

1. Tétel: Neumann és Harvard számítógép-architektúrák összehasonlító elemzése

Számítógép architektúra fogalma: A hardver egy általános absztrakciója: a hardver struktúráját és viselkedését jelenti más rendszerek egyedi, sajátos tulajdonságaitól eltekintve.

Neumann architektúra: Számítógépes rendszer modell:

- CPU, CU, ALU
- Az utasítások és adatok részére egyetlen különálló tárolóelem szolgál.
- Univerzális Turing gépet implementál
- Szekvenciális architektúra (SISD)

A Neumann architektúra De Facto szabványa: „**Single-memory architecture**”. Ez azt jelenti, hogy az adatok és utasítások címei a memória (tároló) ugyanazon címtartományára vannak leképezve (mapping). Ilyen típusú gépek például:

- EDVAC: egyenletmegoldó, tárolt programú gép, ENIAC, UNIVAC (Pennsylvania-i egyetem): numerikus integrátor, kalkulátor. A mai rendszerek modern mini-, mikro, és mainframe számítógépei is ezt az architektúrát követik.

Neumann elvek:

- A számítógép működését tárolt program végzi (Turing)
- A vezérlést vezérlés-folyam (control-flow graph-CFG) segítségével lehet leírni. Fontos lépés itt az adatút megtervezése.
- A gép belső tárolójában a program utasításai és a végrehajtásukhoz szükséges adatok egyaránt megtalálhatók (közös utasítás és adattárolás, a program felülírhatja önmagát-Neumann architektúra definíciója).
- Az aritmetikai és logikai műveletek (programutasítások) végrehajtását önálló részegység, az ALU (Arithmetical Logical Unit) végzi.
- Az adatok és a programok beolvasására és az eredmények megjelenítésére önálló egységek (IO perifériák) szolgálnak.
- 2-es (bináris) számrendszer alkalmazása.

A fix vs. tárolt programozhatóság:

- A korai számítási eszközök fix programmal rendelkeztek (nem tárolt programozható), például: kalkulátorok.
 - A programot csak „átvezetékezéssel”, a struktúra újratervezésével lehet változtatni
 - Az újraprogramozás menete: folyamat diagram készítése-> előterv specifikáció (papíron)
->részletes mérnöki tervek->nehézkés implementáció
- Tárolt programozhatóság ötlete
 - Utasítás-készlet architektúra (ISA): RISC, CISC
 - Változtatható program: utasítások sorozata
 - Nagyfokú flexibilitás: az adatot hasonló módon lehet tárolni és kezelni. (assembler, compiler, automata programozási eszközök)

A Neumann architektúra hátrányai

- „Önmagát változtató” programok (self-modifying code):
 - Már eleve hibásan megírt program kárt okozhat önmagában illetve más programokban is.
 - OS szinten rendszer leállítás
 - Példák: Buffer túlcsordulás: kezelése hozzáféréssel, memória védelemmel.
- „Neumann bottleneck”: A sávzsélesség korlátozott a CPU és a memória között, ezért bevezetésre került a cache memória. Ez a probléma a nagymennyiségű adatok továbbítása során lépett fel.
- A nem cache alapú Neumann rendszerekben egyszerre vagy csak egy adat írás/olvasást, vagy csak az utasítás beolvasását lehet elvégezni (mivel csak egy buszrendszer van).

Harvard architektúra:

Olyan számítógéprendszer, amelynél a programutasításokat és az adatokat fizikailag különálló memóriában tárolják és külön buszon érhetők el.

Eredete: Harvard MARK I. (relés alapú rendszer)

További példák:

- Intel Pentium processzor család L1-szintű különálló adat- és utasítás cache memóriája
- Ti320 DSP jelfeldolgozó processzorok (ROM, RAM)
- Beágyazott (embedded) rendszerek: MicroBlaze, PowerPC buszrendszerei, memóriái
- Mikrovezérlők (MCU) különálló adat-utasítás buszai és memóriái.

A Harvard architektúra tulajdonságai:

- Nem szükséges a memória osztott (shared) jellegének kialakítása:
 - Szóhosszúság, időzítés, tervezési technológia, memória címezés kialakítása is különböző lehet.
 - Az utasítás (program) memória gyakran szélesebb mint az adat memória (mivel több utasítás memóriára lehet szükség).
 - Az utasításokat a legtöbb rendszer esetén ROM-ban tárolják, míg az adatot írható/olvasható memóriában (pl. RAM-ban).
 - A számítógép különálló buszrendszere segítségével egyidőben akár egy utasítás beolvasását és adat írását/olvasását is el lehet végezni (cache nélkül is).

Módosított Harvard architektúra: A modern számítógép rendszerekben az utasítás memória és a CPU között olyan közvetlen adatút biztosított, amellyel az utasítás-szót is olvasható adatként lehet elérni.

- Konstans adat (pl. string, inicializáló érték) utasítás memóriába töltésével a változók számára további helyet spórolunk meg az adatmemóriában.
- A mai modern rendszereknél a Harvard architektúra megnevezés alatt ezt a módosított változatot értjük.
- Gépi (alacsony) szintű assembly utasítások.

Harvard architektúra hátrányai:

- Az olyan egychipes rendszereknél (pl. SoC, System On a Chip), ahol egyetlen chipen van implementálva minden funkció, nehézkes lehet a különböző memória technológiák használata az utasítások és adatok kezelésénél. Ezekben az esetekben a Neumann architektúra alkalmazása megfelelőbb.
- A magas szintű nyelveket (pl. ANSI C szabvány) sem közvetlenül támogatja (a nyelvi konstrukció hiánya az utasítás adatként való elérésére->assembler szükséges).

Harvard-Neumann együttes architektúra megvalósítás: A mai, nagy teljesítményű rendszereknél a kettőt együtt is lehet alkalmazni. Példa: Cache rendszer. Programozói szemlélet (Neumann): cache „miss” esetén a fő memóriából kell kivenni az adatot. Rendszer, hardver szemlélet (Harvard): A CPU on-chip cache memóriája különálló adat- és utasítás cache blokkokból áll.

2.Tétel: Az információ reprezentációi és az ALU felépítése

Egész (integer) típusú számrendszer:

A számítógépek számbábrázolása és a legtöbb kommunikációs eljárás a **bit** fogalmán alapul. A bit olyan változó mennyiség, amely 0 és 1 értéket tud felvenni. Bitek csoportja alkotja a számokat, minden egyes új bitpozíció megduplázza a lehetséges ábrázolható értékek számát, tehát a rendelkezésre álló bitek száma határozza meg a reprezentálható értékek számát (N bit esetén 2^N lehetséges érték van), de nem közvetlenül befolyásolja azok értékhatárát. Az **előjel nélküli egészeket (unsigned integer)** a legegyszerűbb elképzelés szerint bináris számokkal ábrázoljuk. Ebben az esetben a reprezentálható értékek határa 0-tól 2^{N-1} -ig terjedhet. A bitszoporttal ábrázolható érték a következő:

$$V_{\text{UNSIGNED INTEGER}} = \sum_{i=0}^{N-1} b_i \times 2^i$$

Itt b_i az i . pozícióban elhelyezkedő 0-t vagy 1-et jelöli. Pl.: 101101 bitmintázat esetén $1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 45_{10}$.

Ez a számbábrázolás nem alkalmas negatív számok reprezentálására.

A legszélesebb körben használt rendszer a **kettes komplement (two's complement) rendszer**. A 2^N reprezentálható érték határa $-(2^{N-1})$ -től $2^{N-1}-1$ -ig terjed. Pl.: 5 bit esetén $-(2^4)$ -től 2^4-1 -ig. Egy érték negáltját 2^N -ből való kivonással kaphatjuk meg. A kettes komplement megjelenítési értéke a következő módon adható meg:

$$V_{2\text{'S COMPLEMENT}} = -b_{N-1} \times 2^{N-1} + \sum_{i=0}^{N-2} b_i \times 2^i$$

Pl: a 10010011 bitmintázat 2-es komplement ábrázolás esetén $-1 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = -109_{10}$ jelent. A legnagyobb helyiértékű bitet Most Significant Bit-nek (MSB) nevezzük. Ha az MSB 1, akkor a szám értéke is negatív lesz, mivel az MSB-nek van a legnagyobb súlya. Ebből következik, hogy pozitív számoknál az MSB 0-ra van állítva. Egy másik példa: -76_{10} bitmintázata egy 8 bites, 2-es komplement rendszerben: Először megkeressük a 76_{10} 8 bites bináris bitmintázatát, amely a 01001100. Hogy a negáltját megkapjuk, kivonjuk 2^8 -ból:

$$\begin{array}{r} 1\ 00000000 \\ -\ 01001100 \\ \hline 10110100 \end{array} \qquad \begin{array}{r} 256 \\ -76 \\ \hline 180 \end{array}$$

Tehát 10110100 decimális értéke megegyezik 180-al. De -76_{10} bináris bitmintázata a következő módszerrel is megadható: először felírjuk a 76 bináris bitmintázatát, majd bitenként negáljuk és hozzáadunk 1-et, végül a kapott értékeket összeadjuk:

$$\begin{array}{r} 01001100 \\ 10110011 \\ + \quad 1 \\ \hline 10110100 \end{array}$$

A kettes komplement rendszer egyik megvalósítása a **körkörös állapot számláló (circular nature)**. Ez egy 4 bites kettes komplement rendszer, amelynél a számok a kör mentén sorrendben helyezkednek el 1000-tól 0111-ig (-8-tól 7-ig). A számok között lépkedhetünk, egyszerűen 1-el való növeléssel, de a 0111 és az 1000 határán a szám pozitívból negatívba vált át, aminek következtében szakadás lesz a számsorozatban. Ezt túlsordulásnak (overflow) nevezzük, ami azt jelenti, hogy a művelet meghaladja a számrendszer ábrázoló képességét. Az alulcsordulás is hasonló, csak fordított: ha 1000-t dekrementáljuk 1-el, akkor 0111-re lépünk (negatívból pozitív).

Fixpontos számrendszer:

Általánosan elfogadott tény, hogy a tizedespont közvetlenül a legkisebb helyiértékű számjegytől jobbra helyezkedik el. Ezzel a megkötéssel valójában lehetséges egész számokat ábrázolni. Azonban ha a tizedespont a legkisebb helyiértéktől pár bitpozícióval távolabb helyezkedik el, akkor a reprezentálható értékhatárok megváltoznak. Ha a reprezentálni kívánt információ tartalmaz tört értéket, akkor a tizedespont helyét úgy kell meghatározni, hogy az lefedje az ábrázolni kívánt tartományt. Az összeadás és kivonás hasonlóan működik az egész típusú rendszerekhez. A szorzás és osztás elvégzése után a tizedespont helyét is lehet hogy módosítani kell. Például két 16 bites szám szorzata 32 bites lesz, függetlenül a tizedespont helyzetétől. Ha a két 16 bites szám eredményét 16 biten szeretnénk tárolni, akkor ez a határ szabja meg a szám méretét. A fixpontos rendszer is helyiértékes, az egyetlen különbség, hogy a tizedespont helyének változása miatt egy új tényezőt kell bevezetni: ennek jele p , amely a tizedespont feltételezett helyét mutatja (a legkisebb helyiértékű bittől balra hány bitpozícióval van arrébb). A következő egyenlettel egy 2-es komplementes fixpontos szám értékét lehet megadni:

$$V_{\text{FIXED POINT}} = -b_{N-1} \times 2^{N-p-1} + \sum_{i=0}^{N-2} b_i \times 2^{i-p}$$

A legjellemzőbb érték, amivel meghatározhatjuk egy számrendszer használhatóságát, a differencia, amelyet Δr -el jelölünk. A differencia jelentése: két szomszédos szám közötti különbség, abszolút értékben. Az egész típusú rendszerek esetén a differencia 1, fixpontos rendszerek esetén 2^{-p} (finomság).

Egy másik széles körben használt módszer az **egyes komplementes** rendszer. Egy V értékkel rendelkező N bites rendszer matematikailag a következő képlettel határozható meg: $2^N - 1 - V$. A képlet $2^N - 1$ része N db egyesnek felel meg, amiből V -t kivonva a bitmintában 0 lesz ott ahol 1-es volt és 1 lesz ott ahol 0 volt (mivel egy szám negatív alakját bitjeinek kiegészítésével adjuk meg). Egy szám negáltját tehát bitjeinek negálásával kaphatjuk meg, ami nagyon gyors műveletet eredményez. A lefedett értékhatár $(2^{N-1} - 1)$ –től $(-2^{N-1} - 1)$ –ig terjed. Az eddig megismert rendszerekhez képest ez helyiértékes rendszer, mivel a bitek helyiértékbeli eltérése a szám előjelétől függ. További fontos számábrázoláshoz használt kódolási technika az **Excess kódrendszer**. Az egyik leggyakoribb felhasználási módja a lebegőpontos számok kitevőinek tárolása. Legyen S a reprezentálni kívánt érték (eredmény), amit tárolni fogunk, V a szám valódi értéke és E az excess. Ekkor a következő kapcsolat áll fenn közöttük: $S = V + E$. Ennél a műveletnél figyelni kell arra, hogy az eredmény a kívánt határon belül legyen, mivel ha két számot összeadunk, akkor a következő történik: $S_1 + S_2 = (V_1 + E) + (V_2 + E) = (V_1 + V_2) + 2 * E$. Ahhoz, hogy megkapjuk a pontos eredményt-amely a $[(V_1 + V_2) + E]$ -a számított eredményből ki kell vonnunk E -t.

Lebegőpontos számrendszer (Floating Point Number System, FPNS)

Sok probléma megoldásához olyan rendszerre van szükség, ami nagyságrendekkel nagyobb vagy kisebb értéket is meg tud jeleníteni, mint mondjuk a fixpontos rendszer. Egy lebegőpontos szám meghatározásához 7 különböző tényező szükséges: **a számrendszer alapja, előjele és nagysága, a mantissza alapja, előjele és hosszúsága illetve a kitevő alapja**. Az ilyen típusú számok matematikai jelölése a következő:

$$(\text{előjel}) \text{ Mantissza} \times \text{Alap}^{\text{Kitevő}}$$

Az alap a rendszer alapszámára vonatkozik, ami decimális rendszer esetén 10 (0-9-ig). Az alapszámot a rendszer definiálásakor kell meghatározni és közvetlenül befolyásolja az értékhatárokat. A későbbiekben jelöljük ezt r_b -vel a rendszer alapszámát, ami a mantisszára vonatkozik. A mantisszát egy szám értékjegyeinek meghatározásához használjuk. A lebegőpontos rendszer egy fontos jellegzetessége, hogy a mantisszát a jegyek száma reprezentálja, ezt m -el jelöljük. Ezért egy speciális lebegőpontos rendszer esetén minden egyes mantissza m darab r_b alapú számjegyből áll. A mantissza értékét jelöljük V_M -el. Emellett ismernünk kell a mantissza legkisebb és legnagyobb értékét, amit $V_{M(MAX)}$ és $V_{M(MIN)}$ -el jelölünk. Egy lebegőpontos számmal kifejezett valós számsor érték helyeit a kitevő (exponens) határozza meg. Ha az exponens nagy pozitív szám, akkor a lebegőpontos szám

értéke is nagyon nagy lesz, ha pedig az exponens nagy negatív szám, akkor a lebegőpontos szám értéke nagyon kicsi lesz. Ha az exponens 0, akkor a lebegőpontos szám értéke éppen a mantissza értékével lesz egyenlő. A kitevő értékének meghatározásához 3 adat szükséges: előjele, alapja és nagysága. A későbbiekben az exponens alapját r_e –vel jelöljük. A számrendszer alapjához hasonlóan az exponens alapját is a rendszer tervezésekor kell meghatározni. Az exponens maximális értékét jegyeinek száma határozza meg, amit e –vel jelölünk, amely a rendszer alapszámával együtt meghatározza az ábrázolható értékek tartományát. Az exponens e db r_e alapú számjegyből fog állni és meg kell határozni az előjelét is. Általában egy kiválasztott kódolási technika segítségével reprezentálható az exponens pozitív és negatív értékeit, például az excess kódolás. Az exponens értékét V_E -vel jelöljük. Ahogy a mantisszánál, itt is kell ismernünk a legnagyobb és legkisebb ábrázolható exponens értékét.

Ezeket jelöljük $V_{E(MAX)}$ –al és $V_{E(MIN)}$ -nel. Magának a számnak az előjelét is ismernünk kell, amelyet a tudományos módszerekkel egyértelműen meghatározhatunk. Egy ilyen az előjel-hossz (sign magnitude) technika, amelyet a lebegőpontos számok tárolásánál alkalmaznak. Végezetül ismernünk kell a tizedespont helyét is, amelyet p –vel jelölünk a legkisebb helyiértékű bitre vonatkozólag, és ennek segítségével tudjuk meghatározni az (előjel nélküli) mantissza értékét:

$$V_M = \sum_{i=0}^{N-1} d_i \times r_b^{i-p}$$

ahol a mantissza N darab r_b számjegyből áll ($d_{N-1} - d_0$ –ig). Ezért a lebegőpontos szám értékét ($V_{FPN} - t$) a következő módon tudjuk megadni:

$$V_{FPN} = (-1)^{SIGN} V_M \times r_b^{V_E}$$

A mantissza tizedespontjának helye közvetlenül kapcsolódik az exponens értékéhez. tekintsük a következő szám lebegőpontos értékét:

$$32,768_{10} = 3.2768 * 10^4 = 32.768 * 10^3 = 327.68 * 10^2 = 3267.8 * 10^1$$

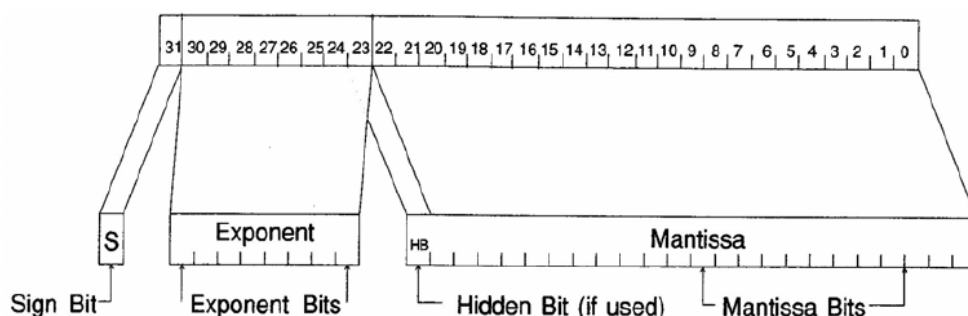
Mindegyik megjelenítési módban helyes eredményt kapunk, csupán a matematikai jelölésben van eltérés és a tizedespont helyét az exponens értéke határozza meg. Ha tehát a tizedespont helye egy adott számon belül jegyről jegyre változik, akkor mindenképpen szükséges p értékét úgy rögzíteni, hogy felhasználható legyen minden egyes későbbi számítás során. Továbbá a legtöbb rendszerben megkötések tesznek a tárolt információ minimalizálása és a műveletvégzés megkönnyítése érdekében. Ezt az eljárást **normalizálásnak** nevezzük, mellyel meghatározhatók a megengedett mantissza értékek. A fenti példában szereplő feltétel: legyen $p=M$ és balról a legszélső mantissza-számjegy ne legyen 0. Így a mantissza egy törtszám, amelynek értéke $1/r_b$ és közelítőleg 1 között változik. Speciálisan, a mantissza maximális értéke: $V_{M(MAX)} = 0.d_m d_m d_m \dots$ egészen a mantissza hosszának végéig, ahol $d_m = r_b$, és ez a szám nagyon közel van az 1-hez ($1 - r_b^{-N}$), míg a mantissza minimális értéke $V_{M(MIN)} = 0.100 \dots = 1/r_b$. Egy normalizált lebegőpontos rendszerben ábrázolt szám tehát megadható a $V_{M(MIN)}$ –től $V_{M(MAX)}$ –ig terjedő legális mantissza és a rendelkezésre álló exponens kombinációjával. Ezekből az észrevételekből a következő kifejezések vezethetők le:

- Ábrázolható maximális érték: $V_{FPN(MAX)} = V_{M(MAX)} * r_b^{V_{E(MAX)}}$
- Ábrázolható minimális érték: $V_{FPN(MIN)} = V_{M(MIN)} * r_b^{V_{E(MIN)}}$
- Legális mantisszák száma: $NLM_{FPN} = (r_b - 1) * r_b^{m-1}$
- Legális exponensek száma: $NLE_{FPN} = V_{E(MAX)} + |V_{E(MIN)}| + I_{ZERO}$
- Ábrázolható értékek száma: $NRV_{FPN} = NLM_{FPN} * NLE_{FPN}$

Ezek az értékek segítenek megadni egy lebegőpontos rendszer jellemzőit.

A számítógépen tárolt információk a következők: előjel, exponens és mantissza. Ezeket úgy csoportosítják, hogy az előjel legyen a legmagasabb helyiértékű bit, amit az exponens, végül a mantissza követ.

Azokat az adatokat, amelyek a számítások során soha nem változnak, természetesen nem tároljuk el. Ilyen például a rendszer alapszáma és az exponens alapja. Szintén ilyen, bár nem ennyire nyilvánvaló a tizedespont helye illetve az exponens tárolásának kódolási technikája. Ezeket mindig a rendszer tervezésekor kell megválasztani illetve konstansként definiálni. Ilyen konstans információ lehet egy 2-es alapú normalizált rendszerben a számok elején szereplő 1-esek ("vezető" egyesek). Csak azokat az értékeket kell eltárolni, amelyek megváltozhatnak. Ilyen a mantissza legmagasabb helyiértékű bitje, amelyet rejtett (hidden) bitnek hívunk, és amely közvetlenül az exponensbitek mögött helyezkedik el. Ha ezt a rejtett bitet beállítjuk, duplájára nő a legális mantisszák és ezzel együtt az ábrázolható értékek száma. Másrészt a nullát nem tudjuk reprezentálni, mivel a legkisebb ábrázolható érték 000000 , ami a rejtett bit használata miatt $0.1000_2 * 2^0 = \frac{1}{2}$ -nek felel meg. A számítógépes tárolás a következőképpen néz ki:



Mégis hogyan lehet a nullát ábrázolni? Egy általános módszer, hogy az exponenst illetően különféle megközelítéseket teszünk. A leggyakoribb technika, hogy excess 2^{e-1} kódolást alkalmazunk az exponens ábrázolásánál. Mivel a kód bináris reprezentációja ennek a kódnak $2^e - 1$ -től 1-ig változik, ezért az exponens ennek megfelelően 2^{e-1} -től $-(2^{e-1} - 1)$ -ig terjedhet. Ha az exponens mindegyik bitje 0 (a mantisszától függetlenül), akkor a feltételezett szám is 0.

ALU felépítése:

ALU: Arithmetic Logical Unit rövidítése. Aritmetikai / Logikai egység. Olyan adatkezelő egység, amely matematikai/logikai műveletek végrehajtását végzi.

Matematikai műveletek lehetnek: összeadás, kivonás, szorzás, osztás. Logikai műveletek: AND, OR, NOT, NOR, NAND, XOR, XNOR, EQ, stb..

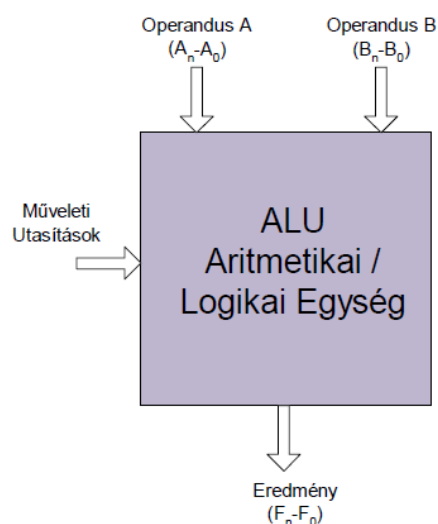
Ezeket a műveleteket minimális kapu számával valósítja meg, ezáltal minimális a késleltetés is.

Kombinációs logikai hálózatok esetén a NAND illetve NOR logikai alapelemek univerzálisak, mivel az összes logikai függvény megvalósítható belőlük különböző kombinációval.

Ez a megállapítás igaz az aritmetikai műveletekre nézve is, ahol egyedül az összeadót tekintjük alapvető építőelemnek. Segítségével az összes aritmetikai művelet származtatható.

Utasítások hatására a_0 (S0-Sn) vezérlőjelek kijelölik a végrehajtandó aritmetikai vagy logikai műveletet.

További adatvonalak kapcsolódhatnak közvetlenül a státusz regiszterhez, amelyek fontos információkat tárolnak el: pl. carry-in, carry-out, átviteletet, előjel bitet (sign), túlcsoordulást (overflow), vagy alulcsordulást (underflow) jelző biteket.



Státuszbit: Az aritmetikai műveletek eredményétől függően hibajelzésre használatos jelzőbit. Általában 4 típusuk van: előjel, zéró, túlsordulás (overflow), átvitel (carry). Ezeket a státuszregiszterekben tároljuk el. A státuszbit megváltozása az utasításkészletben előre definiált utasítások végrehajtásától függ.

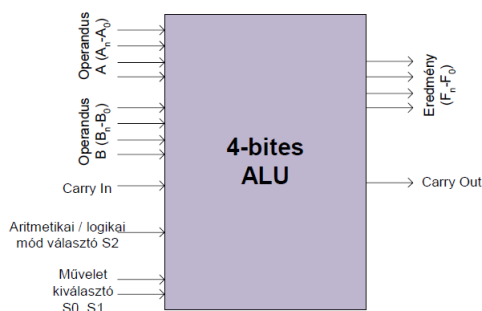
Előjelbit: Legkönnyebb generálni. Az eredmény előjelétől függ: 2-es komplement képzés esetén az MSB (legnagyobb helyiértékű) bit jelöli az előjelbitet. Ha MSB=1 akkor a szám negatív, különben pozitív. Közvetlenül a státuszregiszterbe töltődik. Csak bizonyos utasítások vagy aritmetikai műveletek (pl. összeadás, kivonás) módosíthatják az előjelbitet.

Carry vagy átvitel kezelő bit: Ha egy aritmetikai művelet átvitelt eredményez egyik helyiértékről a másikra, akkor a státuszregiszter beállítja az átvitelbitet.

