Operációs rendszerek 1. – 4. előadás
Holtpont

Soós Sándor

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

E-mail: soossandor@inf.nyme.hu


Tartalomjegyzék


Hol tartunk?


Az operációs rendszer mint virtuális gép

- 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 megvizsgáltuk a folyamatokat:
  - A folyamat fogalma
  - A folyamatokból álló rendszerek
  - Független, versengő és együttműködő folyamatok
  - Folyamatok születése és halála
  - Folyamatok együttműködése
  - Folyamatok szinkronizációja
  - Folyamatok kommunikációja


A folyamat és a szál fogalma

- Folyamat (process)
  - Meghatározott sorrendben végrehajtott műveletek sorozata
  - A számítógépes terminológiában: A folyamat egy végrehajtás alatt álló program
- 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
- Alternatív elnevezések:
  - Folyamat: „Nehézsúlyú (heavyweight) folyamat”
  - Szál: „Pehelysúlyú (lightweight) folyamat”


Folyamatmodell, I

- 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

Folyamatmodell, II

- 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

Folyamatmodell, III

- 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)


Folyamatmodell, IV

- 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


Független, versengő és együttműködő folyamatok, I

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

Független, versengő és együttműködő folyamatok, II

- 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


Független, versengő és együttműködő folyamatok, III

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

Független, versengő és együttműködő folyamatok, IV

- Az együttműködő folyamatok üzenetküldés segítségével hangolják össze működésüket


Folyamatok együttműködése

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


PRAM memória-modell, I

- 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

PRAM memória-modell, II

- 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


PRAM memória-modell, III

- 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 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)

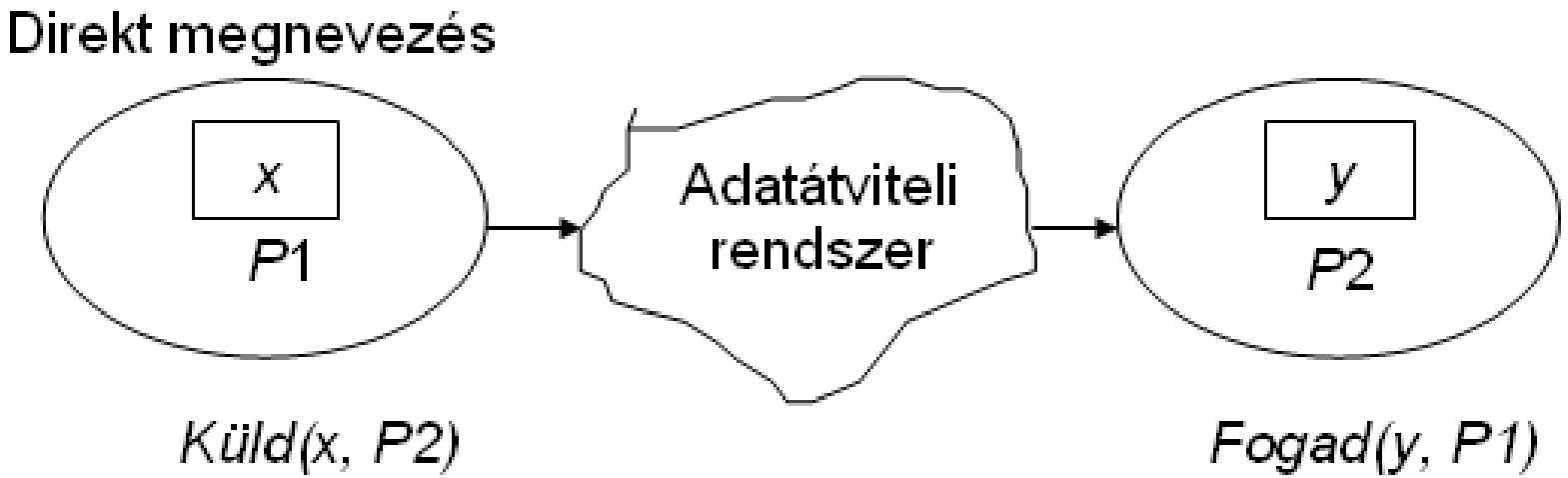

PRAM alapú szoftveres megoldások szinkronizációra

- Foglaltságjelző bit
- Peterson-algoritmus
- Bakery-algoritmus
- A PRAM-modell kiterjesztése OlvasÉsÍr, ill. Csere művelettel


Szinkronizációs eszközök az operációs rendszer szintjén

- Szemafor
- Erőforrás
- Esemény
- Várakozási sorok


Folyamatok kommunikációja



Ismétlés vége


Hol tartunk?


A holtpont jelenség, I

- Az első multiprogramozott rendszerek üzemeltetése közben új típusú furcsa jelenségeket tapasztaltak
- Nehéz volt reprodukálni ezeket
- Miért baj ez?
- Két alapvető hibatípus:
  1. Nem teljesül a kölcsönös kizárás: ezzel foglalkoztunk az előző órán, megtanultunk különböző algoritmusokat a probléma megoldására
  2. Lefagyás: ezzel foglalkozunk ezen az órán
- Mit jelent az, hogy lefagy egy számítógépes rendszer?
- Mi okozhatja ezt?


A holtpont jelenség, II

- Alapos elemzéssel kiderítették, hogy mi történik ilyenkor:
  - Amikor legalább két folyamat van a rendszerben, amelyek valamilyen okból várakoznak
  - Ez önmagában nem okoz holtpontot
  - De...
    - ha az A folyamat arra várakozik, hogy a B folyamat elvégezzen valamit
    - miközben a B folyamat arra vár, hogy az A elvégezzen valamit
  - ...akkor egyik sem tud továbblépni
  - Ezt a helyzetet nevezzük **holtpont**-nak angolul **deadlock**-nak
  - A holtpont kialakulásának valószínűsége általában nagyon kicsi, nehezen állítható elő szándékosan
  - Alattomos hiba!
  - Teszteléssel nem lehet kiszűrni
  - Alapos tervezéssel lehet elkerülni a kialakulását

Egy példa holtpont kialakulására, I

Adott két folyamat A és B, mindkettő használja a nyomtató (Ny) és a mágneslemez (M) erőforrást. Mindkét erőforrásból csak egy van a rendszerben

A **folyamat:**	B **folyamat:**
Lefoglal(M)	*Lefoglal(Ny)*
< mágneslemez használata >	< nyomtató használata >
Lefoglal(Ny)	*Lefoglal(M)*
< mágneslemez és nyomtató együttes használata >	< mágneslemez és nyomtató együttes használata >
Felszabadít(M)	*Felszabadít(Ny)*
< nyomtató használata >	< mágneslemez használata >
Felszabadít(Ny)	*Felszabadít(M)*

Hogyan futhat le ez a két folyamat?

