98-023A : Concurrent and Distributed Programming w/ Inferno and Limbo

Phillip Stanley-Marbell
pstanley@ece.cmu.edu

Lecture Outline

- Communicating Sequential Processes (CSP)
    - Overview of the 1978 paper by C.A.R. Hoare

Syllabus

- Week 1: Introduction to Inferno
- Week 2: Overview of the Limbo programming language
- Week 3: Types in Limbo
- Week 4: Inferno Kernel Overview
- Week 5: Inferno Kernel Device Drivers
- Week 6: NO CLASS
- Week 7: C applications as resource servers: Built-in modules and device drivers
- Week 8: Case study I — building a distributed multi-processor simulator
- Week 9: Platform independent Interfaces: Limbo GUIs; Project Update Spring Break
- Week 10: Programing with threads, CSP
- Week 11: Debugging concurrent programs; Promela and SPIN
- Week 12: Factotum, Secstore and Inferno's security architecture
- Week 13: Case study II — Edisong, a distributed audio synthesis and sequencing engine

Background

- Charles Antony Richard Hoare
  - Quicksort sorting algorithm (1961)
  - Elliot Algol compiler
  - Hoare Logic, Axiomatic Semantics
  - Knighted by the Queen (so he has his own coat of arms ?)

Program Structures

- Programs compute, interact with real world via I/O
- Primitive program structures capture computation
  - *Repetition*
  - *Choice*
  - *Sequencing*
- I/O has generally been 'tacked on'
- Programs execute on hardware, hardware inherently concurrent
  - Even more true when dealing with multiprocessors (which were looking really promising in 1978)

Background: Hardware

- Hardware used to be very expensive

- Rather than implement solution with lots of hardware, reuse blocks of hardware in time
  - Blocks implemented specific tasks or "instructions" which are used over and over
  - Timing of this hardware reuse (in time) usually driven by a clock
  - Hence ISA and clock driven computation as we know it today

- Benefits of multiprocessors and spatial computation
  - Performance *(If your workload has parallelism)*
  - Fault-tolerance *(Still run though individual processor may fail)*
  - Parallel computation can be more energy efficient [A. Martin et al., 2001]

Communicating Sequential Processes

- Previously, communicating components in a multiprocessor used primitives such as
  - Communicate through shared variables : requires synchronization as a separate action
- CSP: Single solution to both communication and synchronization
  - Guards
  - Parallel composition
  - Synchronous (i.e., blocking, unbuffered) I/O on 'Channels'
  - Pattern matching
- Context (1978)
  - Dijkstra's guarded commands
  - Doug McIllroy (irked Ken to implement pipes) : coroutines
  - Algol 60, Pascal

Commands

- Notion of command success and failure

- Null commands
  - Do nothing: `skip`

- Simple Commands

```
x := 5
a : integer; Time?a
console!’c’
```

- Structured Commands

```
a : integer; *[a := 0;  Time?a -> skip;]
```

- Command Lists
  - `n, d, pi: integer; n := 22; d:= 7; pi := n/d;`

Processes and Parallel Composition

- Process is the basic unit of concurrency
  - It is essentially a named command list that can be composed with others
- Process Label
  - This is the process name
  - Used to specify parallel composition of processes
  - Used in communication
  - e.g., `SLAVE :: SLAVEcode`
  - Where `SLAVEcode` is `[MASTER(0)?c; MASTER(i)!sample]`
- Parallel Composition

```
[SLAVE(1..5)::SLAVEcode || MASTER :: MASTERcode]
```

Channels

- Channels *per se* don't exist
  - Communication is on process name (process label)
  - May be subscripted to denote separate channels in process
- Channels are both for communication and synchronization
  - Input and output are synchronous (you can implement buffering in a process)
  - Each send must be matched by a receive to succeed and vice versa
  - Structured value with no type (called a "signal") can be passed on channel
- Receive from process (input)
  - `PROCESSname?`*target variable*
  - e.g., `console(42)?key`
- Send to process (output)

```
PROCESSname!expression
console(42)!ack
```

Alternation and Repetitive Commands

- Repetition
  - * <*alternative command*>
  - Repeat <*alternative command*> until it fails
  - Alternative command made up of guarded command. Fails when all guards fail

- Alternation
  - pick (fairly) one constituent guarded command whose guard succeed
  - Syntax: `GUARD1 → COMMAND1`□`GUARD2`→ `COMMAND2 ...`□`GUARD`*n*→ `COMMAND`*n*

- Example

Structure Matching

- Pattern matching on the structure of terms
  - Assignment commands fail if the LHS and RHS don't have the same structure
  - `x := x+1` (will not fail)
  - `c := P()` (`c` must have same *structure* as constructor `P`, else command fails)
  - `P() := c` (`c` must have the *value* `P()`, otherwise this command fails)
  - `CONSTRUCTOR1(n) := CONSTRUCTOR2(n)` (will fail because LHS and RHS have different structure, since they have different constructors)

- Communication (`?`, `!`) will fail if structure does not match

Implementing Coroutines, Subroutines and Monitors

- Coroutines
  - Rather than a caller-callee organization, both routines run simultaneously with control passing between them
- Subroutines (functions)
  - Implemented in CSP as communication
  - Send arguments to process (via its label)
  - Receive results from process (via its label)
  - (Each process assumed to be servicing only one user)
- Monitors
  - Process serving several users
  - Users connect via distinct channels (process label subscripts) or must have distinct names known to monitor

Examples

- Coroutines: Squash

- Subroutines: Integer division w/ remainder

- Monitors: Bounded buffer

End Notes *(things to think about)*

- CSP was not meant to be a "complete" programming language
  - Paper is about an *idea*, CSP

- Some issues
  - Programming in-the-large : how to connect to processes if you do not know names *a priori*

- Bounds on processes
  - CSP : bounded number of processes (as defined statically in program source)
  - Dynamic creation of processes absent
  - Should the system be the only endpoint for controlling processes ?

Fin.