M4 – Parallelism

Implementation of Locks
Cache Coherence

Outline

- Parallelism
- Flynn's classification
- Vector Processing
  - Subword Parallelism
- Symmetric Multiprocessors, Distributed Memory Machines
  - Shared Memory Multiprocessing, Message Passing
- Synchronization Primitives
  - Locks, LL-SC
- Cache coherence

Synchronization

- Two processors sharing an area of memory
  - P1 writes, then P2 reads
  - Data race if P1 and P2 don't synchronize
    - Result depends of order of accesses

Synchronization

- Two processors sharing an area of memory
  - P1 writes, then P2 reads
  - Data race if P1 and P2 don't synchronize
    - Result depends of order of accesses
- Hardware support required
  - Atomic read/write memory operation
  - No other access to the location allowed between the read and write

Synchronization

- Two processors sharing an area of memory
  - P1 writes, then P2 reads
  - Data race if P1 and P2 don't synchronize
    - Result depends of order of accesses
- Hardware support required
  - Atomic read/write memory operation
  - No other access to the location allowed between the read and write
- Could be a single instruction
  - E.g., atomic swap of register ↔ memory
  - Or an atomic pair of instructions

ISA

lock L

Implementing Locks

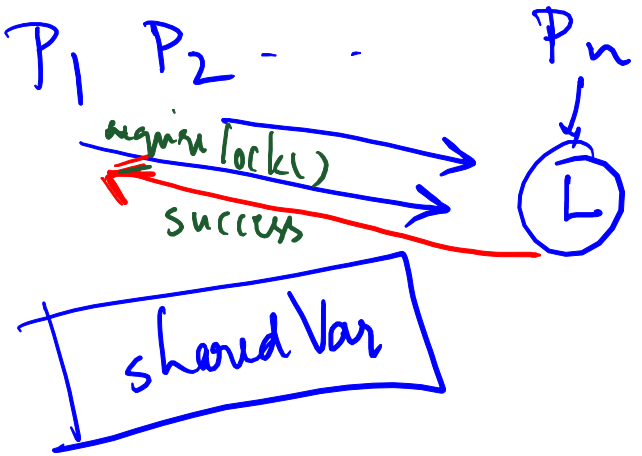
- Must synchronize processes so that they access shared variable one at a time in critical section; called **Mutual Exclusion**

Implementing Locks

- Must synchronize processes so that they access shared variable one at a time in critical section; called **Mutual Exclusion**
- Mutex Lock: a synchronization primitive

Implementing Locks

- Mutex Lock: a synchronization primitive
- AcquireLock(L)
  -
  -
- ReleaseLock(L)
  -
  -



Implementing Locks

- Mutex Lock: a synchronization primitive
- AcquireLock(L)
  - Done before critical section of code
  - Returns when safe for process to enter critical section success
- ReleaseLock(L)
  -
  -

Implementing Locks

- Mutex Lock: a synchronization primitive
- AcquireLock(L) ← $\boxed{L=1}$
  - Done before critical section of code
  - Returns when safe for process to enter critical section
- ReleaseLock(L) ←
  - Done after critical section
  - Allows another process to acquire lock

$\boxed{L=0}$

available

$P_1 \; P_2 \; - \; - \; P_n$

$t_0 \rightarrow L=0$

$t_1 \rightarrow$ all acquirelock(L)

$t_1 + \delta \rightarrow P_i \Rightarrow L \leftarrow 1$

$\underbrace{P_1 - - P_{i-1} \; P_{i+1} - - P_n}$

$L=1$

$\rightarrow$ after t_1: P_i is in CS

$\rightarrow t_2$: $P_i \Rightarrow L \leftarrow 0$

Implementing Locks

Process

```
int L=0;

AcquireLock(L):
    while (L==1) ;
    L = 1;
```

wait till lock release

once lock is released

return

Spin lock

/* BUSY WAITING */

Implementing Locks

lock variable.

```
int L=0;

AcquireLock(L):
    while (L==1) ;
    L = 1;

ReleaseLock(L):
    L = 0;
```

/* BUSY WAITING */

Problem in Implementing Locks

```
AcquireLock(L):
    while (L==1) ;
    L = 1;
```

```
wait: LW R1, Addr(L)
      BNEZ wait
      ADDI R1, R1, 1
      SW R1, Addr(L)
```

