11. előadás

Szinkronizációs feladatok.
Az Ada taszkok (2).

Szinkronizációs feladatok

- Bizonyos erőforrásokat nem szabad konkurrensen használni
- Biztosítsuk, hogy egyszerre csak egy taszk használja
- Kritikus szakasz, kölcsönös kizárás
- Szemafor, monitor, író/olvasó, termelő/fogyasztó, evő és ivó filozófusok

Kritikus szakasz

- Critical section
- Egy tevékenységsorozat a taszkon belül
  - Például bizonyos erőforrások használata
- Garantáljuk, hogy egyszerre csak egy taszk van a kritikus szakaszában
- Olyan, mint amikor egy sínszakaszt védünk attól, hogy több vonat is ráhajtson
- Egy megoldás: szemafor

Szemafor

- E.W. Dijkstra, 1968
- Belépés a kritikus szakaszba, ha a P művelet megengedi
- Kilépéskor a V művelettel a szemafort szabadra állítjuk
- P = passeren (áthaladni)
  V = vrijmaken (szabaddá tenni)

Szemaforral védett kritikus szakaszok

```
... -- kritikus szakasz előtti utasítások
Szemafor.P;     -- blokkol: megvárjuk, amíg beenged
... -- kritikus szakasz utasításai
Szemafor.V;
... -- kritikus szakaszok közötti utasítások
Szemafor.P;
...-- kritikus szakasz utasításai
Szemafor.V;
... -- kritikus szakasz utáni utasítások
```

Blokkoló utasítás

- A szemafor P művelete felfüggeszti a hívó folyamatot, amíg az út szabaddá nem válik
- A hívó folyamat blokkolódik, vár a P hívás befejeződésére
- Az Ada implementációban ezt egy randevúval írhatjuk le a legkényelmesebben

Szemafor Ada taszkkal megvalósítva

```
task Szemafor is
        entry P;
        entry V;
end Szemafor;
```

```
task body Szemafor is
begin
        loop
                accept P;
                accept V;
        end loop;
end Szemafor;
```

Probléma a szemaforral

- A szemafor nagyon alacsony absztrakciós szintű eszköz
- Könnyű elrontani a használatát
- Akár a P, akár a V művelet meghívását felejtjük el, a program megbolondul
- Nem lokalizált a kód, sok helyen kell odafigyeléssel használni

Kölcsönös kizárás

- A szemafor segítségével kritikus szakaszokat tudunk leprogramozni
- Csak egy folyamat tartózkodhat a kritikus szakaszában
- A kritikus szakaszra (a kritikus erőforráshoz való hozzáférésre) **kölcsönös kizárást** (mutual exclusion) biztosítottunk

Monitor

- C.A.R. Hoare, 1974
- Egy másik eszköz a kölcsönös kizárás biztosítására
- Sokkal magasabb szintű, mint a szemafor
- Az objektum-elvű szemlélethez illeszkedik
- A kritikus adatokhoz való hozzáférés csak a szinkronizált műveleteken keresztül
  - egységbe zárás (encapsulation)
  - adatelrejtés
- Megvalósítás: feltételváltozó, wait, signal
- Java

Egy monitor Ada taszkkal

```
task Kizáró is
        entry Kiír( Str: String );
end Kizáró;
task body Kizáró is
begin
        loop
                accept Kiír(Str: String) do
                        Ada.Text_IO.Put_Line(Str);
                end Kiír;
        end loop;
end Kizáró;
```

kritikus tevékenység

Monitorok és az Ada

- Ha nem csak egy művelete van?
  - **select** utasítás
  - Bonyolultabb implementáció...
- Van speciális nyelvi elem, amivel még könnyebb ilyeneket csinálni: **védett egységek**
- A kölcsönös kizárásnál megengedőbb szinkronizáció?
  - író-olvasó feladat?
  - evő vagy ivó filozófusok?

Monitor versus Szemafor

- A monitor magasabb szintű eszköz
- A szemaforokkal rugalmasabban készíthetők kritikus szakaszok
  - nem kell előre elkészíteni az összes kritikus szakaszt
  - több kritikus erőforrás használatát könnyebb kombinálni egy kritikus szakaszon belül

A **select** utasítás

- A hívóban is és a hívottban is szerepelhet
- A randevúk szervezéséhez segítség

- Többágú hívásfogadás
- Feltételhez kötött hívásfogadás
- Időhöz kötött hívásfogadás
- Nem blokkoló hívásfogadás
- Termináltatás

- Időhöz kötött hívás
- Nem blokkoló hívás
- „Aszinkron select”

