Shared Memory Programming

Frédéric Haziza <daz@it.uu.se>

Department of Computer Systems
Uppsala University

Summer 2009
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 $\Rightarrow$

Synchronisation
- 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
Shared-Memory Programming

- SIMD
- MIMD
- Message Passing
- Shared Memory
- Fine-grained
- Coarse-grained
- UMA
- NUMA
- COMA
Cache coherency

Shared Memory

A:

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

B:

Thread
Read B
...
...
Read A

Write A

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

Thread
Read A

OS2’09 | Shared Memory Programming
State

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 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!!
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

\( n \) processes with \( m \) 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) {
5:     look for pattern in line
4:       if pattern is in line {
6:         print line;
7:     }
6:   }
7:   read next line of input;
Independant parts
string line;
Read a line of input from stdin into line
\textbf{while} (!EOF) { 
\footnotesize{$\triangleright EOF =$ end of file}

look for pattern in line
\textbf{if} pattern is in line {
\textbf{print} line;
}
}

read next line of input;
string line1, line2;
Read a line of input from stdin into line1

while (!EOF) {
  
 _padding
  EOF = end of file

  
 _padding
  Process creation overhead. And dominant!

    look for pattern in line1
    if pattern is in line1 {
      print line1;
    }

    line1 = line2;  
    pure overhead! Not in the sequential program
}

read next line of input into line2;
Finding pattern in a file
Better parallel solution

```
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.

```java
int b = 100;  // initially 100 sek on the bank account
```

$\Rightarrow$ Person $A$ tries to withdraw 70 sek
```java
if \( (b - 70 > 0) \) {
    b = b - 70;
}
```

$\Rightarrow$ Person $B$ tries to withdraw 50 sek
```java
if \( (b - 50 > 0) \) {
    b = b - 50;
}
```

Can anything go wrong? $b < 0$ ??
Atomicity
Another bank account example

Balance \( b = 0 \text{ sek.} \)
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>100</td>
<td>add ( R_4, #200 )</td>
</tr>
<tr>
<td>store ( b, R_2 )</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
print(sum)
while (sum < N)
    sum:=sum+1
```

How many additions?

Anything between N and N+3

Anything between N and N*4

What is printed?

21
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.
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!)

```
count++;  
```
Compound 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