Outline for Today

- Objectives:
  - Birrell’s paper
  - Eraser paper

- Administrative details:
  - First programming assignment is now posted on the web!

Template for Readers/Writers

```c
Reader()
{while (true)
{
  read
  startRead();
  read
  endRead();
}
}
```

```c
Writer()
{while (true)
{
  write
  startWrite();
  write
  endWrite();
}
}
```
Boolean busy = false;
int numReaders = 0;
Lock filesMutex;
Condition OKtoWrite, OKtoRead;

void startRead () {
    filesMutex.Acquire( );
    while ( busy )
        OKtoRead.Wait(&filesMutex);
    numReaders++;
    filesMutex.Release( );
}

void endRead () {
    filesMutex.Acquire( );
    numReaders--;
    if (numReaders == 0)
        OKtoWrite.Signal(&filesMutex);
    filesMutex.Release( );
}

void startWrite() {
    filesMutex.Acquire( );
    int index = add_index();
    while (busy || compare_index())
        OKtoWrite.Wait(&filesMutex);
    busy = true;
    remove_index();
    filesMutex.Release( );
}

void endWrite() {
    filesMutex.Acquire( );
    busy = false;
    OKtoWrite.Broadcast(&filesMutex);
    filesMutex.Release( );
}
Boolean busy = false,
writing = false;
int numReaders = 0;
Lock filesMutex;
Condition OKtoWrite, OKtoRead;

void startRead () {
    filesMutex.Acquire( );
    while (busy || writing)
        OKtoRead.Wait(&filesMutex);
    numReaders++;
    filesMutex.Release( );
}

void endRead () {
    filesMutex.Acquire( );
    numReaders--;
    if (numReaders == 0)
        OKtoWrite.Signal(&filesMutex);
    filesMutex.Release( );
}

void startWrite() {
    filesMutex.Acquire( );
    while (busy || numReaders != 0)
        {writing = true;
        OKtoWrite.Wait(&filesMutex);
    writing = false;
    busy = true;
    filesMutex.Release( );
}

void endWrite() {
    filesMutex.Acquire( );
    busy = false;
    OKtoRead.Broadcast(&filesMutex);
    OKtoWrite.Signal(&filesMutex);
    filesMutex.Release( );
}

Fairness?

- No writer starvation
- No reader starvation
- Strictly FIFO
**Tricks (mixed syntax)**

```plaintext
if (some_condition) // as a hint
{
    LOCK m DO
        if (some_condition) // the truth
            {stuff}
    END
    Cheap to get info but must check for correctness; always a slow way
}
```

---

**More Tricks**

General pattern:
```plaintext
while (! required_conditions) wait (m, c);
```

Broadcast works because waking up too many is OK (correctness-wise) although a performance impact.

LOCK m DO
```plaintext
    ... deferred_signal = true;
END
```

if (deferred_signal) signal (c);  

Spurious lock conflicts caused by signals inside critical section and threads waking up to test mutex before it gets released.
Alerts

Thread state contains flag, alert-pending
Exception alerted
Alert (thread)
  alert-pending to true, wakeup a waiting thread
AlertWait (mutex, condition)
  if alert-pending set to false and raise exception
  else wait as usual
Boolean b = TestAlert()
  tests and clear alert-pending

TRY
  while (empty)
    AlertWait (m, nonempty);
    return (nextchar());
EXCEPT
  Thread.Alerted:
    return (eof);

Using Alerts

sibling = Fork (proc, arg);
while (!done)
{ done = longComp();
  if (done) Alert (sibling);
  else done = TestAlert();
}
Wisdom

Do's
- Reserve using alerts for when you don't know what is going on
- Only use if you forked the thread
- Impose an ordering on lock acquisition
- Write down invariants that should be true when locks aren't being held

Don't's
- Call into a different abstraction level while holding a lock
- Move the "last" signal beyond scope of Lock
- Acquire lock, fork, and let child release lock
- Expect priority inheritance since few implementations
- Pack data and expect fine grain locking to work

Practice: Bridge Problem

Synchronize traffic over a narrow light-duty bridge on a public highway. Traffic may only cross the bridge in one direction at a time, and if there are ever more than 3 vehicles on the bridge at one time, it will collapse under their weight. Each car is to be represented by one thread, which executes the procedure OneVehicle in order to cross the bridge:

```java
OneVehicle(int direc) //direc is either 0 or 1;
    //giving the direction in which the car is to cross
{
    ArriveBridge(direc);
    CrossBridge(direc);
    ExitBridge(direc);
}
```
Practice: Fine grain locking

Multiple threads inserting (and deleting) in a linked list

Solution Trick: Lock-Coupling

```c
void InsertList (int key)
{
    firstlock.Acquire();
    if (key <= first->data) {//insert as first, special case
        firstlock.Release(); return;
    } else{
        ptr = first;
        ptr->lock.Acquire(); firstlock.Release(); next=ptr->next;
        while (next != null & key > next->data)    // Next isn't locked - why OK??
            {next->lock.Acquire(); ptr->lock.Release(); ptr=next;
             next=ptr->next;}
        if (next == null) {//append after ptr; unlock ptr;}
        else if (key <= next->data)
                    {//insert between ptr and next; unlock ptr;}
            }
    }
}
```
Eraser: A dynamic data race detector for multithreaded programs

S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson
TOCS Nov. 97

Overview

- Dynamic data race detection tool – testing paradigm instead of static analysis.
- Checks that each shared memory access follows a consistent locking discipline
- Data race – when 2 concurrent threads access a shared variable and at least one is a write and the threads use no explicit synchronization to prevent simultaneous access.
  - Effect will depend on interleaving
Previous Approaches: Lamport’s Happened-Before

Previous work
• If 2 threads access a shared variable and the accesses are not ordered by happens-before then potential race.

