Informatika 1 – Operációs rendszerek

Előszó

A jegyzetben sajnos hibák előfordulhatnak – én is csak ember vagyok – de ha találtok is, akkor nyugodtan írjatok egy mailt a negedes@freemail.hu címre, és kijavítva, újra feltöltöm. Figyelem! A jegyzet helyenként pszeudo kódot tartalmaz. Ez nem létező programozási nyelv, célja a kód könnyebb megértése, ezért tanárok előszeretettel használják. (Informatika 2 tantárgyból visszaköszön.) A jegyzetben az = az értékadó, az == a feltételvizsgáló egyenlőségjel. Használjátok egészséggel, de csak saját felelősségre!

Történelmi áttekintés

A második világháború idején a hadiipar használt először számítógépeket ballisztikai célokra, binárisan kódolt lyukkártyán vagy lyukszalagon adták a gépeknek az információkat. Kötegelt feldolgozást használtak (*batch*), vagyis a feladatokat (*job*-okat) egyszerre adták meg, és azokat sorban, önállóan oldotta meg a számítógép az ember beavatkozása nélkül. Akkor szükség volt operátorokra, olyan emberekre, akik a programozó által meghatározott munkamenetet végrehajtották, fizikailag kezelték a gépet. Gyakorlottságuk gyorsította a gépkezelést (*closed shop*), de nem volt hibakeresési funkció. (A programozó adta meg, mekkora memória kell, mennyi időre van szükség, vagy milyen fordítóprogramot kell használni.) Az operátorok „automatizálásához” egy állandóan futó programra volt szükség, ez a *monitor*, adathordozóra lehetett írni a munkafolyamatokat, paramétereket, a monitor ezeket használta fel.

Az összetettebb számítógépek bonyolultabb operációs rendszereket használtak, megjelentek az operációs rendszerek extra funkciói: memória és egyéb védelmek:

- a monitor állandóan futott, így nem volt megengedhető, hogy kár érje futás közben vagy után, ennek okán megjelentek a ferrites memóriák, mágneses adathordozók, melyek akár kikapcsolás után is megőrizték a biteket
- a memóriában futó programok elkülönítésre kerültek
- a monitornak hardver szempontjából extra jogai lettek a többi program felett, melyeket a *privilégiumszintek* határoznak meg

A processzorok fejlődésével a lyukszalagos I/O egységek lettek a számítógép leggyengébb „láncszemei”, hiszen mechanikailag korlátozott volt az adatbevitel sebessége. Ez kihasználatlan processzorteljesítményhez vezetett, így szükséges volt az I/O egységek fizikai gyorsabbá tételére. Erre külső, úgynevezett *szatellit processzorokat* használtak. A szatellit processzorok kezdetben a számítógépektől külön egységekben voltak, feladatuk a lyukszalagos információk mágnesszalagra történő átírása volt, ezek voltak az *off-line I/O* műveletek. A mágnesszalagon jutott az információ a nagy teljesítményű számítógéphez, ami ezen adta vissza az adatot a szatellit processzornak, ami szintén lyukszalagra írta azt vissza. A legnagyobb hiányossága a számítógépeknek így az lett, hogy a mágnesszalagokhoz külön program tartozott, és ez számítógépenként különbözött, így szükségessé vált a mágneses hordozók gépek közötti problémamentes átviteléhez ezen programok nagyfokú hasonlósága. Az IBM így gépek helyett gépcsaládokat kezdett gyártani, melyeken belül a programok is kompatibilisek lettek.

Most már az I/O egységek nem csupán fizikai egységek voltak, elváltak egymástól a fizikai és logikai perifériák, a program felé a rendszer adatcsatornákon küldte az adatot. A perifériák minden gépre egyedi módon voltak illesztve. Az operációs rendszer kezelte az írást

és olvasást, a program csak az adatot kapta kézhez, már nem volt beleszólása a perifériák irányításába, így a rendszer ellenőrizhette a programok hibáit (például nyomtató esetén korlátozhatta a rendszer a kinyomtatott lapok számát).

A lassító tényező a mágnesszalagok fizikai hordozása és cseréje lett, így összekötötték a szatellit processzort a géppel: hálózati kapcsolatban voltak, vagy egyetlen közös disc-re írtak FIFO elven. Kialakult a *spooling* rendszer. A spooling elvette az adatokat, eltárolta, és az adott periféria ütemében adta vissza. Így a főprocesszort csak az adattárolás idejére kötötte le a folyamat. (Például egy nyomtató esetén a nyomtató saját ütemében nyomtat, olyan, mint ma, csak ekkor ez nem a nyomtatóba volt beépítve.)

Miután a processzorok sebessége gyorsabban nőtt a perifériák sebességének növekedésénél, egyre több volt a várakozási idő, ezért megjelentek az *interaktív* rendszerek (több felhasználós rendszerek). A felhasználó a számítógép előtt ült és kommunikált vele. Ez több program egyidejű futtatását tette lehetővé, ez a *multiprogramozás*, látszólag több program fut egyszerre, valójában a számítógép egy időpillanatban csak egy program utasításait hajtja végre, de a programok között gyorsan vált, hol az egyiken, hol a másikon dolgozik. Ezért tűnik úgy, mintha több program futna egyszerre. A multiprogramozáshoz szükséges:

- a programok állapotainak elmentése
- *time-sharing*: időosztásos programműködés használata

A multiprogramozás által sokkal bonyolultabbá vált a rendszer, a programok közötti átkapcsolások problémája, ezek nyilvántartása, a memóriába történő elhelyezése is gondot jelentett. Ezen kívül fontos szerepet kapott:

- ütemezés kérdése: programok futási sorrendjének meghatározása, prioritások létrehozása, a programok egyenlőtlen erőforrásainak optimalizálása
- a programok a memóriáért versenyeznek: erőforrás-menedzsmentre van szükség
- ráhívás: amíg az egyik program egy adatátviteli műveletet folytat az I/O egységeken, addig a processzor más utasításokat hajt végre, melyben egy másik programnak ad lehetőséget ugyanazon I/O egység használatára – a megoldás a várakozási sor felállítása
- *kölcsönös kizárás*: egy I/O egységet nem használhat egyszerre több folyamat (különben a másodszorra érkező program definiálatlan, inkonzisztens állapotban találja azt)
- *holtpontra jutás*: több program használja ugyanazokat az I/O egységeket, és míg az egyik program az egyik egységet, a másik program a másikat foglalja le, mindkettő megakad a futásban, mert mindkettő kölcsönösen vár a másik I/O egységének felszabadulására

A mai formában a felhasználói programok a hardverekhez közvetlenül nem férhetnek hozzá, csak az operációs rendszerek funkcióit használhatják: az op. rendszer elrejti a felhasználó elől a hardvert.

Ha nem elég egyetlen számítógép egy folyamathoz, multiprocesszoros rendszerre van szükség. Több processzor található egy szekrényben, vagy akár egyetlen chipben is, fizikailag párhuzamosan működnek, de egymással is kommunikálnak. Itt már ténylegesen több program fut egyszerre.

A folyamat fogalma

A folyamat valójában működés közben lévő, vagyis elkezdett, de be nem fejezett program. Multiprogramozott esetben egyszerre egy folyamat zajlik, bár az egymás után gyorsan váltott programok miatt kívülről úgy tűnik, hogy párhuzamos a programvégrehajtás.

- kötegelt feldolgozás (batch): job-ok (munkafolyamatok) feldolgozása történik, az egész program egyszerre kerül beolvasásra, a számítógép kiadja a megoldást, vagy a hibalistát (az utóbbi a *dump*)
- interaktív, időosztásos rendszer: egyszerre több felhasználó is használhatja a számítógépet, több perifériáról, terminálról, hiszen sokkal gyorsabb a számítógépek műveletvégzése, mint az emberi gépelési sebesség, így a CPU kihasználatlan maradt; a problémát az jelentette, hogy a memóriába nem fért be túl sok felhasználó programja
- valós idejű rendszerek:
    - az IC-k elterjedésével a szabályozási folyamatok vezérlését ilyenek látják el
    - az időzítés komoly szerepet kap, valós idejű irányítási problémákat old meg, így az ütemezésnek is szinkronban kell lennie az idővel (a válaszidő-követelményeknek meg kell felelni: nem mindegy, hogy egy tartályban 1 vagy 30 másodperc alatt csökkentem a nyomást)
    - látszólag párhuzamos programjai a task-ok (feladatok)

A folyamat a job-ok és task-ok összefoglaló neve, multiprogramozott és multiprocesszoros esetben is ugyanaz a folyamat fogalma.

Az operációs rendszerek „felületei”

Az operációs rendszerek a kapocs az alkalmazások és a harver között, de a felhasználók is javarészt az operációs rendszerekkel érintkeznek.

