Simulação manual das soluções de Dijkstra e Peterson

Sistemas Concorrentes - Lista 1 - Exercício 1

Sistemas Concorrentes - PEL - 2018.2
Anny Caroline Correa Chagas

Enunciado: Simular as soluções apresentadas por Dijkstra e Peterson (dos slides) explorando vários cenários de execução (cenário(s) cuja sequência "dá certo" e cenário(s) cuja sequência leva ao problema – quando for o caso). A simulação deve ser feita manualmente (como apresentada em sala de aula):

1 Dijkstra

Nesta seção serão discutidas as soluções evolutivas apresentadas por Dijkstra. Todas consistem em algoritmos de programação concorrente para exclusão mútua que visam permitir que dois processos ou threads acessem uma região crítica sem conflitos. 4 soluções apresentam problemas de concorrência, como deadlocks, livelocks e até mesmo a quebra da exclusão mútua. A última apresentada nesta seção, denominada algoritmo de Dekker, é tida como uma solução correta já que garante a exclusão mútua e não apresenta problemas de concorrência.

1.1 Uma variável, buzy waiting

Garante a exclusão mútua no acesso à região crítica (RC) utilizando uma variável global compartilhada chamada `turn`, a qual indica qual entre dois processos pode ter acesso à região crítica. Após acessar a região crítica, um processo deve passar o acesso ao outro alterando o valor de `turn`.

Dessa forma, o algoritmo exige a alternância explícita dos processos, o que pode levar ao desperdício de recursos: um processo pode estar impedido de acessar a regão crítica mesmo estando escalonado para execução e a RC não estar sendo acessada pelo outro processo. Esse desperdício é ilustrado no cenário da Tabela 1, onde o processo 0 permanece em espera ocupada entre os tempos t8 e t12, verificando continuamente o estado da variável `turn` e desperdiçando recursos de processamento.

```
while(1){
    while(turn != 0){
        nothing();
    }
    //região crítica
    turn = 1;
}
```

Processo 0

```
while(1){
    while(turn != 1){
        nothing();
    }
    //região crítica
    turn = 0;
}
```

Processo 1

Pode-se concluir, então, que o algoritmo é adequado quando a aplicação exige a alternância entre os processos. Caso contrário, há um grande desperdício de tempo e processamento.

Tabela 1. Primeira solução apresentada por Djikstra - uma variável, buzy-waiting (espera ocupada)

t	P0	P1	turn	Obs
t0				inicialização
t1			0	-
t2	×		0	inicia P0
t3	×		0	`while(1)`
t4	×		0	teste while falha (`while turn <> 0`)
t5	Ⓡ		0	P0 na RC
t6	×		1	`turn = 1`
t7	×		1	`while(1)`
t8	×		1	teste while ok (`while turn <> 0`)
t9	×		1	`{nothing}`
t10	×		1	teste while ok (`while turn <> 0`)
t11	×		1	`{nothing}`
t12	×		1	teste while ok (`while turn <> 0`)
t13	×		1	`{nothing}`
t14		×	1	inicia P1
t15		×	1	`while(1)`
t16		×	1	teste while falha `while turn <> 1`
t17		Ⓡ	1	P1 na RC
t18	×		1	teste while OK (`while turn <> 0`)
t19	×		1	`{nothing}`
t20		×	0	`turn = 0`
t21	×		0	teste while falha (`while turn <> 0`)
t22	Ⓡ		0	P0 na RC
t23	×		1	`turn = 1`

1.2 Duas variáveis

A segunda solução tenta corrigir o principal problema da primeira: o desperdício de tempo e processamento devido a alternância explícita dos processos. Em comparação à Tabela 1, na Tabela 2 o processo P0 pode acessar a região crítica quantas vezes forem necessárias, desde que esteja escalonado para execução e o processo P1 não esteja acessando-a. Para isso, o algoritmo utiliza duas variáveis compartilhadas, `flag[0]` e `flag[1]`, relativas a cada processo.

