Operating Systems 2010/2011

Atomicity and Interference

Johan Lukkien

Agenda

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

Concurrency

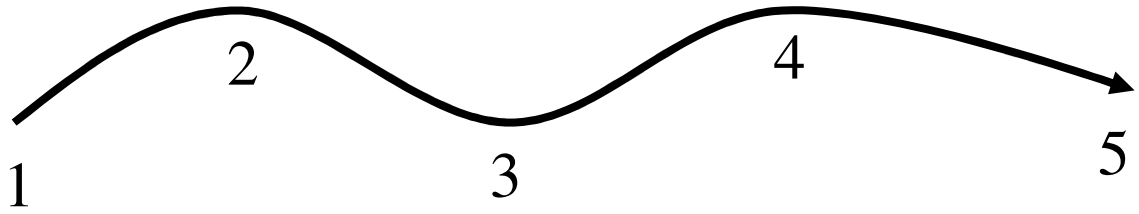
- Interfering activities are a necessary ingredient of Operating Systems and concurrent programs

- 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)?

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)

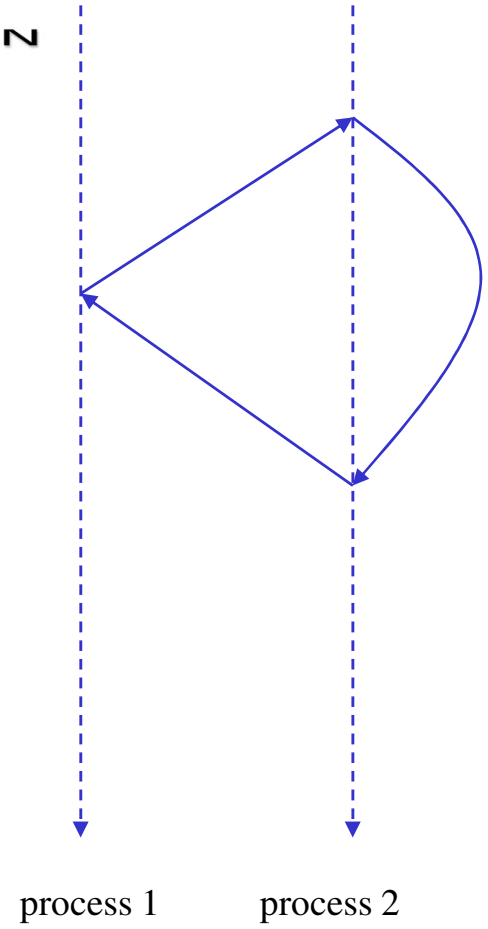
Note:

- discrete steps may be built from smaller ones
- in digital systems, the finest level of detail consists of *atomic actions*

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)*
- **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

