Задачи от лекции по операционни системи

Авторът на този файл не гарантира за достоверността му

1. Два процеса p и q. Изпълняват се паралелно. Искаме p2<q2 (p2 да се изпълни преди q2)
   s.init(0)

p	**q**
p1	q1
	s.wait()
p2	q2
s.signal()	
p3	q3

Обяснение:

1. **Инициализация:**
   Семафорът **s** се инициализира със стойност 0. Това означава, че ако процес q достигне **s.wait()**, той ще бъде блокиран, защото стойността на семафора е 0.
2. **Изпълнение на процес p:**
   - p изпълнява първо p1.
   - След това изпълнява p2. Това е критичният участък, който трябва да се изпълни преди q2.
   - Веднага след завършване на p2, процесът p извиква **s.signal()**, като увеличава стойността на семафора от 0 на 1. Това действие уведомява, че p2 е завършил.
3. **Изпълнение на процес q:**
   - q започва с изпълнението на q1.
   - При достигане на **s.wait()**, q се опитва да намали стойността на семафора. Ако все още p не е завършил p2 (и следователно **s.signal()** не е извикан), стойността ще бъде 0 и q ще блокира, докато семафорът не бъде сигнализиран.
   - Когато p завърши p2 и извика **s.signal()**, стойността на семафора се увеличава и q може да премине от wait() към изпълнението на q2.
   - След това q продължава с изпълнението на q3.

2. Два процеса p и q. Изпълняват се паралелно. Искаме p2<q2 за всеки процес q
   s.init(0)

p	**q_a**	**q_b**	**q_c**	.....
p1	q1	q1	...	
	s.wait()	s.wait()		
	s.signal()	s.signal()		
p2	q2	g2	...	
s.signal()				
p3	q3	q3	...	

3. Mutex - p1..pk и q1...qk са критични секции
   Целта е да се гарантира, че никога няма да има едновременен достъп до критичната секция – в даден момент само един от процесите (p или q) може да изпълнява своята критична секция. За тази цел ще използваме **mutex** (взаимно изключване), който се инициализира със стойност 1. За да осигурим взаимно изключение, всеки процес трябва да извика операцията s.wait() (заключване на mutex-а) преди да влезе в своята критична секция и след приключване да извика s.signal() (освобождаване на mutex-а).
   s.init(1)

p	**q**
..	..
..	..
s.wait()	s.wait()
p1	q1
p2	q2
...	..
pk	qk
s.signal()	s.signal()
..	..
..	..

4. Rendezvous – a1<b2 и b1<a2
   sA.init(0)
   sB.init(0)

A	**B**
a1	b1
sA.signal()	sB.signal()
sB.wait()	sA.wait()
a2	b2

   Ако имаме следното пак работи, но не толкова ефективно
   sA.init(0)
   sB.init(0)

A	**B**
a1	b1
sA.signal()	sA.wait()
sB.wait()	sB.signal()
a2	b2

   Ако имаме следното се получава deadlock
   sA.init(0)
   sB.init(0)

A	**B**
a1	b1
sB.wait()	sA.wait()
sA.signal()	sB.signal()
a2	b2

Така как работи:
sA.init(1)
sB.init(0)

A	B
a1	b1
sB.wait()	sA.wait()
sA.signal()	sB.signal()
a2	b2

5. Barrier

   В паралелното програмиране **бариера** (barrier) е механизъм за синхронизация, който гарантира, че всички нишки (или процеси) достигат определена точка в изпълнението си, преди която и да е от тях да продължи напред. Това е полезно, когато е необходимо всички нишки да завършат дадена фаза, преди да преминат към следващата. barrier – семафор (инициализиран с 0), който блокира нишките, докато всички не достигнат бариерат

   n = the number of threads
   count = 0
   mutex = Semaphore (1)
   barrier = Semaphore (0)

   Решение:
   rendezvous
   mutex . wait ()
   count = count + 1
   mutex . signal ()
   if count == n: barrier . signal ()
   barrier . wait ()
   barrier . signal ()
   critical point

   Обяснение:
   rendezvous: Представлява частта от кода, която всяка нишка изпълнява преди достигане на бариерата.
   mutex.wait() / mutex.signal(): Осигуряват взаимно изключване при увеличаване на count.
   if count == n: barrier.signal(): Когато последната нишка достигне бариерата (count == n), тя сигнализира семафора barrier, позволявайки на една нишка да премине.
   barrier.wait() / barrier.signal(): Всяка нишка изчаква на barrier.wait(). След като премине, тя сигнализира barrier.signal(), позволявайки на следващата нишка да премине. Това гарантира, че всички нишки ще преминат бариерата една по една, след като всички са я достигнали.
   critical point: Кодът, който се изпълнява след бариерата.

   Грешно:
   rendezvous
   mutex . wait ()
   count = count + 1

```
mutex . signal ()
 if count == n: barrier . signal ()
barrier . wait ()
critical point
```

Проблем:
В тази реализация само една нишка ще бъде освободена от barrier.wait(), след като barrier.signal() бъде извикан веднъж. Останалите нишки ще останат блокирани, тъй като няма допълнителни barrier.signal() извиквания, които да ги освободят.

