**LP PAGRINDINĖS SĄVOKOS.**

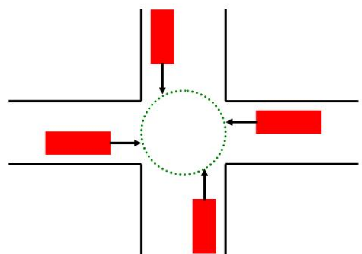
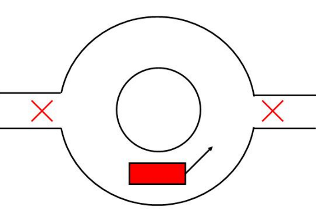
1. **Lygiagrečių programų nauda ir poreikis:**

* Programinis modelis geriau atitinka realųjį pasaulį
* Galima padidinti vykdymo greitį
* Iš anksto nustatyta veiksmų tvarka kai kuriems taikymas nėra priimtina
* Panaudojimas: Gamtos ir inžinerijos mokslų sudėtingų problemų sprendimas:
* atmosferos reiškinių prognozavimas;
* branduolinė ﬁzika, matematika;
* kosminių aparatų˛ ir ginklų projektavimas;
* Pramonė ir komercija:
* naftos ir dujų gavyba;
* farmacija;
* ﬁnansai ir ekonominis modeliavimas;

1. **Lygiagrečiojo programavimo problemos:**

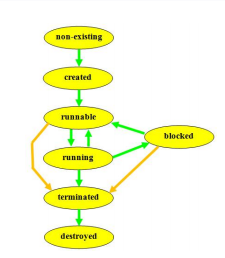
* Skirtingi rezultatai net ir su tais pačiais duomenimis
* Vartotoją tenkina tik tam tikri rezultatai su tam tikra veiksmų atlikimo seka
* Sinchronizuoti procesus
* Procesai turi keistis informacija

1. **Pagrindinės sąvokos:**

* *Atominis veiksmas* - veiksmas, kuriam vykstant nevyksta jokie kiti procesų veiksmai
* *Kritinė sekcija* - programos kodo gabalas, kuris kitų procesų atžvilgiu turi būti atominiu veiksmu
* *Tarpusavio išskyrimas* - priemonės, neleidžiančios dviem procesams vienu metu vykdyti savo veiksmus( naudoti tuos pačius resursus). Dviejų procesų kritinė sekcija turi būti vykdoma su tarpusavio išskyrimu
* *Varžymosi būsena* - tai būsena, kai procesai vienu metu varžosi dėl kokių tai resursų
* *Užimtas laukimas* - būsena, kai procesas laukia kokių tai resursų pastoviai tikrindamas jo užimtumą
* *Badavimas* - tai būsena, kai procesui pastoviai neleidžiama naudotis bendrais resursais
* *Aklavietė* - tai būsena, kai procesas laukia to, kas niekada neįvyks
* *Begalinis ciklas* - tai būsena, kai procesas cikliškai vykdo tuos pačius veiksmus be jokio progreso

**LYGIAGRETIEJI PROCESAI.**

1. **Lygiagretaus proceso sąvoka –** tai programos kodo fragmentas, nuosekliai vykdomas visame procesoriuje. Užrašymo tvarka, nenurodo vykdymo tvarkos. Tie patys duomenys – galimi skirtingi rezultatai. LP procesų skaičius ≥ fizinių procesorių skaičius; procesų skaičius = loginių procesorių skaičius.
2. **Lygiagrečių procesų tipai:**

* Statiniai ir dinaminiai procesai
* Vieno ir daugelio lygio procesai
* procesų inicializavimo ir užbaigimo skirtumai

1. **Lygiagrečių procesų savybės**

* *Komutatyvumas:*
* *Asociatyvumas:*
* *Tranzityvumas:*

1. **Lygiagrečių procesų būsenos**

* Neegzistuojąs (non-existing)
* Sukurtas (created)
* Pasiruošęs (ready, runnable)
* Vykstąs (running)
* Blokuotas (blocked)
* Nutrauktas (terminated)
* Sunaikintas (destroyed)

1. **Lygiagrečių procesų sąveikos tipai**

* *Procesai nepriklausomi* - priemonės procesų sąveika valdyti nereikalingos; variantas retai pasitaikantis
* *Procesai varžosi tarpusavyje* - priemonės procesų prioritetams nustatyti; būdinga sistemoms ,kuriose procesai sprendžia nepriklausomus uždavinius
* *Procesai bendradarbiauja tarpusavyje* - priemonės procesų sąveikai valdyti; būdinga sistemos, kuriose procesai sprendžia bendrą uždavinį.

1. **Lygiagrečiųjų procesų komunikavimas ir sinchronizavimas**
   * *OS:*
     + 1 programa 1 procesas;
     + 1 procesas daug (≥1) gijų;
     + Procesai neturi bendros atminties gijų; atmintis bendra;

* *LP:*
  + 1 programa daug (≥1) procesų;
  + 1 procesas ≡ 1 gija;
* *Bendra procesų atmintis:*
* procesai komunikuoja per bendrus kintamuosius
* procesai sinchronizuojamai naudojant bendrus kintamuosius
* problema- bendrų kintamųjų apsauga
* *Atskira procesų atmintis*
* Procesai komunikuoja siųsdami ir gaudami pranešimus
* Procesai sinchronizuojami siunčiant sinchronizavimo signalus
* Problema- priimti pranešimus iš kelių procesų
* *Atminties modelis ir kompiuterio architektūra:*
  + Programos architekūros modelis (bendra ar atskira atmintis) turi būti nepriklausomas nuo kompiuterio architektūros;
  + Programa vykdoma efektyviai, jei programos modelis sutampa su kompiuterio architektūros modeliu.
* *Sinchronizavimas:*
  + Procesai laukia, kartodami kokius nors veiksmus;
  + Procesai laukia, neatlikdami jokių veiksmų (yra blokuojami);

**JAVA PROCESAI (GIJOS)**

Gijos įtrauktos į bazinę Java kalbos speciﬁkaciją. Gija – lengvasvoris procesas, kurio viduje veiksmai vykdomi nuosekliai. Gijų naudojimas sudaro galimybę tuo pačiu metu (lygiagrečiai) vykdyti kelias programos užduotis. Programoje visada yra bent viena gija: main funkcija – taip pat gija. Kiekviena Java virtualioji mašina (JVM) privalo palaikyti gijas.

1. **Kūrimas ir inicializavimas.** 
   * Panaudojant (paveldint) java.lang.Thread klase˛.
   * Panaudojant (realizuojant) java.lang.Runnable sasaja˛

class Gija extends Thread

{

. . .

public void run()

{

. . .

}

} .

. . Gija g = new Gija();

g.start();

//-------------------------------

class Gija implements Runnable

{

. . .

public void run()

{

. . .

}

}

. . .

Thread g = new Thread(new Gija());

g.start();

1. **Vykdymas.** 
   * Programa lygiagrečiai (concurrent) gali vykdyti bet kokį skaičių gijų.
   * Gijos vykdymas pradedamas, iškvietus gijos-objekto metoda˛ *start*.
   * Gijos vykdymo metu nuosekliai atliekami veiksmai, užrašyti gijos klasės metode run.
   * Visos programos gijos gali atlikti skirtingus veiksmus (jei jos yra skirtingu˛ gijų klasių objektai).
   * Jei programuotojas netaiko papildomų priemonių, gijos-objekto vykdymas nepriklauso nuo kitų˛ gijų vykdymo.
   * Programose *run* metodas neturėtų būti kviečiamas tiesiogiai. Sistema iškvies jį pati.
   * Jei *run* metodas kviečiamas tiesiogiai, programos kodas vykdomas ne lygiagrečiai, bet nuosekliai.
2. **Nutraukimas.**

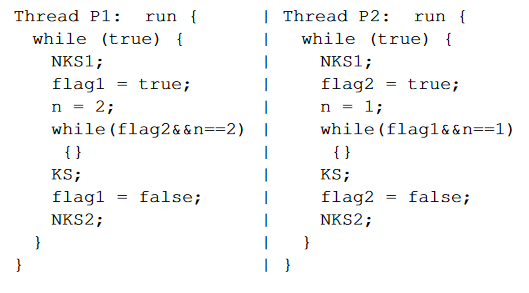
* *public final void join()* throws InterruptedException Waits for this thread to die.

