Course 02158

Barriers and Memory Models

Hans Henrik Løvengreen

DTU Compute

Phased Computation

- Let $task_i$ be some computation step for datapart i
- **process** *Master*
  loop
  co $[i : 1..n]$
  do $task_i$
  oc
  end loop
- **process** *Worker*$[i : 1..n]$
  loop
  do $task_i$
  wait for all tasks to complete ← **BARRIER**
  end loop

Barrier with Shared Variables

Simple solution

- **var** *count* : *integer* := 0;
- **process** *Worker*[*i* : 1..*n*]
  loop

  do task$_i$

  $\langle\, count := count + 1 \,\rangle$;
  await *count* = *n*

  end loop

- Works only once

Barrier with Shared Variables

Cyclic barrier attempt

- **var** *count* : *integer* := 0;
- **process** *Worker*[*i* : 1..*n*]
  loop

  do task$_i$

  $\langle\, count := count + 1 \,\rangle$;
  if *count* = *n* **then** *count* := 0;
  await *count* = 0

  end loop

- Race condition: May get stuck

Barrier with Shared Variables

Double barrier

- **var** $count_1, count_2 : integer := 0;$
- **process** $Worker[i : 1..n]$
  loop

  do $task_i$

  $\langle count_1 := count_1 + 1 \rangle;$
  if $count_1 = n$ **then** $count_1 := 0;$
  await $count_1 = 0$

  $\langle count_2 := count_2 + 1 \rangle;$
  if $count_2 = n$ **then** $count_2 := 0;$
  await $count_2 = 0$

  end loop

Barrier with Shared Variables

Coordinator-based

- **var** $arrive[1..n] : int := 0;$
  $continue[1..n] : bool := 0;$
- **process** $Worker[i : 1..n]$
  loop

  do $task_i$

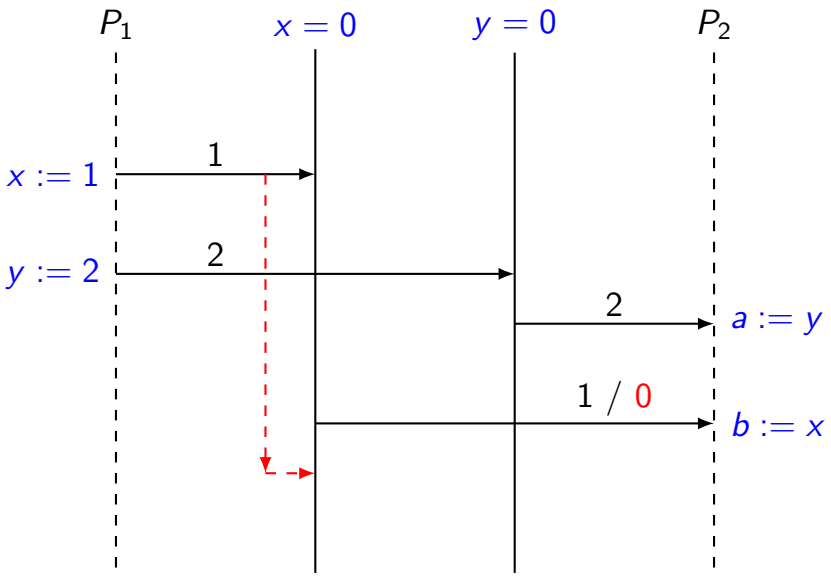
  $arrive[i] := 1;$
  await $continue[i] = 1;$
  $continue[i] := 0;$

  end loop

process *Coordinator*
loop
for $i : 1..n$ **do**
{ **await** $arrive[i] = 1;$
$arrive[i] := 0$ }
for $i : 1..n$ **do**
$continue[i] := 1;$
end loop

