Miskolci Egyetem . . . és Informatikai Kara

Végh János

Bevezetés a számítógépes rendszerekbe – programozóknak

Kivételes utasítás kezelés

Copyright © 2008-2015 (J.Vegh@uni-miskolc.hu)
V0.12@2015.03.12

Ez a segédlet a *Bevezetés a számítógépes rendszerekbe* tárgy tanulásához igyekszik segítséget nyújtani. Nagyrészt az irodalomjegyzékben felsorolt Bryant-O'Hallaron könyv részeinek fordítása, itt-ott kiegészítve és/vagy lerövidítve, néha más jó tankönyvek illeszkedő anyagaival kombinálva. A képzés elején, még a számítógépekkel való ismerkedés fázisában kerül sorra, amikor előbukkannak a különféle addig ismeretlen fogalmak, és megpróbál segíteni eligazodni azok között. Alapvetően a számítógépeket egyfajta rendszerként tekinti és olyan absztrakciókat vezet be, amelyek megkönnyítik a kezdeti megértést.

Ez az anyag még erőteljesen fejlesztés alatt van, akár hibákat, ismétléseket, következetlenségeket is tartalmazhat. Ha ilyet talál, jelezze a fenti címen. Az eredményes tanuláshoz szükség van az irodalomjegyzékben hivatkozott forrásokra, és az órai jegyzetekre, ottani magyarázatokra is.

Tartalomjegyzék i

1 Kivételes utasítás 3

1.1. Kivételek 9

1.1.1. Kivétel kezelés 12

1.1.2. A kivételek osztályozása 17
1.1.3. Kivételek a Linuxban 26
1.2. Folyamatok 34
1.2.1. Logikai vezérlési folyam 36
1.2.2. Konkurrens folyamok 39
1.2.3. Saját címtér 41
1.2.4. Felhasználói és felügyelői mód 43
1.2.5. Környezet átkapcsolás 45
1.3. A rendszerhívások hibakezelése 50
1.4. Folyamat vezérlés 53
1.4.1. Folyamat azonosítók megszerzése 54
1.4.2. Folyamatok létrehozása és lezárása 55
1.4.3. A gyermek folyamatok begyűjtése 62
1.4.4. A folyamatok altatása 64
1.4.5. Programok betöltése és futtatása 65
1.4.6. A **fork** és az **execve** használata program futtatásra 73
1.5. Jelzések 80

1.5.1. A jelzések terminológiája 83
1.5.2. Jelzések küldése 86
1.5.3. Jelzések fogadása 96
1.5.4. A jelzések kezelésének finomságai 101
1.5.5. Hordozható jelzés kezelés 113
1.5.6. A jelzések blokkolásának kezelése 116
1.5.7. Folyamatok szinkronizálása 118
1.6. Nem-lokális ugrások 122
1.7. Segédprogramok a folyamatok kezelésére 129
1.8. Összefoglalás 130

Táblázatok jegyzéke **133**

Ábrák jegyzéke **135**

AT YOUR SERVICE MA'AM!
JOE'S DRUG
ONE-MINUTE PHOTO
RING BELL

FEJEZET

1

Kivételes utasítás kezelés

Attól a pillanattól kezdve, amikor bekapcsoljuk, addig, amíg ki nem kapcsoljuk, a processzor programszámlálója az

$$a_0, a_1, \ldots, a_{n-1}$$

sorozaton lépked végig, ahol az egyes a_k értékek a megfelelő I_k utasítás címét jelentik. Az a_k-ról az a_{k+1}-re való átmenetet vezérlés átadásnak (control transfer) nevezik, az ilyen átmenetek sorozatát pedig a processzor vezérlési folyamának.

A legegyszerűbb vezérlési folyam a "sima" sorozat, amikor az I_k és I_{k+1} utasítások a memória egymás utáni rekeszeiben helyezkednek el. A folyam hirtelen megváltozásait, amikor I_{k+1} utasítás nem az I_k utasítás után következik, olyan ismerős utasítások hozzák létre, mint a jump, call és return. Az ilyen utasítások teszik lehetővé, hogy a programok a program változói által reprezentált belső programállapotokra reagáljanak.

A rendszereknek azonban olyan változásokra is reagálniuk kell, amelyeket nem tükröznek belső változók és amelyek nem feltétlenül kapcsolódnak a program folyamához. Például, egy hardver időzítés rendszeres időközönként lejár és azzal foglalkozni kell. Csomagok érkeznek a hálózati illesztőkártyáról, amelyeket a memóriában el kell tárolni. A programok adatokat kérnek a mágneslemezről, és csendben "alszanak" amíg meg nem kapják az értesítést, hogy készen van az adat. A szülő folyamatok gyermek folyamatokat hoznak létre, és értesítést kell kapniuk, amikor a gyermek folyamataik befejeződnek.

A modern rendszerek ezekre a helyzetekre úgy reagálnak, hogy hirtelen változást

hoznak létre a vezérlő folyamban. A vezérlő eme hirtelen változásait nevezzük kivételes utasítás végrehajtásnak (exceptional control flow, ECF). Egy számítógépes rendszerben minden szinten előfordulnak kivételkezelések. Például, hardver szinten a hardver által kiváltott események hatására a vezérlés a kivételkezelő rutinhoz kerül. Az operációs rendszer szintjén a kernel az egyik folyamattól egy másikhoz környezet átkapcsolással (context switch) viszi át a vezérlését. Az alkalmazások szintjén az egyik folyamat egy másiknak jelzést küldhet, ami hirtelen átviszi a vezérlést a fogadó folyamat jelkezelő rutinjába. Az egyes programok a hibákra a szokásos veremalapú eljárással, vagy más függvényekben tetszőleges helyre történő ún. nem-lokális ugrással reagálhatnak.

Programozóként több okból is fontos megérteni a kivételes utasítás végrehajtást (ECF):

- Fontos rendszer koncepciók megértéséhez nélkülözhetetlen. Az operációs rendszerben az ECF az alap mechanizmus az I/O műveletek, a folyamatok, és a virtuális memória megvalósításához. Azaz, mielőtt azokat megtanulnánk, meg kell érteni az ECF-et.
- Az ECF lehetővé teszi, hogy megértsük, hogyan lép kölcsönhatásba egy alkalmazás az operációs rendszerrel. Az alkalmazások az operációs rendszertől trap

vagy rendszerhívásként ismert ECF használatával kérnek szolgáltatást. Például, adat mágneslemezre írásáshoz, vagy adat beolvasásához a hálózatról, új folyamat létrehozásához, a futó folyamat befejezéséhez az alkalmazói programok mind rendszerhívást használnak. A rendszerhívások mechanizmusának megértése segít abban, hogy megértsük, hogy az alkalmazások milyen módon használják ezeket a szolgáltatásokat.

- Az ECF megértése segít abban, hogy érdekes új alkalmazásokat tudjunk írni. Az operációs rendszer nagyon hatékony mechanizmusokat biztosít az alkalmazásoknak, hogy új folyamatokat hozzanak létre, megvárják folyamatok befejeződését, más folyamatoknak értesítést küldjenek a rendszer rendkívüli eseményeiről, észrevegyék ezeket az eseményeket és válaszoljanak azokra. Ha megértjük ezeket az ECF mechanizmusokat, tudunk jó Unix parancsértelmezőket és WEB kiszolgálókat írni.
- Az ECF megértése segít megérteni a konkurrens végrehajtás fogalmát. A számítógépes rendszerekben az ECF az alapmechanizmus a konkurrencia megvalósítására. Egy kivétel kezelő, amelyik megszakítja egy alkalmazás, folyamat vagy szál végrehajtását, amelynek végrehajtása időben átfed azokkal; egy jelkezelő,

amelyik megszakítja egy alkalmazói program végrehajtását, mind jó példák a konkurrens végrehajtásra. A konkurrens végrehajtás megértéshez az első lépés az ECF megértése.

- Az ECF segít megérteni, hogyan működik a szoftveres kivétel kezelés. Az olyan nyelvek, mint a C++ és Java, szoftveres kivételkezelést biztosítanak a **try**, **catch** és **throw** utasításokkal. A szoftveres kivétel kezelés teszi lehetővé a programoknak, hogy nem-lokális ugrásokat (amelyek megsértik a szokásos hívás/visszatérés elvét) hajtsanak végre, a hiba előfordulására válaszként. A nem-lokális ugrások egyfajta alkalmazás-szintű ECF, és azt C nyelvben a **setjmp** és **longjmp** függvények valósítják meg. Ezeknek az alacsony szintű függvényeknek a megértésével közelebb jutunk ahhoz, hogy a magas szintű nyelvek kivétel kezelése miként valósítható meg.

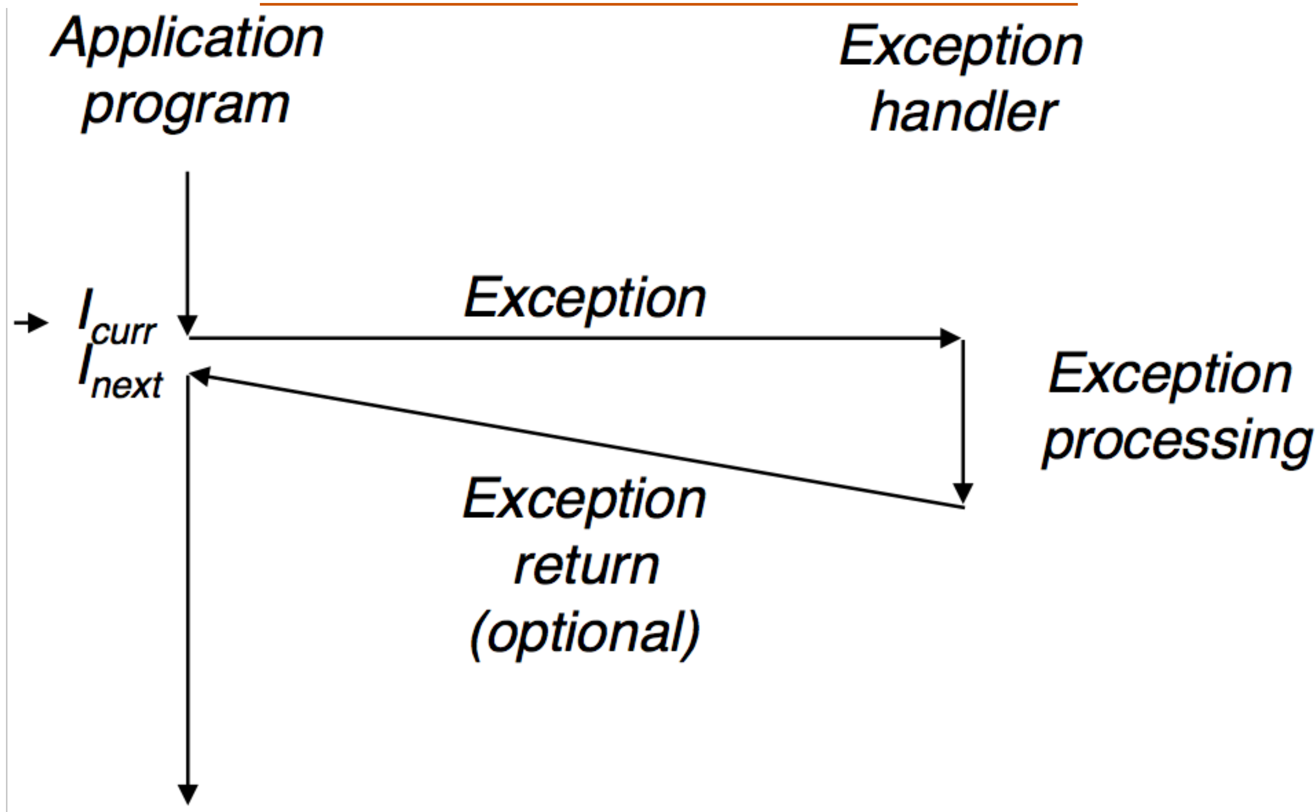
Eddig azt tanultuk meg, hogyan lépnek kölcsönhatásba az alkalmazások a hardverrel. Ez a fejezet alapvető olyan szempontból, hogy ezzel kezdjük el megtanulni, hogyan lép kölcsönhatásba alkalmazásunk az operációs rendszerrel. Érdekes módon, ezek a kölcsönhatások mind az ECF körül forognak. Megtanuljuk az ECF különböző fajtáit, amelyek a számítógépes rendszerek különböző szintjein előfordulnak. Olyan ECF lesz

az első, amelyik a hardver és az operációs rendszer érintkezési pontján található. Tárgyaljuk a rendszerhívásokat is olyan kivételekként, amelyek az alkalmazások számára belépési lehetőséget biztosítanak az operációs rendszerbe. Ezután egy absztrakciós szinttel feljebb lépünk és folyamatokat és a jelzéseket tárgyaljuk, amelyek az alkalmazások és az operációs rendszer érintkezési pontjánál találhatók. Végül tárgyaljuk a nem-lokális ugrásokat, amelyek az ECF alkalmazás-szintű formája.

1.1. Kivételek

A kivételek a kivételes vezérlési folyam olyan formája, amelyet *részben hardver, részben az operációs rendszer* valósít meg. Mivel részben hardver valósítja meg, a részletek rendszerről rendszerre változnak. Az alapötletek azonban ugyanazok, minden rendszer esetén. Ebben a szakaszban általánosságban megértjük a kivételt és annak kezelését, és megpróbáljuk demisztifikálni ezt a modern számítógép rendszerekben gyakran zavarosan használt fogalmat.

A kivételes utasítás végrehajtás a vezérlési folyam olyan hirtelen megváltozása, ami a processzor állapotának megváltozása következtében jön létre. Az 1.1 ábrán látható módon, a processzor éppen az I_{curr} utasítást hajtja végre, amikor a processzor állapotában jelentős változás következik be. Az állapot változása a processzoron belül bitekre és jelzésekre képeződik le. Az állapotban bekövetkező változást eseménynek nevezik. Az esemény lehet közvetlen kapcsolatban az éppen végrehajtott utasítással. Például, amikor a virtuális memória használatakor nincs a keresett lap a memóriában, aritmetikai túlcsordulás történik, vagy az utasítás megpróbál nullával osztani. Másrészt, az esemény lehet független az aktuális utasítás végrehajtásától. Például, lejár egy



1.1. ábra. Egy kivétel anatómiája. A processzor egy állapotváltozása (esemény) egy hirtelen vezérlésátadást (egy kivételt) vált ki az alkalmazástól a kivétel kezelőhöz. Befejeződése után a kezelő a vezérlést vagy visszaadja a megszakított programnak vagy abortál.

időzítés vagy befejeződik egy I/O művelet.

A fenti esetek mindegyikében, a processzor észleli, hogy egy esemény történt, végrehajt egy közvetett eljárás hívást (ez a kivétel kezelés), egy kivételkezelési táblázatnak nevezett címtáblázat segítségével. Az ugrás a címtáblázatban megadott helyre, az operációs rendszer egy kifejezetten ilyen esemény kezelésére tervezett szubrutinjára (a kivétel kezelő eljárás) történik.

Amikor a kivétel kezelő eljárás befejeződik, a következők valamelyike történik, az esemény típusától függően:

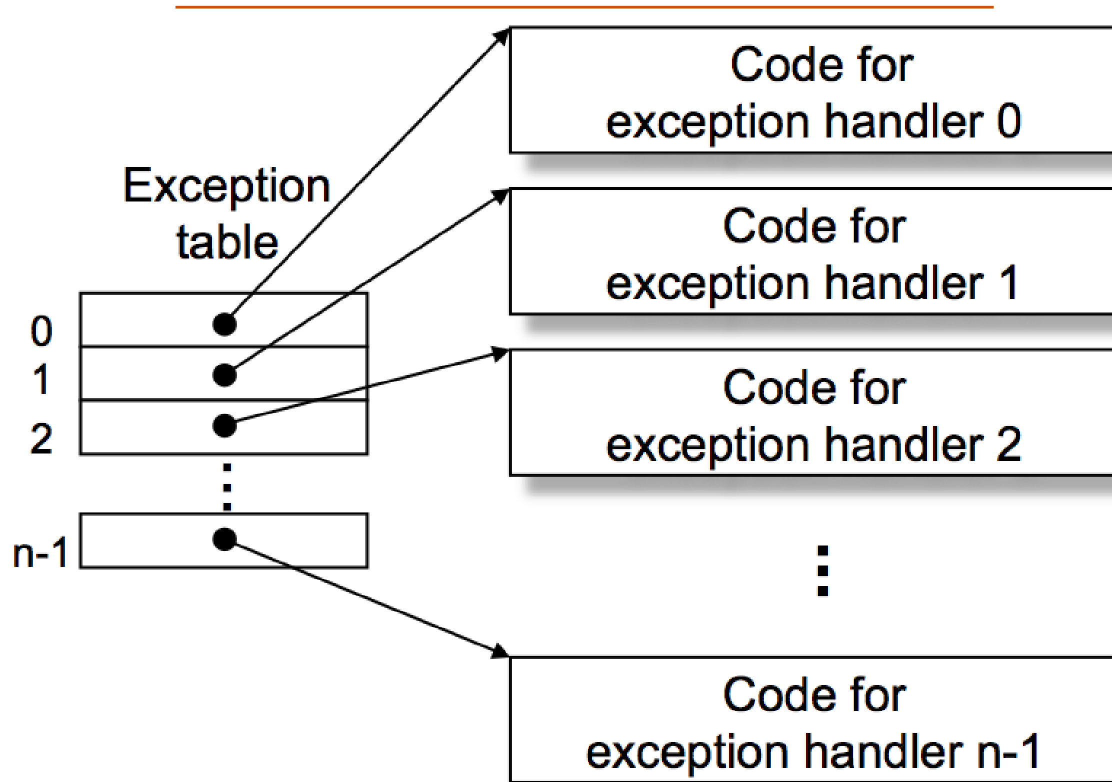
- A kezelő a vezérlést az aktuális I_{curr} utasításnak adja vissza; annak, amelyik éppen akkor hajtódott végre, amikor az esemény történt.
- A kezelő a vezérlést az I_{next} utasításnak adja vissza; annak, amelyik akkor hajtódott volna végre, ha nem történik meg a kivételes végrehajtás
- A kezelő abortálja a megszakított programot

1.1.1. Kivétel kezelés

A kivétel kezelés mechanizmusát nem egyszerű megérteni, mivel hardver és szoftver szoros együttműködését igényli. Könnyen belezavarodhatunk, hogy melyik komponens milyen feladatot végez. Lássuk tehát a munkamegosztás részleteit.

A rendszerben a lehetséges kivétel forrásokhoz hozzárendelnek egy egyedi nemnegatív egész számot, a kivétel számát. Ezen számok egy részének hozzárendelését a processzor tervezői végzik. A másik rész az operációs rendszer kernel tervezőire marad. Az első típusra példa a nullával való osztás, a laphiba, memória hozzáférés sértés, töréspont, aritmetikai túlcsordulás. Az utóbbiak kategóriájába esnek a rendszerhívások és a küldő I/O eszközök jelzései.

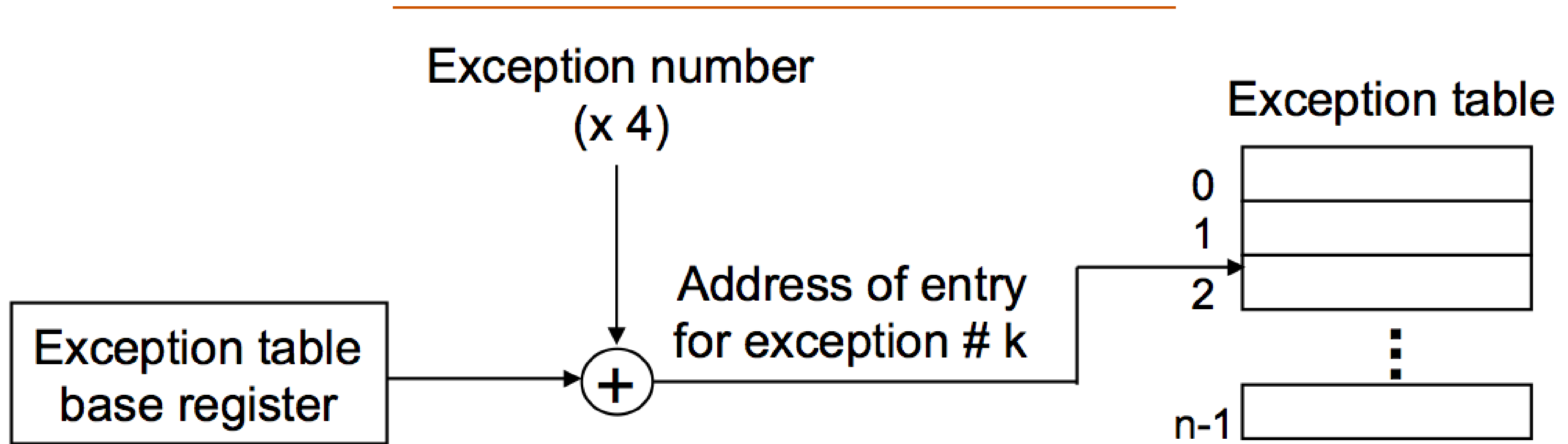
Futási időben (amikor a rendszer valamilyen programot hajt végre) a processzor észreveszi, hogy valamilyen esemény történt és meghatározza a megfelelő k eseményszámot. Ezután processzor kezdeményezi az esemény kezelését egy indirekt eljáráshívással, az eseménykezelési táblázat k-adik elemén keresztül, a megfelelő kezelő eljáráshoz. Az 1.3 ábra mutatja, hogyan használja a processzor az eseménykezelési táblázatot az eseménykezelő címének előállítására. A kivétel száma index a kivétel



1.2. ábra. A kivétel kezelő táblázat: egy ugrási táblázat, amelyben a k elem tartalmazza a k kivétel kezelőjének címét.

