

Pepi tételek

1. **Tétel** 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.

a. OS és HW kapcsolata

A modern beágyazott rendszerek szoftvere sok okból erősen párhuzamos, többnyire eseményvezérelt is, és az ilyen programok írása komoly kihívás...

- HW egyre komplexebb (multicore, heterogén, stb.)

PEP kapcsolata az OS-sel

- PEP könnyebb OS-sel
- De PEP lehetséges OS nélkül is
 - Spagetti kód
 - Round-robin architektúra
 - Round-robin architektúra megszakítással
 - Függvénytípusos ütemezés

Mi is az OS?

Hardware és alkalmazói szoftver közötti interfész

Erőforrás használat és aktivitások menedzsmentje és koordinálása

Lehetővé teszi az alkalmazások futtatását

Egy programozási felületet (API) nyújt az alkalmazások számára, hogy azok el tudják érni az OS szolgáltatásait

Hogy ezeket a funkciókat meg tudja valósítani „erősebbnek” kell lennie, mint az alkalmazói programok...

A beágyazott OS-ekre ezek nem mindig igaz...

- De a kivétel erősíti a szabályt!

b. OS alkalmazásának előnyei és hátrányai, valós idejű OS -ek

Valós idejű operációs rendszer:

- Felépítéséből következően megadott szolgáltatásai képesek valós időben működni.
 - Például a HW megszakítás után a megszakításhoz tartozó kód futása adott időintervallumon belül (többnyire néhány us) megtörténik
 - A futtatott alkalmazásnak is valós idejűnek kell lennie, a valós idejű OS csak lehetővé teszi a valós idejű működést
 - A Linux és a Windows nem ilyen, nem adható ilyen garancia
 - Mindkettőhöz létezik viszont ilyen garanciák megadását lehetővé tévő kiterjesztés (RTLinux, Windows: pl. Ardence RTX)

A rendszer adott eseményekre adott időn belül adott valószínűséggel válaszol (egyébként hibás a válasz, hiába funkcionálisan jó a válasz)

- Pl. Vetélkedő a TV-ben

Fajtái:

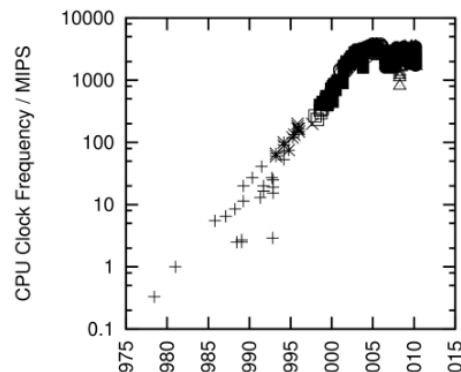
- **Lágy valós idejű (soft real-time): A valószínűség < 1, de elég közel van egyhez.**
 - Akkor használjuk, ha a határidők nem teljesítésének nincsenek katasztrofális következményei, azok bizonyos szintig elviselhetők
 - A rendszer „időnként” késhe
 - Service Level Agreement (a szolgáltatás minősége)
 - NEM feltétlenül prioritásos a működése, de leggyakrabban az
- **Kemény valós idejű (hard real-time): A valószínűség = 1.**
 - Ha nem válaszol időben, a válasz rossz...
 - A határidők nem teljesítésének katasztrofális következményei vannak
 - „A rendszer NEM késhe!”

+ PEP könnyebb OS-sel

c. HW architektúrák, Cache koherencia

Miért van szükségünk sok azonos magra?

- Nem lenne szükségünk, de...
- Sajnos a félvezető technológia bajban van!
 - Max. órajel nem nő
 - Fogyasztás nem nőhet



Miért van szükségünk heterogén architektúrákra?

- CPU, DSP, FPGA, GPGPU, stb. egy rendszeren belül
- Sok feladat nem hajtható végre hatékonyan általános architektúrákon

Igazából a kérdés az architektúra

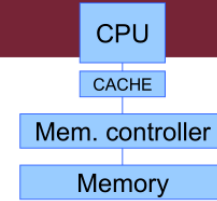
- CPU architektúra
 - Milyen utasítások vannak és azokat hogyan hajtja végre az architektúra?
- Hogyan vannak összekötve a végrehajtó egységek?
 - Amik kódot tudnak futtatni és adatokon műveleteket végezni
- Hogyan érik el az adatok a memóriát?
 - Amiben a kód és az adat van
 - I/O is leképezhető memóriára
- Hogyan érik el a végrehajtó egységek a perifériákat?
 - Amivel a külvilághoz és más, nem memórián keresztül csatlakoztatott végrehajtó egységekhez férnek hozzá
 - Megszakítás a kulcs lesz...

Linux és a FreeRTOS is rengeteg CPU architektúrát támogat

- Egy jól megírt kódot csak újra kell fordítani

Oprendszer hátránya: központi szerep, ezért az oprendszer megfelelő működése rendkívül fontos a rendszer működése szempontjából

Single CPU (Uniprocessor)



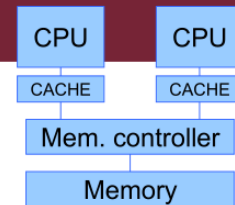
Egy végrehajtó egység (single CPU)

- Néhány évvel korábban ez volt a jellemző
- Beágyazott rendszerekben ma is ez a jellemző!
 - A MCU-k (mikrovezérlő) jellemzően egymagosak
 - Teljesítményben skálázhatóak (architektúra+órajel) egyelőre

DMA, ha van, párhuzamosan kezeli a memóriát a CPU-val

- Versenyhelyzet a DMA vezérlő és a CPU között
 - Input: DMA transzfer $\Rightarrow$ IT $\Rightarrow$ CPU kezeli az adatokat
- A CACHE koherencia sérülhet!
 - Egész CACHE érvénytelenítése/tiltása (a transzfer alatt)
 - Egyszerű, de katasztrofális a hatása a teljesítményre
 - DMA-val kezelt memóriaterületre tiltani kell a CACHE-t (pl. MMU, később)
 - CACHE koherens DMA (HW támogatás)
 - **A CACHE koherencia mindig előkerül, bármikor okozhat gondot!**

SMP



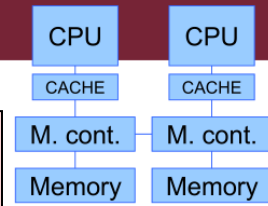
Symmetric multiprocessing

- Több, azonos végrehajtó egység
 - Több CPU, vagy CPU mag
 - Például: AMD és Intel modern CPU-i
- Esetleg saját CACHE hierarchiával
 - CACHE koherencia biztosításával
 - Szinte minden SMP architektúra CACHE koherens
- Közös buszon/vezérlőn a memória
 - A memóriát minden végrehajtó egység azonos tulajdonságokkal (sebesség, késleltetés) éri el
 - Szorosan csatolt architektúra
- Muticore MCU-k megjelenése
 - ARM15, ARM11 MPCore, ARM Cortex-A9 MPCore
 - A CPU mag olcsó (relatív kis felületet használ a félvezetőn)
 - A beágyazott területen is szerepet kap az SMP
- OS támogatás kell hozzá (egyébként 1 CPU látható)

NUMA

Non-Uniform Memory Access

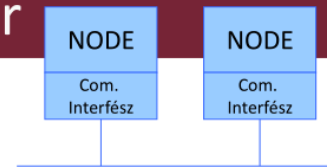
- A memória egyes része „közelebb” vannak egyes végrehajtó egységekhez mint másokhoz
- Összefüggő fizikai memória
- Cache koherencia
 - CACHE koherens (ccNUMA)
 - nem CACHE koherens (Ha igen, ki kezeli, az OS vagy a felhasználó program?)
- A memóriavezérlők egy speciális kommunikációs felületen csatlakoznak
 - QPI az Intelnél, vagy Hypertransport az AMD-nél
- Pl. AMD Opteron vagy a Intel XEON szerver processzorok több CPU esetén ccNUMA architektúrát is használhatnak (CPU-n belül SMP van)
- OS támogatás (egyébként 1 CPU látható)



Elosztott rendszer

Nincs közös memória
Minden NODE tartalmaz

- Végrehajtó egységet/egységeket
- Memóriát



A NODE-ok egy kommunikációs interfészen keresztül üzenetekkel kommunikálnak (azán csatoltak)

- A kommunikáció SW alapú
- A kommunikációs interfész lehet:
 - CAN, FlexRay pl. autóiparban
 - Ethernet, TTEthernet felett TCP/IP ipari folyamatoknál vagy repülőiparban (AFDX)

A kommunikációs interfészen keresztül adatküldés:

- Lényegesen alacsonyabb megbízhatóságú, mint a szorosan csatolt memórián keresztüli kommunikáció
 - Számítani kell az üzenetek elvesztésére, duplikálódására, sérülésére, sorrendjüknek a megváltozására
 - Két hadsereg probléma
 - Rosszindulatú hibák (system security)
- Nagyobbak a késleltetések és a késleltetések ingadozása (jitter) is
- Kisebb a sávszélessége mint a memóriának

d. Spekulatív végrehajtás

A compiler és a CPU egy szekvenciális kódban garantálja azt, hogy a program eredménye konzisztens (program kauzalitása megmarad)

- A compiler átrendezheti az utasítások végrehajtási sorrendjét
- A processzor átrendezheti az utasítások végrehajtási sorrendjét (out of order execution)
- Cél a gyorsabb végrehajtás, a köztes állapotok nem érdekesek valójában a kauzalitás fenntartása mellett

Ez nagyon sok esetben **nagyon nehezen detektálható** véletlen hibákat okoz:

- Párhuzamos eseményvezérelt programokban
- Direkt I/O-t végző programokban
 - akár memory mapped I/O esetén is

Mit ír ki példa

Az x és az f változók azonos memóriaterületekre hivatkoznak mindkét kódban

Processor #1:

```
x = 0;  
f = 0;  
while (f == 0);  
print x;
```

Processor #2:

```
x = 42;  
f = 1;
```

Néha lesz olyan, hogy 0-át fog kiírni pl. egy x86-os PC

- Out of order execution
 - A CPU valami miatt felcseréli a 2. CPU által végrehajtott értékadó utasításokat
 - Az f már be van állítva, de x még nem
 - Ekkor 1. CPU kiírja x értékét
 - Megteheti, azok függetlenek egymástól az adott CPU-n
- Kivéve pl. egy régebbi Intel ATOM-os gépet (az „in order”...)

Ha a megadott helyekre egy a memória hozzáféréseket explicit sorrendező speciális gépi utasítást rakunk, akkor jól fog működni

- A C/C++ „volatile” kulcsszó nem segít (compiler és HW függő), az csak egy CPU-ra vonatkozik, és lényegében a compiler-nek szól
- CPU átsorrendezheti

Megoldás: Barrier vagy memory fence

- A Linux alapesetben „beteszi” őket a szükséges helyeken
- Felhasználói programokban nem lehetne rájuk szükségünk
- De operációs rendszer magjában (védett módban futó kód) szükséges

e. Megszakítás és típusai

Típusai

- **Hardver: periféria használja értesítésre**, időzítő, hálózati kártya, stb. jelzi a kiszolgálási igényét
 - Külső esemény hat a processzor működésére
 - Egy periféria számos okból kérhet kiszolgálást
- **Kivétel (exception) : a rendszer (OS és alkalmazások) futása közben történő esemény (laphiba, numerikus túlcscordulás, nullával történő osztás, védelmi szintnek nem megfelelő működés, vagy egyéb forrás)**
 - A processzor belsejéből származó esemény, hibát vagy valamilyen szoftveres kezelési igényt jelent
- **Szoftver esemény (pl. rendszerhívás)**, speciális utasítás végrehajtása
 - A szoftver hat a CPU működésére, megváltoztatja azt

A modern operációs rendszerek megszakítás vezéreltek.

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

a. Védelmi szintek

Védelmi szintek

- **Végrehajtható utasítások korlátozása működési módtól függően**
 - Pl. kernel/user mode (2 vagy 4 mode jellegzetesen)
 - Pl. user mode: I/O utasítások, CPU speciális regiszterek elérése tiltott
 - **Működési mód váltás megszakítással**
 - **Alacsony szintű CPU erőforrásokhoz történő hozzáférést szabályozza**
 - Végrehajtható utasítások
 - CPU konfigurációs regiszterek
 - I/O végrehajthatósága
 - Memória területek elérése
 - Pl. x86 (286/386-től), ARM Cortex Ax sorozat, stb.
 - Mikrovezérlők (microcontroller) nem támogatják, pl. Atmel AVR, ARM Cortex Mx sorozat, stb.
- 2 vagy 4 szint (esetleg egy 5. a HW virtualizációra)**

 - Jellegzetesen 2-öt használunk
 - User mode (real mode) – korlátozott elérés
 - Kernel mode (protected mode) – korlátlan elérés
- Esetleg egy 3. is felhasználásra kerül
 - Driver és/vagy kernel service mode

b. Rendszerhívás és kontextus

Rendszerhívások: OS szolgáltatásainak használatát teszik lehetővé (pl. Linux esetben, amikor aktívak a védelmi szintek)

Rendszerhívás

- Részletek:
 - HW implementáció architektúra függő (x86-os esetén CPU típus függő)
 - ABI, Application Binary Interface
- Egy OS-nek egy bizonyos OS platformon lehet több ABI-ja...
 - Pl. Linux ARM régen...
 - ABI változhat

Mi az a rendszerhívás?

- Kiindulás: A processzor alkalmazói programot futtat
- A rendszerhívás lényegében megszakítja azt (szoftver megszakítás), és átadja a vezérlést a kernelnek (állapot mentés és visszaállítás, context switch)
- A kernel elvégzi a munkáját
- Visszaadja a futás jogát az alkalmazói programnak (állapot mentés és visszaállítás, context switch)

Hogyan történik a rendszerhívás?

- Erősen implementáció függően...

Mi a következménye a rendszerhívásnak

- Nagy overhead-je van, sok CPU időt emészt fel
 - Rendszer állapotának mentése és visszaállítása 2 alkalommal
- Minimalizálni kell az alkalmazott rendszerhívások számát
- A CPU védelmi szintet is vált: user-kernel-user

A rendszerhívás nem vermet használ a paraméterek átadására, teljesen máshogy működik, mint egy függvény vagy eljárás hívás!

Rendszerhívás a programozó szemével

A programozó az API függvényt hívja meg a használt programozási nyelven

- Pl. C nyelv és Linux esetén
 - Linux: glibc vagy egyéb pehelysúlyú C könyvtárak (korlátozott memória)
 - Egy jelentős része többé-kevésbé (mostly) megfelel a POSIX API szabványnak
 - IEEE Portable Operating System Interface (POSIX) szabvány
- Az API rejtje el a programozók elől a rendszerhívás részleteit
 - Lényegében egy magas szintű C wrapper a rendszerhívások körül
- Az API megvalósítása az alkalmazásba fordul bele, vagyis felhasználói módban fut, és kiváltja a rendszerhívást

Nagyon indokolt esetben készítünk új rendszerhívásokat vagy módosítunk meglévőket!

- Nagyon „vonzó” lehetőség és tulajdonképpen egyszerű is
- Kompatibilitás, hordozhatóság sérül, csak nagyon komoly megfontolások után lehet indokolt
- Többnyire „újra feltalálnánk a kereket” ilyenkor
 - Ahhoz hasonló (sokkal jobb), amit kitalálnánk, az már létezik, csak nem ismerjük...

Szükség esetén hívhatjuk őket direktbe, de nem ajánlott

Mi a része a kontextusnak?

CPU regiszterek

- PC, Státusz regiszter, munkaregiszterek, szegmens regiszterek, stb.
- Pl. Linux esetén a kernel kontextusnak nem részei a lebegőpontos regiszterek
 - Következmény: A Linux kernelben nem használhatunk lebegőpontos aritmetikai műveleteket!

MMU beállításai

- Az adott alkalmazás memóriájának eléréséhez szükséges beállítások

Egyéb alkalmazás specifikus HW beállítások

Az egyik legkomolyabb feladat meghatározni, hogy mit kell, és mit érdemes elmenteni/visszaállítani.

c. OS-ek fajtái, és azok kiválasztása

A választás platformtól és alkalmazástól függ + piaci modellek

CPU (tipikusan MCU) védelmi szintek és MMU nélkül

- FreeRTOS pl.

CPU/SoC védelmi szintekkel és MMU nélkül

- Linux pl.
- Az architektúra tud FreeRTOS-t futtatni
 - Nem indítjuk el a védelmi szinteket és az MMU-t (pl. x86-on)

Ingyenes

- Tisztázni kell, hogy pontosan milyen felhasználásra és feltételekkel, jogi következmények lehetnek
- Pl. Debian Linux, általában a GNU licenc problémás lehet egyes esetekben (a licenc továbbterjedhet)
- Oktatási célra a legtöbb ingyenesen elérhető, persze terméktámogatás nélkül

Terméktámogatás és akár egyes tulajdonságok (feature) fizetősek, de van szabadon elérhető verzió is

- Red Hat Linux, Ubuntu, FreeRTOS (SafeRTOS fizetős), eCos, µC/OS-II

Fizetős

- Fejlesztői licenc (developer license)
 - Többnyire fordítót és fejlesztőrendszert is tartalmaz
- Darabár (royalty)
 - Összetett OS-ek esetén tulajdonságokként, komponensenként, konfigurálható formában kerül az ár megállapításra
 - WxWorks, QNX, Windows CE, stb. ilyen modellel dolgozik

d. Beágyazott oprendszerek csoportosítása

Hard real-time és ha igen, milyen paraméterekkel

- Nem real-time: Linux, Windows Embedded, Windows CE régebbi verziói
- Real-time: Milyen szolgáltatások tekintetében?

Milyen tanúsítványai vannak

- Biztonság (security értelemben)
- Megbízhatóság
- Kódminőség (Pl. MISRA C kompatibilis)

A kisbetűs részt mindig el kell olvasni (nem csak itt)

- Többnyire erősen korlátozóak a tulajdonságok tekintetében

Kód hozzáférhetősége (egyáltalán nem, részben vagy egészben)

- Windows CE 99%-ban elérhető forrású, de nem fordíthatom újra, mert a leglényegesebb részek nem érhetőek el!
 - És jogom sem lenne újrafordítani...
- Sokat segíthet a fejlesztés során felmerülő problémáknál, ha megnézhető a kód, esetleg újrafordítható az OS
- Új processzorokra, processzor verziókra portolás

Piaci modell (Szabad szoftver, fejlesztői licenz, darabár)

e. Beágyazott OS-ek komplexitása

Egyszerű kernelek (5-100k prog. mem., forráskód/lib, egybefordul a kernel és az alkalmazás, MMU és védelmi szintek nem kellenek)

- uC-OS
- FreeRTOS

Közepes komplexitású (40k-2M prog. mem., forráskód/lib, egybefordul a kernel és az alkalmazás, MMU és védelmi szintek alkalmazás OS függő)

- eCos
- Rtems
- Nucleus

Nagy komplexitású (500k + prog. mem., lehet telepíthető, külön alkalmazás, fájlrendszerből bootolás, pl. HDD support, MMU és védelmi szinten kellenek)

- WxWorks
- QNX
- Linux, Embedded Linux és RT Linux variánsok
- mcLinux (MMU nélküli CPU-n futtatható Linux)
- WinCE, Windows Embedded

f. High end vs Low end (Linux VS FreeRTOS)

Szempont	High end	Low end
Alkalmazási terület	általános célú alkalmazás és processzor számára	mikrokontrollerek
támogatás	folyamat, szál, virtuális memória	csak szál és memóriavédelem
Mem + védelem	védelmi szintek és MMU kell	opcionális
előny	hatékony, gyors fejlesztés összetett alkalmazások esetén is	egyszerű, hatékony futási időben egyszerű alkalmazások esetén, jól tesztelhető
hátrány	nagy komplexitás és futási idő, nagy erőforrás igény, tesztelési nehézségek	nincs védelem OS szinten, MPU inkább egy periféria ami megszakításokat csinált
oprendszer	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	driverok kezelik kernel módban, megszakításokat az OS fogadja	HW és IT-t speciális programkönyvtárak kezelik

g. OS felépítése

Réteges szerkezet

- Strukturáltság és futási idejű hatékonyság optimumát kell megkeresni
- A rétegek határán egy virtuális gép valósul meg
 - Magas szintű funkciók, virtuális utasításkészlet
- A legalacsonyabb szinten a valódi CPU és perifériák által megvalósított valódi gép található

Rétegek

- Kernel (az operációs rendszer legalapvetőbb funkcióit tartalmazó rendszermag)
 - Feladatok kezelése, tár (memória) kezelés, védelmi és biztonsági funkciók
- Hardver közeli réteg (driverok, többnyire valamilyen hierarchiába rendezve)
 - Alacsony szintű hardver kezelés, hálózat, keyboard, egér, képernyő, stb.
- Rendszerprogramok (rendszerfolyamatok és alrendszerek az egyéb funkciók megvalósítására)
 - Filerendszer, magasabb szintű hálózatkezelés (TCP/IP), parancsértelmező, stb.
- Felhasználói programokból érkező rendszerhívások fogadó felületét megvalósító réteg
 - API megvalósítása különböző célnyelveken (target language)
 - Az API leképezése rendszerhívásokra
 - Védelmi szintek váltása
 - Rendszerhívás ellenőrzése (paraméterek, jogosultságok, stb.)
 - A feladatok elvégzése (kernelben)

Monolitikus kernel

- Az összes szükséges funkció összefordítva egyetlen programba
- Nem flexibilis, a lehetséges hardver elemeket be kell fordítani a kernelbe
- Egy komponens hibája a teljes operációs rendszer hibáját okozza
- Beágyazott rendszerekre jellemző (nem változik a HW)

Moduláris kernel

- Minimális kernel, és utólagosan, igény szerint, dinamikusan betölthető kernel modulok, flexibilis
- Egy komponens hibája a teljes operációs rendszer hibáját okozza
- Újabb Linux kernel, Windows

Mikrokernel

- Minimális funkciókkal rendelkező kernel, és ahhoz kliens-szerver architektúrában csatlakozó szolgáltatások
 - Többnyire 3 védelmi szintet használva
- Erőforrás igényesebb
- A kernel védett még a hozzá csatlakozó szolgáltatásoktól is
- Mac OS X

Exokernel (új kutatási terület)

- Direkt alkalmazás hozzáférés egyes perifériákhoz (hatékonyság)

h. Példák, Linux és FreeRTOS felépítése

Linux

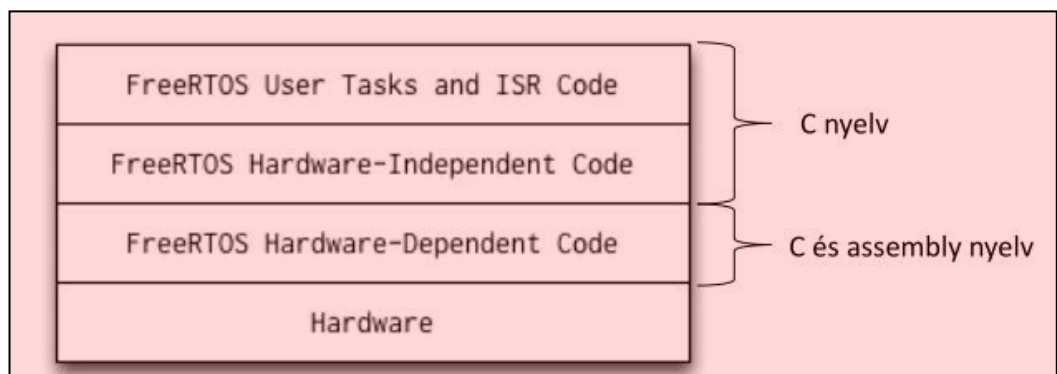
- Alapesetben egy monolitikus kernel mag és modulárisan betölthető kernel modulok alkotják
 - Modul kezelő parancsok: modprobe, insmod, lsmod, rmmod
- Képes monolitikus kialakításban működni
 - Minden drivert és szolgáltatást belefördítünk a kernelbe
 - A modul kezelésre sincs szükség ekkor, kihagyható
 - Nem flexibilis, de sokszor erre nincs is szükség
 - Pl. Nem változik a HW
 - Ez beágyazott rendszerekben előnyös (kis méret, gyorsabb)

Windows

- Moduláris kernel

Apple OS X

- Darwin
 - Mach 3.0 mikrokernél + FreeBSD (Berkeley Software Distribution) UNIX
 - Erősen objektum-orientált keretrendszer
 - Hatékony, mert kevés típusú HW-re erősen optimalizált



Linux felépítése

SCI

- Bejövő rendszerhívások ellenőrzése
- Jogosultság és paraméterek helyessége

PM és MM: Futó feladat és annak a memóriájának a kezelése

VFS: A Unix-ban minden file

- Majdnem, vannak kivételek...