Egy példa holtpont kialakulására, II

Milyen esetek fordulhatnak elő a két folyamat együttes lefutásakor?

1. Egyik megelőzi a másikat. Ekkor biztosan nem alakul ki holtpont
2. Az egyik megszerzi mindkét erőforrást, a másik várakozik, ha szükséges és utána lefut. Ekkor biztosan nem alakul ki holtpont
3. Az egyik lefoglalja az egyik, a másik a másik erőforrást
   - ezután egyik sem tudja megszerezni a másik erőforrást, ezért várakozni kényszerül
   - így nem tudja elengedni az általa lefoglalt erőforrást
   - ezért mindkét folyamat a végtelenségig várakozni kényszerül
   - ez a holtpont

Hol tartunk?


A holtpont definíciója, I

- Definíció:
  - Egy rendszer folyamatainak egy H részhalmaza holtponton van, ha a H halmazba tartozó valamennyi folyamat olyan eseményre vár, amelyet csak egy másik, H halmazbeli folyamat tudna előidézni


A holtpont definíciója, II

- Megjegyzések:
  1. A definíció általános, az esemény nemcsak erőforrás felszabadulása lehet, hanem tetszőleges más valami is, amire egy folyamat várakozni tud
  2. A rendszerben lehetnek futó, élő folyamatok a holtponton lévők mellett, tehát nem biztos, hogy a befagyás teljes
  3. Nem biztos, hogy a holtpont a folyamatok minden együttfutásakor kialakul, sőt az esetek jelentős részében igen kis valószínűséggel alakul ki. Ezért a jelenség nehezen reprodukálható, alattomos hibaforrás
  4. A holtpont egyrészt azon funkciók kiesését okozza, amelyeket a befagyott folyamatok látnak el, másrészt csökkenti a rendszer teljesítőképességét, hiszen a befagyott folyamatok által lekötött erőforrásokhoz a többi folyamat sem tud hozzájutni

Hol tartunk?


Holtpont erőforrásokért versengő rendszerekben, I

- Rendszermodell:
  - Véges számú és típusú erőforrás van a rendszerben
  - Lehetnek egypéldányos és többpéldányos erőforrások
  - A többpéldányos erőforrások egyenértékűek
  - Az erőforrások használati módjuk szerint kétfélék lehetnek:
    1. megosztottan használhatók: az állapotuk elmenthető és visszaállítható, így elvehetők egy folyamattól és később visszaadhatók úgy, hogy a folyamat zökkenőmentesen folytatódhat. Ebben az esetben párhuzamos használatot lehet szimulálni (pl. CPU, memória)
    2. csak kizárólagosan használhatók: az állapotuk nem menthető és nem állítható vissza anélkül, hogy a folyamat károsodna (pl. nyomtató)

Holtpont erőforrásokért versengő rendszerekben, II

Az erőforrások használata során a folyamatok 3 lépést hajtanak végre:

1. Igénylés
2. Használat
3. Felszabadítás

Vegyük sorra ezeket a műveleteket!


Holtpont erőforrásokért versengő rendszerekben, III

1. Igénylés, *Kér* művelet:
   - rendszerhívás
   - több erőforrás több példánya kérhető egyetlen rendszerhívással
   - a paramétere egy n elemű tömb, ha n fajta erőforrás van a rendszerben
   - a tömb i-edik eleme mutatja, hogy hány példányt kérünk az i-edik erőforrásból
   - az igény nem teljesíthető, ha bármelyik erőforrás típusból nem adható ki a kért mennyiség
   - ilyenkor a folyamat várakozni kényszerül
   - az erőforrások egyenértékűek, ezért bármelyiket megkaphatja a folyamat
   - a használathoz viszont azonosítani kell az erőforrásokat, ezért a *Kér* művelet visszaad egy-egy tömböt minden erőforrástípushoz. A tömb tartalmazza a megkapott erőforrások azonosítóit

Holtpont erőforrásokért versengő rendszerekben, IV

- ezek segítségével fogja használni a folyamat az erőforrásokat

2. Használat
   - a megkapott azonosítók segítségével a folyamat használja az erőforrásokat


Holtpont erőforrásokért versengő rendszerekben, V

3. Felszabadítás:
   - kétféle felszabadító műveletet definiálunk
   - az egyik felszabadítja a folyamat által birtokolt összes erőforrást a megadott erőforrástípusokból: *Felszabadít(E1, E2, . . . , Em)*, a paraméterek erőforrástípusok
   - a másik művelet csak a megadott azonosítójú erőforrásokat szabadítja fel: *Felszabadít(V1, V2, . . . , Vn)*, a paraméterek tömbök, amelyek a felszabadítandó erőforrások azonosítóit tartalmazzák az egyes típusokból
   - ha vannak a rendszerben olyan folyamatok, amelyek az erőforrásokra várakoznak, akkor azok megkaphatják a felszabadított erőforrásokat


