Párhuzamosság

Alapelvek

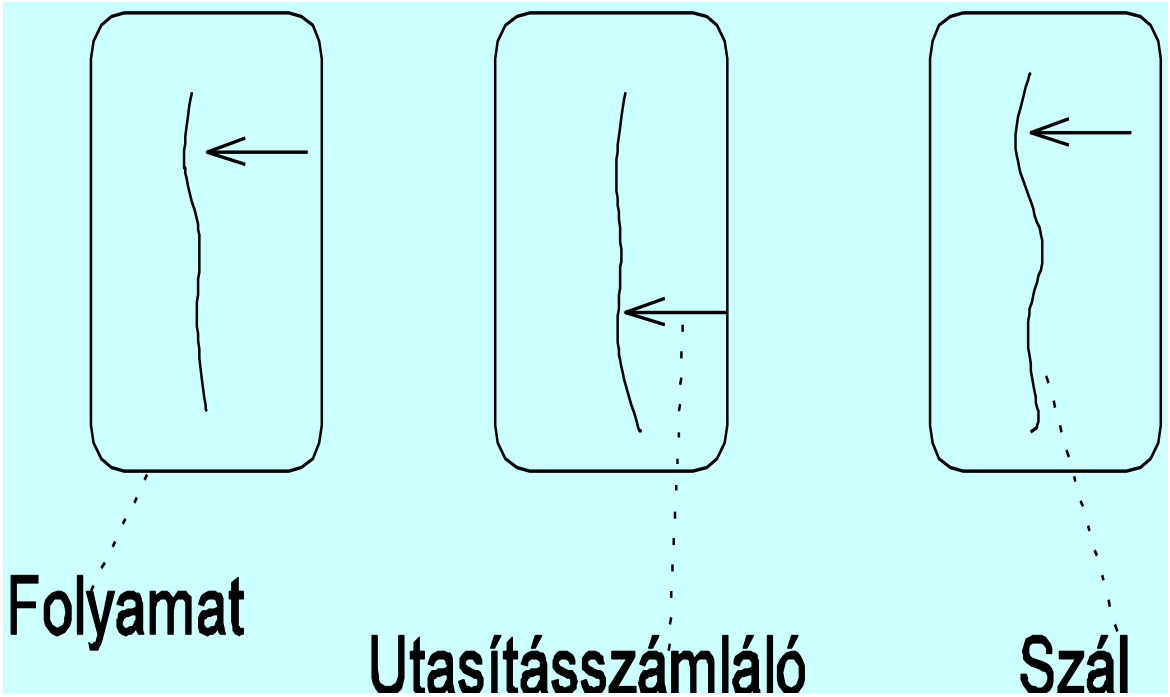
- **A hagyományos számítógépeken:**
  - **Egy adott időben egy program futhat, a számítógép architektúrája pedig egy egyprocesszoros gép. Ez azt jelenti, hogy csak egy program lehet aktív, egy adott pillanatban egy utasítás kerül végrehajtásra. A különböző programok végrehajtása egymás után történhet csak.**

- **A párhuzamosság bevezetése:**
  - **Az első próbálkozás aszinkron működésre az input-output műveleteknek a központi egység műveleteivel egy időben való végzése volt. Ezeket már alapjában véve *konkurens műveletek*nek nevezték.**
  - **A következő nagyobb lépést az operációs rendszer különböző műveleteinek párhuzamos végzése jelentette. Ezáltal megjelent a *multiprogramozás* fogalma.**
    A cél a gép erőforrásainak és aszinkron lehetőségeinek kihasználása volt.
    Különböző feladatok a készültség különböző állapotaiban párhuzamosan létezhetnek. Az ilyen rendszerekben a CPU folyamatosan váltogatja az egyes programokat, melyeket csak néhány századmásodpercig futtat. Így a CPU egyszerre csak egyetlen programot hajt végre, viszont adott idő intervallumban akár több száz programot is kiszolgálhat a párhuzamos futás illúzióját keltve.

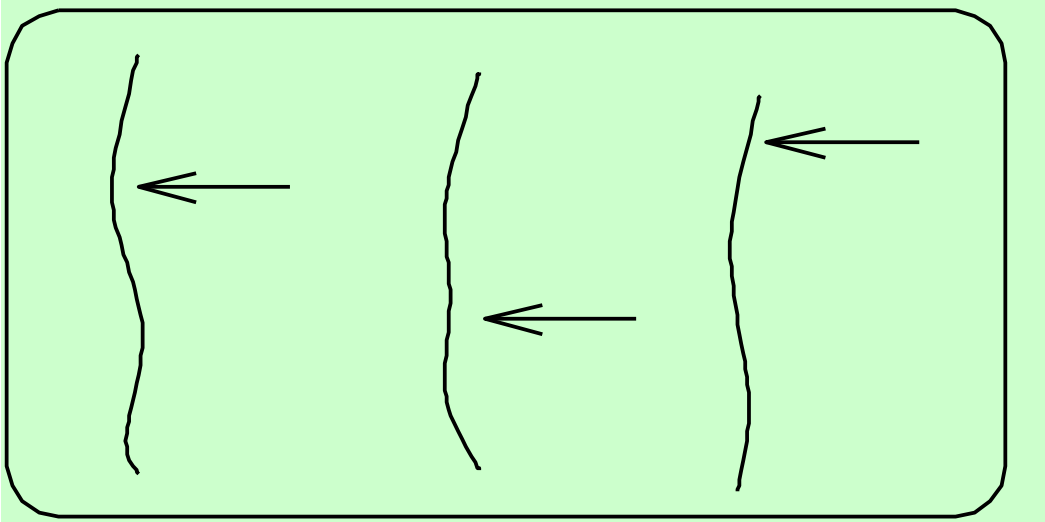
- **Folyamatok (processzek):**
  - **a *processz* egy olyan műveletsor, amelyet egy szekvenciálisan végrehajtott program valósít meg.**
  - **egy folyamat a programból, az ahhoz kapcsolódó bemenő-, kimenő adatokból és egy állapotból áll.**
  - **egy folyamatnak öt állapota lehet:**
    - **Futó: a CPU éppen ezt a folyamatot futtatja**
    - **Kész (ready): futtatható, éppen áll, hogy egy másik folyamat futhasson (ez azt jelenti, hogy futásra kész)**
    - **Blokkolt: vár egy külső esemény bekövetkezésére (pl. I/O)**
    - **Megszakított: a folyamatoknak jeleket (signal) küldhetünk, melyekre megállnak**
    - **Halott: befejeződött a folyamat**

- **Folyamatok (processzek):**
  - Amint a folyamat véget ér, befejezi futását, már nem létezik, mint folyamat többé.
  - Az első két állapot nagyon hasonlít egymásra, az egyedüli különbség az, hogy ideiglenesen a CPU nem elérhető a folyamat számára.
  - Egy egyprocesszoros környezetben a CPU több folyamat között van megosztva, és egy ütemező algoritmus szabályozza, hogy mikor kell megállítani egy folyamat feldolgozását, hogy egy másik folyamatot szolgálhasson ki.
  - A blokkolt állapotban a folyamat nem futhat tovább, még akkor sem, ha a CPU-nak nincs más feladata. Egy folyamat akkor kerül blokkolt állapotba, ha vár valamire, hogy megtörténjék, pl. egy lemez blokk beolvasására, egy karakter leütésére. Amint bekövetkezik az esemény, amelyre a folyamat várt, a processz kész állapotba kerül, és ismét ütemezhetővé válik.

- Egy hagyományos folyamatban egy vezérlő szál és egy utasításmutató található.



- A mai operációs rendszerek képesek már több vezérlő szál kezelésére egy folyamatban is.



- Egy szál koncepcionálisan hasonlít egy folyamathoz.
- A különbség:
  - a folyamat kernel szintű fogalom, ezért minden egyede speciális adatokkal rendelkezik, mint pl. virtuális memória, fájlleíró, UID, GID, PID (User IDentification number, Group ID, Process ID), stb.
  - Más folyamatok csak rendszerhívásokon keresztül férnek hozzá a folyamat adataihoz, állapotához.
  - A folyamatleíró struktúra minden része a kernelben van, így a felhasználói program nem tudja elérni azt.
  - A program eljárásai, függvényei viszont a felhasználói területen vannak, így közvetlenül elérhetőek.

- A szál felhasználói szintű fogalom -
  - A szálleíróstruktúra is a felhasználói területen van, és ezért közvetlenül elérhető.
  - A szálak, mint programokat futtató taszkok, a folyamatokhoz hasonlóan rendelkeznek regiszter adatokkal (utasításszámláló - program counter, veremmutató - stack pointer, stb.) valamint állapotleíró adatokkal. Vagyis minden szálnak például saját veremterülete van.
  - A szálaknak ugyanúgy lehet futó, kész és blokkolt állapota.
  - Egy folyamat minden szála osztozik az erőforrásokon, azaz ugyanazon memória területet látja, ugyanazokat a műveleteket és adatokat használja mindegyik szál. Ha egy szál megváltoztat egy folyamat szintű adatot, akkor a folyamat összes szála érzékelni fogja ezt az adat legközelebbi hozzáférésekor.
    (Ha egy szál megnyit egy fájlt olvasásra, akkor a többi szál is olvashatja ugyanazt a fájlt.)

