CONCURRENT PROGRAMMING CONCEPTS

- Concurrent programming is the simultaneous execution of two or more programs on one or more processors.

- One *abstraction* assumes *interleaved* execution of atomic instructions of multiple processes.

- We will assume that each process is executing in its own processor even though there may be only one processor in reality.

- We have to consider possible interactions among such processes in two cases:

  a. **Contention** – When two processes compete for the same *resource*. This may be access to some computing resource like a *shared variable* or access to a common printer etc.

  b. **Communication** – Two processes may need to communicate to agree upon certain events. This communication may be through checking of a shared variable or though explicit message passing.

Atomic Instructions

- We ignore the difference of time for different atomic instructions. We assume that irrespective of the atomic instruction and irrespective of the CPU speed, each atomic instruction takes *unit* time to complete.

- By *atomic instruction*, we mean that once a process starts executing such an instruction, it cannot be interrupted before the instruction is complete. However, an atomic instruction should be completed within a *reasonable* amount of time.

- In other words, we ignore the *absolute time requirements* for a process. Rather, we concentrate on the execution *sequence* of atomic instructions.

- All execution sequences of all the processes are *interleaved* into a single execution sequence.

Instruction Interleaving

- For example, consider two atomic instructions I_1 and I_2 getting executed by two processes P_1 and P_2.

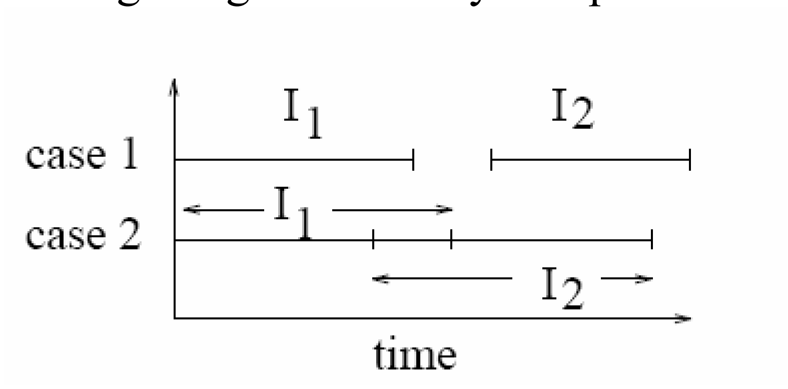


Figure 1: Instruction interleaving.

- In **case 1**, *end* (I_1) $\leq$ *begin* (I_1). So, the instructions are already interleaved in this case.

- In **case 2**, *begin* (I_2) < *end* (I_1). This is quite possible since these two instructions are getting executed in two different processors. However, we do not allow such an ordering or instructions.

- In our abstraction, the effect of two simultaneous instructions is the same as either of the two sequences when the instructions are executed one after another.

Concurrent Programming Abstraction

- This abstraction is not absurd since in most cases, hardware resolves contention by assigning arbitrary orders to simultaneous requests.

- Note that, in a distributed system, there are always instructions which are getting executed simultaneously. However, we still impose a total order on the instruction sequence.

- Such an arbitrary ordering of instructions does not have any effect on the computation unless either there is *contention* or two processes are trying to communicate.

- In case of contention (say, over a shared variable), we assume that the underlying hardware resolves the contention. In other words, the hardware imposes an ordering.

- In case of communication, the underlying protocol imposes an ordering by delivering the messages in some order.

Concurrent Program Correctness

- Note that, under this model, arbitrary interleaving of instructions is possible. Depending on processor speeds, hundreds of instructions from one process may get executed before an instruction of the other process is executed.

- **A concurrent program is required to be correct under all interleavings.**

- So, the correctness of a concurrent program should not depend on *processor speeds, communication time* etc.

- However, one of the important criteria is: **fairness** should be preserved. Eventually, instructions from each process should be present in the interleaved sequence.

