Programozási nyelvek és módszerek

PÁRHUZAMOSSÁG

Bemelegítés

Párhuzamosság

Puzzle analógia – Henry Neeman @ Oklahoma University

Soros programozás: egyvalaki szeretné megcsinálni

- 1000 darabos puzzle, kb. 1 óra

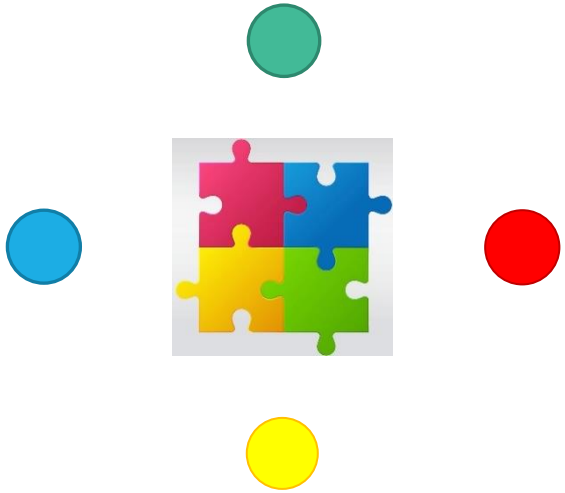
Tegyük fel, hogy egy barát leül és segít kirakni

- Ekkor kommunikálni is kell.
  - Néha ugyanazon a részen dolgoznak, és zavarják egymást.
- 35 perc alatt oldható meg
  - A várt 30 helyett.

Párhuzamosság

Minél többen annál jobb?

- Ha már négyen vagyunk, 20 perc kell (15 helyett), mert egyre többet kell koordinálni egymással



Csökkenő hozadék törvénye

- Ahogy többen próbálnak meg közreműködni, egyre többet kell kommunikálni és egymásra várni vagy egymást kikerülni.
- Ugyanazt a problémát egyre több párhuzamos ágenssel megoldani a csökkenő hozadékok törvényéhez vezet.

Elosztott párhuzamosság

Osszuk szét a puzzle darabokat két kupacba

- Mindenki dolgozik a saját felén teljesen függetlenül,
- A végén össze kell illeszteni a két felet

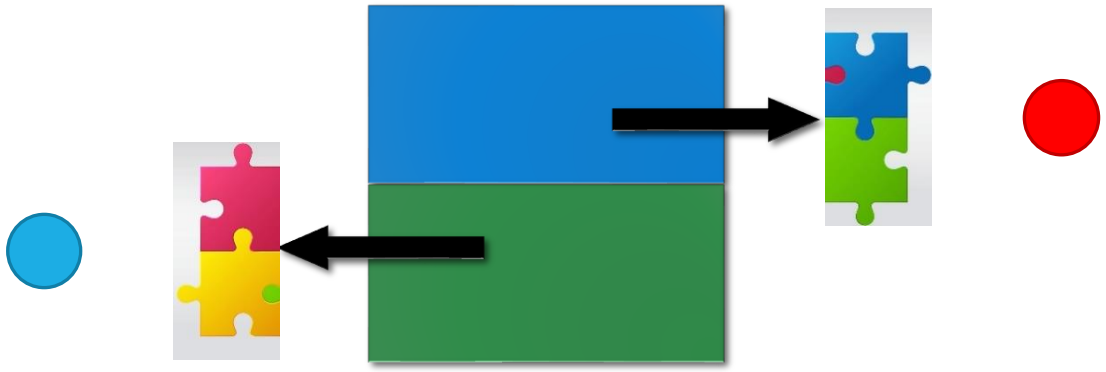
Szét is kell osztani a darabokat az elején

- Attól függően hogy mennyi töltünk egy jó elosztással, több vagy kevesebb kommunikációra van szükség a végén az összeillesztésekor
  - Tökéletes felosztás a végső kép két felére: sok munka az elején, a végén triviális.
  - Véletlen leosztás: triviális az elején, sok munka a végén

Elosztott párhuzamosság

A kezdeti elosztás határozza meg a két fél kiegyensúlyozottságát is.

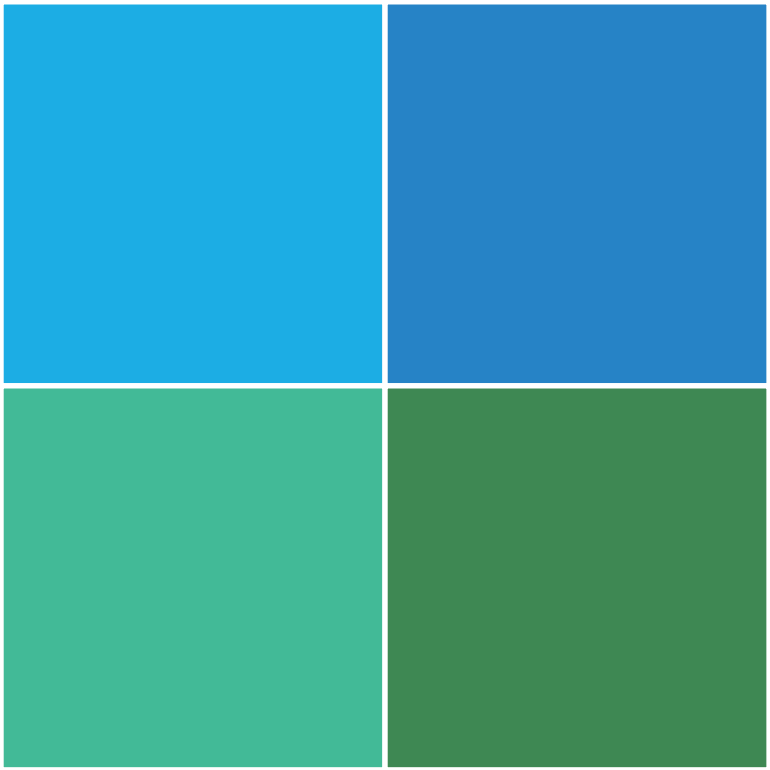
- Ha a kép fele ég a másik föld, könnyű a leosztás, a végén csak a perem mentén kell „kommunikálni”

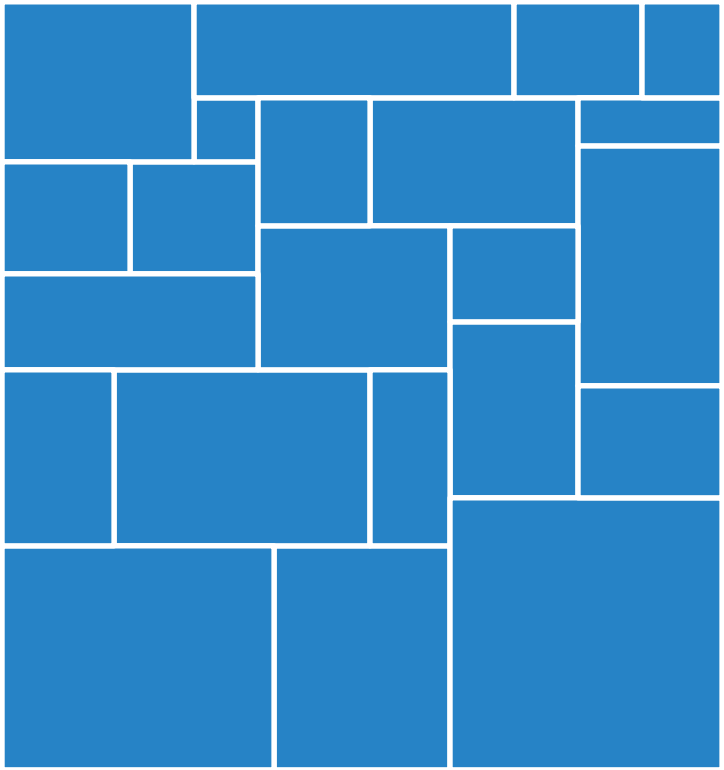


Bizonyos problémák esetén ez a felosztás könnyebb, mint más problémákon

Elosztott párhuzamosság

A munka kiegyensúlyozása nehéz, főleg ha az egyik résztvevő gyorsabb





Párhuzamosság

FOGALMAK

Alapelvek

A hagyományos számítógépeken

- Egy adott időben egy program futhat
- A számítógép architektúrája pedig egy egyprocesszoros gép
  - Csak egy program lehet aktív, egy adott pillanatban egy utasítás kerül végrehajtásra.
  - A különböző programok végrehajtása egymás után történik

Igény a valódi párhuzamosításra

- Hardver szintjén
- Szoftver szintjén

Párhuzamosság

- Van egy rendszer és egy időben több komponense működik
- Előnyei a szekvenciális programozással szemben
  - Gyorsabb programvégrehajtás
  - Hatékonyabb programvégrehajtás
    - Megfelelő hardver esetén

Természetesebb kifejezésmód lehetősége a nagyobb számú elemi művelet segítségével (szoftver szempont)

Összehangolás kérdései:

- Kommunikáció
- Szinkronizáció

Gondok

- Tesztelés
- Holtpont
- Kiéheztetés

Az elosztott (distributed) jelző – többféle jelentés

- Elosztott – a „párhuzamos” speciális esete, amikor a rendszer komponensei térben elkülönülten helyezkednek el
- Elosztott memória használata

Megosztott (shared) jelző

- Erőforrások együttes használata – tipikusan memória

A konkurrens (concurrent) szó gyakran azt hangsúlyozza, hogy a rendszer komponensei vetélkednek egymással az erőforrásokért

Párhuzamos hardverek

A párhuzamos hardver rendszerek sokféleképpen osztályozhatók

Egy ilyen a máig is használt Flynn-féle osztályozás

- Komponensei
  - A processzor
  - A memória és
  - A vezérlő (controller)

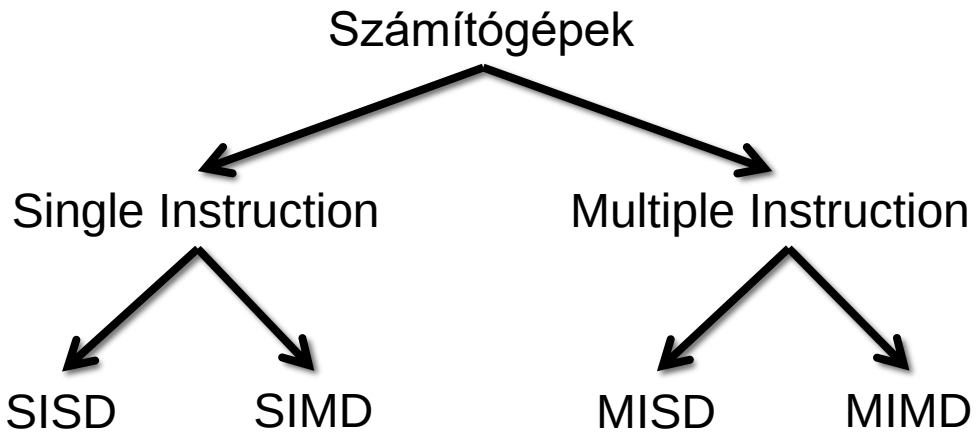
A processzor(ok) a vezérlőtől kapják a végrehajtandó utasítássorozatot (instruction stream)

- A memóriá(k)ból pedig a feldolgozandó adatokat (data stream)

Flynn-féle modell

Csoportok aszerint, hogy hány utasításfolyam (Instruction stream), illetve adatfolyam (Data stream) különíthető el a rendszerben

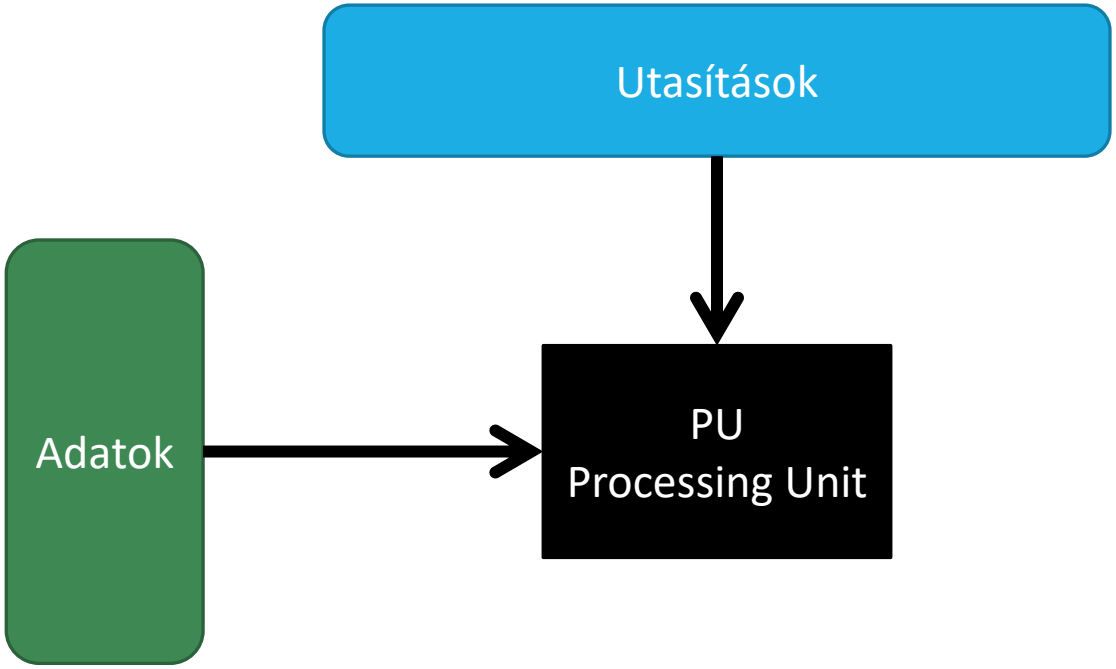
Mindkettő egyszeres (Single), vagy többszörös (Multiple) lehet



SISD (Single Instruction, Single Data)

„Klasszikus” számítógépek

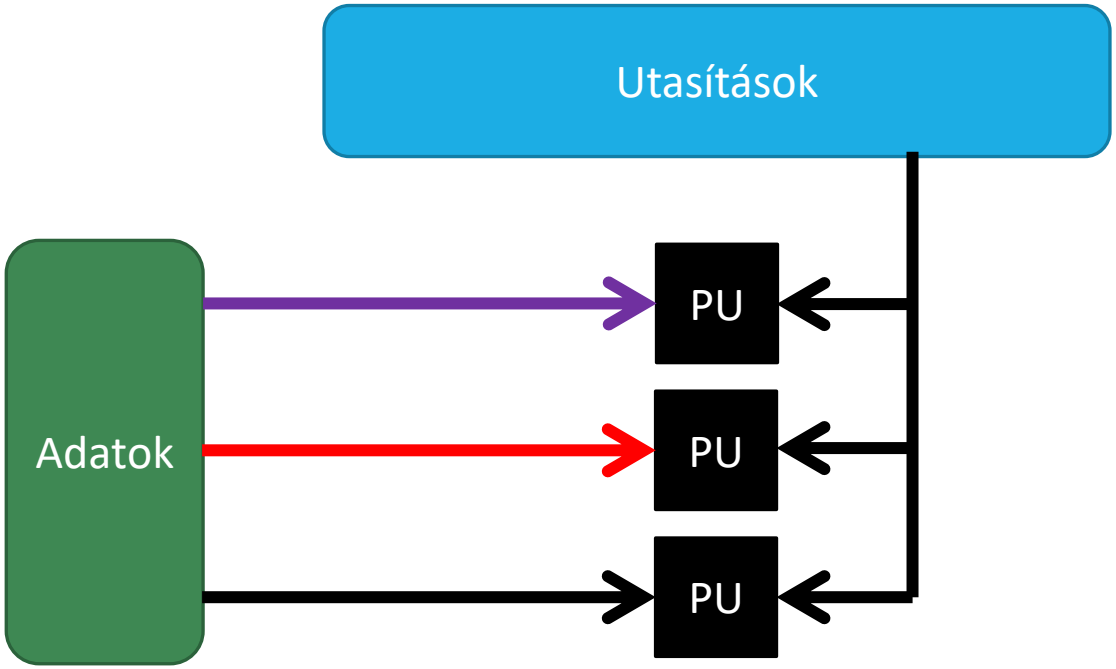
- Egy processzor, egy mag
- Napjainkban: számológép



SIMD (Single Instruction, Multiple Data)

Nagyon sok egyszerű processzorból álló gépek

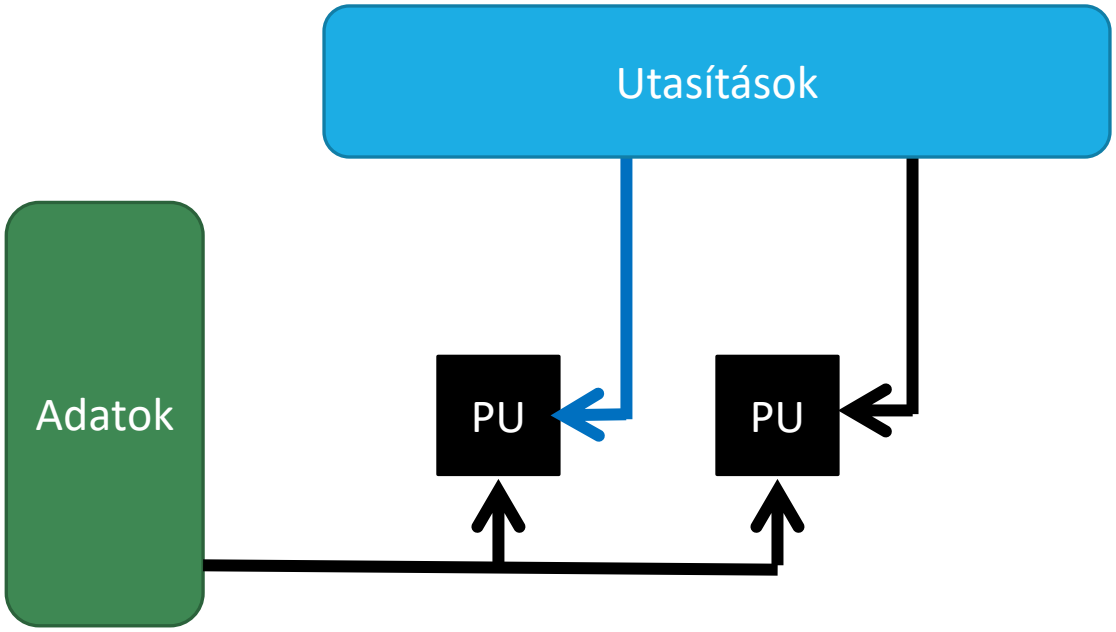
- Mindegyike rendelkezik saját memóriával
- Mindegyiken azonos program fut
- GPGPU



