Edsger’s perspective

“During system conception it transpired that we used the semaphores in two completely different ways. The difference is so marked that, looking back, one wonders whether it was really fair to present the two ways as uses of the very same primitives. On the one hand, we have the semaphores used for mutual exclusion, on the other hand, the private semaphores.”

The structure of the “THE-Multiprogramming System” Communications of the ACM v. 11 n. 5 May 1968.

Semaphores considered harmful

- Semaphores are “low-level” primitives. Small errors
  - can introduce incorrect executions or grind the program to a halt
  - very difficult to debug
- Semaphores conflate two distinct uses
  - mutex
  - condition synchronization (e.g., bounded buffer)

Enter Monitors

- Collect shared data into an object/module
- Define methods for accessing shared data
- Separate the concerns of mutual exclusion and condition synchronization
- They are comprised of
  - one lock, and
  - zero or more condition variables for managing concurrent access to shared data

How did Monitors come about?

- First introduced as an OO programming language construct
  - synchronization object + methods
  - calling a method defined in the monitor automatically acquires the lock
    - Mesa, Java (synchronized methods)
- A programming convention
  - can be defined in any language
Condition Variables

- An abstraction for conditional synchronization associated with a monitor
- Enable threads to wait inside a critical section by releasing the monitor lock
- A misnomer
  - can neither be read nor set to a value
  - think of them as a label associated with a condition on a resource and a queue
  - thread can wait in the queue (inside the CS) until they are notified that condition holds

How do I wait for thee?
Let me count the ways...

- At the entry of the monitor
  - threads can queue on the mutex that protects the monitor, waiting for the thread that is currently in the monitor to exit (or to release the lock by starting to wait on a condition variable)
- On a condition variable
  - threads can queue waiting on the associated condition

Condition Variables: Operations

- Three operations on condition variable x
  - x.wait(lock)
    - Atomically: Release lock and go to sleep
    - sleep by waiting on the queue associated with x
  - x.notify (historically called x.signal())
    - wake up a waiter if any; otherwise no-op
    - wake up by moving waiter to the ready queue
  - x.notifyall (historically called x.broadcast())
    - Only called while holding a lock

Resource Variables

- Condition variables (unlike semaphores) are stateless
- Each condition variable should be associated with a resource variable (RV) tracking the state of that resource
  - It is your job to maintain the RV!
- Check its RV before calling wait() on a condition variable to ensure the resource is truly unavailable
- Once the resource is available, claim it (subtract the amount you are using!)
- Before notifying you are releasing a resource, indicate it has become available by increasing the corresponding RV
Notify() Semantics

Which thread executed once notify() is called on CV?

- if no thread is waiting on CV, notifier continues
- if one or more thread waiting on CV:
  - at least two ready threads: notifier and thread(s) that are moved from the queue of the CV to the ready queue
  - only one can run...
  - ...but which one?

Notify() semantics: Mesa vs. Hoare

Mesa (or Brinch Hansen) semantics:
- signaled thread is moved to ready list, but not guaranteed to run right away

Hoare semantics:
- signaling thread is suspended and, atomically, ownership of the lock is passed to one of the waiting threads, whose execution is immediately resumed.
- notifying thread is resumed if former waiter exits crucial section, or if it waits again

What are the implications?

Mesa/Brinch Hansen
- signal() and broadcast() are hints
- adding them affects performance, never safety
- Shared state must be checked in a loop (could have changed! (tricky tricky...))
- robust to spurious wakeups
- Simple implementation
- Used in most systems
- Sponsored by a Turing Award
  - Butler Lampson

Hoare
- Signaling is atomic with the resumption of waiting thread
- shared state cannot change before waiting thread is resumed
- Shared state can be checked using an if statement
- Makes it easier to prove liveness
- Tricky to implement
- Does not support broadcast/notifyAll
- Used in most books (but not yours!)
- Sponsored by a Turing Award
  - Tony Hoare

notify() vs notifyall() (signal() vs. broadcast())

