1. Szoftvertechnológia fogalma, humán aspektusok, beágyazott SW speciális tulajdonságai, szoftver minőség

Definíciók

- A **technológia**: műveletek sorozata, amely egy adott kezdeti állapotból adott erőforrások felhasználásával megismételhető módon egy kívánt végállapot eléréséhez vezet.

- **Erőforrások**: információ, anyagok, idő, munkaerő, szakértelem, eszközök -> pénz (többé kevésbé lehet konvertálni)

Szoftvertechnológia: A szoftvergyártás technológiája

- A SW terület viszonylag új
  - Kb. 60-65 év, ez semmi pl. az építőiparhoz képest (10e év)
- Nagyon sokszínű (sokféle szoftvert írunk)
  - Operációs rendszer, játék, CD/DVD nyilvántartó program Access-ben
- Nagyon flexibilis látszólag a hibás döntés kisebb kárt okoz vagy nem okoz kárt
  - Egy feladatot nagyon sokféle módon, nagyon sokféle eszközzel (pl. programozási nyelv) meg lehet oldani
  - Valójában a hibás döntésből származó károk súlyosak, csak nehéz a hibák okát megtalálni
  - Időben gyakran nagyon szétválik a hiba keletkezése és megjelenése (észrevétele)

Humán aspektus

- Az SW egyre fontosabb de van egy humán aspektus is…
  - Egyre több és egyre kritikusabb feladatot bízunk SW-re
  - A népesség egyre nagyobb része lesz programozó
    - A populáció képességei adottak: „Any idiot can write code”
    - A kérdés a minőség és költség…
- Képesség és motiváció: kb. 6 hónap alatt egy adott területen profivá válhat bárki az adottságok megléte esetén:
- Tanulság: „Soha ne félj váltani”, „A jó pap holtig tanul”, „Ha az elmúlt 1-2 hónapban nem tanultál újat, halott vagy (legalábbis szakmailag)”
- A mérnök is „emberből” van, még ha ezt mi mérnökök le is akarjuk tagadni (ráció/logika mindenek felett)
- Képességek és igények közötti ellentét
  - Kivételes képességű emberekkel sem könnyű
  - Átlagos képességűekkel sem
  - Sok emberrel meg különösen nehéz (szervezeti kérdések)

Beágyazott SW:

- Szoros kapcsolatban áll a hardware-rel
  - Hardver és szoftver együttes tervezése (Hardware/Software Codesign)
- Szoros információs kapcsolatban áll a külvilággal (beágyazó fizikai környezet)

    - Érzékelés és beavatkozás a fizikai környezetbe
- Biztonságkritikus környezet (járműipar, energetika stb.)
    - A termék és a fejlesztési folyamat minősítése
- Alacsony felhasználói hibatolerancia
- Kívülről látszólag egyszerű működés jellemző rájuk de belül komplexek, különösen a beágyazó fizikai környezettel együtt
- Szoftver és annak környezete stabil
- Költségérzékeny: maga az eszköz olcsó, a hiba javítása drága (visszahívás)
- Eltérő fejlesztési módszerek
    - Fejlesztő eszközök mások, keresztplatformos fejlesztés (PC-én fejleszt, MCU-n tesztel)
    - Oszcilloszkóp, logikai állapot analizátor, stb. az SW fejlesztő asztalán
- Tesztelés nagyon nagy része a fejlesztésnek (70-90%...)

Szoftver minőség

- Funkcionális minőség (functional quality)
    - Megfelel-e az elvárt működésnek (fitness for purpose)
    - A tesztek ezt vizsgálják
    - A kód belsejéről (hogyan?) nem mond semmit
- Strukturális minőség (structural quality)
    - Nem funkcionális követelmények, segítik a funkcionális követelmények teljesítését
    - Robosztusság, karbantarthatóság, áttekinthetőség, stb.
    - Kód, kód struktúra analízise, software architektúra
- Minden tudományág pontosan olyan fejlett, mint az általa használt mérőeszközök
    - "A science is as mature as its measurement tools," Louis Pasteur
    - Hogyan mérjük a szoftver minőséget?
- Motiváció: Kockázatok és/vagy ár kézbentartása (risk and/or cost management)
    - Kockázat centrikus: pl. beágyazott rendszerek
    - Ár centrikus: pl. üzleti szoftver

Minőség jellemzők:

- Megbízhatóság (reliability):
    - Potenciális meghibásodás
- Hatékonyság (efficiency):
    - Futási időben milyen hatékony,
    - Mennyi erőforrást használ
- Biztonság (security)
    - Illetéktelen hozzáférés, stb.
- Karbantarthatóság (maintainability)
    - Módosíthatóság, portolhatóság, fejlesztői csapatok közötti mozgathatóság
- Méret (size)
    - A fentiekkel kombinálva kell figyelembe venni
    - Tipikusan “felfedezett hiba / (kódsor * időegység)” jellegűek lesznek a mérőszámaink
    - Nagyobb szoftvert nehezebb írni
        - Csak a kezdők számára vonzók a nagy projektek…

2. Komplex SW rendszerek problémái és azok kezelése, előre gyártott elemek a SW fejlesztésben

- A komplex rendszerek többnyire hierarchikus felépítésűek
    - Rendszer - alrendszer -…- alrendszer – elemi komponens
        - Kérdés az elemi komponens, pl. newtoni fizika versus kvantumfizika?
        - Az elemi komponensek halmaza absztrakciófüggő
            - nézőpont és cél kérdése az elemi komponens pl.: adatbázisnál: a használó rekordokat, a programozó I/O műveletet lát
- A világ hierarchikus felépítésű, és ennek megfelelően ezt az emberi gondolkodás és képes jó kezelni…
    - Fel kell bontanunk a problémát, hogy meg tudjuk érteni
    - Többnyire tudatosan hierarchikus modelleket alkotunk
- A komplex szoftver rendszerek bonyolultak…
    - De miért?
    - Eddig:
        - Hierarchikus felépítés
        - Absztrakció függő elemi komponensek
- A problémáink miből származnak?
    - A megoldandó probléma bonyolultsága
    - A fejlesztési folyamat kézbentartásának nehézségei
    - A szoftver eszközök rugalmassága
        - Megfelelő nyelv kiválasztása, stb.
    - A diszkrét rendszerek leírásának nehézségei
        - Állapottér, algoritmus ezen hogy operál

A megoldandó probléma bonyolultsága

- Projekt elején ki kell dolgozni a követelményeket
    - A leendő felhasználóval/megrendelővel együtt
        - Emberekről van szó, pszichológia…
            - Kommunikációs problémák
            - A megrendelő és megvalósító között van egy természetes és majdnem mindig meglévő ellentét (a szintje ennek eltérő)
- A követelményrendszer bonyolult
    - Nagyszámú követelmény
    - Magában bonyolult követelmények
    - Inkonzisztens (ellentmondó) követelmények
- A követelményrendszer rögzítésének a nehézségei
    - Régebben egy nagy dokumentum volt…
    - Az emberi felfogóképesség véges
    - A követelményrendszer formalizálása és kezelésének IT eszközökkel történő segítése egyre gyakoribb…

A fejlesztési folyamat kézbentartásának nehézségei

- Megrendelő és megvalósító is esendő ember…
- Megrendelő
    - Saját szakterületének nyelvét beszéli
    - Amelynek kifejezéseit a megvalósító félre is érthető (ugyan az a szó ott mást jelent)
    - Sokszor maga sem tudja, mit akar
- Megvalósító
    - A megvalósításra gondol, programban gondolkodik
- Együtt, folyamatosan kommunikálva, egy iteratív folyamatban kell a teljes fejlesztési folyamatot levezetni

Előregyártott elemek alkalmazása

- SW komponensek
    - Jól definiált funkciók és interfészek
    - Direkt módon beépíthető a programba
    - Sokan használják – vagyis jól tesztelt
        - Vagy legalábbis azt hisszük
- Példakód
    - Hogyan kell csinálni, Copy&Paste szintjén
    - Pl. hogyan kell egy SW komponenst használni
- Minták

3. Programozási paradigmák és azok fejlődése, strukturáltság, tipikus programozási paradigmák beágyazott rendszerekben

Programozási paradigmák:

- A programozás alapvető módja, stílusa, a program belső felépítése kerül megadásra
- Egy időben egy feladathoz akár több paradigmát is felhasználhatunk

- Alapparadigmák:
    - Procedurális programozás (imperatív általánosan)
    - Deklaratív programozás
- Kiegészítő paradigmák
    - Eseményvezérelt programozás
    - Párhuzamos programozás (több szál, GPGPU, beágyra nem jellemző)

Procedurális programozás (Pl focista)

- HOW? kérdés
- Algoritmus…
- Lebontani részproblémákra, a megoldás lépéseit adjuk meg, utasítások szekvenciális sorozata
- A matematikai függvény fogalomból származik
- Moduláris, strukturált, kód újrafelhasználás, „goto” száműzése
- Nagyon mélyen a processzor is procedurális működésű
- Az objektum-orientált programozás ennek a kiterjesztése
- Pl: Numerikus feldolgozás (DSP), beágy rendszerek

Deklaratív programozás (Pl bíró)

- What? Kérdés
- Szabályok
- A feladat megoldásának a logikáját adják meg a megoldás részletei (control flow) nélkül
- Formális logika és következtetések
- Pl: Fordítóprogramok, Db kezelés (SQL)

Nyelvek fejlődése

- 0. generáció: bináris kód: gépi utasítások bináris bevitele
- 1.genereáció /54 - 58/: matematikai kifejezések kezelése
  - Fortran I, Algol 58
  - Gépi típusok használhatók
- 2. generáció /59 - 61/: adattípus, eljárás fv fogalma
  - Fortran II, Algol 60
- 3. generáció /62 - 70/: 2. generációs nyelvek finomítása, összetett adattípusok:
  - Algol 68, Pascal
- Generációs rés: PDP-11: könnyebb számítógép hozzáférés, evolúciós sokszínüség,
  - C nyelv, UNIX OS
- 4. generáció: objektum orientált nyelvek: Smalltalk, C++, Java, C#, deklaratív
  - nyelvek: SQL, Prolog

Struktúráltság

- 0: nincs struktúra
- 1-2: adattípus + műveletek
- 3: modularitás
- 4: adat és rajta végezhető műveletek szoros egymásba kapcsolása

4. Programozási nyelv korlátozásának okai, megoldásai, szabályok, a MISRA C alapjai; MISRA C deklaráció és inicializáció

A probléma:

