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

How fast can we go?

- We have a sequential program that runs too slow
- We have extra hardware resources that could be used to speed it up.
- How fast can we go?

Do the math ....

$T(n)$	Time to run program with n parallel processors
T	Shorthand for $T(1)$
$S(n)$	Speedup with n processors
$E(n)$	Efficiency of Speedup
p	Proportion of T spent executing parallelisable part.
s	Speedup possible for parallelisable part

Speedup and Efficiency

Maximum speedup:

$$T(n) \geq T(1)/n$$

Speedup:

$$S(n) = T(1)/T(n)$$

Efficiency:

$$E(n) = S(n)/n$$

Program Time and Effective Speedup

- Time to run program without parallelism:

$$T = (1-p)T + pT$$

- Parallel (effective) speedup vs. processor count:

$$s \leq n$$

Speedup related to *n* and *s*.

Time to run program with speedup s of parallelisable part:

$$T(n) = (1-p)T + pT/s = (1-p+p/s)T$$

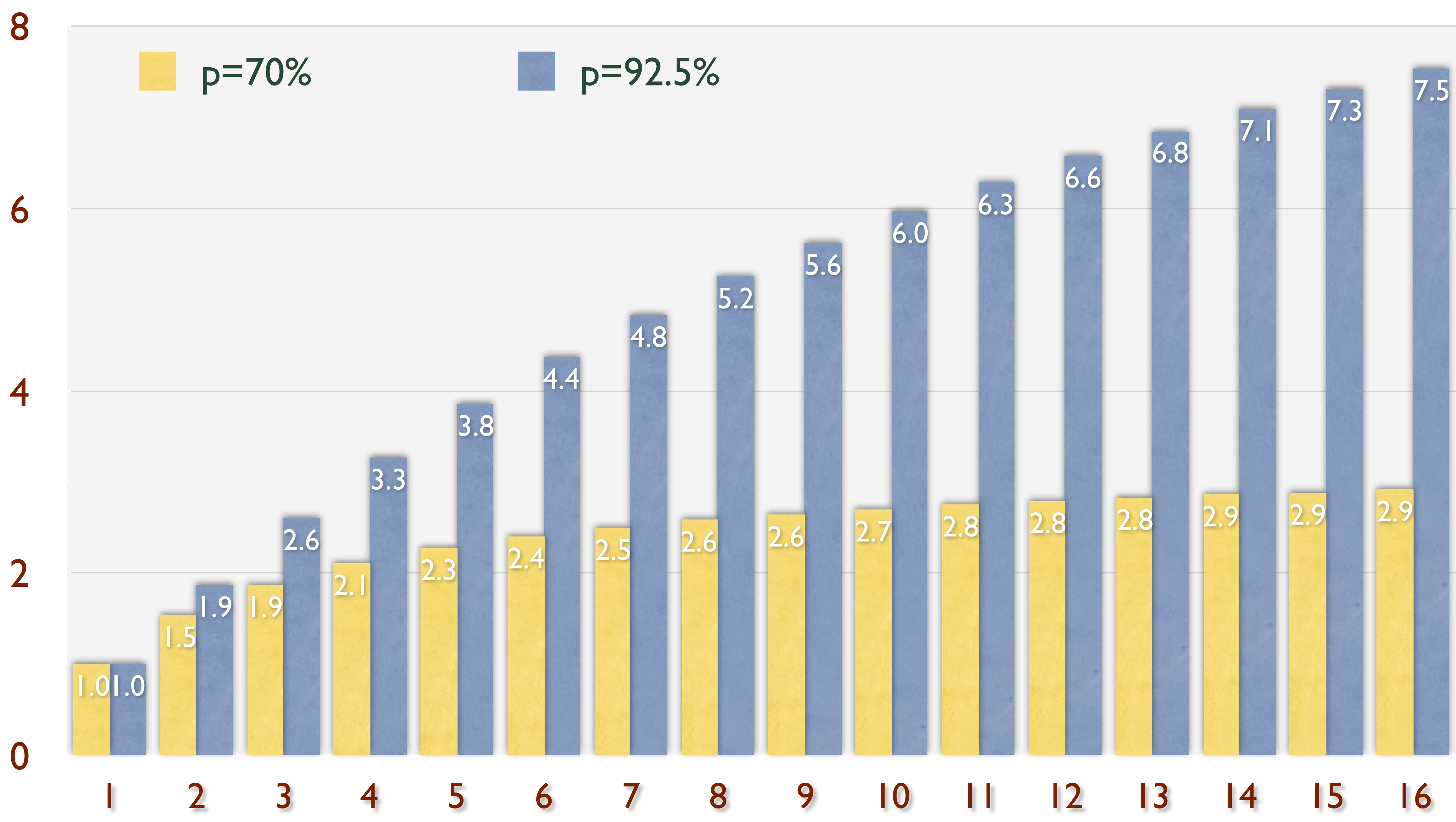
Speedup when running program with parallelisable speedup s:

$$\begin{aligned} S(n) &= T/(1-p+p/s)T \\ &= 1/(1-p+p/s) \end{aligned}$$

Amdahl's Law

$$S(n) = \frac{1}{1 - p + (p/s)} \qquad s \leq n$$

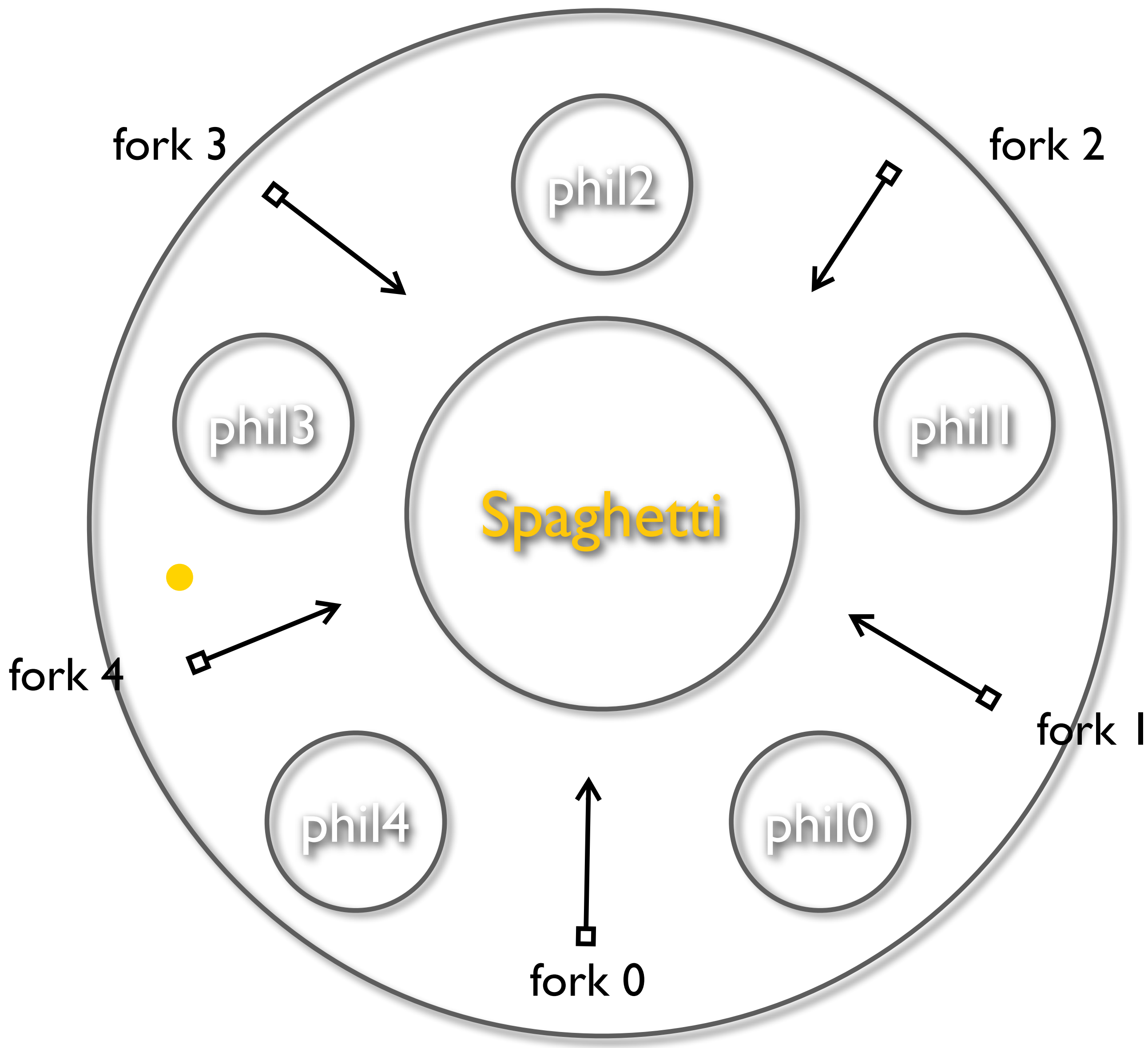
Graphically, Bad News



Implications

- Even a small fraction of sequential code in a program can seriously interfere with speedup.
  - Note that the code protected by a mutex can only run sequentially!