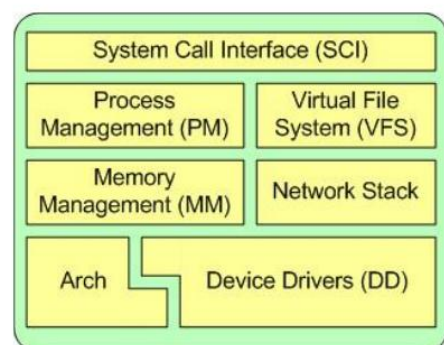
NS: Az egyik kivétel

Arch: Kernel HW architektúra függő része

- HAL tulajdonképpen
- Ezt kell átírni új CPU architektúrára portoláskor

DD: Eszközkezelők

- Eszköz specifikus, C-ben



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

a. Szekvenciális programozás

Szekvenciálisan végrehajtott probléma megoldás

- A kód szekvenciálisan (egymás után) végrehajtott utasítások sorozata, amit az utasításokban felhasznált adatok vezérelnek (ciklusok, elágazó utasítások).
- Más megfogalmazásban: A szekvenciális program utasításainak a végrehajtási sorrendjét a programban megfogalmazott vezérlési szerkezetek önmagukban definiálják.
- Fontos: A program nem csinálja mindig ugyan azt, az függ a bemenő adatoktól (a program adatvezérlés)
 - Kérdés: Hogyan és mikor jelenik meg az adatokban lezajló, kívülről jövő változás?
 - Az adatvezérelt programban csak akkor veszi észre a változást a program, ha azt az adatot valamilyen utasításban felhasználja...

- A matematikai gondolkodás többnyire szekvenciális

Bizonyos problémák (a legtöbb) szekvenciálisan csak nehezen oldhatók meg...

- Mert többnyire természetesen párhuzamosan kell dolgokat csinálnunk, amit nehéz sorossá tenni...

b. Párhuzamosság

A világ természetesen párhuzamos!

- A dolgok párhuzamosan történnek, a tudatos emberi gondolkodáson kívül minden párhuzamosan zajlik...
 - Mivel a tudatos emberi gondolkodás szekvenciális lesz gondunk...
 - A legtöbb programozónak nem megy a párhuzamos programozás, mert szekvenciális gondolkodásra nevelték
 - Kivétel: Villamos és gépészmérnökök (nekik jobban megy az átlagnál)
 - Az általuk kezelt világ (elektronika, mechanika) párhuzamos
- A világ egy erősen párhuzamos rendszer, amelyben a párhuzamos részfeladatok erősen összefüggnek:
 - Pl. atomi szinten
 - Pl. az emberi társadalom
- Akkor oldjuk meg a feladatainkat párhuzamosan, ne kényszerítsük azokat egy soros modellbe!

c. Párhuzamos programozás története

A számítógépeink is régóta képesek a programok „párhuzamos” végrehajtására:

- A UNIX több mint 40 éves, pont erre készült...
 - És előtte voltak kísérletek
- A modern OS-ek ezt teszik lehetővé, pl. dolgozunk és MP3-at hallgatunk közbe
 - Ezek független feladatok persze

- Kérdés: Milyen OS futott az első igazi (a mai értelemben vett CPU-t tartalmazó) beágyazott rendszeren?
 - Válasz: UNIX (mérés- és irányítástechnikai kísérletek a 70-es évek elején PDP-vel)

Hogyan futnak a feladatok párhuzamosan valójában?

- Ha több végrehajtó egységünk van, mint ahány feladat, akkor nincs gond...
- De sok esetben 1 végrehajtó egységünk volt
 - És az MCU-val megoldott feladatok esetén még ma is ez a jellemző

Erőforrás megosztás és gazdálkodás:

- Közös lónak túros a háta...
- Megosztás: az erőforrást többen használnák párhuzamosan, a helyes használat a kérdés
 - Erőforrás: CPU, memória, perifériák, stb.
- Gazdálkodás: erőforrások optimális kihasználása a kérdés
 - CPU kihasználtság, várakozási idők, stb.

d. Kis történelemóra

1950-1965 : 1 számítógép szervezetenként (cég, egyetem)

- Sok, egymást nem ismerő felhasználó
- Függetlenek a feladatok
- Elsősorban erőforrás megosztás a feladat
 - Végrehajtó egység
 - Memória és diszk
- A számítógépek nem kommunikálnak egymással
 - Kivéve a kisteherautón szállított lyukkártyákat

1965-1980 : 1 számítógép divízióként (osztály, tanszék)

- Kevesebb felhasználó jut egy gépre
- Feladatok egymástól függése megjelenik
 - A felhasználók ismerik egymást
 - Tudatosan, együtt dolgoznak feladatokon
- Távoli kommunikáció megjelenik
 - Arpanet, az Internet elődje
- Első beágyazott rendszerhez hasonló megoldások megjelennek
 - Kommunikáció az alapjaiban párhuzamos fizikai környezettel
 - Ipari mérőrendszerek és folyamatirányítás

1980-1995 : 1+ számítógép felhasználóként

- A személyi számítógép megjelenése, elterjedése és gyors fejlődése
- Párhuzamos, erősen összefüggő feladatok (kísérletezés)
 - Multiprocesszoros rendszerek (SMP, NUMA, pl. Cray és SGI)
- Számítógép alapú kommunikáció elterjedése
 - Ethernet és TCP/IP (Internet) egyeduralkodóvá válik
- Mikrovezérlő (MCU) megjelenése

- Speciális, ma beágyazott rendszereknek hívott, alkalmazási körre dedikált processzorok
- Fizikai folyamatok számítógéppel történő műszerezése és irányítása mindennapossá válik hatására

1995- : több számítógép felhasználóként

- Preemptív asztali operációs rendszer (Windows 95, LINUX)
- Elosztott rendszerek elterjedése, WEB-es technológiák

Beágyazott rendszerek elterjedése

- Mindenhol jelenlevő számítástechnika
 - Hány mikrovezérlő van a lakásomban vagy az autómban?
 - Hány mikrovezérlő segíti az életemet?
- Beágyazott operációs rendszerek elterjedése
 - Rengeteg fajta, lefedve a széles alkalmazási kört
 - Mind egy kicsit máshogy működik...
- Elosztott beágyazott rendszerek (TCP/IP, CAN, etc.)
 - Internet és technológiai (TCP/IP)

- Valós idejű kommunikáció elosztott beágyazott rendszerekben
- Vezeték nélküli kommunikáció
- Szenzorhálózatok

e. Párhuzamos program

Konkurens program/rendszer névvel is illetjük

- Olyan rendszer amelyben több, egymással együttműködő, párhuzamosan futó részfeladat együtt valósítja meg a teljes feladatot...

- Részfeladat, feladat (Task, Subtask)?

- Egyelőre általában beszélünk róla, később ebből lesz a folyamat (Process) és szál (Thread)

- A párhuzamos rendszerekben egymással párhuzamosan futó, önmagukban szekvenciális programrészeket, amelyek egymás nélkül többnyire értelmes működésre nem képesek, részfeladatoknak nevezzük

- Részfeladat:

- A végrehajtásához végrehajtó egységhez kell rendelődnie

f. Típusfeladat 1

N részfeladat és M (N-nel összemérhető számú) CPU

- Sok végrehajtó egység és egy nehezen részfeladatokra bontható feladat

- High performance computing

- Végrehajtó egység

- Szuperszámítógépek
- Grid
- Multicore CPU, GPGPU, FPGA

- Kihívás: Az algoritmus (feladat) hatékony szétosztása rendelkezésre álló végrehajtó egységekre

- **Nem ezzel foglalkozunk most!**

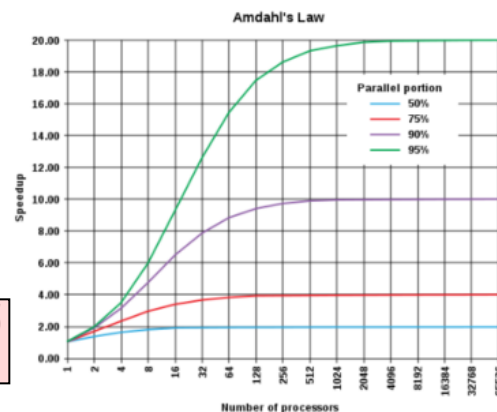
g. Amdahl törvénye

Egy adott feladatkészlet végrehajtási ideje hogyan függ a rendelkezésre álló végrehajtó egységek számától?

- Várhatóan nem egy 1:1 leképezés lesz (pl. 1-ről 4 CPU eredménye nem 4x gyorsulás)
- $S_{latency}(s)$: A végrehajtás gyorsulása a maximálisan elérhető gyorsulás függvényében (pl. 4 CPU esetén 4x-es)
- s : A maximálisan elérhető gyorsulás a végrehajtott kód azon részére, ami tökéletesen párhuzamosítható
 - Kb. a végrehajtó egységek száma...
- p : Azon kód aránya, ami párhuzamosítható

A párhuzamosítási törekvések egy idő után kevés eredménnyel járnak (**law of diminishing returns**)

$$\text{Amdahl törvénye}$$
$$S_{latency}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$
$$S_{latency}(s) \leq \frac{1}{1-p}$$



Daniels220 at English Wikipedia

h. Gustafson törvénye

Amdahl törvénye konzervatív

- Nem veszi figyelembe, hogy a probléma mérete is nő/változik és ezzel a párhuzamosság is kihasználható többnyire

Kicsit „realisztikusabb” Gustafson törvénye

- $S_{latency}(s) = 1 - p + sp$
- Lineáris növekedés, 1-nél kisebb, a párhuzamosíthatóságtól és a végrehajtó egységek számától függő meredekséggel...

i. Típusfeladat 2

N részfeladat és 1 (vagy N-nél jóval kevesebb) CPU

- Egy vagy kevés végrehajtó egység és magukban párhuzamosan végrehajtandó, többé-kevésbé könnyebb azonosítható részfeladatok
- Erőforrás megosztás
- A feladat maga párhuzamos, a kihívás a párhuzamos részfeladatok, és az egész program hatékony és helyes megvalósítása
 - Tipikus feladat beágyazott rendszerekben

- **Ezzel fogunk a továbbiakban foglalkozni!**

j. Eseményvezéreltség

Esemény rendszer életében lezajló változás vagy történés
A program lefutása függ a program működés során lezajló történésektől (esemény)

- Esemény/jelzés (event, signal)

- A rendszer eseményekre reagál

- Reaktív rendszer
- A válaszidő fontos

- Esemény lehet:

- HW megszakítás:
 - Input a felhasználótól (egérkattintás)
 - Input a külvilágból (új mért érték, detektált meghibásodás pl. az irányított folyamatban)
 - Adatcsomag beérkezése a kommunikációs interfészen
- SW megszakítás: A program belső állapotában lezajló tudatos változás (rendszerhívás, system call)
- Kivétel (exception): Nullával való osztás, túlcsordulás, stb.

Az esemény összetett adattípusként (struktúra vagy objektum) jelenik meg a kódban

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

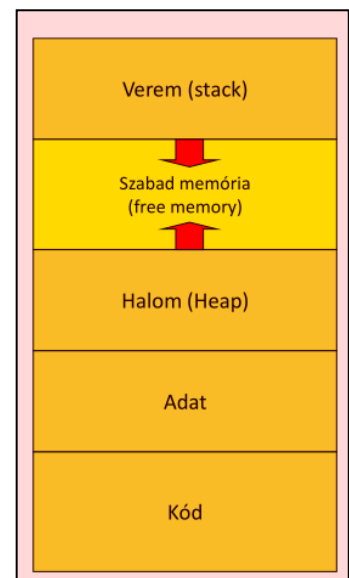
a. Feladat, Task

Részfeladat, **feladat (Task, Subtask)?**

- Egyelőre általában beszélünk róla, később ebből lesz a folyamat (Process) és szál (Thread)
- A párhuzamos rendszerekben egymással párhuzamosan futó, önmagukban szekvenciális programrészeket, amelyek egymás nélkül többnyire értelmes működésre nem képesek, részfeladatoknak nevezzük

A feladat több mint a program:

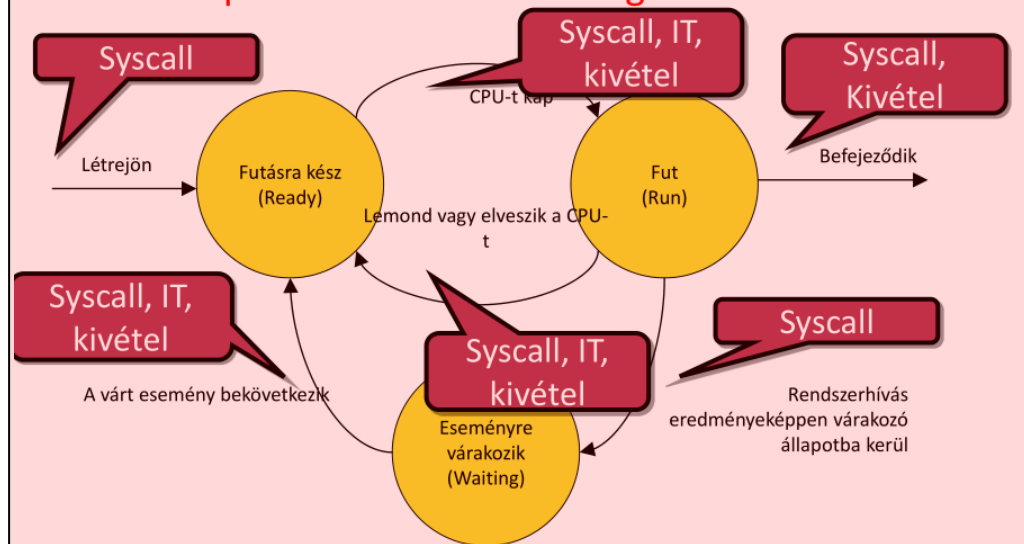
- Aktív, nem passzív!
- Állapota van (hiszen egy futó program):
 - Minimálisan: Fut (run), vár (waiting), futásra kész (ready)
 - Erősen OS függő állapotok és állapotátmenetek
 - Részletesen fogunk vele foglalkozni
- Adatszerkezetek adattal vannak feltöltve (függ a feladat előéletétől):
 - Adat terület (globális adatok ☛)
 - Verem (stack)
 - Halom (Heap)
 - Erősen egyszerűsített ábra jobbra



b. Taskok állapotai

Minden állapotátmenet megszakításra történik!

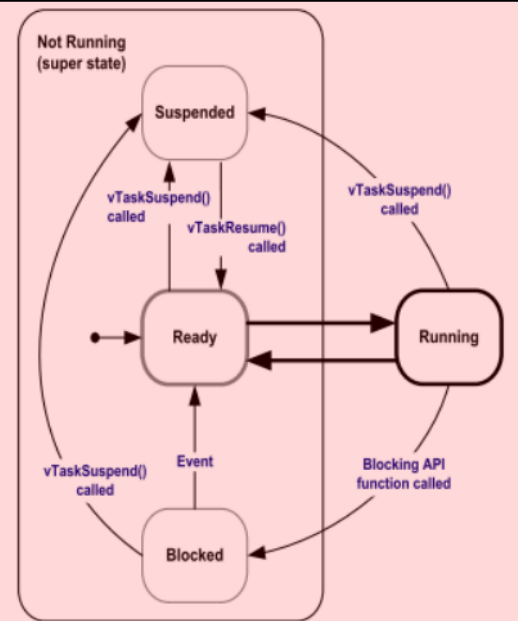
A modern operációs rendszerek megszakítás vezéreltek!



c. FreeRTOS Task állapotok

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

(http://www.freertos.org/Documentation/RTOS_book.html)

d. Feladat leíró

Adatstruktúra a feladattal kapcsolatos adatok **operációs rendszeren** belüli tárolására

- A FreeRTOS-ban konkrétan így is hívják...

Ez alapján végzi az OS a feladatok kezelését

- A feladat bizonyos részéhez hozzáférhet rendszerhívással (többnyire lekérdezheti)

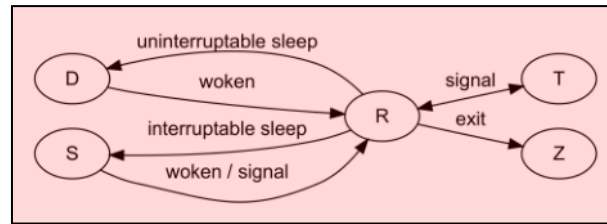
Mit tartalmazhat (nem teljes lista)?

- Task ID (Process ID vagy Thread ID az implementációkban)
- Állapot (Futásra kész, Fut, Eseményre vár, stb., OS specifikus)
- Utasítás számláló (Program Counter, mentés/visszaállítás)
- CPU regiszterek (Registers, mentés/visszaállítás)
- CPU ütemezéssel kapcsolatos információk
- Memória kezeléssel kapcsolatos információk (pl. virtuális memória)
 - MMU állapot (mentés/visszaállítás)
- Jogosultságokkal kapcsolatos információk (tulajdonos, ACL)
- I/O státusz információ (használt I/O erőforrások, és azok állapota)

e. Linux folyamat állapotok

A „man ps” paranccsal megtudható...

- D un interruptible sleep (usually IO)
- R running or runnable (on run queue)
- S interruptible sleep (waiting for an event to complete)
- T stopped by job control signal
- 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 its parent



ready – Futásra kész

blocked – Várakozik valamilyen erőforrásra

running - Fut

terminated – A szál normálisan leállt, de az erőforrásait még nem lehetett felszabadítani, mert még függőben vannak rá „hivatkozások”

- Kb. mint a zombi folyamat állapot, csak folyamaton belül más szálakra vonatkozóan

Állapot-átmeneti diagram?

- Standard + terminated state
 - Terminated state: kb. minden állapotból mehet egy szál ebbe
 - De az elérhető dokumentációk nem egyértelműek
 - A kernel kódot meg nem néztem meg...

5. **Tétel** 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.

a. Feladatok ütemezése

Feladat: Kiválasztani a futásra kész feladatok közül a futót (visszatértünk a feladat megnevezéshez)

- Statikus/dinamikus ütemezés
 - Dinamikus ütemezés a tárgy keretein belül

A feladatokat a többnyire feladat sorokban (task queue vagy job queue) tároljuk

- Ez a legegyszerűbb esetben tényleg egy FIFO, ami TCB típusú objektumokra/struktúrákra mutató pointereket tartalmaz
- Pl. futásra kész sor, adott eseményre vár sor, stb.
- Az ütemezés sorbaállási modellje (Queuing diagram)
- Az ütemező algoritmus ezeken a struktúrákon dolgozva végzi el a feladatát (elsősorban a futásra kész soron)

b. Ütemezés időskálái

Rövid távú (short-term) v. CPU ütemezés

- A futásra kész sorból választ egy futó állapotba átmenő feladatot
- Minimum 10-20 ms-enként végrehajtásra kerül (időzítő megszakítás), de inkább gyakrabban

Hossz távú (long-term) ütemezés batch rendszerekben

- Sokkal több feladatunk van, mint amennyit hatékonyan párhuzamosan végre tudunk hajtani
- Percenként vagy ritkábban fut, ismernie kell a feladatot (az általa okozott terhelést)
- Többnyire maximum időzíteni lehet feladatokat (UNIX: cron)
- Pl. 3-4 nagy file másolása Windows alatt egy diszkről egy másikra
 - 1. lehetőség: Párhuzamos másolás
 - 2. lehetőség: Szekvenciális másolás
 - Kérdés a hallgatóknak: Melyik lesz gyorsabb és miért?
- Beágyazott rendszerekre nem jellemző!

Középtávú (medium-term) ütemezés

- Swapping
 - A rendszerben lévő feladatok memóriájának egyes részei kiírhatóak háttértárra
 - Több feladat fér el a fizikai memóriába (virtuálisan)
 - Egy feladatra több fizikai memória juthat (amikor fut)

○ Swapping

- A rendszerben lévő feladatok memóriájának egyes részei kiírhatóak háttértárra
- Több feladat fér el a fizikai memóriába (virtuálisan)
- Egy feladatra több fizikai memória juthat (amikor fut)

○ Mi van, ha a futó vagy futásra kész (ami várhatóan hamarosan futni fog) feladat egyes szükséges részei a háttértáron vannak?

- Nem tud futni, addig, amíg a szükséges memória részek vissza nem kerülnek a fizikai memóriába
- A középtávú ütemezés ezt irányítja...

○ Beágyazott rendszerekben előfordul, pl. Linux alkalmazása esetén

- Valós-idejűséggel összeegyeztethetetlen!
- Mit lehet ellene tenni? Majd beszélünk róla a memória kezelésnél...

c. Preemptív és nem preemptív ütemezés

Tervezési kérdés, hogy a futó feladattól elvehető-e a CPU.

