Дистрибуирано програмирање

Зоран Јовановић

Дистрибуирано програмирање

- Конкурентно програмирање:
  - Процеси деле заједничку меморију
  - Процеси комуницирају користећи дељене променљиве
- Дистрибуирано програмирање:
  - Меморија је дистрибуирана
    (Процеси не комуницирају (директно) коришћењем дељених променљивих)
  - Процеси комуницирају разменом порука
    (канали, сандучићи, RPC, рандеву, ...)

Размена порука – *message passing*

- Дефинисање специјалних мрежних операција које укључују синхронизацију – *message passing primitives*
- Процеси деле канале – комуникационе путеве између процеса
- Типично канали су једини објекат који процеси деле

Размена порука – *message passing*

- Размена порука омогућава комуникацију између:
  - пошиљаоса (*sender*) и
  - примаоца (*receiver*)
- коришћењем комуникационог канала
  - чува се редослед порука (Queue је FIFO)
  - омогућава се атомски приступ,
  - користе се типови података,
  - води се рачуна о грешкама,
  - ...

Размена порука – *message passing*

- Варијанте:
  - Асинхроно или синхроно
  - Статичко или динамичко именовање
  - Проток информација у једном или два смера
  - Блокирајући или неблокирајући
  - Веза: 1-1, N-1, 1-N, N-N
  - Канали или сандучићи
  - ...

Асинхрони *message passing*

Декларација канала
- chan ch(type1 id1, …, typen idn);
- типови су неопходни, id су опциони

Комуникационе примитиве *send* и *receive*

- send ch(expr1, …, exprn);
  - типови усклађени са онима за канал
  - *send* је неблокирајући
  - неограничен бафер *queue*
- receive var(var1, …, varn);
  - типови усклађени са онима за канал
  - *receive* је блокирајући
  - касни процес све док се не појави бар једна порука
    - Понекад постоји метода empty(c) која враћа *true* ако је канал празан

Асинхрони *message passing*

- Поређење семафора и асинхроне размене порука

Семафори	**Размена порука**
Семафор (Semaphore)	Канал (Chan)
signal	send
wait	receive
Интерна семафорска вредност	Број порука у баферу

- Игнорише се тело поруке

Филтри

- Филтер је процес који има:
  - један или више улазних канала и
  - један или више излазних канала
- Излаз – функција улаза и иницијалног стања
- Предикат – повезује излаз са улазом, сваки пут када филтер шаље на излаз

Пример: токови (*stream*) и проток података (*dataflow*)

Филтерски процес - *mailbox*

```
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-1) {
            i = i + 1;
            receive input(line[i]);
        }
        line[i] = EOL;
        send output(line);
        i = 0;
    }
}
```

Филтри – мрежа за сортирање

- Пример сортирање – улазни канал прима n вредности које треба да буду сортиране, излаз даје сортиране вредности

- Предикат: $\forall$ i: 1 <= i < n: sent[i] <= sent[i+1] $\wedge$ пермутација улазних вредности

- Терминација :
  - n познато унапред,
  - n прва вредност и
  - специјална вредност (*sentinel value*)

Филтри – мрежа за сортирање

- Процеси у ОС – један процес у 3 корака:
  - прими све улазе,
  - сортирај,
  - пошаљи сортиране бројеве

- Мреже за сортирање – мрежа малих процеса који се извршавају у паралели
- Мреже за стапање – од две или више сортираних листа се формира једна сортирана листа

Филтерска мрежа за сортирање са бинарнин стаблом

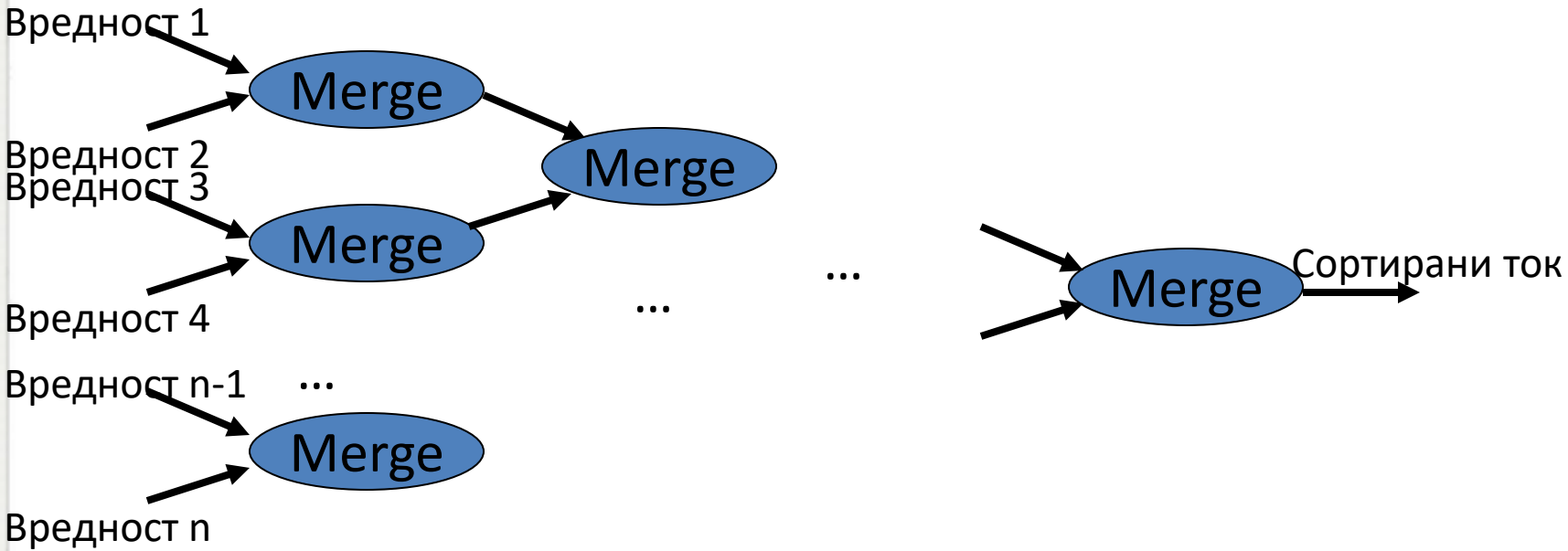
- Предикат:

  in1 and in2 empty $\wedge$ sent[n+1] == EOS $\wedge$

  ($\forall$ i: 1 <= i < n: sent[i] <= sent[i+1]) $\wedge$ пермутација улазних вредности in1 and in2 – када терминира
- Варијанте:
  - прима све и памти или
  - ствара ток креиран сталним поређењем улазних вредности
- Низови који креирају излазе који теку са потенцијалним паралелизмом
- Мрежа процеса и канала
- Погодно када је n степен броја 2

Interconnecting merge processes

- Бинарно стабло са n-1 процеса



Merge (1)

```
chan in1(int), in2(int), out(int);
process Merge {
    int v1,v2;
    receive in1(v1); receive in2(v2)
    while (v1 != EOS and v2 != EOS) {
        if (v1 <= v2)
                { send out (v1); receive in1(v1); }
        else
                { send out(v2); receive in2(v2); }
    }
```

Merge (2)

```
    if (v1 == EOS)
        while (v2 != EOS)
            { send out(v2); receive in2(v2); }
    else
        while (v1 != EOS)
            { send out(v1); receive in1(v1); }
    send out(EOS);
}
```

Статичко и динамичко именовање

- Статичко
  - сви канали су глобалан низ и појаве *Merge* (n-1) се пресликавају у 2n-1 канала
  - идентификатор процеса $Merge_i$ се пресликава у идентификаторе канала
  - уградња стабла у низ
- Динамичко
  - опет сви канали су глобални
  - главни процес динамички креира канале и прослеђује их *Merge* процесима
  - придружује сваком процесу три канала када су креирани
- Флексибилност са динамичким именовањем и креирањем

Клијенти и сервери

- Сервер је процес који непрекидно обрађује захтеве пристигле од клијентских процеса.
- Сервер у петљи опслужује позиве преко *request* и *reply* канала
- Статичко именовање – фиксан број канала и clientID
- Динамичко именовање – сваки клијент креира клијента са приватним *reply* каналом

Сервер са једном операцијом

```
chan request(int clientID, types of input values);
chan reply[n] (types of results);

process Server {
    int clientID; declaration of permanent variables; initialization code;
    while (true) {
        receive request(clientID, input variables);
        results = f(input variables, permanent variables);
        send reply[clientID] (results);
    }
}
```

Сервер са једном операцијом

```
process Client[i = 0 to n-1] {
    send request(i, value arguments);
    receive reply[i] (result arguments);
}
```

Активни монитори са једном операцијом!

Клијенти и сервери – Активни монитори

- Симулација монитора коришћењем серверских процеса и технике *message passing*
- Особине:
  - Један ток контроле,
  - Један улазни канал
- Поређење 1:
  - Сталне променљиве – Локалне променљиве сервера
  - Позиви процедура – send request(); receive reply()
  - Monitor entry – receive request()
  - Procedure return – send reply()

Активни монитори са више операција

- Комплексније је јер има више операција
- Различити су аргументи, операције и резултати
- Аргументи и резултати се разликују по типовима и броју параметара

Активни монитор – више операција

type op_kind = enum(op_1, …, op_n);

type arg_type = union(arg_1, …, arg_n);

type result_type = union(res_1, …, res_n);

chan request(int clientID, op_kind, arg_type);

chan reply[n] (result_type);

Активни монитор – више операција (2)

```
process Server { int clientID; op_kind kind; arg_type args;
    result_type results; permanent variables; initialization code;
    while (true) {
        receive request(clientID, kind, args);
        if (kind == op1)
                {body of op1; }
        …
        else if (kind == opn)
                 {body of opn; }
        send reply[clientID] (results);
    }
}
```