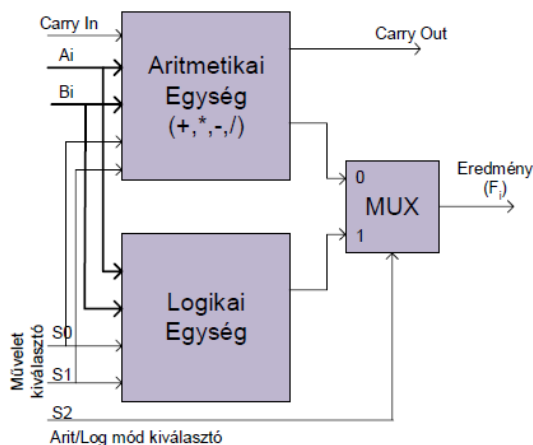
Zéró bit: Ha egy művelet eredménye 0, akkor a státuszregiszterben beállítjuk a zéró bitet. Általában a zéró bit megváltozását a logikai műveletek befolyásolják (az aritmetikai műveletek mellett).

Túlsordulás bit: Jelzi, hogy a rendszer a számbárázolási tartományán egy adott aritmetikai művelet eredménye kívül esik –e vagy sem, tehát a szám ábrázolható vagy nem. Ekkor beállítódik ez a státuszbit a státuszregiszterben. Túlsordulás egy 2-es komplement rendszer esetén akkor fordulhat elő, ha két pozitív számot összeadva eredményként negatív számot kapunk illetve ha két negatív számot összeadva pozitív számot kapunk. A túlsordulás detektálásának megvalósítása, hogy figyeljük a számok előjeleit. Ha a számok pozitívak, vagyis előjelbitjeik 0-ák és az eredmény előjele negatív, akkor túlsordulás következett be. Ennek detektálása két AND illetve egy OR kapuval megvalósítható.

Az ALU felépítése és működése: Az alábbi ábrán két 4-bites A illetve B számon tudunk aritmetikai/logikai műveletet végrehajtani. Az F eredmény a 4-bites kimeneten képződik. A Carry-in / Carry-out az átvitel kezelésére szolgáló bemeneti illetve kimeneti vonal. Az S2 vonal segítségével egy MUX-on (multiplexer) keresztül kiválasztjuk, hogy aritmetikai vagy logikai művelet szeretnénk végrehajtani (aritmetikai műveleteknél S2="0", míg logikainál S2="1"). Az S1, S0 és a Cin megfelelő értékétől függ, hogy pontosan milyen műveletet hajtunk végre. A 4-bites ALU szimbolikus rajza az alábbi:



Az ALU részletesebb felépítése:



3. Tétel: Vezérlőegységek (modell-implementáció)

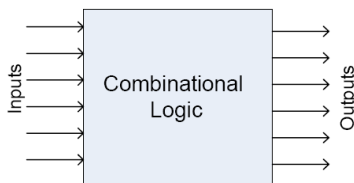
Vezérlőegységek: a számítógép vezérlési funkcióit ellátó szekvenciális egység. Feladata az operatív tárban lévő gépi kódú utasítások értelmezése, részműveletekre bontása, és a szekvenciális (sorrendi) hálózat egyes funkcionális részeinek vezérlése (vezérlőjel- és cím-generálás).

A vezérlő egység feladata a memóriában lévő gépi kódú program utasításainak az értelmezése, részműveletekre bontása, és ezek alapján az egyes funkcionális egységek vezérlése.

Vezérlőegységek fajtái:

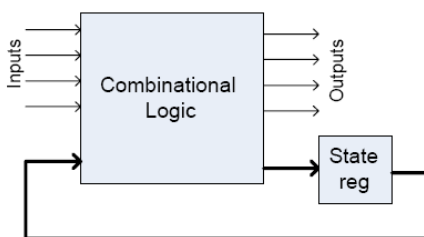
- Huzalozott
 - **Mealy-modell**
 - **Moore-modell**
- Mikroprogramozott (reguláris vezérlési szerkezettel):
 - **horizontális mikrokódos vezérlő**
 - **vertikális mikrokódos vezérlő**
- Programozható logikai eszközök (PLD):
 - **PLA, PAL, PROM, CPLD**
 - **FPGA**

Kombinációs logikai hálózatról beszélünk: ha a mindenkori kimeneti kombinációk értéke csupán a bemeneti kombinációk pillanatnyi értékétől függ (memória nélküli hálózatok).



1. ábra Kombinációs logika

Sorrendi logikai hálózatról beszélünk, ha a mindenkori kimeneti kombinációt nem csak a bemeneti kombinációk, hanem a korábbi bemenetek is befolyásolják (azonos bemenet esetén az állapotregiszter tartalmát is figyelembe véve különböző kimenetet adhat).



2. ábra Sorrendi hálózati logika

Az időzítő (ütemező) határozza meg a vezérlő jelek előállításának sorrendjét. Egy **időzítő-vezérlő** egység általános feladata az egyes funkciók megvalósítását végző áramköri elemek (pl. ALU, memória elemek) összehangolt működésének biztosítása.

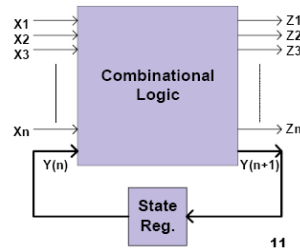
Az **időzítő-vezérlő** áramkörök szekvenciális rendszerek mivel az áramköri egységek tevékenységének egymáshoz viszonyított időbeli sorrendiségét biztosítják melyek az aktuális kimenet értékét a bemenet és az állapotok függvényében határozzák meg.

Az **időzítő vezérlő** lehet:

- **huzalozott:** áramkörökkel, dedikált összeköttetésekkel fizikailag megvalósított (Mealy, Moore, PLD-k)
- **mikroprogramozott:** az adatútvonal (data-path) vezérlési pontjait a memóriából (ROM) kiolvasott vertikális vagy horizontális mikrokódú utasításokkal állítják be.

Huzalozott vezérlő egységek: Mealy-modell, Moore-modell.

Mealy-modell: a sorrendi hálózatok egyik alapmodellje. A kimeneten az eredmény véges időn belül jelenik meg. Korábbi értékek visszacsatolódnak a bemenetre: kimenetek nem csak a bemenetek pillanatnyi állapotától függenek, hanem a korábbi állapotoktól is. A bemenetek és az állapotok közötti szinkronizáció hiánya problémát okoz ezért legtöbb esetben a Moore-modellt használjuk.

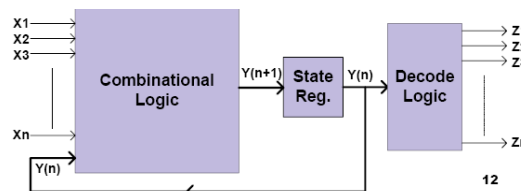


3. ábra Mealy-modell

X: bemenetek, Z: kimenetek, Y: állapotok halmaza

Moore-modell:

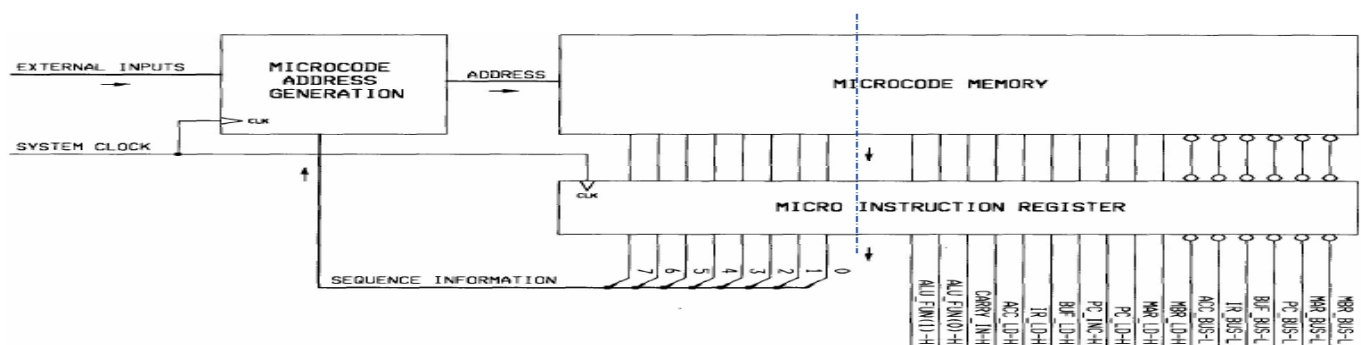
A kimenetek közvetlenül csak a pillanatnyi állapottól függenek (bemenettől független vagy közvetve függő csak) Tehát a kimenetet nem a bemenetekhez, hanem az állapotoknak megfelelően szinkronizáljuk.



4. ábra Moore modell

Mikroprogramozott vezérlő egységek:

Állapotgépekkel (FSM) modellezik a vezérlő egység működését, és ezt a modellt transzformálják át mikrokódot használva. Az adatútvonal vezérlési pontjait a memóriából (ROM) kiolvasott vertikális vagy horizontális mikrokódú utasításokkal állítják be.



5. ábra Általános mikrokódos vezérlő

Mikroprogram: mikroutasítások sorozatából áll. Egy gépi ciklus alatt egy mikroprogram fut le. A műveleit kód a végrehajtandó programot jelöli ki. A mikrokódú memória általában csak olvasható ROM, ha írható is akkor dinamikus mikroprogramozásról beszélünk.

Ha a mikroprogram utasításai szigorúan szekvenciálisan futnak le, akkor a címüket egy egyszerű számláló inkrementálásával megkaphatjuk. Memóriából érkező bitek egyik része a következő cím kiválasztását míg a fennmaradó bitek az adatáramlást biztosítják.

Lassabb mint a huzalozott vezérlő egységek, hiszen itt a memória elérési idejével is számolni kell (nem csak a vissza csatolt állapot késleltetésével, mint a huzalozottaknál).

A mikrokódos vezérlőknek két fajtája van: horizontális és vertikális.

Horizontális mikrokódos vezérlő: minden egyes vezérlőjelhez saját vonalat rendel, emiatt horizontálisan megnő a mikro- utasításregiszter kimeneteinek a száma (vagyis horizontálisan megnő a mikrokód) minél több funkciót valósítunk meg a vezérlőjelekkel annál szélesebb lesz a mikrokód.

Ezeknek köszönhetően a leggyorsabb a mikrokódost technika mivel minden bit független egymástól így egymástól, egy mikrokóddal többszörös utasítás is megadható. Pl a megfelelő regisztereket (memória, ACC) egyszerre egy időben lehet az órajellel aktiválni ezáltal egy órajel ciklus alatt mindkét irányba átvihető az információ. Növekszik a sebesség mivel nincs szükség a vezérlőjelek dekódolására. Azonban ezekkel együtt nagyobb az erőforrás szükséglete, fogyasztása.

Vertikális mikrokódos vezérlő: nem a sebességen van a hangsúly, hanem a takarékoságon (fogyasztás, mikrokódban bitekszám). Egyszerre csak a szükséges biteket kezeljük, egymástól nem teljesen függetlenül, mivel közülük egyszerre csak az egyiket állítjuk be. A jeleket ezután dekódolni kell. A kiválasztott biteket megpróbáljuk a minimális számú vonalon keresztül továbbítani. A műveletek párhuzamos végrehajtása korlátozott. Dekódolás: N bites buszt, $\log_2(N)$ számú bittel próbálunk dekódolni, több mikROUTASÍTÁS szükséges ehez ezért a mikrokódú memóriát „vertikálisan” kell megnövelni.

Horizontális és vertikális mikrokódos vezérlők összehasonlítása: a horizontális mikrokódos vezérlő minden vezérlőjelnek saját vonala van ezért lehetséges a többszörös utasítások megadása. A vertikális mikrokódos vezérlő kevesebb vonallal rendelkezik több mikrokódú utasítással rendelkezik, ezek miatt lassabb (és takarékosabb).

Programozható logikai eszközök (**PLD**)

Két fő típusa:

- makrocellás PLD-k : **PLA, PAL, PROM, CPLD**
- programozható gate array áramkörök FPGA (field Programmable Gate Array)

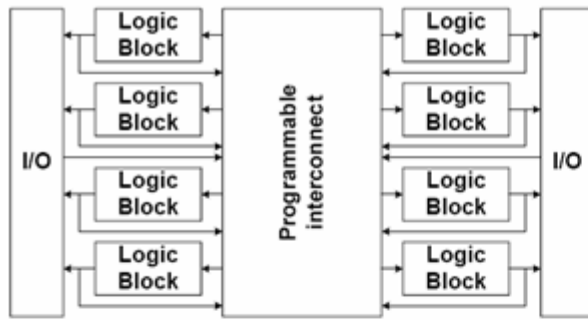
PLA: mind a két része programozható az áramkörnek (OR/AND). Bármely OR/AND kombináció előállítható belőle. Mintermek OR kapcsolata (DNF, diszjunkt normál forma). Programozásuk: a metszés pontokban biztosítékok (fuse) helyezkednek el amik gyárilag a logikai 1-et definiálják. Ha valamilyen eszközzel feszültséget kapcsolnak rá akkor átégethetők az adott részekben, ezek fogják a nullákat reprezentálni. (természetesen a programozás egyszer lehetséges, az átégetés miatt). A kimeneteken D tárolók (visszakapcsolódhatnak a bemenetekre).

PROM: programozható, csak olvasható memória. Egyszer írható, tartalmát tápfeszültség nélkül is megőrzi. Egy programozható része az OR áramkörei, az AND része fix. Gyorsabb mint a PLA. Az OR/AND kapcsolóknak véges kombinációja áll elő. A kimeneteken D tárolók (visszakapcsolódhatnak a bemenetekre).

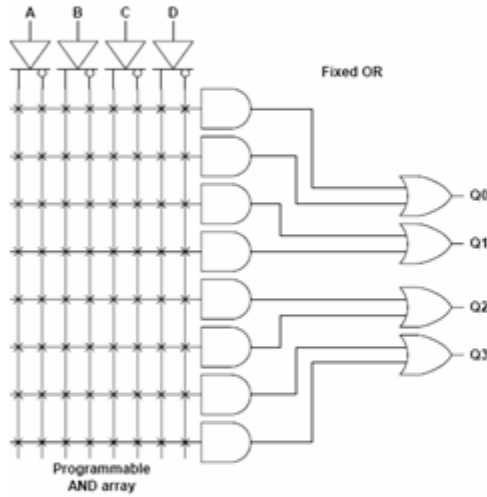
PAL: az AND része programozható amíg az OR része fix. Véges kombinációja áll elő az AND / OR kapcsolatoknak. Metszéspontokban kevesebb kapcsoló szükséges. Gyorsabb mint a PLA. A kimeneteken D tárolók (visszakapcsolódhatnak a bemenetekre).

CPLD (complex programmable logic device):

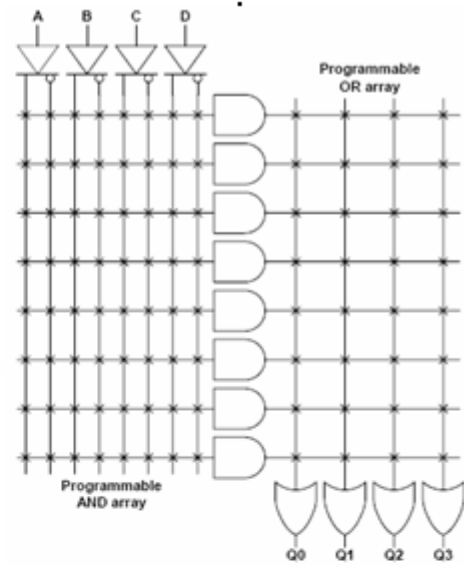
Logikai blokkban PAL/PLA, regiszterek. A logikai blokkok között programozható összeköttetések (teljes vagy részleges összeköttetés is lehet).



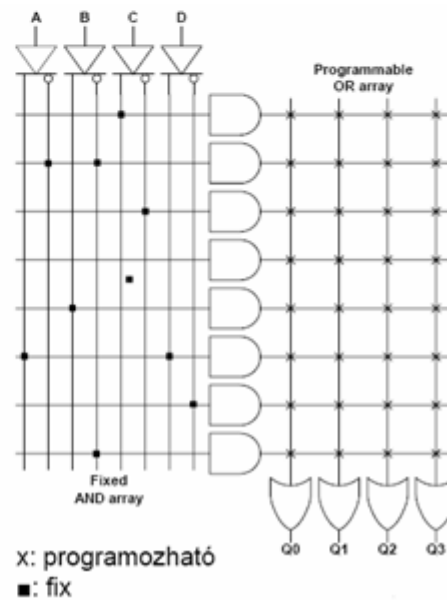
6. ábra CPLD



8. ábra PAL



7. ábra PLA



9. ábra PROM

4. Tétel: Folyamatok kezelése multiprogramozott rendszerekben. Folyamatok ütemezése és szinkronizációja.

Multiprogramozás: Amíg egy program egy művelet befejeződésre (a lemeztől olvasás eredményére) vár, addig egy másik program működhet. Az OS egyszerre több munkát futtat.

A multiprogramozás lépései:

- A rendszer nyilvántartja és tárolja a futtatandó munkákat.
- A kiválasztott munka addig fut, amíg várakozni nem kényszerül.
- Az OS feljegyzi a várakozás okát, majd kiválaszt egy másik futni képes munkát és azt elindítja.
- Ha a félbehagyott munka várakozási feltételei teljesülnek, akkor azt elindítja.

Multiprogramozásban felvetődő problémák:

- Az átkapcsoláshoz több program van a tárban: tárgazdálkodás.
- Egy időben több futásra kész program: CPU ütemezés.
- A gépi erőforrások felhasználásának koordinációja: allokáció, holtpont kezelése.
- Védelmi mechanizmusok, hogy a programok ne zavarják egymást és az OS-t.

A folyamat (process): Végrehajtás alatt álló program. Multitaskos rendszerben több folyamat van a rendszerben egyidejűleg.

Elnevezések:

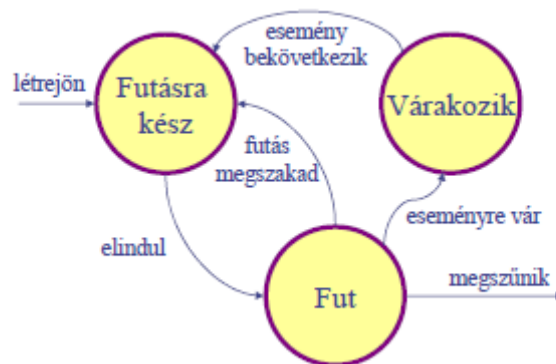
- munka (job) - batch rendszereknél.
- feladat (taszk) - real time.
- felhasználó - időosztásos rendszereknél.

Folyamatok alap állapotai:

- Több folyamat, 1 CPU → látszólagos párhuzamosság.
- CPU kitüntetett erőforrás, egy adott pillanatban csak 1 program hajtódik végre.
- Gráffal lehet ábrázolni.

Állapotok:

- **Fut:** A CPU a folyamathoz tartozó utasításokat hajtja végre, CPU-nként egyetlen ilyen folyamat lehet.
- **Várakozik, blokkolt:** A folyamat várakozni kényszerül, működését csak valamilyen esemény bekövetkezésekor tudja folytatni. Több ilyen is lehet a rendszerben.
- **Futásra kész:** Minden feltétel adott, a CPU éppen foglalt. Több ilyen is lehet a rendszerben.



6. ábra Folyamatok állapot-átmeneti gráfja

Folyamatok állapotátmenetei:

• Folyamat létrehozása

Mikor jön létre egy folyamat?

- Rendszer elindításakor (boot):
- Folyamatot létrehozó rendszerhívás hatására (pl. fork):
- Hierarchia (UNIX): „process group”.
- Minden folyamat egyenlő (Windows).
- Felhasználó: Program indítása.
- Új batch-job indítása:
- Létrehozó folyamat a szülő, a létrejöttek a gyerekek.
 - Ha erőforrás kell neki, az OS-től kapja, a szülő erőforrásain osztozik.
 - Szülőtől paramétereket kap (befolyásolja a futását).
- A gyerek a szülővel párhuzamosan fut, vagy bevárja a gyerekének, gyerekeinek befejeződését.

• Folyamat befejeződése

- Önszántából: végrehajtotta utolsó utasítását vagy hiba miatt leáll.
- OS vagy rokon (általában a szülő) leállítja:
 - Hibás utasítás, erőforrás használat túllépése.
 - kill utasítás (másik folyamat).
- Erőforrások felszabadulnak:
 - attól függően, hogy kitől kapta: felszabadul vagy szülőhöz kerül.
- Szülőnek információt adhat vissza.

• Eseményre vár

• Fut → várakozik. Valamit kér, amire várnia kell: jelzés, erőforrás, stb. OS feljegyezi, hogy ki mire vár.

• Esemény bekövetkezik

- Várakozik → futásra kész. A várt esemény bekövetkezett, de még nem fut!

• Folyamat elindul

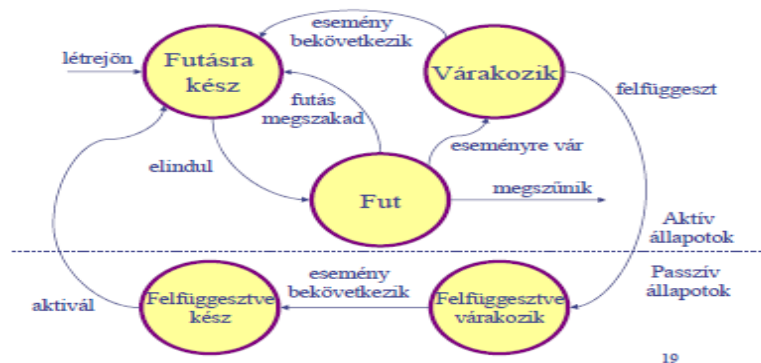
- Futásra kész → fut. Ha a CPU felszabadul, egy folyamat futhat. **CPU ütemezés.**

• Futás megszakad

- Fut → futásra kész.
 - Önként lemond a CPU-ról. Kooperatív viselkedés: hosszú feladatok esetén újraütemezést kér.
 - OS elveheti a CPU-t, még akkor is, ha a folyamat egyébként nem kényszerül várakozásra (preemptív ütemezés). Időosztásos rendszerek

Folyamatok kibővített állapotai:

- Bővítés: az OS felfüggeszthet folyamatokat (középtávú CPU ütemezés)
 - a rendszer túl van terhelve (sok program vetélkedik a futás jogáért, vagy a tár túlzottan megtelt).
 - vészhelyzet esetén.
 - felhasználó kezdeményezésére.
- A felfüggesztett folyamatok erőforrásaikat elvesztik (de a rendszer számon tartja őket, később folytatódhatnak).
- Két új állapot: **felfüggesztve vár** és **felfüggesztve kész**.



7. ábra Folyamatok kibővített állapot-átmeneti gráfja

Új állapotátmenetek:

- **Felfüggeszt:**
 - OS felfüggeszti (futásra kész vagy várok állapotból).
 - erőforrásokat elveszi (pl. memória), néhányat megtarthat (pl. nyomtató).
- **Aktivál:** az OS visszaadja az erőforrásokat.
- **Felfüggesztve várok → felfüggesztve futásra kész:**
 - Esemény bekövetkezik, de CPU-t nem kaphat.

Az **ütemezés (scheduling)** az a tevékenység, amely eredményeként eldől, hogy az adott erőforrást a következő pillanatban mely folyamat használhatja.

Az ütemezés típusai:

1. **Nem preemptív:** ha egy folyamattól, miután megkapta a CPU-t, nem lehet azt elvenni.
 - a folyamat csak az általa kiadott utasítások hatására vált állapotot:
 - erőforrásra, eseményre várakozás.
 - befejeződés vagy a CPU-ról önként lemondás.
2. **Preemptív:** ha az OS elveheti a futás jogát egy folyamattól.
 - futásra kész állapotba teszi a futó folyamatot és egy másik (futásra kész) folyamatot indít el.
 - Pl. időosztásos, valós idejű OS-ek.

Ütemezési algoritmusok:

1. Egyszerű ütemezési algoritmusok:

a.) Legrégebben várakozó (First Come First Served, FCFS).

- A futásra kész folyamatok a várakozási sor végére kerülnek, az ütemező a sor elején álló folyamatot kezdi futtatni.
- Nem preemptív.
- Egyszerűen megvalósítható.
- Konvoj hatás (egy hosszú CPU löketű folyamat feltartja a mögötte várakozókat).

b.) Körforgó (Round-Robin, RR):

- Preemptív algoritmus, az időosztásos rendszerek valamennyi ütemezési algoritmusainak az alapja.
- Folyamatok időszeletet kapnak (time slice).
 - Ha a CPU löket nagyobb mint az időszelet, akkor az időszelet végén az ütemező elveszi a CPU-t, a folyamat futásra kész lesz és beáll a várakozó sor végére.
 - Ha a CPU löket rövidebb, akkor a löket végén a folyamatokat újraütemezzük.
- Körforgó ütemezés időszelete:
 - Nagy időszelet: FCFS algoritmushoz hasonló lesz.
 - Kis időszelet: folyamatok a CPU-t egyenlő mértékben használják, viszont a sok környezetváltás a teljesítményt rontja.

2. Prioritásos ütemező algoritmusok:

- A futásra kész folyamatokhoz egy prioritást (rendszerint egy egész számot) rendelünk.
- A legnagyobb prioritású folyamat lesz a következő futtatandó folyamat.
- Prioritás:
 - Meghatározása:
 - belső: az OS határozza meg.
 - külső: az OS-en kívüli tényező (operátor, a folyamat saját kérése, stb.) határozza meg.
 - Futás során lehet:
 - **statikus (végig azonos).**
Prioritás meghatározása: Prioritást sokszor a löketidő alapján határozzák meg.
 - **dinamikus (az OS változtathatja).**

a.) Legrövidebb (löket)idejű (Shortest Job First, SJF):

- Nem preemptív algoritmus, a futásra kész folyamatok közül a legrövidebb löketidejét indítja.
- Nincs konvoj hatás, optimális körülfordulási idő, optimális várakozási idő.
- Alkalmazása:
 - Hosszú távú ütemezés.
 - Rövid távú ütemezés (RT rendszerek).

b.) Legrövidebb hátralévő idejű (Shortest Remaining Time First, SRTF):

- Az SJF preemptív változata.
- Ha egy új folyamat válik futásra készvé:
 - akkor az ütemező újra megvizsgálja a futásra kész folyamatok, illetve az éppen futó folyamat hátralévő löketidejét.
 - és a legrövidebbet indítja tovább.
- A folyamat megszakítása és egy másik elindítása környezetváltozást igényel, ezt az időt is figyelembe kell vennünk.

c.) Legjobb válaszarány (Highest Response Ratio):

- Az SJF algoritmus változata, ahol a várakozó folyamatok öregednek. A kiválasztás (a löketidő helyett) a löketidő **(löketidő + k * várakozási_idő)/löketidő** képlet szerint megy végbe, ahol k egy jól megválasztott konstans.