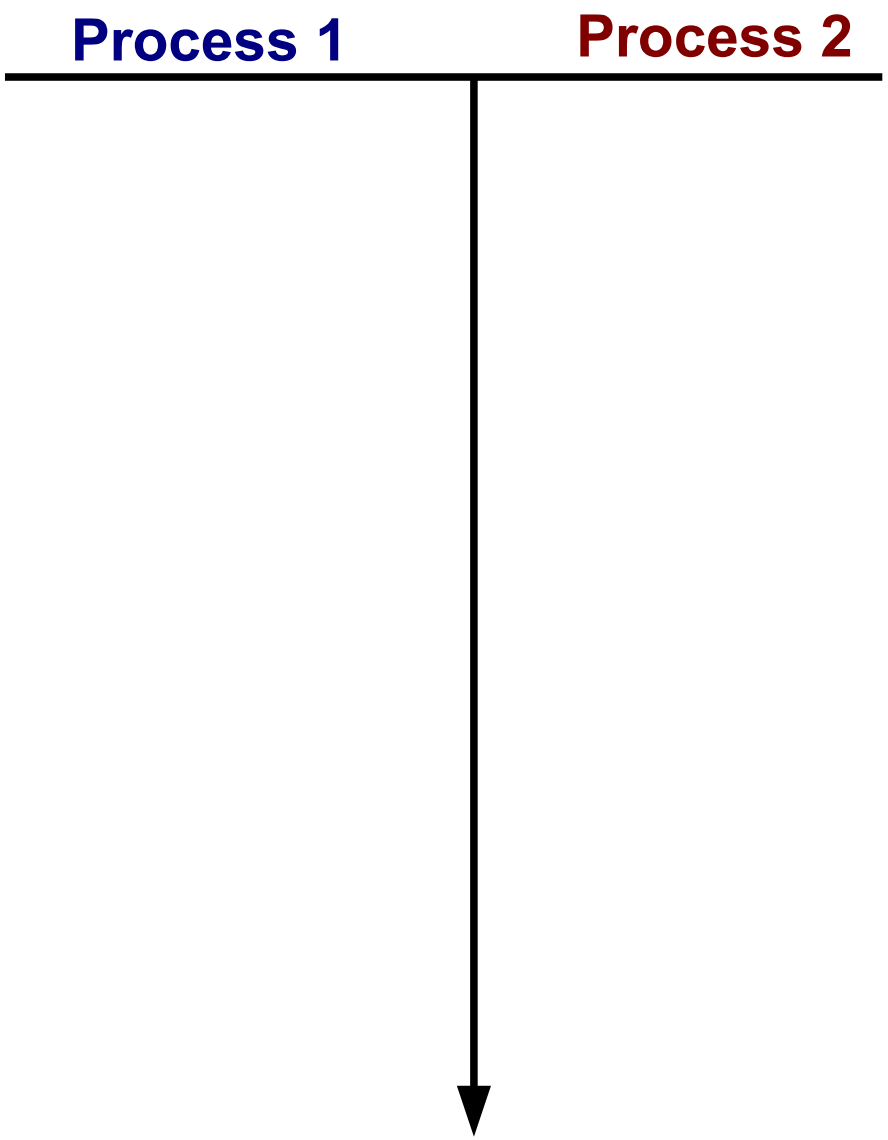
L←1

Problem in Implementing Locks

```
wait: LW R1, Addr(L)
      BNEZ wait
      ADDI R1, R1, 1
      SW R1, Addr(L)
```

Initally L=0. P1 and P2 are in contention to acquire the lock.

Problem in Implementing Locks




```
wait: LW R1, Addr(L)
      BNEZ wait
      ADDI R1, R1, 1
      SW R1, Addr(L)
```

Initally L=0. P1 and P2 are in contention to acquire the lock.

Problem in Implementing Locks

Process 1	Process 2
LW R1, Addr(L)	
Context Switch	

R1=0

```
wait: LW R1, Addr(L)
      BNEZ wait
      ADDI R1, R1, 1
      SW R1, Addr(L)
```

Initally L=0. P1 and P2 are in contention to acquire the lock.

Problem in Implementing Locks

Process 1	Process 2
LW R1, Addr(L)	
R1=0 **Context Switch**	
	LW R1, Addr(L)
	BNEZ wait ← R1=0
	ADDI R1, R1, 1 ←
	# Critical Section # ←
	Context Switch

```
wait: LW R1, Addr(L)
      BNEZ wait
      ADDI R1, R1, 1
      SW R1, Addr(L)
```

Initally L=0. P1 and P2 are in contention to acquire the lock.

Problem in Implementing Locks

Process 1	Process 2
LW R1, Addr(L)	
Context Switch	
	LW R1, Addr(L)
	BNEZ wait
	ADDI R1, R1, 1
	# Critical Section #
	Context Switch
BNEZ wait	
ADDI R1, R1, 1	
# Critical Section #	

```
wait: LW R1, Addr(L)
      BNEZ wait
      ADDI R1, R1, 1
      SW R1, Addr(L)
```

Initally L=0. P1 and P2 are in contention to acquire the lock.

Problem in Implementing Locks

Process 1	Process 2
LW R1, Addr(L)	
Context Switch	
	LW R1, Addr(L)
	BNEZ wait
	ADDI R1, R1, 1
	# Critical Section #
	Context Switch
BNEZ wait	
ADDI R1, R1, 1	
# Critical Section #	

```
wait: LW R1, Addr(L)
      BNEZ wait
      ADDI R1, R1, 1
      SW R1, Addr(L)
```

Initally L=0. P1 and P2 are in contention to acquire the lock.

Both P1 and P2 are executing in the Critical Section !!!

- indivisible
- all or none

Atomic Exchange

- Hardware support for lock implementation

Atomic Exchange

- Hardware support for lock implementation
- Atomic exchange: Swap contents between register and memory.

test if lock=0

Atomic Exchange

set lock=1

- Test&Set
  - Takes one memory operand and a register operand

mem-locs

Test&Set Lock
tmp = Lock
Lock = 1
return tmp

orig lock val

Test&Set occurs **atomically** (indivisibly).

