AEM: saját méretezhető eseménykezelő Linux alá

Tegyük képessé alkalmazásunkat rendszermag visszahívások regisztrálására.

Egy korábbi cikkünkben a telekommunikáció témakörében felvetettük egy saját, általános eseménymechanizmus szükségességét ***Linux*** alá. A ***Linux*** képességeinek javítását célzó legtöbb megoldás kiábrándított bennünket. Mások nem feleltek meg igényeinknek, hiszen egy kommunikációs szintű (carrier-grade) rendszernek komolyabb valós idejű követelményei vannak. A siker érdekében egy ilyen eseménymechanizmusnak szorosan kell kapcsolódnia az otthont adó operációs rendszerhez és ki kell tudnia használni annak képességeit a nagyobb teljesítmény érdekében.

A montreáli ***Open Systems*** laborban (***Ericsson Kutatóintézet***), 2001-ben indítottuk az általános megoldást kereső ***Asynchronous Event Mechanism (AEM)*** projektet. Az ***AEM*** segítségével az alkalmazás bizonyos eseményekhez visszahívható függvényeket adhat meg illetve regisztrálhat, az operációs rendszer pedig az események létrejöttekor aszinkron módon hajtja végre ezeket.

Az ***AEM*** a fejlesztéshez esemény elvű megközelítést kínál. Ezt egy természetes felhasználói csatolófelület bevezetésével értük el, ahol az eseménykezelők paraméterlistájában a végrehajtásukhoz szükséges és a rendszermagnak közvetlenül elküldendő valamennyi adat megtalálható.

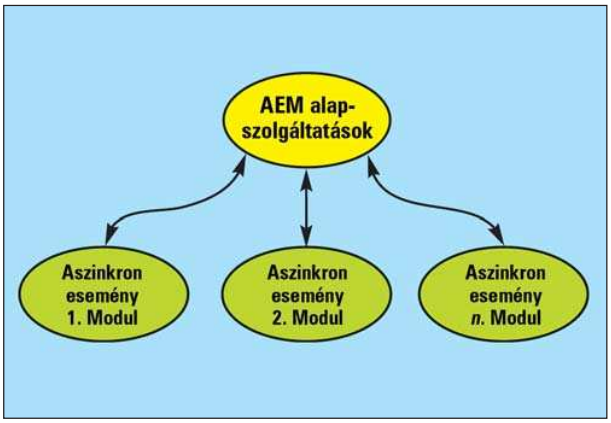
Az ***AEM*** másik mozgatórugója az a tény adta, hogy a többszálú rendszereken futó összetett megoszló alkalmazásokat a kezelő réteg miatt általában igen nehéz több rendszerre fejleszteni vagy azokra átírni. Az ***AEM*** nem pusztán a programfejlesztési idő csökkentését célozza meg, hanem a forráskód létrehozását is leegyszerűsíti, ezáltal megnövelve a különböző rendszerek közti a hordozhatóságot és meghosszabbítva a program életciklusát.

A projekt legnagyobb kihívása egy olyan rugalmas keretrendszer tervezése és kifejlesztése volt ahol a futó rendszerben eseménykezelő megoldásokat tudunk frissíteni és elhelyezni. Alapkövetelmény volt, hogy a rendszerkarbantartás a rendszer újraindítása nélkül is elvégezhető legyen. Az ***AEM*** moduláris felépítésének köszönhetően rendelkezik ezzel a képességgel.

Az ***AEM*** más figyelmeztető mechanizmusok mellett kiegészítő lehetőségként működik. Az egyik nagy előnye, hogy az eseményvezérlet kódot keverhetjük a többi, soros kóddal.

AEM: Szerkezeti áttekintés

Az ***AEM*** magmodulból és a köré épülő betölthető modulokból áll amelyek az adott eseményszolgáltatást nyújtják az alkalmazásoknak. Ilyen például a szoftveres időzítés és az aszinkron ***TCP/IP*** socket csatolófelület (*1. ábra*). Ez a rugalmas szerkezet lehetővé teszi, az AEM képességeinek tetszés szerinti bővítését.



