Concurrency

What?

- concurrency is a design principle: structuring programs to reflect potential parallelism
- sequential vs concurrent program:
- sequential: single thread of control: one instruction pointer is sufficient to manage execution
- concurrent: multiple threads of control, allowing multiple calculations to occur at the same time, and simultaneous interaction with external events
- threads/processes in a concurrent program share data or communicate with 1+ threads in that program

Why?

- natural model: e.g. user interface with keyboard + mouse + multiple windows
- necessity: e.g. autonomous robot requires multiple threads to respond appropriately
- performance: increased performance with multiple processors

What makes it hard?

- processes need to interact:
- **communication:** communication by accessing shared data/message passing
- **synchronisation:** processes need to synchronise certain events: P mustn't reach p until after Q has reached q.
- **non-determinism:** execution is non-deterministic - hence model checkers to formally establish properties

Concurrent Language Paradigms

- **shared-memory:** uses monitors
- e.g. Concurrent Pascal, Java, C#
- **message-passing:** Hoare's idea of **Communicating Sequential Processes (CSP)**
- e.g. Go, Erlang, Occam

Speed Dependence

- **speed-dependent:** when concurrent programs are dependent on relative speeds of components' execution

- fluctuations in processor/IO speed are sources of non-determinism
- **real-time systems:** when absolute speed of system matters (in embedded systems)

Arbitrary interleaving

- model of concurrent behaviour: at level of atomic events, no 2 events occur at exactly the same time
- e.g. process P performs atomic actions a, b. process Q performs x, y, z.
- 10 possible interleavings of these actions while maintaining order
- arbitrary interleaving model: these 10 sequences are the possible outcome of running P and Q concurrently

Concurrent Programming Abstraction

- concurrency is an abstraction to help reason about the dynamic behaviour of programs
- the abstraction can be related to machine language instructions, however there are no important concepts that cannot be explained at the higher level of abstraction
- **concurrent program:** finite set of sequential processes, composed of a finite number of atomic statements
- execution of a concurrent program proceeds via execution of sequence of atomic statements from the processes
- sequence formed as an **arbitrary interleaving** of atomic statements of the processes
- **computation/scenario:** possible execution sequence resulting from interleaving
- NB sequential processes implies ordering of steps is maintained
- **control pointer:** of a process indicates next statement that can be executed

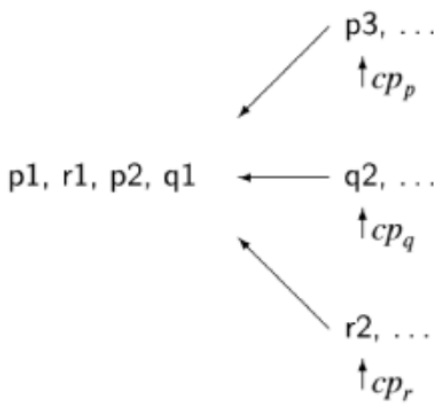

Figure 1: Arbitrary Interleaving

- arbitrary interleaving lets us ignore real-time behaviour, making programs more amenable to formal analysis
- program properties are then invariant under hardware

Atomicity

- assignments such as `n := n+1` are not atomic in most programming languages, as most compilers break them up into more basic machine code instructions: `load`, `increment`, `store`
- if 2 processes attempt to increment a shared variable simultaneously, the interleaving of these atomic instructions could be `P.load`, `Q.load`, `P.increment`, `Q.increment`, `P.store`, `Q.store`, such that the result is only `n+1`
- each process falsely assumes exclusive access to n in the *read-change-write* cycle
- **race condition/interference**
- requires **mutual exclusion**

Correctness

- for a concurrent program to be correct it must be correct for *all* possible interleavings
- correctness of non-terminating concurrent programs is defined in terms of properties: safety, liveness
- **safety properties:** property must always be true. For safety property P to hold, it must be true that in every state of every computation, P is true. “Always, a mouse cursor is displayed”
- safety properties often take form *always, something bad is not true*
- nothing bad will ever happen

- e.g. absence of **interference**
- **liveness properties:** property must eventually become true. For liveness property P to hold, it must be true that in every computation there is some state in which P is true. “If you click on a mouse button, eventually the mouse cursor will change shape”
- something good eventually happens
- e.g. absence of **deadlock**
- safety, liveness are duals of each other: the negation of a safety property is a liveness property and vice versa

Java Threads

- in Java processes are called threads

Creation

Two ways to create: - extend `java.lang.Thread`: as Java doesn’t support multiple inheritance this is not always possible - implement `Runnable` interface: recommended approach

States

Alive thread is in one of these states: - **running:** currently executing - **runnable:** not currently executing, but ready to execute - **non-runnable:** not currently executing, not ready to run - e.g. waiting on input or shared data to become unlocked

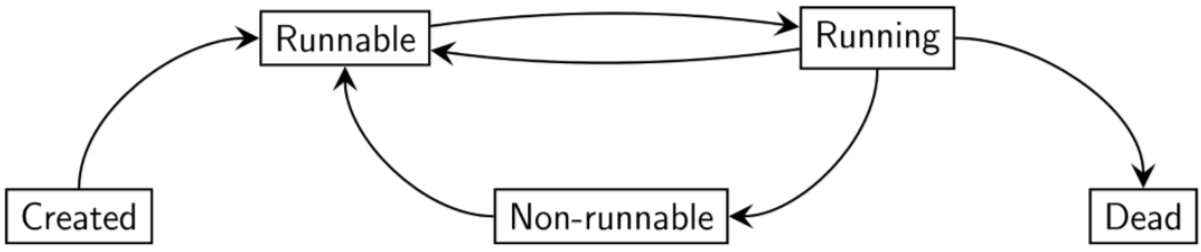

Figure 2: Java thread states

Primitives

- `start()` causes JVM to execute `run()` in a dedicated thread, concurrent with the calling code

