U N I V E R Z A N A P R I M O R S K E M
Fakulteta za matematiko, naravoslovje in informacijske tehnologije

Janez Žibert

SKRIPTA ZA PREDMET

PROGRAMIRANJE 3: VZPOREDNO PROGRAMIRANJE

NA ŠTUDIJSKEM PROGRAMU
RAČUNALNIŠTVA IN INFORMATIKE NA 1. STOPNJI
UP FAMNIT

PRVA IZDAJA

Koper, 2012

Predgovor

Ta skripta so nastala iz prosojnic, ki sem jih uporabljal na predavanjih pri predmetu *Programiranje 3- Vzporedno programiranje* na 1. stopnji študijske smeri Računalništva in informatike na Fakulteti za matematiko, naravoslovje in informacijske tehnologije na Univerzi na Primorskem v letih od 2008 do 2012.

Skripta predstavljajo osnovne koncepte vzporednega in porazdeljenega programiranja, kjer se ukvarjamo predvsem z mehanizmi sinhronizacije in komunikacije med sočasno tekočimi procesi. Predstavljeni so osnovni koncepti sinhronizacije s pomočjo ključavnic, pregrad, semaforjev in monitorjev. Pri porazdeljenem programiranju pa obdelamo komunikacijo med procesi s pomočjo sprejemanja in pošiljanja sporočil. V dodatkih so predstavljena tudi standardna orodja in programski jeziki, ki jih uporabljamo za vzporedno in porazdeljeno programiranje: knjižnica PTHREADS, večnitno programiranje v JAVI in porazdeljeno programiranje v okolju MPI.

Skripta so nastala po knjigi:

- G. R. Andrews:
  Foundations of Multithreaded, Parallel, and Distributed Programming, Addison-Wesley, 2000.

Ljubljana, september 2012 Janez Žibert

Kazalo

1 Osnovni pojmi in koncepti vzporednega in porazdeljenega programiranja

V tem poglavju spoznamo nekaj osnovnih konceptov vzporednega in porazdeljenega programiranja. Seznanimo se z osnovnimi računalniškimi arhitekturami, s katerimi lahko izvajamo vzporedne in porazdeljene programe. Definiramo osnovne pojme vzporednega programiranja, kot so opravilo, proces, stanje procesa, sprememba stanja procesa itd., spoznamo se z dvema konceptoma komunikacije med sočasno tekočimi procesi in predstavimo 5 osnovnih konceptov delovanja vzporednih in porazdeljenih programov.

Poglavje predstavi:

- računalniške arhitekture za izvajanje vzporednih programov,
- osnovne pojme pri vzporednem programiranju
- principe medporcesne komunikacije sočasno delujočih procesov,
- koncepte vzporednega/porazdeljenega programiranja:
  - iterativni paralelizem,
  - rekurzivni paralelizem,
  - princip proizvajalec/porabnik,
  - princip odjemalec/strežnik,
  - princip interakcije enakovrednih procesov.

Sekvenčni in vzporedni program

- Sekvenčni program:
  - predstavlja zaporedje ukazov – operacij, ki se izvajajo ena za drugo z namenom reševanja (izvajanja) določene naloge,
  - program v izvajanju imenujemo proces.

- Vzporedni program:
  - vključuje dva ali več procesov, ki sodelujejo pri reševanju določene naloge:
    - vsak proces izvaja svoje zaporedje ukazov,
    - procesi sodelujejo/komunicirajo med seboj s pomočjo skupnega pomnilnika ali s pomočjo sporočil
  - izvajanje:
    - sočasno na eno-procesorskih sistemih,
    - vzporedno na večprocesorskih ali na večračunalniških sistemih.

Oblike programov glede na izvajanje procesov

- Aplikacije sočasnega računanja
  - procesi se izvajajo sočasno na enem procesorju (oziroma več procesov na en procesor)
  - komunikacija poteka s pomočjo branja in pisanja v skupni (deljeni) pomnilnik
  - aplikacije:
    - moderni operacijski sistemi, grafični vmesniki
    - realno-časovni sistemi

- Aplikacije porazdeljenega računanja
  - procesi se izvajajo na več računalnikih, ki so povezani v omrežje,
  - komunikacija poteka s pomočjo izmenjave sporočil (s sprejemanjem in oddajanjem sporočil),
  - aplikacije:
    - povsod, kjer je zahteva po porazdeljenih resursih:
      - datotečni in podatkovni strežniki, www strežniki, p2p, iptv, …

- Aplikacije vzporednega računanja
  - procesi se izvajajo vzporedno na več procesorjih (toliko procesov kot je procesorjev)
  - komunikacija: s pomočjo skupnega pomnilnika ali s sporočili
  - aplikacije:
    - pohitritev postopkov za obdelavo večjih količin podatkov:
      - modeliranje: optimizacije, statistično modeliranje, obdelava slik in videa (grafične kartice)

Zakaj vzporedni programi?

- Glavne motivacije:
  - pohitritev postopkov (zaradi porazdelitve dela med več procesorjev, računalnikov)
  - zahteve po porazdeljenemu delovanju posameznih storitev (odjemalec/strežnik, p2p),
  - “naraven” pristop pri reševanju določenih problemov (večopravilnost v operacijskih sistemih, grafični vmesniki, …),
  - povečanje skalabilnosti posameznih programov/aplikacij

Učinkovitost vzporednih programov

- Pri vzporednem programiranju iščemo takšne algoritme, ki nas pripeljejo do želenega cilja prej kot strogo zaporedno reševanje problema.

- Merjenje pohitritve:
  - dejanska pohitritev:

$$S(N) = \frac{\text{sekvenčni algoritem na enem procesorju paral. računalnika}}{\text{vzporedni algoritem na N procesorjih paral. računalnika}}$$

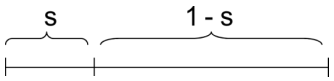
  - relativna pohitritev:

$$S(N) = \frac{\text{vzporedni algoritem na enem procesorju paral. računalnika}}{\text{vzporedni algoritem na N procesorjih paral. računalnika}}$$

Dejavniki, ki omejujejo pohitritev

- stroški dekompozicije problema
  - prirastek števila ukazov – zvečanje programskega bremena (dodatne operacije, nove spremenljivke, …),
  - neenakomerna porazdelitev bremena (neenakomerna porazdelitev podatkov, neuravnotežena zahtevnost podoperacij),
  - porast komunikacijskega bremena (sinhronizacija)
- sekvenčna narava problema:
  - Amdahlov zakon:

    s – stopnja sekvenčnosti problema, odstotek časa, ki ga potrebujemo za reševanje sekvenčnega dela programa.



$$S(n) = \frac{T}{\frac{T(1-s)}{n} + T \cdot s}$$

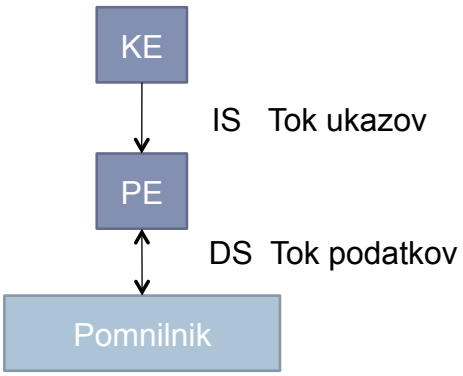
Potem je pohitritev problema navzgor omejena z 1/s.

$$\lim_{n\to\infty} \frac{T}{\frac{T(1-s)}{n} + T \cdot s} = \frac{1}{s}$$

Računalniški modeli za vzporedno programiranje

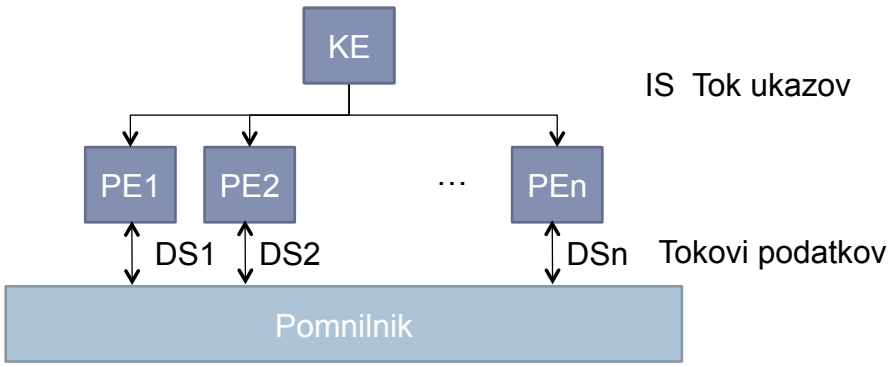
- Računalnik izvršuje ukaze nad podatki:
  - zaporedje ukazov (tok ukazov) določa operacije, ki jih računalnik izvršuje eno za drugo
  - zaporedje podatkov (tok podatkov) določa nad čim se operacije izvršujejo
- Glede na število sočasnih tokov ukazov in podatkov ločimo računalnike na naslednje modele (Flynn):
  - SISD (Single Instruction Single Data Stream)
  - SIMD (Single Instruction Multiple Data Stream)
  - MISD (Multiple Instruction Single Data Stream)
  - MIMD (Multiple Instruction Multiple Data Stream)
    - skupni pomnilnik (shared memory)
    - porazdeljeni pomnilnik (distributed memory)

SISD računalnik



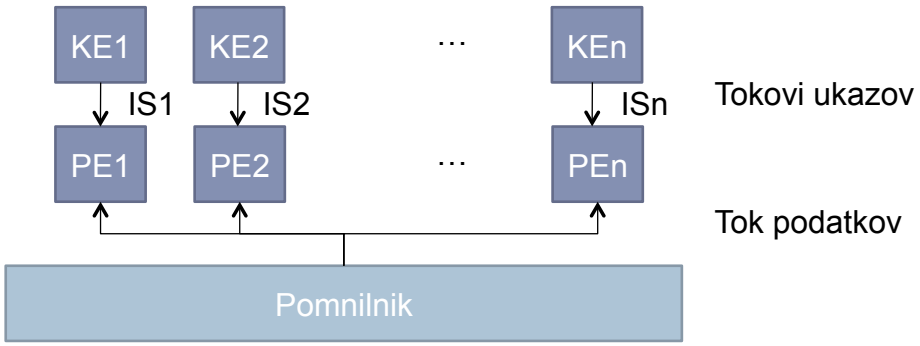
- von Neumannov tip računalnika
- sekvenčni računalnik:
  - računalnik izvršuje ukaze enega za drugim
- Primer: seštevanje N števil

SIMD računalnik



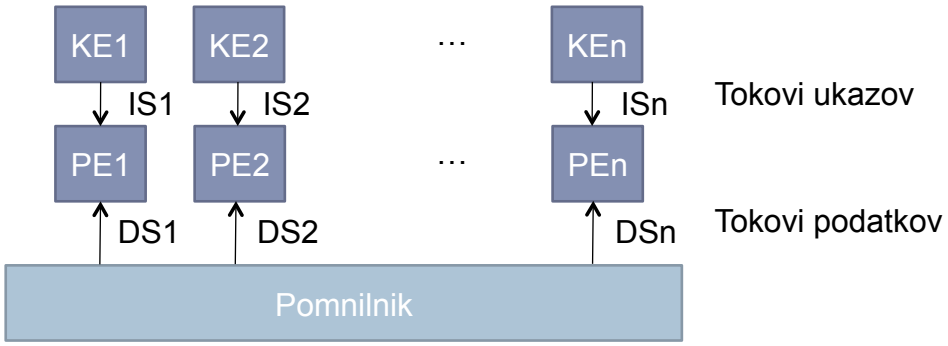
- računalnik izvršuje v danem trenutku eno operacijo nad več kot enim podatkom
- hkraten dostop do podatkov preko vektorjev ali polj
- Primer: hkratno seštevanje več parov števil

MISD računalnik



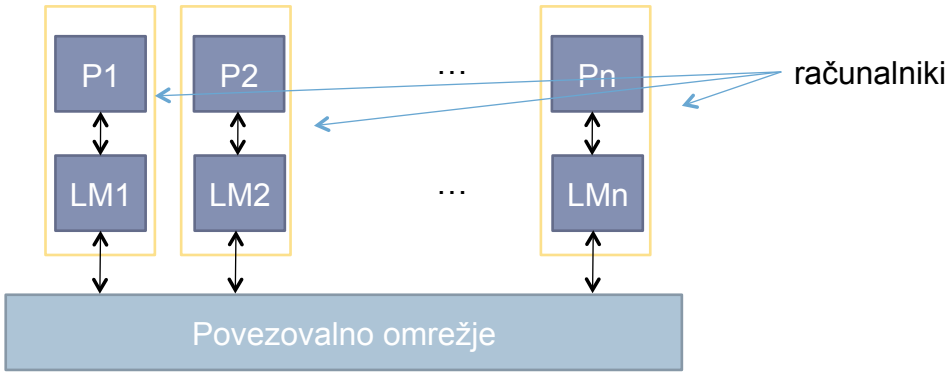
- računalnik izvršuje v danem trenutku več operacij nad enim podatkom
- model ni bil implementiran
- Primer: Ali je število N praštevilo?

MIMD računalnik s skupnim pomnilnikom



- računalnik izvršuje v danem trenutku več operacij nad različnimi podatki
- Večprocesorski sistem:
  - procesorji si delijo skupen pomnilnik
  - procesorji dostopajo do globalnega pomnilnika preko skupnega vodila (ozko grlo, če imamo veliko procesorjev).
  - komunicirajo med seboj z branjem in pisanjem v skupni pomnilnik

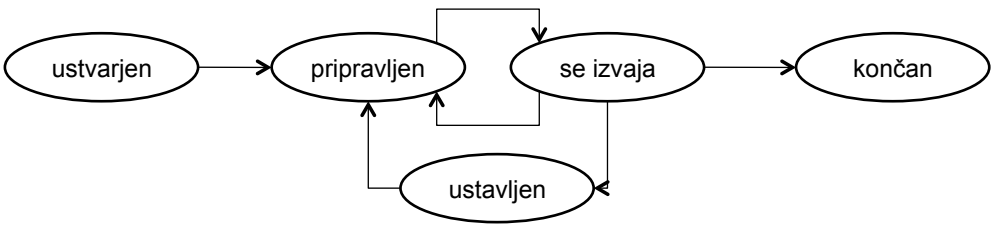
MIMD računalnik s porazdeljenim pomnilnikom



- Večračunalniški sistem:
  - sistem več računalnikov, ki so povezani s povezovalnim omrežjem (ne delijo si skupnega pomnilnika!)
  - komunicirajo med seboj z izmenjavo sporočil
  - različne strukture večračunalniških sistemov:
    - enako/različno zmogljivi računalniki (procesorji)
    - topologije povezovalnih omrežij (mreže, hiperkocke, …)
    - upravljanje virov: centralizirano, porazdeljeno
    - tipi povezav: myrinet, infiniband , ethernet

Osnovni pojmi pri vzporednem programiranju: PROCES

- Opravilo *(ang. task)* – enota izvajanja računalniškega programa, ki opravlja neko nalogo.
  - **pri VP:** ukazi (operacije) v opravilu se izvajajo zaporedoma
- Proces *(ang. process)* – računalniški program v izvajanju (ko program začne teči, postane proces):
  - izvaja lahko eno ali več nalog ali opravil
  - med izvajanjem proces spreminja stanje:



PROCESI v operacijskih sistemih

- V večopravilnem OS obstaja v določenem obdobju več procesov, od katerih je vsak na določeni stopnji napredovanja.

- V stanju izvajanja: en proces na procesor
- V stanju pripravljen ali ustavljen: poljubno mnogo

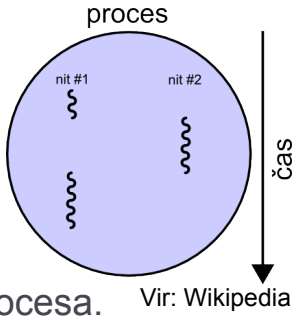
- V vsakem OS imamo razvrščevalca procesov *(ang. scheduler)*: določa, kateri proces se bo izvajal naslednji (in koliko časa)

- Menjavo procesov v procesorju opravi dispečer: izvede se t.i. kontekstni preklop (*ang. context switch*):

- Kontekst procesa – vse kar je potrebno za izvajanje procesa:
  - naslovni prostor: text, data, stack, shared memory ...
  - kontrolni podatki
  - proces ID, program counter, stack pointer, registri, ..

PROCESI in NITI

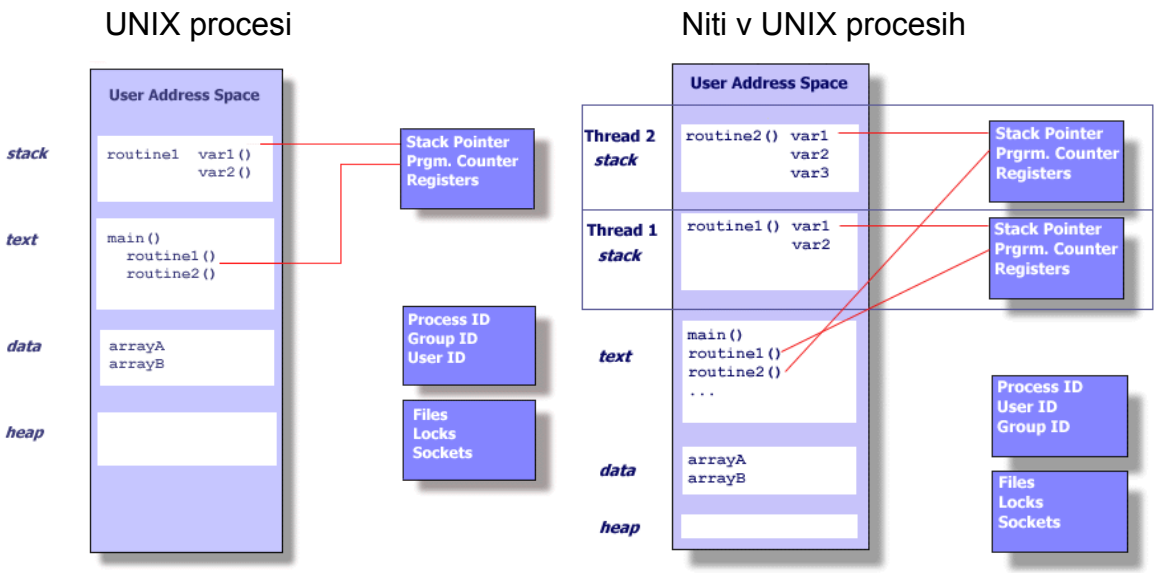
- Vsak proces lahko uporablja eno ali več **niti** izvajanja.
- Nit je nov proces, ki souporablja naslovni prostor očeta:
  - ima svoj sklad, svoje registre,
  - ima svoj ID,
  - z ostalimi nitmi skupnega procesa deli skupni naslovni prostor in globalne spremenljivke.
  - ima dostop do skupnih datotek,
  - ima pravice dostopa do vsega znotraj tega procesa.



Vir: Wikipedia

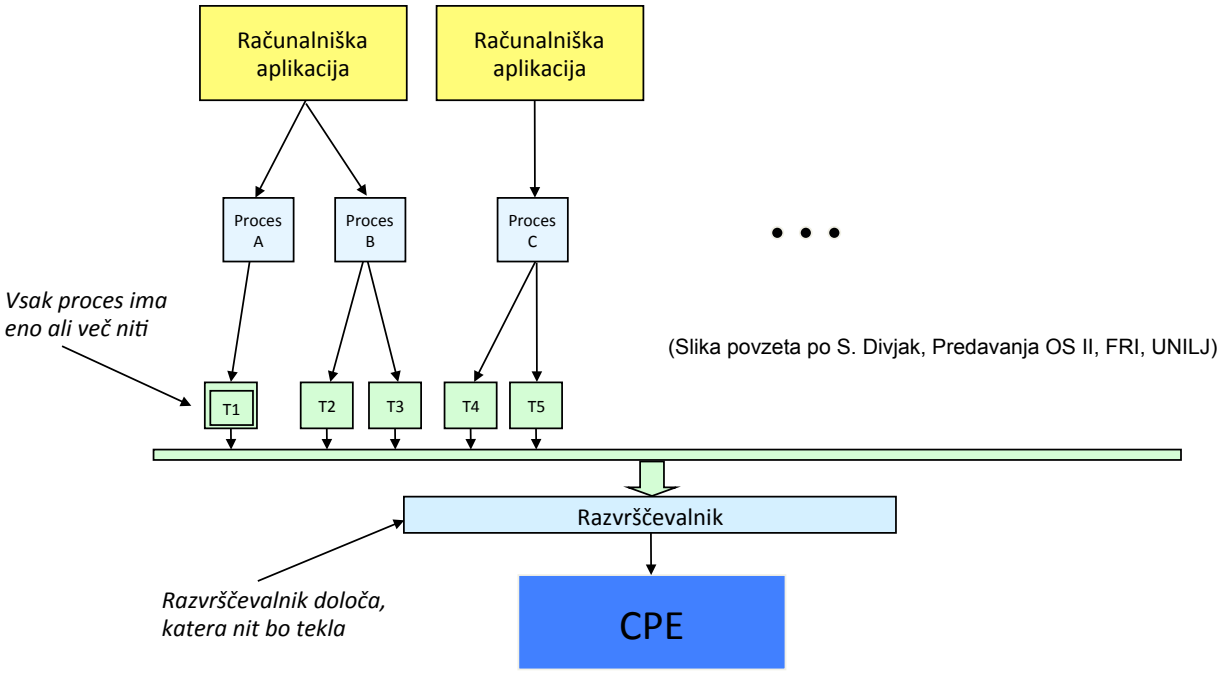
- Primer: LINUX fork() in clone():
  - **fork()** tvori nov proces z novim kontekstom procesa
  - **clone()** tvori nov proces z lastno identiteto, vendar s souporabo podatkovnih struktur očeta.

Procesi in niti



Vir: https://computing.llnl.gov/tutorials/pthreads/

Procesi in niti v enoprocesorskem sistemu



(Slika povzeta po S. Divjak, Predavanja OS II, FRI, UNILJ)

Oblike vzporednega programiranja

- Pri vzporednem programiranju določamo, kako procesi komunicirajo med seboj (medprocesna komunikacija) pri reševanju določene naloge in kakšne metode sinhronizacije se pri tem uporabljajo.

- Programiranje z uporabo skupnega pomnilnika *(ang. shared memory)*
  - procesi si delijo skupen pomnilnik,
  - komunikacija poteka preko branja in pisanja podatkov v skupne (globalne) spremenljivke (podatkovne strukture)
  - potrebna je eksplicitna sinhronizacija, kdaj lahko procesi pišejo/berejo iz skupnih spremenljivk

- Programiranje z uporabo porazdeljenega pomnilnika *(ang. distributed memory)*
  - procesi nimajo skupnega pomnilnika, delijo si povezovalne strukture (komunikacijske kanale)
  - komunikacija poteka s pošiljanjem in sprejemanjem sporočil preko komunikacijskih kanalov
  - sinhronizacija je implicitna: da bi bilo sporočilo sprejeto, mora biti poslano.

Mehanizmi sinhronizacije in komunikacije

- Mehanizmi sinhronizacije (skupni pomnilnik):
  - ključavnice,
  - pregrade,
  - pogojne spremenljivke,
  - semaforji,
  - monitorji.

- Mehanizmi komunikacije (porazdeljeni pomnilnik):
  - s postopki izmenjave sporočil
    - asinhrone in sinhrone izmenjava sporočil
  - s klici za izvajanje oddaljenih procedur (operacij)
    - z metodo RPC (remote procedure call) in po principu “rendezvous”
    - z metodo RMI (remote method invocation)

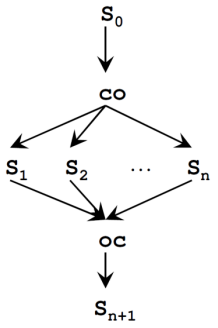
Psevdo koda, uporabljena na predavanjih in v knjigi

- Jezik podoben kot je ***C***.
- Deklaracije – kot v ***C*** ali/in ***matlab***-u
  - spremenljivke: osnovni tipi spr. (**`int`**, **`double`**, **`char`**, …)
  - polja: **`int c[1:n], double c[n,n]`**
  - deklaracija procesa **`process`** – začni z izvajanjem enega ali več procesov v ozadju
- Ukazi:
  - prireditveni ukazi **`a= y*x + f(z)`**
  - ukazi kontrole toka programa: **`if, while, for`** (kot v ***C***-ju)
    - različne izvedbe **`for`** zanke (***matlab*** način)
      - **`for [i=0 to n-1, j=0 to n-1]`**
      - **`for [i=1 to n by 2]`** #vsa liha števila od 1 do n
      - **`for [i=1 to n st i != x]`** #vsa števila od 1 do n, razen i==x (st = "such that")
    - **ukaz za sočasno izvajanje** – **`co`**
      - ukaz začne s sočasnim izvajanjem dveh ali več niti, ki jih čaka, dokler ne končajo
    - **ukaz za sinhronizacijo** – **`await`**
      - ukaz čaka, dokler ni izpolnjen nek pogoj (predstavljen bo v nadaljevanju)

Dva zapisa ukaza `co`

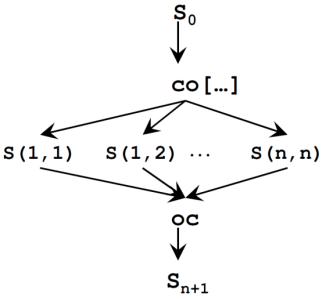
- Vejni zapis
  - vsaka nit je definirana v eni veji
  - niti se izvajajo vzporedno
  - v točki **`oc`** čakamo dokler se vse niti ne zaključijo

```
S0;
co S1; # thread 1
|| ...
|| Sn; # thread n
oc;
Sn+1;
```



- Zapis z iterativnimi spremenljivkami
  - niti so definirane s kombinacijo iterativnih spremenljivk, definiranih v **`co`** ukazu
  - niti se izvajajo vzporedno, v točki **`oc`** čakamo dokler se vse niti ne zaključijo

```
S0;
co [i=1 to n, j=1 to n] { # n x n threads
    S(i,j);
}
oc;
Sn+1;
```



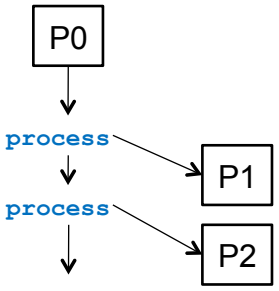
Deklaracija `process`

- Sintaksa je podobna kot pri ukazu `co`.
- Dva zapisa:
  - zapis enega procesa

```
process bar1 {
    for [i=1 to n]
        write(i);
}
```

  - zapis več procesov

```
process bar2 [i= 1 to n] {
        write(i);
}
```



- Proces se začne izvajati ob deklaraciji in se izvaja v ozadju.
- Program nadaljuje izvajanje naprej
  - ne čakamo, da se procesi zaključijo
- Znotraj procesa lahko definiramo lokalne spr. in uporabljamo globalne spr.

Koncepti vzporednega/porazdeljenega programiranja

- iterativni paralelizem
- rekurzivni paralelizem
- princip proizvajalec/porabnik
- princip odjemalec/strežnik
- princip interakcije med enakovrednimi

Iterativni paralelizem

- Paralelizacija med seboj neodvisnih iteracij v sekvenčnih postopkih.
- Npr. paralelizacija zank, kjer se posamezna iteracija zanke izvaja neodvisno od ostalih iteracij.
- Primer: množenje matrik C = A*B
  - **sekvenčni program**

```
double a[n,n], b[n,n], c[n,n]
for [i=0 to n-1] {
  for [j=0 to n-1] {
    c[i,j] = 0;
    for [k=0 to n-1]
      c[i,j] = c[i,j]+a[i,k]*b[k,j];
  }
}
```

  vzporedni program

```
double a[n,n], b[n,n], c[n,n]
co [i=0 to n-1] {
  for [j=0 to n-1] {
    c[i,j] = 0;
      for [k=0 to n-1]
        c[i,j] = c[i,j]+a[i,k]*b[k,j];
  }
} oc;
```

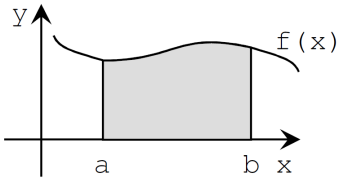
  - zamenjamo `for` s `co`,
  - dobimo n niti za vsako vrstico i, ki se izvajajo sočasno,
  - lahko bi paralelizirali izračun po vrsticah in stolpcih (n^2 niti)
    - `co [i=0 to n-1, j=0 to n-1]`

Rekurzivni paralelizem

- Paralelizacija med seboj neodvisnih rekurzivnih klicev.
  - imamo program, ki se rekurzivno izvaja nad podatki
  - rekurzije so med seboj neodvisne => vsak rekurziven klic izvedemo v svoji niti.
- Primeri:
  - quick sort, rekurzivni izračun določenega integrala, …
  - povsod tam, kjer lahko uporabljamo rekurzijo po principu “deli in vladaj” – torej, kjer lahko uporabimo isti postopek na manjših delih podatkov, da rešimo problem na vseh podatkih.

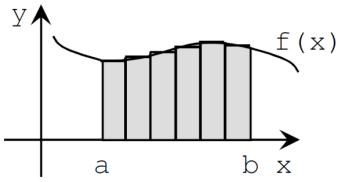
Aproksimacija izračuna določenega integrala

- Naloga:
  - Aproksimiraj izračun določenega integrala funkcije na nekem intervalu.

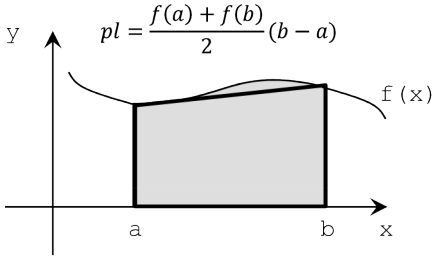


  - **Iterativni postopek (sekvenčni program)**
    - uporabljamo trapezoidno pravilo

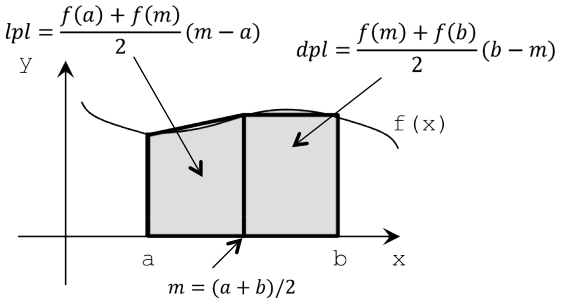
```
double fl = f(a), fr = f(b), pl = 0.0;
double dx = (b-a)/ni;
for [x = (a+dx) to b by dx] {
  fr = f(x);
  pl = pl + (fl+fr)*dx/2;
  fl = fr;
}
```



Rekurzivni izračun določenega integrala



prva aproksimacija
pl



druga aproksimacija
lpl + dpl

- Rekurzija:
  - na danem intervalu izračunamo ploščino **`pl`**
  - interval razdelimo na dve polovici in izračunamo levo in desno ploščino, **`lpl`** in **`dpl`**
  - rekurzijo ustavimo, ko **`|(lpl+dpl) - pl|< eps`**

Rekurzivna aproksimacija izračuna določenega integrala

Sekvenčni program

```
double ninteg(double l,d,fl,fd,pl) {
  double m = (l+d)/2;
  double fm = f(m);
  double lpl = (fl+fm)*(m-l)/2;
  double dpl = (fm+fd)*(d-m)/2;
  if (abs((lpl+dpl)-pl) > eps) {
    lpl = ninteg(l,m,fl,fm,lpl);
    dpl = ninteg(m,d,fm,fd,dpl);
  }
  return (lpl+dpl);
}
```

Vzporedni program

```
double ninteg(double l,d,fl,fd,pl) {
  double m = (l+d)/2;
  double fm = f(m);
  double lpl = (fl+fm)*(m-l)/2;
  double dpl = (fm+fd)*(d-m)/2;
  if (abs((lpl+dpl)-pl) > eps) {
    co lpl = ninteg(l,m,fl,fm,lpl);
    || dpl = ninteg(m,d,fm,fd,dpl);
    oc;
  }
  return (lpl+dpl);
}
```

Začetni klic procedure

```
pl = ninteg(a,b,f(a),f(b),(f(a)+f(b))+(b-a)/2)
```

- Rekurzivni klici so med seboj neodvisni in se zato lahko izvajajo sočasno.

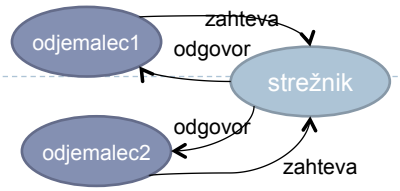
Princip proizvajalec/porabnik

- Proizvajalec proizvaja podatke
- Porabnik uporablja podatke
- Gre za vzajemen odnos, proizvajanje in poraba podatkov se izvaja sočasno.
- Tok podatkov je usmerjen od proizvajalca k porabniku. Vmes imamo lahko “filtre”.
- Primer: pipe v Unix-u

```
$ cat list-1 list-2 list-3 | sort | uniq > final.list
    # Concatenates the list files,
    # sorts them,
    # removes duplicate lines,
    # and finally writes the result to an output file.
```

  - vsak program bere iz vhoda in piše na izhod, program uporablja podatke, ki jih proizvaja predhoden program in proizvaja podatke za svojega naslednika,
  - podatki med programi (procesi) se shranjujejo v medpomnilniku organiziranem s FIFO vrsto.

Princip odjemalec/strežnik



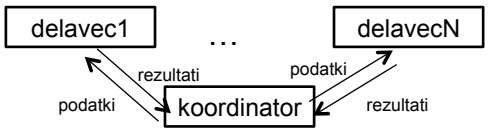
- Odnos odjemalec – strežnik
  - odjemalec zahteva neko storitev,
  - strežnik, ki omogoča storitev, se odzove na zahtevo in pošlje odgovor
  - obojestranska komunikacija: zahteva, odgovor
- Paralelizem mora biti omogočen na strani strežnika, ki mora "streči" več odjemalcem:
  - večnitni proces, včasih potrebna tudi sinhronizacija med storitvami,
  - odjemalcev je lahko več.
- Izvedba:
  - pri porazdeljenih sistemih: komunikacija s pošiljanjem sporočil, s klici oddaljenih procedur (RPC, RMI, …)
  - pri skupnem pomnilniku: zahteve izvedene kot funkcije programa (npr. `open, read, write`), potreba po sinhronizaciji
- Primer: datotečni strežnik
  - zahteve odjemalca: `open, read, write, close`
  - odgovori strežnika: podatki iz datoteke, status izvedbe zahteve (uspelo je, težave …)

Princip interakcije med seboj enakovrednih procesov

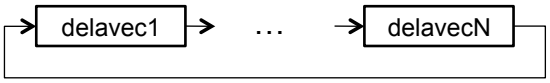
- Tu rešujemo nek problem tako, da delo pri reševanju porazdelimo med več "delavcev", ki vsi opravljajo približno enako količino dela. Delavci s svojim delom in sodelovanjem med seboj rešijo problem.
- Običajno: rešujemo nek problem, ki vključuje obdelavo vhodnih podatkov, podatke razdelimo na manjše dele in potem enak postopek izvedemo na vsakem delu podatkov (delavci). Običajno je za končno rešitev potrebna komunikacija med temi postopki.

Princip interakcije enakih

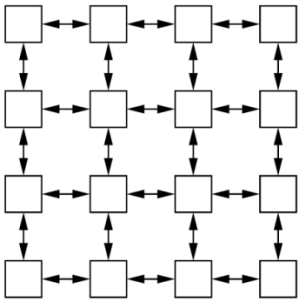
- Načini porazdelitve dela in medsebojne interakcije:
  - koordinator, delavci



  - krožna interakcija

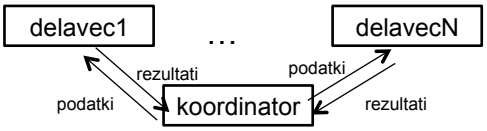


  - mrežne interakcije (grid)



  - …

Koordinator/delavci: množenje matrik



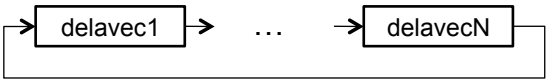
Delavec

```
process worker[i = 0 to n-1] {
  double a[n];     # row i of matrix a
  double b[n,n];   # all of matrix b
  double c[n];     # row i of matrix c
  receive initial values for vector a and matrix b;
  for [j = 0 to n-1] {
    c[j] = 0.0;
    for [k = 0 to n-1]
      c[j] = c[j] + a[k] * b[k,j];
  }
  send result vector c to the coordinator process;
}
```

Koordinator

```
process coordinator {
  double a[n,n];   # source matrix a
  double b[n,n];   # source matrix b
  double c[n,n];   # result matrix c
  initialize a and b;
  for [i = 0 to n-1] {
    send row i of a to worker[i];
    send all of b to worker[i];
  }
  for [i = 0 to n-1]
    receive row i of c from worker[i];
  print the results, which are now in matrix c;
}
```

Krožna interakcija: množenje matrik




```
process worker[i = 0 to n-1] {
  double a[n];        # row i of matrix a
  double b[n];        # one column of matrix b
  double c[n];        # row i of matrix c
  double sum = 0.0;   # storage for inner products
  int nextCol = i;    # next column of results
  receive row i of matrix a and column i of matrix b;
  # compute c[i,i] = a[i,*] × b[*,i]
  for [k = 0 to n-1]
    sum = sum + a[k] * b[k];
  c[nextCol] = sum;
  # circulate columns and compute rest of c[i,*]
  for [j = 1 to n-1] {
    send my column of b to the next worker;
    receive a new column of b from the previous worker;
    sum = 0.0;
    for [k = 0 to n-1]
      sum = sum + a[k] * b[k];
    if (nextCol == 0)
      nextCol = n-1;
    else
      nextCol = nextCol-1;
    c[nextCol] = sum;
  }
  send result vector c to coordinator process;
}
```

2 Sočasno tekoči procesi

V tem poglavju se ukvarjamo s sočasno tekočimi procesi in njihovo sinhronizacijo, definiramo pojem nedeljive operacije ali t.i. atomarnega ukaza ter lastnost enkratnosti in kritične reference. Vpeljemo psevdo ukaz `await`,predstavimo nekaj primerov paralelizacije, kjer ugotavljamo potrebo po medsebojni sinhronizaciji sočasno tekočih procesov, ukvarjamo pa se tudi za lastnostmi izvajanja vzporednih programov.

Poglavje obravnava:

- **Procesi:**
  - izvedba operacij procesa - nedeljiva izvedba,
  - stanja procesa in prepletanje stanj sočasno tekočih procesov.
- **Sinhronizacija:**
  - atomarnost operacij,
  - ukaz `await`.
- **Primeri paralelizacije in sinhronizacije:**
  - kopiranje tabele med proizvajalcem in porabnikom,
  - iskanje maksimalnega elementa v tabeli,
  - iskanje preseka dveh tabel.
- **Lastnosti izvajanja vzporednih programov:**
  - varno izvajanje, živost procesa,
  - razvrščanje izvajanja procesov.

Proces: stanje, nedeljiva operacija

- **Proces** je enota programa, ki izvaja določeno nalogo (opravilo).
- **Stanje procesa** predstavljajo vrednosti vseh spremenljivk, ki jih uporablja proces za izvajanje svojega dela, v določenem časovnem trenutku.
- Vsak proces izvaja določeno zaporedje ukazov.
- Vsak ukaz je sestavljen iz ene ali več **operacij (nedeljivo izvedenih)**, ki spreminjajo eno stanje procesa v drugega.

  (**Nedeljiva) operacija** – sprememba enega stanja procesa v drugo

Atomarni ukazi

- **Atomarni ukaz** – zaporedje operacij (prehodi iz stanja v stanje) pri izvajanju procesa, ki se izvede kot nedeljiva celota.
- Strojni ukazi, ki so implementirani neposredno v procesorski arhitekturi (`read, write, swap`), se izvajajo kot atomarni ukazi.
- V našem primeru: implementacija v programu
  - `<ukazi;>` - zaporedje ukazov, ki morajo biti izvedeni kot atomarni ukaz.
    - torej: atomarni ukaz mora biti izveden kot ena nedeljiva operacija procesa
    - primer: kritična področja (podrobneje v nadaljevanju)

Procesi in strojni atomarni ukazi

- Predpostavimo osnovni model računalnika, ki ga uporabljamo za izvajanje procesov:
  - vrednosti osnovnih tipov spremenljivk (`int, char,...`) so shranjeni kot osnovni pomnilniški elementi (besede) in do njih dostopamo z atomarnimi ukazi: `read, write`
  - operacije nad vrednostmi spr. se izvajajo v registrih, spr. naložimo v register (`load`), izvedemo operacijo (npr. `add`), shranimo rezultat v pomnilnik (`store`)
  - vsak proces ima svoj nabor registrov in svoj sklad. Ko se izvede menjava procesa z dispečerjem (glede na razvrščevalnik) se zamenja cel nabor registrov in sklad (zamenja se kontekst procesa).
  - Vsak vmesni rezultat pri izvedbi kompleksnih ukazov procesa se shranjuje v registre ali v sklad procesa.

Prepletanje stanj sočasno tekočih procesov

- Vsak proces izvaja neko zaporedje atomarnih operacij, pri izvajanju povzroča spremembe svojih stanj.
  - Vsak proces ima svoj potek spreminjanja stanj.
- Več sočasno tekočih procesov, vsak svoj potek:
  - Proces 1: $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots \rightarrow s_n$
  - Proces 2: $p_0 \rightarrow p_1 \rightarrow p_2 \rightarrow \ldots \rightarrow p_n$
  - Proces 3: $q_0 \rightarrow q_1 \rightarrow q_2 \rightarrow \ldots \rightarrow q_n$
- Več različnih skupnih potekov izvajanja sočasnih procesov:
  - Sled 1: $s_0 \rightarrow p_0 \rightarrow s_1 \rightarrow p_1 \rightarrow p_2 \rightarrow q_0 \rightarrow s_2 \rightarrow q_1 \rightarrow \ldots$
  - Sled 2: $p_0 \rightarrow s_0 \rightarrow q_0 \rightarrow q_1 \rightarrow s_1 \rightarrow p_1 \rightarrow p_2 \rightarrow q_2 \rightarrow \ldots$

Sočasna izvedba procesov je nedeterministična!

- Kakšno bo prepletanje operacij med sočasno izvedenimi procesi je nemogoče predvideti:
  - vsakič imamo različno zgodovino spreminjanja stanj (zaporedij operacij),
  - zaporedja operacij ni mogoče ponoviti => nedeterministično obnašanje
- Primer: če imamo vzporeden program z *n* procesi in vsak proces izvede *m* (atomarnih) operacij, potem imamo $(n.m)!/(m!^n)$ možnih prepletanj:
  - *n=3, m=2, 90 možnih izvedb programa*
- Zato tudi:
  - nemogoče je dokazovati pravilnost delovanja programa samo s testiranjem (“program poženemo in spremljamo kaj se dogaja”)

Primer nedeterminističnega obnašanja vzporednega programa

- Program:

```
int y = 0; z = 0;
co x = y + z; ||  y = 1; z = 2; oc
```

- Možnost 1:

```
x = y{0} + z{0}; y = 1; z = 2;      {x == 0}
```

- Možnost 2:

```
y = 1; x = y{1} + z{0}; z = 2;      {x == 1}
```

- Možnost 3:

```
y := 1; z := 2; x := y{1} + z{2};   {x == 3}
```

- Možnost 4:

```
load y{0} to R1; y := 1; z := 2;
  add z{2} to R1{0}; store R1 to x; {x == 2}
```

Kritične reference in lastnost enkratnosti

- **Kritična referenca** v izrazu je referenca na spremenljivko v izrazu, ki jo (lahko) spreminja drugi proces.

