Synchronization: Implementing Barriers Promises + Futures

Chris Rossbach

CS378H

Today

- Questions?
- Administrivia
  - Lab 2 due sooner than you'd like
- Material for the day
  - Barrier implementation
  - Promises & Futures

- Acknowledgements
  - Thanks to Gadi Taubenfield: I borrowed from some of his slides on barriers

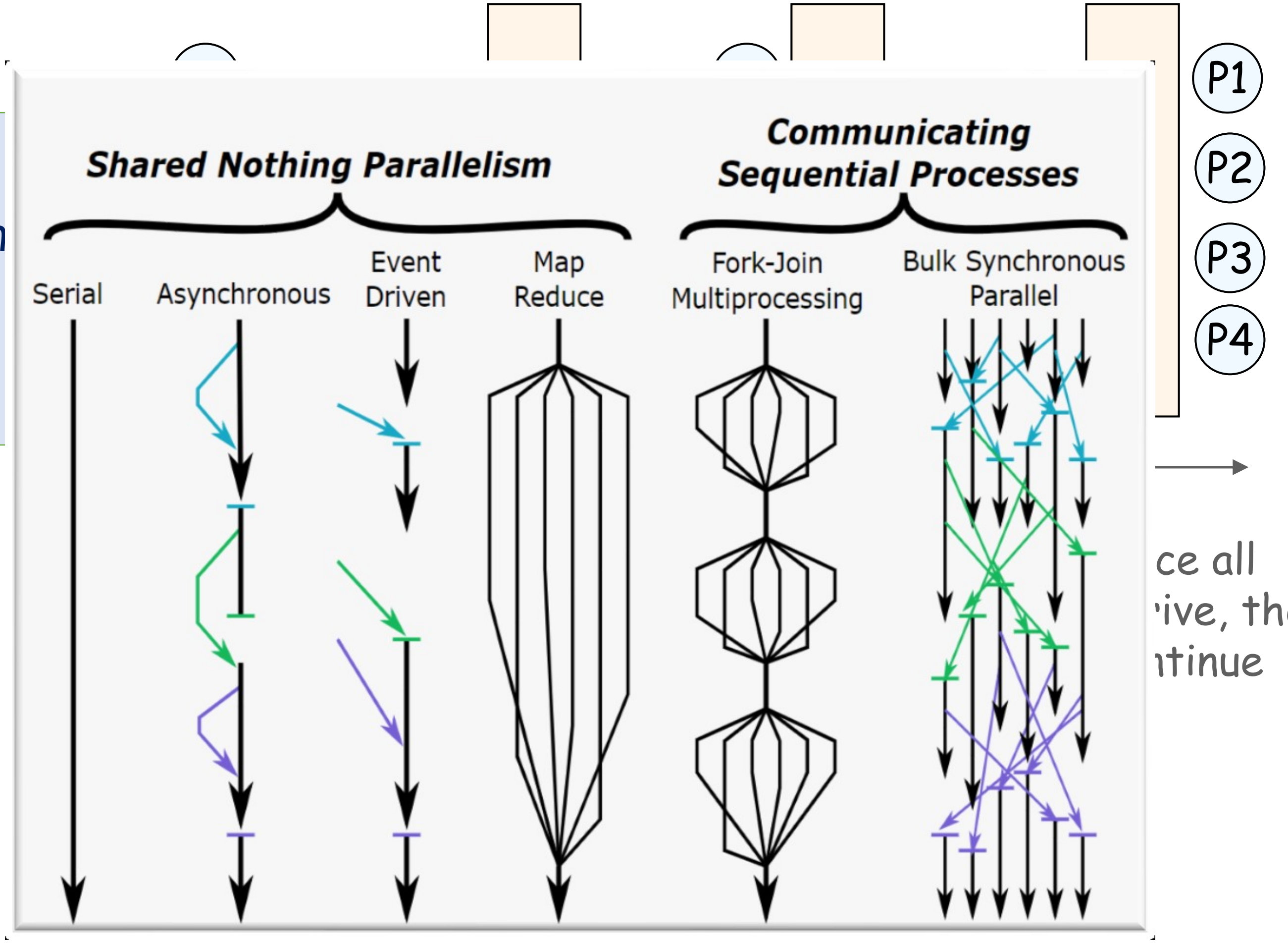
Faux Quiz (answer any N, 5 min)

- How are promises and futures related? Since there is disagreement on the nomenclature, don’t worry about which is which—just describe what the different objects are and how they function.

Barriers

Review: Barrier Basics

- *Coordination mechanism*
- *participants wait until all reach same poin*
- *Once all reach it, all can pass.*
- ***Workhorse of BSP programming models***



ce all
ive, they
ntinue

Review: Barrier API

```
Wait();
Init/Destroy();
```

Type `pthread_barrier_t`

```
int pthread_barrier_init(pthread_barrier_t *barrier,
                         const pthread_barrierattr_t *attr,
                         unsigned count);
int pthread_barrier_destroy(pthread_barrier_t *barrier);
int pthread_barrier_wait(pthread_barrier_t *barrier);
```

Barriers: Goals

Desirable barrier properties:

- Low shared memory space complexity
- Low contention on shared objects
- Low shared memory references per process
- No need for shared memory initialization
- Symmetric: same amount of work for all
- Algorithm simplicity
- Simple basic primitive
- Minimal propagation time
- Reusability of the barrier (must!)

Let’s build a Barrier

(woot!)

```
class Barrier {

        initialize() {

        }

        wait() {

        }

};
```

Barrier Building Blocks

- Conditions
- Semaphores
- Atomic Bit
- Atomic Register
- Fetch-and-increment register
- Test and set bits
- Read-Modify-Write register

Barrier with Semaphores


Barrier using Semaphores

Algorithm for N threads

```
shared    sem_t arrival = 1;           // sem_init(&arrival, NULL, 1)
          sem_t departure = 0;         // sem_init(&departure, NULL, 0)
          atomic int counter = 0;      // (gcc intrinsics are verbose)
```

```
type __sync_fetch_and_add (type *ptr, type value, ...)
type __sync_fetch_and_sub (type *ptr, type value, ...)
type __sync_fetch_and_or (type *ptr, type value, ...)
type __sync_fetch_and_and (type *ptr, type value, ...)
type __sync_fetch_and_xor (type *ptr, type value, ...)
type __sync_fetch_and_nand (type *ptr, type value, ...)
```

Phase I

```
sem_wait(arrival);
if(++counter < N)
  sem_post(arrival);
else
  sem_post(departure);
```

Phase II

First N-1 threa[...]
arrival, wait on

Nth thread post on departure, releasing threads into phase II (what is value of arrival?)

First N-1 threads post on departure, last posts arrival

Semaphore Barrier Action Zone

N == 3

```
shared   sem_t arrival = 0
         sem_t departure = 0
         atomic int counter = 0
```

CPU 0

```
sem_wait(arrival);
if(++counter < N)
  sem_post(arrival);
else
  sem_post(departure);
sem_wait(departure);
if(--counter > 0)
  sem_post(departure)
else
  sem_post(arrival)
```

CPU 1

```
sem_wait(arrival);
if(++counter < N)
  sem_post(arrival);
else
  sem_post(departure);
sem_wait(departure);
if(--counter > 0)
  sem_post(departure)
else
  sem_post(arrival)
```

CPU 2

```
sem_wait(a
if(++counte
  sem_pos
else
  sem_post(departure);
sem_wait(departure);
if(--counter > 0)
  sem_post(departure)
else
  sem_post(arrival)
```

```
// why two phases:
for(…) {
    do_something();
    wait();
}
```

Do we need two phases?

