BOR

ZV kidolgozott tételek

Asbtóth Árpád

2017

Tartalom

1-2. Tétel

1. Mutassa be a uCOS-II felépítését és főbb szolgáltatásait

2. Ismertesse a microOS-II-ben a taszkok nyilvántartásához használt struktúrákat és használatukat! Mutassa be, hogy a nyilvántartásban hogyan lehet egy taszkot futásra késszé tenni, törölni a futásra készek közül, valamint megtalálni a legnagyobb prioritású futásra kész taszkot!

1. microC/OS-II

- általános célú RT OS
- egyszerű és icipici
- jól konfigurálható → jól optimalizálható az erőforrás igénye
- C nyelven írt forráskód elérhető
- nem open source, üzelit célokra meg kell venni
- széles CPU támogatás
- jó commmunity

2. OS felépítése

16 fájl

Processzorfüggő fájlok – Port files

- minimális platformfüggő rész – interfész a platformfüggetlen rész és a HW között
- fájlok:
  - *OS_CPPU.H*
    - proc specifikus definíciók
    - adattípus definíciók
      - pl. nem engedhető meg, hogy az int minden processzoron más legyen (nem kiszámítható memórahasználat pl.)
      - INT8U, INT8S, INT16U, INT16S, INT32U, INT32S, FP32 stb…
    - kritikus szakaszokat kezelő makrók
      - *OS_ENTER_CRITICAL()*
      - *OS_EXIT_CRITICAL()*
  - *OS_CPU_C.C*
    - implementációs source
    - HW időzítők
    - megszakítás kezelők
    - kontextus váltó függvények
    - task kezelő fv-ek
    - néhány extra fv. (az OS kódját tilos változtatni, ezért néhány dolgot itt kell implementálni)
  - *OC_CPU_A.ASM*
    - builded code
- sok platformra meg vannak írva, de ha nincs a miénkre, nekünk kell implementálni

Processzorfüggetlen fájlok

- a legtöbb funkció itt van
- fájlok:
  - *UCOS_II.H, UCOS_II.C*
    - OS szolgáltatásainak függvényei (hogy akarjuk-e őket használni, az a konfig fájlokban állítható)

- *OS_CORE.C*
  - a legalapvetőbb függvények implementációja
- *OS_TASK.C*
  - scheduler
  - task kezelő függvények
- *OS_TIME.C*
  - időzítés
- *OS_FLAG.C* – flagek implementációja
- *OS_SEM.C* – semaphore
- *OS_MUTEX.C* – mutexek
- *OS_MBOX.C* – msg box kezelés
- *OS_Q.C* – queue
- *OS_MEM.C* – memória management

Konfigurációs fájlok

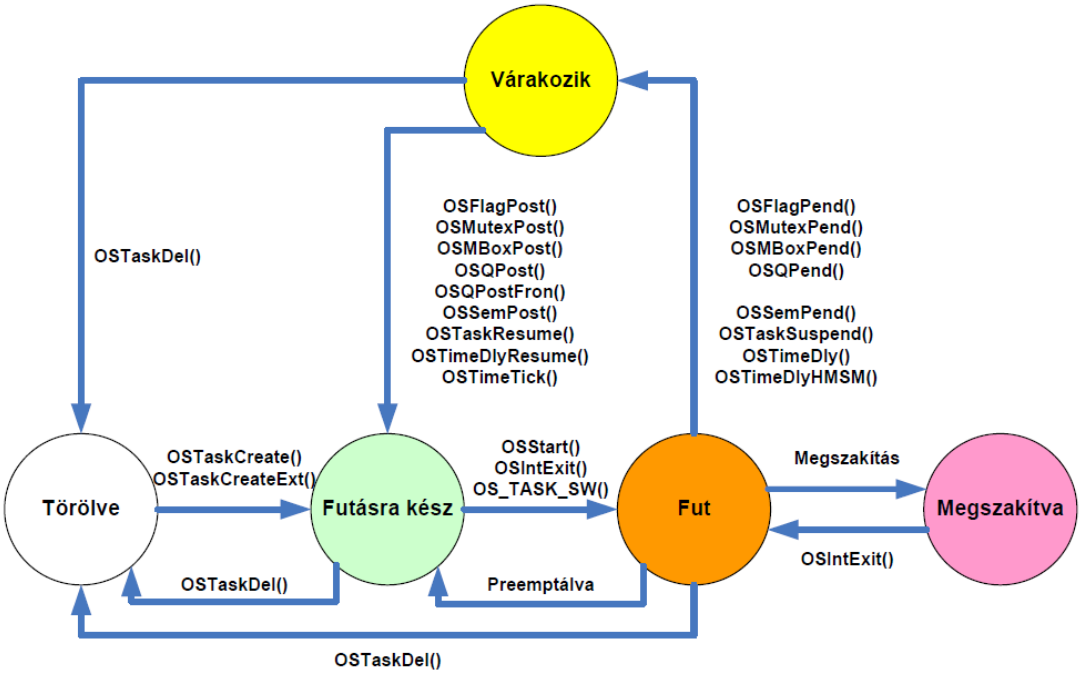
- master include fájl: *INCLUDES.H*
  - az OS minden sourcánál be van includeolva
  - globális hivatkozások, változók
  - regiszterek definíciója
  - könyvtári függvények leírói
- fő config: *OS_CFG.H*
  - alapszolgáltatások (pl. taskok kezelése nem kapcsolható ki)
  - összes kiegészítő szolgáltatás ki és be kapcsolható
  - tiltás lehet
    - globális szolgáltatás tiltás
    - egyes funkciók/függvények tiltása

3. Konfigurálás

- alapszabályok
  - minimális RAM igény
  - minimális kódmemória igény
  - az OS-re épülő alkalmazásnak is kell hagyni erőforrást
- néhány beállítás
  - *OS_MAX_EVENTS*
  - *OS_MAX_FLAGS*
  - *OS_MAX_QS*
  - *OS_MAX_TASKS*
  - *OS_LOWEST_PRI*
  - nem csak funkciókat, hanem azok egyes részeit is egyben lehet állítani
    - flagek
    - mutexek
    - semaphoreok
    - mailboxok
    - queuek
  - memória menedzsment ENABLE/DISABLE
  - memória menedzsment query enable/disable

4. Taszkok

Állapotok



Taskok kezelése

- create: *INT8U OSTaskCreate(void (*task) (void *pd), void *p_data, OS_STK *ptos, INT8U prio)*
  - ütemezőnél történő beregisztrálás
  - **pd* – task kódját tartalmazó fv. címe
  - **p_data* – fv. argumentumai
  - *prio* – **prioritás = a task azonosítója**
  - task-fv. prototípusa: void TaskFunction(void *pd);
- delete: *INT8U OSTaskDel(INT8U prio);*
  - a prioritásnak nem feltétlenül kell fixnek lennie egész élete során
- prioritás változtatása: *INT8U OSTaskChangePrio(INT7U oldprio, INT8U newprio);*
- task felfüggesztése, visszaállítása (ha pl. vezérlőpultról egy funkciót tiltanak, vagy engedélyeznek)
  - *INT8U OSTaskSuspend(INT8U prio);*
  - *INT8U OSTaskResunme(INT8U prio);*
- annyi task lehet, amennyi prioritást engedélyezünk
- taskok nyilvántartása: Task Control Block – TCB
  - a beregisztrált taszk pointereit tárolja
- egy task paraméterei
  - várakozó állapot? ha igen, tároljuk melyik objektumra várunk
  - ha várakozásban van, tárolva van, mennyi systick-ig kell még várni, amikor ez nulla, a task futásra kész
  - taszk státusza (fut, várakozik)
  - prioritása
  - törölhető-e a task

Ütemezés

- nagyon kritikus, hogy gyors legyen, és hatékony
- ütemezéshez: taszkok prioritásán alapuló nyilvántartási rendszer: azt rögzíti, hol van futásra kész task → néhány utasítással
  - be lehet rögzíteni új prioritást

  - el lehet távolítani prioritást
  - meg lehet keresni a legmagasabb prioritásút
- legyen 64 prioritási szintünk → 1 prioritást 6 biten tudunk ábrázolni
- az OS-II-ben létrejön egy 8x8-as mátrix a prioritások nyilvántartására (8 db 1byteos vektor) – *OsrdyTbl*
  - a prioritás felső 3 bitje a sorokat fogja jelölni
  - az alsó 3 bit az oszlopokat
- nyilván tartja mely sorokban van futásra kész task → minden sorhoz egy elem → *OSrdyGrp* vektor
- összesen 9 byte
- segédtábla a taskok futásra késszé tételéhez, illetve eltávolításához a táblázatból:
  - *OSMapTbl* – egy olyan 8 elemű vektorokból összeálló tömb, ahol az adott indexhez tartozó vektornak az annyiadik eleme 1, ahanyadik index
- task futásra késszé tétele: az *OSRdyTbl*-ben a megfelelő elemet 1-re kell írni, az *OSRdyGrp*-ben a megfelelő elemet 1-esre kell írni, ha még nem az (legyen a *prio: 12: 0x001 010b*)
  - *OSRdyGrp |= OSMapTbl[prio>>3];* – veszi a prioritás felső 3 bitjét (a táblázatban a sornak a számát), és az ennyiedik vektort az *OSMapTbl*-ből összevagyolja az *OSRdyGrp*-vel
    - *OSMapTbl[prio>>3] – OSMapTbl[001] – 0x0000 0010b*
  - *OSRdyTbl[prio>>3] |= OSMapTbl[prio & 0x07h]* – kimaszkolja a prio felső 3 bitjét, a maradék alapján veszi az annyiadik elemét, az *OSMapTbl*-nek, ahanyadik oszlopban van a prio → összevagyolja az *OSRdyTbl* megfelelő vektorával
- task eltávolítása a futásra készek közül
  - *if ((OSRdyTbl[prio>>3] &= ~OSMapTbl[prio & 0x07h]) == 0x00h*
    - *OSRdyGrp &= ~OSMabTbl[prio>>3];*
  - maszkolja a prio felső hármasát → ennyiedik elemét veszi az *OSMapTbl*-nek → veszi a negáltját → összeéseli az *OSRdyTbl* megfelelő vektorával → ha az eredmény pontosan 0 → törli a megfelelő elemet az *OSRdyGrp*-ből
- legmagasabb prioritású futásra kész task megtalálása:
  - segédtábla: *OSUnMapTbl[] = { ...}* – speckó tábla, ami segít megadni, hogy egy vektorban hanyadik a legkisebb helyiértékű valid bit (a sorszám alapján)
  - *Y = OSInMapTbl[OSRdyGrp*]; - megmondja melyik a legkisebb helyiértékű valid bit a RdyGroupban – megmondja melyik sorban van a legnagyobb prioritású task
  - *X = OSUnMapTbl[OSRdyTbl[Y]];* - amelyik sorban van a legkisebb helyiértékű valid bit a *RdyTbl*-ben, az ehhez tartozó sorban melyik a legkisebb helyiértékű valid bit
  - *prio = (Y<<3) + X;* - felső 3 bit + alsó 3 bit összerakása
- scheduler működése
  - megszakítások tiltása
  - gyorsan kikeressük a legnagyobb prioritású futásra kész taszkot az előzőekkel
  - amennyiben a futónál elvégezzük a taszkváltást, ha nem fut tovább a jelenlegi
  - engedélyezzük a megszakításokat

5. Megszakítások

- az OS-nek tudnia kell, hogy interruptban vagyunk → különben megszakítódhat
- külön fv. erre: *OSIntEnter()* és *OSIntExit()*

6. OS indulása

- többlépéses folyamat
  - *OSInit();*

- *OSTaskCreate();* // legalább egy taskot létre kell hozni
- *OSStart();*

7. Időzítés

- system tick késleltetés: *void OsTimeDly(INT16U tick)*
  - nem pontos időzítés → a megadott idő letelte után csak akkor fut futni, ha ez lesz a legmagasabb futásra kész task → csak a késeltetés minimum idejét tudjuk megadni, amin belül biztosan nem fut
- késleltetés SI mértékegységgel: *INT8U OSTimeDlyHMSM(INT8U hours, INT8U minutes, INT8U seconds, INT16U milli);*
- máshonnan feloldani a várakozásban levő feladatot: *INT8U OSTimeDlyResume(INT8U prio);*
- indítás óta eltelt idő: *INT32U OSTimeGet(void);*
- OS idejének beállítása: *void OSTimeSet(INT32U ticks);*

8. Szinkronizációs objektumok

Konfiguráció: ugyanolyan struktúrával (*OS_EVENT*)

- *OSEventType* – objektum típusa
- *OSEventTbl[OS_EVENT_TBL_SIZE]* - az ütemezéshez hasonlóan az adott objektumra várakozó taszkok prioritásainak tárolása
- *OSEventGrp* – az *OSEventTbl*-hez tartozó Grp vektor
- *OSEventCnt* – a szemaforok esetén a szemafor számlálója
- **OSEventPtr* – mail-boxes és msg queue-k esetén használatos – a bepakolt dolgok pointerét tartalmazza

Szemaforok

- létrehozása: *OS_EVENT *OSSemCreate(INT16U cnt);*
  - cnt – szemafor számlálójának értéke
- törlése: *OS_EVENT *OSSemDel(OS_EVENT *pevent, INT8U opt, INT8U *err);*
  - **pevent* – melyik szemafort akarjuk törölni
  - törölhető-e a szemafor, ha valaki várakozik rá
- bejelentkezés a várakozási sorba: *void OSSemPend(OS_EVENT *pevent, INT16U timeout, INT8U *err);*
- értékének vizsgálata: *INT16U OSSemAccept(OS_EVENT *pevent);*
- felszabadítás: *INT8U OSSemPost(OS_EVENT *pevent);*

Egyéb kommunikációs eszközök

- **Mutex** – prioritás inverzió ellen prioritás öröklést tartalmaz
- **Eseményjelző flag**
- **Postaláda**
- **Várakozási sor**

9. Taszkok használata

- taszkszervezési megoldások
  - egyszeres lefutású (single-shot)
    - lefut → szól az OS-nek, hogy done → törli önmagát
    - általában init feladatokra
  - végtelen ciklusú
    - általában ezt használjuk
    - célszerű olyan fv-hívásokat implementálni, amelyek várakozó állapotba viszik a taszkot → ha egy magas prioritású sosem kerül várakozásba, az alacsonyak kiéheznek

- végtelen ciklusú task implementálása
    - inicializáló task (magas prioritású)
        - HW init
        - létrehozza a szinkronizációs objektumokat
        - létrehozza a végtelen ciklusú taszkot
        - törli magát
    - main task
        - while(1)
        - megcsinálja a dolgát
        - elmegy aludni egy időre

3. Tétel

Mintapéldán keresztül mutassa be a prioritás inverzió jelenségét és a prioritás öröklés protokollt!

