# Remember Integration with mutexes?

```
pthread_mutex_t  mutex = PTHREAD_MUTEX_INITIALIZER;

void *IntegratePart(void *i) {

  double a,b,area;
  int rc;

  a = (int)i * H ;
  b = a + H;
  area = trapezoid(a,b);

  // critical section with mutex
  rc = pthread_mutex_lock(&mutex);
    checkResults("pthread_mutex_lock()\n", rc);

  answer=answer+area;

  rc = pthread_mutex_unlock(&mutex);
    checkResults("pthread_mutex_lock()\n",rc);

    pthread_exit(NULL);
}
```

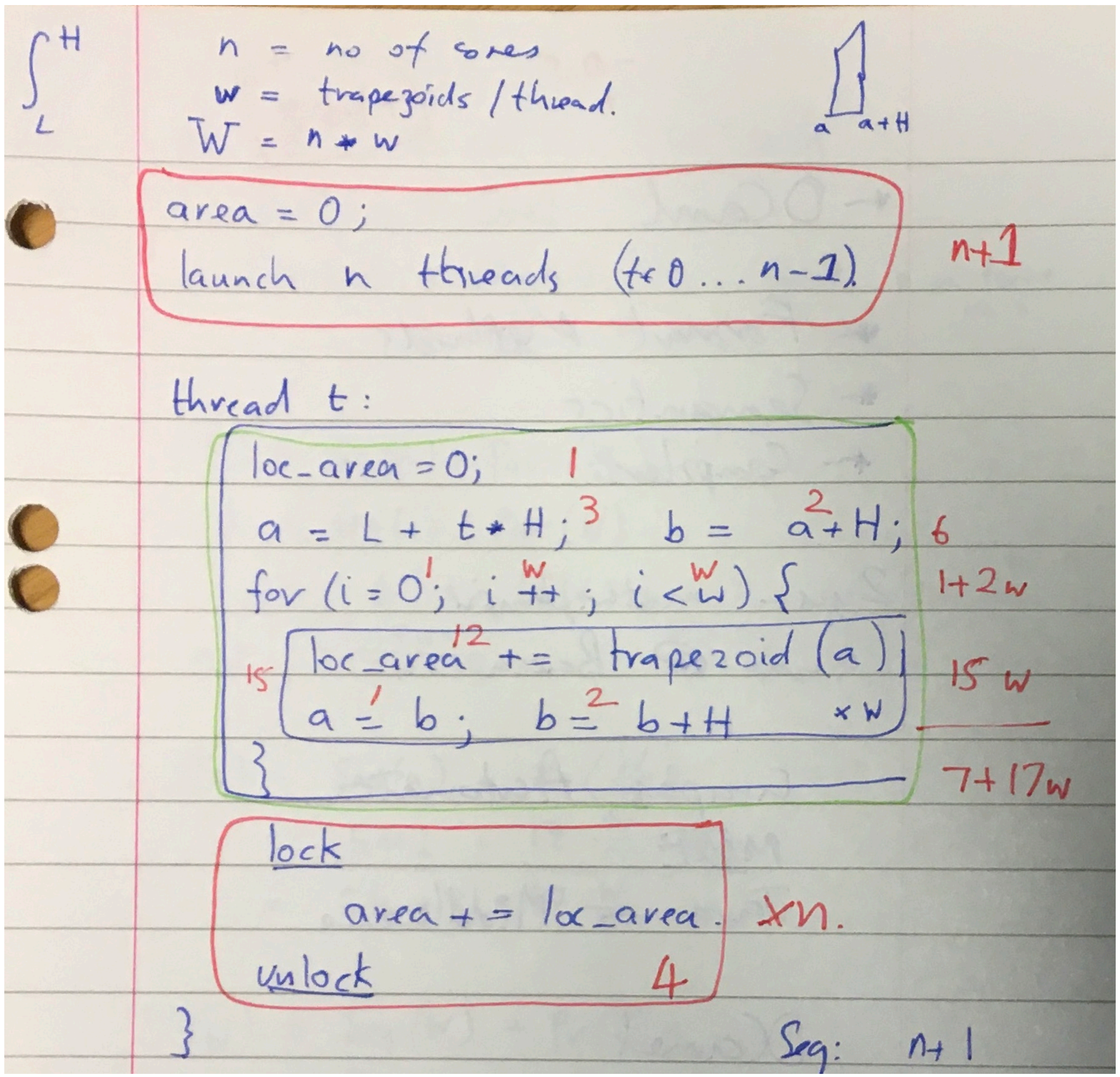
What does Amdahl's law say here?

Assume we have one processor per slice?

# Some (simplifiying) assumptions.

- Assume
  - we can ignore the time spent creating threads
  - assume mutex lock and unlock each require one time-step
  - assume assignment and each arithmetic operation or function call require one time step.:
- Parallelisable code computes a, b, and area (using `trapezoid`, which uses `f`)
  - Cost of `f` is 3
  - Cost of `trapezoid` is 10
- Total cost for a, b and area is 15.
- Serial Code is mutex lock/unlock and answer update: total cost is 4.

# Using a thread to do several trapezoids




We get two formulas

Sequential Part:
Seq(n) = 5n+1

Parallel Part:
Par(w) = 17w+7

Time in terms of n, if W=1000:

T(n) = 5n + 8 + 17000/n

# $T(n) = 5n + 8 + 17000/n$

- A bit of calculus shows T(n) a minimum for n = 92, when it equals 653 steps
  - $T(1) = 17013$
  - $T(92) = 653$
  - $T(1000) = 5025$
  - $T(3400) = 17013$
- So throwing 3400 processors at this problem is as slow as using one!

