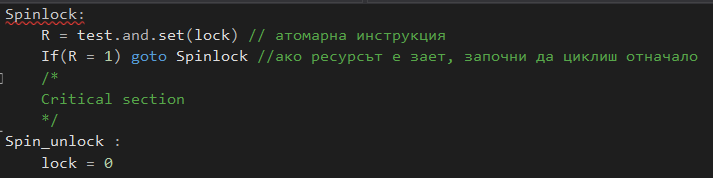
**СИНХРОНИЗАЦИЯ**

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 достъпeни от няколко процеса едновременно) станат атомарни инструкции(да не могат да се прекъсват), тогава проблемът ще бъде разрешен.

**Spinlock**

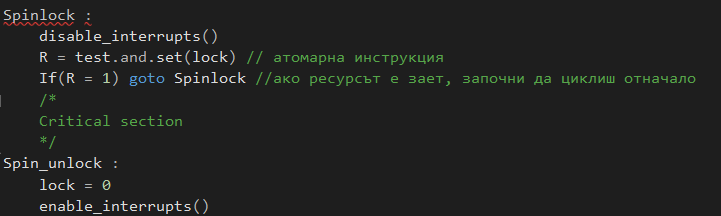
Това в съвременните ОС става чрез защита на данните(примерно с бит, който е 0 ако общите данни са свободни и 1 в противен случай). Този бит се нарича **lock**, а самият метод – **spinlock**.

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:

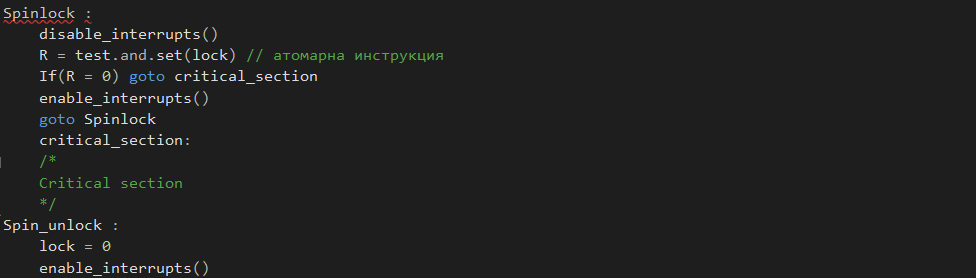


Проблеми при spinlock:

* 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-a. 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:

Тук пак се 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:

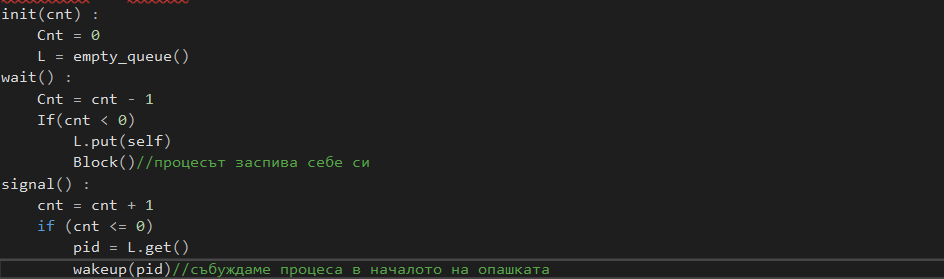


**Semaphore**

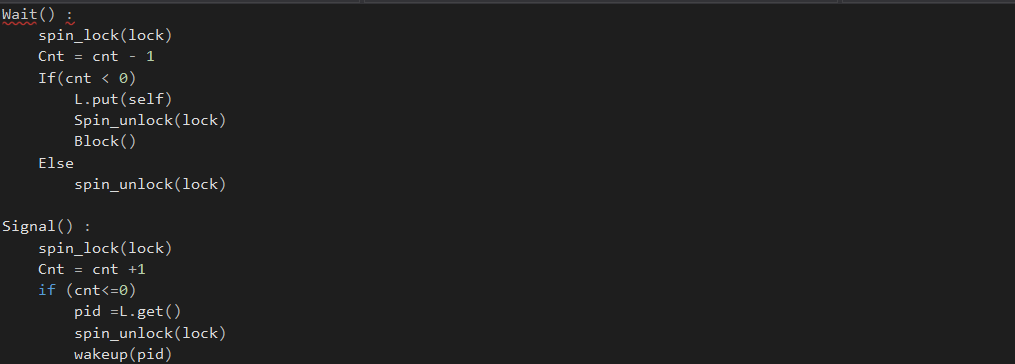
За да се приспи, обаче, един процес, трябва да се запазят всичките негови данни и това, заедно със стартирането на нов процес, е скъпа операция. Това се нарича **смяна на контекста( context switching** ) – запазване на състоянието на даден процес, за да може по-късно да се възстанови и да продължи да 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:

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:



При wait() spin\_unlock() е на правилното място(когато cnt<0 процесът трябва да освободи lock-а, прееди 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.

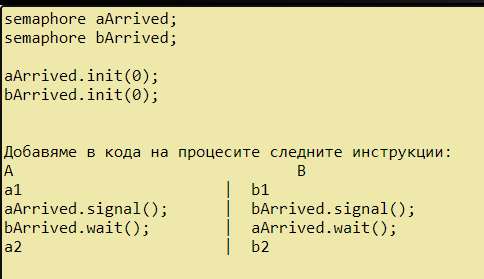
**Synchronization patterns**

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

****

Искаме a1 да се изпълни преди b2 и а2 да се изпълни преди b1.

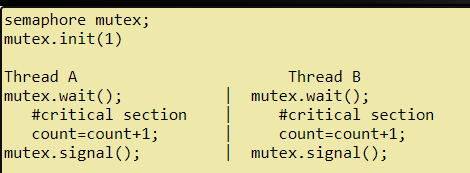
**Решение)**



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

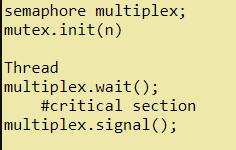
**Мutex)** Имаме 2 процеса, които споделят променлива count. Искаме достъпът до нея да е ексклузивен(mutual exclusion)

**Решение)**



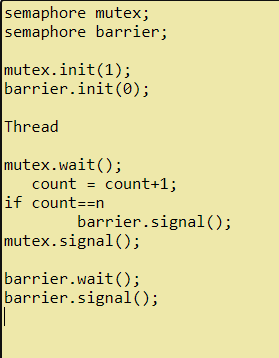
**Multiplex)** Да се генерализира предното решение, така че да работи за произволен брой нишки, но не повече от n нишки могат да достъпват кричитната секция по едно и също временно