- a thread stops executing when `run()` finishes
- `sleep(long milliseconds)` allows you to suspend thread for specified time
- `isAlive()`: indicates whether thread is running
- `yield()`: causes current thread to pause (running -> runnable)
- transition from runnable -> running is up to runtime system's scheduler
- `t.join()` suspends caller until thread `t` has completed (i.e. two threads join together)

More states

Additional states: - having called `sleep()` - having called `join()` - waiting for a lock to be released: having called `wait()`

Interruption

- Threads can be **interrupted** via `Thread.interrupt()`
- if interrupted in one of the 3 above states, the thread returns to a **runnable** state, causing `sleep(), join(), wait()` to throw an `InterruptedException`

Mutual Exclusion (Mutex)

- N processes executing infinite loops, alternating between **critical** and **non-critical** sections
- process may halt in **non-critical** section, but not in **critical** section
  - shared variables are only written to in the critical section: to avoid race condition, only one thread can be in critical section at any time

```
class P extends Thread {
  while (true) {
    non_critical_P();
    pre_protocol_P();
    critical_P();
    post_protocol_P();
  }
}

class Q extends Thread {
  while (true) {
    non_critical_Q();
    pre_protocol_Q();
    critical_Q();
    post_protocol_Q();
  }
```

```
17 }
```

Properties of mutex solution

- **mutual exclusion:** only 1 process may be active in critical section at a time
  - safety: ensure interference prevented
- **no deadlock:** if 1+ processes trying to enter their critical section, one must eventually succeed
  - liveness
- **no starvation:** if a process is trying to enter its critical section, it must eventually succeed

Also desirable: - **lack of contention:** if only one process is trying to enter critical section, it must succeed with minimal overhead - efficient

Assumptions

- no variable used in protocol is used in critical/non-critical sections and vice-versa
- load, store, test of common variables are atomic operations
- must be progress through critical sections: if a process reaches critical section, it must eventually reach the end of it
- cannot assume progress through non-critical sections: a process may terminate or enter an infinite loop

Attempt 1

- single protocol variable: token passed between processes `static int turn = 1;`
- processes wait for their turn
- properties:
  - mutex: yes. Only 1 thread can enter a critical section at a time
  - no deadlock: yes. `turn` can only have values 1 or 2, so one process can always enter
  - no starvation: no: Q can be waiting for its turn while P executes non-critical section indefinitely. Q never gets a turn - starvation.

Attempt 2

- give each thread a flag. Each thread can only modify its own flag

- a thread can only enter the critical region when the other process has lowered its flag.
- a thread raises its flag after waiting, as it is entering its critical region
- properties:
  - mutual exclusion: no. Possible for both processes to enter critical region simultaneously

Attempt 3

- as in attempt 2, give each thread a flag. Each thread can only modify its own flag
- now each process sets the flag prior to waiting
- properties:
  - mutual exclusion: yes
  - no deadlock: no. Both processes set flag prior to entering critical region. Neither can proceed
  - no starvation: no, as there can be deadlock, both processes will starve
  - lack of contention: yes. if P is in non-critical section Q can enter its critical section

Attempt 4

- as in attempt 3, give each thread a flag. Each thread can only modify its own flag
- each process sets the flag prior to waiting
- if both processes have the flag raised, momentarily lower then re-raise the flag
- properties:
  - mutual exclusion: yes
  - no deadlock: yes. Lowering of flags removes deadlock
  - no starvation: no. Can get livelock, with infinite sequence of both processes lowering/raising flags without either entering critical region
  - lack of contention: yes, per attempt 3

Livelock: processes are still moving, but critical section is unable to be completed

Attempt 5: Dekker's Algorithm

- use flags + turn token

```
1 static int turn =1;
2 static int p = 0;
3 static int q = 0;
```

- whoever previously entered critical section has lower priority to enter the critical section

```
while (true) {
  non_critical_P();
  p = 1;
  // repeat while Q has flag raised
  while (q != 0) {
    // if it is Q's turn
    if (turn == 2) {
        // lower flag
        p = 0;
        // wait until its P's turn
        while (turn == 2);
        // raise p's flag
        p = 1;
    }
  }
  critical_P();
  turn = 2;
  p = 0;
}

while (true) {
  non_critical_Q();
  q = 1;
  // repeat while P has flag raised
  while (p != 0) {
    // if it is Q's turn
    if (turn == 1) {
        // lower flag
        q = 0;
        // wait until its Q's turn
        while (turn == 1);
        // raise Q's flag
        q = 1;
    }
  }
  critical_Q();
  turn = 1;
  q = 0;
}
```

- properties:
  - mutex: yes. P only enters critical section if `q != 0`
  - no deadlock: yes, thanks to flag lowering
  - no starvation: yes, no livelock as in attempt 4 due to turn priority
  - lack of contention: yes. If P is in non-critical section Q can enter critical section
- hard to generalise to programs with > 2 processes

