Parallel Systems
Events and Futures

Chris Rossbach + Calvin Lin
CS380p
Outline for Today

• Asynchronous Programming Models
  • Events
  • Futures
Review: Parallel Programming Models
Review: Parallel Programming Models

• Concrete model:
  • CPU(s) execute instructions sequentially
Review: Parallel Programming Models

- Concrete model:
  - CPU(s) execute instructions sequentially

- Dimensions:
  - How to specify computation
  - How to specify communication
  - How to specify coordination/control transfer
Review: Parallel Programming Models

• Concrete model:
  • CPU(s) execute instructions sequentially

• Dimensions:
  • How to specify computation
  • How to specify communication
  • How to specify coordination/control transfer

• Techniques/primitives
  • Threads/Processes
  • Message passing vs shared memory
  • Preemption vs Non-preemption
Review: Parallel Programming Models

• Concrete model:
  • CPU(s) execute instructions sequentially

• Dimensions:
  • How to specify computation
  • How to specify communication
  • How to specify coordination/control transfer

• Techniques/primitives
  • Threads/Processes
  • Message passing vs shared memory
  • Preemption vs Non-preemption

• Dimensions/techniques not always orthogonal
Review: Execution Context Management

“Task” == “Flow of Control”
“Stack” == Task State
Review: Execution Context Management

“Task” == “Flow of Control”
“Stack” == Task State

Task Management
Review: Execution Context Management

“Task” == “Flow of Control”
“Stack” == Task State

Task Management

• Preemptive
  • Interleave on uniprocessor
  • Overlap on multiprocessor
Review: Execution Context Management

“Task” == “Flow of Control”
“Stack” == Task State

Task Management

• Preemptive
  • Interleave on uniprocessor
  • Overlap on multiprocessor

• Serial
  • One at a time, no conflict
Review: Execution Context Management

“Task” == “Flow of Control”
“Stack” == Task State

Task Management

• Preemptive
  • Interleave on uniprocessor
  • Overlap on multiprocessor

• Serial
  • One at a time, no conflict

• Cooperative
  • Yields at well-defined points
  • E.g. wait for long-running I/O
Review: Execution Context Management

“Task” == “Flow of Control”
“Stack” == Task State

Task Management

• Preemptive
  • Interleave on uniprocessor
  • Overlap on multiprocessor

• Serial
  • One at a time, no conflict

• Cooperative
  • Yields at well-defined points
  • E.g. wait for long-running I/O

Stack Management

• Manual
  • Inherent in Cooperative
  • Changing at quiescent points

• Automatic
  • Inherent in pre-emptive
  • Downside: Hidden concurrency assumptions
UI Programming
UI Programming

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

```c
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                  LPTSTR lpCmdLine, int nCmdShow)
{
    WNDCLASSEX wc;
    HWND hwnd;
    MSG Msg;

    //Step 1: Registering the Window Class
    wc.cbSize = sizeof(WNDCLASSEX);
    wc.style = CS_HREDRAW | CS_VREDRAW;
    wc.lpFunction = DefWindowProc;
    wc.cbWndExtra = 0;
    wc.cbWindowExtra = 0;
    wc.hInstance = hInstance;
    wc.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    wc.hCursor = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
    wc.lpszMenuName = NULL;
    wc.lpszClassName = _T("_myWindowClass");
    wc.hIconSm = LoadIcon(NULL, IDI_APPLICATION);

    if(RegisterClassEx(&wc))
    {
        MessageBox(NULL, "Window Registration Failed!", "Error!",
                  MB_ICONEXCLAMATION | MB_OK);
        return 0;
    }

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

    if(hwnd == NULL)
    {
        MessageBox(NULL, "Window Creation Failed!", "Error!",
                  MB_ICONEXCLAMATION | MB_OK);
        return 0;
    }

    ShowWindow(hwnd, nCmdShow);
    UpdateWindow(hwnd);

    // Step 3: The Message Loop
    while(GetMessage(&Msg, NULL, 0, 0) > 0)
    {
        TranslateMessage(&Msg);
        DispatchMessage(&Msg);
    }
    return Msg.wParam;
}
```
UI Programming

// 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);
    return msg.wParam;
}
UI Programming

