Теория по Операционни системи
(Материалите са комбинирани от интернет и лекциии)

49. 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 p1 да се изпълни преди q2, а q2 да се изпълни преди p3

Решение:
За двата момента на синхронизация ще използваме два семафора - u и t, като ги инициализираме с
блокиращо начално състояние:
semaphore u, t
u.init(0)
t.init(0)

След това добавяме в кода на процесите P и Q синхронизиращи функции:

```
process P    process Q
  p_1          q_1
u.signal()   u.wait()
  p_2          q_2
t.wait()     t.signal()
  p_3          q_3
```

Инструкцията q_2 ще се изпълни, след като броячът на семафора u стане положителен. Това се случва след изпълнението на ред u.signal(), който следва след инструкцията p_1.
Инструкцията p_3 ще се изпълни, след като броячът на семафора t стане положителен. Това се случва след изпълнението на ред t.signal(), който следва след инструкцията q_2.

50. Опишете накратко основните процедури и структури данни, необходими за реализация на семафор.
Каква е разликата между слаб и силен семафор?
Опишете максимално несправедлива ситуация, която може да се получи в избирателна секция, ако на входа на секцията пазач-член на изборната комисия пуска гласоподавателите вътре така:
(1) във всеки момент в секцията може да има най-много двама гласоподаватели
(2) пазачът работи като слаб семафор

Решение:
Структурите данни, които са необходими за реализация на семафор, са:
-брояч (cnt), в който се пази броят на процесите, които могат да бъдат допуснати до ресурса, който е охраняван от семафора
-контейнер Q, в който се пази информация кои процеси чакат достъп до ресурса

Процедурите, които са необходими за реализация на семафор, са:

-Конструктор Init(c0:integer), който задава начална стойност на брояча (cnt). Контейнерът Q се инициализира да е празен
-Метод Wait(), който се ползва при опит за достъп до ресурса (заемане на ресурса). Броячът се намалява с единица и ако стане отрицателен, то процесът, който извиква Wait(), се блокира, а номерът му се вкарва в контейнера Q
-Метод Signal(), който се ползва при завършване на достъпа до ресурса (освобождаване на ресурса). Броячът се увеличава с единица и ако контейнерът Q не е празен, един от процесите в него се вади и се активира

Семафорът е силен, когато контейнерът Q е реализиран като обикновена опашка - винаги активираме процеса, който е блокиран най-рано.
Семафорът е слаб, когато контейнерът Q не е реализиран като обикновена опашка - при изпълнение на Signal() се активира процес, който може да не е първи в списъка на чакащите

Ако пазачът на входа на избирателната секция действа като слаб семафор, може да се получи следната неприятна ситуация:
Първите двама гласоподаватели влизат в секцията, пристига трети гласоподавател, който не е познат на пазача и съответно изчаква своя ред. След него започват да пристигат приятели на пазача и той ги пуска с предимство. Може да се получи така, че третият гласоподавател да чака цял ден и да гласува последен.
Подобна несправедлива ситуация при достъп до ресурс се нарича starvation (гладуване).

51. 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 инструкция p1 да се изпълни преди q2 и r2.
Забележка: Решения на задачата с повече от един семафор носят не повече от 20 точки.

Решение:
За момента на синхронизация ще използваме един семафор - u, като го инициализираме с блокиращо начално състояние:
semaphore u
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
u.signal()	u.wait()	u.wait()
p_2	u.signal()	u.signal()
p_3	q_2	r_2
	q_3	r_3

По условие е дадено, че инструкцията p_1 трябва да се изпълни преди инструкциите q_2 и r_2. Това може да се осигури чрез въведения от нас семафор u. Ние пускаме инструкциите p_1, q_1 и r_1 да се изпълняват, без да знаем реда, в който те ще завършат. Ако q_1 и r_1 завършат преди p_1, не трябва да се изпълняват инструкциите q_2 и r_2. Поради тази

причина след инструкции q_1 и r_1 слагаме редовете u.wait(), които ни гарантират, че инструкциите под q_1 и r_1 няма да се изпълнят, докато семафорът u не получи сигнал. Ние искаме останалите инструкции да се изпълнят, след като завърши инструкция p_1. Поради тази причина след инструкция p_1 записваме u.signal(), който уведомява семафора, че е освободен ресурс. Първоначално броячът на семафора е 0, което означава, че след като се изпълни u.signal(), неговият брояч ще се увеличи с 1. Това означава, че само един процес може да премине през семафора. Да предположим, че процес Q преминава през инструкцията си u.wait() преди процес R. Тогава броячът на семафора u ще се намали с 1 и отново ще стане 0, което означава, че процесът R няма да може да мине през инструкцията си u.wait(). Поради тази причина непосредствено след инструкцията u.wait() в процеса Q записваме инструкцията u.signal(), която сигнализира семафора и увеличава брояча с 1, за да може да позволи и на процес R на мине през своята инструкция u.wait(). По този начин се осигурява изпълнението на двете инструкции q_2 и r_2. Аналогично е и ако процес R премине през инструкцията си u.wait() преди процес Q.

52. Преди стартиране на процеси P и Q са инициализирани два семафора и брояч:
semaphore e, m
e.init(1); m.init(1)
int cnt = 0

Паралелно работещи няколко копия на всеки от процесите 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? -не
б) Могат ли едновременно да се изпълняват няколко инструкции p_section? -да
в) Могат ли едновременно да се изпълняват няколко инструкции q_section? -не
г) Има ли условие за deadlock или starvation за някой от процесите? -да за starvation при Q

Упътване:
-Ще казваме, че P е 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 бъда приспани в опашките на двата семафора
-Покажете, че в опашката на семафора е 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:
Забелязваме, че семафорът m се ползва само от P и то 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. Отговорът на 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 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 mutex m ще 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 а) е НЕ

В описаната схема няма условия за deadlock. Q не може да инициира deadlock, защото използва 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 започне 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 време.

53. а) Няколко копия на процеса P изпълняват поредица от три инструкции:

```
process P
  p_1
  p_2
  p_3
```

Осигурете чрез семафор синхронизация на копията така, че най-много един процес да изпълнява инструкция p2 във всеки един момент

б) Опишете разликата при реализация на слаб и силен семафор
в) Възможно ли е в зависимост от начина на реализация на семафора в подусловие а) да настъпят условия за deadlock или starvation? Ако да, опишете сценарий за поява на неприятната ситуация

Решение:
а) Преди стартирането на процеса P инициализираме семафор с отблокирано начално състояние:
semaphore m
m.init(1)

```
process P
   p_1
 m.wait()
   p_2
 m.signal()
   p_3
```

б) Семафорът е силен, когато контейнерът Q е реализиран като обикновена опашка - редът има значение (първи влязъл, първи излязъл)
Семафорът е слаб, когато контейнерът Q е реализиран като приоритетна опашка - редът не е от значение (процесите не преминават според реда си)
При приоритетната опашка е важен така наречения "приоритет". В случая, ако в контейнера е пристигнал даден процес, който не се счита за значителен, а след него пристигнат процеси, които се считат за значителни, то тези, които са значителни, ще бъдат допуснати до ресурса преди този, който е незначителен. По този начин може да се стигне до така наречения starvation (гладуване) - един процес да чака неопределено дълго време за даден ресурс, защото преди него се допускат други процеси, които имат по-висок приоритет.

в) Реализацията на семафора в подусловие а) не може да доведе до deadlock, защото до инструкцията p_2 се допуска само едно копие на процеса P, а останалите го изчакват. След като копието приключи изпълнението на инструкция p_2, то сигнализира за освобождаване на ресурса и веднага след него е допуснато друго копие до инструкцията p_2. Копията не се блокират взаимно, което означава, че няма условие за deadlock.

Реализацията на семафора в подусловие а) може да доведе до starvation, ако тя е осъществена не с обикновена, а с приоритетна опашка. Нека си представим следната ситуация:
Пристига копие1 на процеса P. Това копие се допуска до инструкция p_2, тъй като семафорът m е с начално отблокирано състояние. След като копие1 се допусне до инструкция p_2, копията след него трябва да изчакат, докато той не сигнализира, че освобождава ресурса. Докато копие1 изпълнява инструкция p_2, в опашката пристига копие2, което изчаква копие1. След неопределено време пристига копие3, което има приоритет пред копие2.
След като копие1 приключи изпълнението на инструкция p_2, сигнализира за освобождаване на ресурса. След него би трябвало да бъде допуснато копие2, тъй като то е преди всички останали. Но ако семафорът е слаб (контейнерът е инициализиран чрез приоритетна опашка), то тогава пред копие2 ще влезе копие3, защото има приоритет пред копие2. По този начин копие2 трябва да изчака копие3 да приключи с изпълнението на инструкция p_2. Докато копие2 изчаква реда си, то в опашката могат да пристигнат още копия, които да са с приоритет пред копие2. Тогава всички те ще минат преди копие2 и по този начин се получава

ситуация на starvation - копие2 е пристигнало непосредствено след копие1 и изчаква за приключване на неговото изпълнение, но в опашката пристигат копия, които са с приоритет пред копие2 и те се допускат преди него, независимо от това, че копие2 изчаква преди тях.

54. 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 p1 да се изпълни преди q2, а q1 да се изпълни преди p2.

За синхронизация между процесите ще използваме два семафора - u и t. Те ще бъдат с начално блокиращо състояние.

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

Това също е вярно

```
process P    process Q
 p_1          q_1
u.signal()   u.wait()
t.wait()     t.signal()
 p_2          q_2
```

Това е по-добро решение

```
process P    processQ
 p_1          q_1
u.signal()   t.signal()
t.wait()     u.wait()
 p_2          q_2
```

Да предположим, че инструкция q1 приключва преди инструкция p1. От условието се иска q2 да се изпълни след p1. Това означава, че след инструкция q1 е необходимо да добавим инструкция, която да уведоми процеса да изчака -
за тази цел добавяме инструкция u.wait() след инструкция q1. След като инструкция p1 приключи изпълнението си,
тя трябва да сигнализира на семафора u за освобождаване на ресурса. За тази цел след инструкция p1 добавяме
инструкция u.signal(), която позволява на процес Q да премине към изпълнение на инструкция q2.

Семафорът t ни помага в обратната ситуация - ако инструкция p1 приключи преди инструкция q1. От условието се иска p2 да се изпълни след q1. Ако инструкция p1 се изпълни преди инструкция q2, то тя ще сигнализира на семафора u за освобождаване на ресурса, но непосредствено след това сигнализиране (след инструкция u.signal()) е необходимо да се изчака приключване на изпълнението на инструкция q1. За тази цел добавяме инструкция t.wait(), която информира процес P да изчака, докато не му се подаде сигнал. След като инструкция q1 приключи изпълнението си, тя не се спира при инструкция u.wait(), защото инструкция p1 вече е сигнализирала на семафора u и вече може да бъде допуснат процес през него. Процесът P изчаква сигнал при инструкция t.wait(). Поради тази причина добавяме инструкция t.signal() непосредствено след u.wait(). По този начин процесът Q ще сигнализира на семафора t за освобождаване на ресурса и ще може да продължи, като изпълнени и инструкция q2, а едновременно с това семафорът t ще може да допусне още един процес и по този начин процесът P може да продължи, като изпълнени и инструкция p2.

55. 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 изпълняват в следния времеви ред: p1,q1,p2,q2,p3,q3

Решение:
За синхронизация между процесите ще използваме два семафора u, t, които ще бъдат инициализирани с начално блокиращо състояние.

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

```
process P        process Q
 p_1               u.wait()
u.signal()          q_1
t.wait()           t.signal()
 p_2               u.wait()
u.signal()          q_2
t.wait()           t.signal()
 p_3               u.wait()
u.signal()          q_3
```

Тъй като инструкция p1 трябва да се изпълни преди инструкция q1, то преди инструкция q1 записваме инструкция u.wait(). След като инструкция p1 завърши, тя сигнализира на семафора u и тогава процес Q може да започне с изпълнението на инструкция q1. По условие обаче имаме, че инструкция q1 трябва да се изпълни преди инструкция p2. За тази цел, непосредствено след инструкция u.signal() при процеса P, записваме инструкция t.wait(), която ни гарантира, че процесът P ще изчака изпълнението на инструкция q1 на процеса Q. След като инструкция q1 приключи с изпълнението си, то тя сигнализира на семафора t и процес P може да продължи нататък, като започне изпълнението на инструкция p2. По условие обаче имаме, че инструкция p2 трябва да се изпълни преди инструкция q2.
За тази цел трябва процес Q да изчака инструкция p2 от процеса P. Поради тази причина, веднага след инструкция t.signal() на процеса Q, добавяме инструкция u.wait(). След като инструкция p2 приключи с изпълнението си, тя изпраща сигнал на семафора u и процесът Q може да започне изпълнението на инструкция q2. По условие имаме, че инструкция p3 на процеса P трябва да се изпълни след инструкция q2 на процеса Q. Поради тази причина добавяме инструкция t.wait() след u.signal(). След като инструкция q2 на процеса Q приключи с изпълнението си, то тогава се сигнализира семафорът t и процес P може да започне изпълнението на инструкция p3. Но по условие имаме, че инструкция q3 трябва да се изпълни след инструкция p3, което означава, че е необходимо да добавим инструкция u.wait() след t.signal(), за да накараме процес Q да изчака. След като инструкция p3 приключи с изпълнението си, семафорът u се сигнализира и тогава процес Q може да продължи нататък с изпълнение на инструкция q3.

56. Process States
running - процеси, които активно се нуждаят от процесор/ядро (имат работа за вършене); ако не стигат процесорите в системата, някои от тях да изчакват и да се редуват помежду си; имат нужда от изчислителна мощност; това са процеси, които използват ядрото на процесора в момента

От гледна точка на ядрото са 2 типа: Running и Ready

running - Running - работещият процес се бори за компютърно време
|
Ready - процеси, които чакат да изчисляват (имат готовност), но нямат свободен процесор

sleeping - спящи процеси, които в момента нямат работа; не използват изчислителен ресурс; от гледна точка на потребителя те са работещи програми, но са стигнали до състояние, в което няма нужда да изчисляват нещо, докато не настъпят интересни събития в системата; влизат в комуникация с друг процес или устройство и очакват процесът или устройството да му подадат данни; той не смята; не използва изчислителен ресурс, не използва компютърно време

sleeping - I/O (очаква входно-изходна операция)
sleeping – time (очаква времеви момент – sleep command)
sleepin – signal (очаква сигнал от друг процес; родителят може да изчаква детето да извърши своята работа например)

stopped - не е интересен нито за потребителите, нито за системата
zombie - ненапълно спрян процес; процес, който е започнал спирането на процес, но не е завършило; Това състояние е необходимо, за да се даде възможност на процеса родител да попита ядрото за ресурсите, които са използвани от неговото дете (getrusage(2) е командата). Процесът родител уведомява ядрото, че е приключило с детето си, като извиква waitpid.

Пускането и спирането на процес са бавни събития и са многостъпкови.

