CSCE 4213 Computer Architecture

Ch 5.5 Synchronization

David Andrews

Gaining Perspective

- First Consider Programming Model
  - What Does Synchronization Look Like to Programmer ?
- Operating System + Architecture Support
  - Machine Provides Basic Low Level Primitives
    - Combination of ISA + Cache Protocol
  - Operating System Abstracts into Callable API
    - Ties Scheduling into the Mix

Synchronization Basics

- Build With User-Level Software Routines
  - Policy Given by API
  - Mechanism within library routine
- Two Flavors, Shared Memory and Message Passing
  - Shared Memory Synchronization Through Semaphores
    - Mutex := Simple Binary Semaphore {0,1}
    - pthread_mutex_lock(mutex)
      - Will allow thread to continue if M[mutex] = 0;
        - Will also set M[mutex] = 1; (show locked)
      - Will block calling thread if M[mutex] =1
    - Pthread_mutex_release(mutex)
      - Will set M[mutex] = 0
  - Message Passing
    - int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

Programmers Perspective

- Mutex is "mutual exclusion".
- A mutex variable acts like a "lock" protecting access to a shared data resource.
  - Only one thread can lock (or own) a mutex variable at any given time. No other thread can own that mutex until the owning thread unlocks that mutex. Threads must "take turns" accessing protected data.
- Mutexes can be used to prevent "race" conditions.
  - An example of a race condition involving a bank transaction is shown below:

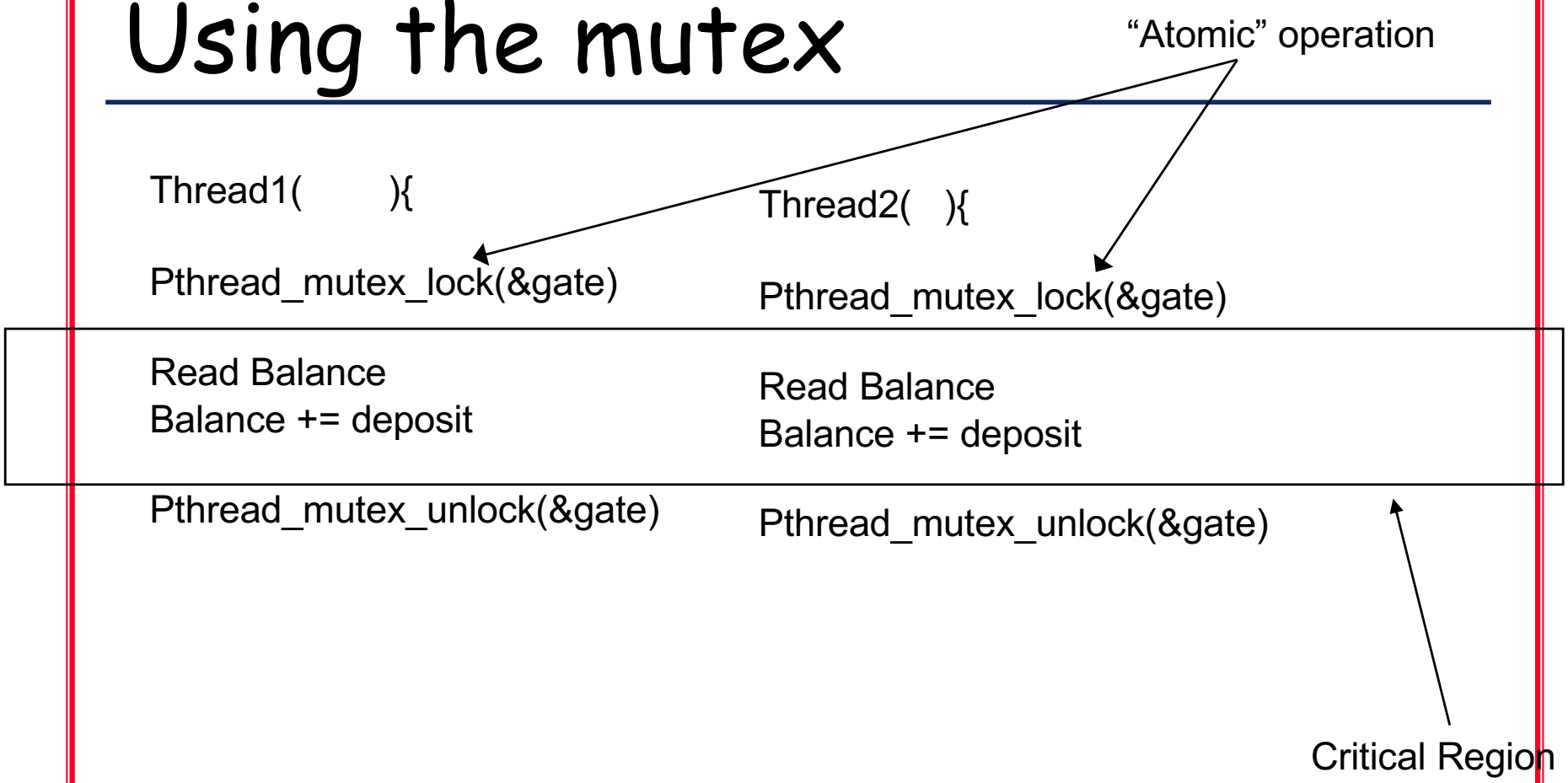
Thread 1	Thread 2	Balance
Read balance: $1000		$1,000
	Read Balance: $1,000	$1,000
		$1,000
Deposit $200		$1,000
Update Balance $1000 + $200 = $1,200		$1,200
	Update Balance $1000 + $200 = $1,200	$1,200