- **A processzek közötti kapcsolatok formái:**
  - Kommunikáció **- ez a processzek közötti adatcserét jelenti, A kommunikáció során egy folyamat hozzáfér egy másik, vele párhuzamosan futó folyamat által szolgáltatott információhoz.**

    Történhet:

    - **osztott változók használatával - a program változóinak egy része (esetleg minden változó) hozzáférhető egyszerre több folyamat számára. Ekkor két párhuzamos folyamat úgy kommunikálhat, hogy az egyik beírja az átadni kívánt információt egy változóba, a másik folyamat pedig kiolvassa onnan. Nagyon fontos a folyamatok összehangolása, mivel ha két folyamat például egyidőben ír ugyanarra a memóriaterületre (ugyanabba a változóba,) a kapott eredmény határozatlan érték lesz… ☹**

- **A processzek közötti kapcsolatok formái:**
  - Kommunikáció (folyt.)
    - **explicit üzenetküldéssel - az egyik folyamat üzenetet küld a másik folyamatnak.**
      - **Az üzenetküldés lehet aszinkron vagy szinkron.**
      - **aszinkron kommunikáció: ha egy folyamat üzenetet küld, akkor nem várja meg, hogy a másik folyamat fogadja az adott üzenetet, a küldő folyamat végrehajtása csak az üzenet elküldésének idejére függesztődik fel.**
      - **szinkron kommunikáció (randevú): ha egy folyamat kommunikálni akar egy másik folyamattal, akkor végrehajtása felfüggesztődik addig, amíg a megszólított folyamat alkalmas nem lesz az üzenetvételre. A két folyamat csak a kommunikáció befejeztével futhat tovább.**

- **A processzek közötti kapcsolatok formái:**
  - szinkronizáció **-** az egyik processz vezérlését kapcsolatba hozza a másik processz vezérlésével - ha pl. p a P processz egy végrehajtási pontja és q egy Q processzé, akkor szinkronizációt használunk arra, hogy megszorításokat tehessünk arra, hogyan éri el
    P p-t és Q q-t.

    A folyamatok összehangolásánál több olyan egyedi probléma is felmerülhet, amelyek elkerülésére a konkrét megvalósításkor figyelnünk kell. Az egyik ilyen a holtpont (deadlock) problémája. A holtpont –leegyszerűsítve - olyan állapot, melynél a folyamatok körkörösen egymásra várakoznak, és egyik sem tud tovább haladni.

A processzek viselkedésük alapján lehetnek:

- egymástól függetlenek (independent),
- egymással versengők (competing)
  - többen szeretnének megszerezni egy adott erőforrást, amit csak kizárólagos módon lehet használni - pl. repülőgépen egy adott járat adott ülőhelye, egy adott nyomtató, stb. - kölcsönös kizárás kell, szinkronizáció
- egymással együttműködők (cooperating)
  - amikor ugyanazon probléma különböző részein dolgoznak

```
with Text_IO; use Text_IO;
procedure A is
        task T;
        task body T is
        begin
                loop
                        Put("T");
                end loop;
        end T;
begin
        loop
                Put("A");
        end loop;
end A;
```

```
with Text_IO; use Text_IO;
procedure A is
        task T;
        task body T is
        begin
           loop
                        Put("T"); delay 0.0;
        end loop;
        end T;
begin
        loop
                Put("A"); delay 0.0;
        end loop;
end A;
```

- A **multiprogramozott** rendszer szekvenciális programok halmazának tekinthető, amelyek üzenetek és szinkronizációs jelek segítségével együttműködnek egymással, és ugyanazon korlátozott számú erőforrásért versenyeznek.
- A multiprocesszoros gépek megjelenése több kérdést vetett fel. Ezek közül az egyik a **közös tár** használata, amikor is a párhuzamos folyamatok a közös (osztott) változókon keresztül kommunikálnak. Az olyan rendszert, amelyik a konkurens programozást így valósítja meg, **osztott** rendszernek nevezzük.

- **Kritikus szakasz:**

  egy multiprogramozott rendszerben különböző programok közös adatterületet használhatnak. A végrehajtás alatt vannak olyan szakaszok, amikor módosítják vagy vizsgálják a közös adatterület elemét. Ezek a szakaszok a "kritikus szakasz"-ok. Megköveteljük, hogy

  - **a kritikus szakasz végrehajtása véges idő alatt befejeződjön,**
  - **egy program, ha akar, mindig véges idő alatt beléphessen a kritikus szakaszába,**
  - **ha egy processz kritikus szakaszon kívül leáll, az ne befolyásolja a többi processzt.**

- **Kölcsönös kizárás**: annak biztosítása, hogy egyidejűleg csak egy program lehessen kritikus szakaszban.
- **Erőforrás**: A rendszer valamely alkotórésze, amelyből véges mennyiségű van csak, és több processz akarhatja egyidejűleg használni.

Kommunikációs modellek

I. Szinkron kommunikáció (randevú):

- ha egy folyamat kommunikálni akar egy másikkal, akkor végrehajtása felfüggesztődik addig, amíg a szólított folyamat hajlandó nem lesz a randevúra. A randevút kérő folyamat csak a randevú kódjának lefutása után futhat tovább. A randevú előnye az aszinkron kommunikációval szemben: az információ áramlása kétirányú lehet.

Ezt a kommunikációs modellt használja pl. az Ada nyelv.

Randevú az Adában

- Taszkok szinkronizációjához és kommunikációjához használható
- Belépési pont (**entry**) definiálható az egyik taszkban, amihez az **accept** utasítással kódot rendelünk
- A másik taszkban meghívhatjuk a belépési pontot
- A belépési pontok is "callable unit"-ok, mint az alprogramok

```
task HF is
        entry Bead;
end HF;
task body HF is
begin
        accept Bead;
end HF;
```

```
task Diák;
task body Diák is
begin
        HF.Bead;
end Diák;
```

Aszimmetrikus

- Megkülönböztetjük a hívót és a hívottat
  - diák és HF
- Egész másként működik a két fél
- Szintaxisban is kimutatható a különbség
- A hívó ismeri a hívottat, de fordítva nem
- A "hívó" és a "hívott" csak szerepek, amik egy randevúra vonatkoznak, nem egy egész taszkra
  - ugyanaz a taszk az egyik randevúban lehet hívó és a másikban hívott

Szinkron:

- A randevú akkor jön létre, amikor mindketten akarják
- Bevárják egymást a folyamatok az információcseréhez
- Az aszinkron kommunikációt le kell programozni, ha szükség van rá

Pont-pont kapcsolat

- Egy randevúban mindig két taszk vesz részt

Az **accept** törzse:

- Az **accept**-nek törzse is lehet, ez egy utasítássorozat
- A randevú az accept utasítás végrehajtásából áll
- Ez alatt a két taszk együtt van
- A **fogadó** taszk hajtja végre az **accept**-et

```
task HF is
        entry Bead;
end HF;
task body HF is
begin
        accept Bead do ... end Bead;
end HF;
```

```
task Diák;
task body Diák is
begin
        HF.Bead;
end Diák;
```

Kommunikáció:

- A randevúban információt cserélhetnek a taszkok
- Paramétereket használunk erre a célra
- Az **entry** specifikációja formális paramétereket is tartalmazhat
- A kommunikáció kétirányú lehet,
  pl. Adában a paraméterek módja szerint
  (**in**, **out**, **in out**)
- Az alprogramok hívására hasonlít a mechanizmus

```
task Tároló is
        entry Betesz( C: in Character );
        entry Kivesz( C: out Character );
end Tároló;
```

Ez a specifikációja

```
task body Tároló is
        Ch: Character;
begin
        loop
                accept Betesz( C: in Character ) do
                        Ch := C;
                end Betesz;
                accept Kivesz( C: out Character ) do
                        C := Ch;
                end Kivesz;
        end loop;
end Tároló;
```

```
        Ch: Character;
begin
        …
        Get(Ch);
        Tároló.Betesz(Ch);
        …
        Tároló.Kivesz(Ch);
        Put(Ch);
        …
end;
```

II. Aszinkron kommunikáció üzenetekkel

- ha egy folyamat üzenetet küld, akkor az üzenet a fogadó folyamatnál egy üzenetsorba kerül. A küldő folyamat végrehajtása csak az üzenet elküldésének idejére függesztődik fel. Az üzenet feldolgozásához a fogadó folyamatnak ki kell venni az üzenetsorából az üzenetet. Ilyen kommunikációs modellt valósít meg például a PVM-rendszer.