Спирането се извършва или с команда exit, или с някой специален сигнал (kill). Когато се спира процес, той трябва да освободи ресурсите, които е заел - освобождава паметта, която заема, затваря приложения

Имаме процес, който работи - running. Когато извикваме командата fork(), ние правим негово копие. Създаденото копие е в състояние Ready. Убиването на процес може да стане само при Running процес. Когато един процес е убит от друг процес, тогава става по друг начин. Процес, който е Running, може да бъде превключен, когато той самият пожелае. Начин за приспиване на процес е с командата wait(), която се прилага върху семафор. Друг начин за приспиване на процес е с командата sleep(), когато искаме да забавим процес. Друг начин за приспиване на процес е да чакаме някой друг процес да свърши нещо.
От Sleeping към Ready може да се случи чрез сигнализиране на семафор, в случай, че е приспан от семафор.
От Sleeping към Ready - приспан от sleep, събужда се, когато настъпи този момент на свършване на времето.
Следи се дали е дошъл моментът на събуждане след командата sleep. Това става само когато ядрото започне да използва процесора, когато става смяна на контекста. Прекратява се работата на някакъв работещ процес и ядрото взима решение кой процес да пусне. При пекъсване на таймера ще се получи такава възможност. От Sleeping към Ready - тези, които

чакат някой процес да си смени състоянието, ще бъдат събудени, когато някой друг процес си изпълни командата exit.
Всички тези събития на диаграмата се извършват по желание на процесите

Когато има прекъсване на времето, се намесва ядрото, то взима решение кой процес да бъде пуснат.
От Running към Ready отново може да се случи с изтичане на времето (timeOut())
От Ready към Running е ключовият момент как работи операционната система. Когато процесорът се освободи, той временно се използва от ядрото, системно време се нарича. Ядрото трябва да вземе решение, след като е спрял един процес, довел го е до приспано състояние, кой да пусне. Трябва решението да е такова, че да не се допуска гладуване. Ако ядрото допуска само процеси с приоритет, може да настъпи гладуване. Трябва да се следи да се избягва това гладуване.

Нов процес се появява с изпълнение на командата fork() -> създаденият процес е в състояние Ready Процес, който в момента се изпълнява, може да бъде "убит" (прекъснат) -> командата exit(); ако един процес убива друг процес, по друг начин стават нещата

57. Процесът P създава тръба (pipe) с извикване на функцията pipe(int pipefd[2]) в ОС GNU/Linux.
а) Кои процеси не могат да ползват тръбата?
б) Опишете друг метод за изграждане на комуникационен канал, който дава възможност на произволни
процеси да изградят и ползват канала. Допълнително искаме новоизграденият канал да е достъпен само
за процесите, които са го създали.

Упътване: Прочетете man-страницата за функцията pipe().

Обяснение:
а) Тръбата е достъпна само чрез файловите дескриптори pipefd[0] и pipefd[1]. Те са видими само за процеса P и неговите наследници. Процесите, които не са наследници на P, не могат да ползват тръбата.
б) За да бъде използван от произволен процес в изчислителната среда, комуникационният канал трябва да бъде видим (адресуем, именуван). В повечето UNIX системи има възможност за създаване на именувана тръба (FIFO), тя обаче се създава от един процес и е достъпна за всички останали, тоест нарушава допълнителното условие на т. б).
Друг вариант е един процес да създаде именуван обект, който да послужи като адрес при изграждането на връзка с друг процес. Използваната абстракция се нарича socket. Сокетът се дефинира като единия край на комуникационния канал. Един процес, наричан обичайно сървър, изпълнява следната поредица:

```
sfd=socket(domain, type, protocol); // създава socket
bind(sfd, &my_addr, addreln); // присвоява име на socket-а
listen(sfd, backlog); // започва приемане на заявки за връзки
cfd=accept(sfd, &peer_addr, addrlen); // приема заявка за изграждане на връзка
```

Друг процес, наричан обичайно клиент, изпълнява следната поредица:

```
fd=socket(domain, type, protocol); // създава socket
connect(fd, &server_addr, addrlen); // подава заявка за изграждане на връзка
```

Сървърът използва сокета sfd и му дава име чрез bind. Използването listen активира процеса на изграждане на връзки. Клиентът създава сокета fd, без да е нужно да го именува. Извикването connect е заявка за изграждане на връзка към именувания сокет sfd. Сървърът приема заявката на клиента чрез accept. Изградената връзка е между файловите дескриптори cfd на сървъра и fd на клиента. Те ги ползват за обмен на информация. Файловият дескриптор sfd на сървъра продължава да приема нови заявки от клиенти. Благодарение на присвоеното му име sfd дава възможност на другите процеси да се свържат със сървъра. Името на sfd определя какви клиенти могат да ползват сървъра. Ако то е име в интернет (IP адрес, порт), всички процеси, изпълнявани на компютри и имащи достъп до интернет, могат да се свържат със сървъра.

58. 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 работещите копия така, че:
а) В 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:
За синхронизация между процесите ще използваме два семафора - u и t, като семафор u ще бъде инициализиран
с отблокирано начално състояние, а семафор t ще бъде инициализиран с блокиращо начално състояние:

```
semaphore u, t
u.init(1)
t.wait(0)
```

```
process P   process Q
u.wait()      t.wait()
 p_1           q_1
 p_2           q_2
t.signal()   u.signal()
```

Семафорът u е инициализиран с отблокирано начално състояние 1, което означава, че може да допусне само едно копие на процес да има достъп до споделения ресурс. Останалите копия на процеса P изчакват - по този начин се изпълнява условието да се изпълни първоначално едно копие на процеса P. Копията на процеса Q
изчакват, тъй като семафорът t е инициализиран с блокиращо начално състояние. След като копието на процеса P изпълни инструкциите p_1 и p_2, сигнализира на семафора t за освобождаване на ресурс и тогава вече копие на процеса Q може да премине към изпълнение на инструкциите q_1 и q_2 -> копията на процеса P все още изчакват, защото не е подаден сигнал и копията на процеса Q изчакват, защото през семафора t може да премине само едно-единствено копие. След като копие от процеса Q приключи изпълнението на инструкциите q_1 и q_2, копието сигнализира на семафора u и вече копие на процеса P може да премине

към изпълнение на инструкции p_1 и p_2. По този начин се изпълнява условието работещите копия да се редуват във времето - след изпълнение на копие на P да следва изпълнение на копие на Q и обратно. В произволен момент от времето работи най-много едно от копията -> в началото през семафора u се допуска едно-единствено копие на процеса P, останалите копия на този процес изчакват в опашката на семафора u. Копията в опашката на семафора t могат да бъдат само от процеса Q. В началото семафорът t не допуска нито едно копие. След завършване на изпълнение на копие P се подава сигнал на семафора t, който позволява на копие на процес Q да започне с изпълнението на инструкциите. Копията на процеса P все още изчакват в опашката, докато някое копие от процеса Q не подаде сигнал на семафора u. По този начин се удовлетворява и условието в произволен момент от времето да работи най-много едно от копията -> не може да има едновременно работещи копия на P и на Q; не може да има едновременно работещи копия на P и не може да има едновременно работещи копия на Q.

59. 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 изпълняват едновременно следните изисквания:
- инструкция p1 да се изпълни преди q2 и r2
- инструкция r2 да се изпълни преди p3

Забележка: 2 семафора носят 30 точки, повече семафори носят 20 точки

Решение:
За синхронизация между процесите ще използваме два семафора - u и t, като ще ги инициализираме
с блокиращо начално състояние:

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

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

От условието се иска инструкция p1 да се изпълни преди инструкции q2 и r2. Поради тази причина, след инструкции q1 и r1, записваме инструкции u.wait(), които ни гарантират, че няма да се изпълнят по-долните инструкции, преди да е подаден сигнал на семафора u. След като p1 завърши изпълнението си, той трябва да сигнализира на семафора u. Поради тази причина под инструкция p1 записваме инструкция u.signal(). По този начин само един от процесите Q и R ще може да продължи с изпълнението си. Да предположим, че процес Q минава първи през инструкция u.wait(). Тогава той ще може да продължи с изпълнението на инструкции q_2 и q_3, но процесът R ще остане блокиран, защото вече процес Q е преминал

през семафора u. Именно поради тази причина под инструкция u.wait() е необходимо да добавим инструкция u.signal(), която да подаде сигнал на семафора u,
който от своя страна на позволи на процес R да продължи с изпълнението на инструкциите си. Аналогична е и другата ситуация - ако процес R преминава първи през инструкция u.wait() - поради тази причина и под u.wait() на процес R поставяме и инструкция u.signal(). По условие имаме още, че инструкция r2 трябва да се изпълни преди p3. Това означава, че след като се изпълни инструкция p2, е необходимо непосредствено под нея да се добави инструкция t.wait(). Тук не може да използваме семафора u - тоест да напишем u.wait() вместо t.wait(), защото броячът на семафора ще е 1 - p1 сигнализира 1 път на семафора u -> тогава през него преминава например процес Q. В този момент броячът на семафора u отново става 0, но процес Q преминава през инструкция u.signal(), което означава, че броячът на семафора u отново става 1. След това процес R преминава през семафора u, което означава, че броячът му отново е 0. Обаче процесът R преминава непосредствено след това през инструкция u.signal() и по този начин увеличава брояча на семафора u с 1. Тоест, ако под инструкция p2 запишем инструкция u.wait(), то процесът P ще може да премине през семафора u и ще може да продължи надолу с изпълнението на инструкция p3, а в този момент е възможно инструкция r2 все още да не е изпълнена и по този начин се нарушава изискването от условието. Поради тази причина използваме другия семафор t, който има начално блокиращо състояние. След като инструкция p2 приключи с изпълнението си, процесът P изчаква семафор t да го допусне към останалите инструкции. Това ще се случи, когато инструкция r2 се изпълни. Под нея записваме инструкция t.signal(), за да осъществим тази цел. По този начин гарантираме, че инструкция p3 ще се изпълни след изпълнението на инструкция r2.

60. 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:
- инструкция p1 да се изпълни преди q2
- инструкция q1 да се изпълни преди r2
- инструкция r1 да се изпълни преди p2
- инструкция r3 да се изпълни след p2 и q2

Решение:
За синхронизация между отделните процеси ще използваме 4 семафора - u, t, f и m, като ще ги инициализираме с блокиращо начално състояние:
semaphore u, t, f, m
u.init(0)
t.init(0)
f.init(0
m.init(0)

process P	process Q	process R
p_1	q_1	r_1
u.signal()	u.wait()	f.wait()
t.wait()	f.signal()	t.signal()
t.signal()	t.wait()	r_2

p_2 t.signal() f.wait()
u.signal() q_2 r_3
m.wait() u.wait()
p_3 m.signal()
f.signal()

61. При споделено ползване на памет от няколко процеса е възможно да настъпи надпревара за ресурси (race_condition)

а) Дефинирайте понятието race condition
б) Възможно ли е да настъпи race condition в еднопроцесорна система? Ако да, при какви условия?
в) Какви инструменти ползваме, за да избегнем race condition?

а) race condition възниква, когато два или повече процеса използват общ ресурс, който едновременно се опитват да достъпят, като се стремят да изпълнят операция по промяна на данните в него. Race condition е специално условие, което може да възникне в критична секция. Критичната секция е парче код, което се изпълнява от множество нишки, като последователността от изпълнение може да доведе до разлика в крайния резултат на критичната секция. Когато резултатът от множество нишки/процеси, които изпълняват критичната секция, се различава в зависимост от последователността на изпълнение, се казва, че критичната секция съдържа условие за race condition. Понятието идва от там, че нишките се "състезават" помежду си в критичната секция, като техните резултати влияят на нейния краен резултат. Нека си представим следната ситуация:

Имаме глобална променлива x с начална стойност 0. Представете си, че два процеса имат за задача да увеличат нейната стойност с 1. Ако двата процеса протекат последователно, ще се получи следната ситуация:

Process1	Process2		Value
			0
read value		<-	0
increase			0
write back		->	1
	read value	<-	1
	increase		1
	write back	->	2

Крайният резултат от изпълнението на двата процеса е 2.

Ако обаче двата процеса протекат едновременно, без да има последователност между тях, то тогава крайният резултат би бил съвсем различен и нереален. Ще се получи следната ситуация:

Process1	Process2		Value
			0
read value		<-	0
	read value	<-	0
incerase			0
	increase		0
write back		->	1
	write back	->	1

В този случай крайният резултат е 1, а не 2, както би трябвало да бъде. Едновременно и двата процеса прочитат началната стойност на променливата x -> в началото тя е 0, което означава, че и процес1, и процес2 ще прочетат стойност 0. След това процес1 увеличава взетата стойност -> взетата стойност от този процес е 0, което означава, че ще я увеличи с 1 (вече взетата променлива x е с увеличена стойност 1). В същото време процес2 увеличава взетата стойност -> взетата стойност от този процес също е 0, което означава, че ще я увеличи с 1 и вече взетата променлива x ще е с увеличена стойност 1. По този начин двата процеса не се изчакват помежду си и променят по едно и също време данните в общия ресурс, който за достъпили. Това се получи по този начин, защото двата процеса не са взаимно изключващи се. Взаимно изключващи се процеси са тези, които не могат да бъдат прекъснати, докато имат достъп до определен ресурс.

Пример от лекциите:
Нека си представим следната ситуация: Имаме опашка M от съобщения, в която имат право да пишат два процеса. Процесите са следните:

```
process P                   process Q
p1: M[l] -> write msgP      q1: M[l] -> write msgQ
p2: l=l+1                   q2: l=l+1
```

```
M: msgA msgB msgC
     |                  |
   first              last
```

Опашката е динамична структура, която може да бъде нарушавана временно за кратък период от време. Идеята на опашката M е да има информация за първия и последния елемент в нея - в случая първият елемент е някое съобщение, което трябва да бъде изпратено, а последният елемент е празен (свободен), като на негово място е позволено да се добави друго съобщение в опашката. Когато в опашката M на последното място l (last) първата нишка добави ново съобщение, опашката ще бъде временно нарушена – последният елемент в нея няма да е свободен, а ще съдържа информация за ново добавено съобщение. След като се изпълни и втората нишка (l=l+1), то тогава опашката Q ще се увеличи с още 1 място и последният елемент отново ще е свободен точно както трябва да бъде спрямо нашата имплементация.

Ако процесите P и Q протичат последователно, би трябвало да се получи следната ситуация:
Пристига съобщение msgP

```
process P
p1: M[l] -> write msgP
```

```
M: msgA msgB msgC msgP
     |                  |
   first              last
```

```
p2: l=l+1
```

```
M: msgA msgB msgC msgP
     |                      |
   first                  last
```

Пристига съобщение msgQ

process Q
q1: M[l] -> write msgQ

M: msgA msgB msgC msgP msgQ
| |
first last

q2: l=l+1

M: msgA msgB msgC msgP msgQ
| |
first last

По този начин и двете съобщения - msgP и msgQ са записани в опашката и чакат своя ред, за да бъдат изпратени

Ако двата процеса не протекат последователно, а едновременно, има опасност от следната ситуация:

M: msgA msgB msgC
| |
first last

Пристига съобщение msgP

process P
p1: M[l] -> write msgP

M: msgA msgB msgC msgP
| |
first last

p2: l=l+1

M: msgA msgB msgC msgQ
| |
first last

Пристига съобщение msgQ

process Q
q1: M[l] -> write msgQ

M: msgA msgB msgC msgQ
| |
first last

q2: l=l+1

M: msgA msgB msgC msgQ
| |
first last

Какво се случи?
Нишка p1 записа на последно място в опашката M съобщението msgP. Едновременно с това и нишка q1 записа съобщението msgQ на абсолютно същото място, на което p1 е записала съобщението msgP -> тоест съобщението msgP се изгуби и на негово място е записано съобщението msgQ. След това нишка p2 увеличава броя на елементите в опашката с 1, но в същия момент и нишка q2 увеличава броя на елементите в опашката с 1 -> тоест добавиха се в опашката M 2 нови свободни места, което нарушава нашата имплементация. Това е пример за race condition

б) Да, възможно е да настъпи race condition в еднопроцесорна система, защото race condition възниква, когато изходът зависи от последователността или времето на други неконтролируеми събития.

