

# Синхронизация между процеси (*Rev:1.3*)

Любомир Чорбаджиев<sup>1</sup>  
lchorbadjiev@elsys-bg.org

<sup>1</sup>Технологическо училище “Електронни системи”  
Технически университет, София

22 ноември 2007 г.

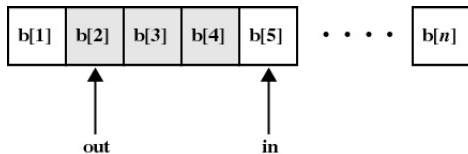
# Съдържание

- 1 Производител/потребител
- 2 Условие на надпревара (race condition)
- 3 Критичен регион (critical section)
- 4 Алгоритми за синхронизация
- 5 Хардуерна поддръжка
  - Забрана на прекъсванията
  - Атомарни инструкции
  - Test And Set
  - Swap
- 6 Семафори

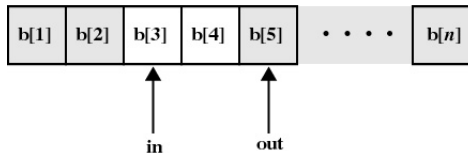
# Производитель/потребитель

```
1 #define BUFFER_SIZE 10
2 typedef struct {
3     //...
4 } item_t;
5 item_t buffer[BUFFER_SIZE];
6 int in=0;
7 int out=0;
```

# Производитель/потребитель



(a)



(b)

# Производитель

```
1  item_t nextProduced;
2
3  while (1) {
4      ...
5      /* produce item in nextProduced */
6      ...
7      while (((in + 1) % BUFFER_SIZE) == out)
8          ; /* do nothing */
9      buffer[in] = nextProduced;
10     in = (in + 1) % BUFFER_SIZE;
11 }
```

# Потребитель

```
1  item_t nextConsumed;
2
3  while (1) {
4      while (in == out)
5          ; /* do nothing */
6      nextConsumed = buffer[out];
7      out = (out + 1) % BUFFER_SIZE;
8      ...
9      /* consume item from nextConsumed */
10     ...
11 }
```

# Производител/потребител

- В общите променливи добавяме нова променлива counter, която брой колко обекта има в момента в буфера buffer.
- Ако counter==0, то буфера е празен.
- Ако counter==BUFFER\_SIZE, то буфера е пълен.

```
1 #define BUFFER_SIZE 10
2 typedef struct {
3     //...
4 } item_t;
5 item_t buffer[BUFFER_SIZE];
6 int in=0;
7 int out=0;
8 int counter=0;
```

# Производитель

```
1  item_t nextProduced;
2
3  while (1) {
4      ...
5      /* produce item in nextProduced */
6      ...
7      while (counter==BUFFER_SIZE)
8          ; /* do nothing */
9      buffer[in] = nextProduced;
10     in = (in + 1) % BUFFER_SIZE;
11     counter++;
12 }
```



# Потребитель

```
1  item_t nextConsumed;
2
3  while (1) {
4      while (counter==0)
5          ; /* do nothing */
6      nextConsumed = buffer[out];
7      out = (out + 1) % BUFFER_SIZE;
8      counter--;
9      ...
10     /* consume item from nextConsumed */
11     ...
12 }
```

# Атомарност

- За да бъде коректно предложеното решение на проблема производител/потребител, е необходимо операциите:

```
counter++;  
counter--;
```

да бъдат атомарни.

- *Атомарна* се нарича операция, която се изпълнява изцяло, без да бъде прекъсвана от операционната система.

# Атомарност

- Операциите

```
counter++;  
counter--;
```

типично не са атомарни.

- Операцията `counter++` на ниво процесор се изпълнява по следния начин:

```
1 register1=counter;  
2 register1=register1+1;  
3 counter=register1;
```

- Операцията `counter--` на ниво процесор се изпълнява по следния начин:

```
1 register2=counter;  
2 register2=register2-1;  
3 counter=register2;
```

# Условие на надпревара

- Когато и производителя и потребителя се опитат конкурентно да променят състоянието на буфера, инструкциите им могат да се преплетат.
- Преплитането на инструкциите на производителя и потребителя зависи от начина, по който двата процеса се планират върху процесора.