Aszinkron kommunikáció Adában:

```
task A;
task body A is
        Ch: Character;
begin
        ...
   Get(Ch);
   Tároló.Betesz(Ch);
        ...
end;
```

```
task B;
task body B is
        Ch: Character;
begin
        ...
   Tároló.Kivesz(Ch);
   Put(Ch);
        ...
end;
```

Az A és a B taszk a Tároló-n keresztül aszinkron módon kommunikálnak.

- **Közös változók**: a folyamatok ugyanazt a memóriarészt látják. Ebből az a probléma adódhat, hogy két folyamat egy időben írja az adott erőforrást (pl. változót), ilyenkor az eredmény határozatlan érték lesz. Hasonló probléma merül fel egy írási és egy olvasási művelet összeakadásánál.

A párhuzamosság célja

- részben a gép erőforrásainak kihasználása,
- részben a különböző rendszerek működésének szimulálása,
pl. gépjárműforgalom irányítása….
Az utóbbi esetben a különböző kölcsönhatásokat próbálják modellezni.

Milyen nyelvi elemekre van szüksége egy programozási nyelvnek a párhuzamosság támogatására?

- Elsősorban a párhuzamosan elindított kód részeit leíró nyelvi elemekre (task, szál, folyamat, process) és ezeknek a végrehajtását szolgáló leírásra, módszerre.
- Egy folyamat most éppen végrehajtható-e?

- **Kommunikáció: Osztott adatok esetén kölcsönös kizárást garantáló eszközök.**

Két fő irányzat:

1. nyelvi mechanizmus az adatok védésére (pl. szemaforok, monitorok).

2. a folyamatok üzenetek révén kommunikálnak - az üzenetek eljáráshívásnak felelnek meg, a megosztott adatok pedig az átadott paraméterek

- **Konkurens részek szinkronizálására megfelelő eszközök**
- **A prioritások meghatározása**
- **Késleltetések, időzítések kezelése**

A gyakrabban használt nyelvi elemek meghatározása

- **Szemafor**

A szemafort, mint típust, Dijkstra vezette be egy 1968-as művében; a szemafor egy egész értékű számláló, és a hozzá tartozó várakozási sor.

Egy inicializált szemafornak két megengedett művelete van:

P = passeren (áthaladni)
V = vrijmaken (szabaddá tenni)

A p -- wait**, a v --** signal.

A műveletek szemantikája az alábbi módon van definiálva:

S: semaphore;

wait(S) : Ha S>0, akkor S:=S-1, különben a folyamat blokkolódik, és a szemaforhoz tartozó várakozási sorba kerül mindaddig, amíg valaki fel nem ébreszti (azaz rá vagyunk utalva a többi folyamatra).

signal(S) : Ha van várakozó folyamat az S-hez tartozó várakozási sorban, akkor felébreszti, különben S:=S+1

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

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

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

Általánosított szemafor

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

- Legfeljebb Max számú folyamat tartózkodhat egy kritikus szakaszában

Általánosított szemafor megvalósítása

```
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;
```

- ***Monitor***

  A monitor a folyamatok szinkronizálására használt eszköz és az osztott adatokról ad információt.

  Szerkezete:

  monitor_név

  begin

  lokális adatok;

  eljárások;

  lokális adatok értékadásai;

  end név.

  A monitor biztosítja, hogy csak egy folyamat hajthat végre egy monitor-eljárást egy adott időben. Mielőtt egy másik folyamat belép a monitorba, az előzőnek véget kell érnie.
  Sok nyelvben megtalálható a monitor, mint könyvtári osztály.

Csak egy taszk írhat ki egyszerre

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

```
protected Védett is
    procedure Kiír ( ... );
end Védett;

protected body Védett is
    procedure Kiír ( ... ) is ... end;
end Védett;
```

```
Védett.Kiír(...);
```

Lehetséges nyelvi elemek még:

- **A cobegin-coend a párhuzamos folyamat kezdeti, illetve végpontját jelöli. Ugyanilyen nyelvi elem a parbegin-parend is.**
- **A select feltételhez kötött párhuzamos indítást valósít meg.**
- **A wait késleltet egy folyamatot.**
- **A fork egy párhuzamos folyamat elindítását váltja ki.**
- **A join újra egyesíti a párhuzamos folyamatokat.**
- **A quit befejez egy processzt.**

A **select** utasítás

- A hívóban is és a hívottban is szerepelhet – lehetséges magatartásformák támogatása
- A randevúk szervezéséhez segítség
  - Többágú hívásfogadás
  - Feltételhez kötött hívásfogadás
  - Időhöz kötött hívásfogadás
  - Feltételes hívásfogadás
  - Termináltatás

hívott

  - Időhöz kötött hívás
  - Feltételes hívás

hívó

Többágú hívásfogadás

```
select
        accept E1 do ... end E1;
or
        accept E2;
or
        accept E3( P: Integer ) do ... end E3;
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
- Ez azt jelenti, amit az előbb mondtunk
  - ha már senki nem akarja használni, akkor terminál jon
- 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;
```

```
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*

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 a csudába...
- 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

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

```
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)

Feltételes hívásfogadás

- Ha most azonnal nem akar velem egy hívó sem randevúzni, akkor nem randevúzom. Nem várok hívóra, futok tovább...
- 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)

```
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

```
select
        accept E1;
or
        when Feltétel => accept E2 do ... end;
else
        delay 3.0;
end select;
```

- Nem ugyanaz, mint az időhöz kötött hívásfogadás!

```
loop
        select
                accept E1;
        else
                null;
        end select;
end loop;
```

busy-waiting

```
accept E1;
```

blokkolás

Amíg a szükséges helyzet elő nem áll, a folyamat dinamikusan várakozik, azaz folyamatosan vizsgálja, hogy a kívánt feltételek teljesülnek-e.

Hátránya, hogy a folyamat várakozás közben foglalja a processzoridőt.

Időhöz kötött hívás

- A hívóban szerepel „**or delay**” a **select** utasításban
- Ilyen select-ben csak két ág van
  - a hívás
  - és az időkorlát – eddig vár a randevúra

```
select
        Lány.Randi;
or
        delay 3.0;
end select;
```

Feltételes hívás

- A hívóban szerepel **else** ág a **select** utasításban
- Ilyen select-ben csak két ág van
  - a hívás
  - és az alternatív cselekvés

```
select
        Lány.Randi;
else
        Csináljunk_Valami_Mást;
end select;
```

Író-olvasó probléma

- többen olvashatnak
- egyszerre csak **egy** írhat

```
protected Változó is
   function Lekérdez return Adat;
   procedure Beállít (Új: in Adat);
private
   V: Adat;
end Változó;

protected body Változó is
   function Lekérdez return Adat is
      begin return V; end Lekérdez;
   procedure Beállít (Új: in Adat ) is
      begin V := Új; end Beállít;
end Változó;
```

Micimackó

```
with Text_Io; use Text_Io;
procedure Micimac is
  task type Mehecske;
  task Mehkiralyno;
  task Micimacko is
     entry Megcsipodik;
  end Micimacko;

  task Kaptar is
    entry Mezetad(Falat: out Natural);
  end Kaptar;
```

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

```
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;
```

```
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;
```

```
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
    Kaptar.Mezetad(Falat);
    if Falat=0 then
       Put_Line("Nincs több mézecske?");
       Elfogyott := True; exit;
    else
       Turelem := Turelem+Falat;
        Put_Line("Nyelem, nyelem, jaj de finom");
    end if;
```

