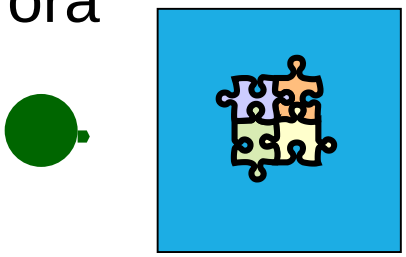
Programozási nyelvek és módszerek

10. ELŐADÁS – PÁRHUZAMOSSÁG

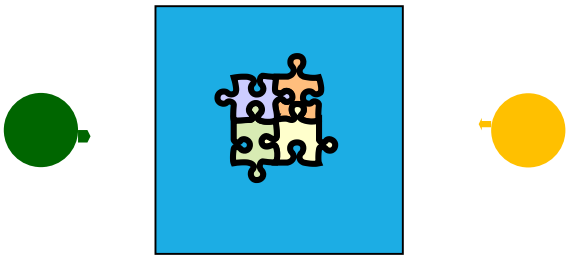
Párhuzamosság

Puzzle analógia – Henry Neeman @ Oklahoma University

Soros programozás: egyedül akarod megcsinálni. 1000 darabos puzzle, kb. 1 óra

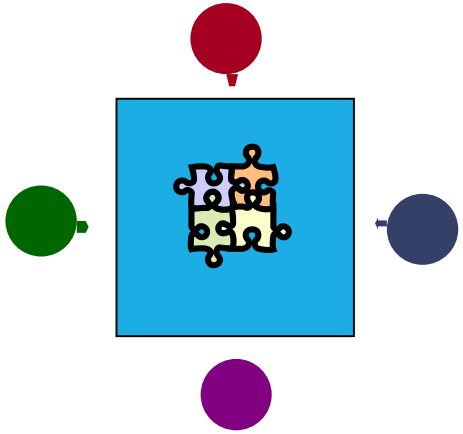


Tegyük fel, hogy egy barátod leül melléd és segít kiratkni: most már viszont kommunikálnotok is kell, és néha ugyanazon a részen dolgoztok, és zavarjátok egymást. 35 perc alatt oldjátok meg, 30 helyett



Párhuzamosság

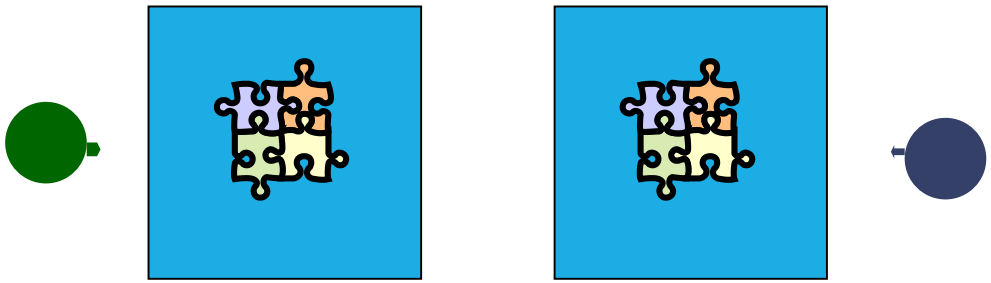
Minél többen annál jobb? Ha már négyen vagytok, 20 perc kell (15 helyett), mert egyre többet kell koordinálni egymással



Csökkenő hozadék törvénye: ahogy egyre többen próbáltok 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, de a végén össze kell illeszteni a két felet

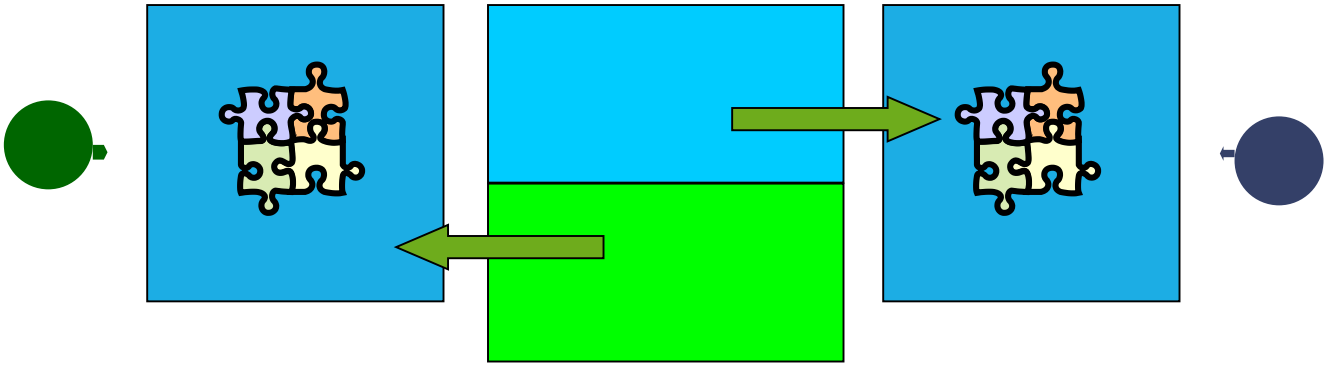


Szét is kell osztani a darabokat az elején – attól függően hogy mennyi időd töltesz egy jó elosztással, több vagy kevesebb kommunikációra van szükség a végén az összeillesztésekor

- 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

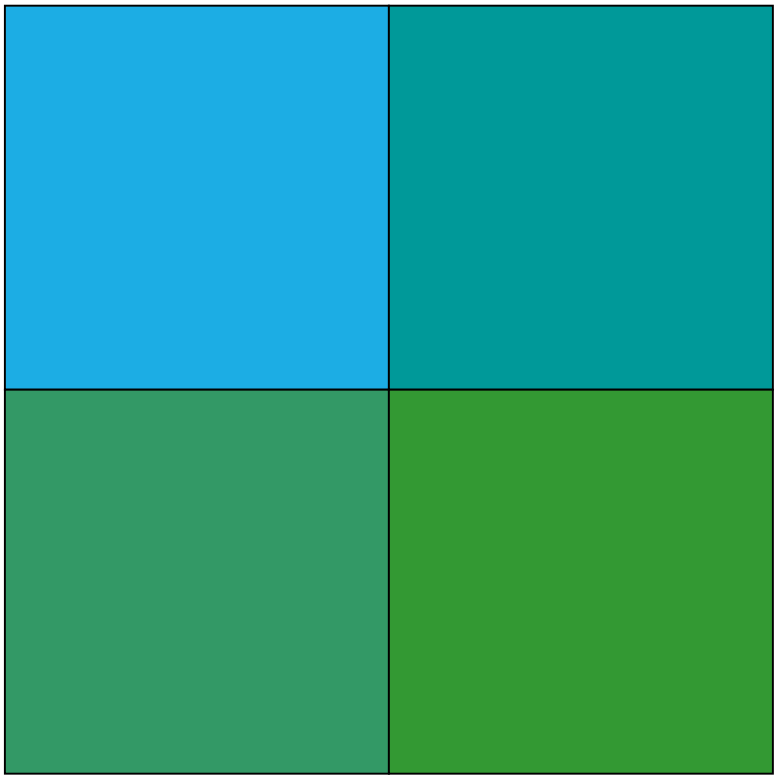
A kezdeti elosztás határozza meg a két fél kiegyensúlyosottságát is. Ha a kép fele ég a másik föld, könnyű a leosztás, a végén csak a perem mentén kell “kommunikálni”

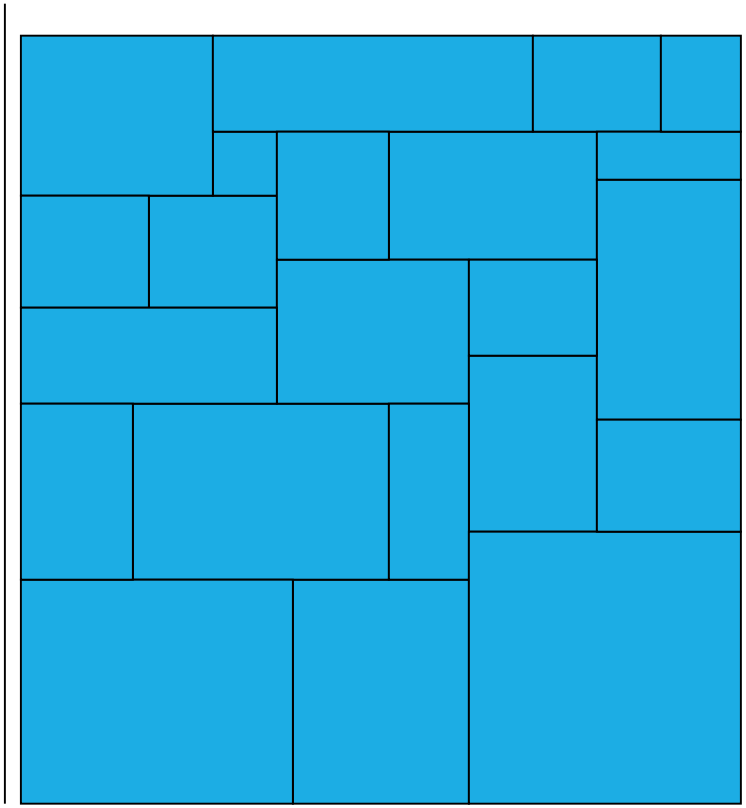


Bizonyos problémák esetén ez a felosztás könyebb mint más problémákon

Elosztott párhuzamosság

A munka kiegyensúlyozása nehéz, főleg ha Jancsi gyorsabb mint Juliska...





