WASHINGTON STATE UNIVERSITY
VANCOUVER

CS 563 Concurrent Programming

Lecture 12: Locks

Critical Section Problem

- what?
  - implementing (often large) atomic actions in software
- why?
  - linked lists in OSes, database records, counters, etc.

Implementing Atomic Actions

- Spin locks (busy waiting)
  - multiprocessor OS or parallel program
- Blocking primitives (e.g., semaphores)
  - higher-level parts of an OS or multithreaded programs

Model for CS Problem

```
process CS[i = 1 to n] {
  while (true) {
     CSenter: entry protocol;
     critical section;
     CSexit:  exit protocol;
     noncritical section;
  }
}
```

Model for CS Problem

- Specifying mutual exclusion
  - `int in[1:n] # initially all zero`
  - `in[i] = 1 when process i is in its critical section`
  - `at all times require`
    - `MUTEX:  0 <= sum of in[i] <= 1`

Spin Locks

- How can we solve the CS problem using machine instructions directly?

Spin Locks

- Observation -- there are only 2 key states:

```
nobody is in its CS
   lock == false
somebody is in its CS
   lock == true
```

Spin Lock

- Using just lock, we get the following code:

```
⟨ await (!lock) lock = true; ⟩
critical section
lock = false; # angle brackets needed here?
```

Test and Set

- The first instruction for implementing spin locks (IBM, mid 1960s)

```
bool TS(bool lock) { # an atomic instruction
   ⟨ bool initial = lock;
     lock = true;
     return initial; ⟩
}
```

Using TS

- We get the following simple solution

```
CSenter: while (TS(lock)) skip;

CSexit: lock = false  # simply reinitialize
```

Properties

```
bool TS(bool lock) { # an atomic instruction
     ⟨ bool initial = lock;
       lock = true;
       return initial; ⟩
  }
```

```
CSenter: while (TS(lock)) skip;

CSexit: lock = false;
```

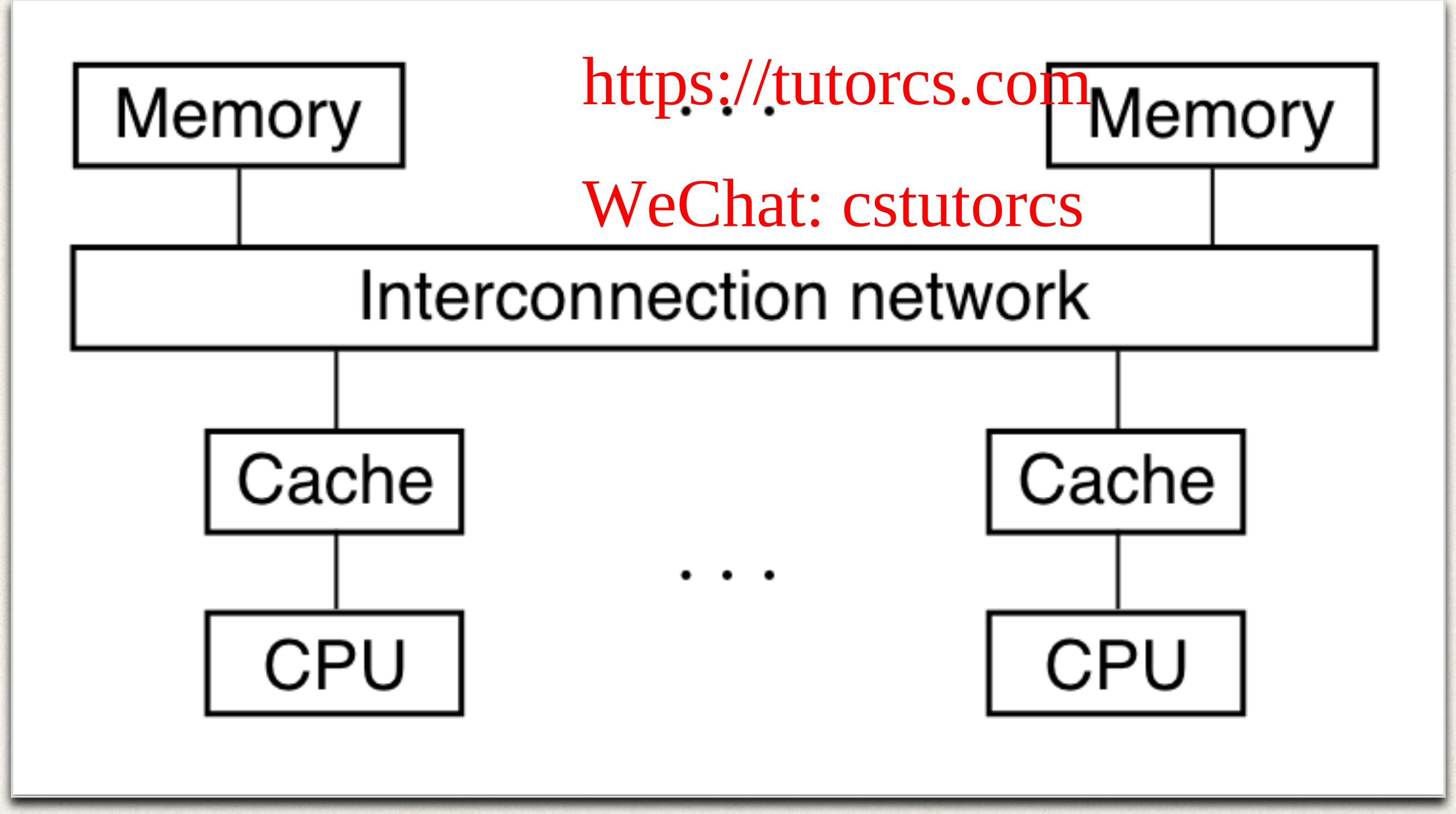
- Mutual exclusion
- Absence of deadlock (livelock)
  - (with weakly fair scheduling)
- Absence of unnecessary delay
  - (with weakly fair scheduling)
- Eventual entry (fairness)
  - not fair -- no GUARANTEE of eventual entry