Активни монитор – више операција (2’)

```
process Server { int clientID; op_kind kind; arg_type args;
    result_type results; permanent variables; initialization code;
    while (true) {
        receive request(clientID, kind, args);
        switch(kind):
            case op_1: {body of op_1; }
            …
            case op_n: {body of op_n; }
        send reply[clientID] (results);
    }
}
```

Активни монитор – више операција (3)

```
process Client[i = 0 to n-1] {
    arg_type myargs; result_type myresults;
    set value arguments in myargs;
    send request(i, opj, myargs);
    receive reply[i] (myresults);
}
```

Активни монитор – више операција

- Поређење 2:
  - Идентификатори процедура – Канал за захтев и типови орерације
  - Тела процедура – Делови *case* исказа зависно од типа операције

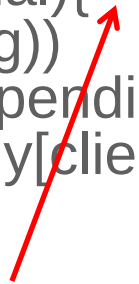
Монитор – условне променљиве

```
monitor FIFOsemaphore {
    int s = 0; /* ili druga inicijalna vrednost */
    cond pos; /* signal kada bi trebalo s > 0 */
    procedure semWait() {
        if (s == 0)         wait(pos);
        else                s := s-1;
    }

    procedure semSignal() {
        if (empty(pos))s:= s + 1;
        else                signal(pos);
    }
}
```

Активни монитор – условне променљиве

```
process SemaphoreServer {
    int clientID; op_kind kind; result_type results; int s = 0;
    queue pending; // inicijalno prazno
    while (true) {
        receive request(clientID, kind);
        if (kind == semWait){
            if (s == 0) add(pending, clientID);
            else     s := s-1;}
        else if (kind == semSignal){
            if (empty(pending))     s:= s + 1;
            else{    remove (pending, clientID);
                send reply[clientID](results); //token
}}}}
```



Сигнал је неблокирајући, иначе би захтевало send reply[clientID](results);

Активни монитор – условне променљиве

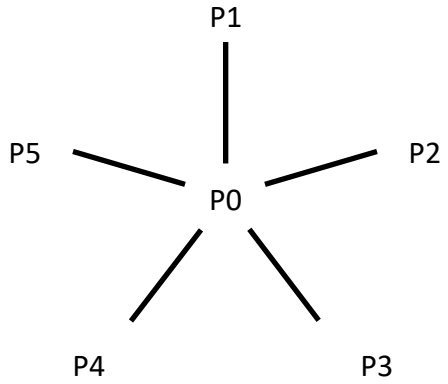
- Поређење 3:
  - Wait исказ – Чување захтева који чекају
  - Signal исказ – Дохватање и обрада захтева који је чекао
  - Може захтевати:
    - чување и рестаурирање локалних променљивих метода (контекста) и
    - адресе за наставак (лабела након *Wait* исказа)

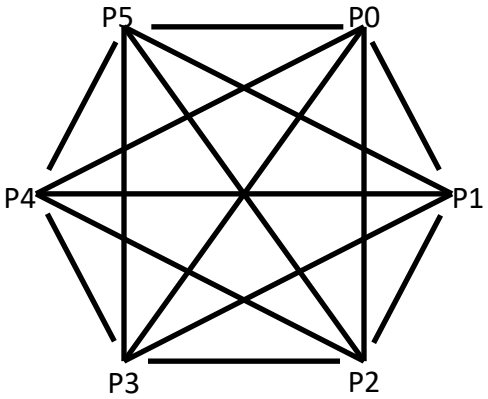
Поређење монитор/активни монитори

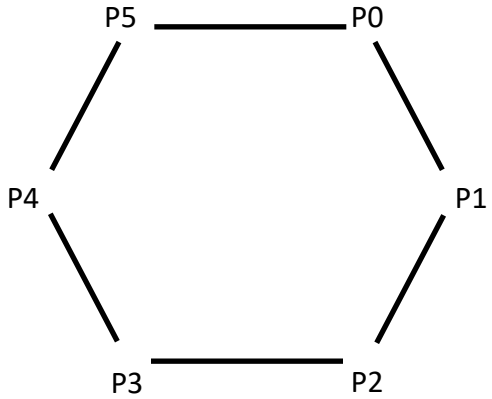
Monitor based	**Message based**
Сталне променљиве	Локалне серверске променљиве
Идентификатори процедура	Канал за захтев и типови op
Позиви процедура	send request(); receive reply()
Улазак у монитор	receive request()
Излазак из монитора	send reply()
Wait исказ	Чување захтева који чекају
Signal исказ	Дохватање и обрада захтева који је чекао
Тела процедура	Делови *case* исказа зависно од типа операције

Интерагујући процеси – *Interacting Peers*

- Комуникациони образац (*pattern*)
  - централизовани,
  - симетрични и
  - прстен







Размена вредности – *Interacting Peers*

- Пример размене вредности:
- Посматра се n процеса (P[0], . . . , P[n-1], n > 1)
- сваки процес има локалну вредност v
- циљ да сваки процес сазна највећу и најмању вредност.