```
 select
   accept Megcsipodik do Turelem := Turelem-1;
          end Megcsipodik;
   Nincs_Tobb_Csipes := False;
   while (not Nincs_Tobb_Csipes) and then (Turelem>0)loop
    select
      accept Megcsipodik do Turelem := Turelem-1;
             end Megcsipodik;
      else
         Nincs_Tobb_Csipes := True;
     end select;
   end loop;
 or
   delay Nyelem;
 end select;
end loop;
 if Turelem=0 then
   Put_Line("Na, meguntam a méheket...");
 end if;
end Micimacko;
```

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

A párhuzamos programozási nyelvek osztályozása és jellemzése

- **Statikus párhuzamosságot támogató nyelveknél (pl. Concurrent Pascal) csak a program legkülső szintjén definiálhatók folyamatok, melyek implicit módon jönnek létre. A folyamatok száma állandó, és fordítási időben meghatározható.**
- **Fél statikus nyelvek esetén folyamatok bármely szinten definiálhatóak, és ezek a megfelelő programszint elérésekor implicit módon jönnek létre. A folyamatok száma ugyan változik a futás során, de fordítási időben felső korlát adható az egyidejűleg futó folyamatok számára (pl. Eiffel, Ada).**
- **Dinamikus párhuzamosságot támogató nyelvek esetében folyamatok a program bármely szintjén definiálhatóak és explicit módon bármikor létrehozhatóak. A folyamatok száma csak futási időben mérhető (pl. Modula-3, Java, C#).**

Másik osztályozás:vezérlési modellek

- **„Control driven” típusú vezérlés esetén az elindított folyamatok ellenőrzése fork, join, wait parancsokkal történik. Ez egy központosított ellenőrzés, mivel a folyamatok közös memóriaésszel dolgoznak.**
- **A „data driven computation” modellben a folyamatok közvetlen módon kommunikálnak, közös memória használata nélkül. A végrehajtási sorrend az adatok közötti összefüggésen alapul.**
- **A „demand driven computation” modellben a folyamatok akkor kerülnek végrehajtásra, amikor ezt egy másik folyamat kifejezetten kéri, a vezérlést mindig a kérések határozzák meg. Például: Parlog, Concurrent Prolog, GHC.**

Ezeknek a modelleknek megfelel egy-egy programozási nyelvosztály.

- **Az ellenőrzéssel vezérelt nyelvek csoportja lehet: szinkron vagy aszinkron programozási nyelv.**

A szinkron nyelvek esetében a folyamatok ugyanolyan típusú műveleteket hajtanak végre ugyanolyan típusú adatcsomagokra. Ezeket a többprocesszoros gépek programozási nyelveiként említik.

Ide sorolhatók pl. a különböző Fortran verziók.

Az aszinkron nyelveket konkurens és osztott programozásra használják. Ezeket három nagy csoportba sorolhatjuk:

- Eljárás orientált nyelvek: a folyamatok közötti kommunikáció az osztott változók segítségével történik. (Például: Concurrent Pascal, Pascal Pluss, Modula-2, Ada.)
- Üzenetküldés orientált nyelvek: a kommunikálás send-receive típusú utasításokkal történik. (Például: Occam, Ada, CSP.)
- Művelet orientált nyelvek: az előbbiek kombinációi. (Például: StarMod, Ada.)

Szinkron párhuzamos programozás

Ez a típusú programozás a tömbök és mátrixok feldolgozását valósítja meg a párhuzamos architektúrájú gépeken. A gépek vektor vagy mátrix processzoros gépek, melyek azonosan strukturált és ugyanolyan természetű adatokkal dolgoznak. A processzek közötti kommunikáció csak az ezek közötti adatcserére szorul, így itt nem merül fel a szinkronizálás és a kölcsönös kizárás problémája.

Csoportosításuk:

- A különböző Fortran verziók (IV Fortran, CFT, Cyber Fortran) a DO ciklusból próbálják kiszűrni a párhuzamosítható utasításokat, így ezek automatikusan derítik ki a párhuzamosságot. Előnye az, hogy nagyon sok programot lehet újrahasznosítani, de ugyanakkor a forráskód átstrukturálását is kéri, ami viszont hátrány.
- A párhuzamosság direkt megadásánál (CFD és DAF Fortran) már szükséges a párhuzamos számítógép szerkezetének ismerete, ezért az ilyen nyelvek szintaktikája és szemantikája elő kell segítse a gép architektúrájának kihasználását. A feladat párhuzamos megoldása nem mindig egyezik a gép által nyújtott párhuzamossággal. Ezért gyakran át kell szervezni az adatstruktúrákat és a programok nem lesznek hordozható programok.
- A szinkron nyelvek harmadik csoportjába sorolhatjuk azokat, amelyek a párhuzamosságot már a gép architektúrájától függetlenül próbálják megvalósítani.

- ***Osztott változókkal rendelkező nyelvek*: Ada, Pascal Plus, Concurrent Pascal, Modula-2.**
- ***Üzenetküldéssel, osztott memóriával rendelkező nyelvek*: Ada, CSP, Occam.**
- ***Nem kifejezetten párhuzamos nyelvek, amelyek azonban rendelkeznek osztott memóriával*: Pure Lisp, Val, Loral, Dataflow, Lucid adat-folyam nyelvek.**
- ***Objektum-orientált párhuzamos nyelvek* (pl. Emerald, Pool).**

Osztott változókkal rendelkező nyelvek

- Concurrent Pascal

A processz, monitor és osztály fogalmaival dolgozik, valamint a delay és continue utasításokkal és a queue várakozási sor típussal. A processz szekvenciális program, mely egy időben futtatható más processzel. A monitor a processzek között osztott változók egységbezárása, az osztály pedig absztrakt adattípus. Az osztály példánya egy processzhez vagy monitorhoz csatolható. A monitor biztosítja adatainak a védelmét, a monitor eljárásai pedig egy várakozási sorban állnak, egy adott pillanatban csak 1 eljárás futtatható. A monitor egy eljárása felfüggeszthető a delay-vel vagy folytatható a continue-val, az inicializálás pedig init-tel történik.

- Modula-2

A processzek fogalmára épít. A processzek közötti kommunikáció kétféleképpen valósul meg: osztott változókkal, melyeket a monitorok tartalmaznak és jelek segítségével.
A jelek a processz modulokból vannak exportálva. Ezeket szinkronizálásra használják, ezek nem tartalmaznak adatot. Egy processz küldhet jeleket (send) vagy pedig várhat jeleket (wait) egy másik processztől. A jelek abban különböznek a szemaforoktól, hogy egy olyan jel, amelyet egy processz sem vár, nulla műveletnek számít.
A korutinok a kvázipárhuzamosság szimulálására alkalmasak. A korutinok párhuzamosan futó folyamatok, ezek egymást aktiválják. Amikor az egyik folyamat újból aktív lesz, akkor onnan folytatódik ahol az előző vezérléscserénél megszakadt.

- Modula-3

A Modula-3 -ban a folyamatok szálak, amelyek közös memóriaterületen kommunikálnak.

A párhuzamosság eszközei a Thread modulban találhatók (Fork, Join).

A kölcsönös kizárást a LOCK parancs valósítja meg.

Üzenetküldéssel, osztott memóriával rendelkező nyelvek

- ***Ezeknél a nyelveknél a következőket vizsgáljuk*:**
  - **Hogyan határozza meg a nyelv a processzeket, a szinkronizálást?**
  - **Milyen az üzenetek szerkezete és küldése?**
  - **Hogyan kezeli az esetleges kommunikációs hibákat?**

- Simula-67

A Simula-67 nyelvben a korutin valamilyen osztályba tartozó objektum, amely azonos vagy más osztályba tartozó objektumokkal működik együtt. A létrehozott objektum (korutin) alárendeltségi viszonyban van azzal az objektummal, ami létrehozta.

A létrehozott objektum a detach utasítással adhatja vissza a vezérlést a létrehozó objektumnak.

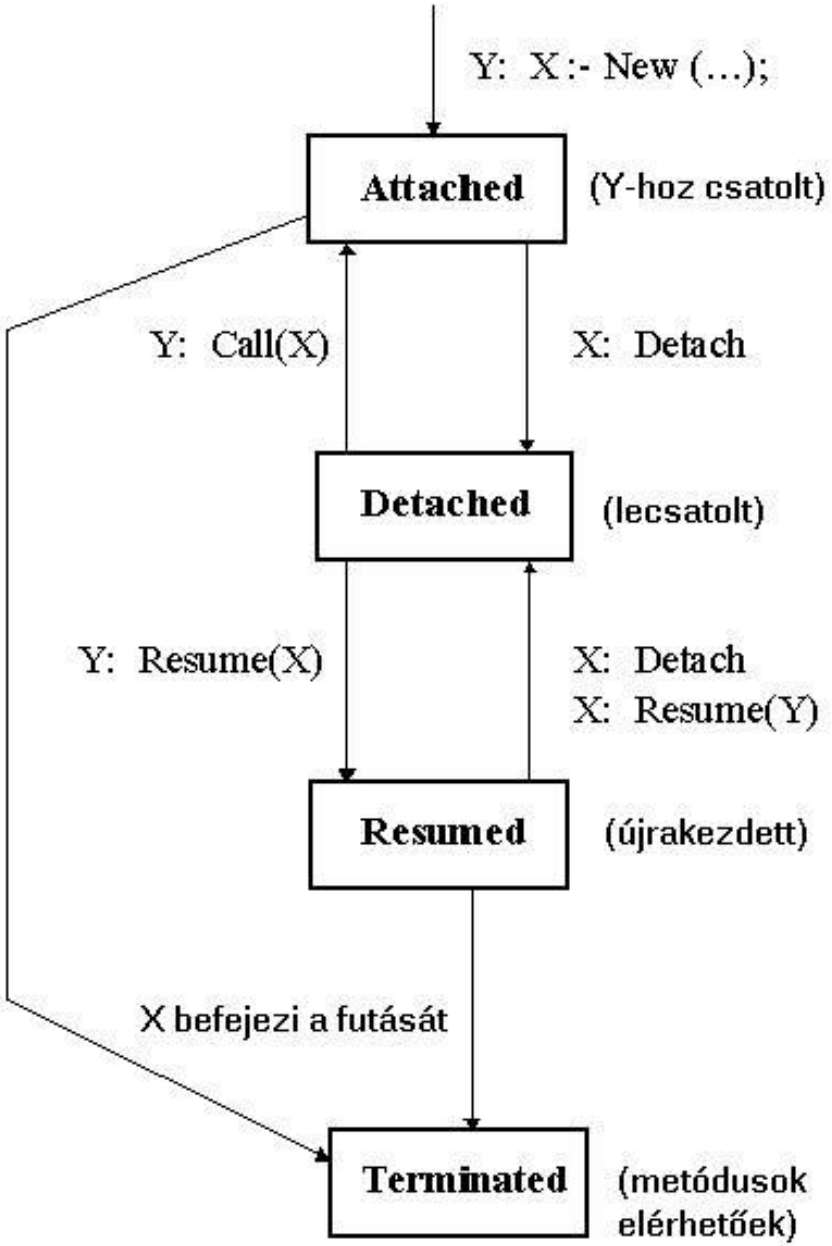
A létrehozó objektum a call(X) utasítással aktivizálhatja újra a leválasztott objektumot.

Az egyik korutinból (objektumból) a vezérlést a resume(X) utasítás segítségével adhatjuk át az X korutinnak (objektumnak).

A vezérlés visszakapásakor az objektumok végrehajtása ott folytatódik, ahol a legutóbbi vezérlésátadáskor megszakadt.

Egy korutin objektum élettartalma a következő:

- **A generálódásakor az objektum elkezdi végrehajtani a törzsében található utasításokat. Az objektum ilyenkor működő állapotban van, és a generátorhoz (az objektumhoz, ami létrehozta) hozzárendeltnek nevezzük.**
- **A korutin egy detach utasítással lemondhat a futásról a generátor javára, ilyenkor az objektumot leválasztottnak, de még nem lezártnak nevezzük.**
- **Az objektum egy resume(X) utasítással egy másik korutin javára mond le a vezérlésről.**
- **Az objektum újra vezérlést kaphat egy rá vonatkozó call(X) (a generátortól) vagy egy resume(X) (egy "testvér-folyamattól") utasítás hatására.**
- **Az objektum futása befejeződik, ha a törzsének végrehajtása elérte a befejező end kulcsszót. Ilyenkor az objektum futását sem call sem resume utasítással nem lehet felújítani. Az objektum azonban továbbra is létezik, tagváltozói elérhetőek, műveleteit meg lehet hívni.**


Y: X :- New (…);
Attached
(Y-hoz csatolt)
Y: Call(X)
X: Detach
Detached
(lecsatolt)
Y: Resume(X)
X: Detach
X: Resume(Y)
Resumed
(újrakezdett)
X befejezi a futását
Terminated
(metódusok elérhetőek)

Lua - lehetővé teszi több végrehajtási szálon működő programok írását.

A Lua *thread* típusa nem keverendő össze az operációs rendszerek szál fogalmával. Egy Lua-beli *thread* akkor is párhuzamos működést szimulál, ha az adott rendszer nem támogatja a párhuzamos működést.

Korutinok

Luában úgynevezett korutinok segítségével érhetünk el több végrehajtási szálon való működést. A korutinokon végezhető műveletek az (újra)meghívás, illetve a felfüggesztés, amikor a vezérlés visszaadódik a korutin meghívójának, így könnyebb követni egy program futását.

Három alapvető művelet: *create*, *resume*, *yield* - egy globális táblában vannak elhelyezve (*table coroutine*).

A *coroutine.create* utasítás létrehoz egy új korutint és lefoglal számára egy saját stacket a futtatásához. Paraméterként egy függvényt kap, ami a korutin törzsét tartalmazza, és egy korutinra mutató referenciát ad vissza - még nem indítja el! A korutin kezdetben *suspended*, azaz felfüggesztett állapotba kerül és a *continuation pointja* (ahonnan folytatni kell) a törzsének első utasítására áll. Gyakran a *coroutine.create* paramétere egy név nélküli (lambda) függvény:
co = coroutine.create(function() ... end)

Lua korutinokat értékül adhatunk változóknak, átadhatjuk őket paraméterként és vissza lehet adni őket eredményként. Explicit művelet nincsen korutinok törlésére - szemétgyűjtés van.

A *coroutine.resume* függvény aktiválja a korutint és első kötelező paramétereként egy referenciát kap az aktiválandó korutinra. Ekkor a korutin futni kezd az eltárolt *continuation pointnál*, amíg fel nem függesztődik újra vagy nem terminál. Akármelyik következik be, a vezérlés a korutin meghívási pontjához kerül vissza.

Korutin felfüggesztésére a *coroutine.yield* szolgál. Meghívásakor a korutin futási állapota elmentődik. A legközelebb, amikor a korutin aktiválódik, az elmentett állapotból folytatódik a futása.

Egy korutin terminál, ha a main függvénye *return* utasításhoz érkezik. Ebben az esetben halott állapotba kerül és nem aktiválható újra. Ugyancsak terminál egy korutin, ha hiba lép fel a futása során. Normális teminálás esetén a *coroutine.resume true-t* ad vissza, illetve minden értéket amit a korutin *main* függvénye visszaad. Hiba esetén *false* és egy hibaüzenetet.

A coroutine.wrap is egy új korutint hoz létre, azonban egy korutin referencia helyett, egy függvényt ad vissza, ami folytatja a korutint. Ennek a függvénynek átadott minden paraméter továbbadódik extra paraméterként a *resume-nak*. Ugyanígy a függvény visszaad minden értéket, amit a resume visszaad. (Nem kapja el a hibákat.)

Korutin és a hívója közti adatcsere pl.:

```
co = coroutine.wrap(function(a)
                local c = coroutine.yield(a+2)
                return c * 2
               end)
```

Amikor a korutin aktiválódik, a meghívásnál átadott minden extra argumentum átadódik a korutin main függvényének.

Pl, ha így hívjuk meg a korutint: b = co (20)   "a" értéke 20 lesz.

Amikor felfüggesztődik, minden, a *yieldnek* átadott argumentum a hívónak adódik. Így a korutin eredménye 22 (a+2).

Amikor újra aktiválódik, minden extra argumentum átadódik a *yieldnek*.

Pl. d = co(b+1) hatására a lokális c változó értéke 23 lesz (b+1)

Végül, ha egy korutin terminál, minden érték amit a *main* függvénye visszaad, az utolsó meghívási ponthoz kerül. Ez az eredmény 46 (c*2) adódik d-nek .

- Ada

Egy Ada programban egy folyamatot egy task-objektum reprezentál.

A taskok rendelkezhetnek belépési (entry) pontokkal. Ezek hívhatóak egy másik taszkból.

A taszknak van egy törzse, ez írja le azt a tevékenységet, amelyet a folyamat végez. A törzsben minden entry-hez tartozik egy accept utasítás, amikor itt tart a törzs végrehajtása, akkor hajlandó elfogadni egy másik taszk hívását erre a randevúra. A kommunikáció szinkron, a hívó folyamat a randevú elfogadásáig és az accept utasítás végéig felfüggesztődik. Az entrynek lehetnek paraméterei (in, out vagy in out típusúak is). Ha egy task meghívja egy másik taszk entry-pontját, akkor futása felfüggesztődik a randevú létrejöttéig és lekezeléséig.

- Ada

Időhöz kötött entry-hívást is alkalmazhatunk, ekkor megadhatjuk azt, hogy a taszk maximum mennyi időt várjon a randevú elfogadására, ha a hívott taszk ennyi idő alatt nem válik fogadókésszé, akkor a hívó taszk végrehajtása folytatódik.

Szelektív várakoztatás lehetősége a fogadó részéről - őrfeltételeket is megadhatunk.

Protected objektumok a kölcsönös kizárás biztosítására.

Korlátos buffer a példa a protected-re:

```
protected type Korlátos_Buffer ( Max: Positive ) is
   entry Betesz( X: in Elem );
   entry Kivesz( X: out Elem );
private
   Adatok: Vektor(1..Max);
   Be, Ki: Positive := 1;
end Korlátos_Buffer;
```

```
protected body Korlátos_Buffer is
   entry Betesz( X: in Elem ) when Be-Ki < Max is
   begin
     Adatok(Be) := X;  Be := Be + 1;
   end Betesz;

   entry Kivesz( X: out Elem ) when Be-Ki > 0 is
   begin
     X := Adatok(Ki);  Ki := Ki + 1;
     if Ki > Max then
        Ki := Ki - Max; Be := Be - Max;
     end if;
   end Kivesz;
end Korlátos_Buffer;
```

- Algol-68

Az Algol-68-ban a vesszővel elválasztott és begin - end közé írt utasítások párhuzamosan futtathatók. A begin-end addig nem fejeződik be, amíg minden vessző közötti egység - párhuzamos klóz- véget nem ér.

A szinkronizálásra Dijkstra szemaforokat vezettek be (sema). A szemaforok párhuzamos klózai elé ki kell tenni a par kulcsszót.

