Engr 664: Theory of Concurrent Programming
Concurrent Programming Introduction, Fall 2016

H. Conrad Cunningham, D.Sc.
Department of Computer and Information Science
The University of Mississippi

22 August 2016

Concurrent Programming Introduction

Motivation

We can model the execution of a (sequential) program as a sequence of *atomic operations* from the program. Each operation atomically changes the *state* (i.e., the values of the variables) of the program.

We can model the set of possible *asynchronous* parallel executions of two programs as the set of all interleavings of the sequences of atomic operations from the two programs. A particular parallel execution corresponds to one of the interleavings, the choice of which is *nondeterministic* (i.e., unpredictable by the programmer). The nondeterministic interleaving models the nondeterministic relative speeds of the executions of the two programs.

Note: By interleaving, we mean a merge of the two sequences in which the relative order of elements in each sequence is preserved. Picture the shuffling of two decks of cards into one.

Consider the parallel execution of the following two assignment statements (identified as programs P1 and P2, respectively):

```
/* P1 */  x := y + z    ||    /* P2 */  y := z + x
```

Assume that the two assignments are executed asynchronously on two different processors. Also assume that the variables `x`, `y`, and `z` are shared between the processors.

On a typical machine, execution of `x := y + z` would correspond to execution of a sequence of atomic "instructions" such as:

```
/* P1 */  reg1 := y ;  reg1 := reg1 + z ;  x := reg1
```

Similarly, execution of `y := z + x` would correspond to execution of the sequence:

```
/* P2 */  reg2 := z ;  reg2 := reg2 + x ;  y := reg2
```

There are 20 different ways that these two sequences might be interleaved. A parallel execution of the two sequences might correspond to any of the possible interleavings. The final values of the shared variables will be different depending upon which interleaving models the execution.

For example, consider the following interleavings of programs P1 and P2 where the initial values of the variables `x`, `y`, and `z` are 1, 2, and 3, respectively.

```
/* P1 */  reg1 := y ;  reg1 := reg1 + z ;  x := reg1
/* P2 */  reg2 := z ;  reg2 := reg2 + x ;  y := reg2

Interleaving #1                  Interleaving #2
/* P1 */ reg1 := y ;             /* P1 */ reg1 := y ;
/* P2 */ reg2 := z ;             /* P1 */ reg1 := reg1 + z ;
/* P1 */ reg1 := reg1 + z ;      /* P2 */ reg2 := z ;
/* P2 */ reg2 := reg2 + x ;      /* P1 */ x := reg1 ;
/* P1 */ x := reg1 ;             /* P2 */ reg2 := reg2 + x ;
/* P2 */ y := reg2               /* P2 */ y := reg2
```

Note that interleaving #1 leaves shared variable `y` with the value 4 and interleaving #2 leaves `y` with the value 8. The result depends upon the relative ordering of the assignment to `x` in P1 and the reference to `x` in P2. The result is time dependent and outside of the programmer's control: the relative execution speeds of P1 and P2 affect the final values of the variables.

As the above example illustrates, the result of parallel execution of statements is, in general, indeterminate because of the nondeterministic order of operations on shared data. A key problem in concurrent programming is thus finding a way to tame the nondeterminism without sacrificing the benefits of parallel execution.

Terminology

The following "definitions" are abstractions; in a specific situation the terms may be used to denote slightly different concepts. Much of the following is adapted from the book Concurrency in Programming and Database Systems by Arthur J. Bernstein and Philip M. Lewis (Jones and Bartlett, 1992) or other sources.

Hardware

This section gives definitions for hardware concepts from the perspective of programmers. These are not necessarily the definitions that would be given in a computer architecture class.

A *processor* is a physical device that executes operations sequentially, atomically, and deterministically.

Note: Here we use the terms *operation* and *execute* in a general sense. An example of a processor is a cpu that can execute instructions.

Sequential means that the processor executes no more than one operation at any one time.

Atomic means that, once the processor selects an operation for execution, it executes the operation completely; the processor does not interrupt an operation at an intermediate stage to execute another operation.

In this context, *deterministic* means that:

- each of the processor's operations performs a well-defined action that always produces the same result when operating on the same input data,
- upon completion of the execution of an operation, the next operation to be executed is determined (except in the case of operations that halt the processor).

That is, if an operation execution is deterministic, its results are predictable.

A *control unit* for a processor controls the sequencing of operations.

A *control point* (or *point of execution*) refers to each operation in a "program" for the processor.

If an operation is eligible for execution, the associated control point is *enabled*.

When the processor executes an operation, the associated control point is *serviced*.

