СИНХРОНИЗАЦИЯ

В една съвременна ОС решенията за това кой процес да използва процесора се взимат от ОС. Когато работи един процес е невъзможно да се детерминира кога работата му ще прекъсне и управлението ще бъде предадено на друг процес. При неправилна синхронизация между процеси, които използват общ ресурс се случва race condition(състезание за ресурс). Решението е да забраним ресурса от това да бъде използван от няколко процеса едновременно. Това се нарича mutual exclusion. Ако критичните секции(сегменти от код, които могат да бъдат достъпени от няколко процеса едновременно) станат атомарни инструкции(да не могат да се прекъсват), тогава проблемът ще бъде разрешен.

Spinlock

Това в съвременните ОС става чрез защита на данните(примерно с бит, който е 0 ако общите данни са свободни и 1 в противен случай). Този бит се нарича **lock**, а самият метод – **spinlock**. Всяка критична секция трябва да започва със spinlock (забраняваме на другите процеси да достъпват секцията) и да завършва със spin-unlock (позволяваме на другите процеси да достъпват секцията). Примерен псевдокод за spinlock:

```
Spinlock:
    R = test.and.set(lock) // атомарна инструкция
    If(R = 1) goto Spinlock //ако ресурсът е зает, започни да циклиш отначало
    /*
    Critical section
    */
Spin_unlock :
    lock = 0
```

Проблеми при spinlock:

- Ако в критичната секция се стартира подпроцес, който се опита да достъпи горния spinlock, ще настъпи безкраен цикъл, защото lock-ът е 1.

- Ако се случи хардуерно прекъсване, предизвикано от периферно устройство, което се опита да достъпи същата критична секция, отново ще се извика процедурата spinlock и зацикляне на процесора, който ползва периферното устройство.

- За да се създаде илюзията за паралелност, ОС използва т.нар. таймер, който представлява периодично прекъсване на сегашния процес и предаване на управлението на някой друг. Ако при прекъсване на процеса, който в момента е в критичната секция, всички други процеси искат да достъпят същата секция, те също ще зациклят в spinlock-а. По този начин спирайки изходния процес, който е в критичната секция, ще приспи всички останали процесори и ще се изгуби много ефективност.

Т.е, за да се избегнат тези проблеми трябва в началото на критичната секция да се забранят прекъсванията на процеса:

```
Spinlock :
    disable_interrupts()
    R = test.and.set(lock) // атомарна инструкция
    If(R = 1) goto Spinlock //ако ресурсът е зает, започни да циклиш отначало
    /*
    Critical section
    */
Spin_unlock :
    lock = 0
    enable_interrupts()
```

Тук се пак наблюдават проблеми. Например ако 2 процеса искат едновременно да достъпят критичната секция и се забранят прекъсванията и при двата, единия започва да изпълнява секцията, но прекъсванията са забранени и за другия(т.е. другият процес цикли и не може да бъде прекъснат, за да извършат други изчисления) , което е недопустимо от гледна точка на ефективност на системата. По тази причина кодът трябва да се усложни:

```
Spinlock :
    disable_interrupts()
    R = test.and.set(lock) // атомарна инструкция
    If(R = 0) goto critical_section
    enable_interrupts()
    goto Spinlock
    critical_section:
    /*
    Critical section
    */
Spin_unlock :
    lock = 0
    enable_interrupts()
```

Semaphore

За да се приспи, обаче, един процес, трябва да се запазят всичките негови данни и това, заедно със стартирането на нов процес, е скъпа операция. Това се нарича **смяна на контекста(context switching**) – запазване на състоянието на даден процес, за да може по-късно да се възстанови и да продължи да работи. Механизъм за синхронизация от високо ниво е т.нар **семафор**.
Всеки процес поддържа процедура block(), която приспива процеса. (Това вече е функция на ядрото на ОС). Когато има нужда от процесът отново, върху него се извиква процедура wakeup(ProcessID), от ОС/друг процес. Идеята на семафора е да защити ресурс от непозволена употреба.

Структурите от данни, необходими за реализацията на един семафор са броят cnt, в който се пази броя на процесите, които могат да бъдат допуснати до ресурса, охраняван от семафора и контейнер Q,в който се пази информация кои процеси чакат да получат достъп до ресурса. Процедурите, необходими за реализация на семафор са: Конструктур init(int), който задава начална стойност на cnt. Q се инициализира да е празен. Метод wait(), който се използва при опит за достъп до ресурса. Броячът се намалява с 1 и ако стане <0, процесът викащ wait() се блокира, а pid-ът му се вкарта в Q. Метод signal(), който се ползва при освобждаването на ресурса. Cnt се увеличава с 1 и ако Q не е празен, един от процесите в него се премахва от Q и се

събужда. Един семафор наричаме силен, ако Q е обикновена опашка и слаб, ако Q е друга структура от данни и при signal() невинаги събуждаме първия приспан процес.
Примерна реализация на семафор:

```
init(cnt) :
    Cnt = 0
    L = empty_queue()
wait() :
    Cnt = cnt - 1
    If(cnt < 0)
        L.put(self)
        Block()//процесът заспива себе си
signal() :
    cnt = cnt + 1
    if (cnt <= 0)
        pid = L.get()
        wakeup(pid)//събуждаме процеса в началото на опашката
```

Тази реализация е проблемна, защото тъй като семафорът е общ ресурс, той може да бъде изпълнен паралелно от 2 процеса. Методите wait() и signal() са критични секции и трябва да бъдат защитени, което няма как да стане със семафор(в момента го реализираме), затова трябва да използваме по-примитивен механизъм(spinlock). В локалните данни на семафора освен cnt и Q, поставяме и 1 бит – lock и променяме имплементацията по следния начин:

```
Wait() :
    spin_lock(lock)
    Cnt = cnt - 1
    If(cnt < 0)
        L.put(self)
        Spin_unlock(lock)
        Block()
    Else
        spin_unlock(lock)

Signal() :
    spin_lock(lock)
    Cnt = cnt +1
    if (cnt<=0)
        pid =L.get()
        spin_unlock(lock)
        wakeup(pid)
```

При wait() spin_unlock() е на правилното място(когато cnt<0 процесът трябва да освободи lock-а, преди да се приспи, иначе lock-ът ще остане заключен завинаги)
При signal() spin_unlock() се намира преди wakeup(pid), защото операцията wakeup() може да отнеме доста време.

