Operációs rendszerek - párhuzamos folyamatok.

Folyamat, szál fogalma és megvalósításaik.

Folyamatok

A folyamat egy futó program példánya. Annyiszor jön létre, ahényszor a programot elindítjuk.

Egy folyamat élettartamának vezérlése az operációs renszer feladata:

- létrehozás
- ütemezés (folyamatok közti váltás)
- megszüntetés

A folyamat állapotai:

- blokkolt: erőforrásra, felhasználói beavatkozásra vár
- futó: Az aktuálisan végrehajtás alatt álló folyamat. (Egy processzoron egyszerre egy futó folyamat lehet. Üres processzoror az *idle* folyamat fut, ami nem csinál semmit.)
- futáskész: ütemezésre váró folyamat

A futó folyamat új erőforrás igénylésekor blokkoltá válik, a blokkolt az erőforrás megszerzésekor futáskésszé. A futó és futáskész folyamatokat az oprendszer váltogatja.

A folyamatváltás menete (folyamatok megvalósítása)

- Megszakítás, kivétel vagy rendszerhívás történik.
- Meg kell őrizni a processzor aktuális állapotát (utasításmutató, regiszterek értéke).
- Be kell tölteni az ütemezendő folyamat előzőleg elmentett állapotát.

A processzor állapotának mentése hardver és szoftverigényes, a gyorsítótár tartalma váltáskor elvész, így a gyakori váltás maga overhead-et eredményez.

Többprocesszoros rendszerek

A processzorok fizikailag és funkcióilag is különbözőek lehetnek, mindegyikre csak a neki megfelelő programot tartalmazó folyamat ütemezhető.

A gyorsítótár tartalmazhatja a nemrég futtatott folyamatok által gyakran hivatkozott memóriacímeket, ezért érdemes ilyen folyamatot választani az ütemezésre.

A folyamatleíró

A folyamat adatait tartalmazza. Lehetséges tartalma:

- folyamatazonosító
- folyamattípus
- futó program neve

- tárterületek, elmentett processzor adatok
- tulajdonos felhasználó
- hozzáférési jogok, különböző erőforrásokhoz
- korlátok, kvóták
- statisztikai adatok
- kommunikációs adatok
- ütemezési adatok

A folyamatok hierarchiája

Fa vagy erdő struktúrát alkotnak.

Az új folyamat a létrehozó folyamat gyerekfolyamata.

Szülő folyamat megszűnésekor a gyerekfolyamat:

- szintén megszűnik, vagy
- leválasztódik, és egy új fa gyökere lesz, vagy
- leválasztódik, és az őt tartalmazó fa gyökere lesz az új szülője

Gyerekfolyamatok a szülő adatait örökölhetik.

Ha a folyamatok erdőt alkotnak, akkor az erdő egy fája a munka, melynek lehetnek saját adatai.

A folyamattáblázat

- Rendszer összes folyamatleíróját tartalmazó táblázat a folyamattáblázat
- Általában folyamatleírókból álló vektor, esetleg láncolt lista
- A folyamattáblázat inicializálása:
  - Rendszerbetöltő beleírja a legelső folyamatot
  - Kezdeti táblázat a háttértárról töltődik be, ami a legelső folyamatot tartalmazza

Szálak

Egy program egymással párhuzamosan futó részei a szálak.

A folyamatoknál szorosabban összetartoznak, a programozó határozza meg őket.

Intenzívebb kommunikáció egymással.

Sok közös adat: például felhasználó, jogok, korlátok, statisztikai adatok stb.

Bizonyos adatokat szálanként külön kell nyilvántartani: pl. processzor regiszterei.

Szálak megvalósítása

1.

- A többszálú program saját ütemezőt tartalmaz, így egyetlen folyamatként fut, a szálak rejtve

maradnak az operációs rendszer elől (green thread).