Нека си представим следния сценарий:

T1: Иска да добави запис на служител към даден файл
T2: Иска да изчисли средната заплата на "IT отдел" във фирмата
T3: Иска да отстрани служител, който е напуснал
T4: Иска да изчисли броя на служителите, които работят във всяко отделение

Ако всички нишки (T1, T2, T3, T4) изчакват на време=0 и се предават на един процесор, той ще реши коя нишка да премине първа, коя втора, коя трета и коя четвърта. Редът, в който нишките са приоритизирани, се различава в различните платформи, сценарии и така нататък. Нишки T2 и T4 е възможно да не дават постоянен резултат.

в) Инструменти за предотвратяване на race_condition
Трябва критичните секции да са атомарни инструкции - това е чисто хардуерно решение. Да се направят специални хардуерни инструкции, които цялото това нещо да го изпълнят като една елементарна команда и докато тя се изпълнява, данните да бъдат блокирани за други процеси/ядро/системи

Другото решение е чисто софтуерно - да се осигури така, че единият процес да приключи своята работа върху своята критична секция, след което другият процес да започне работа по своята критична секция - решение на Декер

Има и решение на Петерсон, но то е дълго и отнема време.

В съвременно време се използват смесени решения - с малко хардуерна помощ.

Докато един процес изпълнява критична секция, да се забрани на останалите процеси да работят по своята критична секция.

Хардуерни средства:
- за еднопроцесорна система - има команди за временна забрана на прекъсванията и команда за разрешаване на прекъсванията

disable event
p1: ...
p2: ...
enable event

В по-ранните версии на операционните системи се е използвал този метод, той е хардуерен метод, но ако инструкциите са много и дълги, то може да се окаже, че са блокирани много важни входни данни

В днешно време повечето са многопроцесорни и дори да сме забранили на едното ядро изпълняването на повече процеси, докато единият процес не приключи своето изпълнение, то на другото ядро няма забрана и процесите могат да си работят и няма блокиране

Друго решение е защита на ресурса - да има още един бит, който да е 0, ако структурата е свободна, и да е 1, ако е заета. По този начин процесите ще могат да гледат бита на структурата/ресурса и да знаят дали да влязат, или да изчакат. Тази защита се нарича lock. Но с нормални инструкции това не може да бъде направено, защото на ниво машинен език трябва заредим в регистъра бита, за да го променим, да го проверим дали е 0, ако е 0, да го направим 1 или пък ако е 1, да циклим, докато стане 0. Битът не е атомарен и представлява

критична секция и може отново да има race condition.

Специални инструкции: Testing set или atomic swap - размяна на две стойности между регистър на процесора и паметта, това са специални хардуерни инструменти

Решението се нарича взаимно изключване - да се блокират останалите процеси, докато един е в критичната си секция - разрешаваме само на един процес да си изпълни критичната секция и то от начало до край, а на останалите да им е забранено да изпълняват своя код, те трябва да чакат.

Алгоритмите на Декер и Петерсон не се използват в днешно време.

Два метода се ползват:
- един за еднопроцесорна система, просто е решението, в началото на критичната секция казваме, че забраняваме да има прекъсвания, а в края ги разрешаваме, процесът ще работи само по критичната си секция и никой друг процес няма да започне работа - единственият разумен хардуерен начин за работа в еднопроцесорна система.
- един за многопроцесорна система, отново има хардуерен проблем. Паралелно работещи процеси дори в малките многопроцесорни системи (например смартфоните). Когато е многопроцесорна система, методът с прекъсването не работи. Друг хардуерен метод е с ползване на допълнителен бит в структурата, в който да пазим информация дали важните данни в критичната секция се ползват от някой процес, или са свободни - тоест дали може някой процес да започне работа в критичната секция. Допълнителният бит се нарича lock и методът, който ползваме, се нарича spinlock.

Spinlock:
Трябва да имаме специални хардуерни инструменти, за да реализираме защита от race condition. Едното е да можем да блокираме прекъсвания, което за еднопроцесорна система е достатъчно - временно да ги спираме
Другото е spinlock механизма - да имаме специални атомарни инструкции, които да променят някакъв бит в паметта и да го прочетат едновременно в централния процесор и това действие да е една единна инструкция, която да не може да бъде прекъсната. Процесът на управление на една такава критична секция трябва да започне с процедура, която да вземе ресурса, и накрая да завършим с освобождаване.

spin_lock test.and.set(lock)

lock има стойност 0, ако структурата е свободна и никой не я ползва в момента
lock има стойност 1, ако структурата е заета и има кой да я ползва в момента

test.and.set чете този байт от паметта, записва го в някакъв регистър в паметта или го тества дали е 0, или е 1, и в същото време записва 1 в байта. Прочитането на байта в паметта и записването на 1 трябва да стане едновременно, тоест да стане атомарна единица. Ако я направим като двойка от инструкции, то тогава отново има условие за race condition. Поради тази причина трябва да е атомарна инструкция.

spin_lock R = test.and.set(lock)
  if R=1 goto spin_lock //Ако общият ресурс е зает в момента, отиваме отново да проверим горе дали е 0, или е 1
          |
Ако е 1, започва да върти целия този цикъл

В момента, в който стане 0, тогава няма да се влезе в този if

```
spin_lock R = test.and.set(lock)
  if R=1 goto spin_lock
  p1
  .
  .
  .
  pk
spin_lock.lock=0
```

Ако имаме процес1, който да изпълнява критичната секция, то тогава R постоянно ще бъде 1, защото общият ресурс ще е зает от процес1. В същото време, ако се появи процес2, който прочита R (R = test.and.set(lock)) и при проверката if R=1 се появи процес3, който прочита R, то тогава ще се получи ситуация, в която процес2 ще се върне при spin_lock, докато той е там, процес3 ще е при проверката if R=1 и така ще се редуват. В момента, когато процес1 приключи изпълнението на критичната си секци, той ще освободи структурата и тогава е възможно процес2 да е на проверката if R=1, а в този момент процес3 да е при R = test.and.set(lock), като в този случай R ще е 0 за процес3 и той няма да влезе в случая R=1, а ще започне с изпълнението на критичната секция, като по този начин процес2 отново ще цикли и е възможно никога да не влезе в критичната секция, ако по този начин се получи разминаване и с други процеси.

Би трябвало да се осигури защита, но възниква допълнителен проблем. Тази техника не е рекурсивна - докато изпълняваме тези редове, не трябва по никакъв начин да пуснем друга програма, която да извика spin_lock. Може да влезем в безкраен цикъл. Друго интересно нещо е таймерът - някакъв процес да може да бъде спрян периодично, за да се даде възможност и на друг процес. Ако възникне прекъсване от таймера, процесът ще се възстанови след определено време. Но може всички процеси да чакат този бит lock да се отключи и тогава всички те ще циклят. Ще има моменти, в които целият компютър ще спи и то за дълго време, за да се изредят всички чакащи процеси. Така че това решение не е идеално. За да го подобрим, трябва да забраним прекъсванията - не е хубаво да има такива прекъсвания и да има рекурсия Не трябва да се вика рекурсивно.

Да кажем, че сме забранили прекъсванията. Това се прави с инструкция: spin_lock disable_interrupt

```
spin_lock disable_interrupt
spin_lock R = test.and.set(lock)
  if R=1 goto spin_lock
  p1
  .
  .
  .
  pk
spin_lock.lock=0
enable.interrupt
```

Да видим какво ще се случи, ако едновременно искаме да влезе в критичната секция.

Важните места, на които трябва да се забранят прекъсванията, е самата критична секция, включително

командата spin_lock.lock=0 след критичната секция и в командата преди критичната секция, в която записваме 1 в spin_lock

```
spin_lock disable_interrupt
   R = test.and.set(lock)
   if R=0 goto critical.set
   enable.interrupt
   goto spin_lock
```

По този начин е по-добре решението

В съвременните операционни системи хардуерът е по-сложен и съответно тези решения са по-сложни

Интернет:
За да се предотврати race condition, е необходимо да се уверим, че критичната секция е изпълнена като атомнарна инструкция. Това означава, че след като един процес/нишка изпълнява критичната секция, никой друг процес/нишка не може да я изпълни, докато първият процес/нишка не напусне критичната секция.
Race condition може да бъде избегнат чрез правилна синхронизация на процеси/нишки в критични секции. Необходимо е да са изпълнени следните условия:
- Взаимно изключване - когато процес Pi се намира в критична секция, никой друг процес да 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, да 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

Общата структура на процеси, които притежават критична секция, е следната:

```
do {
  вход в критичната секция;
  ...
  критична секция;
  ...
  изход от критичната секция;
  ...
  некритична секция;
  ...
  ...
} while (...);
```

Пример:
Нека разгледаме два процеса P0 и P1. Нека с i обозначим номера на единия процес. Тогава номерът на другия процес ще бъде j=i-1.
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 P0 може да влезе в критичната си секция
Ако turn=1, то процес P1 може да влезе в критичната си секция

Примерна структура на процеса Pi:

```
do {
```

```
  while (turn != i)
    ; //do nothing
  ...
  критична секция
  ...
  turn=j;
  ...
  некритична секция
  ...
} while (...);
```

Този алгоритъм удовлетворява условието за взаимно изключване на процесите. Когато единият процес Pi се намира в критична секция, другият процес Pj изчаква, докато дойде неговия ред. Коректността на решението идва от там, че присвояването на стойност turn=j е атомарна операция Това решение обаче не удовлетворява останалите изисквания към механизма за синхронизация

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, защото 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.

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

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 очаква да се получи сигнал от друг процес. Когато даден процес очаква да получи сигнал, той преминава в блокирано състояние. С всеки семафор е асоциирана целочислена променлива. Върху семафорите могат да се изпълняват само две операции, които са атомарни (неделими). 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, процесът, който изпълнява операцията, блокира. Операцията signal() увеличава стойността на семафора с единица. Ако върху семафора има блокирани процеси, то операцията signal() води до отблокиране на някой от блокираните процеси.
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. При стойност 0 семафорът се счита за "зает", а при 1 - за "свободен"

race condition - няколко процеса използват общ ресурс и го ползват по начин, който да допусне възникването на объркващи събития
Настъпва, защото паралелно два или повече процеса се опитват да изпълнят операция по промяна на данните на точно определено място
Решенията са да се забрани временно на конкретни процеси достъп до ресурсите

Критична секция - парчета код, които създават опасно положение

Може да настъпи в еднопроцесорна система

62. Една от класическите задачи за синхронизация се нарича Задача за читателите и писателите (Readers-writers problem).

а) Опишете условието на задачата
б) Опишете решение, използващо семафори

а) Задача за достъп до общ ресурс. Стая, в която я използват хора и в която има публикувана информация. Ако не се правят промени
в текста, много хора могат да четат тази информация. Паралелно могат да четат много процеси - не настъпва race conditions – до ресурса може да пускаме безброй много четящи процеси. Ако обаче трябва да се прави промяна в текста, е хубаво да пускаме само един процес, който ще прави промени по тези данни, защото ако паралелно някой друг читател/писател работи с него, ще настъпи race condition. В тази задача отново имаме защитено пространство и два типа обекти (процеси) - readers и writers. За всички типове процеси искаме да имаме следните изисквания:
- или в стаята да няма никого
- или в стаята да има само 1 писател
- или в стаята да присъстват 1 или повече читатели

Как се осигурява такова поведение с помощта на синхронизиращи семафори?

Ако приемем условието само 1 читател да е в стаята, то тогава решението ни е с обикновен мютекс.

semaphore re (room empty)
re.init(1)

readers

re.wait()

четене

re.signal()

_____

writers

re.wait()

писане

re.signal()

Как да направим това решение да работи така, че да може няколко читатели да присъстват? За да може читатели да започнат да ползват стаята, трябва преди това тя да е празна.

Отново могат да чакат този семафор да ги пусне, но ако влезе 1 читател, то тогава могат всички останали да влязат. Те не трябва да следят дали е празна, а дали вътре има вече читател. Първият читател пали една лампа, а всички други читатели не гледат семафора, а гледат дали свети лампата, а последният читател ще изгаси тази лампа. Решението е с брояч.

int r=0 -> брои колко читатели има

```
r=r+1
if (r == 1)
  re.wait()

четене

r=r-1
if (r == 0)
  re.signal()
```

Обяснение: Когато пристигне читател, ще се увеличава броячът с 1. Ако броят на читателите е 1, тоест е пристигнал първият читател, то тогава той може да запали лампата, тоест да мине през семафора и да го отвори. След това, ако пристигнат още читатели, те няма да се интересуват от семафора, а ще се интересуват дали могат да преминат. След като читателите приключат със своята дейност, тогава те искат да излязат от стаята. Намалява се тяхната бройка с 1 и се проверява дали е останал последният читател. Ако няма други читатели, освен 1, то тогава той трябва да сигнализира на семафора да остави вратата отворена за първия читател или писател, който ще дойде по-късно.

Възниква проблем с брояча - ако няколко читатели го нападнат, тогава ще има race condition.

За да го защитим, слагаме още един семафор, който е обикновен мютекс.

```
sem m
m.init(1)

m.wait()
r=r+1
if (r == 1)
  re.wait()
m.signal()

четене

m.wait()
r=r-1
if (r == 0)
  re.signal()
m.signal()
```

Можем ли да се убедим, че това решение работи? Можем ли да докажем какво ще е състоянието на семофарите и броячът. Колко процеса се намират в опашките на семафорите и от какъв вид? Какви промени могат да настъпят в тази ситуация?
В мютекса влизат само четящи процеси, не могат да присъстват писатели в опашките.
В re семафора (който следи дали в стаята има някого) може да има няколко писатели в опашката и най-много 1 читател (0 или 1 читател). Ако е точно 1 читател, то тогава

някой писател пише (тоест в стаята има писател). Тези свойства в началото са изпълнени. Няма никой в стаята и няма никой в опашките на семафорите. Всяко преминаване през тези секции/бариерите запазват тези инварианти.
- Ако има 1 читател в опашката, може ли да се появи друг читател? Друг читател може да се появи само при re.wait(). Читателят, който е преминал през този ред re.wait(), то тогава мютексът е блокиран и останалите читатели не могат да влязат там. Мютексът е приспан. Трупат се процеси, но никой не може да премине, защото е защитена секция.
if (r==1)

Когато се блокира процес при re.wait() при читателите, то или читател е влязъл в стаята и е блокирал останалите писатели, или има писател, който пише. Ако е писател, след него или ще се пусне друг писател, или читател. Когато писател напусне, е гарантирано, че в стаята няма никой, което означава, че след това сме сигурни, че няма читатели вътре и броячът ще стане от 0 на 1 и ще мине отново през re.wait(), но вече ще влезе, защото е освободена стаята. Останалите читатели ще са 2, 3 и т.н и няма да преминат през re.wait(), ще прескочат тази секция заради проверката. Семафорът re има стойност 0 или 1.

