Operációs rendszerek 1. – 3. előadás
Az operációs rendszer működése

Soós Sándor

Nyugat-magyarországi Egyetem
Simonyi Károly Műszaki, Faanyagtudományi és
Művészeti Kar
Informatikai és Gazdasági Intézet

E-mail: soossandor@inf.nyme.hu

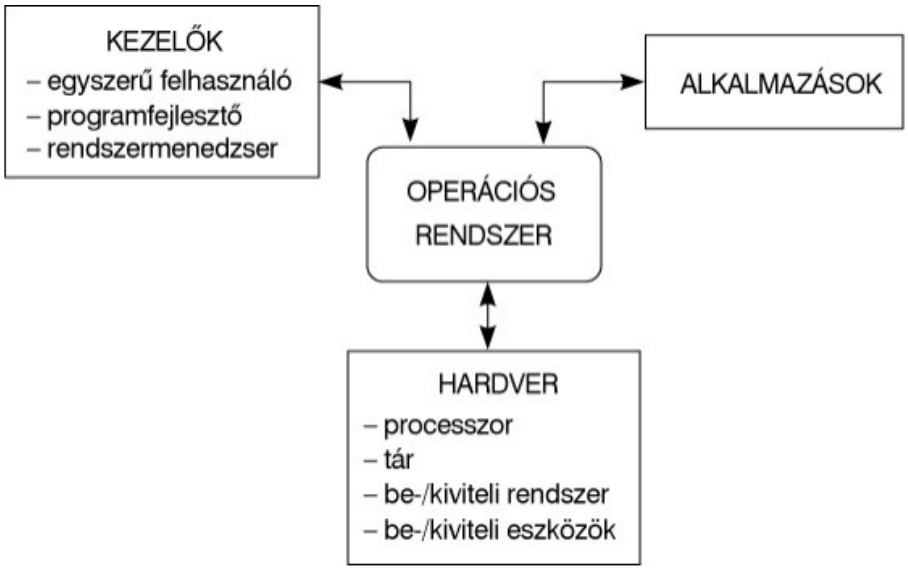
Tartalomjegyzék.

Tartalomjegyzék

1. Ismétlés

1.1. Emlékeztető az előző órákról

Az operációs rendszer környezeti (context) diagramja.



1. ábra. Az operációs rendszer környezeti (context) diagramja

Különböző felhasználók.

- Egyszerű felhasználó
  - Programokat futtat
- Alkalmazásfejlesztő
  - Programokat ír
- Rendszermenedzser
  - A rendszert üzemelteti

Az operációs rendszerek belső szerkezete.

- Ahogyan az eddigiekben láttuk, az operációs rendszerek nagy és összetett rendszerek
- Hogyan épül fel egy ilyen rendszer?
- Hogyan lehet ilyen rendszereket létrehozni?
- Amit most megismerünk, az jól fog jönni akkor is, amikor majd mi írunk programokat (PROG1, PROG2, ...)
- Hogyan álljunk neki egy nagy program megírásának?
  - A Programozás alapjai órán megtanuljuk, hogyan kell megírni egy egyszerű, kisméretű programot?
  - Mit nevezünk kisméretű, egyszerű programnak?
  - Ha már meg tudunk írni kis programokat, hogyan írunk meg egy nagyméretű, összetett programot?
  - A megoldás: problémafelbontás

Problémafelbontás,.

- Nem csak a programozás közben használhatjuk ezt a módszert
- Ha egy feladat nagyobb méretű, mint amit könnyen át tudnánk látni, akkor az első feladat, hogy felbontjuk kisebb részekre
- Ha egy rész már akkora méretű, amit átlátunk és meg tudunk oldani, akkor oldjuk meg
- Ha még ezt sem látjuk át, akkor osszuk tovább még kisebb részekre
- Ezt addig folytatjuk, amíg minden részfeladatot meg nem oldottunk

- Ezután a megoldott részmegoldásokból összeépítjük a teljes megoldást
- A problémafelbontást és a részmegoldások összeépítését sokféleképpen végezhetjük
- Ezt a módszert mindenféle feladat megoldásánál használhatjuk
- Ha programozási feladatról van szó, akkor programozási módszertanoknak nevezzük ezeket a felbontási módszereket
- Sok különböző programozási módszertan létezik, ezek megismerése segítségünkre lesz a programozás során
- Vizsgáljunk meg két felbontási módszert:
    1. Rétegekre bontás
    2. Modulokra bontás

Az operációs rendszerek belső szerkezete,.

- Az operációs rendszerek nagy méretű szoftverrendszerek, vizsgálatuk sok programozási tanulsággal szolgál
- A korai operációs rendszerek (pl. OS/360) úgynevezett monolitikus szerkezetűek voltak, azaz nem rendelkeznek belső szerkezettel, nem bonthatók kisebb részekre
- Ez azt eredményezi, hogy bármilyen módosítás a programban hatással lehet a teljes programra
- Ennek köszönhető a sok hiba az OS/360 operációs rendszerben, és ez okozza azt is, hogy minden javítás sok újabb hibát eredményezett
- Mit lehet tenni ez ellen?
- Részekre kell bontani a rendszert
- Leggyakrabban a két látott módszert kombináltan használjuk
- Kialakítunk néhány réteget és több funkcionális modult
- Tipikus rétegek:
    - a hardverfüggő részek egy alsó, hardverközeli rétegbe kerülnek
    - az operációs rendszer alapfunkciói a rendszermagban (kernel) valósulnak meg
    - a feltétlen szükségest meghaladó kényelmi szolgáltatások
    - a felhasználói programokból érkező rendszerhívások fogadó felületét megvalósító réteg

- Tipikus modulok:
  - folyamatkezelés
  - tárkezelés
  - fájlkezelés
  - be-/kivitel kezelése
  - háttértár kezelése
  - hálózatkezelés
  - védelmi és biztonsági rendszer
  - parancsértelmezés
  - felhasználóbarát kezelői felületek
- A következő órákon részekre szedjük az operációs rendszert, és sorban megvizsgáljuk az egyes részeket, rétegeket és modulokat

.

Ismétlés vége

2. Működés

2.1. Az operációs rendszer működése

A szoftverek működése.

- Hogyan működik egy hagyományos program? (Például, amit az utolsó Programozás alapjai gyakorlaton írtunk)
  - Sorban egymás után hajtódnak végre az utasítások
  - A program elindul
  - Sorban végrehajtja az utasításokat
  - A program befejeződik
- Minden esetben így történik?
  - NEM!
  - Közbeléphet egy megszakítás
- Mit jelent a megszakítás?

Megszakítási rendszer.

- Annak érdekében, hogy a számítógép időben reagálni tudjon a legkülönbözőbb eseményekre, a processzorokat úgy alakítják ki, hogy a normál működés közben is fogadni tudjon speciális jelzéseket
- Ezeket nevezzük megszakításnak
- Ha egy megszakítás érkezik a processzorhoz, akkor félbeszakítja az éppen végrehajtás alatt álló programot
- Megjegyez minden információt, ami a folytatáshoz szükséges
- Elugrik a beállított megszakításkezelő programrészhez
- Végrehajtja a megszakításkezelő programot
- Visszatér a megszakított programhoz
- Folytatja a program futtatását

Milyen megszakítások léteznek?.

- A programozó által szándékosan kiváltott megszakítás
- Valamilyen külső eszköz küld jelzést (signal) – A külső most a processzorhoz képest értendő. Pl.
  - egy eszköz befejezte a processzor által elindított tevékenységét, felpörgött a lemezegység, az író/olvasó fej elérte a kívánt pozíciót, egy memóriablokk írása befejeződött
  - Az órajel generátor küldi a következő időjelzést. Ez szabályos időközönként bekövetkezik, és vezérli az összes áramköri elem munkáját (karmester)
- Valamilyen hiba történt. A hibák különböző jellegűek lehetnek:
  - szoftveres hiba, pl. nullával való osztás történt a programban
  - hardverhiba, valamelyik eszköz hibás működést észlelt, pl. nem sikerül beolvasni a lemez tartalmát

Az operációs rendszer működése,.

Vizsgáljuk meg, hogyan működik az operációs rendszer egy alkalmazás végrehajtása közben!