1. prioritás inverzió

Valamilyen alacsony prioritású taszk foglal egy olyan erőforrást, ami miatt a magasabb nem tud lefutni → teljesen összekeverednek a prioritások

Alaphelyzet

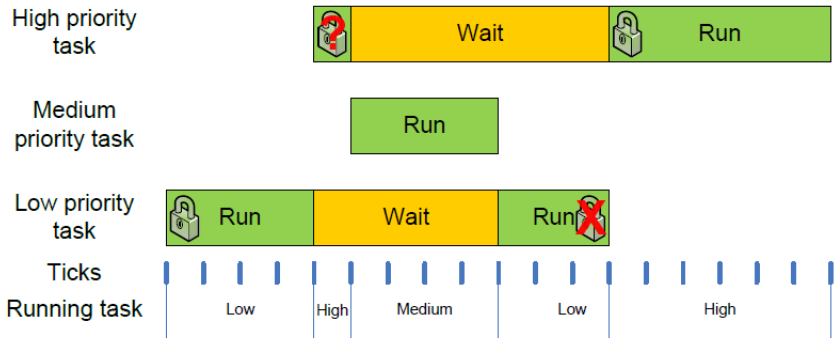
- alacsony prioritású taszk lefoglalja az A szemafort
- közben futásra kész lesz a legmagasabb prioritású taszk, ami futna, de szüksége volna A erőforrásra
- közben egy közepes prioritású taszk is futásra kész állapotba kerül

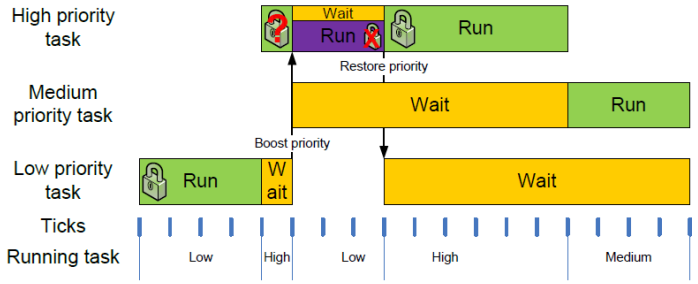
Végrehajtás

- a következő ütemezéskor a közepes prioritású fut le, mivel ő a legmagasabb futásra kész feladat
- ha végzett, a legmagasabb prioritású futásra kész taszk a legalacsonyabb prioritású lesz
- lefut →felszabadul A → lefut a legmagasabb prioritású
- eredmény: valójában a legmagasabb prioritású futott le leglassabban

2. megoldás: prioritás öröklés

Amikor egy magas prioritású taszk elkezd várakozni egy szemaforra, akkor végrehajtatja az ütemező az ehhez a szemaforhoz tartozó taszkokat (virtuálisan felemeli őket erre az időre magas prioritásúvá)





4. Tétel

Mutassa be a microCOS-II-en keresztül a leggyakrabban használt szinkronizációs objektumokat!

1. Szinkronizációs objektumok

Konfiguráció: ugyanolyan struktúrával (*OS_EVENT*)

- *OSEventType* – objektum típusa
- *OSEventTbl[OS_EVENT_TBL_SIZE]* - az ütemezéshez hasonlóan az adott objektumra várakozó taszkok prioritásainak tárolása
- *OSEventGrp* – az *OSEventTbl*-hez tartozó Grp vektor
- *OSEventCnt* – a szemaforok esetén a szemafor számlálója
- **OSEventPtr* – mail-boxes és msg queue-k esetén használatos – a bepakolt dolgok pointerét tartalmazza

2. Szemaforok

- létrehozása: *OS_EVENT *OSSemCreate(INT16U cnt);*
  - cnt – szemafor számlálójának értéke
- törlése: *OS_EVENT *OSSemDel(OS_EVENT *pevent, INT8U opt, INT8U *err);*
  - **pevent* – melyik szemafort akarjuk törölni
  - törölhető-e a szemafor, ha valaki várakozik rá
- bejelentkezés a várakozási sorba: *void OSSemPend(OS_EVENT *pevent, INT16U timeout, INT8U *err);*
- értékének vizsgálata: *INT16U OSSemAccept(OS_EVENT *pevent);*
- felszabadítás: *INT8U OSSemPost(OS_EVENT *pevent);*

3. Mutex

Olyan szemafor, ami védett prioritás inverzióval szemben

4. Event flags – eseményjelző flagek

- bitvektor → minden egyes bit egy eseményhez tartozik
- az események bekövetkeztekor a megfelelő bit bebillen
- ezekre várakoznak processzek/szálak
  - egyes bitekre
  - valamilyen kombinációra
- ha az eseménykezelő elvégezte dolgát, reseteli a megfelelő flaget

5. MailBox – Postaláda

- pointer méretű tároló a közös memóriaterületen
- több taszk helyezhet el benne üzeneteket
- több taszk várakozhat egy adott üzenetre
- adott üzenet kivétele:
  - legmagasabb prioritású taszk veheti ki
  - FIFO (ritkábban)

6. Várakozási sor

- többrekeszes postaláda – a rekeszekben mailboxok (pointerek) vannak
- tetszőleges számú taszk helyezhet el üzenetet
- üzenetek megnézése, kivétele

- csak az aktuálisan legmagasabb prioritású taszk nézheti meg az üzeneteket
- FIFO
- LIFO

7. Pipe – csővezeték

- üzenetek mérete nem fix
- egymás után bele lehet rakni pointereket, de akár tetszőleges objektumokat is

5. tétel

Milyen komponensekből épül fel a GNU/Linux rendszer? Ismertesse az elindulás folyamatát!

1. Fő alapkövek:

- Kernel
- Fejlesztői könyvtárak
- Segédprogramok
- Shell(ek)
- Alacsony szintű grafika (framebuffer)
  - directfb, SDL
- X Window (Xorg, KDrive)
  - widget library: Xib, Qt, Gtk++
  - Desktop Environment: KDE, GNOME

2. Linux betöltése

Általános recept:

bootloader (x86/64: BIOS, ARM: BootStarp) → rendszermag betöltés → rendszermag belépési pontjára ugrás → inicializáció (hardverek, fájlrendszerek stb.) →bejelentkezés →mindenki boldog ☺

Linux esetében:

- Bootloader: LILO (The Linux Loader), Grub, uBoot
  - megkeresi és futtatja a kernelbetöltőt
- Kernelbetöltő
  - Kernel kicsomagolása (általában tömörített formában a lemezen, önkitömörítő)
  - Belépési pontra ugrása
- Kernel inicializálása
  - HW init
  - start_kernel() hívása
    - 0-s azonosítójú processz
    - inicializálja a kernel alrészeit
    - elindítja az init kernelszálat, majd üresjáratba kezd → szerepe elhanyagolható
- Init kernelszál
  - mounting
  - sysinit (dist. dependent: /etc/init/, /bin/init/, /sbin/init/)
  - konfigurációs állomán(/etc/inittab/) alapján új processzeket hoz létre,, amik továbbiakat szülnek
    - pl.: getty processz indítása → login processz indítása

Initrd (újabban: initramfs)

- tömörített állományrendszer: induláskor a kernel kitömörítődik egy ramdiskre → képes állományrendszerként használni
- Miért?
  - kis memóriában futó Linuxokra
    - normál rendszer feltelepítése
    - megjavítása
    - partícióműveletek
  - Live rendszerek
    - debug
  - telepített rendszerek

- boothoz szükséges kernel modulok betöltésére

SysV init

- Kernel init a root fájlrendszer csatolása után
- feladati (/etc/inittab alapján)
  - USB eszközök inicializálása
  - állományrendszerek ellenőrzése + felcsatolása
  - szolgáltatások elindítása
    - futási szintek létrehozása
    - virtuális konzolok
    - X felület

Futási szintek

0. Rendszer leállítása
1. Egy felhasználós mód
   a. minden hálózati és felhasználói szolgáltatás leáll
   b. felhasználók nem léphetnek be
   c. csak a rendszergazda kap egy shellt a konzolon
2. Több felhasználós mód hálózat nélkül
3. Több felhasználós mód
4. Nem használt
5. Több felhasználós mód, X felülettel
6. Restart
7. (általában 9-ig) extra futási szintek (tradícionális UNIX esetén csak 6-ig)

- 0,1,6 szintek rendszer számára fenntartottak
- többi: ajánlott default – de szabadon konfigurálható
- futási szintekhez tartozó scriptek:
  - /etc/rc.d/ecN.d/ (N a futási szint száma)
  - futási szint init: /etc/rc.d/rc.sysinit + /etc/rc.d/rc.local

6. Tétel

Ismertesse a Linux rendszerek által használt állományrendszereket!

1. Minix

- Linux fejlesztése eredetileg Minix OS-en történt →ennek az állományrendszerét használták
- túl sok korlátozás → kikerült

2. Ext fájlrendszerek

- ext, ext2, ext3, ext4
- ext teljesítményproblémáinak kezelése: ext2
- ext2 + journaling = ext3
- ext3 továbbfejlesztése: ext4

3. Journaling fájlrendszerek

- **miért:** állományok írása összetett művelet → áramkimaradás/összeomlás stb. → az írt állományok köztes állapotban maradnak → következő bootnál a rendszernek végig kell néznie a teljes állományrendszert, és megkeresni a hibákat, és megpróbálni kijavítani → sok idő + ha nem sikeres a visszaállítás →fucked
- ***journaling:*** naplózás külön területre
  - módosítás előtt feljegyzi mit fog csinálni
  - összeomlás után:
    - tudjuk mik voltak utoljára módosítva
    - tudjuk mit akartunk módosítani
    - visszaállítható vagy a módosított állapot, vagy az eredeti
- ***journaling fájlrendszerek:***
  - **JFS** (IBM Enterprise rendszerekben, Journaling File System)
  - **XFS** (Silicon Graphics)
  - **ext3** (ext2 + naplózás)
  - **ReiserFS**
  - **JFFS** (Journaling Flash File System)
    - Flash eszközökhöz (NOR)
    - csak a módosításokat írja fel
    - szemétgyűjtés: ha kevés a hely, összeszedi a darabokat, és egyesíti a fájlokat → sok írás, de még mindig kevesebb
    - ☹:
      - külön nyilván kell tartani a darabkák helyeit
      - induláskor be kell olvasni a nyilvántartást → hosszabb mount
  - **JFFS2**
    - NAND Flash eszközökhöz
    - Hard linkek kezelése
    - Tömörítés támogatása
    - Jobb teljesítmény
    - Tárterület kezelése más: nem állományként kezeli az eszközt, hanem blokkokra osztja, és mindig egyet tölt fel → amíg az meg nem telik, nem vált át másikra
      - tiszta blokk – csak aktuális infót tartalmaz
      - piszkos blokk – aktuális és elavult információkkal
      - szabad blokk – üres

        - szemétgyűjtő: feladata minél több tiszta és szabad blokk létrehozása
        - statikus állományokra jobb, mint JFFS
        - ☹:
            - lassú felcsatolás
            - nagyobb memória igény
    - **UBIFS** (Unsorted Block Image File System)
        - JFFS2 + gyorsítótár
        - pesszimista algoritmus a szabad terület megbecsléséhez (gyorsabb)
        - gyorsabb
            - felcsatolás
            - írás /olvasás
        - jobb visszaállíthatóság
- ☹ :
    - több írás → idő + elhasználódás (SSD didn't liked this)

4. Speciális állományrendszer típusok (virtuálisak)

- ***Proc***
    - /proc/
    - kernel belső állapotáról ad infót
- ***Sysfs***
    - /sys/
    - rendszer eszközei fa struktúrában
    - eszköz állapotáról ad infót
    - eszközvezérlő beállításai
- ***Tmpsf***
    - ramdisk
    - gyakran használjuk átmenetei állományok tárolására, folyamatok közötti kommunikációra ← gyors, nem terheli a háttértárat

5. CD/DVD álományrendszere

- CD – ***iso9660***
- DVD – ***udf***

6. Hálózati állományrendszerek

- ***NFS***
    - Natív Unix
    - Network Filesystem
- ***SMB***
    - MS által használt
- ***NCPFS***
    - Novell Netware szerverek által használt

7. Tétel

Milyen állománytípusokat különböztetünk meg a Linux állományrendszerében? Az állományoknak milyen jogosultságokat állíthatunk be?

1. Állománytípusok

- **egyszerű állomány**
  - a Linux egyszerűen byte-ok véletlenszerűen címezhető szekvenciájának tekinti
    - tökmindegy, h az bájtok sorozata, text fájl, program stb.
  - nem látjuk a tároló tulajdonságait
- **könyvtár**
  - szintén állomány – speciális
  - minden olyan infót tartalmaz, amire a rendszernek szüksége van a hozzá tartozó állományok elérésére
  - ext fájlrendszerek esetén tartalmazza
    - állományok neve
    - hozzájuk tartozó i-node indexek
- **eszközök**
  - */dev/*
  - a legtöbb fizikai eszköz állományként a */dev/*-ben található meg
  - eszközökre ezekkel hivatkozhatunk
    - írhatunk beléjük (pl. audio eszköz → a kernel úgy értelmezi, mint ha le szeretnénk játszani a bájtsorozatot)
    - olvashatunk róla (pl. audio eszköz → a kernel úgy értelmezi, hogy be akarjuk digitalizálni a felvett hangot)
  - ***eszköztípusok***
    - blokk
      - nem lehet olvasni bájtonként, csak bájtok blokkjaiként
      - pl. háttértárak
    - karakter
      - lehet bájtonként írni és olvasni
      - pl. nyomtató, audio eszköz, egér, billentyűzet
- **szimbolikus link (symlink)**
  - speciális állomány, amely egy másik állomány elérési információit tartalmazza
  - lekövetés: megnyitjuk →kernel beolvassa az értékét → megnyitja a hivatkozott állományt
- **csővezeték és socket állományok**
  - kommunikációs csatorna: IPC mechanizmus (Inter Process Communication)

2. Jogosultságok

- **id-k**
  - uid – user id
  - gid – group id
  - felhasználó azonosítója, és csoport azonosítója: /etc/passwd
  - többi csoportazonosító: /etc/group/
  - root: supeuser – mindenki felett áll
    - nem vonatkoznak rá a beállított hozzáférési jogok
    - uuid-je: 0

- **védelmi korlátozások:**
  - a tulajdonos az állomány létrehozója (chown paranccsal át lehet ezt ruházni másra)
  - az állományhoz tartozó csoport a tulajdonos alapértelmezett csoportja (módosítás: chgrp)
- **a hozzáféréseknél 3 felhasználó-típus van:**
  - owner (uid azonos)
  - group-member (uid nem azonos, de gid igen)
  - others (sem uid, sem gid nem azonos)
- **listázás:** ls –l → 9+1 karakter
  - 1.: állomány típusa
    - b – blokk orientált állomány
    - c – karakter-orientált állomány
    - d – katalógus (directory)
    - l – symlink – esetében a többi karakternek nincs értelme, a cél állomány jogai a mérvadóak
    - - – egyszerű állomány
  - 2-4: owner jogai
    - r : read – megtekintés
    - w: write – megváltoztathat
    - x: execute – futtatási jog
      - katalógus esetén ennek nincs értelme → kereshetőséget jelent (van –e joga elérni)
      - nem jelenti, hogy tényleg futtatható kód
    - 5-7: group members jogai
  - 8-10: others jogai
- **x helyén s → setuid, setgid mód**
  - owner hármasban: setuid
  - group hármasban: setgid
  - a felhasználó által futtatott program végrehajthatja, amit a programállomány tulajdonosa
- **chmod** <mód> <fájlnév>
  - egész szám – 2-es számrendszerbeli megfelelőjében látható, mely jogokat kell eltávolítani
  - oktális szám
  - szimbolikus karakterek
- jogok elvétele: **umask** <mód inverze>

3. Jogok megváltoztatása

- **szimbolikus karakterekkel**
  - chmod <felh. csoport><opció><jogok> <fájlnév>
  - pl.:
    - -rwxrwxrwx 1 drkrieger drkrieger 5 Jun 7 23:17 muhaha.txt
    - chmod –go-wx muhaha.txt
    - -rwxr--r-- 1 drkrieger drkrieger 5 Jun 7 23:17 muhaha.txt
  - felh. csop.:
    - u – tulaj
    - g - group
    - o – others
    - a – mindenki
  - opció
    - +

        - -
        - = : pontosan ezt az értéket akarom
    - jogok:
        - rwx
        - X: végrehajtási jog, de csak akkor, ha az állomány katalógus, vagy van már másik x bitje
        - s
        - t: sticky bit
            - mai linux nem értelmezi, régi Unixban használták
            - Könyvtárak esetén a jelentése, hogy a benne található állományokat csak a tulajdonos, vagy a root nevezheti át, vagy törölheti (csak az, aki létrehozta → rendszergazda szivató ☺)
            - /tmp/ használja jellemzően
        - u, g, o: a tulaj/csoport/többiek mezőt az eredeti módból veszi
- Oktális számokkal
    - abszolút szám, aminek bináris megfelelője adja a jogosultság bitmaszkját
    - 4000 – setuid
    - 2000 – setgid
    - 1000 – sticky bit
    - 0400 – u+r
    - 0200 – u+w
    - 0100 – u+w
    - 0040 – g+r
    - 0020 – g+w
    - 0010 – g+x
    - 0004 – o+r
    - 0002 – o+w
    - 0001 – o+x
    - 777 – bárki bármit tehet (111 111 111)
    - 700 – csak a tulaj rwx, mások semmit (111 000 000)
    - 666 – bárki írhat, olvashat, de senki nem hajthatja végre (110 110 110)

8. Tétel

Linux parancsértelmezők (shell) milyen szolgáltatásokat nyújtanak?