Problems with TS

- Efficiency
- Fairness

TS Efficiency

- Shared memory multiprocessors



Performance of Test and Set

- TS reads AND writes a lock
- Best case (no contention), i.e. lock is free, 1 process wants in:
    - read lock (50 clocks)
    - write lock (50 clocks)
    - execute CS
    - write lock (1 or 50 clocks)
    - repeated usage by the same process gets cheap reads

```
bool TS(bool lock) { # an atomic instruction
     ⟨ bool initial = lock;
       lock = true;
       return initial; ⟩
   }
```

Performance of Test and Set

- Worst case -- n processes all trying to get into their CS
  - 1 process does read and write and succeeds (100 clocks)
  - other n-1 processes do read, write, fail, repeat
  - hence, the bus is jammed AND the first process might get delayed when it wants to release the lock

Test and Test and Set

```
CSenter:  while (lock) skip;            # test
       while (TS(lock))       # test and set
        while(lock) skip;   # test again

CSexit:   lock = false;
```

```
bool TS(bool lock) { # an atomic instruction
     ⟨ bool initial = lock;
       lock = true;
       return initial; ⟩
  }
```

- One extra clock in best case; no write (or bus use) while spinning

Implementing Await Statements

- We can use a spin lock solution to implement any kind of await statement and hence any kind of atomic action

```
⟨ S; ⟩

 CSenter; S; CSexit;

⟨ await(B) S; ⟩

 CSenter;
 while (!B) { CSexit; Delay; CSenter; }
  S;
 CSexit;
```

Fair Solutions to the CS Problem

- Need a fair way to break ties

Tiebreaker Algorithm

```
bool in1 = false, in2 = false;
int last = 1;
process CS1 {
    while (true) {
    last = 1; in1 = true; /* entry
                          protocol */
    ⟨await (!in2 or last == 2);⟩
    critical section;
    in1 = false; /* exit protocol */
    noncritical section;
    }
}
```

```
process CS2 {
    while (true) {
    last = 2; in2 = true; /* entry
                          protocol */
    ⟨await (!in1 or last == 1);⟩
    critical section;
    in2 = false; /* exit protocol */
    noncritical section;
    }
}
```

Ticket Algorithm

```
shared:  int number = 1, next = 1;
CSenter:  int myturn;   # private variable;
                        # one copy per process
          〈 myturn = number; number++; 〉
          〈 await(myturn == next); 〉
CSexit:  〈 next++ 〉  # different variable,
                     # not a spin lock
```

Fetch and Add Instruction

- Read and increment a variable as a single atomic action:

```
int FA(var, incr) {
  ⟨ int tmp = var; var += incr; return (tmp); ⟩
```

- Ticket drawing is then simply

```
myturn = FA(number, 1);
```

- Pros and Cons:
  - Fair
  - But, hardware has to provide an FA or similar instruction

Bakery Algorithm

```
int turn[1:n] = ([n] 0);

   process CS[i = 1 to n] {
     while (true) {
      ⟨turn[i] = max(turn[1:n]) + 1;⟩

      for [j = 1 to n st j != i]

      ⟨await (turn[j] == 0 or turn[i] < turn[j]);⟩

      critical section;

      turn[i] = 0;

      noncritical section;

      }

  }
```

```
int turn[1:n] = ([n] 0);
  process CS[i = 1 to n] {
    while (true) {
      turn[i] = 1;
      turn[i] = max(turn[1:n]) + 1;
      for [j = 1 to n st j != i]
        while (turn[j] != 0 and
          (turn[i],i) > (turn[j],j)) skip;

      critical section;

      turn[i] = 0;

      noncritical section;

    }

  }
```