

# CONCURRENT PROGRAMMING CONCEPTS & NOTATIONS

---

Dr. Emmanuel S. Pilli

# Three Issues

- How to express concurrent execution
- How processes communicate
- How processes synchronize

# Concurrent Programs

- Sequential program specifies sequential execution of a list of statements
- Execution of the program is called a process
- Concurrent program specifies two or more sequential programs that may be executed concurrently as parallel processes

# Concurrent Programs

- Concurrent program can be:
  - Executed either by allowing processes to share one or more processes – multiprogramming
  - Executed by running each process on its own processor
    - multiprocessing if the processors share common memory as in multiprocessor
    - distributed processing if processors are connected by a communication network

# Process Interaction

- Concurrently executing processors must communicate and synchronize in order to cooperate
- Communication allows execution of one process to influence execution of another
- Interprocess communication can be done
  - Use of shared variables
  - By message passing

# Synchronization

- Necessary when processes communicate
- Synchronization is a set of constraints on the ordering of events
- Programmer employs a synchronization mechanism to delay execution of a process in order to satisfy such constraints

# Concurrent Execution

- fork
- join
- cobegin
- call
- resume
- process

# Synchronization Primitives

- Communication uses shared variables
  - Exclusion
  - Condition
- Communication is by message passing
  - RPC
  - Atomic Transactions



# Sync Prim – Shared Variables

- Busy Waiting
- Semaphores
  - Binary
  - Counting
- Conditional Critical Regions
- Monitors

# Sync Prim – Message Passing

- Specifying channels of Communication
  - send
  - receive
  - blocking
  - non blocking
  - asynchronous message
  - buffered
  - send no-wait

# Concurrent versus Parallel

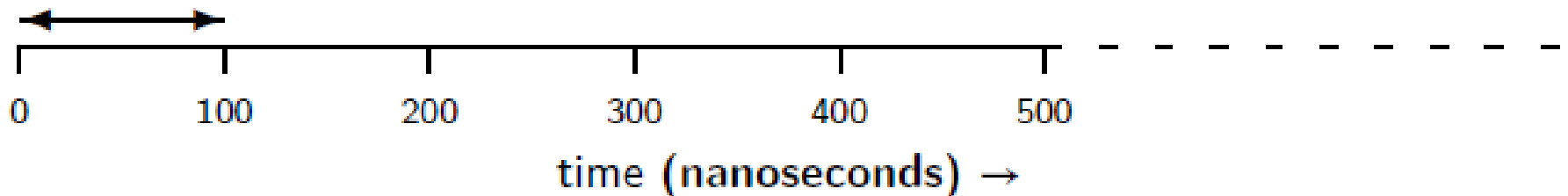
- Concurrent program is a set of sequential programs that can be executed in parallel
- Parallel program is used for systems in which executions of several programs overlap in time by running them on separate processors
- Concurrency can be used to denote potential parallelism – execution may, but need not overlap
- Parallelism may only be apparent since it may be implemented by sharing resources of a small number of processors, often only one

# Concurrent versus Parallel

- Concurrency is a useful abstraction because we can better understand such a program pretending that all processes are being executed in parallel
- Even if the processes on a concurrent program are actually executed in parallel on several processors, understanding its behavior is greatly facilitated if we impose an order on the instructions that is compatible with shared execution on a single processor

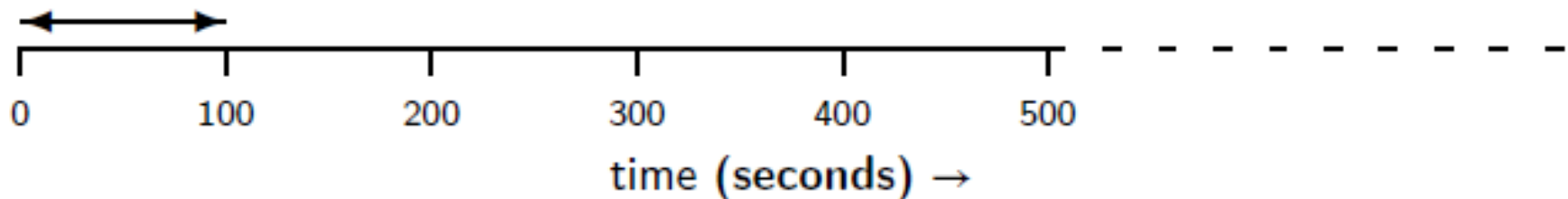
# Concurrency as Abstract Parallelism

- Clock speed of CPU of a PC is of the order of magnitude of one GHz (one billion times a second)
- Every nano second, the clock ticks and the processor performs some operation