©[**BryantHallaron**] 2014

kezelő táblázathoz, aminek a kezdőcímét egy erre szolgáló CPU regiszter, a kivételkezelő táblázat alapcím regisztere tartalmazza.



1.3. ábra. A kivétel kezelő eljárás címének előállítása. A kivétel száma indexeli a kivétel kezelő táblázatot.

©[**BryantHallaron**] 2014

A kivételkezelés nagyon hasonlít egy eljárás hívásra, de vannak fontos eltérések is.

- Az eljárás híváshoz hasonlóan, a processzor elhelyezi a veremtárolóban a visszatérési címet, mielőtt a handlernek adná a vezérlést. A kivétel osztályától függ azonban, hogy a visszatérési cím az esemény bekövetkezésekor éppen végrehajtott utasítás vagy a következő utasítás, amit akkor hajtott volna végre, ha nem következik be az esemény.
- A processzor további processzor állapot részleteket is a verembe ír, amelyek majd a megszakított program újraindításához lesznek szükségesek, miután a megszakítás kezelő visszaadja a vezérlést. Például egy IA32 rendszer, többek között, a pillanatnyi állapotjelzőket tartalmazó EFLAGS regisztert is elhelyezi a veremben.
- Ha a vezérlés a felhasználói programból a kernelnek adódik át, ezek az adatok nem a felhasználói, hanem a kernel veremtárolójába kerülnek.
- A kivétel kezelők kernel módban futnak, ami azt jelenti, hogy teljes mértékben hozzáférnek a rendszer erőforrásaihoz.

Miután a hardver kiváltotta a kivételt, a munka további részét a kivétel kezelő szoftver végzi. Miután a kezelő feldolgozta az eseményt, esetlegesen visszatér a megszakított programhoz egy speciális "visszatérés megszakításból" utasítás végrehajtásával, ami

a veremből visszaállítja a processzor megfelelő adat és vezérlő regisztereit, továbbá visszaállítja a felhasználói módot, ha a kivétel felhasználói programot szakított meg, majd visszatér a megszakított programhoz.

1.1.2. A kivételek osztályozása

A kivételeket négy osztályba sorolhatjuk: megszakítások, csapdák, hibák és abortálások (interrupts, traps, faults, and aborts). Az 1.1 táblázat foglalja össze ezen osztályok attribútumait.

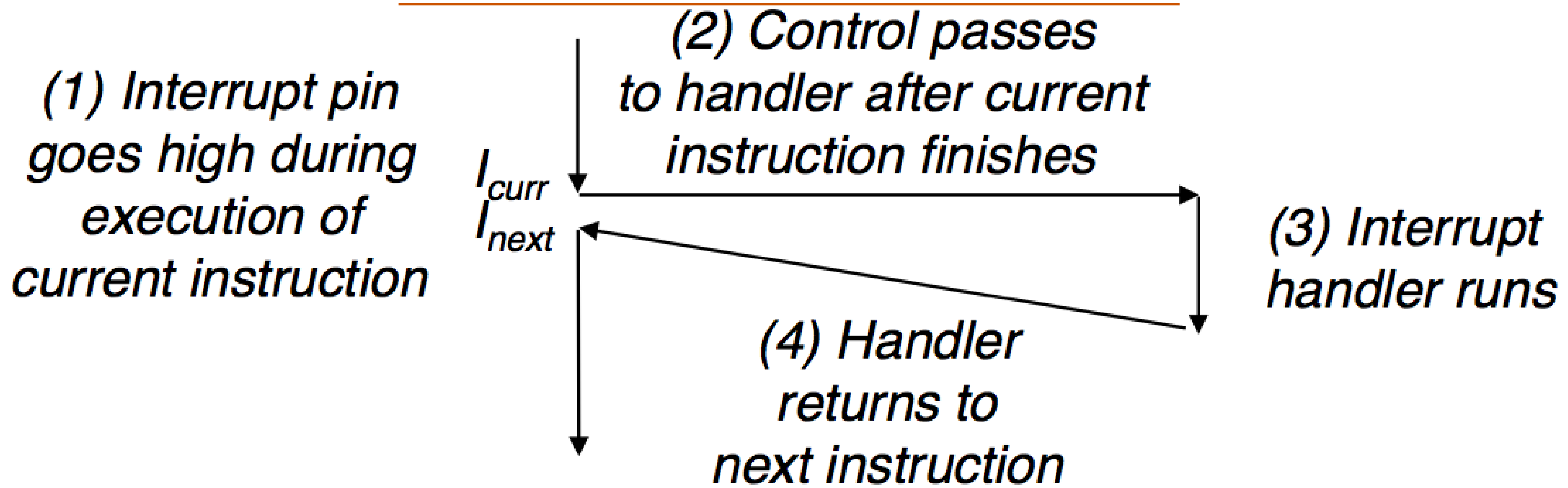
Megszakítások

A megszakítások aszinkron módon történnek, a processzoron kívül eső I/O eszközök jelzése következtében. A hardver megszakítások abban az értelemben aszinkron jellegűek, hogy nem valamely utasítás végrehajtásának eredményeként állnak elő. A hardver megszakítások kivétel kezelőjét megszakítás kezelőnek is hívják.
Az 1.4 ábra foglalja össsze a megszakítás feldolgozását. Az I/O eszközök, mint pl. a hálózati illesztőkártya, mágneslemez vezérlő, időzítő áramkörök a megszakítást a processzor valamelyik lábán jelzik, és a megszakítást okozó eszköz azonosító számát a rendszerbuszra helyezik.
Amikor az éppen végrehajtott utasítás befejeződik, a processzor észreveszi, hogy megszakítás tüske jelszintje magasra változott, beolvassa a kivétel számát a rendszer

1.1. táblázat. A kivételek osztályozása. Az aszinkron kivételek a processzoron kívül eső, I/O eszközök által előállított események következtében jönnek létre. A szinkron események egy utasítás végrehajtásának közvetlen következményei

Osztály	**Ok**
Megszakítás	Jelzés az I/O eszköztől
Csapda	Szándékos kivétel
Hiba	Esetleg kijavítható hiba
Abortálás	Nem javítható hiba

[**BryantHallaron**] 2014



1.4. ábra. Megszakítás kezelés. A megszakítás kezelő a vezérlést az alkalmazói program folyam következő utasítására adja vissza.

©[**BryantHallaron**] 2014

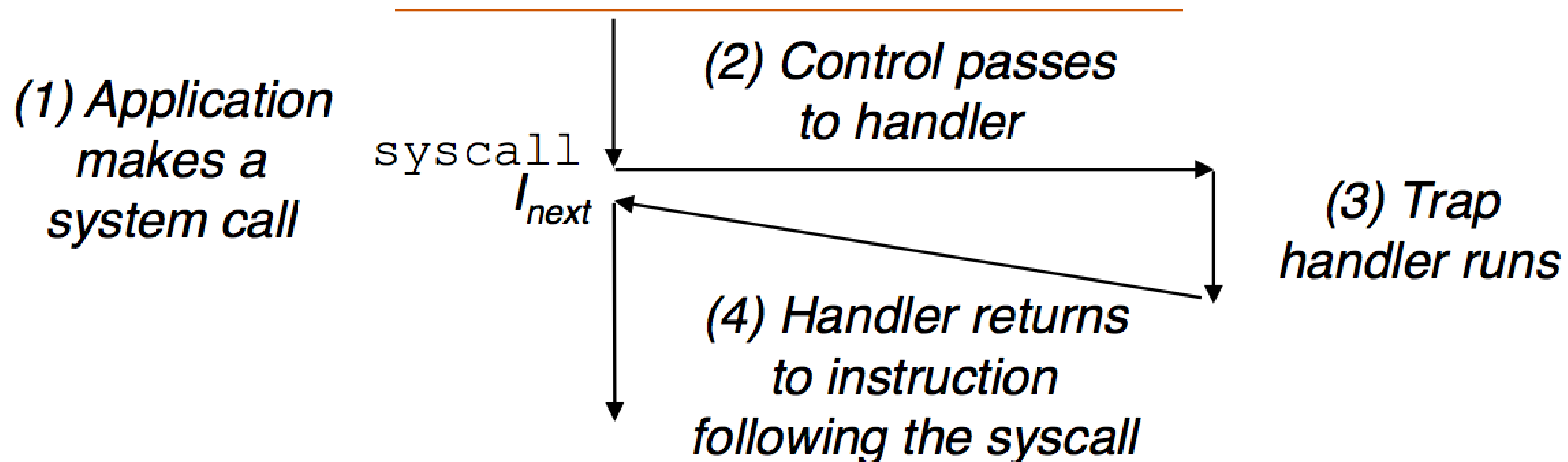
buszról, majd meghívja a megfelelő kezelő programot. Amikor a kezelő program visszatér, a vezérlést visszaadja a következő utasításnak (azaz, annak az utasításnak, amelyik a vezérlési folyam szerint akkor következett volna, ha nem történt volna

megszakítás). Ennek az a hatása, hogy a program úgy folytatódik, mintha nem is történt volna megszakítás. A kivétel többi osztálya (traps, faults, and aborts) az éppen végrehajtott utasítással szinkronban, annak eredményeként áll elő.

Csapdák és rendszerhívások

A csapdák olyan szándékosan előidézett kivételek, amelyek egy utasítás végrehajtásának eredményeként fordulnak elő. A megszakítás kezelőkhöz hasonlóan, a csapda kezelők is a következő utasításra adják vissza a vezérlést. A csapdák legismertebb felhasználása a rendszerhívások, amelyek egy eljárás hívás szerű interfészt biztosítanak a felhasználói program és a kernel között.

A felhasználói programoknak gyakran van szükségük arra, hogy a kerneltől olyan szolgáltatásokat kérjenek, mint fájl olvasás (read), új folyamat létrehozása (fork), egy új program betöltése (execve), vagy éppen az aktuális folyamat befejezése (exit). Hogy ilyen kernel szolgáltatáshoz ellenőrzött körülmények között lehessen hozzáférni, a processzorok biztosítanak egy "syscall n" utasítást, amelyet a felhasználói programok akkor hajtanak végre, amikor az *n* szolgáltatás végrehajtását akarják kérni. A **syscall**



1.5. ábra. Csapda kezelés. A csapda kezelő a vezérlést az alkalmazói programfolyam következő utasítására adja vissza.

©[**BryantHallaron**] 2014

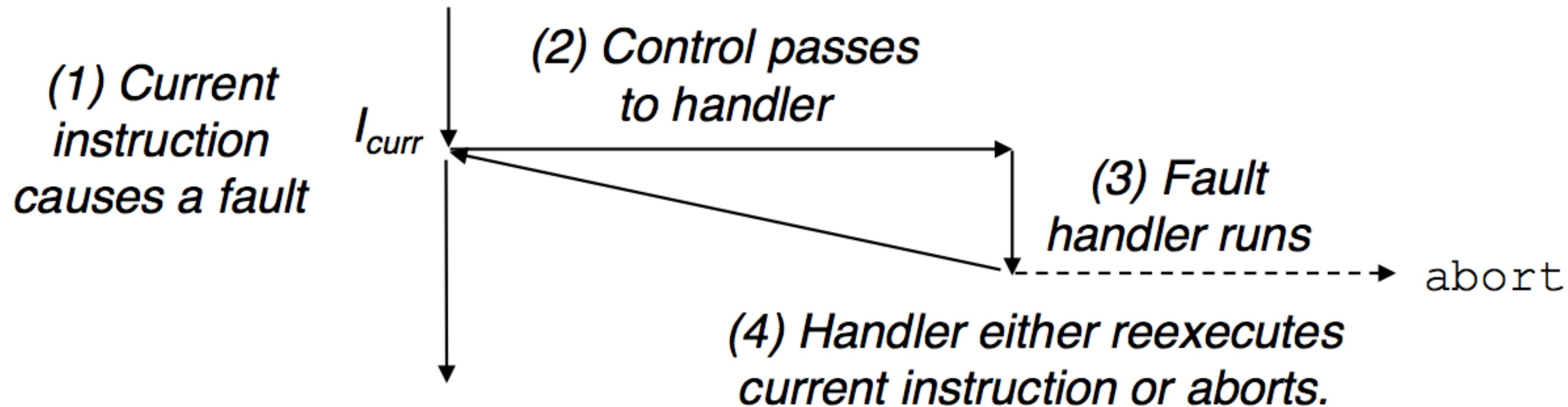
utasítás végrehajtása esetén a program vezérlés egy csapdán keresztül egy kivétel kezelőhöz kerül, amelyik dekódolja az argumentumot, és meghívja a megfelelő kernel rutint. A csapda kezelésének menetét a 1.5 ábra foglalja össze.

A programozó szempontjából a rendszerhívás egy közönséges eljáráshívással egyenértékű. Ezek megvalósítása azonban nagyon különböző. A reguláris függvények felhasználói módban futnak, ami a bennük végrehajtható utasításokat olyanokra korlátozza, amelyeket ilyen módban végre lehet hajtani, és ugyanazt a veremtárolót használják, mint a hívó függvény. A rendszerhívás viszont kernel módban fut, ami lehetővé teszi, hogy csak a kernelben végrehajtható utasításokat is használjon, és az ottani adatszerkezetekkel dolgozzon.

Hibák

A hibák olyan hibafeltételből származnak, amelyeket a kezelő esetleg korrigálni tud. Amikor egy hiba történik, a processzor átadja a vezérlést a hiba kezelőnek. Ha a kezelő ki tudja javítani a hiba okát, a vezérlést a hibát okozó utasításra adja vissza, azaz újra végrehajtja azt. Különben a kezelő egy abortáló rutinhoz "tér vissza" a kernelen belül, ami befejezi a hibát okozó alkalmazói programot. Az 1.6 ábra foglalja össze a hibakezelést.

A hiba egyik klasszikus példája a laphiba, ami akkor történik, ha egy utasítás olyan



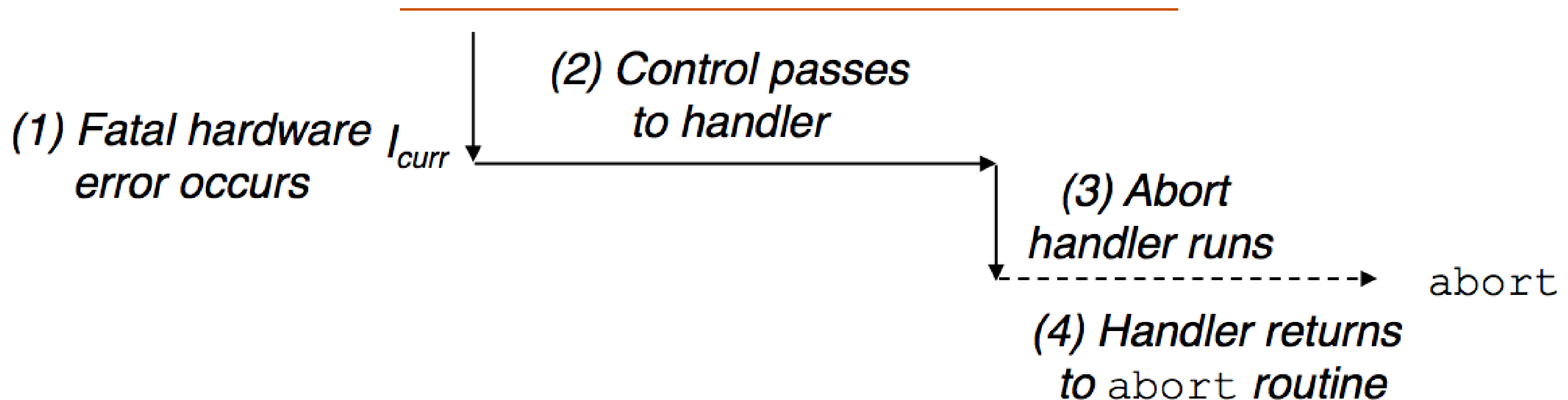
1.6. ábra. Hiba kezelés. Attól függően, hogy a hiba javítható-e vagy nem, a hiba kezelő vagy újból végrehajtja a hibás utasítást, vagy abortál.

©[**BryantHallaron**] 2014

virtuális címre hivatkozik, amelynek megfelelő fizikai cím pillanatnyilag nincs a memóriában, és ezért a mágneslemezről kell azt elővenni. Egy lap a virtuális memória egy folytonos blokkja (tipikusan 4K méretű). A laphiba kezelő betölti a megfelelő blokkot a mágneslemezről és a vezérlést arra az utasításra adja vissza, amelyik a hibát

okozta. Amikor az utasítás ismételten végrehajtódik, a megfelelő fizikai lap már a memóriában van és az utasítás hiba nélkül lefut.

Abortálások



1.7. ábra. Abortálás kezelés. Az abortálás kezelő a vezérlést a kernel abort rutinjába viszi át és befejezi az alkalmazói programot.

©[**BryantHallaron**] 2014

Az abortálás valamely elháríthatatlan fatális hibából ered, tipikusan olyan hardver hibából, mint a paritáshiba, ami akkor fordul elő, ha valamelyik DRAM vagy SRAM bit hibás. Az abortálás kezelő soha nem adja vissza a vezérlést az alkalmazói programnak. Mint azt az 1.7 ábra mutatja, a kezelő a vezérlést az **abort** rutinnak adja át, ami befejezi az alkalmazói programot.

1.1.3. Kivételek a Linux/IA32 rendszerben

Hogy a dolgokat kézzel foghatóbbá tegyük, tekintsünk néhány, az IA32 rendszerekre definiált kivételt. Összesen 256 féle típusú kivétel lehetséges. A 0 és 31 közötti tartományba eső sorszámú kivételeket az Intel mérnökei definiálták, ezért azok valamennyi IA32 rendszerben azonosak. A 32 és 255 közötti tartományban eső számok az operációs rendszer által definiált megszakításoknak és csapdáknak felelnek meg, lásd 1.2 táblázat.

Linux/IA32 rendszer hívások

Az alkalmazói programok számára a Linux százával kínál rendszerhívásokat. Ezeket az alkalmazások akkor használják, amikor szolgáltatásokat kérnek a kerneltől, azaz amikor fájlokat írnak/olvasnak, vagy új folyamatokat hoznak létre. Az 1.3 néhány népszerű Linux hívásra mutat példát. Az egyes rendszerhívásokhoz egy egész számot rendelünk, ami a kernelben egy ugrótáblázat eltolási értéke.

Az IA32 rendszereken a rendszerhívásokat egy `int` n formájú csapda utasítás valósítja meg, ahol n a 256 elemű kivétel kezelő táblázat valamelyik elemének indexe. Történeti

Megjegyzés: Linux/IA32 hibák és abortálások

- Osztási hiba. Osztási hiba (0. kivétel) akkor történik, amikor egy alkalmazás megpróbál nullával osztani, vagy amikor az osztás eredménye túl nagy a cél operandus számára. A Unix nem próbálja meg kijavítani az osztási hibát. A Linux az osztási hibát jellemzően “Floating exception” módon jelenti be.
- Általános védelmi hiba. A közismert általános védelmi hiba (13. kivétel) sokféle okból fordulhat elő, általában azért mert a program a virtuális memória nem-definiált területére hivatkozik, vagy mert a program írni próbál egy csak olvasható kódterületre. A Linux nem próbálja kijavítani ezt a hibát, általában “Segmentation fault” módon jelenti be.
- Laphiba. A laphiba (14. kivétel) olyan kivételre példa, ahol a hibázó utasítást újra végrehajtjuk. A kezelő leképezi a fizikai memória mágneslemezen levő megfelelő lapját leképezi a virtuális memóriára és újraindítja a hibás utasítást.
- Géphiba. A géphiba (18. kivétel) egy fatális hardver hiba következtében áll elő, és a hibázó utasítás végrehajtása alatt vesszük észre. A kezelő soha nem tér vissza az alkalmazói programhoz.

1.2. táblázat. Példák kivételekre IA32 rendszerekben

Kivétel szám	Leírás	Kivétel osztály
0	Osztás nullával	Hiba
13	Általános védelmi hiba	Hiba
14	Laphiba	Hiba
18	Géphiba	Abortálás
32-127	OS-definiált kivétel	Megszakítás vagy csapda

1.3. táblázat. Néhány gyakrabban használt rendszerhívás Linux/IA32 rendszerekben. Forrás: /usr/include/sys/syscall.h.