# Revisiting Matrix Multiplication

(keeping Amdahl's law in mind)

```
for i = 1  to 2 {
  for k = 1 to 3 {
    sum = 0.0;
    for j = 1 to 2
      sum = sum + a[i,j] * b[j,k];
    c[i,k] = sum;
  }
}
```

What is `p`, the proportion of this program that can be sped-up ?

# We need to consider a specific strategy...

e.g., parallelising just the `k` loop:

```
with k = 1;
for i = 1  to 2 {
 for k = 1 to 3 {
    sum = 0.0;
    for j = 1 to 2
      sum = sum + a[1,j] * b[j,1];
    c[i,1] = sum;
 }
}
```

```
with k = 2;
for i = 1  to 2 {
 for k = 1 to 3 {
    sum = 0.0;
    for j = 1 to 2
      sum = sum + a[1,j] * b[j,2];
    c[i,2] = sum;
 }
}
```

```
with k = 3;
for i = 1  to 2 {
 for k = 1 to 3 {
    sum = 0.0;
    for j = 1 to 2
      sum = sum + a[1,j] * b[j,3];
    c[i,3] = sum;
 }
}
```

What is `p`, the proportion of this program that can be sped-up ?

How do we handle updating `sum`?
Make it a critical resource?  Local copies?

# More specific...

parallelising just the `k` loop, with local sums

```
with k = 1;
for i = 1  to 2 {
 ~~for k = 1 to 3 {~~
   sum[1] = 0.0;
   for j = 1 to 2
     sum[1] = sum[1] + a[1,j] * b[j,1];
   c[i,1] = sum[1];
 ~~}~~
}
```

```
with k = 2;
for i = 1  to 2 {
 ~~for k = 1 to 3 {~~
   sum[2] = 0.0;
   for j = 1 to 2
     sum[2] = sum[2] + a[1,j] * b[j,2];
   c[i,2] = sum[2];
 ~~}~~
}
```

```
with k = 3;
for i = 1  to 2 {
 ~~for k = 1 to 3 {~~
   sum[3] = 0.0;
   for j = 1 to 2
     sum[3] = sum[3] + a[1,j] * b[j,3];
   c[i,3] = sum[3];
 ~~}~~
}
```

What is `p`, the proportion of this program that can be sped-up ?

Each `sum[k]` is only used locally!

# Try Six Execution Agents again

```
with k = 1, i=1;
~~for i = 1  to 2 {~~
~~ for k = 1 to 3 {~~
   sum[1,1] = 0.0;
   for j = 1 to 2
     sum[1,1] = sum[1,1] + a[1,j] * b[j,1];
   c[1,1] = sum[1,1];
 ~~}~~
}
```

...

```
with k = 3, i=1;
~~for i = 1  to 2 {~~
~~ for k = 1 to 3 {~~
   sum[1,3] = 0.0;
   for j = 1 to 2
     sum[1,3] = sum[1,3] + a[1,j] * b[j,3];
   c[1,3] = sum[1,3];
 ~~}~~
}
```

```
with k = 1, i=2;
~~for i = 1  to 2 {~~
~~ for k = 1 to 3 {~~
   sum[2,1] = 0.0;
   for j = 1 to 2
     sum[2,1] = sum[2,1] + a[2,j] * b[j,1];
   c[2,1] = sum[2,1];
 ~~}~~
}
```

...

```
with k = 3, i=2;
~~for i = 1  to 2 {~~
~~ for k = 1 to 3 {~~
   sum[2,3] = 0.0;
   for j = 1 to 2
     sum[2,3] = sum[2,3] + a[2,j] * b[j,3];
   c[2,3] = sum[2,3];
 ~~}~~
}
```

\`sum\` is redundant now, and we can simply use\`c\` itself to compute the result.

# Try Six Execution Agents again

```
with k = 1, i=1;
~~for i = 1  to 2 {~~
~~ for k = 1 to 3 {~~
   c[1,1] = 0.0;
   for j = 1 to 2
     c[1,1] = c[1,1] + a[1,j] * b[j,1];
~~}~~
}
```

...

```
with k = 3, i=1;
~~for i = 1  to 2 {~~
~~ for k = 1 to 3 {~~
   c[1,3] = 0.0;
   for j = 1 to 2
     c[1,3] = c[1,3] + a[1,j] * b[j,3];
~~}~~
}
```

```
with k = 1, i=2;
~~for i = 1  to 2 {~~
~~ for k = 1 to 3 {~~
   c[2,1] = 0.0;
   for j = 1 to 2
     c[2,1] = c[2,1] + a[2,j] * b[j,1];
~~}~~
}
```

...

