Concurrent systems

Lecture 2: Hardware, OS and Automaton Views

Dr David J Greaves

(Thanks to Dr Robert N. M. Watson)

From last time …

- Concurrency exploits parallel and distributed computation.
- Concurrency is also a useful programming paradigm and a virtualisation means.
- Race conditions arise with imperative languages in shared memory (sadly the predominant paradigm of last 15 years).
- Concurrency bugs are hard to anticipate.

This time

- Computer architecture and O/S summary
- Hardware support for **atomicity**
- Basic Automata Theory/Jargon and interactions.
- Simple model checking
- Dining Philosophers Taster

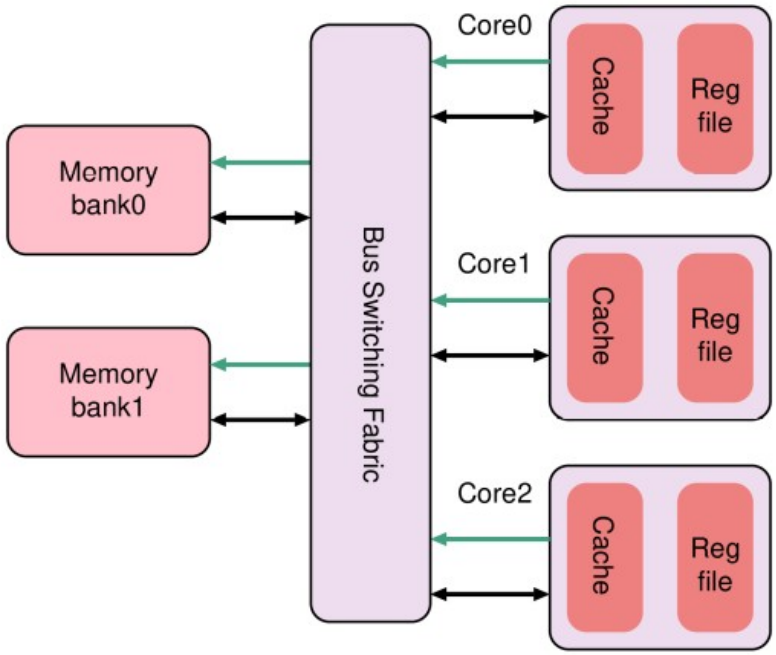
General comments

- Concurrency is essential in modern systems
  - overlapping I/O with computation
  - building distributed systems
  - But throws up a lot of challenges
- need to ensure safety, allow synchronization, and avoid issues of liveness (deadlock, livelock, …)
  - Major risk of over-engineering
- generally worth building sequential system first
  - and worth using existing libraries, tools and design patterns rather than rolling your own!

https://www.cl.cam.ac.uk/~djg11/socdam-patterns-hls-touchstones/soc-design-patterns/sp1-socparts/zhp6c8e57449.html

Computer Architecture Reference Models

Single-core, basic computer model (no hardware concurrency).