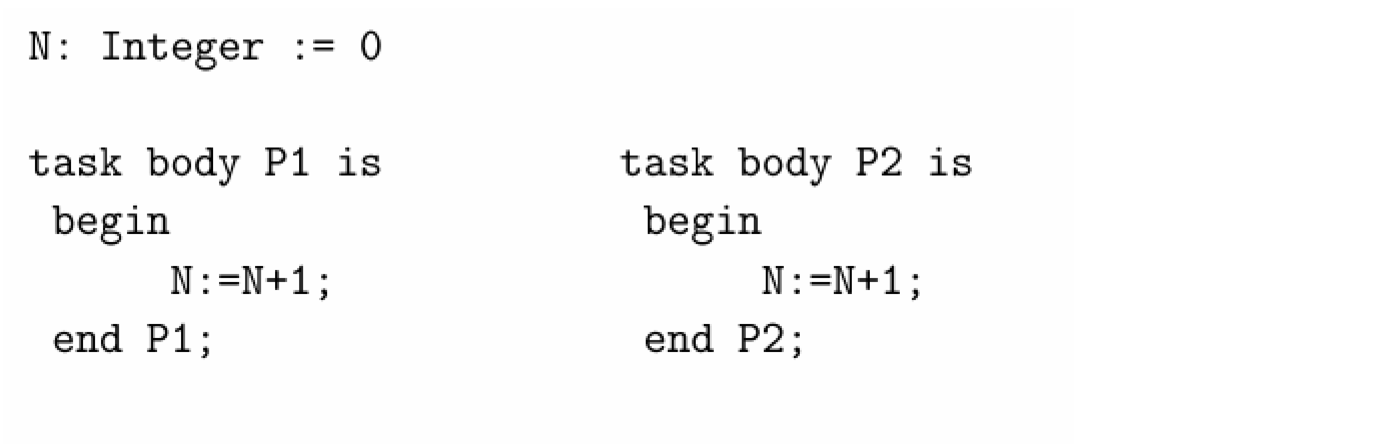
- **Showing a concurrent program incorrect:** Show that there is at least one instruction interleaving for which the program is correct.

- **Proving a concurrent program correct:** Show that for all possible interleavings, the program is correct.

Atomic Instructions

- The nature of atomic instructions is crucial for correctly defining and analysing concurrent programs.

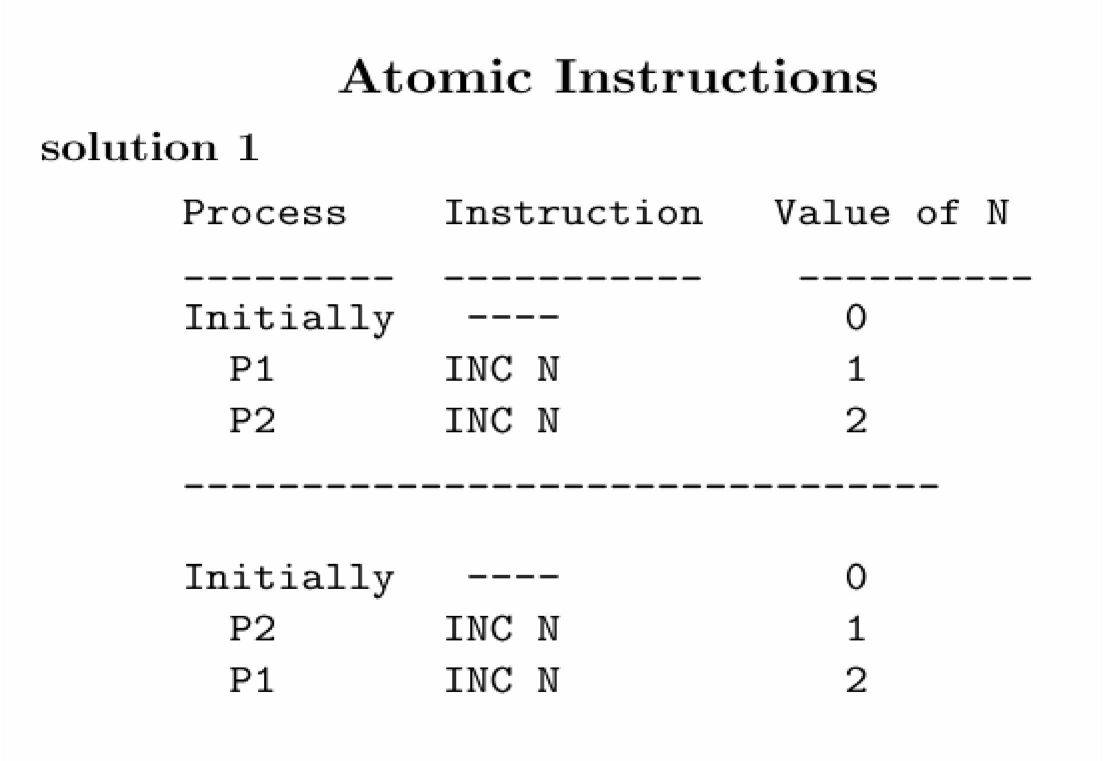
- Consider the following example:



```
N: Integer := 0

task body P1 is              task body P2 is
 begin                        begin
      N:=N+1;                      N:=N+1;
 end P1;                      end P2;
```