3. Többszintű algoritmusok:

- Futásra kész folyamatokat több külön sorban várakoztatják. • A sorokhoz prioritás van rendelve.
- Egy kisebb prioritású sorból csak akkor indulhat el egy folyamat, ha a nagyobb prioritású sorok üresek.
- Az egyes sorokon belül különböző kiválasztási algoritmusok működhetnek.

a.) Statikus többszintű sorok (Static Multilevel Queue, SMQ):

- A folyamatot elindulásakor valamilyen kritérium alapján besorolunk egy várakozó sorba.
- A folyamat élete során végig ugyanabban a sorban marad.
- Egy lehetséges példa a prioritások besorolására:
 - rendszer folyamatok (magas prioritás).
 - interaktív folyamatok (elfogadható válaszütem).
 - interaktív szövegszerkesztők (kevésbé kritikusak).
 - rendszerstatisztikákat gyűjtő folyamatok (alacsony prioritás).

b.) Visszacsatolt többszintű sorok (Multilevel Feedback Queues, MFQ):

- A sorokhoz egy időszület tartozik:
 - minél nagyobb a prioritás, annál kisebb az időszület.
- A folyamatok futásuk során átkerülhetnek másik sorokba.
- Nagyobb prioritásúból kisebb prioritású sorba:
 - Amennyiben nem elég az adott időszület, a folyamat egy kisebb prioritású sorba kerül át.
- Kisebb prioritásúból nagyobb prioritású sorba:
 - A folyamat átlagos löketidejének változása (csökkenése) esetén.
 - A régóta várakozó folyamat öregedése miatt.

4. Többprocesszoros ütemezés:

A futásra kész folyamatok a rendszer bármely processzorán elindulhatnak.

a.) Heterogén rendszerek esetében egy folyamat csak bizonyos processzorokon futhat.

b.) Homogén rendszerekben a futásra kész folyamatokat közös sorokban tárolja.

– Szimmetrikus multiprocesszoros rendszer:

Minden CPU saját ütemezőt futtat, amely a közös sorokból választ. A sorok osztott használatához a kölcsönös kizárást biztosítani kell!

– Aszimmetrikus multiprocesszoros rendszer:

Az ütemező egy dedikált CPU-n fut, ez osztja szét a folyamatokat a szabad CPU-k között.

Folyamatok szinkronizációja: A szinkronizáció a folyamatok közötti információ áramlás megvalósítására szolgál, az adott folyamat további futása egy másik folyamat futásától illetve egy külső esemény bekövetkezésétől függ.

• **Gyakori feladatok:**

a.) Precedencia (előidejűség).

- Meghatározott sorrend biztosítása.

b.) Egyidejűség.

- Két vagy több folyamat bizonyos utasításait (S_k ; S_j) egyszerre kell elkezdeni.
- Két folyamat találkozása (randevú).
- Két folyamat bevárja egymást mielőtt további működését elkezdene.

c.) Kölcsönös kizárás (versenyhelyzet, kritikus szakasz) (mutual exclusion):

- A résztvevő folyamatok utasításainak sorrendjére nincs korlátozás, de egy időben csak egyik futhat.

• **Egyéb kapcsolódó fogalmak:**

Holtpont (deadlock):

Folyamatok egy halmaza akkor van holtponton, ha a halmaz minden folyamata olyan eseményre* vár, amelyet csak egy másik, ugyancsak halmazbeli (várakozó) folyamat tudna előidézni.

Az esemény többnyire egy erőforrás felszabadulását jelenti.

Versenyhelyzet:

- Több párhuzamosan futó folyamat közös erőforrást használ.

A futás eredménye függ attól, hogy az egyes folyamatok mikor és hogyan futnak, ezáltal hogyan (milyen sorrendben) férnek az erőforráshoz.

Kritikus szakasz:

- Kritikus szakaszoknak nevezzük a program olyan (általában osztott változókat használó) utasításszekvenciáit, amelyeknek egyidejű (párhuzamos) végrehajtása nem megengedett.
- Versenyhelyzet elkerülésére a kritikus szakaszok kölcsönös kizárását biztosítani kell. (Ha az egyik folyamat már a kritikus szakaszában van, akkor más folyamat nem léphet be a (természetesen saját) kritikus szakaszába.). Tipikus kritikus szakasz a közös memória használata.

5. Tétel: A tárkezelés korszerű módszerei. Lapok, szegmensek kezelése. A virtuális tárkezelés alapjai.

A tárkezelés korszerű módszerei: Tárkezelés alatt azt a módot értjük, ahogyan a központi tárat a felhasználók (folyamatok) között megosztjuk.

Társzervezési módszerek:

- Egypartíciós rendszer
A nem védett területen felüli cím-tartományt csak egy folyamat használja.
– a program az első szabad címre töltődik.
- Többpartíciós rendszer
A multiprogramozás elve megköveteli, hogy több folyamat legyen egy időben a tárban.
- Lehetséges megoldások:

1. Fix partíciós rendszerek:

Az OS feletti tárterületet fix partíciókra bontják.

A határok nem változnak.

Rossz hatékonyság: belső tördelődés.

Védelem: határ-regiszterek.

2. Változó partíciós rendszerek:

A partíció a program igényeinek megfelelő méretű.

- Problémák: Szabad terület tördelődése.

– Egy folyamat lefutásakor a használt memória felszabadul.

– Az OS nyilvántartja ezeket a területeket, az egymás melletti szabad területeket automatikusan összevonja.

– Sokszor ezen területek nem szomszédosak, így a szabad memória kis részekre oszlik (külső tördelődés).

- Belső tördelődés nincs, hiszen csak a szükséges memóriát kapják meg a folyamatok

Szabad területek tömörítése (compaction, garbage collection):

- A külső tördelődés bizonyos fok után lehetetlenné teszi újabb folyamat elindítását (elég a szabad terület, de a leghosszabb összefüggő szabad terület nem elég a folyamatnak).

- Megoldás: a szabad helyek tömörítése.

Memóriaterületek lefoglalása:

Stratégiák a külső tördelődés csökkentésére.

Az OS a szabad területek közül a következőképpen választhat:

1. Legjobban megfelelő (best fit): legkisebb még elegendő méretű.

2. Első megfelelő (first fit): a kereséssel a tár elejétől indulunk, az első megfelelő méretűt lefoglaljuk.

3. Következő megfelelő (next fit): a kereséssel az utoljára lefoglalt tartomány végéről indulunk, az első megfelelő méretűt lefoglaljuk. Igen gyors algoritmus.

4. Legrosszabban illeszkedő (worst fit): A legnagyobb szabad területből foglalunk le, abban bízva, hogy a nagy darabból fennmaradó terület más folyamat számára még elegendő lesz.

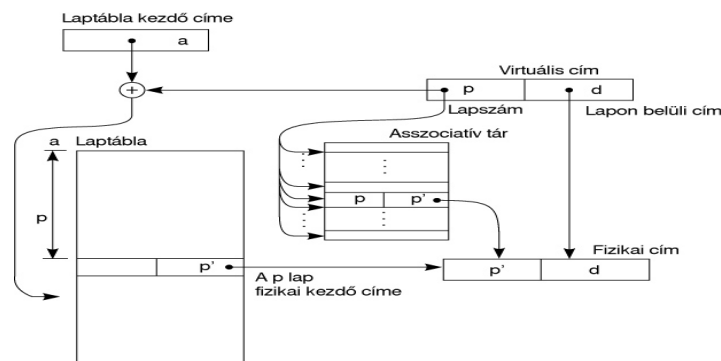
- Tárcsere (swap)

A folyamatoknak a háttértárról a memóriába való bevitelét, illetve a memóriából a háttértárra való kivitelét a **tárcsere (swapping)** technika segítségével oldhatjuk meg. Ennek során az operációs rendszer egy-egy folyamat teljes tárterületét a háttértárra másolja, így szabadítva fel a területet más folyamatok számára. Ehhez természetesen az operációs rendszernek pontosan ismernie kell a folyamatok aktuális tárigényét.

- Korszerű módszerek

- Szegmens szervezés
- Lap szervezés

Lapszervezés: A külső tördelődés megszüntetéséhez azonos, rögzített méretű blokkok és nekik megfelelő fizikai memória keretek (frame) használata szükséges, ennek neve lap (page), az ennek megfelelő társzervezésre pedig a lapszervezés (paging) elnevezést használjuk. A külső tördelődés megszüntetésének ára a belső tördelődés megjelenése. A lap mérete gyakorlati szempontok miatt mindig 2 hatványa.



Címtranszformáció:

- A 2 hatvány miatt az alsó biteken lehet tárolni az eltolást:

Címtranszformáció módszerei:

1. Közvetlen leképezés:

a.) Egyszintű laptábla: A folyamathoz tartozó minden lap fizikai címe egy táblában (laptérkép) van.

b.) Többszintű laptábla:

2. Asszociatív leképezés

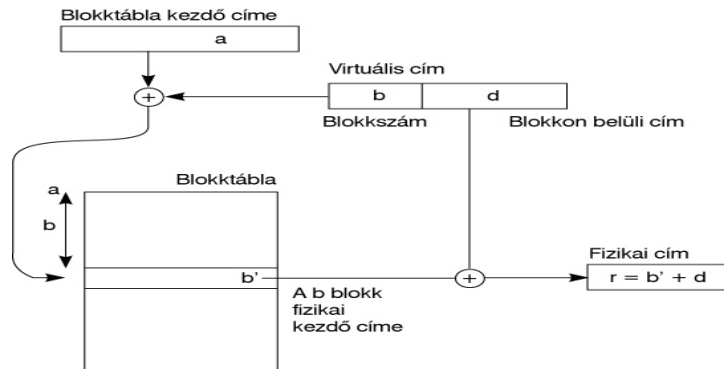
- Speciális gyors elérésű tár (asszociatív tár) segíti a címzést:
 - translation look-aside buffer (TLB).
- A laptábla gyorsító tárban a várhatóan gyakran használt lapok címét tároljuk.
- A tár mérete itt sem elég nagy.
- A gyakorlatban a kombinált technikák (laptábla + gyorsítótár) használhatók.

3. Kombinált technikák

- A fizikai lapcím keresése egyszerre kezdődik mind az asszociatív tárban, mind a direkt laptáblában.
- Ha az asszociatív tárban van találat, akkor a direkt keresés leáll.
- Az asszociatív tárban lévő lapokat frissíteni kell, környezetváltás esetén az asszociatív laptáblát is cserélni kell.

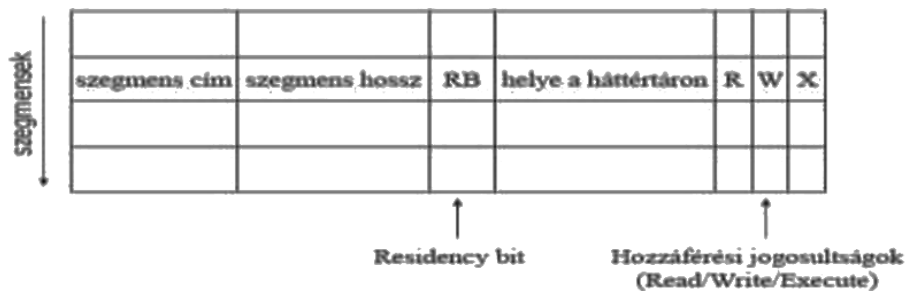
Szegmensszervezés: A logikai címtartományban a program memóriája nem egybefüggő terület, hanem önmagukban folytonos blokkok (szegmensek) halmaza.

- A szegmens logikai egység.
 - főprogram,
 - eljárások, függvények, módszerek, objektumok,
 - lokális változók, globális változók,
 - stack, szimbólumtábla, stb.



blokk tábla tárolja a szegmensek hosszát is, ezzel biztosítható, hogy a folyamat ne címezhesen ki a . A címképző hardver a szegmens területén kívülre mutató *túlcímzést* figyeli, és ha ilyen előfordul, hibamegszakítást generál (segment overflow fault).

Ahhoz, hogy a szegmentált címzési mód segítse a szegmensenkénti tárcserét is, a blokk tábla egy bitje (residency bit) azt mutatja, hogy a szegmens a memóriában van-e vagy sem. Ha olyan szegmensre történik hivatkozás, amelyik nincs a tárban, ennek a bitnek az állása alapján a címképző hardver hibamegszakítást generál (missing segment fault). Természetesen a tárcserék lebonyolításához tárolni kell azt az információt is, hogy a háttértáron hol található meg a szegmens.



A szegmensszervezés lehetőséget ad arra, hogy több folyamat egy-egy logikai szegmense ugyanarra a fizikai címtartományra legyen leképezve, vagyis, hogy a folyamatok közös eljárásai egyetlen példányban tárolódjanak a memóriában (**osztott szegmenshasználat, segment sharing**).

A szegmensszervezésnél használt változó méretű blokkok megoldást jelentenek a belső tördelődésre, azonban fellép a külső tördelődés jelensége.

Kombinált szegmens- és lapszervezés:

- Egyesíti a két technika előnyeit:
 - Lapszervezés: nincs külső tördelődés, nem kell a teljes szegmensnek a tárban lennie.
 - Szegmensszervezés: tükrözi a folyamat logikai társzerkezetét, hozzáférési jogosultság megoldható.
- Címtranszformáció: lényegében egy kétszintű táblakezelés.
 - Első szint: laptábla címeket tartalmazó szegmenstábla.
 - Második szint: szegmensenként egy-egy fizikai lapcímeket tartalmazó laptábla.
- A cím három részre tagolódik (szegmenscím, lapcím, lapon belüli eltolás).
- Hozzáférési jogok ellenőrzése a szegmens szervezésének megfelelően történik.
- Osztott tárhasználat a szegmens szervezésének megfelelően történik.

Virtuális tárkezelés: Olyan szervezési elvek, algoritmusok összessége, amelyek biztosítják, hogy a rendszer folyamatai logikai címtartományainak csak egy - a folyamat futásához szükséges - része legyen a központi tárban.

A virtuális tárkezelés általános elvei:

1.) A programok nem használják ki a teljes címtartományukat:

- tartalmaznak ritkán használt kódrészleteket (pl. hibakezelő rutinok).
- a statikus adatszerkezetek általában túlméretezettek (statikus vektorok táblák stb.).
- a program futásához egy időben nem kell minden részlet (overlay).
- időben egymáshoz közeli utasítások és adatok általában a térben is egymáshoz közel helyezkednek el (lokalitás).

2.) Nem célszerű az egész programot a tárban tartani:

- programok méretét nem korlátozza a tár nagysága, a program nagyobb lehet mint a ténylegesen meglévő memória mérete.
- a memóriában tartott folyamatok száma növelhető (nő multiprogramozás foka, javítható a CPU kihasználtság és az átbecsátó képesség).

Amikor a folyamat érvénytelen, a memóriában nem található címre hivatkozik, hardver megszakítást okoz, ezt az OS kezeli, és behozza a kívánt blokkot.

• Lépései:

- Az OS megszakítást kezelő programrésze kapja meg a vezérlést (1,2):
 - elmenti a folyamat környezetét.
 - elágazik a megfelelő kiszolgáló rutinra.
 - eldönti, hogy a megszakítás nem programhiba-e (pl. kicímzés).
- A kívánt blokk beolvasása a tárba (4,5):
 - Az OS a blokknak helyet keres a tárban; ha nincs szabad terület, akkor fel kell szabadítani egy megfelelő méretű címtartományt, ennek tartalmát esetleg a háttértárra mentve.
 - Beolvassa a kívánt blokkot:
- A folyamat újra végrehajtja a megszakított utasítást (6).

4.) Hardver feltételek:

- Az érvénytelen címhivatkozás megszakítást okozzon.

- A megszakított utasítás újraindítható legyen.

Az érvénytelen címhivatkozás kezelése:

- Emlékeztető: a laptábla mindig tartalmaz egy érvényességi bitet:

– valid: a lap a tárban van.

– invalid: a lap nincs a tárban.

5.) A betöltendő lap kiválasztása:

- **Igény szerinti lapozás (demand pages).**

Előnyei:

- egyszerű a lapot kiválasztani, a tárba csak a biztosan szükséges lapok kerülnek be

Hátrányai:

- Új lapokra való hivatkozás mindig laphibát okoz.

- **Előretekintő lapozás (anticipatory paging):**

- Az OS megpróbálja kitalálni, hogy a folyamatnak a jövőben melyik lapokra lesz szüksége és azokat "szabad idejében" betölti.

Lapcsere stratégiák:

- Akkor lenne **optimális**, ha azt a lapot választaná ki, amelyre legtovább nem lesz szükség.
- A lapcseret nagyban gyorsítja, ha a mentést csak akkor végezzük el, ha az a lap a betöltés óta módosult. A hardver minden lap mellett nyilvántartja, hogy a lapra írtak-e a betöltés óta (modified vagy dirty bit).
- Algoritmusok:
 - a.) Véletlen kiválasztás. b.) Legrégebbi lap (FIFO). c.) Újabb esély. d.) Óra algoritmus.
 - e.) Legrégebben nem használt lap. f.) Legkevésbé használt lap. g.) Mostanában nem használt lap.

6. Tétel: Háttértárak és kezelésük. Állományok kezelése. Az elosztott állománykezelés alapjai.

Háttértárak:

- Központi tár drága és kicsi a tárolókapacitása.
- A kikapcsolással az információk elvesznek a központi tárban.

Háttértárak típusai:

- a.) Mágneslemez (merev, hajlékony).
- b.) Mágnesszalag.
- c.) Magneto-optikai lemez.
- d.) Optikai lemez.
- e.) Flash.
- f.) Holografikus tárolás és MEMS tárolók.

Felépítése:

Mágneses bevonatú forgó korongok, a felület felett író-olvasó fej.

- Sáv (track) a lemezterület (gyűrű) azon része, amelyet a fej elmozdulás nélkül egy fordulat alatt elér.
- Cilinder (cylinder) az összes fej alatti sáv.
- Szektor (sector), a sáv azonos méretű blokkokra osztva.

Az információátvitel legkisebb egysége: a lemezvezérlő egy teljes szektort olvas vagy ír.

A lemezműveletek ütemezése:

- Egyszerre több folyamat verseng a háttértár perifériáért. Egyszerre több kérés várakozhat kiszolgálásra.

Cél az átlagos seek és a latency idő csökkentése.

- Természetesen így egyes folyamatok rosszabbul járnak, de a cél a globális teljesítmény növelése.

Értékelési szempontok:

- Átbocsátó képesség (időegység alatt lebonyolított átvitelek száma).
- Átlagos válaszidő (a kéréstől a végrehajtásig eltelt idő).
- Válaszidő szórása (megjósolható viselkedés, futás ne ingadozzon külső okok miatt).

Fejmozgás optimalizálása:

1.) Sorrendi kiszolgálás (FCFS)

A kérések sorrendjében történik a kiszolgálás.

- kicsi átbocsátó képesség.
- nagy az átlagos válaszidő.
- szórás viszonylag kicsi.

2.) Legrövidebb fejmozgási idő (SSTF)

Azt a kérést szolgálja ki, amely az aktuálshoz a legközelebbi cilinderen van.

FCFS-nél jobb.

- nagy a szórás
- kiéheztetés
- közepes átbocsátás
- kis átlagos válaszidő

3.) Pásztázó (SCAN)

- Az aktuális mozgási iránynak megfelelő kéréseket szolgálja ki, irányváltás, ha nincs több ilyen kérés. Közepes válaszidő. Nagy átbocsátás. Kis szórás.

- Sajátossága, hogy a középső cilindereket többször látogatja meg.

4.) N-lépéses pásztázó (N-SCAN)

Egy irányba mozogva N kérést (amelyek a pásztázás elején már megérkeztek) szolgál ki.

- A közben érkező kérésekre a következő irányváltás után kerül sor. Nagy átbocsátás. Kis válaszidő. Kis szórás (kisebb mint a SCAN).

5.) Körbeforgó pásztázó (C-SCAN)

Csak az egyik irányú fejmozgás során történik a kérések kiszolgálása.

- Ez is lehet N lépéses. Nagy átbocsátás. Kis válaszidő. Kis szórás.

6.) Kombinált módszerek

- A terhelés függvényében változtatja a stratégiákat:
 - alacsony terhelés SCAN.
 - közepes terhelés C-SCAN.
 - nagy terhelésnél C-SCAN és elfordulási idő optimalizálás.

A lemez blokkjainak allokációja:

1.) Folytonos terület allokációja:

- Az összetartozó információkat egymás melletti blokkokban tároljuk.

Első blokk sorszámát és a blokkok számát kell tárolni.

- Hátrányai:

- Külső tördelődés itt is fennáll.
A megoldáshoz itt is használhatók a már megismert algoritmusok (első illeszkedő, legjobban illeszkedő, legrosszabbul illeszkedő).

Nagyméretű tördelődés esetén tömöríteni kell.

- sokszor nem tudjuk előre, hogy hány blokkra lesz szükségünk.

- Előnyei:

- a tárolt információ soros és közvetlen elérése is lehetséges.
- jól használható tárcsere által kirakott szegmensekre (tudjuk előre a méretet).

2.) Láncolt tárolás:

- Blokkokat egyenként allokaljuk, minden blokkban fenntartva egy helyet a következő blokk sorszáma számára (a rendszer az első és az utolsó blokkot tárolja).
- Előnyei:
 - nincs külső tördelődés.
 - rugalmas, a lefoglalt terület növekedhet.
- Hátrányai:
 - csak soros elérés lehet.
 - a blokkok sorszámaival nő az állomány mérete.
 - sérülékeny, egyetlen láncszem hibája a tárolt információnak jelentős részének elvesztését jelenti.
- Módosított változata állomány allokációs tábla (file allocation table, FAT).
 - a lánc elemeket az állományoktól elkülönítve tároljuk.
 - a szabad helyek tárolására is alkalmas a FAT.

3.) Indexelt tárolás:

- Az állományhoz tartozó blokkok címei egy indextáblában vannak.
- Előnyei:
 - közvetlen hozzáférés.
 - "lyukas" állományok tárolása (nem minden blokk tartalmaz valós információt).
- Hátrányai:
 - az indextábla tárolása legalább egy blokkot elfoglal (kis állományok esetében pazarló).
 - az indextábla mérete nem ismert, lehetővé kell tenni, hogy az növekedhessen:
- Láncolt indexblokkok.
- Többszintű indextábla.
- Kombinált mód, kis állományok esetében egy, majd többszintű indextábla.

4.) Kombinált módszerek:

- Hozzáférési módja szerint: Soros hozzáférés esetén láncolt, közvetlen hozzáférés esetén indexelt.

Állomány (file):

- A létrehozó által összetartozónak ítélt információk gyűjteménye.
- egyedi azonosító (név) különbözteti meg őket.
- Az állomány elrejtje a tárolásának, kezelésének fizikai részleteit:
 - melyik fizikai eszközön található (logikai eszköznevek használata).
 - a perifériás illesztő tulajdonságai.
 - az állományhoz tartozó információk elhelyezkedése a lemezen.
 - az állományban lévő információk hogyan helyezkednek el a fizikai egységen (szektor, blokk).

Könyvtár:

- Az állományok csoportosítása OS és a felhasználó szerint.
- A könyvtár tartalmát katalógus írja le.

Az állománykezelő feladatai:

- Információátvitel (állomány és a folyamatok között).
- Műveletek (állományokon, könyvtárakon).
- Osztott állománykezelés.
- Hozzáférés szabályozása:
 - más felhasználók által végezhető műveletek korlátozása (access control).
 - tárolt információk védelme az illetéktelen olvasások ellen (rejtjelezés, encryption).
 - információ védelme a sérülések ellen, mentés.

Az állományrendszer réteges implementációja:

1.) A perifériát közvetlenül kezelő periféria meghajtó:

Feladata a tár és a periféria közötti átvitel megvalósítása. (device driver, átvitelt kezdeményező, megszakítást kezelő eljárások)

2.) Elemi átviteli műveletek rétege: A lineáris címzésű blokkok átvitele, átmeneti tárolása.

3.) Állományszervezés rétege: Háttértár szabad blokkjainak, illetve az állományhoz tartozó blokkoknak szervezése.

4.) logikai állományszervezés: Kezeli a nyilvántartások szerkezetét, azonosító alapján megtalálja az állományt, szabályozza az állományszintű átvitelt.

Állományok tárolása a lemezen:

- Lemezterületet blokkonként (néhány szektor) kezeli.
Mérete a lapmérethez hasonló megfontolások alapján.
- Az állomány tárolásához blokkok kellenek, ezeket nyomon kell követni.
 - Szabad blokkok nyilvántartása.
 - Blokkok allokálása.

Szabad blokkok nyilvántartása:

a.) Bittérkép:

- Minden blokkra egy biten jelzi, hogy szabad-e.
- A bittérkép a lemez kijelölt helyén van.
- A bitvektort a memóriában kell tárolni.

Sokk blokk van, így sok memóriát igényel.

b.) Láncolt lista:

- Az első szabad blokk címének tárolása, arra felfűzve a többi.
- A blokk területéből vesszük le a cím területét.
- Nem hatékony, lassú (sok lemezművelet).

c.) Szabad helyek csoportjainak listája:

- Láncolt lista javítása.
- Minden blokk n db (ennyi cím fér el a blokkban) szabad blokkra hivatkozik.
- $n - 1$ ténylegesen szabad, az n . a lista új elemére mutat.

d.) Egybefüggő szabad terület tárolása:

- Egy táblázatban tároljuk az összefüggő szabad blokkokat (első blokk sorszáma, blokkok száma).