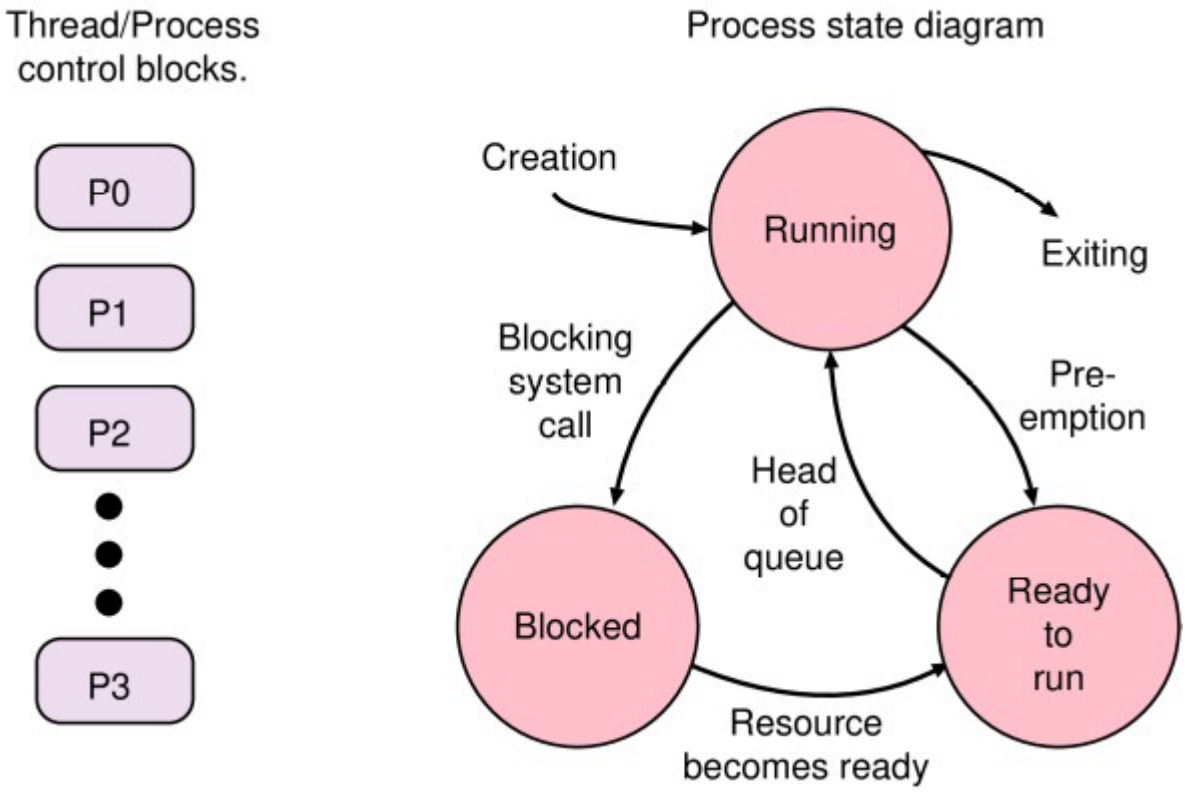


Multi-core, shared memory, flat-address space computer.

Even on a uniprocessor, interrupt routines will ‘magically’ change stored values in memory.

Stop-the-world atomic operations are undesirable on parallel hardware.

Operating System Behaviour



TCB contains saved registers for non-running tasks.
Read-to-run tasks are in a nominal queue.
Blocked TCBs point to semaphore (or similar) they are awaiting.
Most interrupt routines will invoke scheduller as they return.

Hardware foundations for atomicity 1

- On a simple uni-processor, without DMA devices, the crudest mechanism is to **disable interrupts.**
- We bracket critical section with ints_off and ints_on instructions. This guarantees no preemption.
- Can disrupt real-time response
- Not suitable when other CPUs and DMA exist
- Requires supervisor privilege.

Hardware foundations for atomicity 2

- How can we implement **atomic read-and-set**?
- Simple pair of load and store instructions fail the atomicity test (obviously divisible!)
- Need a new **ISA primitive** for protection against parallel access to memory from another CPU
- Two common flavours:
  - Atomic **Compare and Swap** (CAS)
  - **Load Linked, Store Conditional** (LL/SC)
  - (But we also find atomic increment, bitset etc..)

Atomic Compare and Swap (**CAS**)

- Instruction operands: memory address, prior + new values
  - If prior value **matches** in-memory value, **new value stored**
  - If prior value **does not match** in-memory value, **instruction fails**
  - Software checks return value, can loop on failure
- Found on CISC systems such as x86 (`cmpxchg`)?

```
   mov    %edx, 1       # New value -> register
spin:
   mov    %eax, [foo_lock]# Load prior value
   test      %eax, %eax       # If non-zero (owned),
   jnz    spin          #   loop
   lock cmpxchg [foo_lock], %edx    # If *foo_lock == %eax,
   test      %eax, %eax              # swap in value from
   jnz    spin                      # %edx; else loop
```

