Operációs rendszerek (12. rész)

Írásunkban a Unix, illetve a Linux memóriakezelőnek a rendszerhívásokkal való kapcsolatát tárgyaljuk, valamint a jelek, a fork és az exec használatát mutatjuk be.

Ez idáig memóriakezeléssel foglalkoztunk, pontosabban bemutattuk, hogyan működik a virtuális memória. Szó volt ennek gyakorlati megvalósításáról is, méghozzá éles rendszereken bemutatva. A memóriakezelő feladata azonban még nem ér véget ezen a ponton – például kezelnie kell a rendszerhívásokat és a jelekkel is foglalkoznia kell. Folyamatstrukturált rendszerekben a memóriakezelőt az eszközközeli rendszermagvilág és a felhasználói réteg közötti határként is felfoghatjuk. (Ez részben a monolitikus rendszerekről is elmondható, de ott az effajta rétegződés nem ennyire nyilvánvaló). A memóriakezelő szorosan együttdolgozik a maggal, de nem foglalkozik az eszközök alacsony szintű használatával: a felhasználói folyamatokkal törődik.
Ezért a memóriakezelő életében nagyon fontos szerepet töltenek be a rendszerhívások. Emlékezzünk vissza sorozatunk első részére, ahol azt mondtuk, hogy a rendszerhívásokkal érhetjük el az operációs rendszer szolgáltatásait. Ezeket a felhasználótól érkező rendszerhívásokat valakinek értelmeznie kell, a kívánt feladatot végre kell hajtania (vagy egy alacsonyabb szinten lévő alrendszerrrel el kell végeztetnie), majd a végeredményt vissza kell juttatnia a rendszerhívást végrehajtó számára. Erre a feladatra valóban a memóriakezelő a legalkalmasabb.
A rendszerhívásokat egyébként két részre oszthatjuk fel: azokra, amiket a memóriakezelő szolgál ki, és azokra, amiket a fájlrendszer. Az előbbi csoportba nemcsak azok a szolgáltatások tartoznak, amelyek egy-egy memóriablokk lefoglalását, illetve felszabadítását teszik lehetővé, hanem olyanok is, mint például egy gyermekfolyamat létrehozása, egy alkalmazás elindítása vagy például egy jel elküldése. Sorozatunk mostani részében ezekkel a rendszerhívásokkal foglalkozunk, a fájlrendszerrel pedig a következő hónaptól kezdve ismerkedünk meg részletesebben is. Mindenekelőtt elevenítsünk fel pár fogalmat!

Néhány alapvető fogalom

A továbbiakban néhány olyan alapvető unixos fogalommal fogunk dobálózni, amelynek az egyik részéről már szóltunk, a másik részéről még nem – az a közös bennük, hogy nem árt tisztában lenni a jelentésükkel.
Az első ilyen fogalom a folyamatazonosító, vagyis a PID (Process ID). Egy olyan természetes számról van szó, amiből minden folyamat birtokol egyet, ugyanis a rendszer csak ennek alapján azonosítja magát a folyamatot. Talán felesleges megjegyeznünk, hogy két folyamatnak egyidejűleg nem lehet ugyanaz az azonosítója. Sőt, ha a rendszer egy azonosítót egyszer már kiosztott, azt soha többé nem osztja ki újból (pontosabban addig, ameddig újra nem indítjuk).
Nemcsak a felhasználók, de a folyamatok is csoportokba szervezhetők. Minden folyamat rendelkezik egy folyamat-csoport-azonosítóval is. Ilyen módon lehetővé válik, hogy nemcsak egy folyamatnak, hanem egy egész folyamatcsoportnak jeleket küldhessünk (lásd később).
A felhasználókat és a (felhasználói) csoportokat szintén számokkal azonosítjuk, ez az uid és a gid, velük valószínűleg már mindenki találkozott. Ami viszont magyarázatra szorulhat, az az, hogy mi is voltaképpen az az euid és az egid, azaz a tényleges (effektív) felhasználó-, illetve csoportazonosító. Ez az azonosító az, amelyik ténylegesen számít egy feladat elvégzésekor. Gondoljunk a `SETUID`-os programokra! Lényegtelen, hogy ki indítja el, a program akkor is megteheti mindazt, amit a tulajdonosa is megtehet. Ha tehát egy futtatható állomány a rendszergazda tulajdonában van, és rendelkezik a `SETUID` bittel, akkor a tényleges jogosultsága meg fog egyezni a rendszergazdai jogosultságokkal. (Ha a tényleges jogosultságok fogalma egy kicsit homályos, ne bánkódjunk, a következő részben a fájlrendszer kapcsán sokat fogunk vele foglalkozni.)
Az utolsó fogalom, amire szükség lesz, a tgrp-id, más néven terminál-csoportazonosító. Ennek az értéke annak a folyamatnak a PID-jével egyezik meg, amelyik az adott folyamathoz tartozó terminál eszközfájlját legelőször megnyitotta. (ez az esetek többségében nem más, mint a jelszóparancsfájl).
Most, hogy már mindent elsoroltunk, vágjunk is bele a közepébe!

