Parallel Systems
Events and Futures

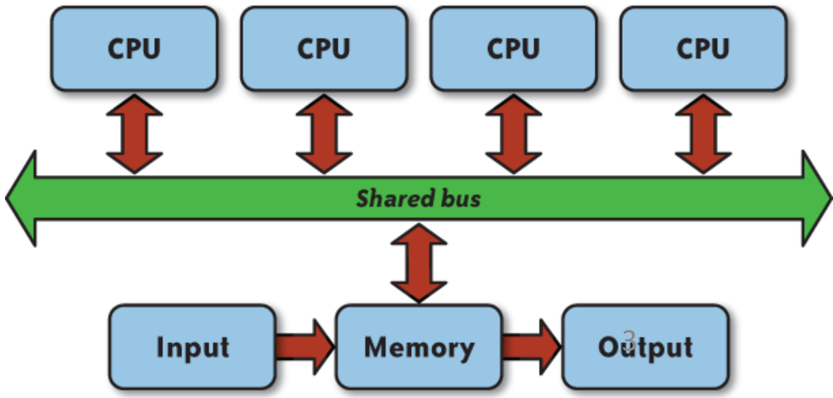
Chris Rossbach + Calvin Lin

CS380p

Outline for Today

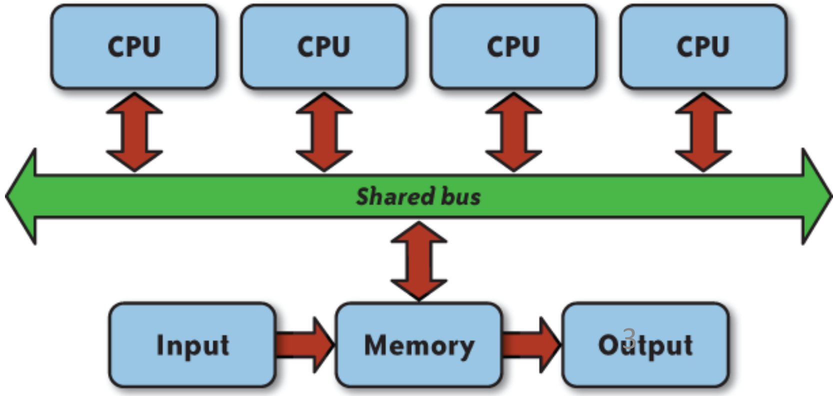
- Asynchronous Programming Models
  - Events
  - Futures

Review: Parallel Programming Models



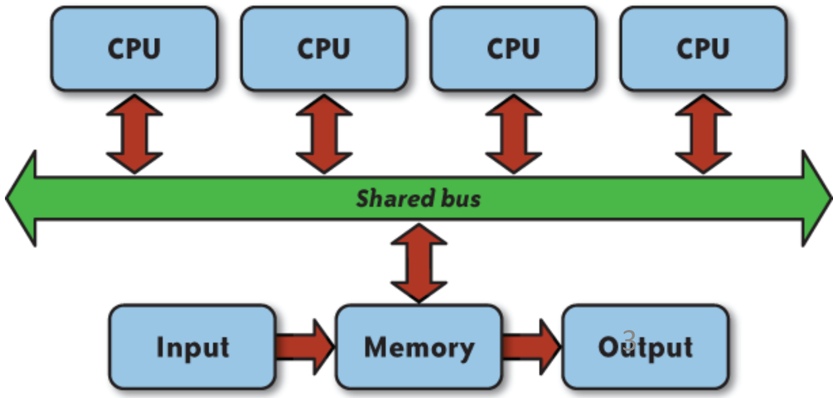
Review: Parallel Programming Models

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



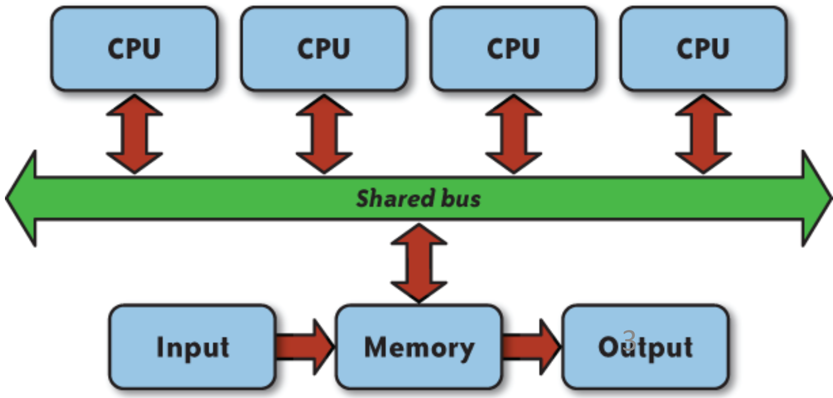
Review: Parallel Programming Models

- Concrete model:
  - CPU(s) execute instructions sequentially
- Dimensions:
  - How to specify computation
  - How to specify communication
  - How to specify coordination/control transfer