Felhasználói felület

A felhasználók csoportosítása:

- naiv user (mezei felhasználó): számítástechnikai ismeretekkel nem rendelkezik, géphasználata során az alkalmazásokat látja
- alkalmazás fejlesztő: ismeri a számítástechnikát, az operációs rendszerből szeretne többet látni
- rendszergazda: feladata az adott számítógépes rendszer optimalizálása, ehhez persze ismernie kell, és megkülönböztetett jogosultságokkal kell rendelkeznie

A felhasználói felület osztályozása:

- karakteres (parancssori) bevitel és megjelenítés:
    - hátrány: áttekinthetetlen, ékezetes karakterek gondot okozhatnak
    - előny: alacsony sávszélességű távoli helyről is stabilan működtethető, parancssorban akár batch fájl is összeállítható, mely programként működtethető
- grafikus: egyszerű, áttekinthető bevitel és megjelenítés (ábrák kirajzolhatók)

Hardware felület

A régi számítógépek felépítése jobban hasonlítottak egymásra, mint a maiak, ezért az operációs rendszeren belül kialakítottak egy részt, mely a különböző hardverek kompatibilitását oldotta meg (Hardware Abstraction Layer), az operációs rendszer többi része pedig teljesen egységes lehet. A BIOS beégetett program, mely a számítógép bekapcsolásakor a perifériás eszközök és az operációs rendszer illesztését biztosítja. A számítógépek széleskörű alkalmazása (a katonai számítógépektől a mobiltelefonokig) megköveteli a különböző operációs rendszerek használatát.

Alkalmazás felület

A monitor kialakulása óta az operációs rendszert védjük a felhasználói programoktól, mert ha olyan memóriaterületet használ, mely a monitort tartalmazza, az veszélyezteti a rendszer működését. Az is probléma lehet, ha két program egymás adataiba kotor bele, előfordulhat, hogy egy program „túlolvas", így egymástól is védeni kell őket, nem is beszélve az I/O védelemről: egy másik program bejövő adatait is beveszi a saját folyamatához.

A védelem abból áll, hogy eltérő jogokat adunk: rendszermódban „mindent szabad", felhasználói módban pedig a privilegizált utasításokat nem szabad végrehajtania a processzornak. Továbbá a megszakításokat is csak az operációs rendszer kezelheti. (Hardveresen az operációs rendszert is megszakítás hívja meg.)

Az operációs rendszer és az alkalmazások is eljárásokból állnak, a rendszerhívásaik között nincs különbség, de az operációs rendszereknél a processzort rendszermódba kell kapcsolni, alkalmazások esetén pedig felhasználói módba.

Az operációs rendszereket és alkalmazások között laza csatolásra törekszünk, hogy ne kelljen az alkalmazásnak nagyon függnie a rendszertől. Az op. rendszer sokszor el van bújtatva az eljáráskönyvtárban vagy a nyelvek utasításaiban. (Például C++ nyelv esetén az I/O műveletek.)

A folyamat fajtái

A folyamat műveletekből, lépésekből áll, lényegében ezek egymásutánisága. Ezért egy folyamatot úgy képzelhetünk el, hogy egy (virtuális) processzor és memória hajtja végre ezeket a lépéseket. Ez a memória adatot, programot és stack-et tartalmaz, a stack az átmeneti változók tárolását szolgálja elsősorban. A processzor állapotát regiszterek tárolják (programszámláló, akkumulátor, adatcímeket tartalmazó regiszterek, flag-ek).

Ha egy multiprogramozott esetet vizsgálunk, ott több folyamatunk van, ezért a modellünkben minden folyamathoz tartozik memória és processzor. Ezek egymástól eltérő, véges sebességgel működnek aszinkron módon, ezért nem tudjuk, hogy melyik gyorsabb, illetve lassabb. Ezt hívjuk *folyamatrendszernek*. Fontos hangsúlyozni, hogy ez egy virtuális modell a folyamatrendszer elképzelésére, a fizikai valóságban (multiprogramozott esetben) továbbra is egy processzor és egy memória van! A virtuális processzorok úgy jelennek meg, hogy a fizikai processzort időben, a valós fizikai memóriát pedig térben és időben is felosztjuk.

Folyamatok viszonyai:

- független folyamatok: nem ismerik, nem látják egymást, nem is törődnek egymással
- versengő folyamatok: logikailag ugyan függetlenek, de ugyanazon a fizikai rendszeren kerülnek végrehajtásra, vagyis azonos az erőforrás-készletük (a kötegelt és időosztásos rendszerek is ilyenek)
- együttműködő folyamatok: a rendszertervező eldöntötte, hogy párhuzamosan végrehajtható komponensek is vannak, a helyes működést segítő információk vezérlő jelek vagy adatok formájában adhatók át a folyamatok között

Együttműködő folyamatok viselkedése

Kétféle módon cserélhetnek információt folyamato: közös memórián keresztül, vagy közös I/O „szférán" át. Az előbbit *közös memóriás együttműködésnek* nevezzük. Az utóbbi az *üzenet alapú együttműködés*, ez esetben egy közös adathálózatra kapcsolódnak az I/O egységek, a processzorok üzeneteket küldenek egymásnak (*send* és *receive* műveletek).

Közös memóriás együttműködés

Minden virtuális processzorhoz tartozik egy virtuális saját lokális memória és az összes processzorhoz tartozik egyetlen közös globális memória. Mindkét fajta memóriához azonosak a műveletek:

- read: a cím bemenő paraméter, az adat kimenő paraméter
- write: a cím és az adat is bemenő paraméter

A processzorok által küldött utasítások nem gabalyodhatnak össze a globális memóriában, ezért szükséges egy bizonyos *kapcsoló hálózat*, mely ezeket szétválasztja.

A RAM (Random Access Memory) memória: lineáris címzésű, rekeszelhető, működik az írás és olvasás művelet, íráskor a régi tartalom fölülíródik, de olvasáskor a régi tartalom nem törlődik, tehát az olvasás nem rombolja az írott tartalmat.

A globális memória egyfajta RAM, de a közös hozzáférés miatt szükség van néhány kiegészítésre, hiszen felléphetnek *ütközések*:

- read és read művelet ütközik: ha a közös rekesztartalom több helyre megy, akkor több olvasó kapja vissza a rekesz tartalmát
- write és read ütközik: ilyenkor „egyszerre” van írás és olvasás, az aszinkron működés miatt nem tudjuk garantálni, hogy a read művelet csak a write előtti vagy csak a write utáni állapotot adja vissza, elképzelhető, hogy valami közbülső „szemetet” kapunk
- write és write ütközik: ha egyszerre kerül két write a sínre, elképzelhető, hogy az „összeéseli” vagy „összevagyolja” az adatokat, vagyis interferálnak az adatok

Ezek kiküszöbölésére a műveleteket sorba kell állítani! Ennek a legjobb módja az, ha az írás és olvasás művelet *atomi, oszthatatlan*, ha ezt el tudjuk érni, akkor garantáltan elkerülhetők az ütközések. Az ilyen memória a PRAM (Pipe-lined, Parallel vagy Párhuzamos, de semmiképp sem „Programozható”, mint a PROM esetében, nem összetévesztendő!) A PRAM fizikai megvalósítása a fent említett kapcsoló hálózat révén történik: ez olyan sín a globális memória előtt, ami biztosítja, hogy egyszerre csak egy művelet érkezhessen be. Ha úgy tetszik, sorba állítja a műveleteket. (A groteszk valóság az, hogy a tananyag azon része következik, mely nehezebben tárgyalható gépelt jegyzet formájában, de előadáson viszonylag könnyen megérthető.)

Vegyünk egy egyszerű alapesetet. Két folyamat működik, az egyik P_i, a másik P_j. P_i folyamat egy „x” memóriacímű rekeszbe állandóan egy „e” értéket ír, P_j pedig ciklikusan kiolvassa P_i behelyezett értékeit. Korábban említettük, hogy a folyamatok működése aszinkron, ezért nem tudjuk, hogy melyik folyamat a gyorsabb. Így azt sem tudjuk, hogy az x memóriarekesz olvasásakor az új adatot beírta-e már P_i, de az is előfordulhat, hogy P_i többször újraírja a rekeszt P_j olvasása előtt, vagy P_j többször kiolvassa azt az új adatok érkezte előtt.

