### 1. ORek feladata, definiciója, felépítésének alapelvei. A kernel és a user mód különbsége

**OR:** Az OR feladata, hogy köztes rétegként működjön a a számítógép hardvere és a számítógép felhasználója között. A OR célja, hogy futtassa a felhasználói programokat, és könnyebbé tegye a felhasználó problémáinak megoldását. Emellett a számítógép használatát egyszerűvé, természetessé, nyilvánvalóvá kell tennie. Az OR fontos feladata még, hogy az elérhető hardver erőforrásokat hatékonyan használja fel (ki).

**Az OR definíciója:** Az OReknek nincsen általánosan elfogadott definíciója. Többféle megközelítésben szokás meghatározni (körülírni), hogy mi is az OR, így pl.: "Bármi, amit a gyártó ad, mikor OR-t rendelsz tőle". Az egyetlen program, ami mindig fut egy számítógépen, az az OR kernele. Minden egyéb vagy rendszerprogram, vagy felhasználói program. Egy OR két fontos feladatot lát el:

•Az OR erőforrásokat oszt el:

oMinden erőforrást az OR kezel

oHatékony és fair döntést hoz, ha versenyhelyzet van egy erőforrásért

•Az OR vezérlő program:

oVezérli a felhasználói programok futtatását, hogy megakadályozza a hibákat és a számítógép helytelen használatát

**Az OR felépítése:** Egy számítógépes rendszert négy nagy egységre lehet felosztani:

•A hardver, ez biztosítja az alapvető erőforrásokat, mint processzor, memória, I/O stb.

•Az OR, ami vezérli és összehangolja a hardverek használatát az egyes programok és felhasználók között

oAz OR-ben közvetlenül a hardver felett helyezkedik el a kernel, ez adja az OR legalapvetőbb funkcionalitását

oA kernel felett helyezkedik el maga az OR, ez rendszer programok, a grafikus felület és minden egyéb OR szolgáltatást magában foglal. Itt található minden olyan funkció is, ami a felhasználói programok futtatását lehetővé teszi.

•A felhasználói programok, amiket a felhasználók futtatnak. Ezek a hardver erőforrásait használják a felhasználók problémáinak megoldására

•A felhasználók, ezek lehetnek emberek, másik számítógépek, rendszerek

Az OR-ek felépítésére több féle megoldás létezik. Minden megközelítés lényege, hogy lehetőleg a feladatot több kisebb részre osszák fel, ne legyen monolitikus a rendszer. Minden modul az OR egy jól körülhatárolt részét alkotja, körültekintően definiálva annak be- és kimeneteit, funkcióit.

A legegyszerűbb felépítés az **egyszerű struktúra**. Az MS-DOS jellemzően egy ilyen rendszer. Ezek általában kis, egyszerű, limitált tudással rendelkező rendszerek, amik aztán fejlődésük során átlépik a saját korábbi terveiket, határaikat. A lehető legkisebb helyfoglalásra törekedtek velük, a moduláris felépítés nem volt cél. Az MS-DOS esetében már látható némi réteges felépítés, de az OR által biztosított rendszerhívás felületek és a funkcionális réteg nem válik el igazán egymástól. Ilyen struktúrájú megközelítés volt még az eredeti UNIX is, ahol két nagy egység volt: a kernel, és a felhasználói programok. A kernel további részekre volt még osztva: interface-ekre, eszközmeghajtókra, amiket a fejlesztés során folyamatosan bővítettek. A kernel tulajdonképpen mindent tartalmazott, ami a hardver felett volt, de a rendszer-hívó felület alatt. Az ilyen monolitikus megközelítés problémája a nehéz implementáció és karbantartás.

A következő szint a **réteges felépítés**. Itt az OR több rétegre van felosztva, ahol a legalsó (nulladik) réteg a HW, a legfelső (n-ik) réteg pedig a felhasználói felület. Az OR egy rétege egy absztrakt objektum implementációja, ami adatokból és az azokat kezelő függvényekből épül fel. Egy jellemző OR réteg különböző adatstruktúrákból épül fel, illetve olyan függvényekből, amikkel felsőbb rétegekből kezelni lehet az adatokat. Ennek a megközelítésnek a legnagyobb előnye az egyszerű felépítése és debuggolása. Mivel minden réteg az alatta lévők eljárásait tudja használni, egyszerű a hibakeresés, és az OR tervezése és implementálása is egyszerűsödik. A problémája a réteges struktúrának, hogy nehéz pontosan definiálni az egyes rétegeket. Mivel minden réteg használ alatta elhelyezkedő rétegeket, a körültekintő tervezés alapvető követelmény. A hatékonyság is gondokat okozhat, mivel ez a megoldás közel sem olyan hatékony, mint más felépítések. Itt ugyanis egy hívás sok olyan rétegen is átvándorolhat a céljáig, amire neki alapvetően nincs szüksége, és ez sok felesleges munkát generál, ami erőforrás és idő pazarló.

Egy újabb típusú megközelítés a **mikrokerneles** megoldás. Itt a kernel csak a legszükségesebb alkotóelemekből áll össze, minden egyéb már rendszer- és felhasználói programok formájában kerül be a rendszerbe. Általában egy mikrokernel biztosít egy minimális folyamat- és memóriakezelő funkcionalitást, illetve némi kommunikációs képességet. A mikrokernel fő feladat, hogy kommunikációt biztosítson a felhasználói rétegben futó programok és szolgáltatások között. Ez jellemzően az üzenetátadás módján valósul meg. A mikrokernel előnye, hogy egyszerűen bővíthető. Minden újdonság a felhasználói szinten kerül be a rendszerbe, így nincs szükség a kernel rendszeres módosítására. A kernel módosítása is egyszerűbb, kisebb a kernel, és könnyebben portolható újabb rendszerekre. Biztonsági szempontból is előny, hogy szinte minden a felhasználói rétegben fut, így a kernelt egy támadás vagy hiba nem befolyásolja. A mikrokernelek hátránya a korlátozott teljesítmény a rendszer függvények megnövekedett overhead-je miatt.

A legmodernebb megközelítés a **moduláris** felépítés, ami már jellemzően objektumorientált nyelveket használ. Itt a kernel áll egy központi magból, és számos dinamikusan csatolt szolgáltatásból. Jellemzően ilyen megoldás a dinamikus betöltött modulok használata, pl.: Linuxban, Unixokban, Solarisban. Ez a megoldás lehetővé teszi, hogy a kernel egy alap funkcionalitást biztosítson, amit aztán később dinamikusan lehet bővíteni (pl.: eszköz driverekkel). A végeredmény emlékeztet a réteges kernelekre, de azoknál flexibilisebb, mivel itt egy modul közvetlenül hívhat egy másik modult. Másrészt pedig hasonlít a mikrokerneles megvalósításra is, de annál meg hatékonyabb, mivel a modulok kommunikációjában nincs szükség üzenetek átadására. A MacOS egy ilyen, hibrid struktúrára épülő kernelt használ.

**A kernel és a user mód különbsége**

A modern OR-ek jellemzően dual-mode-ban működnek, azaz van kernel és user módjuk is. Egy mode bit jelöli a hardver számára, hogy éppen melyik mód van használatban. Hardver erőforrások elérése és kezelése egy modern OR-ben csak kernel módban lehetséges, amit az OR kernele tesz lehetővé. Ez biztosítja, hogy bármilyen (nem megbízható) felhasználói program ne tudjon ellenőrizetlen módon hozzáférni az OR futásához nélkülözhetetlen erőforrásokhoz. Minden felhasználói program user módban fut, viszont ha valamilyen rendszerhívással fordul az OR-hez (pl.: valamilyen hardver erőforrást akar használni), akkor az OR átvált kernel módba, és úgy továbbítja az adott rendszerhívást. Annak lefutása után visszavált user módba, és folytatja a normál futást. Ezzel a kettős megoldással megvédhető az OR a felhasználóktól, és a felhasználók megvédhetőek egymástól is. Ez a védelem az ún. privilegizált utasításokkal hajtható végre, amik kizárólag kernel módban futtathatóak le. User módban ezek az utasítások semmilyen módon nem futnak le. Ilyen utasítás pl.: a kernel-user mód közötti váltást biztosító utasítás, I/O műveletek utasításai, megszakítások kezelése. Ezen utasítások futtatása esetén a hardver visszautasítja a végrehajtást, ha az OR nincsen kernel módban.

### 2.MULTIPROGRAMOZÁS. INTERRUPT VEZÉRELT- ÉS IDŐOSZTÁSOS OPERÁCIÓSRENDSZER

**Multiprogramozás:** A MP célja, hogy hatékonyan lehessen kihasználni a hardver (továbbiakban HW) erőforrásokat. Egy felhasználó képtelen optimálisan és állandóan kihasználni a processzort és az I/O eszközöket, így ez a feladat az ORre hárul. Ebben segít az MP, ami megszervezi a feladatok végrehajtását, hogy a processzornak mindig csak egyetlen feladat futtatása juthasson. A feladatok (és ezzel a rendszer) egy részét a memóriában, az összes feladatatot pedig egy pool-ban (job pool) tartja, és az OR ezek közül mindig egyetlen feladatot választ ki, és futtat a processzoron. Abban az esetben, ha egy feladatnak várakoznia kell (pl.: I/O műveletre), addig a rendszer egy másik feladatot választ, és azt futtatja, és később tér vissza a várakozást már befejező feladathoz.

**Interrupt vezérelt- és időosztásos rendszerek**

Az időosztás az MP legfontosabb kiterjesztése, amiben a processzor nem egyetlen folyamatot futtat folyamatosan (annak befejezéséig), hanem az OR folyamatosan váltogat a folyamatok között, mindegyiknek adva egy időszeletet, amíg használhatja a processzort. Ennek a megoldásnak a célja, hogy a felhasználó úgy érzi, hogy minden programja fut és gyorsan reagál, hiába van csak egyetlen processzor a gépében. Ilyen esetekben a programok reakcióideje gyors, jellemzően 1 sec alatti. Az időosztás használatához legalább egy felhasználói program (folyamat) futtatására van szükség a memóriában. Ha egy program nem fér el a rendelkezésre álló memóriában, akkor a swapping technológia segítségével érhető el plusz szabad memóriaterület. A virtuális memória használatával futtathatóak olyan programok, amik nincsenek teljes mértékben a memóriában.

Az időosztás használatánál elengedhetetlen a job pool-ok használata: egy job pool-ban tároljuk az összes, futtatásra váró folyamatot. Ezek egy része a memóriában van, a többi valamilyen háttértáron. A háttértáron lévő folyamatok arra várnak, hogy bekerülhessenek a memóriába, és ott utána processzor időt kapjanak. Ha több folyamat is a memóriába akar kerülni, de nincsen elég szabad hely ehhez, akkor az OR dönti el, melyik folyamatnak ad helyet: ez az eljárás a folyamat ütemezés.

A kiválasztott folyamat betöltődik a háttértárról a memóriába. Ha a memóriában több folyamat is van, akkor a CPU ütemezés segítségével dönti el az OR, hogy éppen melyik kapjon processzor időt.

Az OR-ek megszakítás vezéreltek, ami azt jelenti, hogy a működésüket alapvetően határozzák meg a megszakítások. Egy megszakítás egy esemény bekövetkeztét jelzi, akár egy HW, akár egy SW részéről. Egy HW bármilyen időpillanatban adhat ki megszakítást, egy SW jellemzően rendszerhívás futtatásakor teszi ezt meg. Szinte minden rendszernek saját megszakítás vezérlési felépítése van, de számos közös jellemző határozható meg:

•A megszakításkezelő architektúrának meg kell őriznie a megszakítás címét

•Minden új megszakítás le van tiltva, amíg egy másik megszakítás folyamatban van

•A trap (csapda) egy SW generálta megszakítás, jellemzően hiba vagy felhasználói kérés miatt

A megszakításkezelő egy memóriaterületen őrzi a megszakítások címét, és ezek a címek különböző eszközökhöz tartozó megszakítás rutinokra mutatnak, amik eljárásokat tartalmaznak a megszakítást kérő eszköz számára (ezeket le kell futtatni az adott megszakítás beérkezése esetén). Ezeknek a címeknek a tömbjét szokás megszakítás vektornak nevezni.

A megszakításokhoz kapcsolható a DMA (Direct Memory Access) vezérlő is, ami az I/O eszközök megszakításait kezeli le. Ez átveszi a CPU-ról az I/O eszközök megszakításainak kezelését, és csak akkor küld a CPU-nak megszakítást, ha egy nagyobb művelet fejeződött be. Ez sok munkát megspórol a CPU-nak, mivel nem kell foglalkoznia az I/O rendszer által byte-onként generált megszakításokkal, hanem csak blokkonként eggyel. Ez főleg kis adatmennyiségek mozgatásánál okoz látványos teljesítmény spórolást.

### 3.Mutassa be az alapvető rendszerprogramokat. Ismertesse a rendszerhívásokat

**Alapvető rendszerszolgáltatások**

Egy OR egy környezetet biztosít a felhasználói programok futtatásához. Ez számos szolgáltatást jelent a programok és azok használói számára is. A felhasználók számára biztosított fontosabb szolgáltatások:

•Felhasználói felület: ez lehet parancssoros (Command-line interface, CLI), grafikus (Graphical user interface, GUI), vagy ún. kötegelt interface. Ez utóbbi olyan megoldás, amiben a vezérlő parancsok egy file-ba vannak írva, és ez a file kerül futtatásra. Egy OR akár több féle felületet is nyújthat.

•Programfuttatás: a rendszer képes betölteni programokat a memóriába, és ott képes futtatni őket, és ezek a programok be is tudják fejezni működésüket.

•I/O műveletek: egy futó program végezhet I/O műveleteket, akár egy file-al, akár pedig egy I/O eszközzel kapcsolatban. Jellemzően ezek az eszközök nem érhetőek el közvetlenül védelmi és hatékonysági okokból.

•Filerendszer műveletek: egy alkalmazás végezhessen írási és olvasási műveleteket file-okon és könyvtárakon, tudjon új elemeket létrehozni, vagy törölni régieket. Legyen lehetőség keresésre, és esetleg jogosultságok kezelésére.

•Kommunikáció: bizonyos esetekben legyen lehetőségük különböző folyamatoknak egymás közötti adatcserére. Ez történhet egyetlen gépen (egyetlen OR-en) belül, de akár különböző gépeken, valamilyen hálózaton keresztül is. A kommunikáció történhet osztott memória segítségével is, vagy üzenetátadással, ahol az információ csomagok formájában mozog a folyamatok között.

•Hibakezelés: egy OR alapvető feladata, hogy észrevegye a keletkező hibákat. Hiba előfordulhat HW szinten (CPU-ban, memóriában, I/O eszközökben), vagy SW szinten felhasználói programokban (buffer overflow stb.). Minden hiba esetén az OR-nek képesnek kell lennie kezelnie a hibát, és az annak megfelelő válasz műveletet kell végrehajtania.

A rendszerfunkciók egy másik csoportja, amelyek az OR hatékony működéséért felelnek. Egy több felhasználós rendszer hatékonysága nagyban növelhető az erőforrások jó elosztásával a felhasználók folyamatai között.

•Erőforrásfogalás kezelése: többfelhasználós és több folyamatot kezelő rendszereknél minden folyamathoz különböző erőforrásokat kell rendelni, amik kezelésének a feladata az OR-re hárul. Figyelni kell az egyes erőforrások jellegzetességeire, speciális igényeire, függőségeire. Az OR feladat összehangolni az egyes folyamatokhoz tartozó erőforrás foglalásokat, hogy azok összege optimális legyen a rendszer szempontjából.

•Nyilvántartás (accounting): a felhasználók nyilvántartása, hogy melyikük mikor és mennyi erőforrást használt fel. Ezekből az adatokból készült statisztikák segítségével a rendszer karbantartója olyan módosításokat végezhet a rendszeren, amelyek segítségével növelhető az erőforrások kihasználásának hatékonysága és javítható a szolgáltatások minősége.

•Védelem és biztonság: egy többfelhasználós rendszerben fontos a személyes adatok védelme az illetéktelen felhasználástól. Az egyes párhuzamosan futó folyamatokat olyan módon el kell különíteni, hogy azok ne legyenek képesek megzavarni egymás, vagy az OR működését. A védelem kiterjed a rendszer erőforrásainak védelmére is az ellenőrizetlen felhasználástól. A biztonság kérdése magában foglalja az OR védelmét a külső támadásoktól (pl.: felhasználói beléptetéssel).

**Rendszerhívások**

A rendszerhívások egy felületet biztosítanak az operációs rendszer (továbbiakban: OR) által elérhetővé tett rendszer funkcióknak. Ezek jellemzően C vagy C++ függvények, vagy alacsony szintű feladatok, amiket assembly-ben kell megírni (pl.: device driver). Általában egy magas szintű nyelven megírt API-n keresztül végzi el a program a rendszerhívások meghívását, direkt hívás nem jellemző. Ilyen API a win32, a Posix API vagy a Java API. Ezek az API-k függvényeket biztosítanak a programozóknak bizonyos rendszer szintű feladatok elvégzésére, és a visszatérési értékek segítségével közlik ezek eredményét.

Szinte minden programozási nyelvben van egy rendszer-hívó felület (system-call interface), aminek a feladata összeszedni azokat a hívásokat, amelyek rendszerhívásokat érnek el, és ezek meghívásakor továbbítani a kérést a megfelelő OR rendszerhívásnak. Általában egy szám azonosít minden rendszerhívást, és a system-call felület tartja nyilván az ezekből az azonosítókból készített táblázatot. Ez a felület felel még a rendszerhívások állapotának nyomonkövetéséért, és az eredmény visszajuttatásáért a hívó API metódushoz. A hívónak semmit nem kell tudnia az általa használt rendszerhívás megvalósításáról, csak tartania kell magát az API előírásaihoz, és tudnia kell értelmezni az OR által visszaadott eredményt. Az OR felületének legnagyobb része rejtve van a hívó elől (az API-n keresztül).

