Concurrency 1

Shared Memory

Jean-Jacques Lévy

jeanjacqueslevy.net/dea
Why concurrency?

1. Programs for multi-processors
2. Drivers for slow devices
3. Human users are concurrent
4. Distributed systems with multiple clients
5. Reduce latency
6. Increase efficiency, but Amdahl's law

\[ S = \frac{N}{b \times N + (1 - b)} \]

\( (S = \text{speedup}, \ b = \text{sequential part}, \ N = \text{processors}) \)
Implicit Communication

Suppose $x$ is a global variable. At beginning, $x = 0$

Consider

$S = [x := x + 1; x := x + 1 || x := 2 \times x]$

$T = [x := x + 1; x := x + 1 || \text{wait} \ (x = 1); x := 2 \times x]$

After $S$, then $x \in \{2, 3, 4\}$

After $T$, then $x \in \{3, 4\}$

$T$ may be blocked

Conclusion

In $S$ and $T$, interaction via $x$ (shared memory)
Atomicity

Suppose $x$ is a global variable. At beginning, $x = 0$

Consider

$S = [x := x + 1 || x := x + 1]$

After $S$, then $x = 2$.

However if

$[x := x + 1]$ compiled into $[A := x + 1; x := A]$

Then

$S = [A := x + 1; x := A] || [B := x + 1; x := B]$

After $S$, then $x \in \{1, 2\}$.

Conclusion

1. $[x := x + 1]$ was firstly considered atomic
2. Atomicity is important
Critical section – Mutual exclusion

Let $P_0 = [\cdots; C_0; \cdots]$ and $P_1 = [\cdots; C_1; \cdots]$.

$C_0$ and $C_1$ are critical sections (i.e., should not be executed simultaneously).

**Solution 1** At beginning, $\text{turn} = 0$.

\[
\begin{align*}
P_0 & : \cdots \\
& \text{while } \text{turn} \neq 0 \text{ do} \\
& \quad \text{; } \\
& \quad C_0; \\
& \quad \text{turn} := 1; \\
& \quad \cdots
\end{align*}
\]

\[
\begin{align*}
P_1 & : \cdots \\
& \text{while } \text{turn} \neq 1 \text{ do} \\
& \quad \text{; } \\
& \quad C_1; \\
& \quad \text{turn} := 0; \\
& \quad \cdots
\end{align*}
\]

$P_0$ privileged, unfair.
Critical section – Mutual exclusion

**Solution 2** At beginning, \( a_0 = a_1 = \text{false} \).

\[
\begin{align*}
P_0 : & \quad \cdots \\
& \text{while } a_1 \text{ do } \quad P_1 : \quad \cdots \\
& \quad ; \\
& a_0 := \text{true}; \\
& C_0; \\
& a_0 := \text{false}; \\
& \cdots \\
\end{align*}
\]

False.

**Solution 3** At beginning, \( a_0 = a_1 = \text{false} \).

\[
\begin{align*}
P_0 : & \quad \cdots \\
& a_0 := \text{true}; \\
& \text{while } a_1 \text{ do } \\
& \quad ; \\
& C_0; \\
& a_0 := \text{false}; \\
& \cdots \\
\end{align*}
\]

\[
\begin{align*}
P_1 : & \quad \cdots \\
& a_1 := \text{true}; \\
& \text{while } a_0 \text{ do } \\
& \quad ; \\
& C_1; \\
& a_1 := \text{false}; \\
& \cdots \\
\end{align*}
\]

Deadlock. Both \( P_0 \) and \( P_1 \) blocked.
Dekker’s Algorithm (CACM 1965)

At beginning, $a_0 = a_1 = \text{false}$, $\text{turn} \in \{0, 1\}$

$P_0 : \ldots$

\begin{align*}
a_0 &:= \text{true}; \\
\text{while } a_1 \text{ do} \\
&\quad \text{if } (\text{turn} \neq 0) \{ \\
&\quad\quad a_0 := \text{false}; \\
&\quad\quad \text{while } (\text{turn} \neq 0) \\
&\quad\quad\quad ; \\
&\quad\quad a_0 := \text{true}; \\
&\quad\} \\
C_0; \\
\text{turn} := 1; a_0 := \text{false}; \\
\ldots
\end{align*}

$P_1 : \ldots$

\begin{align*}
a_1 &:= \text{true}; \\
\text{while } a_0 \text{ do} \\
&\quad \text{if } (\text{turn} \neq 1) \{ \\
&\quad\quad a_1 := \text{false}; \\
&\quad\quad \text{while } (\text{turn} \neq 1) \\
&\quad\quad\quad ; \\
&\quad\quad a_1 := \text{true}; \\
&\quad\} \\
C_1; \\
\text{turn} := 0; a_1 := \text{false}; \\
\ldots
\end{align*}
Peterson’s Algorithm (IPL June 81)