Szán	Név	Leírás	Szán	Név	Leírás
1	exit	A folyamat befejezése	27	alarm	Jel időzítés beállítása
2	fork	Új folyamat létrehozása	29	pause	Folyamat felfüggesztése jel megérkezéséig
3	read	Fájl olvasása	37	kill	Jelzés küldése másik folyamatnak
4	write	Fájl írása	48	signal	Jel kezelő beállítása
5	open	Fájl megnyitása	63	dup2	Fájl leíró másolása
6	close	Fájl lezárása	64	getppi	Szülő folyamat ID elővétele
7	waitpid	Várakozás gyermek folyamat befejeződésére	65	getpgrp	Folyamat csoportkód elővétele
11	execv	Program betöltése és futtatása	67	sigacti	Hordozható jelkezelő beállítása
19	lseek	Fájlon belüli helyre mozgás	90	mmap	Memória lap leképezése fájlra
20	getpi	Folyamat ID elővétele	106	stat	Fájl információ lekérése

okokból a rendszerhívásokat a 128 (0x80) számú kivétellel valósítják meg. A C programok rendszerhívásokat közvetlenül a **syscall** függvény hívásával is elérhetnek, de ez a gyakorlatban csak ritkán szükséges. A sztenderd C könyvtár a legtöbb rendszerhíváshoz tartalmaz burkoló függvényt. A burkoló függvény becsomagolja az argumentumokat, a csapdán keresztül átadja a megfelelő rendszerhívás azonosító számot, majd visszaadja a visszatérési állapotot a hívó függvénynek. A továbbiakban a rendszerhívásokat és a megfelelő burkoló függvényeket közös néven rendszer-szintű függvényeknek nevezzük.

Érdemes szemügyre venni, hogyan használják a programok az **int** utasítást arra, hogy közvetlenül használják a Linux rendszerhívásokat. A Linux rendszerhívások valamennyi paraméterét általános célú regisztereken keresztül adják át, és nem a veremtárolón át. Hagyományosan az %eax regiszter tartalmazza a rendszerhívás azonosító számát, a legfeljebb hat paramétert pedig az %ebx, %ecx, %edx, %esi, %edi és %ebp regiszterek. Az %esp veremmutató azért nem használható, mert a kernel módba lépéskor felülíródik.

Példaként tekintsük a jól ismert **hello** (lásd 1.1 lista) programot, amit most rendszer-szintű függvényekkel írtunk meg.

A **write** első argumentuma 1 a kimenetet **stdout**-ra állítja. A második argumentum a

kiírandó bájtsorozat, a harmadik pedig a kiírandó bájtok számát adja meg.

Programlista 1.1: A "Hello Világ" program forráskódja rendszerhívásokkal

```
#include <stdio.h>

int main()
{
    write(1, "hello, world\n",13); (1)
    exit(0); (2)
}
```

Az 1.2 lista mutatja a **hello** assembly nyelvű változatát, amelyik közvetlenül használja az **int** utasítást a **write** és az **exit** rendszerhívások elérésére. (1) A program meghívja a **write** függvényt. Először, (2) beírja **write** a rendszerhívás kódját az %eax regiszterbe, majd az argumentum listát a többi regiszterbe. Ezután, (3) használja az **int** utasítást a rendszerhívásra. Hasonlóképpen, (4) hívja meg a **exit** rendszerfüggvényt.

Programlista 1.2: A "Hello Világ" rendszerhívásos változatának assembly nyelvű kódja

```
.section  .data
string:
  .ascii  "hello, world\n"
string_end:
  .eq len, string-end - string

.section  .text
.globl  main
main:
;Call write(1,"hello, world\n",13)
    movl  $4, %eax   ; System call number 4 (1)
    movl  $1, %ebx   ; stdout has descriptor 1 (2)
    movl  $string, %ecx; Hello world string
    movl  $len, %edx     ; String length
    int $0x80            ; System call code (3)
  ;Next, call exit(0)
    movl  $1, %eax       ;System call number 1
```

Megjegyzés: Terminológiai megjegyzés

A kivételek különböző osztályainak elnevezése rendszerről rendszerre változik. A processzor szerkezeti leírások gyakran különbséget tesznek aszinkron "megszakítás" és szinkron "kivétel" között, de nem használnak egy olyan közös elnevezést, amelyet ezekre a nagyon hasonló fogalmakra használhatnánk. Annak elkerülésére, hogy állandóan a "kivételek és megszakítások" kifejezést használjuk, a "kivételek" szót fogjuk használni általános elnevezésként és csak ott teszünk különbséget az aszinkron kivételek (megszakítások) és a szinkron kivételek (csapdák, hibák, és abortálások) között, ahol feltétlenül szükséges. Az alapvető ötletek minden rendszerben ugyanazok, de figyelnünk kell arra, hogy bizonyos gyártók kézikönyvei a "kivétel" szót kizárólagosan a szinkron eseményekkel kapcsolatban használják.

1.2. Folyamatok

A számítástudomány egyik legalapvetőbb és legsikeresebb fogalma a folyamat (process). Hogy azt az operációs rendszer biztosítani tudja, ahhoz a kivételek alapvető építőkőnek számítanak.

Amikor programot futtatunk egy modern operációs rendszeren, abban az illúzióban van részünk, hogy a rendszerben egyedül a mi programunk fut. Úgy tűnik, a mi programunk kizárólagosan használja a processzort és a memóriát. A processzor a mi programunk utasításait hajtja végre, egyiket a másik után, megszakítás nélkül. Végezetül, programunk kódja és adatai az egyetlen objektumok a rendszer memóriájában. Ezeket az illúziókat a folyamat fogalma biztosítja.

A folyamat klasszikus definíciója, hogy az *egy program egy példánya végrehajtás közben*. A rendszerben minden program valamely folyamat környezetében fut. A környezet lényegében azt az állapotot jelenti, amelyre a programnak szüksége van ahhoz, hogy helyesen működjön. Ez az állapot tartalmazza a program kódját és adatait a memóriában, a veremtárolóját, általános célú regisztereinek tartalmát, a programszámlálót, a környezeti változókat és a nyitott fájlok leíróit.

Valahányszor egy felhasználó a program nevének a parancs értelmezőbe írásával elindít egy programot, a parancs értelmező egy új folyamatot hoz létre és a végrehajtható objekt fájlt az új folyamat környzetében futtatja. Az alkalmazói programok is új folyamatokat hozhatnak létre és saját kódjukat vagy más alkalmazásokat az új folyamat környezetében futtathatnak.

Annak részletes tárgyalása, hogy az operációs rendszerek hogyan imlementálják a folyamatokat, jóval túlmutat a kurzus keretein. Tárgyaljuk viszont azt a két fontos absztrakciót, amelyet a folyamat biztosít az alkalmazás számára:

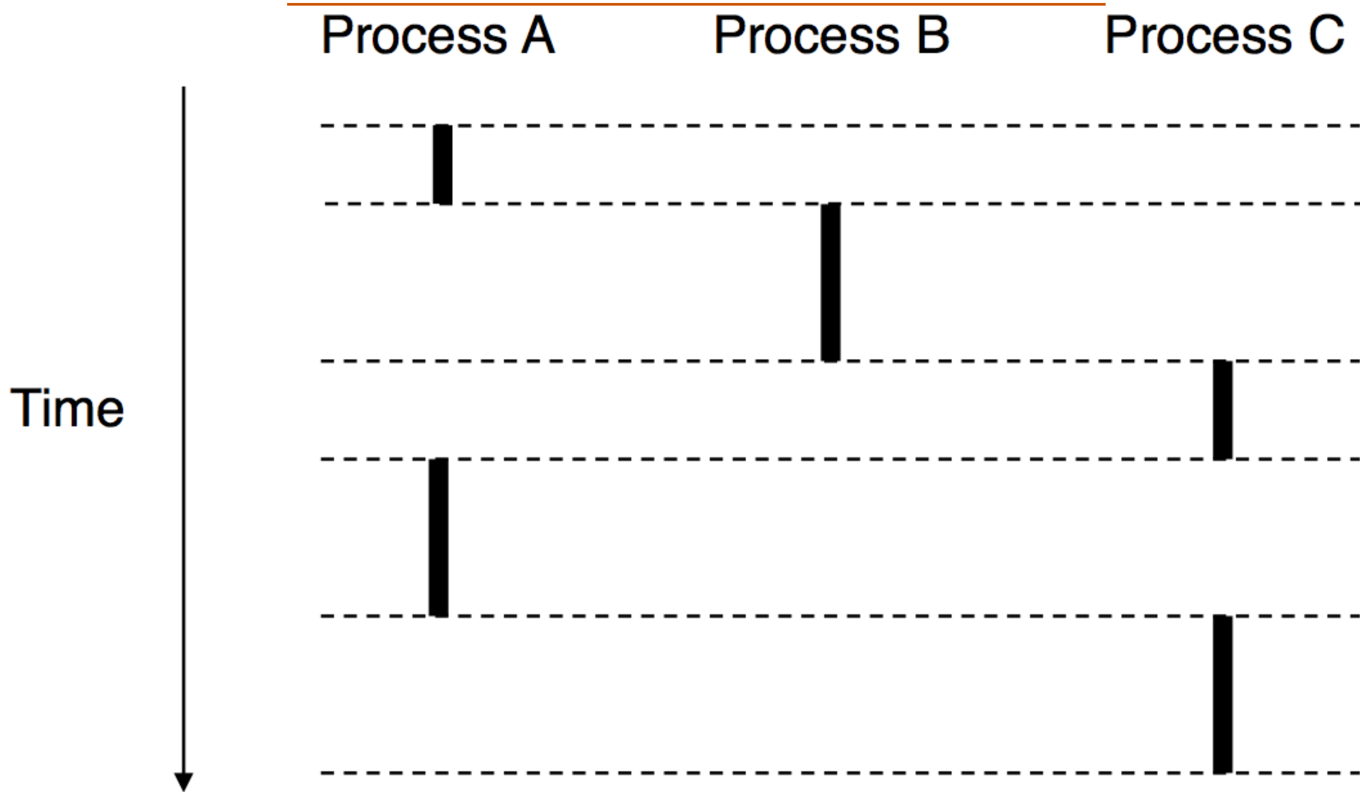
- Egy független *logikai vezérlési folyam*ot, ami bennünk azt az illúziót kelti, hogy a processzort kizárólagosan használjuk.
- Egy *privát címter*et, ami annak illúzióját kelti, hogy programunk kizárólagosan használja a memória rendszert.

1.2.1. Logikai vezérlési folyam

Egy folyamat azt az illúziót adja meg egy program számára, hogy a program kizárólagosan használja a processzort, még akkor is, ha valójában sok más program fut egyidejűleg a rendszerben. Ha egy debuggert használva egyes lépésekkel léptetve hajtjuk végre programunkat, kizárólag olyan programszámláló (PC) értékek sorozatát fogjuk látni, amelyek a programunkban található utasításoknak vagy a programunkhoz futási időben csatolt osztott objektumokban levő utasításoknak felelnek meg. Ez a PC érték sorozat a logikai vezérlési folyam, vagy logikai folyam.

Tekintsünk egy olyan rendszert, amelyik három folyamatot futtat, lásd 1.8 ábra. A processzor egyetlen fizikai vezérlési folyamatát három logikai folyamra osztjuk, mindegyik folyamathoz egyet rendelve. Az egyes függőleges vonalak a folyamathoz rendelt vezérlő folyam egy részét ábrázolják. Ebben a példában a három logikai folyam egymásba ékelődik. Egy ideig az A folyamat fut, ezután B, amelyik be is fejeződik. Majd a C folyamat fut egy ideig, amit A követ és befejeződik. Végül C is eljut a befejezésig.

Az ábra kulcs mondanivalója, hogy a folyamatok több menetben használják a processzort. Az egyes folyamatok a vezérlési folyam egy részét hajtják végre, aztán



1.8. ábra. Logikai vezérlési folyam. A folyamatok minden programnak biztosítják azt az illúziót, hogy kizárólagosan használja a processzort. Az egyes függőleges vonalak az egyes folyamatok logikai vezérlési folyamának egy részét ábrázolják.

©[**BryantHallaron**] 2014

kiszorítódnak (időlegesen felfüggesztődnek) arra az időre, amíg más folyamatok futnak. A valamely folyamat környezetében futó programnak úgy tűnik, kizárólagosan használja a processzort. Ennek az ellenkezőjére az lehetne a bizonyíték, ha pontosan mérnénk az egyes utasításoknál az eltelt időt. Ekkor azt látnánk, hogy a CPU programunk némelyik utasításánál egy időre megáll. Azonban, minden megállás után programunk újra elindul, mégpedig anélkül, hogy a program memória számlálójában vagy regisztereiben változás következne be.

1.2.2. Konkurrens folyamok

A logikai folyamok nagyon különböző formákat vehetnek fel a számítógépes rendszerekben. A kivétel kezelők, folyamatok jelzés kezelők, szálak, Java folyamatok mint példák a logikai folyamokra.

Azt a logikai folyamot, amelynek végrehajtása időben átfed másik folyammal, konkurrens folyamnak nevezzük, és azt mondjuk, hogy a két folyam konkurrens módon fut. Pontosabban, az X és Y egymással akkor és csak akkor konkurrensek, ha X akkor kezdődik, miután Y elkezdődött és mielőtt Y befejeződött, vagy Y akkor kezdődik, miután X elkezdődött és mielőtt X befejeződött, Például, az 1.8 ábrán A és B, valamint A és C konkurrens módon futnak Másrészt viszont B és C nem konkurrens módon futnak, mivel B utolsó utasítása befejeződik C első utasítása előtt.

Általánosságban konkurrens végrehajtásnak nevezzük, amikor több folyam konkurrens módon hajtódik végre. Azt a jelenséget, hogy egy folyamat más folyamatokkal felváltva hajtódik végre, multitaszkingnak is nevezik. Azokat az időszakaszokat, amikor egy folyamat végrehajtja folyamának egy részét, időszeletnek (time slice) nevezik. Emiatt a multitaszkingot időszeletelésnek is hívják. Például, az 1.8 ábrán az A folyamat két idő-

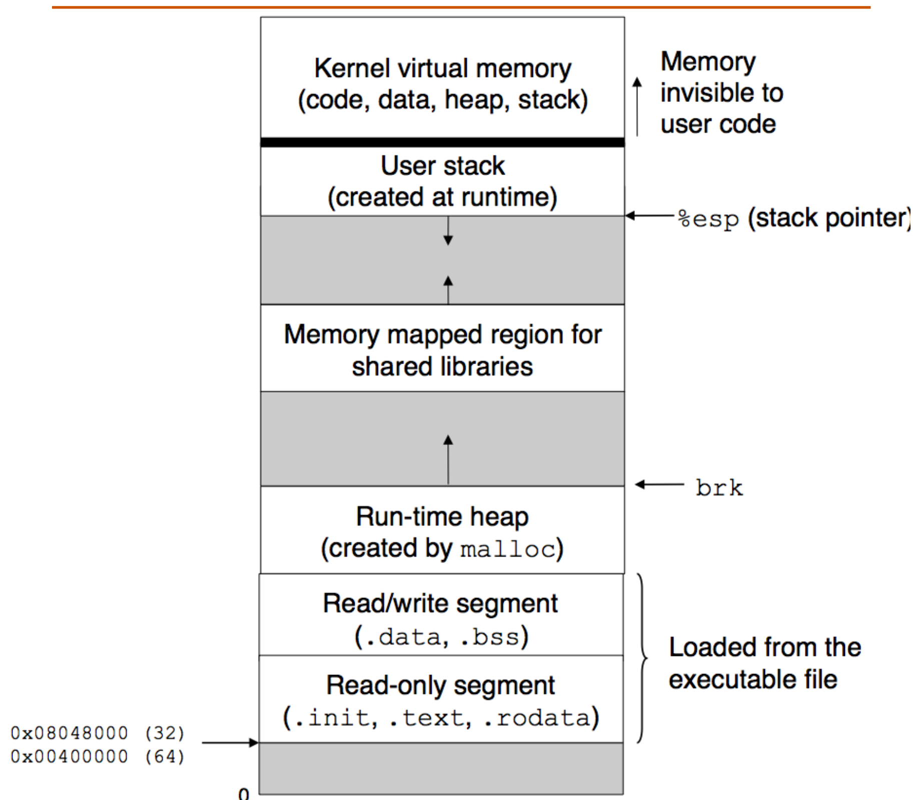
szeletből áll. Vegyük észre, hogy a konkurrens folyam független a processzor magjainak vagy a számítógépeknek a számától, amelyen vagy amelyeken a folyamok futnek. Ha két folyam időben átfed, azok konkurrensek, még ha ugyanazon a processzoron futnak is.

Néha azonban hasznos lesz megkülönböztetni a konkurrens folyamok egy alfaját, a párhuzamos folyamokat. Ha két folyam különböző magokon vagy számítógépeken fut konkurrens módon, akkor azt mondjuk, hogy azok párhuzamos folyamok, párhuzamosan futnak és párhuzamosan hajtódnak végre.

1.2.3. Saját címtér

Egy folyamat minden programot azzal az illúzióval lát el, hogy az a rendszer címterét kizárólagosan használja. Egy n bites címzést használó rendszerben a címtér 2^n lehetséges értékből (0, 1, ..., 2^n-1) áll. Egy folyamat minden egyes programnak saját címteret biztosít. Ez magán címtér abban az értelemben, hogy a címtér egy bizonyos címéhez tartozó memória bájt más folyamat által (általában) nem írható és olvasható.

Bár az egyes magán címterekhez rendelt memória tartalma általában különböző, az egyes terek általános szerveződése megegyezik. Például, az 1.9 ábra egy x86 Linux folyamat címterének szerveződését mutatja. A címtér alsó része a felhasználói program számára van fenntartva, a szokásos **text**, **data**, **heap** és **stack** szegmensekkel. A **code** segmensek a $0x08048000$ (32-bites folyamatok esetén) és a $0x00400000$ (64-bites folyamatok esetén) címen kezdődnek. A címtér felső része a kernel számára van fenntartva. A címtér eme része tartalmazza azokat a kódokat, adatokat, verem memóriát, amelyet a kernel akkor használ, amikor a folyamat "megbízásából" hajt végre utasításokat (azaz amikor az alkalmazói program rendszerhívást hajt végre).



1.9. ábra. A folyamat saját címtere

©[**BryantHallaron**] 2014

1.2.4. Felhasználói és felügyelői mód

Ahhoz, hogy az operációs rendszer biztosítani tudja a folyamat absztrakció kellően szigorú megvalósítását, a processzornak rendelkeznie kell valamilyen mechanizmussal, ami korlátozza egy alkalmazás által végrehajtható utasítások körét valamint az alkalmazás által hozzáférhető címtér részt.

A processzorok ezt a képességet tipikusan egy kontrol regiszter üzemmód bitjével biztosítják, ami azokat a jogokat adja meg, amivel a folyamat pillanatnyilag rendelkezik. Amikor ez a bit "1" értékű, a folyamat kernel (néha felügyelő vagy supervisor) módban fut. A kernel üzemmódban futó folyamat az utasításkészlet bármely utasítását végre tudja hajtani, és a rendszer memóriájának bármely részét el tudja érni. Amikor ez a bit "0" értékű, a folyamat felhasználói módban fut. Egy felhasználói módban futó folyamat számára nem nem engedélyezett privillégizált utasítások (pl. a processzor megállítása, az üzemmód bit megváltoztatása, vagy I/O művelet kezdeményezése) végrehajtása. Nem engedélyezett továbbá a címtér kernel területén levő kód vagy adat elérése sem. Az erre irányuló próbálkozás általános védelmi hibához vezet. Ehelyett a felhasználói programoknak a rendszerhívási interfészen keresztül közvetve lehet elérni a kernel kód

és adat részeit.

A felhasználói alkalmazást futtató folyamat kezdetben felhasználói módban van. A folyamat egyetlen lehetősége, hogy felhasználói módból felügyelői módba kerüljön, egy olyan kivétel, mint amilyen a megszakítás, hiba vagy csapda rendszerhívás. Amikor kivétel történik, és a vezérlés a kivétel kezelőhöz kerül, a processzor az üzemmódot felhasználóiból felügyelői módra váltja. A kezelő már felügyelői módban fut. Amikor a vezérlés visszatér az alkalmazói kódhoz, a processzor az üzemmódot felügyelői módból felhasználói módba kapcsolja vissza.

A linux alatt létezik egy ügyes mechanizmus, amit **/proc** fájlrendszernek neveznek, ami lehetővé teszi a felhasználói módban futó folyamatoknak, hogy a kernel adatszerkezeteihez hozzáférjenek. A **/proc** hierachikus szöveg formájában exportálja számos kernel adatszerkezet tartalmát, amit így a felhasználói porgramok olvasni tudnak. Például, a **/proc** fájlrendszer használatával kideríthetjük a CPU típusát **/proc/cpuinfo**, vagy egy bizonyos folyamat által használt memória szegmenseket **/proc/<process id>/maps**. A 2.6 Linux kerneltől kezdődően bevezettek egy **/sys** fájlrendszert is, amelyik további alacsony-szintű információt tartalmaz a rendszer buszokról és eszközökről.

1.2.5. Környezet átkapcsolás

A operációs rendszer kernel a multitaszking működést a környezet átkapcsolás (context switch) nevű magas szintű kivételes vezérlési folyammal valósítja meg. Ez a környezet átkapcsolás az előző szakaszban megismert alacsony szintű kivétel kezelő mechanizmusra épül.