Paraméterátadás is megvalósítható a rendszerhívásoknál. A paraméterek típusai és az átadott információ mennyisége változhat az adott OR függvényében. Három jellemző megoldás van a paraméterátadásra:

•A legegyszerűbb a regiszteren keresztüli megoldás. De előfordulhat, hogy több paraméter van, mint rendelkezésre álló regiszter.

•Ha felmerül a regisztereknél leírt probléma, akkor használható a blokkos megoldás: a paraméterek egy blokkban vagy táblában tárolódnak a memóriában, és ennek a blokknak a címe kerül átadásra. Ez a megoldás jellemző Linuxon.

•Megoldás lehet verem használata, amiben a hívó program belerakja a paramétereket, az OR pedig kiveszi onnan azokat.

A blokkos és a vermet használó megoldásban nem limitált az átadható paraméterek száma.

**Rendszerhívások típusai**

A rendszerhívásokat öt nagy csoportba lehet sorolni:

•Folyamat vezérlés: ide tartoznak azok a parancsok, melyek egy folyamat futását befolyásolják:

oProcess befejezése (end, abort)

oProcessz indítása, futtatása (load ,execute)

oÚj folyamatot létrehozása, vagy futó folyamat megszüntetése (create/terminate process)

oFolyamat attribútumainak lekérdezése és beállítása (get/set process attributes)

oFolyamat várakoztatása adott ideig (wait for time)

oFolyamat várakoztatása egy eseményre (wait event)

oMemória lefoglalása és felszabadítása (allocate/free memory)

•File kezelés

oFile létrehozása, törlése (create/delete file)

oFile megnyitása, bezárása (open, close)

oFileból olvasás, fileba írás, file pointer pozícionálás (read, write, reposition)

oFile attribútumainak lekérdezése és beállítása (get/set file attributes)

•Eszköz kezelés

oEszköz foglalás kérése, eszköz felszabadítása (request/release device)

oEszközről olvasás, eszközre írás (read, write)

oEszköz attribútumainak lekérdezése és beállítása (get/set device attributes)

oEszközök logikai fel- és lecsatlakoztatása (logically attach/detach device)

•Információ karbantartás

oIdő és dátum lekérdezése és beállítása (get/set time/date)

oEszköz, file vagy folyamat attribútumainak lekérdezése és beállítása (get/set file/device/process attributes)

•Kommunikáció

oKommunikációs kapcsolat létrehozása, törlése (create/delete communication connection)

oÜzenet küldése, fogadása (send/recieve message)

oTávoli eszközök fel- és lecsatlakoztatása (attach/detach remote device)

**Rendszerprogramok**

A rendszerprogramok egy kényelmes környezetet biztosítanak programok fejlesztéséhez és futtatásához. Egy részük egyszerű felület rendszerhívásokhoz, míg mások sokkal összetettebb feladatokat látnak el. Több kateg. lehet sorolni őket:

•File kezelés: ezen programok segítségével lehet file-t létreho, törölni, másolni, átnevezni, listázni stb.

•Állapot információk: az egyszerűbb programok egyszerűen lekérdezik a rendszeridőt, dátumot, az elérhető szabad memóriát vagy tárhelyet, és hasonló információkat. Az összetettebbek már teljesítmény adatokat, logokat vagy debug információkat szolgáltatnak. Jellemzően ezek a programok megformázzák és kiírják a képernyőre (vagy más kimeneti eszközre) a kért adatokat. Bizonyos rendszerekben van registry, ami beállítási információkat tárol.

•File módosítás: számos szövegszerkesztő lehet elérhető egy rendszerben, aminek a segítségével létrehozható file, és módosítható annak tartalma. Rendszerint vannak parancsok file-ok tartalmában való keresésre és szöveg átalakításra.

•Programozási nyelvek támogatása: fordítók, debuggerek, assemblerek és interpreterek a legtöbbet használt nyelvekhez számos operációs rendszerben elérhetőek a felhasználók számára.

•Program betöltés és futtatás: egy program futtatásához azt be kell tölteni a memóriába. Ehhez egy rendszer adhat segítséget különböző loader-ek formájában. Különböző szintű nyelvekhez tartozó debuggerek is állhatnak rendelkezésre.

•Kommunikáció: ezen programok segítségével létrehozhatóak virtuális kommunikációs csatornák, folyamatok, felhasználók vagy számítógép rendszerek között. Ezek segítségével üzenetek küldhetők, fogadhatók, file-ok vihetőek át stb.

### 4.Virtuális gépek elv, típu. Használatának előnyei

**Virtuális gép**

Egy virtuális gép célja, hogy absztrakt módon kezelje egy számítógép hardverét (továbbiakban: HW), és ezt több futtató környezetnek szétossza, ezzel olyan érzést keltve, mintha több futtató környezet működne egyetlen számítógépen. Ez a megoldás hasonlít a **3.** tételben bemutatott réteges felépítésre. Az OR a CPU ütemezés és a virtuális memória használatával képest azt az érzést kelteni, hogy egy futó folyamatnak van saját processzora és memóriája. Egy "normál" folyamat számos funkcióval rendelkezik, mint a file rendszer elérése vagy rendszer hívások. A virtuális gépek ezekkel nem rendelkeznek, helyette viszont van egy felület, ami megegyező a számítógép HW-ével.

Virtuális gépek használatának a legfőbb oka, hogy meg lehessen osztani egyetlen HW-t több OR között. A legnagyobb gond ilyen helyzetekben a lemezkezeléssel van. A fizikai lemezeket lehetetlen jól elosztani a virtuális gépek között, főleg ha több virtuális gép van, mint ahány lemez. Erre megoldás a virtuális lemezek (minidiszk) használata, amik megfelelnek a fizikai lemezeknek, kivéve a méretüket. Az OR a minidiszknek megfelő méretű helyet lefoglal a fizikai lemezen, és arra kerül majd a virtuális OR. Így akár minden felhasználó futtathatja a saját OR-ét, miközben az összes rendszer ugyanazon az egy gépen fut.

A virtuális gépek implementációja meglehetősen nehéz és bonyolult. A legnagyobb munka egy pontos másolatát elkészíteni az aktuális HW-nek. A virtuális OR-nek is tudnia kell kernel módban futni mivel OR, és kell tudnia felhasználói módban futtatnia programot. Maga a virtuális OR csak felhasználói módban tud futni a futtató OR-en, így szükséges használni virtuális kernel és felhasználói módot a futtató OR-en, a normál felhasználói módon belül. Ezek segítségével már elérhetővé válik a virtuális OR számára a futtató gép kernel módja. Ennek a megoldásnak a legnagyobb problémája az idő: az egyes rendszerhívások lényegesen tovább tartanak virtuális gépeknél, mint normál esetben, ez főleg I/O műveleteknél érezhető.

**Virtuális gépek előnyei**

Számos előnye van a virtuális gépek használatának. Biztonsági szempontból nagy előny, hogy minden rendszer teljesen elkülönítve fut egymástól, nem tudnak kommunikálni egymással, így nem is képesek egymás munkáját megzavarni, megszakítani. Ha az egyik virtuális gép elszáll, a többit ez nem befolyásolja. Emiatt semmilyen erőforrás megosztás nem lehetséges a gépek között, így erre más megoldást kell találni. Az első egy minidiszk megosztása minden virtuális gép között, a második pedig, hogy valamilyen hálózati kapcsolatot használva lehet erőforrásokat közösen használni.

Másik nagy előnye a virtuális gépeknek, hogy egyszerűsítik a fejlesztési és tesztelési feladatokat. Egy OR fejlesztése során, ha megváltozik a rendszer, akkor a teljes gépet újra kell indítani, és alaposan kell tesztelni a változásokat. Ha mindezt egy fizikai gépen tesszük, annak a költsége igen magas lehet, hiszen sok időt tölt el a gép érdemi munka nélkül (boot-olás, debug stb). Ha egy fejlesztő virtuális gépeken dolgozik, a hatékonyság hatványozottan növekedhet.

**Virtuális gépek típusai**

Többféle virtuális gép megközelítés létezik, ebből kettőt emelnék ki: a VMWare-t és a Java Virtual Machine-t.

**VMWare:** A VMWare Intel x86-os gépeket emulál. Szimpla alkalmazásként fut a futtató OR-en, és egyszerre akár több vendég OR-t enged futtatni, mint független virtuális gépeket. A VMWare kifejezetten jól alkalmazható pl. olyan esetben, ha egy fejlesztő egy általa készített szoftvert több OR-en is tesztelni szeretne. Ebben az esetben nincs szüksége annyi gépre, ahány rendszer kell neki, elég feltelepítenie az összeset virtuális gépként, és a saját gépén meg tud oldani mindent, akár egyszerre, párhuzamosan egymással. A VMWare egy absztrakciós réteget képez a host OR és a virtuális gépek közé, ami egy virtuális HW szintet ad minden virtuális gépnek. Ezen keresztül a virtuális OR-ek elérhetik a futtató gép minden erőforrását.

**Java Virtual Machine:** A Java VM egy absztrakt virtuális gép specifikáció. Egy class loaderből és egy interpreterből áll, ami a Java bytecode-ot futtatja. A class loader tölti be a lefordított class állományokat, és adja át őket az interpreternek futtatásra. Egy class betöltése után ellenőrzésre kerül, hogy nem fog-e a futtatása verem alul- vagy túlcsordulást okozni, vagy nem fog-e illegális pointer aritmetikát alkalmazni. Ha itt megfelel a class, akkor az interpreterhez kerül futtatásra. A Java VM automatikusan kezeli a memória felhasználást a garbage collector segítségével.

A Java VM bármilyen host OR-en képes futni, amire elérhető implementációja. De akár HW-es formában is implementálható egy chip formájában, ami kifejezetten Java programok futtatására alkalmas.

### 5.Folyamat állap, állapk közötti átmen jell, kivált

**Folyamat**

Általánosan, egy folyamat egy futó program. De egy folyamat több, mint szimpla program kód: hozzá tartozik még az aktuális aktivitás jellemzője is (program számláló, processzor regiszter állapotok). Egy folyamat általában tartalmaz egy stack-et is, amiben az ideiglenes adatokat tárolja, és van egy adat szegmense, amiben a globális változóit tartja. Esetenként még lehet heap része is, ami egy dinamikusan lefoglalt memória rész. Egy program önmagában még nem folyamat, csak akkor válik azzá, amikor futtatásra betöltődik a memóriába.

**Folyamat állapota**

**Folyamat állapot átmenetek**

Egy folyamat a végrehajtása során különböző állapotok vesz fel, amit a folyamat aktuális tevékenysége határoz meg. Öt állapot lehetséges:

•Új: az új folyamat létrehozás alatt áll

•Futó: a prog utasításai végrehajtás alatt (a prog fut)

•Várakozó: a folyamat egy esemény bekövetkezésére vár (pl.: I/O műveletre)

•Kész: a foly várja, hogy processzoridőhöz jusson

•Véget ért: a folyamat befejezte a végrehajtást

Egy processzoron egy időpillanatban mindig csak egyetlen futó folyamat lehet. Várakozó és kész állapotú folyamatból több is lehet egy időben.

Folyamat leíró tábla

Minden folyamat tartalmaz egy ilyen leíró táblát, ami számos adatot tárol az adott folyamatról. A fontosabb részei:

•A folyamat állapota

•A programszámláló: a következő végrehajtandó utasítás címét tartalmazza

•CPU regiszterek:az aktuális architektúrától függ a felépítésük, de jellemzően tartalmaznak akkumulátort, index regisztert, stack pointereket, általános regisztereket. Megszakításkor a programszámlálóval együtt ennek a tartalmát is el kell menteni

•CPU ütemezési információk: itt találtó a folyamat prioritása, mutatók az ütemezési sorokra stb.

•Memóriagazdálkodási információk: itt találhatóak különböző regiszterek értékei, memórialap és -szegmens táblák

•Adminisztrációs információk: a felhasznált CPU idő, időkorlátok, folyamat és feladat azonosítók stb.

•I/O állapot információk: a folyamathoz kapcsolt I/O eszközök listája, megnyitott file-ok listája stb.

Összefoglalva, a folyamat leíró tábla egyfajta regisztrációs adatbázisként szolgál minden, a folyamathoz tartozó információ számára.

Folyamatok közötti váltás

**Processzek közti váltás**

### 6.Jellemezze a folyamatközi kommunikációt és alapvető megvalósítási lehetőségeit

Folyamatközi kommunikáció

Két folyamat egymással két féle kapcsolatban állhat: vagy függetlenek egymástól, ekkor semmilyen módon nem tudják egymás állapotát befolyásolni, vagy pedig együttműködnek, ekkor viszont befolyásolhatják egymás állapotát. Számos előnye van a folyamatok együttműködésének:

•Információ megosztása: egyszerre több felhasználónak is szüksége lehet ugyanarra az információra (pl.: egy file tartalma)

•Számítási sebesség növelése: egy feladat felosztható kisebb részfeladatokra, amiket aztán párhuzamosan futtatunk le egymással. Ez csak akkor ad érdemi sebességnövekedést, ha több számoló egység van a gépben

•Modularitás: egy rendszer modulokra osztható fel, a rendszerfunkciók feloszthatóak folyamatokra és szálakra

•Kényelem: független felhasználók több különböző feladaton dolgozhatnak egy időben

Az együttműködéshez szükséges valamilyen folyamatközi kommunikáció (Interprocess Communication, IPC), amivel adatok cserélhetőek ki. Két jellemző megoldás van erre, az egyik a osztott memória alapú megoldás, a másik az üzenet alapú kommunikáció. Az osztott memóriás megoldás gyors (a számítógép memóriájának sebességével működik) és kényelmes. Az üzenet alapú megoldás kis adatok mozgatásakor előnyös, és egyszerűbb implementálni, mint az osztott memóriás megoldást.

Osztott memóriás kommunikáció

Minden folyamat egy közös memóriaterületen keresztül kommunikál, nincs direkt folyamatközi kommunikáció. A közös memóriaterület az azt létrehozó folyamat címtartományában jön létre, és ezt kell elérnie a saját címtartományából a többi folyamatnak. Viszont az operációs rendszerek (továbbiakban: OR) általában megpróbálják megakadályozni, hogy egy folyamat egy másik folyamat memóriarészét elérhesse, így egy külön "megállapodás" szükséges minden résztvevő process részéről, hogy elérhetik a közös memóriát.

Az adatok formája kizárólag a részt vevő folyamatokon múlik, az OR semmilyen formában nem szól ebbe bele. A folyamatok feladata azt biztosítani, hogy nem írnak párhuzamosan ugyanarra a memóriarészre. Ez a legnagyobb probléma ezzel a megoldással, hogy számos biztonsági elemet kell implementálni a megbízható működéséhez, mivel ennek a felügyelete kivül esik az OR hatáskörén.

Egy jellemző probléma, amire megoldás az osztott memóriás kommunikáció, a termelő-fogyasztó probléma. Egy lehetséges megoldásban egy közös buffert használunk (ez az osztott memória), abba és abból dolgozik mindkét (három, négy stb.) folyamat. A buffer rendelkezhet valamilyen méret korlátozással (bounded buffer), vagy lehet korlátozás nélküli (unbounded buffer).

Üzenet alapú kommunikáció

Az üzenet alapú kommunikációban az egyes folyamatok egy üzenetküldő központon keresztül kommunikálnak. Ennél a megoldásnál a folyamatok úgy tudnak adatokat cserélni, hogy nincsen szükség egy közös címtartományra. Ez hasznos lehet elosztott rendszerek esetében, ahol minden adatcsere egy hálózati kapcsolaton át történik, és lehetetlen közös címtartományt kialakítani. Egy üzenetküldő felület legalább két alapvető eljárást biztosít: a send()-et, és a recieve()et. A küldött adat mérete lehet fix vagy változó. Fix méret esetén a rendszer szintű megvalósítás magától értetődő, viszont az alkalmazási szinten nehezebb. Változó méret esetén a helyzet pont megfordítva igaz.

A kommunikációhoz szükség van valamilyen kommunikációs kapcsolatra a két folyamat között. Ezekben a kapcsolatokban nem annyira a fizikai megvalósítás az érdekes, hanem a logikai kivitelezés.

Név alapú kommunikáció

Direkt kommunikáció

Itt minden folyamatnak egyértelműen meg kell jelölnie címzettet és/vagy feladót egy üzenetben. Itt a send() és a recieve() eljárások két paramétert fogadnak: a címzett és az adat. A kommunikációs kapcsolat a következő tulajdonságokkal rendelkezik:

•A kapcsolat automatikusan jön létre a két kommunikálni akaró folyamat között. Egy folyamatnak csak a másik folyamat azonosítóját kell ismernie kapcsolat kezdeményezéséhez

•Egy kapcsolat kizárólag két folyamat között jöhet létre

•Minden folyamatpár között kizárólag egy kapcsolat lehet

Ez a felépítés szimmetrikus címzést valósít meg. Ennek egy asszimetrikus változata, amikor csak a küldő nevezi meg a címzettet, a fogadó folyamat csak egy változó értékét állítja be a küldő folyamat azonosítójára.

Ezeknek a megoldásoknak a hátránya a modularitás hiánya, és ez egy eléggé kódba bebetonozott megoldás, mivel az azonosítókat mindenhol explicit meg kell adni.