**Решение)**

****

**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.

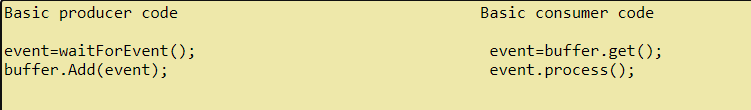
**Решение)**



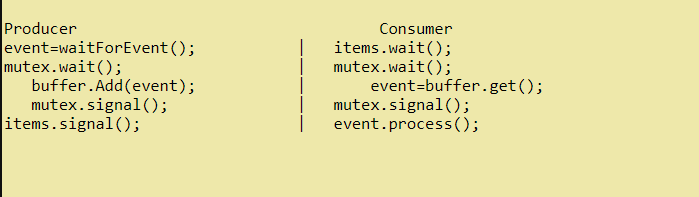
**Producer consumer)** Имаме 2 вида нишки – producers и consumers, които се изпълняват конкурентно. Producer-ите създават някакви обекти и ги добавят в споделена структура от данни(буфер), а consumer-ите премахват тези обекти от буфера и ги обработват. Искаме да наложим следните синхронизационни ограничения:

* 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.:

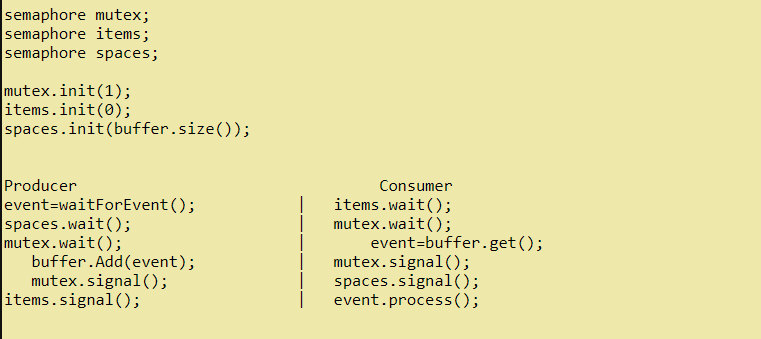


**Решение)**

****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-a.

**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)**

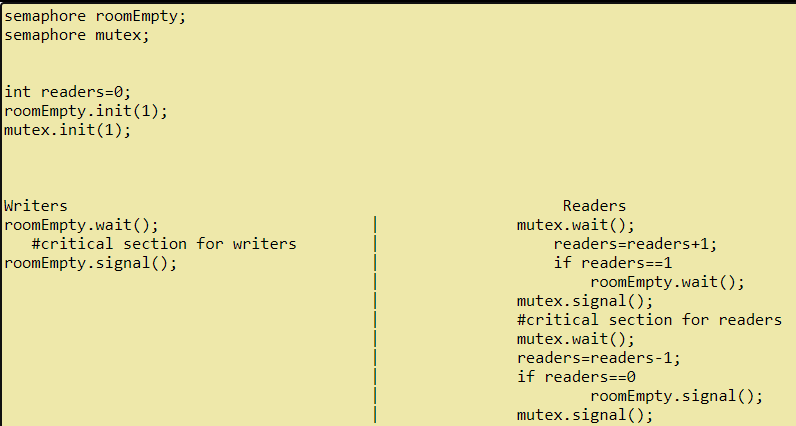
**Readers-writers)** Имаме 2 типа нишки – читатели и писатели и споделен ресурс. Искаме да наложим следните синхронизационни ограничения:

* 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

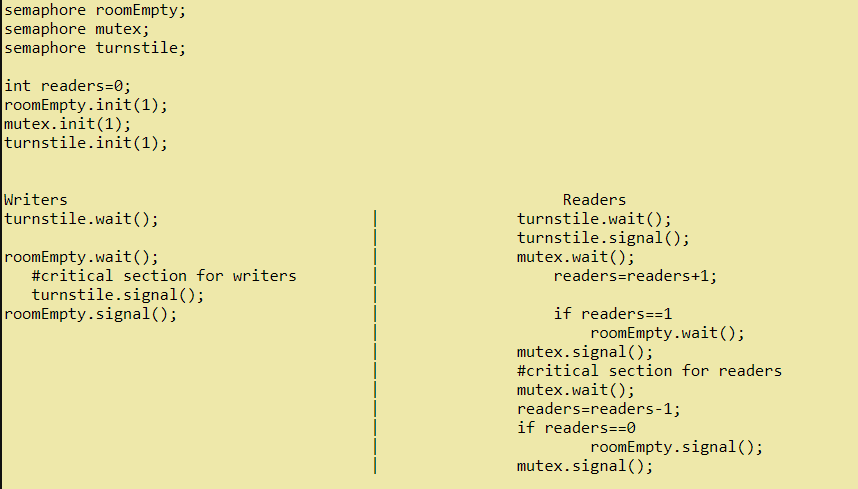
**Решение)**

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

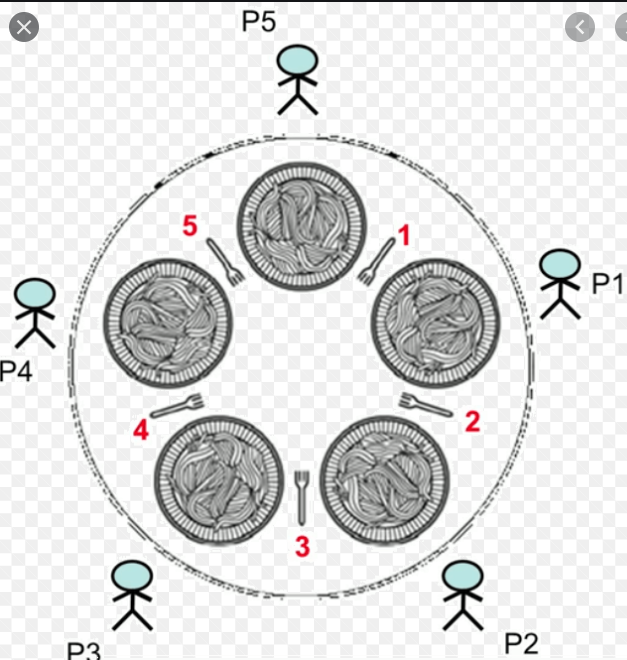
Забележка) Решението е коректно, но имаме starvation на писателите – когато един писател пристигне, ако има читатели в критичната секция, писателят може да блокира завинаги, докато читателите идват и си отиват.