- A programozási nyelvek legtöbbje elég szabados, tehát a szintaktikailag helyes utasítások némely esetben okozhatnak szemantikai problémákat.
- Megbízhatóság / biztonságkritikus rendszereknél csökkenteni kellene az ilyen problémák előfordulásának lehetőségét
- (rakéta, gyorsulásérték tároló változó túlcsordult)

Megoldás:

- Megjelennek a Safetyszabványok
- A Safety szabványok különböző kategóriákba osztják az eszközöket
- SIL (Safety Integration Level), ASIL (Automotive SIL)
  - SIL0: Könnyű személyi sérülés
  - SIL1: Könnyű személyi sérülés, nagyon ritkán visszafordíthatatlan sérülés, vagy halál
  - SIL2: Egy ember halála, vagy ritkán több ember halála
  - SIL3: Számos ember halála
  - SIL4: Tömegkatasztrófa

- Cél a C szabados programozói nyelv leszűkítése úgy, hogy a lehető legmegbízhatóbb nyelvi alrendszert kapjuk.
    - Szintaktikailag helyes, szemantikailag rossz megoldásokra felhívni a figyelmet.
    - Tiltani a nem egyértelmű változó típus használatot.
    - Tiltani a nem strukturált programozást eredményező szerkezeteket.
- Iparban legelterjedtebb korlátozó szabálykészlet a MISRA (Motor Industry Software Reliability Association)
    - C korlátozáson kívül tartalmaz még
        - Simulink Stateflow guideline-okat
        - Automatikus kódgenerálásra vonatkozó szabályokat
        - C++ -ra vonatkozó szabályokat

MISRA C:

- A szabálykészlet a C90 és C99 szabványokat ismeri
- A program be kell, hogy tartsa az alkalmazott szabványkészlet lehetőségeit és megkötéseit
- Kerülni kell a nyelvi kiterjesztések használatát

Néhány példa:

- A project nem tartalmazhat elérhetetlen kódot
- Nem lehet halott kód (dead kód) a programban (végrehajtódik, de a törlése nem befolyásolja a futást)

Deklarációk:

- Minden esetben amikor változót, vagy függvényt definiálunk, azt teljes precíz megadással kell tennünk
- Minden függvénynek kell, hogy legyen prototípusa, ami látszik a függvény megvalósítás és függvény hívás számára is. A megvalósítás és a prototípus definíciónak pontosan meg kell egyeznie.
    - Header file-ban a prototípus, amit aztán mindenki #include-ol.
- Header file nem tartalmazhat definíciót, csak deklarációt. Definíciót mindig C file-ban kell megírni.

Inicializáció:

- Az összes automatikusan foglalt helyű változót inicializálni kell olvasás előtt.
- Többelemű tömbök, struktúrák inicializálásakor {}-ekkel kell jelölni a határokat inicializáláskor is. (y[2][3] = {1,2,3,4,5,6} rossz, y[2][3] = {{1,2},{3,4},{5,6}} jó)
- Tömböket nem szabad részlegesen inicializálni
- Egy elemet csak egyszer szabad inicializálni

5. A MISRA C alapjai, esszenciális típus modell, konverziók, vezérlési szerkezetek, stb.

Esszenciális típus modell

- A változó konverzió egy alapvetően veszélyes terült. Változó konverzió a következő problémákkal járhat:
    - Értékvesztéssel, ha a konverziós típus nem tudja a konvertált típus teljes értéktartományát megjeleníteni.
    - Előjelvesztéssel

    - Precízió vesztéssel
- A típus konverziókat szabályozni kell
- A modell táblázatos formában megadja, hogy milyen típusokkal milyen alapművelet végezhető
    - Pl.: Float típus
        - nem használunk tömb indexnek
        - nincs biteltolást
        - nincs logikai ÉS, VAGY … művelet rajta
    - Boolean
        - Nincs kisebb nagyobb összehasonlítás és biteltolás

- Egy kifejezés eredménye nem adható át automatikusan egy szűkebb értéktartományú elemi típusnak, vagy más elemi típusnak
- Egy aritmetikai műveletnél azonos alaptípusokat várunk el
- Egy kifejezés eredménye nem konvertálható implicit nagyobb értéktartományú esszenciális típusra

Konverzió:

- A MISRA-C a következő változó konverziós szabályokat adja meg implicit konverzióra:
    - Nincs implicit konverzió signed és unsigned típusok között
    - Nincs implicit konverzió egész és lebegőpontos típusok között
    - Nincs implicit konverzió nagyobb tartományról kisebb tartományra
    - Nincs implicit konverzió függvény argumentumoknál
    - Nincs implicit konverzió return értékeknél
    - Nincs implicit konverzió komplex aritmetikai kifejezéseknél
- Explicit konverzió akkor használható komplex aritmetikai kifejezés esetében, ha kisebb értéktartományú típusra konvertálunk, de az előjelet megtartjuk.
- Konverzió probléma:
    - uint16_t + uint16_t = uint32_t -> az aritmetikát nem az eredmény típusa, hanem a compiler belső aritmetikája határozza meg, így lehet túlcsordulás
    - Megoldás castolás

Vezérlési szerkezetek

- A nullával való egyenlőséget vizsgálni kell, hacsak a kifejezés nem Boolean típusú
- Lebegőpontos típust nem lehet egyenlőségre, vagy nem egyenlőségre tesztelni, mert annak eredménye architektúra és compiler implementáció függő
- For ciklus kifejezései nem tartalmazhatnak lebegőpontos típust.
    - Kerekítési, levágási hibák problémákhoz vezethetnek.
- Ciklusváltozót nem lehet a ciklus belsejében módosítani
- Invariáns eredményű boolean kifejezést nem szabad használni
    - Pl.: int < 0…
- A gotohasználata nem javasolt
    - Csak lokálisan ugorhat függvényen belül, akkor is csak ha nagyon muszály

- Az iterációknak egyetlen break kiszálló pontja legyen
- A függvényeknek egyetlen kiszálló pontja kell, hogy legyen
- Egy for, while, case szerkezet mindenképpen { … } kell hogy használjon.
- Egy if szerkezet mindkét ágát { … } kell tenni, ez alól kivétel, ha az else ágat rögtön egy másik if követi.
- Az else – if sorozat mindenképpen else ággal kell záródjon.

Függvények:

- Egy függvény nem hívhatja meg saját magát. Tehát nincs rekurzió
- A deklarációnak és a definíciónak meg kell egyeznie
- Nem voidfüggvényeknek explicit módon megadott return kulcsszóval kell rendelkezniük.

6. Objektum orientált programozás alapjai, osztály és objektum viszonya, objektum részei, hivatkozás, tartalmazás, objektum orientáltság ismérvei, objektum orientáltság a JAVA-ban

OOP:

- A programozási feladat megoldása olyan módon, hogy a létrejövő objektum orientált program (többnyire maga is egy objektum) együttműködő objektumok összessége.

Objektum:

- Objektum = tulajdonságok + az objektumon végezhető műveletek összessége
  - Egységbe foglalva
  - Tulajdonság: belső adatstruktúrák
  - Műveletek: a belső adatstruktúrákon operáló függvények
  - Az objektum tartalmazhat objektumokat
    - Egy objektum tulajdonsága maga is lehet objektum
    - Pl. egy autónak van motorja, ami magában egy complex objektum
  - Az objektumot más objektumok tartalmazhatják
    - Ezt az objektum írójának tudomásul kell vennie
  - Az objektum hivatkozhat más objektumokra
  - Az objektumra hivatkozhatnak más objektumok

Objektum vs osztály:

- A program alapvető építőköve
  - Független összetevő
  - Lehetőleg rejtett belső állapotai vannak
  - Saját viselkedés, működése van
  - Külső objektumok hatnak rá (a belső állapot megváltozik)
- Az objektum egy vagy több osztályba (class) tartozik
  - Az osztály absztrakt adattípus
  - Az osztály példánya (instance) az objektum
  - Az osztályból létrejön az objektum
    - Lesz élettartama is az objektumnak
  - Az osztály tervezési idejű koncepció

- Az objektum futási idejű koncepció

Objektum részei:

- Belső állapot (tulajdonságok, attribútumok, tagváltozó)
- Viselkedés (metódus, tagfüggvény)
- Élettartam (lifecycle)
- Interfész

- Vagyis: Az objektum tulajdonképpen felfogható egy „állapotgép”-nek, van egy belső állapota, és a hozzá tartozó metódusokon keresztül változtatható meg ez a belső állapot.

Tartalmazás:

- Összetétel, tartalmazás (aggregation)
  - Kutya és az orra
- Irányított reláció, az egész tartalmazza a részt
- Megvalósítás:
  - Érték szerinti tartalmazás: Fizikailag egymásba ágyazott memóriaterület (a rész az egész memóriaterületében jön létre)
  - Referencia szerinti tartalmazás: Az egész egy referenciát (hivatkozást) tartalmaz a külön (saját) memóriaterületen létrejövő részre

Hivatkozás (referencia):

- Referencia ≠ pointer
  - Referencia: általános, a konkrét implementációt nem adja meg
  - Pointer: Memória cím, amin a hivatkozott objektum megtalálható
- A pointer a referencia egy megvalósítása C/C++ nyelveken (és egyes más nyelveken)

Objektum orientáltság ismérvei:

- Információ rejtés/összefogás (Encapsulation, Information Hiding)
- Öröklés (Inheritance)
- Polimorfizmus (Polimorphism)

Objektum orientáltság a JAVA-ban:

- Tényleges
- Mindenki az Object-ből származik
- Primitív vs Referencia típusok
  - Van csomagoló osztály a primitív típusokhoz

7. Az objektum orientáltság ismérvei részletesen bemutatva, példákkal. Objektum típusa, a típus által hordozott információ bemutatása, polimorfizmus és a típus kapcsolata

Információ rejtés

- Cél:
  - Az elkészült osztályt az azt felhasználó programozó úgy tudja használni (egyedeket létrehozva), hogy annak interfészét elég megismernie, nem szükséges megismernie a belső implementációt.

