

Concurrency First

... but we'd better get it right !

Peter Welch and Fred Barnes
School of Computing, University of Kent, UK

`{phw, frmb}@kent.ac.uk`

CPA 2009 *Fringe*, Eindhoven, 1st. November

Concurrency and Computer Systems

We see concurrency as a fundamental mechanism of the universe, existing in all structures and at all levels of granularity.

To be useful in this universe, any computer system has to model and reflect an appropriate level of abstraction.

For simplicity, therefore, the system must be *concurrent* – so that this modelling is obvious and correct.

Yet concurrency is thought to be an *advanced* topic, *harder* than serial computing (which therefore needs to be mastered first).



This tradition makes no sense ...

... which has (*radical*) implications on how we should educate people for computer science ...

... and on how we apply what we have learnt ...



This tradition makes no sense ...

So, here's how we *promote* (and *demote*) concurrency at Kent, where we have been running a full undergraduate module (20-25 lectures) on concurrency for 23 years ...

The next 5 slides are *promotional* (and *demotional*) material presented to first year students.

Although we would like to offer this in Term One, our modules (Co538 / Co632) are, *currently*, elective options taken by second and final year students. ☹️



This tradition makes no sense ...

So, here's how we **promote** (and **demote**) concurrency at Keble
we have been running a full undergraduate module for
on concurrency for 23 years ...

The next 5 slides are **promoted** (and **demoted**) material presented
to first year students.

This academic year (2009/10), we have a 30% uptake – that's
around 60 out of 200 CS students, across both available years.
(2005/06 to 2005/06) are – **currently** – elective options taken by second
and final year students.



(Co538) Concurrency – Design & Practice

Concurrency is *many things happening at the same time*:

- so is the real world – and computers, to be useful, have to model relevant bits of it;
- it's needed to support multiple demands (*e.g. internet services, games, robotics, graphics/GUIs, mobile phones, bio-systems experiments, big physics modelling, real-time control, operating systems, ...*), even when running on a single processor;
- it's needed to exploit *multicore* and *multiprocessor* systems;
- it's needed for *distributed* systems and *supercomputing*;
- it's needed for *hardware design, implementation* and *operation*.

... core Computer Science



(Co538) Research Engagement

a language for *concurrency*

occam- π

**JCSP / CHP /
C++CSP**

concurrency libraries for
Java / Haskell / C++

(Co538) Research Engagement

For the past 20 years, this department has been a leading centre of research into the theory and (*especially*) the practice of **concurrency**.

Programming technologies have been (are being) developed here:

- **occam- π** (an industrial strength programming language based on the formal process algebras of **CSP** and the **π -calculus**);
- **JCSP** (a 100% pure Java library providing an API that supports the same concurrency model as **occam- π**);
- **C++CSP / CHP** (a 100% pure C++ / Haskell library providing an API that supports the same concurrency model as **occam- π**).

This module will teach this model ***though the programming technologies*** (we won't be doing the formal mathematics).

There will be lots of ***programming*** in this module. 😊😊😊😊😊😊😊😊😊😊😊😊😊😊😊😊

(Co538) Research Engagement

Key new concepts:

processes (components, water-tight and alive)

synchronised communication (over channels)

multitway synchronisation (over barriers)

networks (processes connected by channels and barriers)

structure (networks within networks)

choice (waiting for and reacting to events)

dynamics (run-time network construction and re-shaping)

mobility (agents)

(Co632) Advanced Concurrency – Design & Practice

Concurrency is *many things happening at the same time*.

This module introduces *dynamics* – the construction, evolution and termination of systems (or sub-systems) *on-the-fly*.

This is needed for systems that:

- scale with demand (*e.g. web services, air-traffic control*);
- evolve with demand (*e.g. peer-to-peer networking*);
- model growing organisms (*e.g. nanite assemblies*);
- configure, load and run supercomputer resources (*e.g. Grid computing, our TUNA and CoSMoS clusters*).

What we Need from Concurrency

- A powerful tool for *simplifying* the description of systems.
- *Performance* that spins out from the above, but is *not* the primary focus.
- A model of concurrency that is *mathematically clean*, yields no engineering surprises and scales well with system complexity.
- The good news is that *we can have it all* – and we don't need to understand the maths! It's burnt into the *languages* (or *libraries*) supporting the model. And we use *diagrams*.

CSP and π -calculus

CSP and the **π -calculus** are *process algebrae* – mathematical theories for specifying and verifying patterns of behaviour arising from interactions between concurrent objects.

CSP has a *formal*, and *compositional*, semantics that is *in line with our informal intuition about the way such things work*.

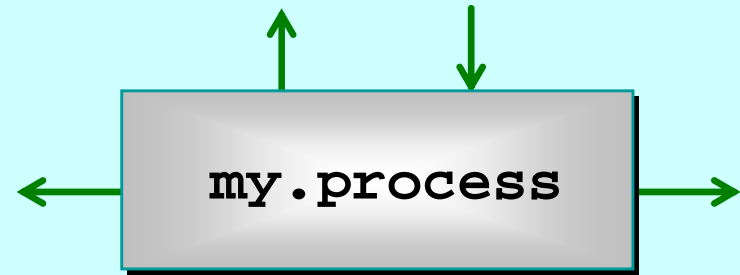
Claim

The **π -calculus** has a *formal*, and *operational*, semantics that is *in line with our informal intuition about the way networks can be constructed, and taken down, dynamically*.

Claim

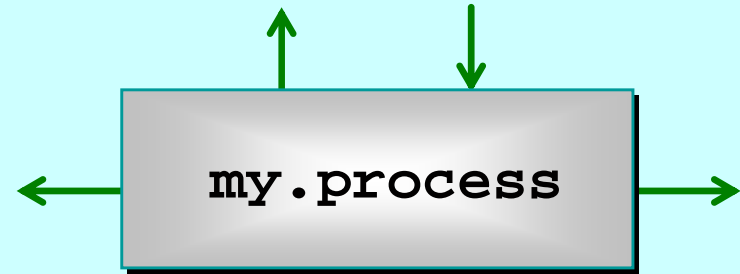
We do not need to be mathematically sophisticated to work with them. *The sophistication is pre-engineered into the model.* We benefit simply by using it.

Processes

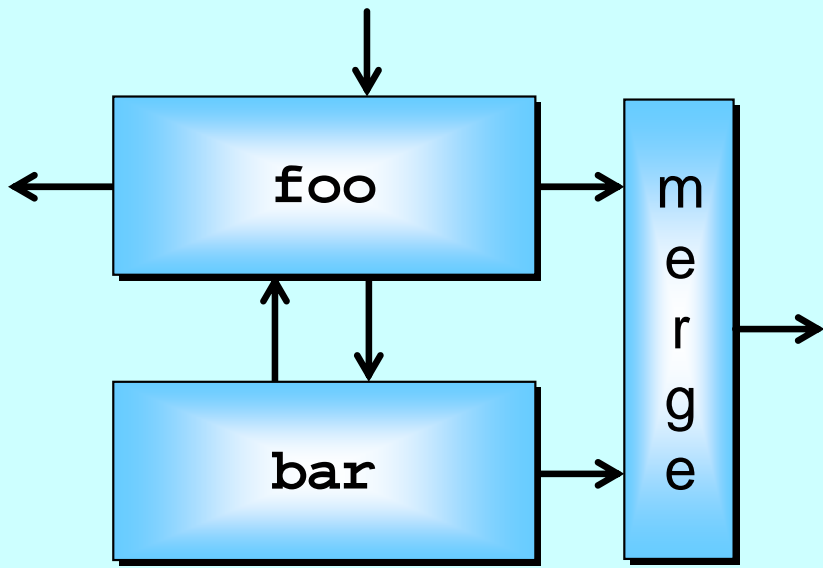


- A **process** is a component that encapsulates some data structures and algorithms for manipulating that data.
- Both its data and algorithms are **private**. The outside world can neither see that data nor execute those algorithms! *[They are not objects ...] [Think chips ...]*
- The algorithms are executed by the process in its own thread (or threads) of control. Each process is **alive**.
- So, how does one process interact with another?
- Well, they communicate ...

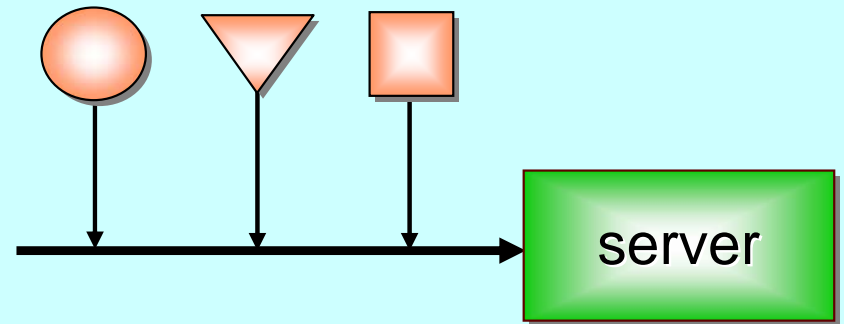
Processes



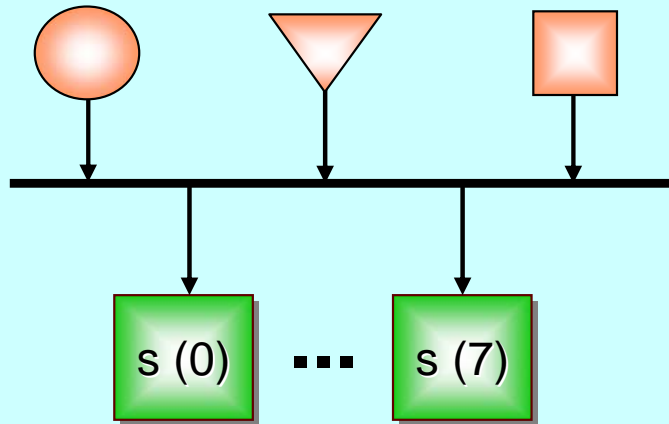
- The simplest form of interaction is **synchronised** message-passing along **channels**.
- The simplest forms of channel are **zero-buffered** and **point-to-point** (i.e. **wires**).
- But, we can have **buffered** channels (**blocking** / **overwriting**).
- And **any-1**, **1-any** and **any-any** channels.
- And **multi-way synchronisation** (**barriers**) ...
- And more ...



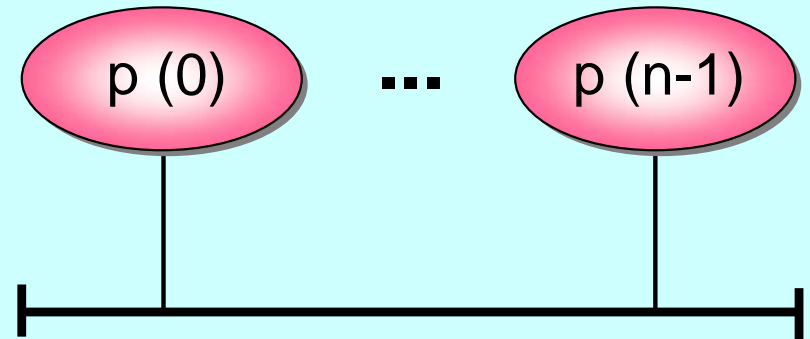
(a) a network of three processes, connected by four internal (hidden) and three external channels.



