Pepi tételek kidolgozása

1. Az operációs rendszer és a HW kapcsolata. OS alkalmazásának előnyei és hátrányai.
HW hatása a párhuzamos végrehajtásra, HW architektúrák, cache koherencia, spekulatív végrehajtás, megszakítás és fajtái.
Valós-idejű végrehajtás és az OS kapcsolata

Operációs rendszer

Hardver és alkalmazói SW közötti interfész. Feladatai az erőforrások és aktivitások menedzselése és koordinálása, lehetővé teszi az alkalmazások futtatását és egy alkalmazói programozói interfészt nyújt az OS szolgáltatásainak eléréséhez.

OS és HW kapcsolata

A modern beágyazott rendszerek szoftverei sok szempontból párhuzamosnak és esemény vezéreltnek mondhatók, az ilyen SW-ek írása komoly kihívást jelent. A PEP (Párhuzamos és eseményvezérelt programozás) egyben egy komplexebb hardvert is igényel (több végrehajtó egység egy platformon, különböző heterogén architektúrák), és megvalósítása könnyebb OS felhasználásával. (Természetesen a PEP lehetséges OS nélkül is, különböző programszervezésekkel, pl. Round-Robin, Round-Robin megszakítással, spagetti kód…)

Valós idejű OS-ek

Felépítésükből adódóan adott szolgáltatások képesek valós időben futni. PL. egy interrupt esetén a Callback függvény adott időintervallumon belül lefut. Nem csak az OS-nek, hanem az adott alkalmazásnak és kódnak is valós idejűnek kell lennie. Linux és Windows esetében nem beszélhetünk ilyen garanciáról, viszont vannak olyan kiterjesztések, amik ezt lehetővé teszik (pl. RTLinux, Ardence RTX Windowshoz)

Valós idejű rendszerek esetében a rendszernek adott időn belül adott valószínűséggel válaszolni kell, másképpen a válasz hibás. Fajtái.

Hard real-time rendszerek: A valószínűség 1, a rendszer nem késhet, ha nem válaszol időben a válasz rossz. PL. ha egy autó kormányát elforgatjuk vagy lenyomjuk a féket, akkor adott időn belül el kell kezdeni a autónak fordulni vagy lassítani, egyébként katasztrofális következmények lehetnek.

Soft real-time rendszerek: valószínűség nem 1, de nagyon közel van az 1-hez. Bizonyos esetekben megengedhető, hogy a rendszer késik, nem fog katasztrofális következményekkel járni.

Hardver architektúrák és Cache koherencia

Miért lehet szükségünk több azonos végrehajtó egységre egy platformon? Válasz: Félvezető technikában keresendő Órajel frekvencia tovább nem nőhet, fogyasztás tovább ne nőjön → más megközelítés kell a hatékonyság növeléséhez. Ugyanakkor sok feladat nem hajtható hatékonyan végre egy általános architektúrákon, a különböző végrehajtó egységek (FPGA, DSP, CPU) előnyeit kihasználva hatékonyabb rendszereket építhetünk.

Kulcskérdés az architektúra:

- Milyen utasítások vannak, és azokat hogyan hajtja az architektúra végre?
- Hogyan vannak a végrehajtó egységek összekötve?
- Hogyan érik el a végrehajtó egységek a kód- és adatmemóriát?
- Hogyan érik a végrehajtó egységek a perifériákat → kulcskérdés lesz a megszakítás.

Nézzük a különböző architektúrákat

- **Single CPU:** Egyetlen végrehajtó egység egy platformon. Beágyazott rendszerekben még többnyire ez a jellemző. DMA ha van, párhuzamosan kezeli a memóriát a CPU-val, versenyhelyzet alakulhat ki a CPU és a DMA között → fontos kérdés a Cache koherencia
  - Tiltsuk le az egész Cache-t a transzfer alatt? teljesítmény :(
  - Tiltsuk le a Cache-t a DMA által használt területre → Memory Management Unit
  - HW támogatás (Cache koherens DMA)

  A Cache koherencia mindig fontos kérdés, ha több végrehajtó egység van

- **SMP (Symmetric multiprocessing)**
  - Több azonos végrehajtó egység (pl. az Intel többmagos processzorai)
  - Végrehajtó egységek saját Cache hierarchiával → koherencia biztosítva
  - Memóriát minden végrehajtó egység azonos tulajdonságokkal (sebesség, késleltetés) éri el
  - Fontos az OS, másképpen csak egy végrehajtó egység látható
- **Non-Uniform Memory Accsess (NUMA)**
  - memória egyes részei “közelebb vannak” az egyes végrehajtó egységekhez
  - összefüggő fizikai memória
  - Cache koherencia : Beszélhetünk Cache koherens, un. ccNUMA, egyéb esetben az OS-nek vagy a felhasználói programnak kell a koherenciát kezelni
  - OS támogatás mindenképpen kell, egyébként csak 1 mag látható
- **Elosztott rendszer**
  - Nincs közös memória, lazán csatolt Node-ok kommunikálnak egymással kommunikációs interfészen keresztül. A kommunikáció SW alapú
  - Minden Node tartalmaz végrehajtó egységet és memóriát
  - Az adatküldés alacsonyabb megbízhatóságú (üzenet elvesztése, duplikálódása, sorrendek módosítása), és kisebb sávszélességű mint a memóriák használata esetében.

A Linux és FreeRTOS számos különböző architektúrát támogat. Az OS-ek hátránya a központi szerep, ugyanis az OS megfelelő működése rendkívül fontos a rendszer szempontjából

Spekualtív végrehajtás

A Compiler és CPU egy szekvenciális kódban garantálja, hogy a program konzisztens marad. A Compiler és a CPU átrendezheti az utasítások végrehajtási sorrendjét, ami nehezen detektálható hibákat okozhat párhuzamos és eseményvezérelt programokban. “Volatile” kulcsszó nem segít, hiszen az a Compilernek szól, azt a CPU még átrendezheti. Megoldás: Memory Fence vagy Barrier. Megadott helyekre a memória hozzáférést explicit módon sorrendező **gépi utasításokat** rakunk (amiket nem lehet megszakítani).

Megszakítások és ezek típusai

- Hardveres megszakítás: Periféria használja értesítésre, jelezve a processzor felé kiszolgálási igényét. Egy periféria számos okból kérhet megszakítást, melyek különböző módon hatnak a processzor működésére.
- Kivétel: Az OS és alkalmazások futása közben történik valamilyen esemény, ami hat a processzor működésére (pl. túlcsordulást, 0-val osztás). Általában valamilyen szoftveres kezelési igényt jelent, pl. egy osztást végző függvényben kell azt az esetet, ha a nevező zérus lenne.
- Szoftver esemény, pl. rendszerhívás vagy valamilyen speciális utasítás futtatása. A szoftver hat a CPU működésére.

Modern operációs rendszerek megszakítás vezéreltek

2. Védelmi szintek és rendszerhívás, kontextus fogalma. Operációs rendszerek és azok felépítése és OS választás szempontjai. Linux és FreeRTOS főbb eltérései és hasonlóságai.

Védelmi szintek

A védelmi szintek a végrehajtható utasításokat korlátozzák az adott működési módtól függően. Jellemzően kettő szintet használunk, egyik az ún. *Kernel mode*, mely korlátlan hozzáférést biztosít, szemben a *user mode*-al. A védelmi szintek az alacsony szintű CPU erőforrásokhoz történő hozzárését szabályozzák, mint pl. CPU regiszterek, periféria és memória hozzáférés. A védelmi szintek közötti megszakítással válthatunk.

Rendszerhívások és kontextus

A rendszerhívások az OS bizonyos szolgáltatásaink használatát teszik lehetővé. A rendszerhívás megszakítja az éppen futó alkalmazói programot, és átadja vezérlés az OS kernelnek (állapotmentés és visszaállítás, kontextus csere). Ezt követően a kernel elvégzi a feladatát, majd visszaadja az alkalmazói programnak (ismét egy állapotmentés, visszaállítás és kontextus csere történik). A rendszerhívás folyamata erősen implementáció függő.

Mi a rendszerhívás következménye? Nagy overhead-et jelent, hiszen egy rendszerhívás több állapotmentéssel és kontextus cserével jár, emellett a védelmi szintek között is váltatunk kell. A rendszerhívás nem vermet használ, a paraméterek átadása teljesen máshogy működik mint pl. egy függvény esetében.

Rendszerhívás a programozó szemével

A programozó az API függvényt hívja meg a használt programozási nyelven. Az API teljes egészében elrejti a rendszerhívás részleteit. Az API megvalósítása a felhasználói kódba fordul bele, tulajdonképpen felhasználói módban fut, és kiváltja a rendszerhívást. Nagyon indokolt esetben készíthetünk új rendszerhívásokat vagy módosíthatjuk a meglévőket, valamint rendszerhívásokat hívhatunk direktben is (de nem ajánlott).

Kontextus részei

- CPU regiszterei: státus regiszter, munkaregiszterek, szegmens regiszterek…
- MMU beállítása az adott alkalmazás memória eléréséhez
- Különböző alkalmazásspecifikus HW beálíltások

Kulcskérdés, hogy mit érdemes elmenti és visszaállítani egy kontextusváltás során

OS kiválasztásának szempontjai

- Platformtól függően, pl. MMU alkalmazása szükséges?
- Piaci modellek: ingyenes (milyen felhasználásra és feltételekkel), bizonyos tulajdonságok fizetősek de van ingyenes verzió is, fizetős (fejlesztői licensz, darabár)
- Hard real-time, és ha igen milyen paraméterekkel
- Milyen tanúsítványai vannak? (Megbízhatóság, kódminőség, biztonság)
- Kód hozzáférhetősége: egyáltalán nem férhetünk hozzá, részben férhetünk hozzá, teljes egészében hozzáférhető

Beágyazott OS-ek komplexitása

- **Egyszerű kernel**: 5-100k program memória, kernel egybefordul az alkalmazással, MMU és védelmi szintek nem kellenek. Például: FreeRTOS
- **Közepes komplexitás**: 40k-2M program memória, kernel egybefordul az alkalmazással, MMU és védelmi szintek alkalmazása OS függő (pl. Nucleus)
- **Nagy komplexitás**: 500k+ program memória, lehet külön telepítő, fájlrendszerből bootolás, MMU és védelmi szintek kellenek. PL: Windows és Linux

High_end vs. Low_end operációs rendszerek

Szempont	High-end (pl. Linux)	Low_end (pl. FreeRTOS)
Alkalmazási terület	Általános célú felhasználás	Mikrokontrollerek számára
Támogatás	Folyamat, szál, virtuális memória	Csak szál és memóriavédelem
Memória menedzsment és védelem	védelmi szintek és MMU kellenek	opcionális
Előnyök	hatékony, gyors fejlesztés összetett alkalmazások esetén is	egyszerű, hatékony futási időben egyszerű alkalmazások esetén, jól tesztelhető
Hátrányok	nagy komplexitás és futási idő, nagy erőforrás igény, tesztelési nehézségek	nincs védelem OS szinten, MPU inkább egy periféria ami megszakításokat csinált
OS jellemzői	OS elkülönül, flexibilis külön fordítható, futtatható, telepíthető OS alkalmazásokkal és IO-val kommunikáció: rendszerhívás	OS és alkalmazás nem különül el, egyetlen végrehajtható fájlba fordulnak OS alkalmazásokkal, IO-val kommunikáció: függvényhívás
Hardverek kezelése	Driverekkel kernel módban, megszakításokat az OS fogadja	Speciális programkönyvtárak

OS felépítése

Réteges szerkezet: legalsó szinten a valósi CPU és perifériák által megvalósított valódi gép található

- Kernel: Operációs rendszer alapvető funkcióit tartalmazó rendszermag. Feladatai többek között Feladatok kezelése, memória kezelése, védelmi és biztonsági funkciók
- Hardver közeli rétegek: Driverek valamilyen hierarchiába rendezve. Feladat alacsony szintű HW kezelés, pl. egér, billentyűzet...
- Rendszerprogramok: Fájlrendszer, magasabb szintű hálózatkezelés, parancsértelmező
- Felhasználói programokból érkező rendszerhívások fogadó felületét megvalósító réteg (API megvalósítása és leképezése rendszerhívásokra, védelmi szintek váltása, jogosultságok ellenőrzése, kernel feladatok elvégzése)

Kernelek típusai

- **Monolitikus kernel**: Ebben az esetben az összes szükséges funkciót egyetlen kódba fordítjuk össze. Nem flexibilis, mivel egy másik platformon a szükséges HW elemeket ismét bele kell fordítani a rendszerbe, ezért beágyazott rendszerek esetén nem nagyon használjuk. Ha egy komponens meghibásodik a teljes OS hibás lesz.
- **Moduláris kernel**: Minimális kernel + igény szerint utólagosan betölthető kernel modulok, emiatt flexibilis. Ez jellemző Windowsra és az újabb Linuxokra. Egy komponens hibája a teljes OS hibáját jelenti.
- **Mikrokernel**: Minimális funkciókkal rendelkező kernel, ahhoz kliens-szerver architektúrában csatlakozó szolgáltatások. Többnyire 3 védelmi szintet használ, a kernel még a hozzá csatlakozó szolgáltatásoktól is védett. Hátránya, hogy több erőforrást igényel. PL. MAC OS.
- **Exokernel**: új kutatási terület, hatékonyság növelése érdekében direkt hozzáférés a periférikához

Linux felépítése

- Alapesetben egy monolitikus kernel, és modulárisan betölthető kernel modulok alkotják.
- Képes monolitikus kialakításban is működni, ekkor tulajdonképpen minden drivert és szolgáltatást belefordítunk a kernelbe. Ekkor nem flexibilis, de erre nincs is szükség. Ezt tipikusan beágyazott rendszerek esetén alkalmazzuk.

Windows

Moduláris kernel.

3. Szekvenciális és párhuzamos programozás. Párhuzamos programozás története. Tipikus párhuzamos feladatok. Amdahl és Gustafson törvényei és következményeik. Eseményvezéreltség.

Szekvenciális programozás

A kód egymás után végrehajtott utasítások sorozata, amit az utasításokban felhasznált adatok vezérelnek. A program adatvezérelt, nem csinálja mindig ugyanazt, a bemenő adatoktól függően változnak a végrehajtott utasítások és a program outputja. Az adatváltozást csak akkor veszi észre az adott program, ha az adott adatot felhasználja valamilyen utasításban.