The control unit is usually driven by a *clock*; the clock generates a sequence of timing pulses that activates the portions of the processor needed to carry out the operation being executed.

Primitive computers are constructed from a single processor with a single control unit.

A computer is *sequential* if no more than one control point can be serviced at time.

A computer made up of only one processor is sometimes called a *uniprocessor*.

A common way to increase the power of a computer is to provide multiple processors, each with its own control unit.

A computer is *parallel* if multiple control points can be serviced at the same time.

A shared memory multiprocessor (sometimes called a *tightly coupled multiprocessor*) is a parallel computer consisting of several processors sharing a common memory. (The processors communicate by exchanging information through the shared memory. Interprocessor communication is fast, but contention for access

to the shared memory can be costly. As a result, this approach limits computers to a small number of processors.)

A *multicore* computer is a shared memory multiprocessor in which all processors are on the same electronic chip.

A *multicomputer* (sometimes called a *loosely coupled multiprocessor* or an *ensemble* computer) is a parallel computer consisting of several processing nodes connected by high-speed message switching hardware. (The processors communicate by sending messages through the switching hardware. Communication among processors is more costly than with shared memory, but memory contention is avoided.)

A *distributed computer system* is a geographically dispersed group of computers connected by a communication network. (The processors communicate by sending messages through the network. Communication among processors is usually more costly than with a multicomputer.)

Two processors are *synchronous* if their operations are driven by a common clock. Otherwise, they are *asynchronous*.

Asynchronous processors can explicitly synchronize their execution from time to time.

Software

A *program* is a finite description of a task in some programming language.

An *address space* is the collection of all the variables that a program references.

The *program state* is the mapping of the variables in an address space to their values.

A program is *concurrent* if during its execution multiple control points can be enabled at the same time.

A program is *sequential* if during its execution no more than one control point can be enabled at a time.

In essence, sequential programs form a special class of concurrent programs.

Note: Concurrent programs are sometimes called *parallel programs*. Here we tend to use the term *concurrent* to describe logically or potentially simultaneous activities and *parallel* to describe physically simultaneous activities.

A *process* is an agent that

- executes operations on a processor in an address space,
- has a unique associated control point that designates the next operation it will execute.

Note: A processor is a physical device; a process is a logical processor.

The above definition is for what some writers call a *sequential process*; they use the term *process* to describe a more general concept.

A program is *determinate* if it always produces the same result when executed from the same initial state. (In this definition we consider just the external effects of the execution—the relationship between "inputs" and "outputs".)

A program is *indeterminate* if it can produce an unpredictable result when executed from some initial state.

A program is *deterministic* if its control points are always serviced in the same sequence when execution is started from the same initial state. (In this definition we consider the internal effects of the execution—the order in which operations are executed.)

A program is *nondeterministic* if its control points are serviced in an unpredictable sequence when execution is started from some initial state.

Note: A program may be nondeterministic but still determinate. In such a case, each of the alternative execution sequences produce the same result.

Note: Nondeterministic is not the same as random. Nondeterministic means that something is unpredictable; in general, probabilities cannot be assigned to the alternatives.

In a context where a system must repeatedly choose among alternatives, *fairness* means that no alternative will be postponed forever. For example, a fair execution of a concurrent program may mean that any continuously enabled control point will eventually be serviced.

A concurrent program is *asynchronous* if, given the identity of one enabled control point, we cannot infer the identities of the others. That is, the concurrent processes are not executed in lock-step; they can be executed on separately clocked processors.

A concurrent program is *synchronous* if, given the identity of one enabled control point, we can always infer the identity of the remaining enabled control points.

We sometimes distinguish between a program and its *environment*—the other agents (i.e., programs, devices, and people) with which the program interacts during its execution.

A *transformational program* interacts with its environment only at initiation and termination. It can be specified in terms of its initial and final states.

A *reactive program* interacts with its environment throughout execution. For some reactive programs (e.g., operating systems) termination may be considered an abnormal occurrence. Thus reactive program cannot be specified purely in terms of initial and final states.

In essence, a transformational program is a special case of a reactive program.

A system (e.g., a program) is *open* if it can interact with (perhaps unknown) agents in its environment; otherwise, the system is *closed*.

A *concurrent programming language* consists of two components (often integrated into a single language):

- a *computation language* to compute values and manipulate local data objects,
- a *composition* (or *coordination*) *language* to combine programs into more complex programs.

Why develop concurrent programs?

- to get results faster—concurrent programs can be executed in parallel iIn particular, to better take advantage of the processor "cores" available)
- to express programs more naturally—the universe is naturally parallel place
- to have fun!? :-)