Решението е коректно, но има недостатък - писателите могат да бъдат приспани дълго време и ще настъпи гладуване. Като почне някой да чете, пък след него влизат други, ама първият не е излязъл и ако читателите четат бавно и има безброй много процеси, то писателите ще бъдат приспани, те ще гладуват и ще чакат неопределено дълго време.

Ако искаме да го избегнем, то можем да променим това решение така, че като се появи писател в системата, да блокираме входа на нови читатели. За тези, които са влезли, нищо не можем да направим - те са влезли. Но можем да блокираме идването на нови читатели. Трябва да спираме читателите да влизат при условие, че има поне 1 писател, който да чака.

Решението: слагаме още един семафор t
sem t
t.init(1)

writers

t.wait()
re.wait()

-> t.signal()
писане

-> t.signal()
re.signal()
-> t.signal()

В един от трите момента трябва да освободи t. Ако го напишем преди писане, то тогава той ще почне да пише и ще е освободил останалите читатели да влизат, но това не е фатално

В момента, в който поне 1 писател се появи да чака, то тогава трябва да блокираме читателите да влизат, а след това ще го освободим.

readers

```
t.wait()
t.signal()

m.wait()
r=r+1
if (r == 1)
  re.wait()
m.signal()
```

четене

```
m.wait()
r=r-1
if (r == 0)
  re.signal()
m.signal()
```

Ако стойността на t е 0 или отрицателна, то читателите не могат да преминат. Първият читател, който дойде, ще направи стойността на t от 1 на 0, след което ще сигнализира, че може и други хора да минат след него.

В книжката има още едно решение, което дава преимущество на писателите - в стаята се пускат писателите с предимство пред читателите, но тогава ще настъпи гладуване за читателите поради същите причини - може писателите да пишат бавно и да се трупат много в стаята, а читателите да си чакат безброй много време.

В книжката има още една интересна задача, поставено от Дейкстра и дълго време не е била решена.

Имаме група процеси и трябва да се защити критична секция

Това е решение за защита на критична секция. Въпросът е може ли много процеси да изпълняват тази критична секция и да настъпи гладуване и в кои случаи?

```
m.wait()
.
.
.
m.signal()
```

Нека семафорът не е силен. Ще може ли да се получи гладуване? Ако е силен, няма да се получи, защото всички ще си влязат според реда си. Но ако е слаб семафор, то тогава може да настъпи гладуване, ако се пререждат процесите помежду си.

Въпросът на Дейкстра е дали може със слаби семафори да се реализира мютекс и да няма гладуване. Не е необходимо да го учим, няма да изпитва за това.

Задача за deadlock - говори се за петима философи, които са професори в някакъв италиански университет. Понеже са философи, те не се разбират помежду си и те всеки ден отиват да обядват заедно и сядат на една маса, на която има 5 чинии, както и 5 вилици. Когато им дадат макарони, всеки философ иска да яде с 2 вилици. Тогава философите влизат в конфликт, защото всеки философ иска да яде с двете вилици

около него. Трябва да се изчакват взаимно. Първо един яде с две вилици, оставя ги, избърсва ги, после друг се храни с две вилици и така нататък. Но философите не си говорят помежду си и не могат да се разберат. Има условие за deadlock - ако не се изчакват взаимно. Може да следват алгоритъм да хванат дясната вилица и никога да не я пуснат. Което означава, че никой философ няма да има 2 вилици. Условие за deadlock. Може да има взаимно изчакване - когато един философ вземе една вилица, но не може да вземе друга, да остави своята и да изчака определено време, докато не се освободят 2 такива. Обаче тук настъпва условие за гладуване, защото може някой философ да яде прекалено дълго време. Ако двама философи започнат да се хранят, всеки с две вилици, то останалите 3ма ги чакат. След това тези двамата решават да си направят почивка, след което отново те да започнат да се хранят. По този начин настъпва гладуване, защото останалите стоят и чакат дълго време и не могат да вземат вилиците.

63. Подобна е на 56 задача

R: running or runnable -> изчаква процесорът да го допусне
S: interruptible sleep -> изчаква някое събитие да приключи
D: uninterruptible sleep -> процеси, които не могат да бъдат убити или прекъснати от сигнал; често, за да се прекъснат, е необходимо да се реши проблемът
Z: zombie -> не е напълно спрян процес; процес, чието спиране не е завършено/осъществено
T: stopped -> процес, който е бил прекъснат

64. Всеки от процесите P, Q и R изплнява 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:
- някоя от инструкциите p2 и q2 да се изпълни преди r2
- ако инструкция p2 се изпълни преди r2, то q2 да се изпълни след r2
- ако инструкция q2 се изпълни преди r2, то p2 да се изпълни след r2

Забележка: Решение с 2 семафора се оценява с 30 точки, а решение с повече семафори носи 20 точки

Решение:
За синхронизация между отделните процеси ще използваме два семафора - u и t, като семафора u ще го инициализираме с начално отблокиращо състояние, а семафора t ще го инициализираме с начално блокиращо
състояние:
semaphore u, t
u.init(1)
t.init(0)

process P	process Q	process R
p_1	q_1	r_1
u.wait()	u.wait()	t.wait()
p_2	q_2	r_2
t.signal()	t.signal()	u.signal()

p_3 q_3 r_3

От условието се иска една от двете инструкции - p_2 или q_2 да се изпълни преди инструкция r_2. За тази цел използваме семафора u с начално състояние 1, което означава, че която от двете инструкции завърши първа - p_1 или q_1, ще може процесът P/Q да продължи надолу към изпълнение на инструкция номер 2. Под инструкция r_1 поставяме инструкция t.wait(), която ни гарантира, че процес R няма да продължи надолу към изпълнение на останалите инструкции, преди да е подаден сигнал на семафора t (неговото начално състояние е 0 и поради тази причина процес R ще изчака някоя от другите две инструкции - p_2 или q_2 да завърши, за да получи сигнал). Нека си представим следната ситуация: Инструкция r_1 завършва преди инструкциите p_1 и q_1. След своето изпълнение, процес R няма да продължи с изпълнението на останалите инструкции r_2 и r_3 заради инструкция t.wait(). Нека инструкция p_1 приключи преди инструкция q_1. Семафорът u е с начално състояние 1, което означава, че ще позволи на процес P да продължи надолу с изпълнение на инструкция p_2. В този момент, ако инструкция q_1 приключи изпълнението си, процес Q няма да има възможност да продължи нататък, защото семафор u ще е в състояние 0, тъй като процес P вече е преминал през негова инструкция u.wait(). Процес P ще изпълнява своята инструкция p_2 и когато приключи с нея, ще сигнализира на семафор t за освобождаване на ресурса, за да може процес R да премине към изпълнение на инструкция r_2 (от условието се иска, ако инструкция p_2 се изпълни преди инструкция r_2, то инструкция q_2 да се изпълни след инструкция r_2). След това процес P може да премине към изпълнение на инструкция p_3 (нямаме ограничение за инструкции с номер 3). Процес R преминава към изпълнение на инструкция r_2, а процес Q изчаква да бъде допуснат за изпълнение на инструкция q_2. След като процес R изпълни своята инструкция r_2, той преминава през инструкция u.signal(), която сигнализира на семафора u за освобождаване на ресурс и пуска изчакващия в опашката процес Q да продължи с изпълнение на останалите инструкции. По този начин се изпълнява условието инструкция p_2 да се изпълни преди r_2, а инструкция q_2 да се изпълни след инструкция r_2. По аналогичен начин е и за случая инструкция q_2 да се изпълни преди инструкция r_2, след което инструкция p_2 да се изпълни след инструкция r_2.

65. Дадена е програма за ОС Linux, написана на езика C:

```
#include <unistd.h>
#include <stdio.h>
int main(void)
{
  int p1, p2;
  p1=fork();
  p2=fork();
  printf("Hello world!\n");
}
```

а) Колко пъти ще се отпечата текста "Hello world!" при изпълнението на програмата? Обосновете
отговора си.
б) Как работи системното извикване fork()?
в) Нарисувайте кореновото дърво с върхове процесите, които ще се стартират в резултат от изпълнението на
програмата и ребра двойките родител-наследник.

а) Текстът "Hello world!" при изпълнението на програмата ще се отпечата 4 пъти. fork() създава нов процес, като дублира този, който го извиква. Новият процес се нарича процес дете, а извикващият процес се нарича процес родител. Процесът дете и процесът родител работят в отделни пространства на паметта. След като командата fork се изпълни, започват да работят точно 2 процеса. В случая се извиква два пъти командата fork, което означава, че ще работят паралелно 4 процеса - 2 са получени при първия fork() и още 2 са получени при втория fork(). При първото извикване на fork() имаме 1 родител и 1 негово дете, които ще работят паралелно и ще изведат текста "Hello world" - 1 път ще го изведе родителят и 1 път ще го изведе детето. При второто извикване на fork() горният родител ще има още 1 дете, а неговото първо дете ще има 1 собствено дете. По този начин ще имаме 4 паралелно работещи процеса, които ще изведат общо 4 пъти текста "Hello world!"

Интернет:
fork() създава изцяло нов процес, като дублира процеса, който го извиква. Новият процес се нарича процес дете, а извикващият процес се нарича процес родител. Процесът дете и процесът родител работят в отделни пространства на паметта.
fork връща pid (process ID). Този pid има 3 случая:
1. pid < 0 - в този случай е настъпила грешка при създаването на процес дете (при дублирането)
2. pid = 0 - този случай е свързан с новосъздадения процес дете
3. pid > 0 - този случай е свързан с процеса родител или този, който го извиква. Родителският процес знае pid на своето дете

По-добро обяснение: https://www.csl.mtu.edu/cs4411.ck/www/NOTES/process/fork/create.html
Системното извикване fork() се използва, за да създава процеси. Това извикване не приема аргументи и връща pid (process ID). Целта на командата fork() е да създаде изцяло нов процес, който се превръща в дете на процеса, който извиква командата fork(). След като новият процес дете е създаден, то тогава
двата процеса - процесът родител и процесът дете, изпълняват следващите инструкции, като следват системното извикване на fork(). По някакъв начин обаче трябва да се различава процесът родител от процеса дете. Това може
да се осъществи чрез проверка на върнатата стойност от командата fork() -> тоест проверка на върнатия pid (process ID):
- ако fork() върне отрицателна стойност, то тогава командата не е могла да се изпълни и не се е създал процес дете (създаването на дете е било неуспешно, възникнала е системна грешка)
- ако fork() върне стойност 0, то това означава, че тази стойност 0 се отнася за новия процес, който е създаден (тоест сме в случая на процеса дете)
- ако fork() върне положителна стойност, то това означава, че тази положителна стойност се отнася за процеса, който извиква командата fork() (тоест сме в случая на процеса родител)

След извикване на командата fork() може да се провери в кой случай сме - дали в процеса дете, или в процеса родител. Командата fork() прави точно копие на адреса на процеса родител и го доставя на процеса дете. Процесът родител и процесът дете използват отделни адресни пространства.

Диаграма:

```
      Parent
 main()
 {
--->fork();
   pid=...;
```

```
        .....
    }
```
_____

Ако извикването на fork() е успешно, то тогава Unix ще направи следното:
- ще направи две еднакви копия на адресни пространства, едното от които ще е за родителя, а другото за детето
- двата процеса ще започнат изпълнението от следващия ред, след извикването на fork(). В този случай двата процеса ще започнат изпълнението си по този начин:

```
     Parent          Child
   main()          main()
   {               {
     fork();         fork();
  --->pid=...;    --->pid=...;
      .....           .....
   }               }
```

Двата процеса започват изпълнението си веднага след системното извикване на fork(). Въпреки че двата процеса имат еднакви, но разделени адресни пространства, променливите, които се намират ПРЕДИ извикването на fork(), ще имат същите стойности и в двете адресни пространства. Всеки процес си има свое лично адресно пространство и всяка промяна ще бъде самостоятелна - независима от останалите. С други думи, ако процесът родител промени стойността на своята променлива, тази промяна ще засегне единствено променливата в адресното пространство на процеса родител. Останалите адресни пространства, които са създадени от fork() извиквания, няма да бъдат засегнати дори да имат променливи с едно и също име.

в) Преди първото извикване на fork() имаме следната програма:

```
#include <unistd.h>
#include <stdio.h>
int main(void)
{
  int p1, p2;
  p1=fork();
  p2=fork();
  printf("Hello world!\n");
}
```

Първо извикване на fork():

```
    Parent1
      /
   Child1
```

След първото извикване на fork(), ние ще имаме паралелно 2 работещи процеса, които ще са на отделно място в паметта. Ще имаме следната ситуация:

```
   Parent1                    Child1
  #include <unistd.h>       #include <unistd.h>
  #include <stdio.h>         #include <stdio.h>
  int main(void)             int main(void)
```

```
  {                                 {
    int p1, p2;                       int p1, p2;
    p1=fork();                        p1=fork();
--->p2=fork();                    --->p2=fork();
    printf("Hello world!\n");         print("Hello world!\n");
  }                                 }
```

И двата процеса ще продължат от реда p2=fork(); Това означава, че fork() ще създаде и в процеса Parent1, и в процеса Child1 по още едно тяхно копие - едно копие на процеса Parent1 и едно копие на процеса Child1. Тоест ще имаме следния случай:

Второро извикване на fork():

```
        Parent1
       /      \
     Child1   Child2
      /
  Child1_Child1
```

```
Parent1                  Child2                   Child1                   Child1_Child1
#include <unistd.h>      #include <unistd.h>      #include <unistd.h>      #include <unistd.h>
#include <stdio.h>       #include <stdio.h>       #include <stdio.h>       #include <stdio.h>
int main(void)           int main(void)           int main(void)           int main(void)
{                        {                        {                        {
  int p1, p2;              int p1, p2;              int p1, p2;              int p1, p2;
  p1=fork();               p1=fork();               p1=fork();               p1=fork();
  p2=fork();               p2=fork();               p2=fork();               p2=fork();
>printf("Hello world!\n"); >print("Hello world!\n"); >printf("Hello wordl!\n"); >printf("Hello
world!\n");
}                         }                        }                        }
```

Така ще се изведе 4 пъти текстът "Hello, world!"