A legtöbb probléma szekvenciálisan csak nehezen oldható meg, A világ erősen párhuzamos, és a különböző párhuzamos feladatok erősen összefüggenek → akkor oldjuk meg a feladatot párhuzamosan!

Párhuzamos programozás számítógépeken

A számítógépeink már régóta képesek programok "párhuzamos" végrehajtására. A UNIX több mint 40 éves OS, pontosan erre készült. A mai OS-ek is lehetővé teszik a párhuzamos végrehajtást, pl. a szakdolgozatom írása közben zenét hallgattam, amik egymástól független feladatok.

Hogyan futnak a feladatok párhuzamosan valójában? Nincsen probléma amennyiben több végrehajtó egységünk van, mint ahány feladat. Kevesebb, vagy csak egyetlen végrehajtó egység esetén a kulcskérdés az erőforrások (CPU, memória, regiszterek, perifériák) optimális megosztása.

Párhuzamos programozás története

1950-1965: Egy számítógép volt cégenként, egyetemenként, melyet több, egymást nem ismerő felhasználó használt egymástól függetlenül. Számítógépek nem kommunikáltak egymással, elsőraban az erőforrások megosztása volt a cél.

1965-1980: Egy számítógép tanszékenként/ osztályonként. Kevesebb felhasználó jutott egy feladatra, és megjelent a feladatok egymástól függése is: a felhasználók ismerték egymást, tudatosan együtt dolgoztak. Megjelenik a távoli kommunikáció és az internet elődje, és az első beágyazott rendszerekhez hasonló megoldások, pl. kommunikációs a párhuzamos fizikai környezettel, és annak befolyásolása, szabályozása.

1980-1995: 1+ számítógép felhasználónként. Ekkor jelentek meg és kezdtek elterjedni a személyi számítógépek, és jellemző lett a párhuzamos és erősen összefüggő feladatok megoldása. Emellett elterjedt a számítógép alapú kommunikáció, és az internet.

1995-: több számítógép felhasználónként. A preemtív asztali operációs és szenzorrendszerek (TCP/IP) mellett ekkor kezdtek elterjedni a beágyazott rendszerek is: Mikrovezérlők a mindennapi életben (lakásban, autóban), és beágyazott OS-ek megjelenése és elterjedése.

Párhuzamos program (konkurens program, konkurens rendszer)

Olyan rendszert jelent, melyben a teljes feladatot több, párhuzamosan futó részfeladat együtt valósítja meg. A feladatok/részfeladatok önmagukban szekvenciális programrészek, melyek együttesen oldják meg a teljes feladatot, önállóan értelmes működésre többnyire nem képesek. Adott részfeladat végrehajtásához az adott részfeladatot egy végrehajtó egységhez kell rendelnünk.

Típusfeladat 1

Legyen N db részfeladatunk és M db végrehajtó egységünk (N és M összemérhető számok). Tehát összességében sok végrehajtó egység áll rendelkezésünkre, de a feladatot csak nehezen tudjuk részfeladatokra bontani. Itt a kihívást tulajdonképpen a a teljes feladat különböző részekre bontását jelenti.

Típusfeladat 2

Legyen N db részfeladatunk és 1 vagy N-nél sokkal kevesebb végrehajtó egységünk. Tehát kevés végrehajtó egységünk van, és sok párhuzamosan végrehajtandó, jól beazonosítható részfeladat. A kulcskérdés az erőforrások megosztása és a teljes feladat hatékony és helyes megvalósítása (ez tipikus probléma beágyazott rendszerekben).

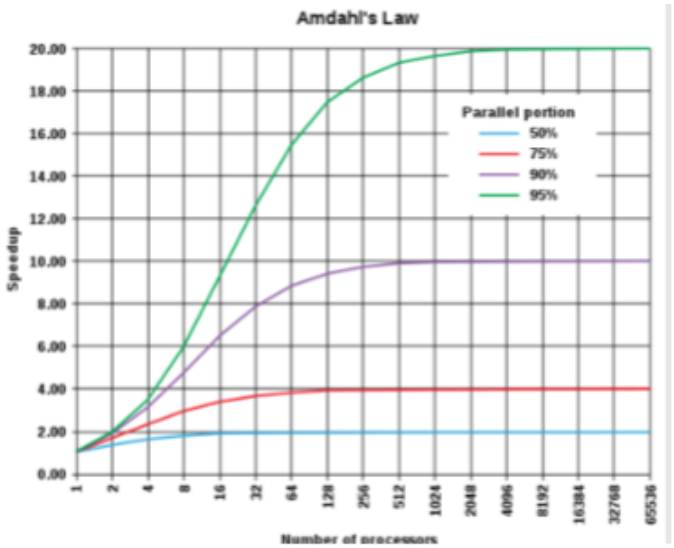
Amdahl törvénye

Egy feladat végrehajtási sebessége hogyan függ a végrehajtó egységek számától? A leképzés nem 1:1, 1-ről 4 CPU mag nem fog 4x végrehajtási sebességet okozni.

Amdahl törvénye

$$S_{latency}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

$$S_{latency}(s) \leq \frac{1}{1-p}$$



$S_{latency}(s)$: végrehajtás gyorsulása a maximálisan elérhető gyorsulás függvényében. Pl. 4 CPU mag esetében 4;

s: A maximálisan elérhető gyorsulás a kód azon részére, ami párhuzamosítható.

p: Azon kód aránya, ami párhuzamosítható.

Láthatóan a párhuzamosítási törekvések egy idő után kevés eredménnyel járnak.

Gustafson törvénye

Amdahl törvénye kicsit konzervatív, nem veszi figyelembe, hogy a probléma mérete is változik. Gustafson törvénye realisztikusabb, jobban tükrözi a valóságot.

$$S_{latency}(s) = 1 - p + s \cdot p \tag{1}$$

Lineáris növekedést láthatunk. A meredekség egynél kisebb, és a végrehajtó egységek számától, valamint a párhuzamosíthatóságtól függ.

Eseményvezéreltség

Egy esemény tipikusan valamilyen lezajló változás vagy történés a rendszer életében. A program futása függ a program közben lezajló eseményektől, a programunk a különböző eseményekre. Egy esemény lehet:

HW megszakítás: input a felhasználótól (egérkattintás), input a külvilágtól (detektált meghibásodás), adatcsomag érkezése kommunikációs interfészen

SW megszakítás: program belső állapotában lezajló tudatos változás, például egy rendszerhívás

Kivétel: túlindexelés, túlcsordulás, osztás 0-val

Az esemény a kódban összetett adattípusként, tipikusan egy struktúraként jelenik meg.

4. Feladat fogalma. Feladat egyszerűsített memóriaképe, user és kernel területen lévő információk, egyszerűsített állapot-átmeneti ábra. Linux és FreeRTOS állapot-átmeneti ábrájának ismertetése

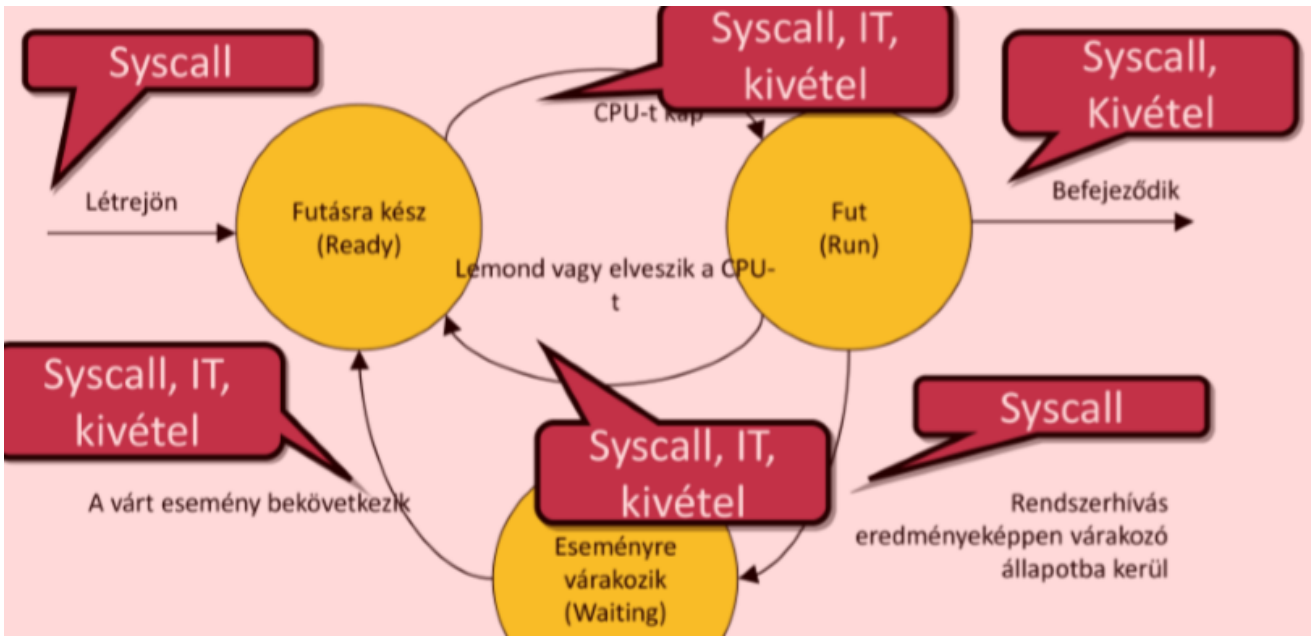
Feladat fogalma

A feladatok/részfeladatok önmagukban szekvenciális programrészek, melyek együttesen oldják meg a teljes feladatot, önállóan értelmes működésre többnyire nem képesek. Adott részfeladat végrehajtásához az adott részfeladatot egy végrehajtó egységhez kell rendelnünk.

A feladat több mint program! Aktív, állapottal rendelkezik (az állapotok nagyon OS függőek, tipikusan fut, futásra kész, és várakozás állapotok vannak), és különböző memóriaterületekkel rendelkezik, melyek előéletétől függően különböző adatokkal vannak feltöltve: kód- és adatmemória, Stack, Heap és szabadon felhasználható memória területek.

Állapotátmenetek taszkokban

A modern OS-ek eseményvezéreltek, így valamennyi átmenet IT hatására történik.

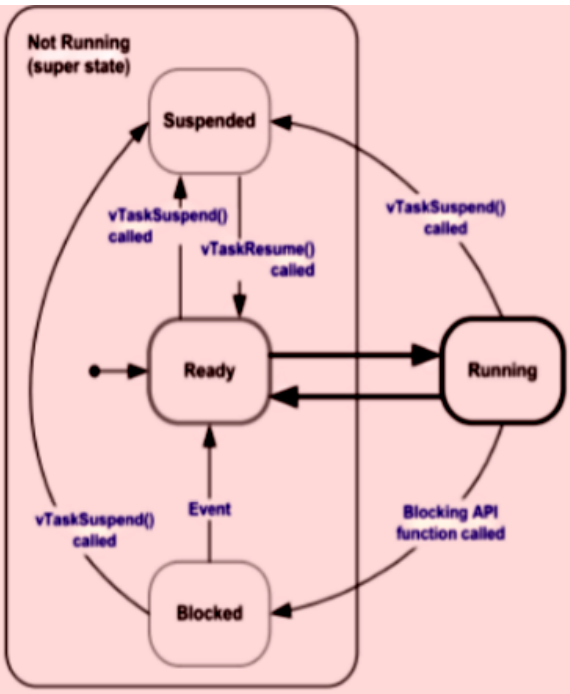


FreeRTOS task állapotok

Az állapotokat a megfelelő API hívásokkal kérdezhetjük le

Le lehet kérdezni megfelelő API hívásokkal

- Running: fut, a saját állapotára kérdez rá
- Ready: futásra kész
- Blocked: várakozik
- Suspended: felfüggesztett állapotban van
- Deleted: törölve lett, de a hozzá tartozó adatstruktúrák törlése még nem történt meg

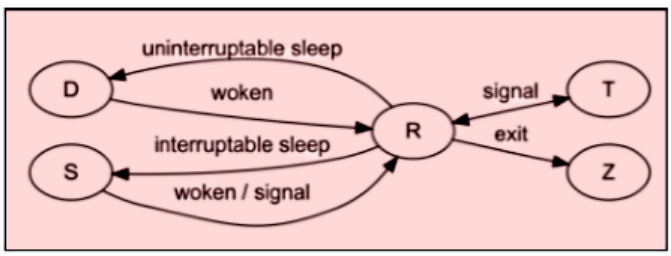


Mastering the FreeRTOS Real Time Kernel - a Hands On Tutorial Guide

Linux állapotok

A „man ps” paranccsal megtudható…

- D uninterruptible sleep (usually IO)
- R running or runnable (on run queue)
- S interruptible sleep (waiting for an event to complete)
- T stopped by job control signa
- t stopped by debugger during the tracing
- W paging (not valid since the 2.6.xx kernel)
- X dead (should never be seen)
- Z defunct ("zombie") process, terminated but not reaped by



Feladat leíró

Tulajdonképpen egy adatstruktúra, melyben egy adott feladattal kapcsolatos adatokat tárolunk az OS-en belül, ez alapján végi az OS a feladatok kezelését. A feladatok bizonyos részeihez rendszerhívásokkal hozzáférhetünk. Mit tartalmaz a struktúra? Csak néhány példa: Task ID, állapot, Utasítás számláló, CPU regiszterek, ütemezéssel és memória kezeléssel kapcsolatos információk, jogosultságok

5. Feladatok ütemezése. Ütemezés időskálái. Nem preemptív és preemptív ütemezés és összehasonlításuk. Ütemező választás. FIFO, RR ütemezők és azok összehasonlítása. Prioritásos ütemezők, egyszerű és többszintű sorok ütemezők és tulajdonságaik. FreeRTOS ütemezője.

Feladatok ütemezése

Az ütemezés során a feladatunk, hogy több futásra kész feladat közül kiválasszuk a következő futó feladatot. A várakozó feladatokat többnyire feladat sorokban (task queue) tároljuk, ami legegyszerűbb esetben lehet egy FIFO ami TBC típusú struktúrákra mutató pointereket tartalmaz. Az ütemező algoritmus ezeken a feladatokon dolgozva végzi el a feladatát.