- Sorban végrehajtódnak az alkalmazás utasításai

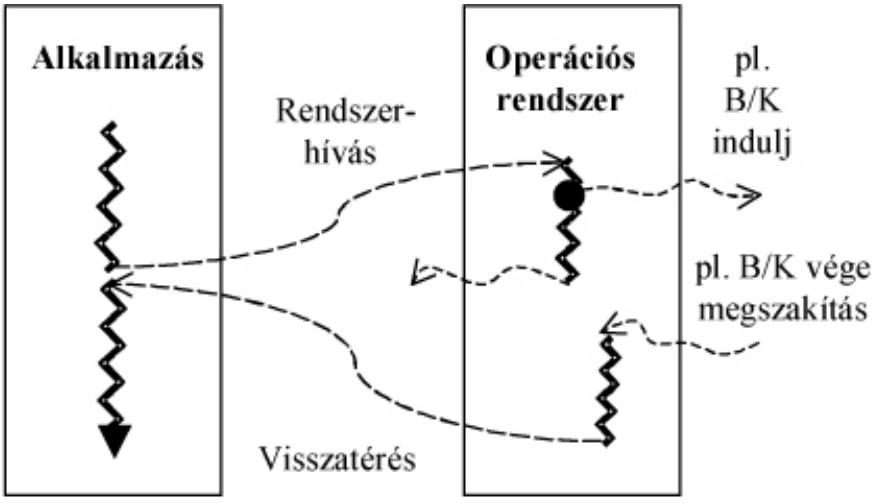
- Az operációs rendszer ilyenkor várakozik. „Nem csinál semmit ebben a pillanatban”
- Meddig tarthat ez legfeljebb?
- Hogyan léphet működésbe az operációs rendszer?
  1. A futó program rendszerhívást hajt végre
  2. A processzor külső megszakításkérést fogad
  3. Hibamegszakítás (exception) következik be
- A legtöbb számítógéprendszerben mindhárom esetet a megszakítási rendszer segítségével valósítják meg
- Ezért mondhatjuk, hogy az operációs rendszer mindig megszakítások útján lép működésbe

Rendszerhívások,.

- A futó alkalmazás kér valamilyen szolgáltatást az operációs rendszertől
- Az OS végrehajtja, és általában visszaadja a vezérlést a hívó programnak
- Vannak azonban olyan speciális esetek, amikor nem kerül vissza a vezérlés a hívóhoz
- Ezek nagyon fontos esetek, ezekkel külön kell foglalkoznunk:
  - A program befejeződését jelző rendszerhívás. A vezérlés visszakerül az operációs rendszerhez, vagy indulhat egy parancsfájl következő parancsa
  - Erőforrás-igénylés történt, amit a rendszer nem tud azonnal kielégíteni, a hívó programnak várakoznia kell
  - Várakozás egy másik program jelzésére, vagy üzenetére, de az még nem érkezett meg
  - Adott időtartamú késleltetés, vagy adott időpontra való várakozás
  - Újraütemezés kérése, azaz a futó program lemond a futásról más programok javára
  - Input/output (bemenet/kimenet) művelet indítása, aminek végrehajtása alatt a hívó várakozik
- Amikor egy rendszerhívás következtében egy program várakozásra kényszerül, akkor az OS elmenti az állapotát és átvált egy másik program végrehajtására
- A hívó program ezt úgy érzékeli, hogy a rendszerhívás lassan hajtódik végre (szinkronművelet)

- Egyes rendszerek lehetővé teszik az aszinkron rendszerhívást is. Ilyenkor a program továbbfuthat, amíg a rendszerhívás végrehajtódik, majd egy jelzést kap, amikor a rendszerhívás befejeződött
- Más rendszerekben arra van lehetőség, hogy a programozó megadjon egy belépési pontot, ahova a rendszer átadhatja a vezérlést az aszinkron rendszerhívás befejeződése után, azaz egy megszakításkezelés jellegű programrészt építsen be a programba
- A három lehetséges vezérlési utat a következő ábrákon láthatjuk:

Szinkron rendszerhívás.



Szinkron rendszerhívás

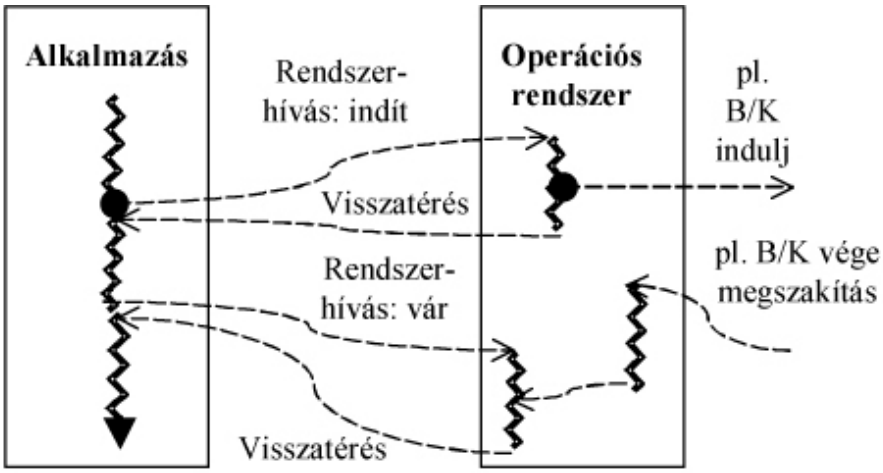
2. ábra. Szinkron rendszerhívás

Aszinkron rendszerhívás várakozással.

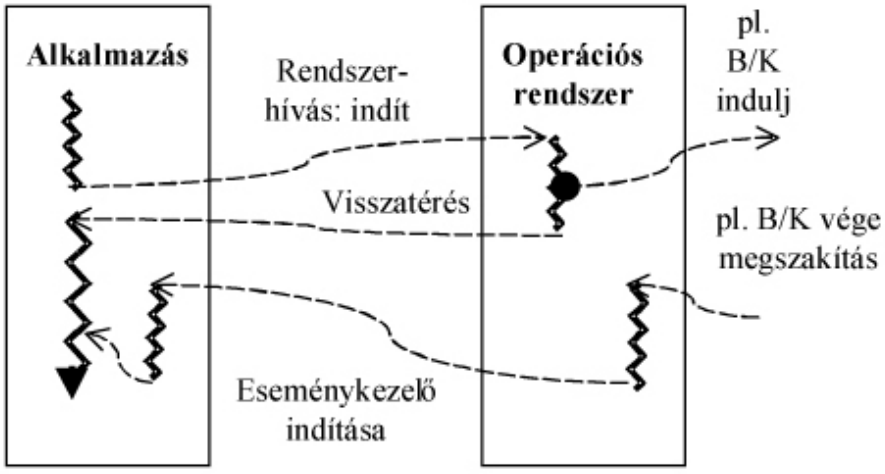
Aszinkron rendszerhívás eseménykezeléssel.

2.2. Az operációs rendszer mint virtuális gép

Az operációs rendszer mint virtuális gép.



3. ábra. Aszinkron rendszerhívás várakozással



4. ábra. Aszinkron rendszerhívás eseménykezeléssel

- A korábbiakban megállapítottuk, hogy az operációs rendszer egyik fő feladata, hogy megvalósítson egy virtuális gépet a felhasználói és a programozói felületen
- Most vizsgáljuk meg ezt a virtuális gépet belülről
- Először vizsgáljuk meg a folyamatokat:
  - A folyamat fogalma
  - A folyamatokból álló rendszerek
  - Független, versengő és együttműködő folyamatok
  - Folyamatok születése és halála
  - Folyamatok együttműködése
  - Folyamatok szinkronizációja
  - Folyamatok kommunikációja
  - Holtpont
  - Éhezés, kiéheztetés
  - Klasszikus konkurens problémák

A folyamat fogalma.

- Eddig különböző típusú számítógépes rendszereket vizsgáltunk: kötegelt, időosztásos, valósidejű, multiprogramozott, stb.
- Ezekben a rendszerekben a feldolgozás egységeire különböző elnevezéseket használunk: feladat, task, munka, job, program, felhasználói munkamenet, session, ...
- Az operációs rendszer szintjén azonban nagyon hasonló feladatokkal találkozunk, ezeket együtt fogjuk vizsgálni, és ennek során folyamatnak nevezzük a feldolgozás egységeit
- Alternatív elnevezések:
  - folyamat, process, job, task, ...
- Ezek alatt ugyanazt fogjuk érteni:
  - Folyamat: Meghatározott sorrendben végrehajtott műveletek sorozata
  - A számítógépes terminológiában: A folyamat egy végrehajtás alatt álló program

A szál fogalma.