1. Shell – parancsértelmező

- ugyanolyan app, mint a többi
- lecserélhető (chsh)
- default: bash (Bourne again shell / Born again shell)

2. Parancssor értelmezés

- formátum: *parancs arg1 arg2 arg3*
- parancs értelmezése: PATH-on levő könyvtárakban keresi a shell parancsokat → végrehajtja
- argumentumok:
  - egyszerű karakterlánc
  - <állománynév>
  - karakterlánc helyettesítő karakterekkel
  - 'parancshelettesítés'
- a shell a command line-ról string tömböt kap

3. A shell beépített parancsai:

Mini-shellek:

- Minimális felület, ami tartalmazza az alap shell parancsokat
- Azok vannak implementálva, amik probléma esetén is biztosan működnek
- pl.: shash (Stand-Alone Shell)

Állománynév helyettesítés

- * - nulla vagy tetszőleges számú tetszőleges karakter
- *?* – pontosan egy tetszőleges karakter
- *[abc]* – a, b vagy c karakterek egyike
- *[m-n]* – a megadott intervallumból egy karakter

Stdin, stdout átirányítás

Egy shell parancsnak 3 kimenete van:

- standard input
- standard output
- error output

Átirányítás:

- *parancs > kimenet* – létrehozza a kimenet állomány, és beleírja a parancs kimenetét (vagy ha már létezik, felülcsapja)
- *parancs >> kimenet* – append
- *parancs 2> kimenet* – stderr kimenet belepakolása a kimenetbe
- *parancs < input* – az inputot a parancs nem a command line-ról várja, hanem az input fájlból

Csővezeték (Pipe)

- *ls | grep „minta" | sort | more*
  - ls-ből kiszűri a grep a mintát, sort sorbarendezi, more oldalakra darabolja
- nem kell ideiglenes fájl, ahova az egyik ír, másik olvas
- PIPE gyors, mivel csak a memóriát használja

- az eredmény a cső végén már azelőtt megjelenhez, mielőtt az első program végzett volna

Parancshelyettesítés

- paraméter, vagy parancs fájlban is lehet
- *du $(cat parameters.txt)* vagy du 'cat paramterts.txt'
    - megmondja a fájlok méretét, amit előre eltároltunk a parameters.txt-ben

Parancssorozatok

- *date; ls* – egymás tán hajtódik végre
- *parancs1 || parancs2* – felt. parancs: a parancs2 akkor hajtódik végre, ha parancs1 hibával állt meg (vagy az első, vagy a második)
- *parancs1 && parancs2* – felt. parancs: parancs4 akkor hajtódik végre, ha parancs3 igaz megállási státusszal állt meg (első és második)

Szinkron és aszinkron folyamatok

- *parancs &*
- a shell azonnal visszaadja a promtot, mielőtt befejezte volna a parancs a végrehajtást
- ***note***: a stdoutot át érdemes irányítani fájlba, különben a consolra pakolja ki az eredményt, ami zavaró lehet

Csoportosítás

zárójellel

4. Változók kezelése

```
    kutya = ugat
    echo $kutya
ugat
    vagy:
    kati = zsuzsi
    echo ${kati}ka
    zsuszika
```

- olyan mint a *#define:* pontosan azt tárolja, amit beleírtunk (szám, karaktersor, rövid fv.)
- *unset* parancs: változó törlése
- *set* parancs: kiírja az összes definiált változót
- környezeti változók nem elérhetőek kapásból a programok számára → ki kell exportálni a változók közé a kívánt környezeti változót, és azt már látni fogja → *export* parancs

5. Aritmetikai kiértékelés

```
a=$((5*2))
a=$((a*3))
```

6. Parancsállományok – shell scriptek – .sh

- interaktív parancsértelmezés
- futtatás:
    - futási jog nélkül: *bash shell_script.sh argumentumok*
    - futási joggal*: bash shell_script.sh argumentumok*

Feltételek kiértékelése

test parancs

- stringek vizsgálata
    - *test s* : true, if s string is not null

  - *test –z s*: true, is s string's length is not zero
  - *= ; !=*
- numerikus vizsgálat
  - *-eq, -ne*: equal, not equal
  - *–lt, -le, -gt, -ge*: lesss than, less or equal, …
- állományok tulajdonságainak ellenőrzése
  - *–f f*: f egy létező állomány/könyvtár?
  - –r f, -w f, -d *f*: f is readable, writable, directory?
  - *–s f*: f létezik, és nem nulla hosszúságú
  - *–t fd:* ha fd egy megnyitott állomány leírója és az egyben egy terminál
- test elhagyható, ha a feltételt *[]* közé írjuk
- ha a vizsgált paraméter space-t tartalmaz: *{}*
- shell tördelésének és állománynév értelmezésének letiltása: *[[adf]]*

Vezérlési szerkezetek:

```
if [$"1" eq 1]
        then echo 'egy'
        else echo 'nem egy'
fi
```

```
for i in w1 w2
        do parancsok
done

for ((i=1; i<=5; i++))
        do echo $"i"
done
```

```
while parancs
        do parancsok
done

while ((i<=5))
        do i=$((i+1))
done
```

```
until test $"i" –le 5
        do I = 'expr $"i" +
1
done

until ((i>5))
        do i=$((i+1))
done
```

```
case $1 in
echo "egy";;
echo "ketto";;
*)    echo "sok"
esac
```

Shell scipt:

```
#!/bin/bash

# file számláló

n=0
for i in  *
do
        if [-f $i]
        then n='expr $n + 1
        fir
done
```

7. Folyamatok monitorozása: *ps*

- Attribútumok:
  - *-A*: összes folyamat
  - *-N*: negálja a folyamatválasztást
  - *a*: terminálhoz kapcsolódó összes folyamat (shellek kivételével)

  - *r*: csak a futó folyamatok
  - *-u user*: userhez tartozó folyamatok
- formzás (kötőjellel és anélkül is ok)
  - *–f*: teljes lista
  - *–j*: job control info
  - *–l*: hosszú fájlformátum
  - *–o* formátum: saját definiált formátum

Hosszú folyamatok: nohup parancs &

- megmondja, hogy ne vegye figyelembe, ha a felhasználó kijelentkezik
- semmiképpen sem az stdout-ot használja (ha nem irányítjuk kimenetbe, akkor nohup.out)

Kommunikáció a folyamatokkal, megszüntetés

- *kill [-szignál] folyamatszál*
- szignál:
  - *SIGHUP(1)* – akkor generálódik, ha kilépünk, miközben a folyamat még fut
    - def: akkor kapja a folyamat, ha a szülője leállt
  - *SIGINT(2)* – interrupt karakter (*Ctrl + C*)
    - def: *ctrl+c*
  - *SIGQUIT(3)* – legerősebb szignál – ezt nem hagyhatja figyelmen kívül a program (nem javasolt)
    - *def*: *ctrl + *
  - *SIGTERM(15)* – Folyamatok leállítása. Alapértelmezett érték

Folyamat vezérlése

- *Ctrl+z:* folyamat felfüggesztése (Pause) → dob egy job számot, amivel hivatkozhatunk rá később
- *bg <job>*: folytatódjon háttér folyamatként
- *fg <job>:* foreground run
- *jobs*: felfüggesztett folyamatok listázása
- *kill [-szignál] %job*

Prioritás állítás

- *renice <prioritás> <processid>*
- *renice <prioritás> -p pid –g pgrp –u user*
  - *–p pid*: processidvel azonosítás
  - *–g pgr*p: egy folyamat csoportra hivatkozás
  - *–u user*: felhasználó összes folyamata manipulálása
- prioritás szintek: 0-20
  - 0: default prioritás → a pozitív egyre előzékenyebb → a 20-as csak akkor fut, ha más folyamatnak már nincs szüksége a processzorra
- root állíthat -20-ig – fontos processzektől szipkázhatja el az erőforrást

9. Tétel

Ismertesse a Yocto Project fejlesztői keretrendszer funkcióját, felépítését, működését!

1. Keresztplatformos beágyazott rendszer készítése

- ha a rendszer elkészítéséhez használt és a cél platform jelentősen eltér egymástól → és általában így van
- keresztfordítós megoldás akkor is jó, ha azonos a két platform: az alkalmazások egy „virtuális" környezetben nőjjenek fel (csak azt használják, amit megadunk nekik, így nem matatva a rendszerben)
- **szükséges cuccok:**
  - platformfüggő kernel-header állományok
  - binutils csomag lefordítása
  - C fordító (pl. gcc-vel glibc fordítása → glibc-vel gcc teljes lefordítása)
  - fejlesztői könyvtárak fordítása
- **automatizált eszközök:**
  - ***Buildroot***
    - makefile-ok és patch-ek gyűjteménye (uClibc-t használ)
    - kicsit nehézkes
  - ***OpenEmbedded***
    - BitBake receptek: fordítási műveletek + csomagok előállításának módjai
    - számos architektúra támogatsása
    - könnyen bővíthető
  - ***YoctoProject***
    - OpenEmbedded alapú
    - egyszerűsíti a beágyazott rendszer készítést, könnyíti a fejlesztést különböző hardverekre
    - jól dokumentált
    - Eclipse-be integrálódik
    - szimulációs környezet és SDK-t is lehet vele készíteni
  - ***Scratbox***
    - OS emulálás

2. Yocto Project felépítése

- *bitbake* – a program, ami a recepteket tartalmazza
- *build* – a munkakönyvtár
- *documentation*
- *Rétegek: meta, meta-yocto, meta-yocto-bsp*
  - *classes* – BBClass állományok, amelyek receptekben felhasználható műveleteket implementálnak
  - *conf* – Konfigurációs állományok
    - bitbake.conf – BB fő konfigja
    - distro - Disztribúciós
    - machine – Architektúrák
  - *files* – Egyéb állományok
  - *recipes*
    - Receptek a különböző SW verziókhoz
    - Verzió független állományok
    - Verziófüggő állományok (patch, config)
  - *site* – OS függő beállítások
- *scripts* – automatizáló szkriptek

Saját rétegek: yocto-bsp create <bsp-name> <krach>

- új réteg neve: bsp-name
- krach – kernel architektúra
- BSP jellegű rétegek, de ebből később törölhetőek a HW konfig részek, vagy asjátot is létrehozhatunk
- *conf/layer.conf* – réteg beállításai
  - réteg neve
  - receptek elérhetősége
  - réteg prioritása
- új architektúra konfig: *conf/machine/*
- saját receptek*: recipes-*/*

3. Yocto Valósidejű rendszer készítésére

- csak a kernel más + néhány SW eszközt hozzáadhatunk
- *linux-yocto-repository repo:*
  - itt találhatóak *preempt-rt* ágakat → ezek megfelelő felhasználásával legenerálható a kernel →recept kell
  - sok architektúrához már készítettek receptet → meg kell keresni a Yocto rendszerébe

4. Beágyazott rendszer telepítése

- **Build eredménye:**
  - deploy könyvtárba ipk, rpm, deb formátumú csomagokat generál
  - ha egyébb fejlesztői eszközöket is előállítottunk, azok csomagjai az sdk könyvtárba kerülnek
- **Telepítés A:**
  - háttértár partícionálása (általában 1 partíció)
  - állományrendszer elkészítése a partíción
  - partíció felcsatolása
  - a buildelt csomagot ki kell tömöríteni a partícióra
  - egyéb konfig, ha szükséges
- **Telepítés B:** egyes architektúrákra már készen van komplett lemezképben a rendszer
  - felmásol: ☺
- **Kernel telepítése**
  - *X86:*
    - kernel berakása a partícióra
    - bootmanager telepítése, elkészítése → az eszköz bootolható, be tudja tölteni a kernelt
  - *ARM, MIPS*
    - belső flash
      - kernel áttolása az eszköz RAMjába (vannak erre programok)
      - flash-ben megfelelő mennyiségű hely felszabadítása
      - kernel bepakol a flash-be
      - boot opciók beállítása
    - SD/MC/CF kártyán particionálás, majd rápakolás
  - *architektúra függő lépések:* gyártói dokumentáció

5. Keresztfordítás (kézi)

- ***szükséges programok, és keresztfordí***tó: sysroots könyvtárban
  - keresztfordítók

- fejlesztői könyvtárak
- headerek
- segédprogramok
- konfigurációs állományok
  - egyszerűbb paramétereés: pkg-config program

6. SDK készítése

- ezt telepítve a fejlesztők gépére képesek lesznek a keresztfordítások elvégzésére a teljes Yocto Project nélkül
- tartalmazza
  - keresztforító
  - szükséges fejlesztői könyvtárak
  - environment setup

7. Virtuális gép generálása

- hasznos eszköz a fejlesztés során
- **virtuális gép imaget** lehet készíteni: x86-ost emuláls → a generálás várhatóan a csomagok újrafordításával jár
- **QEMU** – x86-os gép szimulálása a CPU architekturális eltérése nélkül
- **vmdk** állomány generálása – VMWare Player

10. Tétel

Ismertesse a Makefile szintaktikai elemeit!

1. GNU Compiler Collection

- Fajtái:
  - **GCC** – C
  - **G++** – c++, Objective C, Fortran
  - **GCJ** – Java
  - **GNAT** – Ada
- gyakran használt paraméterek
  - *–o* fájlnév
  - *–c*
  - *–D definíció=x :* definíció makró definiálása x-el
  - *–I könyvtárnév:* új header könyvtár elérési útvonalának megadása
  - *–L könyvtárnév*: source könyvtár hozzáadása
  - *–static*: default dinamikus linkelés helyett statikus használata
  - hibakeresés:
    - *-g:* szabványos hibainfók
    - *-ggdb*: olyan infókat helyezzen el a programban, amit csak a gdb értelmez
    - *-ggdb N*: debug info szint:
      - 0: nincs debug info
      - 1. default
      - 2: extra infók
  - *–O, -O N*: optimalizáció (szintek 0-3 között, 2 default)
  - *–Wall*: az összes warningot dobja consolera

2. Make

Általános szintaktika

- új sorban folytatása az előzőnek: \
  - mert a make minden sort egyesével hajt végre egy-egy subshellben → jelölni kell, ha még nincs vége a sornak
- hívás
  - *make*: első rule
  - *make rule:* argumentumban megadott rule
- Hamis tárgy: *.PHONY: asdf*
  - hogy a make ne értelmezze végrehajtandó szabályként az megadott kulcsszavakat
  - ilyen pl. a *clean*, hogy ha van clean fájlunk
- zárójelezés: *()* és *{}* egyenértékű

Megjegyzések

- # karakterrel kell kezdődnie
- ha nem egy rule-ban van, akkor a make figyelmen kívül hagyja

Explicit rules

- meghatározza, hogy mikor és hogyan kell újrafordítani egy vagy több állományt
- *target : dependency list*
- a targetet akkor kell újrafordítani, ha

- a target nem létezik
- a dependency list-ben valami módosult (az időbéjeg későbbi, mint a targeté)

- commands, receipt
  - TAB-al kell kezdeni, nem szóközzel!!