Ütemezés időskálái

Rövidtávú ütemezés

A futásra kész sorból választja ki a következő futó állapotba kerülő feladatot. Minimum 10-20 ms-onként hívódik meg.

Hosszútávú ütemezés

Hosszú távú ütemezést tipikusan akkor alkalmazunk, amikor sokkal több feladatunk van, mint amennyit hatékonyan párhuzamosan végre tudunk hajtani. Az ütemező percenként vagy ritkábban fut le, és ismernie kell az egyes feladatok okozta terhelést. Erre jó példa 3-4 nagy fájl másolása számítógépről diszkre. Beágyazott rendszerekere nem jellemző.

Középtávú ütemezés

Swapping során a rendszerben lévő feladatok egy részeit kiírjuk a háttértárra, ezáltal több feladat fér el a fizikai memóriában, és egy futó feladatra is több memória jut. MI történik ha egy éppen futtatandó feladat bizonyos része még a háttértáron vannak? A taszk nem tud addig futni, ameddig a szükséges részek vissza nem kerülnek a fizikai memóriába. Ezt a folyamatot irányítja a középtávú ütemezés. Beágyazott rendszerekben előfordul, viszont nem összeegyeztethető a real-time működéssel.

Preemptív és nem preemptív ütemezés

Preemtív ütemezés

A futó taszktól az OS elveheti a CPU-t. Ez tipikusan a modern OS-ben jellemző. Ez bizonyos kernel feladatokra nem feltétlenül igaz, bizonyos kernel feladatok nem megszakíthatók. Ennek vannak következményei pl. a valós idejűség szempontjából.

Nem preemptív ütemezés

A futó feladatnak le kell mondania a CPU-ról, vagy eseményre kell várnia ahhoz, hogy más feladat tudjon futni. Nagyon nagy hátrány, hogy a teljes rendszer működése az egyes alkalmazásoktól függ, tehát ha a futó feladat hibás, például végtelen ciklusba kerül, akkor a teljes rendszer működésképtelenné válik. Milyen OS az ilyen?

Ütemező kiválasztása és ütemezési algoritmusok

Nincsen univerzális ütemező algoritmus, éppen az adott feladatnak megfelelő ütemezést kell kiválasztanunk. Erre számos OS lehetőséget ad. Az ütemezési algoritmusok ismertetésénél feltételezzük, hogy a rendszerben egyetlen végrehajtó egység van, egyszerre egyetlen feladat tud futni, és a futásra kész feladatokat valamilyen várakozási sorban tároljuk, és a feladatok CPU és IO löketekből állnak (~10ms).

FIFO

A FIFO a legegyszerűbb ütemezési algoritmus, melynek implementációja egy feladat leíróra mutató referenciákat tároló FIFO. Definíciószerűen nem preemptív, és nagy lehet az átlagos várakozási idő is, ami erősen függ a löketektől. Az eljárás kis adminisztrációs overhead-el jár. Problémát okozhat a Convoy -hatás, a később beérkező feladatoknak várniuk kell a korábban bejövő feladatokra.

Round Robin

Időosztásos rendszerekre találták ki az egyereű FIFO ütemezési problémáinak javítására. Lényegében a FIFO-t jelenti egy óramegszakítással kiegészítve, a gyakorlatban periodikus óramegszakítást alkalmazunk. Az algoritmus preemptív és jobb az átlagos válaszideje a FIFO-nál.

Prioritásos ütemezők

Prioritásos ütemezés során a taszkokhoz különböző prioritást (fontosságot) rendelhetünk. A prioritás lehet külső vagy belső, attól függően, hogy az Operátor vagy maga a taszk állítja be (külső), vagy a rendszer adja a prioritást(belső). Belső prioritások esetében a rendszer a használt memória, erőforrások és időkorlátok alapján adja meg a prioritásokat. Beszélhetünk még statikus és dinamikus prioritásról is: statikus esetben a prioritások még a tervezés során dőlnek el, és változatlanok maradnak a program futása során, ellentétben a dinamikus prioritással.

Beágyazott rendszerek esetén egy prioritási szinten egy feladat található, különböző operációs rendszerekben azonban egy prioritási szinten tetszőleges számú feladat lehet.

Többszintű prioritásos ütemezők

Ebben az esetben minden prioritási szinthez egy futásra kész várakozási sort találunk. A prioritások meghatározásról és az egyes szinteken alkalmazott ütemező algoritmusokról nem mond semmi, ezek implementáció függők.

Mi határozza meg a prioritási szintek számát? Többnyire 8-16-32 szintet használunk. Ha a szintek túlságosan sok, akkor átmehetünk egyszerű prioritásos ütemezőbe (minden szinten egy feladat, tipikusan beágyazott rendszerek esetén) ezzel együtt az algoritmus komplexitása is jelentősen nő. Tipikusan időosztásos RR ütemezést szoktunk minden szinten alkalmazni.

Prioritási szintek feladatokhoz rendelése (néhány példa)

A legmagasabb prioritással jellemzően a real-time feladatok rendelkeznek. Ezeket követik a rendszer feladatok, majd a különböző on-line (interaktív) feladatok.

Problémák prioritásos rendszerekben

Kiéheztetés: Egyszerű prioritásos rendszerben (minden szinten egy taszk) az alacsony prioritású taszkok éhezhetnek, például ha a magasabb szinteken túl sok vagy sok gyakran futó taszk van. Ez túlterhelést okoz az alsóbb szinteken, mindenképpen kerülendő. A kiéheztetés elkerülhető pl. a prioritási szintek közötti időosztással, például ha egy adott szinthez a CPU idő adott %-át rendeljük.

Mikor fut le az ütemező?

Az ütemezők jellemzően valamilyen külső (pl. óraütés, vagy valamilyen HW-es megszakítás) vagy belső esemény (erőforrás felszabadul, futó feladat várakozó állapotba kerül vagy lemond a futás jogáról, vagy valamilyen kivétel történik) hatására.

FreeRTOS ütemezője

Konfigurálható ütemező több különbféle opcióval, megadott makrókkal a FreeRTOSConfig.h header fájlban. Használhatunk többszintű sorokat időosztással, prioritásos időosztás nélküli ütemezést, vagy kooperatív ütemezést.

6. Feladatok ütemezése általában. Többprocesszoros ütemezés és annak kapcsolata az egyprocesszoros esettel, processzor affinitás, terhelés elosztás. Linux ütemezője.

Többprocesszoros ütemezés

Heterogén processzoros rendszerek esetén a feladat mai napig egy komoly kihívást jelent. A következőkben a homogén többprocesszoros rendszerek ütemezeséi lehetőségeit ismertetem a következő feltételekkel:

- Az architektúra lehet SMP vagy NUMA a memória szempontjából, vagy akár azok keveréke is.
- Egy adott periféria egy adott processzorhoz van rendelve, vagyis ahhoz csatlakozik.
- Lehetséges megoldások: Master-slave megoldás, amikor egy CPU osztja ki a feladatokat. Másik megoldás pedig a sefl scheduling, amikor minden processzor ütemez

Processzor affinitás

A Cache az éppen futó feladat utasítás és adat tartalmának egy részét tartalmazza, különböző processzorok esetén a Cache tartalma különböző lehet, a Cache koherencia csak a közösen tartalmazott adatokra vonatkozik. Tehát ha egy taszk más processzorra kerül, akkor nőhet a végrehajtási idő, mivel nem biztos, hogy a szükséges adatok benne vannak a másik processzor Cache memóriájában. Célunk tehát, hogy a feladatokat ugyanazon a végrehajtó egységen tartsuk.

Az affinitás lehet laza, amikor ugyan az OS törekszik rá, de nincs garancia, hogy a taszk valóban ugyanazon a végrehajtó egységen marad. Kemény affinitás esetén biztosan ugyanazon az egységen fog maradni (ha kell az OS rendszerhívásokkal éri el).

SMP architektúrák esetén az affinitás a Cache találatok szempontjából érdekes (tehát, hogy benne van az adat a Cache-ben), NUMA architektúrák esetén a fizikai memória is érdekes az affinitás szempontjából, ugyanis fontos, hogy a feladat a CPU-hoz *közeli* memóriában fusson. Ez összességében egy bonyolultabb ütemező algoritmus igényel.

Terhelés megosztás

Egy globális futásra kész sor, vagy legyen egy processzoronként?

Amennyiben minden processzor futásra kész sorral rendelkezik, a sorok között mozgathatja az OS kernelje a taszkokat (push), vagy IDLE taszkot végrehajtó processzor próbálthat más sorokból taszkokat szerezni (pull). Természetesen a kettő kombinációja is használható.

Linux ütemezője

A Linux ütemezése erősen kernel verzió és konfiguráció függő, alapvetően egy és több processzoros heterogén architektúrájú rendszerekre optimalizált.

Több különböző ütemezési osztályt különíthetünk el:

- SCHED_NORMAL: minden feladat, ami nem lett egy másik osztályban felsorolva
- SCHED_RR: valós idejű RR
- SCHED_FIFO: valós idejű FIFO ütemezés

7. Feladatok leírása. Alacsony szintű vezérlési szerkezetek, coroutine és fiber. Folyamatok és szálak részletes bemutatása. Folyamatok és szálak megvalósításának HW feltételei.

Milyen architektúrákkal tudunk párhuzamos részfeladatokat leírni?

- Alacsony szintű vezérlési szerkezetek: Round Robin esetleges megszakítással, függvény sor
- Coroutine és Fiber
- Process (folyamat)
- Thread (szál)

Round Robin architektúra

A Round Robin a legegyszerűbb architektúra. Jellemzőn nincsen közös erőforrás kezeléssel kapcsolatos probléma, és nincsen megszakítás. A válaszidő erősen függ a taszkoktól. Általánosságban elmondható, hogy nagy szórással rendelkezik, és worst case értéke lineárisan nő a taszkok számának növekedésével. Maga az architektúra “törékeny”, új taszk hozzáadása felborítja az ütemezést és az időzítéseket. A CPU folyamatosan megy, így ez összességében nagy fogyasztást eredményez.

Round Robin megszakítással

Erről az architektúráról alapvetően elmondható, hogy jobban kezeli az időkritikus részeket, és a processzor altatásával a fogyasztás csökkenthető. A válaszidő ugyanúgy nagy szórással rendelkezik, worst case értéke lineárisan nő a taszkok számával. Az architektúrát ugyanúgy a törékenység jellemzi, emellett biztosítanunk kell a kölcsönös kizárást a interrupt és a fő hurok között.

Függvénysor alapú ütemezés

Az ütemezés képes prioritásokat kezelni mind részfeladat, mind interrupt szinten. A kölcsönös kizárást ugyanúgy biztosítani kell az interrupt és a fő hurok között. A válaszidőről elmondható, hogy kisebb szórással rendelkezik mint a RR architektúra és worst case ideje nem nő lineárisan a taszkok számával. Az ütemezés nem preemptív és új feladat nem borítja fel az architektúrát. A fogyasztás csökkenthető a CPU altatásával.

Kooperatív multitasking

A kooperatív multitasking megvalósulhat egy folyamaton vagy szálon belül akár OS támogatással akár programnyelvi megvalósítással. Az OS szintjén egy coroutine-eket vagy fiber-eket tartalmazó folyamat vagy szál kerül ütemezésre. Az ütemező algoritmus a programozó kezében van, neki kell implementálni (kooperativ). A Corutine egy programnyelv-szintű elem, míg a fiber rendszerszintű eszköz.

Coroutine és Subroutine

A Subroutin egy LIFO-kén képzelhető elé (first called, last returned). Alapvetően egy belépési ponttal és több kilépési ponttal rendelkezik. A visszatérési értékek és paraméterek tárolására a vermet használja. Coroutine esetén az első belépési pont azonos a Subroutine-nal, azután viszont a legutóbbi kilépési pontra tér vissza, átlépés egy külön utasítással lehetséges. A coroutine nem használhat vermet, hiszen azt megtelne (a coroutine nem tér vissza, valójában ide-oda lépked).

Coroutine és fiber értékelése

Jól alkalmazhatók multitaskinggal megoldható problémák esetében. Vermet használó rendszereknél a coroutine implementáció nehézkes (fibernél ez nem gond, az rendszerszintű). Erőforrásokon nem szükséges osztozkodni (kooperativ), ebből következően nincs szükség különböző OS hívásokra, ami összességében kisebb erőforrás igényt jelent. Az ütemezésüket az OS végzi egy szálon vagy folyamaton belül, tehát csak egy végrehajtó egységet tudnak használni.

Foylamatok

A folyamat végrehajtás alatt álló programot jelent. Ugyanabból a programból akár több folyamatot is létre tudunk hozni, melyek saját adat- és kódmemória területekkel, vagy heap-el és stack-el rendelkeznek. A folyamatok egy virtuállis CPU-n futnak, nem férhetnek hozzá a többi folyamathoz és a futás során előálló processzorállapothoz. Amennyiben egy másik folyamat kezd futni, kontextusváltás történik. Minden folyamatnak saját virtuális memóriaterülete van, nem férnek hozzá más folyamatok virtuális memóriájához vagy a fizikai memóriához. Ezen probléma megoldása egy fejlett MMU egységet tesz szükségessé. Lehetőségünk van azonban bizonyos fizikai területeket láthatóvá tenni a folyamat számára olvasási, modernebb MMU-k esetén akár írási joggal is.

Folyamatok létrehozása OS specifikus rendszerhívással történik (Windows esetében CreateProcess, Linux esetén fork()). A FreeRTOS nem támogatja a folyamatok létrehozását. A létrehozó és létrehozott folyamatok között szülő/gyermek viszony áll fenn.

A folyamatok a védelem és szeparáció szempontjából hatékonyak, azonban pont ezen tulajdonságuk teszi nehézzé és erőforrás igényessé a kommunikációt a folyamatok között, az MMU és virtuális memória pont ezt teszi lehetetlenné. A kommunikáció a kernel segítségével különböző rendszerhívásokkal történhet, ami összességében sok adminisztrációval és erőforrás-igénnyel jár.

Folyamatok befejezése is OS specifikus rendszerhívással történik (Windows: TerminateProcess(), Linux: exit()). Ebben az esetben a szülő megkapja a visszatérési értéket, ami egy egész szám. Ha a szülőt hamarabb zárjuk be, mint a gyereket, erre tipikus megoldások az alapértelmezett szülő folyamat, vagy a gyermek bezárása. Hogy melyik megoldást alkalmazzuk, az szintén OS függő.

