50004 - Operating Systems - Lecture 6

Oliver Killane

04/11/21

Lecture Recording

Lecture recording is available here

Lock Granularity

```
1 void Extract(int acc_no , int sum)
2 {
3     lock(L);
4     int B = Acc[acc_no];
5     Acc[acc_no] = B - sum;
6     unlock(L);
7 }
```

$\Rightarrow$

```
1 void Extract(int acc_no , int sum)
2 {
3     lock(L[acc_no]);
4     int B = Acc[acc_no];
5     Acc[acc_no] = B - sum;
6     unlock(L[acc_no]);
7 }
```

- **Lock Overhead** measure of cost associated with a Lock:
  Consider memory use, initialisation cost & the cost of acquiring and releasing locks.
- **Lock Contention** measure of the number of processes waiting on a lock:
  More contention $\rightarrow$ less parallelism

Tradeoffs between fine and coarse grained synchronisation are context dependent, however it is always beneficial to keep critical regions as small as possible.

```
1 void AddAccount(int acc_no, int balance)
2 {
3     lock(L_Acc);
4     CreateAccount(acc_no);
5     lock(L[acc_no]);
6     Acc[acc_no] = balance;
7     unlock(L[acc_no]);
8     unlock(L_Acc);
9 }
```

Read/Write Locks

For two threads accessing data concurrently we can have race conditions if at least one is changing the shared data.

		Thread A	
		read	write
Thread B	read	Safe	Race
	write	Race	Race

```
1 void ViewHistory(int acc_no)
2 {
3     lock_RD(L[acc_no]);
4     print_transactions(acc_no);
5     unlock(L[acc_no]);
6 }
```

```
1 void Extract(int acc_no, int sum)
2 {
3     lock_WR(L[acc_no]);
4     Acc[acc_no] -= sum;
5     add_debit(acc_no, sum);
6     unlock(L[acc_no]);
7 }
```

- **lock_RD** Acquire the lock in read mode, if another lock has write mode, blocks.
- **lock_WR** Acquire lock in write mode, if any other thread has the lock in read or write mode, blocks.

Memory Models

In the course we assume sequential consistency:

- Operations in each thread occur in the order expressed in code/program order.
- Operations in each thread are executed in sequential order atomically.
- Hence race we can reason more easily about the behaviour of a thread in isolation.

However there are other memory models (due to compiler optimisations that split up or re-order operations and hardware (e.g core cache coherence)).

```
1 /* shared data */
2 int a, b;
```

```
1 void A() {
2     a = 1;
3     b = 1;
4 }
```

```
1 void B() {
2     b = 2;
3     a = 2;
4 }
```

Note that we assume a single write is atomic, and immediately visible to threads running on other cpu core, this is typically not the case.

Most architectures attempt to be cache coherent (illusion of coherent memory across cores), there are cases where this coherence is not achieved.

$a = 1$	$a = 1$	$a = 1$	$b = 2$	$b = 2$	$b = 2$
$b = 1$	$b = 2$	$b = 2$	$a = 2$	$b = 1$	$b = 1$
$b = 2$	$b = 1$	$a = 2$	$a = 1$	$a = 2$	$b = 1$
$a = 2$	$a = 2$	$b = 1$	$b = 1$	$b = 1$	$a = 2$
$2, 2$	$2, 1$	$2, 1$	$1, 1$	$2, 1$	$2, 1$

```
1 /* shared data */
2 int flag1 = 0, flag2 = 0;
```

```
1 void A()
2 {
3     flag1 = 1;
4     if (flag2 == 0)
5         critical();
6 }
```

```
1 void B()
2 {
3     flag2 = 1;
4     if (flag1 == 0)
5         critical();
6 }
```

With sequential consistency, it is impossible for $flag1 == flag2 == 0$ however under a weak memory model, due to the if statements not being dependent on the assignments, they may be reordered such that this is possible.

Happens-Before Relationship

Formulated by Leslie Lamport in 1976. It is a partial order realtion between events in a trace denoted by $a \to b$ where a, b are events in a trace.