MISD (Multiple Instruction, Single Data)

Nem gyakori az ilyen gép

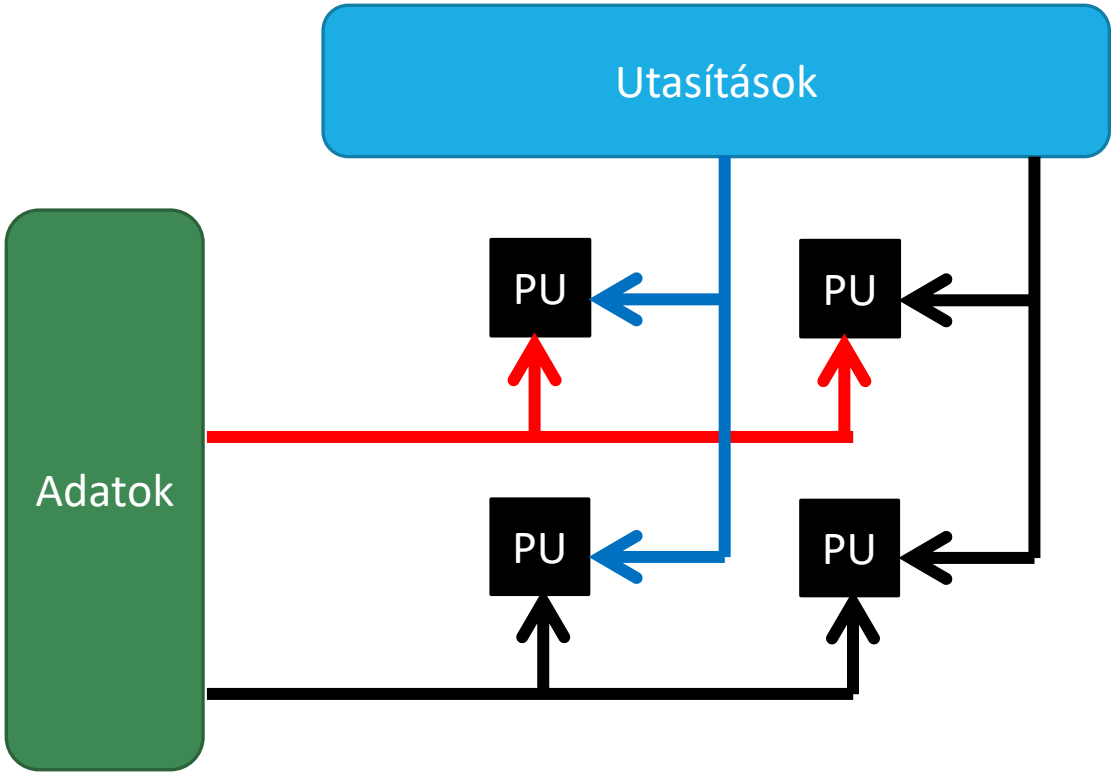
- Hibatűrő rendszerek esetén használatos
- Ugyanazt a feladatot több algoritmus oldja meg
  - Például háromból két azonos eredményt tekintünk jó megoldásnak



MIMD (Multiple Instruction, Multiple Data)

Számunkra most a legfontosabb csoport

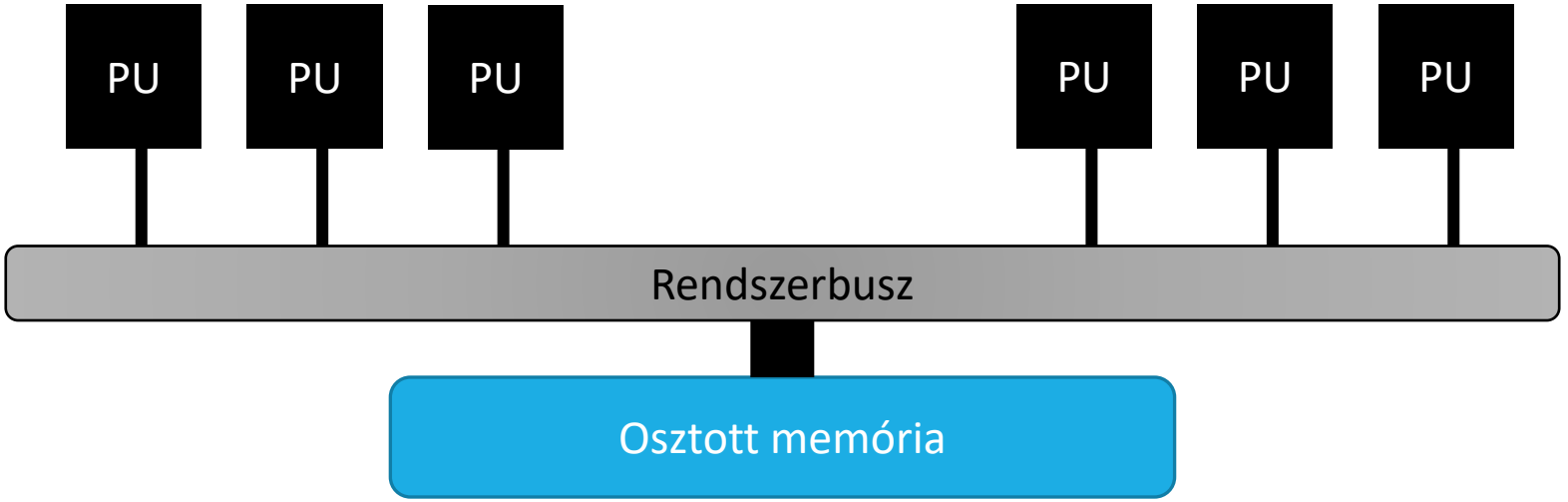
- Többféle elrendezésben
- Egy lehetséges elrendezés:



MIMD (Multiple Instruction, Multiple Data)

MIMD

- SMP (Symmetric multiprocessing) rendszerek
- SM-MIMD (shared-memory MIMD) rendszerek
  - a rendszerbuszra egyetlen fizikai memória csatlakozik, amelyet minden processzor lát. (Pl. többprocesszoros PC)



SM-MIMD architektúra

Előnye

- az osztott memórián keresztül nagyon egyszerű a programok közötti kommunikáció, illetve szinkronizáció megoldása,
- hagyományos programok nagyon könnyen átültethetőek ilyen környezetbe.

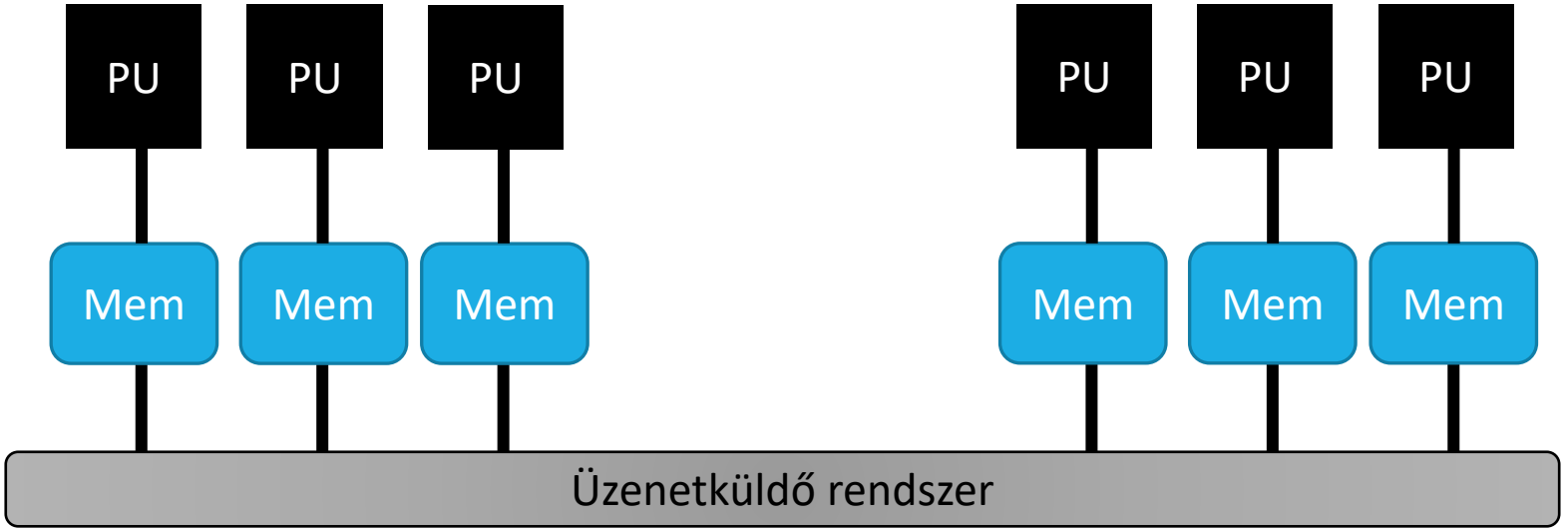
Hátránya

- a memória elérésének fizikai korlátai korlátokat szabnak a lehetséges processzorok számának.

MPP (Massively parallel processing)

Vagy DM-MIMD (distributed memory MIMD) architektúrák

- Önálló memóriával rendelkező processzorokból felépített nagy számítógépek – üzenetküldéssel kommunikálnak

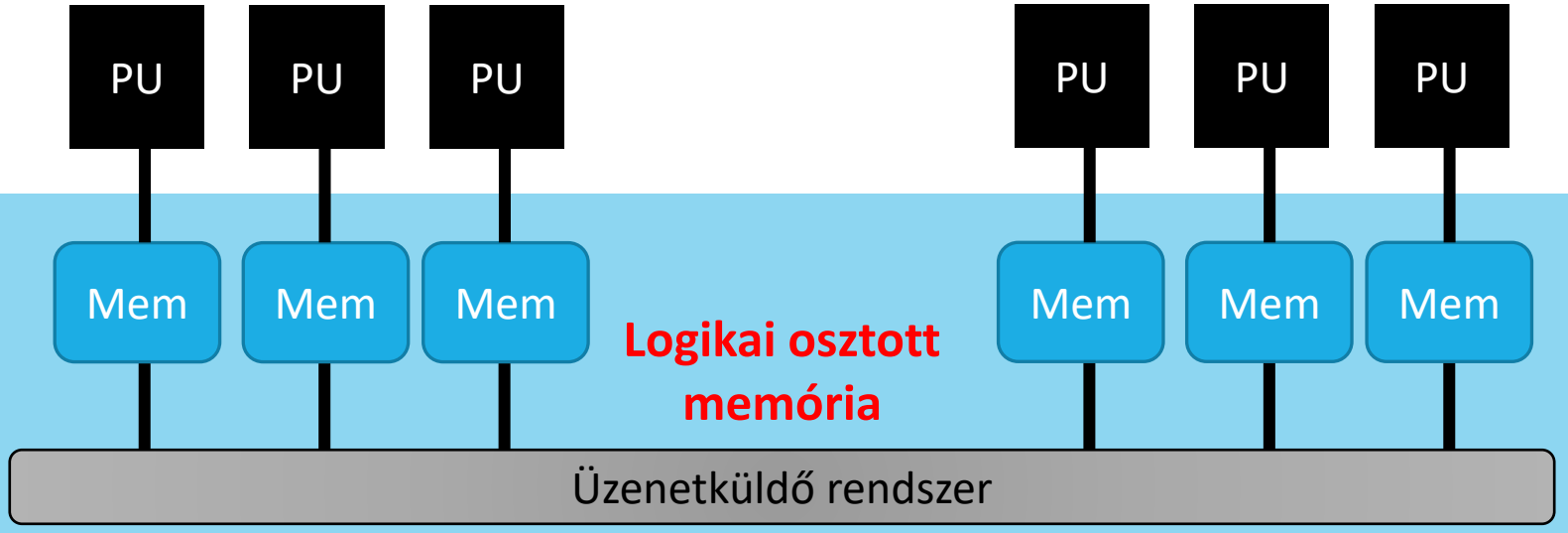


SPP (Scalable parallel processing)

Olyan DM-MIMD rendszer amelyben a processzorok közötti kommunikációs eszköz szimulálni tudja az egyetlen osztott memóriát.

A processzorok számára nem saját memóriájuk elérése csak időbeni különbséget jelent.

Ötvözi az SM-MIMD és DM-MIMD rendszerek előnyeit – sokkal bonyolultabb hardware megoldásokra van szükség.



A szoftveres párhuzamosság

Az első próbálkozás aszinkron működésre az input-output műveleteknek a központi egység műveleteivel egy időben való végzése volt.

- Ezeket már alapjában véve konkurens műveleteknek 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 (process)

A processz egy szekvenciálisan végrehajtott program szerinti műveletsor.

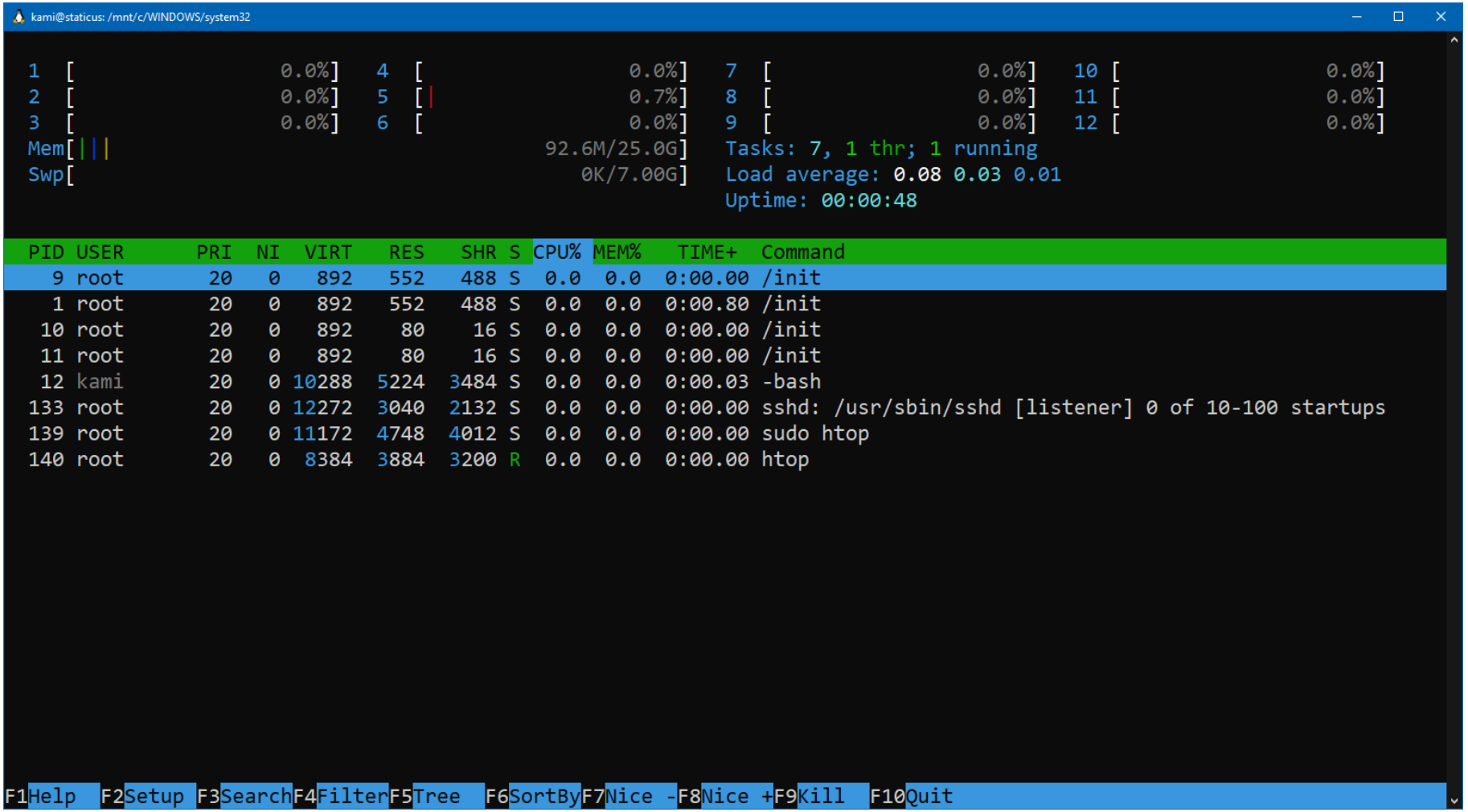
- Részei
  - A műveletsor (program)
  - Kimenő és bemenő adatok
  - Folyamat állapota
- Alapvetően egy folyamatnak öt állapota lehet
  - Futó: a CPU éppen ezt a folyamatot futtatja
  - Kész (ready): futtatható, éppen áll, hogy egy másik folyamat futhasson
  - Blokkolt: vár egy külső esemény bekövetkezésére (pl. I/O)
  - Megszakított: a folyamatoknak jeleket (signal) küldhetünk, melyekre megállnak
  - Halott: befejeződött a folyamat

Folyamatok (process)

Tulajdonságok

- A folyamat kernelszintű fogalom, ezért minden egyede speciális adatokkal rendelkezik:
  - UID (User IDentification number)
  - GID (Group ID)
  - PID (Process ID)
  - Prioritás
  - Virtuális memória
  - Fájlleíró
  - …
- A folyamatleíró struktúra minden része a kernelben van
  - A felhasználói program nem tudja elérni azt.
- Más folyamatok csak rendszerhívásokon keresztül férnek hozzá a folyamat adataihoz, állapotához.
- A folyamat eljárásai, függvényei viszont a felhasználói területen vannak, így a folyamaton belül elérhetőek.

Folyamatok (process)



Folyamatok állapotai

Halott

- Amint a folyamat utasításai elfogynak, akkor befejezi futását. Ez után már nem létezik, mint folyamat többé.

Futó / kész

- A két állapot nagyon hasonlít egymásra, mindkettő esetén a processz alkalmas a futásra
- A lényeges különbség, hogy a futó folyamat éppen használja a CPU-t, míg a kész számára a CPU nem elérhető.
  - Egyprocesszoros/egymagos környezetben a CPU több folyamat között van megosztva
  - Ütemező algoritmus szabályozza, hogy mikor kell megállítani egy folyamat feldolgozását, hogy egy másik folyamatot szolgálhasson ki.