Deadlock

Има 4 причини за възникване на deadlock)

· Mutual exclusion → Даден ресурс да може да бъде ползван само от 1 процес по едно и също време

· Hold&wait → Съществува процес, който притежава някакъв ресурс и чака за друг

· Ползването на ресурс от даден процес не може да бъде прекъснато от друг процес(процесът трябва сам да го освободи)

· Цикличност → Съществува множество от процеси [p1,p2,..,pN] такова че p1 чака за p2, p2 чака за p3,… pN чака за p1(например при dining philosophers)

Има различни стратегии за избягване на deadlock, но няма единно правило)

· Можем да номерираме ресурсите и процесите да се опитват да достъпят ресурсите с най-малък номер. Така няма да настъпи deadlock

· Можем да защитим ресурса със специален процес, който да поеме ролята на употребата на ресурса в режим на mutual exclusion, а останалите ресурси да използват ресурса чрез този процес(като прокси). Типичен пример за това е употребата на периферни устройства, които са уникални за системата (напр. Принтер или мрежова карта). Когато имаме принтер, свързан към системата, обикновено има процес, който е отговорен за печатането. Този процес се нарича spooler и когато някоя програма иска да печати да се обръща към него, а не директно към принтера.

Synchronization patterns

Randevouz) Имаме 2 процеса A и B със следните инструкции:

```
A     B
a1    b1
a2    b2
```

Искаме a1 да се изпълни преди b2 и a2 да се изпълни преди b1.

Решение)

```
semaphore aArrived;
semaphore bArrived;

aArrived.init(0);
bArrived.init(0);


Добавяме в кода на процесите следните инструкции:
A                          B
a1                      |  b1
aArrived.signal();      |  bArrived.signal();
bArrived.wait();        |  aArrived.wait();
a2                      |  b2
```

Забележка) Ако разменим местата на wait() и signal() в един от процесите, ще загубим производителност. Ако го направим и на двете места – ще се получи deadlock

Mutex) Имаме 2 процеса, които споделят променлива count. Искаме достъпът до нея да е ексклузивен(mutual exclusion)

Решение)

```
semaphore mutex;
mutex.init(1)

Thread A                       Thread B
mutex.wait();           |   mutex.wait();
   #critical section    |      #critical section
   count=count+1;       |      count=count+1;
mutex.signal();         |   mutex.signal();
```

Multiplex) Да се генерализира предното решение, така че да работи за произволен брой нишки, но не повече от n нишки могат да достъпват кричитната секция по едно и също временно

Решение)

```
semaphore multiplex;
mutex.init(n)

Thread
multiplex.wait();
    #critical section
multiplex.signal();
```

Бариера) Да се генерализира решението на randevouz, така че да работи с произволен брой нишки – приемаме, че има n нишки и че n е съхранено в променлива, която е достъпна до всички нишки. Когато първите n-1 нишки пристигнат, те трябва да блокират, докато не пристигне и последната.

Решение)

```
semaphore mutex;
semaphore barrier;

mutex.init(1);
barrier.init(0);

Thread

mutex.wait();
   count = count+1;
if count==n
        barrier.signal();
mutex.signal();

barrier.wait();
barrier.signal();
```

Producer consumer) Имаме 2 вида нишки – producers и consumers, които се изпълняват конкурентно. Producer-ите създават някакви обекти и ги добавят в споделена структура от данни(буфер), а consumer-ите премахват тези обекти от буфера и ги обработват. Искаме да наложим следните синхронизационни ограничения:

· Докато се добавят/премахват обекти в/от структурата от данни, тя е в неконсистентно състояние. Нишките трябва да имат ексклузивен достъп до нея.

· Ако consumer пристигне, когато буферът е празен, блокира докато някой producer не добави нов обект в буфера.

Имаме следния първоначален код за producer-и и consumer-и.:

```
Basic producer code                                  Basic consumer code

event=waitForEvent();                                 event=buffer.get();
buffer.Add(event);                                    event.process();
```

Решение)

```
Producer                                  Consumer
event=waitForEvent();          |    items.wait();
mutex.wait();                  |    mutex.wait();
   buffer.Add(event);          |        event=buffer.get();
   mutex.signal();             |    mutex.signal();
items.signal();                |    event.process();
```

Забележка) При Producer може да се разменят местата на последните 2 реда, но по този начин имаме малък performance boost, защото няма да се опитваме да събудим consumer,преди да сме освободили mutex-а.

Producer Consumer2) Да предположим, че буферът има краен размер, който ни е известен. Така се появява ново синхронизационно ограничение:

· Ако producer иска да пише, когато буферът е пълен, блокира докато consumer не премахне обект от буфера.