○ Preemptív: A futó feladattól az OS elveheti a CPU-t

- Ez a tipikus a modern operációs rendszerekben
- Bizonyos kernel feladatokra nem feltétlenül, azok nem megszakíthatóak, ennek vannak következményei (pl. valós idejű működés nehezen biztosítható)

○ Nem preemptív vagy kooperatív

- Pl. Windows 3.x
- A futó feladatnak le kell mondania a CPU-ról vagy eseményre kell várnia, ahhoz hogy más feladat tudjon futni
- Az egyes alkalmazásoktól függ a teljes rendszer működése
 - Lényegében az alkalmazás programozójától, ami nem jó ötlet...
- Ha a futó feladat hibás (végtelen ciklus), akkor a teljes rendszer működésképtelenné válik
 - OS az ilyen OS egyáltalán? ☺

d. Ütemező kiválasztása

Nincs univerzális ütemező algoritmus

A feladatnak megfelelő ütemezőt kell kiválasztani

A legtöbb beágyazott operációs rendszer erre lehetőséget ad

- Az ütemező kiválasztható és testre szabható az operációs rendszer konfigurálása (fordítás előtt) során

Szinten minden operációs rendszerben van lehetőség az ütemező lecserélésére vagy kibővítésére is

e. Ütemezési algoritmusok

Feltételezzük az alábbiakat:

- A rendszerben egy végrehajtó egység van
 - Több processzor esetén a probléma sokkal összetettebb
- Egy időben egy feladat tud futni
- A futásra kész feladatokat valamilyen várakozási sorban tároljuk (task/job queue)
 - Ez speciális esetben lehet FIFO, de általában nem az...
- A feladataink CPU és I/O löketekből (burst) állnak
 - Ez tapasztalat, mérések is vannak
 - Pl. a CPU löket jellegzetesen kisebb 10 ms-nél a mérések szerint
 - Ezek között I/O löketek vannak, ekkor a feladat passzívan vár az I/O lezajlására

f. FIFO

A legegyszerűbb algoritmus:

- Feladat leíróra mutató referenciákat tároló FIFO (First Input First Output) sor az implementáció
 - put()-tal bekerül a folyamat a várakozó sorba
 - get()-tel levesszük a futásra kerülőt
 - FCFS (First Come First Serve)

Nem preemptív definíciószerűen

- I/O-ra várhat!

Az átlagos várakozási idő nagy lehet, és erősen függ a CPU és I/O löket (burst) nagyságoktól

- Ez egyben átlagosan nagy válaszidőt is jelent, on-line felhasználókat is kiszolgáló rendszerek számára nem alkalmas

Kis adminisztrációs overhead

Várakozási idő kicsit részletesebb vizsgálata:

Tiszta CPU használat:

- A később beérkező feladatoknak ki kell várniuk a korábban beérkező feladatokat
- Hosszú feladat sokáig feltartja az utána következőket

CPU és I/O löketek:

- Ha van olyan feladat, aminek hosszú a CPU lökete, akkor az fogja tartani a kis CPU lökettel rendelkező feladatokat
- A nagy CPU lökettel rendelkező még a sor végéről is hamar előre kerül, aztán feltartja a többi
- Convoy hatás...

g. Round Robin (RR)

Időosztásos rendszerek számára találták ki az egyszerű FIFO ütemezés problémáinak kijavítására valamikor a 60-as években

Kedvezőbb az interaktív (külvilággal kapcsolatban lévő) feladatok számára

- Jobb az átlagos válaszideje a FIFO-nál
- Adott időnként garantáltan vált, függetlenül a feladattól

Preemptív:

- Lényegében a FIFO kiegészítése egy egyszeri óra megszakítással
 - Quantum vagy time slice (időszlet)
 - 100-1 ms, tipikusan 10-20 ms
 - Elsősorban elfogadható interaktív feladatokra vonatkozó válaszidő biztosítására

A feladat futtatása előtt az órát elindítjuk, az egy adott idő (időszlet) lejárta után megszakítást kér (fut az OS)

- Ha a feladat befejeződik vagy I/O-t hajt végre az óra megszakítás előtt (fut az ütemező):
 - Újraütemezünk, és az órát újraindítjuk
- Ha a feladat nem fejeződik be:
 - A megszakítás beérkezik, és fut az ütemező
 - A feladat futásra kész állapotba kerül, újraütemezünk és az órát újraindítjuk

A gyakorlatban sok esetben az egyszerűség kedvéért periodikus óra megszakítást alkalmaznak

- Így egyszerűbb az algoritmus, de a matematikai analízis bonyolultabb, és a tulajdonságok is romolhatnak
- A periodikus óra IT használható a rendszeróra és SW időzítők kezelésére is

Tulajdonságai az időszlet nagyságától, CPU löketek eloszlásának statisztikai jellemzőitől, és a kontextus váltás időtartamának a viszonyától függ

Időszlet > átlagos CPU löket → Átmegy FIFO-ba

- Lényegében a CPU löket nagyságától függően az I/O műveletek eredményeképpen fut az ütemező

Időszlet $\approx$ átlagos CPU löket → Normális működés

- A feladatok időosztásos módon osztoznak a CPU-n

Ökölszabály: A CPU löketek 80%-a kisebb az időszletnél

CPU löket > Időszlet, ami viszont összemérhető a kontextus váltás időtartamával:

- Nagyobb adminisztratív terhelés
- Mivel 10-20 ms (1 ms) az időszlet, ami nagyságrendekkel nagyobb a kontextus váltás idejénél, ezért ez ma nem probléma (régén volt)

h. Prioritásos ütemezők

Prioritás = fontosság (0 a legkisebb v. a legnagyobb?)

A feladatokhoz prioritást rendelünk:

- Külső/Belső prioritás:
 - Külső prioritás: Operátor vagy a feladat maga állítja be
 - Belső prioritás: A rendszer adja
- Statikus/Dinamikus prioritás:
 - Statikus prioritás: Tervezési időben dől el, állandó a futás során
 - Dinamikus prioritás: Futási időben dől el, változik a rendszerben lezajló változások hatására
- A gyakorlatban lehet ezek kombinációja is...

Prioritás belső megadása mérhető mértékek alapján:

- Felhasználó adhat egy kiinduló prioritást is
- Az operációs rendszernek kell kiszámolnia, pl. az alábbiak alapján:
 - Időkorlátok (válaszidő, stb.),
 - Memória igények (méret, elérhetőség, stb.),
 - Használt erőforrások és azok száma,
 - CPU és I/O löket,
 - Futási idő, tervezett futási idő,
 - stb.

Egyszerű prioritásos ütemező:

- Egy prioritási szinten egy feladat
- Egyszerű beágyazott rendszerekben használják

Egy prioritási szinten tetszőleges számú feladat:

- Módosításokkal széles körben alkalmazásra kerül
 - UNIX, Windows, stb., szinte minden operációs rendszerben ilyen találunk
- Módosítások:
 - Egy szinten belül a futásra kész feladat kiválasztására használt algoritmust meg kell adni, tipikusan RR
 - Dinamikus, belső prioritás meghatározás

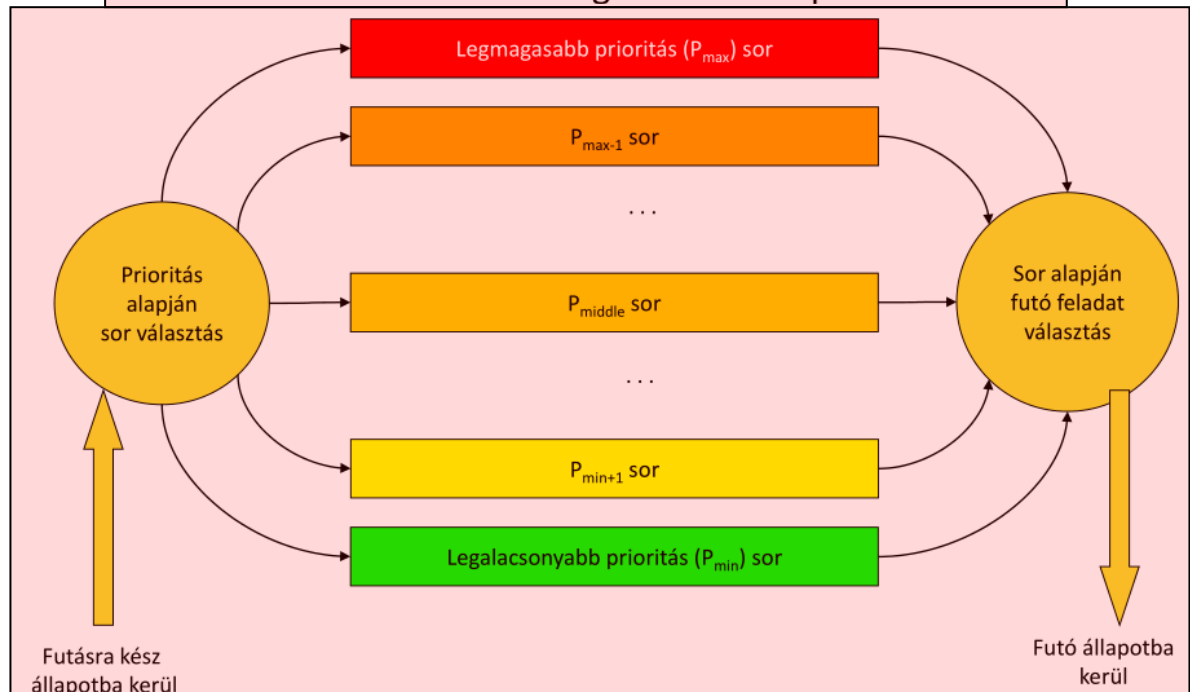
i. Többszintű prioritásos ütemezők

Minden egyes prioritási szinthez „futásra kész” várakozási sor

- A prioritás meghatározásáról nem mond semmit...
- Az egyes szinteken alkalmazott ütemezési algoritmusról nem mond semmit (FIFO, RR, stb.)
 - Implementáció függő

○ Hány prioritási szint szükséges?

- Többnyire néhány (8-16-32) elég
- Túl sok prioritási szint esetén átmehet az egyszerű prioritásos rendszerbe (szintenként 1 feladat)
- Mivel minden prioritáshoz kell egy sor is, a prioritási szintek számának növekedésével az algoritmus komplexitása nő



j. Prioritási szintek feladatokhoz rendelése (többszint esetben)

Többnyire a prioritás a feladat jellegéhez kapcsolódik. Pl.

- Batch feladatok alacsony prioritással
- On-line (interaktív) feladatok közepes prioritással
- Rendszer feladatok magas prioritással
- Valós idejű feladatok legmagasabb prioritással

Tipikusan RR (Időosztás) minden prioritási szinten

- Esetleg FIFO a batch feladatok esetén

k. Prioritásos rendszerek problémái

Alacsony prioritási szinten lévő feladatok éhezhetnek az egyszerű prioritásos rendszerben

- Ha a felsőbb prioritási szinten túl sok, és sokat futó feladat van.
- Mindenképpen kerülendő állapot (túlterhelés)

Időosztás prioritási szintek között (korrektség biztosítására, kiéheztetés elkerülésére)

- Pl. adott % CPU idő egy adott prioritási szinthez

I. Mikor fut az ütemező?

Külső esemény:

- Óraütés
- Más külső HW megszakítás

Belső esemény

- SW esemény bekövetkezése (pl. erőforrás felszabadul, lásd később)
- Futó részfeladat várakozó állapotba helyeződése (ki lesz a következő)
- Ha a futó részfeladat lemond a futás jogáról (yield)
- Kivétel bekövetkezése

A konkrét implementációkban persze más okok is lehetnek...

m. FreeRTOS ütemezője

Választható:

- Konfigurációs opciókkal (define-ok a FreeRTOSConfig.h file-ban)
 - Többszintű sorok időosztással
 - Prioritás + Round-Robin
 - Prioritásos, időosztás nélkül
 - Kooperatív ütemezés
- Idle task + Idle task „Hook”
 - Konfigurálható
 - Speciális követelmények az Idle task Hook függvénnyel kapcsolatban

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

a. Többprocesszoros ütemezés

Multiple-processor scheduling

Homogén többprocesszoros rendszerekkel foglalkozunk.

- Lehet SMP vagy NUMA architektúra a memória szempontjából (vagy azok keveréke).
- Többnyire egy adott I/O periféria egy adott processzorhoz van rendelve (arra csatlakozik).
- Két lehetséges megoldás:
 - Master and slaves (egy CPU osztja ki a feladatokat)
 - Self-scheduling / peering (minden CPU ütemez)

Az ütemezés heterogén többprocesszoros rendszerekben napjaink egyik legnagyobb szakmai kihívása...

- Sokféle ilyen rendszer van, nincs univerzális megoldás...

b. Processzor affinitás

A cache a futó feladat utasítás és adat tartalmának egy részét tartalmazza.

- A processzorok vagy processzor magok cache tartalma különböző lehet (architektúrától és a rajta futó feladatoktól függ).
- A cache koherencia csak a közösen tartalmazott adatokra vonatkozik, a teljes azonosságnak nem lenne értelme.
- A feladat más processzorra, vagy processzor magra kerülése csökkenti a végrehajtás sebességét (a lokális cache-ben nincs benne a kód és/vagy adat)

• Az új CPU cache tartalmat ismét fel kell építeni...

Cél: A feladatot ugyan azon a végrehajtó egységen tartani

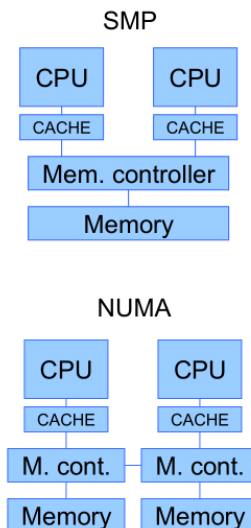
Laza vagy kemény processzor affinitás (soft or hard processor affinity).

- Laza: Nincs garancia, de törekszik rá az OS (többnyire alapeset)
- Kemény: Biztosan ugyan azon a CPU-n marad (rendszerhívással)

SMP esetén a processzor affinitás csak a cache találatok szempontjából érdekes.

NUMA esetén a feladat által használt fizikai memória is érdekes a processzor affinitás szempontjából

- A feladat elsősorban a CPU-hoz közeli memóriából fusson.
- A távoli (más CPU-hoz csatlakozó) memória használata minimalizálendő.
- Összetettebb ütemezési algoritmust igényel.



c. Terhelés megosztás

Egy globális futásra kész sor vagy processzoronkénti futásra kész sor?

Processzoronkénti futásra kész sor

- Push and/or Pull
- Push: OS kernel folyamat mozgatja a sorok között a feladatokat.
- Pull: Az idle állapotban (idle feladatot végrehajtó) CPU próbál a többi sorából feladatot kapni.
- A kettő kombinációja is használható.

Összefüggő, párhuzamosan futtatható feladatok optimalizálása (később szálak/thread)

- pl. Gang scheduler (gang = összetartozó párhuzamos feladatok)
- Nagyobb CPU számig lineáris skálázódás érhető el vele.
- Elsősorban erősen párhuzamos feladatok esetén szükséges.

d. Linux ütemezője

Kernel verzió és konfigurációs opció függő

- Részben fordítási, részben kernel boot parameterrel is konfigurálható

Egy és több processzoros heterogén architektúrájú rendszerekre optimalizált

- Azonos ISA-t feltételez a magokra, órajelben, kisebb architektúráis részletben stb. lehet eltérés
- Pl. ARM-Ax multicore Big-Little architektúrával

Ütemezési osztályok (osztály prioritás növekvő sorrendjében felsorolva)

- Nagyon leegyszerűsítve...
- SCHED_NORMAL:
 - Minden feladat, ami nem speciálisan lett egy más osztályba felsorolva
 - 2.6.23-tól CFS szerint ütemezve
- SCHED_RR: Valós-idejű RR ütemező
- SCHED_FIFO: Valós-idejű FIFO ütemező
- SCHED_DEADLINE: Valós-idejű Earliest deadline first ütemező (3.14 kerneltől)

2.6.23-tól az u.n. Completely Fair Scheduler (weighted fair queuing)

- Bináris fában (red-black tree) tárolt sched_entity struktúrák
- Folyamatosan fejlesztik, javítják, optimalizálják
- Ütemezési csoportok (scheduling groups)

RT-Preempt Patch

- Hard real-time kernel, meg kell patch-elni, majd konfigurálni, majd lefordítani

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

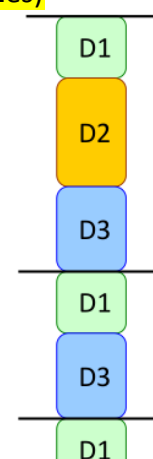
a. Feladatok leírása

Szoftver architektúrák a párhuzamos részfeladatok leírására

- Alacsony szintű vezérlési szerkezet példák
 - Round-robin, Round-robin megszakításokkal
 - Függvény sor
- Coroutine, Fiber
- Process/Folyamat
- Thread/Szál

b. Round Robin architektúra (körforgó programszervezés)

```
void main(void) {  
    while(1) {  
        if ( !! Device 1 needs service ) {  
            !! Handle Device 1 and its data  
        }  
        if ( !! Device 2 needs service ) {  
            !! Handle Device 2 and its data  
        }  
        if ( !! Device 3 needs service ) {  
            !! Handle Device 3 and its data  
        }  
    }  
}
```



c. RR tulajdonságai

A legegyszerűbb SW architektúra

- Nincs megszakítás, nincs közös erőforrás kezelésével kapcsolatos probléma

Nagy a szórása a válaszidőnek

- Függ az egyes részfeladatok tényleges végrehajtási idejétől az egyes ciklusokban, ami adatfüggő

Worst-case válaszidő:

- A részfeladatok válaszidejeinek összege (azaz lineárisan nő a taszkok számával)

Egyes feladatok tekintetében javítható a válaszidő, ha többször hajtjuk végre azokat a főhurokban

Az architektúra „törékeny”

- Új taszk hozzáadása felborítja az ütemezést és az időzítéseket

Hibalehetőség nagy, pl. a részfeladatok leírását a forrásnyelv elemei szeparálják

A CPU folyamatosan fut, nem tud aludni, a fogyasztás nagy

d. RR megszakítással

```
BOOL Device1_flag = FALSE;
BOOL Device2_flag = FALSE;
BOOL Device3_flag = FALSE;

void interrupt Device1_ISR (void) {
    !! Handle Device 1 time critical part
    Device1_flag = TRUE;
}

void interrupt Device2_ISR (void) {
    !! Handle Device 2 time critical part
    Device2_flag = TRUE;
}

void interrupt Device3_ISR (void) {
    !! Handle Device 3 time critical part
    Device3_flag = TRUE;
}
```

```
void main (void)
{
    while(1)
    {
        if ( Device1_flag ) {
            Device1_flag = FALSE;
            !! Handle Device 1 and its data
        }

        if (Device2_flag ) {
            Device2_flag = FALSE;
            !! Handle Device 2 and its data
        }

        if (Device3_flag ) {
            Device3_flag = FALSE;
            !! Handle Device 3 and its data
        }
    }
    ...
}
```

e. RR + megszakítás tulajdonságai

Jobban kezeli az időkritikus részeket

- A megszakítások magasabb effektív prioritáson futnak

Ha az alkalmazott MCU/CPU a megszakításokhoz prioritást tud rendelni, még tovább lehet finomítani az architektúrát

A kölcsönös kizárást meg kell oldani (IT és főhurok között)

- Kölcsönös kizárásról később beszélünk

A válaszidőnek szórása nagy

- Worst-case válaszidő: az összes feladat válaszideje + a megszakítások (arányos a taszkok számával)
- Egyes feladatok tekintetében javítható a válaszidő, ha többször kérdezzük le őket a főhurokban

Az architektúra „törékeny”

- Új taszk hozzáadása felborítja az ütemezést és az időzítéseket
- Hibalehetőség nagy, pl. a részfeladatok leírását a forrásnyelv elemei szeparálják

Lehetséges a fogyasztást csökkenteni a CPU altatásával

- A CPU IT-re fel tud ébredni...

f. Függvénysor alapú ütemezés

```
!! Define queue of function pointers

void interrupt Device1 (void)
{
    !! Handle Device 1 time critical part
    !! Put Device1_func to call queue
}

void interrupt Device2 (void)
{
    !! Handle Device 2 time critical part
    !! Put Device2_func to call queue
}

void interrupt Device3 (void)
{
    !! Handle Device 3 time critical part
    !! Put Device3_func to call queue
}

void main (void)
{
    while(1)
    {
        while(!Function queue not empty)
            !! Call first from queue
    }
}

void Device1_func (void)
{ !! Handle Device 1 }

void Device2_func (void)
{ !! Handle Device 2 }

void Device3_func (void)
{ !! Handle Device 3 }
```

Képes a prioritások kezelésére

- Mind részfeladat szinten (hogyan valósítjuk meg a függvénysort)
- Mind IT szinten (milyen az MCU/CPU megszakítás vezérlője)

A kölcsönös kizárást meg kell oldani (IT és főhurok és az abban futó függvények között)

Worst-case válaszidő (a legmagasabb prioritású feladatra) = a leghosszabb taszk válaszideje + IT

- A worst-case válaszidő nem nő lineárisan a taszkok számával
- A válaszidő szórása kisebb, mint a korábbi megoldásoké
- A legnagyobb futási idejű feladat futási idejével felülről becsülhető

Egy új feladat nem borítja fel az eddigi időzítést

Nem preemptív

Lehetséges a fogyasztást csökkenteni a CPU altatásával

- A CPU IT-re fel tud ébredni...

g. Coroutine és fiber

Kooperatív multitasking

- Folyamaton vagy szálon belül
 - OS támogatással vagy programnyelvi megvalósítás
 - Az OS szinten a coroutine-eket vagy fiber-eket tartalmazó folyamat vagy szál kerül ütemezésre
 - Az ütemezési algoritmus a programozó kezében van, neki kell implementálnia (kooperatív ütemezés)
- **Coroutine: programnyelvi szintű elem**
 - Haskell, JavaScript, Modula-2, Perl, Python, Ruby, etc.
- **Fiber: rendszerszintű eszköz**
 - Win32 API (ConvertThreadToFiber and CreateFiber).