    - Lényegében az elkészült osztályok újrafelhasználását segítjük ezzel
    - Tehát itt egy tervezési idejű információ rejtés valósul meg!Futási időben a futtató környezet ellenőrizheti, hogy az információ rejtés helyesen valósul-e meg...
- Megvalósítás:
    - public, protected, private (mindenki, öröklés, csak az osztály)
    - Setter, getter metódusok
- A modern programozási nyelvek típusosak
    - Adattípus:
        - Jelölés: Egyértelműen megadja, hogy a változó milyen típusba tartozik…
    - Értékkészlet: Lehetséges értékek halmaza
    - Műveletek: Az adattípuson végezhető műveletek
    - Az adattípus fontos információt hordoz:
        - A programozónak ismernie és használnia kell
        - A fordító tudja, a fejlesztőrendszer (IDE) ki tudja találni a kód alapján
    - Az objektum típusa az osztály

Öröklés és polimorfizmus

- Az osztályokat mindig osztályokból származtatjuk
- Létezik egy ősosztály, ennek nincs szülője (csak úgy létrejön, szűznemzés…)
- Egyszeres és többszörös öröklés
- Öröklési fa, a gyökérben van az ősosztály
- A polimorfizmus (sokalakúság?) az öröklődésen alapul
    - Egy objektum minden olyan osztálynak a példánya, amiből örökölt
    - A közös ősből származó minden objektum kezelhető a közös ős egy példányaként
    - Példa:
        - Rajzobjektumból származtatunk pontot, vonalat, sokszögeket, kört, stb.
        - Mindegyik egyben rajzobjektum, vagyis pl. felhelyezhető egy rajzobjektumokat tartalmazó listára
        - Ha a rajzobjektumnak van egy move(dx,dy) metódusa, mindegyiknek (pont, vonal, sokszögek, kör, stb.) van ilyen metódusa (és az tudja hogy kell azt mozgatni), és a rajzobjektum listán iterálva azok meghívhatók. Így pl. egy összetett rajzobjektum is mozgatható (annak is van move()-ja, ami meghívja a részek move()-ját egymás után)
        - Nagyon hatékonnyá teszi a rajzobjektumok kezelését

8. Objektum részei, állapot, viselkedés, élettartam, interfész, JAVA példák

Részek:

- Belső állapot (tulajdonságok, attribútumok, tagváltozó)
- Viselkedés (metódus, tagfüggvény)
- Élettartam (lifecycle)
- Interfész

Belső állapot:

- Típusos osztály- és egyedváltozók
- Osztályváltozó
    - Összes egyedben ugyan az a memóriaterület található meg

    - Ugyan az az értéke az összes egyedben
- Pl. JAVA static kulcsszó
- Példa: Egyedek számának megszámlálása
- Egyedváltozó

Viselkedés:

- Típusos osztály- és egyedmetódusok
    - Osztálymódszer
        - „Static” kulcsszó a JAVA-ban
    - Csak osztályváltozókat és más osztálymetódusokat használhat
- Egyedmódszer
    - Egyed- és osztályváltozókat használ
        - Egyed- és osztálymetódusokat hívhat
        - Létrehozhat lokális változókat
        - Többértelműség (overloading)
            - Több ugyan olyan nevű metódus eltérő nyommal
                - Nyom = (visszatérési érték típusa) + metódus neve + paraméterek típusa

Élettartam:

- Konstruktor
    - Alapértelmezett (default) konstruktor
    - Copy konstruktor (deep copy, a shallow copy csak új referenciát hoz létre az objektumra)
    - Egyéb paraméterezett (inicializáló) konstruktorok
        - Egyes vagy az összes tulajdonságot megadhatjuk
- Megszüntetés
    - Explicit felszabadítás (C++ destructor, free utasítás hatására)
        - Destructor() metódus meghívódik
            - Hatékony, egyszerű
            - Hibalehetőség:
                - Elfelejtjük felszabadítani, túl korán szabadítjuk fel, nem jót szabadítunk fel
                - Memory leak (memória szivárgás)
                    - Példa, akár egy könyvtárban is lehet (példa: Win CE hiba)
                - A fejlesztőeszközök eszközöket adnak a memory leak-ek keresésére (heap nyomkövetés)
    - Automatikus szemétgyűjtés (JAVA, finalize() metódus szerepe fontos)
        - A memória automatikus felszabadítása a nem használt objektumok esetén
            - Pl. „reference count” alapon
            - Erőforrás igényes, valós idejű alkalmazása megkérdőjelezhető
        - Hogyan zárjuk le az objektum által használt erőforrásokat (csak az objektum tudja, mit használ)?
            - JAVA finalize() metódust a környezet meghívja, mielőtt az objektumot megszüntetné

Interface:

- Az osztály rendelkezzen egy adott működéssel
  - Adott metódusokkal, pl. serialize()/deserialize()
- Szabványos interface az őt használni kívánó objektumok felé
- Hogyan kényszeríthető ki?
  - C++: absztrakt osztályból örökölhető többszörös örökléssel
  - Absztrakt osztály: Nem példányosítható…
- JAVA: Serializable interface
  - … implements Serializable
  - JAVA-ban jobban szét van választva az interface és az osztály
  - Nincs többszörös öröklődés

9. Perzisztens objektumok, JAVA példák

Élettartam valós-idejű rendszerekben:

- Egyik megoldás sem jó…
  - Explicit megszüntetés veszélyes
  - A szemétgyűjtés valós-idejű rendszerekben problémás (nyitott kutatási terület)

Megoldás a perzisztens objektum:

- Tranziens objektum: constructorral létrejön, destructorral megszüntetjük egy programon belül
- Ez sokszor nem elégséges: Perzisztens objektum
  - Túlélheti az őt létrehozott programot vagy átkerülhet más programba
  - Új követelmények:
    - Típushelyes visszaállítás (akár más nyelvben, platformon)
    - Verziókövetés (különböző verziójú objektumok ugyan abból a típusból, pl. Word verziók dokumentum formátumai)
    - Platform-függetlenség
- Cél
  - Tárolás
    - Pl. file, adatbázis
  - Programok/gépek közötti továbbítás
    - Pl. copy&paste, hálózati másolás működő programok között (esetleg real-time módon), távoli elérés

Tárolás:

- Objektum-orientált adatbáziskezelők (OODBMS)
- Relációs adatbázisok
- XML (eXtensible Markup Language)
- Sorossá tétel (serialization)
  - JAVA: Serializable interface
    - … implements Serializable

10. Objektumok és osztályok közötti kapcsolatok, sablonok, JAVA példák

Osztályok/objektumok közötti kapcsolatok:

- Eredeti cél:
  - Komplex probléma hierarchikus dekompozíciója

- Együttműködő objektumokként képzeljük el a megoldást
  - Az együttműködéshez kapcsolatok kellenek…
- Kapcsolatok:
  - 1. Öröklés (inheritance): általánosítás/specializáció
  - 2. Tartalmazás (aggregation): rész ⇒ egész (kutya és a lába)
  - 3. Társítás (association): laza kapcsolat (kutya – harap – postás)

Sablonok:

- Az öröklődés sokszor nem alkalmas az általánosítás és a specializáció megfelelő leírására
- Pl. hasonló viselkedése, de eltérő adattípusokon operáló különböző osztályok
  - Pl. int-et kezelő FIFO, double FIFO, speciális osztályokat kezelő FIFO-k
  - Nem írható le hatékonyan örökléssel a típusvizsgálat megtartása mellett
    - Durva futási hibák helyett fordítási hibák
  - Sablon(Típus) → Típusos Osztály → Típusos egyed
  - Pl. tárolási osztályok (container class)
    - C++: Standard Template Library (STL)
    - JAVA: Generics

11. Párhuzamos eseményvezérelt programozás alapfogalmai

Definíció: A programozási feladat megoldása olyan módon, hogy a létrejövő párhuzamos program egymással párhuzamosan futó, együttműködő feladatok együttese.

Nagy kérdés:

- Erőforrás megosztás és gazdálkodás:
  - Megosztás: az erőforrást többen használnák párhuzamosan, a helyes használat a kérdés
  - Gazdálkodás: erőforrások optimális kihasználása a kérdés

Alapfogalmak:

- Szekvenciális program
  - A szekvenciális program kódja szekvenciálisan (egymás után) végrehajtott utasítások sorozata, amit az utasításokban felhasznált adatok vezérelnek (ciklusok, elágazó utasítások)
- Párhuzamos program/rendszer
  - Olyan rendszer amelyben több, egymással együttműködő, párhuzamosan futó részfeladat együtt valósítja meg a teljes feladatot…
  - Részfeladat, feladat (Task, Subtask)?
    - A párhuzamos rendszerekben egymással párhuzamosan futó, önmagukban szekvenciális programrészeket, amelyek egymás nélkül többnyire értelmes működésre nem képesek, részfeladatoknak nevezzük
    - A végrehajtásához végrehajtó egységhez kell rendelődnie
- Eseményvezéreltség
  - Esemény rendszer életében lezajló változás vagy történés
  - A program lefutása függ a program működés során lezajló történésektől (esemény)
    - Esemény/jelzés (event, signal)
    - rendszer eseményekre reagál
      - Reaktív rendszer

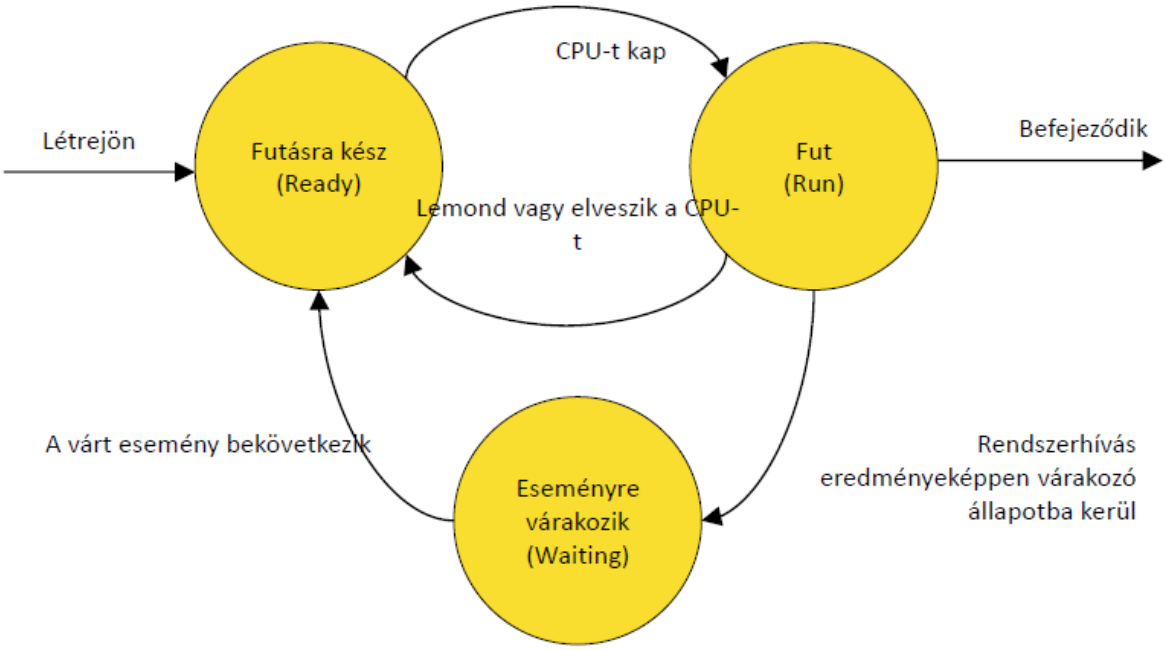
        - A válaszidő fontos
    - Esemény lehet:
        - HW megszakítás:
            - Input a felhasználótól (egérkattintás)
            - Input a külvilágból (új mért érték, detektált meghibásodás pl. az irányított folyamatban)
            - Adatcsomag beérkezése a kommunikációs interfészen
        - SW megszakítás: A program belső állapotában lezajló tudatos változás (rendszerhívás, system call)
        - Kivétel (exception): Nullával való osztás, túlcsordulás, stb.
    - Az esemény összetett adattípusként (struktúra vagy objektum) jelenik meg a kódban

12. Feladat fogalom, feladat állapota, feladat ütemezés, egyszerű ütemezési algoritmusok, freeRTOS elemzése ilyen szempontból

Feladat (task):

- A feladat több mint a program:
    - Aktív, nem passzív!
    - Állapota van (hiszen egy futó program):
        - Minimálisan: Fut (run), vár (waiting), futásra kész (ready).
        - Erősen OS függő állapotok és állapotátmenetek.
- Adatszerkezetek adattal vannak feltöltve (függ a feladat előéletétől):
    - Adat terület (globális adatok).
    - Verem (stack),
    - Halom (Heap).