```
begin
    sema mutex := level 1;
    bool not finished := true;
        # közösen használt puffer deklarációja #
    proc producer = void:
        while not finished
        do
        down mutex
        # pufferbe egy elemet #
        up mutex
        od;
    proc consumer = void:
        while not finished
        do
        down mutex
        # pufferből egy elemet #
        up mutex
        od;
    par (producer, consumer) ;
end;
```

- Delphi

A TThread osztályból képzett alosztályok párhuzamosan végrehajthatók.

A szinkronizálásra és a kommunikálásra a főosztályban létre kell hozni egy eljárást, amely a vezérlésért felelős, majd a synchronize(eljárás_név) segítségével a párhuzamos alosztályok kommunikálását ellenőrizhetjük.

Delphi

- A TThread osztály legfontosabb (virtuális) absztrakt művelete az Execute() metódus, melyet minden leszármazottban felül kell definiálni. Ez tartalmazza a szál fő kódját.
- Szálat a Create konstruktor meghívásával hozhatunk létre, melynek egy Boolean paramétere van (CreateSuspended), mellyel azt adjuk meg, hogy a szál felfüggesztődjön (true), vagy a létrehozás után azonnal elinduljon (false).

Delphi

type TMyThread = **class**(TThread) **protected**

procedure Execute; **override**;

end;

- meg kell adni az Execute törzsét …

használata:

thread:=TMyThread.Create(false);

// el is indítjuk egyben

Delphi

Szemafor:

```
var MySemaphore: TSemaphore;
// szemafor változó deklarálása
begin
MySemaphore := TSemaphore.Create(nil, 5, 5, '');
    // 5 szál számára engedélyezzük az egyidejű használatot ...
MySemaphore.Acquire;
    //várakozás egy szemaforra (lefoglalás)
    ... // itt dolgozhatunk az osztott erőforráson
MySemaphore.Release; // szemafor elengedése
...
MySemaphore.Free; // szemafor megsemmisítése
end;
```

Delphi

A szinkronizációs esemény egy olyan objektum, amelyet egy szál kiválthat, így jelezve egy másik szálnak, hogy folytathatja a tevékenységét.

```
type TMyThread1 = class(TThread)
  protected procedure Execute; override;
end;
      TMyThread2 = class(TThread)
  protected procedure Execute; override;
end;
```

Delphi

```
 var MyEvent: TEvent;  // ez lesz az esemény
       Thread1: TMyThread1; Thread2: TMyThread2;
procedure TMyThread1.Execute;
   begin
   ...
     MyEvent.SetEvent; //kiváltjuk az eseményt
     ...
  end;
procedure TMyThread2.Execute;
   begin
   ...
      MyEvent.WaitFor(INFINITE); //várunk az eseményre
   ...
end;
```

- Java

A Java nyelvben a folyamatok (szálak) leírására a *Thread* osztályt használják, az osztály egy objektuma reprezentál egy folyamatot.

A folyamat törzse a run metódusban leírt kód.

Egy programban tetszőleges számú szálat indíthatunk. Ezeknek megadhatjuk a prioritását, felfüggeszthetjük, újraindíthatjuk, megállíthatjuk.

A szálakat szinkronizálhatjuk a join segítségével. A kommunikáció osztott változókon keresztül történik, a kölcsönös kizárás biztosítására a synchronized kulcsszó szolgál. Ha egy blokk vagy metódus synchronized akkor a hozzá csatolt monitor biztosítja a kölcsönös kizárást.

***Runnable* interfész megvalósítása kell, ha nem tud örökölni a Threadtől.**

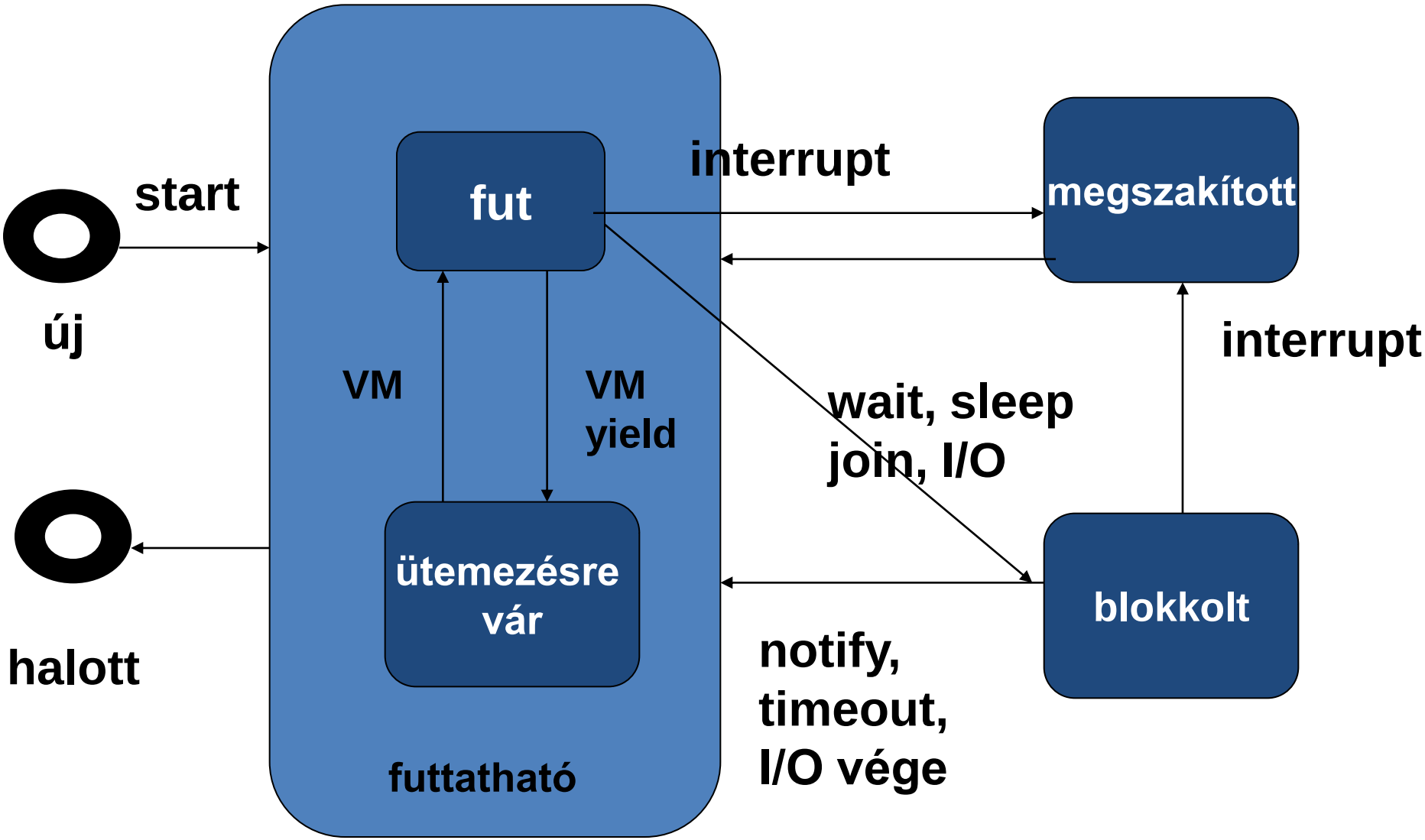
```
class MyThread extends Thread {
 public void run() {
  System.out.println("Helló, ez itt a" + getName() + " Thread" );
    }
 }
 public class ExtendedThread {
  static public void main(String args[]) {
          MyThread a, b;
          a = new MyThread();
          b = new MyThread();
          a.start();
          b.start(); }
}
```

```
class MyThread implements Runnable {
  public void run() {
       System.out.println("Helló, ez itt a " +
          Thread.currentThread().getName() + "    Thread" );
  }
 }
public class RunnableThread {
static public void main(String s[]) {
   MyThread work2do;
   Thread a, b;
   work2do = new MyThread();
   a = new Thread(work2do);
   b = new Thread(work2do);
   a.start();
   b.start();
   }
 }
```

Szálak állapotai:

- Futó: a VM éppen ezt a szálat futtatja
- Kész (ready): futtatható, éppen áll, hogy egy másik szál futhasson
- Blokkolt: vár egy külső esemény bekövetkezésére (pl. I/O)
- Megszakított: a folyamatoknak jeleket küldhetünk, melyekre megállnak
- Halott: befejeződött a folyamat

Szálak állapotai