Subroutine általánosítása

- Subroutine:
 - LIFO (Last In/called, First Out/returns)
 - Egy belépési pont, és több kilépési pont (return/exit)
 - A vermet használja a paraméterek és a visszatérési érték átadására
- Coroutine:
 - Az első belépési pont azonos a subroutine-nal
 - Utána viszont a legutolsó kilépési pontra tér vissza!
 - Átlépés a „yield to Coroutine_id” utasítással lehet
 - **Nem használhat vermet**, hiszen az megtelne (ide-oda lépked, igazából nem tér vissza)

h. Coroutine és fiber értékelése

Kooperatív multitasking-gal megoldható problémák esetén

Verem alapú környezetben (pl. C/C++) nehéz az implementációja a coroutine-nak (fiber rendszerszintű)

Egyes kooperatív együttműködést igénylő feladatok jól megvalósíthatóak vele

Nem szükséges osztozni az erőforrásokon (kooperatív):

- Nem kellenek OS hívások
- Kisebber erőforrás igény

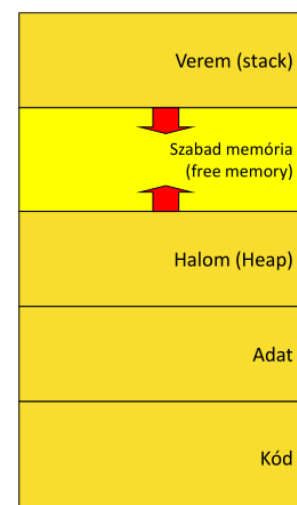
Az OS ütemezi egy szálban/folyamatban őket, vagyis egyetlen végrehajtó egységet tudnak csak kihasználni

i. Task fogalom megvalósítása

Feladat fogalom eredetileg folyamat (process) értelemben került használatra

A folyamat a végrehajtás alatt álló program

- Ugyanabból a programból több folyamat is létrehozható
- Saját kód, adat, halom (heap) és verem memória területtel rendelkezik
- Védettek a többi folyamattól
 - Szeparáció, virtuális gép, sandbox



j. Folyamatok

Virtuális CPU-n futnak:

- Nem férhetnek hozzá a többi folyamat és az operációs rendszer futása során előálló processzorállapothoz
- Kontextus váltás történik, ha más folyamat kerül futásra

Saját virtuális memóriaterületük van (később beszélünk róla)

- Nem férhetnek hozzá más folyamatok virtuális memóriájához vagy direkt módon a fizikai memóriához
- A processzor MMU-ja oldja ezt meg
 - Lehetséges azonos fizikai memóriaterületek megosztása például olvasási joggal (pl. kód terület)
 - A modern MMU-k ezt akár írási joggal is meg tudják tenni
 - Teljes MMU és OS kontroll alatt a biztonság érdekében

k. Folyamatok létrehozása

OS specifikus rendszerhívás:

- Windows: CreateProcess()
- UNIX/Linux: fork() + exec()
- FreeRTOS: Nem támogatja a folyamatokat!

Szülő/gyermek viszony a létrehozó és a létrehozott között.

- Process fa (process tree)
- A szülő erőforrásaihoz hozzáférés többnyire konfigurálható (mindenhez – semmihez)

- A szülő megvárhatja a gyermek terminálódását, vagy futtat vele párhuzamosan
- Paraméterezhető a gyermek (command line)

Sok adminisztráció, erőforrás-igényes

l. Folyamatok kommunikációja

A folyamatoknak együtt kell működniük (később lesz róla részletesen szó)

- Ehhez kommunikálniuk kell
- Külön előadások lesznek a témáról...

Két tetszőleges folyamat nem tud közös memórián keresztül kommunikálni

- Az MMU és a virtuális memória éppen ezt kívánja lehetetlenné tenni (szeparáció)
- Csak OS rendszerhívásokon keresztül tudnak kommunikálni, ami erőforrás igényes

Hatékony a védelem/szeparáció szempontjából

Nem hatékony módja a párhuzamos, erősen összefüggő feladatok megoldásának

- Pl. akár GUI + számításigényes feladat (WORD „újraszedés”)

m. Folyamatok befejezése

OS specifikus rendszerhívás:

- Windows: TerminateProcess()
- UNIX/Linux: exit(), return

Nyitott, használatban lévő erőforrásokat le kell zárni

- Pl. nyitott file-ok, stb.

A szülő megkapja a visszatérési értéket, többnyire egy egész értékű változó (integer) formájában

Mi történik, ha a szülő folyamat befejeződik, de a gyermek nem?

- OS függő megvalósítás, tipikus megoldások:
 - Alapértelmezett szülő folyamat (pl. UNIX init folyamat)
 - A gyermek automatikus befejezése (cascading termination)

Sok adminisztráció, erőforrás-igényes

n. Folyamatok értékelése

Védelmi/szeparációs szempontból jó megoldás, de erőforrás-igényes:

- Folyamat létrehozása és megszüntetése
- Folyamatok közötti kommunikáció és erőforrás-megosztás

Megoldás: Szál (thread) bevezetése:

- A szál a CPU-használat alapértelmezett egysége, magában szekvenciális kód
- Saját virtuális CPU-ja van, és saját verem áll rendelkezésre
- A kód, adat, halom és egyéb erőforrások (pl. file) tekintetében osztozik azokkal a további szálakkal, amelyekkel **azonos folyamat kontextusában** fut

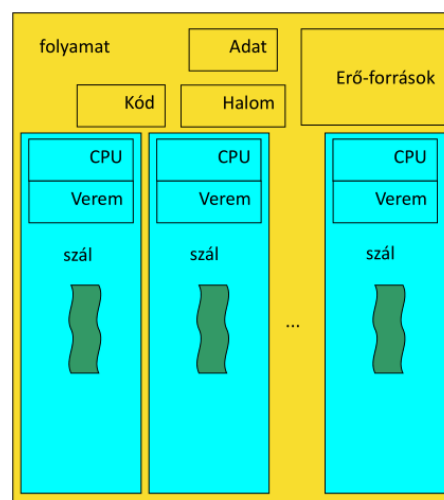
Folyamat = nehézsúlyú folyamat (heavyweight process)

Szál = pehelysúlyú folyamat (lightweight process)

o. Folyamat és szál eltérése



Egyszálú, tiszta folyamat alapú rendszer
(régi UNIX-ok)



Többszálú rendszer folyamat alapon
(modern UNIX és Windows)

p. Szálak támogatása

Jelenleg a modern operációs rendszerek natív módon támogatják szálak létrehozását

Windows:

- Program v. szolgáltatás = folyamat, folyamaton belül szálak
- Az ütemező szálakat ütemez

Linux:

- Program v. daemon = folyamat, folyamaton belül szálak
- Az ütemező task-okat ütemez, amik lehetnek folyamatok vagy szálak

Pl. Win32 API, Pthreads, JAVA thread

Win32: CreateThread() bonyolult paraméterezéssel

Pthreads: POSIX threads pl. Linux és más UNIX variánsok kernel vagy akár user szinten (csak viselkedést ad meg)

- Lesz gyakorlaton

FreeRTOS: xTaskCreate()

- Lesz gyakorlaton

q. Szálak alkalmazásának előnyei és következményei

Kis erőforrás igényű a létrehozásuk és megszüntetésük

- Kb. egy nagyságrenddel gyorsabb mérések szerint

Alkalmazáson belüli többszálúság támogatása

- A GUI válaszol, még ha háttérben csinál is valamit az alkalmazás (pl. számol)

Gyors kommunikáció közös memóriában az azonos folyamat kontextusában futó szálakra

- Csak a verem szál specifikus, a többi osztott
- A másik szál verméhez is hozzáférhet alapértelmezett esetben, amiből nagy problémák lehetnek

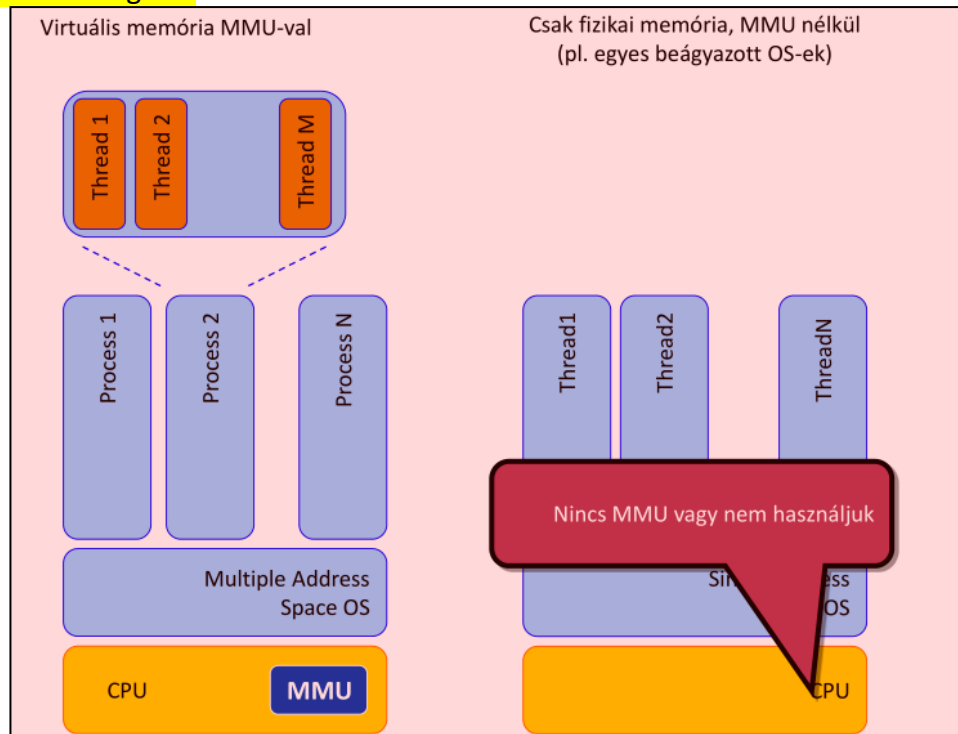
Skálázhatóság

- Alkalmazáson belül kihasználható több CPU

A közös memórián keresztüli kommunikáció veszélyes

- A kommunikációra használt memóriaterületek (struktúrák vagy objektumok) konzisztenciája sérülhet
- Később beszélünk erről a témáról (kölcsonös kizárásnak fogjuk majd a megoldást hívni)
- Eltérő folyamatok kontextusában futó szálak kommunikációja az OS-en keresztül történik
 - Erre ritkábban van szükség, mivel a szorosan összekapcsolódó (sokat kommunikáló) feladatok megvalósíthatóak egy folyamaton belül

r. HW támogatás



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

a. Folyamat létrehozása

OS specifikus rendszerhívás:

- Windows: `CreateProcess()`
- **UNIX/Linux: `fork()` + `exec()`**
- FreeRTOS: Nem támogatja a folyamatokat!

Szülő/gyermek viszony a létrehozó és a létrehozott között.

- Process fa (process tree)
- A szülő erőforrásaihoz hozzáférés többnyire konfigurálható (mindenhez – semmihez)
- A szülő megvárhatja a gyermek terminálódását, vagy futtat vele párhuzamosan
- Paraméterezhető a gyermek (command line)

Sok adminisztráció, erőforrás-igényes

- `/proc/...` file-struktúra a Linuxban (nézzük meg)

b. Szálak létrehozása

Pl. Win32 API, Pthreads, JAVA thread

Win32: `CreateThread()` bonyolult paraméterezéssel

Pthreads: POSIX threads pl. Linux és más UNIX variánsok kernel vagy akár user szinten (csak viselkedést ad meg)

- Lesz gyakorlaton

FreeRTOS: `xTaskCreate()`

- Lesz gyakorlaton

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

a. Feladatok együttműködése: felmerülő kérdések

- Erőforrások használata, közös erőforrások?
- Feladatok kommunikációja?
- Feladatok szinkronizációja?
- Architektúra függő kérdések?
 - Mit és hogyan használunk a feladatok megoldására?
 - Mik a következmények?

b. Bernstein tétele: Mikor hajtható végre két program párhuzamosan

Bernstein feltétele:

- P_i és P_j két darabja egy programnak
- P_i összes bemeneti változója I_i , és az összes kimeneti változója O_i , ugyan ez P_j -re I_j és O_j
- A két program párhuzamosan végrehajtható (vagyis független):

$$I_j \cap O_i = 0$$

$$I_i \cap O_j = 0$$

$$O_i \cap O_j = 0$$

c. Együttműködés lehetőségei:

Közös memórián keresztül (RAM v. PRAM modell):

- Szálak esetén (közös memória)

Üzenetekkel:

- Részletesen beszélünk róla később

d. RAM modell

Klasszikus Random Access Memory (egy végrehajtó egység)

RAM-modell szerint működik, azaz:

- Tárolórekeszekből áll,
- Egy dimenzióban, rekeszenként címezhető, csak rekeszenként, írás és olvasás műveletekkel érhető el,
- Az írás a teljes rekesztartalmat felülírja az előző tartalomtól független új értékkel,
- Az olvasás nem változtatja meg a rekesz tartalmát, tehát tetszőleges számú, egymást követő olvasás az olvasásokat megelőzően utoljára beírt értéket adja vissza.

A memória 1 valószínűséggel megbízható (kitekintés...)

- Ez egy alapvető feltételezés a programjainkban
- De nem igaz...
 - FLASH: Lassan felejt (pl. hőmérséklettől is függ)
 - RAM és I/O regiszterek: Kozmikus sugárzás, HW hibák, rosszindulatú programok, stb.
- Megoldás
 - HW megoldás (ECC, stb.)
 - SW megoldás: Különböző programozási technikák

e. PRAM modell

Parallel/Pipelined Random Access Memory

- sok végrehajtó egység közös memóriában (1CPU + DMA, SMP v. NUMA)

Több végrehajtó egység írhatja és olvashatja párhuzamosan

Változások a RAM modellhez képest:

- Az olvasás-olvasás ütközésekor mindkét olvasás ugyanazt az eredményt adja, és ez megegyezik a rekesz tartalmával,
- Az olvasás-írás ütközésekor a rekesz tartalma felülíródik a beírni szándékozott adattal, az olvasás eredménye vagy a rekesz régi, vagy az új tartalma lesz (versenyhelyzet), más érték nem lehet,
- Az írás-írás ütközésekor valamelyik művelet hatása érvényesül, a két beírni szándékozott érték valamelyike írja felül a rekesz tartalmát (versenyhelyzet), harmadik érték nem alakulhat ki.

A gyakorlatban ezt használjuk...

f. Erőforrások, közös erőforrás

Erőforrás (resource)

- Minden olyan eszköz, amire a párhuzamos programnak futása közben szüksége van
- A legfontosabb erőforrás a végrehajtó egység

- Memória és annak tartalma (tárolt adatstruktúrák)
- Perifériák
- Stb.

Közös erőforrás (shared resource)

- Egy időintervallumban több, párhuzamosan futó feladatnak lehet rá szüksége
- Az erőforráson osztoznak a feladatok
- Többnyire egy időben egy vagy maximum megadott számú feladat tudja helyesen használni (írás és olvasás)

- Egy felhasználó: Printer, Asszinkron soros port (UART), összetett adattípusokból létrehozott változók (string, tömb, struktúra, objektum)
- Több párhuzamos felhasználó: SCSI vagy SATA NCQ HDD (N parancs optimalizált párhuzamos végrehajtására képesek)

- **A rendszertervezőnek és programozónak a legfontosabb feladata, hogy felismerje a közös erőforrásokat, és biztosítsa azok helyes használatát**
- Az OS szolgáltatásokat nyújt a probléma megoldására, de a megoldás a programozó kezében van!

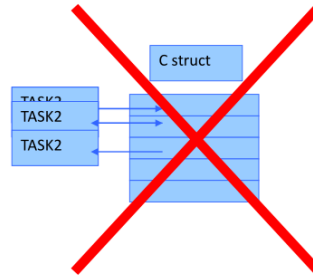
g. Közös erőforrás – probléma

Összetett adattípus:

- C struktúra vagy tömb típusú változó

Közös erőforrás, ha több részfeladat használja:

- Task1 írja
- Task2 olvassa
- Task1 fut először
- Task2 futtatására vált a rendszer a írás közben (pl. preemptív a rendszer)
- Inkonzisztens állapotban olvassa ki a változót Task2



- **Súlyos hiba!**

h. Megoldások – kölcsönös kizárás és kritikus szakasz

Kölcsönös kizárás (mutual exclusion)

- Annak biztosítása, hogy a közös erőforrást egy időben csak annyi magában szekvenciális feladat használja, amely mellett a helyes működése garantálható
- A kölcsönös kizárást meg kell oldanunk a programban
- Többnyire a használt erőforrást lock-oljuk (elzárjuk)
 - Nem engedjük hozzáférni a többi részfeladatot
 - A kérdés az, hogy azt hogyan tudjuk megoldani, és milyen részletességgel kell megoldanunk azt

Kritikus szakasz (critical section)

- A magában szekvenciális feladatok azon kódrészletei, amely során a kölcsönös kizárást egy bizonyos közös erőforrásra biztosítjuk
- A kritikus szakasz a kérdéses közös erőforráshoz tartozik
- A kritikus szakaszt a hozzá tartozó erőforrásra atomi műveletként (nem megszakítható módon) kell végrehajtanunk

i. Atomi műveletek

Atomi művelet (atomic operation)

- Nem megszakítható művelet, amelyet a processzor egyetlen utasításként hajt végre
- Egyprocesszoros (DMA nélkül) rendszerben bármilyen művelet sor atomivá tehető a művelet sor elején az IT teljes tiltásával, majd a művelet sor végén annak engedélyezésével
- TAS, RMW, speciális CPU utasítások az IT tiltás/engedélyezés elkerülésére
 - Test and Set, Read-Modify-Write, stb.
 - Elemi adattípusra (8/16/32/64 bit)
 - A modern processzoroknak vannak ilyen utasításai
- A közös erőforrások lock-olást, a kritikus szakasz megvalósítását atomi műveletekre vezetjük vissza

j. Közös erőforrások védelme

Mik férhetnek hozzá a közös erőforrásokhoz?

- ISR (Interrupt service routine)
- Feladat (folyamat vagy szál)
- DMA

Lehetőségek:

- IT tiltása és engedélyezése (speciális esetben lehetséges)
- Ütemező tiltása és engedélyezése (speciális esetben lehetséges)
- Locking (erőforrás specifikus lefoglalás majd feloldás)

Más, kutatási fázisban lévő megoldások:

- Software/hardware transactional memory (STM/HTM)
- Software-isolated processes (MS Singularity)

A lock megoldás sokak szerint nem jó, de jobb mint bármi eddig használt megoldás, vagyis ez az elfogadott megoldás

k. HTM

A HTM azért perspektivikus...

Első kísérlet az implementációra egy nagyobb gyártótól

- SUN Rock processor
 - Új utasítások: chkpt, commit
 - Új státusz regiszter: cps
 - A CPU végül nem készült el

IBM BlueGene/Q szuperszámítógép

Intel Haswell (2013 környékén megjelent, és hibás)

- Transactional Synchronization Extensions (TSX-NI)
- 2 működési mód
 - Az egyik visszafelé kompatibilis, a másik már nem
 - A core iX-6xxx és újabb CPU-kban javítva
 - Sajnos nem teljes a támogatás (olcsóbb CPU-k nem tudják)

//todo// eltérések a kölcsönös kizárástól

l. Újrahívhatóság

Thread-safe néven is emlegetik...

A közös erőforrás problémájának egyfajta kiterjesztett esete egy függvényen/objektumon belül is felléphet, amennyiben ezt a függvényt (metódust) egyszerre többen is meghívhatják

Hogyan fordulhat ez elő?

- Ugyanazt a függvényt hívjuk egy taszkból is és egy megszakítás rutinból is
- Az ütemezés preemptív, és ugyanazt a függvényt hívjuk két taszkból is

Az újrahívhatóság feltételei:

- Hosszú lista...
- Azt kell vizsgálni, hogy az újrahívott függvény/metódusban használt változók, perifériák, függvények/metódusok, stb. közös erőforrásnak minősülnek, és ha azok, akkor azokat közös erőforrásként megfelelően kezeli-e a függvény?

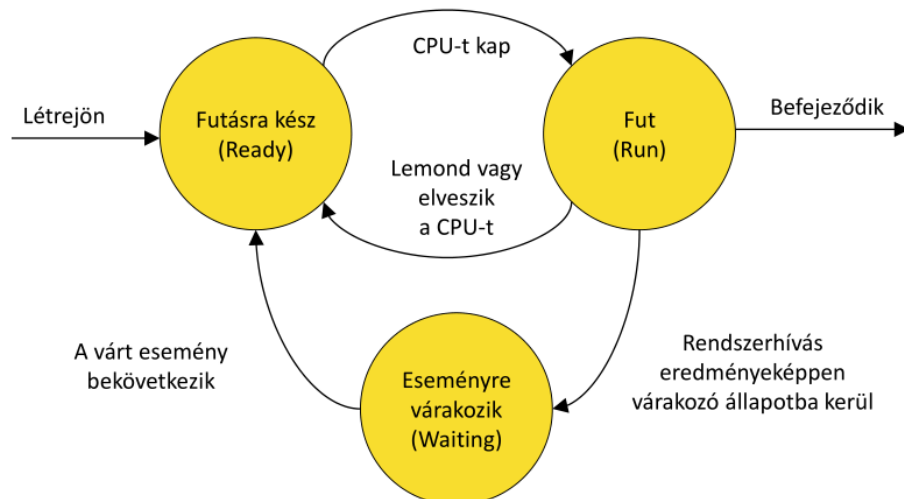
Példák:

- PC BIOS hívások nem újrahívhatóak!
- Preemptív operációs rendszer rendszerhívásai mindig újrahívhatóak, de az API függvényeinek lehetnek újrahívható (thread safe) és nem újrahívható (gyorsabb a kölcsönös kizárás megvalósításának elmaradása miatt) verziói is
- Egyéb programkönyvtárak? Mi a helyzet a kedvenc JPEG kezelő könyvtáraddal?

10. **Tétel:** 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.

a. Várakozás közös erőforrásokra

A más feladat által használt közös erőforrásra is „eseményre várakozik” állapotban várakozik a feladat
Az OS nyújt a közös erőforrások védelmére szolgáltatásokat



A közös erőforrást használni kívánó feladat megpróbálja megszerezni az erőforrást egy OS hívással

- Az erőforrás foglalt (használatban van):
 - A feladat az adott erőforrásra váró feladatok várakozási sorába (tipikusan FIFO), „eseményre vár” állapotba kerül
 - Fut az ütemező a következő futó feladat kiválasztására
- Ha egy másik feladat később felszabadítja az erőforrást:
 - Az erőforrás felszabadítása során az erőforrásra váró feladatok közül kiválasztásra kerül az erőforrást megkapó feladat (tipikusan FIFO az ütemezés)
 - Annak számára lefoglalja az erőforrást
 - Majd „futásra kész” állapotba helyezi azt
 - Fut az ütemező, ami új futó feladatot talál
- Az erőforrás szabad (nem használt):
 - Az erőforrást az OS lefoglalja a feladat számára
 - Visszatér a feladathoz, vagyis az „fut” vagy „futásra kész” állapotba kerül (az OS-től függ), majd CPU megkapása után a hívásból visszatérve fut tovább
- Használja az erőforrást
- A használat végén felszabadítja azt egy OS hívással

b. Kölcsönös kizárás probléma

A lock-olás részletessége (Fine or course grained locking)

- A kölcsönös kizárás megvalósítása erőforrás használattal jár (CPU), minimalizálni kell a használatát
 - Túl sok rendszerhívás jelentős overhead-del jár
- Viszont a túl nagy egységekben végzett kölcsönös kizárás is erőforrás pazarlással jár
 - A rendszerben „nehezebb” futásra kész feladatot találni
- Pl. Periféria egy buszon (félvezető hőmérő I²C buszon)
 - Működés: mérés indítás, 200ms mérési idő, mérési eredmény kiolvasása
 - A teljes mérés idejére megvalósított kölcsönös kizárás a buszra: Hosszú ideig nem érhető el a busz más célra sem
 - A mérés indításra és a mérési eredmény kiolvasására külön-külön valósítsuk meg a kölcsönös kizárást a buszon: Sok OS hívás, mivel a kölcsönös kizárás OS hívást jelent

c. Lock feloldására várakozás passzívan

Sleeplock, blocking call, stb.

