Concurrent (Software) Architectures
SOFTWARE ARCHITECTURE

- **Architecture** is used as a term to describe the gross structure of a system/application in terms of the structure of *components* and *connectors* comprising the system.

- The **client-server** architectures used in previous sections are examples of a standardised structure for a system.
  - the architecture can be described *without knowing the exact functionality and purpose of the system or its components*
  - only need info about how the components interact
  - can implement *framework* for system without knowing “internals”

- We look at some standard architectures for concurrent systems.
PIPS AND FILTERS

- A filter is a process that transforms a stream of inputs into a stream of outputs.
  - multiple input streams and multiple output streams possible
- Filters are connected together via pipes to form complex networks of processes, including feedback.
  - e.g., Unix
  - pipes may have buffered memory
- We study the classical algorithm called the sieve of Aritosthenes
  - provide a concurrent implementation via pipes and filters
SIEVE OF ARISTOTHEINES

- Find all the **primes** between 2 and n:
  - list all the numbers between 2 and n
  - cross out all the numbers in the list divisible by 2
  - move to the next uncrossed out number and remove all multiples from the list
  - repeat the previous step until the end of the list is reached
  - the uncrossed out numbers are the primes

- The primes act as a sieve for the other numbers ...
SIEVE OF ARISTOTHESES

- The concurrent version generates a stream of numbers.
- Multiples (non primes) are removed by filter processes.
- The architecture is depicted below (without action and process labels):
SIEVE OF ARISTOTHEMES

- We must define
  - the filtering processes
  - the interactions defined by pipes

- In the terminology of SW Architectures, pipes are connectors.
  - connectors define interaction between components in the architecture
  - both connectors and components are defined as processes in FSP

\[
\begin{align*}
\text{const } \text{MAX} & = 9 \\
\text{range } \text{NUM} & = 2..9 \\
\text{set } S & = \{[\text{NUM}], \text{eos}\} \\
\text{PIPE} & = (\text{put } [x:S] \rightarrow \text{get } [x] \rightarrow \text{PIPE})
\end{align*}
\]
SIEVE OF ARISTOTHESES

- The pipe processes numbers from 2 to MAX and the signal eos (end of stream signal).
- Need a new FSP mechanism: the conditional process.

```
if B then P else Q behaves as P if B is true and Q otherwise. If the 'else' is missing and B is false, it behaves like STOP.
```

GEN = GEN [2],
GEN [X:NUM] = (out.put [x] ->
    if x<MAX then
    GEN [X+1]
else
    (out.put.eos -> enf -> GEN)
).
```
SIEVE OF ARISTOTHEINES

- The filter process records the first value it gets and filters out multiples of that value.

\[
\text{FILTER} = (\text{in.get [x:NUM]} \rightarrow \\
\text{|in.get.eos} \rightarrow \text{ENDFILTER})
\]

\[
\text{FILTER [P:NUM]} = (\text{in.get [x:NUM]} \rightarrow \\
\text{if x\%p} \neq 0 \text{ then} \\
\text{(out.put [x] \rightarrow \text{FILTER [p]})} \\
\text{else \text{FILTER [p]}} \\
\text{|in.get.eos} \rightarrow \text{ENDFILTER})
\]

\[
\text{ENDFILTER} = (\text{out.put.eos \rightarrow end} \rightarrow \text{FILTER}).
\]
The composite structure is:

\[
\text{PRIMES (N=4) = } \\
\text{ (gen:GEN} \\
\text{ ||pipe [0..N-1]:PIPE} \\
\text{ ||filter [0..N-1]:FILTER} \\
\{ \text{pipe [0]/gen.out,} \\
\text{ pipe [0..N-1]/filter [i].in,} \\
\text{ pipe [i:1..N-1]/filter [i-1].out,} \\
\text{ end/\{filter [0..N-1].end, gen.end\}} \\
\} @\{\text{filter [0..N-1].prime, end}\}.
\]
SIEVE OF ARISTOTHESES

- No deadlocks or errors detected!
  - used ‘end’ to differentiate normal termination in the model so that any error or deadlock was detectable as being different

- Minimised LTS:

- NOTE: model does not compute all primes between 2 and MAX! Computes first N primes where N is number of filters! (Change MAX too 11 and recompute LTS ...)
SIEVE OF ARITOSTHENES

- Previous model uses single slot buffers.
- What happens if we use no buffering?
- Construct a model in which pipes are omitted and filters interact via shared actions (the LTS will be the same!):
  
  \[
  \text{PRIMESUNBUF (N=4)} =
  \]
  
  \[
  (\text{gen:GEN} || \text{filter [0..N-1]:FILTER})
  \]
  
  \[
  \{ \text{pipe [0]/gen.out.put, }
  \]
  
  \[
  \text{pipe [0..N-1]/filter [i].in.get, }
  \]
  
  \[
  \text{pipe[i:1..N-1]/filter[i-1].out.put, }
  \]
  
  \[
  \text{end/{filter [0..N-1].end, gen.end}}
  \]
  
  \[
  @{$\text{filter [0..N-1].prime, end}}.
  \]
ABSTRACTING APPLICATION DETAIL

- If we want to analyse larger models, state space explosion becomes a problem.
- SOLUTION: abstract details of application
  - in this case details of sieve function
- MAGIC: relabel range of values NUM as a single value (independence from values).