1. **Bendros atminties naudojimas.**
   * Java, OpenMP gijos gali tureti bendrus kintamuosius;
2. **Komunikavimas ir sinchronizavimas.**
   * *Tarpusavio išskyrimas* (mutual exclusion) - "užrakinant" objekta˛su *synchronized* (realizuoja JVM).
   * *Sąlygine sinchronizacija* (condition synchronization) – naudojant Object klases wait, notify, notifyAll metodus. Sąlyginė sinchronizacija tai sincronizacijos procesas, kuomet viena gija siunčia pranešimą kitai gijai, kai tenkinama tam tikra sąlyga
     + *notify()* - Wakes up a single thread that is waiting on this object’s monitor. *notifyAll()* - Wakes up all threads that are waiting on this object’s monitor.
     + *wait()* Causes current thread to wait until another thread invokes the *notify()* method or the *notifyAll()* method for this object.
     + *wait(long timeout)* - Waits for notiﬁcation or until the timeout period has elapsed. timeout is measured in milliseconds

**KS APSAUGA IR SĄLYGINĖ SINCHRONIZACIJA.**

Jei vienas procesas yra NKS, o kitas nori įeiti į KS, reikia leisti jam tai daryti. Jei du procesai varžosi dėl perėjimo˛į KS, sprendimas, kuriam procesui leisti tai daryti, negali būti atidėtas neapibrėžtam laikui.

1. **Peterson’o algoritmo analizė.**

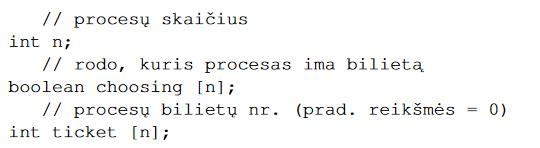


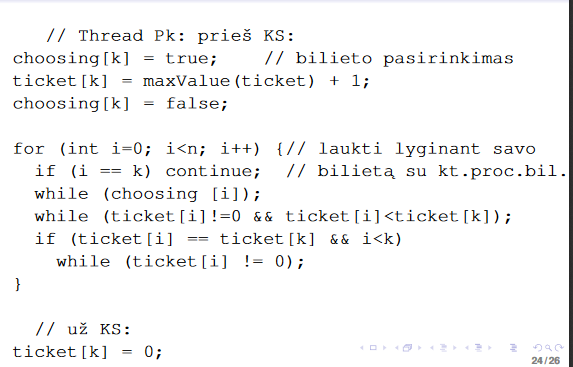
Petersono algoritmas tarpusavio išskyrimui leidžia dviem ar daugiau procesų dalintis vienkartinio panaudojimo resursus išvengiant konfliktų, bendrą atmintį naudoja tik komunikavimui. Naudojami 2 kintamieji *flag* ir *turn*. *Flag[n] true* reikšmė nurodo, kad *n* procesas nori patekti į kritinę sekciją. Įėjimas leidžiamas procesui *P0*, jei *P1* nenori tuo metu į ją patekti arba jei *P1* suteikė pirmenybę *P0* nustatydamas *turn* reikšmę į 0.

Algoritmas patenkina pagrindinius kriterijus kritinės sekcijos problemos išsprendimui, kadangi *turn,flag[0]* ir *flag[1]* reikšmės keičiasi iškart ir atomiškai. Trys kriterijai yra tarpusavio išskyrimas, progresavimas ir ribotas laukimas:

* *Tarpusavio išskyrimas*(tik vienas procesas gali būti KS vienu metu): *P0* ir *P1* niekada negali būti kritinėje sekcijoje tuo pačio momentu. Jei *P0* sekcijoje tai *flag[0]* yra *true*.
* *Progresas*(no deadlock): Jei nei vienas procesas nevykdo kritinės sekcijos ir yra norinčių į ją patekti, tuomet tik tie procesai, kurie tuo metu nevykdo likusių sekcijų gali dalyvauti sprendžiant, kuris procesas pateks į kritinę sekciją sekantis.
* *Apribotas laukimas*(bounded waiting, no livelock): Algoritme procesas niekada nelauks daugiau nei 1 kartą norėdamas patekti į KS.

1. **Bakery algoritmo analizė.**





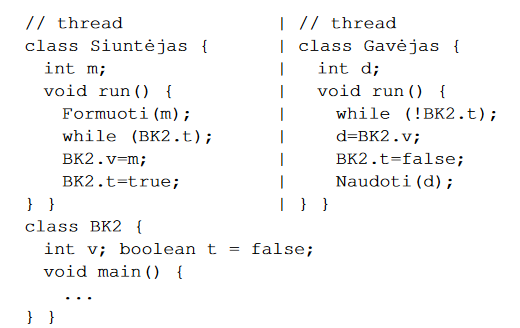
Lamporto kepyklos algoritmas yra skirtas apsaugoti bendrų resurų dąlinimąsi tarp kelių ar daugiau gijų, naudojant *tarpusavio išskyrimą*(mutual exclusion). Tai leidžia išvengti *duomenų sugadinimo*( data corruption), kuris gali įvykti, kuomet dvi ar daugiau gijų bando rašyti į tą pačią atminties vietą. Algoritmas sukurtas taip, kad būtų išvengiama kelių gijų patekimo į kritinė sekciją tuo pačiu metu ir taip sumažinant duomenų sugadinimo tikimybę.

Taip pat algoritme vyrauja tikimybė, kad kelios gijos gaus tuos pačius numerius, tad į pagalbą pasitelkiama ir gijos numeris, kuo mažesnis gijos numeris tuo greičiau ta gija bus aptarnauta.

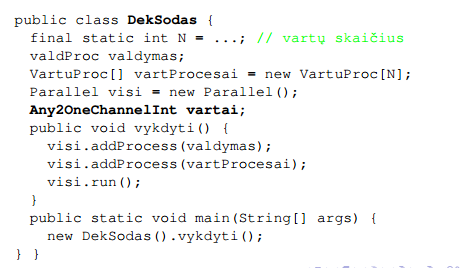
*Bakery koncepcija:*

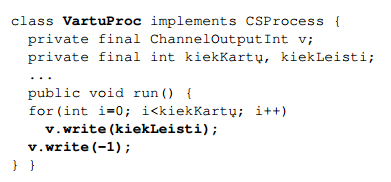
Galvokime apie populiarią parduotuvę su apkrautu klientų skaitikliu. Žmonės ima bilietą iš mašinos. Jei niekas nelaukia, bilietai neturi prasmės. Jei keli laukia, bilieto eiliškumas nusako, kieno eilė apsipirkti. Bakery algoritmas gerai tinka kur busy-waiting nėra itin neefektyvus. Pavyzdžiui, jei turima pakankamai CPU kad kiekviena gija turėtų savo. Algoritmas išprendžia *overflow* jeigu bilietas per didelis nustatant jį į 0 ir bandant iš naujo.

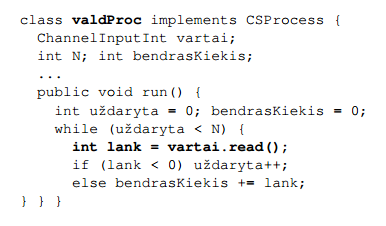
1. **Sąlyginės sinchronizacijos realizavimas nuoseklaus programavimo priemonėmis.**

**LP UŽDAVINIAI.**

1. **Dekoratyvinis sodas:**

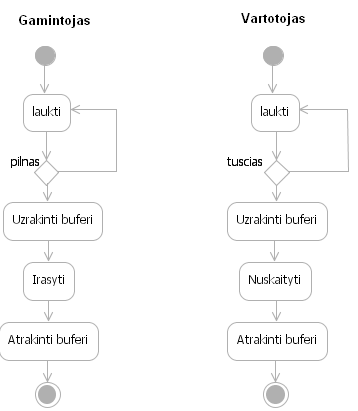


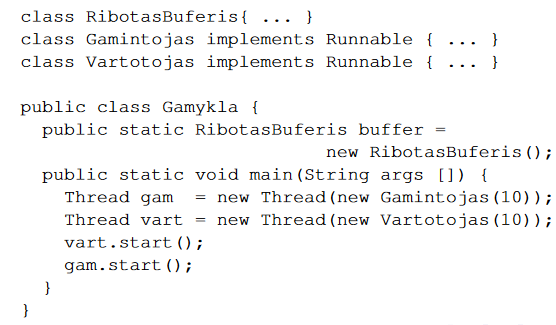


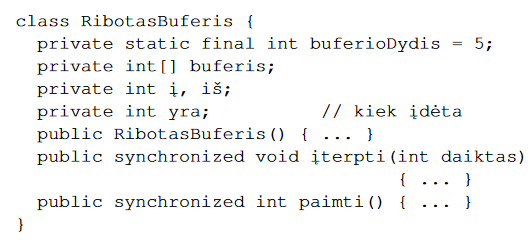


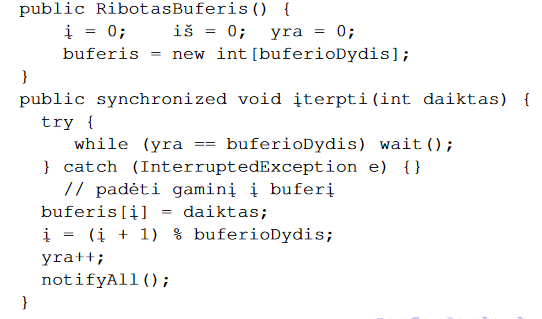
*Idėja:* sodas turi *n* vartų. pro kiekvienus vartus gali praeiti skirtingas žmonių skaičius, bet pačiam sode žmonių skaičius yra ribotas.