Решење 1 – Централизовано

- Слање вредности једном процесу који налази max и min и шаље одговор до сваког
  - процес P[0] процес координатор
  - процес P[0] налази max и min
  - остали процеси шаљи поруку
  - Процеси извршавају различит код

- Број (послатих) порука:
  - P[1], . . . , P[n-1] сваки по 1
  - P[0] шаље n-1 порука, одговор садржи max и min
  - Укупно: (n -1) + (n-1) = 2(n-1)

- Број канала: n

Решење 1 – Централизовано

```
chan values(int), results[n] (int smallest, int largest);
process P[0] { # coordinator
    int v; # inicijalizovana vrednost
    int new, smallest = v, largest = v;
    for [i = 1 to n-1] {
        receive values(new);
        if (new < smallest) smallest = new;
        if (new > largest) largest = new;
    }
    for [i = 1 to n-1]
        send results[i] (smallest, largest)
}
```

Решење 1 – Централизовано (1)

```
process P[i = 1 to n-1] {
    int v; # inicijalizovana vrednost
    int smallest, largest;
    send values(v);
    receive results[i] (smallest, largest);
}
```

Решење 2 – Симетрично

- Сваки процес шаље локалну вредност свим другим процесима и онда прима од свих других процеса и локално израчуна
  - Процеси извршавају идентичан код
- Број (послатих) порука:
  - P[0], P[1], . . . , P[n-1] сваки по n-1
  - Укупно: n (n-1)
- Број канала: n

Решење 2 – Симетрично

```
chan values[n] (int);
process P[i = 0 to n-1] { # symmetric no coordinator
    int v; # inicijalizovana vrednost
    int new, smallest = v, largest = v;
    for [j = 0 to n-1 st j != i]
        send values[j] (v);
    for [j = 0 to n-1st j != i ] {
        receive values[i] (new);
        if (new < smallest) smallest = new;
        if (new > largest) largest = new;
    }
}
```

Решење 3 – Прстен

- Сваки процес шаље *min* и *max* примљених и *min* и *max* од локалних вредности
  - процеси извршавају скоро идентичан код
  - процес P[0] прво шаље па прима поруку
  - остали процеси прво примају па шаљу поруку

- Број (послатих) порука:
  - P[0], P[1], . . . , P[n-2] сваки по 2
  - P[n-1] 1
  - Укупно: 2 n-1 (може и 2 (n-1) ако се и претпоследњи разликује)

- Број канала: n

Решење 3 – Прстен

```
chan values[n] (int smallest, int largest);
process P[0] { # initiator
    int v; # inicijalizovana vrednost
    int smallest = v, largest = v;
    send values[1] (smallest, largest);
    receive values[0] (smallest, largest);
    send values[1] (smallest, largest);
}
```

Решење 3 – Прстен (1)

```
Process P[i = 1 to n-1] {
    int v; # initialized
    int smallest, largest;
    receive values[i] (smallest, largest);
    if (v < smallest) smallest = v;
    if (v > largest) largest = v;
    send values[(i+1) % n] (smallest, largest);
    receive values[i] (smallest, largest);
    if (i < n-1) send values [i+1] (smallest, largest);
}
```

Синхрони *message passing*

Операција слање synch_send

- synch_send ch(expr1, …, exprn);
  - типови усклађени са онима за канал
  - блокирајућа операција
  - постоји граница у величини бафера
  - директно копирање из адресног простора пошиљаоца у адресни простор примаоца је могуће
  - адресе порука које чекају да буду послате
- Операција пријема *receive* је иста као и пре

Синхрони *message passing*

- Предности:
  - могуће одредити максималну величину канала,
  - пошиљалац је синхронизован са примаоцем,
    - прималац има највише једну поруку по каналу необрађену,
    - пошиљалац има највише једну непослату поруку.
- Мане:
  - смањена конкурентност,
  - мртво блокирање (*deadlock*) се лако направи.

Сервер са једном операцијом

```
chan request(int clientID, types of input values);
chan reply[n] (types of results);

process Server {
    int clientID; declaration of permanent variables; initialization code;
    while (true) {
        receive request(clientID, input variables);
        results = f(input variables, permanent variables);
        synch_send reply[clientID] (results);
    }
}
```

Сервер са једном операцијом

```
process Client[i = 0 to n-1] {
    synch_send request(i, value arguments);
    receive reply[i] (result arguments);
}
```

Пример за мртво блокирање (*deadlock*)

```
chan 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);
}
```

Communicating sequential processes (CSP)

- Програмски модел код кога се програми састоје од више секвенцијалних процеса који међусобно комуницирају прослеђивањем порука.
- Директно именовање процеса.
- Излазни и улазни искази за размену порука.
- Синхрона *point-to-point* комуникација.
  - процес који шаље поруку се блокира док она не стигне процесу примаоцу,
  - процес који прима поруку се блокира док порука не стигне.
- Алтернативна команда за спречавање мртвог блокирања.