Drawbacks of Happened-Before
• Difficult to implement efficiently – need per-thread information about access ordering to all shared memory locations.
• Highly dependent on scheduler – needs large number of test cases.
Previous work

- If 2 threads access a shared variable and the accesses are not ordered by *happens-before* then potential race.
- Depends on scheduler

```plaintext
y=y+1;
lock(mutex)
↓
v = v+1;
unlock(mutex)
lock(mutex)
↓
v = v+1;
unlock(mutex)
y=y+1;
```

```plaintext
lock(mutex)
↓
v = v+1;
unlock(mutex)
y=y+1;
```

```plaintext
y=y+1;
lock(mutex)
↓
v = v+1;
unlock(mutex)
```
Idea in Eraser

- Checks that locking discipline is observed.
  - That the same lock(s) is held whenever the shared data object is accessed.
  - Infer which locks protect which data items

Lockset Algorithm

- C(v) – candidate locks for v
- locks-held(t) – set of locks held by thread t
- Lock refinement

For each v, init C(v) to set of all locks

On each access to v by thread t:

\[ C(v) = C(v) \cap \text{locks-held}(t) \]

If \( C(v) = \{\} \) issue warning
Example

<table>
<thead>
<tr>
<th>locks-held</th>
<th>C(v)</th>
</tr>
</thead>
<tbody>
<tr>
<td>{}</td>
<td>{mu1, mu2}</td>
</tr>
<tr>
<td>{mu1}</td>
<td>{mu1}</td>
</tr>
<tr>
<td>{mu2}</td>
<td>{}</td>
</tr>
</tbody>
</table>

More Sophistication

- Initialization without locks
- Read-shared data (written only during init, read-only afterwards)
- Reader-writer locking (multiple readers)
- False Alarms still possible

- Don’t start until see a second thread
- Report only after it becomes write shared
- Change algorithm to reflect lock types
  - On read of v by t:
    \[ C(v) = C(v) \cap \text{locks-held}(t) \]
  - On write of v by t:
    \[ C(v) = C(v) \cap \text{write-locks-held}(t) \]
Per-Location State

- **virgin**
  - *wr, 1\(^{st}\) thread*

- **exclusive**
  - *rd, wr, 1\(^{st}\) thread*
  - *wr, new thread*

- **shared**
  - *rd, new thread*

- **Shared modified**
  - *wr*

  - *C(v) updated, races reported*

  - *C(v) updated, no race reporting*

Implementation

- Binary rewriting used
  - Add instrumentation to call Eraser runtime
  - Each load and store updates C(v)
  - Each Acquire and Release call updates locks-held(t)
  - Calls to storage allocator initializes C(v)

- Storage explosion handled by table lookup and use of indexes to represent sets
  - Shadow word holds index number

- Slowdown by factor of 10 to 30x
  - Will change interleavings
Shadow Memory and Lockset Indexes

Common False Alarms - Annotations

- Memory reuse
- Private locks
- Benign races

```c
if (some_condition)
    LOCK m DO
    if (some_condition)
        (stuff)
    END
END
```

- EraseReuse – resets shadow word to virgin state
- Lock annotations
- EraserIgnoreOn()
- EraserIgnoreOff()
Races inside OS

• Using interrupt system to provide mutual exclusion – this implicitly locks everything affected (by interrupt level specified)
  – Explicitly associate a lock with interrupt level – disabling interrupt is like acquiring that lock
• Signal and wait kind of synchronization
  – V to signal for P which waits -- semaphore not “held” by thread.

An OK Race in AltaVista

```c
if (p->ip_fp == (NI2_XFILE *) 0) {
    NI2LOCKS_LOCK (&p->ip_lock);
    if (p->ip_fp == (NI2_XFILE *) 0) {
        p->ip_fp = ni2_xfopen (
            p->ip_name, “rb”);
    }
    NI2LOCKS_UNLOCK (&p->ip_lock);
}
... // no locking overhead if file
     // pointer is already set
```
Bad Race in Vesta

Combine: XorFPTag::FPVal() {
    if (!this->validFP) {
        // is fingerprint marked valid?
        // no? calculate fingerprint
        NamesFP(fps, bv, this->fp, imap);
        // (NamesFP changes this->fp)
        this->validFP = true;
    }
    return this->fp;
}

This is a serious data race, since in the absence of memory barriers the
Alpha semantics does not guarantee that the contents of the validFP field
are consistent with the fp field.

Core Loop of Lock-Coupling

// ptr->lock.Acquire(); has been done before loop

while (next !null & key > next->data)
    {next->lock.Acquire();
    ptr->lock.Release();
    ptr=next;
    next=ptr->next;
    }

ptr  next