Linux Programozás

Daemon

Daemon

- Nincs felhasználói felületük.
- Nem kapcsolódnak terminálhoz.
- Tipikusan a rendszer indítja.
- Szerver funkciókat látnak el.
- Más processzekkel kommunikálnak valamilyen IPC mechanizmussal. (Gyakran socket)

Processzcsoport

- A processzek egy halmaza a processzcsoport.
- Minden processz egy processzcsoporthoz tartozik.
- A processzcsoport azonosítója: PGID
- Minden csoportnak van egy vezetője. PGID = vezető PID
- A processz létrejöttekor automatikusan a szülő PGID-jét örökli.

Processzcsoport lekérdezése

```
pid_t getpgid(pid_t pid);
```

- pid: a lekérdezendő folyamat PID-je. Ha 0, akkor az aktuális processzét kapjuk.

Processzcsoport beállítása

```
int setpgid(pid_t pid, pid_t pgid);
```

- pid: A folyamat, amelyikre beállítjuk. 0 – az aktuális folyamat.
- pgid: A csoport azonosító. 0 – az aktuális folyamat PID-je.
- A setpgid(0, 0) jelentése: a folyamat új csoportot alkot, amelyiknek ő lesz a vezetője.

Session (munkamenet)

- A processzcsoportok session-be szerveződnek.
- Eredetileg a terminálon bejelentkezett felhasználó processzcsoportjait fogja össze.
- Maximum egy vezérlő terminál kapcsolódik hozzá.
- Egy terminálhoz egy session tartozik.
- Egy vezető processz van. SID = vezető PID

Session lekérdezése

```
pid_t getsid(pid_t pid);
```

- pid: Folyamat PID – 0: aktuális

Session beállítása

```
pid_t setsid(void);
```

- Létrehoz egy új session-t és a folyamat lesz a vezetője.
- A folyamat nem lehet csoport vezető, különben nem működik.
- Az új session-nek nincs kontroll terminálja.

Daemon

- A daemon nem függhet
  - terminálról
  - könyvtártól
  - indító felhasználótól
- Le kell kapcsolódni a terminálról:
  - Saját session-t kell létrehozni.
  - Nem lehet csoport vezető, ezért fork-olunk. De a szülő processzre nincs szükség.
  - Lezárjuk a ki és bemeneteket.
- Az aktuális könyvtárat a gyökérre állítjuk.
- Az umask-ot lenullázzuk.

Daemon

- Az eddigiek alapján összegezve az általános lépések az alábbiak:
  - új folyamat létrehozása a fork függvénnyel,

```
pid = fork();
```

  - új munkamenet létrehozása,

```
setsid();
```

  - az umask beállítása nullára,

```
umask(0);
```

  - a munkakönyvtár beállítása a gyökérkönyvtárra,

```
chdir("/");
```

  - a szabványos állományleírók bezárása.

```
close(STDIN_FILENO);
close(STDOUT_FILENO);
close(STDERR_FILENO);
```

Daemon

Ugyanez egyszerűbben:

```
int daemon(int nochdir, int noclose);
```

- Elvégzi a fork() és setsid() függvényhívásokat.
- Ha a nochdir = 0, akkor elvégzi a könyvtárváltást.
- Ha a noclose = 0, akkor lezárja a be és kimeneteket.

Daemon +

- Célszerű az alábbi lépéseket is megtenni:
  - naplózás,
  - a jelzéskezelők (signal handler) megírása és regisztrálása.

Jogosultságok

Folyamat jogosultságai

- Valódi UID, GID:
  - Az indító felhasználótól örökli.
- Effektív UID, GID:
  - setuid és setgid esetén eltér a „valóditól”
  - Ezt figyeli a kernel.
- Az állomány jogosultságok feldolgozása az effektív UID és GID alapján történik.
- Egy root felhasználó (UID = 0) nevében futó folyamat módosíthatja az azonosítókat.
- A módosított effektív UID/GID a mentett UID/GID-ben tárolódik.

UID és GID átállítása

- A valós és az effektív UID/GID átállítása:

```
int setuid(uid_t uid);
int setgid(gid_t gid);
```

- Effektív UID/GID állítása:

```
int seteuid(uid_t euid);
int setegid(gid_t egid);
```

- Valós/effektív UID/GID lekérdezés:

```
uid_t getuid(void);
uid_t geteuid(void);
gid_t getgid(void);
gid_t getegid(void);
```

Felhasználói nevek

- A felhasználói adatokat klasszikusan az /etc/passwd állomány tartalmazza.
- Lehet szerveren is (Yellow Pages, LDAP, stb.)
- A leíró struktúra:

```
struct passwd
{
  char *pw_name;   /* felhasznaloi nev */
  char *pw_passwd; /* felhasznaloi jelszo kodolva*/
  uid_t pw_uid;    /* felhasznaloi azonosito */
  gid_t pw_gid;    /* csoportazonosito */
  char *pw_gecos;  /* valodi nev */
  char *pw_dir;    /* home konyvtar */
  char *pw_shell;  /* shell program */
};
```