Using the mutex

Thread1( ){	Thread2( ){
Pthread_mutex_lock(&gate)	Pthread_mutex_lock(&gate)
Read Balance Balance += deposit	Read Balance Balance += deposit
Pthread_mutex_unlock(&gate)	Pthread_mutex_unlock(&gate)

Critical Region

Using the mutex



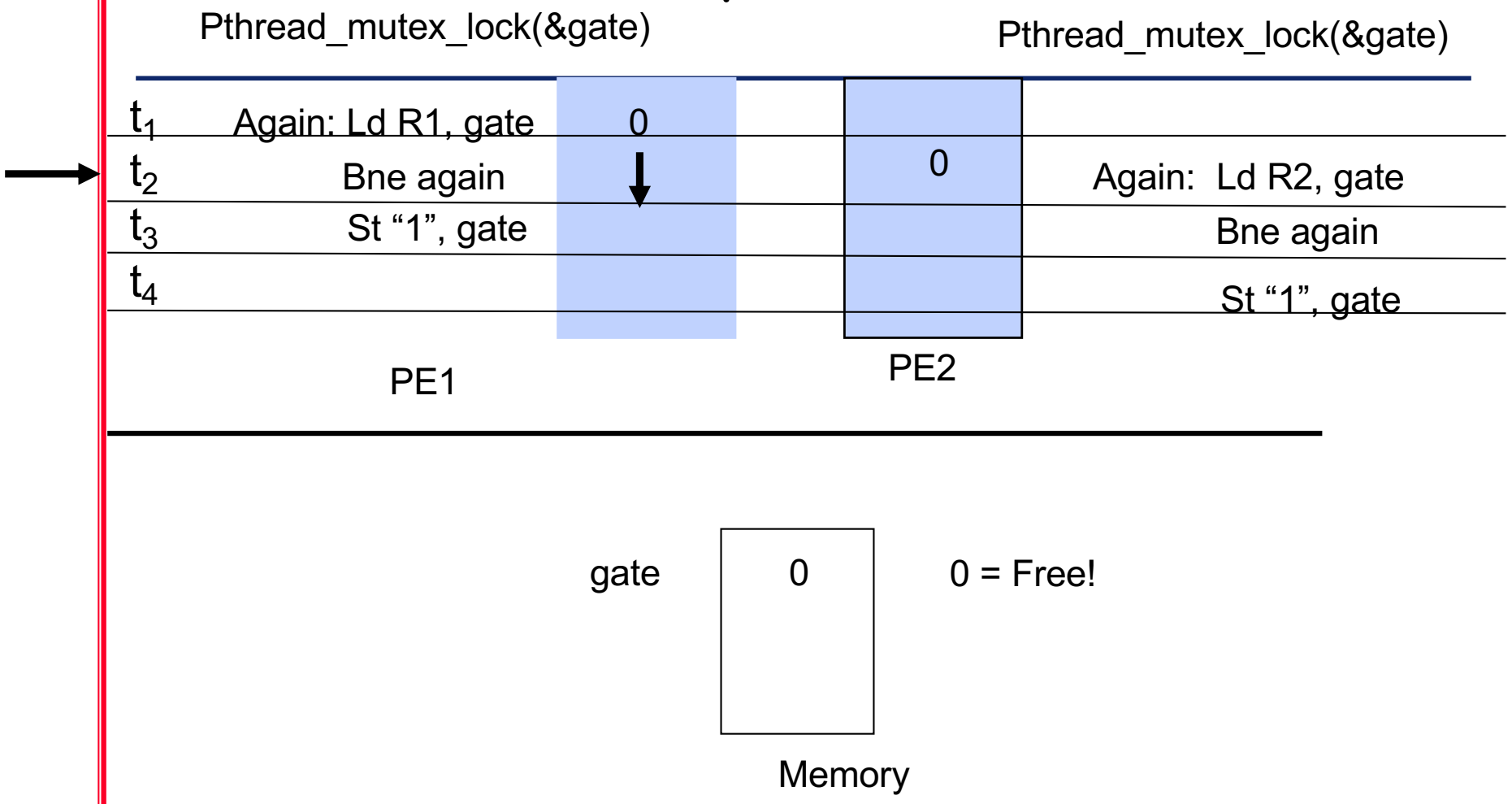
Problem for Multiprocessors if non-atomic

