CS252 Graduate Computer Architecture Fall 2015 Lecture 14: Synchronization and Memory Models

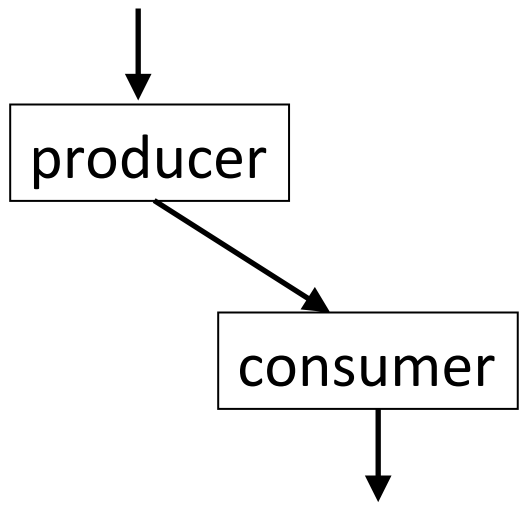
Krste Asanovic
krste@eecs.berkeley.edu
http://inst.eecs.berkeley.edu/~cs252/fa15

Synchronization

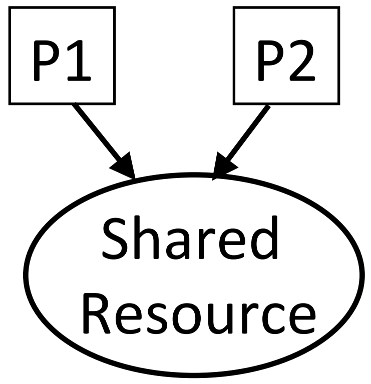
The need for synchronization arises whenever there are concurrent processes in a system *(even in a uniprocessor system)*.

Two classes of synchronization:

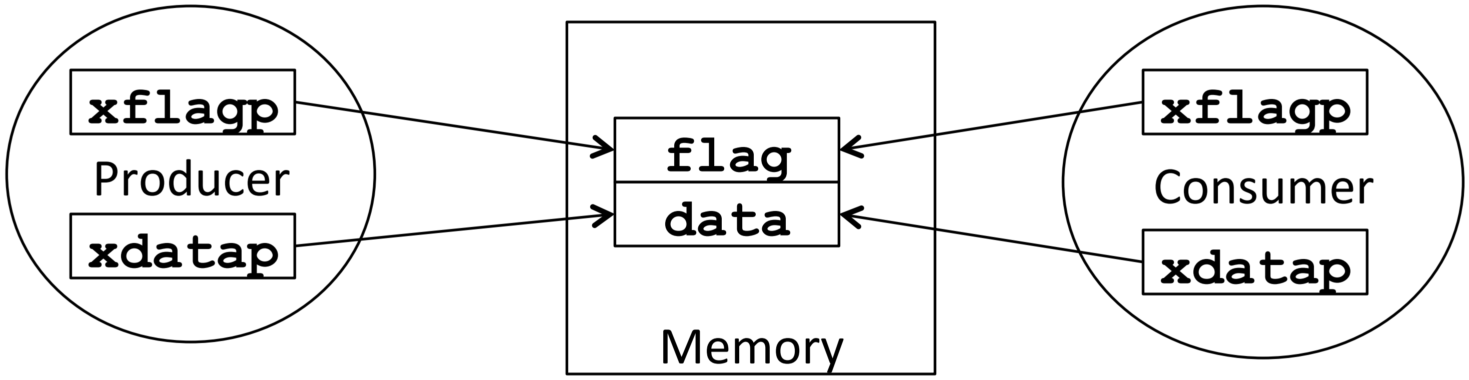
- *Producer-Consumer:* A consumer process must wait until the producer process has produced data



- *Mutual Exclusion:* Ensure that only one process uses a resource at a given time