Indirekt kommunikáció

Itt az üzenetek egy postafiókon keresztül kerülnek kézbesítésre. Egy postafiók olyan hely, ahová egy folyamat berakhat adatokat, és egy másik folyamat pedig kiveheti azokat. Egy folyamat több másik folyamattal több postafiókon keresztül tud kommunikálni. Két folyamat csak akkor tud adatot cserélni, ha van egy közös postafiókjuk. A send() és a recieve() primitívek két paramétert fogadnak: a postafiók azonosítóját és az adatot. A indirekt kommunikáció jellemzői:

•Egy folyamat pár között csak akkor jön létre kapcsolat, ha van egy közös postafiókjuk

•Egy kapcsolat létrejöhet kettőnél több folyamat között is

•Két kommunikáló folyamat között lehet több különböző kapcsolat is, mindegyik kapcsolathoz külön postafiókkal

Egy postafióknak két tulajdonosa lehet: egy folyamat, vagy az OR.

Szinkronizáció

A kommunikáció kétféle lehet: szinkron vagy aszinkron. Ha a kommunikációs csatornának nincs kapacitása (egy folyamat egy üzenet elküldése után addig nem küldhet új üzenetet, amíg az előző célba nem ért), akkor a kapcsolat szinkron. Ha van kapacitása (a kommunikációs csatorna akkor blokkolódik, ha elfogy a kapacitás), akkor a kapcsolat aszinkron. Ennek megfelelően két-két fajtája lehetséges a send() és a recieve() primitíveknek: blocking és nonblocking. A blocking esetében a folyamat blokkolja a kapcsolat forgalmát mindaddig, amíg az üzenet továbbítása tart. Nonblocking esetén viszont elindítja az üzenet kézbesítését, majd folytatja a futását.

**7.Ismertesse a folyamatok ütemezésének fontosságát, alapelveit. Mutassa be a nem preemptív ütemezőket**

**A folyamatok ütemezése**: A multiprogramozott rendszerek lényege, hogy a lehető legjobban ki legyenek használva a számítógép erőforrásai, köztük a CPU. Az időosztásos technológia célja, hogy folyamatosan váltogatva az éppen futtatott programot azt az érzést keltse, hogy minden program folyamatosan fut. Ennek megoldásához szükség van egy folyamat ütemezőre, ami eldönti, hogy éppen melyik folyamatot futtassa a CPU. Egy rendszerben ez a tényező befolyásolja a felhasználói elégedettséget a leginkább, mivel itt dől el, mennyire lesz olyan érzete a felhasználónak, hogy minden programja párhuzamosan fut.

**ütemezési sorok:** Amikor egy új folyamat belép a rendszerbe, bekerül a feladatsorba (job queue), ami minden folyamatot tartalmaz a rendszerben. Azok a folyamatok, amik be vannak töltve a memóriába és futásra kész állapotban vannak, a ready queue-ban vannak. Ez általában egy láncolt lista. Ezek mellett még számos más sor is található egy rendszerben: ilyen lehet az eszközvárakozási sor, ahova azok a folyamatok kerülnek be, amik egy megadott I/O eszköz használatára várnak. Minden eszköznek saját várakozási sora van.

Egy új folyamat jellemzően a ready queue-ba kerül be, és ott várakozik, amíg futtatásra nem kerül. Amikor egy folyamat hozzájut a CPU-hoz, a futtatása során többféle esemény is felléphet: •A folyamat küldhet I/O kérést, és bekerülhet egy eszközvárakozási sorba •A folyamat létrehozhat egy új alfolyamatot, és várakozhat az alfolyamat befejeződésére •A folyamat egy megszakítás hatására el lehet távolítva a CPU-n futtatásból, és így visszakerül a ready queue-ba

**Ütemezők:** Egy folyamat számos ütezemési sorban megfordul a teljes életciklusa során. Az OR mindig kiválaszt folyamatokat az egyes sorokból valamilyen elgondolás alapján. A választást a megfelelő ütezemőkkel végzi el. Két jellemző típusa van: •Hosszú távú ütemező (Long-term scheduler, HTÜ): ez választja ki a feladatsorból a folyamatokat, és teszi át őket a memóriába végrehajtásra •Rövid távú ütemező (Short-term schedulerm, RTÜ): ez dönti el, hogy a ready queue elemeiből éppen melyik folyamat használhassa a CPU-t

A legfőbb különbség a két ütemező között a végrehajtás frekvenciájában van. A RTÜ sűrűn kell, hogy kijelöljön munkát a CPU számára. Egy folyamat adott esetben csak néhány msec-ig fut, pl.: míg kér egy I/O műveletet, és utána új folyamatot kell futtatni. Rendszerint az RTÜ lefut minden 100msec-ban, emiatt viszont gyorsnak kell lennie, hogy ne vigyen el sok időt maga az ütemezés egy 100msec-es időszeletből.

A HTÜ sokkal ritkábban kell, hogy lefusson, akár több perc különbség is lehet két futása között. A HTÜ szabályozza egy OR multiprogramozottságának fokát (a folyamatok száma a memóriában). Ha ez az érték állandó, akkor átlagosan ugyanannyi folyamat fejeződik be adott idő alatt, mint amennyi új jön létre. A HTÜ-t csak egy folyamat befejeződésekor kell meghívni. Mivel két futása között hosszabb ideje van az ütemezőnek, így lehetősége van alaposabb döntést hoznia a folyamatok kiválasztásában. A HTÜ-nél fontos is az alapos és megfontolt döntés. Egy folyamatot általában az I/O vagy a CPU használata felől jellemezhetünk. Egy folyamat I/O-hoz kötött, ha több időt tölt I/O műveletekkel, mint számítással. Egy másik folyamat pedig CPU-hoz kötött, ha inkább számol, és I/O műveletet ritkábban végez. A HTÜ esetében az a fontos, hogy egy jó keveréket állítson össze a kétféle típusú folyamatokból.

Több rendszer esetében a HTÜ csak minimális, vagy egyáltalán nincs is, helyette minden új folyamatot azonnal a memóriába tesznek, és a RTÜ-re bíznak. Ezeknek a rendszereknek a megbízhatósága a hardver fizikai korlátain és a felhasználók alkalmazkodóképességén múlik (ha lassú a gép, valaki kilép, és a többieknek gyorsul a rendszer).

Más időosztásos rendszerek alkalmaznak egy középszintű ütemezőt, aminek a lényege, hogy bizonyos esetekben hasznos egy folyamatot eltávolítani a memóriában, ezzel csökkentve a multiprogramozottság fokát. Később ez a folyamat visszakerülhet a memóriába, és ott folytatódhat, ahol korábban abbamaradt. Ez a megoldás a swappelés.

**A CPU-I/O burst ciklus:** Egy folyamat végrehajtása két részből áll, amik ciklikusan ismétlődnek: egy CPU végrehajtási részből, és egy I/O várakozási részből. Egy foly. e között a két rész között váltogat folyamatosan. A foly. végrehajtása egy CPU burst-el indul, ezt követi egy I/O burst, és ezek ismétlődnek folyamatosan. A legutolsó CPU burst vet véget a folyamat futásának. Egy CPU burst hossza folyamatonként és rendszerenként változhat, de általában egy jellemző frekvencia görbével leírhatók. A görbe rendszerint exponenciális, sok rövid és kevés hosszú CPU burst-el. Egy I/O intenzív folyamat jellemző sok rövid CPU burst-et használ, míg egy CPU intenzív pedig keveset, de hosszút. Ezek eloszlása fontos lehet a megfelelő CPU üzemező algoritm. kiválasztása során.

**A CPU ütemező:** Amikor egy CPU-nak nincs munkája, az OR-nek ki kell választania egy új folyamatot, ami végrehajtható állapotban van. Ezt a RTÜ végzi el. A memóriában található folyamatokat tartalmazó ready queue nem feltétlenül egy FIFO sor. A különböző ütemező algoritmusoknak megfelelően ez lehet prioritásos sor, fastruktúra, de akár egy rendezetlen lista is. A közös, hogy a sorban fel van sorolva minden olyan folyamat, ami futtatható állapotú. A különböző algoritmusok különböző kritériumok és célok szerint működhetnek. Jellemző feltételek egy algoritmus kritériumán belül: •CPU kihasználtság: a cél a CPU-t a lehető legnagyobb mértékben terhelni •Teljesítmény: az adott idő alatt elvégzett munkaegységek száma legyen a lehető legtöbb •Lefutási idő: figyelembe kell venni egy folyamat teljes lefutási idejét, a várakozási sorba kerüléstől a befejezéséig •Várakozási idő: az idő, amit egy folyamat a várakozási sorban tölt el •Válaszidő: az idő, ami eltelik, mire egy folyamat elkezd válaszolni. Ez nem azonos a lefutás idővel!

**Nem preemptív ütemezés:** Egy ütemező esetében akkor beszélünk nem preemptív (kooperatív) ütemezésről, ha az alábbi két feltétel teljesülése esetén van szükség CPU ütemezésre: •Amikor egy foly. futó állapotból várakozó állapotba vált •Amikor egy folyamat kilép

Ezekben az esetekben nincs döntési lehetőség az ütemezés feltételeit illetően. Nem preemptív ütemezés esetén, ha a CPU ki lett jelölve egy foly. számára, akkor a folyamat egészen addig magánál tartja a CPU használat jogát, amíg ki nem lép, vagy várakozni nem kényszerül.

**First-Come-First-Serve algoritmus:** A FCFS ütemező algoritmus lényege, hogy a folyamatok érkezési sorrendben hajtódnak végre. Itt a ready queue egy FIFO sor, aki előbb bekerül, az előbb is kerül ki. Amikor egy folyamat futtatásra kerül, addig használja a processzort, amíg szüksége van rá. Emiatt a sorban hosszú várakozási idők is lehetnek. Ilyen ütemezésnél nehéz optimálisan elosztani az erőforrásokat a CPU és az I/O intenzív folyamatok között, és szinte mindig lesz olyan időszak, amikor vagy a CPU, vagy az I/O egység fog üresen állni. Itt jellemző a konvoj effektus, amikor minden folyamat egyetlen nagy folyamatra vár, hogy az felszabadítsa a CPU-t. Ennek az eredménye alacsonyabb CPU és I/O kihasználtság. Időosztásos rendszereken problémás az FCFS algoritmus használata, mivel túl nagy időkre lefoglalhatja a CPU-t egy folyamat, és addig az egész rendszer is áll.

**Shortest-Job-First algoritmus:** Az SJF ütemező algoritmus lényege, hogy a várakozási sorból mindig a legkisebb következő CPU burst-el rendelkező folyamatot választja ki, és futtatja a processzoron. Az algoritmus bizonyítottan optimális abból a szempontból, hogy a legkisebb átlagos várakozási időt adja egy folyamatcsoportnak. Egy rövid folyamat mozgatása egy hosszabb folyamat elé nagyobb mértékben csökkenti a rövid folyamat várakozási idejét, mint amennyivel növeli a hosszabbét.* Az algoritmus legnagyobb problémája a CPU burst hosszának meghatározása. Mivel a pontos hossz meghatározása egy rendszer válaszideje szempontjából fontos lehet, így az SJF használata HTÜ-ben jellemző. Itt elég egy becslést adni egy időszükséglet hosszára, nem kell pontos értékeket számolni. A RTÜ-ben nincs mód használni az SJF-et, mivel nincs mód arra, hogy egy CPU burst hosszát meghatározhassuk, legfeljebb megjósolni lehet bizonyos valószínűséggel.

Az SJF lehet preemptív is, a különbség egy új folyamat érkezésekor van: ha az új folyamat rövidebb, mint a jelenleg futó, akkor preemptív esetben az új folyamat azonnal futtatásra kerül, míg nem preemptív esetben meg kell várni, míg az éppen futó folyamat befejeződik.

**Prioritásos algoritmus:** Az SJF egy speciális változata, ahol az ütemezés alapját a folyamatokhoz rendelt prioritások adják, és nem a CPU burst hosszak. Egyenlő prioritású folyamatok az FCFS algoritmus alapján kerülnek sorra. Ez az algoritmus is lehet preemptív, hasonlóan az SJF-hez. * Az algoritmus legnagyobb problémája a kiéheztetés esete: előfordulhat, hogy egy alacsony prioritású folyamat soha nem kerül sorra, mivel mindig van előtte egy magasabb prioritású. Két eset fordulhat ilyenkor elő: a folyamat addig vár, amíg végre sorra nem kerül, vagy az OR összeomlik, és elveszik az összes alacsony prioritású folyamat.

**8.Mutassa be a preemptive ütemezőket és a többszintű ütemezést**

**Preemptív ütemezés:** Az alábbi négy feltétel közül az első és a negyedik teljesülése esetén beszélünk nem preemptív ütemezésről, minden más esetben preemptív ütemezés valósul meg: •Egy folyamat futó állapotból várakozó állapotba vált (pl.: I/O várakozás miatt)
•Egy folyamat futó állapotból kész állapotba vált (pl.: interrupt esetén) •Egy folyamat várakozó állapotból kész állapotba vált (pl.: I/O művelet elkészülte esetén)
•Egy folyamat befejeződik

Bizonyos platformokon nem lehetséges preemptív ütemezés használata, mivel speciális ütemező vagy időzítő hardverre van hozzá szükség. A preemptív ütemezés további problémája jelentkezik megosztott adatok esetén is: előfordulhat, hogy míg egy folyamat éppen frissíti az adatokat, az ütemező addig egy másik folyamatot (ami szintén használja a közös adatokat) kezd futtatni a CPU-n, ami viszont egy inkonzisztens állapotban fogja elérni a közös adatterületet. Ilyen esetekre egy koordináló eljárás szükséges.*A preemptív megvalósítás befolyásolja az OR tervezését is. Figyelembe kell venni, hogy egy rendszerhívást kérő folyamat esetében, ha a rendszerhívás lefutása közben kerül ki a CPU-ból a folyamat, akkor a kernel által végzett tevékenység félbe szakadna, nem lenne hova visszaadni az eredményt, és a káosz garantálva lenne. Erre bizonyos OR-ek azt a megoldást adják, hogy egy rendszerhívás lefutása alatt nem távolítanak el egyetlen folyamatot sem a CPU-ból. Ezzel biztosított a kernel konzisztens állapota, viszont real-time és több processzoros rendszerek esetében ez nem használható megoldás.* Megszakítások az OR-ben bármikor előfordulhatnak, és nem minden esetben van lehetősége a kernelnek figyelmen kívül hagynia egy megszakítási kérelmet, ezért a kernel megszakításokkal befolyásolható részeit védeni kell a párhuzamos használattól. Ez megoldható olyan módon, hogy egy megszakítás érkezésekor tiltanak minden további megszakítást, majd feldolgozás után újra lehet engedni a beérkező interruptokat.

**Shortest-Job-First algoritmus:** Az SJF ütemező algoritmus lényege, hogy a várakozási sorból mindig a legkisebb következő CPU burst-el rendelkező folyamatot választja ki, és futtatja a processzoron. Az algoritmus bizonyítottan optimális abból a szempontból, hogy a legkisebb átlagos várakozási időt adja egy folyamatcsoportnak. Egy rövid folyamat mozgatása egy hosszabb folyamat elé nagyobb mértékben csökkenti a rövid folyamat várakozási idejét, mint amennyivel növeli a hosszabbét.*Az algoritmus legnagyobb problémája a CPU burst hosszának meghatározása. Mivel a pontos hossz meghatározása egy rendszer válaszideje szempontjából fontos lehet, így az SJF használata hosszú távú ütemezőkben jellemző. Itt elég egy becslést adni egy időszükséglet hosszára, nem kell pontos értékeket számolni. Egy rövid távú ütemezőben nincs mód használni az SJF-et, mivel nincs lehetőség arra, hogy egy CPU burst hosszát meghatározhassuk, legfeljebb megjósolni lehet azt bizonyos valószínűséggel.

Az SJF lehet preemptív és nem preemptív is, attól függően, hogy hogyan viselkedik olyan esetekben, ha egy új folyamat következő CPU burst-je kisebb, mint az éppen futó folyamaté. Ha hagyja lefutni az aktuális folyamatot, akkor nem preemptív, ha viszont megszakítja az éppen futó folyamatot, és az újat kezdi futtatni helyette, akkor preemptív.

**Prioritásos algoritmus:** Az SJF egy speciális változata, ahol az ütemezés alapját a folyamatokhoz rendelt prioritások adják, és nem a CPU burst hosszak. Egyenlő prioritású folyamatok az FCFS algoritmus alapján kerülnek sorra. Ez az algoritmus is lehet preemptív és nem preemptív is, hasonlóan az SJF-hez. Az algoritmus legnagyobb problémája a kiéheztetés esete: előfordulhat, hogy egy alacsony prioritású folyamat soha nem kerül sorra, mivel mindig van előtte egy magasabb prioritású. Két eset fordulhat ilyenkor elő: a folyamat addig vár, amíg végre sorra nem kerül, vagy az OR összeomlik, és elveszik az összes alacsony prioritású folyamat.

**Round-robin algoritmus:** Ez az algoritmus kifejezetten időosztásos rendszerekhez lett tervezve. Hasonló az FCFS algoritmushoz, de preemptív képességet adtak a folyamatok közötti váltáshoz. Emellett bevezettek egy időegységet, a time quantum-ot vagy időszeletet, értéke általában 10 és 100msec között van. A ready queue megvalósítása egy körkörös sorhoz hasonló, és az ütemező folyamatosan megy végig a soron, és minden folyamatnak ad egy időszeletnyi CPU időt.