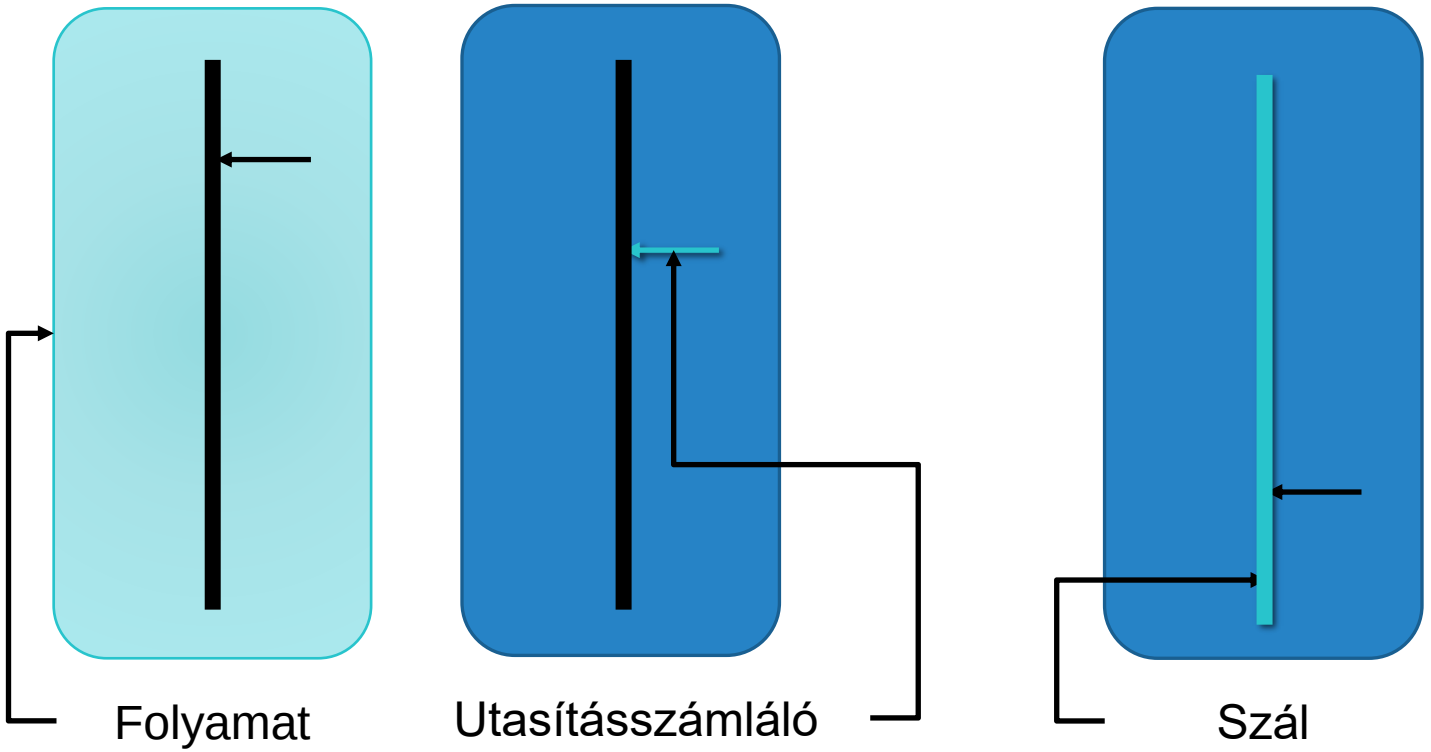
Folyamatok állapotai

Blokkolt

- A folyamat nem futhat tovább (nem tud)
  - Még akkor sem, ha a CPU-nak nincs más feladata.
- A folyamat vár egy esemény bekövetkeztére
  - Egy lemez blokk beolvasására, egy karakter leütésére.
- Amint bekövetkezik az esemény processz kész állapotba kerül
  - Ezzel ütemezhetővé válik.

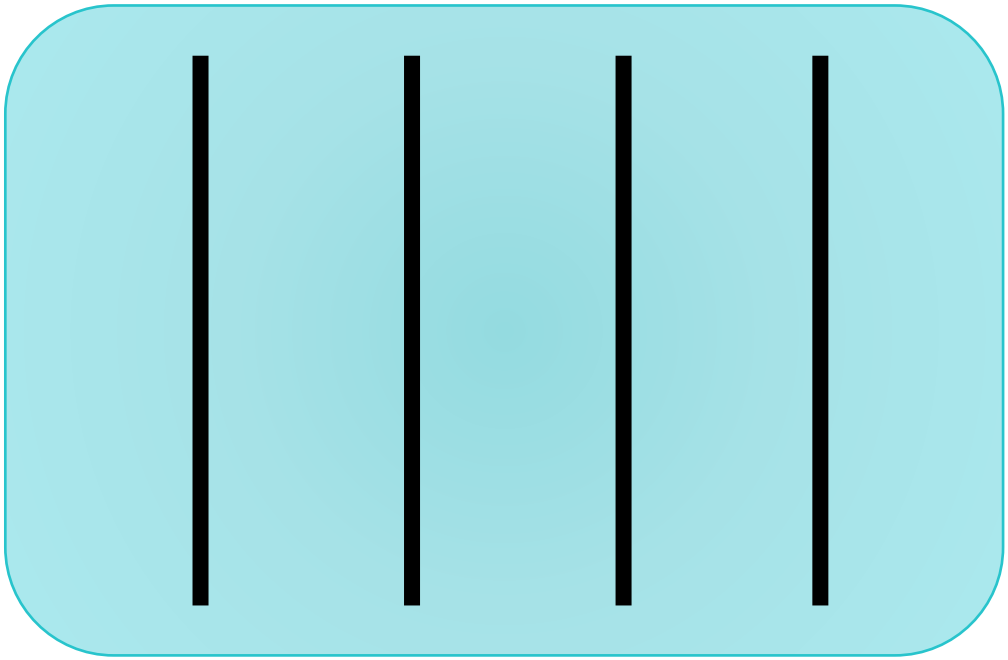
Folyamatok részei

Egy hagyományos folyamatban egy vezérlő szál és egy utasításmutató található:



Több szál egy folyamaton belül

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



Egy folyamat több szállal

Szálak (thread)

Egy szál koncepcionálisan hasonlít egy folyamathoz

A szál felhasználói szintű fogalom

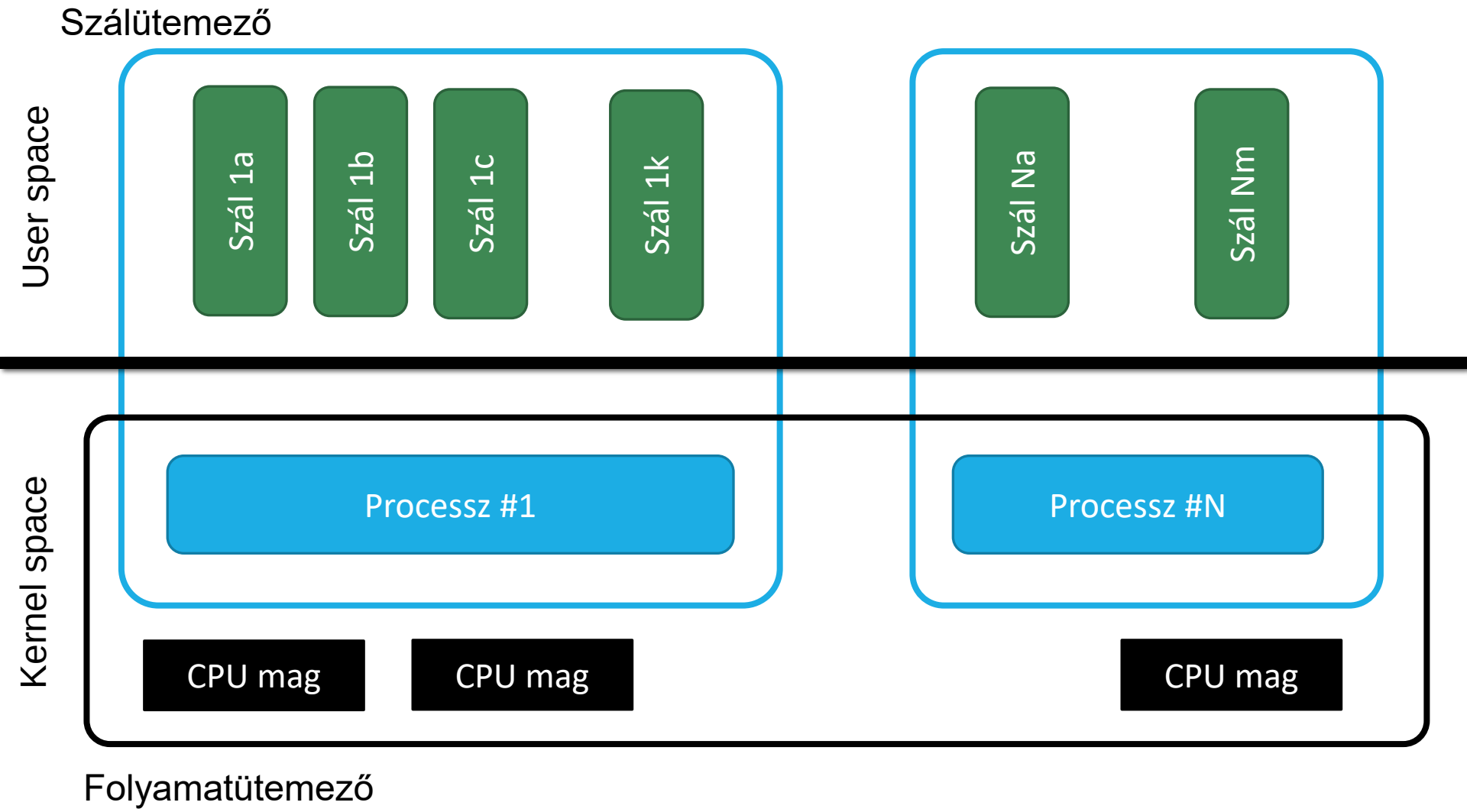
- (Nem kernel szintű)
- A szálleíróstruktúra is a felhasználói területen van, és ezért közvetlenül elérhető.
- A szálak, mint programokat futtató taszkok, a folyamatokhoz hasonlóan rendelkeznek regiszter adatokkal
  - utasításszámláló – program counter
  - veremmutató – stack pointer
    - Vagyis minden szálnak saját veremterülete van.
  - állapotleíró adatok
  - stb.

Szálak (thread)

A szál felhasználói szintű fogalom

- A szálaknak ugyanúgy lehet futó, kész és blokkolt állapota.
- Egy folyamat minden szála osztozik az erőforrásokon
  - ugyanazon memória területet látja
  - ugyanazokat a műveleteket és adatokat használja mindegyik szál.
- Például
  - 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.

Folyamatok és szálak



Kapcsolatok processzek között

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 az egyik beírja az átadni kívánt információt egy változóba, a másik folyamat pedig kiolvassa onnan.
    - Fontos a folyamatok összehangolása
      - Ha két folyamat például egyidőben ír ugyanarra a memóriaterületre (ugyanabba a változóba) a kapott eredmény határozatlan érték lesz…

Kapcsolatok processzek között

Kommunikáció

- Történhet
  - Explicit üzenetküldéssel
    - Az egyik folyamat üzenetet küld a másik folyamatnak.
    - Aszinkron kommunikáció
      - Amikor 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 kerül felfüggesztésre
    - Szinkron kommunikáció (randevú)
      - Amikor egy folyamat kommunikálni akar egy másik folyamattal, akkor végrehajtása felfüggesztésre kerül addig, amíg a megszólított folyamat alkalmas nem lesz az üzenetvételre.
      - A két folyamat csak a kommunikáció befejeztével futhat tovább.

Kapcsolatok

Szinkronizáció

- Az egyik processz vezérlését kapcsolatba hozza a másik processz vezérlésével
- p a P processz egy végrehajtási pontja és q a Q processzé, akkor
- A szinkronizációt használjuk arra, hogy megszorításokat tehessünk arra, hogyan éri el P p-t és Q q-t.
- Több probléma is felmerülhet, amelyek elkerülésére a konkrét megvalósításkor figyelnünk kell.
  - Például deadlock:
  - A holtpont – leegyszerűsítve – olyan állapot, melynél a folyamatok körkörösen egymásra várakoznak, és egyik sem tud tovább haladni.

Példa

```
u1: x:=12;
u2: y:=x/4;
u3: a:=x+y;
u4: b:=x-y;
u5: c:=a*b;
u6: d:=x+1;
u7: e:=c+d;
```

Párhuzamosíthatók?

Példa

```
u1: x:=12;
u2: y:=x/4;
u3: a:=x+y;
u4: b:=x-y;
u5: c:=a*b;
u6: d:=x+1;
u7: e:=c+d;
```

Bizonyos utasítások csak egymás után hajthatók végre, szekvenciálisan

- Például u1 és u2, hiszen u2 használja u1 eredményét

Példa

```
u1: x:=12;
u2: y:=x/4;
u3: a:=x+y;
u4: b:=x-y;
u5: c:=a*b;
u6: d:=x+1;
u7: e:=c+d;
```

Nem mindegyik

- Az u3 és u4 egyszerre, egymással párhuzamosan is végrehajtható
- Nem függenek egymástól.

Precedenciagráf

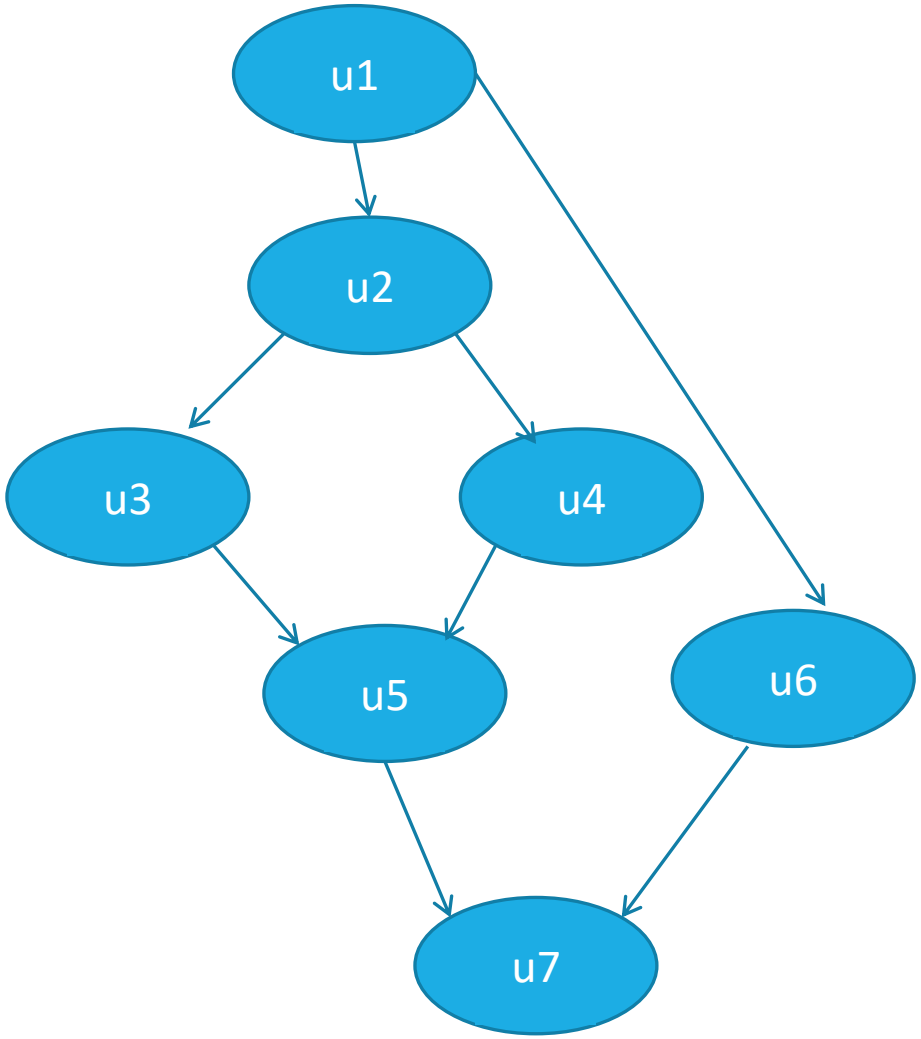
A lehetséges párhuzamosítást mutatja

- A csomópontok reprezentálják az utasításokat
- Két csomópont közötti (A és B) irányított él megadja a két utasítás sorrendjét.
  - A -> B esetén az A utasítás a végrehajtás során meg kell, hogy előzze a B utasítást.

Példa

```
u1: x:=12;
u2: y:=x/4;
u3: a:=x+y;
u4: b:=x-y;
u5: c:=a*b;
u6: d:=x+1;
u7: e:=c+d;
```

Ha több él találkozik egy csomópontban: szinkronizálás!