Peterson's Mutex Algorithm

- 1981 solution, scales more readily than Dekker's algorithm
- also uses flags and turn token

```
static int turn = 1;
static int p = 0;
static int q = 0;

while (true) {
  non_critical_P();
  p = 1;
  turn = 2;
  // give Q a turn.  wait till it is complete
  while (q && turn == 2);
  critical_p();
  p = 0;
}
```

Java: Monitors and synchronisation

- correct algorithms for mutex are tedious and complex to implement
- concurrent programming languages offer higher-level synchronisation primitives
- Java offers
  - **synchronised methods/objects:** a method/object can be declared `synchronized` - only 1 process can execute/modify it at a time
  - **monitors:** set of synchronized methods/data that queue processes trying to access the data

Synchronised methods

- `synchronized` keyword declares method/object as being executable/modifiable by only 1 process at a time
  - marks method as **critical section**

```
synchronized void increment() { ... }
```

Synchronized object

- can declare an object as synchronised, making entire object mutually exclusive:

- disadvantage: requires user of shared object to lock the object, rather than placing this inside shared object and encapsulating
- if user fails to lock object correctly, race conditions can occur

```
class SynchedObject extends Thread {
  Counter c;

  public SynchedObject(Counter c) { this.c = c; }

  public void run() {
    for (int i = 0; i < 5; i++) {
      synchronized(c) {
        c.increment();
      }
    }
  }
}
```

Monitors

- language feature that provides mutual exclusion to shared data
- in Java, a monitor is an object that encapsulates some private data, with access via synchronized methods
  - manages blocking/unblocking of processes seeking access
- e.g. bank account shared between parent and child
- leaving responsibility of wait to client of shared object is bad because
  - user has to continually poll: wasteful
  - code needs to be replicated for multiple clients
  - an incorrect implementation of any client means interference can occur
- monitors alleviate these issues by making the encapsulating class do the work
- **monitor:** encapsulated data + operations/methods
  - maintains queue of processing wanting access
- all objects in Java have monitors, having a lock that allows holding thread to access synchronized methods of the object
- `Object` contains 3 relevant methods:
  - **`void`** `wait()`: causes current thread to wait until another thread invokes `notify()` or `notifyAll()` for this object. (i.e. `wait()` causes the thread to block, and relinquishes the lock the thread holds to other waiting threads)

  - `void notify()`: wakes up a *single* thread waiting on this object's lock
    - choice of thread is arbitrary (up to JVM)
    - not needed for this course
  - `void notifyAll()`: wakes up all threads waiting on object's lock

```
1  class MonitorAccount extends Account {
2      public synchronized void withdraw(int amount) {
3          while (balance < amount) {
4              // withdrawal cannot proceed.  get thread to wait until
                   balance updates
5              try {
6                  wait();
7              } catch (InterruptedException e) {}
8          }
9          super.withdraw(amount);
10     }
11
12     public synchronized void deposit(int amount) {
13         super.deposit(amount);
14         // after deposit, notify all threads waiting for updated
               balance
15         notifyAll();
16     }
17 }
```

Lightweight monitors

- every object has a lock: to execute a `synchronized` method, a process first needs to acquire the object's lock
- the lock is released upon return
- a process P, holding the lock on object o, can relinquish the lock by invoking `wait()`
  - P is then **waiting** on o
- a process Q may execute `o.notify()`, changing some waiting process' state to **locking**
  - Q must have o's lock, and be running, for this to occur
  - the notifier holds the lock until the end of the synchronised method/code block
- Java monitors are lightweight, and don't guarantee fairness
  - some programming languages include priority: Java does not.
  - it is essentially random which thread will be chosen
  - typical pattern: `while (){ wait()}`
    - common mistake: using `if` where `while` should be used

  - original monitor concepts allowed for different wait sets, each waiting for a specific condition to hold. Java does not
    * instead Java uses `notifyAll()` to release all processes waiting on the object

Implementation

- for a class to meet the requirements of a monitor:
  - all attributes **`private`**
  - all methods **`synchronized`**
- if a class satisfies these requirements, all methods are treated as atomic events

Volatile variables

- declaring variable **`volatile`** directs JVM to reload its value every time it needs to refer to it
  - otherwise compiler may optimise code to load value once only
- compilers/VMs load the value of a variable into a cache for efficiency
- if value is modified by one thread, other threads relying on cached values may not detect this, and use the stale cached value
- updates to variable values may be made initially in the cache, and not immediately be written back to memory
- declaration of variables as **`volatile`** directs VM to read/write that variable directly to/from memory

Process states

- synchronisation constructs (e.g. monitors) can produce a different non-runnable state in which the process is blocked
- **blocked** process relies on other processes to **unblock** it, after which it is again **runnable**

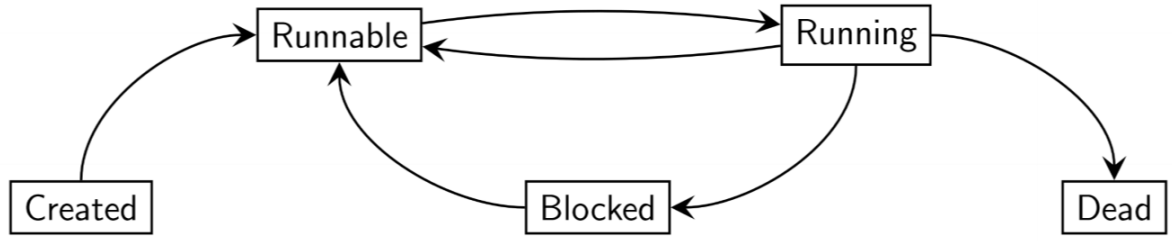

Figure 3: Blocked Java States

Synchronisation constructs

Level of abstraction	Construct
High	Monitor
	Semaphore
Low	Protocol variables

Java: Semaphores

- **semaphore:** (v, W), simple, versatile concurrent device for managing access to a shared resource
  - **value** $v \in \mathbb{N}$**:** number of currently available access permits
  - **wait set** W**:** processes currently waiting for access
- must be initialised $S := (k, \{\})$
  - k: maximum number of threads simultaneously accessing the resource
- atomic operations: `wait`, `signal`

