PENNSTATE

CSE 511: Operating Systems

Spinlocks on Shared Memory Multiprocessors

Lectures 29-30

PENNSTATE

Lock Synchronization

Two common techniques for implementing synchronization

- Blocking
- Busy waiting (spin locks)

When do you choose one approach over another?

Recall: Assumed Hardware Support

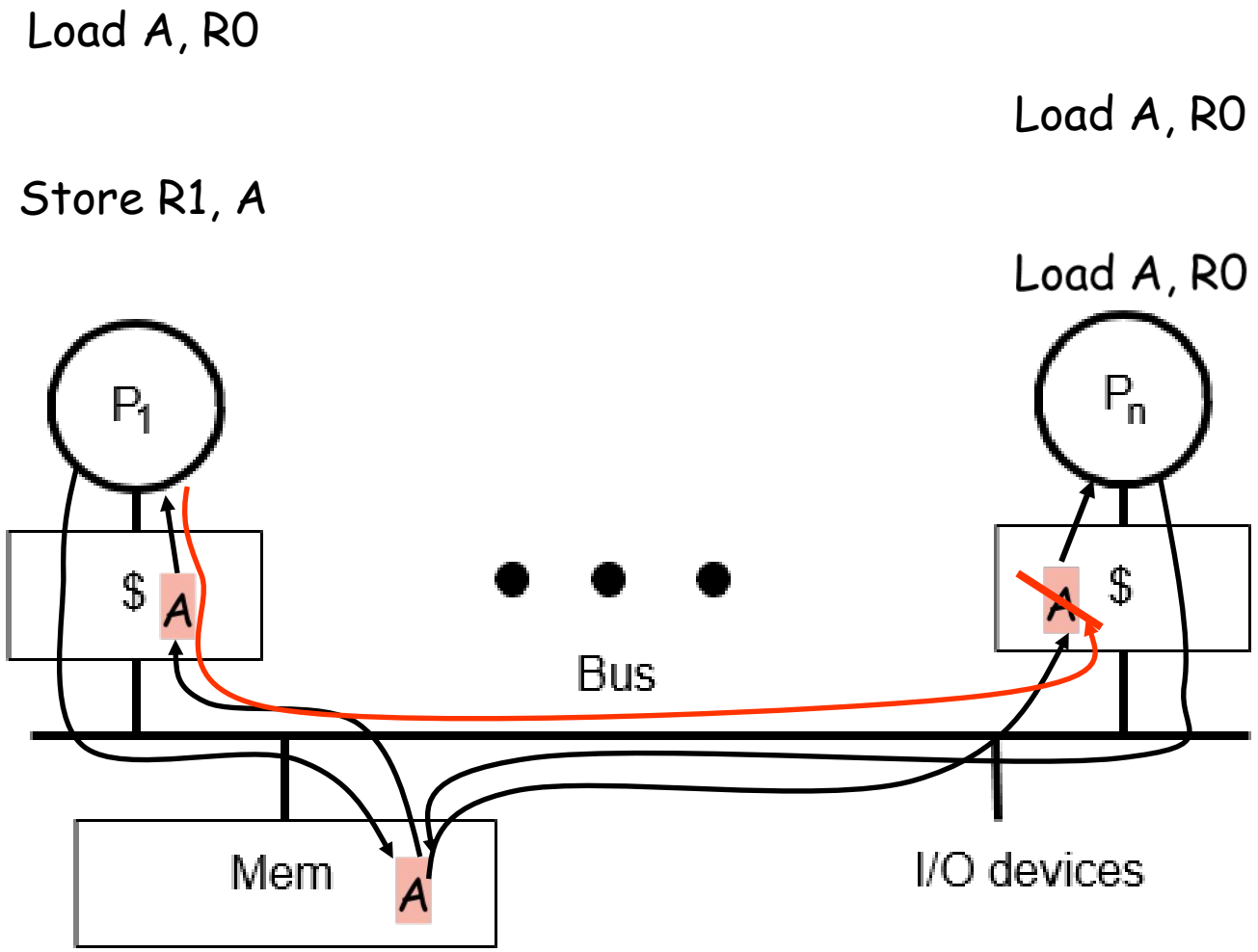
- Atomic Instructions (e.g. test&set, fetch&add,…)
- Atomic test&set(x) {

```
temp = x;
x = BUSY;
Return(temp);
}
```

