Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

Pogány László

DEMONSTRÁCIÓS ALKALMAZÁS FEJLESZTÉSE OPERÁCIÓS RENDSZER FUNKCIÓK BEMUTATÁSÁRA

KONZULENS

Micskei Zoltán

BUDAPEST, 2013

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

Alulírott **Pogány László**, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2013. 05. 24.

...……

Pogány László

Összefoglaló

Az operációs rendszerek területén oktatott algoritmusok megértése nagymértékben járul hozzá a mérnöki gondolkodásmód fejlődéséhez. A hallgató szembekerül az alkalmazott megoldások hasznosságának vizsgálatával és szembesül az egymásnak ellentmondó igények és kritériumok rendszerével. Emiatt folyamatosan kerülnek kifejlesztésre az algoritmusok működését prezentálni képes eszközök. Ez a dolgozat is egy hasonló eszköz kifejlesztésével foglalkozik.

Elsőként meghatározásra kerülnek azok az elvek, melyeket létező alkalmazások már megvalósítanak. Ezután az Operációs rendszerek c. tárgyhoz igazított igények szerint részletesen specifikálásra kerülnek a fentieket teljesítő feladatok. Ehhez szükség van az algoritmusok és metrikák, valamint az azokhoz tartozó problémakörök bemutatására. A témakörből implementáció során kihagyandó részek megemlítését követően részletesen bemutatásra kerül a tervezés folyamata.

Kifejtésre kerülnek a tervezés során alkalmazott fontosabb tervezési minták. Az alkalmazás szerkezetének és működésének körvonalas bemutatása során összefoglalásra kerülnek azok az okok, amelyek felhasznált megoldások alkalmazásához vezettek. Rövid kitérőként szót ejtek a fejlesztés során felhasznált technológiákról, valamint azokról az okokról, amelyek alkalmazásukhoz vezettek.

Végül nem maradhat el az elkészített alkalmazás értékelése sem, melyhez a sajátomon kívül külső forrásból származó, de az operációs rendszerek területét már alapjaiban ismerő hallgatók véleményei alapján állítottam össze. Az elhangzott vélemények alapján megkísérlek objektív tanácsokat adni arra, hogy a későbbiekben hogyan lehetne tökéletesíteni az itt elkészített demonstrációs alkalmazást.

Abstract

Understanding the algorithms taught in the area of the operating systems can contribute significantly to the development of an engineering mindset. This way the student can realize the usefulness of the implemented solutions and can also analyze the system of conflicting demands and criteria. Because of this reason the number of tools visualizing the operation of these algorithms is continuously increasing. In this work I am also developing a tool which can meet the previous description.

In the introductory chapters principles already implemented in existing applications are defined and analyzed. The tasks related to this work were specified and adjusted to the demands of the Operating Systems course at BME. This requires the presentation of algorithms and metrics as well as the issues relating to them. After describing the topics which should be left out of the implementation process the detailed planning process is described.

Next, the most important design patterns utilized in the applications were presented. In the course of the monographic introduction of the construction and operation of the application, the most important reasons leading to the adopted solutions were summarized. A short summary about the technologies used by the development and about the reasons leading to their usage is important as well.

At the end, the evaluation of the developed application can also not be forgotten, where not only my opinion is stated, but the experiences of students with basic knowledge about the operating systems is also presented. Taking this feedback into consideration I summarize how this new demonstrative application can be improved in the future.

1 Bevezetés

A szakdolgozat célja egy olyan demonstrációs alkalmazás elkészítése, amivel szemléltetni lehet az operációs rendszerekben alkalmazott általános algoritmusokat és módszereket. Az alábbiakban körvonalakban összegzésre kerülnek azok az igények, melyek a feladatkiírás során kerültek meghatározásra. Az ott meghatározott pontok egy olyan alkalmazás megtervezését és elkészítését fogalmazzák meg, amely szem előtt tartja a jelenlegi oktatási szempontoknak megfelelő igényeket. Ezen igények megkövetelik, hogy az alkalmazás kellő részletességgel és egyszerűséggel demonstrálja az operációs rendszerek területén kialakított megoldásokat, segítse a műszaki szempontból fontos elvek megértését.

1.1 Igények összegzése

A bemutatandó megoldások a karon oktatott Operációs rendszerek c. tárgy tematikájához igazítva kerültek megválasztásra, mégpedig a következő területeken:

- ütemezési algoritmusok
- lapcsere algoritmusok
- memóriafoglalás
- virtuális memória címleképezés

Legfontosabb cél, hogy a megjelölt területek legtöbbjét sikerüljön futtatható példákkal lefedni, hogy az algoritmusok prezentációja teljes körű legyen. Ehhez egyrészt elengedhetetlen a megfelelő modell kialakítása, amely elvégzi az algoritmusok példákon történő szimulációját és szimulációs időben biztosítja a hozzájuk tartozó metrikák, egyéb statisztikák számítását. Másrészt szükség van különböző nézetekre, melyek a számított adatok megfelelő vizualizálásáért felelnek.

További fontos szempontok közül kiemelendő az alkalmazás bővíthetősége, hogy a megoldás a jövőben bemutatni kívánt algoritmusokkal is kiterjeszthető legyen, valamint a bemenetet képező feladatok könnyű konfigurálhatósága, mely különböző környezeti viszonyok szimulációját teszik lehetővé, akár egy időben, több feladat párhuzamos futtatása által.

Az algoritmusok teljes körű bemutatásához, mérnöki szempontból történő elemzéséhez és összehasonlításához hasznos segítséget nyújt a különböző metrikák futási időben történő számítása. Ezért az alkalmazás nemcsak algoritmusok működését kívánja bemutatni, hanem megkísérli támogatni a valós életben is előforduló esetek létrehozását és szemléltetni az ezeken történő példamegoldást. A fenti indokok miatt az elkészítendő megoldás tekinthető egyfajta szimulációt végző egységnek.

Ezen munka számos helyen egyszerűsített formában valósítja meg a valós rendszerek megoldásait. Az itt kialakított modell csupán absztrakt jelleggel bír, nem tökéletesen tükrözi a valós rendszerek tényleges működését. Szerencsére a demonstráció során az alapelvek bemutatáshoz, tökéletesen részletes megvalósítás kialakítására nincs is szükség. Az egyszerűsítő tényezők néhol a forrásanyagban levő aluldefiniáltság, néhol pedig az egy területen belüli algoritmusok eltérő megvalósításaiból, adatszerkezeteiből fakadnak. Ennek ellenére igyekszem a lehető legnagyobb mértékben közelíteni az alkalmazott megoldásokat, közös alapot adni az különböző szempontoknak és a lehető legkisebb mértékű egyszerűsítésekkel élni.

2 Létező megoldások és háttér információk

A fejezetben összefoglalásra kerülnek a bemutatni kívánt területekhez kapcsolódó legismertebb oktatási célú eszközök legfontosabb tulajdonságai. E szoftverek előnyeinek és hátrányainak kielemzése után, az Operációs rendszerek c. tárgy oktatása során használt jegyzet alapján összefoglalásra kerülnek az érintett területekhez tartozó legfontosabb háttér információk.

2.1 Létező megvalósítások

A demonstrációs eszközökkel való ismerkedést a már létező eszközök felkutatásával és kielemzésével kezdtem. Az eszközök használata során szerzett tapasztalatokat és felfedezett hiányosságokat az általam elkészített alkalmazás megtervezése és implementációja során használtam fel.

Az ismerkedés során a következő eszközöket vizsgáltam meg:

- MOSS Scheduling Simulator [5]
- MOSS Memory Management Simulator [5]
- Visual OS [6]
- Process Scheduling Simulator [7]
- Address Translation Simulator [7]
- MOSS: A Mini Operating-System Simulator [8]
- SOsim: Simulator for Operating Systems Education [9]
- SOS (Simple Operating System) Simulator [10]

2.1.1 A demonstrációs eszközök legfontosabb tulajdonságai

A megvizsgált eszközök többsége vagy parancssorban futó, vagy parancssorból indítható, grafikus felülettel rendelkező alkalmazások, melyek az egyes részterületeket vagy önmagukban, vagy az operációs rendszer összetett működésének részeként mutatják be. Azok, amelyek nem a komplex működés egy elemeként szemléltetik az egyes folyamatot, könnyebben átláthatóbbak, bár a teljes működésből kevesebbet mutatnak.

A MOSS Scheduling Simulator egy Java alapú program, nem rendelkezik grafikus felülettel, működésének beállítása a legtöbb eszközhöz hasonlóan konfigurációs fájlok módosításával lehetséges. Bemenetként pontosan megadhatóak a taszkokat jellemző idők, ezáltal a pontos időviszonyok vizsgálata is lehetséges. A futás során keletkező eredmények fájlok formájában állnak rendelkezésre. Sajnos mind a bemenetek megadásához, mind pedig a kimenetek értelmezéséhez szükséges a dokumentáció értelmezése, így órán használt demonstrációs eszközként nehezen használható. További hátrány, hogy a kimenetben nehezen követhető a lépésenkénti változás. Az alkalmazás bővítéséhez elegendő egy új algoritmust megvalósító osztályt implementálni.

A Process Scheduling Simulator és a Visual OS eszközök az előzőnél már intuitívabb felülettel rendelkeznek. Az előbbi futása során szép grafikus kimenetek, diagramok előállítására is képes. A bemenet megadása bonyolult. Működtetéséhez szintén szükséges egy hosszabb dokumentáció áttanulmányozása, így demonstrációs eszközként nehezebben használható. Több ütemezési algoritmust is támogat, emellett metrikák számítására is képes. Az eszköz inkább szimulációra, mint demonstrációra alkalmas. A másik eszköz a taszkok futtatása során felmerülő problématerületeket egyszerre mutatja be, melynek csak egy-egy elemét képezi a taszkok ütemezése vagy a memóriafoglalás. Ezáltal egy komplexebb képet ad a működésről. Összetettsége miatt a területekkel ismerkedők könnyebben elveszíthetik a fonalat.

Hasonlóan mutatja be a taszkok ütemezését a SOsim: Simulator for Operating Systems Education nevű eszköz is, mely grafikusan jelzi a taszkállapotokat és a virtuális memória tartalmát. A szimuláció gyorsasága azonban elégé megnehezíti az események követését. Az események követésének nehézségét visszapörgethető naplózással próbálja kompenzálni. Előnye, hogy megkülönbözteti a löketidőket azok típusa szerint, valamint a statisztikákat futás időben számolja és ezek lépésenként is megfigyelhetők. Ezen kívül képes prioritásos ütemezések szimulációjára is, felülete animált. Hátránya, hogy a kimenetek értelmezése nem kezdő felhasználók számára nem egyszerű.

A virtuális memória címleképezés témakörét hatékonyan mutatja be az Address Translation Simulator nevű eszköz. Képes akár többszintű laptáblákon történő címleképezés végrehajtására és támogatja TLB használatát is. Képes bitenként megjeleníteni a felbontott logikai címeket. Az alkalmazás a címtartomány összes címét

megjeleníti, mely némileg zavart keltő, néhol túlságosan is részletes. A címleképezés eredményét jeleníti meg, annak folyamata és számítási módja nem mindig nyilvánvaló.

A MOSS Memory Management Simulator az előzőekkel ellentétben már támogatja a lépésenkénti végrehajtást, ami sokkal átláthatóbbá teszi a demonstrációt. A virtuális memória lapokra történő címleképezését valósítja meg. Java alapú program, szintén rendelkezik grafikus felülettel, mely könnyen értelmezhető. Bemenete a laptábla kezdeti állapota, valamint egy megadott cselekvéssorozat. Kimenete azon kívül, hogy megjelenítésre kerül a felületen, egy kimeneti fájlba is belekerül, melynek értelmezéséhez már szükség van a dokumentációra. További előnye, hogy a címek mind bináris és hexadecimális formában is megtekinthetőek.

2.1.2 Létező megoldások kiértékelése

Az előző alfejezetben megvizsgált alkalmazások legtöbbje sajnos csak nehezen használható demonstrációs eszközként az Operációs rendszerek c. tárgy keretén belül. A programok mindegyikének vannak mind hiányos, mind pedig előnyös tulajdonságai, ez utóbbiak átemelhetőek a létrehozandó demonstrációs eszközbe.

Negatív tulajdonságok:

- a programok nagy része parancssoros felülettel rendelkezik, általában nem rendelkeznek grafikus felhasználói felülettel
- a bemenetek előállításához és a kimenetek értelmezéséhez gyakran van szükség nagy terjedelmű felhasználói dokumentáció értelmezésére
- a szimulációk gyakran annyira részletesek és pontosak, hogy a vele ismerkedő felhasználók könnyen elveszhetnek benne
- a különböző problémaköröket gyakran egyszerre, komplex formában próbálják bemutatni
- túl gyorsak, lefutásuk sebessége nehezen szabályozható
- Windows platformon nem mindig, vagy csak nehezen futtatható
- kevés az előre gyártott bemenet

Egy-egy alkalmazásra jellemző pozitív tulajdonságok:

- rendelkezik több nézetes, vagy könnyen átlátható felülettel

- adott problémakört más területektől függetlenül is képes bemutatni
- képes részletes statisztikák futás közben történő elkészítésére
- képes a lépésenkénti végrehajtásra az automatikus, gyors végrehajtás mellett
- a végrehajtás során esztétikus diagramok kerülnek megjelenítésre

Mivel a kedvező tulajdonságok ezekben az alkalmazásokban csak elvétve fordulnak elő és nincs olyan közöttük, melyre egyszerre jellemzőek lennének, ezért van szükség egy olyan alkalmazás elkészítésére, mely képességeiben jobban illeszkedik az Operációs rendszerek c. tárgy elvárásaihoz.

2.2 Témakörökhöz kapcsolódó ismeretek összegzése

Az alábbiakban összegzésre kerülnek az egyes témakörökhöz kapcsolódó legfontosabb háttérismeretek.

2.2.1 Feladatok ütemezése

Az operációs rendszerek elsődleges feladata a felhasználó számára hasznos taszkok futtatásának biztosítása. Ezek lefutásához különböző erőforrásokat kell biztosítani. Alapvető erőforrásoknak számítanak például a kódjukat végrehajtani képes processzor és megfelelő méretű tárterületet. Egyes erőforrásokat azonban csak az operációs rendszer által nyújtott virtuális gép segítségével, rendszerhívásokon keresztül képesek igénybe venni. Ilyenek lehetnek a be- és kiviteli műveleteket végző hardverek, perifériák használata, továbbá a folyamatok közötti kommunikációt, együttműködést vagy kölcsönös kizárást biztosító algoritmusok stb.. A hardver erőforrásokkal történő kommunikáció nagyban épít a megszakítási rendszerre. A taszkok futásuk során gyakran feltételezik, hogy ezen erőforrásokat csak ők veszik igénybe, így az erőforrások folyamatok közötti elosztása tipikusan az operációs rendszer alapfeladata.

Akármilyen architektúráról legyen szó, az erőforrásokkal történő gazdálkodás a rendszer központi részét képezi. Ezáltal megjelenik a multiprogramozás, mely esetében a taszkokhoz csak logikai processzor és memória kerül hozzárendelésre kizárólagos jelleggel, mivel a fizikai erőforrásokat kénytelenek megosztva használni. E „virtuális” erőforrások megfelelő elosztását az operációs rendszer menedzseli. Ezek igénybe vételére is a virtuális gép szolgáltatásain keresztül, rendszerhívások segítségével van

lehetősége a taszkoknak. Ezért lefutásuk időbeli alakulása tipikusan felosztható CPU intenzív valamint IO intenzív részekre. Az előbbi a processzor, utóbbi pedig más ki- és beviteli egységekhez tartozó műveletek végrehajtásának időbeli alakulására van hatással. Egy periféria állapotának változásai és más események jelzése az operációs rendszer és a folyamatok irányába a hardver által biztosított megszakítási rendszeren keresztül történik. A CPU és IO idők függvényében különböző rendszereknél más-más metrikákra történő optimalizálást kell végrehajtani, ami eltérő algoritmusok használatát jelentheti. Optimalizálás célja lehetnek például processzor-kihasználás, áteresztő képesség maximalizálása, feladatok átlagos futási idejének, válaszidejének vagy határidő mulasztásainak minimalizálása, stb.

Az operációs rendszerek folyamatkezelésének bemutatása rendszerint a sorállási- és az állapotmodell bemutatásával történik. A sorállási modell a rendszer egyszerűsített erőforrás használat, míg az állapotmodell a folyamatok végrehajtását mutatja be.

Az operációs rendszer működése során a sorállási modell szerint többféle ütemező is szerepet kaphat. Ilyenek lehetnek a rövidtávú-, hosszú távú- és középtávú ütemezők. A továbbiakban csupán a folyamatok rövidtávú ütemezésével foglalkozom.

A folyamatok ütemezése az állapotmodell alapján egy állapot átmeneti gráf és egy állapotjelző segítségével kivitelezhető. Ebben a modellben előre definiált állapotok és állapot átmenetek találhatóak, melyek bármely pillanatban képesek a rendszerben levő folyamatokat és futásuk körülményeit jellemezni. Az egyes állapotok – pl. a *Kész*, *Futásra kész* és *Várakozó* állapotok melyek alap állapotoknak számítanak – meghatározzák a folyamatok pillanatnyi tulajdonságait és képességeit. Ez szerint dől el, hogy pillanatnyilag ki használhatja az erőforrásokat, a CPU melyik folyamat kódját futtatja éppen stb. Valós rendszerekben a fentieken kívül előfordulhatnak még *Felfüggesztve futásra kész*, *Felfüggesztve várakozó* és egyéb állapotok is. Egyes események bekövetkezésével – pl. az ütemező kiválasztja a folyamatot futásra, az operációs rendszer elveszi a processzort a folyamattól, a folyamat mond le a futás jogáról, eseményre várakozik, vagy a várt esemény bekövetkezik stb. – a folyamatok a fenti állapotok egyikéből a másikba előre meghatározott módon kerülnek.

2.2.2 Memóriafoglalás

Multiprogramozott rendszerekben ahhoz, hogy több folyamat egyidejű futásához hasonló élményt érjünk el és eközben megfelelő működési sebességet is biztosítsunk, szükség van a folyamatokat végrehajtani képes CPU megosztott használatára. Ennek következménye, hogy a futtatott folyamatot adott időszelet letelte után a CPU felfüggeszti és egy másik kiválasztott folyamat kezdi meg, vagy folytatja futását. Ahhoz, hogy ezt a jelenlegi hardveres megvalósításokkal értelmes időkereten belül véghezvigyük, ajánlatos a lehető legtöbb „párhuzamosan” futtatni kívánt folyamat operatív tárban való tárolása.

A folyamatok központi tárban való tárolásához két főbb kérdéskörben kell döntést hozni. Az első szakasz meghatározni a program címeinek kötésének mikéntjét, azaz eldönteni, hogy a program logikai címei miképpen kerülnek leképezésre tényleges fizikai címekké. Ismertek statikus és dinamikus címleképezési megoldások, melyek meghatározzák, hogy a betölteni kívánt programok a tárban mely területekre töltődhetnek be, memórián belül hol helyezhetőek el, valamint szükség esetén áthelyezhetőek-e a futtatott programok a táron belül. A másik szakasz annak eldöntése, hogy a folyamatok hol, milyen algoritmus szerint kerüljenek allokálásra a tárban. Az allokálás helyére az allokációs stratégián kívül a címkötés módja befolyással lehet (olyan megszorításokat alkalmaz, amelyek korlátozzák, hogy az adott folyamat, kódrészletet csak egy megadott kezdőcímre tölthető be).

A fentiek miatt, a memóriában tárolt folyamatok memória igényeinek csökkentése különböző megoldások születtek. Ilyen megoldások például a késleltetett betöltésen alapuló megoldások, mint például a dinamikus betöltés (dynamic loading), dinamikus könyvtárbetöltés/kapcsolatszerkesztés (dynamic linking), vagy az átfedő programrészek (overlay) használata. Korszerűbb megvalósítás, mely szintén képes csökkenteni a memóriában tárolt folyamatok memóriaigényét a mesterséges folytonosságot (artificial continuity) kiaknázó megoldások használata. Blokkok alkalmazásával például szegmens- és lapszervezésű, esetleg ezek kombinált vagy hierarchikus felépítésű megoldás alakíthatók ki, vagy az ezeken alapuló korszerű virtuális tárkezelés.

Az egypartíciós rendszerek esetén egy időben csak egy folyamat számára biztosítunk elegendő helyet a memóriában. Így a folyamatok közötti váltás nagy

időigénnyel bíró tárcsere alkalmazásával oldható meg, mely jelentős mértékben csökkenti a párhuzamos futás illúzióját. Több partíciós esetben eldönthetjük, hogy minden folyamathoz ugyanolyan méretű, vagy különböző méretű területeket rendelünk, hogy partícióink rögzített vagy változó méretűek legyenek. Címleképezés függvényében eldönthető, hogy a programok a táron belül bárhol, vagy csak bizonyos kezdőcímekre helyezve futhatnak-e, valamint az is, hogy a folyamatok logikai címtartományához rendelt fizikai memória területeknek egybefüggőnek kell-e lenniük, vagy egy folyamat a fizikai memóriában össze nem függő területekből is állhat.