A megoldást egyfajta *szinkronizáció* adja: hozzunk létre egy második rekeszt, egy „s” változót, mely 0 vagy 1 értéket vehet fel. Ha P_i beírta az adatot x-be, akkor utána az s értékét 0-ról 1-re változtatja. A P_j csak akkor olvashat ki a memóriarekeszből adatot, ha az s értékét 1-nek látja. Miután kiolvasta, visszaállítja azt 0-ra, ami jelzi a P_i folyamatnak, hogy új adatot írhat az x memóriarekeszbe. Tehát az s változó úgy működik, mint váltófutásnál a váltóbot. (Aki valamennyire ismeri a Digitális technikákat, az tudja, hogy ilyen könnyen nem ússzuk meg a dolgot, lehet, hogy megoldunk egy-egy problémát, de úgyis újak vetődnek föl, melyeket újra meg kell oldani. ☺)

Bonyolítsuk a modellünket: mi van akkor, ha nem egy, hanem n darab folyamat olvas ki párhuzamosan adatot egy x és x+m címtartományú memóriából definiálatlan időben? Ha az egyik olvasó hamarabb olvassa ki, mint ahogy az író folyamat végig írta volna az m+1 darab

rekeszt, félig jó, félig rossz adatot olvasunk ki. Ezért valahogy azt kell megoldani, hogy írás közben ne legyen olvasás, illetve ha olvasás történik, akkor várja meg az író, míg azt befejezi.

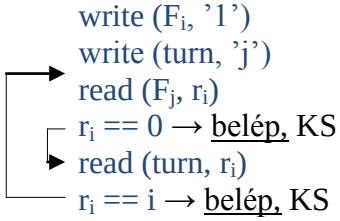
A megoldás a *kölcsönös kizárás*. Definiáljuk a *kritikus szakaszt* úgy, mint egy folyamat azon részét, mely más folyamat kritikus szakaszaival nem eshet egybe. (Például ez esetben egy memóriatartomány írása és olvasása.) Hogy biztosítsuk, hogy a kritikus szakaszok ne ütközhessenek, egy jelző mechanizmust kell megvalósítani. Ha fut egy folyamat egy kritikus szakasza, ez mutatja, hogy nem indulhat új kritikus szakasz, és várakozásra készteti a többi folyamatot. Lényegében egy foglaltsági bittel megoldható a probléma: ha 0 értékű, szabad olvasni, ha 1 értékű, akkor tilos. Ez elméletileg működne, de fizikailag előfordulhatnak hibák: egy P_i folyamat beolvassa a foglaltsági bitet r_i regiszterébe, majd elvégez egy vizsgálatot, hogy az értéke milyen. Ha a P_i vizsgálata alatt egy P_j folyamat szintén beolvassa a foglaltsági bitet az r_j regiszterbe, ha az 0 értékű volt, mindketten szabadnak találják. Lehet, hogy P_i átírja 1-re, de addigra már a 0 értékű foglaltsági bit P_j regiszterében van. Ezért fogják mindketten szabadnak találni a bitet, és létrejön az ütközés.

A megoldás erre: az egész műveletsorozat oszthatatlanná tétele: a beolvasástól a vizsgálaton át a foglaltsági bit írásáig. Az elvi megoldáshoz hozzátartozik, hogy meg kell oldani a folyamatok véges időn belüli belépését, illetve nem lehetnek lyukak a folyamatok között (folyamatok közvetlenül követik egymást), és csak azokra szabad működnie, akik be akarnak lépni. A kölcsönös kizárás elvi megoldására több algoritmust tárgyalunk, majd megmutatjuk a mérnöki, gyakorlatias megoldást.

Peterson-algoritmus

A P_i és P_j folyamathoz egyaránt tartozik egy F_i és F_j flag, mely 0 vagy 1 értéke azt mutatja, hogy akar-e műveletet végrehajtani a közös memórián. Létezik továbbá egy turn változó, mely a két folyamat valamelyikére mutat. Ha P_i be akar lépni, F_i flag-et 1-re állítja, a turn változót pedig j-re állítja. Beolvassa az r_i regiszterbe F_j tartalmát, és vizsgálattal megnézi, hogy milyen az értéke. Ha azt látja, hogy P_j nem akar belépni, vagyis 0 a flag, akkor belép, ha 1-et lát, akkor beolvassa a turn tartalmát az r_i regiszterbe. Ha a turn i-t mutat, az azt jelenti, hogy P_j lassabb volt, ezért belép. Ha j-t mutat a turn, az azt jelenti, hogy P_i volt a lassabb, ezért várakozik, vagyis ciklikusan beolvassa F_j értékét. Ha egy folyamat kilép, a saját flag-jét 0-ba állítja. Az algoritmus hibája, hogy csak két folyamat esetén biztosítja a kölcsönös kizárást. (KS = kritikus szakasz, → igen ág, nem ág a feltételvizsgálat után, read, write és a feltételvizsgálat külön-külön atomi műveletek.)

Belépés:



Kilépés:

write (F_i, ’0’)

Bakery algoritmus

Biztosítja n folyamatra a kölcsönös kizárást. Ez nagyban hasonlít az orvosi ügyelet sorszámozási rendszeréhez. Minden folyamathoz tartozik egy szám: ha ez a szám 0, akkor nem óhajt bemenni, ha 0-tól különböző szám, akkor az maga a sorszám. Az új belépők a legnagyobb sorszámnál eggyel nagyobb értéket adják maguknak. Minden belépő elölről vizsgálja a többiek számait, ha 0, akkor átugorja, ha a sajátjánál kisebb, akkor kivárja, míg 0 nem lesz. Ha az egész sorozatot végigolvasta, és nem talált önmagánál kisebbet, akkor tudja, hogy őt szolgálják ki.

A gond az, hogy a számsorozat végigpörgetése némi időbe telik, és előfordulhat, hogy azonos sorszámot ad magának több új belépő. Ez kiküszöbölhető, ha minden sorszámhoz tartozik egy bit, ami jelez, ha valaki éppen sorszámot választ magának: ilyenkor letiltja a többiek belépését. A beérkező beállítja a jelző bitet 1-re, jelezve, hogy több érkező nem jöhet be. Miután megkapta a sorszámát, ezt visszaállítja 0-ra. (Hogy első féléves sebeket szakítsak fel, halkan megjegyzem, hogy az ötvenes években született egy bizonyos Dijkstra-algoritmus is, mely másfél oldal hosszú, és n folyamatra megoldja a kölcsönös kizárást, de szerencsére a tanszék nem annyira elvetemült, hogy ezt is leadja. ☺)

Gyakorlati megoldás

A gyakorlatban egy egyszerű módon kezeljük a problémát: további oszthatatlan, atomi műveleteket definiálunk. Ezek a TestAndSet illetve Exchange műveletek.

- TestAndSet: olyan atomi függvény, ami visszaadja egy flag értékét, miközben azt foglaltra állítja.

  ```
  read (F, r)
  write (F, ’foglalt’)
  ```

- Exchange: oszthatatlanul megcseréli két memóriarekesz tartalmát. Szükséges hozzá globális és lokális flag a globális és lokális memóriákban (GF, LF), és egy flag típusú átmeneti változó (TMP).

  ```
  read (GF, r)
  write (TMP, r)
  read (LF, r)
  write (GF, r)
  read (TMP, r)
  write (LF, r)
  ```

Ha függvényként használjuk ezeket az atomi műveleteket (a skip lényegében üres utasítás):

```
while TestAndSet(F) == ’foglalt’ do skip
    KS
F = ’szabad’
```

Az Exchange utasítással:

```
LF = ’foglalt’
XCHG(LF, GF)
LF == ’szabad’ → KS
```

Érdemes tudni, hogy egyprocesszoros esetben a párhuzamosságot a beérkező megszakítások hozzák létre, ugyanis ezektől vált a processzor a folyamatok között. Az oszthatatlan műveleteket a megszakítások letiltásával (maszkolásával) hozhatjuk létre. (Assembly kód esetén van rá megfelelő utasítás, mely megvalósítja a TestAndSet-et.)

A legnagyobb hiányossága ennek a gyakorlati megoldásnak, hogy azt a követelményt, mi szerint a kérőknek véges időn belül kell belépniük, nem garantálja. Elvileg elképzelhető, hogy az idők végezetéig körözik egy utasítás, mert mindig egy másik gyorsabb vagy szerencsésebb. Ami nagyobb gondot okoz, az a helyben járó ciklus pazarló jellege: egy folyamat körbe-körbe jár, foglalja a sínt, míg egyszer észre nem veszi, hogy egy flag bebillent szabad állapotba: ez a *busy waiting* – a processzor futtatásával történő várakozás. Ezeket a problémákat nemsokára megoldjuk, de előtte kell egy kis kitérő a megértéshez.