Still correct if counter is not atomic?

Barrier using Semaphores

Properties

- Pros:
  - Very Simple
  - Space complexity O(1)
  - Symmetric
- Cons:
  - Required a strong object
    - Requires some central manager
    - High contention on the semaphores
  - Propagation delay O(n)

Barriers based on counters

Counter Barrier Ingredients

Fetch-and-Increment register

- A shared register that supports a F&I operation:
- Input: register *r*
- Atomic operation:
  - *r* is incremented by 1
  - the old value of r is returned

```
function fetch-and-increment (r : register)
        orig_r := r;
        r:= r + 1;
        return (orig_r);
end-function
```

Await

- For brevity, we use the **await** macro
- Not an operation of an object
- This is just “spinning”

```
macro await (condition : boolean condition)
        repeat
                cond = eval(condition);
        until (cond)
end-macro
```

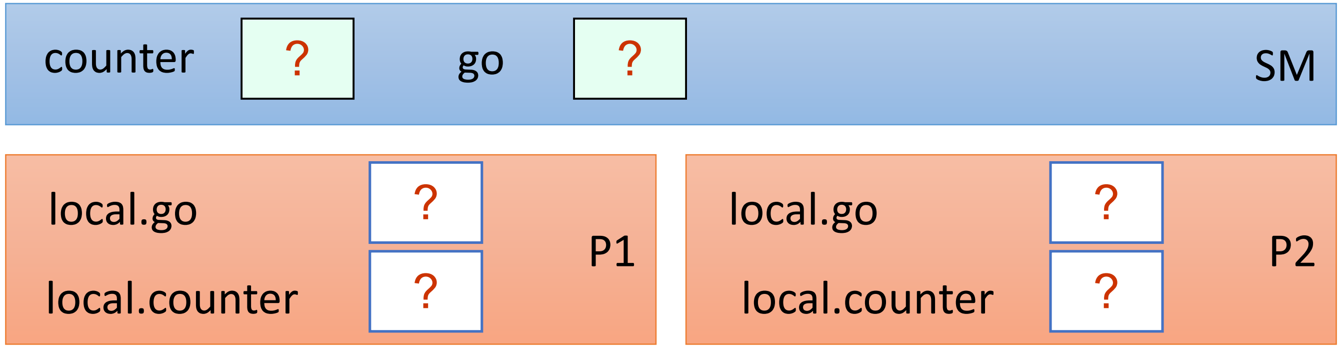
Simple Barrier Using an Atomic Counter

```
shared    counter: fetch and increment reg. – {0,..n}, initially = 0
          go: atomic bit, initial value does not matter
local     local.go: a bit, initial value  does not matter
          local.counter: register
```

```
local.go := go
local.counter := fetch-and-increment (counter)
if local.counter + 1 = n then
        counter := 0
        go := 1 - go
else await(local.go ≠ go)
```

Simple Barrier Using an Atomic Counter

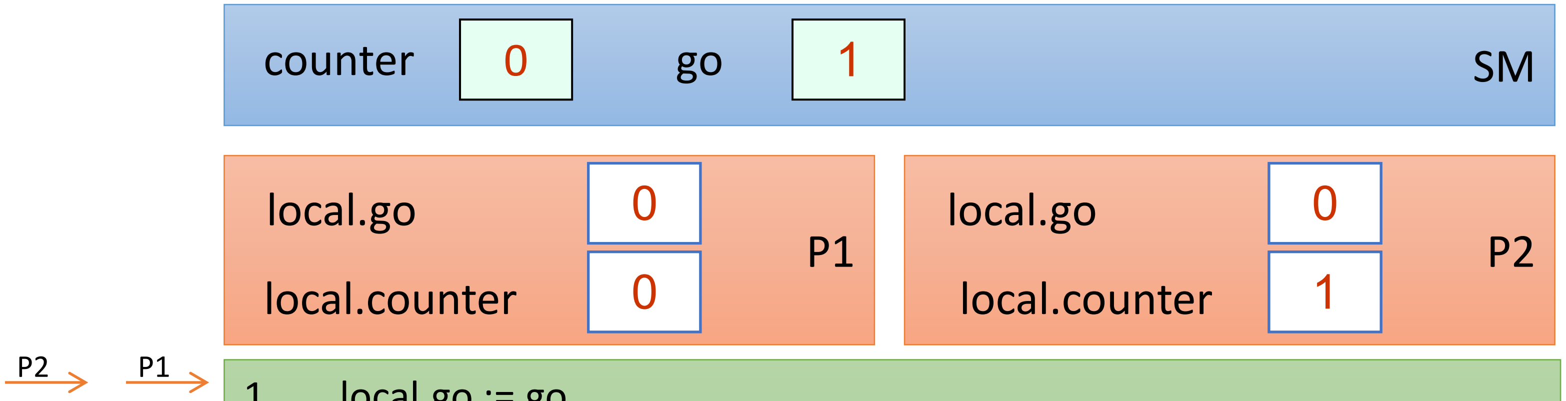
Run for n=2 Threads




```
local.go := go
local.counter := fetch-and-increment (counter)
if local.counter + 1 = n then
        counter := 0
        go := 1 - go
else await(local.go ≠ go)
```

Simple Barrier Using an Atomic Counter

Run for n=2 Threads




```
local.go := go
local.counter := fetch-and-increment
if local.counter + 1 = n then
        counter := 0
        go := 1 - go
else await(local.go ≠ go)
```

1+1=2

P1 Busy wait

Pros/Cons?

- There is high memory contention on *go* bit
- Reducing the contention:
  - Replace the *go* bit with *n* bits: *go*[1],…,*go*[*n*]
  - Process p_i may spin only on the bit *go*[*i*]

A Local Spinning Counter Barrier

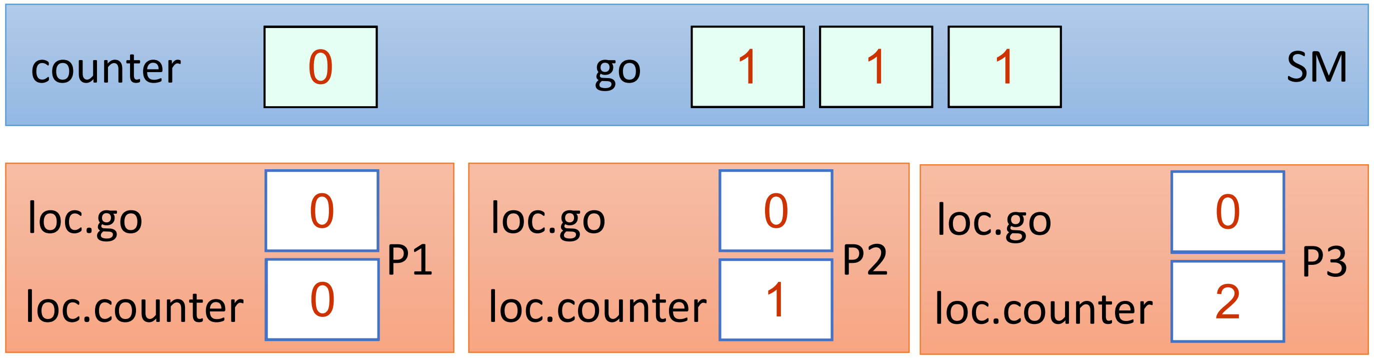
Program of a Thread i

shared	counter: fetch and increment reg. – {0,..n}, initially = 0
	go[1..n]: array of atomic bits, initial values are immaterial
local	local.go: a bit, initial value is immaterial
	local.counter: register

```
local.go := go[i]
local.counter := fetch-and-increment (counter)
if local.counter + 1 = n then
        counter := 0
        for j=1 to n { go[j] := 1 – go[j] }
else await(local.go ≠ go[i])
```