A valós megoldásokban komoly gondot okoz rögzített partíciók esetén a belső, változó partíciók esetén pedig a külső tördelődés. Belső tördelődés akkor keletkezik, ha egy folyamat által igényelt memória nem tölti ki teljesen a folyamathoz rendelt partíciót. Ezáltal a memória egy darabja kihasználatlanná válik, így elérhetetlenné válik a többi folyamat számára. Valamelyest javít a helyzeten, ha az taszkütemező figyelembe veszi a folyamatok memóriaigényét és az alapján választja ki a megfelelő folyamatokat a partíciók mérete szerint. Változó méretű partíciókkal jobb tárkihasználás érhető el és sokkal rugalmasabb a rögzített méretű partícióknál. Tördelődés azonban itt is keletkezik, mégpedig akkor, amikor az ütemező egy folyamat lefutása után felszabadítja az allokált területet. Így a lefoglalt részek között lyukak keletkeznek, melyek előbb utóbb ahhoz vezetnek, hogy a rendelkezésre álló területek annyira eldarabolódtak, hogy már nem rendelhetők hozzá egyetlen folyamathoz sem. A jelenség neve külső tördelődés, mely esetén költséges megoldást jelenthet a szabad helyek tömörítése (garbage collection, compaction). Ez a megoldás sok időt, dinamikus áthelyezhetőséget valamint hardvertámogatást igényel és megzavarhatja a rendszer működését. A folyamatokhoz rendelendő tárterületek kiválasztására különböző algoritmusok kerültek kidolgozásra, melyek csökkentik a külső tördelődést, ezzel optimalizálhatjuk a tárkihasználást.

2.2.3 Virtuális memória címleképezés

A mesterséges folytonosság lehetővé teszi, hogy a folyamatok folytonos logikai címtartományát nem folytonos fizikai címtartományra képezzük le. Korszerűbb hardverelemek segítségével ezeket a nem folytonos címtartományokat, úgynevezett blokkokat – melyeken belül megőrződik a címek megfeleltetésének folytonossága – már könnyebben mozgathatjuk a tár és a háttértár között, ezáltal megközelítve azt a célt,

hogy a folyamatok tárterületének csak futáshoz várhatóan szükséges részei tartózkodjanak a memóriában. A rugalmas tárkihasználás miatt a címek futás időben kerülnek leképezésre, ami mindig többletidővel jár, ezért a gyorsítás érdekében érdemes e célra hardvert használni. A folyamatonkénti blokktábla alkalmazásával kézben tartható a logikai-fizikai címek összerendelésének ideje. A címképző hardver a blokktáblák elérését egy mutatón keresztül végzi, folyamatváltáskor több blokktábla esetén elegendő ennek a mutatónak az átállítása. A virtuális címek egy blokkcímet és egy blokkon belüli eltolást tartalmaznak. A blokktábla tárolja a blokkcímekhez tartozó fizikai kezdőcímeket. Az esetlegesen előforduló érvénytelen hivatkozások feloldását hibamegszakításokon keresztül az operációs rendszer a megfelelő hardver-szoftver összjátékkal teszi a folyamatok számára észrevétlenné.

A fentieken alapulnak a szegmens- és lapszervezésű megoldások, melyekre a virtuális tárkezelés épül. Szegmensszervezés esetén különböző nagyságú blokkok, azaz szegmensek tárolják a programokat azok (eltérő méretű) funkcionális részegységei szerint. Lapszervezésnél a blokkok mérete rögzített, melyeknek a fizikai memóriában fizikai memória keretek felelnek meg. Az azonos méretű blokkokat lapoknak nevezzük, melyek mérete a gyakorlatban 2 hatványainak felel meg. Mind szegmens mind lapszervezés esetén a blokktáblában vagy laptáblában tárolni kell, hogy a hivatkozott lap a memóriában található-e, vagy a háttértáron van. Erre szolgál a Residency bit, vagy a Valid bit. Szegmensszervezés esetén továbbá figyelni kell a túlcímzést is, ez a szegmensek hosszának figyelembe vételével oldható meg, amelyek a blokktáblában kerülnek tárolásra. Lapszervezésnél a kiadható címek nem teszik lehetővé a lapon belüli túlcímzést. Túlcímzést csak a lapcímek tartományában tehetünk, amely a laptábla méretét tartalmazó háttérregiszterrel ellenőrizhető. Több folyamat együttes tárigénye könyvtári eljárások használata esetén jelentősen csökkenthető az osztott szegmens- vagy laphasználatával. Ilyenkor amellett hogy az eljárások fizikailag egyetlen memóriaterületre kerülnek leképezésre, általában a folyamatok is azonos logikai címeken érik el azokat.

Lapszervezés esetén körültekintően kell megválasztani a lapméretet. A címzésre korlátozott számú bitet használhatunk, melyet a végrehajtó egység határoz meg. A bitek egy része a lapok címzésére másik része a lapon belüli eltolás meghatározására használatos. Az előbbi megadja a laptábla méretét, hogy az pontosan hány lap címzésére képes, az utóbbi pedig a lapon belüli maximális eltolás határozza meg, ezáltal egyben a

lapok méretét is. Amennyiben nagy lapmérettel dolgozunk, ugyan csökken a laptábla mérete, de növekszik a belső tördelődésből származó tárveszteség. Kis lapméret esetén több bittel címezhetjük lapjainkat, ezáltal a növekszik a laptábla mérete és csökken a belső tördelődés. Ugyanakkor több blokkban kell ugyanannyi adatmennyiséget a tár és a háttértár között átvinni, tehát csökken a két erőforrás közti adatátviteli sebesség, tehát megnövekszik az adategységre vonatkoztatott keresési (lappangási) idő.

Ez egy hierarchia szint közbeiktatásával könnyen leküzdhetők a nagy laptáblák méreteiből adódó problémák. E megoldásban a lapok címeinek tárolása helyett részlaptáblák címei kerülnek tárolásra. Ennek pozitív következménye, hogy kis részlaptáblák könnyebben elférnek a gyors feldolgozási idővel rendelkező, kevés tárkapacitással bíró hardverekben, így a címfeloldás – kevés hierarchiaszint esetén - még mindig gyorsabban történik, mint a memóriához történő kétszeres hozzáférésnél, ahol a nagyméretű laptábla a tárban került elhelyezésre. Negatív vonzata viszont, hogy az egy laptáblás megoldáshoz képest az effektív hozzáférési idő megnövekszik, ugyanis így egy logikai cím feloldásához már két vagy három laptáblából való információ kinyerésre van szükség. Egy tartalom szerint címezhető asszociatív tár alkalmazásával ez a hozzáférési idő is javítható, amennyiben az a jövőben várhatóan gyakran hivatkozott lapcímeket tartalmazza. Az ilyen speciális tárak hátránya, hogy csak nagyon kevés logikai és fizikai cím összerendelést képesek tárolni, így nem elegendőek egyetlen folyamathoz tartozó laptábla befogadására sem. Amennyiben az asszociatív tárakat a laptáblákkal kombinálva használjuk jelentős sebességnövekedést érhetünk el, ha képesek vagyunk nagy találati arány biztosítani az asszociatív tár számára. Kombinált technika alkalmazásával a lapcím feloldása egy időben kezdődik meg a laptáblában és az asszociatív tárban, így ha az rendelkezésre áll a tartalom szerint címezhető memóriában, az időigényes laptáblákban történő keresés a befejezése előtt leállhat.

2.2.4 Lapcsere algoritmusok

A virtuális címleképezés részletezésekor előzőleg már összefoglalásra kerültek a virtuális tárkezeléssel, azon belül pedig a lapszervezésen alapuló megoldások működésének alapelvei.

A virtuális tárkezelés lap- és szegmensszervezésre épülve lehetővé teszi, hogy a rendszer folyamatainak címtartománya csak részben, a futáshoz éppen szükséges részek

kerüljenek tárolásra a tárban, miközben a folyamatok kicímezhetnek a tartományukban szereplő bármilyen logikai címre. Továbbá lehetővé teszi a fizikai memória méretét meghaladó logikai címtartományokkal rendelkező programok futtatását. A teljes folyamatot az operációs rendszer hibamegszakítások kezelésével a háttérből felügyeli, ehhez speciális hardverekre lehet szükség. Egy folyamat érvénytelen címre hivatkozásakor a címképző hardver hibamegszakítását kezelve az operációs rendszer annak környezetét elmenti, és ha úgy ítéli meg, hogy a megszakítást nem betöltött blokkra történő hivatkozás okozta, a kívánt blokkot behozza, amennyiben van hely számára a tárban. Ellenkező esetben egy áldozatot kiválasztva, elmenti azt a háttértárra, így felszabadítva helyet a betölteni kívánt blokk számára. A perifériális műveletek lassúsága miatt eközben a más a tárban levő folyamat kerül végrehajtásra. A blokk tárba töltésével a folyamatütemező ismételten kiválaszthatja a folyamatot futásra.

A virtuális tárkezelés során megválaszolandó kérdések közé tartozik, hogy melyik lapot kell behozni a tárba, melyik lapot kell kivinni a tárból, ha a behozni kívánt lap számára nincs szabad hely, valamint egy folyamat számára hány lapot érdemes biztosítani, azaz hogyan érdemes gazdálkodni a tárral.

A betöltendő lap kiválasztása során alkalmazható előre tekintő és igény szerinti lapozási stratégia. Ha a behozni kívánt lap számára nem áll rendelkezésre elegendő hely a memóriában, ki kell választani azt a lapot, amelyiket szükség esetén a háttértárra mentve - amennyiben behozatala óta módosult - helyet szabadítunk fel a memóriában. A módosult vagy hivatkozott lapok nyilvántartását a laptáblán alkalmazott jelzőbiteken keresztül (M és R bit) speciális hardver alkalmazásával lehet hatékonyan megvalósítani. Események bekövetkeztével vagy adott időközönként ezeket a biteket az operációs rendszer törölheti (lapbehozatal, laphiba keletkezése, lapcsere algoritmus futása), vagy beállíthatja (lapra írás, hivatkozás).

A rendszer teljesítményét befolyásolhatja, hogy lokális, vagy globális lapcsere algoritmust alkalmazunk. Előbbinél a kivitelre választott lapok a folyamat lapkészletéből, utóbbinál pedig az összes folyamat együttes lapkészletéből kerülnek kiválasztásra. Utóbbi jobb a terhelés elosztásban, hátránya, hogy független folyamatok között kedvezőtlen kölcsönhatást okozhat azáltal, hogy egyes algoritmusoknál a sok lapra hivatkozó folyamatok kiszoríthatják a kevesebb lapra hivatkozókat.

A tárral való gazdálkodás meghatározása nagyban befolyásolja a laphibák keletkezésének számát, ezáltal a rendszer teljesítményét is. CPU tétlenség alakulhat ki,

mely vergődéshez vezethet, ha túl kevés lapot biztosítunk a folyamatoknak, mivel az így a megnövekedő laphibák lassú kiszolgálása miatt, csökken a tárban levő futtatható folyamatok száma. Szintén CPU tétlenség alakulhat ki, ha a folyamatok túl sok lapot kapnak, ekkor ugyanis csökken a multiprogramozás foka. Annak megbecslésével, hogy milyen laphiba gyakoriság mellett marad még a rendszer egyensúlyban, megközelíthető az optimum, amely az adminisztrációs költségekkel csökkentett maximális CPU kihasználtság. A CPU nem marad kihasználatlan, ha nem következik be újabb laphiba egy laphiba kezelése közben, azaz két laphiba közti átlagos futási idő meghaladja a laphiba átlagos kiszolgálási idejét. Egy folyamat munkahalmazának érdemes legalább annyi lapot biztosítani, amennyire egy laphiba kiszolgálás átlagos ideje alatt hivatkozik. Szintén érdemes az eltérő folyamatok változó lapigényét dinamikus lokális tárgazdálkodással megvalósítani, amelyik képes alkalmazkodni a folyamatok aktuális lapigényeihez. Ezt legkönnyebben a laphiba gyakoriság, vagy laphibák között eltelt idő mérésével lehet megközelíteni. Az operációs rendszer így a memóriabőségben levő folyamatoktól lapokat vehetnek el, melyek helyére a memória szűkében levő folyamatok lapjai kerülhetnek, és új folyamatok is csak akkor indulhatnak el, ha van elegendő szabad memóriakeret számukra, akár azáltal is, hogy más folyamatok a bőségesből az optimális tartományba kényszerülnek. Gyakori laphiba esetén az operációs rendszer dönthet a multiprogramozás fokának csökkentése mellett, így egy folyamat összes memóriakerete kimentésre kerülhet a háttértárra.

A rendszer teljesítményét, a megvalósított algoritmusokon kívül egyéb tényezők is befolyásolhatják. Ilyen lehet a megfelelő lapméret megválasztása, mely hatással van a blokkok átvitelének idejére, tördelődés nagyságára, folyamatok munkahalmazának méretére, laptáblák nagyságára stb.. A laphiba gyakoriságra a program kódjának struktúrája is hatással lehet.

3 Követelmények összefoglalása

A fejezetben összegzésre kerül az algoritmusok és metrikák teljes jegyzéke némi magyarázattal kiegészítve, melyek megvalósítása kívánatos a demonstrációs alkalmazásban. Foglalkozok egy tipikus demonstrációs alkalmazás felületre vonatkozó megkötéseivel is.

3.1 Feladatok követelményeinek specifikálása

A feladatok a következő szempontok szerint kerülnek kifejtésre:

- milyen algoritmusokat lenne érdemes tartalmaznia egy, a meghatározott területeket bemutató alkalmazásnak
- milyen szempontok szerint történt a fent tárgyalt feladatok egyszerűsítése
- milyen a megértést segítő jellemzők és metrikák számítását lenne ajánlott a szimuláció lépéseiben automatikusan kiszámítani
- milyen felületi, vagy funkcionális elvárasokat lenne jó az elkészített megoldásnak teljesítenie

3.1.1 Taszkütemezés

Mivel az elsődleges cél az ütemezési algoritmusok bemutatása legfőbbképpen az alap erőforrás, a processzor taszk ütemezésének bemutatásával, ezért a feladat kidolgozás során főleg a taszkok végrehajtásakor alakuló időviszonyok megfigyelésére kerül a hangsúly a rövid távú ütemezési algoritmusok függvényében. Ebből az erőforrásból csak egyetlen egy erőforrás meglétét tételezem fel. Nem kerülnek bemutatásra többprocesszoros rendszerek, ahol a taszkok végrehajtását több egység, vagy több mag is segíti. A tárcsere hiánya miatt a folyamatok így csak a *Futó, Futásra kész,* valamint *Várakozó* állapotokba kerülhetnek, a *Felfüggesztve futásra kész, Felfüggesztve várakozó* és egyéb állapotok nem kerülnek bemutatásra az ütemezés bemutatása során.

Eltekintek a szimulálni kívánt taszkok „kódszerű” végrehajtásától, mivel ezek értelemszerűen nem állnak rendelkezésre. Ehelyett csupán az időszeletek hosszának megadására kerül sor, ezért a demonstrációban nincs lehetőség sem a folyamatok több

szálon történő futásának szimulálására. A bemenő taszkleírásokra úgy tekintünk, mint a programkód egy lefutási lehetőségére. A szimuláció során feltételezhető, hogy a bemenetként megadott folyamatok nem mondanak le önként a futás jogáról. A folyamatok közötti együttműködésre, kommunikációra, szinkronizálásra, kölcsönös kizárásra nincs lehetőség.

A feladat megoldása egy erőforrás készlet köré történő építkezéssel kerül megvalósításra. A szimuláció során a folyamatok igénybe vehetik a szintén bemenetként megadott különböző típusú erőforrásokat. Így a szimulációk során nemcsak a folyamatok CPU időigénye, hanem IO időigényei is megfigyelhetőek. Az egyszerűség kedvéért az operációs rendszer rövid távú ütemezőjének futásakor keletkező adminisztrációs, ütemezési idők és kontextus váltási idők stb. figyelembe vételére csak összevont formában, egy átlagos adminisztrációs idő megadásával van lehetőség. Bár a valóságban nem minden erőforrás típusra igaz, de az egyszerűség kedvéért az erőforrások használatának ütemezése – a processzorütemezéssel ellentétben – csak egyszerű FIFO listák segítségével történik. Továbbá feltételezem, hogy a folyamatok az erőforrásaikat, perifériáikat szekvenciálisan veszik igénybe, egy új erőforrás használata előtt az előzőleg használt erőforrást mindig elengedik. A szimuláció nem foglalkozik az erőforrások használatából következő holtpontra jutás ellenőrzésével.

További egyszerűsítésképpen a különböző be- és kiviteli egységek nem kerülnek megkülönböztetésre, holott a valóságban nagyon is eltérőek lehetnek (blokkos, vagy bitenkénti átvitel, használat idejétől függően folyamatosan, ismétlődően, vagy alkalmanként használt stb.). A perifériák csupán a taszkok megadásban definiált időtartományok alatt kerülnek lefoglalásra. A taszkleírásban megadható, hogy az erőforrás szinkron, vagy aszinkron módon kerüljön lefoglalásra. Előbbi esetben a folyamat a futásához megvárja az erőforrás használati idő leteltét, így a be- és kiviteli egység használatának megkezdésével egy időben szükséges a rövid távú ütemező futtatása is, míg utóbbi esetben a periféria használatának ideje nem kerül kivárásra és a vezérlés a következő CPU szakaszba, vagy egy következő IO szakaszba lép.

Az időviszonyok vizsgálata során olyan technikai részletek, mint a rendszerhívások lefutása és a megszakítási rendszer működése, perifériák rendelkezésre állási ideje stb. nem kerülnek feltérképezésre.

3.1.1.1 Az ütemezési feladatokhoz tartozó algoritmusok összefoglalása

A szimulációk elvégzéséhez és az algoritmusok konkrét értékeken történő futtatásához az alábbi ütemezési algoritmusokat kell implementálni:

- **Legrégebben várakozó (FCFS: First Come First Served / FIFO: First Input First Output)**
- **Körbeforgó (RR: Round Robin)**
- **Legrövidebb löketidejű (SJF: Shortest Job First)**
- **Legrövidebb hátralevő idejű (SRTF: Shortest Remaining Time First)**

A fentiek között mind preemptív és nem preemptív algoritmusok is megtalálhatók. Egyes algoritmusok esetén szükséges további értékek bemenő paraméterként kerülnek megadásra. Az algoritmusok részletesebb leírása a függelékben megtalálható.

3.1.1.2 Az ütemezési feladatokhoz tartozó metrikák összefoglalása

Az algoritmusok szimulációjának minden lépésénél az alkalmazás a lenti metrikák kiszámítását igény szerint végrehajtja. Az algoritmusok különböző tulajdonságainak összehasonlítására szolgáló leggyakrabban használt mérőszámok az következők:

- **Multiprogramozás foka**
- **Központi egység kihasználtság (CPU Utilization)**
- **Átbocsátó képesség (Throughput)**
- **Várakozási idő (Waiting time)**
- **Körülfordulási idő (Turnaround time)**

A feladat megoldása miatt nem implementálható a következő metrika:

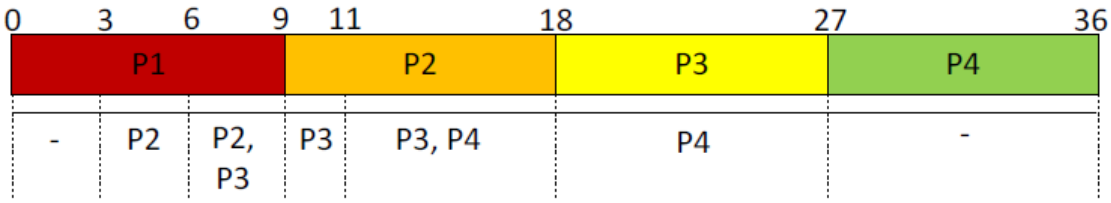
- **Válaszidő (Response time)**

A metrikák részletesebb leírása a függelékben megtalálható.

3.1.1.3 Az ütemezési algoritmusok működésének megértését segítő diagramok

Az algoritmusok szimulációjának minden lépésénél az alkalmazás a lenti diagramok megjelenítésére ad lehetőséget:

- **Gantt diagram**
    - A taszkokat annak függvényében tüntetni fel, hogy milyen időpillanatokban futottak a fizikai processzoron



3.1. ábra: Példa egy Gannt-diagramra [3]

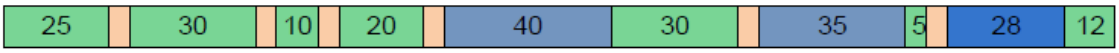
Alkalmazott egyszerűsítés: két taszk futása között minden esetben, valamint preemptív algoritmusok esetén akár ugyanazon taszkot megszakítva a rendszer ütemezője futhat, esetleg más jellegű adminisztrációt végezhet. Valós megvalósítások esetén azonban az itt eltöltött idő legalább egy nagyságrenddel kisebb, mint a taszkok futásával töltött idő, ezért a diagramon e rövid tartományok nem kerülnek megjelenítésre.

3.1.2 Memóriafoglalás

A tárgy igényeinek megfelelően a memóriafoglalás témaköre alatt a leginkább bemutatásra érdemes megoldás a memória, változó méretű partícióin történő, optimális tárkihasználásra törekvő algoritmusok bemutatása. Ebben a megoldásban a külső tördelődést terület felszabadulás után megfelelő területek lefoglalásával optimalizálni próbáló algoritmusok kerülnek bemutatásra. Mivel ennek bemutatásához nem feltétlenül szükséges a folyamatok megkülönböztetése, ezért a továbbiakban a kért memóriaterületek allokációja csupán a felmerülő igények sorozataként kerül értelmezésre. A megoldás elkészítése során az egyszerűség kedvéért eltekintek a szabad helyek tömörítésének végrehajtásától.

