Aula 8

PseuCo

PseuCo: Linguagem de programação concorrente

- Semântica formal traduz-se para CCS e é aí interpretada
- Tem uma semântica executável que é implementada num compilador para Java
- Processos são agentes havendo sempre o `mainAgent`

What's new?

```
mainAgent { int n, j;
n = 1;
for (j = 5; j > 0 ; j--){
n = n*j;
}
println (" O factorial de 5 e " + n + ".");
}
```

$$
\begin{array}{rcl}
MainAgent[a] & := & MainAgent_2[a, 5, 1] \\
MainAgent_1[a, \$j, \$n] & := & MainAgent_2[a, \$j - 1, \$n] \\
MainAgent_2[a, \$j, \$n] & := & when(!(\$j > 0))MainAgent_3[a, \$j, \$n] \\
 & & +when(\$j > 0)i.MainAgent_1[a, \$j, \$n * \$j] \\
MainAgent_3[a, \$j, \$n] & := & println!(\text{"}O\ factorial\ de\ 5\ e\ \text{"}\,{}^{\$}n\,\text{"}.\text{"}).0
\end{array}
$$

$MainAgent[1] \backslash *, println, exception$

Tipos de comunicação entre Agentes

- Por variáveis partilhadas: memória partilhada
- Síncrona via mensagens: canais de capacidade 0
- Assíncrona via mensagens: canais de capacidade > 0 (buffers)

Crash course PseuCO

- variáveis podem ser declaradas: locais ou globais
- instruções terminam com ;
- procedimentos tem um valor de retorno ou são de tipo `void`
- condicionais: **if**; operadores $!, ==, \&\&, ||$
- condicionais inline : $n > 5?"mais";"menos"$
- ciclos: **for** e **while**
- estruturas: **struct**
- Em procedimentos
- call-by-value: tipos simples
- call-by-reference: arrays, struct, monitor, mutex e canais

Agentes

- `agent a1=start(<instrucao>)`
- `start(<instrucao);`
- Esperar pela terminação: `join(a1)`

Canais

- `boolchan chan1;` : sincrono
- `intchan7 chan2` : asíncrono
- canais são FIFO: first-in-first-out
- `chan2 <! 7` : envia 7
- `<?chan2` : recebe 7
- `int x=<? chan2`: x fica com 7
- se vazios não enviam
- se se tentar receber de um canal vazio, quem *recebe fica bloqueado*
- se se tentar enviar para um canal cheio, quem *envia fica bloqueado*

Sincronização

```
void factorial(int z, intchan c) {
    int j, n=1;
    for (j = z; j > 0 ; j--) {
        n= n*j;  }
    c <! n; }
mainAgent {
    intchan cc;
    int mid, fin;
    agent a1 = start(factorial(3, cc));
    println("Agent 1 is working for me.");
    mid = <? cc;
    agent a2 = start(factorial(mid, cc));
    println("Agent 2 is working for me.");
    fin = <? cc;
    println("The factorial of the factorial of
      three is " + fin + ".");}
```

Tabela de factoriais

```
void factorial(intchan in,intchan out) {
    int j,n,z;
    while(true){
        z=<?in;
        n=1;
        for(j=z;j>0;j--){
            n=n*j;
        }
        out <! n;
        }
}
mainAgent{
intchan10 chn1,chn2;
int j;
for(j=1;j<=4;j++){
    start(factorial(chn1,chn2));
}
for(j=1;j<=8;j++){
    chn1<! j;
}
for(j=1;j<=8;j++){
    int res=<? chn2;
    println("o factorial de "+j+ " é "+ res);
}
```

```
}
```

- Como se pode garantir a ordem
- Como se terminam os agentes

`Select-case`

- Vários `case` e um `default`
- não deterministicamente escolhe um que não esteja bloqueado
- `default` nunca está bloquado

```
select {
case chan1 <! a: {
// varias instrucoes}
case b = <? chan2:
// so uma
case <? chan3:
// recebe e esquece
default:
//sempre disponivel
}
```

Tabela de factoriais

- Terminação
- Ordem (outra solução)

Memória partilhada

- agentes podem partilhar variáveis
- se simples têm de ser globais
- as estruturas podem ser partilhadas por argumentos devido ao call-by-reference
- *Race-condition/Data-race*: quando um agente pode escrever numa variável que outro está a ler
- *Data-race*: programas são incorrectos pelo que se tem evitar isso
- Usar **lock** e **monitor** para garantir a exclusão mútua.