A Local Spinning Counter Barrier

Example Run for n=3 Threads



P3 → P2 → P1 →

```
local.go := go[i]
local.counter := fetch-and-increment(
if local.counter + 1 = n then
        counter := 0
        for j=1 to n { go[j] := 1 – go[j] }
else await(local.go ≠ go[i])
```

2+1=3

P1,P2 Busy wait

Pros/Cons?

Does this actually reduce contention?

Comparison of counter-based Barriers

Simple Barrier

- Pros:
- Cons:

Simple Barrier with go array

- Pros:
- Cons:

Comparison of counter-based Barriers

Simple Barrier

- Pros:
  - Very Simple
  - Shared memory: O(log n) ***bits***
  - Takes O(1) until last waiting p is awaken
- Cons:
  - High contention on the go bit
  - Contention on the counter register (*)

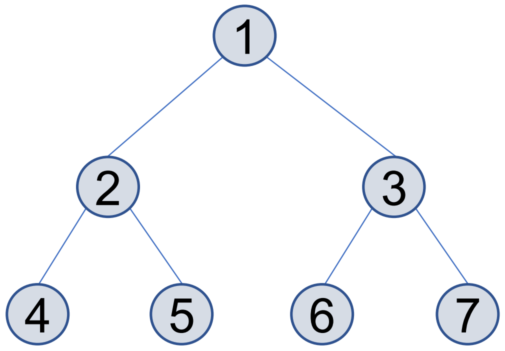
Simple Barrier with go array

- Pros:
  - Low contention on the go array
  - In some models:
    - spinning is done on local memory
    - remote mem. ref.: O(1)
- Cons:
  - Shared memory: O(n)
  - Still contention on the counter register (*)
  - Takes O(n) until last waiting p is awaken

Tree Barriers

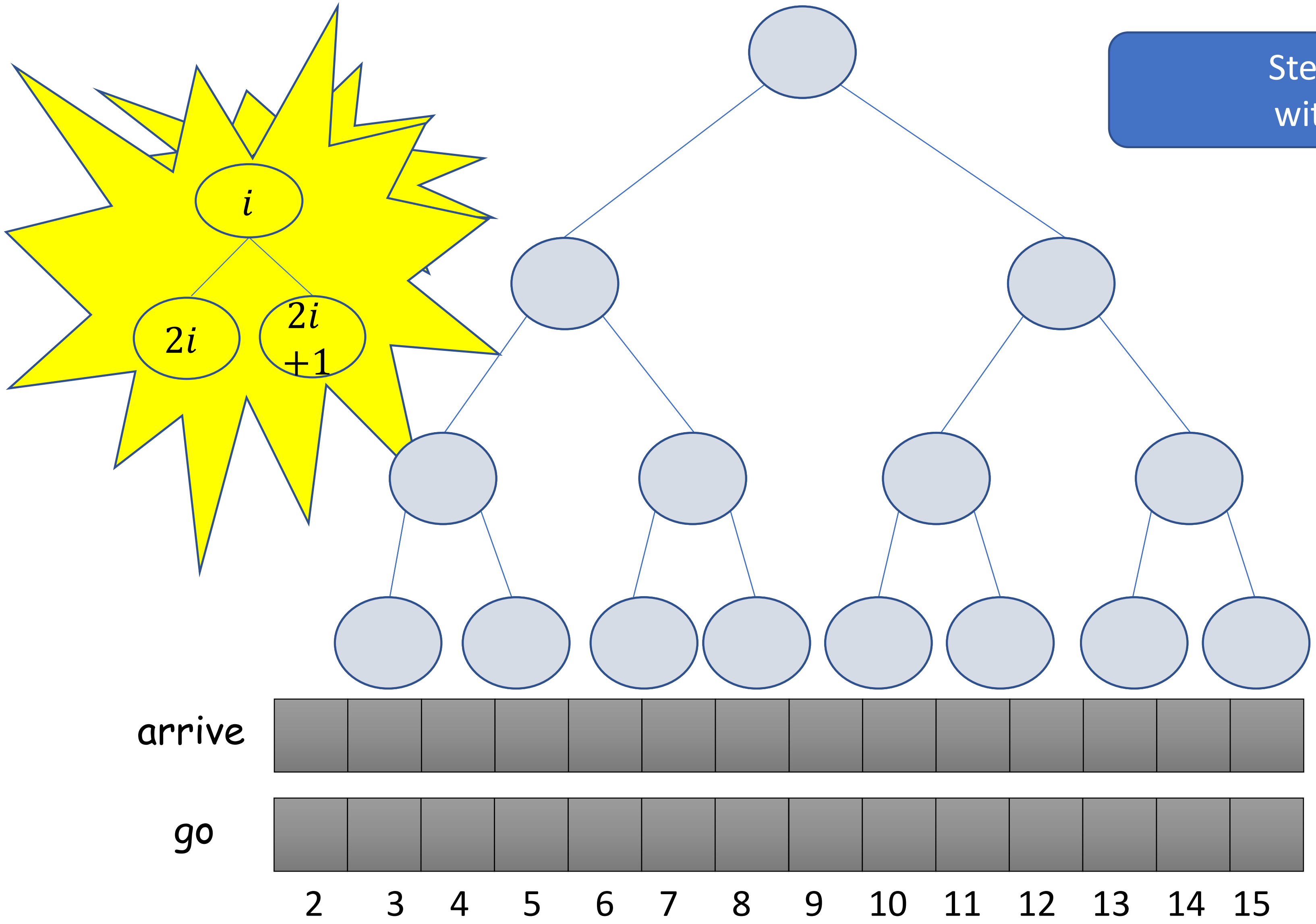
A Tree-based Barrier

- Threads are organized in a binary tree
- Each node is owned by a predetermined thread
- Each thread waits until its 2 children arrive
  - combines results
  - passes them on to its parent
- Root learns that its 2 children have arrived→tells children they can go
- The signal propagates down the tree until all the threads get the message



A Tree-based Barrier: indexing

Step 1: label numerically
with depth-first traveral



Indexing starts from 2
Root → 1, doesn't need wait objects

A Tree-based Barrier

program of thread i

```
shared   arrive[2..n]: array of atomic bits, initial values = 0
         go[2..n]: array of atomic bits, initial values = 0
```

