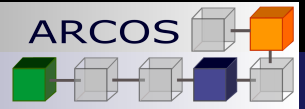
Synchronization

Computer Architecture

J. Daniel García Sánchez (coordinator)
David Expósito Singh
Javier García Blas

ARCOS Group
Computer Science and Engineering Department
University Carlos III of Madrid



1 Introduction

2 Hardware primitives

3 Locks

4 Barriers

5 Conclusion

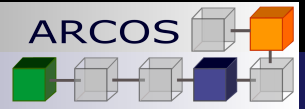
Synchronization in shared memory

- Communication performed through shared memory.
  - It is necessary to synchronize multiple accesses to shared variables.

- **Alternatives**:
  - Communication 1-1.
  - Collective communication (1-N).

Communication 1 to 1

- Ensure that reading (**receive**) is performed after writing (**send**).
- In case of **reuse** (loops):
  - Ensure that writing (**send**) is performed after former reading (**receive**).
- Need to access with **mutual exclusion**.
  - Only one of the processes accesses a variable at the same time.
- **Critical section**:
  - Sequence of instructions accessing one or more variables with mutual exclusion.



Collective communication

- Needs **coordination** of **multiple accesses** to variables.
  - Writes without interferences.
  - Reads **must wait** for data to be available.
- Must **guarantee accesses** to variable in **mutual exclusion**.
- Must **guarantee** that **result is not read** until all processes/threads have executed their critical section.

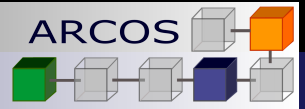
Adding a vector

Critical section in loop

```
void f(int max) {
  vector<double> v = get_vector(max);
  double sum = 0;

  auto do_sum = [&](int start, int n) {
    for (int i=start; i<n; ++i) {
      sum += v[i];
    }
  }

  thread t1{do_sum,0,max/2};
  thread t2{do_sum,max/2+1,max};
  t1.join();
  t2.join();
}
```

Critical section out of loop

```
void f(int max) {
  vector<double> v = get_vector(max);
  double sum = 0;

  auto do_sum = [&](int start, int n) {
    double local_sum = 0;
    for (int i=start; i<n; ++i) {
      local_sum += v[i];
    }
    sum += local_sum;
  }

  thread t1{do_sum,0,max/2};
  thread t2{do_sum,max/2+1,max};
  t1.join();
  t2.join();
}
```



1 Introduction

2 Hardware primitives

3 Locks

4 Barriers

5 Conclusion

Hardware support

- Need to fix a global order in operations.
  - Consistency model can be **insufficient and complex**.
  - Usually **complemented** with **read-modify-write** operations.
  - **Example in IA-32**:
    - Instructions with prefix **LOCK**.
    - Access to bus in **exclusive mode** if location **is not in cache**.

Primitives: Test and set

- Instruction **Test and Set**:
  - **Atomic sequence**:

    1. **Read memory location** into register (will be returned as result).
    2. **Write value 1** in memory location.

- **Uses**: IBM 370, Sparc V9

Primitives: Exchange

- Instruction for **exchange** (swap):
  - **Atomic sequence**:
    1. **Exchanges** contents in a **memory location** and a **register**.
    2. **Includes** a **memory read** and a **memory write**.
  - **More general that test-and-set**.
- **Instruction IA-32**:
  - **XCHG reg**, **mem**
- **Uses**: Sparc V9, IA-32, Itanium

Primitives: Fetch and operation

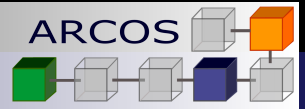
- Instruction for **fetching and applying operation** (fetch-and-op):
  - Several operations: **fetch-add**, **fetch-or**, **fetch-inc**, ...
  - **Atomic sequence**:
    1. Read memory location into a register (return that value).
    2. Write to memory location the result of applying an operation to the original value.
  - **Instruction IA-32**:
    - **LOCK XADD reg**, **mem**
  - **Uses**: IBM SP3, Origin 2000, IA-32, Itanium.

Primitives: Compare and exchange

- Instruction to **compare and exchange** (compare-and-swap o compare-and-exchange):
  - Operation on **two local variables** (registers **a** and **b**) and a **memory location** (variable **x**).
  - **Atomic sequence**:
    1. Read value from **x**.
    2. If **x** is equal to register **a** $\rightarrow$ exchange **x** and register **b**.
  - **Instruction IA-32**:
    - **LOCK CMPXCHG mem**, **reg**
    - Implicitly uses additional register **eax**.
  - **Uses**: IBM 370, Sparc V9, IA-32, Itanium.

Primitives: Conditional store

- Pair of instructions **LL/SC** (Load Linked/Store Conditional).
  - **Operation**:
    - If content of **read variable** through **LL** is **modified** before a **SC** storage **is not performed**.
    - When a **context switch** happens between **LL** and **SC**, **SC is not performed**.
    - **SC** returns a **success/failure code**.
  - **Example in Power-PC**:
    - **LWARX**
    - **STWCX**
  - **Uses**: Origin 2000, Sparc V9, Power PC



1 Introduction

2 Hardware primitives

3 Locks

4 Barriers

5 Conclusion

Locks

- A **lock** is a mechanism to ensure **mutual exclusion**.
- **Two synchronization functions**:
  - **Lock(k)**:
    - Acquires the lock.
    - If several processes try to acquire the lock, n-1 are kept waiting.
    - If more processes arrive, they are kept to waiting.
  - **Unlock(k)**:
    - Releases the lock.
    - Allow that a waiting process acquires the lock.