# Concurrency as Abstract Parallelism

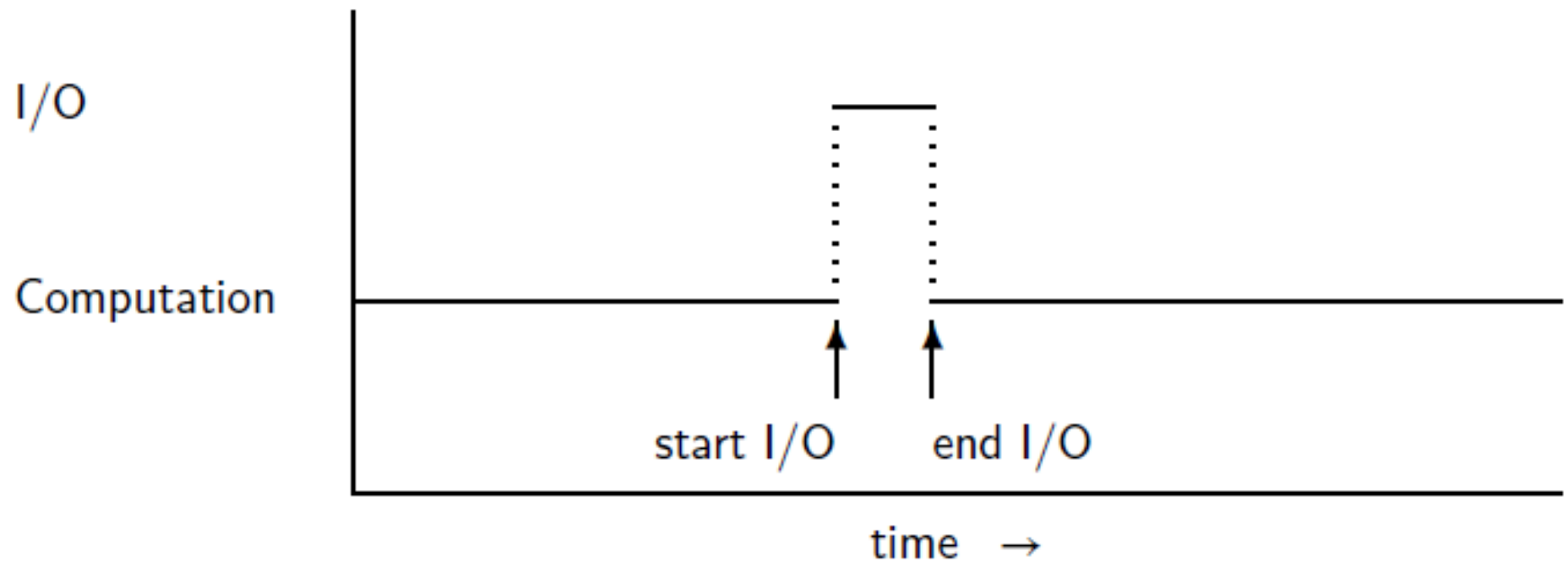
- If we are processing characters by hand, we do not consciously perform operations on the scale of nanoseconds
- Tremendous gap between human and electronic devices led to OS which allow I/O operations in parallel with computation



# Concurrency as Abstract Parallelism

- On a single CPU, processing required for each character typed cannot really be done in parallel
- A fraction of micro second can be stolen from the other computation to process the character
- Degradation is not noticeable even if the overhead of switching between the two computations is included
- I/O processing is performed as the result of an interrupt and is a separate process executed concurrently with a process doing another computation

# Concurrency as Abstract Parallelism





# Multitasking

- Simple generalization of overlapping the computation with that of another computation
- Kernel of all modern OS does *multitasking*
- Scheduler program is run by the OS to determine which process should be allowed to run for the next interval of time
- Time-slicing is implemented where computations are periodically interrupted to allow a fair sharing of resources of CPU

# Multitasking

- Programming languages support *multi threading* - multitasking within programs
- Threads enable the programmer to write concurrent computations within a single program