At beginning, \( a_0 = a_1 = \text{false} \), \( \text{turn} \in \{0, 1\} \)

\[
\begin{align*}
P_0 & : \cdots \\
a_0 & := \text{true}; \\
\text{turn} & := 1;  \\
\text{while } a_1 \land \text{turn} \neq 0 \text{ do}  \\
\hspace{1cm} & ;  \\
C_0;  \\
a_0 & := \text{false};  \\
\cdots & 
\end{align*}
\]

\[
\begin{align*}
P_1 & : \cdots \\
a_1 & := \text{true}; \\
\text{turn} & := 0;  \\
\text{while } a_0 \land \text{turn} \neq 1 \text{ do}  \\
\hspace{1cm} & ;  \\
C_1;  \\
a_0 & := \text{false};  \\
\cdots & 
\end{align*}
\]
Synchronization

Concurrent/Distributed algorithms

1. Lamport: barber, baker, 
2. Dekker's algorithm for \( P_0, P_1, P_N \) (Dijkstra 1968)
3. Peterson is simpler and can be generalised to \( N \) processes
4. Proofs? By model checking? With assertions? In temporal logic (eg Lamport's TLA)?
5. Dekker's algorithm is too complex
6. Dekker's algorithm uses busy waiting
7. Fairness achieved because of fair scheduling

Need for higher constructs in concurrent programming.

**Exercice 1** Try to define fairness.
A generalised semaphore $s$ is integer variable with 2 operations

\begin{align*}
wait(s) : & \text{ If } s > 0 \text{ then } s := s - 1 \\
& \text{Otherwise be suspended on } s.
\end{align*}

\begin{align*}
signal(s) : & \text{ If some process is suspended on } s, \text{ wake it up } \\
& \text{Otherwise } s := s + 1.
\end{align*}

Now mutual exclusion is easy :

At beginning, $s = 1$.

Then $P_1 \parallel P_2$ where

\begin{align*}
P_1 = [\cdots; & \text{wait}(s); \ A; \ signal(s); \cdots] \\
P_2 = [\cdots; & \text{wait}(s); \ B; \ signal(s); \cdots]
\end{align*}
Operational (micro-)semantics (sequential part)

Language

\[ P, Q ::= \text{skip} \mid x := e \mid \text{if } b \text{ then } P \text{ else } Q \mid P; Q \mid \text{while } b \text{ do } P \]

\[ e ::= \text{expression} \]

Semantics (SOS)

\[ \langle \text{skip} , \sigma \rangle \rightarrow \langle \bullet, \sigma \rangle \quad \langle x := e, \sigma \rangle \rightarrow \langle \bullet, \sigma[\sigma(e)/x] \rangle \]

\[ \sigma(e) = \text{true} \]

\[ \frac{\langle \text{if } e \text{ then } P \text{ else } Q, \sigma \rangle \rightarrow \langle P, \sigma \rangle \quad \langle \text{if } e \text{ then } P \text{ else } Q, \sigma \rangle \rightarrow \langle Q, \sigma \rangle}{\sigma(e) = \text{false}} \]

\[ \frac{\langle P, \sigma \rangle \rightarrow \langle P', \sigma' \rangle \quad \langle P; Q, \sigma \rangle \rightarrow \langle P'; Q, \sigma' \rangle \quad (P' \neq \bullet)}{\sigma(e) = \text{true}} \]

\[ \langle \text{while } e \text{ do } P, \sigma \rangle \rightarrow \langle P; \text{while } e \text{ do } P, \sigma \rangle \quad \langle \text{while } e \text{ do } P, \sigma \rangle \rightarrow \langle \bullet, \sigma \rangle \]

\[ \sigma \in \text{Variables} \mapsto \text{Values} \quad \sigma[v/x](x) = v \quad \sigma[v/x](y) = \sigma(y) \text{ if } y \neq x \]
Operational semantics (parallel part)

Language

\[ P, Q ::= \ldots \mid P \parallel Q \mid \text{wait } b \mid \text{await } b \text{ do } P \]

Semantics (SOS)

\[
\frac{\langle P, \sigma \rangle \rightarrow \langle P', \sigma' \rangle}{\langle P \parallel Q, \sigma \rangle \rightarrow \langle P' \parallel Q, \sigma' \rangle}
\]

\[
\frac{\sigma(b) = \text{true}}{\langle \text{wait } b, \sigma \rangle \rightarrow \langle \bullet, \sigma \rangle}
\]

\[
\frac{\langle \text{await } b \text{ do } P, \sigma \rangle \rightarrow \langle P', \sigma' \rangle}{\langle \text{await } b \text{ do } P, \sigma \rangle \rightarrow \langle \bullet, \sigma' \rangle}
\]

Exercice 2  Complete SOS for \( e \) and \( v \)

Exercice 3  Find SOS for boolean semaphores.