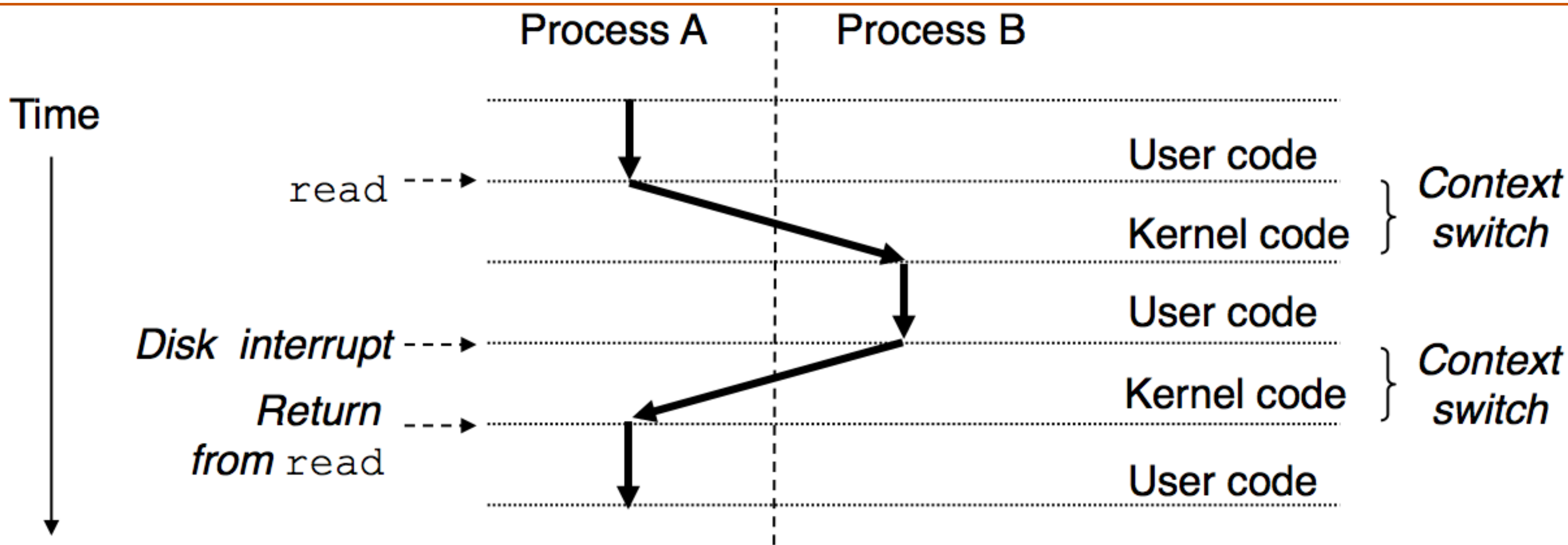
A kernel minden folyamat számára fenntart egy környezetet. A környezet az az állapot, amelybe a kernel eltárolta a kiszorított folyamatot. Olyan objektumoknak az értékeit tartalmazza, mind az általános célú regiszterek, a lebegőpontos regiszterek, a program számláló, a felhasználói verem, az állapotjelző regiszterek, a kernel verem memória, különféle verem adatszerkezetek, úgy mint a címteret leíró laptábla, az aktuális folyamatra vonatkozó információkat tartalmazó folyamat tábla, és a folyamat által megnyitott fájlokra vonatkozó információt tartalmazó fájl táblázat.

A folyamat végrehajtásának egy bizonyos pontján, a kernel dönthet úgy, hogy az éppen végrehajtás alatt levő folyamatot kiszorítja és egy korábban kiszorított folyamatot újraindít. Ezt a döntést hívják ütemezésnek (scheduling), amit a kernelnek az ütemező (scheduler) nevű része kezel. Amikor a kernel egy új folyamatot választ ki futtatásra, azt

mondjuk, hogy a kernel beütemezte a folyamatot. Miután a kernel egy új folyamatot ütemezett be futtatásra, a futó folyamatot kiszorítja és a környezet átkapcsolás (context switching) néven ismert mechanizmus használatával átadja a vezérlést az új folyamatnak. Ennek során

- elmenti a futó folyamat környezetét
- visszaállítja valamely előzőleg kiszorított folyamat környezetét
- átadja a vezérlést ennek az újonnan helyreállított folyamatnak.

Környezet átkapcsolás akkor történhet, amikor a kernel a felhasználó megbízásából rendszerhívást hajt végre. Ha a rendszerhívás blokkolódik, mivel valamely esemény megtörténésére várni kell, a kernel alvó módba helyezi a futó folyamatot és másik folyamatra kapcsol. Például, amikor egy **read** rendszerhívásnak mágneslemez hozzáférésre van szüksége, a kernel azt is választhatja, hogy átkapcsolja a környezetet és egy másik folyamatot futtat, amíg az adat megérkezik a mágneslemezről. Másik példa a **sleep** rendszerhívás, ami egy explicit kérés, hogy a rendszer a hívó folyamatot alvó módba tegye. Általában véve, még ha nem is blokkolódik egy rendszerhívás, a kernel dönthet úgy, hogy környezet átkapcsolást hajthat végre, ahelyett, hogy visszatérne a hívó folyamathoz.



1.10. ábra. Egy folyamat környezet átkapcsolásának anatómiája.

©[**BryantHallaron**] 2014

A környezet átkapcsolás létrejöhet megszakítás eredményeként is. Például, valamennyi rendszerben van valamilyen mechanizmus periodikus órajelek előállítására, tipikusan 1ms vagy 10 ms periódusidővel. Minden idő megszakítás kérés alkalmával,

a kernel dönthet úgy, hogy az aktuális folyamat már eleget futott, és új folyamatra kapcsol át. Az 1.10 ábra az A és B folyamatpár közötti környezet átkapcsolásra mutat példát. Ebben a példában kezdetben az A folyamat felhasználói módban fut, amíg egy rendszerhívás csapda következtében a kernelbe nem kerül a vezérlés. A kernelben a csapda kezelő egy DMA átvitelt kér a mégneslemez vezérlőtől és úgy állítja be a mágneslemezt, hogy az kérjen megszakítást, amikor az adatok átvitele a mágneslemezről a memóriába befejeződött.

A mágneslemez viszonylag sok időt (néhányszor tíz milliszekundum) használ el az adatok elővételére, így a tétlen várakozás helyett a kernel a A-ról a B folyamatra változtatja a környezetet. Megjegyezzük, hogy az átkapcsolás előtt a kernel felhasználói módban hajtott végre utasításokat az A folyamat megbízásából. Az átkapcsolás első részében a kernel az A folyamat megbízásából felügyelői módban hajt végre utasításokat. Ezután valamely ponttól kezdve utasításokat hajt végre (még mindig felügyelői módban) a B folyamat számára. Az átkapcsolás után a kernel felhasználói módban utasításokat hajt végre a B folyamat részére.

Ezután a B folyamat egy ideig felhasználói módban fut, amíg a mágneslemez megszakítást nem kér annak jelzésére, hogy az adatok átkerültek a mágneslemezről

a memóriába. A kernel úgy dönt, hogy a B folyamat már éppen eleget futott, és környezetet vált a B folyamatról az A-ra, ilyen módon visszaadva az A folyamatba a vezérlést, arra az utasításra, amelyik a **read** rendszerhívás után következik. Ezután az A folyamat fut a következő kivétel bekövetkeztéig, és így tovább.

Megjegyzés: Gyorsítótár szennyezés

Általánosságban, a hardveres gyorsítótár (cache) nem igazán jól tud együttműködni a kivételes utasításokat (megszakítások és kontexus váltások) végrehajtó folyamokkal. Ha az éppen futó folyamatot egy megszakítás rövid időre megszakítja, akkor a gyorsítótár "hideg" a megszakítást kiszolgáló rutin számára. Ha ez a megszakítás kezelő rutin a fő memóriából elég sok elemet használ, a megszakított fő folyamat folytatódásakor annak számára is hideg lesz a gyorsítótár. Ilyen esesetben mondjuk, hogy a megszakítás kiszolgáló rutin "elszennyezte" a gyorsítótárat. Hasonló tapasztalatokat szerezhetünk a környezet váltás során. Amikor környezetváltás után folytatódik egy folyamat, a gyorsítótár hideg az alkalmazás számára, és azt újból be kell melegíteni.

1.3. A rendszerhívások hibakezelése

Amikor egy Unix rendszer-szintű függvény hibát talál, általában −1 értéket ad vissza továbbá az **errno** globális változó értékét beállítja, jelzendő, hogy pontosan mi volt a probléma. A programozóknak *mindig* meg kell vizsgálnia, hogy történt-e hiba. Sajnos, sokan ezt nem teszik, mondván, hogy az "felfújja" a kódot és nehezen olvashatóvá teszi. Pédául, nézzük, hogyan vizsgálhatjuk meg a Linux **fork** függvényének használatakor, hogy fordult-e elő ilyen hiba, lásd 1.3 lista.

Programlista 1.3: A **fork** függvény hívása hibaellenőrzéssel

```
if ((pid = fork()) < 0) {
    fprintf(stderr, "fork error: %s\n", strerror(errno));
    exit(0);
}
```

Az **strerror** függvény egy karakter stringet ad vissza, ami a legjobban leírja az **errno** értékének megfelelő hibát. Ezt a kódot valamennyire egyszerűsíthetjük a **??** lista szerinti függvénnyel. Ezzel a függvénnyel már csak két soros lesz a **fork** függvény

hívásunk, lásd 1.4 lista. Tovább egyszerűsíthetjük kódunkat a hibakezeléshez egy

Programlista 1.4: A **fork** függvény hívása hibajelentő függvénnyel

```
if ((pid = fork()) < 0)
   unix_error("fork error");
```

ún. burkoló függvényt használva. Egy **foo** függvényhez definiálunk egy **Foo** függvényt, ugyanolyan argumentumokkal, de a név első betűjét nagybetűvel írva. A burkoló függvény meghívja az alap függvényt, megvizsgálja a hiba előfordulását, és befejezi a végrehajtást, ha hiba történt. Például, a **fork** függvény hibát is kezelő burkoló függvénye látható az 1.5 listán.

Ezt a burkolót használva, programunk egyetlen kompakt sorra zsugorodik, lásd 1.6 lista.

A továbbiakban ilyen hiba kezelő burkoló függvényeket használunk. Ennek az az előnye, hogy tömör lesz a bemutatott kód, de nem kelti azt a hamis illúzót, hogy a hibakezelést el lehet hanyagolni (amikor viszont a szövegben a rendszer-szintű függvények hibakezeléséről beszélünk, a kisbetűs névvel hivatkozunk rájuk, a megfelelő nagybetűs változat helyett).

Programlista 1.5: A **fork** hibajelentő burkoló függvénye

```
pid_t Fork(void)
{
   pid_t pid;

   if ((pid = fork()) < 0)
    unix_error("Fork error");
   return pid;
}
```

Programlista 1.6: A **fork** használata burkolófüggvénnyel és hibavizsgálattal

```
  pid = Fork();
```

1.4. Folyamat vezérlés

A Unix számos rendszerhívást biztosít arra, hogy C programokból folyamatokat manipuláljunk. Ebben a szakaszban a legfontosabb ilyen függvényeket ismerjük meg, használatukat példákkal illusztrálva.

1.4.1. Folyamat azonosítók megszerzése

Minden folyamatnak van egy (nem-negatív) egész folyamat azonosítója (process ID, PID). A **getpid** a hívó folyamat PID értékét adja vissza. A **getppid** a hívó folyamat szülő folyamatának PID értékét adja vissza (azaz, annak a folyamatnak a PID értékét, amelyik az adott folyamatot létrehozta). Lásd: 1.7 lista. A **getpid** és **getppid** rutinok egy **pid_t**

Programlista 1.7: A **GetPID** rendszerhívás prototípusa

```c
#include <sys/types.h>
#include <unistd.h>

pid_t getpid(void);
pid_t getppid(void);
  //Returns: PID of either the caller or the parent
```

típusú egész értéket adnak vissza, amelyeket a **types.h** definiál, Linux rendszeren **int** típusúként.

1.4.2. Folyamatok létrehozása és lezárása

Programlista 1.8: Az **exit** rendszerhívás prototípusa

```
#include <stdlib.h>

void exit(int status);
    // This function does not return
```

Az **exit** függvény a folyamatot véglegesen megállítja, a **status** kilépési állapottal, lásd 1.8 lista. (egy másik lehetőség a **status** beállítására egy egész értéket visszaadni a **main** függvényből való kilépéskor).

Egy szülő folyamat a **fork** függvény használatával hozhat létre egy gyermek folyamatot, lásd 1.9 lista. Az újonnan létrehozott gyermek folyamat ***csaknem*** azonos a szülő folyamattal. A szülő és az újonnan létrehozott gyermek folyamatok közötti legfontosabb különbség, hogy különböző PID-vel rendelkeznek.

A gyermek megkapja a szülő folyamat viruális címterének az eredetivel azonos (de az eredetitől független) másolatát, beleértve a **text**, **data** és **bss** szegmenseket, a heap

Megjegyzés: A folyamat állapotai

A programozó szempontjából úgy tekinthetjük, hogy egy folyamat az alábbi három állapot valamelyikében lehet:

- ***Running*** A folyamat éppen végrehajtódik a CPUn vagy arra vár, hogy végrehajtódjon, miután a kernel beütemezte.
- ***Stopped*** A folyamat végrehajtása felfüggesztődött és nem kerül ütemezésre. Egy folyamat annak következtében áll meg, hogy kap egy **SIGSTOP**, **SIGTSTP**, **SIGTTIN**, vagy **SIGTTOU** jelzést, és ilyen állapotban marad, amíg **SIGCONT** jelzést nem kap, ami után ismét futni kezd.
- ***Terminated*** A folyamat véglegesen megállt. Egy folyamat három okból fejeződhet be véglegesen:
  - A folyamat olyan jelzést kapott, aminek az alapértelmezett hatása a folyamat befejezése
  - A folyamat visszatért a **main** rutinból
  - A folyamat meghívta az **exit** függvényt

Programlista 1.9: A **fork** rendszerhívás prototípusa

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
  //Returns: 0 to child, PID of child to parent, -1 on
    error
```

és a verem memóriát is. A gyermek megkapja a szülőtől annak megnyitott fájljaihoz a leírókat, ami azt jelenti, hogy a gyermek minden olyan fájlt írni és olvasni tud, ami nyitva volt, amikor a szülő meghívta a **fork** függvényt. A **fork** függvény érdekes (és zavarba ejtő) tulajdonsága, hogy *egyszer* hívjuk és *kétszer* tér vissza; egyszer mint hívó (szülő) folyamat, és egyszer mint hívott (újonnan létrehozott gyermek) folyamat. A szülő folyamatban a **fork** visszatérési értéke a gyermek folyamat PID értéke. A gyermek folyamatban a visszatérési érték 0. Mivel a gyermek folyamat PID értéke mindig nullától különböző, a visszatérési érték egyértelműen azonosítja, hogy a program a

szülő vagy a gyermek folyamatban működik.

Az 1.10 lista egy olyan példát mutat be, amelyben a szülő folyamat a **fork** használatával egy gyermek folyamatot hoz létre. Amikor a **fork** visszatér, (1) **x** értéke mind a szülő, mind a gyermek folyamatban 1. A szülő folyamat (2) megnöveli és kinyomtatja saját **x** másolatát. Hasonlóképpen, a gyermek folyamat (3) csökkenti és kinyomtatja saját **x** másolatát, lásd 1.11 lista.

Amikor először tanulunk a **fork** függvényről, általában érdemes felrajzolni a folyamat gráfot, ahol a vízszintes nyilak olyan folyamatoknak felelnek meg, amelyek balról jobbra utasításokat hajtanak végre, a függőleges nyilak pedig a **fork** függvény hívásának felelnek meg, lásd 1.12 listán és a mellette levő ábrán.

Programlista 1.10: Új folyamat létrehozása a fork rendszerhívás használatával. A futtatás eredményét a 1.11 lista mutatja

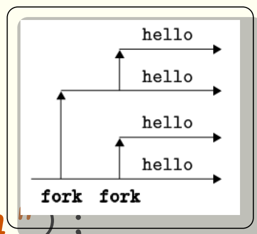
```
#include "csapp.h"

int main()
{
  pid_t pid;
  int x = 1;

  pid = Fork(); (1)
  if (pid == 0) {  /* Child */
    printf("child : x=%d\n", ++x); (3)
    exit(0);
   }

   /* Parent */
   printf("parent: x=%d\n", --x); (2)
   exit(0);
```

Programlista 1.11: A 1.10 listán látható program végrehajtásának eredménye

```
unix> ./fork
parent: x=0
child : x=2
```

Programlista 1.12: Kétszeresen elágazó fork függvény

```
#include "csapp.h"

int main()
{
  Fork();
  Fork();
   printf("hello\n");
   exit(0);
}
```



Megjegyzés: A fork hívás finomságai

- ***Egyszer hívjuk, kétszer jön*** A **fork** függvényt egyszer hívjuk a szülő folyamatban, de az kétszer tér vissza: egyszer a szülő folyamatban, egyszer az újonnan létrehozott gyermek folyamatban. Ez egyetlen gyermeket létrehozó folyamatban elég triviális. A **fork** több példányát is használó programokban azonban nehezen átlátható és gondos elemzést igényel.
- ***Konkurrens végrehajtás*** A szülő és a gyermek különálló folyamatok, amelyek konkurrens módon futnak. A két folyamat utasításait a kernel önkényesen egymásba ékelheti. Az egyik rendszeren a szülő folyamat befejezi a **printf** utasítást, csak ezután következik a gyermek folyamat. A másik rendszeren (vagy egy másik alkalommal) ennek az ellenkezője is igaz lehet. Általában véve: semmit sem tételezhetünk fel arról, hogyan fognak a különböző folyamatok utasításai egymásba ékelődni.
- ***Másolt, de különálló címterek*** Ha meg tudnánk állítani mind a szülő, mind a gyermek folyamatot közvetlenül azután, hogy a **fork** visszatért az egyes folyamatokba, azt látnánk, hogy a két folyamat címtere megegyezik. Mindkét folyamatban ugyanaz a helyi változók értéke, a heap, a globális változók és a kód. Így tehát példaprogramunkban az **x** helyi változó 1 értékű, amikor

1.4.3. A gyermek folyamatok begyűjtése

Amikor egy folyamat –akármilyen okból– véglegesen befejeződik, a kernel nem távolítja el azt a rendszerből azonnal. Helyette, a folyamat –terminált állapotban– megmarad, amíg a szülő folyamata be nem gyűjti. Amikor a szülő begyűjti a befejeződött gyermek folyamatot, a kernel elküldi a gyermek folyamat kilépési kódját a szülőnek, majd eltávolítja a befejeződött folyamatot, és ettől a ponttól kezdve az megszűnik létezni. Azt a véglegesen befejeződött folyamatot, amit a szülő folyamat (még) nem gyűjtött be, *zombi folyamat*nak nevezik.

Ha a szülő folyamat anélkül fejeződik be, hogy begyűjtötte volna zombi gyermekeit, a kernel az **init** folyamatra bízza a begyűjtést. Az **init** folyamat PID-ja 1 és azt a kernel hozza létre a rendszer inicializálásakor. Az olyan hosszasan futó programoknak, mint parancs értelmezők vagy kiszolgálók, mindig be kell gyűjteniük a zombi gyerekeiket. Bár a zombik nem futnak, azért még értékes rendszer erőforrásokat tartanak lekötve.

Egy folyamat a **waitpid** függvény,lásd 1.13 programlista, használatával várja meg, amíg gyermekei befejeződnek vagy megállnak. A **waitpid** függvény meglehetősen bonyolult. Alapértelmezetten (amikor **options=0**), a **waitpid** felfüggeszti a hívó folyamat végrehaj-

Programlista 1.13: A **waitpid** függvény használata

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *status, int options);
  //Returns: PID of child if OK, 0 (if WNOHANG) or -1 on
    error
```

tását addig, amíg a **wait set**-ben levő egyik gyermek folyamat be nem fejeződik. Ha a wait setben egy folyamat a hívás idején már befejeződött, a **waitpid** azonnal visszatér. Mindkét esetben, a **waitpid** visszaadja annak a befejeződött gyermek folyamatnak a PID-jét, amelyik a **waitpid** visszatérését okozta, a befejeződött gyermek folyamat pedig eltávolítódik a rendszerből.