- General principle: Process awaiting flag resets it

Loose Memory



$a > 0 \Rightarrow b > 0$ FAILS

Caveats of Low-level Sharing

- Consider **var** $x, y : int$; $x := 1$; $y := 2$
- x and y may not be assigned as expected:
  - The *compiler* may reorder the assignments
  - The *processor* may reorder the store instructions
  - The *memory system* may reorder the write operations
  - The *compiler/processor* may keep values in registers
- Proper ordring must be ensured by *synchronization instructions*

Java low-level memory model

- Access to variables (except `long` and `double`) is *atomic*
- Joining a thread ensures that its memory operations are completed
- Monitor entry/exit synchronize memory operations
- Access to `volatile` variables is always synchronized

Linux Spin Locks

Linux 2.6.22

- ```
  static inline void __raw_spin_lock(raw_spinlock_t *lock)
  {
      asm volatile("\n1:\t"
          LOCK_PREFIX " ; decb %0\n\t"
          "jns 3f\n"
          "2:\t"
          "rep;nop\n\t"
          "cmpb $0,%0\n\t"
          "jle 2b\n\t"
          "jmp 1b\n"
          "3:\n\t"
          : "+m" (lock->slock) : : "memory");
  }
  ```

Linux Spin Locks

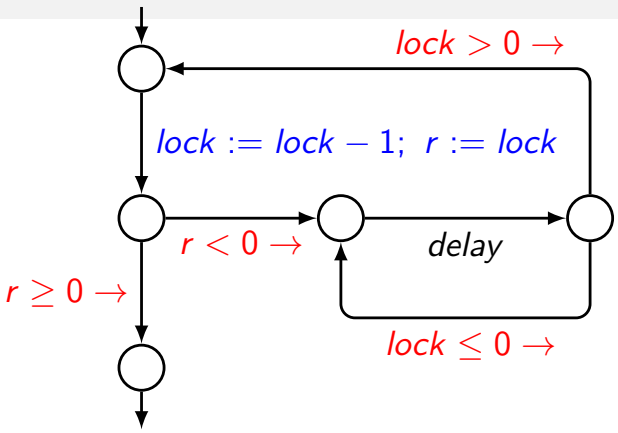
Linux 2.6.22

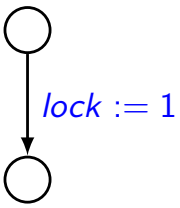
- **var** *lock* : *integer* := 1;
- *Lock*(*lock*):

  $\langle\, lock := lock - 1;\ r := lock \,\rangle$;
  while $r < 0$ **do**
  { **repeat**
  delay;
  until $lock > 0$;
  $\langle\, lock := lock - 1;\ r := lock \,\rangle$;
  }

- *Unlock*(*lock*):

  $lock := 1$;




  ```

Linux Spin Locks

Linux 2.6.22

```
/*
 * Your basic SMP spinlocks, allowing only a single CPU anywhere
 *
 * Simple spin lock operations.  There are two variants, one clears IRQ's
 * on the local processor, one does not.
 *
 * We make no fairness assumptions.  They have a cost.
 *
 * (the type definitions are in asm/spinlock_types.h)
 */
```

Critical Region with Shared Variables

Ticket Algorithm

- **var** $number, next : integer := 1;$
  $turn[i : 1..n] : integer := 0;$
  process $P[i : 1..n]$
  loop
  $\langle turn_i := number;\ \ number := number + 1 \rangle;$
  $\langle$ **await** $turn_i = next \rangle;$
  critical section$_i$;
  $\langle next := next + 1 \rangle;$
  noncritical section$_i$;
  end loop

Linux Spin Locks

Linux 3.11

- ```
  static __always_inline void __ticket_spin_lock(arch_spinlock_t *lock)
  {
      register struct __raw_tickets inc = { .tail = 1 };

      inc = xadd(&lock->tickets, inc);

      for (;;) {
          if (inc.head == inc.tail)
              break;
          cpu_relax();
          inc.head = ACCESS_ONCE(lock->tickets.head);
      }
      barrier(); /* make sure nothing creeps before the lock is taken */
  }
  ```
  ```