Решение)

```
semaphore mutex;
semaphore items;
semaphore spaces;

mutex.init(1);
items.init(0);
spaces.init(buffer.size());


Producer                                  Consumer
event=waitForEvent();          |    items.wait();
spaces.wait();                 |    mutex.wait();
mutex.wait();                  |        event=buffer.get();
   buffer.Add(event);          |    mutex.signal();
   mutex.signal();             |    spaces.signal();
items.signal();                |    event.process();
```

Readers-writers) Имаме 2 типа нишки – читатели и писатели и споделен ресурс. Искаме да наложим следните синхронизационни ограничения:

· Произволен брой читатели могат да са в критичната секция по едно и също време

· Писателите трябва да имат ексклузивен достъп до кричитната секция

С други думи, писател не може да влезе в критичната секция, докато има някаква друга нишка(читател или писател), която я изпълнява**. Също така, докато има писател в критичната секция, никоя друга нишка не може да я достъпи.**
- ако премахнем условието за произволен брой читатели, задачата е еквивалентна на mutex

Решение)

```
semaphore roomEmpty;
semaphore mutex;


int readers=0;
roomEmpty.init(1);
mutex.init(1);



Writers                                          Readers
roomEmpty.wait();                    |           mutex.wait();
   #critical section for writers     |               readers=readers+1;
roomEmpty.signal();                  |               if readers==1
                                     |                   roomEmpty.wait();
                                     |           mutex.signal();
                                     |           #critical section for readers
                                     |           mutex.wait();
                                     |           readers=readers-1;
                                     |           if readers==0
                                     |                   roomEmpty.signal();
                                     |           mutex.signal();
```

Pattern-и подобни на този са често срещани : първата нишка, която влиза в критичната секция заключва семафор и последната, която излиза го отключва. Нарича се LightSwitch

Забележка) Решението е коректно, но имаме starvation на писателите – когато един писател пристигне, ако има читатели в критичната секция, писателят може да блокира завинаги, докато читателите идват и си отиват.

Readers-writers2) Да се подобри последното решение, за да се избегне starvation на писателите.

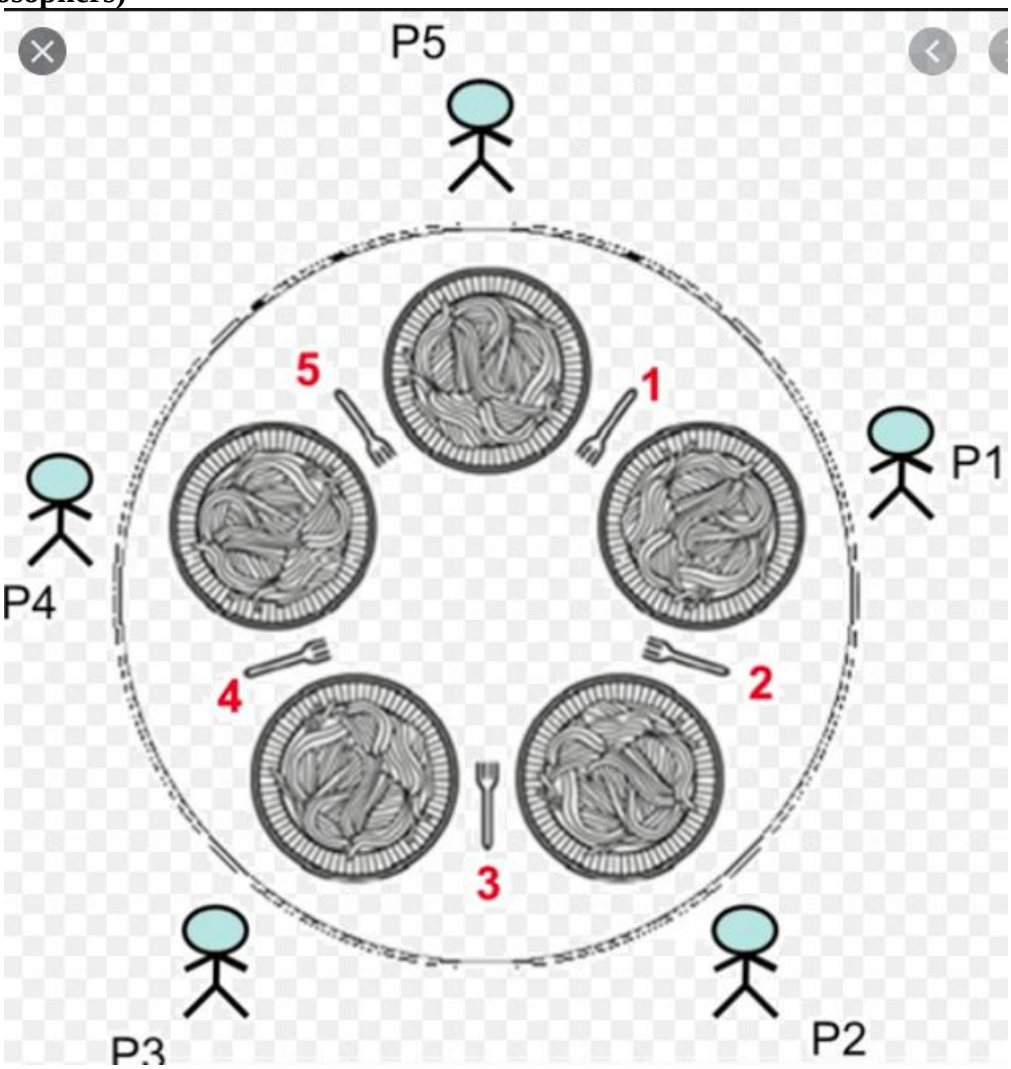
* Достатъчно е да добавим нов семафор, turnstile.init(1). В началото на readers ще имаме turnstile.wait(); turnstile.signal(); а в началото на writers, turnstile.wait();. Така, когато writer иска да пише, повече reader-и няма да бъдат допускани в критичната си секция, докато въпросният writer не излезе от нея.

```
semaphore roomEmpty;
semaphore mutex;
semaphore turnstile;

int readers=0;
roomEmpty.init(1);
mutex.init(1);
turnstile.init(1);
```

```
Writers                                  |          Readers
turnstile.wait();                        |      turnstile.wait();
                                         |      turnstile.signal();
roomEmpty.wait();                        |      mutex.wait();
   #critical section for writers         |          readers=readers+1;
   turnstile.signal();                   |
roomEmpty.signal();                      |          if readers==1
                                         |              roomEmpty.wait();
                                         |      mutex.signal();
                                         |      #critical section for readers
                                         |      mutex.wait();
                                         |      readers=readers-1;
                                         |      if readers==0
                                         |              roomEmpty.signal();
                                         |      mutex.signal();
```

Dining philosophers)



Имаме маса с 5 чинии, 5 вилици и 5 философа, които представляват нишки, които идват на масата и изпълняват следния цикъл:

```
while True:
   think();
   get_forks();
   eat();
   put_forks();
```

Вилиците представляват ресурс, до който нишките трябва да имат ексклузивен достъп, за да прогресират. Философите имат нужда от 2 вилици, за да изпълняват eat(), така че един гладен философ може да трябва да чака съседа си да остави вилица.

