

Operációs rendszerek 1. – 5. előadás

Konkurens rendszerek

Soós Sándor

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

E-mail: soossandor@inf.nyme.hu



Tartalomjegyzék

1 Ismétlés

- Emlékeztető az előző órákról

2 Klasszikus konkurens problémák

- Miért foglalkozunk ezekkel?
- Termelő-fogyasztó probléma
- Írók-olvasók problémája
- Étkező filozófusok problémája
- Adatfolyamok illesztése

3 Nyelvi eszközök a folyamatok programozására

- Folyamatokból álló rendszerek programozása
- A párhuzamosság leírása

4 Befejezés

- Emlékeztető kérdések



Hol tartunk?

1 Ismétlés

- Emlékeztető az előző órákról

2 Klasszikus konkurens problémák

- Miért foglalkozunk ezekkel?
- Termelő-fogyasztó probléma
- Írók-olvasók problémája
- Étkező filozófusok problémája
- Adatfolyamok illesztése

3 Nyelvi eszközök a folyamatok programozására

- Folyamatokból álló rendszerek programozása
- A párhuzamosság leírása

4 Befejezés

- Emlékeztető kérdések



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



A holtpont kialakulásának szükséges feltételei

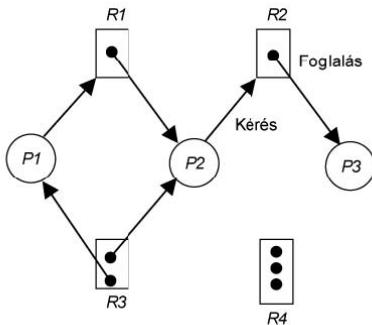
(Mit jelent az, hogy szükséges feltétel?)

- 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, \dots, 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, P_i
 - 2 Erőforrástípusok: téglalapok, R_i , 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 Foglálás 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

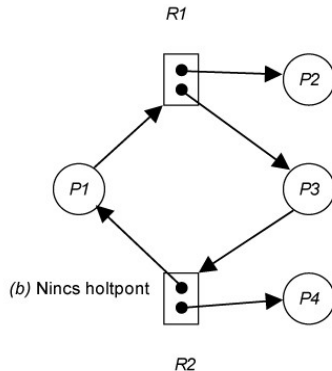
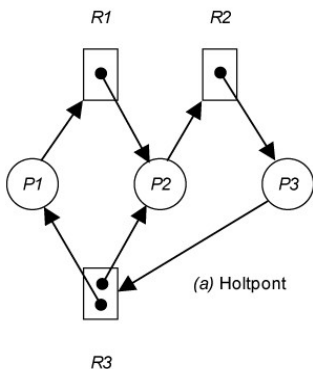


Erőforrásfoglalási gráf, III

- A gráf vizsgálatával következtethetünk a rendszerben előforduló holtponthokra
 - 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 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



Ismétlés vége



Hol tartunk?

- 1 Ismétlés
 - Emlékeztető az előző órákról
- 2 Klasszikus konkurens problémák
 - Miért foglalkozunk ezekkel?
 - Termelő-fogyasztó probléma
 - Írók-olvasók problémája
 - Étkező filozófusok problémája
 - Adatfolyamok illesztése
- 3 Nyelvi eszközök a folyamatok programozására
 - Folyamatokból álló rendszerek programozása
 - A párhuzamosság leírása
- 4 Befejezés
 - Emlékeztető kérdések



Miért foglalkozunk ezekkel?

- A gyakorlatban előforduló feladatok nagy része visszavezethető ezekre az alapesetekre
- Ezeket szoktuk felhasználni egy-egy új együttműködési modell tesztelésére, vizsgálatára, különböző eszközök összehasonlítására
- Mi most arra fogjuk használni, hogy megvizsgáljuk a különböző nyelvi eszközöket



Hol tartunk?