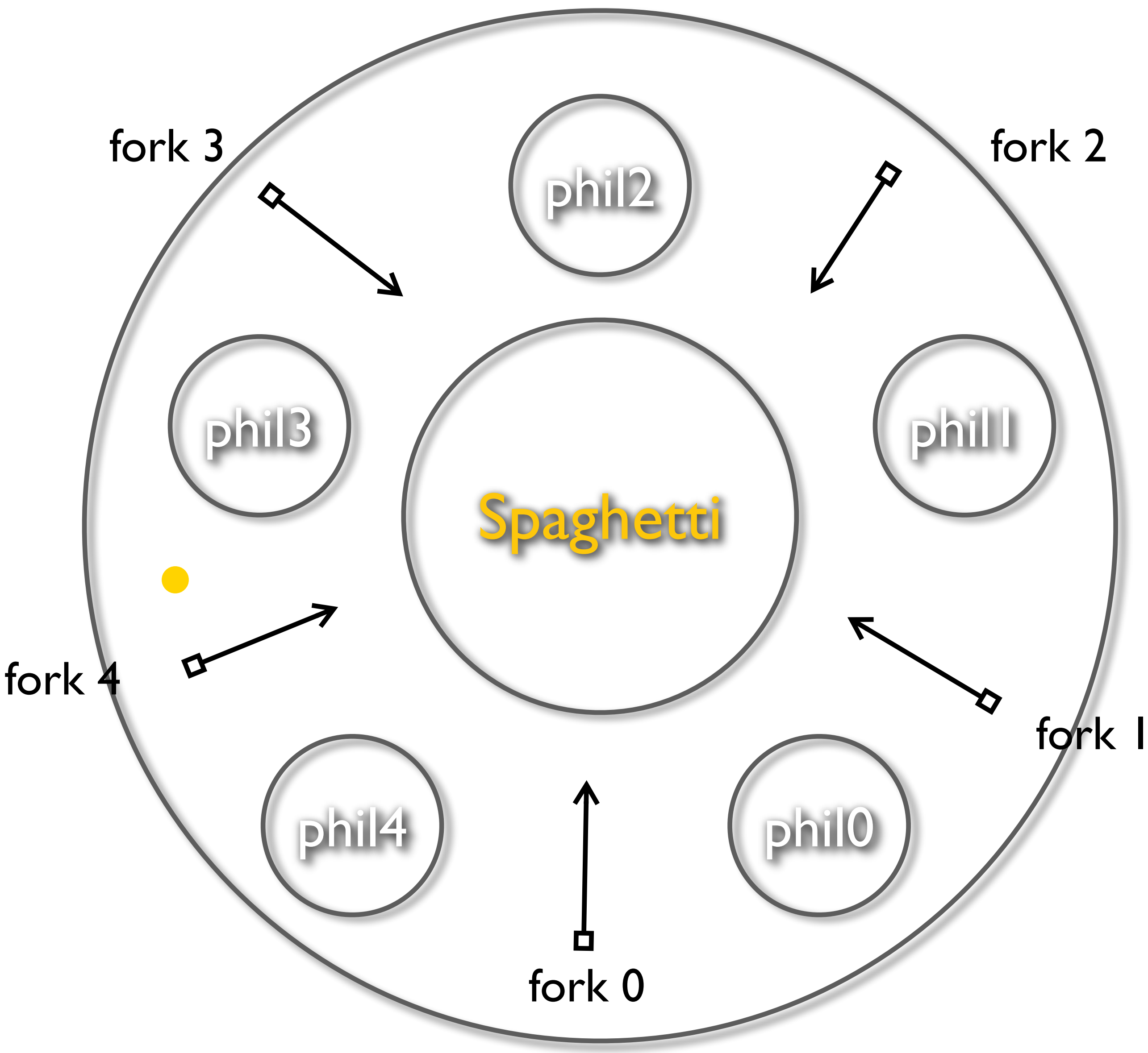
```
with k = 3, i=2;
~~for i = 1  to 2 {~~
~~ for k = 1 to 3 {~~
   c[2,3] = 0.0;
   for j = 1 to 2
     c[2,3] = c[2,3] + a[2,j] * b[j,3];
~~}~~
}
```

Here we see we have full output independence between each component of `c`.

# Matrix Multiply is easy (?)

- Matrix multiply turns out to be highly ("embarrasingly", "massively") parallelisable!
- In principle we should be able to get \`p\` very close to 1.
    - Indeed many supercomputing facilities use massive parallelism to do matrix calculations
- However, for a M x M (square) matrix, the equivalent of the "Six Execution Agents" solution requires $M^2$ processors!
- Also, there are always hidden "non-p" costs in practice
    - Getting fast access by N cores to main memory gets harder to achieve as N grows large
    - While each location in matrix C is only written by one core, each location in both A and B are read by many. Again, this can lead to "bus contention" as processors wait to read shared memory.
        - (Clever understanding of cache behaviour can lead to algorithms that minimise such contention)

# 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 they have 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 ; …*

  - Philosopher 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)

# Communication

- A concurrent or parallel program consists of two or more separate threads of execution, that run independently except when they have to interact
- To interact, they must *communicate*
- Communication is typically implemented by
  - sharing memory
    - One thread writes data to memory; the other reads it
  - passing messages
    - One thread sends a message; the other gets it

# 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 as being a distinct entity that has its own separate program counter (PC).

