# 22. Tétel

Ismertesse a párhuzamosság kezelésének eszközeit a Linux kernelben!

## Párhuzamosan futtatható szálakkal szembeni elvárások

* a kernelben is párhuzamosan futnak a műveletek
* emiatt itt is szükség van a folyamatok szinkronizációjára, erőforrások hozzáférésének ütemezésére

## Atomi műveletek

### Legegyszerűbb, leghasznosabb

* ha a művelet atomi, nincs szükség szinkronizációra, mivel a processzor egy műveletből elintézi a dolgot, vagy olyan műveletsorból, ami nem szakítható meg
* az, hogy melyik művelet atomi, platformfüggő
* a kernel biztosít platformfüggetlen atomi műveleteket
  + kernelfejlesztők dolga, hogy ezek tényleg atomiak legyenek az adott processzorra
* egyszerű műveletek
  + de nem minden processzoron egy-egy művelet csak
  + csak egész számokon értelmezett műveletek
  + érték növelő/csökkentő/tesztelő stb. műveletek

### Használat

* két típus adattípustól függően
  + c**sak atomic\_t típusú változókat fogadnak be**
    - inicializálás csak makróval: ATOMIC\_INIT()
    - static atomic\_t szamlalo = ATOMIC\_INIT(1);
  + **bitműveletek**: csak unsigned long – platformfüggő
    - egy memóriacímmel megadott memóriaterületen végzik el a műveleteket
    - csak annyi bájtra érvényesek, amennyi az unsigned long az adott processzoron
    - bájtsorrend értelmezése szintén architektúra függő

## Ciklikus zárolás (spinlock)

### Működése

* folyamatosan egy CPU-t terhelő ciklusban kísérletezik a zárolás megszerzésével, amíg meg nem szerezte
* csak rövid szakaszokra, különben nagy CPU használatot adna
* a zárolással védett szakasz nem tartalmazhat sleep()-et 🡪 a scheduler új folyamatnak adja meg a vezérlést, ami szintén megpróbálhatja megszereni a zárolást 🡪 örök várakozás, halál, és fagy ☹
* megszakításkezelőben erősen ellenjavalt

### Használata

* létrehozása: spinlock\_t lock;
* init: spin\_lock\_init(&lock);
* egy lépésben a létrehozás, és init makróval: static DEFINE\_SPINLOCK(lock);
* lefoglalás: spin\_lock(&lock);
* felszabadítás: spin\_unlock(&lock);
* kritikus szakasznál nem hívódhat meg a megszakításkezelő sem 🡪 átmenetileg le kell tiltani: spin\_lock\_irqsave(&lock, flags); MŰVELET; spin\_unlock\_irqrestore(&lock, flags)
  + ezek makrók, tehát nem érték szerinti átadás történik, hanem ténylegesen átadjuk a változót is (ezért nem pointer a flag)

## Szemafor

* összetettebb 🡪 több erőforrást eszik 🡪 rövid szakaszokra inkább spinlock, csak komolyabb esetekre
* működése során sleep()-et használ 🡪 nem használható olyan helyen, ahol ez le van tiltva (pl. megszakításkezelőben, spinlock által védett szakaszok)

### Használata:

* létrehozása: struct semaphore sem;
* init: void sema\_init(struct semaphore \*sem, int val);
  + val: semaphore kezdőértéke
* létrehozás + init: static DEFINE\_SEMPHORE(sem);
* lefoglaláskor várakozásra kényszerülhet a fv. Attól függően, mi szakíthatja meg …
  + egyszerű lefoglalás: void down(struct semaphore \*sem);
  + jelzás megszakíthatja a várakozást: int down\_interruptible(struct semaphore \*sem);
  + ha nem akarunk várakozni, csak checkolni lefoglalható-e: int down\_trylock(struct semaphore \*sem);
  + időkorlátos várakozás: int down\_timeout(struct semaphore \*sem, lon jiffies);
    - jiffy: időkorlát mértékegysége (the number of ticks that have occured sincet he system booted)
* semaphore elenegedése: void up(struct semaphore \*sem);

## Mutex

### Megkötések kernelben

* csak a lefoglaló szabadíthatja fel
* rekurzív foglalás, vagy többszörös felszabadítás nem engedélyezett
* nem használható megszakítás kontextusában (sem HW sem SW megszakítást kezelő függvényben)

### Használata

* definiálás: struct mutex asdf;
* init: mutex\_init(mutex);
* létrehozás + init: DEFINE\_MUTEX(mutex);
* lefoglalás: void mutex\_lock(struct mutex \*lock);
* lefoglalása, ha várakozási jelzéssel megszakíthatóvá akarjuk tenni: int mutex\_lock\_interruptible(stuct mutex \*lock);
* trylock: int mutex\_trylock(struct mutex \*lock);
* felszabadítása: void mutex\_unlock(struct mutex \*lock);
* lefoglaltság ellenőrzése: int mutex\_is\_locked(struct mutex \*lock);

## Olvasó/író ciklikus zárolás és szemafor

### Különböző zárolások

* kritikus szakasznál érdemes külön kezelni, hogy a védett változót írjuk, vagy olvassuk 🡪 növeli a rendszer teljesítményét
  + olvasást többen is tehetik egyszerre – megosztott zárolás
  + írni csak egyvalaki írhatja – kizáró zárolás
* elnevezés: író/olvasó spinlock / read/write spinlock

### Használat

* létrehozás: rwlock\_t rwlock;
* init: rwlock\_init(&rwlock);
* létrehozás + init: static DEFINE\_RWLOCK(rwlock);
* olvasás (megosztott) zárolás, feloldás:
  + read\_lock(&rwlock);
  + read\_unlock(&rwlock);
* írási (kizáró) zárolás/feloldás:
  + write\_lock(&rwlock);
  + write\_unlock(&rwlock);
* író/olvasó semaphore létrehozás: struct rw\_semaphore rwsem;
* init: void init\_rwsem(struct rw\_semaphore \*sem);
* lefoglalás olvasásra:
  + void down\_read(struct rw\_semaphore \*sem);
  + int down\_read\_trylock(struct rw\_semaphore \*sem);
* felszabadítás olvasás után: void up\_read(struct rw\_semaphore \*sem);
* lefoglalás írásra:
  + void down\_write(struct re\_semaphore \*sem);
  + int down\_write\_trylock(struct re\_semaphore \*sem);
* felszabadítás írás után: void up\_write(struct rw\_semaphore \*sem);
* read zárolásból nem lehet write zárolásba menni 🡪holtpont a kernelben 🡪 ☹
* DE: írási write zárolás visszaléptethető read zárolásra

## Nagy kernelzárolás (BIG KERNEL LOCK, HAIL CTUHLU!!444!)

* globális rekurzív ciklikus zárolás
* nem javasolt, mert jelentősen korlátozza a kernel működését
  + rontja a valósidejűséget
  + rontja a párhuzamos működést
* már történelem
  + 2.0-s kernelnél vezették be, amikor megjelentek a multiprocesszoros rendszerek támogatása
  + lehetséges konkurenciaproblémák könnyen megelőzhetőek, ha a kritikus szakaszoknál a teljes kernelt lezárjuk
  + brute force – nem kellett gondolkodni
  + 2.6.37-es kernelben purgálták
* ma is be lehet kapcsolni a forráskódban, de nem szeretik
* zárolás: lock\_kernel();
* feloldás: unlock\_kernel();