Mutual Exclusion: Some History, Some Problems, and a Glimmer of Hope

Andrew Birrell
Michael Isard

Microsoft Research, Silicon Valley

Outline

- Goal: create concurrent programs that
  - Are correct
  - Perform well
  - Remain that way a few years later
- This talk:
  - What is this "concurrency" thing?
  - How did we get here?
  - What's wrong?
  - Where can we go instead?

Not Included

- Creating concurrency
- Proving stuff
- Concurrency without shared memory
  - E.g., functional languages, some hardware designs

What is "Concurrency"?

- Program doing multiple things at once
- Two cases:
  - (A) Fake: waiting for a file read, so do something else
  - (B) Real: multiple processors sharing resources
    - Note: disk with DMA is effectively a processor
- (A) is much easier: no arbitrary inter-leavings
  - Sequential events, or non-preemptive threads
  - So this talk is mostly about (B) … but not entirely

Dijkstra (1960's version)

- Abstraction: multi-processor shared memory. Only considering case (B).
- Using atomic read/write (CACM 1965)
- Using ParBegin/Semaphore/P/V (CACM 1968)
- A few provable results but no higher-level abstractions.
- Requires a Ph.D. from a good university.

Hansen and Hoare (1974 version)

- Abstraction: sequential processes and monitors
  - ("condition variable" for in-process blocking)
- Monitor ties mutual exclusion to data:
  - Programmer groups shared data into monitors
  - System guarantees mutual exclusion per monitor
- Programmer must say what needs protected
- Programmer must maintain monitor invariants
- Programmer must layer program hierarchically
- Requires a Ph.D. (but not such a good one)

Hoare (1978 version)

- Abstraction: sequential state machines passing messages (Communicating Sequential Processes).
- Digression: duality (Lauer & Needham 1979)
  - Mapping between CSP-like and Monitor-like program
  - Programming difficulties map across, too.
- Works well with sequential machines (if DAG)
- Doesn't help when using shared memory

Modula-2+ (1984)

- Threads and locks (mutex): not monitors
- Abandoned link between mutex and its data
  - No enforced mutual exclusion at all
- Retained the problems of Hoare monitors
- SRC wrote 1 million lines of concurrent program
  - It mostly worked
  - But we (almost) all had Ph.D.s

Remainder of 19xx

- SRC solution was widely adopted:
  - OSF DCE
  - Posix
  - Windows (somewhat)
  - Java
  - C#
- It's easy: to describe; to use; to get wrong
- "Introduction to Programming with Threads"
- Mostly uni-processors; programs mostly work

Reprise: "classic" thread/mutex/CV

- VAR t = Fork(method, args)
  - "method(args)" executes in new asynchronous thread
- LOCK m {…… }
  - TRY Acquire(m); …… FINALLY Release(m);
- UNTIL b DO Wait(m, cv);
  - UNTIL b DO

    TRY Atomically { Release(m); P(cv); }
    FINALLY Acquire(m);

Variant: Event-Based Programs

- System invokes a method for incoming "event"
- Event executes to completion
- "wait" is replaced by event creation. E.g.:
  - Initiate a file read, then exit from event
  - Later, file completion event gets handled
- When waiting, the event-handler state machine replaces state held in thread stacks
- If concurrent, still needs mutual exclusion
  - Problems are identical to classic thread/mutex/CV

Variant: NT's "Completion Port" (1996)

- System maintains a pool of ready threads
- When event arrives, dispatches it to a thread
  - Subject to not exceeding desired concurrency level
- If thread blocks, dispatches another event
  - Maintains real-time concurrency level
- Low-level implementation, tied to scheduler
- Primary mechanism for "event-based" programming in Windows
- Leaves mutual exclusion problems untouched

So, What's Wrong?