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)

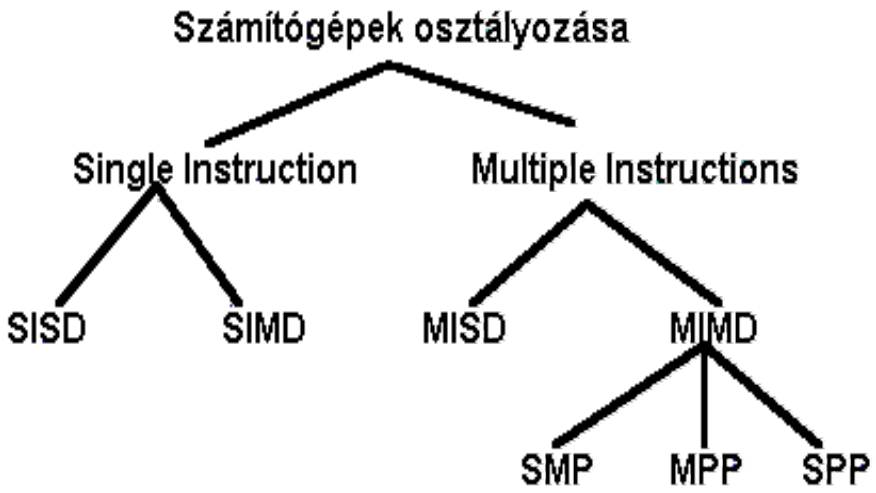
A processzor(ok) a vezérlőtől kapják a végrehajtandó utasítássorozatot (instruction stream)

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

Flynn-féle modell

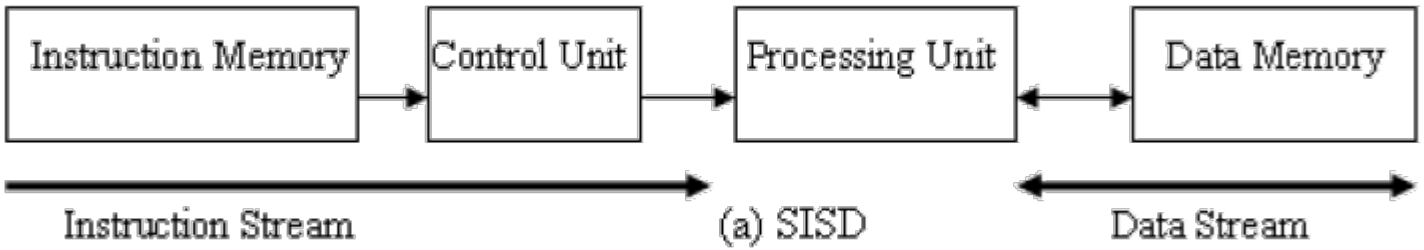
Csoportok - aszerint, hogy hány utasításfolyam (Instruction stream), illetve adatfolyam (Data stream) különíthető el a rendszerben

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



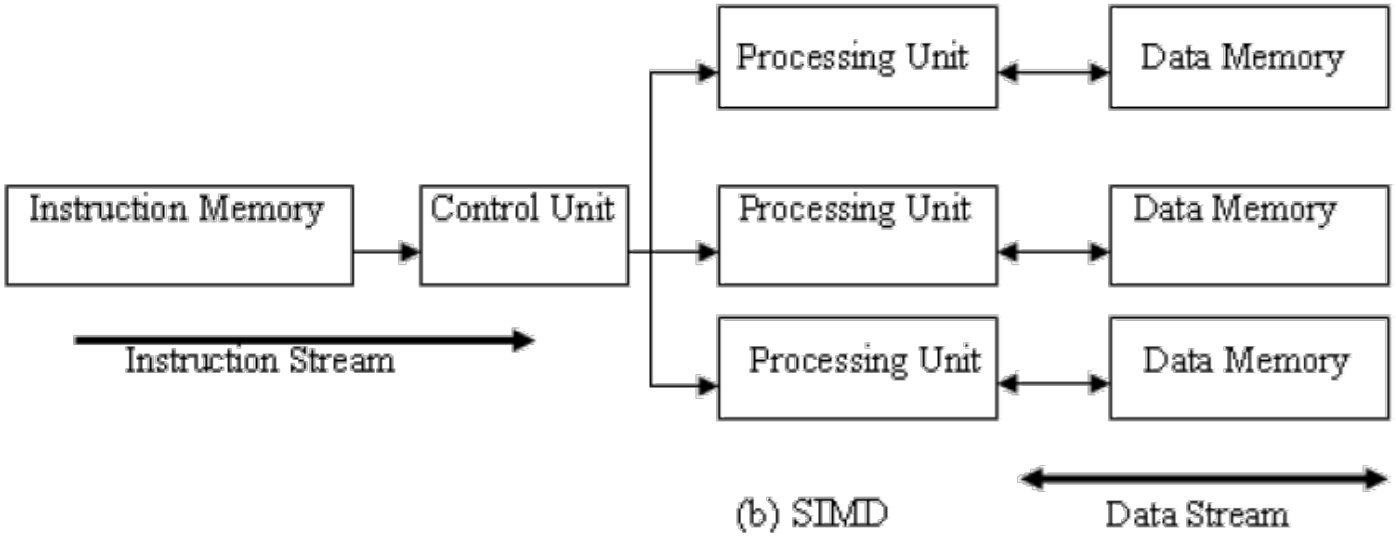
SISD (Single Instruction, Single Data)

Ezek a „hagyományos” számítógépek



SIMD (Single Instruction, Multiple Data)

Nagyon sok egyszerű processzorból álló gépek, amelyek mindegyike rendelkezik saját memóriával, de mindegyiken azonos program fut.

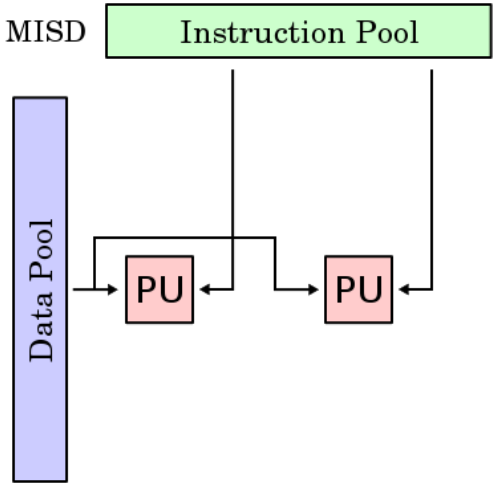


MISD (Multiple Instruction, Single Data)

Nem gyakori az ilyen gép, csak az osztályozás teljessége miatt került bele

Pipeline-szerűen lehet elképzelni

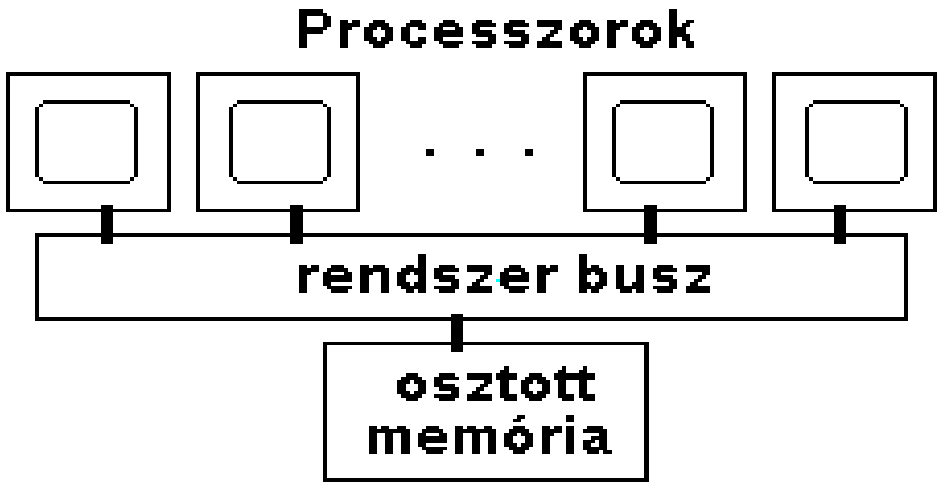
Hibatűrő rendszerekhez pl. 2 a 3-ból logikához



MIMD (Multiple Instruction, Multiple Data)

A legfontosabb csoport – ezen belül

- SMP (Symmetric multiprocessing), vagy 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.

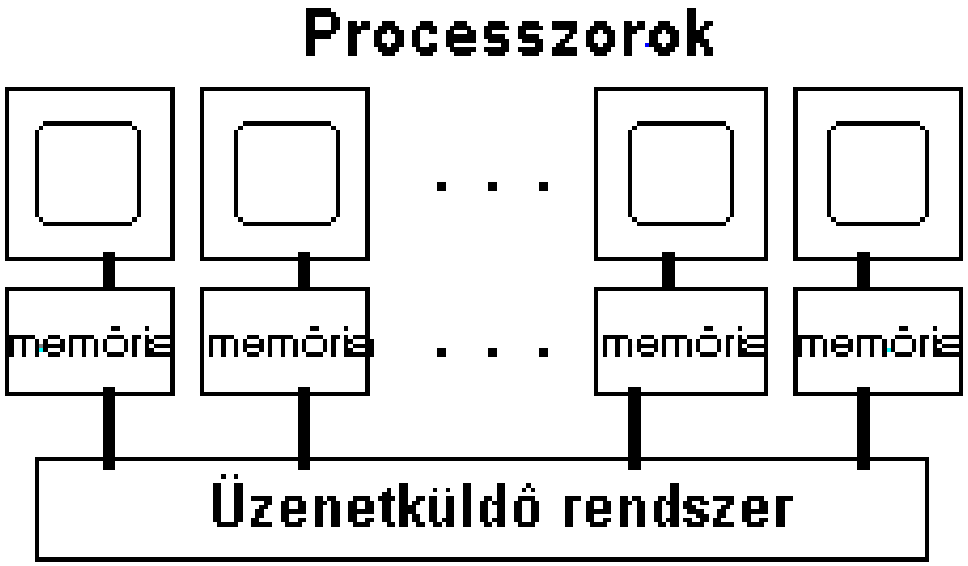
Hátránya

- 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

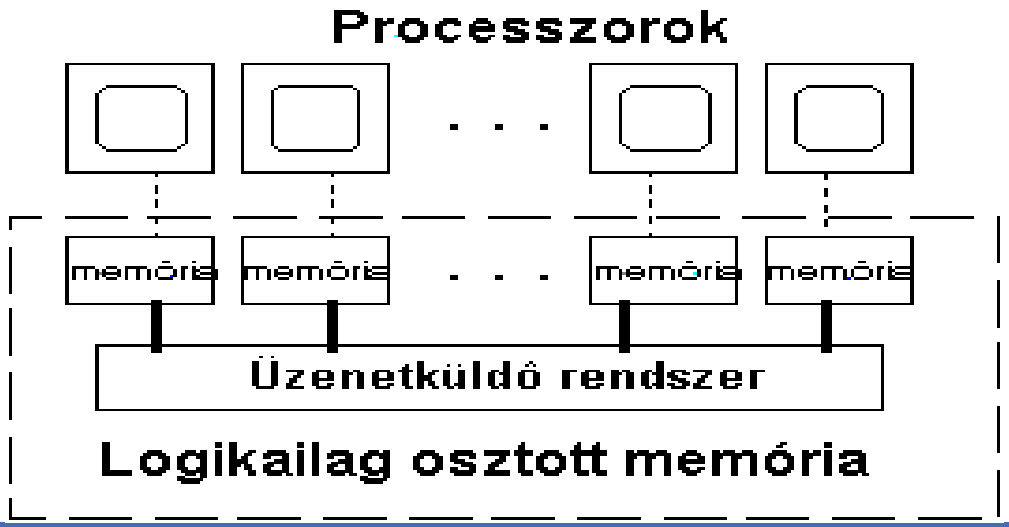


SPP (Scalable parallel processing)

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 (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
  - 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.