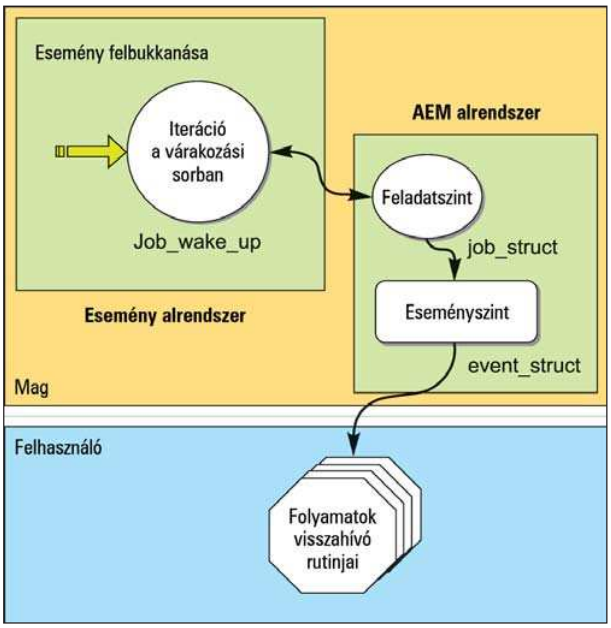
1. ábra Az AEM az alapvető eseményfunkciókat biztosító magmodulból és az alkalmazásoknak aszinkron eseményszolgáltatásokat nyújtó független rendszermag modulokból áll

A modulok által végrehajtott feladatokat száma korlátlan, ugyanis saját, független pszeudo rendszerhívás készletüket adják át az alkalmazásnak. Tulajdonképpen így akár két különböző modul is szolgálhat azonos feladatokat. Érdekes módon éppen ez teszi lehetővé, hogy a továbbfejlesztett változatot a többi alkalmazás zavarása nélkül a rendszerbe illesszük – azok ugyanis továbbra is a régi verziót használják. A felépítés lehetővé teszi, hogy az alkalmazások igénye szerint töltsük be az ***AEM*** modulokat vagy futásidőben fejlesszük a modulokat.

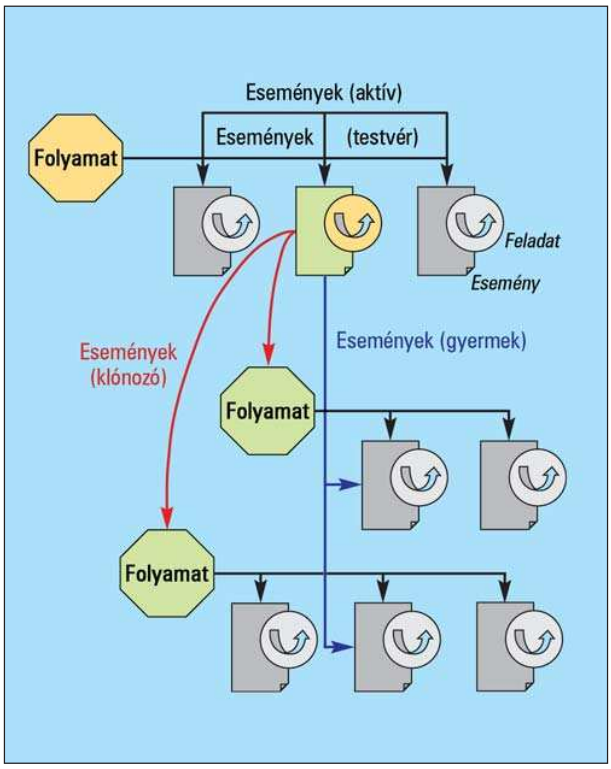
Az ilyesfajta rugalmasság előfeltétele, hogy a rendszermag stratégiai pontján eseményaktiválási pontokat helyezzünk el (*2. ábra*). Minden aktiválási pont egy-egy eseményeket indító AEM sornak felel meg. A következő részben az ***AEM*** belső működését részletezzük.