66. Множество паралелно работещи копия на всеки от процесите P и Q изпълняват 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 работещите копия така, че три инструкции p1, q2 и p3 се редуват циклично:
- първа се изпълнява инструкция p1 на някое от работещите копия на процес P
- след завършването ѝ се изпълнява инструкция q2 на някое копие на Q
- след нея - p3 на някое копие на P
- с това едно минаване през цикъла завършва и отново може да се изпълни инструкция p1 на някое от работещите копия на процес P

Решение:
За синхронизация между отделните процеси ще използваме 3 семафора - u, t и m, като семафор u ще е с начално отблокирано състояние, а семафори t и m ще са с начално блокиращо състояние:

```
semaphore u, t, m
u.init(1)
t.init(0)
m.init(0)

process P    process Q
u.wait()        q_1
 p_1           t.wait()
t.signal()      q_2
 p_2          m.signal()
m.wait()        q_3
 p_3
u.signal()
```

От условието имаме, че инструкциите p1, q2 и p3 се редуват циклично. Това означава, че може едно копие на процеса P да започне работа по своите инструкции. Друго копие на процеса P може да започне да изпълнява инструкциите си, когато завърши цикълът p1-q2-p3. Тоест, като за начало, семафорът u ще е инициализиран с начално отблокирано състояние 1, за да може да пусне точно едно копие на процеса P до инструкция p_1. След като едно копие на процеса P започне работа по инструкция p_1, останалите копия на процеса P ще бъдат блокирани в опашката на семафора u. За инструкция q_1 нищо не се споменава в условието - изисква се едно копие на процеса Q да започне работа по инструкция q_2 само когато някое от копията на процеса P изпълни инструкция p_1. За тази цел сме инициализирали семафор t с начално блокиращо състояние 0, като поставяме инструкция t.wait() непосредствено след q_1, за да блокираме копията на процеса Q след завършване на инструкция q_1. След като копие на процеса P завърши с изпълнение на инструкция p_1, то тогава трябва да сигнализира на семафор t, за да може той да допусне до инструкция q_2 някое от чакащите копия на процеса Q в опашката. За инструкция p_2 не е споменато нищо в условието, което означава, че копие на процес P може да продължи работата с инструкция p_2. Има изискване обаче инструкция p_3 да не се изпълнява, преди да е завършила инструкция q_2 на някое от работещите копия на процеса Q. Поради тази причина поставяме инструкция m.wait() след инструкция p_2. Семафорът m е с начално блокиращо състояние и ни гарантира, че някое от работещите копия на процеса P няма да продължи да работи по инструкция p_3, докато семафорът m не го допусне. За тази цел, веднага след завършването на инструкция q_2 от някое работещо копие на процеса Q, поставяме инструкция m.signal(), която ще сигнализира на семафора m за освобождаване на ресурса, а той от своя страна ще допусне някое от работещите копия на процеса P да продължи с изпълнението на инструкция p_3. За q_3 нищо не се споменава в условието, което означава, че някое от работещите копия на процеса Q може да продължи с изпълнение на инструкция q_3. Обаче цикълът не е приключил. Той приключва, когато инструкция p_3 свърши изпълнението си. След като това се случи, е необходимо да поставим инструкция u.signal(), за да може да се сигнализира на семафора u за освобождаване на ресурса и той да допусне друго работещо копие до инструкцията p_1 и отново да може да се завърти този цикъл - p_1, q_2, p_3. По този начин са удовлетворени и четирите изисквания от условието.

67. Опишете накратко кои системни извиквания изграждат стандартните комуникационни канали в UNIX – неименувана тръба (pipe), връзка процес-файл, двустранна връзка процес-процес (connection)

- неименувана тръба (pipe):
pipe(int fd[2])
fd[2] е масив от два елемента
Най-простият комуникационен канал между два процеса и е анонимен - не е известен на останалия свят освен на процеса, който го създава
Тази комуникационна тръба изглежда като тръба с два края, по която ще текат байтове - един край за четене и един за писане. Идеята на такава тръба е тя да свърже два процеса. Трябва единият й край да е свързан към един процес, а другият край към друг процес, но създаването на такава сложна конструкция ще изисква да кажем кой процес с кой ще си комуникира, кой ще е в единия край, кой в другия. Системното извикване pipe е много просто - то създава тази тръба, аргументът е масив от два файлови дескриптора и двата края са свързани към процеса, който я създава. Изглежда ненужна работа - shell става собственик на двата края, с разликата, че те ще са с два различни номера fd0 и fd1 например. Вътре в масива ще получат стойности 3 и 4 примерно - тоест шел може да говори със себе си. Процесът създава този комуникационен канал и само той го вижда - невидим е за останалия свят. Сега със серия от инструкции ще го направи видим за два процеса.

pipe() създава тръба, която е еднопосочен канал за данни, който може да се използва за междупроцесорна комуникация
Масивът fd[2] връща два файлови дескриптора, които представляват краищата на тръбата. fd[0] се отнася за края на тръбата, от който ще се чете, а fd[1] се отнася за другия край на тръбата, в който ще се пише. Данните, записани в края на тръбата, се буферират от ядрото, докато не бъдат прочетени от края на четенето на тръбата.

Нека си представим следния случай:
some-command | grep

Какво се случва тук?
1. Шел пита ядрото за тръбата. Тя няма име, което означава, че не може да се осъществи чрез open, както при нормалния файл - вместо това се осъществява с pipe, което връща два файлови дескриптора.
2. Извиква се команда fork():
pid1=fork() -> fork() създава копие на процеса родител - и двата края на тръбата се отварят тук, копира се страната за писане на тръбата на fd 1(stdout). Kernel има системно извикване за копиране на номера на файлови дескриптори: dup2(). След това затваря страната за четене и другото копие на страната за писане. Най-накрая се изпълнява some-command. Докато единият край на тръбата е fd 1, то изходът от командата some-command ще се записва в тръбата.
3. Извиква се fork() отново. Този път се копира страната за писане на fd 0 (stdin) и се изпълнява командата grep, за да може grep да чете не от стандартния вход, а от тръбата.
4. След това процесът родител изчаква двете си деца да завършат изпълнението
5. Ядрото забелязва, че тръбата не е отворена повече и разчиства след нея. По този начин се унищожава тръбата.

- процес-файл:
Трябва да се изгради връзка между процеса, който ще обработва информацията с файла и файла, който един вид е пасивен процес, който ще се грижи за файла. Процесът, за да работи с друг процес или файл, трябва да изгради комуникационен канал. Когато работим с файлове, командата, която изгражда такава връзка с файл, е open (man 2 open). Има 2 или 3 аргумента - указател към символен низ е аргумент 1, вторият аргумент е integer flag open(pathname, flag). Всъщност, за да се свържем с файл, приемаме, че това е дълго живеещ информационен обект,

който присъства в пространството на имената, и когато искаме да отворим комуникационен канал към него, трябва да укажем името на обекта. Този низ може да бъде пълното име на файла или името в текущата директория, или някое име, указана по стандартите, когато работим с shell. Не е чисто системно извикване, а е библиотечна функция. Работата с файлове се извършва от цялостна подсистема, която трябва да може да ги открива къде са разположени върху хардуера и да ги изчислява къде и как са разположени. flag са отделни битове, които дават специфични свойства на комуникационния канал - дали ще четем, дали ще пишем, дали ще правим и двете неща. Други свойства на комуникационни канал са какво да е поведението, ако файлът е стар, нов и тн. Например O_CREAT има смисъла на - ако файлът не съществува, създай файл с такова име. Друго поведение е какво ще правим, като завършим работата - дали да затвори файла, с който работи. Един процес, като стартира нови процеси - свои деца, той предава файловите дескриптори на децата си и трябва това свойство да можем да го манипулираме. Друго свойство е O_TRUNC - когато затваряме файла, дали да го отрежем, където е указателят на файла, оттам нататък да го изрежем; когато искаме да намалим размер на файла. Други важни - когато отваряме с опция O_APPEND, указателят към текущата позиция, с която ще работим с четене и писане, ще се позиционира в края на файла.

Създаването на файлов дескриптор се извършва с командата open, която връща нов файлов дескриптор. Затварянето на комуникационен канал се извършва с close(fd). Командите за четене и писане са:
реално прочетени байтове<--read(fd, *buf, count),
реално прочетени байтове<--write(fd, *buf, count)

fd - файлов дескриптор, комуникационен канал, който е готов за работа
*buf - указател към масив от байтове
count - колко байта искаме да прочетем

На най-ниско ниво операциите на четене и писане са такива.
Резултатът, който връщат, е колко байта реално са прочели.

Когато е в синхронен режим - ако файлът има толкова байтове, той ги прочита, и реално прочетените байтове ще съвпаднат с тези, които искаме да прочетем
Когато е в асинхронен режим - искаме да прочетем примерно 1000 байта, обаче изходният канал има 100 готови, а другите 900 ги няма. Тогава ще върне 100 и ще върне грешка - няма да се приспи нашата програма, а ще върне грешка.

O_NONBLOCK означава в какъв режим ще се работи с файла - нормалната работа с един комуникационен канал, която е естествена за програмиста и операционната система, е да работим с отсрещната страна, двата процеса или процес-файл като се свържат и извършват обмен на данни, да работят синхронизирано. Синхронен вход-изход – входно/изходните операции да са подредени във времето; самият входно-изходен канал да управлява поведението на процесите – да решава дали да бъде приспан или събуден даден процес. Когато включим тази опция, това поведение се променя - когато отворим файла по такъв начин или сменим поведението му с друга команда за контрол на тези опции и тогава този синхронен вход/изход при включена опция O_NONBLOCK се превръща в асинхронен вход/изход.

- двустранна връзка процес-процес (connection):
connection (socket()) за разлика от тръбата, може да свърже два процеса, които не знаят един за друг, нямат връзка помежду си.
Може да се окаже между процеси от различни изчислителни системи.

За да се изгради връзка между два непознати процеса, трябва да се направи по някакъв начин те да се видят. Тоест единият процес да разбере за другия и за възможността да се свържат. Методът за научаване е обектът socket. Създава крайна точка за комуникация - тоест двата процеса изграждат нещо, което наричаме socket. То все още не е връзка между тях. То е начало за установяване на такава връзка. Втората стъпка се извършва от системното извикване socket() -> socket() създава един вид такива крайни точки. Следващата фаза е на единия socket да му се присвои име - да стане видим в пространството на имената. Има различни пространства на имената в съвременния свят - 3 пространства на имената са останали.
AF_UNIX - локалната изчислителна система, дървовидна структура, в която са записани файловете, другите виртуални обекти, които не са представени като същински файлове, запаметени на диска. Освен обикновени файлове и директории
в това пространство може да има имена на периферни устройства или виртуални периферни устройства, които се обслужват от драйверите, може да има символни връзки, именувани тръби (рядко се ползват), може да има сокети. Създаването на име на socket() се извършва от системното извикване bind(). При установяване на връзка единият процес наименува сокета си. Като се създаде име на сокета, той става видим за останалите процеси, които работят в системата. В различните адресни пространства, те имат различна структура - 3 адресни пространства има.
AF_UNIX е най-старото и най-простото за разбиране - все едно име на файл във файловата система, само че не е обикновен файл, а има буква s -> специален битов елемент - сокет.
AF_INET
AF_INET6

сокети, които не са видими в локалната система на изчисление, а са видими в целия интернет - интернет връзки
Като се именува този сокет, другият процес може да го вижда. Тогава може да започне изграждането на връзка. Преди да бъде видим, сокетът трябва да бъде активен. Активирането на такъв именуван сокет става със системното
извикване listen(). Активира този сокет и той вече може да почне да работи, ядрото започва да поддържа структури, които започват да подслушват дали други програми не търсят именувания сокет. От страна на клиента, той също
трябва да си създаде сокет, но не е нужно да го именува. Той изпълнява системно извикване connect(), което изпраща заявка към именувания сокет - запитване за установяване на връзка и пасивният (слушащият) сокет получава тази заявка и може вече да изгради връзката, защото чрез заявката той разбира кой е другият процес -отдалеченият, който иска да създаде връзката. Слушащият сокет не замества комуникационния обект с многоизградената връзка, а създава все едно нов комуникационен обект и този комуникационен обект вече е връзката между двата процеса. Можем да си представим изградената връзка като две тръби - едната тръба е от клиента към сървъра, другата е от сървъра към клиента. В едната може да се пращат данни в едната посока, а в другата - в другата посока. Конекцията е еднопосочна тръба.

Слушащият процес при установяване на връзката създава новата конекция, но и тази заявка за създаване на конекция е обслужена, но той я създава като нов комуникационен обект, ама слушащият сокет продължава да работи. Той може да служи за нови връзки. Той е именуван. Идеята му е да е видим за всички останали процеси и те да могат да правят
връзки. След като могат всички процеси да правят връзки с него, този процес започва да играе ролята на сървър, може да установява нови връзки, може да се появи друг процес и той да подаде заявка за установяване на връзка и пак след съответните поредици от команди да се изгради. Процесът на установяване на връзка пак е от две системни извиквания - клиентският процес изпълнява команда connect, която изпраща тази заявка, а сървърът изпълнява команда

accept, която я приема. Тук се трупа опашка от заявки в слушащия сокет и команда accept, извикана от сървъра завършва изграждането на конекцията и създава конекцията и файловите дескриптори към нея. Connect създава този файлов дескриптор клиентския процес. Accept създава този файлов дескриптор сървърния процес.

Дефиниция на сървър - процес, който е създал име или сокет в някакво пространство, което е видимо за други процеси, и по този начин дава възможност на други процеси, които ги наричаме клиенти, да изградят връзка към него и да провеждат разговори с него. Казано с други думи - процес, който позволява на други процеси да се свързват с него и да провеждат комуникация и той да ги обслужва с много комуникационни канали - за всеки клиент по един комуникационен канал.

Сървърът и клиентът, за да използват тази връзка, използват read и write. Няма значение кое от трите вида обекти ще използват - pipe, отворен файл или връзка. Самият файлов дескриптор се създава от сокет (listener), accept създава connect-цията, connect замества файловия дескриптор за сокета на клиента с готовата конекция и read и write пишат в изградената конекция със стандартните си операции за четене и писане. Този обект се унищожава с командата close - както се затварят файловете. Същото е и с pipe - с read и write можем да четем и пишем.

За клиента трябва да се създаде сокет и после да се свърже със сокет - по-сложно е при клиента.

Всички без неименуваната тръба използват пространството на имената.

Канал между два процеса.

Разлика между нишка и процес:
Процесът си има pid, а нишката има pid на процеса и друг свой номер на нишката. Командата clone() създава нова нишка. Пита се дали създадената нишка има същия pid -> това означава, че се създава нова нишка в текущия процес. Пита се дали създадената нишка има нов pid -> това означава, че се създава един вид нов процес с родител, който е извикал командата clone().

Други неща, които се казват в параметрите на clone(),са какви обекти от останалите ресурси да бъдат споделени. Ако кажем нищо да не бъде споделено, това означава да направим fork() -> създава се нова памет на процеса, в нея ще се копира оригиналната памет на извикващия процес, обкръженията ще се копират, няма да са старите, файловите дескриптори са указатели към комуникационните канали, които старият процес използва, при реализация на fork() такива параметри се задават, че се създават копия към файловите дескриптори към комуникационните канали на оригиналния процес.
fork() е един вариант на clone() -> новата нишка получава нов уникален номер на процеса
Простата нишка е да не се правят копия, pid и паметта да останат същите, данните от обкръжението да са същите, да не се създават нови адресни пространства.

68. Опишете какви изисквания удовлетворява съвременната файлова система, реализирана върху блоково устройство (block device). Опишете накратко реализацията и целта на следните инструменти:
а) отлагане на записа, алгоритъм на асансьора
б) поддържане на журнал на файловата система

Отговор:
Описание на изискванията, които удовлетворява съвременната файлова система, реализирана върху блоково устройство (block device):
Изпълняват се нашите функционални изисквания - да е добре представено, добре изобразено множеството файлове върху физическия носител и съответните функции да се адресират сравнително ефективно - бързо да може да се извършват съответните операции и информацията да може да се съхранява.