- Ütemező által karbantartott várakozási sorok (beszéltünk róla)
 - Ha az erőforrás nem lock-olt
 - Megkapja az erőforrást a feladat lezárva és fut tovább
 - Ha az erőforrás lock-olt
 - A feladat megy az erőforráshoz tartozó várakozási sorba, a futásra kész feladatok közül egy futó állapotba kerül
 - Ha az erőforrás felszabadul, akkor az erőforráshoz tartozó sor elején álló megkapja az erőforrást lezárva, és futásra kész állapotba kerül
- Erőforrás-takarékos, de van overhead-je
 - OS fut, ütemezés és kontextus váltás
- Utána csak futásra kész sorba kerül a részfeladat, pontos időzítés nehezen megoldható (a futás kezdete érdekes). Alsó korlátot ad
- A processzor aludhat, ha nincs feladat:
 - Aztán HW IT-re felébred (belső esemény nem történhet)

d. Lock feloldására várakozás aktívan

Livelock, spinlock, busy wait, spinning:

- Aktív várakozás az erőforrás felszabadulására és megszerzésére (CPU erőforrás pazarlás)
 - Ha aktívan vár egy részfeladat, akkor a többi részfeladat hogyan tudja „előidézni” a várakozást megszüntető változást (eseményt) a rendszerben?
 - Nem tudnak futni, az aktívan várakozó fut!
 - Külső HW megszakítás és/vagy több CPU esetén ez nem probléma

- Fogyasztás is nő, hiszen a CPU folyamatosan fut, nem tud aludni (ha nincs éppen feladat)

- Compiler optimalizáció kiszedheti/eltávolíthatja a kódot

- Függ a CPU sebességétől (ha adott időt akarunk várni):
 - A kód hordozhatósága rossz. Az ütemező befolyásolja, alsó korlát csak
 - Mérjük meg : Linux Bogomips : "bogus" MIPS
 - Mi van, ha a CPU változtatja az órajelét?

e. Akkor hogyan is várakozzunk?

Rövididejű várakozáshoz a spinlock elkerülhetetlen:

- **Garantáltan rövid idejű** kölcsönös kizárási problémák kezelésére
- Periféria kezelés során, kb. $n \cdot 1\mu s$ vagy az alatti időtartamú időzítésre

HW Timer is használható adott időtartamú várakozásra, bár többnyire limitált számú Timer periféria van a MCU-ban

- IT overhead megjelenik
- Az IT késleltetésnél csak legalább egy nagyságrenddel nagyobb várakozás indokolható az overhead miatt
- Ez kombinálható az aktív várakozással („kevésbé aktív várakozás”)

Nehéz jó kompromisszumot találni

- Sleeplock (ütemező) – Dedikált Timer IT – spinlock ?
- A pontos határok sok szemponttól függenek

Az ütemező nem használ a feladatok ütemezése során spinlock jellegű hívásokat (egyértelműen sleeplock alapú)

Maga az OS kernel viszont gyakran használhat (Pl. SMP Linux)

11. **Tétel** 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.

a. Eszközök RAM/PRAM modell esetén

Kölcsönös kizárás megoldására:

- Lock bit,
- Szemafor,
- Kritikus szakasz objektum,
- Mutex,
- Stb. (minden OS-ben megvan a megoldás).

Minden OS-ban hasonló eszközöket fogunk találni

- Persze kicsit más lesz a nevük és működésük, lesznek különböző verziók, stb., a dokumentáció olvasása nem kerülhető el...
- Tervezési/architektúráis minta

Tipikus hibák kivédésére és a kölcsönös kizárási probléma megoldására:

- Monitor

b. Lock bit

Legegyszerűbb forma.

- A védendő erőforráshoz tartozik egy logikai változó (Boolean)

Lock bit jelentése:

- Lock bit FALSE $\Rightarrow$ nem használt az erőforrás
- Lock bit TRUE $\Rightarrow$ használt az erőforrás

Belépés művelet:

- Tesztelés, ha
 - Lock bit == FALSE
 - Lock bit = TRUE (az erőforrás lock-olása)
 - Megyünk tovább (belépünk a kritikus szakaszba)
 - Lock bit == TRUE
 - Aktív várakozás, amíg nem lesz FALSE, és utána Lock-oljuk

Kilépés művelet:

- Lock bit = FALSE

A lock bit alkalmazása során egy súlyos hiba jelenhet meg:

- IT érkezik a lock bit tesztelése során:
 - A következő utasítás már nem fog lefutni (az ISR-re kerül a vezérlés), vagyis a kritikus szakaszba lépést nem jelezzük, pedig már abban vagyunk
 - Az IT-ben vagy az utána futó más részfeladatokban az erőforrás szabadnak látszik
 - A védett erőforrás inkonzisztens állapotba kerülhet

Megoldás:

- IT tiltás a teszt előtt, IT engedélyezés a beállítás után (egy CPU esetén)
- Test and Set (TAS), vagy hasonló atomikus gépi utasítás

c. Szemaforok

Bináris és counter típusú szemafor

- Bináris: egy feladat a kritikus szakaszban
- Counter típusú: több feladat a kritikus szakaszban, vagy N darabos erőforrás készletből M darab lefoglalása

OS hívás, a bináris szemafor egy magas szintű lock bit

- Implementációtól függően:
 - Aktívan várhat (ma már nem jellemző)
 - Várakozó állapotba helyezi a részfeladatot

E.W. Dijkstra (EWD 1036?) találta ki a 60-as évek közepén

Két művelet értelmezett rajta:

- Belépés: P(), Wait(), Pend(), ...
- Kilépés: V(), Signal(), Post(), ...
- Sokféle elnevezés, a P()-ről vita van, hogy mit is jelent, a V()) a holland kilépés szóból ered

A counter típusú esetén a belépés és kilépés lehet egy számmal paraméterezett (hány egység erőforrással kívánunk belépni vagy kilépni)

Ha 1-nél több erőforrásra van szükségünk, akkor azokat vagy egyben mind megkapjuk, vagy a töredékeket nem foglaljuk le (más feladatnak szüksége lehet rájuk)

- Ezért paraméterezhető ebben az esetben a belépés és a kilépés a szükséges erőforrások számával
- Az egyenként (pl. for ciklussal lefoglalva őket) N darab erőforrás lefoglalása könnyen versenyhelyezethez, vagy akár holtponthoz vezethet

d. Kritikus szakasz objektum és Mutex

Lényegében bináris szemafor szerűen működnek

Kritikus szakasz objektum

- Használata:
 - Létre kell hozni a CriticalSection objektumot
 - Enter() metódussal lépünk be a kritikus szakaszba
 - Blokkol (sleeplock), ha már valaki benne van a kritikus szakaszban
 - Leave() metódussal lépünk ki a kritikus szakaszból
 - Ha szükséges a CriticalSection objektum megszüntethető

Mutex: Mutual Exclusion rövidítése

- Használata:
 - Acquire()/WaitOne() függvény vagy metódus.
 - Release() függvény vagy metódus

e. Multiple read single write mutex

Egy időben tetszőleges számú szál olvashatja a memóriát, de csak egy írhatja. Különböztessük meg őket...

- Read és Write belépés és kilépés.

Alapműködés, Writer addig nem léphet be, amíg Reader van a mutex-en (ha Writer vár, Reader beléphet).

- Write starvation
- 1. Megoldás: Writer belépése után az utána belépő Reader-ek is várnak a Writerre
 - Prioritás inverzió kezelése (Prioritás öröklés) nagyon bonyolult és erőforrás igényes

Read-copy-update (RCU)

- A Writer egy saját másolatot kap, a korábbi Reader-ek a régi verziót használják, a később belépő Readerek meg megvárják a Writer-t és az új verziót használják.
- Linux kernelben támogatott

f. Részfeladatok közötti szinkronizáció

Részfeladatok közötti szinkronizáció visszavezetése kölcsönös kizárásra:

- Nincs védendő objektum igazából, részfeladatok együttműködése a cél

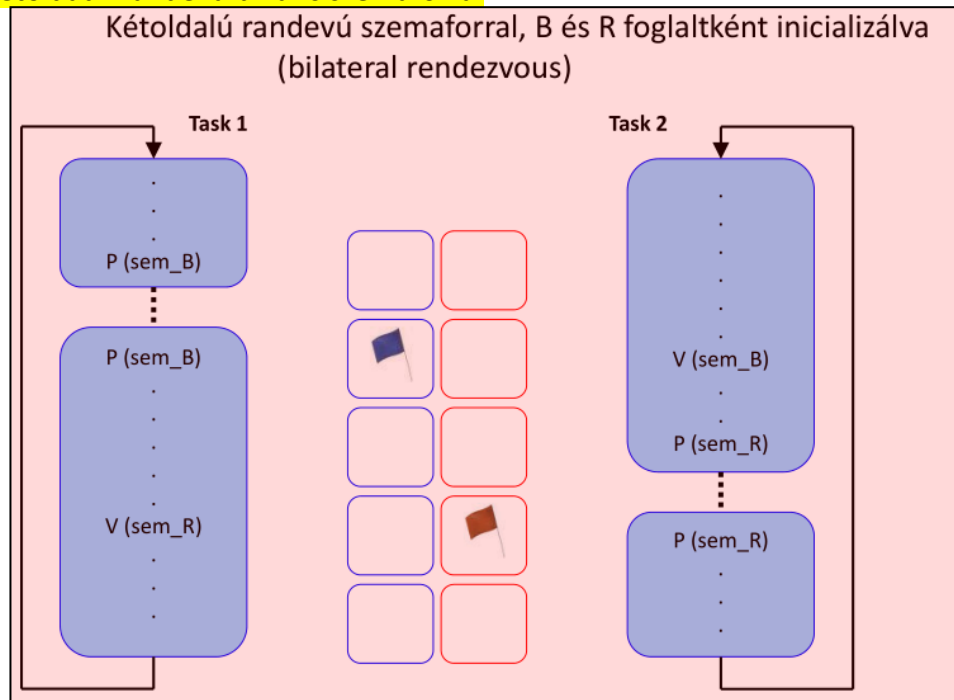
Pl. Rendezvény (rendezvous)

- Két vagy több feladat összehangolt végrehajtása
- Pl. a Coroutine-nál megvalósított termelő-fogyasztó probléma megoldható 2 bináris szemaforral is
- Ebben az esetben az operációs rendszer által nyújtott szolgáltatásokat használunk, és a feladatok passzívan várnak egymásra
- A szemaforok alapesetben foglaltként vannak inicializálva ebben az esetben

Memórián keresztül történő kommunikáció

- A kommunikációra használt memória közös erőforrás

g. Kétoldali rendezvény bináris szemaforral



h. Kommunikáció formája: védett memóriaterület

Szálak közötti kommunikáció során:

- Folyamatok így nem tudnak kommunikálni (MMU)
- Kivétel:
 - A UNIX SystemV vagy POSIX osztott memória (shared memory)
 - A memória kis rész
 - OS szolgáltatáson keresztül csatlakozható (védelem) futási időben
 - **Folyamatok közötti kommunikációt tesz lehetővé osztott memória sebességgel**
 - Később beszélünk róla...

Tömb, Struktúra, Objektum

Egy vagy kétirányú kommunikáció

- Egyirányú: A küldő írja, a vevő/vevők kiolvassák belőle
- Kétirányú: Minden fél írja és olvassa

A kölcsönös kizárást lock-bit, szemafor, mutex, kritikus szakasz objektum, stb. oldja meg

i. Blokkoló és nem blokkoló függvényhívás

Blokkoló hívás = szinkron

- Az eredmény és mellékhatások a visszatérés után jelentkeznek (megtörtént)
- Visszatérési érték kezelés egyszerű...
- Példa: Klasszikus függvényhívás

Nem blokkoló hívás = aszinkron

- Az eredmények és mellékhatások a visszatéréskor még nem jelentkeznek (nem történtek meg)
- Csak a végrehajtás kezdődik el a hívásra
- A visszatérési érték kezelése, és az eredmények és mellékhatások kezelése csak más értesítés után lehetséges:

- Pl. esemény, jelzés (signal), callback függvény, stb.

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

a. Tipikus hibák

Versenyhelyzet (race condition):

- A párhuzamos program lefutása során a közös erőforrás helytelen használata miatt a közös erőforrás vagy az azt használó program nem megfelelő (hibás) állapotba kerül
- Ilyen volt a korábbi összetett adattípusos példa
- A hibás állapot részletei erősen függenek azt használó szekvenciális részfeladatok lefutási sorrendjétől

Kiéheztetés (starvation):

- Ha a párhuzamos rendszer hibás működése miatt egy feladat soha nem jut hozzá a működéséhez szükséges erőforrásokhoz, akkor ki van éheztetve (nem tud futni)
- Nem csak a CPU-ra, de más közös erőforrásokra is felmerülhet
 - CPU-ra "Futásra kész" állapotban vár, az összes többi erőforrásra „Eseményre várakozik” állapotban vár

Holtpont:

- A közös erőforrások hibás beállítása vagy használata miatt a rendszerben a részfeladatok egymásra várnak
- Nincs futásra kész folyamat
- Nem jöhet létre belső esemény
- A rendszer nem tud előrelépni
- Példa: Két kiskecske a hídon...

Livelock:

- Példa: „Két kedves ember összetalálkozik az ajtóban”
- Többnyire a hibás holtpont feloldás eredménye
 - Kölcsönös kizárás + timeout ide vezet
- A rendszer folyamatosan dolgozik, de nem lép előre

b. Holtpont (Deadlock)

A holtpont a legsúlyosabb hiba, ami multiprogramozás során előfordulhat

Pontos definíció:

- Egy rendszer feladatainak egy H részhalmaza holtponton van, ha a H halmazba tartozó valamennyi feladat olyan eseményre vár, amelyet csak egy másik, H halmazbeli feladat tudna előállítani
- H részhalmazról van szó, többnyire csak néhány feladat érint

c. Holtpont: egyszerű példa

2 erőforrás, A és B

2 feladat akarja őket használni az alábbi módon

P1 folyamat:	P2 folyamat:
Lefoglal (A) ;	Lefoglal (B) ;
Lefoglal (B) ;	Lefoglal (A) ;
use A and B ;	use A and B ;
Felszabadít (A) ;	Felszabadít (B) ;
Felszabadít (B) ;	Felszabadít (A) ;

Versenyhelyzet alakul ki P1 és P2 között. Melyik fog előbb lefutni?

Ha azonos sorrendben foglaljuk le az erőforrásokat, akkor a probléma megoldható. Viszont a holtpont kezelés (és detektálás) a gyakorlatban nem ennyire egyszerű)

d. Holtpont a gyakorlatban

Az egyszerű példánál sokkal összetettebb esetek is előfordulhatnak

Nehéz felismeri...

- Versenyhelyzet formájában jelentkezik
 - Hol működik, hol előáll a holtpont...
- Bizonyos feltételek együttes teljesülése esetén áll elő a holtpont

A holtpont más feladatokat is befolyásolhat:

- A H részfeladatok által lefoglalt és soha fel nem szabadított (befagyott) erőforrásokon keresztül

e. Holtpont szükséges feltételei

1. Kölcsönös kizárás (Mutual Exclusion)

Vannak olyan erőforrások, amelyeket csak kizárólagosan lehet használni, és azokat több feladat használná

2. Foglalva várakozás (Hold and Wait)

Legyen olyan feladat, amelyik lefoglalva tart erőforrásokat, miközben más erőforrásokra várakozik

3. Nincs erőszakos erőforrás elvétel a rendszerben (No resource preemption)

A feladatok csak önszántukból szabadítják fel az erőforrásokat (kivéve az ütemezést, az lehet preemption)

4. Körkörös várakozás (Circular Wait)

A rendszerben lévő feladatok között létezik egy olyan $\{P_0, P_1, \dots, P_n\}$ sorozat, amelyben P_0 egy P_1 által lefoglalva tartott erőforrásra vár, P_i egy P_{i+1} -ra épül, és P_n pedig P_0 -ra ilyen szempontból

f. Holtpont kezelése, megelőzése

Tervezési idejű módszer

Olyan rendszert hozunk létre, ami holtpontmentes, vagyis a holtpont szükséges feltételeiből legalább egy nem teljesül

Nincs futási idejű erőforrás foglalás

- Drasztikus, de pl. beágyazott rendszerekben lehetséges

Foglalva várakozás kizárása:

- Az erőforrást birtokló feladat kér újabb erőforrást
- Minden szükséges erőforrást egyben kell lefoglalni, egyetlen rendszerhívással
- Alkalmazástól függ a használhatósága
- Erőforrás-kihasználás romlik

Erőszakos erőforrás elvétel lehetővé tétele:

- Erőforrás menthető állapottal
- Nem kell rollback a feladat szinten

Körkörös várakozás elkerülése:

- Fejlesztés során az erőforrás használat korlátozása
- Pl. teljes sorrendezés (total ordering) algoritmus
 - A programozónak be kell tartania
 - Esetleg a forráskód automatikus vizsgálatával javítható a helyzet
- Szoftvertechnológia, formális megközelítés nyújthat segítséget
 - MSc-ben kerül majd elő

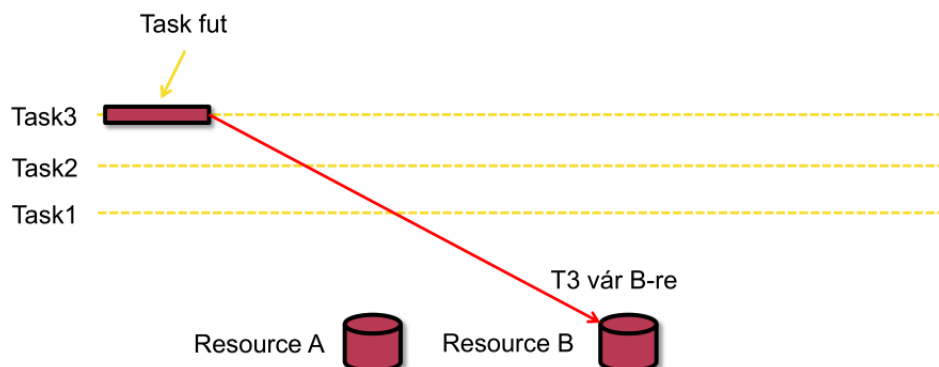
g. Még egy tipikus hiba: a prioritás inverzió

Prioritás inverzió (priority inversion):

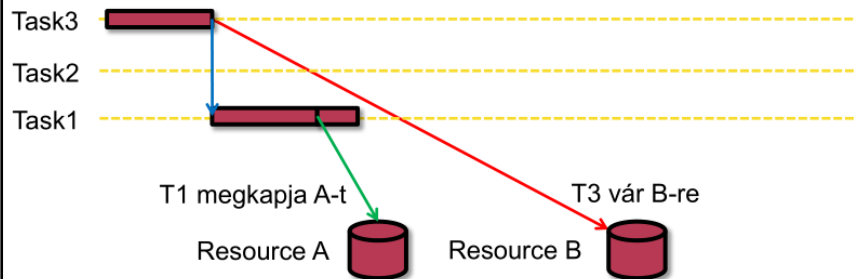
- Prioritásos ütemezőkben fordulhat elő, de az erőforrás használatával is összefügg.
- A legegyszerűbb esetének előfordulásához kell:
 - 3 feladat, különböző statikus prioritással,
 - Egy közös erőforrás, amelyet a 3 feladat közül a legmagasabb és a legalacsonyabb is használni kíván.
 - A közepes prioritású feladatnak CPU intenzívnek kell lennie.
- Kedvezőbb esetben csak a rendszer teljesítménye csökkent, a válaszidők nőnek. Valós idejű rendszer????
- Rosszabb esetben kiéheztetés, vagy akár holtpont is lehet a prioritás inverzió eredménye.

h. Prioritás inverzió: egyszerű példa

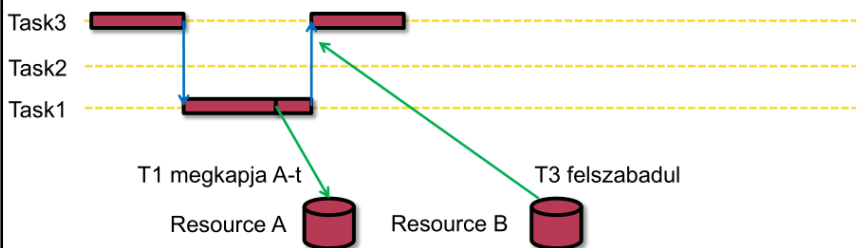
- Task3 magas prioritású feladat valamilyen eseményre vár (de nem A erőforrás felszabadulására). Pl. B erőforrásra...



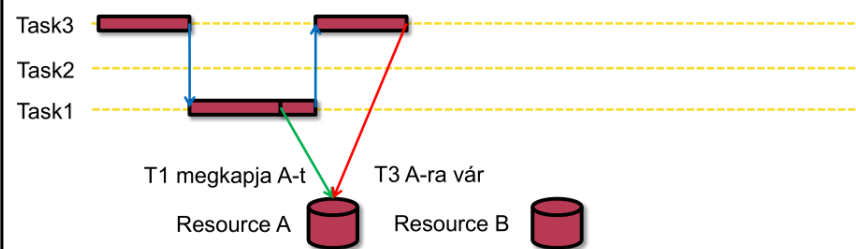
- Task1 alacsony prioritású feladat fut, és megszerzi az A erőforrást, és azt használva fut tovább.