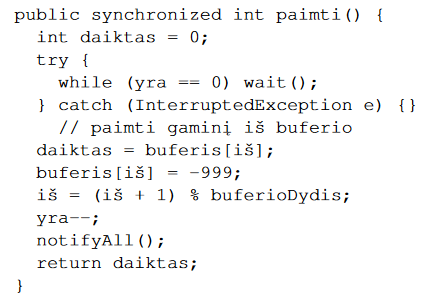
1. **Gamintojas – Vartotojas:**

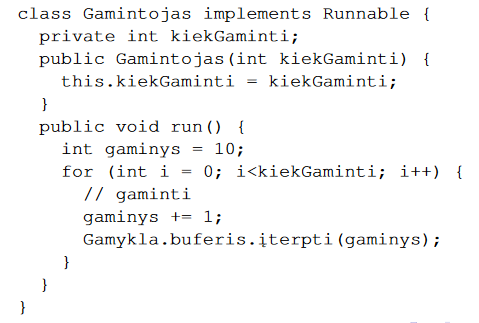


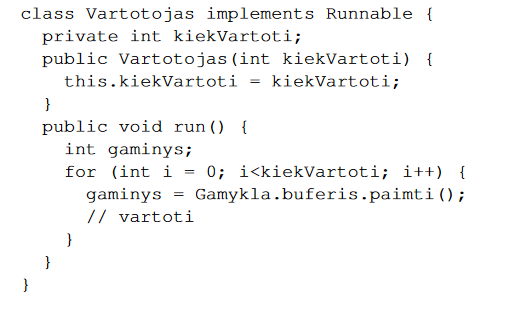












*Gamintojo-vartotojo problema* – tai dar viena problema, parodanti, jog sistemose, kurios naudojasi bendrais resursais būtina gijų sinchronizacija. Tarkime turime bendrai naudojamą fiksuoto dydžio buferį. Viena gija – gamintojas – rašo informaciją į buferį, kita – vartotojas – skaito informaciją iš buferio. Abi gijos veikia lygiagrečiai. Problema susidaro tuomet, kai gija bando įrašyti į pilną buferį arba nuskaityti iš tuščio. Taigi tam, kad nekiltų problemų naudojantis bendru buferiu, t.y. nesusidarytų kova dėl resursų, gijos turi būti teisingai sinchronizuotos. Gamintojas gali įrašinėti į buferį tik tuomet, kai jis nėra pilnas. Vartotojas gali skaityti iš buferio tada, kai jis nėra tuščias. Taigi tiek gamintojas tiek vartotojas gali laukti, kol šios sąlygos bus tenkinamos.

1. **Skaitantieji-Rašantieji:**

Readers/writers problem: – Allow multiple readers to concurrently access a data – Allow only one writer at a time • Two possible solutions using semaphores – Favor readers – Favor writers • Starvation is possible in either case!

Consider a situation where we have a file shared between many people. If one of the people tries editing the file, no other person should be reading or writing at the same time, otherwise changes will not be visible to him/her. However if some person is reading the file, then others may read it at the same time. Precisely in OS we call this situation as the **readers-writers problem.**

Problem parameters:

* One set of data is shared among a number of processes
* Once a writer is ready, it performs its write.
* Only one writer may write at a time.
* If a process is writing, no other process can read it.
* If at least one reader is reading, no other process can write.
* Readers may not write and only read.

**Solution when Reader has the Priority over Writer**

Here priority means, no reader should wait if the share is currently opened for reading. Three variables are used: **mutex, wrt, readcnt** to implement solution

* **semaphore** mutex, wrt; // semaphore **mutex** is used to ensure mutual exclusion when **readcnt** is updated i.e. when any reader enters or exit from the critical section and semaphore **wrt** is used by both readers and writers
* **int** readcnt;  //    **readcnt** tells the number of processes performing read in the critical section, initially 0

**Functions for sempahore :**

* wait() : decrements the semaphore value.
* signal() : increments the semaphore value.

**Writer process:**

Writer requests the entry to critical section. If allowed i.e. wait() gives a true value, it enters and performs the write. If not allowed, it keeps on waiting. It exits the critical section.

do {

// writer requests for critical section

wait(wrt);

// performs the write

// leaves the critical section

signal(wrt);

} while(true);

**Reader process:**

Reader requests the entry to critical section. If allowed: it increments the count of number of readers inside the critical section. If this reader is the first reader entering, it locks the **wrt** semaphore to restrict the entry of writers if any reader is inside. It then, signals mutex as any other reader is allowed to enter while others are already reading. After performing reading, it exits the critical section. When exiting, it checks if no more reader is inside, it signals the semaphore “wrt” as now, writer can enter the critical section. If not allowed, it keeps on waiting.

do {

// Reader wants to enter the critical section

wait(mutex);

// The number of readers has now increased by 1

readcnt++;

// there is atleast one reader in the critical section

**// this ensure no writer can enter if there is even one reader**

**// thus we give preference to readers here**

if (readcnt==1)

wait(wrt);

// other readers can enter while this current reader is inside

// the critical section

signal(mutex);

// current reader performs reading here

wait(mutex);   // a reader wants to leave

readcnt--;

// that is, no reader is left in the critical section,

if (readcnt == 0)

signal(wrt);         // writers can enter

signal(mutex); // reader leaves

} while(true);

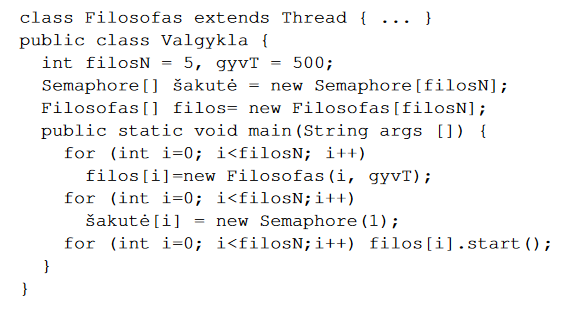
Thus, the mutex ‘**wrt**‘ is queued on both readers and writers in a manner such that preference is given to readers if writers are also there. Thus, no reader is waiting simply because a writer has requested to enter the critical section.

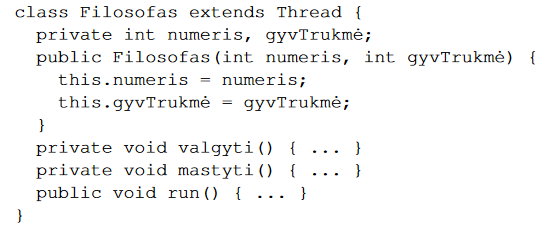
1. **Pietaujantys filosofai:**

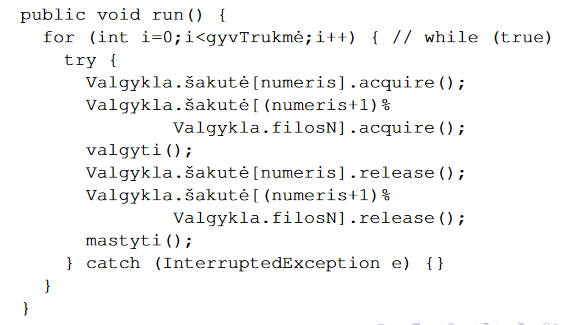
N filosofų sėdi prie apvalaus stalo. Tarp kiekvieno greta sėdinčio filosofo yra šakutė. Kitaip tariant yra N šakučių. Kiekvienas filosofas arba galvoja, arba valgo. Filosofas galvoja tol, kol išalksta. Tam, kad jis galėtų valgyti reikalingos dvi šakutės. Taigi išalkęs jis pasiima dvi šakutes ir pradeda valgyti. Kai filosofas pavalgo, jis padeda šakutes atgal ir ima vėl galvoti.