Root

```
if i=1 then                                              // root
      await(arrive[2] = 1); arrive[2] := 0
      await(arrive[3] = 1); arrive[3] := 0
      go[2] = 1; go[3] = 1
```

Internal

```
else if i ≤ (n-1)/2 then                                 // internal node
      await(arrive[2i] = 1); arrive[2i] := 0
      await(arrive[2i+1] = 1); arrive[2i+1] := 0
      arrive[i] := 1
      await(go[i] = 1); go[i] := 0
      go[2i] = 1; go[2i+1] := 1
```

Leaf

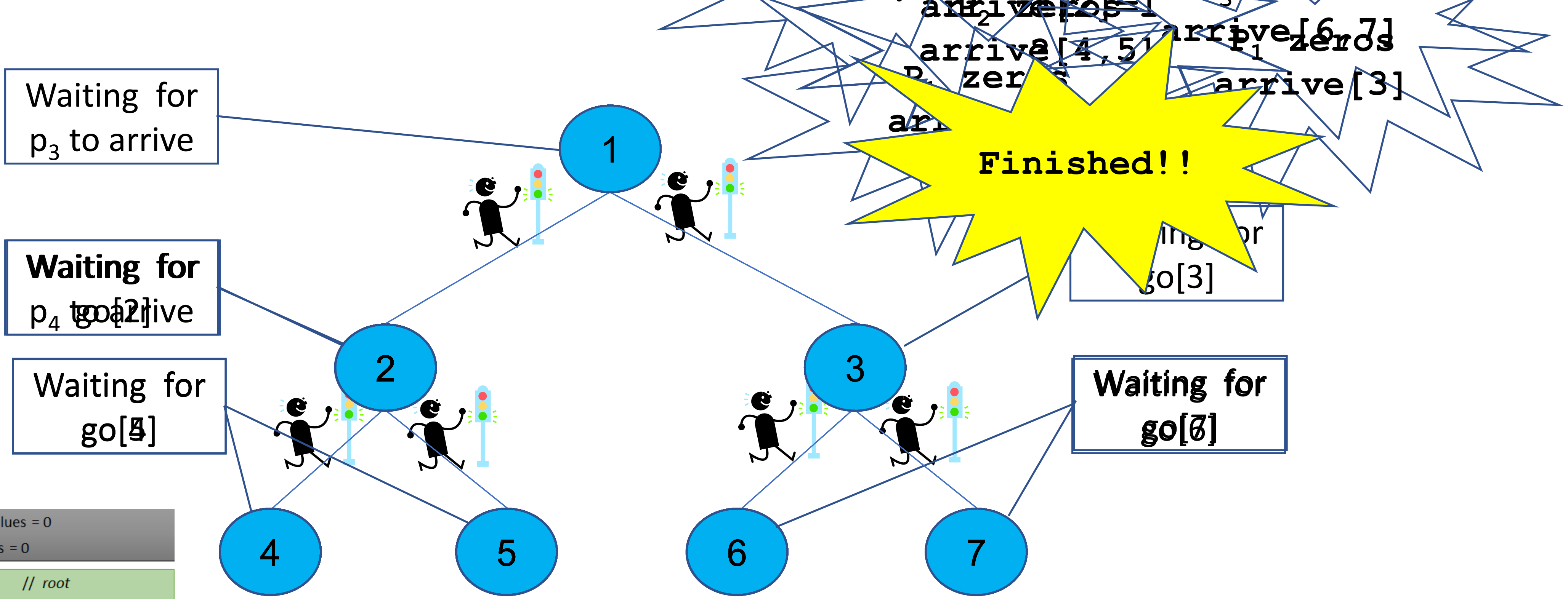
```
else                                                     // leaf
      arrive[i] := 1
      await(go[i] = 1); go[i] := 0 fi
fi
```

Root:
- Wait for arriving children
- Tell children to go

Internal:
- Wait for arriving children
- Tell parents about it
- Wait for parent go signal
- Tell children to go

Child:
- arrive
- Wait for parent go signal

A Tree-based Barrier
Example Run for n=7 threads



shared arrive[2..n]: array of atomic bits, initial values = 0
go[2..n]: array of atomic bits, initial values = 0

```
if i=1 then                                   // root
    await(arrive[2] = 1); arrive[2] := 0
    await(arrive[3] = 1); arrive[3] := 0
    go[2] = 1; go[3] = 1
else if i ≤ (n-1)/2 then                      // internal node
    await(arrive[2i] = 1); arrive[2i] := 0
    await(arrive[2i+1] = 1); arrive[2i+1] := 0
    arrive[i] := 1
    await(go[i] = 1); go[i] := 0
    go[2i] = 1; go[2i+1] := 1
else                                          // leaf
    arrive[i] := 1
    await(go[i] = 1); go[i] := 0 fi
fi
```

	2	3	4	5	6	7
arrive	0	0	0	0	0	0
go	1	1	1	1	1	1

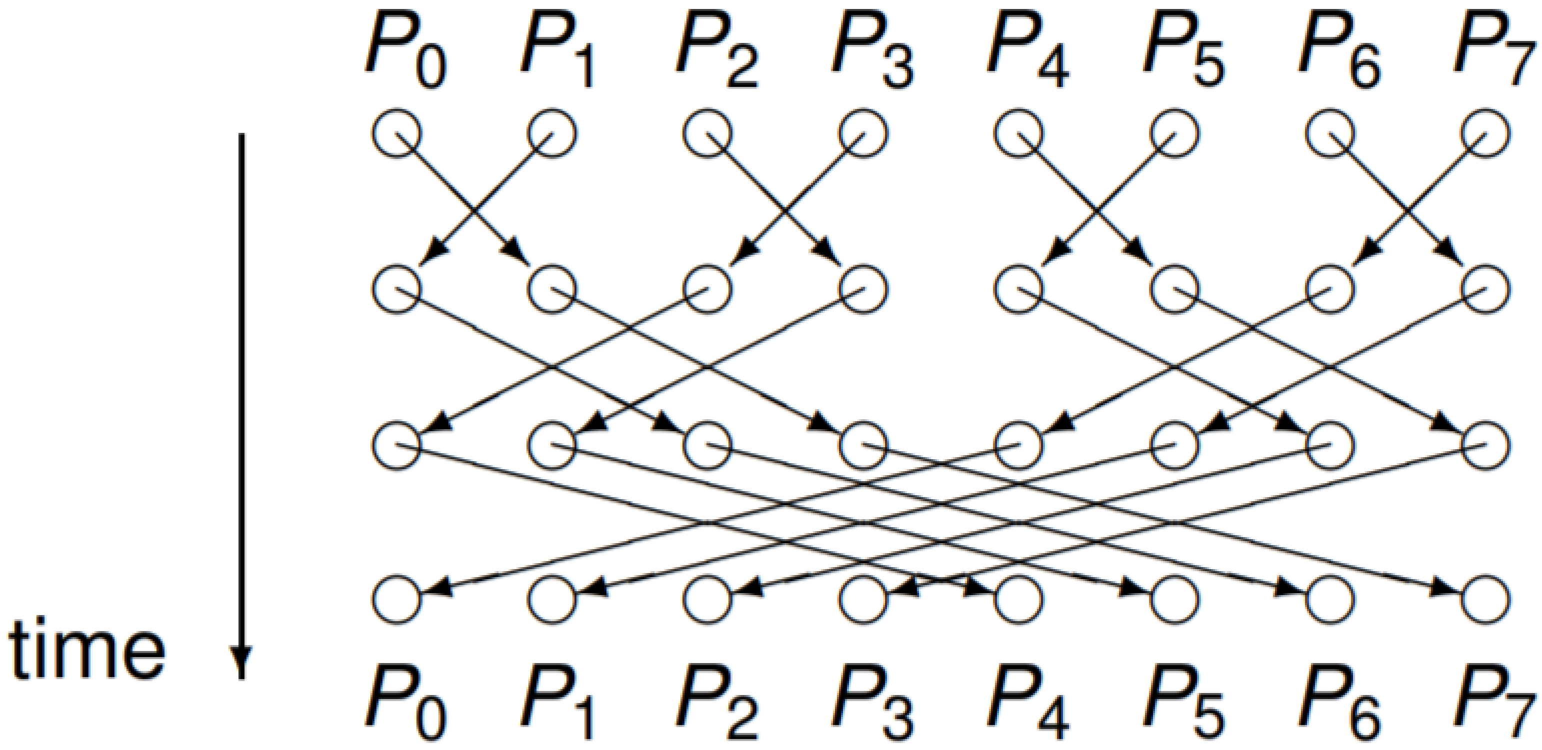
At this point all non-root threads in some await(go) case

