Синхронизация между процеси

Любомир Чорбаджиев[1]
lchorbadjiev@elsys-bg.org

[1]Технологическо училище "Електронни системи"
Технически университет, София

15 март 2015 г.

Съдържание

Примерна програма

```
1  long double sum = 0;
2
3  /* THREAD 1 */
4  void *sumfun1(void *p) {
5    int i;
6    for(i = 0; i < ARRAY_SIZE/2; i++) {
7      sum += sin(i) * sin(i) + cos(i) * cos(i);
8    }
9    return NULL;
10 }
11
12 /* THREAD 2 */
13 void *sumfun2(void *p) {
14   int i;
15   for(i = ARRAY_SIZE/2; i < ARRAY_SIZE; i++) {
16     sum += sin(i) * sin(i) + cos(i) * cos(i);
17   }
18   return NULL;
```

Атомарност

- За да бъде коректно предложеното решение на проблема е необходимо операцията:

```
sum += x;
```

да бъде атомарна.

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

Атомарност

- Операцията

```
sum += x;
```

типично не е атомарна.

- Операцията `sum += x` на ниво процесор се изпълнява по следния начин:

```
1 register1=sum;
2 register1=register1+x;
3 sum=register1;
```

Условие на надпревара

- Когато и двете нишки се опитат конкурентно да променят сумата, инструкциите им могат да се преплетат.
- Преплитането на инструкциите на двете нишки зависи от начина, по който те се планират върху процесора.

Условие на надпревара

- Да предположим, че първоначално sum=5. Един вариант за преплитане е следния:

```
1 thread1: register1=sum;       /*register1 = 5*/
2 thread1: register1=register1+x1; /*register1 = 6*/
3 thread2: register2=sum;         /*register2 = 5*/
4 thread2: register2=register2+x2; /*register2 = 6*/
5 thread1: sum=register1       /*sum = 6   */
6 thread2: sum=register2       /*sum = 6   */
```

- Стойността на sum в някои случаи може да е 6, докато правилния резултат трябва да бъде 7.

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!=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.

Swap

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 is empty) {
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 общ 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. Когата 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(...);
```