Feladat állapota:



Feladat ütemezés:

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

- A feladatokat a többnyire feladat sorokban (task queue vagy job queue) tároljuk
    - Ez a legegyszerűbb esetben tényleg egy FIFO, ami TCB típusú objektumokra/struktúrákra mutató pointereket tartalmaz.
        - Pl. futásra kész sor, adott eseményre vár sor, stb.
    - Az ütemező algoritmus ezeken a struktúrákon dolgozva végzi el a feladatát (elsősorban a futásra kész soron).

Ütemezési algoritmusok:

- Preemptív ütemezés
    - Tervezési kérdés, hogy a futó feladattól elvehető-e a CPU.
        - Preemptív: A futó feladattól az OS elveheti a CPU-t
            - Ez a tipikus a modern operációs rendszerekben.
            - Bizonyos kernel feladatokra nem feltétlenül, azok nem megszakíthatóak, ennek vannak következményei (pl. valós idejű működés nehezen biztosítható)
        - Nem preemptív vagy kooperatív
            - Pl. Windows 3.x
            - A futó feladatnak le kell mondania a CPU-ról vagy eseményre kell várnia, ahhoz hogy más feladat tudjon futni.
            - Az egyes alkalmazásoktól függ a teljes rendszer működése.
                - Lényegében az alkalmazás programozójától, ami nem jó ötlet…
            - Ha a futó feladat hibás (végtelen ciklus), akkor a teljes rendszer működésképtelenné válik.

13. Összetett ütemezési algoritmusok, idő és kezelése

Feltételezések:

- A rendszerben egy végrehajtó egység van
- Egy időben egy feladat tud futni
- A feladataink CPU és I/O löketekből (burst) állnak

Ütemezési algoritmusok:

- Legrégebben várakozó (FIFO, FCFS)
    - A legegyszerűbb algoritmus:
        - Feladat leíróra mutató referenciákat tároló FIFO (FirstInput FirstOutput) sor az implementáció
            - put()-talbekerül a folyamat a várakozó sorba
            - get()-tellevesszük a futásra kerülőt
            - FCFS (First Come First Serve)
    - Nem preemptív definíciószerűen
        - I/O-ra várhat!
    - Az átlagos várakozási idő nagy lehet, és erősen függ a CPU és I/O löket (burst) nagyságoktól
        - Ez egyben átlagosan nagy válaszidőt is jelent, on-line felhasználókat is kiszolgáló rendszerek számára nem alkalmas
    - Kis adminisztrációs overhead

    - Tiszta CPU használat:
        - A később beérkező feladatoknak ki kell várniuk a korábban beérkező feladatokat
        - Hosszú feladat sokáig feltartja az utána következőket
    - CPU és I/O löketek:
        - Ha van olyan feladat, aminek hosszú a CPU lökete, akkor az fel fogja tartani a kis CPU lökettel rendelkező feladatokat
            - A kis CPU lökettel rendelkező feladatok a gyakori I/O löket miatt gyorsan a sor végére kerülnek
            - A nagy CPU lökettel rendelkező még a sor végéről is hamar előre kerül, aztán feltartja a többit
- Körforgó (Round-robin)
    - Kedvezőbb az interaktív (külvilággal kapcsolatban lévő) feladatok számára
        - Jobb az átlagos válaszideje a FIFO-nál
        - Adott időnként garantáltan vált, függetlenül a feladattól
    - Preemptív:
        - Lényegében a FIFO kiegészítése egy egyszeri óra megszakítással
            - Quantum vagy timeslice (időszelet)
            - 100-1 ms, tipikusan 10-20 ms
            - Elsősorban elfogadható interaktív feladatokra vonatkozó válaszidő biztosítására
    - A feladat futtatása előtt az órát elindítjuk, az egy adott idő (időszelet) lejárta után megszakítást kér (fut az OS)
        - Ha a feladat befejeződik vagy I/O-t hajt végre az óra megszakítás előtt (fut az ütemező):
            - Újraütemezünk, és az órát újraindítjuk
        - Ha a feladat nem fejeződik be:
            - A megszakítás beérkezik, és fut az ütemező
            - A feladat futásra kész állapotba kerül, újraütemezünk és az órát újraindítjuk
    - A gyakorlatban sok esetben az egyszerűség kedvéért periodikus óra megszakítást alkalmaznak
    - Időszelet > átlagos CPU löket -> Átmegy FIFO-ba
    - Időszelet ~= átlagos CPU löket -> Normális működés
    - CPU löket > Időszelet, ami viszont összemérhető a kontextus váltás időtartamával:
        - Nagyobb adminisztratív terhelés
    - Ökölszabály: A CPU löketek 80%-a kisebb az időszeletnél
- Prioritásos ütemezők
    - Ütemező család
    - A feladatokhoz prioritást rendelünk:
        - Külső/Belső prioritás:
            - Külső prioritás: Operátor vagy a feladat maga állítja be
            - Belső prioritás: A rendszer adja
        - Statikus/Dinamikus prioritás:
            - Statikus prioritás: Tervezési időben dől el, állandó a futás során
            - Dinamikus prioritás: Futási időben dől el, változik a rendszerben lezajló változások hatására

        - A gyakorlatban lehet ezek kombinációja is...
    - Jellegzetesen preemptív, de lehet nem preemptív is
    - Nem korrekt (fair), nem is akarjuk, hogy az legyen...
    - Kiéheztetés előfordulhat (Előny vagy hátrány?):
        - Nem pontosan a fontosság figyelembe vétele miatt használjuk?
        - Ha előfordul, akkor az alapvetően vagy túlterhelés miatt, vagy tervezési hiba miatt történik
        - A magas prioritású feladatok egy időben a feladat készlet egy kis, biztosan futtatható részét tehetik ki
            - Hagynak CPU-t az alacsony prioritású feladatoknak is
        - Nem lehet minden feladat egyformán fontos!
        - A feladatok öregítése segíthet (régóta vár, egyre fontosabb)
    - Egyszerű prioritásos ütemező:
        - Egy prioritási szinten egy feladat
        - Egyszerű beágyazott rendszerekben használják
    - Egy prioritási szinten tetszőleges számú feladat:
        - Módosításokkal széles körben alkalmazásra kerül
            - UNIX, Windows, stb., szinte minden operációs rendszerben ilyet találunk
        - Módosítások:
            - Egy szinten belül a futásra kész feladat kiválasztására használt algoritmust meg kell adni, tipikusan RR
            - Dinamikus, belső prioritás meghatározás
    - Futási idő figyelembe vétele egyszerű ütemezőkben:
        - Shortest time remaining (STR)
        - Least slack time scheduling (LST, Least Laxity First néven is)
    - Többszintű sorok (multilevel queue)
        - Minden egyes prioritási szinthez „futásra kész” várakozási sor
            - A prioritás meghatározásáról nem mond semmit...
            - Az egyes szinteken alkalmazott ütemezési algoritmusról nem mond semmit (FIFO, RR, stb.)
                - Implementáció függő

Idő:

- Számos ütemező algoritmusnak követnie kell az idő múlását
    - Round-robin, STR, LST
    - Időmérés, időtúllépés (timeout) meghatározása
    - Részfeladatok periodikus vagy adott időpontban történő futtatása
- Az ütemező ezekben az esetekben egy periodikus HW Timer megszakítást használ (systemticks, Linux : jiffies)
    - Periodikusan tud futni, és a szükséges funkciókat meg tudja valósítani
- Gyakoriság:
    - Minél kisebb a periódusidő:
        - Annál nagyobb felbontású ütemezés/időmérés leheséges
        - De annál nagyobb az overhead(gyakori megszakítás miatt) is
    - Tipikus 10 ms, 1-100 msközött jellegzetesen
- Timer szolgáltatás
- Virtuális timer a részfeladatok számára

- Az OS-ben van egy beépített szoftver timer szolgáltatás (a timeslice v. system tick-ből levezetve)
- Az OS rendszeróra óraütésnek a felbontásával dolgozik.