- Manual selection of mutual exclusion:
  - Default is too little (and hence races)
  - Easy fix is too much (deadlocks or blank stares)
- Projects don't create hierarchical abstractions
  - Can't decide and/or maintain acyclic locking order
- "Composition" requires entire new abstractions
- "Clever" optimizations aren't maintainable
  - And are often wrong
- "Stack-ripping" in event-based programs

Locking Order Issues

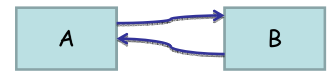
- Class "A":

```
FUNCTION f1() {
    LOCK a { … b.f4(); }
FUNCTION f2() {
    LOCK a { … }; }
```

- Class "B":

```
FUNCTION f3() {
    LOCK b { … a.f2(); }
FUNCTION f4() {
    LOCK b { … }; }
```

- Abstractions and call-graph:



- Caused by cyclic dependencies
- Abstractions should form a DAG
  - Difficult, in general
  - Hard to maintain

Composition Problems with Monitors

- Consider a hash table class with operations:

```
h.insert(k,v);
v = h.read(k);
h.delete(k);
```

- Consider client layering a "move" operation:

```
h.move(k, g) = {
    VAR v = h.read(k); g.insert(k,v); h.delete(k); }
```

- How does client make "move" atomic?

Cleverness: Double-Check Locking

- "Initialize-on-first-use" paradigm:

```
VAR v = NULL;

FUNCTION  ensureInitialized() {
  IF (v == NULL) THEN {
          LOCK m {
                IF (v == NULL) THEN v = NEW Obj();
          };
  };
}
```

Stack-Ripping

- In version 1.0 of a library:
  - "h.read(k)" accesses in-memory data structure
  - Non-blocking event code can call "h.read(k)" safely
- In version 2.0 of the library:
  - "h" has become big, now uses a B-Tree on disk
  - Calling pattern is now "h.startRead(k)", followed later by a completion event delivering the value.
  - Propagates to all callers, and their callers, …
- At best disruptive; often a performance bug

Transactions to the Rescue?

- Mark regions of your program as "atomic"
- System promises:
  - Concurrent transactions execute as if sequentially
  - Transactions really execute in parallel if possible
- Applies equally well to memory as to a database
- Software implementations today; hardware tomorrow (or so)
- Appealing simplicity
- Extremely limited experience with this usage

Mutual Exclusion with atomic blocks

- Thread A: ATOMIC { total = total – debit }
- Thread A: ATOMIC { total = total + debit }
- Removes locking order problems (largely)
- Composability/extension is easy
- Programmer still decides to protect things; simplest code still gets least protection
- Cleverness still not explicit, so not maintainable
- Doesn't help stack-ripping in event-based code

Fun: mix ATOMIC with non-atomic code

- Global: VAR x = 0; VAR shared = TRUE;
- Thread A:

  ATOMIC { x = 0; shared = FALSE };

  VAR temp = x;
- Thread B:

  ATOMIC { if (shared) { … ; x = 17; …… } }

Instead: Mostly-Sequential Programming

- Default to correctly synchronized programs
  - System provides the mutual exclusion
- Let the system do the optimization (mostly)
- Make programmer optimizations explicit
  - And, hopefully, therefore maintainable
- Some examples:
  - SQL query execution
  - The JavaScript part of AJAX
  - Map/Reduce or Dryad
  - AME (Automatic Mutual Exclusion)

Client-side JavaScript (in AJAX)

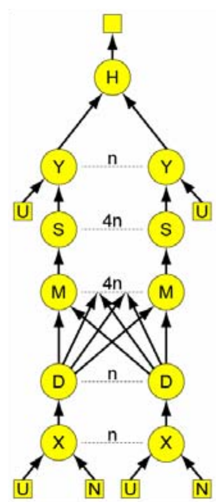
- Pure event-based programs
- Access to the server-side via XMLHttp:
  - Initiate async request
  - Sometime later response arrives as an event
- User interactions via “onclick” (etc.) events
- Single event at a time, execute to completion
- Screen gets updated only between events
  - i.e. UI updates look atomic

- Easy, works well, but very limited applicability

Dryad

- Programmer provides:
  - Sequential vertex programs (e.g. in C++)
  - Dataflow graph instantiating and connecting them
- System provides:
  - Scheduling of vertices onto processors (local or distributed)
  - Communication and synchronization
  - Fault tolerance