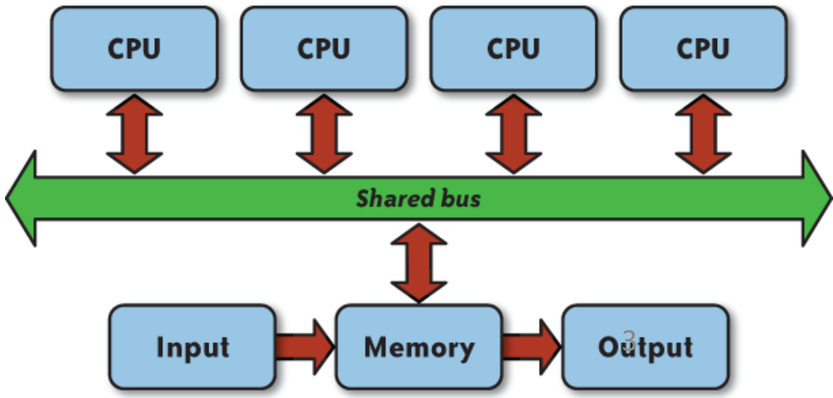
Review: Parallel Programming Models

- 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

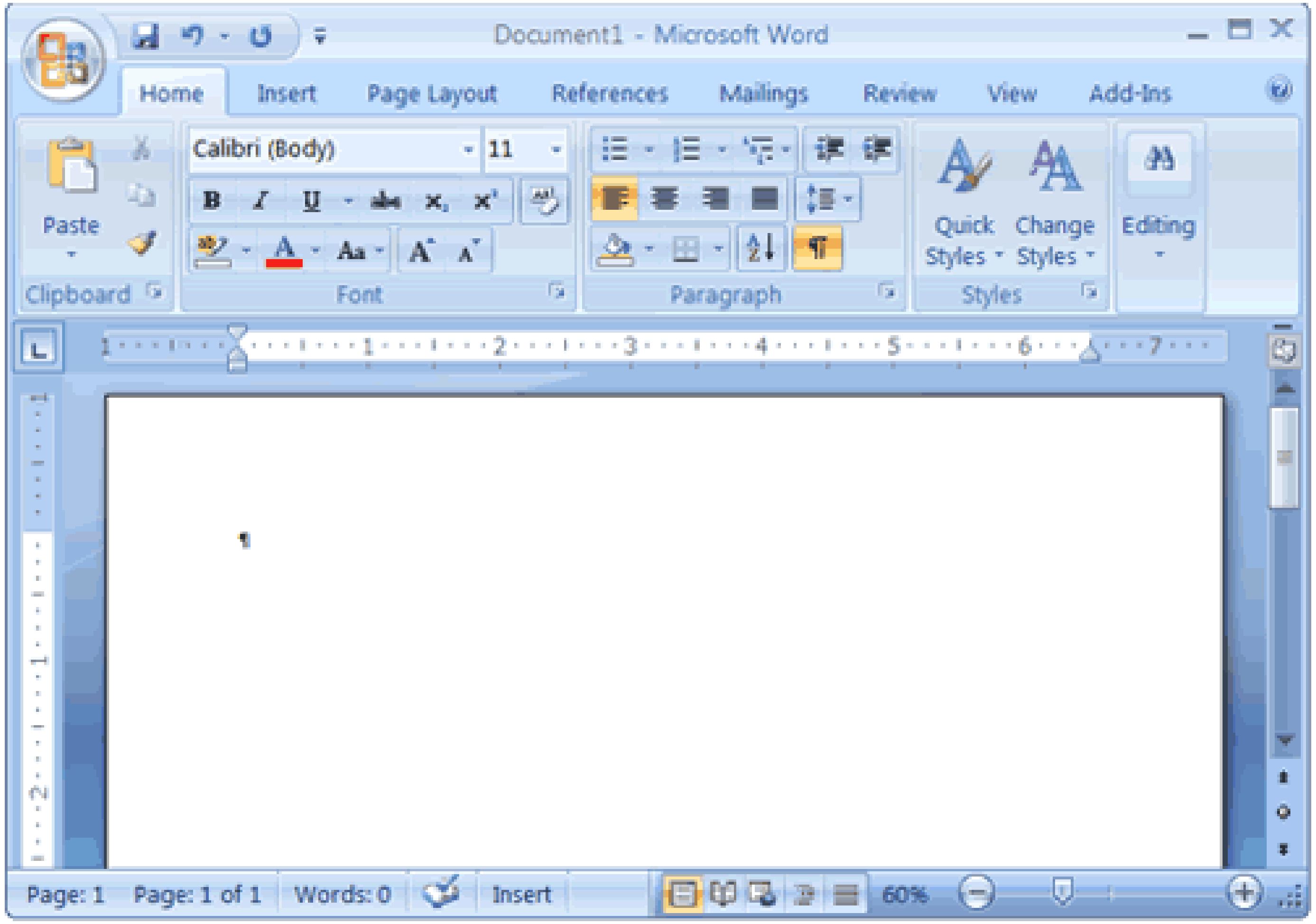
Task Management

- 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

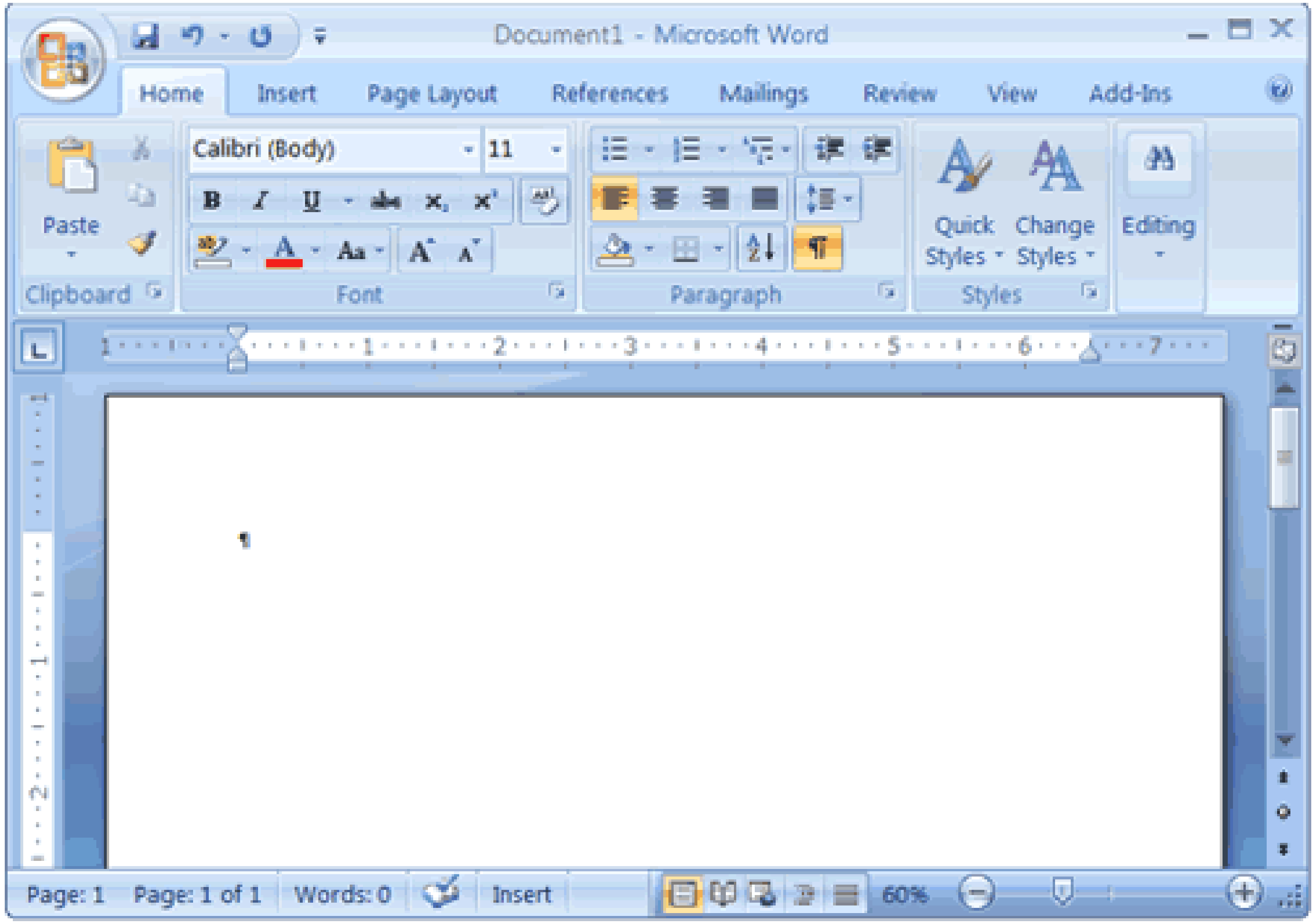
Stack Management

- Manual
  - Inherent in Cooperative
  - Changing at quiescent points
- Automatic
  - Inherent in pre-emptive
  - Downside: Hidden concurrency assumptions

