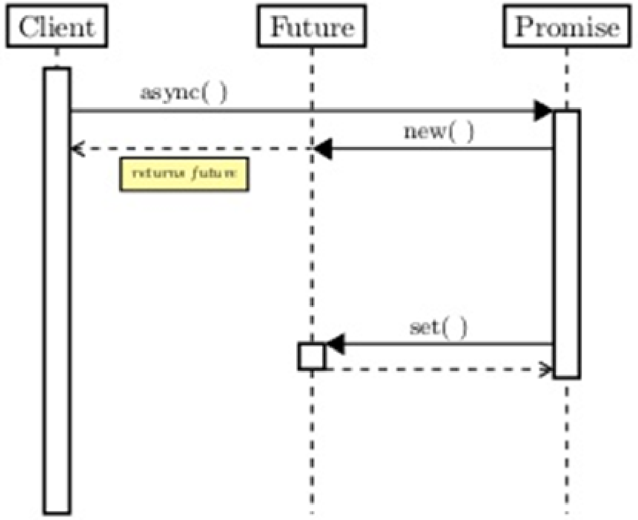
Language	Promise	Future
Algol	Thunk	Address of async result
Java	Future<T>	CompletableFuture<T>
C#/.NET	TaskCompletionSource<T>	Task<T>
JavaScript	Deferred	Promise
C++	std::promise	std::future



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

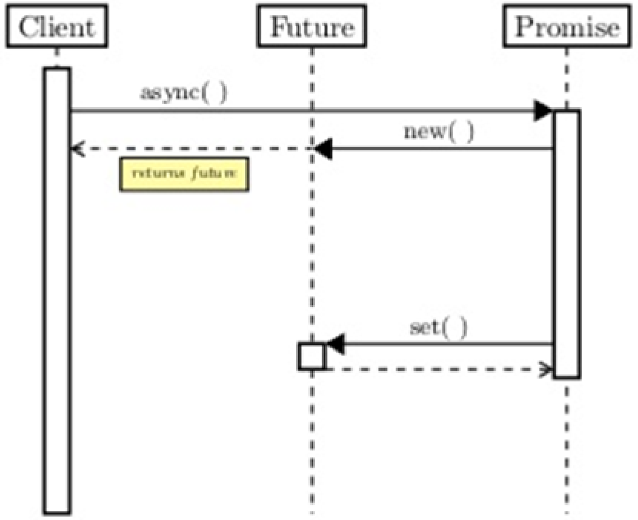


Futures vs Promises

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



GUI Programming Distilled

```
winmain(...) {
  while(true) {
      message = GetMessage();
      switch(message) {
      case WM_THIS: DoThis(); break;
      case WM_THAT: DoThat(); break;
      case WM_OTHERTHING: DoOtherThing(); break;
      case WM_DONE: return;
      }
  }
}
```

GUI Programming Distilled

```
winmain(...) {
   while(true) {
        message = GetMessage();
        switch(message) {
        case WM_THIS: DoThis(); break;
        case WM_THAT: DoThat(); break;
        case WM_OTHERTHING: DoOtherThing(); break;
        case WM_DONE: return;
        }
   }
}
```

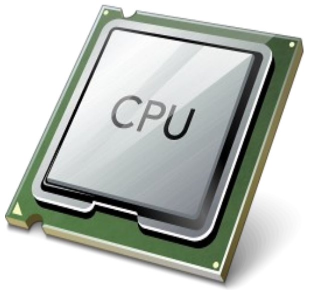
How can we parallelize this?

Parallel GUI Implementation 1

```
winmain(...) {
  while(true) {
      message = GetMessage();
      switch(message) {
      case WM_THIS: DoThis(); break;
      case WM_THAT: DoThat(); break;
      case WM_OTHERTHING: DoOtherThing(); break;
      case WM_DONE: return;
      }
  }
}
```

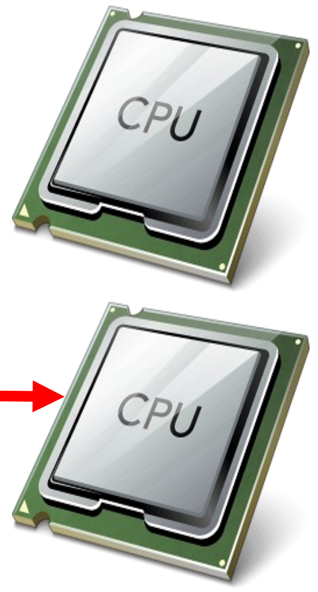
Parallel GUI Implementation 1

```
winmain(...) {
  while(true) {
      message = GetMessage();
      switch(message) {
      case WM_THIS: DoThis(); break;
      case WM_THAT: DoThat(); break;
      case WM_OTHERTHING: DoOtherThing(); break;
      case WM_DONE: return;
      }
  }
}
```



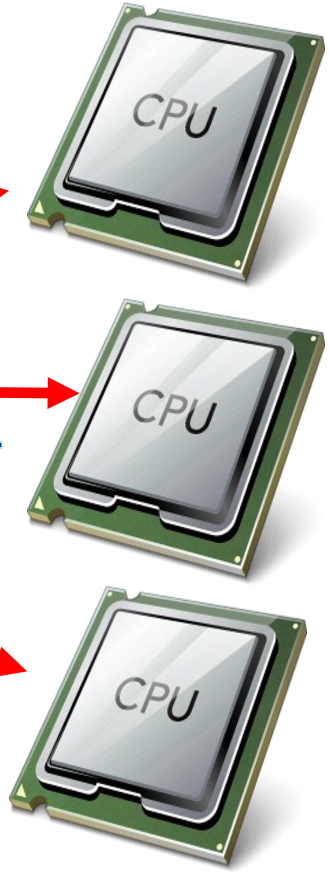
Parallel GUI Implementation 1

```
winmain(...) {
    while(true) {
        message = GetMessage();
        switch(message) {
        case WM_THIS: DoThis(); break;
        case WM_THAT: DoThat(); break;
        case WM_OTHERTHING: DoOtherThing(); break;
        case WM_DONE: return;
        }
    }
}
```