Precedenciagráf

A precedenciagráfok szemléletesen megadják a párhuzamosítási lehetőségeket

- Programozásra közvetlenül nem használhatók

Lehetőségek

- fork / join utasításpár
- parbegin / parend utasításpár

Fork-join pár

fork utasítás

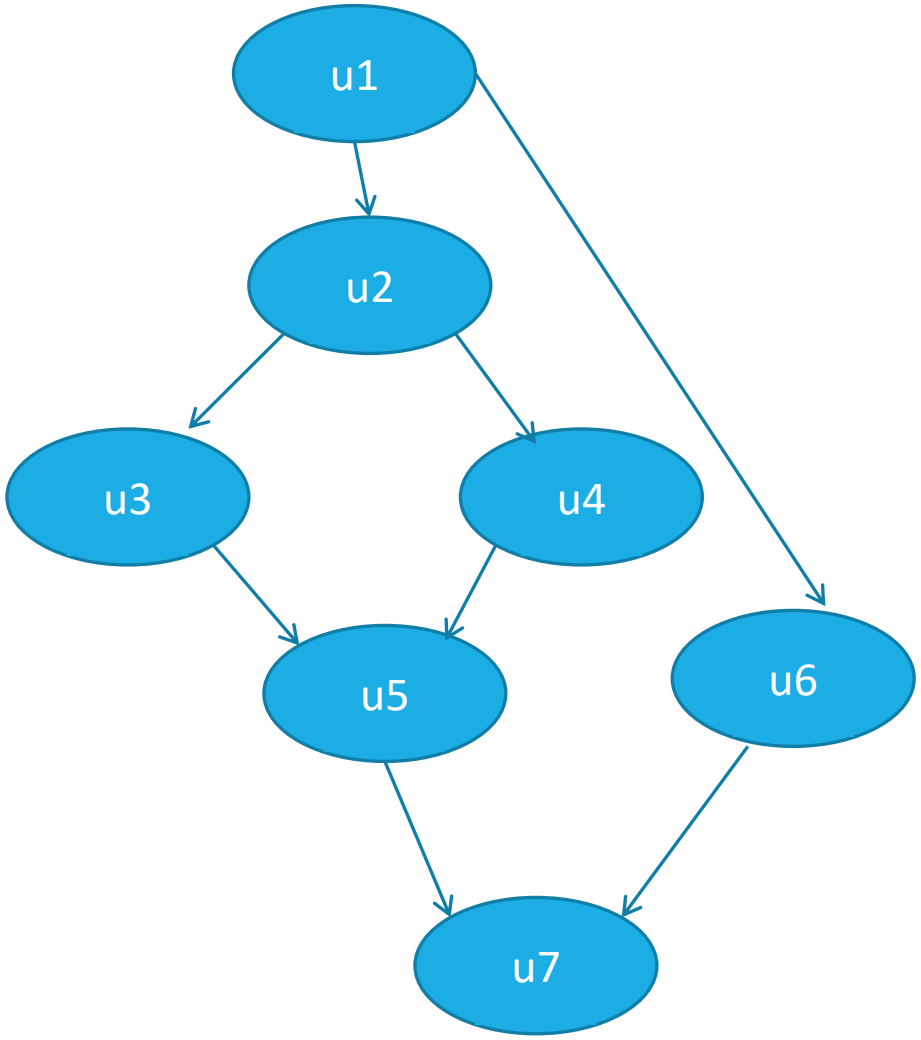
- A végrehajtás két egymással párhuzamosan végrehajtható ágra szakad
  - Az egyik ág közvetlenül a fork utasítás után folytatódik
  - A másik ág a megadott címkénél található

join utasítás

- A két vagy több ág egyesítése
- az ágak „bevárják egymást” (szinkronizáció) és csak a legutoljára „érkező” folytatódik, a többi „meghal”
- az egyesítendő ágak számát egy számláló mutatja

Fork-join pár

```
    száml2 := 2;
    száml1 := 2;
    u1;
    fork L1;
    u2;
    fork L2;
    u3; goto L3;
L2: u4;
L3: join száml1;
    u5; goto L4;
L1: u6;
L4: join száml2;
    u7;
```



Parbegin-parend pár

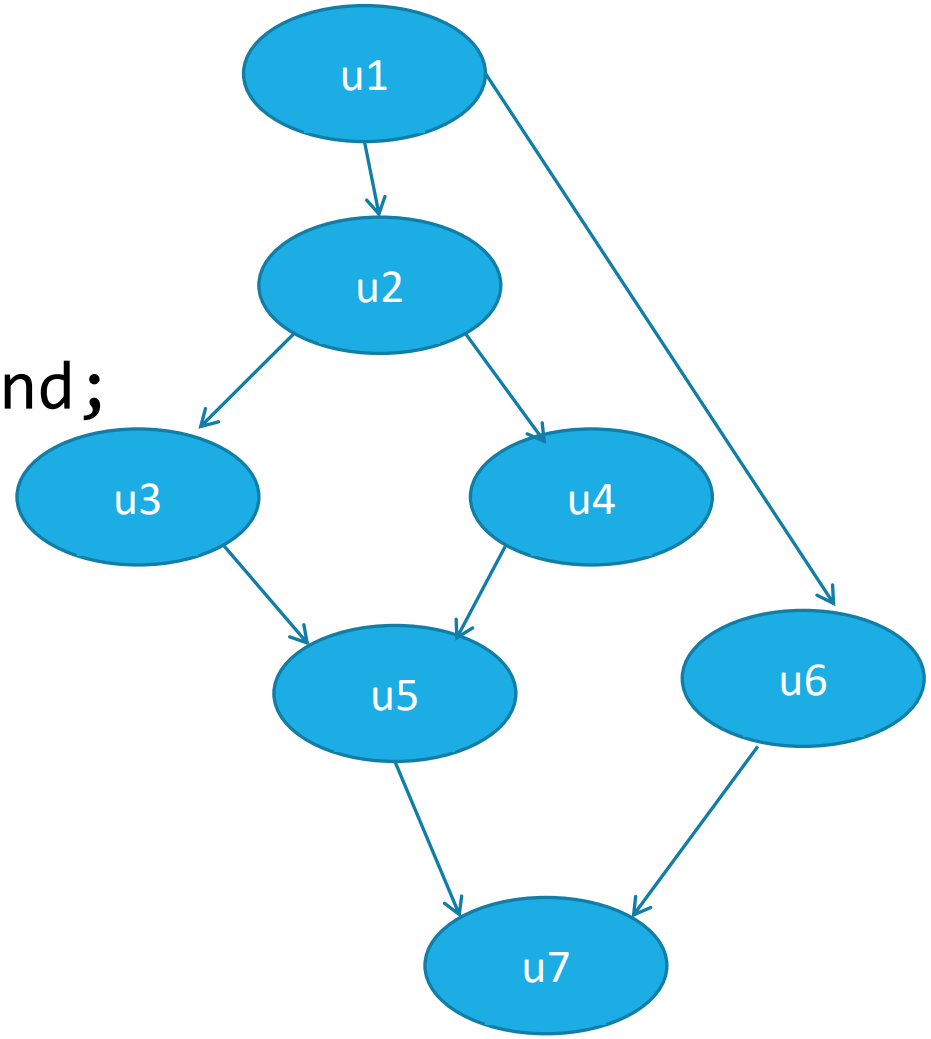
```
u1;

parbegin
u6;

begin
   u2;
   parbegin u3; u4; parend;
   u5;
end;

parend;

u7;
```



Ez nem jó minden precedenciagráfra

- ki kell kiegészíteni ilyenkor
  - Például szemaforokkal

Viselkedés

A processzek viselkedésük alapján lehetnek

- Egymástól függetlenek (independent)
- Egymással versengők (competing)
  - Többen szeretnének megszerezni egy adott erőforrást, amit csak kizárólagos módon lehet használni
    - Repülőgépen egy adott járat adott ülőhelye
    - Adott nyomtató
  - Kölcsönös kizárás kell, szinkronizáció
- Egymással együttműködők (cooperating)
  - Amikor ugyanazon probléma különböző részein dolgoznak
    - Puzzle

Függetlenség?

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Ketszal is
     task Egyik;
     task body Egyik is
     begin
          for i in 1..1000 loop
               Put_Line("Szia!");
          end loop;
     end Egyik;
begin
     for i in 1..1000 loop
          Put_Line("Viszlát!");
     end loop;
end Ketszal;
```

Lehetséges kimenetek

Példaprogramok és eredményfájlok put-ra és print-re

- Ketszalu_put.adb – ketszal_put.txt
- Ketszalu_print.adb – ketszal_print.txt
- Ketszalu_put_print.adb – ketszal_put_print.txt
- Ketszalu_put_line.adb – ketszal_put_line.txt

Tapasztalatok

- Sokszor az egyik (például Viszlát!)
- Utána sokszor a másik
- Nem ugyanannyiszor
- Változik futás közben, hogy melyikből mennyi
- Futásról futásra is változik

Ütemezés jelentősége

A futtató rendszer ütemezi a szálakat

- Operációs rendszer
- Futtató környezet (Előző példában Ada; Java)
- Programozó (Co-rutionk)

Például az Ada nyelv nem tesz megkötést az ütemező algoritmusra

- Egy implementációban bármilyen ütemező lehetséges

Jó program: bármely ütemezésre jó

Időosztásos ütemezés

Jellemzők

- Nem valódi párhuzamosság
- Egy processzor: egyszerre csak egyet
- Időszeletek (time slicing)
- Gyorsan váltogatunk, párhuzamosság látszata
- Egyenlő időszeletek: pártatlanság
- Kooperatív viselkedés

Kompetitív viselkedés

Jellemzők

- „Fusson, amíg van mit csinálnia”
- Amíg számol, addig övé a vezérlés
- Vezérlés elvétele: csak speciális pontokon
  - blokkolódás, szinkronizáció
- Hatékonyabb, mint az időszeletes ütemezés
  - Kevesebb kontextusváltás (context switch)
- Nem pártatlan:
  - Egy taszk kisajátíthatja az erőforrásokat

Pártatlanság, kooperáció

Egy taszk lemondhat a vezérlésről

delay utasítás:

```
loop
    Put_Line("Szia!");
    delay 0.0;
end loop;
```

Jellemző kimenet: felváltva írnak ki, de ez sem mindig

- Ketszalu_put_delay.adb – ketszal_put_delay.txt

Multiprogramozott rendszer

A multiprogramozott rendszer

- Szekvenciális programok halmazának tekinthető
- A programok 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.

Multiprogramozott rendszer

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.

Multiprogramozott rendszer

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

Szinkron kommunikáció

Amikor egy folyamat kommunikálni akar egy másik folyamattal, akkor a végrehajtása felfüggesztésre kerül addig, amíg a megszólított folyamat alkalmas nem lesz az üzenetvételre.

- A két folyamat csak a kommunikáció befejeztével futhat tovább.
- Előnye az aszinkron kommunikációval szemben, hogy az információ áramlása kétirányú lehet.
- Ezt a kommunikációs modellt használja pl. az Ada nyelv.

Ada – szinkron kommunikáció

Randvú – 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

Példa

```
task HF is
     entry Bead;
end HF;

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

```
task Diak;

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

Randevú az Adában

Tulajdonságok

- Aszimmetrikus
  - Megkülönböztetjük a hívót és a hívottat
    - diák és HF
- 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

Randevú az Adában

Tulajdonságok

- Szinkron kommunikáció
  - 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:

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

Példa

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

```
task Diak;

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

Információcsere

Kommunikáció

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

Példa

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

task body Tarolo 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 Tarolo;
```

Példa – próba

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

Aszinkron kommunikáció

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);
     Tarolo.Betesz(Ch);
     ...
end;
```

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

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

Osztott memória használata

Közös változók

- A folyamatok ugyanazt a memóriarészt látják.
- Probléma adódhat abból, hogy két folyamat egy időben írhatja az adott erőforrást (pl. változót), ilyenkor az eredmény határozatlan érték lesz.
- Hasonló probléma merülhet fel egy írási és egy olvasási művelet összeakadásánál.

Nyelvi elemek

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?

Nyelvi elemek

Kommunikációhoz

- 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

Konkurrens részek szinkronizálására megfelelő eszközök

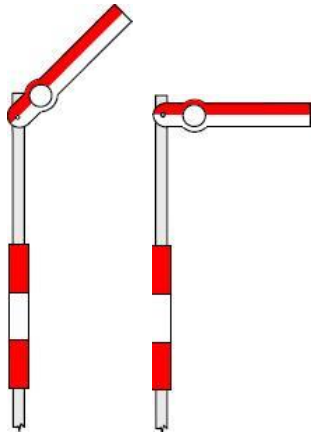
A prioritások meghatározása

Késleltetések, időzítések kezelése

Nyelvi elemek

Szemafor

- A szemafort, mint típust, Dijkstra vezette be egy 1968-as művében
- Egy egész értékű számláló
  - Illetve egy hozzá tartozó várakozási sor.
- Egy inicializált szemafornak két megengedett művelete van:
  - P = passeren (áthaladni)
  - V = vrijmaken (szabaddá tenni)
- A p - wait, a v - signal.



Nyelvi elemek

Szemafor

- 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

Lehetséges megvalósítás Adában

```
task A;                              -- kritikus szakasz előtti
task Szemafor is                     --   utasítások
     entry P;                        Szemafor.P;  -- megvárjuk,
     entry V;                                     -- amíg beenged
end Szemafor;                        -- kritikus szakasz utasításai
                                     Szemafor.V;
task body Szemafor is                -- kritikus szakaszok közötti
begin                                --  utasítások
     loop                            Szemafor.P;
          accept P;                  -- kritikus szakasz utasításai
          accept V;                  Szemafor.V;
     end loop;                       -- kritikus szakasz utáni
end Szemafor;                        --  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;
```

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 – Hoare, 1974

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.

Az étkező filozófusok problémája

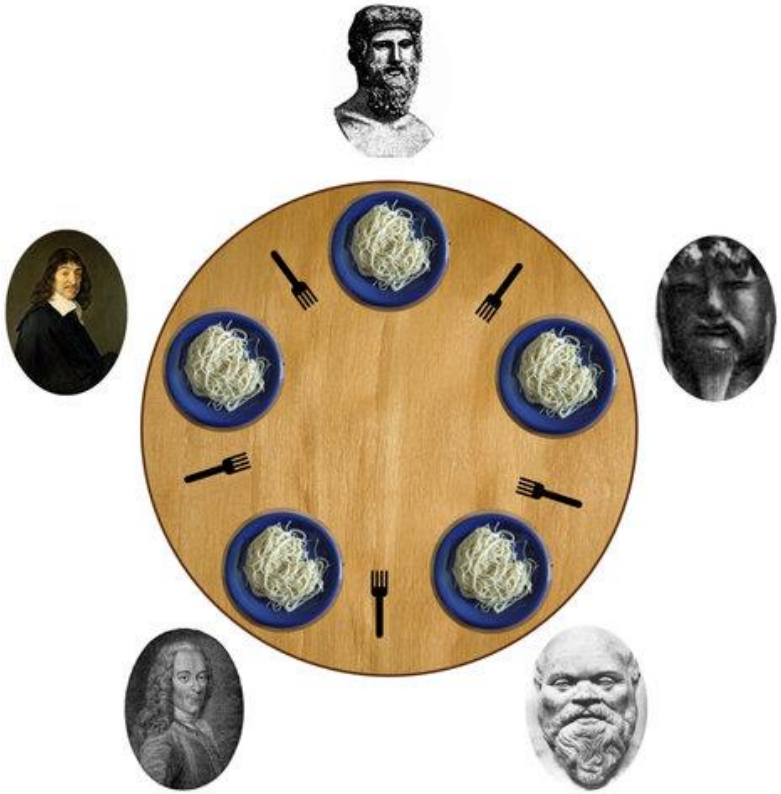
Dijkstra: dining philosophers

Erőforráshasználat modellezése

- Erőforrások: villák
- Folyamatok: filozófusok

Holtpont kialakulása

Éheztetés veszélye



Holtpont

Deadlock

- Ha a folyamatok egymásra várnak
- Ha egyszerre éheznek meg a filozófusok

Elkerülés

- Például a szimmetria megtörésével:
  - Erőforrások lekötése rendezés szerint
  - Véletlenszerű időkorlátok
- Ajtó bevezetésével

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

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

Egy task írhat egyszerre

```
task Kizaro is
   entry Kiir( Str: String );
end Kizaro;

task body Kizaro is
begin
   loop
    accept Kiir(Str: String) do
          Text_IO.Put_Line(Str);
    end Kiir;
   end loop;
end Kizaro;
```

Egy task írhat egyszerre

```
protected Vedett is
      procedure Kiir ( ... );
end Vedett;

protected body Vedett is
      procedure Kiir ( ... ) is ... end;
end Vedett;

Vedett.Kiir(...);
```

További lehetőségek

Lehetséges nyelvi elemek még

- A fork egy párhuzamos folyamat elindítását váltja ki.
- A join újra egyesíti a párhuzamos folyamatokat.
- 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, delay késleltet egy folyamatot.
- A quit befejez egy processzt.

Nyelvek

Nyelvek osztályozása

Vezérlési modellek szerint

- „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.

Nyelvek osztályozása

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.

- Ezek többprocesszoros gépek programozási nyelvei.

Nyelvek osztályozása

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.

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.

Osztott változókkal rendelkező nyelvek

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.

Osztott változókkal rendelkező nyelvek

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.

Simula-67

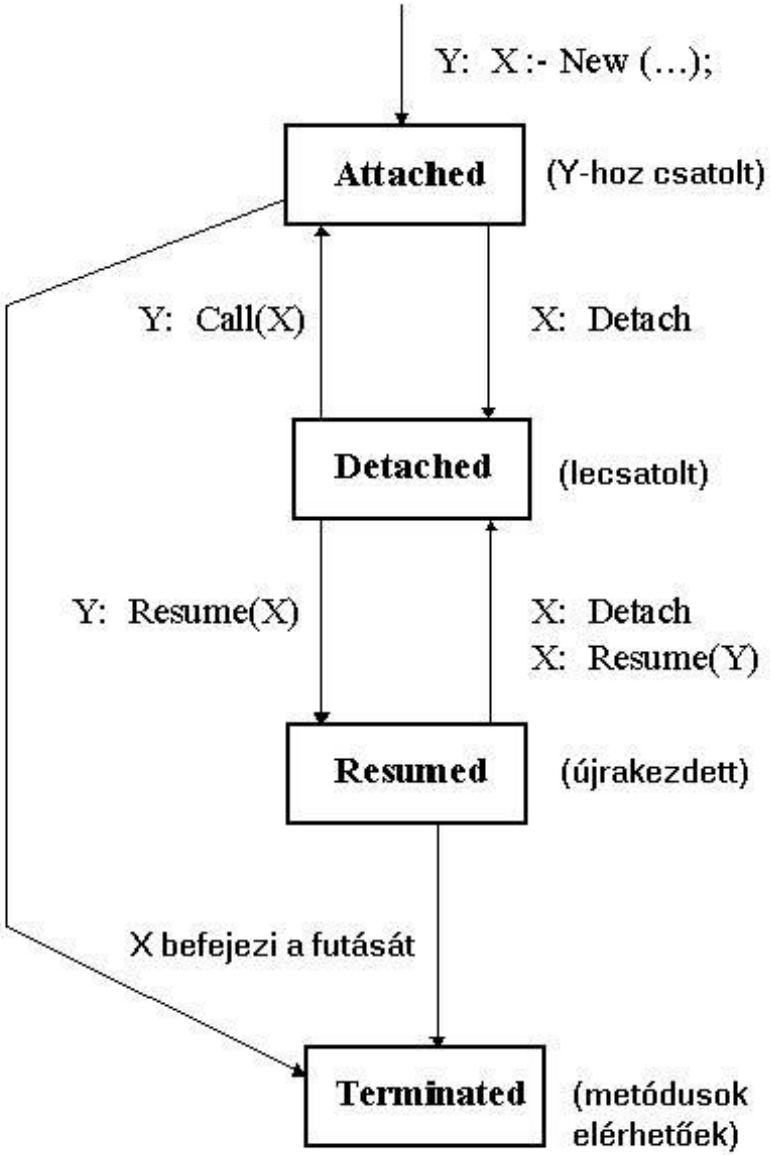
Egy korutin objektum élettartalma a következő:

- 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.