14. Operációs rendszer és kapcsolata a HW-vel, HW architektúrák, freeRTOS és Linux elemzése ilyen szempontból

OS:

- Feladata: A rendszer működtetése, erőforrás megosztás és védelem
- Interfész nyújtása az operációs rendszer szolgáltatásainak igénybevételére (OS API)
- Az egyik legfontosabb része az ütemező
- Real-time operációs rendszer
  - Bizonyos eseményekre megadott feltételek betartása esetén megadott válaszidővel reagál
  - Az ütemezőt minősíti a követelmény
- Az alkalmazásban felmerülő követelményeknek megfelelőt kell választani
  - Jellegzetesen a feladatok egy (kis) része real-time, míg mások nem azok
- Az OS belső működése függ attól, hogy:
  - Hány és hogyan összekapcsolt processzort tartalmazó számítógépen fut?
  - Hogyan szervezzük a memóriát?
  - Hogyan kommunikálunk a perifériákkal?
- Homogén többprocesszoros rendszerben gondolkodunk
  - Azonos processzorok…
- Csoportosítás
  - Single CPU (Uniprocessor)
  - Symmetric multiprocessing
  - Non-Uniform Memory Access
  - Elosztott rendszer

Végrehajtó egységek:

- A legfontosabb közös erőforrás
- CPU/MCU jellemzők
  - 8/16/32/64 bites, architektúra (ATmega, ARMx, x86)
  - Védelmi szintek
    - Végrehajtható utasítások korlátozása működési módtól függően
    - Pl. kernel/usermode(2 vagy 4 mode jellegzetesen)
      - Pl. usermode: I/O utasítások, CPU speciális regiszterek elérése tiltott
  - CPU/MCU memória szervezés (memoryorganization)
    - Neumann, Modified Harvard
    - CACHE hierarchia
      - Bizonyos teljesítmény felett a CACHE elkerülhetetlen
      - Ez a memória sebességétől függ (memorywall)
    - Memory Management Unit (MMU) támogatása
  - Megszakítás vezérlő (Interruptcontroller)
    - Direkt memória hozzáférés (DMA)
  - Buszok, Perifériák
    - Párhuzamos programozás : Timer IT fontos

- Milyen a beágyazott rendszer felépítése, amire programot kell írnunk? (SingleCPU, SMP, NUMA, elosztott rendszer)

15. Feladat megvalósítás/leírása, az egyszerű RR architektúrától az folyamatokig/szálakig, folyamatok és szálak megvalósításának HW aspektusai, freeRTOS és Linux elemzése ilyen szempontból

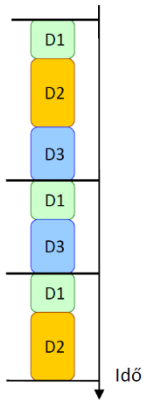
Feladatok leírása:

- Szoftver architektúrák a párhuzamos részfeladatok leírására
  - Alacsony szintű vezérlési szerkezet példák
    - Round-robin, Round-robin megszakításokkal
    - Függvény sor
  - Coroutine, Fiber
  - Process/Folyamat
  - Thread/Szál

Round Robin:

- A legegyszerűbb SW architektúra
  - Nincs megszakítás, nincs közös erőforrás kezelésével kapcsolatos probléma
- Nagy a szórása a válaszidőnek
- Worst-caseválaszidő:
  - A részfeladatok válaszidejeinekösszege (azaz lineárisan nő a taszkok számával)
- Egyes feladatok tekintetében javítható a válaszidő, ha többször hajtjuk végre azokat a főhurokban
- Az architektúra „törékeny”
  - Új taszk hozzáadása felborítja az ütemezést és az időzítéseket

```
void main(void) {

  while(1) {

    if ( !! Device 1 needs service ) {
       !! Handle Device 1 and its data
    }

    if ( !! Device 2 needs service ) {
       !! Handle Device 2 and its data
    }

    if ( !! Device 3 needs service ) {
       !! Handle Device 3 and its data
    }

    ...

  }

}
```



- Hibalehetőség nagy, pl. a részfeladatok leírását a forrásnyelv elemei szeparálják
- A CPU folyamatosan fut, nem tud aludni, a fogyasztás nagy

Round Robin megszakításokkkal:

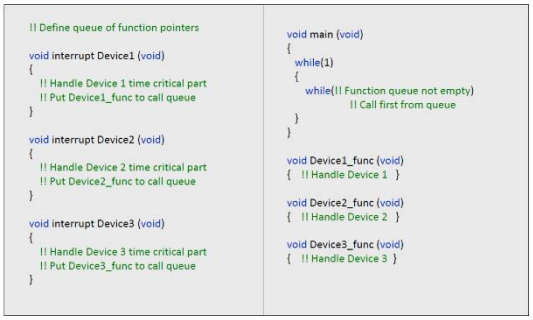
- Jobban kezeli az időkritikus részeket
  - A megszakítások magasabb effektív prioritáson futnak
- Ha az alkalmazott MCU/CPU a megszakításokhoz prioritást tud rendelni, még tovább lehet finomítani az architektúrát
- A kölcsönös kizárást meg kell oldani (IT és főhurok között)
- A válaszidőnek szórása nagy
  - Worst-case válaszidő: az összes feladat válaszideje + a megszakítások (arányos a taszkok számával)
  - Egyes feladatok tekintetében javítható a válaszidő, ha többször kérdezzük le őket a főhurokban

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

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

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

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

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

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

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

    ...
  }
}
```

- Az architektúra „törékeny”
- Lehetséges a fogyasztást csökkenteni a CPU altatásával
    - A CPU IT-re fel tud ébredni…

Függvény-sor alapú ütemezés:



```
// Define queue of function pointers

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

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

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

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

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

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

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

- Képes a prioritások kezelésére
    - Mind részfeladat szinten (hogyan valósítjuk meg a függvénysort)
    - Mind IT szinten (milyen az MCU/CPU megszakítás vezérlője)
- A kölcsönös kizárást meg kell oldani (IT és főhurok és az abban futó függvények között)
- Worst-case válaszidő (a legmagasabb prioritású feladatra) = a leghosszabb taszk válaszideje + IT
- A worst-case válaszidő nem nő lineárisan a taszkok számával
    - A válaszidő szórása kisebb, mint a korábbi megoldásoké
    - A legnagyobb futási idejű feladat futási idejével felülről becsülhető
- Egy új feladat nem borítja fel az eddigi időzítést
- Nem preemptív
- Lehetséges a fogyasztást csökkenteni a CPU altatásával
    - A CPU IT-re fel tud ébredni…

Coroutine és fiber (rost?):

- Kooperatív multitasking
    - Folyamaton vagy szálon belül.
        - OS támogatással vagy programnyelvi megvalósítás.
        - Az OS szinten a coroutine-eket vagy fiber-eket tartalmazó folyamat vagy szál kerül ütemezésre
        - Az ütemezési algoritmus a programozó kezében van, neki kell implementálnia (kooperatív ütemezés).
    - Coroutine: programnyelvi szintű elem
    - Fiber: rendszerszintű eszköz
- Egyes kooperatív együttműködést igénylő feladatok jól megvalósíthatóak vele
- Nem szükséges osztozni az erőforrásokon (kooperatív):
    - Nem kellenek OS hívások
    - Kisebb erőforrás igény
- Az OS ütemezi egy szálban/folyamatban őket, vagyis egyetlen végrehajtó egységet tudnak csak kihasználni

Feladat:

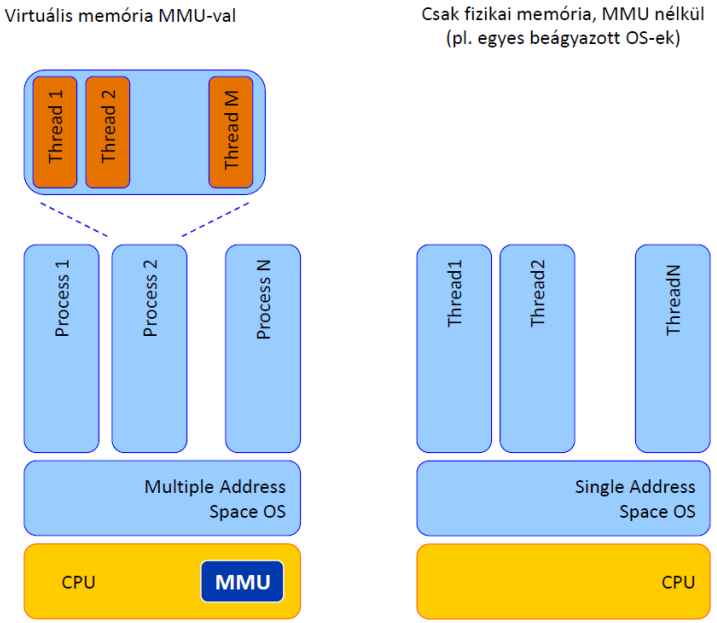
- Feladat fogalom eredetileg folyamat (process) értelemben került használatra
- A folyamat a végrehajtás alatt álló program
    - Ugyanabból a programból több folyamat is létrehozható
    - Saját kód, adat, halom (heap) és verem memória területtel rendelkezik
    - Védettek a többi folyamattól
        - Szeparáció, virtuális gép, sandbox
- Virtuális CPU-n futnak:

    - Nem férhetnek hozzá a többi folyamat és az operációs rendszer futása során előálló processzorállapothoz
    - Kontextus váltás történik, ha más folyamat kerül futásra
- Saját virtuális memóriaterületük van
    - Nem férhetnek hozzá más folyamatok virtuális memóriájához vagy direkt módon a fizikai memóriához
    - A processzor MMU-ja oldja ezt meg
        - Lehetséges azonos fizikai memóriaterületek megosztása például olvasási joggal (pl. kód terület)
        - A modern MMU-kezt akár írási joggal is meg tudják tenni
        - Teljes MMU és OS kontroll alatt a biztonság érdekében

Szál:

- A szál a CPU-használat alapértelmezett egysége, magában szekvenciális kód
    - Saját virtuális CPU-ja van, és saját verem áll rendelkezésre
    - A kód, adat, halom és egyéb erőforrások (pl. file) tekintetében osztozik azokkal a további szálakkal, amelyekkel azonos folyamat kontextusábanfut
- Folyamat = nehézsúlyú folyamat (heavyweight process)
- Szál = pehelysúlyú folyamat (lightweight process)
- Jelenleg a modern operációs rendszerek natív módon támogatják szálak létrehozását
    - Windows:
        - Program v. szolgáltatás = folyamat, folyamaton belül szálak
        - Az ütemező szálakat ütemez
    - Linux:
        - Program v. daemon = folyamat, folyamaton belül szálak
        - Az ütemező task-okat ütemez, amik lehetnek folyamatok vagy szálak

HW támogatás:



16. Feladatok együttműködése általában, közös erőforrásra várás megoldása és annak kapcsolata az OS-sel, azon belül részletesen közös memória esetén alkalmazható megoldások, freeRTOS elemzése ilyen szempontból

Kérdések:

- Erőforrások használata, közös erőforrások?
- Feladatok kommunikációja?
- Feladatok szinkronizációja?

Együttműködés lehetőségei:

- Közös memórián keresztül (RAM v. PRAM modell):
  - Szálak esetén (közös memória)
- Üzenetekkel

RAM modell:

- Klasszikus Random Access Memory (egy végrehajtó egység)
- RAM-modell szerint működik, azaz:
  - Tárolórekeszekből áll,
  - Egy dimenzióban, rekeszenként címezhető, csak rekeszenként, írás és olvasás műveletekkel érhető el,
  - Az írás a teljes rekesztartalmat felülírja az előző tartalomtól független új értékkel,
  - Az olvasás nem változtatja meg a rekesz tartalmát, tehát tetszőleges számú, egymást követő olvasás az olvasásokat megelőzően utoljára beírt értéket adja vissza.