Parallel GUI Implementation 1

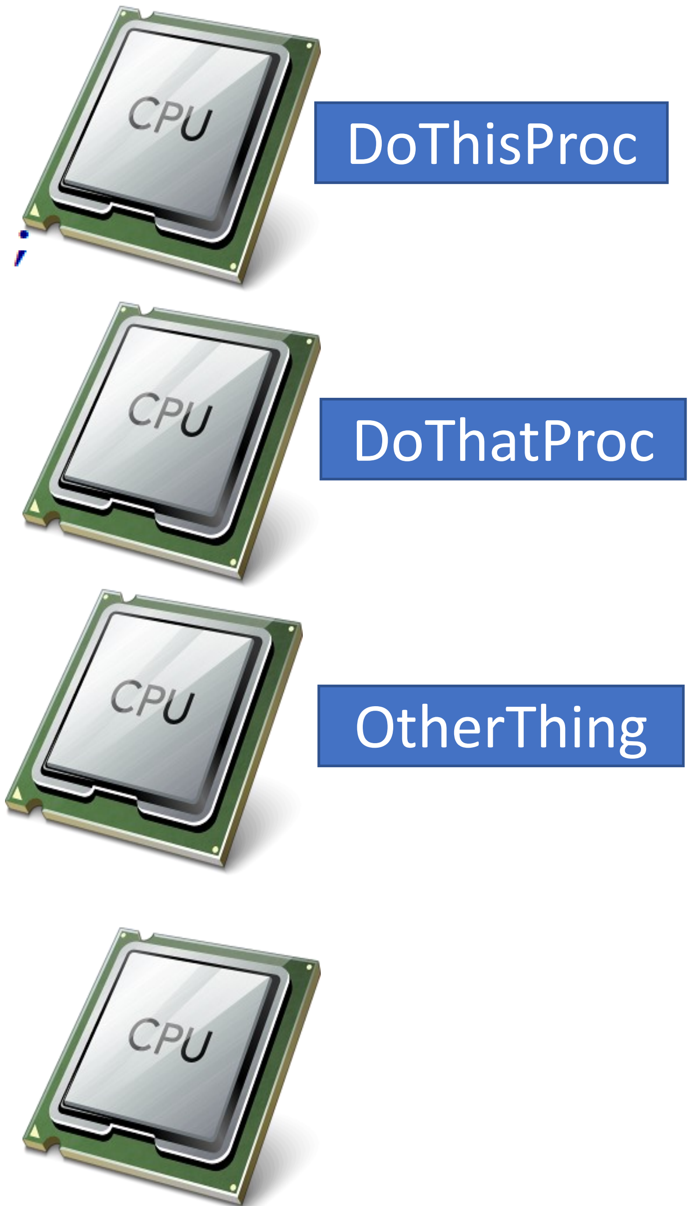
```
winmain(...) {
   while(true) {
         message = GetMessage();
         switch(message) {
         case WM_THIS: DoThis(); break;
         case WM_THAT: DoThat(); break;
         case WM_OTHERTHING: DoOtherThing(); break;
         case WM_DONE: return;
         }
   }
}
```



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();
    }
}
```

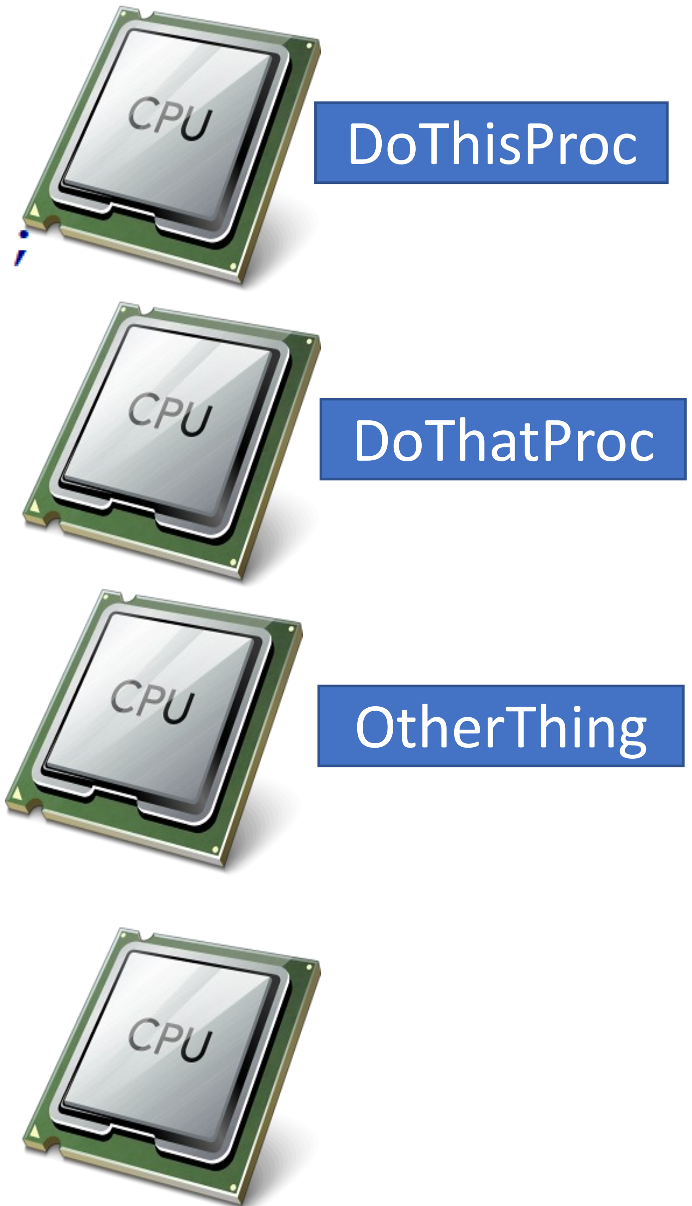


Parallel GUI Implementation 1

```
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]);
}

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

Pros/cons?