- A folyamatokhoz kapcsolódó fogalom a szál (thread)
  - Egymás utáni műveletek fűzére
  - Párhuzamos végrehajtású, közös memóriát használó programrészek a folyamatokon belül
  - Saját logikai processzoruk van, de közös logikai memóriát használnak
  - Az operációs rendszer gyorsabban tud váltani a szálak, mint a folyamatok között
- A szálak és a folyamatok megkülönböztetésére szokás használni a következő elnevezéseket is:
  - Folyamat: „Nehézsúlyú (heavyweight) folyamat"
  - Szál: „Pehelysúlyú (lightweight) folyamat"

Vezérlési szál.

- Ha egy adott programot többször végrehajtunk ugyanabban a környezetben, nem kapunk azonos eredményt minden alkalommal
- Ennek oka, hogy elágazások lehetnek a kódban, amelyek eredménye minden indításkor eltérő lehet
- A programkód alapján előállítható a program összes lehetséges végrehajtása
- Felépíthetünk egy irányított gráfot, aminek csomópontjai az utasítások, az élek pedig a végrehajtás lehetséges sorrendjében kötik össze az utasításokat
- A program egy lefutása megfelel a gráfban egy útvonalnak a starttól egy végcsomópontig, ezt nevezzük vezérlési szálnak

Folyamatmodell,.

- A folyamatok vizsgálatára felállítunk egy modellt a következőképpen:
  - Minden folyamathoz tartozik egy logikai processzor és egy logikai memória
  - A memória tartalmazza a programokat, a konstansokat és a változókat
  - A processzor hajtja végre a programot
  - A programkódban szereplő utasítások és a végrehajtó processzor utasításkészlete megfelelnek egymásnak

  - Egy utasítás végrehajtását általában oszthatatlannak tekintjük, azaz a folyamat állapotát csak olyankor vizsgáljuk, amikor egy utasítás már befejeződött, a következő pedig még nem kezdődött el
  - A programvégrehajtás egy vezérlési szál mentén, szekvenciálisan történik, alapvetően az utasítások elhelyezkedésének sorrendjében, ettől speciális utasítások esetén van eltérés
  - A processzornak vannak saját állapotváltozói (programszámláló, veremmutató, regiszterek, jelzőbitek stb.), amelyek értéke befolyásolja a következő utasítás végrehajtásának eredményét

- A memória a RAM-modell szerint működik, azaz
  - tárolórekeszekből áll
  - egy dimenzióban, rekeszenként címezhető
  - csak rekeszenként, írás és olvasás műveletekkel érhető el
  - az írás a teljes rekesztartalmat felülírja az előző tartalomtól független új értékkel
  - az olvasás nem változtatja meg a rekesz tartalmát, tehát tetszőleges számú, egymást követő olvasás az olvasásokat megelőzően utoljára beírt értéket adja vissza

- A folyamatot egy adott pillanatban leíró információk a következők (ezt nevezzük a folyamat állapotterének):
  - a memória tartalma (a programkód és a változók pillanatnyi értéke)
  - a végrehajtó processzor állapota (a program számláló és a többi regiszter és jelzőbit értéke)

- Az operációs rendszer feladata, hogy a fizikai eszközökön (fizikai processzor és memória) egymástól elkülönítetten (védetten) létrehozza és működtesse a folyamatoknak megfelelő logikai processzorokat és memóriákat

- Ez a modell alkalmazható egy- és többprocesszoros gépeken egyaránt

- Egyprocesszoros rendszerek esetén minden logikai processzort ugyanazon a fizikai processzoron kell megvalósítani

- Multiprocesszoros rendszerekben a logikai processzorok szétoszthatók különböző processzorokra, vagy futhatnak azonoson is

- Szálak esetében az a különbség, hogy minden szálnak saját logikai processzora van, a memóriájuk viszont közös, azaz a programkódjuk és a változóik azonosak

Folyamatokból álló rendszerek.

- Egy számítógéprendszerben szinte mindig több folyamat van jelen egyidőben
- Vannak olyan operációs rendszerek, amelyek saját hatáskörben kezelik ezeket a folyamatokat elsősorban a hatékony erőforrás-kihasználás érdekében
- Más rendszerek a felhasználóknak is lehetőséget adnak folyamatok és szálak kezelésére, ezeket nevezzük multitaszkos rendszereknek
- Napjainkban a legtöbb operációs rendszer ilyen

Miért használunk folyamatokat?.

- **Hatékonyabb erőforrás-kihasználás**: a processzorkihasználás javításának általánosítása
- **A feladat-végrehajtás gyorsítása**: az erőforrások hatékonyabb kihasználása önmagában is gyorsítja a feladatok megoldását, de további gyorsítást eredményezhet, ha a feladatokat párhuzamosan futó részekre bonthatjuk
- (Ismerünk olyan feladatot, amit érdemes párhuzamosítva megoldani?)
- **Többféle feladat egyidejű végrehajtása**: növeljük a felhasználók komfortérzetét, és munkájuk hatékonyságát, ha egyidőben több célra is használhatják a számítógépet. Pl. egy hosszú számítás közben szöveget szerkeszthetnek miközben kedvenc zenéjüket hallgathatják
- **Rendszerstrukturálás**: bizonyos feladatokra könnyebb és hatékonyabb olyan programot készíteni, amiben több folyamat működik párhuzamosan. Tipikusan ilyenek a valós idejű rendszerek, amelyeknek több különböző eseményre kell reagálniuk egyidőben. Ebben az esetben az a legnyilvánvalóbb megoldás, hogy minden esemény bekövetkezésekor elindítunk egy válaszfolyamatot egymástól függetlenül, amelyek lekezelik az adott eseményt. Minden más megoldás feleslegesen bonyolítja a rendszer működését
- **Negatívum**: egy folyamatokból álló rendszer kifejlesztése és tesztelése nehezebb feladat, mint egy szekvenciális programé. Korábban láttuk, hogy egy szekvenciális program összes lehetséges lefutása előállítható, és sorban megvizsgálható, tesztelhető. Ezzel szemben egy párhuzamos folyamatokból felépülő rendszer viselkedése sokkal nehezebben áttekinthető, elméletben sem állítható elő az összes lehetséges állapot. Az észlelt hibák reprodukálása sokkal nehezebb, sokszor nem is lehetséges

Független, versengő és együttműködő folyamatok,.

Egy rendszer folyamatai egymáshoz való viszonyukat tekintve háromfélék lehetnek:

1. **Független folyamatok**:
   - Egymás működését semmiképpen nem befolyásolják
   - Végrehajtásuk teljes mértékben aszinkron
   - Párhuzamosan és egymás után is végrehajtódhatnak tetszőleges sorrendben
   - Külön-külön, önálló programokként vizsgálhatók
2. **Versengő folyamatok**:
   - Nem ismerik egymást, de közös erőforrásokon kell osztozniuk
   - Ilyen folyamatok alakulnak ki például az egymást nem ismerő felhasználói jobok feldolgozásakor
   - Nem kell tudniuk arról, hogy egy multiprogramozott rendszerben fognak futni, programkódjuk ugyanolyan, mintha egy soros feldolgozást végző rendszerre írták volna
   - A folyamatok helyes és hatékony futtatását az operációs rendszernek kell megoldania, pl.
     - minden folyamatnak külön memóriaterülete legyen
     - a nyomtatások ne gabalyodjanak össze
     - hatékonyan használjuk az erőforrásokat
   - Ezeket a feladatokat gyakran együttműködő folyamatokkal oldja meg az operációs rendszer
     - Például ha egy folyamat nyomtatni akar a rendszerhez kapcsolt nyomtatóra, amikor egy másik folyamat nyomtat, akkor meg kell várnia, amíg a másik folyamat befejezi a nyomtatást
   - Az operációs rendszer saját belső folyamatait *rendszerfolyamatnak*, a felhasználók folyamatait *felhasználói folyamatnak* nevezzük
3. **Együttműködő folyamatok**:
   - Ismerik egymást
   - Együtt dolgoznak egy feladat megoldásán
   - Információt cserélnek egymással
   - Egy programozó, vagy egy programozó csapat írta meg az egyes folyamatokat, tudatosan alakította ki az egyes folyamatokat
   - A folyamatok kooperatívan (együttműködve) futnak

- A párhuzamosan futó folyamatok lehetnek szálak is
- Az együttműködés műveletei a programkódban is megjelennek, a logikai processzor utasításkészletében szerepelnie kell ezeknek a műveleteknek:
  - folyamat/szál elindítása
  - erőforrások kizárólagos használatának kérése, befejezése
  - üzenetküldés egy másik folyamatnak
- Az együttműködő folyamatok üzenetküldés segítségével hangolják össze működésüket

Folyamatok születése és halála,.

Hogyan jönnek létre és szűnnek meg a folyamatok?