- It is always safe to use notifyall() instead of notify()
  - only performance is affected

notify() is preferable when
- at most one waiting thread can make progress (e.g., with mutual exclusion)
- any thread waiting on the condition variable can make progress

notifyall() is preferable when
- multiple waiting thread may be able to make progress
- some waiting threads can make progress, others can't
  - e.g., if a single CV is used for multiple predicates
**Condition Variables vs Semaphores**

- `wait()` vs `P()`
  - `P()` blocks threads only if value = 0
  - `wait()` always block and gives up monitor lock

- `notify()` vs `V()`
  - `V` is stateful - if no thread is waiting, `V()` ensures future thread does not wait on `P()`
  - if no waiting thread, `notify()` is a no-op
  - condition variables are stateless

- Code that uses monitors is easier to read
  - Conditions for which threads are waiting are explicit

---

**Producer-Consumer with Bounded Buffer**

```
// add item to buffer
void produce(int item) {
    mutex_in.P();
    buf[in%N] := item;
    in := in+1;
    mutex_in.V();
}

// remove item from buffer
int consume() {
    lock.Acquire();
    while (n == 0)
        wait(nonEmpty);
    int item := buf[out%N];
    out := out+1;
    n := n-1;
    notify(notFull);
    lock.Release();
    return(item);
}
```

---

**Monitor**

```
Monitor Producer Consumer { char buf[N];
    Lock lock;
    int n := 0, in := 0, out := 0;
    Condition notEmpty, notFull;
    }
```

---

**Which is Mesa? Which is Hoare?**

**Producer-Consumer with Bounded Buffer**

```
// add item to buffer
int produce(int item) {
    mutex_in.P();
    buf[in%N] := item;
    in := in+1;
    mutex_in.V();
}

// remove item from buffer
int consume() {
    lock.Acquire();
    while (n == 0)
        wait(nonEmpty);
    int item := buf[out%N];
    out := out+1;
    n := n-1;
    notify(notFull);
    lock.Release();
    return(item);
}
```
Kid and Cook Threads

Monitor BurgerKing {
    Lock mlock;
    int numbureg = 0;
    condition burgerReady;
}

kid_main() {
    dig_in_mud();
    BK.kid_eat();
    bathe();
    draw_on_walls();
    BK.kid_eat();
    facetime_Karthik();
    facetime_oma();
    BK.kid_eat();
}

cook_main() {
    wake();
    shower();
    drive_to_work();
    while(not_5pm)
        BK.makeburger();
    drive_to_home();
    watch_got();
    sleep();
}

void kid_eat() {
    mlock.acquire();
    while (numburgers==0) 
        BK.burgerReady.wait();
    numbureg -= 1
    mlock.release();
}

void makeburger() {
    mlock.acquire();
    ++numburger;
    burgerReady.signal();
    mlock.release();
}

Ready

Running

girl swapped out

Kid and Cook Threads

Monitor BurgerKing {
    Lock mlock;
    int numbureg = 0;
    condition burgerReady;
}

kid_main() {
    dig_in_mud();
    BK.kid_eat();
    bathe();
    draw_on_walls();
    BK.kid_eat();
    facetime_Karthik();
    facetime_oma();
    BK.kid_eat();
}

cook_main() {
    wake();
    shower();
    drive_to_work();
    while(not_5pm)
        BK.makeburger();
    drive_to_home();
    watch_got();
    sleep();
}

void kid_eat() {
    mlock.acquire();
    while (numburgers==0) 
        BK.burgerReady.wait();
    numbureg -= 1
    mlock.release();
}

void makeburger() {
    mlock.acquire();
    ++numburger;
    burgerReady.signal();
    mlock.release();
}