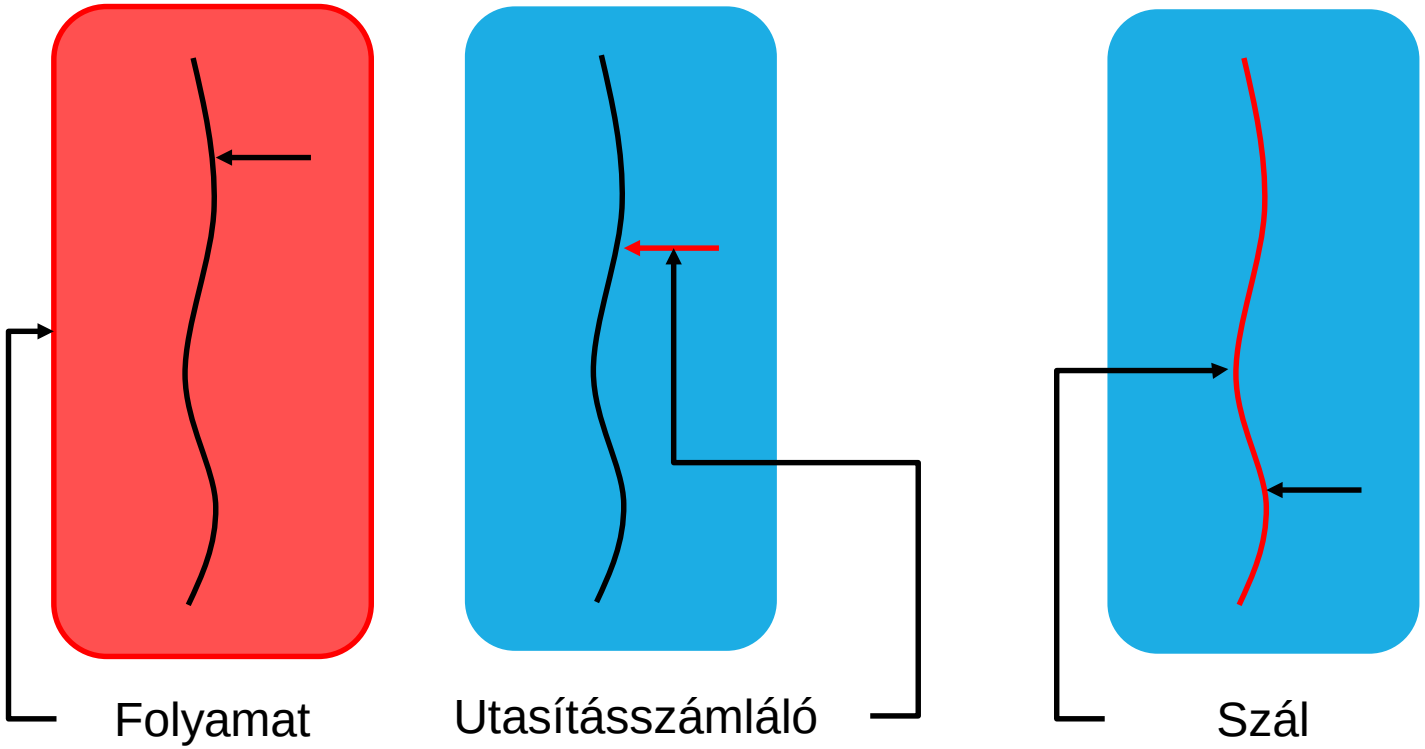
Egy egyprocesszoros környezetben a CPU több folyamat között van megosztva, és egy ütemező algoritmus szabályozza, hogy mikor kell megállítani egy folyamat feldolgozását, hogy egy másik folyamatot szolgálhasson ki.

Folyamatok (processzek)

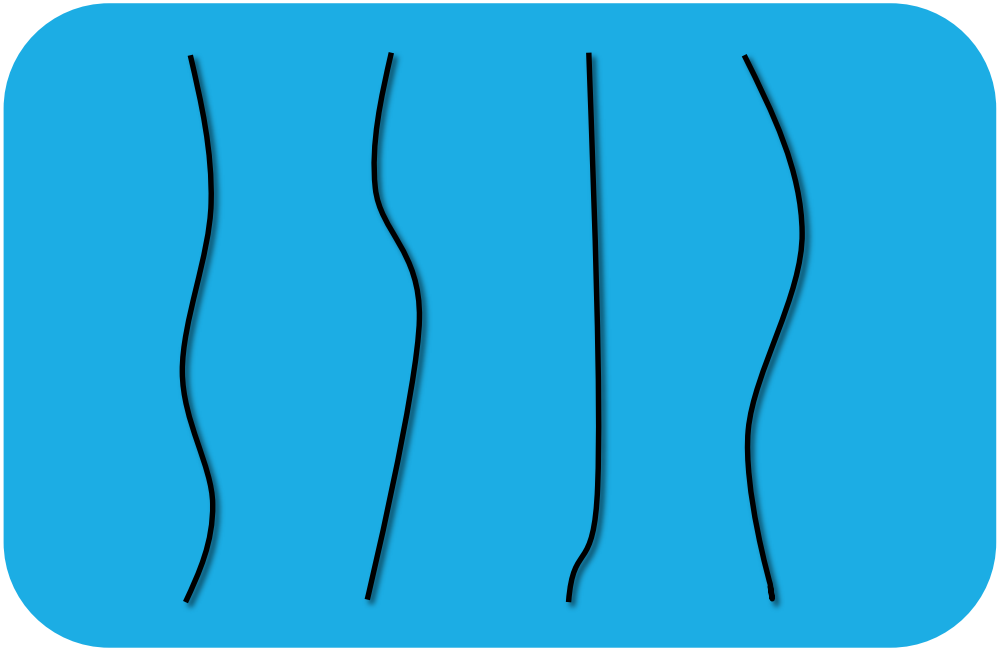
A blokkolt állapotban a folyamat nem futhat tovább, még akkor sem, ha a CPU-nak nincs más feladata.

- 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 folyamat több szállal

Egy szál koncepcionálisan hasonlít egy folyamathoz

A különbség

- A folyamat (processz) 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 szál felhasználói szintű fogalom

- 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…

- Történhet
  - 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 a 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.

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 (szekvenciálisan) hajthatók végre, pl. 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;
```

DE: pl. u3 és u4 egyszerre, egymással párhuzamosan is végrehajtható, hiszen nem függenek egymástól.

Precedenciagráf

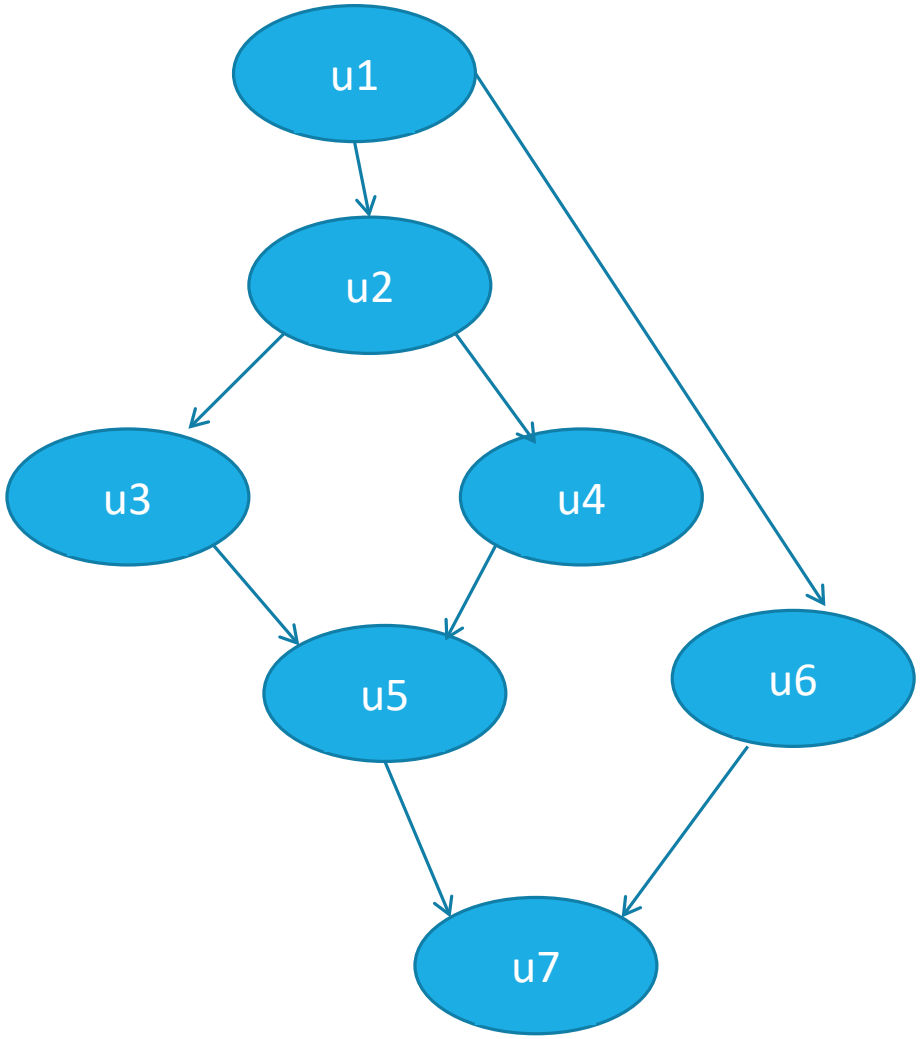
A lehetséges párhuzamosítást mutatja

Csomópontok: UTASÍTÁSOK

Élek: Abban a csomópontban lévő utasítás, ahonnan egy él indul, MEG KELL, HOGY ELŐZZE azt az utasítást, amely abban a csomópontban található, ahová az él mutat.

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!

A precedenciagráfok szemléletesen tükrözik a párhuzamosítási lehetőségeket, de programozásra közvetlenül nem használhatók

- 1. megoldás: fork / join utasításpár
- 2. megoldás: parbegin / parend utasításpár

Fork-join pár

fork utasítás: a végrehajtás két egymással párhuzamosan végrehajtható ágra szakad

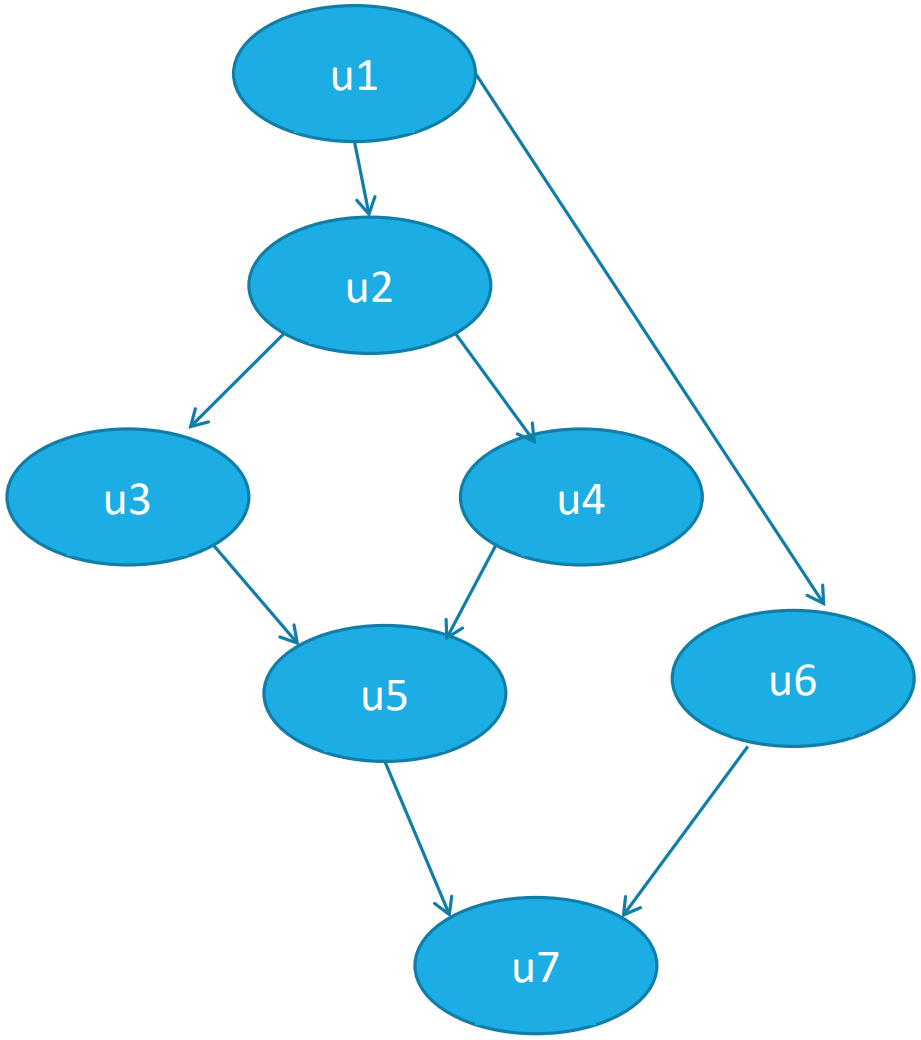
- az egyik ág közvetlenül a fork utasítás után, a másik ág a megadott címkénél található

join utasítás: 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;
```