```plaintext
| | AGEN = GEN/{out.put/out.put [NUM]},
| | AFILTER = FILTER/ {out.put/out.put [NUM]
| | in.get/in.get [NUM],
| | prime/prime [NUM]
| | }.
| | APIPE = PIPE/{put/put [NUM], get/get [NUM]}. 
```
- Deterministic conditional transition replaced here by nondeterministic choice:
- In state 4 after an \texttt{in.get}, the LTS goes to state 5 and does an \texttt{out.put} action or remains in state 4.
MORE BUFFERS

- Use a recursive definition of pipes to simulate multi slot buffer:

\[
||\text{MPIPE (B=4)} = \\
\quad \text{if } B == 1 \text{ then} \\
\quad \quad \text{APIPE} \\
\quad \text{else} \\
\quad \quad (\text{APIPE/\{mid/get\}} \mid \mid \text{MPIPE (B-1)/\{mid/put\}}) \\
\quad \quad \text{@\{put, get\}.}
\]
MORE BUFFERS

- The abstract model of the sieve architecture can be defined as follows, using PRIMEP defined later:

\[
\text{APRIMES (N=4, B=3) =}
\]

\[
\text{(gen:AGEN || PRIMEP (N)}
\]

\[
\text{pipe [0..N-1]:MPIPE [B]}
\]

\[
\text{filter [0..N-1]:AFILTER}
\]

\[
)\{/ \text{ pipe [0]/gen.out, pipe [0..N-1]/filter [i].in, pipe [i:1..N-1]/filter [i-1].out, end/\{filter [0..N-1].end, gen.end}\}.
\]
ARCHITECTURAL PROPERTIES

- Properties of such abstract models of software architectures are called architectural properties.
  - absence of deadlock
  - termination (absence of livelock)

- Termination is assured by the progress property END:

  progress END = \{end\}

- An application specific property is that the prime from filter [0] should be produced before that from filter [1], etc

  property
  PRIMEP\(N=4\) = PRIMEP [0],
  PRIMEP[i:0..N] = (when \(i<n\)
  filter[i].prime -> PRIMEP[i+1] | end -> PRIMEP ).
SIEVES IMPLEMENTATION

- Aplet displays with an unbuffered implementation.
- Top numbers are latest numbers generated/received by corresponding filter.
SIEVES IMPLEMENTATION

- Number generator and filters are implemented as threads.
- Two implementations of the pipe connector.
SIEVES IMPLEMENTATION

- The display is handled by PrimeCanvas.

```java
class PrimesCanvas extends Canvas {

    // display val in an upper box numbered index
    // boxes are numbered from the left
    synchronized void print(int index, int val){...}

    // display val in a lower box numbered index
    // the lower box indexed by 0 is not displayed
    synchronized void prime(int index, int val){...}

