



T-106.420
Concurrent Programming
Atomic actions and synchronization



states, atomic actions & histories

- *atomic action*: an indivisible operation a on process state:
 $s_i \Rightarrow a \Rightarrow s_o$ only the initial and final state, s_i and s_o are visible to outside
- a process *history* (=trace) is a sequence of atomic actions a_i and states s_i :

$$s_0 \Rightarrow a_1 \Rightarrow s_1 \Rightarrow a_2 \Rightarrow s_2 \Rightarrow a_3 \Rightarrow \dots$$

- a history of a concurrent system: an *interleaving* of process traces

process a: $s_0 \Rightarrow a_1 \Rightarrow s_1 \Rightarrow a_2 \Rightarrow s_2 \Rightarrow a_3 \Rightarrow \dots$

process b: $t_0 \Rightarrow b_1 \Rightarrow t_1 \Rightarrow b_2 \Rightarrow t_2 \Rightarrow b_3 \Rightarrow \dots$

a and b: $s_0 \& t_0 \Rightarrow a_1 \Rightarrow s_1 \& t_0 \Rightarrow a_2 \Rightarrow s_2 \& t_0 \Rightarrow b_1 \Rightarrow s_2 \& t_1 \Rightarrow a_3 \Rightarrow s_3 \& t_1 \dots$

- the number of possible traces of n processes of m actions:
 - permutation of $m * n$ actions: $(n*m)! / (m!^n)$ # this grows very fast!!!



basic (von Neuman) machine model

- basic types stored as *memory words*, eg. 8,16, 32, 64 bits
 - words are read & written as atomic actions *load* and *store*
- each process has a *private* register set, i.e. processor state
 - either real or virtual provided by HW or/and multitasking OS
- process' intermediate results are stored in its *private* space:
 - processor registers
 - runtime stack
- Example:
 - assignment "x = y + z" on a high language level is compiled to:
 - *load* values of y and z to processor registers (private space)
 - *evaluate* the expression "y+z" in the processor (private space)
 - *store* results back "x = ..." to memory (either shared or private)



recap: atomic actions & shared state

- *atomic action* examines and/or changes the state invisibly :

$$s_i \Rightarrow a \Rightarrow s_o$$

- only the initial and final state, s_i and s_o are visible to outside, i.e. could be shared by other processes
- primitive (fine grained) atomic actions provided directly by hardware:
 - e.g. *load* and *store* one *memory word* to/from a *processor register*
 - Processor state is *private space*, memory can be private (e.g. runtime stack) or *shared* (e.g. global variables)
- larger (more abstract) atomic actions, eg. critical sections can be built only based on smaller (finer grained) ones.



atomic actions & shared state

- e.g. assignment statement of a programming language *appears* to be atomic in sequential programs, but not in generally for concurrent programs !

```

• int x=0, y=0, z=0;
  co x=y+z;      #process p1
  // y=1;z=2;   #process p2
  oc;

```

At the final state x can be: 0, 1, 3, but also: **2** (This is a non-atomic result in the sense that it could not result from the program if the assignments would be atomic !)

- possible history:

| - | x | y | z | y+z (private for p1) | process | action |
|---|---|---|---|----------------------|---------|---------------------|
| - | 0 | 0 | 0 | _ + _ | p1 | load y |
| - | 0 | 0 | 0 | 0 + _ | p2 | store 1 to y |
| - | 0 | 1 | 0 | 0 + _ | p2 | store 2 to z |
| - | 0 | 1 | 2 | 0 + _ | p1 | load z |
| - | 0 | 1 | 2 | 0 + 2 | p1 | evaluate (y + z) |
| - | 0 | 1 | 2 | 2 | p1 | store (y + z) to x |
| - | 2 | 1 | 2 | 2 | | |



atomicity (disjointness) requirements

- *disjoint expression e* : if e in one process *doesn't refer a variable written by another process*, expression evaluation will appear as atomic, because
 - no values on which e depends can change during evaluation and
 - no other process can see private and/or temporary variables of e
- *disjoint assignment $x = e$* : if e is disjoint
- Disjoint means in practice that processes are disjoint. (eg. matrix multiplication example before)



at-most-once property of an expression

- in a process a reference to a basic variable changed by another process is called *critical*
- an expression e in p satisfies the at-most-once property if:
 - e contains *at most one critical reference*
- evaluation of e can still be considered as one *atomic (read)* action outside of p .