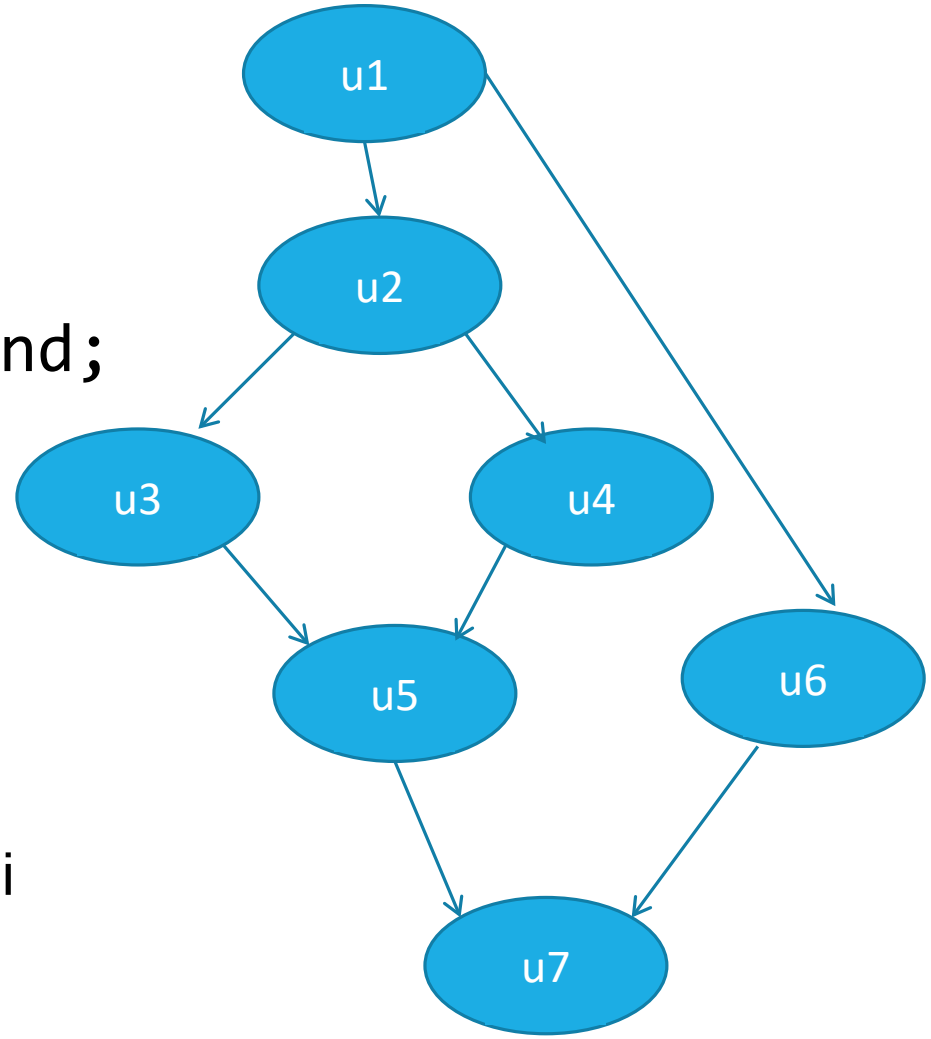
Parbegin-parend pár

```
u1;

parbegin
u6;
begin
   u2;
   parbegin u3; u4; parend;
   u5;
end;

parend;

u7;
```



Sajnos ez nem jó minden precedenciagráfra

- szemaforokkal kell kiegészíteni ilyenkor…

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

Függetlenek

```
with Text_IO; use Text_IO;
procedure Ketszal is
      task Egyik;
      task body Egyik is
      begin
            for i in 1..1000 loop
                  put("E");
            end loop;
      end Egyik;
begin
      for i in 1..1000 loop
            put("O");
      end loop;
end Ketszal;
```

Függetlenek?

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Kétszálú 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 Kétszálú;
```

Lehetséges kimenetek

Ld. 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

Jellemző kimenet

Sokszor az egyik (például Viszlát!)

Utána sokszor a másik

Nem ugyanannyiszor

Változik futás közben, hogy melyikből mennyi

Futásról futásra is változik

Ütemezés

A futtató rendszer ütemezi a szálakat

- Vagy az operációs rendszer, vagy az Ada rendszer
- Co-rutinok: programozó ütemez (Simula 67)

Például az Ada nyelv nem tesz megkötést arra, hogy egy adott Ada implementációban az ütemezés milyen algoritmussal történik

Jó program: bármely ütemezésre jó

Időosztásos ütemezés

Nem valódi párhuzamosság

Egy processzor: egyszerre csak egyet

Időszeletek (time slicing)

Gyorsan váltogatunk, párhuzamosság látszata

Egyenlő időszeletek: pártatlanság

Kooperatív viselkedés

Kompetitív viselkedés

„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

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 konkurrens 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

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, hogy 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:

- 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

```
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,
  - Adában a paraméterek módja szerint (in, out, in out)
- Az alprogramok hívására hasonlít a mechanizmus

Specifikáció

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

Törzse

```
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;
```

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 írhatja az adott erőforrást (pl. változót), ilyenkor az eredmény határozatlan érték lesz.

Hasonló probléma merülhet fel egy írási és egy olvasási művelet összeakadásánál.

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?

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; 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.

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 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;
```

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.

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 -- védi
            Text_IO.Put_Line(Str);
      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 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 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 memóriaréssel 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.

- Ezek többprocesszoros gépek programozási nyelvei.

Ide sorolhatók 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.

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

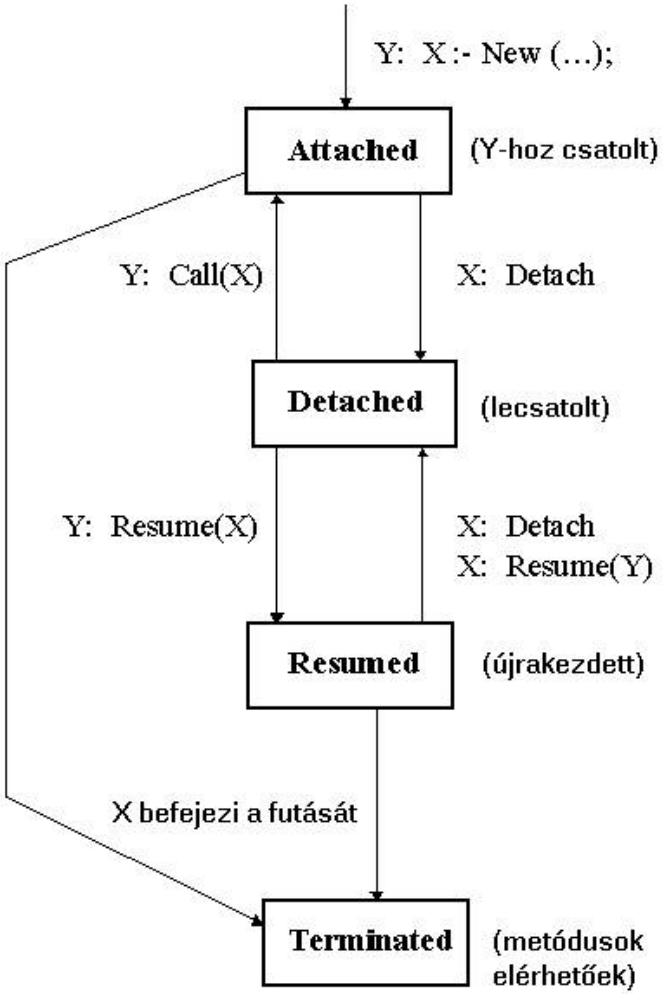
- A Simula-67 nyelvben a korutin valamilyen osztályba tartozó objektum, amely azonos vagy más osztályba tartozó objektumokkal működik együtt. A létrehozott objektum (korutin) alárendeltségi viszonyban van azzal az objektummal, ami létrehozta.
- A létrehozott objektum a detach utasítással adhatja vissza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

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.

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.

Randevú: Fiú és Lány taszkok

```
task Lány is
       entry Randi;
end Lány;

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

```
task Fiú;

task body Fiú is
begin
       …
       Lány.Randi;
       …
end Fiú;
```

Pont-pont kapcsolat

Egy randevúban mindig két taszk vesz részt

Egy fiú és egy lány

Szerepek

- Egy taszk lehet egyszer fiú, másszor lány

Nincs "broadcast" jellegű randevú

Aszimmetrikus

Megkülönböztetjük a hívót és a hívottat
(fiú és lány)

Egész másként működik a két fél

A szintaxisban is kimutatható a különbség

A hívó ismeri a hívottat, de fordítva nem

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

Huzamosabb randevú

```
task Lány is
      entry Randi;
end Lány;

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

