Otimização para ambientes multiprocessados:

```
while (in) skip;   // este não invalida o cache.
while (TS(in)) {
    while (in) skip;
}
```

* Como implementar $\langle S; \rangle$?

protocolo de entrada

$S;$ ← e S pode ser tão grande quanto eu queira.

protocolo de saída.

* Como implementar $\langle await(B)\ S; \rangle$?

```
while (!B)   // x vale no máximo 1 vez.
    protocolo de entrada
    while (!B) { protocolo de saída; skip; protocolo de entrada; }
    S;
    protocolo de saída.
```

Outras primitivas de hardware:

- fetch-and-add
- exchange(a,b)
- compare-and-swap.

```
fetch-and-add (int i)
< int val = i;
  i++;
  return val; >
```

while (in != 0);

protocolo de entrada: while (FA (in) != 0)
while (in!=0)
skip;
[in=1] dentro da seção crítica, pra evitar que e algumas vezes o int estore.

protocolo de saída: in = 0;

27/03

CS enter
while (!B) { CS exit ; Delay; CS enter }
Seção-crítica
CS exit.

equivalente a <await (B) < seção-crítica >

Soluções justas para o problema da seção-crítica.

- spin lock: testa a condição, espera, testa de novo, espera mais um pouco, até que eu mesmo altero a variável

tem { exclusão mútua
livre de deadlock
sem atraso desnecessário.

mas: entrada garantida? não, só com justiça forte

- Tie-breaker (algoritmo de Peterson)

idéia: os processos devem se revezar quando os dois querem entrar na seção crítica.
Usa uma variável adicional para marcar o último a entrar.

Voltando ao algoritmo mais simples:

	①	②
p. entrada:	while(in2) skip	while(in1) skip;
	in1 = true;	in2 = true.
p. saída:	in1 = false	in2 = false.

Como o protocolo de entrada não é atômico, não há exclusão mútua.

1ª idéia: trocar ~~as~~ ordem → pôr o pé na porta

	①	②
p entrada:	in1 = true;	in2 = true;
	while(in2) skip;	while(in1) skip.

Assim, não existe risco nenhum de entrar 2 processos ao mesmo tempo na seção crítica. Mas pode dar deadlock → possível solução: se os dois estiverem com true, escolhe um pra entrar

Usar variável auxiliar para o caso onde in1 e in2 são verdadeiros

last: variável que indica o último que quer entrar.

bodean in1 = false ; in2 = false int last = 1;

```
process CS1 {                          process CS2 {
  while (true) {                         while (true) {
    in1 = true;                            in2 = true;
    last = 1;                              last = 2;
    < await (!in2 or last==2); >           < await (!in1 or last==1); >
    Seção crítica.                         Seção crítica.
    in1 = false;                           in2 = false;
    seção não-crítica                      seção não-crítica
  }                                      }
}                                      }
```

se trocar, pode dar errado a exclusão mútua.

não resolve deadlock

while(in1 and last==2) skip

pode!

é o tie-breaker

A generalização:

```
int in[1:n] = ([n] 0), last [1:n] = ([n] 0);

process CS[i=1 to n] {
    while (true) {
        for [j=1 to n] {                    // protocolo de entrada
            last[j] = i;        in[i] = j;
            for [k=1 to n such that i!=k] {
                while (in[k] >= in[i] and last[j] == i) skip;
            }
        }
        Seção crítica
        in[i] = 0;              // protocolo de saída
        seção não-crítica;
    }
}
```

- Algoritmo de Dekker.

variável compartilhada turn. que começa com 1.

1ª tentativa.

	①	②
p. entrada:	while(turn != 1) skip.	while(turn != 2) skip.
p. saída:	turn = 2;	turn = 1;

tem atraso desnecessário, lembra?

2ª tentativa.

~~inicio~~ want1 = false , want2 = false;

p. entrada:	while(want 2) skip.	while (want 1) skp.
	want 1 = true;	want 2 = true.
p. saída.	want 1 = false;	want 2 = false.

Falha exclusão mútua, se os 2 testarem ao mesmo tempo.

3ª tentativa.

p. entrada:	①	②
	want 1 = true;	want 2 = true
	while (want 2) skip;	while (want 1) skp;

Dá deadlock.

Ideia: desistir se houver conflito.

p. entrada:

①	②
want 1 = true;	want 2 = true;
while(want 2) {	while (want 1) {
want 1 = false	want 2 = false;
skip;	skip;
want 1 = true;	want 2 = true;
}	}

Falha no atraso desnecessário.

Algoritmo de Dekker

+ variável inteira turn = 1;

① (p. entrada)

```
want1 = true;
while (want2) {
    if (turn == 2) {
        want1 = false;
        while (turn != 1) skip;
        want1 = true;
    }
}
```

② (simétrica)

p. saída: turn = 2; want1 = false.

Funciona!

Algoritmo Manna - Pnueli

want1 = 0, want2 = 0,

p. entrada:

①

```
< if (want2 == -1)
    want1 = -1;
  else want1 = 1; >
while (want1 == want2)
    skip;
```

②

```
< if (want1 == -1)
    want2 = 1;
  else want2 = -1; >
while (want1 == -want2)
    skip;
```

p. saída: want1 = 0; want2 = 0;

Se tirar a atomicidade do if, não funciona.