Exemplo

```
int n;
void counter(){
int loop;
for (loop = 0; loop < 5; loop++){
n = n - 1; }
}
mainAgent { n = 10;
agent a1 = start(counter());
agent a2 = start(counter());
join(a1);
join(a2);
println ("The value is "+ n);
}
```

Verificar o data race no pseuCo.com. e examinar caminhos.

lock

- Permitem coordenar o acesso a variáveis evitando o acesso concorrente e assim o *data race*.
- `lock` : bloqueia
- `unlock`: liberta
- Em CCS: $Lock := lock.unlock.Lock$ (qual o LTS?)
- ```
  lock m;
  int i=5;
  void dec() {
  lock(m);
  i--;
  unlock(m);
  }
  ```
- dizemos que i está *guardado* pelo lock/mutex m.
- a instrução `i--` é a *zona crítica*

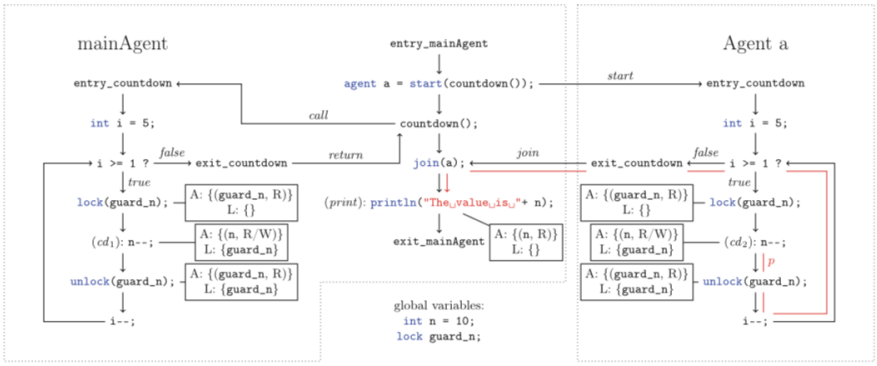
Livre de data race

```
int n=10;
lock guardn;
void countdown() {
for (int i=5; i>=1;i--) {
```
```

```
  lock(guardn);
  n--;
  unlock(guardn);
  }
}
mainAgent {
    agent a=start(countdown());
    countdown();
    join(a);
    println("the value is +n);
}
```

Como detectar Data races?

- Calcular o grafo de programa usando a informação do fluxo de controlo
- Anotar o grafo nos locais em que há acesso a variávies globais (A) e as regiões em que há locks (L) e o tipo de acesso (R ou W).
- Efetuar uma pesquisa no grafo e identificar pares de acesso à memória
- Detectar se algum desses pares é um data race.



Análise do grafo de programa

As seguintes características distinguem pares de acessos à memória inofencivos ou potencialmente perigosos:

1. ambos são no mesmo agente?
2. são ambos de leitura?
3. ambos partilham um mesmo lock?

4. usando o grafo pode-se construir um caminho de causalidade entre os dois acessos?

Se alguma das respostas é *sim* então não é um candidato a *data race* No exemplo, para n temos os acessos cd_1, cd_2 e $print$.

- (cd_1, cd_2) ambos estão protegidos pelo mesmo lock (**??**)
- $(cd_1, print)$ no mesmo agente (**??**)
- $(cd_2, print)$ tem um caminho de causalidade p (**??**).

Notar que para uma mesma variável global

- não se pode ter um agente com o lock e outro sem lock
- não se podem usar locks diferentes para cada agente
- Se não usarmos **unlock** fica em *deadlock*
- Será necessário lock na criação e na escrita?
- Se usarmos uma variável local para decrementar n. Funciona?

Exercício 8.1. *Considerar todos estes casos para o exemplo do contador.* ◇

Mutex

- Um mutex só funciona se todos os agente envolvidos o usarem de forma cooperativa e para o mesmo fim
- Um recurso partilhado é livre se data races se existir um mutex m que independentemente de quando e domo o recurso R é acedido esse acesso só ocorre em fragmentos de código que estão protegidos por m.
- no limite os programas podem ser sequênciais ou de paralelismo puro (intercalagem).

Sequencial

```
int n=10;
lock guardn;

void countdown() {
lock(guardn);
    for (int i=5; i>=1;i--) {
        n--;}
```

```
    unlock(guardn);
}
mainAgent {
    agent a=start(countdown());
    countdown();
    join(a);
    println("the value is "+n);
}
```

Deadlock

Também devemos evitar *Deadlock*