- Atomic fetch&Add(x,y) {

```
temp = x;
X = x + y;
return(temp);
}
```

Bus-based Shared Memory Multiprocessors

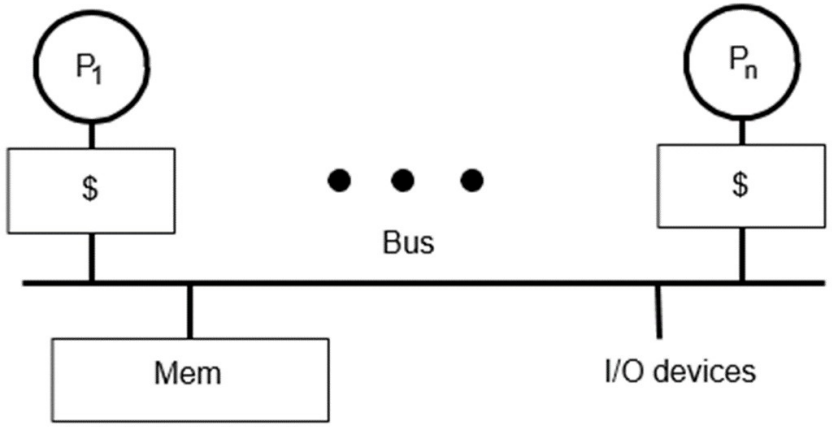


What about non-bus based systems?

1. Spin on test&set

```
Lock() {
        while (test&set(x) == BUSY)
                ;
}

Unlock() {
        x = CLEAR;
}
```

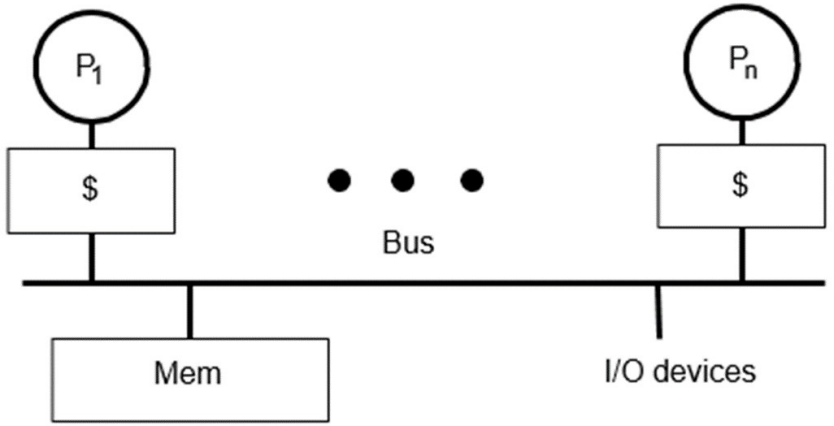


Problems:
-Each attempt creates network traffic
-Unlock also contends with trying for a lock

2. Spin on Read (test-test&set)

```
Lock() {
        while (x == BUSY  ||  test&set(x) == BUSY)
                ;
}

Unlock() {
        x = CLEAR;
}
```



Problems:
-Gap between detecting the lock is clear and trying for it
allows more people to try for it than required
-Cached copies get invalidated with further (possibly unsuccessful attempts)
-Time for network activity to settle down after lock is released (Quiesce Time)
- For reasonable performance: Critical Section >> Quiesce Time

PENNSTATE

3. Delay after Spinning Processor Notices lock is released

```
Lock() {
        while (x == BUSY || test&set(x) == BUSY) {
                while (x == BUSY)
                        ;
                Delay();
        }
}

Unlock() { x = CLEAR}
```

Problems:
-Delay based solutions are not necessarily the best solutions.
-How do you set the delays?
-In invalidation-based schemes, the local busy wait may still incur misses (even if unsuccesful) because of other attempts.

PENNSTATE

4. Delay between each memory reference (for invalidation based schemes)

```
Lock() {
        while (x == BUSY || test&set(x) == BUSY) Delay();
}

Unlock() { x = CLEAR; }
```

Problems:
-Most of the problems of previous solution persist.

PENNSTATE

5. Ticket Lock

```
int next_ticket, now_serving = 0;

Lock() {
        int myticket;

        myticket = Fetch&Add(next_ticket,1);
        while (myticket != now_serving)
                ;
}

Unlock() { now_serving++ }
```

Problems:
-Everyone reading and writing next_ticket, now_serving (particularly the latter)

PENNSTATE

6. Array-based Queueing (Anderson Lock)

```
int Flags[0..P-1];  // Flags[0] = HAS_LOCK and rest are MUST_WAIT
Int tail = 0;

Lock() {
        int myplace = Fetch&Add(tail,1);
        while (Flags[myplace % P] == MUST_WAIT)
                ;
        Flags[myplace %P] = MUST_WAIT;
}
Unlock() { Flags[myplace+1 % P] = HAS_LOCK; }
```

Problems:
-Need to use arrays
-Need to worry about false sharing
-You have no control which position you will wait on. On a system without caches (which is not really the case anymore), this can be bad