A espera ocupada continua desperdiçando recursos. Ao escalonar um processo para execução sem que esse tenha acesso à RC, todo seu tempo de execução será inútil[1] (t17 a t20 da Tabela 2), uma vez que para sair da espera ocupada é necessário que o outro processo libere a RC.

```
1 while(1){
2     while (flag[1]){
3         nothing();
4     }
5     flag[0] = true;
6     //região crítica
7     flag[0] = false;
8 }
```

Processo 0

```
1 while(1){
2     while (flag[0]){
3         nothing();
4     }
5     flag[1] = true;
6     //região crítica
7     flag[1] = false;
8 }
```

Processo 1

Tabela 2. Segunda solução apresentada por Djikstra - duas variáveis

t	P0	P1	flag[0]	flag[1]	Observação
t0			false	false	inicialização
t1	×		false	false	incia P0
t2	×		false	false	`while(1)`
t3	×		false	false	teste while falha (`while flag[1]`)
t4	×		true	false	`flag[0] = true`
t5	Ⓡ		true	false	P0 na RC
t6	×		false	false	`flag[0] = false`
t7	×		false	false	`while(1)`
t8	×		false	false	teste while falha (`while flag[1]`)
t9	×		true	false	`flag[0] = true`
t10	Ⓡ		true	false	P0 na RC
t11	×		false	false	`flag[0] = false`
t12	×		false	false	`while(1)`
t13	×		false	false	teste while falha (`while flag[1]`)
t14	×		true	false	`flag[0] = true`
t15		×	true	false	incia P1
t16		×	true	false	`while(1)`
t17		×	true	false	teste while ok (`while flag[0]`)
t18		×	true	false	`do {nothing}`
t19		×	true	false	teste while ok (`while flag[0]`)
t20		×	true	false	`do {nothing}`
t21	Ⓡ		true	false	P0 na RC
t22	×		false	false	`flag[0] = false`
t23		×	false	false	while falha (while `while flag[0]`)
t24		×	false	true	`flag[1] = true`
t25		Ⓡ	false	true	P1 na RC
t26		×	false	false	`flag[0] = true`

[1] Aqui, refiro-me ao tempo em que um processo executa antes de ser interrompido para que outro processo seja escalonado

Entretanto, a principal falha desse algoritmo é não garantir a exclusão mútia no acesso à região crítica. A Tabela 3 apresenta um cenário em que o escalonamento proposto permite, no tempo t10, que ambos os processos acessem a RC.

Tabela 3. Segunda solução apresentada por Djikstra - Quebra da exclusão mútua

t	P0	P1	flag[0]	flag[1]	Observação
t0			false	false	inicialização
t1		×	false	false	incia P1
t2		×	false	false	`while(1)`
t3		×	false	false	while falha (`while flag[1]`)
t4	×		false	false	incia P0
t5	×		false	false	`while(1)`
t6	×		false	false	while falha (`while flag[0]`)
t7	×		true	false	`flag[0] = true`
t8		×	true	true	`flag[1] = true`
t9		Ⓡ	true	true	P1 na RC
t10	Ⓡ	Ⓡ	true	true	P0 na RC

1.3 Pequena mudança no algoritmo da solução 2

A terceira solução garante a exclusão mútua utilizando as mesmas variáveis da solução 2, modificando ligeiramente o algoritmo. Um cenário em que a exclusão mútua é garantida é apresentado na Tabela 4. O escalonamento deste cenário foi baseado no escalonamento usado na Tabela 3 no objetivo de mostrar como esse terceiro algoritmo soluciona o problema do segundo.

```
1 while(1){
2     flag[0] = true;
3     while (flag[1]){
4         nothing();
5     }
6     //região crítica
7     flag[0] = false;
8 }
```

Processo 0

```
1 while(1){
2     flag[1] = true;
3     while (flag[0]){
4         nothing();
5     }
6     //região crítica
7     flag[1] = false;
8 }
```

Processo 1

Tabela 4. Terceira solução apresentada por Djikstra - Pequena mudança no algoritmo da solução 2

t	P0	P1	flag[0]	flag[1]	Observação
t0			false	false	inicialização
t1		×	false	false	inicia P1
t2		×	false	false	`while(1)`
t3		×	false	true	`flag[1] = true`
t4		×	false	true	while falha (`while flag[0]`)
t5	×		false	true	inicia P0
t6	×		false	true	`while(1)`
t7	×		true	true	`flag[0] = true`
t8	×		true	true	while ok (`while flag[1]`)
t9	×		true	true	`do{nothing}`
t10		Ⓡ	true	true	P1 na RC
t11		×	true	false	`flag[1] = false`
t12	×		true	false	while falha (`while flag[1]`)
t13	Ⓡ		true	false	P0 na RC
t14	×		true	false	`flag[0] = false`

Apesar de garantir a exclusão mútua, a terceira solução não é livre de deadlocks. Cenários em que as `flag[0]` e `flag[1]` são definidas como true fazem com que ambos os processos permaneçam em espera ocupada na linha 3 dos códigos apresentados, se bloqueando e gerando deadlock. A Tabela 9 apresenta um cenário em que ocorre deadlock.

