Asynchronous Programming Promises + Futures

Chris Rossbach

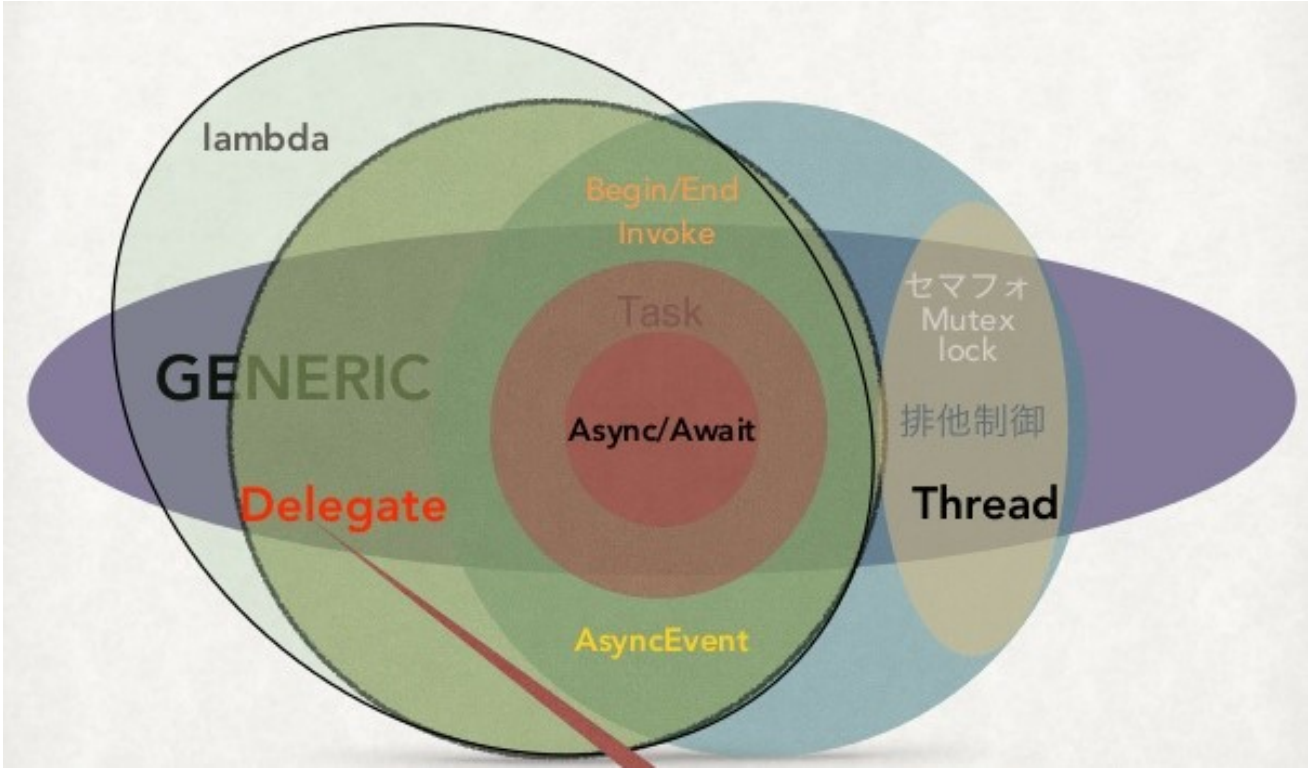
CS378H

Today

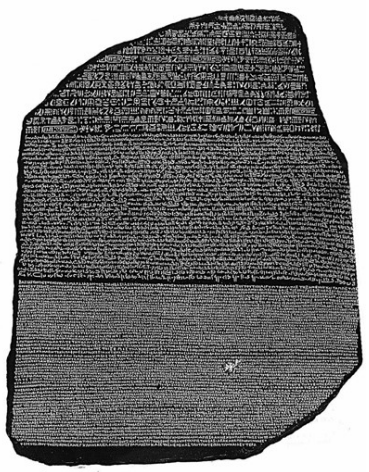
- Questions?
- Administrivia
- Material for the day
  - Events / Asynchronous programming
  - Promises & Futures
  - Bonus: memory consistency models