- **Lastnost enkratnosti** ***(ang. at most once property)*****:**
  če je `x := e;`
  - (1) `e` vključuje samo eno kritično referenco in
    `x` ne bere noben drugi proces (torej samo eden).
  - (2) `e` ne vključuje kritičnih referenc,
    v tem primeru lahko `x` bere več procesov.

Primeri kritičnih referenc in lastnosti enkratnosti

```
int x = 0, y = 0;
co x = x + 1; || y = y + 1; oc
```

- Lastnost enkratnosti velja.

```
int x = 0, y = 0;
co x = y + 1; || y = y + 1; oc
```

- Lastnost enkratnosti velja.

```
int x = 0, y = 0;
co x = y + 1; || y = x + 1; oc
```

- Lastnost enkratnosti NE velja.

Sinhronizacija

- Sinhronizacija je nujno potrebna, da zagotovimo želeno delovanje vzporednih programov.

- Sinhronizacija je mehanizem, s katerim omogočamo zaustavitev in ponovno nadaljevanje procesov.
  - Na ta način lahko kontroliramo prepletanje delovanja med procesi in dovolimo samo tista stanja, ki vodijo do končne (želene) rešitve programov.

- Dva mehanizma sinhronizacije (kontrole):
  - princip **medsebojnega izključevanja** *(ang. mutual exclusion)*
    - *zagotovimo dostop do kritičnih področij samo enemu procesu naenkrat*
  - princip **pogojne sinhronizacije** *(ang. conditional synchronization)*
    - *zaustavimo izvajanje procesa, dokler ni izpolnjen določen pogoj*

Ukaz `await` in oblike sinhronizacije

- **`<await (B) S;>`**
  - Čakaj dokler ne postane pogoj B **`true`**, takoj nato izvedi ukaze S atomarno.
  - Kombinacija principa medsebojnega izključevanja in pogojne sinhronizacije:
    - Izveden je kot atomarni ukaz.
    - Pogoj B je zagotovo izpolnjen, ko se začne izvajanje ukaza S.
    - Ukazi S se izvajajo atomarno in se zagotovo končajo.

- **`<await (B);>`**
  - Čakaj dokler ne postane pogoj B **`true.`**
  - Princip **pogojne sinhronizacije**.

- **`<S;>`**
  - Ukazi S se izvajajo atomarno in se zagotovo končajo.
  - Uporabimo pri kritičnih področjih za zagotavljanje **medsebojnega izključevanja.**

Primeri ukaza `<await (B) S;>`

- **`<await (s>0) s = s-1;>`**
  - ukaz čaka dokler ni **`s`** pozitiven,
  - nato **`s`** zmanjšamo za 1
  - s se zmanjša samo v primeru, ko je pozitiven !!! (vmes se ne zgodi nič)

- **`<await (s>0)>`**
  - ukaz čaka dokler ni **`s`** pozitiven

- **`<s = s-1;>`** atomarni ukaz
  - **`s`** zmanjšamo za 1,
  - ukaz se izvede atomarno: operacije shranjevanja v register (**`load`**), odštevanja (**`sub`**) in zapisa nazaj v pomnilnik (**`store`**) se izvedejo nedeljivo,
  - stanja procesa, ki jih povzročajo te operacije, so za ostale procese nevidna!!! Zato se ne morajo prepletati s stanji ostalih sočasno tekočih procesov.

Izvedba ukaza `<await (B) S;>`

- Ukaz **`<await (B) S;>`** je težko implementirati v programu v splošni obliki.

- S tem ukazom si bomo pomagali v nadaljevanju pri razlagi splošnih konceptov vzporednega programiranja:
  - Konkretna izvedba bo odvisna od primera do primera.

- Posebni primeri implementacije:
  - **`< S;> ≡ S;`**
    - V primeru, ko ukazi v **`S`** izpolnjujeo lastnost enkratnosti.
  - **`<await (B)> ≡ while(not B);`**
    - V primeru, ko ukazi v izrazu **`B`** izpolnjujejo lastnost enkratnosti.

Primer uporabe ukaza **<await (B) S;>**

- Vzporedni program:

```
int x = 1, y = 2, z = 3;
co x = x + 1;
|| y = y + 2;
|| z = x + y;
|| <await (x > 1) x = 0; y = 0; z = 0;>
oc
```

- Ali se vzporedni program konča?
- Kakšne so končne vrednosti spr. **x**, **y** in **z**?
- Ne pozabimo: seštevanje ni atomarna operacija, ampak moramo najprej prebrati spr., prišteti in nato zapisati nazaj v pomnilnik.

Končni rezultat vzporednega programa

- Vzporedni program:

```
int x = 1, y = 2, z = 3;
co x = x + 1;
|| y = y + 2;
|| z = x + y;
|| <await (x > 1) x = 0; y = 0; z = 0;>
oc
```

- Program se konča.
- Možne končne vrednosti spr. so:

```
x == {0}
y == {0,2,4}
z == {0,1,2,3,4,5,6}
```

Primer kopiranja tabele po principu proizvajalec/porabnik

- Naloga:
  - Denimo, da ima en program (proizvajalec) definirano lokalno tabelo celih števil `a[n]`. Kopiraj vsebino tabele `a[n]` v tabelo `b[n]`, ki je pripravljena (inicializirana) v drugem programu (porabnik).
  - Program nima dostopa do lokalne tabele v drugem programu. Komunikacija med programoma poteka preko skupne (globalne) spremenljivke `buf`.
- Rešitev:
  - Proizvajalec piše v `buf`, porabnik bere iz `buf`.
  - To počneta izmenično: najprej proizvajalec zapiše 1. element iz tabele `a` v `buf`, nato porabnik prebere vrednost `buf` in ga zapiše na 1. mesto v tabeli `b`, potem proizvajalec zapiše 2. element iz tabele `a` v `buf`, porabnik ga prebere in zapiše na 2. mesto v tabeli `b`, itn.
  - Potrebno je izvesti sinhronizacijo. Kdaj lahko pišemo v `buf` in kdaj iz njega lahko beremo. To naredimo z dvema števcema, števcem `p`, ki šteje vpise v `buf`, in števcem `c`, ki šteje branja iz `buf`.

Primer kopiranja tabele po principu proizvajalec/porabnik

```
int buf, p = 0, c = 0;
process Producer {
  int a[n];
  while (p < n) {
    ⟨await (p == c);⟩
    buf = a[p];
    p = p+1;
  }
}
process Consumer {
  int b[n];
  while (c < n) {
    ⟨await (p > c);⟩
    b[c] = buf;
    c = c+1;
  }
}
```

- Neodvisnost obeh procesov.
- Skupna spr. je `buf`.
- Sinhronizacija:
  - medsebojno izključevanje:
    - dostop do `buf` vedno samo enemu procesu
  - pogojna sinhronizacija
    - proizvajalec čaka, dokler ni `buf` pripravljen za ponoven vpis
    - porabnik čaka, da je `buf` dobi novo vrednost
  - izvedba sinhronizacije:
    - s števcema `p` in `c`
    - pogoj: `c <= p <= c+1`

Primer: vzporedno iskanje maksimalnega elementa v tabeli

- Naloga:

  Iščemo maksimalni element v tabeli `a[n]`?
  Predpostavimo, da so elementi tabele pozitivna cela števila.

- Rešitev:

  Izberemo si za spr. `m` neko najmanjšo vrednost, ki jo ne pričakujemo v tabeli `a[n]`.
  Gremo po tabeli a z indeksom `i = 0 to n-1` in za `a[i]` preverimo, če je večji od `m`.
  Če je, potem postane do sedaj največji element, trenutni element, torej `m = a[i],`
  če ne, gremo naprej.

Vzporedno iskanje maksimalnega elementa v tabeli

- Sekvenčni program

```
int m = 0;
for [i = 0 to n-1]
   if (a[i] > m) m = a[i];
```

- Vzporedni program

```
int m = 0;
co [i = 0 to n-1]
   if (a[i] > m) m = a[i]; oc
```

  - A je to dovolj?
  - A program deluje pravilno?
    - Ne.
    - Procesi med seboj niso neodvisni. Vsak bere in piše v spremenljivko m. Torej: na začetku vsi procesi štartajo in vsak primerja svojo vrednost s spr. m, ker je m manjša od vseh vrednosti v tabeli, hoče vsak spreminjati vrednost spr. m s svojo vrednostjo.
    - Seveda to računalnik izvaja zaporedno => na koncu m dobi vrednost tistega procesa, ki se izvede zadnji.
    - **Pojav tekmovanja med procesi *(ang. race condition)***
  - Potrebna je sinhronizacija.

Vzporedno iskanje maksimalnega elementa v tabeli

- Prvi poskus:
  - izvedemo primerjavo in prirejanje kot atomarni ukaz

```
int m = 0;
co [i = 0 to n-1]
   ⟨ if (a[i] > m) m = a[i]; ⟩ oc
```

  - S tem se znebimo učinka tekmovanja. Program nam vrne pravi maksimum. Zakaj?
  - Kaj pa učinkovitost izvajanja?
    Enako kot pri sekvenčnem programu. Procesi se zvrstijo zaporedoma, le da v poljubnem vrstnem redu.

- Opažanja:
  - branje `m` in `a[i]` se lahko izvaja vzporedno za vsak `i`
  - Pisanje v `m` se mora izvajati zaporedoma, torej moramo zagotoviti medsebojno izključevanje pisanja v to kritično spr. - **kritični del izvajanja**.
  - Kako se znebiti problema tekmovanja?

Vzporedno iskanje maksimalnega elementa v tabeli

- Drugi poskus:
  - izvedemo neatomarno primerjavo in nato primerjavo in prirejanje kot atomarni ukaz

```
int m = 0;
co [i = 0 to n-1]
   if (a[i] > m)
      ⟨ if (a[i] > m) m = a[i]; ⟩ oc
```

  - Pri prvi primerjavi ni potrebno zagotavljati medsebojnega izključevanja procesov, ker samo beremo spremenljivke.
  - Pri drugi primerjavi in prirejanju zahtevamo atomarnost. S tem zagotovimo medsebojno izključevanje procesov pri prirejanju. Znebimo se problema tekmovanja med procesi.
  - Učinkovitost izvajanja je boljša kot prej:
    - ni potrebno izvajati atomarnega ukaza (kar ni poceni!), v primeru, ko ni izpolnjen pogoj `a[i] > m`
  - Princip **neatomarne primerjave ter atomarne primerjave in prireditve** (ang. test and test-and-set) se pogosto uporablja v izogib problemu tekmovanja med procesi.

Še en primer: `await` in `if`

- Poglejmo si naslednji primer:

```
int x = 0;
co <await (x != 0) x = x - 2 >;
|| <await (x != 0) x = x - 3 >;
|| <await (x == 0) x = x + 5 >;
oc
```

- (a) Ali se program konča? Če se, kakšne so možne končne vrednosti spr. **x**. Če ne, zakaj ne?
- (b) Zamenjaj ukaz **await** z **if** in odstrani <> oklepaje. Sedaj se program gotovo konča. Kakšne so sedaj vrednosti spr. **x**?

Rešitev naloge

- (a) Program se konča. Končna vrednost spr. **x = 0**. Zadnji proces se izvede prvi, prva dva se nato lahko izvedeta v poljubnem vrstnem redu.

- (b)

```
int x = 0;
co if (x != 0) x = x - 2;   //S1
|| if (x != 0) x = x - 3;   //S2
|| if (x == 0) x = x + 5;   //S3
oc
```

Možni poteki izvedbe procesov in končne vrednosti spr. **x**

pogoja S1 in S2 nista izp., potem S3	**x == 5**
pogoj S2 ni izp.; S3, S1;	**x == 3**
pogoj S1 ni izp.; S3, S2;	**x == 2**
S3; S1 || S2	**x == {0, 2, 3}**

Primer 3: Presek dveh tabel

- Naloga:
  - Imamo dve celoštevilski tabeli `a[1:m]` in `b[1:n]`, denimo da sta obe tabeli že urejeni v naraščajočem vrstnem redu.
  - (a) Naredi sekvenčni program, ki prešteje tiste elemente iz obeh tabel, ki imajo enake vrednosti. Torej preštejemo, koliko je elementov v preseku obeh tabel.
  - (b) Prevedi sekvenčni program v vzporedni program.

Primer 3: Presek dveh tabel

- Rešitev problema v linearnem času:
  - sekvenčni program:

```
int i = 0, j = 0, count = 0;
for (i=0; i < m; i++)
  for (j=0; j < n; j++)
    if (a[i] == b[j]) count++;
```

  - sekvenčni program: izboljšana verzija

```
int i = 0, j = 0, count = 0;
while (i < m && j < n) {
  if (a[i] < b[j]) i++;
  else if (a[i] > b[j]) j++;
  else {count++; i++; j++;}
}
```

  - vzporedni program:

```
int i = 0, j = 0, count = 0, A, B;
while (i < m && j < n) {
  A = a[i]; B = b[j];
  co if (A < B) i++;
  || if (A > B) j++;
  || if (A == B) {count++; i++; j++;}
  oc;
}
```

Ker imamo izključujoče pogoje, ni potrebe po sinhronizaciji.
V vsakem koraku bo izveden natanko en proces, kjer se bo izvedlo spreminjanje vrednosti spr..

Kaj smo se naučili?

- Sinhronizacija je potrebna, kadar več procesov dostopa do skupnih spremenljivk (bere ali piše v skupne spremenljivke).
- Z atomarnim izvajanjem ukazov zagotavljamo medsebojno izključevanje prepletanja izvajanja ukazov med procesi. Zagotovimo medsebojno izključevanje.
- Princip **neatomarne primerjave ter atomarne primerjave in prireditve** (ang. test and test-and-set) se pogosto uporablja v izogib problemu tekmovanja med procesi:
  - v primeru, ko sočasno izvajamo več pogojnih prirejanj na podlagi istega pogoja.
  - primer: iskanje maksimalnega elementa v tabeli

Lastnosti izvajanja vzporednih programov

- Pri vzporednem programu sodeluje dva ali več sočasno tekočih procesov. Izvajanje procesov se med seboj prepleta. Zaporedje stanj vzporednega programa imenujemo potek izvajanja vzp. programa. Imamo lahko več različnih potekov.
- Vprašanja:
  - Kako zagotoviti pravilnost izvajanja?
  - Kako zagotoviti, da se program vedno konča?
  - Kako zagotoviti, da se program vedno pravilno konča?

Lastnosti izvajanja vzporednih programov

- Definicija:
  Lastnost (vzporednega) programa je neka značilnost **P**, ki velja za vsako stanje vseh možnih potekov izvajanja programa.

- Dve pomembni lastnosti, ki zagotavljata pravilnost delovanja in ustavitev izvajanja programov:
  - Lastnosti varnega izvajanja *(ang. safety properties)*:
    - V poteku izvajanja programa nimamo nobenega "slabega" stanja.
  - Lastnosti živosti izvajanja *(ang. liveness properties)*:
    - Ko je zagotovljeno, da izvajanje programa v nekem trenutku preide v "dobro" stanje.

"Dobro" in "slabo" stanje procesa

- Denimo, da je **P** neka lastnost.
  - "dobro" stanje procesa – stanje, v katerem lastnost **P** velja.
  - "slabo" stanje procesa – stanje, v katerem lastnost **P** NE velja.

- Varno izvajanje programa za lastnost **P**:
  - Lastnost **P** velja v vsakem stanju v vseh možnih potekih izvajanja programa. Lastnost **P** je *globalno nespremenljiva* glede na program, ki se izvaja.

- Živost izvajanja programa za lastnost **P**:
  - V vseh možnih potekih izvajanja programa pridemo v stanje, kjer **P** velja.

Primeri lastnosti izvajanja procesov

- Lastnosti varnega izvajanja
  - **Delna pravilnost programa** (ang. partial correctness)
    - Če predpostavimo, da se program konča, torej pride v končno stanje izvajanja, potem to končno stanje pravilno reši nalogo, ki jo program rešuje.
  - **Medsebojno izključevanje** izvrševanja kritičnih področij
    - Zagotavljanje, da je v vsakem trenutku izvajanja programa največ en proces v izvrševanju kritičnega področja.
  - **Preprečevanje smrtnega objema** (ang. deadlock)
    - Smrtni objem: proces čaka, da se izpolni nek pogoj, vendar se to nikoli ne zgodi.
- Lastnosti živosti procesa
  - **Ustavitev izvajanja programa** (ang. termination)
    - Vsi možni poteki izvajanja programa vsebujejo končno število stanj. Program se vedno konča.
  - **Zagotovitev vstopa v kritično področje**
    - V primeru, ko nek proces v času izvajanja potrebuje dostop v kritično področje, se mu to v nekem trenutku izvajanja programa tudi omogoči.
- **Popolna pravilnost programa**: ustavitev + delna pravilnost
  - Program se vedno konča, končno stanje dá pravilen odgovor na nalogo, ki jo rešujemo.

Primer: kopiranje tabele, proizvajalec/porabnik

```
int buf, p = 0, c = 0;
process Producer {
  int a[n];
  while (p < n) {
    ⟨await (p == c);⟩
    buf = a[p];
    p = p+1;
  }
}
process Consumer {
  int b[n];
  while (c < n) {
    ⟨await (p > c);⟩
    b[c] = buf;
    c = c+1;
  }
}
```

- Preprečevanje smrtnega objema – dokaz: ***s protislovjem***
- Globalno nespremenljiva lastnost:
  PC: c <= p <= c+1
- await v proizvajalcu:
  WP: PC ∧ (p<n) ∧ (p≠c)
- await v porabniku:
  WC: PC ∧ (c<n) ∧ (p<=c)
- Smrtni objem:
  WC ∧ WP = (p≠c) ∧ (p==c) = false
- Smrtni objem v tem primeru ni mogoč.

Živost izvajanja programa in razvrščanje procesov

- Lastnosti živosti izvajanja programa slonijo na predpostavki, da nam nekdo/nekaj zagotavlja izvajanje programa.
- Kako se izvaja program in v kakšnem vrstnem redu se izvajajo procesi v programu določa razvrščevalnik procesov.
- Kaj pričakujemo od razvrščevalnika procesov?
  Da je razvrščanje "pravično". To pomeni, da mora izvajanje posameznih (atomarnih) ukazov procesa priti slej ko prej na vrsto.

Načini razvrščanja

```
bool continue = true;
co while (continue);
|| continue = false;
oc
```

- Če imamo npr. razvrščanje takšno, da procesor izvaja nek proces vse dotlej, dokler se proces ne konča, potem se ta program lahko nikoli ne konča:
  - prvi proces se začne izvajati in se nikoli ne konča,
  - **livelock** – program se izvaja, vendar ne napreduje.
- Če pa dobi možnost izvajanja tudi drugi proces, potem se program konča.
- Vse je odvisno od načina razvrščanja.
- Pravični način razvrščanja brezpogojnih ukazov:
  Vsak atomarni ukaz, ki ne vsebuje pogoja za izvedbo, se slej ko prej izvede.
- Primer: round robin na enoprocesorskem sistemu, paralelno izvajanje na večprocesorskem.

Način razvrščanja s pogojnimi atomarnimi ukazi

- Manj pravično razvrščanje (ang. weakly fair scheduling):
  - Je pravično v primeru brezpogojnih atomarnih ukazov.
  - V primeru pogojnih atomarnih ukazov: zagotovljena je izvedba pogojnega atomarnega ukaza, če pogoj pogojnega atomarnega ukaza postane **`true`** in ostane **`true`** do izvedbe ukaza.
    - Primer: round-robin
- Strogo pravično razvrščanje (ang. strong fair scheduling):
  - Je pravično v primeru brezpogojnih atomarnih ukazov.
  - V primeru pogojnih atomarnih ukazov: zagotovljena je izvedba pogojnega atomarnega ukaza, če pogoj pogojnega atomarnega ukaza neskončnokrat postane **`true`** (ni nujno, da ostane **`true`** do izvedbe, pomembno je, da je neskončnokrat **`true`**).
    - V praksi neizvedljivo.

Primer razvrščanja pogojnih atomarnih ukazov

```
bool continue = true, try = false;
co while (continue) {try = true; try = false;}
|| <await (try) continue = false;>
oc
```

- Vzporedni program se v primeru strogo pravičnega razvrščanja konča.
- Vzporedni program se v primeru manj pravičnega razvrščanja ne konča.
- Zakaj?

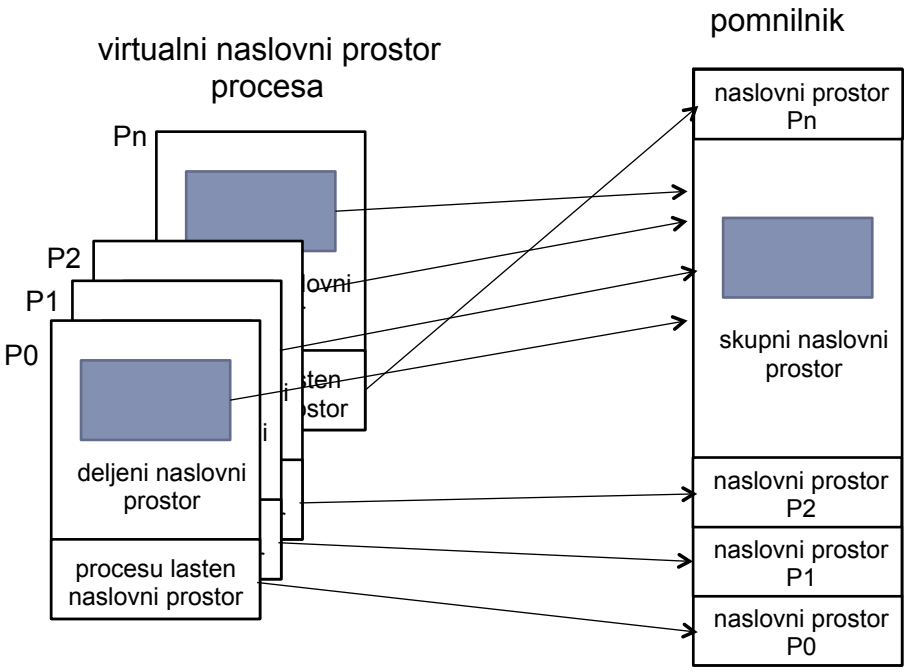
3 Osnovni mehanizmi sinhronizacije med procesi

V tem poglavju se ukvarjamo z osnovnimi tehnikami sinhronizacije med sočasno tekočimi procesi. Definiramo kritična področja procesov in se seznanimo z osnovnimi postopki za vstopanje v kritična področja z uporabo mehanizma ključavnice. Predstavljen je tudi princip pregrade, ki ga uporabljamo za sinhronizacijo med sočasno tekočimi procesi v iterativnih postopkih ter nekaj primerov uporabe teh mehanizmov za reševanje konkretnih problemov.

Poglavje obravnava:

- **Kritična področja:**
  - problem dostopanja do kritičnih področij,
  - princip zaklepanja dostopa do kritičnih področij - ključavnice,
  - različne implementacije ključavnic,
  - vzpostavitev vrstnega reda dostopanja do kritičnih področij:
    - **Petersonov algoritem**,
    - **postopek čakalnih listkov**,
    - **pekovski algoritem**.
  - implementacija ukaza `await` s ključavnicami.
- **Pregrade:**
  - pregrada s števcem,
  - pregrada z zastavicami in koordinatorjem,
  - načini postavitve pregrad med procesi:
    - *asimetrične postavitve pregrad*,
    - *simetrične postavitve pregrad.*
  - primeri uporabe pregrad: podatkovni paralelizem:
    - reševanje diferencialne enačbe z Jakobijevo metodo,
    - *princip vreče opravil.*

Skupni (deljeni) naslovni prostor



(So)uporaba skupnih spremenljivk

- Poglejmo primer:

```
int x = 0;
co x++; || x++; oc
```

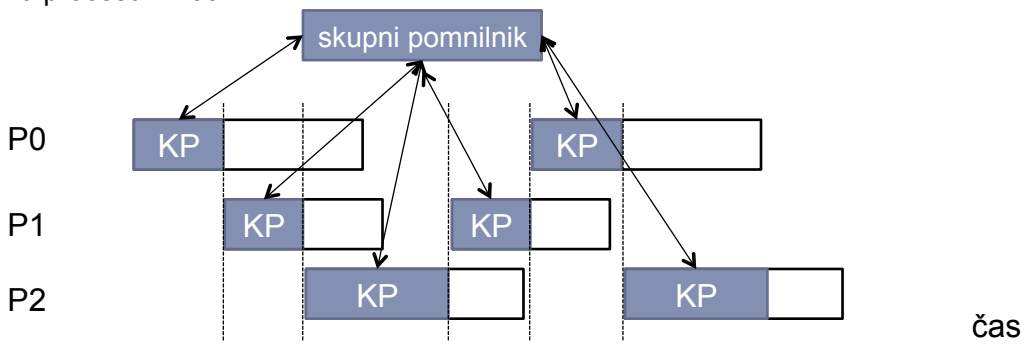
- Kakšne so lahko končne vrednosti spr. **x**? **x == {1, 2}**
- Zakaj **x == 1**?
  - Spomnimo se:

```
P0 :    …;  load x to REG; inc REG, store REG to x; …
P1 :    …;  load x to REG; inc REG, store REG to x; …
```

Kritično področje programa

- Zaporedje ukazov, s katerimi dostopamo do skupnih spremenljivk, imenujemo **kritično področje** programa.
  - Kritično področje programa lahko izvaja samo en proces hkrati, zato se mora zaporedje ukazov kritičnega področja izvesti nedeljivo, kot atomarni ukaz.

- **Problem dostopanja do kritičnega področja:**
  Mehanizem urejanja dostopanja več sočasno tekočih procesov do kritičnega področja programa. Dostopanje in izvajanje kritičnega področja mora biti zagotovljeno samo enemu procesu hkrati.



Kritično področje programa

```
process P[i=0 to n-1] {
  while (true) {
    CSentry;                //vstopni protokol
    critical section;       //kritično področje
    CSexit;                 //izstopni protokol
    non-critical section;
  }
}
```

- Kaj moramo zagotoviti pri implementaciji vstopnih in izstopnih protokolov?
  - **Medsebojno izključevanje izvajanja KP**
    - Največ en proces lahko hkrati vstopi v KP, izvaja KP in izstopi iz KP.
  - **Preprečevanje smrtnega objema**
    - Vsaj enemu izmed procesov, ki tekmujejo za vstop v KP, bo to omogočeno.
  - **Preprečevanje zadržanja izvajanja KP**
    - Če en proces čaka na vstop v KP, vsi ostali pa ne, potem ta proces lahko takoj začne z izvajanjem KP.
  - **Omogočanje vstopa v KP**
    - Vsakemu procesu, ki čaka na vstop v KP, bo slej ko prej to omogočeno.

Zaklepanje dostopa do KP - ključavnice

- Kakšne možnosti izvajanja KP programa imamo?
  - Bodisi samo en izmed procesov izvaja KP, bodisi noben.
  - Uporabimo princip ključavnice: ko vstopimo v KP zaklenemo KP, ko izstopimo odklenemo KP.

- **Ključavnica –** logična spremenljivka, ki je **`true`**, ko je eden izmed procesov v KP in je **`false`**, ko ni noben izmed procesov v KP:
  - zaklepanje in odklepanje ključavnice izvajamo v vstopnem in izstopnem protokolu (CSentry, CSexit).

  - uporabljamo lahko eno ali več ključavnic:
    - dva ekstrema:
      - ena ključavnica za vse skupne spr.;
        lahko neučinkovito, izgubljamo paralelizem
      - vsaka skupna spr. ima svojo ključavnico;
        visoka stopnja paralelizma, veliko breme sinhronizacije
      - Zato vedno tehtamo med stopnjo paralelizma in bremenom sinhronizacije.

Primer izvedbe ključavnice

```
bool lock = false;
process CS1 {
  while (true) {
    ⟨await (!lock) lock = true;⟩  /* entry */
    critical section;
    lock = false;                 /* exit */
    noncritical section;
  }
}
process CS2 {
  while (true) {
    ⟨await (!lock) lock = true;⟩  /* entry */
    critical section;
    lock = false;                 /* exit */
    noncritical section;
  }
}
```

- Rešitev za dva procesa.
- Lahko jo posplošimo na več procesov. Kako?
- Kako implementiramo ukaz **`await`**?

Implementacija ključavnice v ukazu **`await`**

- Obstajajo strojni ukazi, (ki se izvajajo kot atomarni ukazi v računalniku,) s katerimi si lahko pomagamo pri izvedbi ključavnice v ukazu **`await.`**

- Primeri takih strojnih ukazov so:
  - ukaz TS (Test and Set)

```
bool TS (bool lock) {
  < bool initial = lock;  // shrani zacetno vrednost
    lock = true;          // postavi lock na true
    return initial; >     // vrni zacetno vrednost
}
```

  - ukaz FA (Fetch and Add)

```
int FA (int var, int incr) {
  < int tmp = var;       // shrani zacetno vrednost spr. var
    var = var + incr;    // povecaj var za incr
    return tmp; >        // vrni zacetno vrednost spr. var
}
```

  - še drugi ukazi

KP s ključavnico TS

```
bool lock = false;              /* shared lock */
process CS[i = 1 to n] {
  while (true) {
    while (TS(lock)) skip;      /* entry protocol */
    critical section;
    lock = false;               /* exit protocol  */
    noncritical section;
  }
}
```

- Preverjanje lastnosti izvajanja KP:
  - medsebojno izključevanje: DA
  - prepr. smrtnega objema: DA
  - prepr. zadržanja izvajanja: DA
  - omogočanje vstopa v KP: DELNO

(v primeru strogo pravičnega razvrščanja DA, sicer NE, vendar v praksi DA.)

Ključavnica v zanki

```
bool lock = false;              /* shared lock */
process CS[i = 1 to n] {
  while (true) {
    while (TS(lock)) skip;      /* entry protocol */
    critical section;
    lock = false;               /* exit protocol  */
    noncritical section;
  }
}
```

- Poseben primer izvedbe ključavnice:
  Ključavnica v zanki (*ang. spin lock*):
  - procesi se vrtijo v zanki in čakajo, da se ključavnica odklene.

Slabosti ključavnice TS v zanki

```
bool lock = false;              /* shared lock */
process CS[i = 1 to n] {
  while (true) {
    while (TS(lock)) skip;      /* entry protocol */
    critical section;
    lock = false;               /* exit protocol  */
    noncritical section;
  }
}
```

- Spr. **`lock`** je skupna vsem procesom.
- V **`while`** zanki se neprestano piše v spr. **`lock`**.
- Če predpostavimo, da se vsak proces izvaja na svojem procesorju in je shranjena v predpomnilniku tega procesa (za hitrejše dostopanje), potem se mora po vsakem ukazu **`TS`** osvežiti skupni pomnilnik oziroma vsi predpomnilniki ostalih procesorjev. (*ang. memory contention*).

- Zato je cena operacije **`TS`** je večja kot npr. samo branje skupne spremenljivke **`lock`**.

Zato naslednja izvedba: TTS

```
bool lock = false;              /* shared lock */
process CS[i = 1 to n] {
  while (true) {
    while (lock) skip;          /* entry protocol */
    while (TS(lock)) {
      while (lock) skip;
    }
    critical section;
    lock = false;               /* exit protocol  */
    noncritical section;
  }
}
```

- Izvedba: Test and Test & Set
  - Ukaz **`TS`** izvajamo le, če imamo možnost, da napredujemo.
  - Zmanjšamo število operacij **`TS`** in s tem zahteve po neprestanem osveževanju skupnega pomnilnika.
    - Zato pa večkrat samo beremo spr. **`lock`**, kar je cenejše kot **`TS`**.
  - Izstopni protokol samo postavi **`lock`** v prvotno stanje.

Vzpostavitev vrstnega reda izvajanja KP

- Ključavnice v zanki:
  - učinkovit način zagotavljanja izvajanja KP,
  - delovanje po principu:
    procesi čakajo v zanki, da se ključavnica odklene, tisti proces, ki prvi to ugotovi, gre v izvajanje KP, ostali se vrtijo v zanki naprej.
  - kaj lahko se zgodi, da tisti proces, ki je ravnokar izvedel KP, lahko še enkrat ponovno dobi možnost izvajanja.
    To pa ni pošteno. Zato:

- Razvrstitev procesov pred ključavnico:
  - procesi se postavijo v vrsto pred ključavnico in čakajo, da pridejo na vrsto,
  - ko se ključavnica odklene, gre v izvajanje tisti proces, ki je prvi pred ključavnico,
  - več možnih pristopov.

Petersonov algoritem

- Denimo, da imamo dva procesa, ki dostopata do KP.

- Razmislimo:
  - Postavimo ključavnico za vstop v KP.
  - Zapomnimo si, kateri od procesov je zadnji izvajal KP.
  - Če je ključavnica odklenjena in proces ni bil zadnji pri izvajanju KP, lahko začne z izvajanjem KP.

  - Ideja: Nova "ključavnica" za vstop v KP je tako sestavljena iz dveh delov: logične spr., ki je naša "stara" ključavnica in spr., ki vodi vrstni red vstopanja.

Petersonov algoritem

```
bool in1 = false, in2 = false;
int last = 1;
process CS1 {
  while (true) {
    last = 1; in1 = true;     /* entry protocol */
    ⟨await (!in2 or last == 2);⟩
    critical section;
    in1 = false;              /* exit protocol */
    noncritical section;
  }
}
process CS2 {
  while (true) {
    last = 2; in2 = true;     /* entry protocol */
    ⟨await (!in1 or last == 1);⟩
    critical section;
    in2 = false;              /* exit protocol */
    noncritical section;
  }
}
```

- Proces CS1:
  - proces čaka na vstop v KP,
    - če je proces CS2 v izvajanju KP (in2==true) oziroma
    - če je ravnokar prišel iz KP in je na vrsti proces CS2 (last==1).

  - Proces CS2:
    - ravno obratno.

- Program izpolnjuje vse štiri pomembne lastnosti izvajanja KP.

Petersonov algoritem brez atomarnih ukazov

- Zakaj lahko brez atomarnih ukazov ne glede na to, da imamo v vstopnih protokolih dve kritični referenci?
  - Pogoj: **<await (!in2 or last==2)>**
- Proces CS1:
  - Denimo, da je proces v pogoju, kjer čaka na vstop v KP. Denimo, da je pogoj **== true**. Gremo lahko v KP.
  - Če **in2==false**, se lahko zgodi, da je v naslednjem trenutku in2 že **true**. V tem primeru je proces CS2 že postavil **last** na 2.
    Zato je pogoj še vedno **true**, čeprav je **in2** spremenil svojo vrednost.
  - Če je **last == 2**, pogoj še vedno **true**. last ostane 2, dokler CS1 ne izvede KP.
  - Torej: če proces CS1 misli, da je pogoj **true**, je dejansko pogoj **true** (lahko izvede v vstop v KP).
- Za proces CS2 simetrično.
- Torej ne potrebujemo atomarnih pogojnih ukazov, ampak samo zanikanje pogoja v while zanki.

```
bool in1 = false, in2 = false;
int last = 1;
process CS1 {
  while (true) {
    last = 1; in1 = true;     /* entry protocol */
    while (in2 and last == 1) skip;
    critical section;
    in1 = false;              /* exit protocol */
    noncritical section;
  }
}
process CS2 {
  while (true) {
    last = 2; in2 = true;     /* entry protocol */
    while (in1 and last == 2) skip;
    critical section;
    in2 = false;              /* exit protocol */
    noncritical section;
  }
}
```

Postopek čakalnih listkov

- Petersonov algoritem je izveden v primeru dveh procesov.
  - Posplošitev na n procesov je precej zapletena. Potrebno je voditi tabelo ključavnic **in[i]** in tabelo obiskov **last[n]** za vsak proces posebej. Intuitivno je rešitev precej nerazumljiva. Veliko je potrebno sinhronizacije med skupnimi spremenljivkami.
- Obstaja alternativen postopek:
  - posnemanje čakalne vrste pri zdravniku (v naših javnih uradih, itd.)
    - ko pride pacient v čakalnico, vzame listek z zaporedno številko, ki je za ena večja od številke, ki jo ima zadnji pacient v čakalnici,
    - pacient čaka na sprejem, dokler se na zaslonu v čakalnici ne pojavi njegova številka.
    - **Postopek čakalnih listkov.**

Postopek čakalnih listkov

```
int number = 1, next = 1, turn[1:n] = ([n] 0);
process CS[i = 1 to n] {
  while (true) {
    ⟨turn[i] = number; number = number + 1;⟩
    ⟨await (turn[i] == next);⟩
    critical section;
    ⟨next = next + 1;⟩
    noncritical section;
  }
}
```

- Bistvena zahteva je, da vsak proces dobi svojo lastno številko.
- Smo na varni strani: uporabljamo atomarne ukaze.
  - Neatomarna izvedba:
    - 1. atomarni izraz: težave, zapisati moramo v novo spr., pa še prišteti za ena drugo spr. lahko rešujemo z novim KP in ključavnico, vendar ne dobimo poštene rešitve.
    - 2. atomarni ukaz: `while (turn[i] != next) skip;`
    - 3. atomarni ukaz: lahko spustimo < >, ker se izstopni protokol lahko izvaja v samo enem procesu
- Druga težava:
  - Števca **`number`** in **`next`** naraščata čez vse meje.

Postopek čakalnih listkov brez atomarnih ukazov

- Strojni (atomarni) ukaz FA (Fetch and Add)

```
int FA (int var, int incr) {
  < int tmp = var;      // shrani zacetno vrednost spr. var
    var = var + incr;   // povecaj var za incr
    return tmp; >       // vrni zacetno vrednost spr. var
}
```

- Postopek čakalnih listkov, če imamo na razpolago strojni ukaz FA:

```
int number = 1, next = 1, turn[1:n] = ([n] 0);
process CS[i = 1 to n] {
  while (true) {
    turn[i] = FA(number,1);      /* entry protocol */
    while (turn[i] != next) skip;
    critical section;
    next = next + 1;             /* exit protocol  */
    noncritical section;
  }
}
```

Pekovski algoritem

- Kaj pa če nimamo na razpolago ukaza FA?
  - Potrebno izvajanje novega KP in zagotavljanje medsebojnega izključevanja s ključavnicami, kar privede do nepoštene rešitve čakalnice.

- Alternativni postopek: Pekovski algoritem
  - postopek podoben kot postopek čakalnih listkov, le da tu pacient (v našem primeru kupec pri peku) ne dobi listka s številko, ki je za ena večja od številke zadnjega pacienta (kupca), ampak **je za ena večja od največje izmed vseh številk pacientov v čakalnici** (kupcev v prodajalni).
  - pacient (kupec), ki ima najmanjšo številko izmed pacientov (kupcev) v čakalnici je sprejet (postrežen).
  - razlika s prejšnjim postopkom:
    - ne vodimo globalnih števcev, ampak primerjamo številke med trenutnimi številkami, ki so na razpolago.

Pekovski algoritem

```
int turn[1:n] = ([n] 0);

process CS[i = 1 to n] {
  while (true) {
    ⟨turn[i] = max(turn[1:n]) + 1;⟩
    for [j = 1 to n st j != i]
      ⟨await (turn[j] == 0 or turn[i] < turn[j]);⟩
    critical section;
    turn[i] = 0;
    noncritical section;
  }
}
```

- Vsak proces dobi številko, ki je za ena večja od maksimalne številke procesa med vsemi procesi, ki čakajo na vstop v KP.
- V KP lahko vstopi tisti proces, ki ima najmanjšo številko med vsemi procesi, ki čakajo na vstop.
- Vse štiri lastnosti so izpolnjene.

Pekovski algoritem brez atomarnih ukazov

```
int turn[1:n] = ([n] 0);
process CS[i = 1 to n] {
  while (true) {
    turn[i] = 1; turn[i] = max(turn[1:n]) + 1;
    for [j = 1 to n st j != i]
      while (turn[j] != 0 and
                (turn[i],i) > (turn[j],j)) skip;
    critical section;
    turn[i] = 0;
    noncritical section;
  }
}
```

```
(a,b) > (c,d) == true,  če a > c ali če a == c in b > d
              == false, sicer
```

- Ker se **`max`** ne izvaja atomarno, pridobita lahko dva procesa enako številko. V tem primeru razvrstimo procesa s pomočjo zaporednega indeksa procesa.
- ANALOGIJA: dva kupca prideta hkrati k peku in zato dobita isto številko, vendar ju potem pek postreže glede na njuno letnico rojstva

Pekovski algoritem: primer 1

Zakaj?

```
int turn[1:n] = ([n] 0);
process CS[i = 1 to n] {
  while (true) {
    turn[i] = 1; turn[i] = max(turn[1:n]) + 1;
    for [j = 1 to n st j != i]
      while (turn[j] != 0 and
                (turn[i],i) > (turn[j],j)) skip;
    critical section;
    turn[i] = 0;
    noncritical section;
  }
}
```

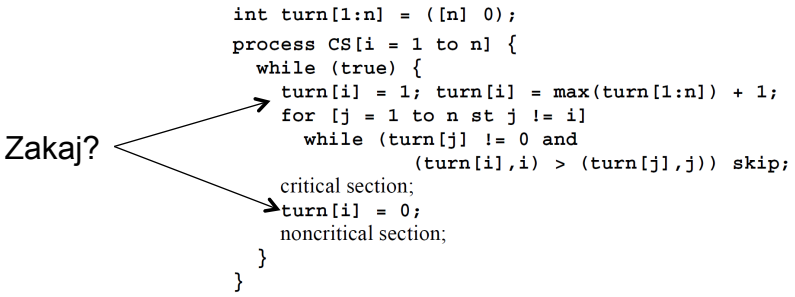
- Primer izvajanja dveh procesov brez **`turn[i]=1`**:

Proces CS1	**Proces CS2**
`turn[2] == 0`	
	`turn[1] == 0`
	`turn[2] = 1  //max + 1`
	`CS2 vstopi v KP //turn[1] == 0`
`turn[1] = 1  // max+1`	
`turn[2] == 1`	
`CS1 vstopi v KP // (1,1) < (1,2)`	

Oba procesa sta v KP.

Proces 2 je prehitel proces 1 (race condition).

Pekovski algoritem: primer 2



- Primer izvajanja dveh procesov če imamo **`turn[i]=1`**:

Proces CS1	**Proces CS2**
`turn[1] = 1  //NOVO!`	
`turn[2] == 0`	
	`turn[2] = 1  //NOVO!`
	`turn[1] == 1  // ne 0, kot prej`
	`turn[2] = 2  //max + 1`
	`CS2 NE vstopi v KP //turn[1] < turn[2]`
`turn[1] = 2  // max+1`	
`turn[2] == 2`	
`CS1 vstopi v KP // (2,1) < (2,2)`	

Proces CS1 vstopi v KP. Proces 2 je prehitel proces 1, toda ne more vstopiti v KP. Ko proces CS1 izstopi iz KP, postavi turn[1]=0, kar pomeni, da lahko CS2 vstopi v KP.

Ponovimo.