Tabela 5. Terceira solução apresentada por Djikstra - deadlock

t	P0	P1	flag[0]	flag[1]	Observação
t0			false	false	inicialização
t1	×		false	false	inicia P0
t2	×		false	false	`while(1);`
t3	×		true	false	`flag[0]=true;`
t4		×	true	false	`while(1);`
t5		×	true	true	`flag[1]=true;`

1.4 Quarta solução - gentileza mútua

Em comparação a solução anterior, a principal ideia dessa é a de gentileza mútua. Ao invés de "travar"o processo em um loop ao identificar que o outro tem intensão de entrar na RC, nessa solução o processo abre mão de sua vontade por um curto período de tempo (linha 5), liberando o outro para executar.

```
1  while(1){
2      flag[0] = true;
3      while (flag[1]){
4          flag[0] = false;
5          //delay for a short time
6          flag[0] = true;
7      }
8      //região crítica
9      flag[0] = false;
10 }
```

Processo 0

```
1  while(1){
2      flag[1] = true;
3      while (flag[0]){
4          flag[1] = false;
5          //delay for a short time
6          flag[1] = true;
7      }
8      //região crítica
9      flag[1] = false;
10 }
```

Processo 1

Apesar de resolver o problema de deadlock, a ideia de gentileza mútua pode acarretar em livelock. Nesse caso, os processos fornecem sua vez ao outro de forma que fiquem se "bloqueando"constantemente, impedindo seu prosseguimento. Essa situação é análoga ao caso em que uma pessoa bloqueia a passagem de outra no corredor: pode acontecer de uma se movimentar para esquerda e outra para direita, de forma que continuem a se bloquear.

Na prática, esse cenário de livelock tende a não se sustentar durante um longo tempo, mas, por ser uma possibilidade, invalida a proposta como solução correta do problema. A Tabela 7 apresenta um cenário onde ocorre livelock.

Tabela 6. Quarta tentativa - livelock

t	P0	P1	flag[0]	flag[1]	Observação
t0			false	false	inicialização
t1		×	false	false	inicia P1
t2		×	false	false	`while(1)`
t3		×	false	true	`flag[1] = true`
t4	×		false	true	inicia P0
t5	×		false	true	`while(1)`
t6	×		true	true	`flag[0] = true`
t7	×		true	true	while ok `while flag[1]`
t8		×	true	true	while ok `while flag[0]`
t9		×	true	false	while ok `flag[1] = false`
t10		×	true	true	while ok `flag[1] = true`
t11	×		false	true	while ok `flag[0] = false`
t12	×		true	true	while ok `flag[0] = true`
t13	×		true	true	while ok `while flag[1]`
t14		×	true	true	while ok `while flag[0]`
t15		×	true	false	while ok `flag[1] = false`
t16		×	true	true	while ok `flag[1] = true`
t17	×		false	true	while ok `flag[0] = false`
t18	×		true	true	while ok `flag[0] = true`

Tabela 7. Quarta solução - gentileza mútua

t	P0	P1	flag[0]	flag[1]	Observação
t0			false	false	inicialização
t1	×		false	false	inicia P0
t2	×		false	false	`while(1)`
t3	×		true	false	`flag[0]=true`
t4	×		true	false	`while falha`
t5	Ⓡ		true	false	P0 na RC
t6	×		false	false	`flag[0] = false`
t7	×		false	false	`while(1)`
t8	×		true	false	`flag[0] = true`
t9		×	true	false	inicia P1
t10		×	true	false	`while(1)`
t11		×	true	true	`flag[1] = true`
t12		×	true	true	while ok ()
t13		×	true	false	`flag[1] = false`
t14		×	true	false	delay
t15		×	true	false	delay
t16		×	true	false	delay
t17		×	true	true	`flag[1] = true`
t18		×	true	true	while ok
t19		×	true	false	`flag[1] = false`
t20		×	true	false	delay
t21		×	true	false	delay
t22		×	true	false	delay
t23	×		true	false	while falha
t24	Ⓡ		true	false	P0 na RC
t25	×		false	false	`flag[0] = false`
t26		×	false	true	`flag[1] = true`
t27		×	false	true	while falhou
t28		Ⓡ	false	true	P1 na RC
t29		×	false	false	flag[1] = false

