Concurrency - Complements

Alexandre David
adavid@cs.aau.dk

1 - Dynamic Systems

- So far, threads
  - are created at initialization
  - run until termination
  - are statically organized (like monitors)
- Now, threads
  - are created and terminate dynamically ⇒ number of active threads varies (common situation in OS)

Problems

- How to model and program such systems?
  - Resource allocation problem: variable amount of resource needed to proceed.
  - Modeling problem: What is the relevance of finite state models to model dynamic systems?
    - Hint: Computers have limited resources…

Problems

- Processes are static in FSP (dynamic in Promela) in structure and number of processes – limits of tools for analysis.

Program vs. Model

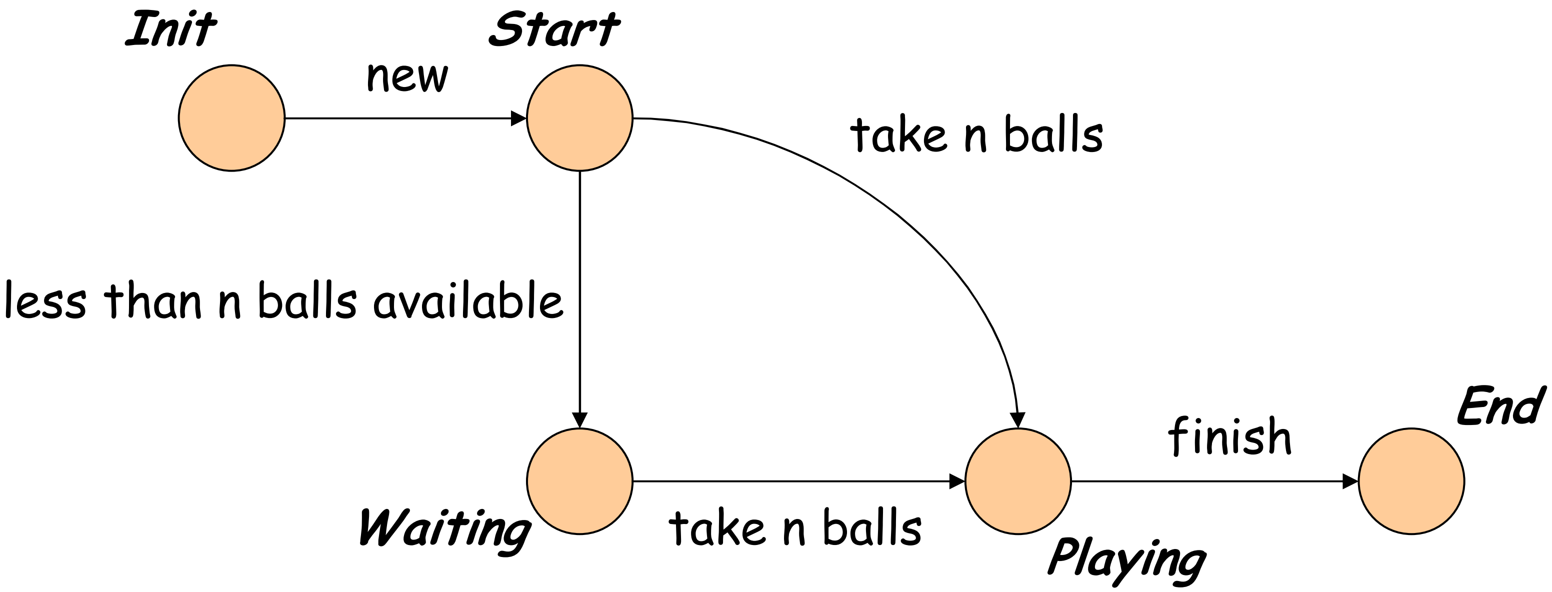
- How much of behaviour of the dynamic system is captured in the static model?
- Is the static model helpful in analysing the behaviour of the dynamic system?

- Let’s answer these questions.
  Golf club example in the book.

Golf Club Example

- Players come to a golf club, hire golf balls, play, and return them.
- Infinite stream of players, limited number of balls.
- *Model: limited number players.*
- Implementation: players are threads that are created dynamically.

Golf Player



- This corresponds to the implementation.
- Starvation problem in ***Waiting***.