Analogy: Hotel with k rooms

- 1 guest per room
- at the door is a receptionist
- outside are people wanting a room

- receptionist gives out the 10 keys they have. Number of keys decreases as each key is handed out
- once all keys have been handed out, others must wait outside until a key is returned

Operations

- `S.wait()`: receive permit if available, otherwise get added to the wait set
- `S.signal()`: return a permit, unblock an arbitrary process

```
S.wait():

if S.v > 0
  # provide permit
  S.v--
else
  # add process p to wait set
  S.w = union(S.W, p)
  p.state = blocked
```

```
S.signal():

if S.W == {}
  # empty wait set, so keep the permit
  S.v++
else
  # hand out permit to someone in the wait set
  choose q from S.W
  # remove q from wait set
  S.W = S.W \ {q}
  q.state = runnable
```

Binary Semaphore: Mutex

- if $S.v \in \{0, 1\}$, S is called **binary/mutex** as it ensures mutual exclusion
- semaphores are implemented in many programming languages, as well as at the hardware level

Solution of Mutex Problem

- using a binary semaphore:

```
binary semaphore S = (1, {});

Process P loop:
```

```
4  p1:  non_critical_p();
5  p2:  S.wait();
6  p3:  critical_p();
7  p4:  S.signal();
8
9  Process Q loop:
10 q1: non_critical_q();
11 q2: S.wait();
12 q3: critical_q();
13 q4: S.signal();
```

State Diagrams

- digraph: nodes - states, edges - transitions
- state gives info about a process at a point in time: values of instruction pointer + local variables
- below shows semaphore solution - only shows `signal`, `wait` operations:

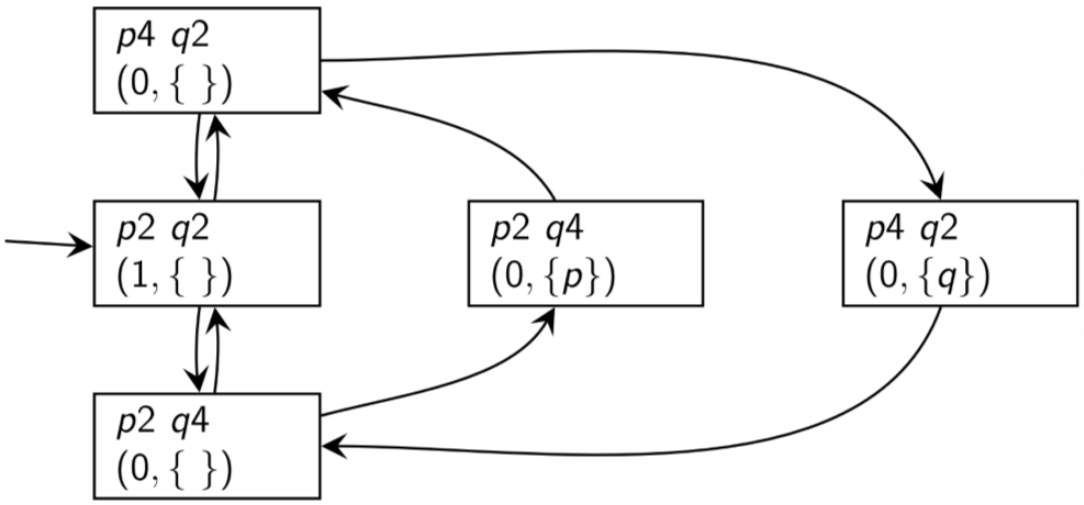

Figure 4: Semaphore mutual exclusion

- can show
  - correctness of solution (i.e. mutual exclusion): there is no state `p4`, `q4`
  - absence of deadlock: deadlock would be a node with no outgoing edges
  - absence of starvation

Controlling execution order

- use of 2 semaphores to control sorting

```
integer array A
binary semaphore S1 = (0, {})
binary semaphore S2 = (0, {})

p1: sort low half
p2: S1.signal()
p3:

q1: sort high half
q2: S2.signal()
q3:

# wait for both semaphores to become available
m1: S1.wait()
m2: S2.wait()
m3: merge halves
```

Strong semaphores

- binary semaphore solution to mutex problem generalises to N processes
- when $N > 2$: no longer guarantee of freedom from **starvation**
  - blocked processes are taken arbitrarily from a set
  - fair implementation: processes wait in a **queue**
  - removes starvation, and then we have **strong semaphore**

Bounded Buffer Problem

- e.g. streaming video via YouTube
  - Producer: server
  - Consumer: browser
  - want regular, consistent speed of video even with noisy data transfer
  - use a buffer to smooth this out, using a queue of video frames
  - need a way to add to/remove from queue
  - buffer has finite size:
    - cannot add data to a full buffer
    - cannot remove data from an empty buffer