UI Programming



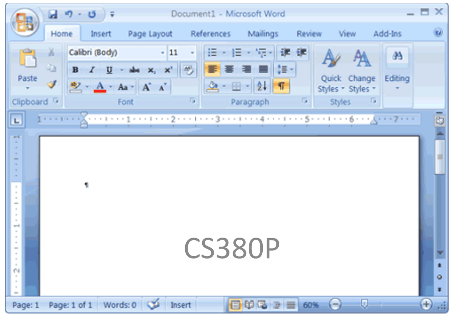
UI Programming




```
do {
  WaitForSomething();
  RespondToThing();
} until(forever);
```

UI Programming

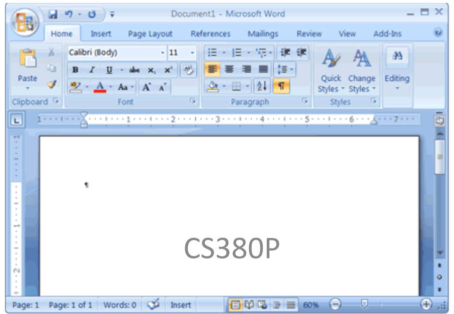
```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    WNDCLASSEX wc;
    HWND hwnd;
    MSG Msg;

    //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 = (HBRUSH)(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

```
// 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);
```

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    WNDCLASSEX wc;
    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 = (HBRUSH)(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;
}
```

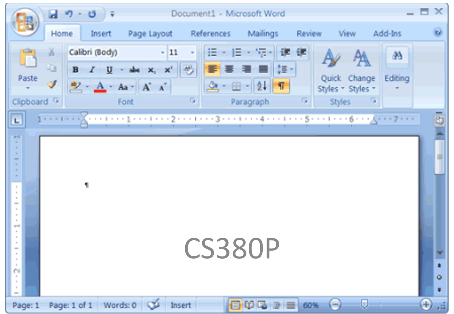


UI Programming

```
// 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);
```

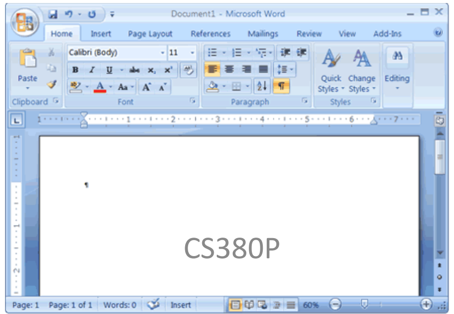
```
// Step 3: The Message Loop
while(GetMessage(&Msg, NULL, 0, 0) > 0)
{
    TranslateMessage(&Msg);
    DispatchMessage(&Msg);
}
```

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    WNDCLASSEX wc;
    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 = (HBRUSH)(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;
}
```



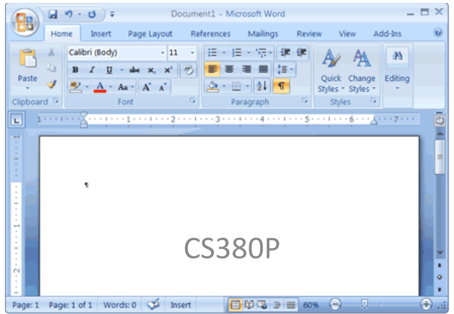
UI Programming

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    WNDCLASSEX wc;
    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 = (HBRUSH)(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

```
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);
}
```



```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    WNDCLASSEX wc;
    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 = (HBRUSH)(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;
}
```

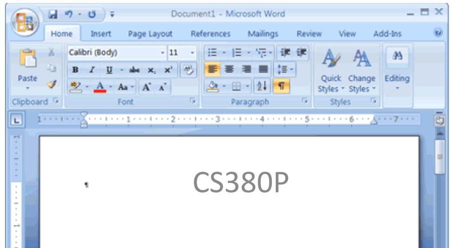
```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    WNDCLASSEX wc;
    HWND hwnd;
    MSG Msg;
```

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 Winc
            // to handle it in a default way
            return DefWindowProc(hWnd, message, wParam, lParam);
}
```

Hex	Decimal	Symbolic
0000	0	WM_NULL
0001	1	WM_CREATE
0002	2	WM_DESTROY
0003	3	WM_MOVE
0005	5	WM_SIZE
0006	6	WM_ACTIVATE
0007	7	WM_SETFOCUS
0008	8	WM_KILLFOCUS
000a	10	WM_ENABLE
000b	11	WM_SETREDRAW
000c	12	WM_SETTEXT
000d	13	WM_GETTEXT
000e	14	WM_GETTEXTLENGTH
000f	15	WM_PAINT
0010	16	WM_CLOSE
0011	17	WM_QUERYENDSESSION
0012	18	WM_QUIT
0013	19	WM_QUERYOPEN
0014	20	WM_ERASEBKGND

```
ION);
ION);
ed!", "Error!",
, "Error!",
        TranslateMessage(&Msg);
        DispatchMessage(&Msg);
    }
    return Msg.wParam;
}
```



UI programming

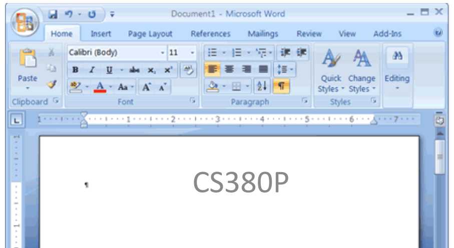
```
switch (message)
{
        //case WM_COMMAND:
          // handle menu selections etc.
        //break;
        //case WM_PAINT:
          // draw our window - note: you must paint something here or not
        //break;
        case WM_DESTROY:
             PostQuitMessage(0);
        break;
        default:
            // We do not want to handle this message so pass back to Winc
            // to handle it in a default way
            return DefWindowProc(hWnd, message, wParam, lParam);
}
```

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

Hex	Decimal	Symbolic
0000	0	WM_NULL
0001	1	WM_CREATE
0002	2	WM_DESTROY
0003	3	WM_MOVE
0005	5	WM_SIZE
0006	6	WM_ACTIVATE
0007	7	WM_SETFOCUS
0008	8	WM_KILLFOCUS
000a	10	WM_ENABLE
000b	11	WM_SETREDRAW
000c	12	WM_SETTEXT
000d	13	WM_GETTEXT
000e	14	WM_GETTEXTLENGTH
000f	15	WM_PAINT
0010	16	WM_CLOSE
0011	17	WM_QUERYENDSESSION
0012	18	WM_QUIT
0013	19	WM_QUERYOPEN
0014	20	WM_ERASEBKGND

Over 1000 last time I checked!

```
ION);
ION);
ed!", "Error!",
 "Error!",
        TranslateMessage(&Msg);
        DispatchMessage(&Msg);
    }
    return Msg.wParam;
}
```



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

UI programming

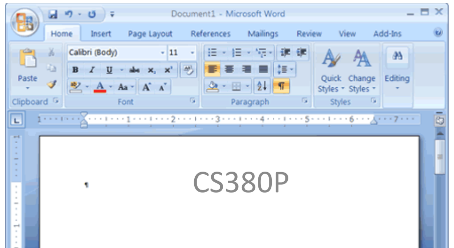
Over 1000 last time I checked!