Да предположим, че философите имат локална променлива i, която идентифицира всеки философ с число 0-4. Подобно на това, вилиците са номерирани от 0-4 така че философ i има вилица i отдясно и вилица (i+1)%5 отляво.

Приемаме, че функциите think() и eat() са имплементирани. Нашата задача е да имплементираме функциите get_forks() и put_forks() по такъв начин, че да задоволим следните синхронизационни ограничения:

- Само 1 философ може да притежава конкретна вилица по едно и също време
- Трябва да е невъзможно да се случи deadlock

- Трябва да е невъзможно един философ да е в starvation(да чака безкрайно за вилица)
- Трябва да е възможно повече от 1 философ да яде едновременно(производителност)

Дефинираме 2 помощни функции, с които всеки философ може да реферира вилиците около себе си:

```
def left(i): return i
def right(i): return (i+1)%5
```

Тъй като трябва да има ексклузивен достъп до вилиците, естествено е да използваме списък от семафори, по 1 за всяка вилица. Първоначално, всички вилици са свободни:

```
forks - [Semaphore(1) for i in rage(5)]
```

Грешно решение)

```
def get_forks(i):
  fork[right(i)].wait();
  fork[left(i)].wait();

def put_forks(i):
  fork[right(i)].signal();
  fork[left(i)].signal();
```

Тук може да се получи deadlock, ако всеки философ вземе дясната си вилица. Ако има само 4 философа на масата, няма как да стане deadlock : в най-лошия случай всеки взима дясната си вилица, след което на масата ще остане вилица с двама съседи, всеки от които вече има 1 вилица. Тогава, 1 от тези съседи ще я вземе и ще изпълни eat(). Можем да контролираме броят на философи на масата с multiplex, с което ще стигнем до следното

Решение)

```
semaphore multiplex;
multiplex.init(4);


def get_forks(i):
  multiplex.wait();
  fork[right(i)].wait();
  fork[left(i)].wait();

def put_forks(i):
  fork[right(i)].signal();
  fork[left(i)].signal();
  multiplex.signal();
```

Освен, че няма как да се получи deadlock, имаме и гаранция, че няма да има starvation. Представи си, че си философ и стоиш на масата и двамата ти съседи се хранят. Блокиран си за дясната си вилица. Рано или късно, десният ти съсед ще остави вилицата между Вас и ти ще си единствената нишка чакаща за нея, така че ще я вземеш. Аналогично и за лявата вилицата

Забележка) Друг вариант за избягване на deadlock е поне един от философите да е левичар(да взима лявата вилица преди дясната). Ако поне 1 от философите на масата е левичар, а другият – десничар, deadlock е невъзможен

Доказателство) Deadlock може да се случи само ако всичките 5 философа държат 1 вилица и чакат завинаги за другата. В противен случай, 1 от тях може да вземе 2-те вилици, да яде и да напусне. Допускаме, че имаме deadlock. Избираме един от философите. Ако той е левичар, тогава всички философи са левичари, което е протиречие. Ако е десничар, тогава всички са десничари, следователно deadlock е невъзможен.

Synchronization tasks

зад.) Имаме множество паралелно работещи копия на всеки от процесите P и Q:

```
process P          process Q
  p_1                q_1
  p_2                q_2
  p_3                q_3
```

Осигурете чрез семафори синхронизация на работещите копия така че p_1, q_2 и p_3 се редуват циклично.

Решение)

```
semaphore p1Done;
semaphore q2Done;
semaphore p3Done;

p1Done.init(0);
q2Done.init(0);
p3Done.init(1);


Process P                    Process Q
   p3Done.wait();       |    q_1
   p_1                  |    p1Done.wait();
   p1Done.signal();     |    q2
   p_2                  |    q2Done.signal();
   q2Done.wait();       |    q_3
   p_3                  |
   p3Done.signal();     |
```

зад.) Имаме множество паралелно работещи копия на процесите P и Q:

```
process P          process Q
  p_1                q_1
  p_2                q_2
```

Осигурете чрез семафори синхронизация на работещите копия, така че инструкцията q_2 на всяко от работещите копия на Q да се изпълни след като инструкция p_1 е завършила изпълнението си в поне 3 работещи копия на P

Решение)

```
semaphore mutex;
semaphore barrier;

mutex.init(1);
barrier.init(0);

count=0;


Process P                          Process Q
   p_1                    |         q_1
   mutex.wait();          |         barrier.wait();
     count++;             |         barrier.signal();
     if count==3:         |         q_2
       barrier.signal();  |
   mutex.signal();        |
   p_2                    |
```

Задача 4. КН1 Всеки от процесите P, Q и R изпълнява поредица от три инструкции:

```
process P          process Q          process R
  p_1                q_1                r_1
  p_2                q_2                r_2
  p_3                q_3                r_3
```

Осигурете чрез семафори синхронизация на P, Q и R така, че да са изпълнение едновременно условията:

(1) инструкция p_1 да се изпълни преди q_2 и r_2.

(2) инструкция p_3 да се изпълни след q_2 и r_2.

