Aula 1

1 Disciplina

Objectivos

- descrição formal dos conceitos básicos de concorrência e sua aplicação
- raciocinar sobre a correção de programas concorrentes em relação a especificações
- desenho e análise de programas concorrentes

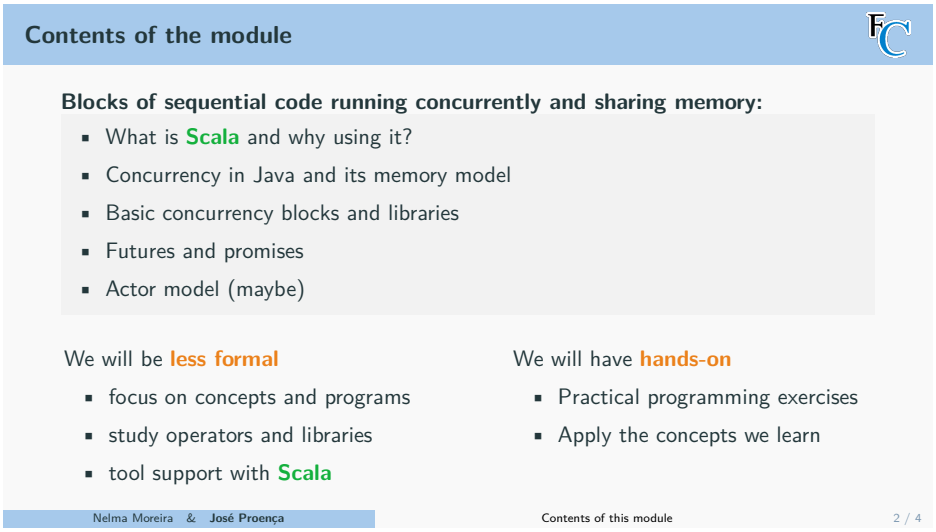
Parte 1

- Conceitos básicos de programação concorrente
- Noções básicas de concorrência
- Sistemas de transições
- CCS: *Calculus of Communicating Systems*: sequência, composição, sincronização; restrição e reetiquetagem; parâmetros e dados
- Comportamento observável
  - relações de equivalência, congruência, bisimulações;
  - congruência observável
  - propriedades algébricas
- Problemas de sincronização: exclusão mútua, deadlock, locks, etc.

Bibliografia e Software

- Reactive systems modelling, specification and verification. Luca Aceto, Anna Ingólfsdóttir, Kim Guldstrand Larsen, Jiri Srba 2007 (Cap. 1-4 e 7)
- Introduction to Concurrency Theory. Roberto Gorrieri and Cristian Versari 2015
- Communication and Concurrency, Robin Milner. Prentice Hall International Series in Computer Science, 1989.
- Modelação usando simuladores de CCS, p.e.:
  - pseuco.Com
  - CAAL

Concurrent Programming - Part II



Programação Concorrente- CC3040

Páginas Web

Sigarra: `https://sigarra.up.pt/fcup/pt/ucurr_geral.ficha_uc_view?pv_ocorrencia_id=529871`

URL Geral: `https://moodle2324.up.pt/`

Específicas:

Parte 1:

`https://www.dcc.fc.up.pt/~nam/Teaching/progcon2324/`

Pass: *bisimulation*

Part II:

`https://fm-dcc.github.io/pc2324/`

Email: institucional

Escolaridade: 2T e 2PL

Frequência: 3/4 de presenças nas aulas PL

Método de avaliação

1. Avaliação distribuída sem exame final
2. Testes (70%) e trabalho (30%)

3. 1 parte:
   - Teste (40%) (nota mínima: 6 valores em 20): DATA 1
   - Trabalho (10%)
4. 2 parte:
5. - Teste (30%) (nota mínima: 6 valores em 20)
   - Trabalho (20%)
6. Exame de recurso: 70% (40%+30%) + Trab (30%)
7. Exame de melhoria: ou 100% ou por como recurso

Programas Sequenciais

- realizam uma função dos dados nos resultados (tese de Church/Turing)
- A sua semântica pode ser analisada considerando o estado (memória) em cada instante:
  $$\mathcal{S}[\![P]\!] : State \to State$$
  onde p.e. $State = [Var \to \mathbb{Z}]$.
  P :
  $x \leftarrow 1$
  $y \leftarrow 0$
  while $x < 10$ **do**
  $y \leftarrow y + x$
  $x \leftarrow x + 1$
  print y
- Terminam (se não terminarem a sua semântica é indefinida)

Equivalência de programas sequenciais

Dois programas são equivalentes se realizam a mesma função. P :
$x \leftarrow 1$

Q :
$x \leftarrow 0$
$x \leftarrow x + 1$

P e Q são equivalentes assim como são equivalentes a

- $P; Q$
- $Q; P$
- $R; P$ e $R; Q$ (para qualquer programa R)
- ...