Felhasználói adatok lekérdezése

- Név alapján

```
struct passwd *getpwnam(const char
*name);
```

- UID alapján

```
struct passwd *getpwuid(uid_t uid);
```

Csoport nevek

- A csoport neveket és azonosítókat klasszikusan az /etc/group tartalmazza.
- Lehet szerveren is.
- A leíró struktúra:

```
struct group
{
  char  *gr_name;   /* csoportnev */
  char  *gr_passwd; /* csoportjelszo */
  gid_t  gr_gid;    /* csoportazonosito */
  char **gr_mem;    /* csoporttagok */
};
```

Csoport adatok lekérdezése

- Név alapján

```
struct group *getgrnam(const char
*name);
```

- GID alapján

```
struct group *getgrgid(gid_t gid);
```

Szálak

A szálak

- A szálak párhuzamosan futó, külön ütemezhető utasítás sorozatok.
- A szálak a folyamatokkal ellentétben közös címtartományban futnak.
- A Linux esetén a szálak könnyű súlyú folyamatok:
  - megosztoznak erőforrásokon, ami könnyebb létrehozást és váltást eredményez
- Egy hibás szál a többit is magával ránthatja a folyamaton belül.
- A legelterjedtebb library specifikáció a POSIX szál API: pthread

pthread fejlesztői könyvtár

- Header: pthread.h
- Linkelés: -lpthread

Szálak létrehozása

```
int pthread_create(pthread_t *thread, const
pthread_attr_t *attr, void
*(*start_routine) (void *), void *arg);
```

- Létrehoz egy új szálat, amely végrehajtja a megadott függvényt.
- thread: szál leíró struktúra
- attr: szál beállítások (NULL esetén örökli)
- start_routine: a szál függvényének mutatója
- arg: a szál függvénye ezzel a paraméterrel hívódik

Szál végének megvárása

```
int pthread_join(pthread_t thread,
void **retval);
```

- Felfüggeszti a hívó szál működését, amíg a várt szál véget nem ér.
- thread: a várt szál leíró struktúrája
- retval: a szál visszatérési értéke (kimeneti paraméter)

Szálak létrehozása C++ esetén

- Egy objektum tagfüggvényét szeretnénk megadni szál függvénynek.
- A probléma a függvény mutató.
- Osztály függvényt (static) megadhatunk függvény mutatóként.
- Az objektum mutatóját átadhatjuk paraméterként.
- Az osztályfüggvényben meghívhatjuk az objektum egy tagfüggvényét.

Szálak attribútumai

- Inicializálás:

```
int pthread_attr_init(pthread_attr_t
*attr);
```

- Megszüntetés:

```
int pthread_attr_destroy(pthread_attr_t
*attr);
```

Szálak attribútumainak beállításai

int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);

int pthread_attr_getdetachstate(const pthread_attr_t *attr, int *detachstate);

int pthread_attr_setschedpolicy(pthread_attr_t *attr,int policy);

int pthread_attr_getschedpolicy(const pthread_attr_t *attr, int *policy);

int pthread_attr_setschedparam(pthread_attr_t *attr, const struct sched_param *param);

int pthread_attr_getschedparam(const pthread_attr_t *attr, struct sched_param *param);

int pthread_attr_setinheritsched(pthread_attr_t *attr, int inherit);

int pthread_attr_getinheritsched(const pthread_attr_t *attr, int *inherit);

int pthread_attr_setscope(pthread_attr_t *attr, int scope);

int pthread_attr_getscope(const pthread_attr_t *attr, int*scope);

Szálak attribútumainak beállításai

Attribútum	Attribútum leírása
detachstate	Két értéke lehet, PTHREAD_CREATE_JOINABLE (alapértelmezés), illetve PTHREAD_CREATE_DETACHED. Az előbbi a szál csatlakoztatható állapota, a másik a lecsatolt.
schedpolicy	Lehet SCHED_OTHER, ami az alapértelmezett nem valósidejű ütemezés, valamint két valósidejű ütemezés, SCHED_RR egy körbeforgó (round-robin), a SCHED_FIFO FIFO-prioritást jelent. A két utóbbihoz a processznek *root* jogokkal kell rendelkeznie.
schedparam	Az ütemezési prioritás a két valósidejű ütemezésre.
inheritsched	Az alapértelmezés PTHREAD_EXPLICIT_SCHED, ha az új szál beállításai (shedpolicy, shedparam) a mérvadóak, PTHREAD_INHERIT_SCHED, ha a létrehozó szál beállításait veszi át az új szál.
scope	Ld. man

Szálbiztos függvények

- Ne használjunk globális és statikus változókat.
- Többszálú környezetben az errno sem globális változó (makró, amely szálbiztos függvényt hív).
- A pthread dokumentáció (man pthreads) tartalmazza a szálbiztos függvényeket.
- Létrehozhatunk szál specifikus adatokat:
  - Egy szálon belül globális adat terület.
  - Más szál által elérhetetlen.
  - Valójában egy asszociatív tömb (kulcs-érték párok).