Задача 1, поправка, КН За синхронизация използваме семафори `s`, `t` и `u`, инициализираме ги така:

```
semaphore s, t, u
s.init(0)
t.init(0)
u.init(0)
```

Добавяме в кода на процесите `P`, `Q` и `R` синхронизиращи инструкции:

```
process P              process Q              process R
  p_1                    q_1                    r_1
  s.signal()             s.wait()               s.wait()
  p_2                    s.signal()             s.signal()
  t.wait()               q_2                    r_2
  u.wait()               t.signal()             u.signal()
  p_3                    q_3                    r_3
```

Всяка от инструкциите `q_2` и `r_2` може да се изпълни след като съответният процес премине бариерата `s.wait()`.

Това се случва за пръв път след изпълнението на ред `s.signal()` в процеса `P`, който следва инструкция `p_1`. Така изпълнението на `p_1` преди `q_2` и `r_2` е гарантирано.

Да допуснем, че процесът `Q` преминава през инструкцията си `s.wait()` преди процеса `R`. Веднага след това той изпълнява `s.signal()`, което ще позвали и на `R` да премине през своята инструкция `s.wait()`. Така ще се осигури изпълнението и на двете инструкции `q_2` и `r_2`.

Аналогична е ситуацията, когато `R` преминава през `s.wait()` преди процеса `Q`.

Работата със семафорите `t` и `u` осигурява изпълнението на условие (2).

Решение)

Задача 4. КН2 Всеки от процесите P, Q и R изпълнява поредица от две инструкции:

```
process P        process Q        process R
  p_1              q_1              r_1
  p_2              q_2              r_2
```

Осигурете чрез три семафора синхронизация на P, Q и R така, че отделните инструкции да се изпълнят в следния времеви ред:

```
p_1, q_1, r_1, p_2, q_2, r_2
```

Решение)

```
semaphore t1, t2, t3
t1.init(1)
t2.init(0)
t3.init(0)
```

Добавяме в кода на процесите синхронизиращи инструкции:

```
process P              process Q              process R
  t1.wait()              t2.wait()              t3.wait()
  p_1                    q_1                    r_1
  t2.signal()            t3.signal()            t1.signal()
  t1.wait()              t2.wait()              t3.wait()
  p_2                    q_2                    r_2
  t2.signal()            t3.signal()            t1.signal()
```

Задача 6: Всеки от процесите P и Q изпълнява поредица от три инструкции:

```
process P              process Q
  p_1                    q_1
  p_2                    q_2
  p_3                    q_3
```

Осигурете чрез два семафора синхронизация на P и Q така, че отделните инструкции да се изпълнят в следния времеви ред:

```
p_1, q_1, p_2, q_2, p_3, q_3
```

Решение)

```
semaphore t1, t2
t1.init(1)
t2.init(0)
```

Добавяме в кода на процесите P и Q синхронизиращи инструкции:

```
process P                process Q
  t1.wait()                t2.wait()
  p_1                      q_1
  t2.signal()              t1.signal()
  t1.wait()                t2.wait()
  p_2                      q_2
  t2.signal()              t1.signal()
  t1.wait()                t2.wait()
  p_3                      q_3
  t2.signal()              t1.signal()
```

Задача: Всеки от процесите P, Q и R изпълнява поредица от три инструкции:

```
process P                process Q                process R
  p_1                      q_1                      r_1
  p_2                      q_2                      r_2
  p_3                      q_3                      r_3
```

Осигурете чрез семафори синхронизация на P, Q и R така, че да се изпълнят следните изисквания:

(а) Инструкция p_1 да се изпълни преди q_2 и r_2.

(б) Ако q_2 се изпълни преди r_2, то и q_3 да се изпълни преди r_2.

(в) Ако r_2 се изпълни преди q_2, то и r_3 да се изпълни преди q_2.

Решение)

За синхронизация използваме семафор `t`, инициализираме го с блокиращо начално състояние:

```
semaphore t
t.init(0)
```

Добавяме в кода на процесите `P`, `Q` и `R` синхронизиращи инструкции:

```
process P          process Q          process R
  p_1                q_1                r_1
  t.signal()         t.wait()           t.wait()
  p_2                q_2                r_2
  p_3                q_3                r_3
                     t.signal()         t.signal()
```

Всяка от инструкциите `q_2` и `r_2` може да се изпълни след като броячът на семафора `t` стане положителен.

Това се случва за пръв път след изпълнението на ред `t.signal()` в процеса `P`, който следва инструкция `p_1`. Така гарантираме изпълнението на изискване (а).

След като броячът на семафора стане 1, един от процесите `Q` и `R` ще достигне пръв до ред `t.wait()` и ще го нулира отново.

Ако процесът `Q` пръв достигне инструкцията `t.wait()`, той ще изпълни редове `q_2` и `q_3`, а процесът `R` ще чака ново увеличение на брояча на семафора, което се случва след изпълнението на последния ред `t.signal()` в процеса `Q`. Така гарантираме изпълнението на изискване (б).

Ако процесът `R` пръв достигне инструкцията `t.wait()`, той ще изпълни редове `r_2` и `r_3`, а процесът `Q` ще чака ново увеличение на брояча на семафора, което се случва след изпълнението на последния ред `t.signal()` в процеса `R`. Така гарантираме изпълнението на изискване (в).

Задача 6. Паралелно работещи копия на всеки от процесите `P` и `Q` изпълняват поредица от две инструкции:

```
process P          process Q
  p_1                q_1
  p_2                q_2
```

Осигурете чрез семафори синхронизация на работещите копия така че:

а) Във произволен момент от времето да работи най-много едно от копията.

б) Работещите копия да се редуват във времето – след изпълнение на копие на `P`, да следва изпълнение на копие на `Q`, и обратно.

в) Първоначално е разрешено да се изпълни копие на `P`.