Esminis veiksmas yra „paimti šakutes“. Filosofai negali paimti tos pačios šakutės(resurso) tuo pat metu, kadangi susidarys kova dėl resursų. Be to filosofai negali laukti šakutės, kurios niekada negaus. Tam, kad išvengtume šių problemų tarkime, kad kiekviena šakutė yra bendrai naudojamas resursas, kuris turi būti apsaugomas užraktu. Tuomet filosofas gaus šakutę tik tuomet, jeigu ji bus laisvas, t.y. užraktas nebus užrakinti. Žemiau pateikiama veiklos diagrama.





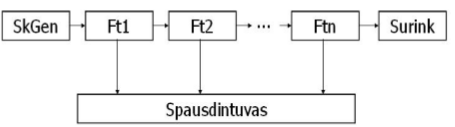




Kiekviena šakutė yra bendrai naudojamas resursas. Todėl gali susidaryti kova dėl resursų, t.y. jeigu filosofas bandys paimti šakutę vienu metu. Kadangi filosofai tarpusavyje nebendrauja, gali susidaryti aklavietė, pavyzdžiui jeigu visi filosofai išalks vienu metu pasiims po šakutę ir lauks antros. Taip pat gali susidaryti badavimas: jeigu visą laiką valgys vienas prieš kitą sėdintys filosofai, o kiti trys nespės paimti šakučių, tuomet jie niekuomet nevalgys. Filosofų atliekami veiksmai yra ciklikliški. Filosofas paima šakutes, pavalgo, padeda šakutes, masto ir vėl kartoja tuos pačius veiksmus.

**LYGIAGRETIEJI ALGORITMAI. Java nuoseklių procesų komunikavimas -2**

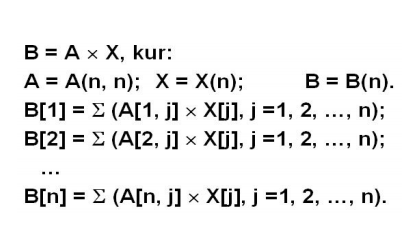
1. **Pirminių skaičių generavimas:**



Vienas iš pirminių skaičių generavimo algoritmų yra naudojant Eratosteno rėtį. Tai yra senai žinomas algoritmas pirminių skaičių radimui. Jo esmė yra išrašyti skaičius nuo 2 iki pasirinkto n. Tada pašaliname 2 kartotinius, sekantis skaičius 3 yra pirminis. Tuomet šaliname 3 kartotinius, tuomet 5 yra pirminis , šaliname 5 kartotinius ir t.t. Algoritmą galime realizuoti naudojant CSP kanalus.

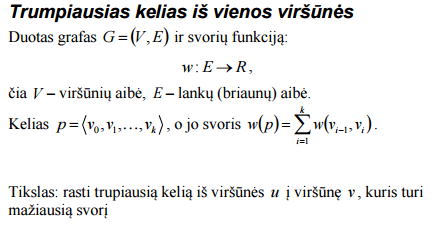
Pradedame sukurdami skaičių generatoriaus procesą, jo funkcija yra sugeneruoti skaičių x , kuris yra nuo 2 iki n ir jį išsiųsti. Kiekvienam gautam skaičiui x sukuriame filtruojanti procesą, jo paskirtis yra perduoti pirminį skaičių x spausdinančiam procesui ir pašalinti visus x skaičiaus kartotinius, tuomet perduoti likusias reikšmes toliau. Filtravimas vykdomas kol gaunami visi skaičiai.

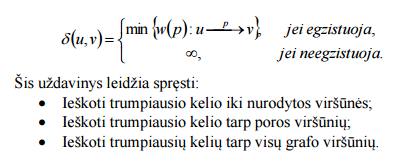
1. **Matricos ir vektroriaus daugyba:**

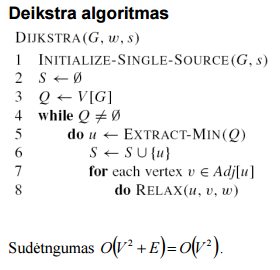


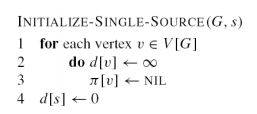
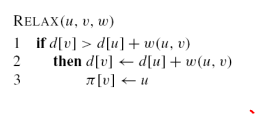
Atskirai kiekvieną matricos eilutę sudauginam su vektoriumi. Gaunamos dalinės sumos, kurias po to sudedame.

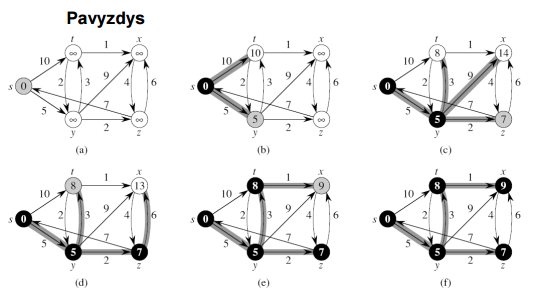
1. **Matricų daugyba, Tiesinių lygčių sistemos sprendimas, Masyvų tvarkymas, Grafo trumpiausių kelių radimas ir kt:**



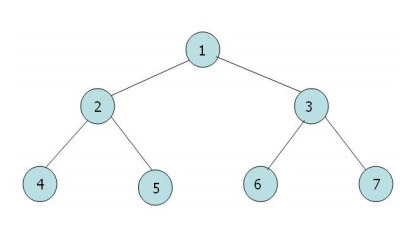


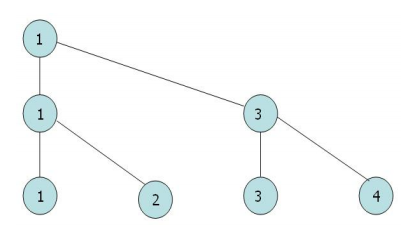




1. **Masyvų tvarkymas:**





The individual processes will sort the sub set of the parent array. But after the Master process gathers all these sub sets there has to be one sorting done for the parent array.

e.g.

original array = {1,7,2,3, 10,4,8,0, 11,5,9,6}

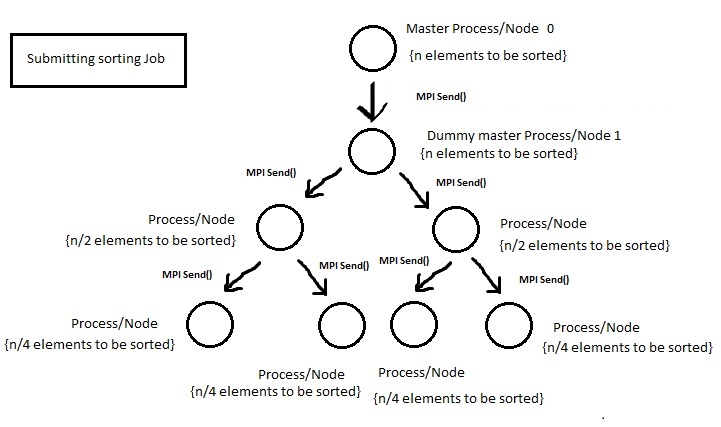
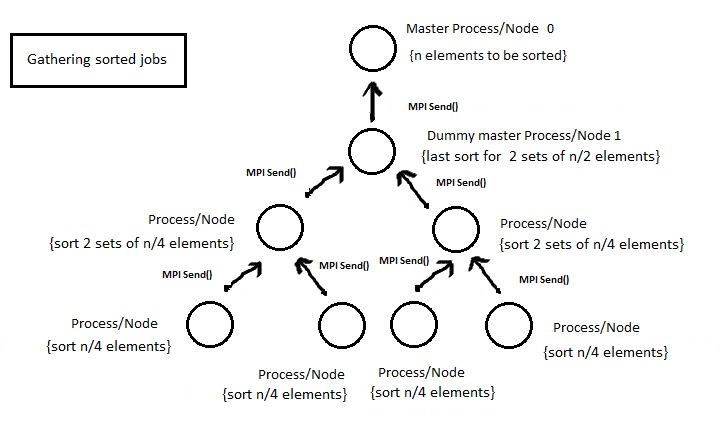
after scatter process 1 : {1,7,2,3} process 2 : {10,4,8,0} process 3 : {11,5,9,6}

and individual process sorting : {1,2,3,7} , {0,4,8,10} , { 5,6,9,11}

thus after gathering you have the original array as {1,2,3,7 , 0,4,8,10 , 5,6,9,11}

Thus there has to be one more sorting pass done.

