G52CON: Concepts of Concurrency

Lecture 3: Synchronisation

Brian Logan
School of Computer Science
bsl@cs.nott.ac.uk

Outline of this lecture

- mutual exclusion and condition synchronisation
- modelling concurrency as interleaving
- the problem of interference
- example: a shared buffer
- example: loss of increment
- the need for mutual exclusion between critical sections
- the archetypical mutual exclusion problem

Synchronising concurrent processes

To cooperate, the processes in a concurrent program must *communicate* with each other:

- communication can be programmed using *shared variables* or *message passing*;
  - when shared variables are used, one process *writes* into a shared variable that is *read* by another;
  - when message passing is used, one process *sends* a message that is *received* by another;
- ***the main problem in concurrent programming is synchronising this communication***

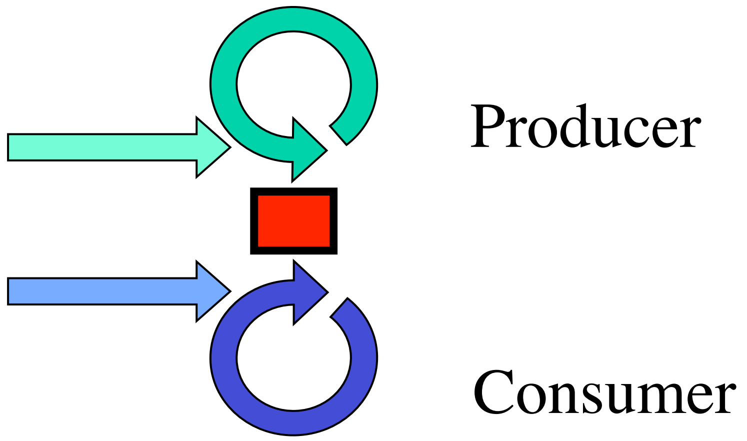
Synchronisation

There are two main synchronisation problems in concurrent programming:

- **Mutual Exclusion**: ensuring that statements in different processes cannot execute at the same time.
- **Condition Synchronisation**: delaying a process until some Boolean condition is true. This is usually implemented by having one process wait for an event that is signalled by another process.

Example: a shared memory location

Communication between a process that produces data and a process which uses it, can be implemented using a *shared memory location*



- one process (the *producer*) writes data to the memory location
- the other (the *consumer*) reads data from the memory location

Shared memory synchronisation

Synchronisation conditions for the shared memory location:

- *mutual exclusion* is necessary to ensure that the producer and consumer do not access the memory location at the same time—i.e., that partial data is not read or that partially read data is not overwritten;

- *condition synchronisation* may be necessary to ensure that the consumer doesn't get too far ahead of the producer and vice versa—i.e., data is not read before it has been written or read twice, and that data is not overwritten before it has been read.

Condition synchronisation

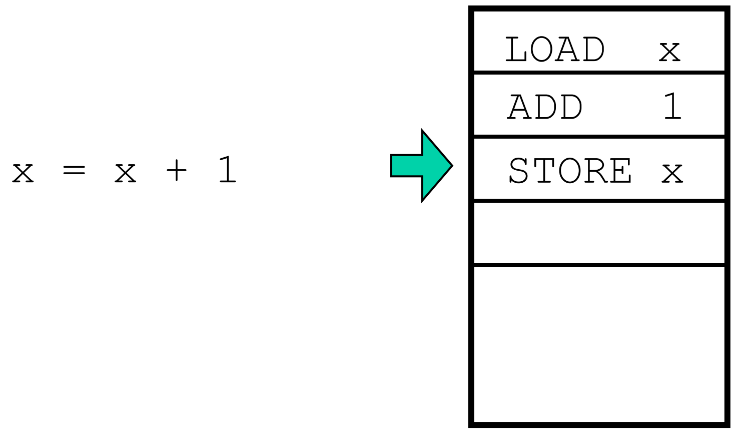
Of the two problems, *condition synchronisation* is the easier to solve.

- the simplest solution is to use *busy waiting*–the process simply sits in a loop until the condition is true
  - e.g., in the shared buffer problem, the consumer can loop repeatedly checking to see if there is a data item ready
- there are other, more efficient, solutions which we will discuss in later lectures.

In this lecture, we will focus on the problem of *mutual exclusion*.

