Multiprocessor Synchronization

Mutual exclusion using **atomic swap**

```
        r := 1; X:=0 {X is the lock}
lockit: swap (r, X);
        if r = 1 then goto lockit;
        {critical section}
        X:=0
```

Study the above spin-lock implementation. How to save bus-bandwidth?

A more *efficient* solution using local cache

```
         r := 1; X:=0
lockit:  if X=0 then **swap** (r, X);
         **if** r    0 **then goto** lockit;
         {critical section}
         X:=0
```

What is the maximum number of bus cycles used when N processes enter (and exit) their critical sections exactly once?

$$2N+1 + 2(N-1) + 1 + 2(N-2) + 1 + \ldots + 2 + 1$$

$= \quad N^2 + 2N$ i.e $O(N^2)$

Load-Linked, Store Conditional (LL, SC)

First used by DEC Alpha for process synchronization.

- **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.

Used by MIPS

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

```
*Initially x=0*

lock:  LL r, x;

       r := r+1;

       SC r, x;

       if r1 = 0 then goto lock
```

Unlike the RMW instructions, there is no need to lock the bus, yet it implements an atomic operation

Solution to Mutual Exclusion using LL, SC

```
        X:=0
        {X is the lock, now it is unlocked}
lockit: LL r, X;
        if r neq 0 then goto lockit;
        r:=1;
        SC r, X;
        {The first successful SC grabs the lock}
        if (r = 0) then goto lockit {failure} else
        {Critical section}
        X:=0 {Lock released}
```

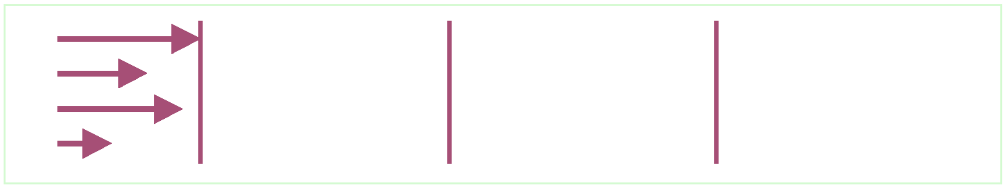
Unlike an RMW operation, the bus is not locked for multiple cycles, but the bandwidth is wasted.

Synchronization in large scale microprocessors

Use exponential back off. When a process fails to grab the lock, it will try after a time delay that will increase exponentially after every failure.

```
        X:=0      {This is the lock}
        r3 :=1    {delay stored in r3}
lockit: LL  r, X;
        if (r neq 0) then goto lockit end if;
        r := 1; SC r, X;
        if (r = 0) then goto gotit;
            r3 := r3 + r3; Pause r3 {Try after pause}
            goto lockit
        end if
gotit:  critical section
        X:= 0
```

Barrier Synchronization



Phase (i+1) does not begin until every process completes phase i. Useful in parallel loops.

```
Shared var      count: integer {initially = 0}
                release: boolean
{lock}
if    count = 0
      then release = false {first process}
end if
count = count + 1      {lock maintains atomicity}
{unlock}
if    count = N (last process)
      then count: = 0; release: = true
      else wait for (release = true) {spin lock}
end if
```

Fetch-and-Add

FA(X, v) ≡ <Return X; X:= X+ v> {atomic operation}

Read-modify-write operation (costs 2 memory cycles}

First used in NYU Ultracomputer to implement locking and drastically reduces memory contention in a novel way. Now included in the instruction set of IA-64.

Example

Let there be N processes.

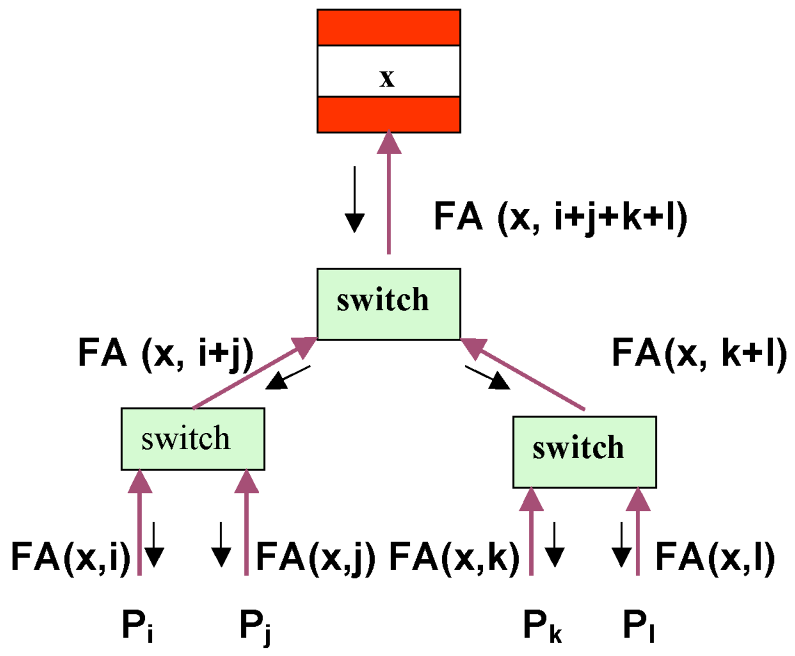
Each process i executes FA(X, i) ≡ <X:= X+ i>

How many memory accesses are required?

2N   {Better than $O(N^2)$, but can be further improved}

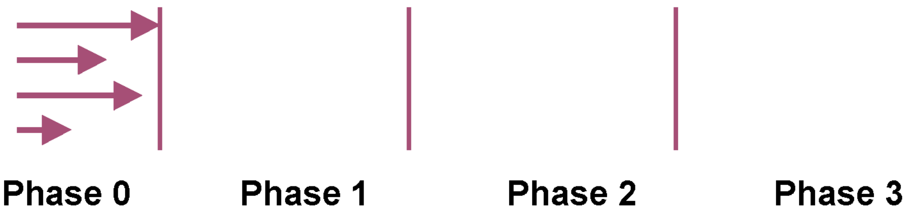
Fetch-and-Add (continued)

Contention can be further reduced using combining switches (only one RMW is enough)



Each switch returns a value that mimics the sequential operation

Barrier Synchronization using Fetch & Add



Phase (i+1) does not begin until every process completes phase i.

```
*initially count = 0, release = false*

FA (count, 1);

if (count = N )

        then release := true;

        else wait for (release=true) do skip;
```

release & count re-initialized before the next phase begins

Requires at most N FA operations. Costs N cache misses for count, and N cache misses for release.