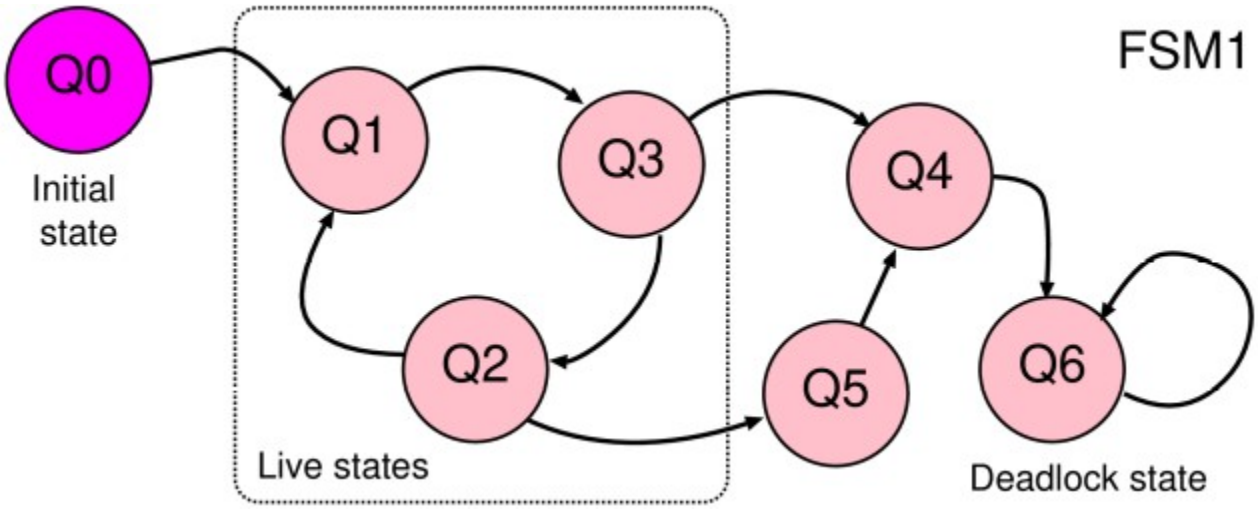
- Atomic **Test and Set** (TAS) is another variation

Load Linked-Store Conditional (**LL**/**SC**)

- Found on RISC systems (MIPS, RISC-V, ARM, …)
  - Load value from memory location with **LL**
  - Manipulate value in register (e.g., add, assign, …)
  - **SC** fails if memory neighbourhood modified (or interrupt) since **LL**
  - **SC** writes back register and indicates success (or not)
  - Software checks SC return value and typically loops on failure
  - An example of optimistic concurrency.
- Preferred since it does not lock up whole memory system while one core makes an atomic operation.

```
test_and_set_bit:    ! RISC-V code
spin:
 movli.l        @mutex, %r_tmp1    ! Load linked
 mov            %r_tmp1, %r_tmp2   ! Copy to second register
 or             %r_bitno, %r_tmp1  ! Set the desired bit
 movco.l        %r_tmp1, @mutex    ! Store-conditional
 bf             spin               ! If store failed, try again
 and            %r_bitno, %r_tmp2  ! Return old value of the bit.
 ret
```

Finite State Machine Revision and Terminology



FSM is tuple: (Q, q_0, Σ, Δ) being states, start state, input alphabet, transition function.

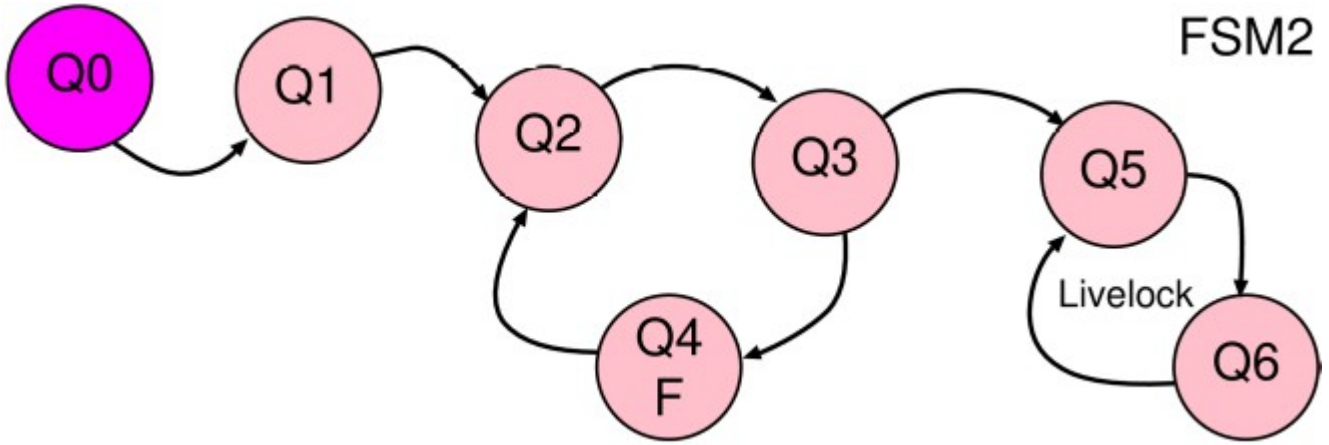
A live state is one that can be returned to infinitely often in the future.