Szálak

A szál a CPU használat alapértelmezett egysége, magában szekvenciális kódot jelent. Saját virtuális CPU és saját verem áll rendelkezésre. A kód, adat és heap tekintetében osztozik a többi szállal, melyek ugyanazon folyamat kontextusában futnak - egy folyamatban tehát több szálat is létrehozhatunk, de ez fordítva nem igaz. A szálak létrehozása (Windows: CreateThread(), FreeRTOS: xTaskCreate()) kis erőforrással és adminisztrációval jár, hasonlóan a megszüntetésükhöz. A kommunikáció gyors a közös memórián keresztül. Ugyanakkor pont ez a közös memórián keresztüli kommunikáció jelent veszélyt, a közös területeken lévő adatok konzisztenciája sérülhet. A közös memóriaterületekre és más erőforrásokra is biztosítanunk kell tehát a kölcsönös kizárást.

8. Folyamatok és szálak FreeRTOS-ban és Linux-ban. Linux folyamat és szál létrehozása, FreeRTOS task (szál) és használata.

Folyamatok Linuxban

Linux OS esetében folyamatokat a *fork()* rendszerhívással tudunk létrehozni (a létrehozás OS függő, pl. Windows esetében CreateProcess()). A létrehozó és létrehozott folyamatok között szülő-gyermek viszony áll fenn. A szülő erőforrásaihoz való hozzárés konfigurálható, (mindenhez - semmihez), a gyermeket parancssoron keresztül paraméterezhetjük. Szálakat a *pthread_create()* meghívásával hozhatunk létre, és a *pthread_join()*-al szüntethetünk meg.

Folyamatok FreeRTOS-ban

A FreeRTOS folyamatok létrehozását nem támogatja, ezért taszkokkal (szálakkal szoktunk dolgozni). Szálakat az xTastCreate() függvénnyel hozhatunk létre.

9. Feladatok együttműködése általában, erőforrás, közös erőforrás, kritikus szakasz, atomi művelet, újrahívhatóság és kölcsönös kizárás definíciója és azok kapcsolata. RAM és PRAM modell, memória mint közös erőforrás. HTM alapötlete és összehasonlítása a kölcsönös kizárással.

Feladatok együttműködése során felmerülő kérdések

- Erőforrások használata, közös erőforrások?
- Feladatok kommunikációja?
- Feladatok szinkronizálása?
- Architektúra függő kérdések: Mit és hogyan használunk feladatok megoldására? Milyen következményei lesznek?

Bernstein tétele: Mikor hajtható végre két feladat párhuzamosan

Legyen P_i és P_j két része egy programnak. P_i összes bemeneti változója legyen I_i és kimeneti változója O_i . P_j esetében hasonlóan I_j és O_j . A két program párhuzamosan végrehajtható, vagyis független, ha

$$
\begin{aligned}
I_j \cap O_i &= \emptyset \\
I_i \cap O_j &= \emptyset \\
O_i \cap O_j &= \emptyset
\end{aligned}
\tag{2}
$$

Együttműködés lehetőségei

- Közös memórián keresztül: RAM vagy PRAM modell
- Üzeneteken keresztül

RAM modell

Klasszikus Random Access Memory (egyetlen végrehajtó egység) modell szerint működik. A modellben a következőket feltételezzük: egyrészt a RAM tárolórekeszek egydimenziós tömbjéből áll. A memória rekeszenként címezhető és rekeszenként érhető el írási és olvasási műveletekkel. Az írás a teljes rekesz tartalmát felülírja az új értékkel az előző értéktől függetlenül, a írás nem változtatja a rekesz tartalmát. A memória megbízhatóságát 1-nek tekintjük (ez nem teljesen igaz, a megbízhatóságot a HW hibák és kozminkus sugárzás csökkentik).

PRAM modell

Paralleled/Pipelined Random Access Memory, sok végrehajtó egység, közös memória. A memóriát több végrehajtó egység írhatja és olvashatja párhuzamosan. Feltételezések: olvasás-olvasás ütközésekor mindekét egység ugyanazt az adatot olvassa ki. Olvasás-írás ütközése esetén a rekesz tartalma felíródik a régi adattal, a kiolvasott adat pedig vagy az előző vagy az új adat lesz, más nem lehet. Írás-írás ütközés során valamelyik írás hatása fog érvényesülni, tehát a rekesz tartalma a két beírandó adat közül valamelyik lesz, más nem lehet. A gyakorlatban főleg ezt használjuk.

Közös erőforrások

Az **erőforrás** minden olyan eszköz, amire a párhuzamos programnak futása közben szükséges lehet. A legfontosabb erőforrás maga a végrehajtó egység, ezen kívül további erőforrások például a memória, perifériák…

Közös erőforrások azon erőforrások, melyeken a párhuzamosan futó feladatok osztoznak, vagyis egyszerre több feladat is *szeretné* használni. Egy adott erőforrást többnyire egy, vagy megadott számú feladat tud helyesen használni. A közös erőforrásból eredő problémákra az OS különböző szolgáltatásokat nyújt, de a megoldás összességében a programozó kezében van. Vagyis a programozónak és rendszertervezőnek kell a közös erőforrásokat felismerni, és azok helyes használatát biztosítani.

Közös erőforrás - probléma

Például egy összetett adattípust (struktúrát két feladat használ). T1 feladat írja, T2 pedig olvassa a struktúrát. Tegyük fel, hogy a struktúra írása közben T2 futásra vált (pl preemptív a rendszer), ennek következtében inkonzisztens állapotban olvashatja ki a struktúrában lévő adatokat.

Megoldások a közös erőforrás problémára

Kölcsönös kizárás (mutual exclusion $\rightarrow$ mutex)

Kölcsönös kizárás során azt biztosítjuk, hogy egy adott erőforrást egy időben legfeljebb annyi taszk használjon, amennyinél a helyes működése még garantálható. A kölcsönös kizárást a programban kell megoldanunk, többnyire a használt erőforrás lockolásával oldjuk meg a problémát, ekkor ugyanis elzárjuk a többi részfeladat elől.

Kritikus szakasznak nevezzük a taszkok azon kódrészleteit, melyek során a kölcsönöz kizárást egy bizonyos erőforrásra biztosítjuk. A kritikus szakasz tehát az adott erőforráshoz tartozik, és atomi (megszakíthatatlan) műveletként hajtjuk végre.

Atomi műveletek

Tulajdonképpen nem megszakítható műveletet jelent, amit a processzor egyetlen utasításként hajt végre. Egyprocesszoros, DMA nélküli rendszerek bármilyen műveletek atomivá tehető az interruptok tiltásával, majd a művelet befejezésével azok engedélyezésével. A közös erőforrások lockolását és a kritikus szakasz használatát tulajdonképpen atomi műveletekre vezetjük vissza.

Közös erőforrások védelem

Mik férhetnek hozzá a közös erőforrásokhoz? Egyrészt a taszkok, DMA, és az ISR-ek (Interrupt Service Routine). Lehetőségek a közös erőforrások védelmére a megszakítások és ütemezések tiltása, vagy az erőforrások lefoglalása és feloldása. Előbbi kettő csak speciális esetekben lehetséges, a lockolást tekintjük az elfogadott megoldásnak.

Újrahívhatóság

Tulajdonképpen a közös erőforrások problémájának egy kiterjesztett, általánosított esete, egy függvényben és objektumban is felléphet, amennyiben az adott függvényt/objektumot egyszerre többen is meghívhatják. Ez úgy lehetséges, hogy ugyanazt a függvényt hívjuk egy taszkból és egy IT rutinból, vagy két különböző taszkból preemptív ütemezés esetén. A hívott függvényen/objektumon belül azt kell megvizsgálni, hogy a benne használt változók, függvények és perifériák közös erőforrásnak tekinthetők-e, amennyiben igen, akkor azokat, mint közös erőforrást, megfelelően kezel-e a függvény.

10. Közös erőforrásra várakozás (lock) lehetőségei és annak illeszkedése a feladatok egyszerűsített állapot-átmeneti ábrájához. Lockolás részletessége. Aktív és passzív várakozás, összehasonlításuk. Megadott időre történő várakozás (timeout) lehetőségei és időkezelés.

Várakozás közös erőforrásokra

Egy adott feladat más feladat által használt erőforrásra *eseményre vár* állapotban várakozik. A közös erőforrások védelmére az OS nyújt különböző szolgáltatásokat. Az erőforrást használni kívánó taszk egy rendszerhívással próbálja az erőforrást megszerezni.

Amennyiben **az erőforrás foglalt**: a feladat az adott erőforrásokra váró feladatok várakozási sorába (tipikusan egy FIFO) *eseményre vár* állapotba kerül. Amennyiben az erőforrás felszabadul, kiválasztásra kerül egy erőforrásra váró feladat (tipikusan FIFO ütemezés) mely *futásra kész* állapotba kerül, és lefoglaljuk az erőforrást számára. **A taszk csak akkor kezd futni., ha megkapja a CPU-t!**

Ha **az erőforrás szabad**, akkor az OS lefoglalja a taszk számára az erőforrást és *futásra kész* állapotba kerül.

Az erőforrást használó taszk egy OS hívással szabadítja fel az erőforrást.

Kölcsönös kizárás probléma

Mint láthattuk az erőforrások lefoglalása és felszabadítása is OS hívásokkal történik, tehát összességében a kölcsönös kizárás nagy overhead-del és sok erőforrással jár, tehát a használatát minimalizálni kell. Az sem jelenthet megoldást, ha nagyobb egységekben végezzük az erőforrások lefoglalását és felszabadítását, ekkor ugyanis nehezebben fog az ütemező futtatható feladatot találni.

Passzív várakozás lock feloldására (sleeplock)

Tulajdonképpen ütemező által karbantartott várakozási sorokat jelent. Ha ez erőforrás nincs lezárva, akkor megkapja a feladat lezárva és fut tovább. Ha lockolva van az erőforrás, a feladat megy az erőforráshoz tartozó várakozási sorba, és egy futásra kész taszk futó állapotba kerül. Ha az erőforrás felszabadul, a sor elején álló taszk megkapja, és futásra kész állapotba kerül.Ez a megoldás erőforrás takarékos, de overheaddel jár az OS futása és rendszerhívások miatt. Ha nincs feladat, a processzor aludhat, ilyenkor egy külső HW IT ébresztheti fel.

Aktív várakozás lock feloldására (spinlock)

Fontos kérdése, hogyha egy részfeladat aktívan várakozik, akkor a többi taszk hogyan tudja előidézni az erőforrás felszabadulását a rendszerben? Nem tudnak futni, ha az aktívan várakozó taszk fut. Ez nem jelent problémát külső HW IT vagy több processzor esetén. Mivel a CPU folyamatosan fut, a fogyasztás nőni fog, tulajdonképpen CPU erőforrás pazarlást okoz. Erősen függ a CPU sebességétől, ebből következően a hordozhatóság is rossz.

Akkor hogyan is várakozzunk?

(Garantáltan) rövid idejű várakozások esetén a spinlock használata elkerülhetetlen. Adott idejű várakozásra HW Timer is használható, ami kombinálható az aktív várakozással ("*kevésbé aktív várakozás*"), ebben az esetben figyelnünk az IT overheadre is. Az ütemező tipikusan nem használ spinlock jellegű hívásokat, a kernel esetében viszont előfordulhat.

11. Kölcsönös kizárás lehetőségei közös memória esetén (PRAM modell). Lockbit-től az RCU-ig, a megoldások és azok összehasonlítása. FreeRTOS és Linux példák. Randevú és kommunikáció PRAM modell esetén.

Tipikus eszközök kölcsönös kizárás megoldására RAM/PRAM modell esetén

- Lock bit
- Szemafor
- Kritikus szakasz objektum
- Mutex

Tipikus hibák kivédésére és kölcsönös kizárási probléma megoldására a Monitor eszköz használjuk.

Lock bit

Védendő erőforráshoz tartozó Bool változó. Aktív (TRUE) értéke esetén az erőforrás foglalt, egyébként pedig szabad. Belépési művelet esetén a lock bitet teszteljük, ha szabad az erőforrás, akkor lefoglaljuk (lock bit := TRUE), és lépünk tovább kritikus szakaszba. Amennyiben foglalt, akkor aktívan várakozunk a FALSE értékre, utána lockoljuk az említett módon. Kilépés esetén egyszerűn FALSE értékre állítjuk a lock bitet.

A Lock bit alkalmazása során egy súlyos hibát jelenthet, ha a bit tesztelése során jön egy interrupt, ami szintén az adott erőforrást szeretné használni. Az IT-ben vagy az utána lefutó részfeladatokban még szabadnak látszik az erőforrás, a lefoglaló taszk pedig még nem került be a kritikus szakaszba (IT miatt nem jutott odáig), pedig már abban lenne. Emiatt inkozisztencia alakulhat ki az erőforrásnál. Megoldásként a lockbit tesztelésének idejére tilthatjuk az interruptokat, vagy *test and set* atomi gépi utsítást használunk a tesztelésre.

Szemafor

A szemaforok között elkülöníthetünk bináris (egy taszk a kritikus szakaszban), és counter típusú (több feladat a kritikus szakaszban, vagy N darabos erőforrásból M-et lefoglalunk) szemaforokat. A bináris szemafor tulajdonképpen egy magas szintű lock bit, implementációtól függően aktívan várhat, vagy várakozási sorba helyezheti az adott taszkot. Szemaforokon két műveletet értzelmezünk:

- Belépés: P(), //Wait(), Pend()
- Kilpés: V(), //Signal(), Post()

Counter típusú szemafor esetén megadhatunk belépésnél és kilépésnél is egy-egy paramétert, hogy hány egység erőforrással szeretnénk belépni vagy kilépni. Ha egynél több erőforrásra van szükségünk, akkor azokat vagy mind lefoglaljuk, vagy nem foglalunk le semmit.

Kritikus szakasz és mutex

Tulajdonképpen bináris szemafor szerűen működnek. **Kritikus szakasz** esetében létre kell hozni egy CriticalSection objektumot, melyben *Enter()* metódussal lépünk be, és *Leave()* metódussal léphetünk ki. Ha szükséges a létrehozott objektum megszüntethető. Mutex használata: *Acquire()*, *WaitOne()*, elengedésnél a *Release()* metódusok.

