Versenyhelyzet és holtpont nélkül

Sorozatunk előző részéből már kiderült, hogy az operációs rendszer összes szolgáltatása a folyamatkezelésen alapul. Az előző részben végig a folyamatokat és a velük szorosan összefüggő fogalmakat tárgyaltuk, korántsem értünk azonban a témakör végére.

Tessék, még mindig a folyamatoknál tartunk! Nehezen tudunk elszakadni ettől a témától, pedig hány más izgalmas dologról nem esett még szó, például a fájl- és memóriakezelésről, vagy a beviteli és kiviteli eszközök vezérléséről. Az előző részben azt tárgyaltuk, hogy az operációs rendszerek legalapvetőbb szolgáltatásokat ellátó részei is – például a memóriakezelő vagy az eszközmeghajtók – egy-egy folyamatként, a felhasználói alkalmazásokkal párhuzamosan futnak (igaz, alacsonyabb, gépközelibb szinten). A folyamatkezelés tehát a legfontosabb, ha nem is a legérdekesebb elem az operációs rendszerek életében.

A feladatkezelés tárgyalásakor még nem érintettük az úgynevezett ***szálakat***, illetve nem esett szó a folyamatok közötti kapcsolattartás megvalósításáról. Ez utóbbi nagyon fontos, az erőforráskezelés ugyanis teljes egészében erre a rendszerre fog épülni.

A szálak

Tegyük fel, hogy barátunktól megkaptuk egy weboldal címét, ahol topmodellek fürdőruha-divatbemutatójáról készült fényképek tömkelege található. A férfi felhasználók többsége az ilyenekre felkapja a fejét, és rögtön be is „izzítja” kedvenc böngészőprogramját.

Az ehhez hasonló oldalak általában rengeteg kis képet tartalmaznak, nem is beszélve az elhelyezett reklámok sokaságáról. Elméletben egy weboldal betöltése a következőképpen zajlana: a böngésző először megkapja a HTML-forrást, amelyben többek között le van írva az oldalon megjelenítendő képek elérési helye. Ezután sorban egymás után letöltjük a képeket a saját gépünkre, majd megjelenítjük az oldalt.

A valóságban ez a módszer nem hatékony. Ahhoz, hogy egy képet letölthessünk a kiszolgálóról, először kapcsolódnunk kell, meg kell várnunk, amíg a kép teljes egészében lejön, majd bontanunk kell a kapcsolatot. Kis képekről lévén szó a kapcsolatok létrehozása és lebontása csaknem annyi időbe telik, mint maga az adatátvitel, tehát rengeteg idő kárba vész. Több száz kép esetében annyira idegölő lehet ez a sok üresjárat, hogy fürdőruhás topmodellek ide vagy oda, a felhasználónak elmegy a kedve az egésztől.

A legjobb megoldás az lenne, ha a letöltéseket ***párhuzamosíthatnánk***. Az ilyen és ehhez hasonló nehézségek feloldására – amikor egy folyamaton belül is további párhuzamosításra van szükség – találták ki az úgynevezett ***pehelysúlyú folyamatokat***, közismertebb nevükön a ***vezérlési szálakat***.

Egy vezérlési szálat úgy is elképzelhetünk, mint utasításokból álló sorozatot. Az első helyen van az első végrehajtandó utasítás, a másodikon a második és így tovább. Minden szálhoz tartozik egy ***utasításszámláló***, amely megmondja, hogy éppen hol tartunk az utasítások végrehajtásában.

Nem meglepő, hogy minden folyamatnak legalább egy vezérlési szállal és egy utasításszámlálóval rendelkeznie kell. A korszerű operációs rendszerek azonban azt is megengedik, hogy egy folyamat további vezérlési szálakat hozzon létre.

A szálak sok tekintetben hasonlítanak a folyamatokhoz. Itt is megtalálható a három alapvető állapot (futó, futásra kész, illetve blokkolt). A tárolásukról egy úgynevezett ***száltáblázatban*** kell gondoskodnunk, amely többek között az utasításszámlálókat tartalmazza, amelyek azt mondják meg, hogy éppen hol függesztettük fel a szálak futását.

A szálak azonban nem folyamatok a folyamatban. Míg ugyanis a folyamatok külön címtartománnyal (memóriaszelettel) rendelkeznek, amelyekbe más folyamatok nem „túrhatnak bele”, addig a szálak közösen osztoznak folyamatuk címtartományán.