- Process runs in its own address space managed by the OS
- Thread runs within the address space of a single process which may be managed by a multithreaded kernel within the process

# Challenge of Concurrent Prog

- Need to synchronize the execution of different processes and enable them to communicate

# Role of Abstraction

- Three levels of abstraction
  - Systems and Libraries: OS and libraries – API – define computational resources that are available to the programmer
  - Programming Languages: Enables to employ the computational power of a computer
  - Instruction Sets: Families of CPUs have the same instruction set as seen by the assembly language programmer

# Encapsulation and Concurrency

- Encapsulation achieves abstraction by dividing a software module into a public specification and a hidden implementation
  - Specification describes the available operations on a data structure or real world model
  - Detailed implementation is written within a separate module that is not accessible from the outside
- Changes in the internal data representation can be made without affecting the rest of the system

# Encapsulation and Concurrency

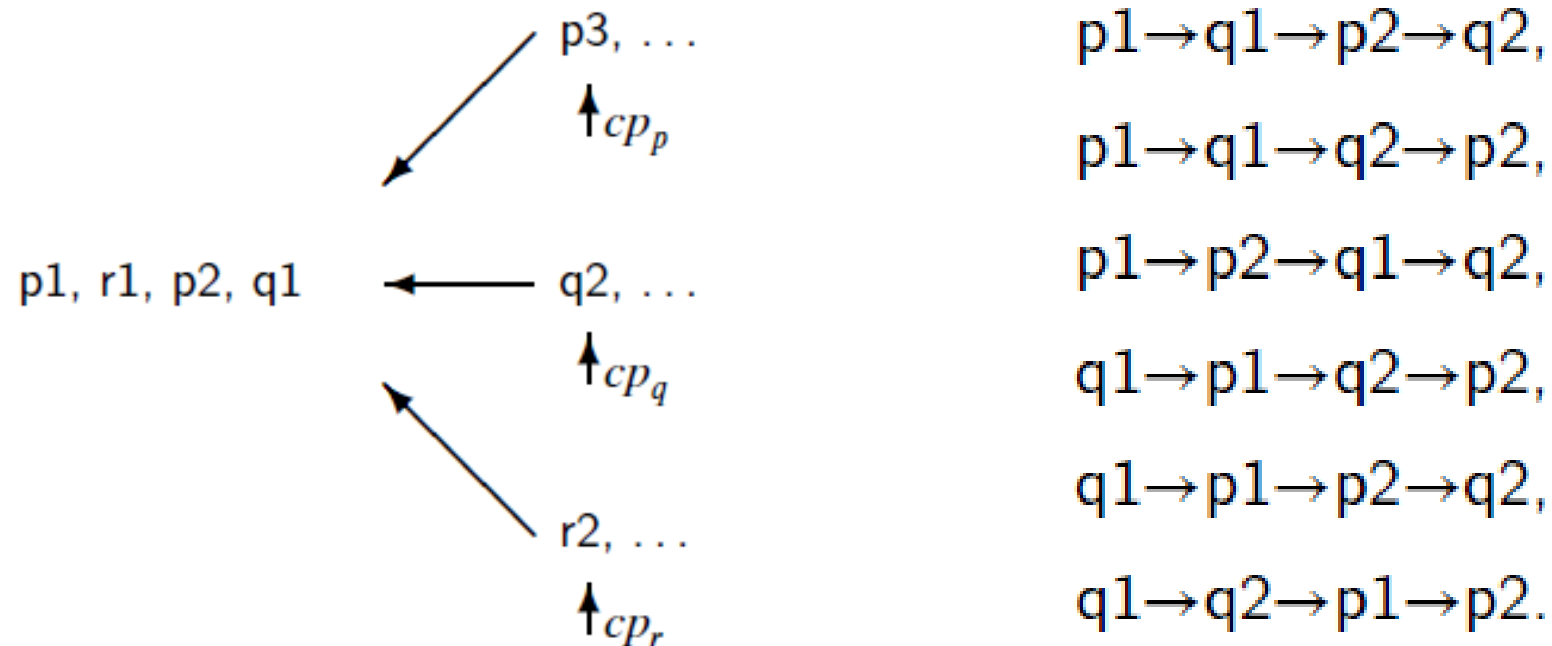
- Concurrency is an abstraction that is designed to make it possible to reason about the dynamic behavior of programs
- Abstraction is based on the concept of a sequential process – a normal program fragment