Simple Producer-Consumer Example



Initially `flag=0`

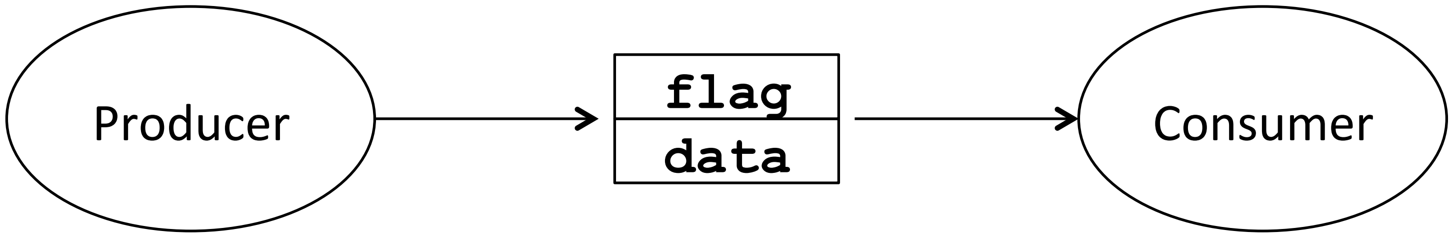
```
sd xdata, (xdatap)              spin: ld xflag, (xflagp)
li xflag, 1                           beqz xflag, spin
sd xflag, (xflagp)                    ld xdata, (xdatap)
```

Is this correct?

Memory Model

- Sequential ISA only specifies that each processor sees its own memory operations in program order
- Memory model describes what values can be returned by load instructions across multiple threads

Simple Producer-Consumer Example



Initially **`flag=0`**

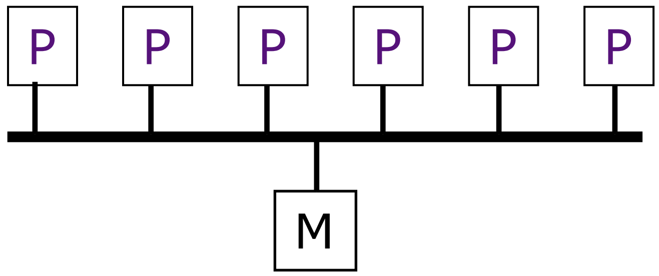
```
sd xdata, (xdatap)
li xflag, 1
sd xflag, (xflagp)
```

```
spin: ld xflag, (xflagp)
      beqz xflag, spin
      ld xdata, (xdatap)
```

Can consumer read **flag=1** before **`data`** written by producer?

Sequential Consistency

A Memory Model

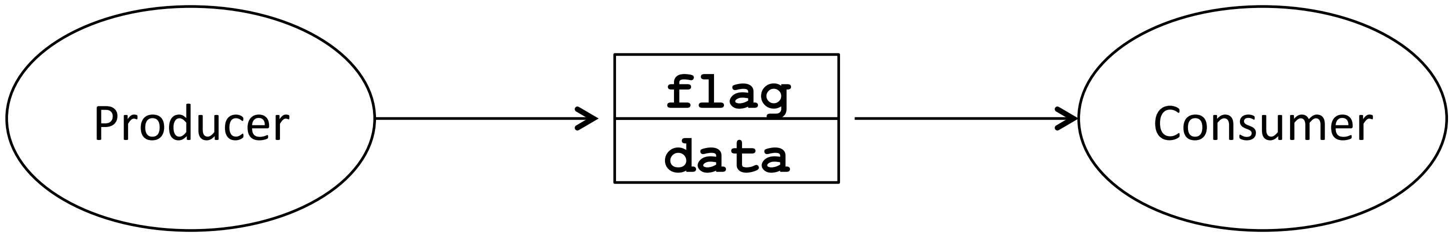


“ A system is *sequentially consistent* if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in the order specified by the program”

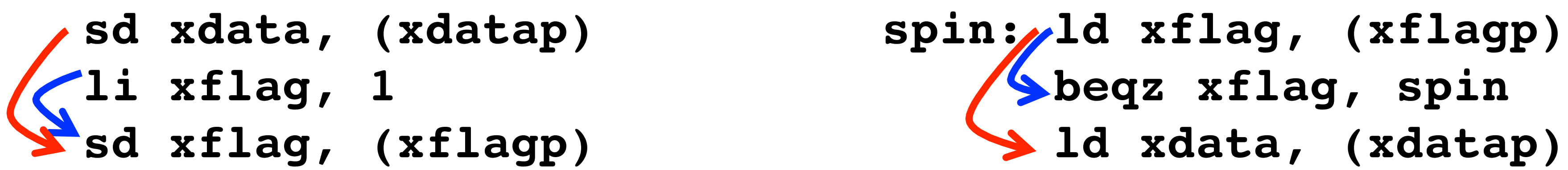
Leslie Lamport

Sequential Consistency = arbitrary *order-preserving interleaving* of memory references of sequential programs