Процеси и паралелизам

- Декларација процеса:

  imeProcesa :: implementacija
- Иза назива процеса могу се налазити и заграде у оквиру којих је наведена листа целобројних вредности (нпр. *proc(1)*, *proc(1,2)* итд.). У том случају и назив и листа представљају јединствени идентификатор, тј. име процеса

- Декларација паралелних процеса:

  [proc1::impl1 || ... || procN::implN]
- Скраћени начин за декларацију: [X(i:1..n)::proc]

  [X(1)::proc(1) || ... || X(n)::proc(n)]

Комуникација

Комуникационе примитиве:

- <process>!<message>
  - слање поруке <*message*> процесу <*process*>
- <process>?<message>
  - пријем поруке <*message*> од процеса <*process*>
- До преноса поруке долази само ако су:
  - називи пошиљаоца и примаоца упарени, и
  - поруке у наредби за слање и пријем компатибилне (*pattern matching*)
- <*message*> прост или сложен израз

Заштићена (*Guarded*) комуникација

- Како спречити чекање на портовима док постоје други процеси који могу да комуницирају на другим портовима
- B; C -> S;
  - B Boolean услов (опциони),
  - C комуникациони исказ заштићен са B,
  - S листа исказа
- Заштита успева ако је B *true* и извршавање C не изазива *delay* – неко чека
- Заштита не успева ако је B *false*.
- Заштита блокира ако је B *true* и C не може да буде извршено – нико не чека

Контролне структуре

- Алтернативна команда

```
[   B1; C1 -> S1;
    [] …
    [] Bn; Cn -> Sn;
]
```

Шта ако је више услова испуњено?

- Варијанте су и само Bi, или само Ci

- Ако неки процес заврши са радом, сви услови у којима се налази наредба пријема од датог процеса не могу бити задовољени.

Контролне структуре

- Репетитивна алтернативна команда (итеративна команда)

```
*[ B1; C1 -> S1;
   [] …
   [] Bn; Cn -> Sn;
 ]
```

- Варијанте су и само Bi, или само Ci

Readers/Writers problem

```
MONITOR :: [ numR, numW : integer; numR := 0; numW := 0;
    *[
            numW = 0; (i : 0..nr) Reader(i)?startread() ->
                                    numR := numR + 1
            [] (i : 0..nr) Reader(i)?endread() ->
                                    numR := numR - 1
            [] numW = 0; numR = 0;(i : 0..nw) Writer(i)?startwrite() ->
                                    numW := numW + 1
            [] (i : 0..nw) Writer(i)?endwrite() ->
                                    numW := numW - 1
    ]
]
```

Linda

- Генерализовање дељених променљивих и асинхроне размене порука
- Није програмски језик – 6 примитива за приступ простору торки - *tuple space*
- *Tuple space* – дељена асоцијативна меморија етикетираних записа - *tagged data records*
- Јединствен дељени комуникациони канал ка простору торки
- Памти дистрибуиране структуре података којима процеси конкурентно приступају – чак може трајно да памти

Примитиве у библиотеци *Linda*

- OUT – као *send*, неблокирајући
- IN – као *receive* који скупља поруку са канала, блокирајући
- RD – као *receive* који чита поруку са канала и оставља је (у ствари оставља у простору торки), блокирајући
- EVAL – креирање процеса
- INP – *nonblocking* IN који даје предикат успешности
- RDP – *nonblocking* RD који даје предикат успешности

Простор торки (*Tuple space*)

- Торка (*Data tuple*) – означени запис (*tagged record*)
- Пример торке: ("tag", value1, … , valuen)
- Памћење торке: OUT("tag", expr1, …, exprn);
- Евалуација израза и памћење у простору торки
- *Process tuples* – процеси који се извршавају асинхроно

Упаривање шаблона – *IN*

- IN ("tag", a:t, 55,…, true, k:t); извлачи торку смештену (запамћену) у простор торки, ако постоји усклађена торка
- IN параметри се зову шаблони (*template*) – морају се упарити: тагови су идентични, број поља је исти и одговарајућа поља имају исти тип – потребан услов
- "tag", 55, true, су стварни параметри и представљају део “назива” торке. a:t, k:t су формални параметри и означавају локације у које ће се сместити одговарајуће вредности поља из прочитане торке. ?a и ?k су алтернативни начини обележавања локација. Торка је усклађена, ако су једнаки сви стварни параметри.

Синхронизација на баријери

- Иницијализација вредности у простору торки (само један процес иницијализује)
- OUT("barrier", 0);
- Сваки процес ради следеће
- IN("barrier", ?counter);
- OUT("barrier", counter + 1);
- Свих n процеса чекају све док се не достигне вредност n
- RD("barrier",n); где n има конкретну вредност !!!