```
task Fiú;

task body Fiú is
begin
      …
      Lány.Randi;
      …
end Fiú;
```

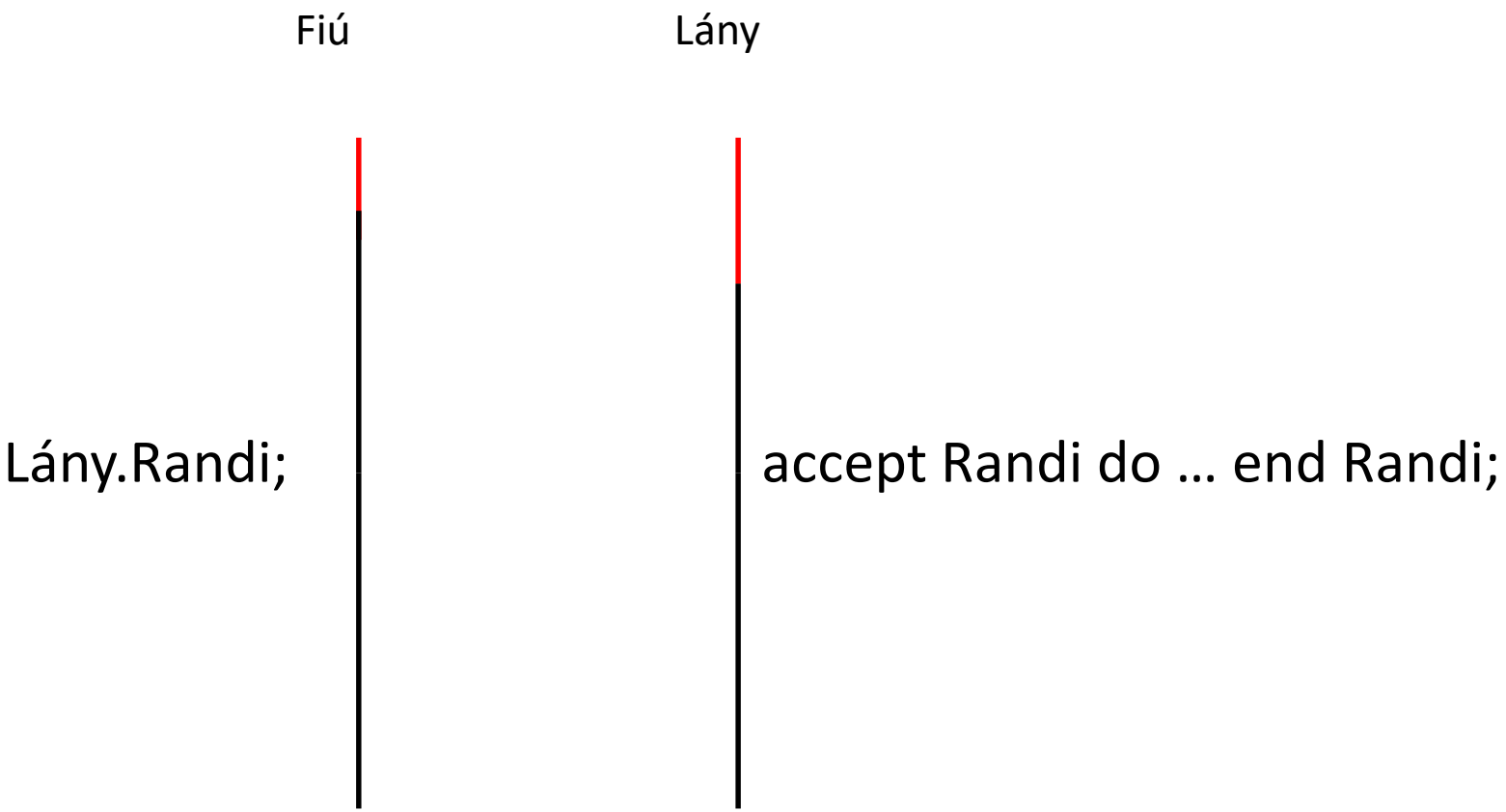
Szinkronitás

A randevú akkor jön létre, amikor mindketten akarják

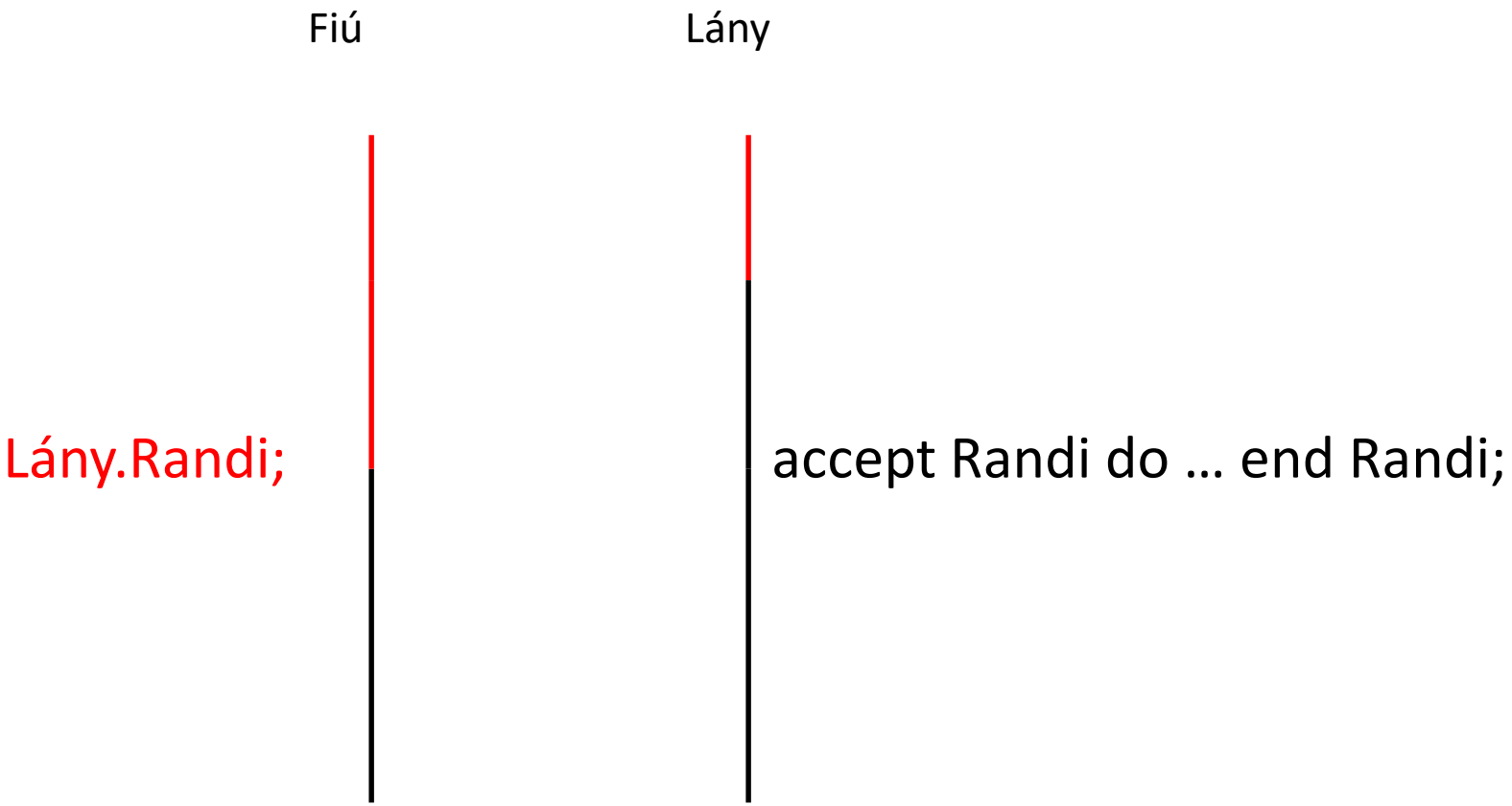
Bevárják egymást a folyamatok az információcseréhez