	Pthread_mutex_lock(&gate)			Pthread_mutex_lock(&gate)
t_1	Again: Ld R1, gate	0		
t_2	Bne again			Again: Ld R2, gate
t_3	St “1”, gate			Bne again
t_4				St “1”, gate
		PE1	PE2	

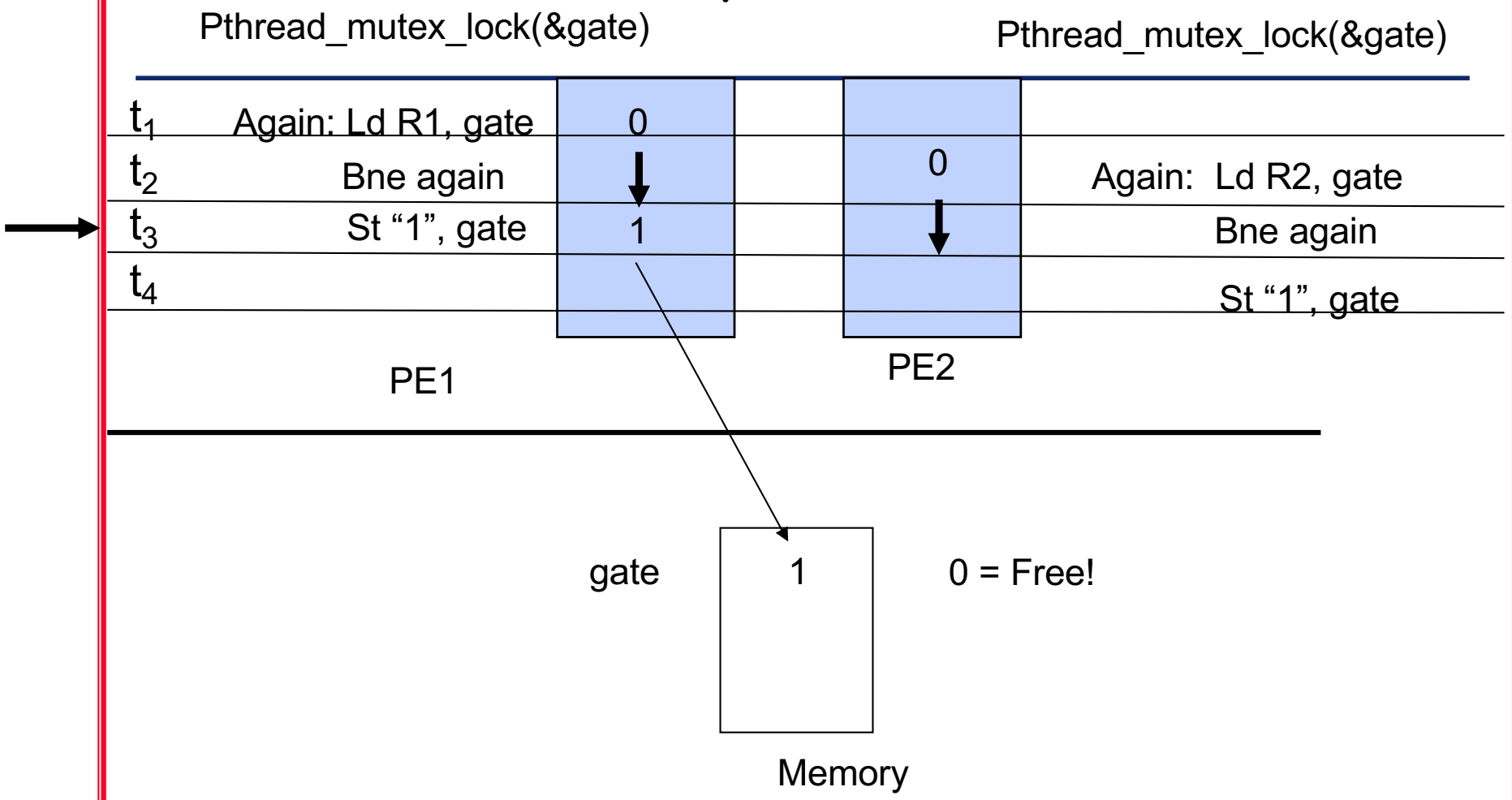
gate 0 0 = Free!