Az RR ütemező implementálásához egy FIFO sorra van szükség. Minden új folyamat a sor végére kerül. A processzor mindig veszi az első folyamatot a sorból, beállít egy megszakítást egy 1 időszeletnyi időzítővel, majd hagyja futni a folyamatot. A futás során két dolog történhet: ha az időszeletnél kevesebb idő alatt végez a folyamat, akkor a CPU felszabadítja saját magát, és az ütemező már adja is a következő folyamatot. Ha a folyamat nem végez az adott időszelet alatt, akkor az időzítő lejárta után megszakítás történik, és a folyamat bekerül a ready queue végére, a sor elejéről pedig egy folyamat elkezd futni. Az RR ütemező teljesítménye erősen függ az időszelet nagyságától. Ha extrém nagy egy időszelet, az RR ütemező szinte teljesen az FCFS algoritmussal egyenértékűen fog működni. Ha az időszelet extrém kicsi, akkor az RR algoritmus a processzor megosztás elvét valósítja meg, azaz elméletileg n folyamat esetén úgy tűnik, hogy minden folyamatnak saját processzora van, 1/n-ed processzor teljesítménnyel.

**Többszintű ütemezés:** A többszintű ütemezés jól használható olyan helyeken, ahol a folyamatok egyszerűen besorolhatóak különböző csoportokba. Jellemző különbségtétel a háttérben (background) és az előtérben (foreground) futó folyamatok megkülönböztetése. Ezek a folyamatok különböző válaszidő elvárásokkal, és így különböző ütemezési elvárásokkal rendelkeznek. A többszintű ütemezés a ready queue-t több, egymástól elválasztott sorra osztja fel. Egy folyamat csak egyetlen sorba kerülhet be (és onnan másikba át nem mozoghat), és a folyamat valamilyen tulajdonsága alapján kerül besorolásra valamelyik sorba. Minden sornak megvan a maga ütemező algoritmusa, és ezek nem feltétlenül azonosak. Szükség van még az egyes sorok között valamilyen sorrend felállítására, általában prioritások megadásával.

**Többszintű, visszacsatolásos ütemezés:** A szimpla többszintű ütemezéshez képest annyi plusz van, hogy egy folyamat itt mozoghat az egyes sorok között. Az elgondolás alapja, hogy a folyamatok CPU felhasználása alapján sorolódnak be egy-egy sorba, és ha egy folyamat az adott sorhoz rendelt határnál több CPU időt használna fel, akkor bekerül egy kisebb prioritású sorba. Ezzel a megoldással az I/O intenzív és az interaktív folyamatok magas prioritással működhetnek, míg a CPU intenzíveknek várniuk kell kicsivel többet a sorukra. Ha egy folyamat túl hosszú ideje lenne egy sorban, akkor az ütemező áthelyezi egy magasabb prioritású sorba, így kerülve el a kiéheztetés esetét. Általánosságban a következő paraméterek határoznak meg egy többszintű, visszacsatolásos ütemezőt: •A sorok száma •Soronként a használt ütemező algoritmusok •Egy folyamat magasabb prioritási szintű sorba emelésének eljárása •Egy folyamat alacsonyabb prioritási szintű sorba süllyesztésének eljárása •Egy folyamat bizonyos sorba besorolásának eljárása, amikor a folyamatnak egy eszközre van szüksége Jelenleg ez a megoldás a legtöbbet használt CPU ütemező. Jól beállítható különböző helyzetekhez, viszont az algoritmusa elég komplex.

**9.Ismertesse a kritikus szekció és a versenyhelyzet fogalmát. Milyen megoldási lehetőségek állnak a rendelkezésre ezek feloldására? Mutassa be Peterson megoldását**

Kritikus szekció

Adott egy n folyamatot tartalmazó rendszer, és minden folyamatban található egy kódrészlet, amiben a folyamat változókat állít be, táblákat frissít, file-t ír ki stb. Ez a kódrészlet a kritikus szekció. A lényeg ebben, hogy amíg egy folyamat a saját kritikus szekciójában van, addig más folyamat nem kezdheti el futtatni a saját kritikus szekcióját, azaz két folyamat nem lehet egyszerre kritikus szakaszban. A kritikus szakasz probléma lényege egy olyan eljáráscsomag összeállítása, amit a folyamatok egymás közötti együttműködésükben felhasználhatnak. Ehhez minden folyamatnak engedélyt kell kérnie, hogy beléphessen a saját kritikus szekciójába, ez a kódban a belépési szakasz. A kritikus szakasz végén lehet egy kilépési szakasz is. A belépési és a kilépési szekció között helyezkedik el a kritikus szakasz.

```
do {
  Belépési szakasz
    Kritikus szakasz
  Kilépési szakasz
      Maradék szakasz
} while (true);
```

Megoldási lehetőségek

A kritikus szekció problémájának megoldására többféle megoldás is lehetséges:

• Kölcsönös kizárás (mutual exclusion): ha egy folyamat a saját kritikus szakaszában van, akkor egyetlen más folyamat sem léphet be a saját kritikus szakaszába

• Haladás? (progress): ha egyetlen folyamat sincs kritikus szakaszban, de van több folyamat is, ami be akar lépni a kritikus szekciójába, akkor csak azok a folyamatok vesznek részt a kritikus szekcióba belépésről döntésben, amelyek nem hajtanak végre semmit a maradék szekcióban (?)

• Határos várakozás: létezik egy határ, hogy folyamatok hányszor léphetnek kritikus szakaszba egy másik folyamat előtt, miután az bejelentette, hogy szintén kritikus szakaszba akar lépni

Az operációs rendszerekben számos helyen, kernel szinten megvalósulhat versenyhelyzet, így ezek fejlesztésénél fokozottan kell ügyelni arra, hogy ne fordulhasson elő kernel módban versenyhelyzet. Az operációs rendszerekben két féle megközelítés van a kritikus szekciók kezelésére: a preemptív és a nem preemptív kernelek megoldásai. A preemptív kernelek megengedik, hogy egy folyamat, amely kernel módban van, kikerüljön a CPU-ból. A nem preemptív kernelek ezt nem engedik meg.

Versenyhelyzet

Adott n folyamat, és adott egy közös erőforrás. Minden folyamat használja a közös erőforrást, módosítja azt. Ilyen esetekben a eredményt az egyes folyamatok elérési sorrendje határozza meg, ami normál esetben nem megjósolható. Az ilyen helyzetet versenyhelyzetnek nevezik, amikor több folyamat versenyez egyetlen erőforrás használati jogáért. Ennek kivédéséhez szükséges biztosítani, hogy egy időben csak egyetlen folyamat módosíthassa az adott erőforrást, ehhez pedig szinkronizálni kell a folyamatokat. A versenyhelyzet egyik jellemző példája a termelő-fogyasztó probléma.

Megoldási lehetőségek

A versenyhelyzet megoldása lehetséges a folyamatok szinkronizációjával, és koordinációval.

**Peterson megoldása:** egy szoftveres megoldást ad a kritikus szekciók kezelésére. Bár a megoldás architektúrális gondok miatt nem biztos, hogy tökéletesen működik (atomi rendszerfüggvény hívások), de mivel jó algoritmikus leírást ad a probléma megoldására, emiatt érdemes vele foglalkozni. Két folyamat esetén valósítható meg kizárólag, amik váltakozva hajtják végre a kritikus és az egyéb szakaszaikat. Minden folyamatra a saját sorszámával hivatkozunk, hogy hányadik folyamat. Szükség van még két adatelemre, egy egész változóra (turn) és egy kételemű boolean tömbre (flag). A turn jelzi, hogy melyik folyamaton van a sor a kritikus szakaszába lépésben, a flag tömb pedig jelzi, hogy egy folyamat kész áll-e belépni a kritikus szakaszába. A kritikus szakaszba lépéshez a folyamat a flag hozzá tartozó értékét true-ra állítja, a turn értékét pedig a másik folyamat értékére. Ha mindkét folyamat egy időben akar kritikus szakaszba lépni, a turn értéke szinte egyszerre lesz mindkét folyamat értékére állítva, de csak az egyik érték fog megmaradni.

A megoldás helyességéhez bizonyítani kell, hogy

1.A kölcsön kizárás védve van,

2.A haladás(?) követelménye ki van elégítve,

3.A határos várakozás követelménye figyelembe van véve.

```
int turn;
boolean flag[2];
do {
  flag[i] = true;
  turn = j;
  while (flag[j] && turn == j);
    Kritikus szakasz
  flag[i] = false;
    Maradék szakasz
} while(true);
```

**10. Jellemezze a hardware-el támogatott versenyhelyzet feloldását segítő megoldásokat. Mutassa be a mutex-et és a szemaforokat**

Hardveres szinkronizáció

A szinkronizáció szoftveres megoldásán túl van egy általános lehetőség, amivel megoldható a kritikus szakaszok védelme: ez a zár (lock). A versenyhelyzetek kivédhetőek, ha a kritikus szakaszokat zárakkal védjük. A hardveres (továbbiakban: HW) védelmek nagyban megkönnyítik a programozási munkákat, és növelik a rendszer teljesítményét. Néhány egyszerű HW-es utasítással könnyen kivédhetőek a versenyhelyzetek.

A kritikus szekció problémája egyszerűen megoldható uniprocesszoros környezetben, ha lehetőség van a megszakítások tiltására, míg egy folyamat a kritikus szakaszában van. Ezzel a megoldással biztosra lehet venni, hogy egy időben két folyamat nem lehet kritikus szakaszban. Ez a megoldás a nem preemptív kernelekből származik. Több processzoros környezetben nem alkalmazható. Ilyen környezetben a megszakítások tiltása időpazarló lehet, hogy minden folyamathoz el kell juttatni a tiltási üzenetet. Ez az üzenet késlelteti a kritikus szakaszba lépést, ez pedig az egész rendszer teljesítményét rontja.

A modern számítógépek speciális HW utasításokat tartalmaznak, melyek atomi módon képesek tesztelni és módosítani egy értéket, vagy kicserélni két értéket. Ezen speciális utasítások segítségével könnyen megoldható a kritikus szakaszok védelme. Az atomi utasítások lényege, hogy oszthatatlanok, tehát nem lehet őket több részletben végrehajtani, nem lehet megszakítani a futásukat.

A hardveres védelem problémája, hogy az alkalmazás fejlesztők számára nehéz az implementációjuk és használatuk.

**Szemafor**

A szemafor egy egyszerűen alkalmazható szinkronizációs eszköz. Egy szemafor egy egész változó, amit az inicializálásától eltekintve, két atomi függvény használ: a wait() és a signal(). A wait() függvény a tesztelést szolgálja, a signal() pedig inkrementál. Mindkét függvényt oszthatatlan módon kell végrehajtani, hogy ha egy folyamat módosítja a szemafor értékét, akkor egyetlen másik folyamat se legyen képes párhuzamosan módosítani azt.

```
wait(S) { while (S<=0);S--;}
signal(S) {S++;}
```

Az operációs rendszerek megkülönböztetnek számláló (counting) és bináris (binary) szemaforokat. A számláló szemaforoknál a szemafor értéke korlátlan tartományban mozoghat. A bináris szemafort mutexnek szokás hívni. Számláló szemaforral olyan erőforrás védelme oldható meg, amelynek véges számú példánya van. A szemafor értéke a példányok számával lesz egyenlő a létrehozásakor. Minden folyamat egy wait() utasítással ellenőrzi, hogy van-e szabad példány, és ha éppen van, akkro csökkenti a szemafor értékét. Ha befejezte az erőforrás használatát, a signal() utasítás hívásával növeli a szemafor számlálóját. Ha a szemafor értéke 0, akkor minden erőforrás példány használatban van.

A szemaforok implementációjának legnagyobb hátránya, hogy szükség van az ún. busy waiting-re. Amíg egy folyamat a kritikus szakaszában van, a többi folyamat várakozik és közben folyamatosan figyeli, hogy felszabadult-e már a védett erőforrás. Ez a folyamatos figyelés a probléma multiprogramozott környezetben, ahol egyetlen CPU van megosztva a folyamatok között. A busy waiting feleslegesen pazarolja a CPU időt. Az ilyen módon működő szemaforokat spinlock-nak szokás nevezni, mivel folyamatok folyamatosan "pörögnek", amíg várnak a sorukra.

A busy waiting problémája megoldható a wait() és a signal() utasítások módosításával. Mikor egy folyamat a wait() futtatása után érzékeli, hogy a szemafor értéke nem pozitív, várakozni kezd. De nem busy waiting-et csinál, hanem blokkolja magát. A blokkolás egy, a szemaforhoz tartozó várakozási sorba teszi a folyamatot, és a folyamat állapotát várakozóra állítja. Ezután a vezérlést a CPU kapja meg, ami egy új folyamatot választ, amit futtathat. A blokkolt folyamat futása akkor folytatódhat, ha egy másik folyamat meghívja a signal() utasítást. A folyamat feléled a wakeup() utasítás hatására, és futásra kész állapotba kerül, és átkerül a ready queue-ba.

A szemaforok legkritikusabb része, hogy atomos módon legyene végrehajtva. Garantálni kell, hogy két folyamat nem tudja egyszerre futtatni a wait() és a signal() utasításokat. Egy processzoros környezetben ez megoldható a megszakítások tiltásával, amíg valamelyik utasítás fut. Többprocesszoros környezetben minden CPU-n tiltani kell a megszakításokat, de ez egyfelől nehéz feladat lehet, és nagymértékben ronthatja a rendszer teljesítményét. Emiatt az SMP rendszerek alternatív lockolási technikákat használnak.

**Mutex**

A mutex, vagy bináris szemafor olyan speciális szemafor, aminek az értéke kizárólag 0 vagy 1 lehet. Ilyen szemaforokkal egyszerűen megoldható a kölcsönös kizárás védelme. Korlátlan számú folyamat esetén is alkalmazható mutex a kölcsön kizárás védelmére.

A szemafor és a mutex közti különbség az, hogy míg utóbbi csak kölcsönös kizárást tesz lehetővé, azaz egyszerre mindig pontosan csakis egyetlen feladat számára biztosít hozzáférést az osztott erőforráshoz, addig a szemafort olyan esetekben használják, ahol egynél több - de korlátos számú - feladat számára engedélyezet a párhuzamos hozzáférés.

eljárás: -1 folyamat hozzá szeretne férni a kritikus szekcióhoz □ mutex_lock meghívása; ha nem zárolt vagyis a kritikus szekció elérhető akkor a hívás sikeres - ha a mutex zárolt akkor a hívó blokkolódik, amíg a kritikus szekcióban lévő folyamat nem végez, utána hívja a mutex_unlock eljárást

**11.Deadlock (halálos ölelés) jellemzése, feltételei. Sorolja fel a megoldási lehetőségeket**

Deadlock

Várakozási sorokkal kombinált szemaforok implementációja során előfordulhat olyan eset, hogy több folyamat vár egy olyan eseményre a sorban, amit egy szintén várakozó folyamatnak kellene kiváltania. Ilyen esemény pl. a signal() utasítás futtatása. Amikor ilyen állapot fordul elő, azt deadlock-nak nevezik. Ilyen helyzetben a folyamat soha nem fejezi be a futását, adott esetben foglalva tart erőforrásokat, ami pedig más folyamatoknál válthat ki deadlock helyzeteket.

A deadlock-ok jól ábrázolhatóak ún. erőforrás foglalási gráfokkal. Ezekben a gráfokban ábrázolva vannak a folyamatok, az erőforrások, és a köztük lévő kapcsolatok. Egy kapcsolat lehet (irányított él) hozzárendeléses (az erőforrástól mutat a folyamat felé) vagy kéréses (a folyamat felől mutat az erőforrás felé). A hozzárendeléses kapcsolatban a folyamat használja az erőforrás, hozzá van rendelve a folyamazhoz. A kéréses éleknél egy folyamat szeretné használni az erőforrást, és vár annak használati jogára.

**Feltételek**: Deadlock egy rendszerben akkor alakulhat ki, ha a következő feltételek egymással párhuzamosan adottak:

•Kölcsönös kizárás: legalább egy erőforrásnak kell lennie, amely nem megosztható több folyamat között

•Tart és várakozik: egy folyamatnak magánál kell tartania legalább egy erőforrást, és közben várakoznia kell egy másik erőforrás használati jogára, amit éppen egy másik folyamat tart magánál

•Nincs preemptivitás: az erőforrásokat nem lehet felszabadítani, míg foglalva vannak, azt csak az erőforrást használó folyamat szabadíthatja fel, ha már nincs az erőforrásra szüksége

•Körkörös várakozás: léteznie kell várakozó folyamatok egy csoportjának, ahol az első folyamat várakozik egy, a második folyamat által foglalt erőforrásra, a második folyamat a harmadik folyamat által foglalt erőforrásra vár, és így tovább

Mind a négy feltételnek teljesülni kell, hogy egy rendszerben deadlock jöhessen létre. A körkörös várakozás magában foglalja a tart és várakozik feltételt, így a négy feltétel nem teljesen független egymástól.

Megoldási lehetőségek

A deadlock-ok elkerülésére többféle módszer kínálkozik:

•Egy protokoll használata, hogy megelőzzük vagy elkerüljük a deadlock kialakulását, így biztosítva, hogy a rendszerben soha nem alakulhat ki deadlock helyzet

•Lehet hagyni, hogy a rendszer deadlock-ba kerüljön, de tudja azt felismeri és kezelni

•Figyelmen kívül lehet hagyni a problémát, és lehet reménykedni, hogy soha nem lesz deadlock

A harmadik változatot használja a legtöbb operációs rendszer (UNIX, Windows), így az alkalmazásfejlesztőkre hárul a deadlock-ok elkerülésének a feladata.