- Amikor bekapcsoljuk a számítógépet, elindul egy rendszerépítési folyamat (boot, inicializálás)
- Ez egy ősfolyamat, amelyik létrehozza a rendszer alapfolyamatait
- A rendszerépítés végén létrejön egy kész operációs rendszer, ami több folyamatból áll (rendszerfolyamatok)
- Minden terminálhoz tartozik egy rendszerfolyamat, amelyik fogadja a felhasználói parancsokat és végrehajtja azokat, szükség esetén elindít újabb folyamatokat (felhasználói folyamatok)
- Ha a rendszer megengedi, akkor a felhasználói folyamatok is elindíthatnak újabb folyamatokat (fork, create művelet)
- Egy folyamat befejeződhet azért mert az utasítássor végére ért (exit művelet), vagy megszüntetheti egy másik folyamat, például hiba miatt (kill művelet)
- **Statikus rendszer**: Csak a rendszer elindulásakor jönnek létre és a leállásakor szűnnek meg folyamatok, működés közben nem
- **Dinamikus rendszer**: A rendszer működése közben bármikor születhetnek és megszűnhetnek folyamatok
- **Folyamat fa**: Ha egy folyamat elindít egy másik folyamatot, akkor szülő, illetve gyermekfolyamatnak nevezzük őket. Ennek alapján egy fába rendezhetjük a folyamatokat. Ezt nevezzük **folyamat fá**nak.
- **Hierarchikus erőforrás-gazdálkodás**: minden folyamat csak addig létezhet, amíg a szülője létezik. A gyerek folyamat csak a szülő erőforrásaiból gazdálkodhat

- **Globális erőforrás-gazdálkodás**: minden folyamat egyenrangú az összes többivel, versenyezhet a rendszerben lévő összes erőforrásért
- Az operációs rendszerek egy része hierarchikus, más része globális erőforrás-gazdálkodást valósít meg

Folyamatok együttműködése.

- Információátadással valósul meg
- Az információátadás történhet:
  - közös memórián keresztül
  - üzenetváltással
- Az átadott információ az 1 bittől a tetszőleges méretű adatbázisokig terjedhet

Folyamatok együttműködése közös memórián.

- A együttműködő folyamatok mindegyike a saját memória címtartományában lát egy közös memóriaterületet
- Valamilyen adatátviteli rendszer biztosítja, hogy minden folyamat helyesen el tudja végezni az olvasás és írás műveleteket a közös memórián
- A folyamatok párhuzamos futása miatt előfordulhat, hogy a közös memóriát egy időben több folyamat is írhatja, vagy olvashatja
- A RAM-modell szerint működő memória erre nincsen felkészítve
- Ezért a memóriát az ún. PRAM-modell szerint kezeljük

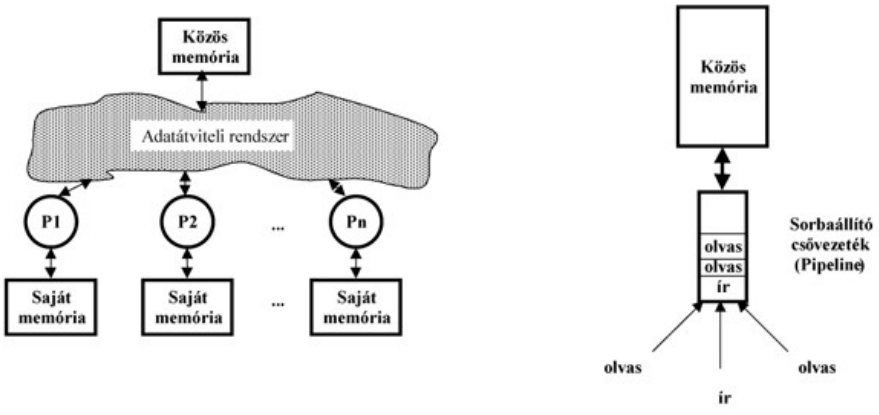
PRAM memória-modell,.

- PRAM-modell – Pipelined Random Access Memory
- A RAM-modellt kiegészíti a következő tulajdonságokkal:
  - **olvasás-olvasás ütközés**: Ha két folyamat egyszerre akarja olvasni a közös memóriarekeszt, akkor mindkettő ugyanazt az értéket kapja és az megegyezik a memóriarekesz tartalmával
  - **olvasás-írás ütközés**: Ha az egyik folyamat írni, a másik ugyanakkor olvasni akarja a közös rekesz tartalmát, akkor az felülíródik a beírni szándékozott adattal, az olvasás eredménye vagy a rekesz régi, vagy az új tartalma lesz, más érték nem lehet

  - **írás-írás ütközés**: Ha két folyamat egyidőben akarja írni a közös rekeszt, akkor valamelyik művelet hatása érvényesül, a rekesz új tartalma a kettő közül valamelyik lesz, harmadik érték nem alakulhat ki

- Azaz az egyidejű műveletek nem interferálhatnak, nem lehet közöttük zavaró kölcsönhatás
- Hatásuk olyan, mintha egy előre nem meghatározható sorrendben, de egymás után hajtódnának végre
- Erre utal az elnevezés: pipeline = csővezeték
- Másképp fogalmazva az írás és olvasás műveletek oszthatatlanok (atomiak)
- A közös memóriával történő adatcseréhez tehát PRAM-modell szerint működő memóriát használunk, és emellett össze kell hangolni a folyamatok működését, szinkronizálni kell a folyamatokat
- Például ha át akarunk adni adatokat a közös memórián keresztül, akkor biztosítanunk kell, hogy a fogadó azután olvassa el a közös memóriát, miután a küldő elhelyezte ott az információt. Ehhez van szükség a folyamatok szinkronizációjára. Ennek megvalósítási lehetőségeivel később foglalkozunk

Folyamatok együttműködése közös memórián.

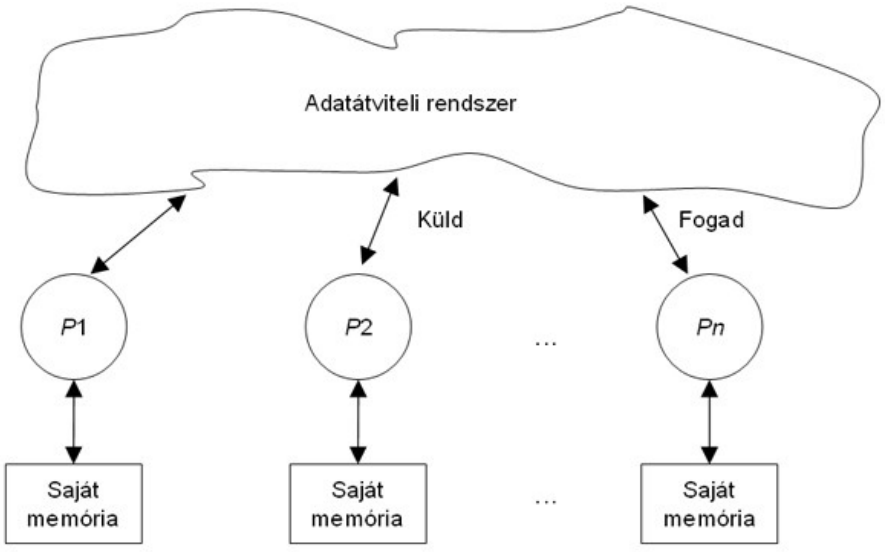


5. ábra. Folyamatok együttműködése közös memórián

Folyamatok együttműködése üzenetváltással.

- Nincsen közös memória
- Az adatátviteli rendszer a logikai processzorokat kapcsolja össze
- Két új utasítás a logikai processzorok utasításkészletében:
  - Küld – Send
    - *Küld(<cím>, <folyamat>)*: a küldő folyamat a *<cím>* címen lévő adatokat elküldi a *<folyamat>* folyamatnak
  - Fogad – Receive
    - *Fogad(<cím>, <folyamat>)*: a fogadó folyamat a *<folyamat>* folyamattól kapott adatokat eltárolja a saját *<cím>* memóriacímén
- A *<cím>* alatt egyszerűen érthetünk változókat!

Folyamatok együttműködése üzenetváltással.



6. ábra. Folyamatok együttműködése üzenetváltással

2.3. Folyamatok szinkronizációja

Folyamatok szinkronizációja.

