

**Захарије Радивојевић**

**Игор Икодиновић**

**Зоран Јовановић**

# **КОНКУРЕНТНО И ДИСТРИБУИРАНО ПРОГРАМИРАЊЕ**

**Друго издање**

Академска мисао  
Београд, 2018.

Захарије Радивојевић, Игор Икодиновић, Зоран Јовановић

## КОНКУРЕНТНО И ДИСТРИБУИРАНО ПРОГРАМИРАЊЕ

### Друго издање

Рецензенти  
Проф. др Јелица Протић  
Проф. др Драган Милићев

Издаје и штампа  
АКАДЕМСКА МИСАО  
Београд

Тираж  
300 примерака

ISBN 978-86-7466-724-8

НАПОМЕНА: Фотокопирање или умножавање на било који начин или поновно објављивање ове књиге у целини или у деловима - није дозвољено без изричите сагласности и писменог одобрења издавача

## Предговор

Област конкурентног и дистрибуираног програмирања се већ дужи низ година проучава у оквиру редовне наставе на Електротехничком факултету у Београду на одсеку за рачунарску технику и информатику и на одсеку за софтверско инжењерство. Проучавањем ове области прикупљен је и анализиран већи број практичних проблема синхронизације и комуникације. Ови проблеми су јединствени и омогућавају генерализацију честих проблема у пракси. Сваком од проблема носи јединствено име ради лакшег праћења, али у упоређивања са постојећим решењима из ове области. Неки проблеми су тако конципирани да на шаљив начин обраде доста сложене механизме синхронизације и комуникације.

Као полазна основа за формирање ове књиге послужила је књига „Конкурентно програмирање: Теоријске основе са збирком решених задатака“ аутора Игора Икодиновића и Зорана Јовановића. Та књига је допуњена и проширења новим проблемима и областима, уочене грешке су исправљене, али је начин излагања и обраде задатака остао исти.

Први део књиге је посвећен конкурентном програмирању помоћу дељених променљивих. Други део књиге обухвата област дистрибуираног програмирања. Трећи део је посвећен моделу програмирања коришћењем виртуелних простора. Четврти део се бави програмским нитима, као координационом моделу који се данас најчешће примењује код писања конкурентних програма. Пети део књиге се односи на мрежно програмирање.

Програмске парадигме су представљене на један од три начина: 1) коришћењем постојећих програмских библиотека и језика за конкурентно програмирање, 2) проширивањем секвенцијалних програмских језика одговарајућим синтаксним елементима и 3) преко формалних програмских модела. Избор начина на који су одређене парадигме представљене заснован је пре свега на критеријумима њихове једноставности, разумљивости и општости. У случају да пракса захтева примену одговарајућих парадигми коришћењем неких других имплементација од оних одабраних за њихов приказ у књизи, у већини случајева је могуће извршити директну синтаксну транслацију. Разумевање основних концепата из књиге ће чак и у случајевима када директна синтаксна транслација није могућа тај посао знатно олакшати.

Од постојећих програмских језика за конкурентно и дистрибуирано програмирање у оквиру ове књиге су коришћени: *Ada* за демонстрацију механизма рандевуа, *Java* за демонстрацију концепта програмских нити, као конкретан пример имплементације монитора, и као пример мрежног програмирања користећи размену порука и удаљене позиве метода и *CON/C* за демонстрацију парадигме асинхроног прослеђивања порука са индиректним именовањем процеса помоћу портова. У другим случајевима су коришћена проширења постојећих секвенцијалних програмских језика: проширени *Pascal* за демонстрацију концепта семафора, условних критичних региона и монитора и *C-Linda* за демонстрацију програмирања помоћу виртуелних простора. Од формалних програмских модела, *BSP* је коришћен за демонстрацију парадигме прослеђивања порука путем јавног емитовања, а *CSP* за демонстрацију парадигме прослеђивања порука са синхронним слањем и пријемом.

Један од проблема код упознавања са облашћу конкурентног и дистрибуираног програмирања је што читалац мора значајан део своје пажње да усмери на учење синтаксе и семантике везаних за имплементације разних парадигми. Да би се тај

проблем ублажио, у уводном делу сваког поглавља је поред прегледа карактеристика одговарајуће парадигме, дат и опис и анализа коришћене синтаксе. Након уводног дела дати су и задаци. Задаци који се налазе на почетку обично су изабрани тако да демонстрирају неке од главних особина парадигме и да покажу како се помоћу ње решавају типични проблеми. Они су посебно детаљно размотрени, како са теоријског тако и са практичног аспекта, јер представљају основ за разумевање материје у оквиру поглавља. Иза њих следе задаци који су по природи нешто сложенији и чија решења обично захтевају дубљу и дуготрајнију анализу, синтетишући стечено знање на вишем нивоу и дајући адекватну представу о величини и сложености реалних проблема. Након поједињих задатака и на крају поглавља се могу наћи и задаци за самосталан рад. Препоручује се њихово решавање као најбоља вежба за проверу усвојеног знања.

Књига је писана на ћирилици, што је допринос очувању нашег језика и писма у области где су енглески језик и латиница често доминантни. Није се, међутим, могло избеги коришћење специфичних страних стручних израза и назива. У том смислу коришћена терминологија одражава затечено стање у овој области код нас. Где је то било сврсисходно и могуће коришћени су термини нашег језика, уз навођење одговарајућих израза који се користе у страној литератури. Програми и примери су писани са именима променљивих, процедура и програмским коментарима који се ослањају на речи енглеског језика, јер је то данас већ стандардна пракса уведена због потребе за комуникацијом у широј стручној јавности. Ово не би требало да има утицаја на разумљивост решења, с обзиром да су она праћена детаљним објашњењима.

Захвальујемо рецензентима – наставницима ЕТФ-а – Јелици Протић и Драгану Милићеву и колеги Милошу Глигорићу на сугестијама којима су помогли да се текст књиге поправи и допуни пре издавања.

У Београду, фебруара 2008. године

## Предговор II издању

У овом издању су постојеће области проширене и допуњене, и исправљене су уочене грешке. Захвальујемо студентима који су јавили уочене грешке и неконзистентности у тексту као и колегиници Санји Делчев на сугестијама које су помогле да се текст књиге поправи и допуни.

У Београду, фебруара 2018. године

# Садржај

Увод.....	1
<b>Програмирање помоћу дељених променљивих .....</b>	<b>7</b>
<b>Семафори.....</b>	<b>8</b>
Проблем критичне секције.....	10
Произвођач и потрошач: условна синхронизација процеса .....	12
Произвођачи и потрошачи: комуникација помоћу кружног бафера .....	15
Филозофи за ручком.....	30
Читаоци и писци .....	46
Недељиво емитовање.....	56
Људождери за ручком.....	63
Медвед и пчеле .....	66
Одгајање птића.....	68
Брига о деци.....	71
Синхронизација на баријери.....	78
Вожња тобоганом .....	79
Изградња молекула воде.....	84
Проблем преласка реке .....	92
Студентска журка.....	96
<b>Условни критични региони .....</b>	<b>99</b>
Проблем критичне секције .....	101
Произвођач и потрошач: синхронизација процеса .....	103
Мост који има само једну коловозну траку.....	107
Филозофи за ручком.....	112
Анализа различитих варијанти решења проблема читалаца и писаца .....	116
Претрага-уметање-брисање .....	120
Дељени рачун .....	126
Нервозни пушачи .....	130
Проблем избора.....	136
Проблем паркинг-а са заглављивањем .....	141
<b>Монитори.....</b>	<b>144</b>
Резервација карата .....	147
Читаоци и писци .....	153
Тајмер .....	160
Улазак у авион .....	163
Улазак у школу .....	164
Кружни FIFO бафер .....	165
Сакупљање гајбица .....	167
Прање веша .....	169
Филозофи за ручком .....	173
Недељиво емитовање .....	178
Алокација ресурса .....	183
Кружни ток .....	185
Т раскрсница .....	188
<b>Дистрибуирано програмирање .....</b>	<b>195</b>

<b>Увод у програмирање прослеђивањем порука.....</b>	<b>197</b>
Синхrona, асинхrona и условна комуникација .....	198
Читаоци и писци .....	200
Произвођачи и потрошачи .....	206
Филозофи за ручком.....	209
Вожња тобоганом .....	216
Игра живота.....	222
Прстен.....	226
Јавно емитовање.....	230
Откривање топологије.....	233
Двоелементни бафер .....	235
Комуникација са поузданим везама.....	237
<b>Синхроно прослеђивање порука (CSP).....</b>	<b>240</b>
Реализација семафора .....	244
Реформатирање текста ( <i>Conway's Problem</i> ).....	245
Пријем, обрада и слање низа знакова.....	246
Потпрограми: Остатак при дељењу.....	248
Рекурзија: Факторијел .....	249
Ератостеново сито.....	250
Множење матрица .....	252
Читаоци и писци .....	254
Произвођачи и потрошачи.....	258
Филозофи за ручком.....	259
Бинарно стабло.....	262
Бинарно стабло - конкурентно претраживање.....	267
Скупови.....	273
Израчунавање интеграла.....	285
Изградња молекула воде.....	287
Дељени рачун .....	289
<b>Јавно емитовање (BSP) .....</b>	<b>292</b>
Стек.....	295
Скуп.....	296
Филозофи за ручком.....	300
Читаоци и писци .....	302
Радио такси .....	304
Гониометарски систем .....	306
<b>Асинхроно прослеђивање порука (CON/C) .....</b>	<b>308</b>
Бафер раздјелник.....	311
Произвођачи и потрошачи: комуникација помоћу кружног бафера .....	312
Промена конфигурације: звезда у прстен .....	313
Бидирекциони прстен типа <i>FDDI</i> .....	314
Клијенти и сервери .....	316
Обрада података .....	321
Филозофи за ручком.....	323
Читаоци и писци .....	325
<b>Рандеву (Ada) .....</b>	<b>327</b>
Селективна наредба .....	332
<i>Timeout</i> опција код селективне наредбе .....	334
Реализација рандеву помоћу примитива за слање и пријем порука.....	335
Једноелементни бафер – проблем коректног завршетка процеса.....	338
Монитор .....	340
Кружни <i>FIFO</i> бафер .....	345
Произвођачи и потрошачи: комуникација помоћу кружног бафера .....	348
Читаоци и писци .....	351
Филозофи за ручком.....	354
Нервозни пушачи .....	356

## **Виртуелни простори ..... 360**

### **Простор торки (*C-Linda*)..... 361**

Филозофи за ручком.....	363
Произвођачи и потрошачи .....	364
Клијенти и сервери – обрада захтева по <i>FIFO</i> принципу (случај са једним сервером) .....	365
Клијенти и сервери – обрада захтева по <i>FIFO</i> принципу (случај са више сервера) .....	367
Клијенти и сервери – селекција сервера по <i>round robin</i> редоследу .....	371
Нервозни пушачи.....	378
Читаоци и писци .....	382
Проблем избора.....	384
Проблем лифтова.....	385
Заједнички тоалет.....	388
Проблем пијаних филозофа .....	390
Израчунавање интеграла.....	392
Проблем тела у гравитационом пољу .....	394

## **Програмске нити ..... 398**

### **Java .....** 400

Семафор.....	408
Произвођач и потрошач: условна синхронизација процеса .....	414
Произвођач и потрошач .....	421
Филозофи за ручком.....	428
Мост који има само једну коловозну траку .....	432
Читаоци и писци .....	438
Берберин који спава .....	447
Вожња тобоганом .....	450
Вожње аутобусом .....	453
Брига о деци.....	457
Јавни тоалет .....	460
Деда Мраз .....	464
Међусобно искључивање - <i>spin locks</i> .....	469
Едитор – рад са корисничким интерфејсом .....	486

## **Мрежно програмирање ..... 493**

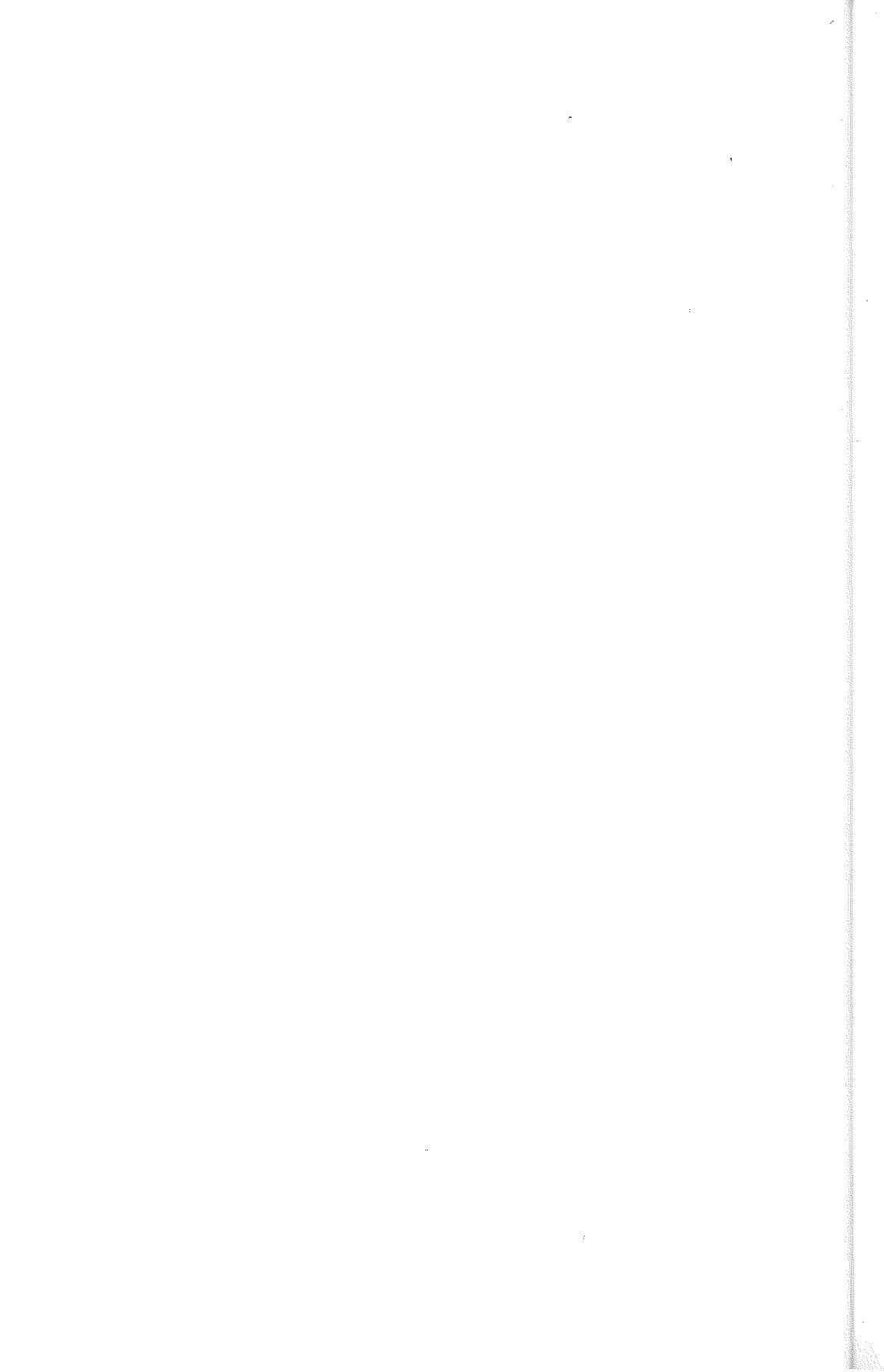
### **Java - Net..... 494**

Ћаскање .....	501
Клијент-сервер архитектура - сервери .....	504
Клијент-сервер архитектура - комуникација.....	513
Клијент-сервер архитектура - протоколи.....	519
Клијент-сервер архитектура - примери.....	522

### **Java - RMI..... 528**

Дељени рачун .....	533
Игра живота.....	537

## **Литература ..... 541**



**Сваки као што је примио благодатни дар, њиме служите једни другима,  
као добри управитељи разноврсне благодати Божије.**

*Прва саборна йосланица свештoї апостола Петра, глава 4*



## Увод

Развој технологија везаних за интернет и локалне рачунарске мреже и појава нових рачунарских и микропроцесорских архитектура омогућили су да конкурентно програмирање превазиђе примене у чисто научним и истраживачким областима или великим корпоративним срединама. Данас се ова врста програмирања широко користи у великом броју комерцијалних апликација и на различитим рачунарским платформама - од великих паралелних рачунара и рачунарских кластера, преко мањих мултипроцесорских сервера и радних станица, све до персоналних и преносних рачунара. Велики број савремених оперативних система, као што су *UNIX* (у разним варијантама: *GNU/Linux*, *Android*, *Sun Solaris*, *SCO System V*, *SGI IRIX*, *IBM AIX*, *HP-UX* итд.), *BeOS*, *Beowulf*, *Windows* и др. подржавају различите облике конкурентног извршавања на разним типовима рачунара. Одговарајући развојни алати су такође достигли висок ниво, тако да данас постоји велики број програмских језика, преводилаца и софтверских библиотека који подржавају конкурентно програмирање. На принципима конкурентног рада заснивају се и многе модерне софтверске апликације - од оних које се користе у специјализованим научно-истраживачким и инжињерским применама, преко веб сервера и система за претрагу великих база података, до разних мрежних апликација (које обухватају широк распон од мрежних рачунарских игара, па све до софтвера у оквиру великих истраживачких пројеката који је дизајниран тако да користи слободно процесорско време рачунара на интернету). Може се рећи да смо на прагу једног новог доба које карактерише прелазак од секвенцијалног ка конкурентном програмирању.

Напредак у области архитектуре мултипроцесорских и дистрибуираних рачунарских система и рачунарских мрежа природно је пратила и потреба за развојем формалних програмских метода који би омогућили њихово ефикасно коришћење. Од почетка шездесетих година двадесетог века, када је почeo развој ове области, па до данас, формулисан је велики број парадигми конкурентног програмирања оличених у разним програмским језицима и библиотекама. Притом се може уочити јака веза између начина писања програма (програмске парадигме) и типа извршне платформе. То је последица чињенице да програмски преводиоци нису у стању да генеришу ефикасан машински код за различите рачунарске архитектуре<sup>1</sup> на основу извornog кода писаног помоћу једне исте програмске парадигме. Нове архитектуре су често изискивале стварање нових, њима прилагођенијих форми конкурентног програмирања.

Одвајање програмске парадигме од извршне платформе ни до данас није постигнуто. Са једне стране постоје асемблерски програми који су потпуно зависни од извршне платформе и имају фиксиран алгоритам рада кога програмски код само имплементира на одговарајућој рачунарској архитектури. Задатак преводиоца (асемблера) је сведен на пуку синтаксну транслацију асемблерских наредби у одговарајуће машинске инструкције. Са друге стране се налазе програми који су потпуно независни од извршне платформе (тј. одређеног начина имплементације) и не специфицирају један фикси алгоритам рада. Одговарајући преводилац би у том случају имао задатак да самостално изналази алгоритме и конструише машинске програме у складу са типом извршне платформе и другим критеријумима (нпр. врста проблема, перформансе и сл.). Чињеница је да данас познати програмски преводиоци имају могућности које се

<sup>1</sup> Овде се мисли на мултипроцесорске архитектуре. Код њих је поред архитектуре самих процесора од виталног значаја и меморијски подсистем, укључујући и интерконекциону мрежу, који је далеко комплекснији и са још већим утицајем на перформансе него меморијски подсистем код једнопроцесорских система.

налазе негде између ове две крајности. Они омогућавају одређени ниво независности програма од извршне платформе, али то и даље није на тако високом нивоу да би преводилац могао генерисати ефикасан код за различите типове мултипроцесорских архитектура. Зато је у пракси и даље присутан велики број парадигми конкурентног програмирања.

## Имплементација парадигми конкурентног програмирања

Подршка за разне програмске парадигме може бити реализована на различитим нивоима:

- 1) на нивоу архитектуре, тј. наредби машинског језика (нпр., одређене архитектуре подржавају хардверски имплементиране *LOCK/UNLOCK* или *BARRIER* примитиве)
- 2) на нивоу оперативног система (нпр., оперативни систем *GNU/Linux* у свом језгру има ефикасну подршку за *preemptive multitasking*<sup>2</sup>, омогућавајући једноставну и ефикасну имплементацију апликација са програмским нитима)
- 3) на нивоу програмских преводилаца за одговарајуће програмске језике (нпр., у оквиру програмског језика *Java* имплементирана је парадигма монитора)
- 4) на нивоу програмских библиотека (нпр., популарна библиотека *MPI* имплементира примитиве за програмирање путем прослеђивања порука)

У зависности од тога на ком нивоу је имплементирана подршка за неку програмску парадигму зависиће перформансе, преносивост и лакоћа развоја и одржавања одговарајућих програма. Програми писани на машинском језику су обично веома ефикасни, али очито нису преносиви и тешки су за развој и одржавање. Насупрот томе, ако је подршка за програмску парадигму имплементирана на нивоу библиотеке за неки програмски језик, ефикасност можда није максимална (мада је обично сасвим прихватљива), али су програми преносиви и релативно лаки за развој и одржавање, с обзиром на виши ниво апстракције коришћених примитива; ове друге две особине обично имају превагу над првом у случају да разлика у перформансама није велика.

## Терминологија

У литератури се могу срести различити појмови везани за област конкурентног програмирања који су често недовољно јасно дефинисани или се користе са различитим значењем у различитим контекстима. Зато ћemo овде дефинисати неке основне појмове.

**Паралелно извршавање** означава истовремено извршавање више рачунарских операција, секвенци операција, програма или делова једног програма.

**Паралелни програм** је сваки програм који користи посебну синтаксу за означавање делова програмског кода који се могу извршавати паралелно.

**Паралелни рачунарски систем** је сваки рачунарски систем који је у стању да истовремено извршава два или више делова једног (или више) програма.

<sup>2</sup> *Preemption* (енг.) значи могућност преузимања процесора од стране другог процеса, без знања или експлицитног захтева процеса који је изгубио процесор. *Preemptive multitasking* је начин рада оперативног система рачунара који подразумева коришћење неког специфичног критеријума приликом одлучивања колико дуго одређени процес може да има контролу над системом, пре него што се контрола преда другом процесу. Најчешћи критеријум који се користи у пракси је протекло време (овакав начин рада се зове *time sharing* или *time slicing*). Неки оперативни системи омогућују да одређени послови могу имати виши приоритет од других; пословима вишег приоритета се контрола предаје чим се иницирају (ако нема процеса још вишег приоритета) и/или им се додељују дужи временски интервали (*time slices*).

**Секвенцијално извршавање** је такво извршавање где се следећа рачунарска операција или програмска наредба извршава тек након што је претходна завршена, у складу са редоследом који је задат програмом.

**Секвенцијални йројрам** је програм код кога постоји само један секвенцијалан ток извршавања у времену. То значи, без обзира на каквој рачунарској платформи се програм извршава, да ће се, када су улазни параметри програма исти, увек извршавати исте наредбе, истим јединственим редоследом који је задат извornим програмом.

Ток извршавања *йројрама* (*execution flow*), *йројрамски Шок* (*program flow*), *инструкцијски шок* (*instruction flow*) или *ниш шока коншроле* (*thread of control*), како се још зове, је назив за скуп наредби програма које се извршавају одређеним секвенцијалним редоследом. Програм може имати један или више програмских токова. Секвенцијални програм има само један ток извршавања у времену, мада може имати више статичких токова извршавања. На пример, свако условно гранање у програму дели одговарајући статички ток на два, али ће током извршавања програма само један од њих бити изабран, тј. постојаће само један ток извршавања у времену. Паралелни програм може имати како више статичких токова извршавања, тако и више програмских токова који се могу извршавати истовремено.

Теоријски гледано, могуће је конструисати такав програмски преводилац који би и секвенцијалне програме према неком алгоритму преводио тако да се извршавају паралелно на одговарајућем рачунарском систему. И обратно, као што паралелно извршавање у крајњој линији не мора подразумевати постојање паралелних програма, тако ни паралелни програми не морају подразумевати паралелно извршавање. Данас постоје преводиоци који омогућавају паралелно извршавање секвенцијалних програма, али само на нивоу паралелног извршавања машинских инструкција (пример – извршавање секвенцијалних програма на суперскаларним процесорима). Други случај, где се секвенцијално извршавају паралелни програми, често се среће у једнопроцесорским системима који имају *preemptive multitasking*, где оперативни систем наизменично различитим процесима додељује део времена на централном процесору.

**Конкурентно извршавање** подразумева извршавање више програмских токова једног програма тако да они напредују у времену (бар два од њих), али се они не морају обавезно извршавати истовремено (што би био случај код паралелног извршавања).

Мада појмови конкурентног и паралелног извршавања звуче слично, међу њима постоји разлика. Паралелно извршавање подразумева да је извршна платформа паралелни рачунарски систем на којем се заиста могу истовремено извршавати делови паралелног програма, док се појам конкурентног извршавања може применити и на једнопроцесорским системима. Термин **конкурентни йројрам**, који би означавао програм код кога се два или више програмских токова конкурентно извршавају, је заправо синоним за паралелни програм, јер би разлика била само у начину њиховог извршавања и извршној платформи, а не у синтакси и семантици самог програма. Као што смо већ раније констатовали, програм и извршна платформа су различите категорије и спаја их, односно дели, програмски преводилац. На сличан начин, термин **конкурентно йројрамирање** би био синоним за паралелно програмирање, под условом да он не подразумева извршну платформу. Због веће општости, у оквиру ове књиге ћемо користити термине конкурентно извршавање и конкурентно програмирање, јер они могу да подразумевају било какву извршну платформу. Појам **конкурентност** означава потенцијал за паралелизам који постоји у програму, односно статички (програмом) дефинисани потенцијални паралелизам у извршавању.

Забуну често уносе и хардверски термини који прате новије микропроцесоре. Користећи тзв. инструкцијски паралелизам (*instruction parallelism*) у програму или тзв.

паралелизам фине грануларности (*fine grain parallelism*), процесори са проточном (*pipelined*) архитектуром, као и суперскаларни (*superscalar*) процесори, омогућавају истовремено извршавање више фаза неколико различитих машинских инструкција. Реч је, међутим, о врсти паралелног извршавања секвенцијалних(!) програма, тако да микропроцесори овог типа немају директан утицај на парадигме конкурентног програмирања. Овакви процесори спадају у тип *SISD* (*Single Instruction Single Data*) архитектура, где свака машинска наредба садржи и податке (операнде) и одговарајући инструкцијски код једне операције која оперише над тим подацима.

Данас се често срећу и процесори са уgraђеном подршком за тзв. *SIMD* (*Single Instruction Multiple Data*) процесирање, где једна иста машинска инструкција специфицира паралелно извршавање исте операције над скупом различитих података. У овом случају је реч о хардверској подршци паралелном програмирању, што подразумева и постојање експлицитних синтаксних елемената у оквиру програмских парадигми које се користе за њихово програмирање. Парадигма програмирања прилагођена оваквој архитектури зове се *data parallel programming*.

*SMT* (*Simultaneous Multi-Threading*) процесори су способни за истовремено извршавање више програмских нити (токова извршавања), што подразумева и коришћење одговарајућих парадигми конкурентног програмирања. Ови процесори спадају у *MIMD* (*Multiple Instruction Multiple Data*) архитектуре. Данас се већ могу наћи комерцијални микропроцесори са оваквом архитектуром. Њихова предност је што користе паралелизам веће грануларности (*coarse grain parallelism*) од инструкцијског паралелизма. Инструкцијски ниво паралелизма ограничен је зависношћу између података и просечном фреквенцијом скокова у програму. Први проблем се ублажава применом техника оптимизације кода (нпр. софтверске проточности - *software pipelining*), а други техникама предвиђања скока (*branch prediction*) и кеширања. Међутим и ове технике имају своје лимите. Паралелизам веће грануларности омогућава да сам програмер, приликом писања програма, имплементацијом одговарајућих паралелних алгоритама омогући веће искоришћење ресурса паралелног система, а тиме и брже извршавање програма. С обзиром да анализа паралелизма који постоји код различитих имплементација конкурентних програма није тема ове књиге, напомене у овом пасусу намењене су пре свега повезивању са одговарајућим областима које се баве постизањем што вишег нивоа паралелизма.

## Класификација паралелних и дистрибуираних рачунарских система

За област конкурентног програмирања од значаја су највише системи са *MIMD* архитектуром који се деле на: системе са дељеном меморијом (*shared memory systems*), мултикомпјутере са дистрибуираном меморијом (*distributed memory multicomputers*) и мреже рачунара (*network of workstations, cluster of workstations*).

*Системи са дељеном меморијом* имају велики број поткласа у зависности од начина међусобног повезивања процесора и меморије. Наведимо само главне:

- системи са униформним приступом меморији (*Uniform Memory Access - UMA*), познати и као системи са симетричним приступом меморији (*Symmetric MultiProcessor systems - SMP*), имају физички јединствену меморију којој сви процесори приступају на исти начин преко интерконекционе мреже, обично једноставније топологије типа магистрале (*bus*) или мреже (*grid*), и то тако да је време приступа свих процесора меморији исто (униформно). Процесори притом могу имати своје приватне кеш меморије.
- системи за неуниформним приступом меморији (*Non-Uniform Memory Access - NUMA*) имају физички јединствену али хијерархијски организовану меморију. Процесори су повезани са меморијом преко интерконекционе мреже која обично има нешто сложенију топологију. Време приступа сваког процесора меморији није униформно и

зависи од локације податка; важи принцип да што се податак налази дубље у меморијској хијерархији, више времена је потребно за приступ до њега.

- системи са дистрибуираном заједничком меморијом (*Distributed Shared Memory systems - DSM*) имају логички јединствену, али физички дистрибуирану меморију. Сваки процесор има своју приватну меморију која је истовремено део заједничког адресног простора са другим приватним меморијама у систему. Процесорски чворови су међусобно повезани преко интерконекционе мреже.

*Мулткомпјутер са дистрибуираном меморијом* је рачунарски систем који се састоји од процесора који су међусобно повезани преко интерконекционе мреже, а сваки од њих има своју приватну меморију. За разлику од *DSM*, приватна меморија сваког процесора није део глобалног адресног простора. Комуникација између процеса се врши прослеђивањем порука кроз мрежу. Постоје различите топологије интерконекционих мрежа (*switch, grid, mesh, hypercube*, итд.) и различити протоколи за слање порука кроз њих.

Мрежа рачунара се, као што име каже, састоји од више независних рачунара који су физички повезани преко рачунарске мреже (нпр. *Ethernet*). Рачунари комуницирају прослеђивањем порука користећи за то предвиђене протоколе.

## Класификација парадигми конкурентног програмирања

Различити програмски модели (парадигме) за конкурентно програмирање су формирани у складу са различитим рачунарским архитектурама и њиховим типичним параметрима (потпут просечног времена кашњења приликом приступа меморији, типу архитектуре меморијског подсистема, броја процесора у систему и сл.), а на вишем нивоу и према домену примене (нпр. научно-истраживачки прорачуни, вештачка интелигенција и сл.). Њихов је задатак да програмеру омогуће да на ефикасан и разумљив начин може да формулише програме који ће постићи жељени циљ.

Постоји више различитих парадигми конкурентног програмирања, а према типу интеракције између процеса могу се поделити на две велике групе:

Код **шаред варивабилнија програмирања** (*shared variable programming*) програм садржи активне процесе и пасивне дельјене променљиве (објекте). Процеси интерагују тако што читају и уписују вредности у дельјене променљиве.

Код **дистрибуираног програмирања** (*distributed programming*) у програму нема дельјених објекта. Интеракција између процеса је заснована на:

- 1) експлицитном слању и примању порука (порука је неки објекат који се прослеђује другом процесу) помоћу комуникационих примитива (нпр. *send* и *receive*).
- 2) удаљеним позивима процедуре (*Remote Procedure Call - RPC*, или *Remote Method Invocation - RMI*), где позивајућа и позвана процедура припадају различитим процесима. Приликом сваког позива удаљене процедуре, позваном процесу се шаљу одговарајући параметри, након чега он извршава позвану процедуру и шаље резултат назад позивајућем процесу (позивајући процес је до тог тренутка блокиран). Прихвататање позива од стране позваног процеса је имплицитно, тј. унутар позваног процеса нема експлицитних наредби за прихвататељ позива, већ се они прихвататују чим стигну. По прихвататељу позива се најчешће креира нова програмска нит или процес у оквиру кога се извршава позвана процедура. Ређе се примењује решење код кога постоји само једна нит или процес за извршавање позване процедуре за све позивајуће процесе. Тада се позивајући процеси сачекују блокирани у реду за позивање удаљене процедуре. Интеракција помоћу удаљених позива процедура је увек потпуно синхронна.

## Увод

3) механизму рандевуа (*rendezvous*), где се приликом позива сервисне процедуре два процеса сачекују пре него што наставе са извршавањем – позивајући процес чека да позвани процес дође до тачке прихватања рандевуа, а позвани процес чека да позивајући процес стигне до тачке позива (зависно од тога који процес стигне први до тачке позива/прихватања, десиће се један од ова два сценарија). Разлика између рандевуа и удаљених позива процедура је што код рандевуа код позваног процеса постоји експлицитно дефинисана тачка прихватања позива (*accept*) на којој се он блокира док не стигне позив. Код удаљених позива процедура нема блокирања позваног процеса, већ се по добијању позива типично одмах креира нова нит или процес од стране позваног процеса.

Поред горе наведених постоје и други параметри према којима се може вршити класификација парадигми конкурентног програмирања. Једна од најопштијих и најчешће коришћених класификација је према моделу координације (*coordination model*).

Код овог типа класификације полази се од чињенице да се све активности програма могу поделити на рачунање (*computation*) и координацију (*coordination*). Ова два аспекта су међусобно ортогонална и могу се посматрати независно. Појам координације поред начина на који међусобно интерагују активни делови програма<sup>3</sup> обухвата и начин на који су они организовани у програмску целину (креирање и уклањање активних делова програма, њихову просторну дистрибуцију и дистрибуцију и синхронизацију у времену).

Сваки координациони модел има три елемента:

- Ентитете чији се рад координира – активни делови програма који чине основне градивне компоненте система; нпр. процеси, нити, агенти, торке, итд.
- Координациони медијум – медијум који служи за координацију програмских ентитета; нпр. канали (*channels*), дељене променљиве (*shared variables*), простори података (*data spaces*), итд.
- Правила координације – Ова правила одређују како ентитети координирају рад коришћењем координационог медијума; нпр. скуп примитива за упис и читање у простор података (*tuple space*), итд.

У пракси се најчешће срећу следећа два координациона модела:

**Модел виртуелних простора** представља врсту проширења модела програмирања помоћу дељене меморије. Виртуелни простор је координациони медијум који представља посебан део програмског простора коме може да приступа више независних активних делова програма. Уколико је тај простор резервисан само за податке, онда се за њега користи назив виртуелна дељена меморија (*virtual shared memory*). У општем случају виртуелни простор осим пасивних објеката може садржати и активне делове програма.

Код **програмирања помоћу програмских нити**, координациони модел се заснива на независним програмским токовима - нитима (*threads*), који међусобно могу да комуницирају применом неке од комуникационих парадигми. Овакав начин програмирања се назива *multithreading*.

Свака од ове четири класе модела конкурентног програмирања је посебно обрађена у оквиру књиге, јер се принципи програмирања у оквиру њих битно разликују.

<sup>3</sup> У овом контексту се користи и назив актери (*actors*).

## Програмирање помоћу дељених променљивих

Програмирање помоћу дељених променљивих најчешће се користи за програмирање на рачунарским системима са дељеном меморијом. Овај начин програмирања подразумева да се за комуникацију између процеса у конкурентном програму користе глобалне променљиве којима сви процеси могу да приступају без ограничења – тзв. дељене променљиве (насупрот њих стоје приватне променљиве којима могу да приступају само процеси којима оне припадају). Постоји више различитих програмских парадигми које се користе за писање програма у којима се комуникација и синхронизација између процеса врши помоћу дељених променљивих. Међу њима су најпознатији семафори, условни критични региони и монитори.

Концепт семафора, као и само име, потиче од механизма који се користи у железничком саобраћају за спречавање судара возова. У програмском смислу „возови“ су процеси, „железничке шине“ су критичне секције, а семафори имају улогу да „скрећу и заустављају“ процесе како се не би десило да „дође до судара“, тј. да више возова (процеса) истовремено уђе у критичну секцију. Семафори, иначе, представљају један од основних механизама за синхронизацију процеса. Помоћу њих се једноставно реализацију међусобно искључивање процеса код приступа критичној секцији и условна синхронизација процеса. Из тог разлога је природна и њихова употреба за реализацију конкурентних програма у виду посебне програмске парадигме. Због једноставности коришћења, семафори су укључени у готово све постојеће библиотеке за конкурентно програмирање. Имплементација семафора може бити на различитим нивоима, а најчешће се реализују на нивоу оперативног система или на нивоу програмских библиотека.

Код програмске парадигме критичних региона, главни акценат је стављен на критичну секцију као део програма у коме се приступа дељеним променљивама. С обзиром да се у конкурентним програмима веома често јавља потреба за приступ критичној секцији, у оквиру ове парадигме је реализована апстракција приступа критичној секцији увођењем посебно означених синтаксних целина – критичних региона. Механизам међусобног искључивања процеса код приступа критичној секцији реализован је транспарентно за програмера, чиме је његов посао значајно олакшан, а читљивост и разумљивост програма повећана.

Код програмске парадигме условних критичних региона је извршено једно додатно синтакско проширивање у односу на обичне критичне регионе, како би се омогућила условна синхронизација процеса. Уведена је наредба *await* која дозвољава да се синхронизациони услов постави унутар критичног региона.

Монитори представљају још виши ниво апстракције синхронизационог механизма од условних критичних региона. Монитор енкапсулира скуп дељених променљивих у објекат над којим су дефинисане могуће операције у виду скупа процедуре. Међусобно искључивање процеса приликом приступа монитору је обезбеђено за корисника транспарентним механизmom забране конкурентног извршавања процедура монитора. Условна синхронизација се обавља експлицитно преко условних променљивих. Оне су донекле сличне семафорима, при чему је ред за чекање на условној променљивој по дефиницији типа *FIFO*<sup>4</sup> (код семафора није било специфицирано ког типа мора бити ред за чекање).

<sup>4</sup> Редови чекања на условним променљивама могу и другачије бити реализовани, као што је случај у програмском језику Java.

## Семафори

Семафор је ненегативна целобројна променљива на којој су дефинисане две операције:

**wait(s)** декрементира вредност семафора ( $s=s-1$ ), под условом да вредност семафора  $s$  задовољава услов  $s>0$ ; позивајући процес потом наставља са радом. Ако услов  $s>0$  није задовољен, позивајући процес се блокира док се тај услов не испуни. Провера задовољености услова  $s>0$  (које обухвата две активности: читање вредности семафора и поређење са нулом) и декрементирање вредности семафора (које обухвата још и одузимање јединице и упис нове вредности семафора) се обављају атомски (недељиво).

**signal(s)** инкрементира вредност семафора ( $s=s+1$ ), под условом да нема блокираних процеса на том семафору; ако има блокираних процеса, онда се један од њих деблокира (семафор притом не мења вредност). Инкрементирање вредности семафора (које обухвата читање вредности семафора, додавање јединице и упис нове вредности семафора) се обавља атомски.

Ако је више процеса истовремено блокирано на семафору, покушава да изврши операцију *wait(s)*, онда ће само један од њих успети да се деблокира након операције *signal(s)*. Који од чекајућих процеса успева зависи од тога како је имплементирано чекање на семафору. Операције *wait* и *signal* се најчешће реализују на нивоу оперативног система и то тако што сваки семафор има ред у који процеси стају ако услов  $s>0$  није задовољен. Тиме се избегава запослено чекање на семафору, а постиже се равноправност код приступа. Код такве реализације би алгоритми операција над семафорима изгледали овако:

```
wait(s)
if s>0
then s=s-1
else begin
    Заустави процес и стави га у стање чекања код ОС
    Стави процес у ред чекања на семафору s
    Ослободи процесор
end

signal(s)
if ред чекања на семафору s празан
then s=s+1
else begin
    Уклони први процес из реда чекања на семафору s
    Активирај тај процес код ОС
end
```

Треба водити рачуна да ниједан паралелни програм (који претендује да буде преносив) не сме да се ослања на неки одређени алгоритам замене, када је у питању ред чекања на семафору, јер се он може разликовати од једног до другог система.

У пракси се рад са семафорима најчешће реализује помоћу програмских библиотека<sup>5</sup>. Други начин да се реализује рад са семафорима јесте проширење неког постојећег програмског језика одговарајућим синтаксним конструкцијама. **Проширени Pascal** је један такав језик који представља стандардни *Pascal* проширен примитивама за рад са семафорима и елементарним конструкцијама за конкурентно извршавање процеса. Он ће овде бити коришћен због своје једноставне синтаксе. Синтаксна проширења у односу на стандардни *Pascal* јесу следећа:

- Декларација семафора *s*:

```
var s : semaphore
```

- Иницијализација семафора *s* на (ненегативну) вредност *n*:

```
init(s, n)
```

- Декларација дељене променљиве *x* типа *type*:

```
var x : shared type
```

- Операције *wait* и *signal* на семафору *s*:

```
wait(s)
```

```
signal(s)
```

- Синтакса којом се обележава део програма у коме се паралелно извршавају процеси *proc1*, ..., *procN*:

```
cobegin proc1; ... procN coend
```

Процес у овом случају може бити једна наредба, низ наредби унутар *begin-end* блока или процедуре.

<sup>5</sup> Један такав пример је *Pthreads* библиотека за језик C, која имплементира POSIX стандард за писање конкурентних програма са више нити (*multithreaded programs*) и рад са семафорима.

## 1 Проблем критичне секције

Дат је упоредни програм на језику проширенi Pascal:

```
program TimeDependent;
var x : shared integer;
begin
  x := 2;
  cobegin
    x := x + 1;
    x := x - 1;
  coend;
  writeln('x = ', x)
end.
```

- Одредити све могуће излазне резултате програма.
- Отклонити временску зависност у датом програму употребом семафора.

Претпоставити да процесор нема машинске инструкције за (атомско) инкрементирање и декрементирање.

## Решење

a) Наредба  $x := x+1$  се састоји из следећих атомских операција:

- читање операнда из меморије – *read* (означићемо ову операцију са  $R1$ ),
- инкрементирање унутар процесорске *ALU*
- упис резултата у меморију – *write* (означићемо ову операцију са  $W1$ )

Наредба  $x := x-1$  се састоји из следећих атомских операција:

- читање операнда из меморије – *read* (означићемо ову операцију са  $R2$ ),
- декрементирање унутар процесорске *ALU*
- упис резултата у меморију – *write* (означићемо ову операцију са  $W2$ )

За обе наредбе важи да се одговарајуће атомске операције увек извршавају истим редоследом (најпре читање, затим инкрементирање/декрементирање, и на крају упис).

		редослед операција →				Излаз
	Операција	R1	W1	R2	W2	
1.	Вредност x након операције	2	3	3	2	2
2.	Операција	R1	R2	W1	W2	
	Вредност x након операције	2	2	3	1	1
3.	Операција	R1	R2	W2	W1	
	Вредност x након операције	2	2	1	3	3
4.	Операција	R2	R1	W1	W2	
	Вредност x након операције	2	2	3	1	1
5.	Операција	R2	R1	W2	W1	
	Вредност x након операције	2	2	1	3	3
6.	Операција	R2	W2	R1	W1	
	Вредност x након операције	2	1	1	2	2

Табела 1. Сви могући сценарији током извршавања програма (редослед извршавања атомских операција читања и уписа у меморију је са лева у десно). За сваки од сценарија дате су међувредности променљиве x и наглашене су вредност које ће се исписати. Могуће излазне вредности су: 1, 2 и 3.

Резултат рада програма је исписивање вредности дељене променљиве  $x$ . С обзиром да се наредбе  $x:=x+1$  и  $x:=x-1$  извршавају конкурентно, независно једна од друге, могућ је различит редослед којим наредбе врше читање и писање у меморију, што утиче на коначну вредност променљиве  $x$  која ће бити исписана. Редослед извршавања операција инкрементирања и декрементирања унутар  $ALU$  не утиче на коначну вредност променљиве  $x$ . Сви могући сценарији приступа меморији и одговарајуће коначне вредности променљиве  $x$  су дате у Табели 1. Треба приметити да резултат извршавања може бити и другачији уколико дате операције нису атомске.

б) Проблем који је овде илустрован се назива утркивање (*race condition*) и последица је истовременог приступа дељеним ресурсима од стране више процеса. Да би се овај проблем решио (елиминисала временска зависност у програму) потребно је да се обе конкурентне операције изврше атомски (недељиво, без прекидања). Низ операција над скупом дељених објеката који треба да се изврши атомски назива се *критична секција*. Проблем избегавања истовременог приступа критичној секцији од стране више процеса (што се још зове и *проблем међусобној искључивања процеса*), се помоћу семафора решава тако што се испред критичне секције ставља *wait(s)*, а иза критичне секције *signal(s)*. За приступ критичним секцијама које оперишу над скупом истих дељених променљивих сви процеси морају да користе исти семафор  $s$ . Наравно, пошто постоје критичне секције које оперишу над потпуно дисјунктним скуповима дељених променљивих, у једном програму може постојати више различитих семафора. Семафори који се користе само за међусобно искључивање процеса могу имати само две вредности (0 и 1), па се зову и бинарни семафори (*binary semaphores*). Ови семафори се морају иницијализовати тако да не дође до блокирања на почетку и имају иницијалну вредност 1. Ако би у свим процесима свакој операцији *signal(s)* претходио *wait(s)*, а бинарни семафор  $s$  за приступ критичној секцији се погрешно иницијализовао на 0, онемогућило би се било ком процесу да уђе у критичну секцију.

Пошто у програму *TimeDependent* постоји само једна дељена променљива ( $x$ ), увођењем бинарног семафора ( $s$ ) и његовом иницијализацијом на вредност 1 можемо обезбедити недељиво извршавање одговарајућих критичних секција (тј. наредби  $x:=x+1$  и  $x:=x-1$ ). На тај начин спречавамо преплитање фаза читања из меморије и уписа у меморију током њиховог извршавања. Тада су могућа само два сценарија: да се прво у целини изврши наредба  $x:=x+1$  па потом  $x:=x-1$ , и обратно (ово одговара варијантама 1 и 6 из Табеле 1). На излазу се у оба случаја добија очекивани резултат 2:

```
program TimeIndependent;
var x : shared integer;
    s : semaphore;
begin
    x := 2; init(s,1);
    cobegin
        begin
            wait(s);
            x := x + 1;
            signal(s)
        end;
        begin
            wait(s);
            x := x - 1;
            signal(s)
        end;
    coend;
    writeln('x = ', x)
end.
```

## 2

## Произвођач и потрошач: условна синхронизација процеса

Дат је упоредни програм на језику проширенi Pascal:

```
program Graph;
const n = ...;
var x, y : shared integer;

procedure makepoints;
var i : integer;
begin
    for i := 1 to n do begin
        x := i; y := i * i;
    end
end;

procedure printpoints;
var i : integer;
begin
    for i := 0 to n do write('(', x, ',', y, ')');
end;

begin
    x := 0; y := 0;
    cobegin
        makepoints;
        printpoints;
    coend
end.
```

Жељени излаз програма је низ парова облика: (0,0) (1,1) (2,4) ... ( $\pi, \pi^2$ )

- Одредити све могуће излазе програма за случај  $n=1$ .
- Отклонити временску зависност у датом програму употребом семафора.

## Решење

- За  $n=1$  петља у процедуре *makepoints* има једну итерацију. У итерацији се најпре врши додељивање вредности променљивој  $x$ , а затим променљивој  $y$ , тим редоследом. Означимо упис вредности у променљиву  $x$  са  $x_m$ , а упис вредности у променљиву  $y$  са  $y_m$ .

Петља у процедуре *printpoints* има две итерације. У свакој итерацији се најпре читаје вредности променљиве  $x$ , а затим променљиве  $y$ , тим редоследом. Означимо читање вредности променљиве  $x$  у првој итерацији ( $i=0$ ) са  $x_{p0}$ , читање вредности променљиве  $x$  у другој итерацији ( $i=1$ ) са  $x_{p1}$ , читање вредности променљиве  $y$  у првој итерацији ( $i=0$ ) са  $y_{p0}$ , и читање вредности променљиве  $y$  у другој итерацији ( $i=1$ ) са  $y_{p1}$ .

Приступ локалним променљивама у обе процедуре не утиче на резултат.

Процедуре *makepoints* и *printpoints* се извршавају конкурентно, па је могуће произвољно учешљавање (*interleaving*) њихових операција. Међутим, с обзиром да се унутар процедуре *makepoints* најпре врши додељивање вредности променљивој  $x$ , а затим променљивој  $y$  (тим редоследом),  $x_m$  мора да се нађе пре  $y_m$ . Такође, у

	Операција	редослед операција →						Излаз
		$x_m$	$y_m$	$x_{p0}$	$y_{p0}$	$x_{p1}$	$y_{p1}$	
1.	Вредности $x/y$ након операције	1/0	1/1	1/1	1/1	1/1	1/1	(1,1) (1,1)
2.	Операција	$x_m$	$x_{p0}$	$y_m$	$y_{p0}$	$x_{p1}$	$y_{p1}$	(1,1) (1,1)
3.	Вредности $x/y$ након операције	1/0	1/0	1/1	1/1	1/1	1/1	(1,1) (1,1)
4.	Операција	$x_m$	$x_{p0}$	$y_{p0}$	$y_m$	$x_{p1}$	$y_{p1}$	(1,0) (1,1)
5.	Вредности $x/y$ након операције	1/0	1/0	1/0	1/0	1/0	1/0	(1,0) (1,0)
6.	Операција	$x_{p0}$	$x_m$	$y_m$	$y_{p0}$	$x_{p1}$	$y_{p1}$	(0,1) (1,1)
7.	Вредности $x/y$ након операције	0/0	1/0	1/1	1/1	1/1	1/1	(0,0) (1,1)
8.	Операција	$x_{p0}$	$x_m$	$y_{p0}$	$x_{p1}$	$y_m$	$y_{p1}$	(0,0) (1,1)
9.	Вредности $x/y$ након операције	0/0	1/0	1/0	1/0	1/0	1/1	(0,0) (1,0)
10.	Операција	$x_{p0}$	$y_{p0}$	$x_m$	$y_m$	$x_{p1}$	$y_{p1}$	(0,0) (1,1)
11.	Вредности $x/y$ након операције	0/0	0/0	1/0	1/1	1/1	1/1	(0,0) (1,1)
12.	Операција	$x_{p0}$	$y_{p0}$	$x_m$	$x_{p1}$	$y_{p1}$	$y_m$	(0,0) (1,0)
13.	Вредности $x/y$ након операције	0/0	0/0	0/0	1/0	1/1	1/1	(0,0) (0,1)
14.	Операција	$x_{p0}$	$y_{p0}$	$x_{p1}$	$x_m$	$y_{p1}$	$y_m$	(0,0) (0,0)
15.	Вредности $x/y$ након операције	0/0	0/0	0/0	1/0	1/0	1/1	(0,0) (0,0)

**Табела 2.** Сви могући сценарији током извршавања програма (редослед извршавања атомских операција читања и уписа у меморију је са лева у десно). За сваки од сценарија дате су међувредности променљивих  $x$  и  $y$  наглашене су вредности које ће се исписати. Могући излази су: (1,1)(1,1), (1,0)(1,1), (1,0)(1,0), (0,1)(1,1), (0,0)(1,1), (0,0)(1,0), (0,0)(0,1) и (0,0)(0,0).

процедури *printpoints* се најпре извршава итерација за  $i=0$ , и то  $x_{p0}$  пре  $y_{p0}$ , па потом итерација за  $i=1$ , и то  $x_{p1}$  пре  $y_{p1}$ . Сви могући сценарији су дати у табели 2.

б) Процедуре *makepoints* и *printpoints* представљају пример производњачког и потрошачког процеса (*producer/consumer processes*). Обе ове процедуре треба да се извршавају недељиво. Временска зависност програма *Graph* потиче управо од тога што није обезбеђено међусобно искључивање ових процедуре приликом приступа дељеним променљивама  $x$  и  $y$ .

Само међусобно искључивање, међутим, није доволично, већ је потребно вршити и условну синхронизацију ова два процеса. Наиме, испис координата треба да се изврши тек када се изгенирише нова тачка. Истовремено важи да нова тачка треба да се генирише тек након што се испише претходна, како се не би десило да *printpoints* више пута прочита координате исте тачке или да *makepoints* више пута препише старе координате пре него што *printpoints* стигне да их прочита.

Условна синхронизација подразумева да се извршавање процеса привремено зауставља, све док се не задовољи неки постављени услов. То се помоћу семафора постиже тако што се сваком услову придржи један семафор *s*, а операција *wait(s)* се ставља на сваком месту где је потребно сачекати испуњење датог услова пре наставка извршавања процеса. Са друге стране, на сваком месту где је потребно сигнализирати да је дошло до испуњења услова синхронизације коме је придружен семафор *s*, ставља се операција *signal(s)*, како би био деблокиран неки од процеса који евентуално чекају на испуњење услова. За условну синхронизацију се такође користе бинарни семафори.

Истовремено међусобно искључивање и условна синхронизација се у случају ове *producer/consumer* интеракције постиже помоћу два семафора: *full* – сигнализира да је производач генерисао нови резултат (генерисана нова тачка), и *empty* – сигнализира да је потрошач прочитао последњи резултат (исписао последњу тачку). Овакав пар семафора се назива и расподељени бинарни семафор (*split binary semaphore*), јер се понаша као један семафор који је подељен на два дела, од којих само један у неком тренутку може имати вредност 1.

Елиминисањем временске зависности помоћу семафора *full* (иницијално је постављен на 1 како би се одмах исписала тачка (0,0)) и *empty* (иницијално је 0), добија се следећи програм:

```
program Graph;
const n = ...;
var x, y : shared integer;
    full, empty : semaphore;

procedure makepoints;
var i : integer;
begin
    for i := 1 to n do begin
        wait(empty);
        x := i;
        y := i * i;
        signal(full)
    end
end;

procedure printpoints;
var i : integer;
begin
    for i := 0 to n do begin
        wait(full);
        write('(' , x, ', ', y, ')');
        signal(empty)
    end
end;

begin
    x := 0; y := 0;
    init(full, 1); init(empty, 0);
    cobegin
        makepoints;
        printpoints;
    coend
end.
```

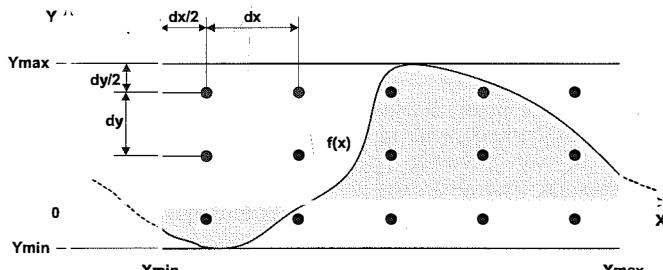
### 3 Произвођачи и потрошачи: комуникација помоћу кружног бафера

Процес *integral* рачуна вредност интеграла функције  $f(x)$  на интервалу  $[Xmin, Xmax]$  на следећи начин: Процес *pointgen* сукцесивно генерише и шаље процесу *integral* парове реалних вредности  $(x, y)$  које представљају тачке у  $x$ - $y$  равни, униформно дистрибуиране по области  $Xmin \leq x < Xmax$ ,  $Ymin \leq y < Ymax$  ( $Xmin$  и  $Xmax$  су границе интервала интеграције, а  $Ymin$  и  $Ymax$  су минимална и максимална вредност функције  $f(x)$  на интервалу  $[Xmin, Xmax]$ , респективно). За сваку од генерисаних тачака процес *integral* утврђује да ли она лежи између криве  $f(x)$  и  $x$ -осе и, ако лежи, да ли је у позитивном или негативном интервалу дате функције; у првом случају инкрементира променљиву  $F$ , а у другом случају је декрементира (иницијално је  $F=0$ ). Након што се генерише свих  $N$  тачака у области, рачуна се приближна вредност интеграла  $I$  функције на датом интервалу из формуле  $I/N = F/[N*(Xmax-Xmin)*(Ymax-Ymin)]$ . Вредности  $Xmin$ ,  $Xmax$  и  $N$  се учитавају са улаза, а вредности  $Ymin$  и  $Ymax$  се рачунају позивањем одговарајућих функција  $fmin(Xmin, Xmax)$ , односно  $fmax(Xmin, Xmax)$ .

- а) Коришћењем семафора написати програм на језику проширенi Pascal који реализацију процесе *pointgen* и *integral*.
- б) Због повећања конкурентности увести кружни бафер (*buffer* (енг.) - прихватник) *buf* тако да истовремено процес *pointgen* може да ставља новогенерисане тачке на крај реда, а процес *integral* да их узима са почетка реда.
- в) Даље повећати конкурентност претпостављајући да може постојати више процеса који генеришу тачке и више процеса који рачунају вредност датог интеграла. Претпоставити и даље да се стављање вредности на крај реда и узимање вредности са почетка реда могу радити истовремено (као у тачки б).
- г) Максимизовати конкурентност тако што ће се дозволити истовремени упис у бафер од стране више процеса који генеришу тачке (на различите, узастопне локације у баферу), као и истовремено читање из бафера од стране више процеса који рачунају вредност датог интеграла (са различитих, узастопних локација).

### Решење

- а) Процес *pointgen* је произвођач, а процес *integral* је потрошач. Треба само применити расподељење бинарне семафоре (*split binary semaphore*) на ова два процеса. У овом примеру процес *pointgen* генерише тачке као на слици 1.



**Слика 1.** Распоред тачака које генерише процес *pointgen*. У овом примеру је укупан број тачака  $N=15$ , број тачака по  $x$ -оси  $N_x=5$ , број тачака по  $y$ -оси  $N_y=3$ ,  $F=5-1=4$ , а приближна вредност интеграла функције  $f(x)$  је  $F/N*(Xmax-Xmin)*(Ymax-Ymin)$ .

```
program Integration;
type point = record x, y : real end;
var Xmin, Xmax, Ymin, Ymax, dx, dy : real;
    N, Nx, Ny, F : integer;
    p : shared point;
    full, empty : semaphore;

function f(x : real) : real;
begin
    f := x * x
end;

function fmin(Xmin, Xmax, dx : real) : real;
var min, x, y : real;
begin
    x := Xmin + 0.5 * dx; min := f(x);
    while x <= Xmax do begin
        y := f(x);
        if y < min then min := y;
        x := x + dx
    end;
    fmin := min
end;

function fmax(Xmin, Xmax, dx : real) : real;
var max, x, y : real;
begin
    x := Xmin + 0.5 * dx; max := f(x);
    while x <= Xmax do begin
        y := f(x);
        if y > max then max := y;
        x := x + dx
    end;
    fmax := max
end;

procedure pointgen;
var i : integer;
begin
    for i := 0 to N-1 do begin
        wait(empty);
        p.x := Xmin + (i mod Nx + 0.5) * dx;
        p.y := Ymin + (i div Nx + 0.5) * dy;
        signal(full)
    end
end;

procedure integral;
var i : integer;
begin
    for i := 0 to N-1 do begin
        wait(full);
        if ((p.y <= f(p.x)) and (p.y >= 0)) then F := F + 1;
        if ((p.y >= f(p.x)) and (p.y <= 0)) then F := F - 1;
        signal(empty)
    end
end;
```

```

end
end;

begin
  write('N = '); readln(N);
  write('Xmin = '); readln(Xmin);
  write('Xmax = '); readln(Xmax);
  dx := (Xmax - Xmin) / N; {Using more dense raster since we
                           don't know dimensions of the integration area}
  Ymin := fmin(Xmin, Xmax, dx);
  Ymax := fmax(Xmin, Xmax, dx);
  Nx := trunc(sqrt(N * (Xmax - Xmin) / (Ymax - Ymin)));
  Ny := trunc(sqrt(N * (Ymax - Ymin) / (Xmax - Xmin)));
  if Nx * Ny < N then begin Ny := Ny + 1; Ny := Ny + 1 end;
  dx := (Xmax - Xmin) / Nx;
  dy := (Ymax - Ymin) / Ny;
  F := 0;
  init(full, 0); init(empty, 1);
  cobegin
    pointgen;
    integral;
  coend;
  writeln('integral = ', F / N * (Xmax - Xmin) * (Ymax - Ymin))
end.

```

б) Увођењем бафера се постиже повећање конкурентности јер процес произвођач и процес потрошач не морају да се сачекују за сваки новогенерисани резултат, већ се њихова тренутна разлика у брзини рада амортизује прихватањем резултата у бафер. Треба приметити, међутим, да бафер помаже само ако нема дуготрајније разлике у брзини генерирања резултата од стране процеса производиоџача и брзине преузимања резултата од стране процеса потрошача. У супротном ће бафер бити или стално пун (ако производиоџач у дужем периоду времена ради брже од потрошача) или стално празан (ако производиоџач у дужем периоду времена ради спорије од потрошача), што поништава амортизујуће дејство бафера и чак донекле погоршава перформансе јер је потребно додатно време за упис и читање из бафера. Величина бафера одређује већи или мањи ниво амортизације и бира се тако да одговара типу процеса производиоџача и потрошача. Овде је произвољно узето да бафер има 10 места.

Ако дефинишемо појам *ресурс* као скуп дељених променљивих истог типа (које можемо назвати јединице ресурса), онда ће бафер *buf* који се уводи представљати управо један ресурс. *Алокација ресурса* представља појам који означава давање дозволе за коришћење јединице ресурса неком процесу (услов је да има слободних јединица ресурса).

Алокација ресурса се реализује тако што се сваком ресурсу пријдружи један *бројачки семафор* (зове се још и *генерални семафор* - *general semaphore*). Бројачки семафори могу имати било коју позитивну целобројну вредност (за разлику од бинарних семафора који могу имати само вредности 1 и 0). Они се у почетку иницијализују на вредност која представља укупан број јединица датог ресурса (овде је то број места у бафери). Испред сваког места коришћења ресурса треба да стоји *wait(s)*. Тиме се омогућава слободан приступ ресурсу док год има слободних јединица; блокирање на семафору настаје тек кад су све јединице ресурса заузете. Након што заврши коришћење ресурса, процес треба да изврши операцију *signal(s)*, чиме се сигнализира да је приступ једној јединици ресурса ослобођен (постао слободан). Обавеза је процеса који приступају ресурсу да том приликом воде рачуна о међусобном

искључивању (овде то значи да процеси *pointgen* и *integral* не смеју да приступају истовремено истој локацији у баферау) и условној синхронизацији (овде то значи да се не сме генерисати нова тачка ако је бафер пун, нити да се може узимати нова тачка ако је бафер празан).

Пандан бинарном семафору *empty* из тачке а) је бројачки семафор *not\_full*, а семафору *full* семафор *not\_empty*.

```
program Integration;
const size = 10;
type point = record x, y : real end;
var Xmin, Xmax, Ymin, Ymax, dx, dy : real;
    N, Nx, Ny, F, front, rear : integer;
    buf : shared array[0..size-1] of point;
    not_empty, not_full : semaphore;

function f(x : real) : real; begin ... end;
function fmin(Xmin, Xmax, dx : real) : real; begin ... end;
function fmax(Xmin, Xmax, dx : real) : real; begin ... end;

procedure pointgen;
var i : integer;
    p : point;
begin
    for i := 0 to N-1 do begin
        p.x := Xmin + (i mod Nx + 0.5) * dx;
        p.y := Ymin + (i div Nx + 0.5) * dy;
        wait(not_full);
        buf[rear] := p;
        rear := (rear + 1) mod size;
        signal(not_empty)
    end
end;

procedure integral;
var i : integer;
    p : point;
begin
    for i := 0 to N-1 do begin
        wait(not_empty);
        p := buf[front];
        front := (front + 1) mod size;
        signal(not_full);
        if ((p.y <= f(p.x)) and (p.y >= 0)) then F := F + 1;
        if ((p.y >= f(p.x)) and (p.y <= 0)) then F := F - 1;
    end
end;

begin
    write('N = '); readln(N);
    write('Xmin = '); readln(Xmin);
    write('Xmax = '); readln(Xmax);
    dx := (Xmax - Xmin) / N; {Using more dense raster since we
                                don't know dimensions of the integration area}
    Ymin := fmin(Xmin, Xmax, dx);

```

```

Ymax := fmax(Xmin, Xmax, dx);
Nx := trunc(sqrt(N * (Xmax - Xmin) / (Ymax - Ymin)));
Ny := trunc(sqrt(N * (Ymax - Ymin) / (Xmax - Xmin)));
if Nx * Ny < N then begin Ny := Ny + 1; Ny := Ny + 1 end;
dx := (Xmax - Xmin) / Nx;
dy := (Ymax - Ymin) / Ny;
F := 0; front := 0; rear := 0;
init(not_empty, 0); init(not_full, size);
cobegin
    pointgen;
    integral;
coend;
writeln('integral = ', F / N * (Xmax - Xmin) * (Ymax - Ymin))
end.

```

Треба приметити да у овом случају променљиве *rear*, *front* и *F* нису дељени ресурси (дељене променљиве) и да није потребно увођење семафора који би ограничио приступ овим променљивама. Ове променљиве нису дељене јер променљивој *rear* приступа само процес *pointgen*, док променљивама *front* и *F* приступа само један процес, *integral*. У случају да овим променљивама може да приступи већи број процеса било би потребно увести додатне семафоре како би се ограничио приступ и обезбедило међусобно искључивање датих процеса.

Такође треба приметити да дато решење може имати нешто већу конкурентност од решења код кога процес *pointgen* прво чека дозволу за приступ ресурсу а након тога генерише нову вредност.

```

procedure pointgen_less_concurrent;
var i : integer;
p : point;
begin
    for i := 0 to N-1 do begin
        wait(not_full);
        p.x := Xmin + (i mod Nx + 0.5) * dx;
        p.y := Ymin + (i div Nx + 0.5) * dy;
        buf[rear] := p;
        rear := (rear + 1) mod size;
        signal(not_empty)
    end
end;

```

в) Ако има више процеса произвођача потребно је обезбедити њихово међусобно искључивање приликом приступа бафера јер се може десити да више произвођача изврши упис у исту локацију бафера. Разлог за то је што сваки од процеса произвођача може извршити наредбу *buf[rear]:=p* пре него што иједан од њих помери показивач на следећу празну локацију у бафера наредбом *rear:=(rear+1) mod size*. Овај пар наредби треба извршити недељиво (оне чине критичну секцију), што се обезбеђује увођењем семафора *mutexP*. Из сличних разлога је потребно обезбедити међусобно искључивање процеса потрошача приликом приступа бафера. Наиме, више потрошача може очитати вредност из исте локације у баферау, јер сваки од њих може извршити наредбу *p:=buf[front]* пре него што иједан од њих помери показивач на следећу локацију у баферау наредбом *front:=(front+1) mod size*. Међусобно искључивање овде се обезбеђује увођењем семафора *mutexC*.

Пошто сви процеси потрошачи приступају истој (сада дељеној) променљивој *F*, потребно је обезбедити њихово међусобно искључивање код извршавања наредбе *F:=*

F+1. Да се не би уводио нови семафор за то, ова наредба се може убацити у постојећу критичну секцију у којој је већ извршено међусобно искључивање потрошача приликом приступа бафери.

Посебно треба водити рачуна и о следећем проблему. Може се десити да у неком тренутку више произвођачких или потрошачких процеса позива једну исту функцију. Ако је нека функција реализована на такав начин да се може истовремено позвати од стране више процеса, а укупан ефекат је као да је та функција позивана у произвољном секвенцијалном редоследу од стране датих процеса, каже се да је она *reentrant*. Најједноставнији (али не и једини) начин да се постигне да функција буде *reentrant* је да се при сваком позиву креира нова инстанца функције (под тим се подразумева да се при сваком позиву креира нов, независан процес, који ће имати своје локалне променљиве). Нпр. сви потрошачки процеси позивају функцију  $f(x)$ . Да би сви потрошачи могли да раде паралелно, сваки од њих би требало да има своју инстанцу функције  $f(x)$  која ће се извршавати независно од других инстанци. У супротном би приликом позива функције  $f(x)$  дошло до колизије. У нашем примеру ћемо подразумевати да је функција  $f(x)$  *reentrant*.

Увек када постоји више процеса (у овом случају више производића и потрошача) поставља се питање на који начин ће се извршити подела целокупног посла ради обраде. Могу се применити два приступа: *Подела уосла унапред* и *Обрада по принципију „шарба послова“*.

**Подела посла унапред** подразумева да сваки процес унапред добија део посла који треба да одради. Притом се мора урадити подела укупног посла на делове (*partitioning*), а како се то ради у време превођења, то се назива статичка подела посла (*static partitioning*). У овом конкретном случају, сваки производић је унапред добија број тачака које треба да генерише, а сваки потрошач колико тачака треба да обради. Тада приступ је добар, јер не захтева додатну синхронизацију између процеса у току извршавања ради поделе посла или ради детекције када је посао завршен (посао је завршен када сви процеси заврше свој део посла). Међутим, неки процеси могу радити брже, а други спорије, па се може десити да бржи процеси, након што заврше рад, чекају беспослени док спорији процеси не заврше свој део посла. Такође, број процеса и величина целог посла морају бити познати унапред. Грануларност поделе целокупног посла на делове мањег обима (*tasks*) који се додељују процесима на обраду треба да буде прилагођена конкретном проблему. Овде је дато једно такво решење:

```
program Integration;
const size = 10; Npointgen = 10; Nintegral = 10;
type point = record x, y : real end;
var Xmin, Xmax, Ymin, Ymax, dx, dy : real;
    N : integer;
    F, front, rear : shared integer;
    buf : shared array[0..size-1] of point;
    not_empty, not_full, mutexP, mutexC : semaphore;
function f(x : real) : real; begin ... end;
function fmin(Xmin, Xmax, dx : real) : real; begin ... end;
function fmax(Xmin, Xmax, dx : real) : real; begin ... end;

function distribp(i : integer) : integer;
begin
    if (i < N mod Npointgen)
    then distribp := N div Npointgen+1
    else distribp := N div Npointgen
end;
function distribi(i : integer) : integer;
```

```

begin
  if (i < N mod Nintegral)
    then distribi := N div Nintegral+1
  else distribi := N div Nintegral
end;

procedure pointgen(j : 0..Npointgen-1; n : integer);
var i : integer;
    p : point;
begin
  for i := 1 to n do begin
    p.x := Xmin + random(trunc(Xmax - Xmin));
    p.y := Ymin + random(trunc(Ymax - Ymin));
    wait(not_full);
    wait(mutexP);
    buf[rear] := p;
    rear := (rear + 1) mod size;
    signal(mutexP);
    signal(not_empty)
  end
end;

procedure integral(j : 0..Nintegral-1; n : integer);
var i : integer;
    p : point;
begin
  for i := 1 to n do begin
    wait(not_empty);
    wait(mutexC);
    p := buf[front];
    front := (front + 1) mod size;
    if ((p.y <= f(p.x)) and (p.y >= 0)) then F := F + 1;
    if ((p.y >= f(p.x)) and (p.y <= 0)) then F := F - 1;
    signal(mutexC);
    signal(not_full)
  end
end;

begin
  write('N = '); readln(N);
  write('Xmin = '); readln(Xmin);
  write('Xmax = '); readln(Xmax);
  dx := (Xmax - Xmin) / N;
  Ymin := fmin(Xmin, Xmax, dx);
  Ymax := fmax(Xmin, Xmax, dx);
  F := 0; front := 0; rear := 0;
  init(not_empty, 0); init(not_full, size);
  init(mutexP, 1); init(mutexC, 1);
  cobegin
    pointgen(0, distribp(0));
    ...
    pointgen(Npointgen-1, distribp(Npointgen-1));
    integral(0, distribi(0));
    ...
    integral(Nintegral-1, distribi(Nintegral-1))
  end
end;

```

```
coend;
writeln('integral = ', F / N * (Xmax - Xmin) * (Ymax - Ymin))
end.
```

Функције *distribp* и *distribi* врше расподелу посла произвођачима и потрошачима; прва одређује број тачака које сваки произвођач треба да генерише, док друга одређује број тачака које сваки потрошач треба да обради. Генерално, расподела посла унапред може да се врши тако да процеси који раде брже добију више посла, а спорији мање, чиме се може минимизирати укупно време обраде. Међутим, за такву расподелу посла је потребно унапред (у време превођења) знати брзину рада процеса. Ако брзина процеса није позната унапред, онда се посао обично унiformно расподељује на процесе, тј. сваки процес добија приближно једнак део посла, што је и овде примењено. Функције *distribp* и *distribi* морају бити *reentrant*.

Функција *pointgen* мора се променити у односу на решења где само један процес генерише тачке (као под а и б), јер би у супротном (с обзиром да процеси користе исти код) сви процеси генерирали идентичан склоп тачака. Овде је примењено решење да функција *pointgen* генерише низ случајно изабраних тачака из области интеграције, тако да сваки од процеса *pointgen(0)...pointgen(Npointgen-1)* генерише различит склоп. Ради прецизности, треба рећи да склопови овако генерирали тачака од стране процеса *pointgen(i)* нису потпуно дисјунктни, јер се неке тачке могу генерирати два или више пута од стране једног или више процеса. Међутим, битно је да генерирали склоп тачака задовољава основне статистичке критеријуме, као што је унiformност.

**Обрада по принципу „торба послова“ (parallel computing with a bag of tasks)** подразумева да се цео посао издели на велики број делова мањег обима (*tasks*) који ће онда бити на располагању за обраду у виду једне „торбе“ (*bag*) пуне таквих мањих послова. Сваки процес онда може узимати из те „торбе“ послове онолико често колико брзо може да их обради. Подела се врши тако да су послови у „торби“ међусобно независни, а грануларност поделе се подешава према природи проблема и броју процеса. Код овакве поделе посла, бржи процеси ће урадити пропорционално већи део посла него спорији процеси, јер ће чешће узимати послове, а на крају ће сви процеси завршити рад у приближно исто време. Предност овог приступа је и што величина целог посла не мора бити позната унапред. Такође, паралелни програми који врше обраду на овај начин су скалабилни, тј. број процеса који врше обраду може бити произвољан и може се динамички мењати током рада, уз одговарајући пораст перформанси са порастом броја процесора. У општем случају процеси који врше обраду (*worker processes*) се могу скицирати на следећи начин:

```
procedure Worker;
begin
  while (има још послова у торби послова) do begin
    узми посао из торбе послова;
    изврши посао (по потреби генериши и нове послове);
  end;
end;
```

Као што се види, потребно је да сваки процес на неки начин утврди да ли има још послова у „торби послова“, односно да зна када треба да прекине са радом, што уводи потребу за додатном синхронизацијом у односу на случај поделе посла унапред.

Код проблема произвођача и потрошача постоје два посла – производња и потрошња. Сваки од ова два типа посла може бити издаљен на делове који ће бити у „торби послова за произвођаче“, односно „торби послова за потрошаче“. Грануларност послова у ове две „торбе послова“ у општем случају не мора бити иста. Овде ћемо узети да је „јединични посао“ у „торби послова за произвођаче“ - генерирање једне тачке, а у „торби послова за потрошаче“ - обрада једне тачке. „Торба послова за

произвођаче" ће у ствари бити једна дељена променљива  $NP$  у којој се иницијално налази  $N$  послова типа „генериши једну тачку”. Узимање из „торбе послова за произвођаче” се своди на декрементирање променљиве  $NP$ , а торба је празна када је  $NP=0$ . Слично, „торба послова за потрошаче” ће бити једна дељена променљива  $NC$  у којој се иницијално налази  $N$  послова типа „обради једну тачку”, а торба је празна када је  $NC=0$ . Експлузиван приступ овим променљивама се обезбеђује помоћу семафора  $mutexNP$  и  $mutexNC$ .

```

program Integration;
const size = 10; Npointgen = 10; Nintegral = 10;
type point = record x, y : real end;
var Xmin, Xmax, Ymin, Ymax, dx, dy : real;
    N : integer;
    F, front, rear, NP, NC : shared integer;
    buf : shared array[0..size-1] of point;
    not_empty, not_full: semaphore;
    mutexP, mutexC, mutexNP, mutexNC : semaphore;
function f(x : real) : real; begin ... end;
function fmin(Xmin, Xmax, dx : real) : real; begin ... end;
function fmax(Xmin, Xmax, dx : real) : real; begin ... end;
procedure pointgen(j : 0..Npointgen-1);
var p : point;
    more : boolean;
begin
    wait(mutexNP);
    if (NP > 0)
        then begin NP := NP - 1; more := true end
        else more := false;
    signal(mutexNP);
    while more do begin
        p.x := Xmin + random(trunc(Xmax - Xmin));
        p.y := Ymin + random(trunc(Ymax - Ymin));
        wait(not_full);
        wait(mutexP);
        buf[rear] := p;
        rear := (rear + 1) mod size;
        signal(mutexP);
        signal(not_empty);
        wait(mutexNP);
        if (NP > 0)
            then begin NP := NP - 1; more := true end
            else more := false;
        signal(mutexNP)
    end
end;
procedure integral(j : 0..Nintegral-1);
var p : point;
    more : boolean;
begin
    wait(mutexNC);
    if (NC > 0)
        then begin NC := NC - 1; more := true end
        else more := false;
    signal(mutexNC);

```

```

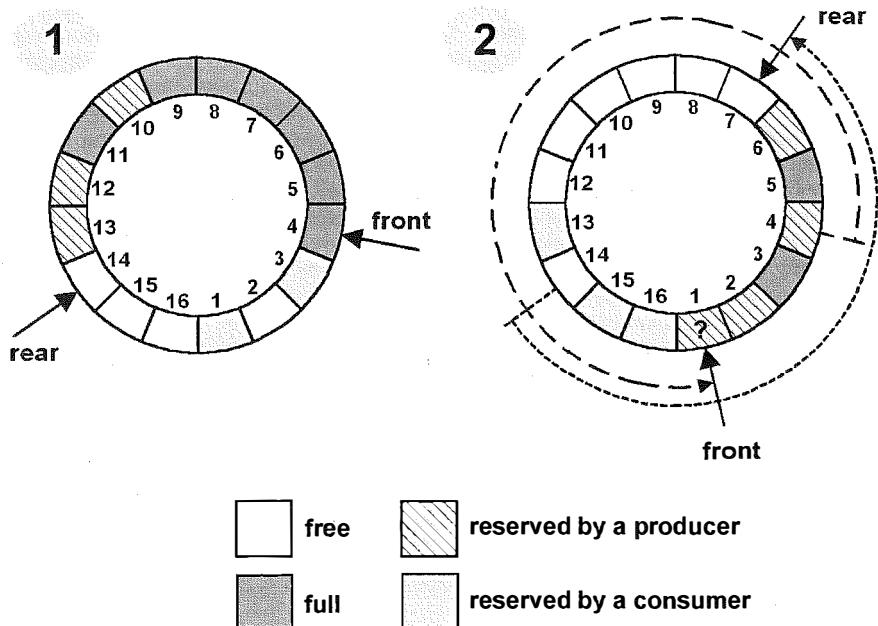
while more do begin
    wait(not_empty);
    wait(mutexC);
    p := buf[front];
    front := (front + 1) mod size;
    if ((p.y <= f(p.x)) and (p.y >= 0)) then F := F + 1;
    if ((p.y >= f(p.x)) and (p.y <= 0)) then F := F - 1;
    signal(mutexC);
    signal(not_full);
    wait(mutexNC);
    if (NC > 0) then begin
        NC := NC - 1; more := true end
    else more := false;
    signal(mutexNC)
  end
end;

begin
  write('N = '); readln(N);
  write('Xmin = '); readln(Xmin);
  write('Xmax = '); readln(Xmax);
  dx := (Xmax - Xmin) / N;
  Ymin := fmin(Xmin, Xmax, dx);
  Ymax := fmax(Xmin, Xmax, dx);
  F := 0; front := 0; rear := 0;
  init(not_empty, 0); init(not_full, size);
  init(mutexP, 1); init(mutexC, 1);
  init(mutexNP, 1); init(mutexNC, 1);
  cobegin
    pointgen(0); ... pointgen(Npointgen-1);
    integral(0); ... integral(Nintegral-1);
  coend;
  writeln('integral = ', F / N * (Xmax - Xmin) * (Ymax - Ymin))
end.

```

г) У претходним решењима смо имали ситуацију да само један процес произвођач може приступати бафери у једном тренутку, чак и када има више слободних локација у њему. Слично, имали смо да само један потрошач може у једном тренутку да узима вредност из бафера, чак и када има више попуњених локација. Максимална конкурентност се постиже када произвођачи могу конкуренћно да уписују у различите слободне локације, односно када потрошачи могу конкуренћно да узимају вредности из различитих локација у бафери.

Да би сви произвођачи и потрошачи могли истовремено да приступају бафери потребно је да сваки од њих приступа *различијој* локацији. То се може обезбедити тако што ће сваки процес приступати бафери у два корака. У првом кораку процес резервише прву локацију која је доступна тако што запамти тренутну вредност показивача *rear* (ако је процес произвођач), односно *front* (ако је процес потрошач), и затим помери показивач за једно место на следећу локацију; на тај начин следећи процес никада не резервише исту локацију као претходни. Приступ променљивама *rear*, односно *front* мора бити ексклузиван (јер више процеса може тражити приступ истовремено), што се постиже уз помоћ семафора *mutexrear*, односно *mutexfront*.



**Слика 2.** Услед великог броја резервација од стране потрошача, може се десити да показивач *front* у неком тренутку дође до локације којој приступ од стране производића још није завршен (могућа је и обрнута ситуација, која овде није илустрована - да показивач *rear* дође до локације којој приступ од стране потрошача још није завршен).

Стање 1: Локације 1-3 су биле пуне, а онда су резервисане од стране потрошача. Потрошач који чита са локације 2 је већ завршио приступ и ослободио је, док су локације 1 и 3 и даље резервисане и приступ њима још траје. Локације 4-9 су попуњене. Локације 10-13 су биле празне, а онда су резервисане од стране производића. Производићај који уписује у локацију 11 је већ завршио приступ и попунио је, док су локације 10, 12 и 13 и даље резервисане и приступ њима још траје. Локације 14-16 су празне.

Стање 2: У наставку је завршен приступ локацијама 1 и 3 од стране потрошача, као и локацијама 10, 12 и 13 од стране производића. Потом су и локације 14-16 попуњене од стране производића. Затим је локацијама 4-12 извршен приступ од стране потрошача и оне су потом и ослобођене. У наставку су локације 1-6 биле резервисане од стране производића, при чему је упис у локације 3 и 5 завршен, а локације 13-16 су резервисане од стране потрошача. Приступ локацији 14 је завршен, чиме је она ослобођена, а приступ локацијама 13, 15 и 16 је још у току. Показивач *front* показује на локацију 1 и следећи потрошач би требало да резервише ту локацију. Међутим, производићај који је претходно резервисао ту локацију још увек је држи јер није завршио упис. Оваква ситуација се може решити увођењем услова да се резервација може извршити једино ако је претходно добијено право ексклузивног приступа локацији.

Треба приметити да ће други процеси тог типа који чекају на приступ баферу бити блокирани док текући процес не заврши резервацију. У другом кораку, након извршене резервације, процес може да приступи датој локацији (уписује у њу или чита из ње) радије паралелно са осталим процесима.

Показивачи *rear* и *front* се померају приликом сваке резервације, не чекајући да се заврши приступ претходно резервисаним локацијама. На тај начин се омогућава паралелан приступ локацијама бафера. Међутим, може се десити да због велике потражње буду резервисане све доступне локације и да почну да се резервишу и локације којима приступ није био завршен (слика 2). Наравно, резервација тих локација не сме бити извршена да се не би десило да више процеса истовремено приступа истим локацијама. Решење овог проблема је у томе да процес мора тражити ексклузиван приступ локацији пре него што изврши резервацију. Зато се за сваку (*i*-ту) локацију у баферу уводи по један семафор (*mutexBuf[i]*, *i*=0..*size*-1). Ако процес добије ексклузиван приступ то значи да ни један други процес тренутно не приступа тој локацији и да је она доступна. Чим изврши резервацију, дати процес може да пусти и друге да врше резервацију, а он сам може да настави са приступом. Ако пак није завршен приступ локацији од стране процеса који је претходно извршио резервацију, процес који тражи резервацију ће бити блокиран, као и сви други процеси који желе да изврше резервацију, док се не изврши резервација текуће локације. Тиме се обезбеђује да никада нема више резервација него што има места у баферу.

Раније смо имали ситуацију да само један процес потрошач може да приступа баферу, тако да је улазак потрошача у критичну секцију уједно обезбеђивао и ексклузивност приступа дељеној променљивој *F* (којој се приступало унутар критичне секције). Међутим, у случају када више процеса потрошача може да приступа баферу истовремено, ексклузиван приступ дељеној променљивој *F* се мора обезбедити увођењем посебног семафора *mutexF*.

Овде је дато решење са поделом посла унапред:

```
program Integration;
const size = 10; Npointgen = 10; Nintegral = 10;
type point = record x, y : real end;
var Xmin, Xmax, Ymin, Ymax, dx, dy : real;
    N, i : integer;
    F, front, rear : shared integer;
    buffer : shared array[0..size-1] of point;
    full, empty : array[0..size-1] of semaphore;
    mutexF, mutexRear, mutexFront : semaphore;

function f(x : real) : real; begin ... end;
function fmin(Xmin, Xmax, dx : real) : real; begin ... end;
function fmax(Xmin, Xmax, dx : real) : real; begin ... end;
function distribp(i : integer) : integer; begin ... end;
function distribi(i : integer) : integer; begin ... end;

procedure pointgen(j : 0..Npointgen-1; n : integer);
var i, slot : integer;
    p : point;
begin
    for i := 1 to n do begin
        wait(mutexRear);
        slot := rear;
        rear := (rear + 1) mod size;
        signal(mutexRear);

        p.x := Xmin + random(trunc(Xmax - Xmin));
        p.y := Ymin + random(trunc(Ymax - Ymin));
        wait(empty[slot]);
        buffer[slot] := p;
```

```

    signal(full[slot])
end
end;

procedure integral(j : 0..Nintegral-1; n : integer);
var i, slot : integer;
p : point;
begin
  for i := 1 to n do begin
    wait(mutexFront);
    slot := front;
    front := (front + 1) mod size;
    signal(mutexFront);

    wait(full[slot]);
    p := buffer[slot];
    signal(empty[slot]);

    wait(mutexF);
    if ((p.y <= f(p.x)) and (p.y >= 0)) then F := F + 1;
    if ((p.y >= f(p.x)) and (p.y <= 0)) then F := F - 1;
    signal(mutexF)
  end
end;

begin
  write('N = '); readln(N);
  write('Xmin = '); readln(Xmin);
  write('Xmax = '); readln(Xmax);
  dx := (Xmax - Xmin) / N;
  Ymin := fmin(Xmin, Xmax, dx);
  Ymax := fmax(Xmin, Xmax, dx);
  front := 0; rear := 0;
  init(mutexRear, 1); init(mutexFront, 1);
  for i := 0 to size-1 do init(empty[i], 1);
  for i := 0 to size-1 do init(full[i], 0);
  F := 0; init(mutexF, 1);
  cobegin
    pointgen(0, distribp(0));
    .
    .
    pointgen(Npointgen - 1, distribp(Npointgen - 1));
    integral(0, distribi(0));
    .
    .
    integral(Nintegral - 1, distribi(Nintegral - 1))
  coend;
  writeln('integral = ', F / N * (Xmax - Xmin) * (Ymax - Ymin))
end.

```

У датом решењу сви потрошачи приступају истој дељеној променљивој  $F$ . Да би се том приликом избегло чекање на семафору  $mutexF$ , сваки потрошач може имати своју локалну променљиву  $fLoc$  која ће акумулирати резултат рада датог процеса и којој може приступати без ограничења. По завршетку рада свих потрошача резултат се израчунава као суме ових локалних вредности.

```

program Integration;
const size = 10; Npointgen = 10; Nintegral = 10;
type point = record x, y : real end;

var Xmin, Xmax, Ymin, Ymax, dx, dy : real;
N, i : integer;
Flocal : array[0..Nintegral-1] of integer;
F, front, rear : shared integer;
buffer : shared array[0..size-1] of point;
full, empty : array[0..size-1] of semaphore;
mutexRear, mutexFront : semaphore;
...
procedure integral(j : 0..Nintegral-1; n : integer;
                    var Floc : integer);
var i, slot : integer;
p : point;
begin
  for i := 1 to n do begin
    wait(mutexFront);
    slot := front;
    front := (front + 1) mod size;
    signal(mutexFront);
    wait(full[slot]);
    p := buffer[slot];
    signal(empty[slot]);
    if((p.y <= f(p.x)) and (p.y >= 0)) then Floc := Floc + 1;
    if((p.y >= f(p.x)) and (p.y <= 0)) then Floc := Floc - 1;
  end;
end;
begin
  ...
cobegin
  pointgen(0,distribp(0));
  ...
  pointgen(Npointgen-1,distribp(Npointgen-1));
  integral(0,distribi(0),Flocal[0]);
  ...
  integral(Nintegral-1,distribi(Nintegral-1),
            Flocal[Nintegral-1]);
coend;
  for i := 0 to Nintegral-1 do F := F + Flocal[i];
  writeln('integral = ', F / N * (Xmax - Xmin) * (Ymax - Ymin))
end.

```

На сличан начин као под в) може се применити и решење са поделом посла по принципу обраде помоћу „торбе послова“.

→ Решити проблем из тачке в) применом метода „поделе посла унапред“, тако да подела посла буде урађена према неком другом критеријуму (нпр. тако што ће сваки процес *pointgen(i)* генерисати тачке из одређене подобласти области интеграције). Да ли промена процеса произвођача утиче на процесе потрошаче?

→ Решити проблем из тачке в) применом обраде по принципу „торбе послова” тако да јединични посао за произвођаче буде генерисање *NPTaskSize* тачака, а за потрошаче *NCTaskSize* тачака. Постоје ли оптималне вредности за *NPTaskSize* и *NCTaskSize*?

→ Шта је потребно урадити како би се по угледу на бафер моделовао систем складишта? Код рада са складиштима постоји већи број камиона који истоварују робу (производићача) у спремишту у складишту и камиона који утоварују робу (потрошача) на друге камионе. Време потребно за утовар или истовар робе је неодређено дуго. Може се десити да је камион који је дошао касније раније завршио истовар своје робе него камион који је дошао раније. Робу би требало утоварати у нови камион чим постане доступна.

Како би се решио овај проблем могу се искористити два бафера коначног капацитета описана у овом задатку. У првом баферу би сечували идентификатори празних спремишта, а у другом идентификатори пуних спремишта. Када нађе камион за истовар он узима индекс првог слободног спремишта из бафера. Ако нема слободних спремишта чека док се неко спремиште не испразни. Када добије идентификатор слободног спремишта креће са истоваром робе. Када заврши са истоваром робе у бафер пуних спремишта убацује идентификатор спремишта које је управо напунио. Када нађе камион који жели да утоварује робу прво приступа баферу пуних спремишта. Уколико нема пуних спремишта чека док се барем једно не напуни. Када добије идентификатор пуног спремишта креће да преносом робе у камион. Када заврши са пражњењем спремишта у бафер слободних спремишта убацује идентификатор спремишта које је управо испразнио. На почетку рада бафер празних спремишта садржи идентификатор свих спремишта, а бафер пуних спремишта је празан.

```

procedure Producer;
var id : integer;
begin
    id := get(buffer1);
    //istovar
    put(buffer2, id);
end;

procedure Consumer;
var id : integer;
begin
    id := get(buffer2);
    //utovar
    put(buffer1, id);
end;

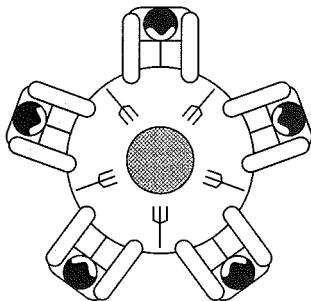
procedure init;
var i : integer;
begin
    for i := 1 to N do
        put(buffer1, i);
end;

```

## 4

## Филозофи за ручком

Пет филозофа седи око округлог стола (*The Dining Philosophers Problem*). Сваки филозоф проводи свој живот тако што наизменично размишља и једе. На средини стола је велика посуда са шпагетима. Пошто су шпагети дугачки и испреплетани, а филозофи нису веома спретни, морају да користе две виљушке када једу. Нажалост, постоји само пет виљушака, између свака два суседна филозофа по једна. Филозофи могу да дохвате само виљушку која је непосредно лево и непосредно десно од њих. Коришћењем семафора написати програм којим се симулира понашање филозофа. Размотрити решење са становишта перформанси и могућности да дође до мртвог блокирања (*deadlock*), живог блокирања ( *livelock*) и изгладњивања (*starvation*) процеса.



Слика 3. Филозофи за ручком. Сваки од пет филозофа наизменично мисли и једе. Да би јео потребне су му две виљушке. Сваки филозоф може да узме само виљушку непосредно лево и десно од њега. Укупно има само пет виљушака, по једна између свака два филозофа.

Јасно је да два суседна филозофа не могу јести истовремено, као и да истовремено могу јести највише два филозофа. Претпоставити да су периоди размишљања и једења случајне величине.

## Решење

Процес који симулира акције филозофа (којих има  $N$ ) се може скицирати на следећи начин:

```
procedure Philosopher(j:0..N-1);
begin
  while (true) do begin
    think;
    acquire_forks;
    eat;
    release_forks;
  end;
end;
```

Пошто два филозофа могу истовремено захтевати приступ једној виљушци, потребно је да се обезбеди међусобно искључивање увођењем семафора. Виљушке се не могу

посматрати као један ресурс са више јединица истог типа, јер са становишта филозофа оне нису једнаке (с обзиром да сваки филозоф може узети само одређене виљушке – оне лево и десно од себе). Приступ виљушкама, dakле, не може бити реализован помоћу генералног (бројачког) семафора, већ се за сваку виљушку мора увести по један семафор, што је учињено увођењем низа семафора *mutexfork*.

Пошто је живот свих филозофа исти (сви имају исти код), сви извршавају исте акције и истим редоследом (само евентуално у различито време). На пример, сваки филозоф када жели да једе узима прво једну виљушку (рецимо леву), а потом и другу (десну). У неком тренутку се чак може десити да сви филозофи узму леву виљушку *ћре* него што је и један стигао да узме и другу (десну). Тада настаје ситуација када сваки филозоф држи виљушку у својој левој руци, чекајући свог десног суседа да спусти виљушку, што доводи до блокирања свих филозофа. Оваква ситуација, где процеси не могу да наставе рад због међусобног чекања на задовољење одговарајућих услова, се назива мртво блокирање (*deadlock*).

*Deadlock* настаје ако су задовољена следећа четири услова:

- Процеси имају право ексклузивног приступа ресурсима (*Mutual exclusion*).
- Процес држи ресурс све време док чека приступ другом ресурсу (*Wait and hold*).
- Процес не може преузети ресурс на који чека (*No preemption*).
- Постоји затворен круг процеса који чекају један другог - један процес чека на испуњење услова који обезбеђује други процес, други процес чека на испуњење услова који обезбеђује трећи процес итд., а последњи процес у том низу чека на испуњење услова који обезбеђује први процес (*Circular wait*).

Елиминацијом неког од услова из ове листе може се спречити настанак мртвог блокирања (*deadlock*). Овде треба водити рачуна о томе да се елиминацијом не наруши исправности рада програма.

Решење треба да спречава могућност изгладњивања процеса (*starvation*). Изгладњивање је ситуација у којој процес не може да настави рад јер када жели да приступи критичној секцији, то никада не успе, већ други процеси увек успевају да добију приступ пре њега. Нпр. ако први и трећи филозоф наизменично размишљају и једу, други филозоф ће изгладнети, јер не може истовремено да дође до две слободне виљушке; увек један од његових суседа једе. Ова ситуација је могућа зато што редослед приступа критичној секцији није ничим гарантован - филозоф који је завршио са јелом и вратио виљушке на сто може опет узети виљушке пре својих суседа.

Решење код кога се ни један процес не може блокирати се назива праведно. Имајући у виду претходну дискусију, праведност (*fairness*) се може дефинисати као одсуство изгладњивања.

Приметимо да се интуитивно значење појма праведност не поклапа са претходном дефиницијом. Код горе наведене дефиниције праведност је логичка вредност (решење је или праведно или није), док интуитивно праведност подразумева континуалну величину. Дефиниција праведности се може проширити тако да представља меру равноправности процеса приликом добијања права приступа критичној секцији. У том смислу посматрајмо просечан период времена  $T(p)=1/Ni*\sum Ti(p)$ , ( $i=1..Ni$ ,  $Ni$  – број приступа критичној секцији од стране процеса), које процес  $p$  ( $p=1..Nr$ ,  $Nr$  – број процеса) мора да чека да би добио право приступа критичној секцији. Што је мања разлика између ових вредности за различите процесе  $p$ , решење је праведније. На основу овог закључка може се дефинисати формула која квантификује праведност применом неке од познатих метрика. С обзиром да би даљим разматрањем ушли у област којом се ова књига не бави, нећемо даље разматрати проблем праведности.

Треба само приметити да ће праведност увек бити максимална када је вероватноћа добијања приступа критичној секцији једнака за све процесе, без обзира на метрику.

Одсуство праведности за последицу може имати погоршање перформанси (дуже укупно време извршавања конкурентног програма). Ако неки процеси не добијају право приступа критичној секцији једнако брзо као други процеси, они ће напредовати спорије, тако да ће укупно време извршавања бити дуже. Као што смо видели, одсуство праведности, у крајњој инстанци, може довести до изгладњивања.

### Прво решење

Из листе потребних услова који доводе до настанка мртвог блокирања (*deadlock*) види се да је за његово избегавање доволно обезбедити да не може да дође до формирања затвореног круга процеса који се међусобно чекају. У ситуацијама када су сви процеси истог типа, што је случај и код проблема филозофа за ручком, настанак кружне зависности се може спречити. То се постиже тако што се бар један од процеса реализације тако да не извршава критичне синхронизационе операције истим редоследом као остали. У конкретном случају, један од процеса (рецимо за последњег филозофа) се може реализовати тако да узима прво своју десну па онда леву виљушку.

```
program DiningPhilosophers;
const N = 5;
var mutexfork : array[0..N-1] of semaphore;
    i : integer;

procedure think; begin ... end;
procedure eat; begin ... end;

procedure Philosopher(i : 0..N-2);
begin
    while (true) do begin
        think;
        wait(mutexfork[i]);
        wait(mutexfork[i+1]);
        eat;
        signal(mutexfork[i]);
        signal(mutexfork[i+1])
    end
end;
end;

procedure Philosopher(i : N-1..N-1);
begin
    while (true) do begin
        think;
        wait(mutexfork[0]);
        wait(mutexfork[N-1]);
        eat;
        signal(mutexfork[0]);
        signal(mutexfork[N-1])
    end
end;
end;

begin
    for i := 0 to N-1 do init(mutexfork[i],1);
    cobegin
        Philosopher(0);
```

```

...
Philosopher(N-1)
coend
end.

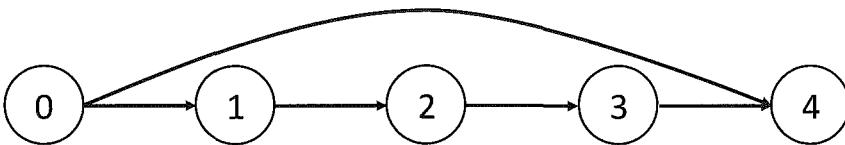
```

Питање праведности код приступа критичној секцији се готово увек решава на нивоу имплементације принципа међусобног искључивања у датој програмској парадигми. Овде то значи да би имплементација семафора морала да буде таква да обезбеди да сваки процес има подједнаку вероватноћу приступа семафору. Ако би се равноправност процеса обезбеђивала на нивоу програма, то би захтевало додатни напор од стране програмера, а притом би се смањила ефикасност и разумљивост програма.

Пошто је синхронизација између процеса минимална, ово решење је добро са становишта брзине извршавања.

Уколико, као код овог проблема, постоји информација о свим ресурсима којима се може приступити онда се правило коришћено за избегавање мртвог блокирања које је примењено у овом решењу може генерализовати. Код ове генерализације је потребно увести уређење свих ресурса као и редослед приступа датим ресурсима који је заснован на датом уређењу. Пошто је релација уређења својствена графовима онда се и ови протоколи приступања називају протоколи засновани на графовима (*Graph-Based Protocols*).

Нека је  $R$  скуп свих ресурса и нека су  $r_i$  и  $r_j$  два ресурса из датог скупа. Уколико се у дати скуп уведе релација парцијалног уређења ( $\rightarrow$ ) код које је ресурс  $r_i$  пре ресурса  $r_j$  ( $r_i \rightarrow r_j$ ) онда и сваки процес који приступа ресурсима  $r_i$  и  $r_j$  тим ресурсима мора да приступи тако што прво приступи ресурсу  $r_i$  па онда ресурсу  $r_j$ . Граф ресурса формиран на овај начин је директан ациклички граф. Граф ресурса би требало формирати на такав начин да води рачуна о томе који процес приступа којим ресурсима.



Слика 4. Коришћени граф ресурса

Специјални начина уређења ресурса је у виду кореног стабла код кога се сваком ресурсу додели позиција у стаблу (протокол у облику стабла - *Tree protocol*). За овај начин уређења се може формирати редослед приступања ресурсима код кога се на почетку може приступити било ком ресурсу, а касније се једном ресурсу може приступити само ако се тренутно приступа и родитељском ресурсу.

Један од начина уређења ресурса је линеарно уређење код кога се сваком ресурсу додели јединствен редни број. За овај начин уређења се може формирати редослед приступања ресурсима код кога се једном ресурсу може приступити само ако се тренутно не приступа ни једном ресурсу који има већи редни број од датог ресурса. Овај начин уређења не захтева познавање којим све ресурсима поједини процес приступа већ само скуп ресурса којим се приступа. Треба приметити да је овај начин уређења и приступа примене у овом решењу (слика 4).

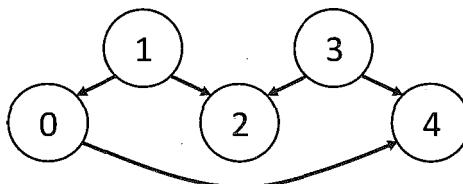
➔ Размислите о следећем: Да ли се редослед *signal* операција у процедуре *Philosopher* може променити? Ако може, има ли разлике између две могућности; ако не, зашто?

### Друго решење

Постоји још једно решење које се често помиње у литератури и које користи идеју да процеси не извршавају критичне синхронизационе операције истим редоследом. Код овог решења, сваки филозоф када жели да једе узима прво непарну вилјушку (може и обрнуто, свеједно). Филозофи и одговарајуће вилјушке су нумерисани. Променљиве *first* и *second* означавају коју од две вилјушке филозоф узима прву, а коју другу (слика 5).

```
program DiningPhilosophers;
const N = 5;
var mutexfork : array[0..N-1] of semaphore;
    i : integer;
procedure think; begin ... end;
procedure eat; begin ... end;

procedure Philosopher(i : 0..N-1);
var first, second: 0..N-1;
begin
    if (i mod 2 = 1) then begin
        first := i; {first fork that will be taken}
        second := (i+1) mod N {second fork that will be taken}
    end
    else begin
        first := (i+1) mod N; {first fork that will be taken}
        second := i {second fork that will be taken}
    end;
    while (true) do begin
        think;
        wait(mutexfork[first]);
        wait(mutexfork[second]);
        eat;
        signal(mutexfork[first]);
        signal(mutexfork[second])
    end
end;
begin
    for i := 0 to N-1 do init(mutexfork[i],1);
    cobegin
        Philosopher(0);
        ...
        Philosopher(N-1)
    coend
end.
```



Слика 5. Коришћени граф ресурса

Предност овог решења над претходним је што су сви процеси филозофи исти. Изгладњивање је и даље могуће из истог разлога као и у првом решењу.

### Треће решење

Други начин да се избегне deadlock је да се ограничи број процеса који приступају ресурсима (вилљушкама), тако да бар један од њих има услове да уђе у критичну секцију (тј. може да узме обе вилљушке). Тиме се спречава настанак кружне зависности. У случају проблема пет филозофа за ручком то значи да највише четири филозофа смеју да покушају да једу истовремено. То се може реализовати помоћу једног бројачког семафора *ticket*:

```
program DiningPhilosophers;
```

```
const N = 5;
```

```
var mutexfork: array[0..N-1] of semaphore;
    ticket: semaphore;
    i: integer;

procedure think; begin ... end;
procedure eat; begin ... end;

procedure Philosopher(i : 0..N-1);
var left, right: 0..N-1;
begin
    left := i; {left fork}
    right := (i+1) mod N; {right fork}
    while (true) do begin
        think;
        wait(ticket);
        wait(mutexfork[left]);
        wait(mutexfork[right]);
        eat;
        signal(mutexfork[left]);
        signal(mutexfork[right]);
        signal(ticket)
    end
end;
end;

begin
    for i := 0 to N-1 do init(mutexfork[i],1);
    init(ticket,N-1);
    cobegin
        Philosopher(0);
        ...
        Philosopher(N-1)
    coend
end.
```

И код овог решења може доћи до изгладњивања, нпр. ако увек иста четири процеса добијају могућност да једу.

Ово решење је нешто сложеније јер захтева додатну синхронизацију због увођења семафора *ticket*, па су перформансе нешто слабије него у првом и другом решењу.

## Дељење променљиве: Семафори

### Четврто решење

Трећи начин да се избегне *deadlock* је да се детектује потенцијално опасна ситуација која може довести до узајамног блокирања и потом предузме одговарајућа активност како би се изашло из тог стања. У случају филозофа за ручком то се ради тако што сваки филозоф који је узео леву виљушку, а онда установио да не може да узме и десну, мора да врати узету виљушку назад на сто.

Међутим, применом овог алгоритма може доћи до тога да након што сви филозофи узму своје леве виљушки, сваки од њих установи да нема десну виљушку пре него што иједан други успе да врати своју виљушку на сто, и да онда сви филозофи врате узете виљушки назад на сто, чиме се поново долази у почетну позицију. Нема гаранција да се овакав сценарио, у коме сви филозофи узимају своје леве виљушки, а затим их сви враћају назад на сто, не може понављати у бескрай. Ситуација коју карактерише непrekидно безуспешно понављање одређеног низа акција са циљем да се прође нека тачка синхронизације у програму зове се *livelock*. Она је по ефекту слична мртвом блокирању (*deadlock*), јер долази до заустављања напретка целог програма. Једина разлика је што у случају живог блокирања (*livelock*) процеси заправо врше запослено чекање.

Да би се спречио *livelock* по претходном сценарију, потребно је омогућити да бар један филозоф стигне да прво врати своју виљушку на сто пре него што неки од његових суседа провери да ли је она ту. То се може постићи тако што би се захтевало да протекне одређени период времена између узимања прве виљушки и тренутка провере да ли је друга виљушка на столу, при чему тај период мора бити различит за сваког филозофа (иначе се само временски продужава сценарио који може довести до живог блокирања (*livelock*)). У пракси се тај проблем може решити тако што се узме да одговарајући период буде нека случајна величина; статистички гледано, са порастом броја покушаја расте вероватноћа да ће неки филозоф успети да врати виљушку на сто на време да његов сусед може да је узме као своју другу виљушку. Вероватноћа да дође до живог блокирања (*livelock*) је једнака вероватноћи да ниједан филозоф не успе да врати виљушку пре него што је други филозоф узме и тежи нули са порастом броја покушаја.

Низ *fork* за сваку виљушку чува информацију да ли је на столу или не, а елементи низа *mutexfork* су семафори који служе за међусобно искључивање процеса код приступа одговарајућим елементима низа *fork*.

```
program DiningPhilosophers;
const N = 5;
      maxpause = 1000;

var mutexfork : array[0..N-1] of semaphore;
    fork : shared array[0..N-1] of boolean;
    i : integer;

procedure think; begin ... end;
procedure eat; begin ... end;

procedure Philosopher(i : 0..N-1);
var left, right : 0..N-1;
    first : boolean;
    again : boolean;
begin
    left := i; {left fork}
    right := (i+1) mod N; {right fork}
    while (true) do begin
```

```

think;
again := true;
while (again) do begin
    {keep testing till both forks are available}
    {take the first fork}
    wait(mutexfork[left]);
    first := fork[left];
    fork[left] := false;
    signal(mutexfork[left]);
    if first then begin
        {is the other fork on the table?}
        wait(mutexfork[right]);
        if fork[right] then begin
            {yes -> take the other fork as well}
            fork[right] := false;
            signal(mutexfork[right]);
            eat;
            {return the forks to the table}
            wait(mutexfork[left]);
            fork[left] := true;
            signal(mutexfork[left]);
            wait(mutexfork[right]);
            fork[right] := true;
            signal(mutexfork[right]);
            again := false;
        end
        else begin
            {no -> return the taken fork to the table}
            signal(mutexfork[right]);
            wait(mutexfork[left]);
            fork[left] := true;
            signal(mutexfork[left]);
            pause(random(maxpause))
        end
    end
    else
        pause(random(maxpause));
end;
end;

begin
    for i := 0 to N-1 do begin
        init(mutexfork[i],1);
        fork[i] := true
    end;
    cobegin
        Philosopher(0);
        ...
        Philosopher(N-1)
    coend
end.

```

Треба приметити да редослед приступа семафорима приликом враћања виљушки битан. Прво је потребно ослободити семафор десне виљушке па је тек након тога

потребно приступити семафору леве вилјушке. У супротном је могуће доћи до узајамног блокирања (*deadlock*), код кога сваки филозоф држи семафор своје десне вилјушке а не може приступити семафору леве вилјушке, што се овим решењем управо и желело избећи.

Могућност да дође до изгладњивања код овог решења мери се вероватноћом да случајно генерисано време чекања за неког филозофа увек буде мање од времена чекања његових суседа, уз претпоставку да се суседи непрестано смењују у јелу. Ова вероватноћа тежи нули са растом броја покушаја.

Перформанс овог решења су слабије у односу на прва два решења јер је потребно да прође неко време док се не појави одговарајућа комбинација генерисаних периода; такође, у случају када се узима вилјушка, извршава се два пута више синхронизационих операција него код првог решења.

#### Пето решење

Пето решење комбинује идеје из првог и другог решења (да до мртвог блокирања (*deadlock*) не може доћи ако бар један филозоф узме вилјушку другом руком у односу на неког од других филозофа) и четвртог решења (да се *deadlock* може спречити тако што ће сваки филозоф који је узео једну, а нема другу вилјушку поред себе, да прву врати назад на сто како би други филозофи могли да је искористе).

Увођењем случајног избора руке којом филозоф узима прву вилјушку, постоји релативно велика вероватноћа једнака  $(2^{N-2})/2^N$ , где је  $N$  број филозофа, да ће бар један од филозофа узети вилјушку другом руком у односу на остале, што онемогућује *deadlock*. Ако се, пак, деси и да сви филозофи узму вилјушку истом руком, ниједан неће имати другу вилјушку, па ће сви вратити своју назад на сто. Пошто се сваки пут поново случајно бира рука којом филозофи узимају вилјушку, вероватноћа да се поново деси да сви филозофи истом руком узму вилјушку је мања него да се то деси само једном. Вероватноћа да дође до живог блокирања (*livelock*), односно да сви филозофи увек истом руком узимају вилјушке, тежи нули са порастом броја покушаја.

Могућност да дође до изгладњивања филозофа се мери вероватноћом да се стално дешава да његови суседи једу увек истом руком и да се наизменично смењују у јелу. Ова вероватноћа тежи нули са порастом броја покушаја.

Перформанс оваквог решења су боље од претходног, јер нема додатног чекања.

```
program DiningPhilosophers;
const N = 5;
      maxpause = 1000;

var mutexfork : array[0..N-1] of semaphore;
    fork : shared array[0..N-1] of boolean;
    i : integer;

procedure think; begin ... end;
procedure eat; begin ... end;

procedure Philosopher(i : 0..N-1);
var left, right, first, second : 0..N-1;
    firstOK : boolean;
    again : boolean;
begin
    left := i; {left fork}
    right := (i+1) mod N; {right fork}
    while (true) do begin
        think;
```

```

again := true;
while (again) do begin
{keep testing till both forks are available}
{choose a hand}
if (random(1) < 0.5) then begin
    first := left;
    second := right
end
else begin
    first := right;
    second := left
end;
{take the first fork}
wait(mutexfork[first]);
firstOK := fork[first];
fork[first] := false;
signal(mutexfork[first]);
if firstOK then begin
{is the other fork on the table?}
    wait(mutexfork[second]);
    if fork[second] then begin
{yes -> take the other fork as well}
        fork[second] := false;
        signal(mutexfork[second]);
        eat;
{return the forks to the table}
        wait(mutexfork[first]);
        fork[first] := true;
        signal(mutexfork[first]);
        wait(mutexfork[second]);
        fork[second] := true;
        signal(mutexfork[second]);
        again := false;
    end
    else begin
{no -> return the taken fork to the table}
        signal(mutexfork[second]);
        wait(mutexfork[first]);
        fork[first] := true;
        signal(mutexfork[first]);
        pause(random(maxpause))
    end
end
else
    pause(random(maxpause));
end;
end;
begin
for i := 0 to N-1 do begin
    init(mutexfork[i],1);
    fork[i] := true
end;
cobegin
    Philosopher(0);

```

```
...  
    Philosopher(N-1)  
  coend  
end.
```

Ово решење је на неки начин разрада другог решења где сваки непарни филозоф прву виљушку узима левом руком, а сваки парни десном руком, са том разликом што је овде избор руке случајан, тако да успешност узимања две виљушке од стране бар једног филозофа није одмах гарантована.

#### Шесто решење

Постоји још један начин да се спречи *deadlock* код приступа виљушкама - атомизацијом промене стања на столу на нивоу једног филозофа. Филозофи, према том алгоритму, не би смели истовремено да приступају виљушкама, већ један по један, при чему би узимали или две виљушке или ниједну. Тада су могуће само две ситуације: (1) неки филозофи су већ узели две виљушке и једу, или (2) све виљушке су на столу, па ће први филозоф који добије право приступа моћи да узме две и почне да једе. Дакле, нема могућности за *deadlock*.

```
program DiningPhilosophers;  
const N = 5;  
  maxpause = 1000;  
  
var  forks_available : shared array[0..N-1] of 0..2;  
  forks : semaphore;  
  i : integer;  
  
procedure think; begin ... end;  
procedure eat; begin ... end;  
  
procedure Philosopher(i : 0..N-1);  
var  left, right : 0..N-1;  
  again : boolean;  
begin  
  left := (i - 1) mod N; {left neighbour}  
  right := (i + 1) mod N; {right neighbour}  
  while (true) do begin  
    think;  
    again := true;  
    while (again) do begin  
      {keep testing till both forks are available}  
      wait(forks);  
      if (forks_available[i] = 2) then begin  
        {take forks}  
        forks_available[left]:=forks_available[left]-1;  
        forks_available[right]:=forks_available[right]-1;  
        signal(forks);  
        eat;  
        {return the forks to the table}  
        wait(forks);  
        forks_available[left]:=forks_available[left]+1;  
        forks_available[right]:=forks_available[right]+1;  
        signal(forks);  
        again := false;  
      end  
      else begin
```

```

        signal (forks);
        pause (random (maxpause) );
    end;
end
begin
    for i := 0 to N-1 do forks_available[i] := 2;
    init (forks, 1);
    cobegin
        Philosopher(0);
        ...
        Philosopher(N-1)
    coend
end.

```

Елементи низа *forks\_available* представљају број слободних виљушки доступних сваком филозофу појединачно. Семафор *forks* служи за међусобно искључивање процеса код приступа виљушкама.

С обзиром да је протокол за приступ виљушкама „пасиван”, односно да се у случају када приступ виљушкама није слободан или када обе виљушке нису на столу не предузимају никакве додатне акције, значи да не постоје услови за настајање живог блокирања (*livelock*).

До изгладњивања може доћи ако механизам семафора не обезбеђује праведност; у том случају би се могло десити да неки филозоф никада не добије право приступа виљушкама. Други начин да дође до изгладњивања филозофа је ако његови суседи наизменично једу, јер он никада неће имати обадве виљушке на располагању.

Недостатак овог у односу на претходна решења је смањена конкурентност због тога што више филозофа не сме да приступа виљушкама истовремено. У зависности од тога колико се времена троши на јело, а колико на синхронизацију, ово може бити од већег или мањег значаја. Такође, провера да ли су обе виљушке слободне се врши у петљи, тј. у виду неке врсте запосленог чекања, што је неефикасно.

Да би се избегло запослено чекање, потребно је да сваки филозоф има семафор на коме ће чекати док се не испуни услов да може да једе. Због тога се уводи низ семафора *OK\_to\_Eat* чији елементи говоре да ли је приступ виљушкама за одговарајућег филозофа дозвољен или не.

У случају да обе виљушке нису на столу када филозоф *i* жели да једе, уместо да ради запослено чекање, он се блокира на семафору *OK\_to\_Eat[i]* и чека на сигнал од неког филозофа који враћа виљушке. Ту се мора обратити пажња на следеће: ако филозоф који враћа виљушке утврди да неки од његових суседа може да једе јер су му обе виљушке на столу, он сме да сигнализира *OK\_to\_Eat[neighbour]* суседу само ако он заиста чека да једе, тј. ако чека на семафору *OK\_to\_Eat[neighbour]*. У супротном се може десити да филозоф који враћа виљушке сигнализира *OK\_to\_Eat[neighbour]*, да потом неко други узме једну или обе виљушке филозофа *neighbour*, а да он и даље мисли да су обе његове виљушке на столу (јер је добио сигнал), што би довело до грешке.

Да би се избегла претходно описана ситуација уводи се низ *state* који за сваког филозофа памти у ком се стању налази; стања су: *thinking* (мисли), *hungry* (гладан, жели да једе) или *eating* (једе). Филозоф који враћа виљушку ће сигнализирати суседу *neighbour* да може да једе само ако је сусед у стању *hungry* и број њему расположивих

## Дељене променљиве: Семафори

виљушака има вредност 2, што одговара ситуацији када филозоф чека на семафору *OK\_to\_Eat[neighbour]*.

```
program DiningPhilosophers;
const N = 5;

type philosopher_state = (thinking, hungry, eating);
var forks_available : shared array[0..N-1] of 0..2;
forks : semaphore;
OK_to_Eat : array[0..N-1] of semaphore;
state : shared array[0..N-1] of philosopher_state;
i : integer;

procedure think; begin ... end;
procedure eat; begin ... end;

procedure Philosopher(i : 0..N-1);
var left, right : 0..N-1;
begin
    left := (i - 1) mod N; {left neighbour}
    right := (i + 1) mod N; {right neighbour}
    while (true) do begin
        think;
        {get forks}
        wait(forks);
        state[i] := hungry;
        if (forks_available[i] = 2) then begin
            forks_available[left] := forks_available[left]-1;
            forks_available[right] := forks_available[right]-1;
            state[i] := eating;
            signal(OK_to_Eat[i]);
        end;
        signal(forks);
        wait(OK_to_Eat[i]);
        eat;
        {return forks}
        wait(forks);
        state[i] := thinking;
        forks_available[left] := forks_available[left] + 1;
        forks_available[right] := forks_available[right] + 1;
        if ((forks_available[left] = 2)
            and (state[left] = hungry)) then begin
            forks_available[(left - 1) mod N]
                := forks_available[(left - 1) mod N] - 1;
            forks_available[(left + 1) mod N]
                := forks_available[(left + 1) mod N] - 1;
            state[left] := eating;
            signal(OK_to_Eat[left]);
        end;
        if ((forks_available[right] = 2)
            and (state[right] = hungry)) then begin
            forks_available[(right - 1) mod N]
                := forks_available[(right - 1) mod N] - 1;
            forks_available[(right + 1) mod N]
```

```

        := forks_available[(right + 1) mod N] - 1;
        state[right] := eating;
        signal(OK_to_Eat[right]);
    end;
    signal(forks);
end
begin
    for i := 0 to N-1 do forks_available[i] := 2;
    for i := 0 to N-1 do init(OK_to_Eat[i], 0);
    for i := 0 to N-1 do state[i] := thinking;
    init(forks, 1);
    cobegin
        Philosopher(0);
        ...
        Philosopher(N-1)
    coend
end.

```

Када филозоф жели да једе, најпре тражи приступ виљушкама и испитује да ли су обе виљушке на столу. Ако јесу, узима их, мења стање у *eating*, ослобађа приступ виљушкама и почиње да једе. Ако нису, ослобађа приступ виљушкама и чека док се обе виљушке не нађу на столу поред њега, што ће му сигнализирати неки од суседних филозофа који враћа виљушке на сто. Када заврши са јелом, најпре чека да добије право приступа виљушкама, а потом, када се то деси, мења стање у *thinking*, враћа виљушке на сто и на крају сигнализира суседима који евентуално чекају на њих (налазе се у стању *hungry*) да могу да их узму.

Промене стања се врше искључиво у критичној секцији коју контролише семафор *forks*, тако да су све промене стања атомске. Таквим решењем је донекле смањена конкурентност, али извршавање у критичним секцијама контролисаним семафором *forks* траје кратко.

→ Коментарисати следеће решење проблема филозофа за ручком: Све виљушке су нумерисане од 1 до *N*. Сваки филозоф узима прво парну виљушку која се налази поред њега. Ако нису доступне обе виљушке, филозоф враћа узету виљушку назад на сто. Након завршетка јела, може се десити да филозоф замени леву и десну виљушку приликом враћања виљушки на сто.

## Одговор

Код овог решења се рачуна са тим да ће се крај бар једног филозофа у неком тренутку наћи две виљушке од којих је бар једна парна. Протокол за приступ виљушкама је „активан”, што значи да процес у случају да не може да уђе у критичну секцију предузима одређене акције (враћа виљушку на сто) и потом покушава поново. Код оваквих решења потенцијално може доћи до живог блокирања (*livelock*), али је то овде немогуће, јер ће увек бар један филозоф моћи да узме две виљушке. Вероватноћа да дође до изгладњивања једнака је вероватноћи да се крај неког од филозофа никад не нађу две виљушке од којих је једна парна, а она са временом тежи нули. Ефикасност није максимална јер филозофи који крај себе имају две непарне виљушке не једу иако постоје услови за то.

➔ Да ли је могуће елиминисати низ *forks\_available* код шестог решења проблема филозофа за ручком?

