Concurrent Systems

3D4 ⟷ CS2016

Operating Systems

Andrew Butterfield
ORI.G39, Andrew.Butterfield@scss.tcd.ie

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

with thanks to Mike Brady

Practical 2

- Advance notice of Practical 2 release on BB, discussed in Class
- Code illustrating Producer-Consumer also on BB
  - link to Oracle web page about condition variables and mutex with documented code snippets
  - Example program code that uses the Oracle snippets

Abstraction Layers

- We can consider concurrency at various levels of abstraction:
  - externally visible concurrent "behaviours"
  - identifiable threads of execution with "atomic" behaviours
  - thread library API implementation (e.g., pthreads)
    - creating/ending threads, using mutexes and condition variables, signalling
  - implementation of the thread library
    - how threads are represented and scheduled, implementation of mutexes and condition variables
  - C/assembler low-level code
  - Hardware support for concurrency
    - interrupts, traps, **atomic** test-and-set (TAS) or compare-and-swap (CAS) instructions

We are hereabouts right now

Upwards...

- We are now going to move up in the abstraction order
  - Look at the "user" view, i.e. focus more on externally visible behaviour
  - If we can come up with a good high-level plan:
    - we can analyse it for obvious pitfalls
    - we have a plan for how we setup and orchestrate the various threads.

Important high level properties

- Things we want to be true for concurrent systems:
  - Safety
    - bad things NEVER happen, or, Safety Properties ("Invariants") are ALWAYS true
  - Liveness
    - important things will ALWAYS EVENTUALLY happen
  - Fairness
    - no work that is ready to run will be overlooked indefinitely

Situations worth avoiding

- Deadlock
    - the system should not "freeze up" due to resource access bugs
- Starvation
    - no work ready to go should be left waiting indefinitely
- Livelock
    - no process or thread should enter an infinite loop where it keeps computing but never produces any visible effect

The Challenge of Concurrency

- Conventional testing and debugging is not generally useful.
  - We assume that the sequential parts of concurrent programs are correctly developed.
- Beyond normal sequential-type bugs, there is a whole range of problems caused by errors in communication and synchronisation. They can not easily be reproduced and corrected.
- So, we are going to use notions, algorithms and formalisms to help us design concurrent programs that are correct by design.

Sequential Process

- A *sequential process* is the execution of a sequence of *atomic statements*.
  - E.g. Process P could consist of the execution of $P_1, P_2, P_3, \ldots$.
  - Process Q could be $Q_1, Q_2, Q_3, \ldots$.
- We think of each sequential process having its own separate PC and being a distinct entity.

Concurrent Execution

- A concurrent system is modelled as a collection of sequential processes, where the atomic statements of the sequential processes can be arbitrarily *interleaved*, but respecting the sequentiality of the atomic statements within their sequential processes.

- E.g. say P is P_1,P_2 and Q is Q_1,Q_2.

Scenarios for P and Q.

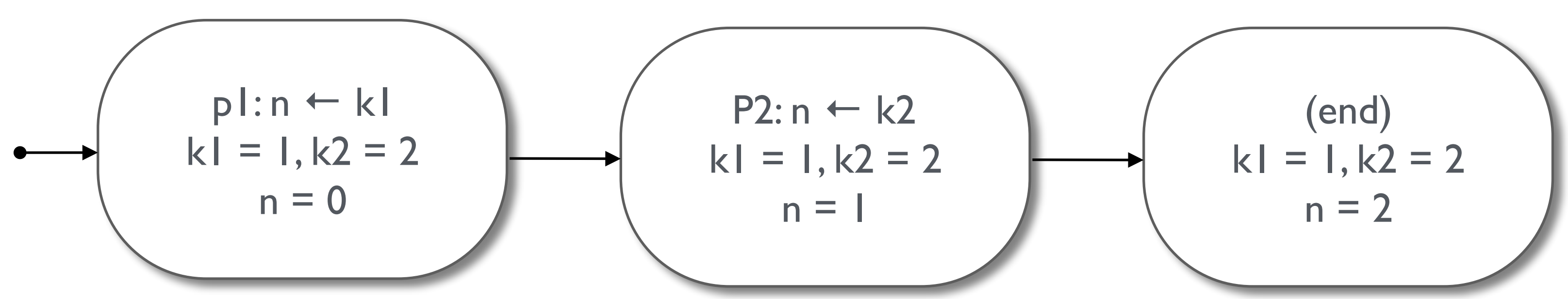
p1→q1→p2→q2
p1→q1→q2→p2
p1→p2→q1→q2
q1→p1→q2→p2
q1→p1→p2→q2
q1→q2→p1→p2