Az aszinkron kommunikációt le kell programozni, ha szükség van rá

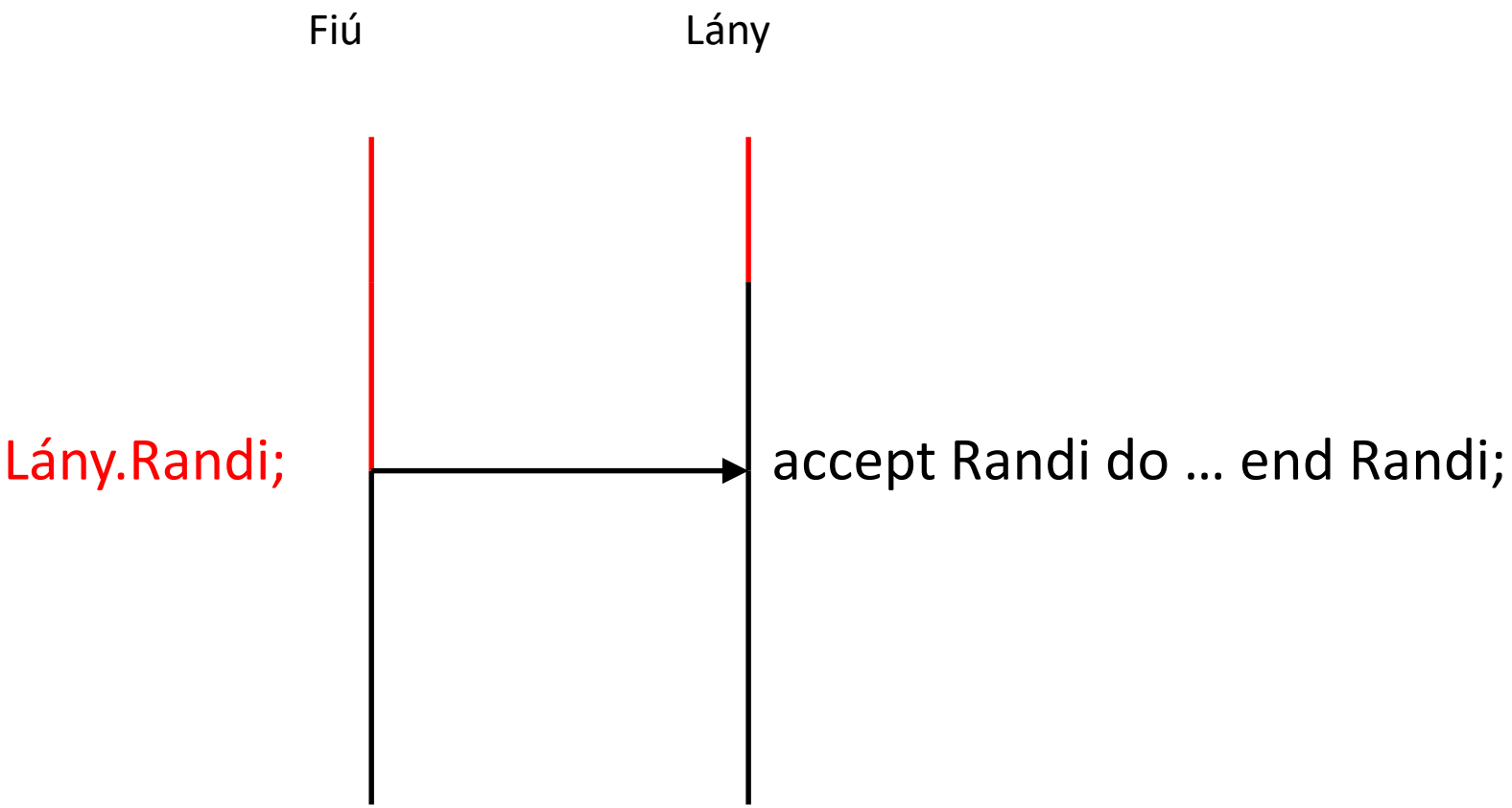
A fiú bevárja a lányt (1)



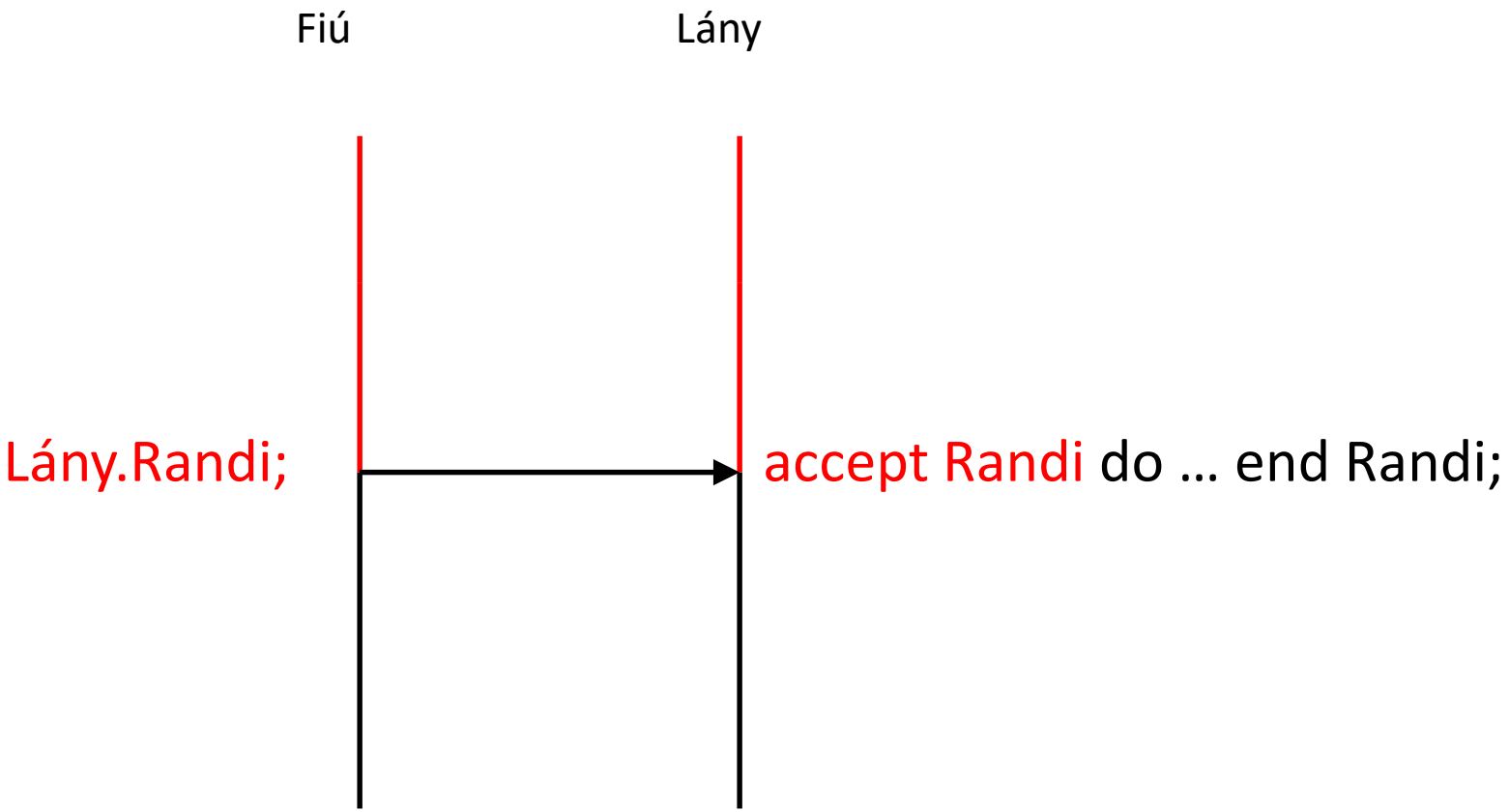
A fiú bevárja a lányt (2)



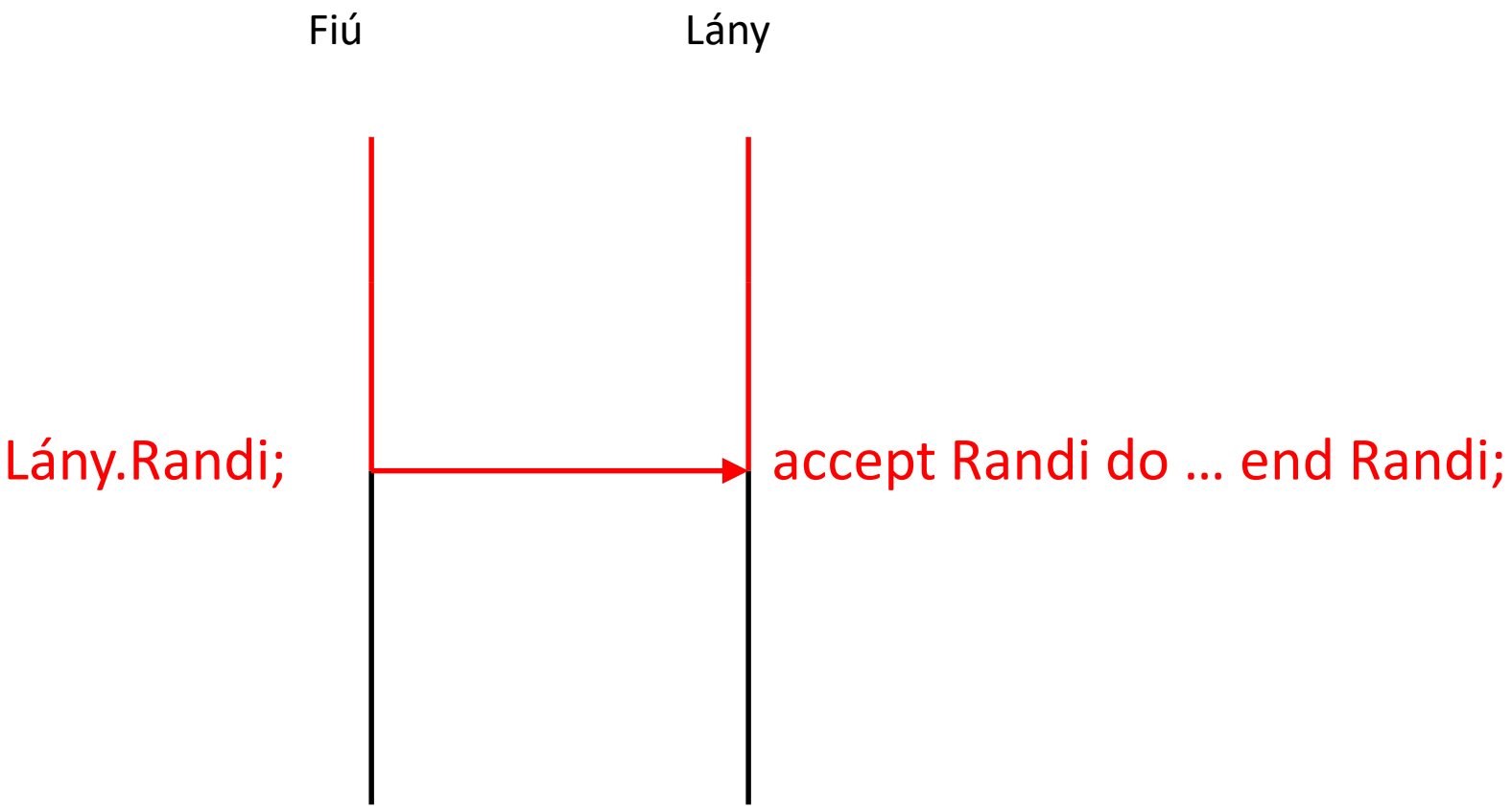
A fiú bevárja a lányt (3)



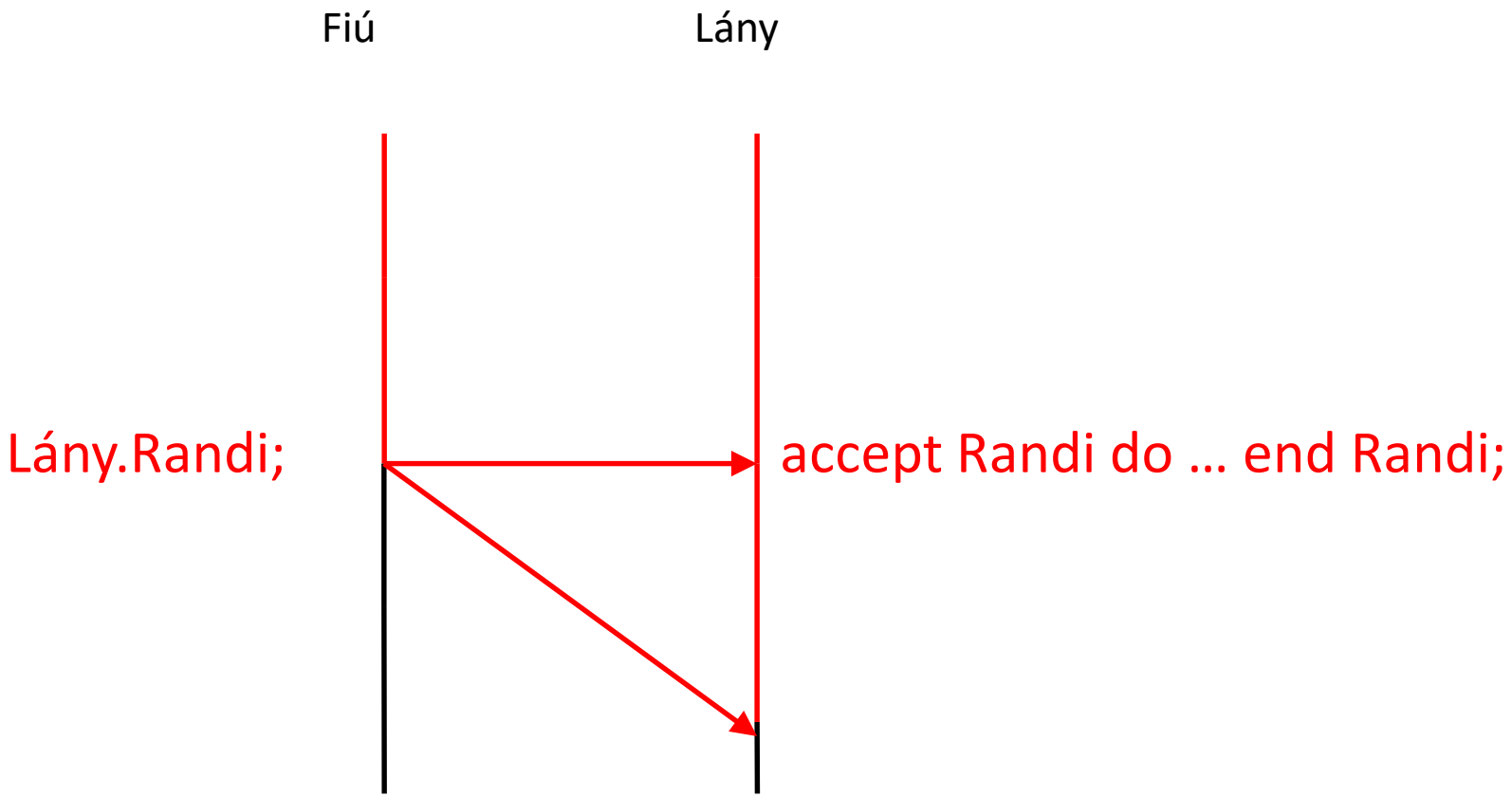
A fiú bevárja a lányt (4)