A dead(lock) state has no successors – machine stops if we enter it.

Start-up states are those before the main live behaviour.

'Bad' states are those that lead away from the main live behaviour.

Finite State Machine: Fairness and Livelock



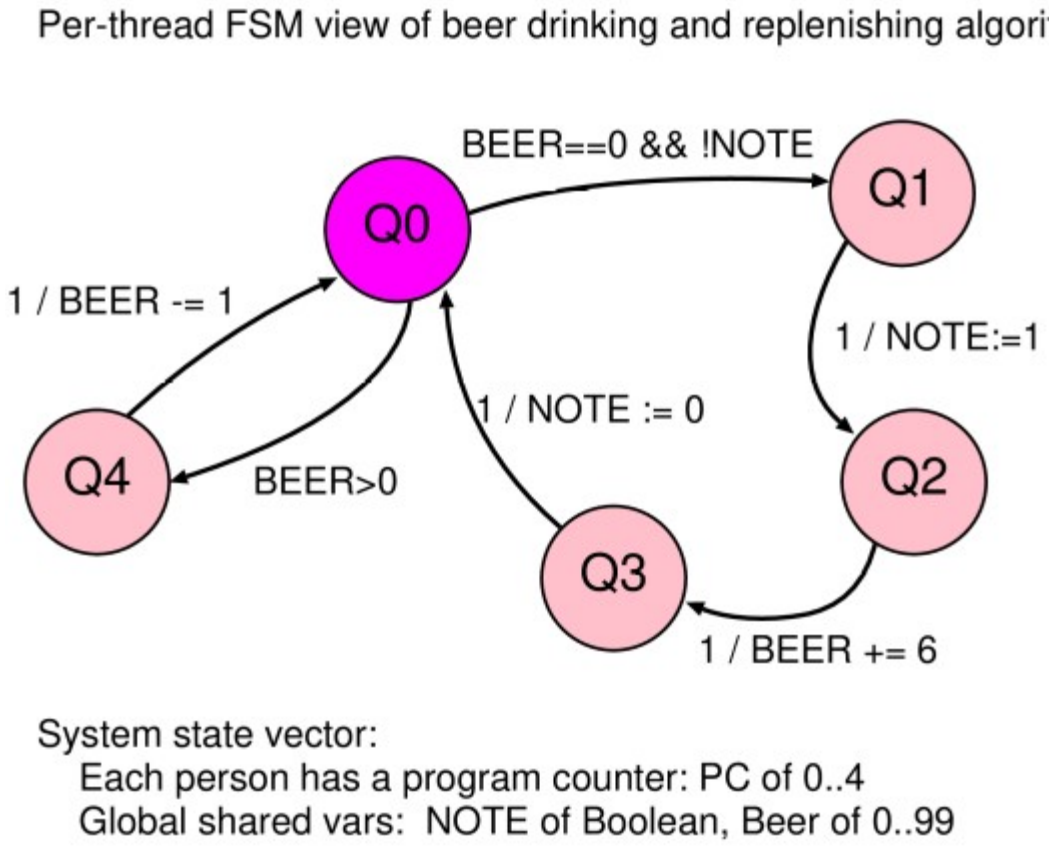
Ignoring the ‘F’, the live states of this FSM include Q5 and Q6.

F has been labelled as a ‘fair’ state. If we also discard the start-up ‘lasso stem’, its existence changes the live states to just Q2, Q3, Q4. Manual labelling defines the intended system behaviour.

Any fair state is live and states from which any fair state cannot be reached are not live. [Hence if we also labelled Q5 as F, fairness cannot be achieved.]