Parallel GUI Implementation 1

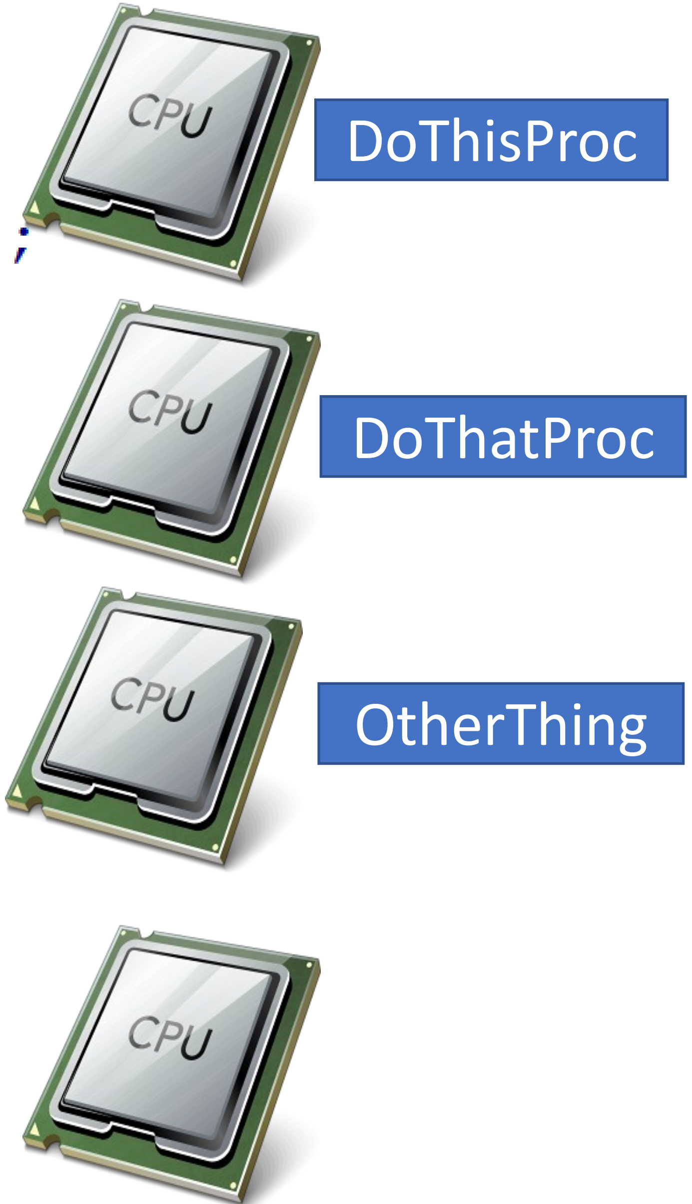
```
winmain() {
    pthread_create(&tids[i++], DoThisProc);
    pthread_create(&tids[i++], DoThatProc);
    pthread_create(&tids[i++], DoOtherThingProc);
    for(j=0; j<i; j++)
        pthread_join(&tids[j]);
}

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

Pros/cons?

Pros:
- Encapsulates parallel work

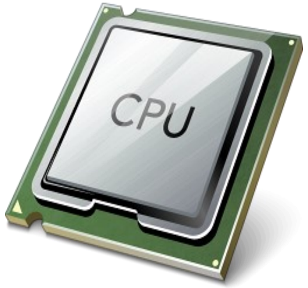
Cons:
- Obliterates original code structure
- How to assign handlers→CPUs?
- Load balance?!?
- Utilization



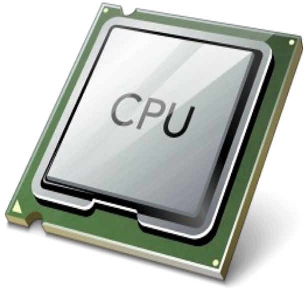
Parallel GUI Implementation 2

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

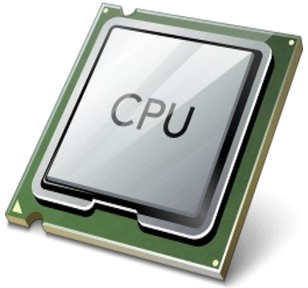
```
threadproc(...) {
  while(true) {
      message = GetMessage();
      switch(message) {
      case WM_THIS: DoThis();
      case WM_THAT: DoThat();
      }
  }
}
```



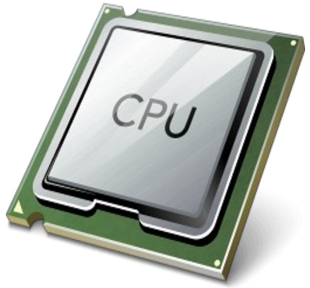

```
threadproc(...) {
  while(true) {
      message = GetMessage();
      switch(message) {
      case WM_THIS: DoThis();
      case WM_THAT: DoThat();
      }
  }
}
```




```
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?

