

1. Concurrent programming

José Proença

(slides mainly from Nelma Moreira)

Concurrent programming (CC3040) 2025/2026

CISTER – U.Porto, Porto, Portugal

<https://fm-dcc.github.io/cp2526>



CISTER - Research Centre in
Real-Time & Embedded
Computing Systems

Concurrent Programming - Part I

- 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

- 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.

- Reactive systems modelling, specification and verification. Luca Aceto, Anna Ingólfssdóttir, 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.:
 - CCS-CAOS
 - mCRL2.org
 - pseuco.Com
 - CAAL

Blocks of sequential code running concurrently and sharing memory:

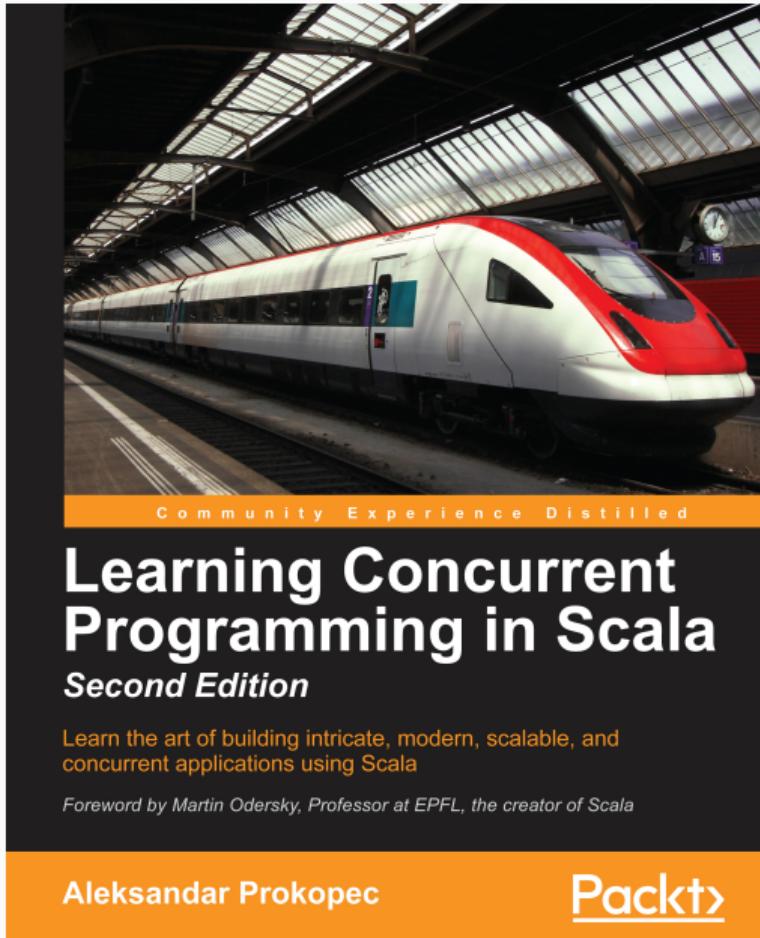
- What is **Scala** and why using it?
- Concurrency in Java and its memory model
- Basic concurrency blocks and libraries
- Futures and promises
- Actor model

We will be **less formal**

- focus on concepts and programs
- study operators and libraries
- tool support with **Scala**

We will have **hands-on**

- Practical programming exercises
- Apply the concepts we learn



Logistics

Relevant class material and announcements will be posted on the website periodically

<https://fm-dcc.github.io/cp2526>

Lecturer

- **José Proença**

<https://jose.proenca.org>

- jose.proenca@fc.up.pt

- tba

More links

- Sigarra: https://sigarra.up.pt/fcup/en/UCURR_GERAL.FICHA_UC_VIEW?pv_ocorrencia_id=570334

(Please send me an email the day before if you wish to meet)

Grading will consist of:

- **35% (T1)** – individual **test** for part 1 (≥ 6)
- **35% (T2)** – individual **test** for part 2 (≥ 6)
- **70% (FE)** – individual **final exam** for parts 1 and 2
- **30% (CW)** – **course work** for parts 1 and 2
 - groups of at most 2 students
 - **10%** for part 1
 - **20%** for part 2

Normal period

$$T1 \times 0.35 + T2 \times 0.35 + CW \times 0.3 \quad (\geq 9.5)$$

Mandatory 75% attendance in PL

Extra period (*recurso*)

$$FE \times 0.7 + CW \times 0.3 \quad (\geq 9.5)$$

Concurrent processes

- realizam uma função dos dados nos resultados (tese de Church/Turing)
- A sua semântica pode ser analisada considerando o estado (memória) em cada instante:

$$S[\![P]\!]: State \rightarrow State$$

onde p.e. $State = [Var \rightarrow \mathbb{Z}]$.

