Egy kis infó … az OpRe 2. ZH-hoz! GL

Kliensek párhuzamos kiszolgálása gyermek folyamatok (child processek) segítségével

Egy folyamaton (process) egy program egy futó példányát értjük. Egy multitaszking operációs rendszer egyszerre több process-t képes működtetni egymással párhuzamosan. Minden folyamatnak van egy azonosítója, PID-je. Minden folyamat beletartozik egy folyamat csoportba (process group), és ezt az ún. GID-el azonosítjuk. Amit a shell-ről egy sorban indítunk, azok kerülnek egy csoportba. Ha a program nevét egyedül gépeltük be, akkor a folyamat egyetlen lesz a csoportjában.
Egy folyamat további folyamatokat indíthat, ekkor az indító folyamatot szülőnek (parent process), az általa indítottat pedig gyermeknek (child process) hívjuk. Tulajdonképpen minden folyamatnak van szülője, hiszen valahogyan elindult. Folyamatok indításának több módja van, az egyiket gyakran használják több klienst egyszerre kiszolgálni tudó szerverek készítéséhez.

Ez a **fork** függvény.

```
#include <unistd.h>
pid_t fork(void);
```

A függvény segítségével a folyamat létrehoz saját magából még egy példányt. A folyamat egész memóriaterülete átmásolódik, és a szülőtől függetlenül elindul. Tehát a két folyamat ***nem*** látja egymás változóit, ugyanaz a programkód fut, de különböző adatokon. Hasonló, mintha kétszer indítottuk volna el ugyanazt a programot. A gyermek örökli a szülőtől a születése pillanatában (fork meghívásakor) nyitva lévő leírókat. Ez azt jelenti, hogy írhat, olvashat a leíróval, képződik számára egy másolat belőle. Ha a szülő az adott leírót lezárja, ez a gyermek számára nem jelent semmit, az övé továbbra is működni fog. A gyermek a szülő process group-jába fog tartozni.A szülő futása a **fork** függvény utáni sorral folytatódik, a gyermek futása pedig ugyanitt kezdődik el. De van egy kis csel. A szülő számára a fork a gyermek PID-jét adja vissza, míg a gyermek számára nullát. Ez első hallásra kicsit ködös lehet, íme egy példa:

```
//itt a szuloben inditjuk a forkot,meg nincs gyermek
pid=fork()
/*a szuloben es a letrehozott gyermekben is itt visszatert a fork()*/
//A gyermek szamara nullaval tert vissza
if (pid==0)
{
  /*IDE A GYERMEK FOLYAMAT JUTHAT CSAK EL*/
  //itt vegzi el a gyermek a feladatat
  //mondjuk varakozik egy kapcsolatra es kiszolgalja
  clientsock=accept(listensock, &clientaddr, &size);
  kiszolgal(clientsock);
  //es nem felejt el meghalni
  exit(0);
  //A gyermek KILEPETT
}
//IDE A SZULO JUTOTT EL
if (pid<0)
{
  //nem sikerült a fork
```

```
  perror("fork");
  exit(1);
}
```

Az, hogy a gyermekből, vagy a kliensből várakozunk kapcsolatra, nem mindegy. Két véglet létezik. Az egyik, mikor a szülőből várakozunk a kapcsolatra, és csak akkor hozzuk létre a gyermek folyamatot, amikor kiépült a kapcsolat. Ennek hátránya, hogy ha egyszerre több kliens próbál kapcsolódni, az elsőnek végig kell várnia, míg a gyermek folyamat elkészül számára, a másodiknak pedig végig kell várnia, míg az első számára elkészül, meg még saját maga számára, és így tovább. Előnye viszont, hogy csak annyi erőforrást használ, amennyi szükséges.

Valami ilyen váz lesz:

```
while(1)
{
  //A szulo varakozik
  clientsock=accept(listensock, &clientaddr, &size);
  if (fork()==0)
  {
    /*CHILD*/
    kiszolgal(clientsock);
    close(clientsock);
    exit(1); //kiszolgalas utan rogton kilep
  }
  /*PARENT*/
  close(clientsock);// a gyermek orokolte, mi bezarhatjuk
  //fut tovabb a ciklus, es var a kovetkezo kliensre
  //az elozo kiszolgalasa mar folyamatban a gyermek processben
}
```

A másik véglet, ha előre legyártjuk az összes gyermek folyamatot, amik mind kapcsolatra várakoznak. Ilyenkor több egy időben kapcsolódó kliens is egyszerre azonnal sorra kerül. Nem kell senkinek sem várakozni a **fork**-ra. Hátránya, hogy a szükségesnél sokkal több erőforrást kötünk le.