AEM: Belső szerkezet

Az aszinkronitás elve akkor okoz komoly gondot, amikor a program végrehajtás fő folyama az eseménykezelők alkalmazásakor figyelmeztetés nélkül megszakad. A bemenetei kérelmeket a korábbi bemeneti állapot nélkül kell kezelni.



2. ábra Az AEM esemény aktiválási és alkalmazás-értesítési szerkezete. Az alvó munkafolyamatokat tartalmazó esemény-várakozási sorokat folyamatosan pásztázzuk. Az adott esemény bekövetkeztekor valamennyi érintett feladat felébred. Ezt követően valamennyi ide tartozó folyamathoz rendelt eseményt elindítjuk.



3. ábra A folyamatok és eseménylistáik közötti kapcsolat

Ezeket közvetlenül a rendszermag vagy a megszakítás-kezelő küldi és fogadásukkor nem feltételezhetünk semmilyen sorrendet. Ez a helyzet bizonyos alkalmazásoknál problémákat okozhat, különösen amelyek *TCP/IP* alapon működnek, hiszen ezek működéséhez fontos ismerni a tranzakció állapotát.

Az *AEM* három rétegű szerkezetét az eseménykezelést végző pseudo-rendszer hívások, az esemény sorosítást kezelő és a felhasználói visszahívások érdekében környezetinformációkat tároló folyamatonkénti `event_struct` struktúrák valamint az eseményaktiválásért felelős eseményenkénti `job_struct` szerkezetek alkotják.

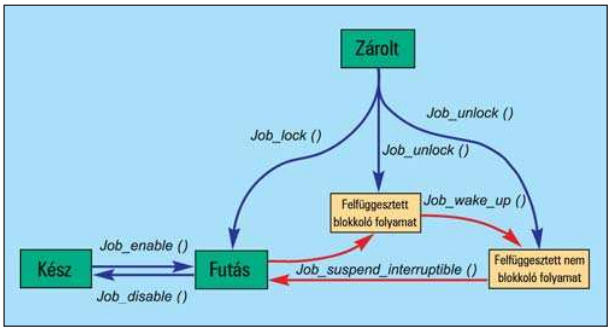
Események

Az *AEM* szemszögéből nézve, az esemény olyan rendszer inger, amely kiváltja az esemény kezelője, azaz a végrehajtó ügynök létrehozását. Ehhez az esemény regisztrációjakor létrehozott `event_struct` struktúra nyújt segítséget, amely egy esemény kezelő végrehajtásához szükséges környezetet tartalmazza. A legfontosabb mezői a felhasználói tér eseménykezelőire hivatkozó mutatók, azok konstruktorai és destruktorai valamint kapcsolatuk más eseményekkel (listaesemények, gyermek események és aktív események – lásd a *3. ábrát*). Folyamatonként annyi regisztrált eseménykezelőt készíthetünk amennyi csak szükséges. Amikor az eseményt észleljük, azt aktiválódásnak nevezzük és hamarosan meghívjuk a felhasználó által megadott kezelő függvényt.

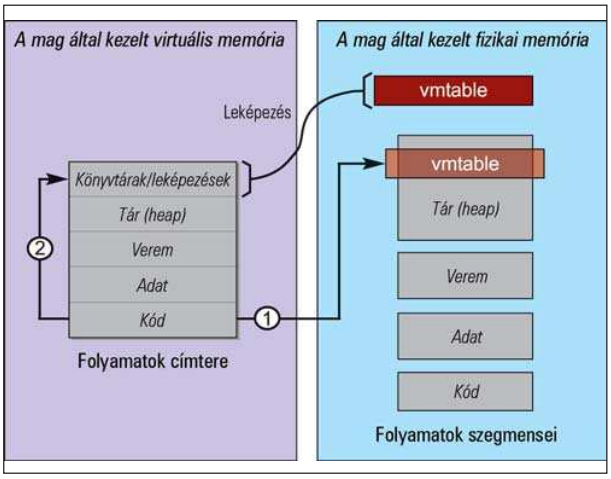
Az események érkezési sorrendben jegyződnek fel a rendszerben. Innentől fogva már az eseménykezelő konstruktorának feladata az adatokat helyesen kezelni és bármiféle sorrendiség feltételezése nélkül sorosítani az egyes folyamatokhoz tartozó eseményeket.