### Одговор

Могуће је. Овај низ служи за праћење броја виљушки поред сваког филозофа. Треба, међутим, приметити да је једино што нас занима да ли је број виљушки поред неког филозофа једнак 2 (а не колико тачно има виљушки поред датог филозофа). Одговарајуће стање се може изразити преко низа *state*:

*forks\_available[i]=2*  $\Leftrightarrow$  *state[left]<>eating and state[right]<>eating*

Заменом услова у програму добија се следеће решење:

```
program DiningPhilosophers;
const N = 5;

type philosopher_state = (thinking, hungry, eating);

var OK_to_Eat : array[0..N-1] of semaphore;
    state : shared array[0..N-1] of philosopher_state;
    mutexstate : semaphore;
    i : integer;

function getleft(i : integer) : integer;
begin
    if (i = 1) then getleft := N else getleft := i - 1
end;

function getright(i : integer) : integer;
begin
    if (i = N) then getright := 1 else getright := i + 1
end;

procedure think; begin ... end;
procedure eat; begin ... end;

procedure Philosopher(i : 0..N-1);
var left, right : 0..N-1;
begin
    left := getleft(i); {left neighbour}
    right := getright(i); {right neighbour}
    while (true) do begin
        think;
        {take forks}
        wait(mutexstate);
        state[i] := hungry;
        if ((state[left] <> eating)
            and (state[right] <> eating)) then begin
            state[i] := eating;
            signal(OK_to_Eat[i]);
        end;
        signal(mutexstate);
        wait(OK_to_Eat[i]);
        eat;
        {put forks}
    end;
end;
```

```

wait(mutexstate);
state[i] := thinking;
if ((state[getleft(left)] <> eating) and
    (state[getright(left)] <> eating) and
    (state[left] = hungry)) then begin
    state[left] := eating;
    signal(OK_to_Eat[left]);
end;
if ((state[getleft(right)] <> eating) and
    (state[getright(right)] <> eating) and
    (state[right] = hungry)) then begin
    state[right] := eating;
    signal(OK_to_Eat[right]);
end;
signal(mutexstate);
end
end;

begin
for i := 0 to N-1 do init(OK_to_Eat[i], 0);
for i := 0 to N-1 do state[i] := thinking;
init (mutexstate, 1);
cobegin
    Philosopher(0);
    ...
    Philosopher(N-1)
coend
end.

```

На крају, ради потпуности, дефинисаћемо појам коректности (*correctness*), који се често среће у литератури. Коректност је особина програма која значи да ће се програм гарантовано извршити у складу са очекивањем. Коректност обухвата следеће две класе особина:

Особине сигурности (*safety properties*) – оне гарантују да се ништа „лоше“ неће дододити у програму, односно да ће програм радити коректно. То подразумева да нема уткривања и мртвог блокирања.

Особине живости (*liveness properties*) – оне гарантују да ће се нешто „добро“ сигурно дододити, односно да ће програм сигурно напредовати. То подразумева одсуство животог блокирања и изгладњивања.

## 5

## Читаоци и писци

Два типа процеса, читаоци и писци, приступају једном запису (у општем случају запис може припадати некој колекцији података - бази података, датотеци, низу, уланчаној листи, табели итд.) (*The Readers – Writers Problem*). Читаоци само читају садржај записа, а писци могу да читају и мењају садржај записа. Да не би дошло до нерегуларне ситуације у којој запису истовремено приступа више писаца или истовремено приступају и писци и читаоци, писци имају право ексклузивног приступа. Са друге стране, дозвољено је да више читалаца истовремено приступа запису (нема ограничења у њиховом броју). Написати програм којим се реализује рад процеса читалаца и писаца. Размотрити решење са становишта праведности приступа.

## Решење

### Reader's preference solution – решење где предност имају читаоци

Приступ који је често применљив код проблема у којима постоји потреба за међусобним искључивањем процеса, јесте да се најпре формира једно рестриктивније, али једноставније решење проблема, а да се то решење затим релаксира до жељеног нивоа.

Код проблема читалаца и писаца рестриктивно решење би било да сваки писац и читалац ексклузивно приступају запису. У том случају је доволно увести један семафор (назовимо га *rw*) који контролише улазак у критичну секцију. Скица таквог решења може се представити на следећи начин:

```
procedure Reader(i:0..Nreaders-1);
begin
    while (true) do begin
        ...
        wait(rw);
        {read the record;}
        signal(rw);
        ...
    end
end;

procedure Writer(i:0..Nwriters-1);
begin
    while (true) do begin
        ...
        wait(rw);
        {write the record;}
        signal(rw);
        ...
    end
end;
```

Ово решење треба релаксирати тако да процеси читаоци могу да се извршавају конкурентно. Треба уочити да је у циљу међусобног искључења читалаца и писаца потребно да само први читалац у групи добије право ексклузивног приступа запису, док наредни читаоци могу да приступају директно, без међусобног искључивања. Право

ексклузивног приступа читалац треба да врати само ако је он последњи процес у групи читалаца. Такво решење изгледа овако:

```

program ReadersWriters;
const Nreaders = 10;
Nwriters = 5;

var rw, mutexR : semaphore;
nr : shared integer;

procedure Reader(i : 0..Nreaders-1);
begin
  while (true) do begin
    ...
    wait(mutexR);
    nr := nr + 1;
    if (nr = 1) then wait(rw);
    signal(mutexR);
    {read the record;}
    wait(mutexR);
    nr := nr - 1;
    if (nr = 0) then signal(rw);
    signal(mutexR);
    ...
  end
end;

procedure Writer(i : 0..Nwriters-1);
begin
  while (true) do begin
    ...
    wait(rw);
    {write the record;}
    signal(rw);
    ...
  end
end;

begin
  init(rw, 1); init(mutexR, 1); nr := 0;
  cobegin
    Reader(0); ... Reader(Nreaders-1);
    Writer(0); ... Writer(Nwriters-1);
  coend
end.

```

Испитивање да ли је процес први или последњи у групи читалаца врши се уз помоћ бројача *nr* који броји читаоце који су приступили запису. Међусобно искључивање читалаца код приступа бројачу контролише се помоћу семафора *mutexR*. Када жели да приступи запису, читалац инкрементира бројач и испитује да ли је *nr=1*; ако јесте, онда је читалац први у групи и тражи право ексклузивног приступа запису; ако није, онда може слободно да чита. Слично, по завршетку приступа читалац декрементира бројач и испитује да ли је *nr=0*; ако јесте, онда је читалац последњи у групи и ослобађа приступ запису; ако није, онда може одмах да изађе.

Ово решење проблема читалаца и писаца назива се *reader's preference solution* јер даје предност читаоцима. Наиме, ако у неком тренутку читалац чита, а неки нови читалац и нови писац чекају да приступе запису, према датом решењу предност ће увек имати читалац. Дакле, ово решење није праведно. Непрекидни низ читалаца може трајно спречавати писце да приступе запису, тако да може доћи до изгладњивања писаца. Особина живости није испуњена.

Изгладњивање се може елиминисати увођењем реда међу све процесе који треба да провере и евентуално сачекају на испуњење различитих услова како би наставили са својим даљим радом. Сваки процеси који долазе да провери да ли су испуњени услови за рад прво чека у датом реду па тек када се претходна (било која) провера услова заврши може да приступи својој провери услова и евентуално блокирању док се тај услов не испуни. Процеси који не треба да чекају на условима не чекају ни у овом јединственом реду већ обављају своје операције. Увођењем овог реда само за процесе који проверавају услове се постиже да само један процес који проверава услов буде у својој провери услова и чека да услов постане испуњен, док процеси који не чекају на неком услову не треба да се блокирају у реду већ могу да постављају нову вредност.

Увођење овог реда се може постићи додавањем семафора који ограничава приступ ресурсима, *enter*. Читаоци и писци на почетку прво чекају на овом семафору, након тога проверавају услов да ли могу да наставе са радом, читањем односно писањем. Када буде испуњен услов за наставак рада пуштају наредни процес да обавља своју проверу услова сигнализацијом на овом семафору. Читаоци/писци након операције читања односно писања постављају нове вредности услова за блокиран процес. На овај начин је постигнуто да када неки процес дође на ред за проверу услова сигурно највише што може да чека је да сви они процеси који су већ започели своју операцију ту операцију и заврше. Није могуће да неки процес, који дође после тренутка када је дати процес стигао до провере услова, крене са извршавањем пре датог процеса. Овде треба нагласити да уколико уведени ред, то јест ограничавајући семафор, није праведни ни само решење неће бити праведно.

```
program ReadersWriters;
const Nreaders = 10;
Nwriters = 5;
var rw, mutexR, enter : semaphore;
    nr : shared integer;
procedure Reader(i : 0..Nreaders-1);
begin
    while (true) do begin
        ...
        wait(enter);
        wait(mutexR);
        nr := nr + 1;
        if (nr = 1) then wait(rw);
        signal(mutexR);
        signal(enter);
        {read the record;}
        wait(mutexR);
        nr := nr - 1;
        if (nr = 0) then signal(rw);
        signal(mutexR);
        ...
    end
end;
procedure Writer(i : 0..Nwriters-1);
```

```

begin
    while (true) do begin
        ...
        wait(enter);
        wait(rw);
        signal(enter);
        {write the record;}
        signal(rw);
        ...
    end
end;
begin
    init(rw, 1); init(mutexR, 1); init(enter, 1); nr := 0;
    cobegin
        Reader(0); ... Reader(Nreaders-1);
        Writer(0); ... Writer(Nwriters-1);
    coend
end.

```

Такође треба приметити да писац може да ослободи ограничавајући семафор и након операције писања. Ово решење може бити нешто мало спорије јер тек након операције уписа први наредни процес може да прође кроз ред и стигне до провере свог услова, где ће се блокирати. Решење би у том случају гласило:

```

procedure Writer_less_concurrent(i : 0..Nwriters-1);
begin
    while (true) do begin
        ...
        wait(enter);
        wait(rw);
        {write the record;}
        signal(rw);
        signal(enter);
        ...
    end
end;

```

#### Passing the baton – „предаја штафетне палице”

У претходном решењу се пошло од принципа међусобног искључивања процеса. Пажња је била усмерена ка томе да писци искључују једни друге и да читаоци као класа искључују писце. Као резултат је добијено решење које представља комбинацију приступа одговарајућим двема критичним секцијама – једне за приступ бројачу *nr*, а друге за приступ самом запису.

Други приступ проблему јесте да се специфицирају услови под којима неки процес сме да приступа запису. Један једноставан начин је да се уведу два бројача који броје писце и читаоце који приступају запису и да се онда услов за приступ изрази преко вредности бројача.

Нека су *nr* и *lw* позитивне целобројне вредности које представљају број читалаца и писаца који приступају запису, респективно. Приступ запису није дозвољен када су истовремено *nr* и *lw* позитивни или када је *lw>1* (први услов значи да писци и читаоци не могу запису приступати истовремено, а други да може постојати само један активан писац), што се може формулисати следећим логичким условом:

noRW:  $(nr > 0 \text{ AND } nw > 0) \text{ OR } nw > 1$

Скуп дозвољених стања  $RW$  је комплементаран скупу недозвољених стања  $noRW$ :

RW:  $(nr = 0 \text{ OR } nw = 0) \text{ AND } nw \leq 1$

Решење проблема се грубо може скицирати на следећи начин: Читалац најпре испитује да ли је испуњен услов да приступи запису; ако је услов испуњен, онда инкрементира бројач  $nr$  и затим чита запис, а ако услов није испуњен, онда треба да чека док се услов не испуни. Након завршетка читања бројач  $nr$  се декрементира. Слично томе, писац најпре испитује да ли је испуњен услов да приступи запису и ако јесте, онда инкрементира бројач  $nw$  и потом врши уписивање; у супротном, мора да чека да се услов испуни. Након завршетка уписа бројач  $nw$  се декрементира.

Остало је да се одреде услови под којима читалац и писац смеју да приступе запису, за шта ћемо искористити услов  $RW$ . Притом имајмо на уму да услов  $RW$  описује дозвољено стање у сваком тренутку. Дакле, он важи и након што су читалац или писац већ приступили запису, тј. након што су одговарајући бројачи инкрементирани.

Претпоставимо да читалац има услов за приступ. Након инкрементирања бројача  $nr$  његова вредност ће бити позитивна ( $nr > 0$ ), па ће се услов  $RW$  свести на ( $false \text{ OR } nw=0$ )  $\text{AND } nw \leq 1$ , што је еквивалентно услову  $nw=0 \text{ AND } nw \leq 1$ , а то је опет еквивалентно услову  $nw=0$ . Услов  $RW$  мора бити задовољен све време, тј. и пре и непосредно после приступа запису. Улазак читаоца у критичну секцију мора бити тако условљен да ћосле сваког инкрементирања бројача  $nr$  важи  $nw=0$ , чиме се услов пре уласка своди само на  $nw=0$ .

Слично томе, претпоставимо да писац има услов за приступ. Након инкрементирања бројача  $nw$  његова вредност ће бити позитивна ( $nw > 0$ ), па ће се услов  $RW$  свести на ( $nr=0 \text{ OR } false$ )  $\text{AND } nw=1$ , што је еквивалентно услову  $nr=0 \text{ AND } nw=1$ . Услов  $RW$  мора бити задовољен све време, тј. и пре и непосредно после приступа запису. Улазак писца у критичну секцију мора бити тако условљен да ћосле инкрементирања бројача  $nw$  важи  $nr=0 \text{ AND } nw=1$ , што се своди на услов пре уласка  $nr=0 \text{ AND } nw=0$ .

Скица процеса читалаца и писаца би сада изгледала овако:

```
procedure Reader(i : 0..Nreaders-1);
begin
    while (true) do begin
        ...
        <await (nw = 0) nr := nr + 1;>
        {read the record;}
        <nr := nr - 1;>
        ...
    end
end;

procedure Writer(i : 0..Nwriters-1);
begin
    while (true) do begin
        ...
        <await (nr = 0 and nw = 0) nw := nw + 1;>
        {write the record;}
        <nw := nw - 1;>
        ...
    end
end;
```

Угласте заграде означавају да се наредбе унутар њих извршавају атомски. Наредба облика `<await (B) S>` означава да се низ наредби `S` може извршити само ако је испуњен услов `B`; ако услов није испуњен, процес се блокира све док се услов не испуни. `await` наредба се може реализовати помоћу семафора коришћењем технике „предаје штафетне палице” (*passing the baton*), коју ћемо сада описати.

Најпре уведимо семафор `e` (име семафора можемо асоцирати са речи *entry* - улаз) који има почетну вредност 1 и служи за контролу уласка у критичну секцију. Даље, сваком различитом услову који се јавља у некој `await` наредби се придржује по један семафор и један бројач. Семафор служи за блокирање процеса који чекају на испуњење услова, а бројач броји колико има блокираних процеса. У случају писаца и читалаца имамо две `await` наредбе са различитим условима, па су нам потребна два семафора и два бројача. Назовимо `r` семафор који је придржен услову у процесу *Reader*, `dr` бројач блокираних читалаца, `w` семафор који је придржен услову у процесу *Writer* и `dw` бројач блокираних писаца. Пошто у почетку нема блокираних писаца ни читалаца, иницијално су све вредности бројача и семафора једнаке 0.

Решење проблема са „предајом штафетне палице” би изгледало овако:

```

program ReadersWriters;
const Nreaders = 10;
      Nwriters = 5;

var   e, r, w : semaphore;
      dr, dw, nr, nw : shared integer;

procedure Reader(i : 0..Nreaders-1);
begin
  while (true) do begin
    {<await (nw = 0) nr := nr + 1;>}
    wait(e);
    if not(nw = 0) then begin
      dr := dr + 1; signal(e); wait(r)
    end;
    nr := nr + 1;
    SIGNAL { SIGNAL code alternatives for
              "passing the baton" are explained later }
    {read the record;}
    {<nr := nr - 1;>}
    wait(e);
    nr := nr - 1;
    SIGNAL
  end
end;

procedure Writer(i : 0..Nwriters-1);
begin
  while (true) do begin
    {<await (nr = 0 and nw = 0) nw := nw + 1;>}
    wait(e);
    if not((nr = 0) and (nw = 0)) then begin
      dw := dw + 1; signal(e); wait(w)
    end;
    nw := nw + 1;
    SIGNAL
  end
end;

```

```
{write the record; }
{<nw := nw - 1;>}
wait(e);
nw := nw - 1;
SIGNAL
end
end;

begin
    init(e, 1); init(r, 0); init(w, 0); dr:=0; dw:=0; nr:=0; nw:=0;
    cobegin
        Reader(0); ... Reader(Nreaders-1);
        Writer(0); ... Writer(Nwriters-1);
    coend
end.
```

Улога дела кода означеног са *SIGNAL* је да активира следећи процес који чека на неком од семафора:

**SIGNAL:**

```
if ((nw = 0) and (dr > 0)) then
    begin dr := dr - 1; signal(r); end
else if ((nr = 0) and (nw = 0) and (dw > 0)) then
    begin dw := dw - 1; signal(w); end
else
    signal(e);
```

Конкретно, ако нема активних писаца и има блокираних читалаца, тада се буди један од блокираних читалаца. Ако нема ни активних читалаца ни писаца, тада се буди један од блокираних писаца. Ако уопште нема блокираних процеса, што значи да ни један процес више није блокиран ни на једном услову из *await* наредбе, онда се ради операција *signal* на семафору е.

Да би се обезбедило међусобно искључивање, приметимо да свака критична секција изван *await* блока која приступа променљивама из неког *await* блока мора на уласку у блок имати *wait(e)*, а на изласку *SIGNAL*.

Семафор е и семафори који су додељени условима из *await* наредби (*r* и *w*) чине један расподељени бинарни семафор (*split binary semaphore*), јер само један од њих може имати вредност 1 у једном тренутку. Пошто свака путања извршавања програма почиње наредбом *wait* и завршава се наредбом *signal* на одговарајућем семафору (било да је то семафор е или неки од семафора којима су додељени услови из *await* наредби), јасно је да ће се наредбе из *await* блока извршавати са ексклузивним правом приступа. Такође, гарантовано је да ће се низ наредби *S* из *await* блока *<await (B) S>* извршавати само ако је услов *B* испуњен, јер се *S* извршава: 1) када процес при првом покушају уласка у *await* блок утврди да је услов *B* испуњен (видети улазни протокол) или 2) након што је процес деблокиран после чекања на семафору који је додељен услову *B*, што се може десити само услед испуњења тог услова (видети излазни протокол *SIGNAL*).

Техником „предаје штафетне палице“ се не уводи *deadlock* јер се операција *signal* на семафорима *r* и *w* у оквиру дела кода означеног са *SIGNAL* ради само ако неки процес већ чека на одговарајућем семафору или ће управостати у ред чекања на семафору (ова друга ситуација се дешава у случају да је управо извршена операција *signal(e)*, а још увек није извршена операција *wait* која јој следи). Пошто увек само један семафор има вредност 1, а сваки пут се деблокира неки од процеса који чекају на неком од семафора *e*, *r* и *w*, то значи да не постоји могућност настанка ситуације међусобног

чекања процеса која би водила настанку мртвог блокирања (*deadlock*). Треба приметити да семафори *e*, *r* и *w* чине расподељени бинарни семафор.

Име технике „предаја штафетне палице“ потиче од начина на који се предаје дозвола за улазак у критичну секцију. Када се процес налази у критичној секцији, то можемо поистоветити са држањем палице која представља дозволу за боравак у њој. Када процес дође до *SIGNAL* дела, онда „предаје палицу“ неком другом процесу који је чекао на семафору придржаном одговарајућем услову у *await* наредби (под претпоставком да је *await* услов испуњен). Ако нема ниједног блокираног процеса чији је *await* услов испуњен, палица се предаје следећем процесу који жели да уђе у критичну секцију по први пут (тј. који чека на семафору *e*).

У датом решењу на многим местима се део кода означен са *SIGNAL* може упростити. Код процеса читалаца, пре извршавања првог *SIGNAL* (на крају улазног протокола код читаоца), *nr* је позитивно, а *nw* је нула. Стога се тај *SIGNAL* фрагмент може заменити са:

```
if (dr > 0)
  then begin dr := dr - 1; signal(r) end
else signal(e);
```

Пре извршавања другог *SIGNAL* (на крају излазног протокола код читаоца) *dr* и *nw* су нула, што своди *SIGNAL* на:

```
if (nr = 0 and dw > 0)
  then begin dw := dw - 1; signal(w) end
else signal(e);
```

Код процеса писаца, пре извршавања дела *SIGNAL* на крају улазног протокола, *nr* је нула, а *nw* је позитивно, па се овај фрагмент може заменити са:

```
signal(e).
```

Пре другог *SIGNAL* (на крају излазног протокола писца), *nr* и *nw* су нула, па се тај *SIGNAL* фрагмент своди на:

```
if (dr > 0) then begin dr := dr - 1; signal(r) end
else if (dw > 0) then begin dw := dw - 1; signal(w); end
else signal(e);
```

Конечно решење изгледа овако:

```
program ReadersWriters;
const Nreaders = 10;
      Nwriters = 5;

var   e,r,w : semaphore;
      dr,dw,nr,nw : shared integer;

procedure Reader(i : 0..Nreaders-1);
begin
  while (true) do begin
    wait(e);
    if (nw > 0) then begin
      dr := dr + 1; signal(e); wait(r)
    end;
    nr := nr + 1;
    if (dr > 0) then begin
      dr := dr - 1; signal(r); end
    else signal(e);
  end;
end;
```

```

{read the record;}
wait(e);
nr := nr - 1;
if ((nr = 0) and (dw > 0)) then begin
    dw := dw - 1; signal(w); end
else signal(e);
end
end;

procedure Writer(i : 0..Nwriters-1);
begin
    while (true) do begin
        wait(e);
        if ((nr > 0) or (nw > 0)) then begin
            dw := dw + 1; signal(e); wait(w)
        end;
        nw := nw + 1;
        signal(e);
        {write the record;}
        wait(e);
        nw := nw - 1;
        if (dr > 0) then begin dr := dr - 1; signal(r) end
        else if (dw > 0) then begin dw := dw - 1; signal(w) end
        else signal(e)
    end
end;

begin
    init(e, 1); init(r, 0); init(w, 0); dr:=0; dw:=0; nr:=0; nw:=0;
    cobegin
        Reader(0); ... Reader(Nreaders-1);
        Writer(0); ... Writer(Nwriters-1);
    coend
end.

```

И код овог решења читаоци имају предност над писцима. Наиме, када писац заврши приступ, прво се проверава да ли има блокираних читалаца и ако их има, деблокира се један од њих. Када тај читалац заврши, он „предаје палицу“ следећем блокираном читаоцу итд. Тако се палица предаје из руке у руку читалаца док се сви не деблокирају, тј. док не постане  $dr=0$ .

Предност технике „предаје штафетне палице“ је, међутим, у томе да се редослед „предаје палице“ може произвољно одређивати, чиме се може регулисати приоритет процеса. На пример, да би се добило решење у коме писци имају предност над читаоцима, потребно је обезбедити следеће:

- ако неки писац чека на приступ, новопридошли читаоци се морају блокирати
- блокирани читалац се деблокира само ако нема блокираних писаца

Први захтев се може обезбедити проширивањем услова за блокирање из прве **if** наредбе процеса читаоца:

```

if (nw > 0 or dw>0) then begin
    dr := dr + 1; signal(e); wait(r)
end;

```

Други захтев се може обезбедити заменом редоследа провере услова код последње *if* наредбе процеса писца:

```
if (dw > 0) then begin dw := dw - 1; signal(w) end  
else if (dr > 0) then begin dr := dr - 1; signal(r) end  
else signal(e)
```

Ове измене редоследа „предаје штафетне палице” не мењају основну структуру решења нити његову коректност.

Једно праведно решење у смислу права приступа запису, претпостављајући да су и same операције над семафорима реализоване праведно (тј. да операције не фаворизују ниједан семафор), јесте оно код кога се, у случајевима када су и писац и читалац блокирани, приоритет даје наизменично, једном писцу, а следећи пут читаоцу. Да би се добило такво решење потребно је:

- блокирати новог читаоца ако неки писац већ чека
- блокирати новог писца ако неки читалац већ чека
- када читалац који приступа запису заврши приступ, деблокирати једног блокираног писца (ако таквих има)
- када писац заврши приступ, деблокирати све блокиране читаоце (ако таквих има); у супротном деблокирати једног блокираног писца (ако таквих има)

Решење које је претходно дато, већ задовољава последња два захтева.

→ Решити проблем читалаца и писаца тако да решење буде праведно, тј. да у случајевима када и читалац и писац чекају да приступе запису, предност нема увек само један од њих.

## 6

## Недељиво емитовање

Постоји један произвођач и  $N$  потрошача који деле заједнички бафер (*The Atomic Broadcast Problem*). Произвођач убацује производ у бафер и чека док свих  $N$  потрошача не узму исти тај производ. Тада започиње нови циклус производње.

а) Коришћењем семафора написати програм на језику проширенi Pascal који реализује процесе *Producer* и *Consumer* уколико се користи једно елементни бафер.

б) Због повећања конкурентности увести кружни бафер капацитета  $B$  тако да конкурентно процес *Producer* може да ставља новогенерисане производе на крај реда, а процеси *Consumer* да их узима са одговарајућег места у реду. Произвођач убацује производ у бафер и то само у слободне слотове на који чекају свих  $N$  потрошача. Сваки потрошач мора да прими производ у тачно оном редоследу у коме су произведени, мада различити потрошачи могу у исто време да узимају различите производе.

в) Максимизовати конкурентност тако што ће се дозволити истовремени упис у бафер од стране производа, као и истовремено читање из бафера од стране више процеса потрошача (са различитих, узастопних локација).

## Решење

а) Овај проблем је сличан проблему производа и потрошача са том разликом што у овом случају сваки од  $N$  потрошача мора да узме из бафера сваки производ. Овај проблем се може јавити у варијанти када је дат јединични бафер или у варијанти када је дат бафер коначног капацитета ( $B > 1$ ).

Приступ решавању дела који се односи на производа је сличан ономе код проблема производа и потрошача. Решење се састоји из два дела. У првом делу производа чека да се ослободи место на које треба убацити нови производ (*wait(empty)*). Место може бити слободно из два разлога: зато што се ради о првој итерацији или зато што су сви потрошачи узели производ из овог једно елементног бафера. У другом делу производа обавештава сваког потрошача да је припремљен нови производ који они треба да конзумирају (*for index := 1 to N do signal(full[index])*). Сам поступак формирања новог производа није од суштинске важности за решење овог задатка јер не утиче на поступак синхронизације, тако да детаљи имплементације процедуре *make\_new(var item : integer)* нису дати.

Део који се односи на потрошаче се такође састоји из два дела. У првом делу производа чека да се у бафер убаци ново креирани производ (*wait(full[id])*). Овде је употребљен низ семафора да би могло да се тачно специфицира о ком се потрошачу ради. Потрошач је идентификован својим идентификационим бројем *id*. Производ ће елемент убацити у бафер тек када се за њега ослободи место. У другом делу потрошач узима елемент и врши ажурирање бројача којиказује колико је других потрошача до сада преузело тај производ. Уколико се ради о последњем потрошачу, то јест о  $N$ -том онда је потребно обавестити производа да може да убаци нови производ. Пошто постоји више потрошача који могу да промене вредност бројача овај део је потребно извршити недељиво. Ово се постиже увођењем посебног семафора који контролише приступ бројачу (*mutex*). У овој реализацији је узето да потрошач који је прихватио производ као  $N$ -ти по реду сам постави бројач на вредност 0. Могуће је ово реализовати и тако да производ ће на крају своје прве фазе, или на почетку друге постави овај бројач на вредност 0. Процедура која овде није реализована (*consume\_item(item)*) служи потрошачу за даљу обраду добијеног производа. Пошто ова процедура обухвата логику обраде производа, а не само синхронизацију зато није ни убачена.

Треба приметити да је приступ самом производу који се налазу у овом баферу реализован изван критичне секције у којој се врши ажурирање бројача приступа. Није неопходно да се овај део налази унутар критичне секције јер је дозвољено да већи број потрошача конкурентно чита променљиву јер нико не може да је мења док сви потрошачи не прочитају дату променљиву.

Заједничке променљиве које се користе приликом решавања овог проблема су: семафори *mutex*, који дозвољава само једном потрошачу да мења бројач коришћења произведеног производа, и *empty*, који сигнализира да ли је потребно креирати нови производ, низ семафора *full*, који сваком потрошачу сигнализира да ли је формиран нови производ, као и бројач конзумираних производа *num*. Да би систем могао правилно да функционише потребно је свим овим променљивама доделити почетне вредности. Додељивање почетних вредности зависи од логике решавања проблема и зависи од конкретне имплементације појединих делова. У овом случају је логика таква да је на почетку бафер био празан. Ово има за последицу да је потребно на почетку пустити прво произвођача да формира производ који ће касније потрошачу конзумирати. Пуштање произвођача је реализовано постављањем семафорске променљиве *empty* на вредност 1. Почетно чекање свих потрошача на производњу првог производа је реализовано тако што су сви семафори који сигнализирају расположивост производа *full* постављени на вредност 0. Бројач који сигнализира колико је потрошача конзумирало производ мора на почетку бити постављен на вредност 0 јер ни један потрошач није конзумирало ништа. Семафор који омогућава јединствен приступ бројачу приступа *mutex* мора да буде постављен на вредност 1.

```

program AtomicBroadcast;
const N = ...;
var mutex, empty : semaphore;
    full : array [1..N] of semaphore;
    data, num : shared integer;
    index : integer;
procedure Producer;
var item, index : integer;
    procedure make_new(var item : integer); begin ... end;
    procedure put_item(item : integer);
    begin
        data := item;
    end;
begin
    while (true) do
begin
    make_new(item);
    wait(empty);
    put_item(item);
    for index := 1 to N do signal(full[index]);
end;
end;

procedure Consumer(id : integer);
var item : integer;
procedure consume_item(var item : integer); begin ... end;
begin
    while (true) do
begin
    wait(full[id]);
    item := data;
end;
end;

```

```

wait(mutex);
num := num + 1;
if (num = N) then
begin
    num := 0;
    signal(empty);
end;
signal(mutex);
consume_item(item);
end,
end;
begin
    init(mutex, 1);
    init(empty, 1);
    for index := 1 to N do init(full[index], 0);
    num := 0;
    cobegin
        Producer;
        Consumer(1);
        ...
        Consumer(N);
    coend;
end.

```

Ово решење не доводи до изгладњивања јер је произвођач мора да сачека да сваки потрошач конзумира производ који је креиран у претходној итерацији. Код овог решења да би се прешло на следећу итерацију садашња итерација мора комплетно да се заврши, тј сви процеси морају да прођу кроз своју основну петљу.

Решење које је дато се идентично може применити и у случају да постоји већи број производића. Произвођачи не би могли у паралели да се извршавају већ један по један јер је логика проблема таква да када се извршава производића сви остали морају да чекају. Никаква додатна синхронизација није неопходна.

б) Ово је варијанта проблема код који је уместо јединичног бафера дат бафер коначног капацитета ( $B > 1$ ). Синхронизациона логика дата у овом решењу је слична логици када је дат само бафер јединичног капацитета.

Као и у претходном случају решење за производиће се састоји из два дела. У првом делу производиће чека да се ослободи место на које треба убасити нови производ ( $wait(empty)$ ). Приликом смештања производа у бафер потребно је одредити на коју је локацију у бафери могуће сместити овај производ. Пошто се ради о кружном бафери алгоритам по ком се рачуна следећа позиција је  $writeToIndex := (writeToIndex \bmod B) + 1$ . Треба напоменути да у језику проширенi Pascal, за разлику од програмског језика C, индекси полазе од 1 а не од 0. У другом делу производиће обавештава сваког потрошача да је припремљен нови производ који они треба да конзумирају ( $for index := 1 to N do signal(full[index])$ ). Овде се обавештава да је нови производ припремљен, али не и о адреси на коју је тај производ смештен. Овај параметар није неопходан зато што и производиће и потрошачи користе исти алгоритам за рачунање следеће позиције. Поступак самог смештања производа у бафер и рачунања следеће локације за смештање је реализован унутар процедуре  $put\_item(var item : integer)$ .

Део који се односи на потрошаче се такође састоји из два дела. У првом делу потрошач чека док се у бафери не појави производ који није конзумирао ( $wait(full[id])$ ).

Овде је употребљен низ семафора да би могло да се тачно специфицира о ком се потрошачу ради. Потрошач је идентификован својим идентификационим бројем *id*. Овде потрошач не зна колико је производа тренутно у баферу, он само зна да постоји барем један који он није конзумирао. Произвођач је тај производ убацио у бафер тек када се за њега ослободило место, а присуство производа је сигнализирано потрошачу користећи одговарајући семафор. На овај начин је постигнута комуникација између производа и потрошача, без директног обавештавања на којој се локацији налази производ. Израчунавање локације се ради по модулу капацитета бафера. У другом делу потрошач врши ажурирање бројача који казује колико је других потрошача до сада конзумирало производ са исте локације у баферу. Овде сада постоји разлика у односу на јединични бафер код кога су сви читали са исте локације док сада сваки од потрошача може да чита са различите локације. Овакав приступ знатно повећава конкурентност програма јер омогућава да више корисника може у паралели да се извршава. Уколико се ради о последњем потрошачу, то јест о *N*-том који конзумира производ са одговарајуће позиције у баферу, онда је потребно обавестити производа да започне генерисање новог производа. Пошто постоји више потрошача који могу да промене вредност бројача овај део је потребно извршити недељиво. Ово се постиже увођењем посебног семафора који контролише приступ бројачу (*mutex*). Пошто се овде ради о баферу капацитета *B*, више није доволно користити само једну бројачку променљиву *U* овом случају је свакој локацији унутар бафера потребно пријужити по једну бројачку променљиву.

Треба приметити да је, као и у претходној варијанти, приступ самом производу који се налази у овом баферу реализован изван критичне секције у којој се врши ажурирање бројача приступа. Није неопходно да се овај део налази унутар критичне секције јер је дозвољено да већи број потрошача конкурентно чита променљиву јер нико не може да је мења док сви потрошачи не прочитају дату променљиву. Такође треба приметити да приступ индексу са кога неки потрошак чита *readFromIndex[id]* није неопходно реализовати унутар критичне секције јер том индексу може да приступи само један потрошач.

Заједничке променљиве које се користе приликом решавања овог проблема су: семафори *mutex*, који дозвољава само једном потрошачу да мења неки од бројача коришћења произведеног производа, и *empty*, који сигнализира да ли је потребно креирати нови производ, низ семафора *full*, који сваком потрошачу сигнализира да ли је формиран нови производ, бафер капацитета *B* за смештање производа *buffer*, низ бројача конзумираних производа *list*, низ показивача на локацију са које треба прочитати следећи производ (за сваког потрошача по један показивач) *readFromIndex*, као и променљива која специфицира на коју је позицију потребно сместити следећи производ *writeToIndex*. Функционалност програма зависи од правилног избора почетних услова. Приликом писања конкурентних апликација није доволно само реализовати синхронизациону логику већ је потребно и на правилан начин дефинисати услове под којима систем треба да започне свој рад. Неправилно постављени почетни услови могу много да поремете рад програма, чак и да доведу до престанка његовог рада или блокирања. Код овог решења узима се да је на почетку бафер био празан и да до тада ни са које локације није учитано ништа. Ово има за последицу да је потребно на почетку пустити прво производа да формира производ који ће касније потрошачи конзумирати и то тако да може да напуни цео бафер. Овим се омогућава производа да на почетку креира *B* производа. Пуштање производа да попуњава цео бафер почев од локације 1 је реализовано постављањем семафорске променљиве *empty* на вредност *B* и променљиве *writeToIndex* на вредност 1. Да би се обезбедило да сви потрошачи чекају да се креира први производ је реализовано тако што су сви семафори који сигнализирају расположивост производа *full* постављени на вредност 0. Поред овога потребно је специфицирати и са којих локације потрошачи треба да прихватају производе. Исто као што производа произвођач производе смешта од локације 1 тако

и потрошачи читају почев од локације 1. Низ бројача који сигнализира колико је потрошача конзумирало производ мора на почетку бити постављен на вредност 0, тј сваки елемент треба бити постављен на вредност 0 јер није било конзумирања производа пре почетка рада програма. Семафор који омогућава јединствен приступ низу бројача приступа *mutex* мора да буде постављен на вредност 1.

```
program AtomicBroadcastB;
const N = ...;
      B = ...;
var  mutex : semaphore;
      empty : semaphore;
      full : array [1..N] of semaphore;
      buffer : shared array [1..B] of integer;
      num : shared array [1..B] of integer;
      readFromIndex : shared array [1..N] of integer;
      writeToIndex : integer;
      index : integer;

procedure Producer;
var item, index : integer;
    procedure make_new(var item : integer); begin ... end;
    procedure put_item(var item : integer);
begin
    buffer[writeToIndex] := item;
    writeToIndex := (writeToIndex mod B) + 1;
end;

begin
    while (true) do
begin
    make_new(item);
    wait(empty);
    put_item(item);
    for index := 1 to N do signal(full[index]);
end;
end;

procedure Consumer(id : integer);
var item : integer;
    procedure consume_item(var item : integer); begin ... end;
begin
    while (true) do
begin
    wait(full[id]);
    item := buffer[readFromIndex[id]];
    wait(mutex);
    num[readFromIndex[id]] := num[readFromIndex[id]] + 1;
    if (num[readFromIndex[id]] = N) then
begin
        num[readFromIndex[id]] := 0;
        signal(empty);
end;
    signal(mutex);
    readFromIndex[id]:= (readFromIndex[id] mod B) + 1;
end;
end;
```

```

        consume_item(item);
end;
begin
    init(mutex, 1);
    init(empty, B);
    for index := 1 to N do init(full[index], 0);
    for index := 1 to N do readFromIndex[index] := 1;
    writeToIndex := 1;
    for index := 1 to B do num[index] := 0;

    cobegin
        Producer;
        Consumer(1);
        ...
        Consumer(N);
    coend;
end.

```

Ни ово решење, као ни претходно, не доводи до изгладњивања јер произвођач мора да сачека да сваки потрошач конзумира производ са локације на коју је потребно сместити следећи производ пре него што започне следећу итерацију. Ово решење у односу на случај када се ради о бафера јединичног капацитета има већу конкурентност јер омогућава да произвођач као и потрошачи раде у паралели приступајући различитим локацијама унутар бафера.

Уколико се жели направити решење код кога је могуће постојање већег броја произвођача потребно је унети извесне модификације. Ове модификације се односе на синхронизацију између произвођача. Потребно је увести семафорску променљиву на којој бисе синхронизовали произвођачи *mutexProducer*. Приступ овој променљивој би се обављао приликом израчунавања на коју је локацију унутар бафера потребно сместити следећи производ. То би допринело да се процедура за смештање у бафер промени и постане:

```

procedure put_item(var item : integer);
begin
    wait(mutexProducer);
    buffer[writeToIndex] := item;
    writeToIndex := (writeToIndex mod B) + 1;
    signal (mutexProducer);
end;

```

Ово је неопходно урадити јер већи број произвођача може да модификује променљиву која каже где треба сместити следећи податак. Приступ и модификацију је потребно урадити јединствено. Семафорску променљиву је потребно иницијализовати на вредност 1.

в) Конкурентност се може још више повећати уколико се сваком слоту у бафера додели по један *mutex* семафор, уместо једног заједничког.

```

program AtomicBroadcastB;
const N = ...;
      B = ...;

var  mutex : array [1..B] of semaphore;
     empty : semaphore;

```

```
full : array [1..N] of semaphore;
buffer : shared array [1..B] of integer;
num : shared array [1..B] of integer;
readFromIndex : array [1..N] of integer;
writeToIndex : integer;
index : integer;

...
procedure Producer; begin ... end;

procedure Consumer(id : integer);
var item : integer;
begin
    while (true) do
    begin
        wait(full[id]);
        item := buffer[readFromIndex[id]];
        wait(mutex[readFromIndex[id]]);
        num[readFromIndex[id]] := num[readFromIndex[id]] + 1;
        if (num[readFromIndex[id]] = N) then
        begin
            num[readFromIndex[id]] := 0;
            signal(empty);
        end;
        signal(mutex[readFromIndex[id]]);
        readFromIndex[id]:= (readFromIndex[id] mod B) + 1;
        consume_item(item);
    end;
end;
begin
    for index := 1 to B do init(mutex[index], 1);
    init(empty, B);
    for index := 1 to N do init(full[index], 0);
    for index := 1 to N do readFromIndex[index] := 1;
    writeToIndex := 1;
    for index := 1 to B do num[index] := 0;

    cobegin
        Producer;
        Consumer(1);
        ...
        Consumer(N);
    coend;
end.
```

- На који начин се може користећи низ бафера капацитета 1 као готове компоненте проектовати бафер капацитета  $B$ ?

## 7

## Људождери за ручком

Племе људождера једе заједничку вечеру из казана који може да прими  $M$  порција куваних мисионара (*The Dining Savages Problem*). Када људождер пожели да руча онда се он сам послужи из заједничког казана, уколико казан није празан. Уколико је казан празан људождер буди кувара и сачека док кувар не напуни казан. Није дозвољено будити кувара уколико се налази бар мало хране у казану. Користећи семафоре написати програм који симулира понашање људождера и кувара.

## Решење

Овај проблем је донекле сличан проблему произвођача и потрошача по томе што у њему постоје две врсте процеса, једна која производи ресурсе и друга која их конзумира. Разлика између ових проблема је у томе што код произвођача и потрошача ови процеси раде независно једни од других и могу се извршити већи број пута, док код проблема људождера који ручавају постоји условљеност између кувара и људождера. Кувар се позива тачно једном за  $M$  приступа које обаве људождери. Проблеми би били идентични у случају да произвођачи увек производе тачно онолико производа колико може да стане у бафер.

Сваки од два типа процеса који се јављају у решењу овог проблема, кувар и људождери, се састоје из две синхронизационе целине. Када се ради о кувару у првом делу чека да се пробуди људождер који је затекао празан казан (*wait(cook)*). Кувар може да буде пробуђен уколико је неки од људождера дошао до казана и констатовао да се и њену не налази ни једна више порција хране. Пошто се ова провера обавља недељиво, тј само један људождер сме у једном тренутку да приступи казану кувар ће бити пробуђен само једном. Када се кувар пробуди он прелази на поступак припреме јела за гладне људождере. Све време докле год кувар припрема јело ниједан нови људождер не може да започне са јелом. У другом делу, када је комплетно припремио храну, кувар буди људождера који је њега био пробудио *signal(savager)*. На овај начин кувар је пробудио тачно једног људождера, и то оног који је последњи приступио казану, и који не дозвољава другима да му приступе. Поступак кувања јела се налази унутар процедуре *PrepareLunch*.

Људождери такође садрже два синхронизационна дела у себи. У првом делу људождер чека да приступи казану из кога може узети хране (*wait(mutex)*). Семафор на коме се ради синхронизација је бинарни, тако да се омогућава тачно једном људождеру да приступи казану. Приликом приступа казану људождер проверава имали у њему довољно хране. Уколико има узима је, умањује број преосталих порција хране за 1 и одлази, и омогућава другим људождерима да приступе казану *signal(mutex)*. Уколико у казану нема довољно хране прелази се на други део овог процеса, тј на синхронизацију са куваром. На почетку он буди кувара, сигнализирајући му да више нема хране у казану. Буђењем кувара људождер напушта ексклузивно право приступа ресурсу. Након тога прелази у фазу чекања све док кувар не припреми довољан број порција хране. Тек тада он наставља своје извршавање и поново добија ексклузивно право приступа ресурсу. Пошто сад у казану има довољно хране он поставља бројач који каже колико је порција преостало у казану на одговарају вредност. У приложеном решењу је узето да људождер једино може да промени вредност бројача слободних ресурса. Део када људождер конзумира порцију коју је узео из казана налази се изван критичне секције. На овај начин је омогућено да већи број људождера једе у паралели. Људождери једу у паралели, али приступ казану је ексклузиван, то јест казану може да приступа један по један.

Заједничке променљиве које се користе приликом решавања овог проблема су: семафори *mutex*, који дозвољава само једном људождеру да приступи казану и

промени број порција хране у казану, *cook*, који сигнализира кувару да је потребно припреми нови ручак, и семафор *savager*, којим кувар обавештава људожђера који га је пробудио да је храна припремљена, као и променљива *servings* која специфицира број порција хране преостале у казану. Почетни услови неопходни за рад се тако одабирају да на почетку први људожђер када дође пробуди кувара јер ће затећи празан казан. Друга ствар коју треба обезбедити је да кувар и људожђери не могу истовремено да приступају казану. Кувар ће почети да ради тек када га први људожђер буде пробудио. Да би се обезбедило да само један људожђер приступа казану у једном тренутку семафор који сигнализирају расположивост казана *mutex* постављен на вредност 1. Поред овога потребно је специфицирати да је на почетку казан био празан постављањем бројача преосталих порција хране у казану на вредност 0. Да би се омогућило да кувар и људожђери не могу да приступају казану истовремену семафор *cook* се поставља на вредност 0. Да бу се успоставила повратна веза између кувара и људожђера који га је пробудио семафор *savager* се поставља на вредност 0. Наведени семафори су расподељени бинарни семафори и максимално један од њих може да има вредност 1.

```
program DiningSavages;
const M =...;
var cook : semaphore;
    savager : semaphore;
    mutex : semaphore;
    servings : shared integer;

procedure PrepareLunch; begin ... end;
procedure GetServingFromPot; begin ... end;
procedure eat; begin ... end;

procedure SavageCook;
begin
    while (true) do
    begin
        wait (cook);
        PrepareLunch;
        signal (savager)
    end;
end;

procedure Savage(i : integer);
begin
    while (true) do
    begin
        wait(mutex);
        if (servings = 0) then
        begin
            signal (cook);
            wait (savager);
            servings := M;
        end;
        servings := servings - 1;
        GetServingFromPot ;
        signal (mutex);
        eat;
    end;
end;
```

```

end;

begin
    servings := 0;
    init(cook, 0);
    init(savager, 0);
    init(mutex, 1);
    cobegin
        SavageCook;
        Savage(1);
        Savage(2);
    coend;
end.

```

Постоји и алтернативно решење код кога је могуће да кувар модификује бројач преосталих порција. Овакво решење се заснива на коришћењу својства расподељених бинарних семафора и технике предаје „штафетне палице”. Потребно је преместити доделу вредности  $servings := M$ ; из кода за лјудожђера у код за кувара. Ово не би утицало на коректност решења јер лјудожђер у тренутку буђења кувара напушта ексклузивно право коришћења ресурса, препушта то право пробуженом кувару и чека да му кувар, када заврши припрему јела, врати ексклузивно право над ресурсом. У том случају део кода кувара би гласио:

```

procedure SavageCook;
begin
    while (true) do
        begin
            wait (cook);
            PrepareLunch;
            servings := M;
            signal (savager)
        end;
end;

```

Што се синхронизације тиче повећање броја кувара не би утицало на изглед решења, то јест исто решење би се могло користити и у случају већег броја кувара.

## 8

## Медвед и пчеле

Постоји  $N$  пчела и један гладан медвед (*The Bear and the Honeybees*). Они користе заједничку кошницу. Кошница је иницијално празна, а може да прими  $H$  напрстака меда. Медвед спава док се кошница не напуни медом, када се напуни медом мада поједе сав мед након чега се враћа на спавање. Пчелице непрестано лете од цвета до цвета и сакупљају мед. Када прикупе један напрстак долазе и стављају га у кошницу. Она пчела која је попунила кошницу буди медведа. Користећи семафоре решити проблем маде и пчела.

## Решење

Овај проблем је доста сличан проблему лъудождера за ручком. Од њега разликује по томе што пчела која пробуди медведа треба да настави са сакупљањем меда, док лъудождер који пробуди кувара мора да чека јер не постоји храна коју може јести, тј активност коју може обављати у паралели. И у овом случају се могу користити расподељени бинарни семафори помоћу којих је обезбеђено са у једном тренутку или нека пчела или медвед могу да приступе ресурсу. Пошто је синхронизација мало другачија мора се водити рачуна о броју потребних семафора и правилном постављању почетних услова.

```
program BearAndHoneybees;
const N = ...;
      H = ...;
var  hive : semaphore;
      full : semaphore;
      pot : shared integer;

procedure Bear;
procedure sleep; begin ... end;
procedure eat; begin ... end;
begin
  while (true) do
    begin
      sleep;
      wait(full);
      eat;
      pot := 0;
      signal(hive);
    end;
end;

procedure Honeybee (id : integer);
procedure collect; begin ... end;
begin
  while (true) do
    begin
      collect;
      wait(hive);
      pot := pot + 1;
      if (pot = H) then signal(full)
```

```
    else signal(hive);
end;
end;

begin
pot := 0;
init(hive, 1);
init(full, 0);
cobegin
    Bear;
    Honeybee(1);
    Honeybee(2);
    ...
coend;
end.
```

## 9

### Одгајање птића

У гнезду живи  $n$  птића и две родитељске птице (*The Hungry Birds Problem*). Птићи једу из заједничке посуде која прима  $F$  црвића. Сваки птић непрекидно једе из посуде по једног црвића, мало спава и поново се враћа да једе. Када посуда постане празна, птић који је испразнио посуду буди једног од родитеља. Родитељска птица може да крене у лов на црвиће само уколико у гнезду остане други родитељ. Из лова се родитељ враћа тек када накупи  $F$  црвића које сипа у посуду. Родитељске птице осим тога што чувају птиће и иду у лов могу да напусте гнездо да би саме јеле, тако да у гнезду увек остане један родитељ. Користећи семафоре написати програм који симулира понашање птића и родитеља.

## Решење

Проблем одгајања птића је у својој основној варијанти у којој постоји само једна родитељска птица еквивалентан проблему Медведа и пчела. У варијанти која је дата у овом задатку, са две родитељске птице у гнезду, постоји разлика између ових проблема. Разлика се састоји у томе што постоје два родитеља који могу да хране птиће, али тако да увек један од њих мора да остане у гнезду. Ово је разлика која се односи само на родитељску страну и захтева само додатну синхронизацију између родитеља.

Сваки од два типа процеса који се јављају у решењу овог проблема, птићи и родитељи, се састоје из два синхронизационих дела.

Процедура која описује птиће има два синхронизационна дела у којима се тражи приступ храни и шаљу обавештења. У првом делу птић чека да приступи заједничкој посуди за храну из које може узети хране (*wait(babyMutex)*). Семафор на коме се ради синхронизација је бинарни, тако да се омогућава тачно једном птићу да приступи посуди. Овде треба нагласити да уколико је птић приступио храни, прошао је *wait(babyMutex)* то значи да хране има и не треба да од родитеља чека потврду да је храна донета као што је лъудождер морао да чека кувара. У другом делу након узимања хране, *put := put - 1*, птић проверава да ли је остало још хране за следећег птића или је потребно да један од родитеља крене у лов. Уколико има још хране дозвољава следећем птићу да јој приступи користећи семафор *babyMutex* (*signal(babyMutex)*). Уколико је узео последњег црвића обавештава неког од родитеља да треба да крене у лов користећи семафор *parent* (*signal(parent)*). Треба напоменути да птић овом приликом не чека одговор од родитеља нити следећем птићу дозвољава приступ храни јер хране нема. Дозволу приступа храни у овом случају може вратити само родитељ који је отишao у лов када се из лова врати.

Када се ради о родитељским птицама за њих су креиране две процедуре. Једна која води рачуна о птићима и друга која омогућава родитељској птици да сама оде и једе. Треба напоменути да родитељска птица у једном тренутку сме да извршава само једну од те две процедуре. У супротном може да се деси да иста птица буде и у гнезду и у потрази за храном за себе. Прва процедура *eat()* омогућава родитељској птици да изађе из гнезда и крене у потрагу за сопственом храном. Као услов изласка стоји правило које каже да један од родитеља мора увек да остане у гнезду. Родитељ може изаћи из гнезда из два разлога: зато што је гладан и лови храну за себи и други разлог је тај што је отишao у лов да накупи храну птићима. Обезбеђивање услова да један родитељ треба увек да остане у гнезду се постиже тако што се пре сваког изласка родитеља из гнезда проверава семафор *toLeave* који је иницијално постављен на вредност за један мању од броја родитеља. Поступак сваког изласка и уласка у гнездо

је реализован користећи следећи пар приступа семафору: *wait(toLeave)* и *signal(toLeave)*. Уколико се ради о процедуре у којој се родитељ брину о птићима онда је прва ствар о којој родитељ брине да прибави довољно хране за гладне птиће. Родитељ се припрема да крене у набавку хране уколико је неки од птића покупио последњег црвића. Птић родитељу то сигнализира помоћу семафора *parent*, односно родитељ чека на датом семафору (*wait(parent)*). Овде се може десити ситуација да било који родитељ који се налази у том тренутку у гнезду крене у лов. Овде није битно ко ће да одради доношење хране битно је да постоји више од једног који може да ово оствари. Када је родитељ добио захтев за храном чека дозволу да напусти гнездо. Дозволу ће добити или уколико је други родитељ у гнезду и мотри на птиће или када се тај родитељ врати из лова. Ово је реализовано приступом семафору *toLeave*. Када се родитељ врати из лова родитељ убације црвиће и поставља променљиву *num*, која означава колико има хране на вредност *F*. Овде не треба бринути да ли неко други може да јој приступи. У једном тренутку овој променљивој може да приступи или птић или родитељ. Пошто су сви птићи који желе да приступе храни тренутно блокирани на семафору *babyMutex*, а само један родитељ доноси храну, не може се десити вишеструки приступ датој променљивој. Када је родитељ поставио количину хране на одговарају вредност даје дозволу птићима да приступе храни користећи семафор *babyMutex* (*signal(babyMutex)*).

Заједничке променљиве које се користе приликом решавања овог проблема су: семафор *toLeave*, који дозвољава само једном родитељу да напусти гнездо, *parent*, којим птић сигнализира родитељу да је потребно уловити залиху црвића, и семафор *babyMutex*, којим се дозвољава само једном птићу приступ храни у једном тренутку, као и променљива *num* која специфицира број преосталих црвића. Почетни услови неопходни за рад се тако одабирају да на почетку родитељ мора да крене у потрагу за храном јер је на почетку нема, а први птић мора да затекне барем једну порцију. Друга ствар коју треба обезбедити је да увек један родитељ мора да остане у гнезду, док други може да иде да се храни. Систем је тако направљен да ће родитељ на почетку радити и ако га ни један птић није пробудио. Да би се обезбедило да само један птић може да приступи хранам, али тек када је родитељ убаци *babyMutex* постављен је на вредност 0. Поред овога потребно је специфицирати да на почетку није било хране постављањем бројача преосталих црвића на вредност 0. Да би се омогућило да родитељ на почетку самоиницијативно крене по храну семафор *parent* се поставља на вредност 1. Да би се ограничило родитељима да напуштају гнездо семафор *toLeave* се поставља на вредност 1 (за један мање од броја родитеља).

```

program HungryBirds;
const F = ...;
      N = ...;
var num : shared integer;
    babyMutex : semaphore;
    parent : semaphore;
    toLeave : semaphore;

procedure babyBird (id : integer);
    procedure eatWorm; begin ... end;
begin
    while(true) do
        begin
            wait(babyMutex);
            num := num - 1;
            if(num <= 0) then signal(parent)
            else signal(babyMutex);
            eatWorm;
        end;
end;

```

```
    end;
end;

procedure parentBird (id : integer);
    procedure eating; begin ... end;
    procedure hunt; begin ... end;

procedure eat;
begin
    wait(toLeave);
    eating;
    signal(toLeave);

end;
procedure guard;
begin
    wait(parent);
    wait(toLeave);
    hunt;
    num := F;
    signal(toLeave);
    signal(babyMutex);

end;
begin
    while(true) do
begin
    ...
    guard;
    ...
    eat;
end;
end;

begin
    num := 0;
    init(babyMutex, 0);
    init(parent, 1);
    init(toLeave, 1);
    cobegin
    ...
    coend;
end.
```

**10****Брига о деци**

У неком забавишту постоји правило које каже да се на свака три детета мора наћи барем једна васпитачица (*The Child Care Problem*). Родитељ у забавиште доводи и одводи једно или више деце. Васпитачица брине о деци, долази и одлази са посла, али сме да напусти забавиште само уколико то не нарушава наведено правило.

- Написати процедуре, користећи семафоре, за родитеље који доводе и одводе децу и васпитачице и иницијализовати почетне услове. Родитељ оставља децу у забавишту само уколико има места, уколико нема места одводи их.
- Решити овај проблем уколико родитељ чека да остави децу све док не буде било довољно места да их све остави

**Решење**

а) Код овог проблема уочавају се две групе послова које треба одрађивати, прва су родитељи који доводе и одводе децу из забавишта и другу групу представљају васпитачице које се старају о деци, али које када им истекне радно време одлазе кући уколико су родитељи покупили своју децу. Треба приметити да деца овде не представљају групу за себе, она представљају само објекте.

Понашање родитеља је описано користећи две акције, једну за довођење деце у обданиште, и друге за њихово одвођење. Прва акција омогућава да се доведу деца до забавишта, и у њему оставе уколико има довољно слободног места, уколико нема довољно слободног места родитељ одводи своју децу. Пошто родитељ треба да добије повратну информацију да ли је сместио свако од своје *num* деце ово мора да буде реализовано као функција *function bringUpChildren (num : integer) : boolean*. Пошто ова функција приступа интерним подацима забавишта (број васпитачица и број деце) и може да их промени потребно је забранити свим осталим процесима да их промене. Да би се остварила недељивост приликом приступа уводи се ексклузивно право приступа интерним параметрима забавишта користећи бинарни семафор *mutex*. Уколико је број расположивих васпитачица (*numNanny*) такав да је могуће примити сву децу коју доводи родитељ (*numChild + num*) то јест уколико важи  $(numChild + num) \leq C - numNanny$  функција треба да врати *true* у супротном *false*. Параметар С представља константу која означава на колико деце долази васпитачица. Овде треба водити рачуна да се ослободи бинарни семафор пре одласка, без обзира да ли је операција смештања успешно прошла или не, да се не би десило да се сви блокирају приступајући овом семафору. Одвођење деце из забавишта је комплексија операција у односу на довођење деце јер поред тога што треба вратити децу потребно је дозволити и свим васпитачицама које су пожеле да напусте забавиште, а то нису могле јер је било превише деце које су чувале, да га напусте. Процедура за одвођење деце има аргумент број деце колико је потребно да напусте забавиште (*bringBackChildren (num : integer)*). Ова акција се увек може остварити зато није неопходно враћати никакав резултат. Пошто је приликом одласка потребно баратати интерним стањем забавишта на почетку ове процедуре се тражи дозвола за приступ тим интерним параметрима стања (*waii(mutex)*). Поступак одвођења деце се састоји из неколико фаза. У првој фази се умањи број деце који се налазе у забавишту за онолико деце колико је доведено, након тога се рачуна број васпитачица које могу да напусте забавиште уколико желе (*out := numNanny - (numChild + C-1) / C*). То је број васпитачица умањен за број васпитачица које морају да остану са децом. Када се утврди колико васпитачица може да напусти проверава се колико васпитачица стварно жели да напусти забавиште (променљива *numWaiting*). Уколико је број оних које желе да напусте мањи од броја колико сме да га напусте онда се пуштају да оду само оне

које то желе. Када је утврђен број оних васпитачица колико може да оде са сваком од њих се то договора. Договор се постиже на следећи начин: *signal(toLeave)* и *wait(confirm)*. Први семафор обавештава неку блокирану васпитачицу да сме да оде, а другим семафором васпитачица обавештава родитеља да је отишла. Поступак обавештавања васпитачица је неопходно остварити јер би се у супротном могло десити да све васпитачице буду блокиране у жељи да напусте обданиште, али не постоји нико ко би могао о томе да их обавести.

Понашање васпитачица је такође описано користећи две процедуре, једне којом васпитачица долази на посао и другом којом га напушта. Када васпитачица долази на посао приступа интерним променљивама па мора да тражи дозволу приступом семафору *mutex* (*wait(mutex)*). Када приступи променљивама увећава број васпитачица за 1. Након тога проверава да ли постоји барем једна васпитачица које жели да напусти забавиште (*numWaiting > 0*). Уколико постоји барем једна која жели да напусти забавиште, једно дозвољава да напусти забавиште, на овај начин број васпитачица у забавишту није промењен. Треба констатовати да се број васпитачица (*numNanny*) неће променити јер васпитачица у тренутку напуштања забавишта умањује ову променљиву за 1. Такође треба приметити да овде неће доћи до неконзистентности података јер једини процес који има право да приступа подацима унутар забавишта је блокиран па васпитачица која одлази може да мења дате податке. Комуникација између васпитачица се постиже на идентичан начин како и између родитеља и васпитачица користећи: *signal(toLeave)* и *wait(confirm)*. Другачије није ни могуће урадити јер васпитачица која одлази не зна ко ју је пустио да оде, родитељ који одводи децу или нова васпитачица која долази. На крају процедуре за улазак васпитачица која улази враћа право приступа *signal(mutex)*. Приликом напуштања забавишта (*nannyExit()*) васпитачица тражи приступ променљивама па мора да тражи дозволу приступом семафору *mutex* (*wait(mutex)*). Када добије право приступа проверава да ли је испуњен услов за одлазак (*(numChild + num) <= C \* (numNanny - 1)*), уколико је услов испуњен васпитачица умањује број васпитачица које се тренутно налазе у забавишту враћа право приступа променљивама и одлази. Уколико јој није дозвољено да напусти забавиште онда увећава број васпитачица које чекају да изађу за 1 и враћа право приступа (*signal(mutex)*) да би могао да се појави неко ко ће јој дозволити да оде. Тада неко може да буде родитељ који води децу или нова васпитачица која долази. Од тог неког се очекује дозвола изласка (*wait(toLeave)*), када се добије дозвола васпитачица умањује број васпитачица које се налазе у забавишту за 1 (*numNanny := numNanny - 1*), али такође умањује за 1 и број васпитачица које чекају да изађу (*numWaiting := numWaiting - 1*). Пошто су ово интерне променљиве било би очекивано да се тражи дозвола за приступ њима. Ова дозвола је добијена имплицитно, тако што се особа која је обавестила васпитачицу да сме да напусти забавиште сама блокирала на семафору *confirm* чекајући да васпитачица која одлази промени интерно стање.

Заједничке променљиве које се користе приликом решавања овог проблема су: семафор *mutex*, који дозвољава само једном родитељу или васпитачици да приступа интерним променљивама, семафор *toLeave*, којим се сигнализира васпитачици која је пожелела да напусти забавиште да то стварно и уради, и семафор *confirm*, којим васпитачица која одлази сигнализира да је завршила приступ интерним параметрима, као и целобројне променљиве *numChild*, *numNanny* и *numWaiting* које специфицирају број деце у забавишту, број васпитачица у забавишту и број васпитачица у забавишту које желе да га напусте, респективно. Почетни услови неопходни за рад се тако одабирају да на почетку у забавишту нема ни деце ни васпитачица. Број деце и васпитачица се мења током времена тако да је потребно једино дозволити приступ интерним променљивама. Да би се обезбедио приступ променљивама *mutex* постављен је на вредност 1, док су семафори *confirm* и *toLeave* постављени на вредност 0. Број деце, васпитачица, било да се налазе у забавишту или желе да напусти забавиште се поставља на вредност 0.

```

program ChildCare;
const C = 3;

var numChild : shared integer;
    numNanny : shared integer;
    numWaiting : shared integer;
    mutex : semaphore;
    confirm : semaphore;
    toLeave : semaphore;
function bringUpChildren (num : integer) : boolean;
begin
    wait(mutex);
    if((numChild + num) <= C * numNanny) then
    begin
        numChild := numChild + num;
        bringUpChildren := true;
    end
    else
        bringUpChildren := false;
    signal(mutex);
end;

procedure bringBackChildren (num : integer);
var out, i : integer;
begin
    wait(mutex);
    numChild := numChild - num;
    out := numNanny - (numChild + C - 1) div C;
    if(out > numWaiting) then out := numWaiting;
    for i := 1 to out do
    begin
        signal(toLeave);
        wait(confirm);
    end;
    signal(mutex);
end;

procedure nannyEnter;
begin
    wait(mutex);
    numNanny := numNanny + 1;
    if(numWaiting > 0) then
    begin
        signal(toLeave);
        wait(confirm);
    end;
    signal(mutex);
end;

procedure nannyExit;
begin
    wait(mutex);
    if (numChild <= C * (numNanny - 1)) then
    begin

```

```

numNanny := numNanny - 1;
signal(mutex);
end
else
begin
    numWaiting := numWaiting + 1;
    signal(mutex);
    wait(toLeave);
    numNanny := numNanny - 1;
    numWaiting := numWaiting - 1;
    signal(confirm);
end
end;

begin
    numChild := 0;
    numNanny := 0;
    numWaiting := 0;
    init(mutex, 1);
    init(confirm, 0);
    init(toLeave, 0);
    cobegin
    ...
    coend;
end.

```

Треба констатовати да повећање конкурентности дозвољавањем већем броју васпитачица да напусте забавиште унутар методе *bringBackChildren* :

```

for i := 1 to out do
    signal(toLeave);
for i := 1 to out do
    wait(confirm);

```

који наизглед повећава конкурентност могао да доведе до неконзистентног стања јер би већи број васпитачица приступао променљивама *numWaiting* и *numNanny* и мењао их.

Уколико није потребно да се сачека потврда од блокиране васпитачице да је отишла, решење се може додатно убрзати. Ово убрзање би се постигло тако што би процес који буди блокирану васпитачицу прво сам променио стање у забавишту а после само обавестио васпитачицу да је све завршено тако да не би морао да чека потврду од блокиране васпитачице да је промена обављена. На овај начин семафор *confirm* постаје сувишан.

```

procedure bringBackChildren (num : integer);
var out, i : integer;
begin
    wait(mutex);
    numChild := numChild - num;
    out := numNanny - (numChild + C - 1) div C;
    if(out > numWaiting) then out := numWaiting;
    for i := 1 to out do
    begin
        numNanny := numNanny - 1;
        numWaiting := numWaiting - 1;
    end;
end;

```

```

        signal(toLeave);
    end;
    signal(mutex);
end;

procedure nannyEnter;
begin
    wait(mutex);
    numNanny := numNanny + 1;
    if(numWaiting > 0) then
    begin
        numNanny := numNanny - 1;
        numWaiting := numWaiting - 1;
        signal(toLeave);
    end;
    signal(mutex);
end;

procedure nannyExit;
begin
    wait(mutex);
    if (numChild <= C * (numNanny - 1)) then
    begin
        numNanny := numNanny - 1;
        signal(mutex);
    end
    else
    begin
        numWaiting := numWaiting + 1;
        signal(mutex);
        wait(toLeave);
    end
end;

```

б) Уколико је потребно да родитељ остави сву децу претходно решење је могуће модификовати увођењем улазног реда *entry*. На овом реду, семафору, чекају родитељи који доводе децу и васпитачице које одлазе са посла, док се приликом одвођења деце и одласка васпитачице овом реду не приступа. Уколико родитељ приликом довођења деце закључи да нема довољно места блокира се на семафору *toLeave* док се не појави слободно место или дође нова васпитачица. Пошто блокирани родитељ не враћа улазни семафор овде се поред тог родитеља који доводи децу блокира било који родитељ који доводи своју децу али и било која васпитачица која жели да оде. Исто тако, уколико васпитачица приликом одласка са посла закључи да нема услова за одлазак блокира се на семафору *toLeave* док се не појави друга васпитачица или неки родитељи не одведу децу. Пошто ни васпитачица не враћа улазни семафор онда се блокира било који која васпитачица која жели да оде али и било родитељ који доводи своју децу.

```

program ChildCare;
const C = 3;

var  numChild : shared integer;
     numNanny : shared integer;

```

```
numWaitingC: shared integer;
numWaitingN : shared integer;
mutex : semaphore;
entry, toLeave : semaphore;

procedure bringUpChildren (num : integer);
begin
    wait(entry);
    wait(mutex);
    if((numChild + num) > C * numNanny) then
    begin
        numWaitingC := num;
        signal(mutex);
        wait(toLeave);
        numWaitingC := 0;
    end;
    numChild := numChild + num;
    signal(mutex);
    signal(entry);
end;

procedure bringBackChildren (num : integer);
begin
    wait(mutex);
    numChild := numChild - num;
    if((numWaitingC > 0)
       and ((numChild + numWaitingC) <= C * numNanny)) then
        signal(toLeave);
    else if((numWaitingN > 0)
            and (numChild <= C * (numNanny-1))) then
        signal(toLeave);
    else
        signal(mutex);
end;

procedure nannyEnter;
begin
    wait(mutex);
    numNanny := numNanny + 1;
    if((numWaitingC > 0)
       and ((numChild + numWaitingC) <= C * numNanny)) then
        signal(toLeave);
    else if((numWaitingN > 0)
            and (numChild <= C * (numNanny-1))) then
        signal(toLeave);
    else
        signal(mutex);
end;

procedure nannyExit;
begin
    wait(entry);
    wait(mutex);
    if (numChild > C * (numNanny - 1)) then
    begin
```

```
    numWaitingN := numWaitingN + 1;
    signal(mutex);
    wait(toLeave);
    numWaitingN := numWaitingN - 1;
end;
numNanny := numNanny - 1;
signal(mutex);
signal(entry);
end;

begin
    numChild := 0;
    numNanny := 0;
    numWaitingC := 0;
    numWaitingN := 0;
    init(mutex, 1);
    init(entry, 1);
    init(toLeave, 0);
    cobegin
    ...
    coend;
end.
```

## 11

## Синхронизација на баријери

Разматра се проблем синхронизације на баријери (*Barrier Synchronization*) без процеса координатора. Баријера омогућава процесима да на њој сачекају док тачно  $N$  процеса не достигне одређену тачку у извршавању пре него што било који од тих процеса не настави са својим извршавање. Користећи семафоре решити овај проблем. Омогућити да се иста баријера може користити већи број пута.

### Решење

Решење проблема синхронизације на баријери се заснива на техници коришћења „преводнице“ (*lock*) која има двоја врата, улазна врата и излазна врата. Прва, улазна, врата су отворена док тачно  $N$  процеса не прође кроз њих и уђу у тампон зону. Када се накупи тачно  $N$  процеса прва врата се затварају и отварају се друга, излазна, врата. Када свих  $N$  процеса прође и кроз друга врата она се затварају и поново се отварају прва врата. Како би се реализовала ова техника потребна су два семафора (двоја врата) који су расподељени бинарни семафор и користи се техника предаје штафетне палице.

```
program Barrier;
const N = ...;
var door1, door2 : semaphore;
var cnt : shared integer;

procedure Passenger(id : integer);
begin
    wait(door1);
    cnt := cnt + 1;
    if(cnt = N) then
        signal(door2)
    else
        signal(door1);
    ...
    wait(door2);
    cnt := cnt - 1;
    if(cnt = 0) then
        signal(door1)
    else
        signal(door2);
    ...
end;
begin
    init(door1, 1);
    init(door2, 0);
    cnt := 0;
    cobegin
        Passenger(0);
        Passenger(1);
        ...
    coend
end.
```

## 12

## Вожња тобоганом

Претпоставити да постоји  $N$  путника и једно возило на тобогану (*The Roller Coaster Problem*). Путници се наизменично шетају по луна парку и возе на тобогану. Тобоган може да прими највише  $K$  путника при чему је  $K < N$ . Вожња тобоганом може да почне само уколико се сакупило тачно  $K$  путника. Написати програм користећи семафоре који симулира описани систем.

## Решење

Проблем вожње тобоганом описује системе код којих је потребно да већи број процеса достави податке једном процесу, серверу, као би он могао да обави обраду свих примљених података и да сваком од процеса врати одговор. Процеси који су упутили захтев чекају док не добију одговор од сервера како би кренули даље са својим радом. Овај проблем доста личи на проблем синхронизације на баријери са процесом координатором.

## Прво решење

Циклус започиње тако што возило дозвољава да тачно  $K$  посетиоца уђе у возила, *signal (start)*. Након тога возило чека да тачно  $K$  путника и уђе у возило, *wait (allAboard)*. Када путник добије дозволу да може да уђе у возило, *wait (start)*, недељиво приступа променљивој у којој је сачуван број путника, *passengers*. Увећава ову променљиву и ако је последњи путник који треба да уђе обавештава возило да су сви путници стигли, *signal (allAboard)*, у сваком случају напушта ексклузивно право приступа. Циклус се завршава на сличан начин тако што возило дозвољава да тачно  $K$  посетиоца изађе из возила, *signal (end)*. Након тога возило чека да тачно  $K$  путника и изађе из возила, *wait (allOut)*. Када путник добије дозволу да може да изађе из возила, *wait (end)*, недељиво приступа променљивој у којој је сачуван број путника, *passengers*. Умањује ову променљиву и ако је последњи путник обавештава возило, *signal (allOut)*, свакако напушта ексклузивно право приступа. Овим се циклус завршава.

```
program RollerCoaster;
const K = ...;
N = ...;

var start : semaphore;
allAboard : semaphore;
stop : semaphore;
allOut : semaphore;
mutex : semaphore;
passengers: shared integer;

procedure Passenger(id : integer);
procedure walking (id : integer); begin ... end;
procedure riding (id : integer); begin ... end;

procedure boardCar (id : integer);
begin
    wait (start);
    wait (mutex);
    passengers := passengers + 1;
    if (passengers = K) then

```

```
begin
    passengers := 0;
    signal (allAboard);
end;
signal (mutex);
end;
procedure leaveCar (id : integer);
begin
    wait (stop);
    wait (mutex);
    passengers := passengers + 1;
    if (passengers = K) then
begin
    passengers := 0;
    signal (allOut);
end;
signal (mutex);

end;

begin
    while true do
begin
    walking(id);
    boardCar (id);
    riding(id);
    leaveCar (id);
end;
end;

procedure Coaster;
procedure riding; begin ... end;

procedure boardingCar;
var i : integer;
begin
    for i := 1 to K do signal (start);
    wait(allAboard);
end;

procedure leavingCar;
var i : integer;
begin
    for i := 1 to K do signal (stop);
    wait(allOut);
end;

begin
    while true do
begin
    boardingCar;
    riding;
    leavingCar;
end;
end;
```

```

begin

    init(start, 0);
    init(allAboard, 0);
    init(stop, 0);
    init(allOut, 0);
    passengers := 0;
    init(mutex, 1);
    cobegin
        Passenger(1);
        ...
        Passenger(N);
        Coaster;
    coend;
end.

```

Треба приметити да су неопходна два семафора *start* и *end* помоћу којих возило обавештава посетиоце, они који желе да се укрцају и они који желе да се искрцају, јер на њима чекају различити процеси ( $K < N$ ). Такође треба приметити да је довољан само један семафор преко кога путници обавештавају возило да је операција завршена.

У случају да није потребно да сервер процесима враћа одговор него је само довољно да сервер обавести процесе да је обрада завршена онда се дато решење може упростићи. Ово упрошћавање се постиже на тај начин што сервер, возило, само прослеђује процесима, посетиоцима, информацију да је операција завршена, а од њих не чека са сви приме ту информацију и да пошаљу потврде да су примили, већ сервер наставља са новим циклусом. Пошто више није неопходно да путници обавештавају возило семафор *allOut* је постао сувишан.

```

program RollerCoaster;
const K = ...;
N = ...;

var start : semaphore;
    allAboard : semaphore;
    stop : semaphore;
    mutex : semaphore;
    passengers: shared integer;

procedure Passenger(id : integer);
...
procedure leaveCar (id : integer);
begin
    wait (stop);
end;

procedure Coaster;
...
procedure leavingCar;
var i : integer;
begin
    for i := 1 to K do signal (stop);
end;

```

```
begin
    init(start, 0);
    init(allAboard, 0);
    init(stop, 0);
    passengers := 0;
    init(mutex, 1);
    cobegin
        Passenger(1);
        ...
        Passenger(N);
        Coaster;
    coend;
end.
```

### Друго решење

Овај проблем се може решити и на начин који примењен код синхронизације на баријери коришћењем више врата. У првој фази, улазак, возило отвара улазна врата за возила, *signal (start)*, након тога чека да накупи довољан број путника који ће му отворити следећа врата, *wait (allAboard)*. Када нађе посетилац он пролази кроз улазна врата, *wait (start)*. Уколико је последњи који треба да стигне онда отвара врата возилу, *signal (allAboard)*, уколико није последњи оставља улазна врата отвореним, *signal (start)*. У другој фази, излазак, возило отвара једна излазна врата за возила, *signal (end)*, након тога чека да накупи довољан број путника и отворе му се врата, *wait (allOut)*. Посетилац пролази кроз излазна врата, *wait (end)*. Уколико је последњи који треба да изађе отвара врата возилу, *signal (allOut)*, уколико није последњи оставља излазна врата отвореним, *signal (end)*. На почетку су сви семафори постављени на постну вредност 0, као и број посетилаца.

```
program RollerCoaster;
...
var  start : semaphore;
      allAboard : semaphore;
      stop : semaphore;
      allOut : semaphore;
      passengers: shared integer;

procedure Passenger(id : integer);
...
procedure boardCar (id : integer);
begin
    wait (start);
    passengers := passengers + 1;
    if (passengers = K) then
    begin
        passengers := 0;
        signal (allAboard);
    end
    else
        signal (start);
end;
procedure leaveCar (id : integer);
begin
```

```
wait (stop);
passengers := passengers + 1;
if (passengers = K) then
begin
    passengers := 0;
    signal (allOut);
end;
    signal (stop);
end;
...
procedure Coaster;

procedure boardingCar;
begin
    signal (start);
    wait (allAboard);
end;

procedure leavingCar;
begin
    signal (stop);
    wait(allOut);
end;
...
begin
    init(start, 0);
    init(allAboard, 0);
    init(stop, 0);
    init(allOut, 0);
    passengers := 0;
    cobegin
        ...
    coend;
end.
```

## 13

### Изградња молекула воде

Постоје два типа атома, водоник и кисеоник, који долазе до баријере (*The H<sub>2</sub>O Problem*). Да би се формирао молекул воде потребно је да се на баријери у истом тренутку нађу два атома водоника и један атом кисеоника. Уколико атом кисеоника дође до баријере на којој не чекају два атома водоника онда он чека да се они сакупе. Уколико атом водоника дође до баријере на којој се не налазе један кисеоник и један водоник он чека на њих. Баријеру треба да напусте два атома водоника и један атом кисеоника. Користећи семафоре написати програм који симулира понашање водоника и кисеоника.

## Решење

У овом проблему се ради о синхронизацији на баријери за два типа процеса, атомима водоника и атомима кисеоника. Код овог проблема је потребно обезбедити да тачно одређен број различитих процеса међусобно интерагује. У систему постоји велики број атома кисеоника и водоника који могу да интерагују или интеракција мора да се обави између тачно два атома водоника и једног атома кисеоника, и само у том случају може да настане молекул воде.

Овај проблем се може посматрати из две перспективе. Прва перспектива се односи на то да ли је потребно да сваки процес сазна идентификаторе свих атома у молекулу или је доволно да то сазна само један процес. Друга перспектива се односи на то ко започиње интеракцију, односно да ли је прво потребно да се појави атом водоника или атом кисеоника да би поступак започео.

#### Прво решење

Решење је дато за случај да је доволно да само атом кисеоника зна идентификаторе атома водоника са којима чини молекул воде, и неопходно је да се прво појави атом кисеоника да би интеракција започела.

Синхронизација се састоји из неколико фаза. Када се ради о атомима кисеоника, који са још два водоника формира молекул воде, потребно је обезбедити да се на баријери појави тачно један атом кисеоника и тачно два атома водоника. Да би се спречило мешање атома кисеоника и да би се тачно могло одредити који се атом кисеоника везује са којим атомима водоника атом кисеоника чека да самостално дође до баријере (*wait (oxyMutex)*). Да би се обезбедило да два атома водоника дођу до баријере атом кисеоника две пута позива *signal (hydroSem)*. Прелази се на следећу фазу у којој атом кисеоника чека да се појаве тачно два атома водоника. Појава два атома водоника на баријери се сигнализира семафором *oxySem (wait (oxySem))*. Тек у овом тренутку атом кисеоника је спреман да се сједини са атомима водоника, процедура *bond(id : integer)*. У последњем кораку атом кисеоника дозвољава следећем атому кисеоника да започне поступак сједињавања са атомима водоника. Ово се постиже помоћу семафора *oxyMutex (signal (oxyMutex))*.

Део који се односи на атome водоника нешто је сложенији, али и овде је потребно водити рачуна и о синхронизацији између атoma водоника, али и према атому кисеоника са којим ће се ступити у реакцију. На почетку атом водоника чека да се појави атом кисеоника да би прешао на синхронизацију са другим атомима водоника (*wait (hydroSem)*). Пошто се у следећој фази модификује променљива која каже колико је до тог тренутка атoma водоника дошло до баријере потребно је урадити међусобно искључивање атoma водоника. Ово се постиже користећи бинарни семафор *hydroMutex (wait (hydroMutex))*. Када је у критичну секцију атом водоника увећава број приспелих

атома водоника. Уколико је број атома водоника постао довољан да се ступи у реакцију са атомом кисеоника (2), прелази се на део за синхронизацију са атомом кисеоника. Уколико није постао два чека се да се појави још један атом водоника. Треба констатовати да у тренутку када се један атом водоника налази у критичној секцији постоји тачно један атом кисеоника спреман да ступи у реакцију. Синхронизација са атомом кисеоника се постиже коришћењем семафора *oxySem* (*signal (oxySem)*). Поред синхронизације са атомом кисеоника потребно је обавестити и првопристигли атом водоника да се појавио и други водоник и да је могуће да се напусти баријера. Дато решење има јасно изражене фазе па је потребно семафор којим се сигнализира да је приспео други атом водоника постави два пута. Једном за први атом водоника, а други пут за исти тај атом водоника. Могуће је направити и другачије решење код кога би се направила мања груписаност по фазама, а код кога би се сигнализирало само блокираном атому водоника да настави даље. Као последњу ствар у критичној секцији потребно је вратити бројач броја пристиглих атома водоника на вредност 0. Након изласка из критичне секције (*signal (hydroMutex)*) прелази се на фазу провере да ли има довољно атома водоника (*wait (hydroSem2)*). Тек након завршетка ове фазе могуће је извршити формирање молекула воде.

Заједничке променљиве које се користе приликом решавања овог проблема су: семафори *hydroMutex*, који служи за међусобно искључивање атома водоника, *oxyMutex*, који служи за међусобно искључивање атома кисеоника, *hydroSem*, којим атом кисеоника сигнализира атому водоника да је спреман да ступи у реакцију, *oxySem*, којим атоми водоника сигнализирају атому кисеоника да је могуће формирати молекул воде, *hydroSem2*, којим атом водоника сигнализира постојање два атoma водоника спремних да ступе у реакцију, као и променљива *count* која специфицира број пристиглих атома водоника. Почетни услови неопходни за рад се тако одабирају да на почетку није било ни кисеоника ни водоника и да је потребно прво дочекати један атом кисеоника па тек онда прећи на чекање два атoma водоника. Ово се постиже постављањем *hydroSem* на 0 и *oxySem* на 0, као и постављањем бројача *count* на вредност 0. Поред тога потребно је поставити семафоре за међусобно искључивање тако да је дозвољено да тачно један од сваке врсте може да приступи баријери (*hydroMutex* се поставља на 1, као и *oxyMutex*). Семафор за међусобну синхронизацију атoma водоника, *hydroSem2*, се такође поставља на 0 јер не постоји претходни атом водоника.

```

program H2O;
var
    hydroSem : semaphore;
    hydroSem2 : semaphore;
    hydroMutex : semaphore;
    oxySem : semaphore;
    oxyMutex : semaphore;
    count : shared integer;
procedure bond(id : integer); begin ... end;
procedure Oxygen(id : integer);
begin
    wait (oxyMutex);
    signal (hydroSem);
    signal (hydroSem);
    wait (oxySem);
    bond (id);
    signal (oxyMutex);
end;
procedure Hydrogen(id : integer);
begin

```

```

wait (hydroSem);
wait (hydroMutex);
count := count + 1;
if (count = 2) then
begin
    count := 0;
    signal (oxySem);
    signal (hydroSem2);
    signal (hydroSem2)
end;
signal (hydroMutex);
wait (hydroSem2);
bond (id)
end;
begin
    init(hydroSem, 0);
    init(hydroSem2, 0);
    init(hydroMutex, 1);
    init(oxySem, 0);
    init(oxyMutex, 1);
    count := 0;
    cobegin
        Oxygen(1);
        Oxygen(2);
        ...
        Hydrogen(1);
        Hydrogen(2);
        ...
    coend;
end.

```

Могуће је направити решење код кога атоми водоника не би имали јасну поделу на фазе, али које би скратило време потребно за извршавање јер би имало једну *signal* и једну *wait* наредбу мање.

```

procedure Hydrogen(id : integer);
begin
    wait (hydroSem);
    wait (hydroMutex);
    count := count + 1;
    if (count = 2) then
    begin
        count := 0;
        signal (oxySem);
        signal (hydroSem2);
        signal (hydroMutex)
    end
    else
    begin
        signal (hydroMutex);
        wait (hydroSem2)
    end;
    bond (id)
end;

```

Недостатак овог решења је у случајевима када је процедури за сједињавање потребно проследити три идентификатора за сваки од атома. Атом кисеоника може да сазна идентификаторе атома водоника пошто су они блокирани у тренутку када он позива процедуру *bond*, али обрнуто није случај. Када атом водоника позове процедуру за сједињавање могуће да је на баријери већ пристигао неки нови атом кисеоника.

### Друго решење

Решење је дато за случај да је доволно да само један атом водоника зна идентификаторе преосталих атома у молекулу воде, и неопходно је да прво пристигне атом водоника.

```
program H2O;
var
    hydroSem : semaphore;
    hydroSem2 : semaphore;
    hydroMutex : semaphore;
    oxySem : semaphore;
    oxyMutex : semaphore;
    count : shared integer;
procedure bond(i : integer); begin ... end;
procedure Oxygen(i : integer);
begin
    wait (oxyMutex);
    signal (hydroSem);
    signal (hydroSem);
    wait (oxySem);
    bond (i);
end;
procedure Hydrogen(i : integer);
begin
    wait (hydroSem);
    wait (hydroMutex);
    count := count + 1;
    if (count = 2) then
        begin
            count := 0;
            signal (oxySem);
            signal (hydroSem2);
            signal (hydroSem2)
        end;
    signal (hydroMutex);
    wait (hydroSem2);
    bond (i);
    wait (hydroMutex);
    count := count + 1;
    if (count = 2) then
        begin
            count := 0;
            signal (oxyMutex)
        end;
    signal (hydroMutex)
end;
begin
```

```
init(hydroSem, 0);
init(hydroSem2, 0);
init(hydroMutex, 1);
init(oxySem, 0);
init(oxyMutex, 1);
count := 0;
cobegin
    Oxygen(1);
    ...
    Hydrogen(1);
    ...
coend;
end.
```

Недостатак овог, као и претходног, решења се јавља у случајевима када је процедуре за сједињавање потребно проследити три идентификатора за сваки од атома. Атоми водоника могу да сазнају идентификатор атома кисеоника као и међусобне идентификаторе, али обрнуто није случај. Када атом кисеоника позове процедуру за сједињавање могуће да је на баријери већ пристигао неки нови атом кисеоника и неки нови атоми водоника.

### Треће решење

Решење је дато за случај да је потребно да сваки атом зна идентификаторе свих атома у молекулу воде, и неопходно је да прво пристигне атом кисеоника.

Ово решење за заснива на коришћењу расподељених бинарних семафора који контролишу приступ баријери. Кисеонику је дата предност постављањем семафора *oxyMutex* на почетну вредност 1. Када кисеоник ексклузивно приступи баријери уписује своју идентификацију и пушта један атом водоника да дође на баријеру (*signal (hydroMutex)*) а сам се блокира док не дођу оба атома водоника (*wait (oxyGo)*). Када први атом водоника дође до баријере (*wait (hydroMutex)*), пошто има ексклузивно право приступа, уписује своју идентификацију на прву позицију и ослобађа баријеру за други атом водоника (*signal (hydroMutex)*) а сам се блокира док не дође други атом водоника (*wait (hydroGo)*). Када други атом водоника дође до баријере (*wait (hydroMutex)*) такође има ексклузивно право приступа уписује своју идентификацију на другу позицију, сједињују се са остала два атома. Након тога буди први атом водоника (*signal (hydroOk)*) а сам се блокира док се први атом водоника не сједини (*wait(hydroOk)*). Постоји сад први атом има ексклузивно право приступа идентификацијама чита их и сједињује се. Када се први атом водоника сједини враћа контролу/права приступа другом атому водоника (*signal (hydroOk)*), и завршава свој рад на баријери. Када се пробуди други атом водоника он буди атом кисеоника (*signal (oxyGo)*) а сам напушта ексклузивно право приступа. Када се атом кисеоника пробуди он може да сазна идентификацију два атома водоника јер има ексклузивно право приступа идентifikatorima. Када се атом кисеоника сједини препушта баријеру првом наредном атому кисеоника.

```
program H2O;

type Molecule = record
    oxId: integer;
    hyId : array [1..2] of integer
end;
```

```

var
    hydroMutex : semaphore;
    hydroGo : semaphore;
    hydroOk : semaphore;
    oxyMutex : semaphore;
    oxyGo : semaphore;
    count : shared integer;
    mol : shared Molecule;

procedure bond(mol : Molecule); begin ... end;
procedure Oxygen(id : integer);
begin
    wait (oxyMutex);
    mol.oxId := id;
    signal (hydroMutex);
    wait (oxyGo);
    bond (mol);
    signal (oxyMutex);
end;
procedure Hydrogen(id : integer);
begin
    wait (hydroMutex);
    count := count + 1;
    mol.hyId[count] := id;
    if (count = 2) then
        begin
            bond(mol);
            count := 0;
            signal (hydroGo);
            wait(hydroOk);
            signal (oxyGo);
        end
    else
        begin
            signal (hydroMutex);
            wait (hydroGo);
            bond (mol);
            signal (hydroOk);
        end;
end;
begin
    init(hydroMutex, 0);
    init(hydroGo, 0);
    init(hydroOk, 0);
    init(oxyMutex, 1);
    init(oxyGo, 0);
    count := 0;
    cobegin
        Oxygen(1);
        ...
        Hydrogen(1);
        ...
    coend;
end.

```

#### Четврто решење

Решење је дато за случај да је потребно да сваки атом зна идентификаторе свих атома у молекулу воде, и сасвим је свеједно који молекул прво долази.

Решење је потпуно симетрично за атоме водоника и атоме кисеоника и заснива се на коришћењу баријере са двоја врата и условној синхронизацији приликом тражења дозволе за приступ баријери. На почетку атом тражи дозволу за приступ баријери. Ово се постиже тако што атоми кисеоника чекају на семафору *oxySem* док атоми водоника чекају на семафору *hydroSem*. Након тога сви атоми долазе на баријеру која има двоја врата, улазна врата и излазна врата. Као и пре прва, улазна, врата су отворена док тачно три процеса не прође кроз њих (један кисеоник и два водоника) и уђу у тампон зону. Када се накупи тачно три процеса која чине молекул прва врата се затварају и отварају се друга, излазна, врата. Када се процес који је био блокиран на излазним вратима пробуди може да прочита идентификаторе преосталих атома у молекулу. Након овога се чека да сва три процеса прође и кроз друга врата како би се она затворила, тада се пушта да нови атом кисеоника и два атома водоника дођу до баријере и поново се отварају прва врата. Семафоре је потребно тако иницијализовани да прва врата буду отворена, друга врата затворена, дозвољен приступ до баријере једном атому кисеоника и двома атомима водоника.

```
program H2O;
type Molecule = record ids : array [0..2] of integer end;

var
    hydroSem, oxySem : semaphore;
    mutex1, mutex2 : semaphore;
    count : shared integer;
    mol : shared Molecule;

procedure bond(mol : Molecule); begin ... end;

procedure Oxygen(id : integer);
var mol : Molecule;
begin
    wait (oxySem);
    mol := barrier(id);
    bond(mol);
end;

procedure Hydrogen(id : integer);
var mol : Molecule;
begin
    wait (hydroSem);
    mol := barrier(id);
    bond(mol);
end;

function barrier(id : integer) : Molecule;
begin
    wait (mutex1);
    mol.ids[count] := id;
    count := count + 1;
    if (count = 3) then
        begin
```

```

        count := 0;
        signal (mutex2)
end
else
        signal (mutex1);

wait(mutex2);
{read mol}
barrier := mol;
count := count + 1;
if (count = 3) then
begin
        count := 0;
        signal (oxySem);
        signal (hydroSem);
        signal (hydroSem);
        signal (mutex1);
end
else
        signal (mutex2);

end;
begin
        count := 0;
        init(mutex1, 1);
        init(mutex2, 0);
        init(hydroSem, 2);
        init(oxySem, 1);

cobegin
        Oxygen(1);
        ...
        Hydrogen(1);
        ...
coend;
end.

```

## 14

## Проблем преласка реке

На обали реке се налази чамац који може да прими тачно четири путника (*The River Crossing Problem*). Постоје две групе људи које могу да користе тај чамац. Чамац може да исплови само ако се у њему налази тачно онолико путника колики му је капацитет, али само под условом да у тренутку испловљавања ни једна група не сме бити у већини над другом (није дозвољена комбинација да из једне групе буде три, а из друге један путник). Користећи семафоре написати програм који симулира понашање путника овог чамца.

## Решење

У овом проблему се ради о синхронизацији на баријери без процеса координатора за два типа процеса, две групе људи. Код овог проблема је познат укупан број процеса који могу проћи кроз баријеру, али за разлику од претходних проблема није познат тачан број процеса из које групе ће проћи кроз баријеру. Могуће су комбинације: четири из прве и ниједан из друге групе, два из прве и два из друге групе, и ниједан из прве и четири из друге групе.

На почетку путник долази до чамца, узима ексклузивно право приступа бројачима путника, и увећава број путника своје групе који чекају да уђу у чамац. Овде је битно нагласити да путници не улазе у чамац већ чекају испред чамца док се не формира група која може да започне вожњу. Уколико ово није путник који је омогућио формирање групе онда враћа ексклузивно право приступа бројачу, и чека да се формира група. Уколико путник детектује да је број путника испред чамца таква да вожња може да започне овај путник се проглашава за капетана и обавештава све путнике у групи (укључујући и себе) да је група формирана, не враћа се ексклузивно право приступа бројачима. Када путник дочека обавештење да може ући у чамац улази у чамац и обавештава капетана (баријеру) да је ушао. Када капетан детектује да су сви путници ушли у чамац започиње се вожња. На крају вожње капетан враћа ексклузивно право приступа бројачима.

```
program RiverCrossing;
var groupASem: semaphore;
    groupBSem: semaphore;
    mutex: semaphore;
    barrier: semaphore;
    groupA, groupB: shared integer;
procedure board (id : integer); begin ... end;
procedure rowBoat (id : integer); begin ... end;

procedure PassengerA(id : integer);

var captain : boolean;
begin
    captain := false;
    wait (mutex);
    groupA := groupA + 1;
    if groupA = 4 then
    begin
        signal (groupASem);
```

```

signal (groupASem);
signal (groupASem);
signal (groupASem);
groupA := 0;
captain := true;
end
else if (groupA = 2) and (groupB >= 2) then
begin
    signal (groupASem);
    signal (groupASem);
    signal (groupBSem);
    signal (groupBSem);
    groupB := groupB - 2;
    groupA := 0;
    captain := true;
end
else
    signal (mutex);
wait (groupASem);

board (id);
signal (barrier);

if captain then
begin
    wait(barrier);
    wait(barrier);
    wait(barrier);
    wait(barrier);
    rowBoat(id);
    signal (mutex);
end;
end;
procedure PassengerB(id : integer) ;

var captain : boolean;
begin
    captain := false;

    wait (mutex);
    groupB := groupB + 1;
    if groupB = 4 then
    begin
        signal (groupBSem);
        signal (groupBSem);
        signal (groupBSem);
        signal (groupBSem);
        groupB := 0;
        captain := true;
    end
    else if (groupB = 2) and (groupA >= 2) then
    begin
        signal (groupASem);
        signal (groupASem);
        signal (groupBSem);

```

```
    signal (groupBSem);
groupA := groupA - 2;
groupB := 0;
captain := true;
end
else
    signal (mutex);

wait (groupBSem);

board (id);
signal (barrier);

if captain then
begin
    wait (barrier);
    wait (barrier);
    wait (barrier);
    wait (barrier);
    rowBoat(id);
    signal (mutex);
end;
end;

begin
groupA := 0;
groupB := 0;
init(groupASem, 0);
init(groupBSem, 0);
init(mutex, 1);
init(barrier, 0);
cobegin
    PassengerA(1);
    ...
    PassengerB(1);
    ...
coend;
end.
```

Недостатак овог решења, код које само капетан позива процедуру за вожњу, је у случајевима када процедура за вожњу чамца потребно да позову сви путници и тек након што сви потврде да су завршили са вожњом чамац може да се врати. У том случају је решење потребно модификовати, на начин описан у претходном задатку, увођењем јединствене баријере приликом изласка.

```
procedure PassengerA(id : integer);
var captain : boolean;
begin
    ...
    board (id);
    signal (barrier);
    if captain then
begin
    wait (barrier);
    wait (barrier);
```

```
    wait(barrier);
    wait(barrier);
    signal (mutex2);
end;

rowBoat(id);

wait(mutex2);
cnt := cnt + 1;
if(cnt = 4) then
begin
    cnt := 0;
    signal(mutex)
end
else
    signal(mutex2);
end;
```

→ Шта је потребно променити у овом решењу како би сви путници сазнали идентификаторе свих преосталих путника у групи?

15

## Студентска журка

У студентском дому студенти могу да организују журке (*The Room Party Problem*). На журку могу да долазе студенти, на журци могу да пију и могу са журке да одлазе. На журци може бити произвољан број студената. Управник студентског дома долази у обиласак соба само у случају да у соби нема студената или у случају да их је више од 50. Када је управник у соби нови студенти не могу да улазе али присутни могу да је напуштају. Управник излази из собе тек када сви студенти из ње изађу. Користећи семафоре написати програм за студенте и за управника.

## Решење

Управник дома улази у собу уколико је број студената у тој соби или 0 или је већи од максималног дозвољеног броја студената. Како би ово проверио управник забрањује другим студентима да улази у собу и тражи ексклузивно права за проверу овог бројача.

Уколико је број студената већи од нуле и мањи од максималног броја студената онда поставља обавештење да чека испред собе док се соба не испразни или док се соба не препуни. Како би омогућио да се соба пуни враћа право студентима да улазе у собу. Такође враћа и ексклузивно право приступа бројачу како би студенти који долазе и одлазе могли да га мењају. Након тога управник чека док га неко не прободи. Управника буди или последњи студент који излази из собе или први који препуњава собу. Након овога управник прелази на проверу да ли је соба препуњена или празна.

Уколико је број студената већи од максималног и соба је препуњена, било да је управни чекао да се соба препуни или је директно наишао на препуну собу, обавештава студенте да је ушао у собу и да је журка завршена, враћа ексклузивно право приступа бројачу како би студенти могли да излазе и да га умањују, и чека да му последњи студент који је био у соби не јави да су сви изашли. Тада враћа дозволу за улазак у собу, обавештава да је изашао из собе и враћа ексклузивно право приступа бројачу.

Уколико је број студената 0 и соба је празна, било да је управни чекао да се соба испразни или је директно наишао на празну собу, обавља претрагу празне собе. Након тога уколико је директно наишао на празну собу враћа дозволу за улазак у собу, уколико није наишао на празну собу него се она у међувремену испразнила не враћа ову дозволу јер ју је у међувремену већ вратио. На крају обавештава да је изашао из собе, и враћа ексклузивно право приступа бројачу.

Студент долази на журку тако што прво тражи дозволу и привремено забрањује другим студентима да улази у собу и тражи ексклузивно права да увећа бројач присутних студената.

Уколико је број студената већи од максималног броја студената и управник чека испред собе онда само дојављује управнику да може да обиђе собу јер је соба пунा, у супротном враћа ексклузивно право приступа бројачу и враћа дозволу уласка у собу.

Након завршене журке студент тражи ексклузивни право приступа бројачу како би умањио број присутних студената. Треба приметити да студент не тражи дозволу за улазак. Уколико је број студента који су преостали у соби 0 и управник чека обавештава га. Управник је могао да чека из два разлога. Први разлог је да наишао на собу која није била пуна па је чекао или да се соба испразни или препуни, и други разлог је да соба била препуњена па је управник наредио да се соба испразни. Уколико број студената није 0 само враћа ексклузивно право приступа бројачу.

Променљиве које су коришћене у овом решењу су: *students* - број студената у соби, и *deanStatus* статус управника дома (чека испред собе, у соби је и није у соби); Семафори коришћени су: *mutex* - за омогућавање ексклузивног права приступа бројачу студената; *clear* - служи за обавештавање управника да су сви студенти изашли из препуњене собе; *comeIn* - служи за обавештавање управника да су сви студенти изашли из собе која није била препуњена; *enter* - служи за дозволу уласка студената у собу; Почетне вредности би биле да је број студената у соби 0, да управник није у соби, да студенти могу да улазе у собу, да је дозвољен приступ бројачу студената, и да нема обавештења за управника. Треба приметити да је довољно да постоји само један од семафора који означавају да управник може ући у собу или су у решењу остављена два семафора како би се повећала читљивост кода.

```

program RoomParty;
const MAXSTUDENTS = 50;
var
    students : shared integer;
    deanStatus : shared String;
    mutex : semaphore;
    clear : semaphore;
    comeIn : semaphore;
    enter : semaphore;

procedure breakup; begin ... end;
procedure search; begin ... end;
procedure party; begin ... end;

procedure Dean;
var empty : boolean;

begin
    empty := false;
    wait(enter);
    wait(mutex);
    if ((students > 0) and (students < MAXSTUDENTS)) then
    begin
        deanStatus := 'waiting';
        signal(mutex);
        signal(enter);
        empty := true;
        wait(comeIn);
    end;
    if (students >= MAXSTUDENTS) then
    begin
        deanStatus := 'in the room';
        breakup;
        signal(mutex);
        wait(clear);
        signal(enter);
    end
    else
    begin
        search;
        if (not empty) then signal(enter);
    end;
end;

```

```
deanStatus := 'not here';
    signal(mutex);
end;

procedure Student(id : integer);
begin
    wait(enter);
    wait(mutex);
    students := students + 1;
    if ((students = MAXSTUDENTS) and (deanStatus = 'waiting')) then
        signal(comeIn)
    else
        begin
            signal(mutex);
            signal(enter);
        end;
    party;

    wait(mutex);
    students := students - 1;

    if ((students = 0) and (deanStatus = 'waiting')) then
        signal(comeIn)
    else if ((students = 0) and (deanStatus = 'in the room')) then
        signal(clear)
    else
        signal(mutex);
end;
begin
    students := 0;
    deanStatus := 'not here';
    init(mutex, 1);
    init(clear, 0);
    init(comeIn, 0);
    init(enter, 1);
    cobegin
        Student(1);
        ...
        Dean;
    coend;
end.
```

## Условни критични региони

Код програмске парадигме критичних региона полази се од чињенице да се у конкурентним програмима често јавља потреба за приступ критичној секцији. Основна одлика ове програмске парадигме је увођење посебне синтаксе за експлицитно означавање критичних секција, који се у овом контексту називају – критични региони. Код приступа критичном региону је имплицитно обезбеђено међусобно искључивање процеса.

Синтакса и семантика критичних региона намећу да се дељене променљиве морају наћи унутар једног и само једног ресурса. Другачије речено, ресурси конкурентног програма су дисјунктни склопови дељених променљивих. Како приступ променљивама ресурса није дозвољен изван критичних региона, преводилац обезбеђује формалну контролу којом се отклања низ потенцијалних грешака у коду.

Механизам критичних региона може се имплементирати преко бинарних семафора тако што се за сваки критични регион који приступа истим дељеним променљивама уведе по један семафор. На почетку сваког критичног региона се поставља наредба *wait* на одговарајућем семафору, а на крају наредба *signal* на том семафору, чиме се обезбеђује међусобно искључивање процеса.

Условни *критични рејон* је критични регион који поред обезбеђивања међусобног искључивања има и механизам за условну синхронизацију процеса преко (опционих) *await* наредби. Када се унутар критичног региона нађи на *await* наредбу чији услов није задовољен, процес се блокира и притом одриче ексклузивног права приступа ресурсу, јер неки други процес можда чека да уђе у тај критични регион да би омогућио да се услов испуни (тиме се избегава могућа ситуација да се два процеса међусобно бескрајно дуго чекају и настајање мртвог блокирања (*deadlock*)). Након испуњења датог услова и поновног добијања права ексклузивног приступа, процес се деблокира и наставља рад. По изласку из критичног региона, један од процеса који су били блокирани чекајући да уђу у критични регион се деблокира и добија право ексклузивног приступа региону. Тај поступак се понавља док год има процеса који чекају на улазак у критични регион. Погодности условних критичних региона у односу на семафоре су: апстракција вишег нивоа у раду са критичним секцијама, имплицитна реализација међусобног искључивања и условне синхронизације, нотациона погодност (богља читљивост програма) и смањење могућности грешке (рецимо због неупаривања *wait* и *signal* семафора).

И механизам условних критичних региона се може имплементирати преко семафора. Међусобно искључивање процеса се може реализовати на исти начин као код обичних критичних региона, а условна синхронизација се може реализовати помоћу механизма „предаје штафетне палице“ тако што се за сваки услов уводи један *split binary semaphore* (видети задатак 5).

За реализацију програма помоћу механизма условних критичних региона користићемо синтаксу језика проширенi Pascal са одговарајућим синтаксним проширењима:

- Декларација ресурса одређеног типа изгледа као и декларација сваке друге дељене променљиве:

```
res: shared type;
```

Уколико има више дељених променљивих које чине ресурс, оне се могу најпре све декларисати као део једног записа, а затим се тај запис декларише као дељена променљива:

```
type resourcetype = record s1:type1; ... sN:typeN; end;
res: shared resourcetype;
```

Променљивој која је ресурс назначен са *shared* (овде - запису *res*) се може приступати само унутар условних критичних региона за ту променљиву.

- Условни критични регион се синтаксно дефинише на следећи начин (*await* наредба је опциона и ако се изостави добијамо обичан критични регион):

```
region res do
begin
  ...
  [await(condition)];
  ...
end;
```

У већини имплементација, *await* наредба је дозвољена само на почетку условног критичног региона. Тада процес на уласку у регион истовремено недељиво испитује услов синхронизације и да ли се неки процес већ налази у критичном региону. Само ако су оба услова истовремено испуњена, процес може да уђе у регион.

## 16 Проблем критичне секције

Дат је упоредни програм на језику проширени Pascal:

```
program TimeDependent;
var x : shared integer;
begin
  x := 1;
  cobegin
    x := x + 1;
    x := x + 3;
  coend;
  writeln('x=', x)
end.
```

- Одредити све могуће излазне резултате програма
- Отклонити временску зависност у датом програму употребом критичних региона.

### Решење

а) Приликом анализе сличног проблема у задатку 1 закључили смо да свака од две конкурентне наредбе ( $x := x+1$  и  $x := x+3$ ) обавља недељиве операције - читање и упис у меморију – чије међусобно преплиттање доводи до тога да се могу појавити различите вредности на излазу. У Табели 3 дати су сви могући сценарији приступа меморији и одговарајуће коначне вредности променљиве  $x$ .

		редослед операција →				Излаз
	Операција	R1	W1	R2	W2	
1.	Вредност $x$ након операције	1	2	2	5	5
2.	Операција	R1	R2	W1	W2	
	Вредност $x$ након операције	1	1	2	4	4
3.	Операција	R1	R2	W2	W1	
	Вредност $x$ након операције	1	1	4	2	2
4.	Операција	R2	R1	W1	W2	
	Вредност $x$ након операције	1	1	2	4	4
5.	Операција	R2	R1	W2	W1	
	Вредност $x$ након операције	1	1	4	2	2
6.	Операција	R2	W2	R1	W1	
	Вредност $x$ након операције	1	4	4	5	5

Табела 3. Сви могући сценарији током извршавања програма (редослед извршавања атомских операција читања и уписа у меморију је са лева у десно). За сваки од сценарија дате су међувредности променљиве  $x$  и наглашене су вредност које ће се исписати. Могуће излазне вредности су: 2, 4 и 5.

б) Као и код проблема 1 закључујемо да временску зависност у програму можемо елиминисати ако обезбедимо међусобно искључивање конкурентних наредби приликом приступа дељеној променљивој  $x$ .

Пошто критични региони по дефиницији имплицитно обезбеђују међусобно искључивање, довољно је сваку од наредби које приступају дељеној променљивој  $x$  прогласити критичним регионом. Тиме се обезбеђује да се наредбе извршавају секвенцијално, прво једна, па друга (то одговара варијантама 1 и 6 из Табеле 3), па ће се у том случају на излазу добити резултат 5:

```
program TimeIndependent;
var x : shared integer;
begin
  x := 1;
  cobegin
    region x do x := x + 1;
    region x do x := x + 3;
  coend;
  writeln('x=', x)
end.
```

## 17

## Произвођач и потрошач: синхронизација процеса

Дат је упоредни програм на језику проширенi Pascal (као у задатку 2):

```
program Graph;
const n = ...;
var x, y : shared integer;

procedure makepoints;
var i : integer;
begin
    for i := 1 to n do begin
        x := i; y := i * i;
    end
end;

procedure printpoints;
var i : integer;
begin
    for i := 0 to n do write('(', x, ', ', ', ', y, ')');
end;

begin
    x := 0; y := 0;
    cobegin
        makepoints;
        printpoints;
    coend
end.
```

Жељени излаз програма је низ парова облика: (0,0) (1,1) (2,4) ... ( $n,n^2$ ). Отклонити временску зависност у датом програму употребом

- а) условних критичних региона
- б) критичних региона

## Решење

а) Приликом анализе овог проблема у задатку 2 закључили смо да временска зависност програма потиче од тога што није обезбеђено међусобно искључивање, као и да је потребно синхронизовати рад процеса производа и потрошача. Решење са условним критичним регионима би изгледало овако:

```
program Graph;
const n = ...;
type point = record x, y : integer end;
var pt : shared record
    p : point;
    full : boolean;
end;

procedure makepoints;
var i : integer;
```

```

begin
    for i := 1 to n do
        region pt do begin
            await(not pt.full);
            pt.p.x := i; pt.p.y := i * i;
            pt.full := true
        end
    end;

procedure printpoints;
var i : integer;
begin
    for i := 0 to n do
        region pt do begin
            await(pt.full);
            write('(', pt.p.x, ', ', pt.p.y, ')');
            pt.full := false
        end
    end;
end;

begin
    pt.p.x := 0; pt.p.y := 0; pt.full := true;
    cobegin
        makepoints;
        printpoints;
    coend
end.

```

Међусобно искључивање је имплицитно обезбеђено приликом уласка у критични регион у коме се приступа ресурсу *pt* (односно дељеним променљивама *p.x* и *p.y*), док се условна синхронизација одвија преко услова у *await* наредби. Синхронизациони услов се изражава помоћу логичке променљиве *full* која сигнализира када је генерисана нова тачка (*full=true*) и када је тачка исписана (*full=false*). *Await* наредбе се у овом случају налазе на почетку условних критичних региона, па се испитивања да ли је неки процес у критичној секцији и испитивања услова *await* наредбе обавља недељиво.

б) Код решења са обичним критичним регионима потребно је на неки начин решити проблем синхронизације између процеса у недостатку *await* наредбе. То се може учинити тако што би се чекање на задовољење услова у *await* наредби из решења са условним критичним регионима реализовало преко запосленог чекања:

```

program Graph;
const n = ...;
type point = record x, y : integer end;
var pt : shared record
    p : point;
    full : boolean;
end;

procedure makepoints;
var i : integer;
    b : boolean;
begin
    for i := 1 to n do begin
        b := true;

```

```

while b do
    region pt do begin
        if not(pt.full) then
            begin
                pt.p.x := i; pt.p.y := i*i;
                b := false;
                pt.full := true;
            end;
        end
    end;
end;

procedure printpoints;
var i : integer;
    b : boolean;
begin
    for i := 0 to n do begin
        b := true;
        while b do
            region pt do begin
                if pt.full then
                    begin
                        write('(', pt.p.x, ', ', pt.p.y, ')');
                        b := false;
                        pt.full := false;
                    end;
                end;
            end;
        end
    end;
end;

begin
    pt.p.x := 0; pt.p.y := 0; pt.full := true;
    cobegin
        makepoints;
        printpoints;
    coend
end.

```

Уколико би било дозвољено приступати променљивама региона и ван блока региона, што неки програмски језици не дозвољавају, онда би решење било:

```

program Graph;
...
procedure makepoints;
var i : integer;
    b : boolean;
begin
    for i := 1 to n do begin
        b := true;
        while b do
            if not(pt.full) then
                region pt do begin
                    pt.p.x := i; pt.p.y := i*i;
                    b := false;
                    pt.full := true;
                end;
            end;
    end;

```

```
        end
    end
end;

procedure printpoints;
var i : integer;
b: boolean;
begin
    for i := 0 to n do begin
        b := true;
        while b do
            if pt.full then
                region pt do begin
                    write('(', pt.p.x, ', ', pt.p.y, ')');
                    b := false;
                    pt.full := false;
                end
        end
    end;
end;
```

**Найомена:** Дато решење важи само за један процес произвођач и један процес потрошач. У случају више произвођача и више потрошача, програм не би био коректан. Јавио би се проблем због чињенице да испитивање синхронизационог услова (који је раније био у *await* наредби) и услова уласка у критичну секцију мора бити недељиво.

- ➔ Решити задатак из тачке б) применом механизма „предаје штафетне палице”.

**18****Мост који има само једну коловозну траку**

Аутомобили који долазе са севера и југа долазе до моста који има само једну коловозну траку (*The One Lane Bridge Problem*). Аутомобили из супротног смера не могу истовремено да буду на мосту.

- Написати процедуре којима се симулира понашање аутомобила ако претпоставимо да се због слабе конструкције моста на њему може наћи највише један аутомобил.
- Дати решење проблема ако претпоставимо да аутомобили који долазе из истог смера могу без икаквих ограничења истовремено да прелазе мост.
- Усавршити претходно решење тако да буде праведно, односно да аутомобили не могу да чекају неограничено дуго како би прешли мост. Да би се то постигло смер саобраћаја треба да се мења сваки пут након што мост пређе одређен број аутомобила из једног истог смера  $N$  (ово важи под условом да је бар један аутомобил чекао да пређе мост са супротне стране; док год се не појави аутомобил са супротне стране аутомобили из истог смера могу слободно да прелазе мост).
- Модификовати решење из тачке б) ако претпоставимо да због ограниченог капацитета највише  $K$  аутомобила из истог смера може истовремено да се нађе на мосту (*The Old Bridge Problem*).

**Решење**

- Мост је, очигледно, дељени ресурс. Према услову задатка само један аутомобил сме да се нађе на мосту, па је услов ступања на мост да нема ниједног другог аутомобила на њему (ни из супротног, ни из истог смера). Дакле, потребно је само извршити међусобно искључивање процеса приликом приступа ресурсу *bridge*:

```

program OneLaneBridge;
const NSouthCars = ...; NNorthCars = ...;
type bridgetype = ...;
var bridge: shared bridgetype;

procedure SouthCar(i : 0..NSouthCars-1);
begin
  region bridge do cross the bridge;
end;

procedure NorthCar(i : 0..NNorthCars-1);
begin
  region bridge do cross the bridge;
end;

begin
  cobegin
    SouthCar(0); ... SouthCar(NSouthCars-1);
    NorthCar(0); ... NorthCar(NNorthCars-1);
  coend
end.

```

б) У случају када више аутомобила из истог смера може истовремено да буде на мосту, услов да аутомобил сме да почне прелаз може се формулисати преко променљивих *ncs* (*number of cars from south* - број аутомобила са југа који се тренутно налазе на мосту) и *ncn* (*number of cars from north* - број аутомобила са севера који се тренутно налазе на мосту). Ове две вредности у потпуности описују стање на мосту. Према услову задатка аутомобили ступају на мост само ако нема аутомобила из супротног смера, што се проверава испитивањем да ли је *lsp=0* (за аутомобиле са југа), односно *ncs=0* (за аутомобиле са севера); треба приметити да у сваком тренутку бар једна од променљивих *ncs* или *lsp* мора бити 0. По испуњењу одговарајућег услова, аутомобил ступа на мост (притом се инкрементира *ncs*, односно *lsp*), а затим се излази из критичног региона како би и други аутомобили могли да провере стање на мосту. Тиме се завршава улазни (*entry*) протокол. Прелаз аутомобила преко моста се одвија у паралели са осталим аутомобилима. По завршетку преласка, аутомобил слизи са моста (притом се декрементира *ncs*, односно *lsp*, као део излазног (*exit*) протокола).

```
program OneLaneBridge;
const SouthCarsNo = ...; NorthCarsNo = ...;
var bridge : shared record
    ncs, ncn : integer;
end;

procedure SouthCar(i : 0..SouthCarsNo-1);
begin
    region bridge do begin
        await (ncn = 0);
        ncs := ncs + 1
    end
    cross the bridge;
    region bridge do ncs := ncs - 1
end;

procedure NorthCar(i : 0..NorthCarsNo-1);
begin
    region bridge do begin
        await (ncs = 0);
        ncn := ncn + 1
    end
    cross the bridge;
    region bridge do ncn := ncn - 1
end;

begin
    bridge.ncs := 0; bridge.ncn := 0;
    ...
end.
```

Недостатак овог решења је што не води рачуна о праведности код приступа ресурсу. Ако аутомобили из једног смера долазе довољно често, може се десити да аутомобили из супротног смера никада не добију право да пређу мост и да дође до изгладњивања процеса.

в) Код претходног решења проблема моста са једном коловозном траком може да се дододи ситуација да аутомобили чекају неограничено дуго како би прешли мост. Да би се овај проблем решио потребно је смер саобраћаја променити сваки пут након што

мост пређе одређен број аутомобила из супротног смера (ово важи под условом да је бар један аутомобил чекао да пређе мост са супротне стране; док год се не појави аутомобил са супротне стране аутомобили из истог смера могу слободно да прелазе мост).

Овакво решење захтева вођење знатно сложеније евиденције од оне у претходном решењу тако да није доволно да водимо рачуна о томе колико се аутомобила у неком тренутку налази на мосту, *crossing*, већ и о онима који чекају да пређу мост, *waiting*, али и о онима који су почели да прелазе мост почев од тренутка када се на другој страни појавио барем један аутомобил који чека, *ahead*. Ову евиденцију је потребно водити за сваку страну моста независно.

Када аутомобил дође до моста он увећава број аутомобила који чекају да пређу мост са одговарајуће стране, *waiting* := *waiting* + 1. Услов да аутомобил сме да почне да прелази мост може се формулсати преко променљивих које говоре колико аутомобила из супротног правца прелази мост, *north.crossing* за аутомобиле са југа и *south.crossing* за аутомобиле са севера, као и преко променљиве која говори колико аутомобила из посматраног правца је започело са преласком моста почев од тренутка када се на другој страни појавио први аутомобил. По услову задатка аутомобили ступају на мост само ако нема аутомобила из супротног смера, као и уколико је број аутомобила са посматране стране који је започео са преласком моста почев од тренутка када се на другој страни појавио први аутомобил није већи од *N*. По испуњењу одговарајућег услова а пре ступања аутомобила на мост мора се умањити број аутомобила који чекају да пређу мост, *waiting* := *waiting* - 1, увећати број аутомобила који тренутно прелазе мост, *crossing* := *crossing* + 1, и проверити да ли са друге стране постоји аутомобил који чека, уколико постоји увећати број аутомобила са посматране стране који је започео са преласком моста почев од тренутка када се на другој страни појавио први аутомобил, *ahead* := *ahead* + 1. Тиме се завршава *entry* протокол. Прелаз аутомобила преко моста се одвија у паралели са осталим аутомобилима. По завршетку преласка, аутомобил сипази са моста, тиме умањује број аутомобила који се налазе на мосту, *crossing* := *crossing* - 1. Ово није доволно да би се завршио прелазак моста, уколико се ради о последњем аутомобилу на мосту потребно је омогућити и другој страни да прелази мост постављањем променљиве *ahead* на вредност 0. Треба приметити да се ова променљива не поставља за страну оног ко прелази мост већ за страну онога ко чека да пређе мост, аутомобили са севера постављају *south.ahead* а аутомобили са југа *north.ahead*.

```

program OneLaneBridge;
const SouthCarsNo = ...;
    NorthCarsNo = ...;
    N = 10;
type direction = record
    waiting, crossing, ahead: integer;
end;
var bridge : shared record
    south, north: direction;
end;

procedure SouthCar(i : 0..SouthCarsNo-1);
begin
    region bridge do
        begin
            south.waiting := south.waiting + 1;
            await((north.crossing = 0) and (south.ahead < N));
            south.waiting := south.waiting - 1;
        end;
    end;

```

```

        south.crossing := south.crossing + 1;
        if (north.waiting > 0) then
            south.ahead := south.ahead + 1;
    end;
    cross the bridge;
region bridge do
begin
    begin
        south.crossing := south.crossing - 1;
        if (south.crossing = 0) then
            north.ahead := 0;
    end
end;
procedure NorthCar(i : 0..NorthCarsNo-1);
begin
region bridge do
begin
    north.waiting := north.waiting + 1;
    await((south.crossing = 0) and (north.ahead < N));
    north.waiting := north.waiting - 1;
    north.crossing := north.crossing + 1;
    if (south.waiting > 0) then
        north.ahead := north.ahead + 1;
    end;
cross the bridge;
region bridge do
begin
    north.crossing := north.crossing - 1;
    if (north.crossing = 0) then
        south.ahead := 0;
    end
end
end;
begin
bridge.south.waiting := 0;
bridge.south.crossing := 0;
bridge.south.ahead := 0;
bridge.north.waiting := 0;
bridge.north.crossing := 0;
bridge.north.ahead := 0;
cobegin
    ...
coend;
end.