Решение)

Задача 6. Използваме два семафора – `s_p` и `s_q`, инициализираме ги така:

```
semaphore s_p, s_q
s_p.init(1)
s_q.init(0)
```

Добавяме в кода на процесите `P` и `Q` синхронизиращи инструкции:

```
process P                process Q
  s_p.wait()               s_q.wait()
  p_1                      q_1
  p_2                      q_2
  s_q.signal()             s_p.signal()
```

Задача 4. СИ Всеки от процесите `P` и `Q` изпълнява поредица от три инструкции:

```
process P                process Q
  p_1                      q_1
  p_2                      q_2
  p_3                      q_3
```

Осигурете чрез два семафора синхронизация на `P` и `Q` така, че да са изпълнени едновременно следните времеви зависимости:
(1) инструкция `p_1` да се изпълни преди `q_2`
(2) инструкция `q_2` да се изпълни преди `p_3`
(3) инструкция `q_1` да се изпълни преди `p_2`
(4) инструкция `p_2` да се изпълни преди `q_3`

Забележка: За решение с повече семафори ще получите 20 точки.

Решение)

Задача 4. СИ Условията (1) и (3) определят времева среща (randevouz) на процесите след първата им инструкция.

Аналогично, (2) и (4) определят randevouz на процесите след втората им инструкция.

За двате срещи използваме два семафора – `t1` и `t2`, инициализираме ги с блокиращо начално състояние:

```
semaphore t1,t2
t1.init(0)
t2.init(0)
```

Добавяме в кода на процесите `P` и `Q` синхронизиращи инструкции:

```
process P                process Q
  p_1                      q_1
  t1.signal()              t2.signal()
  t2.wait()                t1.wait()
  p_2                      q_2
  t1.signal()              t2.signal()
  t2.wait()                t1.wait()
  p_3                      q_3
```

зад

Задача 5. Всеки от процесите P и Q изпълнява поредица от две инструкции:

```
process P          process Q
  p_1                q_1
  p_2                q_2
```

Осигурете чрез семафори синхронизация на P и Q така, че инструкция p_1 да се изпълни преди q_2, а q_1 да се изпълни преди p_2.

решение

Задача 5. За двете искани в условието синхронизации използваме два семафора – t1 и t2, инициализираме ги с блокиращо начално състояние:

```
semaphore t1,t2
t1.init(0)
t2.init(0)
```

Добавяме в кода на процесите P и Q синхронизиращи инструкции:

```
process P          process Q
  p_1                q_1
  t1.signal()        t2.signal()
  t2.wait()          t1.wait()
  p_2                q_2
```

Инструкцията q_2 ще се изпълни след като процесът Q премине бариерата t1.wait(). Това се случва след изпълнението от P на ред t1.signal(), който следва инструкция p_1.

Аналогично, инструкцията p_2 ще се изпълни след изпълнението на ред t2.signal(), който следва инструкция q_1.

Решението на задачата осигурява среща във времето (rendezvous) на двата процеса. Важен е редът на извикване на инструкциите, управляващи семафорите. Ако го обърнем, получаваме класически пример за deadlock:

```
process P          process Q
  p_1                q_1
  t2.wait()          t1.wait()
  t1.signal()        t2.signal()
  p_2                q_2
```

Зад.

Задача 1, СИ, 25.08

Множество паралелно работещи копия на процеса P изпълняват поредица от две инструкции:

```
process P
  p_1
  p_2
```

Осигурете чрез семафори синхронизация на работещите копия, така че:

Инструкцията p_2 на всяко от работещите копия да се изпълни след като инструкция p_1 е завършила изпълнението си в поне 3 работещи копия.

Упътване: Освен семафори, ползвайте и брояч.

Решение.

Задача 1, СИ

За исканите в условието синхронизации използваме брояч cnt и два семафора – m1 и m2, инициализираме ги така:

```
semaphore m1, m2
m1.init(1)
m2.init(0)
int cnt=0
```

Добавяме в кода на процеса P синхронизиращи инструкции:

```
process P
  p_1
  m1.wait()
   cnt=cnt+1
   if cnt=3 m2.signal()
  m1.signal()
  m2.wait()
  m2.signal()
  p_2
```

Семафорът m1 ползваме като мутекс, който защитава брояча.

Стойността на cnt е равна на броя копия на процеса P, които са изпълнили своята първа инструкция.

Семафорът m2 блокира изпълнението на инструкция p_2.

Когато третото копие на процеса P изпълни p_1, към семафора m2 се подава сигнал, който го деблокира и позволява на всички копия да изпълнят втората си инструкция.

Зад.

Задача: Преди стартиране на процесите `P` и `Q` са инициализирани два семафора и брояч:

```
semaphore e, m
e.init(1); m.init(1)
int cnt = 0
```

Паралелно работещи копия на `P` и `Q` изпълняват поредица от инструкции:

```
process P                    process Q
  m.wait()                     e.wait()
   cnt=cnt+1                    q_section
   if cnt=1 e.wait()           e.signal()
  m.signal()
  p_section
  m.wait()
   cnt=cnt-1
   if cnt=0 e.signal()
  m.signal()
```

Дайте обоснован отговор на следните въпроси:

(а) Могат ли едновременно да се изпълняват инструкциите `p_section` и `q_section`?

(б) Могат ли едновременно да се изпълняват няколко инструкции `p_section`?

(в) Могат ли едновременно да се изпълняват няколко инструкции `q_section`?

(г) Има ли условия за deadlock или starvation за някой от процесите?

Упътване:

Ще казваме, че `P` е в критична секция, когато изпълнява инструкцията си `p_section`. Същото за `Q`, когато изпълнява `q_section`.

Изяснете смисъла на брояча `cnt` и какви процеси могат да бъдат приспани в опашките на двата семафора.

Покажете, че в опашката на семафора `e` има най-много едно копие на `P` и произволен брой копия на `Q`.

Покажете, че в момента на изпълнение на `e.signal()` в кой да е от процесите, никой процес не е в критичната си секция.

Решение

Първо забелязаме, че семафорът `m` се ползва само от `P` в ролята на mutex. В неговата опашка може да има само копия на `P` и само едно работещо копие може да намалява/увеличава брояча синхронизирано с блокирането/освобождаването на семафора `e`.