- Ključavnice v zanki:
  - učinkovite, če ni veliko procesov (problem sinhronizacije pomnilnika)
  - izvedene z atomarnimi strojnimi ukazi (ukaz TS, kombinacija TTS, lahko tudi FA)
  - ni zagotovljeno "pošteno" izvajanje procesov
    - lahko se zgodi, da je proces, ki je že v KP, gre takoj še enkrat v KP,
    - ni omogočen vstop v KP vsem procesom, ki čakajo na vstop v primeru šibke pravičnosti razvrščevalnika procesov

- Ključavnice, ki vodijo vrstni red vstopanja v KP:
  - v ključavnici poleg zaklepanja KP vodimo tudi vrstni red vstopanja v KP,
  - primerne so povsod tam, kjer hočemo imeti "pošten" (delno kontroliran) dostop do KP.
  - Algoritmi:
    - Petersonov,
    - postopek čakalnih listkov (potrebuje strojni ukaz FA za pošteno izvajanje)
    - pekovski algoritem

Primer: iskanje maksimalnega elementa v tabeli

- Vzporedni program z atomarnim ukazom:

```
int a[n];
int m = 0;
…
co [i = 0 to n-1]
  if (a[i] > m)
    < if (a[i] > m) m = a[i]; >
oc
```

- Izvedba programa s ključavnico z ukazom **`await`**

```
int a[n];
int m = 0;
bool lock = false;
…
co [i = 0 to n-1]
  if (a[i] > m)
    <await (!lock) lock = true;>
    if (a[i] > m) m = a[i];
    lock = false;
oc
```

- Izvedba programa s ključavnico v zanki s strojnim ukazom TS

```
int a[n];
int m = 0;
bool lock = false;
…
co [i = 0 to n-1]
  if (a[i] > m)
    while (TS(lock)) continue;
    if (a[i] > m) m = a[i];
    lock = false;
oc
```

Izvedba ukaza **`await`** s ključavnicami in KP

- Atomarnost izvajanja KP zagotovimo z medsebojnim izključevanjem procesov, ki KP izvajajo.

- To dosežemo z ustreznim mehanizmom vstopanja, izvajanja in izstopanja iz KP:

  `CSenter;`
  kritično področje;
  `CSexit;`

  - eden izmed mehanizimov: ključavnice

- Kako lahko ta mehanizem uporabimo za izvedbo ukazov
  - **`< S;>`**
  - **`< await(B); >`**
  - **`< await(B) S;>`**

  - Atomarne ukaze v < > obravnavamo kot KP.
  - Skupne spremenljivke v **`B`** in **`S`** zasčitimo s ključavnicami.

Izvedba ukaza **`await`** s ključavnicami in KP

Atomarni ukaz	Implementacija
`< S; >`	**CSenter;** **S;** **CSexit;**
`< await(B); >`	**CSenter;** **while (!B) {CSexit; Delay; CSenter;};** **CSexit;**
`< await(B) S; >`	**CSenter;** **while (!B) {CSexit; Delay; CSenter;};** **S;** **CSexit;**

V splošnem je lahko v ukazih v B in v S več skupnih spremenljivk.
Zakaj:
```
CSenter;
while (!B) continue;
S;
CSexit;
```

ni dovolj dobro? Kdaj pa je v redu?

Različne oblike sinhronizacije v KP

- Čakanje v delovanju *(ang. busy waiting)*
  - proces čaka na sinhronizacijo (na vstop v KP) v zanki, dokler ni izpolnjen pogoj za sinhronizacijo (za vstop v KP),
  - primer: ključavnice
  - Slabosti:
    - čakanje v zanki “krade” procesorski čas

- Čakanje z zaustavitvijo *(ang. blocking)*
  - proces, ki čaka na sinhronizacijo (na vstop v KP) je zaustavljen - ne dela nič
  - iz “spanja” ga “zbudi” drugi proces
  - Implementacija v OS.
  - Dva mehanizma:
    - pogojne spremenljivke (vgrajene v monitorje)
    - semaforji.

Točke usklajevanja pri iterativnih postopkih

- Pri iterativnih postopkih običajno izvajamo neke izračune nad podatki v več korakih: se pravi gremo prvič skozi podatke in izvajamo neke izračune (1. iteracija), gremo drugič skozi podatke, kjer uporabimo informacije iz prvega koraka (2. iteracija) itn. Torej imamo:
  - zanke po podatkih (v vsakem koraku gremo npr. po vseh podatkih)
  - zanke po času (koraki iteracije)
  - Paralelizacija: običajno zanke po podatkih, zanke po času pa ne, ker so npr. izračuni odvisni od prejšnjega koraka iteracije.

- Potreba po usklajevanju procesov trenutnega koraka, da lahko napredujemo v naslednji korak postopka, npr.:
  - razni numerični iterativni postopki, ki konvergirajo k neki iskani rešitvi:
    - izračunamo iz vseh vhodnih podatkov neke nove vrednosti, ki postanejo vhodni podatki za naslednji korak izračuna.
      V tem primeru: da napredujemo v naslednji korak iteracije, moramo počakati, da se vsi procesi, ki računajo nove podatke v trenutnem koraku, zaključijo.

  - Pravimo, da morajo procesi trenutnega koraka doseči neko točko, da se lahko program izvaja naprej.
    Take točke imenujmo točke usklajevanja oziroma pregrade programa.

Pregrade

- **Pregrada (ang. barrier)** je točka izvajanja programa, ki jo morajo doseči vsi procesi, da lahko program nadaljuje z izvajanjem.

```
bool done = false;
process P[i = 0 to n-1] {
  while (!done) {
    ukazi za izvedbo naloge i;
    Barrier(i, n)          // cakaj, dokler se
  }                        // vse naloge ne izvedejo
}
```

- Načini postavljanja pregrad
  - centraliziran način: pregrade s števci, zastavicami (koordinator)
  - decentraliziran način: v obliki drevesa, simetrične pregrade

- Načini izvedbe pregrad:
  - ključavnice, zastavice, števci – čakanje v delovanju
  - ključavnice s pogojnimi spr. – čakanje z zaustavitvijo
  - semaforji – čakanje z zaustavitvijo

Pregrada s števcem

- Uporabimo skupni števec, ki šteje koliko procesov je že prišlo do pregrade.

```
int count = 0;
Barrier(int n) {
  < count = count + 1; >   //nekdo je prisel do tu
  while (count < n);       //cakaj, da pridejo vsi
}
```

- To lahko izvedemo, če imamo strojni ukaz FA (fetch and add):

```
int count = 0;
Barrier(int n) {
  FA(count, 1);            //nekdo je prisel do tu
  while (count < n);       //cakaj, da pridejo vsi
}
```

- Strojni ukaz FA:

```
int FA (int var, int incr) {
  < int tmp = var;      // shrani zacetno vrednost spr. var
    var = var + incr;   // povecaj var za incr
    return tmp; >       // vrni zacetno vrednost spr. var
 }
```

Slabosti pregrade s števcem

- V vsakem koraku iteracije moramo postaviti števec na 0.
  - To lahko izvedemo v proceduri `Barrier` šele, ko vsi procesi prispejo do pregrade:
    - Vodimo dva števca, en je postavljen na začetku na 0, drugi pa na število procesov, prvi števec se povečuje, drugi zmanjšuje za ena. Ko vsi procesi prispejo do pregrade, zamenjamo vlogi števcev za naslednjo iteracijo.
- Vsi skupni števci se morajo povečevati (zmanjševati) atomarno. Potrebujemo posebne strojne ukaze, kot je npr. FA.
- Zgostitev uporabe pomnilnika (memory contention):
  - Vsakič ko števec povečamo za ena, morajo vsi procesi osvežiti svoje lokalne kopije spremenljivke `count` v predpomnilnikih. V najslabšem primeru to dela n-1 procesov.
    - Zato je tak način primeren, ko imamo malo procesov.

Princip signalizacije z zastavicami

- Imejmo zastavico:
  - zastavica signalizira nek dogodek
    - ko je zastavica dvignjena, se je dogodek zgodil (npr. prišli smo v pregrado)
    - ko je zastavica spuščena, se dogodek ni še zgodil.

- Signalizacija z zastavicami v pregradi:
  - Proces pride v pregrado in dvigne svojo zastavico. S tem signalizira ostalim procesom, da je prišel v pregrado.
  - Ko imajo vsi procesi dvignjene svoje zastavice (vsi so v pregradi), lahko nadaljujejo delo.

- Osnovni princip sinhronizacije z zastavicami:
  - Proces, ki čaka na drugi proces, da dvigne svojo zastavico (in mu s tem nekaj sporoči), lahko spusti temu procesu zastavico.
  - Zastavica ne sme biti dvignjena, če ni prej izpuščena.

- Primer uporabe ene zastavice dveh procesov:
  - proces 1 dvigne zastavico za proces 2 (odda signal),
    - `while (flag) continue; flag = 1;`
  - proces 2 jo lahko sam spusti (sprejme signal):
    - `while (!flag) continue; flag = 0;`

Uporaba zastavic v pregradi

- Ideja:
  - Uporabimo tabelo zastavic **`arrive[1:n]`**, ki označujejo prihode na pregrado (ena zastavica za vsak proces).
    - Vsak proces, ki pride do pregrade javi: Prišel sem. Zastavica prihoda procesa i se dvigne, torej **`arrive[i] = 1`**
  - Uporabimo tabelo zastavic **`continue[1:n]`**, ki označujejo, kdaj lahko proces nadaljuje z izvajanjem (ena zastavica za vsak proces).
    - Pogoj: vse zastavice v **`continue`** se postavijo na 1, če so vse zastavice v **`arrive`** že postavljene na 1.
    - Uvedemo dodaten proces – **koordinator**, ki čaka, da vse zastavice v **`arrive`** postanejo 1 in šele nato postavi zastavice v **`continue`** na 1.
  - Proces i nadaljuje z izvajanjem, ko **`continue[i] == 1`**.

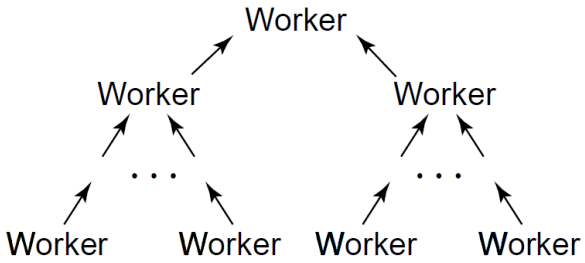
Pregrada z zastavicami in koordinatorjem

```
int arrive[1:n] = ([n] 0),  continue[1:n] = ([n] 0);
process Worker[i = 1 to n] {
  while (true) {
    code to implement task i;
    arrive[i] = 1;
    ⟨await (continue[i] == 1);⟩
    continue[i] = 0;
  }
}
process Coordinator {
  while (true) {
    for [i = 1 to n] {
      ⟨await (arrive[i] == 1);⟩
      arrive[i] = 0;
      }
    for [i = 1 to n] continue[i] = 1;
  }
}
```

Pomanjkljivosti pregrade s kordinatorjem

- Prva pomanjkljivost:
  - Dodaten proces (dodaten procesor) – koordinator, *2n* zastavic
- Druga pomanjkljivost:
  - običajno: delavci imajo približno enako dela in končajo v približno enakem času
  - koordinator pa v zanki čaka, da proces pride do pregrade (izguba časa)
- Izboljšava:
  - odpravimo koordinatorja in porazdelimo njegovo delo med ostale procese(delavce).

Organizacija dela procesov in pregrad v dvojiško drevo



- Vsako vozlišče v dvojiškem drevesu opravlja delo enega delavca – izvaja en proces.
- Delo koordinatorja je porazdeljeno med delavci – vozlišči drevesa.
  - Zastavice **`arrive`** se prenašajo po drevesu navzgor.
  - Zastavice **`continue`** se prenašajo po drevesu navzdol.
- Sinhronizacija:
  - delavec v vozlišču čaka, da zastavice **`arrive`** njegovih otrok postanejo 1, ko še sam zaključi z delom sporoči svojemu staršu, da je končal.
  - ko otroci korenskega vozlišča postavijo zastavice **`arrive`** na 1, to pomeni, da so vsi delavci končali z delom,
  - ko še korensko vozlišče zaključi s svojim delom, lahko sporoči svojim otrokom naj nadaljujejo z delom, ti potem to sporočijo še naprej navzdol po drevesu

Organizacija dela procesov in pregrad v dvojiško drevo

leaf node `L:`

```
arrive[L] = 1;
⟨await (continue[L] == 1);⟩
continue[L] = 0;
```

interior node `I:`

```
⟨await (arrive[left] == 1);⟩
arrive[left] = 0;
⟨await (arrive[right] == 1);⟩
arrive[right] = 0;
arrive[I] = 1;
⟨await (continue[I] == 1);⟩
continue[I] = 0;
continue[left] = 1; continue[right] = 1;
```

root node `R:`

```
⟨await (arrive[left] == 1);⟩
arrive[left] = 0;
⟨await (arrive[right] == 1);⟩
arrive[right] = 0;
continue[left] = 1; continue[right] = 1;
```

Lastnosti takšne porazdelitve dela

- Ni več koordinatorja – en proces manj.
- Število zastavic je enako: *2n*
- Čakanje na naslednji korak iteracije ni več O(n) kot v primeru koordinatorja, ampak O(log n).
- Imamo več komunikacije, kot v prejšnjem primeru. Signalov je več.
- Delo ni enakomerno porazdeljeno med vozlišči:
  - procesi v notranjih vozliščih opravljajo več dela, kot tisti v korenu ali listih.
  - **asimetrično porazdeljeno delo**.

Simetrična postavitev pregrad med procesi

- Ideja:
  - Definirajmo pregrado med dvema procesoma.
  - Uporabimo to pregrado za izvedbo pregrade med n procesi v več fazah, tako da na koncu pregledamo vse procese:
    - v vsaki fazi (koraku) izvajanja sinhronizacije pregledamo nekaj parov procesov,
    - v naslednji fazi pregledujemo druge pare, s tem da že upoštevamo sinhronizacijo s prve faze
    - itn.
  - Če bomo to počeli pametno: bodo na koncu vsi procesi med seboj sinhronizirani.
    - Vsak proces bo sinhroniziran v povprečju z log(n) procesi.
    - Potrebovali bomo log(n) faz za izvedbo.
- To je princip simetrične pregrade.
  - v vsakem procesu opravimo enako dela

Pregrada med dvema procesoma

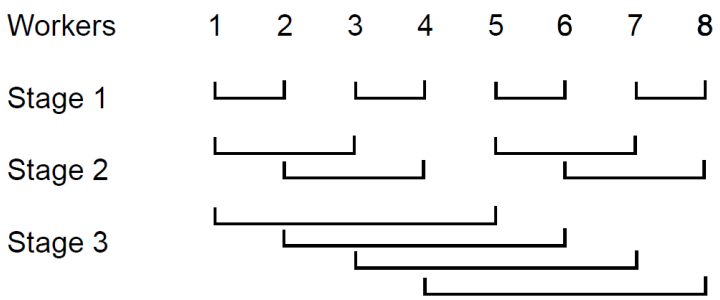
- Procesa i in j naznanita končanje svojega dela, tako da postavita svoji zastavici v arrive[i] in arrive[j] na 1. Proces i (j) potem čaka, da je zastavica procesa j (i), arrive[j]==1 (arrive[i]==1), in ko je, jo brž postavi nazaj na 0, torej arrive[j] = 0 (arrive[i]=0).

```
/* barrier code for worker process W[i] */
⟨await (arrive[i] == 0);⟩  /* key line -- see text */
arrive[i] = 1;
⟨await (arrive[j] == 1);⟩
arrive[j] = 0;

/* barrier code for worker process W[j] */
⟨await (arrive[j] == 0);⟩  /* key line -- see text */
arrive[j] = 1;
⟨await (arrive[i] == 1);⟩
arrive[i] = 0;
```

- Prva vrstica je potrebna, da vemo, da se je zaključilo izvajanje prejšnje faze. Zadnja vrstica pa postavi zastavico na 0 za naslednjo fazo.

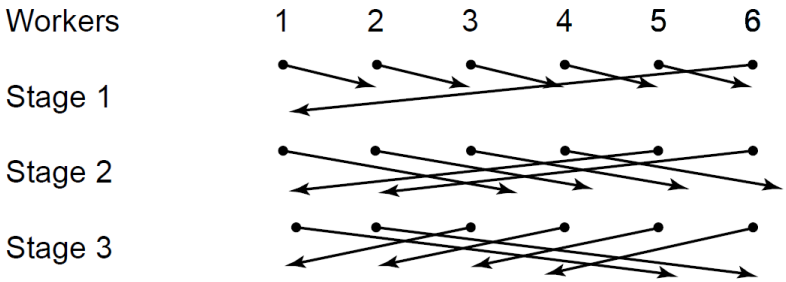
Simetrična pregrada v obliki metulja

- Primerna je, če je število procesov (n) potenca števila 2.
- Število faz: $\log_2 n$
  - v fazi k, se primerjajo vsi procesi, ki so narazen za 2^{k-1} glede na zaporedno številko procesa,
  - vsak proces se sinhronizira z $\log_2 n$ drugimi procesi



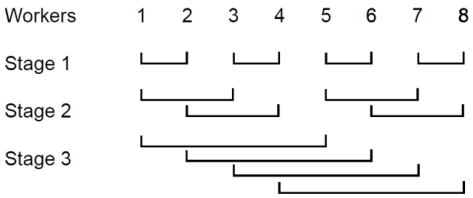
Simetrična pregrada z razširjanjem

- Primerna za poljubno število procesov.
- Število faz: $\lceil \log_2 n \rceil$
  - v fazi *k* je izvedena pregrada procesa *i* s procesom, ki je oddaljen za $(i+2^{k-1}) \bmod n$,
  - Proces v vsaki fazi širi informacijo o svojem prihodu v pregrado procesom na desni:
    - procesu najprej dvigne svojo zastavico v pregradi in nato čaka, da je njemu dvignjena zastavica procesa na desni, da jo lahko spusti.
  - Vsak proces se sinhronizira z $\lceil \log_2 n \rceil$ drugimi.



Problem prehitevanja

- Problem prehitevanja se pojavi, ker hkrati izvajamo več sinhronizacij med dvema procesoma v več fazah. Zgodi se lahko smrtni objem.

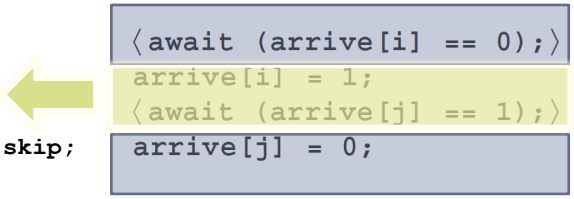


- Poglejmo primer izvedbe pregrade v obliki metulja:
  - Denimo, da se v 1. fazi procesa 3 in 4 sinhronizirata.
  - Proces 1 pa v 1. fazi čaka na proces 2 (ta zamuja). proces 1 torej svojo zastavico arrive[1] postavi na 1 in čaka, da bo arrive[2] postala 1.
  - V 2. fazi proces 3 nadaljuje svojo sinhronizacijo s procesom 1. Čaka, da bo arrive[1] na 1. To pa že je, zato postavi arrive[1] na 0 in nadaljuje svojo sinhronizacijo s procesom 7 v 3. fazi.
  - **Smrtni objem:** proces 3 se je sinhroniziral s procesom 1 in zapustil pregrado, medtem pa proces 2 čaka na vstop v pregrado s procesom 1, kar se ne bo nikoli zgodilo

Kako to preprečiti?

- Prva ideja:
  - Uporabimo različne zastavice za vsako fazo sinhronizacije.
  - Prezahtevno za usklajevanje. Preveč zastavic.
- Druga ideja:
  - vsaka zastavica naj vodi evidenco, v kateri fazi izvajanja sinhronizacije smo.
  - zastavice naj ne bodo binarne vrednosti (0,1), ampak naj štejejo, v kateri fazi izvajanja smo,
  - mimo pregrade gremo lahko, ko:

```
# pregrada za proces i
for [s = 0 to num_stages] {
  arrive[i] = arrive[i] + 1;
  # cakamo, na soseda j
  while (arrive[j] < arrive[i]) skip;
}
```



Oblike paralelizmov

- Podatkovni paralelizem
  - paralelizacija postopka glede na število podatkov, ki se jih da obdelati istočasno.
  - Podatke lahko razdelimo na skupine, podpodročja, na katerih lahko izvajamo posamezne naloge (enake operacije) in na ta način rešimo nalogo.
    - primer: deli in vladaj
  - potrebna je sinhronizacija s pregradami
  - v primeru skupnih spremenljivk sinhronizacija KP s ključavnicami ali čim podobnim.
- Postopkovni paralelizem
  - paralelizacija postopka glede na število opravil (operacij), ki se jih da opraviti istočasno.
    - primer: pipe v Unix-u: končni rezultat je rezultat zaporedja opravil, ki jih izvajamo nad vhodnimi podatki
  - potrebna je sinhronizacija KP: pisanje predhodnega opravila v skupni pomnilnik, ki ga potrebuje (bere) naslednje opravilo za svoje izvajanje
- Kombinacija obeh
  - gre za kombinacijo obeh paralelizmov
  - imenujemo ga tudi asinhroni paralelizem
  - potrebna je sinhronizacija s pregradami in sinhronizacija KP.

Primer podatkovnega paralelizma: delne vsote števil v tabeli

- Naloga:
  - Imamo tabelo števil `a[0:n-1]`. Izračunaj vse delne vsote števil v tabeli. Delna vsota števil do indeksa `i` je vsota vseh števil v tabeli `a[0 : i].`

- Sekvenčni program:

```
sum[0] = a[0];
for [i = 1 to n-1]
  sum[i] = sum[i-1] + a[i];
```

Primer podatkovnega paralelizma: delne vsote števil v tabeli

- Kako paralelizirati dani problem?
- Ideja:
  - Seštevamo sočasno pare števil v tabeli:
    - npr. sosednje pare števil: a[0]+a[1], a[2]+a[3], …
  - Sočasno seštevanje parov nadaljujemo tudi v naslednjih korakih:
    - npr. lahko bi nadaljevali tako da vsoti (a[0]+a[1]) dodamo vsoto (a[2]+a[3]), itn. Princip deli in vladaj. Vendar tako ne dobimo vseh delnih vsot števil, čeprav zmanjšamo število korakov računanja na O(log n).
    - zato:
      - v prvem koraku, sum[i] = a[i] za vse i
      - drugi korak: seštejemo vse pare delnih vsot, ki so za ena narazen sum[i] + sum[i-1] za vse i>=1
      - tretji korak: seštejemo vse pare delnih vsot, ki so za podvojeno razdaljo od prejšnjega koraka (dva) narazen sum[i] + sum[i-2] za vse i>=2
      - …
      - po $\lceil \log_2 n \rceil$ korakih dobimo vse delne vsote.

Primer:

začetne vrednosti a	1	2	3	4	5	6
sum po prvem koraku	1	3	5	7	9	11
sum po drugem koraku	1	3	6	10	14	18
sum po tretjem koraku	1	3	6	10	15	21

Primer podatkovnega paralelizma: delne vsote števil v tabeli

```
int a[n], sum[n], old[n];
process Sum[i = 0 to n-1] {
  int d = 1;
  sum[i] = a[i];   /* initialize elements of sum */
  barrier(i);
  ## SUM:  sum[i] = (a[i-d+1] + ... + a[i])
  while (d < n) {
    old[i] = sum[i];   /* save old value */
    barrier(i);
    if ((i-d) >= 0)
      sum[i] = old[i-d] + sum[i];
    barrier(i);
    d = d+d;   /* double the distance */
  }
}
```

- **`barrier(i)`:**
  - pregrada, ki ima za argument zaporedno številko procesa,
  - proces se vrne iz pregrade, ko pregrado dosežejo vsi procesi,
  - uporabimo enega izmed algoritmov za izvedbo pregrad,
  - dve pregradi:
    - prvo potrebujemo, ker morajo vsi procesi zapisati vrednost vsote do tega koraka kot staro vsoto
    - drugo potrebujemo, ker moramo pridobiti vse delne vsote, preden povečamo razdaljo za naslednji korak.

Podatkovni paralelizem: sočasni izračuni nad geometrijsko urejenimi podatkih

- Geometrijsko urejeni podatki so podatki, ki so organizirani v neko geometrijsko strukturo:
  - npr. predstavljajo mrežo točk.
- Problem:
  - rešitev problema mora biti izračunana v vsaki točki.
  - Običajno v več iteracijah, kjer se v vsaki iteraciji ponovi isti postopek na novo izračunanih podatkih (do končne rešitve ali konvergence k rešitvi).
  - Takšne postopke lahko paraleliziramo.
- Formulizacija problema:

```
inicializacija;
while (pogoj za prekinitev)
   izračunaj nove vrednosti na vsaki točki;
```

- Možnost podatkovnega paralelizma:
  - podatke razdelimo na manjše skupine (bloke, pasove, trakove, odseke);
  - vsak delavec obdeluje posamezne skupine podatkov.

Reševanje PDE: Laplaceva diferencialna enačba

- Laplaceva diferencialna enačba z Dirichletovimi robnimi pogoji je v 2D definirana kot:

$$\frac{\partial^2 \phi}{\partial^2 x} + \frac{\partial^2 \phi}{\partial^2 y} = 0$$

  - ϕ predstavlja neki neznano količino, ki se širi po površini, npr. temperatura
  - vrednosti na robovih površine so konstantne

- Naloga je izračunati porazdelitev te količine po notranjosti površine v nekem časovnem obdobju (oziroma dokler se porazdelitev ne spreminja več).
  - predpostavke: 2D površina, iščemo (aproksimirano) numerično rešitev

- Paralelizacija:
  - uporabimo eno izmed iterativnih metod reševanja PDE, npr. Jakobijevo iterativno metodo,
  - mrežo 2D točk področja razdelimo na podpodročja, npr. trakove, kvadrate in za vsako takšno podpodročje uporabimo svojega delavca (proces),
  - izvedemo sočasni izračun za vsako podpodročje.

Paralelizacija Jakobijeve iterativne metode

- Jakobijeva iterativna metoda:
  - nova vrednost v vsaki točki je izračunana kot povprečje starih vrednosti sosednjih štirih točk

$$\phi_{i,j}^{k} = \frac{1}{4}(\phi_{i-1,j}^{k-1} + \phi_{i+1,j}^{k-1} + \phi_{i,j-1}^{k-1} + \phi_{i,j+1}^{k-1})$$

  - postopek izvajamo toliko časa, dokler se nove in stare vrednosti razlikujejo za manj kot ε.
- Vzporedni program:

```
real grid[n+1,n+1], newgrid[n+1,n+1];
bool converged = false;
process Grid[i = 1 to n, j = 1 to n] {
  while (not converged) {
    newgrid[i,j] = (grid[i-1,j] + grid[i+1,j] +
                    grid[i,j-1] + grid[i,j+1]) / 4;
    check for convergence
    barrier(i);
    grid[i,j] = newgrid[i,j];
    barrier(i);
  }
}
```

Princip vreče opravili

- Podatkovni paralelizem: deli in vladaj
  - razdeli podatke na manjše dele, na njih opravi neko delo, končna rešitev je kombinacija rešitve teh opravil
  - primerni za rekurzivne postopke,
  - paralelizacija: sočasno izvajanje opravil na razdeljenih podatkih

- Alternativa: princip vreče opravil
  - procesi si delijo vrečo z opravili,
  - opravil je več, kot je predvidenih procesov:
    - na začetku vsi procesi dobijo svoje opravilo, ko končajo z delom, vzamejo novo opravilo
    - to se nadaljuje, dokler je še kaj opravil v vreči.

```
bool done = false;
shared bag of tasks;
process Worker[w = 1 to P] {
  while (bag of tasks is not empty or !done) {
        <get a task from the bag;>
        execute the task;
        possibly generate new task(s) and <put it to the bag;>
  }
}
```

  - to je oblika podatkovnega oziroma kombiniranega paralelizma

Princip vreče opravil

- Primerni v primeru, ko predvidevamo fiksno število procesov.

- Enakomerno porazdeljevanje dela med različne procese.

- Ker imamo skupno vrečo, je ta deljena med procesi in zato moramo uporabiti eno izmed metod za zagotavljanje medsebojnega izključevanja:
  - ključavnica (s katero zaklenemo/odklenemo vrečo opravil)

- Kdaj se program konča?
  - ko je vreča opravil prazna in so vsa opravila končana,
  - opravila so končana, ko vsi procesi (delavci) čakajo na naslednje opravilo.

- Primer:
  - paralelno množenje matrik

Primer: množenje matrik po principu vreče opravil

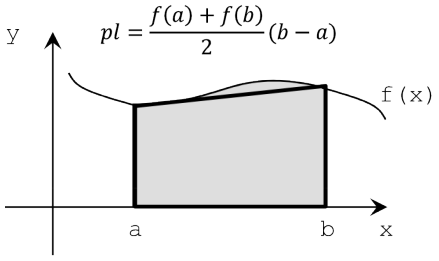
- Število procesov (= številu procesorjev) << n
- Vreča opravil

  `<row = nextrow++;>`
- Vzemi novo opravilo:
  - lahko izvedemo s strojnim atomarnim ukazom FA
  - vreča je prazna, ko je `row >= n`
- Konec programa:
  - ko vsi delavci skočijo ven iz zanke,
- Pregrada:
  - za zanko (vse vrstice so izračunane)
  - za pregrado lahko sledi izpis matrike `c` (v enem izmed delavcev)
  - pregrada je lahko izvedena s števcem `done`

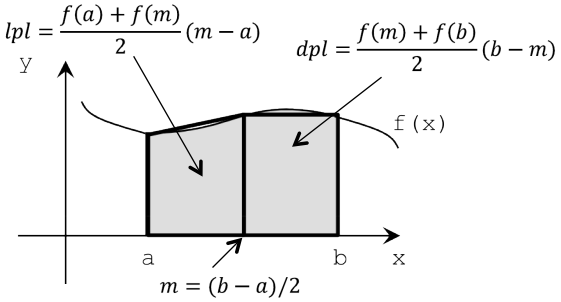
```
int nextRow = 0; # the bag of tasks
double a[n,n], b[n,n], c[n,n];

process Worker[w = 1 to P] {
  int row;
  double sum;  # for inner products
  while (true) {
    # get a task
    ⟨ row = nextRow; nextRow++; ⟩
    if (row >= n)
      break;
    compute inner products for c[row,*];
  }
}

int done = 0;
<done++>;
if (done == n)
  print matrix c;
```

Vzporedni izračun določenega integrala

- Spomnimo se rekurzivnega postopka:



prva aproksimacija
pl



druga aproksimacija
lpl + rpl

- Rekurzija:
  - na danem intervalu izračunamo ploščino `pl`
  - interval razdelimo na dve polovici in izračunamo levo in desno ploščino, `lpl` in `rpl`
  - rekurzijo ustavimo, ko `|(lpl+rpl) - pl|< eps`

Vzporedni izračun določenega integrala po principu vreče opravil

- Vreča:
  - seznam peteric (a, b, f(a), f(b), area)
- Postopek
  - vzamemo eno peterico iz vreče
  - izračunamo ploščino
  - vrnemo dve peterici (levo in desno stran) nazaj v vrečo
  - ponavljamo dokler se vsota leve in desne ploščine ne razlikuje od skupne ploščine za manj kot eps.
- Posebnost:
  - vreča je lahko tudi prazna, pa moramo še vedno čakati na izvajanje (v primerjavi s prejšnjim izračunom produkta matrik, ko smo končali, ko se je vreča izpraznila)
  - še vedno moramo igrati po principu vreče opravil: program se zaključi, ko v vreči ni opravil in vsi delavci čakajo na nova opravila.

Vzporedni izračun določenega integrala po principu vreče opravil

- Vreča je predstavljena kot vrsta peteric.
- Princip vreče opravil:
  - program se konča, ko je
    - **size = 0** in
    - **idle = n** (spr. idle šteje, koliko delavcev počiva)
- Imamo več atomarnih ukazov zaradi skupnih spr. Lahko jih izvedemo s ključavnicami, ker nimamo pogojnih atomarnih ukazov, ampak samo KP.
- Razen enega:
  - tega bomo reševali v nadaljevanju s semaforji ali monitorji.

```
type task = (double left, right, fleft, fright, lrarea);
queue bag(task);      # the bag of tasks
int size;             # number of tasks in bag
int idle = 0;         # number of idle workers
double total = 0.0;   # the total area

compute approximate area from a to b;
insert task (a, b, f(a), f(b), area) in the bag;
count = 1;

process Worker[w = 1 to PR] {
  double left, right, fleft, fright, lrarea;
  double mid, fmid, larea, rarea;
  while (true) {
    # check for termination
    ⟨ idle++;
      if (idle == n && size == 0) break; ⟩
    # get a task from the bag
    ⟨ await (size > 0)
      remove a task from the bag;
      size--; idle--; ⟩
    mid = (left+right) / 2;
    fmid = f(mid);
    larea = (fleft+fmid) * (mid-left) / 2;
    rarea = (fmid+fright) * (right-mid) / 2;
    if (abs((larea+rarea) - lrarea) > EPSILON) {
      ⟨ put (left, mid, fleft, fmid, larea) in the bag;
        put (mid, right, fmid, fright, rarea) in the bag;
        size = size + 2; ⟩
    } else
      ⟨ total = total + lrarea; ⟩
  }
  if (w == 1)   # worker 1 prints the result
    printf("the total is %f\n", total);
}
```

4 Semaforji

V poglavju so predstavljeni semaforji.

Poglavje obravnava:

- definicija semaforja,
- oblike semaforjev: binarni, sestavljeni binarni, splošni
- sinhronizacija z binarnimi semaforji:
  - izvedba dostopanja do kritičnih področij,
  - izvedba pregrad,
  - izvedba pogojne sinhronizacije, reševanje po principu podajanja štafete.
- sinhronizacija s sestavljenimi binarnimi semaforji:
  - izvedba problema proizvajalec/porabnik.
- sinhronizacija s splošnimi semaforji:
  - izvedba sinhronizacije pri uporabi medpomnilnika z omejenim prostorom.
- izvedba selektivnega medsebojnega izključevanja procesov s semaforji:
  - problem petih filozofov,
  - problem branja in pisanja deljenih podatkov.
- nekaj praktičnih primerov uporabe semaforjev:
  - problem proizvajalca/porabnika z uporabo opearacij `broadcast` in `listen`,
  - implementacija opearacij `sleep` in `wakeup`.

Uvedba semaforjev

- Semaforji na železnici:
  - rdeča luč pomeni, da je na progi pred nami drugi vlak, moramo se ustaviti,
  - zelena luč: proga pred nami je prosta, lahko nadaljujemo z vožnjo,
  - ko zapeljemo na progo za semaforjem, se na semaforju prižge rdeča luč (proga je zasedena) in ostane prižgana dokler je naš vlak na tem kritičnem odseku proge,
  - ko vlak zapusti kritično področje (!), se spet prižge zelena luč
  - semaforji so postavljeni, da z njimi preprečimo nesreče (oba vlaka v kritičnem področju)

- Semaforji predstavljajo mehanizem za **pogojno sinhronizacijo** (rdeča, zelena luč) med procesi (vlaki) z namenom zagotavljanja **medsebojnega izključevanja dostopanja procesov do kritičnih področji** (preprečimo nesreče v kritičnem odseku proge).

- Zgodovina uvedbe semaforjev za sinhronizacijo med procesi:
  - Edsger Dijkstra (nizozemski znanstvenik) 1968,
  - izvedba v operacijskem sistemu THE (Technological University of Eindhoven) ,
  - definicija dveh operaciji nad semaforji:
    - P iz nizozemščine “Proberen” – poskušaj
      oziroma “Passeren” – pelji mimo (semaforja v KP)
    - V iz nizozemščine “Verhogen” – povečaj
      oziroma “Vrijgeven” – sprosti (KP)

Definicija semaforja

- Semafor je podatkovna struktura, ki vključuje eno celoštevilsko spremenljivko **`s`**, nad katero sta definirani dve (atomarni) operaciji:
  - poskusi vstopiti (v KP) `P(s):  < await (s > 0) s = s – 1; >`
  - sprosti (KP) `V(s):  < s = s + 1; >`

- Vrednost s je vedno **nenegativna**.

- Deklaracija in inicializacija:
  - `sem s;  #semafor, z vrednostjo s = 0;`
  - `sem s = expr;  #semafor, z vrednostjo s = expr;`
  - `sem s[1:n] ([n] expr) #tabela semaforjev s[n] = expr;`

Implementacija semaforja

- Kako implementirati operaciji P in V v semaforju?
- Ukaz **`await`** lahko implementiramo s čakanjem v delovanju (npr. ključavnice) ali pa s čakanjem z zaustavitvijo (npr. pogojne spr.). Pri semaforjih uporabimo implementacijo čakanja z zaustavitvijo procesa. Zato uporabljamo vrsto FIFO.

- Implementacija operacije P(s):

```
if (s > 0) s = s-1;
else {
   Zaustavi proces in ga postavi v vrsto za čakanje sem_wait_queue.
}
```

- Implementacija operacije V(s):

```
if ( empty(sem_wait_queue) ) s = s+1;
else {
   Vzemi prvi proces iz sem_wait_queue in
   ga postavi  v vrsto za izvajanje ready_queue.
}
```

Oblike semaforjev

- Binarni semafor
  - vrednost semaforja je lahko samo 0 ali 1
  - uporaba:
    - medsebojno izključevanje dostopanja do kritičnih področij
    - pogojna sinhronizacija (podobno kot pri zastavicah)

- Sestavljeni binarni semafor
  - imamo več binarnih semaforjev,
  - v določenem trenutku je lahko samo en izmed semaforjev 1, vsi ostali so 0,
  - torej vsota vrednosti semaforjev je 0 ali 1.
  - uporaba:
    - za medsebojno izključevanje in pogojno sinhronizacijo,
    - primer: proizvajalec/porabnik, uporaba omejenega medpomnilnika

- (Splošni) Števni semafor
  - vrednost semaforja je lahko vsako nenegativna celo število
  - uporaba:
    - štejemo, kolikokrat dostopamo do KP, koliko resursov imamo že porabljenih ipd.
    - za pogojno sinhronizacijo

Kritična področja in binarni semaforji - MUTEXi

- Kritično področje je definirano z < S; >.
- Namesto <> oklepajev lahko uporabimo binarni semafor in oklepaje nadomestimo z operacijama P in V.
- Tak binarni semafor imenujemo mutex, ker z njim zagotovimo medsebojno izključevanje dostopanja procesov do KP.

Semafor mutex je na začetku postavljen na 1, kar pomeni, da je dostop v KP dovoljen.

```
sem mutex = 1;

process CS[i = 1 to n] {
  while (true) {
    P(mutex);
    critical section;
    V(mutex);
    noncritical section;
  }
}
```

Uporaba semaforjev za signalizacijo

- Binarne semaforje lahko uporabimo tudi za komunikacijo med procesi: signaliziramo stanje določenega dogodka oziroma stanje določenega pogoja:
  - običajno pomeni vrednost semaforja 0, da se dogodek še ni zgodil, oziroma, da pogoj ni izpolnjen
  - Proces signalizira (sporoča), da se je dogodek izvršil, z operacijo V – operacija V "pošlje" signal (vsem ostalim procesom)
  - Proces čaka na signal (na sporočilo), da se dogodek izvrši, v operaciji P – proces "sprejema" signale v operaciji P.
- Uporaba: pregrade
  - pregrado med dvema procesoma lahko izvedemo z binarnima semaforjema, ki sporočata prihod do pregrade in kontrolirata izstop iz pregrade.
  - Posplošitev na n-procesov?

```
sem arrive1 = 0, arrive2 = 0;
process Worker1 {
  ...
  V(arrive1);    /* signal arrival         */
  P(arrive2);    /* wait for other process */
  ...
}
process Worker2 {
  ...
  V(arrive2);    /* signal arrival         */
  P(arrive1);    /* wait for other process */
  ...
}
```

Pregrada s koordinatorjem

Pretvori pregrado z zastavicami v pregrado z binarnimi semaforji:

```
int arrive[1:n] = ([n] 0),  continue[1:n] = ([n] 0);
process Worker[i = 1 to n] {
  while (true) {
    code to implement task i;
    arrive[i] = 1;
    ⟨await (continue[i] == 1);⟩
    continue[i] = 0;
  }
}
process Coordinator {
  while (true) {
    for [i = 1 to n] {
      ⟨await (arrive[i] == 1);⟩
      arrive[i] = 0;
      }
    for [i = 1 to n] continue[i] = 1;
  }
}
```

Pregrada v dvojiškem drevesu

Pretvori pregrado z zastavicami v pregrado z binarnimi semaforji:

leaf node `L:`

```
arrive[L] = 1;
⟨await (continue[L] == 1);⟩
continue[L] = 0;
```

interior node `I:`

```
⟨await (arrive[left] == 1);⟩
arrive[left] = 0;
⟨await (arrive[right] == 1);⟩
arrive[right] = 0;
arrive[I] = 1;
⟨await (continue[I] == 1);⟩
continue[I] = 0;
continue[left] = 1; continue[right] = 1;
```

root node `R:`

```
⟨await (arrive[left] == 1);⟩
arrive[left] = 0;
⟨await (arrive[right] == 1);⟩
arrive[right] = 0;
continue[left] = 1; continue[right] = 1;
```

Uporaba binarnih semaforjev za sinhronizacijo

- Medsebojno izključevanje dostopanja do KP **<S;>**
  - Uporaba semaforja MUTEX:

```
sem mutex = 1;   // pogoj za vstop v KP
P(mutex);
S;
V(mutex);
```

- Pogojna sinhronizacija **<await(B) S;>**
  - Uporabimo dva semaforja: semafor entry za dostop do KP in semafor za signalizacijo, ki čaka, da je pogoj B izpolnjen.

```
sem entry = 1;   // pogoj za vstop v KP
sem cond = 0;    // semafor za pogoj
P(entry);
while (!B) {V(entry); P(cond); P(entry);}
S;
V(entry);
```

  - Postopek lahko še izboljšamo z metodo “podajanja štafete”.

Sinhronizacija z metodo “podajanja štafete”

- Poglejmo naslednji primer:
  - Imamo proces W, ki čaka za vstop v KP, dokler se pogoj **B** ne izpolni.

```
W: P(entry); while (!B) {V(entry); P(cond); P(entry);}
```

    - Za nadaljevanje dela procesa W potrebujemo dva semaforja, **cond** in **entry**

  - Imamo proces S, ki postavi **B = true** in to tudi “signalizira”.

```
S: P(entry); B = true; V(cond); V(entry);
```

    - Proces S uporablja semafor **cond** za signalizacijo, da je **B** izpolnjen, semafor **entry** pa za dostop do KP. Vrednosti obeh semaforjev se spremenijo!

  - Imamo podvojitev **P(entry)** ! Tako lahko proces **S** poda semafor **entry** (dovolilo za vstop v KP) procesu **W** preko semaforja **cond**, ki signalizira, da je pogoj **B** izpolnjen.

- Metoda “podajanja štafete”: štafeta je v tem primeru dovolilo za vstop v KP.

```
W: P(entry); while (!B) {V(entry); P(cond); }
S: P(entry); B = true; V(cond);
```

Metoda podajanja štafete na primeru

```
sem entry = 1,/* controls entry to CS */
    cond = 0; /* to signal the condition */
int dp = 0;   /* counts delayed processes */
process W[i = 0 to n - 1] {
   …
   P(entry);
   if (!B) {
      dp++;
      V(entry);
      P(cond); /* delays on cond */
   }
   S;
   if (dp > 0) {
      dp--;
      V(cond); /* pass the "entry baton" */
   } else V(entry);
   …
}
```

```
process S {
   …
   P(entry);
   change B to true;
   if (dp > 0) {
        dp--;
        V(cond);
   } else V(entry);
   …
}
```

Sestavljeni binarni semaforji

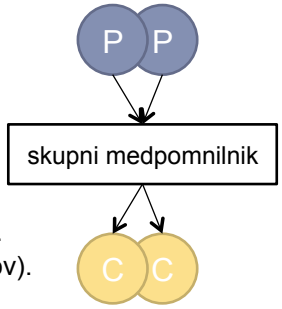
- Sestavljeni binarni semafor je sestavljen iz več binarnih semaforjev, kjer ima lahko v nekem trenutku samo en izmed semaforjev vrednost 1, ostali imajo vrednost 0.

$$0 \leq s_0 + s_1 + \ \ldots + s_{n-1} \leq 1$$

  - omogočajo izvedbo medsebojno izključevanje dostopanja do KP in signalizacijo za pogojno sinhronizacijo več procesov
  - uporabno v primerih, ko nadziramo (sporočamo) stanje spremenljivk (ali podatk. struktur), ki izmenično spreminjajo svoje vrednosti:
    - Npr.: dva binarna semaforja **`full`** in **`empty`** lahko tvorita sestavljeni binarni semafor, s katerim lahko sporočamo ali je neka podatk. struktura (npr. buffer) polna ali prazna.