Programas Sequênciais vs Concorrentes

P :

$x \leftarrow 1$

Q :

$x \leftarrow 0$
$x \leftarrow x + 1$

- Mas se os executarmos em paralelo, $P||Q$?
- Qual o significado? P e Q podem intercalar
  1. não é único:
     - pode ser 1: se for P e depois Q
     - ou 2: se for Q (primeira instrução) , depois P e depois Q (segunda instrução)
  2. a semântica não é determinística
  3. a equivalência não é preservada por $||$.
  4. não é composicional (a semântica de um composta com a semântica do outro)

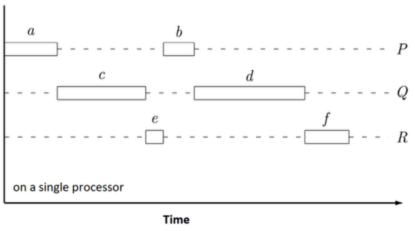
Programas Concorrentes/Sistemas Reactivos

- Normalmente não calculam uma função
  - Sistemas de operação
  - Protocolos de comunicação
  - Sistemas Web
  - Sistemas embebidos
  - Processadores multicore
  - Sistemas de controlo de tráfego
  - Portagens
  - ...
- Então o que fazem?
- Interagem com o ambiente e entre eles, trocando informação.
- Normalmente não terminam.
- Componentes básicas: Processos ou Agentes

Concorrência vs Paralelismo

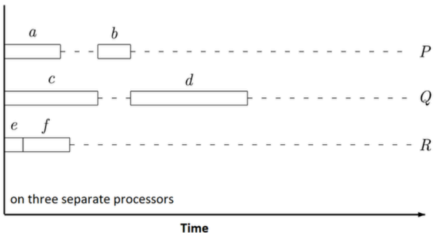
Concorrência

- Trabalho lógico simultaneo
- Não obriga a multiprocessador



Paralelismo

- Trabalho físico simultaneo
- Obriga a multiprocessador ou várias unidades de processamento.



Processos

- Um *processo* é um programa (sequencial) em execução
- É descrito por uma máquina de estados (estado= memória, contador de programa, etc)
- Um *programa multiprocesso* comporta-se como um conjunto de máquinas de estados que cooperam através da comunicação com o meio.
- se cada processo tiver um processador, os processos podem executar em paralelo
- Senão, tem de haver um *escalonador* para atribuir processos a processadores
- Supomos sistemas asíncronos onde *o tempo de execução não interessa*

Síncronização de Processos

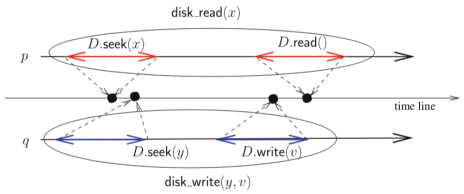
- Quando o progresso de um ou mais processos depende do comportamento de outros processos
- As interações podem ser de dois tipos:
  1. competição
     - Ex: competição por um recurso partilhado
  2. cooperação
     - O progresso de um processo depende do progresso de outros
     - Ex: *rendezvous*: conjunto de pontos de controlo em que cada processo só pode avançar quando todos os processos estiverem no ponto de controlo respectivo.
     - Ex: *produtor/consumidor*

Competição: Ler e escrever num disco D

procedure DISK-READ(x)
  $D.seek(x)$;
  $r \leftarrow D.read()$;
  return r;

procedure DISK-WRITE(x,v)
  $D.seek(x)$;
  $D.write(v)$;
  return;

DISK-READ(x) || DISK-WRITE(y,v)



Pode acontecer que se leia em x o valor de y.

Solução: Não permitir que estas operações executem em simultaneo → *Exclusão Mútua*

Cooperação:produtor/consumidor

- O produtor *produz* produtos

- O consumidor *consume* os produtos, e
- Um produto não pode ser consumido *antes* de ser produzido
- Todos os produtos que são produzidos são consumidos *exactamente* uma vez
- Implementação: um *buffer* partilhado de tamanho $k \geq 1$
- Pode ser uma fila: o produtor acrescenta um novo produto no *fim* da fila e o consumidor consome o produto do *início* da fila
- O produtor tem de esperar quando o buffer *está cheio*
- O consumidor só tem de esperar quando o buffer *está vazio*
- *Invariante de sincronização*: se $\#p$ número de produtos produzidos e $\#c$ número de produtos consumidos:

$$(\#c \geq 0) \wedge (\#p \geq \#c) \wedge (\#p \leq \#c + k)$$

Exclusão Mútua

- *Secção Crítica*: porção de código que só pode ser executado por um processo num dado instante
- Supõe-se que a execução da secção crítica por um só processo termina.
- *MUTEX* o problema consiste em ter
  1. um algoritmo de entrada $acquire\_mutex()$
  2. um algoritmo de saída $release\_mutex()$
- Enquadrando a seção crítica garantem

  Exclusão mútua : que o código da zona crítica é executado no máximo por um processo em cada instante.

  Starvation-freedom : cada processo que invoca $acquire\_mutex()$ termina, permitindo assim que os processos que querem entrar na zona crítica o possam fazer.

Exclusão Mútua

procedure $protected\_code(in)$
  $acquire\_mutex(\ )$;
  $r \leftarrow cs\_code(in)$;
  $release\_mutex(\ )$;
  return r;

Propiedades de *Safety* e *Liveness*

***Safety* (segurança)** Nada de mal acontece. Podem ser invariantes. Têm de ser sempre verdade. Ex: Exclusão mútua

***Liveness* (vivacidade)** Algo de bom irá acontecer. Terão de acontecer ao longo da execução do sistema.

- *Starvation-freedom*
- *Deadlock-freedom*: em cada instante τ se vários processos invocaram $acquire\_mutex$ e essa invocação não terminou, então para $\tau' > \tau$ algum terá que terminar essa invocação.
- *Bypass limitado*: Suponhamos n processos em competição e suponhamos que um ganha. Existe $f(n)$ tal que cada processo que invoca $acquire\_mutex$ perde no máximo $f(n)$ vezes para os outros.

Bypass limitado $\rightarrow$ *Starvation-freedom*(=*Bypass* finito) $\rightarrow$ *Deadlock-freedom*