```

г) У случају када коначно много аутомобила из истог смера може истовремено да буде на мосту, услов да аутомобил сме да почне прелаз може се формулисати преко променљивих *ncs* (*number of cars from south* - број аутомобила са југа који се тренутно налазе на мосту) и *ncn* (*number of cars from north* - број аутомобила са севера који се тренутно налазе на мосту). Ове две вредности у потпуности описују стање на мосту. По услову задатка аутомобили ступају на мост само ако нема аутомобила из супротног смера, што се проверава испитивањем да ли је *ncn=0* (за аутомобиле са југа), односно *ncs=0* (за аутомобиле са севера), као и уколико број аутомобила који се тренутно налазе на мосту није већи од *K* што се проверава испитивањем *ncs < K* (за аутомобиле са југа), односно *ncn < K* (за аутомобиле са севера); треба приметити да у сваком тренутку бар једна од променљивих *ncs* или *ncn* мора бити 0. По испуњењу

одговарајућег услова, аутомобил ступа на мост (притом се инкрементира *ncs*, односно *nsp*), а затим се излази из критичног региона како би и други аутомобили могли да провере стање на мосту. Тиме се завршава *entry* протокол. Прелаз аутомобила преко моста се одвија у паралели са осталим аутомобилима. По завршетку преласка, аутомобил слизи са моста (притом се декрементира *ncs*, односно *nsp*, као део *exit* протокола).

```

program OneLaneOldBridge;
const SouthCarsNo = ...;
    NorthCarsNo = ...;
    K = ...;
var bridge: shared record
    ncs, ncn : integer;
end;

procedure SouthCar(i : 0..SouthCarsNo-1);
begin
    region bridge do begin
        await ((ncn = 0) and (ncs < K));
        ncs := ncs + 1;
    end;
    cross the bridge;
    region bridge do begin
        ncs := ncs - 1;
    end;
end;

procedure NorthCar(i : 0..NorthCarsNo-1);
begin
    region bridge do begin
        await ((ncs = 0) and (ncn < K));
        ncn := ncn+1;
    end;
    cross the bridge;
    region bridge do begin
        ncn := ncn - 1;
    end;
end;

begin
    bridge.ncs := 0; bridge.ncn := 0;
    ...
end.

```

## 19

### Филозофи за ручком

Решити проблем филозофа за ручком, описан у задатку 4, применом условних критичних региона.

## Решење

Решење које највише одговара парадигми условних критичних региона поклапа се са идејом шестог решења задатка 4. Код овог решења deadlock се спречава атомизацијом приступа виљушкама - дозвољава се узимање само обадве виљушке заједно, уз међусобно искључивање филозофа приликом приступа виљушкама.

```
program DiningPhilosophers;
const N = 5;
var forks_available : shared array[0..N-1] of 0..2;
    i : integer;

procedure think; begin ... end;
procedure eat; begin ... end;

procedure Philosopher(i : 0..N-1);
var left, right : 0..N-1;
begin
    left := (i - 1) mod N; {left neighbour}
    right := (i + 1) mod N; {right neighbour}
    while (true) do begin
        think;
        region forks_available do begin
            await (forks_available[i] = 2);
            forks_available[left] := forks_available[left] - 1;
            forks_available[right] := forks_available[right]-1;
        end;
        eat;
        region forks_available do begin
            forks_available[left] := forks_available[left] + 1;
            forks_available[right] := forks_available[right]+1;
        end;
    end
end;
begin
    for i := 0 to N-1 do forks_available[i] := 2;
    cobegin
        Philosopher(0);
        ...
        Philosopher(N-1);
    coend
end.
```

Након добијања дозволе, процес улази у критични регион и проверава да ли су обе виљушке доступне (помоћу `await` наредбе); ако јесу, узима их, а ако нису, приступ критичном региону се ослобађа (у складу са дефиницијом условног критичног региона), чиме се омогућава да други филозофи изврше проверу и узму виљушке; бар један од

њих ће сигурно успети у томе. Блокирани процеси, дакле, не држе виљушке, па се не може се десити ситуација у којој филозофи чекају један другог.

Треба приметити да се улазак у критични регион и провера услова врше недељиво. У једном случају процес након добијања права приступа критичном региону одмах констатује да су обе виљушке на располажању и узима их. У другом случају констатује да виљушке нису расположиве и препушта право приступа критичном региону; међутим, у неком тренутку ће обе виљушке постати доступне за филозофа, па ће право приступа критичном региону бити враћено и он ће моћи да узме виљушке. Ове две акције се одвијају недељиво, јер се право приступа критичном региону враћа само у случају да су обе виљушке постале доступне и да ниједан други процес није у условном критичном региону над ресурсом *forks\_available*.

Све што је важило за одговарајуће решење са семафорима, стоји и овде.

→ Коментарисати следеће решење проблема филозофа за ручком. Елементи низа *forks* представљају одговарајуће виљушке:

```
program DiningPhilosophers;
const N = 5;
type fork = record ... end;
var forks : array[0..N-1] of shared fork;
    i : integer;

procedure think; begin ... end;
procedure eat; begin ... end;

procedure Philosopher(i : 0..N-1);
var left, right: 0..N-1;
begin
    left := i;
    right := (i + 1) mod N;
    while (true) do begin
        think;
        region forks[left] do
            region forks[right] do
                eat;
    end
end;

begin
    cobegin
        Philosopher(0);
        ...
        Philosopher(N-1);
    coend
end.
```

### Одговор

Ово решење може довести до мртвог блокирања (*deadlock*) према сценарију описаном у решењу задатка 4. Треба приметити да тип елемената низа *forks* није специфициран и да може бити било који, као и да одговарајући елементи нису иницијализовани. То је из простог разлога што се они и не користе, већ само формално представљају виљушке; елементи низа служе само за реализацију међусобног искључивања процеса код приступа виљушкама.

→ Коментарисати следеће решење проблема филозофа за ручком. Елементи низа *thinking* имају вредност *true* ако одговарајући филозоф размишља:

```
program DiningPhilosophers;
const N = 5;
var thinking : shared array[0..N-1] of boolean;
    i : integer;

procedure think; begin ... end;
procedure eat; begin ... end;

procedure Philosopher(i : 0..N-1);
var left, right : 0..N-1;
begin
    left := (i - 1) mod N;
    right := (i + 1) mod N;
    while (true) do begin
        think;
        region thinking do begin
            await (thinking[left] and thinking[right]);
            thinking[i] := false
        end;
        eat;
        region thinking do thinking[i] := true
    end;
end;
begin
    for i := 0 to N-1 do thinking[i] := true;
    cobegin
        Philosopher(0);
        ...
        Philosopher(N-1);
    coend
end.
```

## Одговор

Ово решење је семантички еквивалентно решењу из задатка 19. Разлика је у томе што се тренутно стање прати на другачији начин. Сходно томе, све што је важило за решење задатка 19 важи и овде.

За решење је битно само да сваки филозоф може да провери да ли су му две виљушке доступне, а када узме виљушке да остали филозофи буду у стању да то констатују. У првом решењу се стање прати преко броја виљушки на столу, а у другом се то ради преко индикатора о тренутној активности филозофа. И у једном и у другом случају филозоф може испитати да ли су му обе виљушке доступне (у првом испитујући број виљушака поред њега, а у другом провером да ли његови суседи размишљају), као и сигнализирати промену стања на одговарајући начин када узме виљушке (у првом случају смањујући број виљушака које његови суседи имају на располагању, а у другом постављајући индикатор своје активности тако да његови суседи могу да виде да он једе).

→ Коментарисати следеће решење проблема филозофа за ручком: Сваки филозоф узима само парну виљушку. Виљушке су нумерисане.

## **Одговор**

Пошто се не може гарантовати редослед ослобађања вилјушака, односно да ће се икад крај неког од филозофа наћи две парне вилјушке, може доћи до изгладњивања.

## 20

### Анализа различитих варијанти решења проблема читалаца и писаца

Група упоредних процеса који приступају заједничком ресурсу састоји се од читалаца  $Reader_i$  ( $i=1,\dots,m$ ) и писаца  $Writer_i$  ( $i=1,\dots,n$ ). Исправна контрола приступа мора обезбедити међусобно искључивање процеса према уобичајеном правилу за читаоце и писце и спречити узајамно блокирање и изгладњивање. Предложене су различите варијанте решења. У датој таблици свако варијанти одговара по једна колона и у њу треба за свако од наведених тврђења уписати тачно или нетачно.

Тврђења:

а) б) в)

- Међусобно искључивање је осигурено
- Могуће је узајамно блокирање читалаца и писаца
- Могуће је узајамно блокирање писаца при  $r=0$
- Могуће је изгладњивање читалаца
- Могуће је изгладњивање писаца

а)  
**program ReadersWriters;**  
**const** м = ...; н = ...;  
**var** v : shared record r, w : integer **end;**

```
procedure Reader(id : integer);  
begin  
    repeat  
        region v do begin  
            await (w = 0); r := r + 1  
        end;  
        read;  
        region v do r := r - 1;  
        {nekriticne operacije;}  
    until false;  
end;
```

```
procedure Writer(id : integer);  
begin  
    repeat  
        region v do begin  
            w := w + 1; await (r = 0)  
        end;  
        write;  
        region v do w := w - 1;  
        {nekriticne operacije;}  
    until false;  
end;  
begin  
    v.r := 0; v.w := 0;  
    cobegin Reader(1);... Reader(m); Writer(1);... Writer(n); coend  
end.
```

б)

**program** ReadersWriters;

```

const m = ...; n = ...;
var v : shared record r, w : integer end;

procedure Reader(id : integer);
begin
    repeat
        region v do begin
            await (w = 0); r := r + 1
        end;
        read;
        region v do r := r - 1;
        {nekriticne operacije;}
    until false;
end;
procedure Writer(id : integer);
begin
    repeat
        region v do begin
            w := w + 1;
            await ((r = 0) and (w = 1))
        end;
        write;
        region v do w := w - 1;
        {nekriticne operacije;}
    until false;
end;
begin
    v.r := 0; v.w := 0;
    cobegin Reader(1);... Reader(m); Writer(1);... Writer(n); coend
end.

```

Б)

```

program ReadersWriters;
const m = ...; n = ...;
var v : shared record r, w : integer; rturn : boolean end;

procedure Reader(id : integer);
begin
    repeat
        region v do begin
            if rturn then begin
                r := r + 1; await (w = 0)
            end
            else begin
                await (w = 0); r := r + 1
            end
        end;
        read;
        region v do begin
            r := r - 1; rturn := false
        end;
        {nekriticne operacije;}
    until false;
end;

```

```

procedure Writer(id : integer);
begin
repeat
    region v do begin
        if rturn then begin
            await (r = 0); w := w + 1
        end
        else begin
            w := w + 1; await (r = 0)
        end;
    end;
    write;
    region v do begin
        w := w - 1; rturn := true
    end;
    {nekriticne operacije;}
until false;
end;
begin
v.r := 0; v.w := 0; v.rturn := false;
cobegin Reader(1);... Reader(m); Writer(1);... Writer(n); coend
end.

```

## Решење

Прво треба уочити да се у овом примеру *await* наредбе налазе не само на почетку условних критичних региона. Сва решења су са погрешним кодом, пре свега као последица непостојања међусобног искључивања или непоштовања основних правила проблема писаца и читалаца. Упркос нетачном коду, разматрана је свака особина појединачно.

a) Размотрићемо свако тврђење појединачно:

- Читалац не може да чита пре него што сви писци оду (због *await w=0*) и ниједан писац неће почети да пише док има активних читалаца (због *await r=0*), па је међусобно искључивање између читалаца и писаца обезбеђено. Међутим, није обезбеђено међусобно искључивање између писаца; могуће је да више писаца пише истовремено. Дакле, већ на основу ове особине, решење а) није коректно.
  - Читалац мора да чека да сви писци заврше, док писци одмах добијају ресурс (само сачекају да заврше тренутни читаоци). Дакле, нема међусобног чекања, односно међусобног блокирања читалаца и писаца.
  - Ниједан писац не сачекује другог писца при писању, па нема узајамног блокирања писаца.
  - Могуће је да услов *w=0* не буде никад испуњен (ако увек неко пише), па тада читаоци могу да изгладне.
  - Пошто писац чим нађе инкрементира *w*, читаоци не могу стално да читају (због *await w=0*). Писац, када се пријави, добија ресурс чим сви процеси који тренутно читају заврше читање. Значи да је немогуће изгладњивање писаца.
- b) Сада имамо ситуацију да више писаца не може истовремено да пише (због *await w=1*).

1. Читалац не може да чита пре него што сви писци оду (*await w=0*) и ниједан писац неће почети да пише док има активних читалаца (*await r=0*), па је међусобно искључивање између читалаца и писаца обезбеђено. Обезбеђено је и међусобно искључивање између писаца (због *await w=1*).

2. Исто као под а).

3. Може доћи до узајамног блокирања писаца. Кад дође први писац биће *w=1*. Док он још ради може доћи други писац и биће *w=2*, а потом и трећи, па ће бити *w=3*. Кад први заврши биће *w=2* и сада и први и други остају блокирани чекајући да се испуни услов *await w=1*, а *w* више нема ко да декрементира.

4. Исто као под а).

5. Исто као под а).

в) Намера овог алгоритма је да спречи изгладњивање читалаца

1. Обезбеђено је међусобно искључивање читалаца и писаца, али није међусобно искључивање самих писаца.

2. Није могуће узајамно блокирање читалаца и писаца.

3. Није могуће узајамно блокирање писаца јер ни један писац не чека ни једног другог писца.

4. Није могуће изгладњивање читалаца, јер писац кад заврши ради *turn:= true*, а читалац у том случају ради *r:= r+1* и тиме спречава остале писце да више пишу (због *await r =0*). Дакле, читаоци само чекају да текући писци заврше посао и обратно.

5. Исто као под а).

## 21 Претрага-уметање-брисање

Постоје три врсте операција које се могу обављати над једноструком уланчаном листом: претраживање, убаџивање и брисање (*The Search-Insert-Delete Pr*). Претраживање само прегледа листу, тако да се може дозволити да више процеса претражују у паралели. Убаџивање додаје нови елемент на крај листе. Процеси који убаџују су међусобно искључиви, али се убаџивање може радити у паралели са произвољним бројем претраживања. Може се брисати елемент са било које позиције у листи. У једном тренутку може само један процес који брише да приступа листи. Тај процес има ексклузивни приступ листи. Користећи условне критичне регионе решити овај проблем.

### Решење

Овде представљен проблем спада у групу проблема код којих више корисника може да приступа истом ресурсу, али се жели ограничити њихов број у зависности од функције коју обављају над ресурсом. Проблем је тако конципиран да свака од ове три операције има различит ниво ограничења приликом приступа ресурсима који су у овом примеру представљени као једнострука уланчана листа. Уколико се жели претраживање листе довољан услов је да нико не брише елементе из листе, када се ради убаџивања услов је да нико не брише елементе листе нити да ико други покушава да умеће у листу и последњи случај који има највећи степен рестрикције је приликом операције брисања из листе. У том случају се мора остварити да нико не претражује листу, да нико не убаџује нове елементе у листу, нити да ико други покушава да брише елементе из листе.

Листа над којом ће се обављати операције је дефинисана као једнострука уланчана листа код које је информациони садржај целобројна величина.

```
type Element = record
    val : integer;
    next : ^Element;
end;
```

Над овом листом су дефинисане три не синхронизоване процедуре које обављају операције над елементима листе. Те процедуре су: *search(i : integer, var first : ^Element)*, *insert(i : integer, var first : ^Element)* и *delete(i : integer, var first : ^Element)*, које обављају операције претраживања, уметања и брисања, респективно. Сами реализација ових процедуре није битна са ставовишта решење овог проблема јер је комплетна синхронизација измештена на друго место. Приликом реализације ових процедуре уопште не треба водити рачуна о синхронизацији. Како би се омогућио приступ листи већем броју конкурентних процеса уместо ових процедуре приступ листи је дозвољен кроз три процедуре које омотавају дате процедуре, а које обављају комплетну синхронизацију. Трећи процедуре за синхронизацију су: *SearchW(i : integer)*, *InsertW(i : integer)* и *DeleteW(i : integer)* које као аргумент добијају информације неопходне за приступ самој листи.

На почетку сваке од ових процедуре за синхронизацију налази се приступ објекту над којим се ради синхронизација. То је објекат који се понаша као условни критични регион и који води евидентију о броју процеса који претражују, умећу и бришу по листи, польз *numS*, *numI* и *numD*, респективно. Свака процедура за синхронизацију је реализована из три дела. У првом делу се приступа региону, када се уђе у регион проверава се услов за обављање дате операције над листом, када услов за приступ постане испуњен увећава се бројач који каже колико процеса које врсте приступа листи

и напушта се регион. Потошто је добијена дозвола за приступ листи прелази се на други део где се обављају операције над листом. Када се заврше операције над листом прелази се на трећи корак, а то је ажурирање, то јест умањивање за један, броја послова дате врсте коју приступају листи. Треба приметити да решење код кога би целокупни приступ листи био реализован унутар региона не би било доволно конкурентно, јер би се на тај начин ограничио број процеса који се извршавају на 1, чиме би се смањила конкурентност целог програма.

Услов под којима се дозвољава приступ листи када се ради о операцији претраживања је: `await (b.numD == 0)`. Може се приметити да је претраживање дозвољено у свим случајеви осим када неко брише елементе листе. Није неопходно радити проверу колико има оних коју умећу или преосталих који претражују листу. Услов под којима се дозвољава приступ листи када се ради о операцији убацивања новог елемента у листу је: `await (((b.numI == 0) and (b.numD == 0)))`. Приликом уметања неког елемента треба водити рачуна о томе да нико други не модификује листу. Модификација листе се обавља на два начина или операцијом брисања или другом операцијом уметања. Услов под којима се дозвољава приступ листи када се ради о операцији брисања неког елемента листе је: `await(((b.numS == 0) and (b.numI == 0) and (b.numD == 0)))`. Приликом брисања елемента треба водити рачуна о томе да нико други не приступа листи.

```

program SID;
const N = ...;
type Element = record
    val : integer;
    next : ^Element;
  end;
  Barrier = record
    numS, numI, numD: integer;
  end;
var b: shared Barrier;
  first: ^Element;

procedure SearchW(i : integer);
begin
  region b do
    begin
      await (b.numD = 0);
      b.numS := b.numS + 1
    end;
    search(i, first);
    region b do
      b.numS := b.numS - 1;
    end;
procedure InsertW(i : integer);
begin
  region b do
    begin
      await ((b.numI = 0) and (b.numD = 0));
      b.numI := b.numI + 1;
    end;
    insert(i, first);
    region b do
      b.numI := b.numI - 1
    end;
end;
```

```
procedure DeleteW(i : integer);
begin
    region b do
    begin
        await ((b.numS = 0) and (b.numI = 0) and (b.numD = 0));
        b.numD := b.numD + 1;
    end;
    delete(i, first);
    region b do
        b.numD := b.numD - 1
    end;
procedure User(id : integer);
var i : integer;
begin
    ...
    SearchW(i);
    ...
    InsertW(i);
    ...
    DeleteW(i);
    ...
end;
begin
    b.numS := 0;
    b.numI := 0;
    b.numD := 0;
    cobegin
        User(0);
        ...
        User(N);
    coend
end.
```

Овако написано решење може да доведе до изгладњивања неког од процеса. Најпре може доћи до изглављивања процеса који раде брисање елемената листе јер они чекају да сви претходни процеси заврше. Да би се смањиле последице оваквог понашања потребно је урадити сличну ствар као код проблема преласка моста који има једну коловозну траку. Код тог решења је узето да сваки пут када се констатује да постоји процес који је блокиран дуже од неког задатог интервала времена, односно броја приступа њему се препусти право да ради са листом.

#### Друго решење:

Пошто број је процеса који конкурентно приступа листи приликом операције брисања само један, претходно решење се може модификовати тако да се смањи број услова који се проверавају тако што се уведе да процес који ради операцију брисања само брисање обавља унутар условног критичног региона.

```
program SID;
const N = ...;
type Element = record
    val : integer;
    next : ^Element;
end;
Barrier = record
    numS, numI, numD: integer;
```

```

    end;
var b: shared Barrier;
    first: ^Element;

procedure SearchW(i : integer);
begin
    region b do
    begin
        b.numS := b.numS + 1
    end;
    search(i, first);
    region b do
        b.numS := b.numS - 1;
end;
procedure InsertW(i : integer);
begin
    region b do
    begin
        await (b.numI = 0);
        b.numI := b.numI + 1;
    end;
    insert(i, first);
    region b do
        b.numI := b.numI - 1
end;

procedure DeleteW(i : integer);
begin
    region b do
    begin
        await ((b.numS = 0) and (b.numI = 0));
        delete(i, first);
    end;
end;
procedure User(id : integer); begin ... end;
begin
    b.numS := 0;
    b.numI := 0;
    b.numD := 0;
    cobegin
        User(0);
        ...
        User(N);
    coend
end.

```

**Треће решење**

У случају да је потребно гарантовати редослед по коме се приступа листи може се применити тикет алгоритам. Код овог алгоритма на почетку сви процеси који приступају листи узимају јединствен редни број који каже који су они по редоследу пристизања (локална променљива *turn* добија вредност *b.number*), и у истом блоку увећавају овај редни број како би следећи процес имао исправну вредност (*b.number:=b.number+1*). Након тога као додатни услов приликом приступа листи процеси треба да сачекају да редни број који су добили на почетку постане једна редном броју следећег који сме да

приступи листи (*and (turn = b.next)*). Потребно је само још одредити када који процес сме да увећава редни број за приступ листи (*b.next:=b.next +1*). Приликом претраживања се одмах може увећати овај број и доделити приступ листи првом следећем, приликом уметања такође, док се код брисања ова променљива увећава тек када се операција брисања заврши.

```

program SID;
const N = ...;
type ...
    Barrier = record
        numS, numI, numD: integer;
        next, number: integer;
    end;
var b: Barrier;
    first: ^Element;
...
procedure SearchW(i : integer);
var turn : integer;
begin
    region b do
    begin
        turn := b.number;
        b.number := b.number + 1;
        await ((b.numD = 0) and (turn = b.next));
        b.numS := b.numS + 1;
        b.next := b.next + 1
    end;
    search(i, first);
    region b do
        b.numS := b.numS - 1;
    end;
procedure InsertW(i : integer);
var turn : integer;
begin
    region b do
    begin
        turn := b.number;
        b.number := b.number + 1;
        await ((b.numI=0) and (b.numD=0) and (turn=b.next));
        b.numI := b.numI + 1;
        b.next := b.next + 1
    end;
    insert(i, first);
    region b do
        b.numI := b.numI - 1
    end;
procedure DeleteW(i : integer);
var turn : integer;
begin
    region b do
    begin
        turn := b.number;
        b.number := b.number + 1;
    end;

```

```

    await ((b.numS = 0) and (b.numI = 0) and (b.numD = 0)
           and (turn = b.next));
    b.numD := b.numD + 1;
end;
delete(i, first);
region b do
    b.numD := b.numD - 1;
    b.next := b.next + 1
end;
procedure User(id : integer); begin ... end;
begin
    b.numS := 0;
    b.numI := 0;
    b.numD := 0;
    b.number := 0;
    b.next := 0;
    cobegin
        User(0);
        ...
        User(N);
    coend
end.

```

Треба приметити да се код операције брисања увећавање променљиве *next* може обавити и након обављене операције брисања, али и одмах како код претраживања и уметања. Уколико се код операције брисања постави да се увећавање обавља одмах онда је неопходно да остане променљива о броју тренутно активних брисања, *numD*, како би се обезбедио исправан рад. Уколико се увећавање ове променљиве обави тек након операције брисања из листе онда нема потребе за променљивом *numD* нити испитивањем услова над овом променљивом.

## 22

## Дељени рачун

Рачун у банци може да дели више корисника (*The Savings Account Problem*). Сваки корисник може да уплаћује и подиже новац са рачуна под условом да салдо на рачуну никада не буде негативан, као и да види тренутно стање рачуна. Решити проблем користећи условне критичне регионе.

### Решење

Ово је један од проблема код кога више корисника може да приступа истом ресурсу, али се жели ограничити да само један може у неком тренутку радити промену над тим ресурсом, уз ограничење каква промена може да буде. У овом проблему је узето да промена може да се изврши над стањем неког рачуна само у случају да након обављене трансакције стање рачуна остане позитивно. У ситуацији да нема довољно новца на рачуну могу се разматрати две варијанте проблема, једна варијанта је да корисничка трансакција чека све док на рачуну не буде било довољно новца, а друга варијанта је да се трансакција прекине и о томе одмах обавести корисник. Трансакција се у сваком случају прекида ако корисник не сме да приступа одабраном рачуну.

Код овог решења разматра се ситуација када корисник може да уплаћује и да подиже новац са рачуна као и да посматра стање рачуна користећи три функције: *deposit*, *withdraw* и *status*, респективно. Корисник не приступа директно рачунима већ се комплетна обрада остварује користећи функције. Ово решење је тако реализовано да се комплетна синхронизација обавља унутар функција тако да о томе није потребно водити рачуна у клијентском коду. Функције *deposit* и *withdraw* као повратну информацију враћају да ли је трансакција успела, а функција *status* враћа тренутно стање рачуна уколико трансакција успе односно -1 у супротном.

Прва функција, *function deposit(UID, num : integer; funds : real) : boolean*, служи за уплаћивање новца на рачун корисника. Први аргумент *UID* представља идентификацију корисника, други аргумент, *num*, представља рачун на који је потребно уплатити новац у банци, док трећи аргумент представља вредност која се жели уплатити. Повратна информација говори о томе да ли је трансакција успела. Новац на рачун може бити уплаћен само у случају да се ради о позитивном износу уплате. Када се провери да ли је сума новца која се уплаћује позитивна прелази се на операције рада са банком. Да би се приступило рачунима овде није потребно закључати приступ целој банци већ само специфицираном рачуну. Ово се постиже проглашавањем рачуна за критични регион (*region accounts[num]*). Када се обезбеди неометан приступ рачуну приступа се формалној провери да ли корисник сме да приступа датом рачуну, ово је остварено користећи функцију (*isInList(accounts[num], UID)*) која враћа информацију о повезаности корисника и рачуна. Сама имплементација претраживања није од суштинског значаја за решавање овог проблема па зато није ни дата. Уколико корисник сме да приступа рачуну уплата се остварује. Треба приметити да овде није неопходно радити блокирање док се услов не оствари јер уплаћивање позитивне суме новца на рачун не може нарушити правило о недозвољеном негативном салду на рачуну. Када се уплата заврши функција враћа повратну информацију да ли је уплата остварена или не. Уплата је могла да се обустави или зато што је покушано да се уплаћује негативни износ или зато што корисник не сме да приступа одговарајућем рачуну.

Друга функција, *function withdraw(UID, num : integer; funds : real) : boolean*, служи за подизање новца са рачуна корисника. Први аргумент *UID* представља идентификацију корисника, други аргумент, *num*, представља рачун са кога је потребно подићи новац,

док трећи аргумент представља вредност која се жели подићи. Повратна информација говори о томе да ли је трансакција успела. Новац се са рачуна може подићи само у случају да се ради о позитивном износу. На почетку функције се проверава да ли је сума новца која се подиже позитивна, па се тек након тога прелази на операције над самим рачуном у банци. Ни овде није потребно закључавати цели банку ради приступа једном рачуну, већ је довољно закључати тај један рачун користећи регион формиран над њим. За проверу права приступа и овде је искоришћена функција *isInList(accounts[num], UID)*. Уколико корисник сме да приступа рачуну прелази се на део за подизање новца. Овде је потребно блокирати корисника све док се на рачуну на појави довољна количина новца да би салдо остао позитиван. Чекање је остварено користећи *await(saldo >= funds)*. Тек када се на рачуну појави довољно новца прелази се на његово подизање. У првој варијанти проблема, овде је могуће направити и резервације новца тако да се прво опслужују они који више чекају. Када се подизање новца заврши функција враћа повратну информацију да ли је трансакција остварена или не.

Трећа функција, *function status(UID, num : integer) : real;*, служи за проверу стања новца на рачуну корисника. Први аргумент *UID* представља идентификацију корисника, други аргумент, *num*, представља рачун чије стање се испитује. Повратна информација говори о стању на рачуну уколико је трансакција успела односно враћа вредност -1 уколико корисник нема право приступа датом рачуну. За проверу права приступа и овде је искоришћена функција *isInList(accounts[num], UID)*. Уколико корисник сме да приступа рачуну прелази се на део за проверу салда. Нема блокирања корисника јер увид у стање рачуна не нарушава правило о недозвољеном негативном салду на рачуну. Уколико корисник није имао привилегија да приступи рачуну враћа се вредност -1 која јединствено сигнализира ову ситуацију јер стање рачуна никако другачије не би могло да буде негативно.

```

program SavingsAccount;
const MAXUSERS = ...;
NUMOFACCOUNTS = ...;
type account = record
    saldo : real;
    users : array[1..MAXUSERS] of integer;
end;

var
accounts : array [1..NUMOFACCOUNTS] of account;
function isInList(userAccount : account; UID : integer) : boolean;
begin ... end;

function deposit(UID, num : integer; funds : real) : boolean;
var status : boolean;
begin
    status := false;
    if (funds >= 0) then
begin
    region accounts[num] do
begin
        if(isInList(accounts[num], UID)) then
begin
            saldo := saldo + funds;
            status := true;
end;
end;
end;
end;

```

```
        end;
    end;
    deposit := status;
end;

function withdraw (UID, num : integer; funds : real) : boolean;
var status : boolean;
begin
    status := false;
    if (funds >= 0) then
begin
    region accounts[num] do
begin
    if(isInList(accounts[num], UID)) then
begin
        await(saldo >= funds);
        saldo := saldo - funds;
        status := true;
    end;
end;
end;
    withdraw := status;
end;

function status(UID, num : integer) : real;
begin
    region accounts[num] do
begin
    if(isInList(accounts[num], UID)) then
        status := saldo
    else
        status := -1;
end;
end;

begin
    cobegin
    ...
    coend;
end.
```

Друга функција, *function withdraw(UID, num : integer; funds : real) : boolean;*, се може реализовати сходно другој варијанти овог проблема и тако да се корисник не блокира него да се врати повратна информација о томе да трансакција није успела ни у случају да на рачуну нема довољно новца. У том случају код ове функције би био:

```
function withdraw (UID, num : integer; funds : real) : boolean;
var status : boolean;
begin
    status := false;
    if (funds >= 0) then
begin
    region accounts[num] do
begin
    if(isInList(accounts[num], UID)) then
begin
```

```
if(saldo >= funds) then
begin
    saldo := saldo - funds;
    status := true;
end
else
    status := false;
end;
end;
withdraw := status;
end;
```

## 23

### Нервозни пушачи

Користећи условне критичне регионе написати програм који решава проблем и симулира систем „нервозних пушача“ (*The Cigarette Smokers' Problem*). Постоји један агент и три нервозна пушача. Агент поседује резерве три неопходна предмета за лечење нервозе: папир, дуван и шибице. Један од пушача има бесконачне залихе папира, други – дувана, а трећи – шибица. Агент почиње тако што два различита предмета ставља на сто, један по један. Пушач, коме баш та два предмета недостају, узима их, завија и пали цигарету и ужива. Након тога обавештава агента да је завршио, а агент онда ставља два нова предмета на сто, итд.

### Решење

Ово је тип проблема код кога постоји више парова процеса који међусобно треба да се такмиче за расположиви ресурс, а да притом нису сви ресурс доступни. Приликом провере мора се остварити да процес који проверава који су све тренутно расположиви ресурси не буде ометан од стране других процеса. Овде је битно да се комплетна провера обави недељиво јер би у супротном могло доћи до погрешног израчунавања расположивости свих неопходних ресурса.

Сам проблем има четири учесника: агента који периодично ставља артикли на сто, и три пушача тако креираних да сваком недостају по два артикла за даљи рад. Процедуре за сваког од ова четири учесника се извршавају у бесконачним петљама.

Како би се проблем решио користећи условне критичне регионе треба дефинисати структуру података, запис, у којој је могуће чувати све дељене податке о раду система. Од података је неопходно поставити шта се све налази на столу, као и да ли је операција узимања са стола завршена. Одговарајућа структура података би имала следећи облик:

```
type table = record
    paper, tobacco, matches : boolean;
    ok : boolean;
end;
```

Може се приметити да је присуство сваког од артикула представљен једном променљивом (*paper*, *tobacco*, *matches*). Решење ког кога би се пар артикула означио једном вредношћу не би било у духу овог проблема код кога је појента да се на сто стављају артикли један по један, а не као пар. Разлог за ово је да се опише систем код кога процес који производи ресурсе, агент, не зnam коме је шта упућено.

Агент треба да постави неки пар артикула на сто. Да би се симулирала равноправност сваке комбинације може се искористити генератор случајних бројева (*random(2)*) који генерише целобројну вредност из интервала од 0 до 2. Ове три вредности су доволно да се представе сви парови артикула који се постављају на сто. Треба још једном напоменути да је потребно поставити један по један артикал на сто а не један који обједињује њихову функцију. Пошто је сто на који се стављају артикли представљен помоћу датог записа (*table*) потребно је поставити свако од поља које говоре о томе шта се налази на столу на одговарајућу вредност (*paper*, *tobacco*, *matches*). Пошто се поставља променљива којој могу да приступају и коју могу да мењају сви процеси неопходно је учинити приступ овој променљиви недељивом операцијом. То се постигло проглашавањем дате променљиве за регион чиме је гарантована јединственост приступа критичним операцијама. Када агент добије приступ региону он поставља артикле на сто и након тога прелази на фазу чекања да неки од пушача заврши са

коришћењем постављених артикала. Чекање на овом услову је реализовано користећи наредбу *await* којом агент проверава да ли је неко променио стање региона, и уколико услов није испуњен напушта експлузивно право коришћења региона и наставља са чекањем. Услов на који агент чека је *await(p.ok)*, то јест чека на потврду да је конзумирање завршено. Када добије потврду и пређе *await* услов агент поново има експлузивно право приступа региону и он то користи да би себи, за следећу итерацију сигнализирао да конзумирање није завршено, ово се постиже постављањем поља *p.ok* на вредност *false*;

Сваки од три пушача има идентичну структуру, само се разликује скуп артикала на којима чекају. Овде ће бити објашњено понашање за само једног од њих док се понашање за преостале добија аналогијом. Процедура за пушача који поседује неограничену количину шибица (*SmokerWithMatches*) има следећу структуру: прво се приступа региону, онда се чека да се појаве одговарајући артикли, након тога се ти артикли користе и на крају се агент обавештава о завршеној операцији. Пушач на почетку долази до региона узима експлузивно право на његово коришћење и креће на испитивање услова расположивости артикала. Уколико услов (*p.paper and p.tobacco*) није испуњен пушач се одриче експлузивног права коришћења региона све док тај услов не постане испуњен. Када услов постане испуњен пушач троши расположиве артикле и поставља их на неактивне вредности (*false*). Када их је покупио прелази на фазу коришћења ових артикала. Треба констатовати да се ништа не би променило и да пушач није напустио коришћење региона. Ово не би смањило конкурентост јер у једном тренутку само један од ова четири процеса и може да се извршава. Када пушач заврши са коришћењем артикала он обавештава агента да је завршио, а сам напушта експлузивно право коришћења артикала до следеће итерације.

Заједничке променљиве које се користе приликом решавања овог проблема су смештене унутар региона *p* чија структура раније наведена. Поља која означавају расположивост артикала (*paper*, *tobacco* и *matches*) на почетку су постављена на неактивне вредности (*false*), што такође мора бити урађено и са условом који сигнализира крај операције коришћења артикала.

```

program CigaretteSmokers;
type table = record
    paper, tobacco, matches : boolean;
    ok : boolean;
  end;
var p: shared table;

procedure Agent;
var n : integer;
begin
  while (true) do
  begin
    n := produce;//random(2)
    region p do
    begin
      case n of
        0: begin
          p.paper := false;
          p.tobacco := true;
          p.matches := true;
        end;
        1: begin
          p.paper := true;
        end;
      end;
    end;
  end;
end;

```

```
        p.tobacco := false;
        p.matches := true;
    end;
    2: begin
        p.paper := true;
        p.tobacco := true;
        p.matches := false;
    end;
    else ;
    end;
    await(p.ok);
    p.ok := false;
end;
end;
procedure SmokerWithMatches;
begin
    while (true) do
    begin;
        region p do
        begin
            await(p.paper and p.tobacco);
            p.paper := false;
            p.tobacco := false;
        end;
        enjoy;
        region p do
            p.ok := true;
        end;
    end;
procedure SmokerWithTobacco;
begin
    while (true) do
    begin
        region p do
        begin
            await(p.paper and p.matches);
            p.paper := false;
            p.matches := false;
        end;
        enjoy;
        region p do
            p.ok := true;
        end;
    end;
end;
procedure SmokerWithPaper;
begin
    while (true) do
    begin
        region p do
        begin
            await(p.matches and p.tobacco);
            p.matches := false;
        end;
    end;

```

```

    p.tobacco := false;
end;
enjoy;
region p do
    p.ok := true;
end;
end;

begin
    p.paper := false;
    p.tobacco := false;
    p.matches := false;
    p.ok := false;
cobegin
    Agent;
    SmokerWithPaper;
    SmokerWithTobacco;
    SmokerWithMatches;
coend;
end.

```

Уколико би било дозвољено да се формирање новог скупа артикала који је потребно ставити на сто обављало у паралели са коришћењем артикала из претходне итерације могла би се постићи већа конкурентност. У том случају би се једино код за агента и за иницијализацију променио, док би код за пушаче остао непромењен. Агент би у том случају био:

```

procedure Agent;
var n : integer;
begin
    while (true) do
begin
    n := produce;//random(2)
    region p do
    begin
        await(p.ok);
        p.ok := false;
        case n of
            0: begin
                p.paper := false;
                p.tobacco := true;
                p.matches := true;
            end;
            1: begin
                p.paper := true;
                p.tobacco := false;
                p.matches := true;
            end;
            2: begin
                p.paper := true;
                p.tobacco := true;
                p.matches := false;
            end;
        else ;
        end;
    end;
end;

```

```
        end;
    end;
end;
```

Код за иницијализацију би се морао променити јер уколико би остао исти (*p.ok := false*) систем не би могао ни да започне са радом. Да би се дозволила почетна итерација агента потребно би било уместо *p.ok := false* поставити *p.ok := true* у иницијализационом делу који би на тај начин постао:

```
begin
    p.paper := false;
    p.tobacco := false;
    p.matches := false;
    p.ok := true;
    cobegin
        Agent;
        SmokerWithPaper;
        SmokerWithTobacco;
        SmokerWithMatches;
    coend;
end.
```

Систем код кога агент ставља два артикала на сто један по један се уместо коришћења генератора случајних бројева може мало прикладније описати увођењем функције за производњу која производи један по један нов артикал који се ставља на сто. Агент позива први пут функцију и добија један артикал који ставља на сто, онда позива поново ову функцију и добија други артикал који такође ставља на сто. Ову функцију треба тако реализовати да не генерише два пута заредом исту вредност.

```
procedure Agent;
begin
    while (true) do
        begin
            put(produce);
            put(produce);
            region p do
                begin
                    await(p.ok);
                    p.ok := false;
                end;
            end;
        end;
    end;

procedure put(n : integer);
begin
    region p do
        begin
            case n of
                0: begin
                    p.matches := true;
                end;
                1: begin
                    p.paper := true;
                end;
            end;
        end;
    end;
```

```
2: begin
    p.tobacco := true;
end;
else ;
end;
end;
end;
function produce; begin ... end;
```

## 24 Проблем избора

Користећи условне критичне регионе написати програм који решава следећи проблем: Постоје три особе међу којима треба изабрати једну (*The Odd Person Wins Game*). Свака од тих особа поседује новчић који има две стране. Избор особе се одиграва тако што свака особа независно баца свој новчић. Уколико постоји особа којој је новчић пао на другу страну у односу на преостале особе онда се та особа изабира. Уколико све особе имају исто постављен новчић поступак се понавља све док се не изабере једна. Особе на крају сваке итерације морају да знају да ли су изабране или не или се поступак понавља.

### Решење

#### Прво решење

Проблем избора је проблем синхронизације на баријери код кога увек исти процеси приступају баријери и број приступа баријери није унапред познат. Приступ се обавља у већем броју итерација у зависности од резултата бацања новчића. Као и пре ово решење се заснива на технички коришћења „преводнице“ (*lock*) са двоја врата.

Пошто сви процеси учествују у свакој итерацији прва, улазна, врата су отворена и процес долази, поставља свој статус и увећава број оних који су прошли кроз прва врата. Ако је то последњи процес који је поставил свој статус онда отвара друга врата, поставља број који су прошли кроз друга врата на 0. Сви процеси чекај док број процеса који су дошли и поставили свој статус не постане једнак броју процеса који треба да поставе свој статус. Након тога процес чита преостале статусе и самостално ван региона генерише резултат извршавања.

Пошто су сви процеси сигурно већ поставили свој статус друга, излазна, врата су отворена и процес увећава број оних који су прошли кроз друга врата, ако је то последњи процес онда отвара прва врата, поставља број који су прошли кроз прва врата на 0. Сви процеси чекај док број процеса који су довде дошли не постане једнак броју процеса. Након тога се евентуално прелази на наредну итерацију извршавања.

Треба приметити да коришћењем само једне бројачке променљиве како би се реализовала баријера, као што је то био случај код семафора, може довести до погрешног рачунања резултата или до блокирања свих процеса.

```
procedure Person_with_deadlock1(id : 0..2);
var status, winner, again, leftNeighbour, rightNeighbour : boolean;
begin
    winner := false;
    again := true;
    while again do
    begin
        status := flipCoin;
        region result do
        begin
            coins[id] := status;
            cnt := cnt + 1;
            await(cnt = 3);
            cnt := cnt - 1;
        end;
    end;
end;
```

```

    leftNeighbour := coins[(id + 2) mod 3];
    rightNeighbour := coins[(id + 1) mod 3];
end;
    winner := (status xor leftNeighbour)
        and (status xor rightNeighbour);
    again := not ((status xor leftNeighbour)
        or (status xor rightNeighbour));
end;
end;

```

У случају да новчићи тако пали да се не ради о последњој итерацији онда када први процес прође кроз овакву баријеру, срачунава услов и пошто је услов такав генерише нову вредност и поново долази на исту баријеру. Пошто ни један од преосталих процеса у међувремену није могао да прође услов (*cnt* је био 2), овај процес сада поставља свој нови статус и увећава број присутних на 3. Пошто је услов за пролазак испуњен (*await(cnt = 3)*) онда овај или неки други процес проплазе услов и читају статусе који нису конзистентни јер припадају различитим итерацијама. У случају да се ради о последњој итерацији само први процес проплази кроз овакву баријеру, док остали процеси остају заувек блокирани на њој. Први процес који прође услов (*await(cnt = 3)*) тај услов одмах промени (*cnt := cnt - 1*), а како се ради о последњој итерацији нема ко други тај услов да промени и остали процеси остају блокирани (*cnt* је сада 2).

Овде треба приметити да уколико би се раздвојио приступ региону на два дела, први у коме би се бројач увећавао и чекало док не пристигне три процеса и други у коме би се бројач умањивао опет не би гарантовало исправност решења. И даље би било могуће да први процес који у првом делу установи да су сви пристигли, напусти регион и обави рачунање и уђе у други део пре него што остали процеси напусте први део. На овај нечим тај процес умањује бројач присутних процеса што опет може да доведе или до погрешног рачунања коришћењем вредности из више итерација или блокирања осталих процеса.

```

procedure Person_with_deadlock2(id : 0..2);
var status, winner, again, leftNeighbour, rightNeighbour : boolean;
begin
    winner := false;
    again := true;
    while again do
    begin
        status := flipCoin;
        region result do
        begin
            coins[id] := status;
            cnt := cnt + 1;
            await(cnt = 3);
            leftNeighbour := coins[(id + 2) mod 3];
            rightNeighbour := coins[(id + 1) mod 3];
        end;
        ...
        region result do
        begin
            cnt := cnt - 1;
        end;
    end;

```

Како би се отклонило читање вредности из више итерација потребно је обезбедити да су сви процеси прочитали стару вредност пре него што крену са генерирањем нове вредности. Ово би се мого остварити додавање новог услова на коме би процеси чекали док број процеса који су прошли израчунавање не постане нула. Треба приметити да увођење само овога би допринело избегавању узајамног блокирања уколико један процес може да обави израчунавање пре него што преостали процеси прођу први услов.

```
procedure Person_with_deadlock3(id : 0..2);
var status, winner, again, leftNeighbour, rightNeighbour : boolean;
begin
  ...
  while again do
  begin
    status := flipCoin;
    region result do
    begin
      coins[id] := status;
      cnt := cnt + 1;
      await(cnt = 3);
      leftNeighbour := coins[(id + 2) mod 3];
      rightNeighbour := coins[(id + 1) mod 3];
    end;
    ...
    region result do
    begin
      cnt := cnt - 1;
      await(cnt = 0);
    end;
  end;
end;
```

Решење овог проблема је постигнуто увођењем две бројачка променљиве. Једне за оне који су прошли дошли до првог услова и друге оних који су дошли до другог услова. Остаје само да се одреди када ће се ове променљиве умањивати, односно враћати на вредност 0.

```
program OddPerson;
type results = record
  generated, calculated : integer;
  coins : array [0..2] of boolean;
end;
var result : results;
procedure Person(id : 0..2);
var status, winner, again, leftNeighbour, rightNeighbour : boolean;
begin
  winner := false;
  again := true;
  while again do
  begin
    status := flipCoin;
    region result do
    begin
      coins[id] := status;
```

```

generated := generated + 1;
if(generated = 3) then calculated := 0;
await(generated = 3);
leftNeighbour := coins[(id + 2) mod 3];
rightNeighbour := coins[(id + 1) mod 3];
end;
winner := (status xor leftNeighbour)
and (status xor rightNeighbour);
again := not ((status xor leftNeighbour)
or (status xor rightNeighbour));
region result do
begin
calculated := calculated + 1;
if(calculated = 3) then generated := 0;
await(calculated = 3);
end;
end;
function flipCoin; boolean; begin ... end;
begin
result.generated := 0;
result.calculated := 0;
result.coins[0] := false;
result.coins[1] := false;
result.coins[2] := false;
cobegin
Person(0);
Person(1);
Person(2);
coend;
end;

```

Овде треба приметити да уколико је дозвољен приступ променљивама региона и ван самог региона онда би се читање статуса суседа могло обавити и ван региона и на тај начин омогућило да се регион раније напусти.

#### Друго решење

```

program OddPerson;
type results = record
    generated, generatedC, calculated, calculatedC : integer;
    coins : array [0..2] of boolean;
end;
var result : results;
procedure Person(id : 0..2);
var status, winner, again, leftNeighbour, rightNeighbour : boolean;
begin
    winner := false;
    again := true;
    while again do
begin
    status := flipCoin;
    region result do
begin

```

```
coins[id] := status;
generated := generated + 1;
generatedC := generatedC + 1;
await(generated = 3);
generatedC := generatedC - 1;
if(generatedC = 0) then generated := 0;
leftNeighbour := coins[(id + 2) mod 3];
rightNeighbour := coins[(id + 1) mod 3];
end;
winner := (status xor leftNeighbour)
and (status xor rightNeighbour);
again := not ((status xor leftNeighbour)
or (status xor rightNeighbour));
region result do
begin
    calculated := calculated + 1;
    calculatedC := calculatedC + 1;
    await(calculated = 3);
    calculatedC := calculatedC - 1;
    if(calculatedC = 0) then calculated := 0;
end;
end;
function flipCoin: boolean;
begin
    if random(2) = 1 then flipCoin := true
    else flipCoin := false;
end;
begin

    result.generated := 0;
    result.generatedC := 0;
    result.calculated := 0;
    result.calculatedC := 0;
    result.coins[0] := false;
    result.coins[1] := false;
    result.coins[2] := false;
    cobegin
        Person(0);
        Person(1);
        Person(2);
    coend;
end.
```

## 25 Проблем паркинга са заглављивањем

Постоје два паркинг места, при чему се возило паркирано на паркинг месту бр. 1 бива заглављено ако се попуни паркинг место бр. 2. Потребно је контролисати ова два паркинг места по следећој логици:

- 1) процеси аутомобили позивају процедуру *zelim\_da\_se\_parkiram* и паркирају се ако је неко од места слободно. У супротном чекају на ослобађање места за паркинг. Ако је паркинг празан, обавезно се паркирају на место 1.
- 2) Када желе да напусте паркинг позивају процедуру *zelim\_da\_napustim\_parking*
- 3) Ако су паркирани на месту бр. 1, морају да чекају ослобађање места бр. 2, да би отишли са паркинга.

### Решење

Аутомобил може да се паркира ако је број аутомобила који су већ паркирани на паркингу мањи од 3, *await(num < 3)*. Када се утврди да је овај број мањи од 3 аутомобил се паркира на следеће место на паркингу уписивање свог идентификатора на то место, *ids[num] := id*, и увећавањем броја присутних аутомобила, *num := num + 1*.

Аутомобил може да напусти паркинг ако се налази на паркинг месту број 2, или уколико се на паркинг месту не налази нико, *await((lots.ids[2] = free) or (lots.ids[2] = id))*. Када услов постане испуњен аутомобил умањује броја присутних аутомобила, *num := num - 1*, и поставља да је последње паркинг место, *ids[num] := free*. Треба приметити да је овај паркинг реализован по LIFO редоследу.

```

program Parking;
const free = 0;
type ParkingLots = record
  num : integer;
  ids : array [1..2] of integer;
end;
var lots : shared ParkingLots;

procedure zelim_da_se_parkiram(id : integer);
begin
  region lots do begin
    await(lots.num < 3);
    lots.ids[lots.num] := id;
    lots.num := lots.num + 1;
  end;
end;

procedure zelim_da_napustim_parking (id : integer);
begin
  region lots do begin
    await((lots.ids[2] = free) or (lots.ids[2] = id));
    lots.num := lots.num - 1;
    lots.ids[lots.num] := free;
  end;
end;

```

```
procedure Car(id : integer);
begin
    ...
    zelim_da_se_parkiram(id);
    ...
    zelim_da_napustim_parking(id);
    ...
end;
begin
    lots.num := 1;
    lots.ids[1] := free;
    lots.ids[2] := free;
    cobegin
        Car(1);
        Car(2);
    ...
coend
end.
```

Недостатак овог решења је што не води рачуна о праведности код напуштања ресурса. Ако аутомобил на позицији 1 жели да напусти паркинг место може се десити да стално долазе и одлазе аутомобили на позицију 2 и да никада не дође на ред да оди. Ово се може десити у случају да систем увек даје предност онима који проверавају услов за долазак у односу на оне који проверавају услов за одлазак. Како би се овај недостатак решио може се увести променљива која означава да ли неко жели да напусти паркинг, *leave*. Аутомобил који напуштања паркинга аутомобил прво означи да жели да напусти паркинг, *leave* := *true*, па онда чека на услову, и на крају означи да је напустио паркинг, *leave* := *false*. Аутомобил може да се паркира ако има места али и нико не чека да напусти паркинг, *await*((*num* < 3) and (*not leave*)). Треба приметити да је овде довољно да променљива која означава да неко жели да напусти паркинг буде логичког типа, јер аутомобил на позицији 2 увек може да напусти паркинг док аутомобил на позицији 1 једини може да чека. Уколико би се разматрао паркинг већег капацитета требало би користити низове за означавање који све аутомобили желе да напусте паркинг.

```
program Parking;
const free = 0;
type ParkingLots = record
    num : integer;
    ids : array [1..2] of integer;
    leave : boolean;
end;

var lots : shared ParkingLots;

procedure zelim_da_se_parkiram(id : integer);
begin
    region lots do begin
        await((lots.num < 3) and (not lots.leave));
        lots.ids[lots.num] := id;
        lots.num := lots.num + 1;
    end;
end;
```

```
procedure zelim_da_napustim_parking (id : integer);
begin
    region lots do begin
        lots.leave := true;
        await((lots.ids[2] = free) or (lots.ids[2] = id));
        lots.leave := false;
        lots.num := lots.num - 1;
        lots.ids[lots.num] := free;
    end;
end;

procedure Car(id : integer); begin ... end;
begin
    lots.num := 1;
    lots.ids[1] := free;
    lots.ids[2] := free;
    lots.leave := false;
    cobegin
        Car(1);
        Car(2);
        ...
    coend
end.
```

## Монитори

Монитор је скуп сталних (статичких) променљивих које служе за памћење стања неког ресурса и одговарајућих процедура за имплементацију операција над ресурсом, односно сталним променљивама. Приступ сталним променљивама је могућ само преко процедура монитора. Сталне променљиве задржавају вредност између два позива мониторских процедура, док се вредности локалних променљивих унутар мониторских процедура не памте. Синтаксно и семантички, монитор подсећа на објекат неке класе. Међутим, монитор има једну кључну особину која га разликује од обичног објекта - мониторске процедуре се извршавају са ексклузивним правом приступа над сталним променљивама монитора, чиме је обезбеђено њихово међусобно искључивање.

Условна синхронизација код монитора постиже се операцијама *signal* и *wait* на условној променљивој. Уз помоћ условне променљиве се постиже блокирање процеса који је позвао мониторску процедуру, ако стање монитора не задовољава неки логички (*boolean*) услов. Преко условних променљивих се реализује ред (*queue*) блокираних процеса, који нису директно доступни програмеру.

Ако је *cond* условна променљива унутар монитора, она мора бити декларисана као стална променљива. Блокирање процеса се остварује уз помоћ операције *cond.wait*, чиме се процес суспендира и одриче ексклузивног права приступа у монитору. Тада суспендовани процес може бити деблокиран када неки други процес уради *cond.signal*, или тек када процес дође на ред, јер на основу задате дисциплине се прво деблокирају сви процеси који су чекали испред њега. Редови за чекање на условној променљивој су по дефиницији типа *FIFO*, али могу бити и другачије реализовани, као што је случај у програмском језику Java.

Када неки процес уради *cond.signal*, јављају се два потенцијална исхода:

1. Ако ниједан процес није блокиран у реду чекања на тој условној променљивој, не постоји **никакав ефекат** извршавања *cond.signal* наредбе. Ово је основна разлика у односу на синхронизацију помоћу семафора.
2. Ако постоји бар један процес у реду чекања на условној променљивој *cond*, процес који је дошао на ред се деблокира.

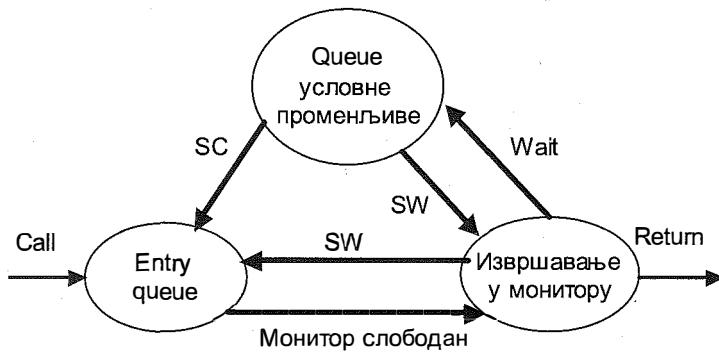
Операцијом *cond.queue* се може добити информација о стању реда чекања. Ако постоји бар један процес у реду чекања на условну променљиву *cond*, ова операција враћа логичку вредност *true*. У супротном се враћа логичка вредност *false*. У великом броју имплементација постоји и *cond.empty* која такође враћа логичку вредност и представља негацију вредности *cond.queue*.

Претпоставимо да процес долази до операције *cond.signal* док има ексклузивно право приступа (*lock*) због извршавања мониторске процедуре. Претпоставимо такође да је *cond.queue* био *true*, те да операција *cond.signal* буди други процес. Кључно је питање који процес се даље извршава у монитору? Постоје две основне дисциплине примењене у језицима за конкурентно програмирање:

*Signal and Continue* (скраћено SC) – *nonpreemptive*, код које процес који је извршио *cond.signal* наредбу задржава ексклузивну контролу над монитором. Ова дисциплина је чешће примењена у програмским језицима и оперативним системима (нпр. Java и UNIX).

*Signal and Wait* (скраћено SW) – *preemptive*, код које се контрола прослеђује пробуђеном процесу, а процес који је извршио *cond.signal* наредбу одлази на ред

чекања процеса спремних за извршавање мониторских процедура. Постоји посебан подслучај овог случаја који се назива *Signal and Urgent Wait*. Код овог подслучаја, процес који је генерирао *cond.signal* добија приоритет у односу на све процесе који спремни чекају на извршавање (и завршавање) мониторских процедура.



Слика 6. Дијаграм стања за мониторе

Понашање монитора у случају обе дисциплине је приказано на датом дијаграму стања (слика 6).

Прелаз из извршавања у монитору у чекање на ред условне променљиве (*Queue*) ослобађа монитор и дозвољава прелазак једног од спремних процеса из чекања на узломном реду (*Entry queue*) у извршавање у монитору (монитор слободан). У случају *SW* дисциплине, процес који се извршавао и извршио је *cond.signal* прелази у *Entry queue*, а процес који је чекао на *Queue* условне променљиве и дошао на ред прелази у извршавање у монитору (два *SW* прелаза се дешавају истовремено). У случају *SC* дисциплине, *cond.signal* изазива само прелаз једног процеса који је чекао из *Queue* условне променљиве у *Entry queue*.

Осим стриктне *FIFO* дисциплине при реализацији реда чекања на условној променљивој, могуће је реализовати и ред чекања са приоритетима. У том случају се код *cond.wait* проширује параметром *rank* којим се дефинише приоритет. Тада наредба добија облик *cond.wait(rank)*. Уколико *rank* има мању вредност, приоритет процеса који је суспендован је већи. Приликом суспендовања процеса се на овај начин одмах сортира *queue* условне променљиве по приоритету. Тада се помоћу *cond.minrank* може добити приоритет процеса који чека на почетку реда чекања те условне променљиве. За процесе који имају исти приоритет, примењује се *FIFO* дисциплина.

*cond.signal\_all* је посебна категорија операције *signal* над условном променљивом којом се комплетан ред чекања на условној променљивој празни и сви процеси који су чекали на тој условној променљивој се пребацују у *Entry queue*. У случају *SC* дисциплине ова операција би имала исти ефекат као и *while(cond.queue) cond.signal*. Ова операција није добро дефинисана код *SW* дисциплине и зато се код овог типа

## Дељене променљиве: Монитори

монитора не користи. Уколико је ред чекања на условној променљивој празан, *cond.signal\_all* нема никаквог ефекта.

Користећи синтаксу језика проширени *Pascal*, декларација монитора се може представити на следећи начин:

```
mname: monitor
var декларације сталних променљивих и условних променљивих
procedure p1(parameters);
var декларације локалних променљивих процедуре p1
begin
    код за имплементацију p1
end
...
procedure pN(parameters);
var декларације локалних променљивих процедуре pN
begin
    код за имплементацију pN
end
begin
    код за иницијализацију сталних променљивих
end;
```

Процедуре монитора и сталне променљиве монитора декларишу се као и стандардне процедуре и променљиве, док се за условне променљиве уводи посебан тип података *condition*.

## 26

## Резервација карата

Авиопревозник на линији Београд-Франкфурт-Лондон треба да врши резервацију појединачних карата помоћу софтвера заснованог на мониторима. Реализовати софтвер тако да се конкурентно могу резервисати карте Београд-Франкфурт и Франкфурт-Лондон, али да се позивом само једне мониторске процедуре могу резервисати и карте Београд-Лондон. Процедуре монитора морају бити угњеждене.

## Решење

Како би се обезбедила конкурентност приликом резервације карата за директне и комбиноване летове потребно је имати више монитора, за сваку категорију по један. За директни лет је потребно реализовати тип монитор који има могућност резервације и отказивања лета. Код комбинованог лета је потребно реализовати монитор који користи инстанце монитора за директне летове, а који такође има процедуре за резервацију и отказивање летова. Треба приметити да за директне летове постоји један тип монитора, али да постоји већи број инстанци ових монитора, један за Београд-Франкфурт и други за Франкфурт-Лондон, док код комбинованог лета постоји само једна инстанца.

Код монитора за директне летове постоје две процедуре које користе корисници *book* за куповину карата, и *cancel* за отказивање карата. У монитору постоји још једна функција коју користе директно, а то је функција која за датум датум проналази одговарајући лет. Код монитора је дозвољено из једне мониторске процедуре позивати другу процедуру и да за то није потребно тражити ново право коришћења монитора. Што се сталних променљивих тиче оне би укључивале листу летова.

У оквиру функције *book* прво се претражује листа свих летова како би се пронашао тражени лет. Онда се проверава да ли на датом лету има барем једно слободно место, задовољење услова *myFlight.available > 0*. Уколико постоји слободно место умањује се број преосталих слободних места и враћа да је карта успешно купљена. Уколико нема слободног места јавља се да карта није купљена.

У оквиру процедуре *cancel* прво се претражује листа свих летова како би се пронашао тражени лет, а онда се број преосталих слободних места увећава за један.

Треба приметити да је и код куповине и код враћања појединачне карте искоришћена особина међусобног искључивања процеса који приступају монитору. Овде је важно да нико не прекида трансакцију куповине односно враћања карата. Нема никаквог блокирања и чекања на испуњење неког услова јер ако нема карата сигурно да ли ће ико од купаца вратити неку купљену карту како би се дати процес одблокирао.

```
DirectFlight : monitor;
var ...
function book(date : Date) : boolean;
var myFlight : Flight;
begin
    myFlight := findFlight(date);
    if(myFlight.available > 0) then
begin
    myFlight.available := myFlight.available - 1;
    book := true
end
end;
```

```
end
else
    book := false;
end;

procedure cancel(date : Date);
var myFlight: Flight;
begin
    myFlight := findFlight(date);
    myFlight.available := myFlight.available + 1;
end;

function findFlight(date : Date) : Flight; begin ... end;

begin
end.
```

Треба приметити да уколико у одговарајућем програмском језику није могуће креирати већ број инстанци монитора, јер подржава само статичке мониторе, онда је потребно креирати два типа монитора један за Београд-Франкфурт и други за Франкфурт-Лондон. Ови монитори би имали идентичну структуру која би одговарала описаној.

Код монитора за комбиноване летове такође постоје две процедуре, *book* за куповину карата и *cancel* за отказивање карата. Нема додатних мониторских процедура. Што се сталних променљивих тиче оне би укључивале мониторе за релацију 1, Београд-Франкфурт, и за релацију 2, Франкфурт-Лондон. На овај начин се обезбеђује угњеждавање монитора.

У оквиру функције *book* се користе оба монитора за директне летове. На почетку се користећи први монитор купује карта за прву дестинацију. Ако мониторска функција врати да карта није доступна јавља се кориснику. Ако је карта купљена купује се карта на користећи други монитор. Ако је и ова карта купљена јавља се кориснику да је комбинована карта купљена. Ако карта није купљена позива се мониторска процедура првог монитора за отказивање прве карте.

У оквиру процедуре *cancel* прво се позива први монитор како би се тамо вратила карта, па се онда позива мониторска процедура другог монитора како би се и тамо вратила карта.

Треба приметити да се код куповине комбиноване карте када се обави куповина прве карте тај први монитор се ослобађа што повећава конкурентност јер је дозвољено да се у исто време продају и карте на првом и на другом монитору. Као и код претходног монитора овде је само искоришћена особина међусобног искључивања процеса који приступају монитору, али нема блокирања.

```
ConnectedFlight : monitor;

var   relation1 : DirectFlight;
      relation2 : DirectFlight;
function book(date : Date) : boolean;
var result : boolean;
begin
    result := relation1.book(date);
    if(result) then
begin
```

```

    result := relation2.book(date);
    if(not result) then
    begin
        relation1.cancel(date);
    end;
end;
book := result;

procedure cancel(date : Date);
begin
    relation1.cancel(date);
    relation2.cancel(date);
end;

begin
end.

```

Код описаног решења се може додати ситуација да дође први купац и да на монитор за комбиновани лет купи последњу карту. Након тога се ослобађа први монитор и овај купац прелази на други монитор како би купио другу карту. Овај први купац користећи комбиновани монитор сада позива други монитор како би купио карту. Претпоставимо да на том другом лету нема места. У међувремену долази други купац директно до првог монитора, који је слободан, проверава и види да више нема карата и одлази. Сада први купац поново приступа првом монитору и враћа купљену карту јер није било места на комбинованом лету. Уколико би сада дошао трећи купац на први лет он би затекао да има карата. Овакво решење би у неким случајевима могло да се сматра неправедним за другог купца који је долазио пре трећег а није добио карту. Како би се ово избегло могуће је за првог купца који нађе на то да нема више слободног места, а у време се обавља куповина карата за комбиновани лет, тај купац сачека док се куповина комплетно заврши. Како би се ово постигло потребан је нешто већи број мониторских процедура и ред за чекање.

Код монитора за директне летове постоје две процедуре које користе корисници *book* за куповину карата, и *cancel* за отказивање карата, и три које користи други монитор да би направио резервацију *reserve*, потврдио резервацију *confirmReservation* и откао резервацију *cancelReservation*. У монитору и даље постоји функција која за дати датум проналази одговарајући лет. Што се сталних променљивих тиче оне би укључивале и условну променљиву на којој би чекао путник који нађе на попуњен лет а да неко на том лету тренутно купује комбиновану карту, *blocked*.

У оквиру функције *book* прво се претражује листа свих летова како би се пронашао тражени лет. Онда се проверава да ли на датом лету има барем једно слободно место, задовољење услова *myFlight.available > 0*. Уколико постоји слободно место умањује се број преосталих слободних места и враћа да је карта успешно купљена. Уколико нема слободног места проверава се да ли се над овим летом тренутно ради резервација комбиноване карте. Уколико се не ради резервација или неко ко је стигао раније већ чека да на овом услову, (*not myFlight.reserve*) or *blocked.queue*, јавља се да карта није купљена. Уколико услов није испуњен онда се чак да се заврши куповина комбиноване карте и провера се обавља поново. Уколико има неста карта се купује уколико нема не купује се.

Резервација карте је доста слична куповини карата у претходном случају, са том разликом што се дати лет означава да се на њему ради резервација.

Отказивање резервације је слични отказивању карте у претходном случају, са том разликом што се лет означава да се на њему не ради више резервација и буђењу блокираног купца. Треба приметити да је купац могао да буде блокиран само на овом лету јер монитору за комбиновани лет у неком тренутку може приступати само један процес како би резервисао комбиновану карту.

```
DirectFlight : monitor;

var blocked : condition;

function book(date : Date) : boolean;
var myFlight : Flight;
begin
    myFlight := findFlight(date);
    if(myFlight.available > 0) then
        begin
            myFlight.available := myFlight.available - 1;
            book := true
        end
    else
        if((not myFlight.reserve) or blocked.queue) then
            book := false
        else
            begin
                blocked.wait;
                if(myFlight.available > 0) then
                    begin
                        myFlight.available := myFlight.available - 1;
                        book := true
                    end
                else
                    book := false
            end
    end;
end;

procedure cancel(date : Date);
var myFlight: Flight;
begin
    myFlight := findFlight(date);
    myFlight.available := myFlight.available + 1;
    if(myFlight.reserve and blocked.queue) then blocked.signal;
end;

function reserve(date : Date) : boolean;
var myFlight : Flight;
begin
    myFlight := findFlight(date);
    if(myFlight.available > 0) then
        begin
            myFlight.available := myFlight.available - 1;
            myFlight.reserve := true;
            reserve := true
        end
    else
        end;
```

```

    reserve := false
end;

procedure confirmReservation(date : Date);
var myFlight: Flight;
begin
    myFlight := findFlight(date);
    myFlight.reserve := false;
    blocked.signal;
end;

procedure cancelReservation(date : Date);
var myFlight: Flight;
begin
    myFlight := findFlight(date);
    myFlight.available := myFlight.available - 1;
    myFlight.reserve := false;
    blocked.signal;
end;

function findFlight(date : Date) : Flight; begin ... end;

begin
end.

```

Монитор за комбиновали лет је доста сличан претходној варијанти, са том разликом што се увједени први монитор уместо за куповину позива за резервацију карте. Такође се за отказивање прве карте код куповине не позива метода за отказивање карте него отказивање резервације. Разлика је и то што је сада на првом монитору позива и процедура за потврду резервације.

```

ConnectedFlight : monitor;

var relation1 : DirectFlight;
    relation2 : DirectFlight;
function book(date : Date) : boolean;
var result : boolean;
begin
    result := relation1.reserve(date);
    if(result) then
        begin
            result := relation2.book(date);
            if(not result) then
                relation1.cancelReservation(date)
            else
                relation1.confirmReservation(date)
        end;
    book := result;
end;

procedure cancel(date : Date);
begin
    relation1.cancel(date);

```

```
relation2.cancel(date);  
end;
```

```
begin
```

```
end.
```

→ Коментарисати следеће решење проблема: Досадашњи монитори за директни и комбиновани лет више нису монитори, већ су монитори дати за сваки појединачни лет. Шта би у решењу било потребно све да се промени уколико би могло да се купује/отказује и више карата уместо рада са појединачним картама као што је сада случај?

## 27

## Читаоци и писци

Решити проблем читалаца и писаца помоћу монитора.

- а) Поред стандардних правила усвојити и следеће: Писци имају право првенства при уласку, али када писац заврши са писањем, дозволити да сви читаоци који чекају могу да читају, без обзира да ли неки писац чека. Користити мониторе који имају *Signal and Wait* дисциплину.
- б) Решити овај проблем користећи мониторе који имају дисциплину *Signal and Continue*. Решење треба да обезбеди да процес који је пре стигао пре и започне операцију читања односно уписа.
- в) Решити овај проблем користећи мониторе који имају дисциплину *Signal and Wait* и приоритетне редове чекања. Решење треба да обезбеди да процес који је пре стигао пре и започне операцију читања односно уписа.

### Решење

```
a)
ReadersWritersMonitor : monitor;

var readcount : integer;
    busy : boolean;
    OKtoread, OKtowrite : condition;

procedure startread;
begin
    if (busy or OKtowrite.queue) then OKtoread.wait;
    readcount := readcount + 1;
    OKtoread.signal
end;

procedure endread;
begin
    readcount := readcount - 1;
    if readcount = 0 then OKtowrite.signal
end;

procedure startwrite;
begin
    if ((readcount <> 0) or busy) then OKtowrite.wait;
    busy := true
end;

procedure endwrite;
begin
    busy := false;
    if OKtoread.queue then OKtoread.signal else OKtowrite.signal
end;
```

```
begin
    readcount := 0;
    busy := false
end.
```

Сваки процес који хоће да чита мора најпре извршити процедуру *startread*, а сваки процес који хоће да пише мора најпре извршити процедуру *startwrite*. Слично, када процес заврши читање мора да изврши процедуру *endread*, односно када хоће да заврши писање мора да изврши процедуру *endwrite*.

*readcount* представља број процеса који тренутно чекају, а имају право да читају. *busy* има вредност *true* ако неки процес тренутно пише, а вредност *false* у супротном. Променљиве *OKtoread* и *OKtowrite* служе са условну синхронизацију. На њима чекају процеси који хоће да читају или пишу ако то није одмах могуће.

Променљива *busy* у оквиру процедуре *startread* служи за међусобно искључивање читалаца и писаца путем условне синхронизације. Услов *OKtowrite.queue* је тачан ако је неки процес блокиран на условној променљивој *OKtowrite*, а нетачан у супротном. Он служи да да предност писцима ако има оних који чекају. Наредба *Oktoread.signal* служи да би се пробудио следећи процес у листи оних који су чекали да читају (последњи читалац у реду ће такође извршити ову наредбу, али без ефекта).

У оквиру процедуре *endread* задовољење услова *readcount=0* значи да је дати процес био последњи који је читao, па се то сигнализира на условној променљивој (реду чекања) за писце који су чекали да сви читаоци заврше.

Услов *readcount<>0* у оквиру процедуре *startwrite* је задовољен ако неки читалац чита. Он обезбеђује међусобно искључивање писаца са читаоцима. Услов *busy* је тачан ако неко други пише. Он обезбеђује међусобно искључивање писаца.

У оквиру процедуре *endwrite* испитује се да ли има читалаца који чекају на приступ, у ком случају се дозвољава да сви читаоци који су чекали могу да почну да читају. У супротном, следећи писац може да почне да пише.

Ово решење је коректно, тј. има све особине сигурности и животности.

б) Ово је решење проблема читалаца и писаца код кога се полази од претпоставке да сваки процес треба да започне извршавање у редоследу у коме је затражио извршавање. Код овог решења се води евиденција о редоследу по коме сваки захтев пристиже и по том редоследу се сваки захтев обрађује. Претпоставка је да нема приоритетних редова чекања на условним променљивама. Циљ решења је да се изврши приоритирање свих приспелих захтева према тренутку доласка и да се тако приспeli захтеви обрађују. Приоритирање се постиже увођењем глобалног индекса који говори који је приспeli посао редни број у целокупној историји извршавања послова над датим монитором. Процес може започети са извршавањем само уколико се појавио његов редни број. Ово је примена тикет алгоритама рада, према коме се реализације и рад шалтерске службе. На почетку сваки процес добија јединствени редни број који означава када ће започети његова обрада. Када је добио редни број процес чека све док се на шалтеру не појави његов редни број. Када се појави његов редни број он има право да крене да ради. Овде треба водити рачуна о додељивању редних бројева, као и о додељивању који је то следећи број процеса који сме да крене да се извршава.

Сваки процес који хоће да чита мора најпре извршити процедуру *startread*, а сваки процес који хоће да пише мора најпре извршити процедуру *startwrite*. Слично, када процес заврши читање мора да изврши процедуру *endread*, односно када хоће да заврши писање мора да изврши процедуру *endwrite*. Треба приметити да је скуп

мониторских метода које корисник може звати исти независно од тога да ли је монитор реализован користећи *Signal and Wait* или *Signal and Continue* дисциплину.

Када неко од читалаца пожели да чита, позива мониторску процедуру *startread*. На почетку ове процедуре се процесу додељује јединствен идентификациони број *turn* на основу интерне променљиве *number*, након чега се интерна променљива *number* увећава за 1. Након тога чека да се његов идентификациони број поклопи са идентификационим бројем процеса коме је дозвољено да крене да се извршава (*next*). Чекање је остварено са условном променљивом *OKtoWork* која је заједничка и за читаоце и за писце. Пошто се овде ради са мониторима који имају дисциплину *Signal and Continue*, а који немају приоритетне редове чекања на условним променљивама потребно је радити проверу у *while* петљи. Када читалац добије дозволу да се извршава он увећава за један променљиву *readcount* која представља број процеса који тренутно читају. Након тога дозвољава следећем процесу у низу, било да је то писац или читалац да се извршава, увећавајући променљиву *next* за 1. Након ових подешавања буди све (*signal\_all*) процесе који чекају да израчунају своје услове и напушта коришћење монитора.

Поступак објављивања завршетка операције читања се налази у мониторској процедуре *endread*. У овој процедуре читалац умањује за један променљиву *readcount* и тиме имплицитно може да омогући неком писцу да започне са извршавањем. Након ових подешавања, уколико се ради о последњем читаоцу, буди све (*signal\_all*) процесе који чекају да израчунају своје услове и напушта коришћење монитора. Услов који је овде промењен може да користи само писцу који очекује да сви читаоци који су дошли пре њега заврше са радом.

Започињање процедуре уписа се добија позивом мониторске процедуре *startwrite*. На почетку ове процедуре се процесу додељује јединствен идентификациони број *turn* на основу интерне променљиве *number*, након чега се интерна променљива *number* увећава за 1. Следи чекање да се његов идентификациони број поклопи са идентификационим бројем процеса коме је дозвољено да крене да се извршава али се чека и да број писаца који је претходно започео извршавање постане 0 (*readcount<>0*). Чекање је остварено на условној променљивој *OKtoWork* која је заједничка и за читаоце и за писце. Пошто се овде ради са мониторима који имају дисциплину *Signal and Continue*, а који немају приоритетне редове чекања на условним променљивама потребно је радити проверу у *while* петљи. Када писац добије дозволу да се извршава напушта монитор не предајући право ни једном процесу. На овај начин је обезбеђено да када писац крене да се извршава нико други неће добити дозволу све док писац не заврши са радом.

Завршетак рада писца је представљен процедуром *endwrite* којом писац додељује право да се извршава првом следећем процесу по редоследу пристизања не разматрајући да ли је то писац или читалац. У овој процедуре писац дозвољава следећем процесу у низу, било да је то писац или читалац да се извршава, увећавајући променљиву *next* за 1. Након ових подешавања буди све (*signal\_all*) процесе који чекају да израчунају своје услове и напушта коришћење монитора.

Почетне вредности мониторских променљивих које представљају јединствен идентификациони број, број дозволе за извршавањем и броја читалаца, *number*, *next*, *readcount*, респективно треба да буду постављене на вредности 0, 0, 0. Овде треба нагласити да променљива *readcount* мора да буде постављена на вредност 0 док почетна вредност променљивих *number* и *next* не мора да буде 0 само је битно да ове две променљиве буду постављене на исту почетну вредност.

```
ReadersWritersMonitor : monitor;
var
    number, next, readcount : integer;
    OKtoWork: condition;

procedure startread;
var turn : integer;
begin
    turn := number;
    number := number + 1;
    while (turn <> next) do OKtoWork.wait;
    readcount := readcount + 1;
    next := next + 1;
    OKtoWork.signal_all;
end;

procedure endread;
begin
    readcount := readcount - 1;
    if(readcount = 0) then OKtoWork.signal_all;
end;
procedure startwrite;
var turn : integer;
begin
    turn := number;
    number := number + 1;
    while ((turn <> next) or (readcount <> 0)) do
        OKtoWork.wait;
end;
procedure endwrite;
begin
    next := next + 1;
    OKtoWork.signal_all;
end;
begin
    number := 0;
    next := 0;
    readcount := 0;
end.
```

в) Мана претходног решења проблема читалаца и писаца се огледала у томе што је приликом сваке итерације потребно будити све блокиране процесе да би се срачнао услов који појединим читаоцима односно писцима дозвољава да почну са извршавањем. Да би се добило на перформансама потребно је смањити број процеса који се буде. Пошто је код овог решења битно да се процеси буде у оном редоследу у коме су и пристизали ово се може постићи коришћењем приоритетних редова чекања на појединим условима променљивама. У претходном решењу ово није било доступно, па се приоритирање обављало програмским путем. Овде треба напоменути да се коришћењем обичних редова чекања не може на једноставан начин постићи FIFO редослед приступа ресурсу, јер да би писац почeo са радом потребно је обезбедити и да је писац први следећи у низу, али и да нема више активних читалаца. Треба приметити да понашање редова чекања не мора увек бити строго FIFO већ може зависити од имплементације, што је случај код програмског језика Java.

На почетку читалац позива мониторску процедуру *startread* на чијем се почетку процесу додељује јединствен идентификациони број *turn* на основу интерне променљиве *number*, након чега се интерна променљива *number* увећава за 1. Након тога читалац проверава да ли има дозволу за извршавање. Уколико има започиње са извршавањем уколико нема чека да га пробуди неко ко има дозволу. Чекање је остварено на условној променљивој *OKtoWork* која је заједничка и за читаоце и за писце и на којој се процеси блокирају према добијеном приоритету то јест на основу јединственог идентификационог броја. Треба приметити да пошто се ради о дисциплини *Signal and Wait* није неопходно непрекидно срачунавање услова, нема *while* петље, него је довољно само једном. Након тога дозвољава првом следећем процесу у низу, било да је то писац или читалац да се извршава, увећавајући променљиву *next* за 1. Ово ради само у случају да у реду постоји барем један блокирани процес.

Након завршетка операције читања потребно је да читалац позове процедуру *endread*. У овој процедуре читалац умањује за један променљиву *readcount* и у случају да нема више читалаца који раде последњи читалац проверава да ли постоји барем један блокирани процес који треба пробудити. Процес са за један већим идентификационим бројем може бити само писац. Уколико постоји такав процес само он се буди.

Писац започињање процедуре уписа позивом мониторске процедуре *startwrite*. На почетку ове процедуре се процесу додељује јединствен идентификациони број *turn* на основу интерне променљиве *number*, након чега се интерна променљива *number* увећава за 1. Након тога чека да се његов идентификациони број поклопи са идентификационим бројем процеса коме је дозвољено да крене да се извршава али се чека и да број читалаца који је претходно започео извршавање постане 0. Чекање је остварено на условној променљивој *OKtoWork* сходно добијеним приоритетима. У овом случају се приликом испитивања услова чекања уместо *if* наредбе користи *while* наредба јер писац може бити пробуђен а да му услови за започињање писања нису испуњени. Писац може бити пробуђен из два разлога: Први је јер га је пробудио читалац који је започео операцију читања, а други је јер га је пробудио или последњи читалац који је завршио са читањем или писац који је завршио са уписом. У првом случају писац треба да остане блокиран и да сачека док читаоци који су стигли пре него што заврши са читањем, док у другом случају писац треба да започне са писањем.

Завршетак рада писца је представљен процедуром *endwrite* којом писац додељује право да се извршава првом следећем процесу по редоследу пристизања не разматрајући да ли је то писац или читалац. Потребно је пробудити само први следећи процес у реду блокираних процеса.

```
ReadersWritersMonitor : monitor;
var
    number, next, readcount : integer;
    OKtoWork: condition;

procedure startread;
var turn : integer;
begin
    turn := number;
    number := number + 1;
    if (turn <> next) then OKtoWork.wait(turn);
    readcount := readcount + 1;
    next := next + 1;
    if(not OKtoWork.empty) then OKtoWork.signal;
end;
```

```

procedure endread;
begin
    readcount := readcount - 1;
    if((not OKtoWork.empty) and (readcount = 0)) then
        OKtoWork.signal;
end;
procedure startwrite;
var turn : integer;
begin
    turn := number;
    number := number + 1;
    while ((turn <> next) or (readcount <> 0)) do
        OKtoWork.wait(turn);
end;
procedure endwrite;
begin
    next := next + 1;
    if(not OKtoWork.empty) then OKtoWork.signal;
end;
begin
    number := 0;
    next := 0;
    readcount := 0;
end.

```

У овом решењу се код писаца приликом испитивања услова чекања уместо *if* наредбе користи *while* наредба јер писац може бити пробуђен, а да му услови за започињање писања нису испуњени. Разлог за ово лежи у томе што читалац када буди први следећи процес не зна да ли је тај процес читалац или писац. Уколико је читалац треба га пробудити, уколико је писац не треба га будити. Процес који буди једино може да сазна приоритет процеса који се буди, али не и који је тај наредни процес, односно који је његов тип. Како би се обезбедило да процес који буди сазна и који је тип првог наредног процеса може се применити техника уграђивања типа процеса у његов приоритет. Техника треба да очува редослед буђења процеса блокираног на условној променљиви. Ово се постиже на тај начин што се у приоритет угради и информација о редоследу и информација о типу. Једноставна трансформација којом се ово постиже је  $rank = N^P + T$ , где *rank* представља генерисани приоритет, *N* број различитих типова процеса, *P* представља улазни приоритет, а *T* представља идентификатор типа процеса који се блокира. Потребно је обезбедити да сваки процес има јединствен идентификатор типа који је у интервалу од 0 до *N*-1. На овај начин се постиже да се информација о типу првог процеса у реду блокираних може извучи користећи остатак при дељењу са бројем различитих типова (*queue.minrank mod N*).

```

ReadersWritersMonitor : monitor;
const R = 0;
      W = 1;
var number, next, readcount : integer;
    OKtoWork: condition;

procedure startread;
var turn : integer;
begin
    turn := number;

```

```

number := number + 1;
if (turn <> next) then OKtoWork.wait(2*turn + R);
readcount := readcount + 1;
next := next + 1;
if((not OKtoWork.empty) and ((OKtoWork.minrank mod 2)=R))
    then OKtoWork.signal;
end;

procedure endread;
begin
    readcount := readcount - 1;
    if((not OKtoWork.empty) and (readcount = 0)) then
        OKtoWork.signal;
end;
procedure startwrite;
var turn : integer;
begin
    turn := number;
    number := number + 1;
    if ((turn <> next) or (readcount <> 0)) then
        OKtoWork.wait(2*turn + W);
end;
procedure endwrite;
begin
    next := next + 1;
    if(not OKtoWork.empty) then OKtoWork.signal;
end;
begin
    number := 0;
    next := 0;
    readcount := 0;
end.

```

→ Коментарисати следећи проблем: Шта ће се дододити уколико се приликом увећавања променљивих *next* и *number* премаши опсег дозвољених вредности (*MAX\_VALUE*)? Да ли би следећи програмски сегмент који би се позивао на крају *endread* и *endwrite* процедуре отклонио дати проблем, ако проблем уопште постоји?

```

if (number = next and OKtoWork.empty and readcount = 0) then
begin
    next := 0;
    number := 0;
end;

```

## 28

## Тајмер

- a) Реализовати монитор који служи као тајмер програму који га позива, на тај начин што ће га пробудити након истека  $n$  јединица времена.
- b) Модификовати претходно решење тако да се буди што мање процеса. Користити *Signal and Wait* дисциплину.

## Решење

a)

```
alarmclock : monitor;
var now : integer;
wakeup : condition;

procedure wakeme (n : integer);
var alarmsetting: integer;
begin
    alarmsetting := now + n;
    while now < alarmsetting do wakeup.wait(alarmsetting);
    wakeup.signal;
end;

procedure tick;
begin
    now := now + 1;
    wakeup.signal;
end;

begin
    now := 0
end.
```

Процес који жели да нареди буђење после  $n$  јединица времена позива процедуру *wakeme*. У процедуре се најпре одређује време буђења (*alarmsetting*) и оно се потом користи као ниво приоритета који се прослеђује условној променљивој *wakeup* приликом стављања позивајућег процеса у ред чекања. На тај начин ће процеси који треба да се раније буде били почетку реда.

Процедуру *tick* позива хардвер (као прекидну (*interrupt*) рутину часовника у реалном времену (*real-time clock*) која се позива сваки пут након што истекне неки дефинисани елементарни период времена) или неки периодични процес. Када се то деси, процедура деблокира први процес у реду чекања на условној променљивој *wakeup* (у оквиру процедуре *wakeme*). Деблокирани процес наставља извршавање петље и проверава да ли дошао тренутак његовог буђења. Ако није, онда се у *while* петљи поново блокира, а ако јесте, онда врши буђење следећег процеса са истим временом буђења ако таквог има. Пошто је увек први процес у реду чекања онај који треба да се први пробуди, нема потребе будити остале процесе.

- б) Овај проблем ће бити решен креирањем монитора који има две процедуре, једну која је доступна кориснику којом тражи буђење након  $n$  јединица времена. Процес који

жели да наручи буђење после  $n$  јединица времена позива процедуру *wakeme* којој као аргумент прослеђује за колико јединица времена треба да буде пробуђен, а процедура интерно мора да води рачуна о тренутном времену. Да би се процес пробудио након  $n$  јединица времена потребно је израчунати у ком временском тренутку је потребно пробудити дати процес. У процедуре се најпре на основу садашњег тренутка, *now*, и времена које процес жели да буде успаван,  $n$ , одређује време буђења (*alarmsetting*) и оно се потом користи као ниво приоритета који се прослеђује условној променљивој *wakeup* приликом стављања позивајућег процеса у ред чекања. На овај начин се позивајући процес блокира и ставља у листу блокираних процеса препуштајући ексклузивно право коришћења монитора неком другом. На тај начин ће процеси који треба да се раније буде били почетку реда. Када процес који је блокиран буде пробуђен напушта процедуре.

Процедуру *tick* позива хардвер, ова процедура означава да је протекла једна јединица времена. Тачније, она представља прекидну рутину за *real-time clock* која се позива сваки пут након што истекне неки дефинисани елементарни период времена. Када се то деси, процедура проверава да ли постоје блокирани процеси које треба пробудити у текућем тренутку, који чекају на условној променљивој *wakeup* (у оквиру процедуре *wakeme*). Процеси се буде у једној петљи и то буде се само они процеси који треба да буду пробуђени у том тренутку, а не први који се налази у листи, без обзира да ли је дошао његов тренутак за буђење. Пошто се ради о сортираном низу процеса буде се само они који се налазе на почетку листе, оне са максималним приоритетом под условом да дати приоритет представља садашњи тренутак (*wakeup.minrank <= now*), нема потребе будити остале процесе. Провера да ли постоји процес у листи се остварује користећи функцију *empty* која постоји дефинисана за услове, док се приликом одређивања да ли процес треба пробудити узима приоритет приликом буђења користећи позив функције *minrank* која постоји дефинисана код услова.

```

alarmclock : monitor;
var now : integer;
    wakeup : condition;
procedure wakeme (n : integer);
var alarmsetting : integer;
begin
    alarmsetting := now + n;
    wakeup.wait (alarmsetting);
end;
procedure tick;
begin
    now := now + 1;
    while ((not wakeup.empty)
        and (wakeup.minrank <= now)) do
        wakeup.signal
    end;
begin
    now := 0
end.

```

Процедура *tick*, коју позива хардвер, у петљи буди само оне процесе који треба да буду пробуђени у том тренутку. Начин буђења се може мало изменити тако да ова процедура буде само први процес у листи ако тај процес треба да буде пробуђен. Када се тај процес пробуди онда проверава исти услов и по потреби буди наредни процес из листе. На овај начин се информација прослеђује све док има процеса који треба да

буду пробуђени у датом тренутку. Пошто је сигнализација на условној променљиви (*wakeup.signal*) последња операција у монитору за методу *tick* програмски преводилац би могао да генерише код за напуштање монитора уместо за чекање на улазном реду. На овај начин би могло да се гарантује колико времена се максимално извршава прекидна рутина за *real-time clock* што је веома битно код система који раде у реалном времену.

```
alarmclock : monitor;
var now : integer;
    wakeup : condition;
procedure wakeme (n : integer);
var alarmsetting : integer;
begin
    alarmsetting := now + n;
    wakeup.wait (alarmsetting);
    if ((not wakeup.empty)
        and (wakeup.minrank <= now)) then
        wakeup.signal
end;
procedure tick;
begin
    now := now + 1;
    if ((not wakeup.empty)
        and (wakeup.minrank <= now)) then
        wakeup.signal
    end;
begin
    now := 0
end.
```

## 29 Улазак у авион

На аеродрому, путници чекају на улазак у авион. Стјуардеса пушта путнике у авион по следећем редоследу група путника:

1. Породице са децом млађом од 5 година се пуштају прве
2. Особе старије од 70 година
3. Путници из пословне класе
4. Путници до 15. реда
5. Путници до 25. реда
6. Сви остали путници.

Реализовати монитор који ће имати две процедуре: *zelim\_da\_udjem\_u\_avion* и *pocinje\_ulazak\_u\_avion*. Прву процедуру треба да позивају путници, а другу стјуардеса. Реализовати, на најједноставнији начин монитор који ће обезбедити улазак у авион.

### Решење

```
Avion : monitor;

var prioritet : condition;

procedure zelim_da_udjem_u_avion(grupa : integer);
begin
    prioritet.wait(grupa);
    if(not prioritet.empty) then prioritet.signal;
end;

procedure pocinje_ulazak_u_avion;
begin
    prioritet.signal;
end;

begin
end.
```

30

### Улазак у школу

Монитором треба да се регулише улазак ђака у основну школу. Редослед уласка треба да буде такав да прво улазе ђаци нижих разреда, и то по редоследу одељења. Сви ђаци се већ налазе у дворишту школе, у тренутку када дежурни наставник објави улазак у школу. Наставник позива процедуру *pocnite\_ulazak\_u\_skolu* након чега почине улазак деце у задатом редоследу. Деца чим дођу у двориште позивају процедуру *zelim\_u\_skolu*. Ни за један разред нема више од 9 одељења. Написати монитор.

## Решење

Приликом блокирања могуће је користити само један приоритетни ред чекања. Овај ред је потребно тако реализовати да се процеси блокирају слодно редоследу по коме их је потребно пробудити. Редослед буђења одељења и разреда је могуће скрити у поље приоритет користећи једначину *razred\*MAXODELJENJA + odeljenje*.

```
Skola : monitor;
const MAXODELJENJA = 9;

var dvoriste : condition;

procedure zelim_u_skolu(razred : integer; odeljenje : integer);
begin
    dvoriste.wait(razred*MAXODELJENJA + odeljenje);
    if((not dvoriste.empty)) then dvoriste.signal;
end;

procedure pocnite_ulazak_u_skolu;
begin
    dvoriste.signal;
end;

begin
end.
```

## 31 Кружни FIFO бафер

Реализовати монитор који служи као кружни FIFO бафер чији су елементи типа  $T$ .

### Решење

```

buffer(T) : monitor;
const N = ...;
var slots : array[0..N-1] of T;
    head, tail : 0..N-1;
    size : 0..N;
    not_full, not_empty : condition;

procedure put(p : T);
begin
    if size = N then not_full.wait;
    slots[tail] := p;
    size := size + 1;
    tail := (tail + 1) mod N;
    not_empty.signal
end;

procedure get(var it : T);
begin
    if size = 0 then not_empty.wait;
    it := slots[head];
    size := size - 1;
    head := (head + 1) mod N;
    not_full.signal
end;

begin
    size := 0; head := 0; tail := 0
end.
```

Реализован је кружни бафер са максимално  $N$  елемената у виду низа *slots*. На први елемент у бафтеру указује показивач *head*, а на прво слободно место у бафтеру показивач *tail*. Приликом стављања елемента у бафер најпре се проверава да ли је бафер пун. Ако јесте, процес се ставља у ред чекања на услову *notfull* да се ослободи место у бафтеру, а ако не, онда се убацује нова вредност у низ и помера показивач *tail*. На крају се сигнализира да бафер није празан у случају да неки процес жели да чита из бафтера. Ако ниједан процес није био блокиран, наредба нема ефекта. Код дохватања вредности из бафтера, најпре се проверава да ли је бафер празан. Ако јесте, процес се ставља у ред чекања на услову *notempty* док се не упише нешто у бафер, а ако не, узима се први елемент из бафтера и помера одговарајући показивач. На крају се сигнализира преко условия *notfull* да бафер није пун, у случају да је неки процес био блокиран на том услову. Ако није било блокираних процеса, наредба нема ефекта.

Треба приметити да дато решење ради коректно у случају монитора са SW дисциплином, док у случају монитора са SC дисциплином може да доведе до нерегуларне ситуације. Пример овакве ситуације би био сценарио код кога је бафер пун, на улазном реду је блокиран један произвођач, на условној променљиви *not\_full* је блокиран други произвођач, а потрошач извршава сигнализацију на променљиви

### Дељене променљиве: Монитори

*not\_full* (*not\_full.signal*) након чега напушта монитор. Уколико би се прво извршио први произвођач он би проверио услов, закључио да има места у баферу и сместио елементу у бафер. Када се након тога пробуди други произвођач он не би проверавао услов и сместио би елементу у пун бафер преко неког другог елемента. Овај проблем се може решити заменом услова *if* условом *while*.

## 32 Сакупљање гајбица

Монитор треба да обезбеди скупљање гајбица са ивице воћњака. Процеси берачи доносе по неколико пуних гајбица (зависно од носивости берача) до пута на ивици њиве. Када поред пута берачи оставе бар *prikolica\_kapacitet* гајбица, треба да се пробуди један од процеса трактора са приколицом, да дођу до пута крај њиве и покупе онолико гајбица колико стаје у приколицу (*prikolica\_kapacitet*, што је веће од носивости берача). Процеси трактори са приколицом позивају мониторску процедуру *spreman\_da\_pokupim* када заврше превоз претходне приколице воћа. Берачи позивају мониторску процедуру *ostavljam\_gajbice*, када донесу гајбице до ивице пута. Подразумевати *Signal and Wait* дисциплину.

### Решење

```

program Gajbice;
const NUM_BERACA = ...;
      NUM_TRAKTORA = ...;

SakupljanjeGajbica : monitor;
  const prikolica_kapacitet = 10;
  var
    njiva : condition;
    tovar : integer;

  procedure spremam_da_pokupim;
  begin
    if (tovar < prikolica_kapacitet ) then njiva.wait;
    tovar := tovar - prikolica_kapacitet;
  end;

  procedure ostavljam_gajbice(nosivost_beraca : integer);
  begin
    tovar := tovar + nosivost_beraca;
    if (tovar >= prikolica_kapacitet ) then njiva.signal;
  end;
begin
  tovar := 0;
end;

procedure Berac(id : integer; nosivost_beraca : integer);
procedure sakupljaj; begin ... end;
begin
  while (true) do
  begin
    sakupljaj;
    ostavljam_gajbice(nosivost_beraca);
  end;
end;

procedure Traktor_sa_prikolicom(id : integer);
procedure prevoz; begin ... end;
begin

```

```
while (true) do
begin
    spreman_da_pokupim;
    prevoz;
end;
end;

begin
cobegin
    traktor_sa_prikolicom(1);
    ...
    traktor_sa_prikolicom(NUM_TRAKTORA);
    berac(1, 10);
    ...
    berac(NUM_BERACA, 23);
coend;
end.
```

## 33 Прање веша

Помоћу монитора регулисати прање веша у домаћинству. Процеси укућани (укупно  $N$  њих) разврставањем прљавог веша пуне 3 корпе веша које садрже: бели веш (прање на  $95^\circ$  Целзијуса), шарени веш (прање на  $60^\circ$  Целзијуса) и осетљиви веш (на  $40^\circ$  Целзијуса) респективно. При убацивању, укућани дефинишу тежину прљавог веша који убацују за сваку корпу појединачно. Процес машина на долазак јефтине струје активира прање једне машине неке врсте (из једне корпе), само ако у било којој корпи постоји више од  $L$  килограма веша. Ако тада у више корпи постоји више од  $L$  килограма, бира се за прање корпа која има највише килограма веша. Машина пере тачно  $L$  килограма веша у сваком прању. Процеси који користе монитор су процеси укућани и процес машине. Процес машина позива процедуру *rokišaj\_prajanja* и добија од монитора информацију: Да ли треба да пере и коју категорију веша треба да пере. Сигнал за долазак јефтине струје је део процеса машине и за њега не треба писати код. Подразумевати *Signal and Continue* дисциплину.

### Решење

#### Прво решење

```

PranjeVesa : monitor;
const L = ...;

var
    korpa_belo, korpa_sareno, korpa_osectljivo, korpekap : integer;
    belo_peri : condition; (* signal kada se pere beli ves *)
    sareni_peri : condition; (* signal kada se pere sareni ves *)
    osetljivo_peri : condition; (* signal kada se pere osetljivi
    ves *)

procedure razvrstavanje(beli : integer;
    sareni : integer; osetljivi : integer);
begin
    while(beli <> 0) do begin
        if(korpa_belo + beli > korpekap) then begin
            beli := korpa_belo + beli - korpekap;
            korpa_belo := korpekap;
        end
        else begin
            korpa_belo := korpa_belo + beli;
            beli := 0;
        end;
        if(beli <> 0) then belo_peri.wait
    end;

    while(sareni <> 0) do begin
        if(korpa_sareno + sareni > korpekap) then begin
            sareni := korpa_sareno + sareni - korpekap;
            korpa_sareno := korpekap;
        end
        else begin

```

```
korpa_sareno := korpa_sareno + sareni;
sareni := 0;
end;
if(sareni <> 0) then sareno_peri.wait
end;

while(osetljivi <> 0) do begin
    if(korpa_osetljivo + osetljivi > korpekap) then begin
        osetljivi := korpa_osetljivo + osetljivi - korpekap;
        korpa_osetljivo := korpekap;
    end
    else begin
        korpa_osetljivo := korpa_osetljivo + osetljivi;
        osetljivi := 0;
    end;
    if(osetljivi <> 0) then osetljivo_peri.wait
end;

end;

procedure pokusaj_pranja(var radi : boolean;
    var tip_pranja : integer);
begin
    radi := false;
    if (korpa_belo > korpa_sareno) then
    begin
        tip_pranja := 1;
        if (korpa_belo < korpa_osetljivo) then tip_pranja := 3;
    end
    else
    begin
        tip_pranja := 2;
        if (korpa_sareno < korpa_osetljivo) then tip_pranja := 3;
    end;
    case tip_pranja of
    1: if korpa_belo > L then
    begin
        radi := true;
        korpa_belo := korpa_belo - L;
        belo_peri.signal_all;
    end;
    2: if korpa_sareno > L then
    begin
        radi := true;
        korpa_sareno := korpa_sareno - L;
        sareno_peri.signal_all;
    end;
    3: if korpa_osetljivo > L then
    begin
        radi := true;
        korpa_osetljivo := korpa_osetljivo - L;
        osetljivo_peri.signal_all;
    end;
    end;
end;
end;
```

```

begin
    korpa_belo := 0;
    korpa_sareno := 0;
    korpa_osectljivo := 0;
    korpekap := 2*L; {* vrednost veca od L *}
end.

```

Друго решење

Уколико је количина веша коју у једном тренутку укућанин може убацити у корпу мања или једнака капацитету машине, а капацитет корпе је барем једнак двоструком капацитету машине, решење се може упростити.

```

PranjeVesa : monitor;
const L = ...;

var
    korpa_belo, korpa_sareno, korpa_osectljivo, korpekap : integer;
    belo_peri : condition; {* signal kada se pere beli ves *}
    sareno_peri : condition; {* signal kada se pere sareni ves *}
    osetljivo_peri : condition; {* signal kada se pere osetljivi
ves *}

procedure razvrstavanje(beli : integer;
    sareni : integer; osetljivi : integer);
begin
    while ((korpa_belo + beli) > korpekap) do belo_peri.wait;
    korpa_belo := korpa_belo + beli;
    while ((korpa_sareno + sareni) > korpekap) do sareno_peri.wait;
    korpa_sareno := korpa_sareno + sareni;
    while ((korpa_osectljivo + osetljivi) > korpekap) do
        osetljivo_peri.wait;
    korpa_osectljivo := korpa_osectljivo + osetljivi;
end;

procedure pokusaj_pranja(var radi : boolean;
    var tip_pranja : integer);
begin
    radi := false;
    if (korpa_belo > korpa_sareno) then
        begin
            tip_pranja := 1;
            if (korpa_belo < korpa_osectljivo) then tip_pranja := 3;
        end
    else
        begin
            tip_pranja := 2;
            if (korpa_sareno < korpa_osectljivo) then tip_pranja := 3;
        end;
    case tip_pranja of
        1: if korpa_belo > L then
            begin
                radi := true;
                korpa_belo := korpa_belo - L;
            end;

```

```
        belo_peri.signal_all;
end;
2: if korpa_sareno > L then
begin
    radi := true;
    korpa_sareno := korpa_sareno - L;
    sareno_peri.signal_all;
end;
3: if korpa_osetljivo > L then
begin
    radi := true;
    korpa_osetljivo := korpa_osetljivo - L;
    osetljivo_peri.signal_all;
end;
end;
end;
begin
    korpa_belo := 0;
    korpa_sareno := 0;
    korpa_osetljivo := 0;
    korpekap := 2*L;
end.
```

## 34 Филозофи за ручком

Решити проблем филозофа за ручком, користити мониторе са *Signal and Wait* дисциплином.

### Решење

#### Прво решење

Проблем филозофа који ручавају се овде решава користећи мониторе који воде рачуна о томе који филозоф може да руча. Филозоф може да руча уколико има на располагању две вилљушке. Ово решење је тако реализовано да се не води рачуна о самим вилљушкама већ се води рачуна о стању у коме се налази сваки филозоф посебно. Сваки филозоф се може наћи у једном од следећа три стања: размишља, жели да једе и у стању једе. Услов који треба да буде испуњен да би филозоф могао да једе је да: жели да једе, то јест налази се у том стању и други услов је да ни један његов сусед не једе. Стање у коме се налази филозоф је описано користећи низ *state*.

Улазни протокол за филозофа је да прво мора да позове мониторску процедуру *pickup* којој прослеђује свој идентификациони број. У процедуре филозоф прво исказује жељу да започне са јелом постављајући своје стање на одговарајућу вредност, *state[id] := hungry*. Након тога позива процедуре у којој проверава да ли сме да једе, *test(id)*. Уколико је повратни резултат процедуре такав да је филозоф из стања жели да једе прешао у стање једе онда се процедура завршава. Уколико филозоф није добио одобрење да једе онда прелази на фазу чекања да му неки сусед који тренутно једе дозволи да започне са јелом, *if (state[id] <> eating) then can\_eat[id].wait*.

Излазни протокол за филозофа захтева да се позове процедура *putdown*. Унутар ове процедуре филозоф обележава своје стање са размишља и проверава да ли неки од суседа има жељу да једе. Позива процедуру за проверу за сваког од својих суседа, која преко својих бочних ефеката буди суседе. Тек када је проверио да ли суседи могу да једу напушта процедуру.

Процедура у којој се проверава да ли специфицирани филозоф сме да једе води се следећом логиком: Филозоф сме да једе уколико је пожелео да једе, *state[k] = hungry*, и уколико му не једе леви сусед, *state[(k+N-1) mod N] <> eating*, и уколико му не једе ни десни сусед, *state[(k+1) mod N] <> eating*. Тек тада се филозофу дозвољава да једе постављањем његовог стања на вредност *eating* и сигнализацијом на одговарајућем услову, *can\_eat[k].signal*. Треба приметити да сигнализација на услову неће увек имати ефекта. Имаће ефекта само уколико обавештавање ради филозоф који је завршио са јелом и који буди неког блокираног филозофа. Неће имати никаквог ефекта у случају да филозоф ради проверу за себе у својој улазној процедуре *pickup*.

```
program DiningPhilosophers;
const N = 5;
    thinking = 0; hungry = 1; eating = 2;

DiningPhilosophersMonitor : monitor;
  var
    can_eat : array [0.. N-1] of condition;
    state : array [0.. N-1] of integer;
    index : integer;
```

```
procedure test (k: integer);
begin
    if ((state[(k+N-1) mod N] <> eating)
        and (state[k] = hungry)
        and (state[(k+1) mod N] <> eating)) then
    begin
        state[k] := eating;
        can_eat[k].signal;
    end;
end;
procedure pickup(id : integer);
begin
    state[id] := hungry;
    test(id);
    if (state[id] <> eating) then can_eat[id].wait;
end;

procedure putdown (id : integer);
begin
    state[id] := thinking;
    test((id+N-1) mod N);
    test((id+1) mod N);
end;
begin
    for index := 0 to N-1 do state[index] := thinking;
end;

procedure Philosopher(id : integer);
procedure think; begin ... end;
procedure eat; begin ... end;
begin
    while (true) do
    begin
        think;
        pickup(id);
        eat;
        putdown(id);
    end;
end;
begin
    cobegin
        Philosopher(0);
        ...
        Philosopher(N-1);
    coend;
end.
```

Треба приметити да дато решење ради коректно и у случају монитора са SW дисциплином и монитора са SC дисциплином. Ово је постигнуто на тај начин што филозоф који завршава са јелом (процес који ослобађа ресурс) проверава услов који треба да буде испуњен да би други процес био пробуђен. Уколико је услов за буђење испуњен процес који ослобађа ресурс пре него што пробуди блокирани процес сам мења услов на вредност која одговара стању у које ће прећи процес који се буди. На овај начин је постигнуто да процес који се буду не мора поново да проверава услов.

Све што је важило за одговарајуће решење са семафорима, стоји и овде.

### Друго решење

У случају да је потребно гарантовати редослед *FIFO* по коме филозофи започињу са јелом може се применити тикет алгоритам и техника угађивања информација у приоритет.

Филозоф на почетку позове мониторску процедуру *pickup* којој прослеђује свој идентификациони број. У процедуре филозоф прво исказује жељу да започне са јелом, *state[id] := hungry*. Након тога позива процедуру у којој проверава да ли сме да једе, *test(id)*. Уколико је повратни резултат процедуре такав да је филозоф из стања жели да jede прешао у стање једе онда се процедура завршава. Уколико филозоф није добио одобрење да једе онда прелази на фазу чекања да му неко, не обавезно сусед, дозволи да започне са јелом. За чекање се примењује тикет алгоритам где филозоф узима први наредни редни број, увећава наредни број и блокира се на услову (заједничком за све филозофе) угађајући у свој приоритет информацију о свом идентификационом броју,  $N^*turn + id$ . Процедура у којој се проверава да ли филозоф сме да једе је слична претходној. Филозоф сме да једе уколико нико ко је дошао пре њега не чека, сам је покелео да једе и уколико не једе ни леви ни десни сусед. Тек тада се филозофу дозвољава да једе постављањем његовог стања на вредност *eating*.

На крају филозоф позива процедуру *putdown* унутар које обележава да је у стању размишљања, проверава да ли су испуњени услови да први блокирани филозоф једе, ако нису испуњени завршава. Ако су испуњени услови буди тог филозофа, и проверава за наредног филозофа. Ако су испуњени услови буди га, ако нису завршава. Ове провере се обављају унутар функције *test* која проверава услове за први процес у листи блокираних.

Процедура у којој се проверава да ли први филозоф у низу сме да једе је слична оној која проверава за конкретног филозофа, са том разликом што се идентификатор првог филозофа добија учитавањем најмањег приоритета свих блокираних филозофа и издавањем угађеног идентификатора, *can\_eat.minrank mod N*, и што се не проверава да ли има других блокираних филозофа.

```
DiningPhilosophersMonitor : monitor;
var can_eat : condition;
    state : array [0..N-1] of integer;
    number : integer;
    index : integer;
procedure testMe (k : integer);
begin
    if (can_eat.empty
        and (state[(k+N-1) mod N] <> eating)
        and (state[k] = hungry)
        and (state[(k+1) mod N] <> eating)) then
    begin
        state[k] := eating;
    end;
end;
function test : boolean;
var k : integer;
begin
    test := false;
    if(can_eat.queue) then
```

```

begin
    k := can_eat.minrank mod N;
    if ((state[(k+N-1) mod N] <> eating)
        and (state[k] = hungry)
        and (state[(k+1) mod N] <> eating)) then
begin
    state[k] := eating;
    can_eat.signal;
    test := true;
end;
end;
procedure pickup(k : integer);
var turn : integer;
begin
    state[k] := hungry;
    testMe(k);
    if (state[k] <> eating) then
begin
    turn := number;
    number := number + 1;
    can_eat.wait(N*turn + k);
end;
end;
procedure putdown (k : integer);
var ok : boolean;
begin
    state[k] := thinking;
    ok := test();
    if (ok) then ok := test();
end;
begin
    number := 0;
    for index := 0 to N-1 do state[index] := thinking;
end;

```

#### Треће решење

У неким случајевима, када се жели већа конкурентност, није неопходно одржавати строги FIFO редослед започињања јела па се редослед буђења може релаксирати. Ова релаксација се огледа у томе да филозоф не чека више од S филозофа како би започео са јелом.

Филозоф на почетку прво исказује жељу да започне са јелом,  $state[id] := hungry$ , узима јединствен редни број, редослед пристизања, који смешта у локалну променљиву  $turn$  и увећавају овај редни број ( $number:=number+1$ ). Након тога позива процедуре у којој проверава који све филозофи смеју да једу. Уколико је повратни резултат процедуре такав да је филозоф из стања жели да једе прешао у стање једе онда се процедура завршава. Уколико филозоф није добио одобрење да једе онда прелази на фазу чекања да му неки филозоф, не мора бити суседни, дозволи да започне са јелом.

Процедура у којој се проверава да ли било који филозоф сме да једе проверава за сваког појединачно филозофа да ли сме да једе. Процедура за појединачне филозофе је слична претходним процедурама. Филозоф сме да једе уколико је пожелео да једе, уколико не једе ни леви ни десни сусед, нити је било леви било десни сусед гладан а

дошли су довољно раније. Тек тада се филозофу дозвољава да једе постављањем његовог стања на вредност *eating*.

На крају филозоф обележава да је у стању размишљања и проверава да ли било који филозоф може да једе.

```
DiningPhilosophersMonitor : monitor;
var can_eat : array [0..N-1] of condition;
    state : array [0..N-1] of integer;
    turn : array [0..N-1] of integer;
    number : integer;
    index : integer;
procedure testAll;
begin
    for index := 0 to N-1 do test(index);
end;

procedure test (k: integer);
begin
    if ((state[(k+N-1) mod N] <> eating)
        and (state[k] = hungry)
        and (state[(k+1) mod N] <> eating)
        and
        (not (
            ((state[(k+N-1) mod N] = hungry)
             and (turn[k] - turn[(k+N-1) mod N] > S))
            or
            ((state[(k+1) mod N] = hungry)
             and (turn[k] - turn[(k+1) mod N] > S)))
        )
    )
    ) then
begin
    state[k] := eating;
    can_eat[k].signal;
end;
end;
procedure pickup(k : integer);
begin
    state[k] := hungry;
    turn[k] := number;
    number := number + 1;
    testAll();
    if (state[k] <> eating) then can_eat[k].wait;
end;
procedure putdown (k : integer);
begin
    state[k] := thinking;
    testAll();
end;
begin
    number := 0;
    for index := 0 to N-1 do state[index] := thinking;
end;
```

**35****Недељиво емитовање**

Постоји један произвођач и  $N$  потрошача који деле заједнички бафер капацитета  $B$  (*The Atomic Broadcast Problem*). Произвођач убацује производ у бафер и то само у слободне слотове на који чекају свих  $N$  потрошача. Сваки потрошач мора да прими производ у тачно оном редоследу у коме су произведени, мада различити потрошачи могу у исто време да узимају различите призводе.

- Решити проблем недељивог емитовања користећи мониторе који имају *Signal and Continue* дисциплину.
- Решити проблем недељивог емитовања користећи мониторе који имају *Signal and Wait* дисциплину.

**Решење**

- Приликом решавања овог проблема констатујемо да постоје две критичне операције које је потребно одрадити. Једна је смештање производа у бафер, а друга за узимање производа из бафера.

Процедура за смештање производа у бафер, *putItem(item : integer)*, као аргумент прима производ, цео број, који је потребно сместити у бафер, то јест проследити га свим потрошачима. На почетку ове процедуре произвођач чека да се ослободи место на које треба убацити нови производ. Место на које се смешта производ дато је помоћу индекса *writeToIndex*. Произвођач констатује да је место слободно тако што утврди да је са истог тог места већ  $N$  потрошача узело претходно смештени производ, *while(num[writeToIndex] <> N)*. Чекање се обавља на условној променљивој *writeCondition*. Једна условна променљива је придружена производијачу. Место може бити слободно из два разлога: зато што се ради о првој итерацији или зато што су сви потрошачи узели производ са позиције на коју треба поставити следећи производ. Овде мора да се води рачуна да се пре прве итерације бројач колико је сваки објекат претходно пута био конзумиран, *num[writeToIndex]*, постави на вредност  $N$ . Бројач је придружен свакој локацији у баферу. Када се производ смести у бафер бројач приступа датом производу, односно одговарајућој позицији у баферу, се постави на вредност 0. Након овога се буде сви потрошачи блокирани на приспеће новог производа на дату локацију, *readCondition[writeToIndex].signal\_all*, и пребацују се из листе блокираних у листу спремних процеса. Они ће приступ локацији, односно монитору добити тек када се производијач одрекне ексклузивног права коришћења монитора напуштањем процедуре за убаџавање производа у бафер. Поред смештања производа у бафер производијач поставља и колико је производа икада до дада сместио у бафер. Овај податак је неопходан да би се могло пратити колико је производа конзумирао који потрошач. Приликом смештања производа у бафер потребно је одредити на коју је локацију у баферу могуће сместити следећи производ. Пошто се ради о кружном баферу алгоритам по ком се рачуна следећа позиција је *writeToIndex := (writeToIndex + 1) mod B*.

Функција за узимање производа из бафера, *function getItem(id : integer) : integer*, као аргумент прима идентификацију потрошача који узима производе из бафера, а као повратну вредност даје сам производ узет из бафера. На почетку ове процедуре потрошач чека да се попуни место са кога треба покупити нови производ. Место са којег се узима производ дато је помоћу индекса *readFromIndex[id]*. Податке о локацији са које се узима следећи производ мора да се чува унутар монитора јер потрошач након повратка из процедуре више нема података о интерном стању монитора, то јест

о локацији са које је прочитao претходну вредност. Поред тога што тај податак нема потрошач, он не треба ни да га има јер је цела концепција монитора таква да комплетна синхронизација треба да се сакрије од крањег корисника који треба да добија само податке, а не да води рачуна око синхронизације. Комплетна синхронизација треба да буде склоњена у монитор. Потрошач констатује да је место попуњено тако што утврди да је произвођач прешао на наредну итерацију, *while(indexW = indexR[id])*. Чекање се обавља на условној променљивој *readCondition[readFromIndex[id]]*, која је придруженда одговарајућој локацији у баферу. Након узимања производа потрошач повећава за један бројач придружен локацији у баферу са које је узео производ. Уколико је број приступа постао једнак броју процеса који су требали да приступе датом производу (*N*), о томе се обавештава произвођач који можда чека *writeCondition.signal*, и пребацују се из листе блокираних у листу спремних процеса. Следећа ствар коју је потребно урадити је увећати променљиву која исказује колико је производа дати процес потрошач конзумирао. Приликом узимања производа из бафера потребно је одредити са које је локације у баферу треба узети следећи производ. Пошто се ради о кружном баферу алгоритам по ком се рачуна следећа позиција за датог потрошача је *readFromIndex[id] := (readFromIndex[id] + 1) mod B*.

Заједничке променљиве које се користе приликом решавања овог проблема су: бафер *buffer* капацитета *B* за смештање производа, низ бројача *num* придружених свакој локацији у баферу у којима се чува колико је пута одређени производ конзумиран, низ индекса *readFromIndex* за сваког потрошача у којима се чува која је следећа локација са које се узима неки производ, бројач *writeToIndex* који каже на коју је локацију потребно сместити следећи производ, низ бројача *indexR* придружених сваком потрошачу у коме се чува колико је производа до сада конзумирао одговарајући потрошач, и бројач *indexW* који каже колико је производа произвођач до сада сместио у бафер. Да би систем могао да почне да функционише потребно је дозволити произвођачу да може да смешта производе на сваку локацију почев од почетне локације. Да би се ово постигло сваки бројач приступа треба поставити на вредност *N* (*for i := 0 to B-1 do num[i] := N*), а почетне индексе за произвођача и потрошаче на исту почетну вредност у овом примеру на нула. Није неопходно да то буде вредност 0 важно је да сви почну да раде од исте локације. Поред тога потребно је поставити и бројаче генерираних односно конзумираних производа на исте почетне вредности, и овде је одабрана вредност 0, мада је све једно о којој се вредности ради све док се бројање врши почев од исте.

```
AtomicBroadcast : monitor;
const N = ...;
B = ...;
var buffer : array [0..B-1] of integer;
num : array [0..B-1] of integer;
readCondition : array [0..B-1] of condition;
writeCondition : condition;
readFromIndex : array [0..N-1] of integer;
writeToIndex : integer;
indexR : array [0..N-1] of integer;
indexW : integer;
i : integer;
procedure putItem(item : integer);
begin
  while (num[writeToIndex] <> N) do
    writeCondition.wait;
  buffer[writeToIndex] := item;
  num[writeToIndex] := 0;
```

```

readCondition[writeToIndex].signal_all;
indexW := indexW + 1;
writeToIndex := (writeToIndex + 1) mod B;
end;
function getItem(id : integer) : integer;
begin
  while(indexW = indexR[id]) do
    readCondition[readFromIndex[id]].wait;

  getItem := buffer[readFromIndex[id]];
  num[readFromIndex[id]] := num[readFromIndex[id]] + 1;
  if num[readFromIndex[id]] = N then writeCondition.signal;

  indexR[id] := indexR[id] + 1;
  readFromIndex[id] := (readFromIndex[id] + 1) mod B;
end;
begin
  for i := 0 to B-1 do num[i] := N;
  for i := 0 to N-1 do readFromIndex [i] := 0;
  writeToIndex := 0;
  for i := 0 to N-1 do indexR [i] := 0;
  indexW := 0;
end.

```

б) Као и у претходној варијанти процедуре за смештање производа у бафер, *putItem(item : integer)*, као аргумент прима производ, цео број, који је потребно сместити у бафер, то јест проследити га свим потрошачима. На почетку ове процедуре произвођач увећава за 1 променљиву *numWIteration* која представља редни број производа који се смешта у бафер. Уколико се тај број разликује од броја колико сме да смести у бафер онда произвођач чека да се ослободи место на које треба убацити нови производ. Чекање се обавља на условној променљивој *writeCondition* на којој чека само произвођач. Када произвођач добије дозволу за смештање податка у бафер он га смешта на одговарајућу локацију. Овде не постоји индекс који представља локацију на коју треба сместити производ него се локација сваки пут срачунава. Након смештања производа у бафер произвођач сваког потрошача обавештава да сме да покупи производ у свој следеће локације која год да је. Ово обавештавање се постиже у два корака. У првом кораку се бројач производа колико сваки потрошач посебно сме да конзумира увећава за 1, *rIteration[i] := rIteration[i] + 1*. Након увећања се сваки потрошач посебно покреће уколико је био блокиран, *readCondition[i].signal*.

Процедура за узимање производа из бафера *getItem(id : integer, var item : integer)*, као аргумент прима идентификацију потрошача који узима производе из бафера, и променљиву *item* у којој ће се налазити повратна вредност. На почетку ове процедуре потрошач са идентификацијом *id* увећава за 1 променљиву *numRIteration[id]* која представља редни број производа који сме да узмем из бафера. Уколико се тај број разликује од броја колико је до тог тренутка смео да узмем из бафера онда потрошач чека да се попуни место са кога треба узети нови производ. Чекање се обавља на условној променљивој *readCondition[id]* на којој чека само потрошач са идентификацијом *id*. Када потрошач добије дозволу за узимање податка из бафера он га смешта у променљиву *item* са одговарајуће локације у бафери. Ни овде не постоји индекс који представља локацију са које треба узети производ, него се локација сваки пут срачунава на основу броја до тог тренутка узетих производа. Као следећи корак потребно је увећати бројачку променљиву која говори о томе колико је пута производ конзумиран. Пошто се приступ обавља користећи индексе за потрошача са идентификацијом *id* добија се следећи израз *num[numRIteration[id] mod B] :=*

$numRIteration[id] mod B] + 1$ . Уколико се ради о  $N$ -том потрошачу који је приступио датом производу произвођачу се дозвољава да произведе још један производ увећавањем променљиве  $wIteration$  за 1. Након увећања ради се ресетовање бројача приступа и обавештава произвођач, уколико је био блокиран, да сме да убаци нови производ,  $writeCondition.signal$ .