Többféle szinkronizációt tárgyaltunk. A memóriák esetén láttuk, hogy sorrendiségre van szükség, *precedenciára* (váltóbot), hogy ne akadjanak össze, ez az egymásutániságról szól. Nagy adatoknál szükség volt kölcsönös kizárásra, mely szerint egyszerre csak egy kritikus szakasz futhat. Adja magát, hogy kell lennie egyidejűségnek (*randevú*) is, mely szerint a folyamatoknak együtt kell történnek. Ugyan ez a memóriák esetén nem jön elő, adatok küldésénél és fogadásánál igen: szükséges úgy szinkronizálni a küldést és fogadást, hogy egyszerre érkezzenek be az adatok. Ami mindhárom problémát egyszerre oldja meg, az a *szemafor*. (Érdemes megjegyezni, hogy ez is Dijkstra nevéhez fűződik. ☺)

Szemafor

A szemafornak van egy integer „s” változója, mely kezdőértékkel rendelkezik. Legyen egy P atomi műveletünk, mely üres tevékenységet folytat, ha $s \leq 0$, ha nagyobb 0-nál, akkor dekrementálja az s változót. Legyen egy V atomi műveletünk, mely inkrementálja s-t. A szemafor definíciója:

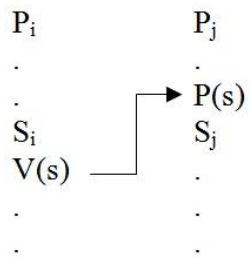
```
        var s:integer = k;
P(s):   begin
            while s ≤ 0 do skip
            s = s-1;
        end
V(s):   s = s+1;
```

Nincs más dolgunk, mint a folyamatok „védett” szakaszai elé berakni egy P műveletet, utána egy V műveletet, így egyszerre maximum annyi védett szakasz futhat, ahány s kezdőértéke. Ugyanis minden belépés a P műveleten keresztül csökkenti a szemafor értékét, ha eléri a 0-t tovább nem csökkenhet, és üres utasításokat hajt végre. Ha valaki kilép, felszabadul egy hely, az s inkrementálása ezt kimutatja, és azonnal be is lép a helyére valaki. A védett szakasz egy kibővített, általánosabb megnevezése a kritikus szakasznak.

Bináris szemafor

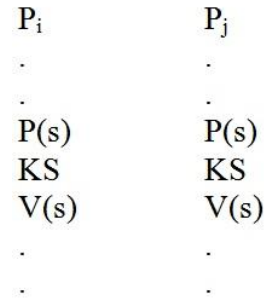
Egy olyan szemafor, mely esetén az s változó csak 0 vagy 1 értékű lehet. Ez azért hasznos, mert azzal, hogy maximum egy folyamat léphet be, kölcsönös kizárás jön létre. Valójában a bináris szemafor univerzális szinkronizációs eszköz, mert ezen kívül biztosíthatja a sorrendiséget (precedencia) és az egyidejűséget (randevú) is.

Precedencia: a P_i és P_j folyamatokban egy S_i művelet után akarjuk biztosítani egy S_j művelet létrejöttét, illetve az $S_i \rightarrow S_j$ sorrendet. Egy mindkét folyamat által olvasható s szemafor kezdőértékét 0-ra állítjuk. Az S_i művelet után egy V(s) műveletet helyezünk el, az S_j művelet előtt egy P(s) műveletet.

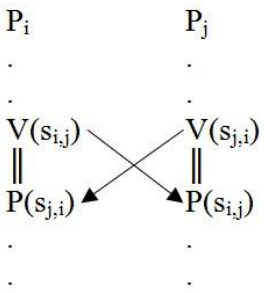


Ha az S_j művelet meg is akarná előzni az S_i műveletet, előtte P(s) művelet van és s értéke nulla. Amíg az s értéke nem nő meg, a P(s) üres utasításokat tesz, ezért S_j nem léphet be addig, míg a V(s) művelet nem állítja be s-t 1-re. A V(s) művelet pedig S_i létrejötte után következik be, ezért S_j kénytelen megvárni S_i-t. Vagyis biztosítottuk a sorrendiséget.

Kölcsönös kizárás: a kritikus szakaszok átlapolódását kell megakadályozni. Ezért a szemafor értékét 1-re állítjuk, és minden kritikus szakasz elé P(s) utasítást rakunk, utánuk pedig V(s) utasítást. Így bármelyik is kezdődik hamarabb, egyszerre nem futhatnak, mert a lassabb P(s) üres utasításokat tesz, hiszen a gyorsabb P(s) kinullázta a szemafort. Ha a kritikus szakasznak vége, a V(s) ismét 1-re állítja a szemafor érékét, ami a lehetővé teszi egy újabb folyamat számára a kritikus szakasz kezdetét.



Egyidejűség: a folyamatoknak találkozniuk kell. A programozásban az „egyszerre” fogalma nehezen értelmezhető. Úgy tekintjük, hogy a kettő folyamatszakasznak egy megadott idősávon belül kell találkoznia. Vagyis az egyik addig nem léphet ki ebből a sávból, amíg a másik be nem kerül. Az egyik folyamatnak jeleznie kell, hogy már bejött, a másiknak pedig azt, hogy kiment. Ezt kölcsönösen mutatniuk kell egymásnak. Kettő darab szemafort használunk: $s_{i,j}$ 1 értéke azt jelzi, hogy P_i folyamat megfelelő szakasza már a megkezdődött sávban van, $s_{j,i}$ 1-es értéke ugyanez a P_j folyamat megfelelő szakaszára. (Értelemszerűen a 0 érték azt jelenti, hogy még nem lépett be.) Mindkét szemafor kezdőértéke zérus. Mindkét kiemelt folyamatszakasz elé a saját magára vonatkozó V függvényt, és utána a másikra vonatkozó P függvényt helyezzük el az ábrán látható módon. Így az első folyamat, ha hamarabb is lépett a sávba, nem léphet ki addig, amíg $s_{j,i}$ értéke nem változik 1-re, vagyis addig, amíg a második is be nem lépett a sávba, és $V(s_{j,i})$ nem növelte meg $s_{j,i}$ értékét.



Most, hogy találtunk egy olyan gyakorlati megoldást, mely orvosolja az adataink összegabalyodását, visszatérhetünk a korábban tárgyalt busy waiting problémára. Talán emlékszünk még, hogy ez a processzor fölösleges terhelését jelentette. A szemafor ez ellen is felhasználható. Ehhez azonban szükséges definiálni új műveleteket:

- rendszerműveletek:
  - *sleep:* a folyamat lemond a futási jogról

- *wakeup:* felébresztenek egy alvó folyamatot, ugye nem tudja egy alvó folyamat önmagát felébreszteni, ezért kell egy függvényparaméter, mely azonosítóval látja el a folyamatot (p:p_id) id = identity

- listakezelő műveletek:
  - felfűz (L:lista, e:elem) – az L listára felfűz egy e elemet
  - lefűz (L:lista):elem – lefűz az L listáról egy elemet

A szemaforváltozó ekkor olyan rekord, aminek integer értéke van, és van neki egy várólistás elme is, mely a különböző folyamatokat azonosíthatja.

```
var s:record of   érték:integer
                  vlist:list of p_id

P(s):  begin
         s.érték = s-1;
         if s.érték < 0 then
           begin
             felfűz(s.vlist, myid);
             sleep;
           end
       end

V(s):  begin
         s.érték = s+1;
         if s.érték ≤ 0 then
           wakeup(lefűz(s.vlist));
       end
```

A szemafor segítségével az alvó (várakozásra került) folyamatokat a listára rakhatjuk, ha az aktuális folyamat befejezte a működését, egyszerűen felébresztünk egy alvó folyamatot. Abban az esetben, ha a lista FIFO (First In First Out) elven működik, még arra is garancia van, hogy minden folyamat véges időben sorra kerül. (Ha prioritásokat állítunk föl, úgy nem.) Tehát a szemaforral nemcsak az adatok összegabalyodását küszöböltük ki, a busy waiting problémát is megoldottuk, mindezt úgy, hogy a folyamatok véges idejű bejutását biztosító elvi kritérium is igaz lett.

Eddig tart a ZH anyaga.

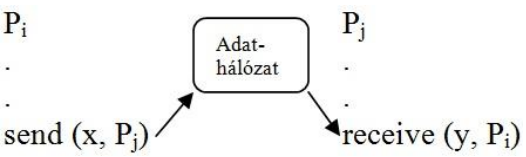
Üzenet alapú együttműködés

Sokat beszéltünk a közös memóriás együttműködésről, mint az információcsere egy lehetőségéről. Üzenet alapú együttműködésnél nem a folyamatok memóriája a közös, hanem egy közös *adatátviteli hálózatra* kapcsolódnak, és ezen keresztül kommunikálnak. Az adatcserét két művelettel érik el: send (küldés) és receive (fogadás). A vizsgált alapprobléma az, hogy egy P_i folyamat el akarja küldeni az x lokális változóját a P_j folyamat memóriájában kialakított y lokális változónak. Lényegében egy y = x műveletről van szó.