Multiple read, single write mutex

Lényege, hogy egy időben tetszőleges szál olvashatja, de csak egy szál írhatja a memóriát. Különböztessük meg a szálakat, nevezzük Reader és Writer szálaknak. A működés során a Writer addig nem léphet be, ameddig Reader van a mutex-en, azonban ha a Writer vár, akkor további Reader-ek is beléphetnek.

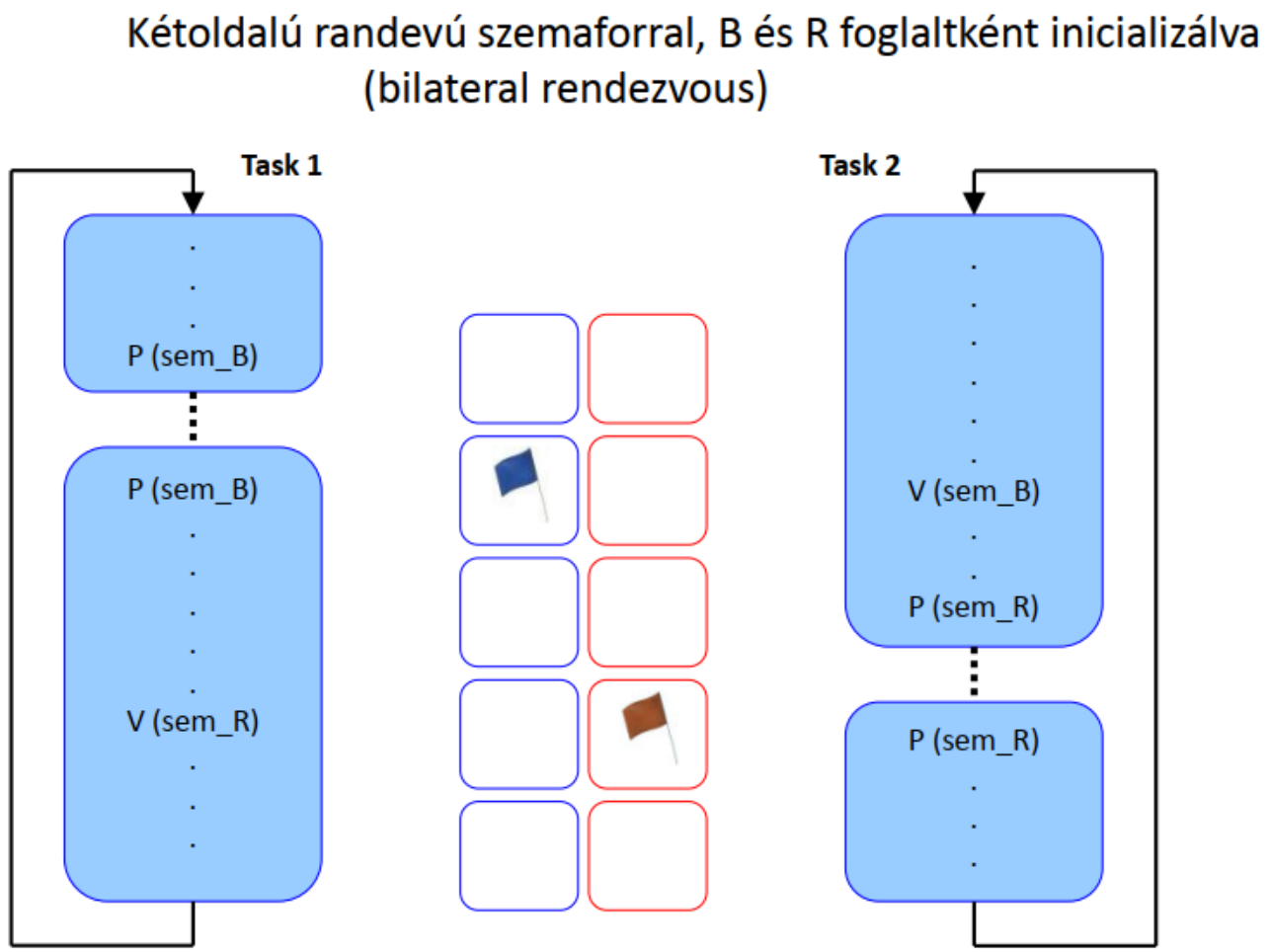
Read-copy-update

Writer egy saját másolatot ír, a Reader szálak pedig a memóriában lévő adatokat olvassák. A később belépő Reader szálak megvárják, amíg a Writer által írt módosítások bekerülnek a memóriába.

Részfeladatok közötti szinkronizáció

A részfeladatok közötti szinkronizációt visszavezethetjük kölcsönös kizárásra. Fontos azonban megjegyezni, hogy nincsen védendő erőforrás, a célunk a részfeladatok hatékony együttműködése.

Randevú bináris szemaforral:



Ebben az esetben az operációs rendszer által nyújtott szolgáltatásokat használunk, és a feladatok passzívan várnak egymásra. A szemaforok alapesetben foglaltként vannak inicializálva.

Kommunikáció védett memóriaterületen keresztül

Alapvetően szálak közötti kommunikációra használható a közös memória, folyamatok nem tudnak ilyen módon kommunikálni. A közös memóriában különböző adattípusokon keresztül (tömb, struktúra, objektum…) megvalósítható egyirányú és kétirányú kommunikáció is: előző esetben a külső ír, a vevő olvas, kétirányú kommunikáció során pedig mindkét fél írhat és olvashat. A kölcsönös kizárást lock bittel, szemaforral, mutexszel, vagy kritikus szakasz objektummal oldhatjuk meg.

Blokkoló és nem blokkoló függvényhívás

Blokkoló függvényhívás (szinkron)

Tulajdonképpen ez jellemző a klasszikus függvényhívásra, az eredmény és a mellékhatások a függvény visszatérése után jelentkeznek, a visszatérési érték kezelése pedig egyszerűen megoldható.

Nem blokkoló függvényhívás (aszinkron)

Csak a végrehajtás kezdődik el a hívásra, az eredmények és mellékhatások visszatéréskor még nem jelentkeznek. A visszatérési érték, eredmények és mellékhatások kezelése csak más értesítés után (pl. esemény, signal) lehetséges.

12. Párhuzamos programozás során elkövetett tipikus hibák és azok elleni védekezés. Holtpont, szükséges feltételek, holtpont kezelése. Priorítás inverzió és elkerülése. Monitor és megvalósítása.

Tipikus hibák párhuzamos programozásban

Versenyhelyzet

Párhuzamos program lefutása során a közös erőforrások helytelen használata miatt a közös erőforrás, vagy az azt használó taszk hibás állapotba kerül. Ilyen volt például a korábbi összetett adattípusos példa amikor a *"félig módosított struktúrát"* olvastuk egy másik taszkban. A hibás állapot nem feltétlenül következik be, a részletei erősen függenek a részfeladatok lefutási sorrendjétől.

Kiéheztetés

Ha a párhuzamos rendszer hibás működése miatt egy adott taszk soha nem jut hozzá a futáshoz szükséges erőforrásokhoz, akkor beszélünk kiéheztetésről. Ez felmerülhet akár a CPU-ra, akár más közös erőforrásokra.

Holtpont

A közös erőforrások hibás beállítása vagy használata miatt a részfeladatok a rendszerben egymásra várnak. Ebből következően nem lesz futásra kész folyamat, nem jöhet létre belső esemény, és a rendszer nem fog tudni előre lépni. Tipikus példa: Két kecske a hídon.

Livelock

Holtpont helytelen feloldása során alakulhat ki. A rendszer folyamatosan dolgozik de sose fog előre lépni. Tipikus példa: Két udvarias ember találkozik az ajtóban.

Holtpont és kialakulásának feltételei

A holtpont a legsúlyosabb programozói hiba, ami multiprogramozás során előfordulhat. Definíciószerűen **A rendszer feladatainak valamely H részhalmaza holtponton van, ha a H halmazba tartozó valamennyi feladat olyan eseményre vár, amit csak egy másik H halmaz beli feladat tudna előállítani** . A holtpontot nehéz felismerni és detektálni, jellemzően versenhelyzet szerűen szokott jelentkezni, van, hogy jól működik a rendszer, van, hogy előáll a holtpont.

Holtpont kialakulásának szükséges feltételei

Kölcsönös kizárás

Foglalva várakozás: Legyen olyan feladat, ami foglalva tart erőforrásokat, mialatt további szükséges erőforrásokra vár. Emiatt a holtpont a a *H* halmazon kívüli feladatok működését is befolyásolhatja a lefoglalt erőforrások miatt.

Ne legyen erőszakos erőforrás elvétel, a feladatok csak maguk szabadíthatják fel az erőforrásokat.

Legyen körkörös várakozás, vagyis a rendszer feladatainak legyen egy $\{P_0, P_1, P2...P_n\}$ sorozata, melyekben P_i a P_{i+1} -ik feladat által foglalt erőforrásra, P_n pedig a P_0 által foglalt erőforrásra vár.

Holtpont elkerülése

Legegyszerűbb megoldás, ha olyan rendszert tervezünk, melyben a holtpont szükséges feltételei közül valamelyik nem teljesül. Egyik lehetőség, hogy nincsen futási idejű erőforrás foglalás, ez lehetséges beágyazott rendszerekben. Másik lehetőség, ha a foglalva várakozást szüntetjük meg, úgy hogy minden szükséges erőforrást egyben foglalunk le, egyetlen rendszerhívással (ez összességében rontja az erőforrás kihasználást). Az is egy megoldás, hogy lehetővé tesszük az erőforrások erőszakos elvételét, vagy a körkörös várakozást kerüljük el.

Prioritás inverzió

Prioritásos ütemezőben fordulhat elő, erőforrások használatával összefüggő hiba. Legegyszerűbb esetben 3 különböző, statikus prioritású taszknál fordulhat elő, melyek közül a legmagasabb és legalacsonyabb prioritású is ugyanazon erőforrásra vár.

Példa: Legyen 3 taszk *Taszk1, Taszk2 és Taszk3* különböző prioritású taszkok (1 jelenti a legalacsonyabb, 3 pedig a legmagasabb prioritást)., és legyenek *A* és *B* közös erőforrások.

- *Taszk3 B* erőforrásra vár.
- *Taszk1* megszerzi a szabad *A* erőforrást és fut tovább.
- *Taszk3* által várt *B* erőforrás felszabadul, és elkezd futni.
- *Taszk3* a *Taszk1* által lefoglalt *A* erőforrásra vár. Mivel ez foglalt várakozik. *Taszk1* ismét elkezd futni.
- *Taszk2* közepes prioritású taszk futásra kész állapotba kerül, és mivel magasabb prioritású elkezd futni, és hosszú ideig CPU intenzív feladatokat hajt végre

Tehát nézzük a kialakult helyzetet: *Taszk3* nem tud futni, mert a *Taszk1* által foglalt *A* erőforrásra vár. *Taszk1* sem tud futni, mert a *Taszk2* által lefoglalt CPU-ra várakozik. Összességében alacsonyabb prioritású taszkok nem engedik futni a magas prioritású taszkot.

Megoldások prioritás inverzióra

Prioritás öröklés: alacsony prioritású feladat megörököli az általa kölcsönös kizárással feltartott feladat prioritását a kritikus szakaszból való kilépésig. Ez csak részben jelent megoldást a problémára

Prioritás plafon: Az örökléshez hasonló, azzal a különbséggel, hogy az alacsony prioritású feladat az adott erőforrást használó legnagyobb prioritású feladat prioritását örökli.

Más megközelítés: Ne speciális protokollal próbáljuk megoldani, hanem tervezzük meg jól a rendszer és a prioritásokat.

Locklolás során elkövetett tipikus hibák

- Belépés és/vagy kilépés elmaradása
- Többszöri belépés és/vagy kilpés
- Más erőforrás lefoglalás
- Erőforrás lefoglalása indokolatlanul hosszú időre: Mindig a minimális időre kell törekedni egy erőforrás lefoglalásánál
- Előfordulhat még prioritás inverzió, Livelock és Deadlock

Monitorozás: hibák elkerülése

A lockolással kapcsolatos feladatokat lokalizáljuk egy erőforrást körülvevő API-val. Ezzel a lockolás nem szétszórtan történik a programban, hanem egyetlen - a közös erőforráshoz szorosan kapcsolódó - programrészben.

Hoare szemnatika: Fusson azonnal az erőforrást megszerző feladat. Ez a szemantika erőforrásigényes, nehézkes a megvalósítása, és nehezen összeegyeztethető a preemptív ütemezéssel.

Mesa szemantika: tegyük a feladatot a futásra kész feladatot közé, és majd az ütemező lefuttatja.

13. FreeRTOS és Linux megoldások kölcsönös kizárásra, szinkronizációra közös memória esetén. A megoldások felsorolása, főbb jellemzőik, összehasonlításuk.

FreeRTOS locking szolgáltatások

FreeRTOS csak szálak (taszkok) létrehozását támogatja, ezért a közös memória alapvető. Milyen lehetőségeink vannak a közös memória védelmére?

Megszakítások engedélyezése és tiltása (különböző makrókkal)

taskDISABLE_INTERRUPTS() és *taskENABLE_INTERRUPTS()* makrók. Viselkedésük konfigurációfüggő.

Kritikus szakasz

Egyben a megszakítások tiltását is jelenti. A kritikus szakaszoknak rövidnek kell lenni, a real-time működés függhet tőlük.
Belépés: *taskENTER_CRITICAL*, vagy ISR estében *taskENTER_CIRITICAL_FROM_ISR*.
Kilépés: *taskEXIT_CIRITICAL*, vagy ISR esetében *taskEXIT_CIRITICAL_FROM_ISR*

Task notify mechanizmus

Minden taszkhoz hozzárendelünk egy 32 bites notification értéket, amit lehet tiltani/engedélyezni. Inicializáláskor zérus értéket kap, ezt tudjuk módosítani: nullázás, inkrementálás, beállítás adott értékre, adott bit beállítása, feltételes beállítás… A notification pending állapotban is lehet. Notify sok funcikcót rejt magában, többek között lehetőségünk van taszkok közötti kommunikációra gyorsabban és egyszerűbben mind pl. szemafor esetén.

Tasky Notify szinkronizáció: Bináris és counter típusú szemafor működés valósítható meg vele.