Sequential programs

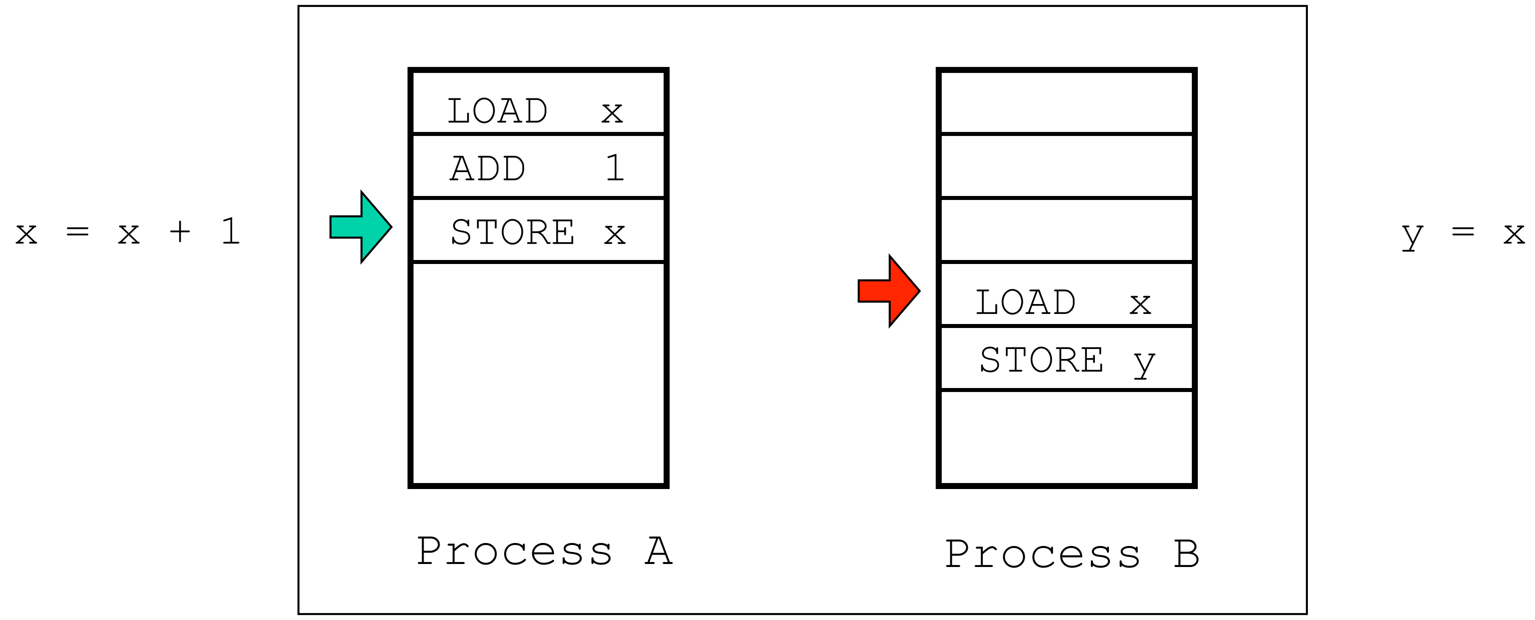
All programs are sequential in that they execute a sequence of instructions in a pre-defined order:



There is a single thread of execution or control.

Concurrent programs

A **concurrent program** is one consisting of two or more processes — threads of execution or control



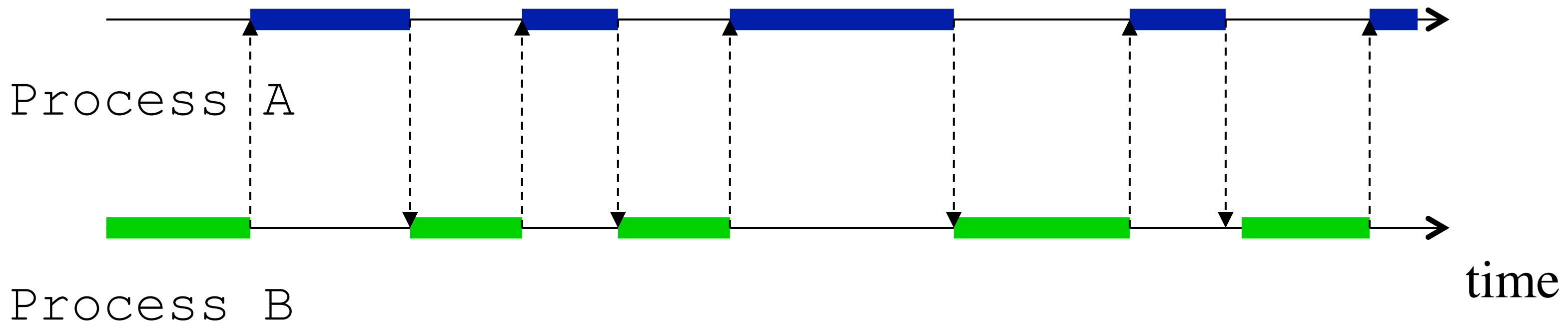
Each *process* is itself a sequential program.