- A folyamatok bizonyos esetekben egymástól függetlenül futhatnak
- Máskor szükség van arra, hogy korlátozzuk az egyes folyamatok „szabadonfutását"
- A műveletek végrehajtására vonatkozó időbeli korlátozásokat nevezzük szinkronizációnak
- A korlátozások alapesetei a következők:
  - Kölcsönös kizárás (mutual exclusion)
  - Egyidejűség (randevú)
  - Előírt sorrend (precedencia)

Kölcsönös kizárás (mutual exclusion),.

- Kritikus szakasz
  - A különböző folyamatokban lehetnek olyan utasítássorozatok amelyek egyidejű (konkurens) végrehajtása nem megengedett, azaz ezek a kritikus szakaszok kölcsönösen ki kell, hogy zárják egymást
  - Tipikusan ilyen esetek a közösen használt erőforrásokon végzett oszthatatlan műveletek végrehajtása
  - Például nyomtatás, adatbázisokon végrehajtott tranzakciók
  - **Formális definíció**: A K kritikus szakasz folyamatok végrehajtási szakaszainak (utasítás-sorozatainak) olyan S_k halmaza, amelynek bármely két $s_{k,i}$ és $s_{k,j}$ elemét tilos átlapolva végrehajtani
  - Ha egy folyamat $s_{k,i} \in S_k$ szakaszának végrehajtása megkezdődik, azt mondjuk, hogy a folyamat *belépett* a kritikus szakaszba, hasonlóan $s_{k,i}$ befejeződésekor azt mondjuk, hogy a folyamat *kilépett* a kritikus szakaszból
  - Bármely folyamat csak akkor léphet be egy K kritikus szakaszba, ha abban egyetlen más folyamat sem tartózkodik, ellenkező esetben várakoznia kell
  - Amikor egy folyamat elhagyja a kritikus szakaszt, akkor a várakozók közül egy beléphet
  - Egy rendszerben lehet több kritikus szakasz is, ezek függetlenek egymástól
  - Egy folyamat tartózkodhat egyszerre több kritikus folyamatban is, például akkor, ha több kölcsönös kizárást igénylő közös erőforrást használ egyszerre

- A kritikus szakaszokat a folyamatokat író programozók definiálják, a belépési szándékot és a kilépést speciális utasításokkal jelölik, pl. *Lefoglal(nyomtató), Felszabadít(nyomtató)*. Ezek az utasítások rendszerhívások, és a program futásakor az operációs rendszer dönti el, hogy a folyamat beléphet a kritikus szakaszba, vagy várakoztatja
- Versengő folyamatok esetén ezt mindenképpen az operációs rendszernek kell megvalósítania, Együttműködő folyamatok esetében a programozók bármilyen egyeztetett megoldást alkalmazhatnak

Egyidejűség (randevú).

- Bizonyos esetekben szükség van arra, hogy két, vagy több folyamat egyidőben tartózkodjon tevékenységének egy adott szakaszában
- A pontos egyidejűség nehezen definiálható, valójában arról van szó, hogy egyik folyamat sem léphet túl egy adott szakaszon, amíg a többi be nem lépett ebbe a szakaszba
- Tipikusan ez a helyzet az átmeneti tároló nélküli kommunikáció esetén, amikor Küld és Fogad jellegű műveleteket hajtanak végre a folyamatok köztes tároló nélkül

Előírt sorrend (precedencia).

- Gyakran előfordul olyan eset, hogy az egyik folyamat előállít egy olyan adatot, amit a másik felhasznál
- Ilyenkor garantálni kell, hogy az előállító folyamat előbb hajtódjon végre, mint a fogyasztó folyamat
- Legyen a P_1 folyamat egy utasítása S_1, P_2-é pedig S_2
- Ekkor az $S_1 \Rightarrow S_2$ precedencia előírása azt jelenti, hogy S_1 végrehajtásának be kell fejeződnie mielőtt S_2 elindulna
- Ha P_1 és P_2 úgy hajtódna végre, hogy S_2 hamarabb kerülne sorra, mint S_1, akkor P_2-nek meg kell várnia, amíg P_1-ben S_1 végrehajtása befejeződik

PRAM alapú szoftveres megoldások szinkronizációra,.

- Manapság már a processzorba épített hardveres megoldásokkal kezeljük a szinkronizációt
- Korábban a hardver nem nyújtott ilyen támogatást, különböző szoftveres megoldásokat dolgoztak ki

- Mivel a leggyakrabban úgyis közös memóriával működnek a folyamatok, kézenfekvő volt, hogy a kölcsönös kizárást is ennek segítségével valósítsuk meg a PRAM-modell felhasználásával
- Elvárások a lehetséges megoldásokkal szemben
  - Minden esetben biztosítsa a kölcsönös kizárást (ne legyen kijátszható a rendszer)
  - Csak azokat a folyamatokat érintse az adminisztráció, amelyek érdekeltek a kizárásban, a többi folyamatnak ne kelljen foglalkoznia ezzel
  - Lehetőleg véges időn belül sorra kerüljön minden várakozó folyamat, kerüljük el a kiéheztetést (ettől a feltételtől időnként eltekintünk)

Foglaltságjelző bit használata,.

- Minden kritikus szakasz előtt helyezzünk el egy foglaltságjelző bitet
- A jelzőnek két lehetséges értéke van: **szabad**, **foglalt**, az egyiket jelentse a 0, másikat az 1 érték
- A jelzőbit kezdőértéke legyen **szabad**
- Ha egy folyamat be akar lépni a kritikus szakaszba, akkor megvizsgálja a jelzőbitet:
  - ha az értéke **szabad**, akkor **foglalt**ra állítja és belép a kritikus szakaszba. A kritikus szakasz elhagyásakor **szabad**ra állítja a bitet
  - ha a bit értéke **foglalt**, akkor addig várakozik, amíg **szabad** nem lesz

Pszeudo kód (Mit jelent ez?)

```
var jelző : {foglalt, szabad} := szabad
```

Minden folyamat esetén a kritikus szakasz használatakor:

```
...
belépés:
  olvas(jelző)
  if jelző = foglalt then goto belépés
  ír( jelző, foglalt )
<kritikus szakasz>
  ír( jelző, szabad )
...
```

Helyes ez az algoritmus?

- Minden esetben jól működik ez az algoritmus?
- NEM!
  - Mivel a jelzőbit a közös memóriában található, előfordulhat, hogy két folyamat egyszerre vizsgálja meg a jelzőbitet. Ha ekkor **szabad** a jelző, akkor mindkét folyamat belép a kritikus szakaszba
  - Ez nyilvánvalóan hibás
  - Mi a helyzet akkor, ha valójában egyprocesszoros rendszeren futnak a folyamatok?
    - Ekkor ki van zárva az egyidejű olvasás
    - Ekkor is előfordulhat, hogy a kiolvasás és a visszaírás között olvas a másik folyamat
  - Szükség van tehát jobb megoldásra

Peterson-algoritmus, 1981.

Alapötlet:

- Ez a megoldás csak két folyamatra működik: P_1, P_2
- Mindkét folyamatnak legyen egy jelzője (tömb), amivel jelzi a belépési szándékát
- Legyen egy közös változó, ami jelzi, hogy egyidejű belépési szándék esetén melyik folyamat a kedvezményezett

Peterson-algoritmus pszeudo kód,.

```
var jelző: array[1..2] of {foglal, szabad} := szabad
    következő: {1, 2}
```

P_1 folyamat:

```
ír( jelző[1], foglal )
ír( következő, 2 )
belépés1:
  olvas( jelző[2] )
  if jelző[2] = szabad then goto belép1
  olvas( következő )
  if következő = 2 then goto belépés1
belép1:
  <kritikus szakasz>
kilép1:
  ír( jelző[1], szabad )
```

P_2 folyamat:

```
ír( jelző[2], foglal )
ír( következő, 1 )
belépés2:
  olvas( jelző[1] )
  if jelző[1] = szabad then goto belép2
  olvas( következő )
  if következő = 1 then goto belépés2
belép2:
  <kritikus szakasz>
kilép2:
  ír( jelző[2], szabad )
```

Hogyan működik az algoritmus?

- Hogyan tudjuk megvizsgálni egy algoritmus működését?
  - Kövessük nyomon lépésről-lépésre!
  - Közben jegyezzük fel a változók aktuális értékeit!
  - Figyeljük, hogy az egyes folyamatok mikor tartózkodnak a kritikus szakaszban!
- Ugyanezt tesszük, amikor nyomkövetéssel vizsgálunk egy programot a fejlesztő környezetben
- Sajnos ez az egyszerű algoritmus csak két folyamattal működik
- Hogyan lehetne több folyamatra is alkalmas megoldás szerkeszteni?

Bakery-algoritmus, (Lamport, 1974).