Többágú hívásfogadás

```
select
        accept E1 do ... end E1;  -- esetleg egyéb utasítások
or
        accept E2;   -- itt is
or
        accept E3( P: Integer ) do ... end E3;   -- itt is
end select;
```

- A hívott választ egy hívót valamelyik várakozási sorból
- Ha mindegyik üres, vár az első hívóra: bármelyik randevúra hajlandó

Több műveletes monitor

```
task body Monitor is
        Adat: Típus;  ...
begin
        loop
                select
                        accept Művelet_1( ... ) do ... end;
                or
                        accept Művelet_2( ... ) do ... end;
                ...
                end select;
        end loop;
end;
```

Végtelen taszk

- Az előző taszk végtelen ciklusban szolgálja ki az igényeket
- Sosem ér véget
- A szülője sem ér véget soha
- Az egész program „végtelen sokáig” fut

- Ha már senki sem akarja használni a monitort, akár le is állhatna...

Termináltatás

- A **select** egyik ága lehet **terminate** utasítás
  - Csak így szerepelhet terminate egy Ada programban
- Ha már senki nem akarja használni, akkor termináljon
  - Ez azt jelenti, amit az előbb mondtunk
- Emlékezzünk vissza a terminálási szabályokra
- Akkor „választódik ki” a terminate ág, ha már soha többet nem jöhet hívás az alternatívákra

A **terminate** használata

```
select
        accept E1;
or
        accept E2( ... ) do ... end;
or
        accept E3 do ... end;
or
        terminate;
end select;
```

Egy taszk terminál, ha

- komplett, és az összes tőle függő taszk terminált már;
- abnormális állapotba jutott és törlődött a várakozási sorokból;
- **terminate** utasításhoz ért, és a taszk olyan programegységtől függ, amely már komplett, és a leszármazottai termináltak már, komplettek, vagy szintén **terminate** utasításnál várakoznak;

A monitor javítva

```
task body Monitor is
        Adat: Típus;  ...
begin
        loop
                select
                        accept Művelet_1( ... ) do ... end;
                or
                        accept Művelet_2( ... ) do ... end;
                ...
                or
                        terminate;
                end select;
        end loop;
end;
```

Feltételhez kötött hívásfogadás

- A select egyes ágaihoz feltétel is rendelhető
- Az az ág csak akkor választódhat ki, ha a feltétel igaz

```
select
        accept E1 do ... end;
or
        when Feltétel => accept E2 do ... end;
...
end select;
```

A feltétel használata

- A select utasítás végrehajtása az ágak feltételeinek kiértékelésével kezdődik
- Zárt egy ág, ha a feltétele hamisnak bizonyul ebben a lépésben
- A többi ágat nyitottnak nevezzük
- A nyitott ágak közül nem-determinisztikusan választunk egy olyat, amihez van várakozó hívó
- Ha ilyen nincs, akkor várunk arra, hogy valamelyik nyitott ágra beérkezzen egy hívás
- Ha már nem fog hívás beérkezni, és van terminate ág, akkor lehet terminálni (terminálási szabályok)
- *A feltételek csak egyszer értékelődnek ki*

Általánosított szemafor (1)

```
task type Szemafor ( Max: Positive := 1 ) is
        entry P;
        entry V;
end Szemafor;
```

- Maximum Max számú folyamat tartózkodhat egy kritikus szakaszában

Általánosított szemafor (2)

```
task body Szemafor is
        N: Natural := Max;
begin
        loop
                select
                        when N > 0 => accept P; N := N-1;
                or
                        accept V; N := N+1;
                or
                        terminate;
                end select;
        end loop;
end Szemafor;
```

Általánosított szemafor (3)

```
task body Szemafor is
        N: Natural := Max;
begin
        loop
                select
                        when N > 0 => accept P do N := N-1; end P;
                or
                        accept V do N := N+1; end V;
                or
                        terminate;
                end select;
        end loop;
end Szemafor;
```

A randevú ideje

- Érdemes a randevút olyan rövidre venni, amilyen rövid csak lehet
- Csak a kommunikáció, meg ami még muszáj...
- Rövid ideig tartjuk csak fel a hívót, a randevú után megint működhetnek aszinkron módon a folyamatok („konkurrensen”)
- **Kivétel**: szimulációs feladatok, amikor épp azt akarjuk szimulálni, hogy a két fél „együtt csinál valamit”