- common pattern in concurrent/async systems
- Producer process `p`
- Consumer process `q`
- `p` generates items for `q` to process
- if they have similar average but varying speed, a **buffer** can smooth overall processing and speed it up, permitting asynchronous communication between `p` and `q`
- general semaphores can be used for this:
  - two semaphores $S1, S2$ maintain a loop invariant $S1.v + S2.v = n$
  - n: buffer size
- let's call the semaphores `notEmpty`, `notFull`

```
1  buffer = empty queue;
2  // no permits available for removal from queue
3  semaphore notEmpty = (0, {});
4  // n permits available for adding to queue
5  semaphore notFull = (n, {});
6
7  Producer
8  idem d
9  loop
10   # produce items
11   p1: d = produce();
12   # wait for buffer to have space
13   p2: notFull.wait();
14   p3: buffer.put(d);
15   # indicate data has been put onto buffer
16   p4: notEmpty.signal();
17
18 Consumer
19 item d
20 loop
21   # wait until the buffer has items to consume
22   q1: notEmpty.wait();
23   q2: d = buffer.take();
24   # indicate item taken from buffer
25   q3: notFull.signal();
26   q4: consume(d);
```

Java semaphores

- `java.util.concurrent` has `Semaphore` class
  - `acquire()` = wait
  - `release()` = signal

- has optional argument to make it a strong semaphore, by default they are weak

Java Thread states in detail

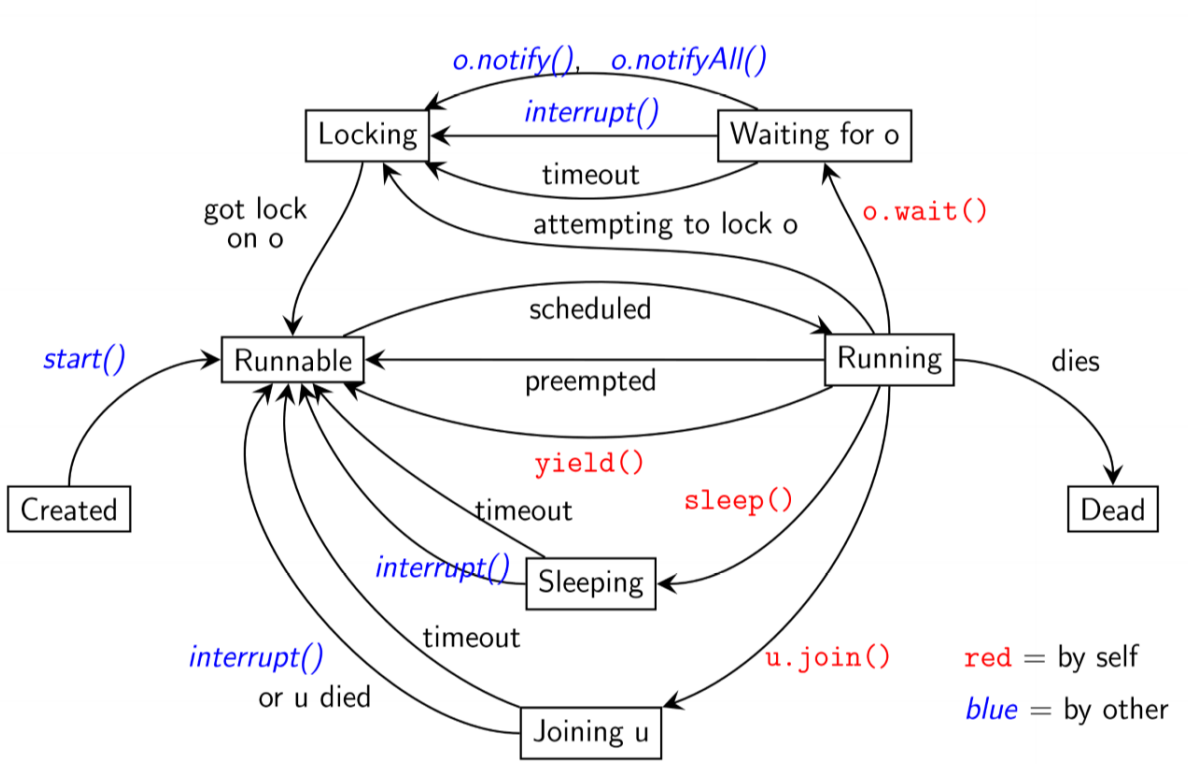

Figure 5: Java thread states

Peterson's mutex algorithm

```
1  static int turn = 1;
2  static int p = 0;
3  static int q = 0;
4
5  while (true) {
6  p1:  non_critical_P();
7  p2:  p = 1;
8  p3:  turn = 2;
9       // give Q a turn.  wait till it is complete
10 p4:  while (q && turn == 2);
11 p5:  critical_p();
12 p6:  p = 0;
13 }
14
```

```
15 while (true) {
16 q1:  non_critical_q();
17 q2:  q = 1;
18 q3:  turn = 1;
19      // give P a turn.  wait till it is comqlete
20 q4:  while (p && turn == 2);
21 q5:  critical_q();
22 q6:  q = 0;
23 }
```

- finite number of states:
  - possibly as many as 288 states, but most are unreachable
- each state is a tuple $(p_i, q_i, p, qturn)$
- can exclude statements that aren't part of the protocol p_1, p_5, q_1, q_5
- 14 states of interest
- can see from state diagram
  - mutex achieved: states (p6, q6, …) are unreachable
  - no deadlock: no state of form (p4, q4, …) is stuck (i.e. can always progress when both are waiting to enter critical region)
  - no starvation: from each state (p4,…) a state (p6,…) can be reached. The same holds for q4.