A deadlock megelőzésének legjobb módja, ha biztosítjuk, hogy a deadlock négy feltétele közül legalább egy biztosan ne lehessen igaz. Ezek a megoldások az erőforrás foglalási kérések szabályozásával érik el ezt.

A deadlock elkerüléséhez szüksége van az operációs rendszernek arra, hogy előre tudja, egy folyamat melyik erőforrásokat fogja kérni, és mennyi időre. Ezekkel az információkkal már megoldható az a döntés, hogy egy folyamatot várakoztatni kell egy erőforrás foglalására, vagy sem. Emellett szükséges tudni, hogy az adott folyamat milyen erőforrásokat birtokol éppen, a többi folyamat miket tart magánál, és a jövőben milyen erőforrás foglalások várhatóak.

A deadlock felismerésére számos algoritmus alkalmazható, ami képes megvizsgálni egy várakozó folyamat állapotát, és ha deadlock szituációt érzékel, akkor képes feloldani a helyzetet.

Ha egy rendszer figyelmen kívül hagyja a problémát, akkor előfordulhat olyan helyzet, hogy a rendszer deadlock-ba kerül, és semmilyen lehetőség nincs kitalálni, hogy mi okozta a problémát. Ez a teljes rendszer összeomlását okozhatja. Mivel deadlock meglehetősen ritkán fordul elő, így az operációs rendszer fejlesztők ezt az utat választották, mert olcsóbb néhány évente egy deadlock, mint egy azt elkerülő rendszer kifejlesztése.

**12.A deadlock (halálos ölelés) elkerülésének és megszüntetésének lehetőségei, értékelése**

**Deadlock megelőzése:** A deadlo kialakulása megelőzhető a négy feltétele közül legalább egynek a kiiktatásával.

**Kölcsönös kizárás:** A kölcsönös kizáráshoz szükség van egy nem megosztható erőforrásra. Ennek kiküszöbölése nem lehetséges, mivel mindig adott lehet olyan erőforrás, amit képtelenség megosztani (pl.: nyomtató).

**Tart és várakozik**: A tart és várakozik helyzet elkerüléséhez alapvető feltétel, hogy biztosítjuk, ha egy folyamat egy erőforrás használatát kéri, akkor semmilyen más erőforrás párhuzamosan ne legyen a folyamat által lefoglalva. Egy lehetséges megoldás, hogy egy folyamat futása kezdetekor minden olyan erőforrást lefoglal, amire szüksége lesz majd. Egy másik megoldás, hogy egy folyamat csak akkor kérheti egy erőforrás használatát, ha nincs egyetlen erőforrás sem lefoglalva nála, így egy újabb erőforrás kérés előtt fel kell szabadítania a korábban használt erőforrásait. Egy kéréssel több erőforrást is lefoglalhat egyszerre. Mindkét megoldásnak számos hátránya van, elsősorban az erőforrás kihasználtság várhatóan igen alacsony szintje. A másik gond, hogy a kiéheztetés esete is megvalósulhat, ha egy folyamat soha nem fér hozzá egy erőforráshoz, mert egy másik folyamat mindig lefoglalja azt előle.

**Nincs preemptivitás:** Ez a feltétel feloldható olyan módon, hogy ha egy folyamat használ egy erőforrást, és közben egy másik folyamat (ami már lefoglalt más erőforrásokat) is szeretné ugyanazt az erőforrást használni, akkor a várakozásra kényszerített folyamat minden addig lefoglalt erőforrása felszabadításra kerül. Az összes így felszabadult erőforrás felkerül a folyamat által lefoglalni akart erőforrások listájára (azon erőforrások listája, amikre a folyamat vár), és a folyamat csak akkor fog tovább futni, amikor minden erőforrás a rendelkezésére áll. Egy másik megoldás, hogy ha egy folyamat egy erőforrást kér, de azt nem kapja meg, akkor megnézi, hogy az erőforrást foglaló folyamat várakozik-e valamilyen más erőforrásra, és ha igen, megpróbáljuk felszabadítani azt, és a kérő folyamat számára átadni. Ha az erőforrás nem elérhető és fel sem szabadítható, akkor a folyamat várakozó állapotba vált. Ilyen állapotban egyes foglalt erőforrásait más folyamatok felszabadíthatják, de csak ha egy másik, futó folyamatnak szüksége van rájuk. A folyamat akkor futhat tovább, ha minden szükséges erőforrás a rendelkezésére áll.

**Körkörös várakozás:** A körkörös várakozás teljes mértékben kizárható, ha készítünk egy teljes rendezést az erőforrások típusairól, és az egyes folyamat növekvő számsorrendben foglalják le az erőforrásokat.

**Deadlock elkerülése:** A deadlock-ok elkerülésének egy másik módszere, hogy plusz információkat kérünk arról, hogy az erőforrások hogyan lesznek lefoglalva. Ennek segítségével minden folyamatnál meghatározható egy kérésekből és felszabadításokból álló sorozat, és a rendszer ezek alapján képes eldönteni minden folyamatnál, hogy annak szabad-e várakozni az erőforrásra, vagy az deadlo-ot okozna. Minden erőforrás kérés egy döntést igényel az operációs rendszertől az éppen szabad és a lefoglalt erőforrások, és a jövőben várható erőforrás foglalások figyelembe vételével.

A különböző algoritmusok elsősorban a kért információ mennyiségében és típusában térnek el egymástól. A legegyszerűbb és leghasználhatóbb modellek csak az egy folyamat által maximálisan használni kívánt erőforrások számát igénylik minden erőforrás típusban az adott folyamattól. Ennyi információ segítségével már konstruálható algoritmus, ami megakadályozza a deadlock kialakulását.

**Biztonságos állapot:** Egy állapot biztonságos, ha rendszer képes erőforrásokat biztosítani minden folyamatnak, és emellett el tudja kerülni a deadlock kialakulását. Másképpen fogalmazva, a rendeszer biztonságos állapotú, ha létezik egy biztonságos sorozat. Folyamatok egy sorozata biztonságos az aktuális lefoglalási állapotban, ha minden folyamat számára egy erőforrás kérelem kielégíthető az aktuálisan szabad erőforrásokból, és a többi folyamat által lefogalt erőforrások közül. Ha ekkor nem minden erőforrás elérhető, amit egy folyamat kér, akkor várakozhat, amíg egy másik folyamat el nem engedi az igényelt erőforrást.

Biztonságos állapot az, amiben nincs deadlock. Nem minden nem-biztonságos állapot deadlock, de esélye van, hogy deadlock-hoz vezet. Amíg egy állapot biztonságos, a rendszer el tudja kerülni a deadlock-ot.

**Bankár algoritmus:** Ez az algoritmus jól használható, ha egy erőforrás típusnak több példánya is van egy rendszerben. Ezt az algoritmus jellemzően banki rendszerekben alkalmazzák.

Minden új folyamatnál meg kell határozni, hogy maximálisan hány erőforrást fog lefoglalni. Ez a szám mindig kisebb, mint a rendszer összes erőforrásának a száma. Amikor a folyamat erőforrásokat igényel, a rendszer megvizsgálja, hogy ezek legfoglalása biztonságos állapotban tartaná-e a rendszer, vagy sem. Ha igen, az erőforrások foglalását engedélyezi, ha nem, elutasítja, és a folyamatnak várakoznia kell.

Számos adatstruktúrát kell kezelni az algoritmus implementálása során. Ezek az adatstruktúrák adják meg az erőforrás foglalási rendszer állapotát. Ha n a folyamatok száma, és m az erőforrás típusok száma, akkor a következő adatstruktúrákra van szükség:

•Egy m méretű vektor, ami minden egyes erőforrás típushoz a szabad erőforrások számát adja meg.

•Egy nxm-es mátrix írja le minden egyes folyamat maximális erőforrásigényét.

•Egy nxm-es mátrix írja le az egyes folyamatokhoz tartozó, még nem kielégített erőforrás igényeket.

**Deadlock megszüntetése**

**Deadlock érzékelése:** Egy deadlock érzékelhető egy algoritmussal, amely figyeli a rendszer állapotát, és megpróbálja felfedezni, ha deadlock alakul ki.

Ha minden erőforrásból csak egyetlen példány van, akkor készíthető egy algoritmus az erőforrás foglalási gráf segítségével. Ez egy speciális ún. wait-for gráf, amiben az egymásra várakozó folyamatokat jelölik. Ha több példány is lehet egy erőforrásból, ez a gráf nem alkalmazható. Az ilyen esetben használható algoritmus számos, idővel változó adatstruktúrát használ, amik hasonlóak a bankár algoritmusban használtakhoz.

**Folyamat megszüntetése:** Ha egy algoritmus deadlock-ot észlel, akkor egy lehetőség a folyamat(ok) megszüntetése. Két választás lehet:

•Az összes, a deadlock-ban részt vevő folyamat megszüntetése: ez biztosan megszünteti a deadlock-ot, de nagy lehet az ára, mivel minden folyamat minden addig végzett munkája elveszik

•Egyszerre egy folyamat megszüntetése, amíg a deadlock meg nem szűnik: ennek a megoldásnak nagy a költsége, mivel minden egyes folyamat megszüntetése után le kell futnia a deadlock érzékelő algoritmusnak

Egy folyamat megszüntetése nem egyszerű dolog, mivel számos olyan helyzet lehet, amikor egy folyamat kilövésével egy erőforrás inkonzisztens állapotban maradhat. A részleges megszüntetés esetében valahogy el is kell dönteni, hogy melyik folyamat legyen megszüntetve. Sok feltétel befolyásolhatja a döntést:

•A folyamat prioritása •Mennyi számítási időt használt el addig, és mennyi van (lehet) neki még hátra a befejezésig •Mennyi és milyen típusú erőforrást használt •Mennyi erőforrásra van még szüksége a befejezéshez •Mennyi folyamatot kell megszüntetni

**Erőforrás felszabadítása:** Egy deadlock megszüntethető egy folyamat által használt erőforrás felszabadításával, és ennek az erőforrásnak más folyamat számára átengedésével megszüntethető a deadlock. Ha ezen a módon megszüntethető a deadlock, három szempontot kell figyelembe venni: •Az áldozat kiválasztása: valamilyen szempontok alapján ki kell választani, hogy melyik folyamat megszüntetésének a legkisebb a költsége. Ezt a döntést számos feltétel befolyásolhatja, hasonlóan az előbb felsoroltakhoz. •Visszagörgetés: el kell dönteni, hogy ha egy folyamattól megvonunk egy erőforrást, akkor mi történjen azzal a folyamattal. Általában visszagörgetik a folyamatot egy olyan korábbi állapotba, ami a megvont erőforrás lefoglalása előtt van, és később innen indítják újra a folyamat futását.

•Kiéheztetés: biztosítani kell, hogy egy folyamatot ne lehessen kiéheztetni. Továbbá garantálni kell, hogy ne mindig ugyanaz a folyamat legyen az áldozat, akitől megvonják az erőforrásokat.

**13.Memóriakezelés alapelvei.** Memóriacímek kezelése különböző időben. Memóriavédelem hardware regiszterekkel

Memóriakezelés

Memóriacímek kezelése különböző időben

Egy program futtatásához be kell azt tölteni a memóriába valamilyen háttértárról. A folyamat futása során nem marad kötelezően mindig a memóriában, hanem mozoghat a memória és a háttértár között bizonyos mértékben. Azokat a folyamatokat, amik a memóriába betöltődésre várnak, a bemeneti (input) sorban vannak. Normál esetben betöltünk egy folyamatot a memóriába, ami a futása során aztán az utasításokat és adatokat a memóriából éri el. A folyamat megszűnése után a foglalt memóriaterület felszabadul.

A legtöbb rendszer megengedi, hogy egy felhasználói program a fizikai memória bármely részében lehessen. A program memóriaterületének címe attól függ, hogy melyik fizikai memóriacímen kezdődik a programhoz rendelt memóriaterület. Egy program futása során a memóriacímek számos átalakuláson mennek át: a forrásprogramban általában szimbolikus címek vannak megadva. Ezeket a fordító jellemzően átköti (meghagyva a régi címet is) áthelyezhető (relocatable) címekké. A loader ezeket a címeket pedig tovább alakítja abszolút címekké. Minden ilyen átalakítás egy mapping egyik címterületről egy másikra. Általában az utasítások és adatok memóriacímekké alakítása a következő időpillanatok bármelyikében megvalósítható a futtatás során:

•Fordítási idő: ha ekkor tudható, hogy a folyamat hol fog majd elhelyezkedni a memóriában, akkor abszolút kód generálható. Viszont ha ez később megváltozik, akkor újra kell fordítani a kódot.

•Betöltési idő: ha fordítási időben még nem tudni a memóriaterületét a folyamatnak, akkor a fordító relocatable kódot generál. Ekkor a végleges címzés kialakítása a betöltési időben történik meg. Ha bármi változás történik, csak újra be kell tölteni a folyamatot a memóriába.

•Futtatási idő: ha egy folyamat futási időben is várhatóan mozogni fog több memóriaterület között, akkor futási időre kell hagyni a címzés kialakítását. Ehhez speciális hardverre van szükség. A legtöbb operációs rendszer ezt a megoldást használja.

Memóriavédelem hardver regiszterekkel

A CPU a főmemóriát és a beépített regisztereit tudja közvetlenül kezelni, más eszközt nem. A gépi utasítások közül sok egy memória címet fogad paraméterként, és másféle címet nem tud fogadni, így minden adatátadás szükségszerűen a memórián keresztül valósul meg. Ahhoz, hogy a CPU dolgozni tudjon adatokon, azokat be kell mozgatni a memóriába, és csak úgy tudja elérni a CPU. A processzorba épített regiszterek jellemzője, hogy egyetlen órajel ciklus alatt elérhetőek. A legtöbb CPU képes utasításokat fogadni és egyszerű eljárásokat végrehajtani egyetlen ciklus alatt. Ez a memória buszon keresztül tranzakciókkal kezelt memóriával nem tehető meg. Egy memóriaművelet több CPU órajel ciklust is igénybe vehet, ami a processzort tétlenségre kárhoztatja egy kis időre, mivel nem kap adatot, amin tudna dolgozni. Ennek a kiküszöbölésére használnak cache-t a főmemória és a CPU között.

A rendszer helyes működéséhez biztosítani kell az operációs rendszer védelmét a felhasználói folyamatoktól, és a felhasználók folyamatait egymástól is meg kell védeni. Erre jellemzően hardveres megoldásokat alkalmaznak, ilyen a regiszteres védelem is. Minden folyamatnak van egy saját memória címterülete, ami valós része a memóriának, és biztosítani kell, hogy ezt elérhesse, és csakis az adott folyamat érhesse el, más nem. Ez a védelem megoldható két regiszterrel, egy bázis és egy határ (limit) regiszterrel. A bázis regiszter tartalmazza a legkisebb elérhető fizikai memóriacímet, míg a határ regiszter a memóriaterület méretét tartalmazza. A védelem ott valósul meg, hogy minden memóriaműveletnél a CPU összehasonlítja a felhasználói módban készült memóriacímet a regiszterek által meghatározott terület címével. Ha egy program az operációs rendszer vagy egy másik folyamat területét akarná használni, az operációs rendszer tiltja az elérést, és a program hibával fog leállni.

A kétféle regisztert csak az operációs rendszer állíthatja be, ami speciális jogokat igényel. Mivel különleges jogosultságú utasítások csak kernel módban indíthatóak, és az operációs rendszer pedig kernel módban fut, világos, hogy van joga, és csak neki van joga beállítani a regiszterek értékeit.

**14.Fizikai és logikai memória összevetése, swap. MMU és a paging bemutatása**

Fizikai és logikai memória

Azokat a memóriacímeket, amelyeket a CPU generál, logikai címeknek nevezzük, míg amiket a memóriakezelő egység lát (amik a memória memóriacím regiszterébe kerülnek), azokat fizikai címeknek hívjuk.

A fordítási és a betöltési időben végzett címátalakítás egymással egyenlő logikai és fizikai címeket ad. A futási időben végzett címátalakítás viszont különböző címeket eredményez. Ilyen esetekben a logikai címekre mint virtuális címekre hivatkozunk. Logikai címek egy csoportját logikai címterületnek hívják, az ezekhez tartozó fizikai címek pedig fizikai címtartományt adnak meg.

MMU

A futásidőben történő mappinget virtuális címekről fizikai címekre egy speciális hardver eszköz, az MMU (memory-management unit) végzi. Számos megoldással megvalósítható ez a leképezés. A legegyszerűbb ilyen megoldás a bázisregiszterek megoldásának egy általános változata, csak itt a bázisregisztert relocation (átirányítási) regiszternek nevezik. Minden olyan memóriacímzésnél, ahol a felhasználói program memóriacímet akar elérni, a kért címhez hozzáadódik a relocation regiszter értéke, így dinamikusan lesz leképezve a memóriacím a programhoz rendelt címekre.

Egy felhasználói program soha nem látja a fizikai címeket. A program képes egy memóriacímre mutató pointert készíteni, azt a memóriában eltárolni, megváltozatni, és azt összehasonlítani más címekkel. Amikor ezeket a címeket használja, az a bázisregiszter értékének segítségével képeződik le fizikai címekre, tehát a felhasználói programok kizárólag logikai címeket használnak. A memória mappelő hardver végzi el a logikai címek fizikai címekké transzformálását. A helyes memóriakezeléshez alapvető, hogy helyesen legyenek a logikai memóriacímek fizikai címekhez rendelve.

Swap

Egy folyamatnak futásakor a memóriában kell lennie, de időlegesen kikerülhet a memóriából egy háttétárra ha szükséges, és majd onnan visszaolvasva tovább futhat. Ideális esetben a memóriakezelő elég gyorsan ki tud swappelni folyamatokat, hogy közben legyen folyamat a memóriában és legyen futásra várakozó folyamat, amikor a CPU ütemezője új feladatot akar a CPU-nak adni.