# 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

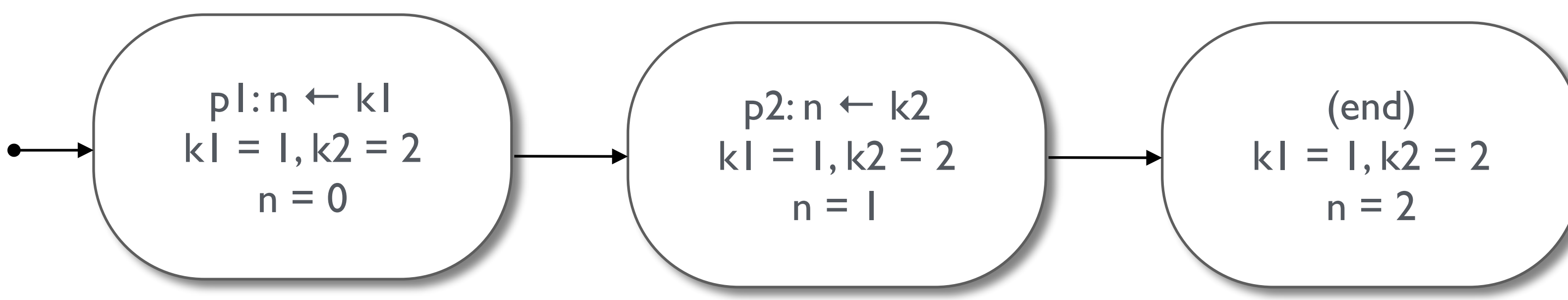
Trivial concurrent program: processes **p** and **q** each do one action that updates global n with the value of their local variable.

| **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 |

All global and local variables are initialised when declared.