Primer uporabe sestavljenega binarnega semaforja

- Problem proizvajalca/porabnika: sinhronizacija skupnega medpomnilnika:
  - proizvajalec (P) polni medpolnilnik, porabnik (C) prazni medpomnilnik,
  - proizvajalcev in porabnikov je lahko več,
  - Poenostavljen problem, ko je skupni medpomnilnik ena sama deljena skupna spr.:
    - Spr. je na začetku prazna (brez podatkov).
    - Proizvajalec mora čakati, dokler ni spr. prazna (brez podatkov).
    - Porabnik mora čakati, dokler spr. ni polna (polna novih podatkov).



  - Rešitev:
    - zagotovimo medsebojno izključevanje pisanja in branja skupne spr.
    - potrebna je še pogojna sinhronizacija:
      - C čaka na nove podatke v spr., čaka dokler ni novih podatkov v spr., čaka, da se spr. napolni
      - P čaka, da se sprosti prostor v spr., čaka, da se spr. sprazni.
    - **Uporabimo sestavljeni binarni semafor za pogojno sinhronizacijo in medsebojno izključevanje branja in pisanja v spr.**

Problem proizvajalca/porabnika – rešitev z binarnimi semaforji

- Vpeljemo dva binarna semaforja **`full`** in **`empty`**, ki tvorita sestavljeni binarni semafor.
- Samo eden izmed njiju je lahko 1:
  `0 <= full + empty <= 1`
- Če proizvajalec hoče pisati v medpomnilnik, mora čakati, da postane **`empty==1`**. Ko zapiše v **`buf`**, postavi **`full=1`**.
- Porabnik čaka, da je **`full==1`**, da lahko prebere iz **`buf`**. Ko prebere, postavi **`empty=1`**.

```
typeT buf;      /* a buffer of some type T */
sem empty = 1, full = 0;
process Producer[i = 1 to M] {
  while (true) {
    ...
    /* produce data, then deposit it in the buffer */
    P(empty);
    buf = data;
    V(full);
  }
}
process Consumer[j = 1 to N] {
  while (true) {
    /* fetch result, then consume it */
    P(full);
    result = buf;
    V(empty);
    ...
  }
}
```

Lastnost sestavljenih binarnih semaforjev

- Predpostavimo,
  - da procesi uporabljajo sestavljen binarni semafor.
  - da ima vsak proces neko zaporedje ukazov med operacijama P in V na (različnih) binarnih semaforjih, ki sestavljajo sestavljeni binarni semafor.

- Potem je zagotovljeno medsebojno izključevanje izvajanja ukazov med operacijama P in V v vsakem procesu.
  - Utemeljitev:
    - Po definiciji je $0 \leq s_0 + s_1 + \ ... + s_{n-1} \leq 1$
    - Opažanje: Vsakič ko izvajamo ukaze v področju med P in V so vsi semaforji postavljeni na 0.
    - Kar pomeni, da ko ima en proces izpolnjen pogoj v P, lahko nadaljuje z izvajanjem KP, medtem pa vsi ostali procesi nimajo izpolnjen pogoj v P in zato čakajo na izvajanje KP, dokler se ne izvede operacija V.

Splošni semaforji

- Spremenljivka v semaforju lahko zavzame katerokoli nenegativno celoštevilsko vrednost:

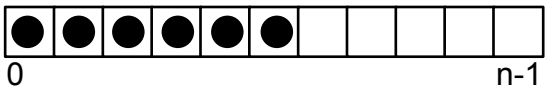
  - semafor je inicializiran na neko začetno vrednost,
  - potem pa se števec povečuje oziroma zmanjšuje,

  - s števcem lahko štejemo različne stvari ali dogodke:
    - koliko enot prostora imamo še na razpolago v (med)pomnilniku,
    - koliko enot podatkov je v podatkovni bazi,
    - kolikokrat smo bili v KP

  - izvedba sinhronizacije s splošnimi semaforji:
    - pogojna sinhronizacija:
      - npr. čakaj, dokler ni dovolj (enot) prostora v medpomnilniku na voljo, potem pa jo lahko uporabiš.

Uporaba medpomnilnika z omejenim prostorom

- Pri problemu proizvajalca/porabnika smo zapisovali v medpomnilnik, ki je bil velik eno enoto. Kaj pa če je velik n enot?

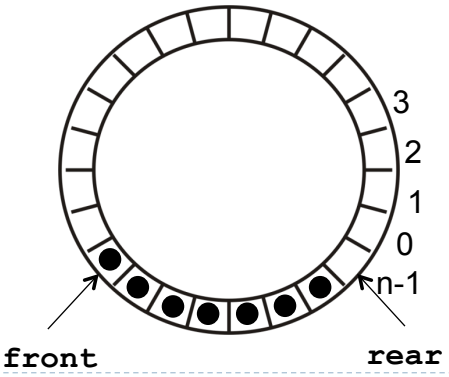


- Medpomnilnik velikosti **n** vodimo kot tabelo tipa **TypeT**

```
TypeT buf[n];
int front = 0; rear = 0;
```

- Organiziramo kot krožno tabelo (krožni medpomnilnik)
  - **rear** kaže na prvo prazno mesto v tabeli
  - **front** kaže na prvo polno mesto v tabeli

```
vstavi:     buf[rear] = data;
(deposit)   rear = (rear+1) % n;

vzemi:      result = buf[front];
(fetch)     front = (front+1) % n;
```



Izvedba proizvajalec/porabnik z omejenim medpomnilnikom

- Sinhronizacija dostopa do medpomnilnika:
  - proizvajalec mora čakati, dokler medpomnilnik ni prazen (prazen je takrat, ko je v njem še kaj prostora za pisanje)
  - porabnik mora čakati, dokler medpomnilnik ne vsebuje vsaj enega podatka za branje
- Uporabimo dva splošna semaforja:

```
sem empty = n;
sem full = 0;
```

  - **empty** šteje prazna mesta
  - **full** šteje polna mesta

```
typeT buf[n];       /* an array of some type T */
int front = 0, rear = 0;
sem empty = n, full = 0;    /* n-2 <= empty+full <= n */
process Producer {
  while (true) {
    ...
    produce message data and deposit it in the buffer;
    P(empty);
    buf[rear] = data; rear = (rear+1) % n;
    V(full);
  }
}
process Consumer {
  while (true) {
    fetch message result and consume it;
    P(full);
    result = buf[front]; front = (front+1) % n;
    V(empty);
    ...
  }
}
```

Uporaba omejenega medpomnilnika med več proizvajalci in več porabniki

- Predpostavimo, da več proizvajalcev polni medpolnilnik in več porabnikov prazni medpolnilnik.
  - operacija vstavi postane KP za proizvajalce

```
buf[rear] = data;
rear = (rear+1) % n;
```

  - operacija vzemi postane KP za porabnike

```
result = buf[front];
front = (front+1) % n;
```

  - operaciji vstavi in vzemi nista kritični ena za drugo, ne glede na to, da souporabljata isti **buf**, vendar na drugih mestih **front** in **rear**

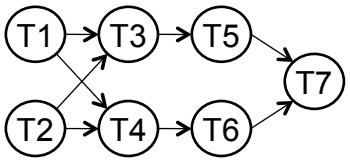
- Potrebno je zagotoviti medsebojno izključevanje uporabe operacije **vstavi** in **vzemi**:
  - **mutexD**, da zaklenemo spr. **rear** v operaciji **vstavi**, ki je v skupni uporabi vseh proizvajalcev,
  - **mutexF**, da zaklenemo spr. **front** v operaciji **vzemi**, ki je v skupni uporabi vseh porabnikov.

Uporaba omejenega medpomnilnika med več proizvajalci in več porabniki

```
typeT buf[n];       /* an array of some type T */
int front = 0, rear = 0;
sem empty = n, full = 0;      /* n-2 <= empty+full <= n */
sem mutexD = 1, mutexF = 1;  /* for mutual exclusion   */
process Producer[i = 1 to M] {
  while (true) {
    ...
    produce message data and deposit it in the buffer;
    P(empty);
    P(mutexD);
    buf[rear] = data; rear = (rear+1) % n;
    V(mutexD);
    V(full);
  }
}
process Consumer[j = 1 to N] {
  while (true) {
    fetch message result and consume it;
    P(full);
    P(mutexF);
    result = buf[front]; front = (front+1) % n;
    V(mutexF);
    V(empty);
    ...
  }
}
```

Še ena vaja

- Zagotavljanje pravilnega vrstnega reda izvedbe opravil s semaforji.
- Denimo, da je za rešitev nekega problema potrebno izvajati opravila po vrstnem redu, kot je prikazano na grafu:



- Denimo, da vsako opravilo izvajamo s procesom:

```
process Ti (i = 1, ..., 7) {
    wait for predcessors, if any;
    execute the task;
    signal successor, if any;
}
```

To pomeni, da npr. proces T5 čaka za svojo izvedbo na proces T3 in ko konča to sporoči procesu T7.

- Napiši program, ki bo izvajal opravila s procesi v vrstnem redu, kot je prikazano na grafu. Sinhronizacijo med procesi izvedi s semaforji. Minimiziraj število semaforjev.

Rešitev

- Rešitev z uporabo 5 semaforjev:

```
sem S[3:7] = ([5] 0);          // init
T1:: ... V(S[3]); V(S[4]);
T2:: ... V(S[3]); V(S[4]);
T3:: P(S[3]); P(S[3]); ... V(S[5]);
T4:: P(S[4]); P(S[4]); ... V(S[6]);
T5:: P(S[5]); ... V(S[7]);
T6:: P(S[6]); ... V(S[7]);
T7:: P(S[7]); P(S[7]);...
```

- Zmanjšanje števila semaforjev na 4:
  - ker se T7 izvaja za T3 in T4, lahko enega izmed semaforjev za ti dve opravili uporabimo tudi za T7, npr. S[3].

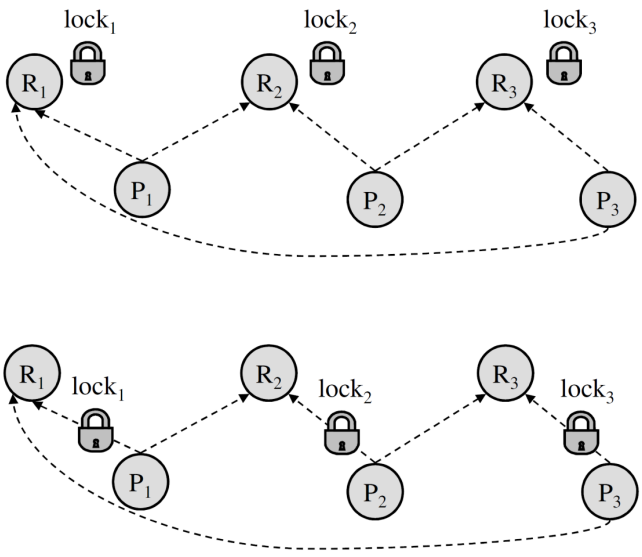
```
sem S[3:6] = ([4] 0);          // init
T1:: ... V(S[3]); V(S[4]);
T2:: ... V(S[3]); V(S[4]);
T3:: P(S[3]); P(S[3]); ... V(S[5]); V(S[5]);
T4:: P(S[4]); P(S[4]); ... V(S[6]);
T5:: P(S[5]); ... V(S[3]);
T6:: P(S[6]); ... V(S[3]);
T7:: P(S[5]); P(S[3]); P(S[3]); ...
```

Oblike selektivnega medsebojnega izključevanja

- Dve obliki:
- Več (enakih) procesov tekmuje za uporabo več različnih skupnih virov, ki si jih delijo procesi med seboj.
  - En proces lahko potrebuje za svoje izvajanje več različnih virov, ki si jih deli z različnimi procesi.
  - Modeliranje: po principu reševanja problema petih filozofov
- Več (različnih) procesov na različne načine dostopa do enega (istega) skupnega vira.
  - glede na način dostopanja lahko enim procesom dovolimo sočasno dostopanje do skupnega vira, npr. branje skupnih spr.
  - drugim procesom pa ne, npr. procesom, ki pišejo v skupno spr.
  - Modeliranje: po principu reševanja problema branja in pisanja

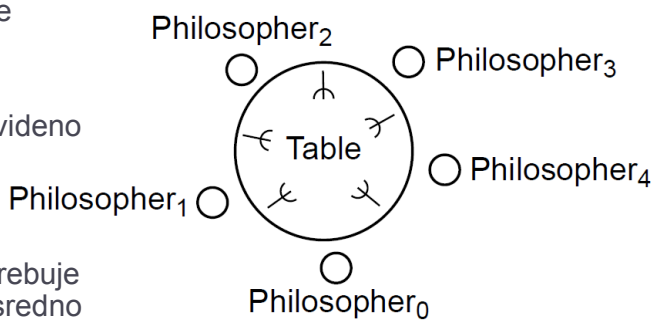
Kje je problem pri deljenju več skupnih virov hkrati več procesom?

- Denimo, da imamo več procesov (P) in več skupnih virov (R), ki so zaščiteni s ključavnicami.
  - Vsak proces mora zakleniti vse vire, ki jih hoče ekskluzivno uporabljati.
- Zgodi se lahko smrtni objem:
  - denimo: vsak proces P[i] potrebuje za svoje izvajanje vir R[i] in R[(i+1)%n].
  - **smrtni objem:** vsak proces P[i] zaklene vir R[i] in potem hoče zakleniti še R[(i +1)%n], kar se nikoli ne zgodi, saj je ta vir že zaklenjen s procesom P[(i+1)%n].



Problem petih filozofov

- Problem petih filozofov:
  - Pet filozofov sedi za okroglo mizo. V svojem življenju delajo samo dve opravili: razmišljajo in jedo.
  - Na mizi je za prehranjevanje predvideno pet vilic (palčk), ki so postavljene med filozofe.
  - Vsak filozof za prehranjevanje potrebuje par vilic, ki jih sestavi iz vilic neposredno na svoji levi in desni.
  - Kako uskladiti prehranjevanje filozofov, da se ne zgodi situacija, ko vsak zgrabi po eno vilico in jo ne spusti. Zato ne more nihče jesti in vsi od lakote umrejo.



Formulacija problema petih filozofov

- Napišimo program, ki simulira delo filozofov:

```
process Philosopher[i = 0 to 4] {
    while (true) {
        think;
        acquire a pair of forks;
        eat;
        release the forks;
    }
}
```

- Predpostavimo, da lahko filozof različno dolgo jé in razmišlja.
- Preprečiti moramo smrtni objem:
  - vsak filozof vzame natanko ene vilice, kar pomeni, da nihče ne more jesti.

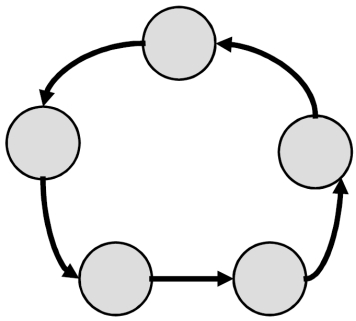
Reševanje z binarnimi semaforji

- Realizirati moramo akciji za pridobitev in sprostitev vilic.
- Ideja: zaščitimo vilice z binarnimi semaforji:
  - vsake vilice lahko uporabi samo en filozof naenkrat,
  - faza prehranjevanja je KP zaradi vilic, potrebno je pridobiti vilice iz leve in desne strani
  - operaciji P(left) in P(right) za pridobitev vilic
  - operaciji V(left) in V(right) za sprostitev vilic
- Prvi poskus rešitve:
  - simetrična rešitev: najprej vzamemo vilice iz desne strani, nato iz leve

```
sem fork[5] = ([5] 0);
process Philosopher[i =0 to 4] {
  while(true) {
    P(fork[i]); P(fork[(i+1)%5]);
    eat;
    V(fork[i]); V(fork[(i+1)%5]);
    think;
  }
}
```

Reševanje z binarnimi semaforji

- Ali smo ustrezno rešili problem?
- NE. Še vedno možnost smrtnega objema.
- Čakanje v krogu.



- Kako se temu izogniti?
  - Razbijmo krog.
    - Prej: simetrična rešitev: vsi najprej čakajo na prosto desno vilico, nato na prosto levo vilico.
    - Asimetična rešitev: Nekdo naj čaka najprej levo in nato desno.
      S tem se znebimo krožnega čakanja.

Rešitev problema petih filozofov

- Štiri filozofi pridobijo vilice enako kot prej:
  - najprej vilico na levi strani,
  - potem vilico na desni strani
- Zadnji filozof pa ravno obratno:
  - najprej vilico na desni strani
  - potem vilico na levi strani

```
sem fork[5] = ([5] 0);
process Philosopher[i =0 to 3] {
  while(true) {
    P(fork[i]); P(fork[i+1]);
    eat;
    V(fork[i]); V(fork[i+1]);
    think;
  }
}

process Philosopher[4] {
  while(true) {
    P(fork[0]); P(fork[4]);
    eat;
    V(fork[0]); V(fork[4]);
    think;
  }
}
```

Rešitev problema petih filozofov

- Filizofe razdelimo na tiste, ki sedijo na sodih sedežih in tiste, ki so na lihih sedežih.
- Sodi filozofi najprej vzamejo desne vilice in nato leve.
- Lihi filozofi pa ravno obratno.
- Tudi tako preprečimo smrtni objem zaradi krožnega čakanja.
- Vse rešitve lahko posplošimo na n procesov.

```
sem fork[5] = ([5] 0);
process Philosopher[i =0 to 4] {
  while(true) {
    if (i%2 == 0) {
      P(fork[(i+1)%5]); P(fork[i]);
    } else {
      P(fork[i]); P(fork[(i+1)%5]);
    }
    eat;
    V(fork[i]); V(fork[(i+1)%5]);
    think;
  }
}
```

Problem branja in pisanja

- Problem branja in pisanja v skupni pomnilnik je konkretna izvedba splošnega principa, kjer do skupnega pomnilnika dostopajo procesi na različne načine (eni berejo, drugi pišejo in berejo ipd.):
  - potrebno je izvajati medsebojno izključevanje dostopanja za različne operacije na različne načine,
  - ta princip je zelo pogost v računalniških aplikacijah.
- Primer: baza podatkov
  - branje podatkov – tu se izvaja samo branje podatkov iz skupne podatkovne baze
  - posodobitev (ažuriranje) podatkov – tu najprej preverimo podatke v podatkovni zbirki (beremo) in nato zapišemo nove podatke v podatkovno zbirko, če je to potrebno
- Problem branja/pisanja na skupnih podatkih:
  - potrebno je razviti mehanizem sinhronizacije, ki bo omogočal pisanje podatkov samo enemu procesu hkrati, tako da bo omogočal
    - sočasno branje podatkov,
    - medsebojno izključevanje operacij pisanja z drugimi operacijami pisanja in tudi branja.

Prva možnost rešitve

- Imejmo problem branja in pisanja na skupni bazi podatkov.
- Izvedemo en binarni semafor, ki ga uporabimo za vsako operacijo, torej za branje in pisanje v bazo.
  - s tem zagotovimo medsebojno izključevanje vseh operacij (ne samo pisanja)
  - preprečimo tudi sočasno branje podatkov.

```
sem rw = 1;
process Reader[i = 1 to M] {
  while (true) {
    ...
    P(rw);     # grab exclusive access lock
    read the database;
    V(rw);     # release the lock
  }
}
process Writer[j = 1 to N] {
  while (true) {
    ...
    P(rw);     # grab exclusive access lock
    write the database;
    V(rw);     # release the lock
  }
}
```

Druga možnost rešitve

- To, da je baza zaklenjena za vse operacije, ko nekdo bere podatke, je premočna omejitev. Kako jo omilimo?

- Ideja:
  - Ko nek proces bere podatke, jih lahko še ostali berejo.
    Ena možnost: ko začne prvi proces brati podatke, to lahko dovolimo še ostalim procesom, ko konča zadnji izmed te skupine pa lahko dovolimo še ostale operacije.
  - Torej: proces, ki začne prvi z branjem, izvede ukaz **`P(rw)`** in s tem zaklene bazo, ostali procesi, ki berejo, pa ne izvajajo tega ukaza (gredo mimo).
  - Zadnji proces, ki izvaja branje, izvede ukaz **`V(rw)`** in s tem odklene bazo. Ostali procesi (ne zadnji) pa ne izvajajo tega ukaza.
  - Kako to naredimo?
    - Vpeljemo dodatni števec **`nr`**, ki atomarno šteje, koliko bralcev trenutno izvaja branje. Ko je **`nr == 1`**, imamo prvi proces, ko je **`nr == 0`** je zadnji že izvedel branje.
    - Za atomarno povečevanje in zmanjševanje števca uporabimo dodatni binarni semafor **`mutexR`**.

Rešitev problema branja/pisanja

```
int nr = 0;       # number of active readers
sem rw = 1;       # lock for access to the database
sem mutexR = 1;   # lock for reader access to nr
process Reader[i = 1 to m] {
  while (true) {
    ...
    P(mutexR);
      nr = nr+1;
      if (nr == 1) P(rw);  # if first, get lock
    V(mutexR);
    read the database;
    P(mutexR);
      nr = nr-1;
      if (nr == 0) V(rw);  # if last, release lock
    V(mutexR);
  }
}
process Writer[j = 1 to n] {
  while (true) {
    ...
    P(rw);
    write the database;
    V(rw);
  }
}
```

Pomanjkljivosti zadnje rešitve

- Zadeva deluje. Zagotovljeno je hkratno branje in medsebojno izključevanje operacij pisanja in branja.
- Vendar:
  - prednost dajemo pisanju: če nek proces izvaja branje in v nadaljevanju prideta hkrati dva procesa, en bi bral, drugi bi pisal, ima prednost tisti, ki bere. To se lahko ponovi in vedno bo dobil prednost tisti, ki bere.
  - Rešitev ni poštena.

- Izvedba rešitve problema: Procesi naj se izvajajo po vrsti tako kot prihajajo zahteve.
  - Potrebno je voditi vrstni red procesov dostopanja do baze.
  - Uporabimo pogojno sinhronizacijo in princip podajanja štafete – semafor, s katerim dostopamo do baze (branje ali pisanje) podamo tistemu, ki je naslednji na vrsti.

Rešitev problema branja in pisanja s pogojno sinhronizacijo

- Definirajmo:
  - nr število aktivnih procesov, ki berejo podatke iz baze (na začetku 0)
  - nw število aktivnih procesov, ki pišejo podatke v bazo (na začetku 0)

- Rešitev z atomarnimi ukazi:

če nihče ne piše, lahko beremo

če nihče ne piše in ne bere, lahko pišemo

```
int nr = 0, nw = 0;
## RW:  (nr == 0 ∨ nw == 0) ∧ nw <= 1
process Reader[i = 1 to m] {
  while (true) {
    ...
    ⟨await (nw == 0) nr = nr+1;⟩
    read the database;
    ⟨nr = nr-1;⟩
  }
}
process Writer[j = 1 to n] {
  while (true) {
    ...
    ⟨await (nr == 0 and nw == 0) nw = nw+1;⟩
    write the database;
    ⟨nw = nw-1;⟩
  }
}
```

Rešitev problema branja in pisanja s pogojno sinhronizacijo izvedeno s semaforji

- Dodamo še
  - **`dr`** – število zadržanih procesov, ki berejo podatke iz baze
  - **`dw`** – število zadržanih procesov, ki pišejo podatke v bazo
  - sestavljeni binarni semafor (**`sem e r w`**) za izvedbo štafete.
- Princip podajanja štafete:
  - če noben proces ne piše v bazo, proces Reader poda štafeto vsem ostalim Reader-jem, če jih je kaj, sicer pa še vsem preostalim Writer-jem, če jih je kaj
  - proces Writer, potem ko zapiše podatke v bazo, poda štafeto vsem ostalim Reader-jem, če jih je kaj, sicer pa še vsem preostalim Writer-jem, če jih je kaj

```
int nr = 0,    ## RW: (nr==0 or nw==0) and nw<=1
    nw = 0;
sem e = 1,     # controls entry to critical sections
    r = 0,     # used to delay readers
    w = 0;     # used to delay writers
               # at all times 0 <= (e+r+w) <= 1
int dr = 0,    # number of delayed readers
    dw = 0;    # number of delayed writers
process Reader[i = 1 to M] {
  while (true) {
    # ⟨await (nw == 0) nr = nr+1;⟩
      P(e);
      if (nw > 0) { dr = dr+1; V(e); P(r); }
      nr = nr+1;
      if (dr > 0) { dr = dr-1; V(r); }
      else V(e);
    read the database;
    # ⟨nr = nr-1;⟩
      P(e);
      nr = nr-1;
      if (nr == 0 and dw > 0) { dw = dw-1; V(w); }
      else V(e);
  }
}
process Writer[j = 1 to N] {
  while (true) {
    # ⟨await (nr == 0 and nw == 0) nw = nw+1;⟩
      P(e);
      if (nr > 0 or nw > 0) { dw = dw+1; V(e); P(w); }
      nw = nw+1;
      V(e);
    write the database;
    # ⟨nw = nw-1;⟩
      P(e);
      nw = nw-1;
      if (dr > 0) { dr = dr-1; V(r); }
      elseif (dw > 0) { dw = dw-1; V(w); }
      else V(e);
  }
}
```

Vaje: naloga 1: proizvajalec/porabnik

- Denimo, da imamo dve operaciji:
  - **`broadcast(message)`**: vsem procesom, ki trenutno sprejemajo sporočila (poslušajo), predaj kopijo sporočila message in počakaj, dokler vsi ne dobijo tega sporočila,
  - **`listen(x)`**: čakaj na naslednje sporočilo, sprejmi kopijo sporočila in jo shrani v lokalno spremenljivko **`x`**
- Proizvajalec z operacijo **`broadcast`** predaja sporočila vsem porabnikom, ki trenutno čakajo na sporočila (**`listen`**). Če ni nobenega takega porabnika, potem **`broadcast(m)`** ne naredi nič. Če obstajajo takšni porabniki, proizvajalec preda vsakemu izmed njih kopijo sporočila in čaka, da ga vsi porabniki dobijo. Torej obe operaciji druga drugo ustavljata, dokler ni sporočilo predano.
- Naloga:
  Izvedi obe operaciji **`broadcast`** in **`listen`**. Predpostavi, da predstavlja sporočilo ena spremenljivka **`int`**. Uporabi semaforje za sinhronizacijo. Pri tem predpostavi, da imamo lahko več proizvajalcev in več porabnikov, toda v vsakem trenutku se lahko izvaja samo ena operacija **`broadcast`** in ena operacija **`listen`**.

Naloga 1: rešitev

```
int buffer, waiting = 0;
sem enter = 1; go = 0;

broadcast(m) {
  P(enter);
  buffer = m;
  if (waiting > 0) {
    waiting = waiting - 1;
    V(go);
  } else
    V(enter);
}

listen(x) {
  P(enter);
  waiting = waiting + 1;  //registriraj še en proces, ki čaka
  V(enter);
  P(go);                  // čakaj, da prideš na vrsto
  x = buffer;
  if (waiting > 0) {
    waiting = waiting - 1;
    V(go);                // štafeta
  } else
    V(enter);
}
```

Vaje: naloga 2: sleep in wakeup

- V jedru Linux-a imamo implementirani dve atomarni operaciji:
  - **`sleep()`**: zaustavi izvajanje procesa
  - **`wakeup()`**: zbudi vse zaustavljene procese

  - Proces, ki izvede operacijo **`sleep()`** se vedno takoj ustavi.
  - Operacija **`wakeup()`** zbudi vse procese, ki so bili zaustavljeni z operacijo **`sleep()`**, med enimi in drugim klicem operacije **`wakeup()`**.

- Izvedi obe operaciji s semaforji.

Naloga 2: rešitev

- Rešitev po metodi podajanja štafete

```
int dp = 0;
sem delay = 0;
sem e = 1;

sleep() {
  P(e);
  dp++;
  V(e);
  P(delay);
  dp--;
  if (dp > 0) V(delay);
  else V(e);
}

wakeup() {
  P(e);
  if (dp > 0) V(delay);
  else V(e);
}
```

- Slaba rešitev: race condition

```
int dp = 0;
sem delay = 0;
sem e = 1;

sleep() {
  P(e);
  dp++;
  V(e);
  P(delay);
}

wakeup() {
  P(e);
  while (dp > 0) {
    dp--;
    V(delay);
  }
  V(e);
}
```

5 Monitorji

V poglavju so predstavljeni monitorji in nekaj primerov uporabe monitorjev.

Poglavje obravnava:

- **Uvedba, izvedba, delovanje monitorjev:**
  - kaj je to monitor,
  - *pogojne spremenljivke*,
  - delovanje monitorjev.
- **Primeri uporabe monitorjev:**
  - uporaba omejenega medpomnilnika v problemu proizvajalec/porabnik: pogojna sinhronizacija,
  - problem sočasnega branja in pisanja: sinhronizacija s signali,
  - problem brivca: princip zmenka (rendezvous),
  - gnezdeni klici monitorjev.

Ponovimo ...

- Mehanizmi sinhronizacije:
  - ključavnice
    - splošen mehanizem zaklepanja/odklepanja KP,
    - čakanje na vstop je čakanje v delovanju.
  - pogojne spremenljivke
    - čakanje na vstop je čakanje v mirovanju,
    - proces ,ki čaka, je zaustavljen in postavljen v vrsto zaustavljenih procesov (FIFO vrsta),
    - ko pogojna spr. dobi signal, se prvi proces iz vrste zaustavljenih procesov postavi v vrsto za izvajanje
  - semaforji
    - kombinacija ključavnic in pogojnih spremenljivk
    - splošen princip za reševanje problemov, ki vključujejo medsebojno izključevanje in pogojno sinhronizacijo
    - Slabosti:
      - programiranje na nizkem nivoju: mešamo mehanizme sinhronizacije z nalogami, ki jih rešujemo z vzporednim programom. Lahko pride do napak.
      - semaforje lahko uporabimo za oba tipa sinhronizacije, medsebojno izključevanje in pogojno sinhronizacijo, kar pomeni, da iste operacije semaforja uporabljamo lahko za oba tipa sinhronizacije
  - monitorji
    - bolj strukturiran mehanizem sinhronizacije,
    - definirani kot objekti, višji nivo programiranja,
    - implementirana oba tipa sinhronizacije (medsebojno izključevanje izvajanja je implicitno doseženo)

Monitorji

- Monitorji združujejo spremenljivke, funkcije nad spremenljivkami in postopke začetnih nastavitev v skupni abstraktni podatkovni strukturi.
  - Spremenljivke:
    - vrednosti spremenljivk monitorja predstavljajo stanje monitorja.
  - Funkcije:
    - dostopanje (spreminjanje/prikazovanje) do spremenljivk je omogočeno samo preko funkcij, ki so definirane v procedurah monitorja.
    - **Procedure v monitorju se izvajajo po principu medsebojnega izključevanja.**
  - Postopki začetnih nastavitev:
    - postopki inicializacije monitorja – začetno kreiranje spr. in nastavitve spr. – stanje monitorja na začetku
  - V programskem jeziku Java jih obravnavamo kot razrede.
- Monitorji so bili prvič predstavljeni v 70-ih letih – T. Hoare:
  - izvedba monitorja s sestavljenim binarnim semaforjem
- Popularizacija monitorjev: Java.

Definicija monitorja

- Obravnavajmo monitor kot objekt, ki ima naslednjo deklaracijo:

```
monitor monitorName {
   deklaracije spremenljivk;     //spremenljivke monitorja
   inicializacija monitorja;     //postopki začetnih nastavitev monitorja
   procedure;                    //operacije, metode nad spremenljivkami
}
```

  - inicializacija monitorja:
    - izvede se ob nastanku primerka monitorja,
    - incializacija je potrebna, preden lahko sploh začnemo uporabljati ostale procedure monitorja
    - ob inicializaciji se kreirajo spr. in procedure in nastavijo na začetne vrednosti
  - Proces lahko dostopa samo do procedur monitorja na sledeči način:

```
call monitorName.procedureName(arguments);
```

Osnovne lastnosti monitorja

- Osnovne lastnosti monitorja:
  - samo imena procedur so vidna navzven.
  - procedure lahko uporabljajo svoje lokalne spr. in t.i. permanentne spremenljivke, ki so skupne ostalim proceduram znotraj monitorja.
  - postopki inicializacije v procedurah ne morejo dostopati do spr. zunaj monitorja.
  - permanentne spremenljivke se inicializirajo ob nastanku objekta monitorja in sicer preden do njih dostopajo procedure.

Mehanizmi sinhronizacije v monitorjih

- Princip medsebojnega izključevanja procedur monitorja:
  - Ob vsaki konkretni izvedbi monitorja dobi izvedba monitorja svoj lasten binarni semafor –ključavnico.
  - Izvedba medsebojnega izključevanja:
    - Preden proces začne izvajati s proceduro monitorja, mora pridobiti monitorjevo ključavnico.
    - Če je ključavnica že zasedena, oddana drugemu procesu, se izvajanje procesa zaustavi in se proces postavi v vrsto za čakanje.
    - Ključavnica se odklene, ko proces konča z izvajanjem procedure monitorja.
- Princip pogojne sinhronizacije:
  - Vsak monitor lahko uporablja pogojne spremenljivke.
  - Proces, ki uporablja monitor, lahko čaka na svoje izvajanje glede na pogojno spr., ki je definirana v monitorju.
    - Proces, ki čaka glede na pogojno spr., sprosti ključavnico.
  - Drug proces, ki tudi uporablja ta monitor, lahko pošlje signal preko pogojne spr. čakajočemu procesu, ki se postavi v vrsto za izvajanje.

Pogojne spremenljivke v monitorju

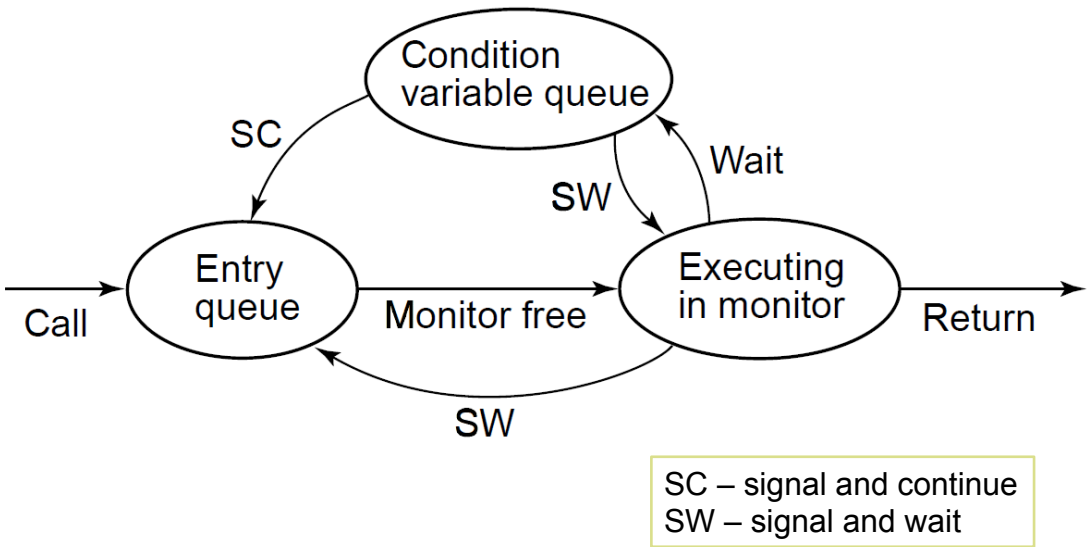
- Pogojne spremenljivke se uporabljajo zato, da zaustavijo delovanje procesov, v primeru, ko ni izpolnjen pogoj pogojne spr.. Ko pa je ta pogoj izpolnjen pa pogojne spr. vključujejo mehanizem, da proces lahko nadaljuje svoje izvajanje.
- Deklaracija pogojne spremenljivke **`cond cv`**
- Operacije:

`wait(cv)`	proces postavi na konec vrste zaustavljenih procesov in sprosti monitorjevo ključavnico
`wait(cv, rank)`	proces postavi v vrsto zaustavljenih procesov na mesto z rangom **`rank`** (procesi so v vrsti urejeni naraščajoče po rangu), sprosti monitorjevo ključavnico
`signal(cv)`	zbudi proces, ki je na začetku vrste zaustavljenih procesov, in nadaljuj z izvajanjem
`signal_all(cv)`	zbudi vse procese iz vrste zaustavljenih procesov in nadaljuj z izvajanjem
`empty(cv)`	**`true`**, če je čakalna vrsta prazna
`minrank(cv)`	vrni rang procesa, ki je na začetku čakalne vrste

Načini signalizacije in izvajanja procesov pri pogojnih spremenljivkah

- Kaj storiti, ko proces pošlje signal drugemu procesu za izvajanje?
  - Sedaj imamo dva procesa, ki lahko nadaljujeta z izvajanjem, vendar samo eden lahko takoj napreduje. Kateri?
- Signal and Continue (SC)
  - proces, ki odda signal, nadaljuje z izvajanjem
  - proces, ki sprejme signal, se zbudi in se postavi v vrsto za izvajanje
  - proces, ki odda signal, ohrani ekskluziven dostop do monitorja
- Signal and Wait (SW)
  - proces, ki prejme signal, začne z izvajanjem
  - proces, ki odda signal, se postavi v vrsto za izvajanje
  - proces, ki prejme signal, dobi ekskluziven dostop do monitorja
- Signal and Urgent Wait (SUW)
  - proces, ki prejme signal, začne z izvajanjem
  - proces, ki odda signal, se postavi na začetek vrste za izvajanje
  - proces, ki prejme signal, dobi ekskluziven dostop do monitorja

Diagram sinhronizacije z monitorji



Implementacija semaforjev z monitorji

```
monitor Semaphore {
  int s = 0;  ## s >= 0
  cond pos;   #  signaled when s > 0
  procedure Psem() {
    while (s == 0) wait(pos);
    s = s-1;
  }
  procedure Vsem() {
    s = s+1;
    signal(pos);
  }
}
```

- Spremenljivka s je vrednost semaforja.
  - procedura Psem je operacija P
    - ko proces kliče proceduro Psem, čaka dokler s == 0, ko lahko nadaljuje z izvajanjem zmanjša s za 1
  - procedura Vsem je operacija V
    - procedura Vsem najprej poveča s za 1 in nato signalizira pogojni spr. pos naj zbudi proces, ki čaka
- Razlika med SC in SW načinom signalizacije
  - SW: ko se izvede operacija signal(pos) v proceduri Vsem se takoj začne izvajati proces, ki je bil zbujen
  - SC: ko se izvede operacija signal(pos) v proceduri Vsem, mora zbujeni proces čakati na izvajanje. To pa pomeni, da se lahko vmes izvede še nek drugi proces, ki v operaciji Psem zmanjša s za 1 in mora zato zbujeni proces še enkrat preverjati, če je s pozitiven. **To ni FIFO semafor.**
  - v primeru SW zato lahko zamenjamo `while (s==0) wait(pos);` z `if (s==0) wait(pos);`

Implementacija FIFO semaforjev z monitorji

```
monitor FIFOsemaphore {
  int s = 0;  ## s >= 0
  cond pos;   #  signaled when s > 0
  procedure Psem() {
    if (s == 0)
      wait(pos);
    else
      s = s-1;
  }
  procedure Vsem() {
    if (empty(pos))
      s = s+1;
    else
      signal(pos);
  }
}
```

- Da bo prejšnji semafor FIFO semafor v obeh načinih signalizacije je potrebno popraviti proceduro Vsem:
  - če obstaja kakšen proces, ki čaka na pogojni spr. pos (v proceduri Psem), potem signal(pos), vendar ne povečaj semaforja s. s povečaj samo v primeru, ko noben proces ne čaka.
  - ustrezno temu je potrebno popraviti tudi Psem:
    - če proces čaka na pos, potem ne zmanjšaj s za 1, ker Vsem s ni povečal,
    - sicer zmanjšaj za 1
- Princip podajanja pogoja (*ang. passing the condition*)
  - procedura, ki pošlje signal, in tako zbudi proces, ki čaka, implicitno pove tudi, da je s pozitiven
  - procesu, ki se zbudi, zato ni potrebno preverjati tega pogoja še enkrat (v primeru SC)
  - vsi ostali procesi pa tega pogoja ne vidijo in zato tudi ne čakajo na pogojni spr.

Razlika med monitorji in semaforji

- operacija **`wait`** v monitorju in operacija **`P`** v semaforju zaustavi proces
- operacija **`signal`** v monitorju in operacija **`V`** v semaforju zbudi proces za izvajanje

- Vendar:
  - 1. razlika:
    - **`wait`** vedno zaustavi proces, dokler ni izvedena operacija **`signal`**
    - operacija **`P`** zaustavi proces, le ko je vrednost semaforja 0
  - 2. razlika:
    - operacija signal nima efekta, če noben proces ne čaka na pogojni spr.
    - operacija **`V`** zbudi proces, ki čaka, tako da poveča vrednost semaforja (vedno nekaj naredi)
    - Torej: izvajanje operacije **`signal`** se ne zapomni.
  - Pogojno sinhronizacijo zato v obeh primerih programiramo različno.

Uporaba monitorjev

- Monitor je podatkovna struktura (razred), ki se uporablja:
  - za izvajanje atomarnih operacij in ukazov nad skupnimi spremenljivkami, ki so zajete v monitorju.
  - za kontrolo dostopanja do skupnih virov zunaj monitorjev.

- V nadaljevanju bomo pogledali uporabe monitorjev za različne tehnike sinhronizacije
  - uporaba omejenega medpomnilnika: pogojna sinhronizacija
  - problem branja in pisanja skupnih podatkov: sinhronizacija s signali
  - problem brivca: princip zmenka (rendezvous)

Sinhronizacija dostopanja do omejenega medpomnilnika med proizvajalcem in porabnikom

- To je primer, ki smo ga že obravnavali. Gre za princip pogojne sinhronizacije pri dostopanju do skupnega medpomnilnika, ki je prostorsko omejen.
- Denimo, da imamo na razpolago medpomnilnik s kapaciteto hranjenja **`n`** podatkov.

- Implementirajmo medpomnilnik za komunikacijo med proizvajalcem in porabnikom kot monitor.
- Monitorjeve spremenljivke:
  - **`buf[n], front, rare, count`** (šteje, koliko mest v medpomnilniku imamo trenutno zasedenih)
  - **`cond not_full`** – zaustavi polnjenje medpomnilnika s strani proizvajalca dokler medpomnilnik ni pripravljen za pisanje (ni prost)
  - **`cond not_empty`** – zaustavi porabnika, da ne bere podatkov iz medpomnilnika, dokler ni novih podatkov v medpomnilniku

- Procedure v monitorju:
  - **`deposit(typeT)`** – vstavi podatek v medpomnilnik (proizvajalec)
  - **`fetch(*typeT)`** – zapiši podatek iz medpomnilnika v spr. tipa **`typeT`** (porabnik)

Implementacija monitorja za uporabo omejenega medpomnilnika

```
monitor Bounded_Buffer {

  typeT buf[n];      # an array of some type T
  int front = 0,     # index of first full slot
      rear = 0;      # index of first empty slot
      count = 0;     # number of full slots
  ## rear == (front + count) % n
  cond not_full,     # signaled when count < n
       not_empty;    # signaled when count > 0

  procedure deposit(typeT data) {
    while (count == n) wait(not_full);
    buf[rear] = data; rear = (rear+1) % n; count++;
    signal(not_empty);
  }

  procedure fetch(typeT &result) {
    while (count == 0) wait(not_empty);
    result = buf[front]; front = (front+1) % n; count--;
    signal(not_full);
  }

}
```

Uporaba:
- Proizvajalec: **`Bounded_Buffer.deposit(a[i]);`**
- Porabnik: **`Bounded_Buffer.fetch(&b[i]);`**

Problem branja/pisanja skupnih podatkov

- To je primer sinhronizacije s signali: uporabili bomo operacijo **`signal_all`**.
- Problem: branje in pisanje podatkov v bazo.