A jelek

A jelekre már sokszor hivatkoztunk, arról azonban, hogy pontosan mik ezek és mire is valók, aligha beszéltünk részletesen. Most változtatunk a helyzeten, és elmesélünk mindent, amit a jelekről tudni érdemes.
Sorozatunk harmadik részében, amikor az IPC-vel (a folyamatok közötti kapcsolattartással) foglalkoztunk, több olyan nehézséget is bemutattunk, amelyeknek a feloldásához elengedhetetlen, hogy két folyamat egymással „beszélni” tudjon. Ezekben a példákban az volt a közös, hogy amikor egy folyamat már nem tudott tovább dolgozni (például az étkező filozófusok esetében nem állt rendelkezésre mind a két villa), akkor a folyamat blokkolt, és egészen addig várt, amíg valaki fel nem ébresztette.
Az előző mondat kulcsszava a „várt”. A gond ott van, hogy jó lenne, ha egy folyamattal akkor is tudnánk adatot közölni, amikor dolgozik, azaz éppen semmiféle bemenetre nem számít. Erre találták ki a jeleket – segítségükkel bármelyik időpillanatban jelezhetünk bizonyos dolgokat az éppen futó folyamatoknak, tekintet nélkül arra, hogy az éppen min munkálkodik. (Az már más kérdés, hogy az adott folyamat mit csinál a beérkező jelzéssel, az is előfordulhat, hogy figyelmen kívül hagyja.)
A jelek csak első hallásra tűnhetnek valamiféle titokzatos dolognak. Valójában teljesen hétköznapi dolgok a rendszer életében, sőt minden bizonnyal mindenki küldött már jelet egy folyamatnak. Gondoljunk csak arra, mit csinálunk, ha egy alkalmazás lefagyott. Vannak rendszerek, amikor a Reset gomb megnyomása az egyetlen kiút, de a Linux nem tartozik közéjük. Elegendő, ha egy `SIGTERM`, súlyosabb esetekben egy `SIGKILL` nevű jelet küldünk az adott folyamatnak, ami ennek hatására az fejvesztve menekül a memóriából. (A Unixban a jelek küldésére a `kill` utasítás szolgál. Sokan úgy gondolják, hogy ezt a parancsot egy folyamat megsemmisítésére találták ki, pedig eredeti feladata a jelküldés. Az más kérdés, hogy