Multiprogramming

- if we ignore pipelining, it is not possible for a single processor to execute more than one instruction at a time
- thus, on the time scale of a single instruction, concurrent processes are not possible
- on a longer time scale, however, several processes may be interleaved so that each runs for a short time, then another is run, and so on
- over a long enough time scale, the processes appear to run truly concurrently, although at any given point in time, only one of them is executing
- this is called *multiprogramming*

Concurrent execution

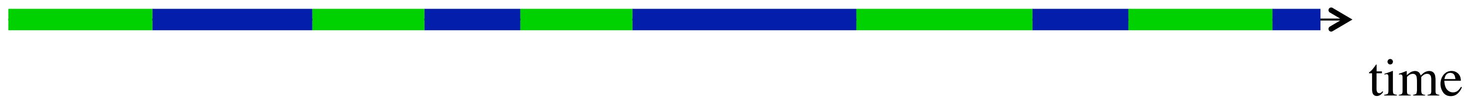
Consider a multiprogramming implementation of a concurrent program consisting of two processes:



- the switching between processes occurs voluntarily (e.g., `yield()` in Java); or
- in response to interrupts, which signal external events such as the completion of an I/O operation or clock tick to the processor.

Interleaving

The processor executes a sequence of instructions which is an *interleaving* of the instruction sequences from each process:



Process switching does not affect the order in which instructions are executed by each process.

Asynchronous process execution

- in multiprocessing systems the processes usually have little or no control over how they are interleaved
- **advantage**: applications programmer can ignore the problems of timesharing the processes
- **disadvantage**: processes effectively run *asynchronously*—we can't predict the relative speed with which they run, which runs first, at which point they will be suspended etc.
- this indeterminism makes debugging much more difficult than is the case for sequential programs

Multiprocessing implementations

Multiprocessing implementations of concurrency can be modelled in the same way:

- each program statement or machine instruction ultimately reduces to a sequence of *atomic actions* on the shared memory, e.g., loading and storing registers

- the effect of executing a set of atomic actions in parallel is equivalent to executing them in some arbitrary serial order, since the state transformations caused by an atomic action are indivisible, and hence cannot [by definition] be affected by atomic actions executed in parallel with it

Serialising parallel atomic actions

Two processes running on different processors can write to a shared memory location in parallel:

- since writing is an *atomic* operation, one of the writes must go first
- which actually goes first is determined by the Memory Management Unit (MMU)

Modelling concurrency

We therefore assume that:

- concurrency is modelled as interleaving;
- processes execute at arbitrary relative speeds—a process can take arbitrarily long to proceed from one instruction to the next; and
- instructions from processes are arbitrarily interleaved.

This is referred to as an *asynchronous* model of execution.

Interference

If instructions from different processes are arbitrarily interleaved, *any* interleaving which is not explicitly prohibited is allowed

- inevitably, some interleavings will have results you don’t want
- *interference* occurs when two processes read and write shared variables in an unpredictable order, and hence with unpredictable results

An example of interference

Process 1

```
  // initialisation code

1 tail = tail + 1;
2 queue[tail] = data1;

  // other code ...
```

Process 2

```
// initialisation code

tail = tail + 1;          1
queue[tail] = data2;      2

// other code ...
```

Shared datastructures

```
Object queue[SIZE];
integer tail;
```

An example trace

Possible interleaving	If the initial value of `tail` is 6
`P1:tail = tail + 1;`	`tail == 7`
`P2:tail = tail + 1;`	`tail == 8`
`P2:queue[tail] = data2;`	`queue[8] == data2`
`P1:queue[tail] = data1;`	`queue[8] == data1;`

Counting traces

- how many distinct traces are there of the example program?
- how many of these traces are *safe* (i.e., do not result in loss of data)?

