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 := expression*
- sequential composition: *S; T*
- parallel composition: *S* ||*T*
- selection: **if** *B* **then** *S* [**else** *T*] **fi**
- repetition: **while** *B* **do** *S* **od**
- extended on-the-fly

Notation

Possible description of a thread/process

```
P_Temp=
   |[  var i: int;
       i  := 0;
       while true do Temp := temperature; i := i+1; DelayUntil (p*i) od
   ]|
```

Block: introduction of local variables

Use **proc**, **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

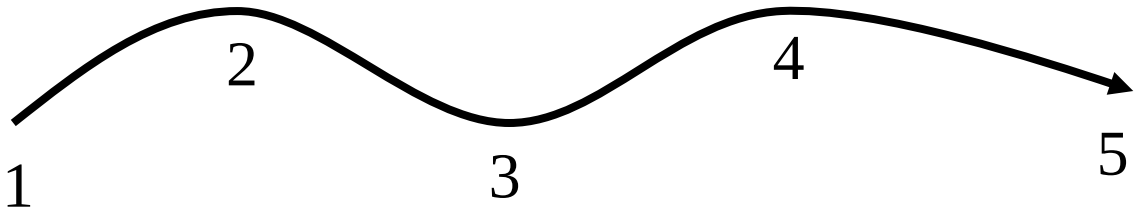
- Processes and threads
  - we use the word “process” for both, unless otherwise indicated

- 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*



Execution: path through state-space

Discrete:

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

Initial state:
StateP = 1

"**Program**":
StateP := 2;
StateP := 3;
StateP := 4;
StateP := 5;

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?

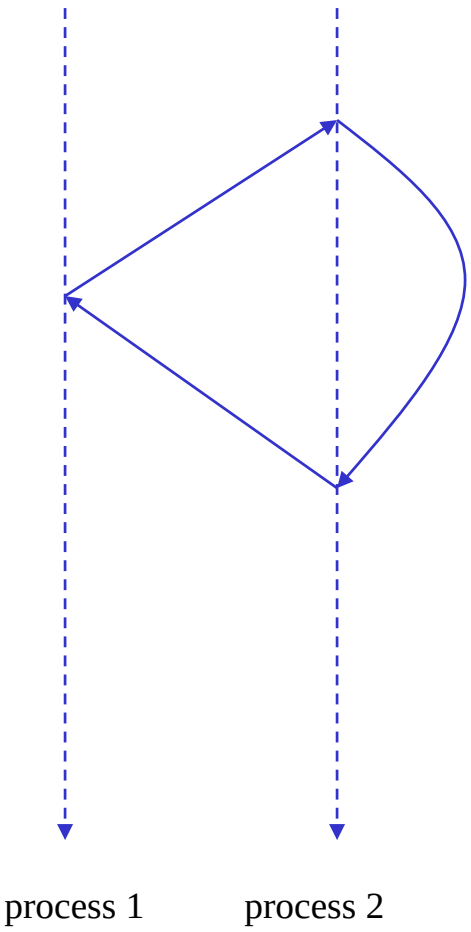
- *x := 1*
  - **mov** *#1, r1*; **st** *r1, @x*
  - no ‘internal’ interference point, hence to be regarded as atomic, assuming a correct implementation of interrupt handling