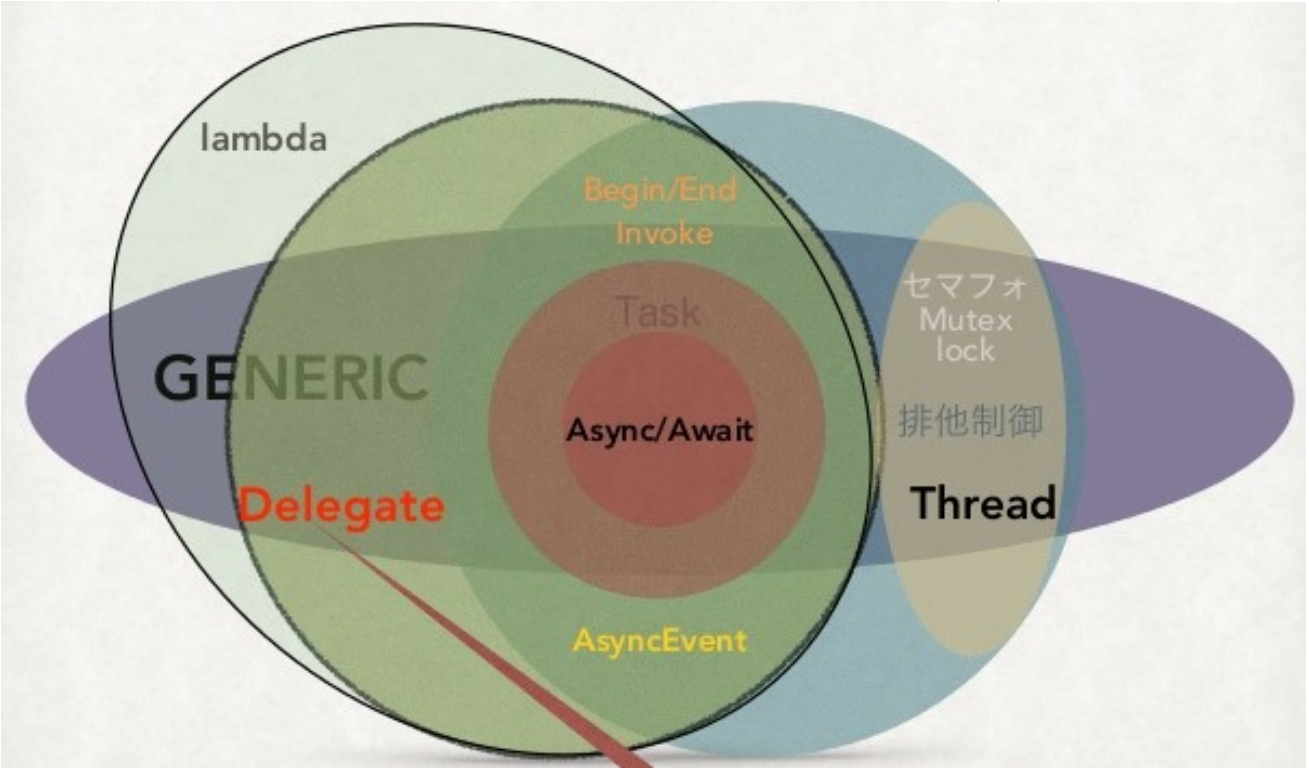
- Acknowledgements
  - Consistency slides borrow some materials from Kevin Boos. Thanks!

Asynchronous Programming
Events, Promises, and Futures



Asynchronous Programming
Events, Promises, and Futures





Review: Java Example

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

Review: Java Example

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

Review: 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

Review: 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

Review: 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

Review: 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)

Review: 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?

Review: Java Example

```
1 static void runAsyncExample() {
```

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 Programming Model between:

- Event-based programming
- Thread-based programming

- Where (on what thread) does the lambda expression run?

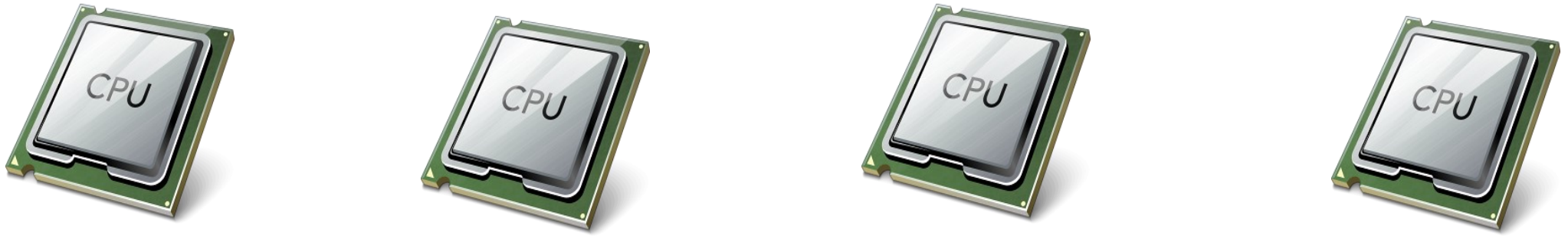
GUI Programming Distilled

```
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?

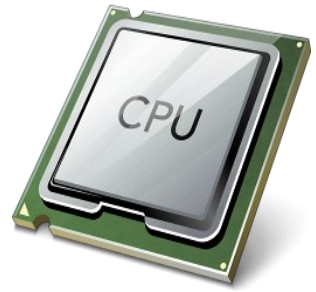


Parallel GUI Implementation 1