Atomic Exchange

- Test&Set
  - Takes one memory operand and a register operand

Test&Set Lock
tmp = Lock
Lock = 1
return tmp

Test&Set occurs **atomically** (indivisibly).

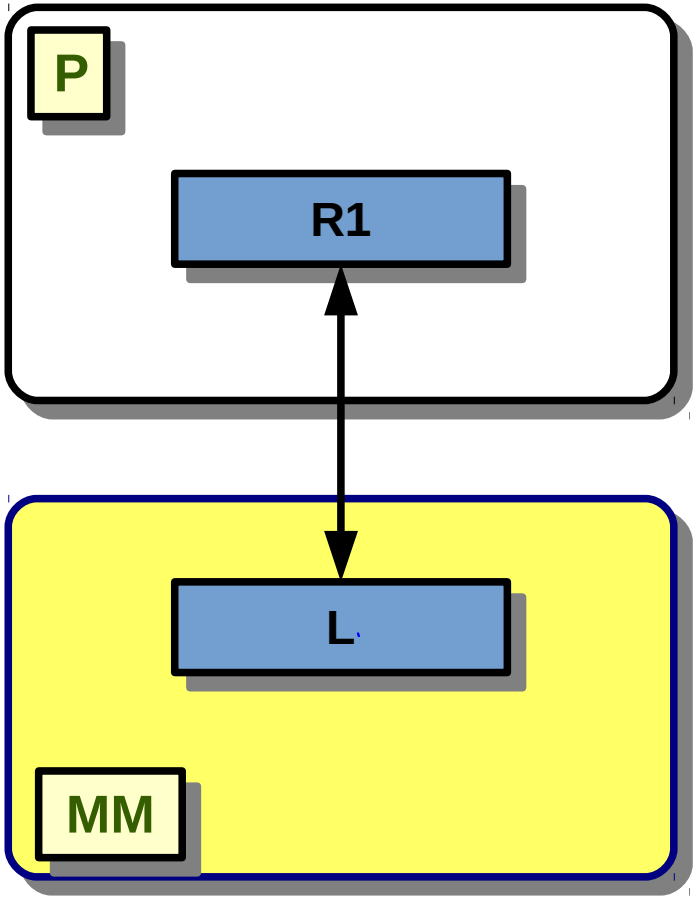
Atomic **Read-Modify-Write (RMW)** instruction

Lock Implementation

R1 → 1 #acquire L

```
lock: Test&Set R1, L
      BNZ R1, lock
      Critical Section
      SW R0, L
```

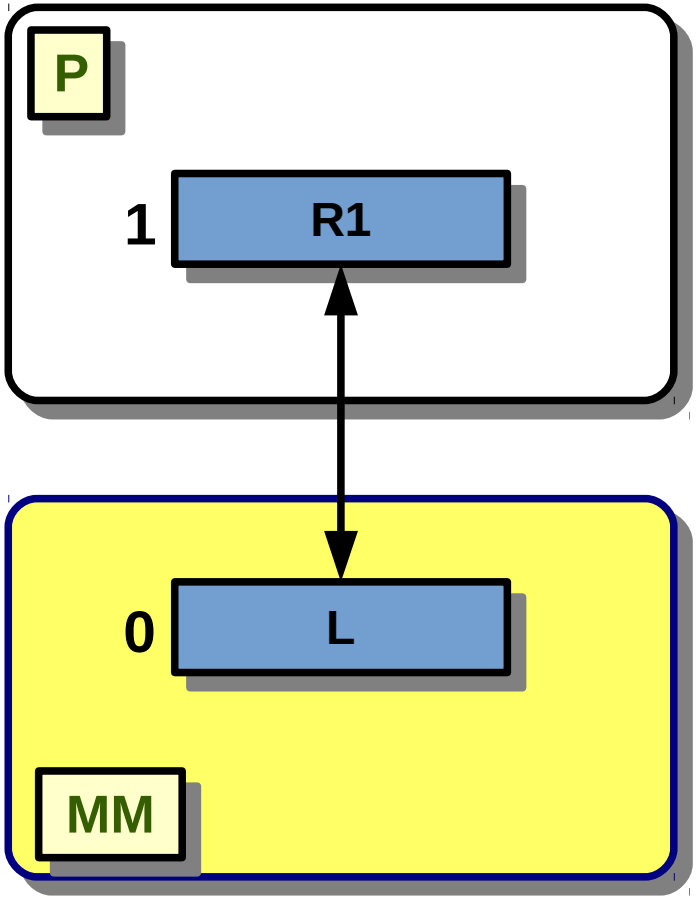
R1 has the orig L value.