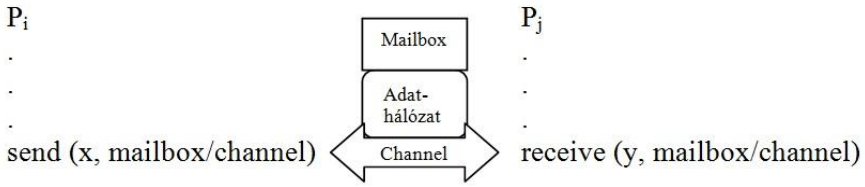
Megnevezés

A megnevezés azzal a problémával foglalkozik, hogyan talál egymásra a küldő és a fogadó.

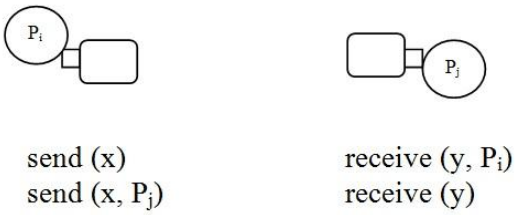
Direkt címzés: a küldő és fogadó „ismerik” egymást, erős a folyamatok közötti csatolás. Ez azt jelenti, hogy a folyamatok futásuk során számítanak egymásra, vagyis nem működhetnek a másik nélkül. Ez azt is jelenti, hogy csak együtt helyezhetők át a folyamatok. Az informatikában a rugalmas felhasználás miatt törekszünk a függőségek lebontására, a direkt címzés pedig nagyon rugalmatlan.



Indirekt címzés: P_i és P_j nem látják egymást, csak az adathálózatot. A kettejük közötti kapcsolat úgynevezett *kommunikációs objektumok* révén állítható fel: ez lehet *postaláda* (mailbox) vagy *csatorna* (channel). A postaládába üzenetet rakhatunk, melyet a fogadó idővel kiolvas, a csatorna pedig üzenetek átadására szolgál. Mindkét esetben egymástól függetlenek a folyamatok, hiszen csak a postaládával vagy a csatornával érintkeznek, a küldő és fogadó műveletek is ezeket célozzák meg, nem a folyamatok lokális változóit.



Aszimmetrikus címzés: ez az előző két módszer átmenete. Az adathálózat input és output portja kötve van a folyamatokhoz, a folyamatok csak a nekik megfelelő portot látják. Vagy P_i folyamat adja meg, melyik porton küld, vagy P_j azt, hogy melyek porton fogad. Így két utasításpár jöhet létre: send(x) és receive(y, P_i) vagy send(x, P_j) és receive(y).



Az aszimmetrikus megnevezés (az Informatika 2 című tantárgyban részletesebben tárgyalt) kliens-szerver kapcsolatot ábrázolja. A kliens a szerver szolgáltatásait igénybe vevő felhasználó. Amikor a szerver programkódját megírjuk, még nem tudjuk, hogy milyen kliensek fognak neki adatot küldeni, így a szerver fogadó műveletének bármilyen klienstől fogadnia kell. Ezért előnyös az, ha a receive művelet paraméterei között nem szerepel a küldő P_i. A kliens ezzel szemben nem kósza adatokat küld az adathálózatra, hanem egy konkrét, megnevezett szerverhez továbbítja az adatokat, így a send művelet paraméterei között szerepel a szerver P_j.

A fordított eset, mikor a fogadó bárkitől kaphat adatot, de a küldő konkrét folyamatot címez, jellemzően a többprocesszoros párhuzamos működés esetén alkalmazható. Egy Master szétosztja a munkát a számos Worker között, így bárkinek küldhet: send(x). A Worker viszont tudja, hogy csak a Master küldhet adatot: receive(y, P_j).

Csoportcímzés: ez egy szélsőséges eset. A küldő üzenete mindenkinek szól, ezt *üzenetszórásnak* (broadcasting) hívjuk. Nagyon hatékony sín topológia esetén, de rádiós hálózatoknál is nagy segítség, hogy nem kell mindenkinek egyesével elküldeni az üzenetet. Hátránya, hogy a címterek, névterek kialakítása elbonyolíthatja.

Szemantika

(Szemantika = jelentéstan.) Az elméleti megfontolások csak addig kecsegtetők, amíg a gyakorlatba át nem kell ültetni azokat. A gyakorlatban nem tudhatjuk, hogy a send parancs kiadása után egészen pontosan mi történik, megérkezik-e az üzenet vagy sem.

Például ha egy levelet beteszek a postaládába, nem tudhatom, mikor olvassák el, ha egyáltalán elolvassák. De az is előfordulhat, hogy nem is tudták elolvasni, adott esetben továbbítás közben adatvesztés történhetett…

Erre jó megoldás lehet az, ha fogadóként nyugtázást küldök arról, hogy megkaptam az üzenetedet, te pedig nem tágítasz, amíg meg nem kaptad azt. Ha nem kapod meg a nyugtázást, újra elküldheted az adatot. Érdemes ügyelni arra, hogy egy üzenetből csak egy példány legyen a hálózatban, ezt valamilyen „együttműködéssel" lehet biztosítani:

- ha nincs pufferelés, biztosítani kell a randevút, mert ha úgy érkezik a send művelet, hogy a partner még nem áll készen a fogadásra, úgy várakoznia kell, míg az el nem készül
- ha pufferelt, ebbe rakhatja a küldő az üzenetet, a fogadó pedig kiolvashatja
    - véges méretű puffer esetén előfordulhat, hogy a puffer megtelik, és kiolvasás előtt még új üzenet érkezik, így az fennakad
    - ha a puffert végtelen méretűnek tekintjük, a küldő sohasem akad el, azonban adatvesztés következhet be, mellyel nem törődünk, ha ritkán van ilyen hiba; esetleg nyugtázó mechanizmust használhatunk a biztonságos működésért

Az üzenetküldés megbízhatóbbá tételéhez védelmek beépítése szükséges. Például, ha a küldő bizonyos idő után nem kap visszaigazolást, nem várakozik tovább, megismétli az üzenetet: *time out* időn belül meg kell érkeznie a válasznak.

Természetesen az is előfordulhat, hogy az üzenetet megkaptad, küldtél nyugtázást, de a nyugtázás nem érkezett meg hozzám. Ilyenkor ha ismételten elküldöm az üzenetet, te azt hiheted, hogy újabb üzenetet kaptál. (Zavaró lehet például, ha megismételnek egy bankszámlaműveletet egy ilyen hiba miatt.) Emiatt szükséges egyfajta védelem kiépítése a hasonló problémák ellen. Ezt az *üzenetek sorszámozásával* érjük el: egy bit információval jelezhetjük a fogadó felé, hogy ez új üzenet-e, vagy egy régebbi megismétlése.

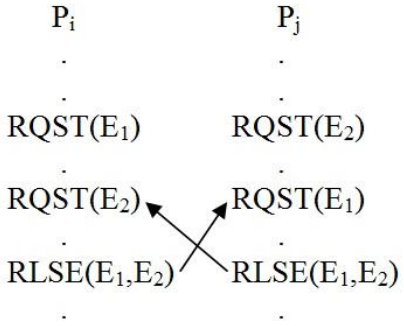
Lehetnek továbbá elektromágneses zavarok, melyek értelmetlenné tehetik az üzenetet. Azonban hibajavító, hibaelemző kódolással (*redundáns kódolással*) kiszűrhetők ezek a selejtes üzenetek. További megoldás lehet az, ha az üzenetet „két borítékba csomagoljuk", ha a külső boríték megsérült, az üzenetet sem szabad elolvasni.

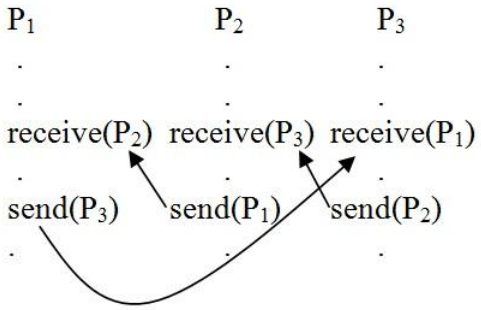
Ha nagyobb adatblokkot szeretnénk elküldeni, a memóriákból való másolgatás helyett sok időt spórolhatunk meg, ha csak az adatblokk pointerét küldjük át. Bajt okozhat, ha ezután mindkét folyamat belepiszkál az adatblokkba, ezért biztosítanunk kell azt, hogy az üzenet küldésének befejeztével ne módosíthassuk később az üzenet tartalmát. (Vagyis az adatblokkot.) Egy jelzőrendszerrel letilthatjuk az adatblokkon való módosításokat addig, amíg a fogadó engedélyt nem ad a visszaállításra.