Ready

Running

cook swapped in

Kid and Cook Threads

Monitor BurgerKing {
    Lock mlock;
    int numbureg = 0;
    condition burgerReady;
}

kid_main() {
    dig_in_mud();
    BK.kid_eat();
    bathe();
    draw_on_walls();
    BK.kid_eat();
    facetime_Karthik();
    facetime_oma();
    BK.kid_eat();
}

cook_main() {
    wake();
    shower();
    drive_to_work();
    while(not_5pm)
        BK.makeburger();
    drive_to_home();
    watch_got();
    sleep();
}

void kid_eat() {
    mlock.acquire();
    while (numburgers==0) 
        BK.burgerReady.wait();
    numbureg -= 1
    mlock.release();
}

void makeburger() {
    mlock.acquire();
    ++numburger;
    burgerReady.signal();
    mlock.release();
}

Ready

Running

cook swapped out

Kid and Cook Threads

Monitor BurgerKing {
    Lock mlock;
    int numbureg = 0;
    condition burgerReady;
}

kid_main() {
    dig_in_mud();
    BK.kid_eat();
    bathe();
    draw_on_walls();
    BK.kid_eat();
    facetime_Karthik();
    facetime_oma();
    BK.kid_eat();
}

cook_main() {
    wake();
    shower();
    drive_to_work();
    while(not_5pm)
        BK.makeburger();
    drive_to_home();
    watch_got();
    sleep();
}

void kid_eat() {
    mlock.acquire();
    while (numburgers==0) 
        BK.burgerReady.wait();
    numbureg -= 1
    mlock.release();
}

void makeburger() {
    mlock.acquire();
    ++numburger;
    burgerReady.signal();
    mlock.release();
}

Ready

Running

cook swapped out

Kid and Cook Threads

Monitor BurgerKing {
    Lock mlock;
    int numbureg = 0;
    condition burgerReady;
}

kid_main() {
    dig_in_mud();
    BK.kid_eat();
    bathe();
    draw_on_walls();
    BK.kid_eat();
    facetime_Karthik();
    facetime_oma();
    BK.kid_eat();
}

cook_main() {
    wake();
    shower();
    drive_to_work();
    while(not_5pm)
        BK.makeburger();
    drive_to_home();
    watch_got();
    sleep();
}

void kid_eat() {
    mlock.acquire();
    while (numburgers==0) 
        BK.burgerReady.wait();
    numbureg -= 1
    mlock.release();
}

void makeburger() {
    mlock.acquire();
    ++numburger;
    burgerReady.signal();
    mlock.release();
}

Ready

Running

cook executes
Kid and Cook Threads

Monitor BurgerKing {
Lock mlock;
int numb burgers = 0;
condition burgerReady;
}

void kid_eat() {
mlock.acquire()
while (numburgers==0)
    burgerReady.wait()
numburgers -= 1
mlock.release()
}
void makeburger() {
mlock.acquire()
++numburger;
burgerReady.signal();
mlock.release()
}

boy swapped in

Kid and Cook Threads

Monitor BurgerKing {
Lock mlock;
int numb burgers = 0;
condition burgerReady;
}

void kid_eat() {
mlock.acquire()
while (numburgers==0)
    burgerReady.wait()
numburgers -= 1
mlock.release()
}
void makeburger() {
mlock.acquire()
++numburger;
burgerReady.signal();
mlock.release()
}

boy executes

Kid and Cook Threads

Monitor BurgerKing {
Lock mlock;
int numb burgers = 0;
condition burgerReady;
}

void kid_eat() {
mlock.acquire()
while (numburgers==0)
    burgerReady.wait()
numburgers -= 1
mlock.release()
}
void makeburger() {
mlock.acquire()
++numburger;
burgerReady.signal();
mlock.release()
}