Увеличаването на `cnt` става преди критичната секция на `P`, а намаляването след нея. Ако не вървят никакви копия на `Q`, лесно се убеждаваме, че могат да се изпълняват произволен брой критични секции на `P`, като броячът съвпада с броя на паралелно изпълняваните критични секции. Така отговорът на въпрос (б) е ДА.

Заемането на семафора `e` в `P` става точно когато `cnt` променя стойността си от 0 в 1. Освобождаването става точно когато `cnt` променя стойността си от 1 в 0.

Тъй като при инициализацията броячът на `e` е 1, а употребата му и в двата вида процеси започва със заемане и завършва с освобождаване, само едно копие от двата типа ще може да премине `e.wait()`. Разглеждаме два случая:

(А) Процесът `Q` преминава. Тогава ще се изпълни критичната му секция, но само от това копие. Останалите копия на `Q` ще бъдат приспани от първата си инструкция. Следователно отговорът на въпрос (в) е НЕ.

Ако версия на `P` пробва `e.wait()`, тя също ще бъде приспана. Това ще стане точно когато `cnt` променя стойността си от 0 в 1, тоест не се изпълняват критични секции на `P`. В момента на приспиване и мутекса `m` е блокиран. Това обстоятелство ще блокира всички опити на други копия на `P` да преминат `m`. В този случай в опашката на семафора `e` има точно едно копие на `P`.

(В) Процесът `P` преминава. Ще започне изпълнение на неговата критична секция и евентуално на други копия на `P`, докато `cnt>0`. През този период всички копия на `Q` ще бъдат приспани от първата си инструкция. Когато `cnt` намалее до 0, никое копие не изпълнява критична секция.

От двата разгледани случая следва, че в един момент могат да се изпълняват няколко критични секции на `P` или една критична секция на `Q`. Следователно отговорът на въпрос (а) е НЕ.

В описаната схема няма условия за deadlock. `Q` не може да инициира deadlock, тъй като ползва само един ресурс. `P` също не може поради реда на заемане на ресурсите (първо заема семафора `m`, после `e`).

В описаната схема има условия за гладуване (starvation) на процес `Q`. Нека критичната секция на `P` се изпълнява бавно и `Q` започва работа след `P`. Ще започне изпълнение на критична секция на `P` и ако постоянно започват работа нови копия, броячът `cnt` може да остане положителен неограничено време. Така `Q` ще бъде приспан неограничено дълго.

Зад

Задача 4. Всеки от процесите `P` и `Q` изпълнява поредица от три инструкции:

```
process P          process Q
  p_1                q_1
  p_2                q_2
  p_3                q_3
```

Осигурете чрез семафори синхронизация на `P` и `Q` така, че инструкция `p_1` да се изпълни преди `q_2`, а `q_2` да се изпълни преди `p_3`.

Решение

Задача 4. За двата момента на синхронизация използваме два семафора – `t1` и `t2`, инициализираме ги с блокиращо начално състояние:

```
semaphore t1,t2
t1.init(0)
t2.init(0)
```

Добавяме в кода на процесите `P` и `Q` синхронизиращи инструкции:

```
process P                process Q
  p_1                      q_1
  t1.signal()              t1.wait()
  p_2                      q_2
  t2.wait()                t2.signal()
  p_3                      q_3
```

Инструкцията `q_2` ще се изпълни след като броячът на семафора `t1` стане положителен. Това се случва след изпълнението на ред `t1.signal()`, който следва инструкция `p_1`.

Аналогично, инструкцията `p_3` ще се изпълни след като броячът на семафора `t2` стане положителен. Това се случва след изпълнението на ред `t2.signal()`, който следва инструкция `q_2`.

Зад

Задача 1. сесия 2 Тройна среща – процесите `P`, `Q` и `R` изпълняват поредица от две инструкции:

```
process P          process Q          process R
  p_1                q_1                r_1
  p_2                q_2                r_2
```

Осигурете чрез семафори синхронизация на `P`, `Q` и `R`, така че:

а) инструкцията `p_1` да се изпълни преди `q_2` и `r_2`

б) инструкцията `q_1` да се изпълни преди `p_2` и `r_2`

в) инструкцията `r_1` да се изпълни преди `p_2` и `q_2`

Решение – not so sure

s1.init(0)
s2.init(0)
s3.init(0)

```
p_1            q_1            r_1
 s1.signal()   s2.signal()  s3.singnal()
   s2.wait()     s1.wait()   s1. wait()
   s3.wait()     s3.wait()     s2.wait()
p_2            q_2          r_2
```

Зад

Задача 1. Множество паралелно работещи копия на всеки от процесите `P` и `Q` изпълняват поредица от две инструкции:

```
process P                process Q
  p_1                      q_1
  p_2                      q_2
```

Осигурете чрез семафори синхронизация на `P` и `Q`, така че поне една инструкция `p_1` да се изпълни преди всички `q_2`, и поне една инструкция `q_1` да се изпълни преди всички `p_2`.

Решение

Задача 1. За двете искани в условието синхронизации използваме два семафора – `t1` и `t2`, инициализираме ги с блокиращо начално състяние:

```
semaphore t1,t2
t1.init(0)
t2.init(0)
```

Добавяме в кода на процесите `P` и `Q` синхронизиращи инструкции:

```
process P                process Q
  p_1                      q_1
  t1.signal()              t2.signal()
  t2.wait()                t1.wait()
  t2.signal()              t1.signal()
  p_2                      q_2
```

Произволна инструкция `q_2` ще се изпълни, след като изпълняващото я копие на процеса `Q` премине бариерата `t1.wait()`. Бариерата ще се отпуши след изпълнението от поне едно копие на `P` на ред `t1.signal()`, който следва инструкция `p_1`.

Копията на `Q` изпълняват поредица `t1.wait()`, `t1.signal()`. Така семафорът `t1` ще събуди всички приспани други копия на `Q` и ще осигури завършването им.