Bizonyos folyamatesemények aktívak és aktív eseménylistához csatlakoznak. Aktiváláskor az esemény folyamatot hozhat létre. Ezeket az eseményeket klónozóknak nevezik. Az események és az általuk létrehozott folyamatok közti kapcsolatot belsőleg tároljuk. A harmadik ábrán megfigyelhetjük a legfelső folyamat által regisztrált eseményt amely két új folyamatot készített. Ezek az eseményhez kapcsolva maradnak és saját eseménylistával rendelkeznek.

Az eseménykezelőket az esemény regisztrálásakor használjuk és a felhasználói szinten kell megvalósítani őket. Megadhatják saját, előre meghatározott számú paramétereiket, hogy az esemény adatok közvetlenül a felhasználói térben futó folyamathoz kerülhessenek. Ezt a műveletet az esemény konstruktorok és destruktorok végzik amelyeket közvetlenül a kezelők előtt és után hívunk meg.



4. ábra Periodikus és reaktív munkafolyamatok állapotátmeneti grafikonja



5. ábra A vmtable felépítése

Az eseménykezelők ugyanabban a környezetben futnak mint a meghívó folyamat. A mechanizmus biztonságos és többszörösen bejárható (re-entrant); a jelenlegi végrehajtás mentésre kerül, végül helyreállítjuk a megszakítás előtti állapotot.

A regisztráció során minden eseményhez valamilyen fontossági szintet rendelünk amely a megfelel annak a sebességnek mellyel az alkalmazás az adott figyelmeztetést fogadni kívánja. Ugyanazon eseményre két különböző fontossági szinten is jelentkezhetünk.

Más valós idejű figyelmeztetési megoldások, például a ***POSIX*** jelzések valós idejű kiterjesztése, az ütemezési döntés során nem veszik figyelemebe a fontossági szinteket. Pedig ez fontos, hiszen így a nagy fontosságú eseményeket fogadó folyamatokat a többi folyamat előtt tudjuk végrehajtani. ***AEM*** alatt az esemény megjelenése a fontossági sorrendnek megfelelően váltja ki a kezelő végrehajtását. Bizonyos szemszögből, az eseménykezelő folyamatnak tekinthető, hiszen van végrehajtási környezete. A folyamat fontosságok dinamikus változtatása akkor okoz igazán komoly gondot, amikor az események beérkezési sűrűsége nagy, hiszen a fontossági szintek ugyanilyen ütemben változnak. Ezt a problémát az esemény fontosságok összességéből számított dinamikus valós idejű érték bevezetésével oldottuk meg. Ez az érték a ***Linux*** ütemező befolyásolása nélkül módosítja az ütemezési döntéseket az alkalmazásoknak pedig valós idejű érzékenységet ad.

Munkafolyamatok

A ***munkafolyamat*** (***job***) egy új rendszermag fogalom amelyet az események figyelmeztető folyamatokat megelőző kiszolgálása érdekében hoztak létre. Ez nem folyamat, bár mind a kettő a végrehajtható elem elképzelésen alapul. A munkafolyamatok által végzett egyik tipikus feladat, hogy elhelyezi magát egy várakozási sorban és itt marad, míg valami fel nem ébreszti. Ekkor gyorsan elvégez valami hasznos dolgot, például ellenőrzi az adat hitelességét vagy elérhetőségét a felhasználói esemény meghívása előtt, majd ismét elalszik. A munkafolyamat azt is garantálja, hogy mialatt valamilyen erőforrást elér, más munkafolyamatok azt nem érhetik el. Egy folyamathoz több munkát is rendelhetünk, de eseményenként csak egy munkafolyamatunk lehet.