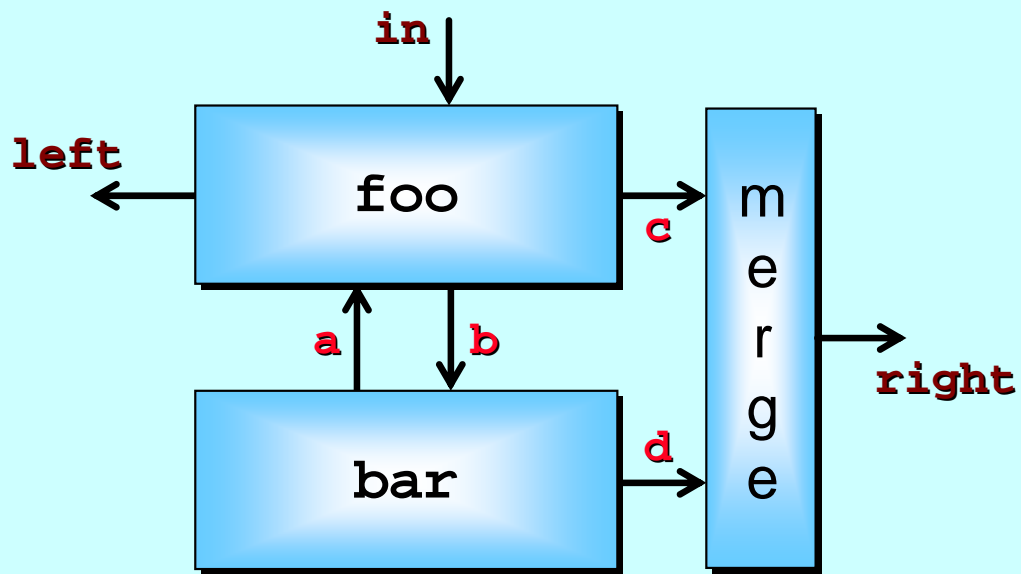
(b) three processes sharing the writing end of a channel to a server process.



(c) three processes sharing the writing end of a channel to a bank of servers sharing the reading end.



(d) n processes enrolled on a shared barrier (any process synchronising must wait for all to synchronise).



(a) a network of three processes, connected by four internal (hidden) and three external channels.

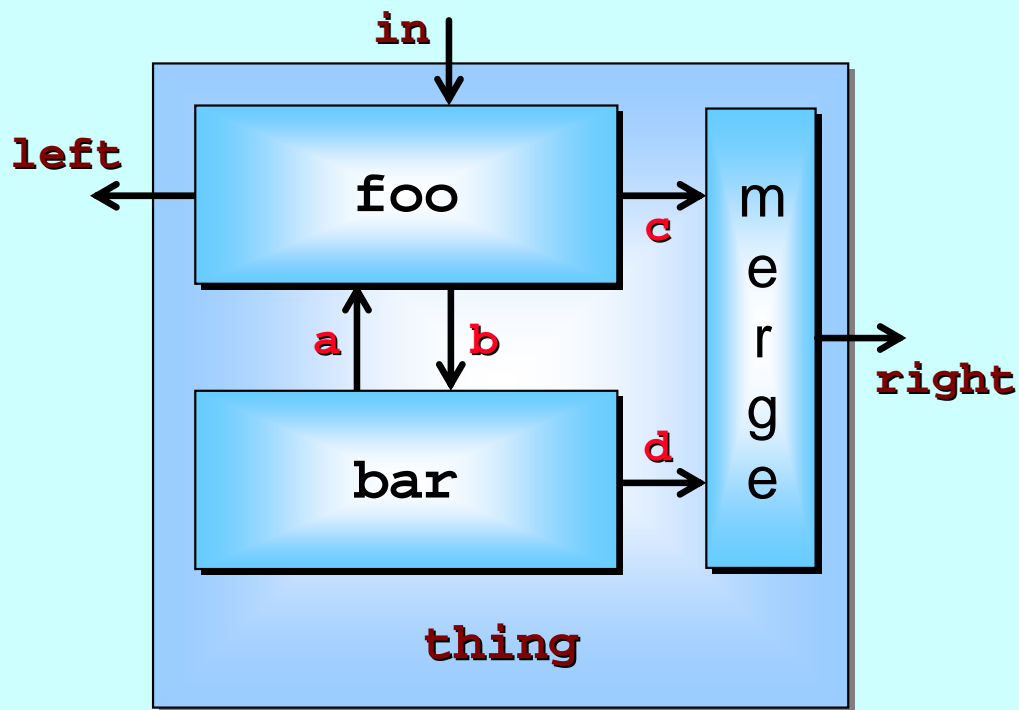
```
CHAN BYTE a, b, c, d:
```

```
PAR
```

```
  foo (in?, left!, a?, b!, c!)
```

```
  bar (a!, b?, d!)
```

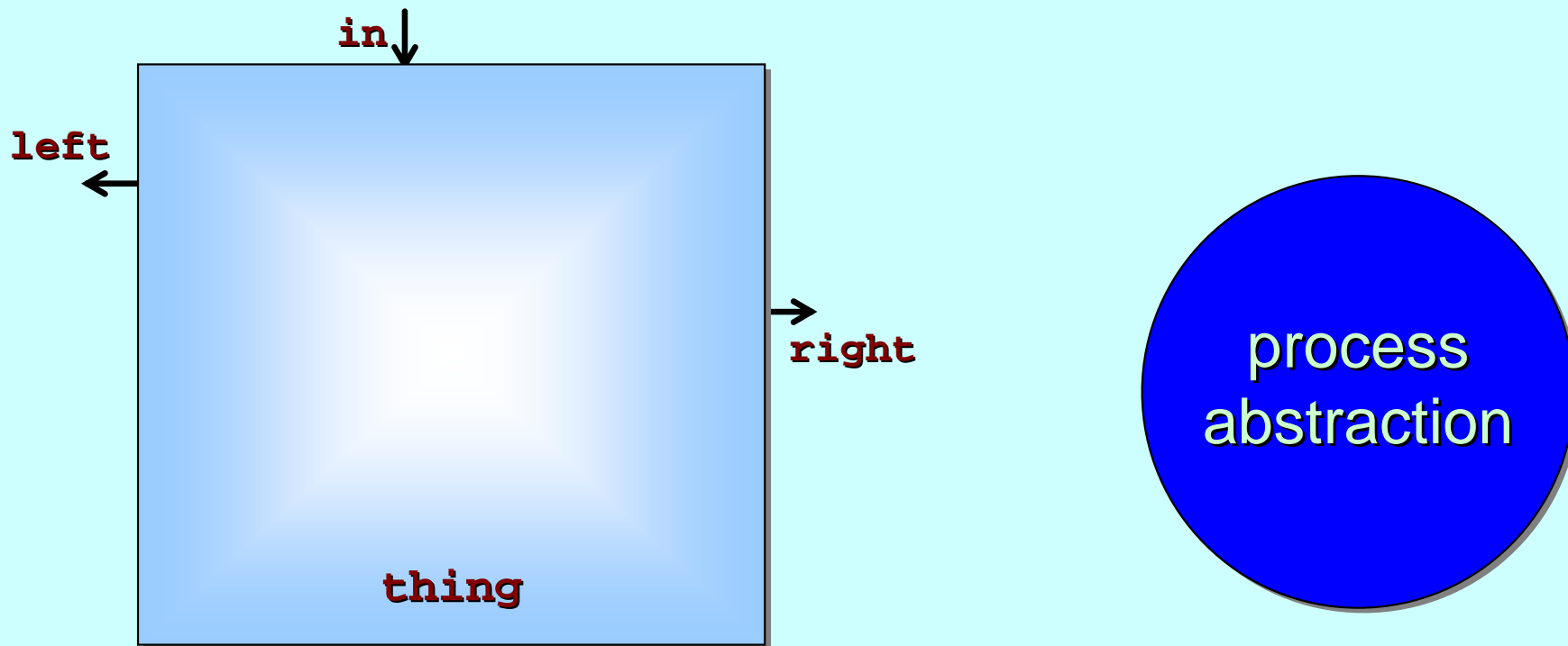
```
  merge (c?, d?, right!)
```

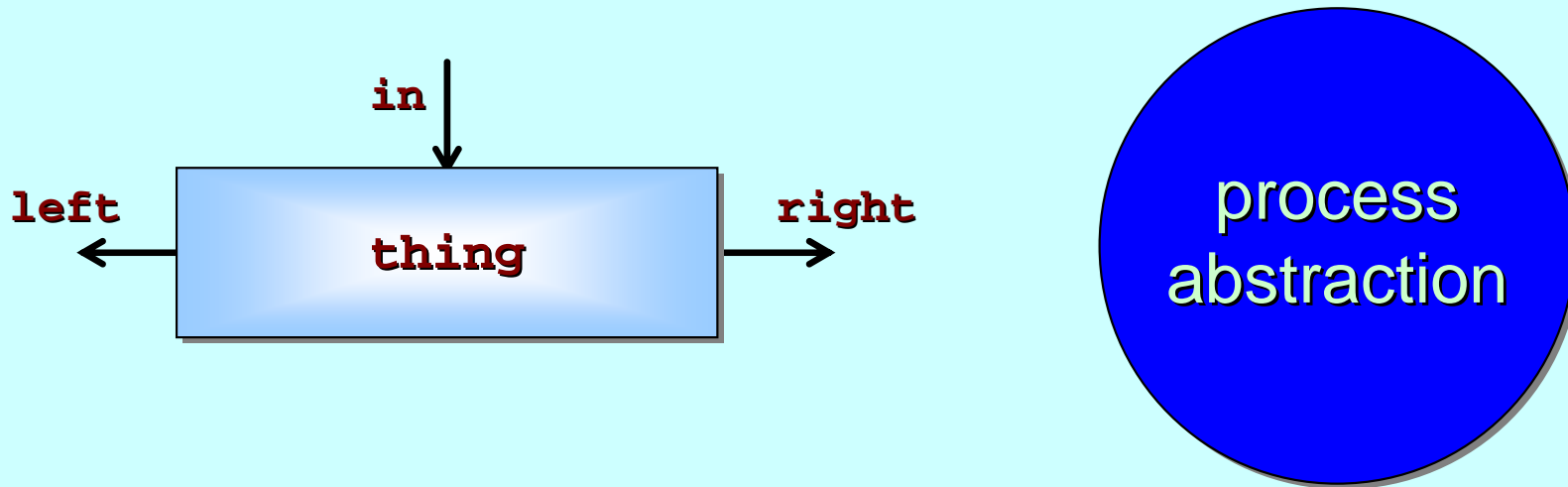
```

PROC thing (CHAN INT in?, left!, right!)
  CHAN BYTE a, b, c, d:
  PAR
    foo (in?, left!, a?, b!, c!)
    bar (a!, b?, d!)
    merge (c?, d?, right!)
  :

```



```
PROC thing (CHAN INT in?, left!, right!)  
  CHAN BYTE a, b, c, d:  
  PAR  
    foo (in?, left!, a?, b!, c!)  
    bar (a!, b?, d!)  
    merge (c?, d?, right!)  
  :
```

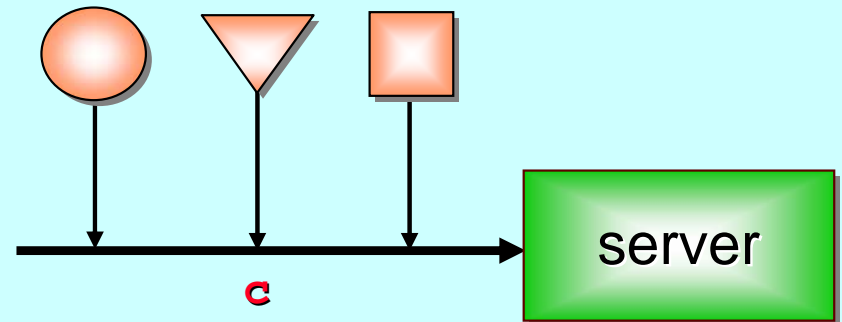


```
PROC thing (CHAN INT in?, left!, right!)
```

Like **foo**, **bar** and **merge** previously, **thing** is a process that can be used as a component in another network.

Concurrent systems have structure – networks within networks.