1.4.4. A folyamatok altatása

Programlista 1.14: A **sleep** függvény prototípusa

```
#include <unistd.h>

unsigned int sleep(unsigned int secs);

  //Returns: seconds left to sleep
```

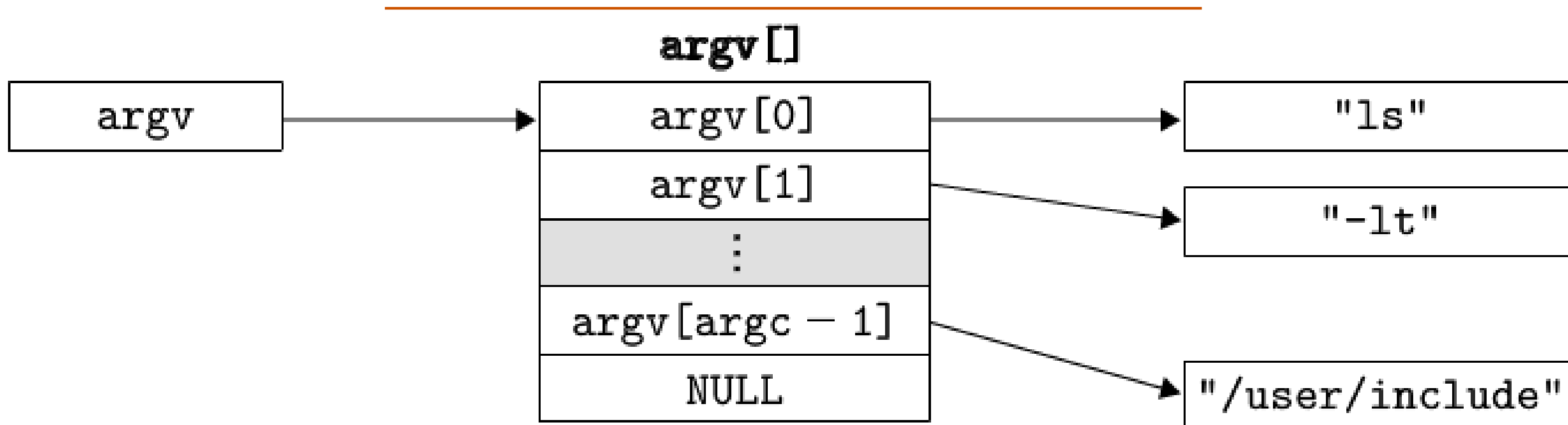
A **sleep** függvény, lásd 1.14 programlista, megadott időtartamra felfüggeszti a folyamatot. A **sleep** nulla értéket ad vissza, ha a megadott idő már eltelt, egyébként pedig a még hátralevő másodpercek számát. Ez utóbbi eset akkor fordulhat elő, amikor a **sleep** hamarabb tér vissza, mivel egy jelzés megszakította. Bővebben az 1.5 szakaszban tárgyaljuk.

1.4.5. Programok betöltése és futtatása

Programlista 1.15: Az **execve** függvény prototípusa

```
#include <unistd.h>

int execve(const char *filename, const char
  *argv[],
  const char *envp[]);

  //Does not return if OK, returns -1 on error
```

Az **execve**, lásd 1.15 programlista, betölt egy új programot és azt futtatja, a jelenlegi környezetben. Az **execve** függvény betölti és lefuttatja a **filename** végrehajtható fájlt az **argv** argumentum listával, az **envp** környezeti változók által megadott környezetben. Az **execve** csak hiba esetén tér vissza a hívó programhoz, mint például ha nem tudta megtalálni a **filename** fájlt. Azaz, a **fork** függvénnyel ellentétben, amit egyszer hívunk és kétszer tér vissza, az **execve** függvényt egyszer hívjuk és sosem tér vissza.

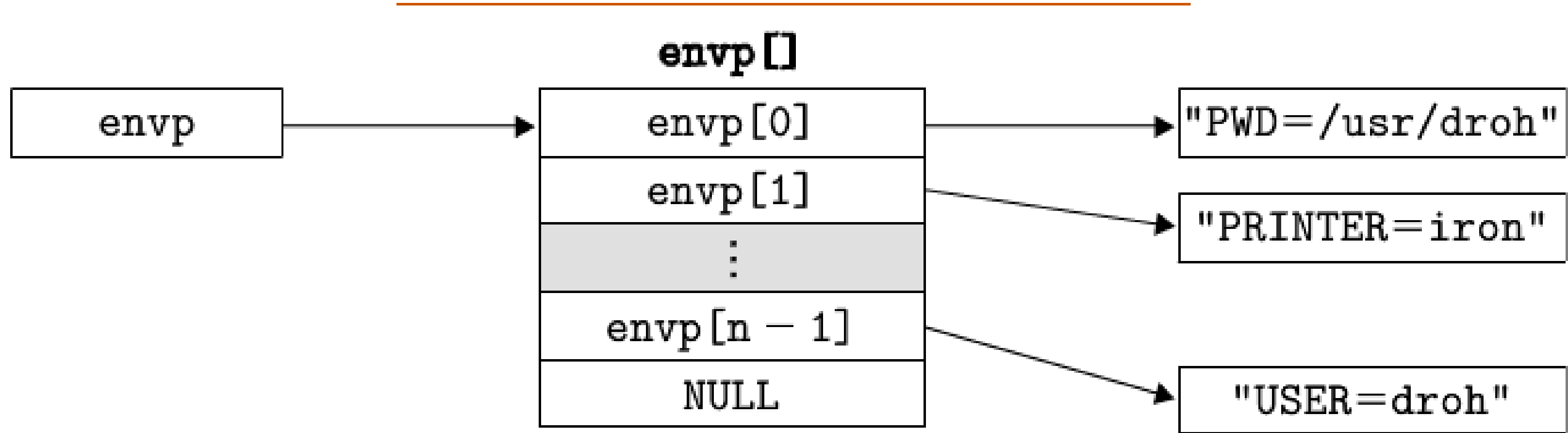


1.11. ábra. Az argumentum lista adatszerkezete

©[**BryantHallaron**] 2014

Az argumentum lista olyan szerkezetű, mint amilyen az 1.11 ábrán látható. Az **argv** változó egy mutatókból álló, nullával határolt tömbre mutat, amely elemek mindegyike egy argumentum sztringre mutat. Megállapodás szerint **argv[0]** a végrehajtható fájl neve.

A környezeti változókat is egy hasonló adatszerkezet adja meg, lásd 1.12 ábra. Az **envp**



1.12. ábra. A környezeti változók lista adatszerkezete

©[**BryantHallaron**] 2014

változó egy nullával határolt, a környezeti változókra mutató pointerkből álló tömbre mutat, amelynek elemei egy "NÉV=ÉRTÉK" párra mutatnak.

Miután az **execve** betölti **filename**-et, meghív egy indító kódot, amelyik beállítja a vermet, majd átadja a vezérlést az új programnak, aminek a prototípusa

```
int main(int argc, char **argv, char **envp);
```

vagy az ezzel egyenértékű

```
int main(int argc, char *argv[], char *envp[]);
```

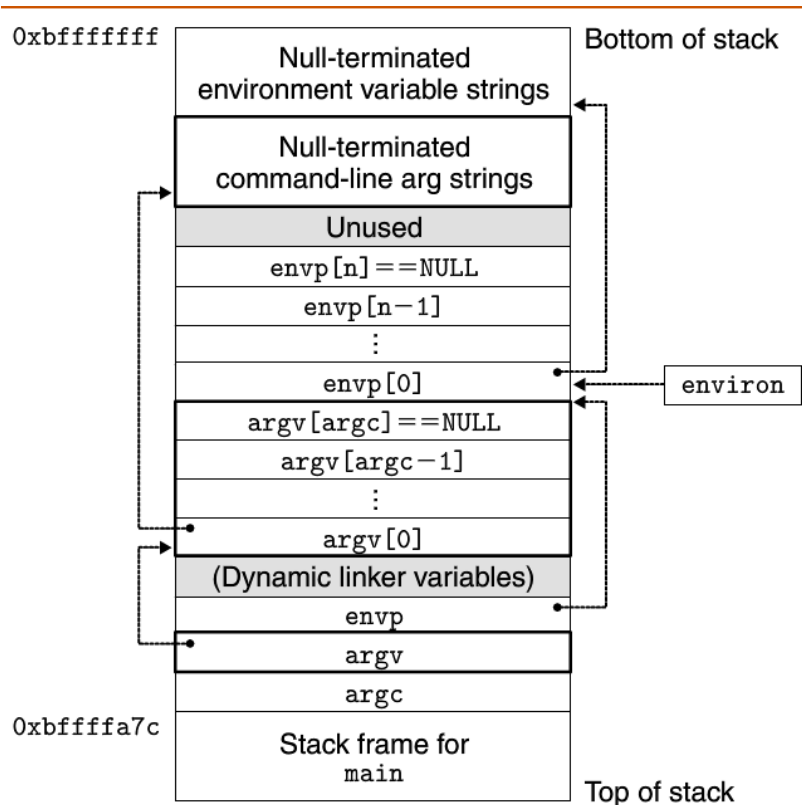
alakú.

Amikor egy 32-bites Linx folyamatban a **main** futni kezd, a felhasználói veremtároló a 1.13 ábra szerinti szerkezetű. Nézzük végig azt a verem aljától (a legmagasabb címtől) a tetejéig (a legalacsonyabb címig). Elsőként az argumentum és környezeti sztringeket találjuk meg, amelyek a veremben folytonosan helyezkednek el, elválasztás nélkül. Ezeket követi a veremben felfelé egy nullával határolt mutatókból álló tömb, amelynek elemei egy szintén a veremben tárolt környezeti változó sztringre mutatnak. Az environ globális környezeti változó ezen pointerek közül az elsőre, **envp[0]**-ra mutat.

A környezeti tömböt közvetlenül követi a nullával lezárt **argv[]** tömb, amelynek minden eleme a szintén a veremben elhelyezett argumentum sztringre mutat. A verem tetején találjuk a **main** rutin három argumentumát:

- **envp**, ami az **envp[]** tömbre mutat
- **argv**, ami az **argv[]** tömbre mutat
- **argc**, ami az **argv[]** tömb nem-nulla elemeinek számát adja meg.

A Unix alatt több függvény is szolgál a környezet kezelésére. A **getenv** függvény, lásd



1.13. ábra. A felhasználó veremtárolójának szerkezete egy új program elindulásakor

©[**BryantHallaron**] 2014

Megjegyzés: Program vs. process

Értsük meg jól a program és a folyamat közötti különbséget. A program kódból és adatokból álló gyűjtemény; a program létezhet objekt modulként a mágneslemezen vagy szegmensként a memória térben. A folyamat a program egy bizonyos példánya végrehajtás közben; egy program mindig egy folyamat által biztosított környezetben fut. Ennek a különbségnek a megértése nagyon fontos ahhoz, hogy megértsük a **fork** és az **execve** függvényeket. A **fork** függvény ugyanazt a programot futtatja egy új gyermek folyamatként, ami a szülő folyamatnak egy másolata. Az **execve** függvény egy új programot tölt be az aktuális folyamat környezetébe és ott futtatja azt. Bár felülírja az aktuális folyamat címterét, de *nem* hoz létre új folyamatot. Az új programnak még mindig ugyanaz lesz a PID-je és örökli az összes fájl leírót, amelyhez tartozó fájlok az **execve** függvény hívásakor nyitva voltak.

1.16 lista, megkeresi a környezeti tömbben a "**name=value**” sztringet. Ha megtalálja, egy annak értékére mutató pointert ad vissza, különben a **NULL** értéket.

Ha a környezet tömb tartalmaz egy “**name=oldvalue**” formájú sztringet, akkor **unsetenv**

Programlista 1.16: A **getenv** függvény prototípusa

```
#include <stdlib.h>

char *getenv(const char *name);

  //Returns: ptr to name if exists, NULL if no match
```

törli azt, és **setenv** helyettesíti az **oldvalue** értéket a **newvalue** értékkel, amennyiben **overwrite** nem nulla értékű. Ha **name** nem létezik, akkor **setenv** hozzáadja a “**name=newvalue**” értéket a tömbhöz.

Programlista 1.17: A **setenv** függvény használata

```
#include <stdlib.h>

int setenv(const char *name, const char *newvalue,
   int overwrite);

    //Returns: 0 on success, -1 on error

void unsetenv(const char *name);

    //Returns: nothing
```

1.4.6. A fork és az execve használata program futtatásra

Az olyan programok, mint a Unix parancs értelmezők és Web kiszolgálók, kiterjedten használják a **fork** és **execve** függvényeket. A parancsértelmező olyan interaktív alkalmazásszintű program, amelyik a felhasználó nevében más programokat futtat. Az eredeti parancsértelmező az **sh** program volt, amelyeket olyan variánsok követtek, mint a **csh**, **tcsh**, **ksh** és **bash**. Egy parancsértelmező olvasás/értelmezés lépések sorozatát hajtja végre, majd kilép. Az olvasási lépésben beolvas a felhasználótól egy utasítás sort. Az értelmezési lépésben értelmezi az utasítás sort és programokat futtat a felhasználó nevében.

Az utasítássor kiszámításának módját mutatja az 1.19 lista. Ennek első feladata hogy meghívja a **parseline** függvényt (1) (lásd 1.21 lista), ami értelmezi a betűközökkel elválasztott parancssori argumentumokat, amelyből összeállítja az **argv** vektort (2), amit majd át kell adnunk (3) az **execve** függvénynek. Az első argumentumról feltételezzük, hogy az vagy a parancs értelmező egy beépített utasítása, vagy pedig egy végrehajtható fájl neve, amit be kell tölteni és egy új gyermek folyamat környezetében futtatni. Ha az utolsó argumentum egy "**&**" karakter (4) , lásd 1.21 lista, akkor **parseline** az **1**

Programlista 1.18: Az egyszerű parancs értelmező **main** rutinja

```
#include "csapp.h"
#define MAXARGS  128

 /* Function prototypes */
void eval(char *cmdline);
int parseline(char *buf, char **argv);
int builtin_command(char **argv);

int main()
{
  char cmdline[MAXLINE]; /* Command line */

   while (1) {
   /* Read */
   printf("> ");
   Fgets(cmdline, MAXLINE, stdin);
   if (feof(stdin))
```

Programlista 1.19: Az egyszerű parancs értelmező

```
/* eval - Evaluate a command line */
void eval(char *cmdline)
{
  char *argv[MAXARGS]; /* Argument list execve() */
   char buf[MAXLINE];  /* Holds modified command line */
   int bg;    /* Should the job run in bg or fg? */
   pid_t pid;  /* Process id */

   strcpy(buf, cmdline);
   bg = parseline(buf, argv); (1)
   if (argv[0] == NULL)
    return;  /* Ignore empty lines */

   if (!builtin_command(argv)) { (2)
       if ((pid = Fork()) == 0) {  /* Child runs user
         job */
         if (execve(argv[0], argv, environ) < 0) { (3)
```

értéket adja vissza, ami azt jelzi, hogy a programot a háttérben kell futtatni (azaz a parancs értelemzőnek nem kell megvárnia, amíg az befejeződik). Egyébként a visszaadott érték **0**, ami azt jelzi, hogy a programot az előtérben kell futtatni (5) (azaz a parancsértelmezőnek meg kell várnia, amíg az befejeződik).

A kapott utasítássor értelmezése után az értelmező függvény meghívja a **builtin_command** függvényt (2), lásd 1.21 lista, ami megvizsgálja, hogy az utasítássor első argumentuma a parancsértelmező beépített parancsa-e. Ha igen, azonnal értelmezi az utasítást, és **1** értéket ad vissza. Egyébként a visszaadott érték **0**. Ebben az egyszerű parancsértelmezőben csak egyetlen beépített parancs van: a **quit**, ami lezárja a parancsértelmezőt. A valódi parancsértelmezőkben számos parancs van, min tpéldául **pwd**, **jobs** és **fg**.

Ha **builtin_command** visszatérési értéke **0**, akkor a parancsértelmező egy gyermek folyamatot hoz létre, és azon belül hajtja végre a kért programot (3), lásd 1.21 lista. Ha a felhasználó azt kérte, hogy a program a háttérben fusson, a parancsértelmező visszatér a ciklus elejére és vár a következő utasítás sorra. Egyébként pedig a parancsértelmező a **waitpid** függvényt használja arra, hogy megvárja az elindított job befejeződését. A befejeződés után a parancsértelmező a következő iterációval folytatja.

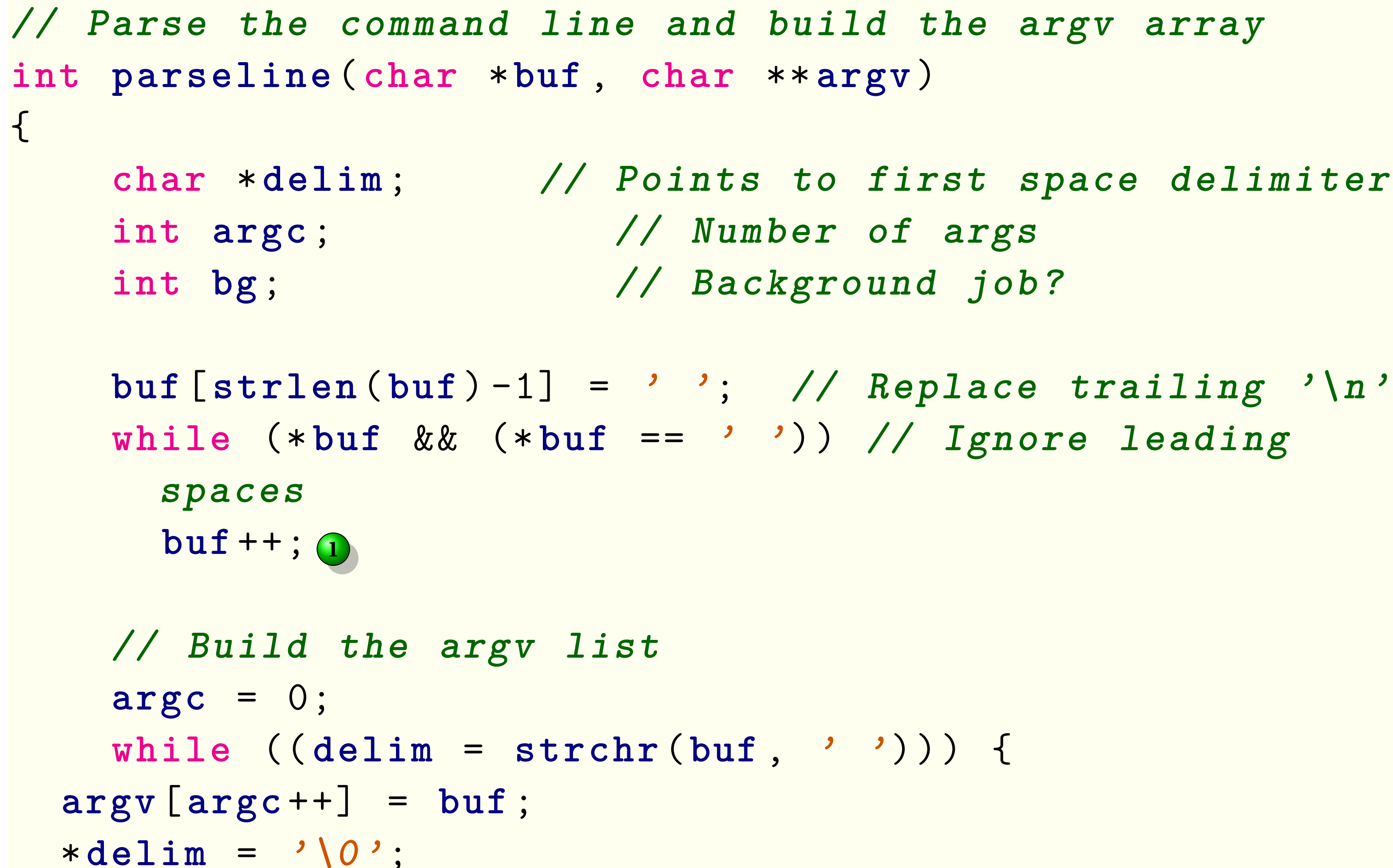
Programlista 1.20: Az egyszerű parancs értelmező beépített parancsa

```
/* eval - Evaluate a command line */
void eval(char *cmdline)
{
  char *argv[MAXARGS]; /* Argument list execve() */
   char buf[MAXLINE];  /* Holds modified command line */
   int bg;     /* Should the job run in bg or fg? */ (1)
   pid_t pid;  /* Process id */

   strcpy(buf, cmdline); (2)
   bg = parseline(buf, argv);
   if (argv[0] == NULL)
    return;  /* Ignore empty lines */

   if (!builtin_command(argv)) {
       if ((pid = Fork()) == 0) {  /* Child runs user
         job */
         if (execve(argv[0], argv, environ) < 0) {
```

Vegyük észre, hogy ez az egyszerű parancsértelmező hibás, mivel nem gyűjti össze a háttérben futó gyermek folyamatokat. A hiba kijavítása a jelzések használatát igényli, amit majd a következő szakaszban tanulunk meg.

Érdemes egy kicsit vesződni a felhasználó által begépelt karakterek lehetséges változataival már az elején. (1) Először eltüntetjük az információt nem hordozó, "láthatatlan" karaktereket. (2) Most már az argumentum lista betűközzel elválasztva tartalmazza az argumentumokat, felépítjük az **argv[]** tömböt. (3) Jelezzük az argumentum lista végét. (4) Elhanyagoljuk az üres sorokat. (5) Jelezzük, ha háttérben futtatást kért a felhasználó.