Lock Implementation

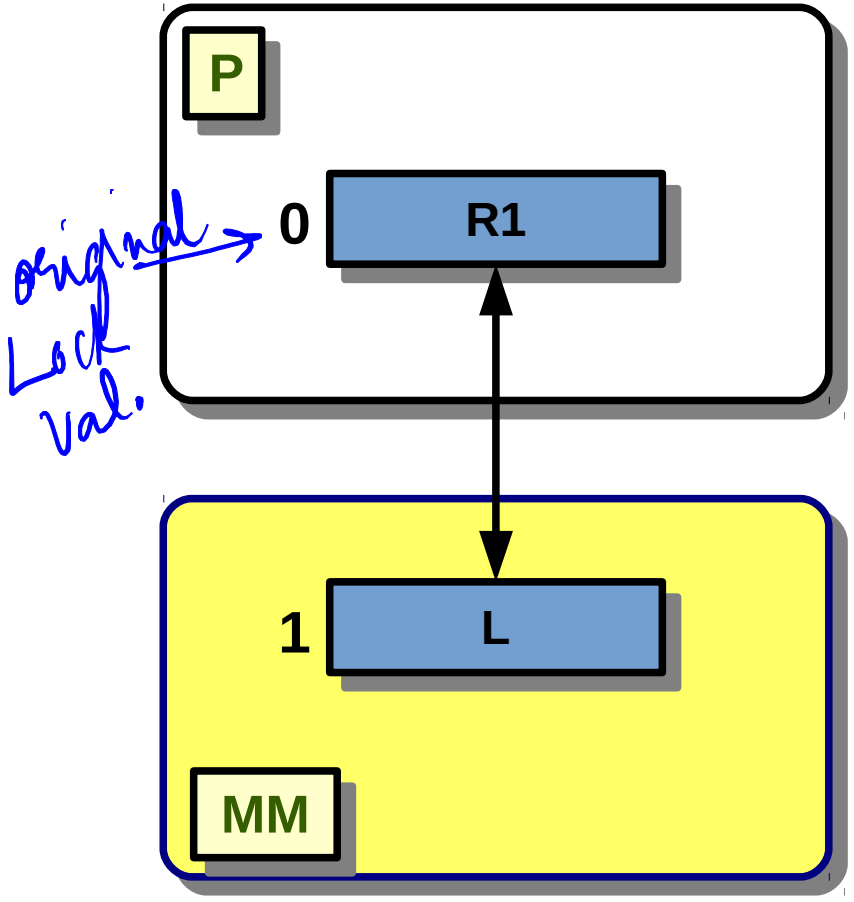
```
lock: Test&Set R1, L
      BNZ R1, lock
      Critical Section
      SW R0, L
```

① before TAS



Lock Implementation

```
lock: Test&Set R1, L
      BNZ R1, lock
      Critical Section
      SW R0, L
```



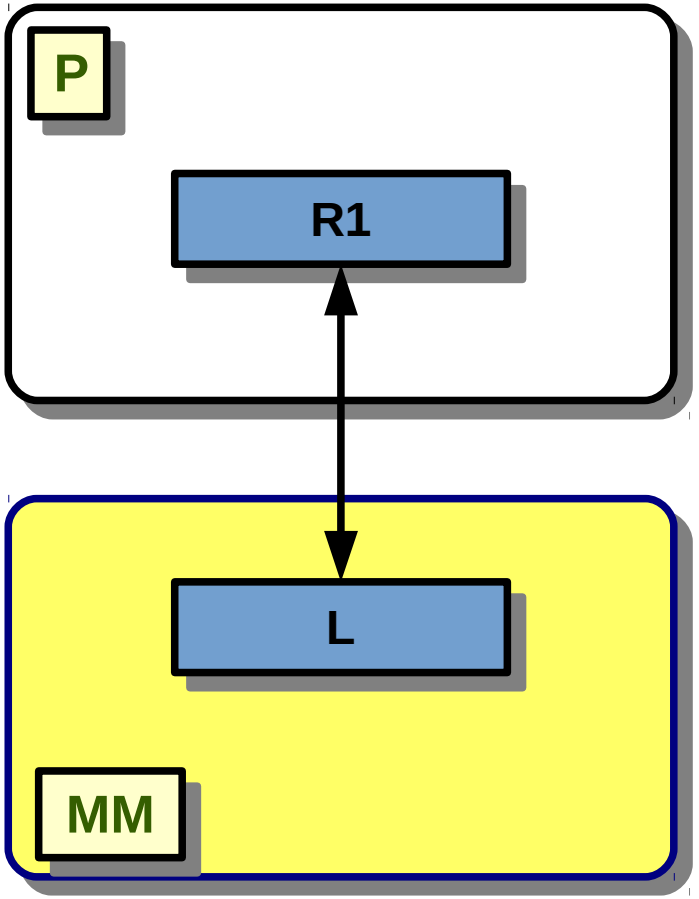
after T&S

Test&Set Implementation

Test&Set R1, L

```
t = L;      # Store a copy of Lock
L = 1;      # Set Lock
R1 ← t; # Write original value
                of the Lock in R1
```

