Corso di Laurea in

Informatica

Approfondimento: Sincronizzazione

Architettura dei Calcolatori

Prof. Andrea Marongiu

andrea.marongiu@unimore.it

Anno accademico 2018/19

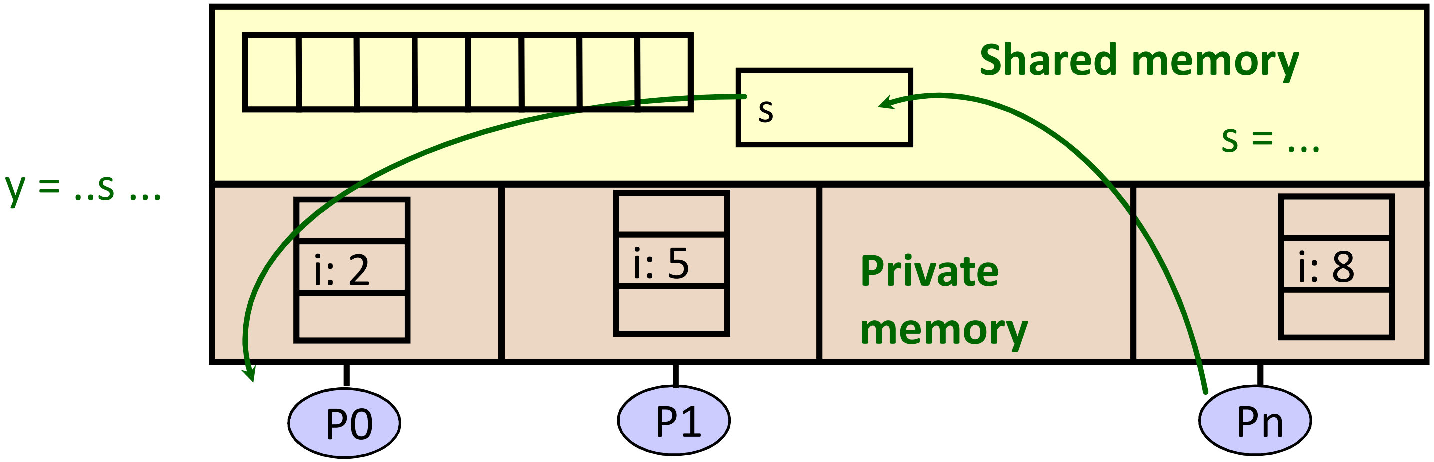
Motivation: “Too much milk”

- Example: People need to coordinate:

Time	Person A	Person B
3:00	Look in Fridge. Out of milk	
3:05	Leave for store	
3:10	Arrive at store	Look in Fridge. Out of milk
3:15	Buy milk	Leave for store
3:20	Arrive home, put milk away	Arrive at store
3:25		Buy milk
3:30		Arrive home, put milk away

Shared Memory Synchronization

- Program is a collection of **processors** (or *threads* of control).
- Each processor/thread has a set of **private variables** (e.g., local stack variables)
- Also a set of **shared variables**, (e.g., static variables, or global heap).
  - Processors communicate **implicitly** by writing and reading shared variables.
  - Processors coordinate by **synchronizing** on shared variables



Shared Memory code for computing a sum

```
static int s = 0;
```

Thread 1

```
  for i = 0, n/2-1
      s = s + sqr(A[i])
```

Thread 2

```
  for i = n/2, n-1
      s = s + sqr(A[i])
```

- Problem is a **race condition** on variable ***s*** in the program
- A race condition or data race occurs when:
  - two processors (or two threads) access the same variable, and at least one does a write.
  - The accesses are concurrent (not synchronized) so they could happen simultaneously

Shared Memory code for computing a sum

A | 3 | 5 |

f = square

static int s = 0;

Thread 1	Thread 2
.... compute f([A[i]) and put in reg0 reg1 = s reg1 = reg1 + reg0 s = reg1 ...	... compute f([A[i]) and put in reg0 reg1 = s reg1 = reg1 + reg0 s = reg1 ...