Simple Producer-Consumer Example



Initially flag =0



```
sd xdata, (xdatap)                spin: ld xflag, (xflagp)
li xflag, 1                             beqz xflag, spin
sd xflag, (xflagp)                      ld xdata, (xdatap)
```

↳ Dependencies from sequential ISA

↳ Dependencies added by sequentially consistent memory model

Implementing SC in hardware

- Only a few commercial systems implemented SC
  - Neither x86 nor ARM are SC
- Requires either severe performance penalty
  - Wait for stores to complete before issuing new store
- Or, complex hardware
  - Speculatively issue loads but squash if memory inconsistency with later-issued store discovered (MIPS R10K)

Software reorders too!

```
//Producer code              //Consumer code
*datap = x/y;                while (!*flagp)
*flagp = 1;                       ;
                             d = *datap;
```

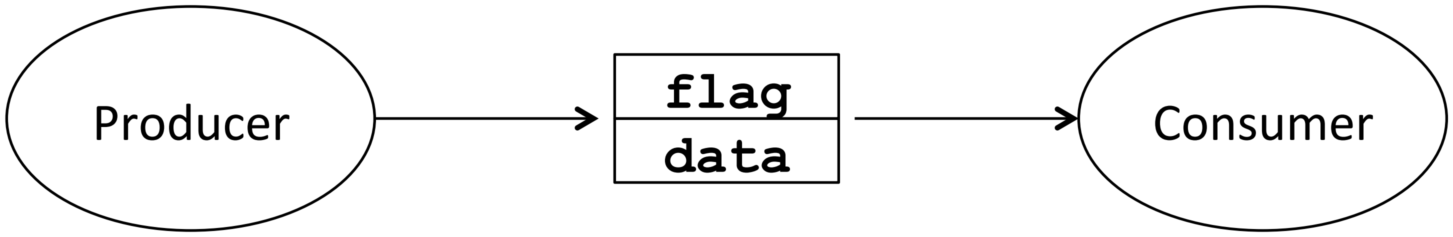
- Compiler can reorder/remove memory operations unless made aware of memory model
  - Instruction scheduling, move loads before stores if to different address
  - Register allocation, cache load value in register, don't check memory
- Prohibiting these optimizations would result in very poor performance

Relaxed Memory Models

- Not all dependencies assumed by SC are supported, and software has to explicitly insert additional dependencies were needed
- Which dependencies are dropped depends on the particular memory model
  – IBM370, TSO, PSO, WO, PC, Alpha, RMO, …
- How to introduce needed dependencies varies by system
  – Explicit FENCE instructions (sometimes called sync or memory barrier instructions)
  – Implicit effects of atomic memory instructions

How on earth are programmers supposed to work with this????

Fences in Producer-Consumer Example

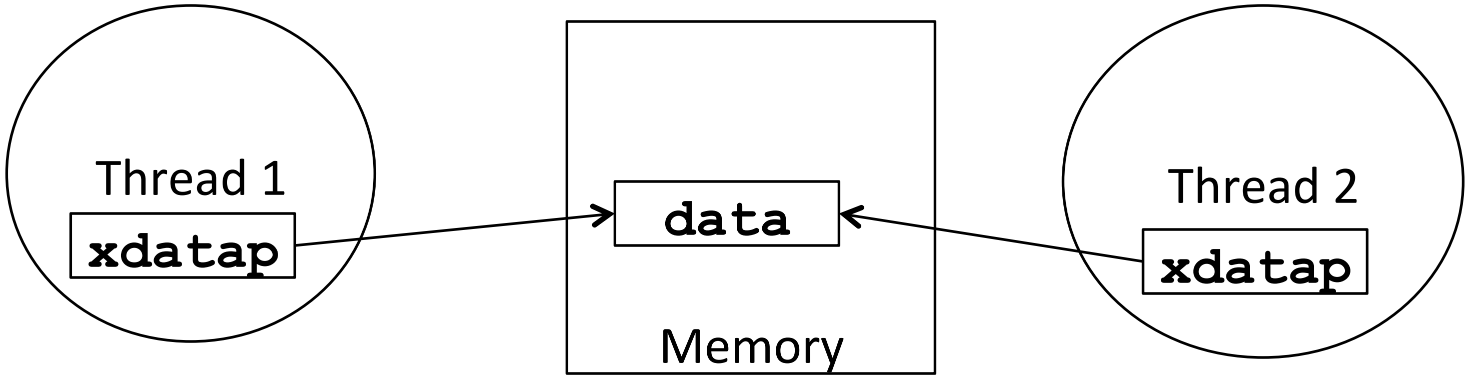


Initially flag =0

```
sd xdata, (xdatap)
li xflag, 1
fence.w.w //Write-write fence
sd xflag, (xflagp)
```

```
spin: ld xflag, (xflagp)
      beqz xflag, spin
      fence.r.r //Read-read fence
      ld xdata, (xdatap)
```