Заједничке променљиве које се користе приликом решавања овог проблема су: бафер  $buffer$  капацитета  $B$  за смештање производа, низ бројача  $num$  придружених свакој локацији у бафери у којима се чува колико је пута одређени производ конзумиран, низ бројача  $rIteration$  за сваког потрошача у којима се чува колико је смо производа да конзумира, низ бројача  $numRIteration$  за сваког потрошача у којима се чува колико је укупно производа до тог тренутка конзумирао, бројач  $wIteration$  у коме се чува колико је укупно производа произвођач смео да смести у бафер, и бројач  $numWIteration$  који каже колико је производа произвођач до сада сместио у бафер. Да би систем могао да почне да функционише потребно је дозволити произвођачу да може да смешта производе на сваку локацију почев од почетне локације. Да би се ово постигло потребно је сваки бројач приступа поставити на вредност 0 ( $for i := 0 \text{ to } B-1 \text{ do } num[i] := 0$ ). Бројаче укупног броја конзумираних и броја одобрених производа за конзумирање треба поставити на 0. Бројач колико производа сме да смести у бафер треба поставити на капацитета бафера  $B$ , јер је бафер на почетку празан, а бројач смештених производа на 0 јер пре почетка рада ничега није било у баферу.

```

AtomicBroadcast : monitor;
const N = ...;
      B = ...;
var  buffer: array [0..B-1] of integer;
      num: array [0..B-1] of integer;
      readCondition: array [0..N-1] of condition;
      writeCondition: condition;
      rIteration : array [0..N-1] of integer;
      wIteration : integer;
      numRIteration : array [0..N-1] of integer;
      numWIteration : integer;
      i : integer;

procedure putItem(item : integer);
begin
    numWIteration := numWIteration + 1;
    if (numWIteration > wIteration) then writeCondition.wait;
    buffer[numWIteration mod B] := item;
    for i := 0 to N-1 do
    begin
        rIteration[i] := rIteration[i] + 1;
        if(not readCondition[i].empty) then
            readCondition[i].signal;
    end;
end;

procedure getItem(id : integer; var item : integer);
begin
    numRIteration[id] := numRIteration[id] + 1;
    if (numRIteration[id] > rIteration[id]) then
        readCondition[id].wait;
    item := buffer[numRIteration[id] mod B];
    num[numRIteration[id] mod B]

```

```
        := num[numRIteration[id] mod B] + 1;
if (num[numRIteration[id] mod B] = N) then
begin
    wIteration := wIteration + 1;
    num[numRIteration[id] mod B] := 0;
    if(not writeCondition.empty) then
        writeCondition.signal;
end;
end;
begin
    for i := 0 to B-1 do num[i] := 0;
    for i := 0 to N-1 do rIteration[i] := 0;
    for i := 0 to N-1 do numRIteration[i] := 0;
    wIteration := B;
    numWIteration := 0;
end.
```

## 36 Алокација ресурса

Потребно је реализовати монитор који ће обезбедити алокацију ресурса коришћењем две процедуре – *request* и *release*. Алокација се обавља на следећи начин: Укупно постоји количина од три ресурса истог типа који су иницијално слободни. Процеси који траже ресурсе могу да захтевају највише количину од 2 ресурса у процедуре *request*. Уколико постоји тражена количина ресурса процес је узима, а уколико не процес чека на ослобађање ресурса. Ако постоје процеси на чекању за ресурсе, а долази до ослобађања ресурса, примењује се следећа логика: ако постоје процеси који чекају и тражили су количину од 2 ресурса, а расположива је количина од бар два слободна ресурса, прво један од тих процеса добија пар ресурса (имају приоритет у односу на процесе који траже један ресурс). Након испитивања те категорије процеса на чекању, ако је бар један ресурс расположив, а постоје процеси на чекању који траже количину од само једног ресурса, додељују се ресурси тој категорији процеса на чекању, док има слободних ресурса и/или процеса на чекању. Међу процесима који траже исти број ресурса, примењује се *shortest next allocation* логика. Увек се ослобађа исти број ресурса који је био алоциран. Написати монитор под претпоставком да важи *Signal and Wait* дисциплина, и да се користе редови за чекање са приоритетима на условној променљивој.

## Решење

Процедура за алокацију ресурса, *request(amount, time : integer)*, као аргументе прима број тражених ресурса и процењено време колико ће ти ресурси бити заузети. Пошто се овде ради о коначном скупу ресурса, 3, као и о максималном броју ресурса који се могу захтевати решење је прилагођено за ту прилику. На почетку се проверава да ли је тражено два ресурса и да ли има слободно два ресурса, више од 1, уколико је услов испуњен оба ресурса се додељују датом процесу, а укупан број расположивих ресурса се умањује за 2. Уколико се тражило два ресурса а слободно их је мање од два процес се блокира на услову *turn2* на коме се блокирају сви они који чекају два ресурса. Блокирање се ради користећи приоритетне редове чекања, који се сортирају према процењеном времену задржавања у реду (*shortest next allocation* логика). Када се процес пробуди, пошто се ради о дисциплини *Signal and Wait* може бити сигуран да су још увек доступна одблокирана два ресурса. Уколико није било захтева за два ресурса онда се ради о захтеву за једним ресурсом и прелази се на проверу да ли је и тај један ресурс слободан. Уколико јесте узима се и умањује број расположивих ресурса за 1. Уколико није процес се блокира на услову *turn1* на коме се блокирају сви они који чекају један ресурс. Када се пробуди процес умањује број расположивих ресурса.

Процедура за ослобађање ресурса, *release(amount : integer)*, као аргумент прима број заузетих ресурса који се ослобађају. На почетку се увећа број расположивих ресурса. Након ослобађања се прелази на проверу да ли је број слободних ресурса барем два и да ли постоје процеси који чекају ослобађање два ресурса. Уколико постоји буди се само један процес. Више од једног ни не може јер је могло да се ослободи само два ресурса. Одмах се напушта монитор не проверавајући да ли постоји неки процес који чека један ресурс. Сигурно не постоји јер су овом приликом заузета два. Уколико нема оних са два прелази се на проверу има ли процеса који чекају један ресурс. Уколико има буде се све док је испуњен услов  $((not(turn1.empty)) \text{ and } (available > 0))$ . Пошто се ради о дисциплини *Signal and Wait* пробуђени процеси мењају број слободних ресурса, а не онај који их буди.

На почетку се систем поставља тако да има 3 слободна ресурса.

```
Shortest_Job_Next : monitor;
var
    available : integer;
    turn2, turn1 : condition;

procedure request(amount, time : integer);
begin
    if((amount > 1) and (available > 1)) then
        available := available - 2
    else if((available < 2) and (amount > 1)) then
    begin
        turn2.wait(time);
        available := available - 2;
    end
    else if (available > 0) then available := available - 1
    else
    begin
        turn1.wait(time);
        available := available - 1;
    end;
end;

procedure release(amout : integer);
begin
    available := available + amout;
    if(available > 1) and not (turn2.empty)) then turn2.signal
    else if(not turn1.empty) then
        while((not turn1.empty) and (available > 0)) do
            turn1.signal;
end;
begin
    available := 3;
end.
```

## 37 Кружни ток

Раскрница кружног тока има 3 двосмерне улице са по једном саобраћајном траком у сваком смеру повезане на кружни ток. Предност у кружном току имају возила која се већ налазе у кружном току. Реализујте монитор у коме ће се регулисати саобраћај кружног тока. Сматрајте да се, ако у претходном сегменту кружног тока има возила, мора чекати на улазу у кружни ток. Такође, возила морају чекати возила испред себе на улазној улици у кружни ток (једна улазна саобраћајна трака). У решењу, непотребно задржавање процеса за возила у монитору није дозвољено. Позивање више различитих мониторских процедура за процесе возила је дозвољено. Смер кретања у кружном току је прва\_ул-друга\_ул-трета\_ул-прва\_ул. Претпоставите да сегмент кружног тока може да прими неограничен број возила. Користити мониторе који имају *Signal and Wait* дисциплину.

### Решење

Приликом пројектовања монитора потребно је, уколико је то могуће, јасно раздвојити делове који се односе на синхронизацију и делове који се односе на обраду која је у складу са проблемом који се решава. Монитори треба да обезбеде синхронизацију и међусобно искључивање процеса а да притом воде рачуна о конкурентности решења. Конкурентност решења се огледа у томе колико процеса може истовремено да обавља обраду у складу са датим проблемом. Конкурентност се не огледа у томе колико процеса може у неком тренутку да приступи монитору, јер је тај број самом конструкцијом монитора ограничен на 1. У складу са овим разматрањем реченице које су дате у поставци задатка: „У решењу, непотребно задржавање процеса за возила у монитору није дозвољено. Позивање више различитих мониторских процедура за процесе возила је дозвољено.“ јесу сувишне јер се од сваког конкурентног програма који користи мониторе очекује да се понаша у складу са њима.

Уколико би код проблема раскрнице постојала процедура која би означавала кретање од улаза у раскрницу до излаз из раскрнице, такво решење не би било добро. Разлог за ово би био тај што би се само кретање кроз раскрницу обављало у монитору, треба напоменути да ово кретање представља обраду и циљ је да она обавља конкурентно са другим таквим обрадама. Са оваквом процедуром процес би непотребно дugo задржавао монитор, а притом би само једно возило могло бити у кружном току раскрнице.

Како би се скратило време које је потребно провести у монитору могуће је решење би се могло организовати на тај начин што би се направиле процедуре од улаза сегмента до излаза тог сегмента, у којима би се задржавао монитор све време док возило путује кроз сегмент. Овакво решење не би било конкурентније у односу на прво јер би још увек само један процес могао да приступи монитору. Дошло би до тога да многи процеси чекају да започну мониторску процедуру, па би чекања када има више возила била примарно због задржавања у улазном реду монитора.

На основу овог разматрања може се закључити да само кретање кроз сегмент треба издвојити изван монитора а у монитору треба да постоје процедуре тражења и враћања дозволе. Ако се разматра ситуација у којој возило проплази кроз само један сегмент у монитору треба да постоје процедуре тражење дозволе за улазак у сегмент и процедура за напуштање сегмента. У случају да је потребно да возило пређе кроз већи број сегмената потребно је направити и процедуру која би овај прелазак потврдила, а само кретање би се обавило изван монитора. Овде треба приметити разлику између процедуре за прелазак између сегмената и процедуре за улазак у

сегмент. Између сегмената се може увек прећи, прелазак није блокирајући, док се приликом уласка чека да се одговарајући сегмент испразни, улазак је блокирајући.

Примери коришћења монитора:

Уколико је потребно ући у првом сегменту и изаћи на другом сегменту онда аутомобил тражи од монитора дозволу за улазак у први сегмент, вози кроз други сегмент (изван монитора), и обавештава монитор да излази из другог сегмента.

```
procedure Carl_2;
var section : Roundabout;
begin
  ...
  section.start(1); {starts at 1, enters 2}
  drive(2);
  section.leave(2);
  ...
end.
```

Уколико је потребно ући у првом сегменту и изаћи на трећем сегменту аутомобил прво тражи од монитора дозволу за улазак у први сегмент, вози кроз други сегмент (изван монитора), и обавештава монитор да прелази из другог у трећи сегмент, вози кроз трећи сегмент (изван монитора), и на крају обавештава монитор да излази из трећег сегмента.

```
procedure Carl_3;
var section : Roundabout;
begin
  ...
  section.start(1); {starts at 1, enters 2}
  drive(2);
  section.movefrom(2);
  drive(3);
  section.leave(3);
  ...
end.
```

Код овог монитора је потребно обезбедити да процес који је раније дошао на улаз раскрснице има предност над процесом који је дошао касније. Како би се ово реализовало може се искористити особина да су редови чекања на условним променљивама код монитора по дефиницији FIFO. Потребно је сваком улазу у раскрсницу доделити по једну условну променљиву на којој би се процеси блокирали. Процес се приликом долaska на раскрсницу блокира ако је сегмент у који улази попуњен, али и ако је неко које дошао раније већ блокиран.

Уколико редови чекања нису FIFO (као што постоји код неких имплементација), а жели се постићи овај редослед буђења, онда се сваком процесу пред само блокирање мора доделити јединствени редни број приступа. Овај број би се користио као параметар приликом блокирања како би се гарантовао редослед буђења. Како би се ово постигло потребно је увести посебну мониторску сталну променљиву која би се инкрементирала сваки пут када се неки процес блокира.

```
Roundabout : monitor;
const N = 3;
var i : integer;
var count : array[1..N] of integer;
enter : array[1..N] of condition;

procedure start(segment : integer);
begin
```

```

if ((enter[segment].queue) or (count[segment] <> 0)) then
    enter[segment].wait;
count[(segment mod N)+1] := count[(segment mod N)+1] + 1;
end;

procedure leave(segment : integer);
begin
    count[segment] := count[segment] - 1;
    while ((enter[segment].queue) and (count[segment] = 0)) do
        enter[segment].signal;
end;

procedure movefrom(segment : integer);
begin
    count[(segment mod N)+1] := count[(segment mod N)+1] + 1;
    count[segment] := count[segment] - 1;
    while ((enter[segment].queue) and (count[segment] = 0)) do
        enter[segment].signal;
end;

begin
    for i := 1 to N do count[i] := 0;
end.

```

Треба приметити да се процедуре за прелазак између сегмената сastoји из два дела у првом делу се увећава броја возила у сегмент на који се прелази а у другом делу се обавештавају возила која чекају на улазак у сегмент који се напушта. Први део доста личи на процедуру за улазак из сегмента, али са том разликом што се не чека ред приликом уласка у сегмент. Други део је исти као процедура за излазак из сегмента тако да се она може позвати уместо последња два реда.

```

procedure movefrom(segment : integer);
begin
    count[(segment mod N) + 1] := count[(segment mod N) + 1] + 1;
    leave(segment);
end;

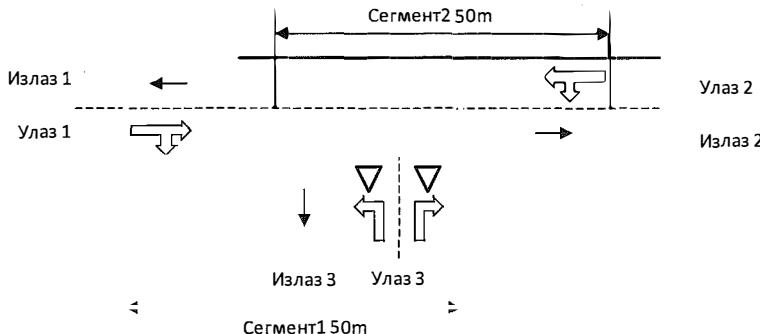
```

## 38

## Т раскрсница

Раскрсница има облик слова Т. Возила која продужавају истим правцем и смером кроз раскрсницу имају предност (главна улица). За случај када се из главне улице скреће лево, ако у сегменту од 50 м испред нема возила из супротног смера, може се скренути. У супротном се чека, али се сматра да је улица доволно широка да се не кочи саобраћај по правцу због возила која скрећу лево, а чекају. Улазак из улице из које се може скретати само лево или десно (споредна улица) се може обавити само ако у одговарајућем сегменту или сегментима од 50 м нема возила која би спречавала улазак у раскрсницу, јер она има троугао. Свака коловозна трака има само једну саобраћајну траку, изузев из споредне улице, где постоје по једна улазна саобраћајна трака за лево и за десно скретање. Написати мониторске процедуре неопходне да се возила крећу из свих улазних у све излазне коловозне траке. Написати како ће клијенти позивати те процедуре за свих 6 могућих кретања возила кроз раскрсницу (нема заокрета 180°). Претпоставити да нема пешака и посматрају се само два сегмента од 50 м са слике, када се наводе сегменти. За време путовања кроз сегменте до саме раскрснице се не сме задржавати монитор. Потребно је обезбедити FIFO редослед на чекању у коловозним тракама. Користити *Signal and Wait* дисциплину.

- Пролазак кроз раскрсницу се сматра тренутним.
- Пролазак кроз раскрсницу се не сматра тренутним, али је краћи од времена потребног за пролазак кроз сегмент до раскрснице.



Слика 7. Изглед Т раскрснице

**Решење**

Како и код проблема кружног тока и овде је потребно јасно раздвојити делове који се односе на синхронизацију и делове који се односе на обраду. Исто тако треба водити рачуна о конкурентности (броју истовремених обрада) и што краће задржавати монитор. Уколико би код проблема Т раскрснице постојала процедура која би означавала кретање од улаза у раскрсницу до излаз из раскрснице такво решење не би било добро. Према дефиницију у монитору може бити само један активно процес што би значило да обраду/кретање обавља само једно возило.

Возила која долазе из улаза 1 и 2 треба да уђу у сегмент, возе се неко време кроз сегмент, онда одлуче да ли настављају право или скрећу, пролазе кроз раскрсницу и излазе из раскрснице. Што се возила која долазе из улаза број 3 тиче она не могу да наставе са вожњом док им се не допусти да скрену и пређу преко неког сегмента, онда

пролазе раскрснику и излазе из ње. На основу овог разматрања може се закључити да само кретање кроз сегмент и пролазак кроз раскрснику треба издвојити изван монитора а у монитору треба да постоје процедуре уласка у сегмент, тражења дозволе и излaska.

а) Пошто се време кретања кроз раскрснику може сматрати тренутним могуће је спојити позиве за тражење дозволе за скретање и за излазак из раскрснице. Овде треба приметити разлику између процедуре за улазак у сегмент и процедуре за улазак у раскрснику, односно излазак из сегмента. Улазак у сегменте, из улаза 1 и 2, се може увек остварити, тако да ова процедура није блокирајућа. Приликом проласка кроз раскрснику се чека да се одговарајући сегмент или сегменти испразне, ово значи да процедура за излазак из раскрснице у неким случајевима блокирајућа.

Примери коришћења монитора:

Уколико је потребно да возило дође на улаз број 1, уђе у сегмент 1 и изађе кроз излаз број 1 или број 3:

```
procedure Carl_x;
var junction : TJunction;
begin
  ...
  junction.start(1, x);
  drive();
  junction.leave(1, x);
  ...
end.
```

Аналогно коришћење би био и за возило које долази на улаз број 2. Разлика која постоји између ова два случаја би била приликом излaska из раскрснице. Излазак из раскрснице за возило које долази из улаза 1 никада није блокирајући, док излазак из раскрснице за возило које долази из улаза 2 и жели да изађе на излазу 3 може да буде блокирајући јер се прелази преко сегмента 1.

Уколико је потребно да возило дође на улаз број 3, уђе у раскрснику и изађе кроз излаз број 1 или број 2:

```
procedure Car3_x;
var junction : TJunction;
begin
  ...
  junction.leave(3, x);
  ...
end.
```

Монитор треба да обезбеди да процес који је раније дошао на улаз раскрснице има предност над процесом који је дошао касније. Како би се ово реализовало могу се искористити приоритетни редови чекања на условним променљивама и принцип доделе приоритета као код тикет алгоритма. Потребно је сваком улазу у раскрснику на коме се возило може блокирати доделити по једну условну променљиву на којој би се возила блокиравала. Возило се приликом доласка на раскрснику блокира ако је сегмент преко кога се пролази попуњен, али и ако је неко које ту дошао раније већ блокиран. Ово значи да возило које је дошло из улаза 2 сме да изађе на излаз 3 ако нема возила у сегменту 1 и нема других возила која чекају да прођу истим путем. Возило које је дошло из улаза 3 сме да изађе на излаз 2 ако нема возила у сегменту 1 и нема других возила која чекају да прођу истим путем. Возило које је дошло из улаза 3 сме да изађе на излаз 1 ако нема возила у сегменту 1, возила у сегменту 2 и нема других возила која чекају да прођу истим путем. На крају возило сме да пусти блокирано возило ако су испуњени исти они услови који потребни да би возило из тог смера прошло раскрснику. Ово значи да се пушта возило које чека да из 3 пређе у 2 ако у сегменту 1 нема возила и постоји возило које чека дати прелаз. Пушта се возило које чека да из 2 пређе у 3 ако

у сегменту 1 нема возила и постоји возило које чека дати прелаз, и на крају се пушта возило које чека да из 3 пређе у 1 ако у сегментима 1 и 2 нема возила и постоји возило које чека дати прелаз.

```
TJunction : monitor;

const S2L = 1; S3R = 2; S3L = 3;

var segment : array[1..2] of integer;
    enter : array[1..3] of condition;
    next : integer;

procedure start(enterRoad : integer; exitRoad : integer);
var turn : integer;
begin
    case enterRoad of
        1: begin
            segment[1] := segment[1] + 1;
        end;
        2: begin
            segment[2] := segment[2] + 1;
        end;
        else ;
    end;
end;

procedure leave(enterRoad : integer; exitRoad : integer);
begin
    case enterRoad of
        1: begin
            segment[1] := segment[1] - 1;
            test;
        end;
        2: begin
            if(exitRoad = 3) then begin //from 2 to 3
                if(segment[1] <> 0 or enter[S2L].queue) then
                    begin
                        turn := next;
                        next := next + 1;
                        enter[S2L].wait(turn);
                    end;
            end;
            segment[2] := segment[2] - 1;
            test;
        end;
        3: begin
            if(exitRoad = 2) then begin //from 3 to 2
                if(segment[1] <> 0 or enter[S3R].queue) then
                    begin
                        turn := next;
                        next := next + 1;
                        enter[S3R].wait(turn);
                    end;
            end
            else begin //from 3 to 1

```

```

if(segment[1] <> 0 or segment[2] <> 0 or
    enter[S3L].queue) then begin
    turn := next;
    next := next + 1;
    enter[S3L].wait(turn);
end;
end;
test;
end;
else ;
end;
end;

procedure test;
begin
    if(segment[1] = 0 and enter[S3R].queue) then enter[S3R].signal;
    if(segment[1] = 0 and enter[S2L].queue) then enter[S2L].signal
    else if(segment[1] = 0 and segment[2] = 0 and
        enter[S3L].queue) then enter[S3L].signal;

    if(enter[S2L].empty and enter[S3L].empty and
        enter[S3R].empty) then next := 0;
end;

begin
    segment[1] := 0; segment[2] := 0; next := 0;
end.

```

Треба приметити да се код овог решења у процедуре *test* променљива *next* поставља на вредност 0 уколико ни на ком скретању нема возила која чекају. Ово је урађено како би се обезбедило да приликом инкрементирања ове променљиве не дође до прекорачења опсега дате променљиве.

б) Попшто се време кретања кроз раскрснику не може сматрати тренутним потребно је обезбедити три мониторске процедуре: прва је за улазак у секцију, друга за улазак у раскрснику (скретање кроз секцију) и изласка из раскрснице. Разлика у односу на претходно решење се огледа у томе шта је претходну процедуру за улазак у раскрснику са преласком преко неког сегмента потребно поделити на два дела. У првом делу се тражи дозвола за прелазак и блокира ако пролазак није тренутно могућ док се у другом делу завршава прелазак и буде блокирани процеси.

Примери коришћења монитора:

Уколико је потребно да возило дође у улаз број 1, уђе у сегмент 1 и изађе кроз излаз број 2:

```

procedure Carl_2;
var junction : TJunction;
begin
    ...
    junction.enterSegment(1, 2);
    drive();
    junction.leave(1, 2);
    ...
end.

```

Аналогно коришћење би био и за возило које долази у улаз број 2 и одлази излазом број 1. Уколико је потребно да возило дође у улаз број 2, уђе у сегмент 2 и изађе кроз излаз број 3:

```
procedure Car2_3;
var junction : TJunction;
begin
  ...
  junction.enterSegment(2, 3);
  drive();
  junction.crossSegment(2, 3);
  drive();
  junction.leave(2, 3);
  ...
end.
```

Уколико је потребно да возило дође у улаз број 3, уђе у раскрницу и изађе кроз излаз број 1 или број 2:

```
procedure Car3_x;
var junction : TJunction;
begin
  ...
  junction.crossSegment(3, x);
  drive();
  junction.leave(3, x);
  ...
end.
```

Остаје да се види још возило које долази на улаз број 1, уђе у сегмент 1 и изађе кроз излаз број 3. Овај код би, због скретања, потенцијално могао да буде аналоган коду када возило долази на улаз 2 и скреће на излаз 3. Али је он ипак аналоган коду код кога возило долази на улаз 1 и излази на излаз 2. Возило које скреће десно из секције 1 има предност у односу на друга возила и нема кога да чека. Такође је по поставци проблема време пролазак кроз раскрницу краће од времена потребног за пролазак кроз сегмент до раскрнице што значи да нема других возила на том месту која скрећу из секције 2.

Возило се приликом скретања на раскрницу блокира ако је сегмент преко кога се пролази попуњен, али и ако је неко ко је ту дошао раније већ блокиран. Ово значи да возило које је дошло из улаза 2 сме да изађе на излаз 3 ако нема возила у сегменту 1, нема других возила која чекају да прођу истим путем и нема возила која из сегмента 3 прелазе у сегмент 1. Возило које је дошло из улаза 3 сме да изађе на излаз 2 ако нема возила у сегменту 1 и нема других возила која чекају да прођу истим путем. Возило које је дошло из улаза 3 сме да изађе на излаз 1 ако нема возила у сегменту 1, возила у сегменту 2, нема других возила која чекају да прођу истим путем и нема возила која из сегмента 2 прелазе у сегмент 3.

```
TJunction : monitor;

const S2L = 1; S3L = 2; S3R = 3;

var segment : array[1..2] of integer;
junctions : array[1..2] of integer;
enter : array[1..3] of condition;
next : integer;
```

```

procedure enterSegment(enterRoad : integer; exitRoad : integer);
var turn : integer;
begin
  case enterRoad of
    1: begin
      segment[1] := segment[1] + 1;
    end;
    2: begin
      segment[2] := segment[2] + 1;
    end;
    else ;
  end;
end;

procedure crossSegment(enterRoad : integer; exitRoad : integer);
var turn : integer;
begin
  case enterRoad of
    1: begin
    end;
    2: begin
      if(exitRoad = 3) then begin //from 2 to 3
        if(segment[1] <> 0 or enter[S2L].queue
           or junctions[S3L] <> 0) then begin
          turn := next;
          next := next + 1;
          enter[S2L].wait(turn);
        end;
        junctions[S2L] := junctions[S2L] + 1;
      end;
    end;
    3: begin
      if(exitRoad = 2) then begin //from 3 to 2
        if(segment[1] <> 0 or enter[S3R].queue) then
          begin
            turn := next;
            next := next + 1;
            enter[S3R].wait(turn);
          end;
      end
      else begin //from 3 to 1
        if(segment[1] <> 0 or segment[2] <> 0 or
           enter[S3L].queue or junctions[S2L] <> 0) then
          begin
            turn := next;
            next := next + 1;
            enter[S3L].wait(turn);
          end;
        junctions[S3L] := junctions[S3L] + 1;
      end;
    end;
    else ;
  end;
end;

```

```
procedure leave(enterRoad : integer; exitRoad : integer);
begin
    case enterRoad of
        1: begin
            segment[1] := segment[1] - 1;
            test;
        end;
        2: begin
            segment[2] := segment[2] - 1;
            if(exitRoad = 3) then begin //from 2 to 3
                junctions[S2L] := junctions[S2L] - 1;
            end;
            test;
        end;
        3: begin
            if(exitRoad = 1) then begin //from 3 to 2
                junctions[S3L] := junctions[S3L] - 1;
            end;
            test;
        end;
        else ;
    end;
end;

procedure test;
begin
    if(segment[1] = 0 and enter[S3R].queue) then enter[S3R].signal;
    if(segment[1] = 0 and enter[S2L].queue and
       junctions[S3L] = 0) then enter[S2L].signal;
    if(segment[1] = 0 and segment[2] = 0 and enter[S3L].queue and
       junctions[S2L] = 0) then enter[S3L].signal

    if(enter[S2L].empty and enter[S3L].empty and
       enter[S3R].empty) then next := 0;
end;

begin
    segment[1] := 0; segment[2] := 0; next := 0;
    junctions[S2L] := 0; junctions[S3L] := 0;
end.
```

### Задаци за вежбу

→ Приликом уписа студената на факултет истовремено се ради на неколико уписних места. На сваком уписном месту ради један службеник који за тај посао користи посебан програм. Том приликом се, између осталог, сваком студенту додељује и број индекса. У случају одустајања неког већ уписаног студента у току трајања уписа, при уписивању нових студената најпре треба попунити упражњене бројеве индекса. Написати монитор који би обезбедио обављање функције уписа, исписа и доделе броја индекса будућем бруцашу, као и основну структуру процеса за службенике који раде на упису.

## Дистрибуирано програмирање

Дистрибуирано програмирање се најчешће користи за програмирање на рачунарским системима са дистрибуираном меморијом. Код њих не постоји заједничка меморија коју деле сви процесори, већ сваки процесор има своју приватну меморију, па међусобно могу да комуницирају само разменом порука преко интерконекционе мреже. Да би размена порука била омогућена у програму, мора се дефинисати одговарајући програмски интерфејс који ће имплементирати примитиве за слање и пријем порука. Он може бити упрошћен, какав би рецимо био интерфејс који дефинише само операције за читање и упис података преко мреже (за приступ подацима који припадају процесима који се извршавају на другим процесорима и не налазе се у локалној меморији) - аналогно операцијама читања и писања над дељеним променљивама. У том случају би, међутим, синхронизација између процеса пошиљаоца и процеса примаоца морала да се ради помоћу запосленог чекања, што је непрактично. Зато се у оквиру комуникационог интерфејса обично дефинишу нешто сложеније примитиве за слање и пријем порука које у себи укључују и синхронизацију. Такве примитиве за слање и пријем порука могу се сматрати неком врстом семафора који осим што врше синхронизацију уједно преносе и податке. Комуникација, тј. прослеђивање порука, између процеса се врши преко комуникационог канала. Канал апстрактује интерконекциону мрежу и физичку везу која постоји између процесора, тако да програм не мора да зна ништа о хардверској архитектури система.

Конкурентни програми који користе прослеђивање порука називају се и *дистрибуирани ћротами*, јер процеси могу бити распоређени и извршавати се на више процесора у оквиру рачунарског система са дистрибуираном меморијом. Међутим, дистрибуирани програм се може исто тако извршавати и на систему са заједничком меморијом, реализацимањем комуникационих канала помоћу дељених променљивих<sup>6</sup>; ово је још један пример који показује да су програмска парадигма и извршна платформа у суштини независни. Наравно, популарност примене неке парадигме је увек условљена ефикасношћу извршавања на датој платформи.

У зависности од тога какви су комуникациони канали (једносмерни или двосмерни), како се именују (директно или индиректно) и како се врши синхронизација приликом слања и пријема порука (синхроно или асинхроно), могу се креирати различити програмски интерфејси и одговарајуће примитиве за комуникацију путем прослеђивања порука. Нека програмска парадигма може, наравно, садржати и примитиве за реализацију слања и пријема порука преко више различитих типова канала. У овом делу књиге ће бити представљено неколико парадигми које имплементирају примитиве за већину од набројаних варијанти комуникације.

*Communicating Sequential Processes* (CSP) је програмски модел којим је представљена програмска парадигма са синхронним слањем и синхронним (блокирајућим) пријемом порука, уз коришћење директног именовања и двосмерних канала. Код слања и пријема порука користи се једна врста једноставног препознавања шаблона (*pattern matching*) како би се лакше структурирале поруке и како би се једноставније приступало њиховим компонентама код пријема, обезбеђујући уједно да процес прималац не прима поруке које не одговарају специфицираном шаблону.

Парадигма програмирања путем јавног емитовања (*broadcasting*) подразумева постојање комуникационог канала који истовремено повезује све процесе. Комуникација се одвија тако што процес пошиљалац истовремено шаље поруку свим

<sup>6</sup> Могуће је и обратно, да се дељене променљиве имплементирају на дистрибуираној платформи.

процесима, а сваки процес прималац одлучује за себе да ли ће је прихватити или не. Ова парадигма је представљена помоћу *Broadcasting Sequential Processes* (*BSP*) програмског модела. Поред јавног емитовања, *BSP* модел нуди и могућност вишеструког емитовања (*multicasting*) и директне (*point-to-point*) комуникације између два процеса. Комуникациони канал за слање и пријем порука је увек једносмеран, а користи се директно именовање процеса. Јавно емитовање увек подразумева асинхроно слање, јер је нереално очекивати да процес пошиљалац чека да сви процеси буду спремни да приме поруку. Такође, пренос може бити баферован или небаферован. Код небаферованог преноса, порука коју процес прималац пропусти да прими зато што није био спреман бива заувек изгубљена. Код баферованог преноса, поруке које стижу остају у баферау процеса примаоца све док он не буде спреман да их прочита. *BSP* има и посебне примитиве за *point-to-point* синхрону комуникацију.

Програмска парадигма са индиректним именовањем процеса, могућношћу коришћења једносмерних или двосмерних канала, асинхроног слања (са могућношћу додавања временски ограниченог пријема одговора) и синхроним пријемом порука, представљена је помоћу програмског језика *CONIC*. Индиректно именовање процеса се реализује увођењем комуникационих портова, при чему се тип канала (једносмеран или двосмеран) дефинише приликом декларације порта. Временски ограничено (користи се и термин условљено) асинхроно слање порука врши се помоћу опционе *wait* примитиве унутар *send* команде, која омогућава временски ограничено блокирање процеса пошиљаоца код чекања на одговор.

Парадигма програмирања која користи рандеву за комуникацију између процеса представљена је помоћу програмског језика *Ada*. Код језика *Ada* паралелизам се заснива на секвенцијалним процесима – тасковима (*tasks*). Рандеву се остварује када процес позивалац стигне до тачке слања позива, а процес прималац стигне до тачке прихватања позива, тј. кад се оба процеса „сретну” (отуд и назив рандеву).

## Увод у програмирање прослеђивањем порука

Како што је речено у уводу о дистрибуираном програмирању, примитиве за слање и пријем порука могу бити реализоване са различитим нивоом комплексности. У пракси се редовно користе примитиве које поред преноса података садрже у себи и синхронизационе елементе. Генерално, примитиве за слање порука могу бити синхроне или асинхроне. Код синхроног слања, процес који шаље поруку се блокира и чека док процес коме је порука послата ту поруку не прими. Процес прималац притом потврђује пријем поруке слањем поруке потврде. Код асинхроног слања, процес који шаље поруку одмах наставља са радом, не чекајући да порука буде примљена. Код синхроног пријема, процес који прима поруку се блокира и чека док порука не стигне. Код асинхроног пријема, процес прималац проверава да ли је порука стигла и одмах наставља са радом, без обзира да ли је порука стигла или не.

Код тзв. временски ограниченог слања, процес се блокира чекајући потврду пријема поруке, слично као код синхроног слања. Међутим, за разлику од синхроног слања, ако потврда о пријему поруке није примљена у одговарајућем року (енглески *timeout*), процес пошиљалац наставља са радом, као код асинхроног слања. Слично томе, код временски ограниченог пријема, процес прималац се блокира чекајући поруку. Ако порука не стигне у одговарајућем року, процес прималац наставља са радом. Можемо рећи да је синхрона и асинхронна комуникација екстремни случај временски ограниченог слања и пријема порука, где је време чекања нула, односно бесконачно.

Синхрона комуникација је по природи једноставнија за програмирање, али се временски ограничена комуникација уводи због смањивања осетљивости система на отказе. Када не би било временског ограничења, у случају прекида канала процеси који комуницирају би трајно остали блокирани.

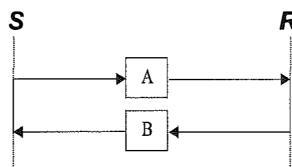
## 39

## Синхрона, асинхрони и условна комуникација

Комуникациони сервиси у једном дистрибуираном систему реализовани су на бази сандучића типа *mbx*. Претпоставићемо да тип *mbx* обухвата целе бројеве и специјални симбол *ack* за потврђивање пријема. Основне операције на сандучићима су следеће: *mbx\_put(m: msg, box: mbx)* која смешта поруку *m* у сандуче *box* и *mbx\_get(var m: msg, box: mbx, t: time, var status: bool)* која узима прву поруку из сандучета *box* и њену вредност додељује променљивој *m*, постављајући статус на *true*; ако је сандуче празно током интервала *t*, статус постаје *false*, а вредност *m* је недефинисана. Време *t* је у опсегу *0..maxtime* или је дато као бесконачно (специјална вредност *INF*). Размотримо једноставан систем који садржи само два процеса, *S* и *R*. Процеси комуницирају преко сандучића *A* и *B* као на слици.

Реализовати следеће интеракције између процеса:

- S* асинхроно шаље целобројну вредност *i*, а *R* извршава синхрони пријем.
- S* асинхроно шаље целобројну вредност *i*, а *R* извршава условни пријем.
- S* асинхроно шаље целобројну вредност *i*, а *R* извршава временски ограничен пријем (*receive with timeout*) са интервалом *d*.
- S* синхроно шаље целобројну вредност *i*, а *R* извршава синхрони пријем.
- У бидирекционалној трансакцији типа захтев-одговор (*request-reply*), *S* шаље целобројну вредност и добијени резултат додељује променљивој *x*; у случају да трансакција не успе током интервала *d*, *x* добија вредност 0. *R* обрађује захтев тако што за дати аргумент *i* враћа вредност функције *f(i)*.



Слика 8.

## Решење

```

a)
procedure send(i : integer);
var m : msg;
begin
    m.data := i;
    mbx_put(m,A)
end;

b)
procedure send(i : integer);
var m : msg;
begin
    m.data := i;
    mbx_put(m,A)
end;

procedure receive(var i : integer);
var m : msg; st : boolean;
begin
    mbx_get(m,A,INF,st);
    i := m.data
end;

function receive(var i : integer) : boolean;
var m : msg; st : boolean;
begin
    mbx_get(m,A,0,st);
    if st then i := m.data;
    receive := st
end;

```

```

В)
procedure send(i : integer);
var m : msg;
begin
    m.data := i;
    mbx_put(m,A)
end;

function receive(var i : integer,
                 d : time) : boolean;
var m : msg; st : boolean;
begin
    mbx_get(m,A,d,st);
    if st then i := m.data;
    receive := st
end;

Г)
function send(i : integer) :
boolean;
var m : msg; st : boolean;
begin
    m.data := i;
    mbx_put(m,A);
    mbx_get(m,B,INF,st);
    send := (m = ack)
end;

Д)
procedure request(i : integer;
                  d : time; var x : integer);
var m : msg; st : boolean;
begin
    m.data := i;
    mbx_put(m,A);
    mbx_get(m,B,d,st);
    if st then x := m else x := 0
end;
procedure reply;
var m : msg; st : boolean;
i : integer;
begin
    mbx_get(m,A,INF,st);
    i := m.data;
    m := f(i);
    mbx_put(m,B)
end;

```

◆ Напомена: У примерима се користи исти назив за сандуче коме приступају различити процеси, али треба приметити да сандучићи у дистрибуираном програмирању нису дељене променљиве у смислу променљивих код система са дељеном меморијом. Сандучић је потребно конфигурисати како би се приступило месту где се поруке смештају, сандуче такође може да користи неки од комуникационих протокола како би се приступило том месту.

Приликом рада са сандучићима полази се од претпоставке да унутар сандучета, места где се поруке чувају, постоји неограничен ред у који се поруке смештају по редоследу пријема. Ово значи да ако је у неко сандуче прво смештена порука *m* па порука *n*, прво ће бити узета порука *m* па порука *n*. Ово не мора значи да ће прво ће бити узета порука *m* па порука *n* ако је прво послата порука *m* па порука *n*. Овде треба имати на уму да је потребно време да се порука коју неки процес проследи испоручи, време између слanja и смештања. У примеру:

```

A:   mbx_put(m, B);
     mbx_put(m, C);
B:   mbx_get(m, B, INF, st);
     mbx_put(m, C);
C:   mbx_get(m, C, INF, st);
     mbx_get(m, C, INF, st);

```

није познато да ли је процес С прво добио поруку од процеса A или процеса B.

Уколико исти процес прво шаље поруку *m* па поруку *n* тада ће сигурно у сандуче прво бити смештена порука *m* па *n*. Треба приметити да је ред могуће и другачије реализовати.

## 40

## Читаоци и писци

Користећи асинхрону комуникацију помоћу сандучића и активне мониторе решити проблем читалаца и писаца.

### Решење

Концепт активних монитора (*Active Monitors*) има за циљ да концепт монитора који се користи код програмирања са дељеним променљивама на неки начин искористи у дистрибуираном програмирању где нема дељених променљивих. Активни монитори врше симулирање понашања монитора тако што се мониторске процедуре не позивају директно, јер нема дељених променљивих, већ се креира један серверски процес и кроз комуникацију са тим сервером се врши индиректно позивање мониторских процедура. Клијентски процеси позивају те мониторске процедуре прослеђивањем одговарајућих порука, захтева, серверу.

Како би се обезбедило да други процеси не могу да приђу мониторским променљивама, те променљиве су локалне променљиве сервера. Сервер води рачуна о овим локалним променљивама, оне се понашају као сталне променљиве, и постоје докле год се извршава серверски процес. Код активних монитора сервер има један ток контроле у коме обавља обраду по принципу догађајима вођеног извршавања (*Event Driven Execution*). Серверски процес у петљи прима захтеве које одговарају позивима мониторских процедура, ове захтеве обрађује и шаље одговор клијентском процесу на дати захтев. Пошто се захтеви обрађују један по један, на овај начин је обезбеђено међусобно искључивање процеса.

Серверски процес кроз један комуникациони канал (сандуче) прима захтеве, а на основу идентификатора који је клијент уградио у послати захтев одређује на који комуникациони канал треба послати одговор. Именовање канала се може извршити статички, када се дефинише се фиксан број канала преко којих се прилази активном монитору. У овом случају клијентски процес који шаље поруку на јединствени серверски канал шаљући свој клијентски идентификатор канала чиме се омогућава да клијент има приватни канал преко кога ће ићи одговор. Именовање канала се може и динамички вршити тако што се у тренутку када се шаље захтев активном монитору на јединствени серверски канал, креира и канал кроз кога ће се примити одговор који сервер шаље.

Типичан изглед сервера који је активни монитор, а који одговара монитору са процедуром *op1*, ..., *opN*, код кога нема условних променљива, као и типичан изглед клијента који позива мониторску процедуру *opj* активног монитор је дат у прилогу.

```
procedure Server;
var сталне променљиве;
begin
    иницијализација сталних променљивих;
    while (true) do begin
        пријем поруке која садржи:
        ID клијента, операцију, аргументу;
        case операција of
            op1 : тело операције op1; end;
            ...
            opN : тело операције opN; end;
        end;
        слање поруке клијенту ID;
```

```
end;  
end;  
  
procedure Client(id : integer);  
begin  
...  
    слање поруке серверу која садржи:  
    ID клијента, операцију, аргументу;  
    пријем поруке која садржи ID клијента и резултате;  
end;
```

У случају да монитори имају и условне променљиве на којима се процеси могу блокирати активни монитори постају сложенији. Разлог за ово усложњавање лежи у чињеници да се активни монитор извршава у једном процесу и да има само један ток контроле. Уколико би се тај ток контроле блокирао на некој условној променљиви цео активни монитор би се блокирао. Због тога је потребно увести додатне структуре података, редове, у којима би се памтило који су захтеви остали неопслужени, док би серверски процес остао слободан за пријем других захтева. Овде треба уочити разлику између редова на условним променљивама на којима су блокирани процеси и редова са неопслуженим захтевима на којима није нико на серверској страни блокиран, блокирани су клијенти који чекају одговор од сервера.

Овде се може увести аналогија између класичних монитора и активних монитора и начина за њихову међусобну трансформацију. Сталне променљиве монитора се пресликају у локалне променљиве серверског процеса. Идентификација мониторских процедура се преслика у постојање долазног комуникационог канала на серверској страни и постојање одговарајућег идентификатора процедуре у *case* исказу. Позив мониторске процедуре се преслика у слање поруке на комуникациони канал која садржи идентификатор пошиљаоца, идентификатор позиване процедуре и аргументе позиване процедуре. Пролазак кроз улазни ред и добијање ексклузивног права приликом уласка у монитор се преслика у извршавање у једној нити где се на почетку петље прима нова порука кроз комуникациони канал. Повратак из мониторске процедуре се преслика на враћање одговора на клијентски комуникациони канал. Блокирање на условној променљиви се преслика на чување неопслужених захтева. Сигнализација на условној променљиви се преслика на дохватање и опслуживање неопслужених захтева. Тело процедуре се преслика на део одговарајуће гране у *case* исказу.

### Прво решење

Полазећи од монитора датих у задатку 27.а) добија се монитор има четири процедуре *startread*, *startwrite*, *endread* и *endwrite*. Ово значи да серверски процес има четири гране у *case* исказу који одговарају овим процедурима. Серверски процес се врти у бесконачној петљи и приhvата и обрађује захтев по захтев. Серверски процес има само једно сандуче, *serverBox*, кроз које прима поруке које су му упућене. Именовање клијентских сандуцића, канала, је извршено статички. Читаоцима су придржани сандуцићи *rBox* у зависности од идентификатора читаоца, а писцима су придржани сандуцићи *wBox* у зависности од њиховог идентификатора.

Мониторским сталним променљивама *readcount* и *busy* одговарају серверске променљиве *readcount* и *busy*. Условним променљивама *OKtoread* и *OKtowrite* одговарају листе неопслужених захтева *OKtoread* и *OKtowrite*.

У оквиру *startread* гране проверава се променљива *busy* и да ли је број елемената у листи неопслужених захтева *OKtowrite* већи од нула. Ово је еквивалентно провери да ли има блокираних процеса на условној променљиви *OKtowrite.queue*. Уколико је услов

испуњен захтев се пребације у листу *OKtoread* и прелази на обраду следећег новог захтева. Ово је еквивалентно блокирању на условној променљиви. Уколико услов није испуњен читалац сме да чита, увећава се број активних читалаца и обавештава се читаоца да сме да почне са читањем. Ово се постиже слањем поруке у сандуче које је специфицирано у поруци. Поред овога поставља индикацију да у следећој итерацији треба да се пређе на обраду неопслуженог захтева из листе *OKtoread* (ако такав постоји) постављањем променљиве *OKtoreadsignal* на вредност *true*. Ово је еквивалентно позиву наредбе *Oktoread.signal*.

У оквиру *endread* гране задовољење услова *readcount=0* значи да је дати процес био последњи који је читao, па се поставља индикација да у следећој итерацији треба да се пређе на обраду неопслуженог захтева из листе *OKtowrite* (ако такав постоји) постављањем променљиве *OKtowritesignal* на вредност *true*. Ово је еквивалентно позиву наредбе *OKtowrite.signal*. Треба приметити да у овом случају није неопходно да се клијенту, читалацу, шаље информација да је операција завршена јер се радило о процедурима која није имала повратну вредност нити се блокирала на некој условној променљиви.

У оквиру *startwrite* гране проверава се променљива *busy* и *readcount<>0*. Уколико је услов испуњен писац нема право да пише и захтев се пребације у листу *OKtowrite* и прелази на обраду следећег новог захтева. Уколико услов није испуњен писац сме да пише, поставља променљиву *busy* и обавештава писца слањем поруке да сме да почне са писањем.

У оквиру *endwrite* гране испитује се да ли има читалаца који чекају на приступ (неопслужених захтева у листи *OKtoread*), у ком случају се поставља индикација да у следећој итерацији треба да се пређе на обраду неопслуженог захтева за читање. У супротном, се у следећој итерацији проверава следећи неопслужени захтев за писањем како би тај писац почeo да пише.

Поруке за обраду се дохватају користећи процедуру *getMsg(m)*. Ова процедура дохвата или нове захтеве или неопслужене захтеве из одговарајуће листе. На основу описа се може закључити да процедура *getMsg(m)* дохвата неопслужени захтев из листе *OKtoread* ако је променљива *OKtoreadsignal* постављена на вредност *true* и ако у тој листи има захтева, дохвата неопслужени захтев из листе *OKtowrite* ако је променљива *OKtowritesignal* постављена на вредност *true* и ако у тој листи има захтева, иначе дохвата нови захтев. На крају се сигналне променљиве поставе на вредност *false*.

```
program ReadersWriters;
const NR = 15;
      NW = 5;
      startRead = 0;
      endRead = 1;
      startWrite = 2;
      endWrite = 3;
var rBox : array [0..NR] of mbx;
    wBox : array [0..NW] of mbx;
    serverBox : mbx;
procedure reading; begin ... end;
procedure writing; begin ... end;

procedure Reader(id : integer);
var m : msg;
    status : boolean;
begin
```

```
while (true) do
begin
    {startRead}
    m.id := id;
    m.opr := startRead;
    mbx_put(m, serverBox);
    mbx_get(m, rBox[id], INF, status);
    reading;
    {endRead}
    m.id := id;
    m.opr := endRead;
    mbx_put(m, serverBox);

end
end;

procedure Writer(id : integer);
var  m : msg;
     status : boolean;
begin
    while (true) do
begin
    {startWrite}
    m.id := id;
    m.opr := startWrite;
    mbx_put(m, serverBox);
    mbx_get(m, wBox[id], INF, status);
    writing;
    {endWrite}
    m.id := id;
    m.opr := endWrite;
    mbx_put(m, serverBox);
end
end;

procedure Server;
var  m : msg;
     status : boolean;
     readcount: integer;
     busy: boolean;
     OKtoread, OKtowrite: list;
     OKtoreadsignal, OKtowritesignal : boolean;
procedure getMsg(var m : msg);
begin
    if(OKtoreadsignal and size(OKtoread)>0) then
        remove(OKtoread, m)
    else if(OKtowritesignal and size(OKtowrite) > 0) then
        remove(OKtowrite, m)
    else mbx_get(m, serverBox, INF, status);
    OKtoreadsignal := false;
    OKtowritesignal := false;
end;
begin
    readcount := 0;
    busy := false;
```

```
OKtoreadsignal := false;
OKtowritesignal := false;
while (true) do
begin
    getMsg(m);
    case m.opr of
        startRead : begin
            if(busy or size(OKtowrite)>0) then put(OKtoread, m)
            else
                begin
                    readcount := readcount + 1;
                    OKtoreadsignal := true;
                    m.opr := ack;
                    mbx_put(m, rBox[m.id]);
                end
            end;
        endRead : begin
            readcount := readcount - 1;
            if (readcount = 0) then OKtowritesignal := true;
            end;
        startWrite : begin
            if(readcount <> 0 or busy) then put(OKtowrite, m)
            else
                begin
                    busy := true;
                    m.opr := ack;
                    mbx_put(m, wBox[m.id]);
                end
            end;
        endWrite : begin
            busy := false;
            if (size(OKtoread) > 0) then OKtoreadsignal := true
            else OKtowritesignal := true
            end;
        end;
    end;
end;
```

#### Друго решење

Проблем читалаца и писаца се може решити и користећи само једну додатну листу неопслужених захтева и без сигнализационих променљивих. Програмски код за читаоце и писце се, као ни код класичних монитора, не мења уколико се промени интерна имплементација монитора јер потписи њихових процедуре остају исти. Оно што је потребно изменити је процедуру за дохватање захтева.

```
program ReadersWriters;
...
procedure Server;
var  m : msg;
    status : boolean;
    msgs : list;
    readcount : integer;
    busy : boolean;
```

```
procedure getMsg(var m : msg);
begin
    if((size(msgs) > 0) and (busy = false) and
       ((readcount = 0) or ((get(msgs))^opr <> startWrite))) then
        remove(msgs, m)
    else
    begin
        mbx_get(m, serverBox, INF, status);
    end
end;

begin
    readcount := 0;
    busy := false;
    while (true) do
    begin
        getMsg(m);
        case m.opr of
            startRead : begin
                if(busy) then put(msgs, m)
                else
                begin
                    readcount := readcount + 1;
                    m.opr := ack;
                    mbx_put(m, rBox[m.id]);
                end
                end;
            endRead : begin
                readcount := readcount - 1;
                end;
            startWrite : begin
                if((readcount <> 0) or busy) then put(msgs, m)
                else
                begin
                    busy := true;
                    m.opr := ack;
                    mbx_put(m, wBox[m.id]);
                end
                end;
            endWrite : begin
                busy := false;
                end;
        end;
    end;
end;
```

## 41

## Произвођачи и потрошачи

Користећи асинхрону комуникацију помоћу сандучића и активне мониторе решити проблем производаца и потрошача (*The Producer – Consumer Problem*).

### Решење

#### Прво решење

Полазећи од монитора датог у задатку 31 добија се активни монитор има две гране, *get* и *put*, у *case* исказу. Користећи поступак трансформације добија се следеће решење овог проблема. Овде је приказан код само за серверску страну, активни монитор, док се код за клијенте, производаче и потрошаче, формира на начин описан у претходном примеру.

```
program ProducerConsumer;
...
procedure Server;
var m : msg;
    status : boolean;
    consumers, producers : list;
    buffer : array [0..B-1] of integer;
    head, tail, cnt : integer;
    notifyConsumer, notifyProducer : boolean;

procedure getMsg(var m : msg);
begin
    if((size(consumers) > 0) and (notifyConsumer = true)) then
        remove(consumers, m)
    else if((size(producers) > 0) and (notifyProducer = true)) then
        remove(producers, m)
    else
        mbx_get(m, serverBox, INF, status);
    notifyConsumer := false;
    notifyProducer := false;
end;

begin
    head := 0; tail := 0; cnt := 0;
    notifyConsumer := false; notifyProducer := false;
    while (true) do
    begin
        getMsg(m);
        case m.opr of
            get : begin
                if(cnt = 0) then put(consumers, m)
                else
                    begin
                        m.data := buffer[tail];
                        tail := (tail + 1) mod B;
                        cnt := cnt - 1;
                    end;
                m.opr := ack;
            end;
    end;
end;
```

```
        mbx_put(m, cBox[m.id]);
        notifyProducer := true;
    end
end;
put : begin
    if(cnt = B) then put(producers, m)
    else
begin
    buffer[tail] := m.data;
    head := (head + 1) mod B;
    cnt := cnt + 1;

    m.opr := ack;
    mbx_put(m, pBox[m.id]);
    notifyConsumer := true;
end
end;
end;
end;
```

### Друго решење

Проблем код претходног решења лежи у чињеници да иако се ради о проблему кога се разматра бафер коначног капацитета ипак се користи бафер, листа, неограниченог капацитета у коју се смештају неопслужени захтеви, као и само сандуче које се понаша као бафер неограниченог капацитета. Ово може да представља проблем у случајевима да су поруке које је потребно у њима чувати веома велике. Како би се избегао овај проблем могуће је процедуре за пријем објекта у бафер поделити на два дела. Први део у коме би се примио захтев за смештање у бафер и други део у коме би се примао сам објекат.

```
program ProducerConsumer;
...
procedure Server;
var m : msg;
    status : boolean;
    consumers, producers : list;
    buffer : array [0..B-1] of integer;
    head, tail, cnt, waiting : integer;
    notifyConsumer, notifyProducer : boolean;
procedure getMsg(var m : msg);
begin
    if((size(consumers) > 0) and (notifyConsumer = true)) then
        remove(consumers, m)
    else if((size(producers) > 0) and (notifyProducer = true)) then
        remove(producers, m)
    else
        mbx_get(m, serverBox, INF, status);
    notifyConsumer := false;
    notifyProducer := false;
end;

begin
    head := 0; tail := 0; cnt := 0; waiting := 0;
```

```
notifyConsumer := false; notifyProducer := false;
while (true) do
begin
    getMsg(m);
    case m.opr of
        get : begin
            if (cnt = 0) then put(consumers, m)
            else
                begin
                    m.data := buffer[tail];
                    tail := (tail + 1) mod B;
                    cnt := cnt - 1;

                    m.opr := ack;
                    mbx_put(m, cBox[m.id]);
                    notifyProducer := true;
                end
        end;
        putRequest : begin
            if ((cnt + waiting) >= B) then
                put(producers, m)
            else
                begin
                    m.opr := ack;
                    mbx_put(m, pBox[m.id]);
                    waiting := waiting + 1;
                end
        end;
        putData : begin
            buffer[tail] := m.data;
            head := (head + 1) mod B;
            waiting := waiting - 1;
            cnt := cnt + 1;
            notifyConsumer := true;
        end;
    end;
end;
end;
```

## 42 Филозофи за ручком

Користећи асинхрону комуникацију помоћу сандучића решити проблем филозофа који ручавају.

- а) Филозофи могу да шаљу поруке само у сандучиће придружене суседним виљушкама. Виљушке су процеси и могу путем сандучића да комуницирају само са филозофима, не и међусобно (*дистрибуирано решење*).
- б) Филозофи могу да шаљу поруке само централном серверу (*централизовано решење*). Користити активне мониторе.
- в) Виљушке нису процеси. Једино су филозофи који ручавају процеси. Приликом решавања задатка потребно је избеги узајамно блокирање тако што филозоф прво узима једну виљушку и уколико другу виљушку не добије у неком фиксном временском периоду враћа прву. Претпоставити да више филозофа може да приступи и чита исто сандуче (*тобални сандучићи*).

### Решење

а) Проблем филозофа који ручавају је овде представљен у својој дистрибуираној варијанти. У овој варијанти филозофи не комуницирају директно, већ комуницирају само са себи суседним виљушкама. Пошто у овом систему не постоји централно тело које води рачуна о томе колико је слободних виљушки на столу и да ли постоји опасност од појаве узајамног блокирања треба наћи начин како да се отклоне ови недостаци. Предложено решење полази од идеје да филозофи увек узимају виљушке у редоследу који не садржи кружну зависност. Један од могућих распореда узимања виљушки је узимање прво виљушке са мањим па тек онда виљушке са већим редним бројем. На овај начин једино филозоф који се налази на kraju, са највећим редним бројем, узима виљушке у обрнутом редоследу у односу на све остале филозофе. Пошто се у том случају филозофи са редним бројевима 0 и 4 боре за исту виљушку како прву не може доћи до ситуације да оба добију виљушку. На тај начин је спречено узајамно блокирање филозофа.

#### Прво решење

Разматра се решење код којег процес може имати већи број сандучића преко којих прима поруке. Дозвољено је да из неког сандучета само један процес чита поруку, а већи број може да је уписује. Обично у неком процесу постоји једно сандуче по једном типу поруке коју може примити.

Код за филозофа се састоји из почетног дела у коме се одређује редослед по коме се узимају виљушке и самог комуникационог дела. У комуникационом делу филозоф у поруку ставља своју идентификацију ( $m.id := id$ ), да би виљушка знала коме је потребно да пошаље одговор. Пошто приликом слања поруке у сандуче филозоф нема повратну информацију о томе да ли је порука прихваћена или не, мора се увести начин за повратну комуникацију. Повратна комуникација је остварена користећи сандучиће придружене филозофима. Приликом комуникације филозоф шаље захтев у сандуче *forkGetBox* које је придружено одговарајућим виљушцима. Слободна виљушка филозофу шаље одговор којим га обавештава да ју је он заузeo у текућој итерацији слањем поруке у сандуче *philosopherBox* које је придружено филозофу који ју је захтевао. Овде имамо синхрону слање поруке и синхрон пријем. Када филозоф прибави обе виљушке прелази на јело. Када је завршио са јелом, ослобађа сваку од виљушки слањем поруке у сандуче *forkPutBox* придруженом виљушкама. Треба приметити да није потребно

остварити повратну информацију од виљушке ка клијенту, тј. постоји асинхроно слање и синхрони пријем.

Код за виљушке се састоји из два дела, фазе заузимања и фазе ослобађања. У фази заузимања виљушка у свом сандечету *forkGetBox* налази поруку која садржи идентификацију филозофа који је тражи (*id := m.id*). Када прими поруку од филозофа, виљушка враћа одговор истом том филозофу (*philosopherBox[m.id]*). Након тога прелази на фазу чекања ослобађајуће поруке. Када прими ослобађајућу поруку, виљушка се враћа у фазу чекања поруке о заузимању.

```
program DiningPhilosophers;

const N = 5;
var   forkGetBox : array [0..N-1] of mbx;
      forkPutBox : array [0..N-1] of mbx;
      philosopherBox : array [0..N-1] of mbx;

procedure Philosopher(id : integer);
var   first, second : 0..N-1;
      m, t : msg;
      status : boolean;
procedure think; begin ... end;
procedure eat; begin ... end;

begin
  if (id <> (N - 1)) then begin
    first := id;
    second := (id + 1) mod N
  end
  else
  begin
    first := (id + 1) mod N;
    second := id
  end;
  while (true) do
  begin
    think;
    m.id := id;
    mbx_put(m, forkGetBox[first]);
    mbx_get(t, philosopherBox[id], INF, status);
    mbx_put(m, forkGetBox[second]);
    mbx_get(t, philosopherBox[id], INF, status);
    eat;
    mbx_put(m, forkPutBox[first]);
    mbx_put(m, forkPutBox[second])
  end
end;
procedure Fork(id : integer);

var   m : msg;
      status : boolean;
begin
  while (true) do
  begin
    mbx_get(m, forkGetBox[id], INF, status);
```

```
    mbx_put(m, philosopherBox[m.id]);
    mbx_get(m, forkPutBox[id], INF, status);
end;
end;
```

### Друго решење

Разматра се решење код којег процес може имати само једно сандуче преко кога прима поруке. Дозвољено је да из неког сандучета само један процес чита поруку, а већи број може да је уписане.

Код за филозофа је готово идентичан као у претходном решењу, са том разликом што филозоф сада виљушци шаље поруке само у сандуче *forkBox*, а не у сандучиће *forkGetBox* и *forkGetBox*.

Код за виљушке се разликује у односу на претходно решење и нешто је комплекснији. На почетку виљушка узима поруку из свог јединог сандучета *forkBox[id]* и проверава да ли се ради о фази заузимања или фази ослобађања. Уколико је идентификатор пошиљаоца различит од идентификатора процеса који тренутно користи виљушки (*m.id*  $\leftrightarrow$  *pid*), онда се ради о фази заузимања, а ако је једнак идентификатору, онда се ради о фази ослобађања.

У фази заузимања виљушка проверава да ли је слободна, односно нико тренутно не користи виљушку (*pid* = -1). Ако је виљушка слободна онда се враћа одговор истом том филозофу (*philosopherBox[m.id]*) и поставља информацију да је тај филозоф тренутно користи (*pid* := *m.id*). Ако је виљушка заузета поставља се информација да филозоф чека (*waiting* := *m.id*).

У фази ослобађања виљушка проверава да ли неко други чека да је користи (*waiting* < $\rightarrow$  -1). Ако чека онда виљушка враћа одговор филозофу који чека (*philosopherBox[waiting]*), поставља информацију да је тај филозоф тренутно користи (*pid* := *waiting*) и да нико не чека (*waiting* := -1). Ако нико не чека онда виљушка поставља информацију да је слободна (*pid* := -1).

```
program DiningPhilosophers;
const N = 5;
var forkBox : array [0..N-1] of mbx;
    philosopherBox : array [0..N-1] of mbx;

procedure Philosopher(id : integer);
var first, second: 0..N-1;
    m, t : msg;
    status : boolean;
    ...

begin
    if (id <> (N - 1)) then begin
        first := id;
        second := (id + 1) mod N
    end
    else
    begin
        first := (id + 1) mod N;
        second := id
    end;
    while (true) do
    begin
```

```
    think;
    m.id := id;
    mbx_put(m, forkBox[first]);
    mbx_get(t, philosopherBox[id], INF, status);
    mbx_put(m, forkBox[second]);
    mbx_get(t, philosopherBox[id], INF, status);
    eat;
    mbx_put(m, forkBox[first]);
    mbx_put(m, forkBox[second])
  end
end;
procedure Fork(id : integer);

var m, t : msg;
status : boolean;
pid, waiting : integer;
begin
  pid := -1;
  waiting := -1;
  while (true) do
    begin
      mbx_get(m, forkBox[id], INF, status);
      if(m.id <> pid) then
        begin {getFork}
          if(pid = -1) then
            begin
              mbx_put(t, philosopherBox[m.id]);
              pid := m.id;
            end
          else
            waiting := m.id
        end
      else
        begin {putFork}
          if(waiting <> -1) then
            begin
              mbx_put(t, philosopherBox[waiting]);
              pid := waiting;
              waiting := -1;
            end
          else
            pid := -1
        end
    end
  end;
end;
```

- б) Код централизованог решења филозофи комуницирају искључиво са сервером. Сервер води евиденцију о томе који филозоф тренутно једе и води рачуна да не дође до међусобног блокирања. Код активних монитора сервер има један ток контроле у коме обавља обраду по принципу догађајима вођеног извршавања (*Event Driven Execution*). Сервер поред једне нити има само једно сандуче кроз које прима поруке које су му упућене.

филозоф, за разлику од претходног решења, више не води рачуна о међусобном блокирању, то сада ради централни сервер. Пошто се ради о активним мониторима, потребно је да филозоф проследи серверу поруку која би одговарала позиву одговарајуће мониторске процедуре (*pickup(id)*). Код активних монитора порука треба да садржи идентификатор пошиљаоца како би активни монитор знао ко упућује поруку и ако је потребно послао одговор назад, информацију о типу захтева и евентуално аргументе које иду уз дати захтев. Ова порука коју филозоф шаље централном серверу садржи идентификатор филозофа (*m.id := id*), информацију о типу захтева (*m.opr := pickup*), али не садржи друге аргументе, јер је аргумент (*id*) одговарајуће процедуре (*pickup*) већ прослеђен. Пошто се ради о блокирајућој мониторској процедуре филозоф чека потврду од сервера. Након јела филозоф шаље серверу поруку која одговара позиву мониторске процедуре (*putdown(id)*). Порука коју филозоф шаље централном серверу садржи идентификатор филозофа (*m.id := id*), информацију о типу захтева (*m.opr := putdown*), али не садржи друге аргументе, јер је аргумент (*id*) одговарајуће процедуре (*putdown*) већ прослеђен. Пошто се ради о неблокирајућој мониторској процедуре филозоф не чека потврду од сервера.

Сервер извршава бесконачну петљу у којој обрађује приспеле догађаје, захтеве. Када прими захтев сервер на основу типа (case наредба) прелази на обраду тог типа захтева. Уколико се ради о захтеву типа *pickup* сервер бележи да је филозоф упутио захтев за јелом (*state[m.id] := hungry*). Након тога позива процедуру у којој проверава да ли сме да једе, *test(m.id)*. Уколико се у процедуре утврди да филозоф сме да једе о томе се и обавештава. Уколико се утврди да филозоф не сме да једе, што би код обичних монитора проузроковало блокирање нити, сервер се не блокира, јер има само једну нит, већ захтев остаје забележен, а сервер се враћа на пријем следећег захтева.

Уколико се ради о захтеву типа *putdown* сервер бележи да је филозоф јавио да је завршио са јелом (*state[m.id] := thinking*). Након тога позива процедуру у којој проверава да ли леви и десни суседи овог филозофа смеју да једу, *test((m.id+N-1) mod N)* и *test((m.id+1) mod N)*. Уколико се у процедуре утврди да неки блокирани филозоф сме да једе, о томе се обавештава тај блокирани филозоф. На крају се сервер враћа на пријем следећег захтева.

Процедура за проверу и обавештавање је дosta слична као она код монитора. Прво се проверава да ли филозоф сме да једе. Филозоф сме да једе ако је исказао жељу да жели да једе и ако ни леви ни десни сусед не једу. Ако је услов испуњен поставља се стаус да филозоф једе, *state[id] := eating*, и о томе обавештава филозоф. Код монитора је обавештавање било путем сигнализације на одговарајућој условној променљиви, *can\_eat[id].signal*, што је код активних монитора еквивалентно слању поруке блокираном процесу, *mbx\_put(m, philosopherBox[id])*.

```
program DiningPhilosophers;
const N = 5;
    pickup = 0; putdown = 1;

    thinking = 0; hungry = 1; eating = 2;
var  philosopherBox : array [0..N-1] of mbx;
    serverBox : mbx;

procedure Philosopher(id : integer);

var  m : msg;
    status : boolean;
...
begin
```

```
while (true) do
begin
    think;
    {pickup}
    m.id := id;
    m.opr := pickup;
    mbx_put(m, serverBox);
    mbx_get(m, philosopherBox[id], INF, status);

    eat;
    {putdown}
    m.id := id;
    m.opr := putdown;
    mbx_put(m, serverBox);

end
end;

procedure Server;
var
    m : msg;
    status : boolean;
    state : array[0..N-1] of integer;
procedure test(id : integer);
begin
    if((state[id] = hungry)
       and (state [(id+N-1) mod N] <> eating)
       and (state [(id+1) mod N] <> eating)) then
    begin
        state[id] := eating;
        mbx_put(m, philosopherBox[id]);
    end;
end;
begin
    while (true) do
begin
    mbx_get(m, serverBox, INF, status);
    case m.opr of
        pickup : begin
            state[m.id] := hungry;
            test(m.id);
        end;
        putdown : begin
            state[m.id] := thinking;
            test((m.id+N-1) mod N);
            test((m.id+1) mod N);
        end;
    end;
end;
end;
```

Све што је важило за одговарајуће решење са мониторима, стоји и овде.

в)

```
program DiningPhilosophers;
const N = 5;
    MAXPAUSE = 1000;

var   fork : array [0..n-1] of mbx;

procedure think; begin ... end;
procedure eat; begin ... end;

procedure Philosopher(id : integer);

var   m : msg;
      status : boolean;
      d : integer;
begin
    mbx_put(m, fork[id]);
    while (true) do
        begin
            think;
            mbx_get(m, fork[id], INF, status);
            mbx_get(m, fork[(id + 1) mod N], d, status);
            if(status) then
                begin
                    eat;
                    mbx_put(m, fork[(id + 1) mod N]);
                    mbx_put(m, fork[id]);
                end
            else
                begin
                    mbx_put(m, fork[id]);
                    pause(random(maxpause));
                end;
        end;
    end;
end;
```

## 43

### Вожња тобоганом

Претпоставити да постоји  $N$  путника и једно возило на тобогану (*The Roller Coaster Problem*). Путници се наизменично шетају по луна парку и везе на тобогану. Тобоган може да прими највише  $K$  путника при чему је  $K < N$ . Вожња тобоганом може да почне само уколико се сакупило тачно  $K$  путника. Написати програм користећи асинхрону комуникацију користећи сандучиће који симулира описани систем.

## Решење

### Прво решење

Решење код којег процес може имати већи број сандучића за пријем порука. Дозвољено је да из неког сандучета само један процес чита поруку, а већи број може да је уписује.

```
program RollerCoaster;
const K = ...;
      N = ...;
type msg = record
           id : integer;
         end;

var coasterIN, coasterOUT : mbx;
    passengerBox : array [1..N] of mbx;

procedure Passenger(id : integer);
procedure riding (id : integer); begin ... end;
procedure walking (id : integer); begin ... end;
procedure boardCar (id : integer);
var m : msg;
    status : boolean;
begin
  m.id := id;
  mbx_put(m, coasterIN);
  mbx_get(m, passengerBox[id], INF, status);
end;
procedure leaveCar (id : integer);
var m : msg;
    status : boolean;
begin
  mbx_get(m, passengerBox[id], INF, status);
  m.id := id;
  mbx_put(m, coasterOUT);
end;

begin
  while true do
  begin
    walking(id);
    boardCar (id);
    riding(id);
  end;
end;
```

```
    leaveCar (id);
end;
end;

procedure Coaster;
var i : integer;
    boarded : array[1..K] of msg;

procedure riding; begin ... end;
procedure boardingCar;
var i : integer;
    m : msg;
    status : boolean;
begin
    for i := 1 to K do
begin
    mbx_get(m, coasterIN, INF, status);
    boarded[i] := m;
end;
    for i := 1 to K do
begin
    mbx_put(m, passengerBox[boarded[i].id]);
end;
end;
procedure leavingCar;
var i : integer;
    m : msg;
    status : boolean;
begin
    for i := 1 to K do
begin
    mbx_put(m, passengerBox[boarded[i].id]);
end;
    for i := 1 to K do
begin
    mbx_get(m, coasterOUT, INF, status);
end;
end;
end;

begin
    while true do
begin
    boardingCar;
    riding;
    leavingCar;
end;
end;
```

#### Друго решење

Разматра се решење код којег процес може имати само једно сандуче преко кога прима поруке. Дозвољено је да из неког сандучета само један процес чита поруку, а већи број може да је уписује.

```
program RollerCoaster;
const K = 3;
      N = 10;
      startRide = 0; endRide = 1;
var coasterBox : mbx;
    passengerBox : array [1..N] of mbx;

procedure Passenger(id : integer);
...
procedure boardCar (id : integer);
var m : msg;
    status : boolean;
begin
    m.id := id;
    m.opr := startRide;
    mbx_put(m, coasterBox);
    mbx_get(m, passengerBox[id], INF, status);
end;
procedure leaveCar (id : integer);
var m : msg;
    status : boolean;
begin
    mbx_get(m, passengerBox[id], INF, status);
    m.id := id;
    m.opr := endRide;
    mbx_put(m, coasterBox);
end;

begin
    while (true) do
begin
    walking(id);
    boardCar (id);
    riding(id);
    leaveCar (id);
end;
end;

procedure Coaster;
var i : integer;
    boarded : array[1..K] of msg;
    boardedSize : integer;
    waiting : array[1..N] of msg;
    waitingSize, waitingW, waitingR : integer;

procedure riding; begin ... end;
procedure boardingCar;
var i : integer;
    m : msg;
    status : boolean;
begin
    boardedSize := 0;
    while(boardedSize <> K) do
begin
    if(waitingSize > 0) then
```

```
begin
    boarded[boardedSize] := waiting[waitingR];
    boardedSize := boardedSize + 1;
    waitingSize := waitingSize - 1;
    waitingR := (waitingR mod N) + 1;
end
else
begin
    mbx_get(m, coasterBox, INF, status);
    boarded[boardedSize] := m;
    boardedSize := boardedSize + 1;
end;
end;

m.opr := ack;
for i := 1 to K do
begin
    mbx_put(m, passengerBox[boarded[i].id]);
end;

end;
procedure leavingCar;
var i : integer;
    m : msg;
    status : boolean;
begin
    m.opr := ack;
    for i := 1 to K do
    begin
        mbx_put(m, passengerBox[boarded[i].id]);
    end;

    boardedSize := 0;
    while(boardedSize <> K) do
    begin
        mbx_get(m, coasterBox, INF, status);
        if(m.opr = startRide) then
        begin
            waiting[waitingW] := m;
            waitingSize := waitingSize + 1;
            waitingW := (waitingW mod N) + 1;
        end
        else if(m.opr = endRide) then
        begin
            boardedSize := boardedSize + 1;
        end;
    end;
end;
end;

begin
    waitingSize := 0;
    waitingW := 0;
    waitingR := 0;
    while (true) do
    begin
```

```
    boardingCar;
    riding;
    leavingCar;
  end;
end;
```

### Треће решење

Разматра се решење код кога се користе активни монитори.

```
program RollerCoaster;
const
  K = 3;
  N = 10;
  startRide = 0; endRide = 1;

var coasterBox : mbx;
  passengerBox : array [1..N] of mbx;

procedure Passenger(id : integer);
procedure riding (id : integer); begin ... end;
procedure walking (id : integer); begin ... end;
procedure boardCar (id : integer);
var m : msg;
  status : boolean;
begin
  m.id := id;
  m.opr := startRide;
  mbx_put(m, coasterBox);
  mbx_get(m, passengerBox[id], INF, status);
end;

procedure leaveCar (id : integer);
var m : msg;
  status : boolean;
begin
  m.id := id;
  m.opr := endRide;
  mbx_put(m, coasterBox);
  mbx_get(m, passengerBox[id], INF, status);
end;

begin
  while (true) do
  begin
    walking(id);
    boardCar (id);
    riding(id);
    leaveCar (id);
  end;
end;

procedure Coaster;
var boarded : array[1..K] of msg;
  numEnter, numExit : integer;
  msgs : list;
```

```
m : msg;

procedure riding; begin ... end;

procedure getMsg(var m : msg);
var status : boolean;
begin
    if((size(msgs) > 0) and (numEnter <> K)) then remove(msgs, m)
    else
        begin
            mbx_get(m, coasterBox, INF, status);
        end
end;
procedure notifyAll;
var m : msg;
    i : integer;
begin
    m.opr := ack;
    for i := 0 to K do
        begin
            mbx_put(m, passengerBox[boarded[i].id]);
        end;
end;
begin
    numEnter := 0;
    numExit := 0;
    while (true) do
        begin
            getMsg(m);
            case m.opr of
                startRide : begin
                    if(numEnter = K) then put(msgs, m)
                    else
                        begin
                            boarded[numEnter] := m;
                            numEnter := numEnter + 1;
                            if(numEnter = K) then
                                begin
                                    notifyAll;
                                    numExit := 0
                                end;
                            end;
                        end;
                endRide : begin
                    boarded[numExit] := m;
                    numExit := numExit + 1;
                    if(numExit = K) then
                        begin
                            notifyAll;
                            numEnter := 0
                        end;
                end;
            end;
        end;
    end;
end;
```

## 44

## Игра живота

Постоји матрица димензија  $N \times N$  таква да свака њена ћелија представља један организам који може да буде жив или мртав (*The Game of Life*). Организми могу да комуницирају само са својим суседима (горе, доле, лево, десно и укосо). Организми у средини ће имати 8 суседа, док ће они у угловима имати само 3 суседа. Правила која важе за сваки организам су следећа:

Жив организам који има мање од два живих суседа умире од усамљености.

Жив организам који има више од три живих суседа умире од пренатрпаности.

Жив организам са два или три живих суседа преживљава и формира следећу генерацију.

Мртв организам са три живих суседа оживљава.

Користећи сандучиће написати програм који симулира организам.

## Решење

Проблем који се овде разматра је један од кључних проблема у итеративној паралелној обради. Овим проблемом се описује група проблема итеративне обраде коју обавља више процеса, где сваки процес одговоран за подскуп података, и код које сваки процес садржи све потребне податке за даље рачунање или је потребно комуницирати само са суседним процесима да би се добили сви подаци и израчунало стање у следећој итерацији. Овакви проблеми су веома погодни за дистрибуирану обраду код које је могуће постићи велики степен паралелизације. Проблеми укључују обраду сплике, решавање парцијалних диференцијалних једначина, програмирање у гриду (*grid computing*), симулације ћелијских система, и слично.

Принцип „торбе послова“ се може применити на проблеме где или већ постоји подела на већи број независних послова, или где се посао може делити на већи број мањих послова који се могу обрађивати независно. Како то овде није случај примењује се другачија парадигма која је заснована на „*паратими уулсирања*“ (*Heartbeat paradigm*). Код ове парадигме обрада се обавља у петљи у којој процес, чвор, који ради обраду пролази кроз три фазе. Процес прво свим својим суседима прослеђује своје тренутно стање (фаза експанзије), након тога прима информације о стању од својих суседа (фаза контракције), да би на основу свог стања и стања примљеног од суседа израчунao своје ново стање (фаза обраде).

```
procedure Node(id : integer);
begin
    иницијализовати променљиве;
    while (not done) do begin
        послати вредности суседима;
        примити вредности од суседа;
        израчунати наредну вредност;
    end;
end;
```

Комуникација између суседа се обавља у фази слања и фази пријема, прво се шаљу подаци, па се чека да се претходна итерација на суседима заврши како би они послали своје податке на основу којих је могуће прећи у наредну итерацију. Овакав концепт комуникације твори ситуацију сличну оној која постоји код синхронизације на баријери.

Баријера омогућава да се суседни процеси синхронизују пре него што пређу на следећу итерацију израчунавања. Овим се постиже да сваки процес добије податке од својих суседа из одговарајуће итерације. Треба приметити да се оваквом баријером може постићи да се суседи разликују за максимално једну итерацију. Може се десити да је у итерацији  $i$  процес послao своје стање, од свих својих суседа примио стање за итерацију  $i$ , процес је завршио своје израчунавање и припремио стање за итерацију  $i+1$ , послao податке за итерацију  $i+1$ , и чека да добије податке од суседа који можда и даље чека неког свог другог суседа како би добио податке за итерацију  $i$  и завршио рачунање. Такође се може десити да се несуседни процеси у неким случајевима можда могу разликовати и за више итерација у зависности од потребе за подацима. Основни проблем о коме треба водити рачуна код ове парадигме је да се приликом пријема података строго разграничи који се подаци односе на текућу итерацију, а који на следећу, јер је можда неки сусед био бржи.

Обрада која се овде примењује је релативно једноставна, израчунава се следећи статус ћелије у матрици на основу датих правила. Сваки живи организам је представљен једним процесом који се извршава и који комуницира са својим суседима. Сваком организму је придружено сандуче (*box* : *array* [1..*N*, 1..*N*] of *mbx*). Овде је узето да се ради о квадратној матрици димензија  $N \times N$ . У систему постоји већи број услужних функција које обрађују који су све организми суседи посматраног, односно које су њихове координате у матрици. Функције *xStart(i)* и *yStart(i)* враћају *x* односно *y* горњег левог суседа док *xEnd(i)* и *yEnd(i)* враћају координате за суседа који се налази доле лево у односу на посматраног. Функција *numOfNeighbours* враћа број суседа датог организма.

Итеративни поступак обраде се понавља све док се не одради довољан број итерација (*numGenerations*). У свакој итерацији се свим суседима шаље комплетна информација о организму. Ове информације садржи статус организма (жив или не), координате организма, али и итерацију за коју се дати статус односи. Информација о итерацији је битна, јер неки организми могу да предњаче у односу на друге, то јест да пређу на следећу итерацију пре него што су сви остали завршили текућу. Овде је потребно нагласити да за прелазак на следећу итерацију није неопходно да сви заврше текућу итерацију него само сви суседи. Такође треба напоменути да број итерација за које неки процеси предњаче сигурно није већи од једне итерације. Када се заврши фаза сланања информације о текућем стању прелази се на фазу прикупљања информација о стању свих суседа. Пошто добијене информације припадају текућој или следећој итерацији морају се правилно интерпретирати. Информације о парним итерацији са парним редним бројем се налазе у врсти 0 матрице *neighbours*, а о непарним у врсти 1. Пошто је највећи број суседа које један организам може да има 8, димензионисање је урађено према том најгорем случају. Да би се пратило колико има пристиглих порука из које итерације користе се низ капацитета 2 за њихово чување (*num*). На крају итерације се на основу добијених информација о текућем стању суседа, информације о текућем стању и броју суседа датог организма израчунава следеће стање организма, то јест стање за следећу итерацију (*calculateState*).

```
program GameOfLife;
const numGenerations = ...;
    N = ...;
type msg = record
    status : boolean;
    i, j, index : integer;
end;
var box : array [0..N-1, 0..N-1] of mbx;

procedure Node(i : integer; j : integer);
```

```
var p, q, k : integer;
status, st : boolean;
neighbours : array[0..1, 0..7] of msg;
num : array[0..1] of integer;
m : msg;
begin
  num [0] := 0;    num [1] := 0;
  for k := 1 to numGenerations do
begin
  m.status := status;
  m.i := i;
  m.j := j;
  m.index := k;

  for p := xStart(i) to xEnd(i) do
    for q := yStart(j) to yEnd(j) do
      begin
        if((p <> i) or (q <> j)) then
          mbx_put(m, box[p, q]);
      end;

  while (num [k mod 2] < numOfNeighbours(i, j)) do
begin
  mbx_get(m, box[i, j], INF, st);
  neighbours[m.index mod 2, num[m.index mod 2]] := m;
  num[m.index mod 2] := num[m.index mod 2] + 1;
end;
  num[k mod 2] := 0;
  calculateState(i, j, k, neighbours, status);
end;
end;

function xStart(i : integer) : integer;
begin
  i := i - 1;
  if(i < 0) then xStart := 0
  else xStart := i;
end;
function yStart(i : integer) : integer;
begin
  i := i - 1;
  if(i < 0) then yStart := 0
  else yStart := i;
end;
function xEnd(i : integer) : integer;
begin
  i := i + 1;
  if(i >= N) then xEnd := N-1
  else xEnd := i;
end;
function yEnd(i : integer) : integer;
begin
  i := i + 1;
  if (i >= N) then yEnd := N-1
  else yEnd := i;
```

```
end;
function numOfNeighbours(i, j : integer) : integer;
begin
    numOfNeighbours := (xEnd(i) - xStart(i) + 1)
        *(yEnd(i) - yStart(i) + 1) - 1;
end;
procedure calculateState(i, j, k : integer;
    neighbours : array[0..1, 0..7] of msg; var status : boolean);
var index, num : integer;
begin
    num := 0;
    for index := 0 to numOfNeighbours(i, j) - 1 do
    begin
        if neighbours[k mod 2, index].status then num := num + 1;
    end;
    if (status and ((num < 2) or (num > 3))) then status := false
    else if ((not status) and (num = 3)) then status := true;
end;
```

**45**

## Прстен

Реализовати прстен од  $N$  процеса, који комуницирају асинхроним прослеђивањем порука коришћењем сандучића, а код кога процес 0 генерише случајан низ од  $M$  природних бројева у интервалу од 1 до 20.000 завршен са *sentinel* вредношћу  $EONAT$  (нпр 20.001). Преосталих  $N-1$  процеса одстрањују из низа све појаве првих  $N-1$  простих бројева. Сваки процес има по један различит прост број за који је задужен. Након одстрањивања простих бројева, модификовани низ се дистрибуира свим процесима. Обезбедити максималну конкурентност.

## Решење

Прстен у коме се обавља описана обрада се састоји од две групе процеса. Прву групу чини почетни процес (*Worker(0)*) који генерише низ бројева које је потребно обрадити, док другу чине преосталих  $N-1$  процеса (*Worker(1..N-1)*) који над примљеним подацима обављају обраду. Извршавање првог процеса се састоји из две фазе. У првој фази се генерише  $M$  података који се прослеђују првом наредном процесу у прстену (*Worker(0)*), а у другој фази се низ података примљен од претходног процеса у прстену (*Worker(N-1)*) прослеђује наредном процесу у прстену (*Worker(1)*). Извршавање се на преосталим процесима такође састоји из две фазе. У првој фази се прима одређен број података од претходника у прстену, над подацима се обавља обрада и обрађени подаци се прослеђују првом наредном процесу у прстену. Друга фаза је идентична као код почетног процеса, примају се подаци од претходника и онда се даље прослеђују следбенику уколико следбеник постоји.

Описани поступак даје генералне назнаке о решавању проблема користећи прстен процеса, али не и броју односно количини података које је потребно разменити између процеса како би се обезбедила максимална конкурентност, односно максимални број истовремено активних обрада. Обрада која је описана у овом задатку се састоји из одстрањивања задатог простог броја из низа примљених бројева. Уколико би се проблем решио тако што би сваки чвор у прстену чекао да прими комплетан низ, онда из тог низа одстрањивао задати прост број и на крају тако формиран нови низ прослеђивао наредном процесу у прстену то не би довело до максималне конкурентности. Такво решење би сигурно било спорије него да цео посао обавља само један процес. Код оваквог приступа у једном тренутку би само један процес обављао обраду док би сви преостали процеси морали да чекају. До максималне конкурентности би се дошло уколико би истовремено сви процеси у прстену могли да обављају обраду. Ово значи да би уместо слања комплетног низа између процеса требало слати мање делове како би процеси могли да обављају обраду над мањим деловима и да шаљу наредном процесу податке на обраду чим буду спремни. Минимално би се могло слати један по један елемент низа. Оптимална количина података која се размењује зависи од времена обраде и времена потребног за транспорт података. Пошто се у овом примеру разматра асинхрона комуникација то значи да када један процес заврши са тренутном обрадом, тај процес те обрађене податке може послати наредном процесу без обзира да ли је тај наредни процес завршио са својом дотадашњом обрадом. Ово је постигнуто коришћењем сандучића за које се сматра да имају неограничен капацитет. Уколико би требало водити рачуна о капацитetu сандучића, било би потребно увести додатну синхронизацију између суседних процеса.

Описани поступак је пример паралелизације рада применом тзв. *бројочне обраде* (*pipelining*) на нивоу процеса. Сваки процес представља један степен проточне обраде, који асинхроно прима извесну количину података од претходног степена, обрађују их и

тако обрађене податке прослеђују наредном степену. На овај начин сви степени могу да раде упоредо, након што приме прве податке. Да би се постигла максимална конкурентност, чим почетни процес генерише појединачни податак, он се одмах шаље следећем процесу у прстену. На тај начин се већ након генерисања N података из почетног низа од стране почетног процеса, може очекивати да буду упослени скоро сви процеси. Иста логика важи и за дистрибуцију финалног низа.

Проточно решење са *sentinel* вредношћу на крају низа и прстеном процеса има још једну значајну предност. Када би се применило решење у коме се пребацију комплетни низови, морали би се негде скраћивати величине низова и препакивати остали елементи низа. То је веома споро, чак и ако се ради само на једном месту (нпр. у процесу *Worker(N-1)*, а остали процеси само означавају који елемент низа треба избацити, јер је ту био прост број). У проточној обради, изостављени елементи се једноставно не би слали од стране сваког процеса који их изоставља, па би процес (*Worker(N-1)*) добио финални препакован низ без икакве додатне обраде, а чак би се смањивао посао процесима, јер би низови одмах били краћи!

```
program Ring;

const EONAT = 20001;
      M = ...;
      N = ...;

var   box : array [0..N] of mbx;

procedure Worker(id : 0);
var   m : msg;
      status, go : boolean;
      index : integer;
      data : array [1..M] of integer;
begin
  for index := 1 to M-1 do
    begin
      m := produce();
      mbx_put(m, box[1]);
    end;
  m.val := EONAT;
  mbx_put(m, box[1]);

  go := true;
  index := 1;
  while (go) do
    begin
      mbx_get(m, box[id], INF, status);
      data[index] := m.val;
      index := index + 1;
      if(id <> N - 1) then mbx_put(m, box[id + 1]);
      if(m.val = EONAT) then go := false;
    end
  end;

procedure Worker(id : 1..N-1; prime : integer);
var   m : msg;
      status, go : boolean;
```

```

index : integer;
data : array [1..M] of integer;
begin
go := true;
while (go) do
begin
    mbx_get(m, box[id], INF, status);
    if(m.val <> prime) then begin
        mbx_put(m, box[(id + 1) MOD N]);
        if(m.val = EONAT) then go := false;
    end;
end;

go := true;
index := 1;
while (go) do
begin
    mbx_get(m, box[id], INF, status);
    data[index] := m.val;
    index := index + 1;
    if(id <> N - 1) then mbx_put(m, box[id + 1]);
    if(m.val = EONAT) then go := false;
end;
end;

```

Описано решење се може додатно убрзати тако што би процеси имали различиту другу фазу при којој шаљу податке. Уместо да почетни процес (*Worker(0)*) почне са формирањем низа то може учинити последњи процес (*Worker(N-1)*). Последњи процес у прстену (*Worker(N-1)*), који први формира финални низ, чим почиње да формира сопствени финални низ, при убацувању сваког елемента у низ одмах шаље појединачни податак следећем процесу у прстену и сваки следећи процес чини исто. Тако се значајно убрзава формирање финалних низова, а ефективна дужина проточног низа процеса је скоро два прстена процеса ( $2^*(N-1)$ ) порука, односно ( $2^*N-1$ ) процеса у проточном низу. Нема потребе да претпоследњи процес у низу (*Worker(N-2)*) резултате шаље последњем процесу јер их је он већ формирао.

```

program Ring;

const EONAT = 20001;
      M = ...;
      N = ...;

var box : array [0..N] of mbx;

procedure Worker(id : 0);
var m : msg;
    status, go : boolean;
    index : integer;
    data : array [1..M] of integer;
begin
for index := 1 to M-1 do
begin
    m := produce();
    mbx_put(m, box[1]);

```

```
end;
m.val := EONAT;
mbx_put(m, box[1]);

go := true;
index := 1;
while (go) do
begin
    mbx_get(m, box[id], INF, status);
    data[index] := m.val;
    index := index + 1;
    mbx_put(m, box[id + 1]);
    if(m.val = EONAT) then go := false;
end
end;

procedure Worker(id : 1..N-1; prime : integer);
var
    m : msg;
    status, go : boolean;
    index : integer;
    data : array [1..M] of integer;
begin
    go := true;
    index := 1;
    while (go) do
begin
    mbx_get(m, box[id], INF, status);
    if(m.val <> prime) then begin
        mbx_put(m, box[(id + 1) MOD N]);
        if(m.val = EONAT) then go := false;
        if(id = N - 1) then
        begin
            data[index] := m.val;
            index := index + 1;
        end;
    end;
end;
    end;

    go := (id <> N - 1);
    while (go) do
begin
    mbx_get(m, box[id], INF, status);
    data[index] := m.val;
    index := index + 1;
    if(id <> N - 2) then mbx_put(m, box[id + 1]);
    if(m.val = EONAT) then go := false;
end;
end;
end;
```

46

## Јавно емитовање

Постоји повезан граф који се састоји од  $N$  чворова. Чворови могу да комуницирају само са суседним чворовима. Користећи сандучиће написати програм који поруку коју шаље један чвр прослеђује свим осталим чворовима у графу.

Реализовати овај проблем уколико:

- а) сваки чвр има информације само о својим суседима.
- б) почетни чвр има информације о комплетној топологији графа.

## Решење

а)

### Прво решење

Пошто се овде ради о повезаном графу који се састоји од  $N$  чворова потребно је обезбедити начин на који ти чворови могу да комуницирају. Комуникација се обавља користећи сандучиће придржане сваком чврту, *probe* : array [1.. $N$ ] of mbx. На почетку сваки чвр у графу очекује поруку коју треба да прими и након тога да је проследи својим суседима. Да би се водило рачуна о суседности сваки чвр у графу поседује листу свих својих суседа, односно низ капацитета  $N$  који показује који су чворови у графу суседи датог чврта, *links* : array [1.. $N$ ] of boolean. Када чвр прими поруку коју треба да проследи суседима пролази кроз листу суседности и свим суседима шаље ту исту поруку. Када је свим суседима послао поруку прелази на део чекања да се истоветна порука врати од свих суседа, осим од оног од кога је сам примио поруку. Сваки чвр је послао *num* порука, а очекује *num-1* порука, где *num* представља број суседа у графу. Уколико се ради о потпуно повезаном графу број разменјених порука је  $N(N-1)$ , што у неким случајевима може да изазове загушење у комуникацији. Да би се број порука у систему смањио потребне су ближе информације о топологији целог графа, у том случају би сваки чвр поседовао информације о читавом графу, а не само информације о суседима.

Да би систем могао да почне са функционисањем потребно је да постоји процес који би иницирао покретање целог поступка слања порука. То је процес *Initiator* који започиње слање порука почетном чврту  $S$ . Да се не би појавила заостала порука у бафери на чврту  $S$  потребно је реализовати да променљива *num* чврта  $S$  буде за један већа од стварног броја суседа.

```
program Broadcast;
const N = ...;
var probe : array [1.. N] of mbx;

procedure Node(p : 1..N);
var links : array [1..N] of boolean;
    m : msg;
    num : integer;
    q : integer;
    st : boolean;
begin
    init(links, num);
    mbx_get(m, probe[p], INF, st) ;
    {send m to all neighbours}
```

```
for q := 1 to N do
    if(links[q]) then mbx_put(m, probe[q]);
{receive num-1 redundant copies of m}
for q := 1 to num-1 do
    mbx_get(m, probe[p], INF, st) ;
end;

procedure Initiator; {executed on source node S}
var   m : msg;
      S : integer;
begin
    mbx_put(m, probe[S]);
end;
```

### Друго решење

Да се не би појавила заостала порука у баферу на чврлу који је иницирао почетак, S, овде је реализовано да он има комплетно другачије тело.

```
procedure Initiator(S : integer); { Initiator + Node}
var   m : msg;
begin
    init(links, num);
    {send m to all neighbours}
    for q := 1 to N do
        if(links[q]) then mbx_put(m, probe[q]);
    {receive num redundant copies of m}
    for q := 1 to num do
        mbx_get(m, probe[S], INF, st) ;
end;
```

### Треће решење

Да се не би појавила заостала порука у баферу на чврлу који је иницирао почетак, S, овде је реализовано да сваки чвр провери да ли је он почетни, и уколико јесте да увећа број суседа за један.

```
procedure Node(p : 1..N);

...
if p = S then num := num + 1;
for q := 1 to num-1 do
    mbx_get(m, probe[p], INF, st) ;
end;
```

- б) Овде ради о повезаном графу који се састоји од  $N$  чвркова код кога је потребно обезбедити начин на који ти чврлови могу да комуницирају. Комуникација се обавља користећи сандучиће придржане сваком чврлу,  $probe : array [1..N] of mbx$ . На почетку сваки чвр у графу очекује поруку коју треба да прими, а након тога да је проследи свим својим суседима који је до тог тренутка нису примили. Постоје у овом случају уводи претпоставка да је позната комплетна топологија графа на основу које се може одредити и пут поруке кроз граф тако да број порука које сваки чвр прими буде минималан. Како се овде ради о повезаном графу минималан број порука које сваки чвр треба да прими износи 1 тако да укупан број порука које се размене износи  $N$ . Један од начина да би се постигло спање минималног броја порука је формирање обухватног стабла кретања порука кроз граф са кореном у почетном чврлу S. Да би се

добили корено стабло може се користити *Spanning tree* алгоритам за уклањање петљи у графу. Податак о кретању поруке односно о стаблу кретања мора да буде уграђено у саму поруку, то јест мора се транспортовати између чворова. Добијање ове топологије је представљено помоћу  $t := m.spanningTree$ . Када се добије стабло кретања потребно је послати поруку само чворовима који излазе из датог чвora. Овим се број порука које треба разменити кроз мрежу знатно смањује,  $N$  порука се размени што представља значајно смањење броја порука у односу на претходно решење, али је овом приликом сама порука постала значајно сложенија.

```

program BroadcastTree;
const N = ...;
type graph = array [1..N, 1..N] of boolean;
type msg = record
    data : message;
    spanningTree : graph;
end;
var probe : array [1..N] of mbx;

procedure Node(p : 1..N);
var t : graph;
    m : msg;
    q : integer;
    st : boolean;
begin
    init(p);
    mbx_get(m, probe[p], INF, st) ;
    t := m.spanningTree;
    {send m to all children}
    for q := 1 to N do
        if(t[p, q]) then mbx_put(m, probe[q]);
            {q is a child of p in t}
end;

procedure Initiator;
var m : msg;
    S : integer;
    topology : graph; {network topology}
    t : graph; {spanning tree of topology}
begin
    initTopology(topology, t);
    m.spanningTree := t;
    mbx_put(m, probe[S]);
end;

```

47

## Откривање топологије

Постоји повезан граф који се састоји од  $N$  чворова (*Network Topology Detection*). Чворови могу да комуницирају само са суседним чворовима. Користећи сандучиће написати програм који открива топологију графа. На почетку сваки чвор има информације само о својим суседима.

### Решење

Поступак откривања топологије је доста сличан поступку јавног емитовања. Састоји се из размене два типа порука између суседних чворова. Један тип је захтев за откривање топологије (*PROBE*), а други тип је одговор по претходно упућеном захтеву (*ECHO*). На почетку чвор прими захтев за откривање топологије. Након тога тај чвор свим својим суседима, осим оном чврору од кога је примио захтев, прослеђује захтев за откривање топологије. Када је упутио захтеве прелази на пријем порука од својих суседа. Потребно је да од свих својих суседа, осим од оног од кога је добио прву поруку прими одговор по упућеном захтеву. Када добије одговор проширује знање о везама својих суседа. Овде треба приметити да поред одговора чвор може да добије и нови захтев за откривање топологије од неког суседа који није директна посредница претходно упућеног захтева већ захтева који је упутио суседов сусед. Одговор на такве захтеве не садржи никакве податке о графу јер ће они на другом месту бити срачунати. Када прикупи резултате од суседа израчунату топологију прослеђује чврору који је иницирао операцију.

```

program ProbeEcho;
const N = ...;
S = ...;
type graph = array [1..N, 1..N] of boolean;
type kind = (PROBE, ECHO);
type msg = record
    tp : kind;
    senderID : integer;
    topology : graph;
end;
var probe_echo : array [1..N] of mbx;
finalecho : mbx;
procedure Node(p : 1..N);
var links : array [1..N] of boolean;
newTopology, localTopology : graph;
first, sender : integer;
k : kind;
need_echo : integer;
m : msg;
q, i, j : integer;
st : boolean;
begin
links := ...
need_echo := ... {number of neighbours -1}
for i := 1 to N do localTopology[p][i] := links[i];
mbx_get(m, probe_echo[p], INF, st) ;
k := m.tp;
first := m.senderID;

```

```
newTopology := m.topology;
for q := 1 to N do
    if (links[q] and (q <> first)) then
begin
    m.tp := PROBE;
    m.senderID := p;
    m.topology := nil;
    mbx_put(m, probe_echo[q]);
end;
while (need_echo > 0) do
begin
    mbx_get(m, probe_echo[p], INF, st) ;
    k := m.tp;
    sender := m.senderID;
    newTopology := m.topology;
    if (k = PROBE) then
begin
    m.tp := ECHO;
    m.senderID := p;
    m.topology := nil;
    mbx_put(m, probe_echo[sender]);
end
else begin
    for i := 1 to N do
        for j := 1 to N do
            localTopology[i][j] :=
                localTopology[i][j] or newTopology[i][j];
            need_echo := need_echo - 1;
end;
m.topology := localTopology;
if p = S then mbx_put(m, finalecho)
else begin
    m.tp := ECHO;
    m.senderID := p;
    m.topology := localTopology;
    mbx_put(m, probe_echo[first]);
end;
end;
procedure Initiator;
var topology : graph;
    m : msg;
    st : boolean;
begin
    m.tp := PROBE;
    m.senderID := S;
    m.topology := nil;
    mbx_put(m, probe_echo[S]);
    mbx_get(m, finalecho, INF, st);
    topology := m.topology;
end;
begin
end.
```

## 48 Двоелементни бафер

Процес  $P$  репетитивно шаље целобројне податке процесу  $Q$  кроз двоелементни бафер  $B$ . Бафер бесконачно понавља следећи двофазни циклус: у првој фази прима улазни податак у променљиву  $x$ , а упоредо са тим шаље на излаз вредност променљиве  $y$ ; у другој фази прима податак у  $y$ , а шаље вредност  $x$ . Да би се ушло у двофазни циклус, неопходно је иницијално напунити податак у променљиву  $y$ .

a) Реализовати бафер  $B$  као активан процес користећи *Pascal* који је проширен следећим конструктима:

- упоредном командом облика

```
cobegin P1::C1 || P2::C2 || ... || Pn::Cn coend
```

извршава се  $n$  упоредних процеса, при чему процес са именом  $P_i$  извршава команду  $C_i$ . Упоредна команда је завршена када се сви процеси  $P_i$  заврше.

- примитивама за синхроно комуникарање облика

<code>&lt;proces&gt;?&lt;promenljiva&gt;</code>	(пријем)
<code>&lt;proces&gt;!&lt;izraz&gt;</code>	(слanje)

при чему се слање и пријем порука врше уз директно именовање процеса, а комуникација је синхрона.

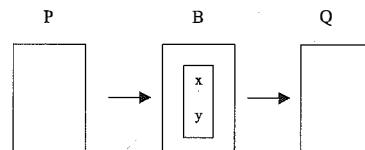
b) Реализовати бафер  $B$  као активан процес који је способан да прими целобројан податак са улазног порта *in* и да га пошаље (на захтев примљен преко улазног порта *req*) кроз излазни порт *out* користећи *Pascal* проширен следећим конструктима:

- упоредном командом облика

```
cobegin P1::C1 || P2::C2 || ... || Pn::Cn coend
```

извршава се  $n$  упоредних процеса, при чему процес са именом  $P_i$  извршава команду  $C_i$ . Упоредна команда је завршена када се сви процеси  $P_i$  заврше.

- типовима података *entryport(T)* и *exitport(T)* који омогућавају декларисање локалних улазних и излазних портова за податке типа  $T$ .
- комуникационим примитивама *receive* и *send* са индиректним именовањем чију синтаксу и семантику је, као део задатка, потребно прецизније дефинисати тако да буду подесне за решавање описаног проблема.



Слика 9. Илустрација комуникације процеса  $P$  и  $Q$  преко бафера  $B$

## Решење

```
a) procedure B;
var x,y : integer;
begin
    P?y;
    while (true) begin
        cobegin
            XIN::P?x || YOUT::Q!y
        coend;
        cobegin
            YIN::P?y || XOUT::Q!x
        coend
    end
```

```

coend
end
end;

```

б) Индиректно именовање је оно код кога се не наводи експлицитно име процеса коме се порука шаље или од кога се порука прима, већ се наводи неко логичко име које се на неки начин и ван овог програма повезује са стварним именом процеса на који се односи. Та локална имена се обично називају *боршовима*.

Дефинишимо најпре комуникационе примитиве:

**receive** *msg* **from** *entryport* је примитива за синхрони (блокирајући) пријем поруке *msg* са улазног порта *entryport*. Користи се синхрони пријем, јер би у случају асинхроног пријема у петљи морало да се ради запослено чекање.

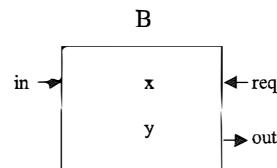
**send** *msg* **to** *exitport* је примитива за асинхроно слање поруке *msg* на излазни порт *exitport*.

Коришћењем новоуведених примитива, процес који представља бафер *B* може се написати на следећи начин:

```

procedure B;
var x, y : integer;
    r : boolean;
    in : entryport(integer);
    req : entryport(boolean);
    out : exitport(integer);
begin
    receive y from in;
    while (true)
        cobegin
            receive x from in;
            begin
                receive r from req;
                send y to out
            end
        coend;
        cobegin
            receive y from in;
            begin
                receive r from req;
                send x to out
            end
        coend
    end
end;

```



Слика 10. Улазни и излазни портови бафера *B*

## 49

## Комуникација са поузданим везама

Језик *Pascal* проширен је комуникационим примитивама са директним именовањем за асинхроно слање облика

```
send <порука> to <процес>
```

и за синхрони (блокирајући) пријем облика

```
receive <порука> from <процес>
```

Језик је такође проширен и алтернативном (селективном) командом облика

```
select A1 or A2 or ... or An endselect
```

$A_i$  су алтернативе, од којих се само једна може изабрати за извршавање. Свака алтернатива је облика

```
<услов> => <акција>
```

где <услов> може бити операција пријема или *delay(t)*, при чему је  $t > 0$ . Алтернатива условљена са *delay(t)* бира се ако и само ако током периода  $t$  није могла бити изабрана ниједна друга алтернатива. Сматрајући да су задате примитиве поуздане, тј. да нема губитака порука, у описаном језику треба:

- a) Реализовати следеће комуникационе операције:

```
function send_swt(Q:process, m1:message, t:duration):boolean;
function receive_swt(P:process, var m1:message):boolean;
```

које остварују временски ограничenu синхрону комуникацију према следећем протоколу: Пошиљалац  $P$  шаље понуду у виду предефинисане поруке *offer* и очекује да прими потврду у виду предефинисане поруке *accept* унутар временског интервала  $t$ . Ако је заиста прими, шаље поруку  $m1$  и враћа вредност *true*, док у супротном шаље опозив у виду предефинисане поруке *cancel* и враћа вредност *false*. Прималац  $Q$  очекује поруку *offer* и одговара са поруком *accept*. Затим прима следећу поруку и ако је то *cancel* поништава пријем (враћа *false*), иначе враћа *true*.

- b) Реализовати бидирекциону трансакцију типа захтев-одговор (*request-reply*) између процеса  $P$  и  $Q$  са асинхроним слањем уз временски ограничен пријем одговора.

## Решење

- a) Временски ограничена синхронна комуникација значи да се поруке морају примити у одређеном временском интервалу или се слање сматра неуспешним. У том случају решење изгледа овако:

```
function send_swt(Q : process, m1 : message, t : duration): boolean;
begin
    send offer to Q
    select
        receive m from Q =>
            if m = accept then begin
                send m1 to Q;
                send_swt := true
            end
            else begin {should never come here}
                send cancel to Q;
                send_swt := false
            end
    end
end
```

```
or
delay(t) =>
    send cancel to Q;
    send_swt := false
endselect;
end;

function receive_swt(P:process, var m1:message):boolean;
begin
    receive m from P;
    if m = offer then begin
        send accept to P;
        receive m1 from P;
        if m1 = cancel then
            receive_swt := false
        else
            receive_swt := true
    end
else begin
    receive_swt := false
end
end;
```

Напомена: Постојање оваквих функција представља основу механизама поуздане комуникације са позитивном потврдом и ретрансмисијом.

б) Код трансакције типа захтев-одговор пошиљалац најпре шаље захтев примаоцу. Прималац прима захтев и шаље одговор пошиљаоцу. Пошиљалац након пријема одговора (ако је стигао у одређеном времену) и наставља са радом.

```
/*process P*/
send request to Q;
select
    receive response from Q => <action1>
or
    delay(t) => <action2>
endselect

/*process Q*/
receive request from P;
send response to P;
```

## Задаци за вежбу

- Посматра се скуп од  $n$  жена и  $n$  мушкараца (*Stable Marriage Problem*). На основу међусобног рангирања потребно је да свака жена и сваки мушкарц нађе свој пар супротног пола. На почетку свака жена разговара са сваким мушкарцем и обрнуто. Након разговора свака жена рангира сваког мушкарца у редослед од 1 до  $n$ ,  $\{m_1, \dots, m_n\}$ , и обрнуто сваки мушкарц рангира сваку жену  $\{w_1, \dots, w_n\}$ . На основу ових рангирања координатор формира парове. Уколико резултујући скуп парова не садржи ниједна два пара  $\{m_i, w_j\}$  и  $\{m_k, w_l\}$  такве да  $m_i$  даје предност  $w_l$  наспрам  $w_j$ , као и да  $w_l$  даје предност  $m_i$  наспрам  $m_k$ , такви парови се називају стабилни.
- Посматра се скуп чворова у графу који могу да комуницирају само са својим суседима (*Distributed Pairing Problem*). Сваки чвор треба да нађе чвор са којим ће се упарити. На крају поступка упаривања сваки чвор ће или имати свог пара, или остати неупарен, али тако да ни једна два суседна чвора не остану неупарена.

## Синхроно прослеђивање порука (CSP)

Код *Communicating Sequential Processes* (*CSP*) програмског модела, како и само име говори, програми се састоје од више секвенцијалних процеса који међусобно комуницирају прослеђивањем порука. Комуникација је потпуно синхронна, тј. процес који шаље поруку се блокира док она не стигне процесу примаоцу, односно процес који прима поруку се блокира док порука не стигне.

Основне синтаксни елементи *CSP* модела описани су у наставку:

### Паралелна команда

Паралелна команда служи за специфицирање процеса који се извршавају конкурентно и има следећи облик:

```
[ proc1::impl1 || ... || procN::implN ]
```

где *proc<sub>i</sub>* означава име секвенцијалног процеса, а *impl<sub>i</sub>* одговарајућу имплементацију (процеси се у овом контексту зову и *корушине*). Команда се успешно завршава када се успешном изврше сви процеси.

Имплементација процеса може бити наведена директно унутар паралелне наредбе у виду низа наредби, а може бити и у виду назива за одговарајући низ наредби који се дефинише на другом месту.

Из назва процеса могу се налазити и заграде у оквиру којих је наведена листа целобројних вредности (нпр. *proc(1)*, *proc(1,2)* итд.). У том случају и назив и листа представљају јединствени идентификатор, тј. име процеса, при чему их други процеси могу референцирати тако што ће у листи на месту целобројних вредности стављати целобројне константе или променљиве.

Постоји скраћени начин да се декларишу процеси чије име се разликује само у индексу иза назива: *[X(i:1..n)::proc]* је исто што и *[X(1)::proc(1) || ... || X(n)::proc(n)]*. Пошто се дефиниција процеса *proc(i)* наводи на само једном месту у програму, на први поглед изгледа да након експанзије имена процеси *proc(1), ..., proc(n)* садрже идентичне дефиниције. Међутим, у оквиру имплементације процеса могуће је користити индекс из његовог назива као променљиву (у овом примеру то је променљива *i*). Ова променљива ће пре извршавања паралелне команде, приликом експанзије имена, у телу процеса бити замењена одговарајућом константом из опсега наведеног у декларацији, тако да ће сваки од процеса имати другу вредност на месту те променљиве.

### Наредба доделе

Израз у *CSP* моделу може по структури бити прост или сложен. Прост израз је било која константа, променљива, аритметички или логички израз. Сложен израз се представља у виду низа простих израза одвојених запетом и уоквирених заградом испред којих може стајати тзв. конструктор:

```
constructor(expr1,...,exprN)
```

Заграде нису потребне ако нема конструктора и постоји само један израз у оквиру поруке.

Примери:

x

(x, y)

calc(c, 2\*x+5, 1)

Вредност сложеног израза након израчунавања такође је по структури сложена. Структура израчунате вредности и конструктор су исти као код израза пре израчунавања, а вредности појединачних компонената резултата одговарају израчунатој вредности одговарајућих компонената из израза. Структурирана вредност са конструктором, али без компонената назива се *сигнал* (нпр.  $P()$ ). Уз помоћ сигнала се обавља синхронизација процеса.

Код доделе изрази са обе стране морају да буду „компабилни“. Ако је са леве стране знака доделе нека променљива, израз на десној страни може бити и прост и сложен израз. Вредност променљиве ће постати једнака вредности сложеног израза. Ако је са леве стране знака доделе неки сложен израз, онда изрази са обе стране морају имати исти конструктор и исту структуру. Приликом доделе, најпре ће се израчунати вредност израза са десне стране знака доделе, а онда ће се вредности елемената израчунатог израза доделити одговарајућим елементима израза који се налази са леве стране. Јасно, елементи који примају вредност на левој страни морају бити променљиве како би могли да прихвате израчунате вредности.

Примери:

x := x+1	вредност x након доделе је за један већа од вредности x пре доделе
(x, y) := (y, x)	x и y размењују вредност
x := cons(left, right)	креира се сложена вредност и додељује променљивој x
cons(left, right) := x	ако x има структуру облика $cons(a,b)$ онда се врши додела $left:= a$ и $right:= b$ ; у супротном долази до грешке
insert(n) := insert(2*x+1)	еквивалентно са $n := 2*x+1$
c := P()	променљивој c се додељује вредност сигнала $P()$
P() := c	није дозвољено јер сигнал не може добити вредност
insert(n) := has(n)	није дозвољено јер су конструктори различити

### Комуникационе примитиве

Могућа је само синхронна *point-to-point* комуникација, уз директно именовање процеса. Примитиве су следеће:

- $<process>!<message>$  – слање поруке  $<message>$  процесу  $<process>$
- $<process>?<message>$  – пријем поруке  $<message>$  од процеса  $<process>$

До преноса поруке долази само ако наредба пријема као извор наводи име процеса пошиљаоца, а процес пошиљалац као одредиште наводи процес прималац, при чему

су одговарајући изрази у наредби за слање и пријем компатibilни (као код доделе вредности)

Порука  $<message>$  може бити било који прост или сложен израз. Код пријема и слања се може применити механизам упаривања (*pattern matching*) ради филтрирања порука према садржају приликом пријема.

Примери:

X!m	пошаљи вредност променљиве $m$ (која може бити скалар, низ, итд.) процесу $X$
X?m	прими поруку од процеса $X$ и смести је у променљиву $m$
X! (3*a+b, 13)	пошаљи поруку процесу $X$ у виду сложеног израза са два елемента
X? (x, y)	прими поруку од процеса $X$ у виду сложеног израза са два елемента и додели одговарајуће вредности променљивама $x$ и $y$
console(j-1) ! "A"	пошаљи знак "A" $j-1$ -вом у низу процеса <i>console</i>
console(i) ?c	прими вредност од $i$ -тог у низу процеса <i>console</i> и додели је променљивој $c$
X(i) ?V()	прими сигнал $V()$ од $i$ -тог у низу процеса $X$ , тј. чекај да процес $X$ пошаље сигнал $V()$ и дотле не примај друге поруке
sem! P()	пошаљи сигнал $P()$ процесу <i>sem</i>

## Контролне структуре

Од основних контролних структура CSP има алтернативну и репетитивну алтернативну команду:

Алтернативна команда има следећи облик:

```
[ guard1 → statement list1
□ guard2 → statement list2
...
□ guardn → statement listn
]
```

*guard* одређује када сме да се изврши одређени низ наредби (*statement list*) које тај *guard* „чува“ (отуда и назив *guard* (енг.) – чувати; у даљем тексту ће се користити еквивалентна реч – услов). Низ наредби се извршава само ако је одговарајући услов задовољен. Ако постоји више услова који су задовољени, онда се између њих недетерминистички (тј. на случајан начин, односно начин одређен имплементацијом и непознат програмеру) бира један чији ће низ наредби бити извршен. Ако ни један услов није задовољен, излази се из алтернативне команде.

Услов се може састојати из три типа израза: декларација променљивих, логичких израза и наредбе пријема порука. Наредба пријема порука мора бити последња. Декларација променљиве унутар услова уводи нову променљиву чија је област важења од места декларације до краја низа наредби које чува дати услов. Услов се израчунава тако што се изрази који га чине извршавају редом са лева на десно. Да би услов био задовољен логички израз мора бити тачан и мора бити примљена порука од

наведеног процеса у оквиру наредбе пријема. Ако неки процес заврши са радом, сви услови у којима се налази наредба пријема од датог процеса не могу бити задовољени.

Алтернативна команда има и скраћени облик:

$[(i:1..n) \text{ guard} \rightarrow \text{statement list}]$

У том случају све алтернативе ће имати исти изглед, осим што се индекс ( $i$  у примеру) може користити као променљива унутар услова или низа наредби. Индекс ће, као и код паралелне наредбе, бити замењен одговарајућом вредношћу приликом експанзије алтернативне команде.

Репетитивна алтернативна команда (често се назива и итеративна команда) има веома сличну синтаксу као и обична алтернативна команда, с том разликом да се алтернативна команда понавља све док се не деси да ни један услов није задовољен:

```
* [ guard1 → statement list1
  □ guard2 → statement list2
  ...
  □ guardn → statement listn
]
```

Примери наредби:

- Одређивање минимума два броја,  $x$  и  $y$   
 $[x \geq y \rightarrow \text{max} := x \quad \square y > x \rightarrow \text{max} := y]$
- Претрага низа  $content$  да би се испитало да ли садржи број  $n$   
 $i := 0; * [i < size; content(i) \neq n \rightarrow i := i+1]$
- Прихваташње знака од процеса  $X$  и прослеђивање процесу  $Y$ . Наредба престаје са радом или када се процес  $X$  заврши или када се утврди да процес  $X$  више не може да пошаље ниједан знак.  
 $* [X?c \rightarrow Y!c]$
- Прихваташње знака са једног од десет процеса  $console$  и прослеђивање процесу  $X$  заједно са одговарајућим бројем конзоле. Сваки пут када се прими знак од конзоле шаље се назад потврда у виду сигнала  $ack()$ . Наредба престаје са радом када стигне знак  $signoff$ .  
 $* [(i : 1..10) \text{continue}(i); console(i)?c \rightarrow X!(i,c); console(i)!ack(); \text{continue}(i) := (c \neq signoff)]$
- Семафор  $S$  коме могу да приступају процеси  $X(1)...X(N)$   
 $S ::= \text{val : integer; val=1;}$   
 $* [(i : 1..N) X(i)?V() \rightarrow \text{val} := \text{val} + 1$   
 $\quad \square \text{val} > 0; (i : 1..N) X(i)?P() \rightarrow \text{val} := \text{val} - 1 ]$

## 50

## Реализација семафора

Проектовати семафор користејќи програмски језик CSP.

### Решење

#### Прво решење

```
v : integer; v := 0;
*[ 
    v > 0; (i : 1..100) X(i)?wait() →
        v := v - 1
    □ (i : 1..100) X(i)?signal() →
        v := v + 1
]
```

#### Друго решење

```
v : integer; v := 0;
(i : 1..100) X(i)?init(v) →
*[ 
    v > 0; (i : 1..100) X(i)?wait() →
        v := v - 1
    □ (i : 1..100) X(i)?signal() →
        v := v + 1
]
```

#### Треће решење

```
v : integer; v := 0;
[
    (i : 1..100) X(i)?init(v) → LOOP
    □ (i : 1..100) X(i)?signal() →
        v := 1;
        LOOP
]
LOOP :: *[ 
    v > 0; (i : 1..100) X(i)?wait() →
        v := v - 1
    □(i : 1..100) X(i)?signal() →
        v := v + 1
]
```

## 51 Реформатирање текста (*Conway's Problem*)

Написати:

- процес *COPY* који прослеђује знаке добијене од процеса *west* процесу *east*.
- процес *SQUASH* модификовањем решења из тачке а) тако да се сваке две суседне звездице замене стрелицом (сматрати да последњи знак није звездица).
- процес *DISASSEMBLE* који чита картице (које садрже по 80 знакова) које добија од процеса *cardfile* и прослеђује процесу *X* низ знакова које картица садржи. На крају сваке картице треба додати још по један бланко знак.
- процес *ASSEMBLE* који добија низ знакова од процеса *X* и штампа их на штампачу по 125 знакова у реду. Последњу линију допунити бланко знацима по потреби.
- програм који чита картице са по 80 знакова на свакој и штампа их на линијском штампачу по 125 знакова у реду. Сваку картицу треба да следи један бланко, а последњи ред треба да буде допуњен бланко знацима по потреби.
- програм са истим захтевима као у тачки д) али тако да се сваке две звездице замењују стрелицом (*Conway's Problem*).

### Решење

- `COPY :: *[c : character; west?c → east!c]`
- `SQUASH :: *[c : character; west?c →
 [ c ≠ "*" → east!c;
 □ c = "*" → west?c;
 [ c ≠ "*" → east!"*"; east!c;
 □ c = "*" → east!"↑" ]
 ]
 ]`
- `DISASSEMBLE :: *[cardimage : (1..80)character; cardfile?cardimage →
 i : integer; i := 1;
 *[i≤80 → X!cardimage(i); i := i + 1]; X!" " ]`
- `ASSEMBLE :: lineimage : (1..125)character;
 i : integer; i := 1;
 *[c : character; X?c → lineimage(i) := c;
 [ i ≤ 124 → i := i + 1
 □ i = 125 → lineprinter!lineimage; i := 1 ]
 ];
 [ i = 1 → skip
 □ i > 1 → *[i ≤ 125 → lineimage(i) := " "; i := i + 1];
 lineprinter!lineimage
 ]
]`
- `[west :: DISASSEMBLE||X :: COPY||east :: ASSEMBLE]`
- `[west :: DISASSEMBLE||X :: SQUASH||east :: ASSEMBLE]`

**52****Пријем, обрада и слање низа знакова**

Реализовати процес *FORMAT* који од процеса *SRC* прима знак по знак, обрађује примљене знаке и тако обрађене их прослеђује процесу *DST*. Обрада се састоји у томе:

- да се свака два или више бланко знакова замењују само једним и да се први небланко знак иза тачке замењује великом словом, а сви остали знаци до следеће тачке малим словом. Претпоставити да се улазни низ не завршава бланко знаком. На располагању су функције *upper(c)* и *lower(c)* које слово с конвертују у одговарајуће велико, односно мало слово.
- да се сваких *N* или више узастопних истих знакова замене секвенцом @<знак>(број понављања знака). Улазни низ се завршава контролним знаком *CR* кога не треба проследити процесу *DST*.

Написати процесе *FORMAT* за случајеве под а) и б).