szinte minden esetben a SIGTERM és a SIGKILL jel elküldésére használják, de bármi mást is továbbíthatunk a segítségével. A jelekről listát a `kill -l` utasítással kaphatunk.)
Hogy egy alkalmazás mit tesz a beérkező jelekkel, az kizárólag a programozón múlik. Mindenesetre a jelek kezelése meglehetősen kényelmes feladat: csupán annyi a feladatunk, hogy a program indulásakor a SIGACTION nevű rendszerhívás segítségével átadjuk annak az eljárásnak a címét, amelyik majd a jel kezeléséért felelős lesz. Például a SIGTERM jelhez is rendelhetünk egy ilyen eljárást, ami nem csinál mást, mint lezárja az összes megnyitott állományt, ezután visszaadja a folyamat által foglalt memóriát. Ennek köszönhetően, ha véletlenül végtelen ciklusba kerül a folyamat, a SIGTERM jel hatására ki fog lépni.
Ugyanennek a rendszerhívásnak a segítségével lehetőségünk nyílik a beérkező jelek figyelmen kívül hagyására is. Ilyenkor hiába küldözgetünk lázasan jeleket, semmi sem fog történni.
A jeleket blokkolhatjuk is, ez a folyamat annyiban különbözik az előzőtől, hogy a beérkező jelek tárolva lesznek, és a folyamat csak a blokkolás megszüntetése után kapja meg őket.
Ha nem készítünk ilyen eljárást, akkor az adott folyamat figyelmem kívül fogja hagyni a beérkező jeleket, még a SIGTERM-et is. (Pontosabban, ha egy folyamat a SIGACTION rendszerhívás segítségével nem rendel jelkezelő eljárást egy jelhez, akkor az adott jel beérkezésekor a memóriakezelő egy alapértelmezett tevékenységet fog elvégezni. Hogy ez pontosan mi is, az az adott jeltől függ, de általában két dolgot jelenthet: vagy nem történik semmi, vagy kilövi a folyamatot.)
Létezik azonban egy jel, amit egyetlen folyamat sem vehet semmibe: ez pedig a SIGKILL. Ezt a jelet ugyanis nem a folyamatnak kell kezelnie, hanem a memóriakezelőnek. A folyamat nem is tud róla, hogy neki SIGKILL-t küldtek, és esélye sincs védekezni ellene. Ilyen jelre nyilvánvalóan szükség van, mert enélkül lehetetlen lenne megszabadulni az olyan lefagyott alkalmazásoktól, amik nem válaszolnak a SIGTERM-re. Viszont érdemes a beragadt programok ellen bevethető legeslegutolsó „fegyvernek” meghagyni, ugyanis ilyenkor nincs lehetősége a memóriában lévő adatokat a lemezre írni, míg a SIGTERM esetében ezt még megteheti.

A jelek és a memóriakezelő kapcsolata

Most már tudjuk, mi történik akkor, ha egy folyamathoz jel érkezik. Az azonban még mindig homályos folt számunkra, hogy miként juttatja őket a rendszer célba.
A jelek alapvetően kétféleképpen keletkezhetnek: vagy egy folyamat által, a `kill` rendszerhívás segítségével (ez az eset megegyezik azzal, amikor a felhasználó kiadja a `kill` utasítást), vagy maga a rendszermag hívja életre őket. A rendszer számára lényegtelen, miképpen is keletkezett az adott jel, a célbajuttatás módja minden esetben megegyezik.
Először is meg kell állapítanunk, hogy ki kinek küldte az adott jelet, és van-e hozzá jogosultsága. A jogosultság ellenőrzésekor az az alapelv, hogy egy rendszergazdai folyamat mindenkinek küldhet jelet, viszont mindenki más csak azoknak, akiknek a felhasználói azonosítói megegyeznek az sajátjáéval. Például minden általunk elindított alkalmazásnak küldhetünk SIGKILL-t, de más felhasználó folyamatait ilyen módon nem tudjuk kiiktatni. A rendszergazdára ilyen megkötés nem vonatkozik, ő bárkinek a folyamatát leállíthatja. A dolog azért nem ilyen egyszerű: a memóriakezelőnek olyasmire is figyelnie kell, hogy a címzett véletlenül nem zombifolyamat-e (mert ilyennek nincs értelme jelet küldeni).
Ezután ellenőrizni kell, hogy a címzett folyamat az adott jelet figyelembe veszi-e, illetve érvényben van-e a jelek blokkolása. Ha érvényben van, akkor fel kell jegyezni, hogy milyen jelet kapott, és – amint a blokkolás megszűnik – ismételten el kell küldenie a memóriakezelőnek (természetesen a SIGTERM jelre a blokkolás nem vonatkozik).
Ha a blokkolás érvénytelen, akkor két eset lehetséges: a folyamat vagy foglalkozik az adott jellel, vagy figyelmen kívül hagyja. Először nézzük meg az előbbi esetet! Ilyenkor a