Memory

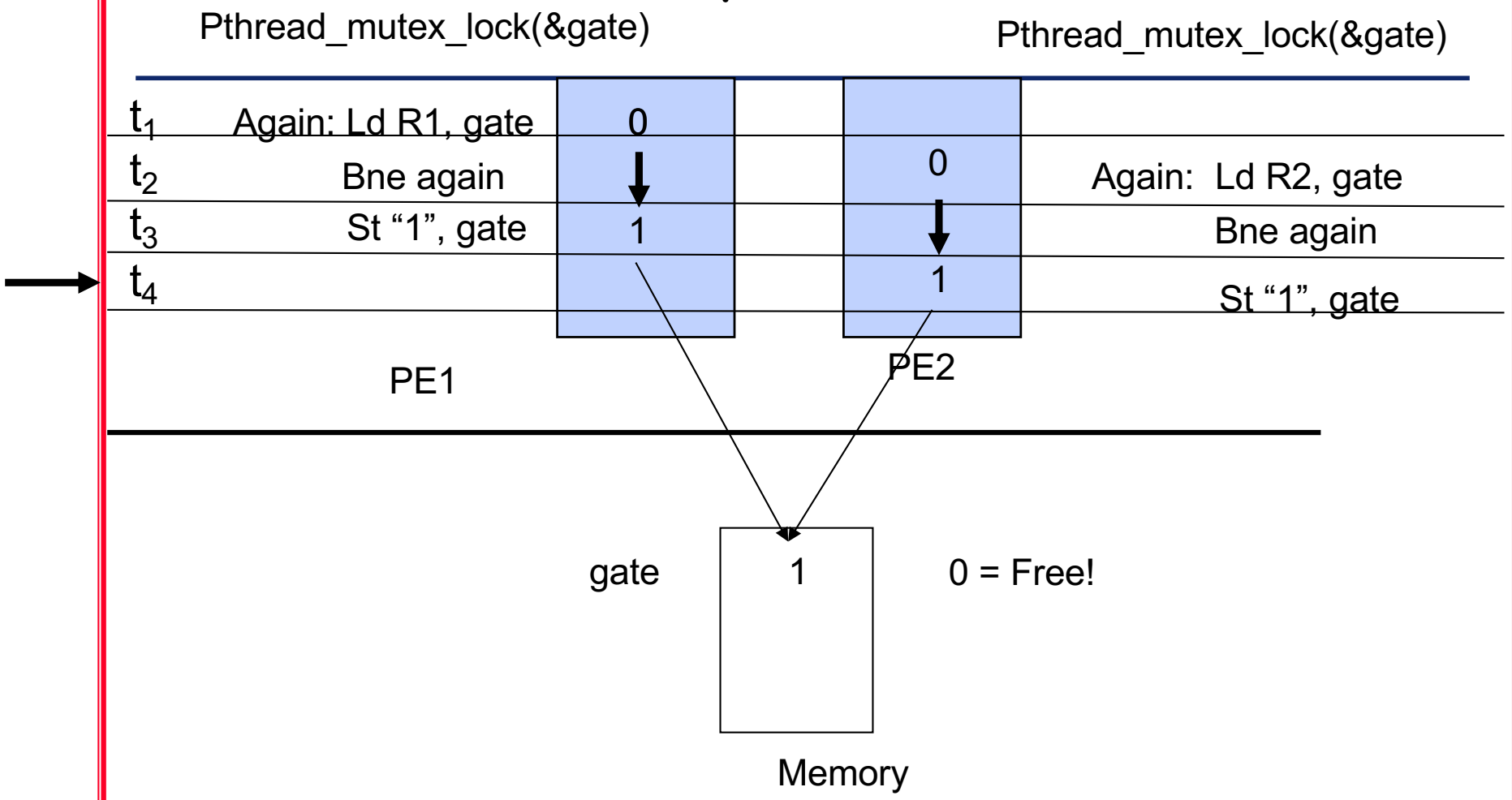
Problem for Multiprocessors if non-atomic



Problem for Multiprocessors if non-atomic



Problem for Multiprocessors if non-atomic



Uninterruptible Instruction to Fetch/Update Memory

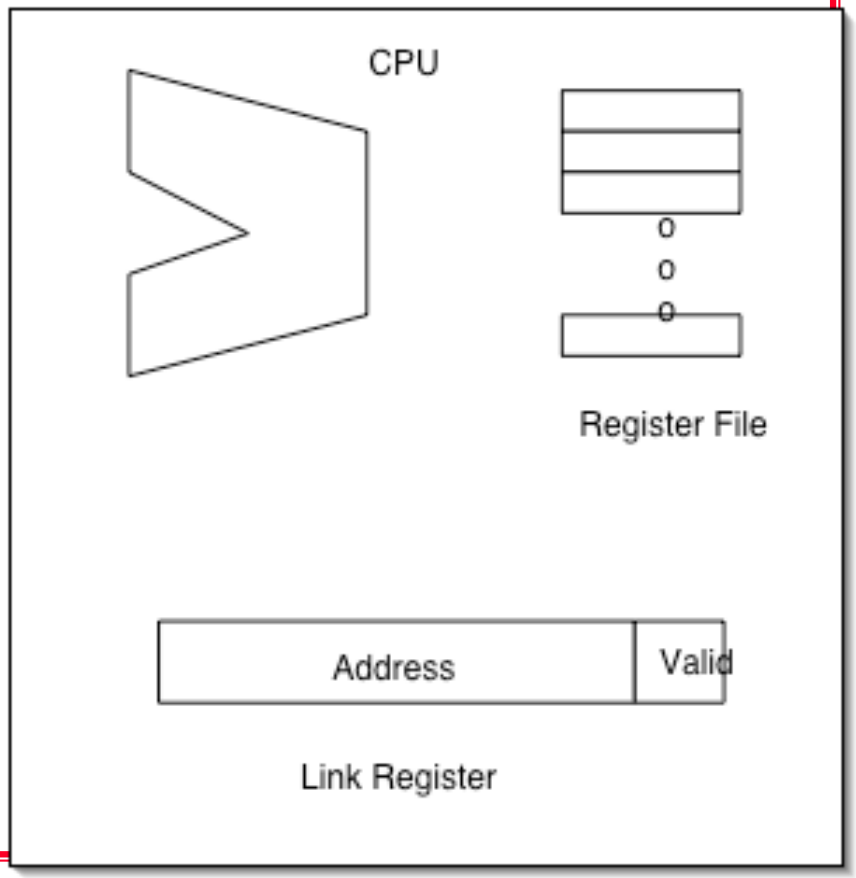
- Atomic exchange: interchange a value in a register for a value in memory
  - 0 => synchronization variable is free
  - 1 => synchronization variable is locked and unavailable
  - Set a register to 1 & swap with memory value
  - New value in register determines success in getting lock
    - 0 if you succeeded in getting the lock (you were first)
    - 1 if other processor had already claimed access
  - Key is that exchange operation is indivisible
- Original Atomic Instructions:
  - Test-and-set: tests a value and sets it if the value passes the test
  - Fetch-and-increment: it returns the value of a memory location and atomically increments it
  - Guaranteed Atomicity by locking bus
    - Very poor performance on a shared bus system