Formal modelling with FSP

- Finite State Processes (FSP): language based on Communicating Sequential Processes (CSP) and Calculus of Communicating Systems (CCS)
- rules for manipulating/reasoning about expressions in these languages: process algebra
- use of FSP vs CSP/CCS
  - machine readable syntax
  - models are finite. The others can have infinite system states, making them much more difficult to reason about
  - allows us to execute them and exhaustively prove properties about them

Advantages of formal modelling

- forces preciseness in thinking
- provides rigour needed to analyse models, compare with physical circumstances and make trade-offs

LTS

- **Labelled transition system (LTS): finite state machine** used as a model of our programs
- doesn't specify timing, only considers sequence
  - alternative formalisms do consider timing

FSP

- graphical representation works for small systems but quickly becomes unmanageable/unreadable for real problems
  - huge number of states/transitions
- hence we use an algebraic language, **finite state processes**, to describe process models
- each FSP model has a corresponding LTS model

Concepts

- process model consists of
  - alphabet: atomic actions that can occur in a process
  - definition of legal sequences of atomic actions
- processes, and synchronisation of concurrent processes is described using algebraic operators

Action prefix operator `->`

If x is an action and P a process, then x -> P describes a process that first engages in action x and then behaves as described by P

- always has
  - atomic action as left operand
  - process as right operand
- repetitive behaviour: use recursion
- atomic actions: lower case
- process names: upper case

Subprocesses `,`

- subprocesses can be defined local to the definition of a process using `,`

```
PROCESS = SUBPROCESS,
SUBPROCESS = (action1 -> SUBPROCESS2),
SUBPROCESS 2 = (action2 -> SUBPROCESS).
```

Choice `|`

- choice operation | describes a process that can execute more than one possible sequence of actions
- `(x -> P | y -> Q)` describes a process which initially engages in either x or y, followed by process P or Q respectively
- FSP does not distinguish input/output actions
  - actions that form part of a choice are usually considered inputs
  - actions that offer no choice are usually considered outputs

Non-deterministic choice

- `(x -> P | x -> Q)`: describes process that engages in x then behaves as P or Q
- x is prefix in both options
- choice is made by process, not environment: x could be an input from the environment, but the choice of P/Q is not controlled by it

Indexed Processes

- can use an index to model a process that can take multiple values
- increases expressiveness of FSP
- e.g. buffer that can contain a single input value, ranging from 0-3, and then outputs the value

```
BUFFER = (in[i:0..3] -> out[i] -> BUFFER).
```

Constants and Ranges

- constants can only take integer values
- ranges are finite ranges of integers

```
const N = 3
range T = 0..N

BUFF = (in[i:T] -> STORE[i]),
STORE[i:T] = (out[i] -> BUFF).
```

Guarded actions

- guarded action allows a context condition to be added to options in a choice
- `(when B x -> P | y -> Q)`:
  - when guard B is true, actions x and y are both eligible to be chosen
  - when guard B is false, action x cannot be chosen
- counter

```
COUNT(N=3) = COUNT[0],
COUNT[i:0..N] = ( when (i<N) inc -> COUNT[i+1]
                | when (i>0) dec -> COUNT[i-1]
                ).
```

STOP process

- `STOP` is a special, predefined process that engages in no further actions
- used for defining processes that terminate

Concurrency in FSP

Parallel composition `||`

- if P and Q are processes, `(P || Q)` represents concurrent execution of P and Q
- semantics specify 2 processes will interleave: only a single atomic action from either will execute at one time
- when a process P is defined by parallel composition, its name must be prefixed `||P`

Parallel composition rules

- algebraic laws - for all P, Q, R:
  - commutativity: `(P || Q)== (Q || P)`

  - associativity: `((P || Q)|| R)== (P || (Q || R))`

- composite processes are 1st class citizens and can be interleaved with other processes
- i.e. we can build up large, complicated systems from simpler systems

Shared Actions

- if processes in composition have actions in common, these actions are **shared**
  - this models process interaction
- unshared actions may be arbitrarily interleaved
- shared actions must be executed simultaneously by all processes that participate in that shared action
  - i.e. other processes will be blocked until able to take that action

Relabelling actions

- sometimes convenient to make actions relevant to the local process, and rename them so that they are shared in a composite process
- `P/{new1/old1, ..., newN/oldN}` is the same as P but with action `old1` renamed to `new1` etc.

Process Labelling

- to distinguish between different instances of the same process, we can prepend each action label of an instance with a distinct instance name
- `a:P` prefixes each action label in P with a
- you can also use an array of prefixes: `||N_CLIENTS(N=3)= (c[i:1..N]:CLIENT).`
- equivalently: `||N_CLIENTS(M=3)= (forall[i:1..M] c[i].CLIENT).`
- to ensure composite process of the server with clients then shares actions, you will need to prepend the prefixes for all action labels and add transitions
- `{a1,..,ax}::P` replaces every action label `n` in P's alphabet with the labels `a1.n, ..., ax.n`. Every transition `n->X` in P is replaced with transitions `({a1.n,..,ax.n}->X)`
  - `{a1,..,ax}`: shorthand for set of transitions `(a1 -> X), ..., (ax -> X)`

Client-Server example

- N clients and one server:

```
CLIENT = (call -> wait -> continue -> CLIENT).
SERVER = (request -> service -> reply -> CLIENT).

||N_CLIENT_SERVER(N=2) =
    (  forall[i:1..N] (c[i]:CLIENT)
    || {c[i..N]}::(SERVER/{call/request, wait/reply})
    ).
```

Variable hiding

- you can hide variables to reduce complexity
- `P\{a1,...,aN}` is the same as P with actions `a1, ..., aN` removed, making them silent.