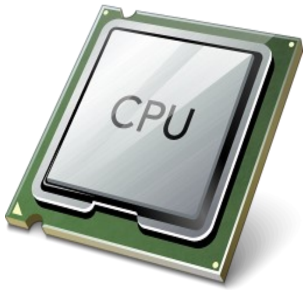
```
winmain() {
    for(i=0; i<NUMPROCS; i++)
        pthread_create(&tids[i], HandlerProc);
    for(i=0; i<NUMPROCS; i++)
        pthread_join(&tids[i]);
}
```

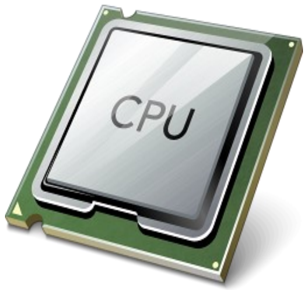
```
threadproc(...) {
  while(true) {
      message = GetMessage();
      switch(message) {
      case WM_THIS: DoThis();
      case WM_THAT: DoThat();
      }
  }
}
```

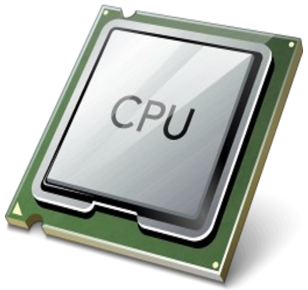
```
threadproc(...) {
  while(true) {
      message = GetMessage();
      switch(message) {
      case WM_THIS: DoThis();
      case WM_THAT: DoThat();
      }
  }
}
```

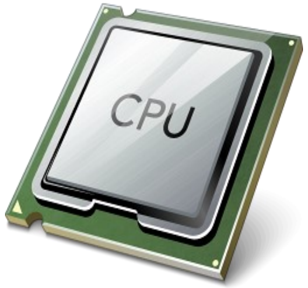
```
threadproc(...) {
  while(true) {
      message = GetMessage();
      switch(message) {
      case WM_THIS: DoThis();
      case WM_THAT: DoThat();
      }
  }
}
```

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









Parallel GUI Implementation 2

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

Pros/cons?

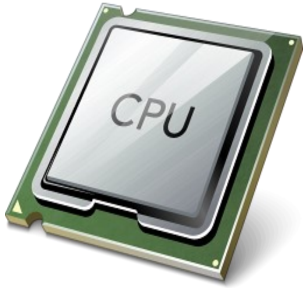
Pros:

- Preserves programming model
- Can recover some parallelism

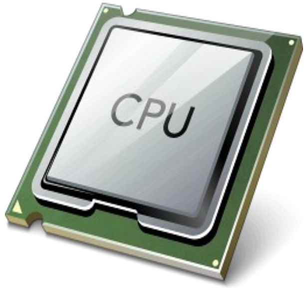
Cons:

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

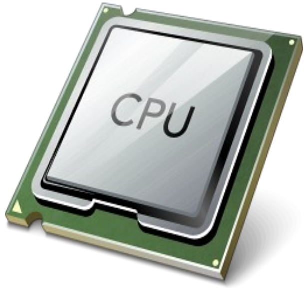
```
threadproc(...) {
  while(true) {
      message = GetMessage();
      switch(message) {
      case WM_THIS: DoThis();
      case WM_THAT: DoThat();
      }
  }
}
```



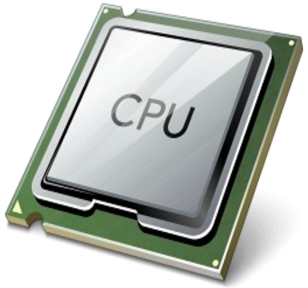

```
threadproc(...) {
  while(true) {
      message = GetMessage();
      switch(message) {
      case WM_THIS: DoThis();
      case WM_THAT: DoThat();
      }
  }
}
```


```
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

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

Pros/cons?

Pros:
- Preserves programming model
- Can recover some parallelism

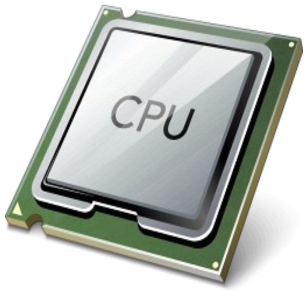
Cons:
- Workers still have same problem
- How to load balance?
- Shared mutable state a problem

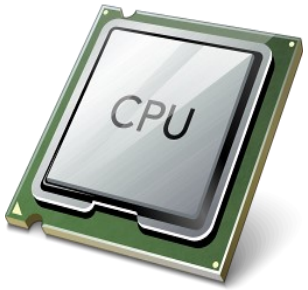
```
threadproc(...) {
  while(true) {
      message = GetMessage();
      switch(message) {
      case WM_THIS: DoThis();
      case WM_THAT: DoThat();
      }
  }
}
```

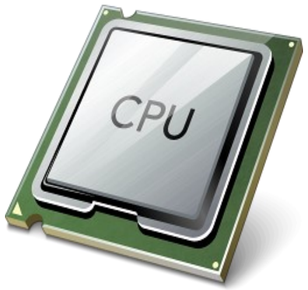
```
threadproc(...) {
  while(true) {
      message = GetMessage();
      switch(message) {
      case WM_THIS: DoThis();
      case WM_THAT: DoThat();
      }
  }
}
```