Programlista 1.21: Egy parancssor értelmezése a parancs értelmező számára



```
// Parse the command line and build the argv array
int parseline(char *buf, char **argv)
{
    char *delim;        // Points to first space delimiter
    int argc;           // Number of args
    int bg;             // Background job?

    buf[strlen(buf)-1] = ' ';  // Replace trailing '\n'
    while (*buf && (*buf == ' ')) // Ignore leading
      spaces
      buf++;

    // Build the argv list
    argc = 0;
    while ((delim = strchr(buf, ' '))) {
  argv[argc++] = buf;
  *delim = '\0';
```

1.5. Jelzések

Az eddigiekben tanultunk a kivételes vezérlési folyamról, láttuk, hogyan működik együtt a hardver és a szoftver, hogy biztosítsák az alapvető alacsony szintű kivétel kezelési mechanizmust. Azt is láttuk, hogyan használja az operációs rendszer a kivételeket arra, hogy a környezetváltás néven ismert kivételes vezérlési folyamot támogassa. Ebben a szakaszban a kivételes vezérlési folyam egy magas szintű szoftver formájával fogunk megismerkedni, ami Unix jelzés (signal) névre hallgat és ami lehetővé teszi, hogy a kernel és a folyamatok más folyamatokat megszakítsanak.

Megjegyzés: Linux signals.

Notes: (1) Years ago, main memory was implemented with a technology known as core memory. “Dumping core” is a historical term that means writing an image of the code and data memory segments to disk. (2) This signal can neither be caught nor ignored.

#	Name	Default action	Corresponding event
1	SIGHUP	Terminate	Terminal line hangup
2	SIGINT	Terminate	Interrupt from keyboard
3	SIGQUIT	Terminate	Quit from keyboard
4	SIGILL	Terminate	Illegal instruction
5	SIGTRAP	Terminate and dump core(1)	Trace trap
6	SIGABRT	Terminate and dump core(1)	Abort signal from **abort** function
7	SIGBUS	Terminate	Bus error
8	SIGFPE	Terminate and dump core(1)	Floating point exception
9	SIGKILL	Terminate (2)	Kill program
10	SIGUSR1	Terminate	User defined signal 1
11	SIGSEGV	Terminate and dump core(1)	Invalid memory reference (seg fault)

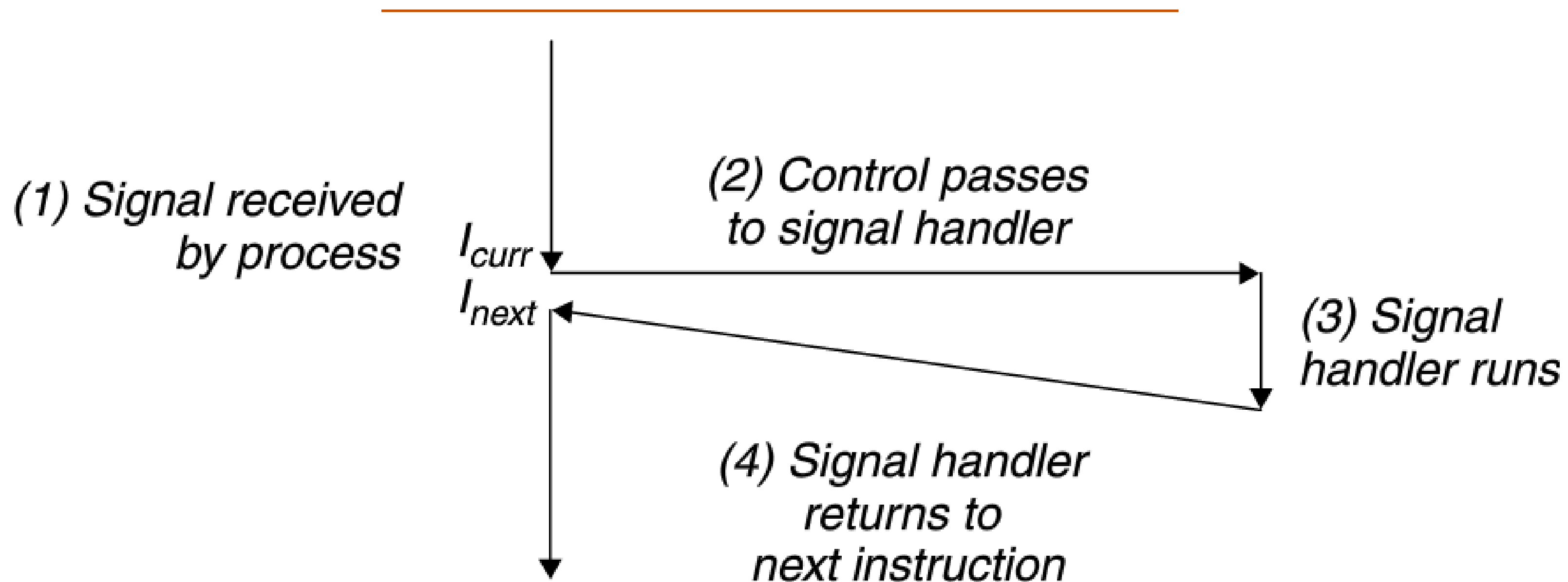
Megjegyzés: Jelzés (signal)

A jelzés (signal) egy apró jelzés, ami arról értesíti a folyamatot, hogy egy bizonyos fajta esemény történt a rendszerben. A Linux rendszer alatt 30 különféle ilyen esemény létezik, a “man 7 signal” beírására a parancsértelmező kiírja ezek listáját. Mindegyik jelzés valamilyen rendszer eseménynek felel meg. Az alacsony szintű hardware kivételeket a kernel kivétel kezelői dolgozzák fel és azok normál körülmények között nem láthatók a folyamatok számára. A jelzések arra szolgálnak, hogy az ilyen kivételek előfordulását láthatóvá tegyék a folyamatok számára. Például, ha egy folyamat nullával próbál osztani, akkor a kernel egy SIGFPE jelzést (8 szám) küld. Ha egy folyamat illegális utasítást hajt végre, a kernel egy SIGILL jelzést (4 szám) küld. Ha egy folyamat illegális memóriára hivatkozik, a kernel egy SIGSEGV jelzést (11 szám) küld. Más jelzések a kernel vagy egymásik felhasználói folyamat magas szintű szoftver eseményének felelnek meg. Például, ha **ctrl-c** gépelünk (a ctrl gombot lenyomva tartva leütjük a c-t) amikor egy folyamat az előtérben fut, a kernel egy SIGINT jelzést (2 szám) küld az előtér folyamatnak. Egy folyamat erőszakkal kilépésre kényszeríthet egy másik folyamatot, ha SIGKILL jelzést (9 szám) küld neki. Amikor egy gyermek befejeződik vagy megáll, a a kernel egy SIGCHLD jelzést (17 szám) küld a szülőnek.

1.5.1. A jelzések terminológiája

Egy jelzés átvitele a cél folyamatba két jól elkülönülő lépésben történik:

- ***Elküldés*** A kernel úgy küldi el (szállítja el) a jelzést a címzett folyamatnak, hogy a cél folyamat környezetének valamely állapotát megváltoztatja. A jelzés elküldésének két oka lehet
  - A kernel észlelt egy olyan rendszer eseményt, mint például nullával való osztás vagy egy gyermek folyamat befejeződése
  - Egy folyamat a **kill** függvény hívásával explicit módon kérte a kernelt, hogy küldjön jelzést a címzett folyamatnak. Egy folyamat saját magának is küldhet jelzést.
- ***Fogadás*** A címzett folyamat akkor fogad jelzést, amikor a kernel arra kényszeríti, hogy valamilyen módon reagáljon a küldött jelre. A folyamat elhanyagolhatja a jelzést, befejeződhet, vagy elkapja a jelzést egy felhasználói szintű függvény (amit jelzés kezelőnek hívnak) hívásával. Az 1.14 ábra mutatja a jelzés elkapásának lényegét.



1.14. ábra. Jelzés kezelés. Egy jelzés fogadása kiváltja a vezérlés átadását a jelzés kezelőbe. Miután befejezte működését, a kezelő visszaadja a vezérlést a megszakított programnak.

©[**BryantHallaron**] 2014

Megjegyzés: Függő jelzés

Azt a jelzést, amit már elküldtek, de még nem kaptak meg, függő jelzésnek nevezik. Egy bizonyos típusú eseményből egyidejűleg csak egyetlen függő jelzés lehet. Amennyiben egy folyamatnak már van egy k típusú függő jelzése, akkor az ennek a folyamatnak küldött további k típusú jelzések nem állnak sorba, egyszerűen elhanyagolódnak. Egy folyamat szelektíven blokkolhatja bizonyos jelzések fogadását. Amikor egy jelzés blokkolt állapotban van, az nem szállítható le, de az így keletkező függő jelzés addig nem érkezik meg, amíg a folyamat nem oldja fel a blokkolást.

Egy függő jelzést legfeljebb egyszer lehet megkapni. Az egyes folyamatokra a kernel a függő jelzéseket egy **pending** bit vektorban tartja nyilván, a blokkolt jelzéseket pedig a **blocked** vektorban. Amikor egy k típusú jelzést elküld, a kernel a **pending** vektorban a k-adik bitet egyre állítja, amikor a jelzés megérkezik, akkor pedig nullázza a bitet.

1.5.2. Jelzések küldése

A Unix számos mechanizmust biztosít arra, hogy jelzéseket küldjünk a folyamatoknak. Valamennyi mechanizmus a jelzés csoport (process group) fogalmán alapszik.

Jelzés csoportok

Minden egyes folyamat pontosan egy folyamat csoporthoz tartozik, amelyet folyamat csoport szám azonosító (process group ID) pozitív egész szám ad meg. A **getpgrp** függvény (l. 1.22 lista) a futó folyamat csoport szám azonosítójának értékét adja vissza.

Programlista 1.22: A **getpgrp** függvény használata

```
#include <unistd.h>
pid_t getpgrp(void);
      //Returns: process group ID of calling process
```

Alapértelmezetten, a gyermek folyamat ugyanahhoz a folyamat csoporthoz tartozik, mint a szülő folyamat. Egy folyamat a **setpgid** függvény használatával megváltoztathatja saját maga vagy más folyamat csoport azonosítóját.

Programlista 1.23: A **setpgrp** függvény használata

```
#include <unistd.h>
int setpgid(pid_t pid, pid_t pgid);

    //Returns: 0 on success, -1 on error
```

A **setpid** a **pid** folyamat csoport azonosítóját **pgid**-re változtatja. Ha **pid** értéke nulla, akkor az aktuális folyamat PID értéke használódik. Ha **pgid** értéke nulla, akkor a **pid** által meghatározott folyamat PID értéke használódik folyamat csoport azonosítóként. Például, ha a 15213 folyamat a hívó folyamat, akkor a setpgid(0, 0); hívás egy új folyamat csoportot hoz létre, amelynek csoport azonosítója 15213, és a 15213 folyamatot ehhez az új csoporthoz adja.

Jelzés küldése a /bin/kill programmal

A **/bin/kill** program tetszőleges jelzést elküld egy másik folyamatnak. Például, a

```
/bin/kill -9 15213
```

utasítás a 9 (SIGKILL) parancsot küldi el a 15213 folyamatnak. Egy negatív értékű PID hatására a jelzés a PID folyamat csoportban minden folyamatnak elküldődik. Például, a

```
/bin/kill -9 -15213
```

elküldi a SIGKILL jelzést a 15213 folyamat csoportban levő összes folyamatnak. Megj: azért használjuk a **/bin/kill** teljes útvonalat, mert bizonyos Unix parancsértelmezők saját (beépített) **kill** utasítással rendelkeznek.

Jelzés küldése a billentyűzetről

A Unix parancsértelmezők a job absztrakciót használják azoknak a folyamatoknak az ábrázolására, amelyek egy utasítás sor kiértékelésének eredményeként jönnek létre. Bármely időpillanatban van legfeljebb egy előtérbeli és nulla vagy több háttérbeli *job*. Például, a

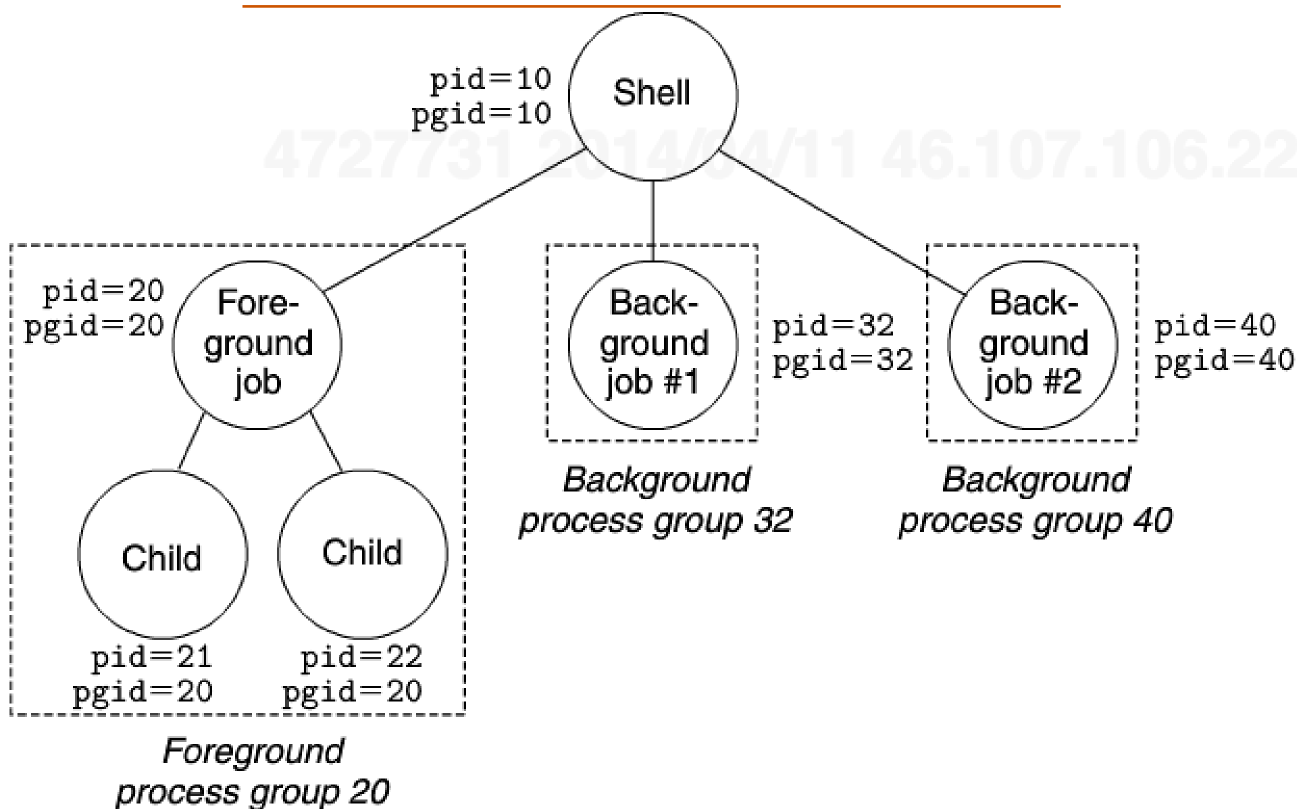
```
ls | sort
```

leírása létrehoz egy olyan előtérbeli *job*-ot, amelyik két, csővezetékkel összekötött folyamatból álló *job*ot hoz létre: az egyik az **ls** programot futtatja, a másik meg a **sort**

programot.

A parancs értelmező minden egyes *job* számára külön folyamatot hoz létre. A folyamat csoport azonosítója (group ID) a *job* egyik szülő folyamatából származik. Például az 1.15 ábra egy előtérbeli és két háttérbeli *job*ot mutat. Az előtérbeli *job* szülő folyamatának PID értéke 20, és a folyamat csoport azonosítója (process group ID) is 20. A szülő folyamat két gyermek folyamatot hozott létre, amelyek mindegyik a 20-as csoportnak a tagja.

Amikor a billentyűzeten a **ctrl-c** karaktert leütjük, annak hatására a parancsértelmezőnek egy SIGINT jel küldődik. A parancsértelmező elkapja a jelzést és sz előtérbeli folyamamt csoport minden folyamatának SIGINT jezést küld. Alapértelmezetten ennek hatására az előtérbeli *job* befejeződik. Hasonlóképpen, **ctrl-z** karaktert begépelve, a SIGTSTP jelzés küldődik el a parancs értelmezőnek, ami azt elkapja és SIGTSTP jelzést küld az előtérbeli folyamat csoport minden folyamatának. Alapértelmezetten ennek hatása, hogy megáll (felfüggesztődik) az előtérbeli *job*.



1.15. ábra. Előtérbeli és háttérbeli folyamat csoportok

©[**BryantHallaron**] 2014

Jelzés küldése a **kill** függvénnyel

A folyamatok más folyamatoknak (de akár saját maguknak is) jelzést küldhetnek a **kill** függvény hívásával, lásd 1.24 lista.

Programlista 1.24: A **kill** függvény használata

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int sig);

      //Returns: 0 if OK, -1 on error
```

Ha **pid** nullánál nagyobb, akkor a **kill** függvény a **sig** számnak megfelelő jelzést küldi a **pid** folyamatnak. Ha **pid** nullánál kisebb, akkor a **kill** függvény a **sig** számnak megfelelő jelzést küldi el az **abs(pid)** folyamat csoportban levő összes folyamatnak, lásd 1.25 lista.

A folyamatok jelzéseket küldenek más folyamatoknak (vagy akár saját maguknak is) a **kill** függvény meghívásával. Ha **pid** nullánál nagyobb, akkor a **kill** függvény a **sig**

értékű jelet küldi el a **pid** folyamatnak. Ha **pid** nullánál kisebb, akkor a **kill** a **abs(pid)** csoport minden folyamatának elküldi a **sig** jelzést. Az 1.25 egy olyan szülő folyamatot mutat, amelyik az általa létrehozott gyermek folyamatnak küld egy **SIGKILL** jelzést a **kill** függvény használatával.

Jelzés küldése az alarm függvénnyel

Egy folyamat a SIGALRM jelzést küldheti magának az **alarm** függvény hívásával, lásd 1.26 lista.

Az **alarm** függvény megbízza a kernelt, hogy **secs** másodperc múlva küldjön a hívó folyamatnak SIGALRM jelzést. Ha **secs** nulla, nem ütemeződik új riasztás. Bármelyik esetben, az **alarm** hívása a korábbi összes riasztást törli, visszatérési értékként pedig az esedékes riasztásig fennmaradt másodpercek számát adja vissza, vagy pedig nulla értéket, ha nem volt beállított riasztás.

Az **alarm** program az 1.27 listán másodpercenként megszakíttatja magát öt másodpercen keresztül. Amikor a hatodik **SIGALRM** is megérkezik, a program befejeződik. Amikor futtatjuk a programot, a következő kimenetet látjuk: öt másodpercig minden

Programlista 1.25: Hogyan használjuk a **kill** függvényt arra, hogy jelzést küldjünk egy gyermek folyamatnak

```
#include "csapp.h"

int main()
{
  pid_t pid;

  /* Child sleeps until SIGKILL signal received, then
    dies */
  if ((pid = Fork()) == 0) {
     Pause();  /* Wait for a signal to arrive */
    printf("control should never reach here!\n");
    exit(0);
  }

  /* Parent sends a SIGKILL signal to a child */
  Kill(pid, SIGKILL);
```

Programlista 1.26: Az **alarm** függvény prototípusa

```
#include <unistd.h>

unsigned int alarm(unsigned int secs);

    //Returns: remaining secs of previous alarm, or 0 if
      no previous alarm
```

másodpercben egy “BEEP”, majd a program befejeződéskor “BOOM”.

Megjegyezzük, hogy az 1.27 listán a program a **signal** függvényt használja a **handler** jelkezelő installálására, ami aztán aszinkron módon hívódik meg, minden alkalommal megszakítva a **main** végtelen ciklusát, amikor a folyamat megkapja a **SIGALRM** jelzést. Amikor a **handler** függvény visszatér, a vezérlés visszakerül a **main**-be, ami ott folytatódik, ahol a jelzés megérkezésekor megszakadt. A jelzés kezelők intallálása és használata elég kényes téma, azért a következő szakaszokban ezzel foglalkozunk.

Programlista 1.27: Példa az **alarm** függvény használatára

```
#include "csapp.h"

void handler(int sig)
{
  static int beeps = 0;

  printf("BEEP\n");
  if (++beeps < 5)
    Alarm(1); /* Next SIGALRM will be delivered in 1
      second */
  else {
    printf("BOOM!\n");
  exit(0);
  }
}

int main()
```

1.5.3. Jelzések fogadása

Amikor a kernel visszatér egy kivétel kezelőből és készen áll arra, hogy átadja a vezérlést a **p** folyamatnak, megvizsgálja a **p** folyamat nem blokkolt függő jelzéseinek halmazát (függő & blokkolt). Ha ez a halmaz üres (általában ez fordul elő), akkor a kernel átadja a vezérlést a **p** logikai vezérlésfolyam következő utasításának (Inext).

Ha azonban a halmaz nem üres, a kernel kiválasztja valamelyik *k* jelzést (rendszerint a legkisebb *k* értéket) és arra kényszeríti a **p** folyamatot, hogy fogadja a *k* jelzést. A jelzés fogadása valamilyen cselekvést vált ki a folyamatban. Amikor a cselekvés befejeződik, a vezérlés visszaadódik a **p** logikai vezérlésfolyam következő utasításának (Inext).

Mindegyik jelzésnek van egy előre meghatározott alapértelmezett hatása, a következők közül:

- A folyamat befejeződik.
- A folyamat befejeződik és kinyomtatja a bináris állapotát.
- A folyamat megáll, amíg egy SIGCONT jelzés újra nem indítja.
- A folyamat elhanyagolja a jelzést.

A **??** táblázat tartalmazza az egyes jelzés típusokhoz tartozó alapértelmezett cselek-

Programlista 1.28: A **signal** függvény prototípusa

```
#include <signal.h>
typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t
  handler);

  //Returns: ptr to previous handler if OK, SIG_ERR on
    error (does not set errno)
```

vést. Például, a SIGKILL alapértelmezett cselekvése, hogy befejezi a fogadó folyamatot. Másrészt viszont, a SIGCHLD esetén az alapértelmezett cselekvés a jelzés elhanyagolása. Egy folyamat a jelzés függvény használatával módosíthatja az alapértelmezett cselekvést. A SIGSTOP és SIGKILL kivételek, amelyeknek az alapértelmezett cselekvése nem módosítható.

A **signal** függvény az alábbi három mód egyikével módosíthatja a **signum** jelzéshez

társuló cselekvést:

- Ha a **handler** értéke SIG_IGN, akkor a **signum** típusú jelzések elhanyagolódnak.
- Ha a **handler** értéke SIG_DFL, akkor a **signum** típusú jelzések cselekvése visszaállítódik az alapértelmezettre.
- Egyébként a **handler** a felhasználó által megadott függvény, amit jelzés kezelőnek neveznek, és ami akkor hívódik majd meg, amikor a folyamat **signum** típusú jelzést kap. Az a művelet, amikor az alapértelmezett cselekvést a kezelő függvény címét a jelzés kezelő függvénynek átadjuk, a jelzés kezelő üzembeállításaként ismert. A jelzés kezelő meghívása a jelzés elkapása néven ismert. A jelzés kezelő végrehajtása pedig a jelzés kezeléseként ismert.

Amikor egy folyamat elkap egy k típusú jelzést, akkor a k jelzéshez üzembe állított kezelő egy k egész argumentummal hívódik meg. Ez lehetővé teszi, hogy ugyanaz a kezelő különböző típusú jelzéseket is elkapjon.

Amikor a kezelő eléri a **return** utasítást, a vezérlés (általában) abba a vezérlési folyamba adódik vissza, amelyben a folyamat megszakítódott a jelzés érkezésekor. Azért általában, mert bizonyos rendszerekben a megszakított rendszerhívás azonnali hibát eredményez.

Programlista 1.29: Hogyan lehet a **SIGINT** jelzést a jelzéskezelő függvénnyel elkapni.