Az üzenet alapú együttműködéshez a PVM és MPI szabványok definiálnak üzenetközvetítő keretrendszereket. Beépített nyelvi támogatás nem társul, de telepíthetünk csomagokat, keretrendszereket. A párhuzamos programrészek közötti együttműködést a konkurens Pascal, az ADA, a Modula 1 és 2, illetve a Java nyelvek biztosítják.

Holtpont problémák kezelése

Mind közös memóriás, mind üzenetküldéses együttműködésnél ugyanaz a jelenség játszódik le. A közös memóriás együttműködésnél erőforrások használati jogáért versenyeznek a folyamatok. Definiálunk egy RQST(E) műveletet, mely az E erőforrás kizárólagos használatának kérése. Ha nem kapja meg a folyamat, akkor várakozik. A RLSE(E) művelettel a folyamat elengedi a számára már hasztalanná vált erőforrást. (Ez a szemafor P(s) és V(s) műveletével is leírható.) Üzenetküldéses együttműködésnél a folyamatok működésük során várnak egy másik üzenetére (receive), illetve maguk is küldenek üzeneteket (send). Ezekkel a műveletekkel könnyen bemutatható a holtpont kialakulása. (Lásd: ábrák.)





Holtpont helyzet akkor alakul ki, ha a folyamathalmaznak van olyan H részhalmaza, hogy H halmazon belül minden folyamat H halmazon belüli folyamatra vár.

A holtpont helyzet kezelésére több módszer és algoritmus létezik.

Strucc algoritmus

Az algoritmus elnevezése onnan ered, hogy úgymond homokba dugjuk a fejünket. Nem foglalkozunk a holtponttal, ha a kivédése nagyobb gondot okoz, mint a holtpont kialakulása. A holtpont azt jelenti, hogy lefagy a rendszer, egy korai processzor esetén ez néhány hetente fordult csak elő, ekkor gyors újraindítás után megint működött hetekig. Cserébe nem kellett foglalkozni a holtponttal.

Megelőzés

A többi módszerhez vegyünk egy alapszituációt: a holtpont az erőforrásokért való versenyben alakul ki.

Az erőforrások lehetnek egypéldányosak (csak egyetlen darab van belőlük) vagy többpéldányosak. Értelemszerűen egy többpéldányos erőforrást akár több folyamat is használhat egyszerre, annyi darabot, ahány példány van belőle:

- a processzor multiprogramozott esetben egypéldányos, multiprocesszoros esetben többpéldányos
- memória többpéldányos, mert bárhova betöltve a fizikai tárba a programot az működik

A RQST és RLSE műveletek érvényesek lehetnek konkrét megnevezett erőforrásokra vagy erőforrástípusokra.

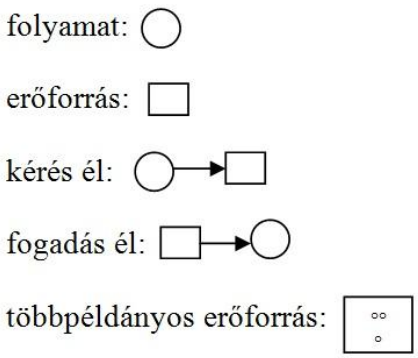
Csoportosíthatók az erőforrások állapotuk elmenthetősége szerint is.

- A processzor állapota regiszterekbe menthető két utasítás között. Előfordulhat, hogy utasítás közben lenne szükség a processzor állapotának mentésére, például hibamegszakítás esetén. Sajnos finomabb léptéket nem lehet választani a mentésre, de

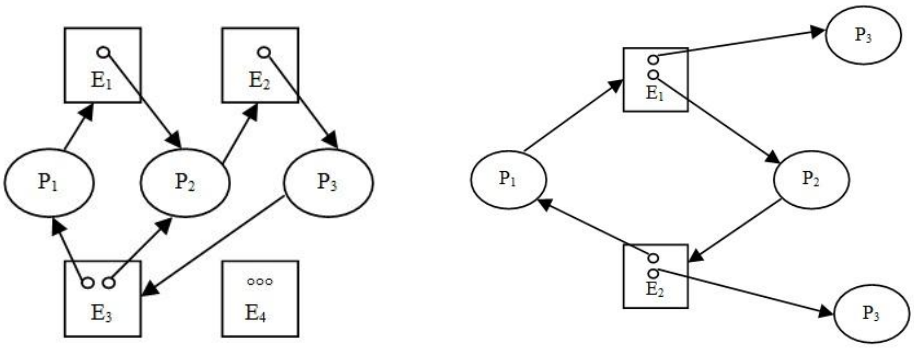
két utasítás között működhet *utasítás rollback*. Utasítás előtt ment, ha utasítás alatt hibamegszakítást kap, akkor visszaállítja az előző állapotot, és visszatér.
- A memória állapota kimenthető háttértárra. (A később tárgyalt virtuális memóriakezelés ezt használja fel.)
- A perifériák, például a nyomtatók állapota nem menthetők. (Nem lenne egyszerű a nyomtatóból kifűzni a lapot, majd visszafűzni és folytatni a nyomtatást.)

A holtpont kialakulásának szükséges feltételei:
- *kölcsönös kizárás* fennállása: legalább egy, csak kölcsönös kizárással működő erőforrás van a rendszerben
- *hold-and-wait* (*foglalva várakozás*) fennállása: legalább egy erőforrást birtokló, emellett más folyamatra várakozó folyamat van a rendszerben
- *no preemption* (*nincs kiszorítás*) a rendszerben: nincs olyan, hogy a folyamattól akarata ellenére veszünk el erőforrást
- *circular wait* (*körkörös várakozás*) van: a folyamatokat és azok várakozásait szemléltető gráfon irányított kör található (egy folyamat vár egy másikra, aki fel tudná szabadítani, aki egy másikra vár, és így tovább addig, amíg vissza nem térünk az elsőhöz)

Erőforrás-foglaltsági gráf: az erőforrások és folyamatok kéréseit és foglaltságát szemlélteti. Az ábrán található jelöléseket használjuk. Többpéldányos erőforrás esetén a téglalapban annyi karika van, ahány példányos az erőforrás.



Gyakorlásképpen vizsgáljuk meg, hogy az alábbi gráfok közül melyekben alakulhat ki holtpont!



Mivel irányított kör mindkét ábrán van, így a holtpontra jutás szükséges feltétele mindkét estben teljesül. Az első ábrán a P_1 folyamat P_2-re vár, ami P_3-ra, mely P_1-re, így kialakul a holtpont. A második ábrán P_1 olyan folyamatra is vár, aki nincs benne az irányított körben, ha P_3 vagy P_4 felszabadít erőforrást, a P_1 és P_2 folyamatok folytatódnak, és nem alakul ki holtpont.

A *megelőzés* azt jelenti, hogy tervezéskor strukturálisan holtpontmentes rendszert hozunk létre. Ezt úgy érjük el, hogy a holtpont kialakulásának legalább egyik szükséges feltételét megtorpedózzuk. A *kölcsönös kizárás* nem elkerülhetetlen, de az átfedések időtartama csökkenthető, így kisebb valószínűséggel alakul ki holtpont. A *foglalva várakozás* megszüntethető, ha a folyamat a működése során felhasznált összes erőforrást a legelején lefoglalja. Ezzel kiküszöböljük a várakozást, de nem túl hatékony módszer, mert olyan erőforrások is lefoglalásra kerülnek, melyek nincsenek használat alatt. A *nincs kiszorítás* feltétel egyszerűen megszüntethető, ha „mégiscsak van kiszorítás". Vagyis a folyamatoktól erőszakosan vesszük el az erőforrásokat. A *körkörös várakozás* csak akkor lép fel, ha irányított kör van a folyamatok és erőforrások gráfjában, tehát ezt kell megszüntetni. Jó módszer lehet, ha megszámozzuk az erőforrásokat, folyamatokat, és sorba állítjuk. Egy folyamat csak nagyobb sorszámú erőforrást foglalhat le, mint amit már birtokol. Mivel nem ér össze a lánc két vége (hiszen n+1 > 1, minden n > 1-re), nem alakulhat ki kör.