boy gets monitor lock
Kid and Cook Threads

Monitor BurgerKing

```c
LOCK mlock;
int numburgers = 0;
condition burgerReady;
```

```c
void kid_eat() {
mlock.acquire()
while (numburgers == 0)
  burgerReady.wait()
numburgers -= 1
mlock.release()
}
```

```c
void makeburger() {
mlock.acquire()
++numburger;
burgerReady.signal();
mlock.release()
}
```

```c
void facetime_Karthik() {
}
```

```c
void facetime_oma() {
}
```

```c
void bathe() {
}
```

```c
void draw_on_walls() {
}
```

```c
void drive_to_work() {
}
```

```c
void drive_to_home() {
}
```

```c
void watch_got() {
}
```

```c
void sleep() {
}
```

```c
while(not_5pm)
  BK.makeburger();
```

```c
while(not_5pm)
  BK.kid_eat();
```

```c
while(not_5pm)
  BK.kid_eat();
```

```c
BK.kid_eat();
```

```c
BK.kid_eat();
```

```c
BK.kid_eat();
```

```c
BK.kid_eat();
```

```c
BK.kid_eat();
```

```c
BK.kid_eat();
```

```c
BK.kid_eat();
```

```c
BK.kid_eat();
```

```c
BK.kid_eat();
```

```
```
Kid and Cook Threads

Monitor BurgerKing {
    lock mlock;
    int numbburgers = 1
    condition burgerReady;
}

kid_main() {
    dig_in_mud();
    BK.kid_eat();
    bathe();
    draw_on_walls();
    BK.kid_eat();
    facetime_Karthik();
    facetime_oma();
    BK.kid_eat();
}

void kid_eat() {
    lock.acquire();
    while (numburgers==0)
        burgerReady.wait();
    numbburgers -= 1
    mlock.release();
}

void makeburger() {
    lock.acquire();
    ++numburger;
    burgerReady.signal();
    mlock.release();
}

Running

girl placed on lock Q

Kid and Cook Threads

cook_main() {
    wake();
    shower();
    drive_to_work();
    while(not_5pm)
        BK.makeburger();
    drive_to_home();
    watch_got();
    sleep();
}

void makeburger() {
    lock.acquire();
    ++numburger;
    burgerReady.signal();
    mlock.release();
}

Running

cook executes

Kid and Cook Threads

Monitor BurgerKing {
    lock mlock;
    int numbburgers = 1
    condition burgerReady;
}

kid_main() {
    dig_in_mud();
    BK.kid_eat();
    bathe();
    draw_on_walls();
    BK.kid_eat();
    facetime_Karthik();
    facetime_oma();
    BK.kid_eat();
}

void kid_eat() {
    lock.acquire();
    while (numburgers==0)
        burgerReady.wait();
    numbburgers -= 1
    mlock.release();
}

void makeburger() {
    lock.acquire();
    ++numburger;
    burgerReady.signal();
    mlock.release();
}

Running

cook tries to enter monitor
Kid and Cook Threads

```
Monitor BurgerKing {
    Lock mlock;
    int numbuer = 0;
    condition burgerReady;
}

void kid_eat() {
    mlock.acquire()
    while (numburgers==0)
        burgerReady.wait()
    numbuer --
    mlock.release()
}

void makeburger() {
    mlock.acquire()
    ++numburger;
    burgerReady.signal();
    mlock.release()
}
```

```
kid_main() {
    dig_in_mud();
    BK.kid_eat();
    bathe();
    draw_on_walls();
    BK.kid_eat();
    facetime_Karthik();
    facetime_oma();
    BK.kid_eat();
}

cook_main() {
    wake();
    shower();
    drive_to_work();
    while(not_5pm)
        BK.makeburger();
    drive_to_home();
    watch_got();
    sleep();
}
```

Ready

Running

Ready

Running

Ready

Running

no burgers!

```
boy swapped in w/lock

boy releases monitor lock & waits for hungrykid signal
```
**Kid and Cook Threads**

```c
 Kid and Cook Threads

``` kid_main() {
dig_in_mud();
BK.kid_eat();
bathe();
draw_on_walls();
BK.kid_eat();
facetime_Karthik();
facetime_oma();
BK.kid_eat();
}

```c
 Monitor BurgerKing {
Lock mlock;

int numbburgers = 1;
condition burgerReady;

void kid_eat() {
lock.acquire();
while (numburgers==0)
    burgerReady.wait();
numburgers -= 1
lock.release();
}

void makeburger() {
lock.acquire();
++numburger;
burgerReady.signal();
lock.release();
}
}