Változódefiníciók

- VÁLTOZÓ = ÉRTÉK
- rá való hivatkozás: $(VÁLTOZÓ) vagy $({VÁLTOZÓ}
- a változó nevében nem lehet spec karakter (:, #, =), üres mező
  - gyakorlatban csak betűk számok és _ lehet benne
- saját változónevek: inkább kisbetű, nagybetűket rendszerváltozóknak meghagyják
- olyan, mint a #define (bármi belekerülhet), más objektumok, rövid makrók stb.
- append, behelyettesítés: NEM SZEKVENCIÁLIS
- speciális esetek:
  - egyszerű kiértékelés helyben: := használatával
  - += hozzáfűzés
  - ?= : feltételes értékadás – a változónak csak akkor adunk értéket, ha még nincs definiálva
    - a = egy
    - a ?= kettő
    - eredmény: egy
  - többsoros változók:
    - define változó =
    - muhaha
    - hehe
    - endef
  - változó törlése: undefine változó
- változó hivatkozások
  - a változóra hivatkozás során az értékét módosítva helyettesítsünk be az adott helyre
  - konverziós szabály (az értéket szavanként kezeljük)
  - pl.:

    srcs = elso.c masodik.c

    objs := $(srcs:.c=.o)
  - számított változónevek:

    src_1 := elso.c

    src_2 = masodik.c

    objs := $(src_$(a):.c=.o)

    az a változó érékétől függően vagy az src_1 vagy az src_2 értékéből állítjuk elő az objs értékét, majd c-ről ó-ra cseréljük a kiterjesztését
- automatikus változók
  - $@ - target
  - $< – dependency list első eleme
  - $? – teljes dependency list-ből azok, akik módosítva lettek
  - $^ – teljes dependency list (ha valami többször szerepel, csak egyszer helyettesíti be)
  - $+ – teljes dependency list
  - $* – target kiterjesztés nélkül

Többszörös target

- targetek listáját is meg lehet adni a target mezőben
- külön-külön végrehajtódik a szabály minden egyes targetre
- a recept lehet hasonló, de a végrehajtott parancsok különbözőek: automatikus változók

Többszörös cél:

```
debug_flags := -g3 –o0
release_flags := -g0 –o3
prgs := debug release

all : $(prgs)

*PHONY: all

$(prgs) : hello.c
        gcc $($(@)_flags) –o $@ $<
```

A make parancs kiadása után két fordítás történik. Létrejön egy debug és egy release állomány eltérő paraméterezéssel

```
gcc –g3 –o0 –o debug hello.c
gcc –g0 –o3 –o release hello.c
```

Minta szabályok

- s statikus szabályokkal ellentétben ez egy alapértelmezett szabályt ad. De ha a makefileban van más specifikus szabály, akkor az fog végrehajtódni
- *TAGET : TARGET-TEMPLATE : DEPENDENCY TEMPLATE*
  - *$(objs) : $.o: $.c*
  - *iterál*
- *TARGET-TEMPLATE : DEP-TEMPLATE*
  - *$.o : $.c*
    - minden *.o-ra a következő szabály hajtódjon végre

Klasszikus ragozási szabályok

- *.c .o: \ receipt \ .SUFFIXES: .c .o*
- ugyanazt csinálja, mint az előző
- .SUFFIXES: speciális tárgy, ami után fel kell sorolnunk párinként melyik kiterjesztésből áll elő egy másik
- nem használjuk

Speciális tárgyak

- *.PHONY* – hamis célok kivételei
- *.SUFFIXES* – a dependency list-ben állománynév kiterjesztések vannak, amelyeket a make a ragozási szabályok keresésénél használ fel
- *.SILENT* – a dependency listbe írt parancsok eredményeit nem teszi ki a kimenetre
- *.ONESHELL* – a parancsok ne subshellekben hívódjanak

Direktívák

- a make „prerocesszora”
- a következő műveleteket definiálhatják
  - más makefile beolvasása
    - include fáljnév
  - a makefile bizonyos részeinek használának engedélyezése, tiltása
    - ifeq ($(változó1) , $(változó2))
    - else
    - endif

3. Make alternatívák

Komplexebb feladatok ellátására, sokszor make alapú programok

Autotools

- make + extra szabálylista + függőség ellenőrzése
- célja a forráskódok hordozhatósága platformok között
- fő részei: *Autoconf, Automake, Libtool*
- a fordításhoz nincs szükség a teljes Autotoolshoz, elég ha a configure script megvan, ami definiálja a műveleteket

CMake

- fordítás
- VS projekt állomány előállítása
- gondoskodik a szükséges állományok legenerálásáról is

qmake

- Qt része, azok fordítását támogatja
- makefile-t generál

SCons

- szoftver konstrukciós eszköz
- függőségek vizsgálata
- platform adaptáció támogatása
- Python alapú

11. Tétel

Ismertesse a C/C++ Linux alkalmazások hibakeresésének főbb eszközeit

1. gdb

Legfőbb hibakereső Linuxra (C-re, más nyelveket csak részlegesen támogat)

Funkciói:

- Program elindítása
- megállítás meghatározott feltételek esetén
- a megállításkori állapot vizsgálata
- a program egyes részeinek megváltoztatása, és a változtatás hatásának vizsgálata

Működése

- program fordítása *–g* kapcsolóval
- indítás
  - gdb program
  - argumentumok
    - CORE
    - process id
- *set width=70* : program kimenetének szélessége

Töréspontok

- ***egyszerű töréspont***
  - *break* függvény: egy adott fv. belépési pontjánál
  - *+- OFFSET*: az aktuális pozíciótól számított sorokkal odébb
  - *fájlnév:sorszám* – adott fájl adott sorába
  - **CÍM* – egy adott címen
  - argumentum nélkül – az aktuális stack fram következő utasítására
  - *... if COND* – feltételhez kötött breakpoint
- ***watchpoint***
  - speciális töréspont, ami akkor állítja meg a programot, amikor az adott kifejezés változik (nem kell helyet megadni)
  - *watch KIF* – a program megáll, ha KIF-t írja valaki (módosul)
  - *rwatch KIF* – a program megáll, ha KIF-t olvassa valaki
  - *awatch KIF* – a program megáll, ha KIF-t olvassák/írják
- ***catch ESEMÉNY***
  - megállás esemény bekövetkezése esetén
  - az esemény lehet:
    - *throw* – C++ exception keletkezése
    - *catch* – C++ exception kezelése
    - *exec* – az exec fv. meghívásakor
    - *fork* – fork fv. meghívásakor
    - *vfork* – vfork meghívásakor
    - *load KÖNYVTÁRNÉV* – adott könyvtár betöltésekor
    - *unload KÖNYVTÁRNÉV* – könyvtár eltávolításakor
- run: futtatás

Töréspont elérése utáni lehetőségek

- *run (r):* folyamat elindítása

- *next (n):* next line
- *step (s):* step into a function
- *print (p):* kiírja egy adott változó értékét
- *backtree (bt):* A stack keretek megjelenítése
- *list (l):* a forráskód kilistázása az aktuális pozíció környékén
- *continue (c):* a program futásának folytatása
- *Ctr+D:* program leállítása
- *quit (q):* kilépés

2. ElectriFence

Memóriaszivárgás, vagy túlírás jellegű hibákra

3. Valgrind

virtuális processzoron futtatja a programot + közben hibaellenőrzések

- memóriakezelési hibák felderítése
- szálkezelési hibák felderítése
- teljesítmény analízis

4. Strace

lefuttatja a programot, és monitorozza a rendszerhíváokat, és jelzéseket

5. lint

forráskód elemzés, szintaktikai hibák keresése, inicializálási hibák, indexelési hibák keresése

6. IDEk

- Integrated Development Environment
- korábban tárgyalt fejlesztői eszközöket foglalják össze grafikus felületbe
- pl.
    - Eclipse
    - NetBeans
    - QtCreator
    - VS Code
    - …

12. Tétel

Ismertesse a Linux rendszerben használatos inode fogalmát, olvasásának és állításának rendszerhívásait és függvényeit!

1. Inode

Leíró adatstruktúra, amely az adott állomány paramétereit tartalmazza (jogok, méret)

Típusai:

- on-disk inode: lemezen tárolt leíró
  - jogok, méret
  - rá hivatkozó fájlnevek száma. Amikor egy állományt, ami erre hivatkozik törlünk, csökken a számláló. Amint ez 0 lesz + egyetlen processz sem tartja nyitva → a fájl véglegeen törlődik
- in-core inode: a processz megnyitásakor a memóriába másolódik az on-disk inode + számon tartja a processzek számát. Ha ez módosul, a processz bezárásakor felülírja az on-disk inodeot

C/C++ használata: #include<unistd.h>

2. Inode infók kiolvasása

jogok olvasásához nem kell különleges jogosultság, csak az adott könyvtár keresési joga

- *stat(const char *file_name, struct stat *buf)*
  - a file_name paraméter által megadott állomány inode infóit adja vissza
- *lstat(const char *file_name, struct stat *buf)*
  - stat a link lekövetése nélkül
- *fstat(int fd, struct stat *buf)*
  - megnyitott állományok inode infóinak elérése

Struct_stat

- *dev_t stdev* – álományt tartalmazó eszköz azonosítója
- *ino_t st_ino* – állomány on-disk inode száma
- *mode_t st_mode* – állomány jogai és típusa
- *nlink_t st_nlink* – referenciák száma erre az inodera
- *uid_t st_uid* – állomány ownerjének uidje
- *dig_t st_gid* – állomány ownerjének gidje
- *dev_t st_rdev* – ha az állomány speciális eszközleíró, akkor ez a mező tartalmazza a major és minor azonosítót
- *off_t st_size* – állomány mérete bájtokban
- *unsigned long st_blksize* – fájlrendszer blokkmérete
- *unsigned long st_blocks* – állomány által allokált blokkok száma
- *time_t st_atime* – legutolsó hozzáférés időpontja
- *time_t st_mtime* – legutolsó módosítás időpontja
- *time_t st_ctime* – legutolsó változtatás időpontja (az állományon, vagy az inodeon)

3. Jogok lekérdezése/állítása

Lekérdezés:

- hiába tartalmazza az st_mode az álomány jogait, ebből az infó kinyerése nem olyan egyszerű
- megoldás: *int access(const char *pathname, int mode)*
- mode
  - *F_OK* – létezik/elérhető?

- *R_OK* – a processz olvashatja az állományt?
- *W_OK* – a processz írhatja az állományt?
- *X_OK* – a processz futtathatja az állományt (kereshet benne könyvtár esetén?)

- visszatérési érték: 0, ha siker

Jogok állítása:

- *int chmod(const char *path, mode_t mode*): a fájl elérési útvonalával és nevével
- *int fchmod(int fd, mode_t mode):* a file descriptorral (egyedi azonsítója a fájlnak)
- *mode*: hozzáférést szabályozó bitek kombinációja (néhány kapott saját konstanst is)
  - *S_ISUID – 04000* – setuid
  - *S_ISGID – 02000* – setgid
  - *S_ISVTX – 01000* – set sticky bit
  - *S_IRUSR – 00400* – csak a tulaj olvashatja
  - …

4. Tulajdonos és csoport beállítása

- fajtái
  - *int chown(const char *path, uid_t owner, gid_t group)*
  - *int lchown(const char *path, uid_t owner, gid_t group)*
  - *int fchown(int fd, uid_t owner, gid_t group)*
- csak root jogosultsággal lehetséges

13. Tétel

Ismertesse az I/O multiplexelés lehetőségeit a Linux alkalmazásokban

1. I/O multiplexelés

Kliens/szerver program párhuzamos állomány olvasása és írása. (pl. web böngésző hálózati kapcsolaton keresztül több oldal komponensét töltse le, hogy gyorsítsa a hozzáférést)

Legegyszerűbb megoldás:

- *read()* műveletek egy ciklusban – böngésző minden kapcsolaton beolvassa az adatokat, majd továbblép a következőre
- ha minden csatornán egyszerre érkezik adat →☺
- ha nem → ☹ → nem blokkolt I/O kezelés
  - utasítani a read()-ot, hogy ne tartsa fenn a sort →ha nem tudod olvasni, 0-t ad vissza (*O_NONBLOCK* opció)
  - a program folyamatosan olvasgatja mindkét leírót →feleslegesen terheli a rendszert

2. Select

Párhuzamos beolvasása az állományoknak.

Lehetővé teszi, hogy a processz blokkolódjon, és több állományra várakozzon

*int select(int n, fd_set *readfs, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);*

*int pselect(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, const struct timespec *timeout, const sigset_t *sigmask);*

- *fd_set* struktúra kezelése (makrókkal) (file descriptor struct)
  - *FD_ZERO(fd_set *set)* – törli az állományleíró listát (init)
  - *FD_SET(int fd, fd_set *set)* – fd leíró hozzáadása a listához
  - *FD_CLR(int fd, fd_set *set)* – fd leíró kitörlése a listából
  - *FD_IDSET(int fd, fd_set *set)* – True, ha az fd benne van a listában
- *readfs*: (readable file descriptors) azok a file descriptorok, amik akkor oldják fel a select() várakozását, ha olvasható állapotba kerülnek
- *writefds*: írásra kész állapotban oldják fel a select()-et
- *exceptfds*: azoknak a file descriptora, amelyeknek valamilyen különleges állapotára várunk
- *n* : a legnagyobb file descriptor a listából + 1
- *timeout*: select() max. várakozási ideje, ami után mindenképpen visszatér
  - *select()* esetén ennek egy módosított értékével tér vissza, amivel jelzi, hogy mennyi idő telt el a várakozással (nem minden rendszer esetén azonos)
  - *pselect()* – semmit nem változtat a timeout paraméterén
- *sigmask*: aktuális signal maskot ezzel módosítja a rendszerhívás idejére (*NULL* érték kikapcsolja ezt a funkciót)
  - signal mask – a signalok halmaza, amelyek kézbesítése jelenleg blokkolt a hívó számára

3. Poll

Slect()-hez hasonló működésű

Linux rendszerekben a select() a poll()-al van implementálva

*int poll(struct pollfd *fds, nfds_t nfds, int timeout)*

- *pollfd struct*:
  - *int fd* – file descriptor
  - *short events* – milyen eseményekre várunk
  - *short revents* – a kapott eseményeket rögzíti a poll() ide (a feldolgozás során ezt kell vizsgálni)

- *nfds* – a file descriptorok száma
- *timeout* – max. várakozási időtartam

A poll() addig várakozik, amíg legalább egy megadott esemény be nem következik, vagy le nem jár a timeout.

14. Tétel

Hogyan hozhatunk létre folyamatokat a Linux alkalmazásokban és milyen eszközeink vannak a folyamatok közötti kommunikációra?