- *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

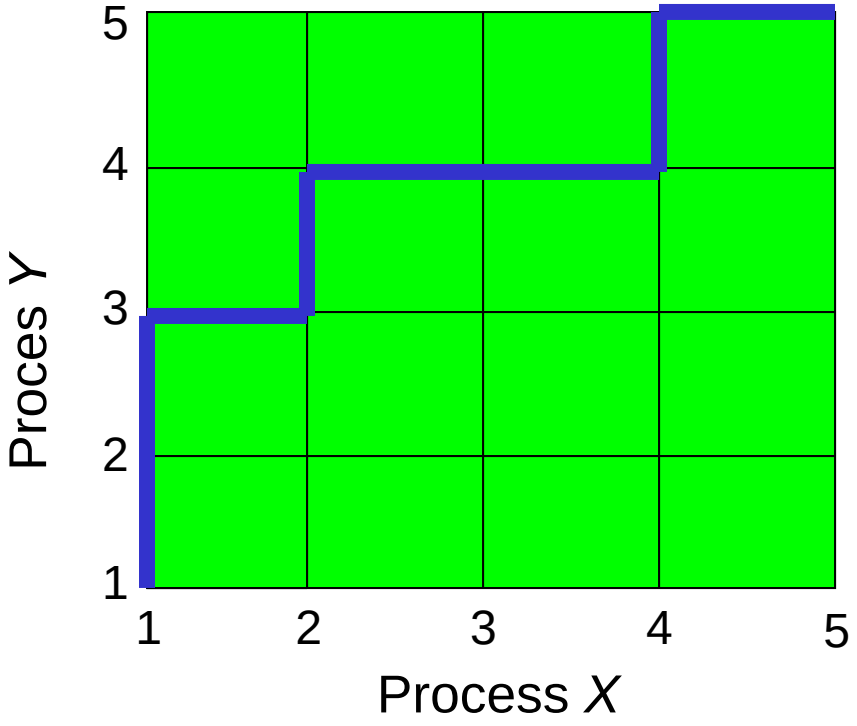
Single reference rule



- 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 || y := x*
  - final values: (1,1), (2,2), (2,1) [(1,2)?]