# Interleaving of Atomic Statements

- *Concurrent program* consists of a finite set of sequential processes
- The processes are written using a finite set of *atomic statements*
- Execution of a concurrent program proceeds by executing a sequence of atomic statements obtained by *arbitrarily interleaving* the atomic statements from the processes
- *Computation* is an execution sequence that can occur as a result of the interleaving

# Interleaving of Atomic Statements

- Computations are also called *scenarios*
- During a computation, the *control pointer* of a process indicates the next statement that can be executed by that process





# Concurrent and Serial Program

## Algorithm 2.1: Trivial concurrent program

integer  $n \leftarrow 0$

**p**

integer  $k1 \leftarrow 1$

p1:  $n \leftarrow k1$

**q**

integer  $k2 \leftarrow 2$

q1:  $n \leftarrow k2$

## Algorithm 2.2: Trivial sequential program

integer  $n \leftarrow 0$

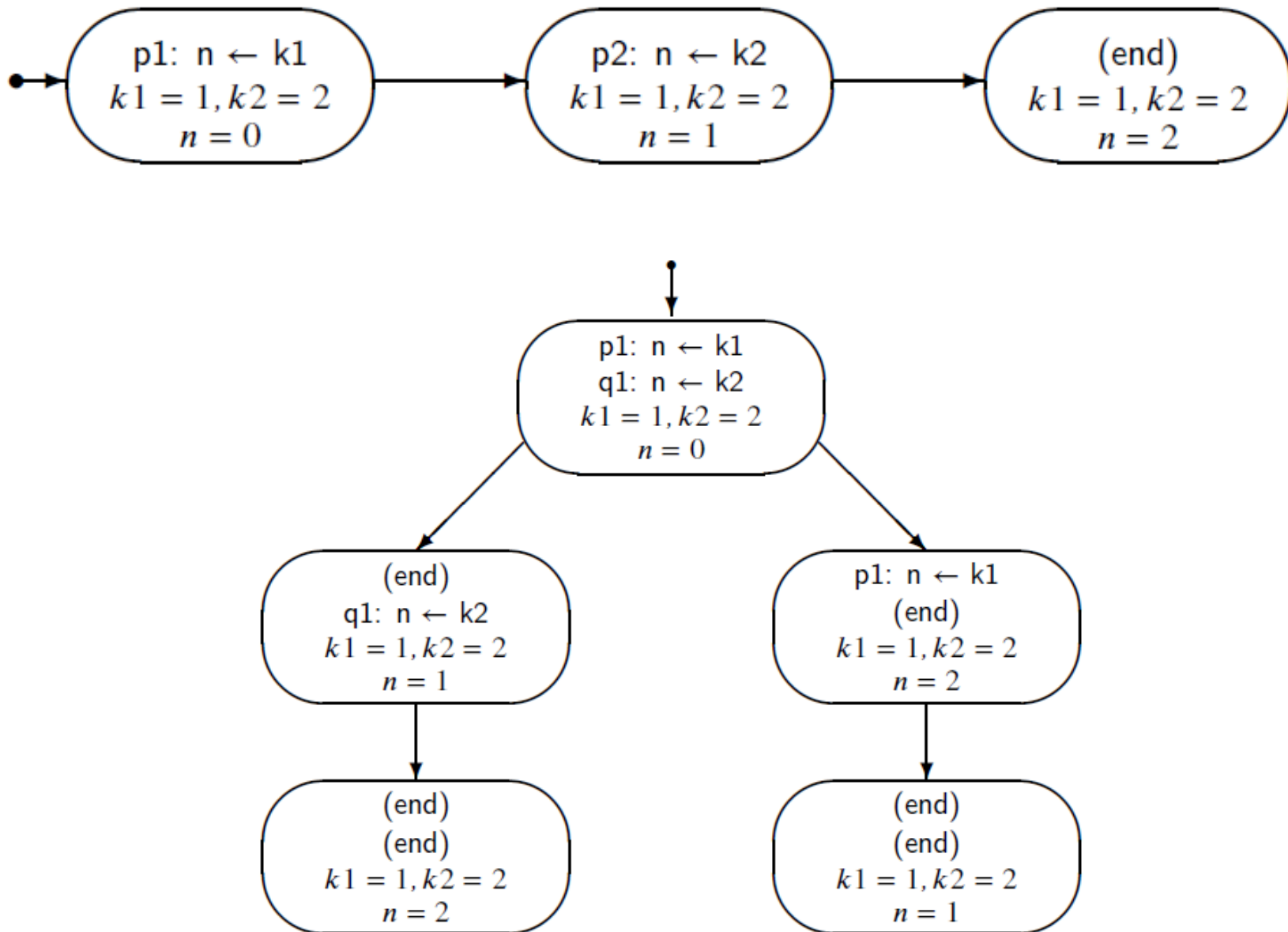
integer  $k1 \leftarrow 1$

integer  $k2 \leftarrow 2$

p1:  $n \leftarrow k1$

p2:  $n \leftarrow k2$

# State Representation



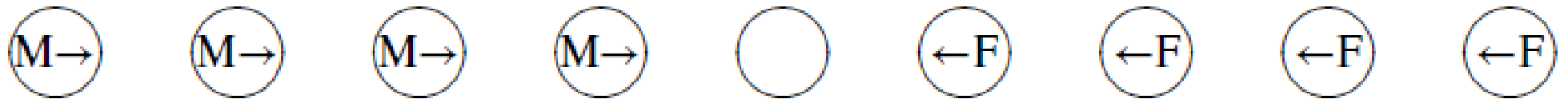
# State and Transition

- The *state* of a concurrent algorithm is a tuple consisting of one element for each process that is a label from that process, and one element for each global or local variable that is a value whose type is the same as type of the variable
- Let  $s_1$  and  $s_2$  be two states, then there is a *transition* between  $s_1$  and  $s_2$  if executing a statement in state  $s_1$  changes the state to  $s_2$

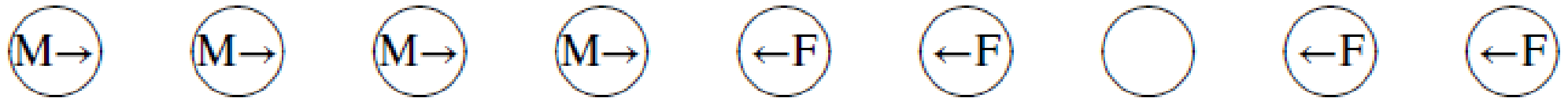
# State Diagram

- A *state diagram* is a graph defined inductively.
- The initial state diagram consists a single node labeled with the initial state.
- If state  $s_1$  labels a node in the state diagram, and if there is a transition from  $s_1$  to  $s_2$ , then there is a node labeled  $s_2$  in the state diagram and a directed edge from  $s_1$  to  $s_2$
- For each state, there is only one node labeled with that state
- Set of reachable states is the set of states in a state diagram