1. Processzek

OS-el szemben támasztott alapvető követelmény: **multitasking**

Egy processz két egyforma processzé bontása:

- *pid_t fork(void);* → ezután a definiált fv, akármi párhuzamosan létezik
- visszatérési érték: PID
- felbontja a processzt szülő, és gyermek processzre
  - szülő visszatérési értékként megkapja a gyermek PID-jét
  - gyermek 0-t kap
  - hiba esetén -1

Processz megölése

int kill(pid_t pid, SIGINT) (hagyományos mód)

A processz saját azonosítójának lekérése

pid_t getpid(void);

Szülő PID-je

pid_t getppid(void)

Processz várakoztatása a gyermekei végéig

*pid_t wait(int *status)*

- visszatér, ha a hívó processz gyermek processzei közül bármelyik befejezte a működését
- ha a hívás pillanatában a gyermek már zombi →a fv. azonnal visszatér

*pid_t waitpid(pid_t pid, int *status, int options) – flexibilisebb*

- pid paraméterrel egy meghatározott PID-ű gyermek kilépésére várakozhatunk, az opciótól függően
- opciók
  - *<-1* – vár bármely gyermek végére, aminek a csoportazonosítója megegyezik a PID paraméterben adottal
  - *-1* – a függvény ugyanúgy működik, mint a sima wait
  - *0* – vár bármely gyerek végére, amelynek a GID-je megegyezik a hívó processzével
  - *> 0* – vár bármely gyerek végére, amelynek a PID-je megeggyezik a pid-el
  - *WNOHANG* – felfüggesztés nélkül visszatér (no hang), ha még egy gyerekfolyamat sem ért véget (OR kapcsolatban valamelyik fentivel általában)

2. Processzek közötti kommunikáció (IPC)

IPC – Interprocess Communication

ipcs program – kiírja a memóriában lévő olyan IPC objektumokat, amelyekhez a hívó processznek olvasási joga van

- csak az üzenetsorokat
- csak a szemaforokat
- csak az osztott memóriát

ipcrm – egyes IPC objektumok eltávolítása a kernelből – pl. *ipcrm sem 81985536* – az adott ID-jű szemafor eltávolítása

Szemfaorok

- *Unsigned int* – számláló – init egy meghatározott értékre – meghatározza hány processz foglalhatja le
- *#include <sys/sem*.h>
- módosításának atominak kell lennie

- erőforrás lockolása – szemafor dekrementálása → ha már 0, több processz nem foglalhatja le
- létrehozása:
    - *int semget(key_t key, nit nsems, int semflg)*
        - *key* – egyedi azonosító – ezzel lehet rá hiatkozni
            - ha már létezik ilyen, a fv. a létező kulcsával fog visszatérni
            - kulcskeresés: ftok fv.
        - *nsems* – létrehozandó szemaforok száma
        - *semflg* – hozzáférési jogosultság
- *int semctl(int semid, int semnum, int cmd, ...)*; - szemaforvezérlés
    - cmd lehet:
        - *IPC_STAT* – szemaforinfó lekérdezése (olvasási jogosultság szükséges)
        - *IPC_SET* - jogosultság uid, gid módosítása
        - *IPC_RMID* – szemafortömb megszüntetése, felébresztve a várakozó processzeket
        - *GETALL* – szemafortömb elemeinek értékét adja vissza
        - *GETNCNT* – egy adott szemaforra várakozó processzek száma
        - *GETPID* – szemafortömb utolsó módosítójának PID-je
        - *GETVAL* – key
        - *GETZCNT* – egy szamfor nulla értékére várakozó processzek száma
        - *SETALL* – szemafortömb összes értékét állítja be
        - *SETVAL* – egy szemafor értékét állítja
- *int semop(int semid, struct sembuf *sops, unsigned nsops);*
    - szemaforra várakozás
    - *semid* – szemafortömb azonosítója
    - *sembuf* – struktúrák egy tömbje – a végrehajtandó műveletet írják elő
        - *sem_num* – szemafor indexe a tömbben
        - *sem_op* – szemafor értékének változtatása
        - *sem_flg* – művelet jelzőbitjei
            - *IPC_NOWAIT* – a műveletet megkísérli azonnal végrehajtani, különben hiba
            - *SEM_UNDO* – a művelet végrehajtódik, amikor a hívó processznek vége lesz

Üzenetsorok (message queues)

- FIFO kommunikációs csatorna, amibe a programozó által meghatározott formátumú adatcsomagokat lehet belerakni
- *#include <sys/msg.h>*
- egy üzenetsorban több üzenetcsatornát lehet használni
- fizikailag: linked list a kernel címterében (*struct msg*)
    - *struct msg *msg_next* – a következő üzenet a sorban
    - *long msg_type* – az üzenet típusa
    - *char *msg_spot* – magára az ünenetre mutató pointer(a kernel nem tud semmit a formátumról)
    - *short msg_ts* – az üzi mérete
- az *msgbuf* struktúra az alapja annak, ahogy a rendszer leír egy üzenetet
    - *long mtype* – üzenettípus azonosítója
    - *char mtext[1]* – üzenet szöveg tartalma
- az *msgbuf* újradefiniálható, tartalmazhat komplex adatot (lehet definiálni egy saját struktúrát)
    - ekkor az msgbuf struktúra:
        - *long mtype*

        - *struct definiált_üzenet_struktúra struktúranév*
- *int msgget(key_t key, int msgflg)*
    - üzenetsor létrehozása
    - key generálása: ftok fv.
    - jelzőbitek
        - *IPC_CREAT* – létrehozás
        - *IPC_EXCL* – IPC_CREAT-el együtt használva: visszatér hamis értékkel, ha a létrehozandó szemafor már létezik
    - amennyiben a megadott kulcs létezik, akkor a függvény a már létező üzenetsor azonosítójával tér vissza. Egyébként az azonosító az újonnan létrehozotté
- *int msgsnd(int msqid, struct msgbuf *msgp, size_t msgsz, int msgflg)*
    - üzenet küldése: írása a sorba
    - *msqid* – üzenetsor azonosítója
    - **msgp* – saját üzenetünk pointere
    - ha a sor tele van → hiba
    - jelzőbit: *IPC_NOWAIT* vagy 0
- i*nt msgrcv(int msqid, struct msgbuf *msgp, size_t msgsz, long msgtype, int msgflg)*
    - annak a memóriaterületnek a pointere, ahova az üzenetet kérjük
    - kiolvasás menete:
        - ha az *msgtype* 0 →a soron következő üzenetet olvassa ki
        - pozitív és az MSG_EXCEPT bit nincs bekapcsolva → legelső üzenet, aminek a típusa *msgtype*
        - ha az MSG_EXCEPT bit bekapcsolva → első üzenet, aminek a típusa nem *msgtype*
        - ha *msgtype* negatív → annak a kiolvasása, amelynek típusa kisebb vagy egyenlő, mint az *msgtype* abszolút értéke
- *int msgctl(int msqid, int cmd, struct msqid_ds *buf)*
    - *msqid* – üzenetsor azonosítója
        - *cmd*
        - *IPC_STAT* – info másolása a buf-ba
        - *IPC_SET* – a buf által mutatott sturktúra némely tagja alapján állítja az üzenetsor tulajdonságait
            - *msg_perm.uid*
            - *msg_perm.gid*
            - *msg_perm.mod*e (alsó 9 bit)
            - *msg_qbythes*
        - *IPC_RMID* – üzenetsor megsemmisítése
    - **buf* - tulajdonságokat rögzítő struktúra
        - *struct ipc_perm msg_perm* – hozzáférési jogosultságok
        - *sturct msg *msg_first* – az első üzenet az üzenetsor lácolt listájában
        - *sturct msg *msg_last* – az utolsó üznenet a láncolt listában
        - *time_t msg_sti*me – legutolsó küldés ideje
        - *time_t msg_rti*me – legutolsó olvasás ideje
        - *time_t msg_cti*me – legutolsó változtatás ideje
        - *struct wait_queue wwait*
        - *struct wait queue_rwait*
        - *ushort msg_cbytes* – az üzenetsorban levő bájtok száma, azaz az üzenetek száma

- *ushort msg_qnum* – az éppen az üzenetsorban lévő üzenetek száma
- *ushort msg_qbytes* – az üzenetsorban levő bájtok maximális száma
- *ushort msg_lspid* – a legutolsó küldő processz azonosítója
- *ushort msg_lrpid* – a legutolsó olvasó processz azonosítója

Megosztott memória/Shared memory

- Közös memóriatartomány, amihez több processz is hozzáférhet → leghatékonyabb, leggyorsabb
- fork →a gyermek processz örökli a szülőhöz csatolt összes megosztott memóriatartományt
- *#include <sys/shm.h>*
- int shmget(key_t key, int_size, int shmflg)
  - közös memóriatartomány létrehozása
  - *key – ftok*
  - *shmflag – IPC_CREAT, IPC_EXCL*
  - *size* – a kívánt memória mérete bájtokban
    - *PAGE_SIZE* – spec érték – memória lap mérete → ajánlott ennek egész szánú többszörösét lefoglalni
- *void *shmat(int shmid, const void *shmaddr, int shmflg)*
  - közös memóriatartományhoz való hozzácsatlakozás (attach)
  - visszatérési érték: memóriatartomány pointere
  - *shmflg*
    - *SHM_RND* – a visszaadott cím az shmaddr értéke lesz lekerekítve a legközelebbi laphatárra
    - gyakorlatban általában 0 – a rendszerre bízza a megfelelő címtartomány kiválasztását
    - *SHM_RDONLY* – csak olvasásra csatolja a megosztott memóriát
- *int shmdt(const void *shmaddr)* – megosztott memóriáról lecsatlakozás (detach)
- *int shmctl(int shmid, int cmd, struct shmid_ds *buf)*
  - *shmid* – shared memory id
  - *cmd*
    - *IPC_STAT* – infó a megosztott memóriáról
    - *IPC_SET* – hozzáférési jogosultságok és/vagy azonosítók megváltoztatása
    - *IPC_RMID* – megosztott memória törlése
      - ha nincs olyan processz, ami csatolva tartaná az erőforrást → azonnal megszűnik
      - ha van → megjelöli megszüntetésre, hogy az utolsó lecsatolás után dögöljön meg
    - *SHM_UNLOCK* – engedélyezi a swappinget
  - **buf*
    - *struct ipc_perm shm_perm* – hozzáférés és azonosítók beállítása
    - *int shm_segsz* – memóriatartomány mérete
    - *time_t shm_atime* – legutolsó attach ideje
    - *time_t shm_dtime* – legutolsó detach ideje
    - *time_t shm_ctime* – utolsó változás ideje
    - *unsigned short shm_c*pid – a létrehozó processz azonosítója
    - *unsigned short shm_lpid* – az utolsó művelet végrehajtójának azonosítója
    - *short shm_nattch* – aktuális felcsatolások száma
    - *shm_npages* – tartomány mérete (nem írható)
    - *unsigned long *shm_pages* – (nem írható)
    - *struct vm_area_struct *attaches* – felcsatolások leírója (nem írható)

15. Tétel

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

1. Szálak: könnyűsúlyú processzek (Lightweight Processes)

Processzek részei:

- kód
- adat
- verem
- fájlleírók
- jelzéstáblák

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

2. Linux száltípusok:

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

3. Szálak létrehozása

- int clone(int(*fn) (void*), void *child_stack, int flags, void *arg)
  - fork fv. kiterjesztése
  - (void*) – az indítandó processz vagy szál belépési pontja (pointer az fn függvényre, aminek void az argumentuma, és a visszatérési értéke int (így ennek a deklaráció is int típusú))
  - *child_stack – veremmutató
  - *arg – a megadott függvénynek átadandó paraméter
  - flags – egyéb opciók
- *int pthread_create(pthread_t *thread, pthread_ *attr_t attr, void * (*start_routine) (void *), void * arg)*;
  - szál elindítása
  - belépési pontja a start_routine fv.
    - *void* start_routine(void* param)* formában deklarálható
    - a param paraméter az prhread_create fv arg paraméterével adhatunk meg

- **thread* argumentumban a szál leíróját kapjuk vissza
- *attr* paraméter
  - *NULL* – alapértelmezett beállíátsok

- *int pthread_join(pthread_t thread, void **thread_return)*;
  - felgüggeszti a hívó szál működését mindaddig, amíg a thread argumentumban megadott szál be nem fejezi a futását
- példaprogram
  - fordítás: ha a pthread könyvtárat használjuk, hozzá kell linkelni → -lpthread

Szál indítása

```
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

void *thread_function(void *arg) {
        int i;
        printf(„A szál indul… \n”);
        for (i=1; i<=20; i++) {
                printf(„%d. Hello szál világ%\n”);
                sleep(1);
        }\
        printf(„A szál kilép…\n”);
        return 0;
}

int main(void) {
        pthread_t mythread;

        if (pthread_create(&mythread, NULL, thread_function, NULL) ){
                fprintf(„stderr, „Hiba a szal letrehozasaban. \n”);
                exit(1);
        }
        if (pthread_joid(mythread, NULL)) {
                fprintf(stderr, „Hiba a szál megvárásában.\n”);
                exit(1);
}
        sleep(5); // késleltetés, mert ha azonnal kilép a main,
                  // megszűnik a főszál, megsemmisíti a létrehozott szálat is
        printf(„ A főszál kilép …\n”);
        return 0;
}
```

4. Szinkronizáció

Külcsönös kizárás (MUTEX)

- *#include <pthread.h>*
- Szálszinkronizációban is használják
- Két állapot: *locked, unlocked*
- A mutec egyszerre csak egy szálé lehet
- **Linux alatt háromféle MUTEX:** megmondja, mi történik akkor, ha egy olyan szál próbál lefoglalni egy MUTEXET, amelynek már birtokában van
  - *gyors*

        - várakozni fog arra, hogy felszabadítsa valaki. De mivel az a valaki ő maga volt, ezért infinite loop → so sad ☹
        - deklarálás
            - makróval: pthread_mutex_t fastmutex = PTHREAD_MUTEX_INITIALIZER;
            - függvénnyel: int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr) ÉRVÉNYES MINDEGYIKRE
    - *rekurzív*
        - újra lefoglalja → inkrementális lefoglalás → ahhoz, hogy felszabaduljon, minden egyes lefoglalást fel kell szabadítani visszafelé
            - deklarálás makróval pthread_mutex_t recmutex = PTHREAD_RECURSIVE_INITIALIZER_NP;
                - NP utótag: non-portable
    - *hibaellenőrző*
        - megnézi, hogy foglalt-e már a MUTEX → ha igen, hibával tér vissza
        - deklarálás makróval: pthread_mutex_t errchkmutex = PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP;
- *int pthread_mutex_destroy(pthread_mutex_t *mutex)*;
    - MUTEX megszüntetése (jelenleg a Linux csak chekolja, hogy szabad-e ☹)
    - feltétele: ne legyen foglalt
- *int pthread_mutex_lock(pthread_mutex_t *mutex)*;
    - MUTEX lefoglalása
    - ha épp szabad
    - a hívó szál addig lesz felfüggesztve, amíg a MUTEX nem szabad → ☹
- *it pthread_mutex_trylock(pthread_mutex_t *mutex)*;
    - a szál nem fog várakozni, mert a fv. kapásból visszatér, ha nem tudja lefoglalni EBUSY értékkel
- *int pthread_mutex_unlock(pthread_mutex_t *mutex)*;
    - unlock

Feltételes változók (conition variables)

- *#include <pthread.h>*
- lehetővé teszi, hogy a kritikus szekcióban levő szálak felfüggesszék futásukat mindaddig, amíg egy erőforrásra igaz nem lesz valamilyen állítás → várakozás közben nm fogják a kritikus szekciót biztosító mutexet
- **két művelet végezhető**
    - várakozásra jelentkezés
    - esemény bekövetkezte
- **a jelzés pillanatszerű** → ha az egyik szál meghívja a várakozó fv-t, pont akkor teljesül a feltétel → lekési
    - **megoldás**:
        - olyan mutex, amit lefoglalunk, mielőtt hívjuk a várakozó fv-t → várakozás megkezdése után elengedi a mutexet
        - a jelzés kiadása előtt is lefoglaljuk a mutexet → ha éppen akkor kezd várakozni a fv. megvárjuk, míg várakozni kezd → felszabadul a mutex → utána szerezzük meg
    - **<u>TEHÁT: a feltételes változóhoz rendelni kell egy mutexet</u>**
- létrehozás
    - *pthread_cond_t cond = PTHREAD_COND_INITIALIZER*;
    - *int pthread_cond_init(pthread_cont_t *cond, pthread_condattr_t *cond_attr)*;
        - *cond_attr*

- *NULL*: alapértelmezett paraméterekkel jön létre
- egyébként, mivel a Linux nem definiál attribútumokat a feltételes változók számára, a cond_attr a pthread_cond_init figyelmen kívül hagyja

- megszüntetés:
  - *int pthread_cond_destroy(pthread_cond_t *cond)*;
  - egy szál sem várakozhat a változóra
- jelzés küldése
  - *int pthread_cond_broadcast(pthread_cond_t *cond)*;
  - előtte le kell foglalni a mutexet, majd után afelszabadítani
- várakozás:
  - *int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)*;
  - előtte le kell foglalni a mutexet
  - utána elengedni
- meghatározott ideig várakozás:
  - *int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime)*;
  - abstime