We must be able to express this! And we can ... 😊 😊 😊



(b) three processes sharing the writing end of a channel to a server process.

```
SHARED ! CHAN SOME.SERVICE c:
```

```
PAR
```

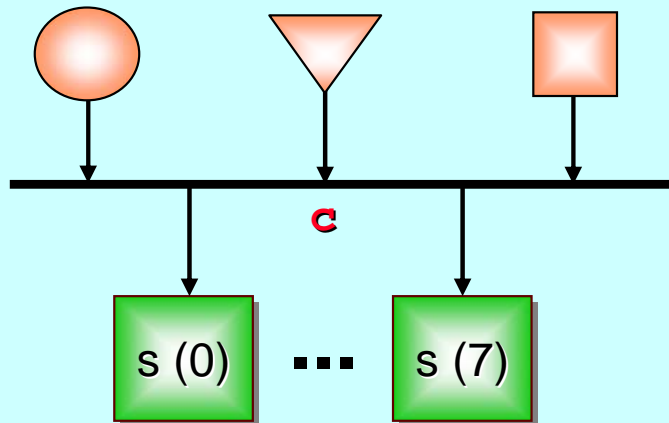
```
  circle (c!)
```

```
  triangle (c!)
```

```
  square (c!)
```

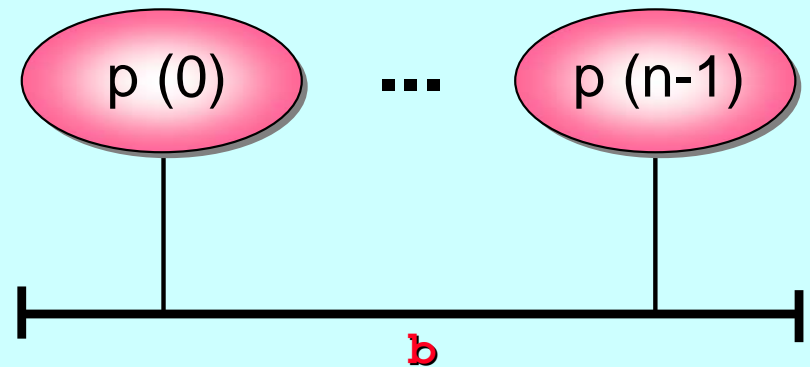
```
  server (c?)
```

```
SHARED CHAN ANOTHER.SERVICE c:  
PAR  
  PAR  
    circle (c!)  
    triangle (c!)  
    square (c!)  
  PAR i = 0 FOR 8  
    s (i, c?)
```



(c) three processes sharing the writing end of a channel to a bank of servers sharing the reading end.

```
BARRIER b:  
PAR i = 0 FOR n ENROLL b  
  p (i, b)
```



(d) n processes enrolled on a shared barrier (any process synchronising must wait for all to synchronise).

Good News!

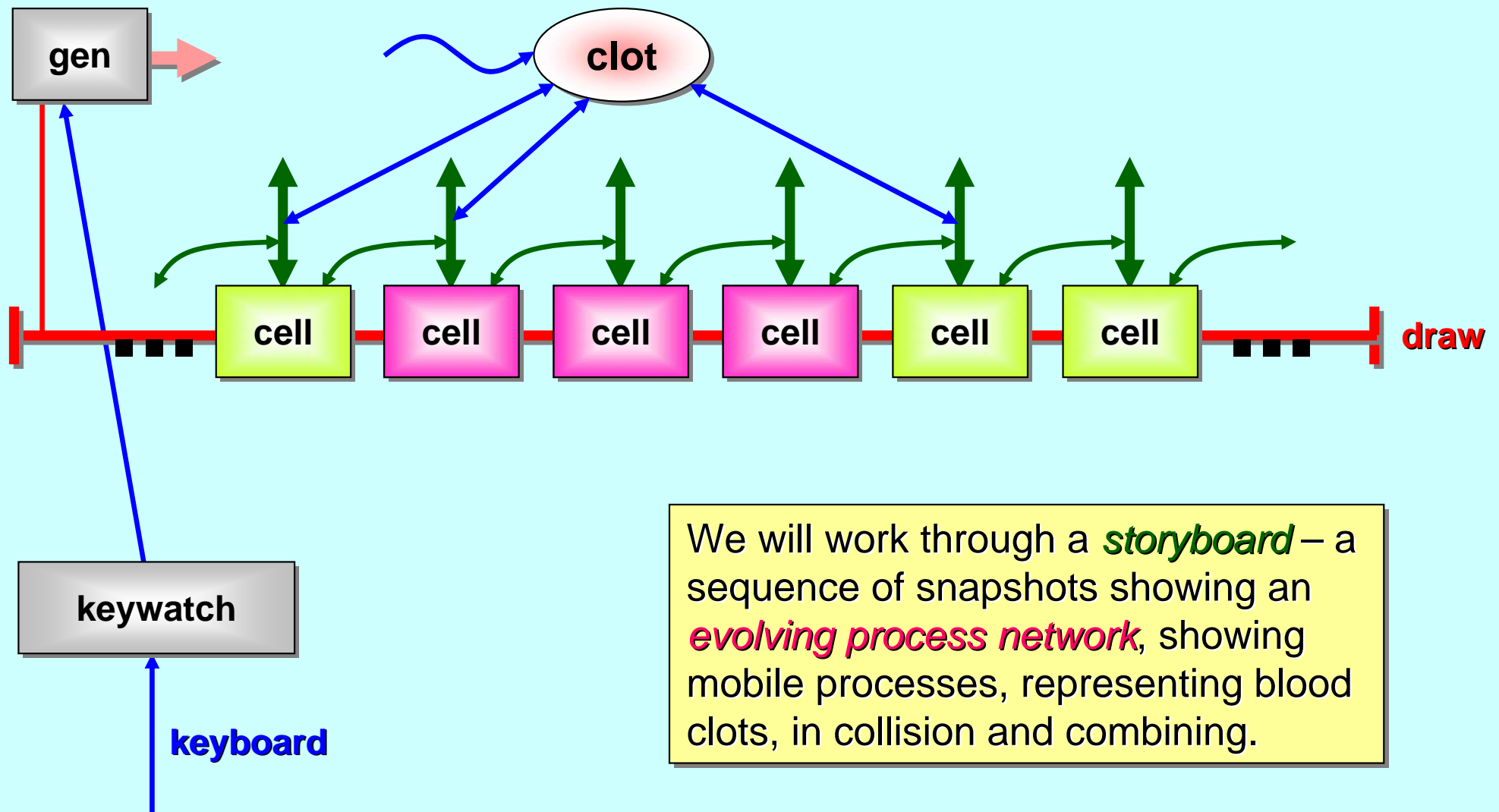
The good news is that we can worry about each process on its own. *A process interacts with its environment through its channels and barriers. It does not interact directly with other processes.*

Some processes have *serial* implementations - *these are just like traditional serial programs.*

Some processes have *parallel* implementations - *networks of sub-processes (think hardware).*

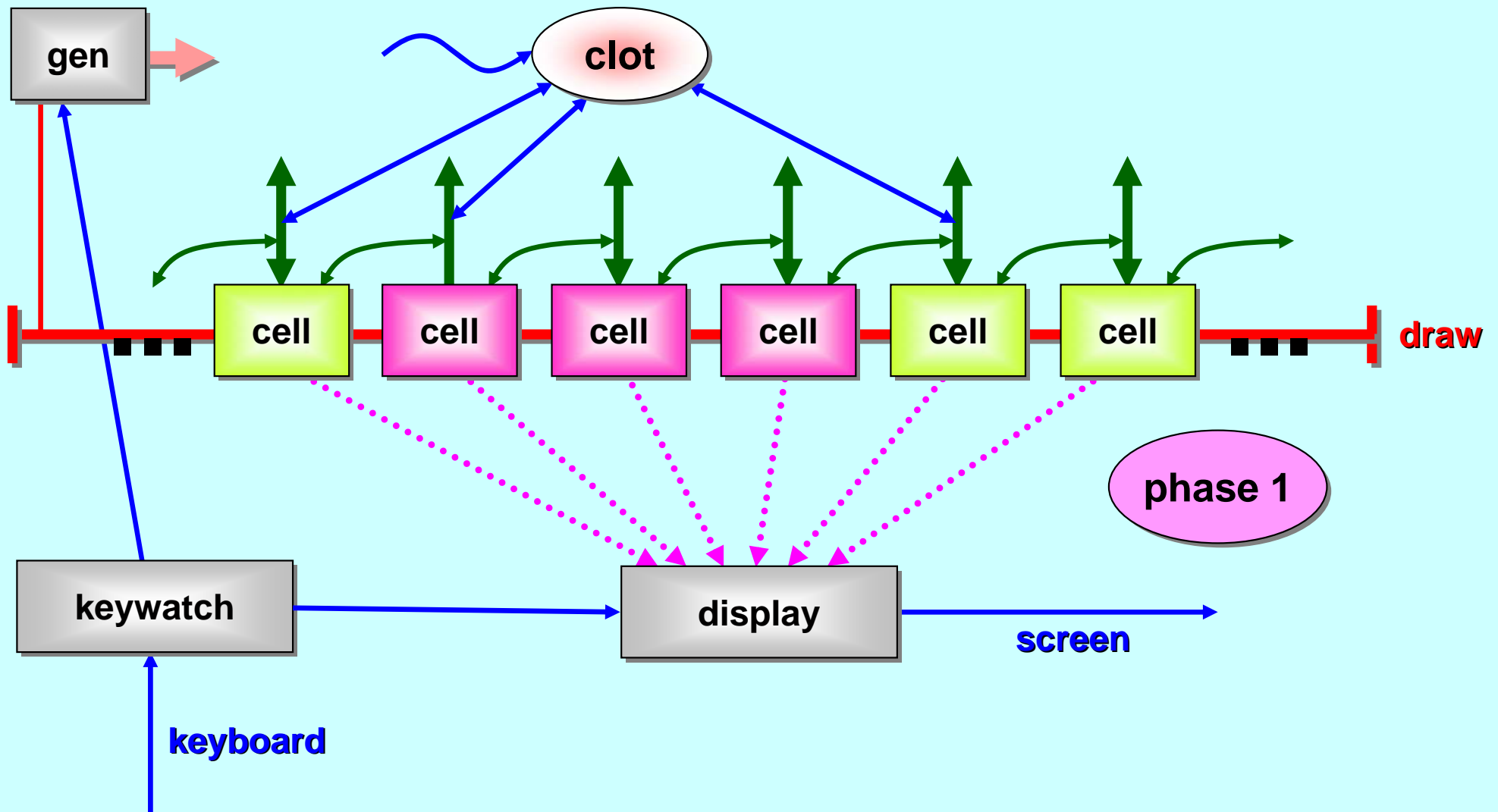
Our skills for serial logic sit happily alongside our new skills for concurrency - there is no conflict. *There will be no race hazards. This will scale!*