memóriakezelőnek ki kell számítania, hogy a címzett folyamat pontosan hol is helyezkedik el a memóriában, továbbá meg kell hívnia a jelkezelő eljárást. Ez azonban nem egyszerű művelet, ugyanis előbb mentenie kell a folyamat pillanatnyi állapotát (erre azért van szükség, hogy a folyamat ugyanonnan folytathassa munkáját, ahol a jel beérkezése előtt tartott). Erre a folyamat vermét szokás használni. Ezzel az a gond, hogy korántsem biztos, hogy a veremben elegendő szabad hely van az összes adat mentésére, ezért még mielőtt bármi érdemlegeset cselekednénk, ellenőrizni kell, hogy valóban elfér-e minden a veremben. Ha nincs elég szabad hely, akkor baj van, mert a rendszer nem tehet mást, minthogy kilövi a folyamatot (szerencsére ma már elég ritkán fordul elő, hogy a folyamat verme megtelik). A folyamat állapotának mentését a memóriakezelő nem tudja elvégezni, ezért az alacsonyabb szinten lévő rendszermaghoz fordul. A jelkezelő eljárás csak ezután kerül meghívásra. Miután a folyamat a szükséges teendőket elvégezte, meghívja a SIGRETURN rendszerhívást, aminek hatására a veremből visszaállítódik a régi állapot (ezt is a mag végzi).
Ha viszont a folyamat nem rendelkezik jelkezelő eljárással, akkor a memóriakezelő az adott jelhez tartozó „alapértelmezett tevékenységet” fogja elvégezni. Ez két dolgot jelenthet: vagy nem fog történni semmi (azaz a memóriakezelő figyelmen kívül hagyja a jelet), vagy kilövi a folyamatot. Az utóbbi esetben mindig készül egy úgynevezett core dump állomány, ami az adott folyamat memóriatérképét tartalmazza (ehhez a fájlrendszer közreműködése is szükséges).

Amikor egy folyamat elindul…

A Unixban kétféleképpen keletkezhet új folyamat. Létrejöhet úgy, hogy egy – már létező – folyamat szétágazik, azaz a `fork` rendszerhívás segítségével gyermekfolyamatot hoz létre. De megszülethet olyan módon is, hogy egy folyamat egy másik programot indít el maga helyett. Erre az `exec` nevű rendszerhívás alkalmas. A két rendszerhívás tehát teljesen másra használható és másképpen is működik, azonban van bennük két közös dolog: mindkettő új folyamatot

1. lista Példa egy gyermekfolyamat létrehozására

```
result = fork(); // gyermekfolyamat
lØtrehozÆsa
if (result &lt; 0)  // ha a visszatØrØsi
                   // ØrtØk negat v ...
{
  // hiba zenetet
} else {
  if (result == 0)
  {
     // ezt fogja csinÆlni a
     // gyermekfolyamat
  }
else
  {
    // ezt pedig a sz lı
}
```

hoz létre és memóriafoglalást hajt végre. A továbbiakban mind a kettővel megismerkedünk.

