Synchronization: Implementing Barriers Promises + Futures

Chris Rossbach
CS378H Fall 2018
9/24/18
Today

- Questions?
- Administrivia
  - Lab 2 due Wednesday
- 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 point.
- Once all reach it, all can pass.
- Workhorse of BSP programming models
Review: Barrier API

Wait();
Init/Destroy();

**Type** pthread_barrier_t

```c
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

```c
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)
```

### Phase I

1. sem_wait(arrival);
2. if(++counter < N)
   3. sem_post(arrival);
   4. else
5. sem_post(departure);
6. sem_wait(departure);
7. if(--counter > 0)
   8. sem_post(departure)
   9. else
10. sem_post(arrival)

### Phase II

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 = \[\]
sem_t departure = \[\]
atomic int counter = \[\]

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)

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)

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)
```

// 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

```plaintext
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”

```plaintext
macro await (condition : boolean condition)
    repeat
        cond = eval(condition);
    until (cond)
end-macro
```
Simple Barrier Using an Atomic Counter

<table>
<thead>
<tr>
<th>shared</th>
<th>counter: fetch and increment reg. – {0,..n}, initially = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>go: atomic bit, initial value does not matter</td>
</tr>
<tr>
<td>local</td>
<td>local.go: a bit, initial value does not matter</td>
</tr>
<tr>
<td></td>
<td>local.counter: register</td>
</tr>
</tbody>
</table>

```plaintext
1  local.go := go
2  local.counter := fetch-and-increment (counter)
3  if local.counter + 1 = n then
4      counter := 0
5  go := 1 - go
6  else await(local.go ≠ go)
```
Simple Barrier Using an Atomic Counter
Run for n=2 Threads

1. local.go := go
2. local.counter := fetch-and-increment (counter)
3. if local.counter + 1 = n then
   4. counter := 0
   5. go := 1 - go
4. else await (local.go ≠ go)
Simple Barrier Using an Atomic Counter
Run for n=2 Threads

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

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 |

1. `local.go := go[i]`
2. `local.counter := fetch-and-increment (counter)`
3. `if local.counter + 1 = n then`
4. `counter := 0`
5. `for j=1 to n { go[j] := 1 – go[j] }`
6. `else await(local.go ≠ go[i])`
A Local Spinning Counter Barrier
Example Run for n=3 Threads

Pros/Cons? Does this actually reduce contention?
Comparison of counter-based Barriers

<table>
<thead>
<tr>
<th>Simple Barrier</th>
<th>Simple Barrier with go array</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Pros:</strong></td>
<td><strong>Pros:</strong></td>
</tr>
<tr>
<td><strong>Cons:</strong></td>
<td><strong>Cons:</strong></td>
</tr>
</tbody>
</table>

- Pros:  
- Cons:
### Simple Barrier

**Pros:**
- Very Simple
- Shared memory: $O(\log n)$ bits
- Takes $O(1)$ until last waiting $p$ is awakened

**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 awakened
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
Assume $n = i2^k - 1$

Step 1: label numerically with depth-first traversal

Indexing starts from 2
Root $\rightarrow$ 1, doesn’t need wait objects
A Tree-based Barrier program of thread i

<table>
<thead>
<tr>
<th>shared</th>
<th>arrive[2..n]: array of atomic bits, initial values = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>go[2..n]: array of atomic bits, initial values = 0</td>
</tr>
</tbody>
</table>

if \( i = 1 \) then
// root
2 \hspace{1em} \textbf{await}(arrive[2] = 1); arrive[2] := 0
3 \hspace{1em} \textbf{await}(arrive[3] = 1); arrive[3] := 0
4 \hspace{1em} go[2] = 1; go[3] = 1

else if \( i \leq (n-1)/2 \) then
// internal node
6 \hspace{1em} \textbf{await}(arrive[2i] = 1); arrive[2i] := 0
7 \hspace{1em} \textbf{await}(arrive[2i+1] = 1); arrive[2i+1] := 0
8 \hspace{1em} arrive[i] := 1
9 \hspace{1em} \textbf{await}(go[i] = 1); go[i] := 0
10 \hspace{1em} go[2i] = 1; go[2i+1] := 1

else
// leaf
12 \hspace{1em} arrive[i] := 1
13 \hspace{1em} \textbf{await}(go[i] = 1); go[i] := 0 fi
14 \hspace{1em} fi

Root: \hspace{1em} • Wait for arriving children
      \hspace{1em} • Tell children to go

Internal: \hspace{1em} • Wait for arriving children
          \hspace{1em} • Tell parents about it
          \hspace{1em} • Wait for parent go signal
          \hspace{1em} • Tell children to go

Child: \hspace{1em} • arrive
       \hspace{1em} • Wait for parent go signal
A Tree-based Barrier Example Run for n=7 threads

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

- 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

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

- CompletableFuture is a container for Future object type
- cf is an instance
- runAsync() accepts
  - Lambda expression
  - Anonymous function
  - Functor
- runAsync() immediately returns a waitable object (cf)
- Where (on what thread) does the lambda expression run?
Futures and Promises:
Why two kinds of objects?

Promise: “thing to be done”

Future: encapsulation
(something to give caller)

Promise to do something in the future