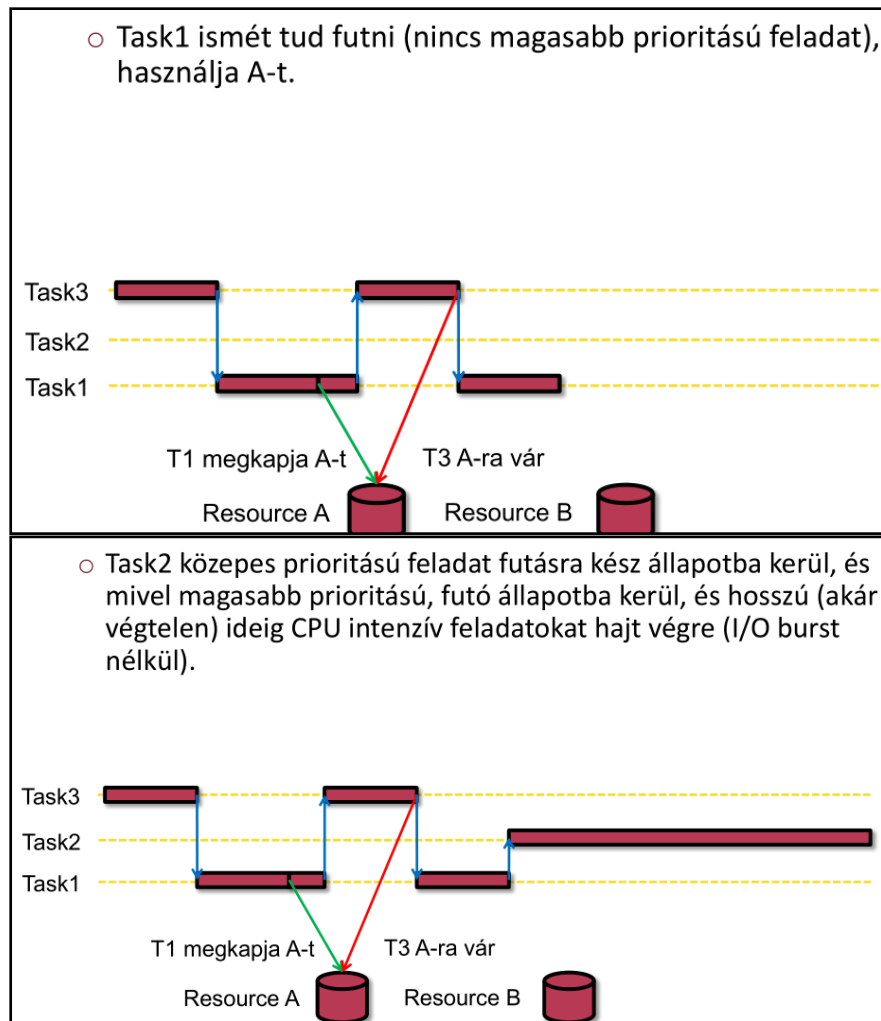


- Task3 magas prioritású feladat által várt esemény megérkezik (B felszabadul), futni kezd (preemptív ütemezés).



- Task3 fut, majd használni kívánja az A erőforrást. Mivel A foglalt, ezért várakozni kezd (lényegében T1-re vár).





Eredmény:

- Task3 nem tud továbblépni, hiszen eseményre vár (A erőforrás felszabadulására).
- Task1 nem tud továbblépni, hiszen CPU-ra vár.
- Task2 (TaskX) nem foglalkozik az A erőforrással, és amíg intenzíven használja/használják a CPU-t:
 - Task1 nem tudja befejezni a munkáját és felszabadítani az A erőforrást.
 - Amíg A erőforrás nem szabadul fel, Task3 nem tud futni, és végrehajtani a magas prioritású feladatát.

Task3 magas prioritású feladatot alacsonyabb prioritásúak nem hagynak futni... ☹️, 🙅, 💣, ☠️

i. Prioritás inverz: megoldások

Prioritás öröklés (Priority inheritance, PI):

- Az alacsony prioritású feladat megörökli az általa kölcsönös kizárással feltartott feladat prioritását a kritikus szakaszból való kilépéséig.
- Csak részben oldja meg a problémát.

Prioritás plafon (Priority ceiling, PC):

- Majdnem ugyan az, de az adott közös erőforrást használó legnagyobb prioritású feladat prioritását örökli meg (ami lehet nagyobb mint az éppen feltartott feladat).
- Az adott erőforrást máskor használó többi feladat sem tud futni (ha esetleg azok is CPU intenzívvé „válnak”).
- Az alacsony prioritású feladat akadályoztatás nélkül le tud futni.

Modern beágyazott OS-ekben választhatóak...

- None, PI, PC (ha preemptív ütemezőt választunk)

Megoldás más közelítésben:

- Nem speciális protokollokkal (PI, PC) kell megoldani, hanem jól meg kell tervezni a rendszert (prioritások kiosztása, kölcsönös kizárás, kölcsönös kizárás időtartama, stb.).

j. Lockolás során elkövetett tipikus hibák

Belépés/Kilépés elmaradása
Többszöri be- vagy kilépés
Más erőforrás lefoglalása
Az erőforrás indokolatlanul hosszan történő lezárása

- Minimális időre kell törekedni

Prioritás inverzió

Deadlock és livelock

k. Monitor: hibák elkerülése

Lokalizáljuk a lock-olással kapcsolatos feladatokat a közös erőforrást körülvevő API-val

- A lock-olás nem szétszórva történik a programban, hanem egyetlen, a közös erőforráshoz szorosan tartozó programrészletben
- A megvalósítás lehet automatikus, például nyelvi szinten (pl. JAVA, C#).
 - A compiler valósítja meg a kölcsönös kizárást biztosító konstrukciókat (pl. szemafor vagy mutex tényleges alkalmazásával)
- Kézzel is készíthetünk hasonló konstrukciót, pl. egy védett objektumot hozhatunk létre, amely a nyilvános metódusaiban elvégzi a lock-olást, és elrejtje a közös erőforrást

I. Monitor fejlődése: Hoare és Mesa szemantika

Hoare szemantika:

- Azonnal az erőforrást megszerző részfeladat fut
 - Erőforrás igényes és nehéz megvalósítani
 - Nem kompatibilis a preemptív ütemezőkkel

Mesa szemantika:

- A futásra kész részfeladatok közé kerül, és később az ütemező futtatja
- Vannak vele gondok, de azok megoldhatóak (most nem megyünk bele)
- "notifyAll" vagy "broadcast" üzenetek küldése is lehetséges
 - Minden adott eseményre váró futásra kész állapotba kerül

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

a. FreeRTOS locking szolgáltatások

Szálakat támogat (task-nak hívják) ezért a közös memória alapvető

Lehetőségek:

- Megszakítás tiltás és engedélyezés
- Kritikus szakasz (globális ebben az esetben)
- TaskNotify mechanizmus
- Semaphore
- Mutex

Interrupt tiltás és engedélyezés (makrók)

- taskDISABLE_INTERRUPTS()
- taskENABLE_INTERRUPTS()
- Konfigurációtól függ a viselkedése

Kritikus szakasz

Egyben tiltják a megszakításokat is!

Rövidnek kell lennie, felfüggeszti az ütemezőt, RT működés függ tőle

Belépés:

- task: taskENTER_CRITICAL()
- ISR: taskENTER_CRITICAL_FROM_ISR()
 - Visszatérési értéke a megszakítási maszk állapota
 - El kell menteni a hozzá tartozó kilépő híváshoz!

Kilépés:

- task: taskEXIT_CRITICAL()
- ISR: taskEXIT_CRITICAL_FROM_ISR()
 - Paraméterként kapja a megszakítási maszkot

Nem szabad benne semmire sem várni (pl. mutex, semafor, stb.) és taskYIELD() sem hívható

- Futó állapotban marad a hívó, és a kilépésig abban is kell maradnia...
- A kilépésen és újra belépésen kívül nem sok mindennek van értelme az API-ból!

b. Task Notify mechanizmus

Minden task-hoz hozzá van rendelve egy 32 bites notification érték (notification value)

A configUSE_TASK_NOTIFICATIONS konfigurációs paraméterrel tiltható/engedélyezhető

- Alapértelmezetten engedélyezve van
- 8 byte-ot foglal el (a notification value + belső adatstruktúrák)

Inicializáláskor ez 0 értéket kap...

Ezt lehet módosítani

- Opcionálisan a blokkolt task-ot futásra kész állapotba helyezheti (ha az vár a notification-ra)
- Nullázás, beállítás, inkrementálás, bitbeállítás, feltételes beállítás (ha van korábbi nem kezelt notification, akkor nem állítja be, és hibával visszatér)
- A notification „be nem fejezett” (pending) állapotban is lehet
- Opcionális szinkronizációs mechanizmusok állnak rendelkezésre

Sok funkció

- Kommunikáció a task-ok között
- Gyorsabb, kisebb erőforrású a mutex és a szemafor megvalósításnál

Task Notify szinkronizáció

Várakozás egyszerű notify-ra:

- xTaskNotifyWait()
- Van hozzá timeout is!

Bináris és a counter típusú szemafor működés valósítható meg vele

- Egyszerűbb API, mert a notification value itt automatikusan beállítja

c. FreeRTOS: szemafor létrehozása és kezelése

xSemaphoreCreateBinary()

- Heap-en csinálja meg a szükséges adatstruktúrákat
 - Csak egy handle-t kapunk vissza hozzá
- xSemaphoreCreateBinaryStatic() esetén a felhasználónak kell a memóriát rendelkezésre bocsátania
 - Erre is visszkapunk egy handle-t
- Lefoglalva kerül létrehozásra
 - Elsődleges célja a szinkronizáció

xSemaphoreCreateCounting()

- Paramétere: maximális érték, kezdőérték
- Heap-en dolgozik, a programozó által megadott memóriát használó párja az xSemaphoreCreateCountingStatic()
- Ha NULL-lal tér vissza, hiba volt, illik kezelni...
- Számlálásra és többpéldányos erőforrások kezelésére alkalmas

Szemafor kezelése

`uxSemaphoreGetCount()`

- Nem szabad kölcsönös kizárással kapcsolatban használni, felhasználói felület és debug célú (tájékoztató jellegű)

`xSemaphoreGetMutexHolder()`

- Ha a mutexet tartó hívja meg, akkor a visszatérési érték megbízható (önmaga tartja), de ha más task tartja, akkor nem biztos (nem feltétlenül konzisztens az állapot, hiszen a hívás és a visszatérés között bármi történhet)
- Ez is tájékoztató jellegű

`xSemaphoreGive()/xSemaphoreTake()`

- Semaphore/Mutex elengedése (give/release) vagy lefoglalása (take/obtain)
 - Timeout a Take-en
- ISR: `xSemaphoreGiveFromISR()/xSemaphoreTakeFromISR()`
 - ISR esetén érdekes dolgok történhetnek, amikor a futó task-nál nagyobb prioritású task-ot engedünk el
 - Nincs timeout a Take-en (miért is?)

`xSemaphoreGiveRecursive()/xSemaphoreTakeRecursive()`

- Rekurzív mutex esetén

d. FreeRTOS mutex létrehozása

`xSemaphoreCreateMutex()`

- Heap-en dolgozik, a programozó által megadott memóriát használó párja az `xSemaphoreCreateMutexStatic()`
- Ha NULL-lal tér vissza, hiba volt, illik kezelni...
- Kölcsönös kizárásra való (szabadon kerül inicializálásra)
 - Prioritás öröklést támogat

`xSemaphoreCreateRecursiveMutex()`

- `configUSE_RECURSIVE_MUTEXES` beállítása szükséges
- Heap-en dolgozik, a programozó által megadott memóriát használó párja az `xSemaphoreCreateRecursiveMutexStatic()`
- Ha NULL-lal tér vissza, hiba volt, illik kezelni...
- Meghívhatók többször a megfelelő rekurzív Give/Take megoldásokkal, de csak egy task által!
 - Csak akkor válik szabaddá, ha pontosan annyiszor engedik el (give), mint ahányszor elfoglaltál (take)
- Támogatja a prioritás öröklést

e. Notify, mutex, szemafor timeout

Közös erőforrásra vonatkozó kölcsönös kizárás megvalósítása esetén csak hibakezelésre!

- Az erőforrás foglalt, inkonzisztens állapotban van/lehet a timeout után
- Azt elvenni nem tudjuk (lásd később holtpont esetén erőszakos erőforrás elvétel)
 - Nem tudjuk a dolgunk csinálni, mert nincs erőforrás...

Szinkronizáció esetén:

- Nem olyan egyértelmű, hogy hibajelenség...
- Pl. ha adott idő alatt nem sikerült valamihez kapcsolódni, próbálkozunk mással/később
 - Pl. ilyesmivel vigyázni kell telepes/akkumulátoros üzemű rendszerekben
 - Miért? Mert ezzel nagyon gyorsan le lehet meríteni a telepet
 - Pl. telepes vezetékek nélküli szenzor node, ami ha nem látja a koordinátort, folyamatosan keresi azt, de ha látja, akkor 5 perente küld adatot...

f. Linux és a közös memória

A Linux-ban a közös memória több dolgot jelenthet:

- Egy folyamaton belül szálak esetén
- De: Létezik folyamatok között megosztott memória is Linux-ban (IPC részben lesz róla szó)
 - Ott is lehet használni
 - Speciálisan jelezni kell a közös memóriát váró megoldásoknak, hogy ilyet használunk...

Itt a szabványos POSIX kompatibilis hívásokat ismertetjük...

Kölcsönös kizárást biztosító megoldások:

- POSIX szemafor
 - POSIX mutex
 - POSIX rwlock
 - POSIX user-space spinlock
- Szinkronizáció
- POSIX Conditional variables

g. POSIX szemafor

Linux kernel támogatás: 2.6.x-től

Counter típusú, de nem támogatja a többszörös erőforrás foglalat

Két féle van:

- **Unnamed semaphores (memory-based semaphores)**
 - Thread-shared semaphore
 - Folyamat-specifikus címtérben (folyamat globális változó) kerül tárolásra
 - Folyamaton belül szálak közötti
 - Process-shared semaphore
 - Folyamatok közötti osztott memóriában létrehozva (POSIX shared memory)
 - Folyamatok között (és persze a bennük futó szálak között is)
- **Named semaphores (inkább IPC)**
 - A kernelben kerül tárolása
 - Megtalálható a /dev/shm könyvtárban egy adott néven
 - Nem tárgyaljuk, létrehozást és törlést eltekintve használni ugyan úgy kell, mint az elnevezettet

h. Szemafor létrehozása

Név nélküli szemafor

- `#include <semaphore.h>`
- `int sem_init(sem_t *sem, int pshared, unsigned int value);`
- Linker: `-pthread`

`sem_t`: a szemafort tároló adatstruktúra

`pshared`: thread vagy process shared szemafor jön létre

- 0 esetén thread shared
- nem nulla esetén process shared, és a szemafor tároló adatstruktúrát osztott memóriában kell létrehozni

Létező szemafor újrainicializálás nem definiált művelethez vezet!

A `value` értéke attól függ, mit akarunk elérni:

- Kölsönös kizárás egypéldányos közös erőforrásra: 1, az erőforrás szabad
- Kölsönös kizárás többpéldányos erőforrásra: az erőforrás példányok száma, az összes szabad
- Szinkronizáció: 0, a védett szakaszba nem lehet belépni

Visszatérési érték: 0 siker esetén, -1 hiba esetén, `errno` beállításra kerül

i. Várakozás a szemaforra

```
#include <semaphore.h>
```

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

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

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

Linker: -pthread

sem_wait: ha 0 a szemafor értéke akkor addig vár, amíg valaki nem növeli azt, vagy egy jelzés nem érkezik be

sem_trywait: ha 0 a szemafor értéke, azonnal visszatér

- errno beállításra kerül EAGAIN értékre

sem_timedwait: ha 0 a szemafor értéke, akkor addig vár, amíg azt nem növelik, vagy a megadott várakozási idő le nem jár, vagy jelzés érkezik be

```
struct timespec {  
    time_t tv_sec;    /* Seconds */  
    long tv_nsec;    /* Nanoseconds [0 .. 999999999] */  
};
```

Hiba esetén -1-gyel tér vissza, és errno-t beállítja

j. Szemafor értékének lekérdezése

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

Linker: -pthread

Lekérdezni az aktuális szemafor értéket

Hiba esetén -1-gyel tér vissza, és errno-t beállítja

- Pl. nem létező szemafor: EINVAL

A szemafor értéke a hívás visszatérésére is már megváltozhat (konkurens rendszer)

- Az érték alapján nem szabad az erőforrás állapotára következtetni!!!
- De pl. „felhasználói felületre” tájékoztató jelleggel kiírható..

k. Szemafor elengedés és törlése

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

Linker: -pthread

Hiba esetén -1-gyel tér vissza, és errno-t beállítja

- A szemafor értéke ekkor nem változik
- #include <semaphore.h>

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

○ Linker: -pthread

Felszabadítja a szemafort, de a memóriával nem foglalkozik, amiben a szemafor van, azt külön kell kezelni, ha szükséges

Visszatérési érték: 0 siker esetén, -1 hiba esetén, errno beállításra kerül

## I. Mutexek

### Lényegében bináris szemafor

- Tetszőleges erőforráshoz rendelhetjük
- A védett erőforrást lehet íni és olvasni is
- Kicsit el van bonyolítva a user-space POSIX mutex a paraméterezésében
  - Lehet, hogy egy kevesebbet tudó, egyszerűbb jobb lenne...

### A használata már egyszerű

- Létre kell hozni (saját memóriában)
- lock/unlock
- Meg kell szüntetni

## m. Mutex létrehozás, megszüntetés

```
#include <pthread.h>
```

```
int pthread_mutex_init(pthread_mutex_t *mutex,
const pthread_mutexattr_t *attr);
```

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

```
pthread_mutex_t mutex =
PTHREAD_MUTEX_INITIALIZER;
```

- Statikusan allokált mutex-eknek
- Ugyan az, mint pthread\_mutex\_init, NULL attr-rel

## n. Mutex lock és unlock

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

- Az adott mutex-et lezárja
- A viselkedése függ a mutex típusától

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

- Ha a mutex le van lokkolva, akkor azonnal visszatér hibával
  - EBUSY - The mutex could not be acquired because it was already locked.

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

- Az adott mutex-et felszabadítja
- A viselkedése függ a mutex típusától

#### o. rwlock

Lényegében egy bináris szemafor írásra vonatkozóan

- Tetszőleges számú párhuzamos szál olvashat, ha nincs író, és szabadon léphetnek be olvasásra új szálak
- Egy író szál írhat (és ő olvashat is persze), de akkor senki nem léphet be sem írásra, sem olvasásra

Akkor merül fel, ha egy komplex, nagy adatstruktúrát sok szál akarja olvasni, de csak kevés és ritkán akarja írni

- Nagyobb viszont az overhead, mint a mutex esetén
- Több végrehajtó egység esetén különösen hasznos, az olvasok tényleg párhuzamosan futhatnak...

Létre kell hozni

Olvasás: rdlock, unlock

Írás: wrlock, unlock

Meg kell szüntetni

#### p. rwlock létrehozás és megszüntetés

```
#include <pthread.h>
```

```
int pthread_rwlock_init(pthread_rwlock_t
*rwlock, const pthread_rwlockattr_t *attr);
```

- Az rwlock létrehozása

```
int pthread_rwlock_destroy(pthread_rwlock_t
*rwlock);
```

- Az rwlock megszüntetése

```
pthread_rwlock_t
```

```
rwlock=PTHREAD_RWLOCK_INITIALIZER;
```

- Statikus default rwlock, az attr NULL értékű

```
int pthread_rwlock_rdlock/wrlock: lock olvasásra/írásra
```

#### q. Spin lock: aktív várakozás

Aktív várakozás az erőforrás felszabadulására...

Akkor ajánlott, ha tudjuk, hogy a várakozás rövid idejű (usec), és az OS-en keresztüli hívás (semaphore, mutex, rwlock) overhead-je nagyobb

Nagy energia fogyasztás (nem tud a CPU aludni várakozás közben)

Egyprocesszoros rendszerben akár holtpont is lehet belőle

- CFS (default) ütemezési osztály esetén inkább csak lassabb futás lesz belőle, de FIFO RT ütemezési osztály esetén már garantált a deadlock

Kicsit eltérő az interface-e a mutex, rwlock, conditional variable (jön) megoldásokhoz képest

```
#include <pthread.h>
```

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

- Nincs default statikus inicializáló makró
- PTHREAD\_PROCESS\_SHARED/PTHREAD\_PROCESS\_PRIVATE

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

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

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

- Vagy lokkolja, vagy hibával tér vissza

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

#### r. Conditinal variable

Események hatására a vezérlés átadása

- Lényegében egyoldalú szinkronizáció
- Azért illik vizsgálni más módon is, hogy az esemény tényleg megtörtént-e

Kombinálni kell egy mutex-szel, ami a conditional variable-t védi

```
#include <pthread.h>
```

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

```
int pthread_cond_init(pthread_cond_t *restrict
cond,
```

```
const pthread_condattr_t *restrict attr);
```

```
pthread_cond_t cond =
PTHREAD_COND_INITIALIZER;
```

- 
14. **Tétel** 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).

#### a. Üzenetek vs közös memória

Közös memórián keresztül (RAM v. PRAM modell):

- Egy folyamaton belül futó szálak esetén (közös memória)

Üzenetekkel:

- Nincs közös memória
  - Folyamatok kommunikálnak egy operációs rendszeren belül
  - Elosztott rendszer

- Vagy pl. mikrokernél alapú az operációs rendszer

Folyamatok közötti kommunikáció (Inter Process Communication, IPC)

## b. Üzenetek

Nem azonos a számítógép hálózatokban küldött üzenetekkel, annál általánosabb fogalom

Üzenet továbbítás (Message passing)

Lehet például:

- Rendszerhívás (akár függvény hívás is)
  - Függvény esetén: Az adat vagy a rá mutató pointer jegyként (token) működik
    - Akinél a jegy van, az kezeli az adatot (a kölcsönös kizárás biztosított)
    - A jegyet versenyhelyzet mentesen kell átadni
      - » Pl. a pointer gépi adattípus, alkalmas módon atomi módon átadható!
- TCP/IP kapcsolat (TCP) vagy üzenet (UDP) gépen belül (localhost) vagy akár gépek között

Többnyire az alkalmazás kódjában OS API függvény/metódus hívásként jelenik meg

Az operációs rendszer valósítja meg szolgáltatásaival

## c. Üzenetovábbítás tulajdonságai

A közös memóriához képest:

- Nagyobb késleltetés
- Kisebb sávszélesség
- A csatorna nem megbízható
  - Közös memóriát 1 valószínűséggel megbízhatónak tartjuk
    - RAM, PRAM modell
  - Az operációs rendszeren belül történő üzenetküldésre ez már nem igaz
    - Túlterhelés!
  - A számítógép hálózaton történő üzenetküldés definíció szerűen nem megbízható (véletlen és szándékos hibák)
    - Két hadsereg probléma...
      - » Lehetséges-e egy nem megbízható kommunikációs csatornán 1 valószínűséggel megbízhatóan kommunikálni?
      - » Válasz: NEM (véges számú üzenetváltással)

## d. Két hadsereg probléma

Nem megbízható csatornán kialakíthatunk-e 1 valószínűséggel megbízható kommunikációt?

- Hadsereg A a két része egyidejű támadása esetén nyer
- Hadsereg A csak Hadsereg B-en keresztül, futárokkal kommunikálva egyeztetheti a támadás időpontját
- Egyszerű csomagvesztést tételezünk fel és más módon nem lehet kommunikálni
- Nincs 100%-os megoldás, A két része nem lehet 1 valószínűséggel biztos a kommunikáció sikerében!



#### e. Üzenetek címezése

Számítógép hálózatok...

Egy adott folyamat (unicast cím)

Minden folyamat (broadcast cím)

- Pl. teljesítmény menedzsment események
  - Standby, Hibernate, PowerOff, stb.