- Assume ***A*** = [3,5], ***f*** is the square function, and ***s***=0 initially
- For this program to work, ***s*** should be 34 at the end

Shared Memory code for computing a sum

A

3	5

f = square

static int s = 0;

Thread 1	Thread 2
....	...
compute f([A[i]) and put in reg0 9	compute f([A[i]) and put in reg0
reg1 = s	reg1 = s
reg1 = reg1 + reg0	reg1 = reg1 + reg0
s = reg1	s = reg1
...	...

- Assume ***A*** = [3,5], ***f*** is the square function, and ***s***=0 initially
- For this program to work, ***s*** should be 34 at the end

Shared Memory code for computing a sum

A | 3 | 5 | f = square

static int s = 0;

Thread 1	Thread 2
....	...
compute f([A[i]) and put in reg0 9	compute f([A[i]) and put in reg0 25
reg1 = s	reg1 = s
reg1 = reg1 + reg0	reg1 = reg1 + reg0
s = reg1	s = reg1
...	...

- Assume ***A*** = [3,5], ***f*** is the square function, and ***s***=0 initially
- For this program to work, ***s*** should be 34 at the end

Shared Memory code for computing a sum

A	3	5

f = square

static int s = 0;

Thread 1		Thread 2	
….		…	
compute f([A[i]) and put in reg0	9	compute f([A[i]) and put in reg0	25
reg1 = s	0	reg1 = s	
reg1 = reg1 + reg0		reg1 = reg1 + reg0	
s = reg1		s = reg1	
…		…	

- Assume ***A*** = [3,5], ***f*** is the square function, and ***s***=0 initially
- For this program to work, ***s*** should be 34 at the end

Shared Memory code for computing a sum

A | 3 | 5 | f = square

```
static int s = 0;
```

Thread 1		Thread 2	
….		…	
compute f([A[i]) and put in reg0	9	compute f([A[i]) and put in reg0	25
reg1 = s	0	reg1 = s	0
reg1 = reg1 + reg0		reg1 = reg1 + reg0	
s = reg1		s = reg1	
…		…	

- Assume ***A*** = [3,5], ***f*** is the square function, and ***s***=0 initially
- For this program to work, ***s*** should be 34 at the end

Shared Memory code for computing a sum

A | 3 | 5 |

f = square

static int s = 0;

Thread 1		Thread 2	
....		...	
compute f([A[i]) and put in reg0	9	compute f([A[i]) and put in reg0	25
reg1 = s	0	reg1 = s	0
reg1 = reg1 + reg0		reg1 = reg1 + reg0	25
s = reg1		s = reg1	
...		...	

- Assume ***A*** = [3,5], ***f*** is the square function, and ***s***=0 initially
- For this program to work, ***s*** should be 34 at the end

Shared Memory code for computing a sum

A | 3 | 5 |

f = square

static int s = 0;

Thread 1		**Thread 2**	
....		...	
compute f([A[i]) and put in reg0	9	compute f([A[i]) and put in reg0	25
reg1 = s	0	reg1 = s	0
reg1 = reg1 + reg0		reg1 = reg1 + reg0	25
s = reg1		s = reg1	25
...		...	

- Assume ***A*** = [3,5], ***f*** is the square function, and ***s***=0 initially
- For this program to work, ***s*** should be 34 at the end

Shared Memory code for computing a sum

A

3	5

f = square

static int s = 0;

Thread 1		Thread 2	
….		…	
compute f([A[i]) and put in reg0	9	compute f([A[i]) and put in reg0	25
reg1 = s	0	reg1 = s	0
reg1 = reg1 + reg0	9	reg1 = reg1 + reg0	25
s = reg1		s = reg1	25
…		…	

- Assume ***A*** = [3,5], ***f*** is the square function, and ***s***=0 initially
- For this program to work, ***s*** should be 34 at the end

Shared Memory code for computing a sum

A | 3 | 5 | f = square

static int s = 0;

Thread 1		**Thread 2**	
....		**...**	
compute f([A[i]) and put in reg0	9	**compute f([A[i]) and put in reg0**	25
reg1 = s	0	**reg1 = s**	0
reg1 = reg1 + reg0	9	**reg1 = reg1 + reg0**	25
s = reg1	9	**s = reg1**	25
...		**...**	

- Assume ***A*** = [3,5], ***f*** is the square function, and ***s***=0 initially
- For this program to work, ***s*** should be 34 at the end