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

```
void OnMove() { … }
void OnSize() { … }

void OnPaint() { … }
```

Hex	Decimal	Symbolic
0000	0	WM_NULL
0001	1	WM_CREATE
0002	2	WM_DESTROY
0003	3	WM_MOVE
0005	5	WM_SIZE
0006	6	WM_ACTIVATE
0007	7	WM_SETFOCUS
0008	8	WM_KILLFOCUS
000a	10	WM_ENABLE
000b	11	WM_SETREDRAW
000c	12	WM_SETTEXT
000d	13	WM_GETTEXT
000e	14	WM_GETTEXTLENGTH
000f	15	WM_PAINT
0010	16	WM_CLOSE
0011	17	WM_QUERYENDSESSION
0012	18	WM_QUIT
0013	19	WM_QUERYOPEN
0014	20	WM_ERASEBKGND

```
ION);

ION);

ed!", "Error!",

, "Error!",

        TranslateMessage(&Msg);
        DispatchMessage(&Msg);
    }
    return Msg.wParam;
}
```



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;
      }
  }
}
```

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

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

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

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

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

```
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
- **Obscures available parallelism**

UI Programming Distilled

```
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**

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;
      }
  }
}
```

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;
    }
  }
}
```

How can we parallelize this?

Parallel UI Implementation 1

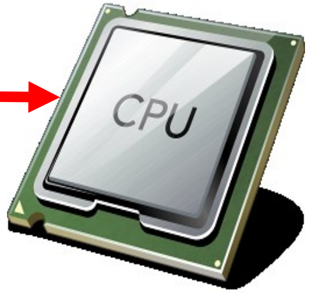
```
winmain(...) {
  while(true) {
        message = GetMessage();
        switch(message) {
        case WM_THIS: DoThis(); break;
        case WM_THAT: DoThat(); break;
        case WM_OTHERTHING: DoOtherThing(); break;
        case WM_DONE: return;
        }
  }
}
```

Parallel UI Implementation 1

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

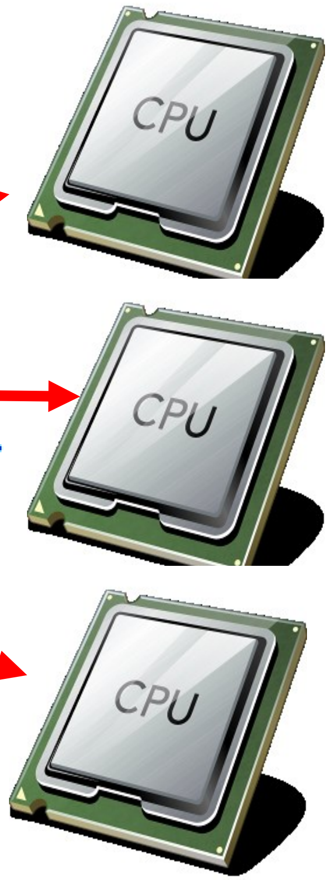
Parallel UI Implementation 1

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



Parallel UI Implementation 1

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



Parallel UI 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

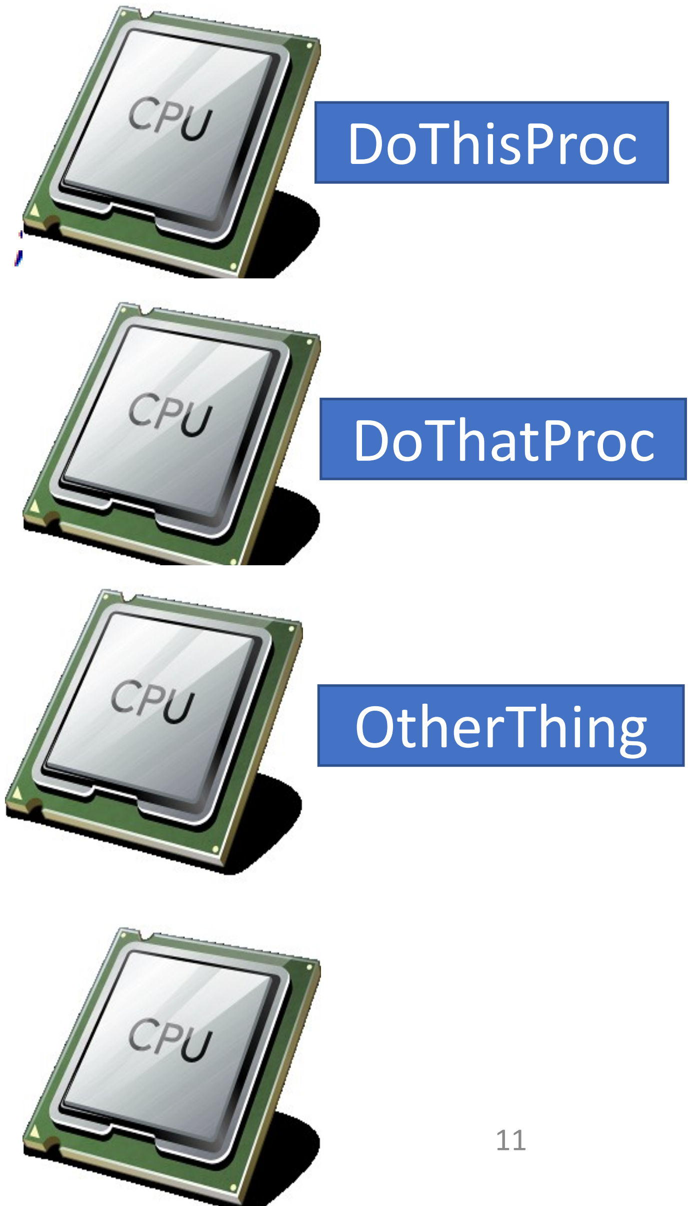
DoThatProc

OtherThing

Parallel UI Implementation 1

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

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

Pros/cons?



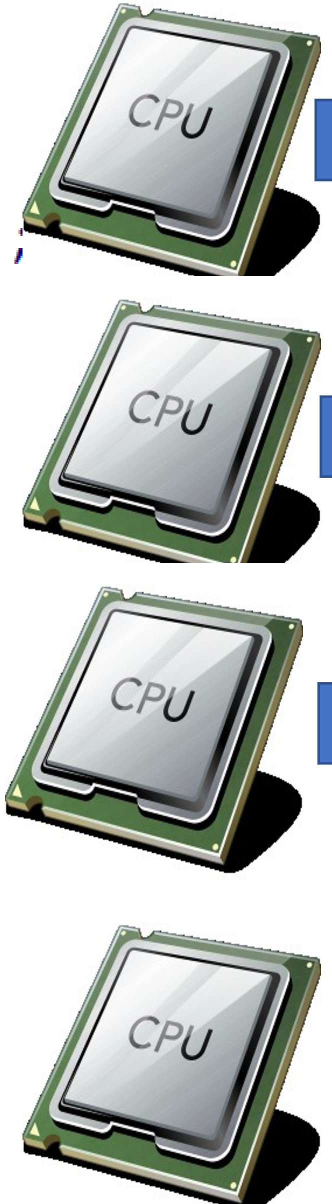
Parallel UI Implementation 1

```
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

Pros/cons?