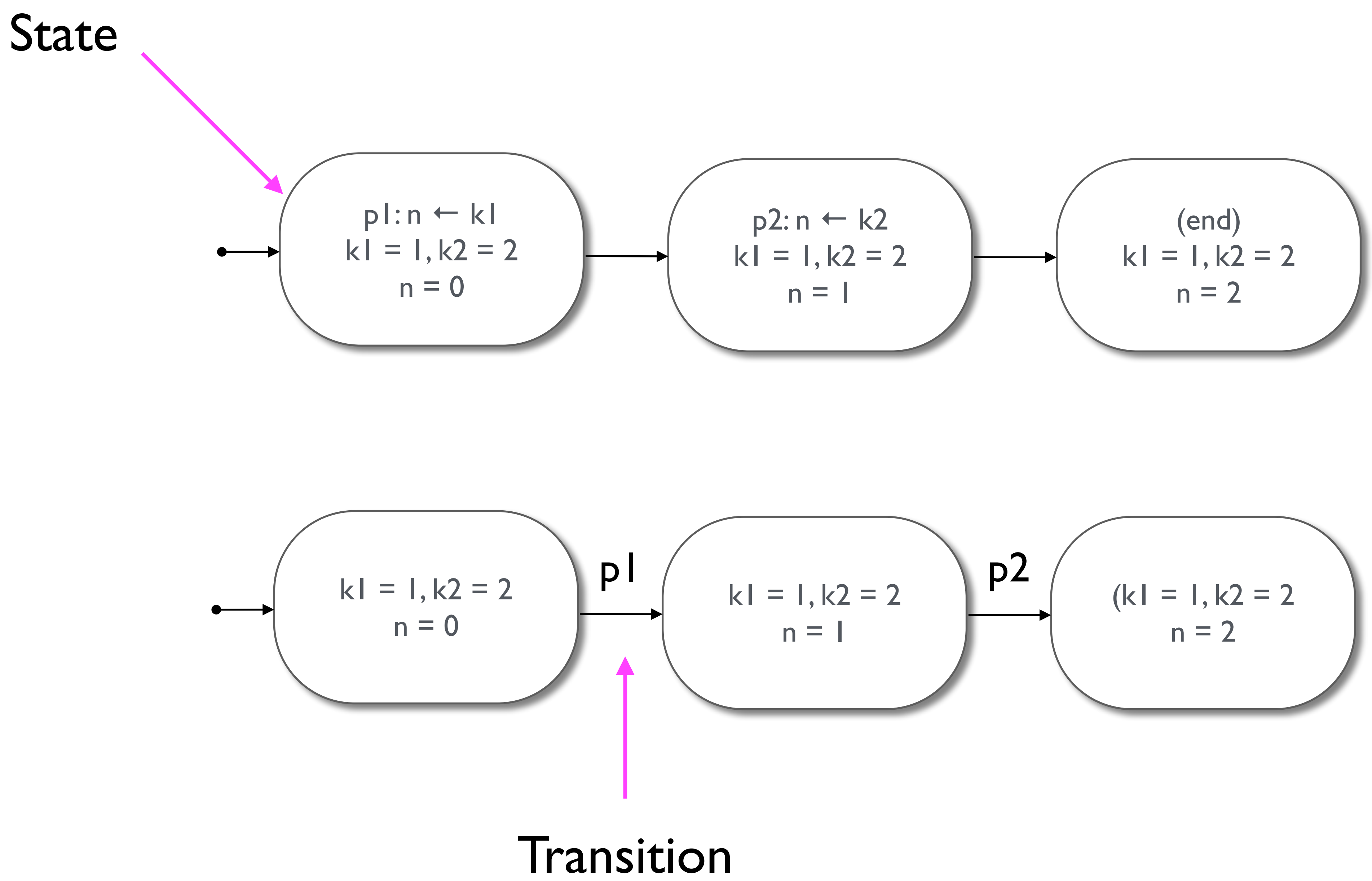
# Simple Sequential Program

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




First line gives next atomic action to be executed

# Transition Diagrams

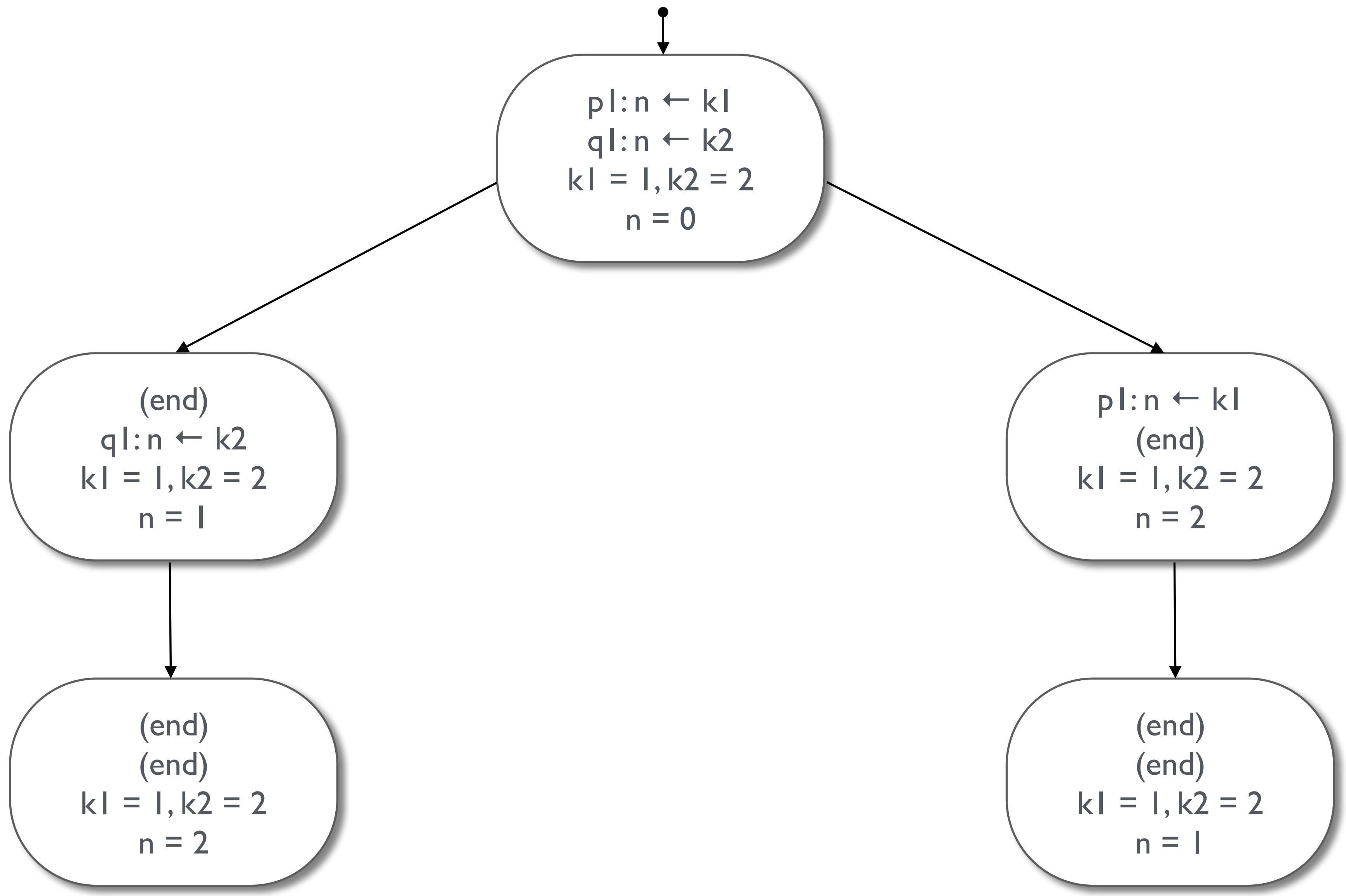


# Simple Concurrent Program (1)

Trivial concurrent program: processes **p** and **q** each do one action that updates global n with the value of their local variable.