- 1 Ismétlés
 - Emlékeztető az előző órákról
- 2 **Klasszikus konkurens problémák**
 - Miért foglalkozunk ezekkel?
 - **Termelő-fogyasztó probléma**
 - Írók-olvasók problémája
 - Étkező filozófusok problémája
 - Adatfolyamok illesztése
- 3 Nyelvi eszközök a folyamatok programozására
 - Folyamatokból álló rendszerek programozása
 - A párhuzamosság leírása
- 4 Befejezés
 - Emlékeztető kérdések

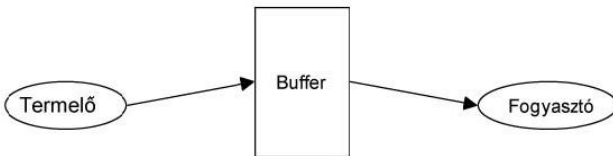


Termelő-fogyasztó probléma, I

- A rendszerben egy termelő és egy fogyasztó folyamat fut egymással párhuzamosan
- Mindkét folyamat saját ütemében dolgozik
- A két folyamatot egy köztes tároló (puffer) segítségével kapcsoljuk össze
- A gyakorlatban a puffer véges kapacitású
- A termelő a saját ütemében előállít egy-egy „terméket” és elhelyezi a pufferben
- A fogyasztó kiveszi a „terméket” a pufferből és felhasználja
- A puffer kiegyenlíti a két folyamat közötti kisebb sebességkülönbségeket



Termelő-fogyasztó probléma, II



Termelő:

loop

<előállít egy elemet>

<beteszi az elemet a Buffer-be>

endloop

Fogyasztó:

loop

<kivesz egy elemet a Buffer-ből>

<felhasználja az elemet>

endloop



Termelő-fogyasztó probléma, III

- Elvárások a rendszerrel kapcsolatban:
 - Ha a puffer üres, akkor a fogyasztó várakozik, amíg nem lesz feldolgoznivaló
 - Ha megtelt a puffer, akkor a termelő várakozik, amíg nem lesz szabad hely a pufferben
 - Elvárjuk, hogy a fogyasztó ugyanolyan sorrendben dolgozza fel az elemeket, ahogyan a termelő előállította azokat
- Feladat:
 - Hogyan tudnánk olyan programot írni valamilyen nyelven, valamilyen operációs rendszer alatt, ami megvalósítja ezt a rendszert?



Hol tartunk?

- 1 Ismétlés
 - Emlékeztető az előző órákról
- 2 **Klasszikus konkurens problémák**
 - Miért foglalkozunk ezekkel?
 - Termelő-fogyasztó probléma
 - **Írók-olvasók problémája**
 - Étkező filozófusok problémája
 - Adatfolyamok illesztése
- 3 Nyelvi eszközök a folyamatok programozására
 - Folyamatokból álló rendszerek programozása
 - A párhuzamosság leírása
- 4 Befejezés
 - Emlékeztető kérdések



Írók-olvasók problémája, I

- Valamilyen adatszerkezetet egyszerre többen akarnak írni és olvasni
- A zavartalan működés érdekében a következő rendszabályokat kívánjuk betartatni:
 - tetszőleges számú olvasó olvashatja az adatokat egyszerre, nem zavarják egymást
 - írás és olvasás nem folyhat egyidejűleg, mert az olvasó félkész adatokat olvasna
 - több írás nem folyhat egyidejűleg
 - tehát új írás csak akkor kezdődhet, ha sem írás, sem olvasás nem zajlik
 - olvasás pedig akkor indulhat, ha nem folyik írás



Írók-olvasók problémája, II

- célszerű lehet egy további szabály bevezetése, hogy elkerüljük az írások végtelen várakozását: ha van várakozó író, akkor újabb olvasó csak akkor kerülhessen sorra, ha a várakozó írók már végeztek. Ezt a változatot szokás írók-olvasók II. problémának nevezni
- Hogyan tudnánk olyan programot írni, ami megvalósítja ezt a rendszert?



Hol tartunk?