```
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 }
```

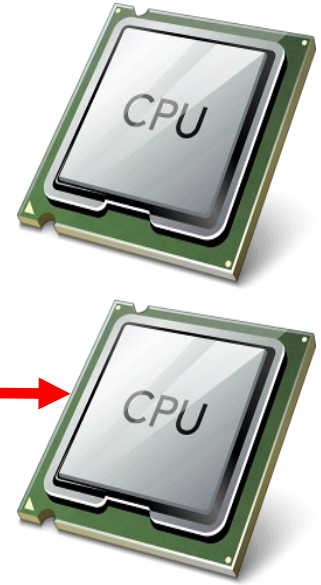
Parallel GUI Implementation 1

```
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 }
```



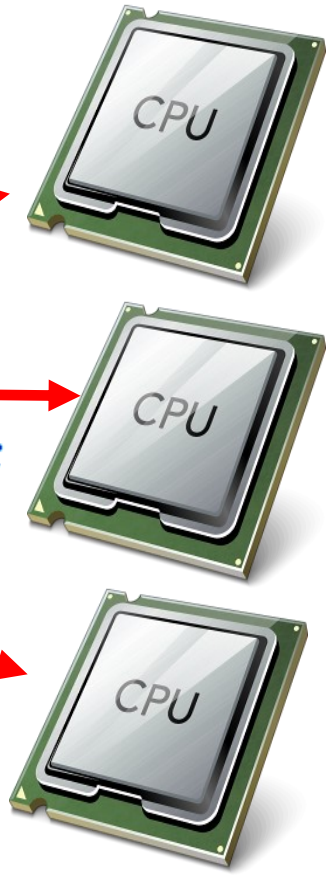
Parallel GUI Implementation 1

```
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 }
```



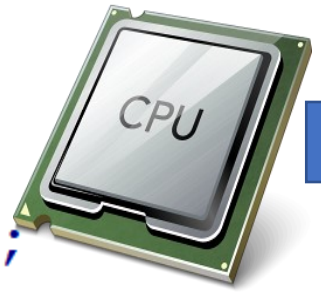
Parallel GUI Implementation 1

```
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 }
```

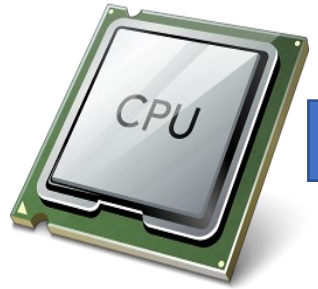


Parallel GUI Implementation 1

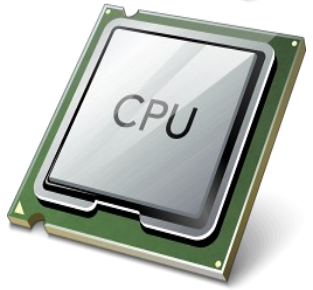
```
winmain() {
    pthread_create(&tids[i++], DoThisProc);
    pthread_create(&tids[i++], DoThatProc);
    pthread_create(&tids[i++], DoOtherThingProc);
    for(j=0; j<i; j++)
        pthread_join(&tids[j]);
}

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



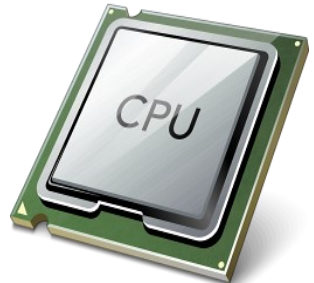
DoThisProc



DoThatProc



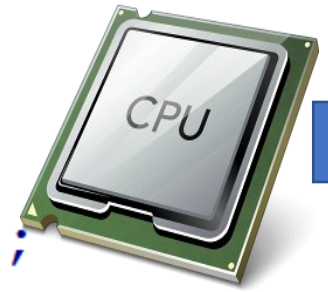
OtherThing



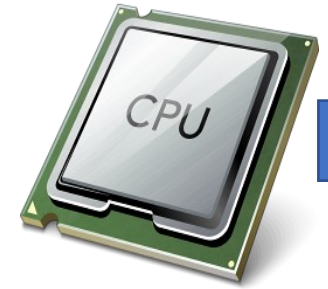
Parallel GUI Implementation 1

```
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();
    }
}
```

