Asynchronous Programming
Promises + Futures

Chris Rossbach
CS378H Fall 2018
9/26/18
Today

• Questions?
• Administrivia
  • Lab 2 due today
• 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
Review: Models for Concurrency

• Concrete execution model:
  • CPUs 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

• 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

```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?

Promise: “thing to be done”

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

<table>
<thead>
<tr>
<th>Language</th>
<th>Promise</th>
<th>Future</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algol</td>
<td>Thunk</td>
<td>Address of async result</td>
</tr>
<tr>
<td>Java</td>
<td>CompletableFuture&lt;T&gt;</td>
<td>Future&lt;T&gt;</td>
</tr>
<tr>
<td>C#/.NET</td>
<td>TaskCompletionSource&lt;T&gt;</td>
<td>Task&lt;T&gt;</td>
</tr>
<tr>
<td>JavaScript</td>
<td>Deferred</td>
<td>Promise</td>
</tr>
<tr>
<td>C++</td>
<td>std::promise</td>
<td>std::future</td>
</tr>
</tbody>
</table>

**Mnemonic:**
Promise to *do* something
Make a promise *for* the future
Putting Futures in Context
My unvarnished opinion

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 Models:

• Event-based
• Thread-based

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

do {
    WaitForSomething();
    RespondToThing();
} until(forever);
GUI Programming

```c
// Step 2: Creating the Window
hwnd = CreateWindowEx(
    WS_EX_CLIENTEDGE,
    g_szClassName,
    "The title of my window",
    WS_OVERLAPPEDWINDOW,
    CW_USEDEFAULT, CW_USEDEFAULT, 240, 120,
    NULL, NULL, hInstance, NULL);

// Step 3: The Message Loop
while(GetMessage(&Msg, NULL, 0, 0) > 0) {
    TranslateMessage(&Msg);
    DispatchMessage(&Msg);
}
```
GUI programming

void OnMove() { ... }
void OnSize() { ... }
void OnPaint() { ... }

Over 1000 last time I checked!
GUI Programming Distilled

Pros
• Simple imperative programming
• Good fit for uni-processor

Cons
• Awkward/verbose
• Obscures available parallelism

```c
winmain(...) {
    while(true) {
        message = GetMessage();
        switch(message) {
            case WM_LONGRUNNING_CPU_HOG: HogCPU(); break;
            case WM_HIGH_LATENCY_IO: BlockForALongTime(); break;
            case WM_DO_QUICK_IMPORTANT_THING: HopeForTheBest(); break;
        }
    }
}```
How can we parallelize this?
Parallel GUI Implementation 1

```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;
        }
    }
}
```
Parallel GUI Implementation 1

Pros:
- Encapsulates parallel work

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

```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();
    }
}
```
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

• 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

• 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

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);
}
Another Event-based Program

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

```plaintext
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();
}
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

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

```cpp
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* pThread = &pDesc->hThread;
            *pThread = CreateThread(NULL, 0, ThreadPoolProc, pDesc, 0, NULL);
            m_vhAvailable.push_back(*pThread);
            m_vhThreadDescs[*pThread] = pDesc;
        }
    }
    m_uiThreads = (UINT)m_vhThreadDescs.size();
    Unlock();
    if (bWaitAllThreadsAlive)
        WaitThreadsAlive();
}
```

Cool project idea: build a thread pool!
Thread Pool Implementation

```c
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) {
            LPSTART_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

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

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

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

Can both P1 and P2 wind up in the critical section at the same time?
Do we need Sequential Consistency?

Initially, $A = B = 0$

P1

A = 1

if (A == 1)

B = 1

P2

Key issue:

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

Write Buffers

• P0 write $\rightarrow$ queue op in write buffer, proceed
• P0 read $\rightarrow$ look in write buffer,
• P(x != 0) read $\rightarrow$ old value: write buffer hasn’t drained
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

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

- **Write Atomicity** relaxations
  - Read returns another processor’s Write

- Combined relaxations
  - Read your own Write (okay for S.C.)

- **Requirement**: synchronization primitives
  - Fence, barrier instructions etc

---

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