# Transactional Memory
# Go

Chris Rossbach

cs378 Fall 2018

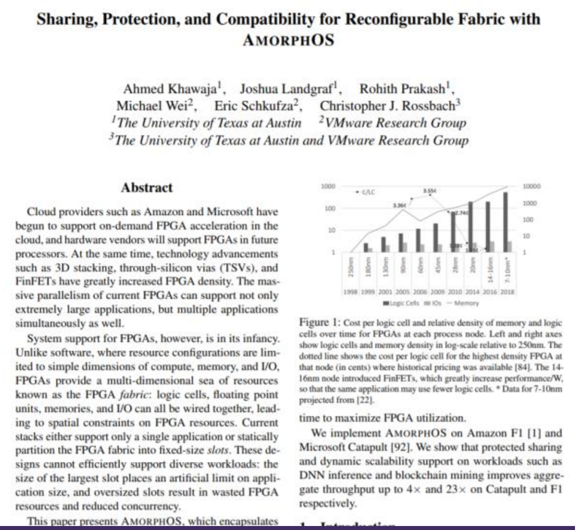
10/3/2018

# Outline for Today

- Questions?
- Administrivia
  - Lab 3 looms large: Go go go!
  - Next week: cameos by Keshav

- Agenda
  - Transactional Memory
  - Go

- Acks: Yoav Cohen for some STM slides
- Rob Pike's 2012 Go presentation is excellent, and I borrowed from it: https://talks.golang.org/2012/concurrency.slide


Sharing, Protection, and Compatibility for Reconfigurable Fabric with AMORPHOS

Ahmed Khawaja[1], Joshua Landgraf[1], Rohith Prakash[1], Michael Wei[2], Eric Schkufza[2], Christopher J. Rossbach[3]
[1]The University of Texas at Austin [2]VMware Research Group
[3]The University of Texas at Austin and VMware Research Group

Abstract

Cloud providers such as Amazon and Microsoft have begun to support on-demand FPGA acceleration in the cloud, and hardware vendors will support FPGAs in future processors. At the same time, technology advancements such as 3D stacking, through-silicon vias (TSVs), and FinFETs have greatly increased FPGA density. The massive parallelism of current FPGAs can support not only extremely large applications, but multiple applications simultaneously as well.

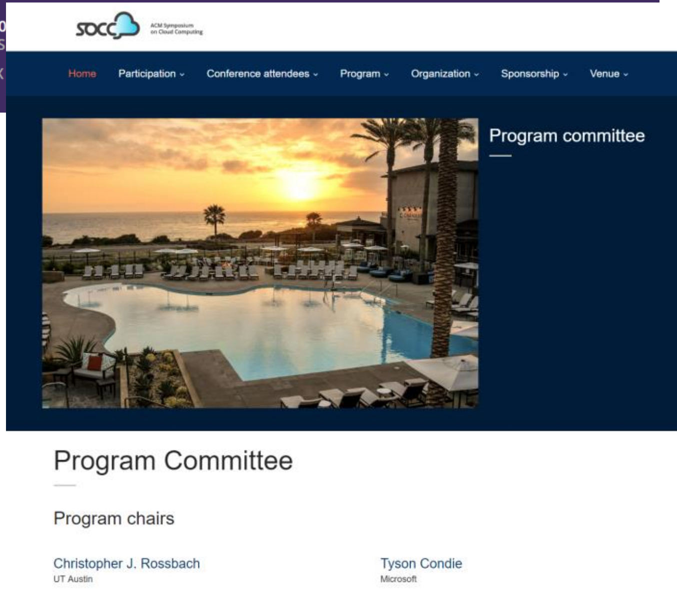
System support for FPGAs, however, is in its infancy. Unlike software, where resource configurations are limited to simple dimensions of compute, memory, and I/O, FPGAs provide a multi-dimensional sea of resources known as the FPGA *fabric*: logic cells, floating point units, memories, and I/O can all be wired together, leading to spatial constraints on FPGA resources. Current stacks either support only a single application or statically partition the FPGA fabric into fixed-size *slots*. These designs cannot efficiently support diverse workloads: the size of the largest slot places an artificial limit on application size, and oversized slots result in wasted FPGA resources and reduced concurrency.

This paper presents AMORPHOS, which encapsulates

Figure 1: Cost per logic cell and relative density of memory and logic cells over time for FPGAs at each process node. Left and right axes show logic cells and memory density in log-scale relative to 250nm. The dotted line shows the cost per logic cell for the highest density FPGA at that node (in cents) where historical pricing was available [84]. The 14-16nm node introduced FinFETs, which greatly increase performance/W, so that the same application may use fewer logic cells. * Data for 7-10nm projected from [22].

time to maximize FPGA utilization.

We implement AMORPHOS on Amazon F1 [1] and Microsoft Catapult [92]. We show that protected sharing and dynamic scalability support on workloads such as DNN inference and blockchain mining improves aggregate throughput up to 4× and 23× on Catapult and F1 respectively.

# Faux Quiz questions

- How does HTM resemble or differ from Load-linked Stored-Conditional?
- What are some pros and cons of HTM vs STM?
- What is Open Nesting? Closed Nesting? Flat Nesting?
- How are promises and futures different or the same as goroutines
- What is the difference between a goroutine and a thread?
- What is the difference between a channel and a lock?
- How is a channel different from a concurrent FIFO?
- What is the CSP model?
- What are the tradeoffs between explicit vs implicit naming in message passing?
- What are the tradeoffs between blocking vs. non-blocking send/receive in a shared memory environment? In a distributed one?

# Transactional Memory: ACI

Transactional Memory :

- Make multiple memory accesses atomic
- All or nothing – Atomicity
- No interference – Isolation
- Correctness – Consistency
- No durability, for obvious reasons

- Keywords : Commit, Abort, Speculative access,
  Checkpoint

```
remove(list, x) {
  lock(list);
  pos = find(list, x);
  if(pos)
     erase(list, pos);
  unlock(list);
}
```

```
remove(list, x) {
  TXBEGIN();
  pos = find(list, x);
  if(pos)
     erase(list, pos);
  TXEND();
}
```

# The **Real** Goal

```
remove(list, x) {
  atomic {
    pos = find(list, x);
    if(pos)
      erase(list, pos);
  }
}
```

- Transactions: super-awesome
- Transactional Memory: also super-awesome, ***but***:
- Transactions != TM
- TM is an ***implementation technique***
- Often presented as programmer abstraction
- Remember Optimistic Concurrency Control

```
remove(list, x) {
  lock(list);
  pos = find(list, x);
  if(pos)
    erase(list, pos);
  unlock(list);
```

```
remove(list, x) {
  TXBEGIN();
  pos = find(list, x);
  if(pos)
    erase(list, pos);
  TXEND();
}
```

# TM Primer

**Key Ideas:**

- Critical sections execute concurrently
- Conflicts are detected dynamically
- If conflict serializability is violated, rollback

**Key Abstractions:**

- Primitives
  - **xbegin, xend, xabort**
- Conflict

$$\varnothing \neq \{W_a\} \cap \{R_b \cup W_b\}$$

- Contention Manager
  - Need flexible policy

# TM basics: example

cpu 0

PC: 0

Working Set
R{}
W{}

```
0: xbegin
1: read A
2: read B
3: if(cpu % 2)
4:   write C
5: else
6:   read C
7: …
8: xend
```

```
0: xbegin;
1: read A
2: read B
3: if(cpu % 2)
4:   write C
5: else
6:   read C
7: …
8: xend
```

cpu 1

PC: 8

Working Set
R{}
W{}

CONFLICT:

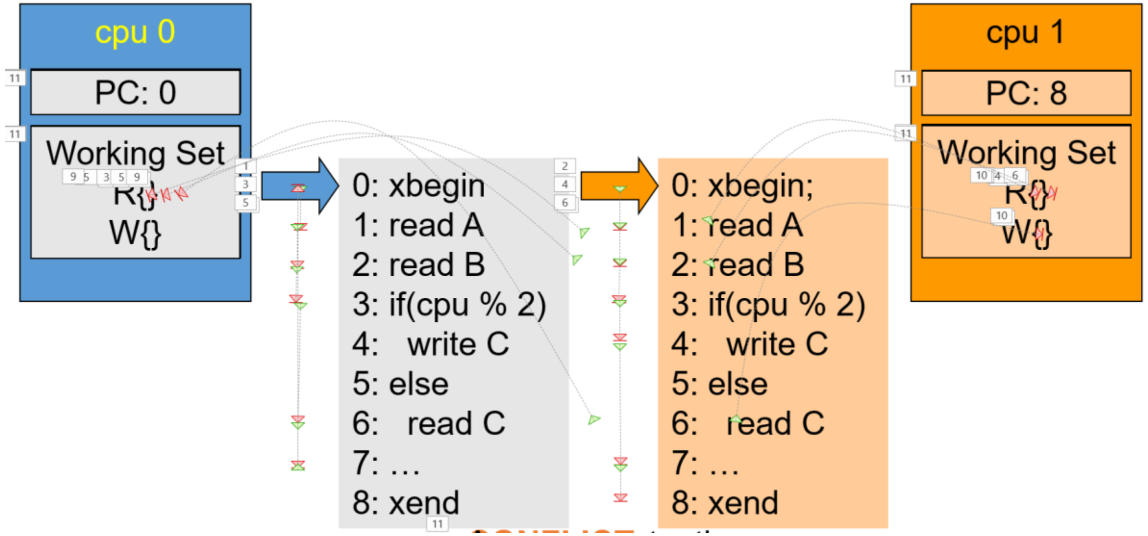
C is in the read set of cpu0, and in the write set of cpu1