1.5 Algoritmo de Dekker

O algoritmo de Dekker foi o primeiro a solucionar o problema de exclusão mútua. Duas variáveis compartilhadas são utilizadas para indicar a intenção de cada processo em entrar na seção crítica (`flag[0]` e `flag[1]`). O algoritmo também utiliza a variável compartilhada `turn` para indicar qual processo tem prioridade de execução, isto é, de quem é a vez.

```
while(1){
    flag[0] = true;
    while (flag[1]){
        if (turn == 1){
            flag[0] = false;
            while (turn == 1) {
                nothing();
            }
            flag[0] = true;
        }
    }
    turn = 1;
    //região crítica
    flag[0] = false;
}
```

Processo 0

```
while(1){
    flag[1] = true;
    while (flag[0]){
        if (turn == 0){
            flag[1] = false;
            while (turn == 0) {
                nothing();
            }
            flag[1] = true;
        }
    }
    turn = 0;
    //região crítica
    flag[1] = false;
}
```

Processo 1

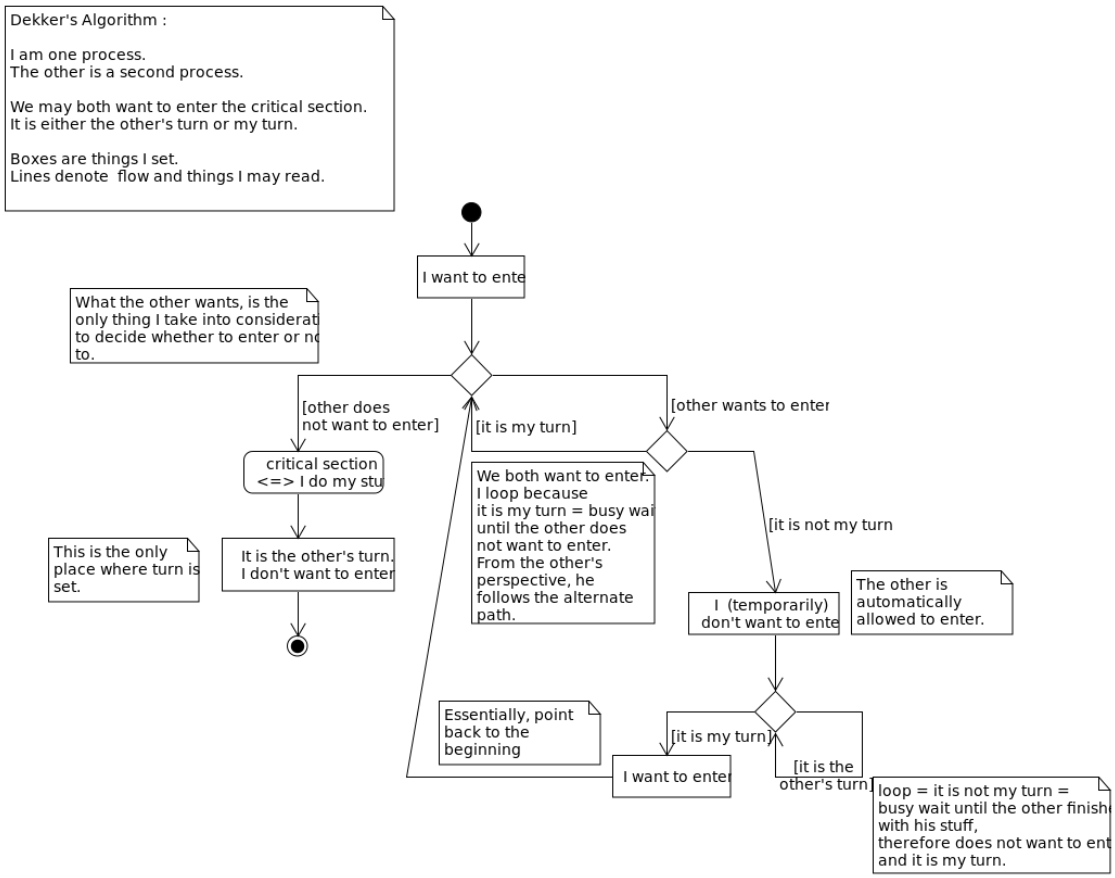

Figura 1. Algoritmo de Dekker - By Lpandelis - Own work, CC BY-SA 4.0[2]