Blood Platelet Model

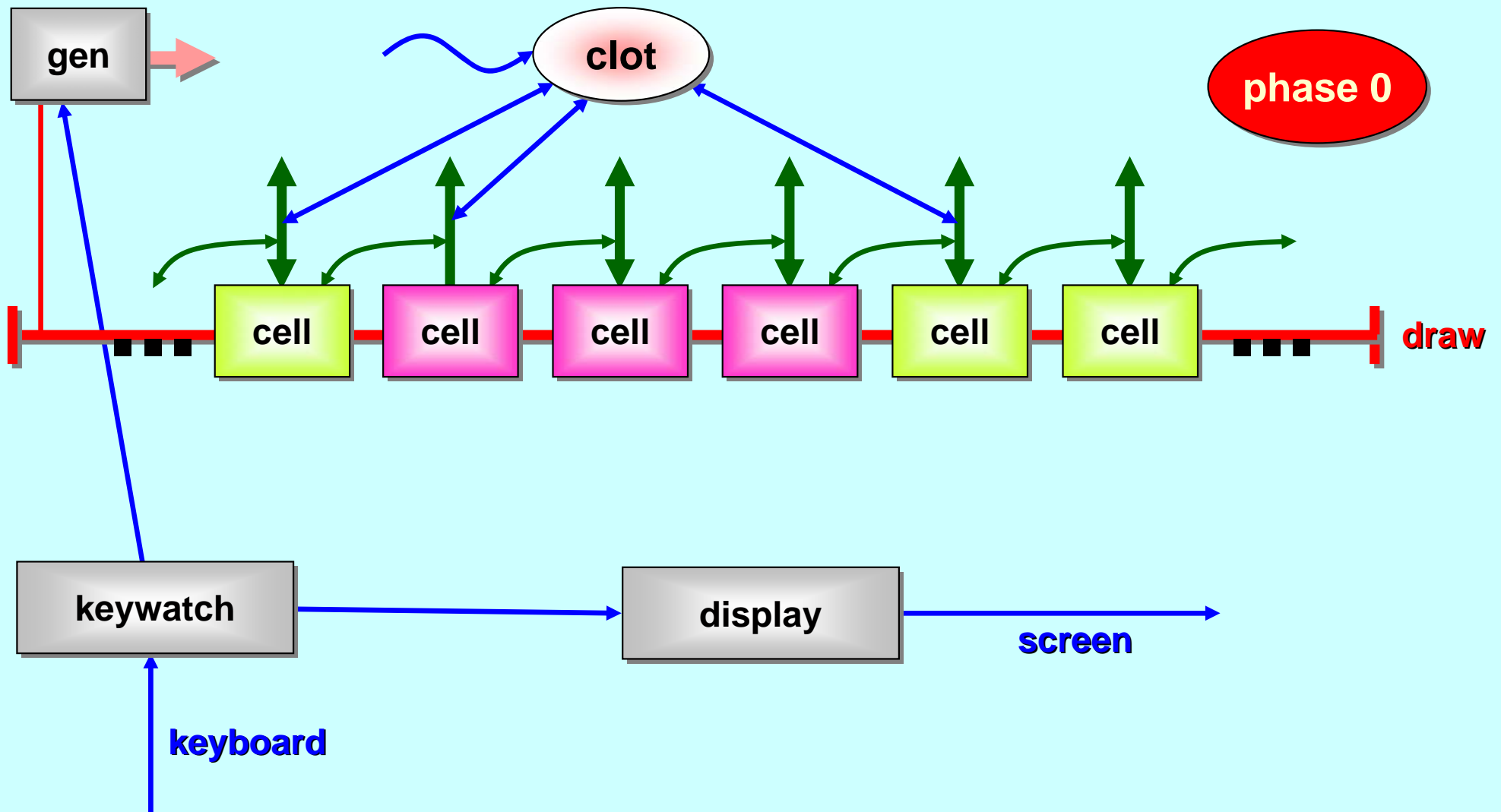


We will work through a *storyboard* – a sequence of snapshots showing an *evolving process network*, showing mobile processes, representing blood clots, in collision and combining.

