Operating Systems, Concurrency and Time

atomicity and interference

Johan Lukkien
Questions

• Where does concurrency occur?
• How do concurrent activities influence each other?
• How can we reason about concurrency?
• What are fundamental concepts in concurrent systems?
Notation for presentation

Regular, C or Pascal-like language

• do nothing: skip
• assignment: \( x := \text{expression} \)
• sequential composition: \( S; T \)
• parallel composition: \( S \parallel T \)
• selection: \( \text{if } B \text{ then } S \text{ [else } T \text{] fi} \)
• repetition: \( \text{while } B \text{ do } S \text{ od} \)
• extended on-the-fly
Notation

Possible description of a thread/process

\[ P_{\text{Temp}} = \]
\[ || \text{ var } i: \text{ int}; \]
\[ i := 0; \]
\[ \text{ while } true \text{ do } \text{ Temp := temperature; } i := i+1; \text{ DelayUntil (p*i) od } \]
\[ || \]

Block: introduction of local variables

Use \texttt{proc}, \texttt{func}, to define procedures/functions. Parameters and types as in Pascal
Agenda

• Concurrency & atomicity
• Correctness concerns, by example
• Concurrency concepts
Concurrency

• Interfering activities are a necessary ingredient of Operating Systems and concurrent programs

• Sources
  – Interrupt servicing
  – OS activity
  – multiple threads on same global data, arbitrarily switched
  – multiple processors
  – multiple active devices, sharing hardware resources (memory, bus)

• We study interference problems first in isolation
  – In this slide set we do not discriminate threads and process. We use the term *process* simply for an activity

• Question:
  – is there a difference between the multiple and single processor cases (‘real’ concurrency versus simulated concurrency)?
Concurrency & communication facilities

• Processes and threads
  – we use the word “process” for both, unless otherwise indicated

• Shared memory, e.g. within kernel, between processes and threads
  – just shared variables: data structures, event flags, spinlocks
  – semaphores, mutexes
  – condition variables with signalling
  – readers/writers locks

• Message passing
  – structured: message queues (channels)
  – streaming: pipes, fifos, sockets

• Signals
Starting point: the **sequential process**

Discrete:
- indivisible, *atomic* steps/actions
- execution never observed to be half-way an atomic action

**Execution: path through state-space**

Initial state:

\[
StateP = 1
\]

“Program”:

\[
\begin{align*}
StateP & := 2; \\
StateP & := 3; \\
StateP & := 4; \\
StateP & := 5;
\end{align*}
\]
Examples

• A computer, executing (indivisible) instructions
• Threads, executing their program code
• A car, driving from milestone 1 to milestone 5 (in fact: continuous process)

Note:
• discrete steps may be built from smaller ones
• in digital systems, the finest level of detail consists of atomic actions
• an execution is an interleaving of atomic actions
**Atomic?**

- $x := 1$
  - `mov #1, r1; st r1, @x`
  - no ‘internal’ interference point, hence to be regarded as atomic, assuming a correct implementation of interrupt handling

- $x := y$
  - `mov @y, r1; mov r1, @x`
  - ‘internal’ interference point: $r1$ may store an old copy of $y$ for a long time while computations with $y$ continue.

- $x := x+1$
  - `mov @x, r1; inc r1; mov r1, @x`

**Single reference rule:** a *statement (expression) in a programming language may be regarded as atomic if at most one reference to a shared variable occurs* (we ignore here compiler optimizations)
  - to make writing program texts more convenient

**Defined atomicity:** when we want to regard a non-atomic statement $S$ as atomic, we write `< S >`, e.g. `< x := x+1 >`
  - needs a motivation, e.g. refer to OS or hardware that guarantees this
Single reference rule

- In between any pair of instructions of one process, (part of) another process or collection of processes can be executed, including the OS.

- OS semantics is that this is transparent for processor state (the process will be restarted in the same processor state).

- Stale copies of shared variables can be stored in internal registers or in memory locations.

- This is problematic only if the final result cannot be seen as a possible interleaving of the (language-level) statements.

- Example:
  - initially: \( x=1, y=2 \)
  - program: \( x := y \parallel y := x \)
  - final values: \( (1,1), (2,2), (2,1) \) \([1,2]?)\)
Concurrent execution

- Joint path through joint state space
- Trace (or execution): sequence of atomic actions, obtained by interleaving of concurrent parts while maintaining the order of the individual processes
  - many possible traces
Concurrent systems

- components in a PC
- processes in a multi-tasking environment
- threads within a process
Major issues in concurrency

- **co-operation**:  
  - sharing resources (hardware like printer, scanner, disk, ..., or most likely: memory and processors)  
  - transfer information: synchronization and communication (needs shared resources)

- **interference**, mutual influence (good or bad):

  
  Assumptions or knowledge that one process has about the state, are disturbed by actions of another process

  - **good interference**: e.g. wait for another process to set a boolean to true indicating delivery of a value in a variable
    
    - $x := \text{false}$;\{ $\neg x \}$ while $\neg x$ do \text{skip} od; “use y” \text{||} .... y := E; x := true .....  
    