```java
code
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

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

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

Futures:

• *abstraction* for concurrent work supported by
  • Compiler: abstractions are *language-level objects*
  • Runtime: scheduler, task queues, thread-pools are *transparent*

• Programming remains *mostly* imperative
  • Threads of control peppered with asynchronous/concurrent tasks

Compromise Programming Model between:

• Event-based
• Thread-based

Events vs. Threads!
GUI Programming

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

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

    // Step 1: Registering the Window Class
    wc.cbSize = sizeof(WMCLASSINFO);
    wc.style = 0;
    wc.lpfnWndProc = WinProc;
    wc.cbWndExtra = 0;
    wc.hInstance = hInstance;
    wc.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    wc.hCursor = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = HRGN(ECHO(COLOR_WINDOW1));
    wc.lpszMenuName = NULL;
    wc.lpszClassName = g_szClassName;
    wc.lpfnWndProc = WinProc;

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

    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 2: Creating the Window
}

switch (message)
{
    case WM_COMMAND:
        // handle menu
        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 Windows
        // to handle it in a default way
        return DefWindowProc(hwnd, message, wParam, lParam);
}

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

```c
void OnMove() {
    ...}

void OnSize() {
    ...}

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

Over 1000 last time I checked!
GUI Programming Distilled

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

Cons
• Awkward/verbose
• Obscures available parallelism
```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;
        }
    }
}
```

How can we parallelize this?
Parallel GUI 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;
    }
  }
}
```
Parallel GUI Implementation 1

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

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

**Pros:**
- Encapsulates parallel work

**Cons:**
- Obliterates original code structure
- How to assign handlers ➔ CPUs?
- Load balance?!?
- Utilization
Pros:
- Preserves programming model
- Can recover some parallelism

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

Extremely difficult to solve without changing the whole programming model... so change it
Event-based Programming: Motivation

• Threads have a *lot* of down-sides:
  • Tuning parallelism for different environments
  • Load balancing/assignment brittle
  • Shared state requires locks →
    • Priority inversion
    • Deadlock
    • Incorrect synchronization
  • ...

• Events: *restructure programming model 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

Is the problem solved?
Another Event-based Program

PROGRAM MyProgram {
    OnOpenFile() {
        char szFileName[BUFSIZE]
        InitFileName(szFileName);
        FILE file = ReadFileEx(szFileName);
        LoadFile(file);
        RedrawScreen();
    } 
    OnPaint();
}
No problem!
Just use more events/handlers, right?

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

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

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

```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();
```
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

- **Event Pros**
  - Easier to create well-conditioned system
  - Easier to express dynamic change in level of parallelism

- **Event Cons**
  - Difficult to program
  - Control flow between callbacks obscure
  - When to deallocate memory
  - Incomplete language/tool/debugger support
  - Difficult to exploit concurrent hardware

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

```c
void ThreadPool::StartThreads(
    __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

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

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

```c++
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

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