Elkerülés (a bankár algoritmussal)

Először vizsgáljunk meg a gyakorlatban egy példát az erőforrások folyamatok közötti szétosztásának súlyára! 10 darab azonos erőforrást szeretnénk P_1, P_2 és P_3 folyamat között úgy kiosztani, hogy nem következik be holtpont. Egy táblázatban foglaljuk össze, hogy a folyamatok külön-külön egyszerre maximálisan hány erőforrást használnak fel, illetve hány erőforrást osztottunk már ki a 10-ből.

folyamatok	**max. erőforrások száma**	**kiosztott erőforrások**	**kérés**
P_1	8	4	
P_2	3	2	1
P_3	9	2	1
		2 szabad erőforrás	

Ha a két szabad erőforrással a két igényt elégítjük ki, akkor P_2 és P_3 is 3 erőforrással rendelkezik. Ez elégséges P_2 lefutásához, de P_3-nak további erőforrásokra lenne szüksége. Miután P_2 lefut, felszabadítja a három erőforrás példányát, de ezt bármelyik másik folyamatnak sem lenne elég a lefutáshoz, mert P_1-nek 8 erőforrásra lenne szüksége, de csak 4 + 3 = 7 erőforrás példánya lenne, P_3-nak pedig 2 + 1 + 3 = 6 példánya. Így holtpontra jutunk.

folyamatok	**max. erőforrások száma**	**kiosztott erőforrások**	**kérés**
P_1	8	4 → 7 ☹	
P_2	3	2 → 3 → 0 ☺	1 → 0
P_3	9	2 → 3 ☹	1 → 0
		2 → 0 szabad példány	

Válasszuk inkább azt a stratégiát, hogy csak P_2 folyamat kérését szolgáljuk ki! Így megmarad egy erőforrás példány, mellyel kiszolgálhatjuk P_1 igényét is, miután P_2 felszabadította a saját 3 példányát. (P_3 kérését nem elégíthetjük ki, mert még mindig nem lenne elegendő erőforrás példányunk, és holtpontra jutnánk.) P_1 folyamat a 8 erőforrás példánnyal lefut, és felszabadítja mind a nyolcat. Ez több, mint elegendő P_3 folyamat lefutásához.

folyamatok	max. erőforrások száma	kiosztott erőforrások	kérés
P_1	8	4 → 8 → 0 ☺	
P_2	3	2 → 3 → 0 ☺	1 → 0
P_3	9	2 → 9 → 0 ☺	1 → 0
		2 → 1→ 0 → 10 szabad	

Biztonságos állapotnak (*safe state*) nevezzük azt a helyzetet, amikor van olyan kiszolgálási sorrend, ami garantálja a folyamatok végigfutását a legrosszabb esetre is. A bankár algoritmus (mely természetesen Dijkstra nevéhez fűződik ☺), ezt garantálja.

A nevét onnan kapta az algoritmus, hogy a probléma analóg egy bankár helyzetével. A bank kölcsönöket ad beruházásokra, de ha egyszer csak elfogy a pénze, nem tud kölcsönt adni minden beruházás befejezésére. Így a bankárnak mérlegelnie kell, mennyi pénzt adhat kölcsön, és kinek.

A helyes működéshez ismerni kell, hogy a folyamatok az adott erőforrástípusokból maximálisan hányra tartanak igényt a futásuk során. N folyamatra és M erőforrástípusra ez egy N×M méretű tömböt jelent. Ezt nevezzük el MAX-nak, melynek MAX_i vektora egy M dimenziós vektor. Jelöljük a lefoglalt erőforrásokat egy FOGLAL N×M tömbben, a még szabad erőforrásokat egy SZABAD N×M tömbben, melyeknek $FOGLAL_i$ és $SZABAD_i$ M dimenziós vektorai. Amennyi erőforrásra még szükség van, azt az alábbi módon határozhatjuk meg: MÉG = MAX – FOGLAL, melynek egy M elemű vektora a $MÉG_i = MAX_i – FOGLAL_i$. Egy folyamat akkor futhat le, ha a maximális igénye kisebb az összes erőforrások számánál. Ha ezt garantáljuk, akkor *biztonságos állapot* jön létre.

Már csak azt kell tudni eldönteni, ha egy folyamat kérését kiszolgálom, az új állapot biztonságos marad-e. Ha igen, akkor bátran odaadom, ha nem, akkor várakoztatom. A kérés legyen egy $KÉR_i$ M elemű vektor.

1. Azt vizsgáljuk, hogy a kérés elfogadásával a folyamat átlépi-e a maximális igényét. Ha átlépi, akkor a folyamat „hazudós”, és hibajel kíséretében kiléptetem. Ha nem tudjuk kiszolgálni a folyamat összes kérését, akkor várakoztatjuk.

$KÉR_i + FOGLAL_i \leq MAX_i$? → nem → ABORT
↓ igen
$KÉR_i \leq SZABAD_i$? → nem → WAIT
↓ igen

2. Ezek után azt vizsgáljuk meg, hogy biztonságos marad-e az állapot, ha odaítélem neki. Ezt úgy teszi meg, hogy végigjátssza a metódust.

$FOGLAL_i = FOGLAL_i + KÉR_i$
$SZABAD_i = SZABAD_i – KÉR_i$

3. Majd megvizsgáljuk, hogy az új állapot biztonságos marad-e. Ha biztonságos, akkor kiszolgálja a folyamatot, ha nem biztonságos, akkor visszaállítja a vektorokat és várakoztat.

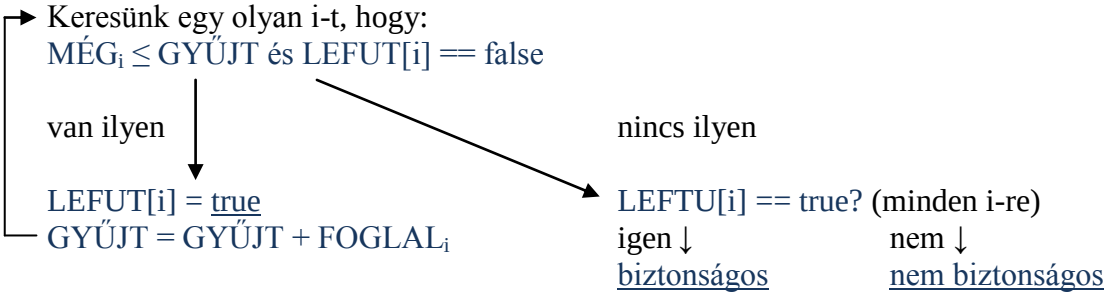
BIZTONSÁGOS? → nem → $FOGLAL_i = FOGLAL_i – KÉR_i$
↓ igen $SZABAD_i = SZABAD_i + KÉR_i$
KISZOLGÁL WAIT

Biztonságosság vizsgálata: ez is több lépésből áll. Legyen a LEFUT egy N elemű vektor, mely igaz/hamis értékeket tárol. Legyen a GYŰJT egy M elemű vektor.

1. Kezdőértékek beállítása.

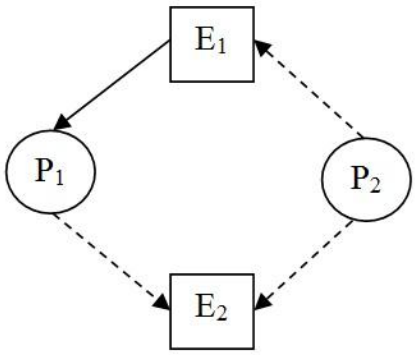
   LEFUT[i] = false (minden i-re)
   $GYŰJT_i$ = SZABAD

2. Olyan folyamatot keresünk, melyek rendelkeznek annyi erőforrással, hogy befejezzék a munkájukat, hogy visszakapjunk erőforrásokat. Így más folyamatokat is kiszolgálhatunk.

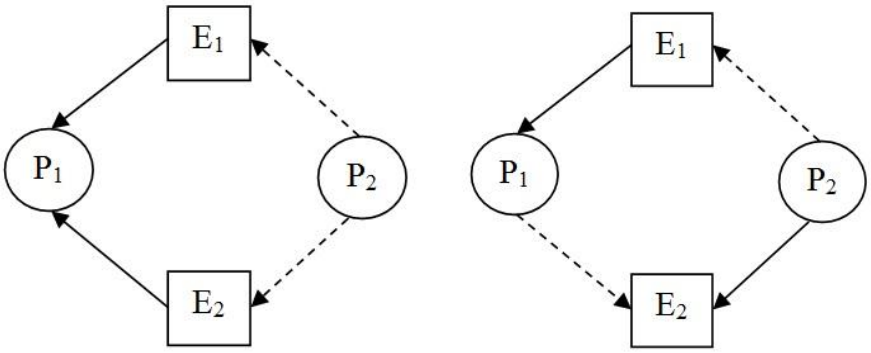


A keresés többszörös bejárást jelent, ez kellőképp lelassítja az algoritmust. Emiatt érzékeny a bankár algoritmus a folyamatok darabszámára. Az erőforrás gráf kibővítésével a bankár algoritmusnál gyorsabb megoldást kaphatunk a probléma megoldására. Nevezzük el a folyamat futása során valamikor előforduló erőforráskérést potenciális kérésnek. Jelöljük ezt szaggatott nyíllal.