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

# Simple Concurrent Program (2)

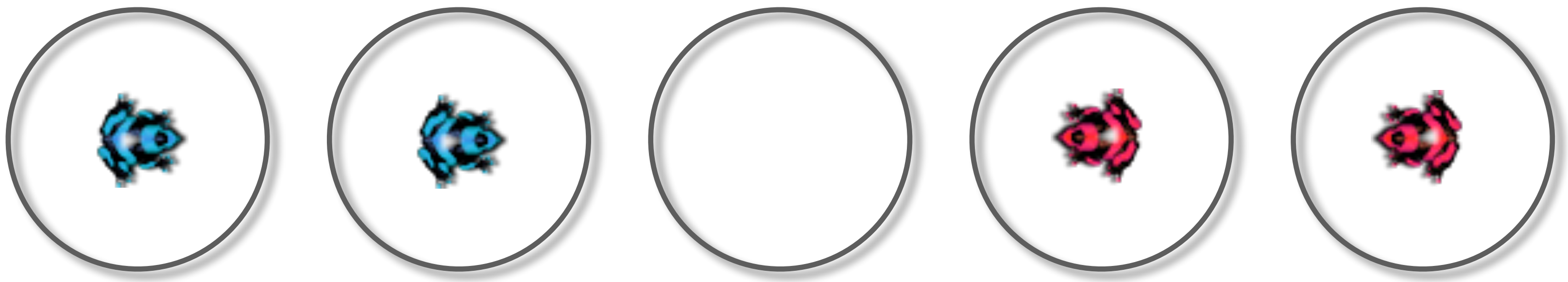



p1, q1          q1, p1

# State Diagrams and Scenarios

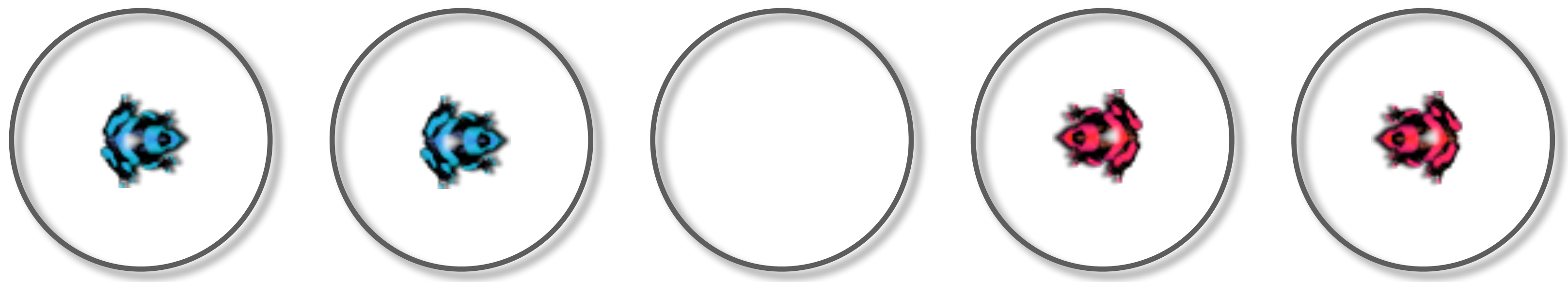
- We could describe all possible ways a program can execute with a state diagram.
  - There is a transition between $s_1$ and $s_2$ ("$s_1$:$s_2$") if executing a statement in $s_1$ changes the state to $s_2$.
  - A state diagram is generated inductively from the starting state.
    - If $\exists$ $s_1$ and a transition $s_1$:$s_2$, then $\exists$ $s_2$ and a directed arc from $s_1$:$s_2$
- Two states are identical if they have the same variable values and the same directed arcs leaving them.
- A *scenario* is one path through the state diagram.

# Example — Jumping Frogs

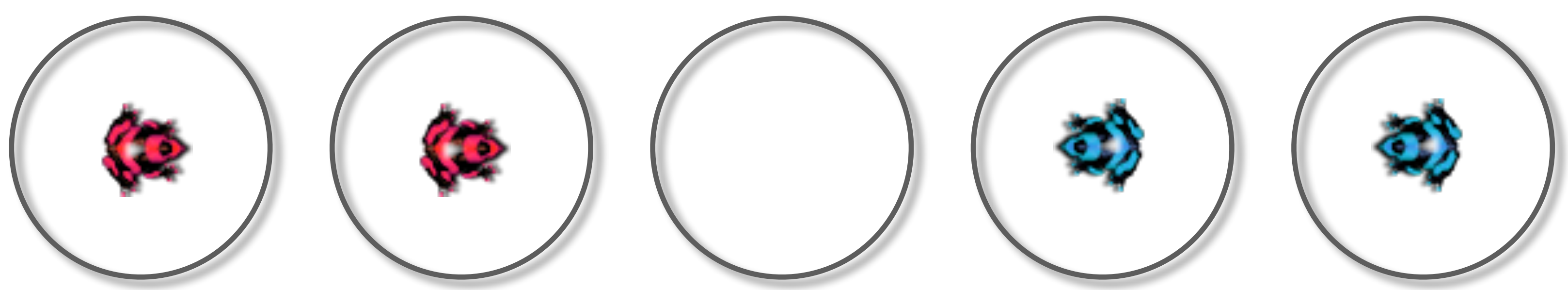


- A frog can move to an adjacent stone if it's vacant.
- A frog can hop over an adjacent stone to the next one if that one is vacant.
- No other moves are possible.

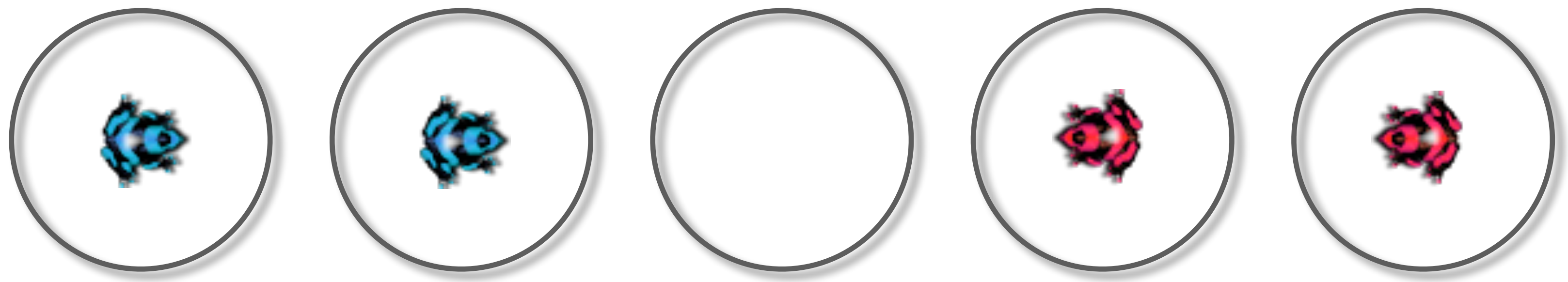
# Can we interchange the positions of the frogs?