- silent actions are name `tau` and are never shared
- alternatively you can list variables that are not to be hidden: `P@{a1,...,aN}` is the same as P with all action names other than `a1,...,aN` removed

FSP Synchronisation

- we can use LTSA to check for problems such as deadlock, interference automatically
- **deadlock:** process is blocked waiting for a condition that will never become true
- **livelock:** busy wait deadlock; process is spinning while waiting for a condition that will never become true
  - can happen if concurrent processes are mutually waiting for each other

Coffman Conditions

4 necessary and sufficient conditions. All must occur for deadlock to happen

1. **serially reusable resources:** processes must **share** some reusable resources between themselves under **mutual exclusion**
2. **incremental acquisition:** processes **hold on** to allocated resources **while waiting** for other resources
3. **no preemption:** once a process has acquired a resource, it can only release it **voluntarily**, i.e. it cannot be preempted/forced to release it

4. **wait-for cycle:** a **cycle** exists in which each process holds a resource which its **successor** is waiting for

- e.g. serially reusable resource: 2 people at dinner order steak, with only 1 steak knife at the table
  - to eat steak, the steak knife is required
  - need to wait for the knife to be available to proceed
- e.g. incremental acquisition: once you have the knife, wait until you also acquire a fork
- any deadlock in concurrent systems can be broken down to 4 Coffman conditions
- Corollary: to **remove** deadlock, break any of the Coffman conditions

Monitors: FSP vs Java

- FSP monitors map well to Java monitors: `when cond act -> NEW_STATE` becomes

```java
public synchronized void act() throws InterruptedException {
    while (!cond) wait();
    // modify monitor data
    notifyAll();
}
```

Bounded buffers using monitors

- buffer with finite size, into which items are inserted by a producer, and removed by a consumer in FIFO manner
- due to finite size, items can only be inserted if buffer is not full, otherwise the producer is blocked.

- Items can only be removed if it is not empty, otherwise the consumer is blocked

```
// bounded buffer using monitor

// buffer size
const N = 4
range U = 0..N

BUFFER = BUFF[0],
BUFF[i:U] = ( when (i < N) put -> BUFF[i+1]
                | when (i > 0) get -> BUFF[i-1]
                ).

PRODUCER = (put -> PRODUCER).
CONSUMER = (get -> CONSUMER).
```

```
14
15 ||BOUNDED_BUFFER = (PRODUCER || CONSUMER || BUFFER).
```

- note FSP implementation is much simpler than Java implementation: FSP is concise and expressive

Bounded buffers using semaphores

Checking Safety in FSP

Counter

```
1  const N = 4
2  range T = 0..N
3
4  VAR = VAR[0],
5  // variable can be read/written to
6  VAR[u:T] = (read[u] -> VAR[u] | write[v:T] -> VAR[v]).
7
8  CTR = ( read[x:T] ->
9          ( when (x<N) increment -> write[x+1] -> CTR
10         | when (x==N) end -> END
11         )
12       )+{read[T], write[T]}.
13
14 // create a shared counter
15 ||SHARED_COUNTER = ({a,b}:CTR || {a,b}::VAR).
```

Alphabet Extensions

- **alphabet:** set of action a process engages in
- in above e.g. CTR process has alphabet `{read[0], ..., read[4], write[1], ..., write[4]}`
- `write[0]` is not part of the alphabet, as CTR never performs this action
- when CTR is composed with VAR, this means `write[0]` can be executed at any time
- extend the alphabet of CTR to prevent this problem

Checking for interference

- find a trace such that both processes write the same value

```
INTERFERENCE = (a.write[v:T] -> b.write[v] -> ERROR).

||SHARED_COUNTER = ({a,b}:CTR || {a,b}::VAR || INTERFERENCE).
```

ERROR is a predefined process signalling an error in the model, causing deadlock.

- now safety check shows deadlock, produced by both processes writing the same value to the variable
- e.g. both processes write value 1: interference

```
a.read.0
a.increment
b.read.0
a.write.1
b.increment
b.write.1
```

Mutual Exclusion

- create a LOCK process to allow synchronisation between counters

```
LOCK = (acquire -> release -> LOCK).
```

- modify CTR so it has to acquire a lock on VAR before executing the critical section, and release afterwards

```
CTR = ( acquire -> read[x:T] ->
        ( when (x<N) increment -> write[x+1] -> release -> CTR
        | when (x==N) release -> END
        )
      )+{read[T], write[T]}.


||LOCKED_SHAREDCOUNTER = ({a,b}:CTR || {a,b}::(LOCK||VAR)).
```

- if we add in the `INTERFERENCE` process, we see it fails to find a trace for interference, but a deadlock is found (because INTERFERENCE expects a to write before b)
- instead of making INTERFERENCE more complicated, we should instead use **properties**
  - the approach used above specifies negative behaviours that can occur: sometimes more powerful to use the inverse