Удвовлетворява изискването да се представи съдържанието на файла.

block_device - входно-изходно устройство, което е масиви от байтове, които не са рам паметта на компютъра, не са процесът и неговите регистри, а външно устройство, което съхранява масив от байтове, като можем да записваме байтове, където искаме. block device може да бъде и част от паметта - да го разглеждаме като диск. Виртуален диск в рам паметта - при зареждане на операционната система, тя се зарежда в един файл, част от рам паметта, все едно там се намира някакъв диск, и от там започва стартирането.

inode - малък масив, в частност може да не е цял сектор, а парче от сектор, в зависимост от конкретната структура на диска. За всеки файл има inode, който описва атрибутите и структурите, където се пазят данните - една част от inode съдържа атрибути, накрая има няколко указателя, които описват съдържанието на файловете. Първите няколко указателя са указатели директно към блокове от данни, след това има 3 указателя, които са към по-сложни структури - първият е блок, който е към масив от указатели, вторият е масив, сочещ към масив от указатели и третият е още по-задълбочено дърво. Със сравнително малко четения на сектори да изчислим произволен елемент на файла (част на файла) да я адресираме лесно, за да четем краен брой сектори и да изобразяваме файлове с малки и големи размери. Не е фиксиран размерът на блока - различна степен на двойката. Може да се правят различни по големина максимални файлове. Адресите са 32-битови.

а) Лекции:
Нека си представим следната ситуация:
```
  p1   p2   p3
|------------------|
```

Това е нашият диск, върху който извършваме дейността. Паралелно работят потребителските процеси, които създават заявки и алгоритъма, който чете или пише върху диска. Можем да си представим диска като дълъг коридор, в който се движат студенти и чистачка - студентите се движат и изхвърлят отпадъците си по земята (това са заявки). В друг момент от времето друг процес замърсява. Паралелно се извършват процесите. (все едно снежинки, които валят по коридора). Единият процес е чистачката, която трябва да разчиства боклуците.
Всички тези обекти работят паралелно - те ползват обща структура, общата структура е тази структура на данните, която описва заявките и работата с диска със споделени ресурси. Ще трябва да синхронизират своята работа, за да не разрушат данните. Чистачката си има позиция в коридора и посока, в която мете. Другите данни са къде са хвърлените отпадъци.
Бърз алгоритъм за разчистване на отпадъците:
Когато процес иска да направи заявка, той ще трябва да ползва тази структура данни и да добави нов отпадък в структурата. Чистачката, когато иска да използва структурата, трябва да разбере коя е най-близката заявка (най-левия в случая) - подходящо е да се реализира с приоритетни опашки - една, в която ще подреждаме по най-малък елемент, и една, в която ще подреждаме по най-голям елемент. Позицията върху диска ще е ключът върху приоритетните опашки. Споделените структури са чистачката и посоката, в която чисти. Когато процес иска да създаде нова заявка, той трябва да ползва монопол на структурата - ще провери къде е

чистачката и ако неговата заявка е надясно от нея, ще я сложи в дясната приоритетна опашка, а ако е вляво - в лявата. Когато дисковият контролер е в готовност - тоест е изпълнил вече някоя операция и иска да изпълни следващата, той ще се опита да заеме тази структура, чрез мутекса да се добере до нея, да разбере дали посоката му на движение има елементи - ако е празна, ще смени посоката на движение - ако е пълна, ще вземе екстремалния елемент.

Множество от процеси, които подават заявки, и процес, който ги изпълнява

```
  p1        p2       p3
PQmax       .       PQmin
 x x x      .        x x x
|-------------------------------|
             ^
             |->
             c
```

Ако обърнем картинката, това ще прилича на асансьор. Заявките си ги представяме като пътници, които чакат да се качат в този асансьор. Той ходи постоянно надолу-нагоре. Тази аналогия с асансьора не е много точна - по-точна е аналогията с чистачката. Този алгоритъм може да не изпълнява оптималния брой решения - посоката, в която се движи, да има разредени заявки, а тези зад гърба на чистачката, да са по-оптимални и да е по-изгодно да се върне при тях. Заявките може така да се конструират, че да се изпадне в състояние на гладуване. При алгоритъма на асансьора няма гладуване.
Някои заявки, които са поръчани по-рано от други, може да се изпълнят след останалите, тоест да се забавят. Това е проблем.

Интернет:
Алгоритъмът на асансьора е алгоритъм за планиране на дискове за определяне на движението на ръката и главата на диска при обслужване на заявки за четене и запис.

Този алгоритъм е кръстен на поведението на строителен асансьор, където асансьорът продължава да пътува в сегашната си посока (нагоре или надолу), докато се изпразни, като спира само за да пусне хората или да вземе нови хора, тръгнали в същата посока.

От гледна точка на внедряването, устройството поддържа буфер от чакащи заявки за четене / запис, заедно със съответния номер на цилиндъра на заявката. (По-ниските номера на цилиндъра обикновено показват, че цилиндърът е по-близо до шпиндела, а по-високите числа означават, че цилиндърът е по-далеч.)

Когато пристигне нова заявка, докато задвижването е на празен ход, първоначалното движение на ръката / главата ще бъде в посока на цилиндъра, където се съхраняват данните, навътре или навън. Когато пристигнат допълнителни заявки, заявките се обслужват само в текущата посока на движение на рамото, докато ръката достигне ръба на диска. Когато това се случи, посоката на ръката се обръща и исканията, които останаха в обратна посока, се обслужват и т.н.

б) Лекции:
Журналът е последователен файл, в който записваме пълното описание на серия от операции. Когато журналът се препълни, то тогава трябва да се спрат транзакциите. Когато журналът е запълнен, той трябва да започне да се

изпразва. Имаме два указателя в журнала, които показват кое е съдържанието, което само в журнала го има. Това място се освобождава, защото се записва в актуалната база. Базата данни има ново съдържание. Ако спре
токът и после пуснем базата, този журнален файл ще пази последните промени, а на другото място ще имаме консистентно съдържание на базата.

Интернет:

Файловата система за поддържане на журнал е файлова система, която поддържа специален файл, наречен журнал, който се използва за поправяне на несъответствия, възникнали в резултат на неправилно изключване на компютър. Такива изключения обикновено се дължат на прекъсване на захранването или на проблем със софтуера, който не може да бъде разрешен без рестартиране.

Файловата система е начин за съхранение на информация на компютър, който обикновено се състои от йерархия от директории (наричана още дърво на директория), която се използва за организиране на файлове. Всяко устройство с твърд диск (HDD) или друго устройство за съхранение, както и всеки дял (т.е. логически независим раздел на HDD), може да има различен тип файлова система.

Файловите системи за поддържане на журнал записват метаданни (т.е. данни за файлове и директории) в журнала, който се предава на твърдия диск, преди всяка команда да се върне. В случай на срив на системата, определен набор от актуализации може да е напълно ангажиран с файловата система (т.е., записан на HDD), в този случай няма проблем, или актуализациите ще бъдат маркирани като все още не напълно ангажирани, в този случай системата ще чете дневника, който може да бъде превърнат до най-новата точка на съгласуваност на данните.

Това е много по-бързо от сканиране на целия HDD при рестартиране и гарантира, че структурата на файловата система винаги е вътрешно последователна. По този начин, въпреки че някои данни могат да бъдат загубени, файловата система за поддържане на журнал обикновено позволява да се рестартира компютърът много по-бързо след срив на системата.

Друго:
Журналът се използва по смисъла на периодичен запис - напр. като дневник. Вместо да променя структурата на файловата система, добавяйки нови директории или разширявайки файлове, първото нещо, което прави файловата система, е да събере тези събития в серия транзакции и да прикачи тези транзакции към лог файл. След това фоновите процеси могат да поемат тези транзакции и да ги добавят към действителната файлова система. Ако има срив, файловата система може да се възстанови незабавно и да започне да извършва транзакции, присъстващи в дневника, но последният запис вероятно ще се повреди и ще отпадне. Всичко това означава, че файловата система ще остане последователна. Файлова система като тази (например ZFS) ще ви помогне да защитите както данните, така и метаданните и няма да изисква програми като fsck, които се опитват да се възстановят от счупване.

69. При реализация на файлова система върху твърд диск файловете и директориите се записват върху сектори от диска. Времето за достъп до секторите зависи от текущото положение на механичните компоненти на диска - над коя пътечка е главата за четене/запис и каква е позицията й над пътечката.
- Защо се прави разместване във времето на операциите по четене и запис върху диска?
- Опишете накратко реализацията и целта на алгоритъма на асансьора.

Отговор:
- Представяме си диска като линейна геометрична структура и ако главата се намира в едно място на диска, за да отиде на друго, тя трябва да губи време - времето е пропорционално на разстоянието, което трябва да измине главата. Представяме си, че имаме съвременна операционна система с много процеси, като всеки процес някакви файлове си ги обработва, опитва се да прави нещо с тях, и създава заявки към ОС за извършване на операции на запис и четене на сектори. Как ще изглеждат тези заявки?
p1 p2 p3 - потребителски процеси, като всеки си е отворил един или няколко файла. Процесът от време на време прави заявки за работа с диска, другият ще си прави други заявки върху диска. (x са заявките)
Дискът в някакъв момент ще се окаже, че има случайно разхвърляни заявки, които трябва да бъдат обработени. Ако ги обработваме в проста структура - например опашка, която ги подрежда по време на постъпване (x1 е пристигнала първа, x2 - втора и така нататък) Дискът ще отиде да изпълни първо заявка x1, после трябва да отиде при x2, после при x3 и така нататък, те не са подредени, а са разпределени хаотично. Ако ги изпълняваме в реда, в който постъпват, главата трябва да направи много разходки и това ще е един много бавен процес.

```
    p1                    p2                  p3
 x8 x3 x x x1 x x6 x x2 x x7 x x4 x x x5 x x x
|-----------------------------------------------------|
```

Можем да оптимизираме, като заявките, които сме натрупали като количество, не ги изпълняваме в реда, в който са постъпили, а ги разместим във времето. Първо да изпълним заявката над стрелката, после другите 2 отляво и после другите от 4-та нататък да се изпълнява 1 по 1 - тоест по геометричния ред, както са разположени, а не във времето, в което са пристигнали. При дадено множество от заявки, геометричното движение ще е най-много 1.5L от дължината на опашката, а горе ще е n.L (при хаотичното)

```
    p1       p2             p3
 x x x x x x x x x x x x x x x
|---------------------------------|
    ^
    |
```

Този алгоритъм се нарича алгоритъм на асансьора.

- Това може да се вземе от предната задача а)

70. Дадена е програма за ОС Linux, написана на езика С:

```
#include <unistd.h>
#include <stdio.h>
int main(void)
{
  int p1, p2, p3;
  p1=fork();
  if (p1=0) {
    p2=fork();
    if (p2>0) p3=fork();
  }
  printf("Hello world!\n");
}
```

а) Колко пъти ще се отпечата текстът "Hello world!" при изпълнението на програмата? Обосновете отговора си.
б) Как работи системното извикване fork()?
в) Напишувайте кореновото дърво с върхове процесите, които ще се стартират в резултат от изпълнението на програмата и ребра двойките родител-наследник.

а) Текстът "Hello world!" при изпълнението на програмата ще се отпечата 4 пъти. Системното извикване fork() създава нов процес - копие на процеса, който го извиква. След едно извикване на командата fork() започват паралелно работа два процеса, които имат един и същ код, но са на отделно място в паметта. Тоест, създадено е копие на адресното пространство на процеса, който извиква командата fork(). Сега по-нагледно ще представя нещата, които ще се случат в тази програма:

Първоначално имаме следната програма:

```
#include <unistd.h>
#include <stdio.h>
int main(void)
{
  int p1, p2, p3;
  p1=fork();
  if (p1=0) {
    p2=fork();
    if (p2>0) p3=fork();
  }
  printf("Hello world!\n");
}
```

Когато се стигне до ред p1=fork();, то тогава ще се създаде изцяло нов процес, който ще е копие на процеса, който извиква командата fork(); Ще имаме следната ситуация:

```
        Parent                      Child
   #include <unistd.h>          #include <unistd.h>
   #include <stdio.h>           #include <stdio.h>
   int main(void)               int main(void)
   {                            {
     int p1, p2, p3;              int p1, p2, p3;
     p1=fork();                   p1=fork();
 --->if (p1=0) {                --->if (p1==0) {
       p2=fork();                     p2=fork();
       if (p2>0) p3=fork();           if (p2>0) p3=fork();
     }                            }
     printf("Hello world!\n");    printf("Hello world!\n");
   }                            }
```

Вече имаме два паралелно работещи процеса - единият процес се нарича родител, а неговото копие - дете. Те продължават работата си под този ред, който съдържа fork(). Тоест, те продължават от реда if (p1==0). Знаем, че командата fork() връща pid (process ID). Този pid може да е с отрицателна стойност, ако е възникнала грешка при създаването на процеса дете, може да е със стойност 0, ако се намираме в процеса дете, и може да е с положителна стойност, ако се намираме в процеса родител. В случая нека първо погледнем процеса родител (Parent).

Тъй като това е parent процеса, след изпълнение на командата fork(), неговият pid ще е положително число. Тоест няма да влезем в случая p1=0, защото p1 ще е положително число (в случая на родителя сме). Прескачаме целия блок if и директно отиваме на printf("Hello world!\n"). Тоест дотук имаме, че текстът ще се отпечата 1 път. Нека сега видим какво ще се случи при процеса дете (Child). Той също продължава работата си от реда if (p1==0). Тъй като процесът дете (Child) е копие на процеса родител (Parent) (Parent е родител на Child, логично), то тогава при изпълнение на командата fork(), процесът дете ще има pid със стойност 0. Тоест p1 ще е със стойност 0. Оттук следва, ще при процеса дете ще влезем в проверката if (p1==0), няма да прескочим блока. Влизаме в него и имаме още едно извикване на fork() -> p2=fork(); Оттук ще получим следния случай:

```
      Child                    Child_Child1
  #include <unistd.h>          #include <unistd.h>
  #include <stdio.h>           #include <stdio.h>
  int main(void)               int main(void)
  {                            {
    int p1, p2, p3;              int p1, p2, p3;
    p1=fork();                   p1=fork();
    if (p1=0) {                  if (p1==0) {
      p2=fork();                   p2=fork();
 --->if (p2>0) p3=fork();       --->if (p2>0) p3=fork();
    }                            }
    printf("Hello world!\n");  printf("Hello world!\n");
  }                            }
```

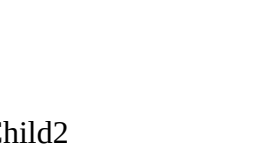
В случая се направи копие на процеса Child (тоест Child е станал родител и неговото дете е процесът Child_Child1). Сега в момента тези два процеса протичат паралелно, заедно с процеса Parent (тук не добавяме Parent, защото фокусът е върху fork(), който се намира в процеса Child и разглеждаме какво се случва в него, но не трябва да забравяме, че процесът Parent протича паралелно с тях). След това извикване на fork() двата процеса Child и Child_Child1 ще продължат да работят паралелно, като започнат от реда if (p2>0). Нека видим какво се случва в процеса Child. След изпълнение на реда p2=fork() се е направило копие на процеса Child. Това означава, че в случая Child се явява родител. Тоест извикването на fork() ще върне pid (process ID), което ще е положително, защото се намираме в случая на родител (Child). Тоест, ще се изпълни блокът if, няма да го прескачаме. Като влезем в блока if (p2>0), виждаме, че има още едно системно извикване на fork -> p3=fork(); Тук в момента ще се случи следната ситуация:

```
      Child                    Child_Child2
  #include <unistd.h>          #include <unistd.h>
  #include <stdio.h>           #include <stdio.h>
  int main(void)               int main(void)
  {                            {
    int p1, p2, p3;              int p1, p2, p3;
    p1=fork();                   p1=fork();
    if (p1=0) {                  if (p1==0) {
      p2=fork();                   p2=fork();
      if (p2>0) p3=fork();         if (p2>0) p3=fork();
    }                            }
 --->printf("Hello world!\n");  --->printf("Hello world!\n");
  }                            }
```

След изпълнението на реда p3=fork(), отново се създава копие на процеса, който го извиква - в случая процесът Child извиква fork(). Създава се още едно негово копие, което е негово дете -> процес Child_Child2. Процесът Child_Child2 започва също паралелна работа с останалите процеси. Нека видим какво се случва в родителския процес Child. Под реда p3=fork(); нямаме нищо друго, освен printf("Hello world!\n"), което означава, че този текст ще се изведе още 1 път (дотук се изведе един път в Parent процеса, сега се извежда още един път в Child процеса - тоест дотук се извежда 2 пъти). Нека видим какво се случва в процеса Child_Child2. По аналогичен начин - под реда p3=fork(); няма нищо друго, освен printf("Hello world!\n"), която означава, че този текст ще се изведе още един път - сега станаха 3 пъти. Нека не забравяме и процеса Child_Child1. Преди малко разгледахме какво се случва след реда p2=fork(); само в Child процеса и влязохме в случая if (p2>0) при процеса Child. Оттам направихме още едно негово копие и стигнахме до извода, че текстът "Hello world" до този момент се извежда 3 пъти. Но не сме разгледали и случая с процеса Child_Child1. След реда p2=fork(); ние имаме блок if, който проверява дали p2 е положително число. Тъй като процесът Child_Child1 е копие на процеса Child, това означава, че Child_Child1 е дете на процеса Child. Тоест, неговият pid ще е със стойност 0. Това означава, че няма да се влезе в случая if (p2>0). По-нататък нямаме нищо друго, освен printf("Hello world!\n"). Това означава, че текстът "Hello world" ще се изведе още един път. Като заключение имаме, че текстът "Hello world" ще се изведе общо 4 пъти.