Simula-67

Egy korutin objektum élettartalma a következő:

- 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)

Ada

Egy Ada programban egy folyamatot egy task-objektum reprezentál.

A taskok rendelkezhetnek belépési (entry) pontokkal.

- Ezek hívhatóak egy másik taszkból
- Így kommunikálhatnak a taszkok.

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 – Randevú

```
task Lany is
    entry Randi;
end Lany;

task body Lany is
begin
    …
    accept Randi;
    …
end Lany;
```

```
task Fiu;

task body Fiu is
begin
    …
    Lany.Randi;
    …
end Fiu;
```

Ada – Randevú

Pont-pont kapcsolat

- Egy randevúban mindig két taszk vesz részt
  - Egy fiú és egy lány
- Szerepek
  - A taszkok szerepi felcserélődhetnek
- Nincs „broadcast” jellegű randevú

Aszimmetrikus

- Megkülönböztetjük a hívót és a hívottat
  - Fiú és lány
- Másként működik a két fél
  - A szintaxisban is kimutatható a különbség
  - A hívó ismeri a hívottat, de fordítva nem

Ada – Randevú

Az accept törzse

- A randevú az accept utasítás végrehajtásából áll
- Ez alatt a két taszk „találkozik”
- A fogadó taszk hajtja végre az accept-et
- Az accept-nek törzse is lehet, egy utasítássorozat

Szinkronitás

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

Ada – Randevú

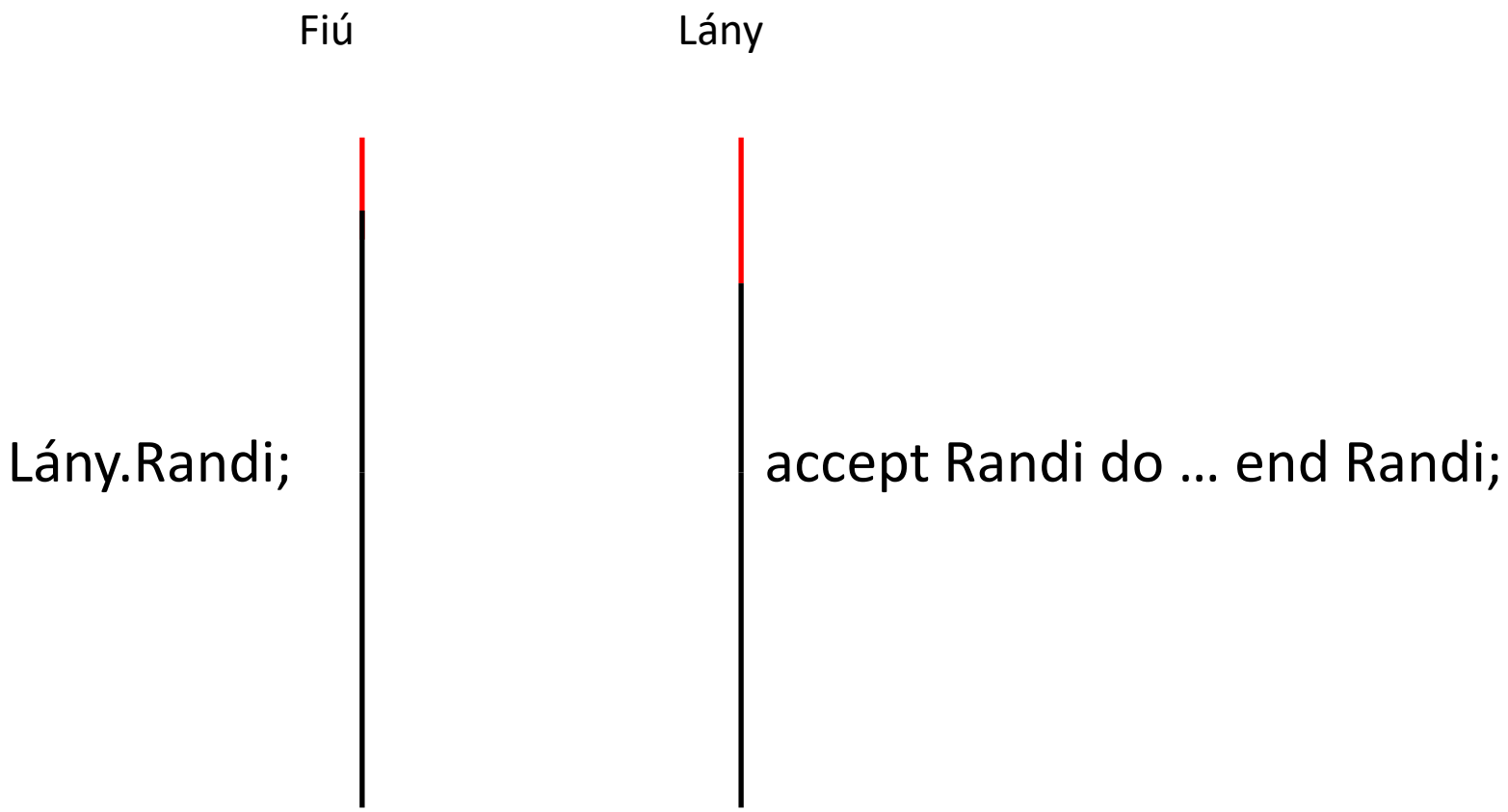
```
task Lany is
     entry Randi;
end Lany;

task body Lany is
begin
     …
     accept Randi do
     end;
     …
end Lany;
```

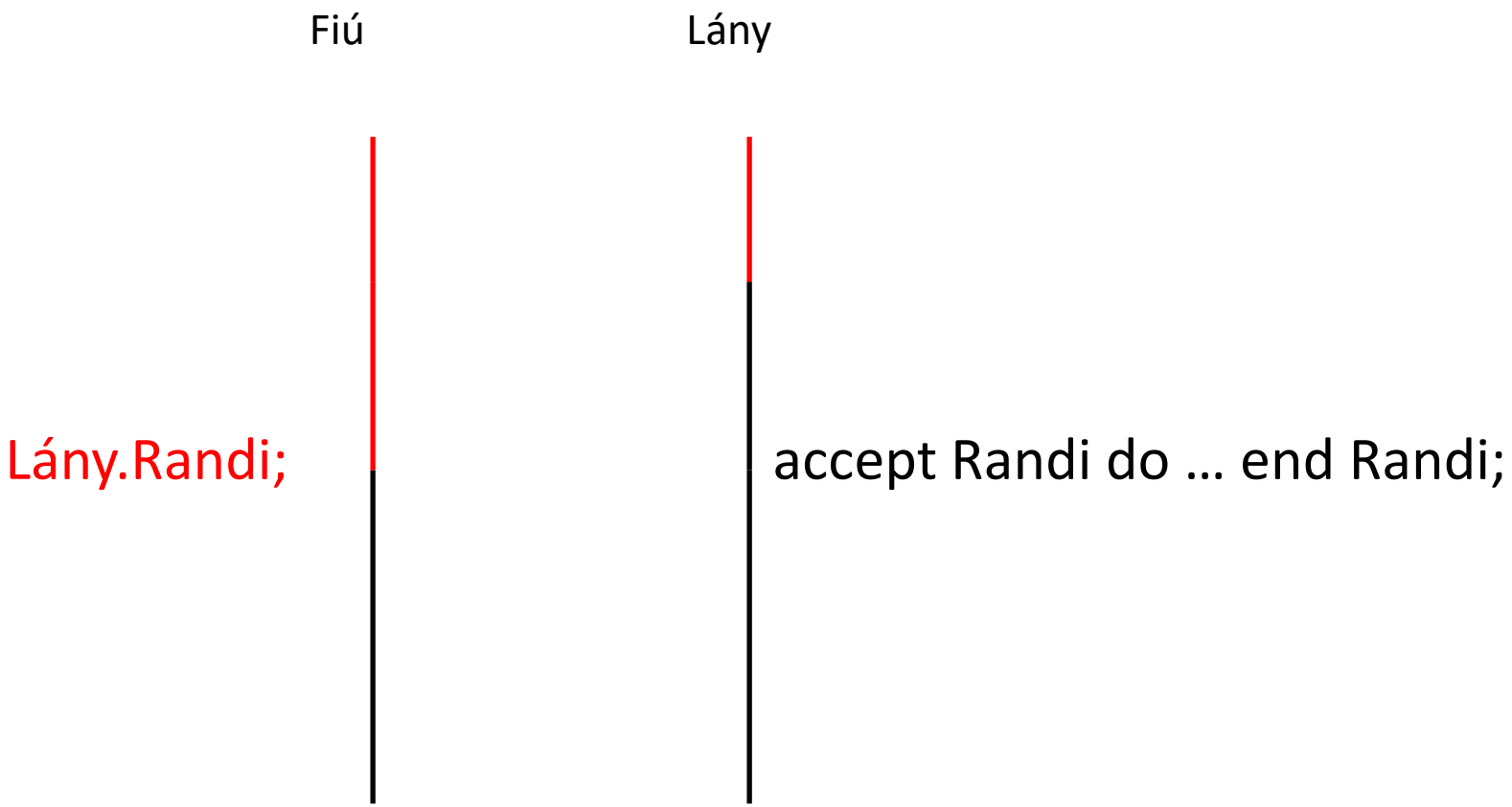
```
task Fiu;

task body Fiu is
begin
     …
     Lany.Randi;
     …
end Fiu;
```