6. Reusable barrier
Тази структура осигурява **повторна използваемост** на бариерата, позволявайки на нишките да преминават през нея многократно без риск от състезания или блокиране.
За да се създаде **повторно използваема бариера**, се използват два семафора:
turnstile – инициализиран с 0, блокира нишките, докато всички не достигнат бариерата.
turnstile2 – инициализиран с 1, блокира нишките след преминаване през първия турникет, докато всички не напуснат критичната секция.

```
turnstile = Semaphore (0)
turnstile2 = Semaphore (1)
mutex = Semaphore (1)
```

Работи:

```
# rendezvous
mutex . wait ()
        count += 1
        if count == n:
                turnstile2 . wait () # lock the second
                turnstile . signal () # unlock the first
mutex . signal ()
turnstile . wait () # first turnstile
turnstile . signal ()
# critical point
mutex . wait ()
        count -= 1
        if count == 0:
                turnstile . wait () # lock the first
                turnstile2 . signal () # unlock the second
mutex . signal ()
turnstile2 . wait () # second turnstile
turnstile2 . signal ()
```

Обяснение:
Първи турникет (turnstile): Блокира нишките, докато всички не достигнат бариерата. Последната нишка, която пристига, отключва този турникет, позволявайки на всички нишки да преминат към критичната секция.

Втори турникет (turnstile2): След критичната секция, нишките преминават през втория турникет. Последната нишка, която напуска, заключва първия турникет и отключва втория, подготвяйки бариерата за следващото използване.

Грешно1:
rendezvous
mutex . wait ()
    count += 1
mutex . signal ()
if count == n: turnstile . signal ()
turnstile . wait ()
turnstile . signal ()
critical point
mutex . wait ()
    count -= 1
mutex . signal ()
if count == 0 : turnstile . wait ()

Проблем:
Ако n− 1-вата нишка бъде прекъсната в този момент и след това n-тата нишка премине през мутекса, и двете нишки ще установят, че count==n и двете нишки ще сигнализират на турникета. Всъщност дори е възможно всички нишки да сигнализират за турникета.

Грешно2:
rendezvous
mutex . wait ()
    count += 1
    if count == n: turnstile . signal ()
mutex . signal ()
turnstile . wait ()
turnstile . signal ()
critical point
mutex . wait ()
    count -= 1
    if count == 0:turnstile . wait ()
mutex . signal ()

Проблем:
Тази реализация използва само един турникет и не осигурява повторна използваемост на бариерата. След първото преминаване, турникетът остава отворен, което може да доведе до състезания и неправилна синхронизация при следващи преминавания.

7. Queue
   В паралелното програмиране **проблемът с опашката (Queue Problem)** е класически пример за синхронизация между два типа нишки — *лидери* и *последователи* — които трябва да се сдвоят, за да изпълнят съвместна задача, например функцията dance(). Целта е да се гарантира, че всяка двойка

се състои от един лидер и един последовател, и че няма повече от една активна двойка по едно и също време.
leaderQueue е опашката, на която чакат лидерите, а followerQueue е опашката, на която чакат последователите.

leaders = followers = 0
mutex = Semaphore (1)
leaderQueue = Semaphore (0)
followerQueue = Semaphore (0)
rendezvous = Semaphore (0)

Queue solution (leaders)

```
mutex . wait ()
if followers > 0:
        followers --
        followerQueue . signal ()
else :
        leaders ++
        mutex . signal ()
        leaderQueue . wait ()
dance ()
rendezvous . wait ()
mutex . signal ()
```

Queue solution (followers)

```
mutex . wait ()
if leaders > 0:
        leaders --
        leaderQueue . signal ()
else :
        followers ++
        mutex . signal ()
        followerQueue . wait ()
dance ()
rendezvous . signal ()
```

8. Producer-consumer problem
   Проблемът „производител-потребител“ (Producer-Consumer Problem) е класически пример за синхронизация в многозадачни системи. Той описва ситуация, в която един или повече производители генерират данни и ги поставят в споделен буфер, докато един или повече потребители извличат тези данни за обработка.
   Основното предизвикателство е да се гарантира, че:

   - Производителите не добавят данни, когато буферът е пълен.
   - Потребителите не извличат данни, когато буферът е празен.
   - Достъпът до буфера е синхронизиран, за да се избегнат състезателни условия (race conditions).

mutex = Semaphore (1)
items = Semaphore (0) № # брои броя на наличните елементи в буфера
local event

Producer solution
event = waitForEvent ()
mutex . wait ()
buffer . add (event)
items . signal ()
mutex . signal ()

Consumer solution
items . wait ()
mutex . wait ()
event = buffer . get ()
mutex . signal ()
event . process ()

9. Readers-writers problem
Читатели (Readers): Процеси, които само четат от ресурса.
Писатели (Writers): Процеси, които пишат (модифицират) ресурса.
Цел: Позволяване на множество читатели да четат едновременно, но осигуряване на изключителен достъп за писателите, така че нито един друг процес (нито читател, нито писател) да не използва ресурса по време на запис.

mutex: Бинарен семафор (инициализиран с 1), който осигурява взаимно изключване при достъп до брояча на читателите.
roomEmpty: Бинарен семафор (инициализиран с 1), който гарантира, че ресурсът е свободен (няма активни читатели или писатели).
readers: Целочислена променлива, която брои броя на активните читатели.

int readers = 0
mutex = Semaphore (1)
roomEmpty = Semaphore (1)

Writers solution
roomEmpty . wait ()
critical section for writers
roomEmpty . signal ()

Readers solution
mutex . wait ()
readers += 1
if readers == 1:

```
        roomEmpty . wait ()      # first in locks
mutex . signal ()
# critical section for readers
mutex . wait ()
readers -= 1
if readers == 0:
        roomEmpty . signal ()            # last out unlocks
mutex . signal ()
```

Тази реализация дава приоритет на читателите, тъй като новите читатели могат да влязат, дори ако има чакащи писатели. Това може да доведе до гладуване на писателите (writer starvation), ако постоянно пристигат нови читатели. За да се избегне това, съществуват други варианти на решението, които дават приоритет на писателите или осигуряват справедлив достъп.