    - Question: are there ‘tricky’ interleavings?

  - **bad**: access a resource (e.g. a printer) after checking its availability; in between the check and the use the resource is accessed by another process
    
    - if avail then \{ avail \} avail := false; “use resource”; avail := true; fi \text{||} ...same
Agenda

- Concurrency & atomicity
- Correctness concerns, by example
- Concurrency concepts
Example shared resource: narrow bridge

Mutual exclusion:
- Only one car at a time on the bridge
- Admissible paths avoid the middle squares
Synchronize the cars

- **Synchronization:**
  - co-ordinate execution of the given programs such that no two cars access the bridge at the same time
  - synchronization refers to ordering, or to restricting possible paths, typically through limiting event occurrences

- **Mutual exclusion:**
  - the particular synchronization problem of exclusive access to a resource, program fragment or data structure

\[
P_X = \begin{align*}
\text{while } & \text{true do} \\
& \text{State}X := 1; \\
& \text{State}X := 2; \\
& \text{State}X := 3; \\
& \text{State}X := 4; \\
& \text{State}X := 5
\end{align*}
\]

\[
P_Y = \begin{align*}
\text{while } & \text{true do} \\
& \text{State}Y := 1; \\
& \text{State}Y := 2; \\
& \text{State}Y := 3; \\
& \text{State}Y := 4; \\
& \text{State}Y := 5
\end{align*}
\]

Initially:
\[\text{State}X = 1 \land \text{State}Y = 1\]
Synchronize through booleans

- Introduce boolean variables $bX$ and $bY$ to record crossing.
  - Initially: $\neg bX \land \neg bY$
  - On the bridge: $\neg (bX \land bY)$

\[
P_X = \text{while true do}
\begin{align*}
  &\text{StateX := 1;} \\
  &\text{StateX := 2;} \\
  &\text{while bY do skip od;}
  \\
  &\{ (1): \neg bY \land \neg bX \} \\
  &\text{bX := true;}
  \\
  &\{ (2): \neg bY \land bX \} \\
  &\text{StateX := 3; StateX := 4;}
  \\
  &\text{bX := false;}
  \\
  &\text{StateX := 5}
\end{align*}
\]

\[
P_Y = \text{while true do}
\begin{align*}
  &\text{StateY := 1;} \\
  &\text{StateY := 2;} \\
  &\text{while bX do skip od;}
  \\
  &\{ (3): \neg bX \land \neg bY \} \\
  &\text{bY := true;}
  \\
  &\{ (4): \neg bX \land bY \} \\
  &\text{StateY := 3; StateY := 4;}
  \\
  &\text{bY := false;}
  \\
  &\text{StateY := 5}
\end{align*}
\]

||
Synchronize through booleans

- Introduce boolean variables $b_X$ and $b_Y$ to record crossing.
  - Initially: $\neg b_X \land \neg b_Y$
  - On the bridge: $\neg (b_X \land b_Y)$

```plaintext
WRONG: both $P_X$ and $P_Y$ may find $b_Y$ resp. $b_X$ false and then proceed onto the bridge.

Assertions (1),(3) can be falsified

$P_X = \text{while true do}$

StateX := 1;
StateX := 2;
while $b_Y$ do skip od;
{ (1): $\neg b_Y \land \neg b_X$ }
bX := true;
{ (2): $\neg b_Y \land b_X$ }
StateX := 3; StateX := 4;
bX := false;
StateX := 5
od
```