Uninterruptible Instruction to Fetch and Update Memory

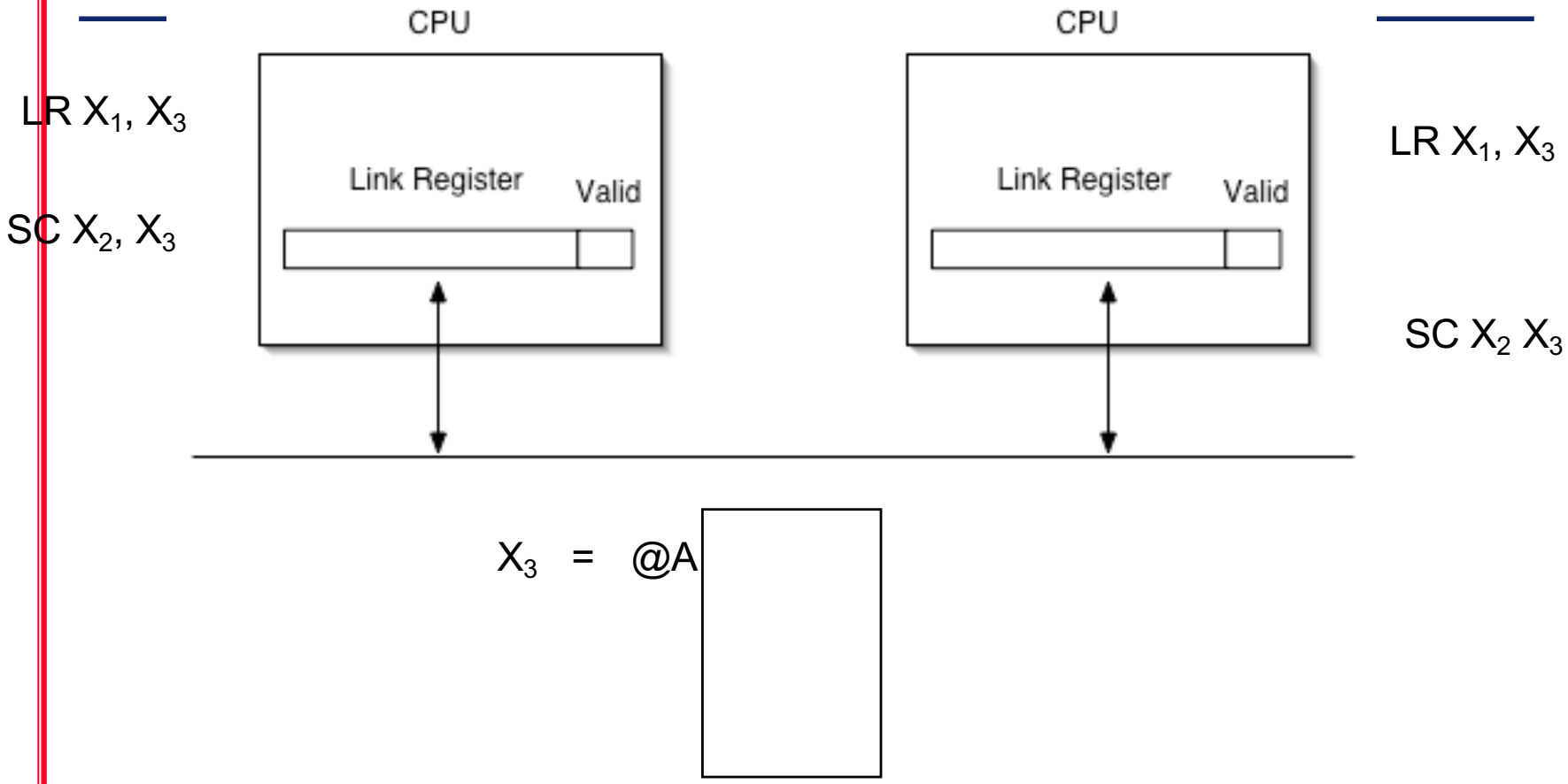
- Modern Version := 2 instructions
- Load linked (or load locked) + store conditional
  - Load linked returns the initial value
  - Store conditional returns 1 if it succeeds (no other store to same memory location since preceeding load) and 0 otherwise

```
LR X1,X2 {
   X1 <- Mem[X2]  sem value from memory
   link_reg <- X2    put address into link reg
   Valid <- 1        set valid bit
}

SC X1,X2  {
   if(X2 == link_reg && valid == 1)
       Mem[X2] <- X1
       X1 <- 0
   else
       X1<- 1
```