A váz például ilyen is lehet:

```
#define MAXCHILD 64
child=0;
while(1)
{
  /*addig gyartunk childokat, amig el nem erjuk a maxot*/
  if (child<MAXCHILD)
  {
    if (fork()==0)
    {
      /*CHILD*/
      //A gyermek var a kapcsolatra
      clientsock=accept(listensock,&clientaddr,&size);
      kiszolgal(clientsock);
      exit(0); //kiszolgalas utan kilep
    }
    /*PARENT*/
    child++;
  }
  else
```

```
  {
    /*elertuk a maximum szamu childot*/
    wait(NULL); //varunk, mig nem exital valamelyik
    child--;
  }
}
```

Láthatólag a ciklusba belépés után addig fog egymás után gyermek folyamatokat gyártani, míg nem lesz belőlük *MAXCHILD*. Ezek mind ott lesik, mikor jön a kliens. Ha már megvan a megfelelő számú gyermek, akkor a szülő vár, hogy mikor lép ki valamelyik. Erre való a **wait** függvény. Ha egyik végzett, utána rögtön fork-ol majd egy újat. Mindig fenntartja a *MAXCHILD* számú gyermeket. Ha 64 kliens egyszerre próbál csatlakozni, akkor mindegyik várakozás nélkül kiszolgálásra kerül. Ha 64-nél több kliens van, akkor valakinek bizony várnia kell.

Child processek kezelésekor gyakran előfordulhat, hogy meg kell várnunk, míg egyik, vagy másik befejezi futását. Ahogy a példában is előjött.

Wait, és **waitpid** függvény.

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status)
pid_t waitpid(pid_t pid, int *status, int options);
```

A **wait** függvény felfüggeszti a folyamat futását, amíg legalább egy gyermek folyamata nem lép ki. Ha az éppen a függvény meghívása előtt lépett ki, akkor a függvény rögtön visszatér. A paraméterként megadott *status*-ba kerül a gyermek által adott kilépési érték. (amit az exit() zárójelei közé írtunk) Persze, *NULL* is lehet, ha nem vagyunk rá kíváncsiak.

A **waitpid** függvény felfüggeszti a program futását, amíg a megadott azonosítójú folyamat véget nem ér. Ha bármelyik gyermek jó lesz, akkor adhatunk meg nullát is. Lehet megadni opciókat is. Jól használható a *WNOHANG*. Ennek hatására a függvény azonnal visszatér, ha nem ért véget egy process sem. Így nem blokkolódik a program végrehajtása. Bővebben 'man waitpid'.
Szükség lehet rá, hogy a gyermek futása közben olyan információt igényel, amelyet csak a szülő birtokol. A gyermek folyamatok, és a szülő közti kommunikáció kicsit nehézkes, mivel egymástól elkülönítve futnak, egymás adataihoz semmi közük. Folyamatok közti kommunikációra (IPC - interprocess communication) több módszer létezik, ezek közül kell egyet használnunk. Itt csak a legegyszerűbbet próbálom bemutatni.

A folyamatok közti kommunikáció pipe-okkal

A pipe olyan, mint egy csővezeték, amit beöntünk az egyik végén, kijön a másikon. Más szavakkal FIFO működésű. Amit először írunk az egyik oldalon, azt először olvashatjuk a másik oldalon. Pozícionálásra nincs lehetőség. A pipe-nak nincs neve, két fájlleíróval azonosítjuk. A két leíró a vezeték két vége, az egyiken csak írni tudunk, a másikon csak olvasni. Mivel a gyermek folyamat örökli a szülő leíróit, a pipe-ot is örökli, így egyszerűen használható a szülővel való kapcsolattartásra.

Pipe létrehozása a **pipe** függvénnyel

