

# Aula 11

Expressividade do CCS e CCS com passagem de valores

## 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.

## Algoritmo de Peterson para a Exclusão Mútua

- $P_1, P_2$  processos
- variáveis partilhadas  $b_1, b_2$  e  $k$ , sendo
- se  $k = i$  então  $P_i$  pode entrar
- $P_1$  faz  $k = 2$  (dá o privilégio a  $P_2$ )
- e simetricamente para  $P_2$
- $b_i = true$  quando  $P_i$  espera
- $b_i = false$  quando  $P_i$  sai da zona crítica.

Processo  $P_i$

```
while true do
  noncritical actions
   $b_i \leftarrow true;$ 
   $k \leftarrow j;$ 
  while  $b_j \wedge k = j$  do
    skip;
  critical actions
   $b_i \leftarrow false;$ 
```

### Algoritmo de Peterson em CCS

O CCS não tem variáveis, mas estas são processos que comunicam a outros processos que necessitam de ler ou escrever os seus valores. Isto é

- As variáveis são *processos* cujos estados são os seus possíveis valores
- Para  $b_1$  temos estado  $B_{1t}$  se o seu valor for **true** e o estado  $B_{1f}$  se o seu valor for **false**.
- Outros processos podem ler ou escrever o valor de variáveis comunicando com o respectivo processo o valor pretendido
- Para um processo ler **true** em  $b_1$  sincroniza com esse processo por uma ação (canal)  $b1rt$
- Para um processo escrever **false** em  $b_1$  sincroniza com esse processo por uma ação (canal)  $b1wf$
- Análogamente se define o comportamento de  $k$  que pode tomar os valores 1 e 2.

### Processos para as variáveis

$$\begin{aligned} B_{1f} &:= b1rf!.B_{1f} + b1wf?.B_{1f} + b1wt?.B_{1t} \\ B_{1t} &:= b1rt!.B_{1t} + b1wf?.B_{1f} + b1wt?.B_{1t} \end{aligned}$$

$$\begin{aligned} B_{2f} &:= b2rf!.B_{2f} + b2wf?.B_{2f} + b2wt?.B_{2t} \\ B_{2t} &:= b2rt!.B_{2t} + b2wf?.B_{2f} + b2wt?.B_{2t} \end{aligned}$$

$$\begin{aligned} K_1 &:= kr1!.K_1 + kw1?.K_1 + kw2?.K_2 \\ K_2 &:= kr2!.K_2 + kw1?.K_1 + kw2?.K_2 \end{aligned}$$

### Processos para $P_1$ e $P_2$

- apenas modelar a entrada e saída da zona crítica
- fazer a inicialização das variáveis  $b_i$  e  $k$
- supomos que não podem terminar na zona crítica ou ficar lá para sempre

- para representar o ciclo **while** para  $P_1$  temos um estado  $P_{11}$  tal que
  - ler os valores de  $b_2$  e  $k$
  - esperar se  $b_2 = \mathbf{true} \wedge k = 2$
  - mudar de estado,  $P_{12}$
  - em  $P_{12}$  entrar e sair da zona critica
  - mas como avaliar  $b_j \wedge k = j?$
  - avaliamos da esquerda para a direita: se  $b_j$  é **true** então avaliamos se  $k = j$ ; a segunda não é avaliada se a primeira for falsa.

### Processos para $P_1$ e $P_2$ em CCS

$$\begin{aligned}
 P_1 &:= b1wt!.kw2!.P_{11} \\
 P_{11} &:= b2rf?.P_{12} + b2rt?.(kr2?.P_{11} + kr1?.P_{12}) \\
 P_{12} &:= enter1.exit1.b1wf!.P_1
 \end{aligned}$$

$$\begin{aligned}
 P_2 &:= b2wt!.kw1!.P_{21} \\
 P_{21} &:= b1rf?.P_{22} + b1rt?.(kr1?.P_{21} + kr2?.P_{22}) \\
 P_{22} &:= enter2.exit2.b2wf!.P_2
 \end{aligned}$$

### Algoritmo de Peterson em CCS

Se  $k = 1$  no inicio

$$Peterson := (P_1|P_2|B_{1f}|B_{2f}|K_1)\backslash L$$

onde  $L$  são todas as ações excepto as de entrada e saída.