``` c
 Running

```c
 Ready

```

**Kid and Cook Threads**

```c
 cook made Ready with release of monitor lock

```

```c
 Kid and Cook Threads

``` kid_main() {
dig_in_mud();
BK.kid_eat();
bathe();
draw_on_walls();
BK.kid_eat();
facetime_Karthik();
facetime_oma();
BK.kid_eat();
}

```c
 Monitor BurgerKing {
Lock mlock;

int numbburgers = 1;
condition burgerReady;

void kid_eat() {
lock.acquire();
while (numburgers==0)
    burgerReady.wait();
numburgers -= 1
lock.release();
}

void makeburger() {
lock.acquire();
++numburger;
burgerReady.signal();
lock.release();
}
}

``` c
 Running

```c
 Ready

```

**Kid and Cook Threads**

```c
 cook swapped in

```

```c
 Kid and Cook Threads

``` kid_main() {
dig_in_mud();
BK.kid_eat();
bathe();
draw_on_walls();
BK.kid_eat();
facetime_Karthik();
facetime_oma();
BK.kid_eat();
}

```c
 Monitor BurgerKing {
Lock mlock;

int numbburgers = 1;
condition burgerReady;

void kid_eat() {
lock.acquire();
while (numburgers==0)
    burgerReady.wait();
numburgers -= 1
lock.release();
}

void makeburger() {
lock.acquire();
++numburger;
burgerReady.signal();
lock.release();
}
}

``` c
 Running

```c
 Ready

```

**Kid and Cook Threads**

```c
 cook acquires monitor lock

```

```c
 Kid and Cook Threads

``` kid_main() {
dig_in_mud();
BK.kid_eat();
bathe();
draw_on_walls();
BK.kid_eat();
facetime_Karthik();
facetime_oma();
BK.kid_eat();
}

```c
 Monitor BurgerKing {
Lock mlock;

int numbburgers = 1;
condition burgerReady;

void kid_eat() {
lock.acquire();
while (numburgers==0)
    burgerReady.wait();
numburgers -= 1
lock.release();
}

void makeburger() {
lock.acquire();
++numburger;
burgerReady.signal();
lock.release();
}
}

``` c
 Running

```c
 Ready

```

**Kid and Cook Threads**

```c
 cook makes a burger

```
Kid and Cook Threads

Monitor BurgerKing {
  Lock mlock;
  int numburgers = 0;
  condition burgerReady;
}

void kid_eat() {
  mlock.acquire()
  while (numburgers==0)
    burgerReady.wait()
  numburgers -= 1
  mlock.release()
}

void makeburger() {
  mlock.acquire()
  ++numburger;
  burgerReady.signal();
  mlock.release()
}

Kid and Cook Threads

void kid_eat() {
  mlock.acquire()
  while (numburgers==0)
    burgerReady.wait()
  numburgers -= 1
  mlock.release()
}

void makeburger() {
  mlock.acquire()
  ++numburger;
  burgerReady.signal();
  mlock.release()
}

Kid and Cook Threads

void kid_eat() {
  mlock.acquire()
  while (numburgers==0)
    burgerReady.wait()
  numburgers -= 1
  mlock.release()
}

void makeburger() {
  mlock.acquire()
  ++numburger;
  burgerReady.signal();
  mlock.release()
}

Kid and Cook Threads

void kid_eat() {
  mlock.acquire()
  while (numburgers==0)
    burgerReady.wait()
  numburgers -= 1
  mlock.release()
}