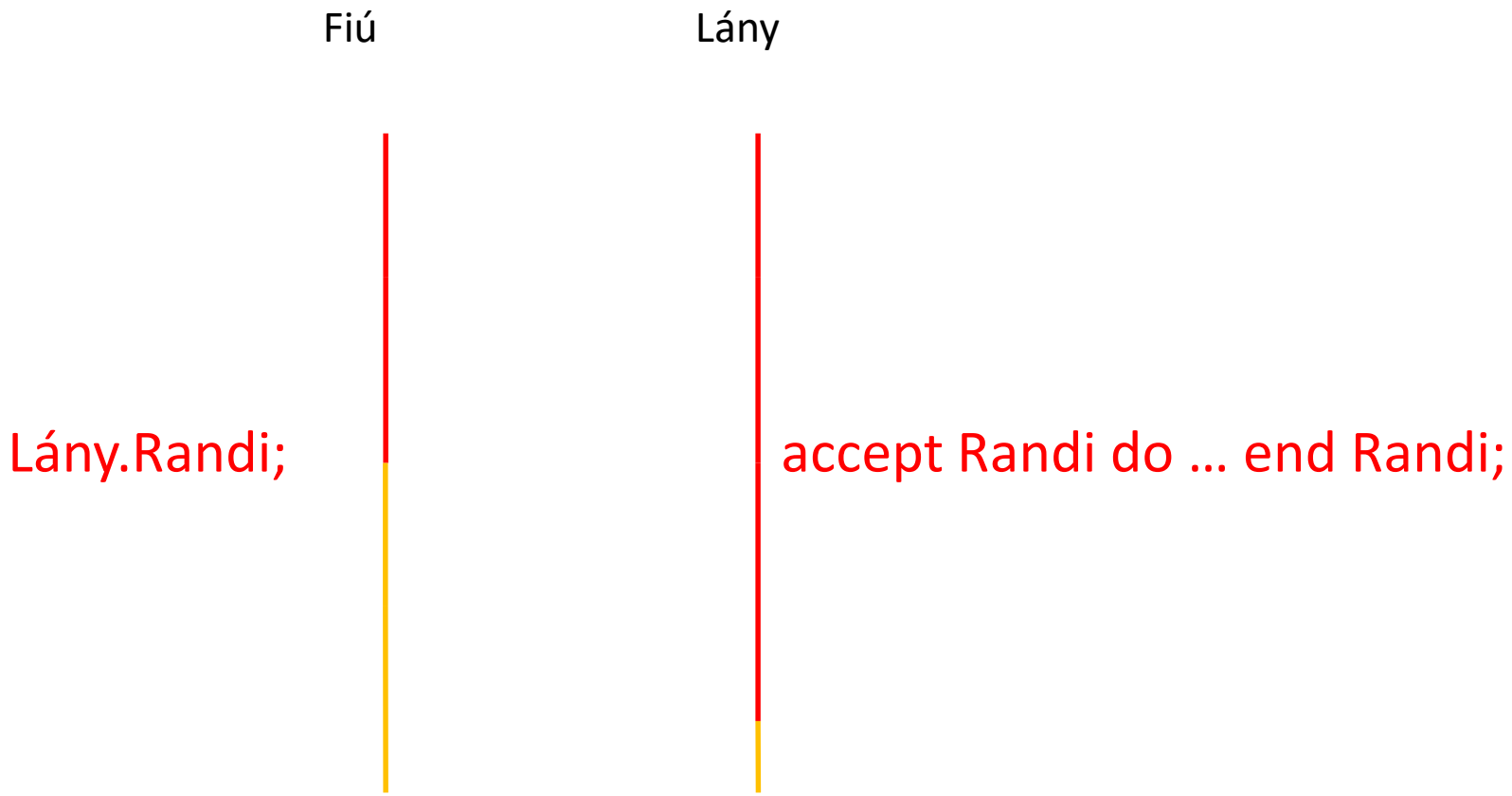
A fiú bevárja a lányt (5)



A fiú bevárja a lányt (6)



A fiú bevárja a lányt (7)

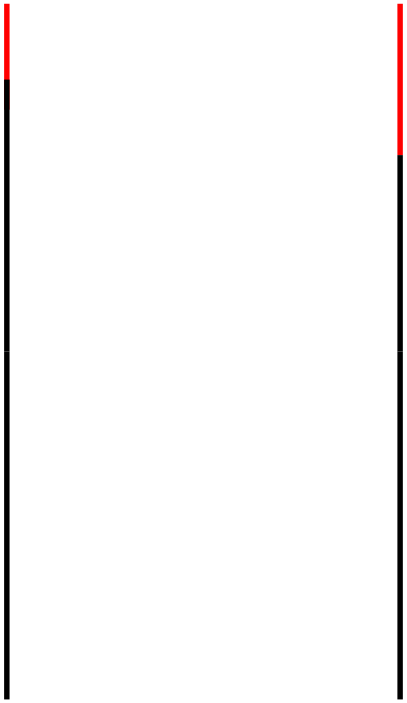


A lány bevárja a fiút (1)

Fiú

Lány

Lány.Randi;



accept Randi do … end Randi;

A lány bevárja a fiút (2)

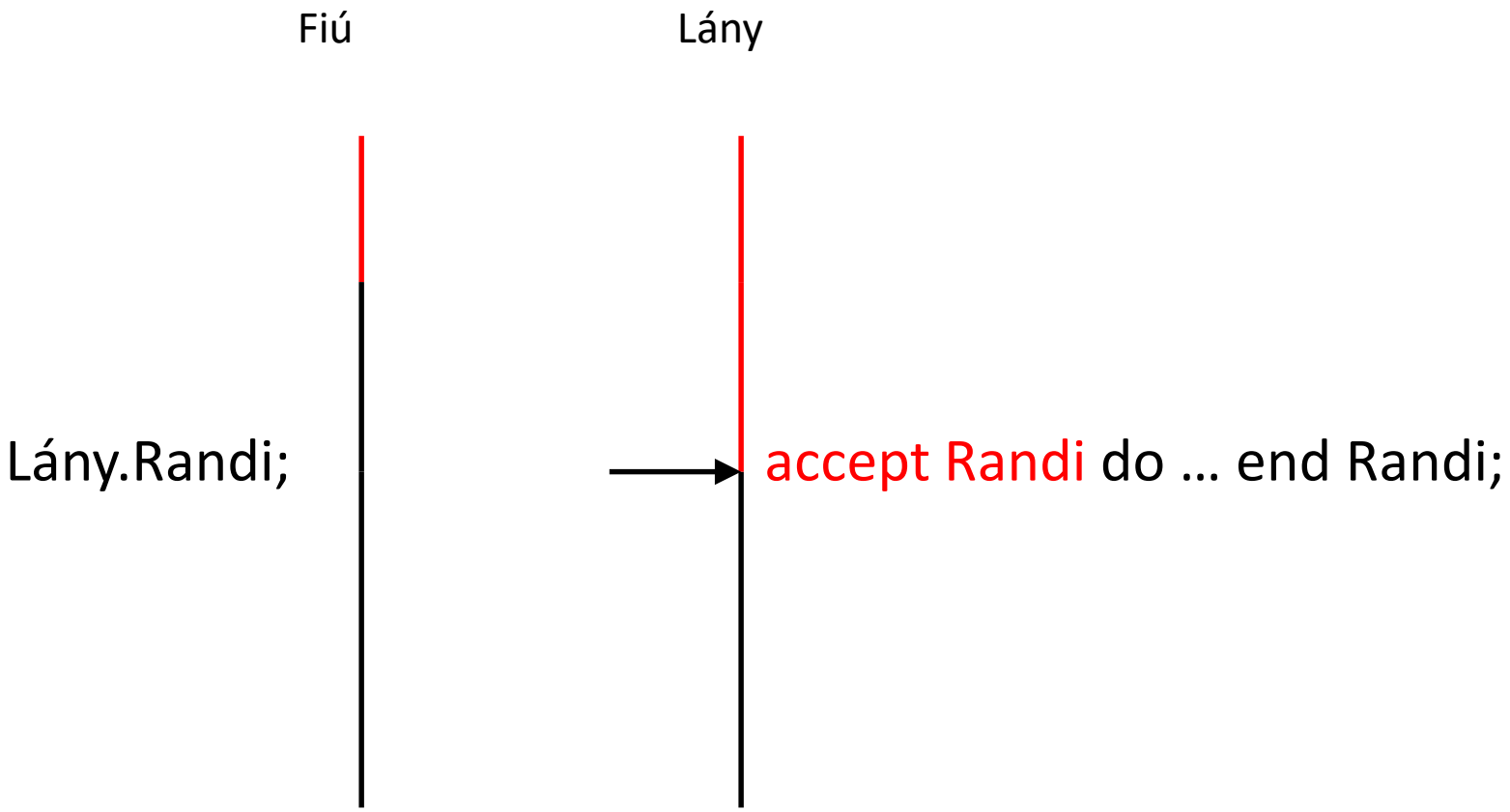