PRAM modell:

- Parallel/Pipelined Random Access Memory (sok végrehajtó egység)
- Több végrehajtó egység írhatja és olvashatja párhuzamosan
- Változások a RAM modellhez képest:
  - Az olvasás-olvasás ütközésekor mindkét olvasás ugyanazt az eredményt adja, és ez megegyezik a rekesz tartalmával,
  - Az olvasás-írás ütközésekor a rekesz tartalma felülíródik a beírni szándékozott adattal, az olvasás eredménye vagy a rekesz régi, vagy az új tartalma lesz (versenyhelyzet), más érték nem lehet,
  - Az írás-írás ütközésekor valamelyik művelet hatása érvényesül, a két beírni szándékozott érték valamelyike írja felül a rekesz tartalmát (versenyhelyzet), harmadik érték nem alakulhat ki.
  - A gyakorlatban ezt használjuk…

Erőforrás: Minden olyan eszköz, amire a párhuzamos programnak futása közben szüksége van

- Közös erőforrás (shared resource)
  - Egy időintervallumban több, párhuzamosan futó feladatnak lehet rá szüksége
  - Az erőforráson osztoznak a feladatok
  - Többnyire egy időben egy vagy maximum megadott számú feladat tudja helyesen használni (írás és olvasás)
    - Egy felhasználó: Printer, Asszinkron soros port (UART), összetett adattípusokból létrehozott változók (string, tömb, struktúra, objektum)

        - Több párhuzamos felhasználó: SCSI vagy SATA NCQ HDD (N parancs optimalizált párhuzamos végrehajtására képesek)
- **A rendszertervezőnek és programozónak a legfontosabb feladata, hogy felismerje a közös erőforrásokat, és biztosítsa azok helyes használatát**
- Az OS szolgáltatásokat nyújt a probléma megoldására, de a megoldás a programozó kezében van!
- Probléma:
    - Kölcsönös kizárás (mutual exclusion)
        - Annak biztosítása, hogy a közös erőforrást egy időben csak annyi magában szekvenciális feladat használja, amely mellett a helyes működése garantálható
        - A kölcsönös kizárást meg kell oldanunk a programban
        - Többnyire a használt erőforrást lock-oljuk (elzárjuk)
            - Nem engedjük hozzáférni a többi részfeladatot
            - A kérdés az, hogy azt hogyan tudjuk megoldani, és milyen részletességgel kell megoldanunk azt
- Megoldás (RAM/PRAM modell esetén)
    - Kritikus szakasz (critical section)
        - A magában szekvenciális feladatok azon kódrészletei, amely során a kölcsönös kizárást egy bizonyos közös erőforrásra biztosítjuk
        - A kritikus szakasz a kérdéses közös erőforráshoz tartozik
        - A kritikus szakaszt a hozzá tartozó erőforrásra atomi műveletként (nem megszakítható módon) kell végrehajtanunk
    - Lock bit
        - True ha foglalt az erőforrás, false ha nem
    - Szemafor
        - Bináris és counter lehet
            - Bináris: egy feladat a kritikus szakaszban
            - Counter típusú: több feladat a kritikus szakaszban, vagy N darabos erőforrás készletből M darab lefoglalása
        - Két művelet értelmezett rajta:
            - Belépés: P(), Wait(),Pend(), …
            - Kilépés:V(), Signal(), Post(), …
    - Mutex
        - Kölcsönösen kizáró flag, aki birtokolja azé az erőforrás

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

17. Feladatok együttműködése általában, közös erőforrásra várás megoldása és annak kapcsolata az OS-sel, azon belül részletesen üzenet alapú kommunikáció esetén alkalmazható megoldások

Lásd 16. + **Feladatok közti kommunikációs módszerek:**

- Üzenetekkel:
  - Nincs közös memória
    - Folyamatok kommunikálnak egy operációs rendszeren belül
    - Elosztott rendszer
  - Vagy pl. mikrokernel alapú az operációs rendszer
  - Üzenet továbbítás (Message passing)
  - Lehet például:
    - Rendszerhívás
    - TCP/IP kapcsolat (TCP) vagy üzenet (UDP) gépen belül (localhost) vagy akár gépek között
  - Többnyire az alkalmazás kódjában OS API függvény/metódus hívásként jelenik meg
  - Az operációs rendszer valósítja meg szolgáltatásaival
  - Kölcsönös kizárás és szinkronizáció üzenetekkel (rendszerhívással, ritka, főleg a várakozás)
  - Van overhead-je:
    - Próbálkozások: Lockless programming, transactional memory etc.
    - Nincs jó megoldás, csak kevésbé rossz (Churchill mondása alkalmazható)
    - Alkalmazás és SW architektúra függő az optimum

18. Tipikus hibák feladatok együttműködése esetén, prioritás inverzió és elkerülése, holtpont, monitor alapötlete és megvalósításai

Tipikus hibák:

- Belépés/Kilépés elmaradása
- Többszöri be-vagy kilépés
- Más erőforrás lefoglalása
- Az erőforrás indokolatlanul hosszan történő lezárása
  - Minimális időre kell törekedni
- Prioritás inverzió
- Versenyhelyzet (race condition)
- Kiéheztetés (starvation)
  - Feladat soha nem jut hozzá a futásához szükséges erőforráshoz
- Deadlock és livelock

Prioritás inverzió (priority inversion)

- A legegyszerűbb esetének előfordulásához kell:
  - 3 feladat, különböző statikus prioritással,
  - Egy közös erőforrás, amelyet a 3 feladat közül a legmagasabb és a legalacsonyabb is használni kíván.
  - A közepes prioritású feladatnak CPU intenzívnek kell lennie.

- Kedvezőbb esetben csak a rendszer teljesítménye csökkent, a válaszidők nőnek. Valós idejű rendszer????
- Rosszabb esetben kiéheztetés, vagy akár holtpont is lehet a prioritás inverzió eredménye
- Megoldás:
    - Prioritás öröklés (Priority inheritance, PI):
        - Az alacsony prioritású feladat megörökli az általa kölcsönös kizárással feltartott feladat prioritását a kritikus szakaszából való kilépéséig.
        - Csak részben oldja meg a problémát.
    - Prioritás plafon (Priority ceiling, PC):
        - Majdnem ugyan az, de az adott közös erőforrást használó legnagyobb prioritású feladat prioritását örökli meg (ami lehet nagyobb mint az éppen feltartott feladat).
        - Az adott erőforrást máskor használó többi feladat sem tud futni (ha esetleg azok is CPU intenzívvé „válnak”).
        - Az alacsony prioritású feladat akadályoztatás nélkül le tud futni.
    - Modern beágyazott OS-ekben választhatóak...
        - None, PI, PC (ha preemptív ütemezőt választunk)
    - Sokak szerint a prioritás inverzió alapvetően egy rendszertervezési hiba eredménye:
        - Megoldás jó tervezéssel.

Monitor:

- Lokalizáljuk a lock-olással kapcsolatos feladatokat a közös erőforrást körülvevő API-val
    - A lock-olás nem szétszórva történik a programban, hanem egyetlen, a közös erőforráshoz szorosan tartozó programrészletben
    - A megvalósítás lehet automatikus, például nyelvi szinten (pl. JAVA, C#).
        - A compiler valósítja meg a kölcsönös kizárást biztosító konstrukciókat (pl. szemafor vagy mutex tényleges alkalmazásával)
    - Kézzel is készíthetünk hasonló konstrukciót, pl. egy védett objektumot hozhatunk létre, amely a nyilvános metódusaiban elvégzi a lock-olást, és elrejti a közös erőforrást
- Hoare szemantika:
    - Azonnal az erőforrást megszerző részfeladat fut
        - Erőforrás igényes és nehéz megvalósítani
        - Nem kompatibilis a preemptív ütemezőkkel
- Mesa szemantika:
    - A futásra kész részfeladatok közé kerül, és később az ütemező futtatja
    - Vannak vele gondok, de azok megoldhatóak
    - "notifyAll" vagy"broadcast" üzenetek küldése is lehetséges
        - Minden adott eseményre váró futásra kész állapotba kerül
- JAVA példa
    - A „synchronized” blokk

```
synchronized (object) {
        // az adott „object”-rebiztosítva van
        // a kölcsönös kizárás a blokkon belül
}
```

    - A „synchronized” kulcsszó

```
synchronized void myMethod() {
        // A metódushoz tartozó objektumra
        // biztosít kölcsönös kizárást
}
```

Holtpont (deadlock):

- A holtpont a legsúlyosabb hiba, ami multiprogramozás során előfordulhat
- Pontos definíció:
    - Egy rendszer feladatainak egy H részhalmaza holtponton van, ha a H halmazba tartozó valamennyi feladat olyan eseményre vár, amelyet csak egy másik, H halmazbeli feladat tudna előállítani
    - H részhalmazról van szó, többnyire csak néhány feladat érint
- Nehéz felismeri...
    - Versenyhelyzet formájában jelentkezik
        - Hol működik, hol előáll a holtpont…
    - Bizonyos feltételek együttes teljesülése esetén áll elő a holtpont
        - Kölcsönös kizárás
        - Foglalva várakozás
        - Nincs erőszakos erőforráselvétel
        - Körkörös várakozás
- Megoldás:
    - Strucc algoritmus (nem foglalkozunk a lehetőséggel)
        - Nem kritikus rendszerekben megtehető
    - Holtpont észlelése és feloldása (deadlock detection and recovery)
    - Holtpont megelőzése tervezési időben (deadlock prevention)
        - Strukturális védelem
        - Analízis
    - Holtpont elkerülése futási időben (deadlock avoidance)

19. Távoli eljárás és metódus hívás alapötletei és megvalósításai részletesen, tipikus elosztott rendszer architektúrák

Távoli eljárás hívás (Remote Procedure Call, RPC):

- Egy másik folyamat kód memóriaterületén elhelyezkedő függvény „meghívása” a hívó folyamatból üzenetek felhasználásával
- A hívó fél blokkolva vár a távoli hívás lefutására
- A meghívott függvény az őt tartalmazó folyamat egyik vezérlési szálában fut le
- Hívása lényegében azonos egy „lokális” függvény hívásával
- A ténylegesen végrehajtott függvény megírása sem különbözik egy lokálisan végrehajtható függvény megírásától

- A kliens oldali csonk feladata a hívás során
    - Feladata a hívási paraméterek összecsomagolása platform független formára és elküldése a végrehajtó folyamatnak
    - Ehhez felhasználja a lokális operációs rendszer és a hálózat szolgáltatásait