  - If code has wait a while for a mutex, then that waiting time, has to be considered sequential.
- To maximise performance, inherently sequential code has to be minimised.

Dining Philosophers Problem



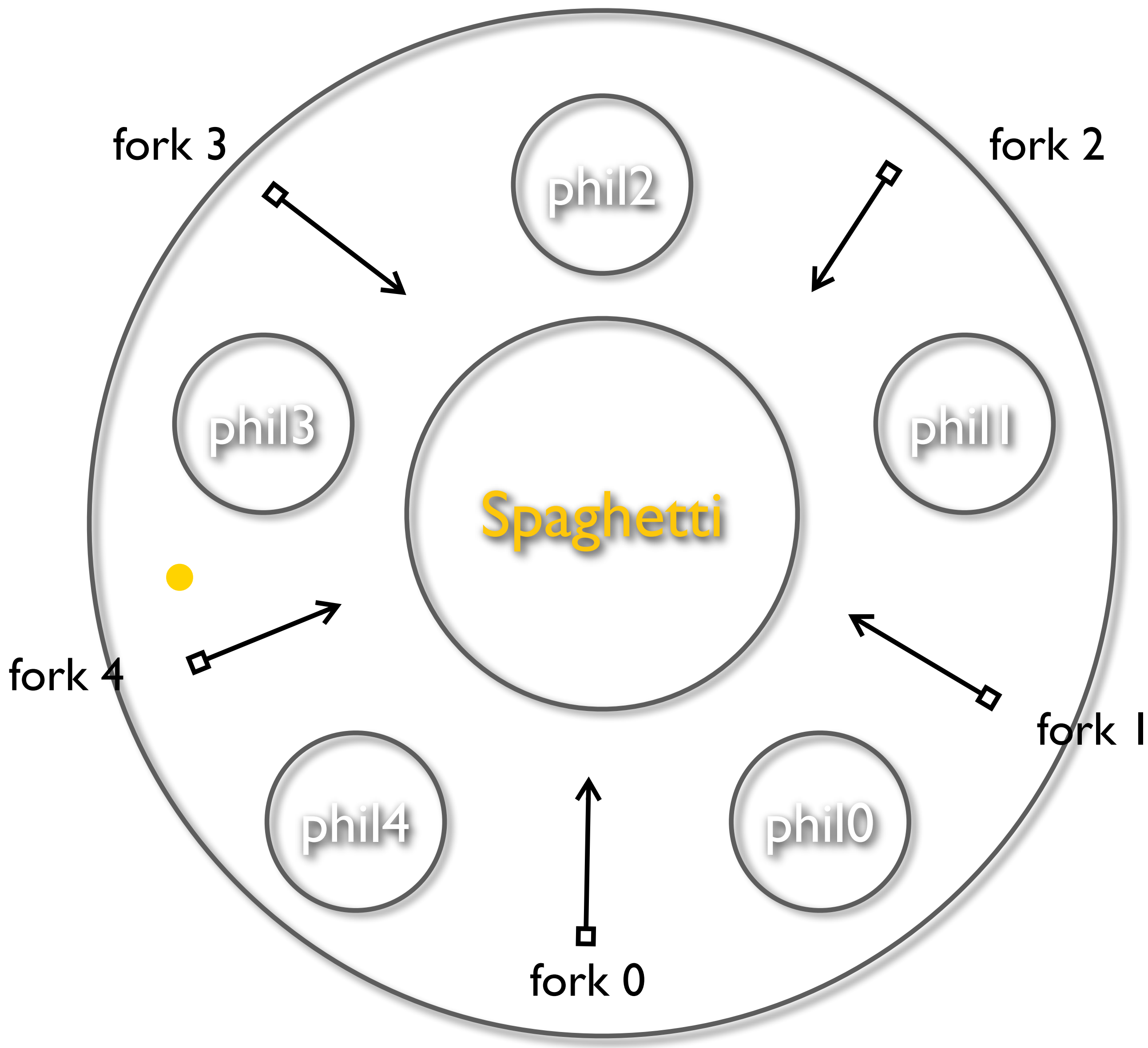
Philosophers want to
repeatedly
think and then eat.
Each needs two forks.
Infinite supply of pasta (ugh!).
How to ensure they don't starve.