    // clear all boxes
    synchronized void clear(){...}
}
```
SIEVES IMPLEMENTATION: Generator

class Generator extends Thread {
    private PrimesCanvas display;
    private Pipe<Integer> out;
    static int MAX = 50;

    Generator(Pipe<Integer> c, PrimesCanvas d) {
        out = c; display = d;
    }

    public void run() {
        try {
            for (int i=2;i<=MAX;++i) {
                display.print(0,i);
                out.put(i);
                sleep(500);
            }
            display.print(0,Primes.EOS);
            out.put(Primes.EOS);
        }
    }
}
class Filter extends Thread {
    private PrimesCanvas display;
    private Pipe<Integer> in,out;
    private int index;

    Filter(Pipe<Integer> i, Pipe<Integer> o, 
           int id, PrimesCanvas d) 
        {in = i; out=o;display = d; index = id;}

    public void run() {
        int i,p;
        try {
            p = in.get();
            display.prime(index,p);
            if (p==Primes.EOS && out!=null) {
                out.put(p); return;
            }
            while(true) {
                i= in.get();
                display.print(index,i);
                sleep(1000);
                if (i==Primes.EOS) {
                    if (out!=null) out.put(i); break;
                } else if (i%p==0 && out!=null) 
                    out.put(i);
            }
        } catch (InterruptedException e){}
    }
}
The implementation reuses classes developed earlier:

- synchronous message passing class `Channel`
- unbuffered pipes
- and bounded buffer class `BufferImpl` used to implement buffered pipes
private void go(boolean buffered) {
    display.clear();

    //create channels
    ArrayList<Pipe<Integer>> pipes = new ArrayList<Pipe<Integer>>();
    for (int i=0; i<N; ++i)
        if (buffered)
            pipes.add(new PipeImplBuf<Integer>());
        else
            pipes.add(new PipeImplUnBuf<Integer>());

    //create threads
    gen = new Generator(pipes.get(0), display);
    for (int i=0; i<N; ++i)
        filter[i] = new Filter(pipes.get(i),
                                i<N-1?pipes.get(i+1):null, i+1, display);
    gen.start();
    for (int i=0; i<N; ++i) filter[i].start();
}
USING BUFFERS OR NOT

- Unbuffered version of the sieve algorithm works correctly.
- Why use buffers?
  - Answer: efficiency!
  - Process or thread suspension involves a context switch and this is expensive.
    - (thread switch less expensive than process switch)
  - Concurrent programs run more efficiently if there are fewer context switches.
  - No buffering means generator and filter threads are suspended every time they produce an item until consumed by next thread.
  - With buffers, only a full buffer causes suspension.
SUPERVISOR-WORKER ARCHITECTURE

- Concurrent architecture to speed up execution of suitable applications.
  - applies when a problem can be split into a number of independent sub-problems
  - referred to as tasks
SUPERVISOR-WORKER ARCHITECTURE

- Supervisor and worker processes interact via a connector bag
  - supervisor puts initial set of tasks in bag
  - supervisor gets results from bag and determines when computation has finished
- Each worker repetitively
  - takes a task from the bag
  - computes the result and puts it back in bag
- Supervisor determines end of computation
- Workers can put new tasks into the bag
- Any number of workers is possible
LINDA TUPLE SPACES

- Interaction mechanism suitable for implementing bag
- Linda is a model of (parallel) computation
- Consists of a set of primitive operations used to access a data structure called a tuple space
- A tuple space is a shared associative memory with a collection of tagged data records
  ("tag", value1, ..., valuen)
  - the tag is a literal string used to categorize tuples
  - valuei is a data value like integer, floating point, other data values
- Basic operations:
  - deposit a tuple in the space `out("tag", expr1, ..., exprn)`
  - execution completes when expri are evaluated and tuple is put in the tuple space
similar to an asynchronous send, but “message” stored in tuple space instead of queue associated with a port

\textbf{in} ("tag", field1, ..., fieldn) removes tuple from space

each fieldi is either an expression or a formal parameter of the form \(?\text{var}\) (\text{var} local to executing process)

arguments to \textbf{in} are called templates

process executing \textbf{in} blocks until there is a tuple in the space to match the template

template match is defined as:

\begin{itemize}