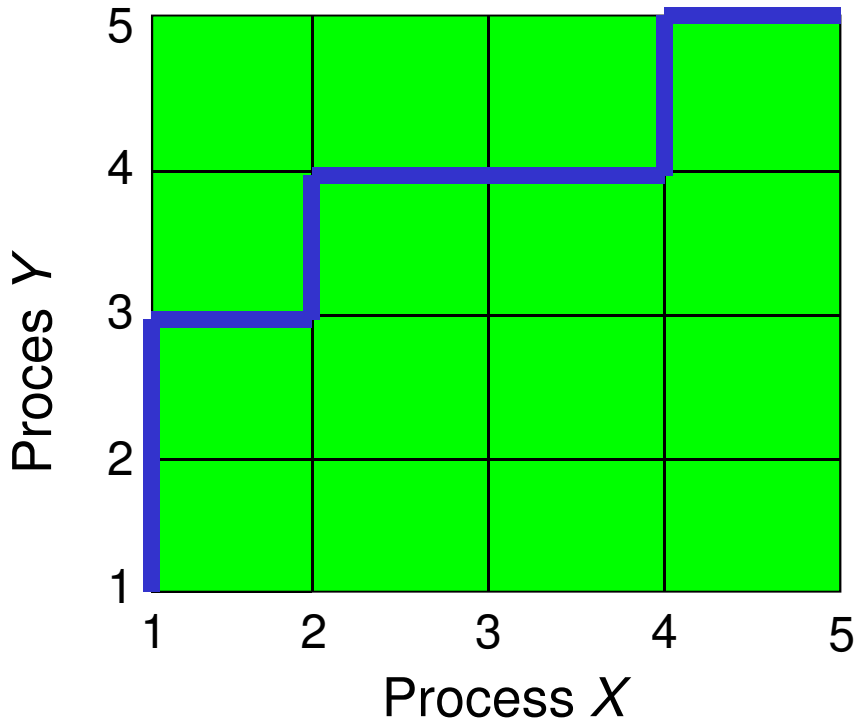
Single reference rule



- 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
- 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)?]

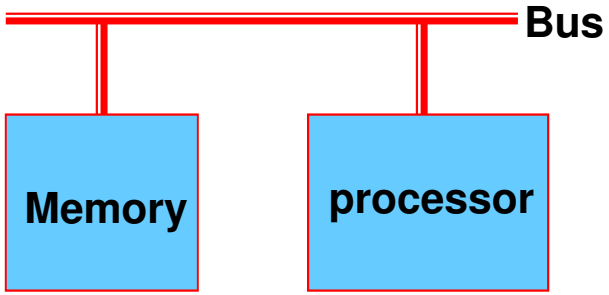
Concurrent execution

- Joint path through joint state space
- *Trace:* sequences of atomic actions, obtained by interleaving of concurrent parts while maintaining the order of the individual processes
  - many possible traces

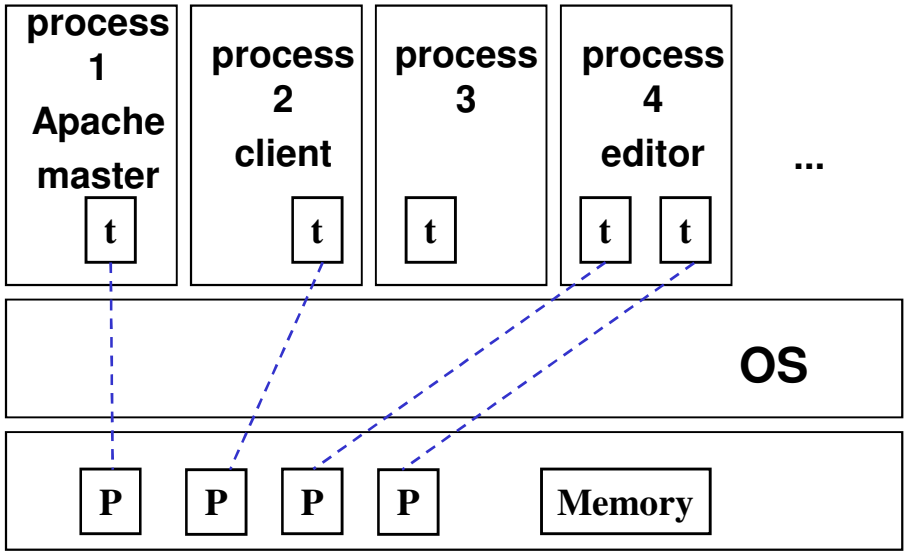


Concurrent systems

- 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*: 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

Traces

- Traces are sequences of (atomic) actions; they represent the steps that a program goes through
  - actions are assignments or tests
    - a trace is a <u>possible</u> one if all its tests yield true
      - being possible depends on initial program state
  - traces are finite or infinite
  - one program text has many traces

- Example (just first repetition of previous slide)
  - *(x:=false)(x),*
    (x:=false)(¬x)(skip)(x),
    (x:=false)(¬x)(skip)(¬x)(skip)(x),
    .....
    - note: *(x)* and *(¬x)* denote tests
  - of this, only the infinite one is actually possible for this program in isolation
    - because *(x)* never yields *true*