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

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 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

```
switch (message)
{
        //case WM_COMMAND:
          // handle menu selections etc.
        //break;
        //case WM_PAINT:
          // draw our window - note: you must paint something here or not trap it!
        //break;
        case WM_DESTROY:
             PostQuitMessage(0);
        break;
        default:
            // We do not want to handle this message so pass back to Windows
            // to handle it in a default way
            return DefWindowProc(hWnd, message, wParam, lParam);
}
```

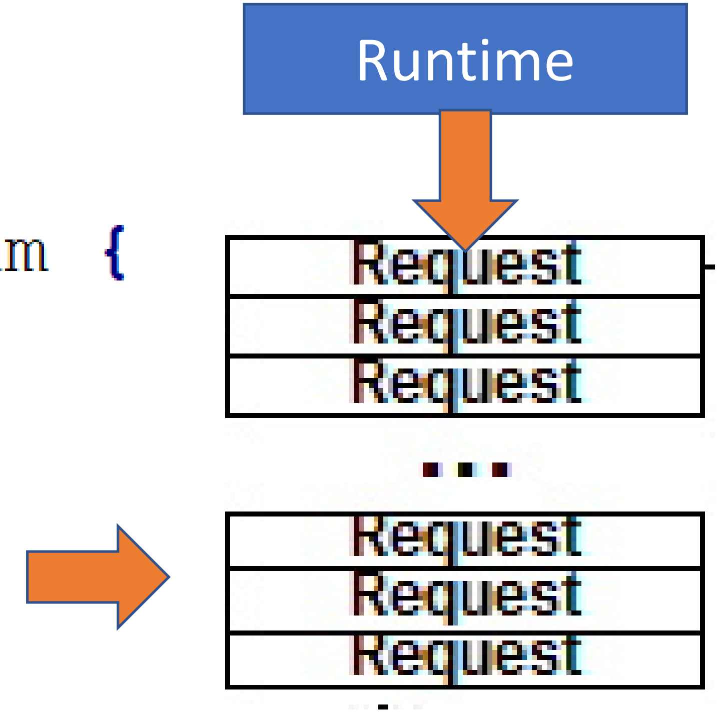
Event-based programming

Event-based programming

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

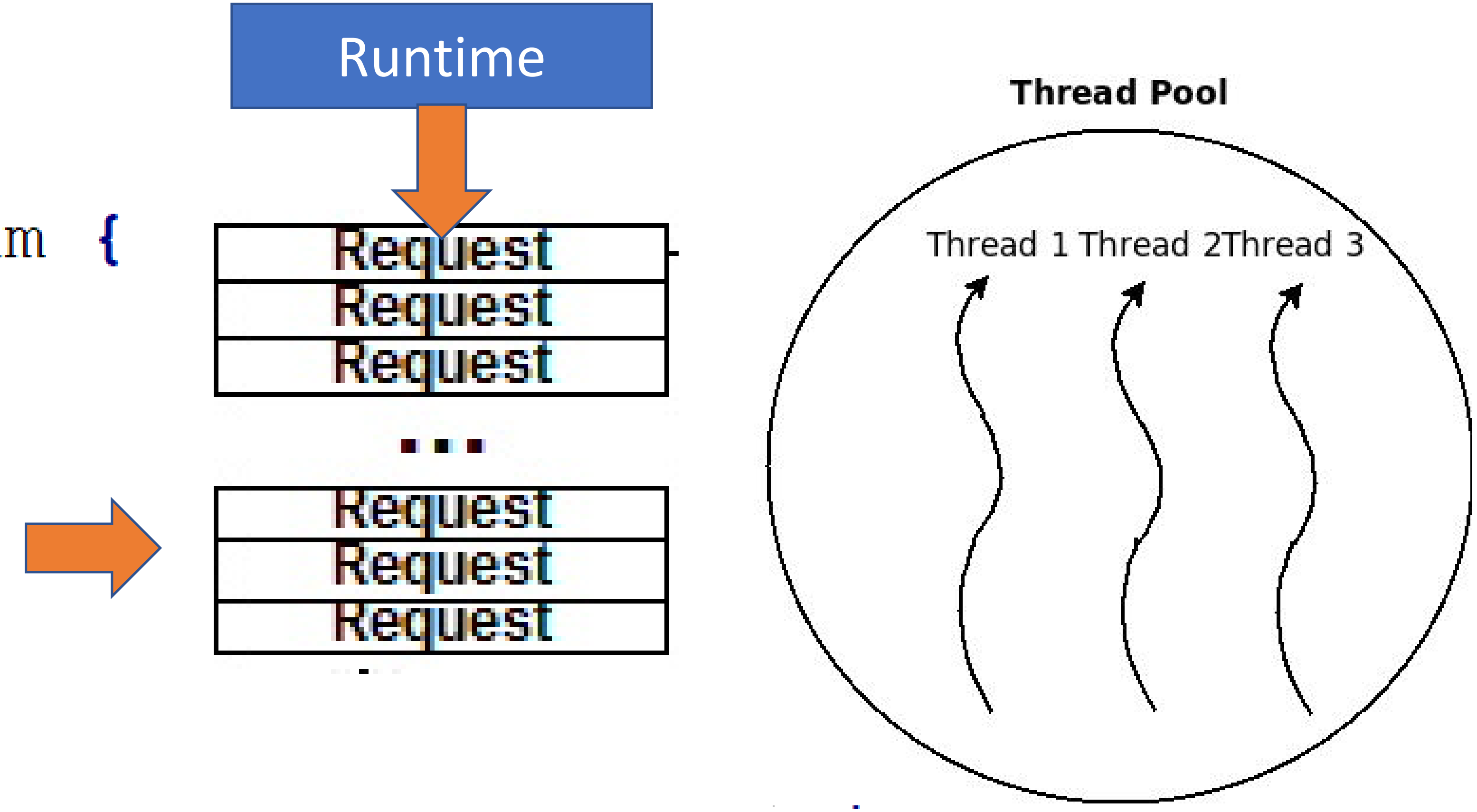
Event-based programming

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



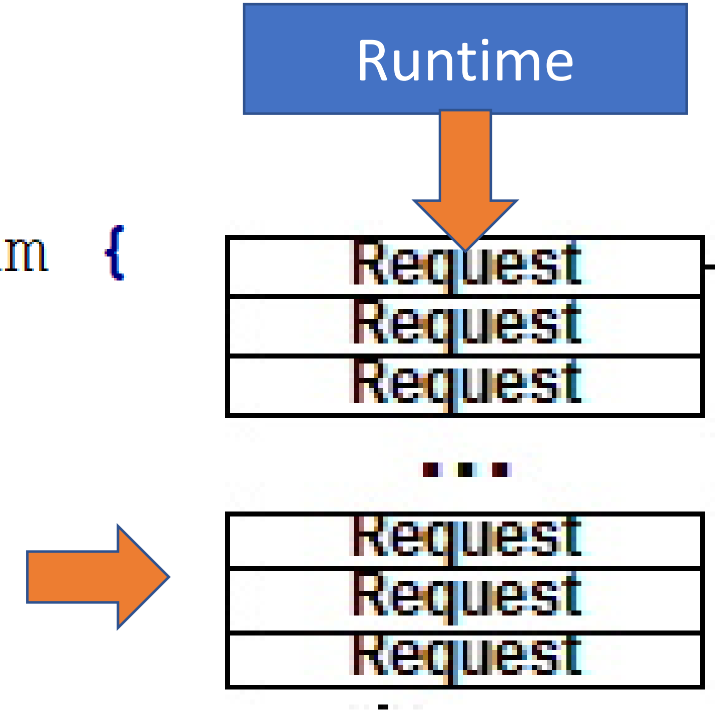
Event-based programming

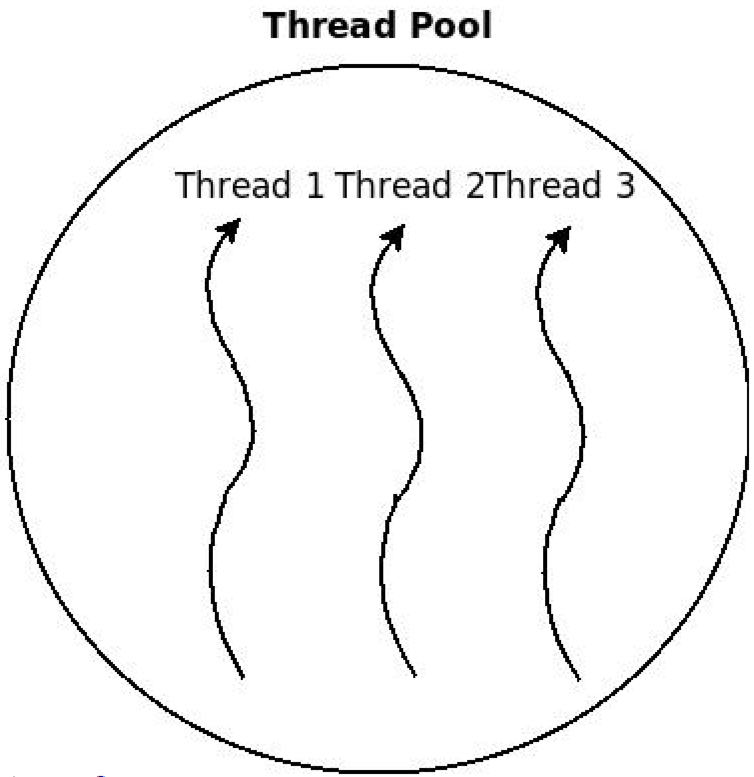
```
PROGRAM MyProgram {
    OnSize()  {}
    OnMove()  {}
    OnClick()  {}
    OnPaint()  {}
}
```



Event-based programming

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





Is the problem solved?

Another Event-based Program

Another Event-based Program

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

Another Event-based Program

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

Blocks!

Another Event-based Program

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

Burns CPU!

Blocks!

Another Event-based Program

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

Uses Other Handlers!
(call OnPaint?)

Burns CPU!

Blocks!

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

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

Continuations, BTW

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

Stack-Ripping

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

Stack-Ripping

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


        InitFileName(szName);