Alapötlet

- Sorbanállás az üzletben
- Mindenki sorszámot kér, és annak sorrendjében jutnak be
- bakery = pékség
- n darab folyamatot fogunk kezelni

Bakery-algoritmus pszeudo kód,.

Adatszerkezetek közös tárban:

```
var számot_kap : array[0..n-1] of boolean := false
    sorszám : array[0..n-1] of integer := 0
```

- *számot_kap* tömb: jelzi, hogy egy folyamat éppen sorszámot kap
- *sorszám* tömb: tárolja a sorszámokat

P_i folyamat:

```
// Sorszámot kér:
  számot_kap[i] := true
  sorszám[i] := max(sorszám) + 1
  számot_kap[i] := false

// Belépés:
// Amíg van nála kisebb sorszámú, várakozik
  for j := 0 to n-1 do begin
    while számot_kap[j] do üres_utasítás
    while sorszám[j] <> 0
          and (sorszám[j], j) < (sorszám[i], i)
          do üres_utasítás
  endfor
<kritikus szakasz>

// Kilépés:
  sorszám[i] := 0
```

Hogy működik az algoritmus?

- A belépni szándékozó folyamatok sorszámot kérnek
- Ez nem atomi művelet, ezért megakadályozza, hogy egy másik folyamat megkérdezze a sorszámot mialatt a számítás folyik
- Előfordulhat, hogy két folyamat azonos sorszámot kap, mert egyidőben számítja ki
- A sorbanálló folyamatok figyelik a sorszámokat, és ha az övé a legkisebb, akkor belép a kritikus szakaszba
- Ha egy folyamat éppen kéri a sorszámot, akkor megvárja amíg befejezi
- Ha talál egy kisebb sorszámú folyamatot, akkor megvárja, amíg elhagyja a kritikus szakaszt

- A **for** ciklus végére érve biztosan beléphet a kritikus szakaszba, mert minden kisebb sorszámú folyamat végrehajtását kivárta, és ha közben újabb folyamat kért sorszámot, az már csak nála nagyobbat kaphatott
- Mivel előfordulhatnak azonos sorszámok, ezért a folyamatok sorszámával kiegészítve hasonlítjuk össze, így mindig van egyértelmű győztes
- Hogyan lehetne egyszerűbb megoldást megvalósítani?
- Az első (foglaltságjelző bites) megoldás nagyon egyszerű volt, de párhuzamos környezetben nem működik jól
- Hogyan lehetne megjavítani a bonyolultság növelése nélkül?

A PRAM-modell kiterjesztése.

- A foglaltságjelző bit esetében az okozza a problémát, hogy több folyamat is kiolvashatja a bitet, mielőtt az első olvasó foglaltra állítaná
- Ha el tudjuk érni, hogy az olvasás és írás egymás utáni végrehajtása atomi művelet legyen, akkor megoldódna a probléma
- Vezessünk be egy új műveletet OlvasÉsÍr (TestAndSet) műveletet a modellben (A gyakorlatban ez azt jelenti, hogy a processzorba építenek be hardveresen egy ilyen műveletet)
- Az OlvasÉsÍr művelet kiolvassa és visszaadja a változó értékét, majd foglaltra állítja. Mindezt oszthatalanul teszi, azaz közben nem hajtódhat végre semmilyen más művelet

Kölcsönös kizárás OlvasÉsÍr művelettel.

```
var közös_jelző : {foglalt, szabad} := szabad
```

Minden folyamat esetében:

```
belépés:
  OlvasÉsÍr( közös_jelző, foglalt )
  if közös_jelző = foglalt then goto belépés
  <kritikus szakasz>

kilépés:
  ír( közös_jelző, szabad )
```

Kölcsönös kizárás Csere művelettel.

- Hasonlóan megoldható a feladat, ha bevezetünk egy oszthatalan Csere (Swap) műveletet
- A Csere utasítás oszthatatlan módon megcseréli a memória két rekeszének tartalmát, az egyik rekesz a közös memóriában van, míg a másik a folyamat saját memóriaterületén
- Hogyan tudjuk megcserélni két memóriarekesz (változó) tartalmát?

```
var közös_jelző : {foglalt, szabad} := szabad
```

Minden folyamat esetében:

```
var saját_jelző : {foglalt, szabad}
...
saját_jelző := foglalt
belépés:
  Csere( közös_jelző, saját_jelző )
  Olvas( saját_jelző )
  if saját_jelző = foglalt then goto belépés
  <kritikus szakasz>

kilépés:
  ír( közös_jelző, szabad )
```

A szoftveres megoldások értékelése.

- Napjainkban a processzorok utasításkészletének része vagy a *Csere*, vagy a *OlvasÉsÍr* művelet
- Ezek megvalósítása olyan, hogy többprocesszoros rendszerben is garantálja az oszthatatlan végrehajtást
- Milyen elvárást nem teljesítenek ezek a megoldások?
  - Nem garantálják, hogy a kritikus szakaszba belépni szándékozó folyamatok véges időn belül bejutnak
  - A problémát az okozza, hogy az algoritmusok nem határozzák meg a kritikus szakaszba való bejutás sorrendjét
  - Ha az egyes folyamatokra bízzuk a kölcsönös kizárás megvalósítását, akkor nem tudjuk garantálni, hogy mindenki bejut véges időn belül
- Mi lehet a megoldás?
- A megoldás:

- Központilag tartsuk nyilván a belépési igényeket
- Központilag ütemezzük a folyamatokat
- De ez még nem elegendő, egyértelműen meghatározottá kell tenni az ütemezési döntést, ne a véletlenen múljon
- Ekkor készíthető olyan ütemező algoritmus, ami „igazságosan", „tisztességesen" sorolja be a folyamatokat
- Ez a központi ütemező az operációs rendszer

Szinkronizációs eszközök az operációs rendszer szintjén.

- Három eszközt fogunk megvizsgálni
- Mindhárom a kiterjesztett PRAM-modellen alapul
  1. Szemafor
  2. Erőforrás
  3. Esemény

Szemafor.

- Dijkstra javasolta először 1965-ben
- Szemafor – vasúti jelző
- Szemafor az informatikában
  - Speciális változó
  - Csak a hozzá tartozó két oszthatatlan művelettel lehet kezelni, másképp nem érhető el
  - Az általános szemafor esetében a változó típusa egész (integer, int)
  - A két műveletet többféleképpen nevezhetjük: *Belép-Kilép, Vár-Jelez, Wait-Signal*
  - Legtöbbször a holland eredeti alapján P és V műveletnek szoktuk nevezni
- A szemafor műveletek általános formája:
  - $P(s)$ (Probieren - ellenőrzés)

    ```
    while s < 1 do üres_utasítás
    s := s - 1
    ```

  - $V(s)$ (Verhogen - felszabadítás)

    ```
    s := s + 1
    ```

- Mindkét művelet oszthatatlan
- Semmilyen más úton nem érhető el a szemafor
- A véletlenre bízza, hogy a várakozók közül melyik folyamat fog bejutni a kritikus szakaszba
- Különböző szemafor típusok:
  - **Általános szemafor:**
    - Ha az s szemafornak k kezdőértéket adunk, akkor k darab folyamatot enged be egyszerre
    - Ha k darab folyamat bejutott, akkor a következő csak akkor léphet be, ha előbb egy bentlévő folyamat kilépett, azaz végrehajtott egy $V(s)$ műveletet
    - Ezzel olyan **általánosított kritikus szakaszt** hozhatunk létre, amelyben egyszerre k darab folyamat tartózkodhat
  - **Bináris szemafor:**
    - A szemafor csak 0, vagy 1 értéket vehet fel
    - 1 kezdőértékkel lehet megvalósítani a **kölcsönös kizárást**, a kritikus szakaszba belépni kívánó folyamat P műveletet hajt végre, a kilépő pedig V műveletet
    - 0 kezdőértékkel lehet megvalósítani a **precedenciát** (meghatározott sorrend), az elsőként végrehajtandó folyamat befejezéskor végrehajt egy V műveletet, a második folyamat pedig P művelettel kezdi a végrehajtást

Erőforrás.

- Az erőforrás egy logika objektum
- Bármelyik folyamat lefoglalhatja és felszabadíthatja
- Egyszerre csak egy folyamat használhatja az erőforrást, azaz egy erőforrás használatára kölcsönös kizárás valósul meg
- Az erőforrásokat névvel, vagy sorszámmal azonosítjuk
- A *Lefoglal(<erőforrás>)* és *Felszabadít(<erőforrás>)* műveletek egyenértékűek egy bináris szemaforra kiadott P és V műveletekkel
- A *Lefoglal(<erőforrás>)* és *Felszabadít(<erőforrás>)* műveletek oszthatatlanok
- A szemaforhoz hasonlóan itt is a véletlenen múlik, hogy a várakozó folyamatok közül melyik tudja lefoglalni az erőforrást
- Csak az garantált, hogy legfeljebb egy folyamat foglalhatja le az erőforrást

Esemény.