- Kommunikáció folyamaton belül történik, közös memóriaterületen, nincs szükség rendszerhívásra.
- Szálak közötti kapcsolás felhasználói védelmi szinten történik, nincs szükség hozzá megszakításra, kivételre vagy renszerhívásra.
- Egyik szál blokkolódik operációs rendszer ezt nem tudja, így az egész folyamatot blokkolja.

2.

- Minden szál külön folyamatként fut, az operációs rendszer nem tud róla, hogy összetartoznak (native thread).
- Folyamatok egymástól függetlenül blokkolódhatnak illetve kerülhetnek vissza futáskész állapotba.
- Kommunikáció rendszerhívásokon keresztül történik.
- Szálak közötti kapcsolás lassú.
- Megoldás: operációs rendszernek tudnia kell a szálakról.

Interaktív, kötegelt és valós idejű folyamatok, ütemező algoritmusok.

- Interaktív folyamat
  - Futás közben állandó kommunikáció a felhasználóval
  - Interakció: felhasználóval való kommunikáció
  - Tipikus interakció:
    - Nagygépes terminálon adatok bevitele Enter-rel lezárva
    - Karakteres terminálon egy billentyű lenyomása
    - Grafikus felületen egér megmozdítása, gombjának megnyomása
  - Tranzakció: két interakció közötti feldolgozás
  - Ha ez lassú (hosszú válaszidő), úgy érezzük, hogy „akad” a gép
  - Példák: munka a számítógép előtt, azaz szövegszerkesztés, rajzolás, bizonyos játékok
- Kötegelt folyamat
  - Felhasználótól menet közben nem vár adatot
  - Futásához szükséges idő néhány másodperctől több napig terjedhet
  - Kisebb csúszásokat általában észre sem veszi a felhasználó
  - Régen lyukkártyákon voltak az elvégzendőműveletek, ma parancsállományban, amit a felhasználó egyik interaktív folyamata ad át az operációs rendszer számára (pl. időzítve)
  - Példák: pénzintézetek éjszakai vagy hétvégi feldolgozásai (pl. kamatok könyvelése),

mentések, hálózati letöltések, víruskeresés

- Valós idejű folyamat
  - Események történnek, melyekre megadott határidőn belül kell reagálnia egy folyamatnak
  - Lágy valós idejű rendszer: kisebb csúszások megengedettek, de az események torlódása nem
  - Szigorú valós idejű rendszer: kisebb csúszás is katasztrofális következményekkel járhat
  - Különböző fontosságú események lehetnek
  - Példák: beágyazott rendszerek (szabályozás), multimédia (játékok is)

ALGORITMUSOK

- Preemptív ütemezés: Az oprendszer elveheti a futás jogát az éppen futó folyamattól, és "futásra kész" állapotúvá teheti. Közben egy másik folyamatot indíthat el. *
- Nem preemptív ütemezés: Az operációs rendszer nem veheti el a futás jogát a folyamattól. A folyamat addig fut, amíg az általa kiadott utasítás hatására állapotot nem vált, azaz csak maga a folyamat válthatja ki az új ütemezést. (Befejeződik, erőforrásra vagy eseményre vár, lemond a futásról)*

Egyszerű ütemezési algoritmusok

- Legrégebben várakozó (FIFO: First In First Out): Nem preemptív. A legrégebben várakozó folyamatot választja ki futásra. A futásra kész folyamatok egy várakozási sor végére kerülnek, az ütemező pedig a sor elején álló folyamatot kezdi futtatni. Hátránya: Nagy lehet az átlagos várakozási idő, mert egy hosszú CPU löketű folyamat feltarja a mögötte levőket és a perifériák is tétlenek. (konvoj hatás).*
- Körbenforgó (RR: Round Robin): Preemptív algoritmus, az időosztásos rendszerek alapja. Minden folyamat, amikor futni kezd kap egy időszeletet. Ha a CPU lökete ennél nagyobb, az időszelet végén az ütemező elveszi a folyamattól a CPU-t és a futásra kész várakozási sor végére rakja. Ha a CPU löket rövidebb az időszeletnél, akkor a folyamatokat újraütemezzük, és a futó folyamat időszelete újraindul. Hátránya: Nehéz az időszelet megfelelő méretének a meghatározása. Túl hosszú esetén átmegy FCFS algoritmusba, túl rövid esetén sok a környezetváltás. *