Safety and Liveness Properties

- methodology:
  - describe concurrent processes using FSP
  - describe property of model, i.e. something true for every possible trace/execution of that model
- categories of properties of interest for concurrent systems:
  - **safety:** nothing bad happens during execution. E.g. deadlock
    - sequential system safety property: satisfies some assertion each time a given program point is reached
    - concurrent system: e.g. absence of deadlock/interference
  - **liveness:** something good eventually happens. e.g. all processes trying to access a critical section eventually get access
    - sequential system: system terminates
    - concurrent system: as non-terminating, relates to resource access

Error States

- `ERROR`: pre-defined process signalling termination in an error state, i.e. a state we don't want to move into
- labelled -1
- no outgoing transitions
- can be used to indicate erroneous behaviour: explicitly identify erroneous action

Safety Properties

- better to consider **desired system behaviour** rather than enumerating all possible undesirable behaviours
- i.e. specify desirable properties and check the model maintains them: **safety properties**
- specified with `property` keyword
- LTSA compiler adds outgoing action to error state for all actions in process alphabet that aren't outgoing actions
  - LTS is then complete: all actions can occur from all states, invalid actions leading to the error state
  - safety properties must be deterministic processes: no non-deterministic choice

- NB safety properties don't affect normal behaviour of original process because all combinations of actions are allowed: all previous transitions remain, and all shared actions can be synchronised
  - if behaviour violating safety property occurs, the result in the composite process is the error state

```
ACTUATOR = (command -> ACT),
ACT = (respond -> ACTUATOR | command -> ACTUATOR).

property SAFE_ACTUATOR = (command -> respond -> SAFE_ACTUATOR).

||CHECK_ACTUATOR = (ACTUATOR || SAFE_ACTUATOR).
```

Safety property: interference

- returning to counter e.g.
- when a value `v` is written, the next value written is `v+1`

```
property NO_INTERFERENCE = ({a,b}.write[v:T] -> (when (v<N) {a,b}.write
    [v+1] -> NO_INTERFERENCE)).

||SHARED_COUNTER = ({a,b}:CTR || {a,b}::VAR || NO_INTERFERENCE).
```

- `{a,b}.write`: either `a` or `b` can engage in `write`
- the property therefore doesn't care who writes the value, as long as the next value is one higher
- the guard `(v<N)` prevents `N+1` being written
- safety property processes must be composed with the other processes

Without lock

```
const N = 4
range T = 0..N

VAR = VAR[0],
VAR[u:T] = (read[u] -> VAR[u] | write[v:T] -> VAR[v]).

CTR = (read[x:T] -> ( when (x<N) increment -> write[x+1] -> CTR
                    | when (x == N) end -> END
                ))+{read[T], write[T]}.

property NO_INTERFERENCE = ({a,b}.write[v:T] -> (when (v<N) {a,b}.write
    [v+1] -> NO_INTERFERENCE)).

||SHARED_COUNTER = ({a,b}:CTR || {a,b}::VAR || NO_INTERFERENCE).
```

- property NO_INTERFERENCE violation

With lock

```
const N = 4
range T = 0..N

VAR = VAR[0],
VAR[u:T] = (read[u] -> VAR[u] | write[v:T] -> VAR[v]).

LOCK = (acquire -> release -> LOCK).
CTR = ( acquire -> read[x:T] ->
        ( when (x<N) increment -> write[x+1] -> release -> CTR
        | when (x==N) release -> END
        )
      )+{read[T], write[T]}.

property NO_INTERFERENCE = ({a,b}.write[v:T] -> (when (v<N) {a,b}.write
    [v+1] -> NO_INTERFERENCE)).

||LOCKED_SHAREDCOUNTER = ({a,b}:CTR || {a,b}::(LOCK||VAR) ||
    NO_INTERFERENCE).
```

- no deadlocks/errors produced, i.e. no interference
- gives more confidence that there is no interference cf. INTERFERENCE process

Safety: Mutual Exclusion with Semaphores

- model of M concurrent loops requiring access to a critical section
- each loop executes the following (mutex = binary semaphore)
  - up/signal: return permit
  - down/wait: acquire permit
  - enter/exit: entering/exiting critical region
- safety property defined for mutual exclusion

```
// example of mutual exclusion with 10 processes attempting to access
    critical region, using binary semaphore

// number of loops
const M = 10

// up/signal: return permit
// down/wait: block until permit acquired
SEMAPHORE(X=1) = SEMAPHORE[X],
SEMAPHORE[i:0..X] =
    ( when (i < X) signal -> SEMAPHORE[i+1]
    | when (i > 0) wait -> SEMAPHORE[i-1]
    ).

```

```
14 LOOP = (mutex.wait -> enter -> exit -> mutex.signal -> LOOP).
15
16 // check safety property: mutual exclusion, when a process enters
      critical
17 // region, the same process must exit critical region
18 property MUTEX = (p[i:1..M].enter -> p[i].exit -> MUTEX).
19
20 // compose M loops with binary semaphore
21 ||M_LOOPS = (  p[1..M]:LOOP
22              || {p[1..M]}::mutex:SEMAPHORE(1)
23              || MUTEX
24              ).
```

- now let semaphore have 2 permits: we get a MUTEX property violation, as multiple processes can enter critical region.
- this approach of deliberately introducing an error is useful for checking safety violations are detected as expected (i.e. that they are correctly specified.