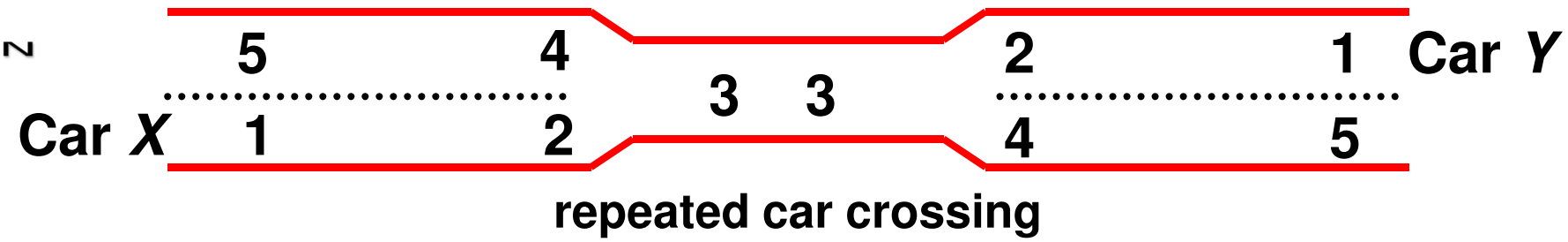
Traces

- The traces of a concurrent program are obtained by interleaving traces of all concurrent parts
  - Example: interleave *(x:=false)(x)* with *(y:=E)(x:=true)*
    - *(y:=E)(x:=true)(x:=false)(x)*, *(y:=E)(x:=false)(x:=true)(x)*, *(y:=E)(x:=false)(x)(x:=true)*, *(x:=false)(y:=E)(x:=true)(x)*, *(x:=false)(y:=E)(x)(x:=true)*, *(x:=false)(x)(y:=E)(x:=true)*
  - note: two traces are now possible, while before *(x:=false)(x)* was not a possible trace
    - meaning that in the repetition example, finite traces are actually possible
- ...impractical to determine them all
  - however, most often considering a limited number suffices
    - e.g. the important issues is whether *(x:=true)* occurs between *(x:=false)* and *(x)* or not
  - they can be generated automatically
  - and they can be useful to obtain insight

Agenda

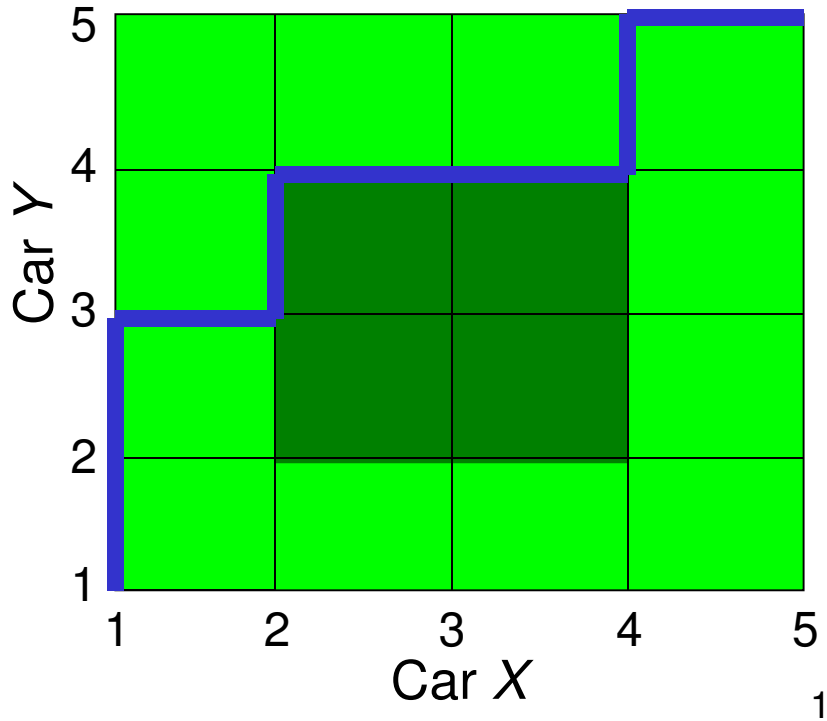
- Concurrency & atomicity
- Correctness concerns, by example
- Concurrency concepts