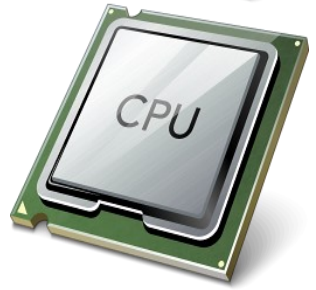
Pros/cons?



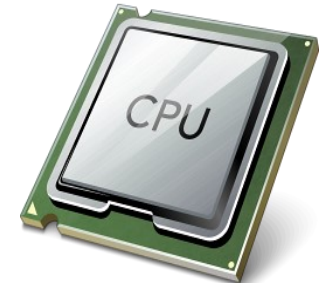
DoThisProc



DoThatProc



OtherThing



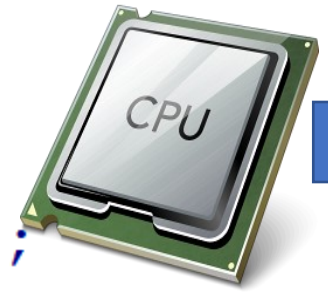
Parallel GUI Implementation 1

```
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();
    }
}
```

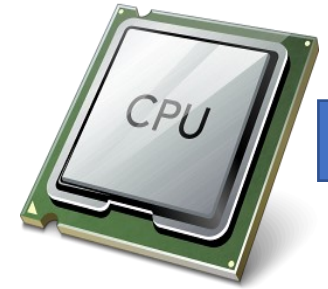
Pros/cons?

Pros:
- Encapsulates parallel work

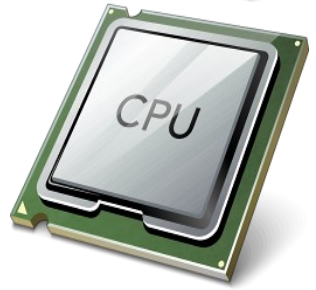
Cons:
- Obliterates original code structure
- How to assign handlers→CPUs?
- Load balance?!?
- Utilization



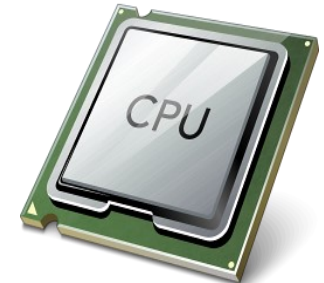
DoThisProc



DoThatProc



OtherThing



Parallel GUI Implementation 2

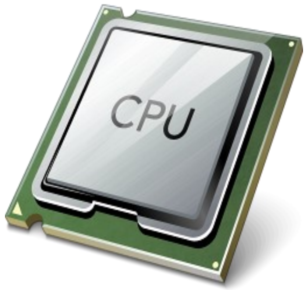
```
winmain() {
    for(i=0; i<NUMPROCS; i++)
        pthread_create(&tids[i], HandlerProc);
    for(i=0; i<NUMPROCS; i++)
        pthread_join(&tids[i]);
}
```

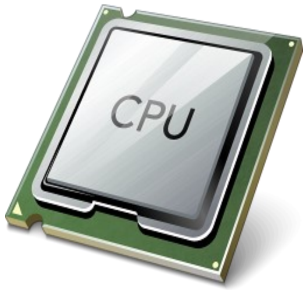
```
threadproc(...) {
  while(true) {
      message = GetMessage();
      switch(message) {
      case WM_THIS: DoThis();
      case WM_THAT: DoThat();
      }
  }
}
```

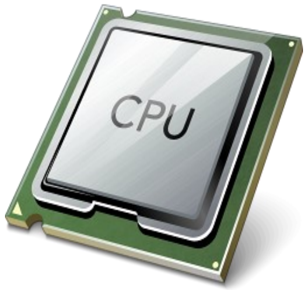

```
threadproc(...) {
  while(true) {
      message = GetMessage();
      switch(message) {
      case WM_THIS: DoThis();
      case WM_THAT: DoThat();
      }
  }
}
```

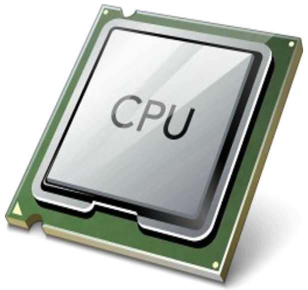
```
threadproc(...) {
  while(true) {
      message = GetMessage();
      switch(message) {
      case WM_THIS: DoThis();
      case WM_THAT: DoThat();
      }
  }
}
```

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







Parallel GUI Implementation 2

Pros/cons?