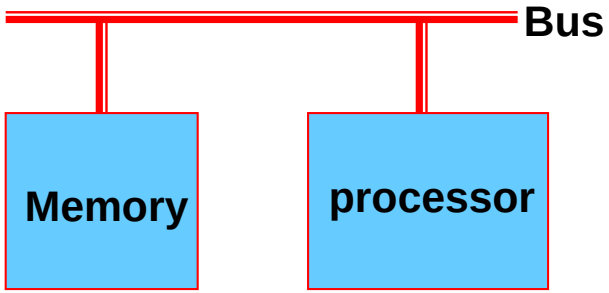
Concurrent execution

- 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

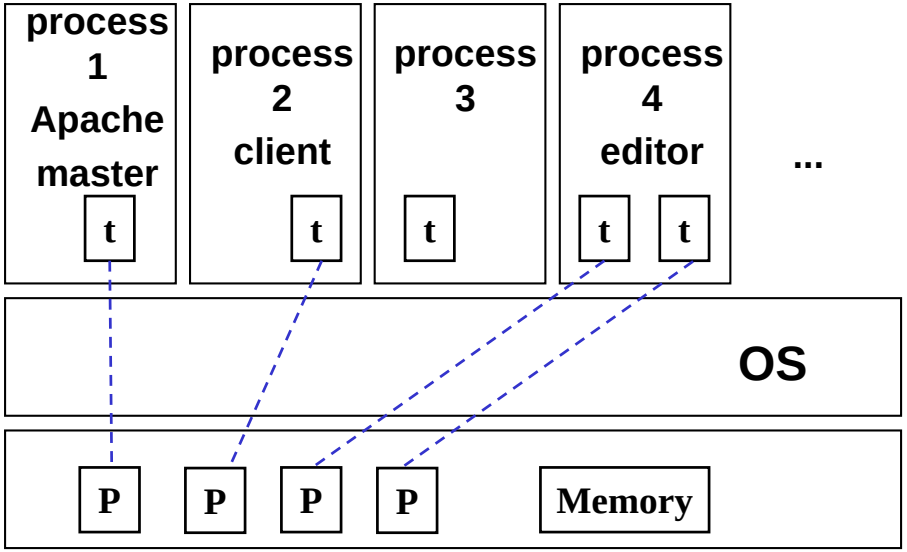


Concurrent systems

- 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

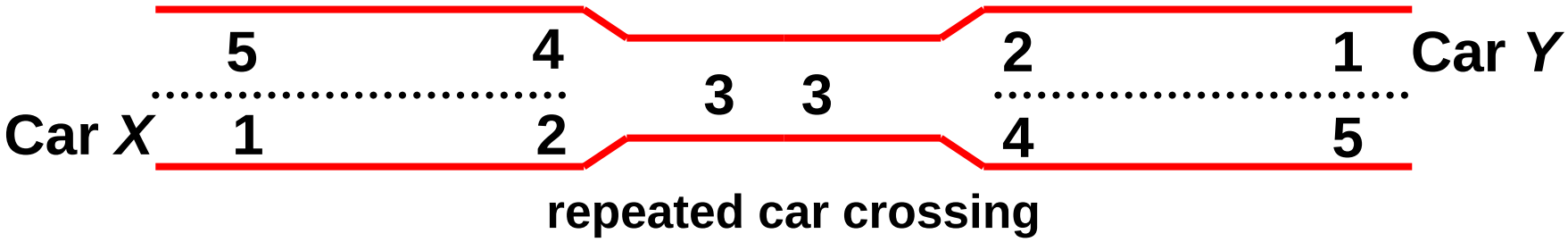
  - *good interference*: e.g. wait for another proces to set a boolean to *true* indicating delivery of a value in a variable
    - *x := false*; { ¬*x* } **while** ¬*x* **do** *skip* **od**; "use *y*" || .... *y := E*; *x := true* .....
    - Question: are there 'tricky' interleavings?

  - *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** || ...same

Agenda

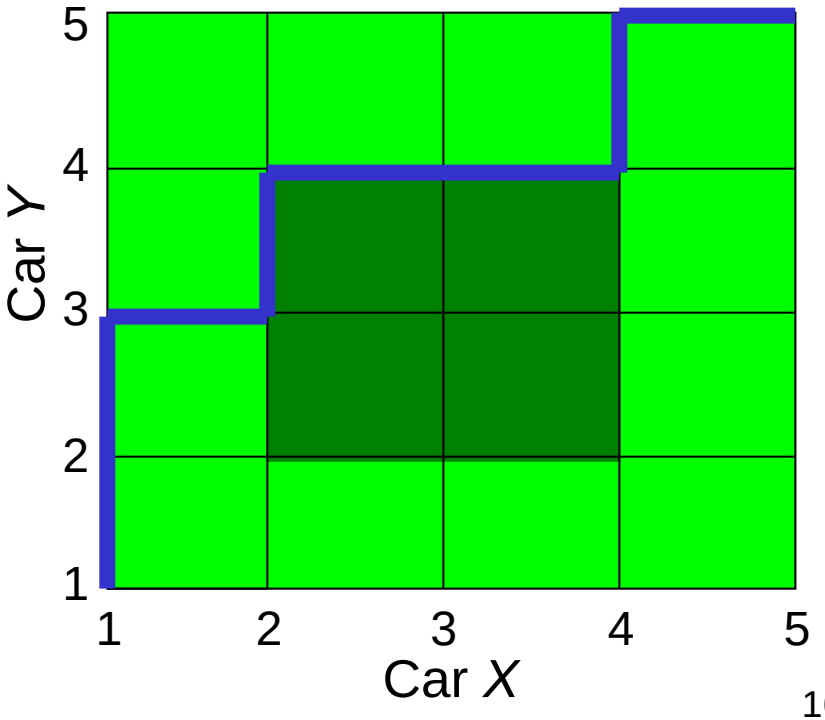
- Concurrency & atomicity
- Correctness concerns, by example
- Concurrency concepts

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

```
P_X =
  while true do
    StateX := 1;
    StateX := 2;
    StateX := 3;
    StateX := 4;
    StateX := 5
  od
```

||

```
P_Y =
  while true do
    StateY := 1;
    StateY := 2;
    StateY := 3;
    StateY := 4;
    StateY := 5
  od
```

Initially:

$StateX = 1 \wedge StateY = 1$

- **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

Synchronize through booleans

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

```
P_X = while true do
        StateX := 1;
        StateX := 2;
        while bY do skip od;
        { (1): ¬bY ∧ ¬bX }
        bX := true;
        { (2): ¬bY ∧ bX }
        StateX := 3; StateX := 4;
        bX := false;
        StateX := 5
      od
```

||

```
P_Y = while true do
        StateY := 1;
        StateY := 2;
        while bX do skip od;
        { (3): ¬bX ∧ ¬bY }
        bY := true;
        { (4): ¬bX ∧ bY }
        StateY := 3; StateY := 4;
        bY := false;
        StateY := 5
      od
```

Synchronize through booleans

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

WRONG: both P_X and P_Y may find *bY* resp. *bX* *false* and then proceed onto the bridge.