Folyamatok csoportja (multicast cím)

Egy folyamat (anycast cím)

- Pl. Egy alkalmas folyamat, ami a kérést ki tudja szolgálni

#### f. Direkt kommunikáció

Szimmetrikus üzenet alapú kommunikáció

- send(P, message)
- receive(Q, message)
- P, Q folyamat azonosítók. A vevő megadja az adót
  - Q-t meg kell adni a receive() hívásakor!
- A message egy adatstruktúra, ami az üzenetet tartalmazza

Aszimmetrikus üzenet alapú kommunikáció

- send(P, message)
- receive(id, message)
- P folyamat azonosító, id a küldő azonosítója. A vevő bárkitől fogad üzenetet!
  - Az id is visszatérési érték, bárki küldhet

- A message egy adatstruktúra, ami az üzenetet tartalmazza

A kódban direkt hivatkozás van a másik félre

- Nem szerencsés...

#### g. Indirekt kommunikáció

Egy köztes szereplőn keresztül történik a kommunikáció

- Proxy tervezési minta

Ez a szereplő pl. lehet: Postaláda (Mailbox), Üzenetsor (MessageQueue), Port, stb.

Interface: létrehozás és megszüntetés +

- send(A, message)
- receive(A, message)

Az „A” a postaláda „azonosítója”

- Elosztott rendszerben ennek része a node (számítógép) azonosító is



#### h. További tulajdonságok

Minimum egy küldő folyamat

Minimum egy vevő folyamat:

- 1. Vétel után törlődik (egyszer olvasható)
- 2. Vétel után megmarad (RAM-szerű működés).
  - Osztott memória üzeneteken keresztül (Pl. SystemV Shared Memory)

Lehet tulajdonosa:

- Operációs rendszer
  - Az őt használó folyamatoktól függetlenül létezhet
- Folyamat (annak a memória területén van)
  - A folyamattal együtt létezhet csak

#### i. Blokkolás/ nem blokkolás (szinkron/aszinkron)

Nem blokkoló hívás = aszinkron

- Az eredmények és mellékhatások a visszatéréskor még nem jelentkeznek (nem történtek meg)
- Csak a végrehajtás kezdődik el a hívásra
- A visszatérési érték kezelése, és az eredmények és mellékhatások kezelése csak más értesítés után lehetséges:
  - Pl. esemény, jelzés (signal), callback függvény, stb.

Blokkoló hívás = szinkron

- Az eredmény és mellékhatások a visszatérés után jelentkeznek (megtörtént)
- Visszatérési érték kezelés egyszerű...

#### j. Blokkolás/nem blokkolás a küldő és fogadó oldalon

|               | Küldő oldal                                                                                                                                                                                                                                                | Fogadó oldal                                                                                                                                                                                   |
|---------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Blokkolás     | <ul style="list-style-type: none"><li>• send() nem tér vissza, amíg az üzenetet nem vették át, vagy nem került a postaládába</li><li>• kommunikációs hiba esetén egy callback függvény hívódik meg, vagy időtúllépés + send() hibával tér vissza</li></ul> | <ul style="list-style-type: none"><li>• A Recive() nem tér vissza, amíg meg nem érkezik az üzenet</li><li>• Pl: TCP/UDP socket listen() függvény</li></ul>                                     |
| Nem blokkolás | <ul style="list-style-type: none"><li>• üzenet lokális elküldése után a send() azonnal visszatér</li><li>• vagy nincs hibakezelés, vagy a hibáról egy függvény által értesül a küldő</li></ul>                                                             | <ul style="list-style-type: none"><li>• Recive() azonnal visszatér</li><li>• Ha nincs üzenet, akkor azt jelzi (ebben az esetben végtelen ciklusban üzenete várás emészti a CPU időt)</li></ul> |

## k. Implementációk

### Mailbox:

- Indirekt kommunikáció
- Egy vagy több üzenet tárolása, de többnyire véges számú
- Operációs rendszer szintű támogatás

### MessageQueue:

- Indirekt kommunikáció
- Többnyire végtelen számú üzenet tárolására alkalmas
  - Természetesen a rendszer erőforrásai korlátozzák
- Üzenet alapú middleware-ek
  - MSMQ, IBM's WebSphere MQ, Oracle Advanced Queuing (AQ), JBoss Messaging, Apache Qpid

### Beágyazott OS-ek is jellegzetesen a Mailbox/MessageQueue megoldásokat „erőltetik”

- Sok esetben csak az információt tartalmazó adatstruktúrákra mutató pointereket tartalmaz ezekben az esetekben a Mailbox vagy MessageQueue

### TCP/IP TCP vagy UDP port:

- Direkt kommunikáció
- Socket interface
- Gépen belül a localhost-on (127.0.0.1/8)
  - Többnyire nem ajánlott beágyazott rendszerekben mert erőforrás igényes
- Alacsony szintű, számos más middleware alapul rajta:
  - Távoli eljárás hívás (Remote Procedure Call, RPC)
  - Távoli metódus hívás:
    - CORBA (Common Request Broker Architecture),
    - JAVA RMI (Remote Method Invocation),
    - DCOM/.NET Remoting,
    - SOAP (Simple Object Access Protocol).
  - Üzenet alapú middleware-ek (korábban volt szó).

### Különböző folyamatok és csővezetékek:

- Többnyire direkt
- Pl. UNIX pipe, Windows Named Pipe, RTLinux FIFO

### System V Shared Memory (UNIX, Linux)

- Direkt
- MMU meglétét és fejlett használatát feltételezi
  - Az MMU teszi elérhetővé a folyamatok címterében (virtuális memória) a megosztott fizikai memóriát
- A megosztott memóriában elhelyezett adatstruktúrákra a kölcsönös kizárást biztosítani kell a már korábban ismertetett módszerekkel
- Nagyobb teljesítményű beágyazott operációs rendszerekben érhető el (QNX, Linux, stb.)

+ jelzések

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

a. Jelzések

**Jelzék (signal)**

- Direkt kommunikáció
- Sok esetben tényleges adat nem kerül átvitelre
  - Inkább szinkronizáció a cél
- Példa: POSIX jelzések Linux-ban
  - kill parancs...

Alapértelmezett módon a folyamat stackjében fut le a signal handler, ha van

- Kb. mint egy interrupt handler
  - Komoly következmények:
    - Rövidnek kell lennie
    - Sok API függvény nem „signal safe”, lista „man 7 signal”
    - Alapértelmezett esetben (ha nem a program tiszta leállítása a feladat), kb. egy flag vagy event számláló beállítása fér bele, amit a főprogram kezel, amikor tud
      - » Még a printf sem jó ötlet (lassú), max. demonstrációs célból...

○ De lehet saját stack-je is (sigaltstack)

- Pl. a SIGSEGV jelzés lekezelése esetén elkerülhetetlen...
- A handler-ek regisztrálása előtt be kell állítani
- fork() esetén a fork-olt gyermek is megkapja az alternatív stack-et, de már az execve() törli azt

○ A rendszerhívások működését is befolyásolja

- Azok hibával leállhatnak (EINTR) vagy újraindulnak, az alapértelmezett viselkedés architektúra és hívás függő...

b. TCP/IP, UDP port

**TCP/IP TCP vagy UDP port:**

- Direkt kommunikáció
- Socket interface
- Gépen belül a localhost-on (127.0.0.1/8)
  - Többnyire nem ajánlott beágyazott rendszerekben mert erőforrás igényes
- Alacsony szintű, számos más middleware alapul rajta:
  - Távoli eljárás hívás (Remote Procedure Call, RPC)
  - Távoli metódus hívás:
    - CORBA (Common Request Broker Architecture),
    - JAVA RMI (Remote Method Invocation),
    - DCOM/.NET Remoting,
    - SOAP (Simple Object Access Protocol).
  - Üzenet alapú middleware-ek (korábban volt szó).

### c. Csővezeték és megosztott memória

Különböző folyamatok és csővezetékek:

- Többnyire direkt
- Pl. UNIX pipe, Windows Named Pipe, RTLinux FIFO

System V Shared Memory (UNIX, Linux)

- Direkt
- MMU meglétét és fejlett használatát feltételezi
  - Az MMU teszi elérhetővé a folyamatok címtérében (virtuális memória) a megosztott fizikai memóriát

### d. Reakció a jelzésekre

Term: Default action is to terminate the process

Ign: Default action is to ignore the signal

Core: Default action is to terminate the process and dump core

Stop: Default action is to stop the process

Cont: Default action is to continue the process if it is currently stopped

### e. Jelzések generálása

```
#include <sys/types.h>
```

```
#include <signal.h>
```

```
int kill(pid_t pid, int sig);
```

- Adott pid-ű folyamatnak küld egy adott sig-et

```
int killpg(int pgrp, int sig);
```

- Az adott process group minden tagjának küld egy adott sig-et

```
int raise(int sig);
```

- A hívó folyamat önmagának küld jelzést, ha vannak szálak, akkor a hívó szál kontextusában fut le a jelzés kezelés

```
int pthread_kill(pthread_t thread, int sig);
```

- Folyamat szintű a hatás, csak a megadott szál kontextusában fog lefutni a jelzés kezelése

```
int tkill(int tgid, int tid, int sig);
```

- Adott folyamaton belül adott szálnak küld jelzést

#### f. Jelzésekre várakozás a főprogramban

`int pause(void)`

- Csak akkor tér vissza, ha meg volt hívva a signal handler, és akkor is -1 és errno be van állítva EINTR-re (lefutott a handler)

`int sigsuspend(const sigset_t *mask);`

- Ideiglenesen lecseréli a hívó folyamat signal mask-ját a megadottra, majd felfüggeszti a folyamatot

`int sigwaitinfo(const sigset_t *set, siginfo_t *info);`

- Szinkron várakozás számban adott jelzésekre (set)
- Visszatér a signal azonosítójával

- Ha az info struktúra nem NULL, akkor a jelzés paramétereivel

`int sigtimedwait(const sigset_t *set, siginfo_t *info, const struct timespec *timeout);`

- Mint sigwaitinfo, csak egy adott idő után visszatér akkor is, ha nem érkezik jelzés

### 16. Tétel FreeRTOS IPC jellegű megoldásai.

#### a. FreeRTOS IPC megoldások

**Mailbox:**

- Indirekt kommunikáció
- Egy vagy több üzenet tárolása, de többnyire véges számú
- Operációs rendszer szintű támogatás

**MessageQueue:**

- Indirekt kommunikáció
- Többnyire végtelen számú üzenet tárolására alkalmas
  - Természetesen a rendszer erőforrásai korlátozzák

- Üzenet alapú middleware-ek

- MSMQ, IBM's WebSphere MQ, Oracle Advanced Queuing (AQ), JBoss Messaging, Apache Qpid

#### b. FreeRTOS Queue viselkedés

**FreeRTOS Queue (sor) viselkedés:**

- Adott számú elemet tartalmazhat
- Érték szerint másolunk a sorba
  - Az alternatíva a referencia szerinti

- Igazából az adatméretet kell megadni (a sor nem típusos, void pointer-t vár)

- Természetesen átvadható pointer is

— De ekkor az átvadott adathoz már az átvadó nem férhet hozzá (kölcsonös kizárás nincs)

- Queue Set (sor készlet) állítható belőlük össze

**A sorba való írásra és olvasásra is lehet/kell várni**

- Van timeout

- Olvasás esetén nincs adat (Queue is empty)
- Írás esetén ha a sor tele van (Queue is full)

- Mindig egy task fut ha adat érkezik/kikerül a sorból

- Mindig a legmagasabb prioritású várakozó fut

**Sor készletre is lehet várni olvasás esetén:**

- Ezzel egy feladat több dologra tud várni

- Kicsit hasonlít a UNIX select()-re funkcióban

### c. Queue API

Queue létrehozása:

- `QueueHandle_t xQueueCreate( UBaseType_t uxQueueLength, UBaseType_t uxItemSize );`

Küldés a sorba:

- LIFO sor küldés: `xQueueSendToFront()` és `xQueueSendToFrontFromISR()`
- FIFO sor küldés: `xQueueSendToBack()` és `xQueueSendToBackFromISR()`
- Ha megadunk timeout-ot, akkor vár, ha a sor tele van
- Ha nem adunk meg timeout-t, akkor hibával tér vissza tele sor esetén

Fogadás sorból

- `xQueueReceive()` és `xQueueReceiveFromISR()`
- Ha megadunk timeout-ot, akkor vár, ha a sor üres
- Ha nem adunk meg timeout-t, akkor hibával tér vissza üres sor esetén

Sor hosszának lekérdezése

- `uxQueueMessagesWaiting()`
- Felhasználó felületre vagy debug célból

### d. Mailbox

Egy elemű sor

- A legutoljára beírt infót lehet tetszőleges számú alkalommal olvasni

Írás:

- `xQueueOverwrite()` és `xQueueOverwriteFromISR()`
- Csak egyelemű sor (mailbox) esetén használható
- Az infót felülírja
- Nincs timeout és mindig sikeresen fut (felülírja az infót, nincs hibakód)

Olvasás

- `xQueuePeek()` és `xQueuePeekFromISR()`
- Csak egyelemű sor (mailbox) esetén használható
- Van timeout és az üres (még nem írt) mailbox-ra vonatkozik
- Ha nincs timeout, és a mailbox még nem lett írva, hibával tér vissza

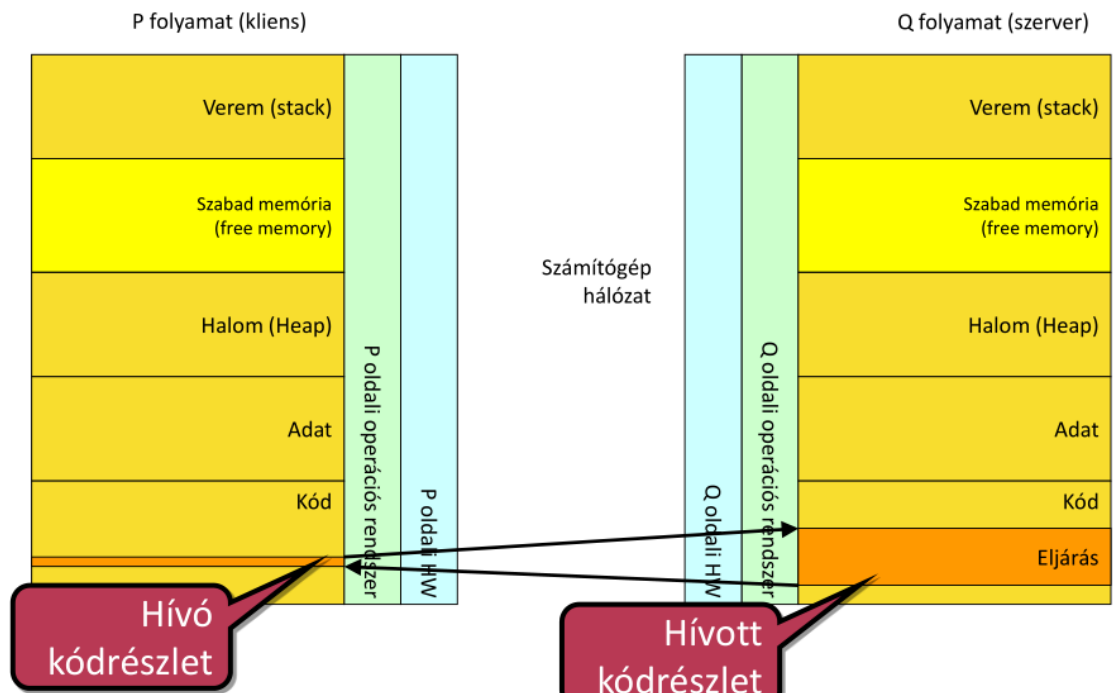
## 17. Tétel Távoli eljárás hívás és annak implementációja Linux alatt.

### a. Távoli eljáráshívás

Távoli eljárás hívás (Remote Procedure Call, RPC):

- Egy másik folyamat kód memóriaterületén elhelyezkedő függvény „meghívása” a hívó folyamatból üzenetek felhasználásával
- A hívó fél blokkolva vár a távoli hívás lefutására
- A meghívott függvény az őt tartalmazó folyamat egyik vezérlési szálában fut le

## b. RPC architektúrája



## c. RPC a programozó szemével

Hívása lényegében azonos egy „lokális” függvény hívásával

- Az lényegében egy programkönyvtárban található függvény
- Nem kell tudnia, és figyelembe vennie milyen platformon fut le

A ténylegesen végrehajtott függvény megírása sem különbözik egy lokálisan végrehajtható függvény megírásától

- Kapunk egy függvény deklarációt (interface-t), és azt meg kell valósítanunk
- Nem kell törődnünk azzal, hogy milyen platformról érkezik a kérés

## d. RPC működése

Típusos a hívás és a visszatérési érték is

- Strukturált adatküldés
- Platform függetlenség megvalósítása az operációs rendszer és a környezet (fordító vagy interpreter) feladata

A kliens program egy teljesen normális lokális függvényt hív meg

- Ez egy automatikusan generált csonk (stub) valójában
- Feladata a kommunikáció részleteinek elrejtése a hívó fél elől

### A kliens oldali csonk feladata a hívás során

- Feladata a hívási paraméterek összecsomagolása platform független formára és elküldése a végrehajtó folyamatnak
- Ehhez felhasználja a lokális operációs rendszer és a hálózat szolgáltatásait

### A szerver oldalon az RPC-t használó szolgáltatás megkapja a szabványos formátumban megérkező hívást

- Platform független formáról lokális formára hozás
- A lokális függvény hívása
- Visszatéréskor a visszatérési érték platform független formára hozása
- A visszatérési érték küldése a kliensnek

### A kliens oldali csonk feladata a hívás visszatérése során

- Feladata a szervertől megkapott visszatérési érték konvertálása platform független formáról lokális formára
- Visszatérés a lokális csonkból a lokális formátumra hozott visszatérési értékkel

### A kliens szál az RPC hívás során várakozik

- Eseményre várakozik a hívás során

### A szerver szál vár a beérkező hívásokra, és amikor van kiszolgálandó hívás, akkor fut

- Eseményre várakozik a hívás beérkezéséig

## e. RPC a gyakorlatban

#### Kérdések:

- Hogyan találjuk meg a hívott felet?
  - Manuális konfiguráció
  - Broker pattern alapján automatikus konfiguráció

#### RPC szabványok:

- DCE/RPC (Distributed Computing Environment)
- SUN RPC

- Tulajdonképpen egy URL is felfogható RPC hívásnak
  - scheme://username:password@domain:port/path?query\_string#fragment\_id
  - Query string: key0=value0&key1=value1
  - Ez akár el is vezethet a távoli metódus híváshoz

#### Szabványos adatformátumok (serialize)

- BER (Binary Encoding Rules)
- XML
- JSON

Ha lesz idő, lesz SUN RPC példa a gyakorlaton

## f. Távoli metódushívás

Egy távoli objektum megadott metódusát hívjuk meg

#### Remote Method Invocation (RMI) szabványok:

- CORBA (COMmon Request Broker Architecture)
- RT CORBA, FT CORBA

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

a. Publish – subscribe

Résztevők

- Publisher = Termelő
- Subscriber = Fogyasztó

A termelő által előállított információra a fogyasztó előfizet

- A termelő nyilvántartja az általa előállított információra előfizetőket
  - Referencia a fogyasztók adatfogadó interfészére
- Ha van információ, elküldi azoknak az előfizetőknek (push)
  - Egyenként mindenkinek, vagy multicast formában
  - Pl. RPC-vel

Bonyolult „hálózatokat” lehet felépíteni olyan komponensekből, amik más termelőkől érkező adatok alapján új adatokat állítanak elő

- Adatvezérelt szerkezet (dataflow)

Megoldások

- Matlab Simulink
- Ni Labview

Az architektúra lehetővé teszi a többprocesszoros működést is

- A blokkok párhuzamosan futhatnak
- Laza csatolás a blokkok között

b. Virtuális adatbusz

Számos kommunikációs hálózat adatbuszként is felfogható (broadcast a fizikai közegben):

- Pl. CAN
- Minden üzenet eljut minden NODE-hoz
- Az üzenet címe alapján egy NODE veszi vagy nem veszi, majd a tartalom alapján válaszol vagy nem

Virtuális adatbusz

- Pont-pont kommunikációs hálózat, de szükség van buszszerű működésre
- Publish-Subscribe alapján, arra ráépülve létrehozható egy virtuális adatbusz
- Minden node „termelő”, amikor a buszra ír
- Minden node „fogyasztó”, amikor a többi node-tól a buszon keresztül adatot kap
- A busz egy központi node az elosztott rendszerben, ahol minden adat megtalálható, sokat kommunikál, szűk keresztmetszet lehet!

c. Proxy

A proxy beáll két vagy több kommunikálni kívánó fél közé

Cél:

- Kompatibilitás megteremtése
- Funkciók rejtése
- Teljesítményillesztés

#### d. Proxy kompatibilitás megteremtése

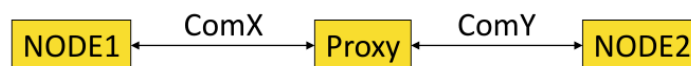
Kettő vagy több fél, akik eltérő kommunikációs felületet használnak

Magas szinten az átjuttatandó információ viszont értelmezhető minden fél számára

Meg kell teremteni a kapcsolatot...

Egy köztes fél végzi:

- A kapcsolat felvételét a felek között
- Az üzenetformátumok közötti transzformációt



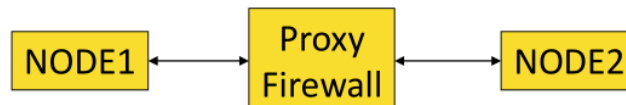
#### e. Proxy – funkciók elrejtése

Felhasználás:

- A NODE1 csak egyes funkcióit szeretnénk elérhetővé tenni NODE2 számára (allow) vagy A NODE1 egyes funkcióit nem szeretnénk elérhetővé tenni NODE2 számára (deny)
- NODE1 nem változtatható meg, vagy a funkciókra más „belső” node-ok esetén szükség van.

NODE1 nem érhető el direkt módon, csak a proxy-n (firewall/tűzfal) keresztül NODE2 számára

A NODE1-nek küldött nem engedélyezett kéréseket a tűzfal nem küldi tovább

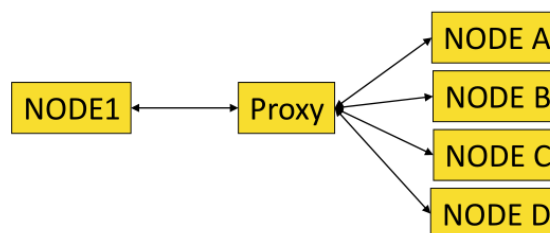


#### f. Proxy – teljesítményillesztés

NODE1 nem képes a beérkező kéréseket kiszolgálni kis teljesítménye miatt

NODE1 hardver nem lecserélhető

A Proxy lekérdezi NODE1-et, és az így elért információt CACHE jelleggel elérhetővé teszi a többi NODE számára (NODE A, NODE D, NODE N)



#### g. Általános eset

Vezeték nélküli szenzorhálózat illesztése az Internetre (elérhetővé kell tenni a gyűjtött adatokat)

- A proxy itt mind a három szerepet fel kell hogy vállalja

1. A vezetéknélküli szenzorhálózat nem TCP/IP alapú, ezért az üzeneteket át kell alakítani

2. A vezetéknélküli szenzorhálózatot meg kell védeni az Internet felől jövő támadásoktól, pl. konfigurálási funkciók, stb.

