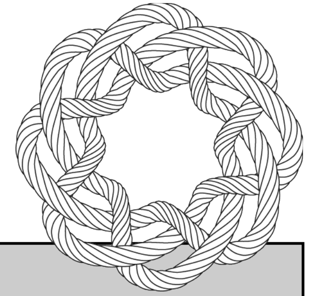
Concurrent Programming: Why and How?



Concurrent Programming

— Programs that do multiple things at once:

- Waiting for disk/network, but responsive to user
- Multiple processors sharing resources

Why Does It Matter; Why Now?

— Web-based services

- Simultaneous activity: user, client program, service
- Failure mode is blank stare when server is slow

— Multi-processors and multi-core

- CPU clock speed peaked at 3 GHz two years ago
- Future performance comes from concurrency
- Failure mode is we don't benefit from Moore's law

Concurrency 101: Shared Memory

— Multiple "threads" accessing same data structure

— Need "mutual exclusion" to preserve invariants

— Need wait/notify to coordinate use of resources

What's wrong

— Selection of mutual exclusion depends on programmer:

- Default is too little (and hence races)
- Easy fix is too much (deadlocks or blank stares)

— Big projects don't create hierarchical abstractions

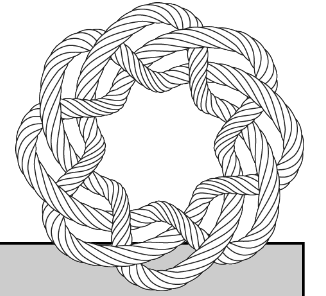
- Even if it's right in V1, it's hard to maintain later

— "Composition" requires entire new abstractions

— "Clever" optimizations aren't maintainable

— Blocking in an event disrupts program structure

Transactional Shared Memory



Transactions

— Invented for databases

— Programmer marks regions of the program as "atomic"

— System promises:

- Concurrent transactions execute as if sequentially
- Transactions really execute in parallel if possible

— Applies just as well to in-memory program execution as to updating a SQL database

— Software implementations today; hardware tomorrow

Obvious Use for Mutual Exclusion

— Thread A: ATOMIC { balance = balance - debit }

— Thread B: ATOMIC { balance = balance + credit}

Removes Some Problems:

— Locking order (somewhat)

— Composibility

Many Problems Remain:

— Programmer is still responsible for deciding what gets protected: default is nothing

— Performance incentive is to protect nothing

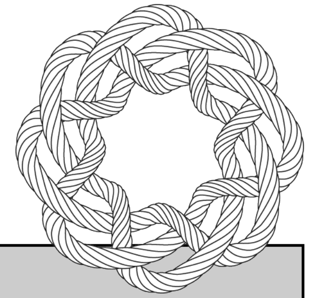
— Cleverness is still not explicit, and so not maintainable

— ATOMIC doesn't help event-based programs' stack-ripping problems

And Also:

— complex interaction between ATOMIC regions and non-ATOMIC regions
(because of pending abort or non-atomic commit)

A New Approach: Automatic Mutual Exclusion



Everything Is In Transactions

- Program executes set of "Asynchronous Method Calls"
- "main" is the first one
- Execution can create new async calls by saying
  - ASYNC x.m(args)
- Forked calls execute iff current transaction commits
- System guarantees:
  - Execution is a serialization of the async method calls
  - Async method calls execute in parallel if possible

Controlling the Schedule

- Async method call can say "BlockUntil(b)"
- Transaction commits only if all BlockUntil calls that it executes have argument "true"
- System uses abort/retry, optimized by when b changes

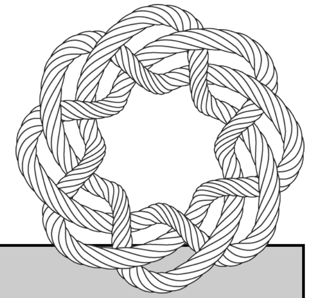
Intermediate Commits

- Async method can say "Yield()"
- Commits this transaction and starts a new one
- Really, program is set of "atomic fragments", and they are what gets serialized
- "Yields(); BlockUntil(b)" is like "Wait(...)" in monitors
- But "Yield()" by itself just lets another transaction make progress, by revealing intermediate results
- Lets us commit state or block with creating the event-based "stack-ripping" problem

Non-transacted Regions

- Async method can say "UNPROTECTED { ... }"
- Commit; do non-transacted code; start new transaction
- Use for side-effects (e.g. I/O) or calling legacy code
- Marshal transacted state in/out of UNPROTECTED

AME Examples



Asynchronous Reads from a File

```
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;
}
```

Independent Thread of Control

```
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);
}
```

Data-Parallel Computation

```
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);
}
```