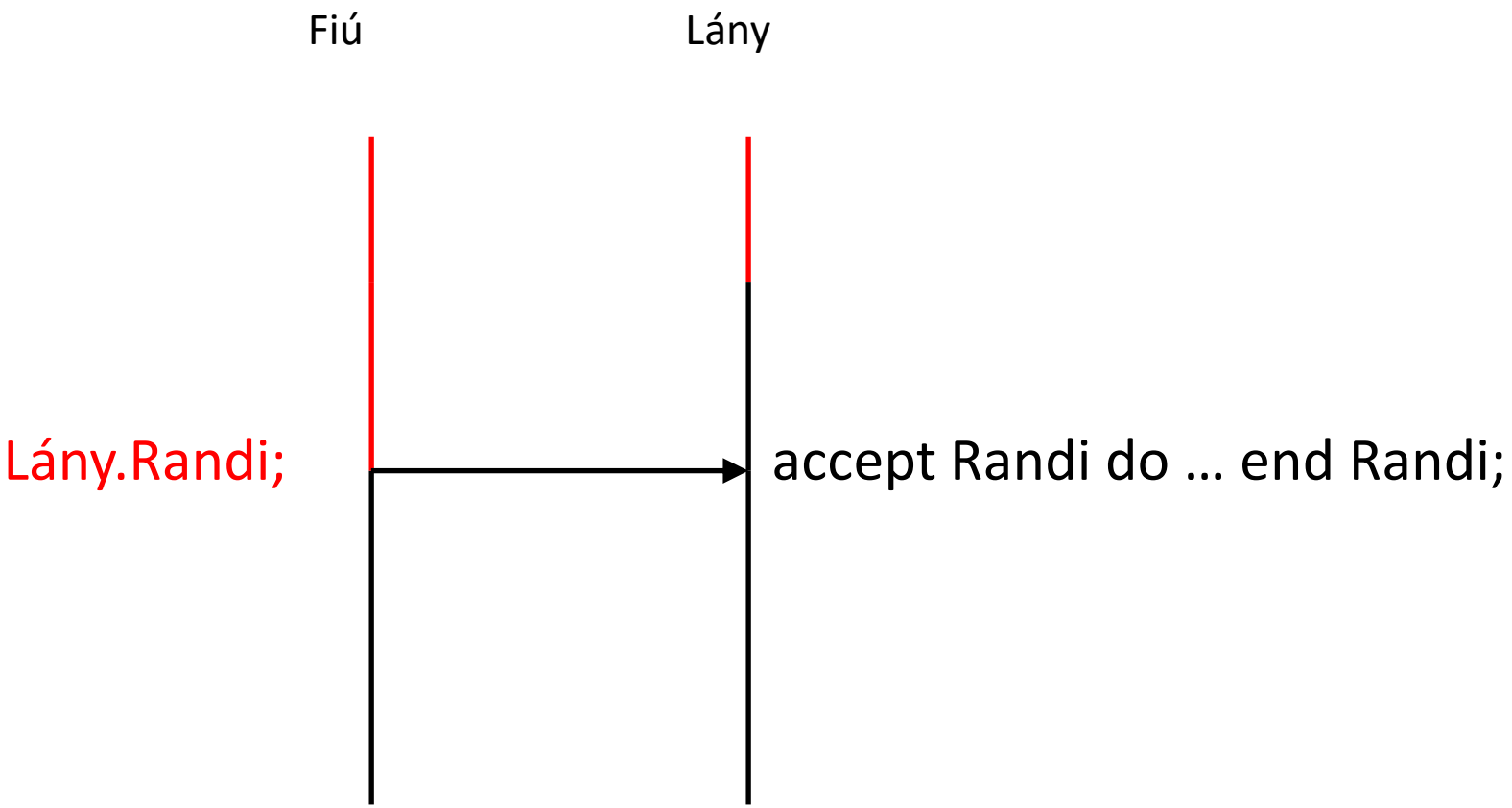
Ada – Szinkronitás



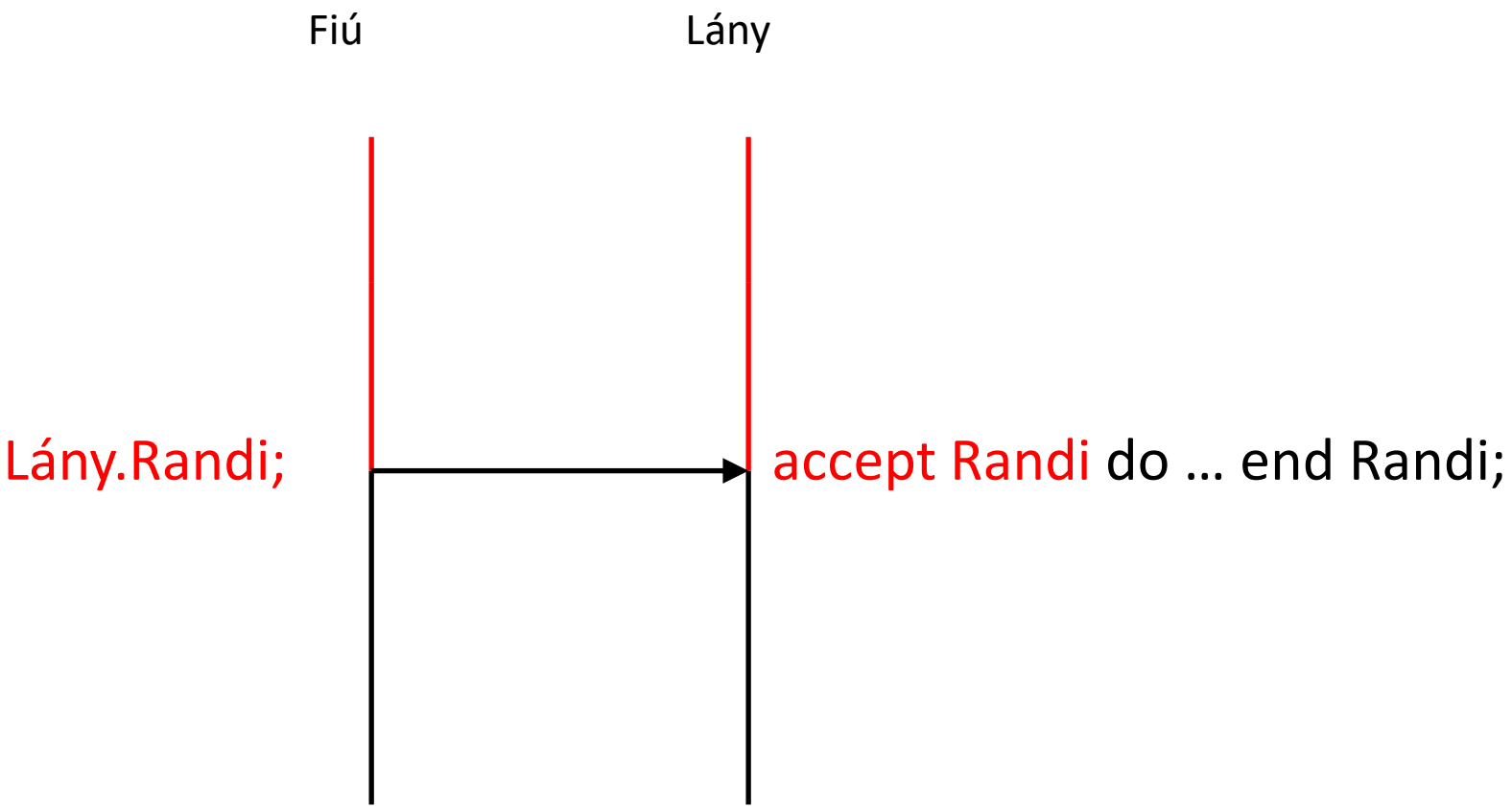
Ada – Szinkronitás



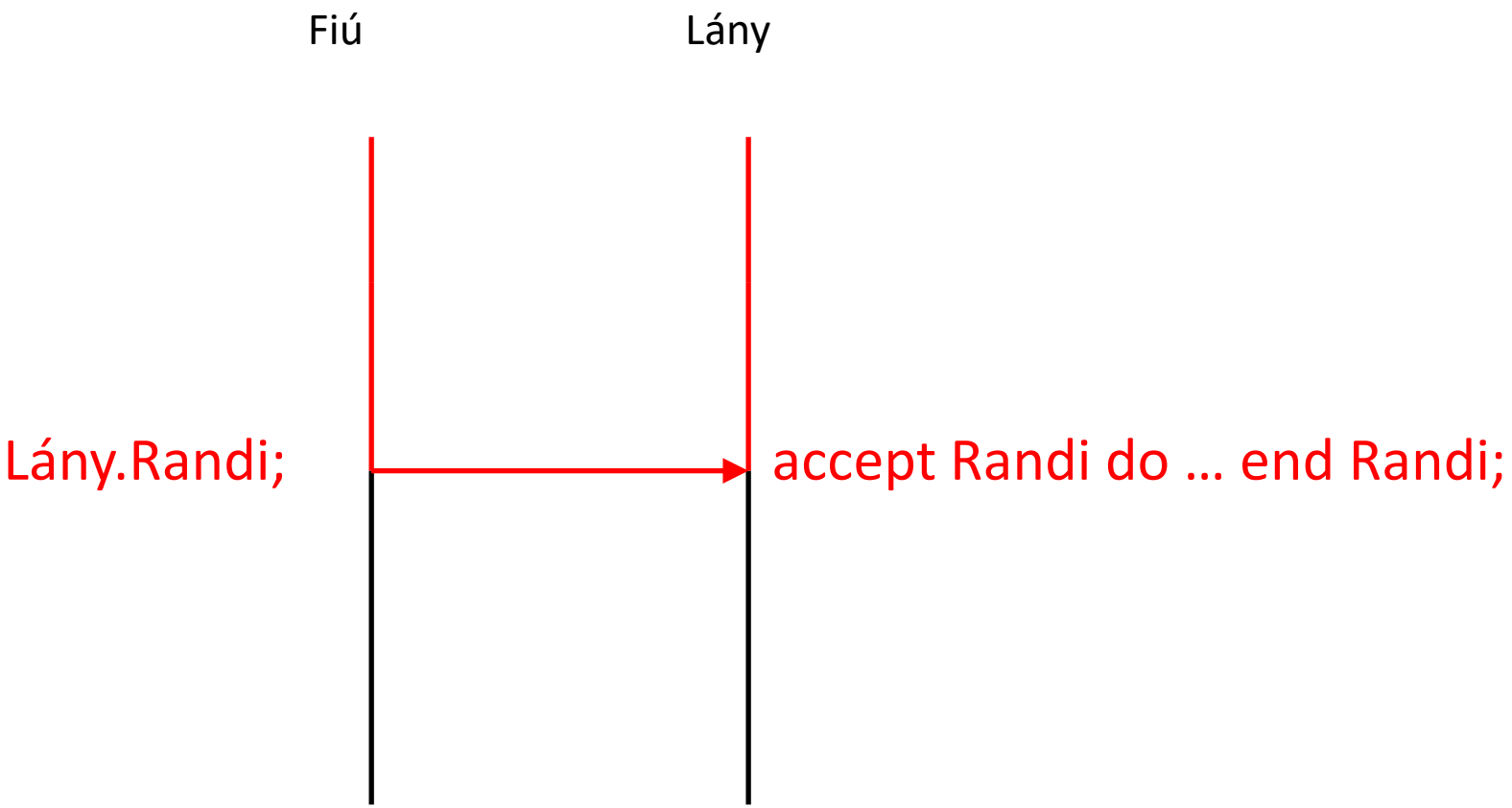
Ada – Szinkronitás



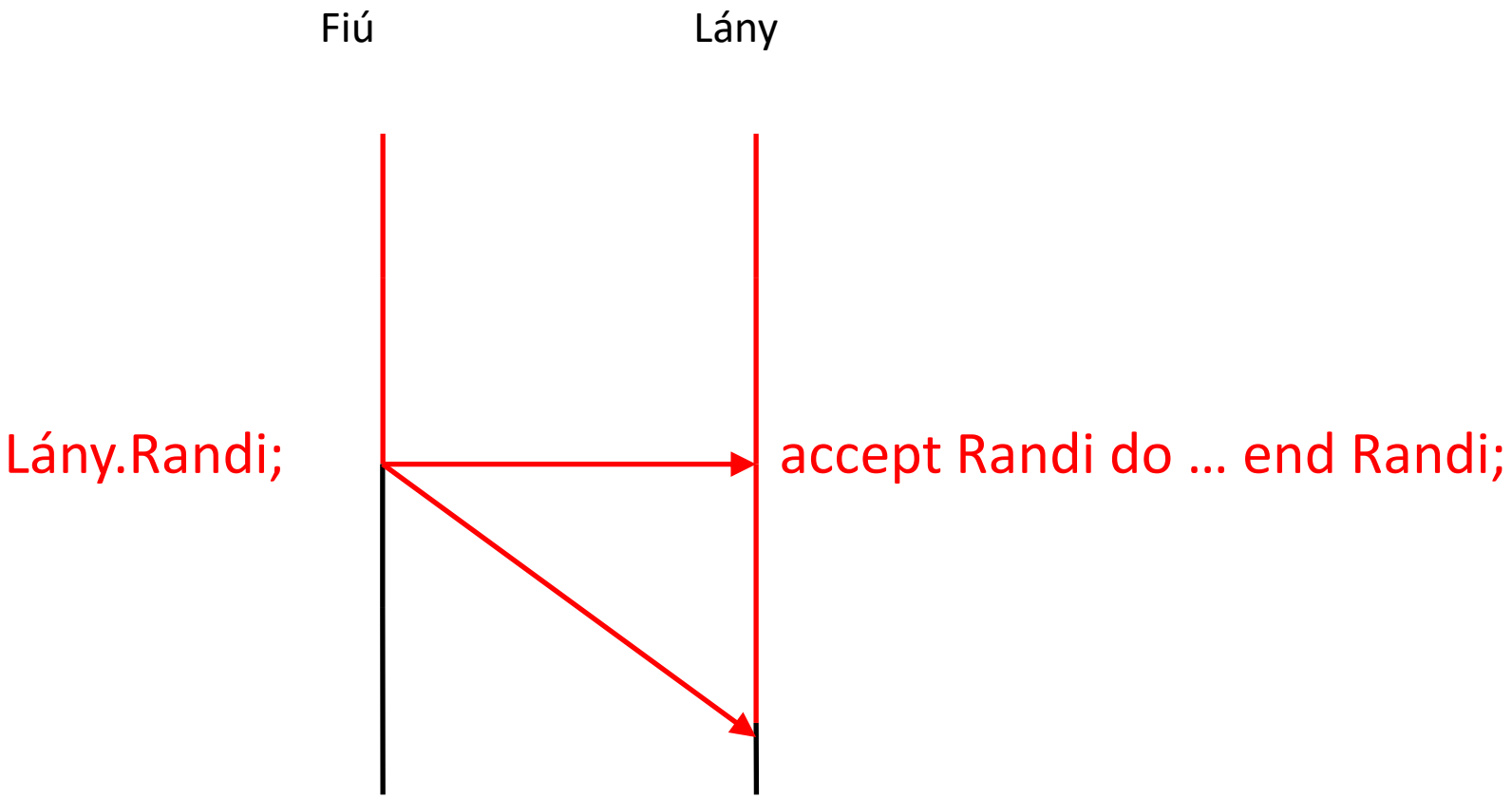
Ada – Szinkronitás



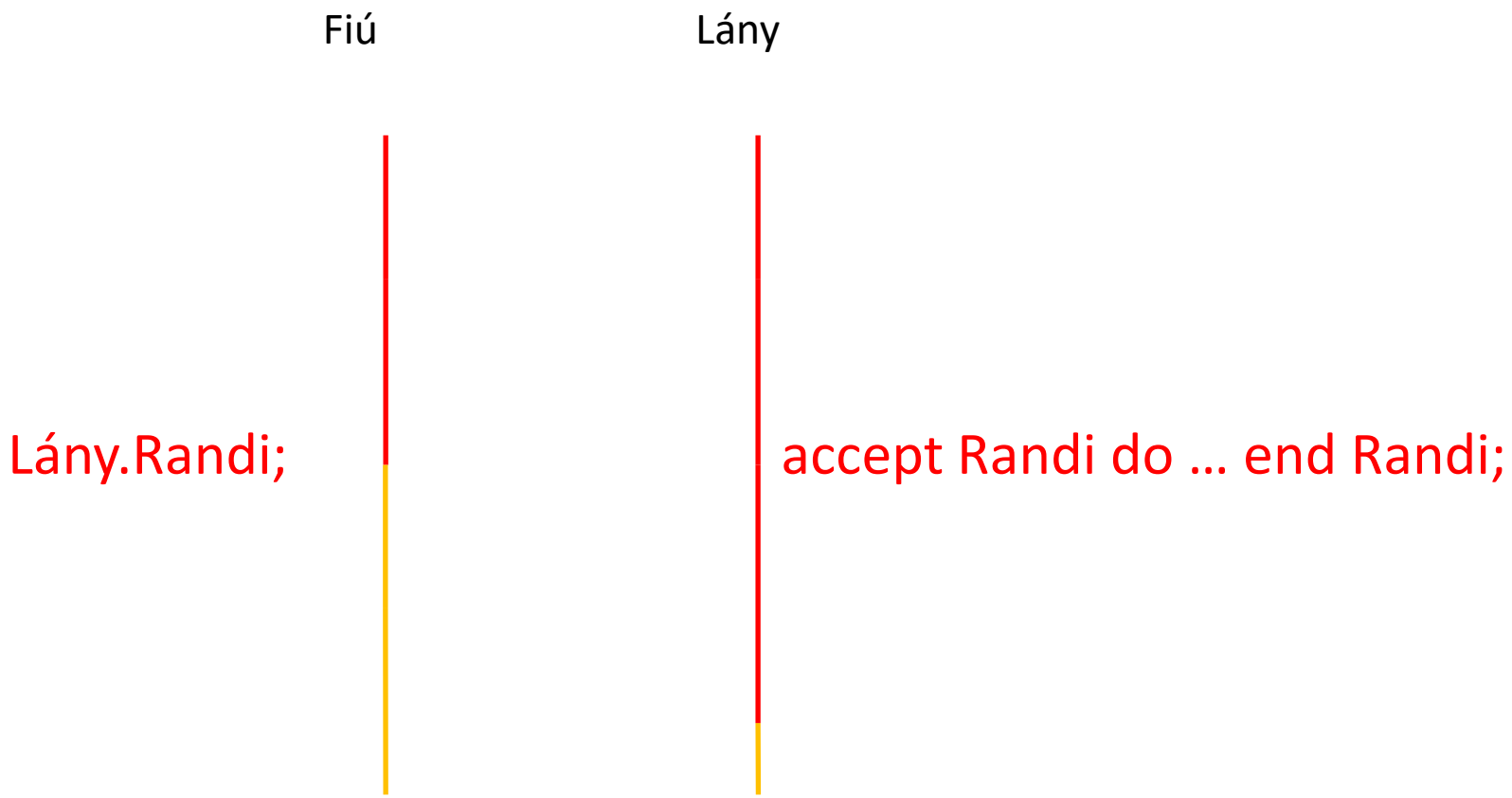
Ada – Szinkronitás



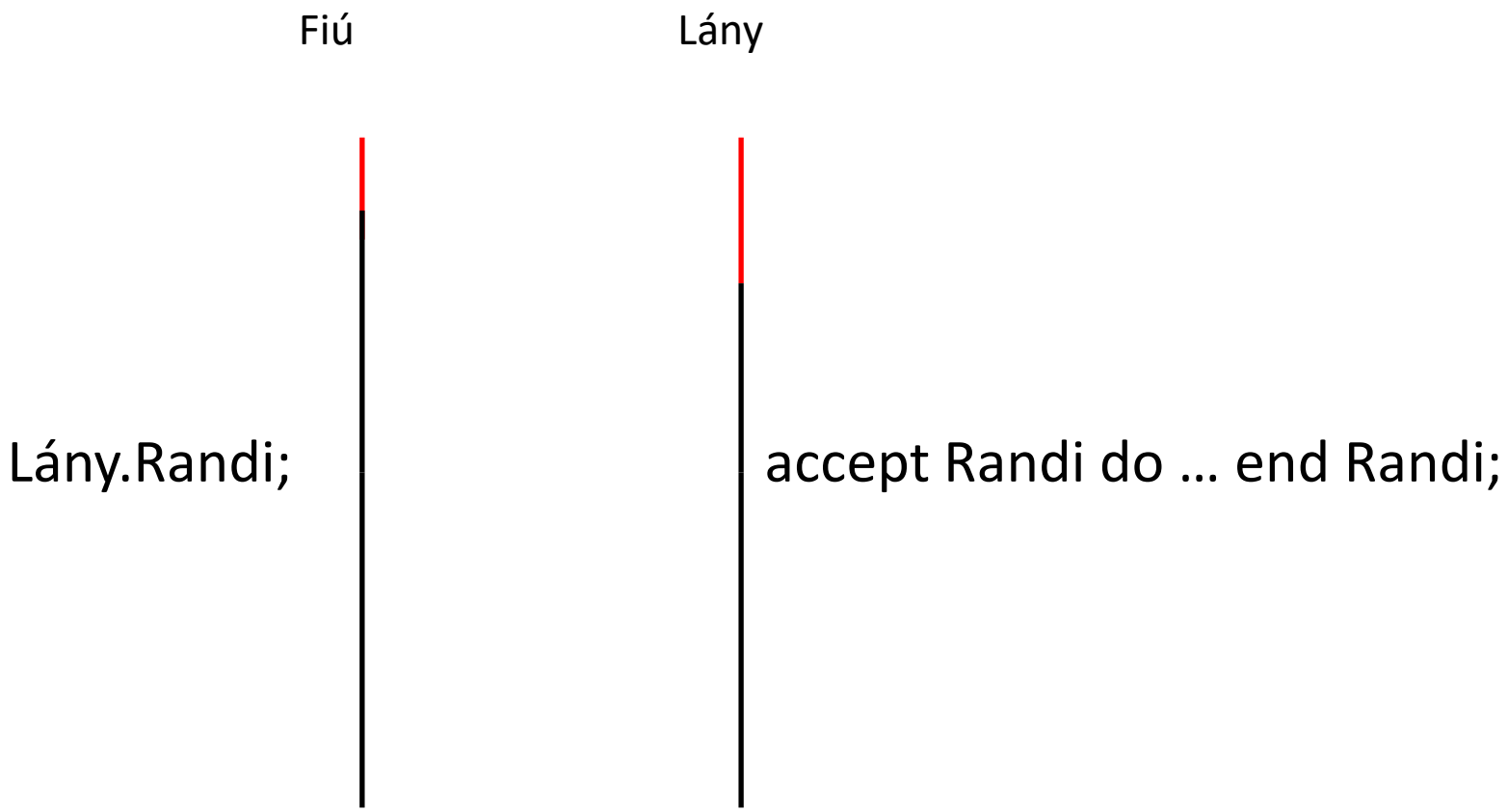
Ada – Szinkronitás



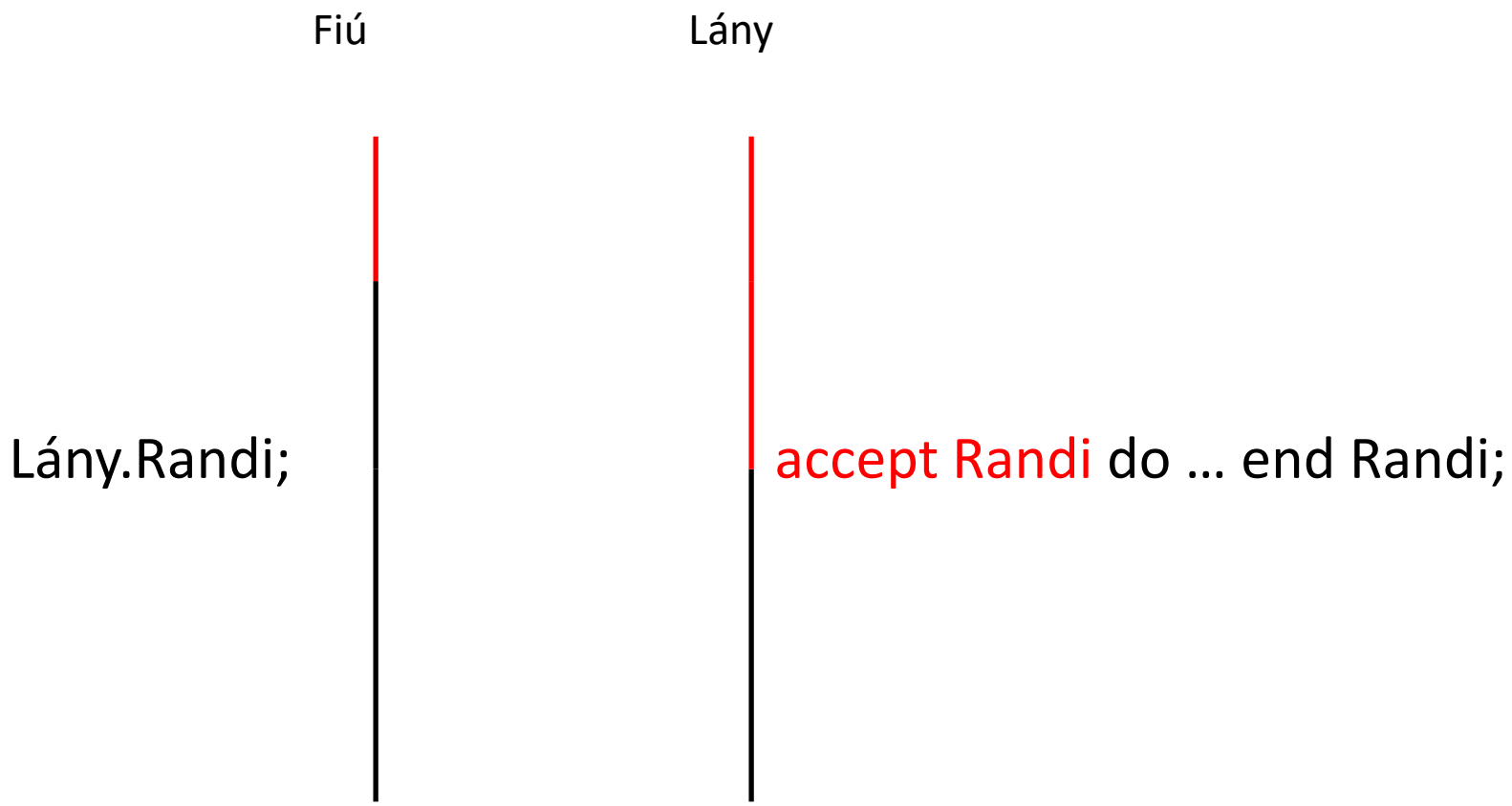
Ada – Szinkronitás



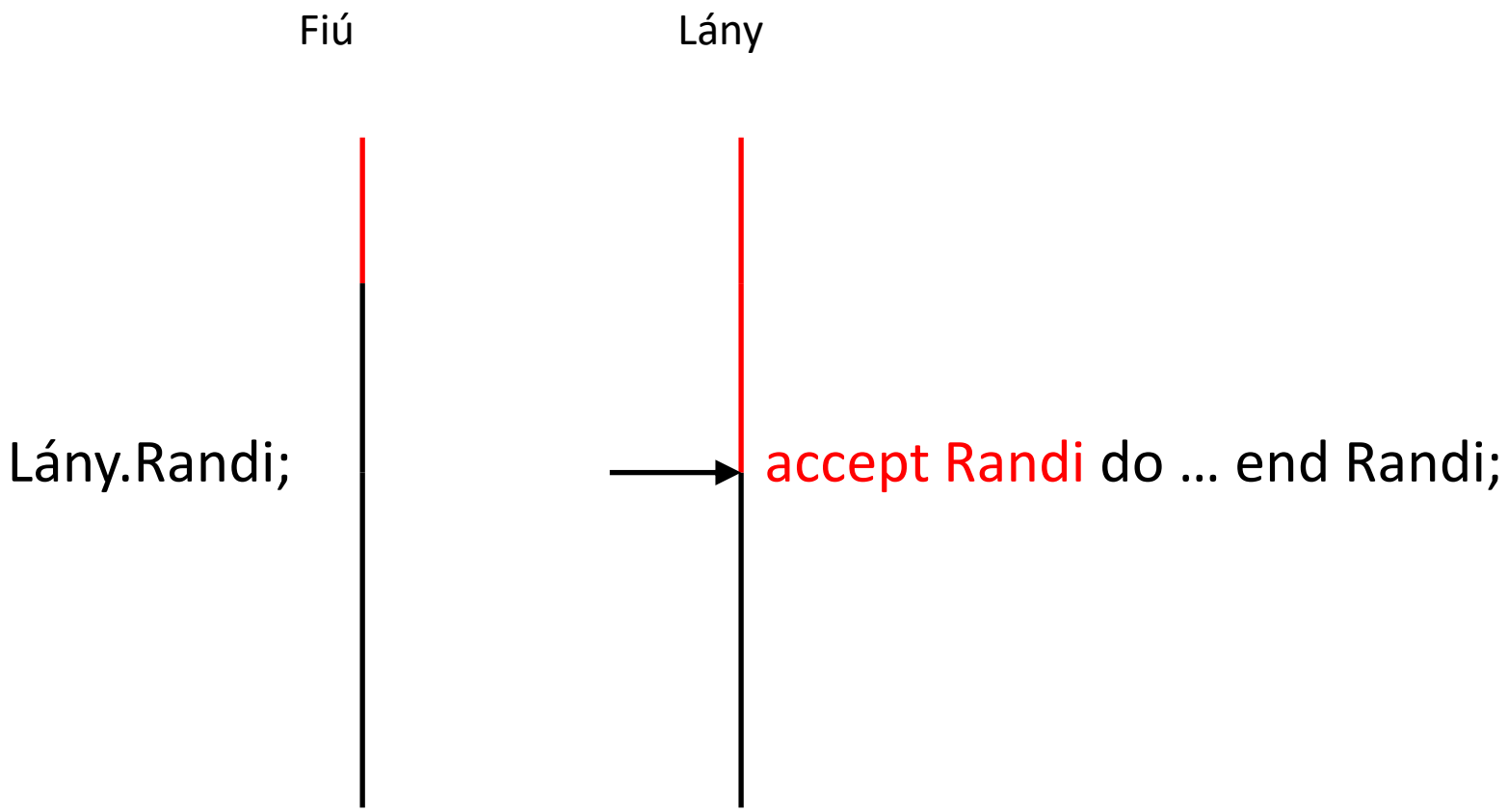
Ada – Szinkronitás



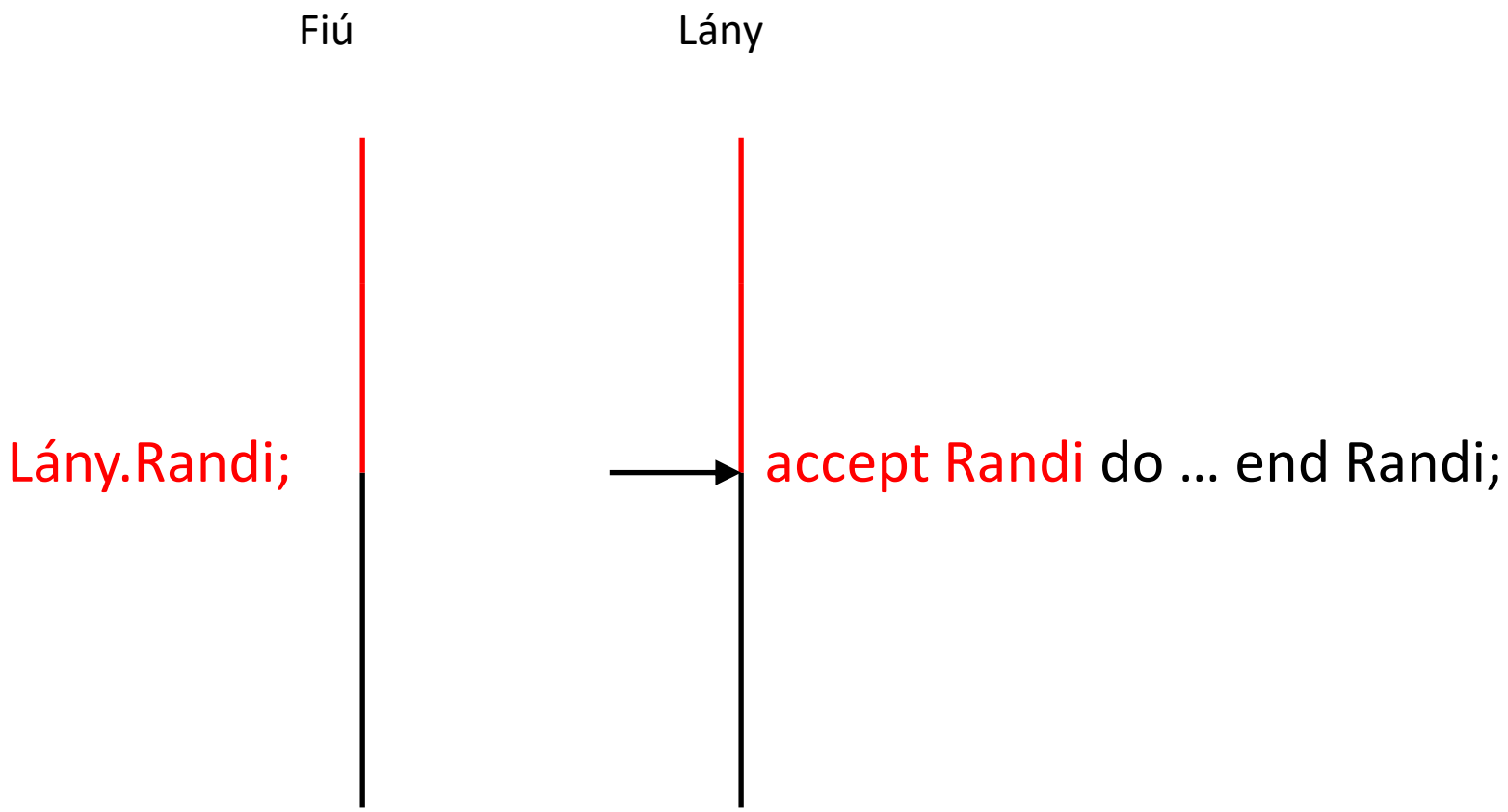
Ada – Szinkronitás



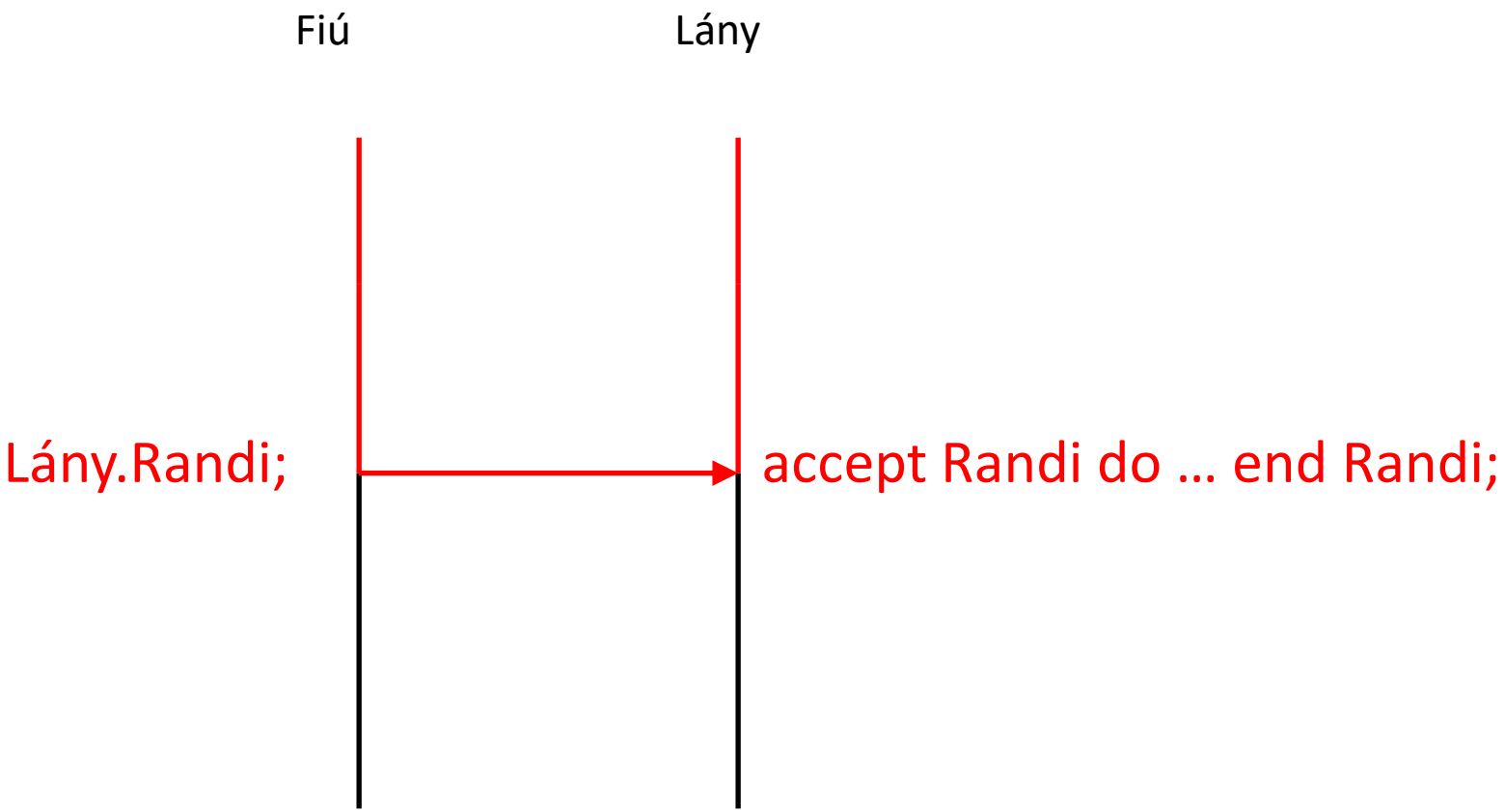
Ada – Szinkronitás



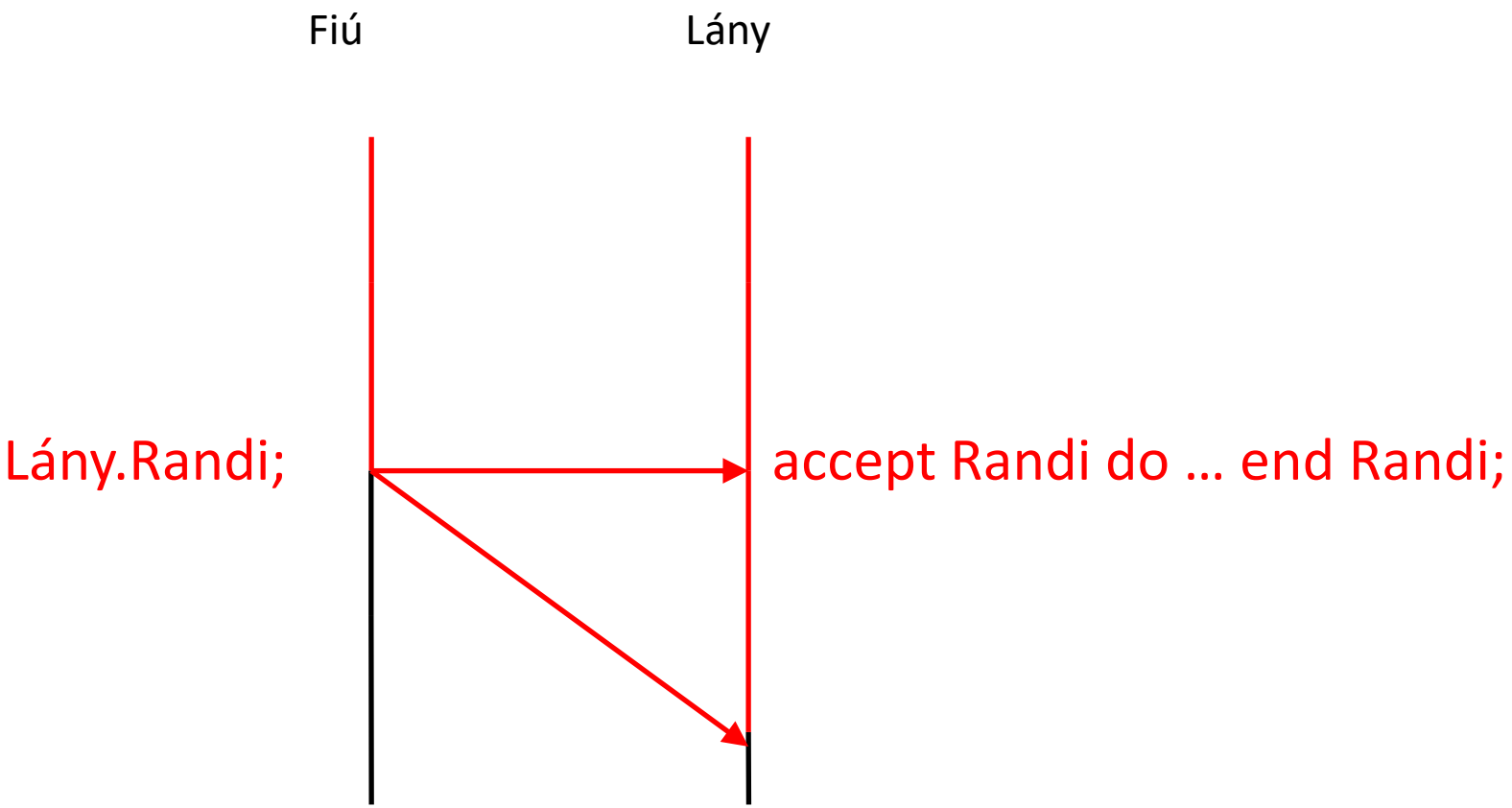
Ada – Szinkronitás



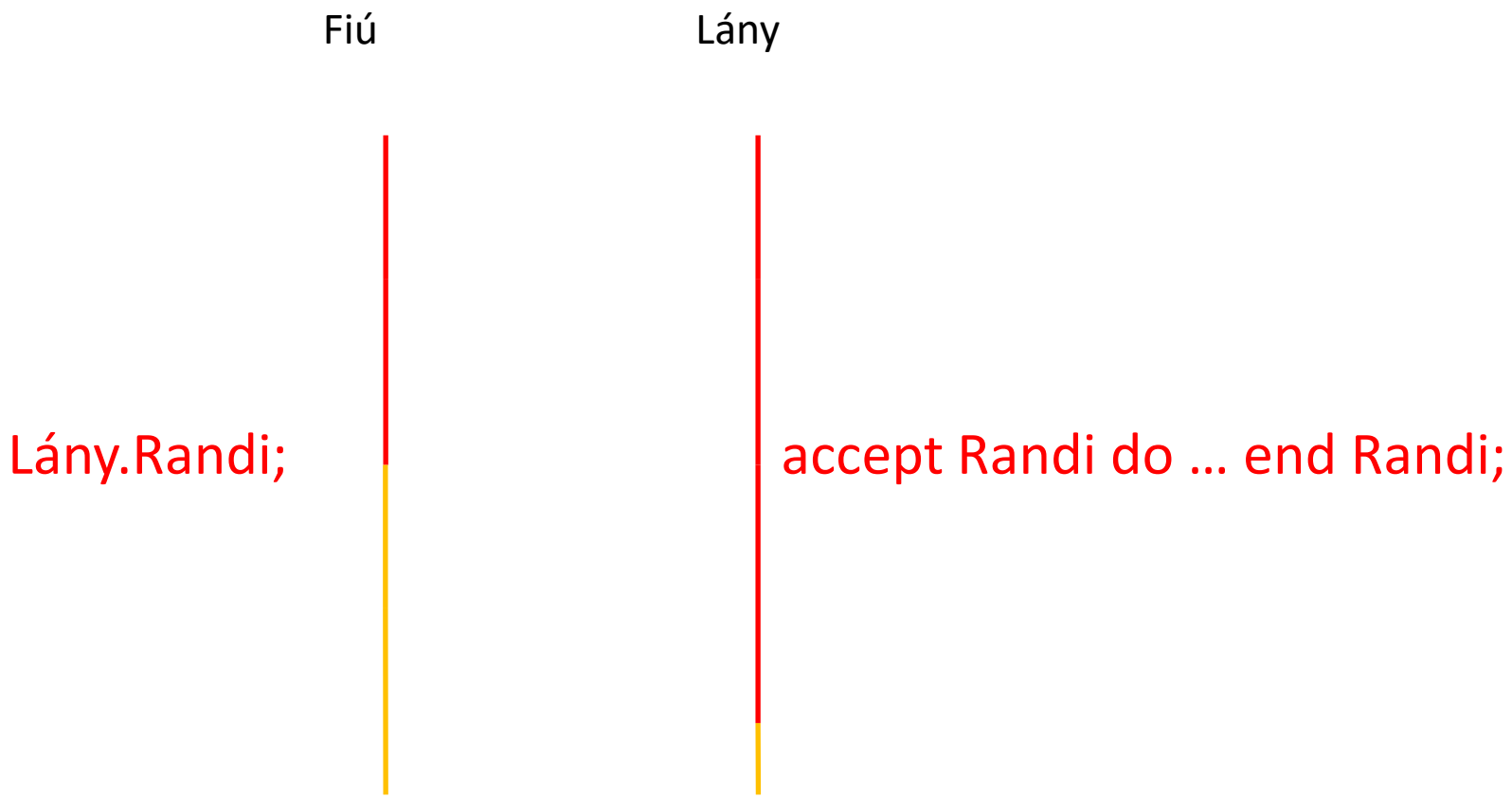
Ada – Szinkronitás



Ada – Szinkronitás



Ada – Szinkronitás



Ada – Várakozás

Egy időre felfüggeszthető a taszk a delay utasítással:

- `delay 2.4;`

A delay után valós típusú érték van, a másodperc megfelelője

Legalább ennyi ideig nem fut a taszk

Szimulációkban gyakran használjuk

Használjuk szinkronizációs utasításokban is

Ada

```
procedure P is
          task T;
          task body T is … begin  … end T;
begin
          -- Itt a T is indul
end P;
-- P bevárja T-t
```

Ada

Szülő egység

- Az a taszk / alprogram / könyvtári csomag / blokk, amelyben deklaráltuk
- Elindulás: a szülő deklarációs részének kiértékelése után, a szülő első utasítása előtt
- A szülő nem ér véget, amíg a gyerek véget nem ér
  - Függőségi kapcsolatok

Befejezés

- Bonyolult szabályok
  - Például
    - elfogytak az utasításai (akár kivétel miatt)
    - és a tőle függő taszkok már terminálták