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

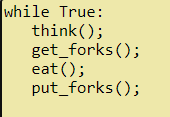
\* Достатъчно е да добавим нов семафор, turnstile.init(1). В началото на readers ще имаме turnstile.wait(); turnstile.signal(); а в началото на writers, turnstile.wait();. Така, когато writer иска да пише, повече reader-и няма да бъдат допускани в критичната си секция, докато въпросният writer не излезе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



**Dining philosophers)**

****

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



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)

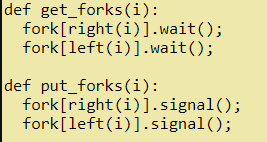
Дефинираме 2 помощни функции, с които всеки философ може да реферира вилиците около себе си:



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:

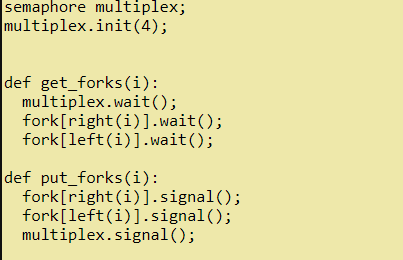


**Грешно решение)**

****

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)**

****

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 протироречие. Ако е десничар, тогава всички са десничари, следователно deadlock е невъзможен.

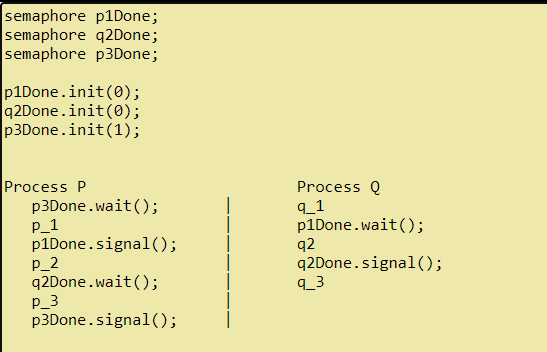
**Synchronization tasks**

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

****

Осигурете чрез семафори синхронизация на работещите копия така че p\_1, q\_2 и p\_3 се редуват циклично.

**Решение)**

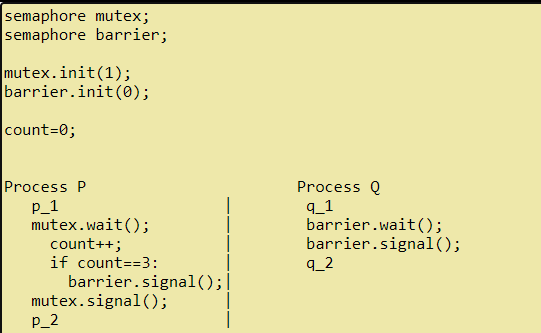
****

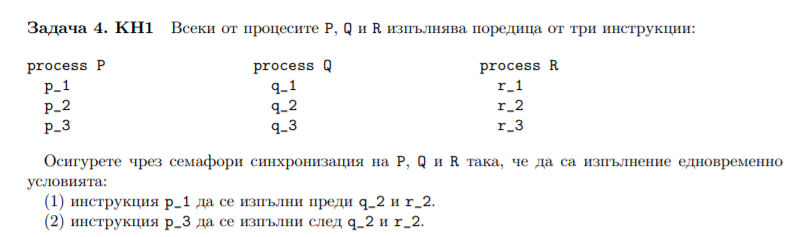
**зад.)** Имаме множество паралелно работещи копия на процесите P и Q:

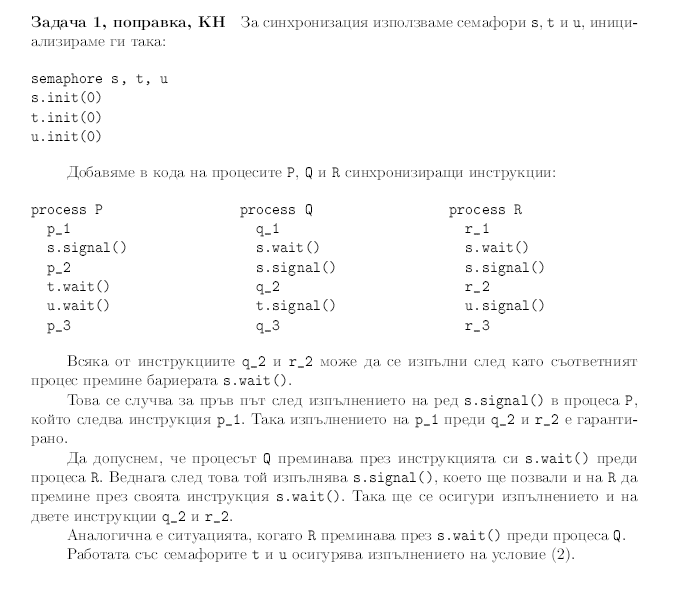
****

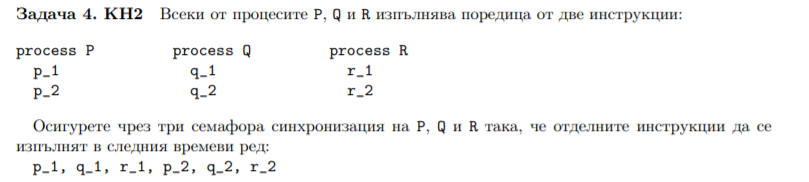
Осигурете чрез семафори синхронизация на работещите копия, така че инструкцията q\_2 на всяко от работещите копия на Q да се изпълни след като инструкция p\_1 e завършила изпълнението си в поне 3 работещи копия на P