Allocation of Balls

```
synchronized public void get(int n)
      throws … // needs n balls
{
      while (n > available) wait();
      available -= n;
}
synchronized public void put(int n)
{
      available += n;
      notifyAll(); // several blocked players
}
```

```
const N = 5
ALLOCATOR = BALL[N],
BALL[b:0..N] = when (b > 0) get[i:1..b]->BALL[b-i] |
put[j:1..N]->BALL[b+j]).
```

Players

- Player thread as usual: *while* loop with
  - get balls
  - sleep
  - give back balls
- Model: how to model the infinite stream of players? We cannot represent an infinite state space in this case but it’s fine with infinite behaviours that are repetitive.

Solution for Modeling

- We don’t need distinct players. It’s fine with a fixed population of players.
- Model: infinite stream of requests from finite set of golfers (~ real implementation since threads are recycled).
- System: finite stream of requests from infinite number of players.
- **This is a very common general technique.**

Player Model

```
range R = 1..N
PLAYER = (need[b:R]->PLAYER[b]),
PLAYER[b:R] = (get[b]->put[b]->PLAYER[b]).
set Experts = {Alice, Bob, Chris}
set Novices = {Dave, Eve}
set Players = {Experts, Novices}
HANDICAP = ({Novices.need[3..N], Experts.need[1..2]}
             ->HANDICAP)+{Players.need[R]}.
```

- Different kinds of players, modeled by the HANDICAP process.
- Progress check: put low priority on put action.

Solving Starvation

- Ticket protocol: tickets in ascending order (like post office).
  Model: round number % # players.
- But: increase size of the model… may need to simplify.
- Not very efficient in the sense that novices may block many experts unnecessarily.

Fair Allocator

```
private long turn = 0; // next ticket to be dispensed
private long next = 0; // next ticket to be served
synchronized public void get(int n)
      throws … // needs n balls
{
      long myturn = turn; ++turn;
      while (n > available || myturn != next) wait();
      ++next; available -
      notifyAll();
}
synchronized public voi
{
      available += n;
      notifyAll(); // se
}
```

No starvation but resources are not used efficiently: expert players are kept by novices although the balls they require are available.

Bounded Overtaking

- We allow experts to overtake novices and we prevent starvation by setting an upper bound on the number of times a novice can be overtaken.
- Idea: a thread has been overtaken if *next>=(myturn+bound)*, in which case a variable *overtaken* is incremented and all other threads are blocked.

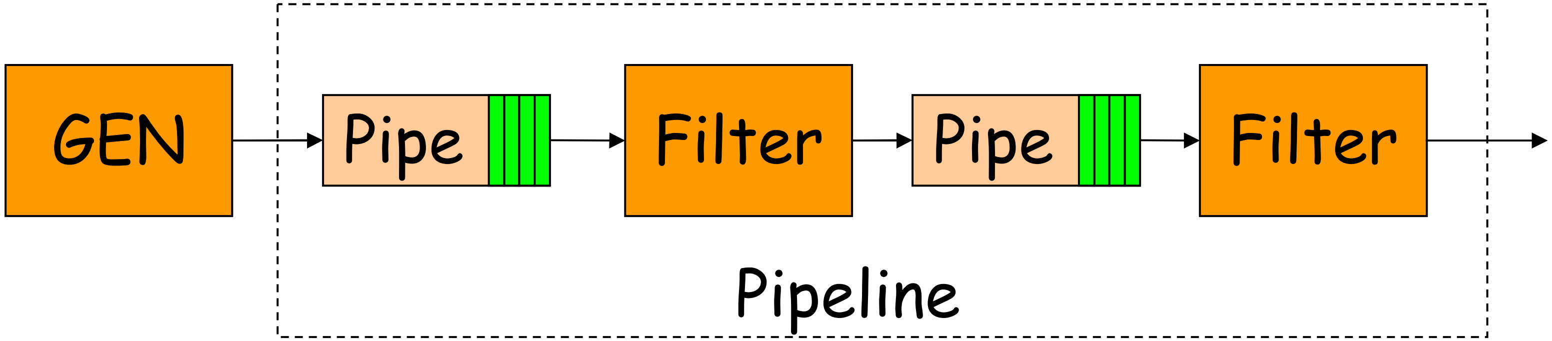
Master-Slave

- In some situations a *master* thread may ask to a (dynamically created) *slave* thread to compute something.
  - the master continues with some activity
  - the slave terminates
  - the master collects the result later
    - can poll with **isAlive()**
    - better: can synchronized with **join()**

2 – Concurrent Architectures

- Filters: component that processes incoming stream(s) of data and output results.
- Filters can be implemented as processes, e.g., pipes in UNIX.
- Very convenient and powerful to implement complex computations from simple operations.