at-most-once property and the atomic assignment statements

- $\mathbf{x} = \mathbf{e}$ assignment statement can be considered as one *atomic (read)action* if:
 - \mathbf{e} is *at-most-once* and \mathbf{x} is not read by others
- $\mathbf{x} = \mathbf{e}$ assignment statement can be considered as one *atomic (write)action* if:
 - \mathbf{e} contains no critical references and \mathbf{x} may be read others
- $\mathbf{x} = \mathbf{e}$ assignment statement can not be an atomic read and write action at the same time.



at-most-once property examples (1)

- no critical references, independent processes:
 - `int x=0, y=0;`
 - `co x=x+1; // y=y+1; oc;`
 - final values: `x=1, y=1`
- critical reference, at-most-once property satisfied:
 - `int x=0, y=0;`
 - `co x=y+1; // y=y+1; oc;`
 - first process
 - has critical reference to `y`, but
 - `x` is not read by the 2nd process
 - 2nd process has no critical reference
 - final values: `x=1 or 2, y=1`



at-most-once property examples (2)

- critical references, at-most-once property not satisfied
 - `int x=0, y=0;`
`co x=y+1; // y=x+1; oc;`
 - both processes have assignment statement, which isn't at-most-once
 - has critical reference, and
 - **assign to** variable **read by** the **other** process
 - possible final values:
 - `x=1, y=2`
 - `x=2, y=1`
 - `x=1, y=1`

This last is a non-atomic result! It could not happen if assignments were atomic.
- Mind you: We don't claim the result is "wrong", it's just not the same as if the assignments were atomic.



Critical section: How to make *a sequence* of atomic actions *atomic* ?

- Need to combine several statements as a single, *coarse-grained* atomic action:
An indivisible sequence of statements, ie. a mutually excluded *critical section*.
- Eg. if an expression or assignment statement doesn't meet the at-most-once property and atomicity is still needed.



two requirements

- *Atomicity*: Create longer, ie. coarse-grained atomic actions, sc. ***critical sections*** by *mutually excluding* their execution in time.
- *Synchronization*: delay process execution till the *shared state* satisfies a *predicate*, ***synchronization condition***.



await statement

- notation: angle brackets $\langle \quad \rangle$
- await statement $\langle \mathbf{await}(\mathbf{B}) \mathbf{S}; \rangle$
 - **B** delay condition
 - **S** sequence of sequential statements
- is executed as one atomic action:
 - **B** is guaranteed to be true when execution of **S** begins
 - the internal state in **S** is invisible to other processes and can not be changed during its execution by them.



await special cases: mutual exclusion

- Unconditional, i.e. $\mathbf{B} = \text{true}$,
 - mutual exclusion, i.e. longer atomic action $\langle \mathbf{S}; \rangle$
 - $\langle x=x+1; y=y+1; \rangle$ outside world cannot see state where x has been incremented, but y has not.
- $\langle \mathbf{S}; \rangle$ has the same effect as \mathbf{S} if
 - \mathbf{S} is single assignment and is at-most-once
 - or atomic machine instruction



await special cases: condition synchronization

condition synchronization `<await (B);>`

- `<await (count > 0);>`
 - process execution will be delayed until **count > 0**
 - no guarantee that “count > 0” will stay after the “`>`”
- if expression **B** is at-most-once `<await (B);>` can be implemented as
while (not B);
- this is a s.c. “*spin lock*” or “*busy wait*”:
 - process does not give up the processor “voluntarily” while waiting
 - in a multiprocessor system this is often ok and needed
 - in a multitasking uniprocessor system the process must be *pre-empted* in order not to monopolize the processor and cause possible a deadlock
 - pre-emption is done by setting a clock to interrupt the system periodically causing the OS-kernel to take control and reschedule the processes.