**Решение)**

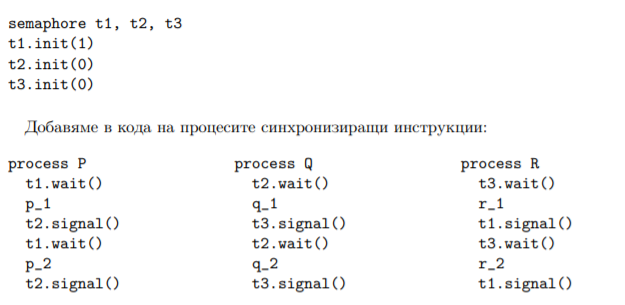


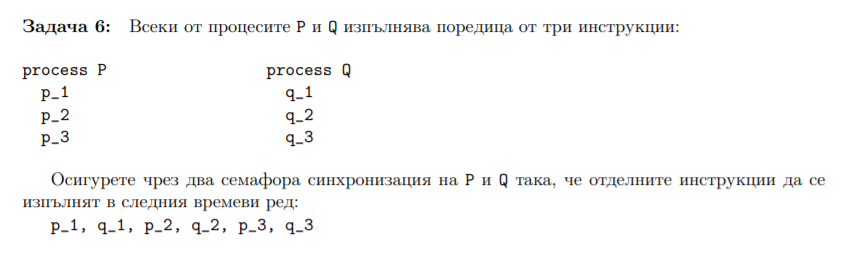


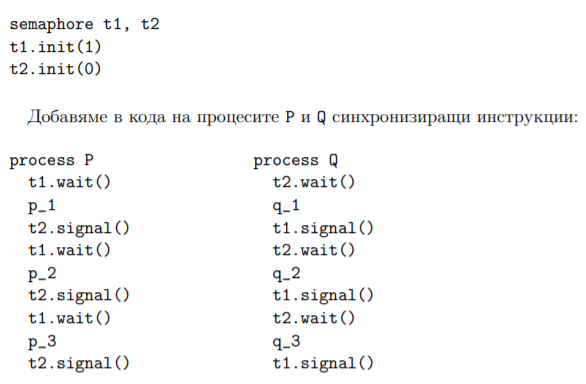
**Решение)** 

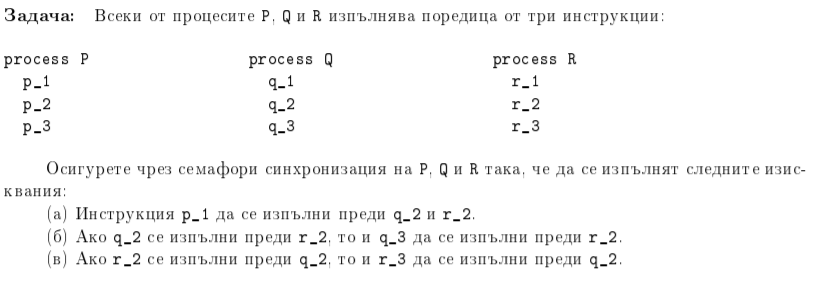
****

**Решение)**

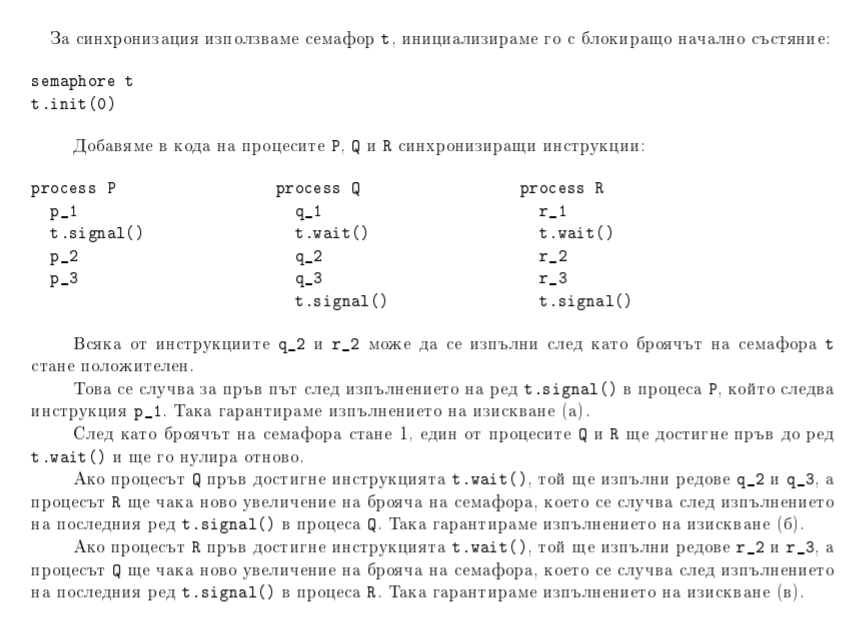
****

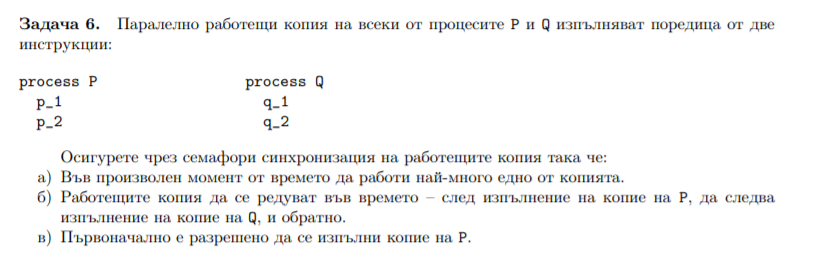
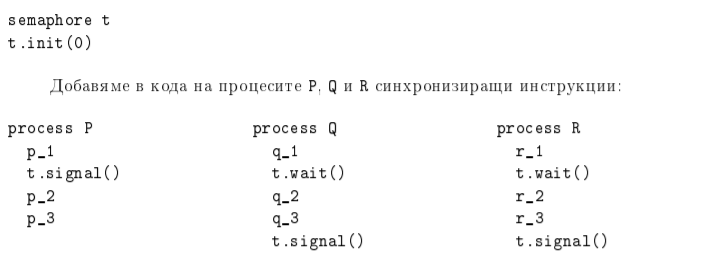
**Решение)**