Szemaforok

- Különbség az POSIX IPC-vel szemben:
  - egyetlen szinkronizációs objektum ⇔ szemafortömb (IPC)
  - szálak és processzek között is használható ⇔ csak processzek között (IPC)
- Létrehozás
  - Névtelen szemafor létrehozása
    - *int sem_init(sem_t *sem, int pshared, unsigned int value)*;
      - pshared: ha nem 0, akkor más processzek is hozzáférhetnek
- Megnevezett szemaforok
  - *sem_t *sem_open(const char *name, int oflag)*;
  - *sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int value)*;
- Szemafor lefoglalása
  - *int sem_wait(sem_t *sem)*;
  - *int sem_trywait(sem_t *sem)*;
    - ha a szemafor értéke pozitív: mindkét fv. lefoglalja, visszatér 0-cal
  - ha nem lehet lefoglalni:
    - *wait* – várakozik, megszakíthatja:
      - szemafor értéke nagyobb lesz, mint nulla
      - vagy egy jelzés
    - *trywait* – visszatér EAAGIN értékkel
- Szemafor felszabadítása
  - *int sem_post(sem_t *sem)*;
- Szemafor aktuális érétkének lekérdezése:
  - *int sem_getvalue(sem_t *sem, int *sval)*;
    - ha lefoglalták
      - 0
      - vagy negatív szám, amelynek abszolút értéke megadja a rá várakozó processzek számát

- Megszüntetés
  - Névtelen szemafor
    - *int sem_close(sem_t *sem)*;
  - Megnevezett szemafor lezárása, majd megszüntetése
    - *int sem_close(sem_t *sem)*;
    - *int sem_unlink(const char *name)*;
    -

16. Tétel

A Linux alkalmazások determinisztikus futásidejének érdekében milyen óvintézkedéseket kell megtennünk az alkalmazásunkban? (Valósidejű Linux alkalmazás fejlesztése)

1. Valósidejű Linux

- RT-Preemt Kernel patch: a kernel nem tér el nagyban
- az RT-ből adódó különbségeket a kernel kezeli → **alkalmazásfejlesztésben nincs különbség**
  - a rendszerhívások ugyanazok, csak a lekezelésük más
- **amire figyelni kell: futásidő determinisztikus legyen**
  - valósidejű ütemezést és prioritást kell használni
  - az alkalmazás memória területének végig a fizikai memóriában kell lennie
  - a stack területéből adódó laphibákat meg kell előzni

2. Ütemezés és prioritás

- **ütemezés típusai** (a legmagasabb prioritású futásra kész folyamatok közül melyiket válassza ki a scheduler)
  - FIFO
  - Round Robin
- **folyamatokra**
  - *#include <sched.h>*
  - stratégia kiválasztása + prioritás közös függvényben*: int sched_setscheduler(pid_t pid, int policy, const struct shed_param *param);*
    - *pid* – process id
    - *policy* – ütemezési stratégia
    - *a ssched_param struct*-ban *int sched_priority* – prioritás érték
- **szálakra**
  - *#include <pthread.h>*
  - *pthread_setschedparam(pthread_t thread, int policy, const struct sched_param *param)*;

3. Memória terület fizikai memóriában tárolása

- a folyamat memórialapjai nem kerülhetnek ki a fizikai memóriából ← nem engedhető meg, hogy a folyamat közben töltödjenek be a lapok
- nem megengedhető, hogy ne férjenek be a memóriába a lapok, így az újabbak betöltésénél fel kelljen szabadítani helyet
- folyamat lapjainak zárolása

#include <sys.mman.h>

- *int lockall(int flags)*;
  - *MCL_CURRENT* – a jelenlegi lapokat lockolja
  - *MCL_FUTURE* – a jövőbenieket lockolja
- *ajánlott: mlockall(MCL_CURRENT|MCL_FUTURE)*;

4. Stack okozta laphibák

- stack növekedése → új memórialapok allokálása vállik szükségessé → késleltetés, nem-determinisztikus
- meglőzés: előre allokálás → ki kell számolni a szükséges területet
- *memset(unsigned char tmp[MAX_STACK], 0, MAX_STACK)*;
  - a program elején

17-18. Tétel

Hogyan valósítunk meg Linux alkalmazásunkban összeköttetés orientált / nélküli kommunikációt

1. Linux hálózatkezelés

- *#include <sys/socket.h>*
- socketek
  - mint a Linux többi erőforrásai → egy-egy hálózati kapcsolat is állományként van reprezentálva
  - kapcsolat létrehozása → létrejön egy socket állomány →*read() write()* → nem ajánlott
  - ajtó az alkalmazás és a hálózat között
    - IP cím
    - portszám
- hálózati kapcsolat felépítése
  - aszimmetrikus művelet

Kliens	Szerver
Socket létrehozása	Socket létrehozása
	Kapcsolódás portjának megadása (*bind()*)
	Hallgatózás (*listen()*)
csatlakozási kérelem küldése: (*connect()*)	
	csatlakozás elfogadása (*accept()*)
	csatlakozás létrejött, létrehoz egy kliens-socket-et → a kommunikáció ezek után ezen keresztül történik

2. Összeköttetés alapú kommunikáció (connection-oriented)

- folyamatos kapcsolatban van a két oldal között
- send, receive
- TCP
  - Transmission Control Protocol
  - megbízható átvitel
  - sorrendhelyes, hibamentes szállítást nyújt
  - ára: késleltetés (kapcsolat felépítése, bontása)
  - e-mail (smtp), ssh, web (http), ftp
- küldés: *int send(int s, const void *msg, size_t len, int flags*
  - *s* – socket leírója
  - **msg* – elküldendő adat buffere
  - *len* – elküldendő adat mérete
  - *flags*
    - *MSG_OOB* – soron kívüli sürgős adatcsomag (out-of-band)
    - *MSG_DONTROUTE* – ne a routeren keresztül, csak közvetlen hálózatra
    - *MSG_DONTWAIT* – engedélyezi a nem blokkoló I/O-t
    - *MSG_NOSIGNAL* – adatfolyam alapú kapcsolat esetén
- fogadás: *int recv(int s, void *buf, size_t len, int flags)*
  - jelzőbit eltérő:

- *MSG_OOB* – soronkívüli adat fogadása
- *MSG_PEEK* – adat beolvasása anélkül, hogy a beolvasott adatot eltávolítaná a bufferből
- *MSG_WAITALL* – addig nem tér vissza, amíg a puffer meg nem tellik, vagy rendhagyó dolog történik (pl. jelzés)
- *MSG_NOSIGNAL*

3. Összeköttetés nélküli kommunikáció (connectionless)

- a kommunikációra „megbízhatóként tekint" → nem foglalkozik azzal, hogy valóban célba ért-e a csomag (ha hiba van, vagy megszakadt a csomag, akkor úgyis dob hibát a kommunikációs interface)
- olyan alkalmazásoknál, ahol nem probléma, ha egy-egy csomag elveszik
- UDP csomagok TCP helyett
  - UDP
    - User Datagram Protocol
    - korlátozott méretű adatcsomagok átvitelére képes, nem megbízható kommunikációs útvonalat biztosító protokoll
    - nem ganadtál célba jutást, csak hibakezelst nyújt
    - gyors
    - Hálózat menedzsment, Voip, média streaming, névfeloldás
    - blokkolás elkerülése: többszállúság, multiplexing
- küldő oldalon: socket létrehozása → adat küldése
  - küldés: *int sendto(int sd, const void *msg, size_t len, int flags, const struct sockaddr *to, socklen_t tolen)*
    - *sd* – socket descriptor
    - **msg* – message címe
    - *len* – message mérete
    - *flags*
    - **to* – socket pointere
    - *tolen* – socket mérete
- vevő oldalon:
  - megadjuk milyen porton figyelünk
  - honnan fogadjuk a csomagot (bárhonnan)
  - bind (socket hozzárendelése a helyi porthoz)
  - buffer törlése
  - üzenet fogadása (várakozás): *int recvfrom(int sd, void *buf, size_t buffsize, int flags, struct sockadds *from, socklen_t *fromlen)*
    - ha a megadott puffer mérete kisebb, mint kéne, az érkező üzenet vége automatikusan levágódik

19. Tétel

Milyen különbségek vannak a Linux kernel modulok és alkalmazások között? (felépítés, fordítás, hibakezelés, további megkötések)

1. Linux kernel

- monolitikus modul
  - egyben minden funkció
  - adatstruktúra
- más megoldás:
  - mikrokernel: minden funkcionális elemet külön egységben tartalmaz + jól meghatározott kommunikáció
- monolitikus modul következménye: új komponens, funkció hozzáadása bonyolultabb
  - korábban újrakonfigurálást, és újrafordítást igényel
  - még most is az nVidia CUDA képes GPU-inál
- 1.2-es verzió óta betölthető modulok suppportja
  - dinamikus hozzáadás, és elvétel a modultól
  - eszközvezérlők

2. Fordítás

- saját makefile-ok
- c fordítás

3. Betöltés – eltávolítás

- modul betöltése: *insmod hellomodule.ko*
  - hozzálinkeli a kernelhez
  - inicializálja
- modul eltávolítása: *lsmod hellomodule*
  - meghívódik a modul tisztító függvény
  - unlinkeli a programot
  - törli a memóriából
- betöltött modulok listázása: *lsmod*
- modprobe:
  - modul → */lib/modules/<kernelverzió>/*
  - *depmod* – modulok függőségi listájának előállítása
  - *modprobe* – betölti a megnevezett modult és a függőségeit is

4. Felhasználói mód – kernel mód

- programok: felhasználói módban
  - megvédi a rendszert az illegális hozzáféréseket az erőforrásokhoz
- modulok: kernel módban
  - bármi megengedett → kernel modulokat csak root tölthet be
- memóriahasználat
- eszközökhöz való hozzáférés

5. Különbségek alkalmazások és modulok között

modulok	szempont	alkalmazások

csak C	nyelv	bármi, amire van fordítód az adott Linux distrora
két belépési pont: 1. betöltéskor inicializálódik csak, majd várja, hogy használja valaki 2. tisztogató: eltávolítás előtt aktivizálódik	belépési pontok, és végrehajtódás	main() függvénynél beugrik, és dolgozik, amíg a kilépési feltétel nem adott
csak a kernellel linkelődik → csak olyan fv. használható, ami már le van implementálva a kernelben → nem használunk fejlesztői könyvárakat → nincsenek headerek a hagyományos értelemben. Ami használható az a linux és az asm alkönyvtárakban van	headerek használata	a program sokszor nem tartalmazza minden adott fv. definícióját → libary-k, headerek fordítási fázisban a linkelés során oldódnak meg ezek a linkek
a kernelnek mindig a memóriában kell lennie, nemkerülhet ki lapcsere során a merevlemezre → spórolni kell a hellyel	memóriakezelés	alkalmazástól függően fizikai memóriában, swap-ban, vagy a merevlemezis használható
fixpontos (tizedes pont előre meghatározott fix helyen + előjelbit)	számábrázolás	ami tetszik (lebegőpontos: értékes számjegy + kitevő)
komoly problémák, akár rendszerösszeomlás bonyolult hibakezelés	hibakezelés, segmentation fault (valami olyasmihez való hozzáférés, ami nem a tied, pl. túlindexelés. Már valaki használja azt a memóriaterületet)	nem veszélyes, kiszűrhető elterjedt hibakezelő megoldások

20. Tétel

Hogyan implemeltálhatuk a paraméter átadást a Linux kernel modulban? Hogyan használhatjuk a mechanizmust?

1. Modulok egymásra épülése

- egyes modulokhoz kiexportálunk más modulok által használt szimbólumokat
- abból, amiből ki akarjuk exportálni: definiálnunk kell az EXPORT_SYMTAB makrót
- exportálandó függvények meghívása: EXPORT_SYMBOL makróval
- Paraméter átadás használata:
  - pl. eszközvezérlő: különböző fontos paraméterei vannak, pl. I/O címtartománya
    - ha ezt hardkódoljuk a kernelbe, a módosítás esetén újra kell fordítani
    - megoldás: modul betöltésénél megadjuk a beállításokat → Linux kernel paraméterátadási mechanizmusa

2. Paraméter létrehozása

- globális változó definiálása az insmod paranccsal (ekkor lehet adni inicializációs értéket neki, amely egyben az alapértelmezett értéke is)
- változó paraméterré tétele a module_param() makrókkal
  - module_param(név, típus, sysfs_jogok)
    - név – változó neve
    - típus – változó típusa
    - sysfs_jogok – paraméter jogosultság a sysfs állományrendszerben
  - module_param_names(név, változónév, típus, sysfs_jogok)
    - változónév – tényleg kétszer kell leírni, implementációs izé
  - module_param_array(név, tíus, méret_mutató, sysfs_jogok)
    - méret_mutató – opcionális, a rendszer eltárolja a tömb elemeinek számát a paraméterátadáskor
  - module_param_array_named(név, változónév, típus, méret_mutató, sysfs_jogok)
  - module_param_string(név, változónév, méret, sysfs_jogok)
    - méret – string típus esetén a szöveg max. hossza
  - module_param(néc, charp, sysfs-jogok)
    - charp – karakter mutató

3. paraméter leírásának megadása a modulban

- MODULE_PARM_DESC(név, leírás) makró
- a felhasználók a modulból megtudhatják a paraméterek jelentését
- a leírás megtekintése: modinfo modelname.ko

4. Paraméter átadása a kernelnek

- insmod modulename.ko param1=”hello” param2=42
- nem kell minden paramétert megadni, amit nem adtunk meg az a default értéket tartja meg (amit a initkor megadtunk)

5. Paraméter kiolvasása

- ha a sysfs jogosultságok beállításánál adtunk olvasási vagy írási jogot
- /sys/module/<modulename>/parameters/ directory

- minden egyes paraméterhez tartozik egy virtuális állomány
- ezeket írhatjuk és olvashatjuk

6. Konkurencia probléma

- utólagos módosítások esetén
- komplexebb paramétereknél könnyen előfordulhat, hogy más alkalmazás is használni akarja
- kritikus vészhelyzet!!!!!!!!!! MINDMEGHALUNK!!!!!

21. Tétel

Ismertesse a karakteres eszközvezérlő implementációját a Linux rendszerben!

1. Eszközkezelők

- állományok Linuxban (*/dev/*)
  - egyszerű állomány
  - könyvtár
  - eszközök
    - blokk
    - karakter
  - szimbolikus link
- eszközvezérlő implementálása: függvények készítése …
  - állománnyal való műveletekre
    - megnyitás
    - írás
    - olvasás
    - bezárás
  - modulkezelésre
    - init
    - tisztogató
  - kernel nyilvántartásába vételére
  - kernel nyilvántartásából eltávolításra

2. Egy eszköz azonosítója

- major
  - azonosítja az állományhoz/eszközhöz tartozó drivert
  - fő eszközvezérlő csoport
    - ttyXX – 4 – char
    - hdXX – 3 – block
    - sdXX – 8 - block
    - kbdXX – 11 – char – keyboard
  - kernel elintézi az ezen a szinten történő dolgokat
- minor
  - meghatározza egész pontosan melyik eszközre hivatkozunk
  - lehetővé teszi, hogy több azonos típusú eszközt csatoljunk
    - a meghajtó használhat eltérő implementációt a különböző minorokhoz, de az egységes viselkedés ajánlott
  - felhasználói programok kezelik
  - eszköz azonosítása
    - ttyS0 – 64
    - hda – 0
    - sda – 0

3. Eszköz regisztrálása, eltávolítása

- **Regisztrálás**
  - virtuális állománykezelő függvényeinek fő azonosítóhoz rendelése

- *int register_chrdev(unsigned int major, const char *name, struct file_operations *fops)*;
  - negatív visszatérési érték – ERROR
  - *major* – fő azonosító
    - rossz meghajtó választásának kockázata
    - 0 érték – dinamikusan a rendszer is meg tudja határozni (visszatérési értékben megadja)
  - *fops* – strktúrára mutató – az állománykezelési funkciókat megvalósító függvények mutatóit tartalmazza

- **Modul eltávolítása**
  - *void unregister_chrdev(unsigned int major, const char *name)*;
    - ezsközvezérlő neve: biztonsági okokból → a kernel összehasonlítja a korábban regisztrálttal, és ltérés esetén nem hajtja végre

4. Eszközállomány dinamikus létrehozása, eltávolítása

- **állományok nyilvántartása**
  - eredetileg minden Linux által támogatott eszközhöz volt egy dummy állomány a /dev/-ben
  - aztán virtuális devfs használata
  - ma: *udev*
- **udev**
  - felhasználói módban futó alkalmazás (udev daemon)
  - fogadja a kernel hotplug üzeneteit → dinamikusan töröl , vagy létrehoz eszközállományokat a memóriában tárolt virtuális eszköz-állományrendszerben
  - hotplug üzenetek
    - Linux eszközmodell
      - Tartalmazza:
        - eszköz típus osztály
        - melyik buszra csatlakozik
        - speciális attribútumok
    - be kell regisztrálni az eszközhöz tartozó eszközmodellt
- **eszközállomány automatikus létrehozása:**
  - *struct device* device_create(struct class *osztály, struct device *szülő, dev_t eszköz, const char *név, ...)*;
    - *osztály* – eszköz-osztály
      - létrehozása: *struct class *class_create(struct module* modul, const char *név)*;
        - modul – a modul makrója, ami az eszközállományhoz tartozik (*THIS_MODULE* makróval lehet átadni a jelenlegit)
        - név – az osztály szöveges megnevezése
      - megszüntetés: *void class_destroy(struct class *osztály)*;
    - *szülő* – szülőeszköz, ha ilyen létezik
    - *eszköz* – major, minor azonosítókat tartalmazó argumentum
      - előállítható: *MKDEV()* makróval
- **eszköz eltávolítása:** *void device_destroy(struct class* osztály, dev_t eszköz)*;

5. Állományműveletek implementálása

- *file_operations* struktúra: függvények mutatóinak struktúrája
  - mit történjen megnyitáskor, olvasáskor, íráskor stb ..