Időhöz kötött hívásfogadás

- Ha nem vár rám hívó egyik nyitott ágon sem, és nem is érkezik hívó egy megadott időkorláton belül, akkor hagyjuk az egészet...
- Tipikus alkalmazási területek
  - timeout-ok beépítése (pl. holtpont elkerülésére)
  - ha rendszeres időközönként kell csinálni valamit, de két ilyen között figyelni kell a többi taszkra is

Időhöz kötött hívásfogadás: példa

```
select
        accept E1;
or
        when Feltétel => accept E2 do ... end;
or
        delay 3.0;
        -- esetleg egyéb utasítások
end select;
```

- A delay után Duration típusú érték van

Timeout: egy másik példa

```
loop
        select
                when Feltétel => accept E do ... end;
        or
                delay 3.0;
        end select;
end loop;
```

- Rendszeresen újra ellenőrzi a feltételt, hátha megváltozott (egy másik folyamat hatására)

Rendszeres tevékenység

- Óránként egyszer kell csinálni valamit
  - pl. naplózni
- Közben randevúzgatni kell a hívókkal
- **delay until** utasítás

- Használhatjuk az idő kezelésére az Ada.Calendar predefinit csomagot

Rendszeres tevékenység: példa

```
task body Rendszeres is
        T: Time := Clock + 3.0;
begin
        loop
                select
                        accept Randevú do … end Randevú;
                or      delay until T;
                        T := T + 3.0;
                        Rendszeres_Tevékenység;
                end select;
        end loop;
end Rendszeres;
```

Nem blokkoló hívásfogadás

- Ha most azonnal nem akar velem egy hívó sem randevúzni, akkor nem randevúzom: nem várok hívóra, futok tovább...
- Csak a hívó blokkolódjon
- Tipikus alkalmazási területek
  - Valamilyen számítás közben egy jelzés megérkezését ellenőrzöm
  - Rendszeres időközönként randevúzni vagyok hajlandó, de egyébként valami mást csinálok
  - Holtpont elkerülése (kiéheztetés veszélye mellett)

Nem blokkoló hívásfogadás: példa

```
select
        accept E1;
or
        when Feltétel => accept E2 do ... end;
else
        Csináljunk_Valami_Mást;
end select;
```

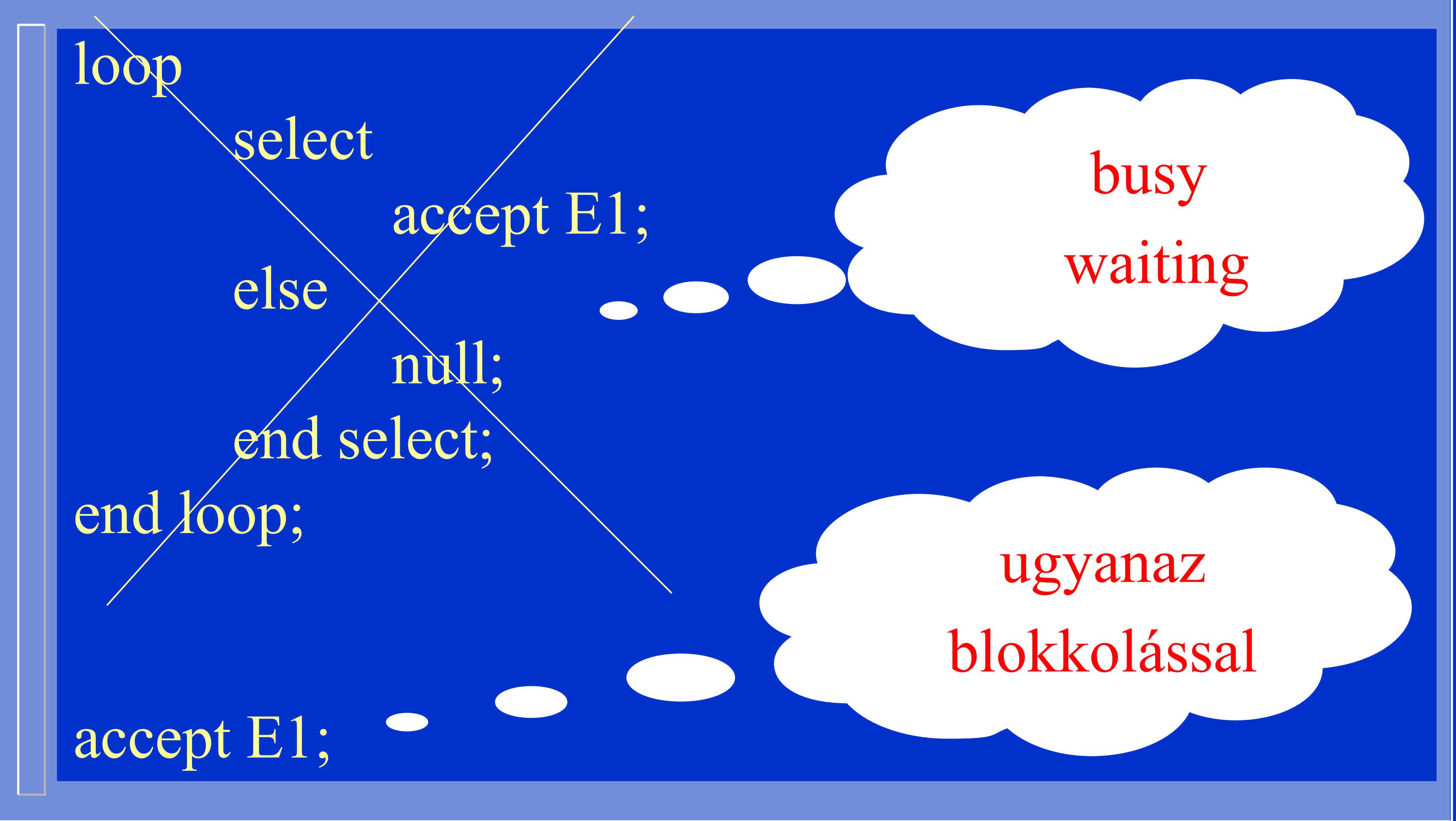
- Ha minden nyitott ág várakozási sora üres, akkor csináljunk valami mást