# Условие на надпревара

- Да предположим, че първоначално counter=5. Един вариант за преплитане на производителя и потребителя е следния:

```

1 producer: register1=counter      /*register1 = 5*/
2 producer: register1=register1+1 /*register1 = 6*/
3 consumer: register2=counter      /*register2 = 5*/
4 consumer: register2=register2-1 /*register2 = 4*/
5 producer: counter=register1      /*counter = 6 */
6 consumer: counter=register2      /*counter = 4 */

```

- Стойността на counter може да бъде 4 или 6, докато правилния резултат трябва да бъде 5.

# Условие на надпревара

- *Условие на надпревара (race condition)* се нарича ситуацията, при която няколко процеса конкурентно манипулират данни, които са общи, поделени между конкурентните процеси.
- Крайната стойност на поделените (общите) данни зависи от начина, по който процесите се планират върху процесора.
- За да се предотврати състоянието на надпревара (race condition), конкурентните процеси трябва да бъдат синхронизирани.

# Критичен секция (critical section)

- Няколко процеса се състезават за достъп до общи (поделени) данни.
- Всеки процес има участък от кода, в който работи с общите за всички процеси данни. Такъв участък от кода се нарича *критична секция*.
- Трябва да се изгради механизъм, чрез който да се гарантира, че когато един процес се намира в критична секция, никой друг процес не може да влезе в своята критична секция.

# Критичен секция (critical section)

Този механизъм трябва да притежава следните свойства:

- Взаимно изключване: когато процесът  $P_i$  се намира в критична секция, никой друг процес не може да навлезе в своята критична секция. Във всеки един момент от време само един процес може да се намира в критична секция.
- Процес, който не се намира в критична секция, не може да влияе на другите процеси.
- Не трябва да се забавя влизането на процес в критична секция, ако в момента няма друг процес в критична секция.
- Механизмът не трябва да разчита на предположения относно скоростта на изпълнение на процесите.



# Критичен секция (critical section)

Общата структура на процеси, притежаващи критична секция, е следната:

```
1 do {  
2   вход в критичната секция;  
3   ...  
4   критична секция;  
5   ...  
6   изход от критичната секция;  
7   ...  
8   некритична секция;  
9   ...  
10  ...  
11 } while (...);
```

# Алгоритми за синхронизация

- Нека разгледаме два процеса –  $P_0$  и  $P_1$ . Нека с  $i$  обозначим номера на единия процес, тогава номера на другия процес ще бъде  $j = 1 - i$ .
- Нека двата процеса имат следната обща променлива, която служи за синхронизация между процесите:

```
int turn=0;
```

- Стойността на променливата `turn` определя кой процес може да влезе в критичната си секция.
- Ако `turn=0`, то процес  $P_0$  може да влезе в критичната си секция.
- Ако `turn=1`, то процес  $P_1$  може да влезе в критичната си секция.

# Алгоритми за синхронизация

Примерна структура на процеса  $P_i$ :

```
1 do {  
2   while (turn != i)  
3     ; // do nothing  
4   ...  
5   критична секция  
6   ...  
7   turn = j;  
8   ...  
9   некритична секция  
10  ...  
11 } while (...);
```

# Алгоритми за синхронизация

- Алгоритъмът удовлетворява условието за взаимно изключване на процесите. Когато единия процес  $P_i$  се намира в критична секция, другия процес  $P_j$  изчаква, докато дойде неговия ред.
- Коректността на решението се основава на факта, че операцията присвояване на стойност  $turn=j$ ; е атомарна операция.
- Това решение, обаче не удовлетворява останалите изисквания към механизма за синхронизация.
- Когато процесът  $P_i$  мине през критичната си секция, променливата  $turn$  става равна на  $j$ . Ако процесът  $P_i$  се опита отново да влезе в критичната си секция, то той ще бъде блокиран, защото  $turn \neq i$ . Процесът  $P_i$  трябва да изчака процеса  $P_j$ , макар че  $P_j$  не се намира в критична секция.