**Решење**

a)

```
FORMAT :: 
blanks : boolean; blanks := false;
UCASE : boolean; UCASE := false;
*[ c : character; SRC?c →
  [ c=" " → blanks := true;
  □ c=". ." → UCASE := true; [blanks → DST!" "]; DST!". .";
    blanks := false;
  □ c≠". ." ; c≠". ." →
    [ blanks → DST!" " ];
    blanks := false;
    [ UCASE → DST!upper(c); UCASE := false
    □ not(UCASE) → DST!lower(c); ]
  ]
]
```

б)

```
FORMAT :: 
cbuf : character; cbuf := CR;
count : integer; count := 0;
OK : boolean; OK := true;
*[ c : character; OK; SRC?c →
  [ c≠CR, count=0 → cbuf := c; count := 1 //only the first time
  □ c≠CR, count<N →
    [ c=cbuf → count := count+1;
    □ c≠cbuf → i : integer; i := 0;
      *[i<count → DST!cbuf; i := i+1]
      cbuf=c; count := 1
    ];
  □ c≠CR, count≥N →
```

```
[ c=cbuf → count := count+1;
  □ c≠cbuf → DST!"@"; DST!cbuf; DST!"("; DST!count;
    DST!")"; cbuf := c; count := 1 ];
□ c=CR →
  [ count=1 → DST!cbuf
  □ count<N → [ c=cbuf → count := count+1;
    □ c≠cbuf → i : integer; i := 0;
      *[i<count → DST!cbuf; i := i+1]
    ];
  □ count≥N → DST!"@"; DST!cbuf; DST!"("; DST!count;
    DST!")"
  ];
OK := false
]
}
```

## 53

## Потпрограми: Остатак при дељењу

Написати процес који :

- a) представља функцијски потпрограм и који од процеса  $X$  прихватава позитиван дељеник и делитец и враќа њихов целобројни количник и остатак при дељењу. Ефикасност није у првом плану. Шта треба урадити како би овај потпрограм могао да користи и процес  $Y$ ?
- б) има исту функцију као у тачки а), али тако да могу да га позивају процеси  $X(1)...X(n)$ .

### Решење

- a) Решење које је овде дато је са сукцесивним одузимањем и стога је неефикасно. Главни циљ је да се илуструје како се помоћу процеса може реализовати функцијски потпрограм :

```
DIV :: *[x,y : integer; X?(x,y) →
        quot,rem : integer; quot := 0; rem := x;
        *[rem≥y → rem := rem-y; quot := quot+1];
        X!(quot, rem)
    ]
```

Главни програм:

```
[ division :: DIV || X :: USER ]
```

Да би потпрограм  $DIV$  могао да користи и процес  $Y$ , потребно је додати још једну тачку улaska у потпрограм (*entry point*) додавањем још једне алтернативе у репетитивној алтернативној команди која је идентична већ постојећој само што је  $X$  замењено са  $Y$ .

- б) Настављајући идеју из тачке а), али уз коришћење скраћеног записа долазимо до следећег решења:

```
DIV :: *[x,y : integer; (j : 1..N)X(j)?(x,y) →
        quot,rem : integer; quot := 0; rem := x;
        *[rem≥y → rem := rem-y; quot := quot+1];
        X(j)! (quot, rem)
    ]
```

Главни програм:

```
[ division :: DIV || X(i : 1..N) :: USER ]
```

Индекс унутар потпрограма  $DIV$  намерно је назван другачије ( $j$ ) од индекса у главном програму ( $i$ ), да не би долазило до забуне. Иначе нема разлога због кога оба индекса не би могли да имају исто име, јер су им различите области важења.

## 54 Рекурзија: Факторијел

Написати програм који рачуна факторијел.

### Решење

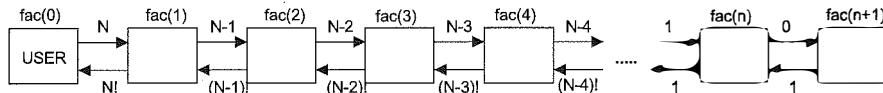
a) Решење које је овде дато демонстрира примену рекурзије:

```
[ fac(i : 1..maxdepth) ::  
  *[n : integer; fac(i-1)?n →  
   [ n=0 → fac(i-1)!1;  
     □ n>0 → fac(i+1)!n-1; r : integer; fac(i+1)?r; fac(i-1)!n*r  
   ]  
  ]  
 || fac(0) :: USER ]
```

Рекурзија се може симулирати низом процеса, по један за сваки ниво рекурзије. Кориснички процес је на нивоу нула. Сваки процес на одређеном нивоу рекурзије прима параметре од процеса са претходног нивоа и ако је потребно прослеђује их процесу на следећем нивоу. Позив рекурзивног потпрограма би у општем случају изгледао овако:

```
[ recsub(0) :: USER || recsub(i : 1..maxdepth) :: RECSUB ]
```

На слици је илустровано како изгледа развијање рекурзије на примеру факторијела.



**Слика 11.** Рекурзивни факторијел. Кориснички процес је на нивоу 0. Сваки ниво рекурзије се симулира једним процесом који прима параметре од процеса са претходног нивоа и шаље резултат процесу на следећем нивоу ако је то потребно, тј. ако се није стигло до краја рекурзије.

## 55

## Ератостеново сито

Написати програм који проналази и штампа у растућем поретку првих  $N=101$  простих бројева (*The Sieve of Eratosthenes*).

## Решење

Идеја решења је следећа: формира се низ процеса тако да први процес у низу филтрира све бројеве деливе са 2, а преостале бројеве прослеђује другом процесу; други процес филтрира све бројеве деливе са другим простим бројем (3), а преостале шаље трећем процесу итд. Сваки процес штампа први број који добије (који је прост), остале умношке тог броја филтрира, а преостале бројеве шаље следећем процесу. Тако сваки следећи процес добија све филтриранји низ бројева. Када последњи ( $N+1$ -ви по реду) процес прими прост број од свог претходника, престаје се са радом.

Први процес *SIEVE(0)* након што одштампа први прост број (2), почиње да генерише непарне бројеве, које потом шаље даље следећем процесу. Сви остали процеси раде по следећем алгоритму: број који први добију од претходног процеса је сигурно прост и он се одмах штампа (назовимо га *prime*). Сваки следећи пут када стигне нови број од претходног процеса (назовимо тај број  $m$ ) испитује се да ли је он умножак од *prime* тако што се нађе највећи умножак простог броја *prime* који је мањи или једнак од пристиглог броја  $m$  (назовимо тај број са *mprime*). То се ради у петљи где се у свакој итерацији *mprime* увећава за *prime*, све док *mprime* не постане већи или једнако  $m$ . Онда се  $m$  и *mprime* упореде. Ако је  $m$  умножак од *prime* (тј. ако су  $m$  и *mprime* једнаки) онда се он филтрира, а ако није (тј. ако је  $m$  мање од *mprime*) онда се шаље следећем процесу. Када процес *SIEVE(N)* добије последњи прост број од свог претходника, штампа га и шаље сигнал процесу *SIEVE(0)* да може да престане са радом. Након тога сви остали процеси, редом, престају са радом.

```
[ SIEVE(0) ::  
    print!2; n : integer; n := 3; b : boolean; b := true;  
    *[ b → SIEVE(1)!n; n := n+2  
        □ b; SIEVE(100)?done() → b := false ]  
|| SIEVE(i : 1..99) ::  
    prime,mprime : integer;  
    SIEVE(i-1)?prime;  
    print!prime;  
    mprime := prime; //multiple of the prime  
    *[ m : integer; SIEVE(i-1)?m →  
        * [m>mprime → mprime := mprime+prime];  
        [ m=mprime → skip //filter out  
        □ m<mprime → SIEVE(i+1)!m //send further ]  
    ]  
|| SIEVE(100) ::  
    n : integer; SIEVE(99)?n → print!n; SIEVE(0)!done()  
    *[SIEVE(99)?n → skip]  
|| print ::*[ (i : 0..100) n : integer; SIEVE(i)?n → /*print n*/ ]
```

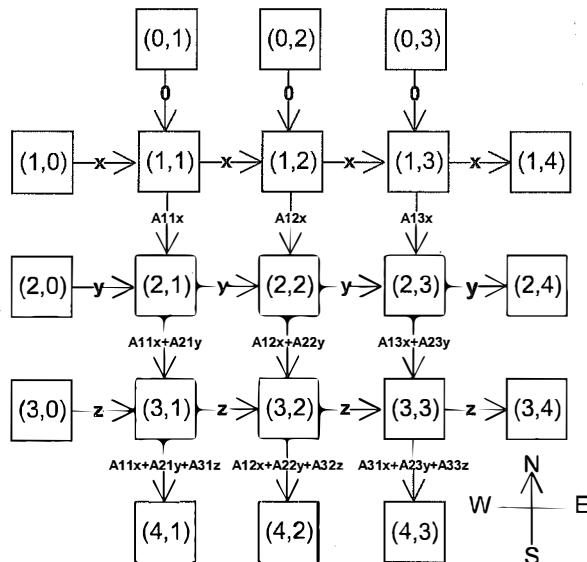
Ово је пример паралелизације рада применом тзв. јроточне обраде (*pipelining*). Сваки процес представља један проточни степен, примајући податке од претходног и прослеђујући их наредном. Сви степени могу да раде упоредо. У овом случају су сви

степени проточне обраде исти, тако да би се принцип рада овог програма могао подвести и под тзв. *сисшоличку обраду* (види следећи задатак).

## 56

## Множење матрица

Дата је квадратна матрица  $A$  реда 3. На улаз долазе три низа података, од којих сваки представља по један елемент вектора  $IN$ . На излазу треба да се појаве три низа података, од којих сваки представља по један елемент производа  $IN \times A$ . Висок ниво паралелизма се може постићи скупом процеса као са слике 7. Сваки неивични процес узима компоненту вектора са запада и парцијалну суму са севера. Сваки од њих прослеђује компоненту вектора на исток, а ажурирану суму на југ. Улазни подаци долазе из западних ивичних чворова, а резултати се шаљу у јужне ивичне чворове. Северни чворови служе као изврз нула, а источни као рупа без дна. После почетног кашњења, резултати се производе истом брзином којом подаци стику на улаз. Матрица  $A$  је фиксна. Реализовати програм који ради према овом алгоритму.



Слика 12. Множење вектора  $IN$  и матрице  $A$  помоћу процеса који симулирају систоличку мрежу.

$$C = IN \times A = [x \ y \ z] \times \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = [C_1 \ C_2 \ C_3]$$

На овакав начин се две квадратне матрице реда 3 могу помножити у три циклуса: у сваком циклусу би се множила по једна врста улазне матрице  $IN$  са матрицом  $A$  и добијала једна врста резултата.

$$C = IN \times A = \begin{bmatrix} (IN_{11} \ IN_{12} \ IN_{13}) \\ (IN_{21} \ IN_{22} \ IN_{23}) \\ (IN_{31} \ IN_{32} \ IN_{33}) \end{bmatrix} \times \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} (C_{11} \ C_{12} \ C_{13}) \\ (C_{21} \ C_{22} \ C_{23}) \\ (C_{31} \ C_{32} \ C_{33}) \end{bmatrix}$$

## Решење

Постоји дадесет и један чвор, коју су организовани у пет група: централна група има 9 чворова и четири групе на ивицама имају по 3 чвора. Западну групу чине улазни кориснички процеси, а јужну групу чине излазни кориснички процеси.

```
[ M(i : 1..3, 0) :: WEST
|| M(0, j : 1..3) :: NORTH
|| M(i : 1..3, 4) :: EAST
|| M(4, j : 1..3) :: SOUTH
|| M(i : 1..3, j : 1..3) :: CENTER
]
NORTH :: *[true → M(1,j)!0]
EAST :: *[x : real; M(i,3)?x → skip]
CENTER :: *[x : real; M(i,j-1)?x →
            M(i,j+1)!x
            sum : real;
            M(i-1,j)?sum;
            M(i+1,j)! (A(i,j)*x+sum)
        ]
WEST :: *[i : 1..3) x : real; IN(i)?x → M(i,1)!x]
SOUTH :: *[result : real; M(3,j)?result → skip]
```

Ово је пример паралелизације рада применом тзв. *систоличке обраде*. Код систоличке обраде постоји велики број ћелија истог типа које су распоређене у виду правилне мреже, при чему ћелије могу да комуницирају само са својим непосредним суседима. Обрада је синхрони, а резултати се након иницијалног кашњења појављују на излазу оним темпом којим се улазни подаци доводе на улаз мреже.

Концепт систоличке обраде у облику тзв. *систоличких низова* (*systolic arrays*) се често користи у хардверским системима за убрзавање и паралелизацију обраде зато што су везе између ћелија кратке и има их релативно мало (постоји веза само са суседним ћелијама), а све ћелије су исте тако да се може постићи велика густина паковања на чипу. Из истих разлога (ћелија има мало веза, а сви процеси су исти) систоличку обраду је лако имплементирати у CSP моделу тако што се свака систоличка ћелија представи у виду једног процеса, што је урађено и у овом примеру.

Код синхроних систоличких поља, да би се постигло генерисање резултата у истом циклусу, неопходно је да се елементи вектора *IN* примају померени за по један циклус. Најраније се прима *x*, па *y*, па *z*. Исто тако, елементи резултата генеришу се са померајем од једног циклуса.

## 57

## Читаоци и писци

Користећи CSP написати програм који решава проблем читалаца и писаца.

### Решење

#### Прво решење

Читаоци и писци на почетку упућују захтев мониторском процесу слањем одговарајућег захтева, *startread* односно *startwrite*. Пошто се ради о синхроној комуникацији ово решење је тако реализовано да чим мониторски процес прихвати овај захтев могу започети са операцијом читања односно писања. На крају обавештавају мониторски процес да је операција завршена слањем одговарајућег захтева, *endread* односно *endwrite*. На мониторској страни проверава се услов за пријем одговарајућег захтева односно рад сваке од операција. За пријем захтева *startread*, односно започињање операције читања потребно је да број писаца буде једнак 0. За пријем захтева *startwrite*, односно започињање операције уписа потребно је да број читалаца и број писаца који тренутно приступају буде једнак нули. Код пријема захтева за завршетак операције, *endread* односно *endwrite*, није потребно проверавати услове. Треба приметити да пошто се не води евиденција о процесима који чекају ово решење има исте особине као прво решења у задатку 5, а које је било *reader's preference solution* јер даје предност читаоцима.

```
[ Reader (i : 0..nr) :: READER || Writer (i : 0..nw) :: WRITER ||
Monitor :: MONITOR]

READER :: * [
    true → [Monitor!startread();
        READING;
        Monitor!endread()
    ]
]

WRITER :: * [
    true → [Monitor!startwrite();
        WRITING;
        Monitor!endwrite()
    ]
]

MONITOR :: [ numR, numW : integer; numR := 0; numW := 0;
* [
    numW = 0; (i : 0..nr) Reader(i)?startread() →
        numR := numR + 1
    □ (i : 0..nr) Reader(i)?endread() →
        numR := numR - 1
    □ numW = 0; numR = 0; (i : 0..nw) Writer(i)?startwrite() →
        numW := numW + 1
    □ (i : 0..nw) Writer(i)?endwrite() →
        numW := numW - 1
]
]
```

### Друго решење

Како би се избегло изгладњивање писаца потребно је на неки начин обезбедити да монитор сазна да ли је писац упутио захтев за започињањем писања, пре него што се тај захтев прихвати. Овде се треба присетити да CSP користи синхрону коминикацију између процеса. Ово значи да се процес који је упутио захтев блокира док процес који прима захтев тај захтев не прихвати. Како би се омогућило да са пријем захтева и његово одобравање развоји потребно је да се део за започињање операције уписа подели на два дела, први у коме се шаље захтев за добијањем дозволе и други у коме се чека дозвола за приступ. На страни монитора ако услов за започињање операције писања није задовољен, неко већ чита или пише, прелази се део где се чека да сви процеси који су претходно започели читање или писање не заврше са том операцијом. Када се заврше све претходно започете операције дати писац који је чекао се обавештава да може да започне са операцијом писања.

```
[ Reader (i : 0..nr) :: READER || Writer (i : 0..nw) :: WRITER ||
Monitor :: MONITOR]
```

```
READER :: * [
    true → [
        Monitor!startread();
        READING;
        Monitor!endread()
    ]
]

WRITER :: * [
    true → [
        Monitor!startwrite();
        Monitor?ok();
        WRITEING;
        Monitor!endwrite()
    ]
]

MONITOR :: [ numR, numW : integer; numR := 0; numW := 0;
* [
    numW = 0; (i : 0..nr) Reader(i)?startread() → [
        numR := numR + 1]
    □ (i : 0..nr) Reader(i)?endread() →
        numR := numR - 1
    □ numW = 0; numR = 0; (i : 0..nw) Writer(i)?startwrite() → [
        numW := numW + 1;
        Writer(i)!ok()]
    □ (i : 0..nw) Writer(i)?endwrite() →
        numW := numW - 1
    □ numW <> 0; (i : 0..nw) Writer(i)?startwrite() → [
        (j : 0..nw) Writer(j)?endwrite();
        Writer(i)!ok()]
    □ numR <> 0; (i : 0..nw) Writer(i)?startwrite() →
        *[ numR <> 0;
            (j : 0..nr) Reader(j)?endread() →
                numR := numR - 1;
                [ numR = 0 → [
                    numW := numW + 1;
```

```

                Writer(i)!ok()]
            ]
        ]
    ]
]
```

Треба приметити да се у односу на прво решење код овог решења може десити да ако неки писац жели да пише онда добија право да пише без обзира колико има читалаца који чекају, *writers's preference solution*.

### Треће решење

Како би се избегло изгладњивање и читалаца и писаца уводи се потпуно симетрично решење. Операција започињања и читања и писања је подељена на два дела. Прво се тражи дозвола па се онда чека да се дозвола добије.

```

[ Reader (i : 0..nr) :: READER || Writer (i : 0..nw) :: WRITER ||
Monitor :: MONITOR]

READER :: *[  

    true → [  

        Monitor!startread();  

        Monitor?ok();  

        READING;  

        Monitor!endread()  

    ]  

]  
  

WRITER :: *[  

    true → [  

        Monitor!startwrite();  

        Monitor?ok();  

        WRITEING;  

        Monitor!endwrite()  

    ]  

]  
  

MONITOR :: [ numR, numW : integer; numR := 0; numW := 0;  

* [  

    numW = 0; (i : 0..nr) Reader(i)?startread() → [  

        numR := numR + 1; Reader(i)!ok()]  

    □ numW <> 0; (i : 0..nr) Reader(i)?startread() → [  

        (j : 0..nw) Writer(j)?endwrite() → [  

            numW := numW - 1;  

            numR := numR + 1;  

            Reader(i)!ok()]])  

    □ (i : 0..nr) Reader(i)?endread() →  

        numR := numR - 1  

    □ numW = 0; numR = 0; (i : 0..nw) Writer(i)?startwrite() → [  

        numW := numW + 1;  

        Writer(i)!ok()]  

    □ numW <> 0; (i : 0..nw) Writer(i)?startwrite() → [  

        (j : 0..nw) Writer(j)?endwrite();
```

```
Writer(i)!ok()]
□ numR <> 0; (i : 0..nw) Writer(i)?startwrite() →
    *[ numR <> 0;
        (j : 0..nr) Reader(j)?endread() →
            numR := numR - 1;
            [ numR = 0 → [
                numW := numW + 1;
                Writer(i)!ok()]
            ]
    ]
□ (i : 0..nw) Writer(i)?endwrite() →
    numW := numW - 1
]
]
```

58

Произвођачи и потрошачи

Решити проблем производача и потрошача.

## Решење

```
[Producer (i:1..Np)::PRODUCER || Consumer (i:1..Nc)::CONSUMER ||  
Buffer::BUFFER]  
  
BUFFER:: [ buffer : (0..B-1) portion;  
          in, out : integer; in := 0; out := 0;  
          *[  
              in<out + B; (i: 1..Np) Producer (i)?buffer(in mod B) →  
                  in:=in + 1  
          ]  
          out < in; (i: 1..Nc) Consumer (i)?get() → [  
              Consumer (i)!buffer(out mod B) → out := out + 1]  
          ]  
      ]  
  
PRODUCER:: *[ true → [data : portion;  
                        PRODUCE;  
                        Buffer!data  
                      ]  
      ]  
  
CONSUMER:: *[ true → [data : portion;  
                         Buffer!get();  
                         Buffer?data;  
                         CONSUME  
                       ]  
      ]
```

## 59 Филозофи за ручком

Решити проблем филозофа за ручком, описан у задатку 4.

### Решење

Прво решење:

Могуће је адаптирати било које решење из задатка 4 за реализацију која користи CSP, јер се у њему могу директно реализовати семафори (видети пример из уводног дела). Међутим, тиме би се изгубило на ефикасности у односу на решења која би користила инхерентне предности синхроне комуникације прослеђивањем порука која постоји у CSP моделу. Узимајући то у обзир једно ефикасно решење би било оно које одговара трећем решењу из задатка 4.

Виљушке се уместо као дељене променљиве, представљају као процеси. Примењен је и исти механизам за избегавање мртвог блокирања (*deadlock*), увођењем процеса *room*. Овај процес приhvата захтеве од филозофа, проверава услов и уколико су испуњени услови да филозоф затражи виљушку, а да не дође до мртвог блокирања, потврђује филозофу да може започети са тражењем виљушки. Уколико услов није испуњен овај процес памти приспели захтев и када услов постане испуњен обавештава филозофа који чека.

```
[philosopher(i : 0..N-1) :: PHILOSOPHER || fork(i : 0..N-1) :: FORK
|| room :: ROOM]

PHILOSOPHER :::
left,right : integer;
left := i; right := (i+1) mod N;
*[ true →
    THINK;
    //entry protocol
    room!enter();
    room?ok();
    fork(left)!pickup(); fork(right)!pickup();
    EAT;
    //exit protocol
    fork(left)!putdown(); fork(right)!putdown();
    room!exit();
]

FORK :::
left,right : integer;
left := i; right := (i+1) mod N;
*[ philosopher(left)?pickup() → philosopher(left)?putdown()
□ philosopher(right)?pickup() → philosopher(right)?putdown() ]

ROOM :::
tickets : integer; tickets := 0;
waiting : boolean; waiting := false;
id : integer;
*[ (i : 0..N-1)philosopher(i)?enter() →
    [ tickets < N-1 → tickets := tickets+1; philosopher(i)!ok()
```

```
□ tickets=N-1 → waiting := true; id := i ]
□ (i : 0..N-1) philosopher(i)?exit() →
    tickets := tickets-1;
    [waiting → waiting := false; philosopher(id)!ok()]
]
```

#### Друго решење:

Као и у претходном решењу и овде су виљушке процеси. Такође постоји и процес *room* који омогућава избегавање мртвог блокирања. За разлику од претходног решења овај процес приhvата захтеве од филозофа само када је услов за спречавање мртвог блокирања испуњен. Ово значи да нема потребе за слањем потврде филозофу јер се ради о синхроној комуникацији.

```
[philosopher(i : 0..N-1) :: PHILOSOPHER || fork(i : 0..N-1) :: FORK
|| room :: ROOM]

PHILOSOPHER :: [ left, right : integer;
    left := i; right := (i+1) mod N;
    *[true → [THINK;
        room!enter();
        fork(rihgt)!pickup();
        fork(left)!pickup();
        EAT;
        fork(left)!putdown();
        fork(right)!putdown();
        room!exit()
    ]
]
]

FORK :: [ left, right : integer; left := (i+N-1) mod N; right := i;
    *[ philosopher(left)?pickup() → philosopher(left)?putdown()
        □ philosopher(right)?pickup() → philosopher(right)?putdown()
    ]
]
]

ROOM :: [tickets : integer; tickets := N-1;
    *[ [
        tickets > 0; (i : 0..N-1) philosopher(i)?enter() →
            tickets := tickets - 1
        □ (i : 0..N-1) philosopher(i)?exit() →
            tickets := tickets + 1
    ]
]
]
```

#### Треће решење

За разлику ос претходног решења, које је пошло од решења са семафорима, у овом решењу нема процеса виљушки. Постоји само процес *room* који одговара монитору. Монитор приhvата захтев од филозофа само ако том филозофу стоје на располагању две виљушке. Уколико нема две виљушке које су тренутно доступне процес не приhvата захтев од филозофа, а филозоф због синхроне комуникације чека. Ово значи да нема потребе за слањем потврде филозофу јер се ради о синхроној комуникацији.

```
[philosopher (i : 0..N-1) :: PHILOSOPHER || room :: ROOM ]  
  
PHILOSOPHER :: [ left, right : integer;  
    left := i; right := (i + 1) mod N;  
    *[true → [  
        THINK;  
        room!getForks();  
        EAT;  
        room!putForks()  
    ]  
]  
]  
  
ROOM :: [ forks : (0..N-1) integer;  
    forks(0) := 2; forks(1) := 2; forks(2) := 2;  
    forks(3) := 2; forks(4) := 2;  
  
    *[ [  
        (i : 0..N-1) forks(i) = 2; philosopher(i)?getForks() → [  
            forks((i+1) mod N) := forks((i+1) mod N)-1;  
            forks((i+N-1) mod N) := forks((i+N-1) mod N)-1  
        ]  
        □ (i : 0..N-1) philosopher(i)?putForks() → [  
            forks((i+1) mod N) := forks((i+1) mod N)+1;  
            forks((i+N-1) mod N) := forks((i+N-1) mod N)+1  
        ]  
    ]  
]
```

## 60

## Бинарно стабло

$N$  процеса  $node(i:1..N)$  се користе као чворови модификованих бинарног уређеног стабла код кога је  $node(1)$  корен. Сваки од њих има локалне променљиве  $left$  и  $right$  које представљају индексе левог, односно десног потомка,  $parent$  која представља индекс родитеља, као и локалну променљиву  $value$  која представља вредност одговарајућег чвора стабла. Уколико леви или десни потомак или родитељ не постоје, вредност одговарајућег индекса је 0. Стабло је уређено тако да су вредности  $value$  свих чворова из левог подстабла увек мањи од вредности  $value$  коју има дати чвор. Подаци се налазе у листовима стабла, а остали чворови представљају приступни део стабла. Процеси смеју да комуницирају само са својим потомцима и родитељем, као и са чворм 0. Процес  $node(0)$  је посреднички процес између корисничких процеса  $U(i:1..N_U)$  и кореног процеса стабла  $node(1)$ . Реализовати процедуре за  $node(0)$  и  $node(i:1..N)$  користећи CSP тако да корисник може извршити следеће наредбе

$has(x)$  – испитује да ли се  $x$  налази у стаблу,

$delete(x)$  – брише  $x$  из стабла,

$insert(x)$  – уметаје  $x$  у стабло.

Обезбедити да процес  $node(0)$  прими наредну наредбу тек када се текућа наредба заврши. Дати пример коришћења ових наредби од стране корисничких процеса.

## Решење

Кључни посао код свих наведених наредби јесте претрага стабла; код брисања и уметања је, наравно, након проналаска траженог чвора, потребно извршити још и превезивање показивача родитељског чвора. Претрага стабла се врши прослеђивањем одговарајућих порука потомцима, почевши од корена. Треба приметити да укупни број чворова бинарног стабла мора бити мањи или једнак  $N$ , при чему број листова износи приближно  $N/2$ .

Код претраживања на страни интерфејса треба водити рачуна о томе да ли постоји barem један елемент у стаблу. Уколико постоји захтев се прослеђује корену стабла,  $node(1)$ , а одговор се очекује од првог чвора који утврди да ли се дата вредност налази у стаблу. Овај одговор се прослеђује кориснику који је упутио захтев. Уколико нема елемената у стаблу кориснику се одмах јавља резултат претраге.

```
[U(i: 1..Nu) ::USER || node(0) ::INTERFACE || node(i: 1..N) :: NODE]

USER :: 
//has(x) - example of usage
x : integer; x := ...; result : boolean;
node(0)!has(x);
node(0)?has(result);
[ result → //x is in the tree
  □ not (result) → //x is not in the tree
]

INTERFACE :: size : integer;
INIT
*[ x : integer; (i : 1..Nu) U(i)?has(x) →
  [size = 0 → U(i)!has(false)
```

```

□ size>0 → node(1) !has(x);
  [ b : boolean; (j : 1..N) node(j) ?has(b) → U(i) !has(b) ]
]

```

Код претраживања на страни чворова у стаблу се проверава да ли је вредност мања, већа или једнака оној вредности која је сачувана у чвиру, као и о повезаности датог чвора и суседа. Уколико је вредност већа,  $x > value$ , проверава се да ли постоји десни потомак. Ако постоји њему се прослеђује захтев за претрагом. Ако не постоји тада се чвиру 0 враћа да вредност није пронађена. Уколико је вредност мања,  $x < value$ , проверава се да ли постоји леви потомак. Ако постоји њему се прослеђује захтев за претрагом. Ако не постоји тада се чвиру 0 враћа да вредност није пронађена. Уколико је вредност једнака,  $x = value$ , треба разликовати два случаја. Први случај је да се ради о чвиру у приступном стаблу, а други случај је да се ради о листу. Уколико се ради о чвиру из приступног стабла,  $right \neq 0$ , захтев се прослеђује десном потомку. Треба приметити да није могуће одмах вратити позитиван одговор јер је могућа ситуација да нека вредност постоји у приступном стаблу, али да се та вредност не налази у листовима. Уколико се ради о листу,  $right = 0 \text{ AND } left = 0$ , враћа се позитиван резултат претраге. Овде треба приметити да је по начину конструкције стабло формирano тако да се подаци налазе у листовима и да се чворови који имају само једног потомка бришу. Ово значи да ако је  $right = 0$  онда је и  $left = 0$  тако да је уместо провере целог услова,  $right = 0 \text{ AND } left = 0$ , довољно проверити да ли је  $right = 0$ .

```

NODE :: 
left,right,parent : integer; value : integer;
*[ x : integer; node(parent) ?has(x) →
  [ x>value →
    [ right=0 → node(0) !has(false) //x is not in the tree
      □ right≠0 → node(right) !has(x)
    ]
  □ x<value →
    [ left=0 → node(0) !has(false) //x is not in the tree
      □ left≠0 → node(left) !has(x)
    ]
  □ x=value →
    [ right=0 → node(0) !has(left=0) // left is always 0!
      □ right≠0 → node(right) !has(x)
    ]
]
]

```

Код брисања и уметања треба водити рачуна да се промена стања у стаблу мора вршити атомски, јер се у супротном може десити да брисање чвора није завршено, а да буде прослеђен нови захтев и да дође до колизије. Ако овај проблем посматрамо као аналоган проблему читалаца и писаца, при чему је стабло заједнички ресурс, операције  $has(x)$  је аналогна читању, а операције  $insert(x)$  и  $delete(x)$  аналогне писању (мењају садржај ресурса). Онда је јасно да је потребно обезбедити међусобно искључивање „писаца”, као и међусобно искључивање „читалаца” и „писаца”. Међусобно искључивање између „писаца”, као и међусобно искључивање између „писаца” и „читалаца” обезбеђено је на тај начин што је посреднички процес реализован као алтернативна команда која приhvата и извршава само један захтев у једном тренутку и не приhvата нови захтев све док се претходни не заврши. Завршетак обраде захтева сигнализира се слањем одговарајућег сигнала. Притом „писци” шаљу сигнал тек када се заврши превезивање чворова, а „читаоци” чим се утврди да ли дати

чвор постоји или не. По услову задатка као и у тачки а) и између „читалаца“ влада међусобно искључивање.

```
[U(i: 1..Nu):: USER || node(0) :: INTERFACE || node(i: 1..N):: NODE]

USER : :
//has(x) - example of usage
x : integer; x := ...; result : boolean;
node(0)!has(x);
node(0)?has(result);
[result → //x is in the tree
□ not (result) → //x is not in the tree
]

//delete(x) - example of usage
x : integer; x := ...; result : boolean;
node(0)!delete(x);
node(0)?deleted(result)
[result → //x successfully deleted from the tree
□ not (result) → //x was not deleted from the tree
]

//insert(x) - example of usage
x : integer; x := ...; result : boolean;
node(0)!insert(x);
node(0)?inserted(result)
[result → //x successfully inserted into the tree
□ not (result) → //x was not inserted into the tree
]

INTERFACE : :
size : integer; size := 0;
*[x : integer; (i : 1..Nu)U(i)?has(x) →
 [size = 0 → U(i)!has(false)
 □ size > 0 →
     node(1)!has(x);
     b : boolean; (j : 1..N)node(j)?has(b);
     U(i)!has(b)
 ]
□ x : integer; (i : 1..Nu)U(i)?delete(x) →
 [size = 0 → U(i)!deleted(false)
 □ size > 0 →
     node(1)!delete(x);
     [b : boolean; node(1)?deleted(b) →
      //root element was deleted
      U(i)!deleted(b);
      size := 0
     □ b : boolean; n1, n2 : integer;
      (j : 1..N)node(j)?deleted(b, n1, n2) →
      U(i)!deleted(b);
      free(n1);
      free(n2);
      size := size - 2
    ]
]
```

```

    ]
□ x : integer; (i : 1..Nu)U(i)?insert(x) →
[size = 0 → // adding root element
    node(1)!setval(x);
    size := 1;
    U(i)!inserted(true)
□ size > 0, size < N - 2 →
    n1, n2 : integer;
    n1 := free();
    n2 := free();
    node(1)!insert(x, n1, n2);
    b : boolean; (j : 1..N)node(j)?inserted(b);
    U(i)!inserted(b);
    [b →
        size := size + 2
    □ not(b) →
        free(n1);
        free(n2)
    ]
□ size >= N - 2 → U(i)!inserted(false)//no more free space
]
]

NODE :: :
left,right,parent : integer; value : integer;
*[x : integer; node(parent)?has(x) →
[x > value →
    [right = 0 → node(0)!has(false) //x is not in the tree
     □ right ≠ 0 → node(right)!has(x)
    ]
□ x < value →
    [left = 0 → node(0)!has(false) //x is not in the tree
     □ left ≠ 0 → node(left)!has(x)
    ]
□ x = value →
    [right = 0 → node(0)!has(left = 0) // left is always 0!
     □ right ≠ 0 → node(right)!has(x)
    ]
]
□ x : integer; node(parent)?delete(x) →
[x > value →
    [right = 0 → node(0)!deleted(false)
     □ right ≠ 0 → node(right)!delete(x)
    ]
□ x < value →
    [left = 0 → node(0)!deleted(false)
     □ left ≠ 0 → node(left)!delete(x)
    ]
□ x = value →
    [right = 0 → node(parent)!deleted(true); parent := 0
     //if right == 0 => left = 0
     □ right ≠ 0 → node(right)!delete(x)
    ]
]
□ b : boolean; node(left)?deleted(b) →

```

```

oldLeft, oldRight : integer;
oldLeft := left;
oldRight := right;
node(right)!remove();
node(right)?removed(value, left, right);
node(0)!deleted(true, oldLeft, oldRight)
□ b : boolean; node(right)?deleted(b) →
    oldLeft, oldRight : integer;
    oldLeft := left;
    oldRight := right;
    node(left)!remove();
    node(left)?removed(value, left, right);
    node(0)!deleted(true, oldLeft, oldRight)
□ node(parent)?remove() →
    [left ≠ 0 → node(left)!setparent(parent)]
    [right ≠ 0 → node(right)!setparent(parent)]
    node(paren)!removed(value, left, right);
    parent := 0;
    left := 0;
    right := 0
□ x : integer; (j : 1..N)node(j)?setparent(x) → parent = x
□ x : integer; (j : 1..N)node(j)?setval(x) → value = x
□ x, nl, n2 : integer; (j : 1..N)node(j)?insert(x, nl, n2) →
    [left = 0, right = 0 →
        [x ≠ value →
            node(i1)!setval(x);
            node(i1)!setparent(i);
            node(n2)!setval(value);
            node(n2)!setparent(i);
            [x > value →
                value := x;
                right := nl;
                left := n2
            □ x <= value
                right := n2;
                left := n1
            ]
            node(0)!inserted(true)
        □ x = value → node(0)!inserted(false)
    ]
    □ left ≠ 0 OR right ≠ 0 →
        [x >= value → node(right)!insert(x, nl, n2)
        □ x < value → node(left)!insert(x, nl, n2)
    ]
]
]

```

## 61

## Бинарно стабло - конкурентно претраживање

У систему описаном у претходном задатку потребно је омогућити конкурентно претраживање већег броја вредности у стаблу.

## Решење

У претходном решењу омогућено је да само један процес у неком тренутку чита (претражује) или пише (умеће или брише). Ово се постигло закључавањем целог стабла и код операције претраживања и код операција уметања и брисања. Како би се дозволило више операција претраживања потребно је да се операција претраживања подели на два дела. Први део је слање захтева за операцијом и други је добијање одговора на дати захтев. Пошто је операција претраживања подељена на два дела, приликом пријема одговора мора се знати на чији је захтев тај одговор приспео. Како би се ово сазвало потребно је код слања захтева за операцију претраживања поред самог захтева проследити и идентификатор онога корисника који тражи операцију претраживања,  $node(1)!has(x, i)$ . Овај идентификатор је потребно све време прослеђивати док се не заврши претраживање и формира одговор. У том одговору је потребно вратити резултат претраживања као и идентификатор онога које затражио операцију. Операција брисања или уметања може да започне тек када сви они који су обављали операцију читања заврше са читањем.

```

...
INTERFACE_WITH_DEADLOCK ::

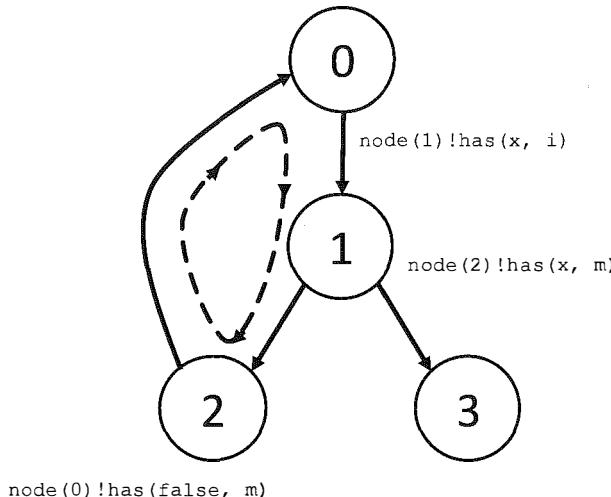
INIT
*[ x : integer; (i : 1..Nu)U(i)?has(x) →
  [size = 0 → U(i)!has(false)
   □ size>0 → node(1)!has(x, i)
  ]
  □ b : boolean; m : integer; (j : 1..N)node(j)?has(b, m) →
    U(m)!has(b)
]

NODE ::

left,right,parent : integer; value : integer;
*[ x, m : integer; node(parent)?has(x, m) →
  [x>value →
    [right=0 → node(0)!has(false, m) //x is not in the tree
     □ right≠0 → node(right)!has(x, m)
    ]
  □ x<value →
    [left=0 → node(0)!has(false, m) //x is not in the tree
     □ left≠0 → node(left)!has(x, m)
    ]
  □ x=value →
    [right=0 → node(0)!has(left=0, m) // left is always 0!
     □ right≠0 → node(right)!has(x, m)
    ]
]

```

Ово решење знатно повећава конкурентност али у случају синхроне комуникације, што је случај код CSP језика, може да доведе до мртвог блокирања. Проблем настаје у ситуацијама када сви процеси у ланцу комуникације чекају да друга страна дође до места где може да прими поруку. На пример, посматра се граф који се састоји из интерфејса, *node(0)*, корена, *node(1)*, и два листа, *node(2)* и *node(3)*, као што је дато на слици. Нека је прво пристигао захтев број 1 који је чвор 0 проследио чвору 1. У време док је чвор 1 обрађивао и прослеђивао први захтев чвору 2 пристигне други захтев чвору 0. Чвор 0 овај захтев проследи чвору 1, док чвор 2 обрађује захтев број 1. У међувремену пристигне и трећи захтев чвору 0. Сада чвор 2 покушава да пошаље одговор чвору 0, пошто чвор 0 чека да пошаље трећи захтев чвору 1 он не може да прими одговор од чвора 2. Истовремено чвор 1 покушава да проследи други захтев чвору 2 али не може јер чвор 2 чека чвор 0, слика 13. На овај начин је настало мртво блокирање.



Слика 13. Пример ситуације са мртвим блокирањем

Један од начина да се избегне мртво блокирање је да се избегне сценарио по коме се сви чворови у ланцу од интерфејса до листа међусобно чекају. У случају да унапред није позната структура графа, као ни вредности које се налазе у појединим чворовима, ово се може постиће увођењем максималног броја конкурентних претрага, *MAX*. Овај број одговара минималној висини стабла. На овај начин је избегнута ситуација да сви чворови у ланцу чекају приликом слања поруке.

```

...
INTERFACE :: 
nHas : integer; nHas := 0;
INIT
*[x : integer; nHas < MAX, (i : 1..Nu)U(i) ?has(x) →
 [size = 0 → U(i)!has(false)
  □ size>0 → node(1)!has(x, i);nHas := nHas + 1]
□ (j : 1..N) b : boolean; m : integer; node(j)?has(b, m) →
 U(m)!has(b);
 nHas := nHas - 1
]

```

Треба приметити да до ове ситуације не може доћи у случају асинхроне комуникације јер се код ње процес који шаље поруку не блокира уколико процес који прима поруку не може дату поруку да прими.

Како би се одредила вредност променљиве **MAX**, минималне висине стабла, потребно је приликом уметања и брисања елемента у стабло ажурирати ову променљиву. Захтев за одређивање минимална висина стабла упућује интерфејс кореном чврлу по добијању информације да је операција уметања или брисања успешно обављена. Чворови графа на рекурзивни начин одређују висину. Уколико се ради о листовима они свом родитељу враћају вредност 0. Чворови у приступном стаблу прослеђују захтеве левом и десном потомку, а као резултат за висину до датог чвора се узима мања вредност која се добије од потомака. Ову вредност чворови приступног стабла увећавају за један и прослеђује родитељу.

```
[U(i: 1..Nu):: USER || node(0) :: INTERFACE || node(i: 1..N):: NODE]
USER :
//has(x) - example of usage
x : integer; x := ...; result : boolean;
node(0)!has(x);
node(0)?has(result);
[result → //x is in the tree
□ not (result) → //x is not in the tree
]

//delete(x) - example of usage
x : integer; x := ...; result : boolean;
node(0)!delete(x);
node(0)?deleted(result)
[result → //x successfully deleted from the tree
□ not (result) → //x was not deleted from the tree
]

//insert(x) - example of usage
x : integer; x := ...; result : boolean;
node(0)!insert(x);
node(0)?inserted(result)
[result → //x successfully inserted into the tree
□ not (result) → //x was not inserted into the tree
]

INTERFACE :
size, nHas : integer; size := 0; nHas := 0;
*[ x : integer; nHas < MAX, (i : 1..Nu)U(i)?has(x) →
    [size = 0 → U(i)!has(false)
     □ size > 0 →
        node(1)!has(x, i);
        nHas := nHas + 1
    ]
  □ b : boolean; m : integer; (j : 1..N)node(j)?has(b, m) →
    U(m)!has(b);
    nHas := nHas - 1
  □ x : integer; (i : 1..Nu)U(i)?delete(x) →
    *[nHas > 0 →
      b : boolean; m : integer; (j : 1..N)node(j)?has(b, m);
      U(m)!has(b);
```

```

nHas := nHas - 1
]
[size = 0 → U(i)!deleted(false)
□ size > 0 →
    node(1)!delete(x);
    [b : boolean; node(1)?deleted(b) →
        //root element was deleted
        U(i)!deleted(b);
        size := 0;
        MAX = 1
    □ b : boolean; n1, n2 : integer;
    (j : 1..N)node(j)?deleted(b, n1, n2) →
        U(i)!deleted(b);
        free(n1);
        free(n2);
        size := size - 2;
        node(1)!getMax();
        node(1)?max(MAX)
    ]
]
□ x : integer; (i : 1..Nu)U(i)?insert(x) →
    *[nHas > 0 →
        b : boolean; m : integer; (j : 1..N)node(j)?has(b, m);
        U(m)!has(b);
        nHas := nHas - 1
    ]
[size = 0 → // adding root element
    node(1)!setval(x);
    MAX := 1;
    size := 1;
    U(i)!inserted(true)
    □ size > 0, size < N - 2 →
        n1, n2 : integer;
        n1 := free();
        n2 := free();
        node(1)!insert(x, n1, n2);
        b : boolean; (j : 1..N)node(j)?inserted(b);
        U(i)!inserted(b);
        [b →
            node(1)!getMax();
            node(1)?max(MAX);
            size := size + 2
        □ not(b) →
            free(n1);
            free(n2)
        ]
    □ size >= N - 2 → U(i)!inserted(false)//no more free space
    ]
]
NODE : :
left,right,parent : integer; value : integer;
*[x : integer; node(parent)?has(x) →
    [x > value →
        [right = 0 → node(0)!has(false) //x is not in the tree

```

```

□ right ≠ 0 → node(right) !has(x)
]
□ x < value →
    [left = 0 → node(0) !has(false) //x is not in the tree
     □ left ≠ 0 → node(left) !has(x)
    ]
□ x = value →
    [right = 0 → node(0) !has(left = 0) // left is always 0!
     □ right ≠ 0 → node(right) !has(x)
    ]
]
□ x : integer; node(parent)?delete(x) →
[x > value →
    [right = 0 → node(0) !deleted(false)
     □ right ≠ 0 → node(right) !delete(x)
    ]
□ x < value →
    [left = 0 → node(0) !deleted(false)
     □ left ≠ 0 → node(left) !delete(x)
    ]
□ x = value →
    [right = 0 → node(parent) !deleted(true); parent := 0
     //if right == 0 => left = 0
     □ right ≠ 0 → node(right) !delete(x)
    ]
]
□ b : boolean; node(left)?deleted(b) →
oldLeft, oldRight : integer;
oldLeft := left;
oldRight := right;
node(right)!remove();
node(right)?removed(value, left, right);
node(0)!deleted(true, oldLeft, oldRight)
□ b : boolean; node(right)?deleted(b) →
oldLeft, oldRight : integer;
oldLeft := left;
oldRight := right;
node(left)!remove();
node(left)?removed(value, left, right);
node(0)!deleted(true, oldLeft, oldRight)
□ node(parent)?remove() →
[left ≠ 0 → node(left)!setparent(parent)]
[right ≠ 0 → node(right)!setparent(parent)]
node(paren)!removed(value, left, right);
parent := 0;
left := 0;
right := 0
□ x : integer; (j : 1..N)node(j)?setparent(x) → parent = x
□ x : integer; (j : 1..N)node(j)?setval(x) → value = x
□ x, n1, n2 : integer; (j : 1..N)node(j)?insert(x, n1, n2) →
[left = 0, right = 0 →
    [x ≠ value →
        node(i1)!setval(x);
        node(i1)!setparent(i);
        node(n2)!setval(value);
    ]
]
```

```
node(n2)!setparent(i);
[x > value →
    value := x;
    right := n1;
    left := n2
    □ x ≤ value
        right := n2;
        left := n1
    ]
    node(0)!inserted(true)
    □ x = value → node(0)!inserted(false)
]
□ left ≠ 0 OR right ≠ 0 →
    [x ≥ value → node(right)!insert(x, n1, n2)
     □ x < value → node(left)!insert(x, n1, n2)
    ]
]
□ node(parent)?getMax() →
    [left = 0, right = 0 → node(parent)!max(0)
     □ left ≠ 0 OR right ≠ 0 →
        node(left)!getMax();
        ml : integer; node(left)?max(ml);
        node(right)!getMax();
        mr : integer; node(right)?max(mr);
        [ml < mr → node(parent)!max(ml + 1)
         □ ml ≥ mr → node(parent)!max(mr + 1)
        ]
    ]
]
```

## 62 Скупови

Реализовати скуп користећи CSP тако да кориснички процеси могу задати следеће наредбе:

- *has(x)* – испитује да ли је *x* елемент скупа
  - *insert(x)* – умети *x* у скуп
  - *delete(x)* – брише *x* из скупа
  - *size()* – враћа број елемената скупа
  - *scan()* – чита један по један све елементе скупа
  - *min()* – враћа минимални елемент скупа
  - *max()* – враћа максимални елемент скупа
  - *clear()* – брише све елементе скупа
- a) скуп представити у виду процеса *S* коме може приступати кориснички процес *U*. Скуп може имати највише *N* целобројних елемената и иницијално је празан.
- b) повећати конкурентност решења проблема из тачке а) тако што ће скуп (од највише *N<sub>S</sub>* елемената) бити реализован у виду низа процеса *S(i:1.. N<sub>S</sub>)*, од којих сваки чува највише један елемент скупа. Приступ скому истовремено може тражити више корисничких процеса *U(i:1.. N<sub>U</sub>)*.

Дати пример коришћења наредби од стране корисничког процеса.

## Решење

### а) Прво решење

Код овог решења елементи скупа се чувају у низу *content*, а број елемената скупа се чува у променљивој *size*. Елементи скупа се налазе на првих *size* позиција у низу и нису сортирани.

Испитивање припадности скупу са *has(x)* врши се тако што се пролази кроз низ и за сваки елемент низа испита да ли је једнак *x*, све док се елемент не пронађе или се не дође до kraja низа.

Уметање са *insert(x)* се врши тако што се најпре испита да ли цео број *x* већ постоји у скому (секвенцијалном претрагом), па ако се установи да не постоји онда се убаци на крај низа, а број елемената скому се повећа за 1.

Брисање са *delete(x)* се врши тако што се најпре испита да ли цео број *x* постоји у скому (секвенцијалном претрагом), па ако се установи да постоји онда се брише тако што се елемент са последњег места убаци на његово место, а број елемената скума се смањи за 1.

Скенирање скума са *scan()* се врши тако што се, почевши од првог елемента скума, позивајућем процесу шаље вредност тог елемента, а потом иде (секвенцијално) на следећи итд. све док се не дође до последњег елемента скума.

Минимум и максимум се налазе проласком (секвенцијално) кроз све елементе скума и тражењем најмањег, односно највећег међу њима.

Скуп се брише тако што се величина скума *size* постави на 0.

```
[ U :: USER || S :: SET ]      //main program
```

```
USER ::  
//has(x) - example of usage  
x : integer; x := ...; result : boolean;  
S!has(x); S?has(result);  
[ result → //x is a member of the set  
□ not(result) → //x is not a member of the set  
]  
  
//insert(x) - example of usage  
c : integer; result : boolean;  
S!size(); S?size(c);  
[ c<N → x : integer; x := ...; S!insert(x); S?insert(result);  
    [ result → //x successfully inserted into set  
        □ not(result) → //x is already a member of the set  
    ]  
□ c=N → //set is full  
]  
  
//delete(x) - example of usage  
c : integer; result : boolean;  
S!size(); S?size(c);  
[ c>0 → x : integer; x := ...; S!delete(x); S?delete(result);  
    [ result → //x successfully deleted from the set  
        □ not(result) → //x is not a member of the set  
    ]  
□ c=0 → //set is empty  
]  
  
//size() - example of usage  
c : integer;  
S!size(); S?size(c);  
  
//scan() - example of usage  
S!scan();  
more : boolean; more := true;  
*[ more; m : integer; S?scan(m) → //deal with the element m  
    □ more; S?scan(nomore) → more := false //no more elements  
]  
  
//min() - example of usage  
m : integer;  
c : integer;  
S!size(); S?size(c);  
[ c>0 → S!min(); S?min(m) //get minimum  
□ c=0 → //set is empty  
]  
  
//max() - example of usage  
m : integer;  
c : integer;  
S!size(); S?size(c);  
[ c>0 → S!max(); S?max(m) //get maximum  
□ c=0 → //set is empty  
]
```

```
//clear() - example of usage
S!clear();

SET ::

N : integer; N := N; content : (0..N-1)integer;
size : integer; size := 0;
*[ x : integer; U?has(x) → HAS
□ x : integer; U?insert(x) → INSERT
□ x : integer; U?delete(x) → DELETE
□ U?size() → U!size(size)
□ U?scan() → SCAN
□ U?min() → MIN
□ U?max() → MAX
□ U?clear() → size := 0
]

SEARCH :: i : integer; i := 0; *[i<size; content(i)≠x → i := i+1]

HAS :: SEARCH; U!has(i<size)

INSERT ::

SEARCH;
[ i<size → U!insert(false) //x is already a member of the set
□ i=size → content(size) := x;
           size := size+1;
           U!insert(true) //x is inserted into set
]

DELETE ::

SEARCH;
[ i=size → U!delete(false) //x is not a member of the set
□ i<size → content(i) := content(size);
           size := size-1;
           U!delete(true) //x is deleted from the set
]

SCAN ::

i : integer; i := 0;
*[ i<size → U!element(content(i)); i := i+1 ]
U!scan(nomore)

MIN ::

m : integer; m := content(0); i : integer; i := 1;
*[ i<size; content(i)<m → m := content(i); i := i+1 ]
U!min(m)

MAX ::

m : integer; m := content(0); i : integer; i := 1;
*[ i<size; content(i)>m → m := content(i); i := i+1 ]
U!max(m)
```

### Друго решење

Као и код првог решења, елементи скупа се држе у низу *content*, број елемената скупа се чува у променљивој *size*, а елементи скупа се налазе на првих *size* позиција у низу. Разлика је једино у томе што се елементи скупа у низу држе сортирани, ради брже претраге.

Испитивање припадности скупу са *has(x)* се врши тако што се ради бинарно претраживање сортираног низа.

Уметање са *insert(x)* се врши тако што се најпре испита да ли цео број *x* већ постоји у скупу (бинарном претрагом), па ако се установи да не постоји онда се убаци на одговарајуће место у низу тако што се сви елементи после датог помере за једно место улево (како би се очувала сортираност), а број елемената скупа се повећа за 1.

Брисање са *delete(x)* се врши тако што се најпре испита да ли цео број *x* постоји у скупу (бинарном претрагом), па ако се установи да постоји онда се брише тако што се сви елементи после датог помере за једно место улево (како би се очувала сортираност), а број елемената скупа се смањи за 1.

Минимални и максимални елементи су једнаки првом, односно последњем елементу низа.

```
SET ::  
N : integer; N := N; content : (0..N-1)integer;  
size : integer; size := 0;  
*[ x : integer; U?has(x) → HAS  
□ x : integer; U?insert(x) → INSERT  
□ x : integer; U?delete(x) → DELETE  
□ U?size() → U!size(size)  
□ U?scan() → SCAN  
□ U?min() → U!(content(0))  
□ U?max() → U!(content(size-1))  
□ U?clear() → size := 0  
]  
  
SEARCH ::  
low : integer; low := 0;  
high : integer; high := size-1;  
i : integer; i := int((high+low)/2);  
b : boolean; b := true;  
*[ b; low<high; x<content(i) → high := i-1; i := int((high+low)/2)  
□ b; low<high; x>content(i) → low := i+1; i := int((high+low)/2)  
□ b; x=content(i) → b := false //element found  
]  
  
HAS :: SEARCH; U!has(not(b))  
  
INSERT ::  
SEARCH;  
[ not(b) → U!insert(false) //x is already a member of the set  
□ b → j : integer; j := size-1;  
    *[ j>i → content(j) := content(j-1); j := j-1 ]  
    content(i) := x;  
    size := size+1;  
    U.insert(true) //x is inserted into set  
]
```

```
DELETE ::  
SEARCH;  
[ b → U!delete(false) //x is not a member of the set  
□ not(b) → j : integer; j := i;  
    *[ j<size → content(j) := content(j+1); j := j+1 ]  
    size := size-1;  
    U!delete(true) //x is deleted from the set  
]  
]
```

б) Прво решење:

Елементи скупа се чувају у процесима  $S(i:1.. size)$  и нису сортирани. Да би се корисничким процесима обезбедио јасно дефинисан програмски интерфејс уводи се посреднички процес  $S(0)$ . Без посредничког процеса сваки кориснички процес би морао да зна интерну структуру процеса скупа. Осим тога, увођењем посредничког процеса омогућава се међусобно искључивање корисничких процеса приликом приступа скупу. Међусобно искључивање је обезбеђено тиме што је овај процес реализован као алтернативна команда која приhvата и извршава само један захтев у једном тренутку, а нови захтев приhvата тек кад се претходни заврши. Процес  $S(0)$  уједно чува и број елемената скупа у променљивој  $size$ .

Испитивање припадности скупу се врши тако што посреднички процес шаље упит  $has(x)$  свим процесима скупа који имају елементе и истовремено прикупља пристигле одговоре, све док неки од процеса не одговори позитивно или док сви процеси не одговоре негативно.

Уметање елемената  $x$  врши се тако што се најпре испита да ли  $x$  већ постоји у скупу (на претходно описан начин). Ако се установи да не постоји онда се додели првом следећем слободном процесу слањем поруке  $insert(x)$ , а број елемената скупа се повећа за 1.

Брисање елемената  $x$  се врши тако што се најпре испита да ли  $x$  постоји у скупу (на претходно описан начин), па ако се установи да постоји, онда садржај процеса чији се елемент брише постаје једнак садржају процеса који садржи последњи елемент скупа, а број елемената скупа се смањи за 1.

Скенирање скупа се врши тако што посреднички процес шаље упит  $scan()$  свим процесима скупа који имају елементе и истовремено прикупља пристигле одговоре све док не стигну одговори од свих процеса.

Минимум и максимум се налазе претрагом по свим елементима скупа (слично као код скенирања) при чему се памти најмања, односно највећа вредност међу њима.

Скуп се брише тако што се  $size$  у процесу  $S(0)$  постави на 0.

Сви процеси престају са радом када престане да постоји последњи кориснички процес.

Како би се обезбедило да се број елемената скупа не промени од тренутка када се тражи број елемената скупа па до тренутка када се операција који корисник обавља не заврши уводи се тражење величине са закључување,  $S(0)!sizeAndLock()$ , као и откључавање,  $S(0)!unlock()$ .

```
//main program  
[U(i : 1..Nu) :: USER || S(0) :: INTERFACE || S(i : 1..Ns) :: SETEL]  
  
USER ::  
/has(x) - example of usage
```

```
x : integer; x := ...; result : boolean;
S(0)!has(x); S(0)?has(result);
[ result → //x is a member of the set
  □ not(result) → //x is not a member of the set
]
//insert(x) - example of usage
c : integer;
S(0)!sizeAndLock(); S(0)?size(c);
[ c<Ns → x : integer; x := ...; result : boolean;
  S(0)!insert(x); S(0)?insert(result);
  [ result → //x is inserted into set
    □ not(result) → //x is already a member of the set
  ]
  □ c=Ns → //set is full
]
S(0)!unlock()

//delete(x) - example of usage
c : integer;
S(0)!sizeAndLock(); S(0)?size(c);
[ c>0 → x : integer; x := ...; result : boolean;
  S(0)!delete(x); S(0)?delete(result);
  [ result → //x is deleted from the set
    □ not(result) → //x is not a member of the set
  ]
  □ c=0 → //set is empty
]
S(0)!unlock()

//size() - example of usage
c : integer; S(0)!size(); S(0)?size(c);

//scan() - example of usage
S(0)!scan();
more : boolean; more := true; x : integer;
*[ more; S(0)?scan(x) → //deal with the element x
  □ more; S(0)?scannomore() → more := false //no more elements
]

//min() - example of usage
m : integer; c : integer; S(0)!sizeAndLock(); S(0)?size(c);
[ c>0 → S(0)!min(); S(0)?min(m) //get minimum
  □ c=0 → //set is empty
]
S(0)!unlock()

//max() - example of usage
m : integer; c : integer; S(0)!sizeAndLock(); S(0)?size(c);
[ c>0 → S(0)!max(); S(0)?max(m) //get maximum
  □ c=0 → //set is empty
]
S(0)!unlock()

//clear() - example of usage
S(0)!clear();
```

```

INTERFACE ::

size : integer; size := 0;
*[ (i : 1..Nu) x : integer; U(i)?has(x) → HAS
  □ (i : 1..Nu) x : integer; U(i)?insert(x) → INSERT
  □ (i : 1..Nu) x : integer; U(i)?delete(x) → DELETE
  □ (i : 1..Nu) U(i)?size() → SIZE
  □ (i : 1..Nu) U(i)?scan() → SCAN
  □ (i : 1..Nu) U(i)?min() → MIN
  □ (i : 1..Nu) U(i)?max() → MAX
  □ (i : 1..Nu) U(i)?clear() → CLEAR
  □ (i : 1..Nu) U(i)?sizeAndLock() → SIZEANDLOCK
]

SETEL ::

el : integer;
*[ x : integer; S(0)?has(x) → S(0)!has(el=x)
  □ x : integer; S(0)?insert(x) → el := x
  □ S(0)?scan() → S(0)!scan(x)
]

SEARCH ::

j : integer; j := 1;
k : integer; k := 1;
res : boolean; res := false;
b : boolean; b := false;
pos : integer;
*[ j<=size → S(j)!has(x); j := j+1
  □ k<=size; (q : 1..size)S(q)?has(res) →
    [res → b := true; pos := q]; k := k+1
]
HAS :: SEARCH; U(i)!has(b)

INSERT ::

SEARCH;
[ b → U(i)!insert(false) //x is already a member of the set
  □ not(b) → size := size+1; S(size)!insert(x); U(i)!insert(true)
]

DELETE ::

SEARCH;
[ b → x : integer; S(size)!scan(); S(size)?scan(x);
  S(pos)!insert(x); size := size-1; U(i)!delete(true)
  □ not(b) → U(i)!delete(false)
]

SIZE :: U(i)!size(size)

SCAN ::

j : integer; j := 1; k : integer; k := 1; x : integer;
*[ j<=size → S(j)!scan(); j := j+1
  □ (q : 1..size) k<=size; S(q)?scan(x) → U(i)!scan(x); k := k+1
]
U(i)!scannomore();

```

```
MIN ::  
m : integer; S(1)!scan(); S(1)?scan(m);  
j : integer; j := 1; k : integer; k := 1;  
x : integer;  
*[ j<=size → S(j)!scan(); j := j+1  
  □ (q : 1..size) k<=size; S(q)?scan(x) → [ x<m → m := x ]; k := k+1  
]  
U(i)!min(m)  
  
MAX ::  
m : integer; S(1)!scan(); S(1)?scan(m);  
j : integer; j := 1; k : integer; k := 1; x : integer;  
*[ j<=size → S(j)!scan(); j := j+1  
  □ (q : 1..size) k<=size; S(q)?scan(x) → [ x>m → m := x ]; k := k+1  
]  
U(i)!max(m)  
  
CLEAR :: size := 0  
  
SIZEANDLOCK :: SIZE;  
[ x : integer; U(i)?insert(x) → INSERT; U(i)?unlock()  
  □ x : integer; U(i)?delete(x) → DELETE; U(i)?unlock()  
  □ U(i)?min() → MIN; U(i)?unlock()  
  □ U(i)?max() → MAX; U(i)?unlock()  
]  

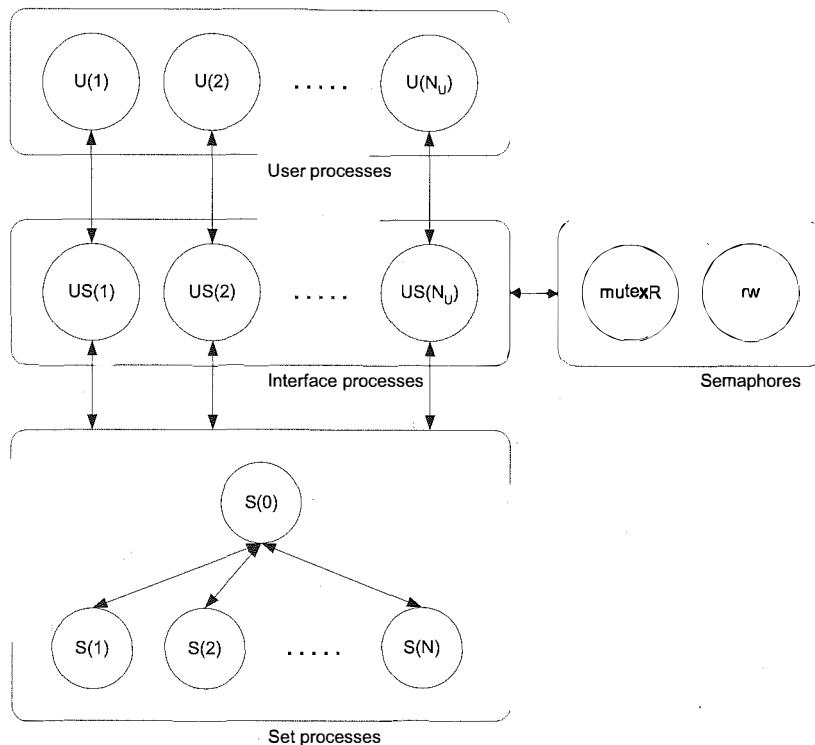
```

#### Друго решење:

Дато решење предвија да само један кориснички процес може да приступа скупу у једном тренутку. Међутим, ако скуп посматрамо као ресурс, а корисничке процесе као писце и читаоце који приступају том ресурсу, конкурентност се може повећати тако што ће читаоци моћи паралелно да врше приступ. Читалац је сваки процес са захтевом којим се не модификује скуп: *has(x)*, *size()*, *scan()*, *min()* и *max()*, док је писац сваки процес који има захтев којим се врши модификација скупа: *insert(x)*, *delete(x)* или *clear()*. Да би кориснички процеси могли да издају захтеве упоредо, при чему би сваки имао исти програмски интерфејс, потребно је увести више посредничких процеса *US(i:1..Nu)*. Број посредничких процеса може бити мањи или једнак броју корисничких процеса (конкурентност је максимална када је њихов број једнак, односно када сваки кориснички процес може да позива различит посреднички процес). Узето је да је број посредничких процеса *Nus* једнак броју корисничких процеса *Nu*. Шема комуникације између процеса дата је на слици 8.

Међусобно искључивање између писаца и међусобно искључивање између читалаца и писаца обезбеђује се по принципу *reader's preference* преко процеса *mutexR* и *rw* који представљају имплементацију семафора.

```
//main program  
[ U(i:1..Nu) :: USER || US(i:1..Nu) :: INTERFACE || S(0) :: SET0 ||  
  S(i:1..Ns) :: SETEL || mutexR :: SEMAPHORE || rw :: SEMAPHORE ]  
  
USER :::  
//has(x) - example of usage  
x : integer; x := ...; result : boolean;  
US(i)!has(result); US(i)?has(result);  
[ result → //x is a member of the set  
  □ not(result) → //x is not a member of the set  
]
```



Слика 14. Шема међусобног позивања процеса.

```

//insert(x) - example of usage
c : integer; US(i)!sizeAndLock(); US(i)?size(c);
[ c<Ns → x : integer; x := ...; result : boolean;
  US(i)!insert(x); US(i)?insert(result);
  [ result → //x is inserted into set
    □ not(result) → //x is already a member of the set
  ]
  □ c=Ns → //set is full
]
//delete(x) - example of usage
c : integer; US(i)!sizeAndLock(); US(i)?size(c);
[ c>0 → x : integer; x := ...; result : boolean;
  US(i)!delete(x); US(i)?delete(result);
  [ result → //x is deleted from the set
    □ not(result) → //x is not a member of the set
  ]
  □ c=0 → //set is empty
]
US(i)!unlock();

```

```
//size() - example of usage
c : integer; US(i)!size(); US(i)?size(c);
//scan() - example of usage
US(i)!scan();
more : boolean; more := true; x : integer;
*[ more; US(i)?scan(x) → //deal with the element x
  □ more; US(i)?scannomore() → more := false //no more elements
]
//min() - example of usage
m : integer; c : integer; US(i)!sizeAndLock(); US(i)?size(c);
[ c>0 → US(i)!min(); US(i)?min(m) //get minimum
  □ c=0 → //set is empty
]
US(i)!unlock();
//max() - example of usage
m : integer; c : integer; US(i)!sizeAndLock(); US(i)?size(c);
[ c>0 → US(i)!max(); US(i)?max(m) //get maximum
  □ c=0 → //set is empty
]
US(i)!unlock();
//clear() - example of usage
US(i)!clear();

SEMAPHORE ::

s : integer; s := 1; f : integer; f := 0; lifo : (1..Ns)integer;
[i : integer; i := 1; *[i<=Ns → lifo(i) := 0; i := i+1]]
*[ (i : 1..Ns) s>0; US(i)?wait() → s := s-1; US(i)!ok()
  □ (i : 1..Ns) s=0; US(i)?wait() → f := f+1; lifo(f) := i
  □ (i : 1..Ns) f=0; US(i)?signal() → s := s+1
  □ (i : 1..Ns) f>0; US(i)?signal() → f := f-1; US(lifo(f+1))!ok()
]

READER_ENTRY ::

mutexR!wait();
mutexR?ok();
nr : integer; S(0)!get(); S(0)?take(nr); S(0)!inc_nr();
[nr=1 → rw!wait(); rw?ok()]
mutexR!signal()

READER_EXIT ::

mutexR!wait();
mutexR?ok();
nr : integer; S(0)!get(); S(0)?take(nr); S(0)!dec_nr();
[nr=0 → rw!signal()]
mutexR!signal()

WRITER_ENTRY ::

rw!wait(); rw?ok()

WRITER_EXIT ::

rw!signal()

INTERFACE ::

*[ (i : 1..Nu) x : integer; U(i)?has(x) →
```

```

READER_ENTRY; HAS; READER_EXIT
□ (i : 1..Nu) x : integer; U(i)?insert(x) →
    WRITER_ENTRY; INSERT; WRITER_EXIT
□ (i : 1..Nu) x : integer; U(i)?delete(x) →
    WRITER_ENTRY; DELETE; WRITER_EXIT
□ (i : 1..Nu) U(i)?size() →
    READER_ENTRY; SIZE; READER_EXIT
□ (i : 1..Nu) U(i)?scan() →
    READER_ENTRY; SCAN; READER_EXIT
□ (i : 1..Nu) U(i)?min() →
    READER_ENTRY; MIN; READER_EXIT
□ (i : 1..Nu) U(i)?max() →
    READER_ENTRY; MAX; READER_EXIT
□ (i : 1..Nu) U(i)?clear() →
    WRITER_ENTRY; CLEAR; WRITER_EXIT
□ (i : 1..Nu) x : integer; U(i)?sizeAndLock(x) →
    WRITER_ENTRY; SIZEANDLOCK; WRITER_EXIT
]

SET0 ::

size : integer; size := 0; nr : integer; nr := 0;
*[ (i : 1..Nus) US(i)?size() → US(i)!size(size)
  □ (i : 1..Nus) US(i)?get() → US(i)!take(nr)
  □ (i : 1..Nus) US(i)?inc_nr() → nr := nr+1
  □ (i : 1..Nus) US(i)?dec_nr() → nr := nr-1
  □ (i : 1..Nus) US(i)?inc_size() → size := size+1
  □ (i : 1..Nus) US(i)?dec_size() → size := size-1
  □ (i : 1..Nus) US(i)?clear() → size := 0
]

SETEL ::

el : integer;
*[ (i : 1..Nus) x : integer; US(i)?has(x) → US(i)!has(el=x)
  □ (i : 1..Nus) x : integer; US(i)?insert(x) → el := x
  □ (i : 1..Nus) US(i)?scan() → US(i)!scan(x)
]

SEARCH ::

c : integer; S(0)!size(); S(0)?size(c);
j : integer; j := 1;
k : integer; k := 1;
res : boolean; res := false;
b : boolean; b := false;
pos : integer;
*[ j<=c → S(j)!has(x); j := j+1
  □ (q : 1..c) k<=c; S(q)?has(res) →
    [res → b := true; pos := q];
    k := k+1
]

HAS :: SEARCH; U(i)!has(b)

INSERT ::

SEARCH;
[ b → U(i)!insert(false) //x already a member of the set

```

```

□ not(b) → c : integer; S(0)!inc_size; S(0)!size();
           S(0)?size(c); S(c)!insert(x); U(i)!insert(true)
]

DELETE ::

SEARCH;
[ b → x : integer; c : integer; S(0)!size(); S(0)?size(c);
  S(c)!scan(); S(c)?scan(x); S(pos)!insert(x);
  S(0)!dec_size(); U(i)!delete(true)
  □ not(b) → U(i)!delete(false)
]

SIZE :: c : integer; S(0)!size(); S(0)?size(c); U(i)!size(c)

SCAN ::

c : integer; S(0)!size(); S(0)?size(c);
j : integer; j := 1;
k : integer; k := 1;
x : integer;
*[ j<=c → S(j)!scan(); j := j+1
  □ (q : 1..c) k<=c; S(q)?scan(x) → U(i)!scan(x); k := k+1
]
U(i)!scannomore()

MIN ::

c : integer; S(0)!size(); S(0)?size(c);
m : integer; S(1)!scan(); S(1)?scan(m);
j : integer; j := 1;
k : integer; k := 1;
x : integer;
*[ j<=c → S(j)!scan(); j := j+1
  □ (q : 1..c) k<=c; S(q)?scan(x) → [ x<m → m := x ]; k := k+1
]
U(i)!min(m)

MAX ::

c : integer; S(0)!size(); S(0)?size(c);
m : integer; S(1)!scan(); S(1)?scan(m);
j : integer; j := 1;
k : integer; k := 1;
x : integer;
*[ j<=c → S(j)!scan(); j := j+1
  □ (q : 1..c) k<=c; S(q)?scan(x) → [ x>m → m := x ]; k := k+1
]
U(i)!max(m)

CLEAR :: S(0)!clear()

SIZEANDLOCK :: SIZE;
[ x : integer; U(i)?insert(x) → INSERT; U(i)?unlock()
  □ x : integer; U(i)?delete(x) → DELETE; U(i)?unlock()
  □ U(i)?min() → MIN; U(i)?unlock()
  □ U(i)?max() → MAX; U(i)?unlock()
]

```

## 63 Израчунавање интеграла

Написати програм за израчунавање интеграла користећи обраду по принципу „торба послова“ (*Parallel Computing With a Bag of Tasks*).

### Решење

Приликом решавања овог проблема разматра ће се обрада по принципу „торба послова“ (*Parallel Computing With a Bag of Tasks*). Детаљи обраде по принципу „торбе послова“ су дати у дати у глави о семафорима у задатку „Произвођачи и потрошачи: комуникација помоћу кружног бафера“. Посао израчунавања интеграла на интервалу од  $X_{min}$  до  $X_{max}$  је издељен на већи број послова мањег обима код којих се раде израчунавање интеграла на крајем интервалу, од  $x$  до  $x+dx$ . На овај начин је израчунавање интеграла подељено на већи број независних послова које је могуће обавити у паралели. Процес који ради обраду (*WORKER*) узима нови посао из торбе (*BAG*) чим заврше са обрадом претходно узетог посла. Интервал на коме сваки од послова ради израчунавање,  $dx$ , рачуна се на основу жељене тачности. Код оваквог решења није унапред познат однос између броја послова ( $N$ ) и броја процеса који ради обраду ( $W$ ).

У општем случају процеси који врше обраду (*WORKER*) на почетку шаље захтев процесу који расподељује послове (*BAG*) да је спреман да обави следећу итерацију рачунања, *Bag!getTask()*. Када се припреми следећи интервал добија се интервал на коме се ради рачунање интеграла, *Bag?getData(left, right)*. Сам поступак израчунања се може описати генералном методом *CALCULATE*. Када се обави израчунавање на задатом интервалу шаље се назад резултат израчунавања, *Bag!putResult(data)*, након чега се поступак понавља.

Процес који расподељује послове на почетку иницијализује опсег на коме се врши интеграција, од  $X_{min}$  до  $X_{max}$ , и број интервала на којима је потребно обавити интеграцију  $N$ . Сам поступак иницијализације није од суштинске важности за даље решавање тако да је представљен методом *INIT*. Након тога се прелази на циклус у коме се на захтев генеришу послови и примају резултати обављених послова. Потребно је омогућити да се резултати интеграције примају чим стигну. Захтеви за обављање послова се примају све док се не покрије цео интервал на коме је потребно радити интеграцију,  $x < X_{max}$ , што је еквивалентно укупном пријему  $N$  захтева за обрадом. Све док је услов за пријем захтева за послом испуњен чека се захтев на који се шаље интервал на коме је потребно обавити интеграцију, након чега се интервал за следећи посао помера унапред. У другој грани алтернативне команде се од процеса који врше обраду очекују резултати. Када се прими резултат увећава се тренутни резултат интеграције као и број примљених резултата. Поступак интеграције се прекида када се приме свих  $N$  резултата.

```
[Worker(p : 1..W) :: WORKER || Bag :: BAG]
WORKER :: [ left, right, data : double;
           *[ true → [
                         Bag!getTask();
                         Bag?getData(left, right);
                         CALCULATE;
                         Bag!putResult(data)
                     ]
           ]
]
```

```
BAG ::      [ Xmin, Xmax, dx, x, F : double; F := 0;
    N, i, j : integer; i := 0; j := 0;
    INIT;
    x := Xmin;
    *[      j < N (k : 1..W)Worker(k)?getTask() -> [
        Worker(k)!getData(x, x+dx); x := x + dx; j := j + 1]
    □ i < N; (k : 1..W)Worker(k)?putResult(data) -> [
        F := F + data; i := i + 1]
    ]
    STOP;
]
CALCULATE :: ...
INIT :: ...
STOP :: ...
```

## 64

## Изградња молекула воде

Користећи CSP написати програм који решава проблем изградње молекула воде (*The H<sub>2</sub>O Problem*). Поред процеса који описују водоник и кисеоник постоји и процес баријера.

### Решење

Проблем изградње молекула воде је проблем код кога је потребно упарити тачно два атома водоника и један атома кисеоника. Ово упаривање се одиграва на баријери на којој се очекује долазак два атома водоника и једног атома кисеоника. Синхронизација између атома водоника и кисеоника који представљају процесе који се извршавају се обавља на баријери која представља синхронизациони процес.

Код и за кисеоник и за водоник изгледа исто и састоји се из два дела: доласка на баријеру одвојених атома и одласка са баријере у оквиру формирања молекула. Долазак на баријеру се остварује спањем поруке *Barrier!Exit()*. Пошто се овде ради о програмском језику CSP, ког кога је размена порука синхронна, атом зна да су се на баријери испунили сви предуслови. Предуслови за кисеоник су да се завршило формирање претходног молекула, да ни један други атом кисеоника није пристигао на баријеру раније и да је пристигло тачно два атома водоника. Предуслов за атом водоника је да се завршило формирање претходног молекула и да је пристигло не више од једног атома водоника. У другој фази атоми од баријере очекују повратну информацију томе да ли се накупило доволично атома са којим могу да начине молекул. Када приме ту информацију, *Barrier?ExitOK()*, ступају у хемијску реакцију, *BOND*. Након обављене реакције јављају баријери да је реакције обављена и да је молекул воде формиран.

Баријере на почетку очекује да се појаве тачно два атома водоника, *id1* и *id2*, и један атом кисеоника, *id3*. Када пристигну сви ови атоми сваки од њих се обавештава да се сакупило доволично атома да би формирали молекул воде. Пре него што баријера пређе на поступак формирања следећег молекула воде чека да сва три атома заврше хемијску реакцију и напусте баријеру.

```
[Hydrogen(h : 1..nh) :: HYDROGEN || Oxygen(h : 1..no) :: OXYGEN ||  
Barrier :: BARRIER]  
  
HYDROGEN : [id1, id2, id3 : integer;  
           Barrier!Exit();  
           Barrier?ExitOK(id1, id2, id3);  
           BOND;  
           Barrier!Confirm()  
]  
  
OXYGEN :: [id1, id2, id3 : integer;  
           Barrier!Exit();  
           Barrier?ExitOK(id1, id2, id3);  
           BOND;  
           Barrier!Confirm()  
]  
  
BARRIER :: [ id1, id2, id3 : integer;
```

```

* [
    Hydrogen(id1 : 1..nh)?Exit() → [
        Hydrogen(id2 : 1..nh)?Exit();
        Oxygen(id3 : 1..no)?Exit();

        Hydrogen(id1)!ExitOK(id1, id2, id3);
        Hydrogen(id2)!ExitOK(id1, id2, id3);
        Oxygen(id3)!ExitOK(id1, id2, id3);

        Hydrogen(id1)?Confirm();
        Hydrogen(id2)?Confirm();
        Oxygen(id3)?Confirm()
    ]
]
STOP;
]

```

Пошто CSP користи синхрону комуникацију између процеса претходни код се може упростити јер није потребно да процес баријера (*Barrier*) посебно прима обавештење од атома да је формирање обављено (*Confirm*) јер су атоми примили идентификаторе свих атома у молекулу и могу учествовати у хемијској реакцији и ван баријере.

```

[Hydrogen(h : 1..nh) :: HYDROGEN || Oxygen(h : 1..no) :: OXYGEN ||
Barrier :: BARRIER]

HYDROGEN : [id1, id2, id3 : integer;
            Barrier!Exit();
            Barrier?ExitOK(id1, id2, id3);
            BOND;
        ]

OXYGEN :: [id1, id2, id3 : integer;
            Barrier!Exit();
            Barrier?ExitOK(id1, id2, id3);
            BOND;
        ]

BARRIER :: [ id1, id2, id3 : integer;
             * [
                 Hydrogen(id1 : 1..nh)?Exit() → [
                     Hydrogen(id2 : 1..nh)?Exit();
                     Oxygen(id3 : 1..no)?Exit();

                     Hydrogen(id1)!ExitOK(id1, id2, id3);
                     Hydrogen(id2)!ExitOK(id1, id2, id3);
                     Oxygen(id3)!ExitOK(id1, id2, id3)
                 ]
             ]
         STOP;
]

```

## 65 Дељени рачун

Рачун у банци може да дели више корисника (*The Savings Account Problem*). Сваки корисник може да уплаћује и подиже новац са рачуна под условом да салдо на рачуну никада не буде негативан, као и да види тренутно стање рачуна. Решити проблем користећи CSP.

### Решење

```
[User(i : 1 .. nu) :: USER || Bank :: BANK || Account(i : 1 .. na)
 :: ACCOUNT]

USER :: [
    ...
    Bank!getAccount(); Bank?accountID(id);
    ...
    Bank!removeAccount(id); Bank?ok();
    ...
    Account(id)!getStatus(); [
        Account(id)? status(value) → ...
        □ Account(id)? noAuthorization() → ...
    ]
    ...
    Account(id)!deposit(value); [
        Account(id)? ok() → ...
        □ Account(id)? noAuthorization() → ...
        □ Account(id)? wrongValue() → ...
    ]
    ...
    Account(id)!withdraw(value); [
        Account(id)? ok() → ...
        □ Account(id)? noAuthorization() → ...
        □ Account(id)? wrongValue() → ...
    ]
]

BANK :: [
    ...
    (i : 1 .. nu) User(i)?getAccount() → [
        GETACCOUNT;
        Account(id)!addUser(i);
        User(i)!accountID(id)
    ]
    ...
    (i : 1 .. nu) User(i)?removeAccount(id) → [
        Account(id)!removeUser(i);
        User(i)!ok()
    ]
]
```

```
ACCOUNT :: [
    saldo, value : double; saldo := 0;

    * [   Bank?addUser(j) → ADDUSER
        □ Bank?removeUser(j) → REMOVEUSER
        □ (i : 1 .. nu)User(i)?getStatus() → [
            SEARCH;
            [
                k < size → User(i)!status(saldo);
                □ k >= size → User(i)!noAuthorization()
            ]
        ]
        □(i : 1 .. nu)User(i)?deposit(value) → [
            SEARCH;
            [
                k < size, value >= 0 → [
                    saldo := saldo + value;
                    User(i)!ok();
                ]
                □ k >= size → User(i)!noAuthorization()
                □ value < 0 → User(i)!wrongValue()
            ]
        ]
    ]
    □
    (i : 1 .. nu)User(i)?withdraw(value) → [
        SEARCH;
        [
            k >= size → User(i)!noAuthorization()
            □ value < 0 → User(i)!wrongValue()
            □ value >= 0, k < size → [
                *[ value > saldo → [value1 : double;
                    (m : 1 .. nu)User(m)?deposit(value1) → [
                        SEARCH;
                        [
                            k < size, value1 >= 0 → [
                                saldo := saldo + value1;
                                User(m)!ok()
                            ]
                            □k>=size→ User(m)!noAuthorization()
                            □value1 < 0 → User(m)!wrongValue()
                        ]
                    ]
                ]
                saldo := saldo - value;
                User(i)!ok()
            ]
        ]
    ]
]
```

Треба приметити да у случају да неки процес чека да се скупи довољно новца на рачуну како би га подигао могуће је реализацивати и варијанту у којој би било дозвољено не само да се уплаћује новац на рачун, него би било и дозвољено гледање стања рачуна.

## Задаци за вежбу

- Постоји  $N$  процеса  $node$  ( $i:1..N$ ) који или чувају елементе листе или су слободни. Сваки од њих може да чува неку вредност ( $value$ ) и показивач на следећи елемент у листи ( $next$ ). Процес  $head$  чува показивач на први елемент у листи. Приликом брисања елемента из листе процес који га представља се пребацује на почетак листе слободних процеса на који указује процес  $free$ . Процес  $start$  иницира брисање елемента са задатом вредношћу из листе. Реализовати процедуре за све наведене процесе користећи  $CSP$ .
- Реализовати клијент-сервер апликацију у којој постоји 100 клијената  $K(i:1..100)$ , три сервера  $S1$ ,  $S2$  и  $S3$  и један помоћни процес  $P$ . Клијенти шаљу захтеве процесу  $P$  који их наизменично прослеђује једном, другом, па трећем серверу, а клијента обавештава о томе ком серверу је захтев прослеђен. Клијент прима одговор директно од сервера коме је прослеђен његов захтев.

## Јавно емитовање (BSP)

*Broadcasting Sequential Processes* (*BSP*) програмски модел омогућава комуникацију помоћу јавног емитовања (*broadcasting*). Код овог модела програми се састоје од више секвенцијалних процеса који међусобно комуницирају путем јавног емитовања, што значи да се порука која је емитована прима од стране свих процеса истовремено. Овакав начин комуникације је прилагођен тзв. *broadcast protocol multiprocessor* (*BPM*) системима, какве су рецимо *Ethernet* рачунарске мреже. *BSP* нуди и могућност слања порука у виду мултикастинга (*multicasting*), где поруку прима само одређени подскуп процеса. У крајњем случају, могуће је реализовати и класично асинхроно *point-to-point* слање (*unicasting*), када поруку прима само један процес. Предност јавног емитовања јесте да се по природи ствари не мора знати колико је процеса активно у систему, тако да тај број може да се мења динамички. То упростљава писање одређене класе програма. Осим примитива за асинхроно слање, *BSP* има и могућност синхроног *point-to-point* слања. Код свих типова преноса пријем је синхрон, тј. блокирајући.

Свака порука *m* која се шаље садржи следеће информације:

- *m.sender* – име процеса који је послao поруку *m*
- *m.info* – информациони део поруке *m*

Приликом слања поруке може се специфицирати да ли је пријем баферован или не. Небаферовано слање се обележава симболом  $\textcircled{O}$ , а баферовано слање симболом  $\textcircled{\times}$ . Разлика између ове две наредбе јесте у томе што ће у првом случају емитована порука бити изгубљена за све процесе који нису спремни да је прихвате када она стигне до њих, а у другом случају порука се памти у баферима процеса прималаца. Величина бафера се сматра неограниченом, иако у реалности то, наравно, није случај.

### Асинхрона комуникација

Примитиве за асинхроно слање порука су следеће (и означава информациони део поруке; ако он има сложену структуру, онда се поља унутар њега одвајају запетом, нпр. *<i1,...,iN>*):

- **all $\textcircled{O}$ i** - небаферовани *broadcast*. Порука се шаље свим процесима.
- **all $\textcircled{\times}$ i** - баферовани *broadcast*. Порука се шаље свим процесима.
- **[p1,...,pn] $\textcircled{O}$ i** - небаферовани *multicast*. Порука се шаље процесима *p1,...,pn*.
- **[p1,...,pn] $\textcircled{\times}$ i** - баферовани *multicast*. Порука се шаље процесима *p1,...,pn*.
- **p $\textcircled{O}$ i** - небаферовано *point-to-point* слање. Порука се шаље само процесу *p*.
- **p $\textcircled{\times}$ i** - баферовано *point-to-point* слање. Порука се шаље само процесу *p*.

Приликом слања се формира порука *msg*, тако да *msg.sender* садржи име процеса пошиљаоца, а *msg.info* садржи информациони део *i*. У случају немогућности слања, извршавање наредбе се одлаже све до тренутка док то не буде могуће.

Примитива за пријем порука је:

- **aluy $\textcircled{O}$ m**

Семантика наредбе зависи од типа преноса, тј. да ли је слање било баферовано или небаферовано. У случају небаферованог преноса, извршење наредбе се одлаже док

порука не стигне, након чега се она додељује променљивој  $m$ . У случају баферованог преноса, узима се прва порука из бафера примљених порука и додељује променљивој  $m$ . Ако нема порука у бафери, извршавање наредбе се одлаже док порука не буде примљена. Дакле, у оба случаја се врши синхрони (блокирајући) пријем.

Поруке које се примају се могу филтрирати према пошиљаоцу и садржају. На пример, наредба

**алу** (info=i, sender=[p1, ..., pn])  $\Theta m$

прима само поруке које имају информациону компоненту једнаку  $i$ , а послате су од стране једног од процеса  $p_1, \dots, p_n$ .

### Синхроне комуникације

Примитиве за класичну синхрону *point-to-point* комуникацију су следеће:

- $p!i$  – слање поруке са информационим садржајем  $i$  процесу  $p$ . Извршавање наредбе се одлаже док процес  $p$  не буде спреман да прими поруку.
- $p?m$  – пријем поруке послате од стране процеса  $p$ . Извршавање наредбе се одлаже док процес  $p$  не пошаље поруку и она не стигне примаоцу.
- **алу?** $m$  – пријем поруке послате од стране било ког процеса. Ако више процеса пошаље поруке истом примаоцу, поруке ће бити бафероване у *FIFO* бафери.

Код пријема се може применити исти механизам за филтрирање пристиглих порука као код асинхроног преноса.

У оба случаја, и код синхроне и код асинхроне комуникације, примитиве за пријем су синхроне и заправо су исте. Синтаксно су раздвојене због тога што је на тај начин у програму лакше препознати о каквом типу комуникације се ради.

### Контролне структуре

Од основних контролних структура *BSP* има *guarded-if* и *guarded-do* наредбе:

- *guarded-if* наредба има следећи облик:

```
if guard1 → statement list1
  □ guard2 → statement list2
  ...
  □ guardn → statement listn
fi
```

*guard*, тј. услов, представља неки логички израз који одређује када се може извршити одговарајући низ наредби (*statement list*). Одговарајући низ наредби се може извршавати само ако његов услов има вредност **true**. Услов **else** може се користити као логички израз који је истинит када ни један од осталих услова није истинит. Ако постоји више услова који су истинити, онда се између њих недетрминистички (тј. на случајан начин, односно на начин одређен имплементацијом) бира један чији ће низ наредби бити извршен. Бар један услов мора бити тачан, иначе се јавља грешка (ако постоји **else** онда ће бити испуњен бар тај услов). За регуларан излазак из *guarded-if* наредбе треба користити наредбу *skip*.

Проширена *guarded-if* наредба (*extended guarded-if*) има исти облик као обична *guarded-if* наредба, уз ту разлику да је сам услов проширен (*extended guard*), тако да може бити облика:

guard или guard; command или command

при чему *command* може бити *delay* наредба или наредба пријема. Наредбом *delay(t)* се одлаже извршавање за време не мање од *t*. Она се може користити у оквиру проширене *guarded-if* наредбе када се жели ограничити време чекања на пријем поруке (временски ограничен пријем).

Каже се да је услов *отворен* (*open*) ако је логички услов у њему тачан и одговарајућа наредба пријема се може одмах извршити, да је *затворен* (*closed*) ако је одговарајући логички услов нетачан, односно да чека *вредност* (*pending evaluation*) ако је логички услов тачан (ако га уопште има) а одговарајућа наредба пријема не може одмах да се изврши. Ако постоји више услова који чекају вредност, а нема отворених услова, онда се извршење *if* наредбе одлаже док се један од услова не отвори.

- *guarded-do* наредба има следећи облик:

```
do guard1 → statement list1
  □ guard2 → statement list2
  ...
  □ guardn → statement listn
od
```

*guarded-do* наредба извршава се док год је бар један услов тачан. Као и код *guarded-if* наредбе, извршава се само онај низ наредби чији услов има вредност тачно, а ако постоји више услова који имају вредност тачно, онда се између њих недетерминистички бира један чији ће низ наредби бити извршен. Аналогно са проширеном *guarded-if* наредбом постоји и проширена *guarded-do* наредба. Она се извршава све док сви услови не постану затворени. За излазак из *guarded-do* наредбе се може користити наредба *exit*.

**66**

Стек

Стек од  $n$  елемената имплементира се помоћу  $n$  процеса, од којих сваки складиши по један елемент. Могуће је додавати елементе на врх стека, брисати их са врха стека и испитивати садржај елемента на врху стека. Те операције процеси обављају када приме неку од одговарајућих порука са одговарајућим информационим делом облика  $\langle add, number \rangle$ ,  $\langle delete, null \rangle$  или  $\langle top, null \rangle$ , респективно. Реализовати процесе који представљају елементе стека.

## Решење

Претпоставка је да постоји један или више контролних процеса који примају захтеве од корисника и прослеђују их упоредо свим процесима који чувају елементе стека (овде није дат изглед таквог контролног процеса). Информациони део поруке коју прима процес који чува елемент стека има структуру облика  $\langle operation, value \rangle$ , тј. задаје операцију и вредност. Процес који чува  $i$ -ти елемент стека се може представити на следећи начин:

```

stack_size:= 0;
any@m
do true →
  if m.info.operation=add →
    if i=stack_size+1 → element:= m.info.value
    □ else → skip
    fi
    stack_size:= stack_size+1
  □ m.info.operation=top and i=stack_size → m.sender!element
  □ m.info.operation=delete → stack_size:= stack_size-1
  fi
  any@m
od

```

Ово решење је потпуно дистрибуирано, тако да евентуални губитак неких елемената не угрожава интегритет осталих елемената на стеку. Мана имплементације стека по принципу који је захтеван у задатку је што имплицира да у систему увек мора да постоји онолико процеса колико максимално може имати елемената скупа, чак и када скуп има мање елемената.

Кориснички процеси који шаљу захтеве са операцијама за рад на стеку би требало да користе баферовано спање, како поузданост рада не би зависила од брзине којом процеси примаоци могу да прихватају нове поруке. Треба приметити и да више корисничких процеса може истовремено да манипулише елементима стека.

67

Скуп

Скуп се имплементира тако што се сваком елементу скупа додељује један процес. Над скупом се могу обављати следеће операције: додавање елемената у скуп, брисање елемената из скупа и испитивање припадности неког елемената скупу, што се ради слањем порука са информационим делом облика  $\langle add, value \rangle$ ,  $\langle delete, value \rangle$  или  $\langle in, value \rangle$ , респективно. Све операције над скупом се морају извршавати преко процеса *SET* који интерагује са осталим процесима скупа. Реализовати процес *SET* и процесе који представљају елементе скупа.

## Решење

Процеси који представљају елементе скупа могу се реализовати на следећи начин:

```
free:= true; m:= null;
do true →
  any@m
    if m.info.operation=are_you_free and free → m.sender!yes
    □ m.info.operation=add and free →
      element:= m.info.value; free:= false
    □ m.info.operation=delete and not free →
      if m.info.value=element → free:= true; m.sender!deleted
      □ else → m.sender!cannot
      fi
    □ m.info.operation=in and not free →
      if m.info.value=element → m.sender!have_it
      □ else → m.sender!dont_have_it
      fi
    □ else → skip
  fi
od
```

Процес *SET* је једини који је задужен да манипулише елементима скупа. Да би се избегло да сваки процес скупа мора да прати колико елемената је тренутно у скупу, тај посао преузима процес *SET*. Он прима захтеве од клијентских процеса у виду порука, а потом их прослеђује процесима елемената скупа. Процес *SET* се може представити на следећи начин.

```
cur_size:= 0;
do true →
  any@r
    if r.info.operation=add →
      if cur_size<max_size →
        all@are_you_free {at least one must be free}
        {choose one process that will accept the new element}
        any(info=yes)?m → m.sender@m.info.operation,m.info.value>
        cur_size:= cur_size+1;
        i:= 0;
        {ignore the other volunteers}
        do i<max_size-cur_size; any(info=yes)?m →
          i:= i+1
      od
```

```

□ else → r.sender@set_full
fi
□ r.info.operation=delete →
all@<r.info.operation,r.info.value>
i:= 0; n:= cur_size;
do i<=n → {n responses expected}
    if any(info=deleted)?m → cur_size:= cur_size-1
    □ any(info=cannot)?m → skip
    fi
    i:= i+1
od
if cur_size<n → r.sender@deleted
□ else → r.sender@dont_have_it
fi
□ r.info.operation=in →
all@<r.info.operation,r.info.value>
i:= 0; n:= cur_size; have:= false;
do i<=n and not have → {n responses expected}
    if any(info=have_it)?m → have:= true
    □ any(info=dont_have_it)?m → skip
    fi
    i:= i+1
od
if have → r.sender@have_it
□ else → r.sender@dont_have_it
fi
□ else → skip
fi
od

```

Елемент се додаје у скуп тако што се најпре испитује да ли је скуп већ пун. Ако није, онда постоје слободни процеси и тражи се да се јави један од њих који би ускладишио нови елемент. Од свих процеса који се пријаве бира се први и њему се шаље елемент, а одговори осталих слободних процеса се игноришу.

Елемент се брише из скупа тако што се емитује (путем бродкаста (*broadcast*) са баферованим пријемом) одговарајућа порука свим процесима који представљају елементе скупа, а потом испитују одговори да би се проверило да ли је дати елемент обрисан.

Код испитивања припадности елемента скупу поступа се слично - шаље се одговарајућа порука свим процесима који представљају елементе скупа, а потом испитују одговори да би се проверило да ли је дати елемент у скупу.

Овако дефинисан процес *SET* има ману да скуп може имати више истих елемената, тј. да не представља прави математички скуп. Решење такође није ефикасно јер *SET* мора да прихвати сваки одговор који процеси који представљају елементе скупа пошаљу пре него што се прихвати следећи захтев. То је неопходно како се не би десило да се неправилно интерпретирају одговори на раније захтеве као одговори на захтеве који су стигли касније. Међутим, такав приступ није ефикасан јер је од свих њих значајан обично само један одговор. Овај проблем се може решити увођењем идентификатора захтева, тако да када процес *SET* шаље захтев процесима који представљају елементе скупа, у оквиру поруке заједно шаље и идентификатор захтева. Овај идентификатор процеси елемената после шаљу заједно са одговором процесу *SET*. На тај начин не може да дође до забуне код примања одговора. Да би

идентификатори захтева били јединствени, најједноставније је усвојити да сваки нови захтев добије број за један већи од претходног.

Поруке ће имати информациони део облика  $\langle operation, value \rangle$ ,  $\langle operation, request \rangle$  или  $\langle operation, value, request \rangle$ , у зависности од тога шта се тражи. Процеси који представљају елементе скупа имаће следећи изглед:

```
free:= true; m:= null;
do true -
    any@m
    if m.info.operation=are_you_free and free ->
        m.sender@<yes,m.info.request>
    □ m.info.operation=add and free ->
        element:= m.info.value; free:= false
    □ m.info.operation=delete and not free ->
        if m.info.value=element ->
            free:= true; m.sender@<deleted,m.info.request>
        □ else -> skip
        fi
    □ m.info.operation=in and not free ->
        if m.info.value=element ->
            m.sender@<have_it,m.info.request>
        □ else -> skip
        fi
    □ else -> skip
    fi
od
```

У наставку је дат изглед процеса  $SET$  који не дозвољава да скуп има више истих елемената. Као додатно унапређење, код одговора који шаљу процеси који представљају елементе скупа користи се асинхроно слање са баферованим пријемом, како не би морало да се чека да процес  $SET$  заврши пријем.

```
cur_size:= 0; req=0;
do true -
    any@r
    if r.info.operation=add ->
        if cur_size<max_size ->
            {is there such an element in the set already?}
            req:= req+1;
        all@<r.info.operation,r.info.value,req>
        i:= 0; n:= cur_size; have_already:= false;
        do i<=n and not have_already ->
            if any(info=<have_it,req>) ?m -> have_already:= true
            □ any(info=<dont_have_it,req>) ?m -> skip
            fi
            i:= i+1
        od
        if have_already -> skip
        □ else -> {add it}
        req:= req+1;
        all@<are_you_free,req>
        if any(info=<yes,req>) ?m ->
            m.sender@<m.info.operation,m.info.value>;
```

```
    cur_size:= cur_size+1
    □ else → skip {should never actually come here}
    fi
fi
□ else → r.sender⊗set_full
fi
□ r.info.operation=delete →
req:= req+1;
all⊗<r.info.operation,r.info.value,req>
i:= 0; n:= cur_size; del:= false;
do i<=n and not del →
    if any(info=<deleted,req>) ?m →
        cur_size:= cur_size-1; del:= true
    □ any(info=<cannot,req>) ?m → skip
    fi
    i:= i+1
od
if del → r.sender⊗deleted
□ else → r.sender⊗dont_have_it
fi
□ r.info.operation=in →
req:= req+1;
all⊗<r.info.operation,r.info.value,req>
i:= 0; n:= cur_size; have:= false;
do i<=n and not have →
    if any(info=<have_it,req>) ?m → have:= true
    □ any(info=<dont_have_it,req>) ?m → skip
    fi
    i:= i+1
od
if have → r.sender⊗have_it
□ else → r.sender⊗dont_have_it
fi
□ else → skip
fi
od
```

И ово решење је потпуно дистрибуирано.

## Задаци за вежбу

- Конкурентност решења из задатка 67 је ограничена због тога што постоји само један процес *SET* који може да прихвата и обрађује само једну поруку (наредбу) коју шаљу клијенти у једном тренутку. Написати решење које би омогутило већу конкурентност тако што би истовремено могло да се прихвата више порука, тј. наредби, послатих од стране више клијената. Водити рачуна о атомизацији приступа елементима скупа, како не би дошло до неконзистентности. Да ли се конкурентност може повећати код све три наредбе или не? Зашто?

## 68

## Филозофи за ручком

Решити проблем филозофа за ручком, описан у задатку 4.

### Решење

Друго решење из задатка 4 је најпримереније парадигми програмирања помоћу јавног емитовања због једноставног модела интеракције и због тога што су сви процеси филозофи исти чиме се омогућава да решење буде потпуно дистрибуирано. Виљушке које су биле представљене помоћу дељених променљивих, овде се у складу са филозофијом дистрибуираног програмирања представљају помоћу независних процеса којима се приступа слањем порука. Филозофи и виљушке су потпуно раздвојени, тако да се њихова структура може мењати без икаквог утицаја на функционалност програма. Процеси из класе филозофа, односно виљушака, се разликују између себе једино преко свог идентификатора у виду броја од 1 до  $N=5$ .

```
{philosopher i}
if (i mod 2)=1 →
    first:= i;
    second:= (i mod N)+1
□ else →
    first:= (i mod N)+1;
    second:= i
fi
do true →
    {wait for the first fork}
    all○<request_fork,first>
    do any(info=<take_fork,first>) ○m1 → exit
    □ delay(d) → all○<request_fork,first>
    od
    {wait for the second fork}
    all○<request_fork,second>
    do any(info=<take_fork,second>) ○m2 → exit
    □ delay(d) → all○<request_fork,second>
    od
    eat;
    {return the forks}
    m1.sender!<return_fork,first>
    m2.sender!<return_fork,second>
    think;
od

{fork i}
do any(info=<request_fork,i>) ○m →
    m.sender○<take_fork,i>
    m.sender(info=<return_fork,i>) ?m_ret
od
```

Треба приметити да се код враћања виљушака користи синхронна комуникација, јер би губљење поруке о враћању виљушке нарушило логику програма.

Једини проблем код овог решења јесте могућност да дође до изгладњивања филозофа, као и у оригиналном решењу са семафорима. Овај проблем је могуће избећи коришћењем преноса порука са баферованим пријемом уместо преноса без баферовања, јер то обезбеђује да захтев сваког филозофа за виљушком буде запамћен код процеса који представља одговарајућу виљушку и задовољен у тренутку када дође на ред. Ово решење је уједно и једноставније:

```

{philosopher i}
if (i mod 2)=1 →
    first:= i;
    second:= (i mod N)+1
else →
    first:= (i mod N)+1;
    second:= i
fi
do true →
    {wait for the first fork }
    all@<request_fork,first>
    any(info=<take_fork,first>) ?m1
    {wait for the second fork}
    all@<request_fork,second>
    any(info=<take_fork,second>) ?m2 → exit
    eat;
    {return the forks}
    m1.sender!<return_fork,first>
    m2.sender!<return_fork,second>
    think;
od

{fork i}
do any(info=<request_fork,i>) @m →
    m.sender!<take_fork,i>
    m.sender(info=<return_fork,i>) ?m_ret
od

```

**69****Читаоци и писци**

Помоћу BSP нотације решити проблем читалаца и писаца.

**Решење**

По угледу на решење проблема филозофа који ручавају и првог решења проблема читалаца и писаца датог у задатку 68 добија се следеће решење.

**Прво решење**

```

{buffer}
numR := 0;
numW := 0;
do numW = 0, any(info.request=startReading) 0m →
    numR := numR + 1;
    m.sender⊗<okReading, m.info.id>
    □ any(info.request=endReading) ?m → numR := numR - 1
    □ numW = 0, numR = 0, any(info.request=startWriting) 0m →
        numW := numW + 1;
        m.sender⊗<okWriting, m.info.id>
    □ any(info.request=endWriting) ?m → numW := numW - 1
od

{reader i}
do true →
    {startReading}
    allO<startReading, i>
    do any(info=<okReading, i>) 0m1 → exit
        □ delay(d) → allO<startReading, i>
    od

    read;
    {endReading}
    m1.sender!<endReading, i>
od

{writer i}
do true →
    {startWriting}
    allO<startWriting, i>
    do any(info=<okWriting, i>) 0m1 → exit
        □ delay(d) → allO<startWriting, i>
    od

    read;
    {endWriting}
    m1.sender!<endWriting, i>
od

```

Друго решење

```
{buffer}
numR := 0;
numW := 0;
do numW = 0, any(info.request=startReading) ?m →
    numR := numR + 1;
    m.sender!<okReading, m.info.id>
    □ any(info.request=endReading) ?m → numR := numR - 1
    □ numW = 0, numR = 0, any(info.request=startWriting) ?m →
        numW := numW + 1;
        m.sender!<okWriting, m.info.id>
    □ any(info.request=endWriting) ?m → numW := numW - 1
od

{reader i}
do true →
    {startReading}
    all@<startReading, i>
    any(info=<okReading,i>) ?m1

    read;
    {endReading}
    m1.sender!<endReading,i>
od

{writer i}
do true →
    {startWriting}
    all@<startWriting, i>
    any(info=<okWriting,i>) ?m1

    read;
    {endWriting}
    m1.sender!<endWriting,i>
od
```

## 70

## Радио такси

Помоћу BSP нотације прикажите рад радио-таксија. У решењу треба да постоје процеси корисника таксија, диспектера позива и самих таксија са радио станицама. Редослед при нормалној резервацији је следећи: корисник телефоном позове диспектера и скаже захтев, диспектер објави захтев на радио-каналу такси службе, јави се такси који ће да обави превоз и то тако да сви чују, диспектер потврђује ко обавља превоз и на крају такси долази до корисника.

### Решење

Телефонски разговор је по природи *point-to-point* комуникација. Приликом успостављања везе корисник ће најпре чекати да се диспектер јави а онда ће пренети поруку, док са друге стране диспектер чека да неко позове телефоном па онда прима поруку. Ово одговара синхроној комуникацији. Објављивање захтева за вожњу и одговарајуће адресе преко радио-канала се, сходно природи радио-везе, моделира асинхроним *broadcast*-ом. То исто важи и за одговор таксија диспектеру.

```
{customer}
dispatcher!<taxi_needed, address>
any(info=<taxi_arrived, address>) @taxi_arrived
{dispatcher}
do any?m →
  if m.info.request=taxi_needed →
    all@<taxi_needed, m.info.address>
    do any(info=<taxi_free, m.info.address>) @m_taxi →
      all@<m_taxi.sender, m.info.address>
      □ delay(d) → all@<taxi_needed, m.info.address>
      od
    □ else → skip
  fi
od

{taxi i}
do any(sender=dispatcher, info.request=taxi_needed) @m →
  if (m.info.address is close to my position) →
    all@<taxi_free, m.info.address>
    any(sender=dispatcher, info.address=m.info.address) @m2
    if m2.info.id=i → {OK, my drive...}
      all@<taxi_arrived, m.info.address>
    □ else → skip
  fi
  □ else → skip
fi
od
```

Приликом објављивања захтева, диспектер шаље поруку која садржи и тип захтева (*taxi\_needed*) и адресу, јер захтеви не морају бити увек везани за кориснике који траже такси (рецимо, захтев може бити тражење неке информације или помоћи од стране другог колеге преко диспектера), док различити захтеви могу бити везани за различите адресе. На овај начин се избегава забуна. Такође, када такси шаље одговор

диспачеру, он из истих разлога шаље поруку са обе информације – тип одговора (*taxi\_free*) и адресу на коју одговара.

Када диспачер прими одговоре од таксија, он јавно потврђује ко је од њих добио вожњу. Ако ни један такси није одговорио, диспачер после неког времена понавља захтев, све док се неко не јави. Таксисти са друге стране када чују захтев од диспачера, најпре процене да ли је адреса близу њиховој позицији и ако јесте шаљу потврду. Након тога чекају одговор од диспачера за ту адресу (јер има и других захтева за друге адресе) и онај такси за кога је диспачер потврдио да је добио вожњу одлази код корисника на задату адресу.

## Задаци за вежбу

- ➔ Решити проблем радио таксија под следећим условима: Када таксисти чују захтев за вожњу од диспачера и установе да им је позиција близу дате адресе, у оквиру потврде морају да пошалу и своју процењену удаљеност од те адресе. Диспачер, са друге стране, када чује одговоре таксиста на објављени захтев, процењује који је од њих најближи и објављује коме је вожња додељена.

## 71

## Гониометарски систем

Помоћу BSP нотације прикажите рад гониометарског система. У решењу треба да постоје процеси централног рачунара и три гониометарске станице. Централни рачунар шаље гониометарским станицама фреквенцију за коју треба да ураде гониометрисање. Свака станица, по пријему податка о фреквенцији, а после фиксног временског периода (који је специфициран за сваку станицу понаособ) шаље одговор о углу у односу на север посматраног објекта. Централни рачунар на основу примљених углова одређује координате објекта. Ако се током неког временског интервала не јаве све три гониометарске станице, централни рачунар поново шаље исту фреквенцију. Ако се опет после неког времена не јаве бар две станице, централни рачунар сигнализира грешку.

## Решење

Комуникација је у овом случају искључиво радио-везом, што значи да се поруке шаљу асинхроним бродкастом (*broadcast*).

```

{goniometric station i}
do any(sender=central_computer) @m →
    angle=f(m.info.freq); {calculate the angle based on frequency}
    delay(di);
    central_computerOangle
od

{central computer}
do true →
    allOfreq
    i:= 0;
    do i = 0; any@m → angle1:= m.info; i:= i+1
    □ i = 1; any@m → angle2:= m.info; i:= i+1
    □ i = 2; any@m → angle3:= m.info; i:= i+1
    □ i=3 → position=f(angle1,angle2,angle3); exit
    □ delay(t1) →
        i:= 0;
        allOfreq
        do i = 0; any@m → angle1:= m.info; i:= i+1
        □ i = 1; any@m → angle2:= m.info; i:= i+1
        □ i = 2; any@m → angle3:= m.info; i:= i+1
        □ i = 2, delay (t1) → position=f1(two available angles)
        □ i=3 → position=f(angle1,angle2,angle3); exit
        □ delay(t2) → {error}
    od
od
od

```

## Задаци за вежбу

→ На *Ethernet* локалној мрежи налази се  $N$  рачунара од којих сваки има свој диск. Потребно је обезбедити да сваки рачунар може да затражи неки диск простор, тако што би добио расположиве делове диск простора од других рачунара на локалној мрежи. Након добијања информације од осталих рачунара о расположивом простору, потребно је послати информацију о резервацији свим рачунарима од којих се узима простор. На крају они јављају потврду о резервацији. Користећи *BSP* нотацију, напишите костур таквог програма тако да важи *FIFO* принцип за секвенцу емитовања захтева – резервација простора. Узети у обзир и случајеве када нема довољно простора на диску на осталим рачунарима.

## Асинхроно прослеђивање порука (CONIC)

CONIC је програмски језик који обезбеђује асинхрону комуникацију између процеса са индиректним именовањем. Осим тога, он има могућност динамичке промене конфигурације комуникационих канала током извршавања програма, што омогућава смањену осетљивост на отказе (ово је омогућено преко посебног конфигурационог језика). Синтакса језика CONIC је слична синтакси језика Pascal, а уведен су додатни синтакски елементи који се користе за декларисања портова, реализацију алтернативне команде и за слање и пријем порука.

### Декларација процеса

Процес (модул) се у синтакси језика CONIC декларише на следећи начин:

```
task module name
декларација улазних и излазних портова
декларација локалних променљивих
begin
    ...
end.
```

### Комуникационе примитиве

CONIC подржава асинхрону комуникацију прослеђивањем порука уз индиректно именовање процеса помоћу тзв. *портова* (*ports*). Портови су улазно-излазне тачке процеса преко којих они комуницирају са другим процесима. Могу бити улазни и излазни, као и унидирекциони и бидирекциони. Декларишу се на следећи начин:

- улазни унидирекциони порт: `entryport IN1:itype`
- улазни бидирекциони порт: `entryport IN2:itype reply ritype`
- излазни унидирекциони порт: `exitport OUT1:otype`
- излазни бидирекциони порт: `exitport OUT2:otype reply rotype`

Слање порука је асинхроно, док је пријем синхрон, односно блокирајући. Постоје два типа примитива за слање и пријем порука. Први, тзв. *notify* тип служи за унидирекциону комуникацију преко једносмерног канала:

- слање поруке на излазни унидирекциони порт:  
`send msg to OUT1`
- пријем поруке са улазног унидирекционог порта:  
`receive msg from IN1`

Други, тзв. *request-reply* тип служи за бидирекциону комуникацију преко двосмерног канала:

- слање поруке на излазни бидирекциони порт:

```
send msg to OUT2
    wait rep => ...
    timeout t => ...
    fail => ...
end
```

Након слања поруке на бидирекциони излазни порт *OUT2*, чека се на одговор у оквиру алтернативе *wait*. Ако одговор не стигне у оквиру времена *t*, бира се алтернатива *timeout*. Ако дође до грешке или до прекида комуникационог канала бира се алтернатива *fail*. Једноставнија варијанта ове наредбе је:

```
send msg to OUT2 wait rep
▪ пријем поруке са улазног бидирекционог порта:
receive msg from IN2 =>
    ...
    reply rep
    ...
end
```

Након пријема поруке преко бидирекционог улазног порта *IN2*, шаље се одговор пошиљаоцу преко двосмерног канала (нема потребе да се именује процес пошиљалац). Постоји и једноставнија варијанта наредбе где се одмах одговара пошиљаоцу:

```
receive msg from IN2 reply rep =>
    ...
end
```

#### Алтернативна команда

Најзначајнија контролна структура јесте алтернативна команда:

```
select
    when cond1; receive m1 from in1 => ...
    or
    ...
    or
    timeout t => ...
    else
    ...
end select
```

У језику CONIC је алтернативна команда детерминистичка; код избора услова се креће од првог и иде се редом до последњег, а бира се прва алтернатива чији је *when* услов задовољен. Ако порука не стигне за време *t*, бира се алтернатива *timeout*. Ако ни један услов није задовољен, бира се алтернатива *else*.

## Конфигурациони језик

CONIC је специфичан по томе што подржава динамичко повезивање процеса помоћу специјалног конфигурационог језика, тако да се топологија веза између процеса, тј. конфигурација комуникационих канала, може мењати у току извршавања.

- инсталација типова процеса који се користе у програму: `use type1,...,typeN;`
- уклањање типова процеса који се користе у програму: `remove type1,...,typeN;`
- креирање инстанци процеса (модула): `create mod1,...,modN:type;`
- брисање инстанци процеса (модула): `delete mod1,...,modN;`
- креирање веза између излазних и улазних портова процеса (модула):  
`link mod1.exitport,...,modN.exitport to mod.entryport;`  
`link mod.exitport to mod1.entryport,...,modN.entryport;`
- уклањање веза између излазних и улазних портова процеса (модула):  
`unlink mod1.exitport,...,modN.exitport from mod.entryport;`  
`unlink mod.exitport from mod1.entryport,...,modN.entryport;`
- рад са групама (фамилијама) код креирања, брисања, креирања и уклањања веза:  
`create family index mod[index]:type;`  
`delete family index mod[index];`  
`link family index mod.exitport[index] to mod.entryport[index];`  
`unlink family index mod.exitport[index] from mod.entryport[index];`
- Декларација конфигурисаног процеса:  
`group module name`  
конфигурациони скрипт;

72

## Бафер разделник

Написати процес који приhvата улазну поруку и шаље њене копије на сваки од два излазна порта. Комуникација треба да буде синхронна.

### Решење

Модул има један улазни и два излазна порта и сви су бидирекциони:

```
task module splitter;
entryport input: String reply signaltype;
exitport output1: String reply signaltype;
          output2: String reply signaltype;
const waitperiod = 150; {3 seconds}
var buffer: String;
begin
  loop
    receive buffer from input reply signal =>
      send buffer to output1
      wait signal
      timeout waitperiod => {do nothing}
    end;
    send buffer to output2
    wait signal
    timeout waitperiod => {do nothing}
  end
end
end.
```

73

## Произвођачи и потрошачи: комуникација помоћу кружног бафера

Написати процес који представља кружни бафер коначног капацитета.

### Решење

```
task module bound;
    entrystore put_char : char reply signaltypes;
        get_char : signaltypes reply char;
    const max_size = 100;
    var inp,
        outp,
        contents : natural;
    buffer : array [1..max_size] of char;
begin
    inp := 1;
    outp := 1;
    contents := 0;
loop
    select
        when (contents < max_size) {buffer not full}
            receive buffer[inp] from put_char reply signal
            =>      inp := (inp mod max_size) + 1;
                    contents := contents + 1;
            end;
        or
        when (contents > 0) {buffer not empty}
            receive signal from get_char reply buffer[outp]
            =>      outp := (outp mod max_size) + 1;
                    contents := contents - 1;
            end;
    end;
end {loop};
end {task}.
```

## 74 Промена конфигурације: звезда у прстен

Дефинисана су два типа модула:

```
task module central;
entryport in1:msg;
    in2:msg;
    in3:msg;
exitport out1:msg;
    out2:msg;
    out3:msg;
begin
    ...
end.
```

```
task module terminal;
entryport in:msg;
exitport out:msg;
begin
    ...
end.
```

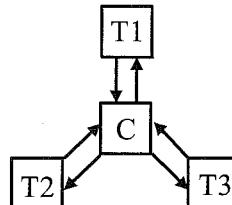
Модули наведених типова употребљени су за симулацију чворова рачунарске мреже. Коришћењем команди конфигурационог језика:

- Креирати један чвр *C* типа *central* и три чвора *T1*, *T2* и *T3* типа *terminal* и повезати их у звездасту мрежу са чвром *C* у центру.
- Специфицирати промену конфигурације која настаје у мрежи под а) када се централни чвр уклони, а терминални чврови повежу у прстенасте топологије.

### Решење

a)

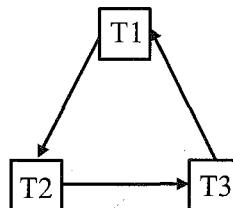
```
user      central, terminal;
create C: central;
        T1, T2, T3: terminal;
link     T1.out   to C.in1;
        T2.out   to C.in2;
        T3.out   to C.in3;
        C.out1  to T1.in;
        C.out2  to T2.in;
        C.out3  to T3.in;
```



Слика 15. Конфигурација „звезда”

b)

```
unlink T1.out from C.in1;
        T2.out from C.in2;
        T3.out from C.in3;
        C.out1 from T1.in;
        C.out2 from T2.in;
        C.out3 from T3.in;
link     T1.out to T2.in;
        T2.out to T3.in;
        T3.out to T1.in;
delete  C;
```



Слика 16. Конфигурација „прстен”

## 75

## Бидирекциони прстен типа FDDI

Дефинисан је тип модула:

```
task module node;
entryport in1:msg;
    in2:msg;
exitport out1:msg;
    out2:msg;
begin
    ...
end.
```

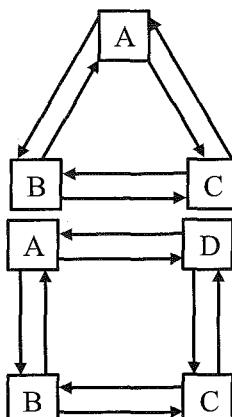
Модули овог типа употребљени су за симулацију чворова локалне мреже која има топологију бидирекционог прстена као што је нпр. *FDDI* (*Fiber distributed data interface*). Коришћењем команда конфигурационог језика:

- Креирати три чвора *A*, *B* и *C* и повезати их у бидирекциону прстенасту мрежу.
- Специфицирати промену конфигурације под а) насталу креирањем новог чвора *D* и његовим укључењем у бидирекциони прстен између *A* и *C*.
- Специфицирати промену конфигурације која настаје у мрежи под б) приликом отказа чвора *B* (приказати реконфигурацију повратним превезивањем на чворовима суседним чвору *B*, као код *FDDI*, а затим уклонити чвор *B*).
- Специфицирати промену конфигурације којом ће мрежа под в) постати бидирекциони прстен са чворовима *A*, *C* и *D*.