****

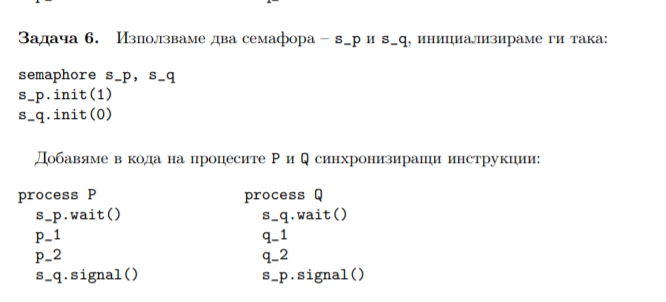
****

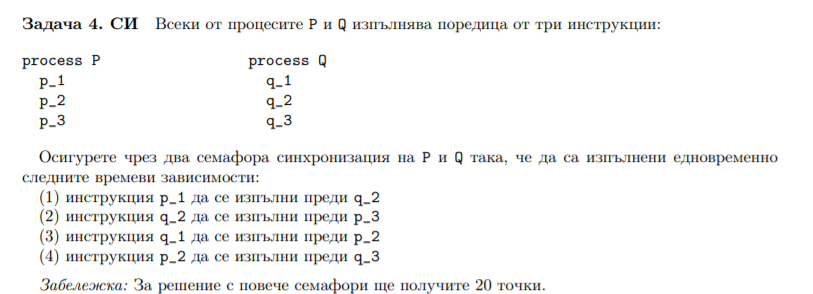
**Решение)**

****

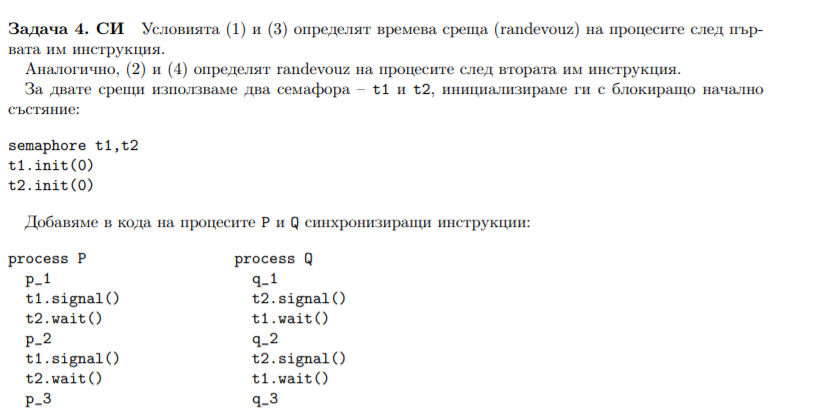
****

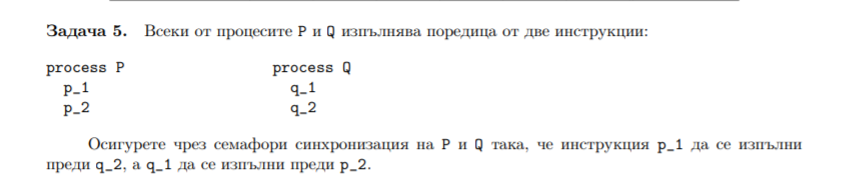
**Решение)**

****

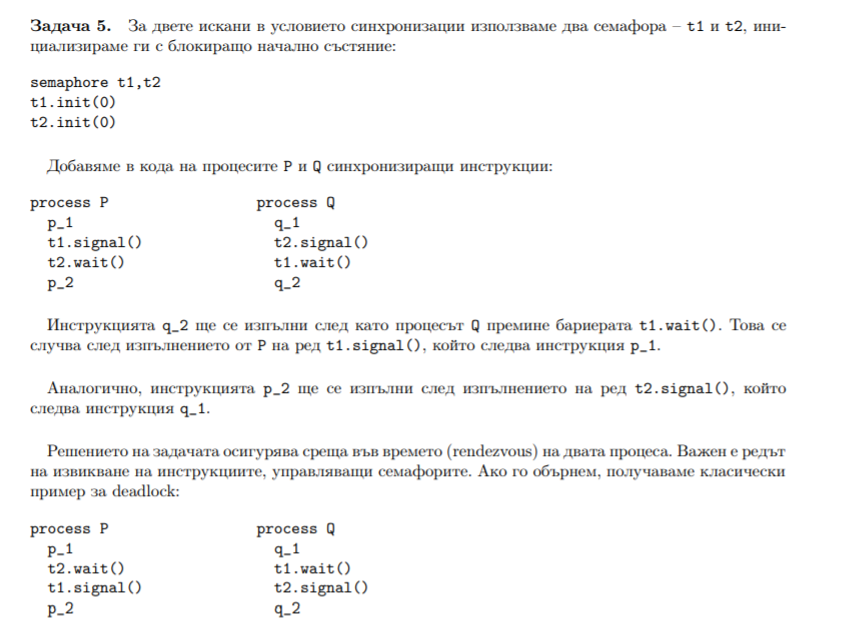
****

**Решение)**

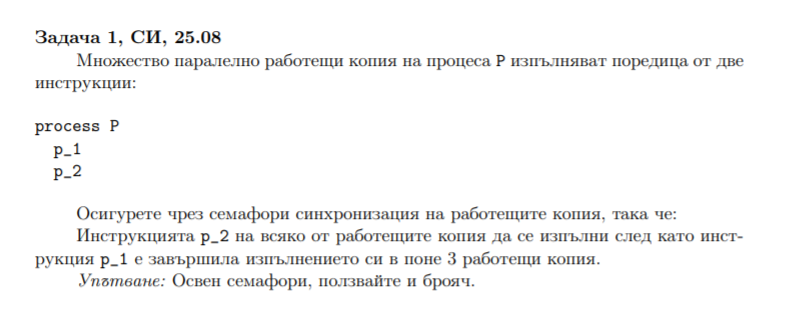
****

**зад **

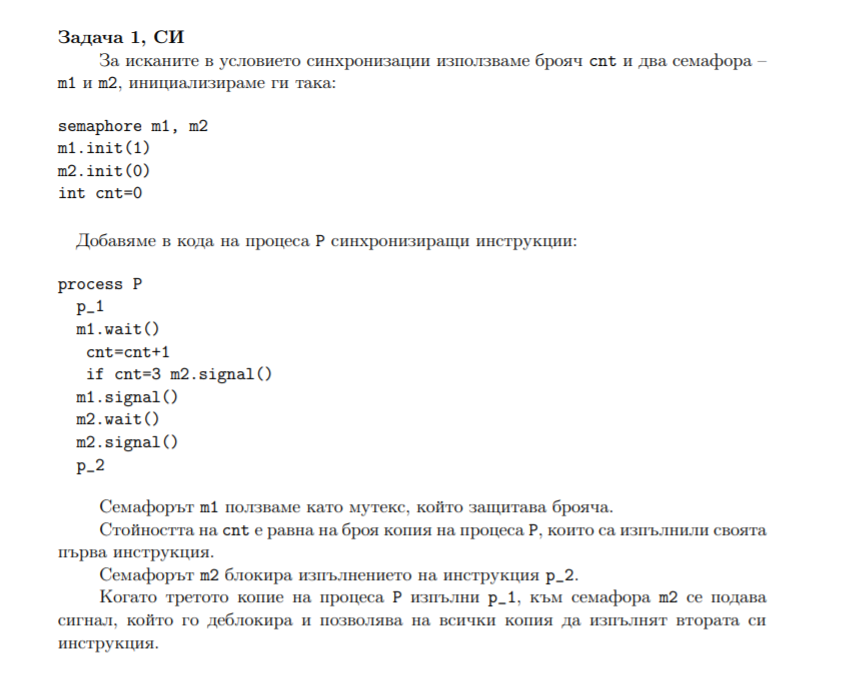
**решение**

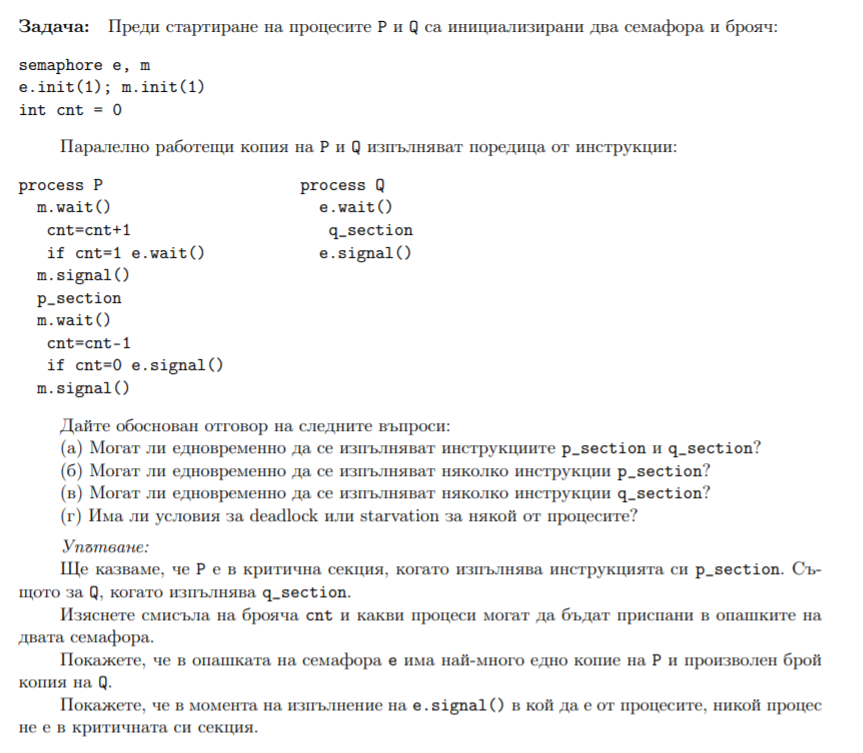