- *Compile* translates to code which uses an INC instruction; *any* interleaving gives same value.

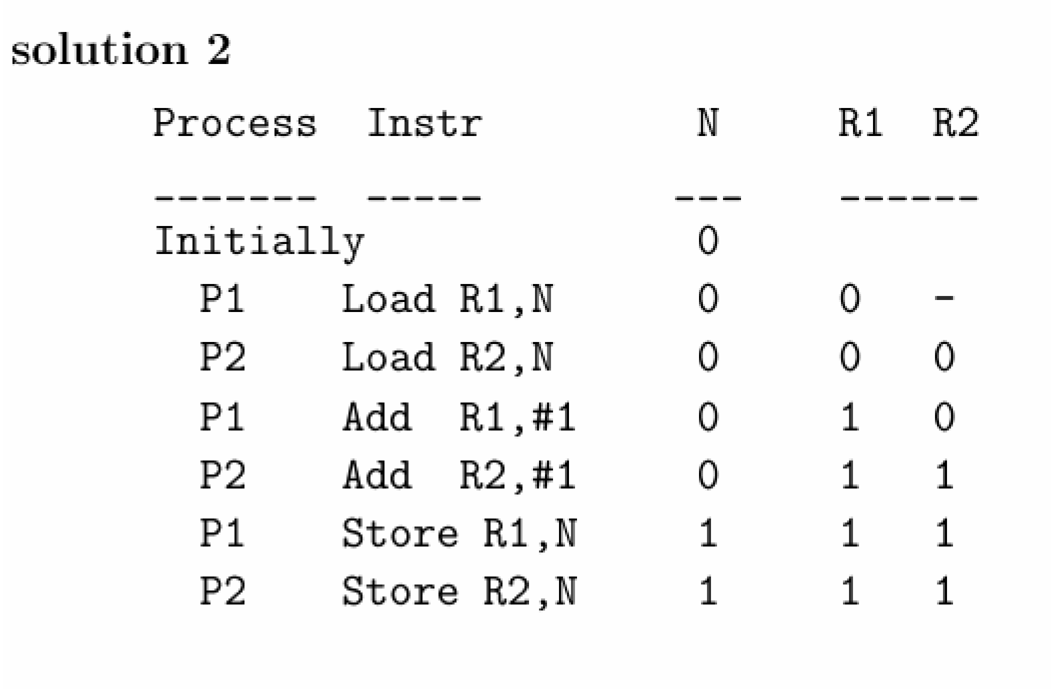


Atomic Instructions

solution 1

Process	Instruction	Value of N
Initially	----	0
P1	INC N	1
P2	INC N	2
Initially	----	0
P2	INC N	1
P1	INC N	2

- If compiler translates to machine code involving *Load, Add* and *Store* instructions *some* interleavings give incorrect answer.



solution 2

Process	Instr	N	R1	R2
Initially		0		
P1	Load R1,N	0	0	-
P2	Load R2,N	0	0	0
P1	Add R1,#1	0	1	0
P2	Add R2,#1	0	1	1
P1	Store R1,N	1	1	1
P2	Store R2,N	1	1	1

Atomic Instructions

- If the instruction set is simple, it is more difficult to write *correct* concurrent programs.

