# 15. Tétel

A Linux rendszer milyen POSIX szálkezelő eszközöket implementál? Hogyan hozhatunk létre szálakat és milyen szinkronizációs eszközök állnak rendelkezésre?

## Szálak: könnyűsúlyú processzek (Lightweight Processes)

Processzek részei:

* kód
* adat
* verem
* fájlleírók
* jelzéstáblák

Processzek között csak a kódrész a közös ⬄ a szállak esetén csak az adatsrtuktúrák + ugyanabban a címtartományban futnak 🡪 sokkal könnyebb váltás + kommunikáció egymás között

## Linux száltípusok:

* **felhasználói módban futó szálak**
  + #include <sys/shm.h>
  + nem használják a kernelt az ütemezéshez – kooperatív multitasking: a processz definiál saját szálkezelést
  + a szálkezelés gyorsabb
  + a szálak nem adják át a processzort más szálaknak 🡪 éhezés a várakozó szálak között 🡪 pl. egy szinkron I/O hívás blokkol egy szálat 🡪 többi szál nem tud futni
  + nem tudja az OS használni a szimmetrikus multiprocesszoros környezetet (SMP)
  + ezekre megoldások: ezek OS-től elvárt funkciókat implementálnak a fülé 🡪 nehézkessé teszi
    - monitorozás éhezés ellen
    - szálak különböző processzoron futtatása
* **kernel módban futó szálak**
  + #include <sched.h>
  + ki tudják használni az SMP előnyeit
  + I/O blokkolás nem probléma
  + szálváltás nem sokkal lassabb már, mint a felhasználói módban futó társainál
  + Kernel feladata a szálkezelés (szükséges struktúrák számon tartása, és szálváltások)
  + 1.3.56 kernelverzió óta a kernel módban futó szálkezelést támogatja a Linux

## Szálak létrehozása

* int clone(int(\*fn) (void\*), void \*child\_stack, int flags, void \*arg)
  + fork fv. kiterjesztése
  + (void\*) – az indítandó processz vagy szál belépési pontja (pointer az fn függvényre, aminek void az argumentuma, és a visszatérési értéke int (így ennek a deklaráció is int típusú))
  + \*child\_stack – veremmutató
  + \*arg – a megadott függvénynek átadandó paraméter
  + flags – egyéb opciók
* int pthread\_create(pthread\_t \*thread, pthread\_ \*attr\_t attr, void \* (\*start\_routine) (void \*), void \* arg);
  + szál elindítása
  + belépési pontja a start\_routine fv.
    - void\* start\_routine(void\* param) formában deklarálható
    - a param paraméter az prhread\_create fv arg paraméterével adhatunk meg
  + \*thread argumentumban a szál leíróját kapjuk vissza
  + attr paraméter
    - NULL – alapértelmezett beállíátsok
* int pthread\_join(pthread\_t thread, void \*\*thread\_return);
  + felgüggeszti a hívó szál működését mindaddig, amíg a thread argumentumban megadott szál be nem fejezi a futását
* példaprogram
  + fordítás: ha a pthread könyvtárat használjuk, hozzá kell linkelni 🡪 -lpthread

**Szál indítása**

#include <pthread.h>

#include <stdlib.h>

#include <stdio.h>

#include <unistd.h>

void \*thread\_function(void \*arg) {

int i;

printf(„A szál indul… \n”);

for (i=1; i<=20; i++) {

printf(„%d. Hello szál világ%\n”);

sleep(1);

}\

printf(„A szál kilép…\n”);

return 0;

}

int main(void) {

pthread\_t mythread;

if (pthread\_create(&mythread, NULL, thread\_function, NULL) ){

fprintf(„stderr, „Hiba a szal letrehozasaban. \n”);

exit(1);

}

if (pthread\_joid(mythread, NULL)) {

fprintf(stderr, „Hiba a szál megvárásában.\n”);

exit(1);

}

sleep(5); // késleltetés, mert ha azonnal kilép a main,

// megszűnik a főszál, megsemmisíti a létrehozott szálat is

printf(„ A főszál kilép …\n”);

return 0;