A szálak megvalósítására két út is kínálkozik. Az első, az operációs rendszer a rendszermag szintjén támogatja, hogy egy folyamat több vezérlési szállal rendelkezzen. A második, a rendszer egyáltalán nem törődik a vezérlési szálakkal, a megvalósítás a felhasználóra van bízva (azaz maga a folyamat fog a saját vezérlési szálainak kezeléséről gondoskodni). Vajon melyik megoldás a nyerő?

Aki úgy gondolja, hogy az első variáció a jobb, azaz a szálak kezelését az operációs rendszerre kell bízni, igaza van. Tegyük fel, hogy van egy folyamatunk két vezérlési szállal, és a szálak kezelése a felhasználói területen történik. Semmi gond sem lesz egészen addig, amíg az egyik szál valami oknál fogva nem blokkolódik (például azért, mert visszajelzésre vár egy eszköztől). Ebben az esetben az ütemező az egész folyamatot blokkolni fogja. Ha azonban az ütemező tudna a szálak létezéséről, akkor egy szál blokkolása esetén az adott folyamat szálai közül kiválasztana egy futásra kész szálat. Ha nincs ilyen, egy másik folyamatban keres futásra kész szálat.

Az sem téved viszont, aki úgy gondolja, hogy a szálak felügyeletének a felhasználó hatáskörébe kell tartoznia. A sebesség szempontjából sokkal hatékonyabb, ha a szálak közötti kapcsolgatást felhasználói szinten végezzük, és nem a rendszermagon keresztül. Mivel tökéletes megoldás nincs, az operációs rendszerek mind a kétféle módszert egyszerre használják.

Sajnos, korántsem ez az egyetlen bonyodalom a szálak kezelésének a kérdésében. Így nehézséget okoz, hogy a folyamat szálai közös adatterületen osztoznak. A legklasszikusabb példa egy rendszerhívás végrehajtása. Az első szál meghív egy rendszerhívást. A Linux esetében ennek sikeréről (vagy kudarcáról) az `errno` nevű globális változó ad felvilágosítást. Mielőtt azonban ellenőrizni tudnánk az `errno` értékét, szálváltás történik, és a második szál is végrehajt egy rendszerhívást, ennek hatására pedig az `errno` értéke megváltozik. Amikor a vezérlés visszakerül az első szálra, az már hamis adatokkal fog dolgozni. Ez viszont csak a jéghegy csúcsa. Mi történik akkor, ha az egyik szál látja, hogy nincs elég memória, ezért újabb blokkokat kezd lefoglalni, miközben egy szálváltás következtében a másik szál ugyanígy cselekszik?

A szálak bevezetése nehezen leküzdhető feladatok elé állítja az operációs rendszerek fejlesztőit. Valószínű, hogyha egy meglévő rendszerbe szeretnénk beültetni a vezérlési szálakat, azt nem úszhatjuk meg a rendszer alapjainak újratervezése nélkül.

A folyamatok kapcsolattartása

Az előző részben alaposan kivesézték a folyamatkezelés egyik legfontosabb elemét, az ütemezést. Most egy másik, nem kevésbé elhanyagolható részt tekintünk át.
A ***folyamatok közötti kapcsolattartás*** (InterProcess Communication, röviden IPC) megvalósítása is a rendszermagra hárul. Ha felidézzük magunkban az operációs rendszerek elvi felépítését bemutató táblázatot, akkor az IPC-t is valahol a legalsó réteg bugyraiban kell keresnünk. Az ütemező-megszakításkezelő réteg tehát újabb feladatot kapott: a folyamatok közötti kapcsolattartás megvalósítását és ellenőrzését.
Az IPC témakörébe nem csak az tartozik, hogy két folyamat miképp tud üzenetet küldeni egymásnak. Az IPC-nek olyan feladatokat is meg kell oldania, mint a versenyhelyzetek feloldása, illetve a folyamatok közötti összehangolás. Hogy ezek pontosan mit jelentenek, az alábbiakból mindjárt ki is derül.