```
#include "csapp.h"

void handler(int sig)/* SIGINT handler */ (1)
{
  printf("Caught SIGINT\n"); (4)
  exit(0); (5)
}

int main()
{
   /* Install the SIGINT handler */
   if (signal(SIGINT, handler) == SIG_ERR)
     unix_error("signal error"); (2)

  pause(); /* Wait for the receipt of a signal */
  exit(0); (3)
}
```

Az 1.29 lista egy olyan programot mutat, amelyik elkapja a SIGINT jelzést, amit a parancs értelmező akkor küld, amikor a felhasználó **ctrl-c**-t üt le a billenytűzeten. A SIGINT alapértelmezett cselekvése, hogy azonnal befejezi a folyamatot. Ebben a példában úgy módosítjuk az alapértelmezett cselekvést, hogy elkapjuk a jelzést, kinyomtatunk egy üzenetet, és befejezzük a folyamatot. A főprogram a jelzés kezelőt (1) üzembeállítja (2), majd aludni tér(3), amíg jelzés nem érkezik. Amikor a **SIGINT** jelzés megérkezik, a handler futni fog, kinyomtatja az üzenetet (4) és befejezi a folyamatot (5).

A jelzés kezelők egy újabb példa a számítógépes rendszerekben előforduló konkurrenciára. A jelzés kezelő végrehajtása megszakítja a fő C rutin végrehajtását, ahhoz hasonló módon, ahogyan egy alacsony szintű kivétel kezelő megszakítja az alkalmazói program futását. Mivel a jelzés kezelő és a főprogram vezérlési folyama időben átfed, a jelzés kezelő és a főprogram konkurrens módon futnak.

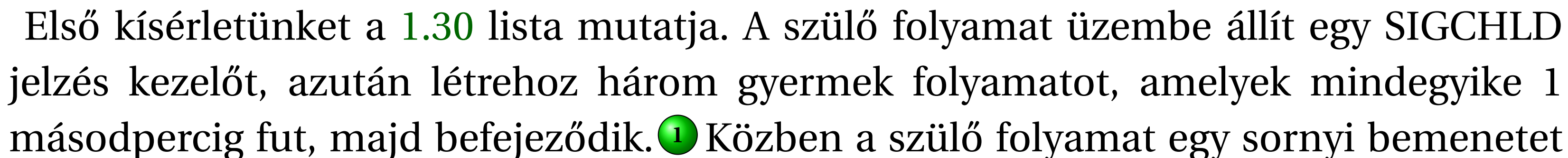
1.5.4. A jelzések kezelésének finomságai

A jelzések kezelése nagyon jól érthető azon programok esetén, amelyek elkapják a jelzést és befejeződnek. Kétséges helyzetek fordulhatnak elő azonban, ha egy program többféle jelzést is fogadhat.

- ***Blokkolt függő jelzés*** A Unix jelkezelői általában blokkolják az olyan típusú függő jelzéseket, amilyeneket a kezelő éppen feldolgoz. Például, tegyük fel, hogy egy folyamat elkapott egy SIGINT jelzést és most éppen futtatja a SIGINT jelzés kezelőjét. Ha egy másik SIGINT jelzést is küldenek a folyamatnak, a SIGINT függővé válik (hiszen elküldték), de nem fogadják amíg a kezelő vissza nem tér.
- ***A függő jelzések nem állnak sorba*** Bármilyen típusú jelzésből legfeljebb csak egy függő lehet. Emiatt, ha két *k* típusú jelzést küldtek egy címzett folyamatnak, amíg a *k* jelzés blokkolva van, akkor a második jelzés egyszerűen eldobódik; nem kerül be várakozó sorba. Az alap elképzelés, hogy egy függő jelzés létezése csupán ezt jelzi, hogy *legalább egy* jelzés érkezett.
- ***A rendszer hívások megszakíthatók*** Az olyan rendszerhívásokat, mint **read**, **wait** és **accept**, amelyek hosszú időszakokra blokkolhatják a folyamatokat, lassú

rendszerhívásoknak nevezik. Bizonyos rendszereken a lassú rendszerhívások, amelyek megszakadnak, amikor a kezelő elkap egy jelzést, nem folytatódnak, amikor a jelzéskezelő visszatér, hanem a felhasználói programhoz térnek vissza hibajelzéssel, az **errno** értékét EINTR értékékre állítva.

Nézzük meg közelebbről a jelzés kezelés eme finomságait egy olyan példán keresztül, amelyik jellegében nagyon hasonlít olyan valódi programokhoz, mint a parancsértelmezők vagy Web kiszolgálók. Az alaphelyzet, hogy a szülő folyamat létrehoz néhány gyermek folyamatot, amelyek egy ideig függetlenül futnak, majd befejeződnek. A szülő folyamatnak be kell gyűjtenie a gyermekeit, hogy ne hagyjon zombi folyamatokat a rendszerben. Emellett azt is szeretnénk, ha a szülő folyamat tudna valami hasznosat végezni, miközben a gyermekek futnak. Ezért úgy döntünk, hogy a gyermek folyamatokat a SIGCHLD kezelővel gyűjtjük be, ahelyett, hogy várnánk a gyermek folyamatok befejeződésére. (Emlékezzünk rá, hogy a kernel SIGCHLD jelzést küld a szülőnek, amikor valamelyik gyermeke befejeződik vagy megáll.)



Első kísérletünket a 1.30 lista mutatja. A szülő folyamat üzembe állít egy SIGCHLD jelzés kezelőt, azután létrehoz három gyermek folyamatot, amelyek mindegyike 1 másodpercig fut, majd befejeződik. (1) Közben a szülő folyamat egy sornyi bemenetet

Programlista 1.30: **signal1: főprogram** Ez a program azért hibás mert nem foglalkozik azzal hogy egy jelzés blokkolva is lehet; a jelzések nem íródnak be a sorba; és a rendszerhívások megszakíthatók.

```
#include "csapp.h"

void handler1(int sig)
{
   pid_t pid;

   if ((pid = waitpid(-1, NULL, 0)) < 0)
     unix_error("waitpid error");
   printf("Handler reaped child %d\n",(int)pid);
   Sleep(2);
  return; (1)
}

int main()
{
```

vár a terminálról, majd azt feldolgozza. Ezt a fajta végrehajtást modellezi egy végtelen ciklus. (2)

Amikor valamelyik gyermek folyamat befejeződik, a kernel egy SIGCHLD jelzés küldésével értesíti a szülő folyamatot. A szülő folyamat elkapja a SIGCHLD jelzést, begyűjti a gyermeket, végez valamiféle egyéb feldolgozást (ezt modellezi a **sleep(2)**), majd visszatér.

A 1.30 lista szerinti program elég egyszerűnek látszik. Amikor azt egy Linux rendszeren futtatjuk, a 1.32 lista szerinti eredményt kapjuk.

A kiírásból azt látjuk, hogy bár három **SIGCHLD** jelet küldtünk a szülő folyamatnak, abból csak kettőt kapott meg, és ennek megfeleően csak két gyermek folyamatot gyűjtött be. Ha felfüggesztjük a szülő folyamatot, azt látjuk, hogy valóban, a 10321 számú gyermek folyamat nem gyűjtődött be, így az zombi maradt (amint azt a "defunct" szöveg jelzi a **ps** utasítás hatására kiírt szövegben, lásd 1.33 lista:)

De mi is a baj? Kódunk nem vette figyelembe azt a tényt, hogy a jelzések blokkolódhatnak és hogy nem állnak sorba. A következő történt: A szülő megkapta és elfogta az első jelzést. Amíg a jelzés kezelő az első jelzés feldolgozásával volt elfoglalva, leszállítódott a második jelzés és hozzáadódott a függő jelzések listájához. Mivel azonban a **SIGCHLD**

Programlista 1.31: A **signal1:jelzéskezelő** alprogramja

```
#include "csapp.h"

void handler1(int sig)
{
   pid_t pid;

   if ((pid = waitpid(-1, NULL, 0)) < 0)
     unix_error("waitpid error");
   printf("Handler reaped child %d\n",(int)pid);
   Sleep(2);
  return;
}

int main()
{
   int i, n;
   char buf[MAXBUF];
```

Programlista 1.32: A 1.30 listán látható program végrehajtásának eredménye

```
linux> ./signal1
Hello from child 10320
Hello from child 10321
Hello from child 10322
Handler reaped child 10320
Handler reaped child 10322
<cr>
Parent processing input
```

a **SIGCHLD** jelzés kezelő blokkolja, a második jelzést nem érkezett meg. Röviddel ezután, amikor a jelzés kezelő még mindig az első jelzés feldolgozásával van elfoglalva, megérkezik a harmadik jelzés. Mivel már van egy függő **SIGCHLD**, ez a harmadik **SIGCHLD** jelzés törlődik. Valamivel később, amikor a jelzés kezelő már visszatért, a kernel észreveszi, hogy függőben van egy **SIGCHLD** jelzés és a szülőt a jelzés fogadására készteti. A szülő elfogja a jelzést és másodszor is végrehajtja a jelzés kezelő rutint. Mikor a jelzés kezelő befejezi a második jelzés feldolgozását, nincs több függő **SIGCHLD**

Programlista 1.33: A 1.30 listán látható program hibájának felderítése

```
<ctrl-z>
Suspended
linux> ps
PID TTY STAT TIME COMMAND
...
10319  p5 T  0:03 signal1
10321  p5 Z  0:00 signal1 <defunct>
10323  p5 R  0:00 ps
```

jelzés, és most már soha nem is lesz, mivel a második **SIGCHLD** jelzésre vonatkozó minden ismeret elveszett. Amit ebből nagyon fontos megtanulnunk, hogy a jelzések nem használhatók a folyamatokban események előfordulásának megszámlálására.

A probléma megoldásához idézzük fel, hogy egy függő jelzés létezése csak annyit jelent, hogy legalább egy ilyen jelzés érkezett azóta, hogy a folyamat utoljára ilyen típusú jelzést fogadott. Tehát, úgy kell módosítanunk a SIGCHLD jelzés kezelőt, hogy minden hívás alkalmával begyűjtse az összes lehetséges zombi gyereket.

Programlista 1.34: A 1.30 listán látható program javított változata. Figyelembe veszi hogy a jelzések blokkolódhatnak és hogy nem állnak sorba. Még ez sem veszi azonban figyelembe hogy a rendszerhívások félbeszakíthatók.

```
#include "csapp.h"

void handler2(int sig)
{
    pid_t pid;

    while ((pid = waitpid(-1, NULL, 0)) > 0)
     printf("Handler reaped child %d\n", (int)pid);
    if (errno != ECHILD)
      unix_error("waitpid error");
    Sleep(2);
      return;
}

int main()
```

Az így módosított SIGCHLD jelzés kezelő (lásd 1.34 lista) már megfelelően begyűjti az összes zombi gyereket, lásd 1.35 lista.

Programlista 1.35: A 1.34 listán látható program futtatásának eredménye.

```
linux> ./signal2
Hello from child 10378
Hello from child 10379
Hello from child 10380
Handler reaped child 10379
Handler reaped child 10378
Handler reaped child 10380
<cr>
Parent processing input
```

Ez azonban még nem minden. Ha **signal2** programunkat a Solaris egy régebbi változatán futtatjuk, az helyesen begyűjti valamennyi zombi gyermekét. A blokkolt **read** rendszerhívás azonban még idő előtt visszatér, még mielőtt a billentyűzeten szöveget tudnánk beírni:

Programlista 1.36: A 1.34 listán látható program végrehajtásának eredménye régebbi Solaris rendszeren

```
solaris> ./signal2

Hello from child 18906
Hello from child 18907
Hello from child 18908
Handler reaped child 18906
Handler reaped child 18908
Handler reaped child 18907
read: Interrupted system call
```

A problémát az okozza, hogy ezen a Solaris rendszeren az olyan lassú rendszerhívások mint a **read**, nem indulnak újra, miután megszakítódnak a jelzés érkezése miatt. Helyette –a Linux rendszerektől eltérően, amelyek automatikusan újraindítják a megszakított rendszerhívást– hibajelzéssel visszatérnek a hívó alkalmazáshoz

Ha hordozható jelzés kezelő kódot akarunk írni, számítanunk kell arra a lehetőségre,

Programlista 1.37: A 1.30 listán látható program olyan változata, amelyik helyesen veszi figyelembe hogy a rendszerhívás megszakítható.

```
#include "csapp.h"

void handler2(int sig)
{
  pid_t pid;

  while ((pid = waitpid(-1, NULL, 0)) > 0)
    printf("Handler reaped child %d\n", (int)pid);
  if (errno != ECHILD)
    unix_error("waitpid error");
  Sleep(2);
  return;
}

int main() {
  int i, n;
```

hogy egyes rendszerhívások idő előtt visszatérnek és azokat "kézileg" újra kell indítani, ha ez előfordul. Az 1.37 lista mutatja a **signal2** azon változatát, amelyik manuálisan újraindul abortált rendszerhívások után. Az EINTR visszatérési kód az **errno**-ban jelzi, hogy a **rendszerhívás** a megszakítás után idő előtt tért vissza.

1.5.5. Hordozható jelzés kezelés

A jelzés kezelés kezelés szemantikája rendszerről rendszerre változik – például, hogy egy megszakított lassú rendszerhívás újraindul vagy idő előtt abortál – ez a Unix jelzés kezelés belső ügye. Hogy ezt a problémát megoldja, a Posix szabvány definiálja a **sigaction** függvényt, (lásd 1.38 lista), ami Posix-jellegű rendszereken (mint Linux vagy Solaris) világosan meghatározza a szükséges szemantikát.

Programlista 1.38: A **sigaction** függvény prototípusa

```
#include <signal.h>
int sigaction(int signum, struct sigaction *act,
 struct sigaction *oldact);


    Returns: 0 if OK, -1 on error
```

A **sigaction** függvény használata elég barátságtalan, mert megköveteli, hogy a felhasználó egy struktúra elemeit töltse ki. Nagyon hasznos a **Signal** burkolófüggvényt lásd 1.39

használni helyette, amit a **signal** függvénnyel egyező módon kell hívni. Ez a függvény installál egy jelzés kezelőt, amely a következő jelkezelési szemantikát használja:

- Csak a kezelő által éppen feldolgozás alatt álló típusú jelzések blokkolódnak
- Mint a többi jelzés implementációnál is, a jelzések nem állnak sorba
- A megszakított rendszer hívások automatikusan újraindulnak, ha lehetséges
- Ha már installáltuk a jelzés kezelőt, az mindaddig intallálva is marad, amíg **Signal**t újra nem hívjuk handler argumentumként **SIG_IGN** vagy **SIG_DFL** értékkel.

A **signal2** program (ezt már láttuk a **??** listán) egy olyan változatát mutatja, amelyik **Signal** burkolófüggvényünket használja, hogy előre látható legyen a jelzés kezelés szemantikus viselkedése különböző számítógép rendszereken. Az egyetlen különbség, hogy a kezelő üzembe állításához a **Signal** függvényt hívtuk **signal** helyett. Ez a program már helyesen fut Solaris és Linux rendszereken egyaránt, és nem kell kézzel újraindítani a megszakított rendszerhívásokat.

Programlista 1.39: A **sigaction** függvény hasznos burkoló függvénye

```
handler_t *Signal(int signum, handler_t *handler)
 {
  struct sigaction action, old_action;

  action.sa_handler = handler;
  sigemptyset(&action.sa_mask); /* Block sigs of type
    being handled */
  action.sa_flags = SA_RESTART; /* Restart syscalls if
    possible */

  if (sigaction(signum, &action, &old_action) < 0)
    unix_error("Signal error");
  return (old_action.sa_handler);
}
```

1.5.6. A jelzések blokkolásának kezelése

Az alkalmazások a **sigprocmask** függvény (lásd 1.40 lista) használatával explicit módon is kezelhetik a kiválasztott jelzéseket.

A **sigprocmask** függvény megváltoztatja az éppen blokkolt függvények halmazát. A pontos viselkedés a **how** értékétől függ:

- *SIG_BLOCK*: Hozzáadja a **set**ben felsorolt függvényeket **blocked**hez (**blocked** = **blocked** | **set**).
- *SIG_UNBLOCK*: Eltávolítja a **set**ben felsorolt függvényeket **blocked**ből (**blocked** = **blocked** & **set**).
- *SIG_SETMASK*: **blocked** = **set**

Ha **oldset** nem **NULL** értékű, a **blocked** bit vektor előző értékét eltárolja **oldset**ben.

A **set** jelzéseket a következő függvényekkel manipulálhatjuk. A **sigemptyset** üres halmazzá inicializálja a **set**et. A **sigfillset** függvény hozzáadja az egyes jelzéseket **set**hez. A **sigaddset** függvény a **signum** jelzést hozzáadja **set**hez, a **sigdelset** törli **signum**ot **set**ből, végül **sigismember** 1 értéket ad vissza, ha **signum** tagja **set**nek, és 0 értéket, ha nem.

Programlista 1.40: A jelzések blokkolását kezelő függvények prototípusa

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *set,
  sigset_t *oldset);
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
      //Returns: 0 if OK, -1 on error

int sigismember(const sigset_t *set, int signum);
      // Returns: 1 if member, 0 if not, -1 on error
```

1.5.7. Folyamatok szinkronizálása konkurrencia hibák elkerülésével

Az a probléma, hogy hogyan kell programozni két olyan kód folyamot, amelyek ugyanazt a tárhelyet írják és olvassák, számítógép tudósok generációit foglalkoztatta. Általában véve, a kód folyamok egymásban ágyazódásának lehetősége az utasítások számával exponenciálisan nő. Bizonyos beágyazódások esetén az eredmény helyes lesz, mások esetén meg nem. Az alapvető probléma a konkurrens kód folyamok egy olyan szinkronizálása, hogy legtöbb féle egymásba ágyazódást engedjük meg, és a lehetséges beágyazódások mindegyike helyes eredményt adjon.

A konkurrens programozás mély és fontos probléma, amelyet itt most nem tárgyalunk. A most tanultakat azonban felhasználhatjuk arra, hogy érzékeltessük a konkurrenciából fakadó intellektuális kihívásokat. Tekintsük például a 1.41 listán mutatott progamot, amely egy tipikus Unix parancs értelmező szerkezetével rendelkezik. A szülő folyamat egy listán követi nyomon a gyermek folyamatai sorsát, minden folyamathoz egy bejegyzést rendelve. Az **addjob** és **deletejob** függvények ehhez a listához hozzáadnak egy bejegyzést vagy kivesznek belőle.