- 1 Ismétlés
 - Emlékeztető az előző órákról
- 2 Klasszikus konkurens problémák
 - Miért foglalkozunk ezekkel?
 - Termelő-fogyasztó probléma
 - Írók-olvasók problémája
 - Étkező filozófusok problémája
 - Adatfolyamok illesztése
- 3 Nyelvi eszközök a folyamatok programozására
 - Folyamatokból álló rendszerek programozása
 - A párhuzamosság leírása
- 4 Befejezés
 - Emlékeztető kérdések

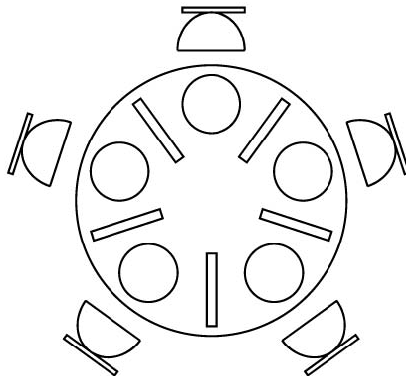


Étkező filozófusok problémája, I

- Egy tibeti kolostorban öt filozófus él
- Minden idejüket egy asztal körül töltik
- Mindegyikük előtt van egy tányér, amiből sohasem fogy ki a rizs
- A tányérok mellett jobb és baloldalon is egy-egy pálcika található
- A helyzetet a következő ábra szemlélteti:



Étkező filozófusok problémája, II



Étkező filozófusok problémája, III

- A filozófusok életüket az asztal melletti gondolkodással töltik
- Amikor megéheznek, étkeznek, majd újra gondolkodóba esnek a következő megéhezésig
- Az étkezéshez meg kell szerezniük a tányérjuk melletti mindkét pálcikát
- Ezért amíg valamelyik szomszédjuk eszik, addig nem ehetnek
- Aki befejezte az evést, az leteszi a pálcikákat, így azokat a két szomszédjuk használhatja



Étkező filozófusok problémája, IV

- Hogyan kell viselkedniük a filozófusoknak, hogy...
 - ne vesszenek össze a pálcikákon
 - ne kerüljenek olyan megoldhatatlan probléma elé, amitől nem tudnak többé sem enni, sem gondolkodni (például, ha mindenki felveszi a baloldali pálcikát és nem teszi le, az holtponthelyzet)
 - senki ne haljon éhen, azaz aki éhes, az egy idő után biztosan tudjon enni (megkapja mindkét pálcikáját)
- Hogyan tudnánk olyan programot írni, ami leírja a filozófusok viselkedését?



Hol tartunk?

- 1 Ismétlés
 - Emlékeztető az előző órákról
- 2 **Klasszikus konkurens problémák**
 - Miért foglalkozunk ezekkel?
 - Termelő-fogyasztó probléma
 - Írók-olvasók problémája
 - Étkező filozófusok problémája
 - **Adatfolyamok illesztése**
- 3 **Nyelvi eszközök a folyamatok programozására**
 - Folyamatokból álló rendszerek programozása
 - A párhuzamosság leírása
- 4 **Befejezés**
 - Emlékeztető kérdések



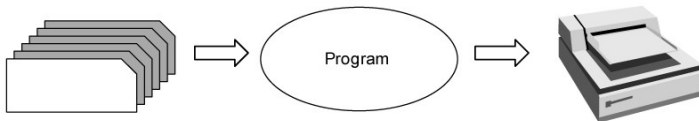
Adatfolyamok illesztése, I

- A probléma klasszikus megfogalmazása a következő:
 - Adott egy kártyaolvasó és egy nyomtató
 - A kártyaolvasóba helyezett kártyákon a kinyomtatandó szöveg karakterei vannak, egy kártyán legfeljebb 80 karakter
 - A bekezdések végét egy speciális karakter (NL) jelzi
 - A szöveget lapokra tördelve, oldalszámozással ellátva, a bekezdéseket új sorban kezdve, soronként 132 karakter írva kell kinyomtatni



Adatfolyamok illesztése, II

Klasszikus adatfolyam-illesztési feladat
kártyaolvasó és nyomtató között



Adatfolyamok illesztése, III

- A problémát általánosíthatjuk napjaink rendszereire is:
 - különböző típusú és szerkezetű adatfolyamok illesztése
 - különböző kommunikációs protokollokkal működő rendszerek illesztése
 - ...
- Hogyan tudnánk olyan programot írni, ami maximális sebességgel tudja működtetni a két oldalon lévő készülékeket?



Konkurens rendszerek programozása

- Ha megpróbálunk programot írni az előbbi rendszerek megvalósítására, hagyományos operációs rendszerek és programozási nyelvek esetén különböző problémákba ütközünk
- Nem tudunk igazi párhuzamosan futó folyamatokat létrehozni
- Enélkül nem tudjuk megvalósítani ezeket a rendszereket
- Milyen eszközökkel kell kiegészítenünk a programozási nyelveket ahhoz, hogy ilyen rendszereket tudjunk programozni?