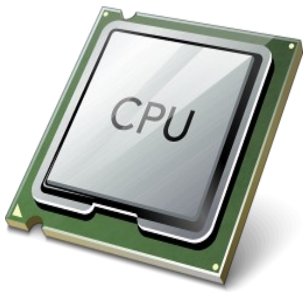
```
winmain() {
    for(i=0; i<NUMPROCS; i++)
        pthread_create(&tids[i], HandlerProc);
    for(i=0; i<NUMPROCS; i++)
        pthread_join(&tids[i]);
}
```

```
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:

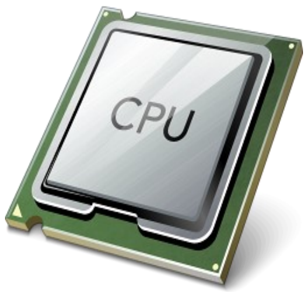
- Workers still have same problem
- How to load balance?
- Shared mutable state a problem

```
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

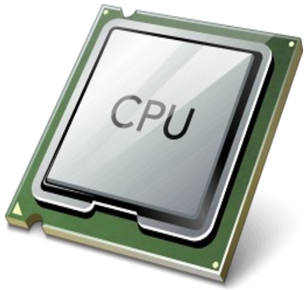
Cons:
- Workers still have same problem
- How to load balance?
- Shared mutable state a problem

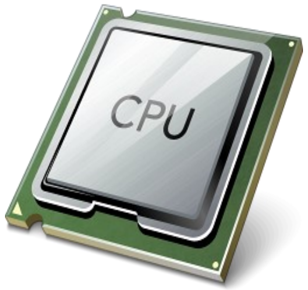
```
threadproc(...) {
  while(true) {
      message = GetMessage();
      switch(message) {
      case WM_THIS: DoThis();
      case WM_THAT: DoThat();
      }
  }
}
```

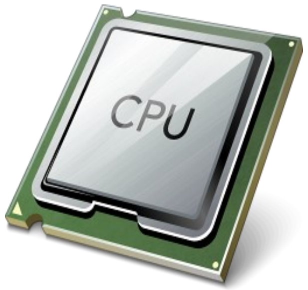
```
threadproc(...) {
  while(true) {
      message = GetMessage();
      switch(message) {
      case WM_THIS: DoThis();
      case WM_THAT: DoThat();
      }
  }
}
```

```
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);
}
```