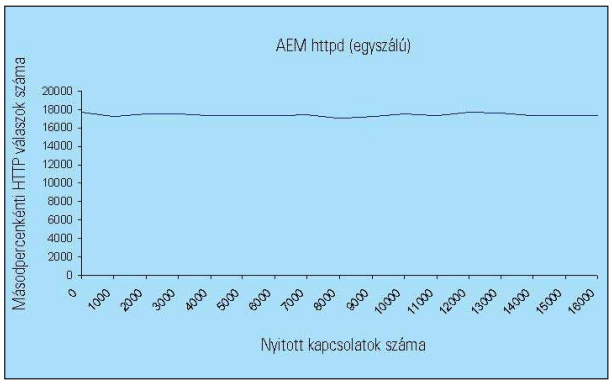
Ez a rendszermag és a felhasználói folyamat közötti elvonatkoztatási réteg nélkülözhetetlen. Nélküle igen nehéz lenne biztosítani a konzisztens adat elérhetőség ellenőrzést vagy a kezelő végrehajtása során azonos eseményekből többet felhalmozni. Amennyiben valami probléma adódik, a folyamat a felhasználói térben pazarolja az esemény kezelésére szánt időt. Az, hogy összevonunk-e több figyelmeztetést általában eseménytől függő és az aktiválás előtt kell végrehajtanunk.

A munkafolyamatok általános megvalósításánál figyelembe kell vennünk a megszakításokat, hogy az esemény bekövetkezése és a folyamat figyelmeztetése közt lehetőleg kevés idő teljen el. Célunk, hogy a folyamatok oldalán végezzük el a műveleteket miközben egy megszakításkezelő és egy rendszermag szál képességeit biztosítjuk, mindezt anélkül, hogy a teljes végrehajtási környezetet magunkkal hurcolnánk.

Két típusú munkafolyamatot alkalmaztunk, a periodikus és a reaktív munkafolyamatokat. A periodikus munkafolyamatokat adott időnként, a reaktív munkafolyamatokat szórványosan, az események észlelésekor hajtjuk végre. A munkafolyamatokat saját ütemező ütemezi. A valós idejű ütemezés elméletet szerint mind a két fajta munkafolyamatot azonos ütemezőnek kell ütemeznie (lásd ***Jeffay és társai*** publikációját a hálózati forrásokban). A mi környezetünkben a munkafolyamat nem preemptív feladat. definíció szerint a feladatoknak nincs megadott határidejűk, bár alacsony szintű jellegükből következően végrehajtási idejük közvetetten mégis kötött. Ez a feltételezés leegyszerűsíti a megvalósítást. Esetünkben előfeltétel, hogy a reagáló munkafolyamatok két meghívás között elhanyagolható idő alatt hajtódjanak végre hogy helyt álljanak folyamátviteli helyzetekben is.

A mi megoldásunk a reagáló és a periodikus esetben is eltérő, hogy a szórványos események esetében jobb átviteli eredményt érjünk el.

A periodikus munkafolyamatokat a ***feladatkiosztó*** (***job dispatcher***) kezeli, a reagáló feladatok ugyanakkor teljesítmény megfontolások miatt saját maguk állítják állapotukat. A *4. ábra* mutatja be a munkafolyamat állapotváltozásokat és az egyik állapotból másikra való áttéréshez használt függvényeket.



6. ábra AEMhttpd, egy szálú HTTP kiszolgáló amelyet az AEM belső megvalósításának skálázhatósági tesztjénél használtunk. Minden mintában 100 aktív kapcsolatot használtunk.

Amint a munkafolyamat aktivizálta a megfelelő eseményt, vagy aszinkron módon végrehajtunk egy folyamatot vagy a felhasználói program jelenlegi futását irányítjuk át a kód másik részére. A felhasználó regisztráláskor dönti el melyik változatot szeretné alkalmazni.