Assume contention manager decides cpu1 wins:

cpu0 rolls back

cpu1 commits

# TM Implementation




## Data Versioning

- Eager Versioning
- Lazy Versioning

## Conflict Detection and Resolution

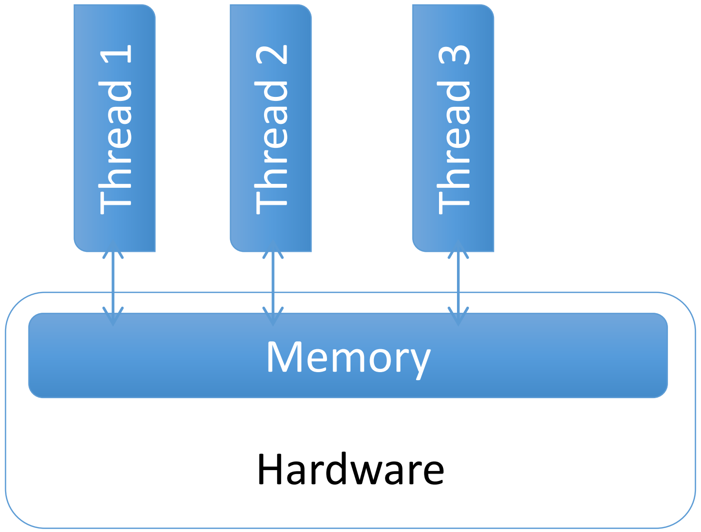
- Pessimistic Concurrency Control
- Optimistic Concurrency Control

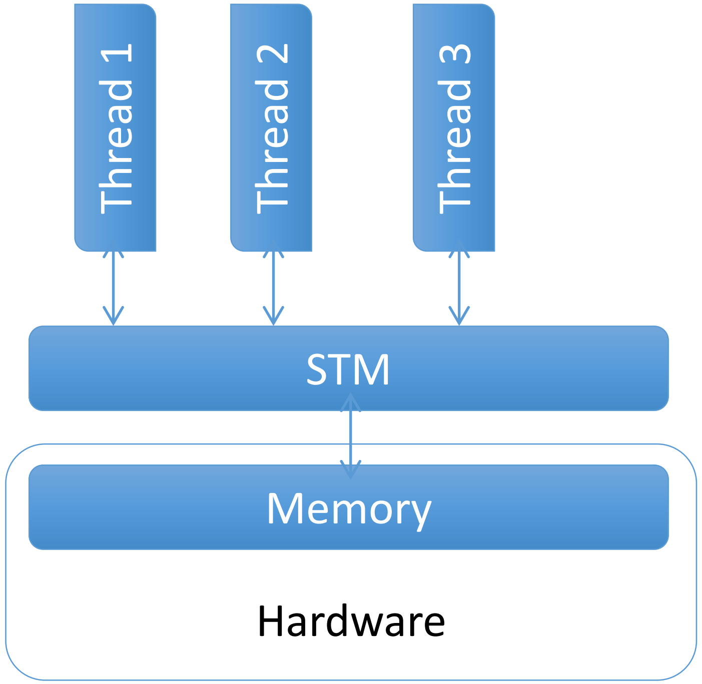
## Conflict Detection Granularity

- Object Granularity
- Word Granularity
- Cache line Granularity

# TM Design Alternatives

- Hardware (HTM)
  - Caches track RW set, HW speculation/checkpoint
- Software (STM)
  - Instrument RW
  - Inherit TX Object



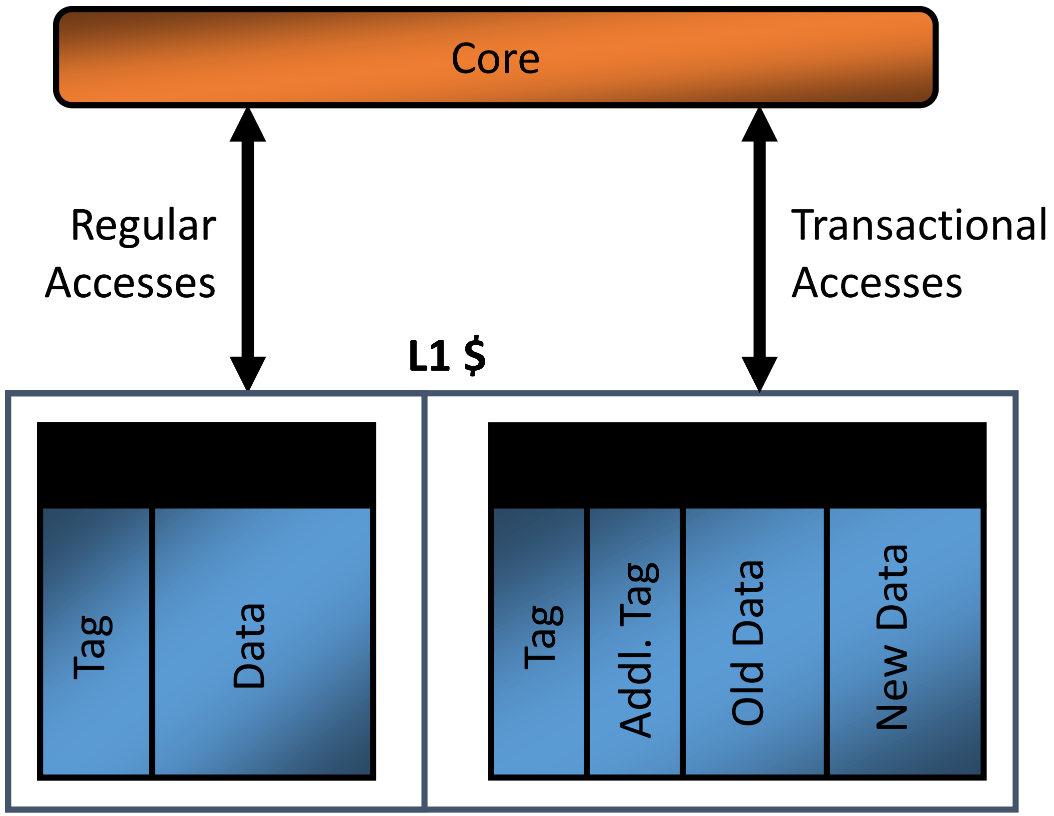



# Hardware Transactional Memory

- Idea: Track read / write sets in HW
  - commit / rollback in hardware as well
- Cache coherent hardware already manages much of this
- Basic idea: cache == speculative storage
  - HTM ~= smarter cache
- Can support many different TM paradigms
  - Eager, lazy
  - optimistic, pessimistic

# Hardware TM

- “Small” modification to cache

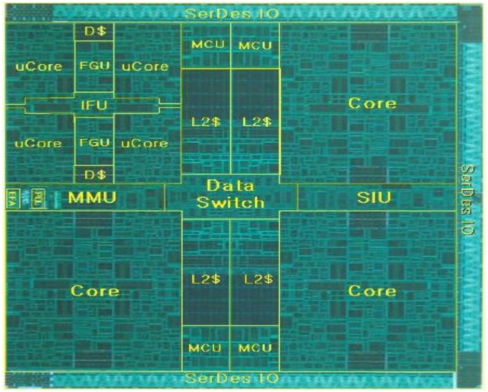



*Key ideas*

- *Checkpoint architectural state*
- *Caches: ‘versioning’ for memory*
- *Change coherence protocol*
- *Conflict detection in hardware*
- *‘Commit’ transactions if no conflict*
- *‘Abort’ on conflict (or special cond)*
- *‘Retry’ aborted transaction*

Pros/Cons?