$P :$

$x \leftarrow 1$

$y \leftarrow 0$

while $x < 10$ **do**

$y \leftarrow y + x$

$x \leftarrow x + 1$

print y

- Terminam (se não terminarem a sua semântica é indefinida)

Dois programas são equivalentes se realizam a mesma função.

Equivalência de programas sequenciais

Dois programas são equivalentes se realizam a mesma função.

$P :$

$x \leftarrow 1$

$Q :$

$x \leftarrow 0$

$x \leftarrow x + 1$

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)
- ...

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)
- ...

A semântica de programas sequenciais é composicional. Por exemplo a semântica de $P; Q$ é obtida da semântica de P e de Q .

Programas Sequênciais vs Concorrentes

$P :$

$x \leftarrow 1$

$Q :$

$x \leftarrow 0$

$x \leftarrow x + 1$

- Mas se os executarmos em paralelo, $P||Q$?

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

$P :$

$x \leftarrow 1$

$Q :$

$x \leftarrow 0$

$x \leftarrow x + 1$

- Mas se os executarmos em paralelo, $P||Q$?
- Qual o significado? P e Q podem intercalar
 1. não é único:

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

$P :$

$x \leftarrow 1$

$Q :$

$x \leftarrow 0$

$x \leftarrow x + 1$

- 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)

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

$P :$

$x \leftarrow 1$

$Q :$

$x \leftarrow 0$

$x \leftarrow x + 1$

- 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 $||$.

Programas Sequênciais vs Concorrentes

$P :$

$x \leftarrow 1$

$Q :$

$x \leftarrow 0$

$x \leftarrow x + 1$

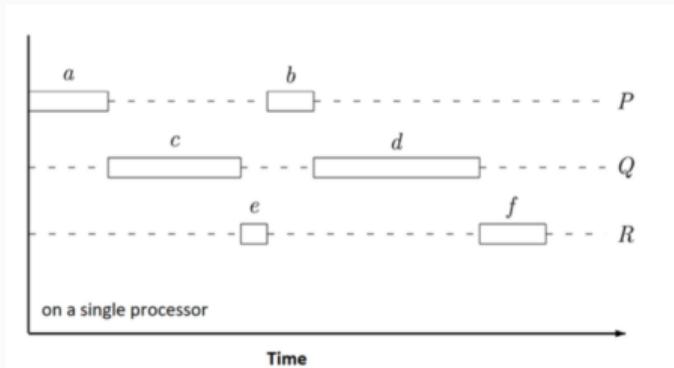
- Mas se os executarmos em paralelo, $P||Q$?
- Qual o significado? P e Q podem intercalar
 1. não é único:
 - pode ser 1: se for P e depois Q
 - ou 2: se for Q (primeira instrução) , depois P e depois Q (segunda instrução)
 2. a semântica não é determinística
 3. a equivalência não é preservada por $||$.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

- 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

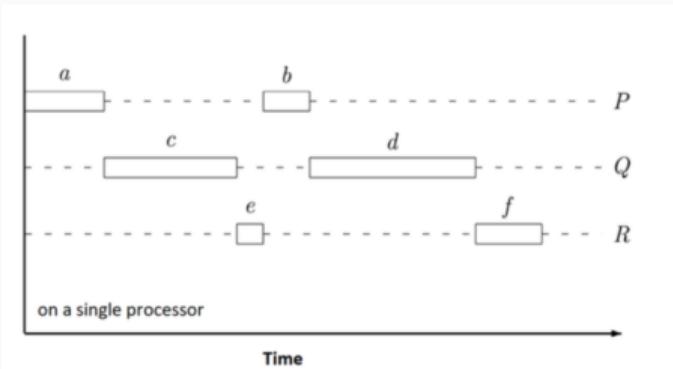
- Trabalho lógico simultâneo
- Não obriga a multiprocessador



Concorrência vs Paralelismo

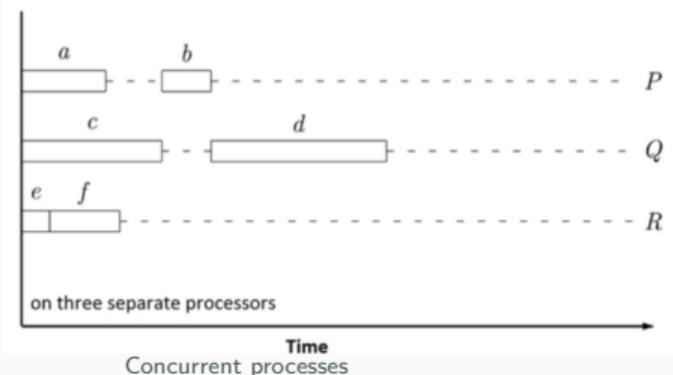
Concorrência

- Trabalho lógico simultaneo
- Não obriga a multiprocessador



Paralelismo

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



- Um **processo** é um programa (sequencial) em execução

- Um **processo** é um programa (sequencial) em execução
- É descrito por uma máquina de estados (estado= memória, contador de programa, etc)

- 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.