- If the instructions are complex (like **INC**), it is difficult to *implement* such instructions as atomic instructions.

- We will initially concentrate on simple instructions like Load-Store to common memory.

Complex instructions like **INC** are implemented by locking the access and manipulation of common variables.

Correctness of Concurrent Programs

- *P(x):* a property of the input variables *x*.
  Q(x,y) is a property of the input variables *x* and output variables *y*.

- Then, for any value *a* of the input variables, the correctness is defined as follows:

- **Partial correctness :** If $P(\overline{a})$ is true and the program terminates when started with $\overline{a}$ as the input variables, then $Q(\overline{a}, \overline{b})$ is true where $\overline{b}$ are the values of the output variables when the program terminates.
  $(P(\overline{a}) \wedge terminates(Prog(\overline{a}\ , \overline{b}))) \supset Q(\overline{a}\ , \overline{b})$

- **Total correctness :** In this case, we require the program to terminate.
  $P(\overline{a}) \supset (terminates(Prog(\overline{a}\ , \overline{b})) \wedge Q(\overline{a}\ , \overline{b}))$

An example : `SQR(x,y)` is a program which computes the square root of **x** and stores in **y**.
$(a \geq 0 \wedge terminates(SQR(a, b))) \supset b = \sqrt{a}$
$a \geq 0 \supset (terminates(SQR(a, b)) \wedge b = \sqrt{a})$

Correctness Properties

There are two kinds of correctness properties:

- **Safety properties:** The property must *always* be true.
- **Liveness property:** The property must *eventually* be true.

Example of safety properties:

- **Mutual exclusion:** Certain instruction sequences should not be interleaved. For example, when a process is writing the name of a file in the printer spool, it should not be interrupted and another process should not overwrite the name of the file.
- **Absence of deadlock:** Two or more processes should not wait forever without doing any useful work. For example, suppose there are two processes in a system and both require a magnetic tape and a printer for completion. There is only one magnetic tape and one printer.

  P1 has taken (sole) access of the magnetic tape and waiting for the printer.
  P2 has taken (sole) access of the printer and waiting for the magnetic tape.

The system is *deadlocked*.

Correctness Properties

Examples of Liveness property: An important example is *fairness*. If there is contention (i.e. competition) for some resource, we should resolve the contention fairly. In other words, no process should be unfairly denied a particular resource for a long period of time.

- **weak fairness:** If a process continuously makes a request for some resource, eventually it will be granted the resource.
- **strong fairness:** If a process makes a request infinitely often, eventually it will be granted the resource.
- **linear waiting:** If a process makes a request, it will be granted the resource before any other process is granted the resource more than once.
- **FIFO:** The processes making the requests are placed in a queue and the requests are granted in that order.

Inductive Proofs

The inductive proof technique will be very useful for proving *safety* properties in concurrent programs. Since many concurrent programs are non-terminating (for example operating system programs), quite often inductive proof is the only way to prove correctness for all possible interleavings.

- **Basis:** Prove the property for the initial state of the program.

- **Inductive step:** (inductive hypothesis) Assume that the property holds after the n^{th} step of the execution sequence.

Prove that it holds after the $(n + 1)^{st}$ step.

We cannot make any assumption on the execution sequence since we do not know in advance how the scheduler will schedule the instructions.

So, we have to prove the inductive step for all possible interleavings.

See textbook for further details if interested.