```
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);
}
```

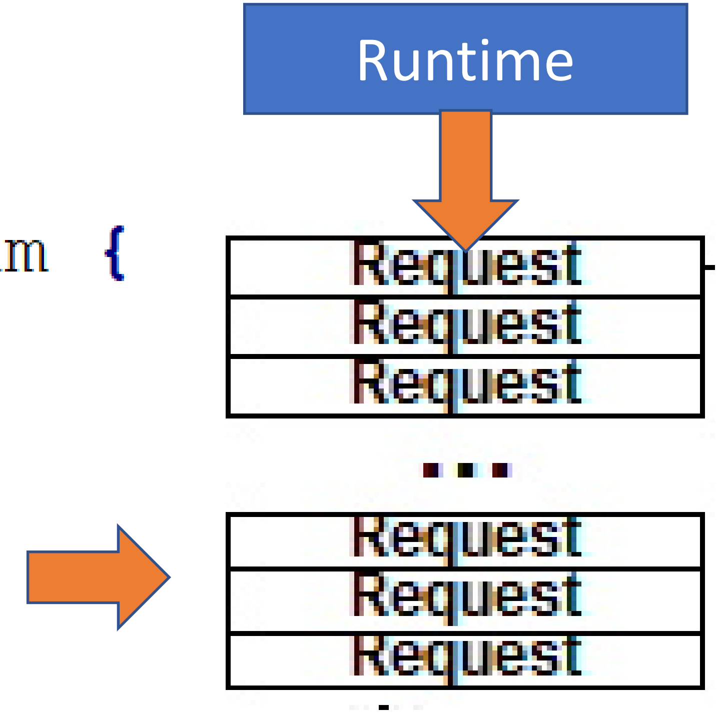
Event-based programming

Event-based programming

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

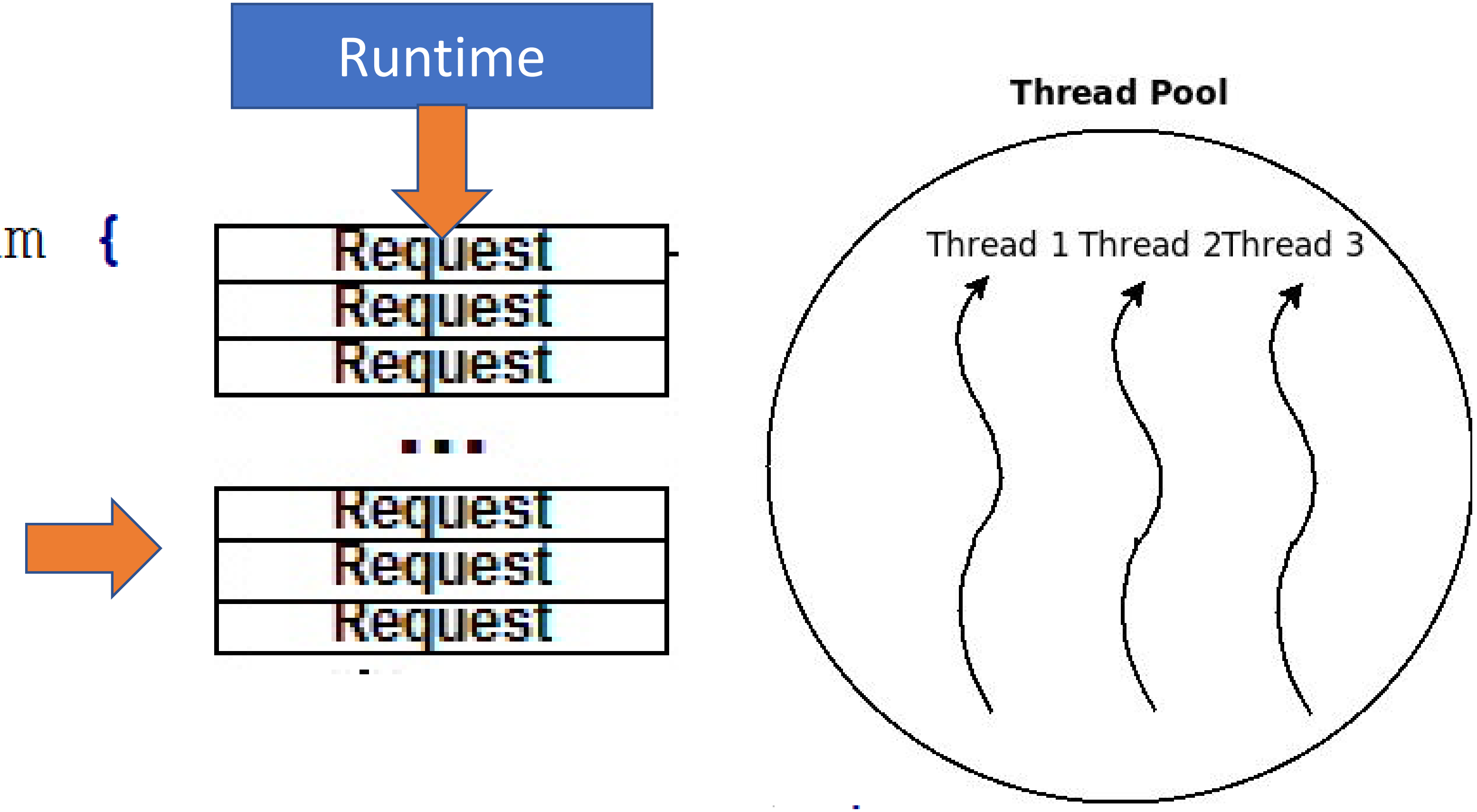
Event-based programming

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



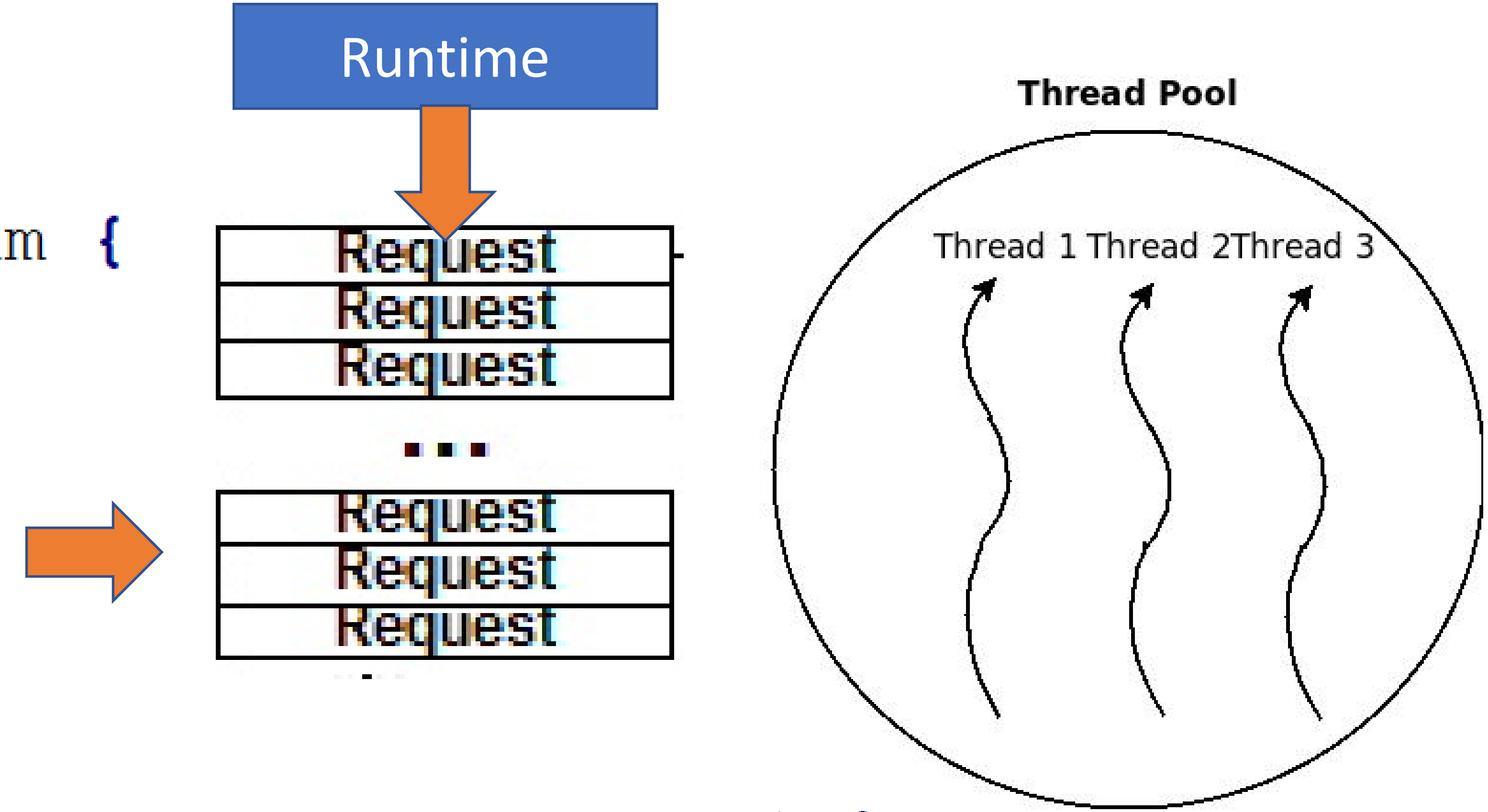
Event-based programming

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



Event-based programming

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



Is the problem solved?

Another Event-based Program

Another Event-based Program