Folyamat aszinkron végrehajtása

Az eseménykezelők végrehajtásához megtörhetjük a fő szál végrehajtását vagy létrehozhatunk egy új folyamatot amelyben az esemény párhuzamosan futhat le. Az esemény mindkét esetben átlátszóan és aszinkron módon fut, és közvetlenül önállóan felügyeli saját futó példányait. Ennek a megoldásnak van néhány fontos következménye ugyanis az alkalmazásnak nem kell idő előtt megőriznie vagy elhasználnia a rendszer erőforrásait. Próbaképpen az ***AEM*** teljesítményméréséhez készítettünk egy egyszerű ***HTTP*** kiszolgálót, amely teljesen egyszálú és az ilyen típusú kiszolgálókhoz képest meglehetősen jó teljesítményű. Cikkünk végén mutatjuk be.

Néha előfordulhatnak olyan helyzetek, amikor az esemény megválaszolásához kénytelenek vagyunk új folyamatot létrehozni. Sajnos a folyamat dinamikus létrehozása elég erőforrás-pazarló; ez alatt az időszak alatt sem az új sem a szülő folyamat nem képes új kérelmeket kezelni. Bizonyos kritikus helyzetekben, például a rendszermemória betelésekor elképzelhető, hogy nincs erőforrás ilyen célra. Ez azért komoly probléma mert a vészhelyzetben szükségünk lehet új folyamatok létrehozására amelyek figyelmesen zárják le a rendszert vagy hívásátadó procedúrákat hajtanak végre.

A probléma orvoslására bevezettük egy új, kapszulának nevezett modellt. A kapszula olyan előre beállított folyamat struktúra amely az egyéb üres kapszulákat tároló tárolóban található. Amikor a folyamat új végrehajtási környezetet akar létrehozni, a kapszulát kicsatoljuk a tárolóból és a jelenlegi folyamat néhány paraméterével alaphelyzetbe állítjuk.

A esemény regisztráláskor külön jelek mutatják hogy a kezelőt a folyamaton belül kell-e végrehajtani. Ha nem adunk meg paramétert vagy 0-t adunk meg akkor a kezelőt a folyamat futásának megtörésével kell végrehajtani. A jelek a következők:

- EFV_FORK: a folyamatot a `fork()` hívással azonos formában kell elvégezni.
- EVF_CAPSULE: a folyamatot a kapszula tárolóból vesszük.
- EVF_NOCLDWAIT: ennek a jelnek a SIG_NOCLDWAIT jelzéssel azonos jelentése van. Amennyiben létezik gyerek folyamat, új szülőt kap a kapszulakezelő szál személyében.
- EVF_KEEPALIVE: megakadályozza a folyamat/kapszula kilépését azáltal, hogy belép egy rendszermag ciklusba (a while(1) utasításhoz hasonlóan) csak felhasználói térben.

Memóriakezelés

Az esemény vezérelt rendszerekben igen fontos kérdés a memóriakezelés, hiszen az eseményeket közvetlenül a rendszermagtól származó alkalmazástérben található visszahívási függvények kezelik. A legegyszerűbb megoldás az lenne, ha csak akkor foglalnánk le a memóriát amikor a folyamat jelentkezik az eseményre. Ugyanakkor gondoljunk bele, mi történik ha rengeteg eseményt kell regisztrálni, vagy ha olyan folyamataink vannak amelyeket újra kell indítani, új eseményeket kell hozzájuk adni, esetleg eseményeket kell eltávolítanunk. Amennyiben az eseménykezelésben hiba történik, a rendszerintegritás sérül. Kevésbé veszélyes az operációs rendszer számára, ha az ilyen erőforrásokat maga kezeli és a memóriát a folyamatok igénylése szerint dinamikusan foglalja le.

Teljesítmény szempontból ezt a memóriát egy előre lefoglalt tárhelyből kell kiosztanunk, hogy elkerüljük a memória feldarabolódását és fenntartsuk a valós idejű jellemzőket. Az általános célú memória foglaló eljárások mint a `glibc malloc()` függvényei, bár általános célokra jól is használhatóak, nem igazán felelnek meg ezeknek a követelményeknek.