Prioritásos ütemezési algoritmusok

- Legrövidebb (löket)idejű (SJF: Shortest Job First): Nem preemptív. A legrövidebb becsült löketidejű folyamatot választja ki futásra. Ennél az algoritmusnál optimális az átlagos várakozási és körülfordulási idő. A felhasználó által megadott és az előzmények alapján becsült löketidőt veszi alapul. *
- Legrövidebb hátralevő idejű (SRTF: Shortest Remaining Time First): Az SJF preemptív változata. Ha új folyamat válik futásra készzé akkor megvizsgálja hogy a futó folyamat hátralevő löketideje vagy az új folyamat löketideje a kisebb. A környezetváltás idejét is figyelembe veszi. *

---

* Nem ELTÉs jegyzetből van. A CPU löket idő az az idő ameddig a folyamat a CPU-t használja.

- Legjobb válaszarány(HRR: Highest Reponse Ratio): A folyamat kiválasztásánál a löketidőt és a várakozási időt is figyelembe veszi. Öregítést alakalmaz (aging), azaz a régóta várakozó folyamatok prioritását növeli.*

Párhuzamosság fajtái, versenyhelyzetek, kritikus szekciók.

- Egyszerre több folyamat fut a rendszerben
- Az ütemező a folyamatok közötti gyors váltogatással teremti meg a párhuzamos futás illúzióját
  - Interaktív rendszerek
  - Időosztás (time sharing, 60-as, 70-es évek)
  - Látszat-párhuzamosság

Valódi párhuzamosság

- Többprocesszoros rendszerek
  - Akár processzorok százai egy számítógépben
  - Közös sín, közös órajel, akár közös memória és perifériák, gyors kommunikáció
  - Gazdaságos teljesítmény-növelés
  - A megbízhatóságot általában nem növeli!
- Klaszterek (fürtök)
  - Önállóan is működőképes számítógépek együttműködése
  - Egységes rendszer illúziója
  - Lassú kommunikáció (tipikusan helyi hálózat)
  - Magas rendelkezésre állás

Versenyhelyzetek

- Versenyhelyzet (race condition): párhuzamos végrehajtás által okozott nemdeterminisztikus hibás eredmény
- Nehezen felderíthető hiba
  - Nemdeterminisztikus
  - A hiba hatása nem azonnal jelentkezik
  - Megértéséhez alacsonyszintű ismeretek szükségesek
- A versenyhelyzeteket el kell kerülni

Kritikus szekciók

- A versenyhelyzetünk feloldásához biztosítanunk kell, hogy a számláló-t egy időben csak az egyik folyamat változtassa
- A számláló növelése és csökkentése rendszerünk kritikus szekciója

- Követelmények:
  - Kölcsönös kizárás: két folyamat egyszerre nem lehet kritikus szekcióban
  - Haladás: a kritikus szekcióhoz egyszerre érő folyamatok közti döntésben csak ők vehetnek részt
  - Korlátozott várakozás: előbb-utóbb minden belépni szándékozó folyamatnak sorra kell kerülnie

Folyamatváltások letiltása

- A kritikus szekció közben tiltsuk le a folyamatváltásokat okozó megszakításokat
- Legkézenfekvőbb megoldás
- Általában nem használható, az OS tiltja
  - Üzembiztonsági megfontolások
  - Az operációs rendszeren belül bevett módszer
- Valódi párhuzamosság esetén nem hatásos