Üzenetküldés

A Linux és Unix-rendszerek a folyamatok közötti kapcsolattartást úgynevezett ***üzenetküldéses rendszerrel*** valósítják meg. Ehhez mindössze két könyvtári eljárásra van szükség: egyre, amellyel üzenetet továbbíthatunk egy másik folyamatnak; és egy másikra, amellyel fogadhatjuk a nekünk küldött üzeneteket. Ezeket az eljárásokat a folyamatok bármikor meghívhatják.
Az üzenetküldéses rendszer előnyeit elsősorban az osztott rendszereknél élvezhetjük. Sorozatunk bevezető részében említettük: az a jól felépített operációs rendszer, amely átültethető akár több gépből összeállított osztott (telepekre) rendszerekre átültethető anélkül, hogy az egészet az alapjaitól fogva újra kellene terveznünk. Az üzenetküldés ebből a szempontból nagyon hatékony, mivel a folyamatok szemszögéből teljesen mellékes, hogy a kapcsolattartásban résztvevő másik folyamat a telep egy másik gépén fut-e vagy sem. Egy gép esetében azonban sokkal gyorsabb megoldásokat is találhatnánk az üzenetküldésnél.
De miként valósítsuk meg az üzenetküldést? A két legkézenfekvőbb megoldás a randevú és a levelesláda. Az első a következőképpen működik: ha egy folyamat üzenetet küld egy másik folyamatnak, akkor egészen addig blokkolódik, amíg azt a másik folyamat nem fogadja. Fordítva is így van. Ha egy folyamat meghívja az „üzenet fogadása” eljárást, mindaddig blokkol, amíg az üzenete meg nem érkezik. Ez a megoldás roppant egyszerű, mivel a rendszernek az üzenetet csak át kell másolnia az egyik folyamattól a másikhoz. Ez a módszer azonban megköveteli, hogy a kapcsolattartásban résztvevő két folyamatnak a futását egymáshoz kell igazítanunk.
A levelesláda ennél sokkal rugalmasabb. Minden folyamatnak fenntartunk egy memóriarekeszt, amely meghatározott számú üzenet ideiglenes tárolására képes. Ha az adott folyamatnak üzenete érkezik, a rendszer a levelesládába teszi, és egészen addig ott is tartja, amíg a folyamat ki nem szedi onnan.
A Unix-rendszerek a levelesláda-módszer egyik egyedi formáját használják, mégpedig a ***csővezetékeket*** (a pipe-okat). A csővezeték olyan levelesláda, amely az üzeneteket nem egymástól elválasztva, külön, hanem egyben, egy fájlként tárolja. A levelesláda ürítésekor a folyamat az összes üzenetet egy darabban fogja megkapni. A folyamatoknak kell gondoskodniuk arról, hogy a fogadó el tudja különíteni egymástól a beérkezett üzeneteket. Például úgy, hogy mindegyiket egy egyedi karakterrel zárja, vagy megegyezünk az üzenetek egy állandó méretében.

Versenyhelyzetek

Két vagy több folyamat olykor arra kényszerül, hogy közös tárterületen dolgozzon. Nem kell messzire mennünk egy hétköznapi példáért, csak a legközelebbi 12 termes multiplex moziba, ahol a helyfoglalásokat számítógépes adatbázisban tartják nyilván.
Tegyük fel, hogy minden pénztárhoz tartozik egy terminál, amelyeken a foglalásokat be lehet táplálni. A terminálok az

osztott adatbázist tartalmazó számítógéppel össze vannak kapcsolva. A rendszer minden terminálhoz külön folyamatot rendel, tehát a helyfoglalásokat az összes pénztárból egyszerre végezhetjük.
Képzeljük el, hogy egy időben két pénztárban is ugyanazt a helyet szeretnék lefoglalni (például azért, mert a teremben már csak egy hely maradt, és az utolsó pillanatban érkező két vendég ugyanazért a helyért verseng). Nézzük meg lépésenként, hogy mi történik a rendszerben!
Az első terminált kezelő folyamat az adatbázisból ellenőrzi, hogy a lefoglalni kívánt hely valóban szabad-e még. Mivel az, buzgón nekilát lefoglalni, azaz frissíteni az adatbázis megfelelő rekordját. Igen ám, de éppen ebben a pillanatban történik egy óramegszakítás, és az ütemező a másik terminál folyamatának adja át a vezérlést. Ámde ott is áll egy vendég, aki ugyanarra a helyre áhítozik. Ez a folyamat is ellenőrzi tehát, hogy szabad-e az adott hely (tudjuk, hogy szabad, mert a másik folyamat az adatbázis frissítését még nem tudta befejezni), majd lefoglalja azt. Ezután a vezérlés visszakerül az első folyamatra, amely mit sem tud arról, hogy közben valaki a „háta mögött” már megkapta a kérdéses helyet, így ez is lefoglalja ugyanazt. Ilyenkor úgynevezett ***versenyhelyzet*** alakul ki. Ennek lényege, hogy két (ritkább esetben több) folyamat egy időben ugyanahhoz az egymás között megosztott adathoz szeretne hozzáférni. Ha nem figyelünk arra, hogy abban a pillanatban valaki más is dolgozik az adott adattal, akkor a művelet végeredménye attól függ, hogy melyik folyamat mikor és hogyan futott. Sohasem jósolhatjuk meg tehát, hogy mi lesz egy létrejött versenyhelyzet eredménye.
Minden olyan rendszerben, ahol két vagy több folyamat megosztott adatokkal dolgozik, számolnunk kell a versenyhelyzetek kialakulásával. Vitatkozhatunk arról, hogy az előbb említett mozis példánál mekkora a valószínűsége egy versenyhelyzet kialakulásának. De a számítástechnika nem valószínűség-szá-