Egy szimulációban előfordulhat

```
select
        accept E1;
or
        when Feltétel => accept E2 do ... end;
else
        delay 3.0;     -- nem csinálunk semmit 3 mp-ig
end select;
```

- Nem ugyanaz, mint az időhöz kötött hívásfogadás!

A „busy waiting” nem szerencsés



Időhöz kötött hívás

- A hívóban szerepel „**or delay**” a **select** utasításban
- Ilyen select-ben csak két ág van
  - a hívást tartalmazó
  - és az időkorlátot megadó (timeout)

```
select
        Lány.Randi;      -- és esetleg egyéb utasítások
or
        delay 3.0;       -- és esetleg egyéb utasítások
end select;
```

Nem blokkoló hívás

- A hívóban szerepel **else** ág a **select** utasításban
- Ilyen select-ben csak két ág van
  - a hívást tartalmazó
  - és az alternatív cselekvést megadó

```
select
        Lány.Randi;        -- és esetleg egyéb utasítások
else
        Csináljunk_Valami_Mást;
end select;
```

Aszinkron select

- Megszakítható tevékenység megadása
- Megszakítás:
  - időkorlát
  - esemény (sikeres hívás)
- Nagy számításigényű tevékenység abortálása
- Ada 95

Aszinkron select: példa (időkorlátos)

```
select
        delay 5.0;
        Put_Line("Calculation does not converge");
then abort
        -- This calculation should finish in 5.0 seconds;
        --  if not, it is assumed to diverge.
        Horribly_Complicated_Recursive_Procedure(X, Y);
end select;
```

Aszinkron select: példa (fiús)

```
loop
    select
        Terminal.Wait_For_Interrupt;
        Put_Line("Interrupted");
    then abort
        -- This will be abandoned upon terminal interrupt
        Put_Line("-> ");
        Get_Line(Command, Last);
        Process_Command(Command(1..Last));
    end select;
end loop;
```

A select összefoglalása (1)

- Hívóban is és hívottban is lehet
  - de hívóban legfeljebb egy „or delay” vagy egy „else” ágat tartalmazhat a híváson kívül (kivéve az aszinkron select esetét)
  - a hívottban több alternatív ág is lehet, esetleg feltétellel
  - egy select-en belül nem lehet a taszk hívó is és hívott is
- or delay: időkorlát szabása
- else: blokkolás kiküszöbölése
- terminate: a hívott termináltatása, ha már senki nem fogja hívni

A select összefoglalása (2)

- Az or-ral elválasztott ágak első utasítása
  - accept
  - when Feltétel => accept
  - delay
  - terminate
- Utánuk bármi lehet még az ágban
- Az else ágban bármi lehet
- A hívottban legfeljebb egy lehet az „or delay”, a „terminate” és az „else” közül
- A hívóban még szigorúbb szabályok...