        EnqueueTask(ReadFileAsync(szName, FinishOpeningFile));
    }
    OnPaint();
}
```

Stack-Ripping

```
PROGRAM MyProgram {
    TASK ReadFileAsync(name, callback) {
        ReadFileSync(name);
        Call(callback);
    }
    CALLBACK FinishOpeningFile() {
        LoadFil
        RedrawScreen();
    }
    OnOpenFile() {


        InitFileName(szName);
        EnqueueTask(ReadFileAsync(szName, FinishOpeningFile));
    }
    OnPaint();
}
```

Stack-Ripping

```
PROGRAM MyProgram {
    TASK ReadFileAsync(name, callback) {
        ReadFileSync(name);
        Call(callback);
    }
    CALLBACK FinishOpeningFile() {
        LoadFil
        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
- Thread Cons
  - Synchronization requir
  - Overflowable stack
  - Stack memory pressure

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

Language-level Futures: the sweet spot?

Threads vs Events

- Thread Pros
  - Overlap I/O and computation
    - While looking sequential
  - Intermediate state on stack
  - Control flow naturally
- Thread Cons
  - Synchronization 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
  - hen to deallocate memory
  - Incomplete language/tool/debugger support
  - Difficult to exploit concurrent hardware

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

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

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

Java Example

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

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

Java Example

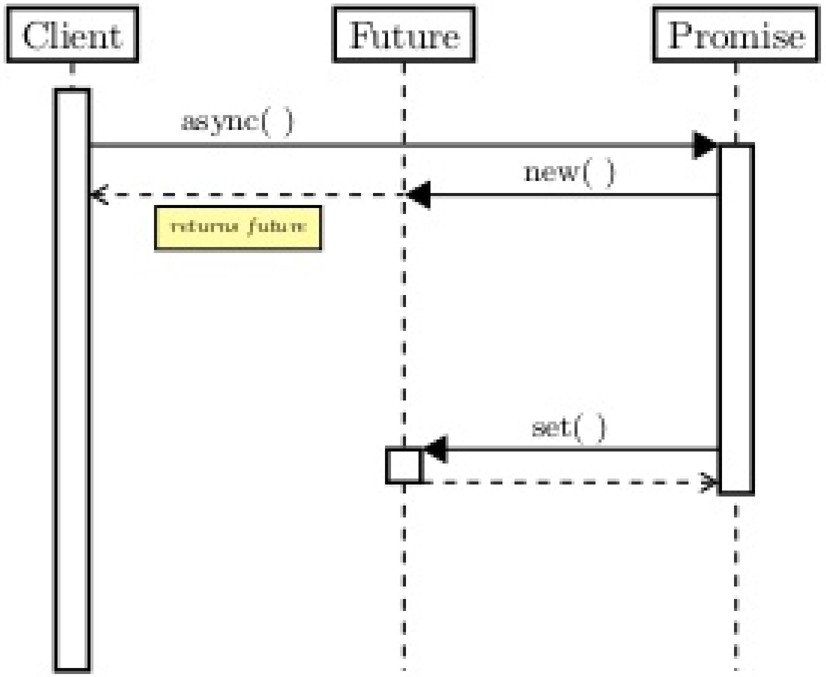
```java
static void runAsyncExample() {
    CompletableFuture cf = CompletableFuture.runAsync(() -> {
        assertTrue(Thread.currentThread().isDaemon());
        randomSleep();
    });
    assertFalse(cf.isDone());
    sleepEnough();
    assertTrue(cf.isDone());
}
```

- CompletableFuture is a container for Future object type
- cf is an instance
- runAsync() accepts
  - Lambda expression
  - Anonymous function
  - Functor
- runAsync() immediately returns a waitable object (cf)
- Where (on what thread) does the lambda expression run?