// 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);
}
UI Programming

```c
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, 
LPTSTR lpCmdLine, int nCmdShow)
{
    WNDCLASSEX wc;
    HWND hwnd;
    MSG Msg;

    //Step 1: Registering the Window Class
    wc.cbSize = sizeof(WNDCLASSEX);
    wc.style = 0;
    wc.lpfnWndProc = WinProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = hInstance;
    wc.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    wc.hCursor = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = (HRGN)(COLOR_WINDOW+1);
    wc.lpszMenuName = NULL;
    wc.lpszClassName = g_szClassName;
    wc.hIconSm = LoadIcon(NULL, IDI_APPLICATION);

    if(RegisterClassEx(&wc))
    {
        MessageBox(NULL, "Window Registration Failed!", "Error!",
        MB_ICONEXCLAMATION | MB_OK);
        return 0;
    }

    // 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);

    if(hwnd == NULL)
    {
        MessageBox(NULL, "Window Creation Failed!", "Error!",
        MB_ICONEXCLAMATION | MB_OK);
        return 0;
    }

    ShowWindow(hwnd, nCmdShow);
    UpdateWindow(hwnd);

    // Step 3: The Message Loop
    while(GetMessage(&Msg, NULL, 0, 0) > 0)
    {
        TranslateMessage(&Msg);
        DispatchMessage(&Msg);
    }
    return Msg.wParam;
}
```
UI Programming

```c
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
{
    HWND hwnd;
    MSG msg;

    //Step 1: Registering the Window Class
    wc.cbSize = sizeof(WNDCLASSEX);
    wc.style = 0;
    wc.lpfnWndProc = WndProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = hInstance;
    wc.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    wc.hCursor = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = (HRBRUSH)(COLOR_WINDOW+1);
    wc.lpszMenuName = NULL;
    wc.lpszClassName = g_szClassName;
    wc.lpszMenuName = NULL;
    wc.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    if(!RegisterClassEx(&wc))
    {
        MessageBox(NULL, "Window Registration Failed!", "Error!", MB_ICONEXCLAMATION | MB_OK);
        return 0;
    }

    //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);
    if(hwnd == NULL)
    {
        MessageBox(NULL, "Window Creation Failed!", "Error!", MB_ICONEXCLAMATION | MB_OK);
        return 0;
    }
    ShowWindow(hwnd, nCmdShow);
    UpdateWindow(hwnd);

    //Step 3: The Message Loop
    while(GetMessage(&msg, NULL, 0, 0) > 0)
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
        return msg.wParam;
    }
}
```
UI 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
        break;
    case WM_DESTROY:
        PostQuitMessage(0);
        break;
    default:
        // We do not want to handle this message so pass back to Win
        // to handle it in a default way
        return DefWindowProc(hWnd, message, wParam, lParam);
}

<table>
<thead>
<tr>
<th>Hex</th>
<th>Decimal</th>
<th>Symbolic</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>WM_NULL</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>WM_CREATE</td>
</tr>
<tr>
<td>0002</td>
<td>2</td>
<td>WM_DESTROY</td>
</tr>
<tr>
<td>0003</td>
<td>3</td>
<td>WM_MOVE</td>
</tr>
<tr>
<td>0005</td>
<td>5</td>
<td>WM_SIZE</td>
</tr>
<tr>
<td>0006</td>
<td>6</td>
<td>WM_ACTIVATE</td>
</tr>
<tr>
<td>0007</td>
<td>7</td>
<td>WM_SETFOCUS</td>
</tr>
<tr>
<td>0008</td>
<td>8</td>
<td>WM_KILLFOCUS</td>
</tr>
<tr>
<td>000a</td>
<td>10</td>
<td>WM_ENABLE</td>
</tr>
<tr>
<td>000b</td>
<td>11</td>
<td>WM_SETREDRAW</td>
</tr>
<tr>
<td>000c</td>
<td>12</td>
<td>WM_SETTEXT</td>
</tr>
<tr>
<td>000d</td>
<td>13</td>
<td>WM_GETTEXT</td>
</tr>
<tr>
<td>000e</td>
<td>14</td>
<td>WM_GETTEXTLENGTH</td>
</tr>
<tr>
<td>000f</td>
<td>15</td>
<td>WM_PAINT</td>
</tr>
<tr>
<td>0010</td>
<td>16</td>
<td>WM_CLOSE</td>
</tr>
<tr>
<td>0011</td>
<td>17</td>
<td>WM_QUERIENDSESSION</td>
</tr>
<tr>
<td>0012</td>
<td>18</td>
<td>WM_QUIT</td>
</tr>
<tr>
<td>0013</td>
<td>19</td>
<td>WM_QUERYOPEN</td>
</tr>
<tr>
<td>0014</td>
<td>20</td>
<td>WM_ERASEBKGD</td>
</tr>
</tbody>
</table>
UI 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
    //break;
    case WM_DESTROY:
        PostQuitMessage(0);
        break;
    default:
        // We do not want to handle this message so pass back to Win
        // to handle it in a default way
        return DefWindowProc(hWnd, message, wParam, lParam);
}
```

Over 1000 last time I checked!
UI programming