  \item tags identical
  \item same number of fields
  \item expressions in template have same values as corresponding values in tuple
  \item variables in template have same type as corresponding values in tuple
\end{itemize}

like receive with matching used to identify “port”
LINDA TUPLE SPACES

- **rd**("tag", field1, ..., fieldn) reads a value from the tuple space without removing it
  - same conditions as **in**

- Linda provides nonblocking versions of **in** and **rd**, called **inp** and **rdp**, that return true tuple if tuple is there, false otherwise

- **eval** creates an active/process tuple
  - like **out**, but one of the arguments is a procedure that operates on other arguments
  - becomes a passive tuple when evaluation terminates
TUPLE SPACE MODEL

- Use finite set of tuple values
- Have to fix maximum number of copies of some tuple allowed
  
  ```
  const N = ...
  set Tuples = {...}
  ```

- $N$ and $Tuples$ depend on context
- Each tuple value modelled by FSP label of the form $tag.val1.....valn$
- Have a process to manage each tuple value and tuple space is parallel composition of all these processes
**TUPLE SPACE MODEL**

\[
\begin{align*}
\text{const } & \text{False} = 0 \\
\text{const } & \text{True} = 1 \\
\text{range } & \text{Bool} = \text{False..True} \\
\text{TUPLE } (T='any) & = \text{TUPLE } [0]. \\
\text{TUPLE } [i:0..N] & = \\
\quad (\text{out}[T]) & \to \text{TUPLE}[i+1] \\
\quad | \text{when } (i>0) \text{ in}[T] & \to \text{TUPLE}[i+1] \\
\quad | \text{when } (i>0) \text{ inp}[\text{True}][T] & \to \text{TUPLE}[i+1] \\
\quad | \text{when } (i==0) \text{ inp}[\text{False}][T] & \to \text{TUPLE}[i] \\
\quad | \text{when } (i>0) \text{ rd}[T] & \to \text{TUPLE}[i] \\
\quad \text{rdp}[i>0][T] & \\
\text{).}
\end{align*}
\]

\[\text{TUPLESPACE} = \text{forall } [t:\text{Tuples}] \text{TUPLE}(t).\]
TUPLE SPACE MODEL

- LTS for tuple value for \texttt{any} with N=2 is:
Exceeding capacity with more than 2 *out* ops leads to an ERROR.

Example of a conditional operation on the tuple space:

\[ \text{inp} [b:\text{Bool}][t:\text{Tuples}] \]

- value of \( t \) only valid when \( b \) is True.

Each specific TUPLE process has in its alphabet the ops on one specific tuple value.

The alphabet of TUPLESPACE is:

\[ \text{set TupleAlpha} = \{\text{in}, \text{out}, \text{rd}, \text{rdp[Bool]}, \text{inp[Bool]}.\text{Tuples}\} \]
TUPLE SPACE IMPLEMENTATION

- Implement as centralized system for demo
- Matching of templates only on tag field
- Use a hash table of vectors to implement the space
- For simplicity, tuples stored in FIFO order for a particular tag
- New tuples appended at end of a vector for a tag and removed from its head
- Naively, all threads woken up whenever a new tuple is added
- More efficiently, we would wake only those waiting for tuple with the same tag
public interface TupleSpace {

    // deposits data in tuple space
    public void out (String tag, Object data);

    // extracts object with tag from tuple space, blocks if not available
    public Object in (String tag)
        throws InterruptedException;

    // reads object with tag from tuple space, blocks if not available
    public Object rd (String tag)
        throws InterruptedException;

    // extracts object if available, return null if not available
    public Object inp (String tag);

    // reads object if available, return null if not available
    public Object rdp (String tag);
}
class TupleSpaceImpl implements TupleSpace {
    private Hashtable tuples = new Hashtable();

    public synchronized void out(String tag, Object data) {
        Vector v = (Vector) tuples.get(tag);
        if (v == null) {
            v = new Vector();
            tuples.put(tag, v);
        }
        v.addElement(data);
        notifyAll();
    }

    private Object get(String tag, boolean remove) {
        Vector v = (Vector) tuples.get(tag);
        if (v == null) return null;
        if (v.size() == 0) return null;
        Object o = v.firstElement();
        if (remove) v.removeElementAt(0);
        return o;
    }
}
public synchronized Object in (String tag) 
    throws InterruptedException {
    Object o;
    while ((o = get(tag, true)) == null) wait();
    return o;
}

public Object rd (String tag) 
    throws InterruptedException {
    Object o;
    while ((o = get(tag, false)) == null) wait();
    return o;
}

public synchronized Object inp (String tag) {
    return get(tag, true);
}

public synchronized Object rdp (String tag) {
    return get(tag, false);
}