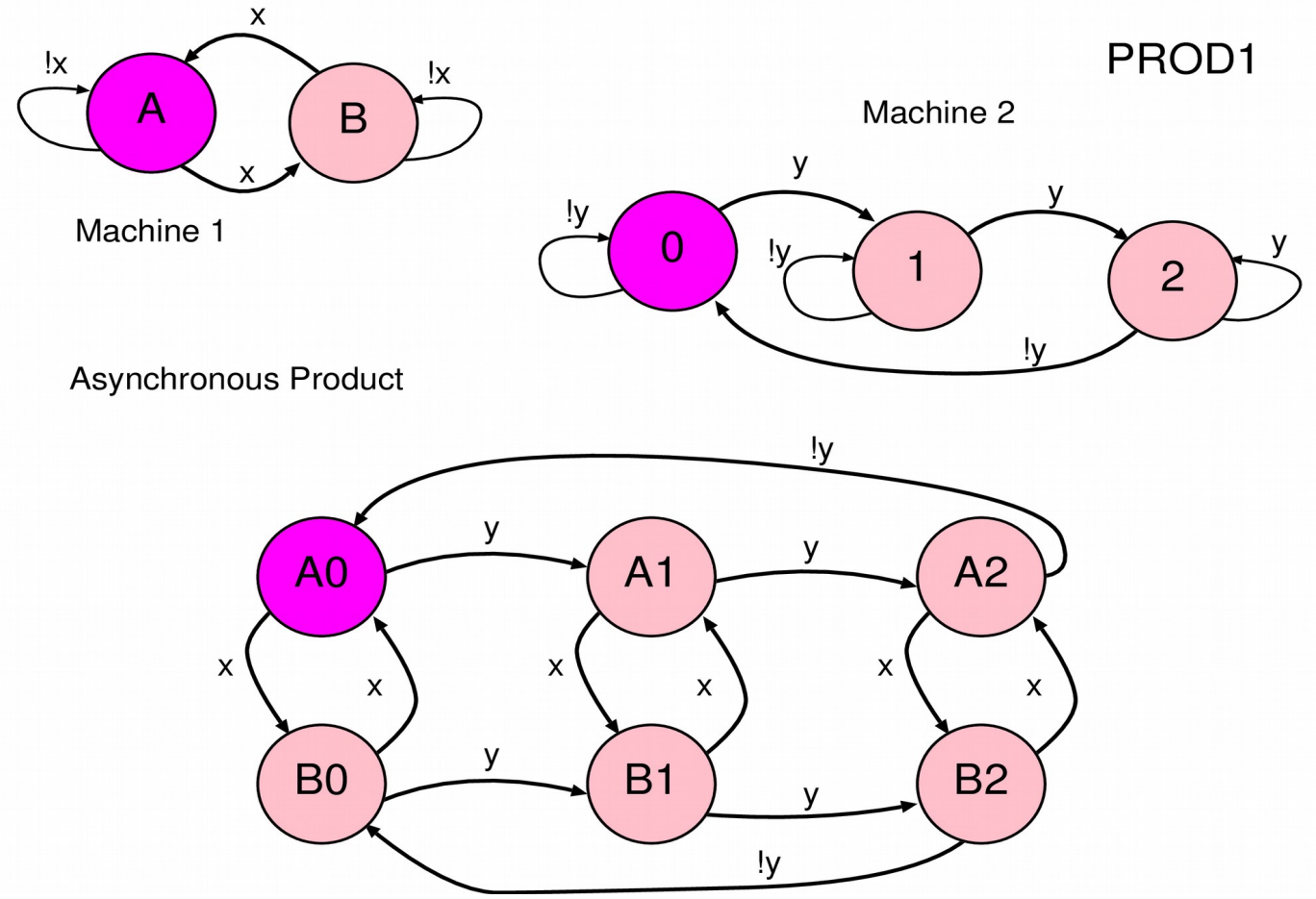
Although more rigorous definitions exist, this is sufficient terminology for us to define livelock as: we have not deadlocked but cannot make ‘useful’ progress.

Finite State Machine: FSM view of thread control flow.



FSM expresses program control flow per thread.
FSM arcs have ‘condition / action’ annotations.
Conditions and actions range over shared global state.

Finite State Machine: Product of Machines 1



Product of uncoupled machines simply multiplies state arities.
Product may be synchronous or asynchronous.
We shall not always show self arcs from now on.

Finite State Machine: Product of Machines 2

Asynchronous product: one machine steps at a time. Interleaving order is undefined (not strict alternation but so-called stuttering).

Synchronous product: all machines step at once (lock-step). We see ‘diagonal’ arcs.