```
PROGRAM MyProgram {
    OnOpenFile() {
        char szFileName[BUFSIZE]
        InitFileName(szFileName);
        FILE file = ReadFileEx(szFileName);
        LoadFile(file);
        RedrawScreen();
    }
    OnPaint();
}
```

Another Event-based Program

```
PROGRAM MyProgram {
    OnOpenFile() {
        char szFileName[BUFSIZE]
        InitFileName(szFileName);
        FILE file = ReadFileEx(szFileName);
        LoadFile(file);
        RedrawScreen();
    }
    OnPaint();
}
```

Blocks!

Another Event-based Program

```
PROGRAM MyProgram {
    OnOpenFile() {
        char szFileName[BUFSIZE]
        InitFileName(szFileName);
        FILE file = ReadFileEx(szFileName);
        LoadFile(file);
        RedrawScreen();
    }
    OnPaint();
}
```

Burns CPU!

Blocks!

Another Event-based Program

```
PROGRAM MyProgram {
    OnOpenFile() {
        char szFileName[BUFSIZE]
        InitFileName(szFileName);
        FILE file = ReadFileEx(szFileName);
        LoadFile(file);
        RedrawScreen();
    }
    OnPaint();
}
```

Uses Other Handlers! (call OnPaint?)

Burns CPU!

Blocks!

No problem!
Just use more events/handlers, right?