- Fogalmak: komplett, abortált, terminált
- `T'Callable T'Terminated`

Ada – Több szál

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Haromszalu is
  task Egyik;
  task body Egyik is
  begin
    for i in 1..1000 loop Put_Line("Egyik!"); end loop;
  end Egyik;

  task Masik;
  task body Masik is
  begin
  for i in 1..1000 loop Put_Line("Másik!"); end loop;
  end Masik;

begin
  for i in 1..1000 loop Put_Line("Viszlát!"); end loop;
end Haromszalu;
```

Ada – Taszk típussal

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Haromszalu is

   task type Udv;
   task body Udv is
   begin
     loop Put_Line("Szia!");end loop;
   end Udv;
   Egyik, Masik: Udv;

begin
   loop Put_Line("Viszlát!"); end loop;
end Haromszalu;
```

Ada – Taszk típus

Taszkok létrehozásához

- Ugyanazt csinálják, ugyanolyan utasítássorozatot hajtanak végre
- Saját példánnyal rendelkeznek a lokális változókból

Korlátozott (limited) típusok

Cím szerint átadandó, ha alprogram-paraméter

Programegységek

- Mint a taszk programegységek
- Specifikáció és törzs különválik
- Nem lehet könyvtári egység
  (be kell ágyazni más programegységbe)

Ada – Taszk típus

```
procedure Haromszal is
  task type Szalak(I : Integer); -- a szál azonosítója
  task body Szalak is
  begin
      for j in 1..1000 loop
              put(I); new_line;
      end loop;
  end Szalak;
  X:Szalak(1);
  Y:Szalak(2);
begin
   for i in 1..100 loop
         put_line("foprogram");
   end loop;
end Haromszal;
```

Ada – Szinkron kommunikáció

Hívott (szolgáltató) részéről

- Meddig várjon?
- Több igényt is kiszolgálhat
- Feltétel lehetősége jó lenne
- Mindörökké fusson?