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*, (restricting *possible* paths or forbidding certain traces) typically through indicating *event occurrences*

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

$P_X = \mathbf{v}$

```
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 := 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**;
  { $(\neg bY \vee P_Y$ *blocked*$) \wedge bX$ }
  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**;
  { $(\neg bX \vee P_X$ *blocked*$) \wedge bY$ }
  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**;

||

P_Y = **while** *true* **do**
    StateY := 1;
    StateY := 2;
    bY := true;
    { *bY* }
    while *bX* **do** *skip* **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 $t \neq X$ **do** *skip* **od;**
  $\{ t = X \}$
  $StateX := 3;\ StateX := 4;$
  $StateX := 5$
od

||

P_Y = **while** *true* **do**
  $StateY := 1;$
  $StateY := 2;$
  $t := X;$
  while $t \neq Y$ **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

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

||

```
    StateY := 2;
    t := X;
    while t ≠ Y do skip od;
    { t = Y }
    StateY := 3; StateY := 4;
    StateY := 5
  od
```

Peterson's algorithm

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

$StateX := 1;$

$StateX := 2;$

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

$t := Y;$ { bX }

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

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

$StateX := 3;$

$StateX := 4;$

$bX := false;$

$StateX := 5$

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

Peterson's algorithm - correctness

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

$StateX := 1;$

$StateX := 2;$

$bX, auxX := true, true;$

$\{ bX \wedge auxX \}$

$t, auxX := Y, false;$

$\{ bX \wedge \neg auxX \}$

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

$\{ bX \wedge \neg auxX \wedge$
$(t = X \vee \neg bY \vee auxY) \}$

$StateX := 3;$

$StateX := 4;$

$bX := false;$

$StateX := 5;$

od

Detailed annotation:

- The annotation shows exclusion and absence of deadlock
- use an auxiliary variable (just for the proof) to record the state of the program in between the two statements
- show that no assignment in P_Y can disturb any assertion
- show that locally these assertions are indeed correct
- in principle the test is not atomic! It can be shown (exercise) that detailing it still leads to a correct program

Alternative: investigate traces

Using the annotation

- Proving facts: use program text and assertions
  - these are, in fact, statements about all possible traces without the need to examine them explicitly
- **Both on the bridge**: both assertions hold

$$
\begin{aligned}
& (bX \wedge \neg auxX \wedge (t = X \vee \neg bY \vee auxY)) \wedge \\
& (bY \wedge \neg auxY \wedge (t = Y \vee \neg bX \vee auxX)) \\
= \quad & \{ \text{all terms with } \neg bX,\ auxY,\ \neg bY \text{ or } auxX \text{ drop out} \} \\
& bX \wedge bY \wedge \neg auxX \wedge \neg auxY \wedge (t = X \wedge t = Y) \\
= \quad & \{ t \text{ cannot be both } X \text{ and } Y \} \\
& false
\end{aligned}
$$

- **Deadlock** (both blocked): both guards hold

$$
\begin{aligned}
& bY \wedge t \neq X \wedge bX \wedge t \neq Y \\
= \quad & \{ t = X \vee t = Y \} \\
& false
\end{aligned}
$$

Non-interference, formally

- Assume that we have two processes and assignments with annotation based on local analysis