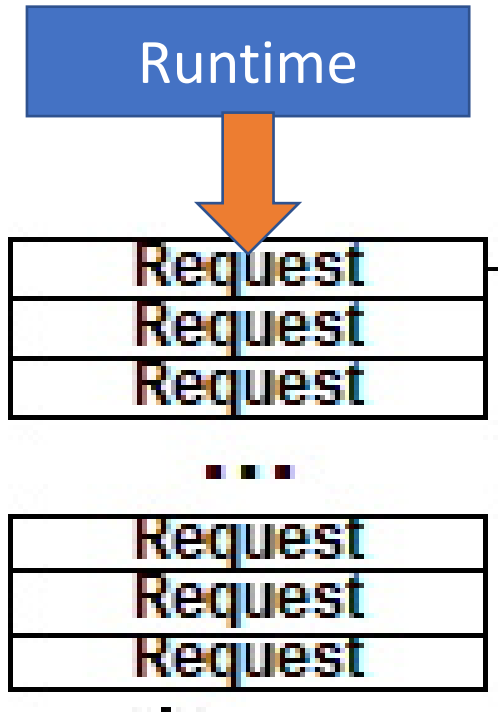
Event-based programming

Event-based programming

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

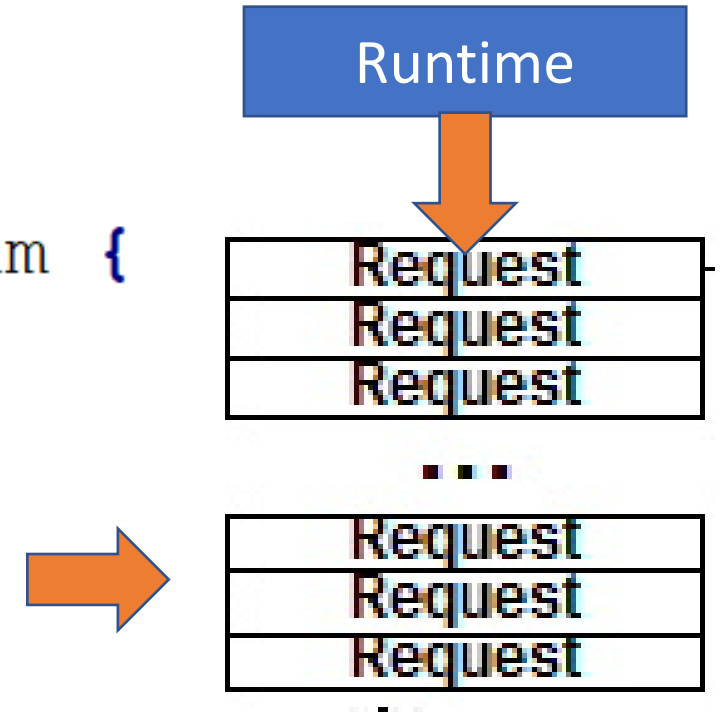
Event-based programming

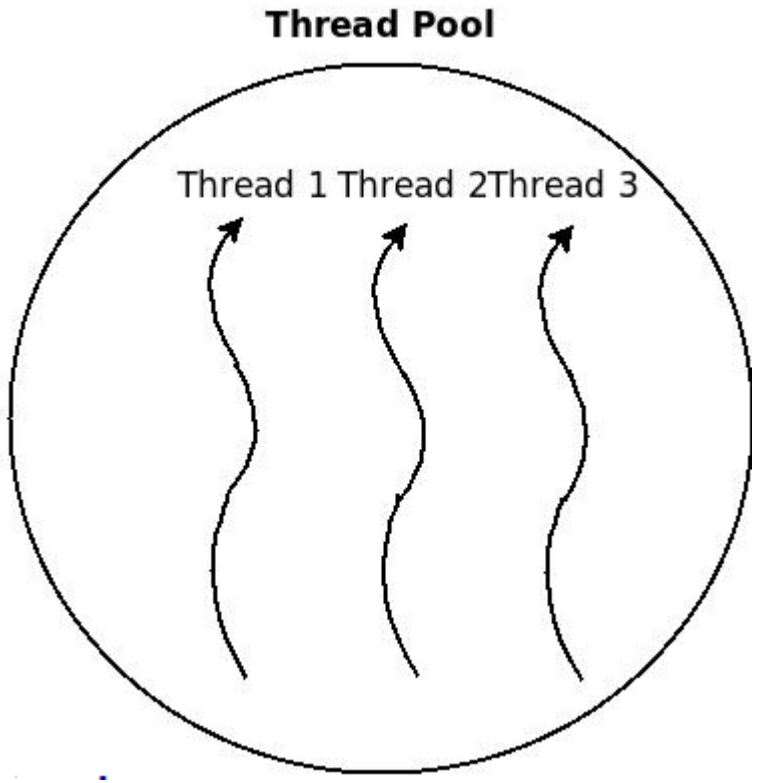
```
PROGRAM MyProgram {
    OnSize() {}
    OnMove() {}
    OnClick() {}
    OnPaint() {}
}
```



Event-based programming

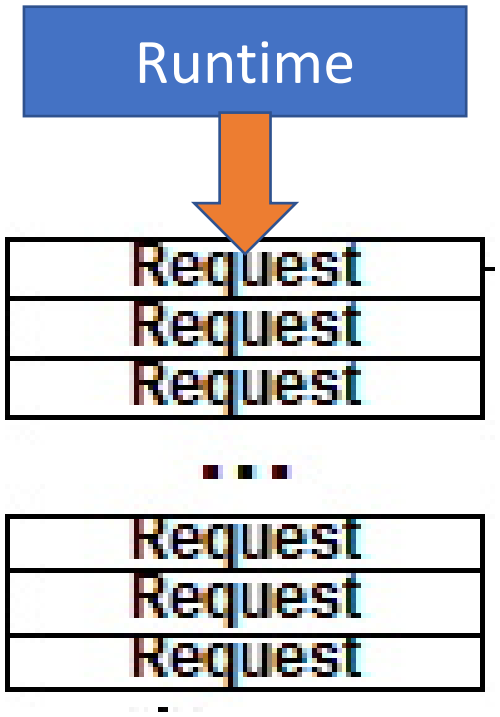
```
PROGRAM MyProgram {
    OnSize() {}
    OnMove() {}
    OnClick() {}
    OnPaint() {}
}
```