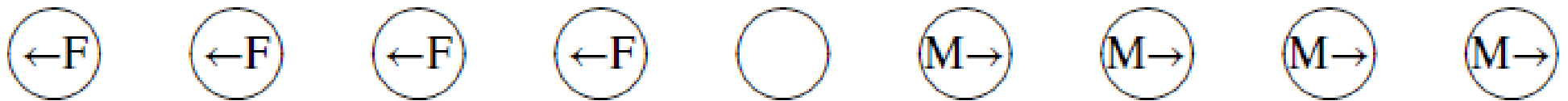
# State Diagram for the Frog Puzzle



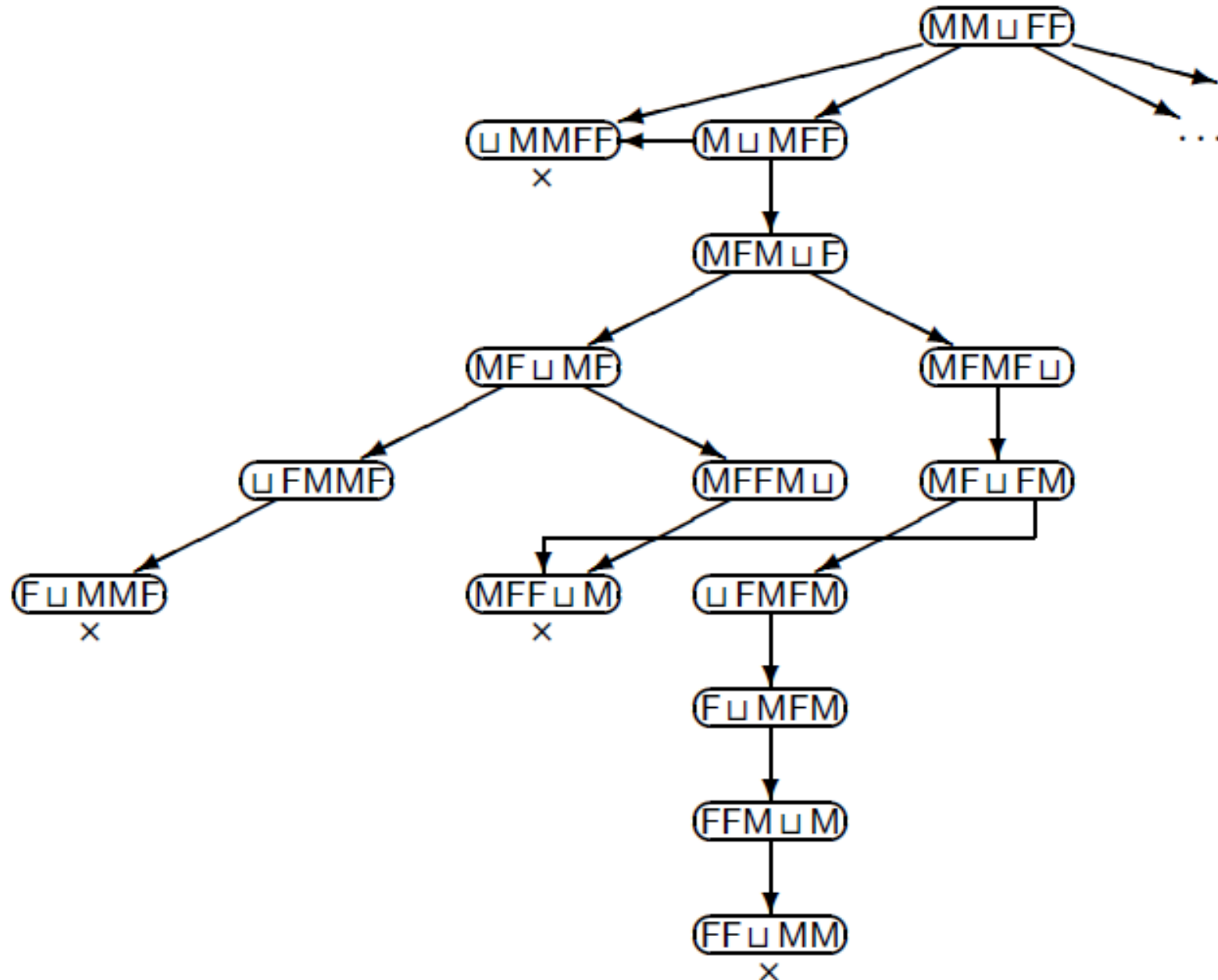
# State Diagram for the Frog Puzzle



# State Diagram for the Frog Puzzle



# State Diagram for the Frog Puzzle





# Scenario

- Scenario is defined by a sequence of states
- Use tables of scenarios instead of diagrams
- List the sequence of states, columns for control pointers are labeled with processes and columns for variable values with the variable names

Process p	Process q	n	k1	k2
<b>p1: n ← k1</b>	q1: n ← k2	0	1	2
(end)	<b>q1: n ← k2</b>	1	1	2
(end)	(end)	2	1	2

# Scenario

- In a state, there may be more than one statement that can be executed
- Bold font is used for denoting the statement that was executed to get to the state in the following row
- Rows represent states
- If a statement executed is an assignment statement, the new value that is assigned to the variable is a component of the next state in the scenario, which is found in the next row.

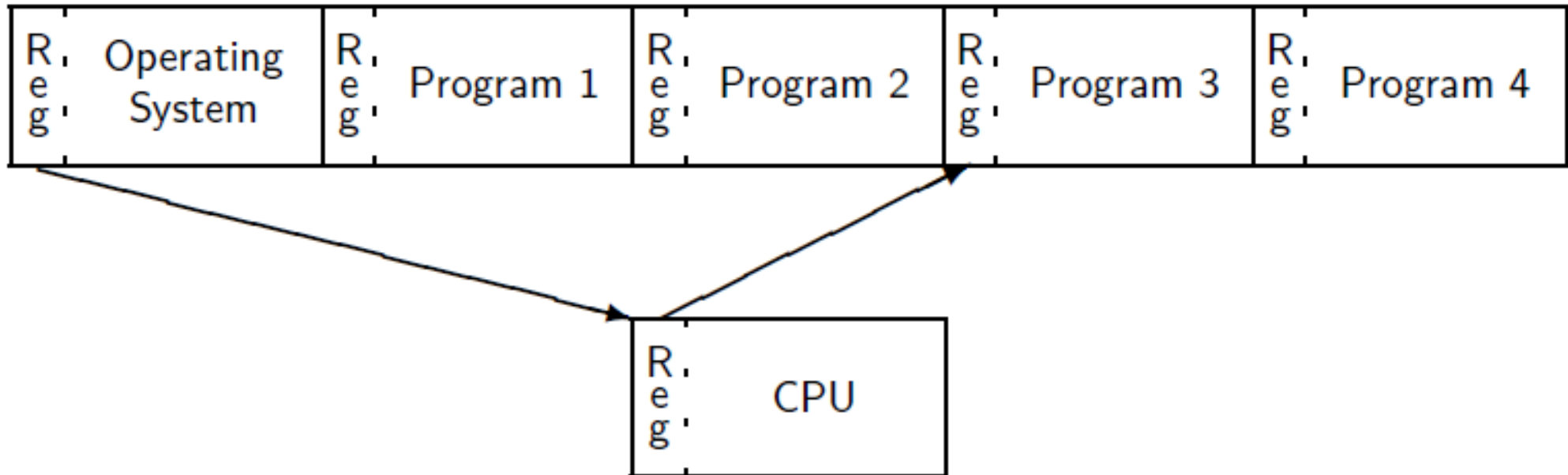
# Justification of the Abstraction

- Coordination between computers cannot be done at the level of individual instructions
- There is not enough time to coordinate individual instructions of more than one CPU
- Execution of a concurrent program is considered to be carried out by a global entity who at each step selects the process from which the next statement will be executed
- The interleaving of processes is arbitrary
- This abstraction is justified for various possible architectures

# Multitasking Systems

- Selection of next instruction to execute is carried out by the CPU and the OS
- Interrupts from I/O devices or interrupt timers will cause execution to be interrupted
- *Interrupt handler* will be executed and upon its completion, the OS scheduler is invoked to select a new process
- This mechanism is called *context switch*