Counting traces

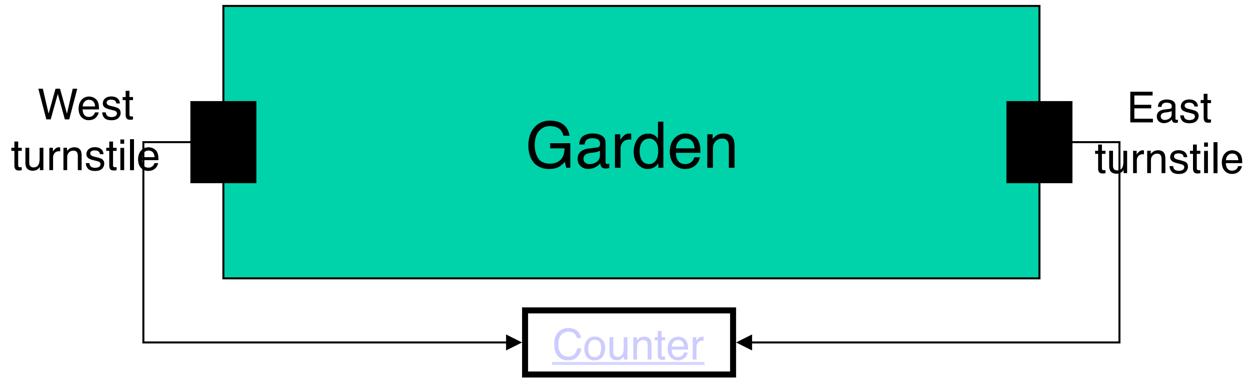
$$\begin{tikzcd}
 & & & P2_1 \arrow[r] & P2_2 & \text{trace 1} \\
 & & P1_2 \arrow[ur] & & & \\
 & P1_1 \arrow[ur] \arrow[ddr] & & & & \\
 & & & P1_2 \arrow[r] & P2_2 & \text{trace 2} \\
 & & P2_1 \arrow[ur] \arrow[dr] & & & \\
 & & & P2_2 \arrow[r] & P1_2 & \text{trace 3} \\
{} \arrow[uuuur] \arrow[ddddr] & & & & & \\
 & & & P2_2 \arrow[r] & P1_2 & \text{trace 4} \\
 & & P1_1 \arrow[ur] \arrow[dr] & & & \\
 & & & P1_2 \arrow[r] & P2_2 & \text{trace 5} \\
 & P2_1 \arrow[uur] \arrow[dr] & & & & \\
 & & P2_2 \arrow[dr] & & & \\
 & & & P1_1 \arrow[r] & P1_2 & \text{trace 6}
\end{tikzcd}$$

Safe traces

trace 1	`queue[7] = data1`	`queue[8] = data2`
trace 2	`queue[7] = ∅`	`queue[8] = data2`
trace 3	`queue[7] = ∅`	`queue[8] = data1`
trace 4	`queue[7] = ∅`	`queue[8] = data1`
trace 5	`queue[7] = ∅`	`queue[8] = data2`
trace 6	`queue[7] = data2`	`queue[8] = data1`

Explaining the Ornamental Gardens problem

A large ornamental garden is open to members of the public who can enter through either of two turnstiles.



- the owner of the garden writes a computer program to count how many people are in the garden at any one time
- the program has two processes, each of which monitors a turnstile and increments a shared counter whenever someone enters via that processes' turnstile.

Ornamental Gardens program

```
// West turnstile

init1;
while(true) {
  // wait for turnstile
  count = count + 1;
  // other stuff ...

}
```

```
// East turnstile

init2;
while(true) {
  // wait for turnstile
  count = count + 1;
  // other stuff ...

}
```

```
count == 0
```

Loss of increment

```
// shared variable
integer count = 10;
```

West turnstile process

```
count = count + 1;
```

East turnstile process

```
count = count + 1;
```

1. loads the value of `count` into a CPU register (`r == 10`) — West

2. loads the value of `count` into a CPU register (`r == 10`) — East

3. increments the value in its register (`r == 11`) — East

4. increments the value in its register (`r == 11`) — West

5. stores the value in its register in `count` (`count == 11`) — East

6. stores the value in its register in `count` (`count == 11`) — West

Avoiding interference

To avoid interference, we need to ensure that no two processes access a shared variable at the same time