A holtpont kialakulásának szükséges feltételei

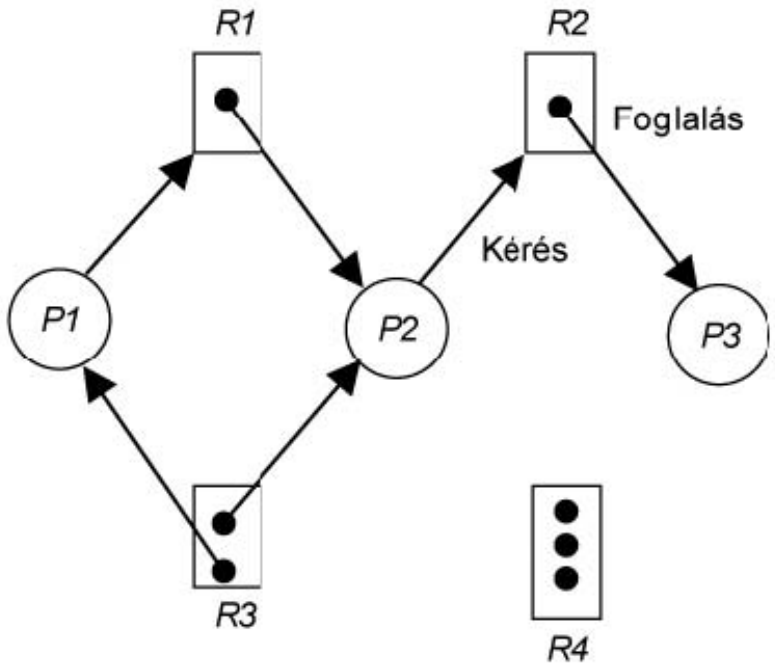
(Mit jelent az, hogy szükséges feltétel?)

1. Kölcsönös kizárás: legyenek olyan erőforrások a rendszerben, amelyeket a folyamatok csak kizárólagosan használhatnak
2. Foglalva várakozás: legyen olyan folyamat a rendszerben, amelyik lefoglalva tart erőforrásokat, miközben más erőforrásokra várakozik
3. Nincs erőszakos erőforrás-elvétel a rendszerben: minden folyamat addig birtokolja az erőforrásokat, amíg ő maga fel nem szabadítja azokat
4. Körkörös várakozás: a rendszerben lévő folyamatok között létezik egy olyan $P_0, P_1, \ldots, P_n$ sorozat, amelyben P_0 várakozik egy P_1 által lefoglalva tartott erőforrásra, P_i egy P_{i+1}-re, P_n pedig P_0-ra várakozik

Ha ezek közül bármelyik feltétel nem teljesül, akkor nem alakulhat ki holtpont

Erőforrásfoglalási gráf, I

A rendszer pillanatnyi állapotának leírására szolgál az erőforrásfoglalási gráf:



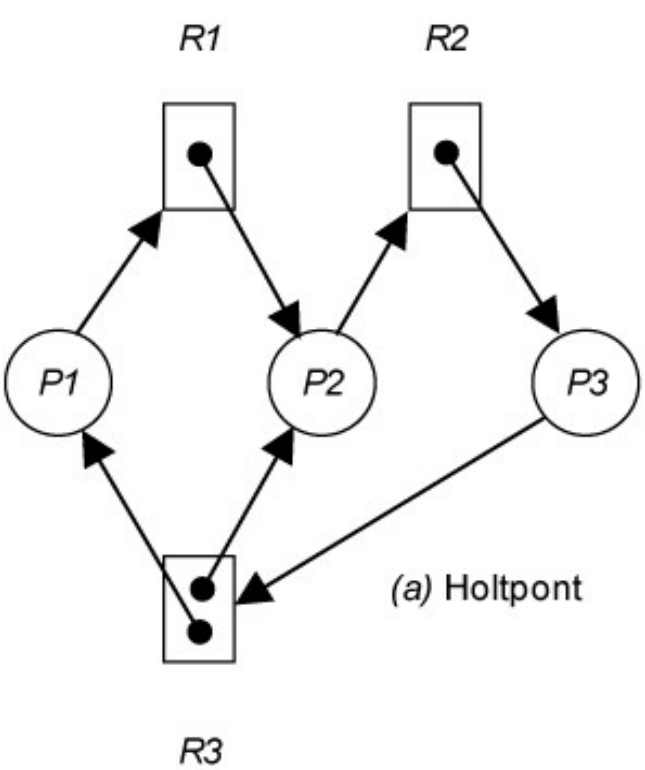
Erőforrásfoglalási gráf, II

- Kétféle csomópont:
  1. Folyamatok: körök, *Pi*
  2. Erőforrástípusok: téglalapok, *Ri*, a konkrét erőforrásokat pontokkal jelöljük a téglalapon belül
- Kétféle él:
  1. Kérés él: irányított él egy folyamattól egy erőforrástípus felé. Azt jelenti, hogy a folyamat igényelt az erőforrásból, de még nem kapta meg
  2. Foglalás él: irányított él egy konkrét erőforrástól egy folyamathoz. Azt jelzi, hogy a folyamat lefoglalta az erőforrást és még nem szabadította fel
- A rendszer működése során a gráf folyamatosan változik, ahogyan kérések, foglalások és felszabadítások történnek

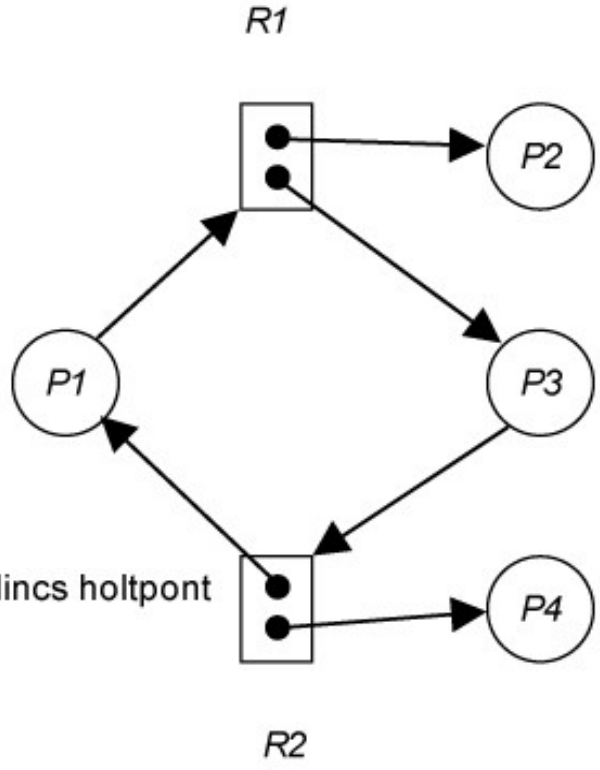
Erőforrásfoglalási gráf, III

- A gráf vizsgálatával következtethetünk a rendszerben előforduló holtpontokra
  - Körkörös várakozás esetén (a holtpont 4. szükséges feltétele) a gráfon is irányított kör van
  - Ha körben lévő minden erőforrás egypéldányos, a gráfon kimutatható kör egyben elégséges feltétel is a holtpont fennállására
  - Ha valamelyik erőforrás többpéldányos, akkor a kör nem jelent feltétlenül holtpontot
  - Vizsgáljuk meg a következő két gráfot!