# Алгоритъм на Петерсон

- Нека отново разгледаме два процеса –  $P_0$  и  $P_1$ . Нека с  $i$  обозначим номера на единия процес, а с  $j = 1 - i$  номера на другия процес.
- Нека двата процеса имат следните общи променливи, които се използват за синхронизация между двата процеса:

```
int turn=0;
bool flag[2];
flag[0]=false;
flag[1]=false;
```

- Стойността на променливата `turn` определя кой процес е на ред да влезе в критичната си секция.
- Стойностите на елементите в масива `flag[]` показват дали съответния процес е готов да влезе в критичната си секция. Когато `flag[i]==true`, процесът  $P_i$  е готов да влезе в критичната си секция.

# Алгоритъм на Петерсон

Примерна структура на процеса  $P_i$ :

```
1 do {  
2   flag[i]=true ;  
3   turn=j ;  
4   while(flag[j]&& turn==j)  
5     ; // do nothing  
6   ...  
7   критична секция  
8   ...  
9   flag[i]=false ;  
10  ...  
11  некритична секция  
12  ...  
13 } while(...);
```

# Забрана на прекъсванията

- Прекъсванията диктуват превключването на контекста на процесора. Когато прекъсванията са забранени, няма какво да предизвика превключване на контекста на процесора.
- Забраната на прекъсванията гарантира взаимното изключване, тъй като това забранява превключването на контекста на процесора и прави невъзможно преплитането на два процеса.
- Този подход работи върху еднопроцесорни системи. Върху многопроцесорни системи забраната на прекъсванията върху единия процесор не гарантира взаимно изключване.

# Атомарни инструкции

- Съвременните процесорни архитектури предоставят атомарни инструкции – инструкции, които гарантирано се изпълняват без да бъдат прекъсвани.
- Тези операции се опират на факта, че на хардуерно ниво обръщането към клетка от паметта забранява всякакви други операции с тази клетка.
- Идеята е за един цикъл на изпълнение да се изпълнят две операции – четене и запис, или четене и проверка на стойността.
- Тъй като тези операции се изпълняват за един цикъл (една инструкция), то върху тях не могат да повлияят никакви други инструкции.



# Test And Set

```
1 bool testandset(int* v) {  
2     if(*v!=0) {  
3         return false;  
4     } else {  
5         *v=1;  
6         return true;  
7     }  
8 }
```

# Test And Set

- Нека разгледаме два процеса –  $P_0$  и  $P_1$ . Нека с  $i$  обозначим номера на единия процес, а с  $j = 1 - i$  номера на другия процес.
- Нека двата процеса имат следната обща променлива, която се използва за синхронизация между двата процеса:

```
bool lock=false ;
```

- Стойността на променливата `lock` определя дали има процес в критична секция. Ако `lock==true`, то това означава, че някой от процесите в момента е в критична секция, и другия процес трябва да чака – не може да влезе в критичната си секция.

# Test And Set

Примерна структура на процеса  $P_i$ :

```
do {
    while (testandset(&lock))
        ; // do nothing
    ...
    // критична секция
    ...
    lock=false;
    ...
    // некритична секция
    ...
} while (...);
```

```
bool testandset(int* v){
    if(*v!=0) {
        return false;
    } else {
        *v=1;
        return true;
    }
}
```

# Swap

```
1 void swap(int* r, int* m) {  
2     int temp=*m;  
3     *m=*r;  
4     *r=temp;  
5 }
```

# Swap

- Нека разгледаме два процеса –  $P_0$  и  $P_1$ . Нека с  $i$  обозначим номера на единия процес, а с  $j = 1 - i$  номера на другия процес.
- Нека двата процеса имат следната обща променлива, която се използва за синхронизация между двата процеса:

```
bool lock=false ;
```

- Стойността на променливата `lock` определя дали има процес в критична секция. Ако `lock==true`, то това означава, че някой от процесите в момента е в критична секция, и другия процес трябва да чака – не може да влезе в критичната си секция.