Ficheiro no Pseuco.com: <https://pseuco.com/\#/edit/remote/tkxz8fpu1t8ke56bj9uo>

### Exclusão mútua em CCS

Se se modelar a entrada e a saída da zona crítica fica

$$MutexSpec := enter1.exit1.MutexSpec + enter2.exit2.MutexSpec$$

Será que  $MutexSpec \approx Peterson?$

Não. Verifica que depois de

$$Peterson \xrightarrow{\tau} (P_{12}|P_{21}|B_{1t}|B_{2t}|K_1)\backslash L$$

o estado atingido não tem transição por  $enter_2$ .

Contudo pode-se provar que são equivalentes por traços fracos.

### Teste da exclusão mútua

Contudo podemos mostrar que no algoritmo de Peterson um processo não entra na zona crítica usando o outro lá está. Para tal usamos um monitor, i.e., outro processo que executando em paralelo irá detectar se há um erro:

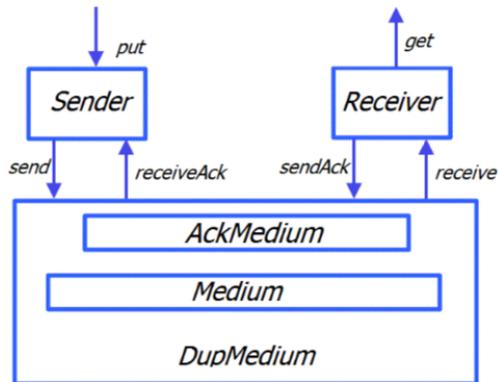
$$\begin{aligned} MutualTest &:= enter1!.MutualTest1 + enter2!.MutualTest2 \\ MutualTest1 &:= exit1!.MutualTest + enter2!.bad!.0 \\ MutualTest2 &:= exit2!.MutualTest + enter2!.bad!.0 \end{aligned}$$

Executando

$$(Peterson | MutualTest) \setminus \{enter1, enter2, exit1, exit2\}$$

ver se a ação *bad!* pode ser executada. Pode-se mostrar que não. Ficheiro no Pseuco.com: <https://pseuco.com/\#/edit/remote/45rwd3v9h5dhvneat3p>

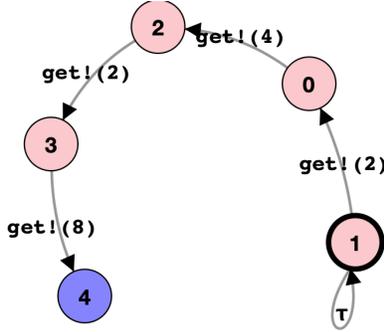
### Exemplo de um Protocolo com erro no meio –Pseuco



$$\begin{aligned}
Sender &:= put?.send!.Sending \\
Sending &:= receiveAck?.Sender + receiveNAck?.send!.Sending \\
Receiver &:= receive?.get?.sendAck!.Receiver + \\
&\quad gargled?.sendNAck!.Receiver \\
Medium &:= send?.(receive!.Medium + i.garbled!.Medium) \\
AckMedium &:= sendAck?.receiveAck!.AckMedium + \\
&\quad sendNAck?.receivedNAck!.AckMedium \\
DupMedium &:= Medium|AckMedium \\
Protocol &:= (Sender | Receiver | DupMedium)\ \\
&\quad \{send, receive, sendAck, receiveAck, \\
&\quad receiveNAck, sendNAck, garbled\}
\end{aligned}$$

### Envio de uma mensagem

$$\begin{aligned}
Sender &:= put?x.send!x.Sending[x] \\
Sending[x] &:= receiveAck?.Sender + receiveNAck?.send!x.Sending[x] \\
Receiver[x] &:= receive?x.get?x.sendAck!.Receiver[x] + \\
&\quad gargled?.sendNAck!.Receiver[x] \\
Medium &:= send?x.(receive!x.Medium + i.garbled!.Medium) \\
AckMedium &:= sendAck?.receiveAck!.AckMedium + \\
&\quad sendNAck?.receivedNAck!.AckMedium \\
DupMedium &:= Medium|AckMedium \\
Protocol &:= (Sender | Receiver | DupMedium)\ \\
&\quad \{send, receive, sendAck, receiveAck, \\
&\quad receiveNAck, sendNAck, garbled\} \\
\\
Protocol|put!2.put!4.put!2.put!8.0\{put\}
\end{aligned}$$