Stores: L, tmp
Loads: R1

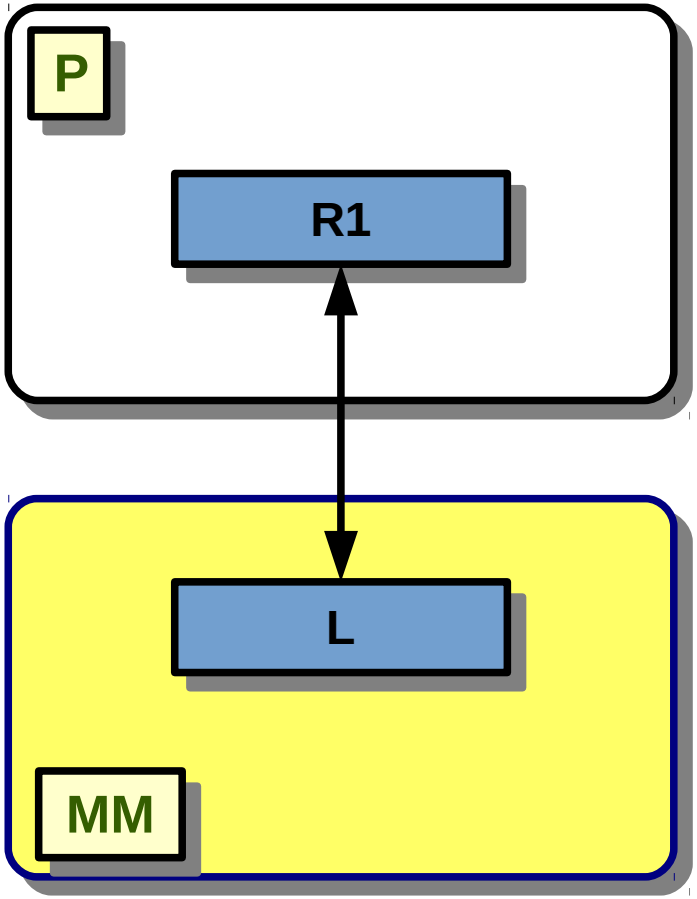


Test&Set Implementation

Test&Set R1, L

```
t = L;     # Store a copy of Lock
L = 1;     # Set Lock
R1 ← t; # Write original value
                of the Lock in R1
```

2 stores (t, L) and 1 load (R1) (atomic – practically 1 instruction)

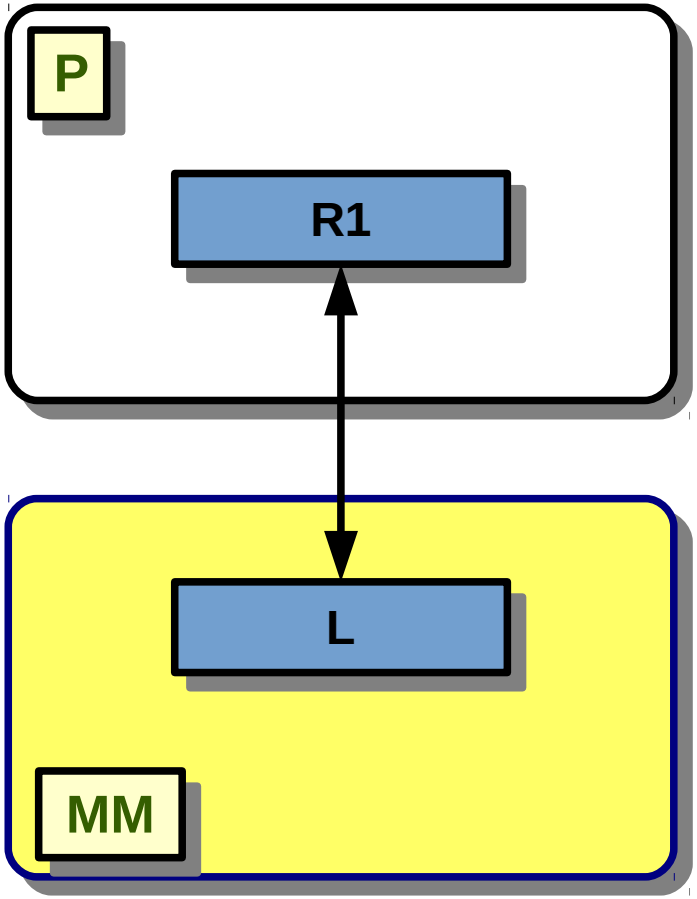


Test&Set Implementation

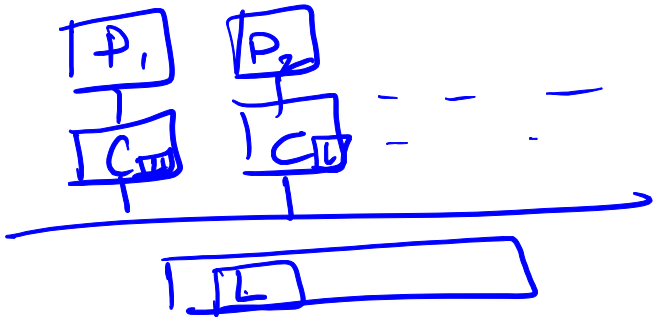
Test&Set R1, L

t = L; # Store a copy of Lock
L = 1; # Set Lock
R1 ← t; # Write original value of the Lock in R1

2 stores (t, L) and 1 load (R1) (atomic – practically 1 instruction)



The atomic **read-modify-write** hardware primitive facilitates synchronization implementations (locks, barriers, etc.)