# Multitasking Systems



# Multitasking Systems

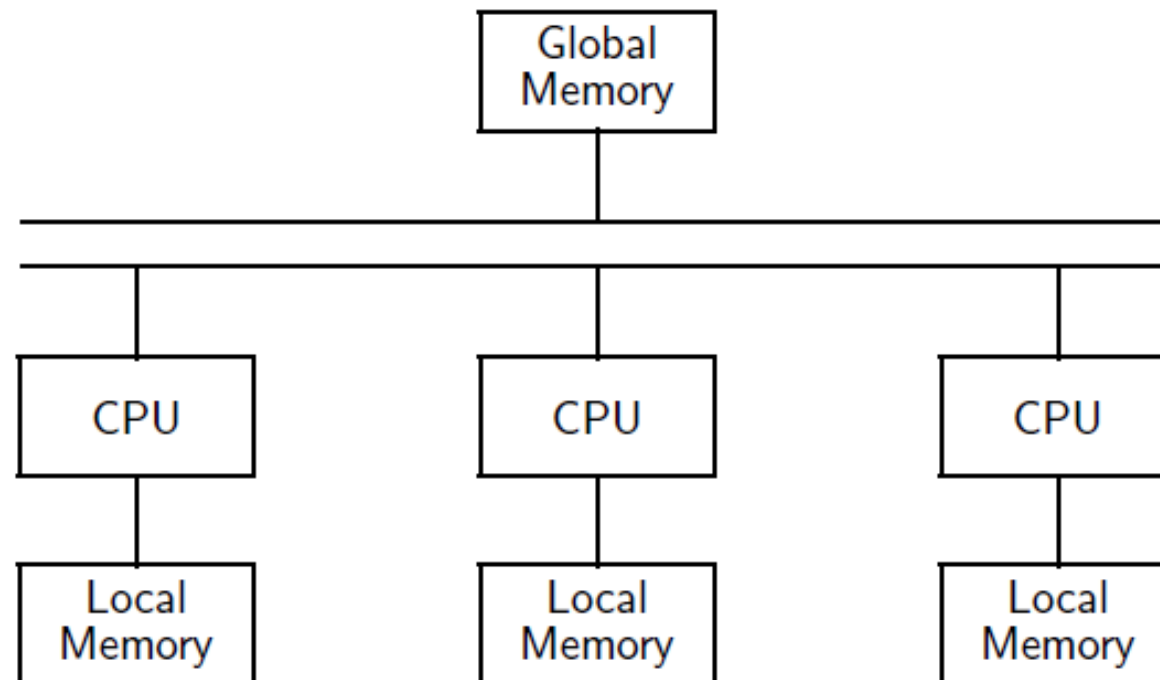
- When execution is interrupted, all the registers (computation, control and others) are saved in a predefined area in the program's memory
- Register contents required to execute the interrupt handler are loaded into the CPU
- After interrupt is processed, the symmetric context switch is performed storing the interrupt handler registers and loading program registers
- OS scheduler decides to perform the context switch with another program and not the one that was interrupted

# Multitasking Systems

- Here, the non intuitive aspect of the abstraction is not the interleaving of atomic statements but the requirement that any arbitrary interleaving is acceptable
- The OS scheduler may be called once in few milliseconds
- Many thousands of instructions may be executed from each process before any instructions are interleaved from another

# Multiprocessor Computers

- Computer with more than one CPU
- Memory is physically divided into banks
- *Local* memory (accessed by one CPU) and *Global* memory (accessed by all CPUs)



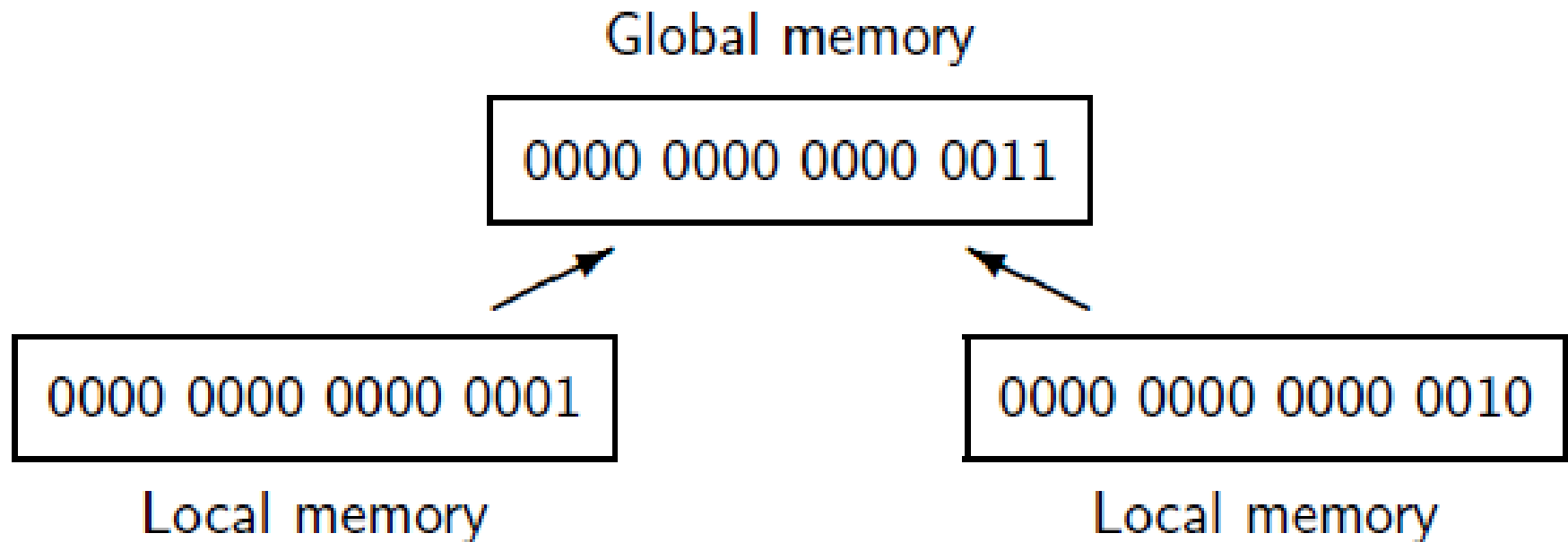


# Multiprocessor Computers

- If sufficient CPUs are present, each process has its own CPU
- Interleaving is no longer real as each CPU is executing instructions independently
- As long as there is no *contention* and two CPUs do not attempt to access the same resource (say global memory)
- Computations defined by interleaving will be indistinguishable from parallel execution
- With contention there is a potential problem