# Case Study: SUN Rock




- Major challenge: diagnosing cause of Transaction aborts
  - Necessary for intelligent scheduling of transactions
  - Also for debugging code
  - debugging the processor architecture / µarchitecture
- Many unexpected causes of aborts
- Rock v1 diagnostics unable to distinguish distinct failure modes

| Mask | Name | Description and example cause |
|---|---|---|
| 0x001 | EXOG | **Exogenous** - Intervening code has run: cps register contents are invalid. |
| 0x002 | COH | **Coherence** - Conflicting memory operation. |
| 0x004 | TCC | **Trap Instruction** - A trap instruction evaluates to "taken". |
| 0x008 | INST | **Unsupported Instruction** - Instruction not supported inside transactions. |
| 0x010 | PREC | **Precise Exception** - Execution generated a precise exception. |
| 0x020 | ASYNC | **Async** - Received an asynchronous interrupt. |
| 0x040 | SIZ | **Size** - Transaction write set exceeded the size of the store queue. |
| 0x080 | LD | **Load** - Cache line in read set evicted by transaction. |
| 0x100 | ST | **Store** - Data TLB miss on a store. |
| 0x200 | CTI | **Control transfer** - Mispredicted branch. |
| 0x400 | FP | **Floating point** - Divide instruction. |
| 0x800 | UCTI | **Unresolved control transfer** - branch executed without resolving load on which it depends |

**Table 1.** cps register: bit definitions and example failure reasons that set them.

# A Simple STM

```
pthread_mutex_t g_global_lock;

begin_tx() {
    pthread_mutex_lock(g_global_lock);
}

end_tx() {
    pthread_mutex_unlock(g_global_lock);
}

abort() {
    // can't happen
}
```

```
remove(list, x) {
  begin_tx();
  pos = find(list, x);
  if(pos)
     erase(list, pos);
  end_tx();
}
```

Is this Transactional Memory?

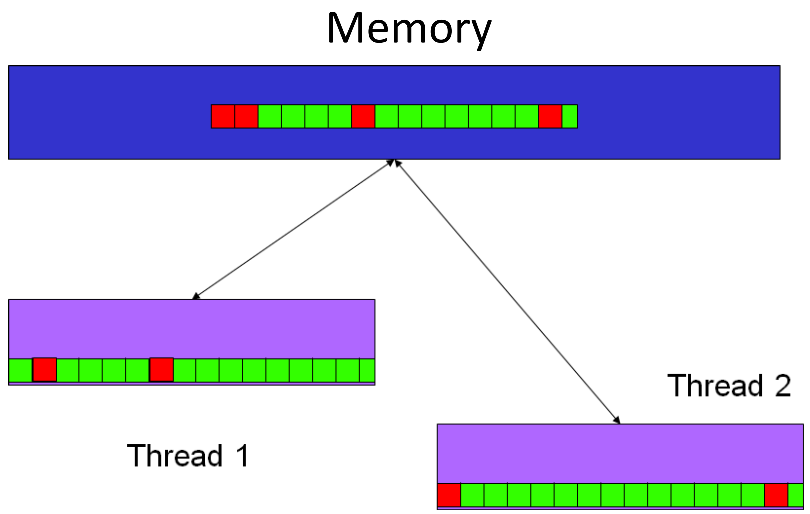
TM is a deep area: consider it for your project!

# A Better STM: System Model

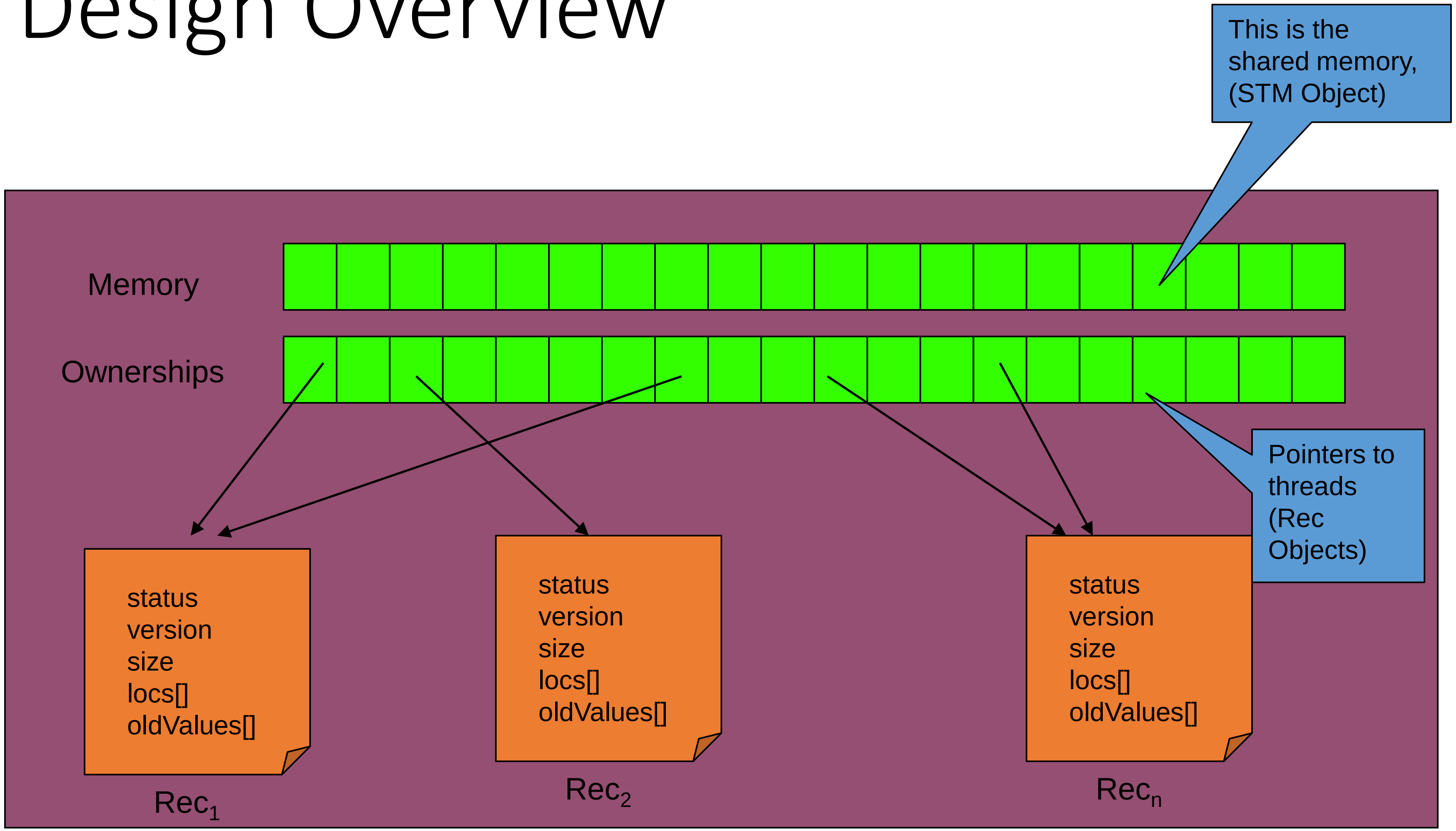
System == <threads, memory>

Memory cell support 4 operations:

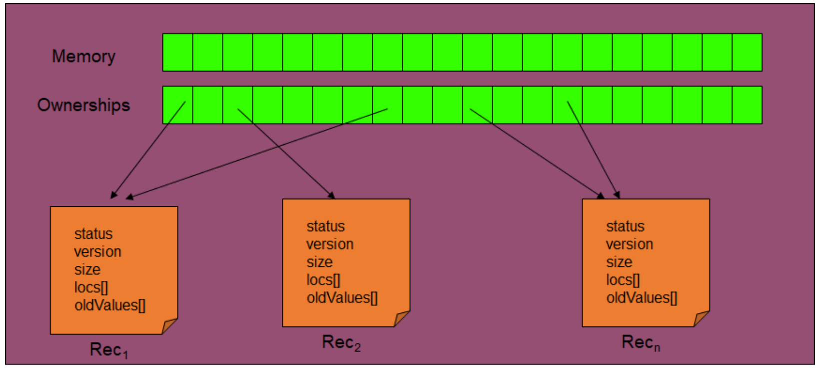
- $Write^i(L,v)$ - *thread i writes v to L*
- $Read^i(L,v)$ - *thread i reads v from L*
- $LL^i(L,v)$ - *thread i reads v from L, marks L read by I*
- $SC^i(L,v)$ - *thread i writes v to L*
  - returns *success* if L is marked as read by i.
  - Otherwise it returns *failure*.