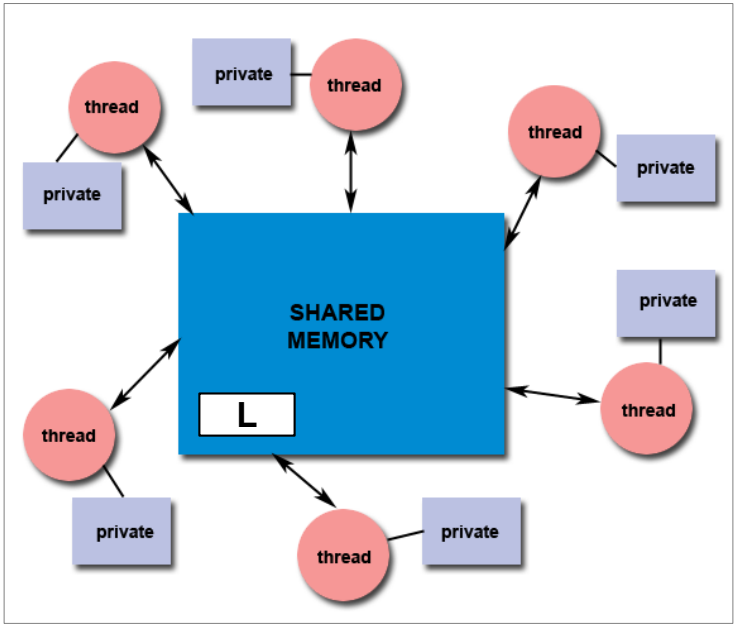


Test&Set – Issues

- T&S is an atomic Read-Modify-Write instruction
  - 2 Stores, 1 Load
  - Atomic

Test&Set – Issues

- T&S is an atomic Read-Modify-Write instruction
  - 2 Stores, 1 Load
  - Atomic
- Consider N processors executing T&S on a single lock variable



Test&Set – Issues

- T&S is an atomic Read-Modify-Write instruction
  - 2 Stores, 1 Load
  - Atomic
- Consider N processors executing T&S on a single lock variable

Test&Set – Issues

- T&S is an atomic Read-Modify-Write instruction
  - 2 Stores, 1 Load
  - Atomic
- Consider N processors executing T&S on a single lock variable
  - Redundant, useless work
  - Performance loss

Test&Set – Issues

- T&S is an atomic Read-Modify-Write instruction
  - 2 Stores, 1 Load
  - Atomic
- Use LL-SC

LL-SC Example

```
LD R2, X
....
....
....
SW R2, X
```

read lock

update lock

t_1: P_1 has lock

$P_2 \ldots P_n$: spin wait.

t_2: P_1 releases lock

$P_2 \ldots P_n$ } attempt acquire lock

$t_2+\delta$: P_2: wins

$P_3 \ldots P_n$ → wasted work

LL-SC Example

Atomic execution required!

```
LD R2, X
....
....
....
SW R2, X
```

LL-SC Example

Atomic execution required!

```
LD R2, X
....
....
....
SW R2, X
```

```
LL R2, X
....
....
....
SC R2, X
```

LL-SC Example

Atomic execution required!

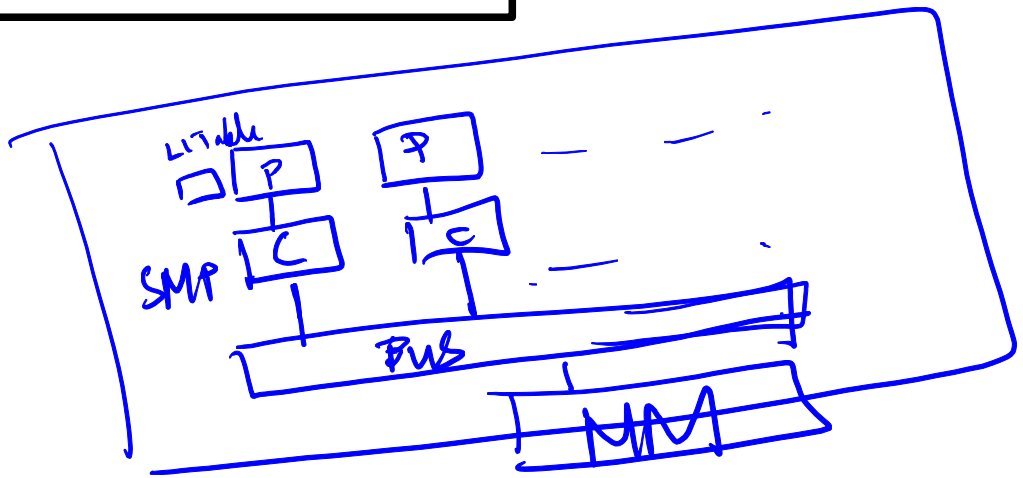
```
LD R2, X
....
....
....
SW R2, X
```

```
LL R2, X
....
....
....
SC R2, X
```

LL-Table

X:

→ monitor writes to X



LL-SC Example

Atomic execution required!

```
LD R2, X
....
....
....
SW R2, X
```

```
LL R2, X
....
....
....
SC R2, X
```

X: F

X:

LL-SC Example

Atomic execution required!

```
LD R2, X
....
....
....
SW R2, X
```

load link

```
LL R2, X
....
R2 ← R2 + 1
....
....
SC R2, X
```

X: ●

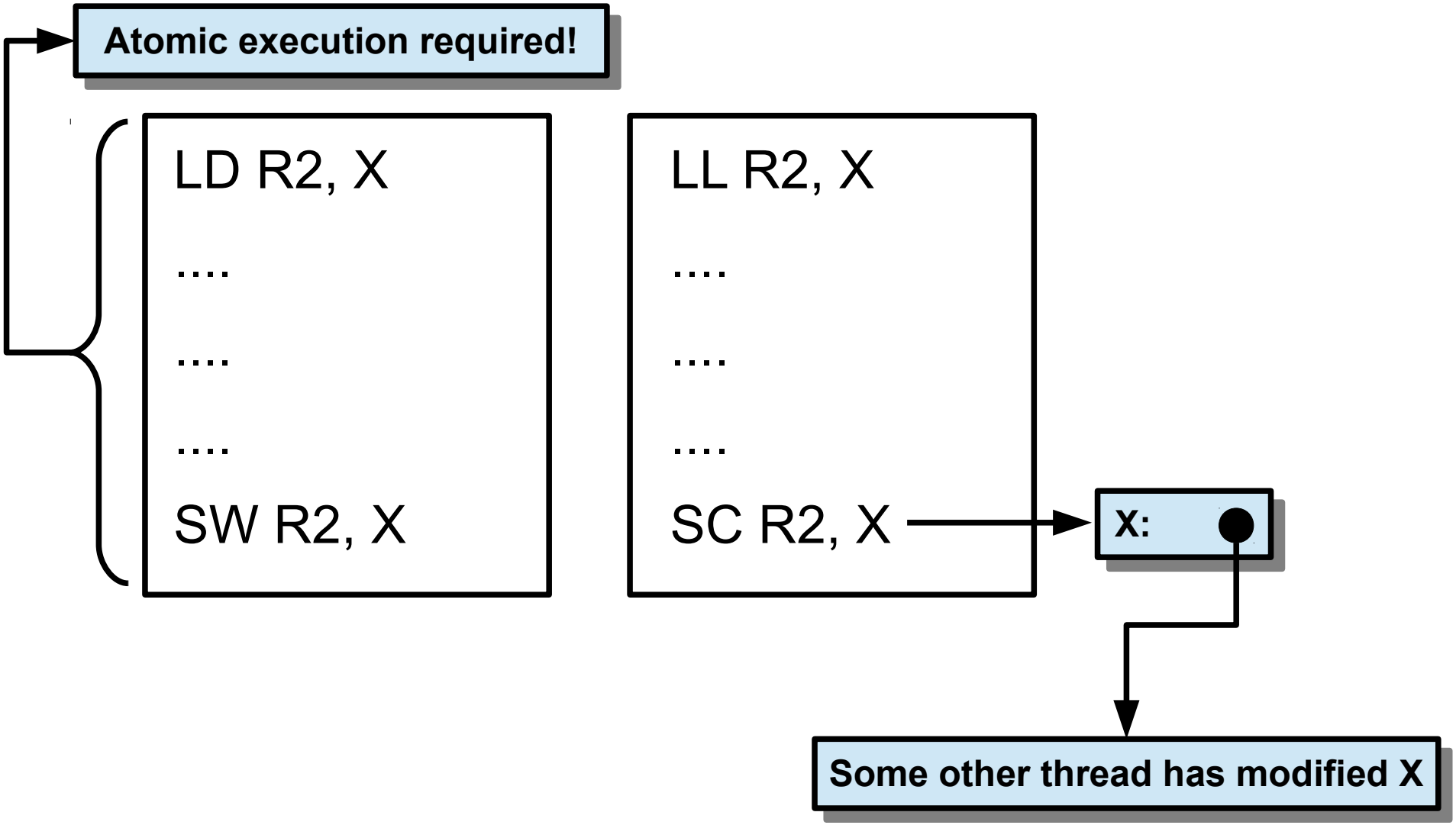
store conditional

wait() ⇒ interrupt

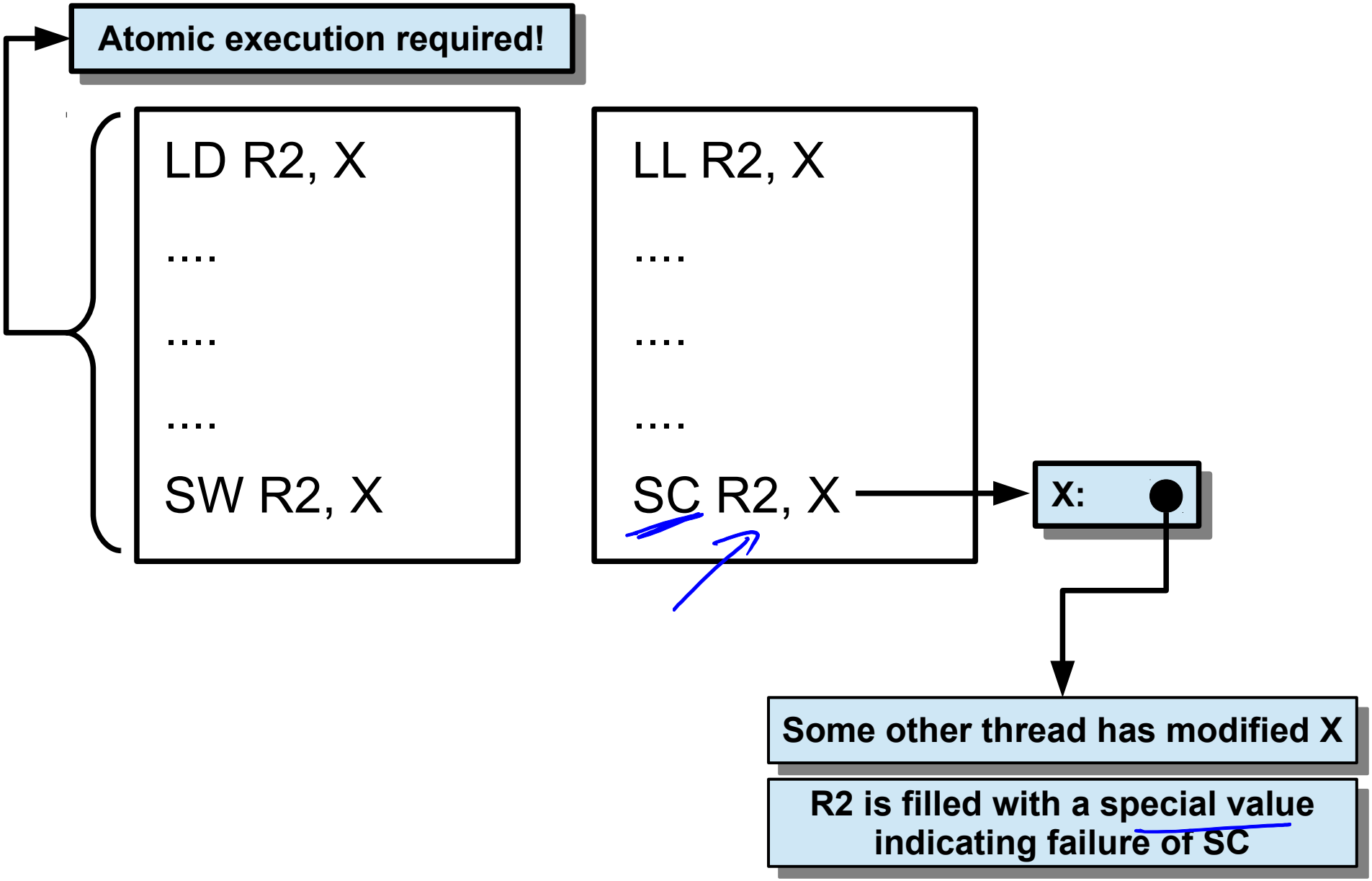
rfid

rfe()

LL-SC Example



LL-SC Example



LL-SC Example

Atomic execution required!

```
LD R2, X
....
....
....
SW R2, X
```

```
LL R2, X
....
....
....
SC R2, X
```

X:

X:

X:

someone else wrote X.

Some other thread has modified X

R2 is filled with a special value indicating failure of SC

```
if (R2 == )
...
```

Load Linked and Store Conditional

- LL records the lock variable address in a table

Load Linked and Store Conditional

- LL records the lock variable address in a table
- Snoop on the bus; Record the first Write Invalidate request on the lock variable in the table.

Load Linked and Store Conditional

- LL records the lock variable address in a table
- Snoop on the bus; Record the first Write Invalidate request on the lock variable in the table.
- Before the thread updates its cache copy, SC checks the table.

Load Linked and Store Conditional

- LL records the lock variable address in a table
- Snoop on the bus; Record the first Write Invalidate request on the lock variable in the table.
- Before the thread updates its cache copy, SC checks the table.
- If flag is set, SC fails (no coherence traffic). Fail is indicated by a special value in the register.

Load Linked and Store Conditional