A memóriafoglalás részfeladatának megoldásához a legjobb kiindulási alap, ha feltételezzük az operációs rendszerhez érkező memóriafoglalási igények sorozatának kielégítése kerül vizsgálatra a memória foglaltságának aktuális állapota mellett. A memória aktuális állapota nagyban függ az alkalmazott társzervezési megoldástól, valamint a rendszerben levő folyamatok korábbi tárigényeitől. A megoldás során mesterséges folytonosság meglétét nem tételezem fel, tehát abból a feltételezésből indulok ki, hogy a folyamatok nem darabolva helyezkednek el a fizikai tárban és

folytonos memóriaterületet használnak. Az egyszerűség kedvéért a címleképezés megszorításaitól is eltekintek (mint pl. egy logikai címtartomány a fizikai tárban csak adott kezdőcímre helyezhető), dinamikus címleképezés meglétét feltételezem.



3.2. ábra: A felhasználói memória pillanatnyi foglaltságának vizualizációja [3]

3.1.2.1 A memóriafoglaláshoz tartozó algoritmusok összefoglalása

A külső tördelődést csökkenteni igyekvő algoritmusok konkrét értékkel történő futtatásához az alábbi algoritmusokat kell implementálni:

- **Első megfelelő (First Fit)**
- **Következő megfelelő (Next Fit)**
- **Legjobban megfelelő (Best Fit)**
- **Legrosszabban illeszkedő (Worst Fit)**

Az alkalmazás lehetőséget ad további tárallokációs stratégiákkal történő kiegészítésre. Az algoritmusok részletesebb leírása a függelékben megtalálható.

3.1.2.2 A memóriafoglaláshoz tartozó metrikák összefoglalása

Az algoritmusok szimulációjának minden lépésénél az alkalmazás a lenti metrikák kiszámítását igény szerint végrehajtja. Az algoritmusok különböző tulajdonságainak összehasonlítására szolgáló leggyakrabban használt mérőszámok az következők:

- **Memória kihasználtsága**
- **Kihasznált és kihasználatlan területek aránya**
- **Kihasznált partíciók száma**
- **Kihasználatlan partíciók száma**
- **Kihasznált partíciók aránya az összes partícióra nézve**
- **Kihasznált és kihasználatlan partíciók aránya**

A metrikák részletesebb leírása a függelékben megtalálható.

3.1.3 Virtuális memória címleképezés

A megoldásban bemutatásra kell kerülnie a laptáblákkal történő virtuális címleképezésnek. Ehhez szükség van egy folyamat jelenlétére, a hozzá tartozó címtartomány megfelelő szétosztása laptábla segítségével. Mivel a virtuális memória címleképezés egy logikai cím fizikai címre való leképzését foglalja magában, érdemes figyelembe venni a tár aktuális kihasználtságát. Ennek egyszerűsített vázlata kerül megvalósításra.

A megoldás elkészítése során nem foglalkozok más folyamatokhoz tartozó laptáblákkal, ezáltal csak egy folyamat futásán belül a logikai címek laptáblán keresztül történő leképezések sorozatának megfigyelésére koncentrálhatunk. Feltételezem, hogy a felhasználói tárterület egésze a bemenetként megadott lapméretű egységekre oszlik fel. A fizikai tár foglaltságát a felhasználó ilyen lapméretű egységek foglaltságának megadásával állíthatja be, ezáltal nem kerül implementálásra allokációs stratégia. Feltételezem, hogy a tárban a folyamat számára rendelkezésre áll annyi lap, amelyen szereplő „utasításokat” a mellettük levő logikai címekkel végre tud hajtani. Amennyiben a bementként megadott logikai címek elfogynak, vagy egyikük olyan lapra hivatkozik, mely nem tartózkodik a tárban levő lapok között, a szimuláció véget ér.

Mivel csak egyetlen folyamat futása során történő címleképezések sorozatát kívánom szemléltetni, így más folyamatok laptábláival nem szükséges nehezíteni a megértést. Ezáltal nincs szükség a folyamatváltásokkal járó lapcserék, vagy a tárkihasználást javítani szándékozó tárcserék figyelembe vételére sem.

A címfeloldás folyamatát jelentős mértékben gyorsíthatják speciális, e célra szánt hardverek, ezért a szemléltetés során érdemes lehet bemutatni az időviszonyok alakulását. Amennyiben például egy asszociatív tárat (ún. TLB-t) alkalmazunk egy többszintű laptábla elérésének gyorsítására, nagy találati arány esetén jelentős gyorsulást érhetünk el a virtuális címek leképzésekor. Emiatt bemenetként megadható az egyes egységekre vonatkozó hozzáférési idők. Az alkalmazás ezen idők figyelembe vételével képes kiszámítani az aktuális bemenetek segítségével mért hozzáférési időt.

3.1.3.1 A virtuális memória címleképezés során kiszámítandó adatok meghatározása

A címleképezés bemutatása előtt az alkalmazás a lenti adatok kiszámítását hajtja végre:

- **Laptábla minimális mérete**
- **Lapok mérete**

Az címleképezés bemutatásának minden lépésénél az alkalmazás a lenti adatok kiszámítását hajtja végre:

- **Átlagos címleképezési idő**

Az adatok részletesebb leírása a függelékben megtalálható.

3.1.4 Lapcsere algoritmusok

A feladat kidolgozása során nagyrészt inkább az igény szerinti lapozás bemutatására kerül a hangsúly. A valós címekről lapcímekre történő leképezését e megoldás elhanyagolja. A megvalósítás során csupán az egymás után beérkező lapcím igények és a választott algoritmusok függvényében hozandó döntések kerülnek bemutatásra.

Mivel az egyes algoritmusok helyes működéséhez elengedhetetlen a laptáblán tárolt jelzőbitek használata, ezért a megoldásban számolni kell a jelölő bitek használatával is. melyek a következők lehetnek: a lap módosulását jelző bit (**M bit**), a lapra történő hivatkozást jelző bit (**R bit**), a lap memóriába történő fagyasztását jelző bit (**F bit**).

Mivel a feladatot bemutató szimuláció során nem kerül végrehajtásra futtatott folyamat kódja, ezért a fenti bitek beállítását a felhasználó a demonstráció bemeneti információjaként határozhatja meg. Ezek a bitek normális – nem demonstrációs – körülmények között a folyamat kódjának végrehajtása során a folyamat vagy az operációs rendszer által kerülnek beállításra.

A megoldásban a laphivatkozások sorozata és az algoritmusok működése az egyszerűség kedvéért csak egyetlen folyamat lapigényeinek felsorakoztatásával kerül bemutatásra. Hasonlóan a virtuális címleképezést bemutató fejezetben nem kerül megvalósításra több folyamat együttes lapigényeinek kiszolgálásának bemutatása. Amint egy szimuláció az összes bemeneti lapigényét feldolgozta, feltételezzük, hogy a folyamat vagy lefutott, vagy a folyamatütemező által felfüggesztés vagy várakozás állapotába került. A fentiek miatt az itt bemutatott algoritmusok csak lokális lapcsere algoritmusként kerülnek megvalósításra, bár hasonlóan alkalmazhatók lennének globális lapcsere algoritmusok bemutatása során is.

3.1.4.1 A lapcsere algoritmusokhoz tartozó algoritmusok összefoglalása

A szimulációk elvégzéséhez és a lapcsere algoritmusok konkrét értékekkel történő bemutatásához az alábbi algoritmusok kerülnek implementálásra:

- **Optimális algoritmus (Opt: Optimal)**
- **Legrégebbi lap (FIFO: First Input First Output)**
- **Újabb esély (SC: Second Chance)**
- **Legrégebben nem használt (LRU: Last Recently Used)**
- **Legkevésbé használt (LFU: Least Frequently Used / NFU: Not Frequently Used)**
- **Utóbbi időben nem használt (NRU: Not Recently Used)**

Az algoritmusok részletesebb leírása a függelékben megtalálható.

3.1.4.2 A lapcsere algoritmusok alkalmazása során kiszámítandó adatok meghatározása

A feladat megoldása során jó lenne találni olyan metrikákat, amelyek az algoritmusok szimulációjának minden lépésénél kiszámíthatók. A virtuális tárkezelés optimalizálását és a lapcsere algoritmusok hatékonyságának javítását az alábbi metrikák mérésével lehet segíteni:

- **Munkahalmaz mérete**
- **Laphiba gyakoriság (PFF: Page Fault Frequency)**
- **Laphibák között eltelt idő (Interfault Time)**

A fenti metrikák a feladat kialakítása miatt nem mérhetőek. A mérésükhöz szükség lenne a folyamatok kódjának szimulált végrehajtására. Itt azonban a szemléletesség szem előtt tartása végett (és ebből az okból alkalmazott egyszerűsítések miatt) csupán a laphivatkozások sorrendje kerül feldolgozásra. Még az egyes algoritmusok futásához szükséges, laptáblán elhelyezendő bitek sem az utasítások végrehajtásával, hanem csupán manuális beállítás által kerülnek megadásra. Mivel ezek az értékek egy szimulációs ciklus periódusaiban kerülnek feldolgozásra, ezért nincs értelme ezekből az értékekből a fenti metrikákat számolni, mivel még távolról sem kapcsolódnának a processzor által végrehajtott utasítás feldolgozás idő vizsgálatához.

Vannak azonban olyan metrikák, amelyet könnyen lehet alkalmazni, bár a lapcsere algoritmusok minőségéről kevesebb információt árulnak el. Ilyen metrikák a következők:

- **Laphibák száma**
- **Átlagos címleképezési idő**

A metrikák részletesebb leírása a függelékben megtalálható.

3.1.5 További megfontolások

A fentiek alapján alkalmazásunkkal szemben támasztott követelmények legnyilvánvalóbb része, hogy szimulációt tudjunk folytatni beérkező adatainkon. Ennek elengedhetetlen következménye, hogy a bemenő feladatok meghatározásra kerüljenek még a futtatás megkezdése előtt, valamint a szimulációt végrehajtó rendszer is megfelelően legyen konfigurálva. Mivel a bemenetek megalkotása nem képezi a demonstráció központi részért, ezért ilyen célokra nem szükséges felület biztosítani, megelégszünk azzal is, ha e leírókat és definíciókat egy megfelelő leírás alapján egyszerű szerkesztővel mi magunk is képesek leszünk elkészíteni.

Más a helyzet a szimuláció során keletkező információk halmazával. Algoritmusaink futásuk során különböző adatszerkezeteket használnak, melyek tartalma időben változhat. Ezen információk hatékony demonstrációja alapfeladata az alkalmazásnak, ezek számára tehát külön felület biztosítása szükséges.

3.1.6 Felmerülő kérdések

Mivel az alkalmazásban több, egymáshoz szorosan kapcsolódó részterület kerül megvalósításra, felmerül a kérdés, hogy érdemes-e egységes modellt biztosítani e négy terület lefedéséhez, mellyel kellő bonyolultságú feladat megadása esetén mind a négy témát, vagy kiválasztott témákat egyszerre érintő szimulációt végezhetünk.

Ennek megválaszolásához érdemes átgondolni, hogy a kitűzött cél alapján elsődleges feladat egy demonstrációs alkalmazás elkészítése lenne. Oktatási szempontokat figyelembe véve a „minél egyszerűbb annál jobb és érthetőbb” elvét követve e szempontból inkább negatív következményei lennének egy ilyen kritériumnak. Véleményem szerint az együttes szimulációval, azaz egy konkrét szimuláció bonyolultságának növelésével a megértés szintje nem javulna, sőt inkább

romlana, tehát e döntés nem indokolt. Egy másik nézőpontból közelítve viszont érdemes lehet nemcsak demonstrációs célokra, hanem szimulációs célokra is hasznosítani az alkalmazást, mely segítségével ez által komplexebb viszonyok szimulációját és feltérképezését is elvégezhetnénk. Itt felmerül a kérdés, hogy érdemes-e és meg tudjuk-e közelíteni a jelenleg alkalmazott valós megoldások bonyolultsági szintjét anélkül, hogy az itt alkalmazott tárgyalási és bonyolultsági szintet növelnénk. Továbbá meg kell vizsgálni, hogy a modellre nézve milyen megszorításokkal hozható össze a két terület, azaz nem túl nagy-e a fogalmi hézag az egyes területek között.

3.2 Felhasználni kívánt technológiák

Mivel az alkalmazás megjelenésével szemben az az igény merült fel, hogy mindenképpen érdemes lenne grafikus felhasználói felülettel rendelkeznie, ezért a modern technológiákat figyelembe vége leginkább kétfajta megvalósítás kialakításában gondolkodhatunk.

Az első megoldás egy Java technológián alapuló megoldás lehetne, másik pedig a .NET technológia felhasználása. Mindkét technológiával könnyen komponálhatunk látványos felhasználói felületeket. Mivel nem kívánjuk alkalmazásunkat platform függetleníteni, ezért az alkalmazás ez utóbbi felhasználása mentén készül.

A .NET technológián belül a Microsoft Windows Presentation Foundation (röviden WPF) alkalmazása mellett döntöttem, ebben ugyanis a Windows Forms-sal ellentétben a felhasználói felületek kialakítása nem csak hagyományos imperatív megoldásokkal végezhető. Lehetőség van Extensible Application Markup Language (röviden XAML) segítségével deklaratív módon is felületet komponálni, melyet WinForms alapokon aligha lehetne kihasználni.

4 Tervezés részletei

A következő alfejezetekben összefoglalásra kerülnek a tervezéssel kapcsolatos megfontolások és annak részletei. Mivel a WPF alkalmazások fejlesztése gyakran kezdődik a felhasználói felületek megtervezésével, melyet az alkalmazás modelljének megalkotása követ, ezért hasonló módon járok el a tervezés folyamán.

4.1 Funkcionális követelmények

A felhasználók számára az alkalmazásnak a következő lehetőségeket szükséges biztosítania:

- Feladatokhoz tartozó bemenetek betöltése
- Szimulációhoz szükséges nézetek kiválasztása
- Szimuláció futtatása

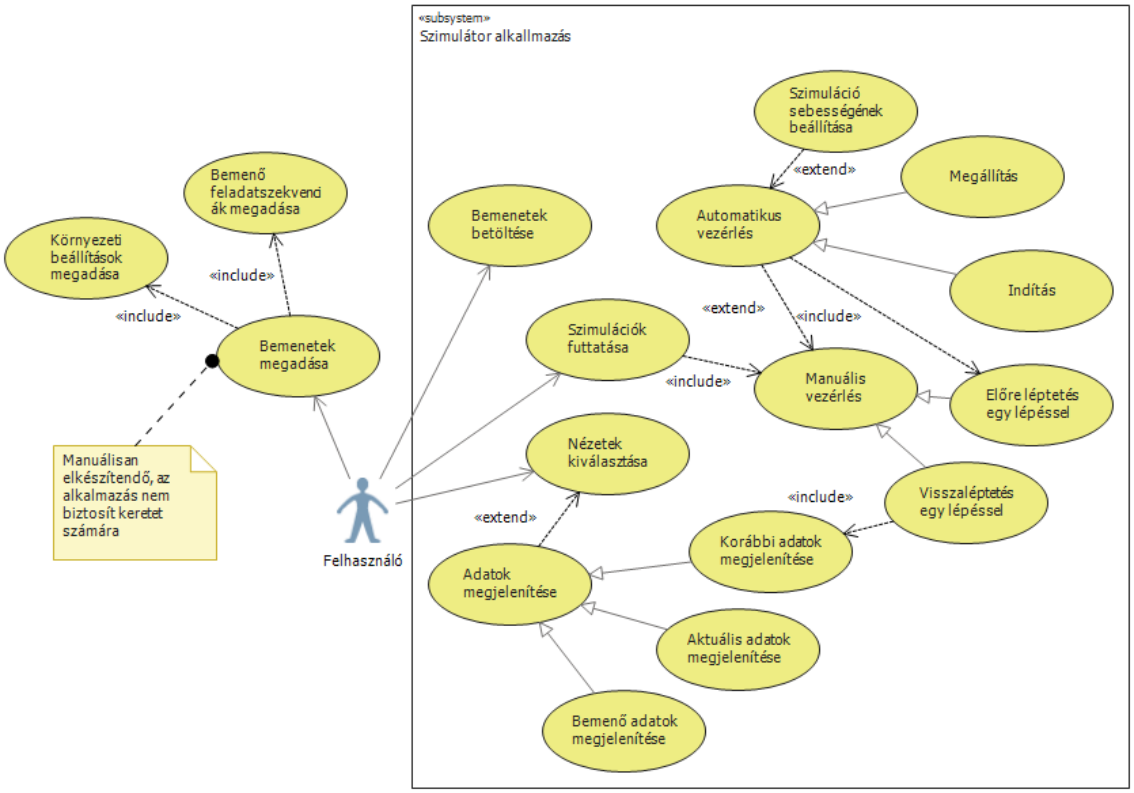
Előzőekben úgy határoztunk, hogy a feladat leírások megadása (egyelőre) nem a felhasználói felületen, hanem manuális úton, egy egyszerű XML szerkesztő segítségével történik. Az alkalmazás felhasználója így megfelelő leírások felhasználásával elő tudja állítani a kívánt bemeneteket.

A 4.1. ábrán látható, hogy az alkalmazás feladatai közé tartozik az előre elkészített XML forrásadatok betöltése. Mivel az alkalmazásnak nincs információja arról, hogy az egyes modulok milyen feladatok végrehajtását képesek elvállalni, ezért a feladat leírások betöltése egy háromlépcsős folyamatban zajlik le.

1. Elsőként az alkalmazás felületén kiválasztott XML fájl sorosítása történik meg, mely később átadásra kerül az azt értelmezni képes modul számára.
2. Második lépésben az alkalmazás kiolvassa a sorosított XML leírás gyökér elemei által kötelezően tartalmazott attribútumait, majd ezek alapján eldönti, hogy van-e azt feldolgozni képes modul.
3. Utolsóként, amennyiben talált megfelelő modult, továbbküldi a sorosított információkat a modul számára, amely miután értelmezte az adatokat vagy elutasítja azokat, vagy létrehozza belőlük a szimulációra használt modellt.

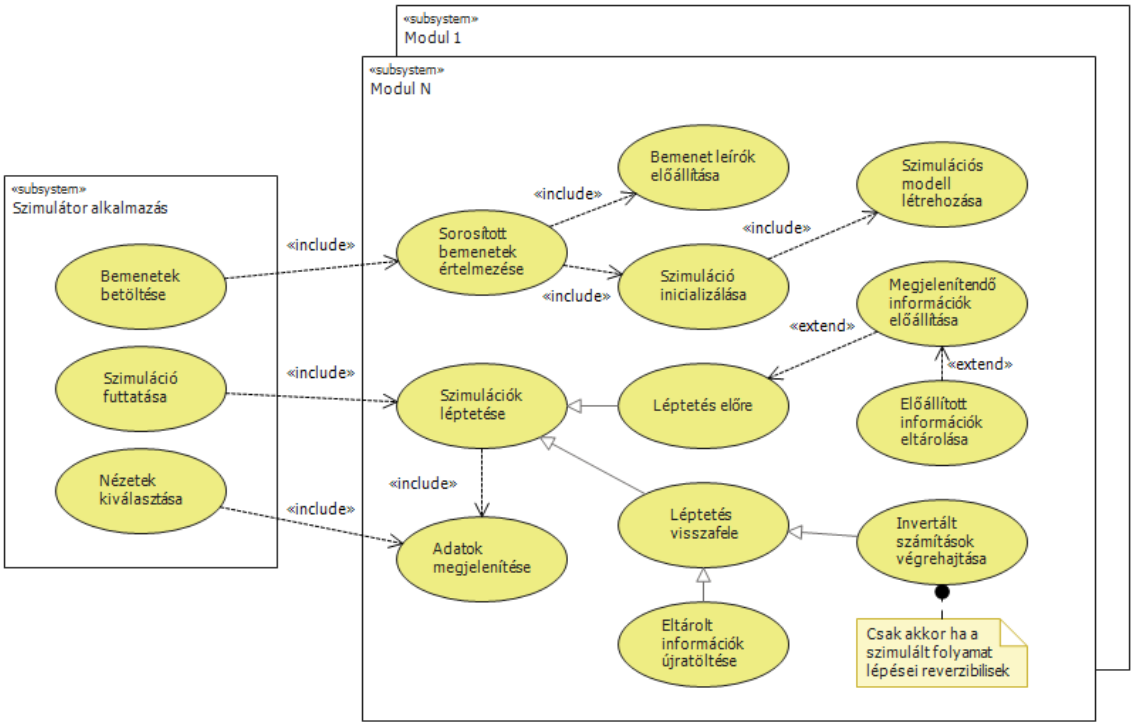
Mivel a felhasználó eldöntheti, hogy milyen nézetben melyik adatokat szeretné megtekinteni, ezért az alkalmazás hatáskörébe tartozik annak eldöntése, hogy a modulok által kínált nézeteket hol jelenítse meg. Miután a felhasználó eldöntötte és kiválasztotta a megfelelő nézeteket (egy szimulációhoz akár több különbözőt, vagy akár minden szimulációhoz az összest egyszerre) más nincs más dolga, mint figyelemmel kísérni a felületen megjelenített adatok változásait.