Szemafor létrehozása és kezelése

xSemaphoreCreateBinary(): Heap területen hozza létre a szükséges adatstruktúrát, csak egy handlet kapunk vissza hozzá
xSemaphoreCreateBinaryStatic() esetén a felhasználónak kell memóriát foglalni. Ugyanúgy egy handlet kapunk vissza.
xSemaphoreCreateCounting(): maximális/kezdő értékkel paraméterezhető. Heap területen dolgozik, 0-s visszatérési értéke hibát jelez. Alkalmazható számlálásra, több példányos erőforrás kezelésére

Szemaforok kezelése: xSemaphoreTake() és xSemaphoreGive() függvényekkel tudjuk lefoglalni illetve felszabadítani.

Mutex létrehozása

xSemaphoreCreateMutex(): Heap területen dolgozik, hiba esetén null értékkel tér vissza
xSemaphoreCreateMutexStatic(): Ez a statikus memóriaterületen dolgozik, a megfelelő erőforrás lefoglalás a programozó feladata
xSeamphoreCreateRecursiveMutex(): konfigurációs beállítás szükséges hozzá, heap területen dolgozik. Egy adott taszkból meghívhatók többször is a megfelelő rekurzív megoldásokkal.

Notify mutex és szemfor timeout

Közös erőforrásra vonatkozó kölcsönös kizárás esetén csak hibakezelésre használjuk. A foglalt erőforrás inkonzisztens állapotban lehet a timeout után.

Linux és a közös memória

Linux esetében a közös memória több dolgot is jelenthet: Egyrészt az egy folyamat kontextusában futó szálak által használt közös memóriát, másrészt pedig a folyamatok közötti megosztott memóriát (emlékezzünk vissza, fejlettebb MMU lehetőséget ad rá).

Megoldások kölcsönös kizárásra, POSIX kompatibilis hívások

POSIX szemafor (#include <semphore.h>)

Counter típusú, de nem támogatja a többszörös erőforrás foglalást. Két típusát különíthetjük el

Unnamed semaphores: Ezen belül beszélhetünk thread-shared és process-shared szemaforokról. Előbbi folyamatspecifikus címtérben, utóbbi pedig a folyamatok közötti osztott memóriában kerül tárolásra.

Named smephaores: a kernelben kerül tárolásra

Szemafor létrehozása : int sem_init(sem_t * sem, int pshared, uint value). Létező szemafor újrainicializálása nem definiált művelethez vezet, kerüljük.

Várakozás szemaforra:
int sem_wait(sem_t * sem): ha a szemafor értéke zérus, akkor addig várakozik amíg valami meg nem növeli, vagy nem érkezik jelzés
int sem_trywait(sem_t * sem): ha a szemafor értéke zérus, azonnal visszatér
int sem_timedwait(sem_t * sem): ha a szemafor értéke zérus, akkor addig várakozik, amíg valami meg nem növeli, vagy nem érkezik jelzés, vagy a megadott várakozási idő le nem jár

Szemafor értékének lekérdezése
int sem_getvalue(sem_t * sem): lekérdezi a paraméterként megadott szemafor értékét. A szemafor értéke már a kihívás végére megváltozhat, inkább csak tájékoztatásra szolgál.

Szemafor elengedése és törlése
int sem_post (sem_t * sem): hiba esetén -1-gyel tér vissza
int sem_destroy(sem_t * sem):
A felszabadított szemafor memóriáját manuálisan kell kezelni/felszabadítani, amennyiben szükséges

Mutexek Linuxban

Lényegében bináris szemaforként viselkedik, tetszőleges erőforráshoz hozzárendelhetjük, amit lehet írni és olvasni is. Használata egyszerű, létre kell hozni, lock/unlock, aztán meg kell szüntetni.

Mutex létrehozása és megszüntetése
int pthread_mutex_init(…) mutex létrehozása
int pthread_mutex_destrox(…) mutex megszütnetése

Mutex lock és unlock
int pthread_mutex_lock(…): adott mutexet lezárja. A *try* változat hibával tér vissza, ha az erőforrás lockolva van
int pthread_mutex_unlock(…) adott mutexet unlockolja, viselkedése függ a mutex típusától

RW lock

Lényegében egy írásra vonatkozó bináris szemafor. Tetszőleges számú szál olvashat, ha nincs Writer szál, és szabadon léphetnek be olvasó szálak. Egyetlen Writer szál írhat, de írás alatt nem léphet be se senki sem írásra sem olvasásra. Többnyire akkor merül fel, ha nagy komplex adatstruktúrát sok szál akar olvasni, de kevés és ritkán akar írni. Alapvetően nagyobb overhead-del jár mint egy mutex, viszont több végrehajtó egység esetén különösen hasznos, a az olvasók tényleg párhuzamosan futhatnak.

RW lock létrehozás és megszütnetés

int pthread_rwlock_init(…), és int pthread_rwlock_destroy hívásokkal.
int pthread_rwlock_rdlock/wrlock: lock olvasásra/írásra

Spin lock

Aktív várakozás erőforrás felszabadulására. Akkor ajánlott, ha tudjuk, hogy a várakozás garantáltan rövid idejű, és az OS-en keresztüli rendszerhívás nagyobb overhad-del jár. Az aktív várakozás nagy energiafogyasztással jár, és egyprocesszoros rendszerben akár holtpont is lehet belőle.

Conditional variable

Események hatására a vezérlés átadását jelenti, felfogható egy egyoldalú szinkronizációnak. Kombinálni kell egy mutexszel, ami az adott contional variablet védi.

14. Folyamatok közötti kommunikáció (IPC) és a közös memória alapú kommunikáció összehasonlítása. Üzenet fogalom és címzés, nyugtázás. Direkt és indirekt, szinkron és asszinkron kommunikáció. IPC megoldások általánosan és FreeRTOS és Linux példák rájuk (részletesen nem kell őket ismertetni).

Üzenetek vs. közös memória

Üzenetek	Közös memória
Nincs közös memória. Üzenetekkel való kommunikáció lehetséges elosztott rendszerekben, vagy folyamatok között az operációs rendszerben (Inter Process Communication, IPC)	Egy folyamaton belül futó szálak esetén használhatunk közös memóriát

Üzenetek nem azonosak a számítógép hálózatokban küldött üzenetekkel, azoknál egy általánosabb fogalom. Üzenetek továbbítása történet akár rendszerhívással és függvényhívással, vagy TCP/IP kapcsolaton keresztül gépen belül vagy akár gépek között

Üzenetek továbbításának tulajdonságai

A közös memóriához képest nagyobb késleltetés és kisebb sávszélesség jellemzi az üzenetek továbbítását. A csatorna nem megbízható definíciószerűen (emlékezzünk vissza, hogy a memória megbízhatóságát 1-nek tekintettük). Üzenetek esetében nem tudunk 1-es (100%-os) megbízhatóságot garantálni (két hadsereg probléma), nem lehetünk biztosak a kommunikációs sikerében (üzenetek elvesztése, duplikálódása, sorrendek megváltozása).

Üzenetek címzése (számítógép hálózatok...)

Címzett	Címzés típusa
Egy **adott** folyamat	Unicast címzés
Minden folyamat	broadcast címzés
Folyamatok egy adott csoportja	multicast címzés
Egy folyamat	anycast címzés

Direkt kommunikáció

Szimmetrikus kommunikáció	Aszimmetrikus kommunikáció
send (P, message) recieve (Q, message), ahol *P* és *Q* a folyamat azonosítói, *message* pedig egy struktúra, ami az üzentet tartalmazza	send (P , message) recieve (id, message) ahol *P* a folyamat azonosítója , *id* a küldő azonosítója. A vevő bárkitől fogadható üzenetet

Indirekt kommunikáció

A kommunikáció egy köztes szereplőn keresztül történik. A köztes szereplő lehet postaláda (MailBox), Üzenet sor (Message queue), port… Ebben az esetben a Send() és recieve() függvények esetén a postaláda azonosítóját adjuk meg első paraméternek.

Üzenet alapú kommunikáció további tulajdonságai

A kommunikációhoz minimum egy vevő és egy küldő folyamat szükséges. A vevő folyamat a vétel után törlődhet (egyszer olvasható), vagy megmaradhat (ekkor beszélünk RAM-szerű működésről). Az üzenteknek lehet tulajdonosa, ami lehet egyrészt maga az OS, vagy egy folyamat, aminek a memóriaterületén van. Előbbi esetben az üzenet az őt használó folyamatoktól függetlenül létezhet, utóbbi esetben pedig az üzenet létezése folyamathoz kötött.

Szinkron és aszinkron kommunikáció

	Szinkron kommunikáció (blokkoló)	**Aszinkron kommunikáció (nem blokkoló)**
Külső oldal	* Send() nem tér vissza, amíg nem vették át az üzenteket, vagy nem kerültek a postaládába * Kommunikációs hiba esetén: Callback függvény hívódik meg, vagy időtúllépés + Send() hibával tér vissza	* Üzenet lokális elküldése után a Send() azonnal visszatér * Vagy nincs hibakezelés, vagy a hibáról egy függvény által értesül a küldő
Fogadó oldal	* Recieve() nem tér vissza, amíg meg nem érkezik az üzenet pl. socketet esetén a listen() függvény	* Recieve() azonnal visszatér, ha nincs üzenet, akkor azt jelezni fogja

Implementációk általánbosságban

Implementáció	**Tulajdonság**
Mailbox (Postaláda)	Indirekt kommunikációt tesz lehetővé, és véges számú üzenet tárolására alkalmas. OS szintű támogatás. (Beágyazott OS-ek, pl FreeRTOS esetében)
Message queue (Üzenetsor)	Indirekt kommunikáció, a tárolt üzenetek számát az erőforrások korlátozzák (Beágyazott OS-ek, pl FreeRTOS esetében)
TCP/IP TCP vagy UDP port	Direkt kommunikáció, socket interfészt igényel. A Távoli eljáráshívás alapja. Erőforrás igénye miatt beágyazott rendszerekre nem jellemző
Folyamok és csővezetékek	Többnyire direkt kommunikáció, pl. Linux esetében a pipe-ok
System V shared Memory	Direkt kommunikáció, fejlett MMU egység alkalmazását teszi szükségessé. Fontos a kölcsönös kizárás biztosítása a megfelelő memória területekre.
Jelzések	Direkt kommunikáció. Jellemzően tényleges adat átvitel nem történik, inkább szinkronizációra szolgál.

15. Linux IPC megoldások a jelzésektől a POSIX IPC főbb eleméig (socket-et kivéve).

Jelzések

A jelzések direkt kommunikációt tesznek lehetővé. Sok esetben tényleges adat nem kerül átvitelre, sokkalinkább szinkronizációs célja van a jelzéseknek. Példa: POSIX jelzések Linuxban (következő bekezdésben). A signal handler alapértelmezett módon a folyamat stackjében fut a signal handler (kb. mint egy IT handler), de lehet saját stackje is (ún. signalstack), és befolyásolhatja a rendszerhívások működését is.

Jelzések Linuxban

Jelzések generálása: #include <signal.h>

Függvény	Tulajdonság
int kill (pid_t pid, int signal)	Adott *pid* azonosítójú folyamatnak küld egy signalt
int killpg (int pgrp, int signal)	Adott process group minden tagjának küld egy signalt
int raise (int signal)	A hívó folyamat önmagának küld jelzést
int pthread_kill (pthread_t pthread, int signal)	Folyamat szintű hatás, csak adott szál kontextusában fut a jelzés kezelése
int tgkill (int pid, int tid, int signal)	Adott azonosítójú folyamatban adott azonosítójú szálnak küld jelzést

Jelzésekre várakozás a fő programban

Függvény	Leírás
int pause()	Csak akkor tér vissza, ha meg hívva a signal handler
int sigsuspend()	Ideiglenesen lecseréli a hívó folyamat signal maszkját, a folyamaton átmenetileg felfüggeszti
int sigwaitinfo()	Szinkron várakozás szálban adott jelzésekre, visszatér a folyamat azonsoítójával

Jelzésekre számos különböző reakciót adhatunk: Egyrészt ignorálhatjuk őket, vagy akár megállíthatjuk és felfüggeszthetjük a taszk futását jelzések hatására

TCP/IP és UDP port, Csővezetékek és megosztott memória

Direkt kommunikációt jelent. Socket interfészt tesz szükségessé. Beágyazott rendszerekre nem jellemző a nagy erőforrás igénye miatt, inkább gépen belül a localhoston, vagy gépek között szoktuk alkalmazni. Alacsony szintű, ezen alapul például a távoli eljáráshívás vagy távoli metódushívás.

Mind a csővezetékek mind a megosztott memória alapvetően direkt kommunikációt tesznek lehetővé, emellett a megosztott memória fejlett MMU egység használatát feltételezi.

16. FreeRTOS IPC jellegű megoldásai.

Kicsit ellentmondásosnak tűnhet, hogy folyamatok közötti kommunikációról beszélünk egy olyan OS esetén, ami nem támogatja folyamatok alkalmazását. Az IPC alatt FreeRTOS esetében taszkok közötti kommunikációt értünk.

FreeRTOS IPC jellegű megoldások

Megoldás	Leírás
Mailbox (Postaláda)	Indirekt kommunikációt tesz lehetővé, és véges számú üzenet tárolására alkalmas. OS szintű támogatás van Postaládákra.
Message Queue	Többnyire indirekt kommunikáció, az üzenetek számának az erőforrások szabnak felső határt

FreeRTOS sor viselkedése

A FreeRTOS sor (queue) adott számú elemet tartalmazhat. A sorba jellemzően érték szerint másolunk, alternatív megoldásként másolhatunk referencia vagy cím szerint. Utóbbi esetben az átadott adathoz már nem fér hozzá a küldő. Sorokból sorkészletek állíthatók össze. A sorba való írás és olvasás során is szem előtt kell tartaunk a várakozást.

Timeout, ha olvasás közben üres a sor, vagy írás közben televan. Ha adat érkezik a sorba vagy adat kerül ki, akkor mindig a legmagasabb prioritású taszk fut.

Sorok kezelése FreeRTOSban

A kezelés egyszerűen megoldható különböző API-kkal.