- A szerver oldalon az RPC-t használó szolgáltatás megkapja a szabványos formátumban megérkező hívást
    - Platform független formáról lokális formára hozás
    - A lokális függvény hívása
    - Visszatéréskor a visszatérési érték platform független formára hozása
    - A visszatérési érték küldése a kliensnek

- A kliens oldali csonk feladata a hívás visszatérése során
    - Feladata a szervertől megkapott visszatérési érték konvertálása platform független formáról lokális formára
    - Visszatérés a lokális csonkból a lokális formátumra hozott visszatérési értékkel
- A kliens szál az RPC hívás során várakozik
    - Eseményre várakozik a hívás során
- A szerver szál vár a beérkező hívásokra, és amikor van kiszolgálandó hívás, akkor fut
    - Eseményre várakozik a hívás beérkezéséig

Elosztott rendszer architektúrák:

- Publish-Subscribe
    - Publisher = Termelő
    - Subscriber= Fogyasztó
    - A termelő által előállított információra a fogyasztó előfizet
        - A termelő nyilvántartja az általa előállított információra előfizetőket
            - Referencia a fogyasztók adatfogadó interfészére
        - Ha van információ, elküldi azoknak az előfizetőknek (push)
            - Egyenként mindenkinek, vagy multicastformában
            - Pl. RPC-vel
    - Bonyolult „hálózatokat” lehet felépíteni olyan komponensekből, amik más termelőktől érkező adatok alapján új adatokat állítanak elő
        - Adatvezérelt szerkezet (dataflow)
    - Megoldások
        - MatlabSimulink
        - Ni Labview
    - Az architektúra lehetővé teszi a többprocesszoros működést is
- Databus
    - Számos kommunikációs hálózat adatbuszként is felfogható (broadcasta fizikai közegben):
        - Pl. CAN
        - Minden üzenet eljut minden NODE-hoz
        - Az üzenet címe alapján egy NODE veszi vagy nem veszi, majd a tartalom alapján válaszol vagy nem
    - Virtuális adatbusz
        - Pont-pont kommunikációs hálózat, de szükség van buszszerű működésre
        - Publish-Subscribealapján, arra ráépülve létrehozható egy virtuális adatbusz
        - Minden node„termelő”, amikor a buszra ír
        - Minden node„fogyasztó”, amikor a többi node-tóla buszon keresztül adatot kap

        - A busz egy központi nodeaz elosztott rendszerben, ahol minden adat megtalálható, sokat kommunikál, szűk keresztmetszet lehet!
- Proxy
    - A proxy beáll két vagy több kommunikálni kívánó fél közé
    - Cél:
        - Kompatibilitás megteremtése
        - Funkciók rejtése
        - Teljesítményillesztés
- Broker
    - Központi szereplő, amely összepárosítja a kommunikálni kívánó feleket
        - Ismert a felek számára
            - Az elérhetősége meg van adva a rendszer konfigurációjában
            - Egyszerű automatizmussal megtudható
        - A felek vele veszik fel a kapcsolatot
        - Megadják a felek igényeiket és/vagy az általuk nyújtott szolgáltatásokat
        - A bróker összeválogatja azokat, megteremti a kapcsolatot a felek között
        - Utána a felek egymással kommunikálnak

20. Modell alapú szoftverfejlesztés alapötlete, kapcsolata a mérnöki problémamegoldás módszereivel, fejlesztési módszerek és módszertanok, a modell használatának lehetőségei a fejlesztési folyamatban, a közös jelölésrendszer fontossága, UML kialakulása és alapjai, UML profilok, SysML

Kérdés:

- Hogyan fejlesszünk jobb minőségű szoftvert minél kevesebb erőforrás befektetésével?
    - A meglévő klasszikus fejlesztési módszerek elérték a határaikat
        - A komplexitás növekedésénél sokkal gyorsabban nő az erőforrásigény
        - Nincs elég jó szoftveres mérnök
    - Hatalmas csapatok dolgoznak a szoftvereken
        - A kommunikáció nehézkessé válik, majd egy csoportméret felett ellehetetlenül
        - Valahogy segíteni, explicitté kell tenni a kommunikációt
    - Nem szoftveres hátterűek, pl. az alkalmazás szakértői (domain expert), dolgoznak a szoftvereken, sőt egyes munkafolyamatokat csak ők tudnak megcsinálni
        - Számukra a szoftveres koncepciók érthetetlenek, olyan eszközöket kell a kezükbe adni, amivel a saját világukban képesek szoftvereket létrehozni
        - Jó példa: Matlab Simulink, NI Labview
        - Támogatni kell az ilyen szakértők és a szoftveresek kommunikációját is

Felismerés:

- A modellalkotás alapvető része a mérnöki folyamatoknak
- Hogyan használjuk ezt a szoftverfejlesztés során?
    - Fejben modelleket alkotunk a problémáról (analízis)
        - Esetleg azt ledokumentáljuk természetes nyelven (WORD)
        - Követelmények rögzítése és specifikáció elkészítése
    - A modell alapján fejben kitaláljuk a rendszert (tervezés)
        - Ezt is ledokumentáljuk, terjesszük, megbeszéljük, finomítjuk a fejlesztő csapattal együttműködve

        - A terv maga modellje az elkészítendő rendszernek valójában
    - A tervek alapján elkészítjük a kódot (megvalósítás)
        - A tervek alapján a fejlesztők megírják a kódot (begépelik, folyamatosan tesztelve egy bizonyos szinten)
        - Újabb modellt alkotunk, hiszen a kód egy modellje a rendszer szoftver komponenseinek
    - Az elkészült kódot/rendszert teszteljük, üzemeltetjük
        - A fejünkben a dokumentáció alapján kialakult modelljével vetjük össze a valós elkészített rendszert
- A MODELL központi szerepet játszik ebben a folyamatban…

Modellalapú szofverfejlesztés:

- Egyre általánosabban használjuk, de a szoftveres területről indult
- Azon az alapvető gondolaton alapul, hogy a fejlesztési folyamatnak alapvető része a modellalkotás és a modellek alapján újabb modellek, és végül a végtermék elkészítése
    - Miért ne támogassuk, helyezzük ezt a fejlesztési folyamat központjába megfelelő szoftver támogatással?
    - Komplex rendszerek egyébként sem hozhatóak létre alacsony szintű (pl. C kód) eszközökkel hatékonyan a tapasztalatok szerint
    - Az absztrakt modellek (az implementációhoz direkt nem köthető) lehetővé teszik a fontos tulajdonságok és azok viszonyának az egyértelmű leírását (nem veszünk el a részletekben)

A modell: A valós világról alkotott (többnyire annak egy jól meghatározott részéről) egyszerűsített konstrukció, amely elégséges információt hordoz a vizsgálat tárgyát képező dolog adott szempontból történő részleges megértéséhez.

Fejlesztési módszerek és módszertanok:

- Módszer (Method)
    - A szoftverfejlesztés során követendő szabályozott lépések sorozata, amely segítségével a fejlesztés alatt álló szoftver különböző aspektusait (azoknak a modelljét) írjuk le.
    - Két részük van:
    - Folyamat, előírt és betartott lépések sorozata
        - Cél a fejlesztési folyamat egyértelmű megadása, a túlzott szabadság korlátozása
    - Jelrendszer, a fejlesztés alatt álló szoftver (rendszer) különböző aspektusainak leírására
        - Közös nyelv a fejlesztők kommunikációjának biztosítására
- Módszertan (Methodology)
    - Azonos elvek alapján dolgozó módszerek (hasonlóak)
    - Például:
        - objektum orientált elemzés és tervezés, valamint megvalósítás
        - modell-alapú fejlesztés (model-driven engineering, model-driven architecture, model-based design)
    - Módszer példák
        - ROPES (Rapid Object-based Process for Embedded Systems),
        - Rational Unified Process

        - Scrum

Modell használhatóságának lehetőségei:

- Modellből automatikus implementáció
    - Modellből kód szintézise (generálása)
    - Automatikus eszközökkel
- Implementációból modell
    - Reverse engineering
    - Meg akarjuk érteni az implementációt
- Modell és kód kétirányú átjárhatósága
    - Ha nincs meg, akkor a kód elválhat a modelltől
    - Ettől a folyamat még bonyolultabb lesz
- Végrehajtható modell
    - A modell tesztelhető a teljes implementáció nélkül is
    - Modellezési hibák korai felderítését teszi lehetővé (teszt)
- Modell automatikus ellenőrzése
- Modell alapján tesztek generálása (más modellek és az implementáció tesztelésére)
- Modell transzformáció
    - Modellek közötti átjárhatóság
    - Pl. fejlesztőeszközök közötti átjárhatóság
    - Általánosan értelmezve az összes korábbi lehetőség ennek egy speciális esete

Közös jelölésrendszer:

- Szükségünk van egy közös jelölésrendszerre
- UML –Unified Modeling Language
    - ObjectManagement Group (OMG, www.omg.org) által kidolgozott szabvány
- Az UML nem csak jelrendszer, hanem tartalmaz egy úgynevezett meta-modellt is
    - Meta-modell: Modellek leírására szolgáló modell
    - Az UML le tudja írni önmagát a saját meta-modelljével
        - Ez a jó kifejezőképesség egy bizonyítéka
        - Az UML meta-modell nem formális, klasszikus matematikai módszerekkel a helyessége nem ellenőrizhető

UML:

- Egy modellezési nyelv
- Grafikus reprezentáció
    - UML diagramok megadott grafikus elemekkel
- XMI formátumban cserélhetők UML-t támogató fejlesztőeszközök
    - XML Metadata Interchange(XMI)

UML profilok:

- UML kiegészíthető, kiterjeszthető
    - UML Profile
    - Tisztán additív, nem lehet ellentétes az UML standarddal az új UML Profile
- Ismert UML Profile-ok:
    - UML Profile for Enterprise Application Integration (EAI)
    - UML Profile for Enterprise Distributed Object Computing(EDOC)
    - UML ProfileforSchedulability, Performance and Time

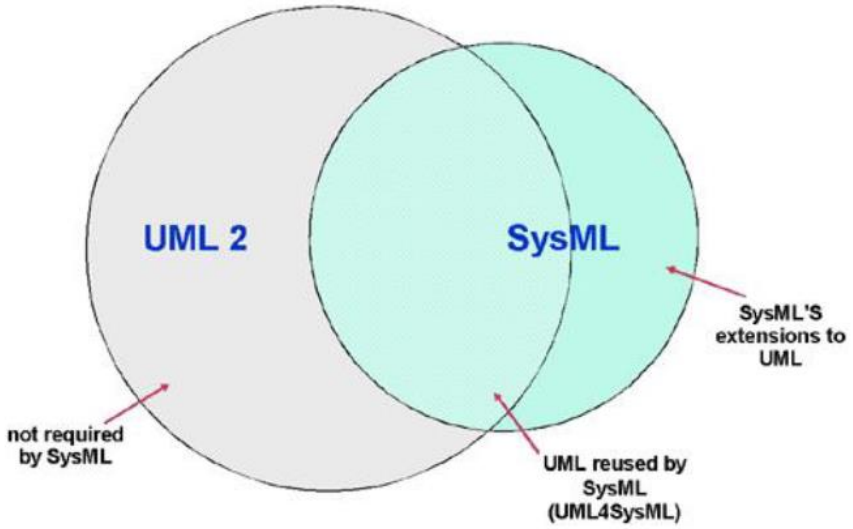
- UML ProfileforSystems Engineering(SysML)
- UML Testing Profile