void makeburger() {
  mlock.acquire()
  ++numburger;
  burgerReady.signal();
  mlock.release()
}
**Kid and Cook Threads**

```c
Monitor BurgerKing {
    Lock mlock;
    int numbburgers = 0;
    condition burgerReady;
}

void kid_eat() {
    mlock.acquire()
    while (numburgers==0)
        burgerReady.wait()
    numbburgers -= 1
    mlock.release()
}

void makeburger() {
    mlock.acquire()
    ++numburger;
    burgerReady.signal();
    mlock.release()
}

void kid_eat() {
    mlock.acquire()
    while (numburgers==0)
        burgerReady.wait()
    numbburgers -= 1
    mlock.release()
}

void makeburger() {
    mlock.acquire()
    ++numburger;
    burgerReady.signal();
    mlock.release()
}

void kid_eat() {
    mlock.acquire()
    while (numburgers==0)
        burgerReady.wait()
    numbburgers -= 1
    mlock.release()
}

void makeburger() {
    mlock.acquire()
    ++numburger;
    burgerReady.signal();
    mlock.release()
}

void kid_eat() {
    mlock.acquire()
    while (numburgers==0)
        burgerReady.wait()
    numbburgers -= 1
    mlock.release()
}

void makeburger() {
    mlock.acquire()
    ++numburger;
    burgerReady.signal();
    mlock.release()
}
```

**Kid and Cook Threads**

**Kid and Cook Threads**

**Kid and Cook Threads**

**Kid and Cook Threads**

**Kid and Cook Threads**

**Kid and Cook Threads**

**Kid and Cook Threads**

**Kid and Cook Threads**

**Kid and Cook Threads**

**Kid and Cook Threads**

**Kid and Cook Threads**

**Kid and Cook Threads**

**Kid and Cook Threads**

**Kid and Cook Threads**
Kid and Cook Threads

Kid and Cook Threads

Kid and Cook Threads

Kid and Cook Threads
Monitor BurgerKing {
    Lock mlock;
    int numburgers = 0;
    condition burgerReady;
    void kid_eat() {
        mlock.acquire()
        while (numburgers==0) burgerReady.wait()
        numburgers -= 1
        mlock.release()
    }
    void makeburger() {
        mlock.acquire()
        ++numburger;
        burgerReady.signal();
        mlock.release()
    }
}

kid_main() {
    dig_in_mud();
    BK.kid_eat();
    bathe();
    draw_on_walls();
    BK.kid_eat();
    facetime_Karthik();
    facetime_oma();
    BK.kid_eat();
}

cook_main() {
    wake();
    shower();
    drive_to_work();
    while(not_5pm)
        BK.makeburger();
    drive_to_home();
    watch_got();
    sleep();
}

girl executes

boy swapped out

Monitor BurgerKing {
    Lock mlock;
    int numburgers = 0;
    condition burgerReady;
    void kid_eat() {
        mlock.acquire()
        while (numburgers==0) burgerReady.wait()
        numburgers -= 1
        mlock.release()
    }
    void makeburger() {
        mlock.acquire()
        ++numburger;
        burgerReady.signal();
        mlock.release()
    }
}

cook_main() {
    wake();
    shower();
    drive_to_work();
    while(not_5pm)
        BK.makeburger();
    drive_to_home();
    watch_got();
    sleep();
}
girl swapped out

boy swapped in

boy acquires monitor lock
Kid and Cook Threads

Monitor BurgerKing {
    Lock mlock;
    int numburegs = 0;
    condition burgerReady;
    q:
    void kid_eat() {
        mlock.acquire()
        while (numburgers==0)
        burgerReady.wait()
        numburegs -= 1
        mlock.release()
    }
    void makeburger() {
        mlock.acquire()
        ++numburger;
        burgerReady.signal();
        mlock.release()
    }
}

void kid_eat() {
    mlock.acquire()
    while (numburgers==0)
    burgerReady.wait()
    numburegs -= 1
    mlock.release()
}

void makeburger() {
    mlock.acquire()
    ++numburger;
    burgerReady.signal();
    mlock.release()
}

Kid and Cook Threads

boy returns from wait

no burgers!

boy releases monitor lock & waits for hungrykid signal
Designing multithreaded programs

- **Familiar steps**
  - Decompose problem into objects; for each object:
    - define a clear interface
    - implement methods that manipulate state appropriately