- NULL pointer: az adott eszköz nem támogatja az adott műveletet
- a default leírását a linux/fs.h állomány tartalmazza (/usr/include/ or /usr/locale/include/)
- elemei (többek között)
  - *owner: THIS MODULE*
  - *read*
  - *write*
  - *open*
  - *release*

6. Adatmozgatás Kernel és User space között

- *#include <linux/uaccess.h>*
- read/write függvények bufferei user-space-ben vannak ⇔ kernel modulok által foglalt területek: kernel space-ben → a kettő nem lát rá egymásra → meg kell oldani az átmozgatást
- előre definiált függvények a problémára
  - *copy_from_user*
  - *copy_to_user*

7. Minor azonosító használata

- pl. többszörös soros kommunikáció
- egy driver kell, de több állomány használata célszerű
- a mellékazonosító megválasztása ránk van bízva
- guideline:
  - olvasást/írást kezelő függvényekben helyezzük az elágazásokat, amelyek mellékazonosító alapján másképp fognak viselkedni
  - állomány megnyitásakor eltérő lekezelő függvényeket állítunk be (paraméter átadás?)

22. Tétel

Ismertesse a párhuzamosság kezelésének eszközeit a Linux kernelben!

1. Párhuzamosan futtatható szálakkal szembeni elvárások

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

2. Atomi műveletek

Legegyszerűbb, leghasznosabb

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

Használat

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

3. Ciklikus zárolás (spinlock)

Működése

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

Használata

- létrehozása: *spinlock_t lock*;
- init: *spin_lock_init(&lock);*
- egy lépésben a létrehozás, és init makróval: *static DEFINE_SPINLOCK(lock);*
- lefoglalás: *spin_lock(&lock);*
- felszabadítás: *spin_unlock(&lock);*
- kritikus szakasznál nem hívódhat meg a megszakításkezelő sem → átmenetileg le kell tiltani: *spin_lock_irqsave(&lock, flags); MŰVELET; spin_unlock_irqrestore(&lock, flags)*
    - ezek makrók, tehát nem érték szerinti átadás történik, hanem ténylegesen átadjuk a változót is (ezért nem pointer a flag)

4. Szemafor

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

Használata:

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

5. Mutex

Megkötések kernelben

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

Használata

- definiálás: *struct mutex asdf;*
- init: m*utex_init(mutex);*
- létrehozás + init: *DEFINE_MUTEX(mutex);*
- lefoglalás: v*oid mutex_lock(struct mutex *lock);*
- lefoglalása, ha várakozási jelzéssel megszakíthatóvá akarjuk tenni: *int mutex_lock_interruptible(stuct mutex *lock);*
- trylock: *int mutex_trylock(struct mutex *lock);*
- felszabadítása: *void mutex_unlock(struct mutex *lock);*
- lefoglaltság ellenőrzése: *int mutex_is_locked(struct mutex *lock);*

6. Olvasó/író ciklikus zárolás és szemafor

Különböző zárolások

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

Használat

- létrehozás: *rwlock_t rwlock;*
- init: *rwlock_init(&rwlock);*
- létrehozás + init: static *DEFINE_RWLOCK(rwlock);*

- olvasás (megosztott) zárolás, feloldás:
  - *read_lock(&rwlock)*;
  - *read_unlock(&rwlock)*;
- írási (kizáró) zárolás/feloldás:
  - *write_lock(&rwlock)*;
  - *write_unlock(&rwlock)*;
- író/olvasó semaphore létrehozás: *struct rw_semaphore rwsem;*
- init: *void init_rwsem(struct rw_semaphore *sem);*
- lefoglalás olvasásra:
  - *void down_read(struct rw_semaphore *sem)*;
  - *int down_read_trylock(struct rw_semaphore *sem)*;
- felszabadítás olvasás után: *void up_read(struct rw_semaphore *sem);*
- lefoglalás írásra:
  - *void down_write(struct re_semaphore *sem)*;
  - *int down_write_trylock(struct re_semaphore *sem)*;
- felszabadítás írás után: *void up_write(struct rw_semaphore *sem);*
- read zárolásból nem lehet write zárolásba menni →holtpont a kernelben → ☹
- DE: írási write zárolás visszaléptethető read zárolásra

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

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

23. Tétel

Ismertesse a megszakítás kezelés implementációját a Linux kernelben! Térjen ki a Bottom Half mechanizmus bemutatására is!

1. Megszakítások

Eszközvezérlő megszakításának lekezeléséhez kell:

- **megszakítás kezelő fv**
  - általános alakja: *typedef irqreturn_t (*irq_handler_t)(int irq, void *devid);*
    - *irq* – megszakítás száma
- **regisztrálni kell a kezelő fv-t az adott megszakításhoz a driverben**
  - regisztráció: *int rquest_irq(unsigned int irq, irq_handler_t handler, unsigned long flags, const char *devname, void *devid),*
    - *handler* – megszakítás kezelő fv
    - *flags* – opciók
      - *IRQF_DISABLED*: gyors interrupt jelzése. Az implementációnak gyorsnak kell lennie, mert a megszakítás kezelése alatt a megszakítás le van tiltva
      - *IRQF_SHARED*: az interruptot megosztjuk más kezelőkkel (pl. több HW használja ugyanazt a megszakítást)
      - *IRQF_SAMPLE_RANDOM*: a megszakítás felhasználható véletlen szám generálásához
      - *IRQF_TIMER*: a megszakítás timer interrupt
    - *devname* – eszközvezérlő neve
    - *devid* – megosztott megszakítás esetén egyedi azonosító
    - visszatérési érték:
      - *IRQ_HANDLED*: a megszakítást lekezelte a fv.
      - *IRQ_NONE*: a megszakítást nem kezelte le a fv.
- **a driver eltávolításakor a regisztrációt is el kell távolítani**
  - törlés: *void free_irq(unsigned int irq, void *devid);*
  - a megszakítást több eszköz is használhatja!! → fontos a pontos *devid*

2. Megszakítások megosztása

- **a megszakítás vonalak száma véges → lehetőség van egy vonalhoz több HW megszakításait is bedrótozni**
- a megszakítás kezelő fv. regisztrációjánál jelezni kell a megosztott használatot (*IRQF_SHARED*)
- devid: egyedi érték, ami a regisztrációt azonosítja
- a megszakítás kezelő fv-nél lehet jelezni, hogy nem ennek a fv-nek szólt a megszakítás: *IRQ_NONE* visszatérési értékkel
- megosztott megszakítások → egy adott vonal letiltása így nem csak egy eszközre van kihatással !!!

3. Kezelő fv-ek megkötései

- nem használható *sleep()* → aktiválná a schedulert
- *kmallock()* csak *GFP_ATOMIC* flaggel, mivel könnyen sleepbe mehet
- kernel és *user space* között nem mozgatunk adatokat
- gyorsaság
- a processz specifikus adatok nem elérhetőek a megszakítás kezelőből (mert nem processz hívta meg)

4. Bottom Half mechanizmus

- ha komolyabb adatfeldolgozást kéne végrehajtani egy megszakítás következtében → nem lehet megcsinálni megszakításban, mert az fáj → solution: BH mechanizmus: felbontja a feladatot két részre
  - **Top half**
    - tényleges megszakítás kezelő rutin
    - gyors letárolás
    - kérvényezése a Bottom Half futásának minél gyorsabban
  - **Bottom half**
    - nem megszakításidőben futó rész
    - nem érvényesek a szigorú megkötések
- implementációjára megoldás: Kernelszálas megszakítás

5. Kernelszálas megszakítás kezelés

Működése

- regisztráció → a kernel létrehoz egy szálat az adott megszakítás lekezeléséhez
- megszakítás érkezik → handler elkapja, gyors feldolgozás → átadja a szálban levő függvénynek a paramétereket → a számolás már nem interruptban fog történni
- ha a szálban történő végrehajtás túl lassú → legrosszabb esetben az adott megszakítás kezelését akadályozza

Használata

- regisztrálása: *int request_threaded_irq(unsigned int irq, irq_handler_t handler, irq_handler_t thread_fn, unsigned long flags, const char *name, void *dev);*
  - *handler* – hagyományos megszakítást kezelő fv (tipikusan NULL, és akkor csak átadja a szállnak a megszakítás paramétereit)
  - *thread_fn* – szál-kontextusban meghívandó fv.
- regisztrálás eszközhöz kötéssel → a felhasználás is együtt történik az eszközzel
  - *int devm_request_threaded_irq(struct device *dev, unsigned int irq, irq_handler_t handler, irq_handler_t thread_fn, unsigned long irqflags, const char *devname, void *dev_id)*;
    - **dev* – eszközt leíró struktúrára mutató

6. Megszakítások tiltása, engedélyezése

- megszakítás tiltása:
  - *void disable_irq(unsigned int irq)*;
    - megvárja a függvény lefutását, ha megszakításban van
  - *void disable_irq_nosync(unsigned int irq)*;
    - azonnali kill
- ismételt engedélyezés
  - *void enable_irq(unsigned int irq)*;
- összes megszakítás tiltása:
  - *void local_irq_disable()*;
- összes IRQ engedélyezése
  - *void local_irq_enable()*;
- érdemes a IRQ-kat menteni, és újra betölteni a disable/enable előtt
  - *void local_irq_save(unsigned long flags)*;
  - *void local_irq_restore(unsigned long flags)*;

24. Tétel

Mintapéldán keresztül mutassa be egy egyszerű WinAPI ablakozó alkalmazás felépítését!

1. Embedded Windows

Fajtái

- Windows 7, 8 Embedded Standard
  - PC alapú eszközökre
  - ATM
  - Játékgépek
  - Pénztárgépek
  - POS terminálok
- Windows Embedded Compact – ami CE ☺
  - PDA-k
  - GPS-ek
  - Ipari automatizálási eszközök
  - valósidejű, komponens orientált, multitaskosz OS
  - preemptív ütemezés
  - felépítés:
    - HW
    - Kernel (Nk.exe)
      - OAM – OEM Abstraction Layer
      - kernel.dll
    - Win32 CE API (felhasználói réteg)

fejlesztés

- verziótól függ
- natív alkalmazások
  - Win32 CE API-cal lehet fordítani bármilyen kódot szinte (C, C++, VB, VBA)
- .NET Compact Framework (MC++, C#, VB.NET)

Saját HW-re illesztés

- esközfüggő rendszerkompoensekből álló csomagra van szükség
- BSP – Boarad Support Package
  - booloader
  - OAL
- driverek

2. Win32 CE API

API – Application Programming Interface

- a rendszert alkotó dll-ek publikus függvényei
- az alapfunkciók a Coredll.dll-ben avnnak
- kiegészítő funkciók külön dll-ekben
- konvenciók: Hungarian Notation

Változótípusok: a nyelvfüggetlenség érdekében saját típusokat definiál

- UINT (unsigned int)
- LONG (long)
- WPARAM (word parameter (UINT))

- LPARAM (long parameter (LONG))
- HANDLE
- LRESULT
- HWND
- WORD
- DWORD

Ütemezés:

- processzek – szálak
- egy processzhez tartozó összes szál ugyanazt a virtuális címtartományt használja → a szálak elérik egymás adatait
- időszelet – quantum – 100 ms, de ez állítható szálanként
- 256 prioritási szint – default: 215

Szinkronizációs objektumok

- Kritikus szakasz – műveletek atomivá tétele
- Mutex – kölcsönös kizárás erőforrások védelmére konkurens hozzáférés ellen
- Szemafor – erőforrásokhoz való hozzáférés szabályozása
- Esemény – események bekövetkezésének várása, és ezek jelzése
- Várakozási sor – szálak közöti szinkronizált adattovábbítás

3. Fejlesztés Windows CE-re

Asztali PC-s verziótól csak kicsit tér el

Ablakozó alkalmazás felépítése API-val

- natív módon futó alkalmazások
  - az OS API-ját felhasználva készítünk közvetlenül futtatható alkalmazásokat
  - nem OOP, mert az OS API-ja sem az (leginkább C)
  - MFC, ATL, STL
    - olyan kiegszítő szolgáltatások, amelyek interfészelik az OOP progit az API-hoz
    - C++
    - VBA
- entry point: WinMain
- az applikáció létrehoz néhány ablakot
  - minden ablakhoz tartozik egy ablakkezelő fv.: CALLBACK fv.
  - nem a mi programunk hívja, hanem a Windows
- egy tipikus Windows program részei:
  - Init
    - ablak osztály regisztrálása
    - változók inicializállása
  - főablak létrehozása
  - üzenetkezelő ciklus
    - kiolvassa a várakozási sorból az üzeneteket, és továbbítja a Windowsnnak
    - a Windows ezekkel a paraméterekkel hívja meg az ablakkezelő f-t
  - főablak kezelő fv-e (Window procedure)
    - üzenetekre való reagálás
  - Kilépés előtti takarítás → kilépés

Menedzselt alkalmazásfejlesztés CE-re

- Windows API-ban a fejlesztés bonyolult

- szoftverfejlesztő cégek: az idő pénz → támogatják a menedzselt alkalmazásfejlesztést
- natív fejlesztés: teljes mértékben ki tudjuk használni az OS API-ja által nyújtot lehetőségeket
- menedzselt fejlesztés (.NET)
    - kell futtató környezet (CLR – Common Language Runtime)
    - köztes nyelv:
        - MSIL – Microsoft Intermediate Language
        - a fejlesztés alatt csak egy köztes kódot használunk
        - a tényleges fordítás
            - telepítéskor
            - első futtatáskor – JIT – Just in Time
        - ☺ - némileg platformfüggetlen
- .NET-es alkalmazások minimálisan maradnak el a WinAPI-soktól
- nyilvánvalóan csak felhasználói programok születhetnek (OAL, service-k csak natív módban)

```
#include ….
// globális változók deklarálása

WinMain(…) {
        // lokális változók deklarálása
        InitProgram();

        // ablakosztály regisztrálása
        WNDCLASSEX wcex;
        …
        wcex.lpfnWndProc = WndProc;
        RegisterClassEx(&wcex);

        // ablak létrehozása
        hWnd = CreateWindow(…);

        // ablak megjelenítése
        ShowWindow(…);
        UpdateWindow(hWnd);

        // Üzenetkezelő ciklus
        while (GetMessage(&msg, NULL, NULL, NULL)) {
                TranslateMessage(&msg);
                DispatchMessage(&msg);
        }

        return msg.wParam
}

// ABLAKKEZELŐ FV.
LONG CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam) {
        // lokális változók deklarálása
        // egy switch-case szerkezetben megvizsgáljuk milyen üzenetet
        // kapott a függvény paraméterként és az üzenettől függően csinálunk
        // vmit
        switch (msg) {
                case WM_CREATE:
                        ...
                return 0;
                case WM_PAINT:
                        DrawSomething();
                return 0;
                case WM_COMMAND:
                        switch (wParam) {
                                case IDM_FILENEW:
                                        // pl. egy menüelem azonosító
                                        DoSomething();
                                        break;
                        }
                        return 0;
                case WM_DESTROY:
                        FreeEverything(); // pl. memória felszabadítása
                        PostQuitMessage(0);
                        return 0;
                }
                return DefWindowProc(hwnd, msg, wParam, lParam);
```

