Conic

Декларација процеса

- Процес (модул) се у синтакси CONIC-а декларише на следећи начин:

task module name

декларација улазних и излазних портова

декларација локалних променљивих

begin

...

end.

Комуникационе примитиве

- **CONIC подржава асинхрону комуникацију прослеђивањем порука уз индиректно именовање процеса помоћу тзв. *портова* (*ports*).**

- Декларишу се на следећи начин:
  - улазни унидирекциони порт: **entryport** IN1:itype
  - улазни бидирекциони порт: **entryport** IN2:itype **reply** ritype
  - излазни унидирекциони порт: **exitport** OUT1:otype
  - излазни бидирекциони порт: **exitport** OUT2:otype **reply** rotype

Слање/пријем порука

Постоје два типа примитива за слање и пријем порука.

- Први, тзв. notify тип служи за унидирекциону комуникацију преко једносмерног канала
- Други, тзв. *request-reply* тип служи за бидирекциону комуникацију преко двосмерног канала

Слање/пријем порука - notify

- слање поруке на излазни унидирекциони порт:
  send msg **to** OUT1

- пријем поруке са улазног унидирекционог порта:
  receive msg **from** IN1

Слање порука - request-reply

- слање поруке на излазни бидирекциони порт:

  send msg to OUT2

  wait rep ⇒ ...

  timeout t ⇒ ...

  fail ⇒ ...

  end

- или

  send msg to OUT2 wait rep

Пријем порука - request-reply

- пријем поруке са улазног бидирекционог порта:

  receive msg from IN2 ⇒

  ...

  reply rep

  ...

  end

- **или**

  receive msg from IN2 reply rep ⇒

  ...

  end

Алтернативна команда

```
select
   when cond1; receive m1 from in1 ⇒ ...
or
   ...
or
   timeout t ⇒ ...
else
   ...
  end select
```

У CONIC-у је алтернативна команда детерминистичка

Конфигурациони језик

- инсталација типова процеса који се користе у програму:
  - USE type1,...,typeN;
- уклањање типова процеса који се користе у програму:
  - REMOVE type1,...,typeN;
- креирање инстанци процеса (модула):
  - CREATE mod1,...,modN:type;
- брисање инстанци процеса (модула):
  - DELETE mod1,...,modN;

Конфигурациони језик

- креирање веза између излазних и улазних портова процеса (модула):
  - LINK mod1.exitport,...,modN.exitport TO mod.entryport;
  - LINK mod.exitport TO mod1.entryport,...,modN.entryport;
- уклањање веза између излазних и улазних портова процеса (модула):
  - UNLINK mod1.exitport,...,modN.exitport FROM mod.entryport;
  - UNLINK mod.exitport FROM mod1.entryport,...,modN.entryport;

Задаци

Копирање поруке

Написати програм на CONIC-у који прихвата улазну поруку и шаље њене копије на сваки од два излазна порта.

Копирање поруке

```
task module splitter;
    entryport       input: string reply signaltype;
    exitport        output1: string reply signaltype;
                    output2: string reply signaltype;
    const waitperiod = 150; 3 seconds
    var buffer: string;
    begin
        loop
                    receive buffer from input reply signal =>
                            send buffer to output1
                                    wait signal
                                    timeout waitperiod => do nothing
                            end;
                            send buffer to output2
                                    wait signal
                                    timeout waitperiod => do nothing
                            end;
                    end;
end.
```

Бидирекциони прстен

- На језику CONIC дефинисан је тип модула

```
TASK MODULE node;
ENTRYPORT      in1: mess;
               in2: mess;
EXITPORT       out1;
               out2;
BEGIN...END.
```

Бидирекциони прстен

- Конкретне инстанце оваквих модула могу се дефинисати и уклањати командама конфигурацијског језика, као што су

  CREATE A, B: node;

  DELETE A;

- а везе између њих могу се успостављати и раскидати командама облика

  LINK A.out1 **TO** B.in1;

  UNLINK A.out1 **FROM** B.in1;

Бидирекциони прстен

- У овом задатку модули типа node употребљени су за симулацију чворова локалне мреже која има топологију бидирекционог прстена, као што је нпр. FDDI. Коришћењем наведених команди конфигурацијског језика система CONIC,
- (a) креирати три чвора A, B, C и повезати их у бидирекциону прстенасту мрежу,
- (b) специфицирати промену конфигурације под (a) насталу креирањем новог чвора D и његовим укључењем у бидирекциони прстен између A и C.