mítás. Másrészt pedig egy olyan rendszerben, ahol egy adatbázist egyszerre több százan írnak és olvasnak, és az adatok akár egy másodpercen belül többször is frissülhetnek, a versenyhelyzetek mindennapossá válhatnak. Ha egy rendszerben lehetőség van a versenyhelyzet kialakulására, de ez nem egészséges helyzet, lehetőleg el kell kerülni.

Egy folyamat „élete során" sok mindet olyasmit is tesz, ami nem fenyeget versenyhelyzettel. A belső vagy a saját memóriatartományán végzett számítások nem járnak ezzel a veszéllyel. Amikor azonban egy megosztott memóriarészhez vagy állományhoz nyúl, egyből megváltozik a helyzet. Ezért a folyamatok tevékenységének ezt a részét a folyamat ***saját kényes (kritikus) területének*** nevezzük.

A versenyhelyzetek elkerülésére a kézenfekvő megoldás a kölcsönös kizárás elve: meg kell tiltanunk, hogy egy időben egynél több folyamat is ugyanahhoz a megosztott dologhoz nyúlhasson. Másképpen szólva el kell érnünk, hogy egyszerre csak egy folyamat „tartózkodhasson" a saját kényes területén. Ennek megvalósítására az egyik legkézenfekvőbb megoldás az, ha valamilyen úton-módon az ütemezést a kényes feladatok idejére felfüggesztjük. Ahogyan sorozatunk előző részében is említettük, az egész ütemezés a megszakításrendszerre épül. Tehát a megszakításokat csak le kell tiltanunk egy erre megfelelő processzorutasítással. A kényes rész után pedig csak „vissza kell kapcsolni" azokat. Na de rendelkezhet egy közönséges folyamat ekkora hatalommal a számítógépünk felett? Mi történne, ha a saját kényes területén végtelen ciklusba kerülne? – lefagyna ez egész szekció. Ez nyilvánvalóan nem jó megoldás, tehát valami másra lesz szükség.

Próbálkozhatnánk esetleg egy olyan globális változó bevezetésével, amelynek megmondja, van-e valamilyen folyamat a saját kényes területén. A gyakorlatban ez úgy zajlana, hogyha egy folyamat be szeretne lépni a kritikus területre, a változó értékét ellenőriznie kell. Ha az 0, akkor belép, és beállítja 1-re; ha eredetileg nem 0 volt az értéke, akkor folyamatosan ellenőrzi, hogy mikor lesz az értéke ismét 0.

Ezt a folyamatos ellenőrzést ***tevékeny várakozásnak*** nevezzük, ugyanis a folyamat semmi érdemlegeset nem tesz, mégis zabálja a processzoridőt. Ám ennek a megoldásnak is akad hátulütője: egy kis szerencsétlenség következtében itt is kialakulhatnak versenyhelyzetek. Egy rosszul jött óramegszakítás következtében előfordulhat, hogy egyszerre két folyamat is bekerül a saját kritikus területére. Például az egyik folyamat ellenőrizte, hogy a változó értéke 0, és át akarja állítani 1-re, azonban közben történik egy váltás, és egy másik folyamat belép a saját kritikus területére. Miután a vezérlés visszakerül az előző folyamatra, az nem fogja ismét ellenőrizni a belépés lehetőségét, így egy időben két folyamat dolgozhat a megosztott adatokkal.

Bizonyos szempontból jobb megoldást jelent az úgynevezett ***szigorú váltogatás***, amikor is a kritikus területre való belépés jogát a folyamatok egymásnak „adogatják", tehát az A folyamat egészen addig nem léphet be, amíg a B folyamat ki nem lépett, viszont ezután a B folyamat sem léphet vissza mindaddig, amíg az A folyamat be nem fejezte a saját kényes területét.