Процесне торке

- EVAL("tag", expr1, …, exprn);
- Један или више израза су позиви процедура или функција
- Теоретски паралелно извршавање за сва поља – у реалности само за поља која су функције
- Постаје пасивна торка када сва поља добијају вредност израза у EVAL

Cobegin и слање/пријем вредности

```
Co [i = 1 to n]
    a[i] = f(i);
```

Еквивалент је:

```
for (i = 1; i <= n; i++)
    EVAL("a", i, f(i));
```

Слање на и пријем са канала

```
OUT("chName", expressions);
IN(“chName", variables);
```

Go

- Подршка за рад са каналима.
- Подршка за асинхрону и синхрону комуникацију
- Подршка за лагане нити (*go-routine*)
- Алтернативна команда
- Синтакса је мешавина:
  - императивног програмирања (програмски језик *C*) и
  - функционалног програмирања (*lambda calculus*)
  - велики утицај CSP језика

Go – Канали

- Креирање канала:
  - c := make (chan int) – синхрони
  - c := make (chan int, 10) – асинхрони, величине 10
  - Нема потпуни асинхроних, неограничене величине
- Слање порука <-
  - chan1 <- x
  - у канал chan1 шаље x
- Пријем поруке <-
  - x = <- chan1
  - из канала chan1 прима податак у променљиву x,
  - <- chan1, пријем поруке без смештања
- Затварање канала *close*
  - close(c)

Go – Функције

Позиви:

- f (x) – обичан (синхрони) позив функције, f је дефинисана функција или дефиниција функције
- go f (x) – функција се позива као засебна (асинхрона) лака нит *go-routine*
  go-routine се завршава раније ако се заврши родитељски процес
- defer f (x) – извршавање позива се одлаже до краја процеса
  ако их има више извршавају се LIFO принципу

Go-routine

```
package main
import ( "fmt"
        "time"
)
func main() {
        foo := make(chan int, 10)
        go func() {
                time.Sleep(1000)
                foo <- 1 // send
        }()
        go func() {
                time.Sleep(1)
                foo <- 2
        }()
        fmt.Println("first = ", <-foo)
        fmt.Println("second = ", <-foo)
}
```

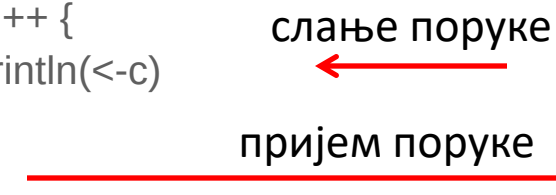
Go – Select

- Наредба *select* омогућава да нит (*goroutine*) чека на више комуникационих канала (алтернативна команда) било за слање било за пријем
- Наредба *select* се блокира докле бар један исказ не постане испуњен (извршан)
- Уколико је више испуњених бира се један на насумичан начин

```
select {
        case receive1:
                ...
        case send1:
                ...
}
```

Go – Select

```
package main
import "fmt“
func main() {
        c := make(chan int)
        quit := make(chan int)
        go func() {
                for i := 0; i < 10; i++ {
                        fmt.Println(<-c)
                }
                quit <- 0
        }()
        fibonacci(c, quit)
}
```




```
func fibonacci(c, quit chan int) {
        x, y := 0, 1
        for {
                select {
                case c <- x:
                        x, y = y, x+y
                case <- quit:
                        fmt.Println("quit")
                        return
                }
        }
}
```

Go – Default

- Код наредбе селекције може се искористити и подразумевана (*default*) грана у случају да ни једна друга грана није испуњена
- Може се користити у случају да се је потребно слање и пријем без блокирања:

```
select {
        case i := <-c: // use i
        default : // receiving from c would block
}
```

Go – Default

```
func main() {
	tick := time.Tick(100 * time.Millisecond)
	boom := time.After(500 * time.Millisecond)
	for {
		select {
		case <-tick:
			fmt.Println("tick.")
		case <-boom:
			fmt.Println("BOOM!")
			return
		default:
			fmt.Println("    .")
			time.Sleep(50 * time.Millisecond)
		}
	}
}
```

Генерисање периодичних догађаја

Go – Баријера

- У пакету *sync* постоји предефинисани тип *WaitGroup* који дефинише баријеру са операцијама:
  - Add(m) – поставља баријеру са m процеса
  - Done() – сигнализира да је позивајући процес завршио.
  - Wait() – чека док се сви процеси не заврше.

Remote Procedure Calls (*RPC*) и *Rendezvouz*

- Размена порука се може користити за развој клијент сервер апликација и при томе
  - сваки клијент захтева засебан *reply* канал
  - клијент/сервер код унидирекционог *message passing* је рађен са две експлицитне размене порука
- Једноставније модел за развој клијент/сервер апликација нуде удаљени позиви метода (*RPC*) и рандевуи (*rendezvous*)
- Уводи се двосмерни (бидирекциони) комуникациони канал
- *Client* – *send* праћен са *receive*
- *Server* – *receive* праћен са *send*

Remote Procedure Calls – идеја

Позивалац

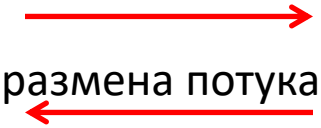
Рачунар А

Позвани

Рачунар Б

op opname(FORMALS); # declaration

...
call opname(ARGS);
…



proc opname(FORMALS) # new process
...
end;

Remote Procedure Calls

- Понаша се као обичан позив процедуре, притом позивалац и позвани могу бити на различитим машинама.
- Позивалац се блокира док се позвана процедура на заврши (комбинација мониторских метода и синхроне размене порука).
- Концептуално сваки позив операције (процедуре) креира нови процес.
- Омогућава двосмерну комуникацију: позивалац шаље аргументе, а позвани враћа резултат.
- Синхронизација се мора експлицитно програмирати.