- Az esemény egy pillanatszerű történés a rendszerben, amelyre folyamatok várakozhatnak
- Az esemény bekövetkezése az összes rá várakozó folyamatot továbbindítja
- Így egy **összetett precedencia** valósítható meg, amivel több különböző műveletre (amelyek különböző folyamatokban lehetnek) ugyanazt az előzményt írhatjuk elő
- Két folyamat esetén az esemény jelzése és az arra való várakozás egyenértékű egy szemaforra kiadott V, illetve P művelettel
- Több folyamat esetén azonban lényeges különbség, hogy a szemafor esetében csak egy várakozó folyamat indulhat el, ezzel szemben az esemény esetében az összes várakozó továbbindul
- Fontos különbség a szemafor, az erőforrás és az esemény között:
  - A **szemafor** és az **erőforrás** felszabadítása megőrződik az objektum állapotában, egy később érkező folyamat beléphet
  - Ezzel szemben az **esemény** esetében nem őrződik meg az állapot, ha éppen nincs várakozó folyamat, akkor nem történik semmi, a később érkező folyamat várakozni kényszerül a következő eseményig

A várakozási sorok kezelése.

- A megvizsgált módszerek nem határozzák meg, hogy a várakozó folyamatok milyen sorrendben léphetnek be a kritikus szakaszba
- Miért baj ez?
  - Nem lehet megjósolni a folyamatok sorrendjét. Ez vagy okoz problémát, vagy nem
  - Lehet, hogy szeretnénk prioritást rendelni a folyamatokhoz, vannak fontosabb és kevésbé fontos folyamatok, ez a módszer ezt nem tudja érvényesíteni
  - Az viszont komoly probléma, hogy nem lehet garantálni, hogy minden várakozó folyamat véges idő alatt beléphet
- Mi lehet a megoldás?
  - A várakozó folyamatok sorbaállítása
  - Várakozási sorokat rendelünk a szemaforokhoz és az erőforrásokhoz
  - Ezeket a sorokat meghatározott algoritmus szerint kezeli a rendszer

– Lehetséges sorrendek:

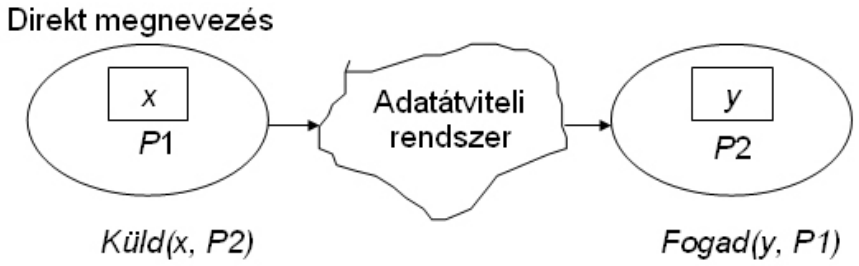
* **FIFO** (First In – First Out): Az először sorbaálló lép tovább először, más szóval érkezési sorrend
* **LIFO** (Last IN – First Out): Az utoljára érkező mehet tovább először (verem). Ebben az esetben ez nem célszerű
* **Prioritás szerint**: Minden folyamathoz rendelünk egy fontossági szintet (prioritás). Ha két folyamat várakozik, akkor a magasabb prioritású (fontosabb) léphet tovább először

2.4. Folyamatok kommunikációja

Folyamatok kommunikációja.

- A folyamatok együttműködésének egyik formája a közös memória használata – *ezt vizsgáltuk eddig*
- A másik módszer az üzenetváltásos együttműködés, azaz a folyamatok közötti kommunikáció – *ezt vizsgáljuk most*
- Akkor került előtérbe, amikor realitássá vált a számítógépek közötti kommunikáció (hálózatok)
- Amíg egy gépen fut a két folyamat, addig egyszerűbb közös memórián keresztül kommunikálni, ha külön gépen futnak, akkor ez már nem lehetséges
- A folyamatok közötti kommunikáció alapsémája a következő ábrán látható

A kommunikáció alapsémája.



7. ábra. A kommunikáció alapsémája

A kommunikáció alapkérdései.

Tudjuk, hogy a logikai processzornak van két művelete: *Küld* és *Fogad*, ezek működése azonban felvet néhány fontos kérdést:

- Hogyan nevezzük meg a partnert?
- A partnert látjuk, vagy egy közvetítőt, csatornát, postaládát?
- Egyszerre egy partner kapja meg az üzenetünket, vagy több is?
- Csak egy partnertől várhatunk üzenetet egy adott pillanatban, vagy többektől is?
- Hogyan működik a *Küld* művelet? Amikor befejeződött a küldés, akkor meg is érkezett a célhoz, vagy csak egy közbülső tárolóba került (postaláda)
- Honnan tudjuk meg, hogy megérkezett-e (nyugtázás)?
- Hogyan működik a *Fogad* művelet?
- Kell-e visszaigazolást küldenünk, vagy elintézi a rendszer automatikusan?

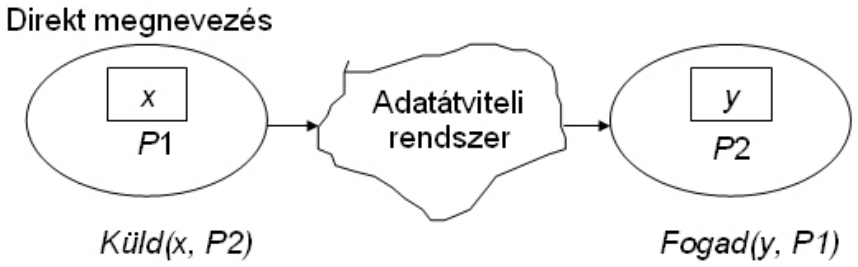
A partner megnevezése.

A partner megnevezése szempontjából a következő alaptípusokat különböztetjük meg:

1. közvetlen (direkt)
2. közvetett (indirekt)
3. asszimetrikus
4. csoportkijelölés
5. üzenetszórás

Közvetlen (direkt) kommunikáció.

- Két folyamat között zajlik
- Mind a *Küld*, mind a *Fogad* művelet megnevezi a partner folyamatot
- Például
  - P_1 elküldi a saját címtartományában tárolt x változót P_2-nek
  - P_2 eltárolja a kapott értéket a saját y változójában
  - ha a változók közös címtartományban lennének, akkor ez megfelelne az $y := x$ értékadásnak, így azonban kommunikációs műveletekre van szükség

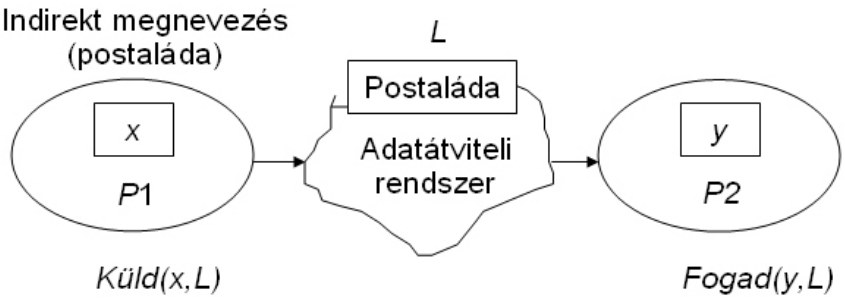


8. ábra. Közvetlen (direkt) kommunikáció

Közvetett (indirekt) kommunikáció.