Az adatokban történő változtatások kikényszerítése a szimuláció futtatásával érhető. Az alkalmazás az események bekövetkeztével jelzi a modulok számára a lépések megtételét. Lehetőség van manuális és automatikus futtatás végzésére is, valamint manuális meghajtás esetén megengedett a visszafele történő szimuláció is, amennyiben a modul ezt támogatja.



4.1. ábra: Az alkalmazás Use Case diagramja

Mivel a szimulációk tényleges kiszámítása a bővíthetőség és elkülönítés miatt modulokba lett kiszervezve, ezért nem csak a fő alkalmazásnak kell funkcionális követelményeket teljesítenie, ugyanez igaz a modulokra is. Így biztosított, hogy a fő alkalmazás a megfelelő interfészeken keresztül képes legyen „meghajtani” a modulokat.



4.2. ábra: Az alkalmazás-modulok Use Case diagramja

A 4.2. ábrán láthatók az alkalmazás által a modulok irányába támasztott követelmények.

A modulok képesek a sorosított bemenetként kapott információról eldönteni, hogy el tudják-e abból végezni a szimulációs környezet kialakítását, vagy sem. Elsőként olyan feladatleíró objektumokba töltik a sorosított bemenetet, amelyek szoros kapcsolatban vannak az XML leírásokkal. Így akár a későbbiekben képesek lennének e leírók segítségével a felületen történő (bemenő) feladat megadásokat kifele is sorosítani. A leírók segítségével megtörténik a bemeneti információk ellenőrzése, azok konzisztenciájának vizsgálata. Amennyiben a modulok a leírókon keresztül képesek értelmezni a bejövő információt, azok alapján megkonstruálják a szimulációhoz használt adatszerkezeteket.

A szimuláció léptetésével az alkalmazás kéri a moduloktól a szükséges lépések megtételét egységnyi idő megtételének függvényében. A fő alkalmazás képes két irányba is (előre és hátra) meghajtani a modulokat, így azoknak fel kell készülni múltbeli állapotok megjelenítésére is. Amennyiben a bemutatandó algoritmusok nem képesek inverz műveletek végrehajtására (márpedig ez a legtöbb bemutatandó algoritmusra igaz), akkor szükség van az adatok lépésenkénti eltárolására is, esetleges

későbbi visszatöltés céljából. Az alkalmazás feltétezi, hogy véges méretű bemenetekre a modulok algoritmusai véges számú lépésben futnak le.

A szimuláció adatszerkezeteinek időbeli változása, pontosabban a szimuláló dinamikus modelljén végzett lépésenkénti transzformációk hatásai megjelennek a felületen. A megfelelő adatszerkezetek felületre történő kivezetése a modulok hatáskörébe tartozik, így erről a szimulációt végző modulnak kell gondoskodnia.

4.2 Felhasználói felület

A felhasználói felület megfelelő kialakítása meghatározó jelentőséggel bír az alkalmazás többi részének megtervezésére nézve. A WPF alapú megoldásoknál a felhasználói felülethez tartozó igények megfogalmazása közvetlenül az alkalmazás üzleti részéhez tartozó funkciók megfogalmazása után következik. Így tehát a felület megfelelő kialakítása nagy hatást gyakorolhat az alatta levő elemek konkrét megvalósításaira nézve. Mindezek mellett természetesen fontos egy olyan felhasználói felület kialakítása, mely nem valósít meg üzleti logikát, független az alatta levő rétegektől, ezáltal könnyen is kicserélhető.

Ha feltesszük, hogy a feladatok demonstrációja során a szimuláció lépéseit vesszük alapul és ezen keresztül szemléltetjük az algoritmusok működését, akkor véleményem szerint a legcélszerűbb megoldás egy több nézetet egyszerre megjeleníteni képes alkalmazás kialakítása. Ez az elv egy több ablakos kivitelben kerül megvalósításra, ahol az alkalmazás fő ablakán keresztül a felhasználó a szimulációk betöltését, kezelését és vezérlését képes elvégezni, míg a korlátlan számban megnyitható mellék ablakok pusztán a konkrét szimuláció adatainak megjelenítéséért felelnek. Ez a két- (vagy több-) ablakos megoldás remek lehetőséget biztosít osztott képernyőn történő demonstrálásra, mely könnyen képezheti egy előadás szerves részét.

4.2.1 Alkalmazás nézetei

Az alkalmazás az operációs rendszer ablakkezelőjére épülve rendelkezik a megfelelő vezérlő funkciókkal, ezáltal az ablakok leállítása/átméretezése stb. egyéb megszokott tulajdonságai biztosítottak, melyek itt most nem tárgyalandók.

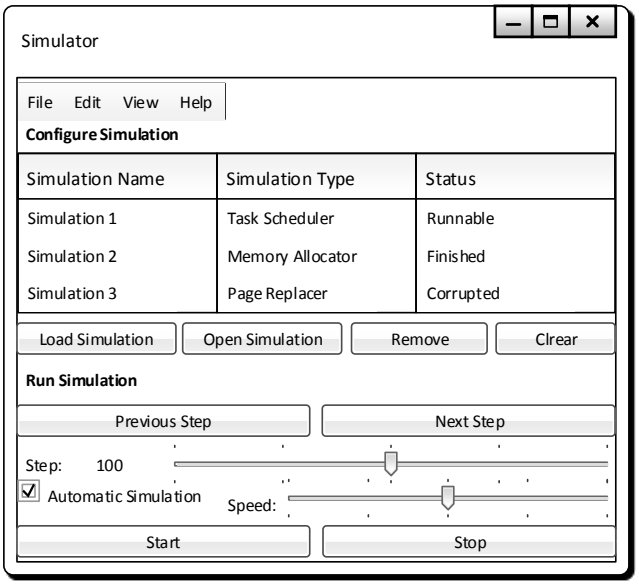
Az alkalmazás a példák hatékony demonstrálása céljából különböző nézetekkel rendelkezik az aktuális feladat típusnak megfelelően. E nézetek be- és kikapcsolása az alkalmazás fő ablakában történik.

4.2.1.1 Az alkalmazás fő ablaka

A fő ablak a szimuláció kezelésére a következő funkciókat biztosítja:

- szimulációs feladatleíró fájlok betöltése
- a szimuláció manuális léptetése előre és hátra
- az automatikus szimuláció elindítása, leállítása, sebességének beállítása
- feladattípusokhoz tartozó nézetek megnyitása és kikapcsolása

A főablak fenti igényeinek egy lehetséges megvalósítása az alábbi drótváz ábrán található:



4.3. ábra: Főablak felülete

4.2.1.2 Az alkalmazás mellék ablaka

Az alkalmazás mellék ablaka a modulok által képviselt tartalom és felület megjelenítéséhez biztosít keretet. A mellék ablak megvalósítása egy héjként képzelhető el, melyben a modulok tartalmakat helyezhetnek el. A beépülő modulok az üzleti logika megvalósítása mellett felületeket is definiálnak, ezért a héj megfelelő kialakításával helyet kell biztosítani e felületeknek.

Ilyen felületek a következők:

- bemeneti adatok
- adatok szimulációs eredménye
- statisztikák és metrikák

- diagramok
- napló

A mellék ablak fenti igényeinek egy lehetséges megvalósítása az alábbi drótváz ábrán található:

4.4. ábra: Héjat tartalmazó mellékablakok sablonja

4.2.2 Modulok nézetei

Az alkalmazás minden modul esetén képes megjeleníteni a bemenetről származó információkat, továbbá felületén keresztül demonstrálható az algoritmus konkrét példákon történő működtetése. Felületén megjelennek az algoritmusok futása során számolt metrikák. Amennyiben a modul futatása közben az aktuális állapotot szemléltető diagramokat is tartalmaz, azok is elhelyezhetők egy erre kijelölt részen. Bár az általam készített feladatok megoldásaiba naplózás nem került beépítésre, amennyiben további feladatok fejlesztői szeretnék kihasználni a lehetőséget, vagy egyéb tartalmakat megjeleníteni, az alkalmazás áttervezése nélkül erre is van lehetőség.

4.2.3 WPF technológia az alkalmazás felületének kialakításában

A WPF [13] ugyan lehetővé teszi többablakos alkalmazások elkészítését, azonban tiszta szerkezet megtartása mellett önmagában csak nagy nehézségek árán képes lazán csatolt módon támogatni a több ablakos, vagy egy ablakban több nézet megjelenítését, főleg akkor, ha azok más modulokból származnak. Ezért célszerű valamilyen keretrendszert alkalmazni a felületek kialakításánál.

A WPF által biztosított Control-oknak, User Control-oknak, Data Template-eknek és Custom Control-oknak, stb. köszönhetően változatos elrendezésű és megjelenésű alkalmazások készíthetők. Nevesített Content Control-okhoz futás időben

tartalmat rendelhetünk. A szimulációk nézeteit így külön megnyíló héjat tartalmazó ablakok helytartó vezérlőihez rendezhetjük. Így a héjak sablonként használhatóak, önmagukban nem tartalmaznak információt, tartalmuk futás időben kerülhet feltöltésre. Tab Control-ok felhasználásával a különböző típusú tartalmak (bemenet, szimulációs eredmények, metrikák, diagramok stb.) az ablakon belül könnyen elszeparálhatóak egymástól.

4.3 A rendszer felépítése

Minden hatékony rendszer – ahol fontos szempont az átláthatóság, könnyű karban tarthatóság és a jövőbeli fejlesztések tervezett egyszerű kivitelezése – megvalósít valamilyen, a szakterületen jártas más egyének által is könnyen értelmezhető mintákat. Nincs ez másképp ennél az alkalmazásnál sem. Az alkalmazás tervezésekor felmerülő problémák egy része már jóval korábban, más alkalmazások tervezésénél is felmerült, ezért fölösleges lenne új megoldásokat kitalálni a gyorsan alkalmazható jól kitalált és bevált megoldások alkalmazása helyett.

Hasonlóan más alkalmazásokhoz, ennél az alkalmazásnál is elengedhetetlen, hogy rendelkezzen egy modellel, amely képes a feladatok leírására oly mértékben, hogy azokon képesek legyünk szimulációk sorát végrehajtani.

Hasonlóan szükséges rendelkeznie egy a megjelenítésért felelős réteggel, amelyik minden időpillanatban képes az újonnan kiszámolt eredmények szemléletes megjelenítésére. Ez a réteg tisztában van azzal, hogyan nyerheti ki a számára hasznos információkat az alatta levő rétegből és azokat miképpen érdemes megjelenítenie.

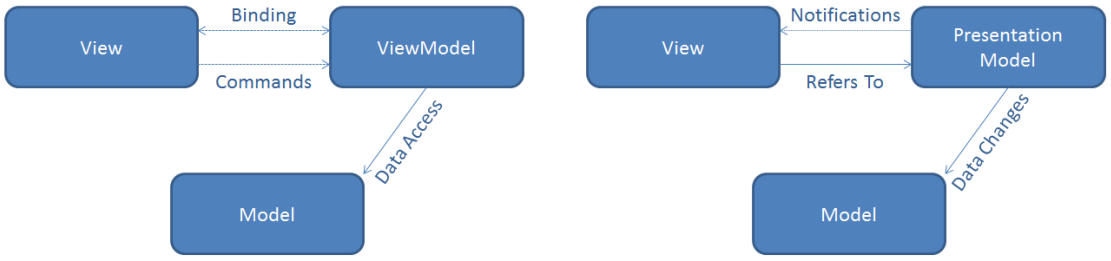
Az alkalmazás utolsó fontosabb részegysége a vezérlést végző egység, melynek hatáskörébe tartozik a felületi elemekből származó információk feldolgozása és modellre történő leképezése. Továbbá hatáskörébe tartozik a modell állapotváltozásairól szóló értesítések eljuttatása a megjelenítést végző rétegek számára.

A fentiek alapján az MVC (Model-View-Controller) architektúra tűnik kézenfekvőnek. Azonban figyelembe kell venni, hogy alkalmazásunkat WPF technológiára építve valósítjuk meg, ezért érdemes az MVC erre a technológiára finomított változatával dolgozni.

4.3.1 Az MVVM architektúrális minta

A Model-View-ViewModel (rövidítve: MVVM) architektúra az MVC architektúra egyik variánsának, a Presentation Model-nek továbbfejlesztett változata. Kimondottan WPF és WPF-E (jelenlegi nevén Silverlight) számára készült és olyan felhasználói felülettel rendelkező alkalmazások fejlesztésére javasolt, melyek esemény vezérelt alapokon működnek. Alkalmazásával tisztán elkülöníthetőek egymástól a grafikus felhasználói felületet, az üzleti logikát (és/vagy adatmodellt) és az alkalmazás logikát megvalósító kódok.

A Model feladata az üzleti logika szakterületet érintő részének, az entitás osztályok, adatforrások, szakterületi modell (domain model) prezentáció független megvalósítása, adatközpontú megközelítésben. A View alá tartoznak a felhasználói felület megjelenítését végző kódok. A tiszta megvalósításban ez a réteg csak a nézettel kapcsolatos kódokat tartalmazza. A ViewModel egyfajta közvetítő szerepet vállal a két réteg között. Megvalósítja az üzleti logika alkalmazás logikához kapcsolható részeit. A nézet számára a modellből elérhetővé teszi a szükséges adatokat, kiegészítve azokat a nézet által igényelt események jelzésével. Továbbá tartalmazza a felhasználói felülethez kapcsolódó állapotokat, így képes a bonyolultabb felületi logikák megvalósításának támogatására, ezáltal egyfajta absztrakt nézetnek is tekinthető. A nézeten keletkező események alapján továbbítja az igényeket az üzleti logika irányába, ezáltal egyfajta átalakítónak is tekinthető. Publikus tulajdonságain és adatkötéseken keresztül képes arra, hogy mentesítse az alkalmazás alsó rétegeit a nézettől való függőségtől, így a rétegek mindig csak az alattuk levő rétegtől függenek.



4.5. ábra: Az MVVM és a Presentation Model architektúrális minták [15]

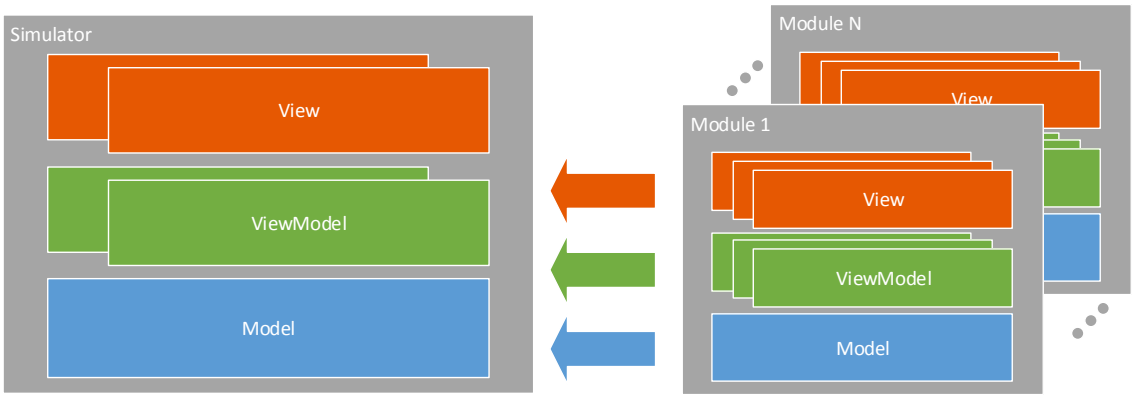
4.3.2 Az MVVM alkalmazása az alkalmazásban

Mivel azt szeretnénk, hogy alkalmazásunk mások által is könnyen érthető, egyszerű felépítésű és könnyen karbantartható legyen, továbbá követelményeink miatt a

későbbiek folyamán történő bővíthetőséget is szeretnénk szem előtt tartani, ezért érdemes a MVVM [17] architektúra mentén történő megvalósításban gondolkodni.

Szerencsére az MVVM architektúra támogatja a modulokba szervezést, ezért célszerű a feladatokat megvalósító szakterületi modelleket (domain model-eket) modulokon belül létrehozni, nem pedig egyetlen alkalmazásba integrálni. Ezáltal az alkalmazás csupán egy keretrendszert biztosít a beépülők számára, így a fő alkalmazásban feladat specifikus információk nem kerülnek tárolásra, mivel azokat a modulok definiálják és használják. A fő alkalmazás interfészein keresztül adja meg a beépülő modulok számára a sikeres beépüléshez teljesítendő kritériumokat. Ezen kívül olyan absztrakt alaposztályokat definiál, melyek közös alapot adnak a modulok által kezelendő adatszerkezeteknek. Így amennyiben a szimulált feladatok halmazát szeretnénk bővíteni, elegendő csupán egy beépülő modult (annak minden részegységével és megkötésével) készíteni az alkalmazáshoz.

A 4.6. ábrán látható, a fő alkalmazás és beépülő moduljainak alapszerkezete. Mivel a beépülők fejlesztőitől nem kényszeríthető ki, hogy moduljaikban a javasolt architektúrát kövessék, így ez a modulok oldalán inkább csak javaslat, mint megkötés. Amennyiben a fejlesztők a megfelelő interfészeket jól definiálják, a modulon belüli MVVM architektúra megtartása nélkül is lehetséges a beépülők létrehozása és működtetése, bár ezt a megoldást nem javaslom.

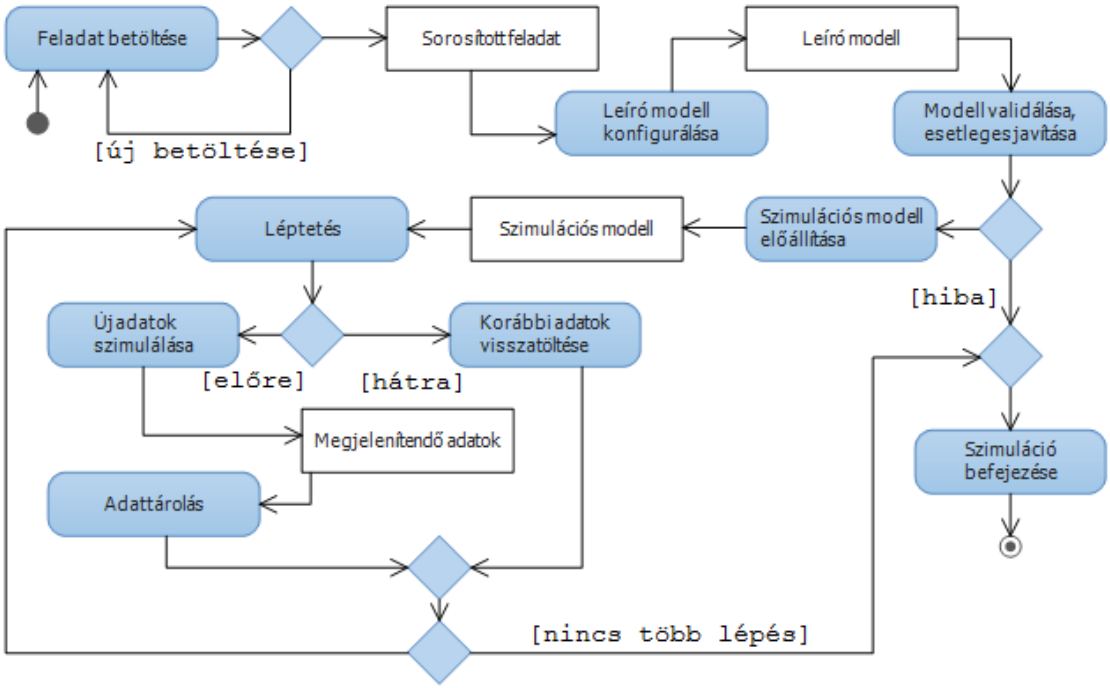


4.6. ábra: MVVM architektúra az alkalmazásban és moduljaiban

4.4 A rendszer modellje

A következőkben összegzésre kerül, hogy a modellnek milyen kritériumokat kell teljesíteni a fenti elvárások, és funkcionális követelmények teljesítése céljából.