****

**Зад.**

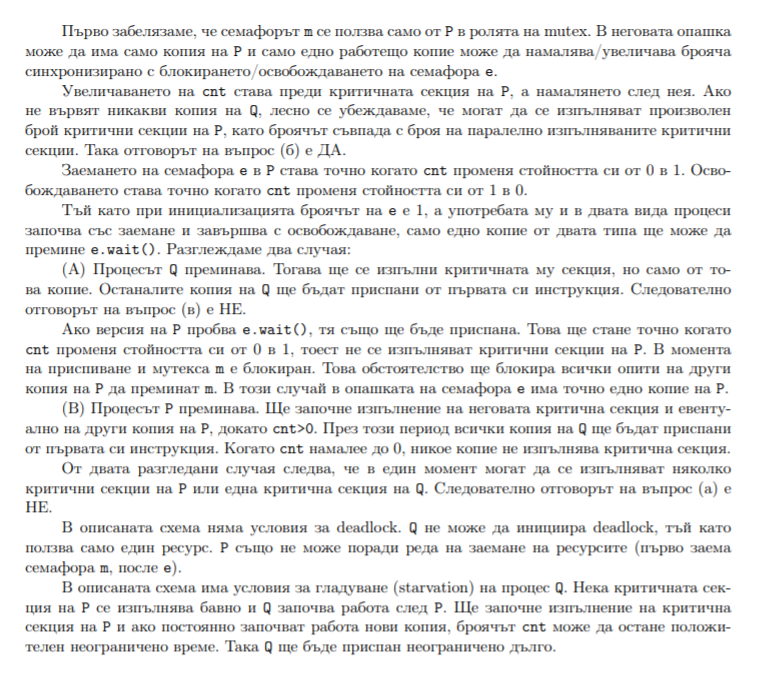
****

**Решение.**

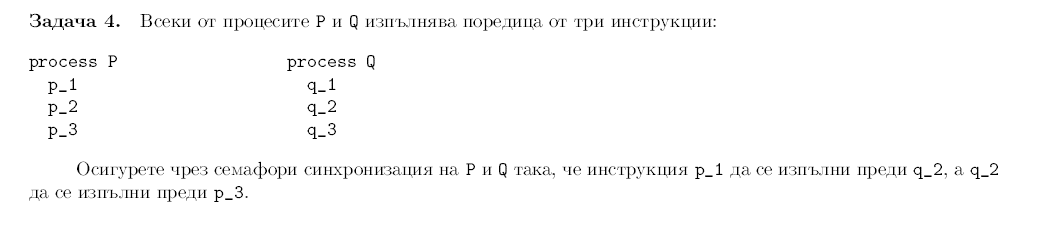
****

**Зад. **

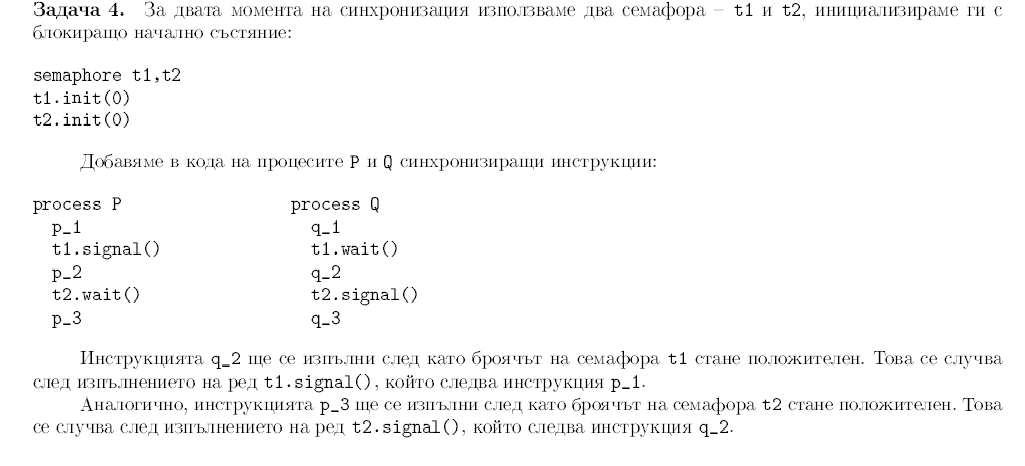
**Решение**

****

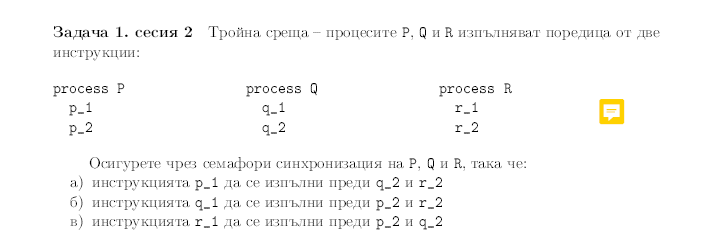
**Зад**

****

**Решение**

****

**Зад**

****

**Решение – not so sure**

s1.init(0)

s2.init(0)

s3.init(0)

p\_1 q\_1 r\_1

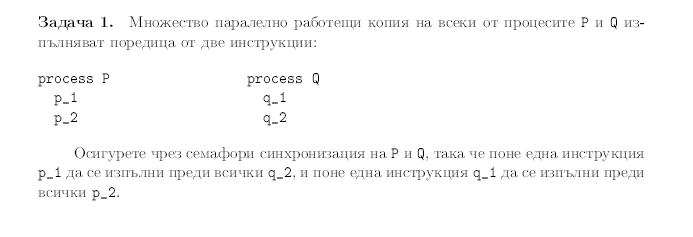
s1.signal() s2.signal() s3.singnal()

s2.wait() s1.wait() s1. wait()