- LL records the lock variable address in a table
- Snoop on the bus; Record the first Write Invalidate request on the lock variable in the table.
- Before the thread updates its cache copy, SC checks the table.
- If flag is set, SC fails (no coherence traffic). Fail is indicated by a special value in the register.
- If flag is not set, SC succeeds. Lock acquired.

Spin Lock using LL-SC

```
lockit:   LL R2, 0(R1)            ; no coherence traffic
          BNEZ R2, lockit         ; not available, keep spinning
          DADDUI R2, R0, #1       ; put value 1 in R2
          SC R2, 0(R1)            ; store-conditional succeeds if no one
                                  ; updated the lock since the last LL
          BEQZ R2, lockit         ; confirm that SC succeeded, else keep trying
```

Transactional Memory

Spin Lock using LL-SC

Spin lock with lower coherence traffic.

```
lockit:   LL R2, 0(R1)          ; no coherence traffic
          BNEZ R2, lockit       ; not available, keep spinning
          DADDUI R2, R0, #1     ; put value 1 in R2
          SC R2, 0(R1)          ; store-conditional succeeds if no one
                                ; updated the lock since the last LL
          BEQZ R2, lockit       ; confirm that SC succeeded, else keep trying
```

Outline

- Parallelism
- Flynn's classification
- Vector Processing
  - Subword Parallelism
- Symmetric Multiprocessors, Distributed Memory Machines
  - Shared Memory Multiprocessing, Message Passing
- Synchronization Primitives
  - Locks, LL-SC
- Cache coherence