Egy jellegzetes swapping megoldás a prioritás alapú: ha egy magasabb prioritású folyamat akar futni és helyet igényel a memóriában, akkor a memóriakezelő alacsonyabb prioritású folyamatokat ki tud swappelni, és a helyükre be tudja tölteni az új folyamatot. Amikor ez a magasabb prioritású folyamat befejeződik, akkor vissza lehet tölteni a swappelt folyamatokat, és folyatathatják a működésüket. Ezt a megoldást a roll in, roll out néven emlegetik.

Általában egy swappelt alkalmazás ugyanarra a memóriaterületre kerül vissza, ahonnan ki lett swappelve. Erre a memóriacímek mappelése miatt van szükség. Ha fordítási vagy betöltési időben történik a memóriacímek meghatározása, akkor nagyon nehéz másik címen tovább használni a programot. A futásidőben történő cím meghatározásnál lehetőség van más címtartományba visszatölteni a folymatot, mivel úgyis futás közben határozódik meg a memóriacím.

A swappeléshez szükséges egy háttértár, ez általában egy gyors diszk. A rendszerben van egy ready queue, amiben azok a folyamatok szerepelnek, amiknek a memóriaképük a háttértáron vagy a memóriában van, és futásra készen állnak. Amikor a CPU ütemező új folyamatot tölt be, hívja a dispatcher-t, ami ellenőrzi, hogy az új folyamat elfér-e a memóriában, és ha nem, akkor valamit kiswappel onnan, és utána betölti az új folyamatot.

A swappelés idejének nagy része az adatátvitel ideje. Fontos pontosan tudni, hogy egy új folyamat mennyi memóriát fog felhasználni, hogy pontosan meg lehessen határozni, hogy elfér-e a rendelkezésre álló memóriában, és ne csak becsülni lehessen ezt. A pontos számításokkal megspórolható a felesleges swappelés, ami nagyban javítja a hatékonyságot. Ehhez a felhasználói programnak folyamatosan jelentenie kell a rendszer felé, hogy mennyi memóriát használ, és hogyan változik a memóriafelhasználása.

A swappeléshez szükséges, hogy a swappelendő folyamat ne csináljon semmit. Problémát jelenthet, ha egy folyamat valamilyen I/O műveletre várakozik, és mivel az I/O aszinkron módon éri el folyamat memóriáját, így a folyamatot nem lehet swappelni.

A swappelés ezt a fajta alap változatát sehol sem használják, mivel túlságosan sok időt visz el a rendszertől. Jellemzőbb, hogy csak akkor kapcsolódik be a swappelés, hogy sok folyamat fut, és azok jelentős mennyiségű memóriát használnak fel. Ha a rendszer terhelése esik, a swappelés megint kikapcsolásra kerül. Ez a megoldás működik a legtöbb UNIX rendszerben.

Paging

A paging, vagy lapozás egy olyan memóriakezelési technika, ami megengedi, hogy egy folyamat fizikai címtartománya ne legyen folyamatos. Ez a megoldás áthidalja a változó méretű memóriaszeletek háttértárra helyezésének problémáját, amikor is egy kódrészletet vagy adatot swappelni kell, és helyet kell neki találni a háttértárolón. A háttértár is töredezik, akárcsak a főmemória, de mivel annál sokkal lassabb, így a tömörítés nem lehetséges.

Az alapvető eljárás a lapozás megvalósítására, hogy fizikai memóriát feldaraboljuk fix méretű blokkokra, ezek lesznek a frame-ek, a logikai memóriát pedig ugyanekkora egységekre osztjuk, ezek lesznek a lapok. Amikor egy folyamat futtatásra kerül, a lapjai beolvasásra kerülnek az elérhető memória framejeibe. A swap háttértára szintén egységnyi méretű blokkokra van osztva. Minden, a CPU által generált cím két részre oszlik: egy lapszámra, és egy lap eltolásra. A lapszám az adott lap azonosítója a laptáblában, ami tartalmazza minden lap báziscímét a fizikai memóriában. Ez a bázis cím az eltolással kombinálva adja meg egy lap pontos fizikai címét. Egy lap méretét az aktuális hardver határozza meg, ez architektúránként változik.

Paging használata esetén nincsen külső töredezettség, mivel bármely frame lefoglalható egy folyamat számára, viszont belső töredezettség előfordulhat. Ha egy folyamat memóriaigénye nem pontosan egyezik meg egy lap (vagy több lap együttes) méretével, az utolsó lap nem lesz teljesen tele. Átlagosan egy fél lap méretének megfelelő töredezettséggel szokás számolni, ami azt sugallja, hogy kis lapméretekkel érdemes foglalkozni. Viszont minden laptábla bejegyzésnek van bizonyos időszükséglete, és ezt nagyban csökkenti a nagyobb lapméret használata. Emellett a diszkek I/O műveletei is hatékonyabban nagyobb adatmennyiségek esetében. Ma már jellemző, hogy nem egy, hanem két féle lapméretet használ egy rendszer, és az aktuális folyamat függvényében használja egyik, vagy másik típust.

Egy új futtatandó folyamat érkezésekor a rendszer meghatározza a lapokban mérhető memóriaszükségletét, ami ugye egyenlő a szükséges frame-ekkel. Ha van elég hely a folyamatnak, akkor egyesével betölti a folyamat memórialapjait frame-ekbe, és minden frame azonosítóját bejegyzi a folyamat laptáblájába.

A paging egy fontos szempontja, hogy teljesen elválasztja a felhasználó által látható memóriát az aktuális fizikai memóriától. Egy alkalmazás a memóriát egy nagy egészként látja, és csak az adott program van szerinte a memóriában. Közben viszont az alkalmazás szét van osztva a teljes fizikai memóriában, és az aktuális fizikai memóriacímet hardveresen alakítja át logikai címekké. A cím mapping teljesen rejtve van a felhasználó elől, és azt az operációs rendszer vezérli. Mivel az operációs rendszer feladata a fizikai memória felügyelete, oda kell figyelnie annak lefoglalására: melyik frame-ek foglaltak, melyek szabadok, hány frame van összesen stb. Ezeket az adatokat a frame táblában szokás tárolni. Emellett a rendszernek felügyelnie kell a felhasználói területen futó programokat, azok logikai memóriaterületeit, és azok mappelését a fizikai területekre.

**15.File rendszerek alapelvei. Műveletek file-okon, directory-kon. Védelem a filerendszerben**

File

A file egy logikai tárolási egység. Az operációs rendszer a saját tárolórendszerének fizikai jellemzőit képezi le logikai szintre, így képződik egy file. A file egy névvel rendelkező gyűteménye egymással kapcsolatban lévő információknak. A felhasználó szemszögéből a file a legkisebb egysége egy háttértárnak, azaz egy adat nem írható ki egy háttértárra, amíg az nem egy file. Egy file lehet program vagy adat, és semmilyen formai megkötéssel nem rendelkezik. A file tartalmát annak létrehozója határozza meg. Egy meghatározott típusú file-nak meghatározott felépítése van, a tartalmától függően.

File attribútumok

Egy file-nak mindig van neve, amivel hivatkozni lehet rá. A név függetlenséget ad egy filenak minden folyamattól, felhasználótól, és akár a rendszertől is. A filenév az egyik legfontosabb file attribútum, és ez az egyetlen olyan információ, ami ember számára olvasható formában tárolódik egy file-ból. Emellett számos attribútum van még:

•Azonosító: egy egyedi cimke, általában szám, ami azonosítja a file-t a filerendszeren belül

•Típus: a file tartalmát jellemzi, nem minden rendszer támogatja

•Hely: egy pointer egy eszközön található helyre mutat, ahol a file tárolódik

•Méret: a file aktuális mérete (mennyi helyet foglal el), és esetenként a maximális mérete

•Védelem: védelem a file eléréséhez, hogy ki milyen műveletet végezhet a file-on

•Időpont, dátum és felhasználó: a létrehozás, utolsó módosítás és utolsó használat ideje stb.

Minden file információ egy könyvtár struktúrában tárolódik, ami szintén a háttértáron tárolódik. Jellemzően egy könyvtár egy file nevét és egyedi azonosítóját tartalmazza. Az azonosító megadja az útvonalat a file további attribútumaihoz.

File műveletek

A file egy absztrakt adattípus, hogy pontosan definiálni tudjuk, meg kell határozni, hogy milyen műveleteket lehet végrehajtani rajta. A legalapvetőbb, az operációs rendszer által biztosított file műveletek:

•File létrehozása: ez két lépésből áll. Első lépésben szabad helyet kell találni a file-nak a filerendszeren. A második lépésben pedig létre kell hozni a file bejegyzését a könyvtárban.

•File írása: file írásához szükséges megadni a rendszerhívásnak a file nevét, egyedi azonosítóját és az írni kívánt információt. A filenév alapján a rendszer megkeresi a könyvtárban a file helyét. A rendszer a write pointert arra a pontra állítja, ahol az írás történni fog, majd az írás befejezése után frissíti a pointert.

•File olvasása: olvasáshoz használt rendszerhívásnak meg kell adni a file nevét, és annak a memóriablokknak a címét, ahová a file tenni kell. Itt is a könyvtárban kell megkeresni a file helyét, és a rendszernek itt a read pointert kell az olvasás helyére állítania, majd az olvasás befejeztéve frissítenie. Mivel általában vagy írunk, vagy olvasunk egy file, elég egy fajta pointert használni, ez a current-file-position pointer.

•Pozícióállítás a file-on belül: a könyvtárban kikeresi a rendszer a file-t, és a pointert a megadott pozícióra állítja át. Ezt a műveletet szokás seek-nek, keresésnek is hívni.

•File törlése: törléshez szintén kell a könyvtárból a file helye, ha ez megvan, akkor felszabadítjuk a lefoglalt helyet, és töröljük a könyvtári bejegyzést.

•File kiürítése: ennek az eljárásnak a segítségével törölhető egy file teljes tartalma a file törlése és újra létrehozása nélkül.

Ez a hat eljárás csak egy minimális része a szükséges file műveleteknek. Jellemző művelet még a hozzáfűzés és az átnevezés is. Ezeket az egyszerű eljárásokat kombinálva még sok más file kezelő eljárás hozható létre.

A legtöbb eljárásnál látható volt, hogy minden műveletnél a könyvtárban meg kellett keresni az adott file bejegyzését. Elkerülendő az állandó keresést, sok rendszerben található egy open() rendszerhívás a file használata előtt. A rendszereknek van egy nyilvántartása a már megnyitott file-okról, azok minden adatával. Ha a file már szerepel ebben a táblában, akkor onnan sokkal gyorsabban megkapja a program a file adatait, és nem kell végigkeresni a könyvtárt. Ha egy file nincs már használatban, akkor egy close() hívás segítségével törlődik a bejegyzése a nyilvántartásból. Olyan rendszereken, ahol egyszerre több folyamat is használhatja ugyanazt a file, egy komplexebb megoldásra van szükség. Ezek a rendszerek általában kétszintű belső táblastruktúrát használnak: egy táblát folyamat szinten, és egyet rendszer szinten. Az első nyilvántartja, hogy egy adott folyamat milyen file-okat nyitott meg, és általában csak az adott folyamat használja ezt a táblát. A folyamat saját táblájának egy bejegyzése egy mutató a rendszer táblájának egy bejegyzésére. Ez a tábla folyamatoktól függetlenül tartalmaz minden file bejegyzést. Ha egy újabb folyamat akar használni egy már megnyitott file, egyszerűen csak bejegyzi ezt a saját táblájába. Számos rendszerben használnak számlálókat, hogy hány folyamat használja éppen az adott file-t, és ha az értéke 0, akkor törlődik a nyilvántartásból.

Könyvtár műveletek

Egy könyvtár felfogható egyfajta táblázatként, ami lefordítja a fileneveket file bejegyzésekké. A könyvtáraknál is többféle műveletre van szükség:

•File keresése: szükséges, hogy a könyvtár struktúrában meg tudjuk keresni egy adott file bejegyzését. Mivel a filenév csak egy szimbolikus kapcsolat, ezért meg kell tudni találni minden olyan file-t, ami az adott névvel rendelkezik.

•Új file létrehozása: létre kell tudni hozni új file-t, és hozzá kell adni a könyvtárhoz.

•File törlése: törölni kell tudni egy file-t a könyvtárból.

•Könyvtár listázása: tudni kell kilistázni a könyvtárban található file-okat.

•File átnevezése: tudni kell megváltoztatni a file nevét, ami együtt jár a könyvtárban található bejegyzésének módosításával is.

Védelem a filerendszerben

Számos rendszer biztosít védelmi eljárásokat a file-oknak. Ez jellemzően zárolást jelent, azaz egy folyamat zárolhatja egy file elérését más folyamatok számára. Ez hasznos lehet, ha egy file-t több folyamat is használna és módosítana, egy időben. A zárolás lehet megosztott, amikor egy zárolt file-t több folyamat is tudja használni (és zárolni) egy időben (pl.: többen olvassák ugyanazt a file-t). Az egyedi zárolás esetén viszont csak egyetlen folyamat fogja tudni használni a file-t (pl.: file írása). Bizonyos operációs rendszerek ezen a két típuson túl még biztosítanak kötelező (mandatory) és ajánlott (advisory) zárolásokat. Egy zárolás kötelező, ha egy folyamat kizárólagosan zárolt egy file-t, akkor másik folyamat nem fogja tudni használni a zárolt file-t. Ha viszont egy zárolás ajánlott, akkor egy folyamatnak nem kötelező azt figyelembe vennie, az operációs rendszer nem fogja tiltani a file elérését. Ilyenkor a felhasználói programon múlik, hogy hogyan viselkedik: az ajánlott viselkedés, hogy ilyenkor zárolást kér a program, és annak sikerétől függően folytatja a futást.

**16.Könyvtárszerkezetek történeti bemutatása. Egy modern operációs rendszer könyvtárszerkezetének ismertetése**

Egyszintű könyvtárszerkezet

A legegyszerűbb szerkezet, minden file ugyanabban a könyvtárban van. Számos hátránya van, főleg ha sok file van a rendszerben, vagy több felhasználó. Mivel minden file egy helyen van, mindegyiknek egyedi nevet kell adni, ami a név hosszának megkötései miatt (pl.: DOS, 8+3 karakter) problémás lehet. Több felhasználó esetén egy felhasználó számára problémás lehet megjegyezni a saját file-jait.

Kétszintű könyvtárszerkezet

A kétszintű szerkezet fejlődése az egyszintűhöz képest az volt, hogy minden felhasználó saját könyvtárat kapott, ezt hívják user file directory (UFD)-nak. Az UFD-k struktúrája hasonló a könyvtárakéhoz, de mindegyik csak a felhasználója saját file-jait tartja nyilván. Amikor a felhasználó elindít egy folyamatot vagy belép a rendszerbe, a rendszer master file directory (MFD)-jában keres. Az MFD felhasználói név alapján van indexelve, és minden bejegyzése egy felhasználó UFD-jére mutat.

Egy felhasználó file-jai mindig a saját UFD-jében kerülnek bejegyzésre, így lehetőség van különböző felhasználóknak ugyanazon a néven file-okat létrehozni. Minden file művelet a felhasználó saját UFD-jében zajlik le. Bár a kétszintű könyvtárszerkezet megoldja a filenevek egyediségének problémáját, vannak még problémái: a felhasználók izolálva vannak egymástól, ami közös munka esetén nagy probléma. Bizonyos biztonsági beállítások módosításával ez a probléma áthidalható.

A kétszintű felépítés felfogható egy 2 magas fastruktúrának is, ahol az MFD a gyökér, a közvetlen leszármazottai az UFD-k, a levelek pedig a file-ok maguk. Egy felhasználói név és egy filenév megadása egy utat jelöl ki a fában (ez az útvonalnév vagy path name). Bizonyos rendszereknél az útvonalhoz hozzáadódik egy kötet név, pl.: c:\. Speciális eset a rendszerfile-ok használata: ha egy felhasználó egy rendszer programot akar meghívni, alap esetben a file-t a rendszer a saját UFD-jében fogja keresni (ahol várhatóan nem lesz). Megoldás lehet minden felhasználóhoz odamásolni az összes rendszer file-t, de ez pazarló. Ehelyett az UFD keresési eljárását kell kicsit módosítani: a rendszerfile-ok egy speciális felhasználói könyvtárba kerülnek, és file keresésekor először a felhasználó saját UFD-jét nézi végig a rendszer, és ha ott nincs a file, akkor következik a rendszer file-ok UFD-je. A könyvtárak sorozatát, ahol a rendszer file-okat keres, keresési útvonalnak hívják.

Fa struktúrájú könyvtárszerkezet

Ez a szerkezet egy általánosítása a kétszintű struktúrának (ami egy 2 szintű faként fogható fel), ahol nincsen megkötés a fa magasságára. Ez lehetőséget ad alkönyvtárak használatára, és ezzel saját file rendezési struktúrákra. A fának van egy főkönyvtára, és minden file innen elérhető egy egyedi útvonalon. Minden könyvtár tartalmazhat file-okat vagy további könyvtárakat. Általában a könyvtár egy speciális file-ként van kezelve, egy egységes belső formátummal. Egy speciális bit mondja meg az adott file-ról, hogy az file, vagy könyvtár.

Minden folyamathoz tartozik egy aktuális könyvtár, aminek tartalmaznia kell a folyamat futásához szükséges file-okat. A file-ok keresése mindig ebben a könyvtárban történik, de a felhasználó adhat meg másik könyvtárat, amiben keresni akar. Egy speciális rendszerhívás van könyvtárváltásra.