Néhány adattípust, például az egészeket, könnyedén át tudjuk adni a felhasználói térbe, az olyan összetettebb adattípusok viszont mint a karaktersorozatok, már külön megvalósítást igényelnek. A folyamatok memóriakezelését a ***glibc*** könyvtár végzi. A dolgok bonyolódnak ha a rendszermagtól szeretnénk memóriát foglalni, mivel arra is ügyelnünk kell, hogy az újonnan lefoglalt memóriaterület jó helyen legyen és a folyamat térben helyezkedjen el.

A felhasználói térből elérhető egyszerű és hatékony memóriafoglalás egyelőre hiányzik a rendszermagból, pedig szükség lenne rá.

Az ***AEM vmtable*** megoldása ezt a memóriafoglalási hiányosságot próbálja orvosolni. A ***„binary buddy allocator”*** egyik változatát valósítja meg (***Knuth*** után, lásd a forrásokat) a felhasználói folyamat memória tárán keresztül. Ennek segítségével majdnem bármilyen előre nem tervezett méretű és típusú adat kezelése megoldható. Kritikus memóriahiány esetén a eseménykezelőkön keresztül a felhasználóra bízza döntést. E képesség segítségével végső megoldásként visszanyúlhatunk glibc változathoz amennyiben valami baj történne.

Az *AEM* úgy készült, hogy amennyiben van még szabad blokk, konstans idő alatt adjon vissza érvényes mutatót. A telekommunikációs alkalmazások esetében amelyek valószínűleg azonos méretet igényelnek azonos alkalmazástípus esetén, általában éppen ilyesmire van szükség. A rövid idő alatt érkező és különböző méretű kérelmek által okozott memória töredezettséget is meg szerettük volna előzni.
A *vmtable* egy érdekes kiterjesztéssel is rendelkezik: felhasználói térből készíthetünk vele alaphelyzetbe állító függvényeket az eszközmeghajtókhoz. Ezt egy mutató használatával tehetjük meg, amelyet az *AEM* alrendszer segítségével a rendszermagtól foglaltunk és visszahívott függvény által adtuk át a felhasználói folyamatnak. A függvény visszatérésekor a terület visszakerül a rendszermaghoz. A visszahívott függvények nem csak események kezelésére jók, hanem nagyon jól használhatóak a rendszermaggal történő biztonságos kapcsolattartásra is.
Ebben a felállásban minden felhasználói folyamathoz rendelünk egy memóriaterületet, amit *vmtable*-nek nevezünk. Az elágaztatott (*forked*) folyamatok és kapszulák *vmtable*-je szülőfolyamatuk *vmtable* táblája alapján automatikusan öröklődik. Két különböző stratégiát alkalmazunk:

1. `VMT_UZONE`: a foglalást a folyamat heap szegmenségben hajtjuk végre. Ez ugyan gyors elérést tesz lehetővé, de a felhasználói folyamat memóriahelyét fogyasztja.

2. `VMT_VZONE`: A foglalást a rendszermag címterében végezzük, majd belapozzuk a folyamat címterébe. Ezzel minimális memóriafogyasztást érhetünk el, de a laphibák kezelése miatt a folyamat több időt vesz igénybe.

Mindkét eljárásnak helyzettől függően vannak előnyei és hátrányai. A kívánt stratégiát az alkalmazás indításakor, futásidőben választhatjuk ki. Az *5. ábra* a vmtable szerkezetét mutatja be.
A jelenlegi megoldásban a *vmtable* alrendszer közvetlenül fizikai lapokat foglal le. Ez azért van így, hogy a foglalások biztosan folyamatosak legyenek és a memóriát biztonságosan használhassák az I/O műveletek. A jövőben hálózati teljesítmény terén is szeretnénk közvetlen ki-/bemenetere felhasználni a *vmtable* rendszert.
A *vmtable* egyszerű és könnyen kezelhető felületet nyújt az *AEM* felhasználóknak és modulfejlesztőknek, a memóriafoglalás összetettségét a rendszermag térbe rejtve. Az átlátszóság kedvéért az *AEM* magmodulba egyszerű memóriafoglalási és felszabadítási rutinokat építettünk. Ez a csatolófelület nagyban leegyszerűsíti a modulok karbantartását és átvitelüket a különféle *Linux* rendszermag kiadásokra.