Futures and Promises:

Why two kinds of objects?

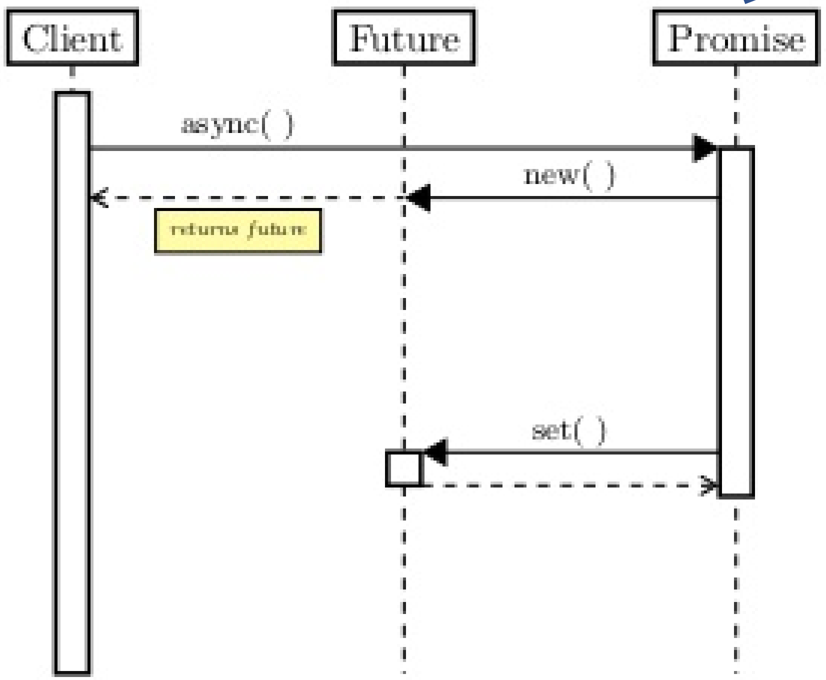
```
future<int> f1 = async(foo1);
...
int result = f1.get();
```



Futures and Promises:
Why two kinds of objects?

Promise: “thing to be done”

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

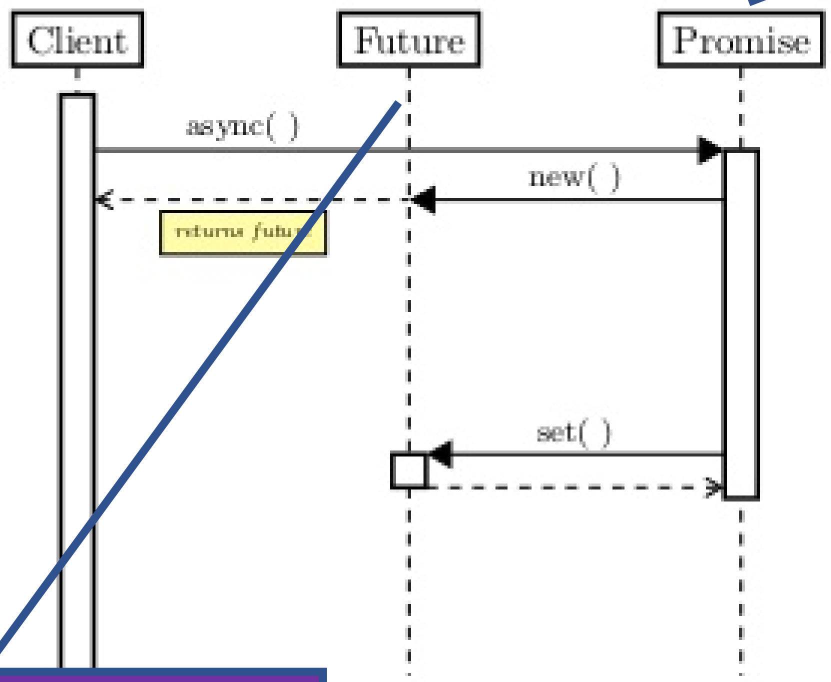


Futures and Promises:

Why two kinds of objects?

Promise: “thing to be done”

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

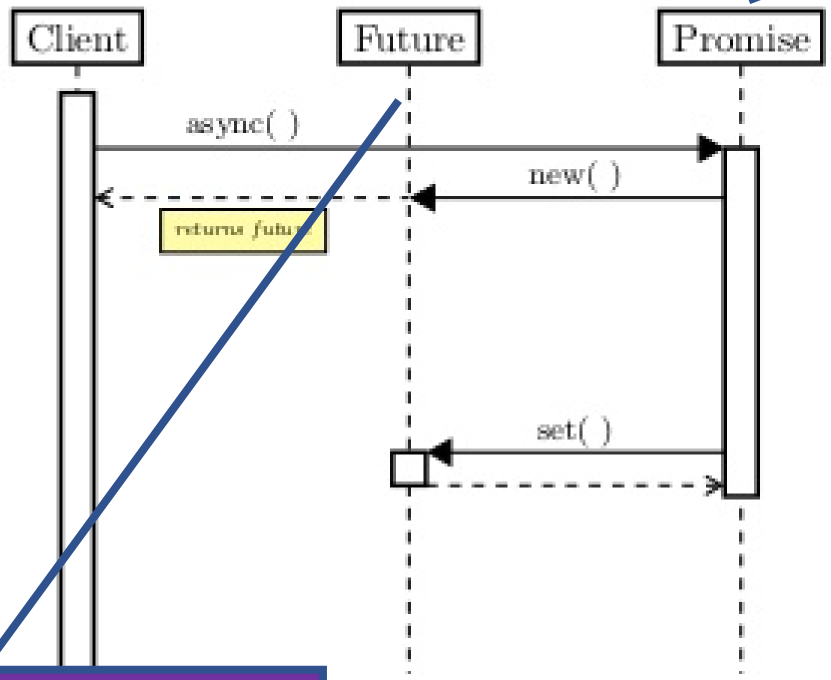


Future: encapsulation
(something to give caller)

Futures and Promises:

Why two kinds of objects?

