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 ⇒

Synchronisation

- Mutual Exclusion
- Condition synchronisation

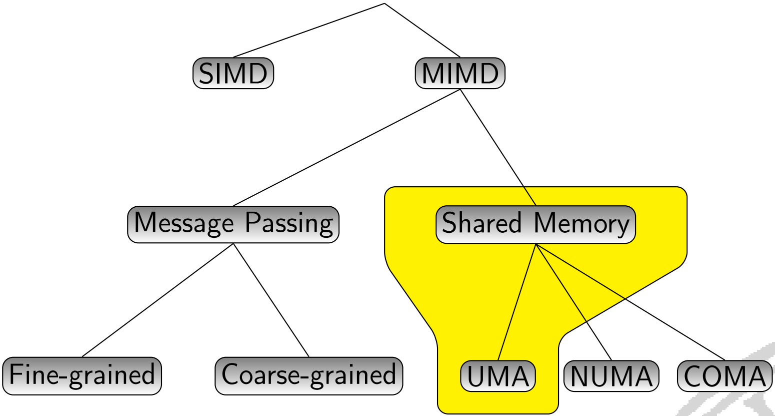
Example (Mutual Exclusion)

Whenever 2 processes need to take runs accessing shared objects

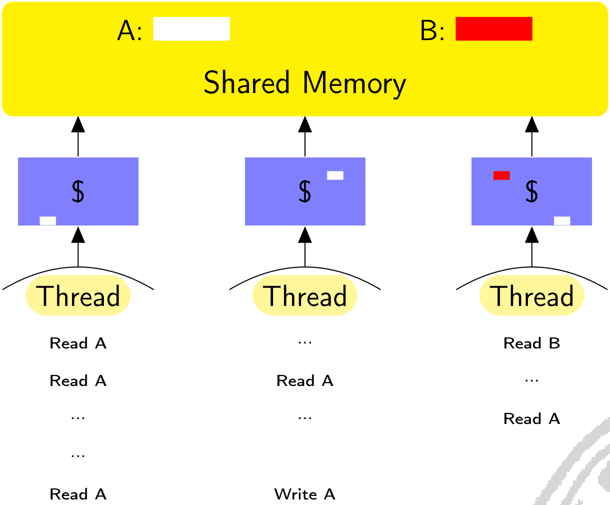
Example (Condition synchronisation)

Whenever one process needs to wait for another

Shared-Memory Programming



Cache coherency



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

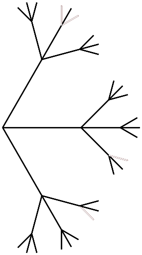
Trace / History

$s_0 \longrightarrow s_1 \longrightarrow s_2 \longrightarrow \ldots \longrightarrow s_n$

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 processes with m atomic actions. Number of histories $= \frac{(n.m)!}{(m!)^n}$

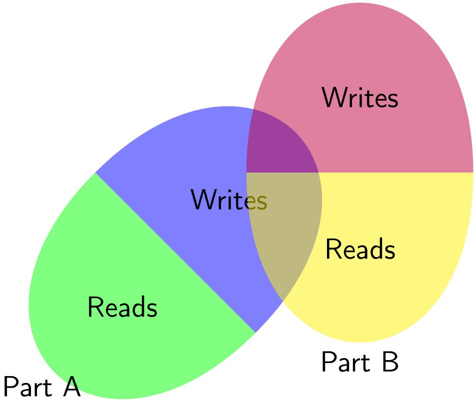
Example: 3 processes with 2 atomic actions. Number of histories = 90

Abstract analysis - Model the states with predicates. Compact representation of states

Finding pattern in a file

```
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;
}
```

Independant parts



Finding pattern in a file - Parallel

```
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 {              read next line of input;
    print line;
}

}
```

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

```
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 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.

```
int b = 100;
```
▷*initially 100 sek on the bank account*

▷*Person A tries to withdraw 70 sek*

if $(b - 70 > 0)$ {
  $b = b - 70;$
}

▷*Person B tries to withdraw 50 sek*

if $(b - 50 > 0)$ {
  $b = b - 50;$
}

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

$\Rightarrow$ Balance should be 300 sek.

A	Balance b	B
	0	load R_4, b
load R_2, b	0	
add $R_2, \#100$		
store b, R_2	100	
		add $R_4, \#200$
	200	store b, R_4

Solution: Synchronisation (Locks,Semaphores,Monitors,Retry loops,...)

The programmer's nightmare begins

The programmer must implement correct synchronization!

Example (lock S and Q)

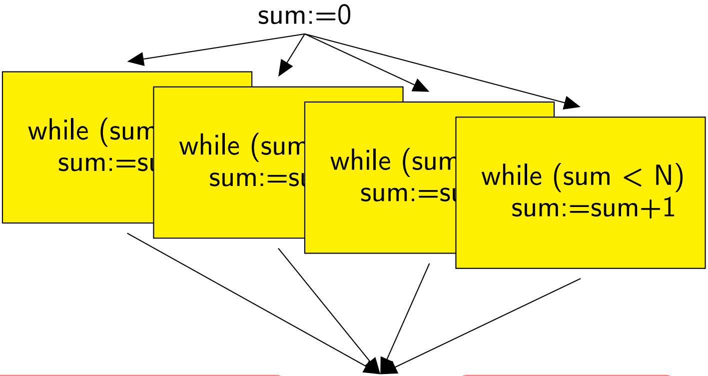
P_1	P_2
lock(S)	lock(Q)
lock(Q)	lock(S)
...	....
unlock(Q)	unlock(S)
unlock(S)	unlock(Q)

Leads to deadlock: Both P_1 and P_2 are waiting for each other

Additionally, bad implementation can lead to starvation.

Atomicity

Another example



How many additions?

print(sum)

What is printed?

Race Condition

Definition

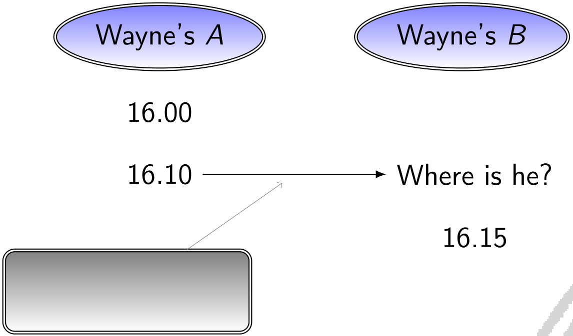
A situation in a shared-variable concurrent program in which one process writes a variable that a second process reads, but the first process continues execution – namely races ahead – and changes the variable again before the second process sees the result of the first change. This usually lead to an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



Check-then-act

Example (Lazy initialization: Not safe)

```
public class LazyInitRace {

    private ExpensiveObject instance = null;

    public ExpensiveObjevt 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)

```
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?