```
// Runnable interfész alkalmazása Applet-ben:
import java.awt.*;
public class Logo extends java.applet.Applet implements Runnable {
  Image img;           // kép handler
  Thread thd = null;       // a szál
  int i = 0;                    // sor mutató
  int imgWidth;         // kép szélesség
  int imgHeight;        // kép magasság
  static String imgName;    // kép file neve

  public void run() {      // a Runnable run() metódusát megvalósítjuk
    Thread en = Thread.currentThread();
    if ((imgName = getParameter("imagefile")) == null) System.exit(-1);
    img = getImage(getCodeBase(), imgName); repaint();
    try {Thread.sleep(500);} catch (InterruptedException e){}
    imgWidth = img.getWidth(this); imgHeight = img.getHeight(this);
    if (img != null) {
      while (thd == en) {
        for (i=0; i<imgHeight; i+=4 ) {
          repaint(); try {Thread.sleep(100);} catch (InterruptedException e){}
        }
      }
    }
  } ….
```

Lefordítjuk, majd egy külön html fájlba írjuk:

```
<html>
<title>Logo Applet külön szálban</title>
<body>
<hr>
 <applet code="Logo.class" height=200 width=300>
 <param name=imagefile value=javaLogo.gif>
 </applet>
</body>
</html>
```

- Szálcsoportok is létrehozhatók – a ThreadGroup segítségével – csoportszinten is kezelhetők a szálak
- Futásvezérlés:
  - Felfüggesztés – újraindítás

    volt: suspend() - resume() pár, ezt ma már nem ajánlják – holtpontveszély (pl. egy szál zárol egy erőforrást, és felfüggesztik …)

    Javaslat:
    segédváltozó és a wait() – notify() pár

– Megszakítás – interrupt()

Appleteknél a stop() metódusban kell a felfüggesztést is elengedni, ill. a megszakítást meghívni.

– Leállítás

volt: stop() – nem biztonságos

most: volatile Thread referencia, ami a futó szálra mutat, ha a szálat leállítják, ez null lesz, amit a run() vizsgál

– Szálak összekapcsolása – join() – szinkronizálásra – megvárja, amíg a másik befejeződik (vagy egy adott ideig vár)

– Prioritások kezelése – 10 szint, állítható

- Versengés
  több szál szimultán szeretne ugyanahhoz az erőforráshoz hozzájutni:

  synchronized blokk
    – objektum- és osztályszinten

  problémák: író/olvasó esetben
    – egyszerre csak egy írhat és olvashat, bár elvben több is olvashatna

A synchronized kulcsszó

- Metódusok elé írhatjuk (de pl. interfészekben nem!)
- Kölcsönös kizárás arra a metódusra
- Kulcs (lock) + várakozási sor
  - A kulcs azé az objektumé, amelyiké a metódus
  - Ugyanaz a kulcs az összes szinkronizált metódusához

1 Mielőtt egy szál beléphetne egy szinkronizált metódusba, meg kell szereznie a kulcsot

2 Vár rá a várakozási sorban

3 Kilépéskor visszaadja a kulcsot

Szinkronizált blokkok

- A synchronized kulcsszó védhet blokk utasítást is
- Ilyenkor meg kell adni, hogy melyik objektum kulcsán szinkronizáljon

```
synchronized(obj){...}
```

- Sokszor úgy használjuk, hogy a monitor szemléletet megtörjük
- Nem az adat műveleteire biztosítjuk a kölcsönös kizárást, hanem az adathoz hozzáférni igyekvő kódba tesszük
- A kritikus szakasz utasításhoz hasonlít
- Létjogosultság: ha nem egy objektumban vannak azok az adatok, amelyekhez szerializált hozzáférést akarunk garantálni

wait - notify

- Szignálokra hasonlít
- Egy feltétel teljesüléséig blokkolhatja magát egy szál
- A feltétel (potenciális) bekövetkezését jelezheti egy másik szál
- Alapfeladat: termelő - fogyasztó
  (korlátos) bufferen keresztül kommunikálnak
  - egy termelő, egy fogyasztó
  - több termelő, több fogyasztó

Működés

- Minden objektumhoz tartozik a sima kulcshoz tartozó várakozási soron kívül egy másik, az ún. *wait-várakozási sor*
- A wait() hatására a szál bekerül ebbe
- A notify() hatására az egyik várakozó kikerül belőle
- A wait() és notify() hívások csak olyan kódrészben szerepelhetnek, amelyek ugyanazon az objektumon szinkronizáltak

```
synchronized(obj){ ... obj.wait(); ... }
```

- A szál megszerzi az objektum kulcsát, ehhez, ha kell, sorban áll egy ideig (synchronized)
- A wait() hatására elengedi a kulcsot, és bekerül a wait-várakozási sorba
- Egy másik szál megkaparinthatja a kulcsot (kezdődik a synchronized)
- A notify() vagy notifyAll() metódussal felébreszthet egy vagy az összes wait-es alvót, aki bekerül a kulcsos várakozási sorba
- A synchronized végén elengedi a kulcsot
- A felébredt alvónak (is) lehetősége van megszerezni a kulcsot és továbbmenni
- A synchronized blokkja végén ő is elengedi a kulcsot

C#

- A párhuzamos végrehajtás során az alábbi lépéseket kell megtenni :
  - Definiálunk egy függvényt, aminek nincsenek paraméterei és a visszatérési típusa *void*. Ez több rendszerben kötelezően run névre hallgat – de nem feltétlenül!
  - Ennek a függvénynek a segítségével egy függvénytípust, delegáltat definiálunk. Könyvtári szolgáltatásként a ThreadStart ilyen, használhatjuk ezt is.
  - Az így kapott delegáltat felhasználva készítünk egy *Thread* objektumot.
  - Meghívjuk az így kapott objektum *Start* függvényét.
- A párhuzamos végrehajtás támogatását biztosító könyvtári osztályok a *System.Threading* névtérben találhatók.

C#

```
using System;
using System.Threading;
class program {
        public static void szal() {
                Console.WriteLine("Ez a szálprogram!");
              }
public static void Main() {
     Console.WriteLine("A főprogram itt indul!");
     ThreadStart szalmutato=new ThreadStart(program.szal);
      // létrehoztuk a  függvénymutatót, ami a szal() függvényre mutat
     Thread fonal=new Thread(szalmutato);
      // létrehoztuk a párhuzamos végrehajtást végző objektumot
      // paraméterül kapta azt a delegáltat (függvényt) amit  majd végre kell
      // hajtani
     fonal.Start();
      // elindítottuk a fonalat, valójában a szal() függvényt
      // a fonal végrehajtása akkor fejeződik be amikor  a szal függvényhívás
      // befejeződik
     Console.WriteLine("Program vége!");
     }
}
```

C#

- **Szálak vezérlése**
  - Sleep (millisec): statikus függvény, az aktuális szál végrehajtása várakozik a paraméterben megadott ideig.
  - Join(): az aktuális szál befejezését várjuk meg
  - Interrupt(): az aktuális szál megszakítása. A szál objektum interrupt hívása ThreadInterruptedException eseményt okoz a szál végrehajtásában.
  - Abort(): az aktuális szál befejezése, valójában a szálban egy AbortThreadException kivétel dobását okozza, és ezzel befejeződik a szál végrehajtása. Ha egy szálat abortáltunk, nem tudjuk a Start függvénnyel újraindítani, a start utasítás ThreadStateException kivételt dob. Ezt a kivételt akár mi is elkaphatjuk, és ekkor, ha úgy látjuk, hogy a szálat mégsem kell „abortálni”, akkor a Thread.ResetAbort() függvényhívással hatályon kívül helyezhetjük az Abort() hívását.
  - IsAlive: tulajdonság, megmondja, hogy a szál élő-e
  - Suspend(): a szál végrehajtásának felfüggesztése
  - Resume(): a felfüggesztés befejezése, a szál továbbindul

C#

```
class program {
    public static void szal() {
        Console.WriteLine("Ez a szálprogram!");
        Thread.Sleep(1000);
        // egy másodpercet várunk
        Console.WriteLine("Letelt az 1 másodperc a
                          szálban!");
        Thread.Sleep(5000);
        Console.WriteLine("Letelt az 5 másodperc a
                          szálban!");
        Console.WriteLine("Szál vég!");
    }
```

C#