Аналогично, произволна инструкция `p_2` ще се изпълни след първото изпълнение на ред `t2.signal()` в някое копие на `Q`, който следва инструкция `q_1`.

Забележка: Някои студенти забелязаха, че може да настъпи препълване на брояча на някой семафор. Това зависи от реализацията на семафора, особено ако брояча се пази в 16 или 32-битово цяло число. Те предложиха по-фино решение, за което даваме до 8 точки бонус:

Задача 1. прецизно решение За двете искани в условието синхронизации използваме два брояча, два семафора – `t1` и `t2` и мутекс `m`, инициализираме ги така:

```
semaphore t1,t2,m
t1.init(0)
t2.init(0)
m.init(1)
int c1=0, c2=0
```

Добавяме в кода на процесите `P` и `Q` синхронизиращи инструкции:

```
process P                process Q
  p_1                      q_1
  m.wait()                 m.wait()
  if c1=0                  if c2=0
     c1=1                     c2=1
     t1.signal()              t2.signal()
  m.signal()               m.signal()
  t2.wait()                t1.wait()
  t2.signal()              t1.signal()
  p_2                      q_2
```

В това решение след изпълнението на `p_1` семафорът `t1` се вдига само веднъж, при първото преминаване през тази критична секция.

Охраняваме броячите с мутекс, но дори да не го направим, броят на паралелно работещи копия, които ще вдигнат семафора е малък и няма да препълни брояча на съответния семафор.

Задача 1, (20 точки)

Всеки от процесите `P`, `Q` и `R` изпълнява поредица от инструкции:

```
process P          process Q          process R
  p_1                q_1                r_1
  p_2                q_2                r_2
```

Процесите `P` и `Q` са единични, процесът `R` се изпълнява в много копия.

Осигурете чрез семафори синхронизация на `P`, `Q` и `R` така, че да се изпълнят едновременно следните изисквания:

(а) Всички инструкции на `P` и `Q` да се изпълнят преди инструкция `r_1` на всяко копие на `R`.

(б) Процесите `P` и `Q` да се изпълнят ефикасно, т.е. да е възможно паралелното им изпълнение, без да се изчакват.

Задача 1. решение 1 За синхронизация използваме семафори `m` и `s`, и брояч `cnt`, инициализираме ги така:

```
semaphore m, s
m.init(1)
s.init(0)
int cnt=0
```

Добавяме в кода на процесите `P`, `Q` и `R` синхронизиращи инструкции:

```
process P
  p_1
  p_2
  m.wait()
  cnt++
  if (cnt==2)
     s.signal()
  m.signal()
```

```
process Q
  q_1
  q_2
  m.wait()
  cnt++
  if (cnt==2)
     s.signal()
  m.signal()
```

```
process R
  s.wait()
  s.signal()
  r_1
  r_2
```

Задача 1. решение 2 За синхронизация използваме семафори `sp` и `sq`, инициализираме ги така:

```
semaphore sp, sq
sp.init(0)
sq.init(0)
```

Добавяме в кода на процесите `P`, `Q` и `R` синхронизиращи инструкции:

```
process P
  p_1
  p_2
  sp.signal()
```

```
process Q
  q_1
  q_2
  sq.signal()
```

```
process R
  sp.wait()
  sp.signal()
  sq.wait()
  sq.signal()
  r_1
  r_2
```

r_2

Задача 1. решение 3 За синхронизация използваме семафори sp и sq, инициализираме ги така:

```
semaphore sp, sq
sp.init(0)
sq.init(0)
```

Добавяме в кода на процесите P, Q и R синхронизиращи инструкции:

```
process P          process Q          process R
  p_1                q_1                sq.wait()
  p_2                q_2                sq.signal()
  sp.signal()        sp.wait()          r_1
                     sq.signal()        r_2
```

Задача 1, (15 точки)

Всеки от процесите P, Q и R изпълнява поредица от три инструкции:

```
process P          process Q          process R
  p_1                q_1                r_1
  p_2                q_2                r_2
  p_3                q_3                r_3
```

Осигурете чрез семафори синхронизация на P, Q и R така, че да се изпълнят едновременно следните изисквания:

(а) Някоя от инструкциите p_2 и q_2 да се изпълни преди r_2.

(б) Ако инструкция p_2 се изпълни преди r_2, то q_2 да се изпълни след r_2.

(в) Ако инструкция q_2 се изпълни преди r_2, то p_2 да се изпълни след r_2.

Задача 1. За синхронизация използваме семафори `f` и `u`, инициализираме ги така:

```
semaphore f, u
f.init(1)
u.init(0)
```

Добавяме в кода на процесите `P`, `Q` и `R` синхронизиращи инструкции:

```
process P              process Q              process R
  p_1                    q_1                    r_1
  f.wait()               f.wait()               u.wait()
  p_2                    q_2                    r_2
  u.signal()             u.signal()             f.signal()
  p_3                    q_3                    r_3
```

Инструкция `r_2` може да се изпълни след като семафорът `u`, който в началото е блокиран, получи сигнал. Това става единствено след изпълнението на някоя от инструкциите `p_2` и `q_2`. Така осигуряваме изпълнението на условие (а).

Броячът на семафора `f` в началото е 1, само един от процесите `P` и `Q` ще премине реда си `f.wait()` и ще го нулира, другият процес ще чака сигнал. Това става само след изпълнението на ред `f.signal()` от процеса `R`, след изпълнение на инструкция `r_2`. Така осигуряваме изпълнението на условия (б) и (в).

Ако процесът `P` пръв достигне инструкцията `f.wait()`, ще се изпълни предпоставката на условие (б), редът на изпълнение на интересните инструкции ще е `p_2, r_2, q_2`.

Ако процесът `Q` пръв достигне инструкцията `f.wait()`, ще се изпълни предпоставката на условие (в), редът на изпълнение на интересните инструкции ще е `q_2, r_2, p_2`.