# STM Design Overview



# Threads: Rec Objects

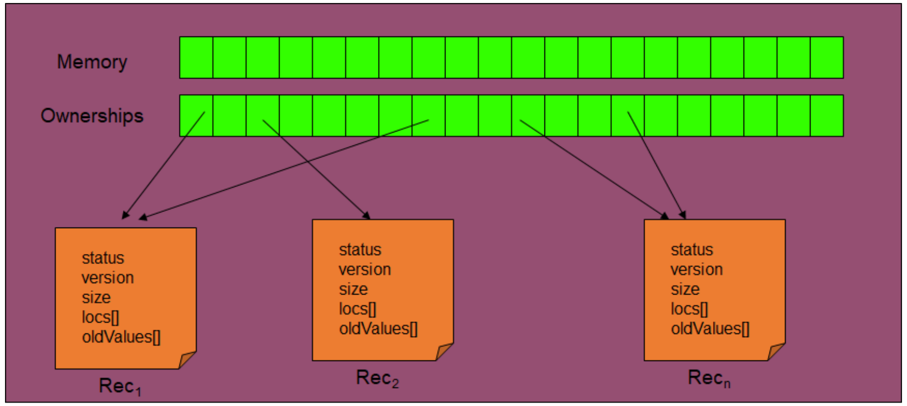



```
class Rec {
	boolean stable = false;
	boolean, int status= (false,0);  //can have two values…
	boolean allWritten = false;
	int version = 0;
	int size = 0;
	int locs[] = {null};
	int oldValues[] = {null};
}
```

Each thread → instance of Rec class *(short for record).*

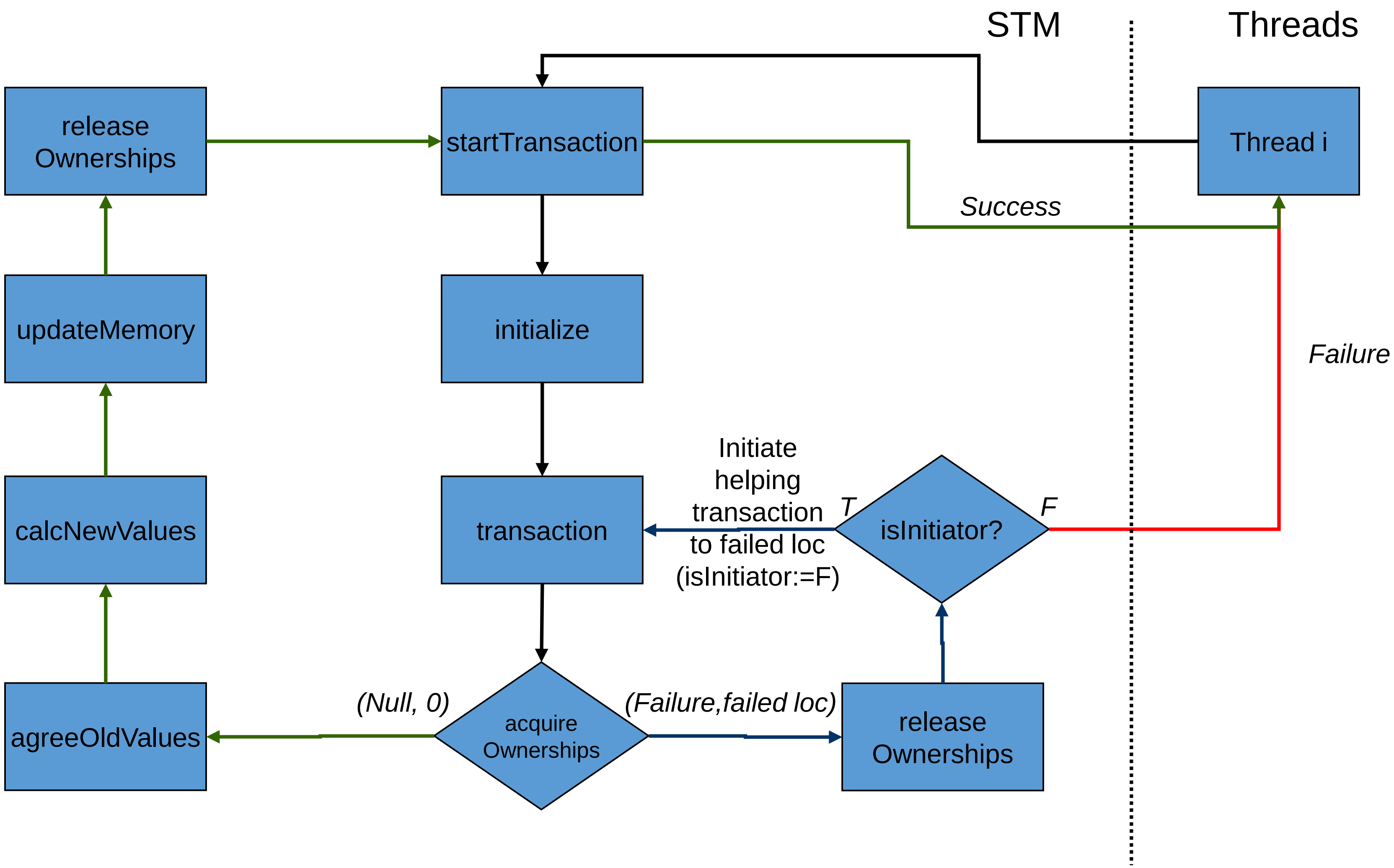
Rec instance defines current transaction on thread

# Memory: STM Object




```
public class STM {
        int memory[];
        Rec ownerships[];

        public boolean, int[] startTranscation(Rec rec, int[] dataSet){...};

        private void initialize(Rec rec, int[] dataSet)
        private void transaction(Rec rec, int version, boolean isInitiator) {...};
        private void acquireOwnerships(Rec rec, int version) {...};
        private void releaseOwnerships(Rec rec, int version) {...};
        private void agreeOldValues(Rec rec, int version) {...};
        private void updateMemory(Rec rec, int version, int[] newvalues) {...};
}
```

# Flow of a transaction



# Implementation

```
public boolean, int[] startTransaction(Rec rec, int[] dataSet) {
        initialize(rec, dataSet);
        rec.stable = true;
        transaction(rec, rec.version, true);
        rec.stable = false;
        rec.version++;
        if (rec.status) return (true, rec.oldValues);
        else return false;
}
```

rec – The thread that executes this transaction.
dataSet – The location in memory it needs to own.

This notifies other threads that I can be helped

# Implementation

```
private void transaction(Rec rec, int version, boolean isInitiator) {
        acquireOwnerships(rec, version); // try to own locations

        (status, failedLoc) = LL(rec.status);
        if (status == null) {                    // success in acquireOwnerships
                if (version != rec.version) return;
                SC(rec.status, (true,0));
        }

        (status, failedLoc) = LL(rec.status);
        if (status == true) {                    // execute the transaction
                agreeOldValues(rec, version);
                int[] newVals = calcNewVals(rec.oldvalues);
                updateMemory(rec, version);
                releaseOwnerships(rec, version);
        }
        else {                                   // failed in acquireOwnerships
                releaseOwnerships(rec, version);
                if (isInitiator) {
                        Rec failedTrans = ownerships[failedLoc];
                        if (failedTrans == null) return;
                        else {                          // execute the transaction that owns the location you want
                                int failedVer = failedTrans.version;
                                if (failedTrans.stable) transaction(failedTrans, failedVer, false);
                        }
                }
        }
}
```

rec – The thread that executes this transaction.
version – Serial number of the transaction.
isInitiator – Am I the initiating thread or the helper?

Another thread own the locations I need and it hasn't finished its transaction yet.

So I go out and execute its transaction in order to help it.

# Implementation

```
private void acquireOwnerships(Rec rec, int version) {
        for (int j=1; j<=rec.size; j++) {
                while (true) do {
                        int loc = locs[j];
                        if LL(rec.status) != null return;    // transaction completed by some other thread
                        Rec owner = LL(ownerships[loc]);
                        if (rec.version != version) return;
                        if (owner == rec) break; // location is already mine
                        if (owner == null) {        // acquire location
                                if ( SC(rec.status, (null, 0)) ) {
                                  if ( SC(ownerships[loc], rec) ) {
                                    break;
                                  }
                                }
                        }
                        else {// location is taken by someone else
                                if ( SC(rec.status, (false, j)) ) return;
                        }

                }

        }
}
```

If I'm not the last one to read this field, it means that another thread is trying to execute this transaction. Try to loop until I succeed or until the other thread completes the transaction

# Implementation

```
private void agreeOldValues(Rec rec, int version) {
        for (int j=1; j<=rec.size; j++) {
                int loc = locs[j];
                if ( LL(rec.oldvalues[loc]) != null ) {
                        if (rec.version != version) return;
                        SC(rec.oldvalues[loc], memory[loc]);
                }
        }
}
```

Copy the dataSet to my private space

```
private void updateMemory(Rec rec, int version, int[] newvalues) {
        for (int j=1; j<=rec.size; j++) {
                int loc = locs[j];
                int oldValue = LL(memory[loc]);
                if (rec.allWritten) return;    // work is done
                if (rec.version != version) return;
                if (oldValue != newValues[j]) SC(memory[loc], newValues[j]);
        }
        if (! LL(rec.allWritten) ) {
             if (rec.version != version) SC(rec.allWritten, true);
        }
}
```

Selectively update the shared memory

# HTM vs. STM

| Hardware | Software |
| --- | --- |
| Fast (due to hardware operations) | Slow (due to software validation/commit) |
| Light code instrumentation | Heavy code instrumentation |
| HW buffers keep amount of metadata low | Lots of metadata |
| No need of a middleware | Runtime library needed |
| Only short transactions allowed (why?) | Large transactions possible |

How would you get the best of both?