Erőforrásfoglalási gráf, IV



(a) Holtpont



(b) Nincs holtpont

A baloldali rendszerben van holtpont, a jobb oldaliban nincsen

A holtpontok kezelése

- Mit tehet a rendszer(gazda) a holtponthelyzetek elkerülésére, feloldására?
- Három stratégiát használhatunk:
  1. Nem csinálunk semmit (strucc algoritmus)
  2. Holtpont észlelése és megszüntetése (detektálás és feloldás)
  3. Holtpont kizárása
     - *megelőzés*: olyan rendszert tervezünk, ami kizárja a holtpont kialakulását, kizárjuk a szükséges feltételek valamelyikét
     - *elkerülés*: a rendszer futása közben csak olyan erőforráskéréseket elégítünk ki, amelyek nem vezetnek holtpontveszélyes helyzethez


Strucc algoritmus, I

- Ne csináljunk semmit a holtpontok ellen!
- Ha bekövetkezik, újraindítjuk a rendszert
- Elfogadható ez a hozzáállás?


Strucc algoritmus, II

- Vannak olyan rendszerek, amikor nem lehetséges az újraindítás, ilyenkor nem megfelelő ez a megoldás
- Máskor belefér, hogy szükség esetén leállítjuk és újraindítjuk a rendszert
- Tipikus mérnöki megközelítés:
  - Mérlegeljük, hogy mekkora a probléma, mekkora a bekövetkezés valószínűsége és mekkora a megoldás költsége
  - Ha a megoldás költsége túlságosan nagy a probléma nagyságához képest, akkor nem érdemes bonyolult, drága megoldást választani
- Az esetek nagy részében a holtpont kialakulásának esélye kicsi

Strucc algoritmus, III

- Növeli az esélyt:
  - ha sokféle típusú erőforrás van a rendszerben
  - ha egy-egy típusból kevés erőforrás áll rendelkezésre
  - ha a folyamatok hosszú ideig foglalnak egy-egy erőforrást
  - ha gyakran történik „rákérés”, azaz egy folyamat úgy igényel újabb erőforrást, hogy másik erőforrást már lefoglalva tart
- Ha a holtpont kialakulásának valószínűsége az elfogadható szint fölé emelkedik és/vagy túlságosan nagy kárt okozna, akkor megoldást kell találnunk arra, hogy felismerjük és megszüntessük a holtpontot, vagy megakadályozzuk annak kialakulását


A holtpont észlelése (detektálás), I

- Hogyan vehetjük észre, hogy holtpont van a rendszerben?
  - bizonyos funkciók nem működnek
  - lelassulhat a rendszer
  - a felhasználói parancsokra egyáltalán nem, vagy a szokottnál lassabban reagál
- Gyanú esetén, rendszeres időközönként, vagy valamilyen eseményekhez kötötten lefuttathatunk egy holtpontdetektáló programot. Ezt megteheti az operációs rendszer, vagy a rendszergazda
- Ez megvizsgálja a rendszert és eldönti, hogy van-e holtpont, és ha van, akkor mely folyamatok vannak holtpontban
- Ha van holtpont, akkor azt megszüntetjük néhány folyamat kilövésével (kill)


A holtpont észlelése (detektálás), II

- Ez drasztikus megoldás, de nincs jobb, az érintett folyamat számára olyan, mint a strucc algoritmus esetén az újraindítás, de a többi folyamatot nem érinti
- Szélsőséges esetben azt is megtehetjük, hogy minden erőforráskérés teljesítése után megvizsgáljuk, hogy nem vezetett-e holtponthoz
- Ez azonban akkora pluszterhelést okozna a rendszerben, ami összemérhető az elkerülés érdekében végzendő ellenőrzéssel
- Akkor inkább előre végezzünk ellenőrzést és kerüljük el a holtpontot!


Hogyan müködhet a holtpontdetektáló algoritmus? I

- Vizsgáljunk meg egy konkrét példát!
- Egy rendszerben van 10 darab egyforma erőforrás, és 4 darab folyamat ($F_1, \ldots, F_4$)
- A folyamatok a következőképpen foglalnak most és kérnek további erőforrásokat:

folyamat	foglal	kér
F_1	4	4
F_2	1	0
F_3	3	4
F_4	1	2

- Vizsgáljuk meg, mi a helyzet ebben a rendszerben!


Hogyan müködhet a holtpontdetektáló algoritmus? II

- Jelenleg 9 erőforrás foglalt, 1 még szabad
- F_1, F_3 és F_4 várakozik, és egyik sem elégíthető ki, ezért tovább várakoznak
- F_2 fut
- Van-e holtpont a rendszerben?
  - F_2 biztosan nincs holtponton
  - F_1, F_3 és F_4 várakozik, az a kérdés, hogy egymásra várakoznak-e, vagy van esélyük a továbblépésre?
  - Jelenleg csak F_2 tud futni, ha véget ér, visszaadja az általa lefoglalt 1 darab erőforrást
  - Ekkor lehet a rendszerben 2 darab szabad erőforrás. Mire elég ez?
  - Csak F_4-nek elég, neki van esélye továbbfutni és befejeződni
  - Ha véget ér, visszaadhatja az összes nála lévő erőforrást, mind a hármat

Hogyan müködhet a holtpontdetektáló algoritmus? III

- Ekkor a rendszerben legfeljebb 3 erőforrás lehet szabad
- Ez kevés a másik két folyamatnak, ezért azok holtponton vannak

- Miért fogalmaztunk végig feltételes módban?
  - A táblázatból csak a jelenlegi igényeket látjuk, azt nem tudjuk, hogy később milyen igényei lesznek a folyamatoknak, lehet, hogy olyanok lesznek, amit szintén nem tudunk kielégíteni, és így bármelyik másik folyamat is holtpontba kerülhet
- Ezt a gondolatmenetet használja a következő algoritmus