Architecture Support



Examples

- Example doing atomic swap with Lr & SC:

```
try:  mov     X3, X4          ; mov exchange value
      lr      X2, X1          ; load linked
      sc      X3, X1          ; store conditional
      bneqz   X3,    try      ; branch store fails , X3 = 1)
      mov     X4, X2          ; put load value in , X4
```

- Example doing fetch & increment with Lr & SC:

```
try:  lr      X2, , X1        ; load linked
      addi    X2,  X2, #1     ; increment (OK if reg-reg)
      sc      X2, , X1        ; store conditional
      bneqz   X2,    try      ; branch store fails (X2 = 1)
```

Architecture Support

- Link Register Doesn't Typically Sit on Bus
  - That's What the Cache is For !
  - Ties into Snoopy Cache Lines to Monitor Address
- Snoopy Cache Also Eliminates Bus Saturation
  - For Spin Locks, We Just Keep Trying…..
    - First Load Linked Actually a Read Miss
      - Address Stored in Both Cache and Link Register
        - Address marked as shared in cache
      - SC is Write
        - If some one else read SC doesn't actually happen (valid == 0)
        - Subsequent LL reloads are read from cache
        - If no-one else attempted to read then SC goes to Cache
        - Now Marked as Exclusive
        - Invalidates everyone else's cache copy
        - Subsequent reloads using LL cause new value to be written back and cache will be updated with new value

Interesting Performance Issues

- Suppose 5 "threads" waiting for value to change (release)
  - Lock gets set back to "0" (in cache) What happens ?
    - Exclusive in owner, invalidates in the 5 waiting caches
    - First requestor causes write back and update
      - Second invalidated and re-reads 0
      - Third requestor invalidated and re-reads 0
      - Fourth requestor invalidated and re-reads 0
      - Fifth requestor invalidated and re-reads 0
      - Hmmm……hardly seems fair to be quick….
    - Also, can cause poor performance through starvation
- For these reasons, blocking semaphores are used
  - Can control "release" order