```
....          ....
{ P0 }        { P1 }
A0      //     A1
....          ....
```

Then we must show that mutual disturbance is impossible:

- *A0* does not disturb *P1; A1* does not disturb *P0*
  - $\{ P0 \wedge P1 \}$ *A0* $\{ P1 \}$
  - $\{ P0 \wedge P1 \}$ *A1* $\{ P0 \}$

Non-interference, in the example

- Consider
  - $\{ bX \wedge auxX \}$ $t, auxX := Y, false;$
  - This must not be disturbed by *any* assignment of P_Y
  - Of course, only assignments that *may disturb the condition* are interesting for consideration
  - Hence, the above one is natural interference-free; both variables are assigned only by P_X

- Consider
  - $\{ bX \wedge \neg auxX \wedge (t = X \vee \neg bY \vee auxY) \}$ $StateX := 3$
  - for this condition, assignments of P_Y
    - $\{ true \}$ $bY, auxY := true, true;$
    - $\{ bY \wedge auxY \}$ $t, auxY := X, false;$

    are problematic and need verification

Verification

- Check
  - $\{ bX \wedge \neg auxX \wedge (t = X \vee \neg bY \vee auxY) \wedge bY \wedge auxY \}$

    $t, auxY := X, false;$

    $\{ bX \wedge \neg auxX \wedge (t = X \vee \neg bY \vee auxY) \}$

- Check
  - $\{ bX \wedge \neg auxX \wedge (t = X \vee \neg bY \vee auxY) \}$

    $bY, auxY := true, true;$

    $\{ bX \wedge \neg auxX \wedge (t = X \vee \neg bY \vee auxY) \}$

- Follows from direct substitution using rule of assignment
  - $\{ P \}\ v := E\ \{ Q \}$ holds iff $P \Rightarrow Q(v := E)$

Where do you find this level of interference and program detail?

- Inside kernel implementations for multiprocessors
  - typically, multiprocessor implementations of shared data structures (scheduler, process management, kernel activities)
  - device drivers
  - interrupt service routines
    - particularly, nested interrupts
  - there are, however, special hardware instructions with similar effects – Peterson's algorithm is not used
- In direct hardware realizations of a program
- In multithreaded programs
  - particularly, with modern hyper threaded processors or multiple cores

Mutual exclusion

- Actions sequences with the bridge (between going on and off) must not be interleaved
- One way of looking at the bridge problem is to regard the bridge as a *shared resource*
  - acquired by cars from both direction
- We say
  - 'the bridge is accessed under *mutual exclusion*'
  - or: 'the entry/exit actions form a *critical section*'
- Programming mutual exclusion can be using the concept of a (binary) *semaphore*
  - in fact, turning *sequences of atomic actions* into a single one

Semaphores (Dijkstra)

- Semaphore s is an integer *s* with initial value $s_0 \geq 0$ and *atomic* operations *P(s)* and *V(s).* The effect of these operations is defined as follows:

  P(s): < await(s>0); s := s-1 >
  V(s): < s := s+1 >

- "< >" denotes again atomicity: the implementation of *P* and *V* must guarantee this
- '*await(s>0)*' represents blocking until '*s>0'* holds. This is indivisibly combined with a decrement of *s*
- a semaphore is therefore always non-negative
- Other names for *P* and *V*: *wait/signal, wait/post, lock/unlock*

- Semaphores can be used to implement mutual exclusion

Example

- Let $s_0 = 1$
- At most one process can 'pass' the semaphore
  - two processes passed would mean the semaphore was decreased twice
  - hence, the value would be negative
- This semaphore *s* can only be 1 or 0 because of the behavior of the program
  - therefore, other terminology is sometimes used, e.g. *lock() / unlock()*

```
P_X = while true do
        StateX := 1;
        StateX := 2;
        P(s);
        StateX := 3; StateX := 4;
        V(s);
        StateX := 5
      od
```

critical section

||

```
P_Y = while true do
        StateY := 1;
        StateY := 2;
        P(s);
        StateY := 3; StateY := 4;
        V(s);
        StateY := 5
      od
```

POSIX: mutex (1003.1c)

- Special, two-state (i.e., 1 / 0) semaphore: *mutex*
  - between threads
  - specifically for mutual *ex*clusion
- Restrictions
  - don't use copies of a mutex in the calls below
  - *lock()* and *unlock()* always by same thread ("ownership")

P(m)