Hol tartunk?

- 1 Ismétlés
 - Emlékeztető az előző órákról
- 2 Klasszikus konkurens problémák
 - Miért foglalkozunk ezekkel?
 - Termelő-fogyasztó probléma
 - Írók-olvasók problémája
 - Étkező filozófusok problémája
 - Adatfolyamok illesztése
- 3 **Nyelvi eszközök a folyamatok programozására**
 - **Folyamatokból álló rendszerek programozása**
 - A párhuzamosság leírása
- 4 Befejezés
 - Emlékeztető kérdések



Milyen eszközökre van szükség?

- Ahhoz, hogy ilyen rendszereket tudjunk programozni, különböző feladatokat kell tudnunk megoldani:
 - 1 Meg kell határoznunk, hogy mely programrészek lesznek önálló folyamatok
 - 2 Meg kell oldanunk a folyamatok szinkronizálását
 - 3 Lehetővé kell tennünk a folyamatok közötti kommunikációt
- Kétféleképpen oldhatjuk meg ezeket a feladatokat:
 - 1 Minden folyamatot külön programban valósítunk meg és a folyamatok kezelését az operációs rendszer rendszerhívásaival valósítjuk meg
 - 2 A programozási nyelv nyújt megfelelő eszközöket



Hol tartunk?

- 1 Ismétlés
 - Emlékeztető az előző órákról
- 2 Klasszikus konkurens problémák
 - Miért foglalkozunk ezekkel?
 - Termelő-fogyasztó probléma
 - Írók-olvasók problémája
 - Étkező filozófusok problémája
 - Adatfolyamok illesztése
- 3 **Nyelvi eszközök a folyamatok programozására**
 - Folyamatokból álló rendszerek programozása
 - **A párhuzamosság leírása**
- 4 Befejezés
 - Emlékeztető kérdések



Precedenciagráf, I

- Bontsuk fel a megoldandó feladatot elemi (tovább már nem bontható) műveletekre! Ezeket tekintsük elemi utasításoknak, vagy folyamatoknak!
- Ha nem mondunk mást, akkor ezek tetszőleges módon futhatnak egyik a másik után, vagy akár egymással párhuzamosan
- Az utasítások között megkövetelt precedenciákat egy gráfban ábrázolhatjuk (**precedenciagráf**)
- A gráf csomópontjai az utasítások
- Él vezet az A csúcsból B -be, ha A lefutása meg kell, hogy előzze B futását

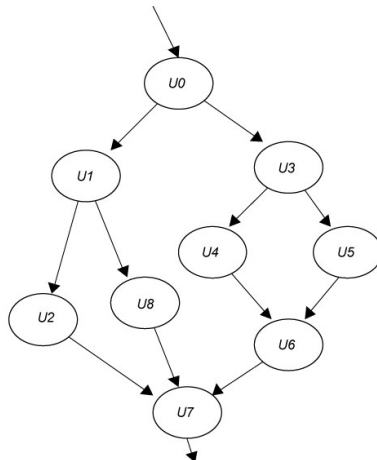


Precedenciagráf, II

- Ekkor az irányított utakon elhelyezkedő elemi utasítások összefoghatók egyetlen folyamattá
- Bármely két utasítás, amit nem köt össze irányított út, párhuzamosan is végrehajtható, azaz konkurens



Precedenciagráf, III



Precedenciagráf, IV

- A precedenciagráf jól kifejezi a folyamatok között fennálló párhuzamosságot
- Ez alkalmas a folyamatok között fennálló kapcsolatok dokumentálására, de csak síkban tudjuk lerajzolni, a programkód viszont lineáris szerkezetű
- Ezért megoldást kell találni arra, hogy a programkódban is le tudjuk írni ezeket
- Először az operációs rendszereket író programozóknak volt szükségük erre
- A megoldások kialakulása után a felhasználói alkalmazások programozói is elkezdtek használni ezeket a technikákat