б) Абсолютно същото е като при задача 65 -> тази подточка е същата

в) Кореново дърво:



71. Опишете как се изгрежда комуникационен канал (connection) между процес-сървър и процес-клиент със следните системни извиквания в стандарта POSIX:
socket(), bind(), connection(), listen(), accept()

За да се изгради връзка между два непознати процеса, трябва да се направи по някакъв начин те да се видят. Тоест единият процес да разбере за другия и за възможността да се свържат. Методът за научаване е обектът socket. Създава крайна точка за комуникация - тоест двата процеса изграждат нещо, което наричаме socket. То все още не е връзка между тях. То е начало за установяване на такава връзка. Втората стъпка се извършва от системното извикване socket() -> socket() създава един вид такива крайни точки. Следващата фаза е на единия socket да му се присвои име - да стане видим в пространството на имената. Създаването на име на socket() се извършва от системното извикване bind(). При установяване на връзка единият процес именува сокета си. Като се създаде име на сокета, той става видим за останалите процеси, които работят в системата.
Като се именува този сокет, другият процес може да го вижда. Тогава може да започне изграждането на връзка. Преди да бъде видим, сокетът трябва да бъде активен. Активирането на такъв именуван сокет става със системното извикване listen(). Активира този сокет и той вече може да почне да работи, ядрото започва да поддържа структури, които започват да подслушват дали други програми не търсят именувания сокет. От страна на клиента, той също трябва да си създаде сокет, но не е нужно да го именува. Той изпълнява системно извикване connect(), което изпраща заявка към именувания сокет - запитване за установяване на връзка и пасивният (слушащият) сокет получава тази заявка и може вече да изгради връзката, защото чрез заявката той разбира кой е другият процес - отдалеченият,

който иска да създаде връзката. Слушащият сокет не замества комуникационния обект с многоизградената връзка, а създава все едно нов комуникационен обект и този комуникационен обект вече е връзката между двата процеса. Можем да си представим изградената връзка като две тръби - едната тръба е от клиента към сървъра, другата е от сървъра към клиента. В едната може да се пращат данни в едната посока, а в другата - в другата посока. Конекцията е еднопосочна тръба.

Слушащият процес при установяване на връзката създава новата конекция, но и тази заявка за създаване на конекция е обслужена, но той я създава като нов комуникационен обект, ама слушащият сокет продължава да работи. Той може да служи за нови връзки. Той е именуван. Идеята му е да е видим за всички останали процеси и те да могат да правят връзки. След като могат всички процеси да правят връзки с него, този процес започва да играе ролята на сървър, може да установява нови връзки, може да се появи друг процес и той да подаде заявка за установяване на връзка и пак след съответните поредици от команди да се изгради. Процесът на установяване на връзка пак е от две системни извиквания - клиентският процес изпълнява команда connect, която изпраща тази заявка, а сървърът изпълнява команда accept, която я приема. Тук се трупа опашка от заявки в слушащия сокет и команда accept, извикана от сървъра, завършва изграждането на конекцията и създава конекцията и файловите дескриптори към нея. Connect създава този файлов дескриптор клиентския процес. Accept създава този файлов дескриптор сървърния процес.

Дефиниция на сървър - процес, който е създал име или сокет в някакво пространство, което е видимо за други процеси, и по този начин дава възможност на други процеси, които ги наричаме клиенти, да изградят връзка към него и да провеждат разговори с него. Казано с други думи - процес, който позволява на други процеси да се свързват с него и да провеждат комуникация и той да ги обслужва с много комуникационни канали - за всеки клиент по един комуникационен канал.

Сървърът и клиентът, за да използват тази връзка, използват read и write. Няма значение кое от трите вида обекти ще използват - pipe, отворен файл или връзка. Самият файлов дескриптор се създава от сокет (listener), accept създава connect-цията, connect замества файловия дескриптор за сокета на клиента с готовата конекция и read и write пишат в изградената конекция със стандартните си операции за четене и писане. Този обект се унищожава с командата close - както се затварят файловете. Същото е и с pipe - с read и write можем да четем и пишем.

За клиента трябва да се създаде сокет и после да се свърже със сокет - по-сложно е при клиента.

72. Опишете накратко основните комуникационни канали в ОС Linux. Кои канали Използват пространството на имената и кои не го правят?

Отговор:
Описание на основните комуникационни канали в ОС Linux има във въпрос 67.

Пространството на имената се използва при всички връзки освен при неименувана тръба.

С файловете е свързано пространството на имената. В съвременната операционна система има два слоя на абстракция:
- пространството на имената - виждаме като единно пространство на имена всички дълготрайни обекти, които съществуват в системата - абстрактна файлова система

- реализацията - как ще ги представим тези неща, как ще бъдат съпоставени на хардуера - конкретната реализация на всеки информационен обект - къде ще стои в хардуера и как ще бъде записан там

Именуван pipe (рядко се използва) и socket - комуникационни канали. Неименуваната тръба е известна само на процеса, който я е създал, за другите процеси е тотално невидима. За да я използва някой друг процес, този процес, който я е създал, трябва да я предаде на други процеси като наследство. Когато искаме процеси, които не се познават, да ползват общ комуникационен канал, трябва да го именуваме - вижда се аналогията между файлове и други неща, които дълго време съществуват и са в пространството на имената. Ако искаме да използваме файл, трябва да му знаем името, за да може процеса ни да го ползва, или ние като потребители да го използваме. По същия начин е и с комуникационни канали - ако искаме една тръба да я ползват процеси, които не се познават, трябва да я именуваме и да я сложим в пространството на имената. Сокетът е по същия начин.

Всички без неименуваната тръба използват пространството на имената.

73. Всеки от процесите P и Q изпълнява поредица от три инструкции: - не е проверена

```
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 p1 да се изпълни преди q2
2. инструкция q2 да се изпълни преди p3
3. инструкция q1 да се изпълни преди p2
4. инструкция p2 да се изпълни преди q3

Решение:
За моментите на синхронизация ще използваме два семафора - u и t, като ги инициализираме с блокиращо начално състояние:

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

След това добавяме в кода на процесите P и Q синхронизиращи функции (2 решения):

```
process P       process Q                process P       process Q
 p_1              q_1                      p_1              q_1
u.signal()       u.wait()                 u.signal()       t.signal()
t.wait()         t.signal()               t.wait()         u.wait()
 p_2              q_2                      p_2              q_2
u.signal()       u.wait()                 u.signal()       t.signal()
t.wait()         t.signal()               t.wait()         u.wait()
 p_3              q_3                      p_3              q_3
```

По условие (1.) имаме, че инструкция p1 трябва да се изпълни преди инструкция q2. Поради тази причина след инструкция q1 записваме инструкция u.wait(), за да може инструкция q1,

ако е приключила преди инструкция p1, да изчака инструкция p1 да приключи. В точка 3 от условието имаме зависимост инструкция q1 да се изпълни преди p2. Поради тази причина непосредствено след u.signal() в процеса P записваме инструкция t.wait(), която ни гарантира, че ако инструкция p1 приключи преди q1, ще изчака и инструкция q1 да приключи с изпълнението си. След инструкция u.wait() в процес Q записваме инструкция t.signal(), която ни гарантира, че ще се сигнализира на семафора t за освобождаване на ресурс и ще може процес P да продължи нататък с изпълнението на своите инструкции. Аналогично е и за точки 2 и 4.

74. 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 работещите копия така, че да са изпълнени едновременно следните условия:
- в произволен момент от времето да работи най-много едно от копията
- работещите копия да се редуват във времето - след изпълнение на копие на P да следва изпълнение на копие на Q и обратно
- първоначално е разрешено да се изпълни копие на P

Решение:
За синхронизация между отделните процеси ще използваме два семафора - u и t, като семафор u ще инициализираме с начално отблокирано състояние, а семафор t ще инициализираме с начално блокиращо състояние:

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

```
process P     process Q
u.wait()       t.wait()
 p_1            q_1
 p_2            q_2
t.signal()    u.signal()
```

Като първи инструкции в процесите P и Q записваме u.wait() (в процес P) и t.wait() (в процес Q), като разликата е, че семафор u е с начално отблокирано състояние 1, а семафор t е с начално блокиращо състояние. Това означава, че през инструкция u.wait() може да премине само едно-единствено копие на процеса P, а през инструкция t.wait() не може да премине нито едно копие на процеса Q. От условието ни е казано, че първоначално е разрешено да се изпълни копие на P. Поради тази причина се допуска първо копие на процеса P до инструкциите p_1 и p_2. Останалите копия на процеса P са блокирани и са в опашката на семафора u. Копията на процеса Q също са блокирани и са в опашката на семафора t. След като допуснатото копие на процеса P приключи с изпълнението на инструкциите p_1 и p_2, то следва да се сигнализира на семафор t, за да може да бъде допуснато копие на процеса Q до инструкциите q_1 и q_2. По условие е казано, че работещите копия трябва да се редуват в ввремето - след изпълнение на копие на P да следва изпълнение на копие на Q и обратно. Поради тази причина се подава сигнал на семафор t, а не на семафор u - копията

на процеса P трябва да останат блокирани, докато някое копие на процеса Q не изпълни инструкциите q_1 и q_2. След като копие на процеса Q изпълни инструкциите, е необходимо да се подаде сигнал за семафор u - поради тази причина се добавя инструкция u.signal() под инструкция q_2. По този начин ще бъде допуснато копие на процеса P до инструкциите p_1 и p_2, останалите копия на процеса P ще са блокирани, както и останалите копия на процеса Q ще са блокирани. По този начин се редува изпълнението на работещите копия на процесите P и Q. В произволен момент от времето винаги ще има най-много едно работещо копие - първоначално ще бъде допуснато копие на процеса P до инструкциите p_1 и p_2, а останалите копия на процеса P ще бъдат блокирани и ще изчакват. Копията на процеса Q също ще са блокирани и ще изчакват, докато някое от копията на процеса P не подаде сигнал на семафор t. След като се подаде сигнал на семафор t, ще бъде допуснато само едно копие на процеса Q до инструкциите q_1 и q_2, останалите копия на процеса Q ще са блокирани. Блокирани ще бъдат и копията на процеса P. След като допуснатото копие на процеса Q приключи с изпълнението на инструкциите q_1 и q_2, ще се подаде сигнал на семафор u, който ще допусне само едно-единствено копие на процеса P и по този начин ще се редуват копията на процесите P и Q, както и винаги ще има само едно-единствено работещо копие - или копие на процеса P, или копие на процеса Q.

75. Процесите P и Q се изпълняват паралелно. Споделената променлива A има начална стойност 4. Променливата R е
локална за двата процеса.