Osztott állománykezelés:

- Ez is egy erőforrás, amelyet egyidejűleg több folyamat is használni akar.
- Csak olvasás esetén osztottan gond nélkül használható.
- Írás esetén kölcsönös kizárással kell védeni a folyamatot:
 - Egész állományra vonatkozó szabályozás (file lock):
- Az állományt először megnyitó folyamat definiálja, hogy a későbbi megnyitási kérelmekből mit engedélyezhetünk.
 - kizárólagos használat.
 - többi folyamat csak olvashatja.
 - több folyamat is megnyithatja írási joggal.
- Írás esetén az olvasó folyamatok mikor veszik észre a változást:
 - azonnal.
 - ha a folyamat lezárta az állományt.
 - Állomány részeire vonatkozó kizárás:
- Rekordonként lehet kizárásokat definiálni.

Elosztott állománykezelés:

- A helyi operációs rendszer állománykezelési szolgáltatásainak kiterjesztése egymással kommunikációs csatornán keresztül kapcsolódó számítógépek halmazára.
- Az állomány lehet helyi (local) illetve távoli (remote), ideális esetben az állomány tényleges elhelyezkedése a felhasználó előtt rejtve van.
- Az állományt tároló számítógép a szolgáltató (szerver) az ügyfelek (kliens) számára szolgáltatásként műveleteket biztosít az állományain.
- Az osztott állománykezelés lassabb (állományok megtalálása, állományok átvitele).

Az állományokra hivatkozás:

- A név egyedileg azonosítja az állományt.
- A felhasználó előtt elrejt a tárolás részleteit (melyik gépen van stb.).

Az állományok nevei:

 - Két szintet különböztetünk meg:
 - felhasználói szintű neveket.
 - rendszerszintű neveket.
- Az állománykezelő feladata a kétszintű azonosító egymáshoz rendelése.

Műveletek végzése:

- A felhasználó a helyi állománykezelésnek megfelelő műveleteket akarja elvégezni az állományon (miután a rendszer megtalálta a hivatkozott állományt).
- Műveletek módjai:

Távoli eljáráshívás (RPC):

- A felhasználói műveletek kérésként jelennek meg a szolgáltató csomópontnál.
- Minden művelethez tartozik egy távoli eljárás. Az ügyfél határozza meg az eljárás paramétereit, a szolgáltató válasza az eljárás visszatérési értéke.
- A szolgáltatói oldalon minden távoli eljáráshoz tartozik egy speciális démon folyamat (daemon), feladata a kliensek felől érkező kérések kiszolgálása.
- A folyamat egy hozzá rendelt kaput figyel, az azon érkező kérést végrehajtja, majd a választ a kérésből megállapítható feladónak küldi vissza.
- A kliens folyamat és a démon között aszimmetrikus kommunikáció van (az ügyfélnek meg kell neveznie a kaput, és ezzel a hozzá tartozó démon).
- Problémái:
 - Hálózati kommunikáció nem megbízható:
 - fontos, hogy minden kérés pontosan egyszer hajtsódjon végre.
 - a kéréseket a kliensek sorszámozzák, időbélyeget fűznek hozzá (time stamp), az egyszer már kiszolgált sorszámú kérést a szerver figyelmen kívül hagyja (ignorálja).
 - Kliens oldalon a távoli eljárásokat a megfelelő kapukra kell képezni, itt a megfelelő démon várakozik. A **leképzés** lehet:
 - **statikus**: minden művelethez előre kijelölt kapu tartozik.
 - **dinamikus**: műveletenként a kapu számát egy kötött kapuban lévő démon szolgáltatja (házasságközvetítő, matchmaker).

Helyi átmeneti tárok segítségével:

- A teljesítmény növelése, hálózati adatforgalom csökkentése érdekében a helyi gépek a szükséges adatállományokat átmenetileg tárolják, a műveleteket azon végzik.
- Az elv azonos a virtuális tárkezelésben alkalmazott átmeneti tárkezeléssel (caching):
 - ha szükséges, az új információknak helyet csinálunk.
 - a szükséges információt a hálózaton keresztül az átmeneti tárba töltjük.
 - a műveleteket a helyi másolaton végezzük el.
 - változás esetén az információkat visszaírjuk.
- Problémák:

1.) Az átviteli egység meghatározása:

- Rendelkezésre álló átmeneti tár mérete.
- Az alapszintű hálózati protokoll (megengedett blokkméret), illetve az RPC-ben (Remote Procedure Calling) megengedett blokkméret.

2.) Az átmeneti tárolás helye:

- Helyi gép központi tárában: gyors és háttértár nélkül is alkalmazható.
- Helyi gép háttértártárában: megbízható és teljes állományokat is tartalmazhat.

3.) A változások érvényre juttatása (update):

- A módosításokat a szolgáltatóval közölni kell:
 - azonnal: lassú és biztonságos.
 - késleltetve (delayed write): gyors és nem biztonságos.
- Visszaírás történhet:
 - ha szükség van helyre az átmeneti tárban.
 - időközönként.
 - az állomány lezárásakor.

4.) Az átmeneti tár konzisztenciája:

- Egy állományt több kliens is használhatja, módosítás után az állomány már nem aktuális.
- Aktualizálni kell:
 - A kliens kérésére (az ügyfél gyanakszik, kéri az ellenőrzést, hogy a saját példánya helyes-e):
 - minden hozzáférésnél.
 - az állomány újra megnyitásakor.
 - időközönként.
 - A szolgáltató kezdeményezésére:
 - a szolgáltató nyilvántartja az ügyfelek helyi tárolásait (speciális üzenetek a kliensektől).
 - ha az állományokban olyan változás van, ami inkonzisztenciát okoz, értesíti a klienseket.
 - értesítés (consistency semantics):
 - azonnal, állomány lezárásakor.
 - előrelátható problémák esetében üzenhet a kliensnek, hogy ne használjon helyi tárolást, inkább használja a távoli szolgáltatásokat.

7. Tétel: A fizikai és az adatkapcsolati réteg jellemzése, legfontosabb feladatai (átviteli közegek, keretezési eljárások, hibajelzés és hibajavítás, elemi és csúszóablakos protokollok), gyakorlati példák (PPP, HDLC)

Vezetékes átviteli közegek:

- Sodort érpár (twisted pair):
 - A legrégebben használt és még ma is a legelterjedtebb átviteli közeg.
 - Két szigetelt részhuzalból áll, melyek megközelítőleg 1mm vastagságúak. A huzalok spirálszerűen egymás köré vannak tekerve, ezzel csökkentve a kettő közötti elektromágneses kölcsönhatást.
 - Alkalmas analóg és digitális jelátvitelre egyaránt. Sáv szélessége a vezeték vastagságától és az áthidalt távolságtól függ. Sok esetben néhány mbps sebesség is elérhető néhány km-es távolságon belül.
 - A 6-os és 7-es kategóriájú verzió, melyeket napjainkban használnak 250 és 600 MHz-es sáv szélességen képesek kezelni a jeleket.
 - Legelterjedtebb elnevezés: UTP (Unshielded Twisted Pair)
- Koaxális kábel:
 - A sodort érpárnál nagyobb árnyékolással rendelkezik, ezért nagyobb sebességgel nagyobb távolságot lehet vele áthidalni.
 - Két fajtája:
 - 50Ω-os kábel: főleg digitális átvitelhez
 - 75Ω-os kábel: elsősorban analóg átvitelhez
 - A kábel közepén egy tömör rézhuzal van, amelyet szigetelő rész vesz körül. A szigetelő körül sűrű szövésű hálóból álló vezető található. A külső vezetőt mechanikai védelmet is biztosító műanyaggal vonják be.
 - Sáv szélessége közel 1 GHz.
- Fényvezető szálak:
 - Az átvitel három komponense: fényforrás, átviteli közeg és detektor.
 - Az átviteli közeg egy rendkívül vékony üvegszál. Ha a detektorba fény jut, akkor villamos jelet állít elő.
 - Az üvegszál átmérőjét lecsökkentve néhány fényhullámhossznyiára lecsökkentve az üvegszál hullámvezetőként viselkedik és a fény visszaverődés nélkül, egyenes vonal mentén terjed a vezetéken. Az ilyen üvegszálat egymódusú szálnak nevezik. Az egymódusú szálak másodpercenként 50 *gigabitet* képesek 100 km-re továbbítani erősítés nélkül.
 - A fényerősség csökkenését az üvegben a fény hullámhossza határozza meg. A decibelben mért csillapítást az alábbi módon számíthatjuk ki:
$$\text{Csillapítás decibelben} = 10 \log_{10} \frac{\text{Kibocsátott teljesítmény}}{\text{Vett teljesítmény}}$$
 - A szálon végigküldött fényimpulzusok hosszanti irányban szétszóródnak terjedés közben. Ezt a szóródást kromatikus diszperciónak nevezik. Mértéke a fény hullámhosszától függ. A szétszóródott impulzusok átfedésének megakadályozásának egyik módja, hogy növeljük a közöttük hagyott távolságot, de ezt csak a jelzési sebesség csökkentésével lehet elérni. Egy másik módszer az impulzusok egy bizonyos alakra formálása, melynek segítségével szinte minden szóródási hatást kiejthetünk. Ezek az impulzusok az úgynevezett szolitonok.
 - A kábel szerkezete a fonott árnyékolástól eltekítve hasonlít a koaxális kábelre. Középen található egy üveg mag, amiben a fény terjed. Többmódusú szál esetén a mag 50 *mikron* átmérőjű, azaz körülbelül olyan vastag, mint egy emberi hajszál. Egymódusú szál esetén a mag 8 – 10 *mikron* átmérőjű. Az üvegmagot olyan törésmutatójú üveggöpeny veszi körül, amelynek törésmutatója kisebb, mint a magé, így a fénysugár a magon belül marad. A szálat kívülről műanyaggal burkolattal látják el.

Programtervező informatikus BSc államvizsga tételsor
Informatika tárgycsoport

- A fényimpulzusok előállítására kétféle fényforrást használnak: az egyik a LED a másika félvezető lézer. Ezek főbb tulajdonságai:

Jellemző	LED	Félvezető lézer
Adatátviteli sebesség	Alacsony	Magas
Módus	Többmódusú	Több- vagy egymódusú
Távolság	Kicsi	Nagy
Élettartam	Hosszú	Rövid
Hőmérsékletérzékenység	Kicsi	Jelentős
Ár	Olcsó	Drága

Vezeték nélküli adatátvitel:

- Rádiófrekvenciás átvitel
 - Egyszerűen előállítható hullámok. Nagy távolságra jutnak el, minden irányba terjednek, így nincs szükség pontos illesztésre.
 - Mivel nagy távolságra jut el, ezért gondot okozhat a felhasználók között interferencia.
 - Frekvenciától függően előfordulhat, hogy nagy frekvencia esetén a hullámok elnyelődnek az esőben.
 - A VLF, LF, MF frekvenciasávokban a rádióhullámok a földfelszínt követik. HF és VHF sávokban a földközeli hullámokat a föld kezdi elnyelni, azok a hullámok azonban amik eljutna az ionoszféráig visszaverődnek.
- Mikrohullámú átvitel
 - 100 MHz feletti hullámokat használnak, mivel ezek már majdnem egyenes vonalban terjednek, ezért jól fókuszálhatóak.
 - Mivel egyenes vonalban terjednek a hullámok a földfelszín görbülete problémát jelent, ha az adótoronyok túl messze vannak egymástól.
 - A mikrohullámok nem képesek áthaladni az épületek falain és bizonyos mennyiségben a levegőben is szóródnak.
 - Megtörténhet, hogy a hullámok egy része megtörik alacsonyabb légköri rétegenél és később ér célba mint a közvetlen beérkező hullámok. A megtört hullámok fázisa nem egyezik meg a közvetlen beérkező hullámmal, így ezek akár ki is olthatják egymást. Ez a jelenség a többutas jelgyengülés (multipath fading).
 - Körülbelül 4 GHz-nél fellép az a probléma, hogy a víz elnyeli a hullámokat. Ebben az esetben a legtöbb szolgáltató leállítja ezeknek a vonalaknak a használatát és megpróbálja a forgalmat más irányba terelni.
 - A technika előnye, hogy a fényvezető kábeles megoldásokkal szemben nagyon olcsó, mert nem kell kábeleket fektetni, hanem elegendő csupán néhány adótorony.
- Infravörös és miliméteres hullámú átvitel:
 - Elsősorban kistávolságú adatátvitelre használják (pl.: távirányítók).
 - Az infravörös hullám jól irányítható, olcsó és könnyen előállítható. Nem hatol át a falakon, így nem zavarja a szomszéd szobákban levő infravörös rendszereket.

Keretezési eljárások:

- Karakterszámlálás: A keretben levő karakterek számának megadása a keret fejrészének egy mezőjében. Átviteli hiba esetén ha megsérül a karakterszám mező, akkor a küldő és a fogadó kiesnek a szinkronból.
- Kezdő és végkarakterek karakterbeszúrással: Az újr szinkronizálás problémájának megoldása képpen minden keret elejét és végét egy-egy különleges bájtal jelzi. Ha a vevő kiesik a szinkronból elegendő a jelzőbájtot megkeresnie. Előfordulhat azonban, hogy ezek a bájtok megjelennek az átvinni kívánt adatok között is. Ebben az esetben az adatkapcsolati réteg egy különleges bájtot szűr be minden olyan jelző bájt elé ami véletlenül került szövegbe.

- Kezdő és végkarakterek bitbeszúrással: Minden keret egy speciális jelző bájtának nevezett bitmintával kezdődik. Ez a 01111110. Amikor az adó adatkapcsolati rétege öt egymást követő 1-est talál az adatok között, automatikusan beszúr egy 0-t a kimenő bitfolyamba. Ez a bitbeszúrással analóg a karakterbeszúrással.
- Fizikai réteg belső kódolássértés: A LAN egy adatbitet két fizikai szinten kódol: az 1-es bit fizikai magas-alacsony pár, a 0-s pedig egy alacsony-magas. A magas-magas és az alacsony-alacsony nem használatos adatbitek kódolására. Ezeket lehet a kerethatárok kódolására használni.

Hibajelzés és hibajavítás: Az átviteli közeg tulajdonságaitól függően használunk hibajelző és hibajavító kódokat. Olyan csatornák esetében, amelyek nagy megbízhatóságúak olcsóbb megoldás csupán hibajelző kódokat használni és a hibás adatokat újraküldeni, mintsem több redundáns adatot felhasználni hibajavításra ezzel növelve az adatforgalmat. Olyan közegek esetében melyek kevésbé megbízhatóak (például a vezeték nélküli átvitel) szükséges a hibajavító kódok használata.

Hamming távolság: Azt mutatja meg, hogy egy m bites kódszóban, hogy egy bites hibának kell történnie, hogy a kódszó egy másik érvényes kódszóba menjen át. Az, hogy egy kódszó hibajelző vagy hibajavító tulajdonságú-e, a kód Hamming-távolságától függ. Ahhoz, hogy d hibát jelezni tudjunk, $d + 1$ Hamming-távolságú kód kell, mert egy ilyen kódszóban d bithiba nem tudja a kódszót egy másik érvényes kódszóba vinni. Amikor a vevő egy érvénytelen kódszót lát tudja, hogy átviteli hiba történt. Hasonlóan: ahhoz, hogy d hibát ki tudjunk javítani, $2d + 1$ Hamming-távolságú kód kell, mert így az érvényes kódszavak olyan távol vannak, hogy még d bit megváltozásakor is közelebb lesz az eredeti kódszó a hibáshoz, mint bármely másik érvényes, így az egyértelműen meghatározható.

Paritás bit: A paritásbitet úgy választjuk meg, hogy a kódszóban levő 1-ek száma páros (vagy páratlan) legyen. Egy paritásbites kód Hamming-távolsága 2 hiszen bármilyen 1 bites hiba rossz paritású kódszót eredményez. Ezért az ilyen kód 1 bitnyi hiba jelzésére alkalmas.

Ciklikus redundancia kód (CRC): A kód azon alapul, hogy a bitsorozatokat polinomok reprezentációjának tekintjük, melyekben csupán 0 és 1 együtthatók szerepelnek. Egy k bites keret tekintsünk egy k tagú polinom együtthatóinak x^{k-1} -től x^0 -ig. Az ilyen polinomot $k - 1$ -ed fokúnak nevezzük. A legnagyobb helyiértékű (bal szélső) bit az x^{k-1} együtthatója; a következő bit az x^{k-2} -é és így tovább. A polinom aritmetika modulo 2 végzendő az algebrai terek elmélete szerint. Nincs átvitel az összeadásnál és a kivánsnál, melyek a kizáró vagy mávelettel azonosak.

Polinomkód alkalmazása során a vevőnek és az adónak előre meg kell egyeznie egy generátor polinomban. Jelöljük ezt $G(x)$ -szel. A generátor legalsó és legfelső bitjének 1-nek kell lennie. Ahhoz, hogy egy $M(x)$ polinomnak megfelelő m bites keret ellenőrző összegét kiszámíthassuk, a keretnek hosszabbnak kell lennie, mint a generátor polinom. Az ötlet, hogy úgy fűzzünk ellenőrző összeget a kerethez, hogy az így kapott keret által reprezentált polinom osztható legyen $G(x)$ -el. Ha van maradék, akkor hiba volt az átvitel során.

Az ellenőrző összeg kiszámításának algoritmus a következő:

1. Legyen r $G(x)$ foka. Fűzzünk r darab 0 bitet a keret alacsony helyi értékű végéhez, így az $m + r$ bitet fog tartalmazni és az $x^r M(x)$ polinomot fogja reprezentálni.
2. Osszuk el az $x^r M(x)$ -hez tartozó bitsorozatot a $G(x)$ -hez tartozó bitsorozattal modulo 2.
3. Vonjuk ki a maradékot (mely mindig r vagy kevesebb bitet tartalmaz) az $x^r M(x)$ -hez tartozó bitsorozatból modulo 2-es kivonással. Az eredmény az ellenőrző összeggel ellátott, továbbítandó keret. Nevezzük ennek a polinomját $T(x)$ -nek.

Adatátviteli protokollok:

- Korlátozás nélküli szimplex protokoll:
 - A vevő és az adó hálózati rétegei mindig készen állnak.
 - A feldolgozási időtől eltekintünk.
 - A puffer terület végtelen.
 - A kommunikációs csatorna sohasem rontja vagy veszíti el a csomagokat.
 - Nem szükséges nyugtázás.
 - A küldő egy végtelen while ciklus ami folyamatosan pumpálja kifelé a vonalra az adatokat, ahogy csak tudja. A ciklusmag három teendője: szerezni egy csomagot, összerakni egy keretet, útnak indítani a keretet.
 - A vevő kezdetben vár, hogy valami történjen. Az egyetlen lehetőség sértetlen keret érkezése. Megérkezik a keret, eltávolítódik a hardver pufferből és bekerül egy változóba. Végül az adatrészt tovább adjuk a hálózati rétegnek.
- Szimplex megállás-vár protokoll:
 - Nincs végtelenül gyors feldolgozás és végtelen puffer.
 - A kommunikációs csatornát továbbra is hibamentesnek feltételezzük, és az adatforgalom egyirányú.
 - Probléma: meg kell akadályozni az adót, hogy gyorsabban adjon, mint ahogy a vevő fel tudja dolgozni a vett adatokat.
 - A vevő visszacsatolást biztosít az adónak, azaz miután a vevő átadta a csomagot a hálózati rétegnek, visszaküld egy kis álkereket, amely valójában engedély a küldő állomásnak arra, hogy továbbítsa a következő keretet.
 - Miután a küldő állomás egy keretet elküldött várakoznia kell amíg a nyugtakeret meg nem érkezik.
 - Az adatáremlés ebben a protokollban csak egyirányú, keretek azonban mindkét irányba utaznak.
- Szimplex protokoll zajos csatornához:
 - A kommunikációs csatorna hibázhat, a keretek megsérülhetnek.
 - Az adó egy sorszámot tesz minden keret fejlécébe, így a vevő megállapíthatja, hogy újonnan érkezett vagy megkettőződött keretről van-e szó.
 - 1 bites sorszámmező alkalmazása elegendő.
 - Az adó állomás a keret továbbítása után elindít egy időzítőt. Ha az időzítő már ment, akkor nullázódik. Az időzítési intervallumnak elegendőnek kell lennie arra, hogy a keretnek legyen ideje eljutni a vevőhöz, a vevőnek feldolgozni azt, majd a nyugta keretnek visszaérkeznie.
 - Ha az időzítő lejárt feltételezhetjük, hogy a keret vagy annak nyugtája elveszett és ekkor küldhetjük egy újat belőle.
 - A keret továbbítása és az időzítő elindítása után a küldő várakozni kezd. Három esemény következhet be: nyugtakeret érkezik sértetlenül, sérült nyugtakeret érkezik be vagy az időzítő lejár.
 - Érvényes nyugta esetén az adó lekéri a soronkövetkező csomagot a hálózati rétegtől és beleteszi a pufferbe, fülülva az előző csomagot és lépteti a számlálót. Ha sérült keret vagy semmi sem érkezik akkor sem a puffer sem a sorszám nem változik így a következő ciklusban egy újabb példányt küldünk el belőle.
 - Érvényes keret érkezése esetén a vevő megvizsgálja a keret sorszámát, hogy lássa nem duplikátum-e. Ha nem az elfogadjuk és továbbítjuk a hálózati rétegnek, majd nyugtát küldünk. A duplikátumokat és sérült kereteket nem adjuk tovább.

- Egybites csúszóablakos protokoll:
 - Az ablak nagysága maximum 1.
 - Ez a protokoll a megáll-és-vár technikát alkalmazza, mivel a küldő állomás küld egy keretet és megvárja ennek nyugtáját, mielőtt a következőt elküldené.
 - A vevő és a küldő számoltatja, hogy melyik keretet kell küldenie / fogadnia. A korábbiakhoz hasonlóan 1 bites sorszámok felhasználásával.
 - A kezdő gép lekéri az első csomagot a hálózati rétegtől, egy keretet épít belőle és elküldi azt.
 - Amikor egy keret megérkezik a vevő adatkapcsolati rétege megnézi, hogy duplikátum-e. Ha az a keret amelyiket várta, átadja a hálózati rétegnek, és a vevő ablakát feljebb csúsztatja.
 - A nyugta mező a legutolsó hibátlanul vett keret sorszámát tartalmazza. Ha ez a szám megegyezik annak a keretnek a sorszámával, amit az adó próbált küldeni, adó tudja, hogy végzett a buffer-ben tárolt csomaggal, és lekérheti a következőt a hálózati rétegtől. Ha a sorszám nem egyezik folytatnia kell a próbálkozást ugyanazzal a kerettel.
- n visszalépést alkalmazó protokoll:
 - Megengedjük a küldőnek hogy 1 keret helyett w darab keretet elküldjön, mielőtt blokkolódik, így w megfelelő megválasztásával az adó állomás folyamatosan küldeni tudja a kereteket a körülfordulási idővel azonos ideig, anélkül, hogy betelne az ablak.
 - A küldő oldalon akkor van szükség nagyméretű ablakra, ha az adatsebesség és az oda-vissza út késleltetésének szorzata nagy. Ha az adatsebesség nagy, akkor az adó még mérsékelt késleltetés mellett is gyorsan kimeríti ablakát, kivéve abban az esetben ha nagy ablaka van. Nagy késleltetés esetén a küldő még kis sebesség mellett is gyorsan kimeríti ablakát (pl.: geoszinkron műholdak).
 - Ha a csatorna kapacitás b bps, a kerethossz l bit és a körülfordulási idő R s, egyetlen keret továbbításához szükséges idő $\frac{l}{b}$ s. Miután az adatkeret utolsó bitjét is elküldtük, az $\frac{R}{2}$ s múlva ér oda a vevőhöz, és még legalább $\frac{R}{2}$ s kell a nyugtának, hogy visszaérjen az adóhoz.
 - A vonal kihasználtsága $= \frac{l}{l + bR}$.
 - A protokoll lényege (visszalépés n-nel): A vevő eldobja az összes keretet, amely a hibás után érkezik meg, és nem küld róluk nyugtát.
- Szelektív ismétlést alkalmazó protokoll:
 - A vevő a rosszul vett kereteket eldobja, de az ezután érkező kereteket tárolja egy ablakban. Egy keret beérkezésekor a vevő ellenőrzi, hogy az adott sorszámú keretet már vette-e vagy újonnan érkezett. Ha új akkor eltárolja az ablakban, amennyiben beleesik a fogadott keret sorszáma a várt sorszámok tartományába.
 - Ezt a technikát gyakran alkalmazzák együtt a negatív nyugtázással, amikor a vevő abban az esetben küld nyugtát ha hibát észlel.

PPP: Kezeli a hibák felderítését, több protokollt is támogat, lehetővé teszi, hogy az IP-címekről a felek az összeköttetések kiépítésekor egyezkedjenek, megengedi a hitelesítést és még sok más lehetőséget is tartalmaz.

Három dolgot biztosít:

1. Olyan keretezési módszert, amely egyértelműen ábrázolja a keret végét és a következő keret elejét. A keretformátum megoldja a hibajelzést is.
2. Kapcsolatvezérlő protokollt a vonalak felélesztésére, tesztelésére, az opciók megbeszélésére és a vonalak elegáns elengedésére, amikor már nincs rájuk szükség. Ezt a protokollt LCP-nek (Link Control Protocol) nevezik. Támogatja a szinkron és asszinkron áramköröket, valamint a bájt és bit alapú kódolásokat.
3. Olyan módot a hálózatréteg-opciók megbeszélésére, amely független az alkalmazott hálózatréteg-protokolltól. A választott módszer az, hogy különböző NCP (Network Control Protocol) van mindegyik hálózati réteghez.

A PPP többprotokollós keretezési eljárás, amely aklamas modem, HDLC bitsoros vonal, SONET és más fizikai rétegek feletti használatra. Támogatja a hibajelzést, az opciókban való megegyezést, a fejléctömörítést és opcionálisan a megbízható átvitelt HDLC keretezéssel.