Edit :

One of the solutions would be (This can be further optimized): instead of using mpi scatter and gather use a plain mpi send and mpi receive.The master process/node would give the data to the dummy master process/node which will further divide it to rest of the nodes. The last line of nodes can sort the sub set of data and return the sorted subsets to their parents.   
after the parents receive the individually sorted sub sets they will sort the sorted subsets and provide their sets to their parents.

This approach can be further optimized.

**BENDRA ATMINTIS. MONITORIAI.**

1. **Monitorių paskirtis ir realizavimas:**

* Monitorius -programos kodo dalis, kuri visada vykdoma su tarpusavio išskyrimu
* Monitorių sudaro: bendrieji saugomi duomenys; atominių veiksmų, skirtų saugomiems duomenims apdoroti, rinkinys; sąlyginių kintamųjų rinkinys.
* Jis užtikrina, jog tik vienai gijai leidžiama vykdyti monitoriaus kodą vienu metu.

1. **Kokios yra monitorių naudojimo problemos?**

* Visų procesų atominiai veiksmai yra vienoje vietoje
* Monitoriaus viduje gali būti kreipiniai į kitus monitorius
* Metodas vykdomas vieno monitoriaus viduje gali kviesti kito monitoriaus metodą
* Metodas, vykdantis *signal* yra monitoriuje, o procesas išblokuojantis *signal*, turi grįžti į monitorių

1. **Java monitorių realizavimo ypatybės**

* Kiekvienas Java objektas turi su juo susieta monitorių
* Kiekviena Java klasė turi su ja susieta monitorių
* Monitorius nerealizuojamas jei nenaudojama sinchronizuoti metodai
* Tarpusavio išskyrimas- užrakinant objektą su *synchronized*( realizuoja JVM)
* Sąlyginė sinchronizacija- naudojant *Object klasės wait, notify, notifyall metodus*
* Jeigu monitoriaus saugomas kodas yra užimtas, tuomet java gija pereina į laukimo būseną, kol jis bus atlaisvintas. Yra du būdai objekto užrakinimui[4].:

1. Kviečiant sinchronizuotą bloką.

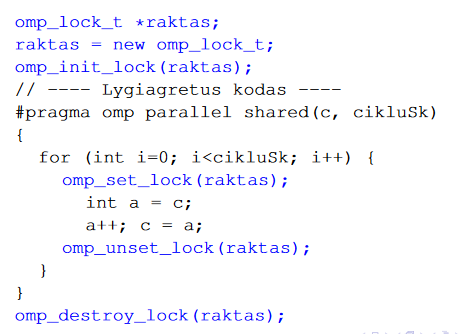
synchronized (anObject) { ... }Čia užrakinamas obejktas *anObject*. Užraktas atlaisvinamas tuomet, kai baigiamas kodo vykdymas sinchronizuotame bloke. Jeigu kita gija jau vykdo kodą sinchronizuotame bloke, tuomet gija yra blokuojama kol blokas atsilaisvins.

1. Metodo sinchronizavimas

public synchronized void aMethod() { ... } Metodo sinchronizavimas, tai tas pats kad objekto *this* užrakinimas sinchronizuotame bloke.

1. **OpenMp monitorių realizavimo ypatybės**

* Galima realizuoti naudojant *#pragma omp critical* sekcija. Kai panaudojam ši sekcija, jos kodas vykdomas tik vienos gijos
* Galim realizuoti naudojant *omp\_lock\_t:*



1. **Kokiu būdo įgyvendinama sąlyginė sinchronizacija OpenMP monitoriuje?**

* Panaudojant *#pragma omp barrier -* Gijos po šio metodo paskelbimo laukia kol visos gijos užbaigs vykdyti savo kodą ir tik kai visos užbaigia toliau vykdo programos kodą lygiagrečiai
* Naudojant *#pragma omp master -* Sinchronizacija realizuojama, kai tik pagrindinė gija vykdo tam tikrą gabalą kodo, o kitų gijų vykdymas yra stabdomas, kitos gijos vykdo kodą tik kai pagrindinė gija baigia master sekcijos vykdymą

1. **Kuo ypatingi OpenMP bendrieji kintamieji?**

* Kompiuterio atmintyje yra tik viena bendrojo kintamojo kopija. Kintamasis matomas kiekvienoje gijų rinkinio gijoje
* Vienoje gijoje pakeista bendro kintamojo reikšmė gali būti matoma kitoje gijoje
* Jei nurodyta kitaip, visi programos kintamieji yra bendri visoms lygiagrečios srities gijoms

1. **Kuo ypatingi OpenMP privatūs kintamieji?**

* Privatus kintamasis turi savo kopijas kiekvienoje gijoje. Kiekviena kopia matoma tik vienoje gijoje
* Vienoje gijoje pakeista privataus kintamojo reikšmė nematoma kitose gijos
* Kintamieji yra privatūs, jei:
  + Lygiagrečiojo (for) ciklo indeksas yra privatus
  + Lygiagrečios srities bloke paskelbti lokalūs kintamieji yra privatūs
  + Visi kintamieji išvardinti *pragma omp* direktyvoje kaip *private, firstprivate, lastprivate* arba *reduction* yra privatūs

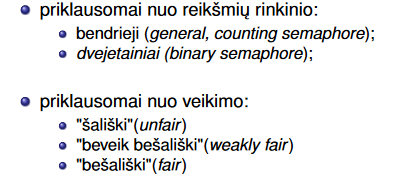
**BENDRA ATMINTIS. SEMAFORAI IR UŽRAKTAI.**

1. **Semaforų paskirtis ir tipai**

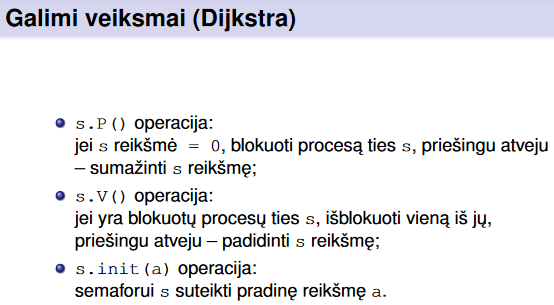
Kiekviena gija, prieš tai darydama, turi gauti iš semaforo leidimą (kviesdama semaforo leidimo pasiėmimo metodą), o baigusi darbą su saugomu objektu – leidimą grąžinti (kviesdama leidimo grąžinimo metodą). Jei semafore daugiau šiuo metu leidimų nebėra (jo likusių leidimų skaitiklis lygus nuliui), leidimo prašanti gija blokuojama kol kokia nors kita gija anksčiau pasiimtą leidimą grąžins. Jei semaforas turi tik vieną leidimą, jo darbas daug nesiskiria nuo užrakto. Tačiau, skirtingai nuo užraktų, semaforai paprastai leidžia grąžinti leidimą ir ne tai gijai, kuri jį pasiėmė. Tai gali būti reikalinga, pavyzdžiui, išsprendžiant kai kuriuos sinchronizacijos aklavietės atvejus. Tačiau šiai galimybei reikalingi sinchronizavimo veiksmai gali sąlygoti lėtesnį darbą nei naudojant užraktus. Skirtingai nuo užraktų, semaforai neskirti pakartotinam rakinimui ta pačia gija bei atskiriems rakinimams skaitymui bei rašymui. Kaip ir užraktas, semaforas gali būti garbingas (angl. *fair*) arba ne. Garbingas semaforas pirmiausia suteikia leidimą tai gijai, kuri jo laukė ilgiausiai. Garbingo semaforo kodas yra sudėtingesnis ir vykdomas ilgiau, todėl naudoti jį verta ne visada.

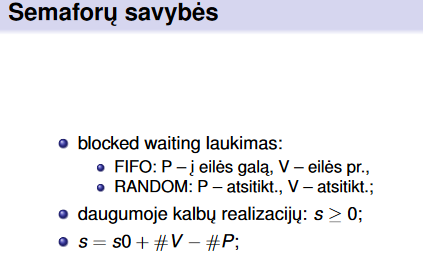
struktūra, turinti ją sukuriant nustatytą leidimų skaičių vykdymo gijoms dirbti su saugoma duomenų struktūra ar vykdyti saugomą kodo sekciją.