Az útvonal neveknek két fajtája van: abszolút és relatív. Az abszolút hivatkozás a főkönyvtárban kezdődik, és a megadott file-ig tart. A relatív az aktuális könyvtárhoz képest ad meg egy hivatkozást.

Egy fastruktúra implementáció egy érdekes döntése, hogy hogyan kezeli egy könyvtár törlését. Ha a könyvtár üres, akkor egyszerűen törlődik. Ha nem üres, akkor két út lehetsége: van az MS-DOS féle megközelítés, hogy addig nem törli le, amíg nem üres. A másik megoldás a UNIX féle, hogy ilyenkor a könyvtár teljes tartalma törlődik.

Egy fastruktúrában egy file könnyen megosztható több felhasználó között, mivel minden felhasználó képes elérni ugyanazt a file-t ugyanazon az útvonalon.

Egy modern operációs rendszer könyvtárszerkezete

Elvárások egy modern rendszer könyvtárszerkezetével szemben:

•Legyen gyors a file-ok elérése

•Különböző felhasználók ugyan olyan néven hozhassanak létre file-okat

•Egy file elérhető legyen különböző nevekkel is

•Egy filenév ne adjon információt arról, hogy milyen háttértáron van a file

•Lehetőség legyen csoportosítani a file-okat (könyvtárak)

**17.Tárolórendszerek elvei. A RAID elveinek és megoldásainak bemutatása**

Tárolórendszerek

Mágneses lemezek

A háttértárolók nagy részét az ilyen elvű tárolók adják. Felépítésük aránylag egyszerű, minden diszk lemez egy lapos korong, különböző átmérőkkel. A lemezek mindkét oldala egy mágneses anyaggal van bevonva, és ezeken a mágneses lemezeken tároljuk az adatokat. Egy lemez mindkét oldala felett mozog egy olvasófej, ami egy diszk karra van csatlakoztatva, és ez mozgatja az összes olvasófejet. A felület logikailag körkörös sávokra van osztva, amik szektorokból állnak. Az egy diszk kar pozícióban elérhető sávok összessége alkot egy cilindert. Használat közben a lemezek forognak. A diszkek sebességét két értékkel szokás mérni:

•Adatátviteli sebesség: az a sebesség, amivel az adatok áramlanak a diszk és a gép között

•A pozícionálási idő több részből tevődik össze

oKeresési idő: a diszk kar beállításának ideje a kért cilinderre

oForgatási késleltetés: a kért szektornak a fej alá forgatásának ideje

Egy diszk lehet eltávolítható, ezek jellemzően csak egyetlen lemezt tartalmaznak egy műanyag tokban. Ilyen a floppy.

Egy diszk mindig valamilyen kábelen át csatlakozik a gép I/O rendszeréhez. Több féle technológia létezik lemezek kezelésére: IDE (integrated device electronics), EIDE (enhanced IDE), ATA (advanced technology attachement), serial ATA, USB, fibre channel, SCSI. Az adatátvitelt egy speciáli processzor végzi, a kontroller. A host kontroller a sín számítógép felőli végén van, míg a diszk kontroller a diszkbe van beépítve.

Mágneses szalagok

Az egyik legrégebbi tárolóegységek. Hosszú távon képesek megőrizni sok adatot, de az elérési idejük nagyon lassú. Elsősorban backup célokra használják még. Egy random adatelérés akár perces nagyságrendű is lehet, azért inkább nagy, egybefüggő adattömegeknél alkalmazható jól.

NAS

A NAS, vagy network-attached storage egy hálózaton keresztül távolról elérhető speciális adattároló rendszer. Kliensek valamilyen elosztott filerendszeren keresztül érhetik el az adatokat. Ezek a megoldások egyfajta RPC technológiát megvalósító szoftverek. Egy NAS egység általában rendelkezik valamilyen RAID funkcionalitással is. A NAS-t fel lehet fogni egyfajta tároló-elérési protokollként is. Segítségükkel egyszerűen lehet hálózaton keresztül nagy kapacitású, közös használatú tárolórendszert üzemeltetni. A hátránya a NAS-nak közvetlenül csatolt tárolókhoz képest a rosszabb hatékonyság és a kisebb teljesítmény (pl.: sávszél korlátok).

SAN

A SAN, vagy storage-area network egy tároló protokollokat (storage protocol) használó privát hálózat, ami szervereket és tárolóegységeket köt össze. A SAN ereje a nagy flexibilitása: számos hoszt és számos storage tömb alkothat egyetlen SAN-t, és a tárolókapacitás dinamikusan adható oda egy-egy hosztnak. Több kiszolgáló között meg is oszthatóak tárolókapacitások, és az azon tárolt adatok. Jellemzően fibre channel-en keresztül kapcsolódnak össze a SAN egységei.

RAID

A RAID, vagy redundant arrays of independent disks egy általánosan használt technika több diszk összekötésére, és ezzel nagyobb teljesítmény és/vagy nagyobb megbízhatóság érhető el.

Megbízhatóság növelése redundanciával

Egy n darab diszkből álló tömb meghibásodási aránya jóval nagyobb, mint egyetlen diszké. A várható élettartama egy tömb elemének lineárisan csökken a darabszám emelkedésével, ami nagy adatveszélyt jelent. Erre a megoldás a rendundancia, amikor is némi extra információt tárolunk, ami csak akkor szükséges, ha hiba esetén újra kell építeni a tömböt.

A legegyszerűbb rendundáns megoldás a tükrözés, amikor két diszken ugyanazokat az információkat tároljuk párhuzamosan. Adatvesztés csak mindkét diszk kiesésével fordulhat elő.

Teljesítmény növelése párhuzamosítással

Adatátviteli teljesítmény úgy növelhető több diszk esetén, hogy egy logikai kötetet több diszket összefogva hozunk létre. A legegyszerűbb esetben minden kiírt byte bitjeit szétterítjük a diszkeken, ez a bit szintű összefogás. De nem csak bit szinten, hanem blokk szinten is lehet megoldást találni, ahol is egy file blokkjai terítődnek szét a különböző diszkeken. De lehet akár szektor, vagy bármilyen más egység is egy ilyen megoldás alapja. A cél minden megoldás esetében az, hogy növeljék a rendszer teljesítményét kis adatelérések esetén terheléselosztással, és csökkentsék a válaszidőket nagy adatkéréseknél.

RAID szintek

Számos szabványos RAID megoldás van, hogy milyen módon lehet összekötni diszkeket bizonyos célok eléréséhez. Egy-egy ilyen szabványos megoldást egy RAID szintnek szokás nevezni. A legismertebb RAID szintek:

•RAID0: diszkek összekötését jelzi blokk szinten, redundancia nélkül (stripe)

•RAID1: diszkek tükrözése (mirror)

•RAID0+1: a RAID0 és a RAID1 kombinációja, ahol a diszkek egy része össze van fogva, és utána ez az egység van tükrözve egy ugyanolyan összefogott egységre.

•RAID1+0: itt diszk párok vannak tükrözve, majd ezek a tükrözött párok vannak összefogva egy egységbe

•RAID4: blokk szintű összefogást alkalmaz, kiegészítve azt egy plusz diszk-el, amin a paritásbiteket tárolja a többi diszkhez.

•RAID5: hasonló a RAID4-hez, de itt az adat és a paritás információk is n+1 diszken vannak szétterítve. Minden blokk esetén egy diszk tartalmazza a paritás adatokat, és a többi az adatokat.

**18.Elosztott file rendszerek. Alapelvek, megoldások. Az AFS rövid ismertetése**

Elosztott filerendszerek

Egy elosztott filerendszernél (továbbiakban: EFR) három fontos fogalom van:

•Szolgáltatás: egy szoftver példány, ami egy vagy több gépen fut, és valamilyen funkciókat tesz a kliensnek elérhetővé

•Szerver: egy szolgáltatást futtató gép

•Kliens: egy folyamat, ami meg tudja hívni a szolgáltatást egy kliens felületen keresztül

Ezekkel a fogalmakkal azt mondhatjuk, hogy egy EFR klienseknek nyújt filerendszer szolgáltatásokat valamilyen hálózaton keresztül. Egy EFR esetében nincsen egy központi adattároló, hanem a rendszer független tárolók hálózatából áll össze. Ideális esetben egy EFR a kliensek számára egyetlen filerendszernek látszik, függetlenül attól, hogy hány szerver szolgálja majd ki az adatokat, azaz az EFR elosztottsága a kliens számára láthatatlan: ezt hívják transzparenciának a kliens oldalán. Egy EFR legfontosabb teljesítménybeli kérdése, hogy milyen gyorsan képes kéréseket kiszolgálni, mivel most nem a helyi diszk vagy CPU teljesítménye számít, hanem az adatokat sok helyről, távolról kell összeszedni. A távoli elérés ad némi plusz időt az adatok elérésénél, amiben benne van a szerver elérésének ideje, és annak válasza is a kliens felé. Emellett a kommunikációs csatorna késleltetése is számottevő lehet.

Egy EFR fontos része a naming, ami egy megfeleltetés logikai és fizikai objektumok között. A felhasználók logikai objektumokkal dolgoznak, de ezek fizikai objektumokként vannak letárolva valahol. Egy többszintű megfeleltetés biztosítja, hogy a felhasználók számára egy file absztrakt megfelelője elfedje a valós tárolást és formátumot. Egy transzparens EFR esetében még egy szint van, ami elrejti a hálózatot, amin keresztül a file meg van osztva.

Egy EFR esetében két fontos fogalom van a naming-el kapcsolatban:

•Tárolási hely átlátszósága: egy file neve nem mond el semmit annak fizikai tárolási módjáról

•Tárolási hely függetlensége: egy file neve nem feltétlenül változik meg annak fizikai tárolási helyének változásával

Három nagy naming séma van, ami egy EFR-ben alkalmazható. A legegyszerűbb megoldásban egy file-t valamilyen kombinációban a hosztjának neve és a helyi neve azonosít, ami biztosan egyedi lesz az EFR-en belül. Ez a megoldás sem nem tárolási hely transzparens, sem nem tárolási hely független. Itt az EFR teljesen izolált egységekből áll, mindegyik saját belső filerendszerrel rendelkezik.

A második sémát a Sun féle NFS használja, aminek segítségével távoli könyvtárak csatlakoztathatóak a helyi könyvtárakba, teljesen transzparens módon. A komponensek integráltan szolgáltatnak egy transzparens hálózatot, de ez az integráció korlátozott és nem egységes, mivel minden gép más és más távoli könyvtárakat csatlakoztathat fel. A kapott struktúra nagyon flexibilis.

Teljes komponens integráció érhető el, ha egyetlen globális névtér fogja össze az összes file-t a rendszerben. Ideális esetben az így kapott filerendszer nagyon hasonló egy hagyományos filerendszerhez.

AFS

Az AFS az Andrew file system rövidítése, az IBM fejlesztése. Az AFS megkülönböztet kliens és dedikált szerver gépeket (ezeket hívják összefoglalóan Vice-nak). A klienseken a file-ok névtere két részre osztott: van egy helyi és egy megosztott névtér. A szerverek a megosztott névteret szolgáltatják a klienseknek egy homogén, transzparens, mindenki számára egységesen látható filerendszerként. A szerverek együtt felelnek a megosztott névtér használatáért. A kliensek egy Virtue elnevezésű protokollon keresztül kommunikálnak a szerverekkel. Az AFS több szempontot szem előtt tartva működik:

•Kliens mobilitás: egy file-t a megosztott névtérben bármely kliens gépről el lehessen érni

•Biztonság: autentikáció és biztonságos adatátvitel megoldott, a kliensek és a szerverek titkosított kapcsolaton át kommunikálnak

•Védelem: lehetőség van elérési listák használatára könyvtáraknál, és a UNIX-oknál megszokott file védelemre

•Heterogenitás: keverhetőek legyenek a helyi és a távoli file-ok egy kliens gépen

A közös névtér ún. kötetekből (volume) épül fel, minden kötet egy kisebb komponens egységet, egy file-t vagy klienst jelöl. A kötetek a unix-os mount-szerű módon vannak összefogva. Egy Vice file vagy könyvtár rendelkezik egy egyedi azonosítóval, ez a fid, ami 3 részből áll: egy kötet azonosítóból, egy vnode azonosítóból és egy egyedi azonosítóból. A kötetek útvonal adatbázisa minden szerveren jelen van, replikált formában. Egy kliens meg tudja határozni minden kötet helyét ennek az adatbázisnak a lekérdezésével.

Fontos része az AFS-nek a file-ok cache-elése a rendszeren belül. A kliensek csak file megnyitásakor és bezárásakor kommunikálnak egy Vice szerverrel. Minden kliens cache-eli az általa megnyitott file-okat, és minden módosítást helyben hajt végre, és csak a file bezárásakor csatlakozik újra a szerverhez a kliens. Egy másik teljesítménynövelő megoldás, hogy amikor egy kliens cache-elni kezd egy file, azt a szerver bejegyzi az adott file vagy könyvtár adataiba, és ha egy másik kliens akarja elérni a cache-elt elemet, akkor értesíti erről az azt cache-elő klienst, és csak utána adja oda a használati lehetőséget az új kliensnek. Ezzel rengeteg cache validálási információcsere spórolható meg.

**19.I/O eszközök típusai, tulajdonságai; eszközkezelő. Programozott I/O (Polling)**

I/O eszközök

Egy számítógéphez számtalan típusú I/O eszköz csatlakoztatható. Mivel ezek rengeteg féle funkciót valósíthatnak meg sokféle sebességgel, ezért szükség van valamilyen vezérlésre. Ezt a kernel I/O alrendszere végzi, ami elválasztja a kernel többi részét az I/O eszközök kezelésétől.

Számtalan eszköz létezik egy számítógéphez, és ezek legtöbbje besorolható valamilyen nagyobb kategóriába (tároló egység, adatátviteli egység, HID stb). Emellett vannak speciális eszközök, speciális (akár egyedi) feladatokra. Eme nagyon nagy változatosság ellenére csak néhány főbb irányelvet kell megérteni, hogy mi módon csatlakoznak az eszközök, és azokat hogyan lehet kezelni.

Egy eszköz egy kábelen vagy a levegőn át küld jeleket a számítógépnek, és egy csatlakozási ponton (port) kommunikál a géppel. Ha egy eszköz egy általános vezetéket használ, akkor azt busznak hívjuk. A busz egy kábelköteg és egy merev protokoll, ami leírja az üzenetek küldésének módját a buszon. Buszokat számtalan helyen alkalmaznak a számítógépes architektúrákban. Az egyik legismertebb a PCI busz, ami összeköti a processzor-memória alrendszert a különböző gyors eszközökkel.

Egy vezérlő különböző elektronikák egy gyűjteménye, amik működtetni tudnak egy portot, egy buszt vagy egy eszközt. Egy soros port vezérlő egy egyszerű eszköz vezérlő, ami egyetlen chipből áll a számítógépen belül, és vezérli a soros port vezetékein átmenő jeleket. Ezzel ellentétben pl. egy SCSI busz vezérlője egyáltalán nem egyszerű, mivel maga a SCSI protokoll is komplex. Általában a vezérlőt egy külön vezérlőkártyaként szokás alkalmazni, amin helyet kap egy microprocesszor, a mikrokód, és némi saját memória is. A diszkek is rendelkeznek valamennyi saját vezérléssel.

A CPU egyszerűen szólva regiszterek írásával és olvasásával kommunikál az egyes I/O vezérlőkkel. Minden vezérlőnek van egy vagy több regisztere az adatai és jelei számára. Egy mód ennek megvalósítására, hogy a kommunikáció speciális I/O utasítások használatával végezhető el, amik megadják az átviendő adatot és a cél memóriacímét. Az I/O utasítás kiold egy busz csatornát és kiválasztja a megfelelő eszközt, és bemozgatja az eszköz regisztereibe a megfelelő értékeket. Egy másik lehetőség, hogy az eszközöket ún. memory-mapped módon kezeli a vezérlő. Ekkor az eszközvezérlő regiszterek a processzor címtartományába vannak mappelve, és a CPU egyszerű adatátviteli utasításokkal írja és olvassa az I/O eszközök regisztereit. Bizonyos rendszerek mindkét megoldást is alkalmazzák, az eszközök egy részét az egyik, egy részét pedig a másik módon kezelve. Egy eszköz bizonyos funkciónként máshogy is lehet kezelve.

Egy általános I/O portnak négy féle regisztere van:

•Data-in regiszter: a hoszt ezt olvassa, hogy adatot kapjon

•Data-out regiszter: a hoszt ezt írja, hogy adatot küldjön

•Státusz regiszter: a hoszt által kiolvasható biteket tartalmaz, ezek különböző állapotokat jeleznek, mint pl.: az aktuális parancs végzett-e, van-e kiolvasható adat a data-in regiszterben, vagy fordult-e elő valahol hiba

•Vezérlő regiszter: a hoszt által írható, ezen keresztül lehet parancsfuttatást kezdeményezni, vagy működési módot beállítani.

Polling

A hoszt és a vezérlő közötti kommunikációs protokoll nagyon összetett is lehet, de a handshaking fogalma meglehetősen egyszerű. A menete a következő lépésekből áll:

1.A hoszt folyamatosan olvassa ki a busy bitet, amíg az nem lesz üres

2.A hoszt beállítja a write bitet a command regiszterben, és beírja az adatokat a data-out regiszterbe

3.A hoszt beállítja a command-ready bitet

4.Amikor a vezérlő észleli a command-ready bitet, beállítja a busy bitet

5.A vezérlő beolvassa a command regiszter tartalmát, benne a beállított write bitet, ezért beolvassa a data-out tartalmát is, és elvégzi az eszközön az I/O műveletet

6.A vezérlő törli a command-ready bitet, törli az error bitet a státusz regiszterben hogy jelezze az eszköz I/O műveletének sikerét, és a busy bit törlésével jelzi, hogy végzett

Ezt a ciklust kell megismételni minden byte-nyi adat küldésénél.