```c
switch (message)
{
    case WM_COMMAND:
        // handle menu select
        break;
    case WM_PAINT:
        // draw our window
        break;
    case WM_DESTROY:
        PostQuitMessage(0);
        break;
    default:
        // We do not want to handle this message so pass back to Win
        // to handle it in a default way
        return DefWindowProc(hWnd, message, wParam, lParam);
}
```

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

Over 1000 last time I checked!
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;
    }
  }
}
UI Programming Distilled

```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;
        }
    }
}
```

Pros
UI Programming Distilled

```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;
        }
    }
}
```

Pros

• Simple imperative programming
UI Programming Distilled

```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;
        }
    }
}
```

Pros

• Simple imperative programming
• Good fit for uni-processor
UI Programming Distilled

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

Cons
UI 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;
    }
  }
}
```

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

Cons
- Awkward/verbose
UI Programming Distilled

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

Cons
• Awkward/verbose
• Obscures available parallelism
UI Programming Distilled

```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;
        }
    }
}
```

**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_THIS: DoThis(); break;
        case WM_THAT: DoThat(); break;
        case WM_OTHERTHING: DoOtherThing(); break;
        case WM_DONE: return;
        }
    }
}
```
UI Programming Distilled

```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;
        }
    }
}
```
Parallel UI 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 UI Implementation 1

```cpp
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;
        }
    }
}```
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 UI Implementation 1

```c
int 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]);
}

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

DoThisProc

DoThatProc

OtherThing
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 UI 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 to CPUs?
- Load balance?!
- Utilization

**Pros/cons?**
Parallel GUI Implementation 2

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

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

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

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

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

```c
void 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();
    }
}
```

Pros/cons?
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

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

```c
threadproc(...) {
    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 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
  • ...

Events+Futures 13
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 to have no threads!*
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

```c
PROGRAM MyProgram {
    OnSize() {}
    OnMove() {}
    OnClick() {}
    OnPaint() {}
}
```
Event-based programming

```java
PROGRAM MyProgram {
    OnSize() {}
    OnMove() {}
    OnClick() {}
    OnPaint() {}
}
```
Event-based programming

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

Runtime

Thread Pool

Thread 1 Thread 2 Thread 3
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

```c
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();
}
```
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

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

```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();
}
```
Stack-Ripping

```c
PROGRAM MyProgram {
    TASK ReadFileSync(name, callback) {
        ReadFileSync(name);
        Call(callback);
    }
    CALLBACK FinishOpeningFile() {
        LoadFile(file);
        RedrawScreen();
    }
    OnOpenFile() {
        InitFileName(szName);
        EnqueueTask(ReadFileSync(szName, FinishOpeningFile));
    }
    OnPaint();
}
```
Stack-Ripping

```cpp
PROGRAM MyProgram {
    TASK ReadFileAsync(name, callback) {
        ReadFileSync(name);
        Call(callback);
    }
    CALLBACK FinishOpeningFile() {
        LoadFile()
        RedrawScreen();
    }
    OnOpenFile() {
        InitFileName(szName);
        EnqueueTask(ReadFileAsync(szName, FinishOpeningFile));
    }
    OnPaint();
}
```
Stack-Ripping

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

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
  • 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?
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

```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());
}
```
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

```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
Java Example

```java
class CompletableFutureExample {
    static void runAsyncExample() {
        CompletableFuture cf = CompletableFuture.runAsync(() -> {
            assert (Thread.currentThread().isDaemon());
            randomSleep();
            assert (!cf.isDone());
            sleepEnough();
            assert (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?

```cpp
future<int> f1 = async(foo1);
...  
int result = f1.get();
```
Futures and Promises:
Why two kinds of objects?

Promise: “thing to be done”

```java
future<int> f1 = async(foo1);
... int result = f1.get();
```
Futures and Promises:
Why two kinds of objects?

Promise: “thing to be done”

Future: encapsulation (something to give caller)

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

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

<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>
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>

- Algol
- Java
- C#/.NET
- JavaScript
- C++
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
Putting Futures in Context
My unvarnished opinion

Futures:
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*
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
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

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

• Event-based programming
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 Model between:

• Event-based programming
• Thread-based programming
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 Model between:

• Event-based programming
• Thread-based programming
Dataflow: a better abstraction?

- nodes $\rightarrow$ computation
- edges $\rightarrow$ communication
- Expresses parallelism explicitly
- Minimal specification of data movement: runtime does it.
- Asynchrony is a runtime concern (not programmer concern)
- No specification of compute $\rightarrow$ device mapping: like threads!

Events+Futures