```
public static void Main() {
    Console.WriteLine("A főprogram itt indul!");
    ThreadStart szalmutato=new ThreadStart(program.szal);
    // létrehoztuk a függvénymutatót, ami a  szal() függvényre mutat
    Thread fonal=new Thread(szalmutato);
    // létrehoztuk a párhuzamos végrehajtást végző objektumot
    // paraméterül kapta azt a delegáltat (függvényt), amit majd végre kell hajtani
    fonal.Start(); // elindítottuk a fonalat, valójában a szal() függvényt
    Thread.Sleep(2000);  // várunk 2 másodpercet
    Console.WriteLine("Letelt a 2 másodperc a főprogramban, a szálat
                        felfüggesztjük!");
    fonal.Suspend(); // fonal megáll – veszélyes!!
    Thread.Sleep(2000);
    Console.WriteLine("Letelt az újabb 2 másodperc a főprogramban, a szál végrehajtását
    folytatjuk!");
    fonal.Resume(); // fonal újraindul
    fonal.Join(); // megvárjuk a fonalbefejeződést
    Console.WriteLine("Program vége!");
    }
}
```

A Suspend és a Resume a .NET 2.0 óta a holtpontveszély miatt nem javasolt!!

C# - .NET

```
class adatok
{
    public void mentes(string s)
    {
        Console.WriteLine("Adatmentés elindul!");
        for(int i=0;i<50;i++)
        {
          Thread.Sleep(1);
          Console.Write(s);
        }
        Console.WriteLine("");
        Console.WriteLine("Adatmentés befejeződött!");
    }
}
```

C# - .NET

```
class program {
    public static adatok a=new adatok();
    // adatmentésmező a programban
    // szál1 definiálás
    public static void szal1() {
            Console.WriteLine("Ez a szál1 program indulás!");
            // egy másodpercet várunk
            Console.WriteLine("Adatmentés meghívása szál1-ből!");
            a.mentes("+");
            Console.WriteLine("Szál1 vége!");
    }
    // szál2 definiálás
    public static void szal2() {
            Console.WriteLine("Ez a szál2 programindulás!");
            // egy másodpercet várunk
            Console.WriteLine("Adatmentés meghívása szal2-ből!");
            a.mentes("-");
            Console.WriteLine("Szál2 vége!");
    }
```

C# - .NET

```
public static void Main() {
    Console.WriteLine("A főprogram itt indul!");
    ThreadStart szalmutato1=new      ThreadStart(program.szal1);
    ThreadStart szalmutato2=new      ThreadStart(program.szal2);
    // létrehoztuk a függvénymutatókat,
    Thread fonal1=new Thread(szalmutato1);
    Thread fonal2=new Thread(szalmutato2);
    fonal1.Start();
    fonal2.Start();
    Console.WriteLine("Program vége!");
    }
}
```

Felváltva ír ki +-t és - -t a képernyőre!!

C# - .NET

- **Erőforráskezelés**
  A .NET keretrendszer számos osztályt és adattípust kínál számunkra, hogy a közös erőforrásokat kezelhessük. A CLR (Common Language Runtime) háromféle elérési módot biztosít globális változók, metódusok, osztályszintű metódusok, objektumok és blokkok számára:
  - Szinkronizált kódterület - Szinkronizálható bármely osztályszintű és példányszintű metódus, vagy annak egy része *Monitor* használatával. Megjegyzendő, hogy nincs lehetőség statikus adattagok szinkronizációjára.
  - Klasszikus kézi szinkronizáció - Számos szinkronizációs osztály használható arra, hogy összhangot teremtsünk a saját elvárásainknak megfelelően.
  - Szinkronizált környezet A *SynchronizationAttribute* használata egyszerű, automatikus szinkronizációt valósít meg *ContextBoundObject* objektumokon. Minden objektum közös környezetben osztozik a záron.

C# - .NET

- A **Monitor osztály** használatával elérhetjük, hogy egy kódrészletet egy adott időpontban csak egy szál használhasson.
  A Monitor osztály minden metódusa statikus, így használatához nem kell példányosítani az osztályt.
  A monitor indítása a *Monitor.Enter(object)* vagy a *Monitor.TryEnter(object)* metódushívással történhet.
  Feloldása a *Monitor.Exit(object)* vagy a *Monitor.Wait(object)* hívással történhet.
  Ha a monitor feloldása megtörtént, akkor a *Monitor.Pulse* vagy a *Monitor.PulseAll* hívás üzenetet küld a hozzáférési sorban álló szálaknak, hogy szabad az elérés.
  Amikor egy szál a lefoglalt kódrészletében *Wait* -et hív, akkor megszakad a hozzáférése, és bekerül egy várakozó sorba, majd a sor első elemét reprezentáló szál meghívja a *Pulse* vagy *PulseAll* metódust és az említett szál lép egyet a hozzáférési sorban.
  Az *Enter* metódus atomi, így ha két szál egyszerre hívja ugyanarra a kódrészletre, akkor is csak az egyik kapja meg a jogot a futtatásra.
  Javasolt a monitort belső objektumokon használni, mivel külső objektum zára deadlock -hoz vezethet. Javasolt, hogy a kódot egy try blokkban helyezzük el, és az *Exit* hívást pedig a finally ágban.
  Ehelyett alkalmazható a *lock(object)* hívás, ami funkciójában megegyezik a Monitor -éval, de amikor a végrehajtás kilép a lock blokkjából, a zár feloldódik.

C# - .NET

- Szinkronizáció:
  - 1. A *Synchronization*() attribútummal kell jelölni az osztályt.
  - 2. Az osztályt a *ContextBoundObject* osztályból kell származtatni.

Példa:

```
[Syncronization()]
class szinkronosztály: ContextBoundObject {
    int i=5; // egy időben csak egy szál fér hozzá
    public void Novel() // ezt a függvényt is csak egy szál
    // tudja egyszerre végrehajtani
    {
        i++;
    }
...
}
```

C# - .NET

```
public void mentes(string s) {
    Monitor.Enter(this);
    Console.WriteLine("Adatmentés elindul!");
    for(int i=0;i<50;i++) {
        Thread.Sleep(1);
        Console.Write(s);
    }
    Console.WriteLine("");
    Console.WriteLine("Adatmentés befejeződött!");
    Monitor.Exit(this);
}
```

Az a blokk, amit a *Monitor* véd, megszakítás nélkül fut le – így az előző példában már nem lesz gond.

C# - .NET

- És még sok-sok osztály segít – pl.
  - **WaitHandle**
  - *Mutex*, *AutoResetEvent*, *ManualResetEvent*
  - **InterLocked**
  - **ThreadPool**
  - **Timer**
  - **Backgroundworker**
  - stb., stb.

Objektumorientált párhuzamosság

A szekvenciális objektumorientált nyelvekre igaz az alábbi három feltétel:

- **A program futása pontosan egy objektum aktivizálásával kezdődik.**
- **Amikor egy objektum üzenetet küld egy másik objektumnak, akkor az eredmény megérkezéséig nem csinál semmit, azaz megvárja az üzenet fogadását és feldolgozását.**
- **Egy objektum csak akkor aktív, amikor egy bejövő üzenet feldolgozására egy metódust futtat.**

A párhuzamosságot úgy lehet elérni, hogy a fenti feltételek valamelyikét elhagyjuk, azaz:

Több processz lehet aktív egyszerre.

Az üzenetküldő nem várja meg az üzenet fogadását és feldolgozását, hanem az üzenet elküldése után rögtön folytatja futását (aszinkron kommunikáció).

Az objektumok nem várnak passzívan az érkező üzenetekre, hanem minden objektumnak egy saját törzse van.

Az objektum létrehozásakor elkezdődik a törzs végrehajtása a többi objektum törzsével párhuzamosan. A törzs meghatározott pontjain megszakítható azért, hogy a beérkező üzenetekre válaszoljon, ez randevú formájában történik, azaz a küldő és a fogadó szinkronizálódik.

Go - gorutinok

- Osztott memória helyett csatornákon való kommunikáció
- goroutine:
  - Konkurens folyamat
  - Közös címterületen, saját veremmel, ami dinamikusan nő
  - Saját szemétgyűjtő
  - A tényleges szálak kezelését elrejti a programozó elől
  - A go kulcsszóval hívott függvény új goroutine-ban indul

Go - példa

```
func IsReady(what string, minutes int64) {
    time.Sleep(minutes * 60*1e9);
    fmt.Println(what, "is ready")
}
  go IsReady("tea", 6);
  go IsReady("coffee", 2);
  fmt.Println("I'm waiting....");
```

Eredmény

I’m waiting... (azonnal)

coffee is ready (2 perc múlva)

tea is ready (6 perc múlva)

Go - channel

Csatorna típus a párhuzamosság támogatására

Kommunikációs módszer két függvény szinkronizációjához

Lehet szinkron vagy aszinkron is

```
chan int // int-et küldhet, fogadhat
chan<- float64 // float64-et küldhet
<-chan int // int-et fogadhat
```

```
make(chan int) // szinkron csatorna  (nincs puffer)
make(chan int, 100) // aszinkron csatorna
close(c) // c bezárása
```