Gyermekfolyamatok

Egy folyamatnak tehát a `FORK` rendszerhívás segítségével gyermekfolyamatok létrehozására nyílik lehetősége. A gyermekfolyamat csaknem pontos másolata lesz a szülőfolyamatnak: ugyanannyi memóriát foglal el és ugyanazt a kódot is használja. A gyermek azonban egy különálló folyamat, azaz a szülővel (és a többi folyamattal) párhuzamosan fog futni.
Hogy megértsük, mit is jelent ez a gyakorlatban, vessük egy pillantást az *1. listára*, ahol egy gyermekfolyamat létrehozására láthatunk példát. Azok kedvéért, akik nem jártasak a C programozási nyelvben, röviden elmeséljük, mi történik. Először is meghívjuk a `fork` rendszerhívást (pontosabban a `fork()` könyvtári eljárást, ami majd intézkedik, hogy a `fork` rendszerhívás végrehajtódjon). A művelet végeredménye egy szám lesz (amit a `result` nevű változóban tárolunk). Ha az értéke negatív, akkor nem jártunk sikerrel, például azért, mert nem volt elég szabad memória. Ilyenkor nincs más teendőnk, mint egy hibaüzenetet kiíratni és kilépni. Ha azonban a `result` értéke 0 vagy nagyobb, akkor a gyermekfolyamat megszületett. Ne felejtsük el, hogy a gyermek és a szülő kódja ugyanaz, tehát innentől kezdve a gyermek is ezt a kódot hajtja végre! A különbség csupán annyi, hogy a gyermek esetében a változó értéke 0, míg a szülőnél a létrejött gyermek folyamatazonosítóját (PID-jét) tartalmazza. Ilyen módon lehetőség nyílik annak eldöntésére, hogy az adott kódot most a szülő- vagy a gyermekfolyamat hajtja-e végre.
Meg kell jegyeznünk, hogy a `fork` rendszerhívást „gonosz” dolgokra is fel lehet használni, például arra, hogy egy folyamat nagyon sok példányban előállíthassa magát, leterhelve ezáltal a processzort és „felzabálva” a szabad memóriát. Az ilyen programokat forkbombáknak nevezzük. Hatékonyan védekezni ellenük csak az erőforrások használatának korlátozásával lehet, de a jelenlegi Linux-terjesztések szerencsére már alapból védettek az efféle támadások ellen.
A gyermekfolyamat nemcsak a szülő kódját, hanem a jogosultságait is örökli. Ha nagyon pontosak szeretnénk lenni, akkor ezt valahogy úgy lehetne megfogalmazni, hogy a gyermekfolyamatok öröklik a szülőt futtató felhasználó- és csoportazonosítót, továbbá a tényleges felhasználó- és csoportazonosítókat is.

2. lista
Példa az exec rendszerhívás végrehajtására az execle() könyvtári függvény segítségével. Ez a sor a /bin/ls program meghívását eredményezi – ugyanaz, mintha a parancsfájlban kiadtuk volna az „ls -l claudia.jpg” utasítást.

```
r=execle( /bin/ls","ls","-l",
➥"claudia.jpg",(char *)0, envp);

Az elsı ØrtØk a binÆris ÆllomÆny nevØt
tartalmazza. A k vetkezı ØrtØk magÆnak a
parancsnak a neve (ez egyØbkØnt elhagyhat ),
a harmadik Øs a negyedik ØrtØkkel a
vØgrehajtand  parancs kapcsol it adhatjuk
meg. Mivel t bb ØrtØket is meg lehet adni,
sz ksØg van rÆ, hogy jelezni tudjuk, mikor
Ør vØget a felsorolÆs. Erre szolgÆl a
(char *)0. Az utols  ØrtØk az œj programnak
Ætadand  parancsfÆjlvÆltoz kra mutat.
```

3. lista Példa az exec és a fork rendszerhívások ötvözésére

```
int result;
if (fork() == 0)
  { execle("/bin/ls","ls","-l",(char
*)0,envp); }
 else {
    wait(&result); }
```

A jogosultságokon kívül a munkakönyvtárban és a jelkezelő eljárásokban sincs különbség a gyermek és a szülő között.
A gyermek ugyanakkor az ütemező számára független folyamat, így egyedi folyamatazonosítóval (PID-del) kell rendelkeznie. Akad még pár dolog, amiből a gyermekfolyamat sajáttal bír, ilyenek például a fájlleírók, de erről majd a fájlrendszer tárgyalásakor beszélünk részletesen.