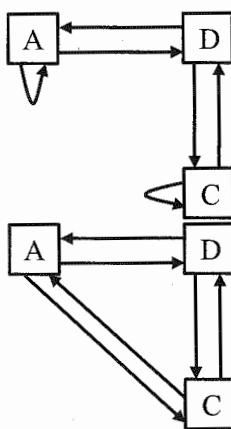
## Решење

```
a) create A, B, C: node;
link A.out1 to B.in1;
    B.out1 to C.in1;
    C.out1 to A.in1;
    A.out2 to C.in2;
    C.out2 to B.in2;
    B.out2 to A.in2;

b) create D: node;
unlink C.out1 from A.in1;
    A.out2 from C.in2;
link C.out1 to D.in1;
    D.out1 to A.in1;
    A.out2 to D.in2;
    D.out2 to C.in2;
```



- в) `unlink A.out1 from B.in1;  
C.out2 from B.in2;  
B.out1 from C.in1;  
B.out2 from A.in2;  
link A.out1 to A.in2;  
C.out2 to C.in1;  
delete B;`
- г) `unlink A.out1 from A.in2;  
C.out2 from C.in1;  
link A.out1 to C.in1;  
C.out2 to A.in2;`

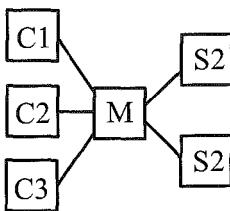


## 76

## Клијенти и сервери

У једној дистрибуираној апликацији три истоветна клијента  $C_1$ ,  $C_2$  и  $C_3$  користе услуге два истоветна сервера  $S_1$  и  $S_2$ . За прослеђивање захтева серверима и враћање резултата клијентима користи се медијатор  $M$  (види слику). Клијенти чекају на резултат пре него што наставе са радом. У случају да су оба сервера заузета, клијенту се враћа специјални резултат *null*.

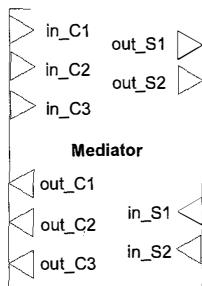
- Реализовати медијатор  $M$ . Обезбедити да не дође до изгладњивања клијената.
- Декларисати тип модула клијента и сервера.
- Составити код за креирање конфигурације са слике.



Слика 17. Конфигурација система

## Решење

a)



Код решења треба првенствено водити рачуна о конкурентности рада процеса. Због начина на који се обавља бидирекциона комуникација (види синтаксу у уводном делу), за сваки захтев који прими од клијента медијатор би морао најпре да чека да стигне одговарајући одговор, па тек потом да прими следећи захтев. Такав приступ има низак ниво конкурентности. Због тога се уместо бидирекционих користе унидирекциони канали. Медијатор у том случају може примити и проследити више захтева преко различитих канала пре него што стигне одговор на неки од њих.

Друго питање о коме треба водити рачуна јесте питање праведности, тј. да не дође до изгладњивања. Пошто је алтернативна команда код језика CONIC детерминистичка,

мора се увести посебан механизам који ће довести до изједначавања приоритета алтернатива како не би дошло до изгладњивања клијената. У том циљу користићемо *round robin* шему за клијенте како би њихови захтеви били једнаког приоритета, али и за сервере како се не би десило да неки сервер има нижи приоритет, па да не може да пошаље назад одговор клијенту који би тако могао да изгладни. Због веће ефикасности решења, пријем одговора од сервера генерално има већи приоритет од прихватања захтева од клијената. На тај начин се омогућава да клијенти добијају одговор у најкраћем могућем року, а сервери се брже ослобађају за пријем и обраду нових захтева.

```

task module Mediator;
  entryport in_C1, in_C2, in_C3: Request;
    in_S1, in_S2: Reply;
  exitport out_C1, out_C2, out_C3: Reply;
    out_S1, out_S2: Request;
  type clients = (C1, C2, C3);
  var req: Request;
    rep: Reply;
    freeS1, freeS2: boolean;
    looping: boolean;
    mapS1, mapS2: clients;
    priorS,priorC: integer;
  begin
    freeS1:= true;
    freeS2:= true;
    priorS:= 1;
    priorC:= 1;
    looping:= false;
  loop
    select
      when 1=priorS or looping; receive rep from in_S1 =>
        case mapS1 of
          C1: send rep to out_C1
          C2: send rep to out_C2
          C3: send rep to out_C3
        end;
        freeS1:= true;
        priorS:= priorS+1 mod 2;
        looping:= false;
      or
        when 2>=priorS or looping; receive rep from in_S2 =>
          case mapS2 of
            C1: send rep to out_C1
            C2: send rep to out_C2
            C3: send rep to out_C3
          end;
          freeS2:= true;
          priorS:= priorS+1 mod 2;
          looping:= false;
        or
          when 1>=priorC or looping; receive req from in_C1 =>
            if freeS1 then begin
              mapS1:= C1;
              send req to out_S1;
            end
          end
        end
      end
    end
  end

```

```

        freeS1:= false;
    end
    else
        if freeS2 then begin
            mapS2:= C1;
            send req to out_S2
            freeS2:= false;
        else begin
            rep:= null;
            send rep to out_C1
        end;
        priorC:= priorC+1 mod 3;
        looping:= false;
    or
        when 2>=priorC or looping; receive req from in_C2 =>
        if freeS1 then begin
            mapS1:= C2;
            send req to out_S1;
            freeS1:= false;
        end
        else
            if freeS2 then begin
                mapS2:= C2;
                send req to out_S2
                freeS2:= false;
            else begin
                rep:= null;
                send rep to out_C2
            end;
            priorC:= priorC+1 mod 3;
            looping:= false;
        or
        when 3>=priorC or looping; receive req from in_C3 =>
        if freeS1 then begin
            mapS1:= C3;
            send req to out_S1;
            freeS1:= false;
        end
        else
            if freeS2 then begin
                mapS2:= C3;
                send req to out_S2;
                freeS2:= false;
            else begin
                rep:= null;
                send rep to out_C3;
            end;
            priorC:= priorC+1 mod 3;
            looping:= false;
        else
            looping:= true;
        end select;
    end loop
end;

```

Алгоритам рада се може формулисати на следећи начин: проверавати да ли је стигла порука, почевши од алтернативе са највишим приоритетом па до последње у *select* наредби; ако је порука стигла, извршити одговарајућу алтернативу; ако порука није стигла, наставити са проверавањем свих алтернатива све док не стигне нека порука и онда извршити ту алтернативу.

*Round robin* принцип се реализује увођењем променљиве *priorC* која одређује који клијент тренутно има највиши приоритет. Услов *i>priorC* одређује да ће прва алтернатива за коју ће услов потенцијално мали да буде задовољен бити она код које је *i=priorC* (тј. она која прима поруку од клијента *C*). Ако је порука од клијента *i* заиста стигла, она се прослеђује првом слободном серверу, а информација о томе од ког клијента је послат захтев серверу се чува у променљивама *mapS1*, односно *mapS2*. На крају алтернативе се *priorC* помера тако да након тога клијент највишег приоритета буде следећи по реду у односу на оног који је претходно имао највиши приоритет, чиме се реализује *round robin* принцип. Ако порука од клијента *i* није стигла, онда се, сходно дефиницији *select* наредбе, наставља са испитивањем услова следеће алтернативе и тако све док се не пронађе прва алтернатива где је порука од клијента стигла. У случају да се утврди да у оквиру проверених алтернатива, укључујући и последњу у *select* наредби, ниједна порука није стигла, треба обезбедити да се у наредним пролазима проверавају све алтернативе (а не само до *i*-те до последње), све док не стигне нека порука. Тако се постиже максимална конкурентност.

*Round robin* принцип је примењен и на сервере. Међутим, пошто пријем одговора од сервера генерално има виши приоритет од пријема захтева клијената, алтернативе са командама за пријем порука од сервера су испред алтернатива са командама за пријем захтева од клијената у оквиру *select* наредбе.

б)



```

task module Client;
entryport in: Reply;
exitport out: Request;
var req: Request;
var rep: Reply;
begin
  ...
  send req to out;
  receive rep from in;
  ...
end

task module Server;
entryport in: Request;
exitport out: Reply;
var req: Request;
var rep: Reply;
begin
  ...
  receive req from in;
  send rep to out;
  ...
end
  
```

```

b) user Client, Server, Mediator;
create C1, C2, C3: Client;
      S1, S2: Server;
      M: Mediator;
link C1.out to M.in_C1;
      C2.out to M.in_C2;
      C3.out to M.in_C3;
      M.out_C1 to C1.in;
  
```

*Дистрибуирано програмирање: Асинхроно прослеђивање порука – CONIC*

```
M.out_C2 to C2.in;  
M.out_C3 to C3.in;  
M.out_S1 to S1.in;  
M.out_S2 to S2.in;  
S1.out to M.in_S1;  
S2.out to M.in_S2;
```

77

## Обрада података

Написати програм на језику CONIC који приhvата улазни податак за обраду, обраћује га и шаље одговор на излазни порт. Поред овога програм треба да враћа и број захтева који се тренутно обраћују. Од формираног модула направити нов модул који садржи *n* модула за обраду података, један модул за приhvатање и прослеђивање захтева.

## Решење

```

task module Machine;
  entryport Orders, Finished: Order;
    Status: Inquiry reply integer;
  exitport
    Confirmation, Start: Order;
  var number_of_orders: integer;
    production_order : Order;
    working : boolean;
  begin
    number_of_orders := 0;
    working := false;
    loop
      select
        when (not working)
          receive production_order from Orders =>
            number_of_orders := number_of_orders + 1;
            send production_order to Start;
            working := true;
        or
          when (working)
            receive production_order from Finished =>
              number_of_orders := number_of_orders - 1;
              send production_order to Confirmation;
              working := false;
          or receive Inquiry from Status
            reply number_of_orders;
      end;
    end;
  end.

task module Worker();
  entryport Start: Order;
  exitport Finished: Order;

  var production_order: Order;
begin
  loop
    receive production_order from Start =>
      work();
      send production_order to Finished;
  end;
end.

```

```
task module Distributor(n: integer := 2);
    entryport In: Order;
    exitport Out[1..n]: Order;
    var number_of_orders: integer;
    var production_order: Order;
begin
    number_of_orders := 1;
    loop
        receive production_order from In =>
            number_of_orders := (number_of_orders) mod n + 1;
            send production_order to Out[number_of_orders];
    end;
end.

group module Manufacturing (n: integer := 2);
    entryport
        Orders: Order;
        Status [1..n]: Inquiry reply integer;
    exitport
        Confirmation: Order;
    use
        Distributor, Machine, Worker;
    create
        Distributor at Knot_X;
    create family k:[1..n]
        Machine [k]: Machine at Knot[k];
        Worker[k]: Worker at W[k];
    link
        Orders to Distributor.In
    link family k:[1..n]
        Distributor.Out[k] to Machine[k].Orders;
        Status[k] to Machine[k].Status;
        Machine[k].Confirmation to Confirmation;
        Machine[k].Start to Worker[k].Start;
        Worker [k].Finished to Machine[k].Finished;
end.
```

## 78

## Филозофи за ручком

Написати програм на језику CONIC који решава проблем филозофа који ручавају.

### Решење

```
define type forktype = (pickup, putdown);

task module Philosopher;
    exitport rightfork : forktype reply signaltypes;
        leftfork : forktype reply signaltypes;
        exitTable : signaltypes reply signaltypes;
        enterTable : signaltypes reply signaltypes;
    var request, ok : signaltypes;
        pickupreq, putdownreq : forktype;

begin
    request := signal;
    pickupreq := pickup; putdownreq := putdown;
    loop
        {thinking}
        thinking ();
        send request to enterTable wait ok;
        {sitting}
        send pickupreq to leftfork wait ok;
        send pickupreq to rightfork wait ok;
        {eating}
        eating ();
        send putdownreq to leftfork wait ok;
        send putdownreq to rightfork wait ok;
        send request to exitTable wait ok;
    end;
end.

task module Fork;
    entryport rightphil : forktype reply signaltypes;
        leftphil : forktype reply signaltypes;

    type allocationotype = (none, left, right);
    var allocated : allocationotype;
        request : forktype;
    begin
        allocated := none;
        loop
            select
                when ((allocated=none) or (allocated=left))
                    receive request from leftphil reply signal
                        => case request of
                            pickup : allocated := left;
                            putdown : allocated := none;
            end;
end;
```

```
        or
when ((allocated=none) or (allocated=right))
    receive request from rightphil reply signal
        => case request of
            pickup : allocated := right;
            putdown : allocated := none;
        end;
    end {select};
end {loop};
end.

task module Table (n : integer); {number of philosophers}

entryport leave : signaltypes reply signaltypes;
sit : signaltypes reply signaltypes;
var sitting : integer;
request, ok : signaltypes;
begin
    sitting := 0;
    ok := signal;
loop
    select
        when sitting > 0
            receive request from leave reply ok
                => sitting := sitting - 1;
        or
        when sitting < (n-1)
            receive request from sit reply ok
                => sitting := sitting + 1;
    end {select};
end {loop};
end.

group module DiningPhilosopherTest(n : integer);
use Philosopher; Fork; Table; {dining philosopher modules}
create Table : Table(n = n);
create family k : [0..n-1]
    Philosopher[k] : Philosopher;
    Fork[k] : Fork;
link family k : [0..n-1]
    Philosopher[k].enterTable to Table.sit;
    Philosopher[k].exitTable to Table.leave;
    Philosopher[k].rightfork to Fork[k].leftphil;
    Philosopher[(k+1) mod n].leftfork to Fork[k].rightphil;
end.
```

79

## Читаоци и писци

Написати програм на језику CONIC који решава проблем читалаца и писаца.

### Решење

```
task module ReadersWriters;
entryport startread : signaltype reply signaltype;
            endread : signaltype reply signaltype;
            startwrite : signaltype reply signaltype;
            endwrite : signaltype reply signaltype;
varnumR : integer := 0;
numW : integer := 0;
begin
    loop
        select
            when (numW = 0);
                receive signal from startread reply signal =>
                    numR := numR + 1;
                end;
            or
                when (numR > 0);
                    receive signal from endread reply signal =>
                        numR := numR - 1;
                    end;
            or
                when ((numW = 0) and (numR = 0));
                    receive signal from startwrite reply signal =>
                        numW := numW + 1;
                    end;
            or
                when (numW > 0);
                    receive signal from endwrite reply signal =>
                        numW := numW - 1;
                    end;
        end;
    end;
end.

task module Reader (id : integer);
exitport startread : signaltype reply signaltype;
            endread : signaltype reply signaltype;
var request : signaltype;
begin
    request := signal;
    loop
        send request to startread wait request;
        {reading}
        send request to endread wait request;
    end;
end.
```

```
task module Writer (id : integer);
    exitport startwrite : signaltypes reply signaltypes;
        endwrite : signaltypes reply signaltypes;
var request : signaltypes;
begin
    request := signal;
loop
    send request to startwrite wait request;
    {writing}
    send request to endwrite wait request;
end;
end.

group module ReadersWritersTest(numR, numW : integer);
use ReadersWriters; Reader; Writer;
create rw : ReadersWriters;
create family k : [0..numR-1]
    Reader[k] : Reader(k);
create family k : [0..numW-1]
    Writer[k] : Writer(k);

link family k : [0..numR-1]
    Reader[k].startread to ReadersWriters.startread;
    Reader[k].endread to ReadersWriters.endread;
link family k : [0..numW-1]
    Writer[k].startwrite to ReadersWriters.startwrite;
    Writer[k].endwrite to ReadersWriters.endwrite;
end.
```

## Рандеву (Ada)

*Point-to-point* комуникација слањем порука по природи је једносмерна. Међутим, у великом броју случајева постоји потреба да интеракција између процеса код дистрибуираних програма буде двосмерна. Такав је случај код клијент-сервер интеракције где клијент од сервера захтева услугу, а потом чека да сервер врати резултат. Овај модел комуникације може се реализовати помоћу слања две *point-to-point* поруке по принципу *request-reply*, али је коришћење јединствене језичке конструкције за такву двосмерну комуникацију ефикасније и семантички јасније. Механизам рандеву представља реализацију таквог модела.

### Структура програма

Код програмског језика Ada програм се састоји од потпрограма (функција или процедура), таскова (у језику Ada, *task* /што значи посао, задатак/ представља назив за независан процес, при чему се процеси могу конкурентно извршавати) и пакета (*packages*). Пакет представља скуп логички повезаних ентитета и који може да садржи декларација променљивих, потпрограма и процеса. Свака од ових компонената има спецификациони део и тело. У спецификацији се декларишу дељени/видљиви објекти, односно интерфејс пакета, а тело садржи локалне декларације и наредбе, односно имплементацију пакета. Заштићени типови су основна јединица за дефинисање заштићених операција приликом координираног приступа дељеним подацима између таскова. Потпрограми и пакети могу бити и генерички, тј. могу бити параметризовани типовима података.

Процедуре се декларишу на следећи начин:

```
procedure <Procedure> (листа формалних параметара) is
    декларације
begin
    листа наредби
end <Procedure>;
```

Функције се декларишу слично као процедуре:

```
function <Function> (листа формалних параметара) return <Type> is
    декларације
begin
    листа наредби
end <Function>;
```

Улазни, излазни и улазно-излазни параметри се декларишу на следећи начин:

```
<Parameter>: [in|out|inout] <Type>
```

Пакети се декларишу на следећи начин:

```
package <Package> is
    јавне декларације
end <Package>;
package body <Package> is
    локалне декларације
    дефиниције јавно декларисаних потпрограма и таскова
end <Package>;
```

Таск се дефинише на следећи начин:

```
task <Task> is
    декларације тачака улаза
end;

task body <Task> is
    локалне декларације
begin
    листа наредби
end <Task>;
```

## Рандеву

Рандеву се у програму реализује преко три кључне тачке: декларације тачке улаза (*entry declaration*), тачке позива (*entry call*) и тачке прихватања позива (*accept statement*). Декларације тачака улаза и тачака прихватања позива чине део серверског процеса, а тачке позива се налазе на страни клијентских процеса.

Декларације тачака улаза дефинишу операције које дати процес сервисира. Синтакса декларације тачке улаза слична је као декларација процедуре:

```
entry <Entry> (листа формалних параметара)
```

Тачка позива изгледа слично као позив процедуре: именује се процес, тачка улаза и прослеђују стварни параметри:

```
<Task>.<Entry> (листа стварних параметара)
```

Тачка прихватања позива је место у коду серверског процеса где се чека на позив од стране клијентског процеса (преко специфициране тачке улаза):

```
accept <Entry> do
    листа наредби
end;
```

У оквиру серверског процеса може постојати више места на којима се прихвата позив преко исте тачка улаза, при чему се сваки пут могу дефинисати различите наредбе у оквиру тела *accept* наредбе. У оквиру клијентског процеса може постојати више тачака позива истог сервера коришћењем исте улазне тачке.

Извршавање *accept* наредбе блокира извршавање серверског процеса све док клијентски процес не изврши позив преко одговарајуће тачке улаза. Када се то деси, најпре се стварни параметри копирају у формалне параметре, а потом се извршава листа наредби унутар *accept*. Када се *accept* наредба заврши, излазне вредности формалних параметара се копирају у одговарајуће стварне параметре (наведене у оквиру *entry* декларације), а потом процес позивалац и позвани процес даље независно настављају са радом.

У случају да клијентски процес стигне до тачке позива пре него што серверски процес стигне до тачке прихватања, блокира се извршавање клијентског процеса. Од тренутка када серверски процес стигне до тачке прихватања, след дешавања је исти као и у претходном случају.

Као што се види, први процес који стигне до тачке синхронизације (рандеву) се блокира и сачекује онај други, а по завршетку обраде упоредо и независно настављају рад. Дакле, интеракција између процеса је потпуно синхронизована.

## Заштићени типови

Заштићени типови представљају језичку конструкцију која је најсличнија концепту монитора. Истанце заштићених типова на исти начин као и монитори енкапсулирају

дeљене променљиве и синхронизацију за приступ овим променљивама. Заштићени тип се могу дефинисати на следећи начин:

```
protected type <ProtectedType> is
    декларације процедура, функција, тачака улаза
    [private декларације приватних променљивих,
     процедура, функција, тачака улаза]
end <ProtectedType>;

protected body <ProtectedType> is
    локалне дефиниције процедура, функција, тачака улаза
end <ProtectedType>;
```

Јавне заштићене процедуре пружају ексклузивно право читања и уписа у приватне променљиве. Јавне заштићене тачне приступа личе на заштићене процедуре, али са том разликом што се може ограничiti када се сме примити позив. Ово ограничење се спроводи коришћењем *when* исказа који као аргумент срачунава булов израз. Уколико је булов израз испуњен дозвољава се пријем позива, уколико није чека се док израз не постане испуњен. Овај израз не зависи од аргумента позива, већ од стања заштићеног објекта. У било ком тренутку максимално један таск може да приступа било процедуре било тачки, док преостали таскови чекају на приступ по *FIFO* редоследу. Позивајући таск остаје блокиран све док се не заврши позивана процедура односно код да тачку улаза. Уколико је потребно да позивајући таск остане блокирана, а да се напусти ексклузивно право приступа заштићеном типу може се користити наредба за преусмеравање обраде *requeue* која омогућава рад који је спичан коришћењу условних променљивих код монитора.

**requeue** назив процедуре или нове тачке улаза [**with abort**];

На овај начин је могуће срачунати неки исказ на основу вредности аргумента и на основу тога одлучити да ли дозволити да се позивајући таск одблокира или је потребно да се сачекају додатни услови. Код *requeue* се приспели позив преусмерава на неку другу процедуру односно тачку приступа.

### Селективна наредба

Постоје три типа селективних наредби: селективно чекање (*selective wait*), условни позив (*conditional entry call*) и временски ограничен позив (*timed entry call*). Селективне наредбе су недетерминистичке.

- Селективно чекање (*selective wait*) има следећи облик:

```
select
    [when condition1 =>] selective_wait_alternative
or
    ...
or
    [when conditionN =>] selective_wait_alternative
[else
    sequence_of_statements]
end select;

selective_wait_alternative ::= accept_alternative |
                             delay_alternative |
                             terminate_alternative
```

```
accept_alternative ::= accept_statement [sequence_of_statements]
delay_alternative ::= delay_statement [sequence_of_statements]
terminate_alternative ::= terminate;
```

Услови *condition*, су логички изрази чија се вредност утврђује пре него што се изабере одговарајућа алтернатива. Када је неки услов задовољен или када *when* опција није наведена, каже се да је алтернатива отворена.

Извршавање селективне наредбе одвија се према следећем алгоритму:

1. Утврђује се које су алтернативе отворене. За отворене алтернативе са *delay* наредбом, ако их има, почиње се са мерењем времена.
2. Ако има отворених алтернатива са *accept* наредбом, издвајају се оне са којима је могуће одмах успоставити рандеву, тј. на чијим улазним тачкама стоје захтеви послати од стране других таскова, и недетерминистички се бира једна од њих.
3. Ако нема захтева послатих од других таскова на улазним тачкама ни једне отворене *accept* алтернативе, тј. ако се не може одмах успоставити рандеву, бира се *else* грана. Ако *else* грана не постоји, извршавање таска се суспендује све док не буде било могуће изабрати неку од отворених алтернатива; потом се недетерминистички бира једна од њих (ако их има више).
4. Ако ни једна алтернатива није изабрана у року специфицираном у некој од *delay* алтернатива, бира се отворена *delay* алтернатива која има најкраћи период чекања; ако има више *delay* алтернатива са истим временом чекања, недетерминистички се бира једна од њих.
5. Између отворених *terminate* алтернатива недетерминистички се бира једна, под условом:
  - Да не постоје активни или блокирани позиви на улазним тачкама таскова који садрже *terminate* алтернативе, а који зависе од текућег таска.
  - Сви таскови који треба да се састану (дођу на рандеву) са текућим таском су завршили рад или и они сви чекају на *terminate* алтернативи
6. Ако су све алтернативе затворене и не постоји *else* грана, онда се подиже изузетак (*exception*) *PROGRAM\_ERROR*.

Пример:

```
select
  when not BUSY =>
    accept SEIZE do
      BUSY := TRUE;
    end;
  or
    accept RELEASE do
      BUSY := FALSE;
    end;
  or
    terminate;
end select;
```

- Условни позив (*conditional entry call*) има следећи облик

```
select
    entry_call_statement
    [sequence_of_statements]
else
    sequence_of_statements
end select;
```

Код условног позива, ако је то могуће, одмах се извршава позив (*entry call*), а ако не, бира се *else* грана. На пример:

```
loop
    select
        R.SEIZE;
        return;
    else
        null; -- busy waiting
    end select;
end loop;
```

- Временски ограничен позив (*timed entry call*) има следећи облик

```
select
    entry_call_statement
    [sequence_of_statements]
or
    delay_alternative
end select;
```

Код временски ограничене селективне наредбе позив се извршава ако процес позивалац не мора да чека дуже од времена специфицираног у *delay* наредби да позвани процес прихвати рандеву; у супротном, извршава се низ наредби из *delay* алтернативе. На пример:

```
select
    CONTROLLER.REQUEST(MEDIUM) (SOME_ITEM);
or
    delay 45.0;
    -- controller too busy, try something else
end select;
```

## Превођење

Како преводилац за програмски језик Ada може се користити *gcc* преводилац. Екstenзија коју треба да имају програми писани на језику Ada је *adb*. Превођење, повезивање и креирање извршне датотеке се постиже користећи следећи скуп команди:

```
gcc -c imefajla.adb
gnatbind imefajla
gnatlink imefajla
gnatmake imefajla.adb
```

## 80

### Селективна наредба

У једној дистрибуираној апликацији већи број клијената  $C_1, \dots, C_n$  користи услуге једног сервера  $S$ , при чему се примењује приоритетни алгоритам по коме у случају да има више пристиглих захтева, сервер најпре треба да послужи клијента  $C_i$  са најмањим индексом  $i$ .

- Предложити погодну синтаксу за наредбу која реализује селективни пријем порука у складу са наведеним алгоритмом и објаснити њено значење.
- Како би се наведени проблем решио у језику CONIC?
- Како би се наведени проблем решио у језику Ada за два клијента  $C1$  и  $C2$ ? Подразумева се да клијенти упућују захтеве преко различитих улазних тачака сервера  $S$ :  $C1$  позива  $S.E1(m1)$ , а  $C2$  позива  $S.E2(m2)$ .

### Решење

- За наведени алгоритам опслуживања одговарајуће решење је детерминистичка селективна наредба:

```
select
    receive m1 from C1 -> обради клијента 1
or
    ...
or
    receive mn from Cn -> обради клијента n
end select;
```

Услови алтернатива у селективној наредби испитују се редоследом којим су наведени, при чему се извршава прва алтернатива за коју је услов испуњен и порука је стигла.

Задати алгоритам опслуживања се реализује тако што се као прва алтернатива наводи она која прима поруку од клијента највишег приоритета ( $C1$ ), а потом се редом наводе остале алтернативе, све до последње, која прима поруку од клијента најнижег приоритета.

- У језику CONIC је селективна алтернативна команда детерминистичка, тако да се може директно применити решење из тачке а). Разлика је једино у томе што је именовање процеса у језику CONIC индиректно, па би  $C_1, \dots, C_n$  заправо означавали имена улазних портова сервера  $S$  везаних на излазне портove одговарајућих клијената.

- У језику Ada је селективна алтернативна команда недетерминистичка, па се приоритет клијентских процеса мора реализовати на другачији начин. Ово је једно решење:

```
select
    accept E1(m1) do обради клијента 1 end;
or
    when E1.count = 0 => accept E2(m2) do обради клијента 2 end;
end select;
```

Пошто је селективна наредба недетерминистичка, могуће је да се изабере било која од две алтернативе. У случају да се изабере друга алтернатива, приоритет првог клијента се чува тако што се прихватље рандеву са другим клијентом условљава тиме да

први клијент није тражио рандеву ( $E_1.count$  враћа број процеса који траже рандеву преко тачке улаза  $E_1$ ).

Друго решење је следеће:

```
select
    accept E1(m1) do обради клијента 1 end;
else
    select
        accept E1(m1) do обради клијента 1 end;
    or
        accept E2(m2) do обради клијента 2 end;
    end select;
end select;
```

Овде је логика следећа: приоритет је дат првом клијенту, јер се *else* грана бира само ако први клијент није тражио рандеву. Ако би у *else* грани стајало само *accept E<sub>2</sub>(m<sub>2</sub>)*..., тада би у случају избора *else* грane сервер остао блокиран чекајући на другог клијента чак и ако први клијент у међувремену тражи рандеву. Зато се уводи селективна наредба у *else* грани, што омогућава да у случају да ни први ни други клијент нису затражили рандеву не дође до блокирања, већ до изласка из *else* грane.

## 81

## Timeout опција код селективне наредбе

Дат је следећи програмски фрагмент клијентског кода на језику Ada:

```
select
    buffer.write(x);
    NORMAL_ACTION;
or
    delay t1;
    TIMEOUT_ACTION;
end select
```

и сличан фрагмент на језику CONIC:

```
send x to outport
    wait signal => NORMAL_ACTION;
    timeout t1 => TIMEOUT_ACTION;
end
```

Претпоставимо да је време  $t1$  истекло у тренутку када је серверски процес прихватио захтев и почeo да га обрађујe, али још није одговорио. Да ли ћe сe извршити *NORMAL\_ACTION* или *TIMEOUT\_ACTION*? Шта су предности и мане наведених решења?

## Решење

У случају примера на програмском језику Ada извршићe сe *NORMAL\_ACTION*, јер је у тренутку када је позвани процес *buffer* прихватио позив *write(x)* у позивајућем процесу дефинитивно изабрана прва грана селективне наредбе. Дакле, алтернатива у клијентском процесу бира сe ако је рандеву прихваћен, односно започет, без обзира да ли је завршен.

Предност оваквог решења јесте једноставнија семантика комуникационих примитива: једном започети рандеву између процеса не може сe прекинути. Недостатак је што овако структурирана наредба не решава неке реално могуће проблеме који сe могу јавити у дистрибуираном систему, где нпр. реализација већ прихваћеног захтева на удаљеном процесу може да сe не заврши успешно (нпр. због отказа процесора на коме сe извршава позвани процес или због отказа комуникационе линије).

У случају примера на програмском језику CONIC извршићe сe *TIMEOUT\_ACTION*, јер услов за отварање прве алтернативе, а то је пријем поруке *signal* од позваног процеса, није био испуњен у току временског интервала  $t1$ .

Предност оваквог решења је да сe процесу који упућујe захтев омогућава наставак рада уколико комплетна интеракција није остварена у задатом интервалу времена, без обзира на разлог неуспеха (па и у случају отказа процесора или комуникационе линије у дистрибуираном систему). Цена тога је компликованија реализација. У случају истека временског периода, позвани процес треба обавестити да је позивајући процес поништио интеракцију пошто резултат није примљен благовремено. Међутим, позвани процес је у међувремену можда извршио одређене радње (чиме је, нпр., можда променио своје стање или извршио неку неповратну промену свога окружења), па сe тада поставља питање евентуалног поништења ефеката интеракције на пријемни процес. Такођe, ако је прекинута комуникациона линија, поставља сe питање како ћe сe уопште обавестити пријемни процес о поништењу интеракције.

## 82

## Реализација рандеву помоћу примитива за слање и пријем порука

Један дистрибуирани оперативни систем за интерпроцесну комуникацију обезбеђује поуздане операције асинхроног слања и времененски условљеног пријема облика

```
procedure snd (P: process, m: message);
function rcv (P: process, var m: message, t: duration) : boolean;
```

где тип порука *message* обухвата целе бројеве и специјалне симболе као што су *ack* и *pack* за позитивне и негативне потврде (а по потреби могу се увести и други симболи), временски тип *duration* обухвата вредности у опсегу  $0..maxtime$ , као и специјалну вредност *INF* (бесконачно,  $\infty$ ) која означава неограничено чекање, а *rcv* враћа логичку вредност која показује да ли је порука примљена у назначеном интервалу времена *t*.

Користећи *Pascal* проширен наведеним операцијама, реализовати следеће бидирекционе трансакције типа рандеву између процеса *C* који тражи услугу *SERVE* и процеса *S* који пружа ту услугу. Именовање не треба решавати на генерални начин, већ сматрати да су ово једини процеси и да знају један за другог.

```
a) {process C}           {process S}
      a,b:integer := 1;   accept SERVE(x:in integer; y:out integer) do
      ...                   y:= f(x);
      S.SERVE(a,b);       end SERVE;

b) {process C}           {process S}
      a,b:integer := 1;   select
      ...                   accept SERVE(x:in integer; y:out integer)
      select               do
      S.SERVE(a,b);       y:= f(x);
      NORMAL_ACTION;     end SERVE;
      or                  else
      delay t1;           DEFAULT_ACTION;
      TIMEOUT_ACTION;    end select;
      end select;
```

## Решење

```
a) {process C}           {process S}
      var a, b: integer;
          st: boolean;
      begin
          ...
          a := 1;
          snd(S, a);
          st := rcv(S, b, INF);
          ...
      end
      var x, y: integer;
          st: boolean;
      begin
          ...
          st := rcv(C, x, INF);
          y := f(x);
          snd(C, y);
          ...
      end
```

Дата бидирекциона трансакција демонстрира принцип рада рандеву. Клијентски процес *C* шаље поруку серверском процесу *S* и блокира се чекајући на одговор.

Серверски процес *S* чека на позив, блокирајући се док позив не стигне; када прими позив, извршава тражену услугу и потом шаље одговор клијентском процесу који на њега чека.

Поуздане операције слања и пријема обезбеђују да не може доћи до неограниченог блокирања процеса приликом чекања на поруку, односно да ће свака послата порука у коначном времену стићи до процеса који треба да је прими. Уколико операције слања и пријема не би биле поуздане, могло би да се деси да један или оба процеса остану блокирани неограничено дуго. То се може десити нпр. услед прекида комуникационе везе између клијентског и серверског процеса. Ако до прекида дође пре пријема захтева од стране серверског процеса, остаће блокиран и клијентски и серверски процес, а ако до прекида дође након пријема захтева од стране серверског процеса, остаће блокиран клијентски процес. Могући су и други сценарији, нпр. у случају да дође до отказа на неком од рачунара на којима се извршавају клијентски, односно серверски процеси.

```
b) {process C}                                {process S}

var a, b: integer;
st: boolean;
rep: message;
begin
...
a := 1;
snd(S, a);
if (rcv(S, rep, t1) and
    rep=accept)
then begin
    snd(S, ack);
    st := rcv(S, b, INF);
    NORMAL_ACTION
end
else begin
    snd(S, nack);
    TIMEOUT_ACTION
end
...
end

var x, y: integer;
st: boolean;
rep: message;
begin
...
if rcv(C, x, 0)
then begin
    snd(C, accept);
    y := f(x);
    st := rcv(C, rep, INF);
    if rep = ack then
        snd(C, y);
    else
        DEFAULT_ACTION
end
else
    DEFAULT_ACTION;
...
end
```

Ова бидирекциона трансакција је формулисана тако да путем „пробног успостављања везе” спречи могуће блокирање клијента или сервера услед прекида везе или отказа.

Клијентски процес шаље захтев серверу и уместо да се одмах блокира чекајући одговор, он чека ограничен временски период *t1*. Ако сервер одговори у наведеном року, то значи да није дошло до прекида комуникационе линије или отказа процесора на коме се извршава серверски процес, па клијент шаље поруку серверу да прихвати рандеву и потом се блокира чекајући на одговор као и код обичног рандеву; по пријему одговора од сервера клијент извршава *NORMAL\_ACTION*. Ако сервер не одговори у току временског периода *t1*, клијент шаље поруку серверу да рандеву није прихватио и извршава *TIMEOUT\_ACTION*.

Сервер се са друге стране такође не блокира неограничено дуго чекајући на клијентски позив, већ ради временски ограничен пријем (у овом случају временски период је 0, тј. ради се асинхрони пријем). Ако клијент није послао захтев до тренутка када је сервер почeo да врши проверу, не долази до рандеву и одмах се извршава *DEFAULT\_ACTION*. У случају да клијент пошаље захтев пре него што сервер изврши

проверу, сервер шаље клијенту потврду да прихвата рандеву. Може се десити, међутим, да клијент не прими од сервера одговор *accept* у наведеном временском периоду  $t1$  и да рандеву ипак мора да се откаже (у ком случају ће клијент послати поруку *nack*).

Треба приметити да овај механизам не спречава у потпуности могућност блокирања процеса услед прекида комуникационе линије или отказа. Ако до прекида линије или отказа дође у моменту када се клијент блокира чекајући на одговор од сервера или када се сервер блокира чекајући на потврду прихватања рандевуа од клијента, један или оба процеса ће остати блокирани неограничено дуго.

83

## Једноелементни бафер – проблем коректног завршетка процеса

Процедура *P* садржи локални једноелементни бафер у облику таска *Buffer*:

```
procedure P is
    ...
    task type Buffer is
        entry Put(X : in Integer);
        entry Get(X : out Integer);
    end;
    task body Buffer is
        V : Integer
    begin
        loop
            accept Put(X : in Integer) do
                V := X;
            end;
            accept Get(X : out Integer) do
                X := V;
            end;
        end loop;
    end Buffer;
    ...
begin
    ...
end P;
```

а) Шта ће се десити са таском *Buffer* када се заврше све остале упоредне активности настале извршавањем процедуре *P* и треба остварити повратак из процедуре?

б) Како треба модификовати таск *Buffer* да би се омогућио коректан завршетак процедуре *P*?

### Решење

а) Таск *Buffer* ће због присуства бесконачне петље наставити да постоји у оквиру процедуре *P*, али неће имати ко да га позове на рандеву. Због тога се процедуре *P* не може нормално завршити (таск се једино може прекинути споља командом *abort*).

б) Да би се спречила описана ситуација, уводи се селективна наредба са *terminate* алтернативом; *terminate* алтернатива се извршава када сви процеси који потенцијално могу позвати таск *Buffer* престану да постоје и изазива излазак из процедуре.

```
task type Buffer is
    entry Put(X : in Integer);
    entry Get(X : out Integer);
end;
task body Buffer is
    V : Integer;
begin
    loop
        select
            accept Put(X : in Integer) do
```

```
V := X;
end;
or
terminate;
end select;
select
accept Get(X : out Integer) do
    X := V;
end;
or
terminate;
end select;
end loop;
end Buffer;
```

## 84

## Монитор

Реализовати пакет који омогућава рад са објектима у складу са дефиницијом монитора.

- Дати решење које не обезбеђује рад са условним променљивама у монитору.
- Размотрити решење које обезбеђује рад са условним променљивама у монитору. Ради једноставности, сматрати да се операција *signal* може наћи само на крају мониторске процедуре
- Проширити решење из тачке б) додавањем функције *function Non\_Empty(C: Condition) return Boolean* која враћа информацију да ли има корисника који чекају на одговарајућем мониторском услову.

## Решење

а) Рад са мониторима омогућава се преко пакета *Monitor\_Package*, који садржи само један таск под именом *Monitor*. Идеја је да се таск *Monitor* придружи сваком корисничком објекту за који се жели обезбедити да му се приступа у складу са семантиком монитора.

```
package Monitor_Package is
    task Monitor is
        entry Enter;
        entry Leave;
    end Monitor;
end Monitor_Package;

package body Monitor_Package is
    task body Monitor is
        begin
            loop
                accept Enter;
                accept Leave;
            end loop;
        end Monitor;
    end Monitor_Package;
```

Енкапсулација корисничког објекта и таска *Monitor* у једну синтаксну целину постиже се тако што се дефинише нови пакет. Као пример, један тако синтетизован монитор *MonitorObject* изгледао би овако:

```
with Monitor_Package; use Monitor_Package;
package MonitorObject is
    -- monitor variables
    ...
    -- monitor procedures
    ...
    procedure Pi(...);
    ...
end MonitorObject;

package body MonitorObject is
```

```

begin
  ...
procedure Pi(...) is
begin
  Monitor.Enter;
  ...
  Monitor.Leave; или сигнализација на монитору
end Pi
...
end MonitorObject;

```

Пошто је рандеву могућ само са једним таском, међусобно искључивање процедура код приступа критичној секцији могуће је реализовати тако што се пре улaska у критичну секцију тражи рандеву преко исте улазне тачке; сви други таскови сем оног са којим је прихваћен рандеву се блокирају. Наредба *accept* на тај начин служи као семафор.

О експлузивности приступа монитору стварају се мониторске процедуре. То се ради тако што се пре извршавања тела процедуре тражи рандеву са придрженим таском *Monitor* преко улазне тачке *Enter*. Овај позив се прихвата само ако већ није прихваћен неки други позив, тј. ако је монитор слободан; то се види из дефиниције таска *Monitor* – следећи позив улазне тачке *Enter* не може бити прихваћен док претходни позивалац не издаје из монитора преко улазне тачке *Leave* или *Signal*.

б) Да би се подржао рад са мониторским условима, у пакету *Monitor\_Package* се поред таска *Monitor* дефинише и тип таска *Condition*.

```

package Monitor_Package is
  task Monitor is
    entry Enter;
    entry Leave;
  end Monitor;

  task type Condition is
    entry Signal;
    entry Wait;
  end Condition;
end Monitor_Package;

package body Monitor_Package is
  task body Monitor is
    begin
      loop
        accept Enter;
        accept Leave;
      end loop;
    end Monitor;

    task body Condition is
      begin
        loop
          select
            when Wait'Count = 0 =>
              accept Signal do
                Monitor.Leave;
            end Signal;
          end select;
        end loop;
      end Condition;
    end body;
  end body;
end package body;

```

or

```
    accept Wait do
        accept Signal;
    end Wait;
    end select;
end loop;
end Condition;
end Monitor_Package;
```

Идеја је слична као у решењу под а) - сваком корисничком објекту се придржује таск *Monitor*, а по потреби и одговарајући број таскова типа *Condition*. Пример монитора *MonitorObject* са мониторским условима *C1,..,Cn* би изгледао овако:

```
with Monitor_Package; use Monitor_Package;
package MonitorObject is
    C1,...,Cn : Condition;
    -- monitor variables
    ...
    -- monitor procedures
    ...
    procedure Pi(...);
    ...
end MonitorObject;

package body MonitorObject is
begin
    ...
    procedure Pi(...) is
begin
    Monitor.Enter;
    ...
    [Cj.Wait;
    ...
    Ck.Signal;]
    ...
    Monitor.Leave;
end Pi
...
end MonitorObject;
```

Разлика у односу на монитор из тачке а) је у томе да се може десити да процедура мора да чека на испуњење неког услова. У том случају процедура мора да препусти монитор другом кориснику (процедури) док чека да се услов испуни (опциони део у телу мониторске процедуре). Чекање на мониторском услову у оквиру таска *Condition* се постиже преко улазне тачке *Wait* на којој се блокирају сви позиваоци сем оног са којим је прихваћен рандеву. Блокирани позиваоци ће бити деблокирани (преко улазне тачке *Signal*) један по један сваки пут када се мониторски услов испуни. Грана селективне наредбе унутар таска *Condition* која прихвата позиве преко улазне тачке *Signal* када нема корисника који чекају на мониторском услову (*Wait'Count = 0*) служи да се спречи блокирања позиваоца на тој улазној тачки које би настало ако таск *Condition* не би прихватао рандеву.

в) Као подршка реализацији функције *Non\_Empty* код таскова типа *Condition* уводи се нова улазна тачка *Waiting*, а селективна наредба је проширења новом граном. Поред тога, у оквиру друге гране селективне наредбе уводи се петља и селективна наредба са граном за прихватање улазне тачке *Waiting*, јер се мора обезбедити да када се

прихвати рандеву на улазној тачки *Wait*, корисник и даље може позвати функцију *Non\_Empty* (а тиме и извршити рандеву преко улазне тачке *Waiting*).

```

package Monitor_Package is
    task Monitor is
        entry Enter;
        entry Leave;
    end Monitor;

    task type Condition is
        entry Signal;
        entry Wait;
        entry Waiting(B: out Boolean);
    end Condition;

    function Non_Empty(C: Condition) return Boolean;
end Monitor_Package;

package body Monitor_Package is
    task body Monitor is
        begin
            loop
                accept Enter;
                accept Leave;
            end loop;
        end Monitor;

        task body Condition is
            begin
                loop
                    select
                        when Wait'Count = 0 =>
                            accept Signal do
                                Monitor.Leave;
                            end Signal;
                        or
                            accept Wait do
                                loop
                                    select
                                        accept Signal;
                                        exit;
                                    or
                                        accept Waiting(B: out Boolean) do
                                            B := True;
                                        end Waiting;
                                    end select;
                                end loop;
                            end Wait;
                        or
                            accept Waiting(B: out Boolean) do
                                B := Wait'Count /= 0;
                            end Waiting;
                        end select;
                end loop;
            end Condition;

```

```
function Non_Empty(C: Condition) return Boolean is
    B: Boolean;
begin
    C.Waiting(B);
    return B;
end Non_Empty;
end Monitor_Package;
```

85

## Кружни FIFO бафер

Реализовати кружни FIFO бафер са  $N=10$  елемената.

- користећи механизам рандевуа
- користећи пакет *Monitor\_Package* (из задатка 84)
- користећи заштићене типове и механизам преусмеравања обраде (*queueing*)

## Решење

а) Основни проблем који треба решити јесте међусобно искључивање код приступа бафери, што се реализује преко *accept* наредбе. Решење је такво да само један процес сме да приступа бафери и у том смислу је еквивалентно решењу из задатка 3 под а):

```
package Buffer_Package is
    task type Buffer is
        entry Put(I : in Integer);
        entry Get(I : out Integer);
    end Buffer;
end Buffer_Package;

package body Buffer_Package is
    N : constant Integer := 10;
    task body Buffer is
        B : array(0..N-1) of Integer;
        Tail, Head : Integer := 0;
        Size : Integer := 0;
    begin
        loop
            select
                when Size < N =>
                    accept Put(I : in Integer) do
                        B(Tail) := I;
                    end Put;
                    Size := Size + 1;
                    Tail := (Tail + 1) mod N;
                or
                    when Size > 0 =>
                        accept Get(I : out Integer) do
                            I := B(Head);
                        end Get;
                        Size := Size - 1;
                        Head := (Head + 1) mod N;
                end select;
        end loop;
    end Buffer;
end Buffer_Package;
```

Пакет *Buffer\_Package* садржи декларацију бафера као посебног типа (*task type Buffer*), у оквиру кога су реализоване операције уписа и читања из бафера. На тај начин корисник може да дефинише више независних инстанци бафера, при чему је свака инстанца нови таск који ради упоредо са осталим инстанцима.

Сам таск је реализован у виду бесконачне петље у оквиру које постоји селективна наредба са две алтернативе, од којих једна прихвата захтеве везане за упис, а друга за читање елемената бафера. Пошто је селективна наредба недетерминистичка, захтеви за упис и читање се прихватају недетерминистички. Алтернативе се прихватају само ако постоје услови да се упис, односно читање изврше - тј. ако има слободних места у баферу, односно, у другом случају, ако бафер није празан. Ако нема места у баферу када стигне захтев за упис, односно ако је бафер празан када стигне захтев за читање, позивајући процес ће се блокирати на улазној тачки *Put*, односно *Get*. Захтев за приступ блокираног процеса ће се прихватити када услов за то буде задовољен. Може се десити и да више процеса буде блокирано на истој улазној тачки, у ком случају ће се процеси деблокирати један по један.

б) Решење које користи пакет *Monitor\_Package* је еквивалентно решењу задатка 31.

```
package Buffer_Package is
    procedure Put(V : in Integer);
    procedure Get(V : out Integer);
end Buffer_Package;

with Monitor_Package; use Monitor_Package;
package body Buffer_Package is
    Not_Empty, Not_Full : Condition;
    N : constant Integer := 10;
    Buffer : array(0..N-1) of Integer;
    Tail, Head : Integer := 0;
    Size : Integer := 0;

    procedure Put(V : in Integer) is
    begin
        Monitor.Enter;
        if Size = Buffer'Length then
            Monitor.Leave;
            Not_Full.Wait;
        end if;
        Buffer(Tail) := V;
        Tail := (Tail + 1) mod N;
        Size := Size + 1;
        Not_Empty.Signal;
    end Put;

    procedure Get(V : out Integer) is
    begin
        Monitor.Enter;
        if Size = 0 then
            Monitor.Leave;
            Not_Empty.Wait;
        end if;
        V := Buffer(Head);
        Head := (Head + 1) mod N;
        Size := Size - 1;
        Not_Full.Signal;
    end Get;
end Buffer_Package;
```

в)

```

N : constant Integer := 10;
Buffer : array(0..N-1) of Integer;

protected type Buffer_Package is
  entry Put(V : in Integer);
  entry Get(V : out Integer);

private
  entry Not_Empty_Wait(V : out Integer);
  entry Not_Full_Wait(V : in Integer);

Tail, Head : Integer := 0;
Size : Integer := 0;
end Buffer_Package;

protected body Buffer_Package is
  entry Put(V : in Integer) when true is
    begin
      if Size = Buffer'Length then
        requeue Not_Full_Wait;
      end if;
      Buffer(Tail) := V;
      Tail := (Tail + 1) mod N;
      Size := Size + 1;
    end Put;

  entry Get(V : out Integer) when true is
    begin
      if Size = 0 then
        requeue Not_Empty_Wait;
      end if;
      V := Buffer(Head);
      Head := (Head + 1) mod N;
      Size := Size - 1;
    end Get;

  entry Not_Empty_Wait(V : out Integer) when Size > 0 is
    begin
      V := Buffer(Head);
      Head := (Head + 1) mod N;
      Size := Size - 1;
    end Not_Empty_Wait;

  entry Not_Full_Wait(V : in Integer) when Size < Buffer'Length is
    begin
      Buffer(Tail) := V;
      Tail := (Tail + 1) mod N;
      Size := Size + 1;
    end Not_Full_Wait;
end Buffer_Package;

```

86

## Произвођачи и потрошачи: комуникација помоћу кружног бафера

Реализовати процесе *Producer* и *Consumer*, од којих први представља произвођача, а други потрошача. Комуникација између процеса треба да се одвија преко кружног FIFO бафера.

- а) Решење концептирали тако да бафер буде реализован коришћењем пакета *Buffer\_Package* из задатка 85.
- б) Унапредити решење из тачке а) тако да процес производач и потрошач могу да се покрену и зауставе по жељи.

### Решење

а) Пакет *Buffer\_Package* има готове процедуре *Put* и *Get* за приступ елементима бафера уз обезбеђено међусобно искључивање процеса.

```
with Buffer_Package; use Buffer_Package;
procedure ProducerConsumer is
    B: Buffer;
begin
    task Producer is
        end Producer;

    task Consumer is
        end Consumer;

    task body Producer is
        N: Integer;
    begin
        loop
            produce (N);
            B.Put (N);
        end loop;
    end Producer;

    task body Consumer is
        N: Integer;
    begin
        loop
            B.Get (N);
            consume (N);
        end loop;
    end Consumer;
begin
    null;
end ProducerConsumer;
```

б) Да би се производачи и потрошачи могли покретати и заустављати по жељи тасковима *Producer* и *Consumer* се додају улазне тачке *Start*, *Stop*, *Pause* и *Continue* за контролу процеса.

```
with Buffer_Package; use Buffer_Package;
procedure ProducerConsumer is
```

```
B: Buffer;

task Producer is
    entry Start;
    entry Pause;
    entry Continue;
    entry Stop;
end Producer;

task Consumer is
    entry Start;
    entry Pause;
    entry Continue;
    entry Stop;
end Consumer;

task body Producer is
    N: Integer;
begin
    accept Start;
    Producer_loop:
    loop
        select
            accept Stop;
            exit;
        or
            accept Pause;
            select
                accept Continue;
            or
                accept Stop;
                exit Producer_loop;
            end select;
        or
            delay 0;
            produce (N);
            B.Put(N);
        end select;
    end loop;
end Producer;

task body Consumer is
    N: Integer;
begin
    accept Start;
    Consumer_loop:
    loop
        select
            accept Stop;
            exit;
        or
            accept Pause;
            select
                accept Continue;
            or
```

```
accept Stop;
  exit Consumer_loop;
end select;
or
delay 0;
B.Get(N);
consume (N);
end select;
end loop;
end Consumer;
begin
  null;
end ProducerConsumer;
```

87

## Читаоци и писци

Написати програм на програмском језику Ada који решава проблем читалаца и писаца.

## Решење

```

with ada.text_io, ada.integer_text_io;
use ada.text_io, ada.integer_text_io;

procedure ReadersWriters is
    NR : constant Integer := 5;
    NW : constant Integer := 2;
    type ReadersID is range 0..NR-1;
    type WritersID is range 0..NW-1;

    task Control is
        entry Start_Read;
        entry Stop_Read;
        entry Start_Write;
        entry Stop_Write;
    end Control;

    task body Control is
        Done : boolean;
        ReadCount : Integer range 0..NR;
        Busy : boolean;
    begin
        ReadCount := 0;
        Busy := false;
        loop
            select
                when (not Busy) =>
                    accept Start_Read;
                    Put_Line ("Start_Read");
                    ReadCount := ReadCount + 1;
                    null;
                or
                    accept Stop_Read;
                    Put_Line ("Stop_Read");
                    ReadCount := ReadCount - 1;
                    null;
                or
                    when (not Busy) and (ReadCount = 0) =>
                        accept Start_Write;
                        Busy := true;
                        Put_Line ("Start_Write");
                        null;
                or
                    accept Stop_Write;
                    Busy := false;
                    Put_Line ("Stop_Write");
                    null;
            end select;
        end loop;
    end;
end ReadersWriters;

```

```
        or
            terminate;
        end select;
    end loop;
end Control;

task type Reader is
    entry Init(ID : ReadersID);
end Reader;

task body Reader is
    Done : boolean;
    I : Integer := 0;
    MyID : ReadersID;
begin
    accept Init(ID : ReadersID) do
        MyID := ID;
        Put_Line ("init Reader " & MyID'Img);
    end Init;

    loop
        Control.Start_Read;
        Put_Line ("Reader " & MyID'Img);
        delay 1.0;
        Control.Stop_Read;
        I := I + 1;
        if I > 10 then
            Done := true;
        end if;
        exit when Done;
    end loop;
end Reader;

task type Writer is
    entry Init(ID : WritersID);
end Writer;

task body Writer is
    Done : boolean;
    I : Integer := 0;
    MyID : WritersID;
begin
    accept Init(ID : WritersID) do
        MyID := ID;
        Put_Line ("init Writer " & MyID'Img);
    end Init;

    loop
        delay 3.0;
        Control.Start_Write;
        Put_Line ("Writer " & MyID'Img);
        delay 1.9;
        Control.Stop_Write;
        I := I + 1;
        if I > 10 then
```

```
        Done := true;
      end if;
      exit when Done;
   end loop;
end Writer;

Readers : array(ReadersID) of Reader;
Writers : array(WritersID) of Writer;

begin
  for J in ReadersID loop
    Readers(J).Init(J);
  end loop;

  for J in WritersID loop
    Writers(J).Init(J);
  end loop;

  delay 20.0;
end ReadersWriters;
```

88

## Филозофи за ручком

Решити проблем филозофа за ручком, описан у задатку 4, коришћењем пакета *Monitor\_Package* из задатка 84.

## Решење

Решење са мониторима одговара шестом решењу из задатка 4.

Пакет *Forks\_Package* је имплементација монитора са процедурима за приступ виљушкама. Елементи низа *ForksAvailable* представљају број виљушака доступних сваком филозофу појединачно, а елементи низа *OK\_to\_Eat* представљају мониторске услове који се користе за контролу приступа виљушкама (блокирање и деблокирање филозофа када приступ није одмах могућ). Филозофи су декларисани у виду типа *Philosopher*, како би се могао инстанцирати њихов потребан број у оквиру процедуре *DiningPhilosophers*. Што се тиче могућности настанка мртвог блокирања (*deadlock*), живог блокирања (*livelock*), изгладњивања, као и других карактеристика овог решења, важе сви закључци изведени у оквиру шестог решења задатка 4.

```

package Forks_Package is
    procedure Take_Fork(I: Integer);
    procedure Release_Fork(I: Integer);
end Forks_Package;

with Monitor_Package; use Monitor_Package;
package body Forks_Package is
    N: Integer := 5;
    ForksAvailable: array(0..N-1) of Integer := (others => 2);
    OK_to_Eat: array(0..N-1) of Condition;

    procedure Take_Fork(I: Integer) is
    begin
        Monitor.Enter;
        if ForksAvailable(I) /= 2 then
            Monitor.Leave;
            OK_to_Eat(I).Wait;
        end if;
        ForksAvailable((I+1) mod N) := ForksAvailable((I+1) mod N)-1;
        ForksAvailable((I-1) mod N) := ForksAvailable((I-1) mod N)-1;
        Monitor.Leave;
    end Take_Fork;

    procedure Release_Fork(I: Integer) is
    begin
        Monitor.Enter;
        ForksAvailable((I+1) mod N) := ForksAvailable((I+1) mod N)+1;
        ForksAvailable((I-1) mod N) := ForksAvailable((I-1) mod N)+1;
        if ForksAvailable((I+1) mod N) = 2 then
            OK_to_Eat((I+1) mod N).Signal;
            Monitor.Enter;
        end if;
        if ForksAvailable((I-1) mod N) = 2 then
            OK_to_Eat((I-1) mod N).Signal;
        end if;
    end Release_Fork;

```

```
else
    Monitor.Leave;
end if;
end Release_Fork;
end Forks_Package;

with Forks_Package; use Forks_Package;
procedure DiningPhilosophers is
    N : Integer := 5;
    type Philosopher_ID is range 0..N-1;

    task type Philosopher is
        entry Init(ID: Philosopher_ID);
    end Philosopher;

    Philosophers: array(Philosopher_ID) of Philosopher;

    task body Philosopher is
        I: Philosopher_ID;
    begin
        accept Init(ID: Philosopher_ID) do
            I := ID;
        end Init;
        loop
            Think();
            Take_Fork(Integer(I));
            Eat();
            Release_Fork(Integer(I));
        end loop;
    end Philosopher;
begin
    for J in Philosopher_ID loop
        Philosophers(J).Init(J);
    end loop;
end DiningPhilosophers;
```

89

## Нервозни пушачи

Написати програм на програмском језику Ada који решава проблем нервозних пушача (*The Cigarette Smokers' Problem*).

### Решење

```
with ada.text_io, ada.integer_text_io, ADA.NUMERICS.DISCRETE_RANDOM;
use ada.text_io, ada.integer_text_io;

procedure CigaretteSmokers is

    MinNum : constant Integer := 1;
    MaxNum : constant Integer := 3;

    task Table is
        entry Yes_Matches;
        entry Yes_Tobacco;
        entry Yes_Paper;
        entry goods(InMatches, InTobacco, InPaper: in boolean);
    end Table;

    task body Table is
        Done : boolean;
        Matches, Tobacco, Paper: boolean;
    begin
        Done := false;
        Matches := false;
        Tobacco := false;
        Paper := false;
        loop
            select
                when (Matches and Tobacco) =>
                    accept Yes_Paper do
                        Matches := false;
                        Tobacco := false;
                        Put_Line ("start Yes_Paper");
                        null;
                    end Yes_Paper;
                or
                when (Tobacco and Paper) =>
                    accept Yes_Matches do
                        Tobacco := false;
                        Paper := false;
                        Put_Line ("start Yes_Matches");
                        null;
                    end Yes_Matches;
                or
                when (Matches and Paper) =>
                    accept Yes_Tobacco do
                        Matches := false;
                        Paper := false;
                    end Yes_Tobacco;
            end select;
            if Done then
                exit;
            end if;
        end loop;
    end Table;
end CigaretteSmokers;
```

```
Put_Line ("start Yes_Tobacco");
null;
end Yes_Tobacco;
or
accept goods(InMatches, InTobacco, InPaper
: in boolean) do
    Matches := InMatches;
    Tobacco := InTobacco;
    Paper := InPaper;
    Put_Line ("goods");
    null;
end goods;
or
terminate;
end select;
exit when Done;
end loop;
end Table;

task Agent is
    entry OK;
end Agent;

task body Agent is
subtype MyNumber is Integer range MinNum .. MaxNum;
package MyRandomNumbers is
    new ADA.NUMERICS.DISCRETE_RANDOM(myNumber);

use MyRandomNumbers;

Done : boolean;
I : integer;
RandomNumber : MyNumber;
Seed : GENERATOR;

begin
    RESET(Seed);
loop
    RandomNumber := RANDOM(Seed);
    I := RandomNumber;
    Put_Line ("ALIVE");
    if (I = 1) then
        Table.goods(false, true, true);
        Put_Line("goods(false, true, true)" & I'Img);
    elsif (I = 2) then
        Table.goods(true, false, true);
        Put_Line ("goods(true, false, true)" & I'Img);
    elsif (I = 3) then
        Table.goods(true, false, true);
        Put_Line ("goods(true, true, false)" & I'Img);
    end if;

select
    accept OK;
    Put_Line ("OK");
end task;
```

```
        or
            terminate;
        end select;

        delay 1.0;
    end loop;
end Agent;

task Smoker_With_Paper is
end Smoker_With_Paper;

task body Smoker_With_Paper is
    Done : boolean;
    I : integer := 0;
begin
    loop
        Table.Yes_Paper;
        Put_Line ("Yes_Paper " & I'Img);
        delay 1.0;
        Agent.OK;
        I := I + 1;
    end loop;
end Smoker_With_Paper;

task Smoker_With_Matches is
end Smoker_With_Matches;

task body Smoker_With_Matches is
    Done : boolean;
    I : integer := 0;
begin
    loop
        Table.Yes_Matches;
        Put_Line ("Yes_Matches " & I'Img);
        delay 1.0;
        Agent.ok;
        I := I + 1;
    end loop;
end Smoker_With_Matches;

task Smoker_With_Tobacco is
end Smoker_With_Tobacco;

task body Smoker_With_Tobacco is
    Done : boolean;
    I : integer := 0;
begin
    loop
        Table.Yes_Tobacco;
        Put_Line ("Yes_Tobacco " & I'Img);
        delay 1.0;
        Agent.ok;
        I := I + 1;
    end loop;
end Smoker_With_Tobacco;
```

```
begin
    delay 1.0;
end CigaretteSmokers;
```

## Задаци за вежбу

- Реализовати пакет *Semaphore\_Package* који имплементира операције *Init*, *Wait* и *Signal* за рад са бинарним и бројачким семафорима.
- Решити проблем читалаца и писаца коришћењем пакета *Monitor\_Package*.
- Решити проблем филозофа за ручком коришћењем пакета *Semaphore\_Package*.
- Решити проблем филозофа за ручком без коришћења других пакета.

## Виртуелни простори

Програмски простор се може дефинисати као простор који види програмер када пише програм. У њему се налазе како активни објекти (процеси, нити, агенти,...) тако и пасивни подаци. Тада простор не мора бити једнодимензионалан, као код класичних парадигми конкурентног програмирања које користе програмски простор и као простор за координацију и као простор за израчунавање, већ се он може концептуално организовати на различите начине. Начин организације програмског простора је често условљен применом у некој области.

Виртуелни програмски простор је издвојени део глобалног програмског простора који служи за координацију активних делова програма – њихов координациони медијум. У општем случају виртуелни простор може обухватати и податке и активне програмске ентитете. Уколико је виртуелни простор резервисан само за податке, онда се за њега користи и назив *виртуелна дељена меморија* (*virtual shared memory*) или *простор података* (*data space*). Сам назив „виртуелни“ потиче од чињенице да издвојени простор постоји само у контексту координационог модела, док је физички он реализован у глобалном програмском простору.

Постоје различите парадигме конкурентног програмирања које су засноване на моделу виртуелних простора. Оне се разликују према разним критеријума, од којих су најчешћи следећи:

- Вишеструки простори - Да ли постоји само један виртуелни простор или више њих?
- Активни ентитети - Да ли виртуелни простор подржава неки извршни модел који омогућава да се у њега поред пасивних података могу смештати и активни програмски ентитети?
- Дистрибутивност - Да ли је виртуелни простор доступан преко мреже удаљеним клијентима?
- Грануларност приступа - Да ли је могуће стављати и узимати више елемената у и из виртуелног простора одједном?
- Објектна-орјентисаност - Да ли имплементација подржава ОО концепте?
- Јављање уписа - Да ли постоји механизам за јављање када се елементи стављају у виртуелни простор?
- Контрола приступа - Да ли је приступ виртуелном простору дозвољен неауторизованим клијентима?
- Механизам приступа - Који алгоритми се примењују код приступа и претраге виртуелног простора?
- Конфигурабилност - Да ли се може мењати семантика и структура виртуелног простора?
- Екстерни клијенти - Да ли виртуелни простор могу користити екстерни клијенти без посебних прилагођавања?

---

## Простор торки (C-Linda)

Познато је да су сви координациони модели засновани на чињеници да је посао координације ортогоналан у односу на посао израчунавања. То значи да је могуће дефинисати координациони модел који се може надовезати на неки постојећи програмски језик у виду скупа правила или наредби који имплементирају тај модел, независно од језика који имплементира рачунски (*computational*) модел. Такав је случај са *Linda* координационим моделом, првим таквим моделом који се појавио. Имплементације библиотеке *Linda* постоје за језике *C*, *Pascal*, *Fortran*, *Ada*, *Prolog*, *Lisp*, *Java*, да набројимо само неке. Овде ћемо говорити о C-*Linda* имплементацији.

*Linda* је име координационог модела који се заснива на виртуелном простору под називом *простор торки* (*tuple space*). Овај назив потиче од математичке дефиниције појма *торка* (*tuple*) - уређен скуп елемената, што асоцира на чињеницу да ентитети којима се приступа у виртуелном простору имају облик торке. Простор торки који користи *Linda* је јединствен и није структуриран. Типови података у оквиру торке се наслеђују из „језика домаћина“ (*host language*), тј. језика у коме је реализован рачунски модел. Претрага у простору торки је асоцијативна, према вредности поља торки.

У *Linda* моделу торка може бити пасивна (када су сви елементи подаци или променљиве), или активна (када је бар један од елемената торке активан процес). Комуникација између процеса се одвија уз помоћ одговарајућих примитива које служе за стављање и узимање торки у и из простора торки. Синхронизација се врши тако што процеси прате појаву торки у простору торки помоћу блокирајућих примитива за приступ, слично као код семафора. Нове торке могу да креирају само постојеће активне торке.

Скуп правила (наредби, примитива) за приступ простору торки је следећи:

**out(t)** – ставља нову пасивну торку  $t$  у простор торки, након што се изврши израчунавање поља торке; позивајући процес одмах наставља рад, не блокирајући се.

**in(p)** – тражи пасивну торку  $t$  у простору торки која одговара торки-шаблону  $p$  (торка-шаблон је торка чија су поља или вредности или променљиве којима претходи знак питања). Ако је не нађе, процес позивалац се суспендује (блокира) док се одговарајућа торка не појави; ако је нађе, додељује вредности поља торке  $t$  одговарајућим променљивама из торке-шаблона  $p$ , а потом брише нађену торку из простора торки. Ако постоји више торки које одговарају торки-шаблону  $p$ , недетерминистички се бира једна од њих.

**rd(p)** – има исто понашање као *in(p)*, само што се пронађена торка не брише из простора торки.

**eval(t)** – ставља активну торку  $t$  у простор торки; у суштини има исто понашање као *out(t)*, сам што се израчунавање поља торке ради након стављања у простор торки. За свако поље торке  $t$  које садржи функцију која враћа неку вредност, имплицитно се креира нов процес. Позивајући процес одмах наставља рад, не блокирајући се. Када сва активна поља буду израчуната (процеси креирани над функцијама заврше рад и врате вредност), торка постаје пасивна.

**inp(p)** - тражи пасивну торку  $t$  у простору торки која одговара торки-шаблону  $p$ . Ако је не нађе враћа *FALSE*; ако је нађе, додељује вредности поља торке  $t$  променљивама из торке-шаблона  $p$ , брише торку  $t$  из простора торки и враћа *TRUE*. Ако постоји више торки које одговарају торки-шаблону  $p$ , недетерминистички се бира једна од њих.

**rdp(p)** – има исто понашање као *inp(p)*, само што се пронађена торка не брише из простора торки.

**Примери:**

- `out ("string", 5.1, 7, "another string")`

Креира пасивну торку са датим вредностима поља и ставља је у простор торки.

- `rd ("string", ?f1, ?i1, "another string")`

Након извршавања наредбе из претходног примера, а потом и ове наредбе, променљива *f1* ће имати вредност 5.1, а променљива *i1* вредност 7.

- `in ("string", ?f2, ?i2, "another string")`

Након извршавања наредби редом из свих претходних примера, а потом и ове наредбе, променљива *f2* ће имати вредност 5.1, променљива *i2* вредност 7, а торка ће бити избрисана из простора торки.

- `eval ("e", 7, exp(7))`

Креира се активна торка и процес позивалац одмах наставља са радом. Израчунавање израза *exp(7)* имплицитно доводи до креирања новог процеса. Када он буде завршен, торка постаје пасивна и њен трећи елемент ће садржавати вредност *e*<sup>7</sup>.

- `rd ("e", 7, ?val)`

Након што израз из претходног примера буде израчунат, променљива *val* ће добити вредност *e*<sup>7</sup>.

- `out ("semaphore")`

Ово је еквивалент наредбе *signal(semaphore)*: Ако се изда више истих наредби, креираје се сваки пут нова торка истог садржаја; да би се семафор иницијализовао на вредност *n*, наредбу треба извршити *n* пута.

- `in ("semaphore")`

Ово је еквивалент наредбе *wait(semaphore)*: Ако у простору торки постоји одговарајућа торка, онда процес наставља са радом; ако нема одговарајућих торки, наредба се блокира док се торка не појави.

90

## Филозофи за ручком

Решити проблем филозофа за ручком описан у задатку 4.

### Решење

Овде је дато решење које користи приступ из трећег решења задатка 4. Сви закључци који су важили за то решење важе и овде. Између осталог, мртво блокирање, *deadlock* се спречава тако што се број филозофа који смеју да приступају виљушкама ограничава на  $N-1$ , што се контролише помоћу торке ("ticket"). Ова торка заправо представља *C-Linda* имплементацију одговарајућег семафора из поменутог решења; иницијализација семафора се ради у оквиру процедуре *initialize* тако што се у простор торки ставља  $N-1$  торка ("ticket").

```
void philosopher(int i) {
    int left=i, right=(i+1)%N;
    while (1) {
        think();
        in("ticket");
        in("fork",left);
        in("fork",right);
        eat();
        out("fork",left);
        out("fork",right);
        out("ticket");
    }
}

void initialize(int N) {
    int i;
    for (i=0; i<N; i++) {
        out("fork",i);
        eval(phiosopher(i));
        if (i<N-1) out("ticket");
    }
}
```

91

## Произвођачи и потрошачи

Решити проблем произвођача и потрошача. Водити рачуна о томе да максимално  $B$  производа у неком тренутку може бити у простор торки.

### Решење

```
const int NC = ...;
const int NP = ...;
const int B = ...;

void consumer(){
    int tail, data;
    while(1){
        in("tail", ?tail);
        out("tail", tail + 1);
        in("buffer", tail, ?data);
        out("space");
        consuming(data);
    }
}

void producer(){
    int head, data;
    while(1){
        producing(data);
        in("space");
        in("head", ?head);
        out("head", head + 1);
        out("buffer", head, data);
    }
}

void init () {
    int i;
    out("head", 0);
    out("tail", 0);
    for (i = 0; i < B; i++)
        out("space");

    for (i = 0; i < NC; i++)
        eval (consumer ());
    for (i = 0; i < NP; i++)
        eval (producer ());
}
```

92

## Клијенти и сервери – обрада захтева по FIFO принципу (случај са једним сервером)

У неком дистрибуираном систему постоји више клијената и један сервер. Сервер треба да обрађује захтеве оним редом којим их клијенти шаљу. Написати код за процесе *client* и *server*, као и одговарајући код за иницијализацију ако:

- клијенти не могу послати нови захтев серверу пре него што приме одговор
- клијенти могу послати више захтева серверу пре него што приме одговор.

### Решење

а) Сваки захтев који клијент шаље је облика ("request", *indexR*, *request*), где *indexR* представља јединствени идентификатор захтева (овде је у питању број, па се користи термин индекс). Вредност индекса следећег захтева чува се у простору торки у облику ("next request", *indexR*).

Када клијент жели да пошаље нови захтев, очитава индекс последњег захтева са *in*, а потом га инкрементира и одмах враћа назад како би неки други клијент који чека да приступи тој торки могао што пре да се деблокира. Сваки нови захтев ће имати различит индекс, за један већи од претходног. Овај једноставни механизам обезбеђује потпуну синхронизацију рада клијената и сервера. Након што пошаље захтев, клијент чека да се у простору торки појави одговор са истим индексом и узима га са *in*.

Сервер обрађује захтеве редом којим се шаљу. Пошто се сваки захтев јединствено обележава индексом могуће је једноставно упаривање захтева и одговора тако што се одговор обележава истим индексом који је имао одговарајући захтев. Треба приметити да је променљива *indexR* код процеса *server* локална.

```
void client() {
    unsigned int indexR; /*index of the request to be sent*/
    request_type request;
    reply_type reply;

    while (1) {
        ...
        in ("next request", ?indexR);
        out("next request", indexR+1);
        out("request", indexR, request);

        ...
        in ("reply", indexR, ?reply);
        ...
    }
}

void server(unsigned int indexR) {
    request_type request;
    reply_type reply;

    while (1) {
        ...
        in ("request", indexR, ?request);
```

```
    ...
    out("reply", indexR, reply);
    indexR++;
    ...
}
}

void initialize() {
    ...
}
```

б) Једина разлика у односу на решење под а) је у томе што клијент шаље више ( $N$ ) захтева одједном, па индекс захтева мора да се увећа за  $N$  када се након читања враћа у простор торки. Одговори се чекају оним редом којим су захтеви послати, тј. редом којим су клијенти приступали торци са индексом следећег захтева. Код сервера се не мења, јер захтеве и даље обрађује по FIFO редоследу.

```
void client() {
    unsigned int indexR; /*index of the request to be sent*/
    request_type request;
    reply_type reply;

    while (1) {
        ...
        in ("next request", ?indexR);
        out("next request", indexR+N);
        out("request", indexR, request);
        ...
        out("request", indexR+N-1, request);
        ...
        in ("reply", indexR, ?reply);
        ...
        in ("reply", indexR+N-1, ?reply);
    }
}
```

93

## Клијенти и сервери – обрада захтева по FIFO принципу (случај са више сервера)

У неком дистрибуираном рачунарском систему постоји више клијената и више сервера. Број клијената је произвољан. У почетку постоји само један сервер, а временом му се могу прикључити и други. Сваки од захтева клијената приhvата један (било који) од сервера, опслужује га и шаље одговор клијенту. Захтев који је упућен раније има предност над оним који је упућен касније. Постоји и процес *summation* који с времена на време захтева од свих сервера да му пошаљу број обрађених захтева до тог тренутка и то тако што сви сервери заврше са обрадом захтева на коме тренутно раде, шаљу процесу *summation* тражени податак и не узимају нови захтев све док овај процес не добије податке од свих сервера. Написати код за процесе *client*, *server* и *summation*, као и одговарајући код за иницијализацију.

### Решење

Код процеса клијената је исти као у претходном задатку. Главна разлика је у томе што може постојати више сервера, који међусобно морају да координирају рад код приhvатања захтева. Координација се изводи на исти начин као и код клијената – уместо локалне променљиве *indexR* уводи се торка ("next request to serve", *indexR*) која садржи индекс следећег захтева који треба да се обради. Пошто су индекси захтева уређени према редоследу слања и обрада ће се вршити по FIFO редоследу. Пошто број сервера може рasti, на почетку сваког серверског процеса стоји код којим он пре почетка свог рада ажурира тренутни број сервера у систему.

Процес *summation* своју активност најављује стављањем торке ("summing") у простор торки. Када детектују да је она присутна, сервери стављају торку са извештајем у простор торки и чекају да се обрада заврши како би наставили са радом.

```
void server() {
    unsigned int indexR; /*index of the request to be served*/
    unsigned int Ns;
    unsigned int served=0;
    request_type request;
    reply_type reply;

    in ("total number of servers", ?Ns);
    out("total number of servers", Ns+1);

    while (1) {
        if (rdp("summing")) {
            out("served", served);
            while (rdp("summing")); /*busy wait*/
        }
        else {
            ...
            in ("next request to serve", ?indexR);
            out("next request to serve", indexR+1);
            in ("request", indexR, ?request);
            ...
            out("reply", indexR, reply);
            served++;
        }
    }
}
```

```
    ...
}
}

void summation(){

    unsigned int Ns;
    unsigned int served;
    unsigned int totalserved;

    ...
    out("summing");
    rd("total number of servers", ?Ns);
    totalserved=0;
    for(i=0; i<Ns; i++) {
        in ("served", ?served);
        totalserved += served;
    }
    in ("summing");
    ...
}

void initialize(){

    unsigned int i;
    unsigned int Nc=10; /*number of clients*/
    out("total number of servers", 0);
    out("next request to serve", 0);
    eval(server());
    out("next request", 0);
    for(i=0; i<Nc; i++) eval(client());
}
```

➔ Унапредити решење задатка тако да за време рада процеса *summation* сервери не морају да раде запослено чекање.

Како би се омогутило да сервери могу да долазе и одлазе али и да се операција сумирања обавља уведен је индикатор стања система, запис *stage* који има вредност *summing* уколико се тренутно обавља операција сумирања и вредност *noaction* уколико се тренутно не обавља сумирање. Дозвољено је да сервер започиње са радом само уколико се не обавља операција сумирања. Дозвољено је да сервер заврши свој рад у било ком тренутку, али уколико је тренутно у току операција сумирања мора поплати своје резултате. Како би се избегло запослено чекање уведен је синхронизација на баријери између процеса сервера, али и процеса суматора. Уколико је започело сумирање и сервер је то приметио, онда шаље своје стање и чека да се сумирање заврши. Када се сумирање заврши суматор иницијализује баријеру, *notsumming* на укупан број сервера који треба да је прођу. Уколико сервер није последњи сервер који је дошао до баријере увећава број сервера који су прошли баријеру и наставља са радом. Уколико је последњи сервер који је дошао до баријере онда поставља индикатор система на вредност *noaction* како би могао да се започне нови циклус рада.

Приликом одласка сервер проверава да ли се ради о операцији сумирања. Ако се ради онда обавља исти посао као и да је захтев за сумирање пристигао док је ради, шаље

стање и чека на баријери. Пошто захтев за ново сумирање може доћи пре него што сервер оде онда се операција обавља у петљи све док има нових захтева за сумирање. Треба приметити да се ово разликује од запосленог чекања. Код запосленог чекања процес непрекидно проверава стање док се у овом случају могло десити да је покренута нова операција сумирања.

Нови процес сумирања може започете уколико је индикатор система на вредност *noaction* што знали да је претходна операција сумирања завршена и да су сви сервери прошли баријеру. Треба приметити да уколико је процес који ради сумирање нашао а није било активних сервера онда се не поставља баријера већ се индикатор поставља на неутралну вредност јер нема процеса сервера који би прошли кроз баријеру и поставили индикатор на неутралну вредност.

```
const int noaction = 0;
const int summing = 1;

void server(){
    int cnt;
    unsigned int indexR; /*index of the request to be served*/
    unsigned int Ns;
    unsigned int served=0;
    request_type request;
    reply_type reply;

    in("stage", noaction);
    in ("total number of servers", ?Ns);
    out("total number of servers", Ns+1);
    out("stage", noaction);

    while (1) {
        if (rdp("stage", summing)) {
            out("served", served);
            in("notsumming", ?Ns, ?cnt);
            if (Ns==cnt+1) {
                out("stage", noaction);
            } else{
                out("notsumming", Ns, cnt+1);
            }
        }
        else {
            ...
            in ("next request to serve", ?indexR);
            out("next request to serve", indexR+1);
            in ("request", indexR, ?request);
            ...
            out("reply", indexR, reply);
            served++;
            ...
        }
    }

    int stage;
    in("stage", ?stage);
    while(stage==summing) {
```

```
    out("stage", stage);
    out("served", served);
    in("notsumming", ?Ns, ?cnt);
    if(Ns==cnt+1){
        out("stage", noaction);
    } else{
        out("notsumming", Ns, cnt+1);
    }
    in("stage", ?stage);
}
in ("total number of servers", ?Ns);
out("total number of servers", Ns-1);
out("stage", stage);
}

void summation(){

    unsigned int Ns;
    unsigned int served;
    unsigned int totalServed;
    ...
    in("stage", noaction);
    out("stage", summing);

    rd("total number of servers", ?Ns);
    totalServed=0;
    for(i=0; i<Ns; i++) {
        in ("served", ?served);
        totalServed += served;
    }
    in("stage", summing);
    if(Ns == 0){
        out("stage", noaction);
    } else{
        out("notsumming", Ns, 0);
    }
    ...
}

void initialize(){

    unsigned int i;
    unsigned int Nc=10; /*number of clients*/
    out("total number of servers", 0);
    out("next request to serve", 0);
    out("stage", noaction);

    eval(server());
    out("next request", 0);
    for(i=0; i<Nc; i++) eval(client());
}
}
```

94

## Клијенти и сервери – селекција сервера по *round robin* редоследу

У неком дистрибуираном систему постоји више клијената и више сервера. Број клијената је произвољан. Сервер који прихвата захтев бира се по *round robin* редоследу. За сваки сервер важи да захтев који му је раније упућен има предност над оним који му је упућен касније. Написати код за процесе *client* и *server*, као и одговарајући код за иницијализацију под следећим условима:

- У току рада се могу прикључивати нови сервери
- Број сервера се може произвољно мењати у току рада.

### Решење

a) Због реализације *round robin* редоследа клијенти морају знати колико има сервера. Пошто број сервера може рasti, ова информација се мора тражити сваки пут када клијент жели да пошаље захтев. Да би реализација *round robin* принципа била што једноставнија, индекси сервера су узастопни цели бројеви.

Сваки сервер има свој ред захтева како би могао да их обрађује оним редом који пристижу. Да би се захтеви могли јединствено идентификовати потребно је знати и индекс сервера и индекс захтева на том серверу. На почетку сваког серверског процеса стоји код којим он пре почетка свог рада ажурира тренутни број сервера у систему.

```
void client() {
    unsigned int indexR; /*index of the request to be sent*/
    unsigned int indexS; /*index of the server to send the
                           request to*/
    unsigned int Ns; /*number of servers*/
    request_type request;
    reply_type reply;

    while (1) {
        ...
        /*Choosing the server using "round robin" scheduling*/
        in ("next server", ?indexS);
        rd ("total number of servers", ?Ns);
        out("next server", Ns>0?(indexS+1)%Ns:0);

        in ("next request", indexS, ?indexR);
        out("next request", indexS, indexR+1);
        out("request", indexS, indexR, request);
        ...
        in ("reply", indexS, indexR, ?reply);
        ...
    }
}

void server() {
    unsigned int indexS; /*index of the server*/
```

```
unsigned int indexR; /*index of the request to be served*/
request_type request;
reply_type reply;

in ("total number of servers", ?indexS);
out("total number of servers", indexS+1);
indexR = 0;
out("next request", indexS, indexR);

while (1) {
    ...
    in ("request", indexS, indexR, ?request);
    ...
    out("reply", indexS, indexR, reply);
    indexR++;
    ...
}

void initialize(){

    unsigned int i;
    unsigned int Nc=10;
    unsigned int Ns=3;
    out("total number of servers", 0);
    out("next server", 0);
    for(i=0; i<Ns; i++) eval(server());
    for(i=0; i<Nc; i++) eval(client());
}

6)
void client(){
    unsigned int indexR; /*index of the request to be sent*/
    unsigned int indexS; /*index of the server to send the request
to*/
    unsigned int Ns; /*number of servers*/
    request_type request;
    reply_type reply;

    while (work()) {

        /* Choosing the server using "round robin" scheduling */
        in("clientLock", 0);
        in("next server", ?indexS);
        rd("total number of servers", ?Ns);
        out("next server", Ns> 0 ? (indexS+1) % Ns: 0);
        in("next request", indexS, ?indexR);
        out("next request", indexS, indexR+1);
        out("request", indexS, indexR, indexS, indexR, request);
        out("clientLock", 0);
        ...
        in("reply", indexS, indexR, ?reply);
        ...
    }
}
```

```
void server() {  
  
    unsigned int indexS; /*index of the server*/  
    unsigned int indexR; /*index of the request to be served*/  
    unsigned int Ns; /*number of servers*/  
    request_type request;  
    reply_type reply;  
  
    unsigned int next, tmpIndexS, tmpIndexR;  
  
    in("serverLock", 0);  
    in("total number of servers", ?Ns);  
    out("total number of servers", Ns+1);  
    indexS = Ns;  
    indexR = 0;  
    next = 0;  
    inp("init", indexS, ?indexR, ?next);  
    out("next request", indexS, next);  
    out("serverLock", 0);  
  
    while (1) {  
        ...  
        in("request", indexS, indexR,  
            ?tmpIndexS, ?tmpIndexR, ?request);  
        ...  
        out("reply", tmpIndexS, tmpIndexR, reply);  
        indexR++;  
  
        in("serverLock1");  
        //compact  
        unsigned int tmp1;  
        while (inp("compaction", indexS,  
                  ?tmpIndexS, ?tmpIndexR, ?tmp1)) {  
            reroute(indexS, tmpIndexS, indexR, tmp1);  
            indexS = tmpIndexS;  
            indexR = tmpIndexR;  
        }  
        out("serverLock1");  
        ...  
    }  
  
    in("serverLock1");  
    //compact  
    unsigned int tmp1;  
    while (inp("compaction", indexS,  
              ?tmpIndexS, ?tmpIndexR, ?tmp1)) {  
        reroute(indexS, tmpIndexS, indexR, tmp1);  
        indexS = tmpIndexS;  
        indexR = tmpIndexR;  
    }  
    end(indexS, indexR);  
    out("serverLock1");  
}
```

```
void compact(unsigned int *indexS, unsigned int *indexR) {
    unsigned int tmpIndexS, tmpIndexR, tmp1;
    while (inp("compaction", *indexS,
        ?tmpIndexS, ?tmpIndexR, ?tmp1)) {
        reroute(indexS, tmpIndexS, *indexR, tmp1);
        *indexS = tmpIndexS;
        *indexR = tmpIndexR;
    }
}

void end(unsigned int indexS, unsigned int indexR) {
    unsigned int clock, slock;
    in("clientLock", ?clock);
    in("serverLock", ?slock);
    unsigned int Ns;
    in("total number of servers", ?Ns);

    if (Ns==1) {
        // odlazi jedini
        unsigned int tmpIndexR;
        in("next request", indexS, ?tmpIndexR);
        out("init", indexS, indexR, tmpIndexR);
        in("next server", ?indexS);
        out("next server", 0);
        out("total number of servers", Ns-1);
        out("serverLock", 0);
        out("clientLock", 0);
    } else if (Ns=indexS+1) {
        // odlazi poslednji index
        unsigned int tmpIndexR;
        in("next request", indexS, ?tmpIndexR);
        out("init", indexS, tmpIndexR, tmpIndexR);

        unsigned int tmpIndexS;
        in("next server", ?tmpIndexS);
        tmpIndexS = nextServer(tmpIndexS, Ns);

        for (; indexR < tmpIndexR; indexR++) {
            unsigned int tmpIndexRFrom, tmpIndexSFrom;
            request_type request;
            in("request", indexS, indexR,
                ?tmpIndexSFrom, ?tmpIndexRFrom, ?request);
            unsigned int indexRTo;
            in("next request", tmpIndexS, ?indexRTo);
            out("next request", tmpIndexS, indexRTo+1);
            out("request", tmpIndexS, indexRTo,
                tmpIndexSFrom, tmpIndexRFrom, request);
            tmpIndexS = nextServer(tmpIndexS+1, Ns);
        }
        out("next server", tmpIndexS);
        out("total number of servers", Ns-1);
        out("serverLock", slock);
        out("clientLock", clock);
    } else {
        // odlazi a nije poslednji index
    }
}
```

```

unsigned int tmpIndexR;
in("next request", indexS, ?tmpIndexR);
unsigned int indexSTo = Ns-1, tmp;
boolean result = findMax(indexS, indexSTo);
out("compaction", indexSTo, indexS, indexR, tmpIndexR);
if (result) {
    unsigned int indexRTo;
    if (inp("next request", indexSTo, ?indexRTo)) {
        out("next request", indexSTo, indexRTo+1);
        request_type request = 1; //dummy request
        out("request", indexSTo,
              indexRTo, indexSTo, indexRTo, request);
    }
}
in("next server", ?tmp);
out("next server", indexSTo);
out("total number of servers", Ns-1);
out("serverLock", slock+1);
out("clientLock", clock+1);
}

// pokupi rezultat nastao samo zbog pomeranja
unsigned int tmpl, tmp2;
while (inp("reply", ?tmpl, ?tmp2, 1));

}

unsigned int nextServer(unsigned int start, unsigned int Ns) {
    unsigned int result = start, tmp;

    boolean found = false;
    for (int i = 0; i < Ns; i++) {
        if (!rdp("next request", result, ?tmp)) {
            result = (result+1) % (Ns-1);
        } else {
            found = true;
            break;
        }
    }
    if (!found) {
        while (!rdp("next request", result, ?tmp)) {
            result = result+1;
        }
    }
    return result;
}

boolean findMax(unsigned int indexS, unsigned int indexSTo) {
    unsigned int tmpl, tmp2, tmp3;
    boolean work = true;
    while (work && inp("compaction", indexSTo,
        ?tmpl, ?tmp2, ?tmp3)) {
        if (tmpl < indexS) {
            out("compaction", indexS, tmpl, tmp2, tmp3);
            work = false;
        } else {

```

```

        out("compaction", indexSTo, tmp1, tmp2, tmp3);
        indexSTo = tmp1;
    }
}
return work;
}

void reroute(int from, int to, int start, int end) {
    unsigned int fromServer = from;
    unsigned int toServer = to;
    unsigned int indexR = start;
    unsigned int next, indexRTo = end;
    in("next server", ?next);
    out("next server", toServer);

    in("next request", fromServer, ?next);

    out("init", fromServer, next, next);
    for (int i = start; i < next; i++) {
        unsigned int tmpIndexSFrom, tmpIndexRFrom;
        request_type request;
        in("request", fromServer, indexR,
             ?tmpIndexSFrom, ?tmpIndexRFrom, ?request);
        out("request", toServer, indexRTo,
              tmpIndexSFrom, tmpIndexRFrom, request);
        indexR = indexR+1;
        indexRTo = indexRTo+1;
    }
    out("next request", toServer, indexRTo);

    unsigned int clock, slock;
    in("clientLock", ?clock);
    out("clientLock", clock-1);
    in("serverLock", ?slock);
    out("serverLock", slock-1);
}

void initialize(){

    unsigned int i;
    unsigned int Nc=10;
    unsigned int Ns=3;
    out("clientLock", 0);
    out("serverLock", 0);
    out("serverLock1");
    out("total number of servers", 0);
    out("next server", 0);

    for(i=0; i<Ns; i++) eval(server());
    for(i=0; i<Nc; i++) eval(client());
}

```

Претходно решење би се могло додатно убрзати уколико сервер не би увек улазио у део за сажимање индекса већ ако би то чинио само ако постоји потреба да се индекси промене. Потреба за сажимањем индекса постоји само ако неки од постојећих сервера

оде, што се доста ретко дешава. На овај начин се у сваком циклусу обавља само једна провера простора торки, пошто су сажимања ретке. Приликом сажимања обавља се једна провера простора торки више.

```
void server() {
    ...
    while (work()) {
        ...
        in("request", indexS, indexR,
           ?tmpIndexS, ?tmpIndexR, ?request);
        ...
        out("reply", tmpIndexS, tmpIndexR, reply);
        indexR++;

        if (rdp("compaction", indexS,
                ?tmpIndexS, ?tmpIndexR, ?tmp1)) {
            in("serverLock1");
            //compact
            unsigned int tmp1;
            while (inp("compaction", indexS,
                       ?tmpIndexS, ?tmpIndexR, ?tmp1)) {
                reroute(indexS, tmpIndexS, indexR, tmp1);
                indexS = tmpIndexS;
                indexR = tmpIndexR;
            }
            out("serverLock1");
            ...
        }
        ...
    }
}
```

95

## Нервозни пушачи

- a) Користећи библиотеку C-Linda написати програм који решава проблем и симулира систем „нервозних пушача“.
- b) Модификовати решење под а) тако да се избегне запослено чекање вишеструким проверавањем садржаја стола.

### Решење

a) Као што је описано у задатку 23, ово је тип проблема код кога један процес производи већи број различитих ресурса које одвојено стављан на располагање процесима потрошачима. Битно је нагласити да процес производи групу ресурса имајући у виду неког конкретног потрошача, али не обавештава тог конкретног потрошача већ оставља потрошачима да се сами утврде коме су ресурси намењени. Ову проверу треба урадити недељиво.

Сам проблем има четири учесника: агента који периодично ставља артикле на сто (производјач), и три пушача (потрошачи) тако креираних да сваком недостају по два артикла за даљи рад. Процедуре за сваког од ова четири учесника се извршавају у бесконачним петљама.

Агент треба да произведе и на сто постави неки пар ресурса. Производња је смештена у методу *produce()* која враћа број из интервала од 0 до 2 и који означава који од три ресурса није произведен. За потребе тестирања, како би се обезбедила равноправност сваке комбинације, може се искористити генератор случајних бројева  $((int)((rand()^3)/RAND\_MAX))$  који генерише целобројну вредност из интервала од 0 до 2. Место на које агент ставља ове артикле је заједнички простор за све процесе (*tuple space*). Чекање на потврду о завршетку конзумирања постављених артикала се остварује помоћу наредбе *in("OK")*.

Сваки од три пушача има идентичну структуру, само се разликује скуп ресурса на којима чекају. Овде ће бити објашњено понашање за само једног од њих док се понашање за преостале добија аналогијом. Процедура за пушача који поседује неограничену количину шибица (*smoker\_with\_Matches*) има следећу структуру: прво се приступа заједничком простору да би се проверио његов садржај, уколико постоје одговарајући ресурси они се узимају у прелазу се на фазу коришћења, на крају се агент обавештава о завршеној операцији. Да би се остварило ексклузивно право провере и промене стања пушач на почетку позива наредбу *in("Watch")* којом осталим пушачима забрањује приступ заједничком простору. Када заврши са коришћењем артикала из простора било зато што се тамо не налази оно што му је неопходно или зато што је завршио са конзумирањем враћа право и другима да приступе заједничком простору користећи наредбу *out("Watch")*. Провера садржаја заједничког простора се обавља помоћу *if(rdp("Paper") && rdp("Tobacco"))*. Треба приметити да оваква провера без међусобног искључивања пушача могла довести до узајамног блокирања. Размотримо следећи сценарио по коме агент на сто папир и шибице. Тада пушач са шибицама крене да проверава услов и провери први део услова (*rdp("Paper")*), онда изгуби право да се извршава. Дође пушач са дуваном коме је садржај и био намењен, провери садржај стола и покупи тај садржај. Након тога агент постави дуван и шибице. Тада може поново да се пробуди пушач са шибицама и провери други део свог услова (*rdp("Tobacco")*) и погрешно закључи да се на столу налази све што му је неопходно. Ово би довело до узајамног блокирања свих учесника.

Торка које се користи за међусобно искључивање пушача на почетку мора да буде постављена `out("Watch")`.

```

void agent() {
    int n;
    while(1) {
        n = produce(); // (int) ((rand() * 3) / RAND_MAX);
        switch(n) {
            case 0: out("Paper");
                out("Tobacco");
                break;
            case 1: out("Tobacco");
                out("Matches");
                break;
            case 2: out("Matches");
                out("Paper");
                break;
        }
        in("OK");
    }
}

void smoker_with_Matches() {
    while(1) {
        in("Watch");
        if(rdp("Paper") && rdp("Tobacco")) {
            in("Paper");
            in("Tobacco");
            enjoy();
            out("OK");
        }
        out("Watch");
    }
}
...
void initialize () {
    eval(agent());
    eval(smoker_with_Matches());
    eval(smoker_with_Paper());
    eval(smoker_with_Tobacco());
    out("Watch");
}

```

Проблем код овог решења представља то што пушачи стално покушавају да приступе столу и сазнају његов садржај, `in("Watch")`. Због тога, у случају да се на столу не налази ништа пушачи се понашају као да запослено чекају на појављивање ресурса на столу. Боле би било да пушачи на неки начин знају о којој се итерација ради и да коначан број пута, пожељно једном, у току итерације проверавају стање стола, а да у остало време буду блокирани и не заузимају процесорске ресурсе. Решења која захтевају непрекидну проверу ресурса треба избегавати.

6)

#### Прво решење

Ово решење има за циљ да избегне непрекидно проверавање садржаја стола користећи баријеру. Баријера се на страни агента се формира додељивањем још једне

акције, то јест постављања поруке о томе да се на столу налази нови садржај, *out("Watch")*. На овај начин се спречава да пушачи више пута приступају столу, већ максимално једном у току сваке итерације. Максимално једном значи да неки од пушача не мора да приступи столу уопште у некој итерацији уколико је столу пре њега приступио пушач коме су намењени артикли постављени на сто. Када агент постави артикле на сто он поставља и обавештење да се артикли налазе на столу. Приликом приступа столу пушач чека да се појави нови садржај стола, *in("Watch")*, након чега прелази на проверу садржаја стола. Уколико се садржај стола разликује од потреба пушача он се блокира све док се не појави пушач коме су намењени артикли на столу. Да би се знато колико има блокираних пушача он увећава *num\_blocked* која се налази у заједничком простору и предаје право провере садржаја стола следећем пушачу, *out("Watch")*, а сам се блокира све до тренутка када се појави пушач коме су намењени артикли на столу. Пушач коме се намењени артикли прво узима артикле са стола, онда их користи и на крају буди све блокиране пушаче, *for(int i = 0; i < n; i++) out("next")*. Да би се избегло да погрешан пушач приступи дозволи, тј да пушач не може прећи у следећу итерацију док се претходна још није завршила, у заједничком простору, *next*, није доволично само да се она постави него је неопходно сачекати да се блокирани пушачи пробуде и да то потврде помоћу *out("nextOK")*.

```

void agent() {
    int n;
    while(1) {
        n = produce(); // (int)((rand() * 3) / RAND_MAX);
        switch(n) {
            case 0: out("Paper");
                out("Tobacco");
                break;
            case 1: out("Tobacco");
                out("Matches");
                break;
            case 2: out("Matches");
                out("Paper");
                break;
        }
        out("Watch");
        in("OK");
    }
}
void smoker_with_Matches() {
    int n = 0;
    while(1) {
        in("Watch");
        if(rdp("Paper") && rdp("Tobacco")) {
            in("Paper");
            in("Tobacco");
            enjoy();
            in("num_blocked", ?n);
            for(int i = 0; i < n; i++) out("next");
            for(int i = 0; i < n; i++) in("nextOK");
            out("num_blocked", - 0);
            out("OK");
        }
        else{
            in("num_blocked", ?n);
        }
    }
}

```

```

        out("num_blocked", n+1);
        out("Watch");
        in("next");
        out("nextOK");
    }
}
}

void initialize () {
    out("num_blocked", 0);
    eval(agent());
    eval(smoker_with_Matches());
    eval(smoker_with_Paper());
    eval(smoker_with_Tobacco());
}

```

### Друго решење

Проблем се може решити и без модификације агента, процеса произвођача, тако што би се омогућило да потрошачи могу да врате производ који су узели уколико нема оба потребна. На почетку би потрошач блокира, користећи методу *in*, док не добије први производ. Када добије први производ проверава да ли је други доступан, користећи неблокирајући методу *inp*. Ако јесте доступан користи их и враћа се на почетак. Ако није доступан други производ враћа први производ, али се након овога се не враћа на почетак него сада чека да се појави други, користећи методу *in*. Када добије други производ проверава да ли је први и даље доступан, користећи неблокирајући методу *inp*. Ако јесте доступан користи их и враћа се на почетак. Ако није доступан први производ враћа се на почетак. Код овог решења се може десити да потрошач више пута провери садржај стола у току једне итерације уколико му је садржај стола намењен. Прво може да се деси да је агент ставио први производ и да је потрошач проверио садржај стола пре него што је ставио други производ. Након тога може да се деси да један па други сусед узимају по један артикал са стола како би проверили да ли је то оно што њима треба. На овај начин се долази до тога да је потребно обавити до 4 провере расположивих ресурса.

```

void smoker_with_Matches () {
    while(1) {
        in("Paper");
        if(inp("Tobacco")) {
            enjoy();
            out("OK");
        }
        else{
            out("Paper");
            in("Tobacco");
            if(inp("Paper")) {
                enjoy();
                out("OK");
            }
            else{
                out("Tobacco");
            }
        }
    }
}

```

96

## Читаоци и писци

Решити проблем читалаца и писаца. Решење треба да обезбеди да процес који је пре стигао пре и започне операцију читања односно уписа.

**Решење**

Решење које је примењено на овом месту полази од идеје рада шалтерске службе (тикет алгоритам). На почетку сваки процес добија јединствени редни број који означава када ће започети његова обрада. Када је добио редни број процес чека све док се не појави његов редни број. Када се појави његов редни број он има право да крене да ради. Овде треба водити рачуна о додељивању редних бројева, као и о додељивању који је то следећи број процеса који сме следећи да крене да се извршава. Додељивање јединственог редног броја се постиже помоћу две наредбе *in("number", ?turn)* и *out("number", turn + 1)*. Прва из заједничког простора узима информацију колико износи текућа вредност идентификатора и смешта је у поље *turn*, док друга у заједнички простор враћа идентификатор увећан за један. Ово је део који је исти и за читаоце и за писце. Поступак чекања на право да се процес извршава је такође јединствен и за читаоце и за писце и остварује се помоћу *in("next", turn)* која чека да се у заједничком простору појави идентификатор процеса коме се дозвољава да започне рад. Тада идентификатор мора да се поклони са јединственим редним бројем који је добио сваки процес. Након овог заједничког дела прелази се на делове који су специфични за читаоце и за писце. Уколико се ради о читаоцу он најпре увећава број читалаца за један а најави тада дозвољава следећем процесу да крене да се извршава. Када заврши са читањем читалац само умањује број активних читалаца. Уколико се ради о писцу он чека да добије дозволу, а након тога чека да сви који су били пре њега и заврше са радом. Пошто је он добио дозволу да ради то значи да је пре њега или био писац који је завршио са радом или су били читаоци који можда још увек читају, писац мора да сачека читаоце. Приликом завршетка операције уписа писац мора да врати податке о броју читалаца који тренутно читају пошто се та информација више не налази у заједничком простору, у претходном кораку је одрађена операција *in("readCount", 0)* које је уклонила информације из заједничког простора. Ова информација се поставља на вредност 0 помоћу *out("readCount", 0)*. Давање дозволе следећем процесу унизу да се извршава постиже се помоћу *out("next", turn + 1)*.

Почетне вредности торки које представљају јединствен идентификациони број, број дозволе за извршавањем и број читалаца, *number*, *next*, *readcount*, респективно треба да буду постављене на вредности 0, 0 и 0 помоћу *out("number", 0)*, *out("next", 0)* и *out("readCount", 0)*. Овде треба нагласити да променљива *readcount* мора да буде постављена на вредност 0 док почетна вредност променљивих *number* и *next* не мора да буде 0 само је битно да ове две променљиве буду постављене на исту почетну вредност.

```
void reader() {
    int turn, num;
    while(1) {
        in("number", ?turn);
        out("number", turn + 1);
        in("next", turn);
        in("readCount", ?num);
        out("readCount", num + 1);
    }
}
```

```

        out("next", turn + 1);
        reading();
        in("readCount", ?num);
        out("readCount", num - 1);
    }
}

void writer() {
    int turn;
    while(1) {
        in("number", ?turn);
        out("number", turn + 1);
        in("next", turn);
        in("readCount", 0);
        writing();
        out("readCount", 0);
        out("next", turn + 1);
    }
}

void initialize () {
    int i;
    out("number", 0);
    out("next", 0);
    out("readCount", 0);
    for (i=0; i<10; i++) {
        eval (reader ());
        eval (writer ());
    }
}

```

Претходно решење би се могло додатно убрзати уколико писац не би прво дохватао број читалаца из простора торки, а на крају га враћао, већ уколико би га само читao.

```

void writer() {
    int turn;
    while(1) {
        in("number", ?turn);
        out("number", turn + 1);
        in("next", turn);
        rd("readCount", 0);
        writing();
        out("next", turn + 1);
    }
}

```

97

## Проблем избора

Решити проблем избора једне особе бацањем новчића користећи библиотеку C-Linda.

### Решење

Проблем избора описује проблем итеративне обраде података. Код овог проблема процеси прво генеришу нову вредност, онда ту вредност шаљу свим суседима, чекају да од свих суседа добију њихове вредности и на крају обављају рачунање на основу примљених вредности. Ово је потребно урадити у истој итерацији, то јест потребно је постићи да сви процеси прелазе из итерације у итерацију истовремено. Проблем се може решити користећи „јардигими ћулсирања“ (Heartbeat paradigm) код које постоји синхронизације на баријери у свакој итерацији.

На почетку сви процеси постављају да се ради о почетној итерацији 0. Процес одабира се обавља у петљи са изласком на дну. Петља се завршава у случају да одабран процес који је поставио новчић на супротну страну у односу на преостала два процеса. У првом кораку итерације се увећава редни број итерације и баца се новчић, *coin = flip()*, што се може симулирати користећи случајне бројеве, (*int((rand() \* 2) / RAND\_MAX)*). Након тога се овај податак ставља у заједнички простор два пута, да би могла да га покупе оба суседа. Податак, *out("result", id, iteration, coin)*, садржи идентификатор операције *"result"*, идентификатор процеса који баца новчић, редни број итерације на коју се односи израчунавање и вредност на коју је пао новчић. У следећем кораку се од оба суседа очекују њихови резултати бацања новчића за текућу итерацију. Када се прикупе сви подаци прелази се на рачунање да ли је дошло до избора. До избора се долази уколико нису сви новчићи пали на исту страну. Посматрани процес је победник уколико се страна на коју је пао његов новчић разликује од страна на које су пали новчићи преосталих корисника. Уколико је изабран победник поступак се прекида у супротном се прелази на следећу итерацију избора.

```
void Player(int id){
    int iteration;
    int coin, coinl, coinr;
    int end, winner;
    iteration = 0;
    do{
        iteration++;
        coin = flip(); // (int)((rand() * 2) / RAND_MAX);
        out("result", id, iteration, coin);
        out("result", id, iteration, coin);
        in("result", (id + 1) % 3, iteration, ?coinr);
        in("result", (id + 2) % 3, iteration, ?coinl);
        end = !(coin == coinl) && (coin == coinr)
            && (coinl == coinr);
        winner = (coin != coinl) && (coin != coinr);
    }while(!end);
}

void initialize() {
    eval(Player(0));
    eval(Player(1));
    eval(Player(2));
}
```

## 98 Проблем лифтова

Користећи библиотеку C-Linda написати програм који решава проблем путовања лифтом. Путник позива лифт са произвољног спрата. Када лифт стигне на неки спрат сви путници који су изразили жељу да сиђу на том спрту обавезно изађу. Након изласка путника сви путници који су чекали на улазак у ју лифт и кажу на који спрат желе да пређу. Тек када се сви изјасне лифт прелази даље. Није потребно оптимизовати пут лифта и путника.

### Решење

Проблем вожње лифтом је врста проблема која захтева поштовање извесног протокола приликом коришћења ресурса. Постоји посебан протокол коришћења ресурса, лифтова, који налаже да путник позове лифт пре него што оде на жељени спрат, али само у случају да лифт није позвао нико ко се на том спрту већ налази и чека лифт. Када лифт пристигне на неки спрат прво се пуштају сви путници који се налазе у лифту, а желе да сиђу на том спрту, и изађу из лифта. Тек тада се нови путници примају да уђу у лифт.

Део који се односи на кретање путника, *passenger(int id, x, y)*, као аргументе прима спрат на коме путник жели да уђе у лифт и спрат на коме путник жели да изађе. Уколико су та два спрата иста процесура се завршава. Уколико се полазни и долазни спратови разликују путник проверава колико путника чека на лифт на полазном спрту, *in("start", x, ?s1)*, у променљиву *s1* се смешта број путника који се већ налазе на том спрту и чекају лифт. Уколико је тај број једнак 0 ради се о првом путнику који жели да уђе на датом спрту и позива се лифт да дође на спрат *x*, *out("floor", x)*. Лифт позива само први путник који нађена неки спрат, јер нема потребе више пута звати лифт уколико га је претходни путник већ позвао. Увећава се број путника који на датом спрту чекају лифт. Након позивања лифта путник чека да лифт дође на спрат на коме се налази, *in("on", x)*. Када пристигне лифт путник учитава број путника који се налазе на том почетном спрту и број путника који жели да сиђе на крајњем спрту, *in("start", x, ?s1), in("stop", y, ?s2)*. Уколико се ради о првом путнику који жели да сиђе на одговарајућем спрту о томе обавештава лифт, *if(s2 == 0) out("floor", y)*. Након овога последњи путник који са датог спрата улази у лифт евентуално повлачи позив лифту да дође на текући спрт користећи наредбу *inp("floor", x)*. Овде се користи не блокирајућа наредба јер је сам лифт приликом доласка на дати спрт можда већ повукао позив за долазак. Разлог зашто је ово неопходно да се реализује је то што је лифт на одређени спрт могао да дође зато што је лифт позна са неког спрата, али и зато што је неко ко се већ налази у лифту тражио да изађе из лифта на том спрту. Када путник уђе у лифт умаљује број оних који чекају, увећава број оних који се налазе у лифту и враћа обавештење о томе на коме се спррату налази лифт.

Када је једном у лифту путник прелази на фазу чекања да лифт стигне на одредишни спрат да би могао да изађе. Овде је обезбеђено чекање на различитим променљивама онима који улазе у лифт и онима који из њега излазе. Када лифт пристигне на одредишни спрат путници из њега излазе. Операција изласка из лифта има предност у односу на операцију уласка у лифт. Када је путник изашао из лифта он умаљује број путника који се налазе у лифту а желе да сиђу на одређеном спрту. Поред овога последњи путник који жели да изађе из лифта на датом спрту евентуално повлачи позив лифту да дође на текући спрт из истих разлога као и приликом уласка у лифт.

Алгоритам по коме функционише лифт се састоји из неколико корака. У првом кораку се чека захтев да се лифт позове на одређени спрт, *x = getFloor()*. Решење није

прилагођено за оптимално кретање лифта нити за оптималан пут путника у лифту, већ према случајном принципу прелази на неки позвани спрат. Комплетно распоређивање и оптимизација кретања лифта би требао да се реализује унутар ове методе. У конкретном решењу је одабрано да лифт оде на неки од спратова на којима је тражен, били као полазна локација, било као крајња локација. Након одабирања спрата лифт поставља обавештење путницима да могу да изађу из лифта, **out("off", x)**. Лифт након овога чека да сви путници који су тражили да из лифта изађу на специфицираном спрату то и ураде, **in("stop", x, 0)**. Када сви ти изађу из лифта укида дозволу за излазак, а поставља дозволу за улазак у лифт. Након постављање дозволе за улазак у лифт чека да сви који су се налазили испред лифта на специфицираном спрату и уђу у лифт, **in("start", x, 0)**. Када сви уђу лифт укида дозволу улазака у лифт и прелази на следећу итерацију.

Приликом иницијализације потребно је поставити да се на сваком спрату испред лифта не налазе путници, као ни да се у лифту не налазе путници који желе да изађу на неком спрату.

```
void elevator(){
    int x;
    while(1){
        x = getFloor();

        out("off", x);
        in("stop", x, 0);
        in("off", x);
        out("stop", x, 0);

        out("on", x);
        in("start", x, 0);
        in("on", x);
        out("start", x, 0);
    }
}

int getFloor(){
    int x;
    in("floor", ?x);
    return x;
}

void passenger(int id, x, y){
    int s1, s2;
    if(x != y){
        in("start", x, ?s1);
        if(s1 == 0) out("floor", x);
        out("start", x, s1+1);

        in("on", x);
        in("start", x, ?s1);
        in("stop", y, ?s2);
        if(s2 == 0) out("floor", y);
        if(s1 == 1) in("floor", x);
        out("stop", y, s2+1);
        out("start", x, s1-1);
        out("on", x);
    }
}
```

```
    in("off", y);
    in("stop", y, ?s2);
    if(s2 == 1) inp("floor", y);
    out("stop", y, s2-1);
    out("off", y);
}

void initialize(){
    int floorNum = N;
    int i;

    for(i=0; i < floorNum; i++) {
        out("start", i, 0);
        out("stop", i, 0);
    }
    eval(elevator());
}
```

**99****Заједнички тоалет**

Постоји тоалет капацитета  $N$  ( $N > 1$ ) који могу да користе жене и мушкарци такав да се у исто време у тоалету не могу наћи и жене и мушкарци (*The Unisex Bathroom Problem*). Написати програм за жене и мушкарце који долазе до тоалета, користе га и напуштају га користећи библиотеку C-Linda.

**Решење**

Проблем заједничког тоалета је сличан проблему моста са једном коловозном траком, али се од њега разликује по томе што поседује и ограничење да се у једном тренутку у тоалету може појавити максимално  $N$  особа. Проблем је потребно решити за две улоге које могу да користе заједнички тоалет то јест за жене и за мушкарце. Пошто се овде ради о симетричном решењу даље ће бити разматрана само женска улога.

На почетку жена проверава ко има ексклузивно право коришћења тоалета, `in("direction", ?dir)`. Узимањем овог објекта из заједничког простора забрањује другим особама, било женама било мушкарцима, улазак у тоалет. Пошто се дати објекат не користи приликом изласка из тоалета особама које тренутно користе тоалет је дозвољено да из тоалета изађу. Уколико мушкарци тренутно имају ексклузивно право коришћења тоалета жена чека да број мушкараца који користе тоалет постане 0, `rd("numm", 0)`. Када број мушкараца који користе тоалет постане 0 мења се право коришћења тоалета и увећава се бројач жена које желе да уђу у тоалет, `"numw"`.

Када је дошла до тоалета жена чека на дозволу да уђе у њега јер тоалет има коначан капацитет  $N$ . Ово је постигнуто узимањем објекта дозволе, `in("bathroom ticket")`. Када заврши са коришћењем тоалета `usebathroom()` ослобађа се једно место у тоалету враћањем објекта дозволе, `out("bathroom ticket")`. Једино што преостаје је да се умањи број жена које желе да користе тоалет.

На почетку није било ни жена ни мушкараца у тоалету а право није било дато ни једној групи. Заједнички простор је иницијализован и са  $N$  дозвола коришћења тоалета.

```
void women(int i) {
    int num, dir;
    while(1) {
        in("direction", ?dir);
        if(dir == MEN) {
            rd("numm", 0);
        }
        in("numw", ?num);
        out("direction", WOMEN);
        out("numw", num + 1);

        in("bathroom ticket");
        usebathroom();
        out("bathroom ticket");

        in("numw", ?num);
        out("numw", num - 1);
    }
}
```

```
void men(int i){  
    int num, dir;  
    while(1) {  
        in("direction", ?dir);  
        if(dir == WOMEN){  
            rd("numw", 0);  
        }  
        in("numm", ?num);  
        out("direction", MEN);  
        out("numm", num + 1);  
  
        in("bathroom ticket");  
        usebathroom();  
        out("bathroom ticket");  
  
        in("numm", ?num);  
        out("numm", num - 1);  
    }  
}  
  
void initialize() {  
    int i;  
    out("direction", 0);  
    out("numw", 0);  
    out("numm", 0);  
    for(i=0; i<N; i++) out("bathroom ticket");  
    for(i=0; i<F; i++) eval(women(i));  
    for(i=0; i<M; i++) eval(men(i));  
}
```

Треба приметити да код овог решења не долази до изгладњивања страна које долазе. Ово је обезбеђено тако што чим се појави нека особа та особа узима објекат у коме се чува информација ко тренутно има ексклузивно право коришћења тоалета и чека док сви они који су приступили тоалету и не заврше са његовим коришћењем и тек тада враћа објекат. Са друге стране овакво решење може у неким случајевима да доведе до смањивања конкурентности. Зато би се и овде могла применити слична стратегија како код моста са једном коловозном траком код кога се смер мења сваки пут након што мост пређе одређен број аутомобила из једног истог смера а да је бар један аутомобил чекао да пређе мост са супротне стране.

100

## Проблем пијаних филозофа

Постоји група од  $N$  филозофа који проводи свој живот тако што наизменично филозофирају, чекају на пиће, и пију (*The Drinking Philosophers Problem*). Филозофи су тако распоређени да је по једна флаша са пићем постављене између суседних филозофа. У неком тренутку филозоф може да постане жедан. Жедном филозофу је потребно неколико суседних флаша да би направио коктел и почне да га пије. Избор пића зависи од тренутног расположења и може се разликовати од прилике до прилике. Када прикупи сва потребна пића филозоф започиње са њиховим испијањем које траје извесно време. Када се напије, филозоф враћа флаше да и други уживају, а он прелази на филозофирање. Написати програм који симулира понашање филозофа користећи библиотеку *C-Linda*.

## Решење

Прво решење:

За разлику од проблема филозофа који ручавају где је постојала међузависност само између два суседна филозофа сада зависност постоји између већег броја филозофа јер су филозофи повезани у граф. Овде је узето оптимистичко решење да филозоф може да пронађе сва пића која га интересују а која чувају његови суседи. Уколико успе да прикупи сва пића која су му потребна за коктел прави коктел, а ако не успе да прикупи сва пића онда она пића која је прикупио враћа и касније покушава поново. Треба приметити да број пића која су потребна за неки коктел може да буде мањи од броја суседа јер од неких суседа у некој итерацији неће узимати пића јер му нису неопходна за жељени коктел.

Програм за филозофа почиње тако што сазна ко су му суседи од којих може да узима пића,  $connections = init(id)$ , након чега прелази у бесконачну петљу у којој наизменично филозофира и пије. Филозоф прво мало филозофира,  $think()$ , па онда одлучи који ће коктел да пије  $cocktail = getNextCocktail(id, connections)$ . Када је сазнао који су све састанци неопходни за прављење коктела прелази у њихову набавку. Филозоф се врти у петљи све док не набави сва пића. Променљива  $allDrinks$  која сигнализира да ли су прибављена сва пића се поставља на вредност  $true$  јер филозоф можда хоће да прави кокtel или му не треба ни један састанак. Пролази кроз листу свих пића и узима она пића која су му неопходна,  $inp("drink", MIN(id, i), MAX(id, i))$ . Постоје се пиће налази између два филозофа узето је да идентификатор пића буде састављен од идентификатора филозофа између којих се налази. Прво се ставља идентификатор филозофа који има мању вредност,  $MIN(id, i)$ , па идентификатор који има већу вредност,  $MAX(id, i)$ . Уколико није пронашао сва пића филозоф враћа све што је до тог тренутка узео и понавља поступак набавке пића. Уколико је набавио све што му је неопходно за коктел онда га спрема и ужива у његовом испијању. Када заврши да испијањем коктела враћа сва пића у заједнички простор.

```
void DrinkingPhilosopher(int id) {
    Matrix connections;
    bool [] cocktail;
    int i, j;
    bool allDrinks;
    connections = init(id);
    while(1) {
        think();
        cocktail = getNextCocktail(id, connections);
        if (cocktail.length > 0) {
            inp("drink", MIN(id, i), MAX(id, i));
            if (allDrinks)
                break;
        }
    }
}
```

```

allDrinks = false;
while(!allDrinks){
    allDrinks = true;
    for(i = 0; i < n; i++){
        if(cocktail[i])
            allDrinks = allDrinks &&
                inp("drink",MIN(id,i),MAX(id,i));
        if(!allDrinks) break;
    }
    if(i != n){
        for(j = 0; j < i; j++){
            if(cocktail[j])
                out("drink",MIN(id,j),MAX(id, j));
        }
        pause();
    }
}
drink();
for(i = 0; i < n; i++){
    if(cocktail[i])out("drink", MIN(id, i), MAX(id, i));
}
}

```

Друго решење:

Полазећи од разматрања датих на крају задатка 4 и протокола заснованих на графовима да би се избегло међусобно блокирање филозофа потребно је обезбедити да филозофи увек узимају пића у истом редоследу. За разлику од претходног решења код овог решења филозоф не враћа пића која је узео када нађе на прво које му недостаје већ се блокира док не добије сва која је тражио. Ово решење обезбеђује да се међусобно искључују филозофи који траже исто пиће, а да притом постоји барем један који може добити сва пића.

```

void DrinkingPhilosopher(int id) {
    Matrix connections;
    bool [] cocktail;
    int i, j;
    bool allDrinks;
    connections = init(id);
    while(1) {
        think();
        cocktail = getNextCocktail(id, connections);
        for(i = 0; i < n; i++) {
            if(cocktail[i])in("drink", MIN(id, i), MAX(id, i));
        }
        drink();
        for(i = 0; i < n; i++) {
            if(cocktail[i])out("drink", MIN(id, i), MAX(id, i));
        }
    }
}

```

Разматрања која су важила у задатку 4 о изгладњивању и праведности важе и у овом случају.

101

## Израчунавање интеграла

Написати програм који израчунава интеграл функције на интервалу  $XMIN$ ,  $XMAX$  у  $N$  корака по принципу „торбе послова“ (*Bag of Tasks*) користећи библиотеку *C-Linda*.

### Решење

Обрада по принципу „торба послова“ је тако реализована да се цео посао израчунавања интеграла на интервалу од  $XMIN$  до  $XMAX$  издељен на  $N$  делова мањег обима који ће се распоређивати расположивим процесима.

```
void worker(int i) {
    double left, right, data;
    while(1) {
        out("getTask", i);
        in("getData", i, ?left, ?right);
        if(left > right) break;
        data = calc(left, right);
        out("putResult", data);
    }
}
void bag(double XMIN, double XMAX, int N){
    double dx, x;
    int i, numNode;
    x = XMIN;
    dx = calcDX(XMIN, XMAX, N);
    for(i = 0; i < N; i++){
        in("getTask", ?id);
        out("getData", id, x, x + dx);
        x = x + dx;
    }
    in("numNode", ?numNode);
    for(i = 0; i < numNode; i++){
        in("getTask", ?id);
        out("getData", id, 0, -1);
    }
}
void collector(int N){
    double F, data;
    int i;
    F = 0;
    for(i = 0; i < N; i++){
        in("putResult", ?data);
        F = F + data;
    }
    out("result", F);
}
void initialize(){
    int i, numNode, N;
    double XMIN, double XMAX;
```

```
...
out("numNode", numNode);
eval(bag(XMIN, XMAX, N));
eval(collector());
for(i = 0; i < numNode; i++) eval(worker(i));
}
```

102

## Проблем тела у гравитационом пољу

У свемиру постоји  $N$  небеских тела која међусобно интерагују својим гравитационим пољем ( $N$  Body Gravitational Problem). Потребно је решити овај проблем користећи торбу послова за дохватање послова и синхронизацију на баријери у свакој итерацији и библиотеку C-Linda.