- 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

- 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

- 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**

Sincronização de Processos

- Quando o progresso de um ou mais processos depende do comportamento de outros processos

Sincronização de Processos

- Quando o progresso de um ou mais processos depende do comportamento de outros processos
- As interações podem ser de dois tipos:

Sincronização 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

Sincronização 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

Sincronização 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

- 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

- 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.

- 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*

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

```
procedure DISK-WRITE( $x, v$ )
   $D.\text{seek}(x);$ 
   $D.\text{write}(v);$ 
  return;
```

Competição: Ler e escrever num disco D

procedure DISK-READ(x)

$D.\text{seek}(x);$

$r \leftarrow D.\text{read}();$

return $r;$

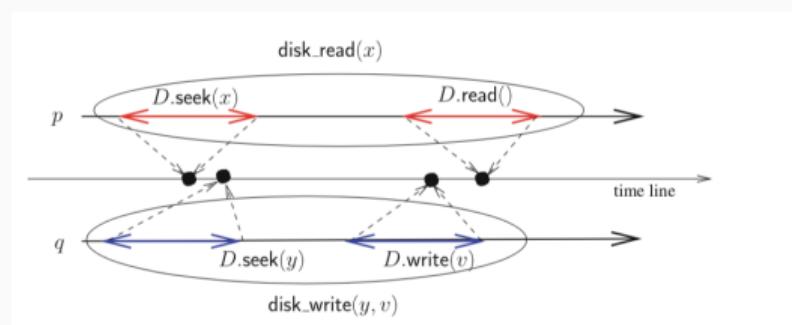
procedure DISK-WRITE(x, v)

$D.\text{seek}(x);$

$D.\text{write}(v);$

return;

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



Competição: Ler e escrever num disco D

procedure DISK-READ(x)

$D.\text{seek}(x);$

$r \leftarrow D.\text{read}();$

return $r;$

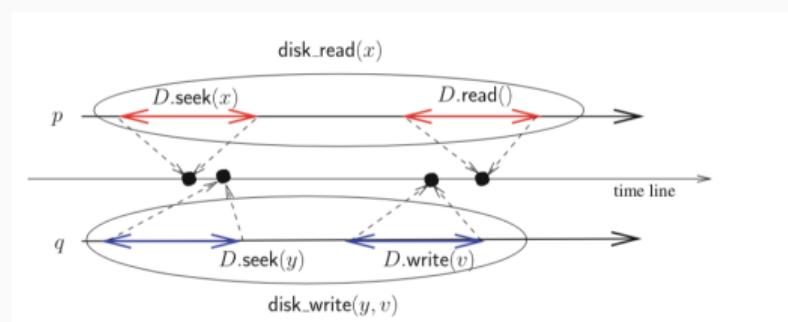
procedure DISK-WRITE(x, v)

$D.\text{seek}(x);$

$D.\text{write}(v);$

return;

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



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

Competição: Ler e escrever num disco D

procedure DISK-READ(x)

$D.\text{seek}(x);$

$r \leftarrow D.\text{read}();$

return $r;$

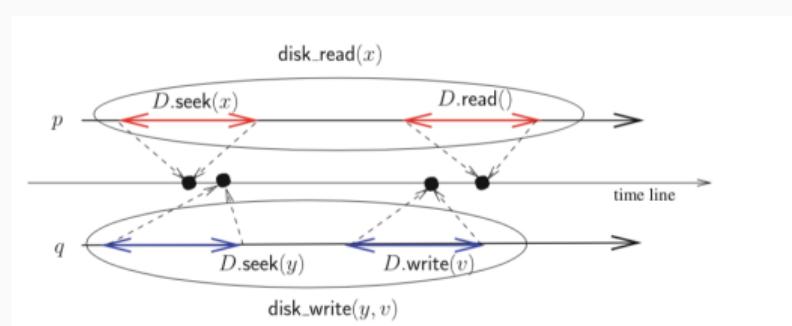
procedure DISK-WRITE(x, v)

$D.\text{seek}(x);$

$D.\text{write}(v);$

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 simultâneo → Exclusão

Cooperação: produtor/consumidor

- O produtor **produz** produtos

Cooperação:produtor/consumidor

- O produtor **produz** produtos
- O consumidor **consume** os produtos, e

Cooperação:produtor/consumidor

- O produtor **produz** produtos
- O consumidor **consume** os produtos, e
- Um produto não pode ser consumido **antes** de ser produzido

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**

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 $k \geq 1$

- 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

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**

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**

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:

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

- **Secção Crítica:** porção de código que só pode ser executado por um processo num dado instante

- **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.