Coffman-féle holtpontdetektáló algoritmus, I

- Változók:
  - N: folyamatok száma
  - M: erőforrástípusok száma
  - $SZABAD$, M elemű vektor: a szabad erőforrások száma
  - $FOGLAL$, $N \times M$ elemű mátrix: az egyes folyamatok által lefoglalt erőforrások száma
  - KER, $N \times M$ elemű mátrix: a kérések száma
  - $FOGLAL[i]$ a $FOGLAL$ mátrix i-edik sora: az F_i által lefoglalt erőforrások száma
  - $KER[i]$ a KER mátrix i-edik sora: az F_i folyamat kérései az egyes erőforrástípusokból
  - $GYUJTO$, M elemű vektor: a visszakapott erőforrások
  - $TOVABB$, N elemű logikai vektor: a továbbléptethető folyamatok jelzésére

Coffman-féle holtpontdetektáló algoritmus, II

- Az algoritmus alapgondolata:
  - Megkeressük azokat a folyamatokat, amelyek az adott helyzetben továbbléphetnek, mert kielégíthetők az igényei
  - Ezeket végigfuttatjuk és visszavesszük az erőforrásaikat
  - Ezt addig ismételjük, amíg elfogynak az ilyen folyamatok
  - Ha nem maradt folyamat, akkor nincs holtpont
  - Ha maradt, akkor a megmaradt folyamatok holtponton vannak

Coffman-féle holtpontdetektáló algoritmus, III

1. Kezdőértékek beállítása:

```
GYUJTO := SZABAD
TOVABB[i] := hamis, minden j=1, 2, ..., N-re
```

2. Továbblépésre esélyes folyamatok keresése

```
Keress i-t, amelyre
     (TOVABB[i] = hamis és KER[i] <= GYUJTO)
Ha van ilyen i, akkor
  GYUJTO := GYUJTO + FOGLAL[i]
  TOVABB[i] := igaz
  ismételd a 2. lépést
egyébként folytasd a 3. lépéssel
```

3. Értékelés:

Coffman-féle holtpontdetektáló algoritmus, IV

```
Ha TOVABB[i] = igaz, minden i = 1, 2, ..., N-re
  akkor NINCS HOLTPONT
  egyébként holtponton vannak azok a folyamatok,
    amelyekre TOVABB[i] = hamis
```

Coffman-féle holtpontdetektáló algoritmus, V

Megjegyzések:

1. Mit jelent a KER[i] <= GYUJTO feltétel?
   - Két vektor összehasonlítása
   - Mikor kisebb egy vektor a másiknál?
   - Különböző módokon definiálhatjuk
   - Telefonkönyv: Először a vezetéknév alapján döntünk, ha azonos, akkor vizsgáljuk a keresztnevet
   - Most nem ezt használjuk: A vektor akkor kisebb B vektornál, ha A minden eleme kisebb B megfelelő eleménél
2. Mi a helyzet egypéldányos erőforrások esetén?
   - Használhatjuk ugyanezt az algoritmust
   - Hatékonyabb, ha kört keresünk az erőforrásfoglalási gráfban

Holtpont feloldása, I

Mit tegyünk, ha holtpontot találtunk?

- Ekkor már csak radikális megoldások jöhetnek szóba:
  1. Kilövünk legalább egy folyamatot
  2. Elvesszük az erőforrásokat legalább egy folyamattól. Ehhez vissza kell görgetni a folyamatot egy korábbi állapotba, amikor még nem használta az erőforrásokat, vagy az indulásra
- Ezt végezheti a rendszer kezelője manuálisan, vagy a rendszer automatikusan


Holtpont feloldása, II

Megvizsgálandó kérdések:

1. Radikális, vagy kíméletes megoldást válasszunk?
   - Radikális:
     - minden holtpontban lévő folyamatot kilövünk, vagy elvesszük az erőforrásaikat
     - ekkor biztosan megszüntetjük a holtpontot
     - nincsen plusz költség (feladat)
   - Kíméletes:
     - csak bizonyos folyamatokat lövünk ki
     - ehhez döntéseket kell hoznunk, amihez plusz információkat kell beszereznünk és nyilvántartanunk:
     - mely folyamatokat lőjünk ki?
     - hány folyamatot kell kilőni, hogy megszűnjön a holtpont?
     - milyen a folyamatok prioritása?

Holtpont feloldása, III

- hol tartanak az egyes folyamatok? Mekkora részét végezték már el a munkájuknak?
- vannak-e menthető állapotú erőforrások? Ezek elvétele esetén kisebb a veszteség

2. Mi történik ha kilövünk egy folyamatot?
   - Lehet, hogy módosította a rendszert, fájlokat hozott létre, vagy módosított
   - Lehet, hogy a holtpontra jutásakor félkész, inkonzisztens állapotban hagyta a rendszert
   - Az ilyen folyamatot nem szerencsés kilőni


Holtpont feloldása, IV

3. Mi lehet a megoldás?
   - Közbenső visszaállítási pontok létrehozása a folyamatokban
   - Amikor a folyamat eléri ezt a pontot, akkor elmenti az állapotát
   - Ha később holtpontra jut, akkor lehetőség van arra, hogy egy ilyen visszaállítási pontig visszagörgetjük a folyamatot (rollback)
   - Ezt az operációs rendszer nem tudja elvégezni, szükség van a folyamatok felkészítésére és közreműködésére is

Mindezeket a problémákat elkerülhetjük, ha megelőzzük a holtpont kialakulását. Persze ez felvet újabb kérdéseket

A holtpont megelőzése, I

- Ha a rendszer olyan, hogy nem engedhető meg a holtpont kialakulása, akkor védekezni kell ellene
- Ennek legegyszerűbb módja, hogy olyan rendszert tervezünk, amiben kiküszöböljük a holtpont kialakulásának valamelyik szükséges feltételét
- Az ilyen rendszer egyszerű és gyors, mert nem kell plusz ellenőrzésekkel foglalkoznia
- Mik a holtpont kialakulásának szükséges feltételei?
  1. kölcsönös kizárás
  2. foglalva várakozás
  3. nincs erőszakos erőforráselvétel
  4. körkörös várakozás
- Hogyan lehet kiküszöbölni ezeket?