Hívó részéről

- Meddig várjon?

Igény a programnyelvi támogatásra!

Ada – Szinkron kommunikáció

A select utasítás

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

Ada – 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 hívást feldolgozza

Ada – Többműveletes monitor

```
task body Monitor is
   Adat: Tipus; ...
begin
   loop
    select
         accept Muvelet_1( ... ) do ... end;
    or
         accept Muvelet_2( ... ) do ... end;
    ...
    end select;
  end loop;
end;
```

Ada – 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, akkor leállhat:

- 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

Ada – terminate

```
task body Monitor is
   Adat: Tipus; ...
begin
   loop
     select
           accept Muvelet_1( ... ) do ... end;
     or
           accept Muvelet_2( ... ) do ... end;
     ...
     or
           terminate;
     end select;
  end loop;
end;
```

Ada

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 Feltetel => accept E2 do ... end;
...
end select;
```

Ada – feltételes fogadás

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

Ada – időhöz kötött fogadás

Ha nem vár rám hívó egyik nyitott ágon sem, és nem is érkezik hívó egy megadott időkorláton belül, akkor továbblép.

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

Ada – időhöz kötött fogadás

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

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

Ada – 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)

Ada

```
select
     accept E1;
or
     when Feltetel => accept E2 do ... end;
else
     Csinaljunk_Valami_Mast;
end select;
```

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

Ada

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

Nem ugyanaz, mint az időhöz kötött hívásfogadás!

Ada – busy waiting

```
loop
      select
            accept E1;
      else
            null;
            -- busy-waiting
      end select;
end loop;
```

Amíg a szükséges helyzet elő nem áll, a folyamat dinamikusan várakozik

- Folyamatosan vizsgálja, hogy a kívánt feltételek teljesülnek-e.
- Hátránya, hogy a folyamat várakozás közben foglalja a processzoridőt.

Ada – Termelő-fogyasztó

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

Ada – Termelő-fogyasztó

```
task type Termelo ( Sor: Sor_Access );
task body Termelo is
  Adat : Tipus;
begin
  loop
    Eloallit( Adat ); Betesz( Sor.all, Adat );
  end loop;
end Termelo;

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

Ada – Taszkok elrejtése

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

Megvalósítás: monitorral

Ada – Taszkok elrejtése

```
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;

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

Ada – Taszkok elrejtése

```
loop
   select
      when Üres(S) => accept Betesz( E: in Elem ) do
                                 Betesz(S,E);
                              end Betesz;
   or
      when Tele(S) => accept Kivesz( E: out Elem ) do
                                 Kivesz(S,E);
                              end Kivesz;
   or
      terminate;
   end select;
end loop;
```

Ada – védett, korlátos pufferrel

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

protected body Korlatos_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 Korlatos_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.

Algol-68

```
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.

- Az osztály `Execute()` metódusát kell felüldefiniálni.
  - (virtuális) absztrakt művelet
  - Ez tartalmazza a szál fő kódját.
- Create konstruktor meghívásával hozható létre
  - Egy Boolean paramétere: (CreateSuspended)
    - Ha igaz, akkor felfüggesztve indul, különben azonnal

Szinkronizáció és kommunikáció

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

Delphi

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

thread:=TMyThread.Create(false);
// el is indítjuk egyben
```

Meg kell adni az Execute törzsét …

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

- Ezzel jelezi egy másik szálnak, hogy folytathatja a tevékenységét.

Delphi

```
type
  TMyThread1 = class(TThread)
    protected procedure Execute; override;
  end;
  TMyThread2 = class(TThread)
    protected procedure Execute; override;
  end;
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.
- Runnable interfész megvalósítása kell, ha nem tud örökölni a Threadtől!

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.

Java

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

Java

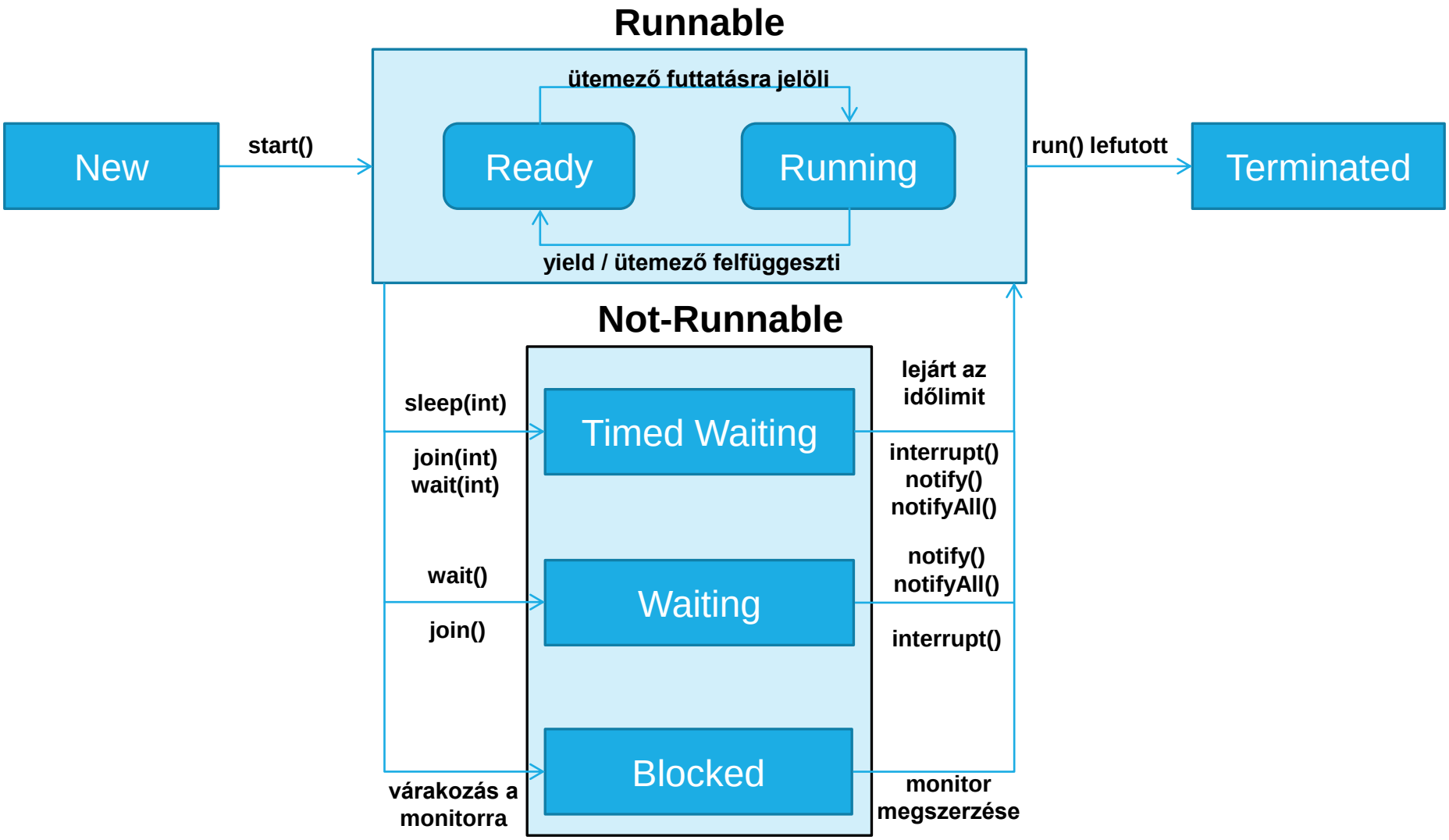
```
class MyThread implements Runnable {
  public void run() {
   System.out.println(Thread.currentThread().getName);
  }
 }
public class RunnableThread {
 static public void main(String s[]) {
   MyThread work2do;
   Thread a, b;
   work2do = new MyThread();
   a = new Thread(work2do);
   b = new Thread(work2do);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

Java

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



Java

Szálcsoportok is létrehozhatók

- A ThreadGroup segítségével csoportszinten is kezelhetők a szálak

Futásvezérlés

- Felfüggesztés – újraindítás
  - suspend() - resume()
    - Nem ajánlják – holtpontveszély
      - pl. egy szál zárol egy erőforrást, és felfüggesztik …
    - Javaslat: segédváltozó és a wait() – notify()

Java

- Megszakítás
  - interrupt()
- 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

Java

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

Java – synchronized

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

Java – synchronized

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…
- Létjogosultság: ha nem egy objektumban vannak azok az adatok, amelyekhez szerializált hozzáférést akarunk garantálni

Java – 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

Java – wait-notify

Minden objektumhoz tartozik a sima kulcshoz tartozó várakozási soron kívül egy másik 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(); ... }`

Java – wait-notify

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.
- 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.

C#

Szálak vezérlése

- `Abort()`
  - Az aktuális szál befejezése, valójában a szálban egy AbortThreadException kivétel dobását okozza
  - Ezzel befejeződik a szál végrehajtása.
  - Ha egy szálat abortáltunk, nem tudjuk a Start függvénnyel újraindítani
    - `ThreadStateException` kivételt dob.
    - Ezt a kivételt elkaphatjuk, sőt visszavonható az abortálás a `Thread.ResetAbort()` függvényhívással
  - IsAlive: tulajdonság, megmondja, hogy a szál élő-e
- `Suspend()`
  - A szál végrehajtásának felfüggesztése
- `Resume()`
  - A felfüggesztés befejezése, a szál továbbindul

C#

```
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#

```
class program {
  public static adatok a=new adatok();

  public static void szal1() {
    Console.WriteLine("Ez a szál1 program indulás!");
    Console.WriteLine("Adatmentés meghívása szál1-ből!");
    a.mentes("+");
    Console.WriteLine("Szál1 vége!");
  }

  public static void szal2() {
    Console.WriteLine("Ez a szál2 programindulás!");
    Console.WriteLine("Adatmentés meghívása szal2-ből!");
    a.mentes("-");
    Console.WriteLine("Szál2 vége!");
  }
```

C#

```
public static void Main() {
  Console.WriteLine("A főprogram itt indul!");
  ThreadStart szalmutato1 = new ThreadStart(program.szal1);
  ThreadStart szalmutato2 = new ThreadStart(program.szal2);
  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#

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.
    - Statikus adattagokat nem lehet szinkrozinálni
  - 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#

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.

C#

Monitor

- Amikor egy szál a lefoglalt kódrészletében `Wait` -et hív
  - Megszakad a hozzáférése, és bekerül egy várakozó sorba
  - A sor első elemét reprezentáló szál meghívja a `Pulse` vagy `PulseAll` metódust és 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.
- Külső objektum zára deadlock-hoz vezethet, ezért érdemes belső objektumon használni

Szinkronizáció:

1. A `Synchronization()` attribútummal kell jelölni az osztályt.
2. Az osztályt a `ContextBoundObject` osztályból kell származtatni.

C# – Példák

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

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);
}
```

C# – További lehetőségek

Számos további osztály

- `WaitHandle`
- `Mutex, AutoResetEvent, ManualResetEvent`
- `InterLocked`
- `ThreadPool`
- `Timer`
- `Backgroundworker`
- …

C++

`std::thread` szabványos könyvtár

- `thread` osztály
- Szálkezelő függvények
- Kölcsönös kizárást biztosító osztályok:
  - `timed_mutex`
  - `recursive_mutex`
  - `recursive_timed_mutex`
  - `shared_timed_mutex`

C++ példa

```
void f1()
{
    cout << "Fuggvenydemo 1.\n";
}
void f2(int x)
{
    cout << "Fuggvenydemo 2. x=" << x << "\n";
}
```

C++ példa

```
thread elso(f1);
 // uj szalat indit, ami az egyik-et hivja
thread masodik(f2, 0);
 // uj szalat indit, ami a masik(0)-t hivja
cout << "Harom szal fut parhuzamosan\n";

// szalak szinkronizalasa:
elso.join();    // var, amig elso befejezodik
masodik.join(); // var, amig masodik befejezodik

cout << "elso es masodik lefutott.\n";
```

C++ példa

```
mutex mtx;     // mutex a kritikus szakaszra

void f2_mutex(int x) {
     mtx.lock();
     cout << "Fuggvenydemo 2. x=" << x << "\n";
     mtx.unlock();
}
```

Az előző indítás helyett:

- `thread masodik(f2_mutex, 0);`

C++ példa

```
#include <iostream> // std::cout
#include <thread>   // std::thread
#include <mutex>    // std::mutex
std::mutex mtx;     // mutex a kritikus szakaszhoz

void print_block (int n, char c) {
  mtx.lock();
  for (int i=0; i<n; ++i) { std::cout << c; }
  std::cout << '\n';
  mtx.unlock();
}
int main () {
  std::thread th1 (print_block, 50,'*');
  std::thread th2 (print_block, 50,'$');
  th1.join(); th2.join();
  return 0;
}
```

Go

Google nyelve

- 2007 – ben kezdték el fejleszteni
- Jelenlegi verzió: 1.14.2 (2020. május)

Tulajdonságok

- Fordított, konkurens, imperatív programozási nyelv
- C-szerű szintaxis
- Statikus (erősen) típusos nyelv
- Szemétgyűjtés

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

```
package main;
import (
       "fmt"
       "time"
)
func IsReady(what string, minutes int64) {
    time.Sleep(time.Duration(minutes));
    fmt.Println(what, "is ready")
}
func main() {
    go IsReady("tea", 4);
    go IsReady("coffee", 2);
       // nem var a befejezesre, ha o
       // befejezte, akkor kesz, ezert kell ide is egy Sleep...
    time.Sleep(time.Duration(6));
    fmt.Println("I'm ready...");
}
```

Go

A Go a párhuzamosságot osztott memória helyett adatcsatornákon történő üzenetküldésekkel támogatja.

- `chan // kétirányú csatorna`
- `<-chan // fogadó csatorna`
- `chan <- // küldő csatorna`

Az elemek típusa tetszőleges

Az inicializálatlan csatorna értéke nil

Új csatornát a make() függvénnyel hozhatunk létre

- `make(típus, kapacitás)`
- `make(chan int, 100)`

Csatorna típusnak kell lennie

Ha a buffer méret 0, akkor blokkol, amíg a fogadó goroutine nem áll készen fogadni, egyébként aszinkron

PVM – Parallel Virtual Machine

Szoftver rendszer, mellyel hálózatba kapcsolt számítógépeket egyetlen nagy párhuzamos számítógépként lehet látni és kezelni.

- Kezdet: 1989 - Oak Ridge National Laboratory
- A párhuzamos programok írásának egyik szabványává vált.
- A publikus változata ingyen elérhető.
- Sok hardver gyártó biztosítja a saját gépére optimalizált, gyorsabb változatát is.

PVM

PVM rendszerbe kapcsolhatunk több számítógépet

- Akár különböző típusúakat is

Tulajdonságok

- A rendszer a rajta futó programok szempontjából ezek után egy nagy, elosztott memóriájú virtuális számítógépnek látszik.
- Különböző processzorokon, különböző programokat indíthatunk el.
- A szinkronizációt és a kommunikációt a PVM könyvtári függvényeivel oldhatjuk meg.

PVM

Működése

- Számítógépek hálózatba szervezve
- A felhasználó jogosult bármelyik gépre bejelentkezni
- Minden gépen futtat egy pvm „démon”-t
- És futtatja a taszkokat, melyek a démonokon keresztül lépnek egymással kapcsolatba
  - Egy gépen több taszk is futhat

PVM

Egy nagy virtuális címtér

- PVM
- Partitioned Global Address Space (PGAS) nyelvek és rendszerek

Üzenetküldés, elosztott memória

- Communicating Sequential Processes (CSP)
- Egymás memóriáját nem érik el, csak üzenetekkel kommunikálhatnak
- Message Passing Interface standard – legelterjedtebb
- Több millió folyamat (process) kezelésére alkalmas

Helyesség

KÖVETKEZŐ ALKALOMMAL