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

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

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

CompletableFuture is a container for Future object type

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

CompletableFuture is a container for Future object type

cf is an instance

runAsync() accepts
  • Lambda expression
  • Anonymous function
  • Functor

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

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?
```c
void 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;
        }
    }
}
```
```c
int 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;
        }
    }
}
```
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;
        }
    }
}
```c
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;
        }
    }
}
```
```c
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;
        }
    }
}
```
```cpp
void 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

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

void DoThisProc() {
    while(true){
        if(ThisHasHappened)
            DoThis();
    }
}
```
Parallel GUI Implementation 1

```c
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:**
- Encapsulates parallel work

**Cons:**
- Obliterates original code structure
- How to assign handlers → CPUs?
- Load balance?!?
- Utilization
```c
int main() {
    for (i=0; i<NUMPROCS; i++)
        pthread_create(&tids[i], HandlerProc);
    for (i=0; i<NUMPROCS; i++)
        pthread_join(&tids[i]);
}

while (true) {
    message = GetMessage();
    switch (message) {
        case WM_THIS: DoThis();
        case WM_THAT: DoThat();
    }
}
```
Parallel GUI Implementation 2

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

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

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

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

```c
while(true) {
    message = GetMessage();
    switch(message) {
        case WM_THIS: DoThis();
        case WM_THAT: DoThat();
    }
}
```
Parallel GUI Implementation 2

Pros:
- Preserves programming model
- Can recover some parallelism

Cons:
- Workers still have same problem
- How to load balance?
- Shared mutable state a problem
Parallel GUI Implementation 2

Pros:
- Preserves programming model
- Can recover some parallelism

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

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

```c
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() {}
}

Runtime
Event-based programming

```plaintext
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

```java
PROGRAM MyProgram {
  OnOpenFile() {
    char szFileName[BUFSIZE]
    InitFileName(szFileName);
    FILE file = ReadFileEx(szFileName);
    LoadFile(file);
    RedrawScreen();
  }
  OnPaint();
}
```
Another Event-based Program

```c
PROGRAM MyProgram {
    OnOpenFile() {
        char szFileName[BUFSIZE]
        InitFileName(szFileName);
        FILE file = ReadFileEx(szFileName);
        LoadFile(file);
        RedrawScreen();
    }
    OnPaint();
}
```
Another Event-based Program

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

Burns CPU!

Blocks!
Another Event-based Program

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

- **Burns CPU!**
- **Blocks!**
- **Uses Other Handlers!** (call OnPaint?)
No problem!
Just use more events/handlers, right?

```c
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

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

```c
PROGRAM MyProgram {
    TASK ReadFileSync(name, callback) {
        ReadFileSync(name);
        Call(callback);
    }

    CALLBACK FinishOpeningFile() {
        LoadFile(file);
        RedrawScreen();
    }

    OnOpenFile() {
        FILE file;
        char szName[BUFSIZE]
        InitFileName(szName);
        EnqueueTask(ReadFileSync(szName, FinishOpeningFile));
    }

    OnPaint();
}
```
Stack-Ripping

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

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

- **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 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 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

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 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

Language-level Futures: the sweet spot?
Thread Pool Implementation

```c
void ThreadPool::StartThreads(
    UINT uiThreads,
    BOOL bWaitAllThreadsAlive
)
{
    Lock();
    if (uiThreads != 0 && _m.vnThreadDesc.size() < _m.uiTargetSize)
        ResetEvent(_m.vnAllThreadsAlive);
    while (_m.vnThreadDesc.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.vnAvailable.push_back(*phThread);
            _m.vnThreadDescs[*phThread] = pDesc;
        }
    }
    _m.uiThreads = (UINT)_m.vnThreadDesc.size();
    Unlock();
    if (bWaitAllThreadsAlive)
        WaitThreadsAlive();
}```
Thread Pool Implementation

```c
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

```
DWORD ThreadPool::ThreadProc(
    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->bRuntimeValid = 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

```java
static void runAsyncExample()
{
    CompletableFuture cf = CompletableFuture.runAsync()
    .thenAccept(
        thread -> {
            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
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

```java
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?