- **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

- **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()*

- **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()*

- **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

- **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.

- **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.

```
procedure protected_code(in)
    acquire_mutex( );
    r ← cs_code(in);
    release_mutex( );
    return r;
```

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

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.

Bypass limitado → *Starvation-freedom*(=Bypass finito) →

Deadline Scheduling

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

Bypass limitado → *Starvation-freedom*(=Bypass finito) →

***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 → *Starvation-freedom*(=Bypass finito) →

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

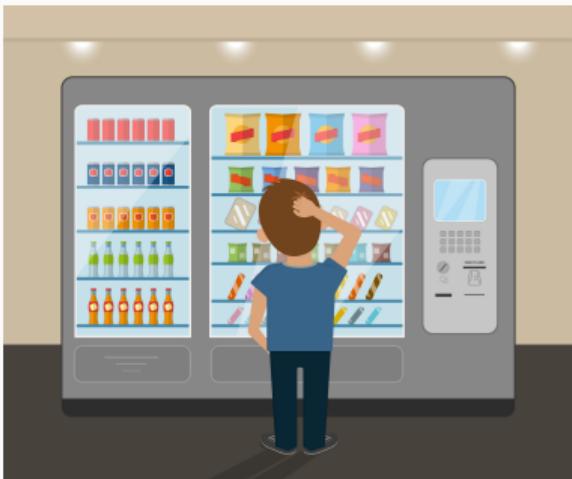
Modelação de Sistemas Concorrentes

Pretendemos modelar os sistemas concorrentes/reactivos de modo a

- poder assegurar as propriedades indicadas
- garantir a sua correção
- determinar se uma implementação está de acordo com a especificação
- em geral é difícil compreender o comportamento de um sistemas concorrentes

Acções de comunicação (observáveis)

- Pressionar um botão
- Inserir moedas
- Injectar uma bebida/comida





Acções de comunicação (observáveis)

- Pressionar um botão
- Inserir moedas
- Injectar uma bebida/comida

Acções internas

(não observáveis)

- Escolher a bebida
- Mecanismo para injectar o produto certo
- Tirar o dinheiro do bolso
- Verificar as moedas inseridas
- Pegar na bebida

- Um processo é descrito usando um conjunto de ações, **alfabeto do processo**

- Um processo é descrito usando um conjunto de ações, **alfabeto do processo**
- *Com*: conjunto de ações de comunicação (actividades observáveis) a, b, c, \dots

- Um processo é descrito usando um conjunto de ações, **alfabeto do processo**
- *Com*: conjunto de ações de comunicação (actividades observáveis) a, b, c, \dots
- *Int*: conjunto de accções internas (computações locais) $[a], [b], [c], \dots$

- Um processo é descrito usando um conjunto de ações, **alfabeto do processo**
- *Com*: conjunto de ações de comunicação (actividades observáveis) a, b, c, \dots
- *Int*: conjunto de accções internas (computações locais) $[a], [b], [c], \dots$
- $Act = Com \cup Int$ conjunto de todas as ações

- Um processo é descrito usando um conjunto de ações, **alfabeto do processo**
- *Com*: conjunto de ações de comunicação (actividades observáveis) a, b, c, \dots
- *Int*: conjunto de accções internas (computações locais) $[a], [b], [c], \dots$
- $Act = Com \cup Int$ conjunto de todas as ações
- A execução de uma **ação** muda o estado do processo:

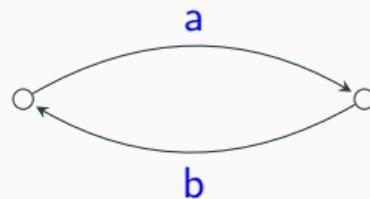
Modelo abstracto de um processo

- Um processo é descrito usando um conjunto de ações, **alfabeto do processo**
- *Com*: conjunto de ações de comunicação (actividades observáveis) a, b, c, \dots
- *Int*: conjunto de accções internas (computações locais) $[a], [b], [c], \dots$
- $Act = Com \cup Int$ conjunto de todas as ações
- A execução de uma **ação** muda o estado do processo:



Modelo abstracto de um processo

- Um processo é descrito usando um conjunto de ações, **alfabeto do processo**
- *Com*: conjunto de ações de comunicação (actividades observáveis) a, b, c, \dots
- *Int*: conjunto de acções internas (computações locais) $[a], [b], [c], \dots$
- $Act = Com \cup Int$ conjunto de todas as ações
- A execução de uma **ação** muda o estado do processo:



Usar só letras para simplificar