Primes Sieve Example



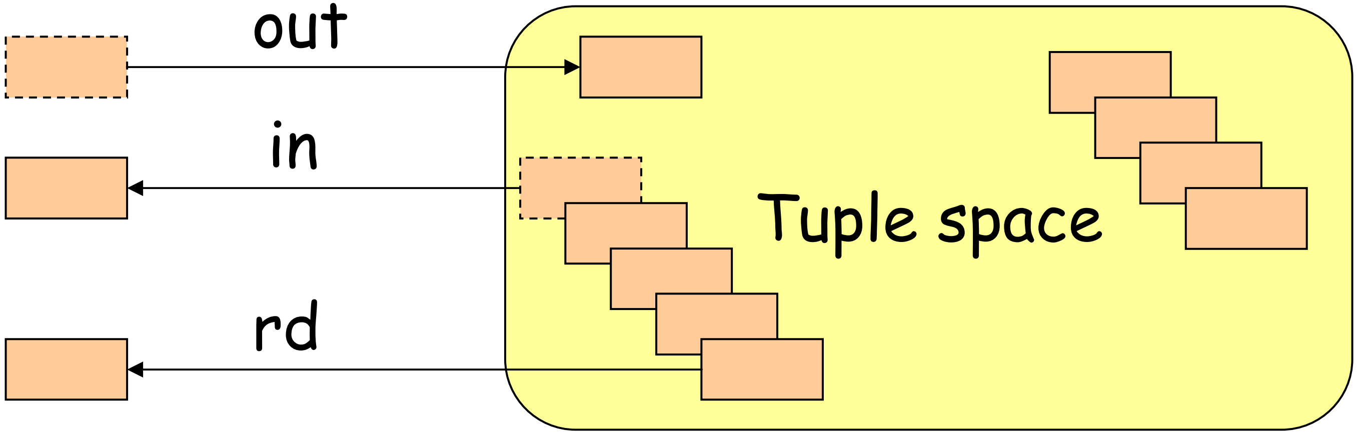
More efficient with buffered pipes (reduces context switches). Pipes in UNIX are buffered.

Supervisor-Worker

- Good to speed up execution of computational problems where it is possible to split the main problem into *independent sub-problems* to be solved in *parallel*.
- Supervisor manages a set of *tasks* to be handled by the workers.
- Workers can generate new tasks as results.

Linda Tuple Space

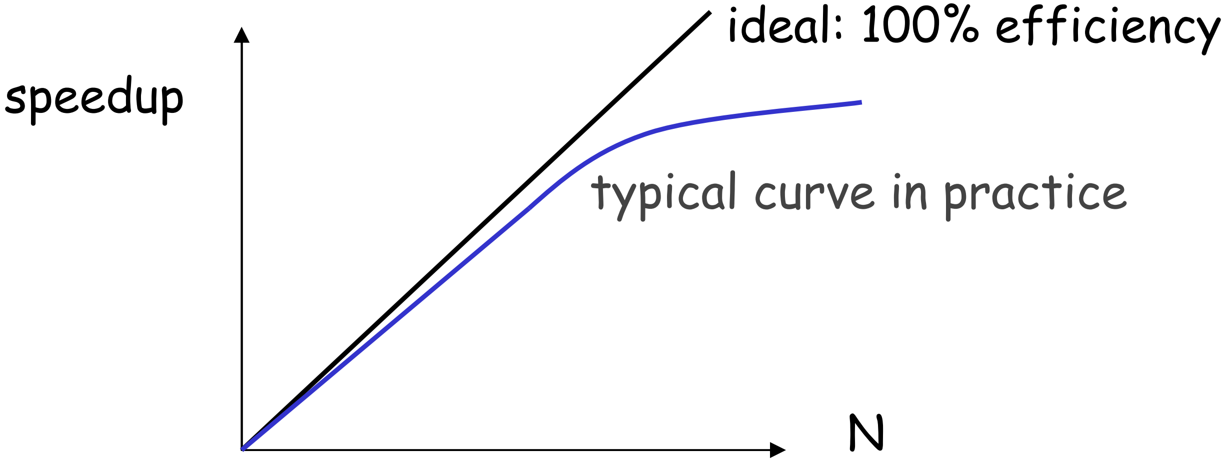
- Name of a distributed shared memory system. Data is organized as *tuples* of the form (“tag”, value, value, …).
- Can be used to implement the set of tasks.