Shared Memory code for computing a sum

A | 3 | 5 | f = square

static int s = 0;

Thread 1		Thread 2	
….		…	
compute f([A[i]) and put in reg0	9	compute f([A[i]) and put in reg0	25
reg1 = s	0	reg1 = s	0
reg1 = reg1 + reg0	9	reg1 = reg1 + reg0	25
s = reg1	9	s = reg1	25
…		…	

- Assume ***A*** = [3,5], ***f*** is the square function, and ***s***=0 initially
- For this program to work, ***s*** should be 34 at the end
  - but it may be 34, 9, or 25

Shared Memory code for computing a sum

```
static int s = 0;
```

Thread 1

```
local_s1= 0
for i = 0, n/2-1
   local_s1 = local_s1 + sqr(A[i])

s = s + local_s1
```

Thread 2

```
local_s2 = 0
for i = n/2, n-1
   local_s2= local_s2 + sqr(A[i])

s = s +local_s2
```

Shared Memory code for computing a sum

```
static int s = 0;
```

Thread 1

```
local_s1= 0
for i = 0, n/2-1
   local_s1 = local_s1 + sqr(A[i])

s = s + local_s1                ATOMIC
```

Thread 2

```
local_s2 = 0
for i = n/2, n-1
   local_s2= local_s2 + sqr(A[i])

s = s +local_s2                 ATOMIC
```

Atomic Operations

- To understand a concurrent program, we need to know what the **indivisible operations** are!

- Atomic Operation: an operation that always runs to completion or not at all
  - It is *indivisible:* it cannot be stopped in the middle and state cannot be modified by someone else in the middle
  - Fundamental building block – if no atomic operations, then have no way for threads to work together

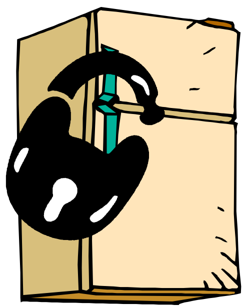
- On most machines, memory references and assignments (i.e. loads and stores) of words are atomic

Definitions

- Synchronization: using atomic operations to ensure cooperation between threads
  - For now, only loads and stores are atomic
  - hard to build anything useful with only reads and writes
- Mutual Exclusion: ensuring that only one thread does a particular thing at a time
  - One thread *excludes* the other while doing its task
- Critical Section: piece of code that only one thread can execute at once
  - Critical section and mutual exclusion are two ways of describing the same thing
  - Critical section defines sharing granularity

More Definitions

- Lock: prevents someone from doing something
  - Lock before entering *critical section* and before accessing *shared data*
  - Unlock when leaving, after accessing *shared data*
  - Wait if locked
    - Important idea: all synchronization involves waiting
- Example: fix the milk problem by putting a lock on refrigerator
  - Lock it and take key if you are going to go buy milk



Shared Memory code for computing a sum

```
static int s = 0;
```

Thread 1

```
local_s1= 0
for i = 0, n/2-1
    local_s1 = local_s1 + sqr(A[i])

s = s + local_s1
```

Thread 2

```
local_s2 = 0
for i = n/2, n-1
    local_s2= local_s2 + sqr(A[i])

s = s +local_s2
```

Shared Memory code for computing a sum

```
static int s = 0;
```

Thread 1

```
local_s1= 0
for i = 0, n/2-1
   local_s1 = local_s1 + sqr(A[i])

s = s + local_s1                ATOMIC
```

Thread 2

```
local_s2 = 0
for i = n/2, n-1
   local_s2= local_s2 + sqr(A[i])

s = s +local_s2                 ATOMIC
```

Shared Memory code for computing a sum

```
static int s = 0;
static lock lk;
```

Thread 1

```
local_s1= 0
for i = 0, n/2-1
   local_s1 = local_s1 + sqr(A[i])
lock(lk);
s = s + local_s1
unlock(lk);
```

Thread 2

```
local_s2 = 0
for i = n/2, n-1
   local_s2= local_s2 + sqr(A[i])
lock(lk);
s = s +local_s2
unlock(lk);
```

How to implement locks?

- Need HW support for **atomic** instructions
- RISCV uses two HW primitives
  - *Load reserved*
  - *Store conditional*
- (see discussion in *Chapter 2: Instructions, language of the computer*, slides 61-63)