3. A vezetéknélküli szenzorhálózatban a kommunikáció lassú, pl. ha a teljes Internet egy bizonyos hőmérséklet mérőt akar lekérdezni, akkor a szenzorhálózat túl fog terhelődni, ha az érték ideiglenesen tárolásra kerül a proxy-ban, az összes kérés CACHE-elve kiszolgálható a proxy-ból

#### h. Bróker

Központi szereplő, amely összepárosítja a kommunikálni kívánó feleket

- Ismert a felek számára

- Az elérhetősége meg van adva a rendszer konfigurációjában
- Egyszerű automatizmussal megtudható

- A felek vele veszik fel a kapcsolatot

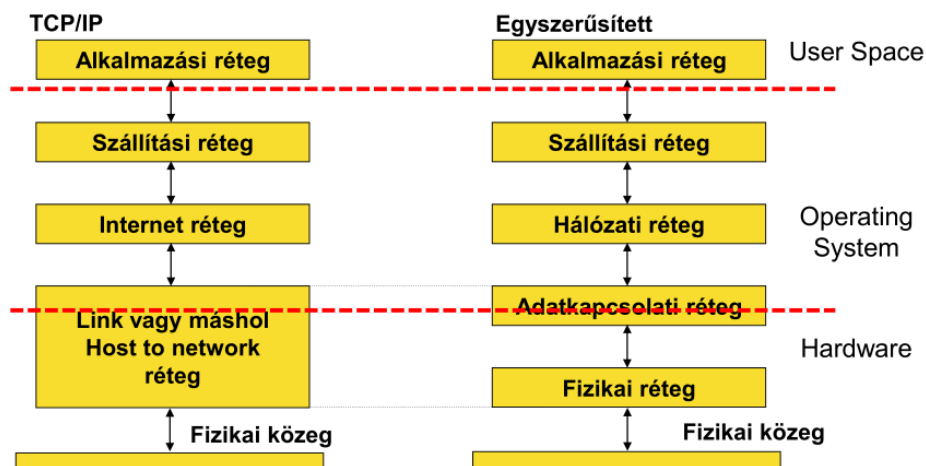
- Megadják a felek igényeiket és/vagy az általuk nyújtott szolgáltatásokat

- A bróker összeválogatja azokat, megteremti a kapcsolatot a felek között

- Utána a felek egymással kommunikálnak

19. **Tétel:** 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.

a. TCP/IP és az egyszerűsített referenciamodell:



**Problémák:**

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

b. Mit támogat a Linux?

**Alacsony szintű HW driver**

- o Ethernet, WI-FI, 3-4-5G, stb.

- o Ha a HW-t nem támogatja, akkor kernel driver-t kell írni (kernel tanfolyam)

- o SW szinten teljesen el van rejtve...

- Van amikor hozzá kell férni a HW-hez, ahhoz is kell interfész...

**Magas szintű SW**

- o IPv4

- Mi most ezzel fogunk foglalkozni

- o IPv6

- Ez egyre fontosabb lesz a jövőben, nem sokban különbözik SW szintjén az IPv4-től

- o IPX

- o Appletalk

- o és néhány régi protokoll...

c. Mi kell a sikeres kommunikációhoz?

**Működő alacsony szintű interfész**

○ ifconfig parancs

- Kommunikációra használt eszköz neve (ethX, lo)
- Ethernet és IP címek, beállítások, statisztikák
- Egyes HW beállítások (IT, Base address)
- Egyes disztribúciókban csak a root számára érhető el (debian), még Ubuntu-ban minden user számára

○ ping parancssal tesztelhető

- ping „IP cím”

○ Működő DNS szolgáltatás

- ping „domain név”
- resolv.conf

○ Működő routing (min. 2 interfészünk van...)

- netstat -r

d. Tűzfal alapok

**Szabályok kezelése (iptables parancs)**

○ Többnyire root jog szükséges hozzá...

○ Aktuális táblák listázása: „iptables -L”

- -v kiírja a számlálókat is, néha jó...

○ Alapértelmezetten 4 szűrő tábla van (tables)

- Default táblák: filter (default), nat, mangle, raw

○ Különböző láncokra (chain) vannak fűzve a szabályok egy táblán belül

- Pl. default láncok a filter tábla esetén: INPUT, OUTPUT, FORWARD
- Egy adott láncon a szabályok (rules) sorrendben kerülnek feldolgozásra, az első „illeszkedő” szabályt hajtja végre a rendszer
  - A default viselkedés a láncon –policy v -P : DROP all -- anywhere anywhere
  - Dobj el mindent, bárhonnan is jött és bárhová is tart

○ A szabályok lehetne interfész specifikusak is

- Csak az adott interfészen bejövő, kimenő, stb. csomagokra érvényesek ekkor

A legtöbb Linux disztribúcióban rendelkezésre áll a tűzfal

e. Tűzfal beállítása

**Direkt manuálisan beállított szabályokkal az iptables hívásával**

- Alap esetben elfelejtődnek a rendszer újraindításakor!
- Disztribúció függő módon lehet őket véglegessé tenni
- Lényegében az alkalmazott rendszerindítási folyamatba vagy a hálózati interfészeket elindító szkriptekbe kell a szabályokat egyesével létrehozni az iptables megfelelően paraméterezett hívásával

**Valamilyen GUI vagy karakteres tűzfal konfiguráló alkalmazással**

- Firestarter
- Gufw
- Firewall Builder

#### f. Protokoll analízátorok

### Protokoll analízátorok (Protocol Analyzer)

- HW vagy SW?
  - Most csak foglalkozunk az SW-vel
  - Részletesebben foglalkozunk velük a „Hálózatba kapcsolt beágyazott rendszerek” tárgyban
- A referencia modell különböző rétegeiben képesek vizsgálatokat végezni
- Kérdés, hogy milyen módon kapcsolódunk az analízált hálózathoz
- Megjelenítik, értelmezik és feldolgozzák az információt, általános célú eszközök
  - Hardver és szoftver fejlesztés, rendszertelepítés, hálózati hiba keresés

#### g. SW protokoll analízátorok

Adatkapcsolati rétegtől kezdve képesek működni

- Nem csak Ethernet estén működnek, WIFI, PPP, USB, stb. kapcsolatokra is működnek bizonyos megkötésekkel

Forgalom rögzítése:

- Speciális capture card
  - Nem foglalkozunk vele
- Standard Ethernet NIC (dedikált vagy nem dedikált)
  - Hibák...
  - Csatlakozás a vizsgált rendszerhez...

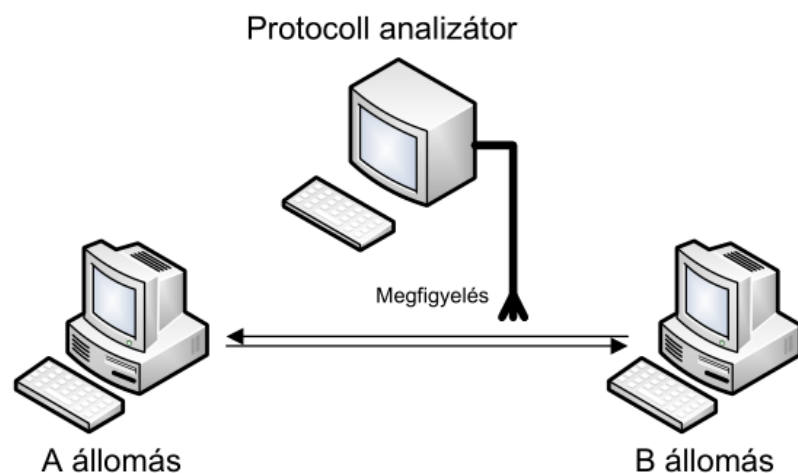
#### h. Analízátor csatlakoztatása a rendszerbe

Két kommunikáló fél esetén

- Dedikált analízátor esetén lásd ábra
- A vagy B állomásokon is futtat egy SW analízátor

Elosztott rendszerben?

- Alkalmas hely megtalálása
- Elosztott analízátor



#### i. Számítógép alkalmazása protokollanalizátorként: TCPDUMP és WireShark

Tcpdump: Karakteres felület...

Wireshark: GUI, de a tshark karakteres

A WireShark a legjobb ingyenes szoftver, a legtöbb alkalmazásban elég a tudása

Hálózati kártyának támogatnia kell a packet capture-t

- WHCL tanúsítvánnyal rendelkező NIC-ek tudják
- TCP Offload Engine (TOE) megzavarhatja
- Többnyire a driverben kikapcsolható Windows alatt

#### j. WireShark

Néha jól jön, hogy Linux alatt tud USB-ét is analizálni...

- Sőt, már Windows alatt is tud, egy speciális driver felinstallálása után...

A WI-FI analízis igazán Linux alatt működik, egy megfelelő NIC-kel (típusfüggő)

#### k. TCPDUMP

Egyszerű karakteres felület

- tcpdump –kapcsolók szűrőkifejezés

Fontosabb kapcsolók

- -i: Adott interfészen hallgatózik
- -B: OS capture buffer méret beállítása, 1MByte a default
- -D: A használható hálózati interfészek listája
- -w és -r file-ba vagy file-ból működés
  - Wiresharkkal kompatibilis a formátum!
- -t, -tt, -ttt, -tttt különböző timestamp formátumok (pl. delta)

Szűrőkifejezés

- Előre definiált konstansok, számok, és az and, or, és not kifejezésekből áll
- Adott host forgalma: tcpdump ... host ip/domain
- Adott host IP forgalma: tcpdump ... ip host
- Forrás és cél: tcpdump dst/src host
- Protocol: arp/ip/stp/udp/tcp
- Port: port X, ahol X a port száma

#### l. TCP és UDP összehasonlítása

TCP

- Kapcsolat orientált
- Megbízható
- Pont-pont
  - Unicast
- Folyamszabályozás és torlódásvezérlés
- Összetett
- Erősen optimalizált
- Erőforrás igényes (CPU és memória is)
- Késleltetés ingadozik hiba esetén (újraadás)
- Kétirányú byte stream interface

UDP

- Üzenet orientált
- Nem megbízható
- Multicast és broadcast is használható
- Nincs korlátozás, az adótól függ
- Egyszerű
- Nincs mit optimalizálni
- Nem igényel sok erőforrást
- Nincs újraadás, az adat hiányzik (multimédia...)
- Csomag alapú interface

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

a. **Socket**

**Socket: univerzális kommunikációs végpont**

- Konfigurálni lehet, hogy milyen kommunikációs szolgáltatást igénybe véve lehessen használni, és hogyan
- Lényegében egy file descriptor
- Ez a kernel és user space közötti alacsony szintű interfész

Eredetileg **a BSD UNIX-ból származik**

- Onnan örökölte meg mindenki

b. **Socket létrehozása, paraméterek**

```
#include <sys/socket.h>
```

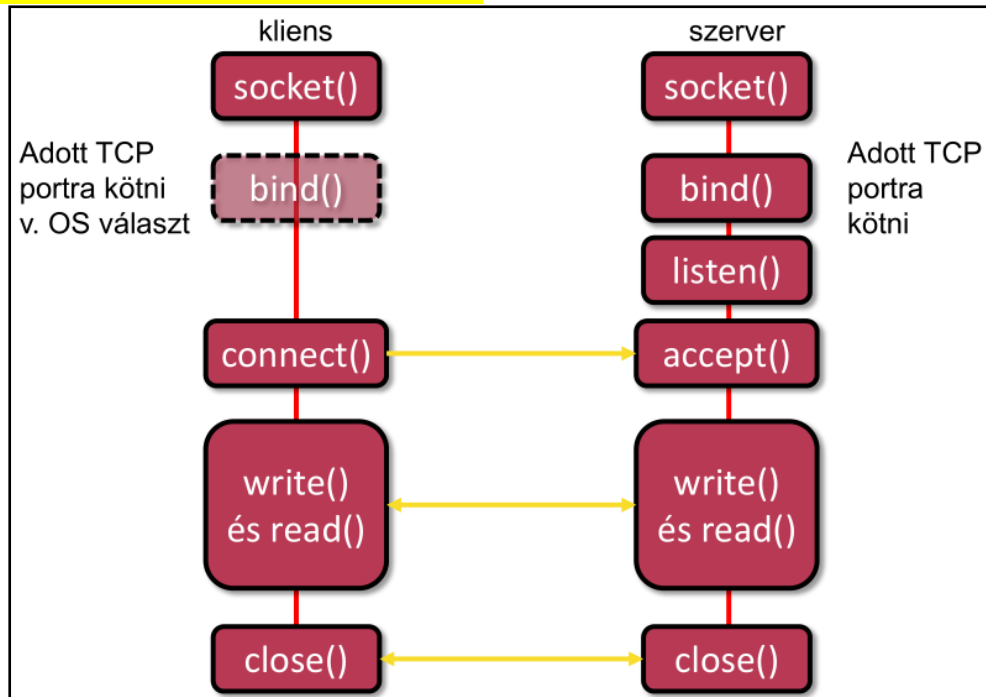
```
int socket(int domain, int type, int protocol);
```

Milyen paraméterek vannak?

| Domain                                                 | Type                                                                                                          | protocol                                     |
|--------------------------------------------------------|---------------------------------------------------------------------------------------------------------------|----------------------------------------------|
| UNIX domain socket                                     | <b>SOCK_STREAM</b> : megbízható, sorrend helyes, kétoldalú, bytfolyam alapú átvitel (TCP)                     | <b>Többnyire 0-t</b> adunk meg, az a default |
| <b>IPV4 socket</b>                                     | <b>SOCK_DGRAM</b> : csomag alapú, kapcsolatfüggő, nem megbízható átvitel (UDP)                                |                                              |
| <b>IPV6 socket</b>                                     | <b>SOCK_SEQPACKET</b> : Csomag alapú, sorrendhelyes, megbízható, kétoldalú (kapcsolat orientált) szolgáltatás |                                              |
| socket alapú kernel és user space közötti kommunikáció | <b>SOCK_RAW</b> : Alacsonyszintű hozzáférés a csomaghoz                                                       |                                              |
|                                                        | <b>SOCK_RDM</b> : Megbízható, csomagalapú, de nem sorrendhelyes szolgáltatás                                  |                                              |

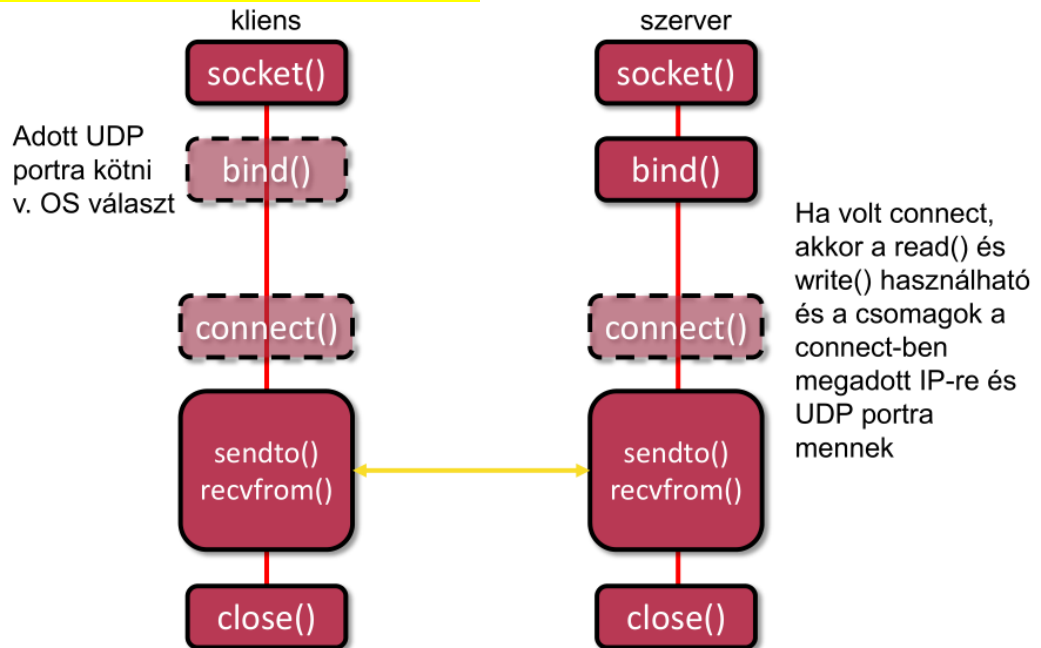
**függvény visszatérési értéke a socket azonosítója (hiba esetén -1)**

### c. TCP socket használata, munkamenet



- **`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 (a választás OS függő)
- **`listen()`** (szerver oldalon): Socketet passzív állapotba tesszük, felkészítjük bejövő kliensek fogadására. A bejövő socketnek SOCK\_STREAM vagy SOCK\_SEQPACKET típusúnak kell lennie
- **`accept()`** (szerver oldal): A visszatérési érték egy socket, amivel a kapcsolatot lehet használni. Erre a socket-re már csak adatátvitellel kapcsolatos függvényeket szabad meghívni. A beállítástól függően az accept blokkol, amíg nincs bejövő kapcsolat, vagy nem blokkol, és ha nincs kapcsolat akkor -1-el tér vissza.
- **`connect()`** (kliens oldal): Kliens oldalon ezzel kapcsolódunk a szerverhez. A sockfd-vel megadott socket-nek SOCK\_STREAM vagy SOCK\_SEQPACKET típusúnak kell lennie. Egyszer lehet meghívni egy ilyen socket-en.
- **Adatok küldése és fogadása**
  - o **`read()/write()`:** mint egy standard fájl esetén
  - o **`send()/recv()`:** `read()/write()`hoz hasonló, csak vannak különböző flag-ek is. (pl. csak LAN-ban elérhető gépeknek, nem blokkol, további adatokat fog küldeni...)
- **`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**

#### d. UDP socket használata, munkamenet



- **bind():** lásd előző eset
- **connect():** 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 a socketen. Visszatérési érték az elküldött byte-ok száma, vagy -1 hiba esetén. Itt meg kell adni paraméternek, hogy kinek küldjük az adatot.
- **recvfrom():** adatok fogadása. Csomag alapú megoldások esetén a teljes üzenetet ki kell olvasni. Visszatérési érték a fogadott byte-ok száma, vagy -1 ha hiba volt.
- **close():** socket lezárása

#### 21. Tétel: Hibakeresés eszközei és azok használata Linux alatt.

##### a. Valgrind

Különböző erőforráskezelést vizsgáló segédprogramok gyűjteménye:

*memcheck*: memóriakezelés vizsgálata

*cachegrind*: cache-hatékonyság vizsgálata

*callgrind*: hívási gráf felvétele

*helgrind*: pthread nyomkövetése

##### Általános használat:

- *valgrind --tool=tool* futtatható
- A legtöbb eszköz a lefordított alkalmazás utasításként vizsgálja, nézi, mit csinál a program
  - Éppen ezért kicsit lassabb, mint rendesen futtatva

## b. Trace

### ltrace

- Dinamikus függvényhívások listázása
  - Használható arra, hogy megtaláljuk, mit is kell majd eltéríteni
- Összegző mód: melyik függvény mennyiszer lett meghívva?

*ltrace futtatható*

*ltrace -c futtatható*

### strace

- Rendszerhívások naplózása:
  - Használható arra, hogy lássuk, milyen argumentumokat is továbbít egy függvény a kernel felé
  - Melyik C-programkönyvtárban megtalálható függvény mit is csinál

• *strace futtatható*

- Összegző mód:

• *strace -c futtatható*

## c. GDB (Gnu Debugger)

### Forrás szintű debugger

- Tudja követni, melyik kódsort hajtja végre a program
- Tudja, adott állapotban melyik változó milyen címen van
  - Debug információ

### Parancssoros/konzolos üzemmód

- Lehet távoli debug módban is használni: hálózat, soros port, stb.
- Léteznek grafikus frontendek: ddd, Eclipse, stb.
- Scriptelhető Python alól

### Watchpoint

- A hagyományos töréspont mellett egy másik hasznos funkció:
  - watch kifejezés [thread threadnum]
    - A kifejezés több változót tartalmazhat
    - A GDB leállítja a programot, ha threadnum szál módosítja a kifejezés egyik változóját, és ennek hatására annak értéke megváltozik
- Megvalósítás általában hardveres watchpointtal (pl. MMU segítségével)
  - Ha nem elérhető, akkor a jóval lassabb, kevésbé rugalmas (csak az aktuális szálon alkalmazható) szoftveres watchpoint működik csak.

#### d. fork() debug

- *fork()* esetén a gdb a szülő processz debuggolását folytatja
  - Ez állítható:
    - *set follow-fork-mode MÓD*
      - » *Mód: parent* vagy *child*
    - *set detach-on-fork (on|off)*
      - » *Off*-módban mind a gyermek, mind a szülő processz debuggolva lesz
- *info inferiors*
  - A debuggolt processzek listájának kiírása
- *inferior n*
  - Az *n*. inferior kiválasztása a debug műveletek számára
- A processzek külön vannak kezelve.

#### e. További feature-k

### Jelzések

- Értesítés, törés és szűrés állítása lehetséges
  - *handle JELZÉS beállítás*
    - *beállítás:*
      - » (no)print: írjon vagy ne írjon üzenetet a konzolra
      - » (no)stop: megállítsa-e a programot jelzés esetén
      - » (no)pass: továbbengedje-e a jelzést a programnak
  - *signal JELZÉS*
    - A program futásának folytatása, *JELZÉS* küldésével

### Többszálú alkalmazások

- Hasonló a processzekhez
  - Minden szál esetén értesül róla a debugger
- Lekérdezés:
  - *info threads*
- Vizsgált szál beállítása
  - *thread szálszám*
- Parancs végrehajtása szálon
  - *thread apply [szálszám] [all] parancs*

## Reverse debugging

- Ritkán, de hasznos lehet a programban visszafelé lépni, „visszavonni” utasítást
  - Valódi hardveren ez értelemszerűen nem könnyű
    - Minden végrehajtott utasítás viselkedését rögzíteni kell
  - Processzorszimulátoron elvben megoldható
    - Természetesen a szimulátornak kell naplóznia ebben az esetben

### A GDB korlátai

- Ugyan Linux fut a beágyazott rendszeren, de előfordulhat, hogy:
  - Nem fér el a GDB
    - Tárhely
    - Memória
  - Nincs interaktív vezérlésre lehetőség
  - Az adott rendszerhez nincsen portolva a GDB
- Párhuzamosan futó szálak és processzek hibakeresése (versenyhelyzet esetén) nehézkes, a hibajelenséget előidézni nem mindig könnyű
  - A GDB scriptelhetősége segíthet

f.