$P_Y = \text{while true do}$

StateY := 1;
StateY := 2;
while $b_X$ do skip od;
{ (3): $\neg b_X \land \neg b_Y$ }
bY := true;
{ (4): $\neg b_X \land b_Y$ }
StateY := 3; StateY := 4;
bY := false;
StateY := 5
do
```
Change the order...

- Apparently, $bX$ and $bY$ should record the interest in using the bridge: change the order

\[
\begin{align*}
P_X = & \textbf{while} \ true \ \textbf{do} \\
& \text{State}X := 1; \\
& \text{State}X := 2; \\
& bX := true; \\
& \{ bX \} \\
& \textbf{while} bY \ \textbf{do} \hspace{1em} \textbf{skip} \hspace{1em} \textbf{od}; \\
& \{ (\neg bY \lor P_Y \ \text{blocked}) \land bX \} \\
& \text{State}X := 3; \text{State}X := 4; \\
& bX := false; \\
& \text{State}X := 5 \\
& \textbf{od} \\
\end{align*}
\]

\[
\begin{align*}
P_Y = & \textbf{while} \ true \ \textbf{do} \\
& \text{State}Y := 1; \\
& \text{State}Y := 2; \\
& bY := true; \\
& \{ bY \} \\
& \textbf{while} bX \ \textbf{do} \hspace{1em} \textbf{skip} \hspace{1em} \textbf{od}; \\
& \{ (\neg bX \lor P_X \ \text{blocked}) \land bY \} \\
& \text{State}Y := 3; \text{State}Y := 4; \\
& bY := false; \\
& \text{State}Y := 5 \\
& \textbf{od} \\
\end{align*}
\]
Change the order...

- Apparently, $bX$ and $bY$ should record the interest in using the bridge: change the order

\[ P_X = \textbf{while } true \textbf{ do} \]
\[ \quad \text{StateX} := 1; \]
\[ \quad \text{StateX} := 2; \]
\[ \quad bX := true; \]
\[ \quad \{ bX \} \]
\[ \quad \textbf{while } bY \textbf{ do} \textbf{ skip} \textbf{ od}; \]
\[ \quad \{ (\neg bY \lor P \land \text{blocked}) \land bX \} \]
\[ \] \[
P_Y = \textbf{while } true \textbf{ do} \]
\[ \quad \text{StateY} := 1; \]
\[ \quad \text{StateY} := 2; \]
\[ \quad bY := true; \]
\[ \quad \{ bY \} \]
\[ \quad \textbf{while } bX \textbf{ do} \textbf{ skip} \textbf{ od}; \]
\[ \quad \{ (\neg bX \lor P \land \text{blocked}) \land bY \} \]
\[ \quad \text{od} \]

\[
\text{WRONG: both } P_X \text{ and } P_Y \text{ may set } bX \text{ resp. } bY \text{ to } true \text{ and then never proceed anymore DEADLOCK}
\]
Take turns...

- Rather than trying to obtain access to the bridge, the processes give this access away using a variable called \( t \) (for turn)
  - Initially: \( t = X \lor t = Y \)

\[
P_X = \textbf{while } \text{true } \text{do} \\
\quad \text{StateX} := 1; \\
\quad \text{StateX} := 2; \\
\quad t := Y; \\
\quad \textbf{while } t \neq X \textbf{ do skip od; } \\
\quad \{ t = X \} \\
\quad \text{StateX} := 3; \text{StateX} := 4; \\
\quad \text{StateX} := 5 \\
\textbf{od}
\]

\[
P_Y = \textbf{while } \text{true } \text{do} \\
\quad \text{StateY} := 1; \\
\quad \text{StateY} := 2; \\
\quad t := X; \\
\quad \textbf{while } t \neq Y \textbf{ do skip od; } \\
\quad \{ t = Y \} \\
\quad \text{StateY} := 3; \text{StateY} := 4; \\
\quad \text{StateY} := 5 \\
\textbf{od}
\]
Take turns...

- Rather than trying to obtain access to the bridge, the processes give this access away using a variable called $t$ (for turn)
  - Initially: $t = X \lor t = Y$

**WRONG:** $P_X$ and $P_Y$ take turns and need each other, even if the partner is not interested

Waiting is not minimal

However, boolean flags can be used to indicate event occurrence

$P_X$:

```
PX =
while true
StateX := 1;
StateX := 2;
t := Y;
while t ≠ X do
  skip
od
{ t = X }
StateX := 3; StateX := 4;
StateX := 5
od
```

$P_Y$:

```
PY =
while true
StateY := 1;
StateY := 2;
t := X;
while t ≠ Y do
  skip
od
{ t = Y }
StateY := 3; StateY := 4;
StateY := 5
od
```
Synchronization with spinlocks

- What is needed is to perform two actions together as a single atomic step, as provided with spinlocks

- Example: `Test&Set` (hardware instructions by the processor):
  - `func Test&Set (lock): <old := lock; lock := true; return old>
  - `<lock := false`

- Use shared variable `lock` (init `false`)

\[ P_X = \textbf{while} \text{true do} \]
\[
  \text{State}X := 1; \\
  \text{State}X := 2; \\
  \textbf{while} \ Test&Set \ (\text{lock}) \ 	extbf{do} \ skip \ 	extbf{od}; \\
  \text{State}X := 3; \ \text{State}X := 4; \\
  \text{lock} := \text{false}; \\
  \text{State}X := 5 \\
\text{od} \]

\[ P_Y = \textbf{while} \text{true do} \]
\[
  \text{State}Y := 1; \\
  \text{State}Y := 2; \\
  \textbf{while} \ Test&Set \ (\text{lock}) \ 	extbf{do} \ skip \ 	extbf{od}; \\
  \text{State}Y := 3; \ \text{State}Y := 4; \\
  \text{lock} := \text{false}; \\
  \text{State}Y := 5 \\
\text{od} \]
Agenda

• Concurrency & atomicity
• Correctness concerns, by example
• Concurrency concepts
Summary: concepts in concurrency

- **Atomic action**: finest grain of detail, indivisible
  - typically, assignments and tests in a program
  - at program level: single reference to shared variable in statement
    - ignoring possible optimization and reordering by compiler/processor
- **Parallel execution**: interleaving of atomic actions
- **Shared variables**: accessible to several processes (thread)
- **Private variables**: accessible only to a single process (thread)
- **Interference**: disturbing assumptions about the state
  - usually, caused by “unexpected” interleaving
  - particularly difficult and unexpected with shared memory
- **Race conditions (critical races)**: situation in which correctness depends on the execution order of concurrent activities (“bad interference”)
  - activity: any level – circuit, hardware component, thread, ...
  - often associated with forms of busy waiting...
    - while (! IntrptFlag) /* wait */; /*assume IntrptFlag */; ....; IntrptFlag = false;
  - ... or related to ‘stale state’ e.g., an old copy of a variable
Summary: requirements on solutions

- **Functional correctness:**
  - satisfy the given specification (e.g., mutual exclusion).

- **Minimal waiting:** ("wait for a reason")
  - waiting only when correctness is in danger.

- **Absence of deadlock:**
  - don’t manoeuvre (part of) the system into a state such that progress is no longer possible.

- **(Absence of livelock):** repeated trying without progress
  - ensure convergence towards a decision in a synchronization protocol)

- **Fairness in competition:**
  - (weak) eventually, each contender should be admitted to proceed.
  - (strong) we can put a bound on the waiting time of a contender.
  - absence of fairness: leads to starvation of processes.

**Note:** in real-time systems, fairness and minimal waiting may be less important than latency and predictability
Exercises

I.1 Consider the parallel execution of statements $P$ and $Q$. Initially, variable $x$ equals 0. What are the possible final values of $x$ with

a. $P: x := 1$ and $Q: x := 2$
b. $P: x := x+1$ and $Q: x := x+2$
c. $P: y := x; x := y+1$ and $Q: x := x+1$

Look at the possible orders.
Exercises

I.2 In most computers, an assignment like \( x := x+1 \) is not an atomic action. It is usually executed through copying via an internal register. In the program below the two processes perform \( x := x+1 \) 100 times each, but it is written in actions that can be regarded as atomic.

\[
\begin{align*}
i &:= 0; \\
\textbf{while} & i \neq 100 \textbf{ do} \\
& r := x; \\
& r := r+1; \\
& x := r; \\
& i := i+1 \\
\textbf{od}
\end{align*}
\]

\[
\begin{align*}
j &:= 0; \\
\textbf{while} & j \neq 100 \textbf{ do} \\
& s := x; \\
& s := s+1; \\
& x := s; \\
& j := j+1 \\
\textbf{od}
\end{align*}
\]

If \( x \) is initially 0, what are the possible final values of \( x \)?
Exercises

I.3 Instead of using a the \textit{Test\&Set()}, an exclusion algorithm can be constructed based on just the assignments to $t$ and $bX$ (and $bY$). This is called \textit{Peterson’s algorithm}. Can you find it? Does it satisfy the correctness criteria?
#include <stdio.h>
#include <pthread.h>

int x;

void *Count_100 ()
{
    int i, s;
    for (i = 0; i < 100; i++) {
        x = x+1;
    }
}

int main ()
{
    pthread_t thread_id;

    x = 0;
pthread_create (&thread_id, NULL, Count_100, NULL);
    Count_100 ();
pthread_join (thread_id, NULL);
    printf("x = %d\n", x);
}

• What are possible final values of x?
Peterson’s algorithm

Peterson’s algorithm for mutual exclusion between two processes

\[ P_X = \textbf{while true do} \]
\[ \quad \text{State}_X := 1; \]
\[ \quad \text{State}_X := 2; \]
\[ \quad b_X := \text{true}; \{ b_X \ (\ast) \} \]
\[ \quad t := Y; \{ b_X \} \]
\[ \quad \textbf{while } \langle b_Y \land t \neq X \rangle \textbf{ do skip od; } \]
\[ \quad \{ b_X \land (t = X \lor \neg b_Y \lor P_Y \text{at } (\ast)) \} \]
\[ \quad \text{State}_X := 3; \]
\[ \quad \text{State}_X := 4; \]
\[ \quad b_X := \text{false}; \]
\[ \quad \text{State}_X := 5 \]
\[ \textbf{od} \]

combine the ideas:

- take turns in crowded circumstances (use \( t \))
- don’t wait if there is no need (look at \( b_Y \))
- (\( \ast \)) denotes the point in between the two assignments

• Note: \( P_Y \) is the symmetric counterpart

Peterson’s algorithm

for mutual exclusion

between two processes