```
process P    process Q
 R=A          R=A
 R=R+3        R=R+2
 A=R          A=R
```

Каква е стойността на A след изпълнението на процесите? Дайте обоснован отговор.

Отговор: Не може да се даде окончателен отговор каква е стойността на A след изпълнението на процесите, защото процесите P и Q се изпълняват паралелно и те не се синхронизират помежду си. Има няколко варианта:

1. Двата процеса P и Q да се изпълнят последователно, като процес P се изпълнява преди процес Q
Първоначално процес P да изпълни инструкцията си R=A, тоест прочита споделената променлива A, като неговата локална променлива R в случая придобива стойност 4. След това процесът P да изпълни инструкцията си R=R+3, като по този начин ще се промени единствено локалната му променлива R (тя ще стане 7), а споделената променлива A ще си остане със стойност 4. След това процес P да изпълни последната си инструкция -> A=R, като по този начин споделената променлива A ще има вече стойност 7. След това да започне работа процес Q. Той ще прочете също споделената променлива A, като запази стойността ѝ в своята локална променлива R -> R=A (в случая A е станала 7
след изпълнението на процес P, което означава, че локалната променлива на процес Q също ще е със стойност 7). Следва процес Q да изпълни и втората си инструкция -> R=R+2, като тук ще се промени само и единствено неговата локална променлива R (от първата си инструкция тя придобива стойност 7, а след изпълнение на тази инструкция придобива стойност 9). Последната инструкция на процеса Q е A=R, като по този начин се променя окончателно споделената променлива A -> тя вече е със стойност 9.

2. Двата процеса P и Q да се изпълнят последователно, като процес Q се изпълнява преди процес P

Първоначално процес Q да изпълни инструкцията си R=A, тоест прочита споделената променлива A, като неговата локална променлива R в случая придобива стойност 4. След това процесът Q да изпълни инструкцията си R=R+2, като по този начин ще се промени единствено локалната му променлива R (тя ще стане 6), а споделената променлива A ще си остане със стойност 4. След това процес Q да изпълни последната си инструкция -> A=R, като по този начин споделената променлива A ще има вече стойност 6. След това да започне работа процес P. Той ще прочете също споделената променлива A, като запази стойността ѝ в своята локална променлива R -> R=A (в случая A е станала 6 след изпълнението на процес Q, което означава, че локалната променлива на процес P също ще е със стойност 6). Следва процес P да изпълни и втората си инструкция -> R=R+3, като тук ще се промени само и единствено неговата локална променлива R (от първата си инструкция тя придобива стойност 6, а след изпълнение на тази инструкция придобива стойност 9). Последната инструкция на процеса P е A=R, като по този начин се променя окончателно споделената променлива A -> тя вече е със стойност 9.

3. Двата процеса P и Q да не се изпълнят последователно, а паралелно
От първите две точки - 1. и 2. споделената променлива A се получава с крайна стойност 9, като процесите P и Q протичат последователно - тоест, те се изчакват взаимно. Какво би се случило, ако двата процеса не се изпълнят последователно, а паралелно? Нека да погледнем следните ситуации:
Нека процес P прочете споделената променлива A -> R=A (в първата инструкция локалната променлива R ще придобие началната стойност на споделената променлива A -> 4). Нека сега и процес Q прочете споделената променлива A → R=A (в първата инструкция локалната променлива R ще придобие началната стойност на споделената променлива A -> 4).
Нека сега процес P изпълни инструкция R=R+3 (тогава локалната променлива ще придобие стойност 7, а споделената променлива A ще остане непроменена - тоест нейната стойност ще продължи да бъде 4). Нека сега и процес Q изпълни инструкция R=R+2 (тогава локалната променлива ще придобие стойност 6, а споделената променлива A ще остане непроменена - тоест нейната стойност ще продължи да бъде 4). Сега има два варианта:
- процес P да изпълни първи инструкцията A=R, като по този начин споделената променлива ще получи стойност 7. След това процес Q да изпълни и своята инструкция A=R. По този начин споделената променлива ще получи стойност 6. В този случай крайната стойност на променливата A е 6.

- процес Q да изпълни първи инструкцията A=R, като по този начин споделената променлива ще получи стойност 6. След това процес P да изпълни и своята инструкция A=R. По този начин споделената променлива ще получи стойност 7. В този случай крайната стойност на променливата A е 7.

Тъй като R е локална променлива и за двата процеса, това означава, че нейната промяна се случва само и единствено или в процес P, или в процес Q - тоест не е видимо за процес Q какво се случва с нея в процес P, както и обратното. Споделената променлива A не се променя пряко - първо се увеличава стойността на локалната променлива R, след което получената стойност се присвоява на споделената променлива A. Поради тази причина не могат да се появят други случаи, освен тези три.

Стигнахме до извода, че споделената променлива A след изпълнението на процесите може да има стойностите 9, 6 или 7. Това се случва в зависимост от изпълнението на процесите.

76. Опишете разликата между синхронни и асинхронни входно-изходни операции. Дайте примери за програми, при които се налага използването на асинхронен вход-изход.

Лекции:
Разлика между синхронен/асинхронен канал
Когато вход/изходът е синхронен, процесът, който извиква операция за вход/изход - четене или писане, може да бъде приспан - може да блокира, може да влезе в такова състояние, процесът продължава да е блокиран. Ако нашата програма през интернет прави връзка с друга програма и си чатим, като кажем искам да прочета какво ми казва човекът отсреща, нашата програма ще бъде приспана при синхронния вход/изход, докато отсрещната страна не ми прати съобщение, защото няма смисъл да заема процесорно време, защото само тя изчаква, и преминава към спящ режим. При асинхронните операции това не може да се случи. Трябва процесът да продължи да работи, независимо, че входно/изходната операция не може да завърши.

Какви проблеми има в поведението на програмите?
Проблемът е, че като кажем read на някакви данни, след като процесът не може да бъде приспан, а пък данни може да не се появят в момента (те могат да пристигнат по-късно) тази команда веднага ще върне резултат, ама резултатът може да не са прочетени данни и това трябва по някакъв начин да се идентифицира. При нормалния read данните ще бъдат прочетени в някакъв участък от паметта, който сме посочили – някаква променлива или друг информационен обект. При асинхронния вход/изход командата ще завърши, но трябва по някакъв начин да ни уведоми дали има данни, или няма данни в канала, който четем. Това състояние се идентифицира. Трябва да знаем, че работим в асинхронен режим и трябва веднага след read да провери по някакъв начин дали има входни данни, или ги няма. Евентуално след време да повтори операцията - да свърши някаква друга работа и след това да се върне на тези действия. Дава се възможност на програмата едновременно от няколко файлови дескриптора да извършва входно/изходни операции. Ако даден файлов дескриптор не е готов, програмата да не блокира там, да не се приспи там, а да използва друг файлов дескриптор, за да чете от него. Когато сме в асинхронен вход/изход тези команди за четене и писане се изпълняват моментално, не блокират процеса, обаче във вход за грешка се появява информация, че има грешка и кодът за грешката е съответното съобщение EAGAIN - липса на готовност операцията веднага да ни върне данните, които искаме да четем или да запишем. Има и други кодове за грешка съответно.

Идеята е, че със синхронния режим ще работим по по-простия, по-обичайния начин, който е естествен за входно-изходни канали и може да доведе до приспиване на процеса.
Когато работим с няколко комуникационни канала и искаме паралелно да следим по кой идват данни и по кой не идват, тогава можем да работим с асинхронен режим и да следим кой файл разполага с данни и кой не. Този режим се използва в следните типични ситуации - имаме диалогова интерактивна програма, която искаме да обслужва различни входно/изходни устройства и понеже те са различни, искаме да разберем дали потребителят е натиснал клавиш, дали е натиснал мишка и т.н. Искаме нашата програма да реагира динамично - когато има действие с мишката, то тя да разбере, че това е мишката и съответно да предприеме действия - примерно да маркира текст. Такава ситуация за асинхронен вход/изход може да се заобиколи: Да направим отделен процес, който да чете само от мишката събития и друг процес, който да чете само от клавиатурата, и тези два процеса да пишат в общ комуникационен канал, а друг такъв да чете от него. По този начин се подреждат във времето байтовете, които единият или другият процес изпращат.

Друга често срещана ситуация е при работа с интернет:
Когато имаме програма Сървър и тя работи с много клиенти - те може да са в интернет или в самата изчислителна система. Сървърът ще има достъп до някакви данни. Ако искаме това да е един процес за икономия на ресурси и за синхронизиране на действията на клиентите и

файловете, които те използват, тогава този сървър трябва да има комуникационни канали, изградени към клиентите и файловете, които те ползват. Той трябва да реагира пъргаво на всички - този клиент, който праща данни, сървърът трябва да разбере за изпратените данни и да му отговори, този клиент, който нищо не прави и не е подал данни, сървърът не трябва да му връща нищо като отговор. Когато е в асинхронен режим, той може да провери всеки от входно-изходните канали, дали има готови данни, и ако има, да ги прочете и да върне отговор.

При синхронна входно-изходна операция системното извикване може да доведе до приспиване (блокиране) на потребителския процес, поръчал операцията.
При нормално завършване потребителският процес разчита на коректно комплектоване на операцията - четене/запис на всички предоставени/поръчани данни във/от входно-изходния канал, или цялостно изпълнение на друг вид операция (примерно, изграждане на TCP връзка).

При асинхронна входно-идходна операция системното извикване не приспива (не блокира) потребителския процес, поръчал операцията.
При невъзможност да се комплектова операцията, ядрото връща управлението на процеса със специфичен код на грешка и друга информация, която служи за определяне на степента на завършеност на операцията. Потребителският процес трябва да анализира ситуацията и при нужда да направи ново системно повикване по-късно с цел да довърши операцията.

Използването на асинхронни операции позволява на един процес да извършва паралелна комуникация по няколко канала с различни устройства или процеси, без да бъде блокиран в случай на липса на входни данни, препълване на буфер на изходни данни или друга ситуация, водеща до блокиране.
Типични примери:
1) Когато използваме WEB-browser, той трябва да реагира на входни данни от клавиатура и мишка, както и на данните, постъпващи от интернет, т.е. на поне 3 входни канала. Браузърът проверява чрез асинхронни опити за четене по кой от каналите постъпва информация и реагира адекватно
2) Сървър в интернет може да обслужва много на брой клиентски програми, като поддържа отворени TCP връзки към всяка от тях. За да обслужва паралелно клиентите, сървърът трябва да ползва асинхронни операции, за да следи по кои връзки протича информация и кои са пасивни

Когато програмата ползва асинхронни операции и никой от входно-изходните канали не е готов за обмен на данни, тя има нужда от специален механизъм за предоставяне на изчислителния ресурс на останалите процеси.
Обикновено в такива случаи програмата се приспива сама за кратък период от време (в UNIX това става с извикване на sleep(), unsleep() или nanosleep()).

77. 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 p2 на всяко от работещите копия да се изпълни, след като инструкция p1 е завършила изпълнението си в поне 3 работещи копия.

Упътване: Освен семафори, ползвайте и брояч.

Решение:
За решението на проблема ще използваме семафор u и брояч cnt, като семафорът ще е инициализиран с начално блокиращо състояние, а броячът ще е с начална стойност 0:

```
semaphore u
u.init(0)
int cnt=0
```

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

Мое решение:

```
process P
 p_1
 cnt=cnt+1
 if (cnt>=3)
   u.signal()
 u.wait()
 p_2
 cnt=cnt-1
```

Друго решение, което изглежда по-добре

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

Мое решение:
Нека си представим следната ситуация: Пристига копие на процеса P, то преминава през инструкцията p_1, завършва я, след което увеличава брояча с 1 (в случая броячът ни отбелязва колко работещи копия са завършили изпълнението на инструкция p_1). Броячът от 0 става 1, след което това копие не влиза в случая if (cnt>=3), защото cnt е 1, и отива при инструкция u.wait(), където изчаква да бъде допуснат, защото по условие имаме, че трябва да има поне 3 работещи процеса, които са завършили изпълнението на инструкция p_1, преди някое от копията да бъде допуснато до изпълнение на инструкция p_2. След това идва още едно копие на процеса P, изпълнява инструкция p_1, броячът cnt се увеличава с 1 и вече става 2, отново прескача блока if и отива при инструкция u.wait() да изчака реда си. Като се появи третото копие, то също преминава през инструкция p_1, завършва изпълнението й, след което увеличава брояча cnt с 1 (cnt вече е 3) и сега вече това копие може да влезе в случая if (cnt >= 3), защото по условие имаме че трябва поне 3 работещи копия на процеса P да са завършили изпълнението на инструкция p_1. След като третото копие влезе в този случай, той минава през инструкция u.signal(), като по този начин сигнализира на семафора, че може да пусне едно копие до инструкция p_2, след което отива при инструкция u.wait() да изчака реда си, заедно с останалите чакащи копия. Нека сега си представим, че пристига четвърто копие на процеса P. То минава през инструкция p_1, завършва я, след което увеличава брояча с 1 (той е бил 3, а сега става 4). Това копие също може да влезе в случая if (cnt >= 3), след което да премине през инструкция u.signal() и да подаде сигнал на семафор u, след което той да допусне още едно копие до инструкция p_2. И така този процес продължава по този начин. След като някое от копията завърши изпълнението на инструкцията p_2, е необходимо да се намали броячът с 1, защото по този начин един вид се сигнализира, че имаме копие, което вече е приключило работата си и повече няма да е работещо. Защо не намаляваме брояча след u.signal(), а го намаляваме след инструкция p_2? Защото в условието се казва, че до инструкция p_2 се допускат копия, след като p_1 е завършила изпълнението си в поне 3 работещи копия. Това копие, което е допуснато до инструкция p_2 е все още работещо и ако броячът се намали, преди това копие да е завършило изпълнението на инструкция p_2, то тогава няма да е изпълнено коректно условието - ще е лъжа да се намали броячът cnt при положение, че все още има работещо копие в инструкция p_2.

Друго решение:
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 позолява на всички копия да изпълнят втората си инструкция.

От другото решение защитата на брояча cnt трябва да се вземе под внимание за моето решение. За другата част не съм напълно сигурна -> в условието се казва да се преминава към инструкция p_2, ако ПОНЕ 3 работещи копия са
завършили изпълнението в инструкция p_1, а според мен краят на процеса идва след приключване на инструкция p_2, там би трябвало да се намали броячът и съответно отново да се защити.

78. Опишете реализацията на комуникационната тръба (pipe) чрез семафори. Предполагаме, че тръбата може да съхранява до n байта, подредени в обикновена опашка. Тръбата се ползва от няколко паралелно работещи изпращачи/получатели на байтове. Процесите изпращачи слагат байтове в края на опашката, получателите четат байтове от началото на опашката.

Отговор:
Проблемът producer-consumer (известен също като проблем с ограничен буфер) е класически пример за многопроцесов синхронизиран проблем. Проблемът описва два процеса - производител и потребител, които споделят общ буфер с фиксиран размер, използван като опашка. Задачата на производителя е да генерира данни, да ги постави в буфера и да започне отново. В същото време потребителят консумира данните (т.е., премахвайки ги от буфера), по едно парче. Проблемът е да се гарантира, че производителят няма да се опита да добави данни в буфера, ако е пълен, и че потребителят няма да се опита да премахне данни от празен буфер.

Решението за производителя е или да заспи, или да изхвърли данни, ако буферът е пълен. Следващият път, когато потребителят премахне елемент от буфера, той уведомява производителя, който започва да попълва отново буфера. По същия начин потребителят може да заспи, ако намери буфера празен. Следващият път, когато производителят постави данни в буфера, той събужда спящия потребител. Решението може да бъде постигнато чрез комуникация между процесите, обикновено използвайки семафори. Неправилното решение може да доведе до задънена улица, при която и двата процеса чакат да се събудят (deadlock). Проблемът може да се обобщи и да има множество производители и потребители.

Производителят може да произведе артикул и може да постави в буфера. Потребителят може да избира артикули и да ги консумира. Трябва още да гарантираме, че когато производителят поставя артикул в буфера, потребителят не трябва да консумира нито един продукт. В този проблем буферът/тръбата е критичната секция.

За да разрешим този проблем, се нуждаем от два броя семафори - пълен и празен. „Full“ следи броя на елементите в буфера по всяко време, а „Empty“ следи броя на незаетите слотове.

Примерно решение със семафори:
mutex = 1
full = 0
empty = n

Решение за производителя:

```
do
{
  //produce an item
  wait(empty)
  wait(mutex)

  //place in buffer

  signal(mutex)
  signal(full)

}while(true)
```

Решение за консуматора:

```
do
{
    wait(full)
    wait(mutex)

     //remove item from buffer

    signal(mutex)
    signal(empty)

   //consumes item

}while(true)
```

Първоначално дефинираме empty да е n (колкото е размерът на тръбата), защото в началото имаме точно n свободни места (все още няма поставени байтове в тръбата). Дефинираме full да е 0, защото в началото нямаме нито едно заето място. mutex е 1, за да може до буфера/тръбата да бъде допуснат един-единствен производител.

Когато производителят произведе артикул, стойността на „empty“ се намалява с 1, защото сега ще бъде попълнен един слот. Стойността на mutex също се намалява, за да попречи на потребител или на друг производител да получи достъп до буфера. Сега производителят е поставил артикула и по този начин стойността „full“ се увеличава с 1. Стойността на мютекса също се увеличава с 1, тъй като задачата на производителя е изпълнена и потребител или друг производител може да получи достъп до буфера.

Първоначално всички потребители са блокирани, защото началната стойност на full е 0. След като има поне 1 поставен артикул в буфера/тръбата и стойността на мютекса е 1, тоест няма друг производител или потребител, който да я ползва, то тогава може да бъде допуснат потребителят до ресурса. Тъй като потребителят премахва артикул от буфер, следователно стойността „full“ се намалява с 1 и стойността мутекс също се намалява, така че производител или друг потребител да не може да получи достъп до буфера в този момент. Сега потребителят е консумирал артикула, като по този начин увеличава стойността на „empty“ с 1. Стойността на мютекса също се увеличава, така че производител или друг потребител да има достъп до буфера сега.