COMPUTER ARCHITECTURE

Synchronization

Introduction.

Hardware support.

Locks.

Barriers

Computer Architecture - 2014

□ Communication performed through shared memory.

◻ It is necessary to **synchronize** access to shared variables.

□ **Alternatives**:

◻ 1-1 Communication.

◻ Collective communication.

Computer Architecture - 2014

□ Ensure that a **read** (receive) happens after **write** (send).

□ In case of reuse (loops):
   ◘ Ensure that **write** (send) happens after to prior **read** (send).

□ **Mutual exclusion** needed:
   ◘ Only one of the processes accesses variable at the same time.

□ **Critical section**:
   ◘ Sequence of instructions accessing to one or

□ Needs coordination of multiple accesses to a variable.

  ◘ Writes without interference.

  ◘ Reads must wait for data to be available.

□ **Guarantees needed**:

  ◘ Accesses to variables in mutual exclusion.

  ◘ Result is not read until all have executed their

Adding a vector

```
for (i=iproc; i<n;i=i+nproc) {
  result = result + v[i];
}
```

```
double partial = 0;
for (i=iproc; i<n;i=i+nproc) {
  partial = partial + v[i];
}
result = result + partial;
```

Computer Architecture - 2014

□ Introduction

□ **Hardware support.**

□ Locks.

□ Barriers

Computer Architecture - 2014

□ Need to fix a global order of operations.

◻ Consistency model could be insufficient and complex.

◻ Usually complemented with read-modify-write operations.

◻ Example in IA-32:

- Instructions with LOCK prefix.

- Access to bus in exclusive mode if position is not in cache.

Test and set:

- Atomic sequence.
  - Read memory location in register (returned as result).
  - Write value 1 in memory location.

- IBM 370, Sparc V9

Computer Architecture - 2014

□ Swap:

◘ Atomic sequence:

  ■ Exchanges contents of a memory location and a register.

  ■ Includes a memory read and a memory write.

◘ More general than test-and-set.

◘ Instruction IA-32:

  ■ XCHG reg, mem

Computer Architecture - 2014

Fetch-and-op:

- Several operations: fetch-add, fetch-or, fetch-inc, …

- Atomic sequence:
  - Read memory position in register (return that value).
  - Write in memory location the result of applying operation to original value.

- Example IA-32:
  - LOCK XADD reg, mem

Computer Architecture - 2014

Compare-and-swap:

- Operation on two local variables (registers a and b) and a memory location (variable x).

- Atomic sequence:
  - Read value from x.
  - If x equals to register a → swap x and register b.

- Example IA-32:
  - LOCK CMPXCHG mem, reg
  - Uses implicitly additional register eax.

Computer Architecture - 2014

Primitives

- LL/SC (Load Linked/Store Conditional):
  - If the content of a read variable through LL is modified before a SC, store is not performed.
  - If between LL and SC a context switch happens, SC is not performed.
  - SC returns success/failure code.

  - Example Power-PC:
    - LWARX
    - STWCX

Computer Architecture - 2014

- Introduction

- Hardware support.

- **Locks.**

- Barriers

Computer Architecture - 2014

□ Mechanism to ensure mutual exclusion.

□ Two synchronization functions:

◻ **Lock(k)**

■ Acquires the lock.

■ If several try to acquire the lock, n-1 of them transition to waiting state.

■ If more processes arrive, they transition to waiting state.

◻ **Unlock(k)**

■ Release the lock.

Computer Architecture - 2014

☐ Two Alternatives.

▫ **Busy waiting**:

- Process waits in a loop that constantly queries wait control variable value.

- **Spin-lock.**

▫ **Blocking**:

- Process suspends and gives processor to another process.

- If a process executes un-lock and there are blocked processes, one of them is released.

- Requires scheduler support.

Computer Architecture - 2014

Acquisition method:

- Used to try to lock acquisition.

Waiting method:

- Mechanism to wait until lock can be acquired.

Release mechanism:

- Mechanisms to release one or more waiting processes.

□ Shared variable **k** with two values:

◘ 0 → open.

◘ 1 → closed.

□ **Lock(k)**

◘ If k=1 → Busy wait while k=1

◘ If k=0 → k=1

◘ Do not allow 2 processes to acquire lock simultaneously.