### $CCS_{vp}$ com passagem de valor

- $a!v$ : saída do valor  $v$  no canal  $a$  (enviar)
- $a?v$ : entrada do valor  $v$  pelo canal  $a$  (receber)
- ou usar variáveis
- $a!x$ : saída do valor guardado em  $x$  no canal  $a$  (enviar)
- $a?x$ : entrada de um valor que se guarda em  $x$  pelo canal  $a$  (receber)
- e os nomes dos processos podem ter variáveis como parâmetros permitindo assim enviar e receber valores ( $A[x, y]$ )

### $CCS_{vp}$ com passagem de valor

Sendo  $\mathbb{V}$  um conjunto de valores e  $\mathbb{K}$  um conjunto de canais temos

$$\begin{aligned}
 A^! &= \{a!v \mid a \in \mathbb{K}, v \in \mathbb{V}\} \cup \{a! \mid a \in \mathbb{K}\}, \\
 A^? &= \{a?v \mid a \in \mathbb{K}, v \in \mathbb{V}\} \cup \{a? \mid a \in \mathbb{K}\}, \\
 Com &= A^! \cup A^? \\
 Act &= Com \cup \{\tau\}
 \end{aligned}$$

$$\begin{aligned}
 P &::= 0 \mid X[r_1, \dots, r_n] \mid P + P \mid \chi.P \mid P|P \mid P \setminus H \\
 \chi &::= \tau \mid a! \mid a? \mid a!v \mid a?v \mid a!x \mid a?x
 \end{aligned}$$

onde  $X \in Var$ ,  $x \in D$ ,  $r_i \in D \cup \mathbb{V} \cup \mathbb{K}$

### Regras do $CCS_{vp}$

$$\text{Pref} \frac{\alpha \in Act}{\alpha.P \xrightarrow{\alpha} P}$$

$$\text{Input} \frac{v \in \mathbb{V}}{a?x.P \xrightarrow{a?v} P\{v/x\}}$$

onde  $P\{v/x\}$  é  $P$  onde  $x$  é substituído por  $v$  (e  $x$  não é uma ação)

$$\begin{aligned} (a!y.P)\{v/x\} &= a!y.P\{v/x\} \quad \text{se } y \neq x \\ (a!x.P)\{v/x\} &= a!v.P\{v/x\} \\ (a?y.P)\{v/x\} &= a?y.P\{v/x\} \quad \text{se } y \neq x \\ (a?x.P)\{v/x\} &= a?x.P \\ X[x]\{v/x\} &= X[v] \\ X[y]\{v/x\} &= X[y] \quad \text{se } y \neq x \end{aligned}$$

Para as restantes expressões é passado para as subexpressões.

**Exemplo 11.1.**  $put?x : 0..9.send!x.0$

$$\text{Rec} \frac{P\{v_1/r_1, \dots, v_n/r_n\} \xrightarrow{\alpha} P' \quad \Gamma(X[r_1, \dots, r_n]) = P}{X[r_1, \dots, r_n] \xrightarrow{\alpha} P'}$$

**Exemplo 11.2.**

$$\begin{aligned} send!y.Sending[x]\{3/x, 5/y\} &= send!5.Sending[3] \\ send!y.Sending[x]\{3/x, 5/y, receive/send\} &= receive!5.Sending[3] \end{aligned}$$

**when: Bloqueador Condicional**

Supomos  $\mathbb{V} = \mathbb{Z}$  ( $CCS_{vp}^Z$ )

**Exemplo 11.3.**

$$B[x] := when(x < 4)put?.B[x + 1] + when(x > 0)get?.B[x - 1]$$

$B[0]$  ou  $B[5]$  o que fazem?

$when(b)P$  se  $b$  é verdade comporta-se como  $P$  senão bloqueia.

Vamos só considerar expressões com inteiros. A avaliação de expressões  $e \Downarrow z$ : a expressão  $e$  avalia para  $z$ .