Waiting mechanisms

- **Two alternatives**: **busy waiting** and **blocking**.
- **Busy waiting**:
  - Process waits in a **loop** that **constantly** queries the wait control variable value.
  - **Spin-lock**.
- **Blocking**:
  - Process remains suspended and yields processor to other process.
  - If a process executes **unlock** and there are **blocked** processes, one of them is un-blocked.
  - Requires support from a scheduler (usually OS or *runtime*).
- **Alternative selection depends on cost**.

Components

- Three **elements of design** in a locking mechanism: **acquisition**, **waiting** y **release**.
- **Acquisition method**:
  - Used to try to acquire the lock.
- **Waiting method**:
  - Mechanism to wait until lock can be acquired.
- **Release method**:
  - Mechanism to release one or several waiting processes.

Simple locks

- **Shared** variable **k** with two values.
  - **0** → **open**.
  - **1** → **closed**.

- **Lock(k)**
  - If **k=1** → **Busy waiting** while **k=1**.
  - If **k=0** → **k=1**.
  - **Do not allow** that 2 processes **acquire a lock simultaneously**.
    - Use read-modify-write to close it.

Simple implementations

Test and set

```
void lock(atomic_flag & k) {
  while (k.test_and_set())
    {}
}

void unlock(atomic_flag & k) {
  k.clear();
}
```

Fetch and operate

```
void lock(atomic<int> & k) {
  while (k.fetch_or(1) == 1)
    {}
}

void unlock(atomic<int> & k) {
  k.store(0);
}
```

Simple implementations

Exchange IA-32

```
do_lock:    mov eax, 1
repeat:     xchg eax, _k
            cmp eax, 1
            jz  repeat
```

Exponential delay

- **Goal**:
  - **Reduce** number of memory accesses.
  - **Limit** energy consumption.

Lock with exponential delay

```
void lock(atomic_flag & k) {
  while (k.test_and_set())
  {
    perform_pause(delay);
    delay *= 2;
  }
}
```

- Time between invocations to **test_and_set()** is incremented **exponentially**

Synchronization and modification

- **Performance can be improved** if using the **same variable to synchronize and communicate**.
  - Avoid using **shared variables** only to synchronize.

Add a vector

```
double partial = 0;
for (int i=iproc; i<max; i+=nproc) {
   partial += v[i];
}
sum.fetch_add(partial);
```

Locks and arrival order

- **Problem**:
  - Simple implementations do not fix a lock acquisition order.
  - Starvation might happen.

- **Solution**:
  - Make the lock is acquired by request **age**.
  - Guarantees FIFO order.

Tagged locks

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

Lock:

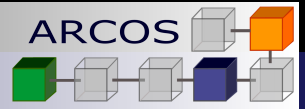
- **Tag** $\rightarrow$ Acquisition counter value.
- Acquisition counter is incremented.
- Process remains waiting until the release counter matches the tag.

Unlock:

- Increments release counter.

Queue based locks

- Keep a **queue** with **processes waiting** to enter into a **critical section**.
- **Lock**:
  - Check if queue is empty.
  - If a process joins the queue it performs busy waiting in a variable.
    - Each process performs busy waiting in a different variable.
- **Unlock**:
  - Removes process from queue.
  - Modifies process waiting control variable.



1 Introduction

2 Hardware primitives

3 Locks

4 Barriers

5 Conclusion

Barrera

- A barrier allows to **synchronize** several processes in some point.
  - Guarantees that no process passes the barrier until all have arrived.
  - Used to synchronize phases in a program.

Centralized barriers

- Centralized **counter** associated to the **barrier**.
  - Counts number of processes that have arrived the barrier.

- **Barrier function**:
  - Increments counter
  - Waits the counter to reach the number of processes to be synchronized.

Simple barrier

Simple implementation

```
do_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 is reused in a loop.

Barrier with way inversion

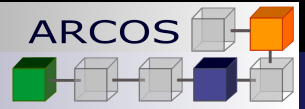
Simple implementation

```
do_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) {}
  }
}
```

Tree barriers

- A simple implementation of barriers is not **scalable**.
  - Contention in access to shared variables.

- Tree structure for process arrival and release.
  - Specially useful in distributed networks.



1 Introduction

2 Hardware primitives

3 Locks

4 Barriers

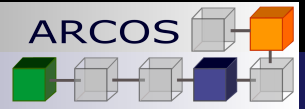
5 Conclusion

Summary

- Need for shared memory access synchronization:
  - Individual (1-1) and collective (1-N) communication.
- Diversity of hardware primitives for synchronization.
- Locks as a mechanism for mutual exclusion.
  - Busy waiting versus blocking.
  - Three design elements: acquisition, waiting, and release.
- Locks may lead to problems if order is not fixed (starvation).
  - Solutions based in tags or queues.
- Barriers offer mechanisms to structure programs in phases.

References

- **Computer Architecture. A Quantitative Approach.**
  5th Ed.
  Hennessy and Patterson.
  Section: 5.5



Synchronization

Computer Architecture

J. Daniel García Sánchez (coordinator)
David Expósito Singh
Javier García Blas

ARCOS Group
Computer Science and Engineering Department
University Carlos III of Madrid