Assertions (1),(3) can be falsified

P_X = **v**

```
StateX := 1;
StateX := 2;
while bY do skip od;
{ (1): ¬bY ∧ ¬bX }
bX := true;
{ (2): ¬bY ∧ bX }
StateX := 3; StateX := 4;
bX := false;
StateX := 5
od
```

||

```
StateY := 1;
StateY := 2;
while bX do skip od;
{ (3): ¬bX ∧ ¬bY }
bY := true;
{ (4): ¬bX ∧ bY }
StateY := 3; StateY := 4;
bY := false;
StateY := 5
od
```

Change the order...

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

```
P_X = while true do
        StateX := 1;
        StateX := 2;
        bX := true;
        { bX }
        while bY do skip od;
        { (¬bY ∨ P_Y blocked) ∧ bX }
        StateX := 3; StateX := 4;
        bX := false;
        StateX := 5
      od
```

||

```
P_Y = while true do
        StateY := 1;
        StateY := 2;
        bY := true;
        { bY }
        while bX do skip od;
        { (¬bX ∨ P_X blocked) ∧ bY }
        StateY := 3; StateY := 4;
        bY := false;
        StateY := 5
      od
```

Change the order...

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

P_X = **while** *true* **do**
    StateX := 1;
    StateX := 2;
    bX := true;
    { *bX* }
    while *bY* **do** *skip* **od**;
    { (¬*bY* ∨ P_Y *blocked*) ∧ *bX* }
    StateX := 4;
od

||

P_Y = **while** *true* **do**
    StateY := 1;
    StateY := 2;
    bY := true;
    { *bY* }
    while *bX* **do** *skip* **od**;
    { (¬*bX* ∨ P_X *blocked*) ∧ *bY* }
    StateY := 4;
od

WRONG: both P_X and P_Y may set *bX* resp. *bY* to *true* 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: $t = X \vee t = Y$

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

||

```
P_Y = while true do
        StateY := 1;
        StateY := 2;
        t := X;
      while t ≠ Y do skip od;
      { t = Y }
        StateY := 3; StateY := 4;
        StateY := 5
      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: $t = X \vee t = Y$

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

while $t \neq X$ **do** *skip* **od;**
$\{ t = X \}$
StateX := 3; StateX := 4;
StateX := 5
od

||

while $t \neq Y$ **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 = while true do
        StateX := 1;
        StateX := 2;
        while Test&Set (lock)
        do skip od;
        StateX := 3; StateX := 4;
        lock := false;
        StateX := 5
      od
```

||

```
P_Y = while true do
        StateY := 1;
        StateY := 2;
        while Test&Set (lock)
        do skip od;
        StateY := 3; StateY := 4;
        lock := false;
        StateY := 5
      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 process (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.

```
i := 0;
while i ≠ 100 do
    r := x;
    r := r+1;
    x := r;
    i := i+1
od
```

||

```
j := 0;
while j ≠ 100 do
    s := x;
    s := s+1;
    x := s;
    j := j+1
od
```

If x is initially 0, what are the possible final values of *x*?

Exercises

I.3 Instead of using a the *Test&Set(),* an exclusion algorithm can be constructed based on just the assignments to *t* and *bX* (and *bY*). This is called *Peterson's algorithm.* Can you find it? Does it satisfy the correctness criteria?

Exercise

```
#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

P_X = **while** *true* **do**

$\quad$ *StateX := 1;*

$\quad$ *StateX := 2;*

$\quad$ *bX := true;* { *bX* (*) }

$\quad$ *t := Y;* { *bX* }

$\quad$ **while** $< bY \wedge t \neq X >$ **do** *skip* **od**;

$\quad$ $\{ bX \wedge (t = X \vee \neg bY \vee P_Y \text{ at } (*))\}$

$\quad$ *StateX := 3;*

$\quad$ *StateX := 4;*

$\quad$ *bX := false;*

$\quad$ *StateX := 5*

od

combine the ideas:

- take turns in crowded circumstances (use *t*)
- don't wait if there is no need (look at *bY*)
- (*) denotes the point in between the two assignments

- Note: P_Y is the symmetric counterpart

Peterson's algorithm for mutual exclusion between two processes