await example: semaphore

- Semaphore: an integer variable to be used as a “synchronization counter”, with the wait-condition to be non-negative:
 - $\langle \mathbf{await} (s > 0) \ s = s - 1; \rangle$ # a P(s) operation on s
 - $\langle s = s + 1; \rangle$ # a V(s) operation on s
-
- We will return to semaphores later



summary: atomic actions

- unconditional
 - HW-level: machine instructions (fine-grained)
 - At-most-once S;
 - $\langle S; \rangle$ same as: $\langle \mathbf{await} \text{ (true) } S \rangle$;
- conditional
 - $\langle \mathbf{await} \text{ (B) } S; \rangle$
 - B becomes true as an action of another process.
It can take an arbitrarily long time.

Mind you: We have not yet discussed of how to implement **await**. It's not trivial.



Example: producer/consumer synchronization (1/3)

- **one element buffer: buf**
- ```
int buf, p=0, c=0;
process Producer {
 int a[n];
 while (p<n) {
 <await (p==c); #buffer empty
 buf = a[p];
 p=p+1; }
 }
}
process Consumer {
 int b[n];
 while (c<n) {
 <await (p>c); #buffer full
 b[c]=buf;
 c=c+1; }
 }
}
```

Note:  $\langle \text{await} \rangle$  is used here both for synchronization and critical section creation.

*We can prove that the processes preserve outside the  $\langle \text{await} \dots \rangle$ -statements the following consistency criteria  $I$  for the global state:*

*( $c \leq p \leq c+1$ ) and  
( $p = c \iff b[c]=a[p]$ ) and  
( $p > c \iff p=c+1$  and  $buff = a[p]$ ).*

*From this we can prove:*

*( $p = c \implies$  (for all  $i: i < p: (b[i] = a[i])$ ) )  
so that especially at the termination:*

*( $p = c = n \implies$   
(for all  $i: i < n: (b[i] = a[i])$  ).*



# Example: producer/consumer synchronization (2/3)

- **one element buffer: buf**
- ```
int buf, p=0, c=0;
process Producer {
    int a[n];
    while (p<n) {
        <await (p==c); #buffer empty
        buf = a[p];
        p=p+1;
    }
}
process Consumer {
    int b[n];
    while (c<n) {
        <await (p>c); #buffer full
        b[c]=buf;
        c=c+1;
    }
}
```

Note: We can take the assignments out from the critical sections, because they are mutually excluded in time anyway. So, **<await >** is used here only for synchronization.

Consistency requirement I is weaker:

($c \leq p \leq c+1$) and

($p = c \Rightarrow b[c] = a[p]$) and

($p > c \Rightarrow p = c+1$ and $buff = a[p]$).

Here we have applied the following:

"Rule #1 for shared datastructures: Ensure the consistency of data before giving access to it."

We can still prove that:

($p = c \Rightarrow$ for all $i: i < p: (b[i] = a[i])$).



Example: producer/consumer synchronization (3/3)

- **one element buffer: buf**
- ```
int buf, p=0, c=0;
process Producer {
 int a[n];
 while (p<n) {
 while not (p==c); #buffer empty
 buf = a[p];
 p=p+1;
 }
}
process Consumer {
 int b[n];
 while (c<n) {
 while not (p>c); #buffer full
 b[c]=buf;
 c=c+1;
 }
}
```

Note: Because the synchronisation conditions ( $p==c$ ) and ( $p>c$ ) are atomic (at-most-once), we can evaluate them in a while loop. (In a uniprocessor this kind of busy wait would not make too much sense.)

*Consistency requirement I still holds:*  
( $c \leq p \leq c+1$ ) and  
( $p = c \Rightarrow b[c]=a[p]$ ) and  
( $p > c \Rightarrow p=c+1$  and  $buff = a[p]$ ).

*and also:*  
( $p = c \Rightarrow$   
(for all  $i: i < p: (b[i] = a[i])$ )  
( $p = c = n \Rightarrow$   
(for all  $i: i < n: (b[i] = a[i])$ ))