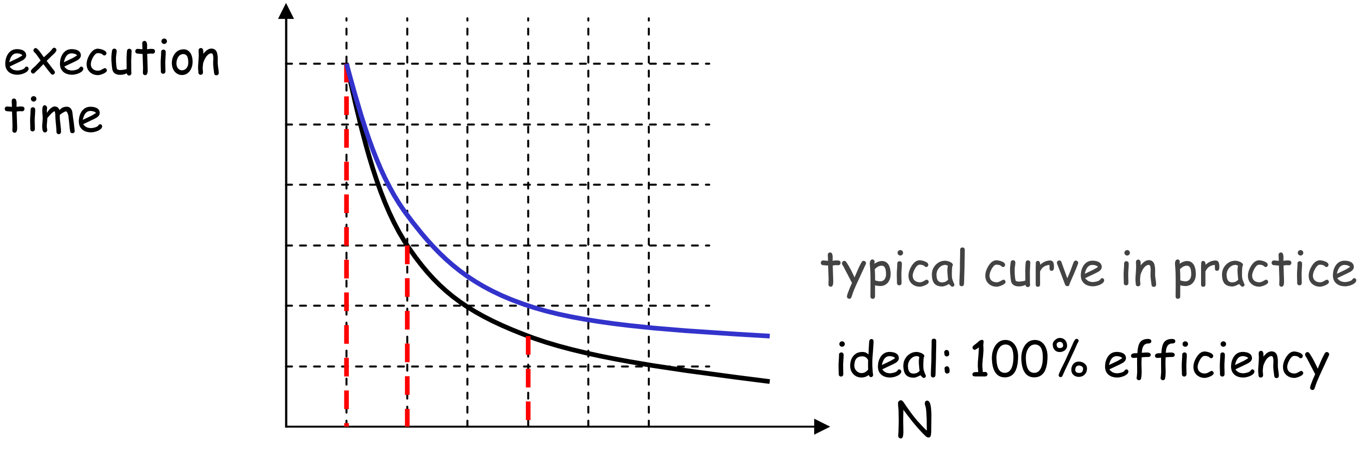


Speedup & Efficiency

- *Speedup = time(1)/time(N)*
  where *time(n)* is the time used to solve a problem on n processors.
- *Efficiency = Speedup/N*
  measures how efficiently the problem is divided. Ideally, the speedup is *N*, which corresponds to 100%.

Speedup & Efficiency





Patterns

- Many examples that we have seen in this course follow *programming patterns*.
- Chapter 11 gives some basic patterns.
- If you want to know more, check books on programming patterns.

3 – Timed Systems

- In fact *Real-time* systems: correctness of the systems is defined as the correct output must be delivered *in due time*.
- Time often discretized as ticks in practice. Even if ticks are not used the implementation of time is always discrete (discrete clocks).

Timed Systems

- Chapter 12 is about modeling time with an un-timed tool, which means using a number of hacked models with a particular interpretation.
- Better tools exist specifically to handle time, e.g., UPPAAL.
- Examples of chapter 12 are interesting since they provide an analysis with models, followed by an implementation.

4 – Operational Semantics

- Appendix C gives the semantics of FSP in terms of *rules*.
- How to read them:

$$\frac{\textit{expression before}}{\textit{result expression}} \quad \textit{condition}$$

5 - Equivalence

- You noticed the functionality of the tool to *minimize* automata. What it means: it computes a smaller automaton (if possible) that exhibit *exactly the same* behaviour of the original automaton.
- **Important point** in the definition (C.6.1): whatever P does something, Q can do the same, **and vice-versa**.
- Weak equivalence: ignore *tau* actions.