API	Leírás
QueueHandle_t xQueueCreate(length, itemSize)	Sor létrehozása
xQueueSendToFront()	LIFO sorba küldés
xQueueSendToBack()	FIFO sorba küldés
xQueueRecieve()	Fogadás sorból

Küldésnél és fogadásnál megadhatunk egy timeout értéket, ekkor várakozni fog ha a sor tele van küldésnél, vagy üres fogadásnál. Ha nincs timeout megadva az API-k hibával térnek vissza. Debuggolási célból lehetőségünk van a sor hosszát is lekérdezni. ISR-ből az API-k ISR_postfixszel ellátott változatait hívhatjuk meg.

Mailbox FreeRTOS-ban

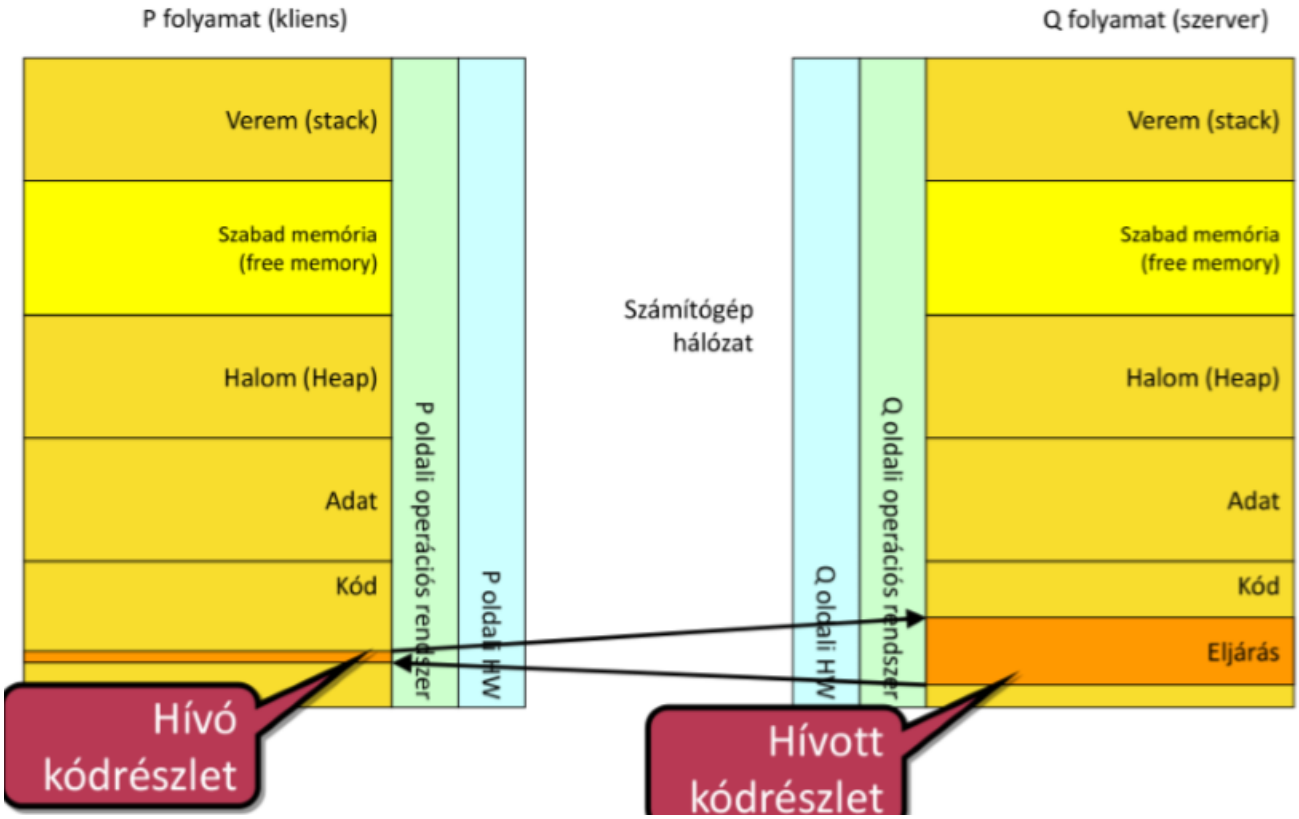
Lényegében egy egy elemű sort jelent, a legutoljára bevitt adatot lehet tetszőleges számú alkalommal kiolvasni.

API	Leírás
xQueueOverWrite	Írás, scak egy elemű Mailbox esetén használható. Nincs timeout, mindig sikeresen lefut.
xQueuePeek	Olvasás, csak egy elemű Mailbox esetén használható. Van timout ami az üres - még meg nem írt - Mailboxra vonatkozik.

17. Távoli eljárás hivás és annak implementációja Linux alatt.

Távoli eljáráshívás (RPC)

Távoli eljáráshívás során egy másik folyamat kódterületén lévő függvényt hívjuk meg a hívó folyamatból üzenetek felhasználásával. A hívást követően a hívó fél blokkolva vár a függvény lefutására, amely az őt tartalmazó folyamat egyik vezérlési szálában fut le.



RPC a programozó szemével

A hívás lényegében azonos egy lokális függvény meghívásával (ami lényegében egy programkönyvtárban található függvény, nem kell tudnia, milyen platformon fut le). A ténylegesen végrehajtott függvény megírása sem különbözik a lokális függvény megírásától. Kapunk egy függvény deklarációt és azt kell megírunk, nem kell törődnünk azzal, hogy milyen platformról érkezik a kérés.

RPC működése

Típusos hívás és visszatérési érték: strukturált adatküldés jellemzi, a platformfüggetlenség megvalósítását a fordító és az OS végzi (nem a stub?????)

Kliens oldali program egy automatikusan generált, ún. **stub** függvényt hív meg, ami elrejti a kommunikáció részleteit a hívó elől. Feladata a megadott paraméterek összecsomagolása szabványos platformfüggetlen formátumba és ezt küldi el a végrehajtó folyamatnak (ehhez használja fel az OS és a hálózat szolgáltatásait). Visszatérés esetén ugyanez a függvény konvertálja vissza a szervertől megkapott visszatérési értéket a lokális formára. A kliens szál eseményre várakozik a hívás során.

A szerver oldalon az RPC-t használó szolgáltatás kapja meg a hívást. Ezt első lépésben lokális formára hozza, végrehajtja a függvény lefutását, annak visszatérési értékét pedig platform független formára hozva küldi vissza a kliensnek. A szerver oldali szál eseményre várakozik állapotban van amikor nincsen hívás

RPC a gyakorlatban

Hogyan találjuk meg a hívott felet? Manuális konfigurációval, vagy az ún. Bróker segítségével.

RPC szabványok: DCE, SUN RPC, tulajdonképpen egy URL is felfogható RPC hívásnak

Szabványos adatformátumok: BER (Binry Encoding Rules), XML

Távoli metódushívás

Távoli metódushívás során egy távoli objektum adott metódusát hívjuk meg… ok

18. Elosztott rendszer architektúrák. Publish-subscribe, virtuális adatbusz, proxy és bróker részletes ismertetése.

Publish-subscribe

Ebben az architektúrában termelők (publisher) és fogyasztók (subscriber) vesznek részt: a fogyasztó előfizet a termelő által előállított információra. A termelő nyilvántartja az előfizetőket (referencia a fogyasztók adatfogadó interfészére), ha van információ elküldi az előfizetőknek, például RPC formájában. Az architektúra lehetővé teszi többprocesszoros rendszerek alkalmazását is: párhuzamosan futó blokkok, köztük laza csatolás. Tipikus megoldás pl. Matlab Simulink.

Virtuális adatbusz

Számos kommunikációs hálózat adatbuszként is felfogható, mint broadcast a fizikai közegben. Mindegyik NODE-hoz eljut az üzenet, az adott NODE pedig a cím alapján vagy veszi az üzenetet, vagy nem, a válasz az üzenet tartalmától függően történik.

Virtuális adatbusz pont-pont kommunikációs hálózat buszszerű működéssel. Adatbusz könnyen létrehozható a Publish-subscribe architektúrát alapul véve: Minden NODE termelő, amikor a buszra ír, minden NODE fogasztó, amikor más NODE-októl keresztül adatokat kap, magát a buszt pedig egy központi NODE jelenti az elosztott rendszerben.

Proxy

A proxy “beáll” a kommunikációban szereplő két fél közé. Feladata a kompatibilitás megteremtése, bizonyos funkciók elrejtése és teljesítményillesztés. Nézzük részletesebben a proxy feladatait!

Kompatibilitás megteremtése

Tekintsük azt az esetet, amikor a kommunikációban 2 vagy több fél eltérő kommunikációs felületet használ, magas szinten azonban mindegyik fél számára értelmezhető az átjuttatandó információ. Első lépésben tehát meg kell teremtenünk a felek között a kapcsolatot, és el kell végeznünk a különböző formátumú üzentek között a megfelelő transzformációt.

Funkciók elrejtése

Tekintsünk két egymással kommunikáló *node1* és *node2* NODE-okat, és tfh. *node1* bizonyos funkciót nem szeretnénk elérhetővé tenni a *node2* számára, anélkül, hogy a *node1*-en bármit is változtatnánk. A *node1* direkt elérését proxy alkalmazásával megakadályozhatjuk, a tűzfal pedig blokkolja a kéréseket azon funkciók felé, melyeket nem szeretnénk láthatóvá tenni a másik fél számára.

Teljesítményillesztés

Tegyük fel, hogy a kommunikációban használt *node1* kis teljesítménye miatt nem képes a beérkező kéréseket kiszolgálni. Amennyiben a hardver lecserélésére nélkül szeretnénk megoldali ezt a problémát, akkor használhatjuk a proxyt teljesítményillesztésre.Ekkor a proxy lekérdezi a *node1* által küldendő információt, és Cache jelleggel elérhető teszi a többi NODE számára.

Proxy alkalmazás - általános eset

Például amikor egy vezeték nélküli szenzorhálózatot szeretnénk illeszteni az internetre, akkor a Proxy mindhárom szerepet betölti

Egyrészt a nem TCP/IP alapú szenzorhálózat miatt az üzeneteket kell átkonvertálni.

Másrészt a szenzorhálózatot meg kell védenünk az internet felől jövő esetleges támamások elől (pl. konfiguráció).

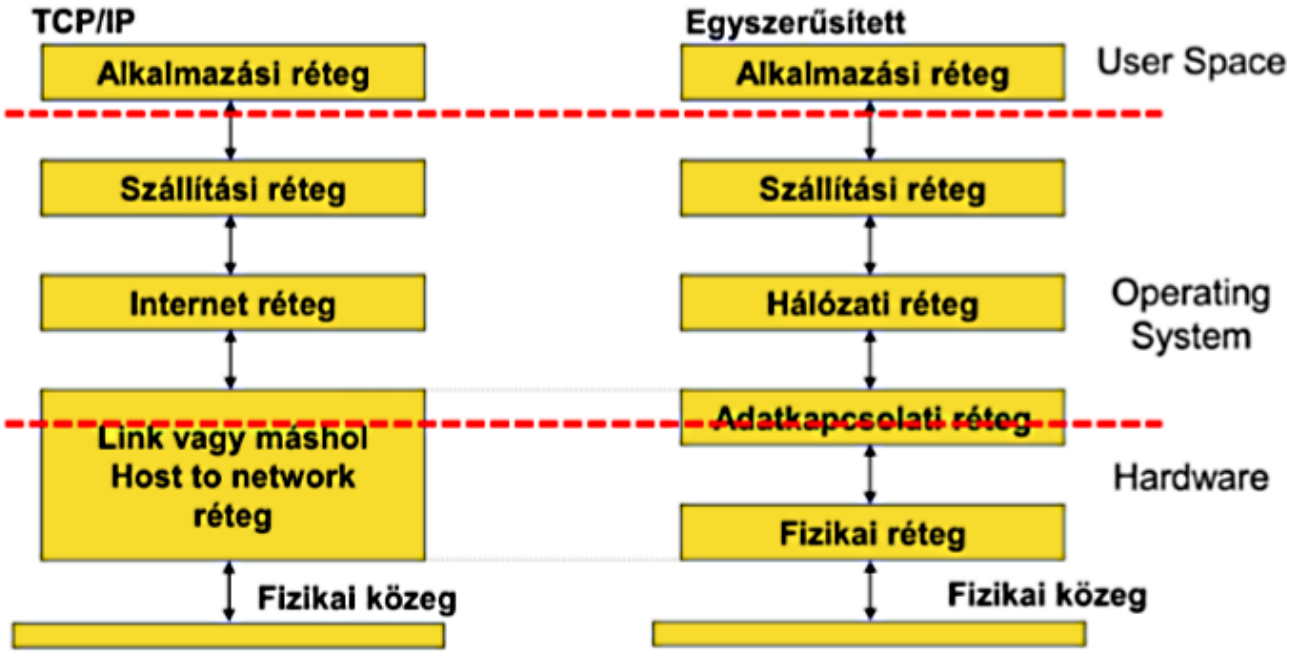
Harmadrészt a vezeték nélküli szenzorhálózatban a kommunikáció lassú. Ezért ha pl. egy adott NODE adatait akarjuk az internetről lekérdezni, akkor túl fok terhelődni a szenzorhálózat. Ennek elkerülése végett az ideiglenes értékek proxy Cacheben tárolásával az összes kérés kiszolgálható.

Bróker

A bórker egy központi szereplőt valósít meg, mely összepárosítja a kommunikálni kívánó feleket. A bróker legfontosabb tulajdonsága, hogy ismert a felek számára, elérhetősége adott a rendszer konfigurációban. A felek megadják a brókernek igényeiket vagy az általuk nyújtott szolgáltatásokat, amiket a bróker összeválogat, és megteremti a kapcsolatot a felek között.

19. TCP/IP protokoll készlet architektúrája és implementációja, tesztprogramjai Linux alatt. Tűzfal alapok. Protokoll analizátorok és működésük alapjai, Wireshark és tcpdump alapok. TCP és UDP alapok, összehasonlításuk.

TCP/IP referencia modell



Problémák: Egyes alkalmazási réteg protokollok a kernelben vannak implementálva (pl. NFS), egyes funkciók HW-ben, pl. ellenőrző összegek számítása.

Milyen támogatást nyújt a Linux?

- Alacsony szintű HW driver
  - Wi-Fi, Ethernet, 3G, 4G…
  - Ha a HW-t nem támogatja a kernel drivert kell írni…
  - SW szinten teljesen el van rejtve
- Magas szintű (ipv4, ipv6 szoftver)

Mi kell a sikeres kommunikációhoz?