Бидирекциони прстен

- (c) специфицирати промену конфигурације која настаје у мрежи под (b) приликом отказа чвора B (приказати реконфигурацију повратним превезивањем, као у FDDI, на чворовима суседним чвору B, а затим уклонити чвор B).
- (d) специфицирати промену конфигурације којом ће мрежа под (c) постати бидирекциони прстен са чворовима A, C и D.

Бидирекциони прстен - а

```
CREATE A, B, C: node;
LINK    A.out1 TO B.in1;
        B.out1 TO C.in1;
        C.out1 TO A.in1;
        A.out2 TO C.in2;
        C.out2 TO B.in2;
        B.out2 TO A.in2;
```

Бидирекциони прстен - б

```
CREATE D: node;
UNLINK C.out1 FROM A.in1;
       A.out2 FROM C.in2;
LINK   C.out1 TO D.in1;
       D.out1 TO A.in1;
       A.out2 TO D.in2;
       D.out2 TO C.in2;
```

Бидирекциони прстен - ц

```
UNLINK A.out1 FROM B.in1;
       C.out2 FROM B.in2;
       B.out1 FROM C.in1;
       B.out2 FROM A.in2;
LINK   A.out1 TO A.in2;
       C.out2 TO C.in1;
DELETE B;
```

Бидирекциони прстен - д

```
UNLINK A.out1 FROM A.in2;
       C.out2 FROM C.in1;
LINK   A.out1 TO C.in1;
       C.out2 TO A.in2;
```