Szigorú váltogatás

- Egy zálog változó körbejár a folyamatokon
- Akinél a zálog, az léphet be a kritikus szekcióba
- Megvalósítás:

```
while (1) {
while (zálog != sorszámom())
; /* várakozás */
kritikus_szekció();
zálog = (zálog + 1) % folyamatok_száma();
nemkritikus_szekció();
}
```

- Követelmények ellenőrzése:
  - Kölcsönös kizárás
  - Haladás (csak felváltva lehet kritikus szekcióba lépni)
  - Korlátozott várakozás

Peterson megoldása (1981)

- Két folyamat esetében működő megoldás
- Nyilvántartja, hogy mely folyamatok akarnak kritikus szekcióba lépni
- A zálog csak akkor számít, ha mindkét folyamat verseng érte
- Implementáció:

```
while ( 1 ) {
verseng[sorszámom()] = TRUE;
```

```
zálog = másik();
while (verseng[másik()] && zálog == másik());
/* várakozás */
kritikus_szekció();
verseng[sorszámom()] = FALSE;
nemkritikus_szekció();
}
```

- Mindhárom követelmény teljesül

Sorszámhúzásos szinkronizáció

- Tetszőleges számú folyamatra
- Ügyfélfogadó rendszerek mintájára
- Implementáció: kezdetben sorszám[1..n] = 0
  - Belépés (i. folyamat):

```
sorszámot_húz[i] = TRUE;
sorszám[i] = max(sorszám[1..n]) + 1;
sorszámot_húz[i] = FALSE;
for (j = 0; j < n; j++) {
while (sorszámot_húz[j]); /* vár */
while (sorszám[j] != 0
&& (sorszám[j], j) <lex (sorszám[i], i))
; /* vár */
}
```

  - Kilépés:

```
sorszám[i] = 0;
```

- Mindhárom követelmény teljesül

TSL utasítás

- Szoftver helyett a hardvert bonyolítjuk
- TSL (Test and Set Lock) utasítás, melynek működése a következő:

```
boolean TSL (boolean & zár) {
boolean érték = zár;
zár = true;
return érték;
}
```

- A TSL utasítás legyen
  - atomi
  - megszakíthatatlan, és
  - ugyanarra a zárra egyszerre csak egy processzor hajthatja végre
- Így jóval egyszerűbb a szinkronizáció:

- Belépés:

```
while (TSL(zár))
; /* várakozik */
```

- Kilépés:

```
zár = false;
```

- A modern processzorokban általában van ilyen utasítás
- Korlátozott várakozás feltétele nem teljesül!

TSL + korlátozott várakozás

- A várakozó folyamatok nyilvántartásával kijavítható a megoldásunk:
  - Belépés:

```
várakozik[i] = TRUE;
while (várakozik[i] && TSL (zár))
; /* vár */
várakozik[i] = false;
```

  - Kilépés:

```
j = i + 1;
do {
j = (j + 1) % n;
} while (j != i && !várakozik[j]);
if (j == i)
zár = false;
else
várakozik[j] = false;
```

- Ez már mindhárom követelménynek eleget tesz

Szemaforok és monitorok.

Szemaforok

- E. W. Dijkstra, 1965
- Általános célú megoldás, a szinkronizáció megkönnyítésére
- A szemafor egy új, egész értékű típus
- Két atomi művelet (plusz kezdőértékadás)
- down(S): Ha S értéke zérus, várakozik, míg S pozitívvá nem válik. Egyébként, ill. ezután S-t eggyel csökkenti. (Eredetileg: P(S), a holland proberen, megpróbálni szóból.)
- up(S): S értékét eggyel növeli. (Eredetileg: V(S), a holland verhogen, megemelni szóból.)
- Megengedett, hogy egyszerre több down művelet legyen várakozó állapotban

Szemaforok alkalmazásai

- A kritikus szekció problémája:

- Kezdetben legyen mutex értéke 1. (mutual exclusion)
- Belépés: down(mutex);
- Kilépés: up(mutex);

```
while (1) {
down(mutex);
kritikus_szekció();
up(mutex);
nem_kritikus_szekció();
}
```