Miután a szülő folyamat létrehozta az új gyermek folyamatot, hozzáadja a gyermek

Programlista 1.41: Egy utasítás értelmező program egy ravasz szinkronizálási hibával. Ha a gyermek folyamat az előtt befejeződik hogy a szülő folyamat el tud indulni akkor az **addjob** és **deletejob** függvények helytelen sorrendben hívódnak.

```
void handler(int sig)
{ (3)
  pid_t pid;
  while ((pid=waitpid(-1,NULL,0))>0) //Reap a zombie
    child
  deletejob(pid); // Delete the child from the job list (4)
  if (errno != ECHILD)
    unix_error("waitpid error");
}

int main(int argc, char **argv)
{
  int pid;
  Signal(SIGCHLD, handler);
  initjobs(); /* Initialize the job list */
```

folyamatot a listához. Amikor a szülő folyamat begyűjti a befejeződött (zombi) gyermek folyamatot a SIGCHLD jelzés kezelőben, törli a folyamatot a listáról. Első pillantásra, a kód helyesnek tűnik. Sajnos, a következő esemény sorozat is lehetséges:

1. A szülő folyamat végrehajtja a **fork** függvényt, és a kernel az újonnan létrehozott gyermek folyamatot ütemezi be futtatásra a szülő folyamat helyett.
2. Mielőtt a szülő folyamat ismét futásképes lenne, a gyermek folyamat befejeződik és zombi lesz belőle, aminek hatására a kernel SIGCHLD jelzést küld a kernelnek.
3. Később, amikor a szülő folyamat ismét futáskésszé válik, de még nem hajtódott végre, a kernel észreveszi a függő SIGCHLD jelzést és a jelzés kezelő végrehajtásával a a szülő folyamatot a jelzés fogadására kényszeríti.
4. A jelzés kezelő begyűjti a befejeződött folyamatot és meghívja a **deletejob** függvényt, amit semmit sem csinál, mivel a szülő folyamat még nem adta hozzá a gyermek folyamatot a listához.
5. Miután a kezelő befejeződik, a kernel futtatja a szülő folyamamatot, amelyik visszatér a **fork**ból és az **addjob** függvény használatával hibásan hozzáadja a (nem létező) gyermek folyamatot a listához.

Azaz, a főprogram és a jelzés kezelő kódfolyamának bizonyos beékelődései esetén

előfordulhat, hogy **deletejob** előbb hívódik meg, mint **addjob**. Ennek eredményeként hibás bejegyzés jön létre a listában, egy olyan folyamatról, amelyik már nem létezik és amely bejegyzés soha nem fog eltávolítódni. Másrészt vannak olyan beékelődések is, amelyek előfordulása esetén az események a helyes sorrendben történnek meg. Például, ha a kernel a **fork** rutinból való visszatérés után a szülő folyamatot ütemezi be a gyermek helyett, akkor a szülő helyesen adja hozzá a gyermek folyamatot a listához, mielőtt a gyermek befejeződne és a jelzés kezelő eltávolítaná a folyamatot a listáról.

Ez a versenyhelyzet (race) néven ismert klasszikus szinkronizálási hiba. Ebben az esetben két versenyző a főprogrambeli **addjob** és a jelzéskezelőbeli **deletejob**. Ha az **addjob** nyeri a versenyt, akkor helyes az eredmény; ha nem, akkor helytelen. Az ilyen típusú hibákat roppant nehéz felderíteni, mivel nem tudunk minden lehetséges beékelődést megvizsgálni. Előfordulhat, hogy milliószor lefut a program minden probléma nélkül, aztán a következő esetben előáll a verseny.

1.6. Nem-lokális ugrások

A **C** nyelvben van egy felhasználói szintű kivételes vezérlő folyam, amit nem-lokális ugrásnak neveznek, és amely a vezérlést közvetlenül átadja az egyik függvényből egy másik, éppen végrehajtás alatt levő függvénybe, anélkül, hogy végig kellene menni a szokásos **call** és **return** utasítás sorozaton. A nem-lokális ugrásokat a **setjmp** és **longjmp** függvények szolgáltatják, lásd 1.42 programlista.

Programlista 1.42: A **setjmp** függvény prototípusa

```
#include <setjmp.h>

int setjmp(jmp_buf env);
int sigsetjmp(sigjmp_buf env, int savesigs);

    Returns: 0 from setjmp, nonzero from longjmps
```

A **setjmp** függvény elmenti a hívási környezetet az **env** pufferben, hogy azt később

a **longjmp** felhasználja, majd **0** értékkel visszatér. A hívási környezet tartalmazza a programszámlálót, a veremmutatót, és az általános célú regisztereket.

A **longjmp** függvény, lásd 1.43 programlista, az **env** pufferből visszaállítja a hívási környezetet, majd kivált egy visszatérést a legutóbbi **setjmp** hívásból, amely hívás az **env** környezetnek értéket adott. Ezután **setjmp** egy nem-nulla **retval** értékkel tér vissza.

A **setjmp** és **longjmp** közötti együttműködés első pillantásra kissé zavaros. A **setjmp** függvényt egyszer hívjuk, de többször is visszatér: egyszer akkor, amikor a **setjmp** függvényt először hívjuk, és a hívási környezetet eltároljuk az **env** pufferben, és egyszer valamennyi megfelelő **longjmp** hívás esetén. Másrészt pedig a **longjmp** függvényt egyszer hívjuk, és sosem tér vissza.

A nem-lokális ugrás fontos alkalmazása, hogy lehetővé tesz közvetlen visszatérést mélyen beágyazott függvényhívásból, általában valamilyen hiba feltétel észlelésekor. Ha egy mélyen beágyazott függvényhívás során hibafeltételt észlelünk, a nem-lokális ugrással közvetlenül visszatérhetünk a lokalizált hibakezelőhöz, és nem kell nehéz munkával feldolgozni a hívási vermet.

Az 1.44 programlista mutat példát arra, ezt hogyan lehet használni. A fő rutin először meghívja a **setjmp** függvényt , hogy elmentse a hívási környezetet, majd meghívja a

Programlista 1.43: A **longjmp** függvény prototípusa

```
#include <setjmp.h>

void longjmp(jmp_buf env, int retval);
void siglongjmp(sigjmp_buf env, int retval);

        Never returns
```

foo függvényt (2), ami meghívja a **bar** függvényt (3). Ha **foo** vagy **bar** hibát észlel, azonnal visszatér a **setjmp**-ból a **longjmp** hívása útján. A **setjmp** nem-nulla visszatérési értéke jelzi a hiba típusát, amit aztán dekódolhatunk és a kód egyazon helyén kezelhetünk.

A nem-lokális ugrások egy másik fontos alkalmazási területe, amikor egy jelzés kezelőből egy bizonyos helyre kell ugrani, ahelyett, hogy abba a kód részbe térnénk vissza, amit a jelzés érkezése megszakított. Az 1.45 programlista egy olyan egyszerű programot mutat, amelyik ezt az alap-technikát használja. Ez a program jelzéseket és nem-lokális ugrásokat használ arra, hogy "puha" újraindulást (soft restart) hajtson

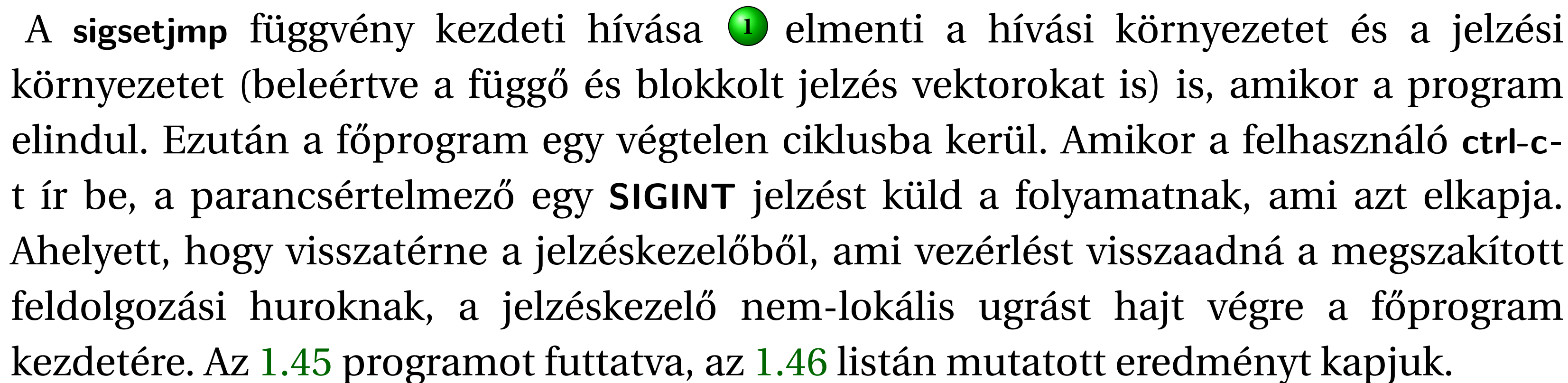
Programlista 1.44: **Nem-lokális ugrás példa program.** Azt mutatja be, hogyan lehet újraindulni egy mélyen beágyazott függvényben észlelt hiba után, a hívási verem teljes felgöngyölítése nélkül.

```
#include "csapp.h"

jmp_buf buf;

int error1 = 0;
int error2 = 1;

void foo(void), bar(void);

int main()
{
    int rc;

    rc = setjmp(buf); (1)
    if (rc == 0)
```

Programlista 1.45: Egy olyan program, amelyik újraindítja magát, amikor a felhasználó **ctrl-c**-t ír be.

```
#include "csapp.h"

sigjmp_buf buf;

void handler(int sig) (2)
{
    siglongjmp(buf, 1);
}

int main()
{
    Signal(SIGINT, handler);

    if (!sigsetjmp(buf, 1)) (1)
      printf("starting\n");
    else
```

végre, amikor a felhasználó **ctrl-c**-t üt be a billentyűzeten. A **sigsetjmp** és a **siglongjmp** függvények a **setjmp** és **longjmp** olyan változatai, amelyeket a jelzéskezelők használnak.



A **sigsetjmp** függvény kezdeti hívása (1) elmenti a hívási környezetet és a jelzési környezetet (beleértve a függő és blokkolt jelzés vektorokat is) is, amikor a program elindul. Ezután a főprogram egy végtelen ciklusba kerül. Amikor a felhasználó **ctrl-c**-t ír be, a parancsértelmező egy **SIGINT** jelzést küld a folyamatnak, ami azt elkapja. Ahelyett, hogy visszatérne a jelzéskezelőből, ami vezérlést visszaadná a megszakított feldolgozási huroknak, a jelzéskezelő nem-lokális ugrást hajt végre a főprogram kezdetére. Az 1.45 programot futtatva, az 1.46 listán mutatott eredményt kapjuk.

Megjegyzés: A C++ és Java szoftveres kivételkezelése

A C++ és Java kivétel kezelési mechanizmusa lényegében a C **setjmp** és **longjmp** függvényeinek strukturáltabb változata. Egy **try** belsejében levő **catch** tekinthető a **setjmp** megfelelőjének. Hasonlóképpen, a **throw** utasítás a **longjmp** függvényre hasonlít.

Programlista 1.46: Az 1.45 listán látható program végrehajtásának eredménye

```
unix> ./restart
starting
processing...
processing...
restarting User hits ctrl-c
processing...
restarting User hits ctrl-c
processing...
```

1.7. Segédprogramok a folyamatok kezelésére

A Linux rendszerek számos hasznos segédprogramot biztosítanak a folyamatok vizsgálatára és befolyásolására:

strace Nyomon követi a futó program és annak gyermekei által végrehajtott rendszerhívásokat. Nagyszerű eszköz egy kíváncsi hallgató számára. Érdemes a programot a **-static** kapcsolóval fordítani, akkor a kimenet sokkal érhetőbb, nem tartalmazza az osztott könyvtárakra vonakozó információt.

ps Felsorolja az aktuálisan a rendszerben levő folyamatokat (a zombikat is).

top Az aktuális folyamatok erőforrás használatát nyomtatja ki.

pmap Kijelzi a folyamatok memória térképét.

proc Egy olyan virtuális fájlrendszer, amelyik számos kernel adatszerkezetet exportál ASCII formában, így az a felhasználói programok számára is olvasható. Például, a“**cat /proc/loadavg**” begépelésével megnézhetjük Linux rendszerünk aktuális átlagos terhelését.

1.8. Összefoglalás

A kivételes utasítás végrehajtási folyam (Exceptional control flow, ECF) a számítógépes rendszerek minden szintjén jelen van, és nélkülözhetetlen a számítógépes rendszeren belüli konkurrencia megvalósításához.

Hardver szinten, a kivételek a vezérlő folyam hirtelen változásai, amelyeket a processzorban történő események váltanak ki. A vezérlési folyam áttevődik egy szoftveres kezelő rutinba, amelyik valamilyen feldolgozást végez, majd visszaadja a vezérlést a megszakított vezérlési folyamba.

Négyféle kivétel létezik: megszakítások, hibák, abortálások és cspdák. A megszakítások aszinkron módon történnek (bármelyik utasításhoz képest), amikor egy külső I/O eszköz (mint egy időzítő áramkör vagy egy mágneslemez vezérlő) a processzor megszakítás tüskéjére jelzést ad. A vezérlés a hibás utasítást követő utasításra kerül vissza. A hibák és abortálások egy utasítás végrehajtásának eredményeként, szinkron módon történnek. A hiba kezelők újraindítják a hibás utasítást, míg az abortálás kezelők soha nem térnek vissza a megszakított kód folyamhoz. Végezetül, a csapdák függvény híváshoz hasonlítanak, amelyekkel rendszer hívásokat valósítunk meg, ezek lényegében

az alkalmazások számára az operációs rendszerbe biztosítanak jól ellenőrzött belépési pontokat.

Operációs rendszer szinten a kernel ECF-et használ a folyamatok alap jellemzőinek megvalósítására. A folyamatok két alapvető absztrakciót biztosítanak az alkalmazások számára: (1) olyan logikai vezérlési folyamot, amelyik mindegyik programnak biztosítja azt az illúziót, hogy kizárólagosan rendelkezik a processzorral, és (2) egy olyan magán címteret, amely azt az illúziót nyújtja, hogy minden egyes program kizárólagos joggal használja a fő memóriát.

Az operációs rendszer és az alkalmazások közötti köztes rétegben, az alkalmazások gyermek folyamatokat kelthetnek; megvárhatják, amíg a gyermek folyamatok megállnak vagy befejeződnek; új programokat futtathatnak; és elkaphatnak más folyamatoktól származó jelzéseket. A jelzés kezelés szemantikája nagyon érzékeny és rendszerről rendszerre változik. A Posix-szabványú rendszereken azonban vannak olyan mechanizmusok, amelyek lehetővé teszik a programok számára, hogy világosan meghatározzák a jelzések kezelésének elvárt szematikáját.

Végezetül, alkalmazási szinten, a C programok is használhatnak nem-lokális ugrásokat a szokásos hívás/visszatérés módszer kikerülésére, hogy közvetlenül az egyik

függvényből a másikba ugorjanak.

Táblázatok jegyzéke

1.1. A kivételek osztályozása. Az aszinkron kivételek a processzoron kívül eső, I/O eszközök által előállított események következtében jönnek létre. A szinkron események egy utasítás végrehajtásának közvetlen következményei 18

1.2. Példák kivételekre IA32 rendszerekben 28

1.3. Néhány gyakrabban használt rendszerhívás Linux/IA32 rendszerekben. Forrás: /usr/include/sys/syscall.h. 29

Ábrák jegyzéke

1.1. Egy kivétel anatómiája. A processzor egy állapotváltozása (esemény) egy hirtelen vezérlésátadást (egy kivételt) vált ki az alkalmazástól a kivétel kezelőhöz. Befejeződése után a kezelő a vezérlést vagy visszaadja a megszakított programnak vagy abortál. 10

1.2. A kivétel kezelő táblázat: egy ugrási táblázat, amelyben a *k* elem tartalmazza a *k* kivétel kezelőjének címét. 13
1.3. A kivétel kezelő eljárás címének előállítása. A kivétel száma indexeli a kivétel kezelő táblázatot. 14
1.4. Megszakítás kezelés. A megszakítás kezelő a vezérlést az alkalmazói program folyam következő utasítására adja vissza. 19
1.5. Csapda kezelés. A csapda kezelő a vezérlést az alkalmazói programfolyam következő utasítására adja vissza. 21
1.6. Hiba kezelés. Attól függően, hogy a hiba javítható-e vagy nem, a hiba kezelő vagy újból végrehajtja a hibás utasítást, vagy abortál. 23
1.7. Abortálás kezelés. Az abortálás kezelő a vezérlést a kernel abort rutinjába viszi át és befejezi az alkalmazói programot. 24
1.8. Logikai vezérlési folyam. A folyamatok minden programnak biztosítják azt az illúziót, hogy kizárólagosan használja a processzort. Az egyes függőleges vonalak az egyes folyamatok logikai vezérlési folyamának egy részét ábrázolják. 37
1.9. A folyamat saját címtere 42

1.10. Egy folyamat környezet átkapcsolásának anatómiája. 47
1.11. Az argumentum lista adatszerkezete 66
1.12. A környezeti változók lista adatszerkezete 67
1.13. A felhasználó veremtárolójának szerkezete egy új program elindulásakor . . 69
1.14. Jelzés kezelés. Egy jelzés fogadása kiváltja a vezérlés átadását a jelzés kezelőbe. Miután befejezte működését, a kezelő visszaadja a vezérlést a megszakított programnak. 84
1.15. Előtérbeli és háttérbeli folyamat csoportok 90

Programlisták

1.1. A "Hello Világ" program forráskódja rendszerhívásokkal 31
1.2. A "Hello Világ" rendszerhívásos változatának assembly nyelvű kódja . . . 32
1.3. A **fork** függvény hívása hibaellenőrzéssel 50
1.4. A **fork** függvény hívása hibajelentő függvénnyel 51
1.5. A **fork** hibajelentő burkoló függvénye 52

1.6. A **fork** használata burkolófüggvénnyel és hibavizsgálattal 52
1.7. A **GetPID** rendszerhívás prototípusa 54
1.8. Az **exit** rendszerhívás prototípusa 55
1.9. A **fork** rendszerhívás prototípusa 57
1.10. Új folyamat létrehozása a fork rendszerhívás használatával. A futtatás eredményét a 1.11 lista mutatja 59
1.11. A 1.10 listán látható program végrehajtásának eredménye 60
1.12. Kétszeresen elágazó fork függvény 60
1.13. A **waitpid** függvény használata 63
1.14. A **sleep** függvény prototípusa 64
1.15. Az **execve** függvény prototípusa 65
1.16. A **getenv** függvény prototípusa 71
1.17. A **setenv** függvény használata 72
1.18. Az egyszerű parancs értelmező **main** rutinja 74
1.19. Az egyszerű parancs értelmező 75
1.20. Az egyszerű parancs értelmező beépített parancsa 77
1.21. Egy parancssor értelmezése a parancs értelmező számára 79

1.22. A **getpgrp** függvény használata 86
1.23. A **setpgrp** függvény használata 87
1.24. A **kill** függvény használata 91
1.25. Hogyan használjuk a **kill** függvényt arra, hogy jelzést küldjünk egy gyermek folyamatnak 93
1.26. Az **alarm** függvény prototípusa 94
1.27. Példa az **alarm** függvény használatára 95
1.28. A **signal** függvény prototípusa 97
1.29. Hogyan lehet a **SIGINT** jelzést a jelzéskezelő függvénnyel elkapni. 99
1.30. **signal1: főprogram** Ez a program azért hibás mert nem foglalkozik azzal hogy egy jelzés blokkolva is lehet; a jelzések nem íródnak be a sorba; és a rendszerhívások megszakíthatók. 103
1.31. A **signal1:jelzéskezelő** alprogramja 105
1.32. A 1.30 listán látható program végrehajtásának eredménye 106
1.33. A 1.30 listán látható program hibájának felderítése 107

1.34. A 1.30 listán látható program javított változata. Figyelembe veszi hogy a jelzések blokkolódhatnak és hogy nem állnak sorba. Még ez sem veszi azonban figyelembe hogy a rendszerhívások félbeszakíthatók. 108
1.35. A 1.34 listán látható program futtatásának eredménye. 109
1.36. A 1.34 listán látható program végrehajtásának eredménye régebbi Solaris rendszeren 110
1.37. A 1.30 listán látható program olyan változata, amelyik helyesen veszi figyelembe hogy a rendszerhívás megszakítható. 111
1.38. A **sigaction** függvény prototípusa 113
1.39. A **sigaction** függvény hasznos burkoló függvénye 115
1.40. A jelzések blokkolását kezelő függvények prototípusa 117
1.41. Egy utasítás értelmező program egy ravasz szinkronizálási hibával. Ha a gyermek folyamat az előtt befejeződik hogy a szülő folyamat el tud indulni akkor az **addjob** és **deletejob** függvények helytelen sorrendben hívódnak. 119
1.42. A **setjmp** függvény prototípusa 122
1.43. A **longjmp** függvény prototípusa 124

1.44. **Nem-lokális ugrás példa program.** Azt mutatja be, hogyan lehet újraindulni egy mélyen beágyazott függvényben észlelt hiba után, a hívási verem teljes felgöngyölítése nélkül. 125
1.45. Egy olyan program, amelyik újraindítja magát, amikor a felhasználó **ctrl-c**-t ír be. 126
1.46. Az 1.45 listán látható program végrehajtásának eredménye 128