Bounded Buffer

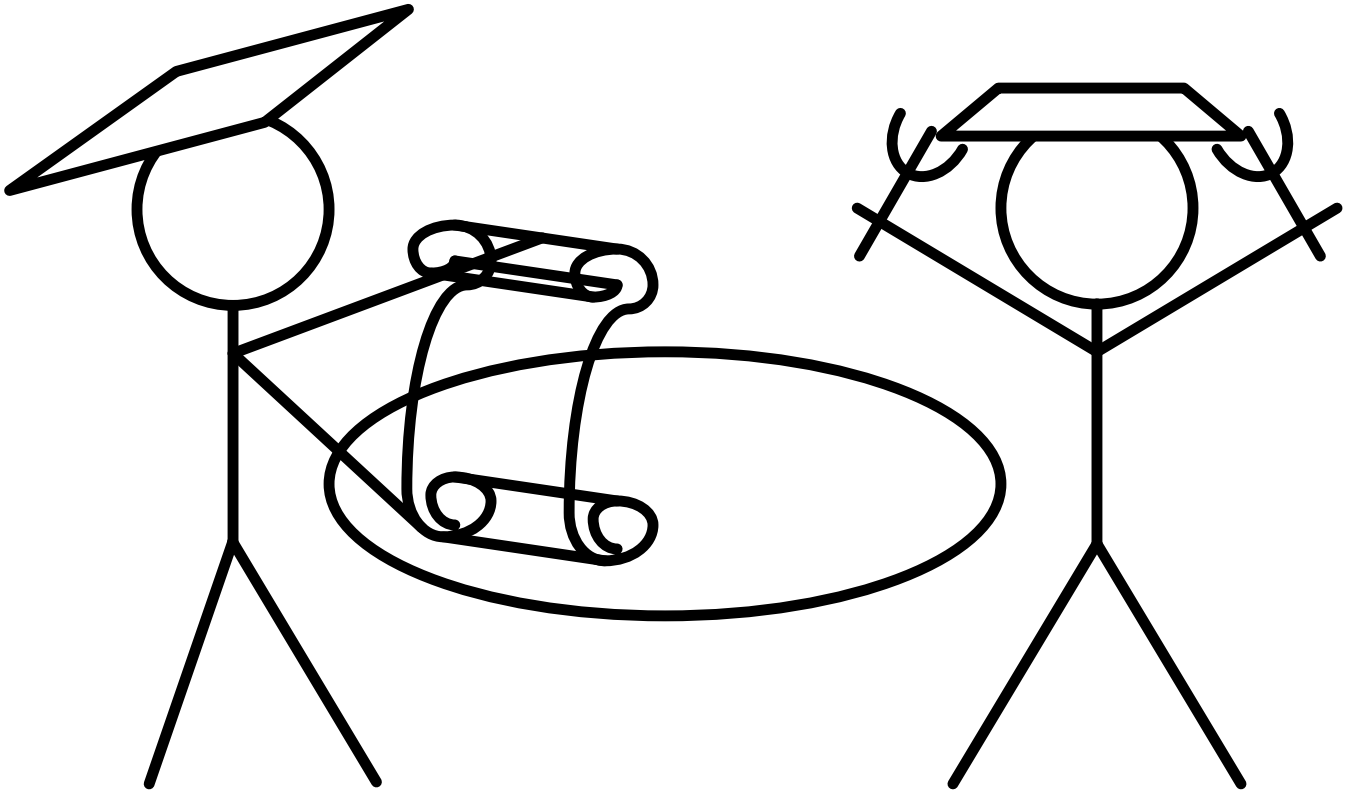
Bounded Buffer

```
task module bound;
    entryport      put_char : char reply signaltype;
                   get_char : signaltype reply char;
    const max_size = 100;
    var   inp,
          outp,
          contents : natural;
          buffer : array [1..max_size] of char;
    begin
          inp := 1;
          outp := 1;
          contents := 0;
```

Bounded Buffer

```
        loop
                select
                        when (contents < max_size) {buffer not full}
                        receive buffer[inp] from put_char reply signal
                        => inp := (inp mod max_size) + 1;
                                contents := contents + 1;
                        end;
                or
                        when (contents > 0) {buffer not empty}
                        receive signal from get_char reply buffer[outp]
                        =>  outp := (outp mod max_size) + 1;
                          contents := contents - 1;
                        end;
                end;
        end {loop}
end {task}.
```

Dining philosophers problem



Dining philosophers problem

```
define philos:forktype;
    type forktype = (pickup, putdown);
end.
task module phil (thinktime, eattime : integer);
    use philos : forktype;
    exitport        rightfork : forktype reply signaltype;
                    leftfork: forktype reply signaltype;
                    leavetable : signaltype reply signaltype;
                    sittable : signaltype reply signaltype;
    type activitytype = (thinking, sitting, eating);
    var   request, ok : signaltype;
          pickupreq, putdownreq : forktype;
```

Dining philosophers problem

```
begin
        request := signal;
        pickupreq := pickup; putdownreq := putdown;
        loop
                {thinking}
                delay (thinktime);
                send request to sittable wait ok;
                {sitting}
                send pickupreq to leftfork wait ok;
                send pickupreq to rightfork wait ok;
                {eating}
                delay (eattime);
                send request to leavetable wait ok;
                send putdownreq to leftfork wait ok;
                send putdownreq to rightfork wait ok;
        end;
end.
```

Dining philosophers problem

```
task module fork;
    use philos : forktype;
    entryport      rightphil : forktype reply signaltype;
                   leftphil : forktype reply signaltype;
    type allocationtype = (none, left, right);
    var   allocated : allocationtype;
          request : forktype;
```