Tree Barrier Tradeoffs

- Pros:

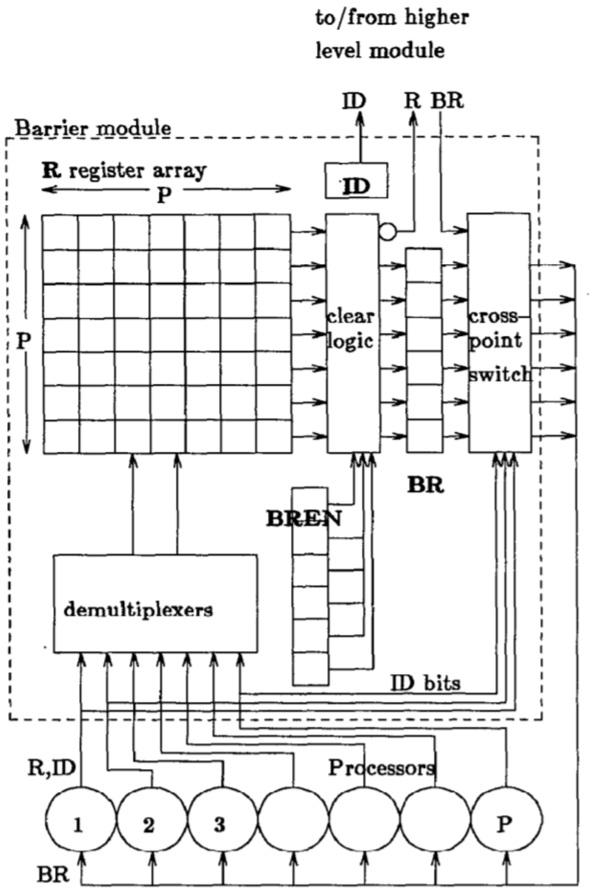
- Cons:

Butterfly Barrier

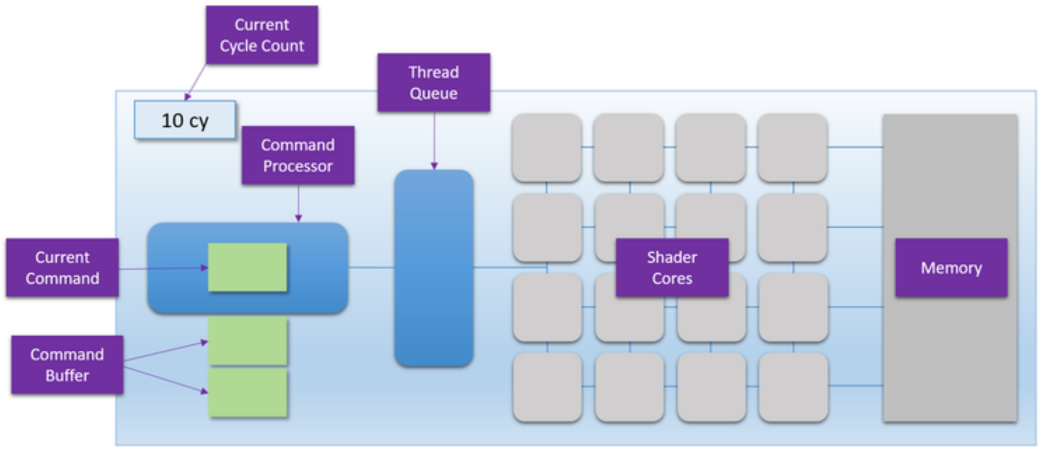


- When would this be preferable?

Hardware Supported Barriers



CPU



GPU

- When would this be useful?

Barriers Summary

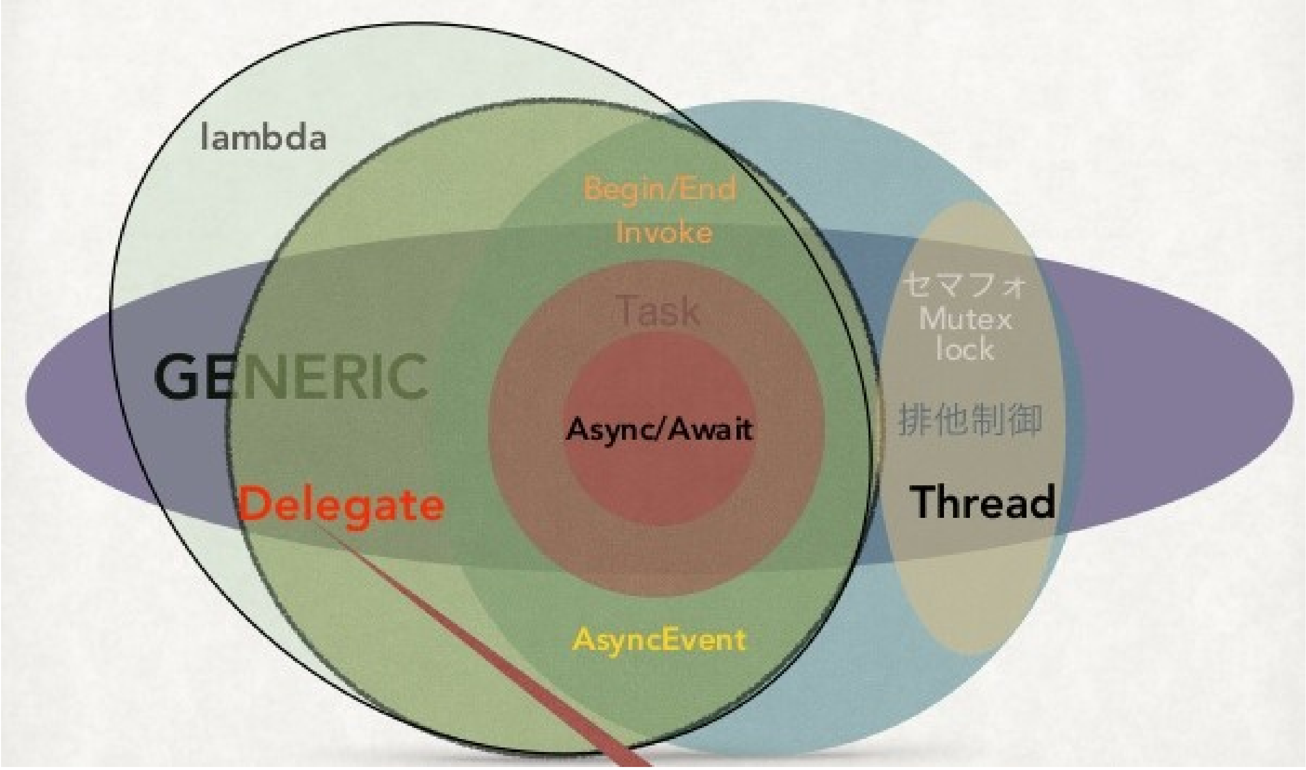
Seen:

- Semaphore-based barrier
- Simple barrier
    - Based on atomic fetch-and-increment counter
- Local spinning barrier
    - Based on atomic fetch-and-increment counter and go array
- Tree-based barrier

Not seen:

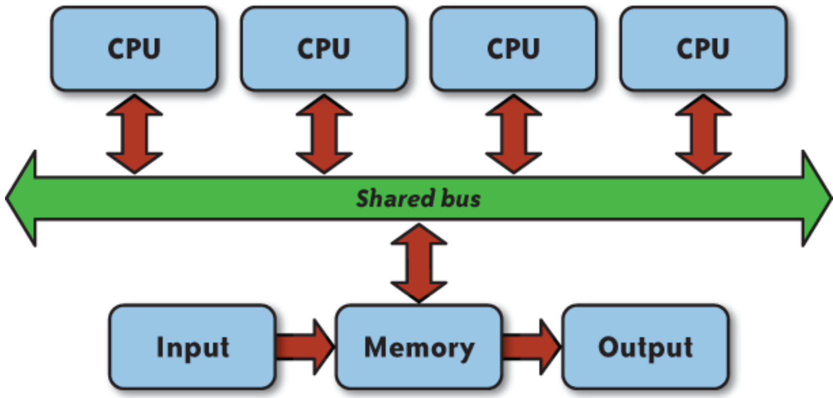
- Test-and-Set barriers
    - Based on test-and-test-and-set objects
    - One version without memory initialization
- See-Saw barrier
- Book has condition barriers

Asynchronous Programming
Events, Promises, and Futures



Programming Models for Concurrency

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

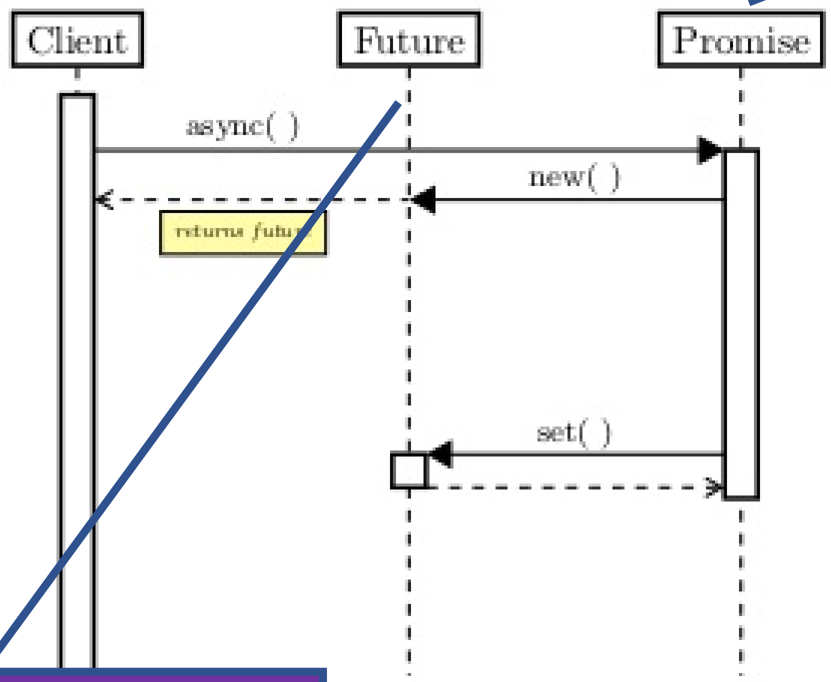
```
static void runAsyncExample() {
    CompletableFuture cf = CompletableFuture.runAsync(() -> {
        assertTrue(Thread.currentThread().isDaemon());
        randomSleep();
    });
    assertFalse(cf.isDone());
    sleepEnough();
    assertTrue(cf.isDone());
}
```

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

Future: encapsulation
(something to give caller)

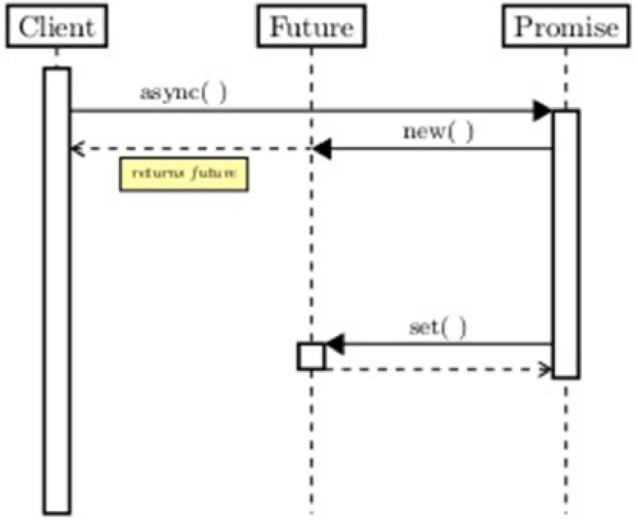
Promise to do something in the future

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

Futures:

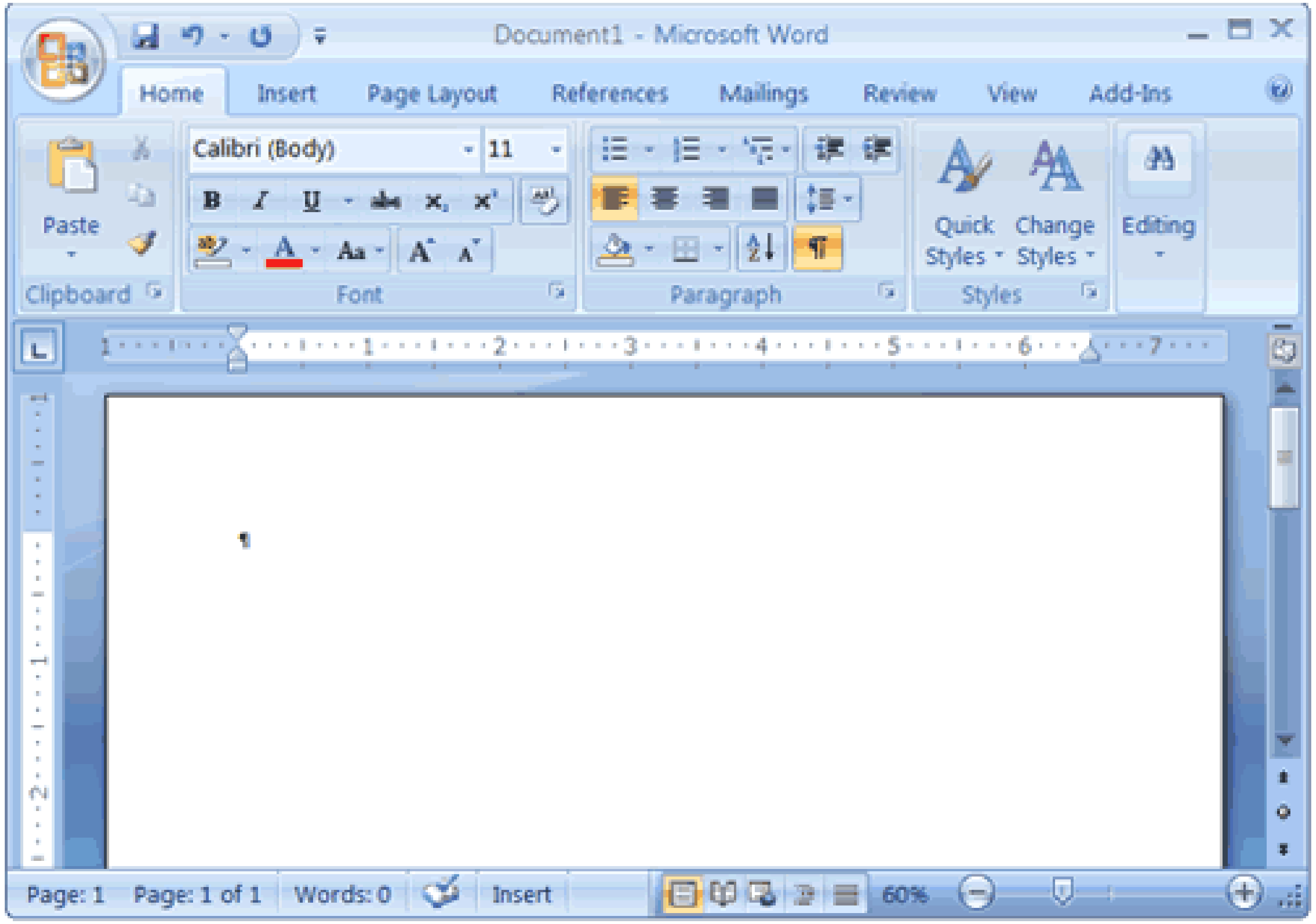
- *abstraction* for concurrent work supported by
  - Compiler: abstractions are ***language-level objects***
  - Runtime: scheduler, task queues, thread-pools are ***transparent***
- Programming remains **mostly** imperative
  - Threads of control peppered with asynchronous/concurrent tasks

Compromise P

- Event-based
- Thread-basec

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

GUI Programming



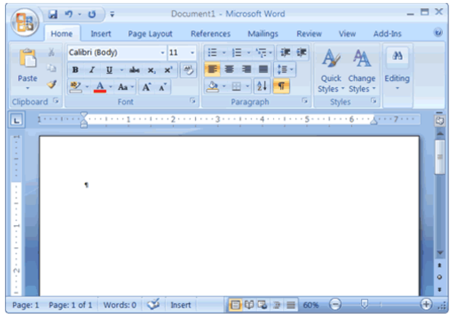

```
do {
  WaitForSomething();
  RespondToThing();
} until(forever);
```

GUI Programming

```
switch (message)
{
        //case WM_COMM
          // handle me
        //break;
        //case WM_PAIN
          // draw our
        //break;
        case WM_DESTRO
             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);
}
```

```
// 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;
}
```

GUI programming

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

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

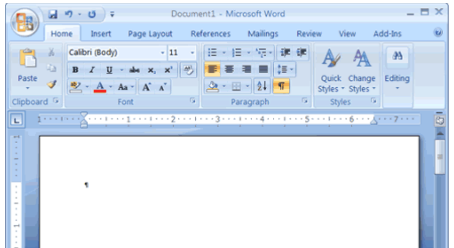
```
void OnMove() { … }
void OnSize() { … }

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

Over 1000 last time I checked!

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



GUI 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**

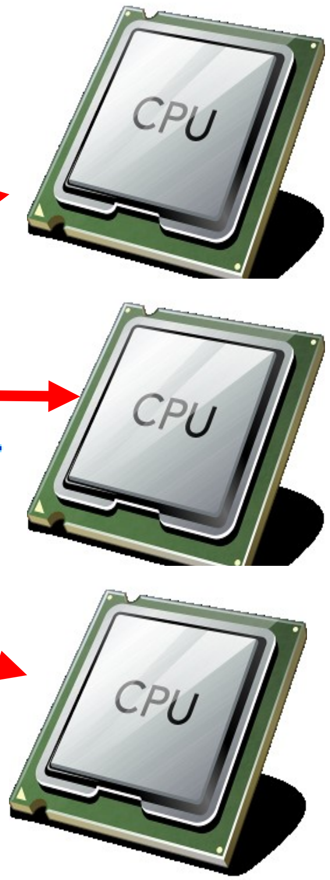
GUI Programming Distilled

```
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 GUI 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 GUI Implementation 1