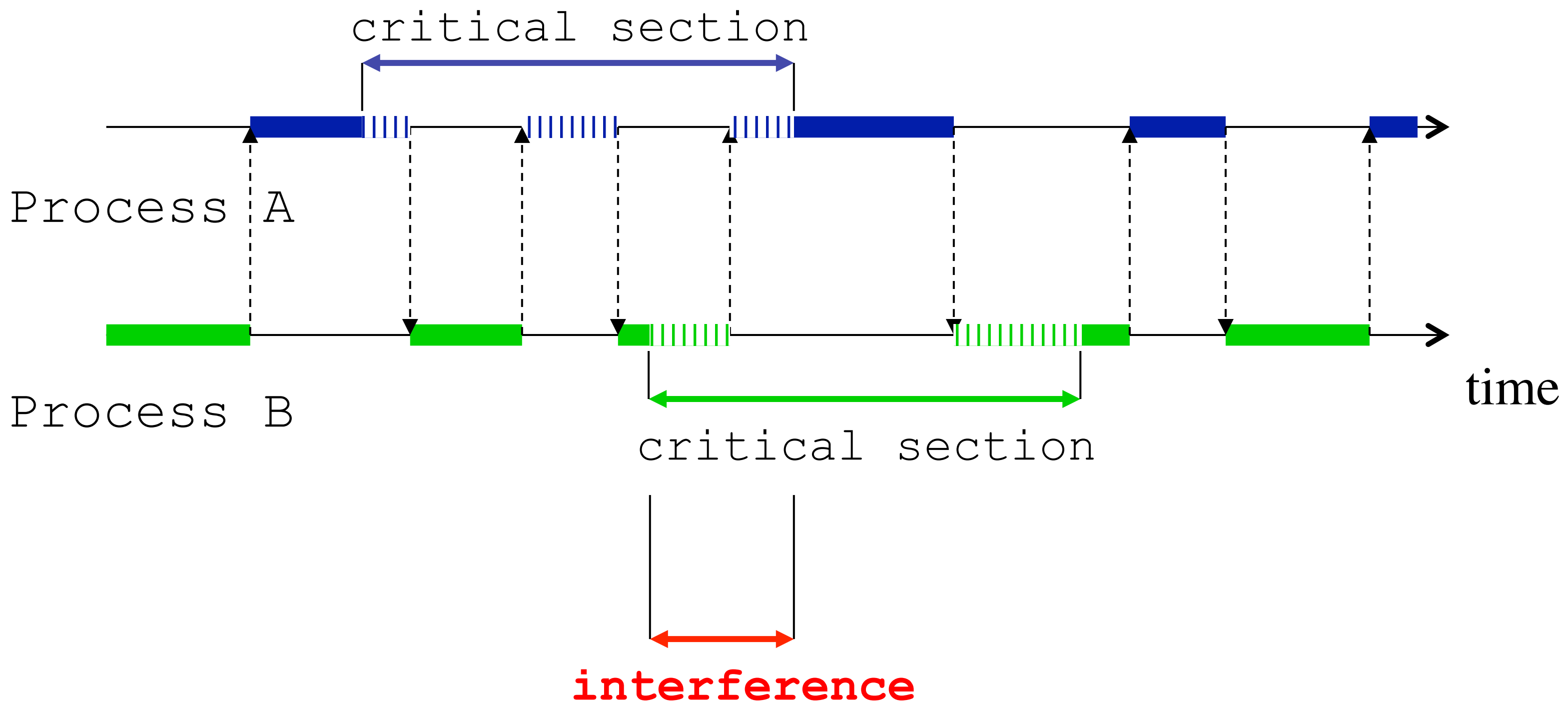
- we do this by marking such sections of code as *critical* and requiring that no two processes are executing critical code at the same time
- this is termed *mutual exclusion*.

Critical sections

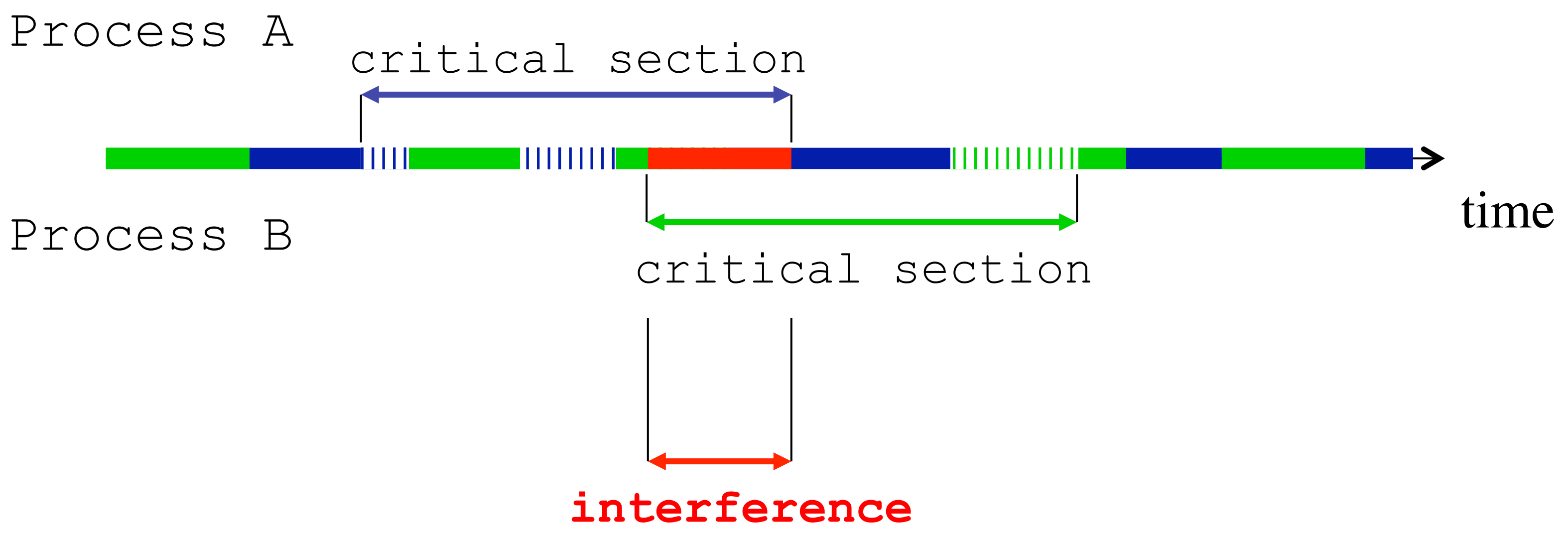
A *critical section* is a section of code belonging to a process in a concurrent program that:

- accesses a shared resource, e.g., a shared variable, shared communication channel, shared file etc.; and
- for correct behaviour of the program only *one* process may access the shared resource at a time.

Interleaving critical sections



Interleaving critical sections



Mutual exclusion

If processes A and B contain critical sections then the overlapped execution of process A and process B *could* result in interference:

- *mutual exclusion* is the requirement that, at any given time, at most one process in a concurrent program is executing a critical section
- once one process has entered a critical section, no other process may enter a critical section until the first process has exited its critical section.

Mutual exclusion of *critical sections*

Mutual exclusion is a constraint on the execution of processes which applies between the process's critical sections, *not* between the processes themselves

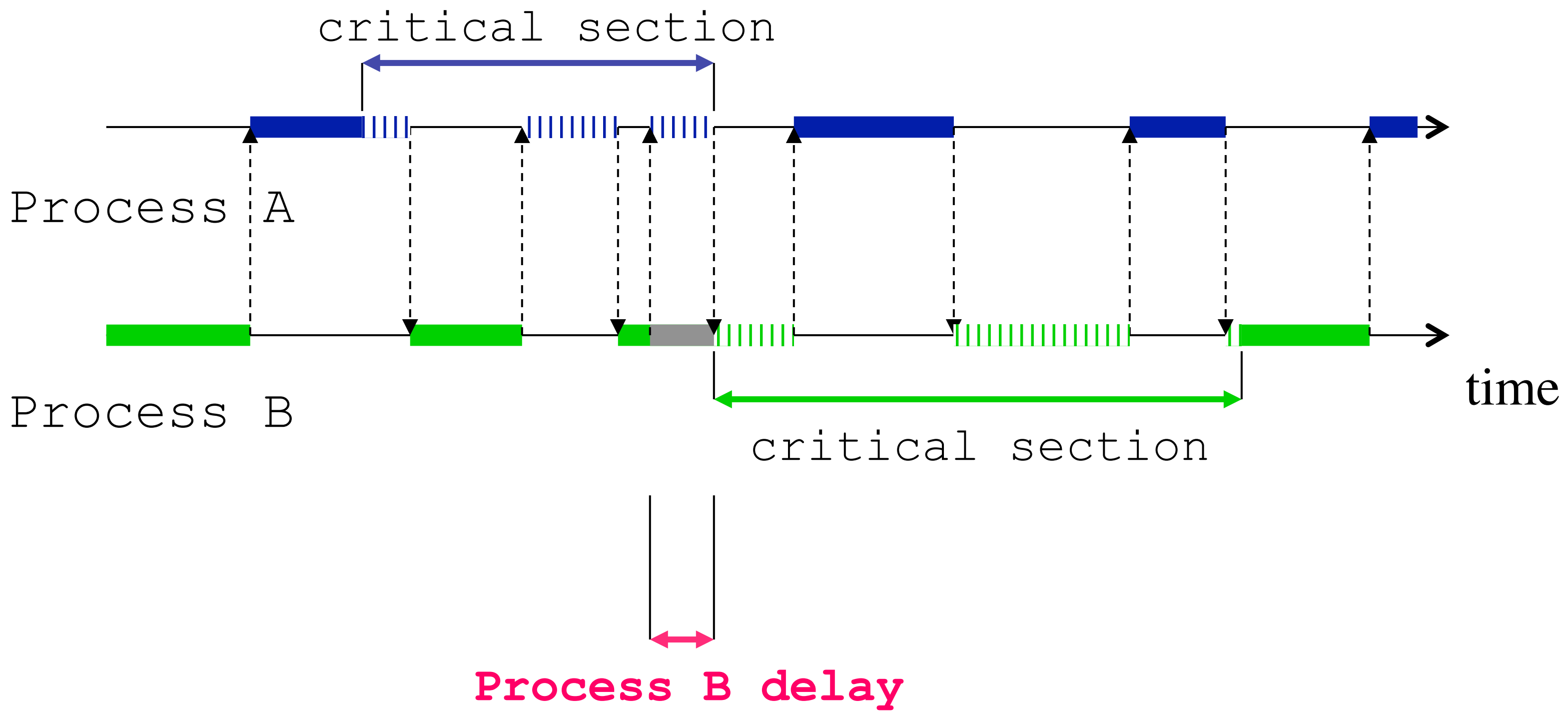
- for example, the fact that A and B contain critical sections does *not* mean that their execution should never overlap, only that the execution of their *critical sections* should never overlap