}

## Szinkronizáció

### Külcsönös kizárás (MUTEX)

* #include <pthread.h>
* Szálszinkronizációban is használják
* Két állapot: locked, unlocked
* A mutec egyszerre csak egy szálé lehet
* **Linux alatt háromféle MUTEX:** megmondja, mi történik akkor, ha egy olyan szál próbál lefoglalni egy MUTEXET, amelynek már birtokában van
  + *gyors*
    - várakozni fog arra, hogy felszabadítsa valaki. De mivel az a valaki ő maga volt, ezért infinite loop 🡪 so sad ☹
    - deklarálás
      * makróval: pthread\_mutex\_t fastmutex = PTHREAD\_MUTEX\_INITIALIZER;
      * függvénnyel: int pthread\_mutex\_init(pthread\_mutex\_t \*mutex, const pthread\_mutexattr\_t \*mutexattr) ÉRVÉNYES MINDEGYIKRE
  + *rekurzív*
    - újra lefoglalja 🡪 inkrementális lefoglalás 🡪 ahhoz, hogy felszabaduljon, minden egyes lefoglalást fel kell szabadítani visszafelé
      * deklarálás makróval pthread\_mutex\_t recmutex = PTHREAD\_RECURSIVE\_INITIALIZER\_NP;
        + NP utótag: non-portable
  + *hibaellenőrző*
    - megnézi, hogy foglalt-e már a MUTEX 🡪 ha igen, hibával tér vissza
    - deklarálás makróval: pthread\_mutex\_t errchkmutex = PTHREAD\_ERRORCHECK\_MUTEX\_INITIALIZER\_NP;
* int pthread\_mutex\_destroy(pthread\_mutex\_t \*mutex);
  + MUTEX megszüntetése (jelenleg a Linux csak chekolja, hogy szabad-e ☹)
  + feltétele: ne legyen foglalt
* int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex);
  + MUTEX lefoglalása
  + ha épp szabad
  + a hívó szál addig lesz felfüggesztve, amíg a MUTEX nem szabad 🡪 ☹
* it pthread\_mutex\_trylock(pthread\_mutex\_t \*mutex);
  + a szál nem fog várakozni, mert a fv. kapásból visszatér, ha nem tudja lefoglalni EBUSY értékkel
* int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex);
  + unlock

### Feltételes változók (conition variables)

* #include <pthread.h>
* lehetővé teszi, hogy a kritikus szekcióban levő szálak felfüggesszék futásukat mindaddig, amíg egy erőforrásra igaz nem lesz valamilyen állítás 🡪 várakozás közben nm fogják a kritikus szekciót biztosító mutexet
* **két művelet végezhető**
  + várakozásra jelentkezés
  + esemény bekövetkezte
* **a jelzés pillanatszerű** 🡪 ha az egyik szál meghívja a várakozó fv-t, pont akkor teljesül a feltétel 🡪 lekési
  + **megoldás**:
    - olyan mutex, amit lefoglalunk, mielőtt hívjuk a várakozó fv-t 🡪 várakozás megkezdése után elengedi a mutexet
    - a jelzés kiadása előtt is lefoglaljuk a mutexet 🡪 ha éppen akkor kezd várakozni a fv. megvárjuk, míg várakozni kezd 🡪 felszabadul a mutex 🡪 utána szerezzük meg
  + **TEHÁT: a feltételes változóhoz rendelni kell egy mutexet**
* létrehozás
  + pthread\_cond\_t cond = PTHREAD\_COND\_INITIALIZER;
  + int pthread\_cond\_init(pthread\_cont\_t \*cond, pthread\_condattr\_t \*cond\_attr);
    - cond\_attr
      * NULL: alapértelmezett paraméterekkel jön létre
      * egyébként, mivel a Linux nem definiál attribútumokat a feltételes változók számára, a cond\_attr a pthread\_cond\_init figyelmen kívül hagyja
* megszüntetés:
  + int pthread\_cond\_destroy(pthread\_cond\_t \*cond);
  + egy szál sem várakozhat a változóra
* jelzés küldése
  + int pthread\_cond\_broadcast(pthread\_cond\_t \*cond);
  + előtte le kell foglalni a mutexet, majd után afelszabadítani
* várakozás:
  + int pthread\_cond\_wait(pthread\_cond\_t \*cond, pthread\_mutex\_t \*mutex);
  + előtte le kell foglalni a mutexet
  + utána elengedni
* meghatározott ideig várakozás:
  + int pthread\_cond\_timedwait(pthread\_cond\_t \*cond, pthread\_mutex\_t \*mutex, const struct timespec \*abstime);
  + abstime

### Szemaforok

* Különbség az POSIX IPC-vel szemben:
  + egyetlen szinkronizációs objektum ⬄ szemafortömb (IPC)
  + szálak és processzek között is használható ⬄ csak processzek között (IPC)
* Létrehozás
  + Névtelen szemafor létrehozása
    - int sem\_init(sem\_t \*sem, int pshared, unsigned int value);
      * pshared: ha nem 0, akkor más processzek is hozzáférhetnek
* Megnevezett szemaforok
  + sem\_t \*sem\_open(const char \*name, int oflag);
  + sem\_t \*sem\_open(const char \*name, int oflag, mode\_t mode, unsigned int value);
* Szemafor lefoglalása
  + int sem\_wait(sem\_t \*sem);
  + int sem\_trywait(sem\_t \*sem);
    - ha a szemafor értéke pozitív: mindkét fv. lefoglalja, visszatér 0-cal
  + ha nem lehet lefoglalni:
    - wait – várakozik, megszakíthatja:
      * szemafor értéke nagyobb lesz, mint nulla
      * vagy egy jelzés
    - trywait – visszatér EAAGIN értékkel
* Szemafor felszabadítása
  + int sem\_post(sem\_t \*sem);
* Szemafor aktuális érétkének lekérdezése:
  + int sem\_getvalue(sem\_t \*sem, int \*sval);
    - ha lefoglalták
      * 0
      * vagy negatív szám, amelynek abszolút értéke megadja a rá várakozó processzek számát
* Megszüntetés
  + Névtelen szemafor
    - int sem\_close(sem\_t \*sem);
  + Megnevezett szemafor lezárása, majd megszüntetése
    - int sem\_close(sem\_t \*sem);
    - int sem\_unlink(const char \*name);