Remote Procedure Calls – декларација

```
Module mname
    op opname1(formals) [return result];
    … # headers of exported operations
Body
Variable declarations;
Initialization code;
Proc opname1(formal identifiers) returns result identifier{
    declarations of local variables;
    statements
}
… # other procedure bodies
End mname;
```

Интермодулни и интрамодулни позиви

- Позивање је call mname.opname(arguments)
- mname се може изоставити код интрамодулних позива
- За операције унутар модула се претпоставља да су у истом адресном простору
- Нови процеси се креирају кад год је позив интермодулни
- Серверски процес може да постане клијентски да би испунио свој део одговорности

Bounded Buffer

```
Module BoundedBuffer
    op deposit(elem);
    op fetch() returns elem;
Body
    elem buf[n];
    int front = 0, rear = 0;
    sem empty = n, full = 0, mutexD = 1, mutexF = 1;
Process deposit(item){
    wait(empty); wait(mutexD);
    buf[rear] = item; rear = (rear + 1) % n;
    signal(mutexD); signal(full);
}
```

Bounded Buffer

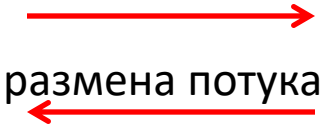
```
…
Process fetch() returns item{
        wait(full); wait(mutexF);
        item = buf[front]; front = (front + 1) % n;
        signal(mutexF); signal(empty);
}
End BoundedBuffer; #Fig of Andrews
```

Rendezvous – идеја

Позивалац

Рачунар А

...
call opname(ARGS);
…



Позвани

Рачунар Б

op opname(FORMALS); # declaration

... # existing process
in opname(FORMALS)
    ...
ni

Рандеву – *Rendezvous*

- Понаша се као обичан позив процедуре, притом позивалац и позвани могу бити на различитим машинама. Позиви су исти као код *RPC*.
- Позиви се опслужују од стране постојећег процеса
  - не креирају се нови процеси.
- Подразумевају и комуникацију и синхронизацију.
- Секвенцијално опслуживање операција.
- На више места се може прихватити захтев.

Рандеву – декларација

```
Module mname
    op opname1(formals);
    … # headers of exported operations
Body
Process P1{
    Variable declarations;
    Initialization code;
    ...
    in opname1(formal identifiers) -> S; ni
    ...
}
… # other processes bodies
End mname;
```

S је заштићени исказ у коме су формални идентификатори видљиви
in (accept) је блокирајуће – како да избегнемо да се блокирају програми?

Заштићене алтернативне операције

in op1(formals1) and B1 by e1 -> S1;

[] …

[] in opn(formalsn) and Bn by en -> Sn;

ni

- Bi – синхронизациони изрази – Boolean – одређују када је алтернатива отворена
- ei – израз за распоређивање (*prioritet*)
- Bi, ei – опционо
- У Ада програмском језику – *select* исказ и *accept* исказ

Заштићени искази

- Успевају ако је операција позвана и синхронизациони исказ је *true*
- Извршавање in је закашњено све док нека заштита не успе
- Ако више од једне заштите успева, о приоритету одлучује израз за распоређивање
- Ако нема израза за распоређивање – најстарији позив операције која успева
- Обезбеђују флексибилност да се испуне различити захтеви

Bounded Buffer

```
Module BoundedBuffer
      op deposit(elem);
      op fetch(result elem);
Body
Process Buffer {
      elem buf[n]; int front = 0, rear = 0, count = 0;
      while(true){
            in deposit(item) and count < n ->
                  buf[rear]=item; count++; rear = (rear + 1) % n;
            [] fetch(item) and count > 0 ->
                  item=buf[front]; count--; front = (front + 1) % n;
            ni
      }
}
End BoundedBuffer; #Fig.8.5 of Andrews
```

Time server

```
Module TimeServer
      op gettime() returns int;
      op delay(int);
      op tick();
Body
Process Timer {
      int tod=0;
      while(true) {
            in gettime() returns time -> time = tod;
            [] delay(waketime) and waketime <= tod -> skip;
            [] tick() -> tod++;
            ni
      }
}
End TimeServer; #Fig.8.7 of Andrews
```

Разлика између *RPC* и рандевуа

- *RPC* – концептуално сваки позив операције (процедуре) – креира нови процес
- *Rendezvous* – у оквиру постојећег процеса, постоји операција која може да се изврши са *accept* исказом, где серверски процес чека да буде позван.
- Оба су синхрона – блокирајућа

Ада

- Програм се састоји од:
  - потпрограма (функција или процедура),
  - таскова (*task* /посао, задатак/ представља назив за независан процес, при чему се процеси могу конкурентно извршавати) и
  - пакета (*packages*).
- Заштићени типови представљају језичку конструкцију која је најсличнија концепту монитора.
- Комуникација по принципу рандева.
- Постојање селективне наредбе.