- Selektivno medsebojno izključevanje:
  - sočasno se lahko izvajajo operacije branja
  - pisanje pa se mora izvajati atomarno glede na druge operacije: torej zagotoviti je potrebno medsebojno izključevanje z drugimi operacijami pisanja in tudi branja.

- Možnosti izvedbe z monitorji:
  - bazo lahko izvedemo kot en monitor: branje in pisanje dve proceduri monitorja
    - s tem zagotovimo ekskluzivno izvajanje operacij branja in pisanja (ne dovolimo sočasnega branja) – to ni naš cilj
  - ideja: monitor uporabimo samo za kontrolo dostopanja do podatkov iz baze
    - monitor, ki izvaja kontrolo dostopanja vseh procesov do podatkov v bazi

Izvedba kontrole dostopanja z monitorjem

- Monitor nadzira dostopanje do baze.
- Spremenljivke v monitorju:
  - **`nr`** – število trenutnih bralcev podatkov,
  - **`nw`** – število trenutnih zapisovalcev podatkov,
  - **`cond oktoread`** – pogojna spr., ki omogoča čakanje procesom, ki berejo podatke, na dostop do baze,
  - **`cond oktowrite`** – pogojna spr., ki omogoča čakanje procesom, ki pišejo podatke, na dostop do baze.

- Procedure v monitorju:
  - **`request_read`**:
    - zagotavlja dostop do branja podatkov v primeru, ko se ne izvaja pisanje v bazo, sicer čaka;
  - **`request_write`**:
    - zagotavlja dostop do pisanja podatkov v primeru, ko se ne izvaja pisanje v bazo, sicer čaka;
  - **`release_read`**:
    - zadnji proces, ki bere, pošlje signal procesu, ki čaka na pisanje (če obstaja);
  - **`release_write`**:
    - signaliziramo možnost nadaljevanja dela vsem procesom, ki čakajo na izvajanje.

Implementacija monitorja

```
monitor RW_Controller {
  int nr = 0, nw = 0;  ## (nr == 0 ∨ nw == 0) ∧ nw <= 1
  cond oktoread;     #  signaled when nw == 0
  cond oktowrite;    #  signaled when nr == 0 and nw == 0

  procedure request_read() {
    while (nw > 0) wait(oktoread);
    nr = nr + 1;
  }

  procedure release_read() {
  nr = nr - 1;
  if (nr == 0) signal(oktowrite);  # awaken one writer
  }

  procedure request_write() {
    while (nr > 0 || nw > 0) wait(oktowrite);
    nw = nw + 1;
  }

  procedure release_write() {
    nw = nw - 1;
    signal(oktowrite);       # awaken one writer and
    signal_all(oktoread);    # all readers
  }
}
```

Uporaba:
- Branje:

```
RW_Controller.request_read();
branje podatkov iz baze;
RW_Controller.release_read();
```

- Pisanje:

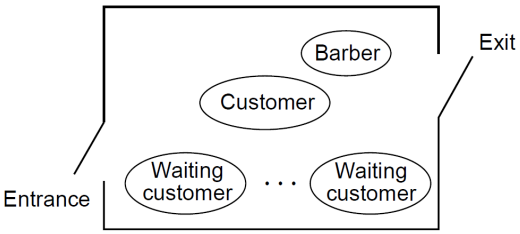
```
RW_Controller.request_write();
pisanje podatkov v bazo;
RW_Controller.release_write();
```

Rendezvous

- To je primer uporabe monitorjev v aplikacijah odjemalec-strežnik:
  - Imamo en strežnik in več odjemalcev.
  - Interakcija je ena-na-ena: dva procesa se dobita skupaj in uredita skupne zadeve (rendezvous) :
    - najprej se morata dogovoriti za zmenek: signalizacija s pogojnimi spr.
    - proces na strežniku mora čakati na proces odjemalca, da ga lahko “postreže”
    - odjemalec mora čakati, da je strežnik prost, da se mu lahko “posveti”
    - strežnik ima lahko zmenke z vsemi odjemalci, odjemalec samo s strežnikom.

Problem brivca: rendezvous

- Brivec ima svoj lokal, kjer striže svoje stranke na sledeč način:
  - počaka na stranko;
  - stranko postriže;
  - dovoli ji, da lahko odide;
  - počaka, da odide.
- Stranka, ki bi se rada ostrigla:
  - pride v lokal;
  - počaka, dokler ni brivec na voljo;
  - se striže;
  - odide iz lokala.



- Brivec je strežnik, stranka je odjemalec.
- Rendezvous: brivec striže stranko.
- Problem: sinhronizacija procesov brivca in stranke
- Izvedba: monitor, je lokal, ki omogoča “zmenke” med brivcem (strežnikom) in strankami (odjemalci)

Monitor za izvedbo brivskega lokala

- Procedure:
  - `get_haircut` – uporablja odjemalec - stranka, da zahteva svoje striženje
  - `get_next_customer` – uporablja strežnik – brivec, da zahteva novo stranko za striženje
  - `finished_cut` – uporablja strežnik – brivec, ko konča eno striženje
- Uporaba monitorja:
  - proces brivca (strežnika):

```
while (true) {
  Barber_Shop.get_next_customer();
  ostriži stranko;
  Barber_Shop.finished_cut();
}
```

  - proces stranke (odjemalca):

```
Barber_Shop.get_haircut();
```

Spremenljivke v monitorju

- Zastavice – uporabljamo za javljanje statusa:
  - **`barber`** – brivec je prost (strežnik lahko sprejema zahteve)
  - **`chair`** – brivski stol je zaseden (strežnik obdeluje zahtevo)
  - **`open`** – vrata v lokal so odprta (zahteva je bila obdelana, odgovor na zahtevo je pripravljen)
    ko je **`open == 0`** – stranka je zapustila lokal

- Pogojne spremenljivke – točke čakanja:
  - uporabljamo jih za sinhronizacijo med brivcem in stranko (rendezvous).

  - Brivec čaka:
    - **`chair_occupied`** – da bo stranka sedla na stol: **`<await (chair > 0);>`**
    - **`customer_left`** – da stranka, ki je obdelana, odide:
      `<await (open==0);>`

  - Stranka čaka:
    - **`barber_available`** – da bo brivec prost: **`<await (barber > 0);>`**
    - **`door_open`** – da se jo postriže: **`<await (door > 0);>`**

Izvedba monitorja za problem brivca

- Uporaba:
  - Stranka

```
Barber_Shop.get_haircut();
```

  - Brivec

```
while (true) {
  Barber_Shop.get_next_customer();
  ostriži stranko;
  Barber_Shop.finished_cut();
}
```

- Monitor ima več točk čakanja.
- S tem zagotovimo "zmenek" strežnika s posamičnim odjemalcem.

- Ali bi se dalo ta monitor izvajati tudi s semaforji?

```
monitor Barber_Shop {
  int barber = 0, chair = 0, open = 0;
  cond barber_available;    # signaled when barber > 0
  cond chair_occupied;      # signaled when chair > 0
  cond door_open;           # signaled when open > 0
  cond customer_left;       # signaled when open == 0
  procedure get_haircut() {
    while (barber == 0) wait(barber_available);
    barber = barber - 1;
    chair = chair + 1; signal(chair_occupied);
    while (open == 0) wait(door_open);
    open = open - 1; signal(customer_left);
  }
  procedure get_next_customer() {
    barber = barber + 1; signal(barber_available);
    while (chair == 0) wait(chair_occupied);
    chair = chair - 1;
  }
  procedure finished_cut() {
    open = open + 1; signal(door_open);
    while (open > 0) wait(customer_left);
  }
}
```

Rešitev problema brivca s semaforji

```
module Barber_Shop_with_Semaphores {
   sem barber = 0, chair = 0, open = 0, left = 0;
   procedure get_haircut() {
      P(barber);
      V(chair);
      P(open);
      V(left);
   }
   procedure get_next_customer() {
      V(barber);
      P(chair);
   }
   procedure finished_cut() {
      V(open);
      P(left);
   }
}
```

Gnezdeni klici monitorjev

- Gnezdeni klic monitorja:
  - Proces, ki izvaja proceduro znotraj enega monitorja, kliče proceduro drugega monitorja, kar pomeni, da začasno zapusti prvi monitor.
- Zaprt klic:
  - ohranimo ekskluzivnost izvajanja monitorja, s katerega kličemo drugi monitor – proces za takšno izvajanje potrebuje dve ključavnici
  - slabost: lahko se zgodi smrtni objem:
    - če je npr. proces, ki je izvedel gnezdeni klic, zaustavljen na čakanju z ukazom **`wait`**, drugi monitor pa izvede gnezdeni klic prvega monitorja.
- Odprt klic
  - sprostimo ekskluzivnost izvajanja monitorja, s katerega kličemo drugi monitor, proces ima ekskluzivni dostop samo do drugega monitorja.
  - ko se procedura drugega monitorja izvede, proces spet pridobi ekskluzivni dostop do izvajanja prvega monitorja.
  - slabost: izvedba takega klica je bolj zahtevna za programiranje.

6 Porazdeljeno programiranje

V tem poglavju se seznanimo z osnovami porazdeljenega programiranja, kjer si procesi ne delijo skupnega pomnilnika, kot je to v primeru vzporednih programov, ampak komunicirajo med seboj preko komunikacijskih kanalov. Porazdeljen program je tako program, ki svoje delo izvaja porazdeljeno med več računalniki ali računalniškimi sistemi. V tem poglavju se seznanimo z eno tehniko komunikacije med porazdeljenimi procesi, ki temelji na komunikaciji s pošiljanjem in sprejemanjem sporočil. Obstaja še drugi način komunikacije s klici oddaljenih procedur, ki pa ga poglavje ne obravnava. Poglavje predstavi osnovne gradnike za porazdeljeno programiranje in obravnava nekaj primerov komunikacije s sporočili. Dejanske implementacije konkretnih primerov porazdeljenega programiranja so izvedene v dodatku C, kjer predstavimo programsko okolje MPI, ki je namenjeno porazdeljenemu programiranju.

Poglavje obravnava:

- **Osnove porazdeljenega programiranja**
- **Porazdeljeno programiranje s pošiljanjem in sprejemanjem sporočil**
  - kanali za komunikacijo s sporočili.
- **Asinhrona komunikacija s sporočili**
  - komunikacija med proizvajalcem in porabniki,
  - komunikacija med strežnikom in odjemalci,
  - komunikacija med enakovrednimi procesi.
- **Sinhrona komunikacija s sporočili:**
  - primerjava med asinhrono in sinhrono komunikacijo.

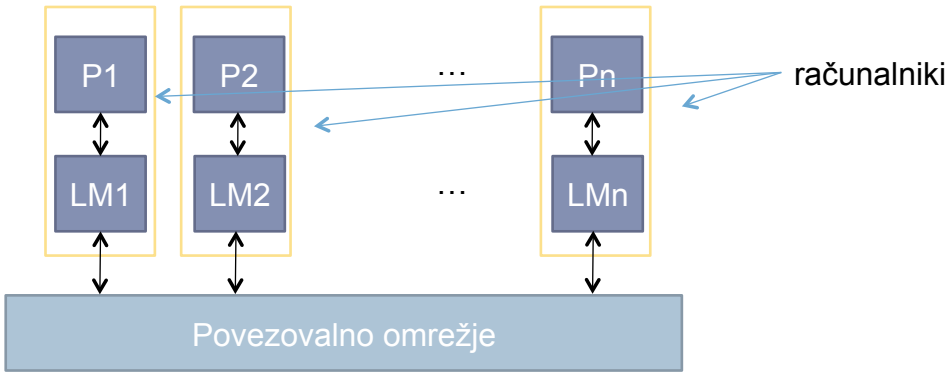
Porazdeljeno programiranje

- Porazdeljen program je program, ki svoje delo izvaja porazdeljeno med več računalniki ali računalniškimi sistemi.
  - Procesi si ne delijo skupnega pomnilnika (shared memory), pravimo da je pomnilnik porazdeljen (distributed memory).
  - Porazdeljeni procesi komunicirajo med seboj preko komunikacijskih kanalov.
  - Procesi sodelujejo med seboj z izmenjavo sporočil, ki se prenašajo preko komunikacijskih kanalov.

- Pošiljanje sporočil omogoča izmenjavo informacij in podatkov med procesi:
  - izmenjava sporočil poteka preko komunikacijskih kanalov, ki si jih delijo porazdeljeni procesi med seboj (namesto skupnih spr. pri modelu skupnega pomnilnika)
  - ker si procesi ne delijo ostalih podatkov, je medsebojno izključevanje izvajanja zagotovljeno.

- Komunikacijski kanali:
  - Omogočajo eno ali dvosmerni tok informacij.
  - Komunikacija med procesi porazdeljenega programa poteka s sprejemanjem (**receive**) in pošiljanjem (**send**) sporočil.
  - Komunikacija je lahko asinhrona ali sinhrona.

Računalniška arhitektura s porazdeljenim pomnilnikom



- Več računalnikov povezanih s povezovalnim omrežjem.
- Vsak računalnik predstavlja eno vozlišče v takšni mreži:
  - računalnik je lahko eno ali več procesorska enota
  - računalniki si ne delijo skupnega pomnilnika
  - komunikacija poteka preko povezovalnega omrežja (abstrakcija kom. kanal):
    - omrežja so različna in skalabilna,
    - veliko število možnih povezav med vozlišči omrežja,
    - omogoča veliko število sočasnih (in tudi neodvisnih) transakcij med vozlišči v omrežju.

Mehanizmi komunikacije in okolja za porazdeljeno izvajanje programov

- Komunikacijski mehanizmi:
  - s sporočili
    - asinhrona komunikacija z izmenjavo sporočil,
    - sinhrona komunikacija z izmenjavo sporočil,
  - s klici za izvajanje oddaljenih operacij
    - RPC (remote procedure call) in rendezvous,
    - RMI (remote method invocation).

- Okolja za porazdeljeno programiranje:
  - Brekeley Sockets API (BSD socket API)
    - standard za večino programskih jezikov za medmrežno komunikacijo, osnova za komunikacijo za porazdeljeno programiranje
  - RPC API, Java socket API, CORBA, Java RMI
  - Microsoft .NET
  - PVM (parallel virtual machine)
  - MPI (message passing interface)
  - …

Oblike izvajanja porazdeljeno delujočih aplikacij

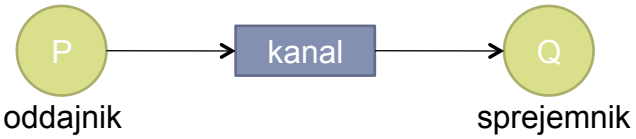
- Procesi v porazdeljeno delujočih računalniških aplikacijah so HW procesi (nimajo skupnih spr.), ki se izvajajo na različnih računalnikih (procesorjih).
  - Začetek izvajanja porazdeljenih procesov je različen.
  - Vsak proces lahko rodi nov proces.
  - Proces lahko izvaja različna opravila.

- 1. model porazdeljene aplikacije: enak program za različne podatke (SPMD):
  - En program uporabljamo za izvajanje več enakih opravil na različnih podatkih.
  - Da razdelimo opravila (podatke) med procese moramo:
    - poznati število procesov, ki bodo izvajali opravilo,
    - poznati velikost podatkov za obdelavo,
    - definirati ustrezno delitev podatkov med opravila.

- 2. model porazdeljene aplikacije: različni programi za različne podatke (MPMD):
  - Različne programe uporabimo za izvajanje različnih opravil (procesov) na različnih podatkih.
    - Primer: strežnik/ odjemalec

Komunikacija z izmenjavo sporočil

- Komunikacija med procesi poteka preko skupnih komunikacijskih kanalov z izmenjavo sporočil:
  - dva osnovna tipa operacij: **send** – pošlji sporočilo in **receive** – sprejmi sporočilo
  - Pri pošiljanju/sprejemanju sporočila gre za prenos podatkov iz pomnilnika procesa, ki oddaja sporočilo, v pomnilnik procesa, ki sprejema sporočilo.



- Komunikacija s sporočili zahteva sinhronizacijo – sporočilo ne more biti sprejeto, če še ni poslano.
- Komunikacijski kanal je lahko:
  - deljeni medpomnilnik v primeru enoprocesorske ali večprocesorske arhitekture z deljenim pomnilnikom,
  - mrežna povezava v primeru računalniške arhitekture s porazdeljenim pomnilnikom.

Dva načina izmenjave sporočil

- Sinhrona komunikacija z izmenjavo sporočil – operaciji **send** in **receive** zaustavita izvajanje procesov, dokler se ne izvedeta:
  - Operacija **send** zaustavi delovanje procesa – oddajnika sporočila, dokler proces - sprejemnik sporočila – ne sprejme sporočilo.
  - Operacija **receive** zaustavi delovanje procesa – sprejemnika sporočila, dokler sporočilo ne pride do sprejemnika.
- Asinhrona komunikacija z izmenjavo sporočil:
  - Operacija **send** ne zaustavi delovanje procesa – oddajnika sporočila (proces nadaljuje svoje delo), operacija **receive** pa zaustavi delovanje procesa – sprejemnika sporočila, dokler ne dobi sporočila.
  - Kanal je v tem primeru implementiran kot neomejena (zelo velika) FIFO vrsta, v katero se nalagajo sporočila, dokler ne pridejo do sprejemnikov.

Osnovne oblike komunikacije in sodelovanja med procesi pri porazdeljenem programiranju

- **Proizvajalec – potrošnik** (filtri) – enosmerna komunikacija:
  - izhod proizvajalca je vhod potrošnika – izhod je funkcija vhodnih podatkov – filter
  - komunikacija: potrošnik z operacijo **receive** čaka na operacijo **send** proizvajalca

- **Odjemalec/strežnik** – dvosmerna komunikacija, en proces bolj pomemben kot drugi
  - strežnik se odziva na zahteve odjemalca,
  - komunikacija: odjemalec **pošlje sporočilo** o zahtevi, strežnik čaka na sporočilo o zahtevi v operaciji **receive** in nato odgovori na zahtevo z operacijo **send,**
    gre za komunikacijo **zahteva/odgovor (request/response).**

- **Interakcija enakovrednih** – dvosmerna komunikacija, vsi procesi so enakovredni
  - gre za interakcijo med enakovrednimi procesi.
  - Običajno en proces sproži interakcijo (**pošlje sporočilo**), vsi procesi morajo biti v pripravljenosti za sprejem sporočil.

Asinhrona komunikacija z izmenjavo sporočil

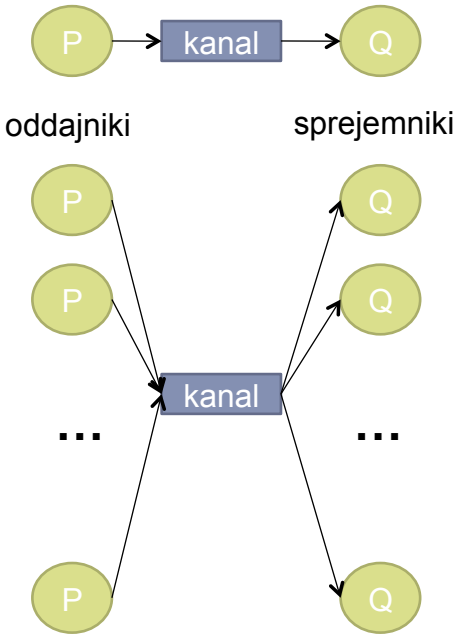
- Komunikacija
  - Predpostavimo, da imamo nek neomejen prostor za shranjevanje sporočil in podatkov, zunaj lokalnih pomnilnikov procesov, ki komunicirajo.
  - Procesi si delijo skupni komunikacijski kanal:
    - kanal je abstrakcija dejanskih fizičnih komunikacijskih povezav,
    - kanal je enosmerna povezava med oddajnikom in sprejemnikom,
    - lahko si ga predstavljamo kot medpomnilnik z neomejenim prostorom za shranjevanje podatkov v sporočilih.

- Osnovne “nedeljive” operacije komunikacije:
  - **send**
    - pošilja podatke (sporočilo) iz lokalnega pomnilnika v skupni komunikacijski kanal, ki si ga delita proces - oddajnik in proces - sprejemnik,
    - pri izvedbi operacije **send** se proces ne zaustavi.
  - **receive**
    - sprejema podatke (sporočilo) iz komunikacijskega kanala in jih shranjuje v lokalni pomnilnik,
    - operacija **receive** zaustavi delovanje procesa, dokler ne prejme celotnega sporočila (vseh podatkov v sporočilu).

Komunikacijski kanal

- Komunikacijski kanal predstavlja povezavo za prenos sporočil med procesom – oddajnikom in procesom – sprejemnikom. Kanal si lahko predstavimo kot FIFO vrsto sporočil, ki so bila poslana, pa še niso bila sprejeta.

- Operaciji:
  - **send** – pripni sporočilo v vrsto kanala na zadnje mesto
  - **receive** – vzemi in zbriši sporočilo iz prvega mesta vrste kanala



Deklaracija kanala **chan**

- Kanal je skupen vsem procesom, ki ga uporabljajo.
- Deklaracija kanala:
  - `chan ch(type1 field1, …, typen fieldn);`
    - seznam `typei fieldi` predstavlja strukturo podatkov, iz katerega je sestavljeno sporočilo, ki se prenaša po kanalu.
  - Imena spr. podatkov `fieldi` lahko tudi izpustimo:
    `chan ch(type1,… , typen);`
  - Tabela kanalov:
    `chan ch[n](type1, …, typem);`
    - definirana je tabela n kanalov, ki prenašajo sporočila enake strukture podatkov.
  - Primeri:
    - `chan input(char);`
    - `chan disk_access(int cylinder, int block, int count, char* buffer);`
    - `chan result[n](int);`

Operacije nad kanali

- **send** – pošlje sporočilo v kanal

  send ch(expr$_1$, …, expr$_n$);

  - operacija **ne zaustavi** delovanje procesa, ki to operacijo izvaja
  - Potek operacije:
    - izračunajo se vrednosti v **expr$_1$, ... , expr$_n$**
    - izračunane vrednosti so zbrane v sporočilu, ki je enake strukture, kot je kanal, ki prenaša sporočila,
    - sporočilo se pripne na zadnje mesto v vrsto kanala **ch**

- **receive** – sprejme sporočilo iz kanala

  receive ch(var$_1$, …, var$_n$);

  - operacija **zaustavi** delovanje procesa, ki to operacijo izvaja, proces je zaustavljen dokler je sporočilo v kanalu
  - Potek operacije:
    - sporočilo iz začetka vrste kanala **ch** se pripravi za prenos;
    - prenos se začne;
    - podatki iz sporočila se shranjujejo v spremenljivke **var$_1$, ... , var$_n$**

Primer uporabe kanalov in operacij

```
chan input(char), output(char [MAXLINE]);
process Char_to_Line {
  char line[MAXLINE]; int i = 0;
  while (true) {
    receive input(line[i]);
    while (line[i] != CR and i < MAXLINE) {
      # line[0:i-1] contains the last i input characters
      i = i+1;
      receive input(line[i]);
    }
    line[i] = EOL;
    send output(line);
    i = 0;
  }
}
```

Program sprejema znake iz enega kanala, jih zloži v vrstico in jih pošlje v izhodni kanal – filter.

Operacije nad kanali

- Operaciji **`send`** in **`receive`** sta atomarni:
  - **`send`** je analogen **`V`**(kanal kot semafor)
  - **`receive`** je analogen **`P`**(kanal kot semafor)

- Pošiljanje in sprejemanje sporočil poteka po principu FIFO – sporočila so sprejeta v istem vrstnem redu kot so poslana.

- Dodatna operacija:

```
bool empty(ch);
```

  - funkcija vrne **`True`**, če kanal ch ne vsebuje sporočil, sicer vrne **`False`**
  - ob uporabi se lahko zgodi Race Condition:
    - npr. funkcija vrne **`False`** (torej kanal vsebuje sporočila), vendar so v tem času sporočila že predana sprejemnikom in postane vrsta sporočil v kanalu prazna. Pozor!

Princip odjemalec/strežnik

- Odjemalec je aktivni proces:
  - Pošilja sporočila – zahteve procesu strežnika preko kanala za zahteve.
  - Čaka, da strežnik zahtevo izpolni, in da prejme rezultat, ki ga pošlje strežnik nazaj.
  - Odziv strežnika se prenaša po kanalu za odzive.

- Strežnik je proces, ki se odziva na zahteve odjemalcev:
  - Čaka na zahteve odjemalcev, ki prihajajo preko kanala zahtev.
  - Sprejete zahteve obdela in proizvede odgovor.
  - Odgovor pošlje po kanalu za odziv, ki si ga deli z odjemalcem.

- V splošnem lahko rečemo, da je odziv strežnika na zahtevo odjemalca funkcija zahteve in stanja strežnika.

Izvedba odjemalca in strežnika

- Vsaka zahteva odjemalca se preslika v operacijo - zaporedje ukazov za izvedbo te zahteve.
- Proces strežnika se izvaja v eni niti. Operacije izvaja zaporedoma glede na tip operacije, ki je v zahtevi. Zato tudi en kanal za zahteve.
- Odjemalcev je lahko več, zato tudi kanalov za odgovor več. Strežnik v nekem trenutku streže samo enemu odjemalcu.

```
type op_kind = enum(op1, ..., opn);
type arg_type = union(arg1, ..., argn);
type result_type = union(res1, ..., resn);
chan request(int clientID, op_kind, arg_type);
chan reply[n](res_type);
process Server {
  int clientID; op_kind kind; arg_type args;
  res_type results; declarations of other variables;
  initialization code;
  while (true) {    ## loop invariant MI
    receive request(clientID, kind, args);
    if (kind == op1)
      { body of op1; }
    ...
    else if (kind == opn)
      { body of opn; }
    send reply[clientID](results);
  }
}
process Client[i = 0 to n-1] {
  arg_type myargs; result_type myresults;
  place value arguments in myargs;
  send request(i, opj, myargs);     # "call" opj
  receive reply[i](myresults);      # wait for reply
}
```

Primer uporabe odjemalca/strežnika: dodeljevanje skupnih virov

- Dodeljevanje skupnih virov:
  - odjemalci zahtevajo delež skupnega vira (pomnilnika, …);
  - ga uporabijo;
  - ko končajo, ga vrnejo upravljavcu virov (strežniku) nazaj;
  - osnovna verzija: odjemalci lahko zahtevajo in sproščajo skupni vir posamično (samo ena nit procesa)
  - skupni vire vodimo v neki podatkovni strukturi in do njih dostopamo z operacijama **`insert/remove`**
- Delovanje:
  - po principu odjemalec/strežnik
  - zahteva/odgovor
  - dva tipa operacij: **`ACQUIRE/RELEASE`**
  - preslikajo se v operaciji **`remove/ insert`** , ki dostopajo do skupnega vira
  - imamo še vrsto **`pending`**, v kateri so shranjene zahteve po skupnem viru, ki se ne morejo takoj izvršiti.

```
type op_kind = enum(ACQUIRE, RELEASE);
chan request(int clientID, op_kind kind, int unitid);
chan reply[n](int unitID);
process Allocator {
  int avail = MAXUNITS; set units = initial values;
  queue pending;  # initially empty
  int clientID, unitID; op_kind kind;
  declarations of other local variables;
  while (true) {
    receive request(clientID, kind, unitID);
    if (kind == ACQUIRE) {
       if (avail > 0) {  # honor request now
          avail--; remove(units, unitID);
          send reply[clientID](unitID);
       } else   # remember request
          insert(pending, clientID);
    } else {   # kind == RELEASE
       if empty(pending) {  # return unitID to units
          avail++; insert(units, unitid);
       } else {  # allocate unitID to a waiting client
          remove(pending, clientID);
          send reply[clientID](unitID);
       }
    }
  }
}
```

Primer uporabe odjemalca/strežnika: dodeljevanje skupnih virov - odjemalec

```
process Client[i = 0 to n-1] {
  int unitID;
  send request(i, ACQUIRE, 0)     # "call" request
  receive reply[i](unitID);
  # use resource unitID, then release it
  send request(i, RELEASE, unitID);
   ...
}
```

- Odjemalec
  - če pošlje sporočilo, v katerem zahteva nekaj prostih enot skupnega vira, čaka, da dobi te enote,
  - če pošlje sporočilo, v katerem sprošča enote skupnega vira, nič ne čaka.
- Vsak odjemalec ima
  - svoj kanal za odgovor.
  - en kanal za zahtevo.

Drugi primer uporabe odjemalca/strežnika: datotečni strežnik

- Primer **večnitnega strežnika** in primer **komunikacije v seji**.
- Datotečni strežnik je lahko del porazdeljenega datotečnega sistema:
  - omogoča dostop odjemalcev do datotek, ki so na disku strežnika (lahko so porazdeljene po več diskih strežniškega sistema).
- Zahteve odjemalcev, ki so posredovane strežniku:
  - `OPEN` – odjemalec želi dostopati do datoteke – po odobritvi zahteve se odpre tudi komunikacijska seja,
  - `READ/WRITE` – datoteka se bere oziroma piše,
  - `CLOSE` – datoteka se zapre – konča se tudi komunikacijska seja
- Strežnik deluje večnitno:
  - Vsak proces obdeluje zahteve ene seje odjemalca.
  - Seja se konča, ko strežnik sprejme zahtevo `CLOSE` od odjemalca.

Procesi in kanali datotečnega strežnika

- **n** procesov strežnika
  - **process File_Server[n]**
    - Strežnik lahko streže n odjemalcem in dostopa do **n** datotek hkrati.
- Kanali z zahtevami:
  - En kanal za vsako zahtevo **OPEN**:
    - **chan open(fname, clientID)**
  - **n** kanalov za zahteve **READ/WRITE/CLOSE** po dostopu do datotek:
    - **chan access[n](kind, args)**
- Kanali z odgovori:
  - **m** kanalov za odgovor na zahteve **OPEN** (en na odjemalca)
    - **chan open_reply[m](serverID)**
  - **m** kanalov za rezultate po dostopanju do datotek (en na odjemalca)
    - **chan access_reply[m](results)**

Interakcija: datotečni strežnik - odjemalec

- Komunikacijska seja:
  - seja se odpre z zahtevo odjemalca po dostopanju do datoteke:
  - Odjemalec:

```
send open(fname, myid);
receive open_reply[myid](serverID);
```

  - Strežnik - proces **myid** streže odjemalca s **clientID**:

```
receive open(fname, clientID);
open the file;
if (successful)
    send open_reply[clientID](myid);
```

  - Seja: izmenjava zahtev po dostopanju podatkov iz datotek in pošiljanjem rezultatov dostopanja:
    - interakcija zahteva/odgovor
    - en odjemalec/en proces strežnika z identifikacijskima številkam **clientID** in **serverID**

Izvedba datotečnega strežnika

```
type kind = enum(READ, WRITE, CLOSE);
chan open(string fname; int clientID);
chan access[n](int kind, types of other arguments);
chan open_reply[m](int serverID);  # server id or error
chan access_reply[m](types of results);  # data, error, ...

process File_Server[i = 0 to n-1] {
  string fname; int clientID;
  kind k; variables for other arguments;
  bool more = false;
  variables for local buffer, cache, etc.;
  while (true) {
    receive open(fname, clientID);
    open file fname; if successful then:
    send open_reply[clientID](i); more = true;
    while (more) {
      receive access[i](k, other arguments);
      if (k == READ)
        process read request;
      else if (k == WRITE)
        process write request;
      else   # k == CLOSE
        { close the file; more = false; }
      send access_reply[clientID](results);
    }
  }
}
```

Izvedba datotečnega odjemalca

```
process Client[j = 0 to m-1] {
  int serverID;  declarations of other variables;
  send open("foo", j);     # open file "foo"
  receive open_reply[j](serverID);  # get back server id
  # use file then close it by executing the following
  send access[serverID](access arguments);
  receive access_reply[j](results);
    ...
}
```

Izboljšave datotečnega strežnika

- Izboljšave datotečnega strežnika:
  - v našem primeru lahko strežemo fiksnem številu odjemalcem
  - alternativa: izvedba strežnika, da streže toliko odjemalcem, kot jih je (število procesov ni navzgor omejeno)
    - kompleksna izvedba strežnika
  - alternativa: datotečnih strežnikov je toliko kot je diskov na strežniku:
    - kompleksna izvedba strežnika ali pa odjemalcev

- Drugačen pristop: NFS (Sun Network File System)
  - dostopanje do datotek je izpeljano s klici oddaljenih procedur,
  - podobno kot pri navadnem dostopanju do datotek na lokalnem disku, le da se to dogaja na oddaljenem disku:
    - zahteva po dostopu do datoteke je v bistvu zahteva po pridobitvi deskriptorja datoteke (file handle v NFS) in statusa datoteke (pisanje/branje)
    - pri vsakem dostopanju se ti (dodatni) podatki prenašajo v argumentih klicev oddaljenih procedur – dodatno prenašanje podatkov, vendar bolj enostavne izvedbe strežnika in odjemalcev
    - v primeru nenadnih prekinitev sistemov:
      - sesutje strežnika – odjemalec poskuša pošiljati zahteve, dokler ne dobi odgovora
      - sesutje odjemalca – strežniku ni potrebno storiti nič .

Komunikacija med enakovrednimi procesi

- Obojestranska komunikacija med enakovrednimi procesi:
  - komunikacija med strežniki,
  - komunikacija med odjemalci,
  - komunikacija P2P.

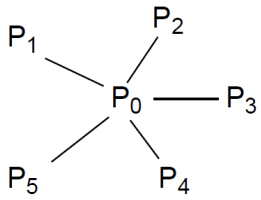
- Aplikacije, ki uporabljajo ta princip:
  - porazdeljene zgoščene tabele (DHT)
  - souporaba datotek (file sharing, torrent-i)
  - servisi P2P (DNS)
  - P2P na grid-ih
  - mrežne igre

Primer: izmenjava vrednosti spremenljivk

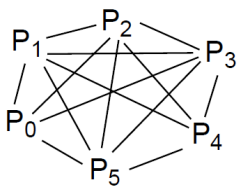
- Tipični problem interakcije med enakovrednimi.
- Imamo, n procesov
  - vsak vključuje neko številko, ki jo želi izmenjati z vsemi ostalimi procesi.
- Naloga:
  - vsak proces mora ugotoviti najmanjšo in največjo vrednost med števili, ki si jih delijo procesi med seboj.
- Izvedba:
  - s pošiljanjem sporočil,
  - tehtanje med učinkovitostjo in številom izmenjanih sporočil.

Možne rešitve

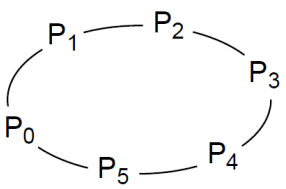
- **Centralizirana rešitev**: koordinator v središču zbira vsa sporočila, naredi primerjavo in sporoči vsem procesom nazaj največjo in najmanjšo vrednost.
- **Simetrična rešitev**: vsak proces se poveže z vsemi ostalimi in tako pridobi informacijo o največjem in najmanjšem številu.
- **Krožna rešitev**: procesi se povežejo v krog in informacije potujejo po krogu toliko časa, dokler vsak izmed procesov ne dobi podatkov o največjem in najmanjšem številu.



centralizirana rešitev



simetrična rešitev



krožna rešitev

Centralizirano reševanje

- En proces - koordinator – zbere vse vrednosti, naredi primerjavo in pošlje rešitev vsem procesom nazaj.
- Skupaj **2(n-1)** sporočil in **n** kanalov:
  - če lahko izvedemo sporočilo tipa “broadcast” imamo samo **n** različnih sporočil.

```
chan values(int), results[n](int smallest, int largest);
process P[0] {   # coordinator process
  int v;   # assume v has been initialized
  int new, smallest = v, largest = v;  # initial state
  # gather values and save the smallest and largest
  for [i = 1 to n-1] {
    receive values(new);
    if (new < smallest)
      smallest = new;
    if (new > largest)
      largest = new;
  }
  # send the results to the other processes
  for [i = 1 to n-1]
    send results[i](smallest, largest)
}
process P[i = 1 to n-1] {
  int v;   # assume v has been initialized
  int smallest, largest;
  send values(v);
  receive results[i](smallest, largest);
}
```

Simetrična rešitev

- Vsak proces
  - pošlje svojo vrednost vsem drugim,
  - zbere vse vrednosti od drugih procesov,
  - naredi primerjavo in določi največjo in najmanjšo vrednost.
- Skupaj:
  - **n(n-1)** sporočil in **n** kanalov
  - če imamo sporočila tipa “broadcast”, potem imamo **n** sporočil

```
chan values[n](int);
process P[i = 0 to n-1] {
  int v;   # assume v has been initialized
  int new, smallest = v, largest = v;  # initial state
  # send my value to the other processes
  for [j = 0 to n-1 st j != i]
    send values[j](v);
  # gather values and save the smallest and largest
  for [j = 1 to n-1] {
    receive values[i](new);
    if (new < smallest)
      smallest = new;
    if (new > largest)
      largest = new;
  }
}
```

Krožna rešitev

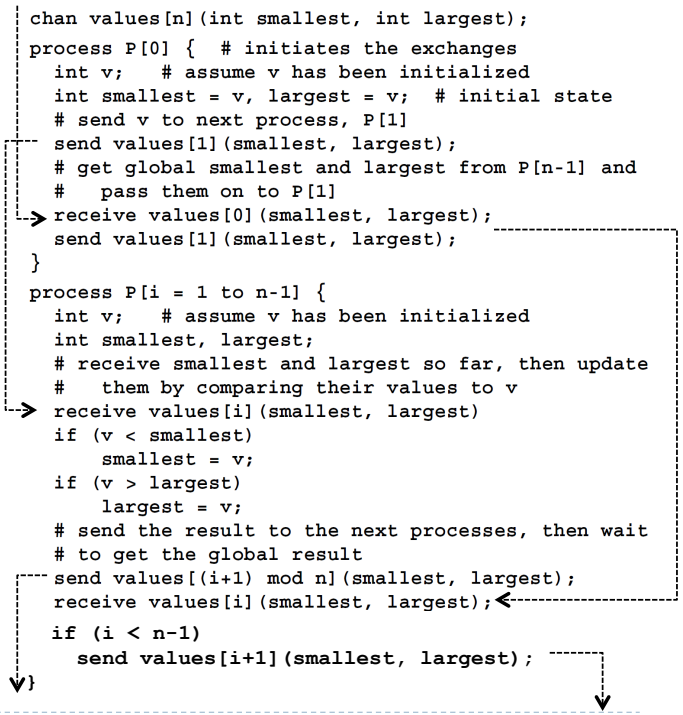
- Dva kroga izmenjave sporočil:
  - 1. krog: določimo minimum in maksimum, kot bi ga reševali v pipi; rešitev dobimo šele, ko pridemo do zadnjega procesa v pipi;
  - 2. krog: sporočimo rešitev vsem procesom.

- Delo procesa P[i] v prvem krogu:
  - sprejme dve vrednosti (trenutni min in max) od svojega predhodnika – procesorja P[i-1];
  - primerja ti dve vrednosti s svojo vrednostjo in ustrezno popravi min ali max;
  - pošlje popravljeni min in max svojemu nasledniku – procesorju P[(i+1) mod n];
  - potrebno se je izogniti **smrtnemu objemu** (spomnimo se problema petih filozofov) – P[0] najprej pošlje, šele potem prejme podatke.

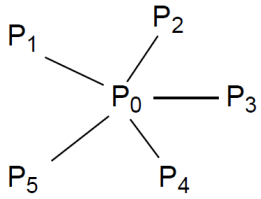
- Delo procesa P[i] v drugem krogu:
  - sprejme dve vrednosti (končni min in max) od svojega predhodnika - procesorja P[i-1];
  - obe vrednosti shrani in ju pošlje naprej k svojemu nasledniku – procesorju P[(i+1) mod n];

Krožna rešitev

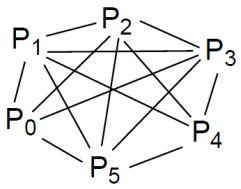
- Skupaj:
  - **2n-1** sporočil in **n** kanalov
  - sporočila tipa “broadcast” niso možna



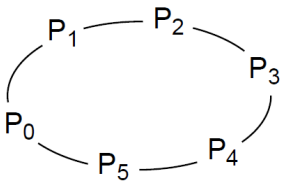
Ponovimo: interakcija med enakovrednimi procesi



centralizirana rešitev



simetrična rešitev



krožna rešitev

- Komunikacija vsakega z vsakim – simetrična rešitev – **O(n^2)** sporočil, procesi so enostavni za programirati (vsi imajo vse informacije)
- Rešitev v obliki zvezde ali kroga – **O(n)** sporočil
- Krožna rešitev je lahko neučinkovita:
  - potrebno je izvesti dva kroga izmenjave sporočil, da dobimo skupno globalno rešitev.

Sinhrona komunikacija z izmenjavo sporočili

- Obe operaciji **`send`** in **`receive`** zaustavita delovanje procesov, dokler s prenos sporočila ne izvede:
  - operacija **`send`** zaustavi proces, ki pošilja sporočilo, dokler to sporočilo ni sprejeto z operacijo receive procesa sprejemnika.
    - operacija:
      `synch_send ch(expr`$_1$ **`, …, expr`$_n$**`);`**
  - operacija **`receive`** zaustavi proces sprejemnika, dokler ne sprejeme celotno sporočilo procesa oddajnika.
- Kanali za izmenjavo sporočil so lahko implementirani brez vmesnih medpomnilnikov za shranjevanje podatkov sporočil.
- Sinhrona komunikacija je bil predlagana s programskim jezikom CSP (1978, Tony Hoare).

Omejitve pri sinhroni komunikaciji

- Primer: komunikacija med proizvajalcem in porabnikom s sinhrono izmenjavo sporočil:
  - denimo, da proizvajalec in porabnik dosežeta točko komunikacije ob različnih trenutkih.
  - Potem se morata čakati. To poveča skupen čas izvajanja procesov. **Omejena sočasnost.**

```
channel values(int);

process Producer {
  int data[n];
  for [i = 0 to n-1] {
    do some computation;
    synch_send values(data[i]);
  }
}

process Consumer {
  int results[n];
  for [i = 0 to n-1] {
    receive values(results[i]);
    do some computation;
  }
}
```

- Drugi primer: sinhrona komunikacija v primeru odjemalca in strežnika
  - nepotrebne zaustavitve izvajanja:
    - ko odjemalec sporoči, da ne potrebuje več skupnega vira, ni nobenega razloga, da čaka, da strežnik dobi to sporočilo.
    - Podobno: ko odjemalec pošlje zahtevo za pisanje v nek zunanji vir (datoteko, na grafični zaslon, …), ki ga kontrolira strežnik, ni potrebno, da čakamo, da je strežnik prejel zahtevo, ampak samo da se enkrat v prihodnosti ta zahteva izvede.

Druga slabost: možnost smrtnega objema

- Primer:
  - dva procesa izmenjujeta svoje vrednosti.

```
channel in1(int), in2(int);

process P1 {
  int value1 = 1, value2;
  synch_send in2(value1);
  receive in1(value2);
}

process P2 {
  int value1, value2 = 2;
  synch_send in1(value2);
  receive in2(value1);
}
```

- Smrtni objem:
  - oba procesa sta zaustavljena na operaciji **`synch_send`**
  - rešitev: v enem procesu zamenjamo vrstni red operaciji **`send`** in **`receive`**.
  - To moramo paziti pri vseh načinih sinhronega komuniciranja: ko en proces izvede **`synch_send`** mora drugi proces izvesti operacije **`receive`**.
  - Pri asinhroni komunikaciji lahko pustimo takšno simetrično rešitev.