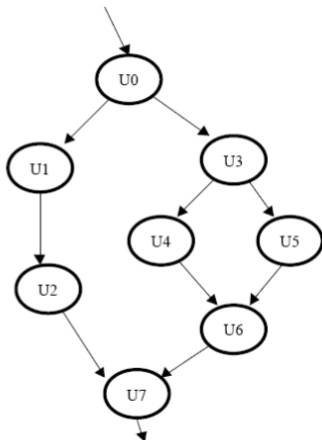
Fork-join utasításpár, I

- Az első megoldás a precedenciagráf nyelvi megvalósítására
- A **fork** címke utasítás egy új szálát indít
- A szál kezdőpontja a címke utáni első utasítás lesz
- A jelenlegi szál pedig folytatódik a **fork** után
- A **join** n utasítás n darab szálát összevár



Fork-join utasításpár, II

```
U0;  
FORK L1;  
U1;  
U2;  
L3: JOIN 2;  
U7;  
EXIT;  
L1: U3;  
FORK L2;  
U4;  
L4: JOIN 2;  
U6;  
GOTO L3;  
L2: U5;  
GOTO L4;  
END;
```



Fork-join utasításpár, III

- A fork-join utasításpár előnyei és hátrányai
 - ⊕ Bármely precedenciagráf leírható vele
 - ⊖ Nehezen tekinthető át
 - ⊖ Nem strukturált megoldás (hasonlít a goto utasításra)
 - ⊖ Miért nem szeretjük a goto utasítást?



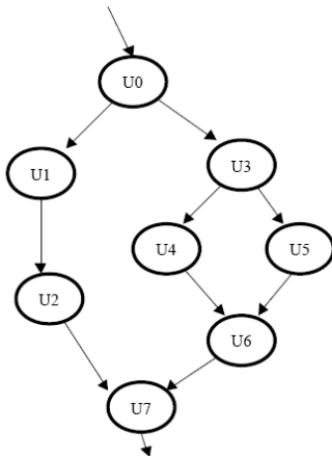
Parbegin-parend utasításpár, I

- Strukturált szerkezetekkel írja le a folyamatok kapcsolatait
- Alternatív elnevezések: **parbegin-parend**, **cobegin-coend**
- A blokkban szereplő utasítások egymással párhuzamosan hajtódnak végre, csak ezután folytatódhat a végrehajtás a **parend** után
- Értékelés:
 - ⊕ strukturáltan, jól áttekinthetően és biztonságosan írja le a folyamatok közötti összefüggéseket
 - ⊖ nem lehet minden precedenciagráfot leírni, szükség van plusz szinkronizációs műveletekre



Parbegin-parend utasításpár, II

```
begin
  U0;
  cobegin
    begin
      U1;
      U2;
    end;
    begin
      U3;
      cobegin
        U4;
        U5;
      coend;
      U6;
    end;
  coend;
  U7;
end;
```



Folyamatdeklaráció

- A ténylegesen megvalósított konkurens programozási nyelvekben (CPascal, MODULA, ADA) folyamatdeklaráció (process declaration) utasításokkal definiáljuk a folyamatokat
- Ez olyan, mint az eljárásdeklaráció a hagyományos nyelvekben



Hol tartunk?

- 1 Ismétlés
 - Emlékeztető az előző órákról
- 2 Klasszikus konkurens problémák
 - Miért foglalkozunk ezekkel?
 - Termelő-fogyasztó probléma
 - Írók-olvasók problémája
 - Étkező filozófusok problémája
 - Adatfolyamok illesztése
- 3 Nyelvi eszközök a folyamatok programozására
 - Folyamatokból álló rendszerek programozása
 - A párhuzamosság leírása
- 4 Befejezés
 - Emlékeztető kérdések



Emlékeztető kérdések

- 1 Mutasd be a Termelő-fogyasztó problémát!
- 2 Mutasd be az Írók-olvasók problémáját!
- 3 Mutasd be az Étkező filozófusok problémát!
- 4 Mutasd be az Adatfolyamok illesztésének problémáját!
- 5 Mutasd be a precedenciagráf fogalmát! Mire használjuk?
- 6 Hogyan lehet leírni a precedenciagráfot különböző nyelvi eszközökkel?
- 7 Hasonlítsd össze a fork-join és a parbegin-parend utasítaspárokat!



Befejezés

Köszönöm a figyelmet!