```
PROGRAM MyProgram {
    TASK ReadFileAsync(name, callback) {
        ReadFileSync(name);
        Call(callback);
    }
    CALLBACK FinishOpeningFile() {
        LoadFile(file);
        RedrawScreen();
    }
    OnOpenFile() {
        FILE file;
        char szName[BUFSIZE]
        InitFileName(szName);
        EnqueueTask(ReadFileAsync(szName, FinishOpeningFile));
    }
    OnPaint();
}
```

Continuations, BTW

```
PROGRAM MyProgram {
    OnOpenFile() {
        ReadFile(file, FinishOpeningFile);
    }
    OnFinishOpeningFile() {
        LoadFile(file, OnFinishLoadingFile);
    }
    OnFinishLoadingFile() {
        RedrawScreen();
    }
    OnPaint();
}
```

Stack-Ripping

```
PROGRAM MyProgram {
    TASK ReadFileAsync(name, callback) {
        ReadFileSync(name);
        Call(callback);
    }
    CALLBACK FinishOpeningFile() {
        LoadFile(file);
        RedrawScreen();
    }
    OnOpenFile() {
        FILE file;
        char szName[BUFSIZE]
        InitFileName(szName);
        EnqueueTask(ReadFileAsync(szName, FinishOpeningFile));
    }
    OnPaint();
}
```

Stack-Ripping

```
PROGRAM MyProgram {
    TASK ReadFileAsync(name, callback) {
        ReadFileSync(name);
        Call(callback);
    }
    CALLBACK FinishOpeningFile() {
        LoadFile(file);
        RedrawScreen();
    }
    OnOpenFile() {
        FILE file;
        char szName[BUFSIZE]
        InitFileName(szName);
        EnqueueTask(ReadFileAsync(szName, FinishOpeningFile));
    }
    OnPaint();
}
```

Stack-Ripping

```
PROGRAM MyProgram {
    TASK ReadFileAsync(name, callback) {
        ReadFileSync(name);
        Call(callback);
    }
    CALLBACK FinishOpeningFile() {
        LoadFile(file);
        RedrawScreen();
    }
    OnOpenFile() {
        FILE file;
        char szName[BUFSIZE]
        InitFileName(szName);
        EnqueueTask(ReadFileAsync(szName, FinishOpeningFile));
    }
    OnPaint();
}
```

Stack-Ripping

```
PROGRAM MyProgram {
    TASK ReadFileAsync(name, callback) {
        ReadFileSync(name);
        Call(callback);
    }
    CALLBACK FinishOpeningFile() {
        LoadFile(file);
        RedrawScreen();
    }
    OnOpenFile() {
        FILE file;
        char szName[BUFSIZE]
        InitFileName(szName);
        EnqueueTask(ReadFileAsync(szName, FinishOpeningFile));
    }
    OnPaint();
}
```

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

Threads vs Events

- Thread Pros
- Thread Cons

- Event Pros
- 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 requir
  - Overflowable stack
  - Stack memory pressure

- Event Pros
  - Easier to create well-conditioned system
  - Easier to express dynamic change in level of parallelism
- Cons
  - cult to program
  - trol flow between callbacks obscure
  - hen 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 requi
  - Overflowable stack
  - Stack memory pressure

- Event Pros
  - Easier to create well-conditioned system
  - Easier to express dynamic change in level of parallelism
- Cons
  - cult to program
  - trol flow between callbacks obscure
  - hen to deallocate memory
  - Incomplete language/tool/debugger support
  - Difficult to exploit concurrent hardware

Language-level Futures: the sweet spot?

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, _ThreadPoolProc, 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, _ThreadPoolProc, 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;
}
```

Redux: Futures in Context

Redux: Futures in Context

Futures:

Redux: Futures in Context

Futures:

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

Redux: 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

Redux: 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

```
static void runAsyncExample() {
    CompletableFuture cf = CompletableFuture.runAsync(() -> {
        assertTrue(Thread.currentThread().isDaemon());
        randomSleep();
    });
    assertFalse(cf.isDone());
    sleepEnough();
    assertTrue(cf.isDone());
}
```

Redux: 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:

```
static void runAsyncExample() {
    CompletableFuture cf = CompletableFuture.runAsync(() -> {
        assertTrue(Thread.currentThread().isDaemon());
        randomSleep();
    });
    assertFalse(cf.isDone());
    sleepEnough();
    assertTrue(cf.isDone());
}
```

Redux: 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

```java
static void runAsyncExample() {
    CompletableFuture cf = CompletableFuture.runAsync(() -> {
        assertTrue(Thread.currentThread().isDaemon());
        randomSleep();
    });
    assertFalse(cf.isDone());
    sleepEnough();
    assertTrue(cf.isDone());
}
```

Redux: 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

```java
static void runAsyncExample() {
    CompletableFuture cf = CompletableFuture.runAsync(() -> {
        assertTrue(Thread.currentThread().isDaemon());
        randomSleep();
    });
    assertFalse(cf.isDone());
    sleepEnough();
    assertTrue(cf.isDone());
}
```

Redux: 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

```
static void runAsyncExample() {
    CompletableFuture cf = CompletableFuture.runAsync(() -> {
        assertTrue(Thread.currentThread().isDaemon());
        randomSleep();
    });
    assertFalse(cf.isDone());
    sleepEnough();
    assertTrue(cf.isDone());
}
```