Semaforai naudojami, kuomet pageidaujama, jog su tam tikra duomenų struktūra ar kompiuterio įrenginiu nedirbtų daugiau nei nurodytas ribotas skaičius lygiagrečių gijų.

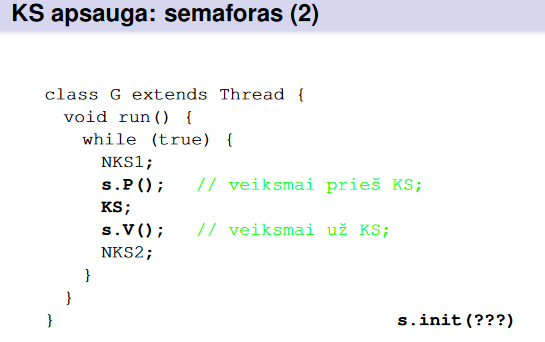


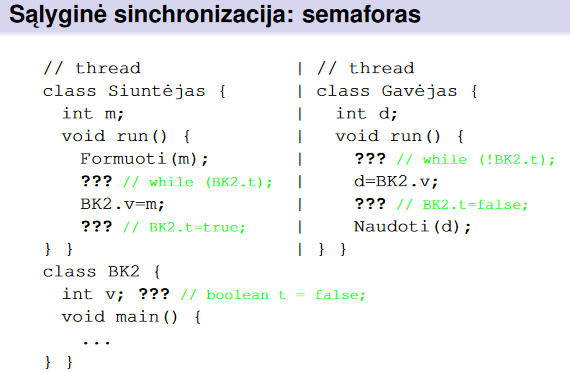
1. **Semaforų savybės, galimi veiksmai**



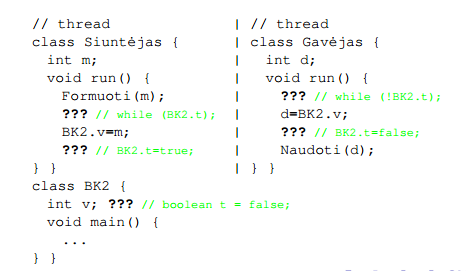


1. **Semaforų taikymas KS apsaugai ir sąlyginei sinchronizacijai realizuoti**

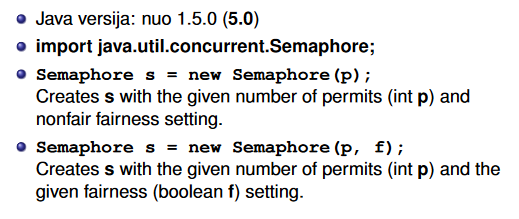


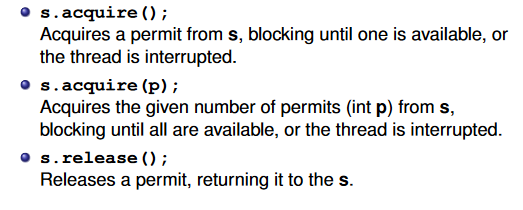


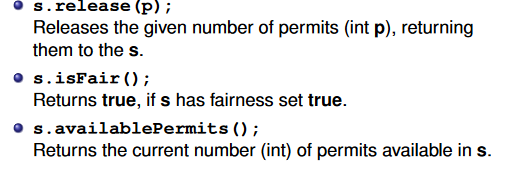
1. **Semaforų naudojimas LP uždaviniams spręsti.**

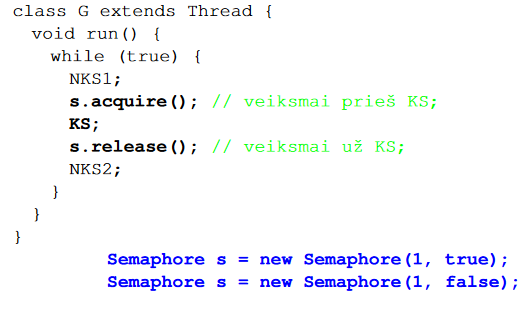


1. **Semaforų realizavimo Java ypatybės**









**ŽINUČIŲ PERDAVIMAS KANALAIS.**

1. **Žinučių perdavimo modeliai ir galimi variantai**

* Modeliai:
  + Sinchroninis siuntimas
  + Asinchroninis siuntimas
* Galimi variantai
  + Vienas vienam
  + Vienas keliems
  + Keli vienam
  + Keli keliems

1. **Kanalai: tipai, savybės, galimi veiksmai:**

CSP procese tiek duomenys, tiek jų apdorojimo metodai yra privatus.

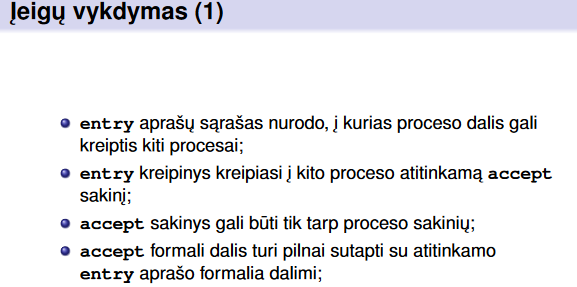
* **"many-to-one"** Komunikavimas "many-to-one": galima supaprastinti programos koda˛.
* public class Channel
  + public static Any2OneChannel any2one()
  + public static Any2OneChannelInt any2oneInt()
* public interface Any2OneChannel
  + AltingChannelInput in()
  + ChannelOutput out()
* public interface Any2OneChannelInt
  + AltingChannelInputInt in()
  + ChannelOutputInt out()
* **"point-to-point" -** Komunikavimas "**one-to-one**": galima sužinoti, kas siunte.
* public interface ChannelOutput
  + void write(Object objektas)
* public interface ChannelOutputInt
  + void write(int i)
* public interface ChannelInput
  + Object read()
* public interface ChannelInputInt
  + int read()
* public class Channel
  + public static One2OneChannel one2one()
  + public static One2OneChannelInt one2oneInt()
* public interface One2OneChannel
  + AltingChannelInput in()
  + ChannelOutput out()
* public interface One2OneChannelInt
  + AltingChannelInputInt in()
  + ChannelOutputInt out()

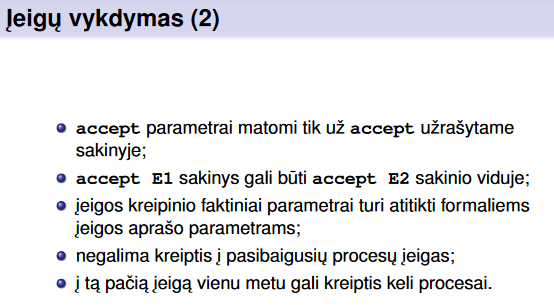
1. **Selektyvus laukimas, alternatyvos ir saugai:**

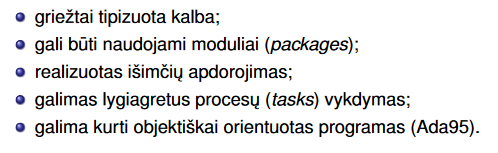
* public abstract class Guard
  + Guard();
* public class CSTimer extends Guard
  + public void after(long msecs);
  + public long read();
  + public void sleep(long msecs);
  + public void setAlarm(long msecs);
  + public long getAlarm();
* public class Skip extends Guard
  + public void run();
* public class Stop extends Guard
  + public void run();
  + public abstract class AltingChannelInput extends Guard
    - public abstract boolean pending();
  + public abstract class AltingChannelInputInt extends Guard
    - public abstract boolean pending();
  + public class Alternative
    - Alternative(Guard[] guard);
    - int select();
    - int fairSelect();
    - int priSelect();
    - int select(boolean[] preCondition);
    - int fairSelect(boolean[] preCondition);
    - int priSelect(boolean[] preCondition);

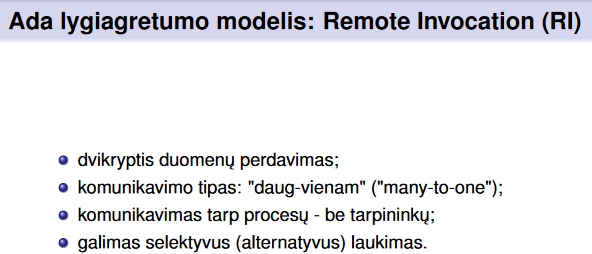
**ADA KALBOS ŽINUČIŲ PERDAVIMO MODELIS.**