Azt kell megvizsgálnunk, hogy a potenciális kérésekkel együtt van-e hurok a gráfban. Vizsgáljuk meg az alábbi egypéldányos esetet.



E_2 erőforrásra tart igény mindkét folyamat. Ha P_1 folyamat kérését elégítjük ki, akkor az rendelkezik elég erőforrással ahhoz, hogy lefusson. Így fel tudja azokat szabadítani, és át tudja adni P_2-nek. Ha viszont a P_2 folyamat kérését teljesítjük, akkor mindkét folyamat egy-egy erőforrást birtokol, amivel egyik sem tud befejeződni, így holtpont alakul ki.



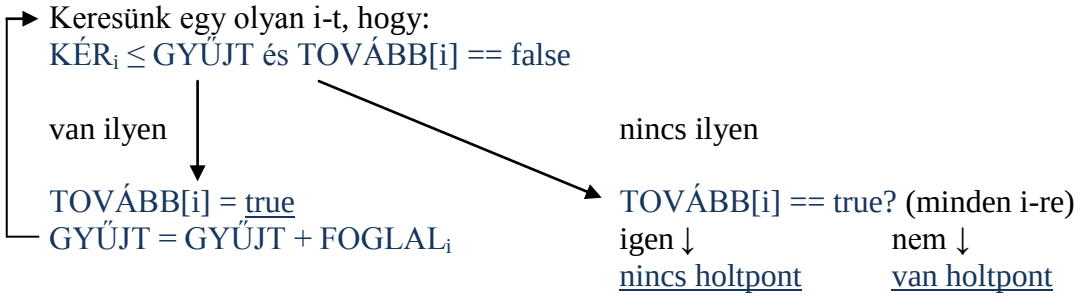
Észlelés

Észlelés: többpéldányos esetre ez a módszer a Soshami-Coffmann algoritmus, ami megtévesztően hasonlít a bankár algoritmusra. Arra keresi a választ, hogy hipotetikus esetben holtpontra jutunk-e.

1. Kezdőértékek beállítása.

   TOVÁBB[i] = false (minden i-re)
   $GYŰJT_i$ = SZABAD

2. Olyan folyamatot keresünk, melyek rendelkeznek annyi erőforrással, hogy befejezzék a munkájukat, hogy visszakapjunk erőforrásokat. Így más folyamatokat is kiszolgálhatunk.



Nézzünk egy feladatot! Van négy folyamatunk, $P_{1,\ldots,4}$. Van három erőforrásunk, R_1-ből 12 darab, R_2-ből 35 darab, R_3 pedig 8 példányos. A már lefoglalt, maximálisan szükséges és a még kérésre váró erőforrásokat az alábbi táblázat tartalmazza. A kérdés az, hogy a P_4-es folyamatnak vajon odaadja-e a bankár algoritmus az RQST(4,3,1) kérésre az erőforrásokat.

	FOGLAL			*MAX*			*MÉG*		
	R_1	**R_2**	**R_3**	**R_1**	**R_2**	**R_3**	**R_1**	**R_2**	**R_3**
P_1	1	2	0	4	13	2	3	11	2
P_2	1	8	1	6	21	6	5	13	5
P_3	1	4	2	5	11	3	4	7	1
P_4	2	2	2	8	28	4	6	26	2

Látható, hogy összesen 5, 16, 5 példányt foglalt le a négy folyamat a különböző erőforrásokból. Az összes erőforrásból ezeket kivonva látható, hogy szabad még 1, 16, 2 erőforráspéldány. Megvizsgáljuk, hogy „hazudós-e" a folyamat, vagyis a kérés elfogadásával átlépi-e a maximális igényét. Nem hazudós. Most megnézzük, hogy van-e elég erőforrás a befejezéséhez? Mivel R_1, R_2 és R_3 erőforrásokból 7, 19 és 3 szabad példány van, és ezekből 4, 3, 1 példányt kér, lesz elég erőforrás a befejezéséhez. Már csak azt kell megvizsgálnunk, hogy biztonságos-e az állapot, ha odaítéljük a folyamatnak az erőforrásokat. Ha megtennénk, az alábbi módon változna a táblázatunk:

	FOGLAL			*MAX*			*MÉG*		
	R_1	**R_2**	**R_3**	**R_1**	**R_2**	**R_3**	**R_1**	**R_2**	**R_3**
P_1	1	2	0	4	13	2	3	11	2
P_2	1	8	1	6	21	6	5	13	5
P_3	1	4	2	5	11	3	4	7	1
P_4	6	5	3	8	28	4	2	23	1

A megmaradt erőforrásokat a legrosszabb esetre osztjuk szét a biztonságos állapot vizsgálatkor. A szabad erőforrásokból megmaradt 3, 16, 2 példány. Ezzel csak P_1 igényeit elégíthetjük ki. Miután P_1 felszabadította az erőforrásait, lesz 4, 18, 2 példányunk a különböző erőforrás-típusokból. Ez csak a P_3 folyamatnak elég, így neki ítéljük oda az erőforrásokat. Miután az is lefutott, felszabadította az erőforrásokat, 5, 22, 4 erőforrás példány lesz a különböző típusokból. Ez nem elég sem P_2, sem P_4 működésének befejezésére. Nem lesz biztonságos az állapot, így nem szolgálja ki az algoritmus a feladatban szereplő kérést.

	FOGLAL			*MAX*			*MÉG*		
	R_1	**R_2**	**R_3**	**R_1**	**R_2**	**R_3**	**R_1**	**R_2**	**R_3**
P_1 ☺	1	2	0	4	13	2	0	0	0
P_2 ☹	1	8	1	6	21	6	5	13	5
P_3 ☺	1	4	2	5	11	3	0	0	0
P_4 ☹	6	5	3	8	28	4	2	23	1

Feloldás

Miután a holtpont kialakult, csak roncsolással lehet azt megszüntetni. Az újraindítás töröl minden folyamatot, ez költséges lehet. Ráadásul egyéb problémák is felléphetnek, ha egy folyamat egy adatbázison hajt végre műveleteket, és újraindítjuk a folyamatot, akkor többszörösen is végrehajthat műveleteket, melyek kárt okozhatnak az adatbázisban. (Például kétszer hajtódik végre egy banki átutalás.) Szükségessé vált az utasítások megismételhetősége, hogy el lehessen felejteni azt, amit a folyamat addig csinált.

Erre jó a korábban már megemlített *rollback* utasítás. Ez mindig párban jár egy *commit* utasítással. A rollback a folyamatnak az utolsó commit kiadásakor megjegyzett állapotát tölti vissza. Ezek a mentések optimalizálhatók, menthető állapotú erőforrások esetén nincs szükség commit utasításra sem. A gyakorlatban viszont folyamatokat állítunk vissza korábbi állapotokba.

Ahol valószínűbb a holtpont kialakulása, ott inkább érdemes *megelőzést* használni. Sőt, különböző technikákat is kombinálhatunk a holtpont kivédésére. Például az operációs rendszerekben erőforrás osztályokat alakítottak ki, erőforrás osztályonként pedig saját holtpont-elkerülést alkalmazhatunk.

A *belső erőforrások* az operációs rendszerek saját erőforrásai, csak az operációs rendszer férhet hozzá, külső folyamatok csak rendszerhívással használhatják. (Ilyenek például a rendszertáblák.) Ezen osztály esetén a *megelőzést* használjuk a holtpont problémák ellen.

Másik erőforrás osztály a *memória*. Itt az *állapotok mentését* használjuk előszeretettel (például háttértárra), és a *megelőzést*.

Harmadik osztály a *job erőforrások*, mely a külső perifériákat tartalmazzák, például adatbázist vagy nyomtatót. A job erőforrások esetén a fejlécben ismertek a használt erőforrások, így ismert a maximális igény is. Óvatos allokációval az *elkerülés* használata az elegáns holtpontvédelem.

A *háttértár* a negyedik ilyen osztály. Ha a folyamatok száma nő, és elfogy a memória, a folyamatokat háttértárra pakoljuk az aktuális állapotukban, ezek visszatöltés után folytathatók. Ezt hívjuk *swapping*-nek. (Kisöpörjük a folyamatokat a háttértárra.) Háttértár esetén a swapping terület ismert, így a *hold-and-wait* kivédésével biztosítjuk a holtpontmentességet. Erre a *megelőzést* használjuk.

Létezik egy modell, mellyel a számítógép holtpont eseteket szimulálhat, vizsgálhat. Ez az úgynevezett Petri-háló. Inkább csak érdekességképpen vettük, így gondoltam, inkább csak megemlítem.