Memory Consistency

Memory Consistency

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

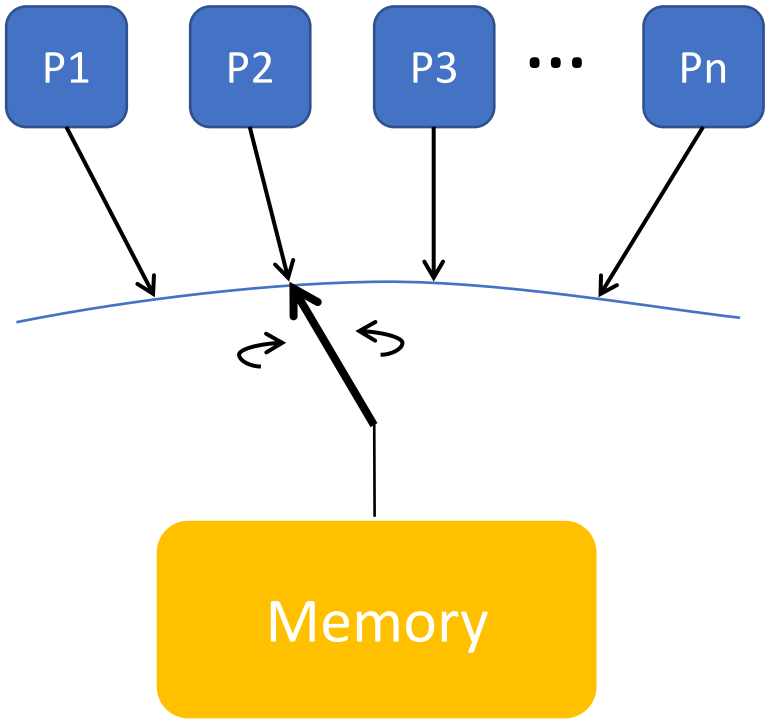
Memory Consistency

- 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

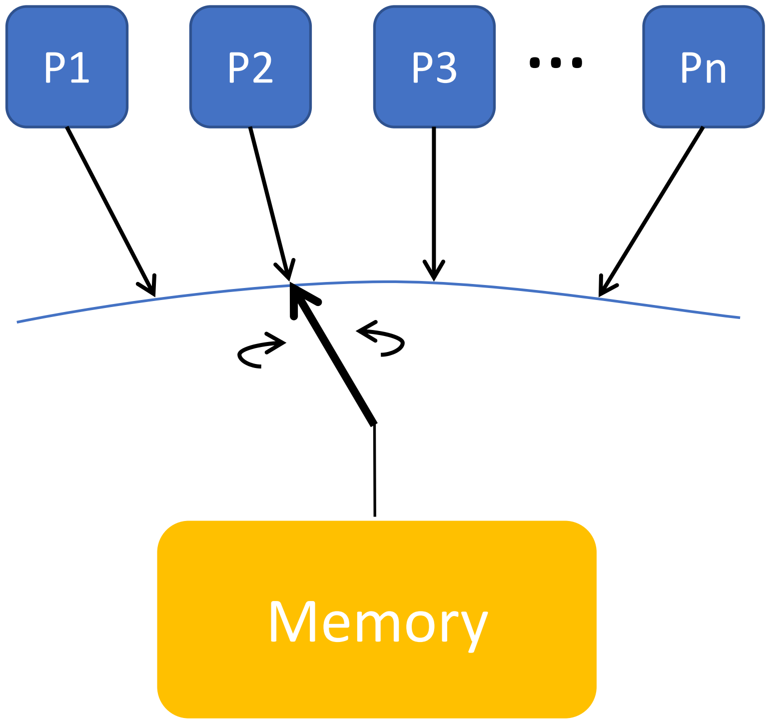
Sequential Consistency

- 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	**P2**	**P3**
A = 1		
	if (A == 1) B = 1	
		if (B == 1) register1 = A

Do we need Sequential Consistency?

Initially, A = B = 0

P1	**P2**	**P3**
A = 1		
	if (A == 1) B = 1	
		if (B == 1) register1 = A

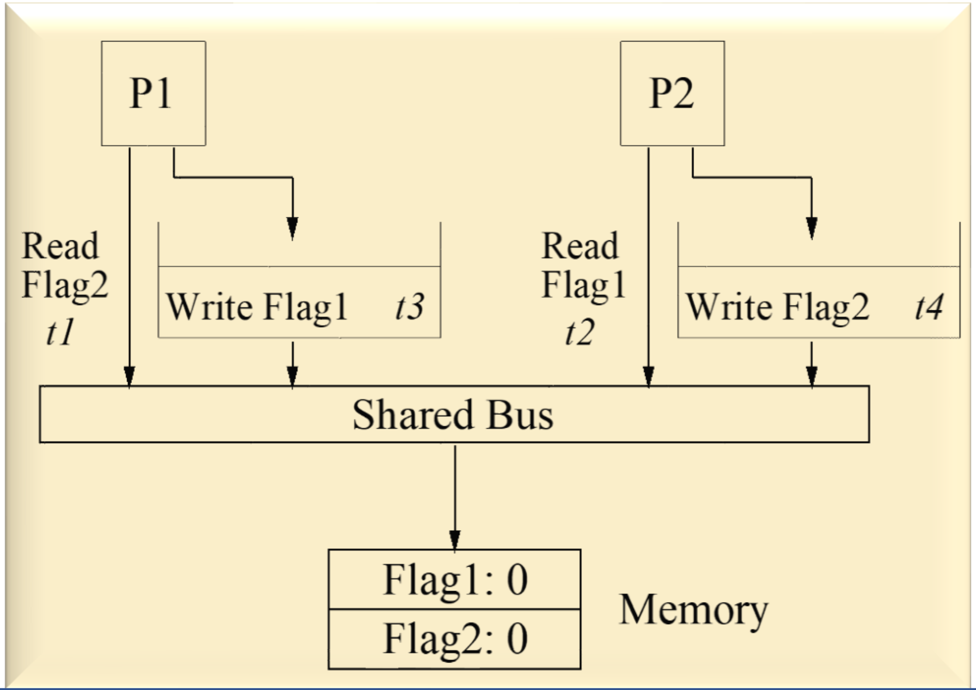
Key issue:

- 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	**P2**
A = 1	
	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

- P_0 write → queue op in write buffer, proceed
- P_0 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 → R; W → W; R → R/W

Relaxed Consistency Models

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

Relaxed Consistency Models

- **Program Order** relaxations *(different locations)*
  - W → R; W → W; R → 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 → R; W → W; R → 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 → R; W → W; R → R/W
- **Write Atomicity** relaxations
  - Read returns another processor's W
- 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?