A holtpont megelőzése, II

- Szükséges feltételek kiküszöbölésének lehetőségei:
  1. Kölcsönös kizárás
     - Nincsen lehetőségünk a kiküszöbölésre
     - Bizonyos műveleteket csak kölcsönös kizárással lehet végrehajtani
     - Viszont csökkenthetjük a kölcsönösen kizárt erőforrások számát és használati idejét. Hogyan?
     - Fájl zárolás helyett rekordzárolás
     - Csak írás esetén kell zárolni a fájlt, vagy a rekordot, olvasáskor nem szükséges
     - A kizárólagos használatú szakaszok helyett oszthatatlan műveletek sorosítása
     - A nyomtató lefoglalása helyett nyomtatás fájlba
     - A fájl elküldése a nyomtatónak (oszthatatlan művelet)
     - A nyomtatóvezérlő folyamat sorban kinyomtatja a fájlokat


A holtpont megelőzése, III

2. Foglalva várakozás
   - Ezt a problémát könnyen kiküszöbölhetjük, de ára van
   - Minden folyamat egyetlen rendszerhívással igényelje az összes szükséges erőforrást!
   - Így biztosan nem lesz holtpont
   - A folyamatok hosszabb ideig foglalják az erőforrásokat, mint az feltétlenül szükséges lenne


A holtpont megelőzése, IV

3. Nincs erőszakos erőforráselvétel
   - Ez a feltétel menthető állapotú erőforrások esetén küszöbölhető ki, ellenkező esetben az erőforrás elvétele a folyamat abortálását, vagy egy korábbi állapotba való visszagörgetését eredményezi
   - Kétféle megoldás jöhet szóba:

1. Az erőforrást kérő folyamatot bünteti:
   - ha egy folyamat olyan erőforrást kér, amit nem tudunk kielégíteni, akkor minden erőforrását elveszi tőle a rendszer
   - csak akkor kapja vissza, ha minden kérése kielégíthető
2. Az erőforrást kérő folyamatot kedvezményezi:
   - ha egy folyamat kérését nem lehet kielégíteni, akkor megpróbáljuk a már várakozó folyamatoktól elvett erőforrásokból kielégíteni
   - ha így sem sikerül, akkor ő is várakozóvá válik, és lehet, hogy elveszíti az erőforrásai egy részét

A holtpont megelőzése, V

- a folyamat akkor folytatódhat, ha a kért és az esetleg közben elveszített erőforrásokat egyaránt megkaphatja egyszerre


A holtpont megelőzése, VI

4. Körkörös várakozás
   - Ez a feltétel kiküszöbölhető, ha minden folyamattal betartatunk egy új szabályt
   - Azt szeretnénk elérni, hogy ne lehessen kör az erőforrásfoglalási gráfon
   - A kör így néz ki:
     $P_i \to R_j \to P_{i+1} \to R_{j+1} \to \cdots \to P_{i+n} \to R_{j+n} \to P_i$
   - Sorszámozzuk meg az erőforrásokat!
   - Egyezzünk meg abban, hogy minden folyamat csak nagyobb sorszámú erőforrást igényelhet, mint amilyeneket már korábban lefoglalt!
   - Ha ezt betartjuk, akkor nem alakulhat ki kör a gráfban, azaz nem lehet holtpont a rendszerben
   - Mit jelent ez a megkötés a folyamatokra nézve?
   - Nem foglalhatnak össze-vissza, de nem is kell minden erőforrást egyszerre lefoglalniuk


A holtpont megelőzése, VII

- Ez igényel egy kis odafigyelést és adminisztrációt, de javítja az erőforrások kihasználtságát

- Ha a négy szükséges feltétel valamelyikét kiküszöböljük, akkor biztos, hogy nem alakulhat ki holtpont a rendszerben
- Ez megkövetel némi pluszmunkát és adminisztrációt az operációs rendszertől, vagy megkötéseket jelent a folyamatok számára, de elkerülhetjük vele a holtpont kialakulását és így a folyamatok kényszerű kilövését, vagy a rendszer újraindítását


Holtpont elkerülése, I

- A holtpont elleni védekezés másik módja az elkerülés
- Minden erőforráskérés kielégítése előtt vizsgáljuk meg, hogy nem vezet-e holtpontveszélyes helyzethez, azaz fennmarad-e a biztonságos állapot
- Ezért előfordulhat, hogy a rendszer akkor sem teljesít egy erőforrásigényt, amikor lenne elegendő szabad erőforrás
- Ez a védekezés dinamikus formája
- Futásidejű helyzetelemzést igényel
- Nem vezet be olyan intézkedéseket, amelyek rontanák az erőforrás-kihasználást
- Adminisztrációs teljesítményveszteséget okoz
- Hogyan lehet eldönteni, hogy egy erőforráskérés kielégítése holtpontveszélyt idéz elő?
- Vizsgáljunk meg egy példát többpéldányos erőforrások esetén!


Holtpont elkerülése, II

- Egy rendszerben van 10 darab egyforma erőforrás, és 4 darab folyamat ($F_1, \ldots, F_4$)
- A folyamatok a következőképpen foglalnak most és kérnek további erőforrásokat:

folyamat	maximális igény	foglal	még kérhet	kér
F_1	7	3	4	0
F_2	7	0	7	0
F_3	8	1	7	2
F_4	6	3	3	1

- Vizsgáljuk meg, mi a helyzet ebben a rendszerben!


Holtpont elkerülése, III

- A rendszerben 3 szabad erőforrás van
- P_3 és P_4 igényel most erőforrást
- Mindkét igény kielégíthető
- Mi történik, ha kielégítjük mindkettőt?
  - Elfogy minden szabad erőforrás
  - Ha most a folyamatok benyújtják az igényeiket, a rendszer holtpontra jut
- Mi történik ha csak P_4 igényét elégítjük ki?
  - Marad 2 szabad erőforrás
  - Ha ekkor minden folyamat benyújtja a maximális igényeit, a rendszer szépen sorban ki tudja elégíteni azokat
  - Tehát, ha a rendszer folyamatosan óvatos taktikát folytat, akkor el tudja kerülni a holtpontot
- Ezt a gondolatmenetet valósítja meg a bankár-algoritmus


Bankár-algoritmus, I

- Dijkstra, 1965
- Az elnevezés oka az, hogy a bankok is hasonló elvek alapján helyezik ki az erőforrásaikat (hitel)
- A hitelkérő megmondja, hogy mennyi hitelre van szüksége
- A bank a megvalósítás ütemében folyósítja a hitelt
- Az a hitelfelvevő tudja visszafizetni a hitelt, aki be tudja fejezni a beruházását, ehhez meg kell kapnia a teljes összeget
- Ha a bank túl sok hitelt kezd el folyósítani, akkor előfordulhat, hogy elfogy a pénze mielőtt elkészülnének a beruházások, és a hitelfelvevők még nem tudják visszafizetni a megkapott pénzt
- Hogyan viselkedjen a bank, hogy elkerülje ezt a helyzetet?