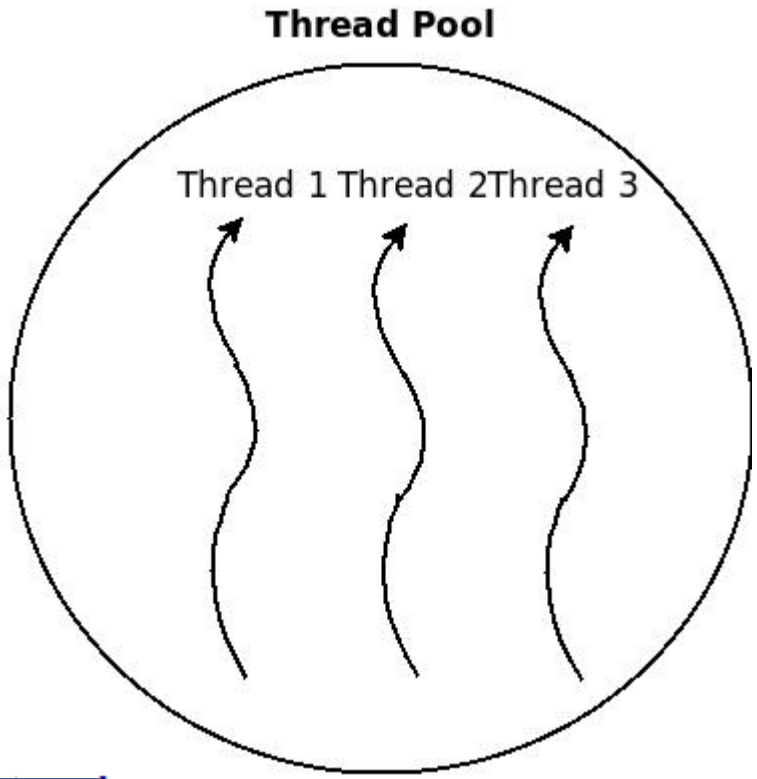




Event-based programming

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





Is the problem solved?

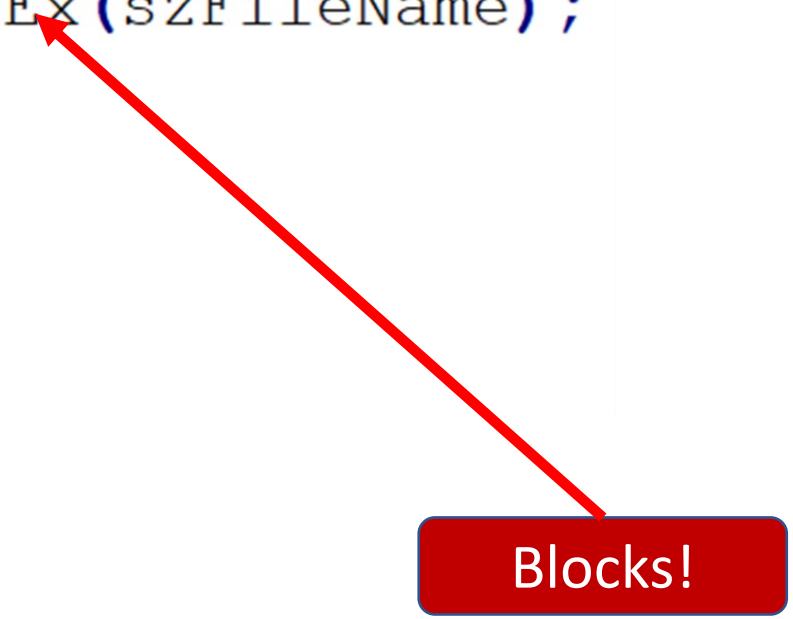
Another Event-based Program

Another Event-based Program

```
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 }
```