HDLC:

- Bit alapú és bitbeszúrást alkalmaz a kódfüggetlenség érdekében.
- CRC-CCITT-t alkalmaz generátor polinomként az ellenőrző összeg mező tartalmának előállításához.
- A kereteket egy jelző sorozat (01111110) határolja. A tétlen kétpontos vonalakon folyamatosan továbbítjuk a jelző sorozatokat.
- Három féle keret van:
 - Információs
 - Felügyelő:
 - Nyugtakeret (receive ready)
 - Elutasítás (reject)
 - Vételre nem kész (receive not ready)
 - Szelektív elutasítás (selective reject)
 - Számozatlan:
 - Néha vezérlési célokra használják, de használható adatok átvitelére is, ha megbízhatatlan összeköttetés nélküli szolgálatra van szükség.
- Csúszóablakot tartalmaz 3 bites sorszámmal, tehát bármely pillanatban legfeljebb 7 nyugtázatlan keret lehet kint.
- A protokoll felhasználja az úgynevezett ráültetett nyugtákat (piggybacking).

8. Tétel: A hálózati réteg jellemzése, legfontosabb feladatai (forgalomirányító algoritmusok, torlódáskezelés), gyakorlati példák (IP)

Feladata: A csomagok eljuttatása a forrástól egészen a célállomásig.

Közege: Az alhálózat, vagyis forgalomirányítók (routerek) halmaza.

Követelmények:

- A hálózati rétegnek ismernie kell az alhálózat topológiáját
- Megfelelő útvonalakat kell találnia az alhálózaton keresztül a célállomásig.
- Ügyelnie kell arra, hogy egyenlő arányban terhelje a forgalomirányítókat.
- Képesnek kell lennie két különböző hálózat közötti kapcsolatteremtésre.

A szállítási rétegnek nyújtott szolgálatok:

- Összeköttetés nélküli (datagram alhálózat): A csomagok egyenként, egymástól függetlenül kerülnek továbbításra, akár különböző útvonalakon.
- Összeköttetés alapú (virtuális áramkör alhálózat): A forrás és a cél forgalomirányítók között előre kiépített útvonal (virtuális áramkör) jön létre, mielőtt egyetlen adatcsomagot is elküldenénk. Ha a kommunikációt befejezték a kapcsolat lebontódik.

A két szolgáltatás összehasonlítása:

A forgalomirányítók memória területe \leftrightarrow sávszélesség:

Összeköttetés nélküli kapcsolat esetén minden csomagnak tartalmaznia kell a teljes célcímet, ami jelentős sávszélességet foglalhat, ha az adatcsomagok amúgy általában elég rövidek.

Összeköttetés alapú kapcsolat esetén az adatcsomagoknak csak áramkörszámokat kell tartalmazniuk, viszont ekkor a forgalomirányítókban belül egy plusz táblázatot kell eltárolni (a virtuális áramkör címeikkel).

Az összeköttetés-felépítési idő \leftrightarrow címfeldolgozási idő:

Az összeköttetés felépítése időbe kerül és erőforrást igényel, viszont ez után könnyű eldönteni, hogy mi történje egy beérkező csomaggal.

Datagram alapú alhálózat esetén, bonyolultabb eljárás szükséges a továbbítás helyének meghatározására (forgalomirányító algoritmusok).

Szolgáltatminőség, torlódásvédelem:

A virtuális áramkörök esetében garantált a szolgáltatás minősége és a torlódások elleni védelem, mivel az erőforrásokat a kapcsolat felépítése fázisban lefoglalják, mielőtt egyetlen adatcsomag továbbításra került volna.

Datagram alhálózatnál bonyolultabb kérdés a torlódás védelem (torlódásvédelmi algoritmusok)

Forgalomirányító algoritmusok: A hálózati réteg szoftverének azon része, amely azért a döntésért felelős, hogy egy bejövő csomag melyik kimeneti vonalon kerüljön továbbításra.

Ha az alhálózat belül datagramokat használ, akkor ezt a döntést újra meg újra meg kell hozni. Ha az alhálózat virtuális áramköröket használ, akkor a forgalomirányítási döntéseket csak új virtuális áramkör felépítéskor kell meghozni. (viszony-forgalomirányítás)

Egy forgalomirányító két külön folyamatot végez: forgalomirányítást és továbbítást.

Továbbítás: Egy beérkező csomag továbbítása a tárolt táblázatok alapján.

Forgalomirányítás: A forgalomirányító táblázatok feltöltése és karbantartása.

Elvárt tulajdonságok: helyesség, egyszerűség, robusztusság, stabilitás, igazságosság, optimalitás és hatékonyság.

Két nagy osztályba sorolhatóak: adaptív és nem adaptív algoritmusok.

Nem adaptív algoritmusok: Nem támaszkodnak döntéseikben mérésekre vagy becslésekre az aktuális forgalomról és topológiáról. Statikus forgalomirányítás.

Adaptív algoritmusok: Úgy változtatják döntéseiket, hogy tükrözzék a topológiában és rendszerint a forgalomban bekövetkezett változásokat. Dinamikus forgalomirányítás.

Az optimalitási elv: Ha A router a B routertől C router felé vezető optimális útvonalon helyezkedik el, akkor az A-tól C-ig vezető útvonal ugyanerre esik. Ennek következménye, hogy az összes forrásból egy célba tartó optimális útvonalak egy fát alkotnak, aminek a gyökere a célállomás. Ezt a fát nevezik nyelőfának (sink tree). A forgalomirányító algoritmusok célja a nyelőfák felderítése és használata az összes router számára.

1. Legrövidebb útvonal alapú forgalomirányítás:

Statikus algoritmus. Alapötletként, az alhálózathoz felépítünk egy gráfot, aminek minden csúcspontja egy router, az élek pedig a közöttük lévő kommunikációs vonalakat jelentik. Az algoritmus megkeresi két router közötti legrövidebb útvonalat. Az út hosszát többféle módon mérhetjük: pl. ugrások száma, földrajzi távolság, tesztcsomagra mért sorbanállási és átviteli késleltetés.

Számos algoritmus ismert két csomópont közötti legrövidebb útvonal megtalálására.

Pl. Dijkstra-féle

Minden csomópontot felcímkézünk a forrásponttól való, legrövidebb ismert út mentén mért távolságával. Kezdetben nem ismertek ilyen utak, így minden címke végtelen. Egy címke lehet ideiglenes vagy állandó. Kezdetben minden címke ideiglenes, amikor kiderül, hogy a címke a legrövidebb utat jelzi, állandóvá tesszük.

2. Elárasztás:

Statikus algoritmus. Egy bejövő csomagot minden kimenő vonalon kiküldünk, kivéve azon, amelyiken érkezett. Megfelelő intézkedések nélkül végtelen számú csomagot generál a hálózat.

Intézkedés lehet, hogy egy ugrás számlálással látjuk el a csomagokat, ami minden ugrás során eggyel csökken, és ha nulla lesz az értéke akkor eldobjuk. Ideális esetben az ugrások száma a forrástól célig vezető út hossza, legrosszabb esetben az alhálózat teljes átmérője.

Másik intézkedés lehet, hogy a forrás router elhelyez egy sorszámot minden csomagban. Majd minden routernek szüksége van forrásrouterenként egy listára, amely tartalmazza a már megkapott csomagok sorszámain. Ha a sorszám rajta van a listán, akkor eldobjuk. A lista korlát nélküli növekedését meggátolhatjuk egy számlálással, k -val, aminek a jelentése: minden k alatti sorszám előfordult már. Ebben az esetben a k alatti listaelemekre nincsen tovább szükség.

Az elárasztás egy ésszerűbb változata a szelektív elárasztás, ami csak azokon a kimenő vonalakon továbbítja a csomagot, amelyek megközelítőleg a jó irányba mutatnak.

Az elárasztás használható katonai alkalmazásokban, elosztott adatbázis-alkalmazásokban, vezetékek nélküli hálózatokban például.

3. Távolságvektor alapú forgalomirányítás:

Más megnevezések: elosztott Bellman-Ford forgalomirányítási algoritmus, vagy Ford-Fulkerson-algoritmus. Dinamikus algoritmus. Alapja, hogy minden routernek egy táblázatot kell karbantartania, amelyben minden célhoz szerepel a legrövidebb ismert távolság, és a célhoz vezető vonalnak az azonosítója. A szomszédos routerek időközönként információt cserélnek, így frissítve az adatokat.

A távolság mértékegysége lehet ugrások száma, időbeni késleltetés milliszekundumban, az út mentén sorban álló összes csomag száma, vagy valami hasonló. Probléma: gyorsan reagál a jó hírekre, de ráérősen a rosszakra.

A végtelenig számolás problémája: Lényege, hogy ha X router elmondja Y-nak, hogy van egy valahová vezető útja, Y-nak sehogyan sem áll módjában megtudnia, hogy vajon ő maga rajta van-e ezen az úton. Ha X router meghibásodik (vagy a kapcsolat megszakad), akkor egy másik szomszédja Y-nak (Z) azt fogja üzeni, hogy van egy útvonala X-hez. Ezután Y azt fogja hinni, hogy Z-n keresztül elérheti X-et.

4. Kapcsolatállapot alapú forgalomirányítás:

Dinamikus algoritmus.

Lépések:

1. Felkutatni a szomszédait és megtudni a hálózati címeket.
Speciális HELLO csomagot küld ki minden vonalon. Elvárja, hogy válasz érkezzen rá, amiben a szomszédos router globálisan egyedi azonosítója található.
2. Megmérni a késleltetést vagy költséget minden szomszédjáig.
Speciális ECHO csomagot küld ki, amit azonnal vissza kell küldeni. A késleltetés megmérése és kettővel osztva reális becslés kapható a késleltetésre. A folyamatot többször elvégezheti, és átlagolhat. Kérdés, hogy a terhelést figyelembe vegyék-e. Ha figyelembe vesszük, akkor a forgalomirányító táblázatok ingadozhatnak. Ennek ellenére is érdemes több vonal között megosztani a terhelést.
3. Összeállítani egy csomagot, amely a most megtudottakat tartalmazza.
Csomag: Saját azonosító, sorszám, korérték, szomszédok listája. Könnyű összeállítani, azt viszont nehéz eldönteni, hogy mikor tegyék meg ezt. Periodikusan vagy jelentős változás esetén.
4. Elküldeni ezt a csomagot az összes többi routernek.
Elárasztással küldjük szét a csomagok szétosztására. Sorszámozva, a routerek számon tartják, hogy melyikeket látták már. Problémák: (a) sorszám átfordulás \rightarrow 32 bites sorszámok; (b) router összeomlás; (c) sorszám megsérül. \rightarrow sorszám után a csomag korát is betesszük, ami csökken és 0-nál eldobják.
Robosztusabbá tétel: Nem küldik azonnal tovább a csomagokat, hanem várakozó területre kerülnek. Ha beérkezik még egy csomag ugyan attól a forrástól, akkor összehasonlítják őket, egyezés esetén az utóbbit, egyébként az előbbit dobják el, és küldik ki a másikat.
5. Kiszámítani az összes többi routerhez vezető legrövidebb utat.
Alhálózat gráfjának megszerkesztése, minden kapcsolat kétszer szerepel benne, mindét irányba egyszer-egyszer. Ezek átlagolhatóak, vagy külön használhatóak. Helyileg lefuttatott Dijkstra-algoritmussal megállapítják a legrövidebb utat minden célhoz. Ha az alhálózat n routerből áll és mindegyiknek k szomszédja van, akkor a bejövő adat tárolásához szükséges memória terület $k \cdot n$ -nel arányos. példaprotokollok: OSPF, IS-IS (Intermediate System – Intermediate System).

5. Hierarchikus forgalomirányítás:

A routereket tartományokra osztjuk, a tartományokat kerületekbe, azokat zónákba egészen addig, ahány hierarchia szintre van szükségünk. A routerek forgalomirányító táblázatának mérete csökken, de mivel átlagos úthossz alapján történik a forgalomirányítás, így előfordulhat, hogy egy állomáshoz létezik rövidebb útvonal. N routerből álló alhálózat esetén a szintek optimális száma $\ln N$, amely $e \cdot \ln N$ bejegyzést igényel routerenként.

6. Adatszóró forgalomirányítás

Egy csomag mindenhol történő egyidejű elküldését adatszórásnak (broadcasting) nevezzük.

Megvalósítások:

1. A forrás egyszerűen külön csomagot küld minden egyes rendeltetési helyre. Sávzsélesség pazarló és teljes listával kell rendelkezni az összes célról.
2. Elárasztás, probléma: túl sok csomagot generál és túl nagy sávzsélességet fogyaszt.
3. Többcélú forgalomirányítás: Minden csomag tartalmaz vagy egy listát a rendeltetési helyekről vagy egy bittérképet, amely a kívánt rendeltetési helyeket jelzi. Egy bejövő csomagot tovább küldünk minden olyan kimenő vonalon, amely legjobb út egy rendeltetési hely felé.
4. Nyelőfa vagy valamilyen feszítőfa alapján, minden router továbbküldi a feszítőfához tartozó adatvonalain a csomagot, kivéve ahonnan érkezett. Ez az eljárás használja ki legjobban a sávzsélességet. Probléma: ismernie kell minden routernek valamilyen feszítőfát. (pl. kapcsolat alapú forgalomirányítás)
5. Visszairányú továbbítás: Minden beérkezett csomagra ellenőrzi a router, hogy azon a vonalon érkezett, amelyen ő szokott küldeni a forráshoz küldeni. Ha igen, akkor minden vonalon továbbítja, egyébként eldobja, mint lehetséges másodpéldány.

8 Forgalomirányítás mozgó hosztok esetében:

Feltétel: minden hosztnak van egy állandó lakhelye. A világot felosztjuk körzetekre: LANvagy vezeték nélküli cella. Minden körzetnek van egy idegen ügynöke, amely nyilvántartja minden, ebbe a körzetbe látogató mozgó felhasználót. Valamint egy hazai ügynöke, amely azéppen másik cellában lévő felhasználóit tartja nyilván.

Kapcsolat teremtés lépései:

1. az idegen ügynökök periodikusan hirdetik a címeiket, amire válaszolhatnak a mozgó hosztok. Vagy saját magukat hirdetik az ügynökök felé.
2. A mozgó hoszt bejegyezteti magát az ügynöknél.
3. Az idegen ügynök kapcsolatba lép a mozgó hoszt hazai ügynökével. Valamilyen biztonsági információval ellátott csomagot küld neki.
4. A hazai ügynök ellenőrzi a biztonsági információt, ha elégedett utasítja a másik ügynököt a folytatásra.
5. Az idegen ügynök bejegyzi a táblázataiba a mozgó hoszt adatait és értesíti a regisztrációról.

9. Forgalomirányítás ad hoc hálózatokban:

Olyan csomópontokból álló hálózat, amelyek véletlenül éppen egymás közelében tartózkodnak. Csomópontok: egy routerből és egy hosztból áll, általában egyazon gépen belül.

Ad hoc igény szerinti távolságvektor alapú algoritmus: A Bellman-Ford-féle távolságvektor algoritmus távoli rokona, amit hozzáigazítottak a korlátozott sávzsélességhez és akkumulátor-kapacitáshoz. Igény szerint működik, azaz csak akkor határoz meg egy utat valamilyen cél felé, ha ténylegesen csomagot küldenénk oda.

Az algoritmus minden csomópontban karbantart egy táblázatot az egyes címzettekhez vonatkozó információkkal. Ha a nincs ebben a táblázatban információ a címzetről, akkor ROUTE REQUEST csomagot kezd el sugározni. Ebben benne van a forrás címe, a cél címe, egy kérelem-azonosító valamint egyéb számlálók.

Ha egy csomópont ROUTE REQUEST csomagot kap:

1. Megnézi, hogy van-e Forráscím, Kérelem-azonosító páros bejegyzés a helyi előzménytáblázatában. Ha nincs a csomópont felveszi a párost a táblázatba.
2. A vevő kikeresi a címzettet a forgalomirányító táblázatából. Ha van ismert út a címzett felé, egy ROUTE REPLY csomagot küld vissza.
3. Ha nem ismer útvonalat, akkor az Ugrásszámlálót megnöveli eggyel és újra szétküldi a ROUTE REQUEST csomagot. A csomag tartalmát pedig eltárolja a visszairányú útvonaltáblázatában. Ez alapján építi fel a visszafelé vezető útvonalat majd.

A topológia bármikor megváltozhat, és ehhez alkalmazkodnia kell az algoritmusnak. Ennek érdekében minden csomópont periodikusan szétküld egy Hello üzenetet, amire ha nem kap választ egy csomóponttól, tudja, hogy az elérhetetlenné vált. Minden N csomópont minden lehetséges címzetthez nyilvántartja azon szomszédait, melyek az elmúlt ΔT idő során csomagot továbbítottak neki az adott címzetthez. Ezeket hívjuk az N adott célra vonatkozó aktív szomszédainak.

Torlódásvédelmi algoritmusok

Túl sok csomag van jelen a hálózatban -> teljesítőképesség visszaesik (torlódás).

Előfordulás okai:

1. Hirtelen nagy mennyiségű csomag érkezik több bejövő vonalon, és mindegyiknek ugyan arra a kimenő vonalra van szüksége. Várakozó sor épül fel.
2. Kevés memória, hogy mindet befogadja -> csomagok vesznek el. Több memória tudja kezelni egy ideig a problémát, de kimutatták, hogy végtelen kapacitású memória esetén a torlódás nem javul, hanem rosszabbá válik (a várakozás miatt lejár az időzítőjük).
3. Lassú processzorok a routerekben

A torlódás képes önmagát gerjeszteni, pl. Ha egy vevőnek nincs szabad puffere az újonnan érkező csomagokat el kell dobni. Ha az adó időzítője lejár, akkor újra küldi a csomagot (akár végtelenszer), így nem szabadul fel az ő puffere sem soha, vagyis a torlódás ebben az esetben visszafelé terjed.

A torlódás védelem azzal foglalkozik, hogy az aláhálózat képes legyen elszállítani a kért forgalmat.

Szabályozás elméleti szempontból két csoportra oszthatóak: nyílthurkú és zárthurkú megoldások. Nyílthurkú rendszerek esetén eleve nem engedik a torlódást kialakulni (tervezés), zárthurkú rendszerek a visszacsatolt kör elvén alapulnak: Figyelni kell a rendszert, továbbadni a torlódásra vonatkozó információkat, módosítani a rendszer működését, hogy helyrehozzuk a problémát.

Torlódásvédelem virtuális áramkör alapú alhálózatokban

Belépés ellenőrzése: Miután jelezték a torlódást, nem építünk fel új kapcsolatot. Durva, egyszerű és könnyű kivitelezni. Alternatív megközelítés, hogy engedünk új virtuális áramköröket, de úgy irányítjuk ezeket, hogy elkerüljék a problémás területeket.

Más megközelítés: megegyezés kezdeményezés a virtuális áramkör felépítésekor a hoszt és az alhálózat között. Meghatározza a forgalom nagyságát, alakját, a kívánt szolgálat minőségét és egyéb paramétereit. Az alhálózat erőforrásokat foglal le az útvonal mentén, így valószínűtlen a torlódás kialakulása. A lefoglalást mindig megtehetjük, vagy csak torlódás esetén. Ha mindig megtesszük, akkor előfordulhat, hogy erőforrást pazarlunk. (kihasználatlan sávszélesség)

Torlódásvédelem datagram alapú alhálózatokban

Alkalmazható virtuális áramkörök esetén is. Minden router megfigyeli a kimeneti vonalai és egyéb erőforrásainak kihasználtságát. Ezt jelöljük u -val. Periodikusan mintákat veszünk f -ből (0 vagy 1), és a következő képlet alapján frissítjük u -t:

$$u_{új} = au_{rég} + (1 - a)f$$

Az a konstans határozza meg, hogy mennyi ideig emlékszik a router a közelmúlt történéseire. Ha u a küszöbszint felé emelkedik a vonal, figyelmeztető állapotba kerül. Ha figyelmeztető állapotban lévő vonalon kellene továbbítani egy csomagot, akkor valamilyen intézkedést kell hoznunk:

1. Figyelmeztető bit

A csomag fejrészét egy figyelmeztető bittel látjuk el, a nyugtába is bekerül ez a bit, így a forrás a visszaérkező nyugták figyelmeztető bitjeinek alapján szabályozza a forgalmat. A forgalom csak akkor nőhet, ha egyetlen routernek sincsenek gondjai.

2. Lefojtó csomagok

A router egy lefojtó csomagot küld vissza a forrás hosztnak, amiben benne van a célsomópont címe. Ennek hatására a hosztnak csökkentenie kell a célsomópontnak küldött forgalmát valamilyen százalékkal. Egy ideig figyelmen kívül hagyja az ugyan azon célsomópontra vonatkozó lefojtó csomagokat, majd újra figyel, ha nem érkezik újabb akkor növeli a forgalmat, egyébként tovább csökkenti.

3. Léprésről lépésre ható lefojtó csomagok

A lefojtó csomag minden routerre kifejti hatását, ami a forrás hoszt felé vezető úton található. Több pufferre van szükség, de a torlódást csomagvesztés nélkül elfojthatjuk.

Terhelés eltávolítása

Ha a routereket elárasztják olyan csomagok, amikkel nem tudnak megbirkózni egyszerűen eldobják őket.

Melyik csomagokat dobják el:

- Bor politika (régibb, mint az új): Az újonnan érkező csomagokat dobják el. (pl. fájlátvitel)
- Tej politika (újabb, mint a régi): A régi csomagokat dobják el (pl. multimédia).

Véletlen korai detektálás (Random Early Detection):

Némely szállítási protokollban (pl. TCP) a forrás lassítással reagál az elveszett csomagokra. Mivel a vezetékes hálózatok nagyon megbízhatóak, így valószínűsíthető, hogy a csomagvesztést a pufferek túlsordulása okozta.

A routerek figyelik a sorhosszaik átlagát, és ez alapján határozzák meg, mikor kell csomagokat eldobni. Már az előtt elkezdik a folyamatot, mielőtt tényleges torlódás alakulna ki. Véletlenszerűen választ egy csomagot a sorból, amit eldob. (mivel a torlódás forrását nem tudja meghatározni).

Forrás értesítése:

- lefojtó csomag: tovább terheli a hálózatot
- nem szól senkinek -> előbb-utóbb észreveszi a forrás a nyugta hiányát és lassít

Vezeték nélküli hálózatokban a második módszer nem alkalmazható (kevésbé megbízható hálózat).

Dzsitterszabályozás: A csomagok megérkezési idejének ingadozása a dzsitter. Kiküszöbölésére vevőoldali puffereket használhatunk, vagy az útvonal mentén minden átvitelre kiszámoljuk a várható átviteli időt, majd a routerek ehhez képest tartják vissza illetve indítják el a csomagokat.

Az IP-protokoll: Az IP-datagram egy fejrészből és egy szövegrészből áll. A fejrésznek van egy 20 bájtos rögzített része valamint egy változó hosszúságú opcionális része.

Rögzített részek:

Verzió: 4 bites mező, megadja a használt protokoll verzióját.

IHL: 4 bites mező a fejrész méretét adja meg -> max 60 bájt lehet a fejrész.

Szolgálat típusa: 6 bit, az eltérő szolgáltatási osztályok között tesz különbséget.

Teljes hossz: 16 bit, a csomag teljes hossza, max érték: 65535.

Azonosítás: A célhoszt el tudja dönteni, melyik datagramhoz tartozik a darab.

DF: 1 bit, jelentése ne darabold!

MF: 1 bit, több darab, egy datagram minden darabjában kivéve az utolsó

Darabeltolás: 13 bit, megmondja, hogy hová tartozik a darab a datagramban.

Élettartam: 8 bit, átvitelcsökkentik, ha nullára ér el kell dobni.

Protokoll: 8 bit, melyik szállítási folyamatnak kell adni a csomagot. (pl. TCP/UDP)

Fejrész ellenőrző összege: 16 bit, csak a fejrészt ellenőrzi

Programtervező informatikus BSc államvizsga tételsor
Informatika tárgycsoport

Forrás és cél cím: 32-32 bit, a hálózat és a hoszt száma.

Opcionális mezők, pl: biztonság, időbélyeg, útvonal feljegyzése.

Pontokkal elválasztott decimális jelölésrendszer a címeknek (32 bit, 4 oktet). Alhálózati maszk segítségével határozzák meg, hogy egy cím melyik része a hálózat illetve az állomás címe.

Osztályos címzést használnak:

A: 1.0.0.0 – 127.255.255.255, Első oktet a hálózat, maradék három a hoszt

B: 128.0.0.0 – 191.255.255.255, Első két oktet a hálózat, maradék kettő a hoszt

C: 192.0.0.0 – 223.255.255.255, Első három oktet a hálózat, maradék egy a hoszt

D: 224.0.0.0 – 239.255.255.255, Többesküldéses cím

E: 240.0.0.0 – 255.255.255.255, Jövőbeli felhasználásra fenntartva

CIDR – Osztálynélküli körzetek közti forgalomirányítás:

Elméletben több mint 2 milliárd cím létezik, de gyakorlatban a címtartomány osztályok szerinti felosztása ebből milliókat elveszteget.

Három medve problémája: A szervezetek számára az A osztályú cím túl nagy, a C túl kicsi, így a B osztályú címekből használnak a legtöbbet, még akkor is, ha nem használják ki a teljes tartományt.

Megoldás a CIDR, aminek az alapötlete, hogy a megmaradt IP-címeket változó méretű blokkokban osszák ki.

A forgalomirányító táblázatokban minden bejegyzést egy 32 bites maszk hozzáadásával egészítik ki. Így most egyetlen forgalomirányító táblázat van az összes hálózathoz, ami hármas tömbökből áll (IP-cím, alhálózati maszk, kimeneti vonal).

NAT – hálózati címfordítás:

Privát IP-címtartományok:

A: 10.0.0.0 – 10.255.255.255 (16 milliónál több hoszt)

B: 172.16.0 – 172.31.255.255 (1 milliónál több hoszt)

C: 192.168.0.0 – 192.168.255.255 (65536 hoszt)

Belső, privát címeket fordít külső címekre, így több hoszt használhatja egyidejűleg ugyan azt az IP címet. A fordításhoz felhasználják a szállítási réteghez tartozó portszámokat (forrás és célport, pl. TCP esetén). Rövidtávú, gyors javításnak szánták, amíg az IPv6 el nem terjed teljesen (128 bites címekkel).