Primerjava sinhrone in asinhrone komunikacije

- Prednosti in slabosti asinhrone izmenjave sporočil:
  - bolj enostavno programiranje
  - dovoljuje več sočasnosti izvajanja porazdeljenih procesov, manj situacij za smrtni objem
  - Toda: predvideva neomejen prostor za shranjevanje sporočil oziroma podatkov zunaj dosega pomnilnika posameznega procesa
    - strojna in programska izvedba je zato bolj zahtevna
    - količina prostora potrebnega za shranjevanje sporočil v kanalih je odvisna od komunikacije in se spreminja, zato je potrebno dodeljevanje prostora za izvedbo kanalov organizirati dinamično.
- Prednosti in slabosti sinhrone izmenjave sporočil:
  - Kanali so lahko implementirani brez vmesnikov za shranjevanje sporočil:
    - podatki se prenašajo direktno iz pomnilnika oddajnika v pomnilnik sprejemnika, ni potrebno izvajati nobenega “medpomnenja” in kopiranja
  - Toda:
    - omejena sočasnost izvajanja
    - večja nevarnost za situacije smrtnega objema; vedno je potrebno zagotoviti pravilni vrstni red oddajanja in sprejemanja sporočil.

7 Primer vzporednega in porazdeljenega programiranja

V tem poglavju je predstavljen primer reševanja Laplaceove diferencialne enačbe po Jacobijevi metodi s sekvenčnim, vzporednim in porazdeljenim programom. Na začetku je podan pregled, kje in zakaj izvajati paralelizacijo procesov pri reševanju računskih problemov. Potem pa je izvedena paralelizacija pri numeričnem reševanju Laplaceove diferencialne enačbe, kjer se računa rešitve enačbe na dvodimenzionalni mreži točk. Pri tem se sekvenčni program prevede na vzporedni in nato še na porazdeljeni. Algoritmi so predstavljeni v psevdo kodi, konkretne implementacije pa so izvedene v programskem jeziku C s pomočjo knjižnice PTHREADS za vzporedni program in v okolju MPI za porazdeljeni program. Knjižnica PTHREADS je opisana v dodatku A, programsko okolje MPI pa v dodatku C.

Vzporedni programi v znanosti

- Glavni namen: pohitritev postopkov za obdelovanje velikih količin podatkov
  - Pohitritev: T_1/T_n na n procesorjih
  - "Dober" sekvenčni algoritem prevedemo na vzporedni algoritem.
  - Amdahlov zakon: stopnja pohitritve je odvisna od stopnje sekvenčnosti reševanja problema. Če je stopnja ocenjena na delež *s*, potem je pohitritev omejena z <= *1/s*.
- Paralelizacija:
  - običajno podatkovni paralelizem (princip "deli in vladaj")
  - najbolj enostavno je, ko imamo neodvisne izračune znotraj zank – paraleliziramo zanke,
  - če to še ni dovolj: paralelizacija opravil znotraj zank.
- Učinkovita paralelizacija:
  - procesi naj se kreirajo enkrat, bolje da so znani vnaprej (računska farma),
  - enakomerna porazdelitev dela med procesorje ("bag of tasks"),
  - minimizacija sinhronizacije in učinkoviti algoritmi za dostopanje do kritičnih področij in izvedbo pregrad.

Tipični postopki podatkovnega paralelizma pri modeliranju

- Podatkovni paralelizem:
  - podatke razdelimo na manjše skupine (bloke, pasove, trakove, odseke);
  - vsak delavec obdeluje posamezne skupine podatkov.
- Tipični potek takega algoritma:

vzporedni program

```
postavi vse potrebno za modeliranje;
//iteriraj po času
for [t = start to finish] {
   Izračunaj/obdelaj podatke;
   BARRIER;
   Osveži podatke;   // novi podatki postanejo stari podatki
   BARRIER;
}
```

porazdeljeni program

```
postavi vse potrebno za modeliranje;
//iteriraj po času
for [t = start to finish] {
   Izračunaj/obdelaj podatke;
   IZMENJAVA SPOROČIL;
   Osveži podatke;   // novi podatki postanejo st. podatki
   IZMENJAVA SPOROČIL;
}
```

Osnovne oblike računalniškega modeliranja v znanstvenih problemih

- Računanje dogajanja nekega pojava v prostoru (mrežnih točk):
  - reševanje difierencialnih enačb, ki opisujejo spreminjanje nekega pojava v prostoru/času z metodami numeričnega računanja,
  - modeliranje pojavov v zveznih sistemih.
- Računanje dogajanja nekega pojava na posameznem delcu:
  - modeliranje spreminjanja lastnosti posameznega delca in njegovo sovplivanje na ostale delce v prostoru zaradi nekega pojava (npr. gibanje nebesnih teles v vesolju zaradi gravitacije)
  - modeliranje pojavov v diskretnih sistemih.
- Računanje matričnih operacij:
  - reševanje sistemov linearnih enačb,
  - optimizacijski problemi: linearno programiranje,
  - obdelava slik, videa, obdelava signalov.
  - izračun (matričnih) operacij na velikih količinah podatkov.

Reševanje PDE na mreži točk

- Aproksimacija modeliranja nekega pojava v prostoru (ravnini), ki se časovno spreminja:
  - prostor je mreža točk, čas se diskretno spreminja (iteracije),
  - večja natančnost modeliranja: bolj gostejša mreža točk in manjši časovni koraki.
  - Posledica: večje število operacij v določenem času, več časovnih korakov.
  - Uporaba: modeliranje vremena, klime, gibanje tekočin in plinov, …
- Paralelizacija: ne po času, ampak po podatkih (podatkovna paralelizacija)
  - mrežo točk, po kateri računamo PDE, razdelimo na posamezne bloke ali pa trakove, kjer izvedemo paralelizacijo računanja tako, da vsakemu delavcu dodelimo svojo skupino podatkov,
  - to večkrat ponovimo po času – iteracija po času (ne paraleliziramo, ker je potrebno izračunati vse rezultate za nov izračun po času).

Reševanje PDE: Laplaceva diferencialna enačba

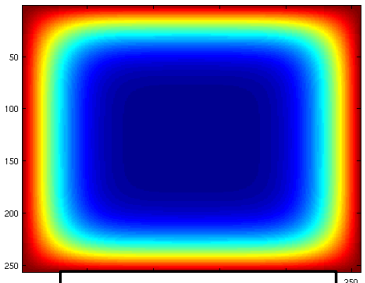
- Laplaceva diferencialna enačba z Dirichletovimi robnimi pogoji je v 2D definirana kot:

$$\frac{\partial^2 \phi}{\partial^2 x} + \frac{\partial^2 \phi}{\partial^2 y} = 0$$

  - ϕ predstavlja neko neznano količino, ki se širi po površini, npr. temperatura
  - vrednosti na robovih površine so konstantne

- Naloga je izračunati porazdelitev te količine po notranjosti površine v nekem časovnem obdobju (oziroma dokler se porazdelitev ne spreminja več).
  - predpostavke: 2D površina, iščemo (aproksimirano) numerično rešitev

- Rešitev:
  - ploskev predstavimo kot mrežo ***n x n*** točk, ki so enakomerno razporejene po ploskvi,
  - Postopek:
    - iteriramo po času,
      tako da izračunamo PDE v vsaki točki na mreži, dokler se novo izračunane vrednosti ne razlikujejo od starih vrednosti (iz prejšnjega koraka) za več kot vnaprej določen ε.



Mreža 256x256 točk, 1000 iteracij postopka.

Iterativne tehnike numeričnega izračuna

- Jacobijeva metoda:

```
newG[i,j] = 0,25 x (oldG[i,j-1]+oldG[i-1,j]+
                    oldG[i+1,j]+oldG[i,j+1])
```

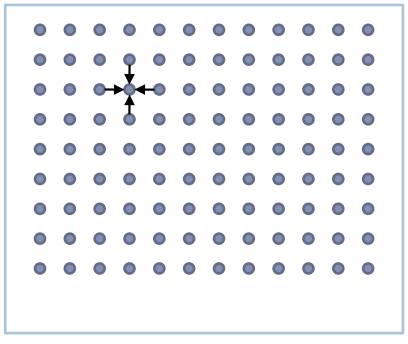
- Gauss-Seidelova metoda:

```
G[i,j] = 0,25 x (G[i,j-1]+G[i-1,j]+G[i+1,j]+G[i,j+1])
```

- Metoda SOR (*ang. successive over-relaxation*):

```
G[i,j] = ω x 0,25 x (G[i,j-1]+G[i-1,j]+G[i+1,j]+G[i,j+1])+
         (1 – ω)x G[i,j]
```

- Multigrid:
  - spreminjamo gostoto točk v mreži
    - gostota točk in s tem natančnost izračuna se spreminja od iteracije do iteracije,
    - lahko nekje bolj gosto, nekje pa ne,
    - lahko v enem koraku, v drugem pa ne



Jacobijeva metoda: sekvenčni postopek

- Oris:

```
while (true) {
  # izracunaj nove vrednosti za vse notranje tocke
  for[i=1 to n, j=1 to n]
    new[i,j] = (grid[i-1,j] + grid[i+1,j] +
                grid[i,j-1] + grid[i,j+1])/4;
  iters++;

  #izracunamo maksimalno razdaljo za pogoj za prekinitev
  maxdiff = 0.0;
  for[i=1 to n, j=1 to n]
    maxdiff = max(maxdiff, abs(new[i,j]-grid[i,j]));

  #preverimo pogoj za prekinitev
  if (maxdiff < EPSILON)
    break;

  #skopiramo nove vrednosti v stare
  for[i=1 to n, j=1 to n]
    grid[i,j] = new[i,j];
}
```

Sekvenčni postopek brez pogoja za prekinitev

```
for[iters = 1 to MAXITERS]{
  # izracunaj nove vrednosti za vse notranje tocke
  for[i=1 to n, j=1 to n]
    new[i,j] = (grid[i-1,j] + grid[i+1,j] +
                grid[i,j-1] + grid[i,j+1])/4;

  if (i == MAXITERS) break;

  #skopiramo nove vrednosti v stare
  for[i=1 to n, j=1 to n]
    grid[i,j] = new[i,j];
}

#izracunamo maksimalno napako, da javimo
maxdiff = 0.0;
for[i=1 to n, j=1 to n]
  maxdiff = max(maxdiff, abs(new[i,j]-grid[i,j]));
```

Jacobijeva metoda: izboljšani sekvenčni postopek

```
real grid[0:n+1,0:n+1], new[0:n+1,0:n+1];
real maxdiff = 0.0;
initialize the grids, including the boundaries;
for [iters = 1 to MAXITERS by 2] {
  # compute new values for all interior points
  for [i = 1 to n, j = 1 to n]
    new[i,j] = (grid[i-1,j] + grid[i+1,j] +
                grid[i,j-1] + grid[i,j+1]) * 0.25;
  # compute new values again for interior points
  for [i = 1 to n, j = 1 to n]
    grid[i,j] = (new[i-1,j] + new[i+1,j] +
                 new[i,j-1] + new[i,j+1]) * 0.25;
}
# compute the maximum difference
for [i = 1 to n, j = 1 to n]
  maxdiff = max(maxdiff, abs(grid[i,j]-new[i,j]));
print the final grid and maximum difference;
```

Vzporedni program: definicija nalog

- Paralelizacija problema:
  - Pri iterativnih postopkih v splošnem paraleliziramo izračune v zankah:
    - če je to še premalo, pogledamo če lahko paraleliziramo še naloge znotraj enega koraka iteracije
  - Lahko pa pozabimo na zanke in paraleliziramo naloge, ki jih imamo ne glede na zanke.
    - V tem primeru moramo vsekakor uporabiti pregrade, najmanj zato da lahko nadaljujemo naslednji korak iteracije.
- V našem primeru izvajamo podatkovni paralelizem:
  - razdelimo podatke v manjše skupine in vsak proces – delavec – obdela svojo skupino podatkov,
    - skupine so lahko bloki, pasovi ipd.

Vzporedni program: razdelitev nalog

- Imejmo p procesov (toliko kot procesorjev), potrebno je razdeliti n vrstic med procese: torej vsak proces naj bi obdelal n/p vrstic
  - statično dodeljevanje dela: vrstice razdelimo na p trakov, v katerih je n/p sosednjih vrstic
    - komunikacijsko breme je majhno
  - dinamično dodeljevanje dela: po principu vreče z opravili, vsak proces obdeluje eno vrstico, ko konča, vzame novo vrstico iz skupne vreče in nadaljuje z delom
    - komunikacijsko breme je večje.
- V našem primeru bomo uporabili statično dodeljevanje dela:
  - vrstice bomo razrezali na trakove glede na število procesov

Jacobijeva metoda: vzporedni program z deljenim pomnilnikom

- Primer: če je n=100, PR=4, vsak delavec obdela 25 vrstic:
  - 1. delavec (1-25), 2. delavec (26-50), …
- Število iteracij izračuna v vsakem delavcu je MAXITERS.
- Zanko v vsaki iteraciji dvakrat odvijemo, da ni potrebno izvesti prepisa novih vrednosti v stare za naslednji korak.
- Po vsaki iteraciji je potrebna pregrada, da počakamo, da vsi procesi izračunajo svoje vrstice.

```
real grid[0:n+1,0:n+1], new[0:n+1,0:n+1];
int HEIGHT = n/PR;   # assume PR evenly divides n
real maxdiff[1:PR] = ([PR] 0.0);

procedure barrier(int id) {
  # efficient barrier algorithm from Section 3.4
}

process worker[w = 1 to PR] {
  int firstRow = (w-1)*HEIGHT + 1;
  int lastRow = firstRow + HEIGHT - 1;
  real mydiff = 0.0;
  initialize my strips of grid and new, including boundaries;
  barrier(w);
  for [iters = 1 to MAXITERS by 2] {
    # compute new values for my strip
    for [i = firstRow to lastRow, j = 1 to n]
      new[i,j] = (grid[i-1,j] + grid[i+1,j] +
                  grid[i,j-1] + grid[i,j+1]) * 0.25;
    barrier(w);
    # compute new values again for my strip
    for [i = firstRow to lastRow, j = 1 to n]
      grid[i,j] = (new[i-1,j] + new[i+1,j] +
                   new[i,j-1] + new[i,j+1]) * 0.25;
    barrier(w);
  }
  # compute maximum difference for my strip
  for [i = firstRow to lastRow, j = 1 to n]
    mydiff = max(mydiff, abs(grid[i,j]-new[i,j]));
  maxdiff[w] = mydiff;
  barrier(w);
  # maximum difference is the max of the maxdiff[*]
}
```

Vzporedni program s porazdeljenim pomnilnikom

- Porazdeljeni program:
  - uporabljamo enak pristop kot prej, torej statično dodeljeno delo za vsak proces,
    - preoblikujemo vzporedni program v porazdeljenega,
    - vsak delavec obdeluje svoj trak vrstic.
  - procesi so porazdeljeni,
  - komunikacija z izmenjavo sporočil:
    - vrstice distribuiramo do delavcev, da izvajajo računanje,
    - procesi – delavci – še dodatno komunicirajo, da si sporočajo kdaj so končali oziroma sporočajo še druge rezultate.

Vzporedni program s porazdeljenim pomnilnikom

- Kaj mora dobiti vsak delavec za izračun?
  - svoj trak vrstic v matriki grid in new,
  - robove sosednjih trakov za izračun prve in zadnje vrstice v traku.
- Izračun:
  - vsak delavec izračuna nove vrednosti v svojih točkah,
  - po vsakem izračunu si morajo delavci izmenjati robne vrstice podatkov, da lahko nadaljujejo svoje delo (izmenjava podatkov, nadomesti pregrade pri vzporednem programu), vsak delavec ima zato še rezervirani prostor za eno vrstico od spodaj in eno vrstico od zgoraj,
  - na koncu si delavci izmenjajo še rezultate svojih lokalnih maksimalnih razlik med starim in novim izračunom, da lahko določimo maksimalno razliko po vsakem koraku.
- Izmenjava sporočil:

```
if (w > 1)   send up[w-1](new[1,*]);
if (w < PR)  send down[w+1](new[HEIGHT,*]);
if (w < PR)  receive up[w](new[HEIGHT+1,*]);
if (w > 1)   receive down[w](new[0,*]);
```

Porazdeljeni program: prva verzija

- Prva verzija porazdeljenega programa.

```
if (w > 1)  send up[w-1](new[1,*]);
if (w < PR) send down[w+1](new[HEIGHT,*]);
if (w < PR) receive up[w](new[HEIGHT+1,*]);
if (w > 1)  receive down[w](new[0,*]);
```

```
if (w > 1)  send up[w-1](grid[1,*]);
if (w < PR) send down[w+1](grid[HEIGHT,*]);
if (w < PR) receive up[w](grid[HEIGHT+1,*]);
if (w > 1)  receive down[w](grid[0,*]);
```

Vsak delavec izračuna svojo lokalno maksimalno razliko in jo pošlje naprej.

Delavec 1 pobere vse lokalne razlike in med njimi določi največjo.

```
chan up[1:PR](real edge[0:n+1]);
chan down[1:PR](real edge[0:n+1]);
chan diff(real);

process worker[w = 1 to PR] {
  int HEIGHT = n/PR;   # assume PR evenly divides n
  real grid[0:HEIGHT+1,0:n+1], new[0:HEIGHT+1,0:n+1];
  real mydiff = 0.0, otherdiff = 0.0;
  initialize grid and new, including boundaries;
  for [iters = 1 to MAXITERS by 2] {
    # compute new values for my strip
    for [i = 1 to HEIGHT, j = 1 to n]
      new[i,j] = (grid[i-1,j] + grid[i+1,j] +
                  grid[i,j-1] + grid[i,j+1]) * 0.25;
    exchange edges of new -- see text;
    # compute new values again for my strip
    for [i = 1 to HEIGHT, j = 1 to n]
      grid[i,j] = (new[i-1,j] + new[i+1,j] +
                   new[i,j-1] + new[i,j+1]) * 0.25;
    exchange edges of grid -- see text;
  }
  # compute maximum difference for my strip
  for [i = 1 to HEIGHT, j = 1 to n]
    mydiff = max(mydiff, abs(grid[i,j]-new[i,j]));
  if (w > 1)
    send diff(mydiff);
  else       # worker 1 collects differences
    for [i = 1 to w-1] {
      receive diff(otherdiff);
      mydiff = max(mydiff, otherdiff);
    }
  # maximum difference is value of mydiff in worker 1
}
```

Porazdeljeni program z manj komunikacije

- Manj komunikacije:
  - Ne izmenjujemo izračunanih vrednosti po vsaki iteraciji, ampak po vsaki drugi iteraciji:
    - izračunamo **new**,
    - pošljemo robove sosedom,
    - izračunamo **grid** brez robov
    - sprejmemo robove od sosedov za naslednji izračun
  - robovi se osvežijo vsak drugi korak.
- Vendar:
  - konvergenca postopka je še vedno zagotovljena.

```
chan up[1:PR](real edge[0:n+1]);
chan down[1:PR](real edge[0:n+1]);
chan diff(real);

process worker[w = 1 to PR] {
  int HEIGHT = n/PR;   # assume PR evenly divides n
  real grid[0:HEIGHT+1,0:n+1], new[0:HEIGHT+1,0:n+1];
  real mydiff = 0.0, otherdiff = 0.0;
  initialize grid and new, including boundaries;
  for [iters = 1 to MAXITERS by 2] {
    # compute new values for my strip
    for [i = 1 to HEIGHT, j = 1 to n]
      new[i,j] = (grid[i-1,j] + grid[i+1,j] +
                  grid[i,j-1] + grid[i,j+1]) * 0.25;
    # send edges of new to neighbors
    if (w > 1)
      send up[w-1](new[1,*]);
    if (w < PR)
      send down[w+1](new[HEIGHT,*]);
    # compute new values for interior of my strip
    for [i = 2 to HEIGHT-1, j = 1 to n]
      grid[i,j] = (new[i-1,j] + new[i+1,j] +
                   new[i,j-1] + new[i,j+1]) * 0.25;
    # receive edges of new from my neighbors
    if (w < PR)
      receive up[w](new[HEIGHT+1,*]);
    if (w > 1)
      receive down[w](new[0,*]);
    # compute new values for edges of my strip
    for [j = 1 to n]
      grid[1,j] = (new[0,j] + new[2,j] +
                   new[1,j-1] + new[1,j+1]) * 0.25;
    for [j = 1 to n]
      grid[HEIGHT,j] = (new[HEIGHT-1,j] +
                        new[HEIGHT+1,j] + new[HEIGHT,j-1] +
                        new[HEIGHT,j+1]) * 0.25;
  }
  compute maximum difference
}
```

Gauss-Seidlova in SOR metoda

- Obe metodi konvergirata hitrejše kot Jacobijeva metoda.
- Uporabljata kombinacijo starih in novih vrednosti (vrednosti nad in levo od trenutno točke so nove, ostale stare):
  - Gauss-Seidelova metoda:

```
G[i,j] = 0,25 x (G[i,j-1]+G[i-1,j]+G[i+1,j]+G[i,j+1])
```

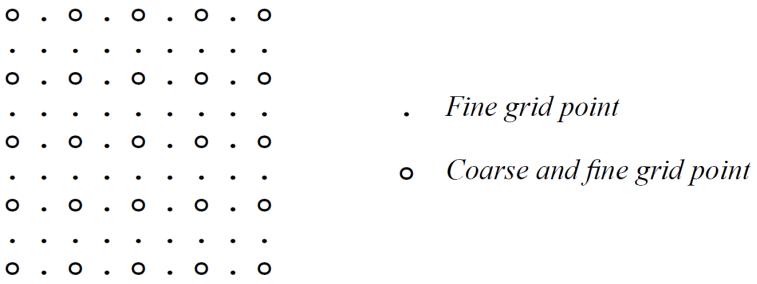
  - Metoda SOR (*ang. successive over-relaxation*):

```
G[i,j] = ω x 0,25 x (G[i,j-1]+G[i-1,j]+G[i+1,j]+G[i,j+1])+
         (1 – ω)x G[i,j]
```

- Ne potrebujemo dveh mrež.
- Vendar:
  - to lahko delamo, če računamo sekvenčno,
  - paralelizacija je bolj zahtevna:
    - barvamo točke na mreži rdeče in črno izmenično,
    - v enem koraku izračunamo rdeče točke,
    - v drugem koraku izračunamo črne točke.

Multigrid metode

- Imamo dve različno goste mreže:
  - bolj gosto mrežo točk
  - manj gosto mrežo točk
- Manj gosto mrežo interpoliramo na bolj gosto.
- Bolj gosto mrežo ekstrapoliramo na manj gosto.
- Iteriramo nekaj časa po eni, potem pod drugi mreži in se izmenjujemo.



Reševanje po principu vreče opravil

- Reševanje po principu vreče opravil:
  - P delavcev
  - N enot dela (npr. vsaka vrstica)
  - vreča opravil:
    - vsak delavec vzame eno opravilo
    - ko konča, vzame naslednje,
    - ko ni več opravil in so vsi delavci končali s svojim delom, gremo lahko v novo iteracijo
    - princip računske farme.

A Predstavitev programske knjižnice PTHREADS

V dodatku predstavimo knjižnico PTHREADS, ki jo lahko uporabljamo za vzporedno programiranje v programskem jeziku C. Predstavljeni so osnovni tip in funkcije knjižnice za delo z nitmi. Pregledamo funkcije za kreiranje in ukinitev niti, funkcije za določanje lastnosti niti, dva koncepta izvajanja niti z združevanjem in brez združevanja, mehanizme sinhronizacije med nitmi: ključavnice, pogojne spremenljivke, semaforje.

Obdelani so tudi primeri vzporednega seštevanja elementov matrike, več problemov proizvajalca in porabnika ter problem petih filozofov.

Izvorna koda je pripravljena ali povzeta po:

- knjigi *G. R. Andrews: Foundations of Multithreaded, Parallel, and Distributed Programming*, primeri na spletni strani: http://www.cs.arizona.edu/~greg/mpdbook/
- spletni strani *POSIX Threads Programming*: https://computing.llnl.gov/tutorials/pthreads/

Pthreads = POSIX Threads

- Pthreads je standardizirna zbirka funkcij za programski jezik C, ki omogočajo večnitno programiranje ob uporabi skupnega pomnilnika:
  - Standard:
    IEEE Portable Operating System Interface, POSIX, section 1003.1 standard, 1995
- Večnitno programiranje:
  - izvedba več niti v enem (HW) procesu in
  - sodelovanje med njimi,
- Glavni cilj razvoja programske zbirke Pthreads:
  - razvoj enovite zbirke orodij, ki omogočajo razvoj programov za sočasno računanje na enoprocesorskih in večprocesorskih sistemih.

Programska knjižnica Pthreads API

- Knjižnica Pthreads vključuje:
  - v osnovni verziji nad 60 funkcij
  - z razširitvami pa skupaj okoli 100 funkcij za večnitno programiranje
- Upravljanje z nitmi (38 funkcij):
  - create, exit, detach, join, get/set attributes, cancel, test cancellation, …
- Ključavnice (mutex locks, 19):
  - init, destroy, lock, unlock, try lock, get/set attributes
- Pogojne spremenljivke (condition variables, 11):
  - init, destroy, wait, timed wait, signal, broadcast, get/set attributes
- Ključavnice za branje in pisanje (read/write locks, 13):
  - init, destroy, write lock/unlock, read lock/unlock, get/set attributes
- Signali (3): pošlji signal niti, maska signalov
- Razširitve:
  - semaforji (semaphore.h)

Delo s knjižnico Pthreads

- Deklaracijske datoteke:

```
#include <pthread.h>
#include <sched.h>
#include <semaphore.h>  // za delo s semaforji
```

- Prevajanje programov s knjižnico pthreads:

```
linux$ gcc -lpthread -o test test.c
```

  - Razširitev: `-lposix4`

- Program izvajamo na običajen način:

```
linux$ ./test parameters
```

Imena spremenljivk in funkcij

Routine Prefix	Functional Group
`pthread_`	Threads themselves and miscellaneous subroutines
`pthread_attr_`	Thread attributes objects
`pthread_mutex_`	Mutexes
`pthread_mutexattr_`	Mutex attributes objects.
`pthread_cond_`	Condition variables
`pthread_condattr_`	Condition attributes objects
`pthread_key_`	Thread-specific data keys

- Tipi: `pthread`[_object][**_np**]`_t`
- Funkcije: `pthread`[_object]**_action**[_np]
- Konstante in makroji `PTHREAD`_PURPOSE[**_NP**]

- **object**: tip objekta – npr. attr, mutex, …
- **action**: operacija, ki jo izvajamo na objektu, npr. init, setscope,
- **np** ali **NP**: non-portable extension – neprenosljiva oblika izvedbe
- **PURPOSE**: namen uporabe konstante ali makroja

Osnovni tipi spremenljivk

type	Description
`pthread_attr_t`	Thread creation attribute
`pthread_cleanup_entry_np_t`	Cancelation cleanup handler entry
`pthread_condattr_t`	Condition variable creation attribute
`pthread_cond_t`	Condition Variable synchronization primitive
`pthread_joinoption_np_t`	Options structure for extensions to pthread_join()
`pthread_key_t`	Thread local storage key
`pthread_mutexattr_t`	Mutex creation attribute
`pthread_mutex_t`	Mutex (Mutual exclusion) synchronization primitive
`pthread_once_t`	Once time initialization control variable
`pthread_option_np_t`	Pthread run-time options structure
`pthread_rwlockattr_t`	Read/Write lock attribute
`pthread_rwlock_t`	Read/Write synchronization primitive
`pthread_t`	Pthread handle
`pthread_id_np_t`	Thread ID. For use as an integral type.
`struct sched_param`	Scheduling parameters (priority and policy)

Deklaracija in ustvarjanje niti

- Deklaracija: `pthread_t tid;  // nit tid`
- Funkcija za nastanek niti:

```
int pthread_create(pthread_t* tid,
                   const pthread_attr_t* attr,
                   void* (*start_routine)(void *),
                   void *arg);
```

  - Na začetku se program štarta v rutini `main`(). V primeru `pthreads` se `main`() obravnava kot nit, torej na začetku štarta program v `main`() v eni niti. Znotraj `main`() lahko kreiramo ostale niti z zgornjim ukazom.
  - Z ukazom `pthread_create` naredimo novo nit in jo že začnemo izvajati.
    Funkcija `pthread_create` se lahko kliče kjerkoli v programu in se lahko uporablja poljubno mnogokrat.
    Ko naredimo s funkcijo `pthread_create` novo nit, ta postane enakovredna vsem prejšnjim. Med nitmi ni hierarhije.

  - Argumenti funkcije `pthread_create`:
  - * `thread`: podatki o niti, ki jo naredimo, ko se izvede `pthread_create`, dobimo ID niti,
  - * `attr`: ko je attr = NULL se uporabljajo privzete nastavitve za izvedbo niti, sicer jih lahko definiramo s strukturo `attr`.
  - * `start_routine`: C-jevska rutina, ki jo nit izvaja,
  - * `arg`: argumenti rutine, ki jo nit izvaja. Argument je lahko samo en. Rutini `start_routine` ga posredujemo kot referenco na kazalec argumenta, ki mora biti tipa (`VOID` *). V primeru, če rutina ne zahteva argumentov, podamo kazalec tipa `NULL`, v primeru, če imamo več argumentov, jih moramo združiti v strukturo in rutini poslati kazalec na to strukturo.

Primer: Hello World.

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS     5

void *PrintHello(void *threadid)
{
   long tid;
   tid = (long)threadid;
   printf("Hello World! It's me, thread #%ld!\n", tid);
   pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc;
   long t;
   for(t=0; t<NUM_THREADS; t++){
      printf("In main: creating thread %ld\n", t);
      rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
      if (rc){
         printf("ERROR; return code from pthread_create() is %d\n", rc);
         exit(-1);
      }
   }
   pthread_exit(NULL);
}
```

Nastavitve **`attr`** za izvedbo niti

- Nastavitve za izvedbo niti se določijo pred klicem funkcije **`pthread_create`**
  - če jih ne določimo (**`attr==NULL`**), se izvaja nit s privzetimi nastavitvami
- Osnovna uporaba:
  - deklaracija spremenljivke attr za nastavitev niti
    - **`pthread_attr_t attr;`**
  - inicializacija spremenljivke attr
    - **`pthread_attr_init(&attr);`**
  - nastavitev vrednosti spr. attr
    - **`pthread_attr_set`*`attribute`*`(&attr, value);`**
  - brisanje spremenljivke attr (ko je ne potrebujemo več):
    - **`pthread_attr_destroy(&attr);`**

Možne nastavitve za niti

- Način izvajanje niti:
  - detachstate – pove ali je nit detached ali joinable, torej ali bo nit izvajana z možnostjo združevanja ali ne:
    - **PTHREAD_CREATE_JOINABLE, PTHREAD_CREATE_DETACHED**
- Nastavitve sklada niti:
  - stackaddr – določi naslov sklada (kazalec VOID), ki ga uporablja nit za svoje izvajanje
    - **int pthread_attr_setstackaddr(pthread_attr_t *_attr_, void *_stackaddr_);**
  - stacksize – določi minimalno potrebno količino naslovnega prostora (pomnilnika) sklada za izvajanje niti
    - **int pthread_attr_setstacksize(pthread_attr_t *_attr_, size_t _stacksize_);**
- Razvrščanje delovanja niti:
  - schedparam – določimo parametre delovanja niti, npr. prioriteto izvajanja
    - **THREAD_PRIORITY_NORMAL**
  - schedpolicy – določimo način razvrščanja delovanja niti:
    - **SCHED_OTHER** (regular, non-real-time scheduling), **SCHED_RR** (real-time, round-robin) **SCHED_FIFO** (real-time, first-in first-out))
  - inheritsched – določimo ali naj nit deduje lastnosti razvrščanja od svoje ustanoviteljice ali ne
    - **PTHREAD_EXPLICIT_SCHED, PTHREAD_INHERIT_SCHED**
  - scope – določimo način izvajanja razvrščanja niti med procesi (globalen ali lokalen)
    - **PTHREAD_SCOPE_SYSTEM (delimo razvrščanje z vsemi procesi v OS), PTHREAD_SCOPE_PROCESS (delimo razvrščanje samo s nitmi znotraj našega procesa)**

Primer: Nastavitve lastnosti niti

thread_stack_size.c

```
pthread_attr_init(&attr);
pthread_attr_getstacksize (&attr, &stacksize);
printf("Default stack size = %li\n", stacksize);
stacksize = sizeof(double)*N*N+MEGEXTRA;
printf("Amount of stack needed per thread = %li\n",stacksize);
pthread_attr_setstacksize (&attr, stacksize);
printf("Creating threads with stack size = %li bytes\n",stacksize);
```

Vhodni parametri funkcije, ki jo izvaja nit

- **`pthread_create`** dovoljuje samo en argument za funkcijo, ki jo izvaja nit:
  - argument mora biti funkciji posredovan kot referenca kazalca tipa (VOID *)
  - v primeru, če funkcija ne potrebuje parametrov je NULL v primeru, če funkcija potrebuje več argumentov, jih združimo v strukturo in pošljemo funkciji v niti kazalec na to strukturo
  - **Pomembno:** podatki argumentov funkcije, ki jo izvaja nit, in ki jih vodimo v strukturi, ostanejo v niti starša (tistega, ki je naredil to nit), ki jih zato lahko spreminja. Pazimo, da jih ne spreminjamo, dokler se izvaja v niti funkcija s temi argumenti.

Primer: Hello World.

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS     5

void *PrintHello(void *threadid)
{
   long tid;
   tid = (long)threadid;
   printf("Hello World! It's me, thread #%ld!\n", tid);
   pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc;
   long t;
   for(t=0; t<NUM_THREADS; t++){
      printf("In main: creating thread %ld\n", t);
      rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
      if (rc){
         printf("ERROR; return code from pthread_create() is %d\n", rc);
         exit(-1);
      }
   }
   pthread_exit(NULL);
}
```

Identifikacija niti

- Vsaka nit dobi ob nastanku svoj identifikacijsko številko, ki jo lahko uporabljamo za naslavljanje ali delo s to nitjo.

```
pthread_t tid;
...
pthread_create(&tid, ...);
...
pthread_join(tid, NULL);
```

- V vsaki niti lahko tudi izvemo nitin lastni id:

```
pthread_t mytid = pthread_self();
```

- Za primerjavo med dvema nitima uporabimo

```
pthread_equal(tid1, tid2);
```

  - == (ne deluje, ker je to podatkovna struktura in ne ena spr. osnovnega tipa)

Prekinitev delovanja niti

- Delovanje niti lahko prekinemo na dva načina:
  - Eksplicitno z
    `void pthread_exit(void * return_value);`
    - končamo nit z vrnjeno vrednostjo **`return_value`**
    - če **`main()`** konča s klicem funkcije **`pthread_exit()`** in ostale niti še niso končale s svojim delom, lahko z delom nadaljujejo
    - če v **`main()`** ne uporabimo **`pthread_exit()`** in **`main`** konča svoje delo (pride do konca programa), ostale niti pa tečejo, se izvajanje vseh niti prekini in program se konča
    - **`pthread_exit()`** ne zapre datotek, če so bile odprte v niti
  - Implicitno, ko se konča izvajanje funkcije v niti
    `return(status)`
    - to je enako, kot če kličemo na koncu funkcije **`pthread_exit()`**

Primer: Hello World.

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS     5

void *PrintHello(void *threadid)
{
   long tid;
   tid = (long)threadid;
   printf("Hello World! It's me, thread #%ld!\n", tid);
   pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc;
   long t;
   for(t=0; t<NUM_THREADS; t++){
      printf("In main: creating thread %ld\n", t);
      rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
      if (rc){
         printf("ERROR; return code from pthread_create() is %d\n", rc);
         exit(-1);
      }
   }
   pthread_exit(NULL);
}
```

Združevanje niti

- V niti, s katero smo kreirali druge niti, lahko čakamo na končanje teh niti z ukazom:

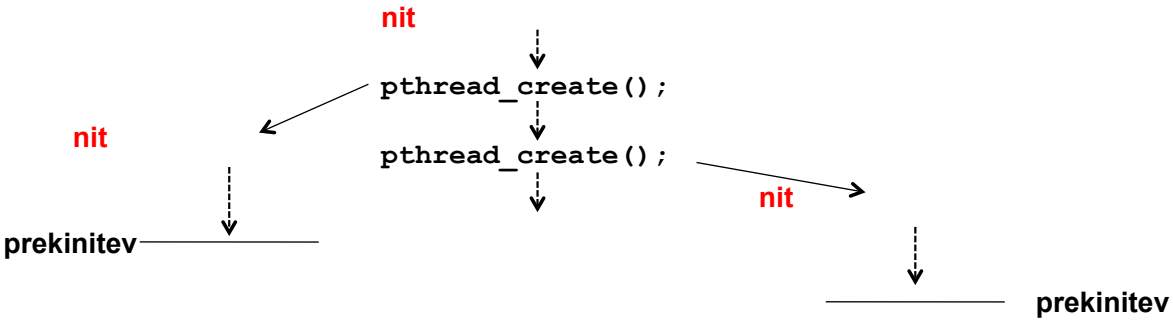
  `int pthread_join(pthread_t tid, void ** value);`

  - **`tid`** – id niti, ki jo čakamo, da konča;
  - **`value`** – naslov prostora v pomnilniku, kjer je shranjena vrednost rezultata izvajanja niti **`tid`**;
  - Nit, v kateri uporabimo ta ukaz, se pri tem ukazu ustavi, dokler se nit, ki jo čakamo ne konča.
  - Nujno je potrebno podati ID niti, ki jo čakamo. Čakanje na katerokoli nit s tem ukazom ni izvedljivo.

- Vse niti delimo na takšne, ki se lahko združujejo ali pa ne.

Niti, ki se ne združujejo

- Niti, ki jih ne združujemo, imenujemo ang. *detached threads*.
  - nit, ki ustvari novo nit, ni nujno, da to nit tudi čaka, da se konča,
  - ko se takšna nit prekine, se tudi uniči (za to ne skrbimo posebej)



- Takšne nit lahko ustvarimo na začetku s **`pthread_create`**:
  - `pthread_attr_setdetachstate(&tattr, PTHREAD_CREATE_DETACHED);`
  - `pthread_create(&tid, &tattr, ...);`
- Lahko pa jim eksplicitno določimo način izvajanja:
  - `int pthread_detach(pthread_t tid);`

Primer: detached threads

```
/* Initialize and set thread detached attribute */
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);

for(t=0;t<NUM_THREADS;t++) {
   printf("Main: creating thread %ld\n", t);
   rc = pthread_create(&thread[t], &attr, BusyWork, (void *)t);
   if (rc) {
     printf("ERROR; return code from pthread_create() is %d\n", rc);
     exit(-1);
     }
  }
```

Primer: vse skupaj

- HelloWorld.

- Program požene 10 niti.
- Vsaka nit izpiše
  - Hello from …
- Večino časa niti spijo:
  - vsaka naključno časa spi
  - s tem dosežemo prepletanje niti
- Vsaka nit deluje samostojno.

- Na koncu programa počakamo,
  - da se vse niti končajo,
  - izpišemo: Vse niti so …
  - končamo.

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <pthread.h>

#define P 10

void *sayhello (void *id)
{
  sleep(rand()%5);
  printf("Pozdrav iz niti %ld\n", (long) id);
}

int main (int argc, char *argv[]) {
  long i;
  pthread_t thread[P];
  time_t t;

  t = time(NULL);  // seed the random number
  srand((int) t);  // generator from outside

  printf("Pozdrav iz glavne niti.\n");
  for (i=0; i<P; i++) {
    pthread_create(&thread[i], NULL, sayhello, (void *)i);
  }

  for (i=0; i<P; i++) {
    pthread_join(thread[i], NULL);
  }

  printf("Vse niti so koncale. Tudi jaz bom.\n");
  exit(0);
}
```

Mehanizmi sinhronizacije v Pthreads

- **Mutex** – ključavnice, ki zagotavljajo medsebojno izključevanje izvajanja KP

- **Pogojne spremenljivke** – pogojna sinhronizacija

- **Ključavnice** za probleme **branja** in **pisanja** – zagotavljajo skupno branje in ekskluzivno pisanje skupnih podatkov

- **Semaforji** – to je razširitev standarda (stand. POSIX 1b)
  - implementacija semaforjev z mutex-i in pogojnimi spremenljivkami
  - `#include <semaphore.h>`
    pri prevajanju `-lposix4`

Mutex-i

- Mutex – podatkovna struktura tipa **`pthread_mutex_t`**
- Zagotavlja medsebojno izključevanje procesov/niti pri dostopanju do KP:
  - operacije: **`lock`**, **`unlock`**, **`trylock`**
  - preden jih uporabimo, jih je potrebno inicializirati,
  - v niti lahko izvemo/nastavimo parametre mutex-a,
  - mutex-e uničimo, ko jih ne potrebujemo več.

- Deklaracija:

```
pthread_mutex_t mutex;
```

Funkcije z mutex-i

- **`pthread_mutex_init(&mutex, &attr)`**
  - ustvarimo in inicializiramo novo ključavnico **`mutex`** s parametri **`attr`**,
  - na začetku je **`mutex`** odklenjen (unlock)
- **`pthread_mutex_destroy(mutex)`**
  - uničimo **`mutex`**, ki ga ne potrebujemo več
- **`pthread_mutex_lock(&mutex)`**
  - ključavnico **`mutex`** zaklenemo. Če je mutex že zaklenjen, se izvajanje niti zaustavi, dokler se **`mutex`** ne odklene (potem ga pa mi zaklenemo).
- **`pthread_mutex_trylock(&mutex)`**
  - poskušamo zakleniti **`mutex`**. Če je **`mutex`** zaklenjen, vrne vrednost različno od 0, če ne 0. Izvajanje niti se nadaljuje.
- **`pthread_mutex_unlock(&mutex)`**
  - odklenemo ključavnico **`mutex`** . Izvajanje niti se nadaljuje.

Delo z mutex-i

- Tipična uporaba mutex-ov v pthreads:
  - inicializacija mutex-a, …, ustvarimo niti, …, zaklenemo mutex, izvedemo KP, odklenemo mutex, …, uničimo mutex.