# Hybrid-TM

- Best-effort HTM (use STM for long trx)
- Possible conflicts between HW, SW and HW-SW Txns
  - What kind of conflicts do  SW-Trx care about?
  - What kind of conflicts do  HW-Trx care about?
- Some proposals:
  - HyTM: uses an ownership record per memory location
  - PhTM: HTM-only or (heavy) STM-only, low instrumentation
  - TSX, PPC: fall-back to SGL

# Message passing

# ~~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
  - …

Remember this slide?

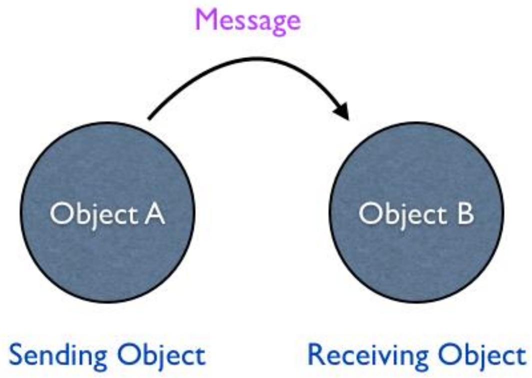
- Events: *restructure programming model to have no threads!*

# **Message Passing**: 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
  - …
- Message passing:
  - *Threads aren't the problem, shared memory is*
  - *restructure programming model to avoid communication through shared memory (and therefore locks)*

# Message Passing

- Threads/Processes send/receive messages
- Three design dimensions
  - Naming/Addressing: *how do processes refer to each other?*
  - Synchronization: *how to wait for messages (block/poll/notify)?*
  - Buffering/Capacity: *can messages wait in some intermediate structure?*



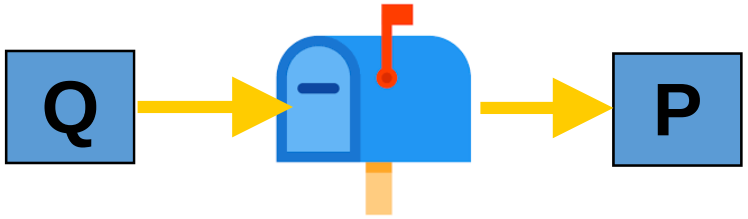
Message Passing

# Naming: Explicit vs Implicit

Also: Direct vs Indirect

- Explicit Naming
  - Each process must explicitly name the other party
  - Primitives:
    - send(receiver, message)
    - receive(sender, message)


- Implicit Naming
  - Messages sent/received to/from mailboxes
  - Mailboxes may be named/shared
  - Primitives:
    - send(mailbox, message)
    - receive(mailbox, message)



# Synchronization

- Synchronous vs. Asynchronous
  - Blocking send: sender blocks until received
  - Nonblocking send: send resumes before message received
  - Blocking receive: receiver blocks until message available
  - Non-blocking receive: receiver gets a message or null
- If ***both send and receive block***
  - "Rendezvouz"
  - Operation acts as an ordering primitive
  - Sender knows receiver succeded
  - Receiver knows sender succeeded
  - Particularly appealing in distributed environment

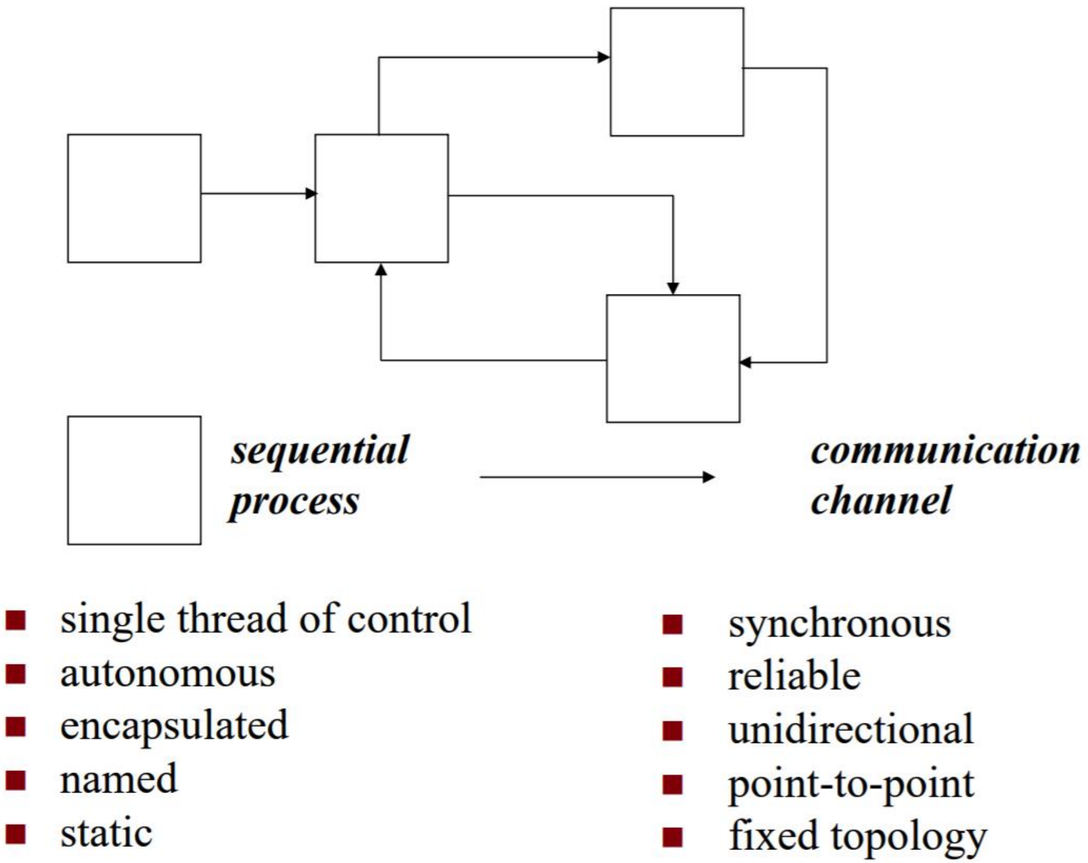
***To block…***
***or not to block:***
Which is better?

Blocking:
+ simple
+ avoids wasteful spinning
- Inflexible
- Can hide concurrency

Non-blocking:
+ maximal flexibility
- error handling/detection tricky
- interleaving useful work non-trivial

# Communicating Sequential Processes
## Hoare 1978

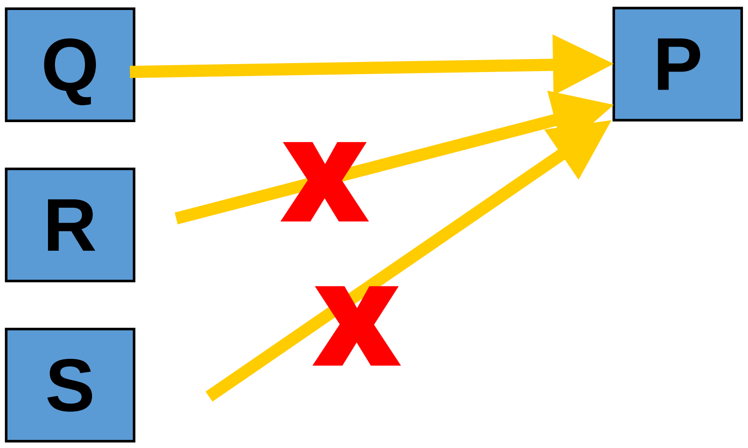
CSP: language for multi-processor machines
- Non-buffered **message passing**
  - No shared memory
  - **Send/recv are blocking**
- **Explicit naming** of src/dest processes
  - Also called direct naming
  - Receiver **specifies source** process
  - Alternatives: *indirect*
    - Port, mailbox, queue, socket
- **Guarded** commands to let processes wait




← Transputer!

# An important problem in the CSP model:

- Processes need to receive messages from different senders
- Only primitive: blocking receive(<name>, message)

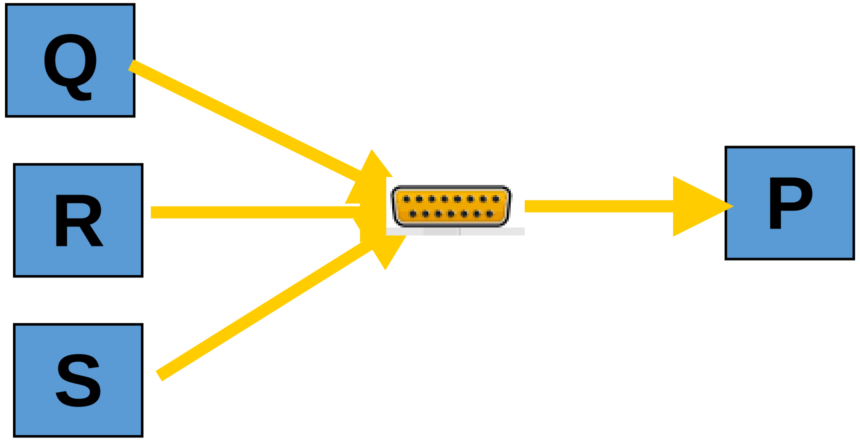



```
recv_multi(P) {
  receive(Q, message)
  receive(R, message)
  receive(S, message)
}
```

Is there a problem with this?

