Aula 1

1 Disciplina

Objectivos

- descrição formal dos conceitos básicos de concorrência e sua aplicação
- raciocinar sobre a sua correção 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
- pseuCo: linguagem de programação para agentes concorrentes
  - pseuCo e CCS
  - cooperação por passagem de mensagens (canais síncronos e assíncronos)
  - cooperação por partilha de memória (Mutex, locks e monitors)

Bibliografia e Software

- Reactive systems modelling, specification and verification. Luca Aceto, Anna Ingólfsdóttir, Kim Guldstrand Larsen, Jiri Srba 2007
- 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: pseuCO.com, CAAL ou CWB

Concurrent Programming - Part II

Design and implementation of concurrent code using the shared-memory thread model:

- The thread model.
- Correctness properties: mutual exclusion, atomicity, consistency models, linearizablity.
- Recurrent problems and related "bug patterns": race conditions, deadlocks, atomicity violations, ...
- Concurrency primitives: locks, monitors, atomic instructions, barriers, semaphores, condition variables, futures, ...
- Concurrent objects and data structures.

Concurrent Programming - Part II

Pratice:

- Working language: we will use Java.
- Testing concurrent code using the Cooperari system.
- Programming project(s).

Concurrent Programming - Part II

Books:

- The art of multiprocessor programming - Revised reprint M. Herlihy and N. Shavit, Morgan Kaufmann, 2012.
- Concurrent Programming: Algorithms, Principles, and Foundations, Michel Raynal, 2012.
- Concurrent Programming in Java: Design Principles and Pattern, 2nd Edition, Doug Lea, Addison-Wesley, 1999.

Programação Concorrente

URL:`www.dcc.fc.up.pt/~nam/Aulas/procon`

Escolaridade: 2T e 2PL

Método de avaliação

1. Avaliação distribuída com exame final

2. Cada parte: 10 valores
3. Em cada parte: 4 valores de avaliação distribuída e 6 valores em Exame

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

- dois programas são equivalentes se realizam a mesma função.

Programas Sequenciais vs Concorrentes

P :
$x \leftarrow 1$

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

- P e Q são equivalentes assim como equivalentes a $P;Q$, $Q;P$ ou $R;P$ e $R;Q$, ...
- Mas se os executarmos em paralelo? $P||Q$?
- Qual o significado?
  1. não é único: pode ser 1 ou 2
  2. não determinístico
  3. a equivalência não é preservada por $||$.
  4. não é composicional

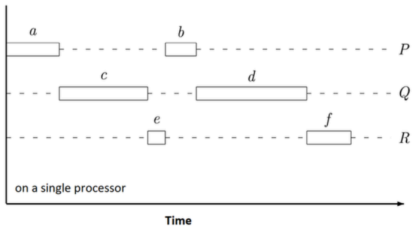
Programas Concorrentes/Sistemas Reactivos

- 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

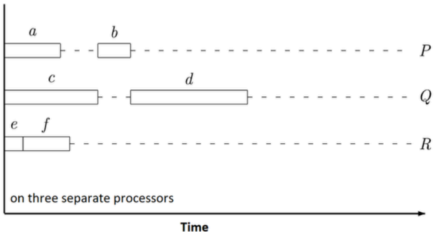
Concorrência

- Trabalho lógico simultaneo
- Não obriga a multiprocessador



Paralelismo

- Trabalho físico simultaneo
- Obriga a multiprocessador ou unidades de proceasamento.



Processos

- Um *processo* é um programa (sequencial) em execução
- É descrito por uma máquina de estados (ex: 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 *o tempo de execução não interessa*

Síncronização de Processos

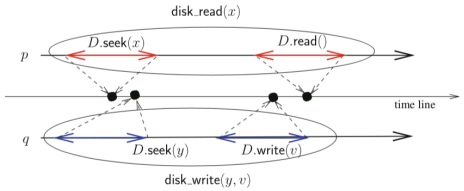
- Quando o progresso de um ou mais processos depende do comportamento de outros processos
- As interações podem ser de dois tipos:
  1. competição
     - Ex: competição por um recurso partilhado
  2. cooperação
     - O progresso de um processo depende do progresso de outros
     - Ex: *rendezvous*: 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 cítica 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

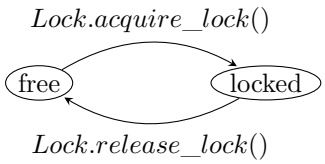
Liveness **(vivacidade)** Terão de acontecer ao longo da execução do sistema.