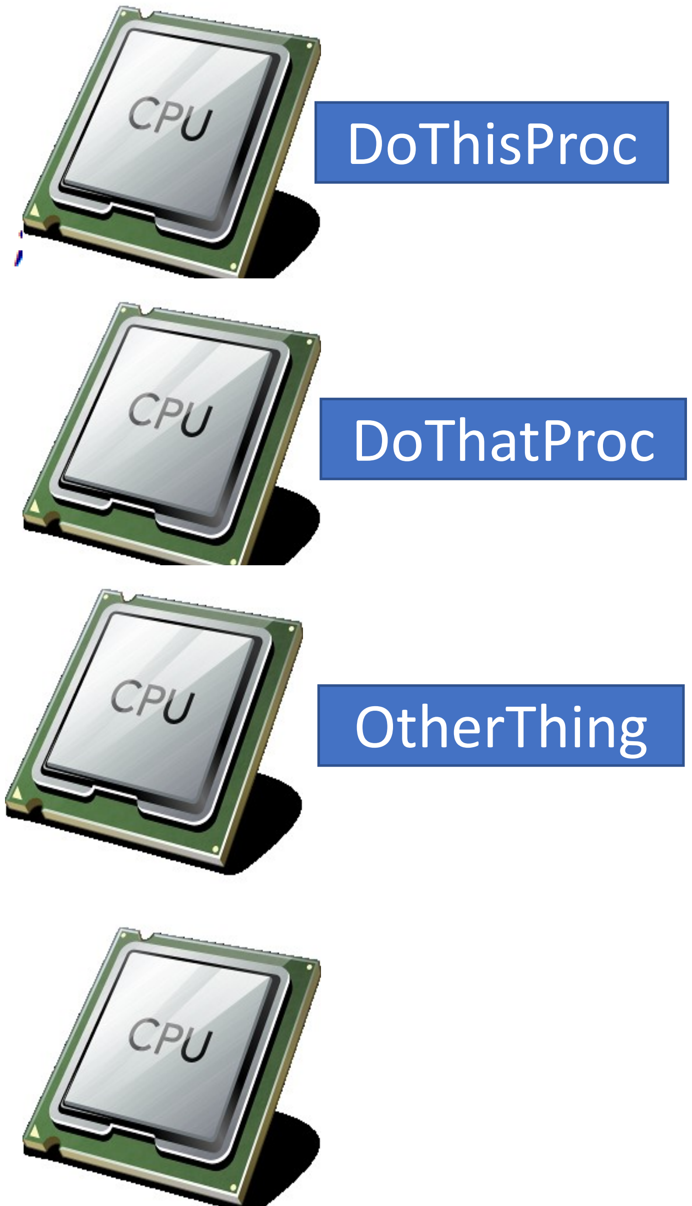
```
winmain() {
    pthread_create(&tids[i++], DoThisProc);
    pthread_create(&tids[i++], DoThatProc);
    pthread_create(&tids[i++], DoOtherThingProc);
    for(j=0; j<i; j++)
        pthread_join(&tids[j]);
}

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

Pros/cons?

Pros:
- Encapsulates parallel work

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



Parallel GUI Implementation 2

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

Pros/cons?

Pros:
- Preserves programming model
- Can recover some parallelism

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

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

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

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

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

Extremely difficult to solve without changing the whole programming model…so ***change it***

Event-based Programming: Motivation

- 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!*

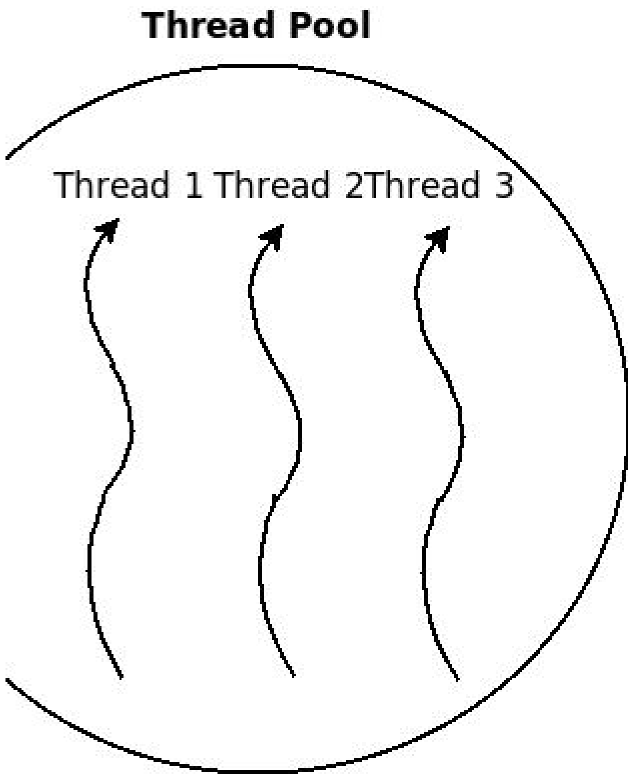
Event Programming Model Basics

- 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

Runtime

```
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);
}
```



Is the problem solved?

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() {
        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
- 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
  - 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
- Event Pros
- Thread Cons

Language-level
Futures: the
eet spot

Cons

Thread Pool Implementation

```
///------------------------------------------------------------------------------------------------
/// <summary>   Starts the threads. </summary>
///
/// <remarks>   crossbac, 8/22/2013. </remarks>
///
/// <param name="uiThreads">            The threads. </param>
/// <param name="bWaitAllThreadsAlive"> The wait all threads alive. </param>
///------------------------------------------------------------------------------------------------

void
ThreadPool::StartThreads(
    __in UINT uiThreads,
    __in BOOL bWaitAllThreadsAlive
    )
{
    Lock();
    if(uiThreads != 0 && m_vhThreadDescs.size() < m_uiTargetSize)
        ResetEvent(m_hAllThreadsAlive);
    while(m_vhThreadDescs.size() < m_uiTargetSize) {
        for(UINT i=0; i<uiThreads; i++) {
            THREADDESC* pDesc = new THREADDESC(this);
            HANDLE * phThread = &pDesc->hThread;
            *phThread = CreateThread(NULL, 0, _ThreadPoolProc, pDesc, 0, NULL);
            m_vhAvailable.push_back(*phThread);
            m_vhThreadDescs[*phThread] = pDesc;
        }
    }
    m_uiThreads = (UINT)m_vhThreadDescs.size();
    Unlock();
    if(bWaitAllThreadsAlive)
        WaitThreadsAlive();
}
```

Cool project idea: build a thread pool!

Thread Pool Implementation

```
DWORD
ThreadPool::ThreadPoolProc(
    __in THREADDESC * pDesc
    )
{
    HANDLE hThread = pDesc->hThread;
    HANDLE hStartEvent = pDesc->hStartEvent;
    HANDLE hRuntimeTerminate = PTask::Runtime::GetRuntimeTerminateEvent();
    HANDLE vEvents[] = { hStartEvent, hRuntimeTerminate };


    NotifyThreadAlive(hThread);
    while(!pDesc->bTerminate) {

        DWORD dwWait = WaitForMultipleObjects(dwEvents, vEvents, FALSE, INFINITE);
        pDesc->Lock();
        pDesc->bTerminate |= bTerminate;
        if(pDesc->bRoutineValid && !pDesc->bTerminate) {
            LPTHREAD_START_ROUTINE lpRoutine = pDesc->lpRoutine;
            LPVOID lpParameter = pDesc->lpParameter;
            pDesc->bActive = TRUE;
            pDesc->Unlock();
            dwResult = (*lpRoutine)(lpParameter);
            pDesc->Lock();
            pDesc->bActive = FALSE;
            pDesc->bRoutineValid = FALSE;
        }
        pDesc->Unlock();
        Lock();
        m_vhInFlight.erase(pDesc->hThread);
        if(!pDesc->bTerminate)
            m_vhAvailable.push_back(pDesc->hThread);
        Unlock();
    }
    NotifyThreadExit(hThread);
    return dwResult;
}
```

ThreadPool Implementation

```
///-------------------------------------------------------------------------------------------------
/// <summary>   Starts a thread: if a previous call to RequestThread was made with
///             the bStartThread parameter set to false, this API signals the thread
///             to begin. Otherwise, the call has no effect (returns FALSE). </summary>
///
/// <remarks>   crossbac, 8/29/2013. </remarks>
///
/// <param name="hThread">  The thread. </param>
///
/// <returns>   true if it succeeds, false if it fails. </returns>
///-------------------------------------------------------------------------------------------------

BOOL
ThreadPool::SignalThread(
    __in HANDLE hThread
    )
{
    Lock();
    BOOL bResult = FALSE;
    std::set<HANDLE>::iterator si = m_vhWaitingStartSignal.find(hThread);
    if(si!=m_vhWaitingStartSignal.end()) {
        m_vhWaitingStartSignal.erase(hThread);
        THREADDESC * pDesc = m_vhThreadDescs[hThread];
        HANDLE hEvent = pDesc->hStartEvent;
        SetEvent(hEvent);
        bResult = TRUE;
    }
    Unlock();
    return bResult;
}
```

Redux: Futures in Context

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());
}
```

Questions?