# Blocking with Indirect Naming

- Processes need to receive messages from different senders
- ***blocking receive*** with ***indirect naming***
  - Process waits on port, gets first message first message arriving at that port

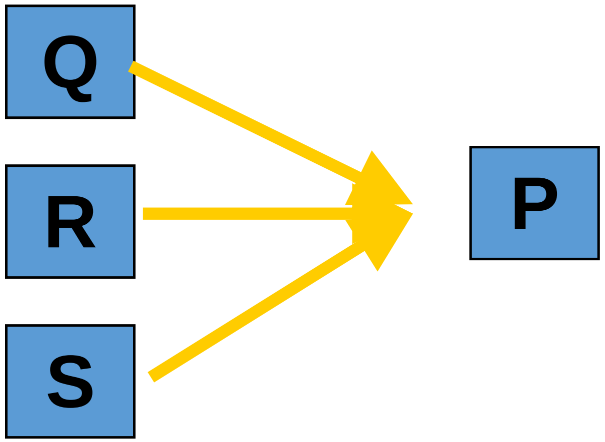



receive(port, message)

OK to block (good)
Requires indirection (less good)

# Non-blocking with Direct Naming

- Processes need to receive messages from different senders
- ***Non-blocking receive*** with ***direct naming***
  - Requires receiver to poll senders

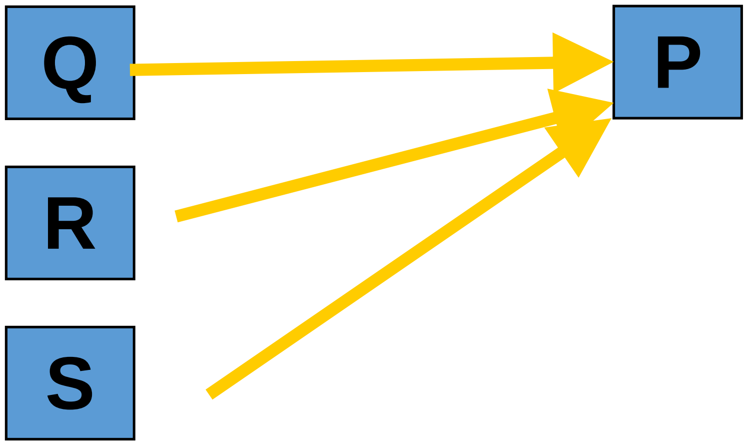



```
while(…) {
  try_receive(Q, message)
  try_receive(R, message)
  try_receive(S, message)
}
```

Polling (bad)
No indirection (good)

# Blocking and Direct Naming

- How to achieve ***it?***
- ***CSP provides abstractions/primitives for it***

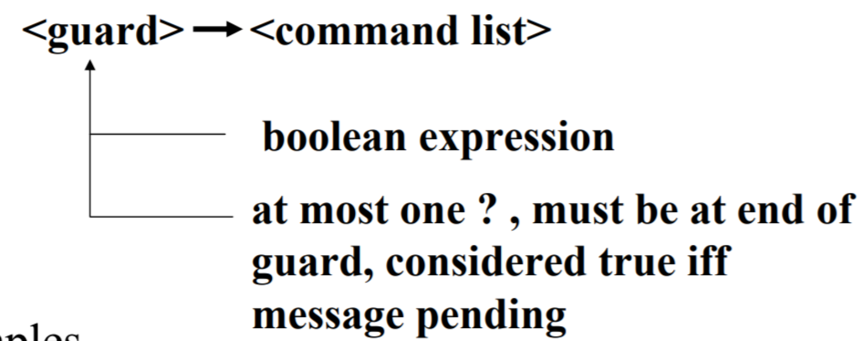


# Alternative / Guarded Commands

Guarded command is ***delayed*** until either

- ***guard succeeds*** → cmd executes *or*
- ***guard fails*** →command aborts

Guarded Commands




Examples

**n < 10 → A!index(n); n := n + 1;**
**n < 10; A?index(n) → next = MyArray(n);**

Alternative command:

- list of one or more guarded commands
- separated by ”||”
- surrounded by square brackets

**[ x ≥ y -> max:= x || y ≥ x -> max:= y ]**

- Enable *choice* preserving concurrency
- *Hugely influential*
- goroutines, channels, select, defer:
    - ***Trying to achieve the same thing***

# Go Concurrency

- CSP: the root of many languages
  - Occam, Erlang, Newsqueak, Concurrent ML, Alef, Limbo
- Go is a Newsqueak-Alef-Limbo derivative
  - Distinguished by ***first class channel support***
  - Program: ***goroutines*** communicating through ***channels***
  - Guarded and alternative-like constructs in ***select*** and ***defer***

# A boring function

```
func boring(msg string) {
    for i := 0; ; i++ {
        fmt.Println(msg, i)
        time.Sleep(time.Duration(rand.Intn(1e3)) * time.Millisecond)
    }
}
```

```
func main() {
    boring("boring!")
}
```

```
boring! 0
boring! 1
boring! 2
boring! 3
boring! 4
boring! 5
```

# Ignoring a boring function

- Go statement runs the function
- Doesn't make the caller wait
- Launches a goroutine
- Analagous to & on shell command

```
package main

import (
    "fmt"
    "math/rand"
    "time"
)

func main() {
    go boring("boring!")
}
```

```
I'm listening.
boring! 0
boring! 1
boring! 2
boring! 3
boring! 4
boring! 5
You're boring; I'm leaving.

Program exited.
```

- Keep main() around a while
- See goroutine actually running

```
func main() {
    go boring("boring!")
    fmt.Println("I'm listening.")
    time.Sleep(2 * time.Second)
    fmt.Println("You're boring; I'm leaving.")
}
```

# Goroutines

- Independently executing function launched by go statement
- Has own call stack
- Cheap: Ok to have 1000s…100,000s of them
- Not a thread
  - One thread may have **1000s** of go routines!
- Multiplexed onto threads as needed to ensure forward progress
  - Deadlock detection built in

*How do goroutines relate to promises & futures?*

# Channels

- Connect goroutines allowing th

- When main executes <-c, it blocks
- When boring executes c <- value it blocks
- Channels communicate *and **synchronize***

```
func main() {
    c := make(chan string)
    go boring("boring!", c)
    for i := 0; i < 5; i++ {
        fmt.Printf("You say: %q\n", <-c) // Receive expression is just a value.
    }
    fmt.Println("You're boring; I'm leaving.")
}
```

```
func boring(msg string, c chan string) {
    for i := 0; ; i++ {
        c <- fmt.Sprintf("%s %d", msg, i) // Expression to be sent can be any s
        time.Sleep(time.Duration(rand.Intn(1e3)) * time.Millisecond)
    }
}
```

```
You say: "boring! 0"
You say: "boring! 1"
You say: "boring! 2"
You say: "boring! 3"
You say: "boring! 4"
You're boring; I'm leaving.

Program exited.
```

# Select: Handling Multiple Channels

- All channels are evaluated
- Select blocks until one communication can proceed
  - Cf. Linux select system call, Windows WaitForMultipleObjectsEx
  - Cf. Alternatives and guards in CPS
- If multiple can proceed select chooses randomly
- Default clause executes immediately if no ready channel

```
select {
case v1 := <-c1:
    fmt.Printf("received %v from c1\n", v1)
case v2 := <-c2:
    fmt.Printf("received %v from c2\n", v1)
case c3 <- 23:
    fmt.Printf("sent %v to c3\n", 23)
default:
    fmt.Printf("no one was ready to communicate\n")
}
```

# Google Search

- Workload:
- Accept query
- Return page of results (with ugh, ads)
- Get search results by sending query to
    - Web Search
    - Image Search
    - YouTube
    - Maps
    - News, etc
- How to implement this?