- **New steps**
  - Add a lock, and code to acquire and release the lock
  - acquire a lock at the start of each public method
  - release the lock at the end of each public method
  - Identify synchronization points and add condition variables
  - add loops to check resource status and wait using condition variables
  - Add notify(), notifyAll() calls

12 Commandments of Synchronization

- Thou shalt name your synchronization variables properly.
- Thou shalt not violate abstraction boundaries nor try to change the semantics of synchronization primitives.
- Thou shalt use monitors and condition variables instead of semaphores whenever possible.
- Thou shalt not mix semaphores and condition variables.
- Thou shalt not rely on wait.
- Thou shalt protect all shared state.
- Thou shalt not split predicates.
- Thou shalt help make the world a better place for the creator’s mighty synchronization vision.

Kid and Cook Threads

cook swapped in

Monitor BurgerKing {
  Lock mlock;
  int numburegers = 0;
  condition burgerReady;
  void kid_eat() {
    mlock.acquire()
    while (numburgers==0)
      burgerReady.wait()
    numburegers -= 1
    mlock.release()
  }
  void makeburger() {
    mlock.acquire()
    ++numburger;
    burgerReady.signal();
    mlock.release()
  }
}

cook_main() {
  wake();
  shower();
  drive_to_work();
  while(not_5pm)
    BK.makeburger();
  drive_to_home();
  watch_got();
  sleep();
}
kid_main() {
  dig_in_mud();
  BK.kid_eat();
  draw_on_walls();
  BK.kid_eat();
  facetime_Karthik();
  facetime_oma();
  BK.kid_eat();
}
Readers/Writers

Safety

What about fairness?
- the last thread to live the critical section will give priority to writers
- To implement this policy, one needs to keep track of waitingWriters, waitingReaders, activeWriters, and activeReaders

Monitor ReadersNWriters {
    int waitingWriters=0, waitingReaders=0, activeReaders=0, activeWriters=0;
    Condition canRead, canWrite;

    void BeginWrite()
        with monitor.lock:
            ++waitingWriters
            while (activeWriters > 0 or activeReaders > 0)
                canWrite.wait();
            --waitingWriters
            activeWriters = 1;

    void EndWrite()
        with monitor.lock:
            activeWriters = 0
            if waitingWriters > 0
                canWrite.signal();
            else if waitingReaders > 0
                canRead.broadcast();
}

void BeginRead()
    with monitor.lock:
        ++waitingReaders
        while (activeWriters > 0 or waitingWriters > 0)
            canRead.wait();
        --waitingReaders
        activeReaders = 1;

    void EndRead()
        with monitor.lock:
            activeReaders = 0
            if waitingWriters > 0
                canWrite.signal();
            else if waitingReaders > 0
                canRead.broadcast();
}
Readers/Writers

Monitor ReadersNWriters {
    int waitingWriters=0, waitingReaders=0, activeReaders=0, activeWriters=0;
    Condition canRead, canWrite;

    void BeginWrite()
        with monitor.lock:
            ++waitingWriters
            while (activeWriters > 0 or activeReaders > 0)
                canWrite.wait();
            --waitingWriters
            activeWriters = 1;

    void EndWrite()
        with monitor.lock:
            if waitingWriters > 0
                canWrite.signal();
            else if waitingReaders > 0
                canRead.broadcast();

    void BeginRead()
        with monitor.lock:
            ++waitingReaders
            while (activeWriters > 0 or waitingWriters > 0)
                canRead.wait();
            --waitingReaders
            ++activeReaders

    void EndRead()
        with monitor.lock:
            --activeReaders;
            if (activeReaders == 0 and waitingWriters > 0)
                canWrite.signal();
}

Contra Threads: Events

John Ousterhout: “Why Threads Are a Bad Idea (for most purposes)”

Event-driven Programming

- No concurrency: one execution stream
- Register interest in events (callbacks)
- Wait for events; invoke (short-lived) handlers
- Complicated only for unusual cases
- Easier to debug

Event Loop

H1 H2 H3 H4 H5