- Primer:

```
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, NULL);
...
pthread_mutex_lock(&mutex);

kritično področje;

pthread_mutex_unlock(&mutex);

ne-kritično področje;
```

Primer: Mutex-i

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUM_THREADS 10 /* default number of threads */

/* shared variables */
double total;
pthread_mutex_t lock;

void *Counter (void *null) {
    int i;
    double *result = (double *) calloc (1, sizeof (double));

    for (i = 0; i < 100; i++)
        *result = *result + (double) (random ()%100);

    pthread_mutex_lock (&lock);
    total += *result;
    pthread_mutex_unlock (&lock);

    pthread_exit ((void *) result);
}
```

Primer: Mutex-i

```
int main (int argc, char *argv[]) {
    int n = NUM_THREADS;
    double *result;
    pthread_t thread[NUM_THREADS];
    int t;

    if (argc > 1)
        n = atoi (argv[1]);
    if (n > NUM_THREADS || n < 1)
        n = NUM_THREADS;

    for (t = 0; t < n; t++)
        pthread_create (&thread[t], NULL, Counter, NULL);
    for (t = 0; t < n; t++) {
        pthread_join (thread[t], (void **) &result);
        printf ("Completed join with thread %d. Result =%f\n", t, *result);
    }

    printf ("The total = %f\n", total);
}
```

Pogojne spremenljivke

- Pogojna spremenljivka je definirana s strukturo **`pthread_cond_t`**.
- Uporabljamo jih za zaustavitev in ponovni zagon niti v kombinaciji z mutex-i.
  - Torej: za pogojno sinhronizacijo
  - Vedno: v kombinaciji z mutex-i
- Deklaracija:
  `pthread_cond_t cond`;
- Operacije:
  - **`wait`**, **`signal`**, **`broadcast`**
  - način izvajanja: Signal and Continue Signaling discipline

Funkcije nad pogojnimi spremenljivkami

- **`pthread_cond_init(&cond, &attr)`**
  - ustvari in inicializira novo pogojno spremenljivko cond glede na nastavitve attr
- **`pthread_cond_destroy(&cond)`**
  - uničimo pogojno spremenljivko, ki jo ne potrebujemo več
- **`pthread_cond_wait(&cond, &mutex)`**
  - čakaj, dokler pog. spr. cond ne dobi signal za nadaljevanje,
  - nit, ki izvede ukaz se zaustavi in postavi v vrsto zaustavljenih niti glede na pog. spr. cond, in sicer na zadnje mesto,
  - za uporabo te funkcije, je potrebno imeti mutex, vezan na to pogojno spr., zaklenjen, v času čakanja se ta mutex sprosti (npr. da lahko ostale niti delajo naprej).
  - Ko pogojna spr. sprejme signal, se nit zbudi in **mutex zaklene.**
- **`pthread_cond_signal(&cond)`**
  - pošljemo signal pogojni spr.,
  - prva nit, ki čaka v vrsti pogojne spr. se postavi v vrsto za izvajanje in lahko nadaljuje svoje delo.
- **`pthread_cond_broadcast(&cond)`**
  - pošljemo signal pogojni spr., da vse niti, ki čakajo na izvajanje, lahko nadaljujejo svoje delo

Delo s pogojnimi spremenljivkami

- Izvedemo neko akcijo v niti, ko je x == 0

```
action() {
  ...
  pthread_mutex_lock(&m);
  while (x != 0)
    pthread_cond_wait(&cv, &m);
  izvedi akcijo;
  pthread_mutex_unlock(&m);
  ...
}
```

```
counter() {
  ...
  pthread_mutex_lock(&m);
  x--;
  if (x == 0)
    pthread_cond_signal(&cv);
  pthread_mutex_unlock(&m);
  ...
}
```

Primer: Seštevanje elementov v matriki

$$\boldsymbol{A} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n-1} & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n-1} & a_{2n} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{n\,n-1} & a_{n\,n} \end{pmatrix}$$

Naloga: $total = \sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij}$

Sekvenčni program:

```
double A[n, n];
...
total = 0;
for (i = 1 to n)
    for (j = 1 to n)
      total += A[i][j];

write(total);
```

Primer: Vzporedno seštevanje elementov v matriki

- Seštevanje po trakovih.
- Vsak delavec sešteje elemente v enem traku.
- Delavec 0 izpiše končni rezultat.
- Potrebno je izvesti pregrado.

$$\boldsymbol{A} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n-1} & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n-1} & a_{2n} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{n\,n-1} & a_{n\,n} \end{pmatrix}$$

trak 0 – delavec 0

trak Nw – delavec Nw

Primer: Vzporedno seštevanje elementov v matriki

Rešitev v psevdo kodi

```
double A[n, n];
double sums[Nw] = ([Nw] 0);
int stripSize = n/Nw;     // predpostavimo, da je celo
število

process Worker[id= 1 to Nw] {
  int total, i, j, first, last;

  /* determine first and last rows of my strip */
  first = id*stripSize;  last = first + stripSize - 1;

  /* sum values in my strip */
  total = 0;
  for (i = first to last)
    for (j = 0 to n-1)
      total += A[i][j];
  sums[id] = total;

  Barrier();       //pocakamo, da vsi izracunajo,

  if (id == 0) {   //prvi delavec izpise koncni rezultat
    total = 0;
    for (i = 0 to n) total += sums[i];
    write("the total is %d", total);
  }
}
```

Pregrada za vzporedno seštevanje elementov v matriki

Rešitev v C s PTHREADS

```
pthread_mutex_t barrier;
pthread_cond_t go;
int numArrived = 0;

void Barrier() {
  pthread_mutex_lock(&barrier);
  numArrived++;
  if (numArrived == numWorkers) {
    numArrived = 0;
    pthread_cond_broadcast(&go);  }
  else
    pthread_cond_wait(&go, &barrier);
pthread_mutex_unlock(&barrier);}
```

```c
/* sestevanje elementov matrike z uporabo pthreads

   uporaba:
     prog_name size numWorkers

*/

#include <pthread.h>
#include <stdio.h>
#define SHARED 1
#define MAXSIZE 10000  /* maximum matrix size */
#define MAXWORKERS 20   /* maximum number of workers */

pthread_mutex_t barrier;  /* mutex lock for the barrier */
pthread_cond_t go;        /* condition variable for leaving */
int numWorkers;
int numArrived = 0;       /* number who have arrived */

/* a reusable counter barrier */
void Barrier() {

  pthread_mutex_lock(&barrier);
  numArrived++;

  if (numArrived == numWorkers) {
    numArrived = 0;
    pthread_cond_broadcast(&go);

  } else
    pthread_cond_wait(&go, &barrier);

  pthread_mutex_unlock(&barrier);
}



int size, stripSize;  /* assume size is multiple of numWorkers */
int sums[MAXWORKERS];
int matrix[MAXSIZE][MAXSIZE];
void *Worker(void *);

/* read command line, initialize, and create threads */
int main(int argc, char *argv[]) {
  long i, j;
  pthread_attr_t attr;
  pthread_t workerid[MAXWORKERS];

  /* set global thread attributes */
  pthread_attr_init(&attr);
  pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

  /* initialize mutex and condition variable */
  pthread_mutex_init(&barrier, NULL);
  pthread_cond_init(&go, NULL);

  /* read command line */
  size = atoi(argv[1]);
  numWorkers = atoi(argv[2]);
  stripSize = size/numWorkers;

  /* initialize the matrix */
  for (i = 0; i < size; i++)
    for (j = 0; j < size; j++)
      matrix[i][j] = 1;

  /* create the workers, then exit */
  for (i = 0; i < numWorkers; i++)
    pthread_create(&workerid[i], &attr, Worker, (void *) i);
```

```
67    pthread_exit(NULL);
68  }
69
70  /* Each worker sums the values in one strip of the matrix.
71     After a barrier, worker(0) computes and prints the total */
72  void *Worker(void *arg) {
73    long myid = (long) arg;
74    long total, i, j, first, last;
75
76    printf("worker %ld has started\n", myid);
77
78    /* determine first and last rows of my strip */
79    first = myid*stripSize;
80    last = first + stripSize - 1;
81
82    /* sum values in my strip */
83    total = 0;
84    for (i = first; i <= last; i++)
85      for (j = 0; j < size; j++)
86        total += matrix[i][j];
87    sums[myid] = total;
88    Barrier();
89
90    if (myid == 0) {
91      total = 0;
92      for (i = 0; i < numWorkers; i++)
93        total += sums[i];
94      printf("the total is %ld\n", total);
95    }
96  }
```

Standardne programske funkcije za varno delo z nitmi

- Standardne programske funkcije morajo biti pripravljene oziroma morajo omogočati varno delo z nitmi.
- To pomeni, da več sočasno izvajanih verzij iste standardne funkcije ne sme vplivati ena na drugo.
- Dva načina standardnih funkcij:
  - **re-entrant function:** več izvajanj iste funkcije v večnitnem programu (lahko sočasno, prepleteno, gnezdeno) ne sme imeti vpliva na posamično izvajanje.
  - **thread-safe function:** to so funkcije, ki so bodisi re-entrant ali pa je njihovo izvajanje zaščiteno z nekim mehanizmom medsebojnega izključevanja

- Da povemo, da hočemo v programu uporabiti to varno obliko standardnih funkcij (ukazov) to napišemo kot makro pred prvim include-om:

```
#ifndef _REENTRANT
#define _REENTRANT
#endif

#include <pthread.h>
#include <stdio.h>
...
```

Semaforji v PTHREADS

- Semaforji so bili v pthreads dodani kot razširitev osnovnega standarda. V knjižnici pthreads so implementirani s pogojnimi spremenljivkami in ključavnicami – mutex-i.

- Delo s semaforji:

```
#include <pthread.h>
#include <semaphore.h>
```

  - Prevajanje: `-lposix4`

- Semafor je struktura oblike **`sem_t`**, lahko zavzame poljubno nenegativno število, ki ga spreminjamo z operacijama **`P`** (zmanjšujemo) in **`V`** (povečujemo)
  - Če je vrednost semaforja 0, potem operacij **`P`** zaustavi izvajanje niti, ki je to operacijo izvedla, dokler vrednost semaforja ne postane pozitivna.

Semaforji v PTHREADS

- Deklaracija **`sem_t sem`**;
- Funkcije:
  - **`sem_wait(&sem) – P(sem)`** – zmanjšaj vrednost semaforja za 1, ko je vrednost semaforja pozitivna.
  - **`sem_post(&sem) – V(sem)`** – povečaj vrednost semaforja za 1.
- Primer:

```
sem_init(&sem, SHARED, 1);    // sem = 1
…
sem_wait(&sem);               // P(sem)
kritično področje
sem_post(&sem);               // V(sem)
```

  - kjer je:
    - **`SHARED`** (1 ali 0), ki nastavi ali je semafor **`sem`** viden za vse procese OS.
    - **`SHARED`** mora bit v primeru Linux ali Windows vedno 0.

Preprost primer P/C s semaforji mutex

```
int buf;
sem full = 0, empty = 1;

process Producer {
  while(1) {
    P(empty);
    buf = data;
    V(full);
  }
}

process Consumer {
  while(1) {
    P(full);
    result = buf;
    V(empty);
  }
}
```

```c
/* primer uporabe semaforjev v primeru preprostega P/C problema

    usage:
     pc.sems numIters

*/

#include <pthread.h>
#include <stdio.h>
#include <semaphore.h>
#define SHARED 0

void *Producer(void *);  /* the two threads */
void *Consumer(void *);

sem_t *empty, *full;    /* the global semaphores */
int buf;              /* shared buffer         */
int numIters;

/* main() -- read command line and create threads, then
             print result when the threads have quit */

int main(int argc, char *argv[]) {
  /* thread ids and attributes */
  pthread_t pid, cid;
  pthread_attr_t attr;
  pthread_attr_init(&attr);
  pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

  numIters = atoi(argv[1]);
  //sem_init(empty, SHARED, 1);  /* sem empty = 1 */
  empty = sem_open("m_empty",O_CREAT,0644,1);  /* to je isto kot zgoraj, le zaradi OSX mora biti tako */
  //sem_init(full, SHARED, 0);   /* sem full = 0  */
  full = sem_open("m_full",O_CREAT,0644,0); /* to je isto kot zgoraj, le zaradi OSX mora biti tako */

  printf("main started\n");
  pthread_create(&pid, &attr, Producer, NULL);
  pthread_create(&cid, &attr, Consumer, NULL);
  pthread_join(pid, NULL);
  pthread_join(cid, NULL);

  sem_close(empty);
  sem_close(full);

  printf("main done\n");


  return 0;
}


/* deposit 1, ..., numIters into the data buffer */
void *Producer(void *arg) {
  int produced;
  printf("Producer created\n");
  for (produced = 0; produced < numIters; produced++) {
    sem_wait(empty);
    buf = produced;
    printf("P: vstavil sem %d\n", buf); fflush(stdout);
    sem_post(full);
  }
}

/* fetch numIters items from the buffer and sum them */
void *Consumer(void *arg) {
  int total = 0, consumed;
  printf("Consumer created\n");
```

```
69    for (consumed = 0; consumed < numIters; consumed++) {
70      sem_wait(full);
71      total = total+buf;
72      printf("C: vzel sem %d\n", buf); fflush(stdout);
73      sem_post(empty);
74    }
75    printf("for %d iterations, the total is %d\n", numIters, total); fflush(stdout);
76  }
```

P/C:
več proizvajalcev in porabnikov – medpomnilnik 1 enota

- Vpeljemo dva binarna semaforja **`full`** in **`empty`**, ki tvorita sestavljeni binarni semafor.
- Samo eden izmed njiju je lahko 1:
  `0 <= full + empty <= 1`
- Če proizvajalec hoče pisati v medpomnilnik, mora čakati, da postane **`empty==1`**. Ko zapiše v **`buf`**, postavi **`full=1`**.
- Porabnik čaka, da je **`full==1`**, da lahko prebere iz **`buf`**. Ko prebere, postavi **`empty=1`**.

```
typeT buf;      /* a buffer of some type T */
sem empty = 1, full = 0;
process Producer[i = 1 to M] {
  while (true) {
    ...
    /* produce data, then deposit it in the buffer */
    P(empty);
    buf = data;
    V(full);
  }
}
process Consumer[j = 1 to N] {
  while (true) {
    /* fetch result, then consume it */
    P(full);
    result = buf;
    V(empty);
    ...
  }
}
```

pc.sems.multi.c

```c
/* primer uporabe semaforjev v primeru preprostega P/C: vec P in C

    usage:
     pc.sems.multi

*/

#include <pthread.h>
#include <stdio.h>
#include <semaphore.h>
#define SHARED 0

#define P 3
#define C 2


void *Producer(void *);  /* the two threads */
void *Consumer(void *);

sem_t *empty, *full;    /* the global semaphores */
long buf;              /* shared buffer         */
int numIters_P;
int numIters_C;

/* main() -- read command line and create threads, then
             print result when the threads have quit */

int main(int argc, char *argv[]) {

  long i;

  /* thread ids and attributes */
  pthread_t pid[P];
  pthread_t cid[C];



  pthread_attr_t attr;
  pthread_attr_init(&attr);
  pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

  numIters_P =  2;   /* da se nam izide */
  numIters_C =  3;

  //sem_init(&empty, SHARED, 1);  /* sem empty = 1 */
  empty = sem_open("m_empty",O_CREAT,0644,1);  /* to je isto kot zgoraj, le zaradi OSX mora biti tako */
  //sem_init(&full, SHARED, 0);   /* sem full = 0  */
  full = sem_open("m_full",O_CREAT,0644,0); /* to je isto kot zgoraj, le zaradi OSX mora biti tako */

  printf("main started\n"); fflush(stdout);

  /* tri proizvajalci */
  for (i=0; i<P; i++) {
    pthread_create(&pid[i], &attr, Producer, (void*)i);
  }

  /* dva porabnika */
  for (i=0; i<C; i++) {
    pthread_create(&cid[i], &attr, Consumer, (void*)i);
  }

  /*  pocakamo vse proizvajalce */
  for (i=0; i<P; i++) {
    pthread_join(pid[i], NULL);
  }

  /* pocakamo vse porabnike */
  for (i=0; i<C; i++) {
    pthread_join(cid[i], NULL);
  }
```

```
69
70    sem_close(empty);
71    sem_close(full);
72
73    printf("main done\n"); fflush(stdout);
74  }
75
76
77  /* deposit 1, ..., numIters into the data buffer */
78  void *Producer(void *id) {
79    int produced;
80
81    printf("Producer %ld created\n", (long)id ); fflush(stdout);
82    for (produced = 0 ; produced < numIters_P; produced++) {
83      sem_wait(empty);
84      buf = (long)id * 10 + produced;
85      printf("P %ld: vstavil sem %ld\n", (long) id, buf); fflush(stdout);
86      sem_post(full);
87    }
88  }
89
90  /* fetch numIters items from the buffer */
91  void *Consumer(void *id) {
92    long result = -1;
93    int consumed;
94    printf("Consumer %ld created\n", (long)id ); fflush(stdout);
95    for (consumed = 0; consumed < numIters_C; consumed++) {
96      sem_wait(full);
97      result = buf;
98      printf("C %ld: vzel sem %ld\n", (long) id, result); fflush(stdout);
99      sem_post(empty);
100
101   }
102 }
```

P/C:
več proizvajalcev in porabnikov – medpomnilnik n enot

```
typeT buf[n];       /* an array of some type T */
int front = 0, rear = 0;
sem empty = n, full = 0;      /* n-2 <= empty+full <= n */
sem mutexD = 1, mutexF = 1;  /* for mutual exclusion   */
process Producer[i = 1 to M] {
  while (true) {
    ...
    produce message data and deposit it in the buffer;
    P(empty);
    P(mutexD);
    buf[rear] = data; rear = (rear+1) % n;
    V(mutexD);
    V(full);
  }
}
process Consumer[j = 1 to N] {
  while (true) {
    fetch message result and consume it;
    P(full);
    P(mutexF);
    result = buf[front]; front = (front+1) % n;
    V(mutexF);
    V(empty);
    ...
  }
}
```

```c
/* primer uporabe semaforjev v primeru P/C: vec P in C medpomnilnik n enot

    usage:
     pc.sems.multi_boundbuf

*/

#include <pthread.h>
#include <stdio.h>
#include <semaphore.h>
#define SHARED 0

#define P 3
#define C 2
#define BUF_MAX 5

void *Producer(void *);  /* the two threads */
void *Consumer(void *);

sem_t *empty, *full;       /* the global semaphores */
sem_t *mutexD, *mutexF;    /* semaforja za buffer   */
long buf[BUF_MAX];         /* shared buffer         */
int numIters_P;
int numIters_C;
int front, rear;           /* stevca za buffer*/


/* main() */

int main(int argc, char *argv[]) {

  long i;

  /* thread ids and attributes */
  pthread_t pid[P];
  pthread_t cid[C];



  pthread_attr_t attr;
  pthread_attr_init(&attr);
  pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

  numIters_P =  2;   /* da se nam izide */
  numIters_C =  3;

  front = 0;
  rear = 0;

  //sem_init(&empty, SHARED, 1);  /* sem empty = 1 */
  empty = sem_open("m_empty",O_CREAT,0644,BUF_MAX);  /* to je isto kot zgoraj, le zaradi OSX mora biti tako */
  //sem_init(&full, SHARED, 0);   /* sem full = 0  */
  full = sem_open("m_full",O_CREAT,0644,0); /* to je isto kot zgoraj, le zaradi OSX mora biti tako */

  mutexD = sem_open("mutex_D",O_CREAT,0644,1); /* mutex za Deposit */
  mutexF = sem_open("mutex_F",O_CREAT,0644,1); /* mutex za Fetch */

  printf("main started\n"); fflush(stdout);

  /* tri proizvajalci */
  for (i=0; i<P; i++) {
    pthread_create(&pid[i], &attr, Producer, (void*)i);
  }

  /* dva porabnika */
  for (i=0; i<C; i++) {
    pthread_create(&cid[i], &attr, Consumer, (void*)i);
  }

  /*  pocakamo vse proizvajalce */
  for (i=0; i<P; i++) {
    pthread_join(pid[i], NULL);
  }

```

```
73    /* pocakamo vse porabnike */
74    for (i=0; i<C; i++) {
75      pthread_join(cid[i], NULL);
76    }
77
78    sem_close(empty);
79    sem_close(full);
80    sem_close(mutexD);
81    sem_close(mutexF);
82
83    printf("main done\n"); fflush(stdout);
84  }
85
86
87  /* deposit 1, ..., numIters into the data buffer */
88  void *Producer(void *id) {
89    int produced;
90
91    printf("Producer %ld created\n", (long)id ); fflush(stdout);
92
93    for (produced = 0 ; produced < numIters_P; produced++) {
94      sem_wait(empty);
95      sem_wait(mutexD);
96      buf[rear] = (long)id * 10 + produced;
97      printf("P %ld: vstavil sem %ld na mesto %d\n", (long) id, buf[rear], rear); fflush(stdout);
98      rear = (rear+1) % BUF_MAX;
99      sem_post(mutexD);
100     sem_post(full);
101   }
102 }
103
104 /* fetch numIters items from the buffer */
105 void *Consumer(void *id) {
106   long result = -1;
107   int consumed;
108   printf("Consumer %ld created\n", (long)id ); fflush(stdout);
109
110   for (consumed = 0; consumed < numIters_C; consumed++) {
111     sem_wait(full);
112     sem_wait(mutexF);
113     result = buf[front];
114     printf("C %ld: vzel sem %ld iz mesta %d\n", (long) id, result, front); fflush(stdout);
115     front = (front+1) % BUF_MAX;
116     sem_post(mutexF);
117     sem_post(empty);
118
119   }
120 }
```

Problem petih filozofov

- Filizofe razdelimo na tiste, ki sedijo na sodih sedežih in tiste, ki so na lihih sedežih.
- Sodi filozofi najprej vzamejo desne vilice in nato leve.
- Lihi filozofi pa ravno obratno.
- Tudi tako preprečimo smrtni objem zaradi krožnega čakanja.
- Vse rešitve lahko posplošimo na n procesov.

```
sem fork[5] = ([5] 0);
process Philosopher[i =0 to 4] {
  while(true) {
    if (i%2 == 0) {
     P(fork[(i+1)%5]); P(fork[i]);
    } else {
     P(fork[i]); P(fork[(i+1)%5]);
    }
    eat;
    V(fork[i]); V(fork[(i+1)%5]);
    think;
  }
}
```

philosophers.c

```c
/* primer uporabe semaforjev v primeru 5-ih filozofov

    usage:
     philosophers

*/

#include <pthread.h>
#include <stdio.h>
#include <semaphore.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>

#define SHARED 0

#define P 5
#define NUM_ITER 2


void *Philosopher(void *);  /* nit filozofov */
sem_t *m_fork[P];             /* semaforji za vilice */



/* main() */

int main(int argc, char *argv[]) {

  long i;
  char ime_semaforja[10];



  /* thread ids and attributes */
  pthread_t pid[P];

  // inicializiramo vse semaforje
  // ker MAC OSX uporablja named semforje, moramo tako kompliciratи,
  // sicer bi uporabili sem_init(m_fork[i], SHARED, 1)
  for (i=0; i<P; i++) {
        sprintf(ime_semaforja, "mm_fork_%ld", i);
        m_fork[i] = sem_open(ime_semaforja,O_CREAT,0644,1);
  }


  printf("main started\n"); fflush(stdout);

  /* 5 filozofov */
  for (i=0; i<P; i++) {
    pthread_create(&pid[i], NULL, Philosopher, (void*)i);
  }

  /*  pocakamo vse filozofe */
  for (i=0; i<P; i++) {
    pthread_join(pid[i], NULL);
  }

  for (i=0; i<P; i++) {
    sem_close(m_fork[i]);
  }



  printf("main done\n"); fflush(stdout);
```

```
67  }
68
69
70  /* Filozof */
71  void *Philosopher(void *id) {
72    int iter;
73    long myid;
74
75    myid = (long)id;
76
77    //printf("Folozof %ld narejen ...\n", myid); fflush(stdout);
78
79    for (iter = 0 ; iter < NUM_ITER; iter++) {
80          if (myid %2 == 0) {
81                sem_wait(m_fork[myid]);
82                sem_wait(m_fork[(myid+1)%P]);
83          }
84          else {
85                sem_wait(m_fork[myid]);
86                sem_wait(m_fork[(myid+1)%P]);
87          }
88          printf("Filozof %ld sedaj je...\n", myid); fflush(stdout);
89
90          sem_post(m_fork[myid]);
91          sem_post(m_fork[(myid+1)%P]);
92
93          printf("Filozof %ld sedaj razmislja...\n", myid); fflush(stdout);
94          sleep(1);
95    }
96  }
97
98
```

B Osnove večnitnega programiranja v JAVI

V dodatku predstavimo programski jezik JAVA za vzporedno programiranje. Predstavimo tvorjenje vzporednih procesov z nitmi in mehanizme medsebojne sinhronizacije med nitmi v javi. Uporaba večnitnega programiranja je prikazana na na več problemih proizvajalec/porabnik in v primeru problema branja in pisanja, ki ga rešujemo na različne načine.

Izvorna koda je pripravljena ali povzeta po:

- knjigi *G. R. Andrews: Foundations of Multithreaded, Parallel, and Distributed Programming*, primeri na spletni strani: http://www.cs.arizona.edu/~greg/mpdbook/
- spletni strani *The Java Tutorials: Concurrency*: http://docs.oracle.com/javase/tutorial/essential/concurrency/

Programski jezik JAVA - osnove

- JAVA je objektni programski jezik s sintakso podobno programskemu jeziku C.
- Program v JAVI je sestavljen iz:
  - zbirke razredov (ang. *classes*) in objektov (konkretnih izvedb razredov)
  - Programiranje v JAVI: deklaracija razredov in vmesnikov (ang. *interfaces*)
- Razred:
  - definiramo spremenljivke in metode, ki uporabljajo te spremenljivke,
  - razred lahko izpeljemo samo iz enega razreda (super razred), toda iz več vmesnikov
  - vsaka konkretna izvedba razreda, ki ga uporabljamo v programu, je objekt
- Primarni razred (*ang. primary class*) je razred, kjer je statično deklarirana metoda main:
  - Vsaka java aplikacija se začne v metodi main.
  - Primarni razred se najprej naloži v pomnilnik za izvajanje, metoda main se kliče prva.
  - Vhodni argumenti metode main so parametri iz ukazne vrstice zagona programa.

Programski jezik JAVA - osnove

- **Interface** (vmesnik) je zbirka deklaracij metod brez vsebine.
  - Metode so samo deklarirane, izvedba programskega dela metode je prepuščena razredom, ki implementirajo ta vmesnik.
  - V vmesniku so lahko definirane konstante.
  - Razred lahko implementira enega ali več takšnih vmesnikov.
- **Object** (objekt) je konkretna uporaba razreda v programu.
  - Za izvedbo razreda v objekt uporabimo operator **`new`**.
    - **`MyClass obj = new MyClass(parameters);`**
  - Dostop do spremenljivk in metod objekta poteka po referenci (npr. **`obj.var`**).
  - Z metodami spreminjamo stanje spremenljivk v objektu.

Niti v JAVI

- Niti so definirane kot LW procesi:
  - imajo svoj sklad in kontekst izvajanja,
  - lahko dostopajo do vseh spremenljivk v svojem dosegu, (*scope*)

- Niti v javi programiramo z razširitvijo razreda **`Thread`** oziroma z implementacijo vmesnika **`Runnable.`**
  - ključni metodi za zagon in izvajanje niti sta **`start`** in **`run.`**

- Obe strukturi sta vgrajeni v standardni java knjižnjici: java.lang.

Programiranje niti v JAVI

- Nit ustvarimo z ukazom
  - `Thread foo = new Thread();  //konstruktor za nit`

- Nit zaženemo z ukazom
  - `foo.start();`
    - start je samo ena izmed metod v razredu **`Thread`**
    - s tem, ko poženemo nit **`foo`**, se ne zgodi nič, ker metoda **`start()`** kliče metodo **`run()`** v razredu **`Thread`**. Ta pa v razredu **`Thread`** ne dela nič.

- Zato, moramo implementirati metodo run iz razreda **`Thread`**:
  - 1. način: z razširitvijo razreda **`Thread`**
  - 2. način: z implementacijo metode **`run`** v vmesniku **`Runnable`**

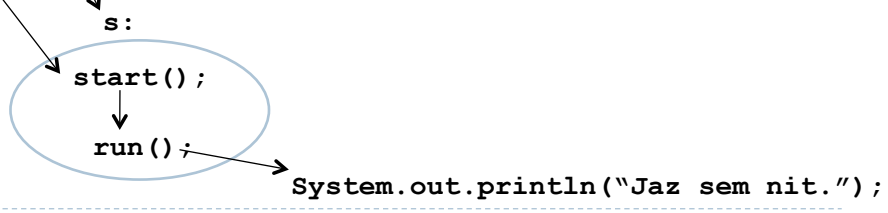
Programiranje niti v JAVI: 1. način

- 1. način: z razširitvijo razreda **`Thread`**:
  - Definiramo nov (pod)razred iz razreda **`Thread`**, tako da na novo definiramo metodo run in ostale metode iz razreda **`Thread`**, če je potrebno.
- Primer:

```
class Simple extends Thread {
  public void run() {
    System.out.println("Jaz sem nit.");
  }
}
```

- Uporaba:

```
Simple s = new Simple();
s.start();     // kličemo metodo run v objektu s
```



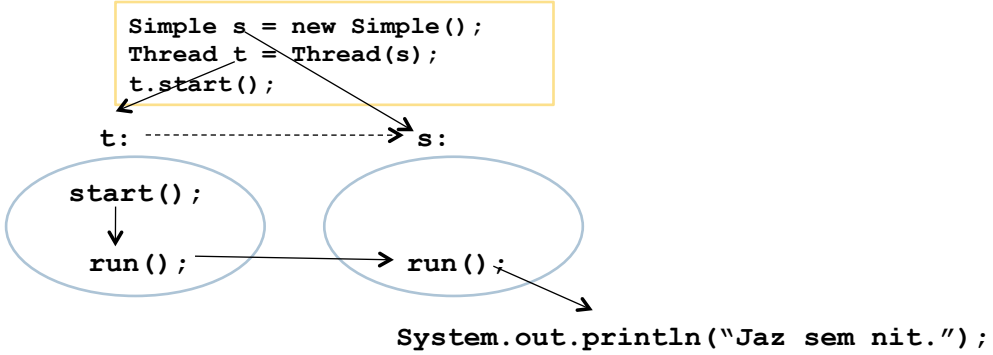
Programiranje niti v JAVI: 2. način

- Implementacija metode **`run`** vmesnika **`Runnable`** v svojem razredu.
- Primer:

```
class Simple implements Runnable {
  public void run() {
    System.out.println("Jaz sem nit.");
  }
}
```

- V tem primeru ustvarimo nit tako, da najprej ustvarimo objekt iz tega razreda in ga pošljemo kot parameter za konstruktor iz razreda **`Thread`**. Nit zaženemo podobno kot prej z metodo start().

```
Simple s = new Simple();
Thread t = Thread(s);
t.start();
```



Programiranje niti v JAVI: povzetek

- 1. definiraj nov razred bodisi tako, da razširiš razred **`Thread`** oziroma implemenitraš vmesnik **`Runnable`**
- 2. sprogramiraj metodo **`run`** v novem razredu: to je program, ki ga bo nit izvajala.
- 3. ustvari objekt iz novega razreda z operatorjem **`new`**
- 4. zaženi izvajanje niti z metodo **`start`**.

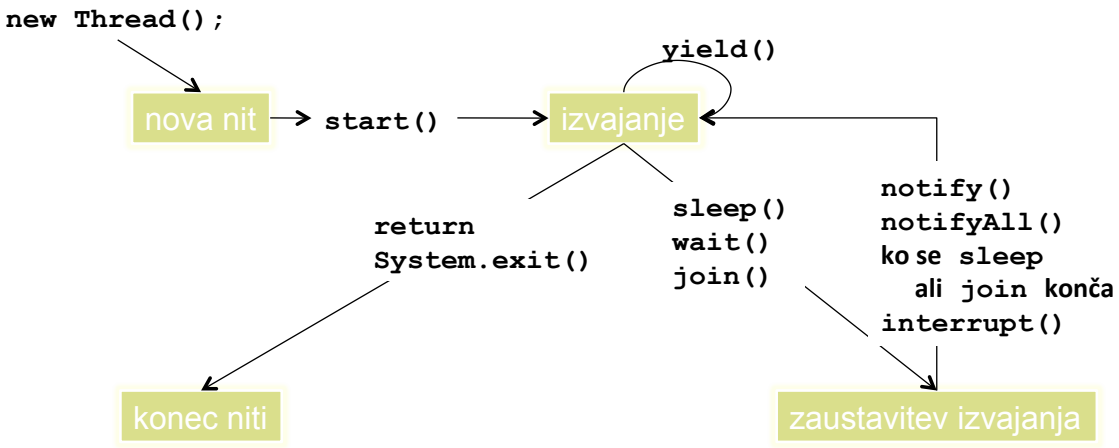
Konstruktorji razreda Thread in lastnosti niti

- Vsaka nit se v javi ustvari s svojim imenom, pripada določeni skupini niti in ima svojo prioriteto. Konstruktorji niti so:
  - `Thread()`
  - `Thread(String)`
  - `Thread(ThreadGroup, String)`
  - `Thread(Runnable)`
  - `Thread(ThreadGroup, Runnable)`
  - `Thread(Runnable, String)`
  - `Thread(ThreadGroup, Runnable, String)`
- Lastnosti niti se lahko nastavijo ali preberejo z metodami iz razreda Thread:
  - ime, prioriteta, skupina, tip niti (daemon)
  - Skupino in tip niti ne moremo spreminjati med izvajanjem, ostale lastnosti lahko.
- Stopnje prioritete:
  - `MAX_PRIORITY`
  - `MIN_PRIORITY`
  - `NORM_PRIORITY`

Metode razreda Threads

- `run()`
  - To je metoda, ki jo na novo definiramo s svojo kodo, ki se bo izvajala v niti. Če jo ne, ne naredi nič.
  - Ne kličemo jo direktno, ampak z metodo start.
- `start()`
  - Začni z izvajanjem niti: JVM začne izvajati ukaze metode run.
- `join()`
  - Počakaj, da se ta nit konča.
- `yield()`
  - Povzroči kontekstni preklop.
- `sleep(long)`
  - Nit zaustavi svoje izvajanje za čas (v milisekundah), ki je podan s parametrom tipa long.
- `interrupt()`
  - Prekini z izvajanjem niti.
- Nastavitev/poizvedba po lastnostih niti:
  - `setPriority(int)`, `getPriority()`
  - `setName(String)`, `getName()`,
  - `setDaemon(boolean)`, `isDaemon()`

Izvajanje niti in kako delujejo njene metode



Komunikacije med nitmi v JAVI

- Niti se v JAVI izvajajo sočasno – vsaj konceptualno.
- Komunikacija med nitmi:
  - dostopanje do metod in spremenljivk v skupnem dosegu je omogočeno vsem nitim iz tega dosega;
  - komunikacija preko pip;
  - komunikacija preko skupnih (deljenih) objektov.
- Objekt je skupen več sočasno tekočim nitim takrat, ko več niti lahko uporablja metode tega objekta za dostopanje do (skupnih) spremenljivk objekta.
  - Razred lahko sprogramiramo kot monitor, če vse ali nekaj metod objekta deklariramo s tipom **`synchronized`**, kar pomeni, da se lahko te metode izvajajo z upoštevanjem medsebojnega izključevanja.
  - Razred lahko vključuje metode tipa **`synchronized`** in/ali pa običajne metode – te se ne izvajajo z upoštevanjem medsebojnega izključevanja.

Metode in ukazi tipa **`synchronized`**

- Beseda **`synchronized`** je rezervirana za definiranje medsebojnega izključevanja:
  - metod, če je metoda definirana s tipom **`synchronized`**,
  - zaporedja ukazov, ki so zajeti v ukazu **`synchronized`**.
- Vsak objekt, ki uporablja metode (ukaze) tipa **`synchronized`** pridobi svojo lastno ključavnico (da se lahko izvaja medsebojno izključevanje izvajanja)
- Izvajanje metode ali zaporedja ukazov tipa **`synchronized`** je sledeče:
  - metoda (ali zaporedje ukazov) počaka, da pridobi ključavnico;
  - metoda (ali zaporedje ukazov) se izvede;
  - ko konča, se ključavnica sprosti;
- Torej:
  - metoda tipa **`synhronized`** – procedura v monitorju
  - zaporedje ukazov, ki so zajeti v **`synhronized`** – ukaz **`await`**

Primeri metode in ukazov tipa `synchronized`

```
class Interfere {
  private int data = 0;
  public synchronized void update() {
    data++;
  }
}
```

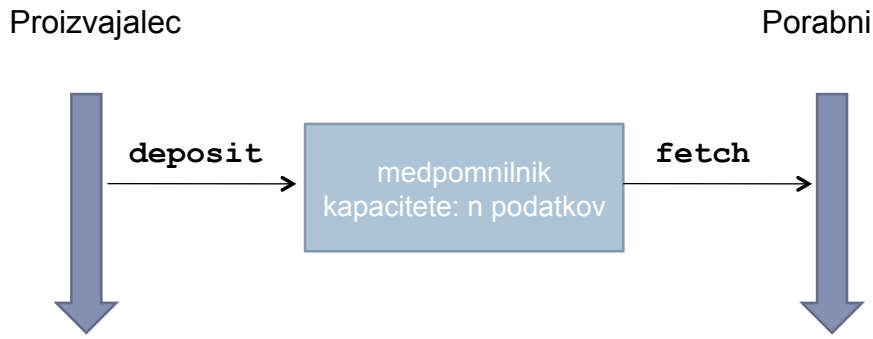
```
class Interfere {
  private int data = 0;
  public void update() {
    synchronized(this)     // lock this object
    data++;
  }
}
```

Monitorji v JAVI

- Monitorji v javi so objekti konkretnih izvedb razredov, ki imajo metode definirane tipa synchronized:
  - kar pomeni, da se v večnitnem programu, takšne metode izvajajo največ ena hkrati,
  - razredi imajo lahko metode tipa synchronized ali pa tudi ne
- Vsak objekt – monitor – pridobi eno ključavnico in eno pogojno spremenljivko:
  - ključavnice in pogojne spremenljivke ni potrebno eksplicitno definirati, ampak se ustvarijo ob samem zagonu niti ali programa
- Operacije s pogojno spremenljivko so naslednje:
  - v kombinaciji z metodami tipa synchronized:
    - wait() – monitorjeva procedura wait, sprosti ključavnico in postavi nit v vrsto za čakanje izvajanja (vrsta FIFO)
    - notify() – monitorjeva procedura signal, zbudi nit, ki je prva v vrsti za čakanje in jo postavi v vrsto za izvajanje
    - notifyall() – monitorjeva procedura signal_all, vse niti ki čakajo zbudi in jih postavi v vrsto za izvajanje
  - Pogojna spremenljivka v JAVI ni implementirana s FIFO vrsto, ki omogoča urejanje po rangu (prioriteti).
  - Način signalizacije v JAVI: Signal & Continue
- Dovoljeni so gnezdeni klici znotraj monitorjev, ti so zaprtega tipa
  - Lahko privede do smrtnega objema: metoda tipa synchronized kliče metodo tipa synchronized v drugem objektu, ta pa nazaj.

Primer: Sinhronizacija dostopanja do omejenega medpomnilnika med proizvajalcem in porabnikom

- To je primer, ki smo ga že obravnavali. Gre za princip pogojne sinhronizacije pri dostopanju do skupnega medpomnilnika, ki je prostorsko omejen.
- Denimo, da imamo na razpolago medpomnilnik s kapaciteto hranjenja **`n`** podatkov.



Implementacija monitorja za uporabo omejenega medpomnilnika

Implementirajmo medpomnilnik za komunikacijo med proizvajalcem in porabnikom kot monitor.

• Monitorjeve spremenljivke:

`buf[n], front, rare, count` (šteje, koliko mest v medpomnilniku imamo trenutno zasedenih)

`cond not_full` – zaustavi polnjenje medpomnilnika s strani proizvajalca dokler medpomnilnik ni pripravljen za pisanje (ni prost)

`cond not_empty` – zaustavi porabnika, da ne bere podatkov iz medpomnilnika, dokler ni novih podatkov v medpomnilniku

• Procedure v monitorju:

`deposit(typeT)` – vstavi podatek v medpomnilnik (proizvajalec)

`fetch(*typeT)` – zapiši podatek iz medpomnilnika v spr. tipa **`typeT`** (porabnik)

```
monitor Bounded_Buffer {

  typeT buf[n];      # an array of some type T
  int front = 0,     # index of first full slot
      rear = 0;      # index of first empty slot
      count = 0;     # number of full slots
  ## rear == (front + count) % n
  cond not_full,     # signaled when count < n
       not_empty;    # signaled when count > 0

  procedure deposit(typeT data) {
    while (count == n) wait(not_full);
    buf[rear] = data; rear = (rear+1) % n; count++;
    signal(not_empty);
  }

  procedure fetch(typeT &result) {
    while (count == 0) wait(not_empty);
    result = buf[front]; front = (front+1) % n; count--;
    signal(not_full);
  }
}
```

Uporaba:
- Proizvajalec: **`Bounded_Buffer.deposit(a[i]);`**
- Porabnik: **`Bounded_Buffer.fetch(&b[i]);`**

```
1  //  Producer/Consumer
2  //
3  // Usage:
4  //         javac PC_BoundedBuffer.java
5  //         java Exchange table_length
6
7
8
9  class BoundedBuffer {
10     private int[] buf;
11     private int n, count = 0, front = 0, rear = 0;
12
13     public BoundedBuffer(int n) {
14         this.n = n;
15         buf = new int[n];
16     }
17
18
19     public synchronized void deposit(int obj) {
20         while (count == n)
21             try { wait(); }
22             catch (InterruptedException e) { }
23         buf[rear] = obj; rear = (rear + 1) % n; count++;
24         System.out.println ("Deposited value = "+ obj); System.out.flush();
25         notifyAll();
26     }
27
28     public synchronized int fetch() {
29         while (count == 0)
30             try { wait(); }
31             catch (InterruptedException e) { }
32         int obj = buf[front];
33         front = (front + 1) % n; count--;



34         System.out.println ("Fetched value = "+ obj); System.out.flush();
35         notifyAll();
36         return obj;
37     }
38 }
39
40
41 class Producer extends Thread {
42
43     private BoundedBuffer buf;
44     private int[] a;
45
46
47     public Producer (int[] a, BoundedBuffer buf) {
48         this.buf = buf;
49         this.a = a;
50     }
51
52
53     public void run () {
54         for (int i=0; i < a.length; i++)
55             buf.deposit(a[i]);
56     }
57 }
58
59
60 class Consumer extends Thread {
61     private BoundedBuffer buf;
62     public int[] b;
63
64     public Consumer(int[] b, BoundedBuffer buf) {
65         this.buf = buf;
66         this.b = b;
```

```
67      }
68
69      public void run () {
70          for (int i = 0; i < b.length; i++)
71              b[i] = buf.fetch();
72      }
73  }
74
75
76
77
78  class Exchange {
79
80
81      public static void main(String args[]) {
82
83          int[] a;
84          int[] b;
85
86          int n = Integer.parseInt(args[0], 10);
87
88          a = new int[n];
89          b = new int[n];
90
91          System.out.println("Data a:");
92          for(int i = 0; i < n; i++) {
93              a[i] = (int) (Math.random () * 100);
94              System.out.print (a[i]+" ");
95          }
96
97          System.out.println(); System.out.flush();
98
99
100         BoundedBuffer buf = new BoundedBuffer (5);
101         Producer p = new Producer (a, buf);
102         Consumer c = new Consumer (b, buf);
103
104         p.start ();
105         c.start ();
106
107         try {
108             c.join ();
109         } catch (InterruptedException e) { };
110
111         b = c.b;
112         System.out.println("Data b:");
113         for(int i = 0; i < n; i++)
114             System.out.print (b[i]+" ");
115         System.out.println(); System.out.flush();
116
117
118     }
119 }
```

Primer: pisanje in branje podatkov iz baze

- Program za branje in pisanje podatkov iz baze / v bazo.
- Baza je samo ena celoštevilska spremenljivka.

- Prva verzija:
  - sekvenčni program, izmenoma beremo in pišemo podatke.

  - Spremenljivka **`data`** je tipa **`protected`**, kar pomeni, da je vidna samo znotraj razreda, v katerem je deklarirana oziroma v razredih, ki so znotraj istega paketa oziroma so izpeljani iz tega razreda.

```
1  // a simple, unsynchronized database
2
3  // usage:
4  //    javac rw.seq.java  (this file)
5  //    java Main rounds
6
7
8  class RWbasic {  // basic read or write (no synchronization)
9    protected int data = 0;  // the "database"
10
11   public void read() {
12     System.out.println("read:  " + data);
13   }
14
15   public void write() {
16     data++;
17     System.out.println("wrote:  " + data);
18   }
19 }
20
21 class Main {  // driver program
22   static RWbasic RW = new RWbasic();
23
24   public static void main(String[] arg) {
25     int rounds = Integer.parseInt(arg[0],10);
26     for (int i = 0; i < rounds; i++) {
27       RW.read();
28       RW.write();
29     }
30   }
31 }
```

Vzporedno pisanje in branje podatkov iz baze

- Vzporedni program za branje in pisanje podatkov iz baze / v bazo.
- Baza je samo ena celoštevilska spremenljivka.

- Druga verzija:
  - vzporedni program za pisanje in branje,
  - slaba verzija, ne preprečujemo medsebojnega izključevanja,
  - samo za primer, kako naredimo večnitni program v javi.