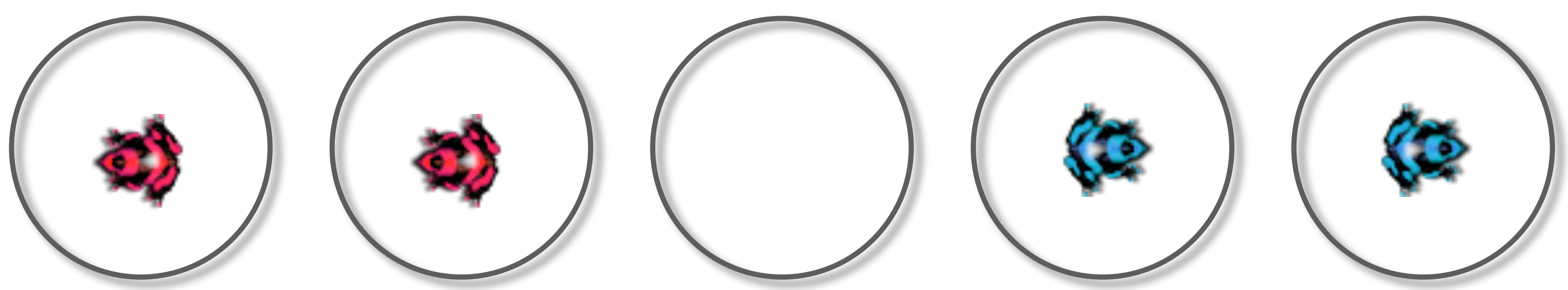
to



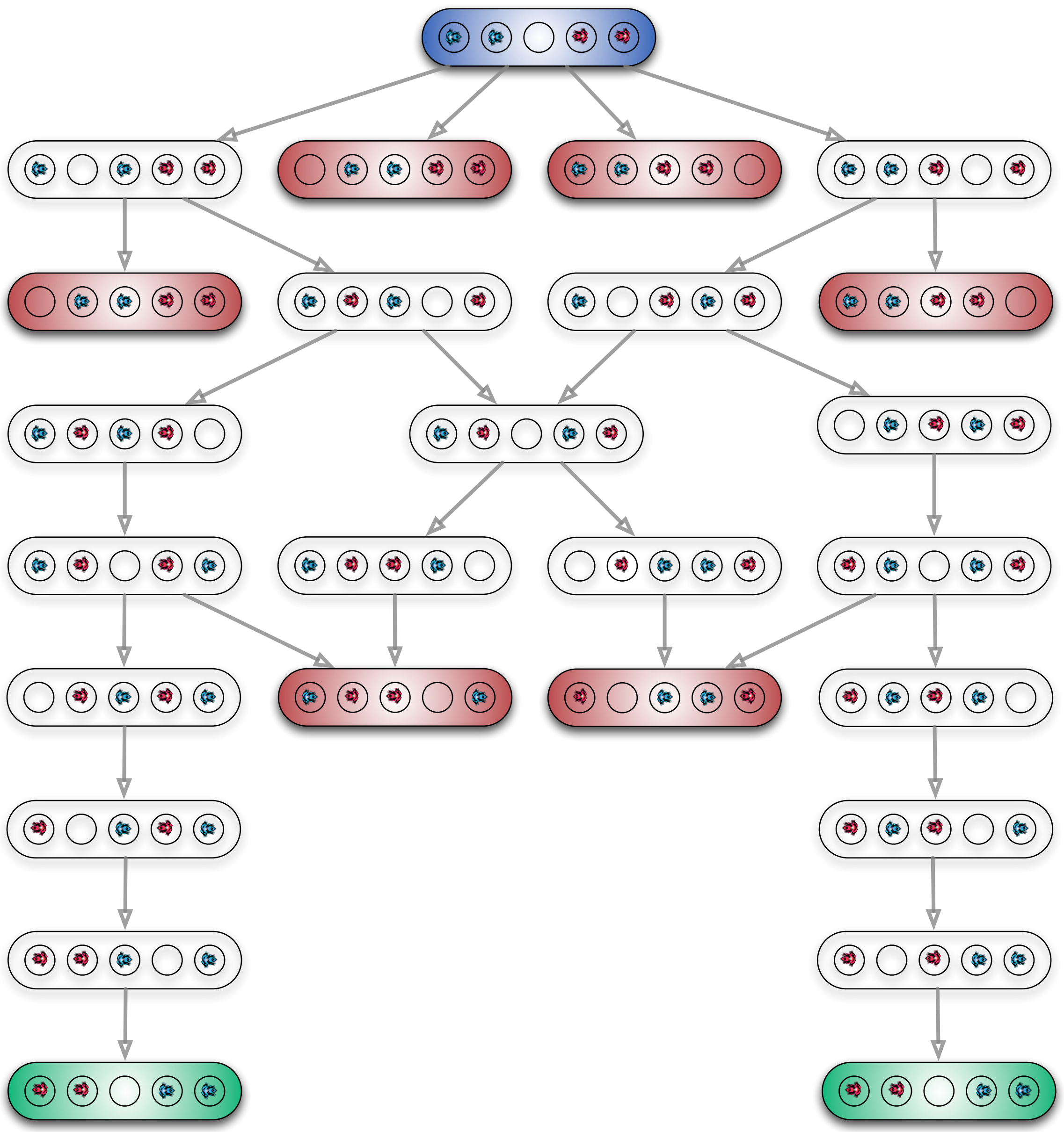
# If the frogs can only move “forwards”, can we:



move from above to below?