Promise: “thing to be done”



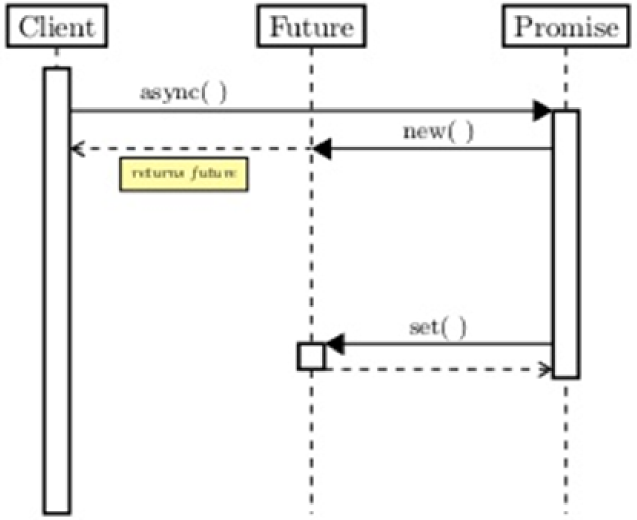

```
future<int> f1 = async(foo1);
...
int result = f1.get();
```

Future: encapsulation
(something to give caller)

Promise to do something in the future

Futures vs Promises

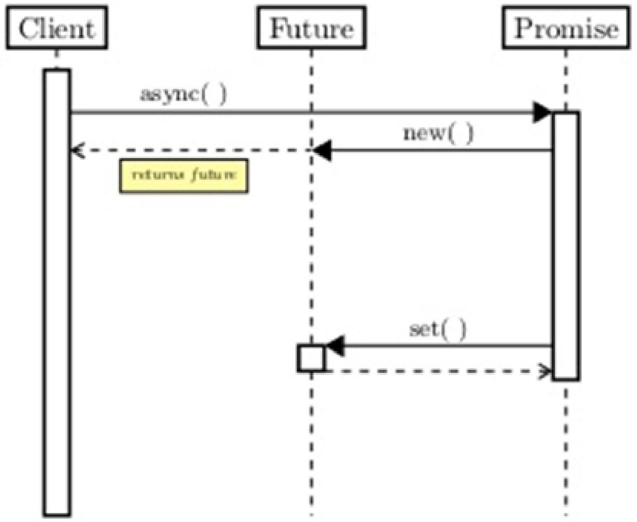
- **Future:** read-only reference to uncompleted value
- **Promise:** single-assignment variable that the future refers to
- Promises *complete* the future with:
  - Result with success/failure
  - Exception



Futures vs Promises

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

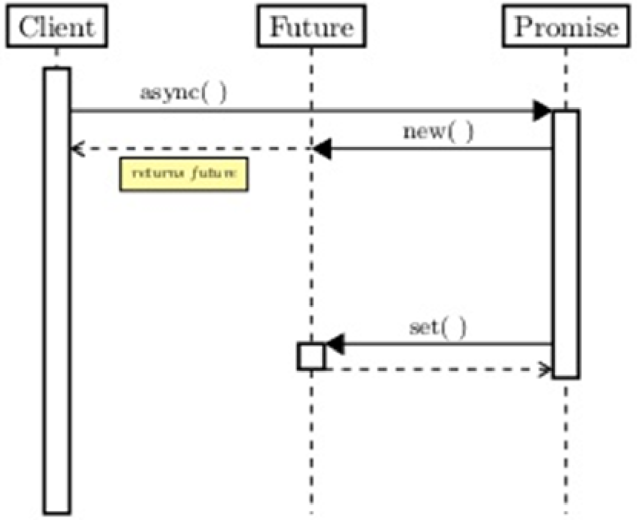
Language	Promise	Future
Algol	Thunk	Address of async result
Java	CompletableFuture<T>	Future<T>
C#/.NET	TaskCompletionSource<T>	Task<T>
JavaScript	Deferred	Promise
C++	std::promise	std::future



Futures vs Promises

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

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

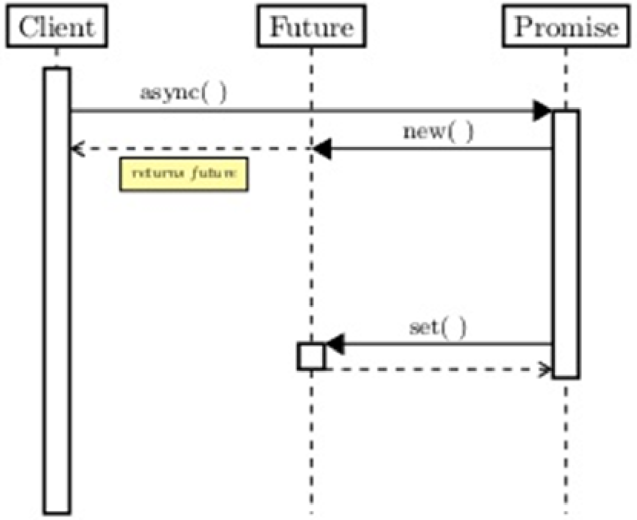


Futures vs Promises

Mnemonic:
Promise to ***do*** something
Make a promise ***for*** the future

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

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

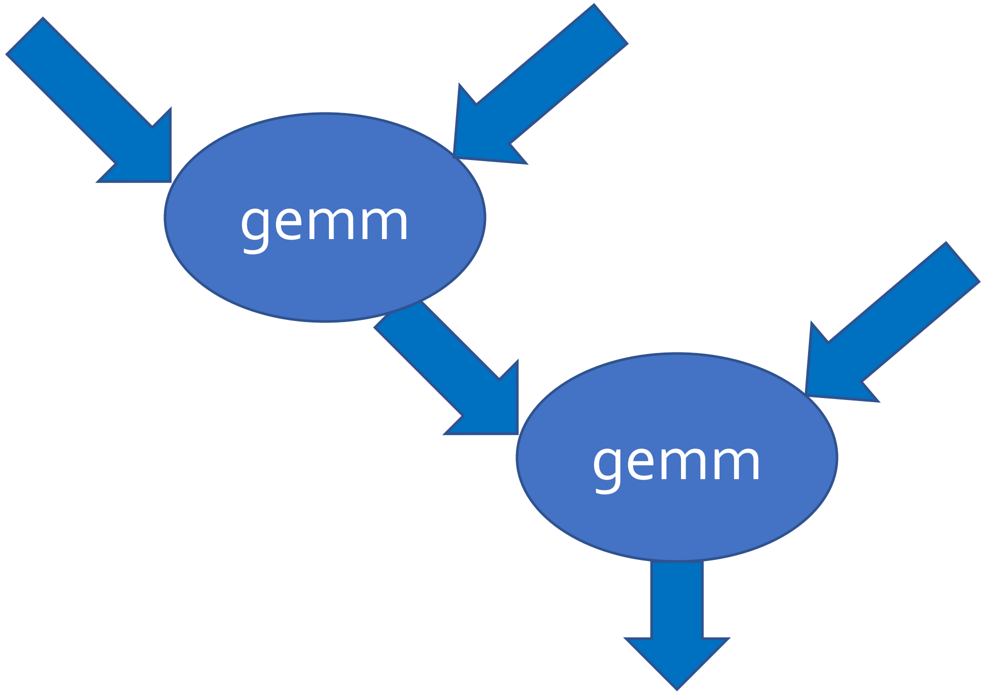
- *abstraction* for concurrent work supported by
  - Compiler: abstractions are ***language-level objects***
  - Runtime: scheduler, task queues, thread-pools are ***transparent***
- Programming remains **mostly** imperative
  - Threads of control peppered with asynchronous/concurrent tasks

Compromise Programming Model between:

- Event-based programming
- Thread-based programming

Events vs. Threads!

Dataflow: a better abstraction?



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