Synchronous product corresponds to synchronous hardware in digital logic.

Asynchronous product is relevant for this course.

Finite State Machine: Product of Machines 3

Coupling of FSMs reduces behavio

Arc removal can lead to deadlock.

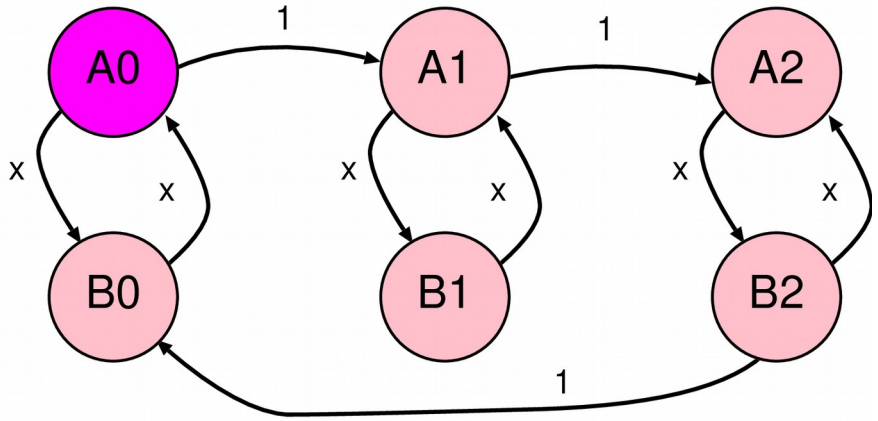
Couple FSMs by making input of or
depend on the state of the other.

Example couplings:
Half coupled:
Let y = M1 in state A.

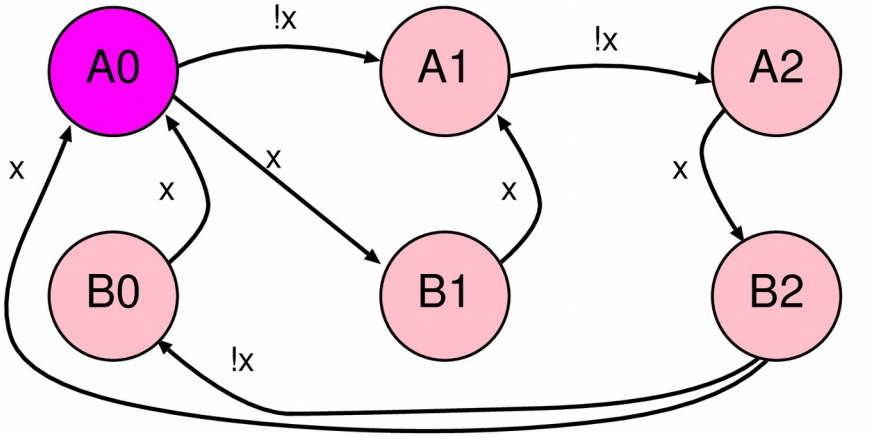
Full coupling:
Let y = M1 in state A
and x = M2 in state 0.

Another form of coupling is through
variables: those written by one FSM
appear in edge guards of another.