- Működő alacsony szintű interfész
  - Az **ifconfig parancs** a Linux Ethernet interfészek (hálózati vezérlők) hálózati paramétereinek beállítására, és az aktuális beállítások és állapotok kiíratására szolgál. Paraméterek nélkül használva, csak az éppen aktív hálózati interfészeket sorolja fel, a legfontosabb paraméterekkel. (HW beállítások, Ethernet és IP címek beállítások statisztikák, kommunikációra használt eszköz neve…)
  - **ping parancs:** tesztelés (ping *IP cím*)
  - Működő DNS szolgáltatás (ping *domain név*, resolv.conf)

Tűzfal alapok

A tűzfal feladata a különböző szabályok kezelése, mely különböző szűrő táblákon keresztül történik (filter(default), nat, mangle, raw). Az egyes táblákban a szabályok különböző láncokra vannak fűzve. Egy adott láncon a szabályok sorrendben kerülnek feldolgozásra, az első illeszkedő szabályt hajtja végre a rendszer. A szabályok lehetnek interfész specifikusak.

Tűzfal beállítása

Lehetőségünk van direkt módon manuális beállított szabályokat beállítani az *iptables* hívásával. Ezek alapesetben a rendszer újraindításakor elfelejtődnek, disztribúció függő módon lehet őket véglegessé tenni.A konfigurációt elvégezhetjük még valamilyen GUI vagy karakteres konfiguráló alkalmazással (pl. Firewall Builder, Firestarter. Gufw)

Protokoll analizárotok

A protokoll analizátorok a referenciamodell különböző rétegeiben képesek vizsgálatot végezni. Általános célű eszközök, melyek megjelenítik, értelmezik és feldolgozzák az információkat, jól alkalmazhatók például rendszertelepítés során és hálózati hibák keresésére.

A **SW-es protokoll analizátorok** az adatkapcsolati rétegtől kezdenek működni. Etherneten kívül alkalmazhatók még pl. USB, WiFi és 3G kapcsolatokra bizonyos megkötésekkel. A forgalmat rögzítik speciális capture card-okkal.

Analizátorok csatlakoztatása a rendszerbe

Két kommunikáló fél esetén beiktathatunk egy dedikált analizátort is két állomás közé, de akár valamelyik állomáson is futtathatjuk az analizátor szoftvert.

Elosztott rendszerek esetén alkalmazhatunk elosztott analizátort, vagy egy alkalmas helyet keresünk az analizátor számára.

Számítógépes alkalmazások között a legelterjedtebb protokoll analizátorok a WireShark és TCPDUMP. A WireShark egy grafikus felületet nyújt, Linux alatt alatt képest akár USB-t is vizsgálni. A TCPDUMP egy karakteres felület, felépítése a következő táblázatban látható:

tcpdump	**- kapcsoló**	**szűrő kifejezés**
	- i: adott interfészen hallgatózik	előre definiált konstansokból, számokból és az *and*, *or* és *not* kifejezésekből áll
	- B: OS capture buffer méret beállítása	
	- D: használható hálózati interfészek listája	
	- w és - r: fájlból és fájlba működés, wireShark alkalmazással kompatibilis formátumban	

TCP és UDP különbségek

TCP	UDP
▪ Kapcsolat orientált	▪ Üzenet orientált
▪ Megbízható	▪ Nem megbízható
▪ Pont-pont ○ Unicast	▪ Multicast és broadcast is használható
▪ Folyamszabályozás és torlódásvezérlés	▪ Nincs korlátozás, az adótól függ
▪ Összetett	▪ Egyszerű
▪ Erősen optimalizált	▪ Nincs mit optimalizálni
▪ Erőforrás igényes (CPU és memória is)	▪ Nem igényel sok erőforrást
▪ Késleltetés ingadozik hiba esetén (újraadás)	▪ Nincs újraadás, az adat hiányzik (multimédia...)
▪ Kétirányú byte stream interface	▪ Csomag alapú interface

20. Socket programozás. Socket-ek típusai. Socket használata, tipikus munkamenet TCP és UDP socket esetén.

Mi is az a socket?

A socket nem más mint univerzális kommunikációs végpont, melyben konfigurálni lehet, hogy hogyan és milyen kommunikációs szolgáltatást igénybevéve kommunikáljon. A socket lényegében egy fájl leíró, és egy kernel és a user space közötti alacsony szintű interfészt határoz meg.

Socket létrehozása és paraméterek

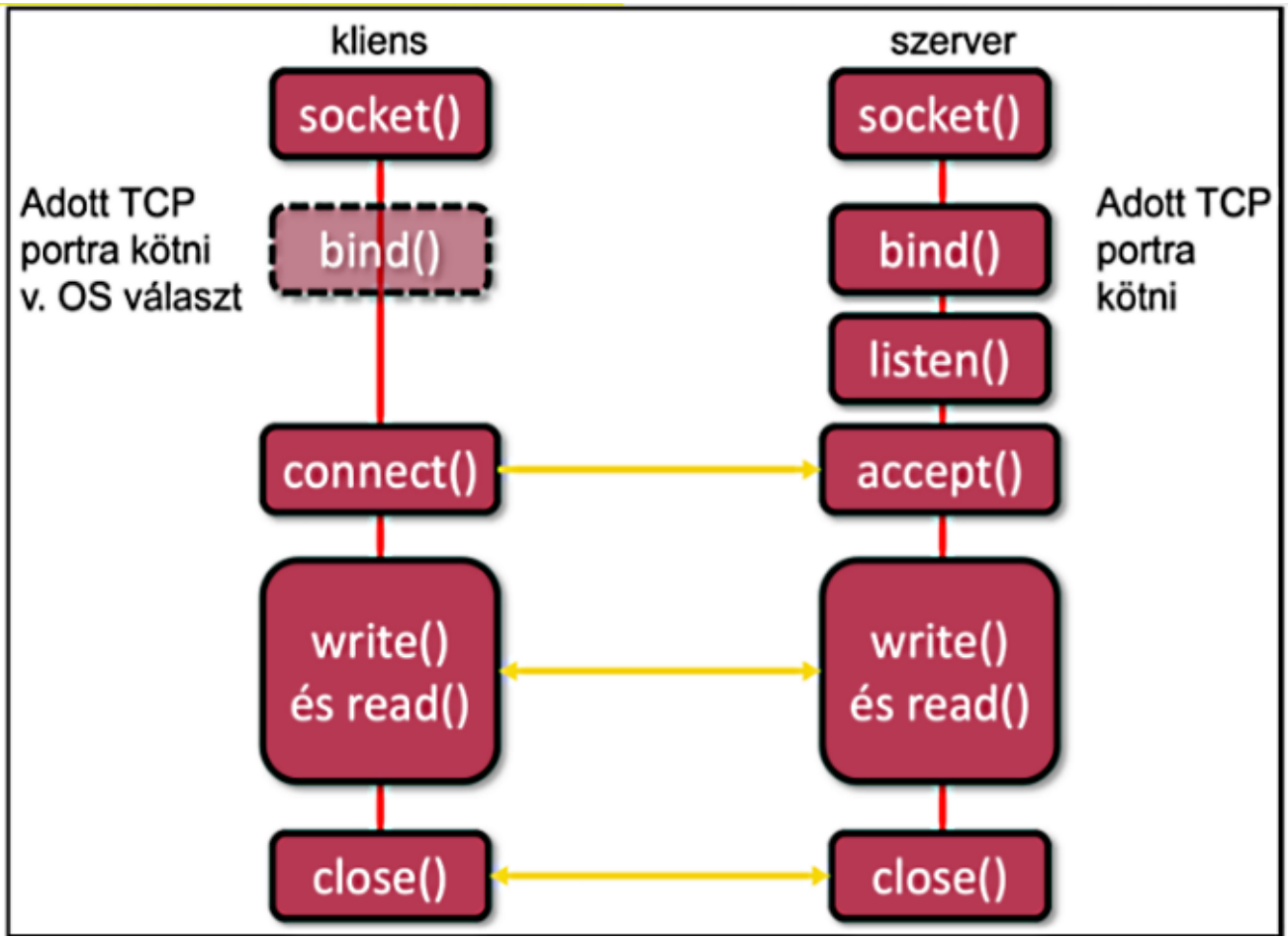
```
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

Nézzük a lehetséges paramétereket

int domain	int type	int protocol
UNIX domain socket	**SOCK_STREAM**: megbízható, sorrend helyes, kétoldalú, bytefolyam alapú átvitel (TCP)	**többnyire 0**-t adunk meg, az a default
IPV4 socket	**SOCK_DGRAM**: csomag alapú, kapcsolatfüggő, nem megbízható átvitel (UDP)	
IPV6 socket	**SOCK_SEQPACKET**: Csomag alapú, sorrendhelyes, megbízható, kétoldalú (kapcsolat orientált) szolgáltatás	
socket alapú kernel és user space közötti kommunikáció	SOCK_RAW: Alacsonyszintű hozzáférés a csomaghoz	
	SOCK_RDM: Megbízható, csomagalapú, de nem sorrendhelyes szolgáltatás	

socket() függvény visszatérési értéke a függvény azonosítója, hiba esetén -1-gyel tér vissza

TCP socket használata, munkamenet

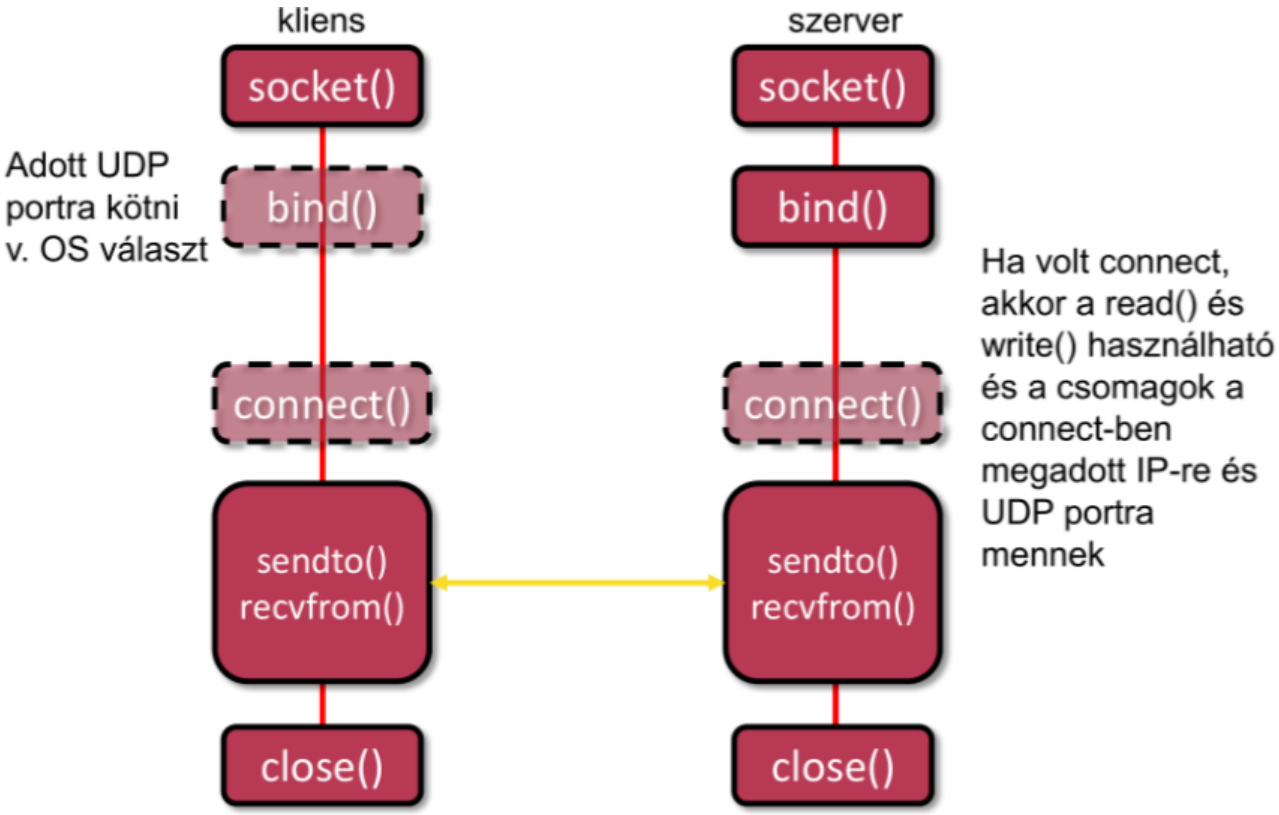


Nézzük részletesen a függvényeket

Függvény neve	Leírás
bind()	Lényegében egy adott, address family specifikus lokális címhez rendeljük a socket-et (pl. IP cím, UDP/TCP port). Szerver socket-nél kötelező, Kliens esetén a hívás elhagyása esetén a rendszer automatikusan választ egy portot.
listen()	(szerver oldalon): Socketet passzív állapotba tesszük, felkészítjük bejövő kliensek fogadására.
accept()	(szerver oldal): A visszatérési érték egy socket, amivel a kapcsolatot lehet használni. Erre a socket-re már csak adatátvitellel kapcsolatos függvényeket szabad meghívni. Beállítástól függően az accept blokkolhat vagy nem.
connect()	(kliens oldal): Kliens oldalon ezzel kapcsolódunk a szerverhez.
read() és write()	mint egy standard fájl esetében
close()	kommunikáció végeztével a socket-et lezárjuk

Valamennyi függvény esetén a paraméterek a socket azonosítója, a megfelelő sock_addr struktúra címe, és az adatbájtok hossza

UDP socket használata és munkamenet



Nézzük részletesen a függvényeket

Függvény neve	Leírás
bind()	Lényegében egy adott, address family specifikus lokális címhez rendeljük a socket-et (pl. IP cím, UDP/TCP port). Szerver socket-nél kötelező, Kliens esetén a hívás elhagyása esetén a rendszer automatikusan választ egy portot.
Cpnnect	Kapcsolat távoli végét adjuk meg (IP és PORT), amin keresztül, majd a send()-el kommunikálni fognak. Nincs kapcsolatfelvétel, beállítja a címet és azonnal visszatér
sendto()	Adatok küldése
recvfrom()	Adatok fogadása
Close()	kommunikáció végeztével a socket-et lezárjuk