[2]CC BY-SA 4.0: `https://creativecommons.org/licenses/by-sa/4.0`; Link: `https://commons.wikimedia.org/w/index.php?curid=65104901`

Tabela 8. Quinta tentativa - Algoritmo de Dekker correto

t	P0	P1	flag[0]	flag[1]	turn	Observação
t0			false	false	0	inicialização
t1	×		false	false	0	inicia P0
t2	×		false	false	0	`while(1)`
t3	×		true	false	0	`flag[0] = true`
t4		×	true	false	0	inicia P1
t5		×	true	false	0	`while(1)`
t6	×		true	false	0	while falha `while(flag[1])`
t7	×		true	false	1	`turn = 1`
t8	Ⓡ		true	false	1	P0 na RC
t9		×	true	true	1	`flag[1]=true`
t10		×	true	true	1	while ok `while(flag[0])`
t11		×	true	true	1	if falha `if turn == 0`
t12		×	true	true	1	while ok `while(flag[0])`
t13		×	true	true	1	if falha `if turn == 0`
t14	×		false	true	1	`flag[0] = false`
t15	×		false	true	1	`while(1)`
t16	×		true	true	1	`flag[0] = true`
t17	×		true	true	1	while ok `while(flag[1])`
t18	×		true	true	1	if ok `if (turn == 1)`
t19	×		false	true	1	`flag[0] = false`
t20	×		false	true	1	while ok (`while(turn==1)`)
t21	×		false	true	1	`nothing();`
t22		×	false	true	1	while falha (`while(flag[0])`)
t23		×	false	true	0	`turn = 0`
t24		Ⓡ	false	true	0	P1 na RC
t25	×		true	true	0	`flag[0] = 0`
t26	×		true	true	0	while ok (`while(flag[1])`)
t27	×		true	true	0	while ok (`while(flag[1])`)
t28		×	true	false	0	`flag[1]=false`
t29	×		true	false	0	while falha (`while(flag[1])`)
t30	×		true	false	1	`turn = 1`
t31	Ⓡ		true	false	1	P0 na RC
t32	×		flase	false	1	`flag[0]=false`

2 Algoritmo de Petterson

O algoritmo de Petterson garante a exclusão mútua e justiça, e apesar de ter sido implementado inicialmente para 2 processos, pode ser facilmente generalizado para n processos.

```
while(1){
    flag[0] = true;
    turn = 1;
    while (flag[1] && turn==1){
        nothing();
    }
    //região crítica
    flag[0] = false;
}
```

Processo 0

```
while(1){
    flag[1] = true;
    turn = 0;
    while (flag[0] && turn == 0){
        nothing();
    }
    //região crítica
    flag[1] = false;
}
```

Processo 1

Tabela 9. Algoritmo de Petterson

t	P0	P1	flag[0]	flag[1]	turn	Observação
t0			false	false	0	inicialização
t1		×	false	false	0	inicia P1
t2		×	false	false	0	`while(1)`
t3		×	false	true	0	`flag[1] = true`
t4	×		false	true	0	inicia P0
t5	×		false	true	0	`while(1)`
t6	×		true	true	0	`flag[0]=true`
t7	×		true	true	1	`turn = 1`
t8	×		true	true	1	while ok `while(flag[1] && turn==1)`
t9	×		true	true	1	`nothing()`
t10		×	true	true	1	`turn=0`
t11		×	true	true	1	while falha `while (flag[0] && turn == 0)`
t12		Ⓡ	true	true	1	P1 na RC
t13		×	true	false	1	`flag[1] = false`
t14		×	true	false	1	`while(1)`
t15		×	true	true	1	`flag[1] = true`
t16		×	true	true	0	`turn = 0`
t17		×	true	true	0	while ok `while (flag[0] && turn == 0)`
t18		×	true	true	0	nothing
t19		×	true	true	0	while ok `while (flag[0] && turn == 0)`
t20	×		true	true	0	while falha `while (flag[1] && turn == 1)`
t21	Ⓡ		true	true	0	P0 na região crítica
t22		×	true	true	0	`nothing()`
t23		×	true	true	0	while ok `while (flag[0] && turn == 0)`
t24		×	true	true	0	`nothing()`
t25	×		false	true	0	`flag[0] = false`
t26		×	false	true	0	while falha `while (flag[0] && turn == 0)`
t27		Ⓡ	false	true	0	P1 na RC
t28		×	false	false	0	`flag[1]=false`