Kivételek

- Taszkok törzse is tartalmazhat kivételkezelő részt
- Taszkokkal kapcsolatos predefinit kivétel

  Tasking_Error

- Ha a főprogramot kivétel állítja le: kiíródik
- Ha más taszkot kivétel állít le: csendben elhal

Kivételek terjedése

- A taszkok egy „processzen” belül
- Közös memória (oprendszer)
- Minden taszknak saját végrehajtási verem
- Az automatikus változók így szeparáltak
- A kivételek terjedése: a végrehajtási verem felgöngyölítésével
- A kivételek egy-egy taszkra hatnak

Mikor hol?

- Ha kivétel lép fel taszk indításakor, akkor a szülőjében **Tasking_Error** váltódik ki
  - deklarációs: a szülő begin-je után közvetlenül
  - allokátoros: az allokátor kiértékelésének helyszínén
- Ha komplett/abnormális/terminált taszk belépési pontját hívjuk, a hívóban **Tasking_Error** váltódik ki a hívás helyszínén
- Ha egy randevú kivétellel zárul, akkor a kivétel mindkét randevúzó félre átterjed

Termelő-fogyasztó feladat

- Egy folyamat adatokat állít elő, egy másik adatokat dolgoz fel
- Kommunikáció: egy adatokat tartalmazó sor
  - Termelő => sor => fogyasztó
  - Levelesláda, futószalag; a Tároló általánosítása
  - Aszinkron kommunikáció
- Korlátos vagy korlátlan sor (buffer)
- Általánosítás: több termelő, több fogyasztó

Termelő folyamat

```
task type Termelő ( Sor: Sor_Access );
task body Termelő is
        Adat: Típus;
begin
        … loop
                Előállít( Adat );
                Betesz( Sor.all, Adat );
        end loop;
end Termelő;
```

Fogyasztó folyamat

```
task type Fogyasztó ( Sor: Sor_Access );
task body Fogyasztó is
        Adat: Típus;
begin
        … loop
                Kivesz( Sor.all, Adat );
                Feldolgoz( Adat );
        end loop;
end Fogyasztó;
```

A sor (buffer, FIFO)

- Több folyamat használja egyidőben
- Közös változó
- Védeni kell, például monitorral (blokkolhatjuk a termelőket/fogyasztókat)
  - Ha üres, nem lehet kivenni belőle
  - Ha tele van (korlátos eset), nem lehet beletenni
  - Egy időben egy műveletet lehet végrehajtani
    - Ha több elem van benne, esetleg lehet

      Betesz || Kivesz

Szálbiztos sor

- Osztott_Sorok sabloncsomag:
  - Sor típus
  - Szálbiztos (taszkok közötti kommunikációhoz)
  - Támaszkodik a Sorok-ra
  - Plusz szinkronizációs aspektus (monitor)

- Sorok sabloncsomag:
  - Sor típus
  - Hagyományos, nem szálbiztos

Taszkok elrejtése (1)

```
generic
        type Elem is private;
package Osztott_Sorok is
        type Sor(Max_Méret: Positive) is limited private;
        procedure Betesz( S: in out Sor; E: in Elem );
        procedure Kivesz( S: in out Sor; E: out Elem );
private
        task type Sor( Max_Méret: Positive) is
                entry Betesz( E: in Elem );
                entry Kivesz( E: out Elem );
        end Sor;
end Osztott_Sorok;
```

Taszkok elrejtése (2)

```
with Sorok;
package body Osztott_Sorok is
        procedure Betesz( S: in out Sor; E: in Elem ) is
        begin
            S.Betesz(E);
        end;
        procedure Kivesz( S: in out Sor; E: out Elem ) is
        begin
            S.Kivesz(E);
        end;
        task body Sor is … end Sor;
end Osztott_Sorok;
```

Taszkok elrejtése (3)

```
task body Sor is
    package Elem_Sorok is new Sorok(Elem);
    use Elem_Sorok;
    S: Elem_Sorok.Sor(Max_Méret);
begin
        loop
                ...
        end loop;
end Sor;
```