A model problem for thinking about
problems in Concurrent Programming:
Deadlock
Livelock
Starvation/Fairness
Data Corruption

Dining Philosophers

- Features:
  - A philosopher eats only if s/he has two forks.
  - No two philosophers may hold the same fork simultaneously
- Characteristics of Desired Solution:
  - Freedom from deadlock
  - Freedom from starvation
  - Efficient behaviour generally.

Dining Philosophers Problem



Philosophers want to
repeatedly
think and then eat.
Each needs two forks.
Infinite supply of pasta (ugh!).
How to ensure they don't starve.

A model problem for thinking about
problems in Concurrent Programming:
Deadlock
Livelock
Starvation/Fairness
Data Corruption

Dining Philosophers

- Features:
  - A philosopher eats only if s/he has two forks.
  - No two philosophers may hold the same fork simultaneously
- Characteristics of Desired Solution:
  - Freedom from deadlock
  - Freedom from starvation
  - Efficient behaviour generally.

Modelling the Dining Philosphers

- We imagine the philosophers participate in the following observable events:

Event Shorthand	Description
think.p	Philosopher *p* is thinking.
eat.p	Philosopher *p* is eating
pick.p.f	Philosopher *p* has picked up fork *f*.
drop.p.f	Philosopher *p* has dropped fork *f*.

What a philosopher does:

- A philosopher wants to: *think ; eat ; think ; eat; think ; eat ; ….*
- In fact each philosopher needs to do: *think ; pick forks ; eat ; drop forks ; …*
- We can describe the behaviour of the *i*th philosopher as:

  Phil(i) = think.i ; pick.i.i ; pick.i.i+ ; eat.i ; drop.i.i ; drop.i.i+ ; Phil(i)

  - Here *i+* is shorthand for *(i+1) mod 5.*
- What we have are five philosophers running in parallel (||):

  Phil(0) || Phil(1) || Phil(2) || Phil(3) || Phil(4)

What can (possibly) go wrong ?

- Consider the following (possible, but maybe unlikely) sequence of events, assuming that, just before this point, all philosophers are *think.p*-ing…

  pick.0.0 ; pick.1.1 ; pick.2.2 ; pick.3.3 ; pick 4.4 ;

- At this point, every philosopher has picked up their left fork.
  - Now each of them wants to pick up its right one.
  - But its right fork is its righthand neighbours left-hand fork!
  - Every philosopher wants to *pick.i.i+* , but can't, because it has already been *pick.i+.i+*-ed!
  - Everyone is stuck and no further progress can be made
- DEADLOCK !

“Implementing” *pick* and *drop*

- In effect *pick.p.f* attempts to lock a mutex protecting fork *f*.
- So each philosopher is trying to lock two mutexes for two forks before they can *eat.p*.
- The *drop.p.f* simply unlocks the mutex protecting *f*.

You can't always rely on the scheduler…

- A possible sequence we might observe, starting from when philosophers 1 and 2 are thinking, could be:

  pick.1.1 ; pick.2.2 ; pick.2.3 ; eat.2 ; drop.2.2

  - now, philosopher 1 has picked fork 1 but is waiting for it to be dropped by philosopher 2.
  - But philosopher 2 is still running, and so drops the other fork, has a quick think, and then gets quickly back to eating once more:

    pick.1.1 ; pick.2.2 ; pick.2.3 ; eat.2 ; drop.2.2 ; drop.2.3 ; think.2 ; pick.2.2 ; …

  - Philospher 1 could get really unlucky and never be scheduled to get the lock for fork 2. It is queuing on the mutex for fork 2, but when philosopher 2 unlocks it, somehow the lock, and control is not handed to philosopher 1.
- STARVATION (and its close friend UN-FAIRNESS)