Bankár-algoritmus, II

- Változók:
  - N: folyamatok száma
  - M: erőforrástípusok száma
  - MAX, $N \times M$ elemű mátrix: a folyamat maximális igénye az egyes erőforrástípusokból
  - $SZABAD$, M elemű vektor: a szabad erőforrások száma
  - $FOGLAL$, $N \times M$ elemű mátrix: az egyes folyamatok által lefoglalt erőforrások száma
  - MEG, $N \times M$ elemű mátrix: még ennyit kérhet a folyamat az egyes erőforrásokból ($MAX - FOGLAL$)
  - KER, $N \times M$ elemű mátrix: a kérések száma
  - $FOGLAL[i]$ a $FOGLAL$ mátrix i-edik sora: az F_i által lefoglalt erőforrások száma
  - $KER[i]$ a KER mátrix i-edik sora: az F_i folyamat kérései az egyes erőforrástípusokból
  - $MEG[i]$ a MEG mátrix i-edik sora: az F_i folyamat maximális kérései az egyes erőforrástípusokból

Bankár-algoritmus, III

Bankár-algoritmus:

1. A kérés ellenőrzése:

   ```
   Ha KER[i] > MEG[i] akkor STOP  // nem kérhet ennyit
   Ha KER[i] > SZABAD akkor VÉGE  // nincs elég erőforrás
   ```

2. A nyilvántartás átállítása az új állapotra:

   ```
   SZABAD := SZABAD - KER[i]
   FOGLAL[i] := FOGLAL[i] + KER[i]
   ```

3. Biztonságosság vizsgálata külön algoritmussal
4. Döntés:

Bankár-algoritmus, IV

```
Ha nem BIZTONSAGOS akkor
  az állapot visszaállítása:
    SZABAD := SZABAD + KER[i]
    FOGLAL[i] := FOGLAL[i] - KER[i]
    VÉGE  // várni kell
egyébként
  a kérés teljesítése
  VÉGE
```

Bankár-algoritmus, V

A biztonságosság vizsgálata:

- Változók:
  - *GYUJTO*, M elemű vektor: a visszakapott erőforrások
  - *LEFUT*, N elemű logikai vektor: a továbbléptethetőnek talált folyamatok jelzésére
- Alapötlet:
  - Sorban keressük meg azokat a folyamatokat, amelyek a legrosszabb esetben is le tudnak futni
  - A legrosszabb eset az, amikor minden folyamat igényli a maximálisan igényelhető mennyiségű erőforrást
  - Amelyik folyamat ilyen, azt „futtassuk le”, adja vissza az erőforrásokat
  - A keresést most már a bővebb erőforráskészlettel folytathatjuk
  - Az algoritmus akkor áll le, ha elfogytak a megfelelő folyamatok

Bankár-algoritmus, VI

- Ha minden folyamat lefuttathatónak bizonyult, akkor az állapot biztonságos, ha nem, akkor a megmaradt folyamatok holtpontra juthatnak a legrosszabb esetben, ezért az állapot nem biztonságos

Bankár-algoritmus, VII

A biztonságosság ellenőrzése:

1. Kezdőértékek beállítása:

```
GYUJTO := SZABAD
LEFUT[i] := hamis, minden i=1, 2, ..., N-re
```

2. Továbblépésre esélyes folyamatok keresése:

```
Keress i-t, amire (LEFUT[i]=hamis és MEG[i]<=GYUJTO)
Ha van ilyen i, akkor
  GYUJTO := GYUJTO + FOGLAL[i]
  LEFUT[i] := igaz
  ismételd a 2. lépést
egyébként folytasd a 3. lépéssel
```