Simple Mutual-Exclusion Example



```
// Both threads execute:
ld xdata, (xdatap)
add xdata, 1
sd xdata, (xdatap)
```

Is this correct?

Mutual Exclusion Using Load/Store

A protocol based on two shared variables c1 and c2.
Initially, both c1 and c2 are 0 *(not busy)*

Process 1

```
     ...
     c1=1;
L:   if c2=1 then go to L
       < critical section>
     c1=0;
```

Process 2

```
     ...
     c2=1;
L:   if c1=1 then go to L
       < critical section>
     c2=0;
```

What is wrong? *Deadlock!*

Mutual Exclusion: *second attempt*

To avoid *deadlock*, let a process give up the reservation (i.e. Process 1 sets c1 to 0) while waiting.

Process 1

```
     ...
L:  c1=1;
    if c2=1 then
        { c1=0; go to L}
      < critical section>
    c1=0
```

Process 2

```
     ...
L:  c2=1;
    if c1=1 then
        { c2=0; go to L}
      < critical section>
    c2=0
```

- Deadlock is not possible but with a low probability a *livelock* may occur.

- An unlucky process may never get to enter the critical section $\Rightarrow$ *starvation*

A Protocol for Mutual Exclusion

T. Dekker, 1966

A protocol based on 3 shared variables c1, c2 and turn.
Initially, both c1 and c2 are 0 *(not busy)*

Process 1

```
    ...
    c1=1;
    turn = 1;
L: if c2=1 & turn=1
                then go to L
      < critical section>
    c1=0;
```

Process 2

```
    ...
    c2=1;
    turn = 2;
L: if c1=1 & turn=2
                then go to L
      < critical section>
    c2=0;
```

- turn = *i* ensures that only process *i* can wait
- variables c1 and c2 ensure *mutual exclusion*

Solution for n processes was given by Dijkstra and is quite tricky!

Analysis of Dekker's Algorithm

Scenario 1

```
    ...              Process 1
    c1=1;
    turn = 1;
L: if c2=1 & turn=1
                 then go to L
      < critical section>
    c1=0;
```

```
    ...              Process 2
    c2=1;
    turn = 2;
L: if c1=1 & turn=2
                 then go to L
      < critical section>
    c2=0;
```

Scenario 2

```
    ...              Process 1
    c1=1;
    turn = 1;
L: if c2=1 & turn=1
                 then go to L
      < critical section>
    c1=0;
```

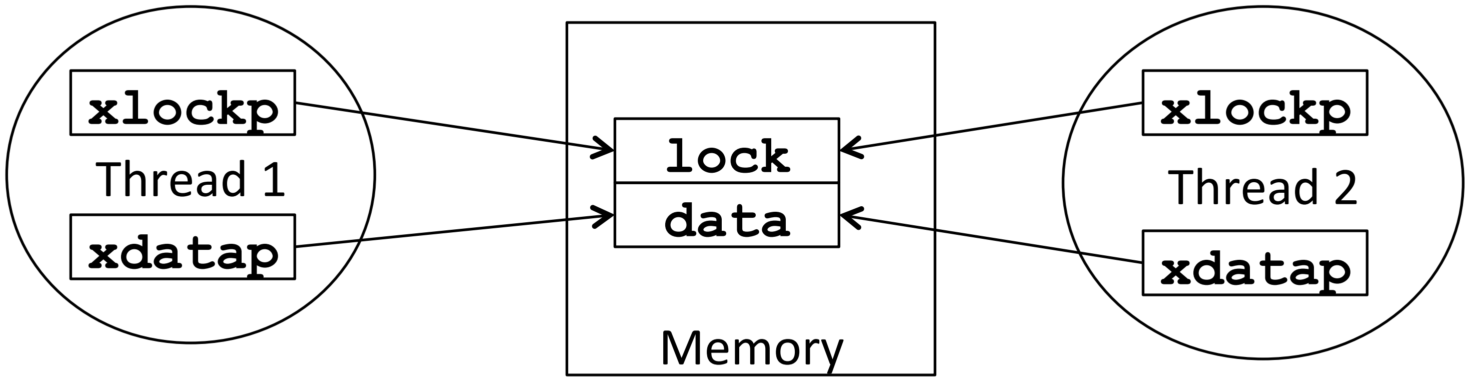
```
    ...              Process 2
    c2=1;
    turn = 2;
L: if c1=1 & turn=2
                 then go to L
      < critical section>
    c2=0;
```