# Test And Set

Примерна структура на процеса  $P_i$ :

```
bool val=false ;
do {
    val=true ;
    while (val==true)
        swap(lock, val);
    ...
    // критична секция
    ...
    lock=false ;
    ...
    // некритична секция
    ...
} while (...);
```

```
void swap(int* m1, int* m2){
    int temp=*m1;
    *m1=*m2;
    *m2=temp;
}
```

# Хардуерна поддръжка

- Основно предимство на разгледаните средства за синхронизация е че те са приложими към произволен брой процеси, които работят върху произволен брой процесори, които си поделят обща памет (uniform memory access).
- Разгледаните механизми са прости и могат лесно да се провери тяхната коректност.
- Могат да бъдат използвани за поддръжка на множество критични секции.
- *Основният недостатък* на всички разгледани механизми за синхронизация е, че използват “работно чакане” (busy waiting). Докато даден процес чака своя ред за влизане в критична секция, той не спира да работи и да използва процесорно време.

# Семафори

- Семафорът е специален вид променлива, която се използва за комуникация между процесите. Използвайки семафори, даден процес може да изпраща сигнал на друг процес и/или да очакват да получи сигнал от друг процес.
- Когато даден процес очаква да получи сигнал, той преминава в състояние “блокиран” (blocked).
- С всеки семафор е асоциирана целочислена променлива. Върху семафорите могат да се изпълняват само две операции, които са атомарни (неделими).



# Семафори

- Примерна реализация на семафор може да се изгради върху следната структура:

```
1 struct Semaphore_t {  
2     int value;  
3     QueueType_t blocked;  
4 }
```

- Членът `value` е целочислената стойност, асоциирана със семафора.
- Членът `blocked_queue` съдържа опашката от процеси, които са блокирани върху семафора.

# Семафори

- Върху семафорите могат да се изпълняват само две операции — `wait()` и `signal()`.
- При създаването си семафорът може да бъде инициализиран с неотрицателно цяло число.
- Операцията `wait()` намалява стойността на семафора с единица. Ако стойността на семафора стане отрицателна, процеса изпълняващ операцията блокира.
- Операцията `signal()` увеличава стойността на семафора с единица. Ако върху семафора има блокирани процеси, то операцията `signal()` води до разблокиране на някой от блокираните процеси.

# Семафори

- Примерна реализация на операцията wait():

```
1 void wait(Semaphore_t* s) {  
2     s->value--;  
3     if (s->value < 0) {  
4         // добави процеса към s->blocked  
5         // блокирай процеса  
6     }  
7 }
```

# Семафори

- Примерна реализация на операцията `signal()`:

```
1 void signal(Semaphore_t* s) {  
2     s->value++;  
3     if (s->value <= 0) {  
4         // извади един процес P от s->blocked  
5         // постави процеса P в опашката на готовите процеси  
6     }  
7 }
```

# Семафори

- Операциите `wait()` и `signal()` са атомарни.
- Реализацията на механизма на семафорите изисква поддръжка от операционната система.
- Често като специален случай се разглежда така наречения *бинарен семафор* или *мутекс*. Това е семафор, чиято стойност може да приема само две стойности – 0 и 1.

# Бинарен семафор

```
1 struct BinSemaphore_t {  
2     enum {zero,one} value;  
3     QueueType_t blocked;  
4 }  
5 void wait(BinSemaphote_t* s) {  
6     if(s->value==one) {  
7         s->value=zero;  
8     } else {  
9         добави процеса към s->blocked;  
10        блокирай процеса  
11    }  
12 }
```

# Бинарен семафор

```
1 void wait(BinSemaphote_t* s) {  
2     if (s->blocked е празна) {  
3         s->value=one;  
4     } else {  
5         извади един процес P от s->blocked;  
6         постави процеса P в опашката на готовите процеси;  
7     }  
8 }
```

# Синхронизация със семафори

- Нека разгледаме два процеса –  $P_0$  и  $P_1$ . Нека с  $i$  обозначим номера на единия процес, а с  $j = 1 - i$  номера на другия процес.
- Нека двата процеса имат обща семафор:

```
Semaphore_t sem=създаден и инициализиран с 1;
```

- Когато даден процес иска да влезе в критичната си секция, той вика операцията `wait()` върху общия семафор. Когато процеса напуска критичната си секция вика операцията `signal()`.



# Синхронизация със семафори

Примерна структура на процеса  $P_i$ :

```
do {  
    wait(&sem);  
    ...  
    критична секция  
    ...  
    signal(&sem);  
    ...  
    некритична секция  
    ...  
} while(...);
```