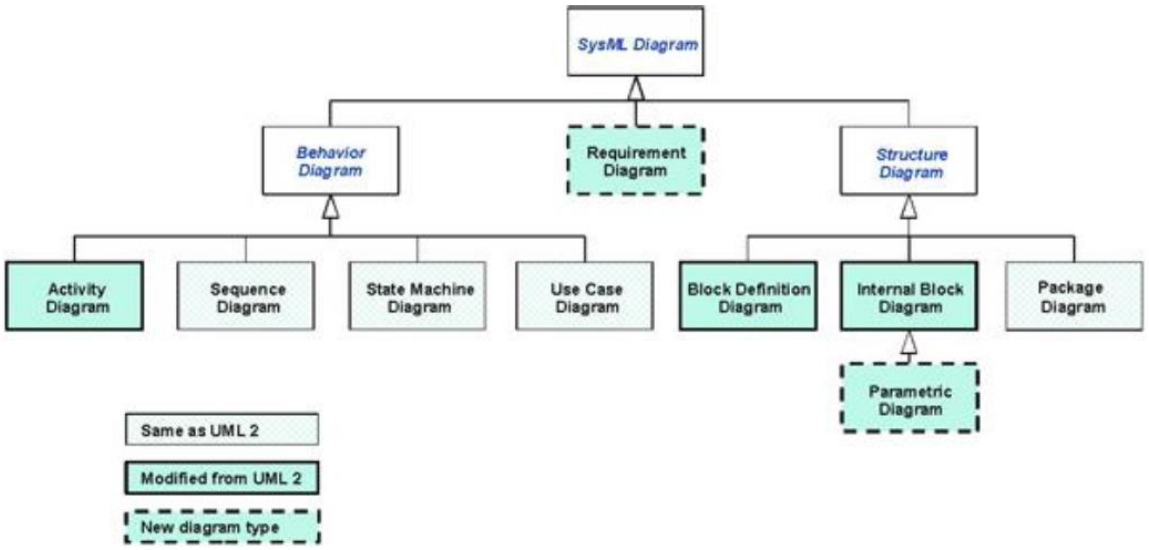
SysML:

- Az UML szoftver modellezésére lett kifejlesztve, de tulajdonképpen bármit modellezhetünk vele
  - Ugyanakkor erre a célra nem ideális
  - Pl. beágyazott rendszerek leírására is korlátozottan képes
  - Kell egy speciális nyelv rendszerek leírására
    - HW és SW, humán szereplők, folyamatok, infrastruktúra leírása

21. UML és a SysML diagramok és azok kapcsolata, az egyes diagramok felhasználása vízesés modell esetén

SysML és UML kapcsolata:

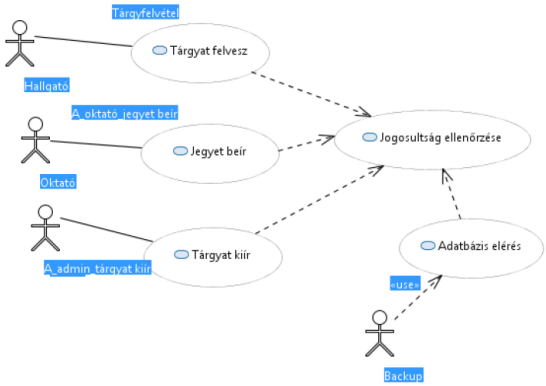




22. UML use-case és osztálydiagram, valamint azok felhasználása

Use-case diagram:

- Követelmények és a specifikáció rögzítése során használják tipikusan
- A rendszer és azokat körülvevő aktorok rögzítése
    - Aktor: külső szereplő (nem feltétlenül humán, még a pálcikaember jelölés esetén sem)
        - Kapcsolatban van a rendszerrel, igénybe veszi a rendszer szolgáltatásait
        - Bizonyos módon használja azt (használati eset ebből jön)
        - A használati esetek között is vannak kapcsolatok (relations)
        - Sztereotípiákkal írjuk le (pontosítjuk) a kapcsolatokat



- Rendszer és részrendszer (<<subsystem>> sztereotípia) határainak rögzítése
- Comment
    - Szimpla szöveges megjegyzés a humán olvasóknak
- Kényszerek (constraint)
    - Object Constraint Language(OCL) nyelven kerülnek leírásra a constraint-ek
        - Ezek fordíthatók a célnyelvre

        - Bizonyos tool-ok célnyelvű kényszereket is támogatnak
- Felhasználás:
    - Gyenge diagram
        - Nagyon szemléletes, de további automatikus felhasználása korlátozott
    - A feladat analízisét segíti
        - Specifikáció feldolgozása, finomítása
        - Követelmények kidolgozása
    - Tesztek kidolgozásánál érdekes lehet
        - Tudja-e a rendszer azt, amit a használati eset diagram leír?

Osztálydiagram:

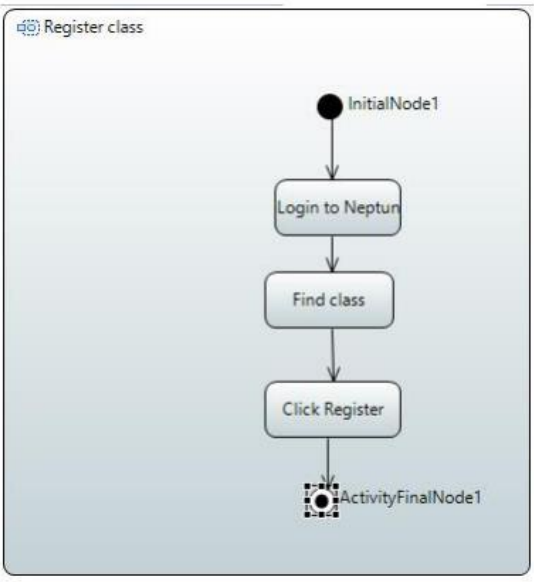
- Osztály diagram: A szoftverben használt osztályok tervezési idejű leírása
    - Analízis során a rendszer környezetének megértése
    - Tervezés során a fejlesztett szoftver leírása
    - Kész célnyelvi kódból az osztályhierarchia és az osztályok kapcsolatainak vizualizációja (Reverseengineering)
    - Tesztelés során osztály specifikus tesztek
- Objektum diagram: A szoftver futása során az objektumok kapcsolatának leírás
    - Valós rendszerek megértése analízis során
    - Futási idejű koncepciók tárgyalása tervezés során
    - Hibakeresés/tesztelés során is automatikusan előállítható
- Felhasználás:
    - Erős diagram
        - Kód szintézis megfelelő részletességű diagramok esetén
        - Reverse engineering: Kódból UML osztály diagram előállítása
    - A modern fejlesztő eszközök mindkét utat támogatják

23. UML aktivitás és szekvencia diagram, valamint azok felhasználása

Szekvencia diagram:

- Két vagy több osztály vagy objektum interakcióját írja le
    - Szekvencia, nem idő
    - Az idő specifikálható a constraintek közt
- Egy diagram egy esetet ír le
    - Adott dolog megtörténhet vagy sem
    - Több diagrammal leírható a teljes viselkedés
- Elemek:
    - Osztály/objektum
    - Életút
    - Akció/viselkedés
    - Osztály/objektum destrukciója
    - Üzenetváltások
        - Sync
        - Async
        - Reply
        - Delete/create
        - Lost

- Felhasználás:
  - Osztályok, objektumok interakciója írható le vele a fejlesztési folyamat alatt
  - Kódot ritkán generálnak belőle (kísérletek folynak)
- Főleg tesztesetek generálására szolgál



Aktivitás diagram:

- Jól definiált kiterjesztett flowchart
  - Irányított gráf
- Szemantikája a token folyamon alapul
- Kezdeti node:
  - Token megszületik
- Végső node:
  - Token meghal
- Aktivitás
  - Token benne lehet
  - "Csinál valamit"
- Vezérlő folyam
  - Élek az irányított gráfban
- Adat node
  - Téglalap…
- Döntés node
  - Gyémánt forma
  - Egy input – több output logikai kifejezéssel
- Összegyűjtő node
  - Több input – egy output
- Felhasználás:
  - „Mindenki" ismeri a flowchartokat, ismerős
  - Nem gyakran, de használt
    - Kódgenerálás
    - Reverse engineering

24. UML állapotdiagram és felhasználása, UML állapotdiagram implementációja

Állapotdiagram:

- Harel statechartokon alapul
- Osztályok és objektumok belső működésének leírására szolgál
- Kiegészítése a véges állapotterű állapotgépnek
  - Hierarchikus
- Népszerű, főleg beágyazott rendszerekben
- Főbb komponensek:
  - Állapot
    - A rendszer lehet benne jelentős ideig
    - A rendszer ebben végez valamit
  - Átmenet
    - Megtörténik, ha egy esemény bekövetkezik vagy egy specifikus logikai kifejezés TRUE-ra értékelődik ki

    - Esemény
        - Valami történt a rendszerben
    - Pszeudo állapot
        - Tranzies viselkedés/állapot
        - Jobb mód az átmenetek leírására
- Az állapotgép egy irányított gráf
    - Csúcsok az állapotok és a pszeudo állapotok
    - Élek az átmenetek
- Állapot:
    - Van neve (három rész: név, aktivitás, belső állapotok)
    - Egyszerű vagy összetett
        - Egyszerű: nincsenek régiók vagy alrendszerek
    - Belépési- és kilépési ponttal, belső tranzakciókkal és elvégzendő akciókkal rendelkezik
- Felhasználás:
    - Legjobb diagram az osztályok/objektumok viselkedésének leírására
        - FSM (Finite State Machine)-ek segítségével kódolunk
        - Szoftver = nagy, nehezen átlátható FSM
    - Erős diaram
    - Bármely pontján használható a fejlesztési folyamatnak
- Implementáció:
    - Objektumorientált módon
        - Nehéz, de működik
    - UML állapotgépből transzformáció
        - FSM bonyolult lesz