# Multiprocessor Computers

- If two processes try to read or write to a cell simultaneously so that the operations overlap



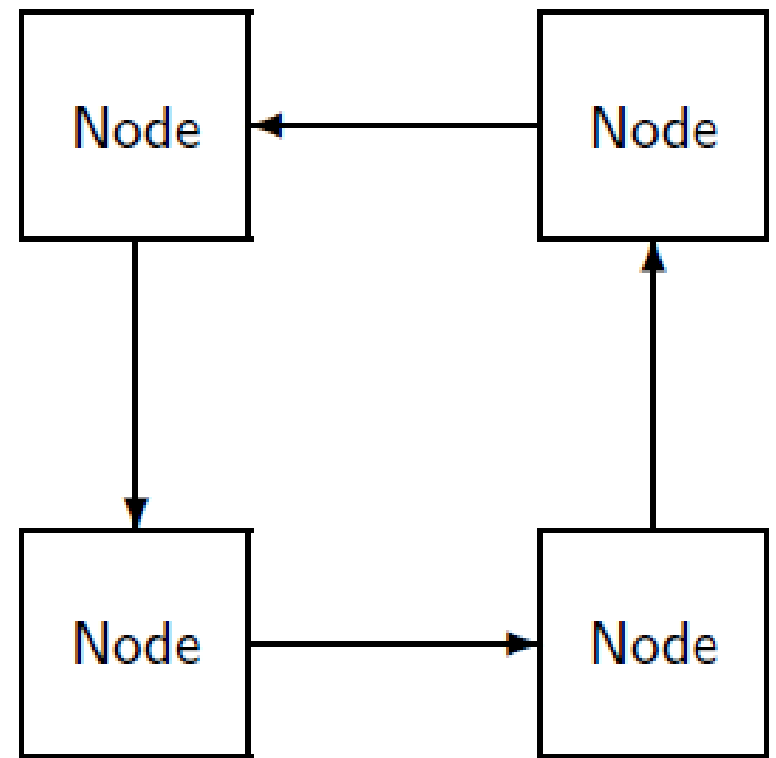
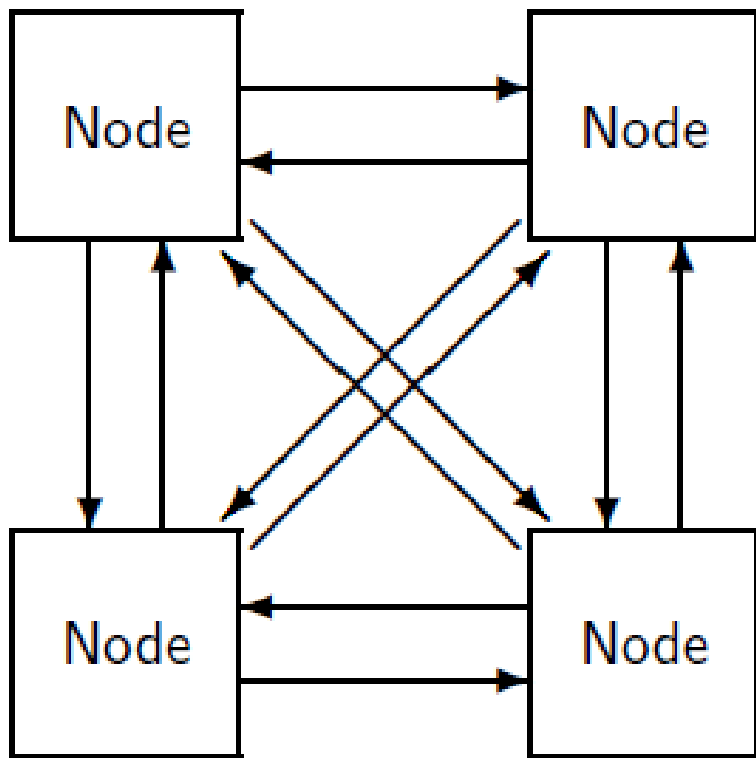
# Multiprocessor Computers

- Interleaving is handled as memory hardware is designed so that one access completes before the other commences
- If two CPUs attempt to read or write the same cell in memory, the result will be same as if the two instructions were executed in either order
- Atomicity and interleaving are performed by the hardware
- Arbitrary interleaving makes more sense as there is no central scheduler, any computation resulting from interleaving may occur

# Distributed Systems

- Distributed system is composed of several computers that have no global resources
- They are connected by communication *channels* enabling them to send messages to each other
- Abstraction of interleaving is not applicable
- Each node is either executing one of its statements, sending a message or receiving a message
- Any interleaving of all the events of all the nodes can be used for reasoning
- Interleaving is consistent with the statement sequences of individual node and with the requirement that message be sent before it is received