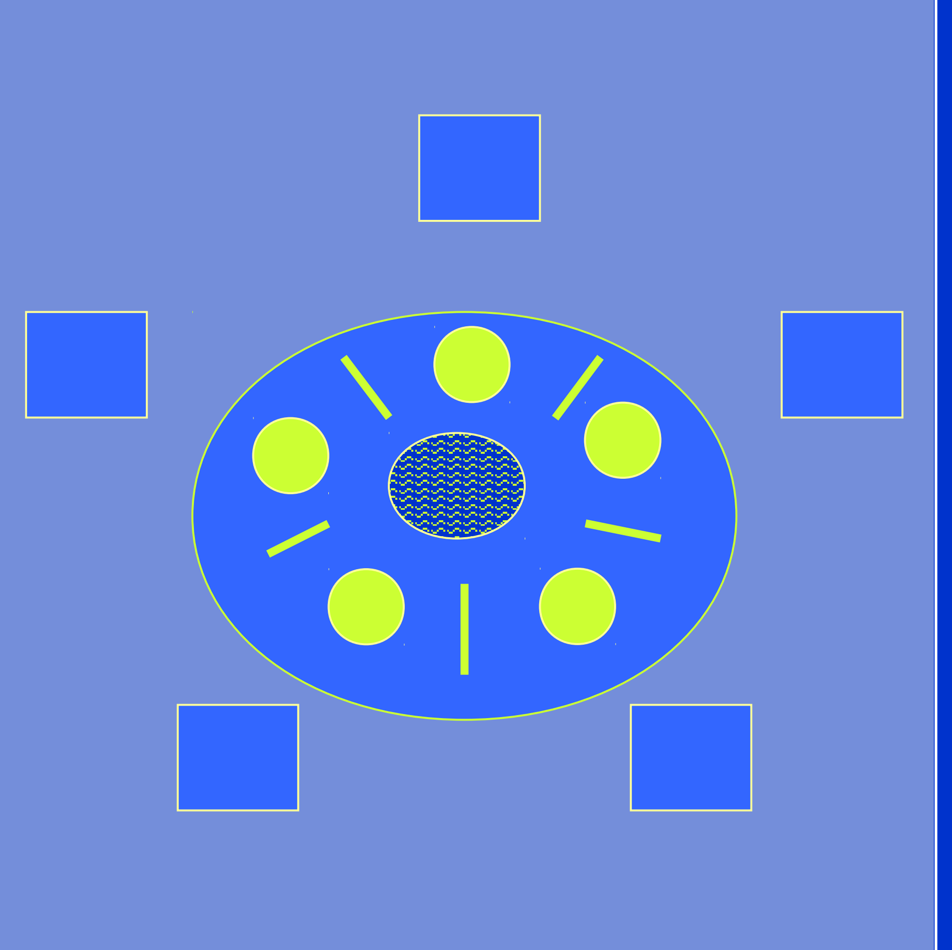
Taszkok elrejtése (4)

```
loop
    select
        when not Tele(S) => accept Betesz( E: in Elem ) do
                                Betesz(S,E);
                            end Betesz;
    or  when not Üres(S) => accept Kivesz( E: out Elem ) do
                                Kivesz(S,E);
                            end Kivesz;
    or  terminate;
    end select;
end loop;
```

Az evő filozófusok problémája

- Dijkstra: dining philosophers
- Erőforráshasználat modellezése
  - Erőforrások: villák
  - Folyamatok: filozófusok
- Holtpont kialakulása

- Általánosítás: ivó filozófusok



Holtpont

- Deadlock
- Ha a folyamatok egymásra várnak
- Tranzakciók
- Elkerülés
  - Például a szimmetria megtörésével
    - Erőforrások lekötése rendezés szerint
    - Véletlenszerű időkorlátok
- Felismerés és megszüntetés (abort)

A villák (erőforrások) modellezése

```
task type Villa is
        entry Felvesz;
        entry Letesz;
end Villa;
task body Villa is
begin
        loop
                accept Felvesz;    accept Letesz;
        end loop;
end Villa;
```

A filozófusok (folyamatok)

```
Villák: array (1..5) of Villa;
task type Filozófus ( Bal, Jobb: Positive );

Platon: Filozófus(1,2);
Arisztotelész: Filozófus(2,3);
Descartes: Filozófus(3,4);
Kant: Filozófus(4,5);
Hegel: Filozófus(5,1);
```

Az erőforrások lekötése rendezés szerint

```
task body Filozófus is
        Kisebb: Positive := Positive’Min(Bal, Jobb);
        Nagyobb: Positive := Positive’Max(Bal, Jobb);
begin
        loop
                -- gondolkodom
                Villák(Kisebb).Felvesz;
                Villák(Nagyobb).Felvesz;
                -- eszem
                Villák(Kisebb).Letesz;
                Villák(Nagyobb).Letesz;
        end loop;
end Filozófus;
```

Blokkolhatna
k

Véletlenszerű időkorlát (1)

```
task body Filozófus is
        procedure Éhes_Vagyok is … end Éhes_Vagyok;
begin
        loop
                -- gondolkodom
                Éhes_Vagyok;
                -- eszem
                Villák(Bal).Letesz;
                Villák(Jobb).Letesz;
        end loop;
end Filozófus;
```