9. Tétel: A szállítási réteg jellemzése, legfontosabb feladatai (összeköttetés-kezelés, kapcsolat felépítés és bontás), gyakorlati példák (TCP)

Feladata: Megbízható, gazdaságos adatszállítást biztosítson a forráshoztól a célhosztig, függetlenül magától a fizikai hálózattól.

Szállítási entitás: Ami a tényleges munkát végzi a rétegen belül.

A szállítási réteg létezése azt teszi lehetővé, hogy a szállítási szolgáltat megbízhatóbb lehessen, annál a hálózati szolgáltatnál, amelyre ráépül.

Szállítási szolgáltató: 1-4. réteg

Szállítási szolgáltat felhasználó: 5. rétegtől.

A szállítási szolgáltat lényege, hogy elrejti a hálózati szolgáltat hiányosságait.

A szállítási primitíveket sok alkalmazás használja, így a szállítási szolgáltatnak kényelmesnek és könnyen használhatónak kell lennie.

Kétféle szállítási szolgáltat létezik: összeköttetés alapú, és összeköttetés nélküli.

Primitívek:

Primitív	Elküldött TPDU	Jelentés
LISTEN	nincs	Vár, amíg egy folyamat kapcsolódni nem próbál
CONNECT	CONNECT REQ.	Összeköttetést próbál létrehozni
SEND	DATA	Adatot küld
RECEIVE	nincs	Vár, amíg adat nem érkezik
DISCONNECT	DISCONNECTION REQ.	Ez az oldal bontani kívánja az összeköttetést

TPDU – Szállítási protokoll adategység: szállítási entitások közötti üzenetek.

Beágyazódás menete: (Keret(csomag(TPDU)))

Összeköttetés felépítése és üzenet váltás menete:

1. szerver (cél): LISTEN, kliens (forrás): CONNECT
2. szerver: CONNECTION ACCEPTED, kliens: SEND/RECEIVE
3. szerver: SEND/RECEIVE, kliens: SEND/RECEIVE

Összeköttetés bontása: *aszimmetrikus* vagy *szimmetrikus* módon.

Aszimmetrikus: Valamelyik fél DISCONNECT primitívet hajt végre, a kérelem beérkezte után bomlik a kapcsolat.

Szimmetrikus: Mindkét irányt a másiktól függetlenül zárják le.

Berkeley TCP-primitívek:

A Berkeley UNIX-ban használt TCP-socket primitívek:

Primitív Jelentés

SOCKET	Új kommunikációs végpont létrehozása
BIND	Helyi cím hozzárendelése a csatlakozóhoz
LISTEN	Összeköttetés-elfogadási szándék bejelentése. várakozási sor hosszának megadás
ACCEPT	Hívó blokkolása összeköttetés-létesítési kísérletig
CONNECT	Próbálkozás összeköttetés-létesítésére
SEND	Adatküldés az összeköttetésen keresztül
RECEIVE	Adatfogadás az összeköttetésről
CLOSE	Összeköttetés bontása

Címzés: TSAP (Transport Service Access Point – szállítási szolgálatelérési pont) portok.

Kezdeti összeköttetés-protokoll: Minden olyan gépnek van egy folyamatszervere, amely szolgálatokat akart felkínálni a távoli felhasználóknak. Több portot figyel egyszerre. Miután megkapja a beérkező kérést, létrehozza a megfelelő szervert, aminek átadja a felhasználóval már fennálló összeköttetést. A módszer nem használható pl. állományszolgáltatók esetében.

Ilyen esetben egy alternatív megoldást használnak: Névszolgáltatónak (katalógusszolgáltató) nevezett speciális folyamat működik ilyenkor. A felhasználó üzenetet küld a kért szolgáltatás nevével, mire a névszolgáltató visszaküldi annak a TSAP címét. Lényeges, hogy a névszolgáltató (vagy folyamatszolgáltató) jól ismert TSAP-címe valóban mindenki által ismert legyen.

Összeköttetés létesítése: Probléma lép fel, ha az alhálózat csomagokat veszít, tárol, vagy akár megkettőz. Legfőbb probléma a késleltetett kettőzések létezése.

Többféle megoldás létezik, de egyik sem teljesen kielégítő:

1. Használjunk eldobható szállítási címeket -> a folyamatszolgáltató-modell működését lehetetlenné teszi.
2. Minden összeköttetésnek adjunk egy egyedi összeköttetés-azonosítót, amit a kezdeményező fél generál és minden TPDU-ba beletesz. Az összeköttetés lebontása után minden szállítási entitás frissíti a befejeződött összeköttetések tábláját, ami párokból áll (társ szállítási entitás, összeköttetés sorszám). Probléma, hogy minden szállítási entitásnak határozatlan ideig történeti információt kell tárolnia. -> Ha egy gép összeomlik, ez az információ elveszik.
3. A csomagok élettartamát korlátozzuk:
 - a. *Korlátozott alhálózat tervezése:* Olyan módszerek, amelyek megelőzik, hogy a csomagok hurokba kerüljenek, valamint a torlódási késleltetést is korlátozni tudják az ismert leghosszabb útvonalon.
 - b. *Átugrásszámláló alkalmazása a csomagokban:* Az átugrás számot valamilyen alkalmas értékre állítják be, ami folyamatosan csökken. A hálózati protokoll minden olyan csomagot eldob, amelynek az átugrásszámlálója nulla.
 - c. *A csomagok időbélyeggel való ellátása:* Minden csomagot ellátnak keletkezésének időpontjával, majd a routerek minden olyan csomagot eldobnak, ami öregebb a közösen meghatározott legnagyobb lehetséges élettartamnál. A routerek órát szinkronizálni kell.

Nem elég biztosítanunk, hogy egy csomag halott, hanem ennek igaznak kell lennie minden rá vonatkozó nyugtára is, ezért bevezetjük T időtartamot, ami a valódi maximális csomagélettartam kis egész számú többszöröse. Egy csomag elküldését követően T idő várakozás után biztosak lehetünk abban, hogy minden nyom nélkül eltűnt a csomag.

Az összeomlás utáni helyreállítás megoldása:

Minden hosztot időt mutató órával lássanak el, aminek a hoszt meghibásodása után is tovább kell járnia. Az óra egy bináris számláló, úgy, hogy a számlálóban lévő bitek számának egyenlőnek vagy nagyobbak kell lennie, mint a sorszámokban lévő bitek száma. Az óráknak nem szükséges szinkronban járniuk.

Az algoritmus nem engedi két azonos sorszámú TPDU egyidejű létezését. El kell kerülnünk a sorszámok használatát a potenciális kezdeti sorszámként történő alkalmazás előtti T időtartamban (tiltott tartomány). Bármely TPDU bármely összeköttetésen történő továbbítása előtt a szállítási entitásnak le kell olvasnia az óráját, hogy ellenőrizze, nem a tiltott tartományban van-e.

Háromutas kézfogás: Nem igényli, hogy mindkét fél azonos sorszámmal kezdje az adást. Az első hoszt kiválaszt egy x sorszámot, és egy CONNECTION REQUEST TPDU-ban elküldi a 2. hosztnak. Az egy CONNECTION ACCEPTED TPDU-val nyugtázza az x értékét, és bejelenti saját y kezdeti sorszámát. Végül az első hoszt jóváhagyja a 2. hoszt által választott kezdeti sorszámot az első általa küldött adat TPDU-ban.

Az összeköttetés bontása

Lehet aszimmetrikus és szimmetrikus módon kapcsolatot bontatni. Az aszimmetrikus összeköttetés-bontás váratlanul történik és adatvesztéssel járhat. A szimmetrikus bontás esetén egy hoszt azután is fogadhat adatot, hogy már elküldött egy DISCONNECT REQUEST TPDU-t.

Két-hadsereg probléma: Bebizonyítható, hogy nem lehet működő protokollt létrehozni. Ha egyik fél sem készült föl a bontásra addig, míg meg nem győződött arról, hogy partnere is fölkészült, az összeköttetés bontására sohasem kerül sor.

Félig nyitott összeköttetések kilövése: Bevezetünk egy olyan szabályt, miszerint ha adott ideig nem érkezik TPDU, az összeköttetést automatikusan bontjuk. Ennek érdekében alkalmazunk egy időzítőt, amit minden TPDU elküldése után nullázunk és újraindítunk. Ha lejár, akkor egy üres TPDU küldésével tartjuk vissza a vevőt az összeköttetés bontásától.

Forgalomszabályozás és puffereles

Csúszóablakos (vagy valamilyen más) módszer szükséges, hogy megvédhessük a lassú vevőt a túl gyorsan adó üzenetömegetől.

Megbízhatatlan hálózati szolgálat esetén a küldő minden kimenő TPDU-t pufferelni kényszerül. Megbízható hálózat esetén ha a vevő garanciát ad arra, hogy mindig van kellő mennyiségű szabad puffere, akkor nem kell az adó oldalon pufferelni, egyébként igen.

Puffer méret: Ha a legtöbb TPDU azonos méretű, kézen fekvő a puffertérületet azonos méretű pufferek készletébe szervezni. Másik lehetőség a változó méretű pufferek használata, aminek előnye a jobb memória kihasználtság, viszont bonyolultabb pufférkezeléssel jár.

Harmadik lehetőség, hogy összeköttetésenként egyetlen nagy körpuffer alkalmazása. Ez a módszer szintén jó memóriakihasználtságot eredményez, ha az összeköttetések erősen terheltek.

Kis sávszélességű löket szerű forgalom esetén célszerű a küldőnél megvalósítani a pufferelést, míg egyenletes, nagy sávszélességű adatfolyamot a vételi oldalon előnyösebb pufferelni.

Dinamikus pufférkezelés: változó méretű csúszóablakok alkalmazása. Vezérlő TPDU-kal, vagy a visszairányú forgalomra ültetik a vezérlő információkat az pufferek állapotáról.

Holtpont alakulhat ki, ha az adó oldal nem kap információt a vevő oldali puffer méretéről. Ennek elkerülésére mindkét hoszt rendszeres időközönként vezérlő TPDU-kat küld társának, amelyekben nyugta és az összeköttetésekhez tartozó pufferek állapotáról található információ.

Nyalábolás

Szükség van valamilyen módszerre, amellyel eldönthetjük, hogy a beérkező TPDU-kat melyik folyamatnak kell továbbítani. Ezt nevezzük feltöltési multiplexelésnek.

Ha egy felhasználónak nagyobb sávszélességre van szüksége, mint amennyit egyetlen virtuális áramkör nyújt, akkor több hálózati összeköttetést kell megnyitni, és ezeken körforgásos alapon elosztani a forgalmat. Ezt nevezzük letöltési multiplexelésnek.

TCP – Transmission Control Protocol (átvitel-vezérlési protokoll)

Kifejezetten arra tervezték, hogy megbízható bájtfolyamot biztosítson a végpontok között egy egyébként megbízhatatlan összekapcsolt hálózaton.

Mind a küldő, mind a fogadó létrehoz egy csatlakozónak (socket) nevezett végpontot. A csatlakozó címe, ami a hoszt IP-címéből és egy hoszton belül érvényes 16 bites számból, apert azonosítójából tevődik össze.

Egy csatlakozó egyidejűleg több összeköttetés kezelésére is használható.

Az 1024 alatti protokat jól ismert portoknak hívják: 21 – FTP, 23 – TELNET, 80 – http, 110 –POP3

A különböző szolgáltatásokhoz tartozó kiszolgáló folyamatokat démonoknak hívjuk. Hogy memóriát gazdaságosan lehessen kihasználni, nem kell a gép indulásakor létrehozni az összes démont minden folyamathoz. Helyette (pl. UNIX) egy inetd-nek nevezett démon fut, ami egyidejűleg több portot figyel. Ha valamelyiken kérés érkezik, akkor létrehoz egy új folyamatot, amiben lefuttatja a megfelelő démont.

Minden TCP összeköttetés duplex és kétpontos. Nem támogatja az adatszórást és a többesküldést.

Az összeköttetésen bájtfolyamok áramlanak, és a rendszer nem őrzi meg az üzenethatárokat. A TCP-szegmensek egy rögzített 20 bájtos fejrészből (+ opcionális mezők), valamint 0 vagy több adatbájtból állnak. Méretre vonatkozóan két korlát van: a 65515 bájtos IP-adatmező (fejrésszel együtt), valamint a legnagyobb átvihető adategység (MTU), ami Ethernet esetében 1500 bájt.

A TCP-szegmens fejrésze:

Forrásport: 16 bit

Célport: 16 bit

Sorszám: 32 bit

Nyugta: 32 bit, a következő várt bájt sorszámát tartalmazza.

TCP fejrész hossz: 3 bit, megmondja milyen hosszú a fejrész (opciók miatt)

URG: 1 bit, sürgősségi mutató.

ACK: 1 bit, nyugtát tartalmaz

PSH: 1 bit, késedelem nélküli adat továbbítást.

RST: 1 bit, összezavart összeköttetés helyreállítására.

SYN: 1 bit, összeköttetés létesítésére.

FIN: 1 bit, összeköttetés bontására.

Ellenőrzőösszeg: 16 bit, ellenőrzi a fejrész, az adat és a pszeudofejrész épségét.

Sürgősségi mutató: 16 bit

Opciók: 0 vagy több 32 bites szó.

Adatok: opcionális (üres szegmens -> nyugta, vezérlő információk)

Pszeudofejrész:

Forráscím: 32 bit (IP)

Rendeltetési cím: 32 bit (IP)

8 db nulla

Protokoll azonosító: 8 bit, PID = 6 (TCP)

TCP szegmens hossza: 16 bit

Azért alkalmazzák, mert így a tévesen továbbított csomagok könnyebben detektálhatók.

Összeköttetés létesítésére: háromutas kézfogás technikája.

Összeköttetés lebontása: Mindkét irányba külön külön (FIN = 1, szegmens küldésével). A két-hadsereg probléma elkerülésére időzítőket használnak.

TCP átviteli politika:

Nagle-féle algoritmus: Amikor a küldőhöz bájtanként érkezik az adat, csak az elsőt továbbítja, a többi puffereli, amíg az elküldött bájt nyugtája meg nem érkezik. Ezután a pufferben tárolt összes karaktert egyetlen TCP-szegmensben elküldi, és újra kezdi a pufferelést, amíg az összes nyugta meg nem érkezett.

Buta ablak jelenség (silly winows syndrome): Amikor a küldő TCP-entitás nagy blokkokban kapja az adatokat, de a fogadó oldalon futó interaktív alkalmazás bájtanként olvassa be. Megoldásként, nem szabad megengedni a vevőnek, hogy 1 bájt változásra ablakméret frissítést küldjön. Ehelyett várakozni kell addig, amíg elegendő hely szabaddá nem válik, majd ezt kell a küldővel közölni. Ezenkívül a küldő is segíthet, ha nem küld apró szegmenseket. Kiegészíti a Nagle-féle algoritmust.

A TCP torlódáskezelés:

Két potenciális probléma létezik: a hálózat kapacitása és a vevő kapacitása Ennek érdekében minden adó két ablakot használ: az egyik ablakot a vevő szabályozza, a másik pedig a torlódási ablak. A ténylegesen továbbítható bájtok száma a két ablak értékének a minimuma.

A küldő a torlódási ablak kezdőértékét az összeköttetésben használt legnagyobb szegmensméretre állítja be. Ezután elküld egy maximális szegmenst. Ha a szegmensre nyugta érkezik, mielőtt az időzítő lejárna, egy szegmensméretnyi bájtal növeli a torlódási ablak méretét, ami így a maximális szegmensméret kétszerese lesz, és két szegmenst küld el.

Minden sikeresen nyugtázott adatlöket hatására a torlódási ablak megduplázódik (exponenciálisan növekszik). Ezt nevezzük lassú kezdést biztosító algoritmusnak.

Az Internet torlódásvédelmi algoritmus: Egy harmadik paramétert is használ, aminek neve torlódási küszöb, aminek kezdőértéke 64 kilobájt. Amikor időtúllépés következik be ezt a küszöböt az aktuális torlódási ablak méretének felére állítjuk be, és a torlódási ablakot a maximális szegmensméretre állítjuk vissza. Az exponenciális növekedés véget ér, amikor az ablakméret eléri a torlódási küszöböt, innentől lineárisan növeli a torlódási ablakot.

Ha nem történik időtúllépés, a torlódási ablak addig növekszik, amíg el nem éri a vevő ablakméretét. Itt abba marad a növekedés és állandó méretű marad.

A TCP időzítéskezelése:

Ismétlési időzítő: Egy szegmens elküldésekor elindítja, majd ha nem érkezik időben nyugta újra küldi azt. Kérdés mennyi ideig fusson az időzítő?

A megoldás az, ha erősen dinamikus algoritmust használunk, ami a hálózat teljesítőképességének folyamatos mérése alapján újra beállítja az időintervallumot.

Jacobson nevéhez fűződő módszer:

A TCP minden összeköttetés részére fenntart egy RTT-nek nevezett változót, ami a szóban forgó rendeltetési helyig terjedő körülfordulási idő legjobb jelenlegi becsült értéke. Egy szegmens elküldésekor megmérjük mennyi ideig tart, amíg visszaérkezik a nyugta (M).

Ezután az

$$RTT = \alpha RTT + (1 - \alpha)M$$

képlet szerint frissíti az RTT értékét, ahol α egy átlagoló tényező, azt határozza meg mekkora súlyt kapjon a régi érték. Tipikusan $\alpha = 7/8$.

A TCP általában βRTT -t használt, ahol $\beta 2$ volt. Viszont a konstans érték rugalmatlanul viselkedik. Ezért β úgy kell meghatározni, hogy a szórás átlagos szórásával történő olcsó közelítése legyen. (D).

$$D = \alpha D + (1 - \alpha)|RTT - M|$$

Az időzítés hossza: $Időzítés = RTT + 4 * D$.

Újraküldött szegmensek esetén ne frissítsük az RTT értékét, hanem az időzítés hosszát minden kudarc esetén duplázzuk meg, amíg a szegmens végül át nem jut. (Karn-féle algoritmus)

Folytatódó időzítő: Az ablakméret információk elvesztéséből származó holtponthoz elkerülésére.

Ha az időzítő lejár, a küldő egy kérést küld a vevőnek, az aktuális ablakméretről. Életben tartó időzítő: Amikor egy összeköttetés már régóta tétlen, az időzítő lejár és ennek hatására a TCP ellenőrzi, hogy a partnere még mindig működik-e.

10. Tétel: Absztrakt modell, program, a program mint termék, szoftvertechnológia, nagy méretű programrendszer. Az alapvető szoftvergyártási modellek, nagy rendszerek fejlesztésének általános modellje. Nagy rendszerek fejlesztésének lépései, azok jellemzői.

Szoftvertechnológia:

- Gyártási technológiára van szükség.
- A problémamegoldást egyre több szoftvereszköz támogatja.
- Létrehozuk a megoldás absztrakt modelljét, és csak azután kerül sor a megoldás realizálására. (Pl: először a megépítendő ház rajzait terveit, rajzait készítjük el és csak azután fogunk hozzá annak felépítéséhez.)
- A felmerülő probléma: Melyik legyen az a rendszer, amelyben az absztrakt megoldást felírjuk? Válasz: UML.
- Az UML gondolata az, hogy szedjük össze azokat a diagramokat, amelyeket a programozási technológiákban eddig is használtunk. Ezeknek hozzuk létre a jól definiált formáit, amelyeket egy egységes rendszerben foglalhatunk össze.

A program mint termék:

- A programról elmondhatjuk, hogy termékké vált és előállításához technológiára van szükség.
- Termék, mert:
 - van szolgáltatási funkciója,
 - van minősége,
 - van előállításiköltsége,
 - van előállítási határideje.

Probléma a nagy méretű programrendszerek előállításának jelentkezik igazán.

A szoftvertechnológia tárgya a nagy méretű programrendszerek előállítása.

Nagyméretű programrendszerek jellemzői:

- Nagy bonyolultságú rendszer, azaz fejben tartva nem kezelhetőek a kidolgozás során használandó részletek: az objektumok, azok jellemzői, összefüggései.
- Csapatmunkában készül.
- Hosszú élettartamú, amelynek során számos változatát (verzióját) kell előállítani, azokat követni, karbantartani, stb. kell.

Nagyméretű rendszerek fejlesztési lépései és azok jellemzői:

- Megvalósíthatósági tanulmány készítése:
 - Egy adott probléma megoldása előtt meg kell vizsgálni annak megvalósíthatóságát, illetve annak mikéntjét. Ez egy elemzés tárgya, amelynek eredménye a megvalósíthatósági tanulmány.
- Követelmények leírása:
 - Iteratív módon állítjuk elő, és ebben a folyamatban a prototípust használjuk a leírás finomítására, pontosítására.
 - Résztvevők: felhasználó, tervező és a manager.
 - Funkcionális követelmények leírásának módszerei: Beszélt nyelven történő leírás, formáknak alávetett leírás, leíró nyelvek és grafikus modellezési eszközök használata, formális/matematikai leírások.
 - Nem funkcionális követelmények leírásának szintjei: Követelmény definíció, követelmény specifikáció, program specifikáció.
- Követelmények elemzése:
 - Validáció, megvalósíthatósági vizsgálat, tesztelhetőség vizsgálata, nyíltsági kritériumok vizsgálata.
- Programspecifikáció:
 - Bemenő adatok, eredmények és az ezek közti kapcsolatok vizsgálata.
 - Módszerek: Informális, formáknak alávetett és formális módszerek.

Programtervező informatikus BSc államvizsga tételsor
Informatika tárgycsoport

- Tervezés:
 - Statikus, dinamikus és funkcionális modellek megalkotása.
 - Tervezési módszerek:
 - Procedurális: a megvalósítandó funkciókból indulunk ki, és ezek alapján bontjuk fel a rendszert kisebb összetevőkre, modulokra.
 - Objektumelvű tervezés: a rendszer funkciói helyett az adatokat állítjuk a tervezés középpontjába. A rendszer által használt adatok felelnek meg majd bizonyos értelemben az objektumoknak.
- Implementáció:
 - Elterjedt staratégiák:
 - Bottom-up: könnyebb a fejlesztés során történő tesztelés, de a szerkezeti hibákra csak későn derül fény.
 - Top-down: hamar kiderülnek a szerkezeti hibák, de a tesztelés csak a teljes rendszeren lehetséges.
- Verifikáció, validáció:
 - Verifikáció: a specifikáció szerinti helyesség igazolása
 - Validáció: annak ellenőrzése, hogy a rendszer teljesíti-e az előírt minőségi tulajdonságokat.
 - Teszt: az ellenőrzés folyamata.
- Rendszerkövetés, karbantartás:
 - A szoftverrendszer üzembe állítása után szükségesség váló, szoftver jellegű munkák (karbantartás).
 - A felhasználókkal való kapcsolattartás menedzsment jellegű dokumentációs feladatainak ellátása (rendszerkövetés).
- Dokumentáció:
 - Ideális esetben a rendszerkészítés megfelelő fázisaiban történik.

A szoftvertechnológiában megoldandó feladatok:

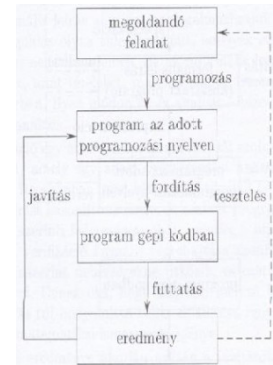
- A rendszerrel szemben támasztott (funkcionális, szolgáltatási, minősége, stb.) követelményeket előre pontosan meg kell határozni, azokat írásban rögzíteni kell.
- A program kidolgozásának menetét meg kell tervezni, meg kell határozni az ún. mérföldköveket; azaz ellenőrzési pontokat, határidőket és a hozzájuk tartozó megoldandó feladatokat.
- Gondoskodni kell a kidolgozáshoz szükséges hardver, szoftver, anyagi és emberi erőforrásokról.
- Dokumentálni kell a programkészítés fázisainak menetét és eredményeit.
- Szervezni, irányítani kell a kidolgozásban részt vevő csapat munkáját és az erőforrások felhasználását.
- Igazolni kell, hogy az elkészült termék megfelel az előre rögzített követelményeknek.
- Meg kell tervezni és szervezni a rendszer követésének, karbantartásának hosszú évekre elnyúló munkáját.

A szoftvertechnológia célkitűzései:

- Előírt minőségű programtermék
- Előre megállapított határidőre
- Előre meghatározott költségen történő előállítás

Egyszerű programfejlesztési modell:

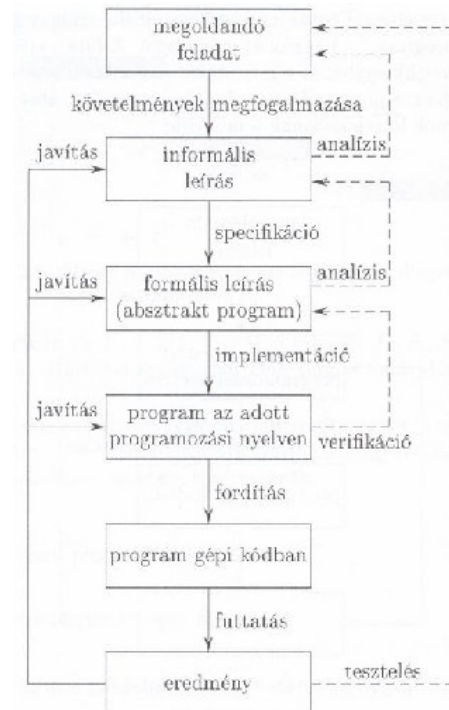
- Rendszerint ezt a modellt alkalmazzuk egyszemélyes programfejlesztésnél.
- Ebben az esetben a megoldandó feladat könnyen áttekinthető, modellezhető, ezért a probléma azonnal megfogalmazható egy adott programozási nyelven.
- Ezután az eredmény előállításáig szoftvereszközök veszik át a feladatot. A futási eredményeket a feladattal vetjük egybe, és a javításokat közvetlenül a programozási nyelvű leírásban, a programkódban hajtjuk végre.



Programfejlesztés specifikációval:

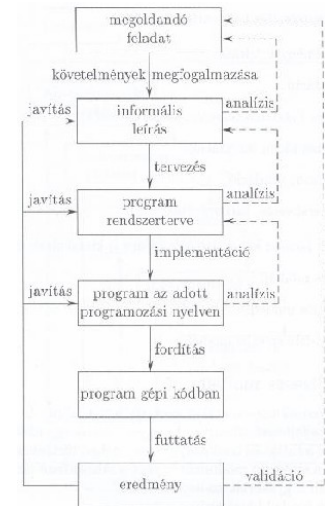
A programkészítés folyamatát bizonyos körülmények esetén matematikailag is kezelni tudjuk.