A fő alkalmazás a kialakítandó modellektől több részfeladat igény szerinti teljesítését várja el. Mivel ezt oly módon kell lehetővé tenni az alkalmazásban, hogy ezt az elvet a beépülő modulok lehetőleg kövessék, ezért ez az elvárás hatással van a fő alkalmazás modell szerkezetére is, az alkalmazandó interfészek és absztrakt osztályok által. A beépülő modelleket a következő cselekvési sorozathoz illeszkedve kell megvalósítani:

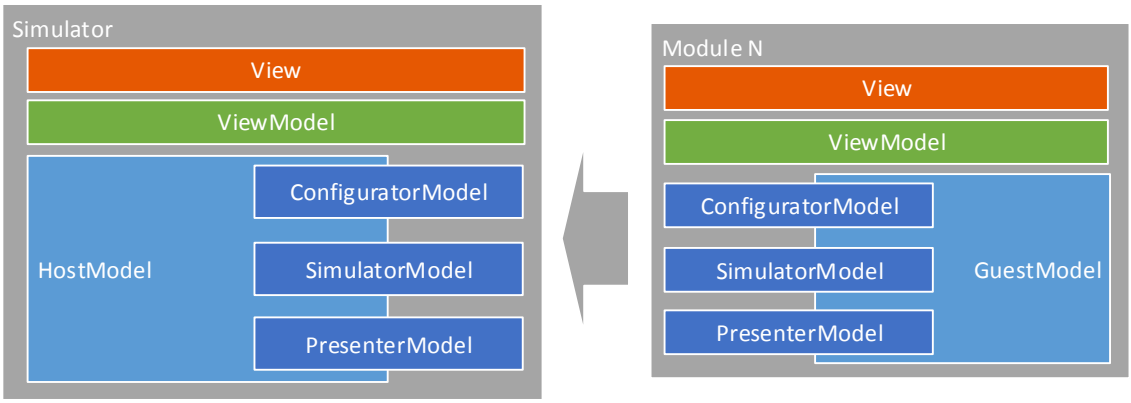


4.7. ábra: A szimulációs alkalmazás és modul együttes működésének Activity Diagram-ja

A legelső lépés a feladatok és konfigurációk betöltése. Ez a fő alkalmazás hatáskörébe tartozik, annak szolgáltatásaira épülve kerül megvalósításra. Amennyiben a fájl sorosítása sikeres, egy leíró modell kerül kitöltésre a feladatot értelmezni képes modulban. Ez egy statikus leírás, tartalmazza a dinamikusabb szimulációs modell előállításának pontos receptjét, miközben illeszkedik a bemenetként használt XML leírások szerkezetéhez is. Ennek segítségével előállítható a szimulációs modell, mely az adatok termeléséért felelős. A szimulációs modell meghajtásával új adatok állnak elő. Erre épül a modell adattárolásért felelős része, amely a megjelenítési rétegek számára lépésenként tárolja a pontos információkat, amennyiben azok fordított irányba történő generálása nem lehetséges. E dinamikusan változó adatok kerülnek kivezetésre a felületre a korábban kivezetett bemenő adatokat leíró statikus adatok mellett.

4.4.1 Üzleti logika a modellben

Minden modulban levő modellnek érdemes illeszkednie erre a szerkezetre, hogy képes legyenek megvalósítani az irányukban támasztott követelményeket. A fentiek miatt a következő modellszerkezetet ajánlott megvalósítani az alkalmazás és a modulok modelljeiben:



4.8. ábra: A fő alkalmazás és a modulok kapcsolódása az üzleti logika domain részén keresztül

A 4.8. ábrán látható, hogy az alkalmazás a modulok modelljeitől az üzleti logikájukat tekintve a következő funkcionális részegységek megvalósítását várja el:

A modell első főbb része (továbbiakban `ConfiguratorModel`) a szimulációs környezet beállításáért és a beérkező feladatok végrehajtó modellbe történő leképezéséért felel. Ez a modell állítja elő a bemenő sorosított objektumból a megfelelő feladatleírásokat, majd ezek alapján ellenőrzi a bemenő adatok helyességét. Amennyiben lehetséges, előállítja a feladatleírásnak pontosan megfelelő szimulációra alkalmas modellt is.

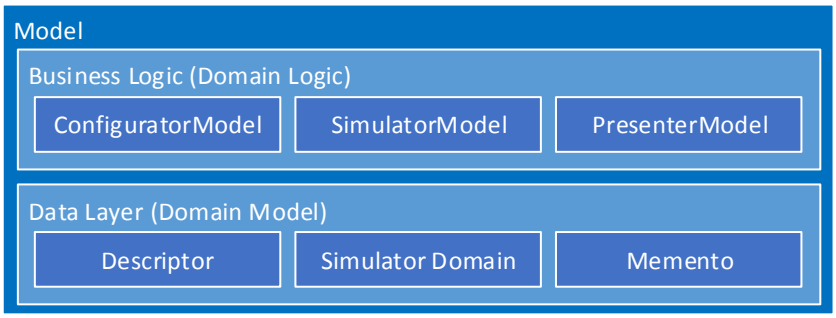
Modellünk második része (továbbiakban `SimulatorModel`), tartalmazza és ismeri az előzőekben a statikus konfigurációs leírás receptje alapján létrehozott dinamikus modellt. Az üzleti logika legfontosabb részét képviseli, ezért képes e dinamikus modellen szimulációkat végezni azáltal, hogy lépésenként meghajtja azt. Tisztában van azzal, hogy az algoritmusok léptetett futásuk során a transzformációkat végeznek a dinamikus modellen, így kialakítva az új rendszerállapotokat. Így termelhetők ki az időben keletkező újabbnál újabb adatok.

Az üzleti modell harmadik része (továbbiakban `PresenterModel`) azoknak a folyamatoknak a lebonyolításáért felel, melyek a szimuláció léptethetőségéből adódó problémákat hivatott megoldani. E modell ismeri azon adatszerkezeteket, melyek a

vizualizáció szempontjából bírnak nagy jelentőséggel. E adatszerkezetek elsődleges feladata az adatok felülethez történő kötése. Az üzleti logika mechanizmusain keresztül biztosított, hogy a korábban kiszámolt eredmények tárolásuk után igény esetén betöltésre kerüljenek, hogy a korábbi lépések eredményeit ne csak a szimuláció megismétlésével tudjuk ismételten elérni. Amennyiben egy szimuláció képes időben visszafele léptetve, inverz módon futni és pontosan ugyanazokat a korábbi adatokat előállítani fordított sorrendben, úgy nincs szükség e modellre. Amennyiben ez nem igaz, biztosítani kell a megfelelő adatszerkezeteket és mechanizmusokat, melyek képesek megoldani ezt a kihívást.

4.4.2 Adatszerkezetek a modellben

Az MVVM-et megvalósító alkalmazások modelljei az üzleti logika szakterületi (domain) logikáján kívül tipikusan entitásokkal is definiálnak, melyeken az üzleti modell működése során felhasznál. Az alkalmazás modelljében a korábban leírtakkal összhangban a következő entitások kerültek definiálásra:



4.9. ábra: A modell adatszerkezetei

Az ábrán levő `Descriptor` a feladatok statikus leírói, mely az XML fájl sorosított objektumának feldolgozásával kerül kitöltésre. Az ellenőrzések elvégése után, a dinamikus `Simulator Domain` objektumai kerülnek megkonstruálásra. Az ezeken történő tényleges futás eredményei a `Memento` objektumaiban kerülnek tárolásra.

4.4.3 Alkalmazás modellje

Mivel a tényleges szimulációk a modulok által kerülnek végrehajtásra, ezért a főalkalmazás az üzleti logika biztosításán kívül nem tartalmaz fontos adatszerkezeteket. Bár szerkezetileg a modulok modelljeivel hasonló felépítésű, azonban csak megkötéseket tartalmaz, melyeket a moduloknak kell teljesíteni, hogy az alkalmazás képes legyen lazán csatolt módon, absztrakt szinten kezelni őket. Információval

rendelkezik a modulok elérési módjáról. Az alkalmazás és a modulok közötti kommunikáció biztosításához interfészeket ír (`IConfigurator`, `ISimulator`, `IPresenter`) elő és absztrakt osztályokat definiál (`DescriptorBase`), melyeket a beépülők valósítanak meg, így adva egységes keretet a moduloknak.

A fentieken túlmenően ez modell szoros kapcsolatban áll a feladatok beolvasását végző szolgáltatásokkal (`IOService`), valamint gondoskodik a modulok azonosításáról és a betöltött feladatok nyilvántartásáról (`Repository`, `SimulationStatus`, `SimulationRecordWithModuleInfo`).

4.4.4 A modulok modelljei

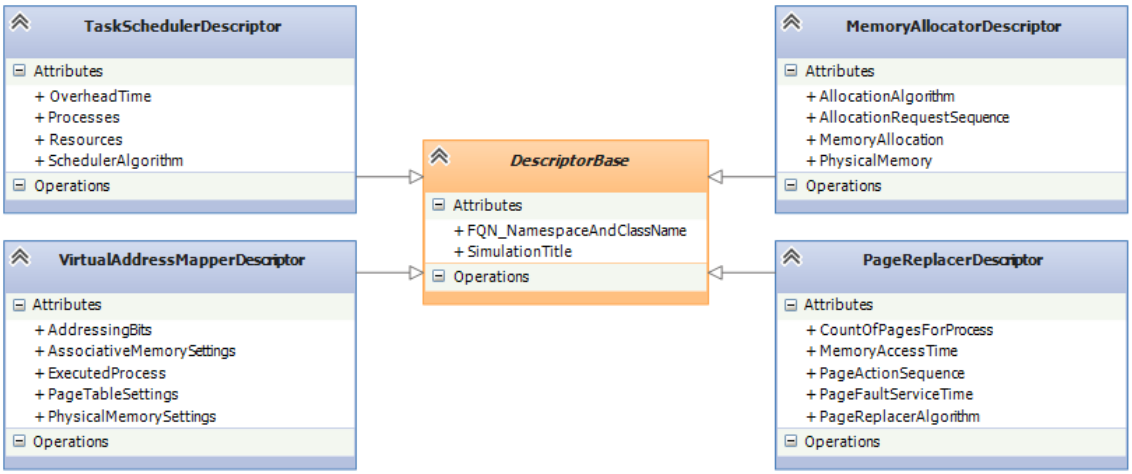
Minden konkrét feladattípushoz egy a modul által megvalósított (`ConfiguratorModel`, `SimulatorModel`, `PresenterModel`) hármas tartozik, melyek a fő alkalmazás által definiált interfészek (`IConfigurator`, `ISimulator`, `IPresenter`) implementációi. Ezáltal minden feladattípus saját maga számára biztosíthatja a betöltött feladatok értelmezését, a szükséges adatszerkezetek létrehozását, a rajtuk történő szimuláció végrehajtását, az adatok tárolását és visszatöltését amennyiben ez utóbbi szükséges.

Bár jelenleg nem támogatott, az alkalmazás továbbfejlesztésével elképzelhetőek olyan konfigurációs és feladatleírások, amelyek nemcsak egy modul modelljére épülnek, hanem egyszerre több modul együttes szimulációját próbálják megvalósítani modulok közötti kommunikációval. Amennyiben egy ilyen összetettebb feladatleírás érkezne be, egy felülvizsgálat során meghatározásra kerülhetne, hogy melyik feladattípusok képesek az együttes szimulációra, majd e információk alapján történhetne a különböző feladattípusok szerinti hármasok inicializálása és használata. Ez esetben minden modulban a `ConfiguratorModell` gondoskodna a bemenő adatok megfelelő `SimulatorDomain`-ekbe történő leképezéséről, a `SimulatorModel` pedig biztosítaná a különböző modellek együttfutásának lehetőségét.

4.4.5 Feladatok leíró, statikus modelljei

Mivel a dinamikus modelleket létrehozó folyamatok a bemenő statikus feladat leírások által tartalmazott paraméterek függvényében nagyon sokféle kimenetet produkálhatnak, ezért fontos e paraméterek megfelelő leírása. A továbbiakban

definiálásra kerülnek a részfeladatokhoz tartozó bemeneti paraméterek, a hozzájuk meghatározott értéktartományok és korlátok.

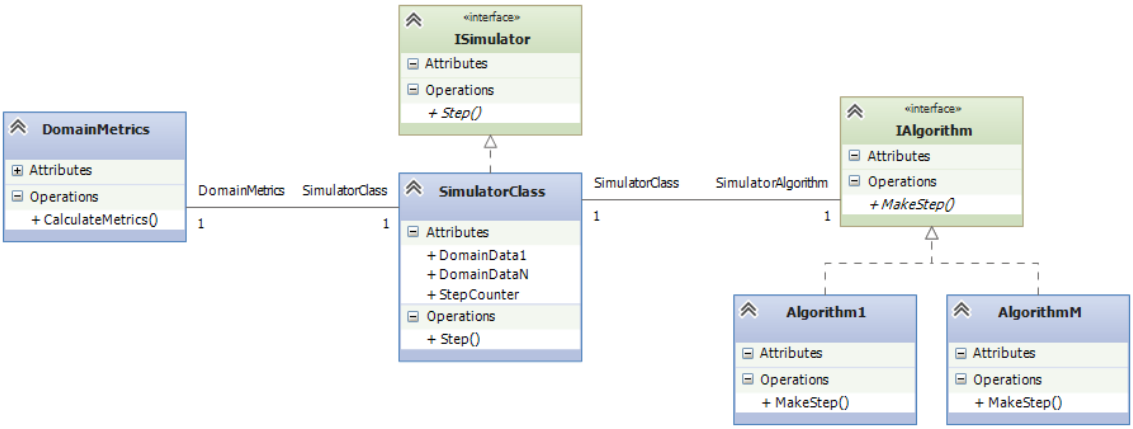


4.10. ábra: A BaseDescriptor megvalósítása feladatonként különbözőképpen

4.4.6 Feladatok szimuláló, dinamikus modelljei

A 4.10. ábrán látható, ahogy az egyes feladatok az alkalmazás által rendelkezésre álló bemenő adatokat leíró adatszerkezetet kiegészítik. Az alkalmazás a `DescriptorBase` segítségével képes a bemenő feladatok eljuttatására az azt feldolgozni képes modulokhoz.

A különböző feladatok Simulator Model-jei tipikusan a következő sémára illeszkedve kerültek megvalósításra:



4.11. ábra: A feladatok dinamikus modelljének tipikus felépítése

Bár egyes feladat megoldásokban a sémától alkalmanként eltérés tapasztalható (hiányozhat annak egy-egy eleme), ez a kialakítás megkönnyítette a különböző feladatok hatékonyabb és átláthatóbb végrehajtását.

A 4.11. ábrán jól látható, hogy viszonylag nagy a függőség a domain model elemei között (mivel az algoritmus a saját adatszerkezetein kívül akár a SimulatorClass adatszerkezeteit is elérheti és módosíthatja), így az egyik osztály megváltoztatása hatást gyakorolhat a többi osztályra is. Ez a szorosan csatoltság sajnálatos következménye annak, hogy megpróbáljuk a különböző algoritmusok futását közös alapokra helyezni, miközben futásuk során teljesen eltérő adatszerkezeteket is használnak, melyeket más komponenseknek tudni kell értelmezni. Mivel a nem algoritmus által számoltakat minden esetben el kell végezni, mint például a metrikák kiszámítását, ezért nem lehet őket teljesen függetleníteni az szimulálni kívánt algoritmusoktól.

4.4.7 Feladatok megjelenítő modelljei

Mivel a teljes körű demonstrációhoz nélkülözhetetlen a visszafele irányba történő lépkedés, azért ezt az igényt szükséges modell szintjén is kielégíteni. Amennyiben egy algoritmus nem képes inverz lépések megtételére, úgy a legfrissebb állapotuk sorrendezett tárolására van szükség.

Ezt a problémát oldja meg a **Memento** tervezési minta, amely képes a szükséges állapotokat eltárolni, azokról nyilvántartást vezetni és igény esetén újra visszatölteni, hogy a felületre kivezetve ismét megjelenhessen egy korábbi állapot.

A minta alkalmazásban történő implementációjára nincs semmilyen megkötés, így ha a modul tervezője úgy dönt, hogy azt a SimulatorModel-en belül szeretné megvalósítani, nyugodtan megteheti. Az itt javasolt megoldás inkább külön osztályok implementálását javasolja, esetlegesen a Simulator Domain osztályainak felhasználásával.

4.5 A részegységek közötti kommunikáció

A következőkben kifejtésre kerül, hogy az előzőleg definiált különböző részegységek hogyan kapcsolódnak egymáshoz és milyen mechanizmusok szükségesek az alkalmazás működésének biztosításához.

4.5.1 WPF technológiák az alkalmazás felületének és egyéb részeinek kialakításában

Szerencsére mind a Prism [11], mind pedig az MVVM Light Toolkit keretrendszer jó lehetőségeket biztosít tisztább szerkezetű megoldások elkészítésére. A

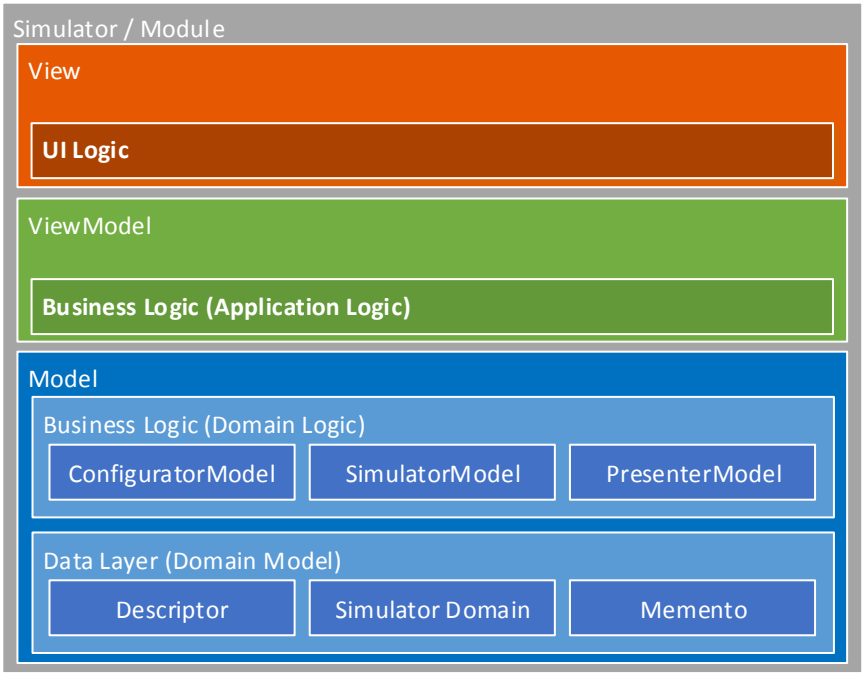
Prism régióin keresztül területeket definiálhatunk nézeteinkben, melyek tartalmát függőség befecskendezés útján MEF [12], Unity vagy más konténer segítségével futás időben határozhatjuk meg. Az így képzett héjba olyan modulok, beépülők felhasználásával adhatunk tartalmat és funkcionalitást, melyek nem részei az alkalmazást tartalmazó szerelvénynek. A Prism további, modulok közötti függetlenséget biztosító szolgáltatások sorát nyújtja, mint például Dependency Injection Container, Service Locator, többfajta navigációs megvalósítás (állapot alapú, nézet alapú), a lazán csatolt kommunikációt biztosító Event Aggregator, vagy a Commanding. Így a WPF Prism-mel történő kiegészítésének használata tökéletesen alkalmas kompozit alkalmazások megvalósítására, mely segítségével üzleti oldalról igényes alkalmazások fejleszthetők.

Mivel a Prism-mel történő kivitelezés során technikai nehézségek merültek fel, egyszerűbbnek láttam elállni az „üzletileg szép” alkalmazás továbbfejlesztésétől és áttérni a szerényebb képességekkel rendelkező MVVM Light Toolkit [16][16] használatára. Bár ez a keretrendszer nem rendelkezik minden támogatással, hogy a modulokat teljesen lazán csatoltan kezelhessük, számos könnyítést biztosít e cél megközelítése érdekében. Ezek közül főkét a Messaging és a Commanding megoldásai kerültek alkalmazásra. Mivel ez a keretrendszer sajnos nem tartalmaz a Prism régióihoz hasonló szolgáltatásokat, ezért a héjak az üzletileg bevett megoldástól kissé eltérő technikai megoldás alkalmazásával lettek kialakítva, melyhez fel kellett adni a modulok futás idejű betöltésének elvét.

4.5.2 Kommunikáció az alkalmazásban és moduljaiban

Mivel az alkalmazás az MVVM architektúrára alapozva került megvalósításra, így belső kommunikációjában is a javasolt megoldásokat követi.

A modell réteg megvalósítja az üzleti szempontból igényelt entitásokat és szolgáltatásokat, nem rendelkezik információval az alkalmazás aktuális állapotát tekintve. A nézet az alatta levő réteg által biztosított adatokat jeleníti meg, felelősségi körébe csupán az adatok helyes megjelenítése tartozik, az alkalmazás futásának kezelése nem az ő dolga. Ez a köztes réteg, a ViewModel feladata, amely lefordítja a felületről érkező instrukciókat a modell által definiált szolgáltatásokra és metódushívásokra, cserébe elérhetővé teszi az alatta levő réteg objektumait a nézet számára és felruházza azokat a változás értesítésének képességével.



4.12. ábra: Az alkalmazás teljes architektúrája

A ViewModel nem rendelkezik referenciával a felette levő nézetekre, mivel nem ismeri azokat. Az irányukba történő kommunikáció az MVVM Light Toolkit Messaging szolgáltatásával, üzenetküldés útján történik. Ez a megoldás ugyan hasonlít a .NET által is támogatott event-eken keresztül történő kommunikációra, azonban annál kifinomultabb módon támogatja a lazán csatoltságot.

A nézetek adat kontextusukon keresztül ugyan közvetlenül is elérhetik az alattuk levő ViewModel-eket, az igényeiket mégsem direkt metódushívásokkal, hanem köztes objektumokon keresztül, a **Command** tervezési mintát megvalósító RelayCommandokon adják a ViewModel-ek tudtára.

4.5.3 Kommunikáció az alkalmazás és a modulok között

Mivel a fő alkalmazás futásának akkor van tényleges haszna, ha az képes a moduljaival együttműködve a szimulációk végrehajtására, ezért elengedhetetlen a modulok és az alkalmazás közötti kommunikáció. Ahogy majd a későbbiekben is láthatjuk, az alkalmazás nem rendelkezik közvetlen referenciával sem a modulok nézeteire, sem azok alkalmazás logikáját megvalósító objektumokra, ezért a kommunikáció csak itt is csak üzenetküldés útján oldható meg.