Half-coupled Asynchronous Product



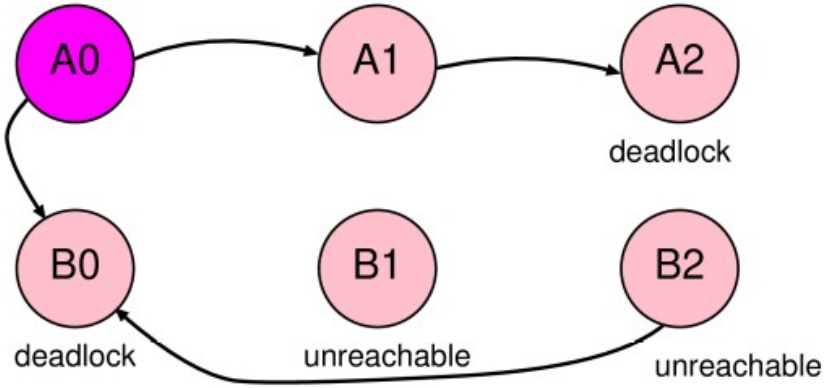
Half-coupled Synchronous Product



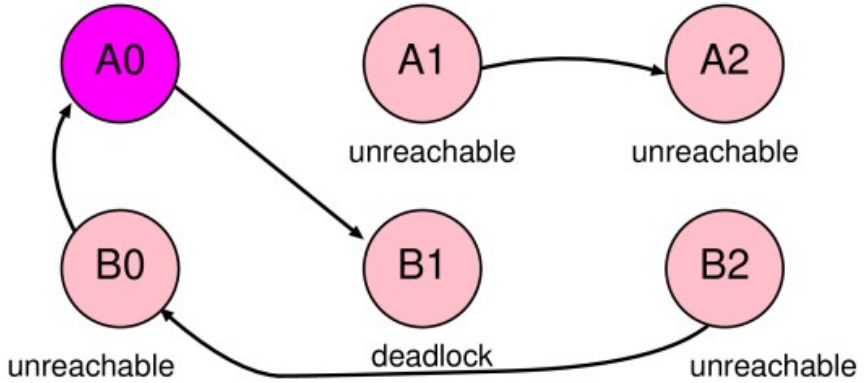
Finite State Machine: Product of Machines 4

Fully coupled:
Let y = M1 in state A
and x = M2 in state 0.

Fully-coupled Asynchronous Product



Fully-coupled Synchronous Product



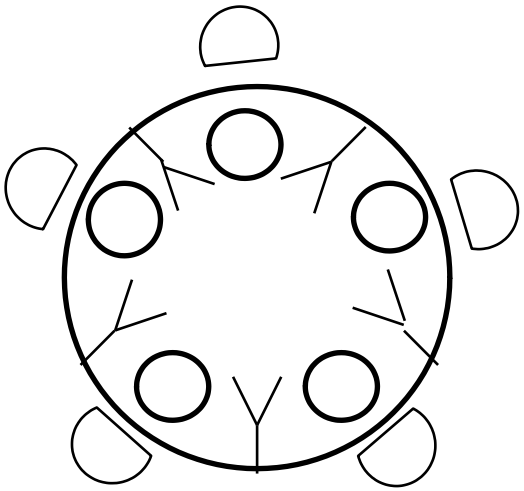
Composite machine has
no remaining external inputs.

Example: Dining Philosophers

- 5 philosophers, 5 forks, round table…

```
Semaphore forks[] = new Semaphore[5];
```

```
while(true) {          // philosopher i
   think();
   wait(fork[i]);
   wait(fork[(i+1) % 5];
   eat();
   signal(fork[i]);
   signal(fork[(i+1) % 5];
}
```



- For now, read ‘wait’ as ‘pick up’ and ‘signal’ as `put down’
- See next time for definitions.
- Exercise: Draw out FSM product for 2 or 3 philosophers.

Reachable State Space Algorithm

- 0. Input FSM = (Q, q_0, Σ, Δ)
- 1. Initialise reachable $R = \{ q_0 \}$
- 2. while(changes)

$$R = R \cup \{ q' \mid q' = \Delta(q, \sigma), q \in R, \sigma \in \Sigma \}$$

The `while(changes)' construct makes this a fixed-point iteration.

A common requirement is to check that a condition holds in all reachable states. This is called a safety property.

A model checker tool can either check the condition on each iteration for early violation detection, or else check after R is fully computed.

Live States Algorithm

- 0. Input FSM = (Q, q_0, Σ, Δ)
- 1. Initialise live set L = Q (or perhaps R)
- 2. while(changes)
  - $L = L \cap \{ q \mid q' = \Delta(q, \sigma), q' \in L, \sigma \in \Sigma \}$

Premise: A state is live or a start-up state if it has a successor that is live.

This finds the whole ‘lasso’.

To discard start-up states intersect the result with the same computation on the inverse transition function.

(This slide for interest only: not examinable.)

Model Checking Quick Demo

- If time permits, CBMC demo in lectures.
- Materials are (will be) on course site and developed a little further next time.
- Otherwise try in your own time.

Summary + next time

- We looked at underlying hardware structures (but this was for completeness rather than for examination purposes)
- We looked at finite-state models of programs and a model checker, but do note that today's tools can cope only with highly-abstracted models or small sub-systems of real-world applications.
- Next time
  - Access to hardware primitives via O/S
  - Mutual exclusion using semaphores
  - Producer/consumer and one generalisation