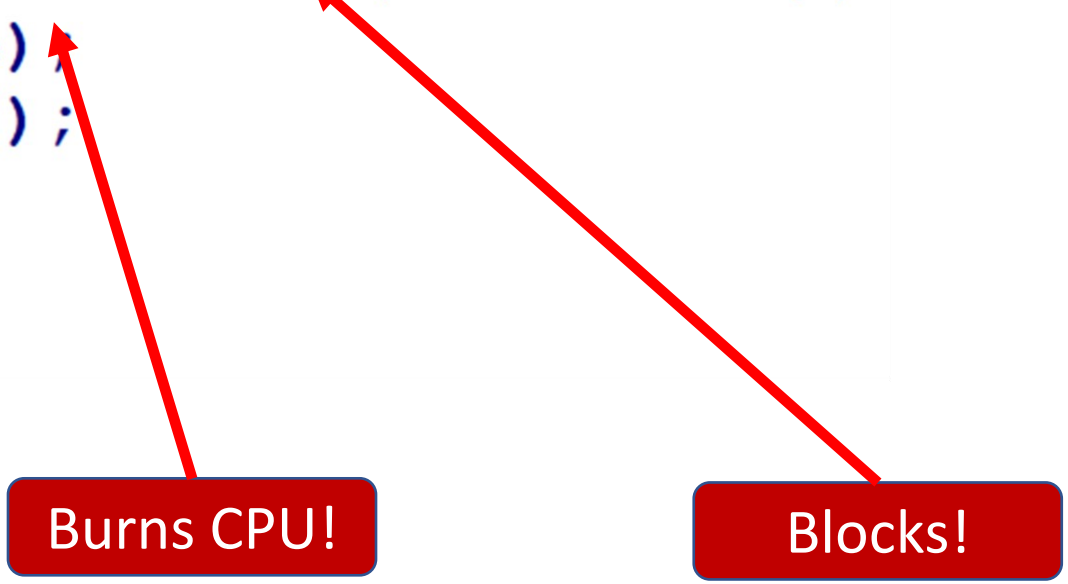
Another Event-based Program

```
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 }
```



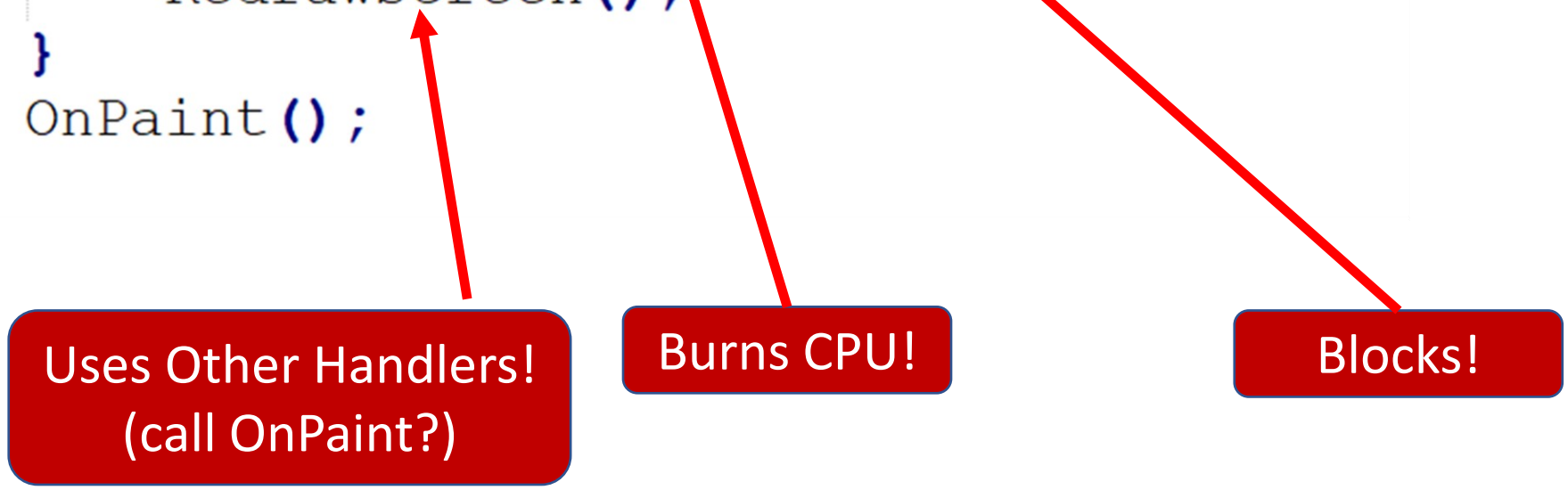
Another Event-based Program

```
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 }
```



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
- 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

Language-level Futures: the sweet spot?

- 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
  - 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

Language-level Futures: the sweet spot?

- 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
  - When to deallocate memory
  - Incomplete language/tool/debugger support
  - Difficult to exploit concurrent hardware

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

```
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 }
```

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:

```
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 }
```

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

```
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 }
```

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

```
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 }
```

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

```
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 }
```

Questions?