Az exec rendszerhívás

A `fork` rendszerhívást abban az esetben használhatjuk, ha a gyermeknek valamilyen a szülőfolyamathoz hasonló feladatot kell ellátnia. A gyakorlatban azonban ilyesmi viszonylag ritkán fordul elő. A helyzet az, hogy legtöbbször valamiféle a szülőfolyamattól teljesen független dolgot kell tennie.
Sőt van olyan eset is, amikor csak a folyamat futása közben dől el, hogy pontosan mi is lesz a feladata. (Gondoljunk a parancsértelmezőre, amiben egy – a felhasználó által megadott – állományban lévő kódot kell végrehajtatni.) Ilyenkor mindig az `exec` rendszerhívást kell alkalmazni, ami mind közül a legbonyolultabb.
(A `fork` és az `exec` közötti különbséget a legkönnyebben úgy érthetjük meg, hogy a `fork`-ot úgy képzeljük el, mint a saját programunk szétágaztatását, azaz a különböző részfeladatokat egymással párhuzamosan, külön folyamatként végeztetjük el. Az `exec` esetében viszont egy, a miénktől teljesen független programot hívunk meg. Fontos, hogy az ilyenkor nem a mi folyamatunkkal párhuzamosan, hanem ahelyett fut. A mi programunk csak akkor folytatódik, ha az `exec`-kel indított alkalmazás már lefutott.)
Az `exec` rendszerhívásnak nemcsak a megvalósítása nehéz,

hanem a meghívása is egy kissé bonyodalmas. Ráadásul nemcsak egy, hanem többféle könyvtári eljárás is létezik az `exec` rendszerhívás végrehajtására. Leggyakrabban az `execle()`-t szokás használni, aminek használatára – érdekesség gyanánt – a *2. listán* láthatunk példát.
Az esetek többségében a `fork`-ot és az `exec` rendszerhívást a *3. listán* látható módon egyszerre szokás használni. Az elv nagyon egyszerű: a `fork`-kal egy gyermekfolyamatot hozunk létre, ami nem tesz mást, minthogy az `exec`-kel elindítja a megadott programot. Az ilyen módon indított alkalmazás a szülőfolyamattal párhuzamosan képes futni. Ez az alapja a Unix-parancsértelmezőknek (például sh, bash) is.
Felmerülhet a kérdés, hogy miért nincs olyan rendszerhívás, ami ilyen formában egyesíti a `fork` és az `exec` tulajdonságait. A válasz az, hogy felesleges lenne. Ezzel csak bonyolítanánk az életünket, mivel gondoskodnunk kellene a szabványos ki- és bemenet (az az eszköz, ahová a program az eredményt kiírja, illetve ahonnan az adatokat bekéri) oda-visszairányításáról. Ehelyett a létrejött gyermekfolyamatnak csak le kell zárnia a pillanatnyi be- és kimenetet, majd egy új megnyitása után meghívni az `exec`-et (a gyermekfolyamatok ugyanis az átirányított ki- és bemenetet is öröklik).
Az `exec` rendszerhívás megvalósítása jóval bonyolultabb a `fork`-énál. Az utóbbinál „mindössze” annyi a feladat, hogy a folyamattáblában létre kell hozni egy új bejegyzést, majd a szülő adatait be kell másolni a gyermekhez. Az `exec` esetében azonban már egy teljesen új memóriatérképet kell létrehozni, továbbá a fájlrendszerrel is együtt kell működni, mivel mégis-csak a lemezről olvassuk be a futtatni kívánt kódot.
Amikor elindítunk egy programot (például a parancssorban kiadjuk az `ls` utasítást, ami a */bin/ls* program meghívását eredményezi), akkor mind a memóriakezelőnek, mind a fájlrendszernek rögtön ellenőrzést kell végrehajtania. Először is meg kell állapítaniuk, hogy egyáltalán van-e jogosultságunk futtatni az adott állományt (ez a fájlrendszer feladata), továbbá utána kell járniuk, hogy van-e elég szabad memória (ezt pedig a memóriakezelő intézi). Ha ezek közül egy is negatív eredménnyel jár, a program indítása meghiúsul.
Azt, hogy pontosan mennyi memóriára is van szükségünk, a bináris állomány fejlécéből olvashatjuk ki, ami a szakaszok, illetve az egész program méretét tartalmazza. Ezután következhet a memóriafoglalás. A hívó program memóriáját akár fel is szabadíthatjuk. A következő lépés a kód betöltése és a tényleges jogosultságok beállítása (ellenőrizni kell a `SETUID` és `SETGID` biteket). A folyamattáblába való beírás csak ezt követően mehet végbe. Utolsó lépésként a rendszermagot meg kell kérnünk a folyamat futtatására.