```
#include <unistd.h>
int pipe(int filedes[2]);
```

Paraméterként a két leíró tömbjét kell megadni. A tömb nulladik eleme lesz a pipe eleje, azaz ide lehet írni, és csak írni. A tömb első eleme a pipe vége, itt csak kiolvasni lehet a másik oldal által írt adatokat. Siker esetén a visszatérési érték nulla.Mivel a gyermek folyamatok öröklik a pipe két végének leíróit, annak több eleje és több vége lesz. Ajánlott, hogy mind a gyermek, mind a szülő minél hamarabb bezárja azt a véget, amit nem használ.

Ha a pipe összes írható végét bezárjuk, akkor a másik felén fájlvége jelet olvasunk, azaz a read nullával tér vissza. Ha az olvasható vég(eket) zárjuk be, a rendszer nem enged a másik végen írni, a write hibával tér vissza. A pipe csak addig él, amíg legalább egy eleje, és egy vége van.

Annak az esetnek, hogy több olvasható vége legyen a pipe-nak, nincs sok értelme, hiszen, mikor az egyikről kiolvassuk az adatot, az eltűnik a pipe-ból, és a többi nem fogja megkapni. Fordítva, azaz több írható vég esetén a másik oldal ugyanúgy sorrendben olvassa ki az adatokat, mintha azokat egy végen írták volna, így ez használható helyzet.

Ha a pipe üres, akkor a read függvény blokkolódik, egyébként azonnal visszatér.

pl.:

```
int main (void)
{
  pid_t pid;
  int mypipe[2];
  /*pipe létrehozása*/
  pipe (mypipe)
  /* child process letrehozasa */
  pid = fork ();
  if (pid == 0)
  {
    char c;
    /*CHILD*/
    //ezt a parent hasznalja, itt bezarhatjuk
    close(mypipe[1]);
    //olvas, amig a szülő be nem zárja a pipe másik végét
    while (read(mypipe[0],&c,1))
    {
      putchar(c);
    }
    exit(EXIT_SUCCESS);
   }
  else
  {
    /*PARENT*/
    //ezt a veget a child hasznalja
    close(mypipe[0]);
    //irunk a pipe-ra, a child majd kiolvassa
    write(mypipe[1],"hello!\n",strlen("hello, world!\n"));
    write(mypipe[1],"bye!\n",strlen("bye, world!\n"));
    exit(EXIT_SUCCESS);
  }
}
```

A szülő kiír a pipe-ra, a gyermek pedig olvassa, és kiírja a képernyőre.

A példaprogram szintén pipe-okkal kommunikál a gyermek folyamatokkal. A program ugyanazt próbálja csinálni, mint amit a select-es példa. Amit az egyik child küld, azt kiírja minden másik kliensnek is. Kezdetleges névazonosítás van, de akár többen is fent lehetnek egy néven. A már megismert select függvényt is használom a programban, nehogy túl egyszerű legyen. Ld. chat_fork.c (alább: forráskódok)
Mint láttuk, a folyamatok használatakor problémánk adódik abból, hogy az egyes folyamatok egymástól elzárva futnak. Külön módszerekkel tudnak csak egymással kommunikálni. Az erőforrásokkal is kicsit pazarlóan bánunk, hiszen jobb esetben minden klienshez újra lemásolódik az egész folyamat programkódja, ezzel felélve a memóriát. De, van egy nagy előnye. Az egyes gyermek processek működésük során esetleg fellépő hibák semmiképpen sem hatnak ki végzetesen a többire, mivel külön memóriatartományban, elszigetelten működnek. Ha egy összeomlik, azt csak az általa kiszolgált kliens fogja rossz néven venni, a szerver menni fog tovább.

Ezt az előnyt kell feladnunk annak érdekében, hogy a program gyorsabb legyen és kevesebb erőforrást igényeljen.

Párhuzamos kiszolgálás szálak segítségével

Alapesetben egy folyamatnak egy programszála (thread) van, de több szálat is indíthatunk. Ez hasonló kicsiben, mint amit az operációs rendszer csinál a process-ekkel. Párhuzamosan hajtja őket végre, megosztja köztük az erőforrásokat. Ha több szálat készítünk, azok egymás mellett fognak működni, a program egyszerre több dolgot csinál. Szálak létrehozása sokkal gyorsabb, mint process-eké, hiszen a process-nek van saját azonosítója, memóriaterülete, külön örökölt leírói, stb.. Ezeket mind a kernelnek kell létrehoznia, ami sok rendszeridőt, felemészt, nem beszélve a memóriáról. Egy folyamat programszálai ugyanazt az egy programkódot futtathatják, folyamat minden szála osztozik a folyamat erőforrásain, azaz ugyanazon memóriaterületet látják, ha az egyik átír egy változót, akkor a másik is az új értéket fogja látni.,ha a folyamat egyik szála megnyit egy leírót, azt az összes többi is használhatja.

Tehát a szálak gyorsabban létrehozhatóak, egymással könnyen kommunikálhatnak, de használatuk óvatosságot igényel, hiszen, ha az egyikben hiba történik, könnyen magával ránthatja a többit. Ezen kívül problémát jelent még a szálak szikronizációja. Nem nyerő, például, ha az egyik akkor olvas ki egy változót, mikor a másik még csak az egyik byte-ját írta át.

Szál létrehozása