Enforcing mutual exclusion

To ensure mutual exclusion, one (or more) process may have to *wait* to enter their critical section(s):

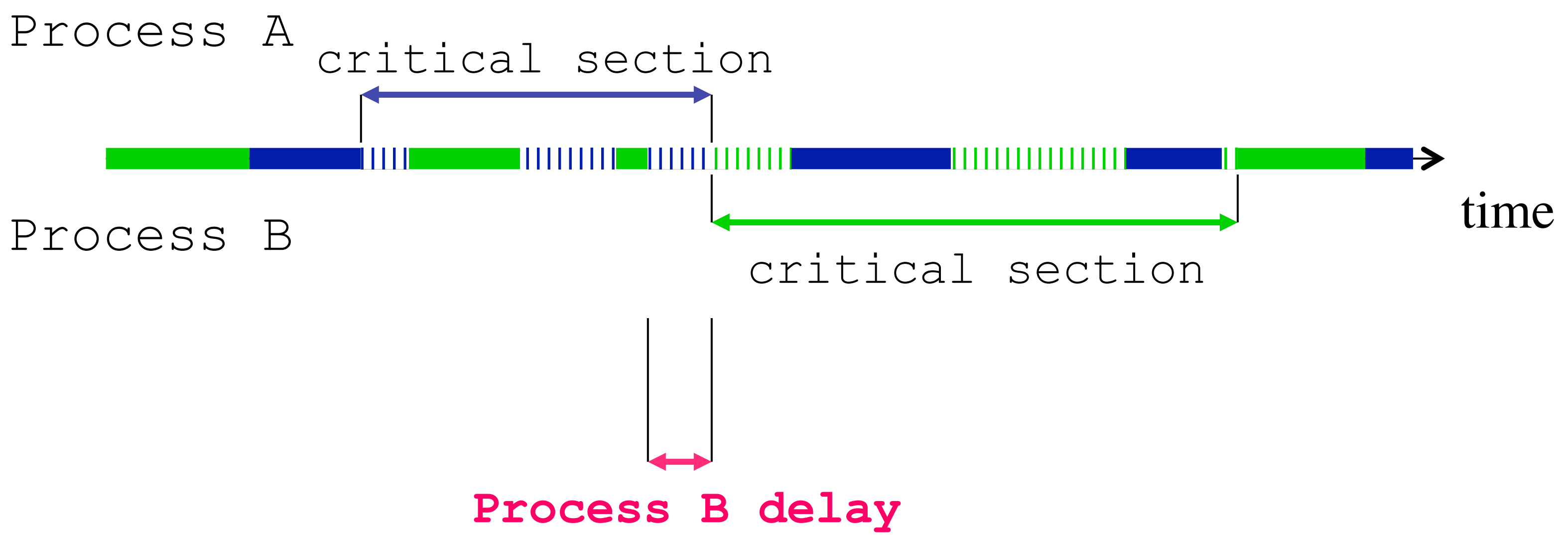
- for example, if Process A is already in its critical section when process B tries to enter its critical section, then Process B will have to wait
- this prevents interleaving of instructions in the critical sections
- in a multiprogramming implementation, this needn't increase the overall run time of the application—the same instructions are executed, only in a different order