Exercice 4  Avoid spurious silent steps in \( \text{if}, \text{while} \) and \( \parallel \).
SOS reductions

Notations

\[
\langle P_0, \sigma_0 \rangle \rightarrow \langle P_1, \sigma_1 \rangle \rightarrow \langle P_2, \sigma_2 \rangle \rightarrow \cdots \langle P_n, \sigma_n \rangle \rightarrow
\]

We write

\[
\langle P_0, \sigma_0 \rangle \rightarrow^* \langle P_n, \sigma_n \rangle \text{ when } n \geq 0,
\]

\[
\langle P_0, \sigma_0 \rangle \rightarrow^+ \langle P_n, \sigma_n \rangle \text{ when } n > 0.
\]

Remark that in our system, we have no rule such as

\[
\sigma(b) = \text{false} \quad \Rightarrow \quad \langle \text{wait } b, \sigma \rangle \rightarrow \langle \text{wait } b, \sigma \rangle
\]

Ie no busy waiting. Reductions may block. (Same remark for \text{await } b \text{ do } P).
Atomic statements (Exercices)

Exercice 5  If we make following extension

\[ P, Q ::= \ldots \mid \{P\} \]

what is the meaning of following rule?

\[
\begin{align*}
\langle P, \sigma \rangle & \rightarrow^+ \langle \bullet, \sigma' \rangle \\
\langle \{P\}, \sigma \rangle & \rightarrow \langle \bullet, \sigma' \rangle
\end{align*}
\]

Exercice 6  Show \texttt{await b do P} \equiv \{ \texttt{wait b; P} \}

Exercice 7  Meaning of \{\texttt{while true do skip}\}? Find simpler equivalent statement.

Exercice 8  Try to add procedure calls to our SOS semantics.
Producer - Consumer
A typical thread package. Modula-3

INTERFACE Thread;

TYPE
  T <: ROOT;
  Mutex = MUTEX;
  Condition <: ROOT;

A Thread.T is a handle on a thread. A Mutex is locked by some thread, or unlocked. A Condition is a set of waiting threads. A newly-allocated Mutex is unlocked; a newly-allocated Condition is empty. It is a checked runtime error to pass the NIL Mutex, Condition, or T to any procedure in this interface.
PROCEDURE Wait(m: Mutex; c: Condition);

The calling thread must have m locked. Atomically unlocks m and waits on c. Then relocks m and returns.

PROCEDURE Acquire(m: Mutex);

Wait until m is unlocked and then lock it.

PROCEDURE Release(m: Mutex);

The calling thread must have m locked. Unlocks m.

PROCEDURE Broadcast(c: Condition);

All threads waiting on c become eligible to run.

PROCEDURE Signal(c: Condition);

One or more threads waiting on c become eligible to run.
Locks

A LOCK statement has the form:

```
LOCK mu DO S END
```

where S is a statement and mu is an expression. It is equivalent to:

```
WITH m = mu DO
    Thread.Acquire(m);
    TRY S FINALLY Thread.Release(m) END
END
```

where m stands for a variable that does not occur in S.
A statement of the form:

```
TRY S_1 FINALLY S_2 END
```

executes statement $S_1$ and then statement $S_2$. If the outcome of $S_1$ is normal, the TRY statement is equivalent to $S_1 ; S_2$. If the outcome of $S_1$ is an exception and the outcome of $S_2$ is normal, the exception from $S_1$ is re-raised after $S_2$ is executed. If both outcomes are exceptions, the outcome of the TRY is the exception from $S_2$. 
Concurrent stack

Popping in a stack:

VAR nonEmpty := NEW(Thread.Condition);

LOCK m DO
    WHILE head = NIL DO Thread.Wait(m, nonEmpty) END;
    topElement := head;
    head := head.next;
END;

Pushing into a stack:

LOCK m DO
    newElement.next := head;
    head := newElement;
    Thread.Signal (nonEmpty);
END;

Caution: WHILE is safer than IF in Pop.
VAR table := ARRAY [0..999] of REFANY {NIL, ...};
VAR i:[0..1000] := 0;

PROCEDURE Insert (r: REFANY) =
BEGIN
IF r <> NIL THEN

    table[i] := r;
    i := i+1;

END;
END Insert;

Exercice 9  Complete previous program to avoid lost values.
Deadlocks

Thread $A$ locks mutex $m_1$
Thread $B$ locks mutex $m_2$
Thread $A$ trying to lock $m_2$
Thread $B$ trying to lock $m_1$

Simple strategy for semaphore controls

Respect a partial order between semaphores. For example, $A$ and $B$ uses $m_1$ and $m_2$ in same order.
Exercices

Exercice 10  Simulate conditions with semaphores. Hint: count number of waiting processes on condition.

Exercice 11  Readers and writers. A buffer may be read by several processes at same time. But only one process may write in it. Write procedures StartRead, EndRead, StartWrite, EndWrite.

Exercice 12  Give SOS for operations on conditions.