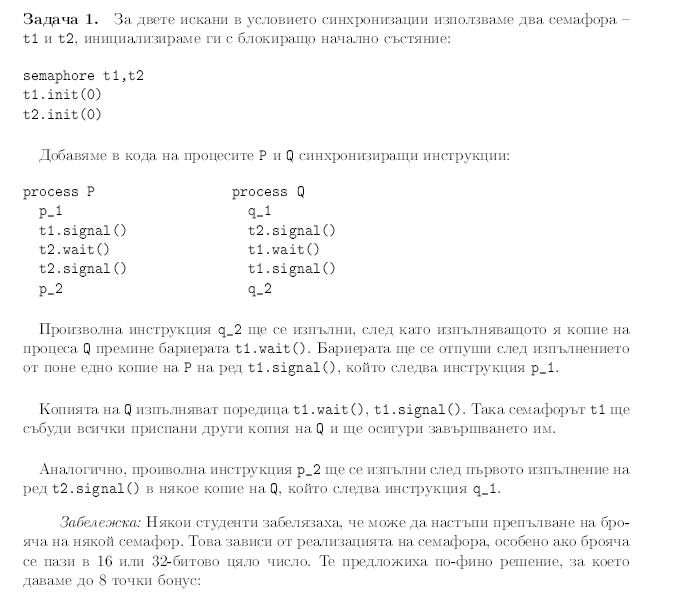
s3.wait() s3.wait() s2.wait()