- Works beautifully, for the set of programs that fit this pattern
- Scales extremely well



AME (Automatic Mutual Exclusion)

- Everything is in transactions (almost)

- Execution = set of “asynchronous method calls”
  - “main” is the initial async method call
  - Program creates more by saying “ASYNC x.m(args)”
  - Forked calls execute iff this transaction commits

- System guarantees that:
  - Execution is a serialization of the async calls
  - The async calls execute in parallel if possible

AME: BlockUntil

- Within an async method can say "BlockUntil(b)"
- A transaction commits only if all its executed "BlockUntil" calls have the argument TRUE
- Otherwise, the transaction aborts, and retries later
- The system is responsible for wise scheduling of transaction (expression "b" is a good hint)

AME Example: Concurrent File Reading

```
void OpenRead(FileName name) {
  File f = AsyncOpenFile(name);
  async StartRead(f);
}

void StartRead(File f) {
  BlockUntil(f.Opened);
  g_nextOffset = 0;
  g_nextOffsetToEnqueue = 0;
  for (int i = 0; i < 4; ++i) {
    ReadBlock block = new ReadBlock;
    block.offset = g_nextOffset;
    block.file = f;
    g_nextOffset += block.size;
    f.StartAsyncRead(block);
    async WaitForBlock(block);
  }
}
```

```
void WaitForBlock(ReadBlock block) {
  BlockUntil(block.ready &&
        g_nextOffsetToEnqueue ==
            block.offset);
  if (block.EOF) {
    g_endOfFile = true;
  } else {
    g_queuedBlocks.PushBack(block);
    block.offset = g_nextOffset;
    g_nextOffset += block.size;
    block.file.StartAsyncRead(block);
    async WaitForBlock(block);
  }
  g_nextOffsetToEnqueue +=
        block.size;
}
```


AME: Yield

- A mechanism to allow intermediate commits:
  - Within an async method can say "Yield()"
  - Commits this transaction and starts a new one
- Program is now a set of "atomic fragments", and they're what gets serialized
- A general mechanism for allowing other transactions to make progress
  - "Yield(); BlockUntil(b)" is like "Wait(…)" in monitors
  - But also, solves the stack-ripping problem

Two (Separate) Examples Using "Yield"

```
void RunZombie() yields {
  Zombie z;
  z.Initialize();
  do {
    Yield();
    Time now = GetTimeNow();
    BlockUntil(now - z.lastUpdate >
                z.updateInterval);
    z.lastUpdate = now;
    MoveAround(z);
    if (Distance(z, g_player) <
        DeathRadius) {
      KillPlayer();
    }
  } while (Distance(z, g_player) >=
        DeathRadius);
}
```

```
void DoQueue(Queue inQ,
      Queue outQ) yields {
  do {
    Yield();
    BlockUntil(inQ.Length() > 0 ||
        g_finished);
    while (inQ.Length() > 0) {
      Item i = inQ.PopFront();
      async DoItem(i, outQ);
    }
  } while (!g_finished);
}

void DoItem(Item i,
      Queue outQ) yields {
  DoSlowProcessing(i);
  Yield();
  outQ.PushBack(i);
}
```

AME: Unprotected

- An async method can say "UNPROTECTED{ … }
  - Commit current transaction, then
  - Execute non-transacted code, then
  - Start a new transaction
- Use for code with side-effects (e.g I/O), or for calling legacy code
- Transacted state must be marshalled in/out
- Better default: dangerous code is labelled

Summary

- Existing mutual exclusion mechanisms are too difficult to use
- Transactions probably help
- Atomic blocks don't help enough
- AME:
  - Gets it right with little programmer assistance
  - High-enough level to allow a lot of optimization

Bibliography

- Historical papers:
  - http://birrell.org/andrew/concurrency/
- Dryad:
  - http://research.microsoft.com/research/sv/dryad/
- Automatic Mutual Exclusion:
  - http://research.microsoft.com/research/sv/ame/