Non-interleaved critical sections

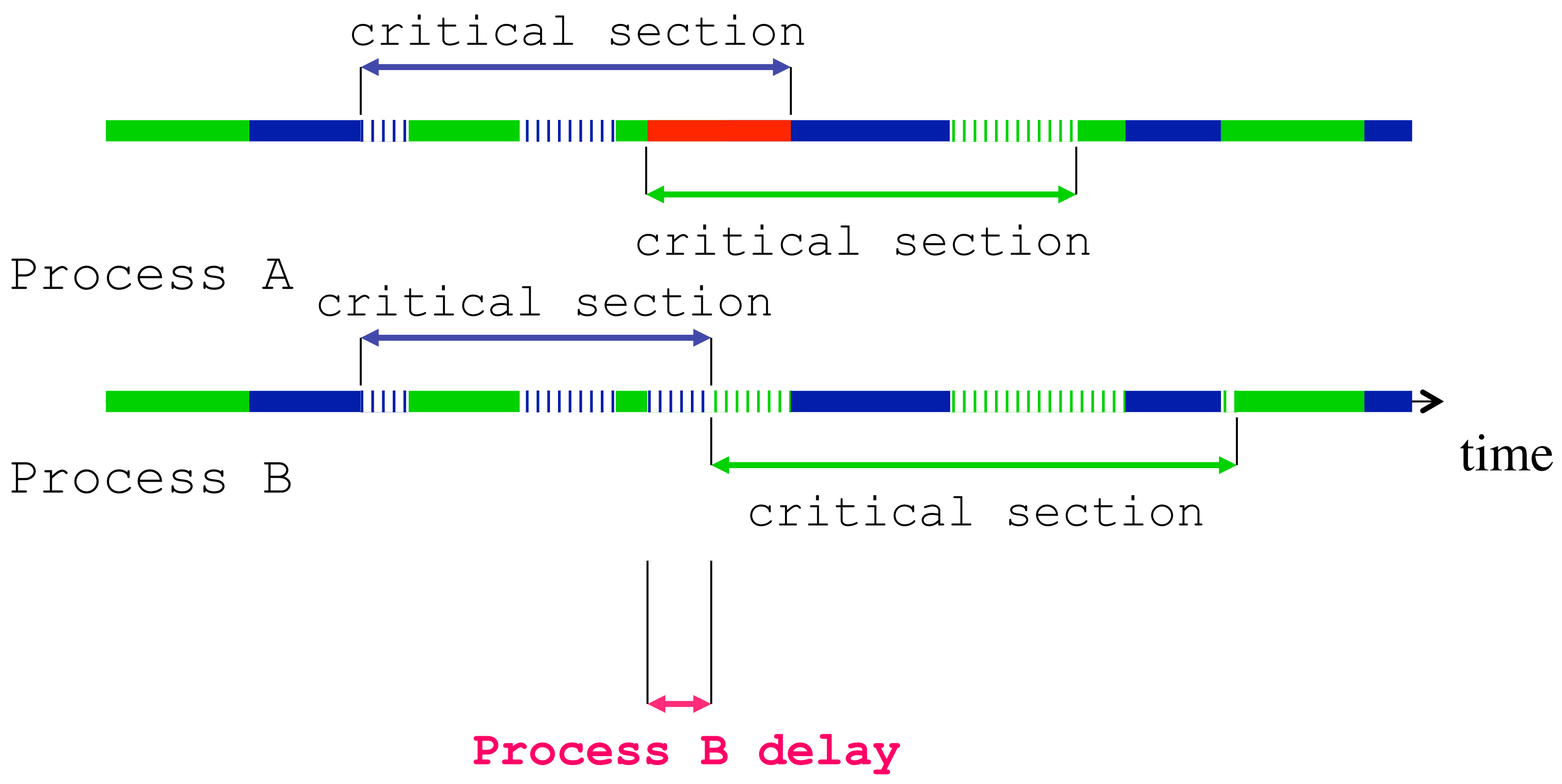


Non-interleaved critical sections

Note that the execution of Process B can be interleaved with the execution of Process A's critical section, *so long as B is not in it's critical section* (and vice versa)



Mutual exclusion of critical sections



Classes of critical sections

In concurrent programs there are often a large number of critical sections which do not all need to be mutually exclusive with each other:

- a *class of critical sections* is a set of critical sections, all of which must be mutually exclusive with others *in the same class*
- critical sections in different classes do not need to be mutually exclusive

Archetypical mutual exclusion

Any program consisting of *n* processes for which mutual exclusion is required between critical sections belonging to just one class can be written:

```
// Process 1       // Process 2   ...   // Process n
init1;             init2;               initn;
while(true) {      while(true) {        while(true) {
 crit1;             crit2;               critn;
 rem1;              rem2;                remn;
}                  }                     }
```

where $\texttt{init}_i$ denotes any (non-critical) initialisation, $\texttt{crit}_i$ denotes a critical section and $\texttt{rem}_i$ denotes the (non-critical) remainder of the program, and *i* is 1, 2, … *n*.

Archetypical mutual exclusion

We assume that `init`, `crit` and `rem` may be of any size:

- `crit` must execute in a finite time
- `init` and `rem` may be infinite.
- `crit` and `rem` may vary from one pass through the `while` loop to the next

With these assumptions it is possible to rewrite *any* process with critical sections into the archetypical form.

The next lecture

Atomic Actions

Suggested reading:

- Andrews (2000), chapter 2, sections 2.1 and 2.4, chapter 3, section 3.2;
- Ben-Ari (1982), chapter 2.