Az üzenetek feldolgozása és küldése a modulok és az alkalmazás oldalán is a ViewModel-ekben kerülnek megvalósításra. Mivel a ViewModel-ek nem rendelkeznek

direkt referenciával a nézetekre, így az ablakok létrehozása, nézetekkel történő feltöltése és azok ViewModelhez történő csatlakoztatása külön nehézségeket okoz, melyeket gondosan csomagolt esemény paraméterek és **Singleton** objektum alkalmazásával lehet feloldani.

4.5.4 Alkalmazáslogika

Az MVC architektúrában levő Controller-hez hasonlóan, nem meglepő módon itt is a ViewModel feladata e funkciók betöltése. Egyetlen különbség, hogy az MVVM architektúránál az alkalmazás logika már nincs egybeintegrálva az üzleti logika domain logic-jával, mivel az Model szintre került. Feladata az absztrakt nézet modell megvalósításán kívül a felülettől érkező igények megfelelő összekötése az üzleti logika szakterülethez kapcsolódó részeivel.

4.6 Az alkalmazás fejlesztése során felhasznált további tervezési minták

Az előzőekben már volt szó, hogy az alkalmazás üzleti igények miatt megvalósította a Memento tervezési mintát és indirekt használta a Command mintát is, mely az MVVM Light Toolkit segítségével RelayCommand formában állt rendelkezésre. A feladat megoldása során felmerülő problémák miatt az alábbi tervezési minták kerültek alkalmazásra:

- **Memento**
    - A nem invertálható lépések esetén az állapotok eltárolását, igény esetén azok visszatöltését segíti. Az alkalmazásban a modulfejlesztők többféle módon implementálhatják, egy ilyen megoldás, ha az Originator a `SimulatorDomain`-ben, a CareTaker pedig az `IPresenter` interfészt megvalósító osztályban kerül elhelyezésre.
- **Command**
    - A tervezési minta csak felhasználásra került, implementációját az alkalmazott keretrendszer tartalmazza
- **Abstract Factory**

- Ez a tervezési minta biztosítja, hogy a fő alkalmazás részei függetlenek legyenek a modulokban definiált objektumoktól. Mivel az alkalmazás fejlesztésének idejében nem ismert az összes lehetséges implementálható modul leírói, így azok létrehozását a modulokra bízzuk, mivel azok alkalmazásba drótozása nem lehetséges

- **Singleton**
  - Az alkalmazásban egy ilyen objektum tartja nyilván a már példányosított ViewModel-eket, hogy egy ablak bezárása, majd ugyanazon feladat megnyitása után, ne kelljen újra létrehozni a nézet alatt levő rétegeket, ezáltal a feladat szimulációját végző objektumokat sem. Segítségével bármilyen nézet elérheti és lekérdezheti az aktuális ViewModel-ek listáját. (Helyettesíti a Prism DI Containerét, vagy Service Locator-át)

- **Facade**
  - A modulok által definiált `ModuleViewModel`-ekben fogja össze a modulokon belül implementálandó interfészeket, mint pl. az `IConfigurator`, `ISimulator` és az `IPresenter`. Célja az egyszerűsítés, hogy az alkalmazás és a modulok közös felületen legyen képesek egymással kommunikálni. (Kiváltja a Prism és a MEF együttesét)

5 Implementációs részletek

A következőkben kifejtésre kerül néhány a tényleges megvalósítással kapcsolatos részlet, melyek az alkalmazás helyes működéséhez kiemelkedő mértékben járulnak hozzá.

5.1 A bemeneti fájlok szerkezete

Mint már korábban is esett róla szó, a bemeneti információk XML formátumban kerülnek megadásra. Minden feladattípus különböző XML leírásokat képes feldolgozni. Azonban arra is szükség van, hogy ne csak a modul, hanem az alkalmazás is felismerje, hogy az adott XML leírást melyik modul számára készítették, hogy a megfelelőhöz tudja továbbítani a fájlt tartalmazó sorosított objektumot. Ez az egyik oka annak, hogy a fájlok leírói a `DescriptorBase` absztrakt osztályból származnak, így biztosítva azt, hogy az XML fájlok gyökér eleme a szükséges információkat tartalmazza. Ezek az információk a következők:

- **[gyökér elem neve]** – megadja a szerelvény nevét, amely a modult tartalmazza
- **FQN_NamespaceAndClassName** attribútum – megadja a szerelvényen belül az alkalmazás által igényelt `Module` interfész megvalósításnak a helyét, a névtér és osztálynév, pontokkal elválasztott formában
- **Title** attribútum – megadja a szimuláció megnevezését, amely megjelenik a programban

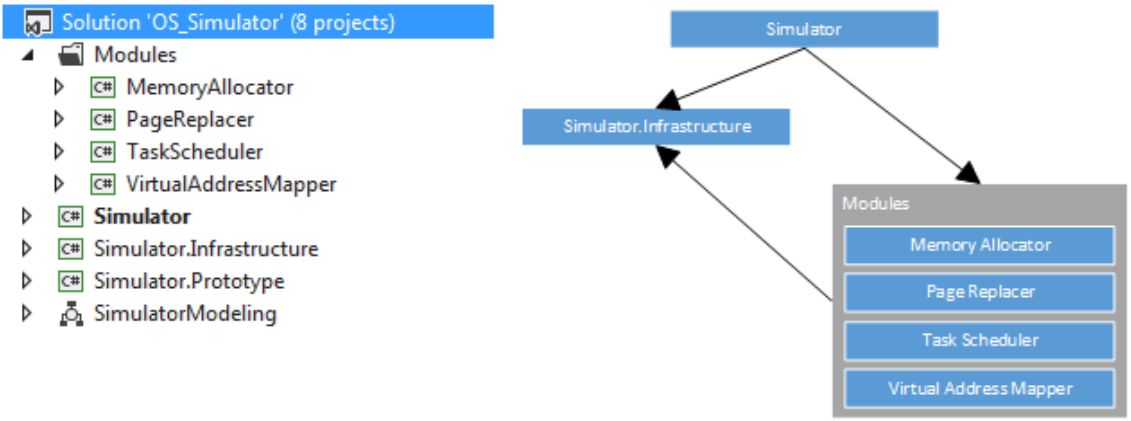
A két utolsó attribútum együttese az alkalmazásban összetett kulcsként azonosítja a szimulációt. Példa a három érték használatára:

```
<?xml version="1.0" encoding="utf-8"?>
<PageReplacer
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                FQN_NamespaceAndClassName="PageReplacer.PageReplacer"
                Title="SimComp1 FIFO(4)" > <!-- … --> </PageReplacer>
```

Ahhoz, hogy ha valaki képes legyen új XML fájlokat létrehozni, érdemes érvényesíteni a készített leírásokat, mielőtt még betöltésre kerülnek. Az ehhez szükséges XSD leírások elérhetőek a mellékleten az `inputs\validation` könyvtárból.

5.2 A szerelvények közötti függőség

Az alkalmazás fejlesztése során a modulok az alkalmazástól elkülönítve, más-más projektekben kerültek elhelyezésre. Ezért a projektek a más projektek által megvalósított funkciók elérése miatt referenciákat tartalmaznak. Az alábbi képen láthatóak az egyes projektek és a projekteken belül más projektekre elhelyezett referenciák:



5.1. ábra: Függőségek az alkalmazás projektjei között

- **Simulator** projekt – tartalmazza a WPF alkalmazást
- **Simulator.Infrastructure** projekt – tartalmaz minden olyan interfészt és absztrakt osztályt, mely a modulok számára szükséges lehet a beépülés megvalósításához, vagy az alkalmazás és modulok közötti kommunikációért felelős. Itt kerültek létrehozásra a fájlbeolvasáshoz (`Services.IOBaseService`), modulok kezeléséhez (`Module`) és a feladatok nyilvántartáshoz (`Repository.SimulationStatus`, `Repository.SimulationRecordWithModuleInfo`) szükséges interfészek is. Alaptámogatást nyújt a beépülő ViewModel-eknek a WPF felületi értesítéseinek elvégzésére (`Notifier`) is.
- **Modul** projektek – tartalmazzák az előbbi interfészek és absztrakt osztályok modulok számára szükséges részeinek konkrét megvalósításait, így épülve be az alkalmazásba.

5.3 Lazán csatolt komponensek közötti kommunikáció

Az MVVM architektúrából fakadóan sem a modell, sem pedig az absztrakt nézet nem rendelkezik referenciával a felette levő rétegekre. Ennek megfelelően a felső rétegek irányába történő kommunikáció lazán csatolt módszerrel, üzenetküldés útján került kialakításra az MVVM Light Toolkit Messenger szolgáltatása segítségével.

A nézetek így képesek a metódusok megfelelő időben történő végrehajtására. Ilyen például egy mellék ablak létrehozása az Open gomb megnyomásakor.

Üzenetküldés rövidített implementációja a MainViewModel-ben:

```
private void sendOpenShellWindowMessage(){
    // . . .
    Messenger.Default.Send(new OpenSimulationMessage(simulationRecord));
    // . . .
}
```

Üzenetküldő esemény meghívása a nézethez csatolt Command-on keresztül a MainViewModel-ből:

```
private RelayCommand previousStepCommand;
public RelayCommand PreviousStepCommand
{
     get
     {  return  openSimulationCommand??  (openSimulationCommand  =  new
RelayCommand(this.sendOpenShellWindowMessage,
this.isAtLeastOneRecordSelected));
     }
}
```

Esemény elkapása és a szükséges metódusok meghívása a nézetben:

```
public MainWindow()
{
    InitializeComponent();
    // . . .
    Messenger.Default.Register<OpenSimulationMessage>(this, message =>
    {
        openShellWindow(message);
    });
    // . . .
}
```

Az alkalmazásban nemcsak a lazán csatolt rétegek között, hanem a fő alkalmazás és a modulok között is szükség van kommunikációra. A modulok közötti horizontális kommunikáció egyszerű metódushívások segítségével történik, miután az alkalmazás Reflection [14] segítségével felderítette a rendelkezésre álló modulokat. A fő alkalmazás csak a megfelelő interfészeken keresztül kommunikál a modulokkal,

melyek az `Infrasructure` projektben kerültek deklarálásra. Erre példa a léptetés megvalósítása:

```
private void NextStep()
{
  if (SelectedItems != null)
  {
    foreach (SimulationRecordWithModuleInfo simulationRecord in
SelectedItems)
    {
      ShellViewModel shellViewModel =
shellViewModelSingletonContainer.GetOrCreateShellViewModelInstance
(simulationRecord);
      shellViewModel.NextStep();
    }
  }
  StatusText = "Next state calculated.";
}
```

5.4 Ablakok létrehozása és tartalommal feltöltése

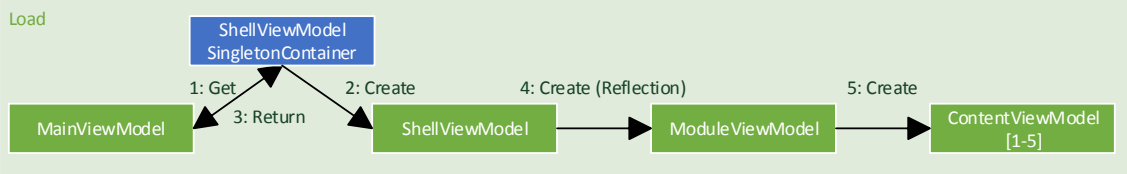
Az alkalmazás megvalósítása során komoly gondot okozott annak kivitelezése, hogy a megnyíló ablakok mindig a helyes információkat tárolják. A probléma forrása az MVVM architektúra és a lazán csatoltság igényének együttese.

A ViewModel-ek nem rendelkezhetnek referenciával a View-kra, csak a View-k érhetik el a ViewModel-eket DataContext-jükön keresztül. Az új nézet létrehozásáról azonban a ViewModel-ben születik döntés, melyet az alkalmazás főablakát megvalósító nézethez kell eljuttatni (melyhez metódushívást nem alkalmazhat, mivel lazán csatoltság és MVVM miatt nincs referenciája a nézetre). E információ alapján kell, hogy a nézet létrehozza a Shell-t, majd ehhez a héjhoz olyan nézeteket csatol, melyek másik modulokban kerültek kialakításra. További probléma, hogy a nézetek létrehozásával az egy ablakos alkalmazások esetében általában automatikusan megtörténik az adatkontextus kialakítása is a ViewModel-ek példányosítása mentén. Erre a mechanizmusra azonban a második ablak tartalmának kialakításánál nem lenne jó építeni, mivel így egy már korábban más ablakban megnyitott, de ugyanezt a szimulációt képviselő ablak tartalma független lenne az ugyanehhez a szimulációhoz megnyitott másik ablak tartalmától, ami azt sejtetné, hogy egyazon szimuláció bejegyzéshez a háttérben több különböző szimuláció kerül hozzárendelésre.

Mivel több ablakos alkalmazásról van szó és a bonyolultságot főleg az MVVM architektúra felépítése okozza, így megfontolandó más architektúra használata. Az MVC sajnos nem elég modern a WPF alkalmazásokhoz, a tiszta MVP pedig képtelen

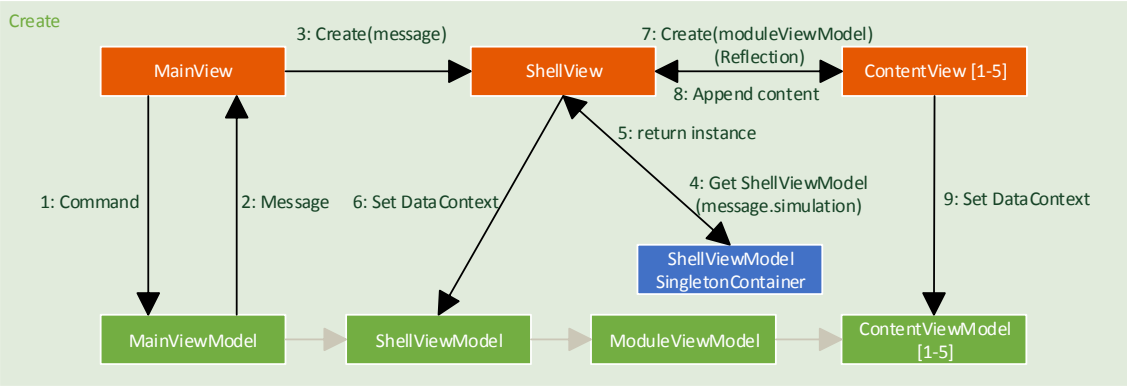
többablakos alkalmazás kezelésére minden osztályt ismerő Presenter nélkül, így kénytelen voltam ragaszkodni az MVVM-hez. A Prism ugyan tartalmaz megoldást a lazán csatoltság leküzdésére, de ezt a keretrendszert is főleg egy ablakos, kompozit alkalmazások fejlesztésére szánták, így a felmerülő technikai nehézségek leküzdése nem volt egyszerűen megoldható.

A kritériumok teljesítéséhez egyedi megoldást kellett kitalálni, ami a következő:



5.2. ábra: A ViewModel-ek létrehozásának lánca betöltés után

A folyamatok betöltésekor a `MainViewModel` a `SingletonContainer` segítségével létrehozza a héjhoz és a szimulációkhoz tartozó ViewModel-eket, még mielőtt a hozzájuk tartozó nézetek egyáltalán létrejönnének. A `SingletonContainer` eltárolja a szükséges információkat, és biztosítja, hogy a későbbiekben a nézetek, a konténerre való referencia megszerzése nélkül is képesek legyenek a szükséges ViewModel-ekről referenciát szerezni.



5.3. ábra: ViewModel-ek megszerzése az ablakok létrehozásakor

A megoldásban az alkalmazás logika nem tolódik el a View-k irányába, mivel a modulok tényleges kezelése a ViewModel-ekben marad, a héj csupán a héjba betölthető nézeteket kérik el a moduloktól reflection segítésével, majd csatolják is azokat. Ez az architektúra javaslata szerint amúgy is a nézet felelősségi körébe tartozik.

A singleton osztály alkalmazása egyszerűbb felépítést biztosít az absztrakt nézet rétegnek. A feladat megvalósítható lenne ezen osztály nélkül is, pusztán az üzenetekben történő ViewModel referenciák átadásával, így azonban jóval kisebb a rétegek közötti függőség érhető el, és könnyebb a tesztelhetőség is.

A megoldáshoz használt Messaging paramétereként használt objektum egy összetett objektum. Az eseményt jelölő típussal (`EventArgs.OpenSimulationMessage`, `EventArgs.RemoveSimulationMessage`) a modulok azonosításához szükséges információkat tároló objektum (`Repository.SimulationRecordWithModuleInfo`) is átadásra kerül. Az üzenetküldés során az üzenet a `MainViewModel`-ből indul és a `MainView`-ban kerül értelmezésre, azonban a shell példányosításakor kibontásra kerül és a héj ablak alá tartozó alsóbb rétegekbe már csak a belsejében tárolt információk jutnak el.

A mellékablakok tartalommal feltöltését végző kód legérdekesebb részei terjedelme miatt a függelékben megtekinthető.

5.5 Modulok sablonszerkezete

A modulok elkészítésekor törekedtem a modulok egységes felépítésére, így azokat egy előre kitalált sablon szerint implementáltam.

A modulok mindegyike megvalósítja az `Infrastructure.Module` absztrakt osztályt. Az alkalmazás ezen az osztályon keresztül képes megállapítani, hogy a modul milyen nézeteket szeretne regisztrálni a héjban, pontosabban a mellékablak lapjain. E osztályokat mindegyikét az aktuális feladat nevével megegyező nevű osztályában alakítottam ki.

```
public class TaskScheduler : Module
{
    public TaskScheduler(): base() { }

    public override Type GetModuleViewModelType()
    {
        return (new TS_ModuleViewModel()).GetType();
    }
    public override Type GetInputViewType()
    {
        return (new TS_InputView()).GetType();
    }
    // . . .
}
```

A csatoláshoz szükséges alapinformációk megadása után az alkalmazás példányosítja a modul egyik legfontosabb elemét, a Facade mintát megvalósító absztrakt nézetet. Ezek XY_ModuleViewModel név alatt kerültek implementálásra, ahol XY a feladattípus angol nyelvű rövidítése.

```
public class TS_ModuleViewModel : Notifier, IModuleViewModelBaseFacade
{
    // . . .
}
```

Az alkalmazás ennek az osztálynak egy példányával kommunikál, így az ő felelőségi körébe tartozik az alkalmazástól érkező igények továbbküldése, ViewModel-ek példányosítása, a modell kialakítása, futtatása valamint a modell réteg nézethez csatolása stb.

Miután az `XY_ModuleViewModel` példányosításra került, az alkalmazás példányosítja az `Infrastructure.Module`-t megvalósító osztály által megjelölt nézeteket. A csatolni kívánt nézetek mindegyikétől elvárja az `IModuleViewBase` interfész megvalósítását, melynek `GetViewsDataContextAsPropertyNameOfModuleViewModel` nevű metódusán keresztül lekérdezi azokat a tulajdonságokat, melyekhez a nézetek az adatkörnyezetüket kötni szeretnék és amennyiben tudja, hozzáadja azokat.

```
public partial class TS_SimulationView : UserControl, IModuleViewBase
{
    public TS_SimulationView() { InitializeComponent(); }

    public string GetViewsDataContextAsPropertyNameOfModuleViewModel()
    {
        return "SimulatorViewModel";
    }
}
```

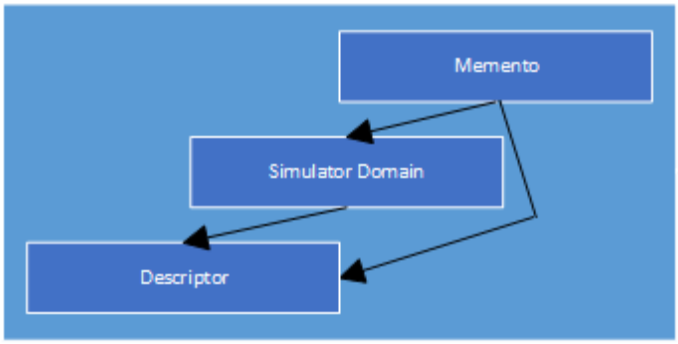
A fenti kódon látható, ahogy a taszkütemező egyik héjhoz csatolandó nézete megadja, hogy az `XY_ModuleViewModel` melyik tulajdonságához szeretne kapcsolódni.

Ezek után már nincs más hátra, mint a szimulációhoz szükséges osztályszerkezet kialakítása. A 4.10.-es ábrának megfelelő üzleti logikához kapcsolandó modell osztályokat, melyek megvalósítják az `Infrastructure` projektbeli `IConfigurator`, `ISimulator`, `IPresenter` interfészeket, rendszerint az `XY_ConfiguratorModel`, `XY_SimulatorModel` valamint az `XY_PresenterModel` osztályokban helyeztem el.

Az adatrétegbeli osztálycsoportok közül (`Descriptor`, `SimulatorDomain`, `Memento`) csak egyetlen osztályt, a bemenetleírókat megvalósító osztályt láttam el `XY_Descriptor` névvel. Ezek megvalósítják az `Infrastructure`-beli `DescriptorBase` alaposztályokat, tulajdonságaik el vannak látva a megfelelő XML attribútumokkal és minden e csoportba tartozó osztály sorosítható és rendelkeznek Serializable címkével is.