Skálázhatóság

Végeztünk egy tesztet, hogy megtudjuk miképpen viselkedik az *AEM* két távoli folyamat között zajló egyszerű adatcsere közben. A teszt fő célja annak kiderítése volt, hogy a eseménykezelő környezetváltása miatt fellépő időveszteség nem okoz-e teljesítmény problémákat. Mostanában elvégeztünk egy teljesítménytesztet is amely az *AEM* skálázhatóságát tette próbára valamint ellenőrizte az AEM magvát alkotó munkafolyamatok és várakozási sorok belső megvalósítását.
Könnyen összehasonlítható számokat keresve, úgy döntöttünk, hogy egy már létező webkiszolgálót keresünk és azt alakítjuk az *AEM* csatolófelületéhez. Az *AEMhttpd* egyszerű, egy szálon futó *HTTP* kiszolgáló (lásd a forrásokat.)
Az egyszálú kiszolgáló teljes egészében a folyamat fő végrehajtási szálában fut. Azaz nem hozunk létre sem felhasználói sem rendszermag szálakat a *HTTP* kérelmek kezelésére. A méréseket ilyen típusú kiszolgálóval végeztük és inkább a megvalósításra semmint a kiszolgáló tényleges teljesítményére koncentráltunk.
A *6. ábrán* bemutatott példán 100 aktív tranzakciót futtattunk. Minden tranzakcióban megnöveltük a nyitott kapcsolatok számát ezzel megnövelve a munkafolyamatok számát a socket kapcsolat várakozási sorában. A `select()`-en alapuló hagyományos megoldásban ez megnövelte volna a kérelmekre adott válaszidőt, hiszen a leírókat a rutin sorban nézné végig. Az *AEM* alkalmazásával azonban csak az aktív munkafolyamatok (azaz a kész adatot tartalmazó socketek) hajtják végre a megfelelő eseménykezelőiket. Ezzel a módszerrel az *AEM* képes általános és skálázható megoldást nyújtani.

Összefoglalás

Az iparban széles körben használják a *Linuxot*, amely büszkén nevezheti magát vállalati szintű megoldások választott operációs rendszerének. Nem jár messze attól sem, hogy a következő generációs IP-alapú telefonszolgáltatások választott operációs rendszerévé váljon. A *Linux* képességeit ma már széles körben elismerik, de a skálázhatóságot, teljesítményt és megbízhatóságot váró, következő generációs szolgáltatókat a rendszermag szinten végezett fejlesztések fogják leginkább vonzani.
Az *AEM* erőteljes próbálkozás az aszinkron folyamatértesítés és esemény adatkiegészítés terén. Segítségével kihasználhatjuk az esemény vezérelt programozás előnyeit, amely biztonságos alkalmazásfejlesztést tesz lehetővé. Ezen felül az *AEM* olyan részeket is tartalmaz, melyek célja az alkalmazás megbízhatóságának és hordozhatóságának növelése könnyen kezelhető felülete révén.

Köszönetnyilvánítás

Köszönet az *Open Systems Lab* összes kutatójának az *Ericsson*-tól pedig *Lars Hennert*-nek hasznos tanácsaikért, valamint az *Ericsson Research*-nek, hogy jóváhagyta e cikk megjelenését.

Linux Journal 2004. november, 127. szám

Frédéric Rossi (Frederic.Rossi@ericsson.ca)
Az Ericsson Research Open Systems Lab fejlesztője Kanadában, Montreál városában. Az AEM atyja és a fejlesztés vezetője.