25 – 26. Tétel

25. Ismertesse a QNX alapvető architektúráját és az üzenetek segítségével történő kommunikáció folyamatát!

26. Ismertesse a QNX Inter Process Communication lehetőségeit!

1. QNX Neutrino

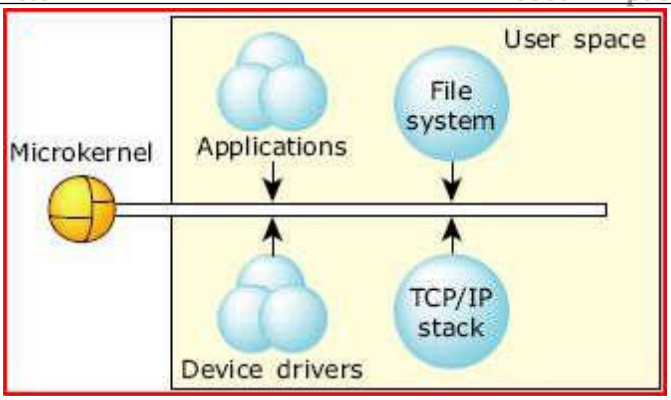
- jelenlegi generáció: QNX 6.x.x (6.4.0)
- OS neve: QNX Neutrino
- teljes rendszer neve: QNX Mommentics
- POSIX szabványt követ
  - egységes szemlélet
  - nem UNIX, de hasonló
- RT
- support:
  - beágyazott rendszerek
  - nagy teljesítményű elosztott rendszerek
- támogatott CPU-k:
  - x86
  - ARM
  - XScale
  - PowerPC
  - MIPS
- hordozható kód a verziók között
- többfelhasználós
  - virtuális konzol
  - futó folyamatok többfelhasználósak lehetnek

2. Mikrokernel

monolitikus ⇔ mikro

- **monolitikus**
  - egy mag ban az összes szükséges rendszerszolgáltatás (fájlrendszerek, kommunikáció, eszközkezelés)
  - közös memória terület
  - korai Linux kernelek (ma már moduláris, bár nem mikrokernel)
  - Windows NT kernel (performancia miatt. bár némi primitív mikrokernel-je van a drivereknek)
  - performancia
  - ha bukik, minden bukik → kékhalál → ☹
  - általában stabilabb, megbízhatóbb
- **mikrokernel**
  - a HW felett minimális réteg csak az OS magja
  - feladata:
    - alap rendszerhívások
    - memóriakezelés
    - processzekkel kapcsolatos műveletek
    - processzek közötti kommunikáció

- a rendszerszolgáltatások ugyanolyan szoftverek, mint a felhasználói programok
- előnye:
  - védeni lehet az OS-t és a különálló SW komponenseket
  - bármilyen szolgáltatás leáll, nem hal meg a kernel. Újraindítja, és működik minden tovább (bár így kisebb hibák simán elfedődhetnek a fejlesztő előtt, mert mindig megoldódnak)



- hátránya: túlzott számú rendszerhívás + túl gyakori kontextusváltás → nagy overhead

3. QNX mikrokernel

- **az OS magja egy busz**
  - mindenki erre kapcsolódik
  - mindenki ide küldi az üzeneteit
  - az SW komponensek egyenrangúak
- a kernel magától sosem kerül ütemezésre, csak …
  - közvetlen kernelhívás
  - exception
  - HW IRQ hatására
- alapvető működése:
  - üzenetkezelés
  - egyes üzenetekhez nem társít külön értelmezést, azokat csak a küldő-fogadó érti
  - feladata ezek szinkronizálása, ütemezése, erőforrások megfelelő kihasználása
  - párhuzamosan futó szálak
    - számítógépen belül és hálózaton keresztül is képesek kommunikálni egymással
    - bármilyen erőforrást használhatnak (helyi, hálózati)
- komplex rendszerek üzemeltetésére

4. Kernelszolgáltatások

Threads

- alapegység: process → szálak tárolója (egy processz – min. egy thread)
- egy processzhez tartozó összes thread egy memóriaterületen
- a threadeknek egyedi stackje van
- lehetséges állapotok
  - *CONDVAR* – feltételes változóra várakozik
  - *DEAD* – befejeződött, másik szál csatlakozására vár
  - *INTERRUPT* – megszakításra vár
  - *JOIN* – blokkolt és egy másik szálhoz való csatlakozásra vár
  - *MUTEX* – egy mutex miatt blokkolt
  - *NANOSLEEP* – alszik egy kicsit
  - *NET_REPLY* – hálózati válaszra vár
  - *NET_SEND* – hálózati jelre vár (pulse, signal)
  - *READY* – futásra kész, épp valaki magasabb prioritású dolgozik

- *RECEIVE* – üzenet érkezését várja
- *REPLY* – elküldött üzenetre vár választ
- *RUNNING* – épp fut
- *SEM* – egy szemafor küldésére várakozik
- *SEND* – üzenetet küldött, a kézbesítésre vár
- *SIGSUSPEND* – blokkolt állapotban van, egy jelre vár (sigwaitsuspend())
- *SIGWAITINFO* – blokkolt állapotban van, egy jelre vár (sigwaitinfo())
- *STACK* – a szál arra vár, hogy a hozzá tartozó stack le legyen foglalva (a szülő pl. ThreadCreate() fv-t hívott)
- *STOPPED* – a szál blokkolt és SOGCONT jelre vár
- *WAITCTX* – arra vár, hogy nem fixpontos tartalom használható legyen
- *WAITPAGE* – fizikai memórialapra vár
- *WAITHREAD* – a gyermek szál létrejöttére vár (ThreadCreate())

- **Ütemezés**
  - 256 prioritási szint
    - user processes: 0-53
    - idle: 0
  - azonos prioritási szinten
    - FIFO: a szál addig fut, amíg nem blokkolódik, vagy egy magasabb prioritású futásra késszé nem válik
    - Round-Robin: A szál addig fut, amíg nem blokkolódik, egy magasabb szál meg nem szakítja, fel nem használja az időszeletét
    - Sporadikus: megadja, hogy egy adott időtartamon mennyi ideig futhat a szál → ha ezt elhasználta, alacsonyabb prioritásúvá vállik, és a feltöltési idő után újra visszakapja a prioritását

Signals

Message transmitting

Thread Synchronization

- *Mutex* – kölcsönös kizárás – csak egy szál lockolhatja az erőforrást, a többiek várakoznak. Ha egy magasabb prioritású kezd el várakozni, ideiglenesen az aktuálisan foglaló prioritása afölé emelkedik – prioritás inverzió
- *Condvars* – feltételes változó, egy esemény/feltétel beteljesülésére átbillenő változó (mutex-el együtt kell használni)
- *Barriers* – sorompó. A szálaknak be kell várniuk egymást egy adott ponton. Az inicializáló szál adja meg, hogy hány további szálnak kell megérkeznie. Ha a feltétel teljesül, akkor minden szál egyszerre folytathatja a tevékenységét. Mátrixműveleteknél hasznos
- *Sleepon locks* – condvarhoz hasonló. Nem csak egy feltétel teljesülését figyeli, több feltételes változó értékét figyeli (változók száma max a várakozó szálak száma lehet)
- *Reader*/*Writer locks* – írás olvasás külön lockolása
- *Semaphores* – általános lefoglalás/felszabadítás szinkronizáció
- *FIFO scheduling* – nem SMP rendszerekre szálak szinkronizációjára. Egy prioritási szál addig fut, amíg egy magasabb prioritású meg nem szakítja, vagy blokkolt állapotba nem kerül. Nem tartalmaz semmilyen explicit szinkronizációs mechanizmust.
- *Send*/*Receive*/*Reply* (*Messages*)
- *Atomic operations*: Műveletsorok megszakítás nélkülivé tétel

Scheduling

Timer Services

Process Management – IPC – Inter Process Communication

- **Message parsing – szinkron üzenettovábbítás (kernel szolgáltatás)**
  - üzenetküldő (kliens) – fogadó (szerver)
  - üzenetküldési lehetőségek
    - egyszerű üzenet küldése: memóriacímmel + hosszúsággal
      - a rsz. nem foglalkozik az adatokkal, csak átmásolja a kliens memóriatartományáról a szerverére az adatot – a művelet szinte csak memóriakezelés → gyors
    - több részből álló üzenet küldése
      - I/O vektor az egyes üzenetrészek címével + hosszával
  - üzenetküldés folyamata: kliens küld egy üzenetet + blokkolt állapotba kerül, amíg nem kap receive-t → szerver fogadja, és válaszol rá (ha épp nem egy másik üzenetre vár, ami blokkolja)
  - hálózaton keresztül
    - QNX natív hálózata: Qnet
    - lassabb, de kényelmes
  - csatornák
    - szerver szálnak kell létrehoznia → kliens csatakozik hozzá
    - Pulse:
      - rögzített méretű adatok továbbítására (1 byte kód + 4 byte adat)
      - nem blokkolja a résztvevőket
      - interrupt handlerekben szeretik használni
      - prioritásöröklődés – prioritás inverzió elkerülése miatt
  - alapszabályok
    - két szál közvetlenül ne küldjön üzenetet egymásnak
    - a szálakat hierarchiába rendezzük, és mindig az alacsonyabb küldjön üzenetet a magasabbnak → deadlock elkerülése
- **Signals – jelek (kernel szolgáltatás)**
  - olyan, mintha SW megszakítás lenne
  - a jeleknek is handlerei vannak, amit a kernel hív
  - a végrehajtható feladaot, hívások korlátozottak (mert kernelben hajtódik végre)
  - jelenként különböző az alapértelmezett tevékenység
  - a kernel 64 jelet támogat (32 alapvető POSIC jelet)
  - 1 bájt kód + 4 bájt adat, ami várakozási sorba kerülhet (mint a Pulse)
  - működési szabályok
    - a jelek kezeléséhez kapcsolt tevékenység az egész folyamatra érvényes
      - lehet figyelmen kívül hagyás
      - lehet feldolgozás
    - maszkolásuk szál szintjén érvényes
    - közvetlenül szálnak küldött jel csak adott szálnak kézbesíthető
    - folyamatnak küldött jel az első olyan szálhoz kerül, amelyik nem blokkolja a jelet
- **POSIX message queues – (külső folyamat szolgáltatása)**
  - külső szerver folyamat végzi a kezelést
  - két szerver implementáció
    - mqueue – hagyományos erőforrás kezelő
    - mq – alternatíva

  - az üzenet sorok névvel rendelkeznek + elérhetőek az állományrendszereken keresztül is
    - /dev/mqueue/
    - /dev/mq/
- **Shared Memory – (külső folyamat szolgáltatása)**
  - legnagyobb sávszélességet biztosítja
  - hozzáférés asszinkron – szinkronizálni kell (szemaforral, vagy mutexel)
  - egy processzen belül az összes szál azonos memóriaterületen osztozik – elérik egymás területét
  - Process Manager intézi
  - használata
    - létrehozás → fájlleírót ad: shm_open()
    - méret kezdetileg nulla → be kell állítani: shm_ctl()
    - elkészült a memóriaszelet → be kell állítani, hogy elérhető legyen: mmap()
    - védelmi paraméterek
      - végrehajtható program?
      - írható? olvasható?
      - gyorsítótárba kerülhet-e a tartalma?
- **Pipes – (külső)**
  - IO csatorna folyamatok között
  - külső szoftverkomponens menedzseli
  - shell haszálja általában: egy parancs kimenetét egy másik bemenetére tegye
  - egyoldalú kommunikációk
- **FIFOs – (külső)**
  - hasonló a csövekhez
  - ugyanaz a szoftver kezeli
  - különbség: névvel rendelkeznek, elérhetőek a fájlrendszeren keresztül

Interrupt handling

- feladat: a külső eseményekre reagálás → késleltetés a rendszerben
  - minél gyorsabb
  - minél hatékonyabb
  - minél rövidebb ideig tartson
- Megszakítás késleltetés – Interrupt Latency
  - a megszakítás után nem azonnal hajtódik végre a lekezelés
  - függ
    - le vannak-e tiltva a megszakítások
    - éppen egy másik megszakítás fut
- Ütemezési késleltetés – Scheduling Latency
  - megszakítás kezelés után egy magasabb szálat kell végrehajtani (valamilyen esemény segítségével)
  - késleltetés a megszakítás kezelő utolsó utasítása és az eredeti szál első utasítása között eltelt idő

5. Rendszerszolgáltatások

Fejlesztői kernel

- kernel események megfigyelése valós időben
- minimális méretnövekedés
- 97-98%-a valós kernel sebességének
- naplózás

Többprocesszoros rendszerek

- akár diszkrét akár többmagos
- működési módok
  - aszimmetrikus – különböző OS-ek vagy azonos rendszer különböző példányai futnak egyes processzorokon
  - szimmetrikus – egy OS több CPU-n, processzek, threadek tudnak váltani az egyikről a másikra
  - csatolt többprocesszoros – az OS egy példánya kezeli a CPU-kat, de az alkalmazások nem tudnak futás közben váltani

Process Manager

- procnto SW modul
- feladatai
  - folyamatok kezelése (létrehozás, felszámolás …)
  - memória kezelése
  - névtér kezelése (azonos névtérben vannak a háttértárolók, perfiériák, rendszerkomponensek, rendszerszolgáltatások)

Dynamic linking

- hagyományos C tulajdonság – libary-k közös használata, nem statikusan mindenhova bemásolva

Resource Manager

- HW perifériák, virtuális eszközök kezelése
- nem igényel speciális kapcsolatot a maggal – leálítható
- támogatott erőforrások (ahogy ő bontja fel)
  - device
  - filesystem

Filesystems

- állományrendszerek kezelése a kernelen kívüli SW modulban
- ennek előnye:
  - egyes állományrendszerek igény szerinti elindíthatóság, leállíthatósága
  - általános kommunikációs megoldásokkal elérhetőek
  - általános névtéren keresztül elérhetőek az állományrendszerek kacsolódási pontokon keresztül
  - egy állománykezelőből több tucat futhat egyszerre
  - hálózton keresztüli elérés könnyen biztosított
- támogatott állományrendszerek
  - image – OS modulok (pl. boot image)
  - block – blokkorientált állományrendszerekhez (merevlemezek, optikai meghajtók)
  - flash – FFS2 for NOR devices, ETFS for NAND devices
  - network: távoli hozzáféréshez: NFS, CIFS
  - virtuális:
    - csomag
    - tömörített

Karakteres IO

- elérési útvonal állományként: /dev/
- eszközök csoportosítása/elérhető eszközök
  - soros vonali terminál
  - párhuzamos port (nyomtató pl.)
  - szöveges üzemmódú konzol

- pszeudo terminál

Hálózatkezelés

- kernelen kívüli app
- egységes interfész az összes áhlózati kapcsolathoz
- Qnet
  - natív
  - átlátszó hálózat – az egyes csomópontok szorosan kapcsolódnak egymáshoz
  - legfontosabb feladatai
    - állományrendszerek távoli elérése
    - alkalmazások rugalmas skálázhatósága
    - elosztott alkalmazások készítése
    - processzor erőforrás megsokszorozása
  - saját névfeloldás, de támogatja a TCP/IP DNS-t is
  - nem routeolható
  - bővítés: Internet Protocol-ba ágyazás
- TCP/IP hálózat