ISA Support for Mutual-Exclusion Locks

- Regular loads and stores in SC model (plus fences in weaker model) sufficient to implement mutual exclusion, but inefficient and complex code
- Therefore, atomic read-modify-write (RMW) instructions added to ISAs to support mutual exclusion

- Many forms of atomic RMW instruction possible, some simple examples:
  - Test and set (reg_x = M[a]; M[a]=1)
  - Swap (reg_x=M[a]; M[a] = reg_y)

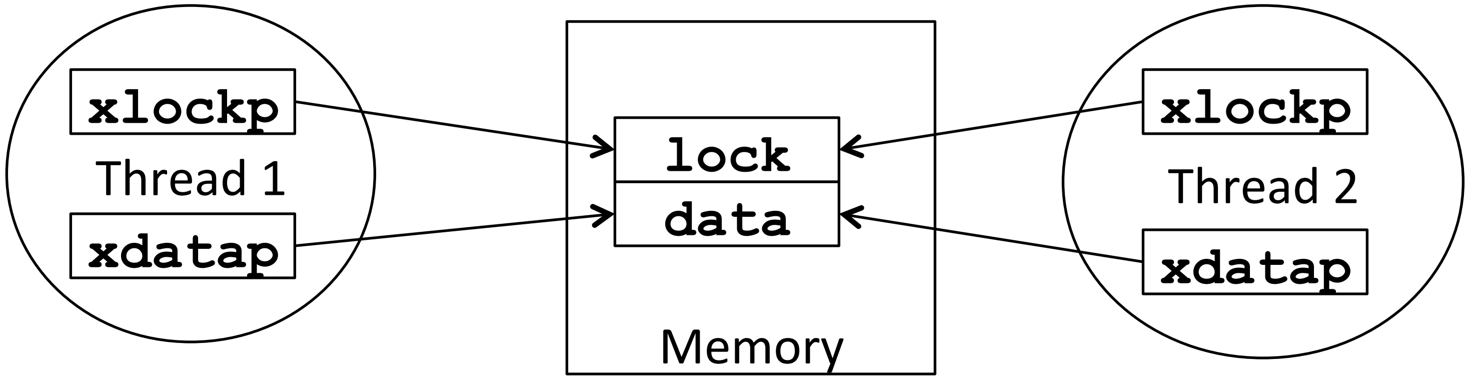
Lock for Mutual-Exclusion Example



```
// Both threads execute:
      li xone, 1
spin: amoswap xlock, xone, (xlockp)      Acquire Lock
      bnez xlock, spin
      ld xdata, (xdatap)
      add xdata, 1                       Critical Section
      sd xdata, (xdatap)
      sd x0, (xlockp)                    Release Lock
```

Assumes SC memory model

Lock for Mutual-Exclusion with Relaxed MM



```
// Both threads execute:
      li xone, 1
spin: amoswap xlock, xone, (xlockp)
      bnez xlock, spin
      fence.r.r
      ld xdata, (xdatap)
      add xdata, 1
      sd xdata, (xdatap)
      fence.w.w
      sd x0, (xlockp)
```

