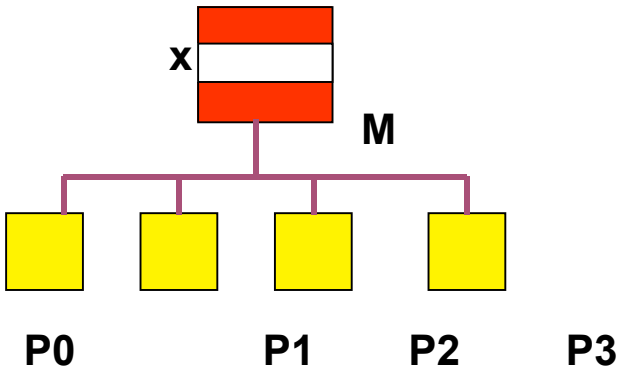
Multiprocessor Synchronization



Understanding interleaving semantics

Question. If each process executes x:= x+1, then what will be final value of x?

Answer. It may be 1 or 2 or 3 or 4. Why?

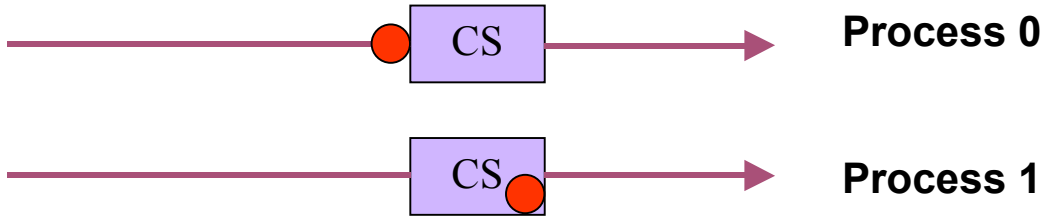
P0	P1	P2	P3
R ← x	R ← x	R ← x	R ← x
R ← R+1	R ← R+1	R ← R+1	R ← R+1
x ← R	x ← R	x ← R	x ← R

The final result depends on the interleaving pattern.

Atomicity of operations is important. It is determined by the largest sequence of operations that can be executed by any single process without interruption.

What kind of atomicity does a processor support?

The critical section problem



At most one process can be in the critical section at any time (mutual exclusion). The CS may involve shared data or resources.

Process 0	Process 1
LOCK	LOCK
CS operations	CS operations
UNLOCK	UNLOCK

How to implement locks? Depends on the kind of atomicity or granularity supported by the hardware.

What kind of atomicity a processor support?

In multiprocessors, ordinarily, read x and write x (x is a shared data in the memory) are the only two atomic operations. Can we implement locks using read x and write x only?

First attempt.

{free is a shared boolean, initially true }

Processor 0	**Processor 1**
while not free **do** nothing;	**while** not free **do** nothing;
free := FALSE;	free := FALSE;
{CS codes};	{CS codes};
free := TRUE	free := TRUE

Does not work. Why? Safety violation!

Second attempt

{**turn0** and **turn1** are shared boolean, initially false}

P0	**P1**
turn 0 := TRUE;	turn 1 := TRUE
while turn1 **do** nothing;	**while** turn0 **do** nothing;
{CS codes}	{CS codes}
turn0 := FALSE;	turn1 := FALSE

Does not work. Why? Danger of deadlock!

First solution proposed by Dekker in 1965. It implements a spin-lock. To implement a queuing lock, you need the help of the OS.

What is the difference between spin-lock and queuing lock?

Implementing Spin-Lock

The simplest solution is due to Gary Peterson (1979):

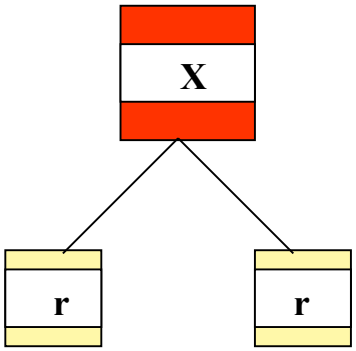
Process 0	**Process 1**
try(0),try(1) : shared variables, initially false.	
{Lock}	{**Lock**}
try(0):= true ;	try(1) := true;
turn := 0;	turn := 1;
while (turn=0 && try(1))	**while** (turn=1 && try(0))
do nothing;	**do** nothing;
critical section;	***critical section***;
{Unlock}	{**Unlock**}
try(0) := false	try(1):= false

It works. How will you generalize it to N processes?

Plan a tournament.

Atomic Read-Modify-Write (RMW) instructions

Locking is frequently needed, so it must be efficient.



Test & Set (X) **(return X; X:= 1)**

Swap (r,X) **(r, X := X, r)**

Fetch & Add (X) **(return X; X:=X+1)**

These simplify the implementation of locking.

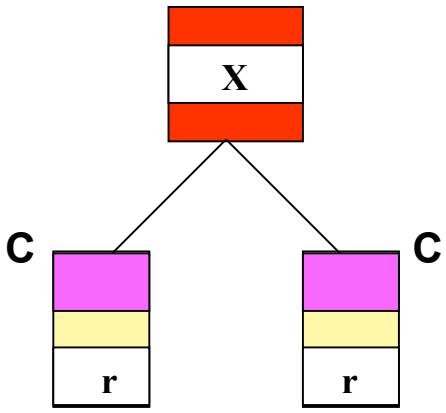
How to implement an RMW instruction?

Implementing Spin-lock using Test & Set (TS)

```
*Initially,      X=0*

lock:       r:= TS(X);

            if r ≠ 0  then goto lock;

            critical section;

unlock:     X:=0
```

Note that if CS is large, then bus-bandwidth is wasted. This is bad.

How to save bus-bandwidth using cache?




```
                *Initially  X=0, r=1*
lock:       if  X= 0 then r:= TS (X);
            if r  ≠ 0 then goto lock;
            critical section;
unlock:     X:=0
```

Note. As long as X=1, X is read from the local cache, When X changes to 0 in the shared memory, the cache entry is invalidated.

Load-Linked, Store Conditional (LL, SC)

First used by DEC Alpha for process synchronization.

Works with a snooping cache.

LL r, x loads the value of x into register r, and saves the address x into a *link register*.

SC r, x stores r into address x only if it is the first store (after LL r, x). The success is reported by returning a value (r=1). Otherwise, the store fails, and (r=0) is returned.

Example. Implement **atomic** x:= x+1 using **LL, SC**

```
*Initially x=0*

lock:  LL r1, x;

       r1 := r1+1;

       SC r1, x;

       if r1 = 0 then goto lock
```

Unlike the RMW instructions, there is no need to lock the bus.