- Tetszőleges számú folyamatra működik
- Kölcsönös kizárás és haladás feltétele teljesül
- Korlátozott várakozás teljesítése az up implementációjától függ
- Ha mutex értékét k-ról indítjuk, egyszerre legfeljebb k darab folyamat lehet a kritikus szekcióban
- Ha az A folyamat P utasítását mindig a B folyamat Q utasítása előtt kell végrehajtani:
  - A folyamat:

```
P;
up(szinkr);
```

  - B folyamat:

```
down(szinkr);
Q;
```

  - A szinkr szemafor kezdeti értéke legyen 0.

Szemaforok implementációja

- Egyszerű implementáció:
  - down(S):

```
while (S <= 0); /* vár */
S = S – 1;
```

  - up(S):

```
S = S + 1;
```

- Feltételezések:
  - S értékváltoztatásai atomiak
  - A down műveletben a ciklusfeltétel hamissá válása és az értékcsökkentés együtt is oszthatatlan művelet
- Általában processzorok speciális utasításai segítik az implementációt

- Neve: spinlock (kb. „pörgő zár”)
  - Multiprocesszoros operációs rendszerek implementációja
- Tevékeny várakozás
  - A várakozó folyamatok ciklusban futnak
  - Csak nagyon rövid ideig szabad így várakozni
  - Eddig minden megoldásunk ilyen volt!
- Hogyan küszöbölhetjük ki a tevékeny várakozást?
- A szemafor értéke mellett jegyezzük meg, hogy mely folyamatok várakoznak rá!

Monitorok

- A monitor a szemaforoknál magasabb, nyelvi szintű szinkronizációs eszköz
- Hoare (1974) és Hansen (1975)
- Közös használatú változókat és a rajtuk végezhető műveleteket fogja egységbe
- A változók csak a monitorműveleteken keresztül érhetők el
- Egyszerre csak egy művelet lehet aktív
- Szintaxis:

```
monitor név {
változók deklarációja
állapotok deklarációja (lásd később)
eljárások deklarációja
}
```

- Példa: védett változó

```
monitor védett_int {
  int v;
  void növel(int d) { v = v + d; }
  void csökkent(int d) { v = v – d; }
  int beállít(int új) { /* TSL */
    int i = v;
    v = új;
    return i;
  }
int lekérdez() { return v; }
}
```

- A műveletek törzsei kritikus szekciók: a fordítóprogram garantálja a kölcsönös kizárást
- A monitorok állapotai segítségével más típusú szinkronizációs feladatokat is megoldhatunk
- Állapotok deklarációja: condition x;
- Két állapotművelet
  - x.wait(): Az állapot bekövetkeztére vár, eközben más folyamatok is beléphetnek a

monitorba

- x.signal(): Jelzi az állapot bekövetkeztét. Ha nincs az állapotra váró folyamat, nem csinál semmit; egyébként kiválaszt egy várakozó folyamatot, és felébreszti

- A signalt hívó folyamat a felébresztett monitorhívás befejeződésig várakozik

Osztott memória és üzenetküldés.