Fiú

Lány

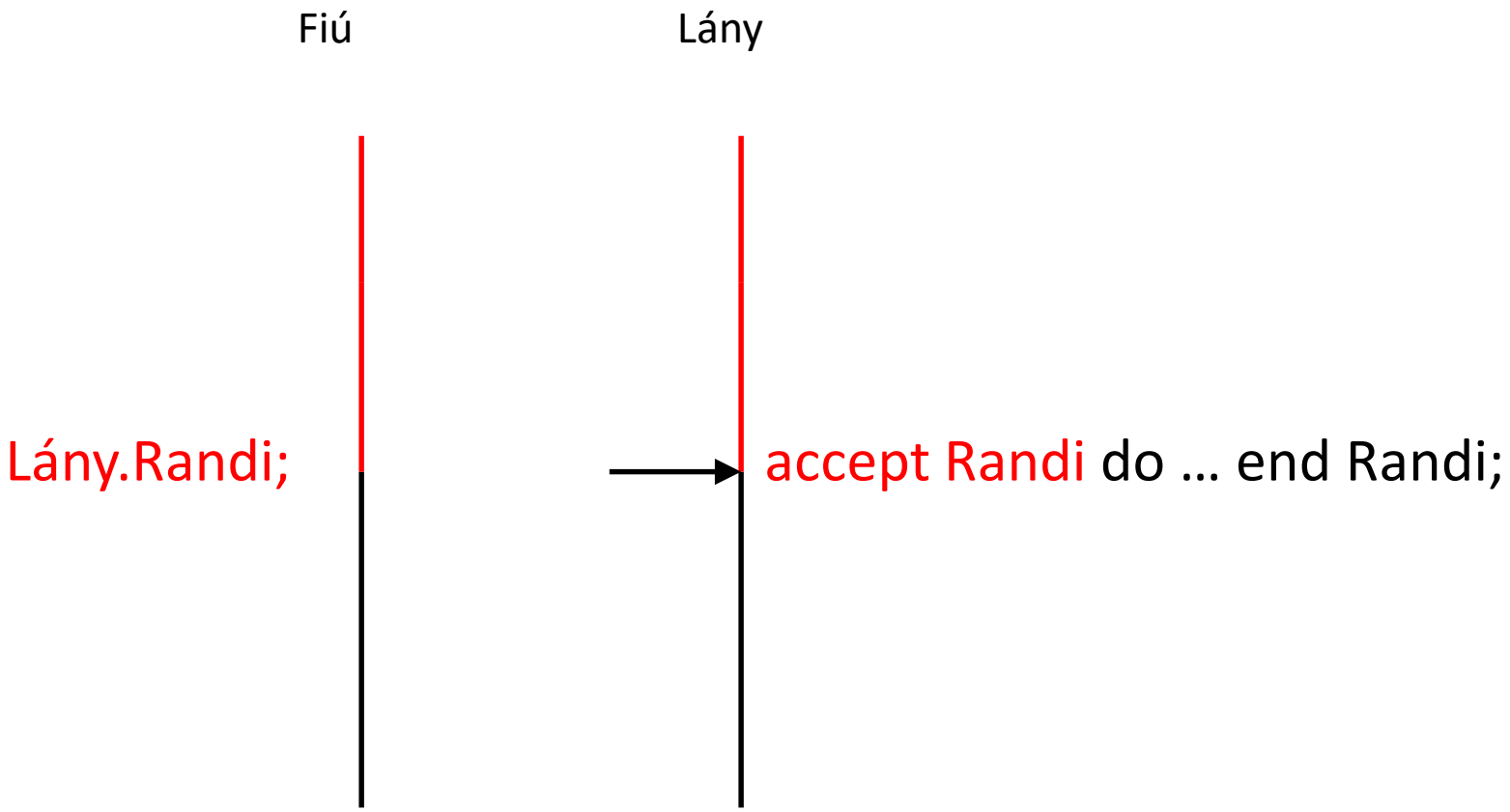
Lány.Randi;

accept Randi do … end Randi;

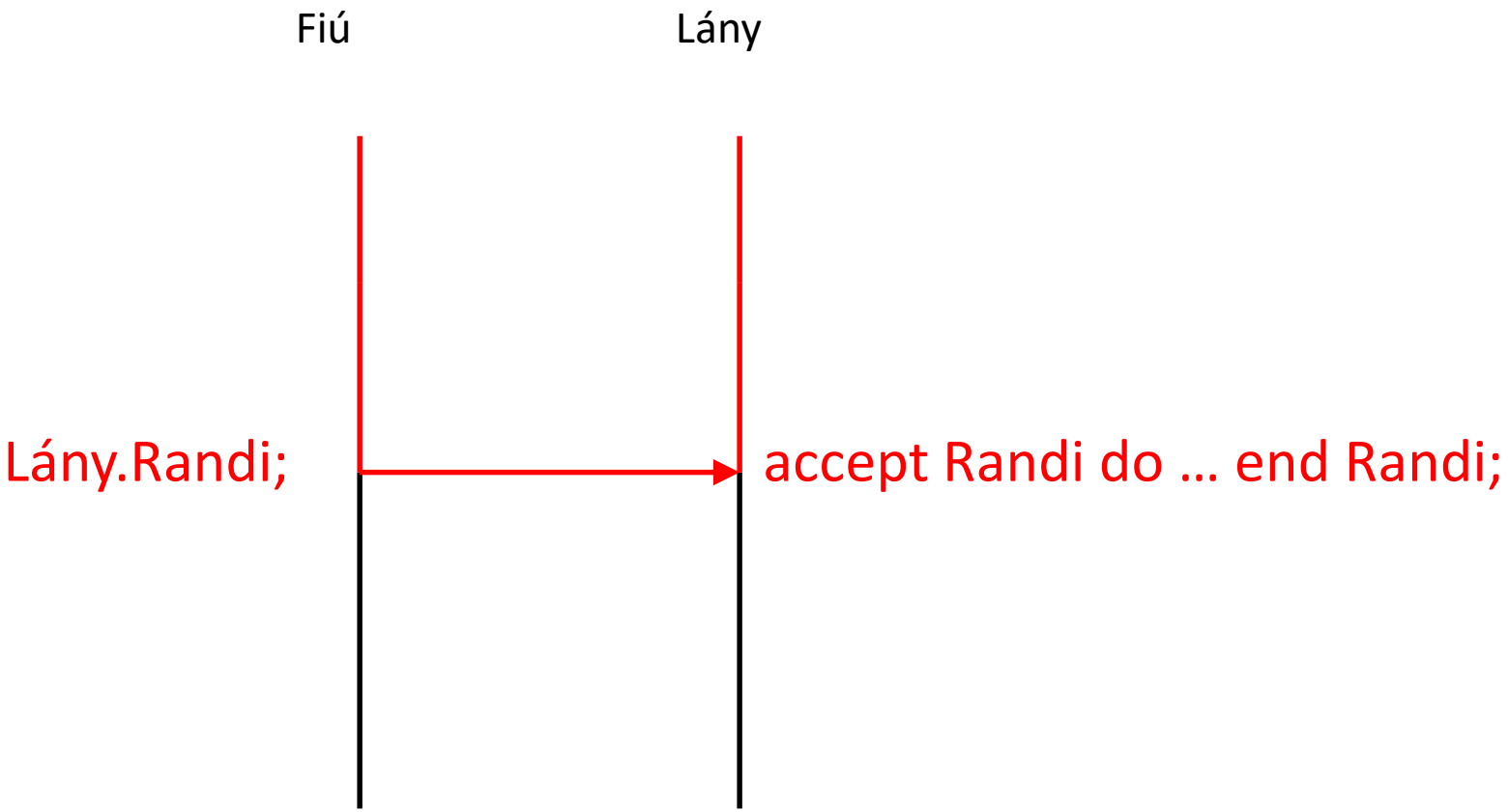
A lány bevárja a fiút (3)



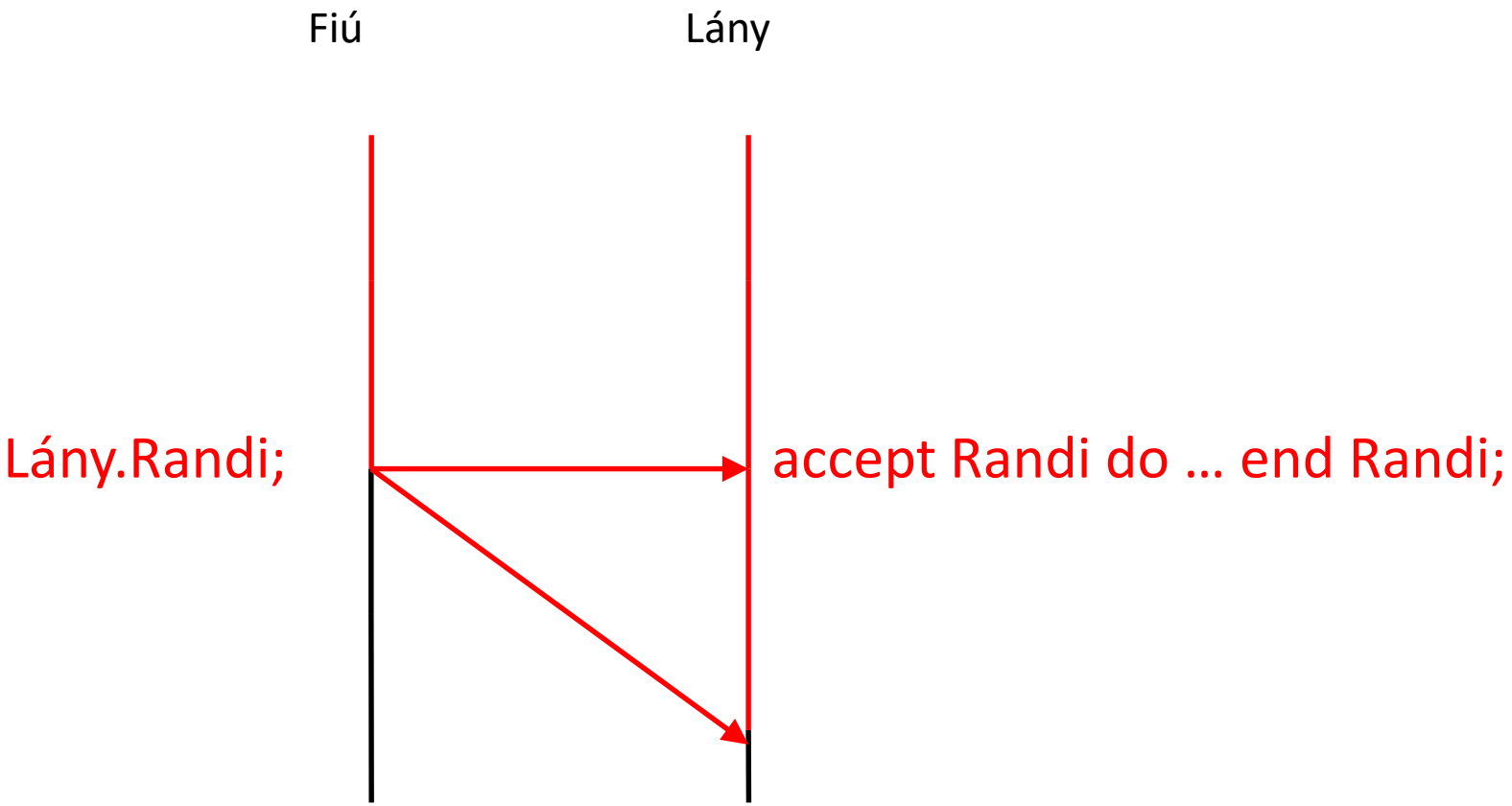
A lány bevárja a fiút (4)



A lány bevárja a fiút (5)



A lány bevárja a fiút (6)



A lány bevárja a fiút(7)