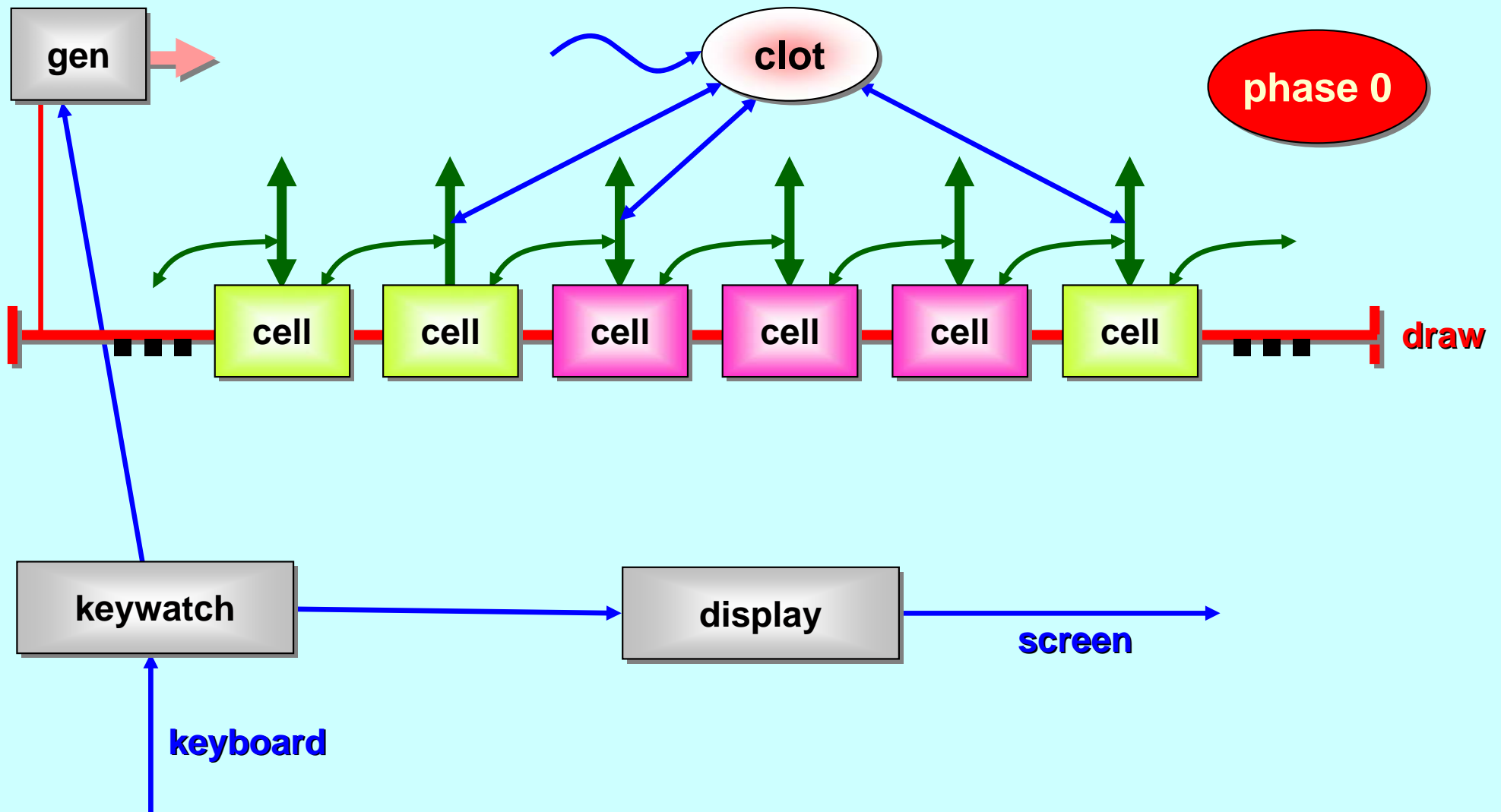
Blood Platelet Model



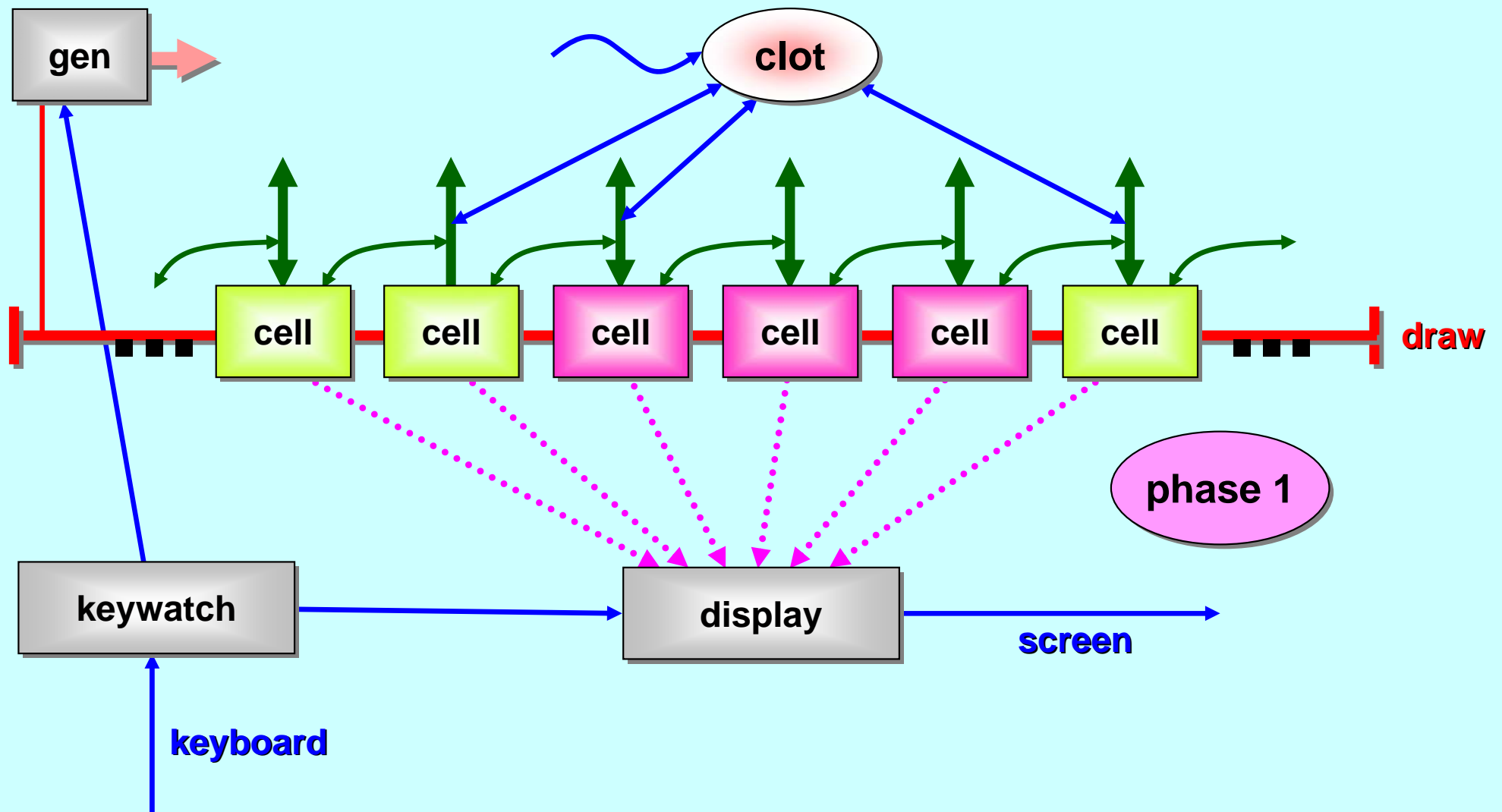
Blood Platelet Model



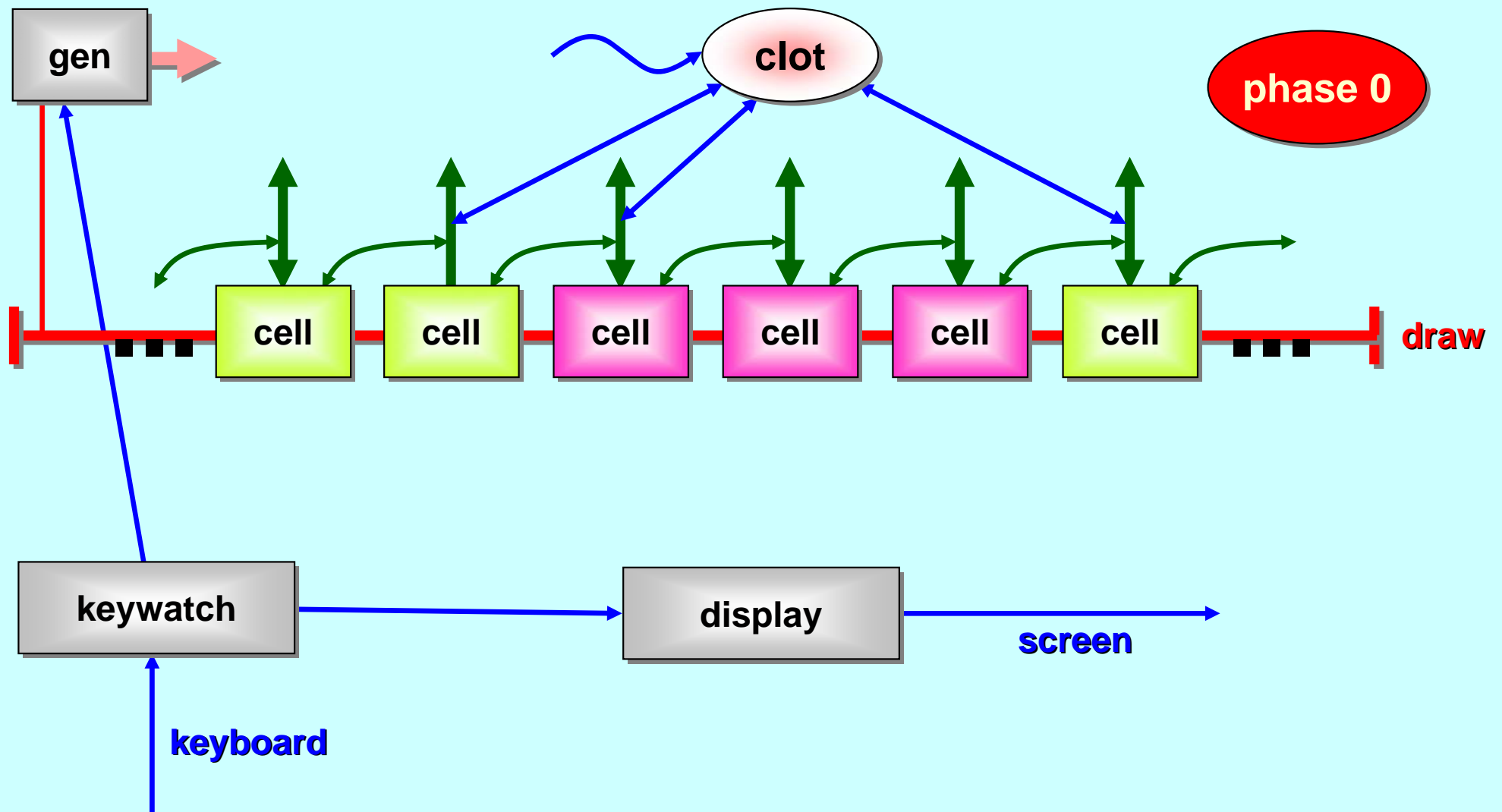
Blood Platelet Model



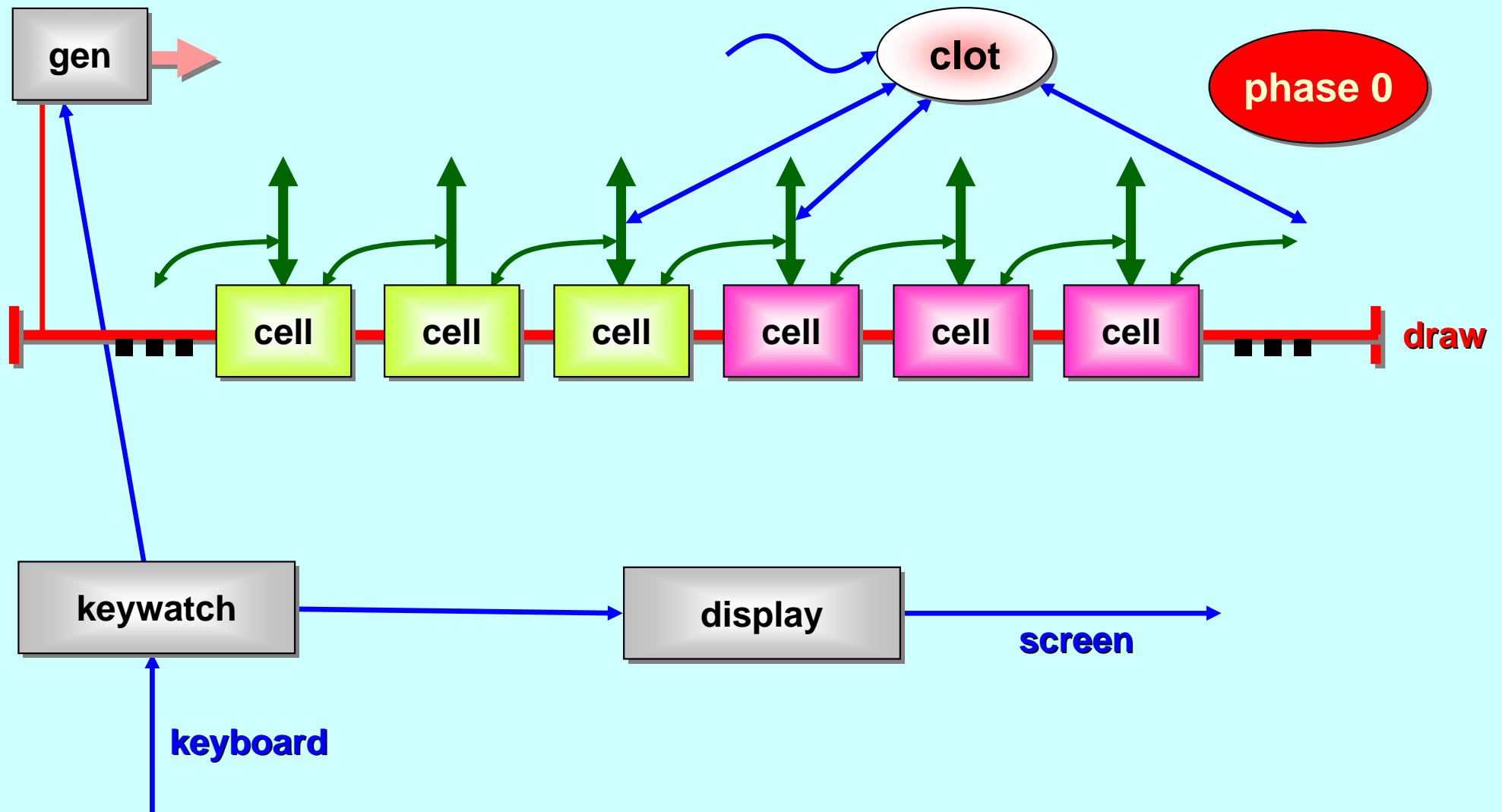
Blood Platelet Model



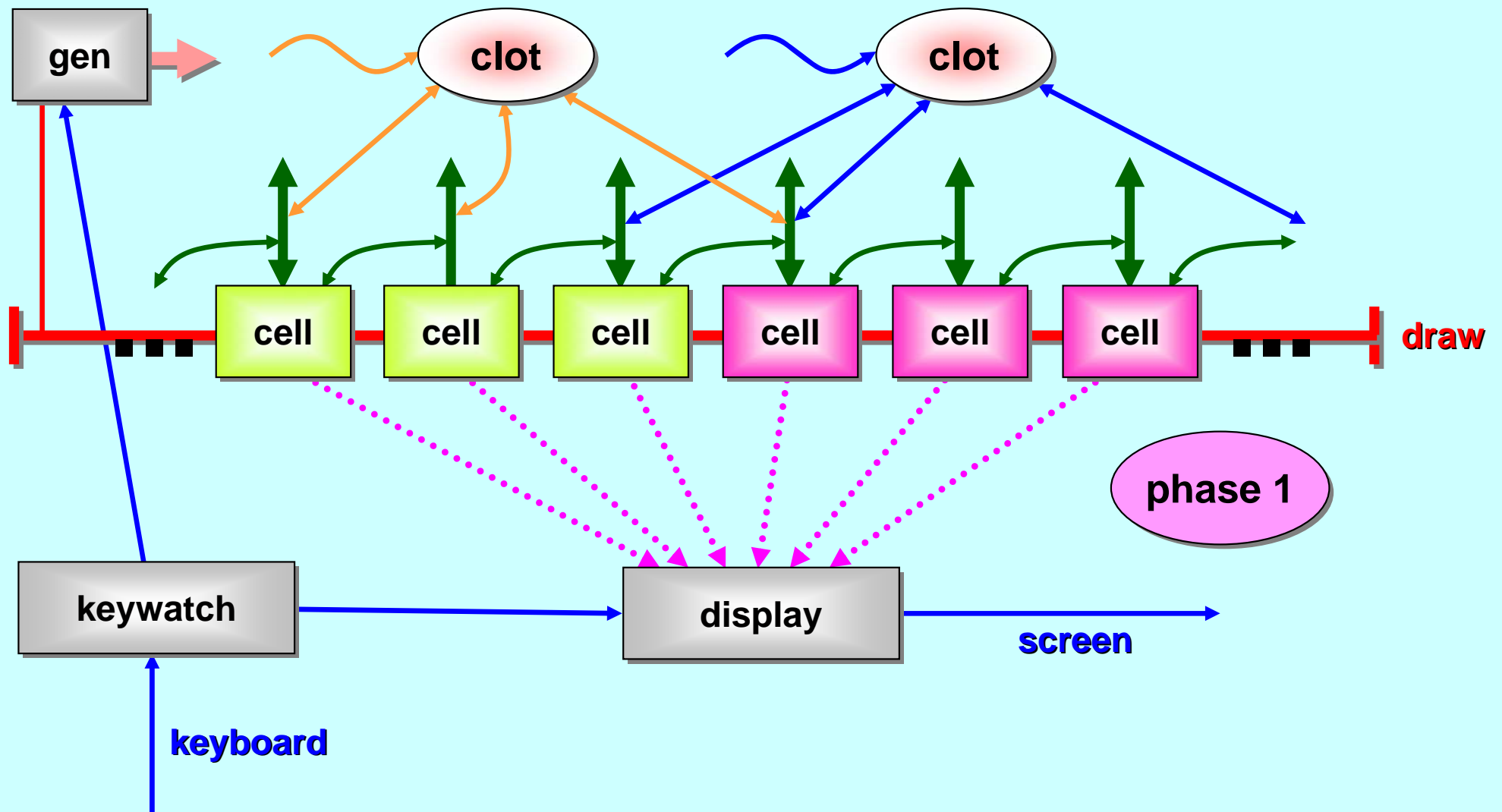
Blood Platelet Model



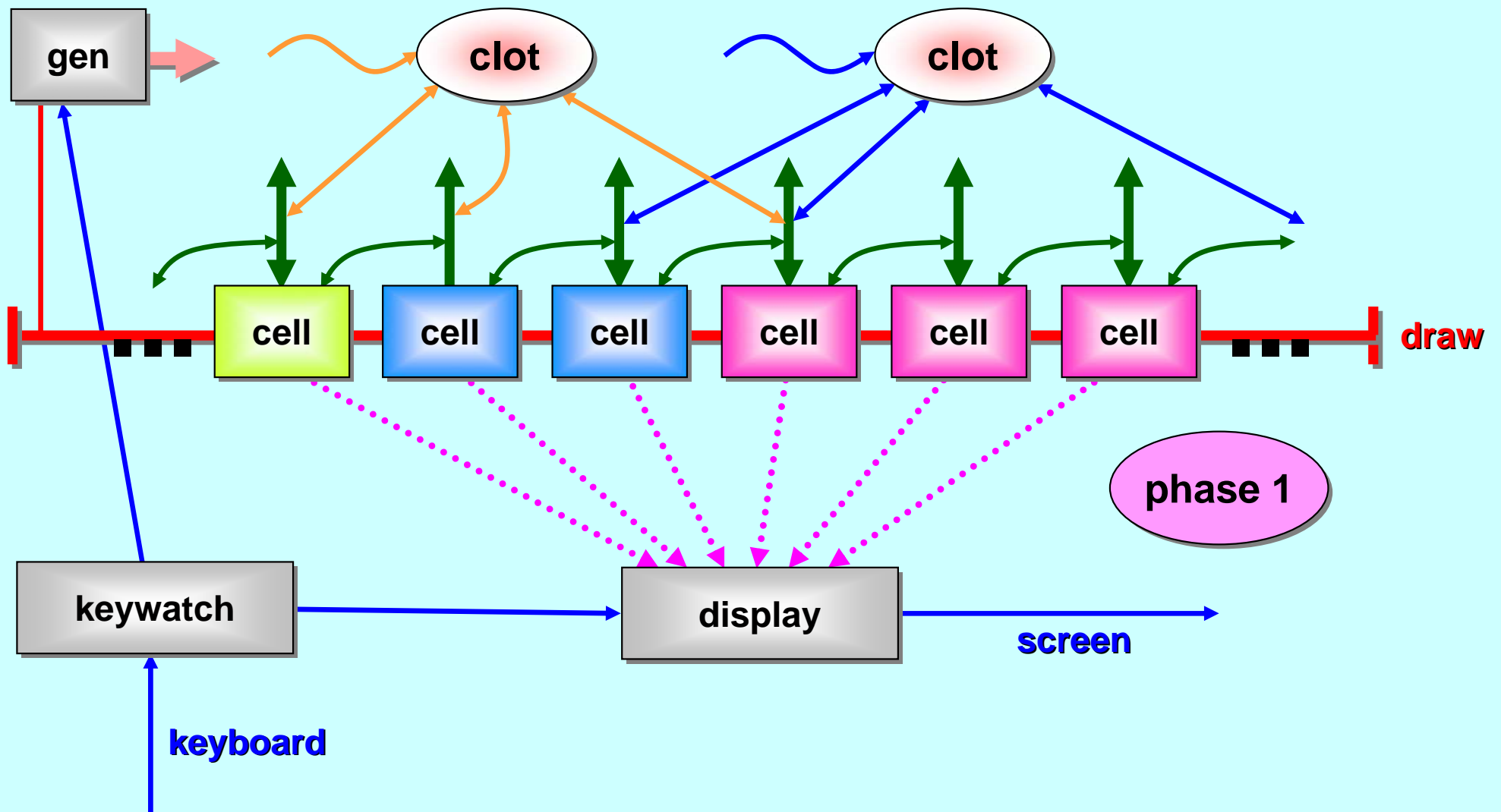
Blood Platelet Model



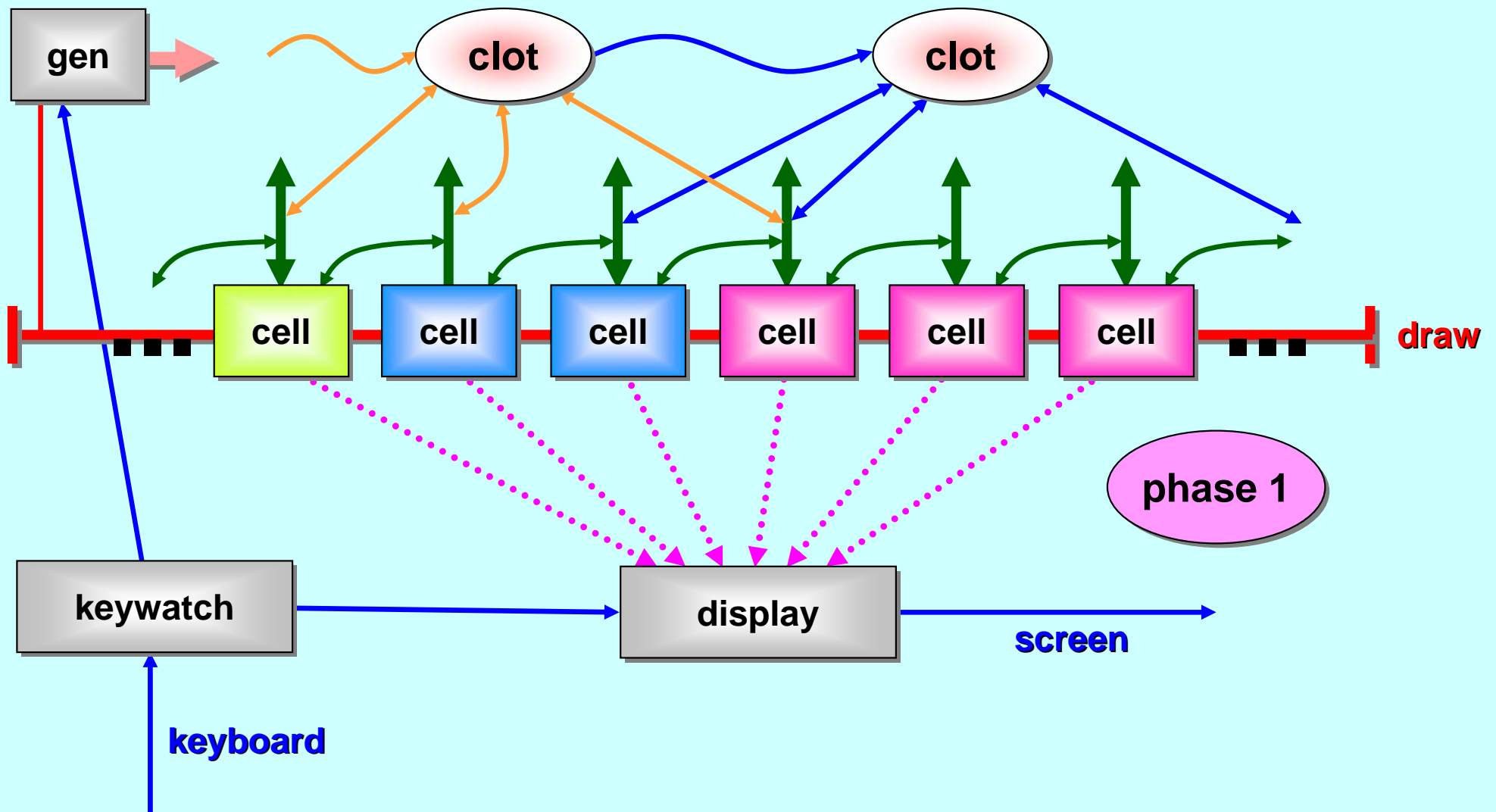
Blood Platelet Model



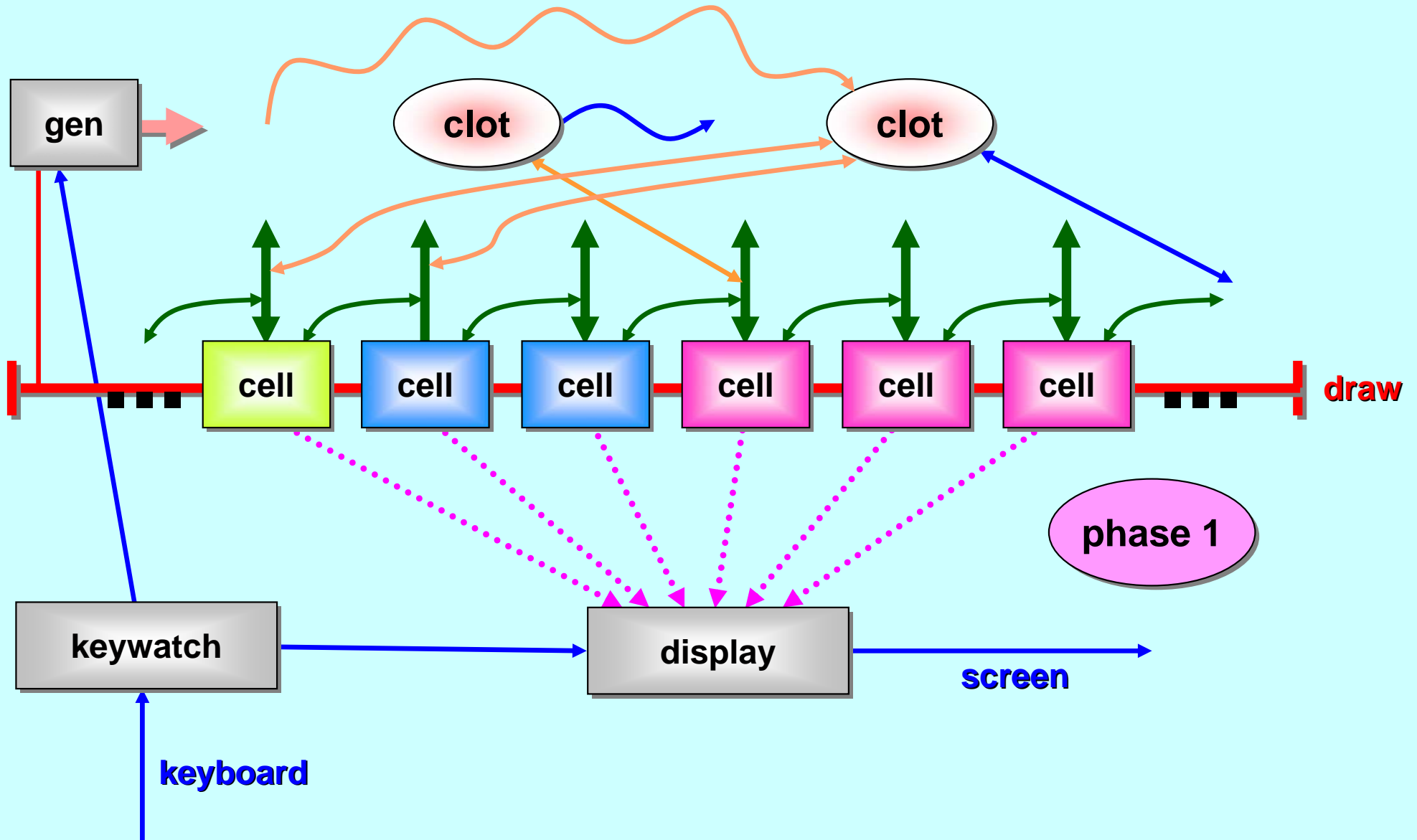
Blood Platelet Model



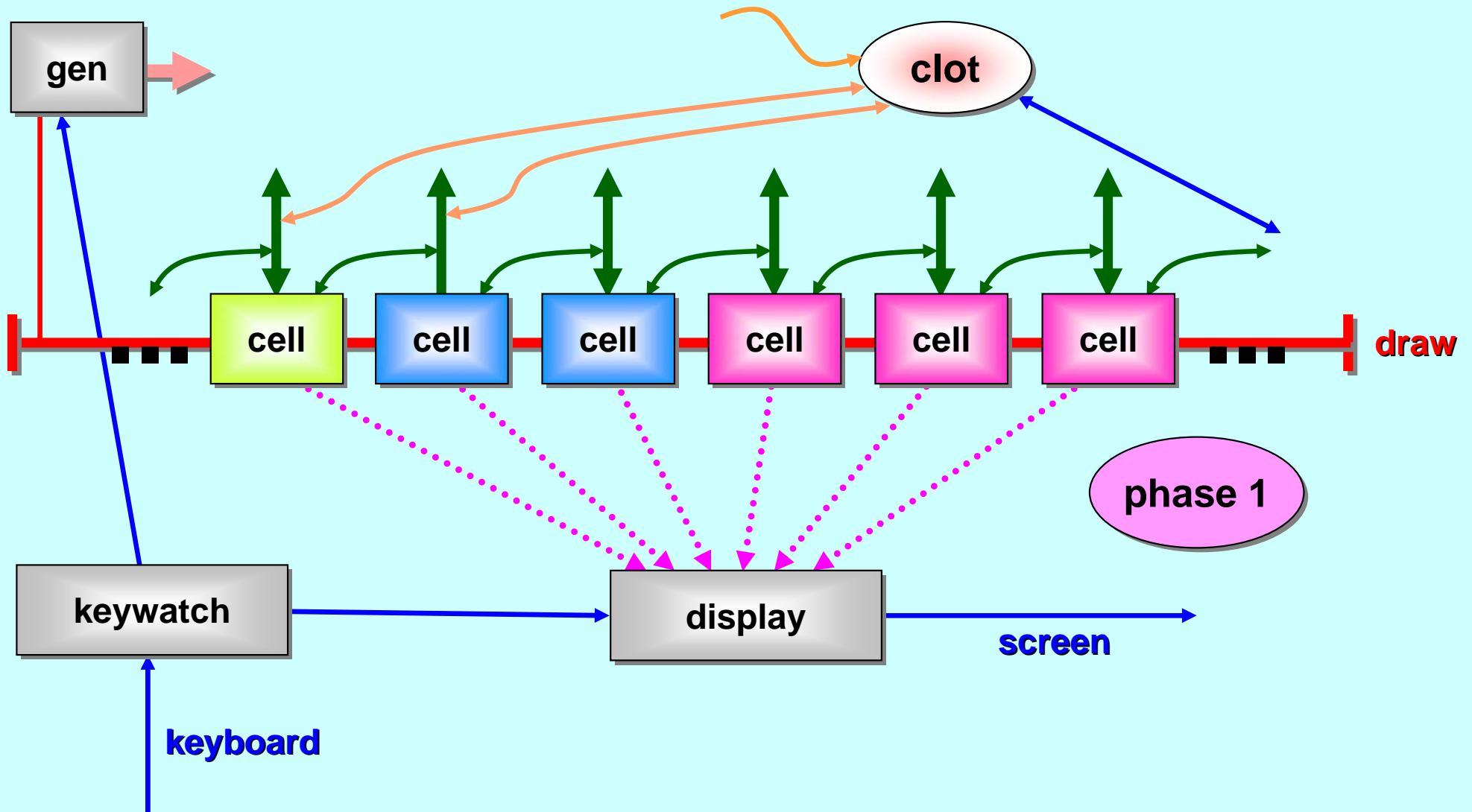
Blood Platelet Model



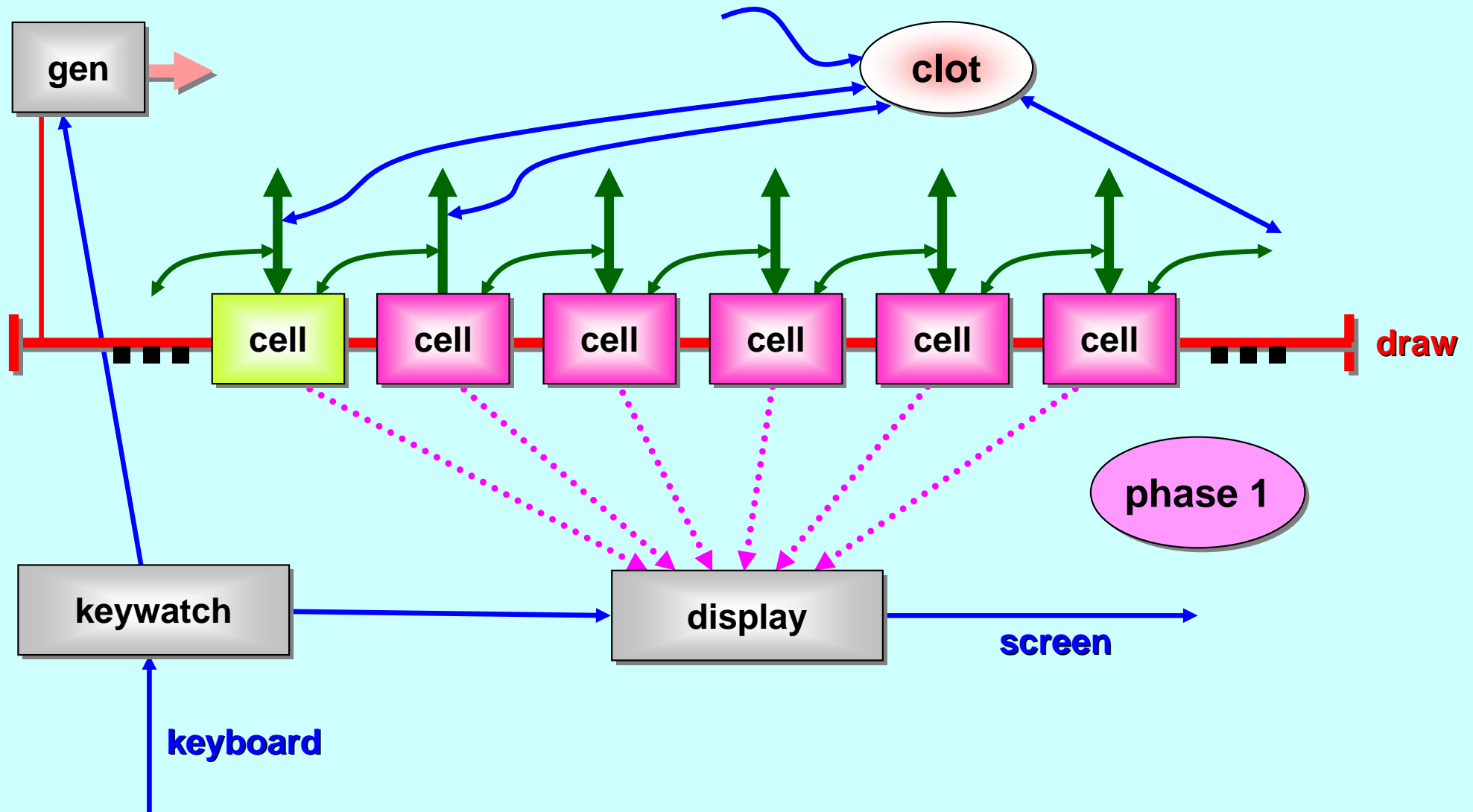
Blood Platelet Model



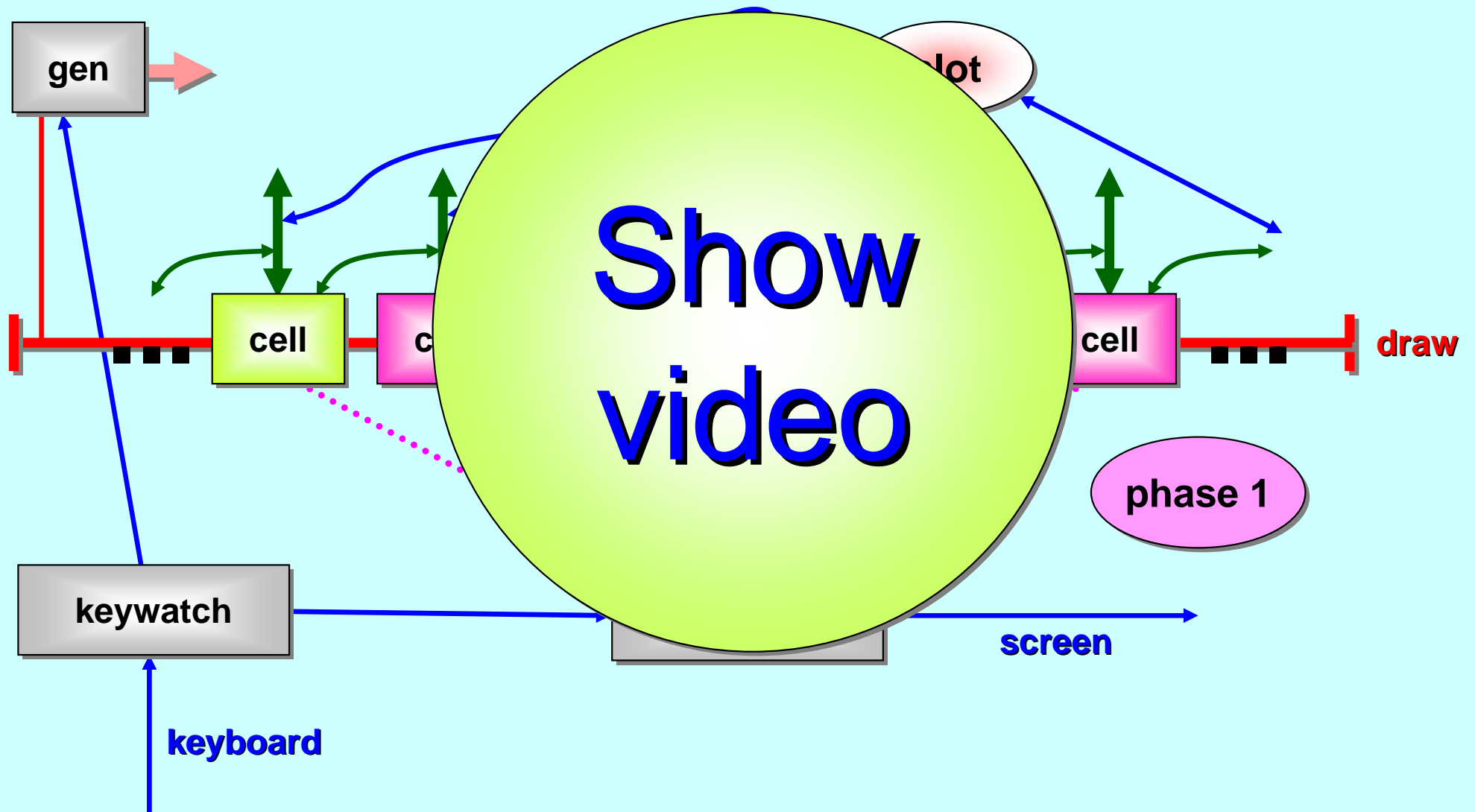
Blood Platelet Model



Blood Platelet Model



Blood Platelet Model



Blood Platelet Model



Show
video

https://www.cs.kent.ac.uk/research/groups/plas/wiki/3D_Blood_Clotting

https://www.cs.kent.ac.uk/research/groups/plas/wiki/3D_Blood_Clotting_Gallery

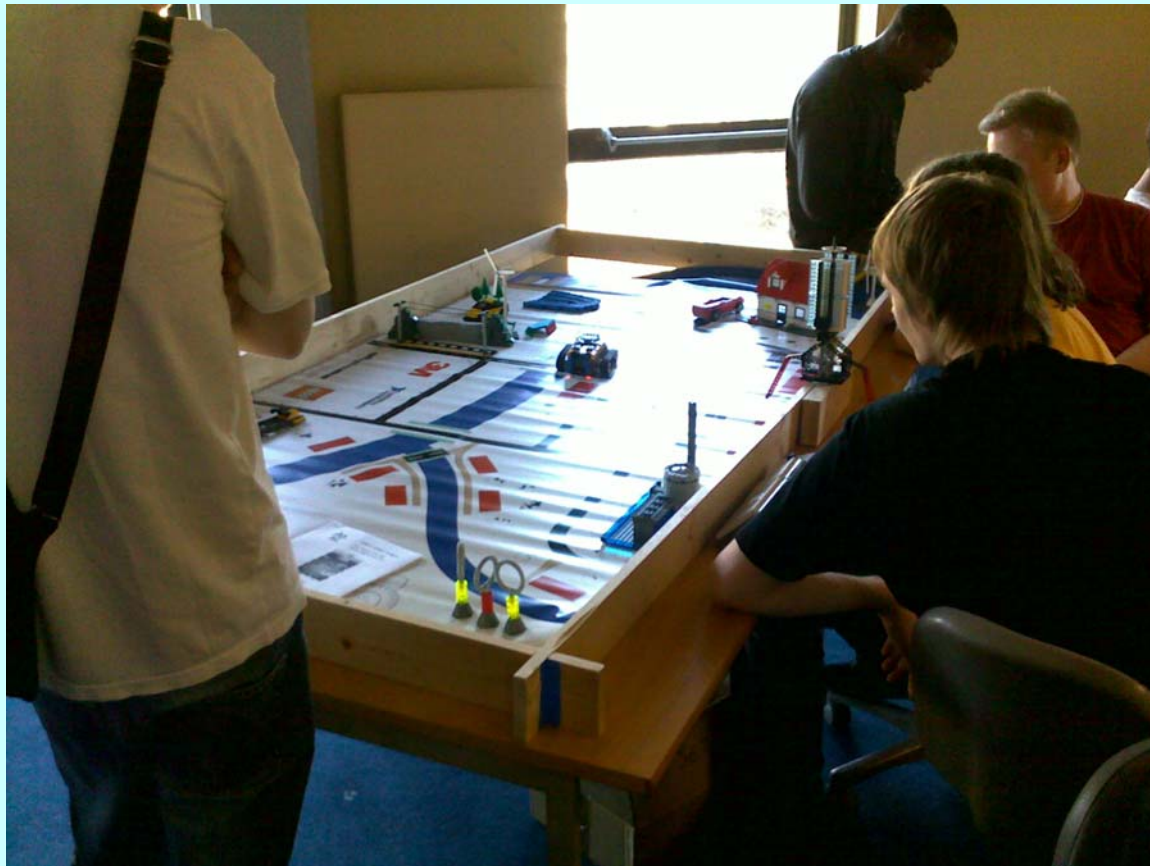
Working with Robots and Stuff

Lego Mindstorms and **occam- π** (*Transterpreter*)
– *in Freshers' Week (pre-term 1)*



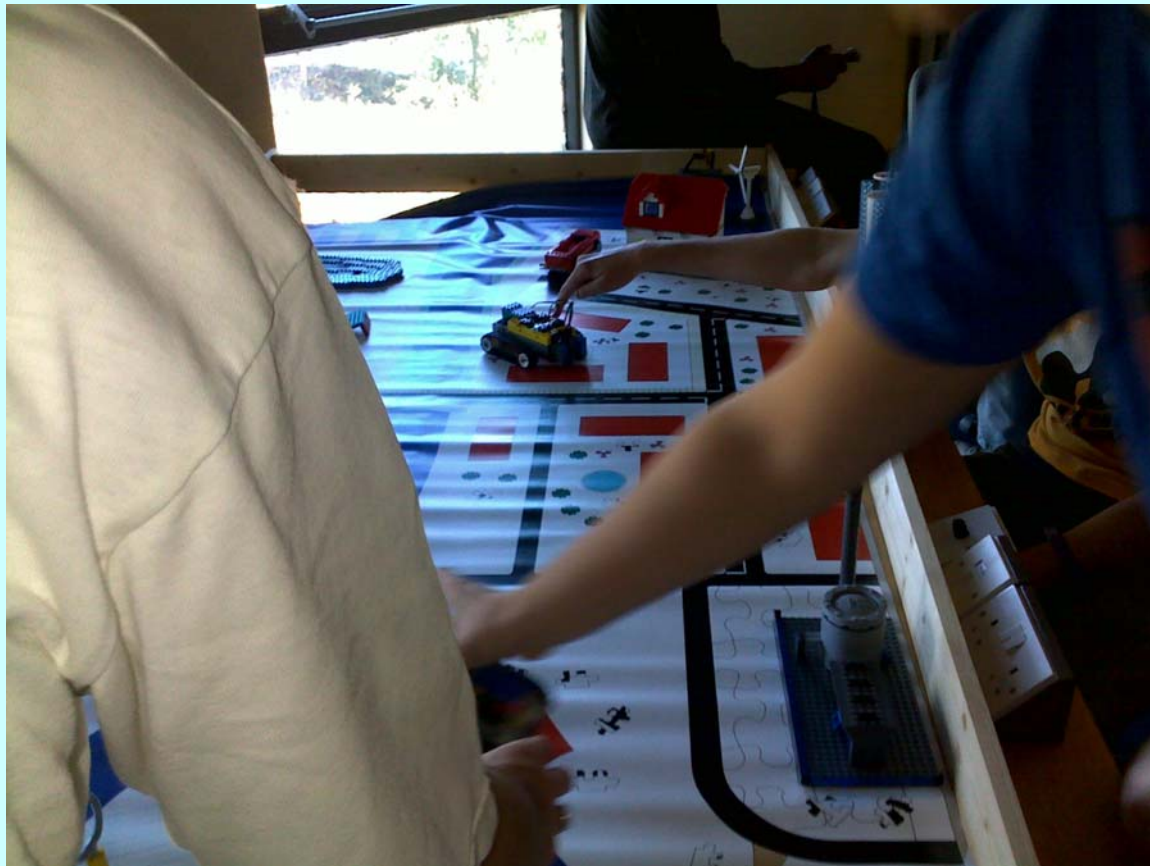
Working with Robots and Stuff

Lego Mindstorms and **occam- π** (*Transterpreter*)
– *in Freshers' Week (pre-term 1)*



Working with Robots and Stuff

Lego Mindstorms and **occam- π** (*Transterpreter*)
– *in Freshers' Week (pre-term 1)*



Working with Robots and Stuff

Lego Mindstorms and **occam- π** (*Transterpreter*)
– *in Freshers' Week (pre-term 1)*



Working with Robots and Stuff

Lego Mindstorms and **occam- π** (*Transterpreter*)