A konkrét megvalósításokban az osztályok nagy száma miatt nem törekedtem a tiszta adatszerkezet kialakítására. Így eltekintettem attól, hogy „tiszta” ViewModel osztályokat készítsek, azaz a modell osztályokhoz nem hoztam létre külön viewModel osztályokat, hanem csak felhasználtam a modelleket az absztrakt nézetekben.

Bár nem volt mindig rá lehetőség, igyekeztem ugyanezt az elvet alkalmazni az adatrétegben is. Ha egy bemenetleíró a szimuláció során is felhasználható volt, nem foglalkoztam azok áttöltésével szimulációs adatszerkezetekbe. Ugyanez igaz a `SimulatorDomain` adatszerkezetei és a `Memento` adatszerkezeteire.



5.4. ábra: Modell osztálycsoportok egymásra épülése

A fenti ábrán látható, ahogy a `SimulatorDomain` osztályai leírók szerény képességeire épülve szimulálhatóvá bővítik az alsóbb szintű osztályokat. Hasonlóan a `Memento` osztályai is visszatölthetővé terjesztik ki az alatta levő adatszerkezeteket.

5.6 Bemenetek felhasználása

Az elkészített részfeladatok mindegyike a szimuláció körülményeit a bemenő feladatleírások alapján állítja össze. A modulok az alkalmazáson keresztül jutnak hozzá a bemeneti információhoz. A feladatleírások kiválasztása és beolvasása nem az ő felelősségi körükbe tartozik, csupán azok értelmezése. A moduloknak a bemenő információk megfelelő kezelését a nélkül kell elvégezni, hogy az alkalmazás ismerné az

általuk definiált feladatleírót. Ezért szükség van egy egységesen kezelhető absztrakt bemeneti objektumra, melyet az alkalmazás képes átadni számukra.

Ezért esett a választás a .NET által tartalmazott `System.IO.Stream` osztályra. A moduloknak a stream feldolgozásával képesek eldönteni, hogy fel tudják-e belőle tölteni leíró objektumaikat. Amennyiben ez sikeres, értelmezik azokat, kiszűrik az inkonzisztenciákat és futtatható modell-t készítenek belőlük. Erre szolgáló metódus az `IConfigurator`-ban deklarált `InicializeModuleByStream` nevű metódus:

```
public SimulationStatus InicializeModuleByStream
                        (System.IO.Stream XMLInputstream){ }
```

A betöltés folyamata akár sikeres, akár nem, a modul visszajelzést küld az alkalmazás számára a `SimulationStatus` felhasználásával.

5.7 Felületi elemek

A modulok által tartalmazott nézeteket igyekeztem UserControl-okba elkülöníteni, hogy azok átláthatóak és többször felhasználhatóak legyenek. A héj régióihoz kötni kívánt nézetekkel ellentétben ezeknek a nem kell megvalósítaniuk az `Infrastructure`-ban deklarált interfészeket, elegendő csupán az ilyen nézetekben történő hivatkozásuk és adatkörnyezetük biztosítása. Az alábbi ábrán látható egy ilyen többször felhasználható felületi elem használata:

```
<UserControl x:Class="TaskScheduler.Views.TS_DiagramsView"
             . . .
             xmlns:local="clr-namespace:TaskScheduler.Views.UserControls">
<StackPanel>
<local:GanntDiagram DataContext="{Binding SimulatorModel.GanntHistory}"
                    Margin="10,0,0,3">
</local:GanntDiagram>
</StackPanel>
</UserControl>
```

Az ehhez hasonló felhasználói vezérlőkön belül az adatkötött objektumok megjelenítését végző vezérlők mellett példányosításra kerültek a hozzá tartozó adatsablonok, sablonválasztók és átalakítók is. E objektumok a felhasználói vezérlők erőforrástárában kaptak helyet.

```
<UserControl x:Class="TaskScheduler.Views.UserControls.GanntDiagram">
    <UserControl.Resources>
        <local:WidthMultiplicatorConverter
x:Key="widthMultiplicatorConverter" />

        <DataTemplate x:Key="otherGanntElementTemplate"><!-- elemek -->
        </DataTemplate>
```

```
        <DataTemplate x:Key="firstGanntElementTemplate"><!-- elemek -->
        </DataTemplate>

        <local:GanntTemplateSelector
            FirstGanntElement="{StaticResource firstGanntElement}"
            OtherGanntElement="{StaticResource otherGanntElement}"
            x:Key="ganntTemplateSelector" />
    </UserControl.Resources>
</UserControl>
```

Az megjelölt erőforrásokhoz tartozó osztályok a nézeteken belül tipikusan `UserControls`, `Converters` és `DataTemplateSelectors` nevű könyvtárakban kerültek elhelyezésre.

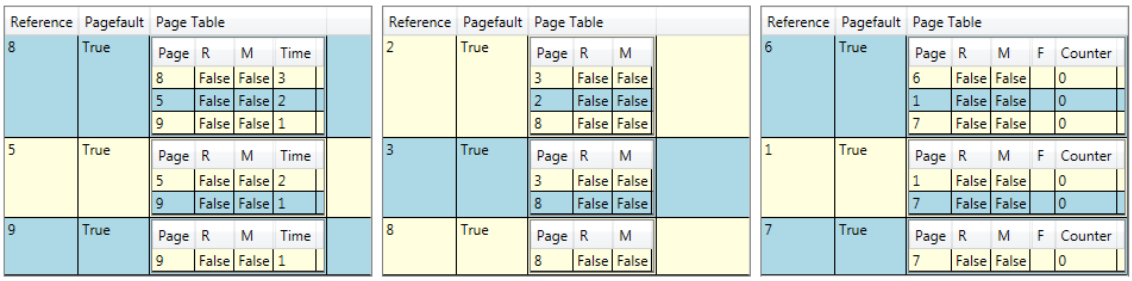
5.8 Historikus adatszerkezetek

Az alkalmazás moduljai közül kettőben is, a lapcsere algoritmusok, valamint a taszk ütemezés megvalósítása során olyan felületet kellett kialakítani, amely képes a pillanatnyi értékek megjelenítésével egy időben a korábbi adatok megjelenítésére is. E adatsablonok mögött a háttérben összetett adatszerkezetek meglétére van szükség, melyek lépésenként egy elemmel bővülnek.

Sajnálatos módon egy idő után az adatszerkezetek tárolása nagyon sok memória használatához vezethet, mivel a felhasznált memória mennyisége lépésenként lineárisan növekszik. E megvalósításokban kifejezetten kerültem a Memento minta használatát, mivel N méretű bemenet esetén a k. lépésben lépésenként $O(k * N)$ memóriaigény növekedéssel kellene számolni. Ezekben az esetekben a visszaléptetés során a szimulációk újrafuttatása történik a tárolt adatok előhívása helyett.

5.8.1 Laptábla

A laptábla ábrázolása során nem sikerült megvalósítani az egyetlen táblázatban történő összegzést, ezért egy kevésbé szép lehetőséget alkalmazása mellett döntöttem:



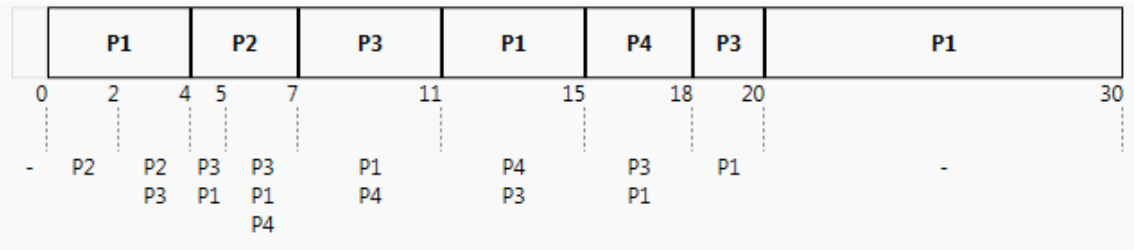
5.5. ábra: Különböző adatokat megjelenítő laptáblák alakulása az időben

A fenti képeken látható az egymásba ágyazott táblaszerkezet. Az alsó sorok jelölik a korábbi iterációkat. Egy iterációban a laptáblán a memóriában levő lapok, a hozzájuk tartozó jelzőbitek, számlálók jelennek meg a futtatott algoritmus típusának megfelelően.

5.8.2 Gannt diagram

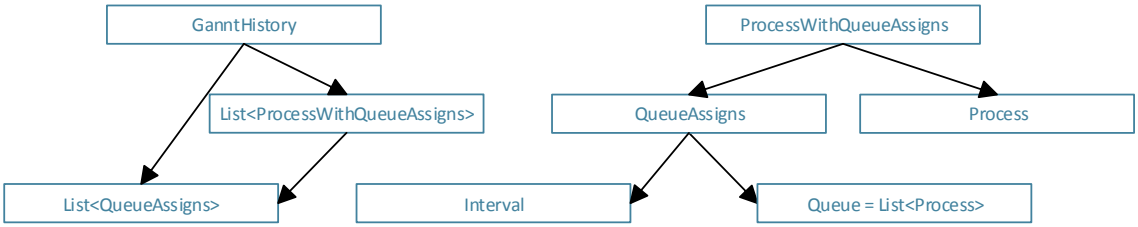
A folyamatütemező megoldásának egyik leginkább megértést segítő eleme a folyamat időbeli viszonyainak grafikus ábrázolását lehetővé tevő Gannt-diagram.

Egy ilyen ábra helyes megrajzolásához egy historikus, 3 szinten egymásba ágyazott listás adatszerkezet kezelésére van szükség. Az implementációban első szinten a folyamatok listája jelenik meg. Minden folyamathoz az ütemezési fázisok listáját kell nyilvántartani és futás időben megfelelően alakítani, adott esetben bővíteni. Az ütemezési fázisok tartalmazzák az intervallum határokat és az aktuális várakozási sor tartalmát.



5.6. ábra: Gannt-diagram megvalósítása az alkalmazásban

Mivel a WPF-ben nehezen lehet triviális megoldással 3 szinten listás adatszerkezeteket elcsúszások nélkül pontosan megjeleníteni, ezért szükség volt a három szintű listaszerkezet levelét képező ütemezési sorok tárolására közvetlenül a folyamatok alatt is és mellett is. Az így kialakuló adatszerkezet a következő struktúrában ábrázolható:



5.7. ábra: Három szintű listaszerkezetről transzformálva

A furfangosan létrehozott adatszerkezettel már lehetséges két egymásba ágyazott ItemsControl és a megfelelő DataTemplate-ek kiválasztásával egyszerre elcsúszások nélkül megjeleníteni az adatokat és hierarchikusan tárolni is azokat.

5.9 Egyéb szolgáltatások

Az elkészített modulok szinte mindegyikében szükség volt további szolgáltatások felhasználására. Ezek közül a legáltalánosabbak kerülnek kifejtésre.

Bár az alkalmazás ViewModeljei az MVVM Light Toolkit ViewModelBase alaposztályának felhasználásával a `RaisePropertyChanged` eseményen keresztül képesek a felületet értesíteni a modell állapotának megváltozásáról, a modulok ViewModel-jei erre képtelenek, tekintve, hogy megírásukkor a fenti osztálykönyvtár szándékosan nem került felhasználásra. Ezért szükséges volt számukra biztosítani egy ugyanilyen szolgáltatásokat nyújtó felületet, ez volt az `Infrastructure` projektben megvalósításra kerülő `Notifier` osztály.

```
[Serializable]
public abstract class Notifier : INotifyPropertyChanged
{
    [field: NonSerialized]
    public event PropertyChangedEventHandler PropertyChanged;

    protected virtual void OnPropertyChanged(string propertyName)
    {
        if (PropertyChanged != null)
        {
            PropertyChanged(this,
                            new PropertyChangedEventArgs(propertyName));
        }
    }
}
```

A modulok ViewModel-jei a `Notifier` osztálból történő örökléssel képessé váltak az alkalmazás felületének értesítésére a modell megváltozása esetén.

A historikus adatszerkezetek feltöltésekor gyakran volt szükség az egy iterációval korábbi adatokra történő építkezésre. A lépésenkénti transzformációhoz szükséges ezeknek az adatoknak egy másolatával rendelkezni. Referenciákat nem másoló megoldásként többféle megoldás is választható. Az újbóli felhasználhatóság miatt eltekintettem az `ICloneable` interfész megvalósításától és egy lassabb, de egyszerű megoldás mellett döntöttem. Mély másolatok készítésére általában a következő osztályt használtam:

```
class MyCloner
{
    public static T DeepClone<T>(T obj)
    {
        using (var ms = new MemoryStream())
        {
            var formatter = new BinaryFormatter();
            formatter.Serialize(ms, obj);
            ms.Position = 0;
            return (T)formatter.Deserialize(ms);
        }
    }
}
```

A fenti megoldás valamivel lassabb a `Clone()` metódus implementációjánál, itt azonban fontosabb szempont volt, hogy ne kelljen minden apró részletet megvalósító osztályban külön implementálni az `ICloneable` interfészt, ehelyett egy könnyen kódolható megoldás legyen alkalmazható.

6 Eredmények kiértékelése

Az alábbiakban összefoglalásra kerülnek azok a szempontok, amelyek szerint az elkészített alkalmazás minősítése történik.

A felület egyértelműségének és kezelhetőségének összefoglalását az informatikában jártas, az operációs rendszerek területével már megismertetett, bár szakértőnek nem mondható hallgatók véleménye alapján állítottam össze. Az operációs rendszerek tárgyhoz tartozó jegyzetek példái alapján elkészített bemenetek szerint megtörténik az algoritmusok futtatása és az általuk produkált kimenetek kiértékelése. Miután az itt tapasztalt esetleges eltérések okai is kifejtésre kerültek, javaslatot teszek az alkalmazás jövőbeli továbbfejlesztésének lehetséges megvalósításaira.

6.1 Felhasználói vélemények

Az elkészített alkalmazás véleményezéséhez a futtatható állományokat, valamint a betölthető bemeneteket küldtem el a felkért személyeknek. Mivel kíváncsi voltam arra, mennyire intuitíven használható az elkészített felhasználói felület, ezért technikai leírást nem mellékeltem az alkalmazáshoz. Összesen három személyt kértem fel az alkalmazás véleményezésére, közülük ketten a BME korábbi, vagy jelenlegi hallgatói, egy személy pedig egy másik magyarországi egyetem volt hallgatója. Tanulmányaik során mindannyian találkoztak azokkal a problémakörökkel, melyet az alkalmazás demonstrálni hivatott.

Leginkább azokra a kérdésekre kerestem a választ, hogy a felhasználó mennyire találta hasznosnak az alkalmazást, az jól működött-e, a szimulációk eredményeit helyesnek vélte-e, mennyire találta intuitíven használhatónak és ergonomikusnak a felületet, talált-e hibákat benne, mi a véleménye a működéssel kapcsolatban és mit lenne érdemes javítani rajta. Az alábbiakban a vélemények összesítése került összefoglalásra.

„Tetszik, hogy van tálca ikonja az alkalmazásnak. A kezdő felhasználó a Help menüt próbálja ki. File menüben az Exit megfelelően működik. Load input file ablakban látszik, hogy csak XML dokumentumokat képes a program beolvasni. Felhasználói élményt növelné, ha a program helyénél nyílna meg, nem a futtató felhasználó Dokumentum mappájában, de emlékszik az utolsó hozzáadás helyére.”

„Hasznos funkció lenne, ha a szimulációkat újra alapállapotra lehetne tenni és nem úgy működne, hogy Remove from list, majd Add to list. Ellenben hasznos funkció, hogy lehet a szimulációt automatikusan futtatni, valamint léptetni." – Valóban hasznos lenne egy Újratöltés gomb. Erre igény az alkalmazás fejlesztésekor nem mutatkozott.

„A szimulátor ablakaiban jól működik a Zoom slider. Ez egy igen megfontolt tervezői döntés, amely a különböző felbontású monitorokon teszi lehetővé a felhasználónak a jó olvashatóságot." – E funkció egy példaalkalmazás tanulmányozása során felmerülő ötleten alapul, a tervezési fázisban

„Ha a 'Pause' gombra kattintunk, akkor a felirat átváltozhatna, hogy 'Run', vagy 'Play'." – Véleményem szerint egy ilyen célalkalmazásnál e funkció megvalósítása csak csekély előnyökkel járna.

„Ha egy simulációt már futtatok, és automatikus tesztre kattintok, és menet közben adok hozzá egy új szimulációt, akkor a régi leáll, és az új kezd futni. Én azt várnám, hogy a régi fusson tovább és az új álljon meg." – Érdekes észrevétel, ha több felhasználót is összezavar ez a működés, valóban érdemes lehet módosítani rajta.

„Több bemenet betöltődése során néha az alábbi figyelmeztetésre lettem figyelmes: This simulation with the same Title and Type is already loaded. Bővebb információt lehetett volna adni arról, hogy melyik fájl betöltődése okozta a hibát." – A fejlesztés során e paraméter értéke nyilvánvaló volt, így ezzel az igénnyel egyetértek.

„Általánosságban elmondható, hogy olyan helyeken, ahol sok szimuláció lefuttatására van szükség akár többmonitoros munkaállomásokon, egy hasznos alkalmazás született. Oktatási célokra is kiválóan alkalmas."

Az alkalmazást véleményező személyek közül akadt olyan, aki rendelkezett a futtatáshoz szükséges .NET 4.5-ös keretrendszerrel, ám mégsem tudta elindítani az alkalmazást. Mivel ez a két másik véleményezőnek és jómagamnak is sikerült, ezért feltehetőleg a hibát valamilyen helytelen környezeti beállítás okozhatta, mindenesetre érdemes lenne kivizsgálni a problémát.

A másik két véleményező személy kipróbálta az elkészített mintafeladatokat és teszteseteket, végig lépegetett a szimulációkon valamint áttekintette az lépések eredményeit. Bár tevékenységükkel sikerült objektív visszajelzéseket kapni, a véleményezés mindenképpen hasznosnak mondható, de eredmények nem tekinthetők teljes körű Usability tesztnek.

6.2 Tesztelés

A teszteléshez olyan feladatleírásokat vettem alapul, melyek a tárgy oktatásában használt segédanyagban megtalálhatóak. Ennek oka, hogy a felhasznált anyagok az évek során letisztultak, így vélhetőleg minimális számú hibát tartalmaz. Kiértékelésükkel a modulokban implementált algoritmusok minőségéről viszonylag pontos kép kapható.

A modulokhoz tartozó teszteket a mellékleten az `inputs/tests` könyvtáron belül helyeztem el. Mivel a segédanyagokban a virtuális memória címleképezésnek csak az elvi háttere került bemutatásra, így konkrét ellenőrzött példák híján ehhez tesztbemeneteket saját számításaim alapján hoztam létre. A másik három részfeladat algoritmusaival a diasorban [2][4] szereplő példákat is leteszteltem, melyekben a produkált kimenetek szerint apróbb elírásokat fedeztem fel.

A tesztelés legfőbb célja az algoritmusok és metrikák által előállított eredmények ellenőrzése és a helyes működésről való megbizonyosodás volt. A tesztelés nem terjedt ki a felületi logika teljes körű ellenőrzésére. Az elkészített nézetek, diagramok is csak érintőlegesen, a modellek által létrehozott adatszerkezetek helyes megjelenítésén keresztül kerültek ellenőrzésre.

A tesztelés menete a korábbról ismert mintapéldák tesztbementként történő felhasználásával és lefuttatásával, majd az előállított eredmények kézi kiértékelésével zajlott. Az eredmények könnyű összehasonlításnak tekinthetőek.

6.2.1 Feladat ütemezés

A taszkütemező algoritmusok tesztjeinek eredményeit az alábbi táblázat tartalmazza:

Algoritmus	**Tesztsorozat**		**Eredmény**
FCFS	OFS-1211	PASS	Elvárt eredményeknek tökéletesen megfelelt.
	OFS-1221	PASS	Elvárt eredményeknek tökéletesen megfelelt.
	OFS-1231	PASS	Elvárt eredményeknek tökéletesen megfelelt.
SJF	OFS-1212	PASS	Elvárt eredményeknek tökéletesen megfelelt.
	OFS-1222	PASS	Elvárt eredményeknek tökéletesen megfelelt.
	OFS-1232	PASS	Elvárt eredményeknek tökéletesen megfelelt.

SRTF	OFS-1213	PASS	Elvárt eredményeknek tökéletesen megfelelt.
	OFS-1223	PASS	Elvárt eredményeknek tökéletesen megfelelt.
	OFS-1233	WARN	Eltérés mutatkozott egy metrikában. Minden más eredmény az elvárt eredményeknek tökéletesen megfelelt.
RR	OFS-1214	WARN	Eltérés mutatkozott egy metrikában. Minden más eredmény az elvárt eredményeknek tökéletesen megfelelt.
	OFS-1224	WARN	Eltérés mutatkozott egy metrikában. Minden más eredmény az elvárt eredményeknek tökéletesen megfelelt.
	OFS-1234	WARN	Eltérés mutatkozott egy metrikában. Minden más eredmény az elvárt eredményeknek tökéletesen megfelelt.

Az OFS-1233 teszt kielemzése során kiderült, hogy a CPU kihasználtságnál eltérés mutatkozott a segédanyagban feltüntetett értékhez képest. Ennek oka, hogy az algoritmus az [1] segédlet alapján került megvalósításra, mely szerint az SRTF algoritmusnál ütemező akkor is lefut, ha egy új taszk válik Futásra kész-é. Az algoritmus ennek megfelelően a CPU kihasználtságot csökkentette a megfelelő adminisztrációs idővel, holott ez a teszt megalkotásához használt segédletben nem történt meg.