Üzenetküldés

- Az eddigi példáinkban mindig osztott memóriát használtunk
  - Egy számítógépen belül hatékony
  - Klaszterek, elosztott rendszerek esetében általában nem áll rendelkezésre, vagy túl lassú
  - Egyes feladatokat nem így a legkézenfekvőbb megoldani
- Másik fő általános célú kommunikációs eszköz: üzenetküldés
  - Különböző számítógépeken futó folyamatok kommunikációjának természetes módja (hálózatok)
  - Gyakran kényelmesebb üzenetváltásokban gondolkodni
- Példák üzenetalapú kommunikációra:
  - UNIX rendszerek csővezeték (pipe) szolgáltatása cat szöveg.txt | tr " " "\n" | sort | uniq -c | sort -nr | head -10
  - Mindennemű hálózati kommunikáció
  - PVM, MPS
- Két alapművelet
  - Üzenetek küldése: send(címzett, üzenet);
  - Üzenetek fogadása: üzenet = receive(feladó);
  - Ha feladó == –1, akkor bárkitől elfogadjuk az üzenetet
- Felmerülő tervezési kérdések:
  - A kommunikációs csatornát, vagy a csatorna végpontjain ülő folyamatokat választjuk a feladó, ill. címzett azonosítására?
  - Megvárja-e a send, hogy a címzett fogadja az üzenetet?
  - Megvárja-e a receive, míg kap egy üzenetet, vagy mindig azonnal visszatér?
  - Ha mindkettő várakozik, randevúról beszélünk
  - Van-e várakozási sor a még feldolgozatlan üzenetek számára?
  - Ha igen, mekkora? Nulla, véges, vagy dinamikusan növekedő?

Üzenetküldés implementációja

- Az üzenetküldést meg lehet valósítani osztott memória segítségével
  - A gyártó-fogyasztó probléma speciális esete
  - Hatékonyan megoldható
  - Egy számítógépen belüli üzenetküldésre gyakorta használják
- Fordítva is működik: osztott memóriát is emulálni lehet üzenetküldéssel
  - Nem egyszerű
  - Általában rendkívül kevéssé hatékony
  - Csak speciális célokra használják (VMS zárak)

Holtpontok, jellemzésük, megelőzésük, elkerülésük és felismerésük.

Holtpontok

- Egy folyamathalmaz holtpontban van, ha minden eleme valamelyik másik folyamat által kiváltható eseményre várakozik

Erőforrások

- A holtpontban lévő folyamatok tipikusan egy erőforrás felszabadítására várnak
- Erőforrások:
  - Szemaforral védett memóriaterület
  - I/O eszközök
  - Fájlok, rekordok olvasásának/írásának joga
  - Processzorhasználat joga
  - Új memóriaterület igénylése
- Az egyes erőforrástípusokból több példány is rendelkezésre állhat
  - Mindegy, hogy melyiket kapja meg a folyamat
  - Példa: memóriafoglalás
- Holtpont létrejöttéhez az alábbi négy feltételnek egyszerre teljesülnie kell:
  - Kölcsönös kizárás: legyen legalább egy megoszthatatlan erőforrás
  - Birtoklás és várakozás: A folyamatoknak nem kell feladniuk az eddig megszerzett erőforrásokat ahhoz, hogy újakat igényelhessenek
  - Megszakíthatatlanság: az erőforrások nem vehetők el a folyamatoktól a beleegyezésük (felszabadításuk) nélkül
  - Körkörös várakozás: lennie kell egy két vagy több folyamatból álló ciklikus listának, amiben minden folyamat a következő egy erőforrására várakozik

Holtpontkezelés

- Holtpont-megelőzés: valamelyik szükséges feltétel kizárása
- Holtpont-elkerülés: erőforrás-gazdálkodás a szükségletek előzetes ismeretében
- Holtpont-felismerés: a már létrejött holtpontok utólagos felszámolása
- Strucc-módszer: tegyünk úgy, mintha a probléma nem létezne
  - A holtpontok viszonylag ritkák
  - A lekezelésük erőforrásigényes, és kényelmetlen szigorításokat okoz
  - A legtöbb kisgépes operációs rendszerben nincs holtpontkezelő rendszer (UNIX, Windows)

Holtpont-megelőzés

- Ha garantálni tudjuk, hogy a négy feltétel valamelyike biztosan nem áll fenn, a holtpontokat el tudjuk kerülni
- Első feltétel: kölcsönös kizárás
  - Ha az összes felhasznált erőforrás megosztható, akkor sohasem kell várni, a rendszer nem kerülhet holtpontba
  - Példa: csak olvasható fájlok
  - Háttértárolásos nyomtató (spooling)