## Решење

Прво решење:

Подела послова коришћењем торбе послова се састоји у подели неког послова на већи број мањих, независних, послова које је потребно одредити и чије је резултате потребно прикупити. Како би се ово остварило издвојене су следеће улоге: подела послова на већи број мањих послова (Bag), обрада мањих послова (Worker) и прикупљање резултата (Collector). Проблем кретања тела је потребно обављати у већем броју итерације, а прикупљање резултата се обавља на крају сваке итерације.

```
void worker(int id){
    int i, iteration, it, start, end;
    body space[100];
    body result[100];
    out("get task", id);
    in("task", id, ?iteration, ?start, ?end);
    it = iteration;
    while(start > 0){
        if(it != iteration){
            rd("space", ?space);
            it = iteration;
        }
        for(i = start; i < end; i++){
            result[i - start] = calculate(space, i);
        }
        out("result", start, end, result);
        out("get task", id);
        in("task", id, ?iteration, ?start, ?end);
    }
}

void bag(){
    int i, id, iteration, it, start, end, step, n, num, numPlanets;
    iteration = 0;
    start = 0;
    end = step;
    rd("num workers", ?n);
    rd("num planets", ?numPlanets);
    rd("num iterations", ?num);
    step = n == 0? 1 : numPlanets / 3*n; //empirical
    while(iteration < num){
        in("get task", ?id);
        out("task", id, iteration, start, end);
```

```
start = end + 1;
end = (end + step) < numPlanets? end + step : numPlanets;
if(start > numPlanets){
    in("next iteration");
    start = 0;
    end = step;
    iteration++;
}
}
for(i = 0; i < n; i++){
    in("get task", ?id);
    out("task", id, -1, -1, -1);
}
out("end");
}

void collector(){
    int i, iteration, num, start, end, planets, numPlanets;
    body space [100];
    body result[100];
    body temp[100];
    iteration = 0;
    planets = 0;
    rd("num iterations", ?num);
    rd("num planets", ?numPlanets);
    while(iteration < num){
        in("result", ?start, ?end, ?result);
        planets = planets + end - start + 1;
        for(i = start; i < end; i++){
            space[i] = result[i - start];
        }
        if(planets == numPlanets){
            in("space", ?temp);
            out("space", space);
            iteration++;
            out("next iteration");
            planets = 0;
        }
    }
}

void initialize(){
    int num, numPlanets, n, i;
    body space[100];
    ...
    out("space", space);
    out("num iterations", num);
    out("num planets", numPlanets);
    out("num workers", n);
    eval(bag());
    eval(collector());
    for(i = 0; i < n; i++){
        eval(worker(i));
    }
    in("end");
}
```

```
    in("space", ?space);  
}
```

Друго решење:

У односу на претходно решење у овом решењу су обједињени подела посла (*Bag*) и прикупљање резултата (*Collector*), док је сама обрада (*Worker*) остала иста као у претходном решењу.

```
void bag(){  
    int i, id, iteration, start, end, step, n;  
    int num, planets, numPlanets;  
    body space[100];  
    body result[100];  
    body temp[100];  
  
    iteration = 0;  
    start = 0;  
    end = step;  
    rd("num workers", ?n);  
    rd("num planets", ?numPlanets);  
    rd("num iterations", ?num);  
    step = n == 0? 1 : numPlanets / 3*n;  
    while(iteration < num){  
        in("get task", ?id);  
        out("task", id, iteration, start, end);  
        start = end + 1;  
        end = (end + step) < numPlanets? end + step : numPlanets;  
        if(start > numPlanets){  
            planets = 0;  
            while(planets != numPlanets){  
                in("result", ?start, ?end, ?result);  
                planets = planets + end - start + 1;  
                for(i = start; i < end; i++){  
                    space[i] = result[i - start];  
                }  
            }  
            in("space", ?temp);  
            out("space", space);  
            start = 0;  
            end = step;  
            iteration++;  
        }  
        for(i = 0; i < n; i++){  
            in("get task", ?id);  
            out("task", id, -1, -1, -1);  
        }  
        out("end");  
    }  
  
    void initialize(){  
        int num, numPlanets, n, i;
```

```
body space[100];
...
out("space", space);
out("num iterations", num);
out("num planets", numPlanets);
out("num workers", n);
eval(bag());
for(i = 0; i < n; i++){
    eval(worker(i));
}
in("end");
in("space", ?space);
}
```

Треба приметити да се код овог решења резултат комплетног израчунавања следеће итерације може наћи у простору торки. Код овог решења се са прикупљањем обрађених података креће тек када се заврши фаза поделе података. Уколико је просто торки ограничен онда оваква решења треба избегавати и у заједничком простору чувати што мање података како би се остварио оптималан рад система.

## Програмске нити

Код парадигме програмирања помоћу *програмских нити* (*multithreading*) координациони простор и рачунски простор нису раздвојени. Главне градивне јединице програма код овог координационог модела јесу програмске нити. Програмске нити су независни програмски токови који се конкурентно извршавају на датом рачунарском систему. Синхронизација и координација рада нити врши се помоћу неких од синхронизационих и комуникационих парадигми (семафори, монитори, прослеђивање порука, итд.).

Као подршка нитима потребно је да постоји неки извршни модел који ће подржавати њихово извршавање. Основни извршни модел који подржава оперативни систем су процеси; процеси оперативног система се још називају и „тешки процеси“ (*heavy-weight processes* - *HWP*). Контекст *HWP* процеса је прилично гломазан и садржи многе елементе који су неопходни да би се подржао рад више независних програма унутар оперативног система. Време потребно за промену контекста код смене процеса је веома велико. То је последица тога што процеси не деле адресни простор и друге ресурсе система; контекст сваког процеса који се мора памтити стога садржи много вредности.

Већина оперативних система разликује процесе и нити, јер нити не захтевају тако обиман контекст као процеси. Нити које су део истог програма могу делити исти адресни простор (меморијску табелу) и друге елементе контекста извршавања програма, тако да приликом промене контекста код нити није потребно трошити толико времена као код процеса. Нити оперативног система су извршни модел са мање обимним контекстом од процеса и обично је баратање нитима посао језгра (*kernel*) оперативног система. Нити оперативног система називају се још и „лаки процеси“ (*light-weight processes* - *LWP*).

Конечно, постоје нити које корисник дефинише у свом програму. У зависности од тога какво је пресликовање (*mapping*) корисничких нити на нити оперативног система, постоји подела на тзв. *user-space* и *kernel-space* имплементације. Код *user-space* имплементација сва контрола и управљање нитима врше се на нивоу корисничког програма у оквиру једне нити оперативног система. Промена контекста се своди на чување и рестаурирање вредности процесорских регистара на стеку, што се изводи брзо и транспарентно за језгро оперативног система коришћењем корисничких рутина и библиотека. Мана *user-space* имплементација је у томе што једна корисничка нит може трошити своје време процесора и изгладњивати остale. Такође, ове имплементације не могу користити погодности извршавања на мултипроцесорским архитектурама јер ОС може да бара само са нитима оперативног система, тако да све корисничке нити унутар једне такве нити увек бивају алоциране заједно на исти процесор. Конечно, ако се једна корисничка нит блокира на улазно-излазним операцијама оперативног система, све остале корисничке нити такође ће бити заустављене. То се догађа јер оперативни систем цео скуп корисничких нити унутар *HWP* или *LWP* (већ у зависности од имплементације језгра) види јединствено. Неки од ових проблема се могу и заобиди, али то онда компликује писање програма.

Код *kernel-space* имплементација промену контекста приликом смењивања нити ради сам оперативни систем на нивоу свог језгра, транспарентно за сам програм. Предност је што само језгро ОС води рачуна о томе да свака нит добије довољно процесорског времена, што је аутоматски могуће искористити извршавање на мултипроцесорским системима и што блокирање на улазно-излазним операцијама није проблем. Промена контекста је нешто спорија него код *user-space* нити, али не значајно.

Неке имплементације користе оба приступа. Постоји подела према моделу пресликовања корисничких нити на нити оперативног система:  $M:N$  пресликовање значи да се  $M$  корисничких нити пресликава на  $N$  нити оперативног система, а постоје и  $M:1$  и  $1:1$  пресликовања. У зависности од примене, корисник може изабрати жељену имплементацију.

## **Java**

Java подржава рад са више парадигми конкурентног програмирања. Она обухвата парадигме за програмирање помоћу дельених променљивих (монитори су инхерентни део језика, постоје библиотеке за рад са семафорима, итд.) и дистрибуирано програмирање (библиотеке за рад са удаљеним позивима процедура, прослеђивањем порука, итд.). Међутим, овде ће пажња превасходно бити посвећена координационом моделу језика Java - програмирању помоћу програмских нити.

Рад са нитима у језику Java имплементиран је на нивоу корисничких нити.

### **Креирање и контрола рада нити**

Свака класа се може декларисати као нит на два начина. Први начин је простим наслеђивањем класе *Thread* која је уgraђена у језик и редефинисањем методе *run*:

```
import java.lang.*;
public class MyThread extends Thread{
    public void run() {
        ...
    }
}
```

Други начин за креирање нити јесте имплементирањем интерфејса *Runnable* и дефинисањем методе *run*. Треба приметити да класа *Thread* имплементира интерфејс *Runnable*.

```
import java.lang.*;
public class MyThread implements Runnable{
    Thread T;
    public void run() {
        ...
    }
}
```

Нит се инстанцира са:

```
myThread = new MyThread(...);
```

а покреће са:

```
myThread.start();
```

Нит се зауставља сама када дође до краја процедуре *run()*.

Након покретања нити, нит која ју је покренула наставља са радом упоредо са њом. За чекање да се извршавање нити заврши, користи се:

```
myThread.join();
```

Наредба

```
myThread.yield();
```

представља експлицитни захтев за преузимање процесора од стране друге нити, који се не мора испунити (зависи од имплементације).

Нит се може суспендовати на временски период  $t$  са:

```
myThread.sleep(t);
```

Приликом суспендовања нит не напушта ексклузивно право над мониторима које нит држи, а тренутак буђења зависи од тога како су распоређене друге нити као и од доступности процесора на коме се дата нит извршава.

Треба приметити да ни метода *sleep* нити *yield* немају никакву синхронизацију. Ово значи да преводилац не мора привремене променљиве које су у регистрима да смести у дељену меморију приликом позива ових метода, нити да их поново чита приликом повратка. Ово значи да уколико би нека нит чекала да нека друга нит постави дељењу променљиву (која није нестабилна - *volatile*) на одговарајућу вредност у следећем програмском сегменту би дошло до појаве бесконачне петље

```
while (!done)
    sleep(1000);
```

Приликом рада са нитима треба водити рачуна о томе које се методе класе *Thread* позивају. Методе *stop*, *suspend*, *resume* и *destroy* су застареле (*deprecated*) и треба их избегавати јер могу систем да доведу у неконзистентно стање. На пример позивом методе *stop* се ослобађају сви монитори које је дата нит држала. Током приступа било ком монитору тај монитор у неком тренутку може да буде у неконзистентном стању док се постављају вредности променљивих. Уколико је монитор био у неконзистентном стању позивањем методе *stop* нит која је користила монитор прекида његово коришћење и монитор остаје у том неконзистентном стању тако да све остале нити које касније приступе том монитору затичу то неконзистентно стање. Уместо позивања ове методе за заустављање нити приликом пројектовања нити би било добро да сама нит проверава променљиве које означавају да ли та нит треба и даље да се извршава или не. Исто тако би требало у случајевима да се нит блокира на неком услову користити механизам прекида, *interrupt*, како би се дата нит пробудила.

## Синхронизација

Синхронизација је у програмском језику Java реализована користећи мониторе. Сваком објекту је придружен монитор који нека нит може да закључча или откључча уколико држи његов кључ. У једном тренутку само једна нит може држати тај кључ неког објекта односно држати закључан неки монитор. Све остале нити које покушавају да закључчају дати монитор се блокирају док не добију право да закључчају дати монитор. Једна нит може исти објекат закључчати већи број пута, при чему једно откључавање поништава ефекте једног закључавања. Постоје два основна типа закључавања. Један је закључавање на нивоу блока што би одговарало регионима, и друго је закључавање на нивоу методе што би одговарало мониторима.

## Синхронизација на нивоу блока

```
synchronized (<референца на објекат>) { ... }
```

закључава приступ датом објекту користећи референцу датог објекта. Прво се дохвата референца објекта. Након тога се покушава са закључавањем датог објекта и не наставља се даље са радом док дата нит не успе са закључавањем датог објекта. Када се објекат закључча нит улази у блок и од свих нити које су покушале да приступе датом објекту једина има ексклузивно право приступа датом објекту. Када се заврши извршавање датог блока аутоматски се откључчава приступ датом објекту. Ово је еквивалентно коришћењу критичних региона.

## Синхронизација на нивоу методе

```
public class SomeClass{
    public synchronized void f(int value) {
```

```

    }
}
}
```

автоматски закључава приступ датом објекту чим се метода позове. Тело методе започиње тек када нит успе са закључавањем датог објекта. Ово значи да процедура *f* не може да се изврши истовремено ако је две нити позову преко истог објекта (али може ако се позива преко два различита објекта исте класе!). Само једна нит добија право приступа монитору коме припада *synchronized* процедура, док остале нити морају да чекају да претходна нит изађе. Када се заврши извршавање дате методе аутоматски се откључава приступ датом објекту.

Треба приметити да само једна нит добија ексклузивно право приступа инстанци класе користећи синхронизација, *synchronized*, било да је тај објекат закључан на нивоу блока или на нивоу методе. Веза између ова два типа синхронизације је таква да се код синхронизације на нивоу методе на почетку закључава дати објекат, *this*. Синхронизација на нивоу методе би имала исти ефекат, не нужно и реализацију, као:

```

public class SomeClass{
    public void f(int value) {
        synchronized(this) {
            ...
        }
    }
}
```

Такође треба приметити да објекту истовремено може да приступа више нити али тако да максимално једа нит приступа користећи синхронизовани приступ. Не синхронизовани приступ из више нити може довести до проблема конзистентности и утвркивања.

Уколико се ради о методи инстанце класе онда се закључава монитор који је придружен датом објекту. Уколико се ради о статичкој методи, *static*, онда се закључава монитор који је придружен *Class* објекту који представља класу у којој је метода дефинисана:

```

public class SomeClass{
    public static synchronized void f(int value) {
        ...
    }
}
```

Ово би имало исти ефекат, не нужно и исту реализацију, као:

```

public class SomeClass{
    public static void f(int value) {
        synchronized (SomeClass.class) {
            ...
        }
    }
}
```

Метода *wait* прекида извршавање нити и ставља је у стање чекања; привремено се скира забрана другим нитима да приступају монитору. Нит се буди тек када нека друга нит пошаље *notifyAll()* или *notify()* наредбу за циљни објекат. Постоје три облика ове методе:

```
wait();
wait(tmillisec);
wait(tmillisec,tnanosec);
```

*wait()* чека неограничено, а у друга два случаја се чека одређено време да стигне нотификација да сме да се приступи циљном објекту. Уколико је интервал у мили секундама негативан, *tmillisec*, или је аргумент у нано секундама, *tnanosec*, изван интервала 0-9999999 емитује се изузетак *java.lang.IllegalArgumentException*. Постављање вредности 0 за *tmillisec* и *tnanosec* је еквивалентно позиву методе *wait()*. Блокирана нит се може пробудити и уколико је нека друга нит над датом нити позвала методу *interrupt*, али и у неким другим специјалним ситуацијама које зависе од имплементације виртуелне машине.

#### Метода

```
notifyAll();
```

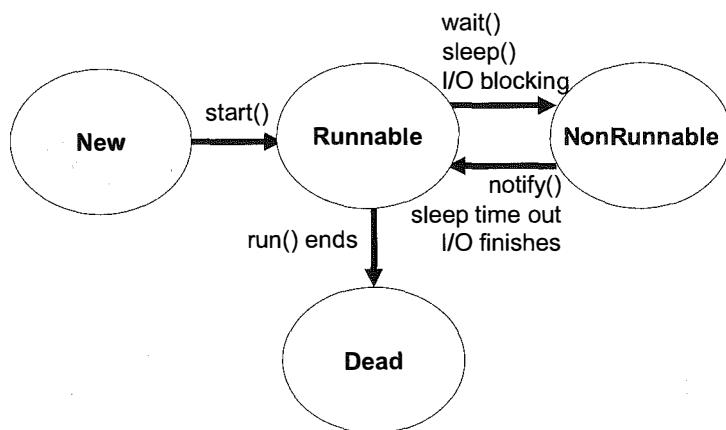
шаље нотификацију свим нитима које чекају да приступе циљном објекту помоћу методе *wait()* да то могу учинити. Ако има више таквих нити, бира се једна, а остale опет чекају. Метода *notify()* ради исто што и *notifyAll()*, али нотификује само једну нит. Језиком није специфицирано која се нити нотификује.

Методе (*notify*, *notifyAll*, и *wait*) је дозвољено позивати искључиво уколико је претходно узето ексклузивно право приступа објекту. Ексклузивно право приступ се добија у синхронизованом (*synchronized*) блоку над датим објектом који је настао или позивом синхронизоване мониторске методе или директном синхронизацијом над објектом. Уколико ове методе нису позване из синхронизованог блока за дати објекат долази до појаве изузетка:

```
java.lang.IllegalMonitorStateException
```

Синхронизација на објекту је реализована тако да се приликом позива методе над неким објектом та инстанца класе понаша у складу са *Signal and Continue* дисциплином. Нит која изврши *notify* или *notifyAll* методу класе *Object* задржава ексклузивну контролу над датом инстанцом објекта. Ексклузивно право приступа објектну се напушта или када се заврши *synchronized* блок или када се дата нит блокира на методи *wait* ове инстанце објекта. Треба приметити да постоје разлике између монитора са *Signal and Continue* дисциплином разматраних раније и монитора у програмском језику Java. У програмском језику Java блокирања се обавља искључиво на инстанцом над којом се обавља и синхронизација, нема језичке подршке за већим бројем условних променљивих.

Поред коришћења кључне речи *synchronized* за синхронизацију се могу користити и друге технике које се заснивају на читању и упису у нестабилне променљиве које су означене као *volatile* или користећи класе унутар *java.util.concurrent* пакета.



Слика 18. Дијаграм животног циклуса нити

Животни циклус нити је приказано на датом дијаграму стања (слика 18).

На почетку се нит налази у стању које означава да је креирана, *New*, али се још увек не извршава. У овом стању инстанца нити се не разликује од инстанци других класа, односно још увек нема свој ток контроле.

Нит из почетног стања позивом методе *start*, прелази у стање у коме може да се извршава, *Runnable*. Позивом методе старта контактира се виртуелна машина која креира засебан ток контроле за дату нит. Приликом позива ове методе алокирају се ресурси потребни за извршавање ове нити. Када се поступак алокације изврши нит се пребачује у листу спремних нити и у неком тренутку се позива метода *run*. У зависности од стања виртуелне машина ова нит може да се извршава, али може и да чека у реду спремних нити док се нека друга нит извршава. Метода *start* може да се позива само једном приликом покретања нити. Уколико се покуша позивање ове методе над инстанцом која је већ покренута долази до појаве изузетка:

`java.lang.IllegalThreadStateException`

Нит из стања у коме може да се извршава прелази у стање у коме не може да се извршава, *NonRunnable*, може прећи у више случајева. Први је случај када се позове метода *wait* над неким објектом над којим дата нит држи кључ, *synchronized*. Други је ако је позвана метода *sleep* за одговарајући интервал времена. Трећи случај је када је нит блокирана на неко улазно/излазно операцији.

Нит из стања у коме не може да се извршава враћа у стање у коме може да се извршава у више случајева. Уколико је нит била блокирана због позива метода *wait* па је нека друга нит позвала методу *notify/notifyAll* над објектом над којим је дата нит блокирана или је истекао специфицирани интервал времена. Уколико је код методе *sleep* истекао специфицирани интервал времена, или уколико се завршила дата улазно/излазна операција. Поред ових случајева нит може у извесним случајевима прећи из стања у коме не може да се извршава у стање у коме може да се извршава уколико је нека друга нит над датом нити позвала методу *interrupt*. Уколико је нит била блокирана на метод *wait*, *join* или *sleep* онда ће бити пробуђена и биће емитован *InterruptedException* изузетак. Уколико је нит била блокирана на улазно излазним

операцијама остаће и даље блокирана. Изузетак представља рад са каналима комуникације који користе класе *InterruptibleChannel* и *Selector*.

На крају извршавања методе *run* нит прелази у стање да постаје мртва нит, *Dead*. Из овог стања нит не може да изађе.

Уколико се нит и обавести (*notify/notifyAll*) и прекине (*interrupt*) док чека блокира на методи *wait* могућа су два равноправна сценарија даље рада блокиране нити. У првом нит нормално изађе из методе *wait* али има постављен индикатор да је дошло до прекида (*Thread.interrupted*), док се у другом нит пробуди због емитовања изузетка *InterruptedException*.

### Меморијски модел

За меморију коју могу да деле различите нити се каже да је *дељена меморија*. Меморијски модел програмског језика Java описује могућа понашања програма у време извршавања приликом читања и уписа у променљиве које се налазе у дељеној меморији. Сва поља инстанци класа, статичка поља и низови елемената се смештају у дељену меморију и се означавају као променљиве. Локалне променљиве, формални параметри метода и изузетци се никада не деле између нити и на њих не утичи параметри меморијског модела. Редослед приступа променљивама у дељеној меморији може бити другачији од редоследа који је специфициран изворним кодом, а имплементацијама је остављена слобода да оптимизују извршавање програма и генеришу код све док је тај код у складу са меморијским моделом. Програмским преводиоци и процесори могу начине различите да обаве оптимизацију кода узимајући у разматрање само програмски сегмент који извршава једна нит како би га учинили што бржим, све у складу са семантиком програмског језика Java. Уколико се не води рачуна о правилној синхронизацији резултати извршавања програмског кода могу бити забуњујући.

На пример, посматра се програмски сегмент у коме нема синхронизације, код кога су *I1* и *I2* локалне променљиве, а променљиве *A* и *B* се налазе у дељеној меморији и имају почетну вредност 0.

Нит 1 – оригинални код

```
11 = A;
B := 1;
```

Нит 2 – оригинални код

```
12 = B;
A = 2;
```

Пошто програмски преводилац приликом превођења програмског сегмента који ће извршавати једна нит не мора да посматра друге програмске сегменте које ће извршавати друге нити резултат превођења и извршавања може бити неочекиван. У датим програмским сегментима након њиховог завршетка као неочекивани резултат се може појавити да *I1* постане 2 а *I2* постане 1. Програмски преводилац сасвим валидно може да генерише програмски код у коме нит 1 прво додељује вредност 1 дељеној променљиви *B*, па тек онда чита дељену променљиву *A* јер су ова два приступа што се датог сегмента тиче потпуно независна. Исто тако нит 2 може прво да додели вредност 2 променљиви *A*, па тек онда да прочита променљиву *B* јер су и ова два приступа што се датог програмског сегмента тиче независни. Ово би произвео да преведени код одговара следећем коду:

Нит 1 – преведени код

```
B = 1;
11 = A;
```

Нит 2 – преведени код

```
A = 2;
12 = B;
```

У овако преведеном коду је могућ редослед извршавања:

```
нит 1: B = 1;  
нит 2: A = 2;  
нит 2: 12 = B; // 12 = 1 и  
нит 1: 11 = A; // 11 = 2
```

Треба приметити да овакав код није правилно синхронизован јер једна нит обавља упис, а онда чита променљиву друге нити, а редослед читања и уписа у овом случају није био гарантован.

Меморијским моделом се специфицирају акције унутар једне нити, *inter-thread action*, које се могу детектовати и на које могу утицати друге нити:

- Читање променљиве која није означенa сa *volatile*.
  - Упис у променљиву која нијe означенa сa *volatile*.
  - Синхронизационe акцијe:
    - Читање променљиве која је означенa сa *volatile*.
    - Упис у променљиву која је означенa сa *volatile*.
    - Закључавање монитора.
    - Откључавање монитора.
    - Прва (синтетичка) и последња акција у нити.
    - Акције које покрећe нит или детектују да је нит завршена.
  - Спољашње акције које се могу посматрати изван извршења и које имају резултат који је заснован на окружењу изван извршења.
  - Дивергентне акције нити које се налазе у бесконачној петљи.
- Меморијским моделом се дефинише „догодило се пре“ релација (*happens-before relationship*). Код ове релације важи:
- Свака акција у нити која се налази пре ове релације се дешава пре сваке акцију у овој нити која долази касније током извршавања програма.
  - Откључавање монитора се дешава пре следећег закључавања тог монитора.
  - Упис у нестабилну променљиву, означену сa *volatile*, се догађа пре сваког наредног читања те променљиве. Ово значи да уколико нит *A* уписује вредност у неку *volatile* променљиву и уколико нит *B* након тога чита ту променљиву онда види ту нову вредност. Поред овога и акције нити *A* које су претходиле приступу датој променљиви су такођe видљиве нити *B*. Ово значи да се приликом превођења и извршавања програма нећe мењати редослед по коме инструкције приступају *volatile* променљивама у односу на редослед тих инструкција у извornom коду. Инструкцијама које се налазе пре или после приступа *volatile* променљивама се приликом превођења и извршавања може мењати редослед, али се међусобно не мешају. Оне инструкције које су биле пре остају пре и оне које су биле после остају после.
  - Позивање *start()* методе неке нити се увек догађа пре покретања те нити.

- Све акције унутар нити се увек обављају пре успешног завршетка методе `join()` те нити.
- Подразумевана иницијализација објекта се обавља пре било које друге акције програма.

Уколико две акције деле ову релацију, оне не морају обавезно да се појаве у том поретку према било ком другом коду према коме не деле ову релацију.

У програмском језику Java постоје акције које се обављају недељиво/атомски:

- Операције читања или уписа у променљиве већине простих типова је атомска операција. Изузети су променљиве типа `long` и `double`.
- Операције читања или уписа у све променљиве декларисане као `volatile`, укључујући и променљиве типа `long` и `double`.
- Приступ за читање и упис `volatile` променљива ствара релацију уређеног приступа, *happens-before relationship*.

Упис и читање у променљиву означену са `volatile` има сличан ефекат што се конзистенције меморије тиче као откључавање и закључавање монитора, али не захтевају закључавање са међусобним искључивањем. Треба приметити и разлике између ова два приступа. Монитори постоје само за објектне типове података, док променљиве означене са `volatile` могу бити и објектни и прости типови података, а такође и објекти са `null` вредношћу. Монитори омогућавају блокирајући приступ објекту, док `volatile` то не омогућава. Описан меморијски модел се примењује почев од верзије 1.5. Коришћење атомског приступа и `volatile` променљивих може да буде ефикасније од приступа користећи синхронизоване блокове, али захтева више пажње како би се избегле грешке.

103

Семафор

Написати класу који реализује функционалност семафора у програмском језику Java.

## Решење

Прво решење

```
public class Semaphore {
    private int s;

    public Semaphore(int val) {
        s = val;
    }

    @Deprecated
    public synchronized void initS(int i) {
        s = i;
    }

    public synchronized void waits() {
        while (s <= 0) {
            try {
                wait();
            } catch (InterruptedException e) {
            }
        }
        s = s - 1;
    }

    public synchronized void signals() {
        s = s + 1;
        notify();
    }
}
```

Приликом рада са семафорима потребно је дефинисати две операције *wait(s)* и *signal(s)* које се обављају атомски.

Семафорска метода *wait* декрементира вредност семафора ( $s=s-1$ ), под условом да вредност семафора *s* задовољава услов  $s>0$ ; позивајући процес потом наставља са радом. Ако услов  $s>0$  није задовољен, позивајући процес се блокира док се тај услов не испуни. Како би се ова метода реализацивала у програмском језику Java потребно је урадити следеће: Интерна семафорска променљива *s* постаје поље класе *Semaphore*. Пошто метода *wait* већ постоји у основној класи *Object* како не би дошло до забуне која се метода позива потребно ју је преименовати. Одабран је назив *waitS()*. Како би се обезбедило атомско извршавање метода је декларисана као синхронизована метода (*synchronized*). Синхронизација означава да када једна нит приступи инстанци класе над којом се синхронизација обавља (синхронизација на објекту или синхронизована метода) та нит узима кључ над датим објектом. Узимање кључа омогућава датој нити ексклузивно право приступа објекту, а онемогућава било којој другој нити да приступи датој инстанци све док дата нит која има ексклузивно право приступа то право не напусти и врати кључ који држи над датом инстанцом објекта. Треба приметити да се

ово ограничење односи само на друге нити које покушавају да приступе датом објекту користећи *synchronized* приступ, уколико нит не тражи ексклузивно право приступа објекту онда може приступити том објекту чак и када неко држи ексклузивно право приступа. Какве ће последице ово произвести зависи од тога да ли ове нити приступају истим деловима. Напуштање ексклузивног права приступа објекту се постиже на два начина: завршетком *synchronized* блока и блокирањем на методи *wait()* класе *Object*. Како је код семафора потребно омогућити да се нит блокира све док услов не постане испуњен у методу *waitS* је могуће поставити петљу са супротним условом на коме се нит блокира све док не постане испуњен. Треба приметити да метода *wait()* класе *Object* може да емитује изузетак типа *InterruptedException* када неко позове методу *interrupt()* над датом нити па је потребно ухватити и обрадити овај изузетак. У конкретном случају обрада се своди на поновну проверу услова.

Семафорска метода *signal(s)* инкрементира вредност семафора ( $s=s+1$ ), под условом да нема блокираних процеса на том семафору; ако има блокираних процеса, онда се један од њих деблокира. У складу са преименовање претходне методе и ова метода је преименована у *signalS()*. Како би се омогутило атомско извршавање метода је декларисана као синхронизована метода (*synchronized*). Како класа *Object* не пружа информацију о броју блокираних нити, а приликом блокирања у методи *waitS* није увећана променљива којом би се бројале блокиране нити, овде је реализовано да се увек позива метода *notify()*. Метода *notify()* једну од блокираних нити из листе нити блокираних на датом објекту пребацује у улазни ред где са другим нитима чека ексклузивно право да приступи објекту и провери свој услов. Треба приметити да ако је постојала блокирана нит која је чекала и методом *notify()* се дата нити пребаци у улазни ред то не значи да ће та нит прва наредна добити ексклузивно право приступа објекту. У програмском језику Java редослед по коме се пуштају нити кроз улазни ред није гарантова. Синхронизовани приступ објектима се понаша на исти начин као и монитори који имају дисциплину *Signal and Continue* и само једну условну променљиву на којој нити могу да се синхронизују.

Иницијализација интерне семафорске променљиве се може обавити на два начина. Први начин је коришћењем посебне синхронизоване методе *initS*, а други начин је у самом конструктору дате класе. Треба приметити да у овом другом случају није неопходно да конструктор буде синхронизован јер ни један друга нит неће приступити инстанци класе док се конструктор не заврши.

Треба приметити да у методи *signalS()* није доволично да се обавештавање блокираних нити обавља само ако је интерна променљива била једнака нули.

```
public class Semaphore_With_Deadlock {
    ...
    public synchronized void signalS() {
        s = s + 1;
        if(s == 1)
            notify();
    }
}
```

У случају да на семафору стоје две нити блокиране у методи *waitS()* на блокирајућој методи *wait()* и тада наиђу две нити које треба да позову *signalS()* односно *notify()* може се десити следећи сценарио. Прво наиђе једна нит и позове методу *signalS()* и пошто нико не држи ексклузивно право приступа објекту приступа извршавању ове методе. Док прва нит извршава ову методу наиђе друга нит која такође хоће да позове исту ову методу, *signalS()*. Како прва нит већ држи кључ ова друга нит чека на улазном реду док не добије ексклузивно право приступа објекту (не нужно након прве нити). Када прва

нит увећа интерну променљиву и закључуји да је интерна променљива постала позитивна ( $s == 1$ ) позове методу *notify()*. Овом методом се једна од две блокиране нити, не зна се која, пребацује у улазни ред над датим објектом. Прва нит завршава приступ објекту и враћа ексклузивно право приступа. Ово омогућава да једна од нити које стоји у улазном реду датог објекта добије ексклузивно право приступа. Ако би се сада пустила она друга нит која позива методу *signalS()* и када та нит увећа интерну семафорску променљиву пошто променљива није постала позитивна, била је 1 а сада је 2, никога не буди и само напушта ексклузивно право приступа објекту. Сада нит која је била блокирана а након тога стајала у улазом реду добије ексклузивно право приступа и умањи семафорску променљиву на вредност 1 и напушта ексклузивно право приступа. Семафорска променљива има позитивну вредност, 1, и постоји једна нит која чека тај услов, али нико дату нит није пробудио како би проверила овај услов.

#### Друго решење

Проблем претходног решења је то што се не омогућава FIFO редослед буђења блокираних нити. Овај проблем је настало јер у програмском језику Java редослед по коме се буде нити позивањем метода *notify()* и *notifyAll()* није гарантован. Једини случај када се гарантује која ће нит бити пробуђена је случај да постоји само једна блокирана нит на неком објекту. Ово се може искористити како би се формирао FIFO семафор. Решење се заснива на коришћењу листе објеката на којима су нити блокиране (*WaitObject*), при чему је једна нит блокирана на једном објекту и на једном објекту је блокирана једна нит. Метода *waitS()* није синхронизована, већ се из ње приступа до две синхронизоване методе на различитим објектима. Унутар методе *waitS()* се прво приступа приватној синхронизованој методи *lock* која враћа објекат *WaitObject* на коме ће се нит блокирати, уколико треба да се блокира, односно *null* објекат уколико нит не треба да се блокира. Метода *lock()* недељиво проверава да ли је постоји нека блокирана нит (*queue.size() > 0*) или је интерна променљива мања или једнака нули ( $s \leq 0$ ). Уколико је ово случај креира се објекат класе *WaitObject* који се убацује на крај листе блокираних и враћа се референца на дати објекат. Уколико то није случај, што значи да нема блокираних нити и да је семафорска променљива већа од нуле, умањује се интерна семафорска променљива и враћа *null* објекат. Уколико је метода *lock()* вратила *null* објекат завршава се приступ семафору из *waitS()* методе. Уколико је вратила *WaitObject* објекат различит од *null* унутар методе *waitS()* чека се на синхронизованој методи *await()* тог *WaitObject* објекта. Метода *signalS()* је синхронизована и прво проверава да ли има блокираних нити, односно објекта на којима су нити блокиране. Уколико постоји нека блокирана нит из листе се преузима *WaitObject* објекат над којим се позива синхронизована метода *signal*. На овај начин се буди једна блокирана нит која је чекала баш над тим објектом. Пошто је из листе недељиво преузет први објекат над којим се обавља синхронизација обезбеђена је FIFO дисциплина буђења нити. Уколико није било блокираних нити увећава се интерна семафорска променљива. Пошто се у методи *waitS()* напушта синхронизована метода *lock* па се тек онда прелази на синхронизовану методу *waitS()* може се десити да баш у том тренутку нека друга нит позове методу *signalS()* из које се позове метода *signal*. Како би се обезбедило да се нит, која се блокира над датим *WaitObject* објектом, пробуди потребно је да уместо безусловног чекања на методи *wait()* прво стоји провера услова (*state != DONE*) да ли је можда већ нека нит позвала *notify()* методу и поставила дату услов (*state = DONE*).

```
public class Semaphore {  
    private int s;  
    private LinkedList<WaitObject> queue;  
  
    public Semaphore(int val) {
```

```

        s = val;
        queue = new LinkedList<WaitObject>();
    }

@Deprecated
public synchronized void initS(int val) {
    s = val;
    queue = new LinkedList<WaitObject>();
}

public void waits() {
    WaitObject lock = lock();
    if (lock == null) {
        return;
    }
    lock.await();
}

private synchronized WaitObject lock() {
    WaitObject result = null;
    if (queue.size() > 0 || s <= 0) {
        result = new WaitObject();
        queue.add(result);
    } else {
        s--;
    }
    return result;
}

public synchronized void signals() {
    if (queue.size() > 0) {
        WaitObject waitObject = queue.remove();
        waitObject.signal();
    } else{
        s++;
    }
}

class WaitObject {
    static final int DONE = 0;
    static final int NOTDONE = 1;

    int state;

    public WaitObject() {
        this.state = NOTDONE;
    }

    public synchronized void await() {
        while (state != DONE) {
            try {
                wait();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}

```

```
        }
    }

    public synchronized void signal() {
        state = DONE;
        notify();
    }

}
```

### Треће решење

За синхронизацију се у програмском језику Java поред развоја нових монитора, кључна реч *synchronized*, могу користити и већ развијене класе унутар *java.util.concurrent* пакета. Овај пакет пружа већи број класа и интерфејса кроз које је већ реализована синхронизација неопходна у већем броју класичних проблема синхронизације. Синхронизација је у неким случајевима реализована користећи мониторе, али је у великом броју случајева синхронизација реализована на нижем нивоу користећи карактеристике Java меморијског модела. Постојећа класа која омогућава рад са семафорима и која има доста више могућности у односу на описану класу је класа *Semaphore* пакета *java.util.concurrent*. Основне методе класе *Semaphore* би биле њен конструктор, *acquire*, *release* што би одговарало иницијализацији (*init*), чекању на семафору (*wait*) и ослобађању семафора (*signal*), респективно.

Конструктором класе *Semaphore* се поставља почетна вредност интерне семафорске променљиве која се задаје параметром *permits*. Ова вредност може бити и негативна. Поред постављања ове вредности може се поставити и параметар за правичност датог семафора, *fair*. У случају да се параметар постави на вредност на *true* радиће се о поштеном семафору који поштује *FIFO* редослед буђења уколико више нити чека. Подразумевана вредност овог параметра је *false*. Треба приметити да вредност *false* не значи да семафор сигурно неће будити по *FIFO* редоследу него да у неким случајевима можда то неће радити у зависности од интерне имплементације.

Метода *acquire* умањује интерну вредност семафорске променљиве уколико је она већа од нуле, иначе се нит блокира док нека друга нит не увећа интерну семафорску променљиву или док нека друга нит не позове прекид, *interrupt*, блокиране нити. Постоје два облика ове методе:

```
void acquire()
void acquire(int permits)
```

код првог се чека да интерна семафорска променљива буде барем 1, док се у другом случају чека да буде барем *permits*. Уколико не позван прекид емитује се изузетак типа *InterruptedException*. Уколико је параметар *permits* негативан емитује се *IllegalArgumentException* изузетак.

Поред метода које реагују на прекид класа *Semaphore* има и две методе које обављају исту функцију као и метода *acquire* или које не реагују на прекид. То су методе:

```
void acquireUninterruptibly()
void acquireUninterruptibly(int permits)
```

Поред блокирајућих метода *acquire* постоје и неблокирајуће и временски ограничени методе које обављају исту функцију. Ове методе се разликују по томе што имају

повратну вредност која специфицира да ли је операција умањивања интерне семафорске променљиве успела. Ове методе би биле:

```
boolean tryAcquire()
boolean tryAcquire(int permits)
boolean tryAcquire(int permits, long timeout, TimeUnit unit)
boolean tryAcquire(long timeout, TimeUnit unit)
```

#### Метода

```
void release()
```

увећава интерну семафорску променљиву за 1. Уколико је нека нит покушавала да умањи интерну променљиву и није успела, а тренутно чека одабира се једна нит која се враћа у листу активних нити и ажурира се интерна променљива. Није неопходно да нит која позива методу *release* претходно позове методу *acquire*. Не проверава се коректност датог програмског кода апликације. Треба приметити да је нит која се поново активира можда чекала да вредност интерне семафорске променљиве буде већа од 1, *permits*.

Постоје и мало сложенији облик ове методе

```
void release(int permits)
```

код које се увећава интерну семафорску променљиву за *permits*. Одабира се једна нит која чека на интерној семафорској променљиви. Уколико услов за буђење те нити није испуњен завршава се са методом. Уколико је услов испуњен нит се пребацује у листу активних и ажурира се интерна променљива. Након овога се поступак понавља све док се или листа блокираних не испразни или не нађе на неку нит код које услов није испуњен. Уколико је параметар *permits* негативан еmitује се *IllegalArgumentExeception* изузетак.

Поред ових стандардних метода класа *Semaphore* поседује и известан број специфичних метода које омогућавају да се сазна вредност интерне променљиве (*availablePermits*), да се ова вредност постави на нулу и сазна колико је та вредност била (*drainPermits*), да се сазна колико има блокираних нити (*hasQueuedThreads*), колико блокираних нити (*getQueueLength*) и које су то нити (*getQueuedThreads*). Приступом објектима типа *Semaphore* успоставља се релација да се тај приступ догодио пре (*happens-before*) првог наредног наведеног приступа.

**104**

## Произвођач и потрошач: условна синхронизација процеса

Дат је упоредни програм на језику Java:

```

public class Point{
    public int x, y;
    public Point(int x, int y){
        this.x = x;
        this.y = y;
    }
}
public class Makepoints extends Thread {
    Point p;
    int n;
    public Makepoints(Point p, int n) {
        this.p = p;
        this.n = n;
    }
    public void run() {
        for (int i = 1; i < n; i++) {
            p.x = i;
            p.y = i * i;
        }
    }
}
public class Printpoints extends Thread {
    Point p;
    int n;
    public Printpoints(Point p, int n) {
        this.p = p;
        this.n = n;
    }
    public void run() {
        for (int i = 0; i < n; i++) {
            String s = p.x + " " + p.y;
            System.out.println(s);
        }
    }
}
public class Graph {
    public static final int N = ...;
    public static void main(String[] args) {
        int n = N;
        Point p = new Point(0, 0);
        Makepoints makepoints = new Makepoints(p, n);
        Printpoints printpoints = new Printpoints(p, n);
        makepoints.start();
        printpoints.start();
    }
}

```

- a) Одредити све могуће излазе програма за случај  $n=1$ .

б) Шта би се променило у случају да су променљиве  $x$  и  $y$  типа *long* са почетним вредностима -1.

Отклонити временску зависност у датом програму:

в) употребом семафора.

г) користећи синхронизацију на објекту налик на условне критичне регионе.

## Решење

а) За  $n=1$  петља у нити *Makerooints* има једну итерацију. У итерацији се врши додељивање вредности променљивој  $x$ , и променљивој  $y$ . За разлику од језика проширењи *Pascal* у програмском језику *Java* није гарантован редослед по коме се ове две доделе извршити. Гарантовано је да ће резултат уколико се извршава само једна нит бити истоветан као у секвенцијалном коду код кога се прво додељује вредност променљиви  $x$  па онда променљиви  $y$ . Означимо упис вредности у променљиву  $x$  са  $x_m$ , а упис вредности у променљиву  $y$  са  $y_m$ .

Петља у нити *Printpoints* има две итерације. У свакој итерацији се врши читање вредности променљиве  $x$ , и променљиве  $y$ , по неком редоследом. Након читања ових вредности обавља се њихов испис који садржи синхронизациону акцију приступа монитору која обезбеђује да се прва итерација догађа пре наредне итерације. Читање вредности променљиве  $x$  у првој и другој итерацији су означені са  $x_{p0}$  и  $x_{p1}$ , док су

		редослед операција →						Излаз
1.	Операција Вредности $x/y$ након операције	$x_m$ 1/0	$y_m$ 1/1	$x_{p0}$ <b>1/1</b>	$y_{p0}$ <b>1/1</b>	$x_{p1}$ <b>1/1</b>	$y_{p1}$ <b>1/1</b>	(1,1)(1,1)
7.	Операција Вредности $x/y$ након операције	$x_m$ 1/0	$y_{p0}$ <b>1/0</b>	$y_m$ 1/1	$x_{p0}$ <b>1/1</b>	$x_{p1}$ <b>1/1</b>	$y_{p1}$ <b>1/1</b>	(1,0)(1,1)
14.	Операција Вредности $x/y$ након операције	$x_m$ 1/0	$x_{p0}$ <b>1/0</b>	$y_{p0}$ <b>1/0</b>	$y_{p1}$ <b>1/0</b>	$y_m$ 1/1	$x_{p1}$ <b>1/1</b>	(1,0)(1,0)
25.	Операција Вредности $x/y$ након операције	$x_{p0}$ <b>0/0</b>	$x_m$ 1/0	$y_m$ 1/1	$y_{p0}$ 1/1	$x_{p1}$ 1/1	$y_{p1}$ 1/1	(0,1)(1,1)
29.	Операција Вредности $x/y$ након операције	$x_{p0}$ <b>0/0</b>	$x_m$ 1/0	$y_{p0}$ <b>1/0</b>	$y_m$ 1/1	$x_{p1}$ 1/1	$y_{p1}$ 1/1	(0,0)(1,1)
34.	Операција Вредности $x/y$ након операције	$x_{p0}$ <b>0/0</b>	$x_m$ 1/0	$y_{p0}$ <b>1/0</b>	$y_{p1}$ <b>1/0</b>	$y_m$ 1/1	$x_{p1}$ <b>1/1</b>	(0,0)(1,0)
73.	Операција Вредности $x/y$ након операције	$x_{p0}$ <b>0/0</b>	$y_{p0}$ <b>0/0</b>	$x_{p1}$ <b>0/0</b>	$x_m$ 1/0	$y_m$ 1/1	$y_{p1}$ 1/1	(0,0)(0,1)
77.	Операција Вредности $x/y$ након операције	$x_{p0}$ <b>0/0</b>	$y_{p0}$ <b>0/0</b>	$x_{p1}$ <b>0/0</b>	$x_m$ 1/0	$y_{p1}$ <b>1/0</b>	$y_m$ 1/1	(0,0)(0,0)
81.	Операција Вредности $x/y$ након операције	$y_m$ 0/1	$x_{p0}$ <b>0/1</b>	$y_{p0}$ <b>0/1</b>	$x_{p1}$ <b>0/1</b>	$x_m$ 1/1	$y_{p1}$ 1/1	(0,1)(0,1)

**Табела 4.** Неки могући сценарији током извршавања програма (редослед извршавања атомских операција читања и уписа у меморију је са лева у десно). За сваки од сценарија дате су међувредности променљивих  $x$  и  $y$  и наглашено су вредности које ће се исписати. Могуће је 9 различитих излаза: (1,1)(1,1), (1,0)(1,1), (1,0)(1,0), (0,1)(1,1), (0,0)(1,1), (0,0)(1,0), (0,0)(0,1), (0,0)(0,0) и (0,1)(0,1). За разлику од табеле 2 овде је могућа и комбинација (0,1)(0,1). До ове комбинације може доћи у више сценарија (9) у којима се прво додељује вредност променљиви  $x$  па након тога променљиви  $y$ .

читање вредности променљиве у у првој и другој итерацији означени са  $y_{p0}$  и  $y_{p1}$ .

Приступ локалним променљивама у обе нити не утиче на резултат.

Нити *Makernotes* и *Printpoints* се извршавају конкурентно, па је могуће произволно учешљавање (*interleaving*) њихових операција. Што се нити *Makernotes* тиче редослед додељивања вредности променљивама  $x$  и  $y$  није познат. У нити *Printpoints* се најпре извршава итерација за  $i=0$ , и то  $x_{p0}$  и  $y_{p0}$  у неком редоследу па потом итерација за  $i=1$ , па  $x_{p1}$  и  $y_{p1}$  у неком редоследу. Укупан број различитих комбинација учешљавања ових нити је 120.

б) У случају да су променљиве  $x$  и  $y$  типа *long* број различитих сценарија се драстично повећава јер читање и упис у ове променљиве није атомска операција. Приступ овим променљивама се састоји из два дела први део је читање/упис у ника 32 бита и други у виша 32 бита. У нити *Makernotes* би биле 4 атомске операције  $x_{ml}$  и  $x_{mh}$  за упис вредности у променљиву  $x$  и  $y_{ml}$  и  $y_{mh}$  за упис вредности у променљиву  $y$ . У нити *Printpoints*, која има две итерације, би се прво обавило читање са  $x_{plo}$  и  $x_{pho}$ , као и  $y_{plo}$  и  $y_{pho}$ , док би се после обавило читање са  $x_{pl1}$  и  $x_{ph1}$ , као и  $y_{pl1}$  и  $y_{ph1}$ . Укупан број различитих комбинација учешљавања ових нити је 106920. Број могућих излаза програма је 36: (-1,-1)(-1,-1), (-1,-1)(-1,0), (-1,-1)(0,-1), (-1,-1)(0,0), (-1,0)(0,0), (-1,0)(-1,0), (0,-1)(0,-1), (0,-1)(0,0), (0,0)(0,0), (-4294967296,-4294967296)(0,-4294967296), (-4294967296,-1)(0,0), (0,-1)(-4294967296), (-4294967296,-4294967296)(-4294967296,0), (-4294967296,0)(-4294967296,0), (-1,-4294967296)(-4294967296), (-1,-4294967296)(-4294967296,-4294967296), (-1,-1)(-1,-4294967296), (-1,-4294967296)(-1,0), (-4294967296,-1)(-4294967296,1), (-4294967296,-4294967296)(0,0), (-1,-4294967296)(0,0), (-4294967296,-4294967296)(-4294967296,-4294967296), (-4294967296,-1)(0,-1), (-4294967296,0)(0,0), (-1,-4294967296)(-1,-4294967296), (-4294967296,-1)(0,-4294967296), (-4294967296,-1)(-4294967296,0), (-1,-1)(-4294967296), (-1,-1)(-4294967296,-1), (-1,-1)(-4294967296,0), (-4294967296,0)(0,0). Разлог зашто је одабрана почетна вредност -1 је тај да би се приказало да се виша 32 битна реч 64 битних променљивих  $x$  и  $y$  мења. У случају да је почетна вредност била 0 број могућих комбинација би остао исти, али би број различитих излаза био 9, као и у случају 32 битних променљивих, јер се виша реч није мењала.

```

в)
public class Point {

    public int x;

    public int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

public class Makernotes extends Thread {

    Point p;

    int n;

    Semaphore full, empty;
}

```

```

public Makepoints(Point p, int n,
                  Semaphore full, Semaphore empty) {
    this.p = p;
    this.n = n;
    this.full = full;
    this.empty = empty;
}

public void run() {
    for (int i = 1; i < n; i++) {
        empty.waits();
        p.x = i;
        p.y = i * i;
        full.signals();
    }
}

public class Printpoints extends Thread {

    Point p;

    int n;

    Semaphore full, empty;

    public Printpoints(Point p, int n,
                       Semaphore full, Semaphore empty) {
        this.p = p;
        this.n = n;
        this.full = full;
        this.empty = empty;
    }

    public void run() {
        for (int i = 0; i < n; i++) {
            full.waits();
            String s = p.x + " " + p.y;
            System.out.println(s);
            empty.signals();
        }
    }
}

public class Graph {
    public static final int N = 89;

    public static void main(String[] args) {
        int n = N;
        Point p;
        p = new Point(0, 0);
        Semaphore full = new Semaphore();
        Semaphore empty = new Semaphore();
    }
}

```

```

        Makepoints makepoints =
            new Makepoints(p, n, full, empty);
        Printpoints printpoints =
            new Printpoints(p, n, full, empty);
        full.initS(1);
        empty.initS(0);
        makepoints.start();
        printpoints.start();
    }
}
}

```

Треба приметити да у овом решењу постоји и скривена синхронизација. Метода *println* која се позива над статичком инстанцом *System.out* класе *PrintStream* у себи такође садржи синхронизацију. Приликом тестирања конкурентне апликације потребно је бити веома обазрив према оваквим позивима, уколико су уведени само због исписа у фази тестирања. Може се десити да је позив ка овој методи увео скривену баријеру над којом су нити почеле правилно да се синхронизују, а чега неће бити када се заврши фаза тестирања и уклоне дати исписи.

г) Овде се може увести аналогија између класичних региона и синхронизације на објекту у програмском језику Java. Дељени запис *res* се пресликава у *res* инстанцу класе над којом ће се обавити синхронизација. Добијање ексклузивног права *region res* се пресликава у закључавање еквивалентног објекта *synchronized(res)*. Услов *await(condition)* се пресликава у *while(!condition) res.wait()*. Напуштање ексклузивног права коришћења региона се пресликава у *res.notifyAll()*. Треба приметити да се напуштање ексклузивног права коришћења региона чини у два случаја. Први је напуштање региона, а други је блокирање на услову. Како би се омогући да решење у програмском језику Java исправно ради потребно је елиминисати непотребну сигнализацију *res.notifyAll()*. Ову сигнализацију је потребно оставити само у случају да је у време док је нит имала ексклузивно право приступа објекту променила садржај објекта на такав начин да је неки од услова блокираних нити могао да постане испуњен. Уколико нису познати сви услови блокирања, као гранични случај када нема потребе за сигнализацијом се може узети случај ако нит није мењала стање објекта од тренутка када је последњи пут добила ексклузивно право приступа том објекту до тренутка када је напустила то ексклузивно право.

```

public class Point{

    public int x;
    public int y;
    public boolean full;
    public Point(int x, int y, boolean full){
        this.x = x;
        this.y = y;
        this.full = full;
    }
}

public class Makepoints extends Thread{

    Point p;
    int n;
}

```

```

public Makepoints(Point p, int n) {
    this.p = p;
    this.n = n;
}

public void run(){
    for(int i = 1; i < n; i ++){
        synchronized(p){
            while (p.full) {
                try {
                    p.wait();
                } catch (InterruptedException e) { }
            }
            p.x = i;
            p.y = i*i;
            p.full = true;
            p.notify();
        }
    }
}

public class Printpoints extends Thread{

    Point p;
    int n;

    public Printpoints(Point p, int n){
        this.p = p;
        this.n = n;
    }

    public void run(){
        for(int i = 0; i < n; i ++){
            synchronized(p){
                while (!p.full) {
                    try {
                        p.wait();
                    } catch (InterruptedException e) { }
                }
                String s = p.x + " " + p.y;
                System.out.println(s);
                p.full = false;
                p.notify();
            }
        }
    }
}

public class Graph{
    public static final int N = 89;

    public static void main(String []args){
        int n = N;
    }
}

```

```
    Point p;
    p = new Point(0, 0, true);

    Makepoints makepoints = new Makepoints(p, n);
    Printpoints printpoints = new Printpoints(p, n);

    makepoints.start();
    printpoints.start();
}
}
```

105

## Произвођач и потрошач

Написати програм који реализује интеракцију типа производа-потрошач помоћу кружног бафера, уз обезбеђену правилну синхронизацију.

## Решење

Како би се решио овај синхронизациони проблем формира се дељени објекат, бафер, у који се могу смештати и из кога се могу преузимати смештени објекти. Узето је да дељени објекти треба да имплементирају генерички интерфејс *Buffer* који има методу за убаџивање генеричког елемента у бафер, *put*, и методу за дохватање генеричког елемента из бафера, *get*. Пошто се ради о бафери коначног капацитета може се десити да нити морају да чекају док се не појави објекат у баферу, односно док се не ослободи место у баферу. Како би се омогућило да се блокиране нити пробуде позивањем прекида *interrupt* постављено је дате методе емитују одговарајући изузетак, *InterruptedException*, када се позове прекид а дата нит је била блокирана на баферу.

```
public interface Buffer<T> {

    void put(T x) throws InterruptedException;

    T get() throws InterruptedException;

}
```

## Прво решење

Међусобно искључивање потрошача и производа се постиже преко монитора. Методе генеричке класе *BoundedBufferS*, која имплементира интерфејс *Buffer* су декларисане као *synchronized*. Синхронизација са производачима унутар *get* метода се врши преко услова *count==0*. Процес потрошач чека (преко методе *wait*) све док тај услов не престане да важи. Извршење методе *wait* доводи до ослобађања монитора од стране процеса потрошача. Потом се чека на нотификацију од стране производа. Када производач стави нови елемент у бафер, он о томе обавештава све процесе који чекају са *wait*. Један од њих добија право приступа, а остали поново чекају. Конструкција *try-catch* служи за хватање изузетака који може да се генерише, а петља служи да процес у том случају може да настави са чекањем на услову. Аналогно раде и процеси производачи. Једини разлика је у услову синхронизације. Елементи се чувају у генеричком низу *buffer*, а приступа и се користећи позицију првог слободног, *tail*, и последњег попуњеног елемента, *head*. Пошто се ради о генеричком низу на почетку се формира низ објекта који се експлицитно конвертује у генерички низ. Пошто се ради о конверзији у генерички тип поставља се анотација да се не проверавају обавештења везана за конверзију *@SuppressWarnings("unchecked")*.

```
public class BoundedBufferS<T> implements Buffer<T>{

    private T[] buffer;
    private int tail, head, count;

    @SuppressWarnings("unchecked")
    public BoundedBufferS(int size) {
        buffer = (T[]) new Object[size];
        tail = 0;
```

```
    head = 0;
    count = 0;
}

@Override
public synchronized void put(T x) throws InterruptedException {
    while (count == buffer.length) {
        wait();
    }
    buffer[tail] = x;
    tail = (tail + 1) % buffer.length;
    count++;
    notifyAll();
}

@Override
public synchronized T get() throws InterruptedException {
    while (count == 0) {
        wait();
    }
    T x = buffer[head];
    head = (head + 1) % buffer.length;
    count--;
    notifyAll();
    return x;
}
}
```

### Друго решење

Уколико је потребно да се не резервише комплетан простор за смештање елемената у бафтер уместо статичке алокације простора користећи низове могу се користи листа за смештање елемената низа. Стандарде структуре података, укључујући и листе се могу наћи у пакету `java.util.*` а које су развијене унутар оквира *Java Collections Framework*. Овај оквир омогућава рад са стандардним и конкурентним структурама података. Стандардне структуре података би укључивале скупове (*Set*), листе (*List*), декове (*Deque*) и мапе (*Map*). Генерички интерфејс за рад са листама је *List<T>*, а неке од класа које имплементирају овај интерфејс су уланчана листа, *LinkedList<T>*, и листа низова, *ArrayList<T>*. Прва је погодна за уметање на и узимање са краја односно почетка, док је друга погодна за приступ произвољном елементу.

```
import java.util.*;

public class BoundedBufferListS<T> implements Buffer<T> {

    private List<T> buffer;
    private int size;

    public BoundedBufferListS(int size) {
        buffer = new LinkedList<T>();
        //buffer = new ArrayList<T>();
        this.size = size;
    }
}
```

```

@Override
public synchronized void put(T x) throws InterruptedException {
    while (buffer.size() == size) {
        wait();
    }
    buffer.add(x);
    notifyAll();
}

@Override
public synchronized T get() throws InterruptedException {
    while (buffer.size() == 0) {
        wait();
    }
    T x = buffer.remove(0);
    notifyAll();
    return x;
}
}

```

### Треће решење

Недостатак претходних решења је тај што се буде све блокиране нити како би се информација о томе да ли је елемент убачен у бафер или је ослобођен прости у баферу проследила одговарајућој нити. Монитори у програмском језику Java који су реализовани користећи кључну реч *synchronized* имају само један ред блокираних нити, немају већи број условних променљивих као што је било код класичних монитора. Како би се отклонио овај недостатак и како би могла да се користе готова решења која су прилагођена за мониторе са већим бројем условних променљивих који имају *Signal and Continue* дисциплину почев од верзије 1.5 на располагању је пакет *java.util.concurrent.locks*. Овај пакет пружа скуп класа и интерфејса чијом је правилном применом омогућен други приступ раду са мониторима у програмском језику Java. Кроз интерфејс *Lock* је омогућено закључавање објекта налик на оно које постоји код класичних монитора. Омогућено је да се из једне мониторске методе може позвати друга мониторска метода без тражења нове дозволе (*reentrant*) као и правичност на улазном реду датог монитора. Једна од класа које имплементирају овај интерфејс је класа *ReentrantLock*. Условне променљиве које се могу везати за неки монитор су омогућене кроз интерфејс *Condition*. Инстанце ове класе се маду добити позивање одговарајуће методе над објектом *Lock* који је придружен датом монитору.

Поступак трансформације неке класе настале према класичним мониторима у мониторску класу у програмском језику Java користећи пакет *java.util.concurrent.locks* захтева поштовање неколико правила.

Свака инстанца монитора треба да има свој објекат над којим се обавља њено закључавање, *Lock lock*, и добијање ексклузивног права приступа датој инстанци монитора. Над овим објектом се формира улазни ред чекања за добијање ексклузивног права приступа монитору. Овај објекат се најчешће креира унутар конструктора датог монитора. Класа која имплементира овај интерфејс *Lock* је *ReentrantLock*. Конструктору класе *ReentrantLock* се може поставити да ли се очекује поштено решење за улазни ред (*true*) или не (*false*). Подразумевана вредност је *false*. Исто као и код семафора, *java.util.concurrent.Semaphore*, постављањем овог аргумента на *false* не значи да монитор сигурно неће бити поштен приликом буђења него да у неким случајевима можда то неће бити, у зависности од интерне имплементације.

```
Lock lock = new ReentrantLock();
или
Lock lock = new ReentrantLock(true);
```

Добијање условних променљивих, *Condition*, над датим монитором се постиже позивањем методе *lock.newCondition()*. Условне променљиве, односно њихове методе, имају слично понашање као мониторске методе класе *Object* (*wait*, *notify* и *notifyAll*), али како би се разликовале од њих имају мало другачија имена. Имена одговарајућих метода су *await*, *signal* и *signalAll*. Уколико се ове методе позову а претходно није била позвана метода за закључавање оваквог монитора, *lock*, емитује се изузетак *IllegalMonitorStateException*. Када се позове метода *await* напушта се ексклузивно право приступа, чека се док је нека друга нит не обавести, а приликом повратка из ове методе ексклузивно право се поново добија. Уколико неко позове методу *interrupt* над датом нити док је она била блокирана на *await* методи емитује се изузетак *InterruptedException*. За разлику од класе *Object* овде се нити које чекају обавештавају по *FIFO* редоследу. Треба приметити да иако се нити на условној променљиви обавештавају по *FIFO* редоследу оне добијају ексклузивно право приступа сходно дисциплини на *lock* објекту. Ова дисциплина није подразумевано дефинисана, али како је приликом иницијализације аргумент *fair* постављен на вредност *true* онда ће нит која најдуже чека имати предност. Као и код класе *Object* једна нит може исти објекат закључати већи број пута, при чему једно откључавање поништава ефекте једног закључавања.

Како би нека метода одговарала мониторској методи о обезбедила ексклузивно право приступа треба да започне закључавањем објекта *lock* (*lock.lock()*) након чека целокупно тело методе треба ставити у један *try* блок. У завршној, *finally*, грани овог *try* блока дати објекат треба откључати (*lock.unlock()*). На овај начин је обезбеђено међусобно искључивање приликом приступа датом објекту.

```
public void criticalSection() {
    lock.lock();
    try {
        ...
    } finally {
        lock.unlock();
    }
}
```

Треба приметити да методе класе која се трансформише у монитор не треба да буду означене као синхроне, *synchronized*. Такође треба приметити да се класе и објекти овог пакета могу користити и самостално и независно од поступка рада са мониторима. Приступом објектима типа *Lock* и *Condition* успоставља се релација да се тај приступ додгио пре (*happens-before*) првог наредног наведеног приступа.

Поред описаног начина могуће је и не блокирајуће, односно временски ограничено сачекати ексклузивно право приступа објекту коришћењем методе *tryLock*. Уколико метода врати *true* то значи да је добијено ексклузивно право приступа објекту. У том случају је потребно обезбедити откључавање објекат на крају рада са њим што се постиже у *finally* грани. Уколико метода врати *false* то значи да није добијено ексклузивно право приступа објекту и да вероватно треба покушати касније поново, уколико то логика проблема који се решава налаже.

```
public boolean criticalSection() {
    boolean locked = false;
    try {
```

```

locked = lock.tryLock();
if(locked){
    ...
}
} finally {
    if (locked) {lock.unlock();}
}
return locked;
}
}

```

Решење полази од првог решења овог проблема, а заснива се на монитору описаном у задатку 31, који је прилагођен, додавањем *while* петље, дисциплини *Signal and Continue*.

```

import java.util.concurrent.locks.*;

public class BoundedBufferL<T> implements Buffer<T> {
    private Lock lock;
    private Condition notFull;
    private Condition notEmpty;

    private T[] buffer;
    private int tail, head, count;

    @SuppressWarnings("unchecked")
    public BoundedBufferL(int size) {
        buffer = (T[]) new Object[size];
        tail = 0;
        head = 0;
        count = 0;
        lock = new ReentrantLock();
        notFull = lock.newCondition();
        notEmpty = lock.newCondition();
    }

    @Override
    public void put(T x) throws InterruptedException {
        lock.lock();
        try {
            while (count == buffer.length)
                notFull.await();
            buffer[tail] = x;
            tail = (tail + 1) % buffer.length;
            count++;
            notEmpty.signal();
        } finally {
            lock.unlock();
        }
    }

    @Override
    public T get() throws InterruptedException {
        lock.lock();
        try {
            while (count == 0)

```

```
        notEmpty.await();
    T x = buffer[head];
    head = (head + 1) % buffer.length;
    count--;
    notFull.signal();
    return x;
} finally {
    lock.unlock();
}
}
```

#### Четврто решење

Проблем Произвођача и потрошача коришћењем бафера представља један од најчешћих синхронизационих проблема који се среће у пракси. За решавање овог проблема се могу користити претходно описана решења, али и већ развијене класе и интерфејси који се налазе унутар `java.util.concurrent` пакета. Интерфејс који описује рад са блокирајућим бафером коначног капацитета је `BlockingQueue`. Овај интерфејс представља проширење интерфејса `Queue` из пакета `java.util`, али са том разликом што декларише и методе које дозвољавају и блокирајуће приступе, као и временски одграђиване приступе. Интерфејс пружа 4 различита начина за смештање и дохватање елемената који су у баферу.

Први је не блокирајући са изузетима код кога се елементи смештају користећи методу `add(e)`, избацију и дохватају користећи методу `remove()`, и дохватају без избацивања користећи методу `element()`. Уколико се покуша уметање у пун бафер или узимање из празног бафера емитује се изузетак `IllegalStateException`.

Други је не блокирајући са специјалним вредностима код кога се елементи смештају користећи методу `offer(e)`, избацију и дохватају користећи методу `poll()`, и дохватају без избацивања користећи методу `peek()`. Уколико се покуша уметање у пун бафер враћа се вредност `false` да операција није успела, уколико се покуша узимање из празног бафера добија се повратна вредност `null`.

Трећи је блокирајући временски неограничен приступ код кога се елементи смештају користећи методу `put(e)`, избацију и дохватају користећи методу `take()`, нема методе за дохватање без избацивања. Уколико се покуша уметање у пун бафер чека се док се место не ослободи, уколико се покуша узимање из празног бафера чека се док се не појави елементу у баферу. Уколико неко позове методу `interrupt` над датом нити док је она била блокирана емитује се изузетак `InterruptedException`.

Четврти је блокирајући временски ограничени приступ са повратном вредношћу код кога се елементи смештају користећи методу `offer(e, time, unit)`, избацију и дохватају користећи методу `poll(time, unit)`, ни овде нема методе за дохватање без избацивања. Уколико се покуша уметање у пун бафер чека се док се место не ослободи или док не истекне задати интервал времена. Уколико истекне интервал времена враћа се вредност `false` да уметање није успело. Уколико се покуша узимање из празног бафера чека се док се не појави елементу у баферу или истекне интервал времена. Уколико истекне интервал времена враћа се вредност `null` да дохватање није успело. Уколико неко позове методу `interrupt` над датом нити док је чекала емитује се изузетак `InterruptedException`.

Пошто се као индикатор да операција није успела користи специјална вредност, `null` вредност, онда није дозвољено уметање `null` објекта у бафер. У случају да се покуша са уметањем `null` вредности емитује се изузетак `NullPointerException`. Објекти типа

*BlockingQueue* могу да имају коначан капацитет, и овај капацитет се најчешће поставља приликом креирања објекта. Уколико се капацитет на постави подразумевани дозвољени капацитет износи *Integer.MAX\_VALUE*. Приступом објектима овог типа се, као и у претходним случајевима коришћења датог пакета, успоставља се релација да се тај приступ догодио пре (*happens-before*) првог наредног наведеног приступа.

Класа која имплементира овај интерфејс укључују класе *ArrayBlockingQueue<E>* и *LinkedBlockingQueue<E>*.

```
import java.util.concurrent.*;  
  
public class BoundedBufferP<T> implements Buffer<T> {  
    private BlockingQueue<T> buffer;  
  
    public BoundedBufferP(int size) {  
        buffer = new ArrayBlockingQueue<T>(size);  
    }  
  
    @Override  
    public void put(T x) throws InterruptedException {  
        buffer.put(x);  
    }  
  
    @Override  
    public T get() throws InterruptedException {  
        return buffer.take();  
    }  
}
```

106

## Филозофи за ручком

Решити проблем филозофа за ручком описан у задатку 4.

### Решење

Прво решење

```
public class Philosopher extends Thread {
    int id;
    int firstodd, secondeven;
    Semaphore [] forks;
    public Philosopher (int id, int n , Semaphore [] forks){
        this.id = id;
        this.forks = forks;
        if((id % 2) == 1){
            firstodd = id;
            secondeven = (id + 1) % n;
        }
        else{
            firstodd = (id + 1) % n;
            secondeven = id;
        }
    }

    public void run(){
        while(true){
            think();
            forks[firstodd].waitS();
            forks[secondeven].waitS();
            eat();
            forks[secondeven].signals();
            forks[firstodd].signals();
        }
    }

    private void think(){
        ...
    }
    private void eat(){
        ...
    }
}

public class DiningPhilosophersTest{
    public static final int N = 5;
    public static void main(String [] args){
        int n = N;
        Semaphore [] forks = new Semaphore[n];
        Philosopher [] philosopher = new Philosopher[n];
        for(int i = 0; i < n; i++) forks[i] = new Semaphore();
```

```

        for(int i = 0; i < n; i++)
            philosopher[i] = new Philosopher(i, n, forks);
        for(int i = 0; i < n; i++) forks[i].initS(1);
        for(int i = 0; i < n; i++) philosopher[i].start();
    }
}

```

Друго решење

```

public class Philosopher extends Thread {
    int id;
    int left, right;
    Semaphore [] forks;
    Semaphore ticket;
    public Philosopher (int id, int n,
                        Semaphore [] forks, Semaphore ticket){
        this.id = id;
        this.forks = forks;
        this.ticket = ticket;
        left = id;
        right = (id + 1) % n;
    }

    public void run(){
        while(true){
            think();
            ticket.waitS();
            forks[left].waitS();
            forks[right].waitS();
            eat();
            forks[right].signalS();
            forks[left].signalS();
            ticket.signalS();
        }
    }

    private void think(){
        ...
    }

    private void eat(){
        ...
    }
}

public class DiningPhilosophersTest{
    public static final int N = 5;
    public static void main(String [] vpar){
        int n = N;
        Semaphore ticket = new Semaphore();
        Semaphore [] forks = new Semaphore[n];
        Philosopher [] philosopher = new Philosopher[n];

        for(int i = 0; i < n; i++) forks[i] = new Semaphore();
    }
}

```

```
        for(int i = 0; i < n; i++) philosopher[i] =
            new Philosopher(i, n , forks, ticket);
        ticket.initS(n-1);
        for(int i = 0; i < n; i++) forks[i].initS(1);
        for(int i = 0; i < n; i++) philosopher[i].start();
    }
}
```

### Треће решење

Ово решење одговара решењу проблема филозофа које је реализовано помоћу монитора који имају дисциплину *Signal and Continue*.

```
public class Philosopher extends Thread {
    private int id;

    private Forks forks;

    public Philosopher(int id, Forks forks) {
        this.id = id;
        this.forks = forks;
    }

    public void run() {
        while (true) {
            think();
            forks.pickup(id);
            eat();
            forks.putdown(id);
        }
    }

    private void think() {
        ...
    }

    private void eat() {
        ...
    }
}

public class Forks {
    public static final int N = DiningPhilosophersTest.N;
    private int forks_available[];

    public Forks() {
        forks_available = new int[N];
        for (int i = 0; i < N; i++) {
            forks_available[i] = 2;
        }
    }

    public synchronized void pickup(int i) {
        while (forks_available[i] != 2) {
```

```
try {
    wait();
} catch (InterruptedException e) {
}
}

forks_available[(i + 1) % N]--;
forks_available[(i + N - 1) % N]--;

}

public synchronized void putdown(int i) {
    forks_available[(i + 1) % N]++;
    forks_available[(i + N - 1) % N]++;
    notifyAll();
}

}

public class DiningPhilosophersTest {
    public static final int N = 5;

    public static void main(String[] args) {
        Philosopher philosophers[] = new Philosopher[N];
        Forks forks = new Forks();
        for (int i = 0; i < N; i++) {
            philosophers[i] = new Philosopher(i, forks);
        }
        for (int i = 0; i < N; i++) {
            philosophers[i].start();
        }
    }
}
```

107

## Мост који има само једну коловозну траку

Решити проблем моста који има само једну коловозну траку.

а) Дати решење проблема ако претпоставимо да аутомобили који долазе из истог смера могу без икаквих ограничења истовремено да прелазе мост.

б) Усавршити претходно решење тако да буде праведно, односно да аутомобили не могу да чекају неограничено дуго како би прешли мост, него максимално док  $N$  аутомобила из супротног смера не пређе мост (ово важи под условом да је бар један аутомобил чекао да пређе мост са супротне стране; док год се не појави аутомобил са супротне стране аутомобили из истог смера могу слободно да прелазе мост).

### Решење

а)

```
public class Bridge{  
  
    public int north;  
    public int south;  
    public Bridge(int north, int south){  
        this.north = north;  
        this.south = south;  
    }  
  
    public abstract class Car{  
  
        int id;  
        Bridge bridge;  
  
        public Car(int id, Bridge bridge){  
            this.id = id;  
            this.bridge = bridge;  
        }  
        public void crossing(){  
            ...  
        }  
        public void starting(){  
            ...  
        }  
        public abstract void start();  
    }  
  
    public class North extends Car implements Runnable {  
  
        private Thread thread;  
  
        public North(int id, Bridge bridge){  
            super(id, bridge);  
        }
```

```

        thread = null;
    }

public void start() {
    if (thread == null) {
        thread = new Thread(this);
        thread.start();
    }
}

public void run() {

    starting();
    synchronized(bridge) {
        while (bridge.south != 0) {
            try {
                bridge.wait();
            } catch (InterruptedException e) { }
        }
        bridge.north++;
    }
    crossing();

    synchronized(bridge) {
        bridge.north--;
        if(bridge.north == 0) bridge.notifyAll();
    }
}
}

public class South extends Car implements Runnable {

    private Thread thread;

    public South(int id, Bridge bridge){
        super(id, bridge);
        thread = null;
    }

    public void start() {
        if (thread == null) {
            thread = new Thread(this);
            thread.start();
        }
    }
    public void run() {

        starting();
        synchronized(bridge){
            while (bridge.north != 0) {
                try {
                    bridge.wait();
                } catch (InterruptedException e) { }
            }
        }
    }
}

```

```
        bridge.south++;
    }

    crossing();

    synchronized(bridge) {
        bridge.south--;
        if(bridge.south == 0) bridge.notifyAll();
    }
}

public class OneLaneBridgeTest{
    public static final int N = 89;
    public static void main(String []args){
        int n = N;
        Bridge bridge = new Bridge(0, 0);
        Car [] south = new Car[n];
        Car [] north = new Car[n];

        for(int i = 0; i < n; i++) {
            south[i] = new South(2*i, bridge);
            north[i] = new North(2*i+1, bridge);
        }

        for(int i = 0; i < n; i++) {
            south[i].start();
            north[i].start();
        }
    }
}
```

6)

```
public class Direction{

    public int wait;
    public int cross;
    public int ahead;
    public Direction(){
        wait = 0;
        cross = 0;
        ahead = 0;
    }
}
public class Bridge{

    public Direction north;
    public Direction south;
    public Bridge(){
        north = new Direction();
        south = new Direction();
    }
}
```

```

public abstract class Car{

    int id;
    Bridge bridge;

    public Car(int id, Bridge bridge) {
        this.id = id;
        this.bridge = bridge;
    }
    public void crossing(){
        ...
    }
    public void starting(){
        ...
    }
    public abstract void start();
}

public class North extends Car implements Runnable {

    private Thread thread;

    public North(int id, Bridge bridge) {
        super(id, bridge);
        thread = null;
    }

    public void start() {
        if (thread == null) {
            thread = new Thread(this);
            thread.start();
        }
    }

    public void run() {

        starting();
        synchronized(bridge){
            bridge.north.wait++;
            while (!((bridge.south.cross == 0) &&
                    (bridge.north.ahead<10))) {
                try {
                    bridge.wait();
                } catch (InterruptedException e) { }
            }
            bridge.north.wait--;
            bridge.north.cross++;
            if (bridge.south.wait > 0) bridge.north.ahead++;
        }
        crossing();

        synchronized(bridge) {
    }
}

```

```
        bridge.north.cross--;
    if (bridge.north.cross == 0) {
        bridge.south.ahead = 0;
        bridge.notifyAll();
    }
}
}

public class South extends Car implements Runnable {

    private Thread thread;

    public South(int id, Bridge bridge) {
        super(id, bridge);
        thread = null;
    }

    public void start() {
        if (thread == null) {
            thread = new Thread(this);
            thread.start();
        }
    }
    public void run() {

        starting();
        synchronized(bridge) {
            bridge.south.wait++;
            while (!((bridge.north.cross == 0)
                    && (bridge.south.ahead<10))) {
                try {
                    bridge.wait();
                } catch (InterruptedException e) { }
            }
            bridge.south.wait--;
            bridge.south.cross++;
            if (bridge.north.wait > 0) bridge.south.ahead++;
        }

        crossing();

        synchronized(bridge) {
            bridge.south.cross--;
            if (bridge.south.cross == 0) {
                bridge.north.ahead = 0;
                bridge.notifyAll();
            }
        }
    }
}

public class OneLaneBridgeTest{
```

```
public static void main(String []args) {
    int n = 89;
    Bridge bridge = new Bridge();
    Car [] south = new Car[n];
    Car [] north = new Car[n];

    for(int i = 0; i < n; i++) {
        south[i] = new South(2*i, bridge);
        north[i] = new North(2*i+1, bridge);
    }

    for(int i = 0; i < n; i++) {
        south[i].start();
        north[i].start();
    }
}
```

108

Читаоци и писци

Решити проблем читалаца и писаца у програмском језику Java.

## Решење

Проблем читалаца и писаца описује системе код којих се извршавање може поделити у три фазе. Прво долази фаза тражења дозволе за приступ неком ресурсу, након тога долази фаза коришћења ресурса у складу са добијен дозволом и на крају долази фаза ослобађања коришћеног ресурса. Дозвола која се тражи може бити дељена или ексклузивна. У случају дељене дозволе потребо је да буду испуњени услови који ограничавају колико нити и ког типа може користити неки ресурс. Узето је да објекат који додељује дозволе за приступ дељеном ресурсу има 4 методе. Две се односе на читаоце (*startRead* и *endRead*), а две се односе на писце (*endWrite* и *startWrite*). Уместо коришћења 4 методе алтернатива овом приступу би било коришћење 3 методе. Тражење дељене дозволе (*lockShared*), тражење ексклузивне дозволе (*lockExclusive*) и враћање дозволе (*unlock*). Овде треба приметити да се синхронизација не обавља над објектом коме се приступа, јер је дозвољено да једном објекту приступа већи број нити (читалаца). Такође треба приметити да објекат који додељује дозволе за приступ не мора бити објекат над којим се обавља синхронизација, *synchronized*. То може бити неки други унутрашњи објекат.

```
public interface ReadersWriters {
    public void startRead();
    public void endRead();

    public void startWrite();
    public void endWrite();
}
```

Као и код проблема произвођача и потрошача овде се ради о проблему синхронизације код кога се може десити да нити морају да чекају док се не испуни одговарајући услов. Као и код наведеног примера овде би се такође могло дозволити буђење блокиране нити позивањем методе *interrupt* и емитовањем одговарајући изузетак, *InterruptedException*.

### Прво решење

Ово решење представља директу имплементацију решења датог у задатку 5 код кога је уведен ред приликом започињања операције читања и уписа.

```
public class ReadersWritersSem implements ReadersWriters {
    private Semaphore rw;
    private Semaphore mutexR;
    private Semaphore enter;
    private int readCount;

    public ReadersWritersSem() {
        rw = new Semaphore(1);
        mutexR = new Semaphore(1);
        enter = new Semaphore(1);
        readCount = 0;
    }
```

```

}

@Override
public void startRead() {
    enter.acquireUninterruptibly();
    mutexR.acquireUninterruptibly();
    readCount++;
    if (readCount == 1)
        rw.acquireUninterruptibly();
    mutexR.release();
    enter.release();
}

@Override
public void endRead() {
    mutexR.acquireUninterruptibly();
    readCount--;
    if (readCount == 0)
        rw.release();
    mutexR.release();
}

@Override
public void startWrite() {
    enter.acquireUninterruptibly();
    rw.acquireUninterruptibly();
    enter.release();
}

@Override
public void endWrite() {
    rw.release();
}
}
}

```

### Друго решење

Ово решење користи мониторе са *Signal and Continue* дисциплином на начин сличан ономе датом у задатку 27. Разлика је у томе што није искоришћен тикет алгоритам.

```

public class ReadersWritersSync implements ReadersWriters {
    private int readCount;
    private int writeCount;

    public ReadersWritersSync() {
        readCount = 0;
        writeCount = 0;
    }

    @Override
    public synchronized void startRead() {
        while (writeCount != 0) {
            try {
                wait();
            } catch (InterruptedException e) {
            }
        }
    }
}

```

```
        }
        readCount++;
    }

@Override
public synchronized void endRead() {
    readCount--;
    if (readCount == 0)
        notifyAll();
}

@Override
public synchronized void startWrite() {
    while ((writeCount != 0) || (readCount != 0)) {
        try {
            wait();
        } catch (InterruptedException e) {
        }
    }
    writeCount++;
}

@Override
public synchronized void endWrite() {
    writeCount--;
    notifyAll();
}
}
```

Треба водити рачуна приликом коришћења тикет алгоритма. Колико год да дати алгоритам делује праведно у случајевима када велики број нити конкурентно и непрестано приступа неком објекту може се догодити велика деградација перформанси. Код малог броја нити које конкурентно приступају обично не долази до деградације перформанси, а може се догодити да такво решење буде брже.

Како би се смањила могућност изгладњивања писаца ово решење се може учити поштенијим тако што би се увело праћење колико има блокираних нити. Ако постоје блокиране нити у тренутку када нека нова нит наилази та нит се блокира.

```
public class ReadersWritersSync2 implements ReadersWriters {
    private int readCount;
    private int writeCount;
    private int waitCount;

    public ReadersWritersSync2() {
        readCount = 0;
        writeCount = 0;
        waitCount = 0;
    }

    @Override
    public synchronized void startRead() {
        if ((waitCount != 0) || (writeCount != 0)) {
            waitCount++;
            while (writeCount != 0) {
                try {

```

```

        wait();
    } catch (InterruptedException e) {
    }
}
waitCount--;
}
readCount++;
}

@Override
public synchronized void endRead() {
    readCount--;
    if (readCount == 0)
        notifyAll();
}

@Override
public synchronized void startWrite() {
    if ((waitCount != 0) || (writeCount != 0) || (readCount != 0)) {
        waitCount++;
        while ((writeCount != 0) || (readCount != 0)) {
            try {
                wait();
            } catch (InterruptedException e) {
            }
        }
        waitCount--;
    }
    writeCount++;
}

@Override
public synchronized void endWrite() {
    writeCount--;
    notifyAll();
}
}

```

### Треће решење

Проблем читалаца и писаца представљај један од кључних синхронизационих проблема и доста често се у практичним ситуацијама користи у свом неизмењеном облику. Како би се овај проблем што једноставније решио почев од верзије 1.5 у пакету `java.util.concurrent.locks` на располагању је скуп класа и интерфејса који чине једно готово решење овог проблема. Основни интерфејс је интерфејс `ReadWriteLock` односно класа `ReentrantReadWriteLock` која имплементира дати интерфејс. Конструктору класе `ReentrantReadWriteLock` се може поставити да ли се очекује поштено решење приликом буђења нити (`true`) или не (`false`). Подразумевана вредност је `false`. Слично као код класе `ReentrantLock` или `Semaphore`, постављањем овог аргумента на `false` не значи да монитор сигурно неће бити поштен приликом буђења него да у неким случајевима можда то неће бити, у зависности од интерне имплементације. Али за разлику од наведених класа у случају да се аргумент постави на вредност `true` овде не значи да ће се поштовати стога *FIFO* дисциплина. Овде се користи приближни алгоритам буђења према томе која не нит колико чекала. Када се ради напуштање ексклузивног права коришћења ресурса гледа се ко је најдуже чекао.

Уколико је то био писац буди се тај писац и додељује му се ексклузивно право приступа. Уколико је то био читалац буди се тај читалац, али и цела група читалаца која је чекала и додељује им се право на дељени приступ ресурсу. У случају коришћења овог поштеног решења у случају када нађе читалац уколико неки писац већ чека дати читалац такође почине да чека. Такође код поштеног решења ако нађе неки писац а неки други писац или читалац који су стигли пре већ чекају дати писац такође чека.

```
ReadWriteLock rwLock = new ReentrantReadWriteLock();
или
ReadWriteLock rwLock = new ReentrantReadWriteLock(true);
```

Ова класа пруже две методе помоћу којих се дохватају синхронизациони објекти, *Lock*, преко којих се обавља синхронизација читалаца, *readLock()*, и писаца, *writeLock()*. Синхронизациони објекат за операцију читања дозвољава креирање нових условних променљивих на њему, *newCondition()*, док синхронизациони објекат за читање емитује изузетак *UnsupportedOperationException* уколико се ова метода позове.

```
import java.util.concurrent.locks.*;

public class ReadersWritersLock implements ReadersWriters {
    ReadWriteLock rw;
    Lock readLock, writeLock;

    public ReadersWritersLock() {
        // rw = new ReentrantReadWriteLock(true);
        rw = new ReentrantReadWriteLock();
        readLock = rw.readLock();
        writeLock = rw.writeLock();
    }

    @Override
    public void startRead() {
        readLock.lock();
    }

    @Override
    public void endRead() {
        readLock.unlock();
    }

    @Override
    public void startWrite() {
        writeLock.lock();
    }

    @Override
    public void endWrite() {
        writeLock.unlock();
    }
}
```

Као и у случају рада са интерфејсом *Lock* ни овде методе објекта који додељује дозволу не треба да буду означене као синхроне, *synchronized*. Исто тако успостављена је релација да се приступ догодио пре (*happens-before*) првог наредног наведеног приступа.

## Четврто решење

Поред решења проблема читалаца и писаца које укључује ексклузивно и дельено закључавање проблем је могуће решити користећи верзионисање за различите модове рада. Почек од верзије 1.8 у пакету `java.util.concurrent.locks` на располагању стоји и класа `StampedLock` која омогућава рад са верзијама. Приликом приступа у одговарајућим модовима рада добија се и ознака верзије, `stamp`, у којој могу бити садржане и додатне информације. Приликом откључавања објекта прослеђује се и ознака верзије како би се проверило којој верзији закључавања одговарају. Приступ односно закључавање објекта је омогућено кроз три основна мода рада: мод за писање, мод за читање и мод за оптимистично читање.

Улазак у мод за писање је омогућен користећи методу `writeLock`. Ово је блокирајући метода код које се чека ексклузивно право приступа објекту, а која враћа ознаку везану за текућу верзију. Ово ознаку је потребно проследити методи `unlockWrite` како би се омогућило враћање ексклузивног права приступа. Уколико се нека нит налази у овом моду ни једа друга нит не може да пређе у мод за упис и читање, док ће нити које обављају оптимистично читање морати да операцију извеште поново јер неће проћи валидацију.

Улазак у мод за читање је омогућен користећи методу `readLock`. Ово је блокирајући метода код које се чека дельено право приступа објекту, а која враћа ознаку везану за текућу верзију. Ово ознаку је потребно проследити методи `unlockWrite` како би се омогућило враћање ексклузивног права приступа. Као и код метода `writeLock` и код ове методе постоје верзије са и без ограничених чекања на добијање права приступа.

Улазак у мод за оптимистично читање је омогућен користећи методу `tryOptimisticRead`. Ова метода враћа ознаку ако нико тренутно не пише по датом објекту, у супротном враћа вредност 0 која означава да операција није успела. Метода је намењена за краткотрајне операције читања или у случајевима када се не очекује да неко уписује вредност. На крају операције је неопходно проверити да ли је можда било уписа у међувремену позивом методе `validate`. Уколико је неко писао у међувремену операцију читања је потребно поновити.

Дозвољен је прелазак између модова рада. Метода помоћу које се ово омогућава је `tryConvertToWriteLock`. Овој методи се прослеђује ознака са којом се тренутно приступа објекту који год да је текући мод рада. Треба водити рачуна да ове методе не воде рачуна о томе да ли нека нит већ држи закључан неки објекат, за разлику од синхронизације користећи `synchronized` као и објекта типа `Lock`. Исто тако треба водити рачуна о томе да ознаке моду да се после неког периода понављају. Пример коришћења ове класе би био:

```
import java.util.concurrent.locks.StampedLock;
...
private final StampedLock sLock = new StampedLock();
...
public void writeSection() {
    long stamp = sLock.writeLock();
    try {
        ...//write operation
    } finally {
        sLock.unlockWrite(stamp);
    }
}
public void shortReadOnlySection() {
    long stamp = sLock.tryOptimisticRead();
    ...//read
    if (!sL.validate(stamp)) {
        stamp = sLock.readLock();
```

```

        try {
            ...//read again
        } finally {
            sLock.unlockRead(stamp);
        }
    }

public void longReadOnlySection() {
    long stamp = slock.readLock();
    try {
        ...//read operation
    } finally {
        sLock.unlockRead(stamp);
    }
}

```

Како искористити *StampedLock* за решавање овако постављеног проблема читалаца и писаца када је потребно проследити ознаку верзије добијену на почетку операције методи која се позива на крају операције а да се притом потпис успостављеног интерфејса *ReadersWriters* не мења? Потребно је некако унутар саме класе која имплементира интерфејс *ReadersWriters* сачувати која је нит коју ознаку верзије добила како би се та ознака касније искористила. Како унутар неке класе сазнати боло шта ономе ко позива методу а да тај неко не проследи тај аргумент? Решење лежи у томе што се увек могу позвати статичке методе које дају информације у томе шта се тренутно извршава. Једна таква статичка метода је и метода *currentThread()* класе *Thread*. Користећи ову методу унутар класе која имплементира интерфејс *ReadersWriters* могуће је сазнати која ју је нит позвала. Пошто је дозвољено да једна нит само једном позове методу за закључавање, након које позива методу за откључавање могуће је информације о токе која држи коју верзију сачувати унутар дате класе у објекту који садржи парове кључ вредност. Објекат у коме се могу чувати информације облика кључ вредност се може наћи у пакету стандардних класа *java.util*. Ради се о интерфејсу *Map<Key, Value>* односно класама које имплементирају овај интерфејс. Две основне имплементације претраживања и чувања кључева унутар ове структуре је користећи хеширање *HashMap<Key, Value>* и користећи стабла *TreeMap<Key, Value>*. Ове мале су оптимизоване за појединачни приступ и нису прилагођене за конкурентни приступ. Како синхронизовати приступ датим мапама? Треба приметити да као и у случају рада са интерфејсом *Lock* ни овде методе *ReadersWriters* објекта не треба да буду означене као синхроне, *synchronized*. Ово значи да је потребно да се дате класе некако учине синхронизованим. Један начин је њихова директна трансформација користећи статичку методу *synchronizedMap* класе *Collections*. Класа *Collections* пружа велики број метода за рад са колекцијама података (сортирање, претраживање, тражење минимума/максимума, премештање, копирање, ротирање, синхронизација, ...). Други начин би био коришћење конкурентне мапе, *ConcurrentHashMap* која се налази у пакету *java.util.concurrent*. У овом пакету се могу наћи и преостале конкурентне структуре података.

```

import java.util.*;
import java.util.concurrent.*;
import java.util.concurrent.locks.*;

public class ReadersWritersStamp implements ReadersWriters{
    private final StampedLock sl;
    private Map<Thread, Long> stamps;

    public ReadersWritersStamp() {

```

```

    sl = new StampedLock();
    stamps = new ConcurrentHashMap<>();
    // stamps = Collections.synchronizedMap(
    //           new HashMap<Thread, Long>());
}

public void startRead() {
    long stamp = sl.readLock();
    stamps.put(Thread.currentThread(), stamp);
}

public void endRead() {
    long stamp = stamps.remove(Thread.currentThread());
    sl.unlockRead(stamp);
}

public void startWrite() {
    long stamp = sl.writeLock();
    stamps.put(Thread.currentThread(), stamp);
}

public void endWrite() {
    long stamp = stamps.remove(Thread.currentThread());
    sl.unlockWrite(stamp);
}
}
}

```

Уместо коришћења синхронизованих листи у којима се чувају ознаке могуће их је чувати користећи класу *ThreadLocal<T>* која пружа могућност за рад са променљивама које су локалне за неку појединачну нит. Ово решење је нешто брже у односу на претходно јер не захтева синхронизацију приликом приступа локалним променљивама НИТИ.

```

import java.util.concurrent.locks.*;

public class ReadersWritersStampThreadLocal
    implements ReadersWriters {
    private final StampedLock sl;
    private final ThreadLocal<Long> stamps;

    public ReadersWritersStampThreadLocal() {
        sl = new StampedLock();
        stamps = new ThreadLocal<Long>();
    }

    public void startRead() {
        long stamp = sl.readLock();
        stamps.set(stamp);
    }

    public void endRead() {
        long stamp = stamps.get();
        sl.unlockRead(stamp);
    }
}

```

```
public void startWrite() {
    long stamp = sl.writeLock();
    stamps.set(stamp);
}

public void endWrite() {
    long stamp = stamps.get();
    sl.unlockWrite(stamp);
}

}
```

**109****Берберин који спава**

Решити проблем берберина који спава (*The Sleeping Barber Problem*). Берберница се састоји од чекаонице са  $N=5$  столица и берберске столице на којој се лјуди брију. Уколико нема муштерија, брица спава. Уколико муштерија уђе у берберницу и све столице су заузете, муштерија не чека, већ одмах излази. Уколико је берберин заузет, а има слободних столица, муштерија седа и чека. Уколико берберин спава, муштерија га буди.

**Решење**

Решење се састоји од три класе: *Barber*, *Customer* и *BarberShop*. Класе *Barber* и *Customer* представљају нити које се синхронизују користећи мониторку класу *BarberShop*. У овом решењу је предвиђено да постоји само један берберин, али се мањом модификацијом решење може изменити тако да ради већи број берберина. *Customer* је класа која има за циљ да симулира долазак муштерија на улаз бербернице. Могуће је формирати више инстанци ове класе, а уз мање модификације се може симулирати да има више врата. Берберница је основна класа на чијим методама се заснива рад читавог система. Ова класа чува информације о стању у берберници, односно о броју муштерија који су у њој. Столице се симулирају помоћу кружног бафера, при чему се редослед опслуживања чува коришћењем FIFO алгоритма.

```
public class Barber extends Thread {
    private BarberShop barbershop;

    public Barber(BarberShop b) {
        barbershop = b;
    }

    public void run() {
        int customer;
        while (true) {
            customer = barbershop.getNextCustomer();
            shaving(customer);
            barbershop.finishedCut();
        }
    }

    public void shaving(int customer) {
        ...
    }
}

public class Customer extends Thread {
    private BarberShop barbershop;

    private int id;

    public Customer(BarberShop barbershop, int id) {
        this.barbershop = barbershop;
        this.id = id;
    }
}
```

```
}

public void run() {
    starting();
    boolean result = barbershop.getHaircut(id);
    System.out.println("Customer " + id + " was " +result);
}
public void starting(){
    ...
}

public class Barbershop {
    private int numChairs;
    private int numPresent;
    private int[] chairs;
    private int nextChair;
    private int nextToShave;
    private boolean done;

    public Barbershop(int nc) {
        chairs = new int[nc];
        numChairs = nc;
        numPresent = 0;
        nextChair = 0;
        nextToShave = 0;
        done = false;
    }

    public synchronized int getNextCustomer() {
        while (numPresent == 0) {
            try {
                wait();
            } catch (InterruptedException e) {}
        }
        return chairs[nextToShave];
    }

    public synchronized boolean getHaircut(int id) {
        if (numPresent == numChairs) {
            return false;
        }
        numPresent++;
        int myChair = nextChair++;
        chairs[myChair] = id;
        nextChair %= numChairs;
        while (myChair != nextToShave) {
            try {
                wait();
            } catch (InterruptedException e) {}
        }
        notifyAll();
        while (!done) {
            try {
                wait();
            }
        }
    }
}
```

```

        } catch (InterruptedException e) {}
    }
done = false;
notifyAll();
return true;
}

public synchronized void finishedCut() {
    done = true;
    notifyAll();
    while (done) {
        try {
            wait();
        } catch (InterruptedException e) {}
    }
    nextToShave++;
    nextToShave %= numChairs;
    numPresent--;
    notifyAll();
}
}

public class SleepingBarberTest {
    final static int NUMSEATS = 5;
    final static int NUMCUST = 12;

    public static void main(String [] args) {
        Barbershop barbershop = new Barbershop (NUMSEATS);
        Barber barber = new Barber(barbershop);
        barber.start();

        Customer customers [] = new Customer [NUMCUST];
        for(int i = 0; i < NUMCUST; i++){
            customers[i] = new Customer(barbershop, i);
            customers[i].start();
        }
    }
}
}

```

# 10

## Вожња тобоганом

Написати програм на језику Java који решава проблем вожње тобоганом (*The Roller Coaster Problem*).

### Решење

```
public class Passenger extends Thread {
    private int id;
    private Semaphore start;
    private Semaphore allAboard;
    private Semaphore stop;
    private Semaphore allOut;
    private Semaphore mutex;
    private RollerCoaster coaster;

    public Passenger( int id, Semaphore start, Semaphore allAboard,
                      Semaphore stop, Semaphore allOut,
                      Semaphore mutex, RollerCoaster coaster) {
        this.id = id;
        this.start = start;
        this.allAboard = allAboard;
        this.stop = stop;
        this.allOut = allOut;
        this.mutex = mutex;
        this.coaster = coaster;
    }

    public void run() {
        while (true) {
            starting();

            boardCar();
            ride();
            leaveCar();
        }
    }

    public void boardCar() {
        start.waits();

        mutex.waits();
        coaster.passengers++;
        System.out.println("boardCar " + id);
        if (coaster.passengers == RollerCoaster.C) {
            allAboard.signals();
        }
        mutex.signals();
    }

    public void leaveCar() {
        stop.waits();
    }
}
```

```

        mutex.waitS();
        coaster.passengers--;
        System.out.println("leaveCar " + id);

        if (coaster.passengers == 0) {
            allOut.signals();
        }
        mutex.signals();
    }

    public void starting() {
        ...
    }

    public void ride() {
        ...
    }
}

public class RollerCoaster extends Thread{
    public static final int C = 5;
    private Semaphore start;
    private Semaphore allAboard;
    private Semaphore stop;
    private Semaphore allOut;
    public int passengers;

    public RollerCoaster(Semaphore start, Semaphore allAboard,
                         Semaphore stop, Semaphore allOut, int passengers) {
        this.start = start;
        this.allAboard = allAboard;
        this.stop = stop;
        this.allOut = allOut;
        this.passengers = passengers;
    }

    public void run() {
        while (true) {
            boardCar();
            ride();
            leaveCar();
        }
    }

    public void boardCar() {
        for (int i = 1; i < C; i++)
            start.signals();
        allAboard.waitS();
    }

    public void leaveCar() {
        for (int i = 1; i < C; i++)
            stop.signals();
        allOut.waitS();
    }
}

```

```
}

public void ride() {
    ...
}

public class RollerCoasterTest {
    public static final int NUMP = 100;

    public static void main(String arg[]) {
        Semaphore start = new Semaphore();
        Semaphore allAboard = new Semaphore();
        Semaphore stop = new Semaphore();
        Semaphore mutex = new Semaphore();
        Semaphore allOut = new Semaphore();
        int passengers = 0;
        Passenger[] passenger = new Passenger[NUMP];
        RollerCoaster coaster = new RollerCoaster(start,
            allAboard, stop, allOut, passengers);

        passengers = 0;
        start.initS(0);
        allAboard.initS(0);
        stop.initS(0);
        mutex.initS(1);
        allOut.initS(0);

        coaster.start();
        for (int i = 0; i < NUMP; i++) {
            passenger[i] = new Passenger(i, start, allAboard,
                stop, allOut, mutex, coaster);
            passenger[i].start();
        }
    }
}
```

111

## Вожње аутобусом

Проблем вожње аутобусом (*The Bus Problem*). Путници долазе на аутобуску станицу и чекају први аутобус који нађе. Када аутобус нађе сви путници који су били на станици пробају да уђу у аутобус. Уколико има места у аутобусу путници улазе у њега. Капацитет аутобуса је  $K$  места. Путници који су дошли док је аутобус био на станици чекају на следећи аутобус. Када сви путници који су били на станици у тренутку доласка аутобуса провере да ли могу да уђу и уђу уколико има места аутобус креће. Уколико аутобус дође на празну станицу одмах продужава даље. Написати програм на језику Java који симулира описани систем.

## Решење

```

public class Passenger extends Thread {
    public static int pid = 0;
    int id;
    Station start, end;

    public Passenger(Station start, Station end) {
        this.id = Passenger.nextId();
        this.start = start;
        this.end = end;
    }

    public static synchronized int nextId() {
        return pid++;
    }

    public void run() {
        while (true) {
            Bus bus = start.waitBus();
            travel();
            bus.exitBus(end);
        }
    }

    private void travel() {
        ...
    }
}

public class Busdriver extends Thread {
    public static int did = 0;
    int id;
    private Bus bus;

    public Busdriver(Bus bus) {
        this.id = Busdriver.nextId();
        this.setName("Passenger:" + id);
        this.bus = bus;
    }
}

```

```
}

public static synchronized int nextId() {
    return did++;
}

public void run() {
    while (true) {
        Station nextStation = bus.getNextStation();
        ride(nextStation);
        bus.permissionToExit();
        nextStation.busEnter(bus);
    }
}

private void ride(Station station) {
    ...
}

import java.util.*;

public class Bus {
    private static final int MAXNUMPASSENGER = 50;
    private static int bid = 0;

    private int id;
    private int numFree;
    private List<Station> sl;

    private Iterator<Station> iterator;
    private Station nextStation;
    private boolean toExit;

    public Bus() {
        this.id = Bus.nextId();
        this.numFree = MAXNUMPASSENGER;
        this.sl = new LinkedList<Station>();
        this.iterator = sl.iterator();
        this.toExit = false;
    }

    public static synchronized int nextId() {
        return bid++;
    }

    public synchronized void addStation(Station newStation) {
        sl.add(newStation);
    }

    public synchronized Station getLastStation() {
        return sl.get(sl.size() - 1);
    }
}
```

```

public synchronized Station getNextStation() {
    nextStation = iterator.next();
    notifyAll();
    return nextStation;
}

public synchronized void exitBus(Station exitStation) {
    while (!toExit || !exitStation.equals(nextStation)) {
        try {
            wait();
        } catch (InterruptedException e) {
        }
    }
    numFree++;
}

public synchronized void permissionToExit() {
    toExit = true;
    notifyAll();
    try {
        wait(1500);
    } catch (InterruptedException e) {
    }
    toExit = false;
}

public synchronized int getFree () {
    return numFree;
}

public synchronized void setNumFree(int numFree) {
    this.numFree = numFree;
}

}

public class Station {
    private String name;
    private int numPassengers, numFreePlaces;
    private boolean busIN;
    private Bus bus;

    public Station(String name) {
        this.name = name;
        this.numPassengers = 0;
        this.numFreePlaces = 0;
        this.busIN = false;
        this.bus = null;
    }

    public synchronized Bus waitBus() {
        while (true) {
            while (busIN) {
                try {
                    wait();
                }

```

```
        } catch (InterruptedException e) {
    }
}
numPassengers++;
while (!busIN) {
    try {
        wait();
    } catch (InterruptedException e) {
    }
}
numPassengers--;
if (numPassengers == 0)
    notifyAll();
if (numFreePlaces > 0) {
    numFreePlaces--;
    return bus;
}
}
}

public synchronized void busEnter(Bus b) {
    while (busIN) {
        try {
            wait();
        } catch (InterruptedException e) {
        }
    }
    busIN = true;
    bus = b;
    numFreePlaces = b.getNumFree();
    notifyAll();
    while (numPassengers != 0) {
        try {
            wait();
        } catch (InterruptedException e) {
            System.out.println(e);
        }
    }
    busIN = false;
    bus = null;
    b.setNumFree(numFreePlaces);
    notifyAll();
}
}
```

**112****Брига о деци**

Решити проблем чувања деце (*The Child Care Problem*) у програмском језику Java.

**Решење**

```

public class Parent extends Thread{
    private int id;
    private ChildCare cc;
    private int num;

    public Parent(int id, ChildCare cc, int num) {
        this.id = id;
        this.cc = cc;
        this.num = num;
    }

    public void run(){
        while(true){
            starting();

            boolean temp = cc.bringUpChildren(num);
            if(temp) {
                work();
                cc.bringBackChildren(num);
            }
        }
    }

    protected void work() {
        ...
    }

    protected void starting() {
        ...
    }
}

public class Nanny extends Thread {
    private int id;
    private ChildCare cc;

    public Nanny(int id, ChildCare cc) {
        this.id = id;
        this.cc = cc;
    }

    public void run() {
        while (true) {
            starting();
            cc.nannyEnter();
        }
    }
}

```

```
        working();
        cc.nannyExit();
    }
}

protected void starting() {
    ...
}

protected void working() {
    ...
}

}

public class ChildCare {
    public static final int C = 3;

    private int numChild, numNanny, numWaiting;
    Semaphore mutex, confirm, toLeave;

    public boolean bringUpChildren(int num) {
        boolean result = false;
        mutex.waits();

        if ((numChild + num) <= C * numNanny) {
            numChild = numChild + num;
            result = true;
        } else {
            result = false;
        }
        mutex.signals();
        return result;
    }

    public void bringBackChildren(int num) {
        int out;
        mutex.waits();
        numChild = numChild - num;
        out = numNanny - (numChild + C - 1) / C;
        if (out > numWaiting)
            out = numWaiting;
        for (int i = 0; i < out; i++) {
            toLeave.signals();
            confirm.waits();
        }
        mutex.signals();
    }

    public void nannyEnter() {
        mutex.waits();
        numNanny++;
        if (numWaiting > 0) {
            toLeave.signals();
        }
    }
}
```

```

        confirm.waits();
    }
    mutex.signals();
}

public void nannyExit() {
    mutex.waits();
    if (numChild <= C * (numNanny - 1)) {
        numNanny--;
        mutex.signals();
    } else {
        numWaiting++;
        mutex.signals();
        toLeave.waits();
        numNanny--;
        numWaiting--;
        confirm.signals();
    }
}

public ChildCare() {
    numChild = 0;
    numNanny = 0;
    numWaiting = 0;
    mutex = new Semaphore();
    mutex.initS(1);
    confirm = new Semaphore();
    confirm.initS(0);
    toLeave = new Semaphore();
    toLeave.initS(0);
}
}

public class ChildCareTest {
    public static final int NUMPARENT = 10;
    public static final int NUMNANN = 10;

    public static void main(String[] args) {
        ChildCare cc = new ChildCare();
        Parent[] parent = new Parent[NUMPARENT];
        for (int i = 0; i < NUMPARENT; i++) {
            parent[i] = new Parent(i, cc,
                1 + (int) (Math.random() * 5));
            parent[i].start();
        }
        Nanny[] nanny = new Nanny[NUMNANN];
        for (int i = 0; i < NUMNANN; i++) {
            nanny[i] = new Nanny(i, cc);
            nanny[i].start();
        }
    }
}
}

```

113

Јавни тоалет

Постоји тоалет капацитета  $N$  ( $N > 1$ ) који могу да користе жене, мушкарци, деца и домар такав да важе следећа правила коришћења: у исто време се у тоалету не могу наћи и жене и мушкарци; деца могу да деле тоалет и са женама и са мушкарцима; дете може да буде у тоалету само ако се тамо налази барем једна жена или мушкарац; домар има ексклузивно право коришћења тоалета (*The Single Bathroom Problem*). Написати програм за жене, мушкарце, децу и домара који долазе до тоалета, користе га и напуштају га користећи језик Java.

## Решење

```
public class Woman extends Thread{
    private Toilet toilet;

    public Woman(Toilet toilet){
        this.toilet = toilet;
    }
    public void run(){
        ...
        toilet.enterWoman();
        useToilet();
        toilet.exitWoman();
        ...
    }
    public void useToilet(){
        ...
    }
}
public class Man extends Thread{
    private Toilet toilet;

    public Man(Toilet toilet){
        this.toilet = toilet;
    }
    public void run(){
        ...
        toilet.enterMan();
        useToilet();
        toilet.exitMan();
        ...
    }
    public void useToilet(){
        ...
    }
}
public class Child extends Thread{
    private Toilet toilet;

    public Child(Toilet toilet){
        this.toilet = toilet;
    }
}
```

```

public void run() {
    ...
    toilet.enterChild();
    useToilet();
    toilet.exitChild();
    ...
}
public void useToilet() {
    ...
}

}

public class Janitor extends Thread{
    private Toilet toilet;

    public Janitor(Toilet toilet){
        this.toilet = toilet;
    }
    public void run(){
        ...
        toilet.enterJanitor();
        work();
        toilet.exitJanitor();
        ...
    }
    public void work(){
        ...
    }
}

public class Toilet {
    private static int N = 10;
    private int numWoman;
    private int numMan;
    private int numChild;
    private int numJanitor;

    public Toilet() {
        numWoman = 0;
        numMan = 0;
        numChild = 0;
        numJanitor = 0;
    }
    public synchronized void enterWoman(){
        while (((numWoman + numChild) == N)
            || (numMan != 0) || (numJanitor != 0)){
            try {
                wait();
            } catch (InterruptedException e) { }
        }
        numWoman++;
        if((numWoman == 2) && (numChild != 0)) notifyAll();
        //mozda postoji zena koja ceka da izadje
    }
    public synchronized void enterMan(){
}

```

```
while ((numWoman != 0)
       || ((numMan + numChild) == N) || (numJanitor != 0)){
    try {
        wait();
    } catch (InterruptedException e) { }
}
numMan++;
if((numMan == 2) && (numChild != 0)) notifyAll();
//mozda postoji muskarac koji ceka da izadje
}
public synchronized void enterChild(){
    while (((numWoman + numMan) == 0)
           || ((numWoman + numMan + numChild) == N)
           || (numJanitor != 0)){
        try {
            wait();
        } catch (InterruptedException e) { }
    }
    numChild++;
}
public synchronized void enterJanitor(){
    while (((numWoman + numMan + numChild) != 0)
           || (numJanitor != 0)){
        try {
            wait();
        } catch (InterruptedException e) { }
    }
    numJanitor++;
}
public synchronized void exitWoman(){
    while ((numWoman == 1) && (numChild != 0)){
        try {
            wait();
        } catch (InterruptedException e) { }
    }
    numWoman--;
    notifyAll();
}
public synchronized void exitMan(){
    while ((numMan == 1) && (numChild != 0)){
        try {
            wait();
        } catch (InterruptedException e) { }
    }
    numMan--;
    notifyAll();
}
public synchronized void exitChild(){
    numChild--;
    notifyAll();
}
public synchronized void exitJanitor(){
    numJanitor--;
    notifyAll();
}
```

```
}

public class TestToilet {
    public static final int NUMWOMAN = 15;
    public static final int NUMMAN = 10;
    public static final int NUMCHILD = 17;
    public static final int NUMJANITOR = 2;
    public static void main(String [] args){
        Toilet t = new Toilet();
        Woman [] w = new Woman[NUMWOMAN];
        for(int i = 0; i < NUMWOMAN; i++){
            w[i] = new Woman(t);
            w[i].start();
        }
        Man [] m = new Man[NUMMAN];
        for(int i = 0; i < NUMMAN; i++){
            m[i] = new Man(t);
            m[i].start();
        }
        Child [] c = new Child[NUMCHILD];
        for(int i = 0; i < NUMCHILD; i++){
            c[i] = new Child(t);
            c[i].start();
        }
        Janitor [] j = new Janitor[NUMJANITOR];
        for(int i = 0; i < NUMJANITOR; i++){
            j[i] = new Janitor(t);
            j[i].start();
        }
    }
}
```

114

Деда Мраз

Деда Мраз који живи на северном полу већи део свог времена проводи спавајући (*The Santa Claus Problem*). Могу га пробудити или уколико се испред врата појаве свих 9 његових ирваса или 3 од укупно 10 патуљака. Када се Деда Мраз пробуди он ради једну од следећих ствари: Уколико га је пробудила група ирваса одмах се спрема и креће на пут да подели деци играчке. Када се врати са пута свим ирвасима даје награду. Уколико га је пробудила група патуљака онда их он уводи у своју кућу, разговара са њима и на крају их испрати до излазних врата. Група ирваса треба да буде опслужена пре групе патуљака. Написати програм на језику Java који симулира описани систем.

## Решење

```

public class Elf extends Thread{
    private int id;
    private SantaClausHouse sc;

    public Elf(int id, SantaClausHouse sc){
        this.id = id;
        this.sc = sc;
    }
    private void work(){
        ...
    }
    private void talk(){
        ...
    }
    public void run(){
        for(;;){
            work();
            sc.wakeupSantaE();
            talk();
            sc.exitTheRoom();
        }
    }
}
public class Reindeer extends Thread{
    private int id;
    private SantaClausHouse sc;

    public Reindeer(int id, SantaClausHouse sc){
        this.sc = sc;
        this.id = id;
    }
    private void rest(){
        ...
    }
    private void riding(){
        ...
    }
    public void run(){

```

```

        for(;;){
            rest();
            sc.wakeupSantaR();
            riding();
            sc.exitTheSleigh();
        }
    }

public class SantaClaus extends Thread{
    private int dir;
    private SantaClausHouse sc;
    public SantaClaus(SantaClausHouse sc) {
        this.sc = sc;
    }
    private void talk(){
        ...
    }
    private void riding(){
        ...
    }
    public void run(){

        for(;;){
            dir = sc.sleeping();
            if(dir == SantaClausHouse.ELVES){
                talk();
                sc.showoutElves();
            }
            else{
                riding();
                sc.unharnessReindeers();
            }
        }
    }
}

public class SantaClausHouse {
    public static final int NUMREINDEER = 9;
    public static final int NUMELVES = 10;
    private static final int MINELVES = 3;
    private static final int MINREINDEER = 9;
    private int elvesAtTheDoor;
    private int reindeerAtTheDoor;
    private int elvesInTheRoom;
    private int reindeerInTheSleigh;
    private boolean wakeupE, wakeupR;
    private boolean enterElves, exitElves;
    private boolean enterReindeers, exitReindeers;
    private boolean isSleeping;
    public static final int ELVES = 0;
    public static final int REINDEERS = 1;
    public SantaClausHouse() {
        elvesAtTheDoor = 0;
        reindeerAtTheDoor = 0;
        elvesInTheRoom = 0;
    }
}

```

```
reindeerInTheSleigh = 0;
wakeupE = false;
wakeupR = false;
enterElves = false;
exitElves = false;
enterReindeers = false;
exitReindeers = false;
isSleeping = true;
}
public synchronized void wakeupSantaE() {
    while (!isSleeping) {
        try {
            wait();
        } catch (InterruptedException e) {
        }
    }
    elvesAtTheDoor++;
    if (elvesAtTheDoor == MINELVES) {
        wakeupE = true;
        notifyAll();
    }
    while (!enterElves) {
        try {
            wait();
        } catch (InterruptedException e) {
        }
    }
    elvesAtTheDoor--;
    elvesInTheRoom++;
    if (elvesAtTheDoor == 0)
        notifyAll();
}
}

public synchronized void exitTheRoom() {
    while (!exitElves) {
        try {
            wait();
        } catch (InterruptedException e) {
        }
    }
    elvesInTheRoom--;
    if (elvesInTheRoom == 0)
        notifyAll();
}

public synchronized void wakeupSantaR() {
    while (!isSleeping) {
        try {
            wait();
        } catch (InterruptedException e) {
        }
    }
    reindeerAtTheDoor++;
    if (reindeerAtTheDoor == MINREINDEER) {
        wakeupR = true;
    }
}
```

```

        notifyAll();
    }
    while (!enterReindeers) {
        try {
            wait();
        } catch (InterruptedException e) {
        }
    }
    reindeerAtTheDoor--;
    reindeerInTheSleigh++;
    if (reindeerAtTheDoor == 0)
        notifyAll();
}

public synchronized void exitTheSleigh() {
    while (!exitReindeers) {
        try {
            wait();
        } catch (InterruptedException e) {
        }
    }
    reindeerInTheSleigh--;
    if (reindeerInTheSleigh == 0)
        notifyAll();
}

public synchronized int sleeping() {
    isSleeping = true;
    notifyAll();
    while (!(wakeupR || wakeupE)) {
        try {
            wait();
        } catch (InterruptedException e) {
        }
    }
    isSleeping = false;
    if (wakeupR) {
        wakeupR = false;
        enterReindeers = true;
        notifyAll();
        while (reindeerAtTheDoor != 0) {
            try {
                wait();
            } catch (InterruptedException e) {
            }
        }
        enterReindeers = false;
        return REINDEERS;
    } else {
        wakeupE = false;
        enterElves = true;
        notifyAll();
        while (elvesAtTheDoor != 0) {
            try {
                wait();
            }
        }
    }
}

```

```
        } catch (InterruptedException e) {
    }
}
enterElves = false;
return ELVES;
}

}

public synchronized void showoutElves() {
    exitElves = true;
    notifyAll();
    while (elvesInTheRoom != 0) {
        try {
            wait();
        } catch (InterruptedException e) {
        }
    }
    exitElves = false;
}

public synchronized void unharnessReindeers() {
    exitReindeers = true;
    notifyAll();
    while (reindeerInTheSleigh != 0) {
        try {
            wait();
        } catch (InterruptedException e) {
        }
    }
    exitReindeers = false;
}

}

public class SantaClausTest{
    public static void main(String [] vpar){
        SantaClausHouse sc = new SantaClausHouse();
        SantaClaus santa = new SantaClaus(sc);
        Elf [] elves = new Elf[SantaClausHouse.NUMELVES];
        Reindeer [] reindeers =
            new Reindeer[SantaClausHouse.NUMREINDEER];
        for(int i = 0; i < SantaClausHouse.NUMELVES; i ++){
            elves[i] = new Elf(i, sc);
            elves[i].start();
        }
        for(int i = 0; i < SantaClausHouse.NUMREINDEER; i ++){
            reindeers[i] = new Reindeer(i, sc);
            reindeers[i].start();
        }
        santa.start();
    }
}
```

115

Међусобно искључивање - *spin locks*

Користећи само приступ променљивама означеним као *volatile* и пакет за рад са атомским променљивама, *java.util.concurrent.atomic.\**, обезбедити међусобно искључивање приликом приступа неком ресурсу.

## Решење

Као једна од техника за синхронизацију између нити може се користити и запослено чекање (*busy waiting*) на дељеној променљиви (*spin locks*). Код ове технике се непрекидно испитује да ли је услов за наставак рада испуњен. Код запосленог чекања треба водити рачуна о променљивама којима се приступа, њиховим карактеристикама, оптимизацијама које преводилац може да учини, као и атомским инструкцијама или функцијама које су доступне. Известан број алгоритама користи само стандардне инструкције, док неки користе специјалне инструкције које раде већи број операција недељиво. У програмском језику Јава се за синхронизацију између нити могу користити променљиве означене као нестабилне, *volatile*, за које преводилац гарантује да ће им се у преведеном коду приступати у истом редоследу као и у извornом коду, као и да ће им се директно приступати, а не сачуваним копијама. Такође се за синхронизацију могу користити класе из пакета *java.util.concurrent.atomic* које дозвољавају рад са атомским методама. Ове методе у позадини користе методе класе *VarHandle* која омогућава директан и атомски приступ до променљивих. Уколико се запослено чекање, уз периодично одлагање провере, користи на правilan начин могу се добити перформансе које превазилазе остала решења. Такође уколико се ова технике не користи на правilan начин може да доведе до губитка перформанси и до загушења ресурса и у мултипроцесорским системима.

### Петерсонов алгоритам (*Peterson's algorithm*) – решење за две нити

Један од основних симетричних алгоритама за међусобно искључивање два процеса се назива Петерсонов алгоритам (*Peterson's algorithm*). Код овог алгоритма процес *X* који жели да уђе у критичну секцију поставља информацију о томе у дељену променљиву *inX* (*inX = true*). А неко тога поставља и информацију о томе да је то последњи пристигли процес који жели да приступи датој секцији уписом свог идентификатора у дељену променљиву (*last = X*). Ова два уписа у дељене променљиве се морају извршити у редоследу који је наведене. Након овога процес добија право да приступи критичној секцији или ако други процес не жели да приступи критичној секцији или ако је други процес стигао касније и уписао вредност различиту од *X* у дељену променљиву *last*. Како би се гарантовало да је редослед по коме ће се у време извршавања приступати дељеним променљивама које једна нити обавља исти као редослед који је наведен у извornом коду потребно је дате променљиве означити као нестабилне (*volatile*). Упис у променљиву означену са *volatile* се догађа пре сваког наредног читања или уписа у ту променљиву. Инструкцијама које се налазе пре или после приступа *volatile* променљивама се приликом превођења и извршавања може мењати редослед, али се међусобно не мешају. Оне инструкције које су биле пре остају пре и оне које су биле после остају после. Ово се може искористити приликом синхронизације тако што се променљиве *in0*, *in1* и *last* означе као нестабилне, *volatile*.

```
public class Peterson extends SimpleLock {
    private volatile boolean in0, in1;
```

```
private volatile int last;

public Peterson() {
    in0 = false;
    in1 = false;
    last = 0;
}

class CS0 extends SimpleLock {

    @Override
    public void lock() {
        /* entry protocol CS0 */
        in0 = true;
        last = 0;
        while (in1 && last == 0) {
            Thread.onSpinWait();
        }
    }

    @Override
    public void unlock() {
        /* exit protocol CS0 */
        in0 = false;
    }
}

class CS1 extends SimpleLock {

    @Override
    public void lock() {
        /* entry protocol CS1 */
        in1 = true;
        last = 1;
        while (in0 && last == 1) {
            Thread.onSpinWait();
        }
    }

    @Override
    public void unlock() {
        /* exit protocol CS1 */
        in1 = false;
    }
}

@Override
public SimpleLock create(int id) {
    SimpleLock lock = (id == 0 ? new CS0() : new CS1());
    return lock;
}
}
```

Запослено чекање захтева непрекидну проверу услова блокирања. Ова провера захтева читање и евентуалну промену једне или више променљивих, као и њихово поређење. Уколико се услови блокирања споро мењају процеси ће велики број пута обавити проверу пре него што стекну право приступа критичној секцији. Сама непрекидна провера може да изазове велико оптерећење процесора, загушење кеш и оперативне меморије, као и мрежних ресурса (*High contention*). Како би се смањило загушење у неким случајевима се уместо непрекидне провере може применити одлагање провере како би се дао простор нити која држи критичну секцију да из ње изађе. У класи за рад са нитима, *Thread*, се могу наћи три статичке методе које се могу искористити у ове сврхе: *yield*, *onSpinWait* и *sleep*.