**Exemplo 11.4.**

$$\begin{aligned} send!(x + y)3/x, 5/y &= sent!(8) \\ 3 + 5 &\Downarrow 8 \end{aligned}$$

Acrescentamos à gramática

$$P ::= 0 \mid X[r_1, \dots, r_n] \mid P + P \mid \chi.P \mid P|P \mid P \setminus H \mid when(b)P$$

**Exemplo 11.5.**

$$\begin{aligned} IterMult[z, x, y] &:= when(x > 0)i.IterMult[z + y, x - 1, y] \\ &\quad + when(x == 0)println!z.0 \\ IterMult[0, 3, 7] &\end{aligned}$$

**Regras  $CCS_{vp}^Z$**

$$\begin{aligned} \text{Pref} & \frac{\alpha \in Act}{\alpha.P \xrightarrow{\alpha} P} \\ \text{Input} & \frac{v \in \mathbb{V}}{a?x.P \xrightarrow{a?v} P\{v/x\}} \\ \text{Rec} & \frac{P\{v_1/r_1, \dots, v_n/r_n\} \xrightarrow{\alpha} P' \quad \Gamma(X[r_1, \dots, r_n]) = P}{X[r_1, \dots, r_n] \xrightarrow{\alpha} P'} \\ \text{Output} & \frac{e \Downarrow z}{a!e.P \xrightarrow{a!z} P} \\ \text{Valor} & \frac{e \Downarrow z}{a?e.P \xrightarrow{a?z} P} \\ \text{cond} & \frac{P \xrightarrow{\alpha} P' \quad b \Downarrow True}{when(b)P \xrightarrow{\alpha} P'} \end{aligned}$$

**Exemplo 11.6.** *O exemplo do factorial*

$$\begin{aligned} Fac[n, j] & := when(j > 0)i.Fac[n * j, j - 1] \\ & + when(j == 0)println!n.0 \end{aligned}$$

*Calcular  $Fac[1, 5]$*

**Exemplo 11.7.** *Podemos definir um processo que corresponde a uma célula de memória que pode ser partilhada entre processos. Permite obter o valor guardado e guardar um valor*

$$Cell_x[cur] := get_x!cur.Cell_x[cur] + set_x?new : 0..2.Cell_x[new]$$

*Mais geralmente:*

$$Cell[rd, wr, x] := rd!x.Cell[rd, wr, x] + wr?y.Cell[rd, wr, y]$$

- $Cell[rd, wr, 5]$
- $Cell[rdA, wrA, 0] | Cell[rdB, wrB, 0]$

$$\begin{aligned}
Cells &:= Cell[rdA, wrA, 0] | Cell[rdB, wrB, 0] \\
Serve &:= mult?.rdA?x : R.rdB?y : R.IterMult[0, x, y] \\
IterMult[z, x, y] &:= when(x > 0)i.IterMult[z + y, x - 1, y] \\
&\quad +when(x == 0)println!z.Serve \\
Use &:= wrA!7.wrB!5.mult!.0
\end{aligned}$$

$(Cells|Serve|Use)\{rdA, wrA, rdB, wrB, mult\}$

Qual o resultado?

Ficheiro no Pseuco.com:

**Exemplo 11.8.** *O exemplo do factorial em versao iterativa*

$$\begin{aligned}
Fak &:= rdJ?j : R.(when(j > 0)rdN?n.wrN!(n * j).wrJ!(j - 1).Fak \\
&\quad +when(j == 0)rdN?n.println!n.0) \\
Cell[v, rd, wr] &:= rd!v.Cell[v, rd, wr] + wr?x : R.Cell[x, rd, wr] \\
Cells &:= Cell[0, rdN, wrN] | Cell[0, rdJ, wrJ]
\end{aligned}$$

$(wrN!1.wrJ!5.Fak|Cells)\{rdN, wrN, rdJ, wrJ\}$

Ficheiro no Pseuco.com:

<https://pseuco.com/#/edit/remote/5bglrm4937vze10dulqi>

**O  $CCS_{vp}$  pode ser embebido no CCS**

..logo é só "syntatic sugar"...

$CCS_{vp}$	$CCS$
$a!v.P$	$a_v!.P$
$a?x.P$	$\sum_{v \in \mathbb{V}} a_v?.P\{v/x\}$
$X[u_1, \dots, u_n]$	$X_{u_1, \dots, u_n}$

Isto é basta usar ações e nomes indexados, podendo ser considerados conjuntos infinitos de índices ( $\mathbb{V}$  ou  $D$ )