# Distributed Systems



# Distributed vs Concurrent

- In concurrent systems, global memory is accessible to all systems
- In distributed systems, nodes may be geographically distant from one another.
- Each node may not send a message *directly* to another node
- We have to consider the *topology* or *connectedness* of the system – fully or ring
- Hardware / software failures catastrophic in CS
- In DS, failures are catastrophic for single nodes, but faulty nodes can be diagnosed and messages can be relayed thro alternate paths

# Atomic Statements

- Concurrent programming abstraction has been defined in terms of interleaving of atomic statements
- An atomic statement is executed to completion without the possibility of interleaving statements from another process
- An important property of atomic statements is that if two are executed simultaneously the result is the same as if they had been executed sequentially (in either order)
- It is important to specify the atomic statements precisely as the correctness of the algorithm depends on the specification

# Atomic Statements

## Algorithm 2.3: Atomic assignment statements

integer  $n \leftarrow 0$

**p**

**p1:**  $n \leftarrow n + 1$

**q**

**q1:**  $n \leftarrow n + 1$

Process p	Process q	n
<b>p1:</b> $n \leftarrow n + 1$	<b>q1:</b> $n \leftarrow n + 1$	0
(end)	<b>q1:</b> $n \leftarrow n + 1$	1
(end)	(end)	2

Process p	Process q	n
<b>p1:</b> $n \leftarrow n + 1$	<b>q1:</b> $n \leftarrow n + 1$	0
<b>p1:</b> $n \leftarrow n + 1$	(end)	1
(end)	(end)	2



# Atomic Statements with Global Reference

**Algorithm 2.4: Assignment statements with one global reference**

integer  $n \leftarrow 0$

**p**

**q**

integer temp

p1:  $\text{temp} \leftarrow n$

p2:  $n \leftarrow \text{temp} + 1$

integer temp

q1:  $\text{temp} \leftarrow n$

q2:  $n \leftarrow \text{temp} + 1$

Process p	Process q	n	p.temp	q.temp
<b>p1: temp←n</b>	q1: temp←n	0	?	?
<b>p2: n←temp+1</b>	q1: temp←n	0	0	?
(end)	<b>q1: temp←n</b>	1	0	?
(end)	<b>q2: n←temp+1</b>	1	0	1
(end)	(end)	2	0	1

# Atomic Statements with Global Reference

**Algorithm 2.4: Assignment statements with one global reference**

integer  $n \leftarrow 0$

**p**

**q**

integer temp

integer temp

p1:  $\text{temp} \leftarrow n$

q1:  $\text{temp} \leftarrow n$

p2:  $n \leftarrow \text{temp} + 1$

q2:  $n \leftarrow \text{temp} + 1$

Process p	Process q	n	p.temp	q.temp
<b>p1: temp←n</b>	q1: temp←n	0	?	?
p2: n←temp+1	<b>q1: temp←n</b>	0	0	?
<b>p2: n←temp+1</b>	q2: n←temp+1	0	0	0
(end)	<b>q2: n←temp+1</b>	1	0	0
(end)	(end)	1	0	0

# Correctness

- In sequential program, running a program with the same input will always give the same result
- In a concurrent program, some scenarios may give correct output while others don't
- Correctness of sequential programs is defined in terms of computing a final result
- Correctness of concurrent programs is defined in terms of properties of computation
  - There are two properties – *Safety and Liveness*

# Correctness

- ***Safety***. The property must always be true
  - For a safety property  $P$  to hold, it must be true that in every state of every computation,  $P$  is true
- ***Liveness***. The property must eventually become true
  - For a liveness property  $P$  to hold, it must be true that in every computation, there is some state in which  $P$  is true

# Correctness

- The challenge of concurrent programs is to ensure that they do useful things, satisfying liveness properties – without violating safety properties
- Safety and liveness are duals of each other – negation of safety property is liveness and v. v
- Safety property will be true if and only if the Liveness property will be false
- Linear Temporal Logic can be used (BTL, CTL) can be used to demonstrate correctness

# Fairness

- A scenario is (weakly) *fair* if at any state in the scenario, a statement that is continually enabled eventually appears in the scenario
- Does this program halt?

Algorithm 2.5: Stop the loop A	
integer $n \leftarrow 0$ boolean $\text{flag} \leftarrow \text{false}$	
p	q
p1: while $\text{flag} = \text{false}$ p2: $n \leftarrow 1 - n$	q1: $\text{flag} \leftarrow \text{true}$ q2:

# Example

## Algorithm 2.9: Concurrent counting algorithm

integer  $n \leftarrow 0$

**p**

**q**

integer temp

integer temp

p1: do 10 times

q1: do 10 times

p2: temp  $\leftarrow$  n

q2: temp  $\leftarrow$  n

p3: n  $\leftarrow$  temp + 1

q3: n  $\leftarrow$  temp + 1

# C Code

```
int n = 0;
```

```
void p() {  
    int temp, i;  
    for (i = 0; i < 10; i++) {  
        temp = n;  
        n = temp + 1;  
    }  
}
```

```
void q() {  
    int temp, i;  
    for (i = 0; i < 10; i++) {  
        temp = n;  
        n = temp + 1;  
    }  
}
```

```
void main() {  
    cobegin { p(); q(); }  
    cout << "The value of n is " << n << "\n";  
}
```



# Java Code

```
class Count extends Thread {  
    static volatile int n = 0;  
  
    public void run() {  
        int temp;  
        for (int i = 0; i < 10; i++) {  
            temp = n;  
            n = temp + 1;  
        }  
    }  
}
```

```
        public static void main(String[] args) {  
            Count p = new Count();  
            Count q = new Count();  
            p.start ();  
            q.start ();  
            try {  
                p.join ();  
                q.join ();  
            }  
            catch (InterruptedException e) { }  
            System.out.println ("The value of n is " + n);  
        }  
    }  
}
```