Birtoklás és várakozás

- További erőforrások igénylésének tiltása
  - Az összes szükséges erőforrást egyszerre kell igényelni, a folyamat indulásakor
  - Néha a program indulásakor még nem tudja, milyen erőforrásra lesz szüksége
  - Pazarló erőforrás-gazdálkodást eredményez
- Új erőforrás igényléséhez el kell engedni az eddig megszerzetteket
  - Kiéheztetéshez vezethet
  - Kényelmetlen

Megszakíthatatlanság

- Ha a folyamat erőforrásra várakozik, akkor eközben a már megszerzett erőforrásait mások elvehetik tőle
- Csak néhány erőforrástípus esetén alkalmazható
  - Pl. processzor, fizikai memória
  - I/O eszközök, fájlok esetén nem!

Körkörös várakozás

- Számozzuk meg az erőforrásokat, és kötelezzük a folyamatokat, hogy növekedő sorszám szerint igényeljék azokat

- Kizárjuk a hurkok lehetőségét
- Lehetetlen minden folyamatnak megfelelő sorrendet kitalálni
- Pazarló erőforrás-gazdálkodás

Holtpont-elkerülés

- Ötlet: ha előre ismerjük a folyamatok viselkedését, el tudjuk kerülni a holtpontokat

Biztonságos állapotok

- Tegyük fel, hogy előre ismerjük a folyamataink maximális erőforrásigényeit
  - Az egyes erőforrásfajtákból igényelt egységek maximuma
- A rendszer állapota biztonságos, ha a folyamatokat valamilyen sorrendben be lehet úgy fejezni, hogy közben garantáltan nem áll elő holtpont
  - Biztonságos folyamatsorrend: bármely folyamat maximális erőforrásigénye teljesíthető a rendelkezésre álló és a sorrendben előtte álló folyamatok által lefoglalt erőforrásokból
  - Egy holtpontban lévő rendszer állapota nem lehet biztonságos
- Holtpont-elkerülés: tartsuk rendszerünket biztonságos állapotban

Bankár-algoritmus

- Kövessük a készpénzkészletükkel gazdálkodó kisvárosi hitelezők viselkedését
- Legyen m típusú erőforrásunk, és n folyamatunk
- Ismert adataink:
  - Szabad erőforrásegységek: Szabad[1..m]
  - Az i. folyamat maximális igénye: Max[i,1..m]
  - Az i. folyamat aktuális foglalása: Foglalt[i,1..m]
  - Hátralévő foglalások: Többi[i,1..m]
  - Többi[i] = Max[i] – Foglalt[i]
  - (Jelölés: Többi[i] alatt a Többi[i,1..m] vektor értendő, a kivonás tagonkénti kivonás)
- Biztonságosság eldöntése:
  - Legyen Munka[1..m] = Szabad[1..m], Végzett[1..n] = false
  - Ha a Végzett vektorban nincs hamis érték, az állapot biztonságos. (Végeztünk)
  - Ha nincs olyan i, hogy Többi[i] ≤ Munka, akkor az állapot nem biztonságos. (Végeztünk)
  - Legyen Munka := Munka + Foglalt[i] és Végzett[i] := true, majd folytassuk a 2. lépésnél
- Erőforrásigény: O(m·n2)
- Erőforrás-foglalás:
  - Tegyük fel, hogy az i. folyamat a Kérés[1..m] vektorban adott erőforrásokat igényli

- Ha Kérés > Többi[i], akkor a kérést megtagadjuk
- Ha Kérés > Szabad, akkor várakozni kell
- Egyébként döntsük el, hogy az új állapot biztonságos lenne-e. Ha igen, akkor a foglalást engedélyezzük, és karbantartjuk az állapotváltozóinkat (lásd alább); különben várakoztatjuk

```
Szabad := Szabad – Kérés
Foglalt[i]:= Foglalt[i] + Kérés
Többi[i] := Többi[i] - Kérés
```