Élőhalottak a memóriában?

Minden folyamat számára eljön egyszer a vég. Ez a vég beköszönhet egyrészt egy jel képében, amikor is a program felszólítást kap a memóriából való távozásra. Másrészt úgy is vége szakadhat létezésének, ha egyszerűen az összes feladatát elvégezte. Ha egy gyermekfolyamat befejezte munkáját, akkor meg kell hívnia az `exit` nevű rendszerhívást. Ennek a rendszerhívásnak egy értéket is át kell adnia, mégpedig egy 0 és 255 közé eső számot. Ez az úgynevezett kilépési helyzet (exit-status), ami arra szolgál, hogy a szülőfolyamat értesülhessen róla, hogy a gyermeke milyen eredménnyel végezte el feladatát. (Ha például a gyermek 0-s státusszal tér vissza, akkor sikeresen, ha valami mással, akkor sikertelenül.)
A gyermekfolyamat végleges megszűnéséhez azonban egy második lépésre is szükség van, mégpedig arra, hogy a szülő meghívja a `wait` rendszerhívást. Ez arra szolgál, hogy a szülő várakozhasson arra, hogy a gyermeke meghívja az EXIT-et, és utána visszakaphassa annak kilépési helyzetét.
Amíg azonban a szülő nem hívja meg a `wait`-et (de a gyermek már végrehajtotta az `exit` rendszerhívást), addig a gyermekfolyamat úgynevezett zombiállapotba kerül. Ez amolyan átmenet az „élet és a halál” között, azaz már visszaadta a számára lefoglalt memóriaterületet és ütemezésre sem kerül, viszont a folyamattáblában továbbra is jelen van. Ez az állapot általában csak rövid ideig tart. Ha azonban valamilyen okból kifolyólag a szülőfolyamat még azelőtt megszűnik, hogy kiadta volna a `wait`-et, a gyermekfolyamat sosem lesz képes kikerülni a zombiság csapdájából.
Hogy ne kísérthessenek mindenféle élőhalott folyamatok a folyamattáblában, egy szülőfolyamat megszüntetésekor bizonyos intézkedések megtételére van szükség – szülőt kell találnunk az árván maradt gyerekfolyamatok számára.
A legkézenfekvőbb „apajelölt” az `init` – ami az egész folyamatfa gyökerét képezi –, mivel ez a folyamat sosem fog megszűnni. Az `exec` rendszerhívást a gyermek hajtja végre, így a szülőfolyamat tovább dolgozhat. Igaz, ebben a példában a szülő csak annyit tesz, hogy vár a gyermekfolyamat befejeződésére.
(Az `init`-ről már volt korábban szó, amikor is a `boot` műveleteket tárgyaltuk. Ez egy olyan egyszerű program, amit maga a rendszermag indít el, miután betöltődött a memóriába és befűzte a gyökérlemezrészt. Az `init`-nek egyetlenegy feladata van: minden terminál számára elindítja a login nevű bejelentkező programot. Ezután blokkolt állapotba kerül, és így is marad, amíg magát a rendszert le nem állítjuk – pontosabban az `init` sosem hajt végre `exit` rendszerhívást, de azért időnként meghív egy `wait`-et, hogy az árván maradt zombifolyamatoktól megszabaduljunk.)
A következő részben utolsó nagy témakörünk kivesézését kezdjük meg, mégpedig a fájlrendszerét. Ez egy nagyon összetett alrendszer, ugyanis nemcsak hatékonyan és biztonságosan kell tárolnia a lemezen lévő adatokat, de a felhasználóhitelesítés kérdése is kulcsszerepet kap az esetében; mindemellett a memóriakezelővel is barátkoznia kell.

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.