Shared Memory Programming

Frédéric Haziza <daz@it.uu.se>

Department of Computer Systems
Uppsala University

Summer 2010
**Shared Resource**

**Remark**
Sequential program use **shared variables**
Convenience for Global data structure

**Necessity**
Concurrent program **MUST** use shared components
The only way to solve a problem: Communicate
The only way to communicate:
One writes into **something** which the other one reads.

**Communication**
**Synchronisation**

Communication ⇒

- Mutual Exclusion
- Condition synchronisation

**Example (Mutual Exclusion)**
Whenever 2 processes need to take runs accessing shared objects

**Example (Condition synchronisation)**
Whenever one process needs to wait for another
Cache coherency

A: Thread
   Read A
   Read A
   ...  
   Read A

B: Thread
   Read A
   ...  
   Read A

Write A
State

All values of the program variables at a point in time

Explicit and implicit variables

Atomic actions

Examine or change the program state

Concurrent program execution

A particular interleaving of atomic actions

Trace / History

\[ s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots \rightarrow s_n \]
State Explosion

For each execution, a history

Number of history: ENORMOUS!!

Synchronization
Program Verification

Does my program satisfy a given property?

Run the program and see what happens

Exhaustive case analysis - Consider ALL interleavings of atomic actions

\[ n \text{ processes with } m \text{ atomic actions. Number of histories } = \frac{(n \cdot m)!}{(m!)^n} \]

Example: 3 processes with 2 atomic actions. Number of histories = 90

Abstract analysis - Model the states with predicates. Compact representation of states
Finding pattern in a file

1: string line;
2: Read a line of input from stdin into line
3: while (!EOF) {
   \( \triangleright EOF = end \ of \ file \)
4:   look for pattern in line
5:   if pattern is in line {
6:     print line;
7:   }
8:   read next line of input;
9: }
Independant parts

- Part A: Reads
- Part B: Writes
- Intersection: Both Reads and Writes
string line;
Read a line of input from stdin into line
while ( !EOF ) {
    $EOF = end of file$
    look for pattern in line
    if pattern is in line {
        print  line;
    }
    read next line of input;
}
Finding pattern in a file - Parallel

string line1, line2;
Read a line of input from stdin into line1
while ( !EOF ) {
    EOF = end of file
    // Process creation overhead. And dominant!
    look for pattern in line1
    if pattern is in line1 {
        print line1;
    }
    read next line of input into line2;
}

line1 = line2; // pure overhead! Not in the sequential program
Finding pattern in a file
Better parallel solution

```c
string buffer;  // contains one line of input
bool done = false;  // to signal termination

// process 1 finds patterns
string line1;
while ( true ) {
    wait for buffer to be full or done to be true;
    if (done) break;
    line1 = buffer;
    signal that buffer is empty
    look for pattern in line1;
    if pattern is in line1 {
        print line1;
    }
}

// process 2 reads new lines
string line2;
while ( true ) {
    read next line of input into line2;
    if (EOF) done = true; break;
    wait for buffer to be empty;
    buffer = line2;
    signal that buffer is full;
}
```

buffer is the shared variable: the
line1 and line2 are local copies
On the way to atomicity
A Bank account example

Balance \( b \) with initially 100 sek.
Person \( A \) wants to withdraw 70 sek, if possible.
Person \( B \) wants to withdraw 50 sek, if possible.
\( \Rightarrow \) Balance should not be negative.

\[
\text{int } b = 100; \quad \triangleright \text{initially 100 sek on the bank account}
\]

\( \triangleright \text{Person } A \text{ tries to withdraw 70 sek} \)
\[
\text{if } (b - 70 > 0) \{ \\
\quad b = b - 70;
\}
\]

\( \triangleright \text{Person } B \text{ tries to withdraw 50 sek} \)
\[
\text{if } (b - 50 > 0) \{ \\
\quad b = b - 50;
\}
\]

Can anything go wrong? \( b < 0 \) ??
**Atomicity**

Another bank account example

Balance $b = 0$ sek.
Person A does a deposit of 100 sek.
Person B does a deposit of 200 sek
⇒ Balance should be 300 sek.

<table>
<thead>
<tr>
<th>$A$</th>
<th>Balance $b$</th>
<th>$B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>load $R_2, b$</td>
<td>0</td>
<td>load $R_4, b$</td>
</tr>
<tr>
<td>add $R_2, #100$</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>store $b, R_2$</td>
<td>100</td>
<td>add $R_4, #200$</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>store $b, R_4$</td>
</tr>
</tbody>
</table>

Solution: Synchronisation (Locks, Semaphores, Monitors, Retry loops,...)
The programmer’s nightmare begins

The programmer must implement correct synchronization!

### Example (lock S and Q)

<table>
<thead>
<tr>
<th>$P_1$</th>
<th>$P_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock(S)</td>
<td>lock(Q)</td>
</tr>
<tr>
<td>lock(Q)</td>
<td>lock(S)</td>
</tr>
<tr>
<td>...</td>
<td>....</td>
</tr>
<tr>
<td>unlock(Q)</td>
<td>unlock(S)</td>
</tr>
<tr>
<td>unlock(S)</td>
<td>unlock(Q)</td>
</tr>
</tbody>
</table>

Leads to deadlock: Both $P_1$ and $P_2$ are waiting for each other.
Additionally, bad implementation can lead to starvation.
Atomicity

Another example

sum := 0

while (sum < N)
    sum := sum + 1

print(sum)

How many additions?  What is printed?

Anything between N and N + 3
Anything between N and N * 4

21 OS2'10 | Shared Memory Programming
Race Condition

Definition
A situation in a shared-variable concurrent program in which one process writes a variable that a second process reads, but the first process continues execution – namely races ahead – and changes the variable again before the second process sees the result of the first change. This usually lead to an incorrectly synchronized program.

Definition (alternative)
The possibility of incorrect results in the presence on unlucky timing in concurrent programs – getting the right answer relies on lucky timing.
Every day life race condition

Meeting at Wayne’s coffee at 16.00, downtown

Wayne’s A

16.00

Wayne’s B

16.10 → Where is he?

16.15
Check-then-act

Example (Lazy initialization: Not safe)

```java
public class LazyInitRace {

    private ExpensiveObject instance = null;

    public ExpensiveObject getInstance() {
        if (instance == null) {
            instance = new ExpensiveObject();
        }
        return instance;
    }
}
```
Read-Modify-Write

Example (We’ve seen that before!)

```java
count++;```

OS2’10 | Shared Memory Programming
Compounds actions

Example (Caching Factorizer: Not safe)

```java
public class UnsafeCachingFactorizer implements Servlet {
    private final AtomicReference<BigInteger> lastNumber = new AtomicReference<BigInteger>();
    private final AtomicReference<BigInteger[]> lastFactors = new AtomicReference<BigInteger[]>();

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = getNumberFromRequest(req);
        if (i.equals(lastNumber.get()))
            encodeIntoResponse(resp, lastFactors.get());
        else {
            BigInteger[] factors = factor(i);
            lastNumber.set(i);
            lastFactors.set(factors);
            encodeIntoResponse(resp, factors);
        }
    }
}
```
Out-of-thin-air safety

Example

Non-atomic 64-bit operations

Hardware may read in 2 steps, and use a temporary value in between.

*Better than random.*
What's next?

- Locks/Barriers
- Semaphores
- Monitors
- Implementation