Dining philosophers problem

```
begin
    allocated := none;
    loop
    select
        when ((allocated=none) or (allocated=left))
                receive request from leftphil reply signal
                        => case request of
                                pickup: allocated := left;
                                putdown: allocated := none;
                        end;
        or
        when ((allocated=none) or (allocated=right))
                receive request from rightphil reply signal
                        => case request of
                                pickup: allocated := right;
                                putdown: allocated := none;
                        end;
                end {select};
        end {loop};
    end.
```

Dining philosophers problem

```
task module table (n:integer); {number of philosophers}
    entryport       leave : signaltype reply signaltype;
                    sit : signaltype reply signaltype;
    var     sitting : integer;
            request, ok : signaltype;
    begin
            sitting := 0;
            ok:= signal;
            loop
                    select
                            when sitting > 0
                                    receive request from leave reply ok
                                            => sitting := sitting - 1;
                    or
                            when sitting < (n-1)
                                    receive request from sit reply ok
                                            => sitting := sitting + 1;
                    end {select};
            end {loop};
    end.
```

Dining philosophers problem

```
group module diners(n:integer);
    use phil; fork; table; {dining philospher modules}
    create table : table(n=n);
    create          family k:[0..n-1]
                    phil[k] : phil(n-k+2,2);
                    fork[k] : fork;
    link family k : [0..n-1]
                    phil[k].sittable to table.sit;
                    phil[k].leavetable to table.leave;
                    phil[k].rightfork to fork[k].leftphil;
                    phil[(k+1) mod n].leftfork to fork[k].rightphil;
end.
```

Обрада података

- Написати програм на CONIC-у који прихвата улазни податак за обраду, обрађује га и шаље одговор на излазни порт. Поред овога програм треба да враћа и број захтева који се тренутно обрађују. Од формираног модула направити нов модул који садржи n модула за обраду података, један модул за прихватање и прослеђивање захтева.

Обрада података

```
task module Machine;
    entryport        Orders, Finished: Order;
      Status: Inquiry reply integer;
    exitport
      Confirmation, Start: Order;
    var    number_of_orders: integer;
           production_order : Order;
           working : boolean;
```

Обрада података

```
begin
    numberof_orders := 0;
    workin := false;
        loop
                select
                        when (not working)
                        receive production_order from Orders =>
                        number_of_orders:=number_of_orders + 1;
                        send production_order to Start;
                        working := true;
                or
                        when (working)
                        receive production_order from Finished =>
                        number_of_orders := number_of_orders - 1;
                        send production_order to Confirmation;
                        working := false;
                or receive Inquiry from Status
                        reply number_of_orders;
        end;
end;
```

Обрада података

```
task module Worker();
    entryport       Start: Order;
    exitport        Finished: Order;

    var production_order: Order;
begin
    loop
        receive production_order from Start =>
            work();
        send production_order to Finished;
    end;
end.
```

Обрада података

```
task module Distributor(n: integer := 2);
    entryport        In: Order;
    exitport         Out[1..n]: Order;
    var number_of_orders: integer;
    var production_order: Order;
begin
    number_of_orders := 1;
    loop
        receive production_order from In =>
                number_of_orders := (number_of_orders) mod n + 1;
                send production_order to Out[number_of_orders];
    end;
end.
```

Обрада података

```
group module Manufacturing (n: integer := 2);
entryport
    Orders: Order;
    Status [1..n]: Inquiry reply integer;
exitport
    Confirmation: Order;
use
    Distributor, Machine, Worker;
create
    Distributor at Knot_X;
create family k:[1..n]
    Machine [k]: Machine at Knot[k];
    Worker[k]: Worker at W[k];
link
    Orders to Distributor.In
link family k:[1..n]
    Distributor.Out[k] to Machine[k].Orders;
    Status[k] to Machine[k].Status;
    Machine[k].Confirmation to Confirmation;
    Machine[k].Start to Worker[k].Start;
    Worker [k].Finished to Machine[k].Finished;
end.
```

Питања?

Захарије Радивојевић
Електротехнички Факултет
Универзитет у Београду
zaki@etf.rs