- *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*

Lock

- Objecto partilhado que permite implementar MUTEX
- tem dois métodos: $Lock.acquire\_lock()$ e $Lock.release\_lock()$
- tem dois estados: *free* e *locked*
-



Objectos Concorrentes

- Um objecto concorrente é um objecto que pode ser acedido por diversos processos
- A sua especificação é sequencial
- Ex: um stack é definido pelos métodos $push()$ e $pop()$, sendo uma descrição uma sequência de $push()$ e $pop()$
- Um stack concorrente pode ser implementado usando locks.
- **procedure** $C\_stack.conc\_push(i, v)$
  $Lock.acquire\_lock(i)$;
  $S\_stack.push(v)$;
  $Lock.release\_lock(i)$;

  procedure $C\_stack.conc\_pop(i)$
  $Lock.acquire\_lock(i)$;
  $r \leftarrow S\_stack.pop(\ )$;
  $Lock.release\_lock(i)$;
  return r;

Semáforo S

- Tem dois métodos atómicos $S.down()$ e $S.up()$ tal que
- S é inicializado com um valor $s_0 \geq 0$
- $S \geq 0$ é sempre verificado
- $S.down()$ diminui S por 1 (atómicamente)
- $S.up()$ incrementa S por 1 (atómicamente).
- A operação $S.up()$ pode ser sempre realizada
- A operação $S.up()$ pode ser realizada se $S \geq 0$ for mantido.
- Se uma operação não puder ser realizada fica o processo fica bloqueado
- Invariante: sendo $\#S.down$ ($\#S.up$) o número de invocações,
$$S = s_0 + \#S.up - \#S.down$$
- Um lock é um semáforo com $s_0 = 1$ onde $S.down$ é $acquire\_lock$ e $S.up$ é $release\_lock$

Implementação de um Semáforo

- Um contador: $S.count$ inicializado em s_0
- uma fila: $S.queue$ que é iniciada em 0
- **procedure** $S.down(\ )$
  $S.count \leftarrow S.count - 1$
  if $S.count < 0$ **then** ▷ O processo bloqueia e é adicionado à fila $S.queue$
  return;
  procedure $S.up(\ )$
  $S.count \leftarrow S.count + 1$
  if $S.count < 0$ **then** ▷ remover o primeiro processo da $S.queue$ e atribuí-lo a um processador.
  return;

Semáforos

- Sendo $nb\_blocked(S)$ o número de processos bloqueados em $S.queue$, o invariante é
  if $S.count \geq 0$ **then**
  $nb\_blocked(S) = 0$
  else
  $nb\_blocked(S) = S.count$
- notar que se $S.count \geq 0$ o seu valor é o do semáforo S.

Monitores

- Um *monitor* garante que no máximo uma operação interna é invocada em cada instante
- Suponhamos um recurso que tem de ser acedido em exclusão mútua
- Podemos definir um monitor que contenha o recurso e definir um método $use\_resource()$ que o monitor oferece aos processos.
- Para tal usam-se *condições* C dentro do monitor que têm os seguintes métodos que podem ser invocados pelos processos: $C.wait()$, $C.signal()$ e $C.empty()$.

Monitores

$C.wait()$: o processo pára a execução e espera numa fila C; o monitor liberta o MUTEX

$C.signal()$ Se nenhum processo está bloqueado na fila C, não tem efeito.Senão é reactivado o primeiro processo bloqueado. Mas para não ficarem dois processos activos no monitor, o processo que invocou o $C.signal()$ fica inativo mas tem prioridade para se activar (caso que continuará a sua execução)

$C.empty()$ retorna um valor Booleano que indica se a fila C está ou não vazia.