Véletlenszerű időkorlát (2)

```
procedure Éhes_Vagyok is
    Ehetek: Boolean := False;
begin
    while not Ehetek loop
        Villák(Bal).Felvesz;
        select
                Villák(Jobb).Felvesz;   Ehetek := True;
        or      delay Véletlen_Idő;     Villák(Bal).Letesz;
        end select;
    end loop;
end Éhes_Vagyok;
```

Véletlenszámok

- Álvéletlenszámok (pseudo-random numbers)
- Valamilyen (determinisztikus) algoritmussal
- Float vagy diszkrét típusú
  - Ada.Numerics.Float_Random (csomag)
  - Ada.Numerics.Discrete_Random (sablon csomag)
- Generator, Reset, Random stb.
- Szimulációkhoz
- Taszkok esetén: szinkronizáltan

Monitorral védetten

```
task body Véletlen is
        G: Ada.Numerics.Float_Random.Generator;
begin
        Reset(G);
        loop
                select
                        accept Random ( F: out Float ) do
                                F := Random(G);
                        end Random;
                or      terminate;
                end select;
        end loop;
end Véletlen;
```

```
task Véletlen is
    entry Random ( F: out Float );
end Véletlen;
```

A véletlenszerű várakozáshoz

```
Villák: array (1..5) of Villa;

task Véletlen is entry Random( F: out Float ); end;  …
function Véletlen_Idő return Duration is
    F: Float;
begin
    Véletlen.Random(F);    return Duration(F);
end Véletlen_Idő;

task type Filozófus ( Bal, Jobb: Positive );
…
```

Író-olvasó feladat

- Van egy több folyamat által használt erőforrás
- Lehet változtatni („írni”) és lekérdezni („olvasni”)
- Az olvasások mehetnek egyidőben
- Az írás kizárólagos
- A monitornál megengedőbb

Megvalósítás Adában

- Taszkok segítségével
  - Bonyolult
  - Kell pár select bele
  - Igazságosság, pártatlanság (fairness)
- Védett egységgel

Kiéheztetés

- Livelock, starvation
- Nincs holtpont (deadlock)
- Fut a rendszer
- De: egy folyamat mindig rosszul jár
- Nem képes elvégezni a feladatát
- Példa: ügyetlen „író-olvasó” megvalósításnál
  - Az olvasók könnyen kiéheztethetik az írókat
- Példa: ügyetlen „evő filozófusok” megvalósításnál
  - Szimmetria megtörése a filozófusok beszámozásával
  - A kisebb sorszámú előnyt élvez a nagyobb sorszámúval szemben

Belépésipont-családok

- Belépési pontokból álló tömb

```
type Prioritás is (Magas, Közepes, Alacsony);
task Kijelző is
    entry Üzenet(Prioritás) ( Szöveg: in String );
end Kijelző;

accept Üzenet(Magas) ( Szöveg: in String ) do ... end;

Kijelző.Üzenet(Magas)(”Nemsokára ZH!”);
```

Egymásba ágyazott randevúk

- Egy taszk törzsében bárhol elhelyezhető accept utasítás
  - de csak a törzsben, alprogramjában már nem
- Akár még egy randevún belül is
  - „*accept E*” –ben nem lehet másik „*accept E*”
- Sőt, egy hívás is lehet randevún belül
- Nagyon veszélyes, több taszk is várni kezd
  - Holtpont is könnyen kialakulhat
- Megfontoltan csináljunk csak ilyet
  - Pl. kössük időkorláthoz a belső randevút

Appendix

Író-olvasó: Ada taszk segítségével (1)

```
task Scheduler is
        entry Start_Reading;
        entry Start_Writing;
        entry Stop_Reading;
        entry Stop_Writing;
end Scheduler;
```

- Szemafor-szerű

Író-olvasó: Ada taszk segítségével (2)

```
task body Reader_Writer is
begin
  loop
      Scheduler.Start_Reading;
      -- itt olvashatom az erőforrást
      Scheduler.Stop_Reading;
      …
      Scheduler.Start_Writing;
      -- itt írhatom az erőforrást
      Scheduler.Stop_Writing;
  end loop;
end Reader_Writer ;
```

Író-olvasó: Ada taszk segítségével (3)

```
task body Scheduler is
    Nr_Of_Readers: Natural := 0;
    Writing: Boolean := False;
begin
    loop
        select   … accept Start_Reading  …
        or       … accept Start_Writing   …
        …
        or terminate;
        end select;
    end loop;
end Scheduler;
```