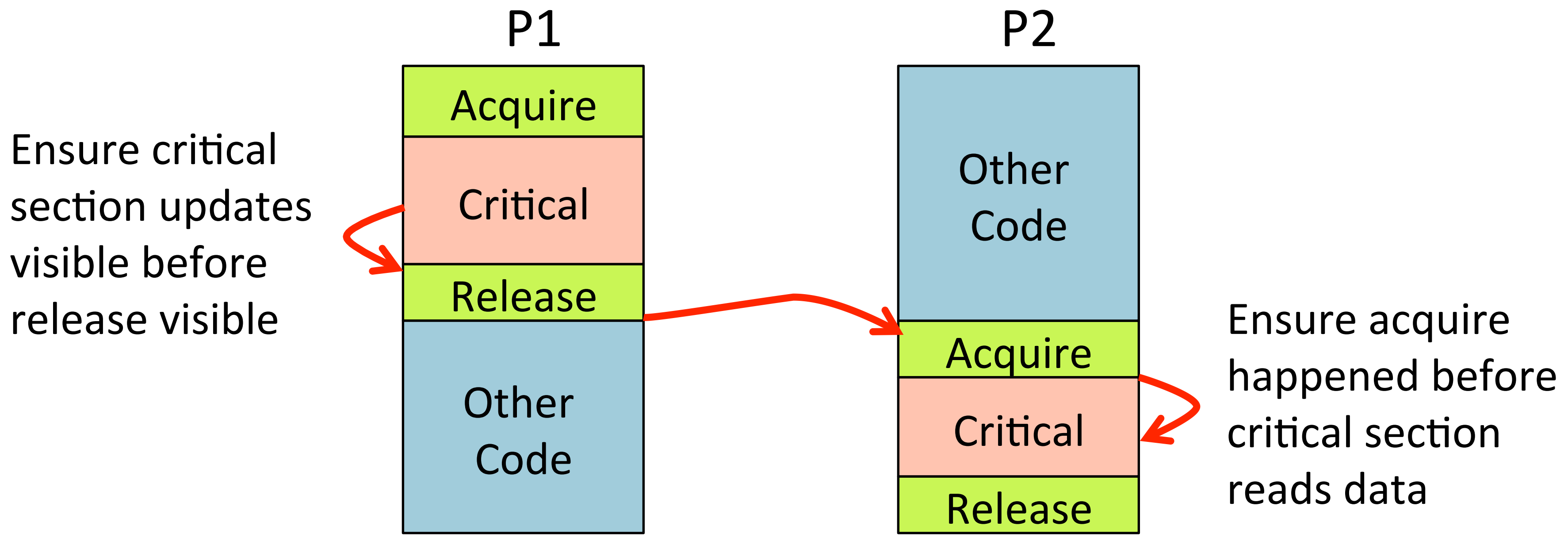
Acquire Lock

Critical Section

Release Lock

Release Consistency

- Observation that consistency only matters when processes communicate data
- Only need to have consistent view when one process shares its updates to other processes
- Other processes only need to ensure they receive updates after they acquire access to shared data


P1
P2
Acquire
Critical
Release
Other Code
Other Code
Acquire
Critical
Release
Ensure critical section updates visible before release visible
Ensure acquire happened before critical section reads data

Release Consistency Adopted

- Memory model for C/C++ and Java uses release consistency
- Programmer has to identify synchronization operations, and if all data accesses are protected by synchronization, appears like SC to programmer

- ARM v8.1 and RISC-V ISA adopt release consistency semantics on AMOs

Nonblocking Synchronization

```
Compare&Swap(m), $R_t$, $R_s$:
  if ($R_t$==M[m])
      then  M[m]=$R_s$;
       $R_s$=$R_t$ ;
       status $\leftarrow$ success;
      else   status $\leftarrow$ fail;
```

status is an *implicit argument*

```
try:    Load $R_{head}$, (head)
spin:   Load $R_{tail}$, (tail)
        if $R_{head}$==$R_{tail}$ goto spin
        Load R, ($R_{head}$)
        $R_{newhead}$ = $R_{head}$+1
        Compare&Swap(head), $R_{head}$, $R_{newhead}$
        if (status==fail) goto try
        process(R)
```

Load-reserve & Store-conditional

Special register(s) to hold reservation flag and address, and the outcome of store-conditional

```
Load-reserve R, (m):
    <flag, adr> ← <1, m>;
    R ← M[m];
```

```
Store-conditional (m), R:
    if <flag, adr> == <1, m>
    then  cancel other procs'
            reservation on m;
          M[m] ← R;
          status ← succeed;
    else  status ← fail;
```

```
try:    Load-reserve $R_{head}$, (head)
spin:   Load $R_{tail}$, (tail)
        if $R_{head}$==$R_{tail}$ goto spin
        Load R, ($R_{head}$)
        $R_{head}$ = $R_{head}$ + 1
        Store-conditional (head), $R_{head}$
        if (status==fail) goto try
        process(R)
```

Acknowledgements

- This course is partly inspired by previous MIT 6.823 and Berkeley CS252 computer architecture courses created by my collaborators and colleagues:
  - Arvind (MIT)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)