– *in Freshers' Week (pre-term 1)*

Cylons and **Life on Mars**
– *open-ended exercises (last 4 weeks of module)*

Parrot Attack and **Dining Philosophers Animation**
– *open-ended exercises (last 4 weeks of module)*

occoids and **complex system modelling / simulator**
– *final year projects*

Working with Robots and Stuff

Lego Mindstorms and **robotics (Interpreter)**
– *in Freshers' Week*

Cylons and **Life**
– *open-ended (of module)*

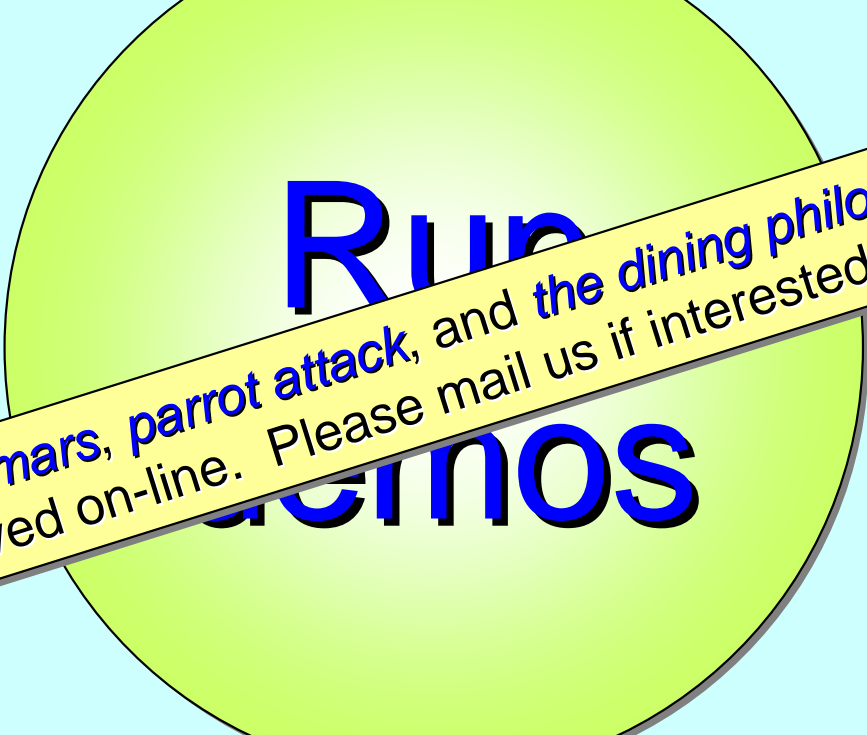
Parrot Attack and **animation**
– *open-ended exercises (of module)*

occoids and **complex system modelling / simulator**
– *final year projects*



Run
demos

Working with Robots and Stuff



Sorry: *cylons*, *life on mars*, *parrot attack*, and *the dining philosophers animations* have still to be archived on-line. Please mail us if interested!

<http://www.cosmos-research.org/demos>

(ocoids and more)

Summary (1/4)

Concurrency is fundamental across all aspects of computer science *(and life)*.

Therefore, it should be taught *at the same time as and a necessary complement to sequential* programming.

With the right concurrency model, this can be done. Going *parallel* and *looping* (for example) are equally fundamental and important structuring concepts for algorithms. *Why would we teach one and not the other? Why would we use one and not the other?!!*

The *compositional* properties of **CSP** ensures no conflict between sequential and concurrent logic. Our skills and intuition about serial programming remain valid – directly supporting our concurrency.

Summary (2/4)