```
lock guardr; lock guards;
int r; int s;
void swap(){
    lock(guards);
    lock(guardr);
    int x=r;
    r=s; s=x;
    lock(guards);   lock(guardr);}
mainAgent{
    r=13;s=17;
    agent a=start(swap());
    lock(guards);lock(guardr);
    println("troca de r e s"+r+" "+s);
    unlock(guards);unlock(guardr);
}
```

`struct`

Admite a definição de métodos

```
MyInt z;
struct MyInt{
    int n;
    void set(int x){n=x;}
    int get(){return n;}
    void dec(){n--;}}
MyInt z;
void decr(){
    z.dec();}
void setpr(){
    z.set(3);
    println("o valor é ",z.get());}
mainAgent {
```

```
    agent a1=start(decr());
    agent a2 = start(setpr());
}
```

Pode ter data races...

`struct` com locks

```
MyInt z;
struct MyInt{
lock g;
    int n;
    void set(int x){lock(g);n=x;unlock(g);}
    int get(){int tmp; lock(g);tmp=n; unlock(g);return tmp;}
    void dec(){lock(g);n--;unlock(g);}}
MyInt z;
void decr(){
    z.dec();}
void setpr(){
    z.set(3);
    println("o valor é ",z.get());}
mainAgent {
    agent a1=start(decr());
    agent a2 = start(setpr());
}
```

Um monitor faz isto automáticamente.

monitor

- É um struct especial
- tem lock implícito: **lock** é usado antes de qualquer acesso a um método do monitor e **unlock** quando o método retorna.

```
monitor AtomicInt{
    int n;
    void set(int x){
        n=x; }
    int get(){
        return n;}
    void dec(){
        n--;}
}
```

Não tem Data races mas não chega: é não determinístico o valor calculado.

monitor

- Os monitores podem esperar por certas condições (**condition**)
- `waitForCondition` : só procede quando a condição é satisfeita
- `signal`: acorda um agente e indica que a condição se verifica
- `signalAll`: acorda todos os agentes
- quem re-adquire o lock só procede se a condição é satisfeita

monitor

```
monitor Count {
int i;
condition notNull with (i > 0);
void inc() {
i++;
 // alguem pode usar dec()
 signal(notNull);
}

void dec() {
waitForCondition(notNull);
 i--;
}
}
```

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 $S.down()$ pode ser realizada se $S \geq 0$ for mantido.
- Se a operação não puder ser realizada ($S < 0$) o processo fica bloqueado.
- Quando o semáforo é incrementado, um dos processos que está à espera é desbloqueado.
- Invariante: sendo $\#S.down$ ($\#S.up$) o número de invocações,

$$S = s_0 + \#S.up - \#S.down$$

Semáforo

Construção dum semáforo com um monitor

```
monitor Semaphore { int value;
condition valueNonZero with (!( value ==0));
void init(int v) { value = v;
}
void up() { ++value;
signalAll valueNonZero; }
void down () {
waitForCondition valueNonZero --value;
} }
```

Propriedades dum Semáforo

- Quando um processo usa $S.down()$ não sabe se vai ou não ficar bloqueado
- Quando um processo usa $S.up()$, não sabe se vai ou não desbloquear algum processo
- Quando um processo usa $S.up()$,não se sabe se será ele ou o processo que ficou desbloqueado que irá proseguir imediatamente
- O valor do semáforo se positivo indica o número de processos que podem decrementar sem bloquar
- O valor do semáforo se negativo o número de processos bloqueados
- O valor do semáforo se zero indica que não há processos bloqueados mas se algum decrementar será bloqueado.

Exercício 8.2. *Com semáforos implementar os seguintes problemas:*

- *(*Rendezvous*) Supor dois processos A e B cada um com duas instruções* a_1, a_2 *e* b_1, b_2, *respectivamente. Pretende-se garantir que* a_1 *ocorra antes de* b_2 *e que* b_1 *ocorra antes de* a_2
- *(*Mutex*) Dados dois processos A e B garantir ambos não acedem simultaneamente a uma zona crítica (p.e ambos têm uma instrução* $n++$*, para um n variável inteira global).*
- *(*Multiplex*) Generalizar o problema anterior para o caso de no máximo max processos podem aceder a uma zona de código.*
- *(*Barreiras*) Generalizar o* Rendezvous *para n processos. Nenhum processo pode chegar à zona crítica antes de efectuar o* rendezvous*. Sugestão: usar um mutex/lock para incrementar um contador que indique quantos processos chegaram ao* rendezvous*.*

◇