Рандеву

- Тачака улаза (*entry declaration*) дефинише операцију коју дати процес сервисира:

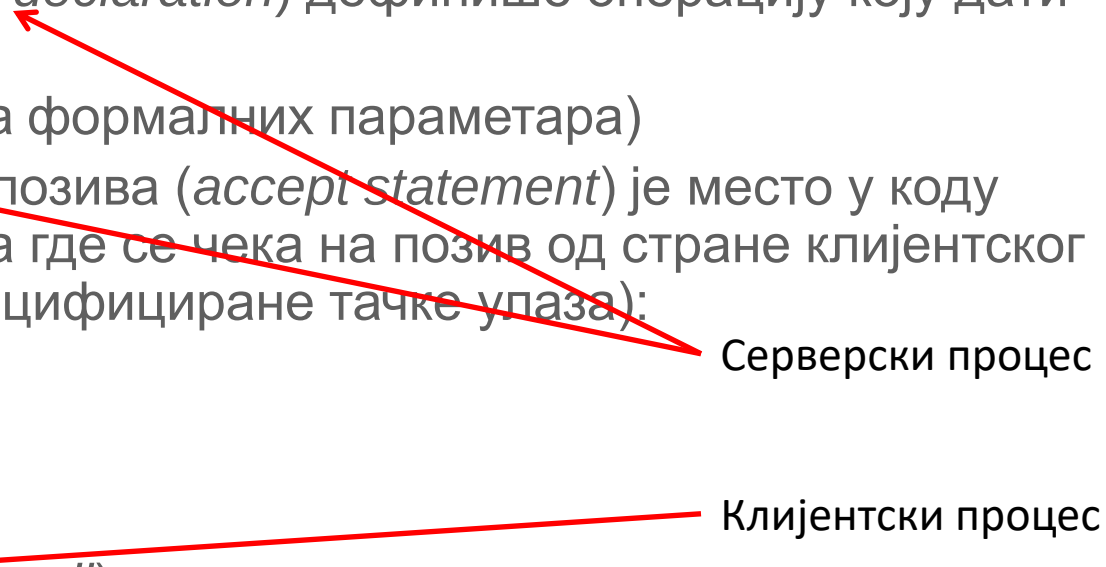
  entry <Entry>(листа формалних параметара)
- Тачка прихватања позива (*accept statement*) је место у коду серверског процеса где се чека на позив од стране клијентског процеса (преко специфициране тачке улаза):

  accept <Entry> **do**

  листа наредби

  end;
- Тачка позива (*entry call*) изгледа слично као позив процедуре: именује се процес, тачка улаза и прослеђују стварни параметри:

  <Task>.<Entry>(листа стварних параметара)



Селективна наредба

- Селективна наредба:
  - селективно чекање (*selective wait*),
  - условни позив (*conditional entry call*) и
  - временски ограничен позив (*timed entry call*).
- Селективне наредбе су недетерминистичке.

select
  [**when** condition$_1$ =>] selective_wait_alternative
or
...
or
  [**when** condition$_N$ =>] selective_wait_alternative
[**else**
  sequence_of_statements]
end select;

Клијент/сервер

```
SERVER
task A is
   entry check_point
        (Input : INTEGER; Output : out INTEGER)
   end A;
task body A is
   Local : INTEGER;
begin
   ...
   accept check_point
       (Input : INTEGER; Output : out INTEGER) do
           Local := Input;
           Output := f(Input);
   end check_point;
   ...
end A;
```

Клијент/сервер

```
CLIENT
task B;
task body B is
  To_A, From_A : INTEGER;
begin
   To_A := ...;
   ...
   A.check_point(To_A, From_A);
   ...
end B;
```

Лицитација у Ади

```
task BID is
   entry raise_bid (From:BIDDER_ID;By:NATURAL);
   entry look_at (Value:out NATURAL);
end BID;
task body BID is
   Current_Bid : NATURAL := 0;
   Last_Bidder : BIDDER_ID;
begin
   accept raise_bid (From:BIDDER_ID;By:NATURAL) do
      Last_Bidder := From;
      Current_Bid := Current_Bid + By;
   end raise_bid;
```

Лицитација у Ади(1)

```
loop
      select
         accept raise_bid (From:BIDDER_ID;By:NATURAL) do
            Last_Bidder := From;
            Current_Bid := Current_Bid + By;
         end raise_bid;
      or accept look_at (Value:out NATURAL) do
            Value := Current_Bid;
         end look_at;
      or delay 10.0;
         exit;
end select;  ...
end BID;
```

Измењена лицитација у Ади

```
select
   accept raise_bid ...
or when raise_bid'COUNT = 0 =>
accept look_at ...
or when Current_Bid >= Reserve_Value =>
   delay 10.0;
   exit;
end select;
```

Питања?