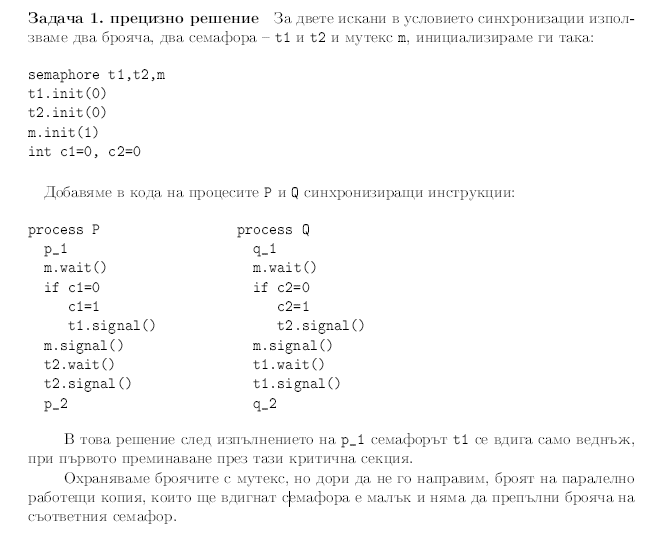
p\_2 q\_2 r\_2

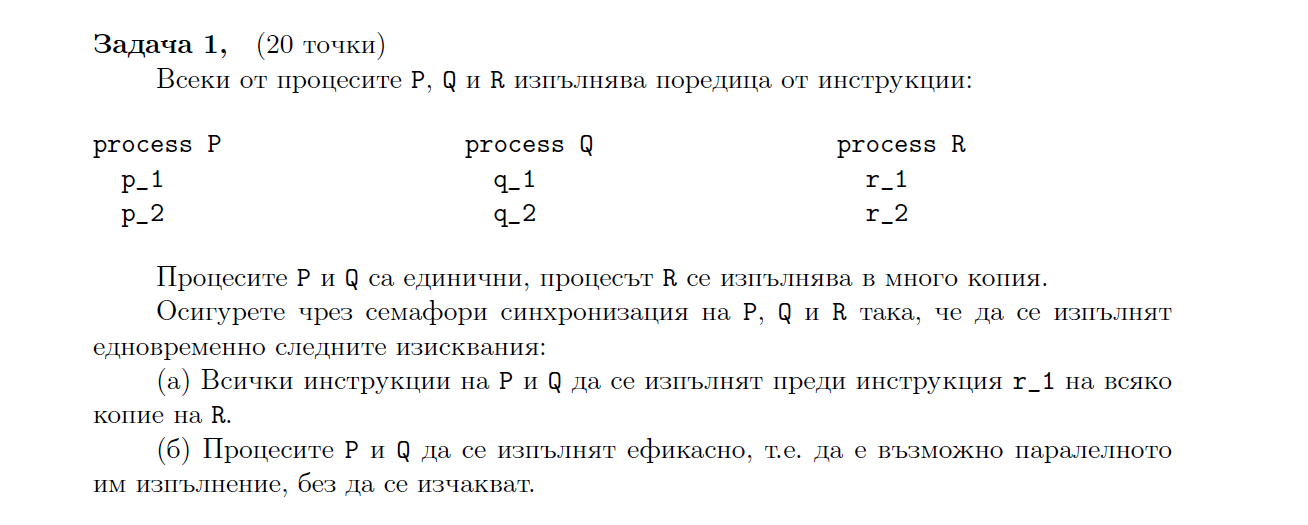
**Зад**

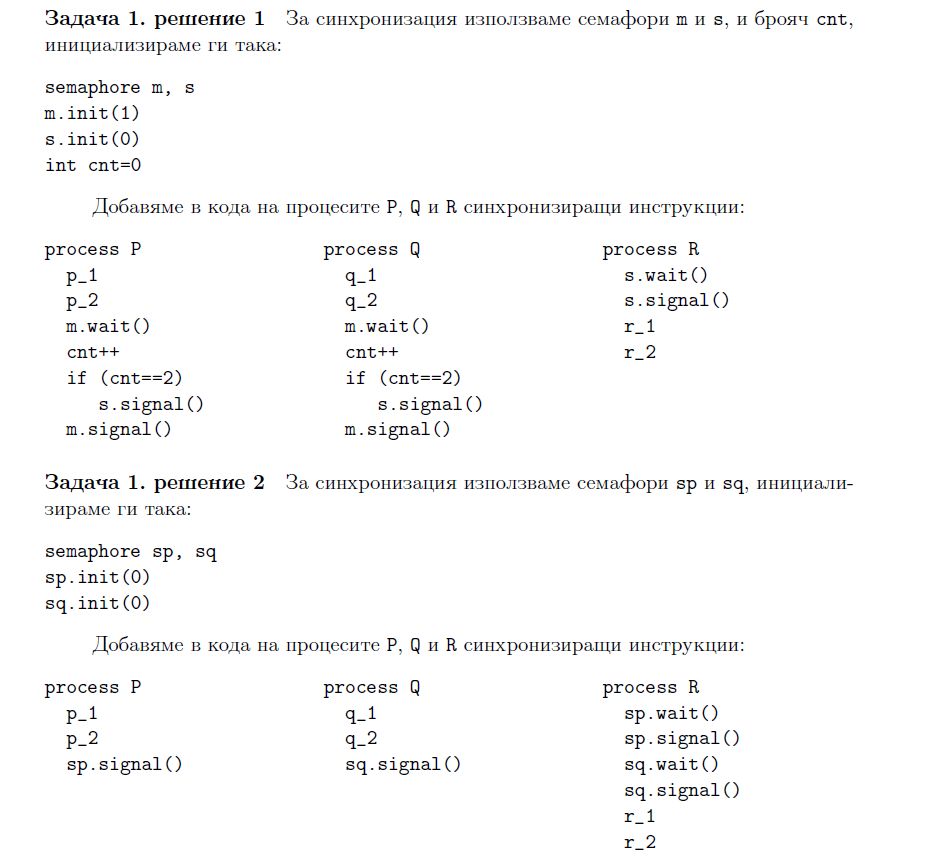
****

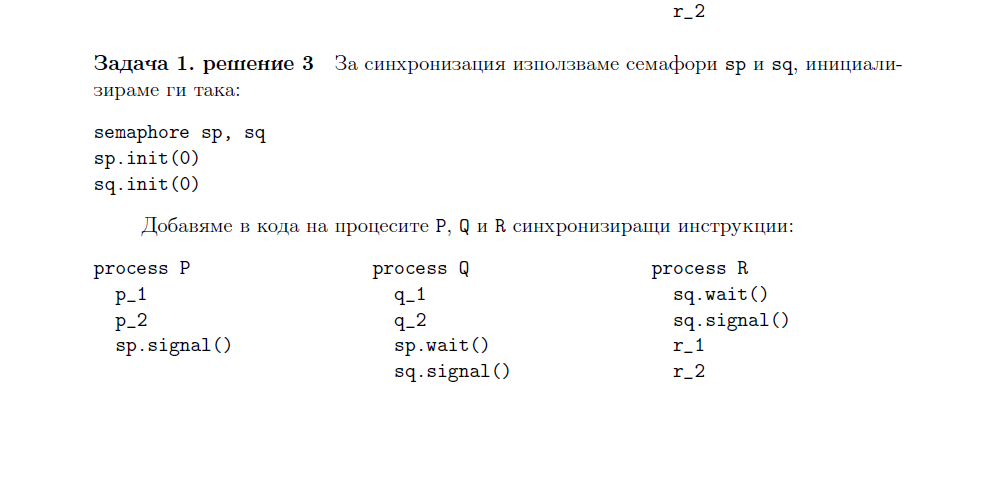
**Решение**

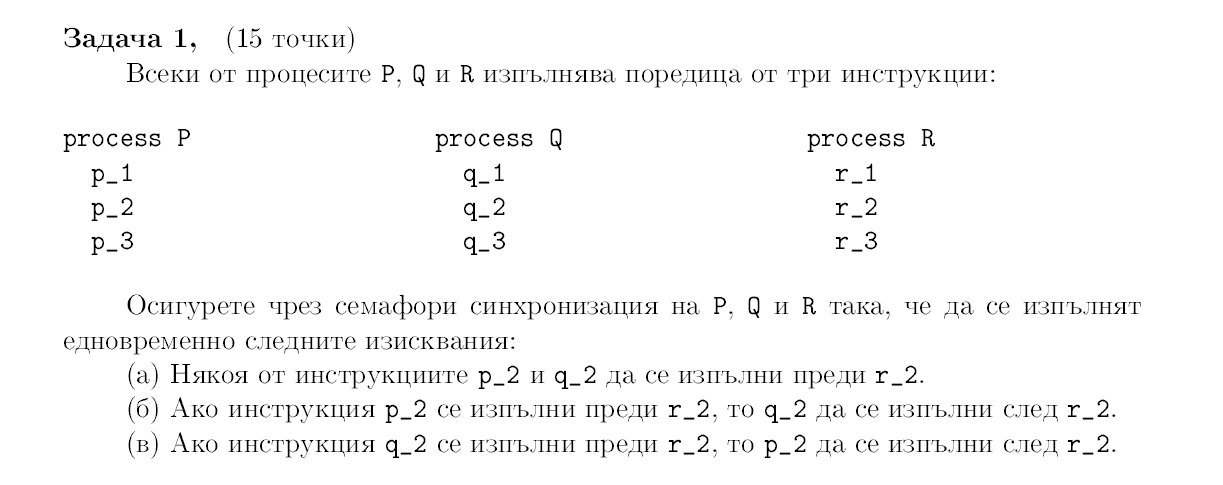
****

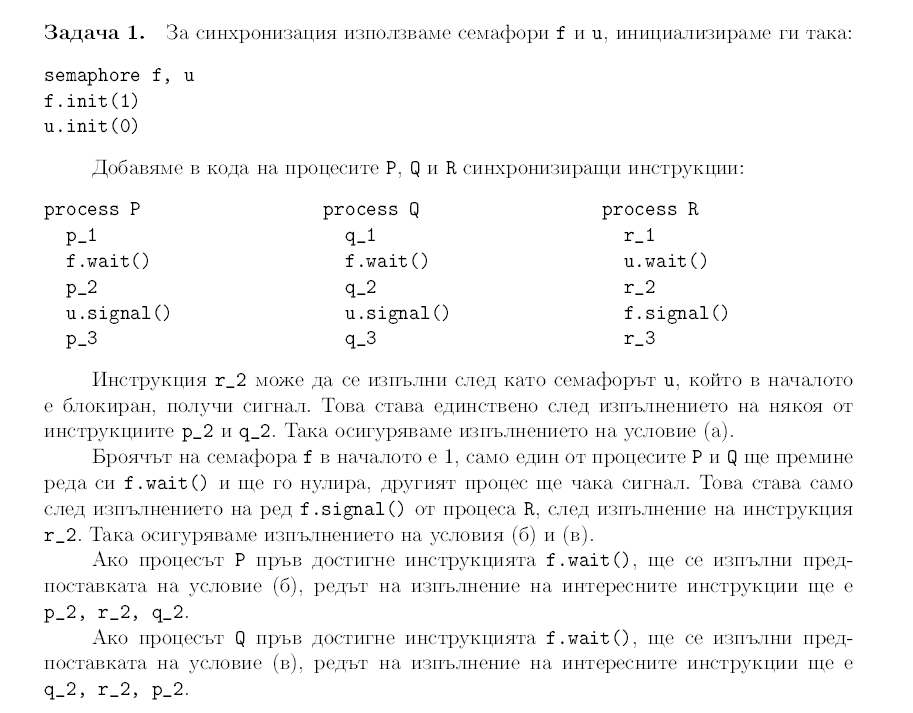
****

****

****

****

****

****