Метода *public static void yield()* даје наговештај распоређивачу да дата нит жели својевољно да привремено обустави своје извршавање како би контролу добила нека друга нит. Распоређивач може да игнорише овај наговештај. Имплементација ове методе је зависна од коришћеног оперативног система и верзије Јаве.

Метода *public static void onSpinWait()* индицира да је нит која је позива ову методу тренутно није у могућности да даље напредује. Напредовање ће бити могуће када се одређени услови испуне. Позивањем ове методе у свакој итерацији нит даје до знања окружењу да се ради о запосленом чекању на дељеним променљивама. Окружење може, али не мора, да предузме мере како би побољшао перформансе. Ова метода се појавила почев од верзије 9. Тренутна имплементација ове методе је празна, а индикација окружењу је дата кроз анотацију *@HotSpotIntrinsicCandidate*.

Методе *public static void sleep(long millis) throws InterruptedException* изазива да се извршавање тренутне нити привремено заустави на одређени временски период који је специфициран у милисекундама. У време док је низ заустављена не губи се ексклузивно право приступа мониторима. Уколико се из неке друге нити у време док је дата нит била привремено заустављена позове метода *interrupt* емитоваће се изузетак, *InterruptedException*. Поред ове методе постоји и метода *public static void sleep(long millis, int nanos) throws InterruptedException* која се блокира одређен број милисекунди и наносекунди, али ово време зависи од могућности система на коме се извршава.

У случају позивања метода *yield* и *onSpinWait* распоређивачу и окружењу се препушта да дату нит заустави, док је у случају методе *sleep* на кориснику да одреди временски период за чекање. Како одредити овај временски период? Могуће је поставити да све нити чекају исти временски период. Ово може да доведе до проблема уколико временски период није добро одабран. Уместо овог приступа чешће се примењује приступ код кога нити чекају случајни период времена, који није дужи од задатог максималног периода. Након истека периода нити се буде и поново проверавају услов. Уколико услов није испуњен нит поново чека. У овим случајевима се такође често примењује варијанта код које се интервал чекања увећава, најчешће удвостручује (*exponential back-off protocol*) све док се на постигне нека максимална вредност преко које се не иде. Овај механизам увећавања интервала је преузет из синхронизације код мрежних протокола.

### Тикет алгоритам (*Ticket algorithm*)

Тикет алгоритам има за циљ да обезбеди правичан улазак у критичну секцију уколико постоји два или више процеса. Код овог алгоритма је потребно да на почетку сваки процес добија јединствени редни број (*number*), сходно редоследу доласка, који такође означава и редослед по коме ће започети приступ критичној секцији. Када је добио редни број процес чека све док се редни број не изједначи са редним бројем процеса који следећи сме да приступи критичној секцији (*next*). Када заврши приступ критичној секцији увећава редни број процеса који следећи сме да приступи критичној секцији. Овде треба водити рачуна о додељивању и инкрементирању редних бројева, *number*,

као и о додељивању који је то следећи број процеса који сме да крене да се извршава, *next*.

Уколико би се променљиве *number* и *next* само означиле као нестабилне, *volatile*, и заједничке за читаву класу добило би се следеће решење које не би дало исправно резултат.

```
public class Ticket_deadlock extends SimpleLock {  
  
    private volatile int number;  
    private volatile int next;  
  
    public Ticket_Deadlock() {  
        number = 0;  
        next = 0;  
    }  
  
    @Override  
    public void lock() {  
        /* entry protocol */  
        int turn = number++;  
        while (turn != next) {  
            Thread.onSpinWait();  
        }  
    }  
  
    @Override  
    public void unlock() {  
        /* exit protocol */  
        next = next + 1;  
    }  
}
```

Проблем би настао код читања вредности и увећавања јединственог редног броја (*number++*), јер се ово састоји из више операција које су атомске. Прва је читање старе вредности редног броја, друга је увећавање вредности за један и трећа је смештање резултата. Овде треба приметити да постоји велика разлика између приступа променљиви која је означене као нестабилна, *volatile*, и синхронизованог приступа, *synchronized*. Код *volatile* променљиве читање и упис су атомски, али већи број читања и уписа могу да изазову утркивање (*race condition*) и последица је истовременог приступа дељеним ресурсима од стране више процеса. Да би се овај проблем решио (елиминисала временска зависност у програму) потребно је да се обе конкурентне операције изврше атомски (недељиво, без прекидања). Код синхронизованог приступа све акције се извршавају атомски, али захтевају закључавање монитора.

Механизма мониторског закључавања читавог објекта може да буде спор у случају да је потребно обавити једноставну операцију над променљивама нумеричког типа. Како би се разрешио овај проблем без употребе мониторског закључавања објекта у програмском језику Java поред развоја нових монитора, кључна реч *synchronized*, могу користити и већ развијене класе унутар *java.util.concurrent.atomic* пакета. Овај пакет се појавио почев од верзије 1.5 и пружа већи број класа које без закључавања објекта (*lock-free*) на сигуран начин (*thread-safe*) омогућавају коришћење појединачних променљивих. Ове класе имају за циљ да прошире појам нестабилне променљиве, *volatile*, на атомску манипулацију са поглјима и низовима. Почев од верзије 9 инстанце атомских класа одржавају вредности којима се приступа и које се ажурирају користећи методе атомске класе *VarHandle*.

За рад са нумеричким типовима података, `java.lang.Number`, се могу користити класе `AtomicInteger` и `AtomicLong`. Основне методе класа би обухватале дохватање вредности (`get()`), постављање вредности (`set(value)`), дохватање старе и постављање нове вредности (`getAndSet(newValue)`), дохватање и инкрементирање/декрементирање (`getAndIncrement()`/`getAndDecrement()`), инкрементирање/декрементирање и дохватање (`incrementAndGet()`/`decrementAndGet()`), дохватање и увећавање (`getAndAdd(value)`), увећавање и дохватање (`addAndGet(value)`), упореди и постави (`compareAndSet(int expectedValue, int newValue)`), упореди и замени (`compareAndExchange(expectedValue, newValue)`), као и низ других метода које се могу примењивати уколико је познато да нема других нити које истовремено приступају датом објекту. Почек од верзије 9 меморијски ефекти које ове методе имају су специфицирани на исти начин као код метода истог назива класе `VarHandle`.

```
import java.util.concurrent.atomic.AtomicInteger;

public class Ticket extends SimpleLock {

    private AtomicInteger number;
    private volatile int next;

    public Ticket() {
        number = new AtomicInteger(0);
        next = 0;
    }

    @Override
    public void lock() {
        /* entry protocol */
        int turn = number.getAndAdd(1);
        // int turn = number.getAndIncrement();
        while (turn != next) {
            Thread.onSpinWait();
        }
    }

    @Override
    public void unlock() {
        /* exit protocol */
        next = next + 1;
    }
}
```

Овде треба приметити да нема потребе да променљива `next` буде атомска, `AtomicInteger`, јер се увећавање ове променљиве објављује у тренутку када дата нит има ексклузивно право приступа ресурсу.

### Приступ „Провери и постави“ (`Test and Set`)

Међусобно искључивање се може реализовати и у случају да је могуће атомски реализовати провера (дохватање) постојеће вредности и постављање нове вредности неке променљиве. У том случају би услов за проверу да ли процес може приступити критичној секцији, `<await (!lock) lock = true;>`, могао исказати као `while (TS(lock)) skip`. За атомски рад са логичким типовима података се може користити класа `AtomicBoolean`. Ова класа има сличан скуп метода као и класе `AtomicInteger` и `AtomicLong`. Основне

методе класа би обухватале дохватање вредности (`get()`), постављање вредности (`set(value)`), дохватање старе и постављање нове вредности (`getAndSet(newValue)`), упореди и замени (`compareAndExchange(expectedValue, newValue)`), као и друге методе. Меморијски ефекти ових метода су специфицирани на исти начин као код метода истог назива класе `VarHandle`.

```
import java.util.concurrent.atomic.AtomicBoolean;

public class TestAndSet extends SimpleLock {

    private final AtomicBoolean lock;

    public TestAndSet() {
        lock = new AtomicBoolean(false);
    }

    @Override
    public void lock() {
        /* entry protocol */
        while (lock.getAndSet(true)) {
            Thread.onSpinWait();
        }
    }

    @Override
    public void unlock() {
        /* exit protocol */
        lock.set(false);
    }
}
```

#### Приступ „Провери и провери и постави“ (*Test and Test and Set*)

Приликом извршавања атомске функције `getAndSet` врши се додела нове вредности променљивој и читање старе вредности ове променљиве. Семантика доделе вредности је иста као да је интерна променљива означена као *volatile*. Код описаног решења троши се доста време на ажурирање података у меморији јер је при сваком позиву потребно обезбедити важење релације „догодило се пре“ (*happens-before relationship*) јер се интерна променљива понаша као *volatile*. Како би ублажили ефекти може се применити следеће решење код кога се меморијски модел ажурира само када неко уђе у критичну секцију, а не стално:

```
public class TestAndTestAndSet extends SimpleLock {

    private final AtomicBoolean lock;

    public TestAndTestAndSet() {
        lock = new AtomicBoolean(false);
    }

    @Override
    public void lock() {
        /* entry protocol */
```

```

        while (lock.get()) {
            Thread.onSpinWait();
        }
        while (lock.getAndSet(true)) {
            while (lock.get()) {
                Thread.onSpinWait();
            }
        }
    }

@Override
public void unlock() {
    /* exit protocol */
    lock.set(false);
}
}

```

### Бекери алгоритам (*Lamport's Bakery algorithm*) – решење за две нити

Овај алгоритам представља модификацију тикет алгоритма, са том разликом шта није потребно постојање додатних атомских операција или метода. Разлика у односу на тикет алгоритам се огледа у томе што процес који жели да приступи критичној секцији добија вредност за један већу од вредности другог процеса. Одређивање ко може да приступи критичној секцији се врши узајамном провером вредности које су процеси поставили. Један процес поставља променљиву *turn0* а проверава и *turn1*, док други поставља променљиву *turn1* а проверава и *turn0*. Како би се решио овај проблем променљиве *turn0* и *turn1* означе као нестабилне, *volatile*, и заједничке за читаву класу.

```

public class Bakery extends SimpleLock {

    private volatile int turn0, turn1;

    public Bakery() {
        turn0 = 0;
        turn1 = 0;
    }

    class CS0 extends SimpleLock {

        @Override
        public void lock() {
            /* entry protocol CS0 */
            turn0 = 1;
            turn0 = turn1 + 1;
            while (turn1 != 0 & turn0 > turn1) {
                Thread.onSpinWait();
            }
        }

        @Override
        public void unlock() {
            /* exit protocol CS0 */
            turn0 = 0;
        }
    }
}

```

```
    }

}

class CS1 extends SimpleLock {

    @Override
    public void lock() {
        /* entry protocol CS1 */
        turn1 = 1;
        turn1 = turn0 + 1;
        while (turn0 != 0 && turn1 >= turn0) {
            // Thread.onSpinWait();
        }
    }

    @Override
    public void unlock() {
        /* exit protocol CS1 */
        turn1 = 0;
    }

}

@Override
public SimpleLock create(int id) {
    SimpleLock lock = (id == 0 ? new CS0() : new CS1());
    return lock;
}
}
```

Овде треба приметити да је претходно решење асиметрично. Први процес, CS0, проверава да ли други процес, CS1, жели да приступи критичној секцији,  $turn1 \neq 0$ , и да ли је стигао касније,  $turn0 > turn1$ . Други процес, CS1, проверава да ли први процес, CS0, жели да приступи критичној секцији,  $turn0 \neq 0$ , и да ли је стигао касније или истовремено са њим,  $turn1 \geq turn0$ . Разлог за ово лежи у чињеници да се желело избеги мртво блокирање које би настало уколико би оба процеса проверавала услов веће а имају исту вредност променљива. До ситуације да имају исту вредност би се могло доћи уколико би први процес поставио променљиву  $turn0$  на вредност 1 па након тога други процес постави променљиву  $turn1$  на вредност 1. Ако након тога први процес прочита променљиву  $turn1$  она има вредност 1, а ако тада а други процес прочита променљиву  $turn0$  она има такође вредност 1. Тек након овога оба процеса постављају своје променљиве на вредност 2.

Претходно решење би се могло учинити симетричним увођењем оператора веће за скуп од два елемента  $(a,b) > (c,d)$ . Овај оператор враћа *true* ако и само ако је  $a > c$  или  $a == c$  и  $b > d$  и *false* у свим другим случајевима. Тада би се услов могао приказати као:  $(turn[i], i) > (turn[j], j)$ .

Овде треба приметити да оператор не мора да се обавља атомски.

Класа *ThreadLocal<T>* пружа могућност за рад са променљивама које су локалне за неку појединачну нит. Ове променљиве се разликују од уобичајених променљивих по том што свака нит која им приступа има своју независну променљиву, која представља независно иницијализовану копију променљиве. Приступ се обавља кроз *set* и *get*

методе. Ове променљиве су типично означене као приватна и статичка, *private static*, поља класе и користе се за идентификацију стања нити (нпр. идентификатор нити или трансакције). Ова класе се може искористити за идентификацију нити у класи, *id*. Када нека нит први пут позове статичку методу *ThreadId.getId()* додељује се јединствен идентификатор датој нити. Овде је узето да све инстанце класе *SimpleLock* имају као своје поље инстанцу класе *ThreadId*, а метода *getId* није статичка јер је број нити који приступа неком објекту фиксан, а у описаним алгоритмима нити имају идентификаторе у интервалу од 0 до *N-1*:

```
import java.util.concurrent.atomic.AtomicInteger;

public class ThreadId {
    private final AtomicInteger nextId = new AtomicInteger(0);

    private final ThreadLocal<Integer> threadId =
        new ThreadLocal<Integer>() {
            @Override
            protected Integer initialValue() {
                return nextId.getAndIncrement();
            }
        };

    public int getId() {
        return threadId.get();
    }
}
```

Могло би се помислiti да би претходно решење могло сажети тако што би се променљиве *turn0* и *turn1* сместиле као елементи низа, *turn[]*, који би био декларисан као нестабилан, *volatile*, и заједнички за читаву класу. Овакво решење би могло бити неисправно јер је низ декларисан као нестабилан, а не елементи низа.

```
public class Bakery_Deadlock extends SimpleLock {

    private volatile int[] turn;

    public Bakery_Deadlock() {
        int[] temp = { 0, 0 };
        turn = temp;
    }

    @Override
    public void lock() {
        /* entry protocol */
        int id = getId();
        int j = other(id);
        turn[id] = 1;
        turn[id] = turn[j] + 1;
        while (turn[j] != 0 &&
               (turn[id] > turn[j]
                || ((turn[id] == turn[j]) && (id > j)))) {
            Thread.onSpinWait();
        }
    }
}
```

```
    @Override
    public void unlock() {
        /* exit protocol */
        int id = getId();
        turn[id] = 0;
    }

    private static final int other(int id) {
        return id == 0 ? 1 : 0;
    }
}
```

Како би се разрешио овај проблем уместо нестабилног низа целобројних вредности се користи низ атомских целобројних вредности, *AtomicInteger*.

```
import java.util.concurrent.atomic.AtomicInteger;

public class Bakery2_Atomic extends SimpleLock {

    private final AtomicInteger[] turn;

    public Bakery2_Atomic() {
        AtomicInteger[] tmp = {
            new AtomicInteger(0), new AtomicInteger(0) };
        turn = tmp;
    }

    @Override
    public void lock() {
        /* entry protocol */
        int id = getId();
        int j = other(id);
        turn[id].set(1);
        turn[id].set(turn[j].get() + 1);
        while (turn[j].get() != 0
                && (turn[id].get() > turn[j].get()
                    || ((turn[id].get() == turn[j].get())
                        && (id > j)))) {
            Thread.onSpinWait();
        }
    }

    @Override
    public void unlock() {
        /* exit protocol */
        int id = getId();
        turn[id].set(0);
    }

    private static final int other(int id) {
        return id == 0 ? 1 : 0;
    }
}
```

## Тај брејкер алгоритам (*Tie-Breaker algorithm*) – решење за N нити

Ово решење представља генерализацију Петерсеновог алгоритма који је разматран за два процеса. Основна идеја алгоритма је иста као код Петерсоновог алгоритма за два процеса где процес може да напредује само ако није последњи процес који је пристигао. Уводи се  $N$  стања кроз која се пролази. Процес може да пређе у наредно стање само ако није последњи процес који је ушао у то стање а хтето је да уђе у то стање. Ово обезбеђује да у наредно стање може да пређе за један процес мање од броја процеса који су могли да уђу у дато стање. Потошто има  $N$  стања кроз која се пролази и у сваком кораку отпада по један процес не може да се деси да свих  $N$  процеса чека на улазак у критичну секцију већ само онај који је у највишем стању. Из последњег стања у критичну секцију улази онај процес који је последњи ушао у последње стање. У случају  $N$  процеса за сваки процес се уводи променљива која представља информацију о стању до код је дошао дати процес у покушају да дође до критичне секције,  $in[N]$ , такође се уводи и променљива за свако стање кроз које процеси могу да прођу која садржи информацију о томе који је процес последњи дошао до датог стања,  $last[N]$ . Ове променљиве,  $in$  и  $last$  се у поступку иницијализације, а променљива  $in$  и по завршеном приступу критичној секцији постављају на вредност 0 како би поставили индикацију да се налазе у почетном стању.

Сваки процес пролази кроз стање тако што чека докле год има неки процес који је у вишем стању од његовог стања и чека да неки други процес уђе у његово стање, да би он могао да пређе у следеће стање, тј. процеси истискују процесе који су пре њих стigli у то стање и на крају је један слободан да уђе у критичну секцију. Овде треба приметити да редослед по коме су процеси долазили у претходна стања не мора да буде редослед по коме ће долазити у наредно стање. Ово може да доведе до тога да процес који је последњи кренуо са радом стигне до последњег стања пре осталих процеса.

Овај проблем би могао, као претходни проблем, да се реши коришћењем низ атомских целобројних вредности, `AtomicInteger`. Други начин је да се користи постојећа класа `AtomicIntegerArray` из пакета `java.util.concurrent.atomic` која омогућава атомску модификацију елемената низа.

```
import java.util.concurrent.atomic.AtomicIntegerArray;

public class TieBreakerN Atomic extends SimpleLock {
    private final int size;

    private final AtomicIntegerArray in;
    private final AtomicIntegerArray last;

    public TieBreakerN Atomic(int size) {
        this.size = size;
        in = new AtomicIntegerArray(size);
        last = new AtomicIntegerArray(size);
        for (int i = 0; i < size; i++) {
            in.set(i, 0);
            last.set(i, 0);
        }
    }

    @Override
    public void lock() {
```

```
/* entry protocol */
int id = getId();
for (int j = 1; j < size; j++) {
    in.set(id, j);
    last.set(j, id);
    for (int k = 0; k < size; k++) {
        while ((id != k) && (in.get(k) >= in.get(id))
               && (last.get(j) == id)) {
            Thread.onSpinWait();
        }
    }
}
}

@Override
public void unlock() {
    /* exit protocol */
    int id = getId();
    in.set(id, 0);
}
```

Овде треба приметити да се у последњем стању може наћи само један процес, односно да је довољно имати за један елемент краћи низ него што има процеса. Односно нулти елемент низа се користи да специфицира да процес не тражи приступ критичној секцији. Овај алгоритам се понекада назива и Филтерски алгоритам јер се у сваком кораку, стању, филтрира по један процес.

Бекери алгоритам (*Lamport's Bakery algorithm*) – решење за N нити

Симетрично решење за две нити се може генерализовати за случај да постоји више процеса. Основна идеја алгоритма је иста као код верзије за два процеса где процес узима тренутни максимум редоследа по коме су процеси пристизали, увећава ту запамћену вредност за један, и може да приступи критичној секцији само ако има запамћену минималну вредност променљиве од свих процеса који тренутно желе да приступи критичној секцији. Када заврши приступ поставља да тренутно не жели да приступи критичној секцији.

У случају  $N$  процеса за сваки процес се уводи променљива која представља информацију о редоследу пристизања,  $turn[N]$ . Вредност 0 означава да процес тренутно не тражи приступ критичном региону, што представља и иницијализациону вредност. На почетку процес поставља свој редослед на вредност 1,  $turn[i] = 1$ , чиме сигнализира да жели да приступи критичном региону. Након тога проналази максимум међу свим процесима који увећава за 1,  $turn[i] = \max(turn[1:n]) + 1$ . Када се одреди максимум процес иде и упоређује своју вредност са сваким другим процесом. Процес чека поређење ако је други процес трајио приступ и ако има већи редни број редоследа приспећа:  $for\ j = 1\ to\ n\ st\ j \neq i\ while\ (turn[j] \neq 0\ and\ (turn[j],i) > (turn[j],j))\ skip$ . На крају када заврши приступ процес поставља свој редослед на вредност 0,  $turn[i] = 0$ .

Овде треба приметити да два или више процеса могу да имају исту вредност променљиве са редоследом,  $turn[]$ , исто као што је могло да се деси и у ситуацији са два процеса. Такође треба приметити да унутрашња петља, у којој се пореди редоследи доласка два процеса, уводи редослед чак и када два или више процеса имају исту вредност ове променљиве. Ово је омогућено коришћењем уведеног оператора поређења. Атомске акције су исте као и у случају два процеса, што значи да

оператор за поређење није атомска операција. Ово такође значи да је одређивање максимума апроксимирано.

```

import java.util.concurrent.atomic.AtomicIntegerArray;

public class BakeryN_Atomic extends SimpleLock {
    private final int size;

    private final AtomicIntegerArray turn;

    public BakeryN_Atomic(int size) {
        this.size = size;
        turn = new AtomicIntegerArray(size);
        for (int i = 0; i < size; i++) {
            turn.set(i, 0);
        }
    }

    @Override
    public void lock() {
        /* entry protocol */
        int id = getId();
        turn.set(id, 1);
        turn.set(id, max(turn) + 1);
        for (int j = 0; j < size; j++) {
            while ((j != id) && (turn.get(j) != 0)
                    && (turn.get(id) > turn.get(j))
                    || (turn.get(id) == turn.get(j)
                        && id > j))) {
                Thread.onSpinWait();
            }
        }
    }

    private int max(AtomicIntegerArray turns) {
        int max = 0;
        for (int i = 0; i < turns.length(); i++) {
            max = Math.max(max, turns.get(i));
        }
        return max;
    }

    @Override
    public void unlock() {
        /* exit protocol */
        int id = getId();
        turn.set(id, 0);
    }
}

```

### Закључавање на нивоу елемента низа (Array Lock) – решење за N нити

Међусобно искључивање нити код кога се гарантује редослед приступа критичној секцији се може постићи користећи FIFO ред. Основна идеја је да свака нит запослено

чека на различитим позицијама у том реду сходно времену доласка. Како би се омогућио приступ различитим елементима у реду користи се кружни бафер (*Array Lock*) који има капацитет који одговара броју нити како се не би водило рачуна о прекорачењу. Пошто већи број нити може бафери да приступи истовремено, а да би свако могао независно да чита и увећава тренутну позицију, може се искористити атомска функција за дохватање и увећавање дељене променљиве. Када нека нит заврши са приступом критичној секцији позиција на којој је дата нит чекала се поново означава да је слободна за чекање, а наредна позиција се означава тако да нит која чека на њој сме да приступи критичној секцији. Овај алгоритам се понекада назива и Андерсонов алгоритам (*Anderson's Lock*) према аутору. Елементи низа на којима нити чекају треба да имају две вредности: слободан и није слободан. Ово би се могло постићи користећи низ логичких променљивих, *boolean [] flag*, али треба бити опрезан и водити рачуна да не дође до мртвог блокирања због Јавиног меморијског модела приликом приступа појединачним променљивама. Постављање низа за нестабилну вредност би могло да помогне, не да би се та вредност стално ажурирала већ да би се преводиоцу сигнализирало да не ради оптимизацију. Уместо овога би се могао користити низ атомских логичких типова, *AtomicBoolean [] flag*, код кога нама датог проблема. Уместо овога је одабрано да се користи низ елемената, *Node[] flag*, код кога сваки елемент има само једно нестабилну променљиву, *volatile boolean free*, како би се отклонио проблем конзистентности. За тренутну позицију се користи атомски цео број, *AtomicInteger tail*. Како би се обезбедило да нит може да ослободи текућу позицију а пошто није дозвољено да се аргументи преноси путем позива методе, *unlock()*, потребно је да дата нит има локалне променљиве у датом објекту. Ово се може постићи употребом локалних променљивих нити, *ThreadLocal<Integer> mySlotIndex*.

```
import java.util.concurrent.atomic.*;

public class AndersonLock extends SimpleLock {
    private final int size;

    private ThreadLocal<Integer> mySlotIndex;
    private AtomicInteger tail;
    private Node[] flag;

    public AndersonLock (int size) {
        this.size = size;
        mySlotIndex = new ThreadLocal<Integer>() {
            protected Integer initialValue() {
                return 0;
            }
        };
        tail = new AtomicInteger(0);
        flag = new Node[size];
        for (int i = 0; i < size; i++) {
            flag[i] = new Node();
        }
        flag[0].free = true;
    }

    @Override
    public void lock() {
        /* entry protocol */
        int slot = tail.getAndIncrement() % size;
        mySlotIndex.set(slot);
    }

    private class Node {
        volatile boolean free;
    }
}
```

```

        mySlotIndex.set(slot);
        while (!flag[slot].free) {
            Thread.onSpinWait();
        }
    }

    @Override
    public void unlock() {
        /* exit protocol */
        int slot = mySlotIndex.get();
        flag[slot].free = false;
        flag[(slot + 1) % size].free = true;
    }

    static final class Node {
        volatile boolean free = false;
    }
}
}

```

Код описаног решења идеја је да се смањи загушење кеш меморије које настаје када већи број нити непрекидно приступа истој меморијској локација тако што нити приступају различитим локацијама. Овај проблем може и даље бити присутан, али у нешто блажем облику, јер се већи број позиција може наћи у истој кеш линији којој приступа већи број нити. Проблем би се могао елиминисати уколико би се у свакој кеш линији налази по једна позиција. Други проблем који се може уочити код овог решења је да је ограничен максималан број нити које могу да приступе неком ресурсу. Како би се омогућило да што већи број нити може да приступи неком ресурсу потребно је алоцирати низ према максималном дозвољеном броју нити, без обзира на то колико нити стварно у неком тренутку жели да приступи неком ресурсу.

### Закључавање користећи листе (CLH Lock) – решење за N нити

Други начин за међусобно искључивање нити код кога се гарантује редослед приступа критичној секцији се може постићи користећи листе. Основна идеја је да свака нит запослено чека на различитим елементима листе сходно времену доласка. За разлику од алгоритма код кога се закључавање обавља на нивоу елемента низа овде није потребно унапред знати број нити које желе да приступе критичној секцији. Како би се одржавала уређена листа унутар класе се чува референцу на последњи елемент листе, *tail*, на коме треба да чека нит која прва следећа нађије. Када нека нит жели да започне приступ критичној секцији она формира нови последњи елемент листе, *node = myNode.get()*, и поставља га на недоступну вредност, *node.locked = true*. Атомски замењује затечени последњи елемент листе тим новим елементом и чита затечени последњи елемент листе. Овим затечени последњи елемент листе постаје претпоследњи елемент, *pred*. Нит чува код себе локално референцу на тај претпоследњи елемент, у објекту *myPred*, и чека док неко не ослободи тај претпоследњи елемент листе. Овде треба приметити да нит такође чува и референцу на наредни елемент листе, *myNode*. Када нека нит заврши са приступом критичној секцији нит дохвата локално сачувану референцу на наредни елемент листе, *node = myNode.get()*, и поставља га на доступну вредност, *node.locked = false*. Како би се приликом започињања приступа критичној секцији фаза формирања новог последњег објекта убрзала уместо креирања новог објекта позивање конструктора, *new*, могуће је рециклирати стари објекат који је сачуван у претходној итерацији, уколико такав објекат постоји. Због тога се на крају приступа критичној секцији у локалном објекту

класе, *myNode*, чува затечени последњи елемент листе, *myPred*. Алгоритам је добио назив по иницијалима ауторима *Craig, Landin* и *Hagersten - CLH Lock*. За атомски рад са референцама које се могу ажурирати користити се класа *AtomicReference<V>*. Мемориски ефекти које метода ове класе имају су специфицирани на исти начин као код метода истог назива класе *VarHandle*.

```
import java.util.concurrent.atomic.*;

public class CLHLock extends SimpleLock {
    AtomicReference<Node> tail;
    ThreadLocal<Node> myPred;
    ThreadLocal<Node> myNode;

    public CLHLock() {
        tail = new AtomicReference<Node>(new Node());
        myNode = new ThreadLocal<Node>() {
            protected Node initialValue() {
                return new Node();
            }
        };
        myPred = new ThreadLocal<Node>() {
            protected Node initialValue() {
                return null;
            }
        };
    }

    @Override
    public void lock() {
        /* entry protocol */
        Node node = myNode.get();
        node.locked = true;
        Node pred = tail.getAndSet(node);
        myPred.set(pred);
        while (pred.locked) {
            Thread.onSpinWait();
        }
    }

    @Override
    public void unlock() {
        /* exit protocol */
        Node node = myNode.get();
        node.locked = false;
        myNode.set(myPred.get());
    }

    static final class Node {
        volatile boolean locked = false;
    }
}
```

Варијанта овог алгоритма синхронизације се користи у програмском језику Јава за синхронизацију на најнижем нивоу. Класу *AbstractQueuedSynchronizer* из пакета *java.util.concurrent.locks*, користе основне синхронизационе класе као што су класа *Semaphore*, из пакета *java.util.concurrent* и класа *Lock*, из пакета *java.util.concurrent.locks*. Модификација овог алгоритма се састоји у томе што је подржано и отказивање приступа ресурсу након истека временског интервала.

Уколико би се приликом сваког започињања приступа критичној секцији креирао нови последњи елемент листе, уместо да се као до сада рециклира, решење би било:

```
public class CLHLock2 extends SimpleLock {
    AtomicReference<Node> tail;
    ThreadLocal<Node> myNode;

    public CLHLock2() {
        tail = new AtomicReference<Node>(new Node());
        myNode = new ThreadLocal<Node>();
    }

    @Override
    public void lock() {
        /* entry protocol */
        Node node = new Node();
        node.locked = true;
        myNode.set(node);
        Node pred = tail.getAndSet(node);
        while (pred.locked) {
            Thread.onSpinWait();
        }
    }

    @Override
    public void unlock() {
        /* exit protocol */
        Node node = myNode.get();
        node.locked = false;
    }

    static final class Node {
        volatile boolean locked = false;
    }
}
```

116

## Едитор – рад са корисничким интерфејсом

Написати програм који омогућава учитавање текстуалне датотеке, промену (редовање) њеног садржаја и снимање сачуваних измена.

### Решење

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import javax.swing.*;

public class Editor extends JFrame {

    private static final long serialVersionUID = 1L;

    JButton open, save;
    JTextArea area;

    String path;

    public Editor() {
        super("Editor");
        area = new JTextArea();
        JScrollPane scroll = new JScrollPane(area);

        open = new JButton("Open");
        open.addActionListener(new OpenListener(this));
        save = new JButton("Save");
        save.addActionListener(new SaveListener(this));

        JPanel buttonPanel = new JPanel();
        buttonPanel.setLayout(new GridLayout(1, 2));
        buttonPanel.add(open);
        buttonPanel.add(save);

        this.setLayout(new GridLayout(2, 1));
        this.add(scroll);
        this.add(buttonPanel);
        this.setSize(800, 600);
        this.setDefaultCloseOperation(EXIT_ON_CLOSE);
        this.setVisible(true);
    }

    class OpenListenerLong implements ActionListener {
        Editor editor;

        public OpenListenerLong(Editor editor) {
            this.editor = editor;
        }
    }
}
```

```

@Override
public void actionPerformed(ActionEvent event) {
    JFileChooser jc = new JFileChooser("./");

    int fStatus = jc.showOpenDialog(editor);
    if (fStatus == JFileChooser.APPROVE_OPTION) {
        String path = jc.getSelectedFile().getPath();
        this.editor.area.setCursor(
            new Cursor(Cursor.WAIT_CURSOR));
        try (FileReader reader = new FileReader(path);
             BufferedReader buffer =
                 new BufferedReader(reader)) {
            String s;
            StringBuilder builder = new StringBuilder();
            while ((s = buffer.readLine()) != null) {
                builder.append(s).append("\n");
            }
            this.editor.area.setText(builder.toString());
            this.editor.area.setCursor(
                new Cursor(Cursor.DEFAULT_CURSOR));
        } catch (Exception e) {
            e.printStackTrace();
        }
        this.editor.path = path;
    }
}

class SaveListenerLong implements ActionListener {
    Editor editor;

    public SaveListenerLong(Editor editor) {
        this.editor = editor;
    }

    @Override
    public void actionPerformed(ActionEvent event) {
        if (this.editor.path == null ||
            this.editor.path.equals("")) {
            JFileChooser jc = new JFileChooser("./");

            int fStatus = jc.showOpenDialog(editor);
            if (fStatus == JFileChooser.APPROVE_OPTION) {
                String path = jc.getSelectedFile().getPath();
                this.editor.path = path;
                save(path);
            }
        } else {
            save(path);
        }
    }

    private void save(String path) {
}
}

```

```
this.editor.area.setCursor(
    new Cursor(Cursor.WAIT_CURSOR));
try (FileWriter writer = new FileWriter(path);
PrintWriter buffer = new PrintWriter(writer, true);) {
    String s = editor.area.getText();
    buffer.append(s);
} catch (Exception e) {
    e.printStackTrace();
}
this.editor.area.setCursor(
    new Cursor(Cursor.DEFAULT_CURSOR));

}
```

Претходно решење у случају учитавања појединих датотека може исказати извесне проблеме. Комплетна обрада догађаја корисничког интерфејса се обавља у једној нити (*Event Dispatch Thread*). Ово значи да комплетан кориснички интерфејс остаје блокиран док се метода која обавља обраду приспелих догађаја (*Event Handler*) не заврши. Ово може да представља проблем уколико је операција која се обавља дуготрајна или блокирајућа. У том случају се може десити да се комплетан кориснички интерфејс „замрзе“ и да престане да реагује на спољашње догађаје и да се исцртава. Како би се овај проблем разрешио уместо коришћења једини нити за обраду догађаја потребно је креирати већи број нити које се могу блокирати или чекати на дуготрајним операцијама. На овај начин је могуће креирати апликације које могу да се одазову на задату команду без блокирања комплетног интерфејса. Овакве нити би се могле креирати и покретати у методи која обавља обраду догађаја. Пошто се обрада обавља у више нити треба водити рачуна о утркивању нити и конзистентности стања објекта којима се приступа.

```
class OpenListenerError implements ActionListener {
    Editor editor;

    public OpenListenerError(Editor editor) {
        this.editor = editor;
    }

    @Override
    public void actionPerformed(ActionEvent event) {
        JFileChooser jc = new JFileChooser("./");

        int fStatus = jc.showOpenDialog(editor);
        if (fStatus == JFileChooser.APPROVE_OPTION) {
            String path = jc.getSelectedFile().getPath();
            this.editor.area.setCursor(
                new Cursor(Cursor.WAIT_CURSOR));
            Thread t = new Thread() {
                public void run() {
                    try (FileReader reader =
                        new FileReader(path);
                        BufferedReader buffer =
                            new BufferedReader(reader);) {
                        String s;
```

```
        String builder =  
            new StringBuilder();  
        while (  
            (s = buffer.readLine()) != null)  
            builder.append(s).append("\n");  
        }  
  
        editor.area.setText(  
            builder.toString());  
        editor.area.setCursor(new  
            Cursor(Cursor.DEFAULT_CURSOR));  
  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
}  
};  
t.start();  
this.editor.path = path;  
}  
}
```

Проблем са предложените решењем представља потенцијално конкурентан приступ компонентама корисничког интерфејса (`editor.area.setText(builder.toString());`; `editor.area.setCursor( new Cursor(Cursor.DEFAULT_CURSOR));`). Обрада догађаја и исцртавање корисничког интерфејса се извршава у посебној нити (*Event Dispatch Thread*). Програми који приступају *Swing* компонентама сам приступ би требале да обављају унутар ове нити. Ово је неопходно јер већина метода и објекта унутар *Swing* пакета није сигурна за приступ из више нити (*thread safe*). Може доћи до постављања стања објекта због утврђивања нити или због консистенције меморије код меморијског модела који користи Java. Само је известан број метода унутар *Swing* пакета означен као безбедан за конкурентан приступ из више нити. У осталим случајевима може доћи до проблема.

## Прво решење

Како би се решио проблем конкурентног приступа *Swing* корисничком интерфејсу могуће је користити услуге које пружа једна постојећа нит корисничког интерфејса. Ова нит је задужена за креирање послова који обављају иницијализацију објеката корисничког интерфејса као и за њихово прослеђивање нити за обраду догађаја корисничког интерфејса. Задавање посла који је потребно обрадити у нити корисничког интерфејса се постиже позивањем одговарајућих метода класе *javax.swing.SwingUtilities*. Прва метода је:

```
public static void invokeLater(Runnable doRun)
```

Метода омогућава да се у *AWT* нити за обраду догађаја асинхроно изврши метода *run()* објекта *doRun* који имплементира интерфејса *Runnable*. Ова асинхронна обрада ће се извршити после свих догађаја који већ чекају на обраду. Уколико је метода позvana из нити за обраду догађаја обрада ће се опет извршити тек када се сви постојећи догађаји изврше. Ову методу би требало користити уколико је потребно из неке нит ажурирати кориснички интерфејс. Уколико се додги неки изузетак нит за обраду догађаја прихвати овај изузетак.

Друга метода за приступ је:

```
public static void invokeAndWait(Runnable doRun)
    throws InterruptedException, InvocationTargetException
```

Метода обавља исти подао као и метода *invokeLater*, али са том разликом што се нит која је позвала ову методу блокира док се асинхронна обрада не заврши. Уколико се догоди неки изузетак он се прослеђује нити која је позвала дату методу.

```
class OpenListener SwingUtilities implements ActionListener {
    Editor editor;

    public OpenListener SwingUtilities(Editor editor) {
        this.editor = editor;
    }

    @Override
    public void actionPerformed(ActionEvent event) {
        JFileChooser jc = new JFileChooser("./");

        int fStatus = jc.showOpenDialog(editor);
        if (fStatus == JFileChooser.APPROVE_OPTION) {
            String path = jc.getSelectedFile().getPath();
            this.editor.area.setCursor(
                new Cursor(Cursor.WAIT_CURSOR));
            Thread t = new Thread() {
                public void run() {
                    try (FileReader reader =
                        new FileReader(path);
                        BufferedReader buffer =
                            new BufferedReader(reader);) {
                        String s;
                        StringBuilder builder =
                            new StringBuilder();
                        while (
                            (s = buffer.readLine()) != null) {
                            builder.append(s).append("\n");
                        }
                    Runnable r = new Runnable() {

                        @Override
                        public void run() {
                            editor.area.setText(
                                builder.toString());
                            editor.area.setCursor(
                                new Cursor(
                                    Cursor.DEFAULT_CURSOR));
                        }
                    };
                    SwingUtilities.invokeLater(r);

                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

```
        }
    }
};

t.start();
this.editor.path = path;
```

## Друго решење

Други начин за решавање проблема дуготрајне обраде и конкурентног приступа корисничком интерфејсу је коришћење готовог наменског скупа нити за конкурентну обраду коме се прослеђују објекти чије се поједиње метода позивају из нити која обавља обраду догађаја. Ради се о апстрактној класи `SwingWorker<T,V>` пакета `Swing`. Ова класа је намењена за опис дуготрајних или блокирајућих послова који приступају корисничком интерфејсу било током извршавања било по завршетку обраде. Најчешћи сценарио коришћења ових објеката укључује интеракцију више нити. На почетку нека нит (најчешће нит за обраду догађаја) позива методу `execute()` која распоређује дати објекат како би се извршио у некој од доступних нити из расположивог скупа нити. Нека од доступних нити за обраду прихвата овај објекат и позива његову методу `dolnBackground()` коју је потребно имплементирати. Унутар ове методе се обавља комплетна обрада, али ова метода не обавља приступ корисничком интерфејсу. Како би се омогућила промена корисничког интерфејса током обављања ових дуготрајних послова из ове методе је могуће позвати методу са променљивим бројем аргумента `publish(V... chunks)` која креира листу међурезултата (догађаја) који ће се обрадити у нити за обраду догађаја. Како би се омогућила промена корисничког интерфејса након завршетка дуготрајних послова користи се генеричка повратна вредност `T` методе `dolnBackground`. Нит за обраду догађаја обрађује све догађаје које је креирала ова метода. Уколико постоје међурезултати (догађаји) који креирани са `publish` обрађују се у методи `process(List<V> chunks)` коју је потребно имплементирати. Овде треба приметити да број колико је пута позвана метода `process` не мора да се поклопи са бројем позивања методе `publish`, овај број може бити мањи. Може се десити да прикупи већи број делова (`chunks`) насталих из више позива методе `publish` и да се са свима њима позове метода `process`. Резултати који су добијени кроз повратну вредност се могу приказати на корисничком интерфејсу и за то је потребно имплементирати методу `done()`. Како ова метода нема аргумента повратној вредности се приступа користећи постојећу методу `get()`.

Треба приметити да апстрактна класа `SwingWorker<T, V>` имплементира интерфејсе `Runnable`, `Future<T>`, `RunnableFuture<T>` кроз које је омогућена комуникација са нитима у позадини, али и могућност за отказивање посла, дохватање прогреса и резултата посла, као и провера статуса посла.

```
class OpenListenerSwingWorker implements ActionListener {
    Editor editor;

    public OpenListenerSwingWorker(Editor editor) {
        this.editor = editor;
    }

    @Override
    public void actionPerformed(ActionEvent event) {
```

```
JFileChooser jc = new JFileChooser("./");

int fStatus = jc.showOpenDialog(editor);
if (fStatus == JFileChooser.APPROVE_OPTION) {
    String path = jc.getSelectedFile().getPath();
    this.editor.area.setCursor(
        new Cursor(Cursor.WAIT_CURSOR));
    SwingWorker<String, Void> sw =
        new SwingWorker<String, Void>() {
            @Override
            public String doInBackground() {
                String result = null;
                try (FileReader reader =
                    new FileReader(path);
                     BufferedReader buffer =
                         new BufferedReader(reader);) {
                    String s;
                    StringBuilder builder =
                        new StringBuilder();
                    while (
                        (s = buffer.readLine()) != null) {
                        builder.append(s).append("\n");
                    }
                    result = builder.toString();
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
            return result;
        }

        @Override
        protected void done() {
            try {
                editor.area.setText(get());
                editor.area.setCursor(new Cursor(
                    Cursor.DEFAULT_CURSOR));
            } catch (Exception e) {
            }
        }
    };
    sw.execute();
    this.editor.path = path;
}
}
```

На почетку се креира бафер у који се убацују приспели послови које је потребно обрадит користећи неку од расположивих нити, али се не креира се цео скуп нити већ се нити креирају једна по једна како пристижу нови послови на обраду. Приликом коришћења скупа нити треба водити рачуна да је овај скуп фиксне величине. Када се досегне максималан број нити (**MAX\_WORKER\_THREADS = 10**) нове нити се не креирају, већ се само послови смештају у бафер. Уколико су све нити блокиране може доћи до проблема јер тренутно није могуће на директан начин повећати расположиви број нити.

## Мрежно програмирање

Мрежно програмирање има за циљ да представи технике писања дистрибуираних апликација које се ослањају на коришћење интернета и одговарајућих мрежних протокола. Мрежно програмирање се заснива на размени порука између крајњих тачака у комуникацији. Крајња тачка комуникације се састоји из паре *IP* адреса рачунара коме се приступа и порт преко кога се приступа том рачунару. Порт представља целобројну вредност у интервалу од 0 до 65535 која служи да би се унутар једног рачунара могло разликовати коме је програму, или процесу упућена нека порука, преко којих се може комуницирати. Да би могло да се комуницира преко одређеног порта програм мора код оперативног система да се пријави и да резервише коришћење одређеног порта. Комуникација се код мрежног програмирања заснива на постојању две стране које треба да комуницирају. Једне која очекује захтеве за комуникацијом, серверске стране, и оне која иницира комуникацију, клијентске стране.

Серверска страна може да представља било који програм којем се може приступити преко мреже, то јест представља једну фиксну крајњу тачку комуникације. Да би програм био серверски поред доступности потребно је да по добијању одређеног захтева преко мреже обради тај захтев и врати одговор страни која је тражила обраду захтева. Клијентска страна представља програм који може да креира и пошаље захтев који треба упутити серверу и чека на одговор од сервера.

Кораци рада серверске стране:

- Креирање серверског порта. Сервер мора да креира порт на који му клијенти могу слати захтеве.
- Чекање на клијентски захтев ослушкивањем серверског порта.
- Успостављање комуникације користећи две крајње тачке комуникације. Серверска адреса и порт на серверској страни чине прву крајњу тачку комуникације, а клијентска адреса и порт на клијентској страни чине другу крајњу тачку комуникације. Треба приметити да се након успостављања везе са клијентом преко серверског порта комуникације може пребацити на неки други порт на серверској страни.
- Покретање нити која ће наставити даљу комуникацију са клијентом.
- Повратак на ослушкивање серверског порта.
- Ослобађање серверског порта. Порт на неком рачунару представља ресурс који је на крају рада потребно ослободити како би касније могао поново да се користи.

На овај начин је омогућено да већи број клијената може да комуницира са једним сервером преко једног серверског порта.

## Java - Net

Мрежно програмирање у програмском језику Java се у највећој мери заснива на коришћењу развијених библиотека, скупу класа и интерфејса кроз које се приступа мрежним протоколима и интернету. Ове библиотеке су већим својим делом унутар пакета `java.net`. У овом пакету се налазе основне класе кроз које је могуће успоставити комуникацију и комуницирати са другом страном. Овде треба нагласити да се ради о дистрибуираном програмирању користећи мрежне протоколе тако да се са друге стране не мора налази програм који је писан користећи програмски језик Java све док се поштује успостављени протокол комуникације.

Класе и интерфејси у овом пакету се грубо могу поделити на оне који омогућавају мрежно програмирање на ниском и на високом нивоу. Класе и интерфејси који омогућавају рад на ниском нивоу би укључивали адресе и приклjučнице:

- **InetAddress** – Представља интернет IP адресу. Пружа рад са различитим типовима адреса, као и методе за добијање имена рачунара, ... Из ове класе су изведене класе `Inet4Address`, `Inet6Address` за различите верзије протокола `IPv4` и `IPv6`.
- **Socket** – Приклjučница обезбеђује обављање комуникације на мрежи.
- **ServerSocket** – Серверска приклjučница преко које се на одређеном порту послушују клијентски захтеви.
- **DatagramSocket**, **DatagramPacket**, **MulticastSocket**, **DatagramPacket**, ...

Класе и интерфејси који омогућавају рад на високом нивоу би укључивали идентификаторе ресурса и протоколе за приступ:

- **URI (Universal Resource Identifier)** – Представља идентификатор ресурса у складу са RFC 2396 спецификацијом.
- **URL (Uniform Resource Locator)** – Представља референцу на неки ресурс на интернету, али и обезбешује средства за приступ ресурсу.
- **URLConnection**, **HttpURLConnection**, ...

### Креирање серверске приклjučнице

Да би се подигао серверски програм на неком рачунару потребно му је доделити слободан порт преко кога се може остварити комуникација са осталим рачунарима. Ово мора да буде серверски порт, тј. Java треба да буде способна да прима поруке упућене рачунару на коме је покренута и то оне поруке које су адресиране на дати порт.

```
import java.net.*;  
...  
ServerSocket serverSocket = null;
```

```

try {
    serverSocket = new ServerSocket(port);
} catch (IOException e) {
    ...
}

```

Приликом подизања, отварања, серверског порта може доћи до појаве изузетака. Ови изузетци су изведене из основне класе улазно/излазних изузетака *IOException*. Пример би био да тражени порт није доступан. Најчешћа порука која прати овај изузетак би била:

```
java.net.BindException: Address already in use ...
```

Други тип изузетка који се може дододити приликом позива конструктора се односи на права приступа (*SecurityException*), уколико су успостављене безбедносне провере (*SecurityManager*) које забрањују коришћење датог порта (*checkListen*). Најчешћа порука која прати овај изузетак би била:

```
java.security.AccessControlException: access denied ...
```

Уколико је аргумент изван интевала од 0 до 65535, што представљају могуће вредности портова, долази до изузетка. Најчешћа порука која прати овај изузетак би била:

```
java.lang.IllegalArgumentException: Port value out of range: ...
```

Поред конструктора који као једини аргумент има број серверског порта, постоје и конструктори код којих је могуће специфицирати и максималну величину бафера за пријем долазећих захтева, као и локалну адресу, ако их има више, са којом ће сервер бити повезан.

На крају рада сервера је потребно обавити ослобађање серверског порта затварањем серверске приклучнице.

### **Успостављање везе сервера са клијентом**

Како би се успоставила веза са клијентом када клијент упути захтев потребно је да сервер формира приклучницу, *Socket*, кроз коју ће ићи даља комуникација. Крајња тачка комуникације специфицирана овом приклучници, на серверској страни, садржаће локалну адресу на којој је сервер подигнут као и порт кроз који иде комуникација. У верзијама Јаве почев од 1.5 комуникација иде користећи серверски порт, док се у претходним верзијама комуникација није обављала кроз серверски порт, већ преко новог порта који се лоцира код оперативног система.

```

Socket clientSocket = null;
try {
    clientSocket = serverSocket.accept();
} catch (IOException e) {
    ...
}

```

```
}
```

Оно што треба приметити је да серверски порт након операције `accept()` и даље слободан за успостављање нових веза. Пошто је комуникација одређена са две крајње тачке, то крајња тачка на серверској страни може бити иста за све захтеве. Потребно је само на основу друге крајње тачке приспеле поруке прослеђивати у одговарајућу прикључницу.

Приликом успостављања везе може доћи до већег броја изузетака. Пример за ово би био да је серверска прикључница затворена или да није ни повезана. Порука у случају да је затворена би била:

```
java.net.SocketException: Socket is closed ...
```

Уколико је истекао постављени интервала чекања, постављен са `setSoTimeout`, такође долази до изузетка. Порука у случају би била:

```
java.net.SocketTimeoutException: Accept timed out
```

Ова два типа изузетака се могу користити у случају да је потребно пробудити серверску нит, која чека на улазно/излазном догађају, како би се регуларно завршило извршавање програма. Нити које чекају на улазно/излазним уређајима није могуће пробудити коришћењем `interrupt` методе.

Следећи тип изузетка који се може догодити приликом чекања се односи на права приступа (`SecurityException`), уколико су успостављене безбедносне провере (`SecurityManager`) и ако није дозвољено чекање захтева (`checkAccept`). Порука би била:

```
java.security.AccessControlException: access denied ...
```

Такође се могу десити изузетци приликом рада са каналима комуникације уколико је канал у неблокирајућем моду, а нема конекције која се може прихватити:

```
java.nio.channels.IllegalBlockingModeException ...
```

### **Успостављање везе клијента са сервером**

Да би дошло до комуникације потребно је на клијентској страни формирати прикључницу, `Socket`, кроз коју ће ићи даља комуникација. Комуникација започиње тако што клијент формира своју крајњу тачку комуникације и упућује захтев серверу на серверски порт. Када сервер прихвати захтев обавештава клијента о другој крајњој тачки комуникације.

```
Socket server = null;
try {
    Socket server = new Socket(host, port);
} catch (IOException e) {
    ...
}
```

Аргумент `host` представља име рачунара коме се приступа, док поље `port` представља серверски порт на серверском рачунару преко кога започиње комуникација.

Приликом креирања прикључнице може доћи до појаве изузетака зато што неки параметар није исправан.

Уколико се не може утврдити *IP* адреса сервера на основу поља *host* долази до изузетка. Порука би била:

```
java.net.UnknownHostException: ...
```

Уколико је аргумент *port* изван интевала од 0 до 65535 такође долази до изузетка, чија би порука била:

```
java.lang.IllegalArgumentException: Port value out of range: ...
```

Други тип изузетка који се може дододити се односи на права приступа (*SecurityException*) када су успостављене безбедносне провере (*SecurityManager*) која забрањују повезивање (*checkConnect*). Клијент се повеже са сервером, али даља комуникација није могућа. Порука би била:

```
java.security.AccessControlException: access denied ...
```

Поред ових изузетака може се појавити више изузетака који се односе на рад улазно/излазних уређаја. Ови изузетци су изведени из основне класе улазно/излазних изузетака *IOException*. Пример за овај тип изузетака био би случај да серверска страна постоји, али не ослушкује серверски порт, када би порука била:

```
java.net.ConnectException: Connection refused: connect
```

Други пример би се односио на ситуацију када се специфицирани сервер налази унутар мреже која није доступна, па би порука била:

```
java.net.SocketException: Network is unreachable: connect
```

Поред конструктора код кога се на неки начин специфицира серверски рачунар и серверски порт, могуће је специфицирати и локалну адресу, ако их има више, која ће се користити у комуникацији као и жељени локални порт. Такође је могуће специфицирати и посредника, *Proxy*, који ће обављати комуникацију.

На крају комуникације клијента и сервера је и на клијентској и на серверској страни потребно затворити прикључнице.

### Добијање блокирајућих токова података

Комуникација која се обавља између клијента и сервера иде разменом порука које се транспортују кроз улазни и излазни ток података. Прикључница омогућава двосмерну комуникацију, тј. она даје и улазни и излазни ток података.

Добијање улазног тока података се остварује на следећи начин:

```
InputStream in = socket.getInputStream();
```

Овде се ради о току података који омогућује читање из тока података на нивоу бајта.

Приликом дохватања улазног тока података могу се појавити изузетци који се односе на рад улазно/излазних уређаја. Ови изузетци су изведени из основне класе улазно/излазних изузетака *IOException*. Примери би укључивали то да је прикључница затворена, или није повезана или улаз затворен, када би поруке биле:

```
java.net.SocketException: Socket is closed  
java.net.SocketException: Socket is not connected  
java.net.SocketException: Socket input is shutdown
```

Полазећи од тока који ради на нивоу бајтова могу се добити токови података који раде са карактерима и линијама текста на следећи начин:

```
InputStreamReader inr = new InputStreamReader(in);  
BufferedReader br = new BufferedReader(inr);
```

Добијање излазног тока података се остварује на следећи начин:

```
OutputStream out = socket.getOutputStream();
```

Овде се ради о току података који омогућује упис у ток података на нивоу бајта.

На сличан начин као приликом дохватања улазних токова података и приликом дохватања излазног тока података могу се појавити изузети који се односе на рад улазно/излазних уређаја. Ови изузети би укључивали то да је приклучница затворена, или није повезана или излаз затворен, када би поруке биле:

```
java.net.SocketException: Socket is closed  
java.net.SocketException: Socket is not connected  
java.net.SocketException: Socket output is shutdown
```

Сложенији токови података који раде са карактерима и линијама текста, се добијају на следећи начин:

```
OutputStreamWriter outw = new OutputStreamWriter(out);  
PrintWriter pw = new PrintWriter(outw, true);
```

Други аргумент код конструктора класе *PrintWriter* означава да ће се приликом комуникације бафер за смештање података приликом спања аутоматски празнити, тј. неће се чекати да се напуни, па тек онда да се пређе на спање комплетног садржаја.

Рад са блокирајућим токовима података је омогућен користећи пакет *java.io.\**.

### Рад са каналима комуникације

Поред рада са блокирајућим токовима података, приклучнице омогућавају и рад са комуникационим каналима и асинхрону и неблокирајућу комуникацију. Интерфејси и класе за рад са каналима комуникацију су дати у пакету *java.nio.\**.

Добијање канала комуникације се остварује на следећи начин:

```
SocketChannel channel = socket.getChannel();
```

### Серијализација објекта

Токови података који раде на нивоу бајта, карактера или низа простих типова су погодни за транспортуовање једноставних података. Постоји проблем када је потребно проследити сложен тип података, објекат, користећи токове података. У програмском језику Java постоји начин да се комплетан објекат, са свим својим локалним пољима, транспортује користећи токове података. Пошто објекти могу имати сложену структуру,

која може да укључује повратне везе између објеката приликом смештања објекта у ток мора се водити рачуна о редоследу по коме ће бити смештен сваки од аргумента, тј. начин да се изврши његова серијализација.

Да би се објекат сместио у ток података потребно је да он, и сва његова поља рекурзивно, који нису простог типа имплементирају интерфејс *Serializable*. Овај интерфејс нема метода које је потребно имплементирати, већ даје индикацију виртуелној машини о начину на који треба сместити дати објекат.

```
import java.util.*;

class SomeClass implements Serializable {
    ...
}
```

Уколико класа или неко њено поље не имплементирају интерфејс *Serializable* приликом покушаја уписа у објектни ток података долази до појаве изузетка. Такође и приликом покушаја читања таквог објекта долази до појаве изузетка.

```
java.io.WriteAbortedException: writing aborted;
```

```
java.io.NotSerializableException:
```

Уколико се жели избегти серијализација неког поља потребно је том пољу доделити квалификатор *transient*. На овај начин се неће у ток уписати дато поље.

```
import java.util.*;

class SomeClass implements Serializable {
    private transient int attribute;
    ...
}
```

Треба напоменути да се на овај начин чувају само локална поља док заједничка поља на нивоу класе, која имају квалификатор *static*, неће бити сачувана.

Добијање улазног, односно излазног тока података који ради са објектима се остварује на следећи начин:

```
ObjectOutputStream oos = new ObjectOutputStream(out);
ObjectInputStream ois = new ObjectInputStream(in);
```

Приликом отварања објектних токова података треба водити рачуна о редоследу по коме се ови токови података отварају. У конструктору улазног тока података чита се заглавље тока како би се верификовало да је ток одговарајући. Конструктор ће се блокирати све док одговарајући излазни ток, *ObjectOutputStream*, са друге стране не упише заглавље у ток и док га не проследи. Уколико би обе стране прво покушале да отворе улазни објектни ток података дошло би до мртвог блокирања, *deadlock*, јер би се ове две стране међусобно чекале. Како би се ово отклонило потребно је да барем једна страна прво отвори излазни објектни ток, па тек онда улазни. Уколико је потребно остварити симетричну комуникацију могу обе стране истовремено да отворе прво излазне, па тек онда улазне објектне токове података.

Упис, односно читање објекта из тока података се остварује на следећи начин:

```
oos.writeObject(obj);
SomeClass newObj = (SomeClass) ois.readObject();
```

Приликом слања објекта користећи објектне токове података ради се њихово кеширање. Када се објекат први пут појави за упис у ток, он се уписује у ток и чува се у локалном кешу. Када се следећи пут појави исти објекат прво се провери да ли је већ у кешу и када се утврди да јесте у кешу, у ток се не уписује поново тај објекат, већ се уписује само његов идентификатор. На овај начин се објекат уколико је већ једном био смештен у ток података не смешта по други пут. Поступак кеширања убрзава поновно слање објекта, али и пружа начин за слање веома сложених објекта који имају кружну зависност између поља. Мана овог приступа може се појавити у случају да су токови служили за транспорт велике количине података. Сви ти подаци ће остати у локалним кешевима што може да доведе до тога да виртуелна машина остане без расположиве меморије.

```
java.lang.OutOfMemoryError: Java heap space
```

117

Таскање

У програмском језику Java је потребно написати скуп класа за интерактивну комуникацију између корисника (*Chat*). Потребно је реализовати решење код кога корисник може да пошаље већи број порука без чекања одговора на претходне.

## Решење

```

import java.net.*;

public class Chat {
    Socket client;

    public Chat(Socket client) {
        this.client = client;
    }

    public void communicate() {

        try (Socket client = this.client;) {
            ReadThread read = new ReadThread(client);
            WriteThread write = new WriteThread(client);
            read.start();
            write.start();
            read.join();
            write.join();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

import java.io.*;
import java.net.*;

public class ReadThread extends Thread {
    Socket client;

    public ReadThread(Socket client) {
        this.client = client;
    }

    @Override
    public void run() {
        try (Socket client = this.client;
             InputStream in = client.getInputStream();
             BufferedReader pin = new BufferedReader(
                     new InputStreamReader(in));) {

            String s;
            while ((s = pin.readLine()) != null) {
                System.out.println("> " + s);
            }
        }
    }
}

```

```
        }
    } catch (Exception ex) {
    }
}

import java.io.*;
import java.net.*;

public class WriteThread extends Thread {
    Socket client;

    public WriteThread(Socket client) {
        this.client = client;
    }

    @Override
    public void run() {
        try (Socket client = this.client;
             OutputStream out = client.getOutputStream();
             PrintWriter pout = new PrintWriter(out, true);

             InputStream is = System.in;
             InputStreamReader isr =
                 new InputStreamReader(is);
             BufferedReader br = new BufferedReader(isr)) {

            String s;
            while (!pout.checkError() &&
                   ((s = br.readLine()) != null)) {
                pout.println(s);
            }
        } catch (Exception ex) {
        }
    }
}

import java.net.*;

public class Client {

    public static void main(String[] args) {
        String host = args[0];
        int port = Integer.parseInt(args[1]);

        try (Socket server = new Socket(host, port);) {
            Chat chat = new Chat(server);
            chat.communicate();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

```

import java.net.*;

public class Server {

    public static void main(String[] args) {
        int port = Integer.parseInt(args[0]);

        try (ServerSocket listener = new ServerSocket(port)) {
            Socket client = listener.accept();
            Chat chat = new Chat(client);
            chat.communicate();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}

```

Друго решење

```

import java.io.*;
import java.net.*;
import java.util.*;

public class WriteThread2 extends Thread {
    Socket client;

    public WriteThread2(Socket client) {
        this.client = client;
    }

    @Override
    public void run() {
        try (Socket client = this.client;
             OutputStream out = client.getOutputStream();
             PrintWriter pout = new PrintWriter(out, true);
             Scanner scanner = new Scanner(System.in)) {

            while (!pout.checkError() && scanner.hasNextLine()) {
                pout.println(scanner.nextLine());
            }
        } catch (Exception ex) {
        }
    }
}

```

118

## Клијент-сервер архитектура - сервери

Написати систем класа које имплементирају серверску страну клијент-сервер архитектуре.

## Решење

Серверске апликације су задужене за интеракцију са клијентским апликацијама. Ова интеракција најчешће започиње тако што клијент упути серверу одговарајући захтев, након чега се обавља посао у складу са примљеним захтевом. На овај начин се може описати већина серверских апликација: имеил сервер, веб сервер, сервер базе података, *FTP* сервер, *REST* сервиси, *SOAP* сервиси, комуникација са *EJB*, као и многи други.

Приликом развоја серверских апликација потребно је водити рачуна о пословима које је потребно обавити као одговор на захтеве које је клијент упутио. Основна структура серверских апликација је таква да од клијента приме захтев, обраде тај захтев и клијенту врате одговор који се односи на тај захтев. Интеракција може да буде и сложенија тако да посао који је потребно да се одради на серверској страни захтева више корака комуникације са клијентом. Већина серверских апликација је таква да се интеракција са сваким клијентом и посао које се тада обавља може ставити у засебан део, најчешће нит, и тамо обрађивати.

Основни костур серверске апликације се састоји из дела подизања серверске приклучнице (`listener = new ServerSocket(port)`), петље у којој се примају и обрађују клијентски захтеви (`while (running)`), која се састоји из дела за пријем клијентских захтева (`Socket client = listener.accept()`) и методе за обраду клијентских захтева (`processRequest(client)`). На крају рада серверске апликације (`finally`) је потребно ослободити све заузете ресурсе током рада серверске апликације (`close()`).

```
public void run() {
    try {
        listener = new ServerSocket(port);
        while (running) {
            Socket client = listener.accept();
            processRequest(client);
        }
    } catch (Exception ex) {
        ex.printStackTrace();
    } finally {
        close();
    }
}
```

Како би се обезбедило више различитих начина за обраду клијентских захтева, метода за обраду захтева (`processRequest`) је дата као апстрактна, те свака класа која жели да реализације свој начин формирања ове обраде то може да учини на другачији начин.

Овде треба приметити да иако класа `ServerSocket` имплементира интерфејс `AutoCloseable` овде није искоришћено да се њена метода `close` аутоматски позове у финалној (`finally`) грани `try-with-resources` блока. Разлог зашто је ово учињено је заустављање серверске нити. Уколико би неко покушао да заустави серверску нит само постављањем променљиве `running` (која је означена са `volatile`) на вредност `false`,

и позивањем прекида за дату нит *interrupt* ова нит се не би зауставила уколико је чекала на пријему новог захтева *listener.accept()*, све док тај захтев не пристигне. Како би се нит сигурно завршила потребно ју је пробудити и из методе *listener.accept()* која чека на улазно-излазном систему. Ово се може постићи затварањем серверске приклучнице у истој методи у којој се поставља и *running* и позива *interrupt*.

```

import java.io.*;
import java.net.*;

public abstract class Server implements Runnable {
    private static int id = 0;

    public static final int DEFAULTPORT = -1;
    public static final String DEFAULTPROTOCOL = "ServerProtocol1";

    protected String protocol;
    protected String host;
    protected int port;

    public Server() {
        this(DEFAULTPROTOCOL, DEFAULTPORT);
    }

    public Server(int port) {
        this(DEFAULTPROTOCOL, port);
    }

    public Server(String protocol, int port) {
        this.port = port;
        this.protocol = protocol;
        thread = null;
        running = false;
    }

    ServerSocket listener = null;

    public void run() {
        try {
            listener = new ServerSocket(port);
            while (running) {
                Socket client = listener.accept();
                processRequest(client);
            }
        } catch (Exception ex) {
            ex.printStackTrace();
        } finally {
            close();
        }
    }

    public abstract void processRequest(Socket client);

    protected volatile Thread thread;
    protected volatile boolean running;
}

```

```

public void start() {
    if (thread == null) {
        thread = new Thread(this, "Server");
        running = true;
        thread.start();
    }
}

public void stop() {
    running = false;
    thread.interrupt();
    close();
}

public void close() {
    try {
        listener.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

public boolean isRunning() {
    return running;
}

public static synchronized int nextId() {
    return id++;
}
}
}

```

**Прво решење**

Ово решење представља имплементацију сервера који комплетну обраду обавља у једној јединој серверској нити. Унутар методе `processRequest` креира се објекат `WorkingThread` који проширује основну класу нити `Thread`, а који уме да комуницира са клијентом. Како би се комплетна комуникација обавила унутар серверске нити уместо позивања методе `start` која би комуникацију обавила у засебном току контроле позива се метода `run` тако да се комуникација обавља у текућем току контроле.

```

import java.net.*;

public class OneThreadServer extends Server {

    public OneThreadServer() {
        super();
    }

    public OneThreadServer(int port) {
        super(port);
    }

    public OneThreadServer(String protocol, int port) {
        super(protocol, port);
    }
}

```

```
    @Override
    public void processRequest(Socket client) {
        try {
            WorkingThread wt =
                new WorkingThread(nextId(), client, protocol);
            wt.run();
        } catch (Exception ex) {
        }
    }
}
```

## Друго решење

Ово решење представља имплементацију сервера који формира засебну нит за сваки приспели захтев. Унутар методе *processRequest* креира се објекат *WorkingThread* који проширује основну класу нити *Thread* и позива се његова метода *start*. На овај начин се покреће засебан ток контроле у коме се обавља комуникација са клијентом. Креирањем засебног тока контроле је омогућено да се главна серверска нит врати на пријем нових захтева. Ово има за последицу да сервер може прихватити наредни захтев пре него што је претходни захтев обрађен, чиме се постиже бољи одзив система. Обрадом која се обавља у засебним токовима контроле може се постићи да сервер може обрадити већи број захтева у јединици времена, јер због блокирања и чекања на улазноизлазне операције део нити може бити блокиран. За разлику од сервера који има само једну нит, код кога није било потребе за синхронизацијом и закључавањем објекта, код сервера који има већи број нити треба водити рачуна о синхронизацији између нити.

```
import java.net.*;

public class MultiThreadsServer extends Server {

    public MultiThreadsServer() {
        super();
    }

    public MultiThreadsServer(int port) {
        super(port);
    }

    public MultiThreadsServer(String protocol, int port) {
        super(protocol, port);
    }

    @Override
    public void processRequest(Socket client) {
        new WorkingThread(nextId(), client, protocol).start();
    }
}
```

Креирање већег броја нити где се једна нит креира по сваком примљеном захтеву обично има боље перформансе него решење које има само једну нит. Проблеми могу настати уколико постоји потреба да се у неком кратком временском периоду креира велики број нити који превазилази карактеристике сервера. За креирања нити потребно

је потрошити не тако кратко време на интеракцију са виртуелном машином и оперативним системом. Уколико је ово време дуже од времена обраде код система са једном нити онда се не исплати креирати нову нит за сваки приспели захтев. Такође треба водити рачуна о томе да виртуелна машина може у неком тренутку успешно водити рачуна о коначном броју нити. Овај број може бити последица или ограниченог простора за креирање нових нити или ограниченог времена који нит може да проведе у извршавању. У случају напада на сервер злонамерни корисници креирају велики број захтева што у случају оваквог сервера може да изазове да систем остане без меморије и да већ приспели захтеви не могу да се извршавају.

Додавање нових нити које обрађују приспеле захтеве обично доводи до побољшања перформанси све док се не пређе одређена граница. Након те границе долази до деградације перформанси. Како би се деградација перформанси избегла користе се решења која не креирају нову нит са сваким приспелим захтевом већ користе унапред креиран скуп нити коме се прослеђују приспели захтеви на обраду. На овај начин се постиже да се ресурси система боље користе, јер се не чека на креирање нове нити при сваком захтеву, већ се уколико има слободних нити некој од њих прослеђује захтев за обрадом.

### Треће решење

Код овог решења се креира коначан број нити који могу да обављају тражени посао. Када приспе захтев креира се објекат типа *Runnable* који се убацује у бафер, а нити у свом току контроле преузимају те објекте и извршавају њихову *run* методу. Ове нити, *BufferWorker*, креирају се унутар конструктора серверске класе, *BufferServer*. Свакој од ових нити се прослеђује референца на дељени бафер у који се умећу послови које је потребно обрадити и из кога ове нити узимају послове. Ови послови су класе *WorkingThread* која проширује класу *Thread*, а који имплементира интерфејс *Runnable*. Како би се обезбедило да се серверске нити угасе када у систему престану да се извршавају преостале нити, постављено је да дате нити буду демонске, *setDaemon(true)*;

```
import java.net.*;
import java.util.*;
import java.util.concurrent.*;
public class BufferServer extends Server {
    public static final int MAXTHREADS = 10;

    protected BlockingQueue<Runnable> buffer;
    protected List<BufferWorker> threads;

    public BufferServer() {
        this(MAXTHREADS, "", DEFAULTPORT);
    }

    public BufferServer(String protocol, int port) {
        this(MAXTHREADS, protocol, port);
    }

    public BufferServer(int num, String protocol, int port) {
        super(port);
        // 2 * num empirical factor
        buffer = new ArrayBlockingQueue<Runnable>(2 * num);

        threads = new LinkedList<BufferWorker>();
    }
}
```

```

    for (int i = 0; i < num; i++) {
        BufferWorker bw = new BufferWorker(i, buffer);
        bw.setDaemon(true);
        bw.start();
        threads.add(bw);
    }
}

@Override
public void processRequest(Socket client) {
    try {
        buffer.put(
            new WorkingThread(nextId(), client, protocol));
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

@Override
public void stop() {
    super.stop();
    buffer.clear();
    for (BufferWorker bw : threads) {
        bw.shutdown();
    }
}
}

public class BufferWorker extends Thread {
    BlockingQueue<Runnable> buffer;
    private volatile boolean running;

    public BufferWorker(int id, BlockingQueue<Runnable> buffer) {
        super("BufferWorker" + id);
        this.buffer = buffer;
        this.running = true;
    }

    public void run() {
        try {
            while (running) {
                Runnable r = buffer.take();
                r.run();
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public void shutdown() {
        this.running = false;
        this.interrupt();
    }
}

```

## Четврто решење

Код овог решења се користи постојећа инфраструктура која се налази унутар пакета `java.util.concurrent`, а које могу обављају тражени посао. Основни интерфејс је `Executor`, прихватавају објекат типа `Runnable` и извршавај `run` методу. Овај интерфејс пружа могућност за раздвајање креирања неког захтева од његовог извршавања. Уместо експлицитног креирања и покретања нове нити која онда обавља дати задатак са `new Thread(new(RunnableTask())).start()` Java препоручује коришћење класа које имплементирају `Executor` интерфејс и код којих се креира нови задатак и који се онда прослеђује на извршавање, `executor.execute(new RunnableTask())`. Треба приметити да у зависности од имплементације овог интерфејса обрада задатка може бити у засебној новој нити, у некој нити уз скупа нити, али и у истој нити.

Интерфејс `ExecutorService` проширује интерфејс `Executor` и пружа могућност за заустављање обраде и престанак пријема нових захтева. За ово се користе две методе. Прва је метода `shutdown()` која дозвољава претходно примљеним задацима да се обаве, али не дозвољава пријем нових задатака. Друга је метода `shutdownNow()` која забрањује пријем нових задатака, спречава извршавање већ приспелих задатака, и покушава да заустави постојеће задатке. Поред ових метода пружа и друге методе међу којима су и методе за чекање на крај заустављања.

Креирање скупа нити је омогућено користећи класу `Executors` и скуп њених статичких метода. У зависности од жељеног типа скупа нити могу се користити методе:

Метода `newFixedThreadPool` се користи за креирање скупа нити фиксне величине. На почетку се креира бафер у који се убацују приспели задаци, не креира се цео скуп нити, већ се нити креирају једна по једна како пристижу нови задаци на обраду. У било ком тренутку максимално онолико нити колико је задато ће се извршавати. Нове нити се додају само уколико се унутар нити додати изузетак због кога се прекине извршавање нити.

Метода `newSingleThreadExecutor` креира једну нит која обавља задатке секвенцијално, а на почетку се креира бафер у који се убацују нови задаци које ова нит извршава. Уколико се унутар ове нити додати изузетак због кога се прекине њено извршавање креира се нова нит. Ово доста личи на креирање скупа који се састоји од само једне нити, `newFixedThreadPool(1)`, али се од њега разликује по томе што се гарантује да се неће поново конфигурирати да користи више нити.

Метода `newCachedThreadPool` креира скуп нити код кога се нове нити убацују у кеш када је то потребно, али и користи постојеће нити када су доступне. Намењена је за обраду великог броја асинхроно направљених кратких послова. Када пристигне нови задатак проверава се кеш да ли постоји нека слободна нит, ако постоји, тој нити се даје да обави задатак, а ако не постоји креира се нова нит. Нити које нису биле коришћене дуже од 60 секунди се заустављају и избацују из кеша. Постоји и могућност за постављање фабрике која креира нити.

Метода `newScheduledThreadPool` креира скуп нити задате величине који подржавају одлагање извршавања неког задатка и периодично извршавање задатака, `ScheduledExecutorService`. Слична овој методи, само са скупом непроменљиве величине 1, је метода `newSingleThreadScheduledExecutor`.

Метода `newWorkStealingPool` креира скуп нити који могу да подрже задати ниво паралелизма, и користи више редова за задатке како би се смањило чекање. Ниво паралелизације одговара максималном броју нити које активно могу да учествују у

обради задатака или који је доступан. Стваран број нити може да варира током времена динамички. Код овог скупа се не гарантује редослед по коме ће предати задачи бити одрађени.

```

import java.net.*;
import java.util.concurrent.*;

public class ExecutorServer extends Server {
    public static final int MAXTHREADS = 10;

    protected ExecutorService pool;

    public ExecutorServer() {
        this(MAXTHREADS, "", DEFAULTPORT);
    }

    public ExecutorServer(int numOfThreads,
                         String protocol, int port) {
        super(protocol, port);
        pool = Executors.newFixedThreadPool(numOfThreads);
    }

    @Override
    public void processRequest(Socket client) {
        pool.execute(new WorkingThread(0, client, protocol));
    }

    @Override
    public void stop() {
        super.close();
        pool.shutdown();
        try {
            if (!pool.awaitTermination(60, TimeUnit.SECONDS)) {
                pool.shutdownNow();
                if (
                    !pool.awaitTermination(60, TimeUnit.SECONDS))
                    System.err.println("Pool did not terminate");
            }
        } catch (InterruptedException ie) {
            pool.shutdownNow();
            Thread.currentThread().interrupt();
        }
    }
}

```

#### Пето решење

Код овог решења се полази од претходног решења и демонстрира се како се од нити која је обрађивала неки посао може добити повратна вредност, информације о томе да ли је посао завршен, као и да се посао уколико још увек није започео откаже. Интерфејс *ExecutorService*, поред могућности наведених у претходном решењу, пружа и могућност за задавање задатака у два основна облика и дохваташње резултата које оне враћају. Први начин је у облику класе која имплементира интерфејс *Runnable* када

се у зависности од позване методе добија или само информација да је обрада завршена када се враћа *null* вредност или и информација да је обрада завршена као и унапред задата вредност. Други начин је у облику класе која имплементира интерфејс *Callable* који имају методу *call* чији се резултат враћа као повратна вредност. На овај начин се поред информације да је обрада завршена добија и резултат обраде. Код оба начина задавања задатака резултат који се враћа је дат интерфејсом *Future* у који ће бити упакована повратна вредност. Кроз објекат типа *Future* могуће је проверити да ли је обрада завршена, сачекати резултат обраде, отказати обраду, или проверити да ли је обрада већ отказана.

```

import java.net.*;
import java.util.concurrent.*;

public class ExecutorFutureServer extends ExecutorServer {

    public ExecutorFutureServer(int num, String protocol, int port) {
        super(num, protocol, port);
    }

    @Override
    public void processRequest(final Socket client) {
        Future<Void> future = pool.submit(new Callable<Void>() {
            public Void call() {
                WorkingThread wt =
                    new WorkingThread(0, client, protocol);
                wt.run();
                return null;
            }
        });
        try {
            future.get();
        } catch (Exception ex) {
        }
    }
}

```

Треба приметити да пошто се блокирајући пријем одговора (*future.get()*) обавља унутар исте нити у којој се задатак предаје скупу нити на обраду овај сервер се ефективно понаша као да се целокупни посао обавља унутар једне једине нити.

➔ Уколико је потребно обезбедити да се неки посао покрене у неком тачно одређеном тренутку у будућности или га је потребно покретати периодично може се користити класа *Timer*. Ова класа омогућава извршавање послова, инстанце класе *TimerTask*, у позадинској нити. Сваком *Timer* објекту одговара једна позадинска нит у којој се задати послови секвенцијално извршавају. Ови послови би требали да буду краткотрајни. Уколико то није случај може се десити да се преостали послови не могу обавити у специфицирано време и са задатом динамиком.

**119****Клијент-сервер архитектура - комуникација**

Написати систем класа које омогућавају слање и пријем порука различитог типа у клијент-сервер архитектури.

**Решење**

Комуникација између клијентске и серверске стране се најчешће обавља користећи приклучнице. Како би се моделовао и потенцијално другачији вид комуникација уводи се апстракција објекта за комуникацију *Communicator*. Ради се о објекту који уме да транспортује поруке између клијента и сервера, али који не уме да тумачи примљене поруке. Овај објекат има методе које су типа *read* и *write* помоћу којих може да прима и шаље поруке одређеног типа. Узето је да објекат за комуникацију уме да транспортује најчешће податке које размењују клијент и сервер, објекте стрингове, поруке, и низове бајтова као најопштији вид комуникације. Поред ових метода објект има и методе за иницијализацију (*init*), пражњење бафера за слање (*flush*), и ресетовање комуникације и свих интерних бафера како би се избегло да систем остане без расположиве меморије (*reset*). Пошто објекат за комуникацију користи и приступа ресурсима узето је да овај објекат може да се користи унутар *try-with-resources* блока како би се аутоматски ослободили коришћени ресурси. Ово је постигнуто на тај начин што је стављено да је комуникациони објекат у сродству са интерфејсом *AutoCloseable* који има методу *close* која се аутоматски позива у финалној (*finally*) грани *try-with-resources* блока.

```
public interface Communicator extends AutoCloseable{

    public <T> Message<T> readMessage()
        throws CommunicationException;
    public <T> void writeMessage(Message<T> data)
        throws CommunicationException;

    public Object readObject()
        throws CommunicationException;
    public void writeObject(Object data)
        throws CommunicationException;

    public String readString()
        throws CommunicationException;
    public void writeString(String data)
        throws CommunicationException;

    public byte[] read() throws CommunicationException;
    public void write(byte[] data) throws CommunicationException;

    public void init();
    public void flush() throws CommunicationException;
    public void reset() throws CommunicationException;
}
```

Класа која имплементира интерфејс *Communicator* користећи објектне токове података је *ObjectSocketCommunicator*. Приликом иницијализације отварају се објектни токови

података између две стране које комуницирају, клијентске и серверске стране. Отварање токова података је реализовано тако да се исти код може користити и на клијентској и на серверској страни. Овде још једном треба напоменути да приликом отварања објектних токова података треба водити рачуна о редоследу по коме се ови токови података отварају. Постоји конструктор улазног тока података блокирајући чита заглавље тока како би се верификовало да је излазни ток са којим се повезује одговарајући барем једна страна прво отвара излазни ток података па тек онда улазни ток података. Уколико би обе стране прво покушале да отворе улазни објектни ток података дошло би до мртвог блокирања, *deadlock*. Узето је да је максимална дужина низа који се прима 1024. То се може променити променом вредности *BUFFSIZE* или тако што би се пре уписа низа у ток прво слала дужина низа користећи методу *writeln(int val)* класе *ObjectOutputStream* па онда сам низ, док би се на страни пријема прво читала послата дужина низа користећи методу *readInt()* класе *ObjectInputStream* а после тога читao низ ове дужине.

```

import java.net.*;
import java.util.*;
import java.io.*;

public class ObjectSocketCommunicator implements Communicator {

    protected Socket client;
    protected ObjectOutputStream oout;
    protected ObjectInputStream oin;

    public ObjectSocketCommunicator() {
    }

    public ObjectSocketCommunicator(Socket client) {
        this.client = client;
    }

    public void init(Socket client) {
        this.client = client;
        init();
    }

    @Override
    public void init() {
        getOutputStream();
        getInputStream();
    }

    private boolean getOutputStream() {
        try {
            OutputStream out = client.getOutputStream();
            oout = new ObjectOutputStream(out);
            return true;
        } catch (Exception e) {
            return false;
        }
    }
}

```

```
private boolean getObjectInputStream() {
    try {
        InputStream in = client.getInputStream();
        oin = new ObjectInputStream(in);
        return true;
    } catch (Exception e) {
        return false;
    }
}

@Override
public void close() {
    try {
        oin.close();
    } catch (IOException ex) {
    }
    try {
        oout.close();
    } catch (IOException ex) {
    }
    try {
        client.close();
    } catch (IOException ex) {
    }
}

@Override
public Object readObject() throws CommunicationException {
    try {
        Object m = oin.readObject();
        return m;
    } catch (Exception ex) {
        ex.printStackTrace();
        throw new CommunicationException();
    }
}

@Override
public void writeObject(Object m)
    throws CommunicationException{
    try {
        oout.writeObject(m);
    } catch (Exception ex) {
        ex.printStackTrace();
        throw new CommunicationException();
    }
}

@SuppressWarnings("unchecked")
@Override
public <T> Message<T> readMessage()
    throws CommunicationException {
    return (Message<T>) this.readObject();
}
```

```
@Override
public <T> void writeMessage(Message<T> m)
    throws CommunicationException {
    this.writeObject(m);
}

@Override
public String readString() throws CommunicationException {
    return (String) this.readObject();
}

@Override
public void writeString(String s)
    throws CommunicationException{
    this.writeObject(s);
}

@Override
public void flush() throws CommunicationException {
    try {
        oout.flush();
    } catch (Exception e) {
        e.printStackTrace();
        throw new CommunicationException();
    }
}

@Override
public void reset() throws CommunicationException {
    try {
        oout.reset();
        oin.reset();
    } catch (Exception e) {
        e.printStackTrace();
        throw new CommunicationException();
    }
}

static final int BUFFSIZE = 1024;
@Override
public byte[] read() throws CommunicationException {
    byte[] b = new byte[BUFFSIZE];
    try {
        int num = oin.read(b);
        b = Arrays.copyOf(b, num);
    } catch (Exception e) {
        e.printStackTrace();
        throw new CommunicationException();
    }
    return b;
}

@Override
public void write(byte[] data) throws CommunicationException {
    try {
```

```

        oout.write(data);
    } catch (Exception e) {
        e.printStackTrace();
        throw new CommunicationException();
    }
}

public class CommunicationException extends Exception {
    private static final long serialVersionUID = 1L;

    public CommunicationException() {
        super();
    }

    public CommunicationException(String error) {
        super(error);
    }
}
}

```

Објекат који се транспортује користећи објектне токове података треба да, као и сва његова поља рекурзивно који нису простог типа имплементирају интерфејс *Serializable*. Овај интерфејс нема метода које је потребно имплементирати већ даје индикацију виртуелном машини о начину на који треба сместити дати објекат. Узето је да класа порука која се транспортује имплементира генерички интерфејс *Message<T>*. Овај интерфејс пружа методе за дохватање и постављање тела поруке и за дохватање и постављање идентификатора поруке.

```

import java.io.*;

public interface Message<T> extends Serializable {

    public void setBody(T body);

    public T getBody();

    public void setId(long id);

    public long getId();
}

```

Класа која ради са текстуалним порукама *TextMessage* имплементира интерфејс *Message*, али са конкретном вредношћу за генерички параметар *<T>* који треба да буде стринг, *Message<String>*. Овде још једном треба напоменути да приликом транспорта користећи објектне токове података класа која се транспортује треба да имплементира интерфејс *Serializable*. Ово значи да је било могуће поставити да интерфејс *Message* не буде у сродству са интерфејсом *Serializable*, већ да то буде само класа која имплементира интерфејс *Message*. Ово је могуће јер је у програмском језику Java дозвољено да нека класа имплементира већи број интерфејса.

```

public class TextMessage implements Message<String> {
    private static final long serialVersionUID = 1L;
}

```

```
static long cnt = 0;

String body;
long id;

public TextMessage() {
    this("", createID());
}

public TextMessage(String s) {
    this(s, createID());
}

public TextMessage(String body, long id) {
    this.body = body;
    this.id = id;
}

public TextMessage(TextMessage m) {
    this(m.getBody(), m.getId());
}

public String toString() {
    String s = "";
    s += "Message ID " + id + " message: " + body;
    return s;
}

@Override
public void setBody(String body) {
    this.body = (String) body;
}

@Override
public String getBody() {
    return body;
}

@Override
public void setId(long id) {
    this.id = id;
}

@Override
public long getId() {
    return id;
}

public static synchronized long createID() {
    return cnt++;
}
}
```

**120****Клијент-сервер архитектура - протоколи**

Написати систем класа које омогућавају комуникацију између клијента и сервера користећи различите типове и редоследе порука.

**Решење**

Поред увођења објекта за комуникацију, *Communicator*, који уме да транспортује поруке између клијента и сервера, али који не уме да тумачи примљене поруке, уводи се још једна апстракција - то је објекат *Protocol*. Овај објекат уме да тумачи поруке, познаје редослед по коме се поруке примају и шаљу, али сам не уме да прима и шаље поруке. Коришћењем интерфејса *Communicator* и *Protocol* комуникација је подељена на слој за транспорт порука (не нужно користећи прикључнице) и слој за тумачење порука и поступање према тим порукама. Слој за тумачење порука се разликује на клијентској и на серверској страни. Због тога ће инстанце ове класе обично бити упарене. Постојаће две верзије протокола, једна је серверски протокол, а друга је клијентски протокол. Основна метода овог интерфејса је метода *conversation* кроз коју се обавља сва комуникација између две стране. Клијентски и серверски протокол треба да имају упарену ову методу. Ако једна страна шаље поруку, друга страна треба да очекује поруку, и обратно, како би се избегло мртво блокирање. Поред методе за комуникацију овај интерфејс пружа и методе за повезивање са интерфејсом за транспорт порука.

```
public interface Protocol {
    public void conversation();
    public void addCommunicator(Communicator communicator);
    public boolean removeCommunicator(Communicator communicator);
    public Communicator getCommunicator();
}
```

Нит која обавља комуникацију између клијента и сервера (*WorkingThread*) је идентична без обзира на то да ли се ради о клијентској или серверској страни, као и од тога на који начин се комуникација обавља. Основна метода ове нити је метода *work* која се позива из методе *run*. Ова нит на почетку дохвата комуникациони објекат као ресурс који ће на крају бити аутоматски затворен. Након овога користећи фабричку класу *ProtocolFactory* креира нову инстанцу протокола који ће обављати сву комуникацију између клијента и сервера. Повезује креирани протокол и објекат за комуникацију и започиње комуникацију. Када се комуникација заврши аутоматски затвара комуникациони објекат.

```
import java.net.*;
public class WorkingThread extends Thread {
    protected long id;
    protected String protocolName;
    protected Socket client;

    public WorkingThread(long id, Socket client, String protocol) {
        super("Working Thread " + id);
        this.id = id;
```

```
        this.client = client;
        this.protocolName = protocol;
    }

@Override
public void run() {
    work();
}

Protocol protocol;

public void work() {
    try (Communicator communicator = getCommunicator();) {
        protocol = pf.createProtocol(protocolName);
        if (protocolName == null)
            return;
        communicator.init();
        protocol.addCommunicator(communicator);
        protocol.conversation();

    } catch (Exception ex) {
        ex.printStackTrace();
    }
}

public Socket getClient() {
    return client;
}

public Communicator getCommunicator() {
    return new ObjectSocketCommunicator(getClient());
}

static ProtocolFactory pf;
private static Object protocolHandlerLock = new Object();

static void setProtocolFactory(ProtocolFactory protocol) {
    synchronized (protocolHandlerLock) {
        if (pf != null) {
            throw new Error("factory already defined");
        }
        SecurityManager security =
            System.getSecurityManager();
        if (security != null) {
            security.checkSetFactory();
        }
        pf = protocol;
    }
}

static {
    setProtocolFactory(new ProtocolFactory());
}
}
```

Фабричка класа *ProtocolFactory* креира нову инстанцу протокола користећи име протокола који треба да се имплементира и особину рефлексије у програмском језику Java. У програмском језику Java је могуће креирати инстанцу неке класе само на основу имена. Како би се ово постигло прво се дохвата *Class* објекат који представља класу објекта који ће бити креiran. Дохватање овог објекта је могуће остварити користећи методу *forName* класе *Class*. Овом приликом се може дододити изузетак, *ClassNotFoundException*, да класа из неког разлога није креирана, можда зато што је креирана након покретања датог програма. У овом случају се може поново учитати класа за учитавање класа, *ClassLoader*, и позвати њена метода *loadClass*. Креирање нове инстанце нека класе се постиже позивом методе *newInstance* над декларисаним конструктором без аргумента инстанциом класе *Class*. Дохватање конструктора (*Constructor<T>*) без аргумента се постиже позивом методе *getDeclaredConstructor()*. Уколико тај конструктор не постоји десиће се изузетак *NoSuchMethodException*. Приликом креирања нове инстанце класе узето је да се задаје само кратко име класе, а да се име пакета коме та класа припада дохвата динамички. То име се формира на основу имена пакета у коме се налази интерфејс *Protocol*, добијеног са *Protocol.class.getPackage().getName()*, унутрашњег пакета ".protocol." и кратког имена класе. У програмском језику Java је, поред позивања конструктора, могуће позивате методе и приступати променљивама користећи рефлексију.

```

public class ProtocolFactory {
    public Protocol createProtocol(String protocolName) {
        Protocol protocol = null;
        String protocolPackageRoot =
            Protocol.class.getPackage().getName();
        try {
            String clsName = protocolPackageRoot
                + ".protocol." + protocolName;
            Class<?> cls = null;
            try {
                cls = Class.forName(clsName);
            } catch (ClassNotFoundException e) {
                ClassLoader cl =
                    ClassLoader.getSystemClassLoader();
                if (cl != null) {
                    cls = cl.loadClass(clsName);
                }
            }
            if (cls != null) {
                protocol = (Protocol)
                    cls.getDeclaredConstructor().newInstance();
            }
        } catch (Exception e) {
        }
        return protocol;
    }
}

```

## 121

## Клијент-сервер архитектура - примери

Написати систем класа које користећи развијене класе реализују клијет сервер архитектуру.

## Решење

У овом задатку су дата два примера комуникације између клијентске и серверске стране. У првом примеру комуникација се обавља између клијента који користи протоколе *ClientProtocol1* и сервера који користи протокол *ServerProtocol1*. Ова комуникација је доста једноставна и састоји се из слања захтева серверу, пријема одговора од сервера и слања потврде серверу.

Друга комуникација је знатно сложенија и има за циљ да опише дечију игру „Куц, куц, ко је?“. Код ове комуникације клијент (*ClientProtocolKnockKnock*), који представља интерактивног корисника, прво шаље серверу своју идентификацију, па када сервер (*ServerProtocolKnockKnock*) потврди идентитет клијента креће у даљу комуникацију. Комуникација се састоји из тражења расположивих ресурса. Уколико постоје узимају се уколико не постоје покушава се поново све док се не успе. Уколико у било ком тренутку сервер прими неисправну поруку од клијента комуникација се прекида.

```

public interface Buffer<T> {
    public static final int MAXBUFFERSIZE = 150;

    public T get();

    public void put(T data);

    public void remove(T data);

    public int size();

    public int capacity();
}

import java.util.concurrent.*;

public class BlockingBuffer<T> implements Buffer<T> {
    protected BlockingQueue<T> buffer;
    protected int capacity;

    public BlockingBuffer() {
        this(MAXBUFFERSIZE);
    }

    public BlockingBuffer(int capacity) {
        if ((capacity < 0) || (capacity > MAXBUFFERSIZE)) {
            capacity = MAXBUFFERSIZE;
        }
        buffer = new LinkedBlockingQueue<T>(capacity);
        this.capacity = capacity;
    }
}

```

```

}

public T get() {
    while (true) {
        try {
            return buffer.take();
        } catch (InterruptedException e) {
        }
    }
}

public void put(T value) {
    while (true) {
        try {
            buffer.put(value);
            return;
        } catch (InterruptedException e) {
        }
    }
}

public void remove(T data) {
    buffer.remove(data);
}

public int size() {
    return buffer.size();
}

public int capacity() {
    return capacity;
}
}

public class ClientProtocol1 extends OneCommunicatorProtocol {

public ClientProtocol1() {
}

public ClientProtocol1(Communicator communicator) {
    this.communicator = communicator;
}

public void conversation() {

    try (Communicator communicator = getCommunicator();) {
        communicator.writeString("Hello!");
        System.out.println("Sent Hello!");

        String response = communicator.readString();
        System.out.println("Received " + response);
        TextMessage msg =
            new TextMessage("Message tekst " + response);
        communicator.writeObject(msg);
    } catch (Exception ex) {
}
}
}

```

```

        ex.printStackTrace();
    }
}
}

public class ServerProtocol1 extends OneCommunicatorProtocol {
    public ServerProtocol1() {
    }

    public ServerProtocol1(Communicator c) {
        this.communicator = c;
    }

    public void conversation() {
        try (Communicator communicator = getCommunicator();) {
            String response = communicator.readString();
            System.out.println("received " + response);

            communicator.writeString("Goodbye!");
            System.out.println("Sent Goodbye! ");

            TextMessage m =
                (TextMessage) communicator.readObject();
            System.out.println(m);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

import java.io.*;

public class ClientProtocolKnockKnock extends
OneCommunicatorProtocol {

    public ClientProtocolKnockKnock() {
        super();
    }

    public void conversation() {
        String fromServer = "";
        String fromUser = "";
        try (Communicator communicator = getCommunicator();) {
            BufferedReader stdIn = new BufferedReader(
                new InputStreamReader(System.in));
            System.out.print("Klient: ");
            fromUser = stdIn.readLine();
            communicator.writeString(fromUser);
            while (
                (fromServer = communicator.readString()) != null) {
                System.out.println("Server: " + fromServer);
                if (fromServer.equals(byeAnswer))
                    break;
            }
        }
    }
}

```

```

        System.out.print("Klijent: ");
        fromUser = stdIn.readLine();
        if (fromUser != null) {
            communicator.writeString(fromUser);
        }
    }
    System.out.println("-----");
} catch (Exception e) {
    e.printStackTrace();
}
}

import java.util.*;

public class ServerProtocolKnockKnock extends
OneCommunicatorProtocol {
    private static final int WAITING = 0;
    private int state = WAITING;

    public static String[] eggs = { "Belo", "Plavo", "Crveno",
        "Zeleno", "Zuto", "Crno" };
    private static Set<String> eggsSet = new HashSet<String>();
    public static String[] questions = { "Kuc kuc",
        "Djavo 's neba", "Jedno jaje" };
    public static String[] answers = { "Ko je?", "Sta Vam treba?",
        "Koje boje?" };
    public static String yesAnswer = "Ima. ";
    public static String noAnswer = "Nema ";
    public static String byeAnswer = "Zdravo!";

    public ServerProtocolKnockKnock() {
        super();
    }

    public void conversation() {
        try (Communicator communicator = getCommunicator();) {
            String outputLine = null;
            String inputLine = null;
            while (
                (inputLine = communicator.readString())!=null) {
                System.out.println("Klijent: " + inputLine);
                outputLine = processInput(inputLine);
                communicator.writeString(outputLine);
                System.out.println("Server: " + outputLine);
                if (outputLine.equalsIgnoreCase(byeAnswer))
                    break;
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public String processInput(String input) {

```

```

        String output = null;
        if (input == null)
            input = "";
        switch (state) {
        case 0:
        case 1:
        case 2:
            if (input.equalsIgnoreCase(questions[state])) {
                output = answers[state];
                state++;
            } else {
                output = byeAnswer;
                state = 0;
            }
            break;
        case 3:
            if (eggsSet.contains(input.toLowerCase())) {
                output = yesAnswer;
                state++;
            } else {
                output = noAnswer + input + ". " + answers[2];
            }
            break;
        default:
            output = byeAnswer;
            break;
        }
        return output;
    }

    static {
        for (String egg : eggs) {
            eggsSet.add(egg.toLowerCase());
        }
    }
}

import java.net.*;

public class ClientTest1 {
    static final int port = 8080;
    static final String host = "127.0.0.1";
    static final String protocol = "ClientProtocolKnockKnock";

    public static void main(String[] args) {
        for (int i = 0; i < 10; i++) {
            try (Socket server = new Socket(host, port);) {
                System.out.println("Iteration " + i);
                WorkingThread wt =
                    new WorkingThread(i, server, protocol);
                wt.run();
                Thread.sleep(500+(int)(Math.random() * 1000));
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}

```

```
        }
    }
}

public class ServerTest1 {

    static final int port = 8080;
    static final String protocol = "ServerProtocolKnockKnock";
    static final int numThreads = 10;

    public static void main(String[] args) {
        Server server = new OneThreadServer(protocol, port);
        Server server = new MultiThreadsServer(protocol, port);
//        Server server =
//            new BufferWorkingServer(numThreads, protocol, port);
        Server server =
            new ExecutorServer(numThreads, protocol, port);
        Server server =
            new ExecutorFutureServer(numThreads, protocol, port);
        server.start();
    }
}
```

## **Java - RMI**

Објекти у програмском језику Java којима се до сада приступало су имали особину да морају да се налазе у истом адресном простору, на истом рачунару, као и ток контроле из кога им се приступало. Уколико је било потребно комуницирати са неким другим рачунаром није било могуће директно приступити његовим објектима. Било је могуће по одређеном протоколу разменити одређен скуп порука како би се комуникације обавила. Ова комуникација је понекад уређена на тај начин што се приликом слања порука шаље коју је операцију потребно урадити, као и аргументе операције, након чега позвани извршавају позвану процедуру и шаље резултат назад. Овај начин комуникације би одговарао процедуралном програмирању, и назива се удаљеним позивима процедура (*Remote Procedure Call - RPC*).

Како би се олакшао директан приступ дистрибуираним објектима у програмски језик Java је уведен концепт удаљеног позива метода објекта (*Remote Method Invocation - RMI*). Коришћење удаљених позива метода у програмском језику Java представља комплетно нову парадигму која подржава концепте објектно оријентисаног програмирања. Код ове парадигме циљ је да се објекти не морају налазити на истом рачунару већ се могу налазити на оном рачунару на коме постоји ресурси (процесор, меморија, привилегије, фајлови, базе података, ...) за дати објекат. Комуникација између објекта се и овде обавља разменом порука, само су те поруке сакrivене од корисника. Овде је одрађен известан ниво енкапсулације и олакшано дистрибуирано програмирање. Објектима се приступа користећи додатни спој удаљених референци. Класе и интерфејси која омогућавају овај вид комуникације су већим својим делом унутар пакета *java.rmi*.

### **Креирање интерфејса удаљеног објекта**

За све објекте којима ће се приступати удаљено мора се креирати интерфејс преко кога се приступа одговарајућем објекту:

```
import java.rmi.*;  
  
public interface MyRemote extends Remote{  
  
    public void f(...) throws RemoteException;  
  
    ...  
}
```

Овај интерфејс мора да буде изведен из интерфејса *Remote* који нема ни једну методу у себи коју је потребно имплементирати, али омогућује преводиоцу да генерише одговарајући код. Свака метода овог интерфејса мора да декларише могућност емитовања изузетка типа *RemoteException*.

### **Имплементација интерфејса удаљеног објекта**

Приликом имплементације интерфејса потребно је ставити да ова класа имплементира предложени интерфејс. Може се поставити и да конструктор удаљеног објекта еmitује изузетак.

```
import java.rmi.*;  
  
public class MyRemoteImpl implements MyRemote{
```

```

...
public MyRemoteImpl() {
    ...
}
public void f(...) throws RemoteException {
    ...
}
}

```

У верзијама пре 1.5 било је потребно и да класа наслеђује класу за рад са удаљеним објектима *UnicastRemoteObject*. Овако креирана класа се преводи на посебан начин да би им се обезбедио приступ удаљеном објекту.

```
xmic MyRemoteImpl
```

Приликом тог превођења креирала се класа која треба да води рачуна о удаљеним референцима. Том прилико се креирала класа *\_Stub*.

#### Повезивање имена објекта са инстанцом објекта

Да би дошло до комуникације потребно је на серверској страни креирати објекат и након тога повезати га са додељеним именом.

```

import java.rmi.*;
import java.rmi.registry.*;
import java.rmi.server.*;

...
try {
    if (System.getSecurityManager() == null) {
        System.setSecurityManager(new SecurityManager());
    }
    MyRemoteImpl myObject = new MyRemoteImpl();
    MyRemote stub = (MyRemote)
        UnicastRemoteObject.exportObject(myObject, 0);
    Registry registry = LocateRegistry.createRegistry(port);
    String name = "/instanceName";
    registry.rebind(name, stub);
} catch(RemoteException e) {
    ...
}

```

Пошто се овде ради о приступу референцима које нису унутар једне виртуалне машине да би приступ функционисао потребно је поставити ниво заштите на одговарајућу вредност. То се постиже постављањем класе за управљање заштитом,

`System.setSecurityManager(new SecurityManager())`, уколико већ није постављена. Уколико је одговарајући ниво заштите већ постављен и није дозвољено постављање нове заштите (`checkPermission`) долази до изузетка:

```
java.lang.SecurityException
```

Формирање објекта коме се може удаљено приступити се постиже користећи класу `UnicastRemoteObject`. Како би се постојећи објекат, који имплементира интерфејс `Remote`, прилагодио да прима долазне позиве потребно је позвати статичку методу `exportObject`. Ова метода поред датог објекта још може да прими порт преко кога се ради експорт, као и фабрике који креирају долазне и одлазне прикључнице које се користе приликом комуникације. Други начин да се ова класа искористи за приступ објекту је да класа која имплементира интерфејс `Remote` постави и да проширије класу `UnicastRemoteObject`. На овај начин се може постићи да класа учини доступном споља без директног уписивања у регистар.

Када се припреми објекат потребно је пронаћи одговарајући регистар у који се дати објекат може уписати. Класа која омогућава проналажење одговарајућег регистра је `LocateRegistry`. Ова класа пружа скуп метода које проналазе постојећи регистар, `getRegistry`, и оних које покрећу нови регистар на датом рачунару, `createRegistry`. Покретање и припрема регистра на неком порту датог рачунара се постиже користећи статичку методу `createRegistry(port)` класе `LocateRegistry`. Аргумент `port` се може изоставити и у том случају је подразумевани порт 1099. Уколико регистар не може да се припреми еmitује се изузетак `RemoteException`.

Уколико се жели приступ постојећем регистру који се извршава као независна серверска апликација потребно је покренути посебан програм који води рачуна о спољним позивима и ради повезивање. Овај програм је независтан од датог Java програма и се покреће са:

```
rmiregistry [-J<runtime flag>] [port]
```

Да би се омогућио приступило неком удаљеном објекту потребно је у одговарајући регистар доступних удаљених објеката под одговарајућим именом убацити дати објекат. Повезивање имена са објектом се постиже позивом методе `rebind(name, stub)` класе `Registry`. Уколико је раније постојало исто име, објекат на који је оно указивало ће бити игнорисано, а уместо њега ће бити постављен нови објекат. Приликом повезивања може доћи до већег броја изузетака. Уколико је дошло до прекида комуникације еmitује се `RemoteException`. Уколико је регистар локални и ако не дозвољава да се дата метода позове еmitује се `AccessException`. Такође се еmitује `NullPointerException` уколико је или име или објекат `null`.

Поред ове методе класе `Registry` омогућава и дохватање објекта из регистра (`lookup`), излиставање свих имена у регистру (`list`), брисање објекта из регистра (`unbind`), и повезивање имена са објектом (`bind`).

### Добијање референце на удаљени објекта на клијентској страни

На клијентској страни је потребно извршити повезивање удаљене референце са именом објекта.

```
import java.rmi.registry.*;  
...  
try {  
    if (System.getSecurityManager() == null) {  
        System.setSecurityManager(new SecurityManager());  
    }  
}
```

```

Registry registry = LocateRegistry.getRegistry(host, port);
String name = "/instanceName";
Object remote = registry.lookup(name);
MyRemote reply = (MyRemote)remote;

} catch(RemoteException nbe) {
    ...
}

```

Пошто се и овде ради о приступу референцама које нису унутар једне виртуалне машине потребно је поставити ниво заштите на одговарајућу вредност користећи `System.setSecurityManager(new SecurityManager())`. Уколико је заштита већ постављена и није дозвољено постављање нове заштите (`checkPermission`) долази до изузетка:

```
java.lang.SecurityException
```

Како би се приступило удаљеном објекту прво је потребно пронаћи регистар у коме се чува удаљена референца на објекат. Класа која омогућава проналажење одговарајућег регистра је и у овом случају `LocateRegistry`. За разлику од креирања удаљеног објекта, где је било могуће позивати било `getRegistry` било `createRegistry`, у овом случају је потребно позвати методу за дохватање постојећег регистра `getRegistry`. Овој методи се као параметри прослеђују име рачунара и порт рачунара на коме се налази подигнут регистар. Уколико референца на регистар не може да се креира еmitује се изузетак `RemoteException`.

Дохватање удаљене референце објекта се постиже позивом методе `lookup(name)` класе `Registry`, при чему поље `name` представља име под којим је дељени објекат уписан у регистар. Приликом повезивања може доћи до већег броја изузетака. Уколико под датим именом није било уписаных објекта долази до изузетка `NotBoundException`. Уколико је дошло до прекида комуникације еmitује се `RemoteException`. Уколико је регистар локални и ако не дозвољава да се дата метода позове еmitује се `AccessException`. Такође се еmitује `NullPointerException` уколико је име `null`.

На крају је потребно извршити експлицитну конверзију типа.

## Заштита

Да би се приступало удаљеном објекту потребно је поставити одређен ниво заштите користећи политике заштите која пружа Java. Нивои заштите се дефинишу у посебном документу који садржи већи број додељених привилегија (*grant*). Свака привилегија садржи једну или више дозвола (*permission*) и спецификацију на шта/кога се односи. Основни облик дозволе је:

```

grant signedBy "signer_names", codeBase "URL",
principal principal_class_name "principal_name",
...
permission permission_class_name "target_name", "action",
signedBy "signer_names";

```

};

Привилегија се може односити на место где се налази код (*codeBase*), на онога кој је потписао код (*signedBy*), као и на класу која мора бити присутна у скупу одговараних класа дате нити (*principal*). Код коме се дозвољава приступ не мора да садржи са само адресу (*URL*) одакле потиче, већ и референцу на одговарајуће јавне кључеве који одговарају тајним кључевима којима је код потписан. За код се разматра не само ко га је написао, већ и ко извршава дати код.

Свака привилегија започиње речју *permission*. Након ње следе о ком типу привилегије се ради (*permission\_class\_name*). Ови типови би укључивали рад са фајловима *java.io.FilePermission* или приступ извршном окружењу *java.lang.RuntimePermission*. Поље са специфицираном акцијом, *action*, је опционо, а ближе специфицира приступ. Поље *signedBy* специфицира онога ко је потписао класу у којој су специфициране привилегије.

Овде је дат пример једне такве политике која свима дозвољава комплетан приступ ресурсима:

```
java.policy

grant {

    // Allow everything for now

    permission java.security.AllPermission;

};

}
```

Укључивање заштите и додавање одговараће политике се постиже користећи аргументе виртуелној машини.

**-Djava.security.policy=путања\_до\_java.policy\_фајла**

Спецификација адресе са које се могу преузети класе битне за дељене објекте (на пример: класа дељеног објекта (*stub*), класе повратних типова метода удаљених објеката, или интерфејса које користе посредничке (*proxy*) или дељене класе).

**-Djava.rmi.server.codebase=file:\\ путања**

122

## Дељени рачун

Решити проблем заједничког рачуна у банци (*The Savings Account Problem*) користећи удаљене позиве метода у Јави.

**Решење**

```

import java.rmi.*;
public interface Bank extends Remote {
    public UserAccount getUserAccount(String name)
        throws RemoteException;
}

import java.rmi.*;
public interface UserAccount extends Remote {
    public float getStatus() throws RemoteException;
    public void transaction(float value) throws RemoteException;
}

import java.rmi.*;
import java.util.*;

public class BankImpl implements Bank {
    private static transient HashMap<String, UserAccount> users;

    public BankImpl() {
        users = new HashMap<String, UserAccount>();
    }

    @Override
    public UserAccount getUserAccount(String name)
        throws RemoteException {
        synchronized (this) {
            UserAccount user = users.get(name);
            if (user != null) {
                return user;
            }
            user = new UserAccountImpl(name);
            users.put(name, user);
            return user;
        }
    }
}

```

```
import java.rmi.*;
import java.rmi.server.*;
import java.io.*;

public class UserAccountImpl extends UnicastRemoteObject
    implements UserAccount, Serializable {

    private static final long serialVersionUID = 1L;
    private float status = 0;
    private String name;

    public UserAccountImpl(String name) throws RemoteException {
        this.name = name;
    }

    private void work() {
        ...
    }

    @Override
    public float getStatus() throws RemoteException {
        synchronized (this) {
            work();
            return status;
        }
    }

    @Override
    public void transaction(float value) throws RemoteException {
        synchronized (this) {
            work();
            while (status + value < 0) {
                try {
                    wait();
                } catch (InterruptedException ex) {
                }
            }
            status += value;
            notifyAll();
        }
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

```

import java.rmi.*;
import java.rmi.registry.*;
import java.rmi.server.*;

public class BankServer {

    public static String host = "127.0.0.1";
    public static int port = 8080;

    public static void main(String[] args) {

        try {
            if (System.getSecurityManager() == null) {
                System.setSecurityManager(
                    new SecurityManager());
            }

            Bank bank = new BankImpl();
            System.out.println("Kreirana je banka");

            Bank stub =
                (Bank)UnicastRemoteObject.exportObject(bank, 0);
            System.out.println("Eksportovana je banka");

            Registry registry =
                LocateRegistry.createRegistry(port);
            System.out.println("Definisan Registry "+registry);

            String name = "/Bank";
            registry.rebind(name, stub);
            System.out.println("Banka je sada spremna");

        } catch (RemoteException e) {
            System.out.println("Desila se greska: " + e);
            e.printStackTrace();
        }
    }
}

import java.rmi.registry.*;

public class BankClient {

    public static String host = "127.0.0.1";
    public static int port = 8080;

    public static void main(String[] args) {

        try {
            Bank bank = null;
            UserAccount userAccount = null;
            String name = args[0];

            if (System.getSecurityManager() == null) {

```

Треба приметити да класа *BankImpl* само имплементира интерфејс *Bank*, док класа *UserAccountImpl* проширује класу удаљених објеката *UnicastRemoteObject* и имплементира интерфејсе *UserAccount* и *Serializable*. Ово је урађено како би се омогућило да инстанцима обе класе може да се приступи као дељеним објектима, али на два различита начина. Први начин приступа је применењен приликом приступа удаљеној инстанци класе *BankImpl* која се добија користећи регистар у који је претходно уписана користећи методу *exportObject* класе *UnicastRemoteObject*. Други начин приступа је применењен код удаљене инстанце класе *UserAccountImpl* којој се приступа на основу референце коју враћа метода *getUserAccount* над удаљеним објектом типа *Bank*. Зато је потребно да класа *UserAccountImpl* проширује класу *UnicastRemoteObject* како би се урадио експорт оваквог објекта. Уколико класа *UserAccountImpl* не би проширивала класу *UnicastRemoteObject* већ само имплементира интерфејсе *UserAccount* и *Serializable* програм би имао другачије понашање. Инстанца класе која би се добила позивом методе *getUserAccount* над удаљеним објектом типа *Bank* би била копија објекта, а не удаљена референца.

123

Игра живота

Написати програм на језику Java који симулира проблем игре живота (*The Game of Life*) при чему сваки организам представља независтан програм.

## Решење

```

import java.io.*;
public class Msg implements Serializable {
    private static final long serialVersionUID = 1L;
    public int i, j;
    public int index;
    boolean status;

    public Msg(boolean status, int i, int j, int index) {
        this.i = i;
        this.j = j;
        this.index = index;
        this.status = status;
    }

}

import java.rmi.*;

public interface MsgBox extends Remote {
    public Msg get() throws RemoteException;
    public void put(Msg val) throws RemoteException;
}

import java.rmi.*;
public class MsgBoxImpl implements MsgBox {
    public transient Buffer<Msg> buffer;

    public MsgBoxImpl(){
        buffer = new ArrayBuffer<Msg>();
    }

    @Override
    public Msg get() throws RemoteException {
        return buffer.get();
    }

    @Override
    public void put(Msg val) throws RemoteException {
        buffer.put(val);
    }
}

```

```

public class Cell extends Thread {
    int i;
    int j;
    int numGenerations;
    MsgBox[][] box;
    boolean status;

    public Cell(int i, int j, MsgBox[][] box, int numGenerations) {
        this.i = i;
        this.j = j;
        this.box = box;
        this.numGenerations = numGenerations;
        status = createStatus(i, j);
    }

    @Override
    public void run() {
        try {
            Msg[][] neighbours = new Msg[2][8];
            int[] num = new int[2];
            num[0] = 0;
            num[1] = 0;
            for (int k = 1; k < numGenerations; k++) {
                Msg m = new Msg(status, i, j, k);
                for (int p = start(i); p <= end(i); p++) {
                    for (int q = start(j); q <= end(j); q++) {
                        if ((p != i) || (q != j)) {
                            box[p][q].put(m);
                        }
                    }
                }
                int numNeighbours = numOfNeighbours(i, j);
                while (num[k % 2] < numNeighbours) {
                    m = box[i][j].get();
                    neighbours[m.index%2][num[m.index%2]] = m;
                    num[m.index % 2] = num[m.index % 2] + 1;
                }
                num[k % 2] = 0;
                status = calculateState(i, j, k, neighbours);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public int start(int coordinate) {
        if (coordinate <= 0) {
            return 0;
        } else {
            return coordinate - 1;
        }
    }

    public int end(int coordinate) {
        if (coordinate < (box[0].length - 1)) {

```

```

        return coordinate + 1;
    } else {
        return box[0].length - 1;
    }
}

public int numOfNeighbours(int x, int y) {
    return (end(x)-start(x)+1) * (end(y)-start(y)+1)-1;
}

public boolean calculateState(int i, int j, int k,
    Msg[][] neighbours) {
    return ...
}

public boolean createStatus(int i, int j) {
    return ...
}

import java.rmi.registry.*;
import java.rmi.server.*;

public class GameOfLifeServer {
    public static final int N = 3;

    public static String host = "127.0.0.1";
    public static int port = 8080;

    public static void main(String[] args) {

        try {
            if (System.getSecurityManager() == null) {
                System.setSecurityManager(
                    new SecurityManager());
            }

            Registry registry =
                LocateRegistry.createRegistry(port);
            System.out.println("Definisan Registry port");

            System.out.println("Kreiranje sanducica");
            MsgBox[][] box = new MsgBox[N][N];
            for (int i = 0; i < N; i++) {
                for (int j = 0; j < N; j++) {
                    MsgBox msgBox = new MsgBoxImpl();
                    MsgBox stub = (MsgBox)
                        UnicastRemoteObject.
                            exportObject(msgBox, 0);
                    box[i][j] = stub;
                }
            }
            String name = "/Box";
            for (int i = 0; i < N; i++) {
                for (int j = 0; j < N; j++) {

```

```

        registry.rebind(name+"-"+i+"-"+j,box[i][j]);
    }
}
System.out.println("Sanducici su povezani");

} catch (Exception e) {
    e.printStackTrace();
}
}

import java.rmi.registry.*;

public class GameOfLifeClient {
    public static final int N = 3;
    public static final int NUMGENERATION = 10;

    public static String host = "127.0.0.1";
    public static int port = 8080;

    public static void main(String[] args) {
        MsgBox[][] box = new MsgBox[N][N];
        Cell[][] cells = new Cell[N][N];

        try {
            if (System.getSecurityManager() == null) {
                System.setSecurityManager(
                    new SecurityManager());
            }

            Registry registry =
                LocateRegistry.getRegistry(host, port);

            String name = "/Box";
            for (int i = 0; i < N; i++) {
                for (int j = 0; j < N; j++) {
                    box[i][j] = (MsgBox) registry.
                        lookup(name + "-" + i + "-" + j);
                }
            }
            for (int i = 0; i < N; i++) {
                for (int j = 0; j < N; j++) {
                    cells[i][j] =
                        new Cell(i, j, box, NUMGENERATION);
                    cells[i][j].start();
                }
            }
        } catch (Exception e) {
            System.err.println("Exception " + e);
            e.printStackTrace();
        }
    }
}

```

## Литература

- Andrews G. R., *Concurrent Programming: Principles and Practice*. Menlo Park, CA: Benjamin/Cummings, 1991
- Andrews G. R., *Foundations of Multithreaded, Parallel, and Distributed Programming*. Reading, MA: Addison-Wesley, 2000
- Bal H. E., Steiner J. G., Tanenbaum A. S., *Programming Languages for Distributed Computing Systems*. ACM Computing Surveys, Volume 21, Number 3, pp. 261-322 (September), 1989
- Ben-Ari M., *Principles of Concurrent and Distributed Programming*. Second edition, Pearson, 2006
- Carriero N., Gelernter D., *How to Write Parallel Programs: A First Course*. The MIT Press, 1990, Second Printing 1991
- Carriero N., Gelernter D., *How to Write Parallel Programs: A Guide to the Perplexed*. ACM Computing Surveys, Volume 21, Number 3, pp. 323-357, (September), 1989
- Carriero N., Gelernter D., *Linda in Context*. Communications of the ACM, Volume 32, Number 4, pp. 444-458 (April), 1989
- Downey B. Allen, *The Little Book of Semaphores*, Second edition. <http://greenteapress.com/semaforos/LittleBookOfSemaphores.pdf>, 2016
- Goetz B., Peierls T., Bloch J., Bowbeer J., Holmes D., Lea D., *Java Concurrency in Practice*. 1st Edition, Addison-Wesley, 2006
- Gosling J., Joy B., Steele G., Bracha G., Buckley A., *The Java Language Specification Java SE 8 Edition*. <https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf>, 13.02.2015
- Herlihy M., Shavit S., *The Art of Multiprocessor Programming*, Revised Reprint, Morgan Kaufmann, 2012
- Hoare C. A. R., *Communicating Sequential Processes*. Communications of the ACM, Volume 21, Number 8, pp. 666-677 (August), 1978
- Hoare C. A. R., *Communicating Sequential Processes*. <http://www.usingcsp.com/cspbook.pdf>, May 18, 2015
- Lea D., *Concurrent Programming in Java*, Second Edition. Reading, MA: Addison-Wesley, 2000
- Magee J., Kramer J., *Concurrency: State Models and Java Programs*, Second Edition. Wiley, 2006
- Tanenbaum A., van Steen M., *Distributed Systems: Principles and Paradigms*, Second Edition. CreateSpace, 2016

**Издавач**  
**Академска мисао**

Приморска 21, Београд  
Тел: +381 11 3218 354

Марко Вујадиновић, дипл. ел. инж.  
+381 63 30 10 75  
marko.vujadinovic@akademska-misao.rs

Александар Рашковић, дипл. ел. инж.  
+381 63 30 10 65  
sasa.raskovic@akademska-misao.rs

**www.akademska-misao.rs**  
**office@akademska-misao.rs**

СИР - Каталогизација у публикацији  
Народна библиотека Србије, Београд

004.42(075.8)  
РАДИВОЈЕВИЋ, Захарије, 1978-  
Конкурентно и дистрибуирано програмирање / Захарије  
Радивојевић, Игор Икодиновић, Зоран Јовановић. - 2. изд. -  
Београд : Академска мисао, 2018 (Београд : Академска мисао).  
- 541 стр. : граф. прикази ; 24 см

Тираж 300. - Библиографија: стр. 541.

ISBN 978-86-7466-724-8

1. Икодиновић, Игор, 1972- [автор]
2. Јовановић, Зоран, 1953- [автор]
- а) Програмирање

COBISS.SR-ID 257388812