Író-olvasó: Ada taszk segítségével (4)

```
select
   when not Writing =>
                  accept Start_Reading;
                  Nr_Of_Readers := Nr_Of_Readers + 1;
or when (not Writing) and then (Nr_Of_Readers = 0) =>
                  accept Start_Writing;   Writing := True;
or accept Stop_Reading;
   Nr_Of_Readers := Nr_Of_Readers - 1;
or accept Stop_Writing;   Writing := False;
or terminate;
end select;
```

az írók
éhezhetnek

Író-olvasó: Ada taszk segítségével (5)

```
select
    when (not Writing) and then Start_Writing'Count = 0) =>
                    accept Start_Reading;
                    Nr_Of_Readers := Nr_Of_Readers + 1;
or when (not Writing) and then (Nr_Of_Readers = 0) =>
                    accept Start_Writing;   Writing := True;
or accept Stop_Reading;
    Nr_Of_Readers := Nr_Of_Readers - 1;
or accept Stop_Writing;   Writing := False;
or terminate;
end select;
```

az olvasók
éhezhetnek

Író-olvasó: Ada taszk segítségével (6)

- Egy ügyesebb stratégia:
  - Egy sorban tároljuk a bejövő olvasási és írási kérelmeket
  - A sor elején álló olvasókat (az első íróig) engedjük egyszerre olvasni
  - Ha a sor elején író áll, akkor engedjük írni
- Ebben nincs kiéheztetés
- Bizonyos olvasó folyamatokat nem engedek olvasni, amikor pedig épp „olvasási szakasz” van

Micimackós szimuláció (1)

```
with Text_Io; use Text_Io;
procedure Mézvadászat is
        task type Mehecske;
        task Mehkiralyno;
        task Micimacko is
                entry Megcsipodik;
        end Micimacko;
        task Kaptar is
                entry Mezetad(Falat: out Natural);
        end Kaptar;
```

Micimackós szimuláció (2)

```
task body Kaptar is
    Mez: Natural := 3;
begin
    loop
        select   accept Mezetad (Falat: out Natural) do
                      if Mez > 0 then Mez := Mez-1; Falat := 1;
                      else Falat := 0;
                      end if;
                 end Mezetad;
        or       terminate;
        end select;
    end loop;
end Kaptar;
```

Micimackós szimuláció (3)

```
task body Micimacko is
        Nyelem: constant Duration := 1.0;
        Turelem: Natural := 10;
        Falat: Natural;
        Elfogyott: Boolean := False;
        Nincs_Tobb_Csipes: Boolean;
begin
        while (not Elfogyott) and then (Turelem>0)    loop   …
   end loop;
        if Turelem=0 then
                Put_Line("Na, meguntam a méheket...");
        end if;
end Micimacko;
```

Micimackós szimuláció (4)

```
while (not Elfogyott) and then (Turelem>0)   loop
    Kaptar.Mezetad(Falat);
    if Falat=0 then     Put_Line("Nincs több mézecske?");
                        Elfogyott := True; exit;
    else                Turelem := Turelem+Falat;
                        Put_Line("Nyelem, nyelem, majd bele szakadok");
    end if;
    select      accept Megcsipodik; Turelem := Turelem-1;
                -- további csípések (következő oldal)
    or          delay Nyelem;
    end select;
end loop;
```

Micimackós szimuláció (5)

```
-- további csípések (előző oldalról)

Nincs_Tobb_Csipes := False;
while (not Nincs_Tobb_Csipes) and then (Turelem>0) loop
    select
                    accept Megcsipodik; Turelem := Turelem-1;
    else
                    Nincs_Tobb_Csipes := True;
    end select;
end loop;
```

Micimackós szimuláció (6)

```
task body Mehkiralyno is
        Mehlarva: Natural := 10;
        Mehkeszites: constant Duration := 1.0;
        type Meh_Access is access Mehecske;
        Meh: Meh_Access;
begin
        while Mehlarva > 0 loop
                delay Mehkeszites;
                Meh := new Mehecske;
                Mehlarva := Mehlarva-1;
        end loop;
end Mehkiralyno;
```

Micimackós szimuláció (7)

```
task body Mehecske is
        Odarepul: constant Duration := 1.0;
begin
        delay Odarepul;
        Micimacko.Megcsipodik;
        Put_Line("Jol megcsiptem Micimackot!");
exception
        when Tasking_Error =>
                Put_Line("Hova tunt ez a Micimacko?");
end Mehecske;
```

Micimackós szimuláció (8)

```
begin         --- itt indulnak a taszkok
      null;
end Mézvadászat;
```