Bankár-algoritmus, VIII

3. Kiértékelés:

```
Ha LEFUT[i] = igaz, minden i=1, 2, ..., N-re
  akkor BIZTONSÁGOS
egyébként NEM BIZTONSÁGOS
```

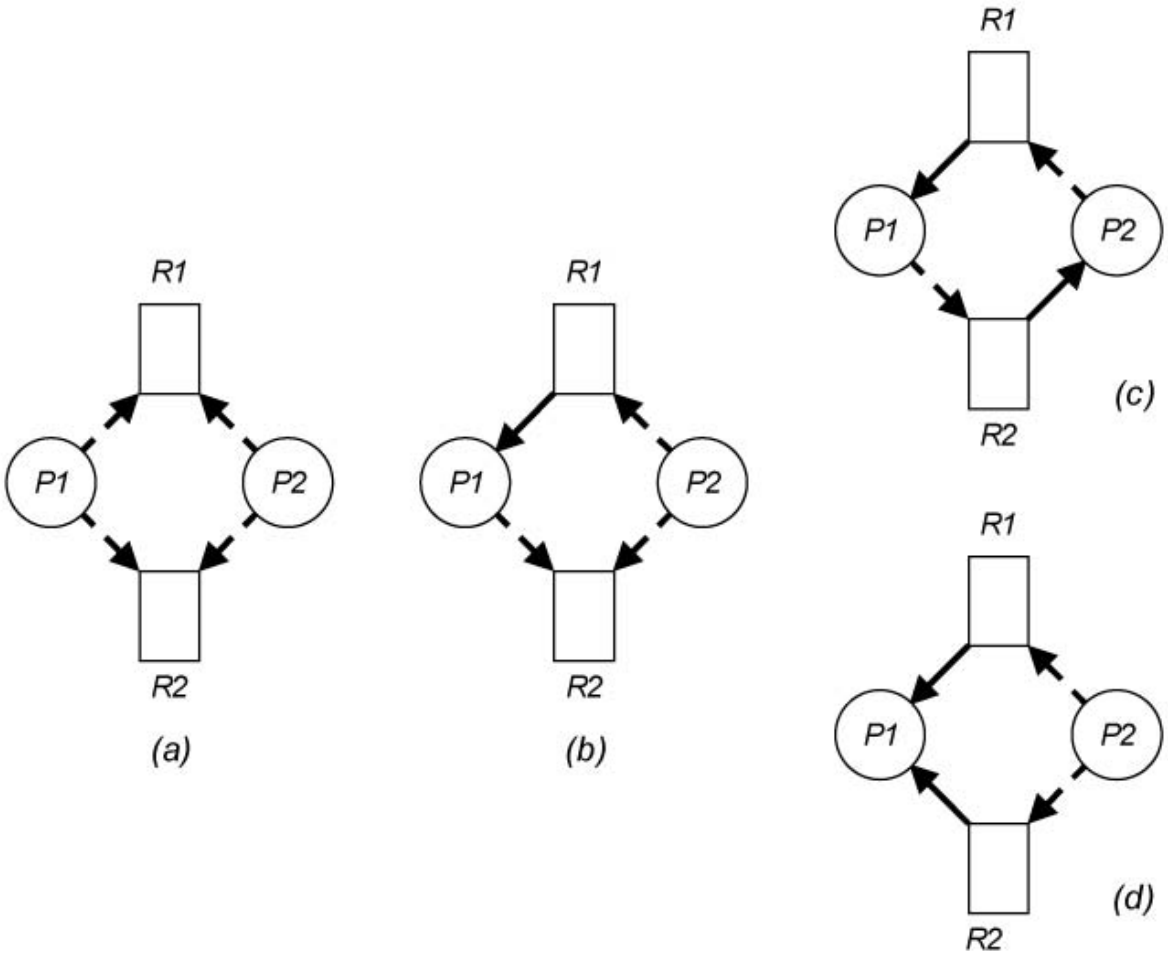
(Azok a P_i folyamatok, amelyekre LEFUT[i] = hamis, holtpontra juthatnak)

Bankár-algoritmus, IX

Megjegyzések:

- Az algoritmus emlékeztet a Coffman-féle holtpontdetektáló algoritmusra, de ez időben korábban született
- Egypéldányos erőforrások esetén itt is érdemes az erőforrásfoglalási gráfot használni.
  - Vezessünk be egy új éltípust a lehetséges kérések jelzésére
  - Ez egy folyamattól vezet egy erőforráshoz, ha a folyamat a jövőben kérheti az erőforrást
  - Ezzel a jelöléssel a legrosszabb eset azt jelenti, hogy az összes lehetséges kérés él valóságos kéréssé alakul
  - A biztonságosság mérlegelésekor ezt az élet megfordítjuk (teljesítjük a kérést) és az így kialakult gráfban ellenőrizzük kör meglétét
- Ezt illusztrálja a következő ábra

Bankár-algoritmus, X



Kombinált stratégiák

- Az eddig tárgyalt módszereket kombináltan is használhatjuk
- Az erőforrásokat osztályokba sorolhatjuk és a különböző osztályba tartozókat eltérően kezelhetjük
- Például osszuk négy osztályba az erőforrásokat:
  1. Rendszererőforrások: ezeket csak a rendszerfolyamatok érhetik el
  2. Memória: menthető állapotú erőforrás (háttértárra mentés, visszatöltés) használható az erőszakos elvétel
  3. Készülékek és fájlok: elkerülés alkalmazható a használati igény előzetes bejelentése alapján
  4. Munkaterület a lemezen: ismert igények vanak, egyszerre kell igényelni, nincs rákérés


Éhezés, I

- A holtponttal rokon jelenség, de nem azonos azzal!
- Azt jelenti, hogy egy várakozó folyamat nincs holtponton, de nincs rá garancia, hogy véges időn belül továbbindulhat
- Ha mindig véletlen módon választunk a várakozó folyamatok közül, akkor nem tudjuk garantálni, hogy nem jelentkezik az éhezés problémája
- Az éhezés elkerülhető, ha megfelelő ütemező algoritmusokat használunk (FIFO, LIFO, prioritásos)


Éhezés, II

- Egy ütemező algoritmus tisztességes (fair), ha garantálja, hogy egy várakozási sorból minden folyamat véges időn belül továbbindulhat, amennyiben a rendszerben véges számú folyamat működik és a rendszerben nincs holtpont vagy hibás folyamat (amelyik például nem enged el egy megszerzett erőforrást)
- Ellenkező esetben tisztességtelen (unfair)
- Tisztességes ütemezés a FIFO
- Tisztességtelen a prioritásos ütemezés, ha statikusan rögzített rangsort használ
- A tisztességtelen ütemezés nem feltétlenül rossz, használhatjuk szándékosan is, de számolni kell az éhezés lehetőségével
- Az éhezés nem holtpont!!!


Éhezés, III

- Ha külső eseményre (például a kezelőre) várakozik egy folyamat, akkor az nem számít sem holtpontnak, sem éhezésnek, ilyenkor a folyamat nem kész a futásra


Hol tartunk?


Emlékeztető kérdések, I

1. Mit nevezünk holtpontnak?
2. Milyen rendszermodellben vizsgáljuk a holtpont jelenségét?
3. Mik a holtpont kialakulásának szükséges feltételei?
4. Mit nevezünk erőforrásfoglalási gráfnak?
5. Mire használható az erőforrásfoglalási gráf?
6. Milyen stratégiákat használhatunk a holtpontok kezelésére?
7. Hogyan tudjuk detektálni a rendszerben lévő holtpontot?
8. Hogyan működik a Coffman-algoritmus?
9. Hogyan tudjuk feloldani a rendszerben lévő holtpontot?
10. Hogyan lehet megelőzni a holtpont kialakulását?
11. Hogyan lehet elkerülni a holtpont kialakulását?
12. Hogyan működik a bankár-algoritmus?


Emlékeztető kérdések, II

13. Honnan származik a bankár-algoritmus elnevezése?
14. Hogyan kombinálhatjuk a különböző holtpontkezelési módszereket?
15. Mit nevezünk éhezésnek? Hogyan kerülhetjük el?


Befejezés

Köszönöm a figyelmet!