Notation

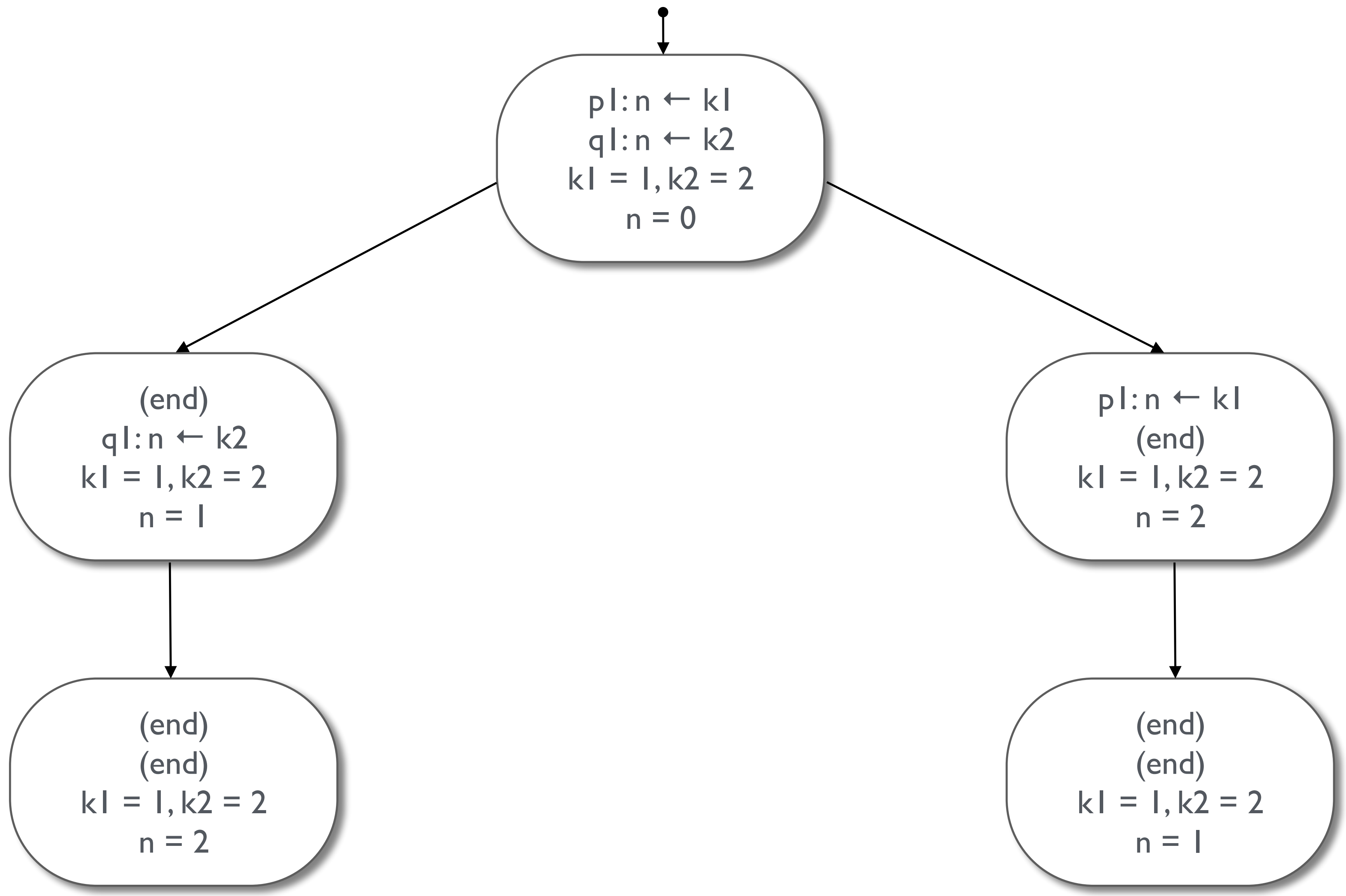
Trivial Concurrent Program (Title)		
integer n ← 0 (Globals)		
p (Process Name)	**q**	
integer k1 ← 1 (Locals)	integer k2 ← 2	
p1: n ← k1 (Atomic Statements)	q1: n ← k2	

Sample Sequential Program

Trivial Sequential Program	
integer n ← 0	
	integer k1 ← 1
	integer k2 ← 2
p1:	n ← k1
p2:	n ← k2



Trivial Concurrent Program



p1, q1

q1, p1