# Search 1.0

- Google function takes query and returns a slice of results (strings)
- Invokes Web, Image, Video search serially

```
func Google(query string) (results []Result) {
    results = append(results, Web(query))
    results = append(results, Image(query))
    results = append(results, Video(query))
    return
}
```

# Search 2.0

- Run Web, Image, Video searches concurrently, wait for results
- No locks, conditions, callbacks

```
func Google(query string) (results []Result) {
    c := make(chan Result)
    go func() { c <- Web(query) } ()
    go func() { c <- Image(query) } ()
    go func() { c <- Video(query) } ()

    for i := 0; i < 3; i++ {
        result := <-c
        results = append(results, result)
    }
    return
}
```

# Search 2.1

- Don't wait for slow servers: No locks, conditions, callbacks!

```
c := make(chan Result)
go func() { c <- Web(query) } ()
go func() { c <- Image(query) } ()
go func() { c <- Video(query) } ()

timeout := time.After(80 * time.Millisecond)
for i := 0; i < 3; i++ {
    select {
    case result := <-c:
        results = append(results, result)
    case <-timeout:
        fmt.Println("timed out")
        return
    }
}
return
```

# Search 3.0

- Reduce tail latency with replication. No locks, conditions, callbacks!

```
c := make(chan Result)
go func() { c <- First(query, Web1, Web2) } ()
go func() { c <- First(query, Image1, Image2) } ()
go func() { c <- First(query, Video1, Video2) } ()
timeout := time.After(80 * time.Millisecond)
for i := 0; i < 3; i++ {
    select {
    case result := <-c:
        results = append(results, result)
    case <-timeout:
        fmt.Println("timed out")
        return
    }
}
return
```

```
func First(query string, replicas ...Search) Result {
    c := make(chan Result)
    searchReplica := func(i int) { c <- replicas[i](query) }
    for i := range replicas {
        go searchReplica(i)
    }
    return <-c
}
```

# Other tools in Go

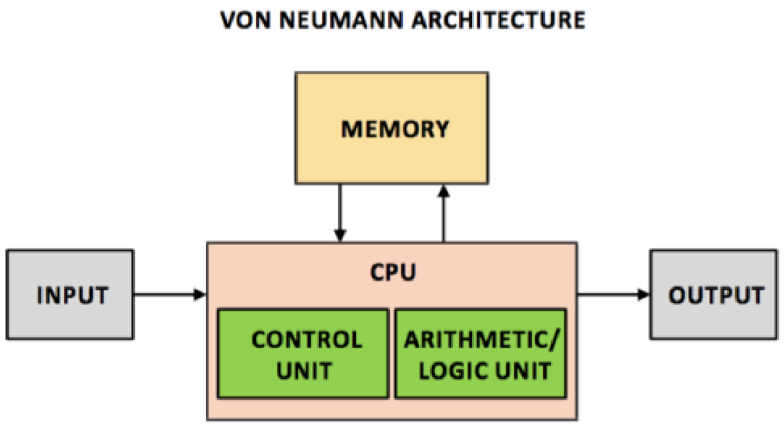
- Goroutines and channels are the main primitives
- Sometimes you just need a reference counter or lock
  - “sync” and “sync/atomic” packages
  - Mutex, condition, atomic operations
- Sometimes you need to wait for a go routine to finish
  - Didn’t happen in any of the examples in the slides
  - WaitGroups are key

# WaitGroups

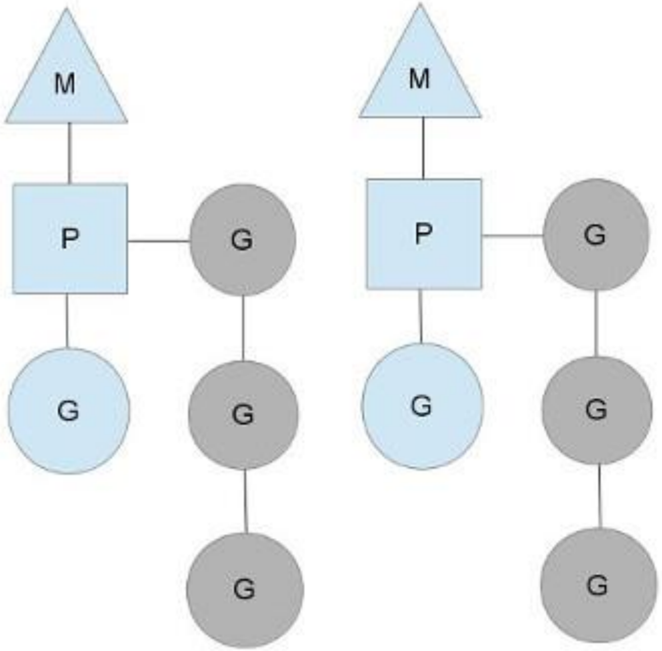
```
func testQ() {
    var wg sync.WaitGroup
    wg.Add(4)
    ch := make(chan int)
    for i:=0; i<4; i++ {
        go func(id int) {
            aval, amore := <- ch
            if(amore) {
                fmt.Printf("reader #%d got %d value\n", id, aval)
            } else {
                fmt.Printf("channel reader #%d terminated with nothing.\n", id)
            }
            wg.Done()
        }(i)
    }
    time.Sleep(1000 * time.Millisecond)
    close(ch)
    wg.Wait()
}
```

# Go: magic or threadpools and concurrent Qs?

- We've seen several abstractions for
  - Control flow/exection
  - Communication
- Lots of discussion of pros and cons
- Ultimately still CPUs + instructions
- Go: just sweeping issues under the language interface?
  - Why is it OK to have 100,000s of goroutines?
  - Why isn't composition an issue?



# Go implementation details




- M = “machine” → OS thread

```
struct Sched {
  Lock;               // global sched lock.
                      // must be held to edit G or M queues

  G *gfree;           // available g's (status == Gdead)
  G *ghead;           // g's waiting to run queue
  G *gtail;           // tail of g's waiting to run queue
  int32 gwait;        // number of g's waiting to run
  int32 gcount;       // number of g's that are alive
  int32 grunning;     // number of g's running on cpu
                      // or in syscall

  M *mhead;           // m's waiting for work
  int32 mwait;        // number of m's waiting for work
  int32 mcount;       // number of m's that have been created
  ...
};
```

# 1000s of go routines?

- Creates a channel
- Creates “consumers” goroutines
- Each of them tries to read from the channel
- Main either:
  - Sleeps for 1 second, closes the channel
  - sends “consumers” values

```go
func testQ(consumers int) {
    startTimes["testQ"] = time.Now()
    var wg sync.WaitGroup
    wg.Add(consumers)
    ch := make(chan int)
    for i:=0; i<consumers; i++ {
        go func(id int) {
            aval, amore := <- ch
            if(amore) {
                info("reader #%d got %d value\n", id, aval)
            } else {
                info("channel reade
            }
            wg.Done()
        }(i)
    }
    time.Sleep(1000 * time.Millise
    close(ch)
    wg.Wait()
    stopTimes["testQ"] = time.Now(
}
```

```
PS C:\Users\chris\go\src\cs378\lab3> .\lab3.exe -testq -qproducers 10
testQ:  1.0016706s
PS C:\Users\chris\go\src\cs378\lab3> .\lab3.exe -testq -qproducers 100
testQ:  1.0011655s
PS C:\Users\chris\go\src\cs378\lab3> .\lab3.exe -testq -qproducers 1000
testQ:  1.0084796s
PS C:\Users\chris\go\src\cs378\lab3> .\lab3.exe -testq -qproducers 10000
testQ:  1.0547925s
PS C:\Users\chris\go\src\cs378\lab3> .\lab3.exe -testq -qproducers 100000
testQ:  1.3907835s
PS C:\Users\chris\go\src\cs378\lab3> .\lab3.exe -testq -qproducers 1000000
testQ:  4.2405814s
```

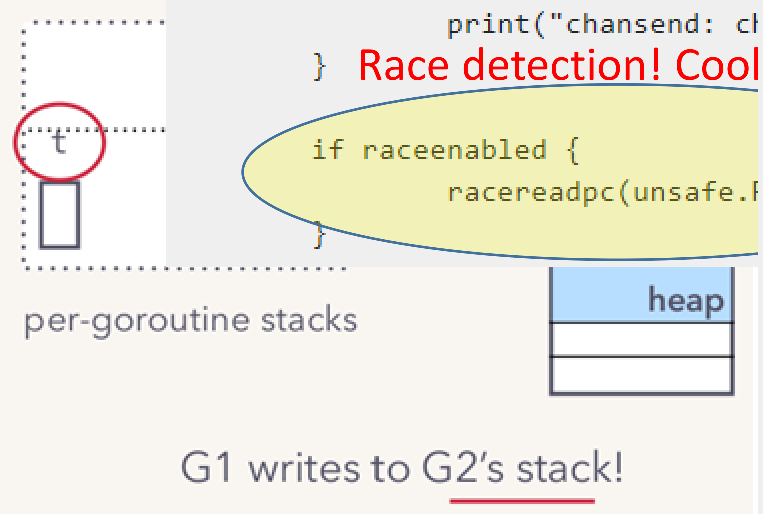
# Channel implementation

- Y
- S

```go
// entry point for c <- x from compiled code
//go:nosplit
func chansend1(c *hchan, elem unsafe.Pointer) {
        chansend(c, elem, true, getcallerpc())
}

/*
 * generic single channel send/recv
 * If block is not nil,
 * then the protocol will not
 * sleep but return if it could
 * not complete.
 *
 * sleep can wake up with g.param == nil
 * when a channel involved in the sleep has
 * been closed.  it is easiest to loop and re-run
 * the operation; we'll see that it's now closed.
 */
func chansend(c *hchan, ep unsafe.Pointer, block bool, callerpc uintptr) bool {
        if c == nil {
                if !block {
```

```go
func chansend(c *hchan, ep unsafe.Pointer, block bool, callerpc uintptr) bool {
        if c == nil {
                if !block {
                        return fal
                }
                gopark(nil, nil, "
                throw("unreachable"
        }

        if debugChan {
                print("chansend: ch
        }

        if raceenabled {
                racereadpc(unsafe.P
        }
```