```
#include <pthread.h>
int pthread_create(pthread_t * thread, pthread_attr_t *attr,
void * (*start_routine)(void *), void * arg);
```

thread A szál adatai itt fognak helyet kapni.
attr A szál tulajdonságait tudjuk megadni. Ha jó az alapértelmezett, lehet NULL.
start_routine Ez a függvény lesz a szál függvénye, ez fog elindulni.
*arg E*zt az argumentumot fogja megkapni a függvény.

Siker esetén nullával tér vissza, és a szálat azonosító adatok *thread* -be kerülnek. Hiba esetén negatív értékkel tér vissza.
Egy szál indulásakor meghívódik a megadott függvény az argumentummal.

Szál befejezése

```
#include <pthread.h>
void pthread_exit(void *retval);
```

A szál kétféleképpen érhet véget. Vagy visszatér a függvény, ekkor a visszatérési érték lesz a szál visszatérési értéke, vagy a szál meghívja a pthread_exit függvényt a visszatérési értékkel.

Szál tulajdonságai

A tulajdonságot tartalmazó változó előkészítése.

int pthread_attr_init(pthread_attr_t *attr);

A függvény előkészíti a tulajdonságot, feltölti az alapértelmezett értékekkel. A szálaknak számos tulajdonságát befolyásolhatjuk. Ennek teljes listáját mellőzném, aki kíváncsi, annak 'man pthread_attr_init'.

Egy tulajdonságot emelnék ki, ez a *detachstate*.

A tulajdonság határozza meg, hogy egy adott szálhoz szinkronizálhatunk-e egy másikat.

Ha az értéke *PTHREAD_CREATE_JOINABLE*, akkor igen.

Ez azt jelenti, hogy egy másik szál képes várakozni (ld. man pthread_join), amíg ez leáll. Ezzel az a probléma, hogy a szál azonosításához használt memória nem szabadítódik fel rögtön a szál befejeződésekor, hanem csak akkor, ha egy másik szinkronizálja magát hozzá. Ezért, ha nincs szükség a szinkronizációra, kapcsoljuk ki. Adjunk neki *PTHREAD_CREATE_DETACHED* értéket.

Ezt a **pthread_attr_setdetachstate** függvénnyel tehetjük meg.

Egy példa:

```
pthread_t chld_thr;
pthread_attr_t attr;
....
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
while(1)
{
  if (threads<MAXTHREADS && (clientsock=clientconnect(listensock))>0)
  {
    /* uj szal keszitese a cliensnek */
    pthread_create(&chld_thr, &attr, chat_thread, (void *) clientsock);
  }
}
```

A szálakkal ugyanúgy többféleképpen felépíthetjük a szervert, mint a folyamatokkal. Létrehozhatjuk az összes szálat egyszerre, vagy csak akkor hozunk létre egyet, mikor a kapcsolat felépült. Az folyamatok, és szálak kombinációja is elképzelhető, azaz minden kapcsolathoz létrehozunk egy folyamatot, ahol egyszerre több szál szolgál ki egyetlen klienst, ezzel felgyorsítva a kiszolgálást.

A szálakat használó programot *-lpthread* kapcsolóval kell lefordítani, ezzel megadjuk a linker-nek, hogy a szálakat kezelő könyvtárat is bele kell szerkeszteni a programba.

A szálakat használó példaprogram ugyanazt végzi el, mint az eddigi kettő, egyszerre több klienssel tart kapcsolatot, és amit az egyiken beírtak, azt kiküldi az összes többinek. Ld. chat_thread.c
Az itt leírt kevéske szálkezelés a POSIX szabványhoz illeszkedik. Ez nem minden UNIX rendszeren így működik, de a Linux-okon mennie kell. Ez egy hátrány a process-ekkel szemeben, ott ugyanis minden UNIX rendszeren hasonlóan mennek a dolgok.