1. A feladat megoldását először beszélt nyelven, megfelelően rendszerezve írjuk le. Ennek az eredménye egy informális leírás.
2. Ezt elemezzük és összevetjük a megoldandó feladattal annak érdekében, hogy valóban a kívánt megoldást rögzíti-e.
3. Az informális leírás alapján készítjük el egy specifikációra alkalmas nyelven a probléma megoldásának első formális, ezért szintaktikailag egyértelmű leírását.
4. A specifikáció egy absztrakt program leírását szolgáltatja. A konkrét programot, amely egy adott programozási nyelven készül, ezzel vetjük össze.
5. A tesztelés eredménye alapján a javítások érinthetik a programozási nyelven írt váltotzatot vagy az analízis hiányosságai miatt a specifikáció során létrehozott absztrakt programot vagy a beszélt nyelven megfogalmazott informális leírását a probléma megoldásának.



Nagy rendszerek általános fejlesztési modellje:

- Nagy rendszerek esetén a formális specifikációs módszerek egféljebb kisebb részfeladatok – pl. adattípusok – formális leírását, specifikációját teszik lehetővé.
- Ezért ebben az esetben a specifikációs modell formális leírásának helyébe a programrendszer tervének elkészítése lép. Ez egy jól strukturált, többszintű leírás, amely tartalmazza, az implementáció számára szükséges ajánlásokat, előírásokat.



A programkészítés hagyományos fázisai:

- Követelmények leírása
- Specifikáció
- Tervezés (vázlatos tervezés, finom tervezés)
- Implementáció, integráció
- Verifikáció, validáció
- Rendszerkövetés, karbantartás

A fázisok közötti kapcsolatok leírására kialakult modellek:

- Vizesés modell
- Evolúciós modell
- Boehm-féle spirális modell

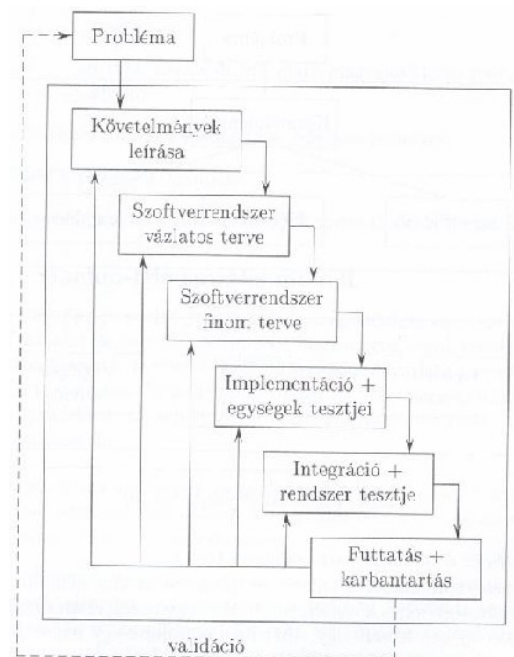
Vizesés modell:

Az egyes fázisok egymást követik, a módosítások a futtatási eredmények ismeretében történnek.

Egy bizonyos fázisban elvégzett módosítás az összes rákövetkező fázist is érinti.

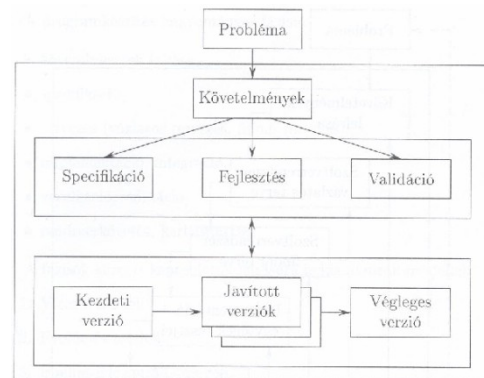
Hátrányai:

- A projekt munkájának megszervezése rendkívül nehézkes (Párhuzamos munka, ellenőrzési pontok elhelyezése).
- Új szolgáltatások utólagos bevezetése szinte minden fázison módosítást kíván meg.
- A validáció az egész életciklus megismétlését követelheti meg.



Evulúciós modell:

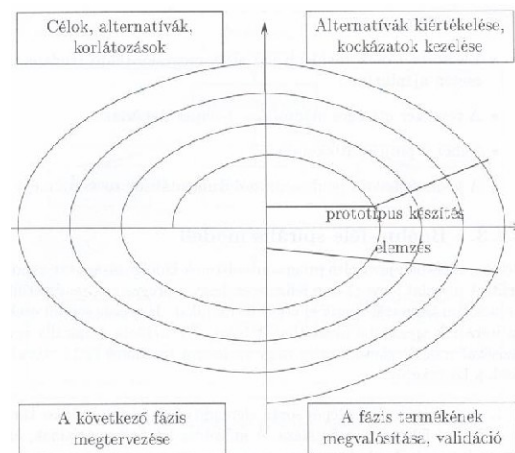
- Prototípusok sorozatát állítjuk elő, és így ahaladunk egészen a végleges megoldásig.
- A programfejlesztések során rendszerint nem áll rendelkezésünkre egy teljes követelményleírás.
- Előállítjuk tehát a megoldás egy első verzióját. Ehhez elkészítjük a felhasználói felületet szimuláló prototípust. Ezt egyeztetjük a felhasználóval, és az elemzés alapján döntünk a következő verzió előállításáról.
- A verziók sorozatának fejlesztésével közelítünk a végső megoldáshoz.



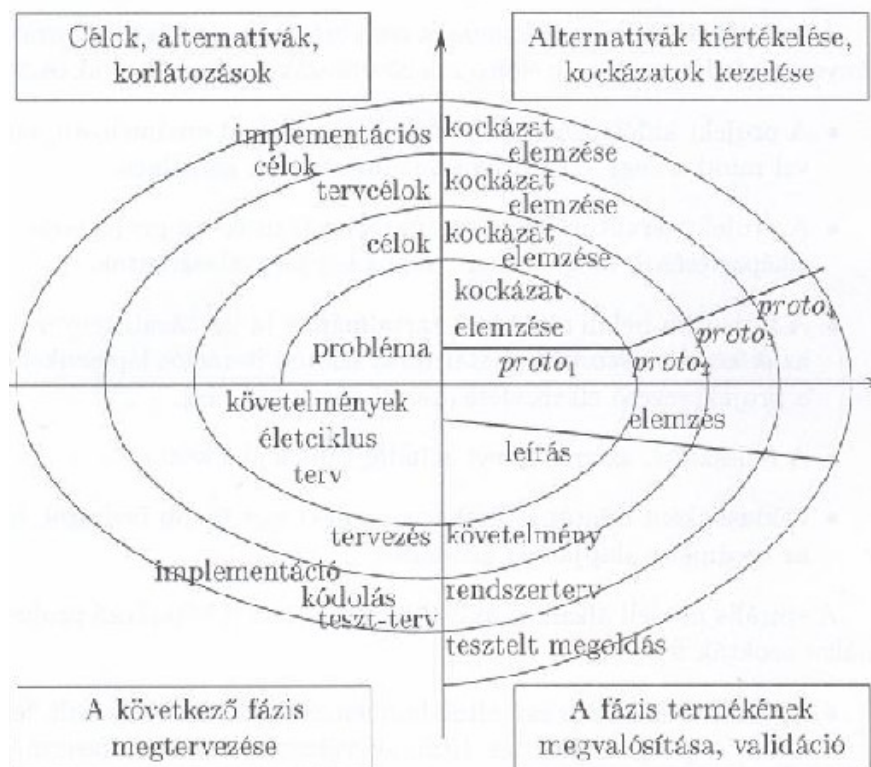
Boehm-féle modell:

A programok rendszerint iterációkon keresztül nyerik el végső formájukat. Az iterációk spirálisba olvashatók össze. Az iteráció a spirális egy fázisával modellezhető, amely 4 szakaszra bontható:

1. elérendő célok és a megoldást korlátozó feltételek definiálása, az elternatívák meghatározása
2. kockázati tényezők elemzése, stratégiák kidolgozása a kockázati tényezők hatásának csökkentésére
3. az iterációs lépésfeladatainak megoldása, validáció
4. a következő iterációs lépés megtervezése



A Boehm-féle spirális modell használatát nagy rendszerek készítése esetén:



11. Tétel: Az objektumorientált szoftvertervezés. Főbb tervezési modellek az UML-ben: Statikus modellek (osztálydiagram, objektumdiagram), dinamikus modellek (állapotdiagram, szekvenciadiagram) és funkcionális modellek (együttműködési diagram, aktivációs diagram).

Az objektumorientáltság egyik célkitűzése az alkalmazásvilág és a megvalósítás-világ közötti szemantikai részesztüntetése. Az objektumorientáltság eszméje az, hogy a való világot a számítógépen modellezzük, azaz a szakmai objektumokat közvetlenül technikai objektumokra képezzük le. Ennek előnyei:

- újrafelhasználhatóság
- jobb érthetőség
- a követelmények gyorsabb, egyszerűbb, hibátlanabb implementálása

Objektumdiagram:

- Egyetlen kidolgozási állapot modellezésére szolgál. pl. tesztet definiálása, rendszerfolyamat megjelenítése.
- Kizárólag osztályok és asszociációk példányait tartalmazzák. A példányok neve alá van húzva.
- Egyszeresen összefüggő gráf, melynek csomópontjai objektumok élei pedig az objektumokat összekötő kapcsolatok.

Az osztály és az objektumdiagram közti különbség, hogy az osztálydiagramok hasonló objektumok csoportjait írják le, míg az objektumdiagramok csupán egy-egy példányt jelenítenek meg.

Osztálydiagram:

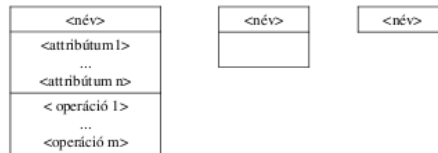
- Az osztálydiagramok az alkalmazás fontos osztályait és azok más osztályokhoz fűződő kapcsolatait mutatják.
- Az osztályok hasonló objektumok egy csoportját írják le, olyan sablonok, amelyek az objektumok létrehozására szolgálnak az alkalmazásban.
- Tehát az osztály a leírás, az objektum a megvalósítás.

Az osztály jellemzői:

1. Hasonló tulajdonságú objektumok egy halmaza.
2. Az osztálynak van neve, amelyet az osztályba tartozó összes objektum örököl.
3. Az osztálynak lehetnek attribútumai, paraméterei, amelyek az objektumoknak is közös építőkövei.
4. Tartoznak hozzá szolgáltatások, operációk, műveletek (export felület).
5. Az osztályhoz tartozhat import felület, amely az általa igényelt szolgáltatások definícióját tartalmazza.
6. Az osztály rendelkezhet megvalósítási résszel. Az osztály specifikációja a következőkből áll:
 - a. paraméterek leírása
 - b. szolgáltatások leírása
 - c. import felület leírása
 - d. megvalósítás leírása
7. Az osztálynak van látható része, és van láthatatlan része. A látható részt a paramétere, a szolgáltatások és az import felület tartalmazza, a láthatatlan részt a megvalósítás leírása tartalmazza.
8. Az osztály lehet absztrakt osztály.
9. Az osztály lehet konkrét osztály.
10. Az osztály lehet paraméteres osztály (sablon), A sablon közös formával rendelkező osztályok egy osztályát definiálja.

Az osztály jelölése:

Programtervező informatikus BSc államvizsga tételsor
Informatika tárgycsoport



Az osztálydiagram definíciója: Az osztálydiagram a problématerben a megoldás szerkezetét leíró, összefüggő gráf, amelynek:

- csomópontjaihoz az osztályokat
- éleikhez az osztályok között relációkat rendeljük

Az osztályok között a következő relációk állhatnak fenn:

- Öröklődés
- Asszociáció
- Aggregáció
- Kompozíció

Kapcsolatok:

- Két osztály közötti vonal egy asszociáció (association).
- A „honnan” és „hova” szavak szerepköröket jelölnek, azt mondják meg, hogy egy járat egy repülőtérről egy másik repülőtérig tart.
- Az asszociációban résztvevő szerepek (role):
 - névvel azonosított szerep
 - kiemelt szerep
 - sorrendiség szerep
- A vonalak végén * áll, amely az egy repülőgépről induló, illetve oda érkező járatok mennyiségét fejezi ki; ez a számosság (multiplicity):
 - pontosan i , jele: i
 - i és j közötti, jele: $i..j$
 - 0 vagy több, azaz valamennyi, jele: *
 - legalább i , jele: $i..*$

Több osztály között fennálló asszociáció: Az asszociáció nem feltétlen két osztályt kapcsol össze. Előfordulhat, hogy több osztályt hozunk kapcsolatba. Ekkor ezt a vonalak metszéspontjában elhelyezkedő rombuszszal szemléltetjük.

Asszociációs osztály: Két osztály közti kapcsolat, amely saját identitással rendelkezik.

Attribútumok:

- A fogalmak pontosabb leírását attribútumok hozzáadásával lehet elérni.
- Az attribútumok sorrendjének jelentősége van. bár ez rendszerint csak a megvalósítási osztályoknál válik fontossá.
- Az attribútumokat típussal is el lehet látni. Ez lehet alaptípus, mint az „int” vagy lehet másik osztály.
- Az attribútumok számossággal is elláthatóak, akár asszociációvégek.
- Láthatóság: public, protected, package, private.

Kompozíció: Az asszociáció szemantikailag egy hivatkozást jelöl, azaz mutatót egy másik objektumra. A kompozíció azt jelenti, hogy az egyik osztály objektumai a másik osztály objektumait fizikailag tartalmazzák. Jelölése: Az egyik osztályból a másikba mutató „nyíl” teli rombusz végződéssel.

Öröklődés:

- Az UML általánosításról (generalization) beszél.
- Az öröklődést fehér hegyű nyíl jelzi, amely a speciális felől az általános felé mutat.

- Az általánosítás a modellalkotásban az általános tulajdonságokkal rendelkező dolog és a kevésbé általános, speciálisabb dolog között fennálló reláció.

Az osztály és **állapotdiagram** kapcsolata: Az osztály objektumainak dinamikus viselkedését a probléma megoldása során az állapotdiagram írja le.

Életciklus: Minden objektumnak van egy életciklusa. Az objektu létrejön, aktív életet él, azaz más objektummal működik együtt különböző feladatok megoldásávan, majd megsemmisül.

Állapot: Az objektum aktív állapotaiban különböző objektumokkal kerül kapcsolatba, és ennek eredményeként különböző állapotokba kerül. Ezt az állapotot az attribútumok konkrét értékeinek n-ésével is jellemezhetjük.

Esemény: Eseménynek nevezzük azt a tevékenységet, történést, amely valamely objektum állapotát megváltoztatja. Az objektum állapota rendszerint nem egy adott pillanatban képezi a vizsgálat tárgyát.

Állapotinvariáns:

- Általában az osztály objektumainak viselkedése kapcsán az azonos állapotban levő objektumokat egy halmazba vonjuk össze.
- Az osztályhoz rendelt állapot esetén az események egy bizonyos halmazára az objektumok azonos módon reagálnak.
- Az osztályhoz rendelt állapot az állapotinvariánssal jellemezhető: $\langle \text{állapot neve} \rangle \{ (attr_1, \dots, attr_n) \mid I(attr_1, \dots, attr_n) \}$.

Az állapot tulajdonságai:


- Az állapotnak van azonosítója.
- Az állapot általában esemény, eseménysorozat hatására jön létre. Ezeket az eseményeket megelőző eseményeknek (pre-events) nevezzük.
- Az állapot időben mindaddig fennmarad, amíg az objektumok attribútumainak értékei kielégítik az állapothoz rendelt invariánst.
- Az állapot fennállása során belső átmenetek fordulhatnak elő. Ezek nem változtatják meg az objektum állapotát, azaz érvényes marad továbbra is az állapotinvariáns.

Állapot:

- Az állapotokat gyakran nehéz megfelelően kifejező névvel ellátni: gyakran azonosítóként használjuk a belső tevékenységek, belső műveletek nevét. Ennek jelölése:

do / <művelet neve>
- Az állapot megszűnlse egy esemény hatására következik be. A megszűnéshez egy eseménysorozat köthet (rákövetkező események, post-events). Azaz az objektum külső állapotátmenetek hatására egy másik állapotban kerül.
- Az objektum megszűnése ugyancsak egy állapotátmenet hatására következik be. Ekkor az objektum egy rendszeren kívüli, úgynevezett befejező állapotba kerül.

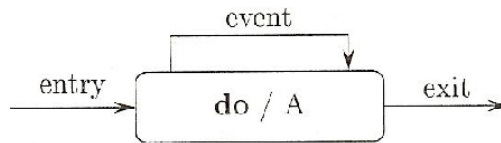
Az állapotdiagram definíciója:

- Az állapotdiagram egy összefüggő irányított gráf, amelynek csomópontjaihoz az állapotokat rendeljük, éleihez pedig az eseményeket.
- Állapotátmenetek: $\text{Események} \left[\begin{matrix} \text{Értékeadások} \\ \text{feltétel} \end{matrix} \right] / [\text{tevékenység}]$
kezdő állapot befejező állapot
- További jelölések: 

Események: Az eseményeknek általában három fázisuk van:

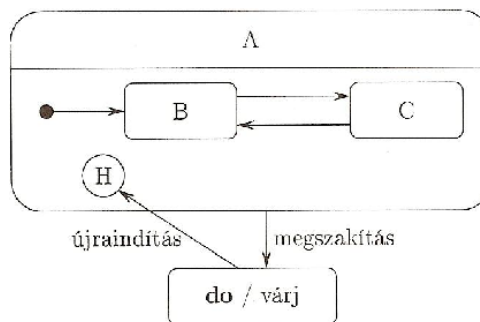
- Egy entry fázis, a belépés akciója, amely elindítja azt az eseményt, eseménysorozatot, amelynek hatására létrejön egy eseményhez rendelt állapot.

- Egy event fázis, amely az adott állapothoz kötődik, azaz belső események sorozata, amely az adott állapothoz kötődő belső állapotokat jelenti.
- Egy exit fázis, a kilépési akció, amely az esemény befejezését, a hozzá rendelt állapotból való kilépést eredményezi.



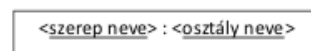
Amennyiben túl sok állapot és állapotátmenet szerepel a diagrammon, úgy a könnyebb átláthatóság kedvéért több állapotot és azok átmeneteit összevonhatjuk egy állapotba (aggregáció), amit külön fejtünk ki.

Az állapot hisztorizációs indikátorának, illetve a hisztorizációs állapotának jelölése: Lehetőséget ad arra, hogy egy adott pillanatban felfüggeszük az állapotokat, majd bizonyos várakozás után újra a felfüggesztés pillanatától folytatódjon az állapotok változása.

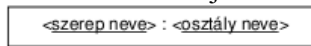


Szekvenciadiagram:

- A probléma megoldása során az objektumok egymásnak üzeneteket küldenek.
- Az üzeneteket időbeli sorrendjének szemléltetése gyakran megkönnyíti a probléma megoldásának megértését.
- A szekvenciadiagram komponensei a következők:
 - osztályszerep
 - osztályszerep életvonal
 - aktivációs életvonal
 - üzenet
- Az osztály szerepét az osztályok közötti üzenetekben megtestesítheti az osztály egy vagy több objektuma.

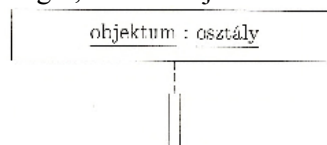


- Az életvonal az osztályszerep időbenvaló létezését jelenti.

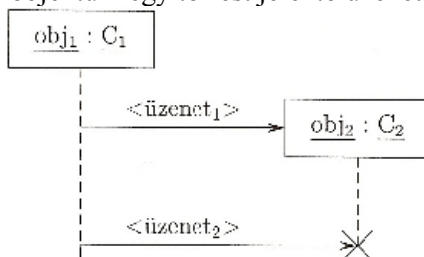


x

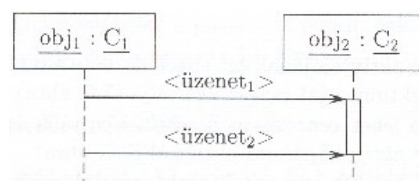
- Az aktivációs életvonal az osztályszerepnek azt az állapotát jelöli, amelyben az osztályszerep megtestesítői műveletet hajtanak végre, és más objektumok vezérlése alatt állnak.



- Egy objektum létrejöhet egy másik objektum létrehozó üzenetének hatására, és megsemmisülhet, ha a másik objektum egy törlést jelentő üzenetet ad ki.

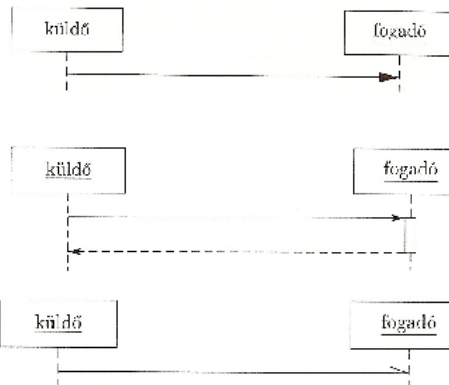


- Életvonala során az objektum aktív módon viselkedhet, amíg valamilyen tevékenységet végrehajt. Az aktív állapotot előidézheti egy másik objektum üzenetét, illetve üzenettel meg is szüntetheti azt.



Üzenet:

- Az üzenet az objektumok közötti információátadás formája.
- Az üzenet egy példányának átadása általában egy esemény bekövetkezése.
- Az üzenet küldésének az a célja, hogy az objektum működésbe hozza a másik objektumot.
- Az üzenet azok között az objektumok között jöhet létre, amelyek az objektumdiagramban kapcsolatban állnak.
- Típusai:
 - Kérés: A külső objektum elküldi az üzenetet, és a küldő blokkol állapotba kerül, amíg a fogadó nem fogadta az üzenetet.
 - Válasz: Az üzenetet egy aktivizált objektum küldi az aktivizáló objektumnak akkor, amikor befejezi a tevékenységet, és a vezérlést visszaadja.
 - Szignál: A küldő folyamat nem szakad meg, nem érdekli, hogy a fogadó mikor kapja meg az üzenetet. (asszinkron üzenet)

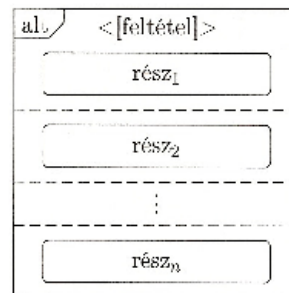
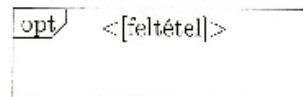


A szekvenciadiagram kiegészítései:

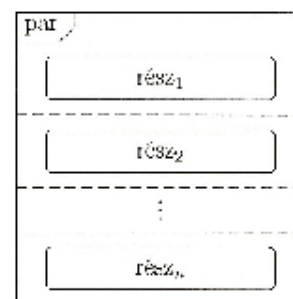
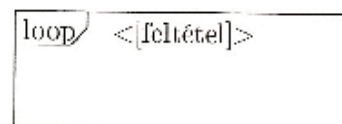
- Példány dekompozíció** esetén a szekvenciadiagram egyik objektumát, illetve osztályszerepét egy másik szekvenciadiagramban fejtjük ki részletesen. A részletezés során az objektum vagy szerep helyét több objektum vagy szerep veszi át, és pontosítjuk, hogy melyik üzenetet melyik részobjektum küldi, illetve fogadja. Lényeges megfigyelés, hogy az üzenetek sorrendje a dekompozíció során nem változhat meg.
- Hivatkozás** esetén a szekvenciadiagram egy időintervallumnak megfelelő részét kiemeljük egy külön diagramba. Az időintervallumnak megfelelő rész rendszerint egy külön diagramba. Az időintervallumnak megfelelő rész rendszerint egy tevékenységnek felel meg. A hivatkozást egy lekerekített sárga téglalappal jelöljük, ami egy névvel (azonosítóval) rendelkezik. A kifejtést tartalmazó szekvenciadiagramot ezzel a névvel látjuk el. (ugrójel)

A változások ábrázolása:

- Egy szekvenciadiagramon belül lehetséges, hogy bizonyos rész üzeneteire csak akkor kerül sor, ha valamilyen feltétel teljesül. Ezt nevezzük opcionális résznek.
- Az is előfordulhat, hogy egy kifejezés értékétől függően eltérő részek lehetnek a szekvenciadiagramban. Ezt nevezzük alternatív résznek.

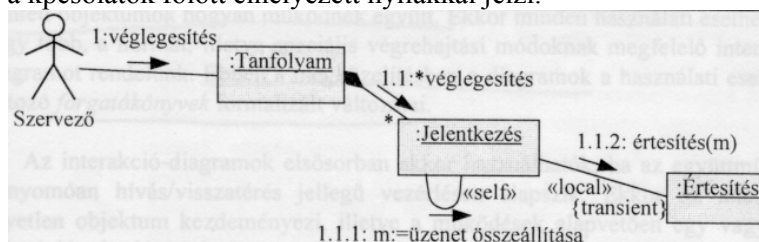


- Egy rendszer működése során bizonyos tevékenységek ciklikusan ismétlődhetnek. Ennek megfelelően a szekvenciadiagram egyes részeinek üzenetei is többször fordulhatnak elő. Az ilyen részt nevezzük iterációs résznek, ciklusnak.
- Párhuzamos, osztott rendszerek esetén bizonyos folyamatok egy időben zajlanak. Ezekben az esetekben a szekvenciadiagramban is biztosítanunk kell a párhuzamosság megjelenítését.



Együttműködési diagram:

- Segítségével az együttműködő objektumok szerveződését, kapcsolódási módjait mutathatjuk be.
- Az időbeli sorrendiséget számozással jelöljük.
- Vácsolhatjuk az objektumoknak az interakció során lezajló állapotváltozásait is
- Az együttműködési diagram az objektumdiagramból indul ki.
- Az üzeneteket a kpcsolatok fölött elhelyezett nyilakkal jelzi.