Az OFS-1214, OFS-1224, OFS-1234 tesztek lefutásakor apró eltérés volt tapasztalható a CPU kihasználtság mérésénél a segédanyagban feltüntetett értékhez képest. Az elemzése során kiderült, hogy az alkalmazásban az ütemezési és kontextus váltási idők a tesztsorozattól eltérően kerültek értelmezésre. A kontextus váltási idő a feladatmegoldás során alkalmazott egyszerűsítés miatt kétszerese az ütemezési időnek, így foglalja magában azt az időtartamot is. Ütemezési idővel az alkalmazás csak akkor számol, ha kontextus váltás nem történt, csak az időkeret miatt került sor az ütemező futására. Az eltérés további elemzést igényel.

Annak oka, hogy a két idő egy egyszerű adminisztrációs idővel lett közelítve abban keresendő, hogy az alkalmazás modelljének kialakítását korábbi diasor és

tesztfeladatok felhasználásával készítettem, amelyeknél még nem mutatkozott igény a két idő megkülönböztetésére. Erre az igényre csak a tesztelés fázisában derült fény.

Az alkalmazás a feladat értelmezéséből fakadóan nem számol a folyamatok átlagos válaszidejével, holott ez a tesztfeladatokban mégis rendelkezésre áll. A tesztfeladatokkal ellentétben az alkalmazás lépésről-lépésre számítja az áteresztőképesség metrikát.

6.2.2 Memóriafoglalás

A memóriafoglaló algoritmusok tesztjeinek eredményeit az alábbi táblázat tartalmazza:

Algoritmus	**Tesztsorozat**		**Eredmény**
FF	OSF-313FF	PASS	Elvárt eredményeknek tökéletesen megfelelt.
	OSF-315FF	WARN	Elvárt eredményeknek részlegesen megfelelt.
NF	OSF-313NF	PASS	Elvárt eredményeknek tökéletesen megfelelt.
	OSF-315NF	WARN	Elvárt eredményeknek részlegesen megfelelt.
BF	OSF-313BF	PASS	Elvárt eredményeknek tökéletesen megfelelt.
	OSF-315BF	WARN	Elvárt eredményeknek részlegesen megfelelt.
WF	OSF-313WF	PASS	Elvárt eredményeknek tökéletesen megfelelt.
	OSF-315WF	WARN	Elvárt eredményeknek részlegesen megfelelt.

Az OSF-315FF, OSF-315NF , OSF-315BF, OSF-315WF tesztek lefuttatásakor kiderült, hogy a feladat modelljének megtervezésekor – hasonlóan a folyamatok ütemezéséhez – itt sem állt még rendelkezésre olyan igény, mely a feladatok számításának nehézségét ilyen irányból növelné. A tesztfeladatok megkövetelik, hogy a memóriafoglalás csak 4 kbyte-os blokkokban történhessen. Mivel az alkalmazás az operációs rendszerhez beérkező igények kielégítésével számol, ezért a feladat bemenete manuálisan transzformálható a tesztsorozat által igényelt formába. Az így létrehozott tesztsorozat, valamint a blokkos kritérium elhagyásával létrehozott tesztsorozat az elvárásoknak megfelelően működött.

6.2.3 Lapcsere algoritmusok

A lapcsere algoritmusok tesztjeinek eredményeit az alábbi táblázat tartalmazza:

Algoritmus	Tesztsorozat		Eredmény
OPT	OFS-3241	PASS	Elvárt eredményeknek megfelelt.
FIFO	OFS-3242F3	PASS	Elvárt eredményeknek tökéletesen megfelelt.
	OFS-3242F4	PASS	Elvárt eredményeknek tökéletesen megfelelt.
SC	OFS-3243	WARN	Elvárt eredményeknek közelítőleg megfelelt.
LRU	OFS-3244	WARN	Elvárt eredményeknek részlegesen megfelelt.
LFU	OFS-3245	FAIL	Elvárt eredményeknek nem felelt meg.
NRU	OFS-3246	WARN	Elvárt eredményeknek közelítőleg megfelelt.

Az OFS-3241 teszt esetében a laptábla tartalma a végrehajtás utolsó fázisában nem volt azonos a feladatban levővel. Ezt azonban nem algoritmikus hiba okozta, hanem az adott időpillanattól két lap is olyan állapotban volt, hogy később már egyikükre sem volt szükség, így bármelyik választható lett volna.

Az OFS-3243 teszt esetében a feladat definiálásnak megfelelően az R bitek nem kerültek automatikusan beállításra. Ennek beállítását egy esemény formájában, a lapkérés megadása után kell külön bemenetként megadni. Ezt leszámítva minden eredmény tökéletesen megfelelt.

Hasonlóan az előzőhöz, a bitek a feladat megvalósítása miatt az OFS-3246 esetén sem kerültek automatikusan beállításra. Ezeket bemeneti eseményként felvéve, és a táblázat megfelelő sorait összeolvasva a helyes eredményt kaptam.

Az OFS-3244 teszt esetében az algoritmus az elvárásoknak megfelelően teljesített. Különbség csak a megvalósítás módjában volt, mivel az utolsó laphasználat ideje a példában a 0-3 számokkal volt jelölve, míg az implementációban egy időbélyeggel, amely az utolsó használat idejét jelezte.

Az OFS-3245 tesztnél a várt eredménytől való eltérést az okozta, hogy a feladat modelljének megalkotásakor nem vettem figyelembe a lapok tárba fagyasztásának lehetőségét különböző időkre. Ennek oka az volt, hogy a modellezésnél a diasorban levő példákat vettem alapul, nem pedig a teszteléshez később megkapott példasort. Az

algoritmus csak a lehető legegyszerűbb módon lett megvalósítva, a tárba fagyasztás lehetősége nélkül.

6.2.4 Virtuális memória címleképezés

Mivel a tárgy jegyzetei és példái között nem találtam a működés ellenőrzésére alkalmas előre számolt példákat, ezért e feladat tesztelése az előzőektől eltérőek zajlott.

Először meghatároztam a lap-keret leképezések sorozatát, majd találomra megválasztottam a virtuális címeket, melyekhez manuálisan kiszámoltam a megfelelő értékeket. Ezt követően a fentieknek megfelelő bemeneteket gyártottam és azok segítségével leteszteltem az adatokat. Minden teszteset az manuális számításoknak megfelelt.

6.3 Értékelés

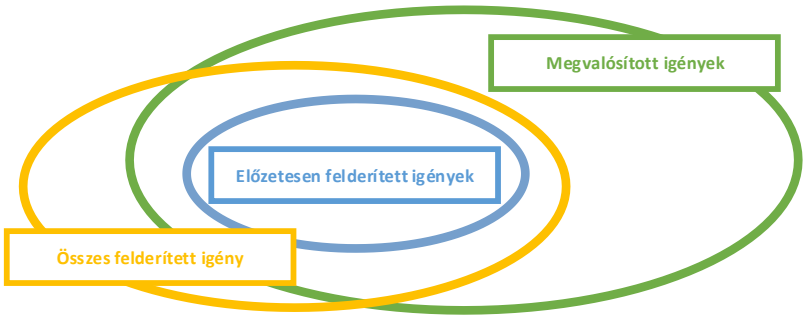
Ha összegezzük az eddigieket és megvizsgáljuk azokat a szempontokat, melyek alapján az alkalmazás megvalósításra került látható, hogy az előzetesen meghatározott követelményeket sikerült legalább részben, de legtöbbször egészben is teljesíteni.

Az alkalmazás rendelkezik könnyen kezelhető, egyszerű, átlátható felhasználói felülettel, képes egyszerre több nézetben ugyanannak a történésnek a megjelenítésére és egymással összefüggő problémaköröket egymástól elválasztva történő szemléltetésére. Az algoritmusok futásának idejében a megfelelő metrikák automatikusan kiszámításra kerülnek, továbbá a megértés diagramokkal, könnyen érthető táblázatokkal, más grafikus eszközökkel van támogatva.

A feladatok futtatása egy média lejátszó program kezelőfelületének megfelelő bonyolultsággal rendelkezik, miközben az egyszerű léptetés mellett az automatikus szimuláció is rendelkezésre áll. Architektúrális szempontból lazán csatoltsága miatt felépítése könnyen módosítható, továbbá az modulokba történő szervezés miatt további feladattípusok is könnyen megvalósíthatóak a megfelelő interfészek megvalósításával.

Az előző alfejezetben kiértékelt tesztek alapján elmondható, hogy az elkészített megoldás bár teljesíti az irányába támasztott legfontosabb elvárásokat, mégsem tökéletes mértékben illeszkedik a tesztfeladatok eredményeire. Ennek oka leginkább abban keresendő, hogy a témakörökhöz tartozó modellek megkonstruálásakor még nem minden lehetséges tesztfeladat állt rendelkezésre. Így fordulhatott elő, hogy a kidolgozás során a feladatok ütemezésénél nem kezeltem elkülönítve a különböző

adminisztrációs időket (mint pl. kontextus váltási idő, ütemezési idő stb.), vagy a laptábla algoritmusok implementálásakor sem fordítottam elegendő figyelmet a lapok tárba fagyasztásának lehetőségére, mivel a használt forrásanyag is csak említés szintjén kezelte. Ezek a hiányosságok egy következő fejlesztési ciklusban kerülnek kiküszöbölésre.



6.1. ábra: Az alkalmazás képességei az igényekhez viszonyítva

A fenti ábrán látható, hogy az alkalmazás megvalósítása véleményem szerint hogyan illeszkedik a fejlesztés előtt, majd a fejlesztés után felmerülő igényekre. A fejlesztés előtt felmerülő igényeket és az előre látható szempontokat véleményem szerint az alkalmazásban igen jól sikerült lefedni. Kisebb gondokat okoztak azonban a minősítéshez használt kritériumok, mivel azokat csak a tervezési és fejlesztési ciklus után tudtam figyelembe venni. Minőségi kritériumoknak jobban megfelelő alkalmazást lehetett volna készíteni, ha ezek az apró eltérések a fejlesztés korábbi fázisaiban megmutatkoznak.

7 Összefoglalás

Az előzőekben egy olyan alkalmazás létrehozásának folyamata került bemutatásra, amely illeszkedik az Operációs rendszerek c. tárgy tematikájához és teljesíti a demonstrációs eszközökkel támasztott legfontosabb követelményeket.

Elsőként körvonalakban összefoglaltam a már létező megoldásokat. Miután már több alkalmazás is született, melyek megkísérelték bemutatni az előzetesen meghatározott tárgyterületek problémaköreit, megkíséreltem ezek előnyös és hátrányos tulajdonságait alapul venni egy új alkalmazás létrehozásához. A továbbiakban az itt összesített kedvező tulajdonságok elérésére fókuszáltam.

Miután a legfontosabb elvek összeírásra kerültek, a meghatározott tárgyterület problémakörei kerültek feltárásra. Összefoglalásra került, hogy mely problémákat miért érdemes bemutatni.

Ezt követően a tárgyterületek kritériumai kerültek összefoglalásra. Összesítettem azokat az algoritmusokat, metrikákat és diagramokat, melyeket az alkalmazásnak érdemes tartalmaznia, mivel hasonló alkalmazások többsége is tartalmazza.

Az igények és elvárások pontosítása után az alkalmazás megtervezésének részleteivel foglalkoztam. Felvázoltam a megvalósítandó alkalmazás szerkezetét, működésének alapelveit, valamint kifejtésre került az is, hogy egyes tervezési minták miképpen használhatók fel a megvalósítás során.

Rövid kitérőként körvonalakban bemutatásra került az alkalmazott WPF technológia néhány apró részlete. Kitértem az alkalmazott MVVM architektúra néhány alapelvére, valamint áttekintettem, hogy az különböző osztálykönyvtárak milyen segítséget tudnak nyújtani a célok elérésében.

Ezt követően kifejtésre kerültek a megvalósítás technikai részletei. Áttekinthettünk néhány működési elvet és kódrészletet, melyek az alkalmazás működése szempontjából kulcsfontossággal bírnak.

Végül nem maradhatott el az elkészített alkalmazás értékelése sem, melyhez a saját véleményemen kívül külső forrásból származó, de az informatika területén jártas egyének, valamint hallgatók véleményeket használtam fel. A tárgy segédanyagai

alapján létrehoztam a megfelelő teszteseteket, melyeket felhasználtam az algoritmusok működésének tesztelésére.

A következőkben a kapott vélemények és eredmények összesítése alapján megkíséreltem objektív tanácsokat adni arra, hogy a későbbiek folyamán miképpen lehetne tökéletesíteni a demonstrációs alkalmazást.

7.1 Továbbfejlesztési lehetőségek

Mivel az alkalmazásban mutatkozik egy-két kisebb eltérés a tesztek elvárt eredményeihez képest, melynek pontos okait az előzőekben sikerült részletesen feltárni, ezért a továbbfejlesztés első lépésében ezen apró hiányosságok kiküszöbölésére fordítanék figyelmet.

Számomra az alkalmazásban mégsem ezek az apróságok a leginkább szemet szúróak. Egy következő iterációban a bemenő feladatok létrehozását és kimentését tenném lehetővé a felhasználói felületről, így téve az alkalmazást még´inkább felhasználóbaráttá. A leírások manuális szerkesztése a korábban megvizsgált programoknál eléggé kedvelt módszer, azonban egy grafikus felhasználói felülettel rendelkező programnál ezt nem tartom járható útnak.

Az Operációs rendszerek c. tárgyhoz kapcsolódóan mindenképpen megfontolnám egyéb algoritmusok implementálását is. Többek között demonstrálásra érdemesnek tartom a holtpontdetektálás, erőforrás elosztás problémaköreit is.

Megfontolható az egyes feladattípusok közötti együttműködés megvalósítása. Ez esetben a jelenlegi architektúra kibővítésére lenne szükség, mivel a jelenlegi megoldás ezt nem képes támogatni.

A feladat modellek bővítését vagy továbbfejlesztését érintő észrevételek terjedelmük miatt a függelékben kerültek elhelyezésre.

Irodalomjegyzék

[1] Benyó Balázs, Fék Márk, Kiss István, Kóczy Annamária, Kondorosi Károly, Mészáros Tamás, Román Gyula, Szeberényi Imre, Sziray József: *Operációs rendszerek mérnöki megközelítésben*, Panem, 2000., http://www.tankonyvtar.hu/hu/tartalom/tkt/operacios-rendszerek/adatok.html

[2] Operációs rendszerek tantárgy előadásai, http://www.mit.bme.hu/oktatas/targyak/vimia219

[3] Darvas Dániel, Horányi Gergő, Jámbor Attila, Micskei Zoltán, Szabó Tamás: *Operációs rendszerek feladatai, Számolási példák és algoritmusok*, 2012 http://mit.bme.hu/~micskeiz/opre/files/opre-feladatok-segedanyag.pdf

[4] Operációs rendszerek tárgyhoz tartozó diasorok , 2010 https://wiki.sch.bme.hu/Oper%C3%A1ci%C3%B3s_rendszerek

[5] Ray Ontko, Alexander Reeder: MOSS Scheduling Simulator, MOSS Memory Management Simulator, http://www.ontko.com/moss/

[6] Visual OS, http://freshmeat.net/projects/visualos/

[7] Steve Robbins: Process Scheduling Simulator, Address Translation Simulator http://vip.cs.utsa.edu/simulators/

[8] MOSS: A Mini Operating-System Simulator: http://www.cs.kent.ac.uk/people/staff/frmb/moss/

[9] SOsim: Simulator for Operating Systems Education, 2003 http://www.training.com.br/sosim/indexen.htm

[10] SOS (Simple Operating System) Simulator: http://www.cs.unm.edu/~crowley/osbook/sos.html

[11] Microsoft patterns & practices: *Prism 4.1*, http://msdn.microsoft.com/en-us/library/ff648465.aspx

[12] MSDN, *Managed Extensibility Framework (MEF)*, http://msdn.microsoft.com/en-us/library/dd460648.aspx

[13] Christopher Bennage, Rob Eisenberg: *Tanuljuk meg a WPF használatát 24 óra alatt*,Kiskapu Kiadó, 2009

[14] Reiter István: *C# jegyzet*, LSI OMAK Alapítvány, 2000

[15] Árvai Zoltán, Csala Péter, Fár Attila Gergő, Kopacz Botond, Reiter István, Tóth László: *Silverlight 4 A technológia, és ami mögötte van — fejlesztőknek, HTML 5 ismertetővel bővített kiadás,2010*, https://devportal.hu/

[16] MVVM Light Toolkit segédanyagok és dokumentáció, http://mvvmlight.codeplex.com/, http://www.galasoft.ch/mvvm/

[17] MSDN, *WPF Apps With The Model-View-ViewModel Design Pattern*, http://msdn.microsoft.com/en-us/magazine/dd419663.aspx

Függelék

A következőkben találhatók azok az információk, melyek terjedelmi okok miatt nem kerültek be a dolgozatba, azonban az alkalmazás használatához és továbbfejlesztéséhez szükségesek lehetnek.

A függelék nagyobbik része szintén terjedelmi okok miatt a mellékleten elektronikus formában került elhelyezésre.

A melléklet tartalma

Az alábbiakban látható a szakdolgozat mellékletének könyvtárszerkezete:

- **`dist`** – tartalmazza a futtatható állományokat
- **`docs`** – tartalmazza a Szakdolgozat elektronikus változatát, valamint a Függelék nagyobbik részét
- **`inputs`**
    - **`examples`** – feladattípusonként rendezve tartalmazza a szimulációra kiválasztott példákat
        - **`MemoryAllocator`**
        - **`PageReplacer`**
        - **`TaskScheduler`**
        - **`VirtualAddressMapper`**
    - **`tests`** – feladattípusonként rendezve tartalmazza a teszteléshez használt példákat. Ebben a könyvtárban kaptak helyet a teszteléshez szükséges segédanyagok, valamint a tesztesetek neveinek kódolását feloldó rövidítések összefoglalása is.
        - **`MemoryAllocator`**
        - **`PageReplacer`**
        - **`TaskScheduler`**
        - **`VirtualAddressMapper`**

- **`validation`** – tartalmazza az új bemenetek előállítása során azok érvényesítését végző XSD leírásokat

- **`src`** – tartalmazza az elkészült forrásfájlokat

A mellékablakok tartalommal feltöltését végző kód legérdekesebb részei:

A héjat tartalmazó mellékablak létrehozása:

```
public ShellWindow(OpenSimulationMessage openSimulationMessage)
{
     InitializeComponent();
 // ...

     // View típusok lekérése
     Type inputViewType      = moduleInstance.GetInputViewType();

     // ModuleView-k beállítása a megfelelő nevesített területekhez és
DataContext-jük ellátása ModuleViewModel-el vagy azon belül egy a
ModuleViewModel által kezelt ViewModel-lel
     setShellWindowAreas(shellViewModel.ModuleViewModel,    inputViewType,
"InputInformationView",
 // ...
}
```

A mellék ablak területeinek feltöltése:

```
public  void  setShellWindowAreas(IModuleViewModelBaseFacade  moduleViewModel,
Type viewType, string contentControlName, string tabControlName)
{
if (viewType != null)
{
// nézet csatolása
var moduleView = Activator.CreateInstance(viewType);

typeof(ShellWindow).GetMethod("AppendModuleViewToContentControl").MakeGenericM
ethod(viewType).Invoke(this, new[] { contentControlName, moduleView });

// ha nem elég a ModuleViewModel DataContext-je és egy azon belül elhelyezett
Property-t szeretne használni DataContext-ként
IModuleViewBase view  = (IModuleViewBase)moduleView;

// a view tól el kell kérni, hogy van-e a ModuleViewModel-en belül property,
amelyet DataContext-ként szeretne használni
string             viewsDataContextPropertyNameInModuleViewModel             =
view.GetViewsDataContextAsPropertyNameOfModuleViewModel();

// ha van olyan Property, melyhez lehet csatolni és ha van ModuleViewModel
objektum is (különben nem lenne objektum, melynek property-jeihez referenciát
kelle állítani)
if (viewsDataContextPropertyNameInModuleViewModel != null
    && (! viewsDataContextPropertyNameInModuleViewModel.Equals(""))
    && moduleViewModel != null)
{
```

```
System.Reflection.PropertyInfo[] properties =
moduleViewModel.GetType().GetProperties();
foreach (System.Reflection.PropertyInfo property in properties)
{
if (property.Name.Equals (viewsDataContextPropertyNameInModuleViewModel))
{
object dataContext = property.GetValue(moduleViewModel, null);

((ContentControl)FindName(contentControlName)).DataContext = dataContext;
break;
}
}
}
// ha nincs akkor a ModulViewModel-t veszi alapértelmezett DataContext-nek
else
{
((ContentControl)FindName(contentControlName)).DataContext = moduleViewModel;
}
// ...
}
```

Felhasználói vezérlők csatolása:

```
public void AppendModuleViewToContentControl<T>
(string contentControlXName, T view)
where T: UserControl
{
((ContentControl)FindName(contentControlXName)).Content = view;
}
```

A mellékleten elhelyezett további információk

Terjedelmi okok miatt a függelék további részei a mellékleten szereplő `docs` könyvtárban kerültek elhelyezésre. Ezek a következő témaköröket érintik:

- felhasznált algoritmusok jellemzői
- a számított metrikák matematikai leírásai
- bemenetek előállításához szükséges segéd információk
- feladatkörök továbbfejlesztésének javaslatai