Partial Order

A partial order is:

- **Irreflexive** $\forall a.[a \not\to a]$
- **Antisymmetric** $\forall a, b.[a \to b \Rightarrow b \not\to a]$
- **Transitive** $\forall a, b, c.[a \to b \wedge b \to c \Rightarrow a \to c]$

Consider a and b in a trace where a occurs before b.

- if a, b are in the same thread, then $a \to b$.
- if a, b are not in the same thread and $a : unlock(L)$ and $b : lock(L)$ then $a \to b$ (cannot lock until it is unlocked by other thread.)

There are other orderings deducible from the synchronisation primitives. A **Race Condition** occurs between a and b iff:

- the access the same memory location
- at least one is a write
- they are unordered according to **Happens-Before**

```
1 /* shared data */
2 int a, b;
```

```
1 void A() {
2     a = 1;
3     b = 1;
4 }
```

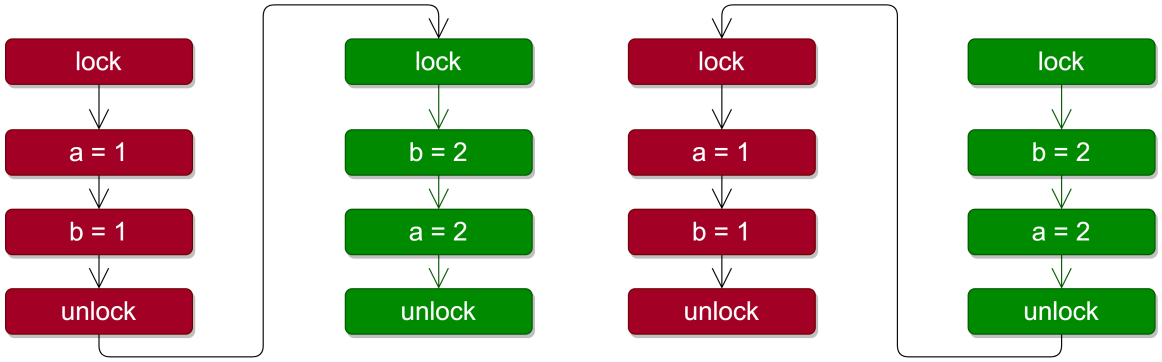
```
1 void B() {
2     b = 2;
3     a = 2;
4 }
```

Hence there is a data race between $a = 1, a = 2$ and $b = 2, b = 1$

```
1 void A() {
2     lock(L);
3     a = 1;
4     b = 1;
5     unlock(L);
6 }
```

```
1 void A() {
2     lock(L);
3     b = 2;
4     a = 2;
5     unlock(L);
6 }
```

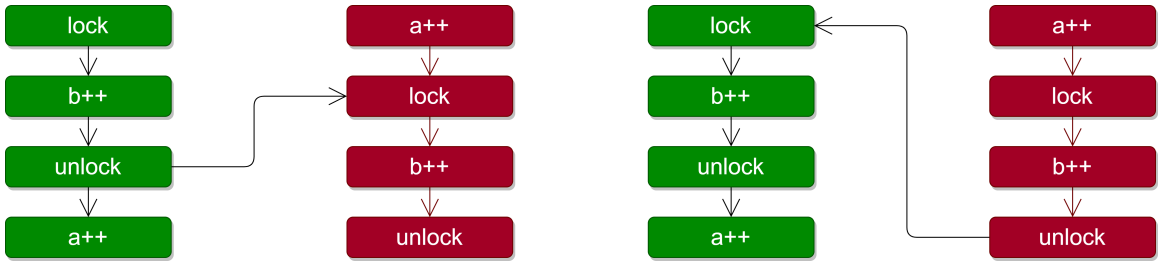
Two possible execution traces would be:



There are no race conditions in either.

```
1  void A() {
2      a++;
3      lock(L);
4      b++;
5      unlock(L);
6  }
```

```
1  void A() {
2      lock(L);
3      b++;
4      unlock(L);
5      a++;
6  }
```



Notice that while the right execution trace has no race condition, the left does.