- Öndelegáció: az objektumra mutató hurok, mellet a *< self >* szócskával.
- Elágazás: azonos sorszámú üzenetek, de egymást kizáró feltételű örszemek.
- Konstrukció: az üzenet megfelelő végén *< new >* szócska
- Desztrukció: az üzenet megfelelő végén *< destroy >* szócska

Aktivációs diagram: Az aktivációs diagram a hagyományos adatfolyam módosított változata. Az aktivációs diagram a probléma megoldásának lépéseit szemlélteti, a párhuzamos zajló folyamatokkal együtt. Az aktivációs diagram az állapotdiagram egy változatának is tekinthető, amelyben az állapotok helyére a végrehajtandó tevékenységek kerülnek és az átmenetek a tevékenységek befejezésének eredményeként valósulnak meg. Az aktivációs diagramot elsősorban az üzleti életben előforduló szervezeti munkafolyamatainak modellezésére használják. Két fajtája van: a sávós és az életfolyamalapú.

Programtervező informatikus BSc államvizsga tételsor
Informatika tárgycsoport

Az egyszerű aktivációs diagram nem jelöli, hogy melyik tevékenységet ki hajtja végre. Ezt sávok felhasználásával azonban megtehetjük. A sávok tetején jelöljük a végrehajtót, alatta pedig a végrehajtott tevékenységet.

12. tétel: Függvények rekurzív megadása, a mester módszer (rekurzív egyenletek megoldása mester módszerrel, a mester tétel), Dinamikus programozás (a rekurzió-memorizálás módszer, a rekurzió megoldása táblázat-kitöltéssel módszer, illetve a rekurzió megoldása lineáris táblázat-kitöltéssel módszer jellemzői).

A mester módszer: A mester módszer a

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

típusú rekurzív egyenletek megoldására ad receptet, ahol $a \geq 1$ és $b > 1$ konstansok, továbbá $f(n)$ asszimptotikusan pozitív függvény.

A $T(n)$ képlet olyan algoritmis futási idejét adja meg, amely n méretű feladatot a darab részproblémára bont, mindegyik mérete n/b , valamely a és b pozitív konstansokra és az a darab részproblémát rekurzívan oldja meg, mindegyiket $T(n/b)$ időben.

A részproblémákra bontás és a részproblémák megoldásaiból akiindulási probléma megoldásának összerakásának idejét az $f(n)$ függvény adja meg.

A mester tétel: Legyenek $a \geq 1$ és $b > 1$ konstansok, $f(n)$ függvény, $T(n)$ pedig a nemnegatív egészekre a

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

rekurzív egyenlettel definiált függvény, ahol n/b jelentheti akár az $\lfloor n/b \rfloor$, akár az $\lceil n/b \rceil$ értéket. Ekkor $T(n)$ -re a következő asszimptotikus korlátok adhatók.

1. Ha $f(n) = O(n^{\log_b a - \varepsilon})$ valamely $\varepsilon > 0$ konstansra, akkor $T(n) = \theta(n^{\log_b a})$.
2. Ha $f(n) = \theta(n^{\log_b a})$, akkor $T(n) = \theta(n^{\log_b a} \lg n)$.
3. Ha $f(n) = \Omega(n^{\log_b a + \varepsilon})$ valamely $\varepsilon > 0$ konstansra, és ha $af\left(\frac{n}{b}\right) \leq cf(n)$ valamely $c < 1$ konstansra és eléggé nagy n -re, akkor $T(n) = \theta(f(n))$.

Példák a mester módszer használatára:

1. Tekintsük a

$$T(n) = 9T\left(\frac{n}{3}\right) + n$$

rekurzív egyenletet. Ebben az esetben $a = 9, b = 3, f(n) = n$, tehát $n^{\log_b a} = n^{\log_3 9} = \theta(n^2)$. Mivel $f(n) = O(n^{\log_3 9 - \varepsilon})$, ahol $\varepsilon = 1$, ezért a mester tétel 1. esetét alkalmazhatjuk és kapjuk a $T(n) = \theta(n^2)$ megoldást.

2. Tekintsük a

$$T(n) = T(2n/3) + 1$$

egyenletet, ahol $a = 1, b = 3/2, f(n) = 1$. $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$. A 2. esetet alkalmazhatjuk, mivel $f(n) = \theta(n^{\log_b a}) = \theta(1)$, tehát a megoldás $T(n) = \theta(\lg n)$.

3. Legyen

$$T(n) = 3T\left(\frac{n}{4}\right) + n \lg n$$

rekurzív egyenletet, ahol $a = 3, b = 4, f(n) = n \lg n$ és $n^{\log_4 3} = O(n^{0.793})$. Mivel $f(n) = \Omega(n^{\log_4 3 + \varepsilon})$, ahol $\varepsilon \approx 0.2$, a 3. esetet alkalmazhatjuk, feltéve, hogy $f(n)$ -re teljesül, hogy asszimptotikusan pozitív. Eléggé nagy n -re $af\left(\frac{n}{b}\right) = 3\left(\frac{n}{4}\right) \lg\left(\frac{n}{4}\right) \leq \left(\frac{3}{4}\right)n \lg n = cf(n)$ ahol $c = 3/4$, tehát $T(n) = \theta(n \lg n)$.

Rekurzió memorizálással:

A partíció probléma rekurzív algoritmussal $\Omega(2^{\sqrt{n}})$ eljárás hívást végez, pedig a lehetséges részproblémák száma csak n^2 (vagy $n(n+1)/2$, ha csak az $n \leq k$ eseteket vesszük). Ennek az az

oka, hogy ugyanazon a részprobléma megoldása több más részprobléma megoldásához kell, és az algoritmus ezeket mindig újra kiszámítja. Tehát egyszerűen gyorsíthatjuk a számítást, ha minden részprobléma (azaz $P2(n, k)$) megoldását tároljuk egy tömbben. Ha hivatkozunk egy részprobléma megoldására, akkor először ellenőrizzük, hogy kiszámítottuk-e már, és ha igen, akkor kiolvassuk az értéket a táblázatból, egyébként rekurzívan számítunk, és utána tároljuk az értéket a táblázatban.

A táblázat inicializálásához válasszunk olyan értéket, amely nem lehet egyetlen részprobléma megoldása sem. A partíciós probléma esetén ez lehet a 0.

```
public class ParticioRM{
    private static long[][] T2;

    private static long P(int n){
        T2 = new long[n+1][n+1];
        return P2(n,n);
    }
    private static long P2(int n, int k) {
        if (T2[n][k] != 0)           //P2(n,k)-t már kiszámoltuk
            return T2[n][k];       //értéke a T2 táblázatban
        long Pnk;
        if (k==1 || n==1)           //rekurzív számítás
            Pnk=1;
        else if (k>=n)
            Pnk=P2(n,n-1)+1;
        else
            Pnk=P2(n,n-1)+P2(n-k,k);
        T2[n][k]=Pnk;               //memorizálás
        return Pnk;
    }
}
```

Nyilvánvaló, hogy az algoritmus futási ideje $\theta(n^2)$, és tárigénye is $\theta(n^2)$ lesz, ha csak n^2 méretű táblázatnak foglalunk memóriát dinamikusan az aktuális paraméter függvényében.

A partíció probléma megoldása táblázat-kitöltéssel: A rekurziót teljesen kiküszöbölhetjük táblázat-kitöltéssel. Egy táblázatot használunk a részproblémák megoldásainak tárolására. Tehát a $T2[n, k]$ táblázatelem tartalmazza a $P2(n, k)$ részprobléma megoldását. A táblázat első sora azonnal kitölthető, mert $P2(n, 1) = 1$. Olyan kitöltési sorrendet keresünk, hogy minden $(n, k), k > 1$ részprobléma kiszámítása esetén azok a részproblémák, amelyek szükségesek $P2(n, k)$ kiszámításához, már korábban kiszámítottak legyenek.

Általánosan, rekurzív összefüggésekkel definiált problémamegoldás esetén egy r (rész)probléma összetevői azok a részproblémák, amelyek megoldásáról r megoldása függ. Tehát a táblázat-kitöltéssel alkalmazásához meg kell állapítani a részproblémáknak egy olyan sorrendjét, hogy minden r részprobléma minden összetevője előbb álljon a sorrendben, mint r . A

1. $P2(1, k) = 1, P2(n, 1) = 1$
2. $P2(n, n) = 1 + P2(n, n - 1)$
3. $P2(n, k) = P2(n, n)$ ha $n < k$
4. $P2(n, k) = P2(n, k - 1) + P2(n - k, k)$ ha $k < n$

rekurzív összefüggések megadják az összetevőket:

1. $P2(1, k)$ -nak és $P2(n, 1)$ -nek nincs összetevője
2. $P2(n, n)$ összetevője $P2(n, n - 1)$
3. $P2(n, k)$ összetevője $P2(n, n)$, ha $(n < k)$
4. $P2(n, k)$ összetevői: $P2(n, k - 1)$ és $P2(n - k, k)$, ha $(k < n)$

Tehát a táblázat kitöltése (k -szerint) soronként balról jobbra haladó lehet.

Az algoritmus futási ideje és tárigénye is $\theta(n^2)$.

```
public static long P(int n) {
    long[][] T2=new long[n+1][n+1];
    for (int i=1; i<=n; i++)
        T2[i][1]=1; //az első sor kitöltése
    for (int ki=2; ki<=n; ki++) { //a ki. sor kitöltése
        T2[ki][ki]=T2[ki][ki-1]+1; //P2(n,n)=P2(n,n-1)+1
        for (int ni=ki+1; ni<=n; ni++) { //P2(ni,ki)=T2[ni,ki] számítása
            int n1=(ni-ki<ki) ? ni-ki : ki; //P2(n,k)=P2(n,n), ha k>n
            T2[ni][ki]=T2[ni][ki-1]+T2[ni-ki][n1]; //P2(n,k)=P2(n,k-1)+P2(n-k,k)
        }
    }
    return T2[n][n];
}
```

A partíció probléma megoldása lineáris táblázat-kitöltéssel: Látható, hogy elegendő lenne a táblázatnak csak két sorát tárolni, mert minden (n, k) részprobléma összetevői vagy a k -adik, vagy a $k - 1$ -edik sorban vannak. Sőt elég egy sort tárolni balról-jobbra (növekvő n -szerint) haladó kitöltésnél, mert amelyik részproblémát felírjuk $((n - k, k))$, annak éppen az új értéke kell összetevőként.

```
public static long P(int n) {
    long[] T = new long[n+1];
    for (int i=1; i<=n; i++) //első sor kitöltése
        T[i]=1;
    for (int ki=2; ki<=n; ki++) { //a ki. sor kitöltése
        ++T[ki]; //P2(n,n)=P2(n,n-1)+1
        for (int ni=ki+1; ni<=n; ni++) //P2(n,k)=P2(n,k-1)+P2(n-k,k)
            T[ni]=T[ni]+T[ni-ki];
        }
    }
    return T[n];
}
```

13. Tétel: Mohó algoritmusok (a mohó stratégia elemei, mohó-választási tulajdonság, optimális részproblémák tulajdonság), Mintaillesztés (mintaillesztés véges determinisztikus automatával, a Knuth-Morris-Pratt mintaillesztő algoritmus, a Rabin-Karp algoritmus).

A mohó stratégia elemei: A mohó algoritmus úgy alkotja meg a probléma optimális megoldását, hogy választások sorozatát hajtja végre. Az algoritmus során minden döntési pontban azt az esetet választja, amely az adott pillanatban optimálisnak látszik. Ez a heurisztikus stratégia nem mindig ad optimális megoldást, azonban néha igen, mint például az esemény-kiválasztási probléma esetén.

Az a módszer, amit követünk mohó algoritmus kifejlesztésére, egy kicsit bonyolultabb az általános esetről. A következő lépések sorozatán kell keresztül mennünk:

1. A probléma optimális szerkezetének meghatározása.
2. Rekurzív megoldás kifejlesztése.
3. Annak bizonyítása, hogy minden rekurzív lépésben az egyik optimális választás a mohó választás. Tehát mindig biztonságos a mohó választás.
4. Annak igazolása, hogy a mohó választás olyan részproblémát eredményez, amelyek közül legfeljebb az egyik nem üres.
5. A mohó stratégiát megvalósító rekurzív algoritmus kifejlesztése.
6. A rekurzív algoritmus átalakítása iteratív algoritmussá.

A gyakorlatban általában egyszerűsítik a fenti lépéseket a mohó algoritmus tervezésekor. A részproblémák kifejtésekor arra figyelünk, hogy a mohó választás egyetlen részproblémát eredményezzen, amelynek optimális megoldását kell megadni.

Általánosan, mohó algoritmus tervezését az alábbi lépések végrehajtásával végezzük:

1. Fogalmazzuk meg az optimalizációs feladatot úgy, hogy minden egyes választás hatására egy megoldandó részprobléma keletkezzen.
2. Bizonyítsuk be, hogy mindig van olyan optimális megoldása az eredeti problémának, amely tartalmazza a mohó választást, tehát a mohó választás mindig biztonságos.
3. Mutassuk meg, hogy mohó választással olyan részprobléma keletkezik, amelynek egy optimális megoldásához hozzátéve a mohó választást, az eredeti probléma egy optimális megoldását kapjuk.

Meg tudjuk-e mondani, hogy adott optimalizációs feladatnak van-e mohó algoritmusú megoldása? Erre nem tudunk általános választ adni, de a mohó választási tulajdonság és az optimális részproblémák tulajdonság két kulcsfontosságú összetevő. Ha meg tudjuk mutatni, hogy a feladat rendelkezik ezzel amikét tulajdonsággal, nagy eséllyel ki tudunk fejleszteni mohó algoritmusú megoldást.

Mohó-választási tulajdonság: Az első alkotóelem a mohó-választási tulajdonság: globális optimális megoldás elérhető lokális optimum (mohó) választásával. Más szóval, amikor arról döntünk, hogy melyi választást tegyük, azt választjuk, amelyik az adott pillanatban jobbnak tűnik, nem törődve a részproblémák megoldásaival. A mohó algoritmus során az adott pillanatban legjobbnak tűnő választást hajtjuk végre, bármi legyen is az, és aztán oldjuk meg a választás hatására fellépő részproblémát. A mohó algoritmus során végrehajtott választás függhet az addig elvégzett választásoktól, de nem függhet a későbbi választásoktól, vagy részproblémák megoldásától.

Természetesen bizonyítanunk kell, hogy a lépésenkénti mohó választásokkal globálisan optimális megoldáshoz jutunk, és ez az ami lelményességet igényel.

A mohó-választási tulajdonság gyakran hatékonyságot eredményez a részprobléma választásával. Például az esemény-kiválasztási feladatnál, feltételezve, hogy az események befejezési idejük szerint monoton nem-csökkenő sorrendbe rendezett-tek, minden eseményt csak egyszer kell vizsgálni. Gyakran az a helyzet, hogy a bemeneti adatokat alkalmasan elő-feldolgozva vagy alkalmas adatszerkezetet használva (ami gyakran prioritási sor), a mohó választás gyorsan elvégezhető, és ezáltal hatékony algoritmust kapunk.

Optimális részproblémák tulajdonság: Egy probléma teljesíti az optimális részproblémák tulajdonságot, ha az optimális megoldás felépíthető a részproblémák optimális megoldásából. Ez az alkotóelem kulcsfontosságú.

Általában sokkal közvetlenebb alkalmazását használjuk az optimális részproblémák tulajdonságnak, amikor algoritmus kifejlesztése során. Mint már említettük, szerencsénk van, amikor feltételezzük, hogy az eredeti probléma mohó választása megfelelő részproblémát eredményez. Csak azt kell belátni, hogy a részprobléma optimális megoldása, kombinálva a már elvégzett mohó választással, az eredeti probléma optimális megoldását adja. Ez a sima implicit módon használ részproblémák szerinti indukciót annak bizonyítására, hogy minden lépésben mohó választást végezve optimális megoldást kapunk.

Hátizsák probléma:

- Egy általunk ismert tároló kapacitású hátizsákba teszünk be tárgyakat. Minden tárgy rendelkezik egy bizonyos súlyal és értékkel. Az érték határozza meg, hogy mennyire hasznos számunkra ez a tárgy. A tárgyakat a súlyoktól és értékektől függően kell bepakolnunk a táskába úgy, hogy végeredményben a bepakolt tárgyak hasznosságának összértéke a lehető legnagyobb legyen (anélkül, hogy túllépnénk a táska tároló kapacitását).
- Egészértékes hátizsák probléma: Egy tárgyat csak egészben tehetünk be a táskába.
- 0-1 hátizsák probléma: Egy-egy tárgy esetén el kell dönteni, hogy a tárgyat betesszük-e vagy sem. Egy tárgyat csak egyszer tehetünk a táskába maximum.
- Tört értékű hátizsák probléma: Ebben az esetben egy tárgynak egy bizonyos részét is választhatjuk (pl. téglapor, só, cukor....) annak érdekében hogy a zsákot feltöltsük.

DFA (Fogalmak):

- Legyen Σ véges jelkészlet, ábécé. Σ elemeit betűknek nevezzük.
- Jelölje Σ^* a Σ elemeiből képezhető összes véges jelsorozat halmazát, beleértve az üres sorozatot is, amit λ -val jelölünk. $\Sigma^* = \{x_1, \dots, x_n : x_i \in \Sigma, i = 1, \dots, n\} \cup \{\lambda\}$
- Σ^* elemeit szavaknak hívjuk.
- $X \in \Sigma^*$ szó hossza az $X = x_1, \dots, x_n$ sorozat elemeinek a száma, jele $|X| = n$.
- A továbbiakban feltételezzük, hogy a szavakat tömb ábrázolja, tehát az X szó i -edik elemére az $X[i]$ tömbelem-kiválasztással hivatkozunk.
- $x = x_1, \dots, x_m, y = y_1, \dots, y_n \in \Sigma^*$ szavak konkatenációja: $xy = x_1, \dots, x_m, y_1, \dots, y_n$.
- Az u szó kezdőszelete, vagy prefixe v -nek, jele $u \subset v$, ha $\exists w \in \Sigma^*$, hogy $uw = v$.
- Az u szó végződése, vagy szuffixe v -nek, jele $u \supset v$, ha $\exists w \in \Sigma^*$, hogy $wu = v$.
- Az $X \in \Sigma^*$ szó első i betűjéből álló prefixére az $X_i = X[1 \dots i]$ jelölést használjuk.
- Általában, egy $X = x_1, \dots, x_n \in \Sigma^*$ szó és $1 \leq i \leq j \leq |X|$ esetén $X_{i,j} = X[i \dots j] = x_i, \dots, x_j$ jelölést használjuk.
- Azt mondjuk, hogy az S szó előfordul i eltolással az A szóban (vagy más szóval S i eltolással illeszkedik az A -ra), ha $A[i + 1 \dots i + m] = S$, ahol $m = |S|$.

Mintaillesztési probléma:

- Bemenet: $A, S \subseteq \Sigma^*$, A a szöveg, S minta.
- Kimenet: A legkisebb (összes) olyan i , amelyre $A[i + 1 \dots i + m] = S$, ahol $m = |A|$ a minta hossza.

Mintaillesztés véges determinisztikus automatával: Véges determinisztikus automata olyan $M = (Q, q_0, F, \Sigma, \delta)$ rendezett ötös, ahol

- Q véges halmaz, az állapotok halmaza,
- $q_0 \in Q$ a kezdőállapot,
- $F \subseteq Q$ a végállapotok (elfogadó állapotok) halmaza,
- Σ véges halmaz, a bemeneti jelek halmaza,
- $\delta: Q \times \Sigma \rightarrow Q$, az automata átmenetfüggvénye.

Jelölje $\delta^* Q \times \Sigma^* \rightarrow Q$ a δ átmenetfüggvény kiterjesztését szavakra, tehát

$$\delta^*(q, w) = \begin{cases} q & \text{ha } w = \lambda \\ \delta(\delta^*(q, x_1, \dots, x_{n-1}), x_n) & \text{ha } w = x_1, \dots, x_n \end{cases}$$

Legyen $M = (Q, q_0, F, \Sigma, \delta)$ olyan véges determinisztikus automata, amely a Σ^*S nyelvet ismeri fel, azaz $L(M) = \Sigma^*S$. Mivel Σ^*S az összes olyan szavak halmaza, amelyek az S mintára végződnek, ezért $A_i \in L(M)$ akkor és csak akkor, ha az S minta $i - m$ eltolással előfordul az A szövegben.

Tehát ha van olyan $M = (Q, q_0, F, \Sigma, \delta)$ véges determinisztikus automatánk, amely a Σ^*S nyelvet ismeri fel, azaz $L(M) = \Sigma^*S$, akkor az alábbi algoritmus mintaillesztést valósít meg.

```

q:=0;
n:=|A|;
m:=|S|;
for i:=1 to n do begin
    q:=δ(q,A[i]);
    if q∈F then begin {S=A[i-m+1...i]}
        WriteLn(i-m+1);
        Exit;
    end;
end;
end;
```

Az algoritmus futási ideje, eltekintve az automata előállítás költségétől $O(n)$.

Hogyan és mekkora költséggel lehet előállítani az M automatát (átmenetfüggvényét)?

Legyen $\sigma: \Sigma^* \rightarrow \{0, 1, \dots, m\}$ az S minta suffix-függvénye,

$$\sigma(W) = \max \{k: S_k \supset W\}$$

Nyilvánvaló, hogy az S minta akkor és csak akkor illeszkedik az A szövegre i eltolással, ha $\sigma(A_{i+m}) = m$.

1. Lemma: Ha $q = \sigma(W)$, akkor $\sigma(Wx) = \sigma(S_q x)$ minden $W \in \Sigma^*$ és $x \in \Sigma$ -re.

2. Lemma: Minden $W \in \Sigma^*$ szóra $\delta^*(0, W) = \sigma(W)$.

Az átmenetfüggvény kiszámítása:

```

m:=|S|;
For q:=0 To m Do
    For x∈Σ Do Begin
        q:=min(m+1,q+2);
        Repeat
            k:=k-1;
        Until S_k]Q_q x;
        δ(q,x):=k;
    End;
End;
```

Knuth-Morris-Pratt algoritmus:

Alapötlet: Ha egy helyen a minta egy kezdőszelete illeszkedik, akkor a következő néhány elemnél nem feltétlenül kell ellenőrizni az illeszkedést, hanem a mintától és az aktuális illeszkedéstől függően lehet előre léptetni az összehasonlító algoritmust. A léptetés mértéke előre kiszámítható a mintára.

Definíció: A minta prefix függvénye $Pre: \{1, \dots, m\} \rightarrow \{0, \dots, m-1\}$ a következőképpen adható meg.

$$PRE(q) = \max\{k: k < q \cap S_k \supset S_q\} \quad q = 1, \dots, m$$

```
Prefix(S)
  m:=hossz(S)
  PRE(1):=0
  k:=0
  for q=2 to m
    while k>0 and S[k+1]!=S[q]
      k:=PRE(k)
    if S[k+1]=S[q]
      then k:=k+1
    PRE(q):=k
  return PRE
```

A mintaillesztő algoritmus:

```
KMP(A,S)
  n:=hossz(A)
  m:=hossz(S)
  PRE:=Prefix(S)
  q:=0
  for i=1 to n
    while q>0 and S[q+1]!=A[i]
      q:=PRE(q)
    if S[q+1]=A[i]
      then q:=q+1
    if q=m
      then PRINT „illeszkedik i-m+1 eltolással”
      q:=PRE(q) //következő illeszkedés keresése
```

Az eljárás ugyanazon az elven alapul, mint a Prefix eljárás, a helyessége ugyanúgy igazolható, és az is, hogy a futási ideje $O(n)$.

Rabin-Karp algoritmus: Az algoritmus a Σ feletti szavakat, egy $|\Sigma| = d$ alapú számrendszerben felírt számként tekint. Ekkor az S mintát egyértelműen megadja a hozzárendelt szám értéke, amit jelöljön s . Ez az s érték kiszámítható $O(m)$ időben.

Az algoritmus alapötlete, hogy az A szövegre, minden i -re kiszámolva az $A[i+1, \dots, i+m]$ számértéket, azon esetekben, ahol ez a szám s illeszkedik a minta. Ezt hatékonyan számíthatjuk, mivel

$$A[i+1, \dots, i+m+1] = (A[i+1, \dots, i+m] - d^m A[i+1])dA[i+m+1],$$

így az első érték után a többiek mindegyike konstans időben kiszámítható.

A módszerrel az a probléma, hogy a létrejövő számok túl nagyok, időigényesek az elemi műveletek, továbbá előfordulhat, hogy nem is ábrázolhatóak a számítógépen a szokásos számítástípusokkal. A megoldás az, hogy a számok helyett csak egy megfelelően nagy, de mégis kezelhető q érték szerinti osztásmaradékokat használjuk.

Természetesen így egy olyan algoritmust kapunk, amely esetén a minta illeszkedik válasz hamis lehet, ha a két szám maradéka megegyezik. Ezért a maradékosztályok megegyezése esetén, a helyességet ellenőrizni kell. Viszont az algoritmus gyors elutasítási heurisztikaként jól használható, mivel ha azt kapjuk, hogy a minta nem illeszkedik, akkor valóban nincs illeszkedés.

```

Rabin-Karp( $A, S, d, q$ )
   $n := \text{hossz}(A)$ 
   $m := \text{hossz}(S)$ 
   $h := d^{m-1} \bmod q$ 
   $s := 0$ 
   $t_0 := 0$ 

  for  $i = 1$  to  $n$            //előfeldolgozás
     $s := (ds + S[i]) \bmod q$ 
     $t_0 := (dt_0 + A[i]) \bmod q$ 
  for  $r = 0$  to  $n - m$        //keresés
    if  $s = t_r$ 
      then if  $S[1, \dots, m] = A[r+1, \dots, r+m]$ 
        then PRINT „illeszkedik  $r$  eltolással”
    if  $r < n - m$ 
      then  $t_{r+1} := (d(t_r - A[r+1]h) + A[r+m+1]) \bmod q$ 

```

Az algoritmus futási ideje legrosszabb esetben $\Omega(nm)$, de az átlagos esetben sokkal jobb: $O(n + m)$.