# Solution Graph



- So, we have a *finite state-transition diagram* of a *finite state machine (FSM)* as a *complete description* of the behaviour of the four frogs, operating concurrently, no matter what they do according to the rules.

- By examining the FSM, we can state properties as definitely holding, i.e. we can prove properties of the system being modelled.

# Solution Graph

- The solution graph makes it clear that this concurrent system—of four frogs that share certain resources—can experience *deadlock*.
- *Deadlock* occurs when the system arrives in a state from which it can not make any transitions (and which is not a desired end-state.)
- *Livelock* (not possible in this system) is when the system can traverse a sequence of states indefinitely without making progress towards a desired end state.

# Proof

- We can prove interesting properties by trying to construct a state diagram describing
  - each possible state of the system and
  - each possible move from one state to another
- We might use a state diagram to show that
  - A property always holds
  - A property never holds in any reachable state
  - A property sometimes holds
  - A property always holds eventually

# State Diagrams for Processes

- A *state* is defined by a tuple of
  - for each process, the label of the statement available for execution.
  - for each variable, its value.
- Q: What is the maximum number of possible states in such a state diagram?

# Abstraction of Concurrent Programming

- A *concurrent program* is a finite set of [sequential] *processes*.
- A process is written using a *finite set* of *atomic statements*.
- Concurrent program execution is modelled as proceeding by executing a sequence of the atomic statements obtained by *arbitrarily interleaving* the atomic statements of the processes.
- A *computation* [a.k.a. a *scenario*] is one particular execution sequence.

# Atomicity

- We assume that if two operations s1 and s2 really happen at the same time, it's the same as if the two operations happened in either order.
- E.g. simultaneous writes to the same memory locations:

| **Sample** | |
|---|---|
| integer g ← 0; | |
| p | q |
| p1: g ← 2; | q1: g ← 1 |

- We assume that the result will be that g will be 2 or 1 after this program, not, for example, 3.

# Interleaving

- We model a scenario as an arbitrary interleaving of atomic statements, which is somewhat unrealistic.
- For a *concurrent* system, that's OK, it happens anyway.
- For a *parallel* shared memory system, it's OK so long as the previous idea of atomicity holds at the lowest level.
- For a *distributed* system, it's OK if you look at it from an individual node's POV, because either it is executing one of its own statements, or it is sending or receiving a message.
  - Thus *any* interleaving can be used, so long as a message is sent before it is received.

# Level of Atomicity

- The level of atomicity can affect the correctness of a program.

| Example: Atomic Assignment Statements | |
|---|---|
| integer n ← 0; | |
| p | q |
| p1: n ← n+1; | q1: n ← n+1; |

Value **before** atomic statement on same row

| process p | process q | n |
|---|---|---|
| **p1: n ← n+1;** | q1: n ← n+1; | 0 |
| (end) | **q1: n ← n+1;** | 1 |
| (end) | (end) | 2 |

| process p | process q | n |
|---|---|---|
| p1: n ← n+1; | **q1: n ← n+1;** | 0 |
| **p1: n ← n+1;** | (end) | 1 |
| (end) | (end) | 2 |

# Different Level of Atomicity

| **Example: Assignment Statements with one Global Reference** | |
|---|---|
| integer n ← 0; | |
| p | q |
| integer temp | integer temp |
| p1: temp ← n | q1: temp ← n |
| p2: n ← temp + 1 | q2: n ← temp + 1 |

# Alternative Scenarios

| process p | process q | n | p.temp | q.temp |
|---|---|---|---|---|
| **p1: temp ← n** | q1: temp ← n | 0 | ? | ? |
| **p2: n ← temp+1** | q1: temp ← n | 0 | 0 | ? |
| (end) | **q1: temp ← n** | 1 | | ? |
| (end) | **q2: n ← temp+1** | 1 | | 1 |
| (end) | (end) | 2 | | |

| process p | process q | n | p.temp | q.temp |
|---|---|---|---|---|
| **p1: temp ← n** | q1: temp ← n | 0 | ? | ? |
| p2: n ← temp+1 | **q1: temp ← n** | 0 | 0 | ? |
| **p2: n ← temp+1** | q2: n ← temp+1 | 0 | 0 | 0 |
| (end) | **q2: n ← temp+1** | 1 | | 0 |
| (end) | (end) | 1 | | |

# Atomicity & Correctness

- Thus, the level of atomicity specified affects the correctness of a program
  - We will typically assume that:
    - assignment statements and
    - condition statement evaluations
- are atomic
  - Note: this is not true for programs written in C using concurrency libraries like `pthreads` or similar.

# Concurrent Counting Algorithm

| Example: Concurrent Counting Algorithm | |
|---|---|
| integer n ← 0; | |
| **p** | **q** |
| integer temp | integer temp |
| p1: do 10 times | q1: do 10 times |
| p2: temp ← n | q2: temp ← n |
| p3: n ← temp + 1 | q3: n ← temp + 1 |

- Increments a global variable *n* 20 times, thus *n* should be 20 after execution.
- But, the program is faulty.
  - Proof: construct a scenario where *n is 2* afterwards.
- Wouldn't it be nice to get a program to do this analysis?