Race detection! Cool




```go
// Sends and receives on unbuffered or empty-buffered channels are the
// only operations where one running goroutine writes to the stack of
// another running goroutine. The GC assumes that stack writes only
// happen when the goroutine is running and are only done by that
// goroutine. Using a write barrier is sufficient to make up for
// violating that assumption, but the write barrier has to work.
// typedmemmove will call bulkBarrierPreWrite, but the target bytes
// are not in the heap, so that will not help. We arrange to call
// memmove and typeBitsBulkBarrier instead.

func sendDirect(t *_type, sg *sudog, src unsafe.Pointer) {
        // src is on our stack, dst is a slot on another stack.

        // Once we read sg.elem out of sg, it will no longer
        // be updated if the destination's stack gets copied (shrunk).
        // So make sure that no preemption points can happen between read & use.
        dst := sg.elem
        typeBitsBulkBarrier(t, uintptr(dst), uintptr(src), t.size)
        memmove(dst, src, t.size)
}
```

```go
                unlock(&c.lock)
                panic(plainError("send on closed channel"))
        }

        if sg := c.recvq.dequeue(); sg != nil {
                // Found a waiting receiver. We pass the value we want to send
                // directly to the receiver, bypassing the channel buffer (if any).
```

Transputers did this in hardware in the 90s btw.

# Channel implementation

- You can just read it:
  - [https://golang.org/src/runtime/chan.go](https://golang.org/src/runtime/chan.go)
- Some highlights:
  - Race detection built in
  - Fast path just write to receiver stack
  - Often has no capacity → scheduler hint!
  - Buffered channel implementation fairly standard

```go
// entry point for c <- x from compiled code
//go:nosplit
func chansend1(c *hchan, elem unsafe.Pointer) {
	chansend(c, elem, true, getcallerpc())
}

/*
 * generic single channel send/recv
 * If block is not nil,
 * then the protocol will not
 * sleep but return if it could
 * not complete.
 *
 * sleep can wake up with g.param == nil
 * when a channel involved in the sleep has
 * been closed.  it is easiest to loop and re-run
 * the operation; we'll see that it's now closed.
 */
func chansend(c *hchan, ep unsafe.Pointer, block bool, callerpc uintptr) bool {
	if c == nil {
		if !block {
			return false
		}
		gopark(nil, nil, "chan send (nil chan)", traceEvGoStop, 2)
		throw("unreachable")
	}

	if debugChan {
		print("chansend: chan=", c, "\n")
	}

	if raceenabled {
		racereadpc(unsafe.Pointer(c), callerpc, funcPC(chansend))
	}

	// Fast path: check for failed non-blocking operation without acquiring the lock.
	//
	// After observing that the channel is not closed, we observe that the channel is
	// not ready for sending. Each of these observations is a single word-sized read
	// (first c.closed and second c.recvq.first or c.qcount depending on kind of channel).
	// Because a closed channel cannot transition from 'ready for sending' to
	// 'not ready for sending', even if the channel is closed between the two observations,
	// they imply a moment between the two when the channel was both not yet closed
	// and not ready for sending. We behave as if we observed the channel at that moment,
	// and report that the send cannot proceed.
	//
	// It is okay if the reads are reordered here: if we observe that the channel is not
	// ready for sending and then observe that it is not closed, that implies that the
	// channel wasn't closed during the first observation.
	if !block && c.closed == 0 && ((c.dataqsiz == 0 && c.recvq.first == nil) ||
		(c.dataqsiz > 0 && c.qcount == c.dataqsiz)) {
		return false
	}

	var t0 int64
	if blockprofilerate > 0 {
		t0 = cputicks()
	}

	lock(&c.lock)

	if c.closed != 0 {
		unlock(&c.lock)
		panic(plainError("send on closed channel"))
	}

	if sg := c.recvq.dequeue(); sg != nil {
		// Found a waiting receiver. We pass the value we want to send
		// directly to the receiver, bypassing the channel buffer (if any).
		send(c, sg, ep, func() { unlock(&c.lock) }, 3)
```

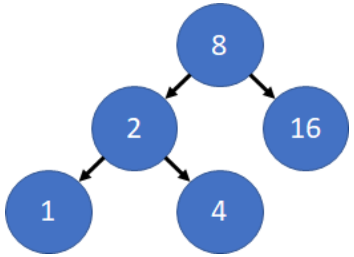
# Go: Sliced Bread 2.0?

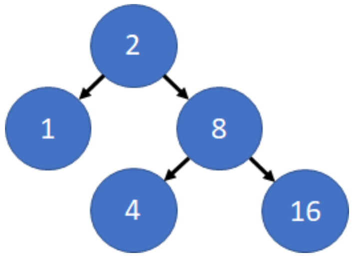
- Lacks compile-time generics
  - Results in code duplication
  - Metaprogramming cannot be statically checked
  - Standard library cannot offer generic algorithms
- Lack of language extensibility makes certain tasks more verbose
  - Lacks operator overloading (Java)
- Pauses and overhead of garbage collection
  - Limit Go’s use in systems programming compared to languages with manual memory management
- *Right tradeoffs? None of these problems have to do with concurrency!*

# Questions?

# Now. Let’s discuss Lab 3

# Binary Search Trees




- Each node has a value
- Left nodes have smaller values
- Right nodes have greater values

- Want to detect duplicate trees
  - Insertion order affects layout
- Linearize trees for comparison
  - Makes comparison expensive

# Hashing BSTs

```
func initialHash() uint64 {
	return 1
}

func hash(uint64 hash, uint64 val) {
	val2 = val + 2
	prime = 4222234741
	return (hash*val2+val2)%prime
}
```

- Initialize hash
- Traverse tree in-order
- Incorporate values into hash

- Hash function doesn’t have to be very complex
- Just make sure it handles zeros and similar numbers nicely

# Processing pipeline

- Read in trees from file
  - Array / slice of BSTs
- Hash trees + insert hashes
  - Map from hash to tree indexes
- Compare trees
  - Equivalence matrix
    - num trees x num trees

# Parallelizing the pipeline

**Step 2**

- Implement just hashing first
- Goroutines
  - 1 per tree
  - Dedicated inserter goroutine(s)
    - Communicate via channel
- Thread pool
  - hash-workers threads
  - Acquire lock(s) to insert
- Multiple data-workers optional

**Step 3**

- Goroutines
  - 1 per comparison
- Thread pool
  - comp-workers threads
  - Send work via channel
    - (Optional) custom implementation
    - Queue, mutex, and conditions
- Store results directly in matrix

# Go: command-line flags

```
import "flag"
func main() {
	intPtr = flag.Int("num", 0, "number argument")
	flag.Parse()
	num : = *flagPtr
}

./my_program -num=1
```

# Go: file parsing

```
import ("io/ioutil" "strconv" "strings")
func main() {
	fileData, err := ioutil.ReadFile(fileName)
	fileData = fileData[:len(fileData)-1]   // remove EOF
	fileLines := strings.Split(string(fileData), "\n")
	for _, line := range fileLines {
		// parse line with strings.Split and strconv.Atoi()
	}
}
```

# Go: timing

```
import "time"
func main() {
	start := time.Now()
	// do some work
	timeTakenStr:= time.Since(start)
	fmt.Printf("Doing work took %s\n", timeTakenStr)
}
```

# Go: functions and return values

```
func notMain() (int, bool) {   // multiple return values
        return (3, false)
}

func main() {
        i, b := notMain()
        j, _ := notMain()   // throw away value
}
```

# Go: synchronization

```
import "sync"   // contains WaitGroups
func main() {
        var *mutex = &sync.Mutex{}   // pointer to mutex
        var *cond = &sync.NewCond(mutex)   // mutex condition
        mutex.Lock()
        cond.Wait()   // releases lock on mutex
        cond.Signal()   // wakes threads waiting on cond
        mutex.Unlock()
}
```

# Go: slices

```
func main() {
        mySlice := make([]int, 2)
         mySlice[1] = 5   // can use like an array
         mySlice = append(mySlice, 10)   // can use like a list
        l := len(mySlice)
        subSlice := mySlice[0:1]   // can slice like in Python
        fromStartToTwo := mySlice[:2]
        fromOneToEnd := mySlice[1:]
}
```

# Go: maps

```
func main() {
	mapIntBool := make(map [int] bool)   // map from ints to bools
	 mapIntBool[5] = true
	for key, value := range mapIntBool {
		// use key or value
	}
}
// map value can be a slice
```

# Go: misc

```
type myStruct struct {
	mySlice []int
	myChan chan int
	mySliceOfSlice [][]bool
	myPtr *myStruct
}
var ms myStruct   // declare variable without initialization
// use dot operator for structs, pointers, and pointers to structs
ms.myPtr.mySlice[2]
```

# Questions?