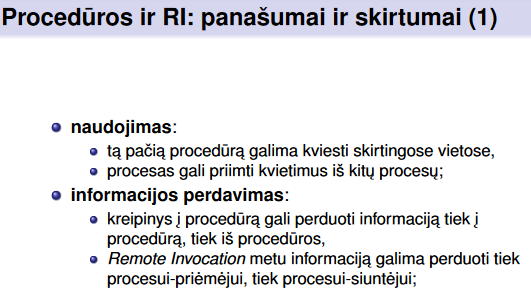
1. **Modelio realizavimas ir ypatybės:**
   * Remote Invocation (RI)
   * dvikryptis duomenu˛ perdavimas;
   * komunikavimo tipas: "daug-vienam" ("many-to-one");
   * komunikavimas tarp procesu˛ - be tarpininku˛;
   * galimas selektyvus (alternatyvus) laukimas.
2. **Įeigos ir jų naudojimas; skirtumai tarp įeigų ir procedūrų**
   * naudojimas:
     + ta˛pacia˛procedura˛galima kviesti skirtingose vietose,
     + procesas gali priimti kvietimus iš kitu˛ procesu˛;
   * informacijos perdavimas:
     + kreipinys˛i procedura˛gali perduoti informacija˛tiek˛i procedura˛, tiek iš proceduros,
     + Remote Invocation metu informacija˛galima perduoti tiek procesui-priemejui, tiek procesui-siuntejui;
   * vykdymo salygos:
     + proceduros veiksmai vykdomi visada, kai tik aptinkamas kreipinys, RI kvietimas vykdomas tik tada, kai procesas-savininkas tai leidžia;
   * vykdytojas:
     + proceduros veiksmai atliekami jos kvietejo vardu,
     + kvietimo metu nurodytus veiksmus vykdo procesas-savininkas
3. **Selektyvus laukimas ir įeigos:**

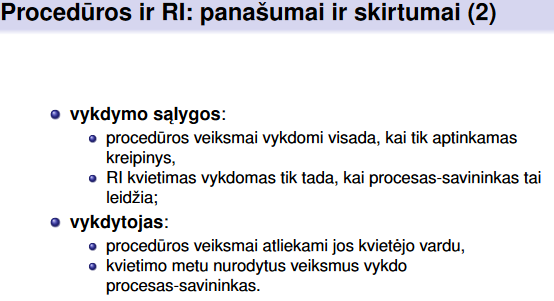


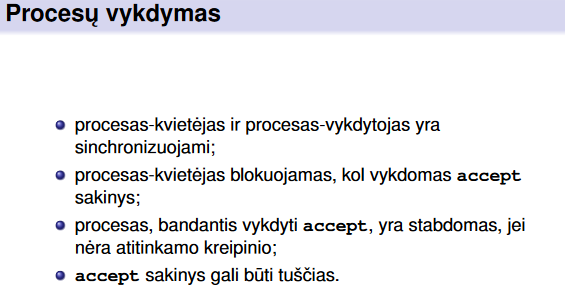












1. **Ada modelio taikymas LP uždaviniams spręsti.**
   * procesas-kvietėjas ir procesas-vykdytojas yra sinchronizuojami;
   * procesas-kvietėjas blokuojamas, kol vykdomas accept sakinys;
   * procesas, bandantis vykdyti accept, yra stabdomas, jei nėra atitinkamo kreipinio; accept sakinys gali būti tuščias.

**PROGRAMAVIMAS, NAUDOJANT OPENMP BIBLIOTEKĄ.**

1. **OpenMP paskirtis, ypatybės.**
   * OpenMP – tai papildomos priemonės, suteikiančios galimybes kurti lygiagrečias programas Fortran, C, C++ kalbomis;
   * OpenMP skirta kurti lygiagrečias programas, kuriose procesai (gijos) naudojasi bendra atmintimi;
   * OpenMP sudaro: kompiliatoriaus direktyvos, specialių funkciju˛ biblioteka, aplinkos kintamųju˛ rinkinys.
2. **Pagrindinės OpenMP direktyvos ir funkcijos.**
   * Nuo programos darbo pradžios iki pabaigos vykdoma pagrindine gija (master thread).
   * Naujos gijos kuriamos kompiliatoriaus direktyvomis.
   * Vienu metu vykdomu˛ giju˛ rinkinys sudaro lygiagrecia srit (parallel region).
   * Lygiagrečioje srityje vykdomu˛ giju˛ skaičiu˛ galima nurodyti programos vykdymo metu.
   * Visos lygiagrečios srities gijos vykdo tą patį struktūrinį bloka˛.
   * Visos lygiagrečios srities gijos pradedamos vykdyti tuo pačiu metu ir vykdomos lygiagrečiai.
   * Lygiagreti sritis baigiama vykdyti, kai baigiamos vykdyti visos tos srities gijos. Kiekvienai lygiagrečios srities gijai suteikiamas unikalus numeris; master gijos numeris – 0.
   * Programos vykdymo metu nėra galimybiu˛ keisti gijos numerį.
   * Lygiagrecioje srityje vykdomu˛ giju˛ skaicius nurodomas pragma omp parallel atributu num\_threads() arba prieš lyg. sriti funkcijos omp\_set\_num\_threads() parametru.
   * Vykdomu˛ giju˛ skaiciu˛ gražina funkcija omp\_get\_num\_threads().
   * Gijos numeri gražina funkcija omp\_get\_thread\_num()
3. **Pragramavimo naudojant OpenMP priemonių pavyzdžiai:**

int main(){

int gijuSk = 2, gijosNr = 0;

omp\_set\_num\_threads(gijuSk);

// ------ Lygiagretus kodas -------

#pragma omp parallel

private(gijosNr) {

gijosNr = omp\_get\_thread\_num();

if (gijosNr == 0) vykdyti("Pirma");

else vykdyti("Antra"); }

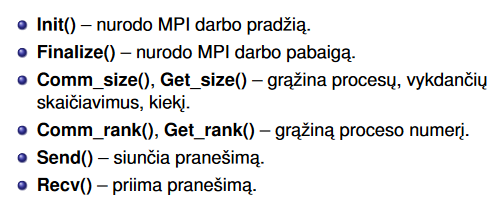
cout << " Programa baig˙e darba˛";

**PROGRAMAVIMAS, NAUDOJANT MPI BIBLIOTEKĄ.**

1. **Kokia yra pagrindinė MPI paskirtis?**

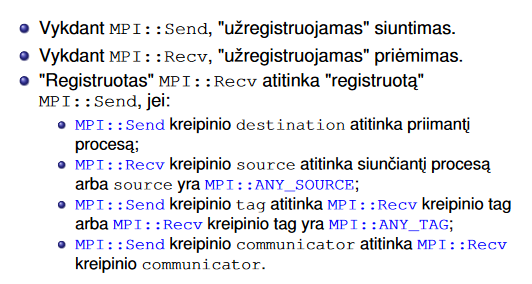
Tai pranešimų perdavimo funkcijų, skirtų realizuoti lygiagrečiuosius algoritmus, standartas

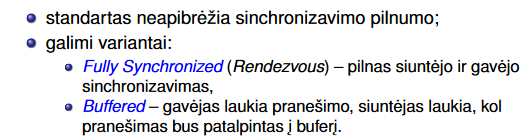
1. **Kokios pagrindinės MPI funkcijos?**

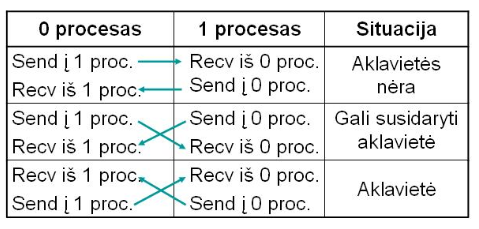


1. **MPI pranešimų perdavimo variantai ir ypatybės**

* Send/Recv:



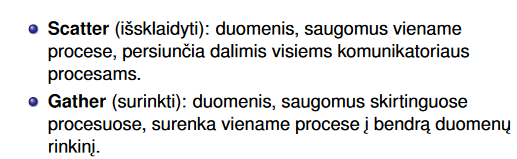




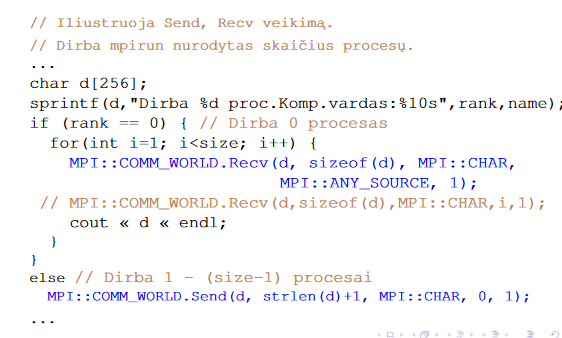
* Bcast:

Tai siuntimas visiems

* Scatter ir Gather:

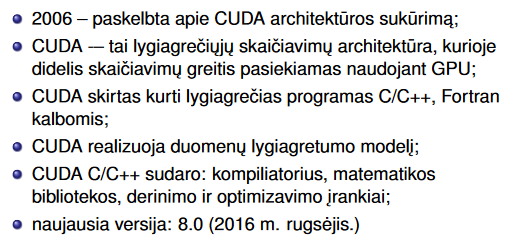


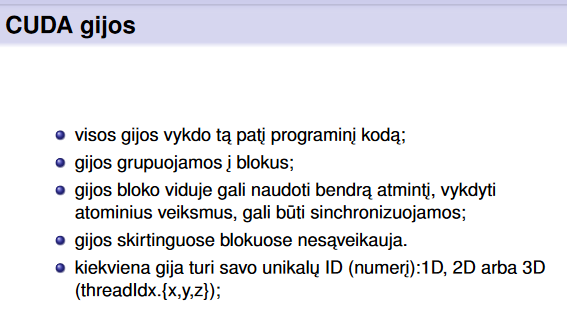
1. **Programavimo naudojant MPI priemones pavyzdys**



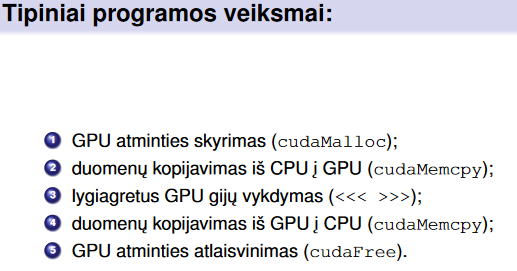
**CUDA LYGIAGRETUMO MODELIS.**

1. **CUDA paskirtis, ypatybės:**



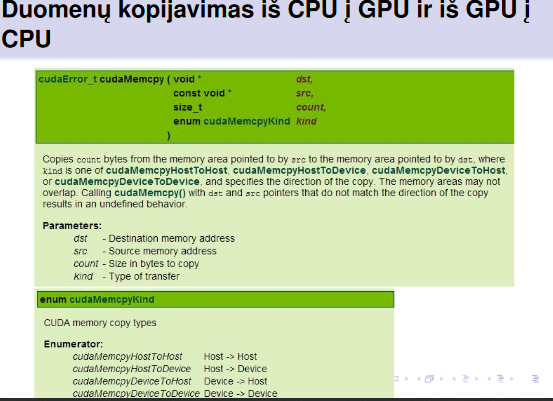


1. **Atminties modelis, skyrimas, atsisakymas:**

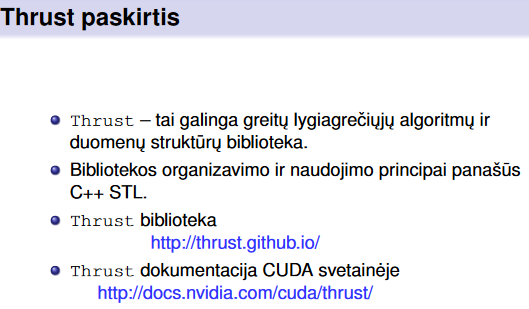


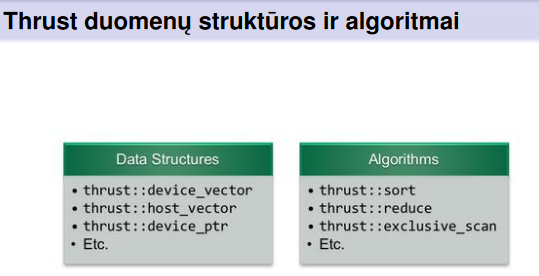


1. **Duomenų mainai GPU/CPU:**



1. **Gijų kūrimas ir vykdymas:**
2. **Thrust vektoriai:**





**JAVA KALBŲ YPATYBĖS (PROCESŲ KŪRIMAS IR KOMUNIMAVIMAS).**

1. **Java,** 
   * Panaudojant (paveldint) java.lang.Thread klase˛.
   * Panaudojant (realizuojant) java.lang.Runnable sasaja˛.
   * notify() Wakes up a single thread that is waiting on this object’s monitor.
   * notifyAll() Wakes up all threads that are waiting on this object’s monitor.
   * wait() Causes current thread to wait until another thread invokes the
   * notify() method or the notifyAll() method for this object. wait(long timeout) Waits for notiﬁcation or until the timeout period has elapsed. timeout is measured in milliseconds.
2. **C++,** 
   * konstruktorius su parametrais: thread(fn, args), kur fn – gijoje vykdoma funkcija, args – funkcijai perduodamu˛ argumentu˛ sa˛rašas;
   * gijos darbo pabaigos laukimo metodas: join()
3. **C#;**
   * using System.Threading;
   * Thread InstanceCaller = new Thread(

new ThreadStart(serverObject.InstanceMethod));

InstanceCaller.Start();

sleepingThread.Start();

Thread.Sleep(2000);

* Thread.Interrupt wakes a thread out of any wait it might be in and causes a ThreadInterruptedException to be thrown in the destination thread.
* Thread.Abort wakes a thread out of any wait it might be in and causes a ThreadAbortException to be thrown on the thread. For details, see Destroying Threads.

1. **Ada;**

* procesas-kvietejas ir procesas-vykdytojas yra sinchronizuojami;
* procesas-kvietejas blokuojamas, kol vykdomas accept sakinys;
* procesas, bandantis vykdyti accept, yra stabdomas, jei nera atitinkamo kreipinio; accept sakinys gali buti tušcias.

1. **occam;**

* Palaiko point to point komunikavimą
* Visas komunikavimas vykdomas per kanalus
* Komunikavimas yra sinchronizuotas
* Palaiko CSP
* Naudoja Saugus(guards) Saugai
* Lygiagretuma apraso sakinys PAR

SEQ

PAR

...knit body .

..knit left sleeve

...knit right sleeve

...knit neck .

..sew sweater

ALT

count1 < 100 & c1 ? data

SEQ

count1 := count1 + 1

merged ! data

count2 < 100 & c2 ? data

SEQ

count2 := count2 + 1

merged ! data

status ? request

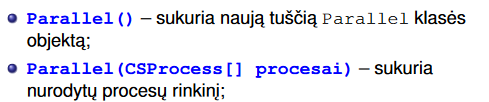
SEQ

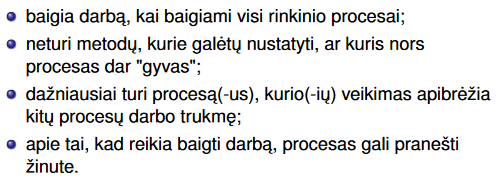
out ! count1

out ! count2

1. **Google Go.**
   * go sakiniuose kviečiamos funkcijos vykdomos kaip atskiros gijos, t.y., lygiagrečiai (concurrent)
   * nera klasių, taciau kiekvienam tipui galima kurti metodus (funkcijas);
   * komunikavimui tarp goroutines taikomas Hoare CSP (occam, JavaCSP) modelis; komunikavimui naudojami tipizuoti kanalai:
     + nebuferizuotas: ch := make(chan Tipas)
     + buferizuotas: ch := make(chan Tipas, n)
     + rašymas˛i kanala˛(dvinare operacija): ch <- a
     + skaitymas iš kanalo (vienanare operacija): <- ch
   * sinchronizavimui naudojami nebuferizuoti (sinchroniniai) kanalai.

**Java procesai: kūrimas ir inicializavimas**





**Kuo ypatinga tvarkos abstrakcija?** Užrašymo tvarka nenurodo vykdymo tvarkos

**Kuo pasižymi funkcinis lygiagretumas?** Lygiagrečiai vykdomi skirtingi veiksmai

**Kuo pasižymi duomenų lygiagretumas?** Lygiagrečiai vykdomi tie patys veiksmai