```
pthread mutex_t m = PTHREAD_MUTEX_INITIALIZER;
                                  /* static initialization, not always possible */
status = pthread_mutex_init (&m, attr); /* attr: NULL; should return 0     */
status = pthread_mutex_destroy (&m);  /* should return 0                   */
status = pthread_mutex_lock (&m);     /* should return 0                   */
status = pthread_mutex_trylock (&m);  /* returns EBUSY if m is locked      */
status = pthread_mutex_unlock (&m);   /* should return 0                   */
```

V(m)

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
  - **sufficient 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
- **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). An assertion must not be disturbed by another process.
- **Minimal waiting**: ("make progress")
  - 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**:
  - 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.

- **Notice**: the last four are new aspects, coming from the concurrency

Reasoning about parallel programs

- Annotation:
  - *assertions* at *control points*
- Assertion:
  - predicate (boolean function) in terms of the program variables
    - e.g. $x=y$, $x \geq 0$
- Control point:
  - place in the program text where one might inspect the current state (i.e., values of variables) – between atomic actions
  - **Meaning**: "*when the program is started in a state that satisfies the initial assertion then, when a control point is reached during execution, the corresponding assertion holds*"
- Invariant:
  - special assertion that holds at every control point
- Traces
  - all interleavings

Example: maintaining an invariant

- In order to maintain an invariant, places in the program have to be found where it may be disturbed
- These places must be guarded, i.e., we must make sure that at such a place an assertion holds strong enough to not disturb the invariant
- This usually leads to (blocking) synchronization and communication
- **Examples**
  - **Invariant**: A datastructure (more general: resource) is accessed only by a single thread at a time
    - "mutual exclusion"
    - usually, to remove unwanted interference
  - **Assertion** (before accessing): no other thread is accessing the resource
  - **Invariant**: either *x* or *y* must be positive
  - **Corresponding assertions:** suppose there is a statement *x := x-10*, then before this statement, "*y>0* or *x > 10*" must hold

Exercises

I.1 Consider the third solution for the mutual exclusion problem (page 20). Does the addition of the assignment *t := Y* (resp. *t := X*) after crossing the bridge solve the problem? Discuss this.

I.2 Which variables are shared/private in this program fragment?

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

||

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

Exercises

I.3 Consider the program in I.2. Given that the initial value of *x* is 0, what will the final value be? (See also exercise **I.5**)

I.4 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 traces.

Exercises

I.5 In most computers, an assignment like *x := x+1* is not an atomic action. It is usually executed through copying via an internal register. For exercise I.2, the result looks like the program below.

```
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.6 The order of the statements in Peterson's algorithm is of crucial importance. Show that the program is wrong if we interchange in *X* the assignments to *t* and to *bX* (and in *Y* the assignments to *t* and b*Y*) by giving a counter example (a partial trace). Is the algorithm fair?

I.7 [not easy, much work, but brings insight] Notice that in Peterson's algorithm the test is not atomic according to the single assignment rule. Replace it with a sequence of statements that is indeed atomic and try to complete the annotation. (You may need to introduce new private variables, but now ones that indeed do exist and are not only for the proof.) Notice that this can be regarded as a theorem, describing the implementation ("refinement") of such a test.

I.8 Determine the number of traces corresponding to a single iteration of the programs on page 10. What if we interchange the order of *y:=E; x:=true* into *x:=true;y:=E ?* (Given that the purpose of the program is to 'catch' updates of *y,* do we get 'wrong' behavior?)

Exercises

I.9 Show correctness of Peterson's algorithm. Show

1. that the annotation on page 23 is correct;
2. that no assignment in *Y* can falsify this annotation.

Using this annotation, show

1. that no two cars can be on the bridge;
2. that no deadlock occurs;
3. (argue) fairness

Alternatively, show correctness by investigating the possible traces.

I.10 What happens if we use busy waiting (e.g. Peterson's algorithm) to synchronize two threads in a multithreaded program?

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);
}
```

- Is statement '*x=x+1*' atomic? Motivate your answer.
- Replace it with a sequence that can be regarded as atomic.
- What are possible final values of *x?*