Computer Architecture - 2014

Test and set

```
Lock(k) {
  while (k.test_and_set()) {}
}
```

Fetch and op

```
Lock(k) {
  while (k.fetch_and_or(1) == 1) {}
}
```

Swap IA-32

```
Lock:     MOV     eax, 1
Repetir:  XCHG    eax, k
          CMP     eax, 1
          jz      Repetir
```

Computer Architecture - 2014

Goal: Minimize memory writes

Test and set

```
Lock(k) {
  while (k.test_and_set()) {
    while (k==1) {}
  }
}
```

If it is very likely that lock is open

Test and set

```
Lock(k) {
  do {
    while (k==1) {}
  } while (k.test_and_set() );
}
```

If it is very likely that lock is closed

Computer Architecture - 2014

Goal:

- Memory access reduction.
- Limit power comsumption.

```
Lock(k) {
  while (k.test_and_set()) {
    pause(delay);
    delay *=2;
  }
}
```

□ Performance can be improved if same variable used to synchronized and communicate.

▣ Avoid using shared variables only for synchronization.

```
double partial = 0;
for (i=iproc; i<n;i=i+nproc) {
  partial = partial + v[i];
}
result.fech_add(partial);
```

Computer Architecture - 2014

☐ **Problem**:

  ◘ Simple implementations do not fix acquisition order of a lock.

  ◘ Starvation could be possible.

☐ **Solution**:

  ◘ Make that lock is acquired by request age (oldes acquires first).

  ◘ Guarantees a FIFO ordering.

Cartagena99

Computer Architecture - 2014

- **Two counters**:
  - **Acquisition counter**: Number of processes that requested the lock.
  - **Release counter**: Number of times that a lock has been released.

- **Lock**
  - Tag → Acquisition counter value.
  - Increment acquisition counter.
  - Process stays waiting until release counter equals to tag.

- **Unlock**

- Keep a queue with processes waiting to enter into a critical section.

Lock

- Check if queue is empty.
- If a process joins a queue make busy waiting in a variable.
  - Each process busy waits in a different variable.

Unlock

- Remove process from queue

Computer Architecture - 2014

- □ Introduction

- □ Hardware support.

- □ Locks.

- □ Barriers

Computer Architecture - 2014

□ Allow to synchronize several processes in some point.

▫ Guarantees that no process passes the barrier until all of them have arrived.

▫ Used to synchronized program phases.

Computer Architecture - 2014

□ Centralized counter associated to the barrier.

  ◘ Counts the number of processes that have arrived the barrier.

□ Barrier function:

  ◘ Increment counter.

  ◘ Wait until counter reaches the number of processes to be synchronized.

Cartagena99

Computer Architecture - 2014

```
Barrier(barrier, n) {
  lock(barrier.lock);
  if (barrier.counter == 0) {
    barrier.flag=0;
  }
  local_counter = barrier.counter++;
  unlock(barrier.lock);
  if (local_counter == NP) {
    barrier.counter=0;
    barrier.flag=1;
  }
  else {
    while (barrier.flag==0) {}
  }
}
```

Problem if barrier reused in loop.

Computer Architecture - 2014

```
Barrier(barrier, n) {
  local_flag = !local_flag;
  lock(barrier.lock);
  local_counter = barrier.counter++;
  unlock(barrier.lock);
  if (local_counter == NP) {
    barrier.counter=0;
    barrier.flag=local_flag;
  }
  else {
    while (barrier.flag==local_flag) {}
  }
}
```
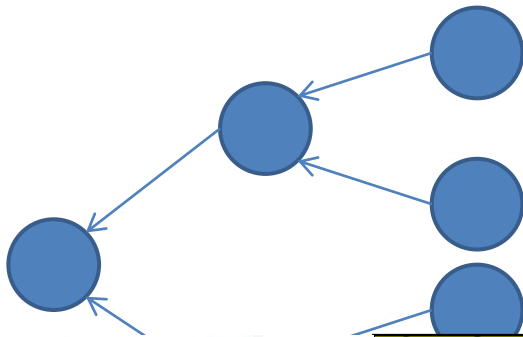
Computer Architecture - 2014

□ A simple implementation of barriers is not scalable.

  ◘ Contention in access to shared variables.

□ Tree structure for arrival and release processes.

  ◘ Specially used in distributed networks.

- Synchronization necessary for access to shared variables.
  - Alternatives for 1-1 and collective communication.
- Hardware support needed to fix global order of operations.
  - Variety of approaches in different processor families.
- Locks as a higher level synchronization mechanism.
  - Waiting mechanisms: busy waiting and blocking.
  - Mechanisms for acquisition, waiting, and release.
- Program phases can be synchronized with

Computer Architecture - 2014