Szál leállítása

- Szálon belülről:
  - Visszatérünk a szál függvényéből.
  - `void pthread_exit(void *retval);`
- Másik szálból:
  - Csak úgy nem „lőhetjük” ki, mert nem szabadulnak fel erőforrások. A szinkronizációs objektumok „beragadhatnak”.
  - Vannak törlési pontok (cancellation point). (man pthreads)

```
void pthread_testcancel(void);
```

  - Törlés kérése:

```
int pthread_cancel(pthread_t thread);
```

Szálak és a fork

- A hívó szálból készül az új folyamat.
- A többi szál nem jön létre, csak a pillanatképe másolódik:
  - Nem hívódnak meg a tisztogató műveletek.
  - Inkonzisztens lehet az állapot.
- Lehetőleg csak úgy használjuk, ha utána exec jön!

POSIX Szinkronizálás

- Alapja a futex (fast user-space mutex)
- Kölcsönös kizárás (mutex)
- Feltételes változók (conditional variable)
- Szemafor
- Spinlock
- További lehetőségek:
  - POSIX megosztott memória
  - üzenetsorok

Kölcsönös kizárás

- `pthread_mutex_t`
- Típusai:
  - gyors (PTHREAD_MUTEX_INITIALIZER)
  - rekurzív (PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP)
  - hibaellenőrző (PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP)
- Inicializálása:
  - Létrehozáskor
  - `int pthread_mutex_init(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t *restrict attr);`
- Megszüntetés:

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Kölcsönös kizárás

- Foglalás:

```
int pthread_mutex_lock(pthread_mutex_t
*mutex);
```

- Foglalás várakozás nélkül:

```
int pthread_mutex_trylock(pthread_mutex_t
*mutex);
```

- Felszabadítás:

```
int pthread_mutex_unlock(pthread_mutex_t
*mutex);
```

Feltételes változók

- A szál egy feltételre vár. A feltétel teljesülését egy másik szál jelezheti neki.
- `pthread_cond_t`
- Inicializálás:
  - `pthread_cond_t cond = PTHREAD_COND_INITIALIZER;`
  - `int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond_attr);`
- Megszüntetés:

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

Feltételes változók

- Jelzés:
  - Mindenkinek:

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

  - Egy valakinek:

```
int pthread_cond_signal(pthread_cond_t *cond);
```

- Várakozás:
  - Végtelen:

```
int pthread_cond_wait(pthread_cond_t *restrict
cond, pthread_mutex_t *restrict mutex);
```

  - Timeoutos (abs idő, gettimeofday()):

```
int pthread_cond_timedwait(pthread_cond_t *restrict
cond, pthread_mutex_t *restrict mutex, const struct
timespec *restrict abstime);
```

POSIX szemafor

- Header: semaphore.h
- Létrehozás:
  - Névtelen:

```
int sem_init(sem_t *sem, int pshared,
unsigned int value);
```

  - Megnevezett:

```
sem_t *sem_open(const char *name, int
oflag);
sem_t *sem_open(const char *name, int
oflag, mode_t mode, unsigned int value);
```

POSIX szemafor

- Kernel < 2.6:
  - Csak névtelen szemafor
  - Csak szálak között
- Kernel >= 2.6 + glibc NPTL szálkezeléssel:
  - Teljes implementáció
  - Névtelen és megnevezett szemafor
  - Mindkettő működik processzek között (névtelen esetén megosztott memóriában kell lennie)

POSIX szemafor

- Foglalás:

```
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_timedwait(sem_t *sem, const
struct timespec *abs_timeout);
```

- Felszabadítás:

```
int sem_post(sem_t *sem);
```

- Aktuális érték:

```
int sem_getvalue(sem_t *sem, int
*sval);
```

POSIX szemafor

- Megsemmisítés / lezárás:
  - Névtelen:

```
int sem_destroy(sem_t *sem);
```

  - Megnevezett:

```
int sem_close(sem_t *sem);
```

- Megnevezett szemafor törlése:

```
int sem_unlink(const char *name);
```

Spinlock

- Ha a mutexet csak rövid ideig foglalják a szálak, akkor a várakozás helyett érdemes próbálkozni.
- A folyamatos próbálkozással hosszú foglalás esetén pazaroljuk a CPU-t.
- `pthread_spinlock_t`
- Inicializálás:

```
int pthread_spin_init(pthread_spinlock_t
*lock, int pshared);
```

- Megszüntetés:

```
int pthread_spin_destroy(pthread_spinlock_t
*lock);
```

Spinlock

- Foglalás:

```
int pthread_spin_lock(pthread_spinlock_t
*lock);

int pthread_spin_trylock(pthread_spinlock_t
*lock);
```

- Felszabadítás:

```
int pthread_spin_unlock(pthread_spinlock_t
*lock);
```