Az első lépésben a hoszt ún. busy-waiting-et végez, vagy polling-ol: egy ciklusban van, olvassa a státusz regisztert újra és újra, amíg a busy bit fel nem szabadul. Ha a vezérlő és az eszköz is gyors, ez a művelet elfogadható. Gond akkor van, ha sokáig kell várakoznia, akkor közben a hoszt egy másik feladatra vált, viszont akkor nem fogja tudni, mikor szabadul fel a vezérlő. Számos eszköznél a várakozás nem is elfogadható, mivel az adat elveszik, ha várni kell vele.

Számos számítógép architektúra esetén 3 CPU órajel ciklus elég egy eszköz lekérdezéséhez: kiolvasni az eszköz regisztert, logikai és művelettel meghatározni a státusz bitet és elágazni, ha nem nulla az értéke. Világos, hogy az egyszerű polling eljárás hatékony, de a hatékonysága sokat romlik, ha ritkán ismételve próbálkozik egy eszköz lekérdezésével, és közben fontos CPU feladatok pedig nem végződnek el emiatt. Ezekben az esetekben hasznosabb valamilyen hardveres eszközre bízni a CPU értesítését, ha egy eszköz felszabadult.

**20.Irq és DMA vezérelt I/O bemutatása. Szinkron és aszinkron I/O**

IRQ vezérelt I/O

A CPU-ban van egy vezeték, amit megszakítás kérési vonalnak neveznek, amin a CPU érzékel minden egyes utasítás végrehajtása után. Ha a CPU érzékeli, hogy egy eszköz beállított egy jelet a vonalon, akkor a processzor elmenti az állapotát és átlép a megszakításkezelő eljárásba, ami a memória egy fix címén található. A megszakításkezelő eldönti, hogy mi volt a megszakítás oka, végrehajtja a szükséges számításokat, visszaállítja a CPU állapotát, és lefuttat egy return form interrupt utasítást, hogy a processzor visszatérjen a megszakítás előtti állapothoz. Azt mondjuk, hogy az eszköz előidézett egy megszakítást azzal, hogy jelet küldött a vonalra, a CPU elkapta a megszakítást és elküldte a megszakításkezelőnek, az pedig törölte a megszakítást azzal, hogy kiszolgálta az eszközt. Ezt az eljásárt hívják megszakításvezérelt I/O ciklusnak.

Az általános megszakítási eljárás lehetővé teszi a CPU-nak, hogy reagálhasson egy aszinkron eseményre, amikor is egy eszközvezérlő szabaddá válik. Egy modern operációs rendszerben összetettebb kezelésre van szükség:

•Szükséges, hogy kiiktatható legyen a megszakításkezelés kritikus adatok kezelésénél

•Szükséges egy hatékony mód a megszakítás továbbadására az eszköz megszakításkezelőnek anélkül, hogy a polling módszerével végigmennénk az össze eszközön hogy eldönthessük, kitől jött a megszakítás

•Többszintű megszakításokra van szükség, hogy az operációs rendszer választani tudjon különböző prioritású megszakítások között, és a megfelelő sürgősséggel tudjon válaszolni

Egy modern számítógépben ezeket a feladatokat a CPU és a hardveres megszakításkezelő látja el.

A legtöbb CPU-ban két megszakítás jelző vonal van: egy a nem maszkolható megszakításoknak, ami az olyan eseményeknek van fenntartva, mint pl. nem feloldható memóriahibák. Egy másik vonal pedig a maszkolható megszakításoké, ez a vonal kikapcsolható kritikus számítások előtt. Ezt a vonalat használják az eszközvezérlők.

Egy megszakítási eljárás egy címet vár, ami alapján kiválasztja a megszakítás kezelésének módját egy kisebb csoportból. Legtöbbször ez a cím egy offset érték, egy megszakítás vektornak nevezett táblában. Ez a vektor speciális megszakításkezelő memóriacímeit tartalmazza. A vektor használatának oka, hogy csökkentse az önálló megszakításkezelőknek a keresési idejét, amíg megtalálja az összes megszakítás forrását, hogy meghatározza, melyiknek van szüksége kiszolgálásra. Gyakorlatban ez azt jelenti, hogy több eszköz van, mint ahány cím eleme van a vektornak. Erre megoldás a megszakítás láncolás, ahol a vektor minden eleme egy megszakításkezelő lista elejére mutat. Amikor egy megszakítás érkezik, minden lista végig van szólítva, amíg meg nem lesz a megfelelő.

Egy megszakításkezelőben megtalálható a megszakítások priorizálása is. Ennek segítségével a CPU elhalaszthatja egy alacsony prioritású megszakítás kezelését, ha van előtte egy magasabb prioritású a sorban.

A megszakításkezelő rendszerek a megszakítások mellett kezelnek még kivételeket is: ilyen esemény pl. a 0-val osztás, tiltott memóriaterület elérési kísérlete stb.

DMA vezérelt I/O

Nagy adatmennyiségeket mozgató eszközöknél (pl.: diszkek) pazarló megoldás a CPU drága idejét arra felhasználni, hogy adatokkal lássa el a vezérlőt byte-onként. Az így működő rendszereket hívják programmed I/O-nak (PIO), innen ered a pio mode is. Számos rendszer kiiktatta a CPU állandó terhelését egy speciális célprocesszor alkalmazásával, ez a DMA vezérlő. Egy DMA átvitel indításához a hoszt egy DMA utasításblokkot beír a memóriába. Ez a blokk tartalmaz egy pointert az átvitel forrására, egy pointert a célra, és egy számlálót az átvinni szükséges byte-ok számáról. A CPU ennek a blokknak a címét beírja a DMA vezérlőbe, majd csinálja tovább a dolgát. A DMA vezérlő közvelenül kezeli a memóriabuszt, címet tesz a buszra a CPU nélkül. Ma már mindig számítógépben található ilyen vezérlő.

Ha a DMA vezérlő végzett a teljes adatátvitellel, akkor küld egy megszakítást a CPU-nak. Amikor a DMA vezérlő elfoglalja a memóriabuszt, a CPU egy pillanat idejére nem éri el a memóriát, de a cache-ében még lesz adat. Ez a órajel-lopás lassíthatja a CPU-t, de a DMA vezérlő által átvett munka sokkal nagyobb mértékben gyorsítja a teljes rendszert. Egyes rendszerek fizikai memóriát használnak a DMA vezérlőnek, másik viszont ún. direkt virtuális memóriaelérést végeznek (direct virtual memory access, DVMA), logikai címeket használva, és azokat átalakítva fizikai címekké. A DVMA képes adatátvitelre két memóriában mappelt eszköz között a CPU vagy a memória használata nélkül.

Szinkron és aszinkron I/O

Szinkron I/O

A szinkron működésű I/O esetében egy I/O hívás után a hívó program csak akkor kap választ, ha az I/O művelet már lefutott. Ilyenkor a program kénytelen megvárni a teljes művelet idejét, és csak utána folytathatja a futását.

Aszinkron I/O

Az aszinkron működésű I/O esetében egy I/O hívás azonnal ad választ a hívónak, így a program folytathatja a működését, miközben a háttérben lefut az I/O művelet, és egy későbbi időpontban közli a programmal az eredményt. Ez a közlés történhet az alkalmazás egy memóriacímének beállításával, egy szoftveres megszakítás küldésével stb.

**21.Támadások számítástechnikai rendszerek ellen, típusai (mi a támadás tárgya, és módszere). Mutassa be a buffer overflow technikát, és az ellene való védekezés módszereit**

Támadások típusai

A számítógépek elleni támadások lehetnek szándékosak (ártó szándékúak), vagy véletlenek. A véletlenek ellen könnyebb védekezni, mint a kifejezetten rosszindulatú támadások ellen. A legjellemzőbb támadási formák (véletlen és szándékos is):

•Bizalmasság megsértése: ide tartozik, hogy valaki jogosulatlanul olvas el adatokat. Jellemzően ez egy szándékos támadási forma. Bizalmas adatok ellopása, adatfolyamokból kiszűrése közvetlen anyagi hasznot eredményezhet.

•Integritás megsértése: ide tartozik az adatok jogosulatlan módosítása. Ilyen támadás lehet egy program forráskódjának jogosulatlan módosítása.

•Elérhetőség megsértése: ide tartozik adatok jogosulatlan megsemmisítése. Ilyen támadás a weboldalak jogosulatlan átalakítása, deface-elése.

•Szolgáltatás ellopása: ide tartozik erőforrások jogosulatlan használata. Ilyen, amikor egy támadó saját programot futtat valaki más szerverén.

•Szolgáltatás megtagadás: ilyen támadásnál megakadályozzák a törvényes felhasználót, hogy igénybe vegyen egy szolgáltatást. Bizonyos esetekben ez lehet véletlen támadás is.

A támadások során számos technikát szoktak használni. Az egyik ilyen a maszkolás, amikor a támadó fél valaki másnak adja ki magát. Ezzel a támadó az azonosítást játssza ki, lehetetlenné téve a pontos azonosítást. Egy ismétléses támadás során egy rosszindulatú vagy hamis ismétlését küldik el egy valós adatátvitelnek. Ez a támadási forma rendszeresen együtt jár az üzenetmódosítással. Egy újabb típusú támadás a man-in-the-middle támadási forma, amikor a támadó a két kommunikáló fél közé állva hallgatja le és módosítja meg az adatfolyamot.

A védekezéshez négy szinten kell megvalósítani:

1.Fizikai szint: a számítógépeket fizikailag védeni kell a támadók ellen, pl.: zárható szerver szobák stb.

2.Ember: szükséges a felhasználói azonosítás, a felhasználói jogok körültekintő kiosztása.

3.Operációs rendszer: a rendszernek önmagát kell megvédenie a támadásoktól. Akár egyetlen "elszabadult" folyamat is okozhat véletlen támadást.

4.Hálózat: figyelni kell a hálózati forgalmat, és ki kell szűrni belőle a rendszer számára hasznos információkat, a többit pedig el kell dobni.

Buffer overflow technika

Ez az egyik legáltalánosabb technika a távoli betörésre. Alapvetően a támadás alapja egy hiba a programban. Ha egy támadó tudomást szerez egy hibalehetőségről, megír egy programot, amivel a következőket teszi:

1.Túltölt (eláraszt, "túlcsordít") egy beviteli mezőt, parancssori paramétert, input buffert egészen addig, amíg az nem ír a stack-re.

2.Felülírja az aktuális visszatérési címet a stack-en a harmadik pontban betöltött exploit címére

3.Megír egy kis programot a stack következő részére, ami az általa kívánt parancsot futtatja le

Az program futásának eredménye egy root shell vagy más, privilégiumokkal rendelkező parancs futtatása lehet.

A buffer overflow kivédésének lehetőségei:

•Ha a támadó a parancssori paraméterek elárasztásával tud betörni, akkor erre megoldás lehet a program argumentumait tartalmazó argv[] tömb határainak ellenőrzése.

•A stack-en való programfuttatás megakadályozható, ha a CPU-ba beépítik ezt a tiltást (pl.: Sun Sparc rendszerek)

•A mai operációs rendszerek támogatják az NX bit (nonexecutable bit) használatát, ami egy új bit a CPU lapozó táblában. Ez a bit a hozzá tartozó lapot nem futtathatónak jelöli, megtiltva ezzel utasítások olvasását és futtatását. Amint ez a megoldás elterjedt lesz, a buffer overflow támadások nagy mértékben vissza fognak szorulni.

Egy buffer overflow-ra hajlamos program:

```
#include <stdio.h>
#define BUFFER_SIZE 256

int main(int argc, char *argv[]) {
  char buffer[BUFFER_SIZE];

  if (argc<2) {
    return -1;
  else {
    strcpy(buffer,argv[1]);
    return 0;
  }
}
```

**22.Kriptográfiai védelem lehetőségeinek értékelő bemutatása**

Kriptográfiai lehetőségek

Egy hálózati kapcsolatnál fontos, hogy megbízhatóan azonosítható legyen a kommunikációban részt vevő mindkét fél. A kriptográfia módszereivel kikényszeríthető a küldő és/vagy fogadó kilétének azonosítása. A modern kriptográfia alapját a kulcsok képezik, amelyeket bizonyos számítógépek kapnak meg, hogy ezek segítségével fogadhassanak üzeneteket. A kriptográfia segítségével egy fogadó megbizonyosodhat a küldő kilétéről, illetve hogy a küldő tényleg az-e, akinek vallja magát. Megoldható, hogy a küldő úgy kódolja az üzenetét, hogy kizárólag a fogadónál lévő kulccsal lehet azt kikódolni. A kulcsokat olyan algoritmusokkal hozzák létre, hogy azok visszafejtése az adott technológiai fejlettséggel szinte örökké tartana. A kriptográfia napjainkban legfontosabb területei, amik az operációs rendszerekre tartoznak:

Adatvédelmi kódolás (Encryption)

Ez egy általános eljárás, ezért számos helyen alkalmazzák. A kódolás nagymértékben csökkenti a fogadók számát. A kódoló algoritmussal a küldő biztosíthatja, hogy csak egy megfelelő kulccsal rendelkező számítógépen tudják majd elolvasni az üzenetét.

Szimmetrikus titkosítás

Ilyen kódolásnál ugyanaz a kulcs végzi el a be- és a kikódolást is, azaz a kommunikáció mindkét résztvevője ugyanazt a kulcsot használja. Az elmúlt időben a legtöbbet használt ilyen eljárás a DES (dataencryption standard) volt, de mára ez könnyen törhetővé vált. A DES egy blokk kódoló, 64 bites blokkokban kódolja az adatokat. A DES egy módosítása a 3DES, amiben a DES kódolást kell alkalmazni, háromszor egymás után (két bekódoló, és egy dekódoló lépés), két vagy három kulcsot alkalmazva. A legújabb szabványok egyike az AES (advanced encryption standard), ami már egy jóval fejlettebb kódoló, de mégis kicsi és hatékony. Emellett számos más eljárás is létezik, pl. a twofish vagy az RC5. Ez utóbbi elég általános megoldás ahhoz, hogy számos architektúrán futtatható legyen.

Aszimmetrikus titkosítás

Ilyen kódolásnál két külön kulcs van, amik egy kulcspárt alkotnak. Ekkor a bekódolást és a dekódolást másik kulcs végzi el. A legismertebb ilyen algoritmus az RSA, ami egy blokk kódoló algoritmus, és a jelenlegi ismeretek szerint nem törhető fel belátható időn belül. Az algoritmus alapját a nagy számok törzstényezős felbontásának problémája adja.

Az aszimmetrikus titkosítás esetén a kulcspár egyik fele, a public (nyílt) kulcs szabadon terjeszthető, akár a netre is kitehető. Ha valaki kommunikálni akarna ezzel a kulccsal, a fogadó oldalnak mindenképpen szüksége lenne a kulcspár másik felére, a titkos kulcsra, hogy dekódolni tudja az üzenetet. A titkos kulcsot szigorúan őrizni kell, mivel ez biztosítja a kulcspár alapú azonosítás alapját.

Autentikáció

Ha egy üzenet esetében megszabjuk a potenciális küldők körét, akkor azonosítjuk az üzenetet. Az azonosítás kiegészíti a titkosítást, pl.: egy titkosított üzenetnél azonosítással biztosítható, hogy átvitel közben az üzenet nem lett módosítva.

Az azonosító algoritmusoknak két nagy csoportját különböztetjük meg, de ehhez szükség van a hash eljárásokra. Egy hash eljárás egy kicsit, fix méretű adatot készít az adott üzenetből, amit hash értéknek vagy üzenet kivonatnak neveznek. Az algoritmusok determinánsak, ugyanarra a bemenetre mindig ugyanazt a kimenetet adják. Ma népszerű algoritmus az MD5 és az SHA-1. Az algoritmusok két féle csoportja:

•Szimmetrikus titkosítást használó eljárások. Egy ún message-authentication code (MAC)-ban egy ellenőrző érték készül el az üzenetből egy titkos kulcs használatával.

•Aszimmetriksu titkosítást használó eljárások, ezek a digitális alárírást készítő algoritmusok. Itt jellemzően az RSA-t szokták felhasználni.

Felhasználói azonosítás

Ez az operációs rendszerek nagy problémája, hogy milyen módon lehetséges a felhasználókat megbízhatóan azonosítani.

Jelszavak

A legegyszerűbb védelem. Számos helyen lehet alkalmazni, akár teljes rendszerek védelménél, de akár file szinten is, az adott operációs rendszertől függ az alkalmazása. Általában egy rendszer egyszer, belépéskor kér egy jelszót egy felhasználótól.

A jelszavak nagy problémája, hogy ha gyenge a jelszó, akkor könnyen kitalálható. Ilyen hiba lehet, hogy a felhasználó önmagára nézve túl egyértelmű jelszót ad meg (pl.: macskája neve), vagy rövid a jelszó és próbálgatással is gyorsan rá lehet jönni. A jelszavak kifigyelhetőek, mikor a felhasználü beírja őket, illetve hálózaton keresztül a titkosítatlan jelszó lehallgatható (pl.: ftp esetén).

A rendszer szempontjából a jelszavak tárolása okoz problémát, hogy biztonságosan, de mégis könnyen hozzáférhető módon tárolja el a jelszavakat. Általában valamilyen titkosítással szokás elmenteni egy file-ba a jelszavakat.