At Kent, we teach using **occam- π** . Students (and staff) should not be concerned with learning another language, if the benefits are there. **occam- π** has a careful blend of the **CSP** and **π -calculus** process algebrae built in to its core. *Going parallel is as simple as looping* – both syntactically and semantically. Race hazards are not possible.

KRoC is an **occam- π** compiler and runtime for x86 architectures under Linux, OS X or *raw metal* (**RMoX**). Coming soon, *via LLVM*, this will target most things. Memory and processor overheads are ultra-low. In current practice, *hundreds of thousands* of processes can be managed on modern processors. *It eats multicore for breakfast.*

The **Transterpreter** is an **occam- π** interpreter, available stand-alone or as a plugin for the jEdit IDE. Available for Windows, it runs almost anywhere. It is easy to port to new platforms. It runs with a tiny memory footprint (e.g. for the Lego Mindstorms).

KRoC and the **Transterpreter** are (GPL) open-source downloadable.

Summary (3/4)

We also teach using **JCSP**, usually after **occam- π** . It is certainly possible and effective to teach **JCSP** first. There is an obvious advantage to those already familiar with Java. Against this are the syntactic overheads of Java (where going *parallel*, even with **JCSP**, is not quite so immediate as *looping*), the runtime overheads of the underlying threads (on which **JCSP** is built) and the *self-policing* necessary against retaining references to objects owned by other processes (which would result in race hazards). Nevertheless, we have done this in the past and may do so again. Some other universities – Surrey and Napier – are teaching now with **JCSP**.

Functional programmers may like to try **CHP** (Communicating Haskell Processes). This is implemented on top of *software transactional memory*, though the user need not know this.

There is also **CSO** (Communicating Scala Objects) from, and taught at, Oxford. This is pretty much the same as **JCSP** – the benefit being the more sympathetic syntax provided by Scala.

JCSP, **C++CSP** and **CHP** are (L-GPL) open-source downloadable.

Summary (4/4)

Our students seem to have fun. They are working with systems with *thirty* (or so) synchronising and communicating processes within the first three weeks ... and *thousands* by the end of the course. None of the exercises have a heavy maths content. Working with robots (real and virtual) and animations is very motivating. Many students put in extraordinary amounts of (their own) time ...

Feedback is, almost always, very positive. Grades are higher than the norm – but the module is elective, taken by about one-third of the cohort. We warned off those that don't want to program ... sadly, CS courses in the UK have lots of these ... it's not their fault (but that's another story not for here and now).

All our teaching materials for **occam- π** and **JCSP** are freely available (under GPL). There's quite a lot – follow the links from our position paper to this workshop. Please let us know if you use them. We will support you as best we can ... we want you to have fun as well!

And now to write the book ... 😊 😊 😊

**Any
Questions
???**