A szigorú váltogatás megvéd minket a versenyhelyzetek kialakulásától, csakhogy olyankor nem hatékony, ha a két folyamat sebessége különböző, mivel a lassabb a gyorsabb folyamatot hátráltatni fogja.

Sokféle más megoldást is kitaláltak a tevékeny várakozásra alapozva, amellyel szigorú váltogatás nélkül is elkerülhetjük a versenyhelyzeteket. Az alapkérdés azonban még mindig fennáll: a tevékeny várakozás közben sok processzoridő megy kárba, azonkívül könnyen el lehet képzelni olyan egyedi helyzetet, ahol ez a fajta megoldás nem alkalmazható.

Erre találták ki az ***altatás-ébresztés rendszerét***. Egy folyamat, ha nem tud belépni a kényes területre, blokkolja magát. Az ébresztésről majd a másik folyamat fog gondoskodni, miután kilépett a kritikus területről.

Vannak azonban olyan esetek, amikor ez sem használható eredményesen (például az úgynevezett összekötött tároló megvalósításához). Erre találták ki a különböző jelzőket (semaphore), figyelőket (monitor) és még sok minden mást. A feladat és megoldási módjainak ismertetése azonban meghaladja e cikk kereteit, de a kíváncsibb olvasók az Interneten rendkívül sok anyagot találhatnak erről és az IPC más kérdéseiről (az összekötött tároló, közismertebb nevén a gyártó-fogyasztó kérdésére a legjobb megoldást az üzenetküldés alkalmazása jelenti).

A lényeg az, hogy olyan megoldást találni, amely minden helyzetben megvéd minket a versenyhelyzetektől, nagyon nehéz. De szerencsére sok esetben az üzenetküldés is felhasználható erre a célra.

Filozófusok és holtpontok

Bizonyos esetekben nemcsak a versenyhelyzetek elhárítására kell figyelmet fordítanunk, hanem arra is, hogy bizonyos eseménysorozatok ne következhessenek be. Ezt ***összehangolásnak*** nevezzük. Nézzünk erre egy ma már klasszikussá vált példát, az étkező filozófusok kérdését. Öt filozófus egy kerek asztal körül ül, előttük egy-egy tál spagetti, a tányérok között egy-egy villa. A filozófusok élete meglehetősen sivár, mivel csak gondolkodásból és evésből áll.

Ha egy filozófus sokáig gondolkozik, előbb-utóbb megéhezik, és ennie kell. Igen ám, de a spagetti annyira csúszós, hogy egy villával képtelenség megfogni, ezért a sikeres étkezéshez legalább két villára van szükség. A filozófus csak a tányérja melletti villákat próbálhatja megszerezni, és egyszerre csak egyet.

A feladat egy jó algoritmus megírása az étkezésre, amely sohasem akad el.

Ez nem is olyan egyszerű feladat. Nézzük például legegyszerűbb megoldást, amikor a filozófus először a bal, majd a jobb villát próbálja meg megszerezni. De mi történik, ha az összes filozófus egyszerre éhezik meg, és egy időben kaparintja meg a tőle balra lévő villát? Ebben az esetben egyik filozófusnak sem lesz esélye arra, hogy a jobb oldalit megszerezze, és kialakul az úgynevezett ***holtpont***.

A holtpontok és a versenyhelyzetek elkerülése is az IPC témakörébe tartozik, még ha ez nem is jár mindig közvetlen adatátvitellel. Nem jó, ha egy rendszerben előfordulhatnak holtpontok és versenyhelyzetek fordulhatnak elő.

Az étkező filozófusokkal tulajdonképpen ugyanaz a gond, mint a korlátozott számú beviteli-kiviteli eszközök vezérlésével, ha a villákat úgy fogjuk fel, mint egy-egy ilyen eszközt. A holtpontok tehát a következő részben is visszaköszönnek, amikor is a különböző eszközök kezelésével foglalkozunk, továbbá néhány olyan megoldást mutatunk be, amelyek segítségével elkerülhetjük a holtpontok kialakulását.

Garzó András
(garzoand@interware.hu) körülbelül három éve foglalkozik Linux- és más Unix-rendszerekkel. Legjobban az operációs rendszerek lelkivilága érdekli, de nyitott egyéniség. Kedvenc étele a palacsinta, és van egy Richard nevű macskája. Minden észrevételt, megjegyzést, levelet szívesen fogad.


Linuxvilág
www.linuxvilag.hu