```
1  //  Readers/Writers with parallel access
2  //
3  // Usage:
4  //         javac rw.par.java
5  //         java Main rounds
6  //
7
8  class RWbasic { // basic read or write; no exclusion
9    protected int data = 0;  // the "database"
10   protected void read() {
11     System.out.println("read:  " + data);
12   }
13   protected void write() {
14     data++;
15     System.out.println("wrote:  " + data);
16   }
17 }
18
19 class Reader extends Thread {
20   int rounds;
21   RWbasic RW;
22   public Reader(int rounds, RWbasic RW) {
23     this.rounds = rounds;
24     this.RW = RW;
25   }
26   public void run() {
27     for (int i = 0; i<rounds; i++) {
28       RW.read();
29     }
30   }
31 }
32
33 class Writer extends Thread {
34   int rounds;
35   RWbasic RW;
36   public Writer(int rounds, RWbasic RW) {
37     this.rounds = rounds;
38     this.RW = RW;
39   }
40   public void run() {
41     for (int i = 0; i<rounds; i++) {
42       RW.write();
43     }
44   }
45 }
46
47 class Main {  // driver program
48   static RWbasic RW = new RWbasic();
49   public static void main(String[] arg) {
50     int rounds = Integer.parseInt(arg[0],10);
51     new Reader(rounds, RW).start();
52     new Writer(rounds, RW).start();
53   }
54 }
```

Vzporedno pisanje in branje podatkov iz baze

- Vzporedni program za branje in pisanje podatkov iz baze / v bazo.
- Baza je samo ena celoštevilska spremenljivka.

- Tretja verzija:
  - vzporedni program za pisanje in branje,
  - izvajamo medsebojno izključevanje s tem, da v izpeljemo nov razred **`RWexclusive`** iz razreda **`RWbasic`**, tako da deklariramo metodi **`read`** in **`write`** tipa **`synchronized`**.
  - ustrezno popravimo tudi razrede **`Reader`**, **`Writer`** in **`Main`**, da bodo uporabljali **`RWexclusive`** namesto **`RWbasic`**
  - to ni prava verzija, ker se vse operacije, torej pisanje in branje, izvajajo z medsebojnim izključevanjem

```
1  //  Readers/Writers with exclusive access.
2  //
3  // Usage:
4  //         javac rw.exclusive.java
5  //         java Main rounds
6
7  class RWbasic {         // basic read or write (data; no synch)
8    protected int data = 0;  // the "database"
9    protected void read() {
10     System.out.println("read:  " + data);
11   }
12   protected void write() {
13     data++;
14     System.out.println("wrote:  " + data);
15   }
16 }
17
18 class RWexclusive extends RWbasic {  // exclusive read and write
19   public synchronized void read() {
20     System.out.println("read:  " + data);
21   }
22   public synchronized void write() {
23     data++;
24     System.out.println("wrote:  " + data);
25   }
26 }
27
28 class Reader extends Thread {
29   int rounds;
30   RWexclusive RW;
31   public Reader(int rounds, RWexclusive RW) {
32     this.rounds = rounds;
33     this.RW = RW;
34   }
35   public void run() {
36     for (int i = 0; i<rounds; i++) {
37       RW.read();
38       try { sleep(10); }
39           catch (InterruptedException ex) {return;}
40     }
41   }
42 }
43
44 class Writer extends Thread {
45   int rounds;
46   RWexclusive RW;
47   public Writer(int rounds, RWexclusive RW) {
48     this.rounds = rounds;
49     this.RW = RW;
50   }
51   public void run() {
52     for (int i = 0; i<rounds; i++) {
53       RW.write();
54       try { sleep(5); }
55           catch (InterruptedException ex) {return;}
56     }
57   }
58 }
59
60 class Main {  // driver program
61   static RWexclusive RW = new RWexclusive();
62   public static void main(String[] arg) {
63     int rounds = Integer.parseInt(arg[0],10);
64
65     new Reader(rounds, RW).start();
66     new Reader(rounds, RW).start();
```

```
67      new Writer(rounds, RW).start();
68
69    }
70  }
```

Vzporedno pisanje in branje podatkov iz baze

- Četrta – končna – verzija:
  - tako kot želimo delujoči vzporedni program za pisanje in branje podatkov,
  - lahko se izvaja sočasno branje podatkov iz baze in ekskluzivno pisanje podatkov v bazo
  - izpeljemo nov razred **`ReadersWriters`** iz razreda **`RWbasic`** in ustrezno popravimo razrede **`Reader`**, **`Writer`** in **`Main`**, da uporabljajo novi razred.
  - V razredu **`ReadersWriters`**
    - odvzamemo ekskluzivno izvajanje metode read,
    - dodamo dve novi metodi **`startRead`** in **`endRead`**, uporabljamo ju pred in po operaciji branja podatkov,
      - metoda **`startRead`** povečuje spr. **`nr`**, ki pove koliko bralcev bere podatke,
      - metoda **`endRead`** zmanjšuje spr. **`nr`**, ko je **`nr == 0`**, to sporoči potencialnim zapisovalcem podatkov (če kakšen sploh čaka)
      - metode **`startRead`**, **`endRead`** in **`write`** so tipa **`synchronized`**, kar pomeni, da se njihovo izvajanje izključuje

```
1  //  Readers/Writers with concurrent read or exclusive write
2  //
3  // Usage:
4  //         javac rw.real.java
5  //         java Main rounds
6
7
8  class ReadersWriters{  // Readers/Writers
9    protected int data = 0; // the "database"
10
11   int nr = 0;
12   private synchronized void startRead() {
13     nr++;
14   }
15
16   private synchronized void endRead() {
17     nr--;
18     if (nr==0) notify();  // awaken waiting Writers
19   }
20
21   public void read() {
22     startRead();
23     System.out.println("read:  " + data);
24     endRead();
25   }
26
27   public synchronized void write() {
28     while (nr>0)
29       try { wait(); }
30         catch (InterruptedException ex) {return;}
31     data++;
32     System.out.println("wrote:  " + data);
33     notify();    // awaken another waiting Writer
34   }
35 }
36
37 class Reader extends Thread {  //reader thread
38   int rounds;
39   ReadersWriters RW;
40   public Reader(int rounds, ReadersWriters RW) {
41     this.rounds = rounds;
42     this.RW = RW;
43   }
44   public void run() {
45     for (int i = 0; i<rounds; i++) {
46       RW.read();
47       try { sleep(10); }
48         catch (InterruptedException ex) {return;}
49     }
50   }
51 }
52
53 class Writer extends Thread {  //writer thread
54   int rounds;
55   ReadersWriters RW;
56   public Writer(int rounds, ReadersWriters RW) {
57     this.rounds = rounds;
58     this.RW = RW;
59   }
60   public void run() {
61     for (int i = 0; i<rounds; i++) {
62       RW.write();
63       try { sleep(5); }
64         catch (InterruptedException ex) {return;}
65     }
66   }
```

```
67  }
68
69  class Main {  // driver program -- two readers and one writer
70    static ReadersWriters RW = new ReadersWriters();
71    public static void main(String[] arg) {
72      int rounds = Integer.parseInt(arg[0],10);
73      new Reader(rounds, RW).start();
74      new Reader(rounds, RW).start();
75      new Writer(rounds, RW).start();
76    }
77  }
```

C Uvod v MPI

V dodatku predstavimo osnovne koncepte porazdeljenega programiranja s pošiljanjem sporočil v programskem okolju MPI. Pregledamo osnovne tipe spremenljivk in deklaracije funkcij v knjižnici MPI, spoznamo sinhrono in asinhrono komunikacijo z ukazoma `MPI_Send` in `MPI_Recv` ter operacije za skupinsko komunikacijo. Primeri uporabe porazdeljenega programiranja so izvedeni na primeru preproste izmenjave vrednosti med porazdeljenimi procesi, na primeru izračuna decimalk števila π in v primeru preproste računske farme.

Viri in literatura, po katerih je pripravljeno poglavje, so:

- *G. R. Andrews: Foundations of Multithreaded, Parallel, and Distributed Programming*,
- *P. S. Pacheco: A Users Guide to MPI*,
- spletna stran *The Message Passing Interface (MPI) standard*: `http://www.mcs.anl.gov/research/projects/mpi/`

MPI: Message Passing Interface

- Knjižnica MPI je namenjena delu s porazdeljenimi procesi na večporcesorskih ali večračunalniških arhitekturah (cluster, …):
  - implementirana je komunikacija med procesi z izmenjavo sporočil - API,
  - implementacija API-ja in komunikacije v knjižnici,
  - možne izvedbe porazdeljenih programov s knjižnico MPI v programskih jezikih **C** in **Fortran**

- Namen:
  - omogočiti razvoj programov za vzporedno in porazdeljeno računanje,
  - omogočiti izvedbo porazdeljenih programov na različnih porazdeljenih računalniških sistemih in različnih komunikacijskih protokolih,
  - MPI postaja standard za porazdeljeno računanje.

- Implementacije MPI:
  - MPICH: splošna izvedba, omogočena izvedba na različnih računalniških platformah – izkorišča prednosti posameznih računalniških platform (najbolj razširjena)
    - UNIX, Windows
    - deluje: porazdeljeni in deljeni pomnilnik, različne topologije računalnikov, različne povezave med računalniki
  - MPICH2: nova verzija MPICH,
  - LAM: namenjena uporabi na računalniških sistemih, ki komunicirajo preko TCP/IP
  - Obstajajo še druge.

Knjižnica MPI

- Knjižnica MPI vključuje več kot 130 funkcij, ki omogočajo komunikacije med procesi:
  - z izmenjavo sporočil (eden-drugemu, skupinsko)
  - združevanje procesov v skupine in različne topologije.

- Po drugi strani lahko samo s 6-imi osnovnimi funkcijami izvajamo porazdeljeno porgramiranje:
  - **`MPI_Init, MPI_Finalize, MPI_Comm_size, MPI_Comm_rank, MPI_Send, MPI_Recv`**
  - oziroma drugih 6:
    - **`MPI_Init, MPI_Finalize, MPI_Comm_size, MPI_Comm_rank, MPI_Bcast, MPI_Reduce`**

- Vse funkcije in podatkovne strukture knjižnice MPI imajo predpono **`MPI_`**

Tipi spremenljivk v MPI in povezava s tipi v C

Podatkovni tipi v MPI	Podatkovni tipi v C
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	8 binary digits
MPI_PACKED	data packed or unpacked with MPI_Pack()/ MPI_Unpack

MPI procesi

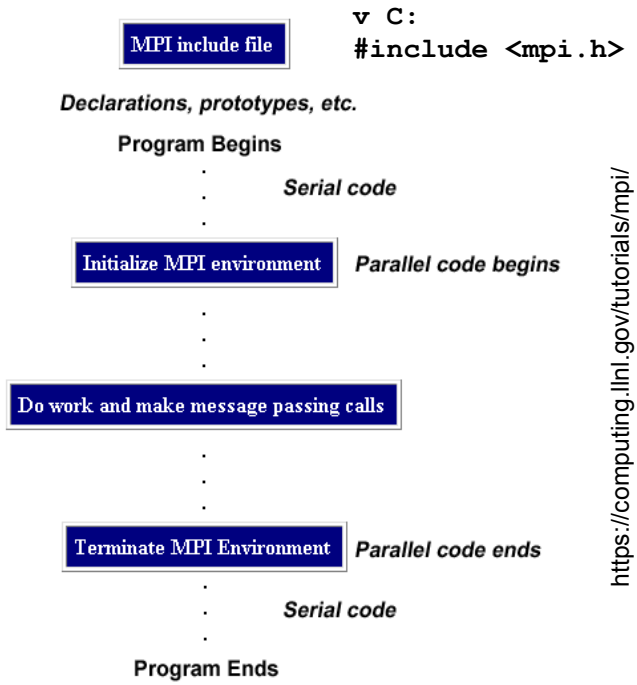
- MPI proces:
  - Naš proces – izvedba dela programa na enem procesorju (računalniku).
  - Vsak proces ima svoj ID, svoj rang in pripada neki skupini procesov.
  - Procesi lahko komunicirajo samo s procesi iz iste skupine, za katere skrbi en komunikator. Komunikacija znotraj komunikatorja (kontekst komunikatorja) poteka z izmenjavo sporočil.
  - Proces lahko pošilja sporočila sinhrono ali asinhrono.
  - Vsak proces sprejema sporočila, ki so mu namenjena.
  - Procesi komunicirajo med seboj z uporabo funkcij send/ receive, ki sta implementirani bodisi tako, da zaustavljata procese ali pa ne.
  - Procesi lahko komunicirajo med seboj tudi s skupinsko izmenjavo sporočil (znotraj komunikatorja) z operacijami tipa broadcast, gather/scatter, barrier.

MPI procesi:
komunikator, skupina procesov, rang procesa

- Komunikator:
  - Podatkovna struktura - objekt, kjer je definirana skupina procesov, ki lahko komunicirajo med seboj. Komunikacija znotraj komunikatorja je lahko omejena na posamezne podskupine procesov:
    - Podskupina procesov rešuje neko nalogo z izmenjavo sporočil, ki poteka preko komunikacije, ki je vnaprej definirana v komunikacijskem kontekstu znotraj komunikatorja.
- Skupina procesov:
  - Skupina procesov, ki so vključeni v komunikator.
    - Procesi znotraj skupine lahko komunicirajo neposredno eden z drugim ali pa skupinsko (eden z vsemi, …).
    - Vsaka komunikacija poteka samo znotraj skupine.
    - Vsak proces ima svoj rang znotraj skupine procesov.
- Rang procesa:
  - Vsak proces ima svojo ID številko, ki določa rang procesa znotraj skupine procesov (v komunikatorju).

MPI program

- Struktura MPI programa:
- En program za vse procese.
- Število sočasno tekočih procesov predpišemo ob zagonu programa.
  - najbolje je, da sprogramiramo toliko procesov, kot jih pričakujemo ob zagonu programa (en procesor na proces).



https://computing.llnl.gov/tutorials/mpi/

MPI aplikacija

- Inštaliramo eno izmed implementacij MPI,
  - npr. MPICH
- Napišemo program za aplikacijo:
  - en program za vse procese
- Prevedemo program:
  - **`mpicc -o myprog myprog.c`**
    - Lahko tudi Makefile za večje projekte:
      - primer: **`/usr/share/doc/openmpi-doc/examples/Makefile`**
- Poženemo program:
  - **`mpirun -np 2 myprog`**
    - Z opcijo **`–help`** dobimo vse opcije **`mpirun`**.

Osnovne funkcije knjižnice MPI

- **`int MPI_Init(int *argc, char **argv[])`**
  - Zaženi MPI. Inicializiraj vse potrebno za MPI procese.
- **`int MPI_Finalize()`**
  - Zaustavi MPI. Počisti vse za seboj.
- **`int MPI_Comm_size(MPI_Comm comm,int *size)`**
  - Določi in vrne število procesov v skupini komunikatorja:
    - **`comm`** – komunikator, npr. **`MPI_COMM_WORLD`** (vsi procesi v isti skupini)
    - **`size`** – število procesov v skupini (to je tudi vrnjena vrednost)
- **`int MPI_Comm_rank(MPI_Comm comm,int *rank)`**
  - Določi in vrne rang procesa, ki se izvaja v skupini komunikatorja:
    - **`comm`** – komunikator, npr. **`MPI_COMM_WORLD`**
    - **`rank`** – rang procesa v skupini komunikatorja (to je tudi vrnjena vrednost) ima vrednost med 0 in **`size-1`**
- **`int MPI_Abort(MPI_Comm comm)`**
  - Prekine delovanje vseh procesov v skupini komunikatorja.

Primer: “Hello world!”

```c
#include "mpi.h"
#include <stdio.h>

int main (int argc, char *argv[]) {

    int  numtasks, rank, rc;

    rc = MPI_Init(&argc,&argv);
    if (rc != MPI_SUCCESS) {
        printf ("Error starting MPI program. Terminating.\n");
        MPI_Abort(MPI_COMM_WORLD, rc);
    }

    MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    printf ("Number of tasks= %d My rank= %d\n", numtasks,rank);

    /*******  do some work *******/

    MPI_Finalize();
}
```

Izhod:

```
Number of tasks= 10 My rank= 1
Number of tasks= 10 My rank= 0
Number of tasks= 10 My rank= 2
Number of tasks= 10 My rank= 3
Number of tasks= 10 My rank= 4
Number of tasks= 10 My rank= 5
Number of tasks= 10 My rank= 7
Number of tasks= 10 My rank= 8
Number of tasks= 10 My rank= 9
Number of tasks= 10 My rank= 6
```

Prevajanje in start programa:

```
user@linux:$ mpicc hello2.c -o hello2
user@linux:$ mpirun -np 10 hello2
```

Operacija **send**

- **`int MPI_Send(void *buf, int count, MPI_datatype dt, int dest, int tag, MPI_Comm comm)`**
  - Pošlje sporočilo z oznako **`tag`** procesu **`dest`**, ki je definiran znotraj skupine komunikatorja:
    - **`buf`** podatki sporočila
    - **`count`** število podatkov v sporočilu
    - **`dt`** tip podatkov v sporočilu
    - **`dest`** rang procesa, kateremu je namenjeno sporočilo
    - **`tag`** oznaka sporočila
    - **`comm`** komunikator
  - Funkcija vrne celoštevilsko vrednost, ki pomeni stanje izvedbe ukaza, najbolj pogosto **`MPI_SUCCES`**.
  - **`datatype`** je lahko katerakoli elementarni tip spremenljivke (prejšnje prosojnice), oziroma sestavljene strukture iz teh spremenljivk **`MPI_Type_contiguous MPI_Type_vector, MPI_Type_indexed, MPI_Type_struct`**

Operacija receive

- `int MPI_Recv(void *buf, int count, MPI_datatype dt, int source, int tag, MPI_Comm comm, MPI_Status *status)`
  - Sprejme sporočilo z oznako **`tag`** od procesa **`source`**, ki je definiran znotraj skupine komunikatorja:
    - **`buf`** medpomnilnik (podatkovna struktura) za shranjevanje podatkov sporočila
    - **`count`** število podatkov. ki jih bomo shranili iz sporočila
    - **`dt`** tip podatkov v sporočilu
    - **`source`** rang procesa, ki je poslal sporočilo
    - **`tag`** oznaka sporočila
    - **`comm`** komunikator
    - **`status`** vrnjeni status operacije
  - Pri sprejemanju sporočila lahko uporabljamo tudi:
    - za oznako sporočil: **`MPI_ANY_TAG`**: sprejmi vsako sporočilo,
    - za oznako ranga procesa – oddajnika: **`MPI_ANY_SOURCE`**: sprejmi sporočilo iz kateregakoli procesa, ki pošilja sporočilo

Sprejemanje sporočil

- Če uporabljamo oznake za množično sprejemanje sporočil **`MPI_ANY_TAG`** ali pa za sprejemanje sporočil iz več virov **`MPI_ANY_SOURCE`** lahko preverjamo stanje sprejemanja sporočil s strukturo **`MPI_Status`**, ki ima tri komponente: **`MPI_SOURCE`**, **`MPI_TAG`**, **`MPI_ERROR`**:

```
MPI_Status status;
MPI_Recv(..., &status);
int tag_received = status.MPI_TAG;
int rank_of_source = status.MPI_SOURCE;
MPI_Get_count(&status, datatype, &count);
```

  - **`MPI_Get_count`** pove, koliko sporočil tipa **`datatype`** smo že prejeli.

Primer: izmenjava vrednosti polja med štirimi procesi

```c
#include "mpi.h"
#include <stdio.h>
#define SIZE 4

int main (int argc, char *argv[]) {

    int numtasks, rank, source=0, dest, tag=1, i;
    float a[SIZE][SIZE] =
        {1.0, 2.0, 3.0, 4.0,
         5.0, 6.0, 7.0, 8.0,
         9.0, 10.0, 11.0, 12.0,
         13.0, 14.0, 15.0, 16.0};

    float b[SIZE];

    MPI_Status stat;
    MPI_Datatype rowtype;

    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

    MPI_Type_contiguous(SIZE, MPI_FLOAT, &rowtype);
    MPI_Type_commit(&rowtype);

    if (numtasks == SIZE) {
      if (rank == 0) {
         for (i=0; i<numtasks; i++)
           MPI_Send(&a[i][0], 1, rowtype, i, tag, MPI_COMM_WORLD);
         }

       MPI_Recv(b, SIZE, MPI_FLOAT, source, tag, MPI_COMM_WORLD, &stat);
       printf("rank= %d  b= %3.1f %3.1f %3.1f %3.1f\n",
              rank,b[0],b[1],b[2],b[3]);
       }
     else
       printf("Must specify %d processors. Terminating.\n",SIZE);

    MPI_Type_free(&rowtype);
    MPI_Finalize();
}
```

MPI_ Type_contiguous

```
count = 4;
MPI_Type_contiguous(count, MPI_FLOAT, &rowtype);
```

1.0	2.0	3.0	4.0
5.0	6.0	7.0	8.0
9.0	10.0	11.0	12.0
13.0	14.0	15.0	16.0

a[4][4]

```
MPI_Send(&a[2][0], 1, rowtype, dest, tag, comm);
```

9.0	10.0	11.0	12.0

1 element of rowtype

Primer: izmenjava sporočil (ki vedno uspe)

```c
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

if (rank == 0) {
   myvalue = 14;
   MPI_Send(&myvalue, 1, MPI_INT, 1, tag, MPI_COMM_WORLD);
   printf("Process 0 sending %d to process 1.\n", myvalue);

   MPI_Recv(&othervalue, 1, MPI_INT, 1, tag, MPI_COMM_WORLD, &status);
   printf("Process 0 received a value %d from process 1.\n", othervalue);
}

 if (rank == 1) {

   MPI_Recv(&othervalue, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, &status);
   printf("Process 1 received a value %d from process 0.\n", othervalue);

   myvalue = 25;
   MPI_Send(&myvalue, 1, MPI_INT, 0, tag, MPI_COMM_WORLD);
   printf("Process 1 sending %d to process 0.\n", myvalue);


}
```

- Program se bo izvedel do konca, tudi če za izmenjavo sporočil ni predvidenega medpomnilnika,
- kar pomeni da se komunikacija izvaja sinhrono.

- Izvajanje:
  - ko send iz procesa 0 pošlje, receive iz procesa 1 prejme,
  - oba procesa lahko nadaljujeta svoje delo:
  - Ko send iz procesa 1 pošlje, receive v procesu 0 prejme.
  - Procesa končata.

Primer: izmenjava sporočil (ki nikoli ne uspe)

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

if (rank == 0) {

   MPI_Recv(&othervalue, 1, MPI_INT, 1, tag, MPI_COMM_WORLD, &status);
   printf("Process 0 received a value %d from process 1.\n", othervalue);

   myvalue = 14;
   MPI_Send(&myvalue, 1, MPI_INT, 1, tag, MPI_COMM_WORLD);
   printf("Process 0 sending %d to process 1.\n", myvalue);

}

 if (rank == 1) {

   MPI_Recv(&othervalue, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, &status);
   printf("Process 1 received a value %d from process 0.\n", othervalue);

   myvalue = 25;
   MPI_Send(&myvalue, 1, MPI_INT, 0, tag, MPI_COMM_WORLD);
   printf("Process 1 sending %d to process 0.\n", myvalue);

}
```

- Program se ne bo nikoli izvedel do konca, ker oba procesa hočeta najprej sprejemati sporočila, zato zaustavita delovanje procesov, dokler sporočilo ni prejeto.

Primer: izmenjava sporočil (ki pogojno uspe)

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

if (rank == 0) {

 myvalue = 14;
   MPI_Send(&myvalue, 1, MPI_INT, 1, tag, MPI_COMM_WORLD);
   printf("Process 0 sending %d to process 1.\n", myvalue);

   MPI_Recv(&othervalue, 1, MPI_INT, 1, tag, MPI_COMM_WORLD, &status);
   printf("Process 0 received a value %d from process 1.\n", othervalue);
}

 if (rank == 1) {

   myvalue = 25;
   MPI_Send(&myvalue, 1, MPI_INT, 0, tag, MPI_COMM_WORLD);
   printf("Process 1 sending %d to process 0.\n", myvalue);

   MPI_Recv(&othervalue, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, &status);
   printf("Process 1 received a value %d from process 0.\n", othervalue);
}
```

- Program se bo izvedel do konca, če je možno vsaj eno sporočilo shraniti v medpomnilnik pri komunikaciji (odvisno od implementacije MPI).
- = če so send operacije asinhrone.
- Medpomnilnik v tem primeru ja lahko velik eno enoto tipa **`int`**.

Izmenjava sporočil brez zaustavitve delovanja procesov

- **`int MPI_Isend(void *buf, int count, MPI_datatype dt, int dest, int tag, MPI_Comm comm, MPI_Request *request)`**
  - Pošlje sporočilo z oznako **`tag`** procesu **`dest`**, ki je definiran znotraj skupine komunikatorja. Proces lahko takoj nadaljuje z delom, saj se v lokalnem pomnilniku rezervira prostor za shranjevanje sporočila, ki se začne prenašati preko komunikacijskega kanala.
    Informacije o stanju sporočila so shranjene v podatkovni strukturi **`&request.`**
- **`int MPI_Irecv(void *buf, int count, MPI_datatype dt, int source, int tag, MPI_Comm comm, MPI_Request *request)`**
  - Sprejme sporočilo z oznako **`tag`** od procesa **`source`**, ki je definiran znotraj skupine komunikatorja. Proces lahko takoj nadaljuje z delom, saj se v bistvu samo pripravi prostor v pomnilniku za (kasnejši) sprejem sporočila.
    Informacije o stanju sporočila so shranjene v podatkovni strukturi **`&request.`**

Nadzor izmenjave sporočil

- **`int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)`**
  - Testiramo, če je se operacija brez zaustavitve (Isend, Ireceive) dejansko že izvedla. Če se je izvedla (sporočilo je bilo v celoti oddano oziroma prejeto), je **`flag=true`**, sicer **`false`**. Ukaz se konča takoj.
- **`int MPI_Wait(MPI_Request *request, MPI_Status *status)`**
  - Ukaz čaka dokler se operacija povezana z **`request`** ne izvede v celoti.
- **`MPI_Iprobe(source, tag, comm, flag, status)`**
  - Proces, ki izvaja ta ukaz preveri, če mu je namenjeno sporočilo z oznako **`tag`** procesa **`source`**, v spr. flag vrne **`true`**, če ga čaka sporočilo , sicer **`false`**.
- **`MPI_Probe(source, tag, comm, status)`**
  - Proces, ki izvaja ta ukaz preveri, če mu je namenjeno sporočilo z oznako **`tag`** procesa **`source`**. Ukaz se vrne šele, ko mu proces, ki ga preverja, nameni sporočilo, sicer se proces, ki izvaja ta ukaz, zaustavi.

Primer: pošiljanje in sprejemanje sporočil brez zaustavitve izvajanja procesov.

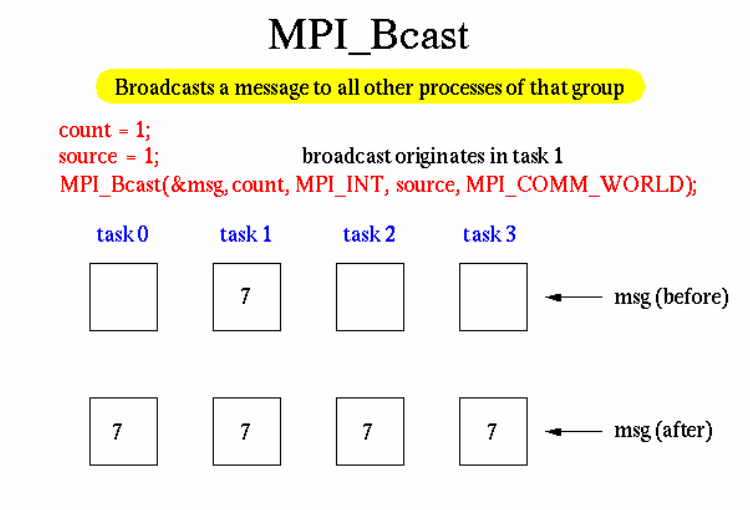
- Procesa se na zaustavita na operacijah send in receive.
- Vrednost pri procesu 1 dobimo šele:
  - ko jo je proces 0 v celoti poslal (MPI_Wait),
  - ko jo je proces 1 v celoti prejel (MPI_Wait).

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

if (rank == 0) {

   myvalue = 14;
   MPI_Isend(&myvalue, 1, MPI_INT, 1, tag, MPI_COMM_WORLD, &request);

   /***** do some work ****/
   sleep(rand()%10);

   MPI_Wait(&request, &status);
   printf("Process 0 send value %d to process 1.\n", myvalue);

}

 if (rank == 1) {

   MPI_Irecv(&myvalue, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, &request);

   /***** do some work ****/
   sleep(rand()%10);

   MPI_Wait(&request, &status);
   printf("Process 1 received value %d from process 0.\n", myvalue);
}
```

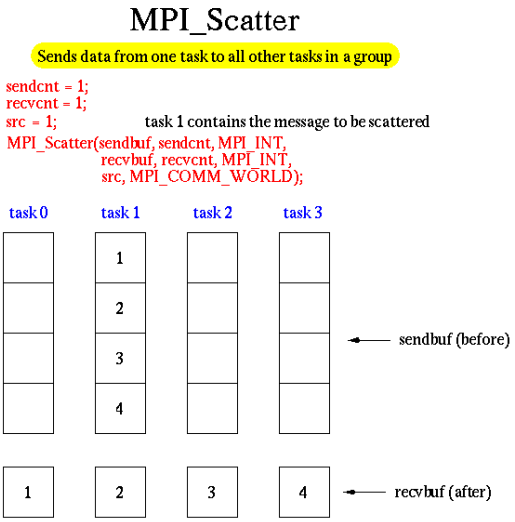
Operacije za skupinsko komunikacijo

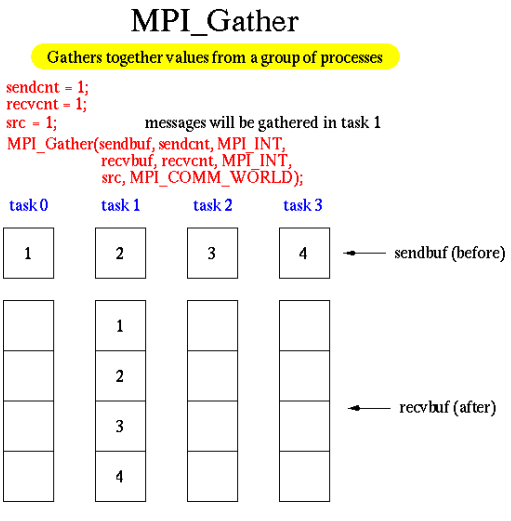
- Operacije za skupinsko komunikacijo omogočajo interakcijo enega procesa z vsemi ostalimi procesi v isti skupini:
  - **`MPI_Barrier()`**
    - Javi prihod v točko pregrade. Funkcija zaustavi delovanje procesa, dokler vsi procesi v skupini ne pridejo do te točke.
  - **`MPI_Bcast()`**
    - Pošlji kopije sporočila vsem procesom znotraj skupine.
  - **`MPI_Scatter()`**
    - Pošlji sporočila iz tabele a velikosti size drugim procesom: sporočilo a[i] pošlji procesu i.
  - **`MPI_Gather()`**
    - Zberi sporočila več procesov skupaj in jih shrani v tabelo a velikosti size: sporočilo procesa i shrani v a[i].
  - **`MPI_Reduce()`**
    - Zberi sporočila iz vseh procesov v isti skupini skupaj in jih združi v eno sporočilo glede na operacijo, ki je predpisana v ukazu. Možne operacije: **`MPI_SUM`**, **`MPI_MAX`**, **`MPI_LAND`** (logični AND), **`MPI_BOR`** (binarni OR), …
  - **`MPI_Allreduce()`**
    - Enako kot **`MPI_Reduce`**, samo da vsak proces v skupini dobi to združeno sporočilo.

Broadcast

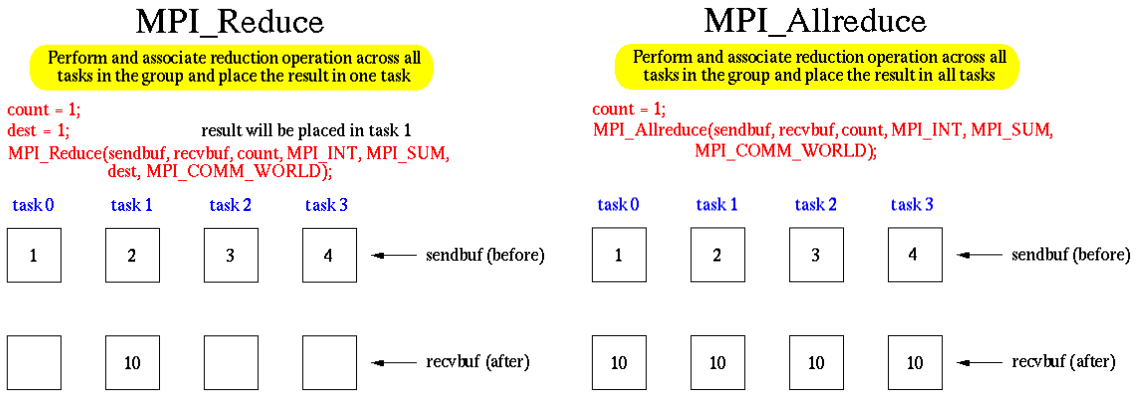


Scatter/Gather





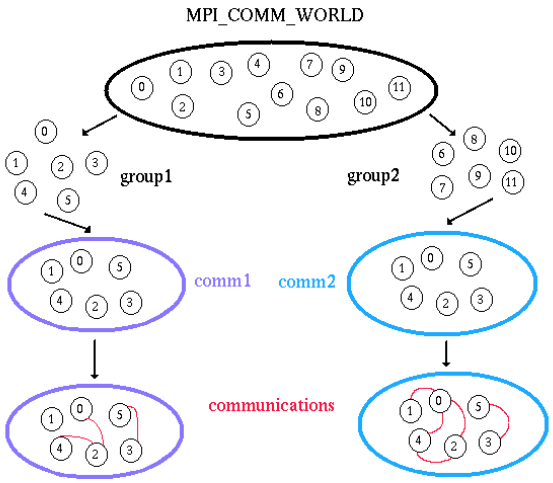
Reduce/Allreduce



Primeri vseh funkcij.

- Primeri uporabe funkcij.
  - `MPI_Bcast`
  - `MPI_Gather`
  - `MPI_Scatter`
  - `MPI_Reduce`

Skupine in komunikatorji



- V komunikatorjih so združene skupine procesov, ki lahko komunicirajo med seboj.
  - Komunikatorje lahko obravnavamo tudi kot dodatno identifikacijo sporočilom (ne samo tag, ampak tudi v kateri skupini (komunikatorju) pripada to sporočilo).
- V komunikatorju MPI_COMM_WORLD so združeni vsi procesi MPI aplikacije.
- V MPI so implementirane funkcije
  - za nastanek novih komunikatorjev, za brisanje, združevanje komunikatorjev, …
  - za delo z rangi procesov znotraj komunikatorja in delo s skupinami procesov, …

Večji primer: Izračun decimalk števila PI

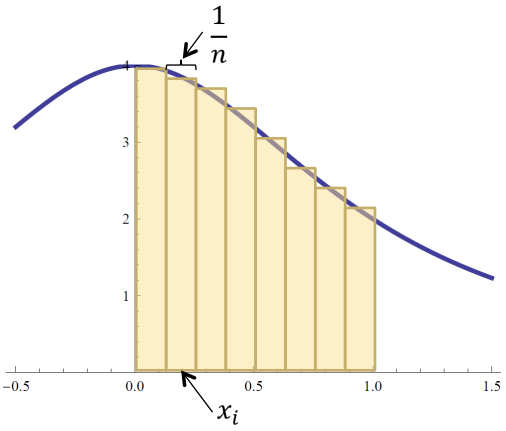
- Vrednost določenega integrala:

$$\int_0^1 \frac{4}{1+x^2}\,dx = \pi$$

- Aproksimacija določenega integrala:

$$\frac{1}{n}\sum_{i=1}^{n}\frac{4}{1+x_i^2} \qquad x_i = \frac{1}{n}(i-0.5)$$

- Torej izračunamo vsoto in dobimo približek za Pi. n mora biti čim večji, da bo približek boljši.



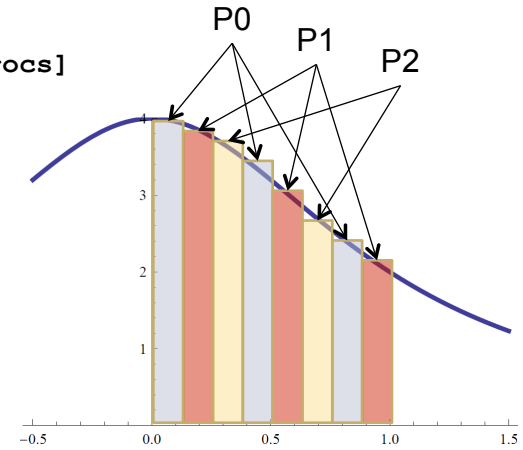
Večji primer: Izračun decimalk števila PI

- Razdelimo računanje vsot na posamezne procese:
- Vsak proces:

```
for[i=rank+1 to n step by numprocs]
```

$$\frac{1}{n}\sum_{i=1}^{n}\frac{4}{1+x_i^2} \qquad x_i = \frac{1}{n}(i-0.5)$$

- Nato uporabimo funkcijo MPI_Reduce, da pošljemo procesu 0 vse delne vsote, ki jih še sproti seštejemo.
- Končni rezultat je število Pi.



Izračun števila PI

Vsak proces dobi število intervalčkov, da lahko računa svoje vsote.

Vsota je sestavljena iz vrednosti x_i izračunane na vsakem intervalčku razmaknjenih za numprocs.

Vse delne vsote so posredovane do procesa 1 in vmes že seštete. To je končni Pi.

```
int main(int argc,char *argv[]) {
    int done = 0, n, myid, numprocs, i;
    double PI25DT = 3.14159265358979323846264643;
    double mypi, pi, h, sum, x;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);

    while (!done) {
        if (myid == 0) {
            printf("Enter the number of intervals: (0 quits) ");
            (void)scanf("%d",&n);
        }

        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

        if (n == 0) break;

        h = 1.0 / (double) n;
        sum = 0.0;
        for (i = myid + 1; i <= n; i += numprocs) {
            x = h * ((double)i - 0.5);
            sum += 4.0 / (1.0 + x*x);
        }

        mypi = h * sum;
        MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,MPI_COMM_WORLD);

        if (myid == 0)
            printf("pi is approximately %.16f, Error is %.8e\n",
                   pi, fabs(pi - PI25DT));
    }

    MPI_Finalize();

    return 0;
}
```

Velik primer: Izvedba računske farme

- Gre za princip vreče z opravili (*ang. bag of tasks*)
  - en proces (koordinator, *ang. farmer*) deli opravila med delavci,
  - procesi delavci (*ang. worker*) opravljajo opravila, ko ga opravijo, javijo, da so prosti, da lahko dobijo novo opravilo,
  - število opravil in naloge opravil so znane pred začetkom programa,
  - dodeljevanje procesov je dinamično – glede na hitrost in uspešnost izvajanja procesov posameznega delavca.

- Predpostavimo:
  - število opravil je večje ali enako kot je število procesov,
  - opravila v našem primeru so zgolj seštevanje celoštevilskih vrednosti, rezultati so celoštevilske vrednosti (običajno so opravila bolj zahtevna in rezultati bolj kompleksne podatkovne strukture).

Izvedba računske farme

Proces št. 0 je koordinator,
vsi ostali procesi so delavci.

Koordinator izve, koliko delavcev ima na voljo za deljenje opravil. Vsak delavec dobi svoj ID, ki je kar rang procesa delavca.

```
#define MAX_TASKS 100
#define NO_MORE_TASKS MAX_TASKS+1

void farmer (int workers);
void worker (int rank);

int main(int argc, char *argv[])
{

  int np, rank;
  time_t t;

  t = time(NULL);  // seed the random number
  srand((int) t);  // generator from outside

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &np);
  if (rank == 0) {
     farmer(np-1);
  } else {
     worker(rank);
  }
  MPI_Finalize();
}
```

Koordinator

Za opravila izbiramo naključna cela števila med 0 in 4, ki pomenijo, koliko enot dela opravijo delavci.

Prvih **n** opravil pošljemo delavcem: opravilo i pošljemo delavcu **i** (procesu i+1), oznaka opravila je **i**.

Vsako nadaljnje opravilo delimo naprej med delavce:
ko delavec **who** konča z opravilom, mu dodelimo novo opravilo.

Prišli smo do konca: Opravil ni več. Poberemo še zadnje rezultate in pošljemo še eno sporočilo z oznako: **NO_MORE_TASKS**, kjer sporočamo delavcem, naj prenehajo delati.

```
void farmer (int workers)
{
   int i, task[MAX_TASKS], result[MAX_TASKS], temp, tag, who;
   MPI_Status status;

   for (i=0; i<MAX_TASKS; i++) {
     task[i] = rand()%5;   // set up some "tasks"
   }

   // Assume at least as many tasks as workers
   for (i=0; i<workers; i++) {
     MPI_Send(&task[i], 1, MPI_INT, i+1, i, MPI_COMM_WORLD);
   }

   while (i<MAX_TASKS) {
     MPI_Recv(&temp, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
                                 MPI_COMM_WORLD, &status);
     who = status.MPI_SOURCE;
     tag = status.MPI_TAG;
     result[tag] = temp;
     MPI_Send(&task[i], 1, MPI_INT, who, i, MPI_COMM_WORLD);
     i++;
   }

   for (i=0; i<workers; i++) {
     MPI_Recv(&temp, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
                                 MPI_COMM_WORLD, &status);
     who = status.MPI_SOURCE;
     tag = status.MPI_TAG;
     result[tag] = temp;
     MPI_Send(&task[i], 1, MPI_INT, who, NO_MORE_TASKS,
                                         MPI_COMM_WORLD);
   }
}
```

Delavec

```
void worker (int rank)
{
   int tasksdone = 0;
   int workdone  = 0;
   int task, result, tag;
   MPI_Status status;

   MPI_Recv(&task, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD,
                                               &status);
   tag = status.MPI_TAG;
   while (tag != NO_MORE_TASKS) {
     sleep(task);
     result = rank;
     workdone+=task;
     tasksdone++;
     MPI_Send(&result, 1, MPI_INT, 0, tag, MPI_COMM_WORLD);
     MPI_Recv(&task, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD,
                                                 &status);
     tag = status.MPI_TAG;
   }
   printf("Worker %d solved %d tasks totalling %d units of work \n", rank, tasksdone, workdone);
}
```

Delavec prejme nalogo od koordinatorja
procesa 0.
Preverja oznako sporočila, če je NO_MORE_TASKS, ne dela nič, samo izpiše rezultate dela.

Delavec prejme nalogo od koordinatorja - procesa 0.
Opravilo simuliramo tako, da spimo toliko časa, kot je enot opravila.
Preštejemo, koliko opravil smo reševali in koliko dela smo naredili.
Rezultat opravila pošljemo koordinatorju (samo eno celo število).
In zopet prejme opravilo od koordinatorja.
Delavec komunicira samo s koordinatorjem.
Preverja oznako sporočila, če je NO_MORE_TASKS, skoči ven iz zanke.