- Két folyamat között zajlik
- De a felek nem egymást nevezik meg, hanem egy közvetítő közeget (pl. postaládát, vagy csatornát)
- A **postaláda** egy általában véges, de elméletileg esetleg korlátlan befogadóképességű, az üzenetek sorrendjét megtartó (FIFO) tároló, amely a *Küld-Fogad* (betesz-kivesz) műveletpárral kezelhető
- A *Küld(<cím>, <postaláda>)* művelet a saját memóriában lévő üzenetet a postaláda következő szabad tárolóhelyére másolja. Ha a postaláda tele van, akkor várakozik
- A *Fogad(<cím>, <postaláda>)* művelet a postaládában legrégebben várakozó üzenetet kimásolja a saját memóriában lévő címre és felszabadítja a helyét
- A **csatorna** olyan kommunikációs objektum, amelyik két folyamatot kapcsol össze
- A csatorna lehet:
  - egyirányú (szimplex)
  - osztottan kétirányú (félduplex), azaz egyidejűleg egyirányú, de az irány változtatható
  - kétirányú (duplex)
- A csatorna tartalmazhat 0, véges, vagy végtelen kapacitású átmeneti tárolót
- A postaláda és a csatorna lazítja a folyamatok közötti csatolást

- Olyan folyamatok is kommunikálhatnak egymással, amelyek nem ismerik egymást
- Időben is eltávolodhatnak egymástól a kommunikáló folyamatok



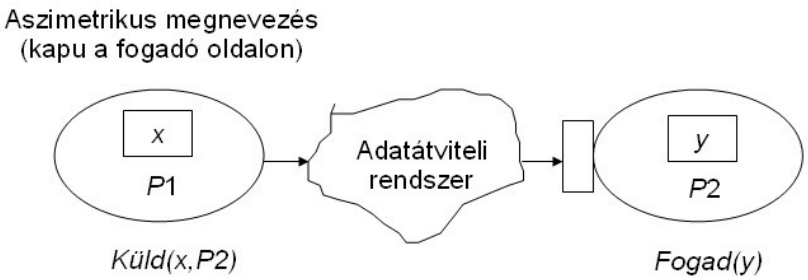
9. ábra. Közvetett (indirekt) kommunikáció

Asszimetrikus kommunikáció.

- Az egyik folyamat (az adó, vagy a vevő) megnevezi, hogy melyik folyamattal akar kommunikálni, a másik viszont egy saját be-/kimeneti kaput (port) használ
- Ha csak egy portja van, akkor nem is kell megnevezni
- Ha a vevő használ bemeneti kaput, akkor a műveletek alakja: *Küld(<cím>, <folyamat>), Fogad(<cím>)*
- Ez akkor hasznos, ha a fogadó nem ismeri a küldőt, de a küldő ismeri a fogadót. Például egy ügyfél küld egy kérést egy szolgáltató folyamatnak (kliens-szerver modell)
- Az ellentétes irányú asszimetriára példa lehet egy feladatokat generáló és az azokért versengő végrehajtó folyamatokból álló rendszer (farmer-worker modell)

Csoportkommunikáció és üzenetszórás.

- Az üzenet küldője folyamatok egy csoportját nevezi meg
- Egyetlen üzenetküldő művelettel a csoport összes tagja megkapja az üzenetet

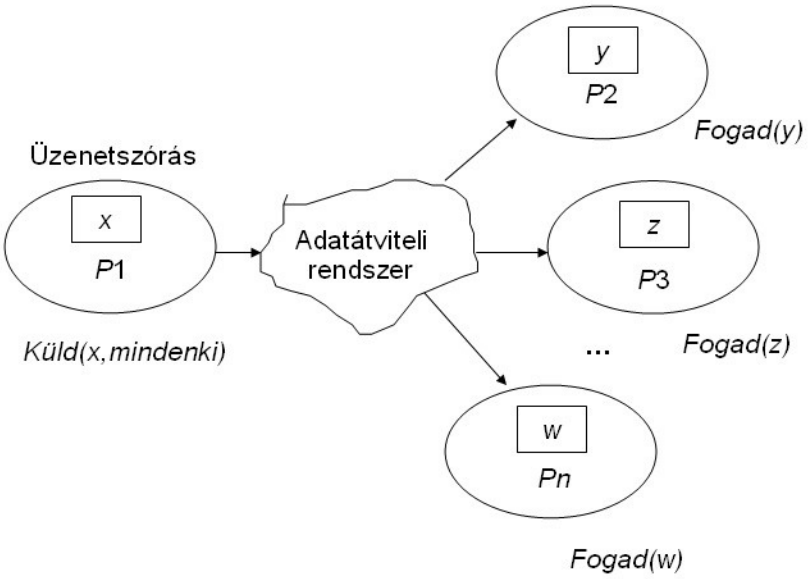


10. ábra. Asszimetrikus kommunikáció

- Az üzenetszórás ennek speciális esete, amikor az üzenet a rendszer összes tagjához eljut, címzés nélkül
- Lényegesen egyszerűsíti a kommunikációt, amikor több címzetthez kell eljuttatni az üzenetet
- Vannak olyan fizikai kommunikációs közegek, ahol a csoportkommunikáció nagyon hatékonyan megvalósítható, pl. sín-topológiák, rádiókommunikáció

A kommunikáció és a szinkronizáció kapcsolata.

- Milyen kapcsolat van a folyamatok szinkronizációja és kommunikációja között?
- A kommunikáció megkövetel bizonyos szintű szinkronizációt, hiszen csak azok a folyamatok tudnak kommunikálni egymással, amelyek valamilyen mértékben össze vannak hangolva
- A kommunikáció típusa meghatározza, hogy milyen szigorú szinkronizációra van szükség:
  - **Tárolás nélküli átvitel** esetén
    - szigorú szinkronizációra van szükség (egyidejűség)
    - a *Küld* és *Fogad* műveletek randevúja történik
  - **Véges kapacitású tároló**
    - bizonyos határok között kiegyenlíti a küldő és fogadó folyamatok sebességingadozásait
    - a fogadó folyamat várakozik, ha üres a tároló
    - a küldő folyamat várakozik, ha megtelt a tároló



11. ábra. kommunikáció

  - egy adott üzenet elküldése meg kell, hogy előzze a fogadását, ezért egy sorrendi szabály (precedencia) érvényesül a két folyamat között
  - a tároló egy adott elemének kezelésére kölcsönös kizárás vonatkozik

- **Végtelen kapacitású tároló**
  - csak a modellben létezik, valóságban nem
  - a küldő folyamatnak soha nem kell várakoznia
  - ezenkívül a véges tárolónál elmondottak érvényesek

3. Befejezés

3.1. Emlékeztető kérdések

Emlékeztető kérdések.

1. Mit jelent a megszakítás? Mire használjuk, hogyan működik?
2. Milyen megszakítások léteznek?

3. Mit nevezünk rendszerhívásnak? Hogyan működik?
4. Hogyan működnek a szinkron és aszinkron rendszerhívások?
5. Mit jelent a folyamat és a szál? Mi a hasonlóság és mi a különbség közöttük?
6. Mit nevezünk vezérlési szálnak?
7. Mi az a folyamatmodell?
8. Hogyan modellezzük a folyamatokat?
9. Mit nevezünk RAM-modellnek?
10. Mit nevezünk a folyamat állapotterének?
11. Miért használunk folyamatokat?
12. Mi a hátránya a folyamatok használatának?
13. Milyen kapcsolatban állhatnak egymással a különböző folyamatok?
14. Hogyan jönnek létre és szűnnek meg a folyamatok?
15. Milyen rendszereket nevezünk statikusnak, illetve dinamikusnak?
16. Mit nevezünk folyamat fának?
17. Mit nevezünk hierarchikus, illetve globális erőforrás-gazdálkodásnak?
18. Hogyan működhetnek együtt a folyamatok?
19. Hogyan működnek együtt a folyamatok közös memória segítségével, illetve üzenetküldéssel?
20. Mit nevezünk PRAM-modellnek?
21. Mi értünk folyamatok szinkronizációja alatt?
22. Mi az a kölcsönös kizárás?
23. Mit nevezünk kritikus szakasznak?
24. Mit nevezünk randevúnak (a folyamatok esetében)?
25. Mit nevezünk precedenciának?
26. Hogy működik a foglaltságjelző bit? Mire használjuk?
27. Mit csinál a Peterson-algoritmus? Hogyan működik?
28. Mit csinál a Bakery-algoritmus? Hogyan működik?
29. Mit nevezünk kiterjesztett PRAM-modellnek?

30. Mit nevezünk szemafornak? Hogyan működik?
31. Hogy működik az erőforrás és az esemény, mint szinkronizációs eszköz?
32. Hogyan kezelhetjük a várakozási sorokat?
33. Mit nevezünk a folyamatok közötti kommunikáció alapsémájának?
34. Milyen alapkérdéseket kell megválaszolni a folyamatok közötti kommunikáció megértéséhez?
35. Milyen kommunikációs alaptípusokat különböztetünk meg a partnerek megkülönböztetése szempontjából?
36. Mi jellemzi ezeket?
37. Milyen kapcsolat áll fenn a folyamatok közötti kommunikáció és szinkronizáció között?

Befejezés.

Köszönöm a figyelmet!